

多线程相关面试题

线程包括哪些状态，状态之间是如何变化的

在java中wait和sleep方法的不同

线程池中有哪些常见的阻塞队列

线程与进程的区别

如何停止一个正在运行的线程

谈谈你对ThreadLocal的理解

并行与并发的区别

ReentrantLock的实现原理

新建 T1、T2、T3 三个线程，如何保证它们按顺序执行

线程的 run()和 start()有什么区别

notify()和 notifyAll()有什么区别

runnable 和 callable 有什么区别

线程创建的方式有哪些

高频

难以回答

什么是AQS

为什么不建议用Executors创建线程池

如何控制某个方法允许并发访问线程的数量

死锁产生的条件是什么

如何确定核心线程数

CAS 你知道吗

synchronized和Lock有什么区别

synchronized关键字的底层原理

你谈谈 JMM (Java 内存模型)

说一下线程池的核心参数 (线程池的执行原理知道嘛)

线程池的种类有哪些

如何进行死锁诊断

导致并发程序出现问题的根本原因是什么

线程池使用场景(你们项目中哪里用到了线程池)

请谈谈你对 volatile 的理解

聊一下ConcurrentHashMap

线程的基础知识

线程与进程的区别

并行与并发的区别

线程创建的方式有哪些

Runnable 和 Callable 有什么区别

线程包括哪些状态，状态之间是如何变化的

在Java中wait和sleep方法的不同

新建三个线程，如何保证它们按顺序执行

notify()和 notifyAll()有什么区别

线程的 run()和 start()有什么区别

如何停止一个正在运行的线程

线程中并发安全

synchronized关键字的底层原理

你谈谈 JMM (Java 内存模型)

CAS 你知道吗

什么是AQS

ReentrantLock的实现原理

synchronized和Lock有什么区别

死锁产生的条件是什么

如何进行死锁诊断

请谈谈你对 volatile 的理解

聊一下ConcurrentHashMap

导致并发程序出现问题的根本原因是什么

线程池

说一下线程池的核心参数 (线程池的执行原理知道嘛)

线程池中有哪些常见的阻塞队列

如何确定核心线程数

线程池的种类有哪些

为什么不建议用Executors创建线程池

使用场景

线程池使用场景(你们项目中哪里用到了线程池)

如何控制某个方法允许并发访问线程的数量

谈谈你对ThreadLocal的理解

线程的基础知识

线程与进程的区别

并行与并发的区别

线程创建的方式有哪些

Runnable 和 Callable 有什么区别

线程包括哪些状态，状态之间是如何变化的

在Java中wait和sleep方法的不同

新建三个线程，如何保证它们按顺序执行

notify()和 notifyAll()有什么区别

线程的 run()和 start()有什么区别

如何停止一个正在运行的线程

线程和进程的区别？

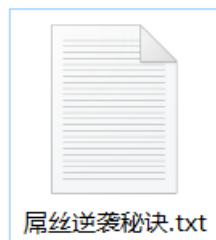
难易程度： ★★☆☆☆

出现频率： ★★★☆☆

线程和进程的区别？

程序由**指令**和**数据**组成，但这些指令要运行，数据要读写，就必须将指令加载至 CPU，数据加载至内存。在指令运行过程中还需要用到磁盘、网络等设备。进程就是用来加载指令、管理内存、管理 IO 的。

