多线程

1. synchronized关键字

1.0 synchronized锁定的是对象，不是代码块

1.1 锁的是堆内存对象

1.2 synchronized可以修饰代码块，代码块执行完锁释放

```
public class T {

    private int count = 0;
    private final Object lock = new Object();

    public void m() {
        synchronized (lock) {
            // 任何线程要执行下面的代码，都必须先拿到lock锁，
            // 锁信息记录在堆内存对象中的，不是在栈引用中
            // 如果lock已经被锁定，其他线程再进入时，就会进行阻塞等待
            // 所以 synchronized 是互斥锁
            count--;
            System.out.println(Thread.currentThread().getName() + " count = " + count);
        }
        // 当代码块执行完毕后，锁就会被释放，然后被其他线程获取
    }
}
```

1.3 synchronized可以锁定this对象，这样的话，一定要得到某个类实例的this对象，才能执行其中的代码

```
public class T {

    private int count = 10;

    public void m() {
        synchronized (this) { // 任何线程要执行下面的代码，必须先拿到this锁
            // synchronized 锁定的不是代码块，而是 this 对象
            count--;
            System.out.println(Thread.currentThread().getName() + " count = " + count);
        }
    }
}
```

1.4 synchronized可以修饰方法，此时默认锁定this对象

```java
public class T {

    private int count = 10;

    public synchronized void m() { // 等同于 synchronized (this) {
        count--;
        System.out.println(Thread.currentThread().getName() + " count = " + count);
    }

}
```

1.5 synchronized修饰静态方法或静态方法中的代码块，则锁定的是类对象

```java
public class T {

    private static int count = 10;

    public static synchronized void m() { // 等同于 synchronized (c_004.T.class) {
        count--;
        System.out.println(Thread.currentThread().getName() + " count = " + count);
    }

}
```

1.6 同步方法和非同步方法可以同时调用

```java
public class T {

    public synchronized void m1() {
        System.out.println(Thread.currentThread().getName() + " m1 start");
        try {
            TimeUnit.SECONDS.sleep( timeout: 10); //休眠10秒
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " m1 end");
    }

    public void m2() {
        System.out.println(Thread.currentThread().getName() + " m2 start");
        try {
            TimeUnit.SECONDS.sleep( timeout: 5);//休眠5秒
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " m2 end");
    }

    public static void main(String[] args) {
        T t = new T();
        new Thread(t::m1).start();
        new Thread(t::m2).start();
    }
}
```

m2能在m1线程执行过程中能够运行

```
D:\javaJdk_18\bin\java.exe ...
Thread-0 m1 start
Thread-1 m2 start
Thread-1 m2 end
Thread-0 m1 end
```

1.7 业务代码写代码需要加锁，最好读代码也要加锁，否则可能会脏读

1.8 synchronized 是可重入锁，也就是一个同步方法可以调用另一个同步方法

```java
/**
 * synchronized 是可重入锁
 * 即一个同步方法可以调用另外一个同步方法，一个线程已经拥有某个对象的锁，再次申请时仍然会得到该对象的锁
 */
public class T {

    synchronized void m1() {
        System.out.println("m1 start ");
        try {
            TimeUnit.SECONDS.sleep( timeout: 1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        m2();
    }

    synchronized void m2() {
        try {
            TimeUnit.SECONDS.sleep( timeout: 2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(" m2"); // 这句话会打印，调用m2时，不会发生死锁
    }
}
```

毕竟都是同一把锁

1.9 子类的同步方法可以调用父类的同步方法

```java
public class T {

    synchronized void m() {
        System.out.println("m start ");
        try {
            TimeUnit.SECONDS.sleep( timeout: 1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("m end ");
    }

    public static void main(String[] args) {
        TT tt = new TT();
        tt.m();
    }
}

class TT extends T {
    @Override
    synchronized void m() {
        System.out.println(" child m start ");
        super.m();
        System.out.println(" child m end ");
    }
}
```

因为锁定的是同一个对象

1.10 同步方法或代码块中异常抛出不处理会释放锁，所以最好加上异常处理

```java
public class T {

    int count = 0;

    synchronized void m() {
        System.out.println(Thread.currentThread().getName() + " start");
        while (true) {
            count++;
            System.out.println(Thread.currentThread().getName() + " count=" + count);
            try {
                TimeUnit.SECONDS.sleep( timeout: 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            if (count == 5) {  // 当count == 5 时，synch
                int i = 1 / 0;
            }
        }
    }
```

```java
    public static void main(String[] args) {
        T t = new T();
        Runnable r = new Runnable() {
            @Override
            public void run() { t.m(); }
        };
        new Thread(r,  name: "t1").start(); // 执行到第5秒时，抛出 ArithmeticException
        // 如果抛出异常后，t2 会继续执行，就代表t2拿到了锁，即t1在抛出异常后释放了锁

        try {
            TimeUnit.SECONDS.sleep( timeout: 3); //不释放锁，本例的t2线程不会执行到
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        new Thread(r,  name: "t2").start();
    }

}
```