当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程。



多个实例进程

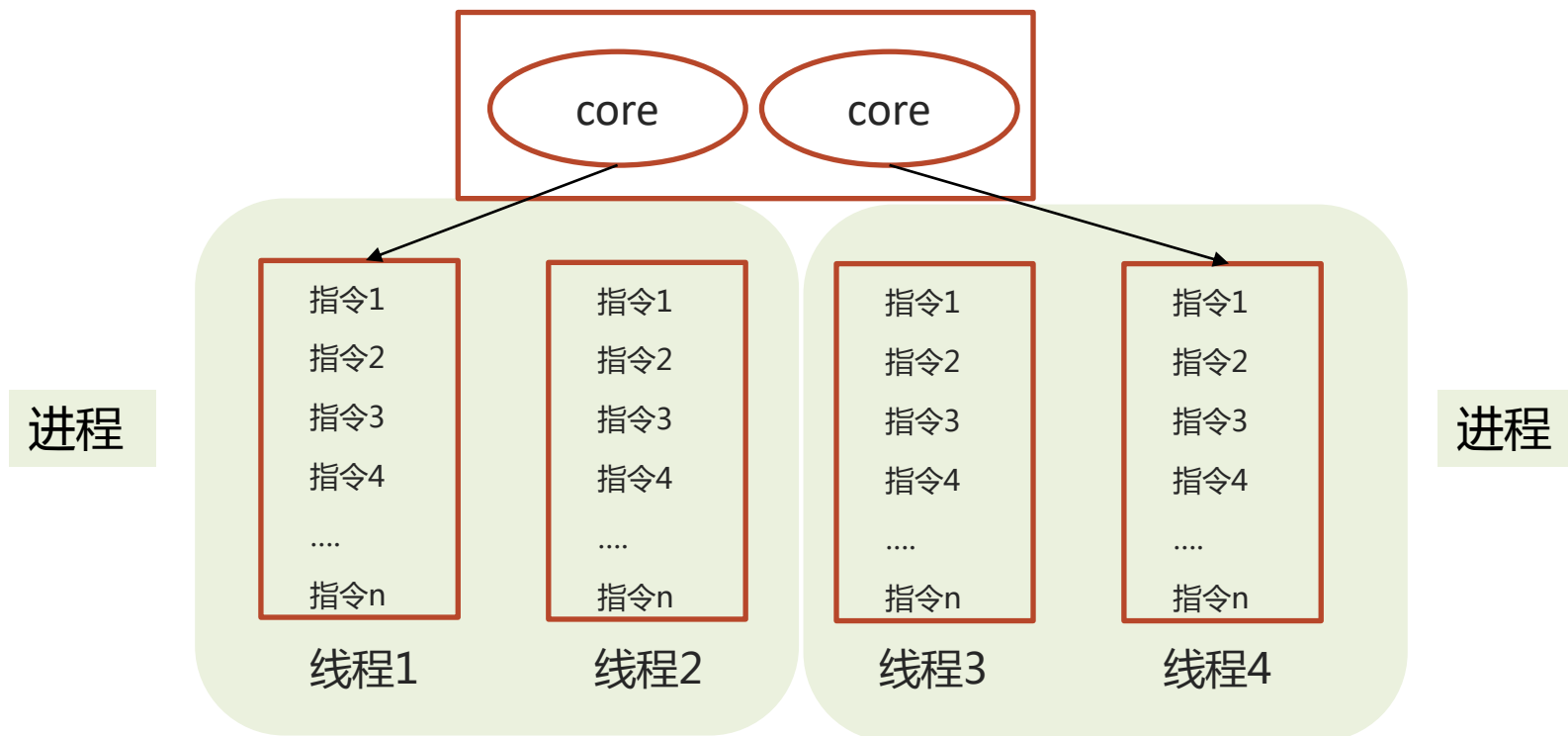
单实例进程



线程和进程的区别？

一个线程就是一个指令流，将指令流中的一条条指令以一定的顺序交给 CPU 执行

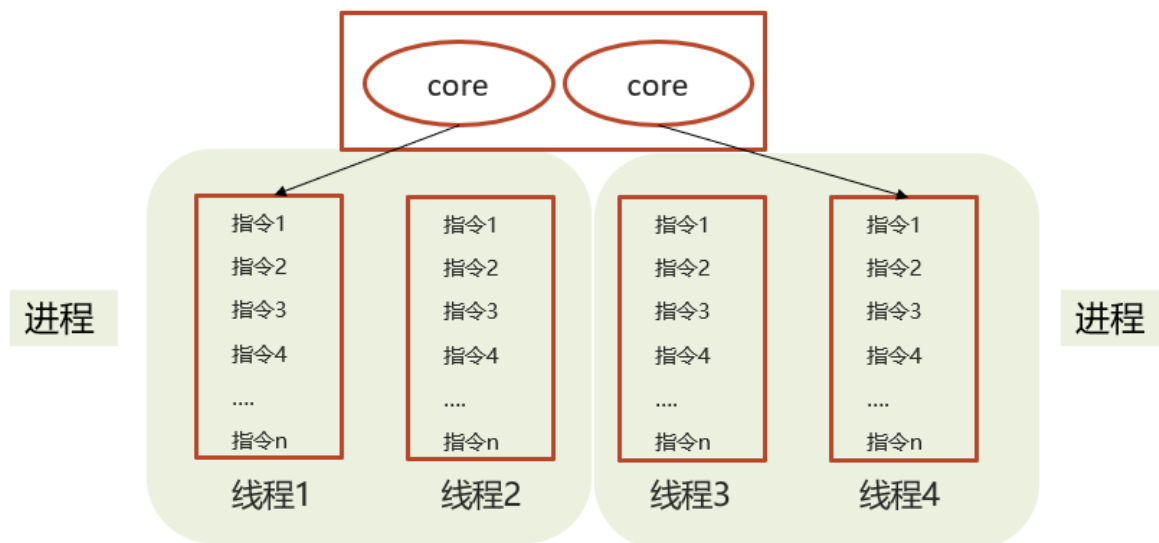
一个进程之内可以分为一到多个线程。



线程和进程的区别？

二者对比

- 进程是正在运行程序的实例，进程中包含了线程，每个线程执行不同的任务
- 不同的进程使用不同的内存空间，在当前进程下的所有线程可以共享内存空间
- 线程更轻量，线程上下文切换成本一般上要比进程上下文切换低(上下文切换指的是从一个线程切换到另一个线程)



并行和并发有什么区别？

难易程度： ★★☆☆☆

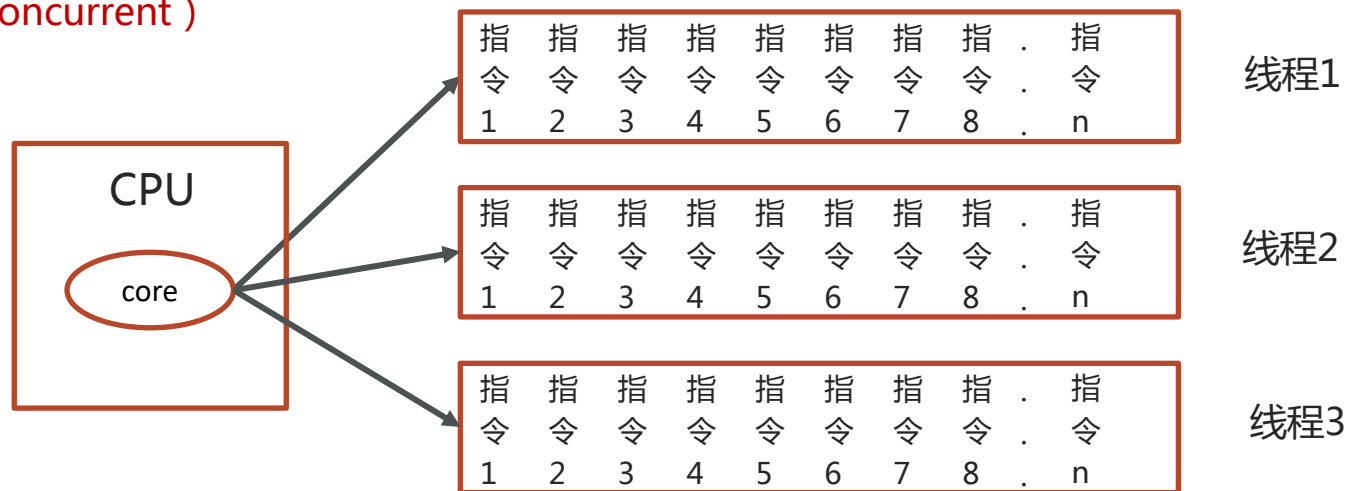
出现频率： ★★☆☆☆

并行和并发有什么区别？

单核CPU

- 单核CPU下线程实际还是串行执行的
- 操作系统中有一个组件叫做任务调度器，将cpu的时间片（windows下时间片最小约为 15 毫秒）分给不同的程序使用，只是由于cpu在线程间（时间片很短）的切换非常快，人类感觉是同时运行的。
- 总结为一句话就是：微观串行，宏观并行
- 一般会将这种线程轮流使用CPU的做法称为并发（concurrent）

CPU	时间片1	时间片2	时间片3
core	线程1	线程2	线程3

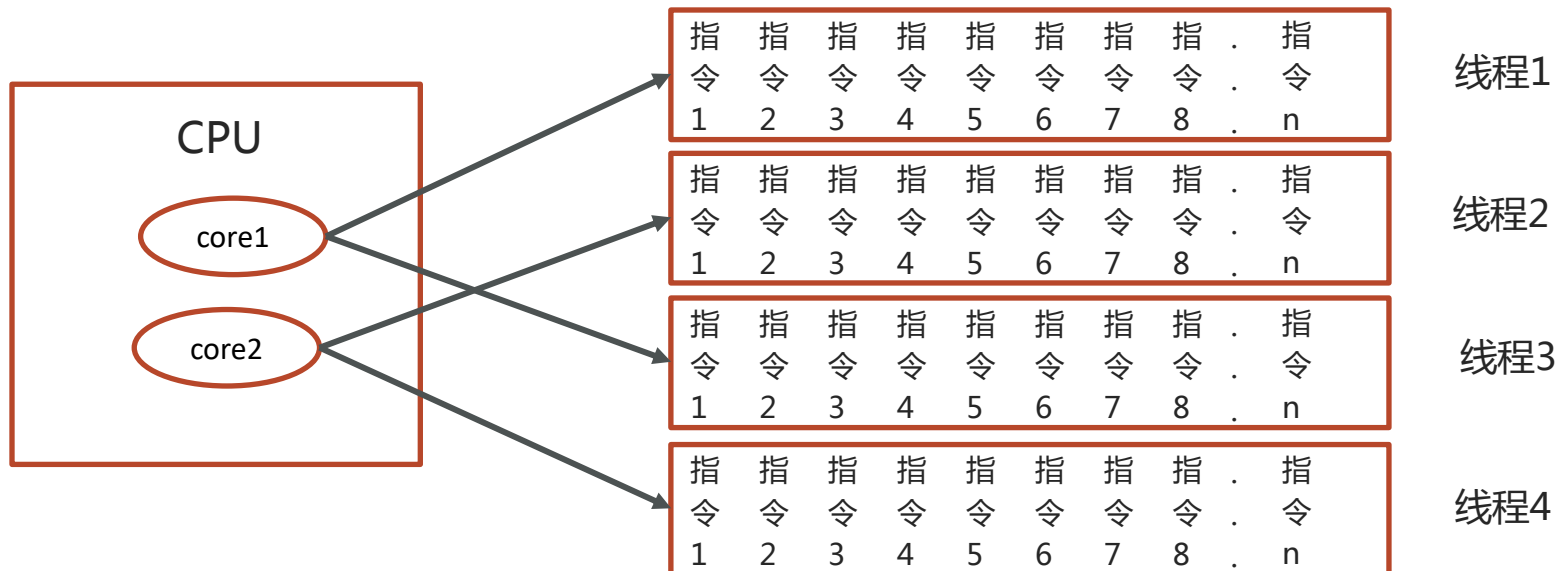


并行和并发有什么区别？

多核CPU

每个核（core）都可以调度运行线程，这时候线程可以是并行的。

CPU	时间片1	时间片2	时间片3	时间片4
core1	线程1	线程1	线程3	线程3
core2	线程2	线程4	线程2	线程4



并行和并发有什么区别？

并发 (concurrent) 是同一时间应对 (dealing with) 多件事情的能力

并行 (parallel) 是同一时间动手做 (doing) 多件事情的能力

- 家庭主妇做饭、打扫卫生、给孩子喂奶，她一个人轮流交替做这多件事，这时就是并发
- 家庭主妇雇了个保姆，她们一起这些事，这时既有并发，也有并行（这时会产生竞争，例如锅只有一口，一个人用锅时，另一个人就得等待）
- 雇了3个保姆，一个专做饭、一个专打扫卫生、一个专喂奶，互不干扰，这时是并行



总结

并行和并发有什么区别

现在都是多核CPU，在多核CPU下

- 并发是同一时间应对多件事情的能力，多个线程轮流使用一个或多个CPU
- 并行是同一时间动手做多件事情的能力，4核CPU同时执行4个线程

创建线程的方式有哪些？

难易程度： ★★☆☆☆

出现频率： ★★★★★

创建线程的方式有哪些？

共有四种方式可以创建线程，分别是：

- 继承Thread类
- 实现Runnable接口
- 实现Callable接口
- 线程池创建线程

继承Thread类

```
public class MyThread extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("MyThread...run...");  
    }  
  
    public static void main(String[] args) {  
  
        // 创建MyThread对象  
        MyThread t1 = new MyThread();  
        MyThread t2 = new MyThread();  
  
        // 调用start方法启动线程  
        t1.start();  
        t2.start();  
  
    }  
}
```


实现Runnable接口

```
public class MyRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("MyRunnable...run...");  
    }  
  
    public static void main(String[] args) {  
  
        // 创建MyRunnable对象  
        MyRunnable mr = new MyRunnable();  
  
        // 创建Thread对象  
        Thread t1 = new Thread(mr);  
        Thread t2 = new Thread(mr);  
  
        // 调用start方法启动线程  
        t1.start();  
        t2.start();  
  
    }  
}
```