1.11 锁对象的属性改变不会影响锁的使用，但是锁的对象改变会影响

```java
/**
 * 锁定某个对象o，如果o属性发生变化，不影响锁的使用
 * 但是如果o变成另一个对象，则锁定的对象发生变化，
 * 所以锁对象通常要设置为 final类型，保证引用不可以变
 */
public class T {

    Object o = new Object();

    void m() {
        synchronized (o) {
            while (true) {
                System.out.println(Thread.currentThread().getName());
                try {
                    TimeUnit.SECONDS.sleep( timeout: 1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

```java
    public static void main(String[] args) {
        T t = new T();
        new Thread(t::m,  name: "线程1").start();

        try {
            TimeUnit.SECONDS.sleep( timeout: 3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Thread thread2 = new Thread(t::m,  name: "线程2");
        t.o = new Object(); // 改变锁引用，线程2也有机会运行，否则一直都是线程1 运行
        thread2.start();
    }

}
```

1.12 不要把字符串常量对象作为锁对象

```java
/**
 * 不要以字符串常量作为锁定对象
 * 在下面的例子中，m1和m2其实是锁定的同一对象
 * 这种情况下，还会可能与其他类库发生死锁，比如某类库中也锁定了字符串 "Hello"
 * 但是无法确认源码的具体位置，所以两个 "Hello" 将会造成死锁
 * 因为你的程序和你用的类库无意间使用了同意把锁
 */
public class T {

    String s1 = "Hello";
    String s2 = "Hello";

    void m1() {
        synchronized (s1) {

        }
    }

    void m2() {
        synchronized (s2) {

        }
    }
}
```

2. volatile关键字

2.1 volatile可以实现无锁同步，但能力有限

```java
/**
 * volatile 关键字，使一个变量在多个线程间可见
 * cn: 透明的，临时的
 *
 * JMM(Java Memory Model):
 * 在JMM中，所有对象以及信息都存放在主内存中（包含堆，栈）
 * 而每个线程都有自己的独立空间，存储了线程使用到的变量的副本
 * 线程对共享变量的操作，都会在自己的工作内存中进行，然后同步给主内存
 *
 * 下面的代码中，running 是位于堆内存中的t对象的
 * 当线程t1开始运行的时候，会把running值从内存中读到t1线程的工作区，在运行过程中直接使用这个copy，并不会每次都会去读取堆内存，
 * 这样，当主线程修改running的值之后，t1线程感知不到，所以不会停止运行
 *
 * 使用volatile，将会强制所有线程都去堆内存中读取running的值
 *
 *
 */
public class T {

    /*volatile*/ boolean running = true;    // 对比有无volatile的情况下，整个程序运行结果的区别

    void m() {
        System.out.println(" m start ");
        while (running) { // 直到主线程将running设置为false，T线程才会退出

        }
        System.out.println(" m end ");
    }

    public static void main(String[] args) {
        T t = new T();
        new Thread(t::m, name: "t1").start();
        try {
            TimeUnit.SECONDS.sleep( timeout: 1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        t.running = false;
    }

}
```

不加volatile将不会停掉t1线程

2.2 volatile可以保证可见性，但并不保证原子性，synchronized

2.3 Actomic类可以解决volatile的不保证原子性的问题，但是不保证连续调用该类的方法的原子性，此时不需要加synchronized

```java
/**
 * volatile 关键字，使一个变量在多个线程间可见
 * volatile并不能保证多个线程共同修改running变量所带来的不一致的问题，也就是说volatile不能替代synchronized
 * 即 volatile只能保证可见性，不能保证原子性
 */
public class T {

    volatile int count = 0;
    /*AtomicInteger count = new AtomicInteger(0);*/

    /*synchronized*/ void m() {
        for (int i = 0; i < 10000; i++) {
            count++;
            /*count.incrementAndGet();*/
        }
    }
```

//也可以加AtomicXXX等类解决，并使用该类的方法完成写操作，
这种类的所有方法都是原子性的， 但不能保证连续调用也是原子性的

//当然，synchronized是首先想到的解决方法

//我可以保证count对于每个线程都可见，但我不能保证线程的执行顺序，CPU的调度不是我能解决的，所以线程之间改变count的操作从何开始我并不知道