实现Callable接口

```
public class MyCallable implements Callable<String> {  
  
    @Override  
    public String call() throws Exception {  
        System.out.println(Thread.currentThread().getName());  
        return "ok";  
    }  
  
    public static void main(String[] args) throws ExecutionException, InterruptedException {  
        // 创建MyCallable对象  
        MyCallable mc = new MyCallable();  
        // 创建FutureTask  
        FutureTask<String> ft = new FutureTask<String>(mc);  
        // 创建Thread对象  
        Thread t1 = new Thread(ft);  
        Thread t2 = new Thread(ft);  
        // 调用start方法启动线程  
        t1.start();  
        // 调用ft的get方法获取执行结果  
        String result = ft.get();  
        // 输出  
        System.out.println(result);  
    }  
}
```

线程池创建线程

```
public class MyExecutors implements Runnable{

    @Override
    public void run() {
        System.out.println("MyRunnable...run...");
    }

    public static void main(String[] args) {

        // 创建线程池对象
        ExecutorService threadPool = Executors.newFixedThreadPool(3);
        threadPool.submit(new MyExecutors());

        // 关闭线程池
        threadPool.shutdown();

    }
}
```

总结

创建线程的方式有哪些？

- 继承Thread类
- 实现Runnable接口
- 实现Callable接口
- 线程池创建线程(项目中使用方式)

面试官追问：刚才你说过，使用Runnable和Callable都可以创建线程，它们有什么区别呢？

runnable 和 callable 有什么区别？

参考回答：

1. Runnable 接口run方法没有返回值
2. Callable接口call方法有返回值，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果
3. Callable接口的call()方法允许抛出异常；而Runnable接口的run()方法的异常只能在内部消化，不能继续上抛

面试官再追问：在启动线程的时候，可以使用run方法吗？run()和 start()有什么区别？

线程的 run()和 start()有什么区别？

start(): 用来启动线程，通过该线程调用run方法执行run方法中所定义的逻辑代码。start方法只能被调用一次。

run(): 封装了要被线程执行的代码，可以被调用多次。



总结

1. 创建线程的方式有哪些？

- 继承Thread类
- 实现Runnable接口
- 实现Callable接口
- 线程池创建线程(项目中使用方式)

2. Runnable 和 Callable 有什么区别

- Runnable 接口run方法没有返回值
- Callable接口call方法有返回值，需要FutureTask获取结果
- Callable接口的call()方法允许抛出异常；而Runnable接口的run()方法的异常只能在内部消化，不能继续上抛

3. run()和 start()有什么区别？

- start(): 用来启动线程，通过该线程调用run方法执行run方法中所定义的逻辑代码。start方法只能被调用一次。
- run(): 封装了要被线程执行的代码，可以被调用多次。