```java
    public static void main(String[] args) {
        // 创建一个10个线程的list，执行任务皆是 m方法
        T t = new T();
        List<Thread> threads = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            threads.add(new Thread(t::m, name: "t-" + i));
        }

        // 启动这10个线程
        threads.forEach(Thread::start);

        // join 到主线程，防止主线程先行结束
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // 10个线程，每个线程执行10000次，结果应为 100000
        System.out.println(t.count);  // 所得结果并不为 100000，说明volatile 不保证原子性
    }

}
```

2.4 细粒度的锁要比高粒度的锁性能高很多

```java
synchronized void m1() {
    try {
        TimeUnit.SECONDS.sleep( timeout: 2);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // 业务逻辑中，只有下面这句代码需要 sync， 这时不应该给整个方法上锁
    count++;

    try {
        TimeUnit.SECONDS.sleep( timeout: 2);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```java
    void m2() {
        try {
            TimeUnit.SECONDS.sleep( timeout: 2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 业务逻辑中，只有下面这句需要 sync，这时不应该给整个方法上锁
        // 采用细粒度的锁，可以使线程争用时间变短，从而提高效率
        synchronized (this) {
            count++;
        }

        try {
            TimeUnit.SECONDS.sleep( timeout: 2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

3. 淘宝面试题：实现一个容器，提供两个方法，add，size 写两个线程，线程1添加10个元素到容器中，线程2实现监控元素的个数，当个数到达5时，线程2给出提示并结束

3.1 错误示例：连可见性都没保证

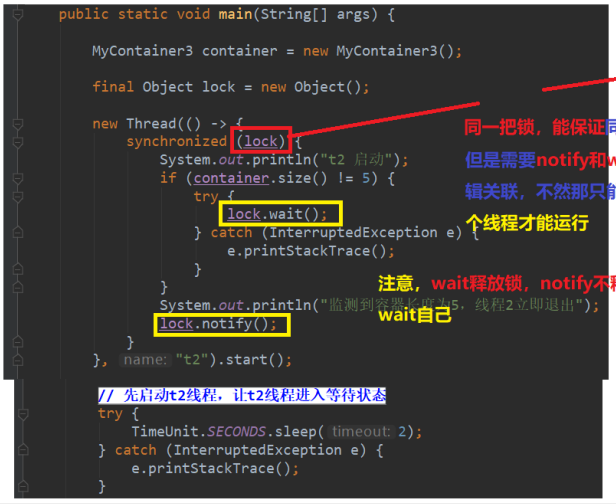
```java
/**
 * 面试题（淘宝？）
 * 实现一个容器，提供两个方法，add，size
 * 写两个线程，线程1添加10个元素到容器中，线程2实现监控元素的个数，当个数到达5时，线程2给出提示并结束
 */
public class MyContainer1 {

    private List<Object> list = new ArrayList<>();

    public void add(Object ele) { list.add(ele); }

    public int size() { return list.size(); }
```

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

        MyContainer1 container = new MyContainer1();

        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                container.add(new Object());
                try {
                    TimeUnit.SECONDS.sleep( timeout: 1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("add " + i);
            }
        }, name: "t1").start();

        new Thread(() -> {
            while (true) {
                if (container.size() == 5) {
                    break;
                }
            }
            System.out.println("监测到容器长度为5，线程2立即退出");
        }, name: "t2").start();