线程包括哪些状态，状态之间是如何变化的

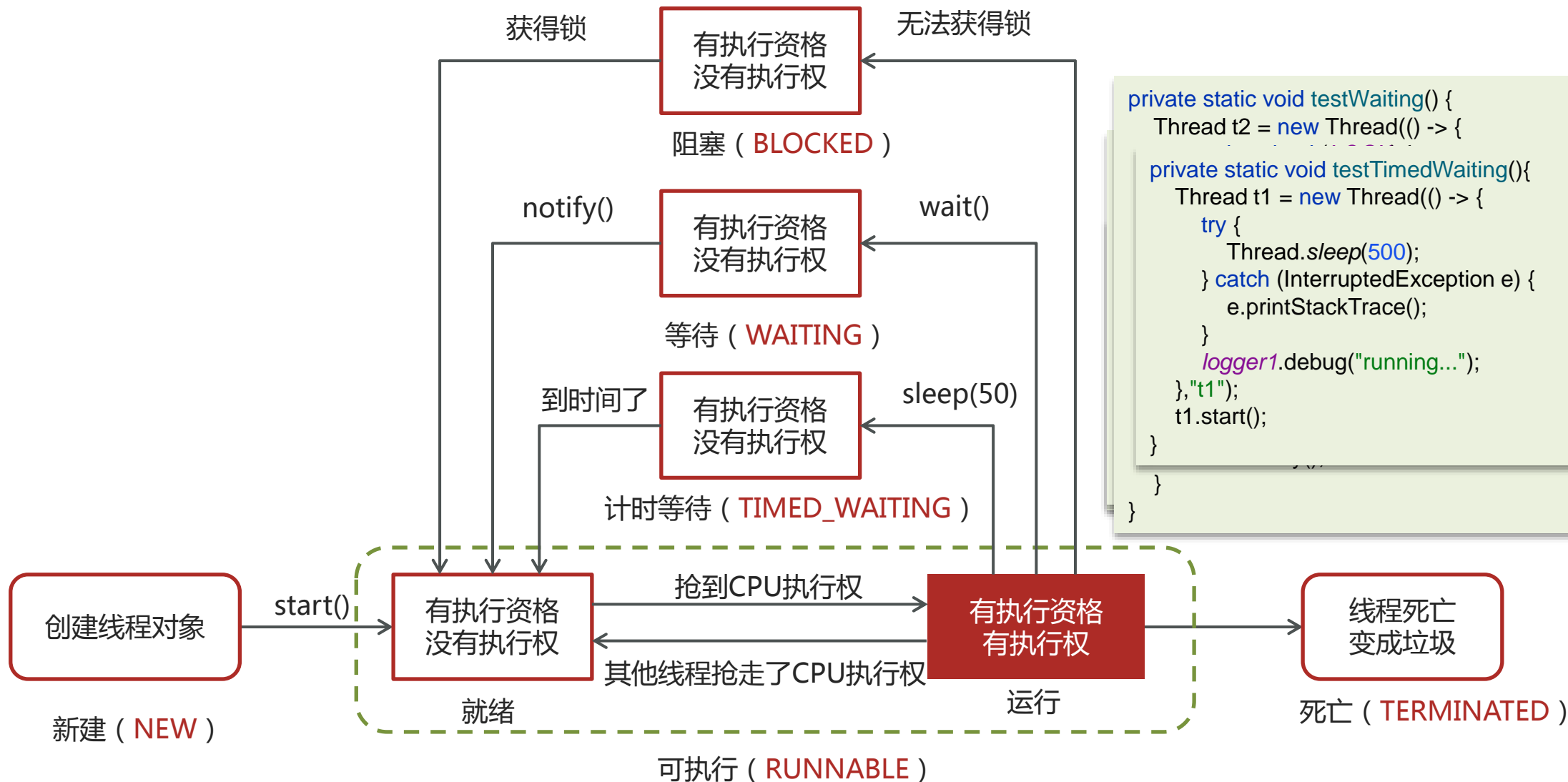
难易程度： ★★★★★

出现频率： ★★★★★

线程包括哪些状态，状态之间是如何变化的

线程的状态可以参考JDK中的Thread类中的枚举State

```
public enum State {  
  
    //尚未启动的线程的线程状态  
    NEW,  
  
    //可运行线程的线程状态。  
    RUNNABLE,  
  
    //线程阻塞等待监视器锁的线程状态。  
    BLOCKED,  
  
    //等待线程的线程状态  
    WAITING,  
  
    //具有指定等待时间的等待线程的线程状态  
    TIMED_WAITING,  
  
    //已终止线程的线程状态。线程已完成执行  
    TERMINATED;  
}
```



```
private static void testWaiting() {
    Thread t2 = new Thread() -> {
        ...
    };
    t2.start();
}

private static void testTimedWaiting() {
    Thread t1 = new Thread() -> {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        logger1.debug("running...");
    }, "t1";
    t1.start();
}
```



总结

1. 线程包括哪些状态

新建 (NEW)、可运行 (RUNNABLE)、阻塞 (BLOCKED)、等待 (WAITING)、时间等待 (TIMED_WAITING)、终止 (TERMINATED)

2. 线程状态之间是如何变化的

- 创建线程对象是**新建状态**
- 调用了start()方法转变为**可执行状态**
- 线程获取到了CPU的执行权，执行结束是**终止状态**
- 在可执行状态的过程中，如果没有获取CPU的执行权，可能会切换其他状态
 - ◆ 如果没有获取锁 (synchronized或lock) 进入**阻塞状态**，获得锁再切换为可执行状态
 - ◆ 如果线程调用了wait()方法进入**等待状态**，其他线程调用notify()唤醒后可切换为可执行状态
 - ◆ 如果线程调用了sleep(50)方法，进入**计时等待状态**，到时间后可切换为可执行状态

新建 T1、T2、T3 三个线程，如何保证它们按顺序执行？

难易程度： ★★☆☆☆

出现频率： ★★★☆☆

新建 T1、T2、T3 三个线程，如何保证它们按顺序执行？

可以使用线程中的join方法解决

join() 等待线程运行结束

小例子：

t.join()

阻塞调用此方法的线程进入timed_waiting

直到线程t执行完成后，此线程再继续执行

```
Thread t1 = new Thread() -> {
    System.out.println("t1");
};
Thread t2 = new Thread() -> {
    try {
        t1.join(); // 加入线程t1,只有t1线程执行完毕以后，再次执行该线程
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("t2");
};
Thread t3 = new Thread() -> {
    try {
        t2.join(); // 加入线程t2,只有t2线程执行完毕以后，再次执行该线程
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("t3");
};
// 启动线程
t1.start();
t2.start();
t3.start();
```

notify()和 notifyAll()有什么区别？

难易程度： ★★☆☆☆

出现频率： ★★☆☆☆

notify()和 notifyAll()有什么区别？

- notifyAll：唤醒所有wait的线程
- notify：只随机唤醒一个 wait 线程

java中wait和sleep方法的不同？

难易程度： ★★★★★

出现频率： ★★★★★

在java中wait和sleep方法的不同？

共同点

wait() , wait(long) 和 sleep(long) 的效果都是让当前线程暂时放弃 CPU 的使用权，进入阻塞状态

不同点

1.方法归属不同

- sleep(long) 是 Thread 的静态方法
- 而 wait() , wait(long) 都是 Object 的成员方法，每个对象都有

2.醒来时机不同

- 执行 sleep(long) 和 wait(long) 的线程都会在等待相应毫秒后醒来
- wait(long) 和 wait() 还可以被 notify 唤醒，wait() 如果不唤醒就一直等下去
- 它们都可以被打断唤醒

3.锁特性不同（重点）

- wait 方法的调用必须先获取 wait 对象的锁，而 sleep 则无此限制
- wait 方法执行后会释放对象锁，允许其它线程获得该对象锁（我放弃 cpu，但你们还可以用）
- 而 sleep 如果在 synchronized 代码块中执行，并不会释放对象锁（我放弃 cpu，你们也用不了）

如何停止一个正在运行的线程？

难易程度： ★★☆☆☆

出现频率： ★★☆☆☆

如何停止一个正在运行的线程？

有三种方式可以停止线程

- 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止
- 使用stop方法强行终止（不推荐，方法已作废）
- 使用interrupt方法中断线程
 - ◆ 打断阻塞的线程（sleep，wait，join）的线程，线程会抛出InterruptedException异常
 - ◆ 打断正常的线程，可以根据打断状态来标记是否退出线程

线程的基础知识

完成

线程与进程的区别

并行与并发的区别

线程创建的方式有哪些

Runnable 和 Callable 有什么区别

线程包括哪些状态，状态之间是如何变化的

在Java中wait和sleep方法的不同

新建三个线程，如何保证它们按顺序执行

notify()和 notifyAll()有什么区别

线程的 run()和 start()有什么区别

如何停止一个正在运行的线程

线程中并发安全

synchronized关键字的底层原理

你谈谈 JMM (Java 内存模型)

CAS 你知道吗

什么是AQS

ReentrantLock的实现原理

synchronized和Lock有什么区别

死锁产生的条件是什么

如何进行死锁诊断

请谈谈你对 volatile 的理解

聊一下ConcurrentHashMap

导致并发程序出现问题的根本原因是什么

线程池

说一下线程池的核心参数（线程池的执行原理知道嘛）

线程池中有哪些常见的阻塞队列

如何确定核心线程数

线程池的种类有哪些

为什么不建议用Executors创建线程池

使用场景

线程池使用场景(你们项目中哪里用到了线程池)

如何控制某个方法允许并发访问线程的数量

谈谈你对ThreadLocal的理解

synchronized关键字的底层原理

难易程度： ★★★★★

出现频率： ★★★★★

基本使用回顾

```
public class TicketDemo {  
  
    static Object lock = new Object();  
    int ticketNum = 10;  
    public void getTicket() {  
        synchronized (lock){  
            if (ticketNum <= 0) {  
                return;  
            }  
            System.out.println(Thread.currentThread().getName() + "抢到一张票,剩余:" + ticketNum);  
            // 非原子性操作  
            ticketNum--;  
        }  
    }  
  
    public static void main(String[] args) {  
        TicketDemo ticketDemo = new TicketDemo();  
        for (int i = 0; i < 20; i++) {  
            new Thread() -> {  
                ticketDemo.getTicket();  
            }.start();  
        }  
    }  
}
```

Synchronized【对象锁】采用互斥的方式让同一时刻至多只有一个线程能持有【对象锁】，其它线程再想获取这个【对象锁】时就会阻塞住

Monitor

```
public class SyncTest {  
  
    static final Object lock = new Object();  
    static int counter = 0;  
    public static void main(String[] args) {  
        synchronized (lock) {  
            counter++;  
        }  
    }  
}
```

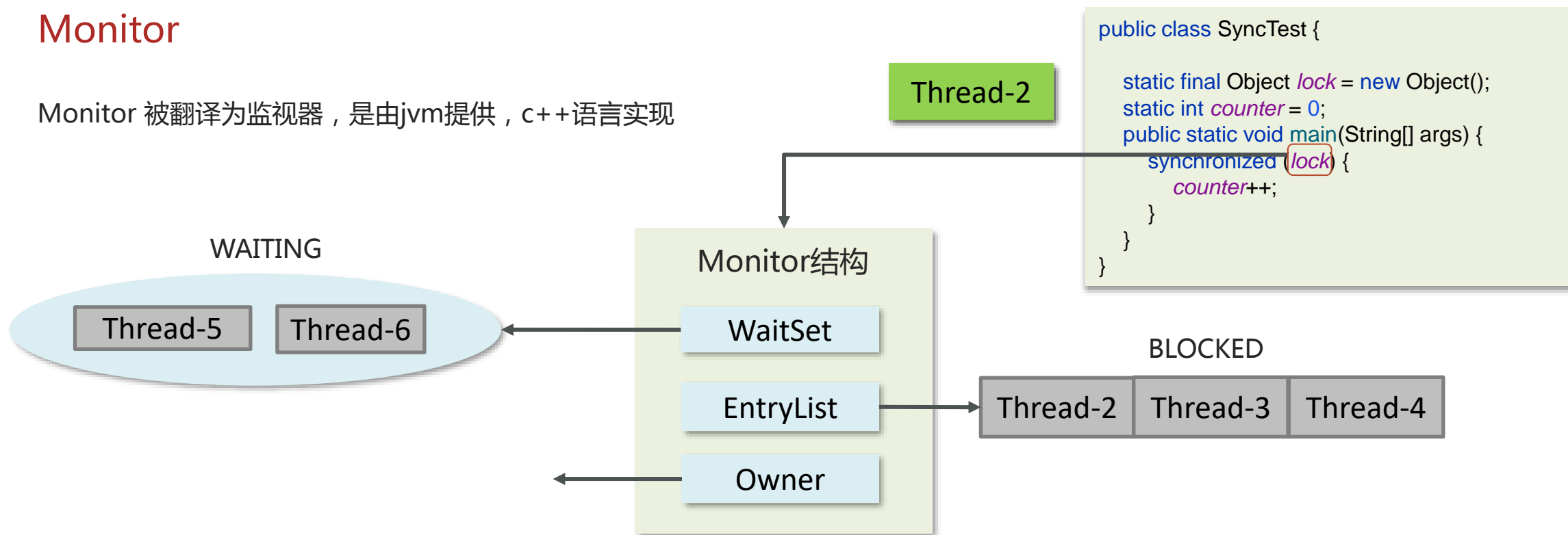
javap -v xx.class 查看class字节码信息

class反汇编

```
public static void main(java.lang.String[]);  
  descriptor: ([Ljava/lang/String;)V  
  flags: ACC_PUBLIC, ACC_STATIC  
  Code:  
    stack=2, locals=3, args_size=1  
     0: getstatic      #2                // Field lock:Ljava/lang/Object;  
     3: dup  
     4: astore_1  
     5: monitorenter    上锁 (对象锁)  
     6: getstatic      #3                // Field counter:I  
     9: iconst_1  
    10: iadd  
    11: putstatic      #3                // Field counter:I  
    14: aload_1  
    15: monitorexit    解锁 (对象锁)  
    16: goto          24  
    19: astore_2  
    20: aload_1  
    21: monitorexit    解锁 (对象锁)  
    22: aload_2  
    23: athrow  
    24: return
```