    }
}
```

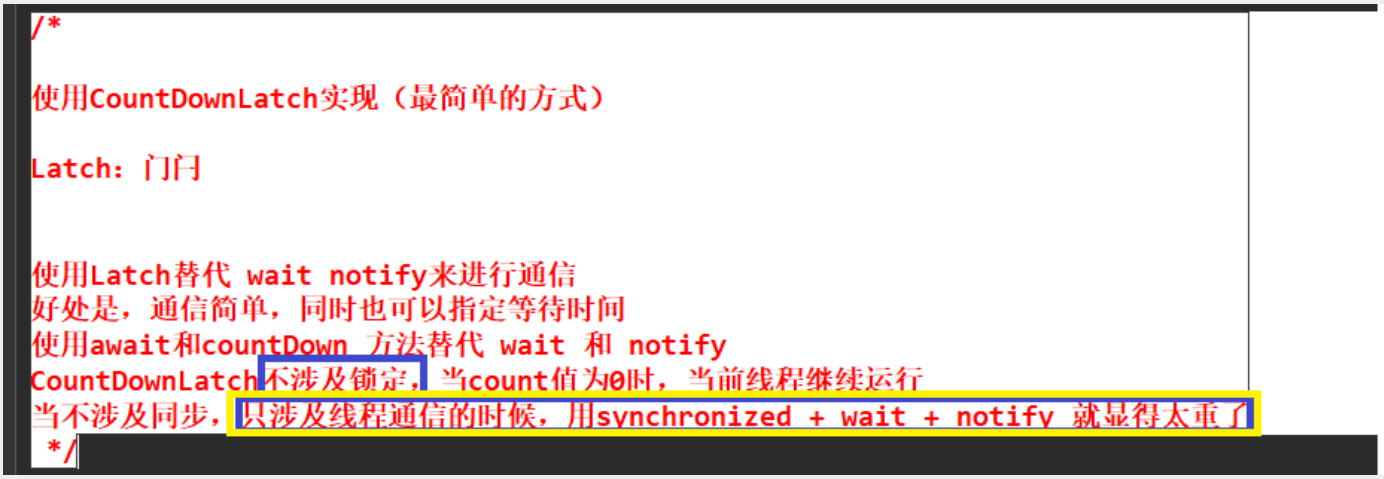
3.2 勉强合格：对容器对象加volatile，但是还是有问题

```
/*

添加 volatile ，使list发生变化时，主动通知其他线程，更新工作空间

上述代码，共有以下几个问题：
1. 不够精确，当container.size == 5 还未执行break时，有可能被其他线程抢占；或者 container.add() 之后，还未打印，就被 t2 判断size为5直接退出了
2. 损耗性能，t2 线程，一直在走while循环，很浪费性能

 */
```

3.3 不加volatile，加锁,应用notify和wait方法，注意wait释放锁，notify不释放锁



3.4 用线程间通信的方法：CountDownLatch



4. 替代synchronized： ReentrantLock手动锁

4.1 ReentrantLock不会自动释放锁，需要在手动释放锁

```java
/**
 * ReentrantLock替代synchronized
 */
public class ReentrantLock2 {

    ReentrantLock lock = new ReentrantLock();

    void m1() {
        lock.lock(); // 相当于 synchronized
        try {
            for (int i = 0; i < 10; i++) {
                try {
                    TimeUnit.SECONDS.sleep( timeout: 1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(i);
            }
        } finally {
            lock.unlock(); // 使用完毕后，必须手动释放锁
            // 不同于synchronized，抛出异常后，不会自动释放锁，需要我们在finally中释放此锁
        }
    }

    void m2() {
        lock.lock(); // 相当于 synchronized
        try {
            System.out.println("m2...");
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        ReentrantLock2 r1 = new ReentrantLock2();
        new Thread(r1::m1, name: "t1").start(); // m1 已经执行，被t1占有锁this
        try {
            TimeUnit.SECONDS.sleep( timeout: 1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        new Thread(r1::m2, name: "t2").start(); // 锁已经被其他线程占用，m1执行完毕后执行
    }
}
```

4.2 ReentrantLock可以用尝试去拿锁的方法

```java
boolean locked=false;
try {
    locked=lock.tryLock( timeout: 10, TimeUnit.SECONDS);

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

}finally {
    if(locked){
        System.out.println("m2 start ...");
        for (int i = 10; i >0 ; i--) {
            System.out.println(i);
            try {
                TimeUnit.SECONDS.sleep( timeout: 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        lock.unlock();
    }
    else{
    System.out.println("m2没拿到锁");
    }
}
```

这是说尝试获取10秒，10秒内看是否能不能够拿到锁，

10秒

```java
/*   // 尝试获取锁，返回true拿到了
    if (lock.tryLock()) {  立即去拿锁
        // lock.tryLock(5, TimeUnit.SECONDS) // 等5s内还没拿到就返回false
        System.out.println("m2...");
        lock.unlock();
    } else {
        System.out.println(" m2 没拿到锁");
    }*/
```

4.3 lockInterruptibly这个方法可以响应打断本线程的等待

```
Thread t2 = new Thread(() -> {

    try {
        //t2.tryLock();//用这种方法会一直等下去，就不可以被锁就来到中断
        lock.lockInterruptibly(); // t2 尝试获取锁 没拿到锁不会执行try中的下面代码块，
                                  // 被打断就会执行catch中的代码