Monitor

Monitor 被翻译为监视器，是由jvm提供，c++语言实现



- Owner：存储当前获取锁的线程的，只能有一个线程可以获取
- EntryList：关联没有抢到锁的线程，处于Blocked状态的线程
- WaitSet：关联调用了wait方法的线程，处于Waiting状态的线程



总结

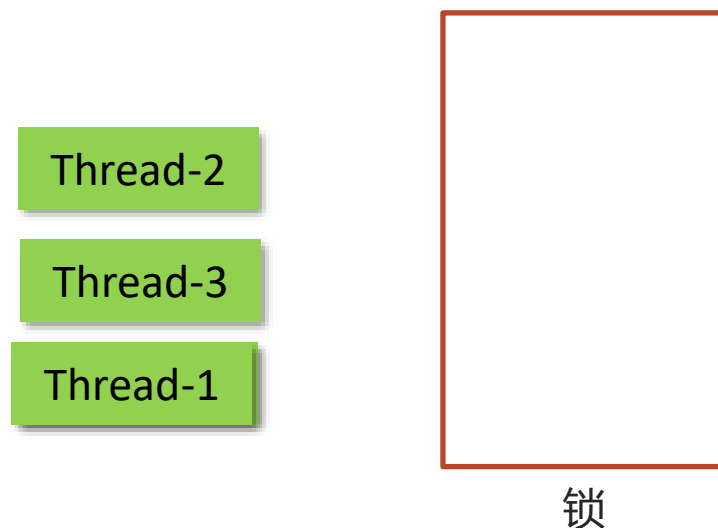
synchronized关键字的底层原理

- Synchronized【对象锁】采用互斥的方式让同一时刻至多只有一个线程能持有【对象锁】
- 它的底层由monitor实现的，monitor是jvm级别的对象（C++实现），线程获得锁需要使用对象（锁）关联monitor
- 在monitor内部有三个属性，分别是owner、entrylist、waitset
- 其中owner是关联的获得锁的线程，并且只能关联一个线程；entrylist关联的是处于阻塞状态的线程；waitset关联的是处于Waiting状态的线程

synchronized关键字的底层原理-进阶

Monitor实现的锁属于重量级锁，你了解过锁升级吗？

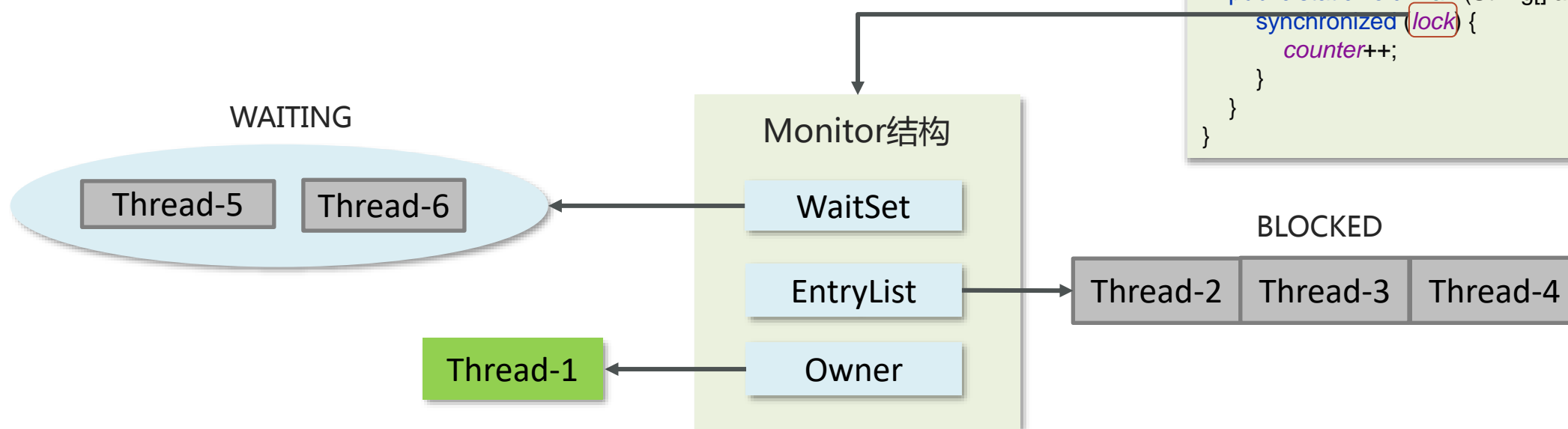
- Monitor实现的锁属于重量级锁，里面涉及到了用户态和内核态的切换、进程的上下文切换，成本较高，性能比较低。
- 在JDK 1.6引入了两种新型锁机制：**偏向锁**和**轻量级锁**，它们的引入是为了解决在没有多线程竞争或基本没有竞争的场景下因使用传统锁机制带来的性能开销问题。