        System.out.println("t2 start");
        lock.unlock();
    } catch (InterruptedException e) {
        System.out.println("t2 等待中被打断");
    }
}, name: "t2");
t2.start();

try {
    TimeUnit.SECONDS.sleep( timeout: 10);
} catch (InterruptedException e) {
    e.printStackTrace();
}                                          //等待十秒

// 打断线程2的等待
t2.interrupt();                            如果另一个线程独占了锁，线程2在这里等待是浪费资源的，最好被打断
```

4.4 ReentrantLock可以指定为公平锁，synchronized不可以

```
/**
 * ReentrantLock 可以指定为公平锁，synchronized 是不公平锁
 * 公平锁，先获取锁的人，在锁被释放时，优先获得锁
 * 不公平锁，无论先后，线程调度器将会随机给某个线程锁，不用计算线程时序，效率较高
 */
public class ReentrantLock5 extends Thread {

    private static ReentrantLock lock = new ReentrantLock( fair: true);// 指定锁为公平锁

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            lock.lock();
            try {
                System.out.println(Thread.currentThread().getName() + "获取锁");
            } finally {
                lock.unlock(); // 公平锁 t1 unlock 后，等待时间长的一定是 t2 所以下次一定是 t2 执行
            }
        }
    }

    public static void main(String[] args) {
        ReentrantLock5 t1 = new ReentrantLock5();
        ReentrantLock5 t2 = new ReentrantLock5();
        t1.start();
        t2.start();
    }
}
```

```
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
Thread-1获取锁
Thread-0获取锁
```

5. 面试题：生产者消费者模式实现

```java
public class MyContainer1<T> {

    private final LinkedList<T> list = new LinkedList<>();
    private final int MAX = 10;
    private int count = 0;

    public synchronized void put(T t) {
        while (MAX == count) { // 如果容量最大，释放锁等待      ///【这里为什么使用while，而不是使用if？？】
            try {                  当前线程判断线程池满了，就会wait，并释放锁，然后生产者的其他线程得到锁之后会继续判断并wait，释放锁；很可能多个生产者线程等在这里
                this.wait();       如果消费者线程唤醒了生产者线程，不用while的话将会直接执行添加操作，这样不安全
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 否则 put
        list.add(t);
        ++count;
        this.notifyAll(); // 通知消费者线程，可以消费了
        // 【这里为什么调用 notifyAll 而不是 notify ？】
    }
```

这种方法不能保证消费者线程wait时精确叫醒生产者线程，相反同理

虽然，消费者线程停了，生产者线程也只能一个进行，但他们可以竞争上岗呀

```java
    public synchronized T get() {                    得到锁之后
        while (list.size() == 0) { // 如果容量为空，释放锁等待
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // 否则获取
        T t = list.removeFirst();
        count--;
        this.notifyAll(); // 通知生产者线程生产
        return t;
    }
}
```

：保证叫醒消费者线程，注意：消费者线程和生产者线程用的是同一把锁

同理，这里用notifyAll()也是为了保证叫醒生产者线程

相反同理

```java
public class MyContainer2<T> {

    private final LinkedList<T> list = new LinkedList<>();
    private final int MAX = 10;
    private int count = 0;

    private Lock lock = new ReentrantLock();
    private Condition producer = lock.newCondition();
    private Condition consumer = lock.newCondition();

    public synchronized void put(T t) {
        lock.lock();
        try {
            while (MAX == count) {
                producer.await();
            }
            list.add(t);
            ++count;
            System.out.println("生产了第"+count+"个包子");
            consumer.signalAll();  //精确叫醒消费者
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
```

```java
    public synchronized T get() {
        lock.lock();
        try {
            while (list.size() == 0) {
                consumer.await();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
        T t = list.removeFirst();
        count--;
        System.out.println("消费了第"+count+"个包子");
        producer.signalAll();
        return t;
    }
```

6. 线程局部变量ThreadLocal

```java
public class ThreadLocal2 {

    static ThreadLocal<Person> p = new ThreadLocal<>();

    public static void main(String[] args) {
        new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep( timeout: 2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(p.get()); // 2. 虽然threadLocal时共享变量，但是取不到其他线程放入的值，所以此处为null
        }).start();

        new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep( timeout: 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            p.set(new Person()); // 1. 往线程局部变量放入一个person
        }).start();
    }

    static class Person {
        String name = "zhangsan";
    }
}
```