Monitor重量级锁

Monitor 被翻译为监视器，是由jvm提供，c++语言实现

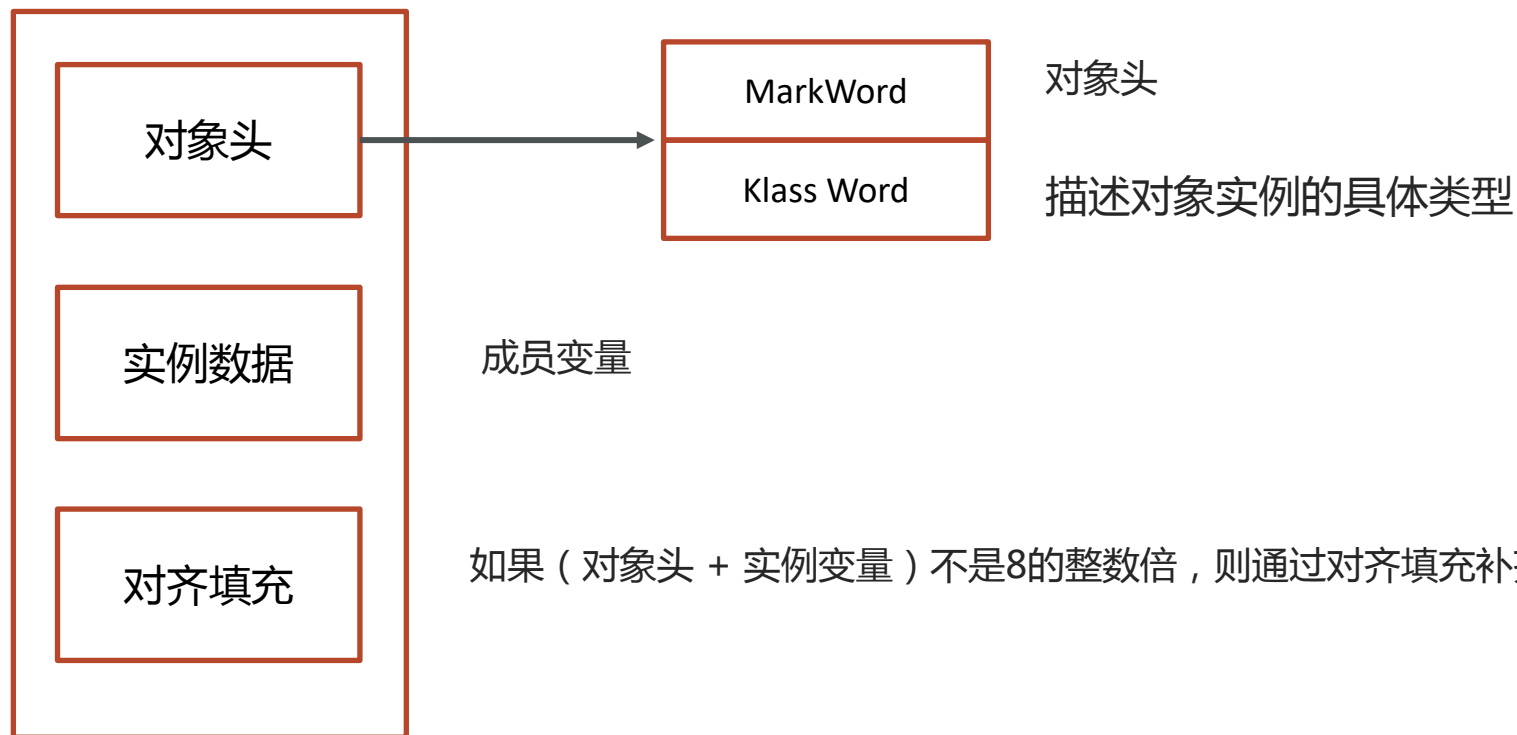
对象怎么关联上的Monitor



- Owner：存储当前获取锁的线程的，只能有一个线程可以获取
- EntryList：关联没有抢到锁的线程，处于Blocked状态的线程
- WaitSet：关联调用了wait方法的线程，处于Waiting状态的线程

对象的内存结构

在HotSpot虚拟机中，对象在内存中存储的布局可分为3块区域：对象头（Header）、实例数据（Instance Data）和对齐填充



MarkWord

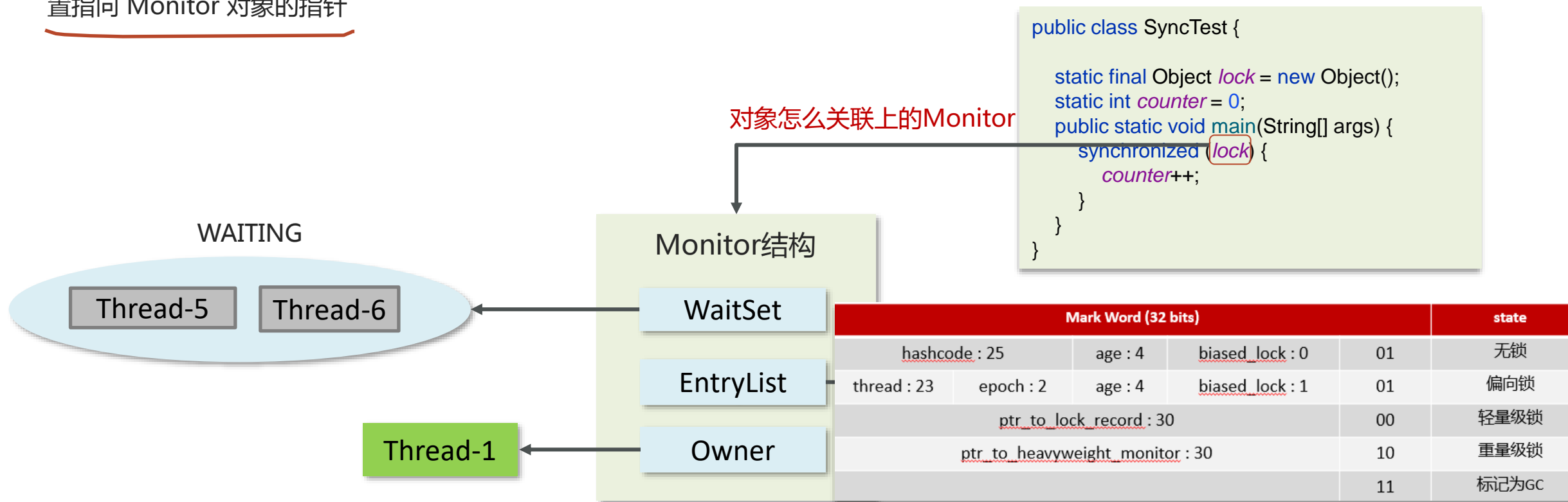
Mark Word (32 bits)					state
hashcode : 25		age : 4	biased_lock : 0	01	无锁
thread : 23	epoch : 2	age : 4	biased_lock : 1	01	偏向锁
ptr_to_lock_record : 30				00	轻量级锁
ptr_to_heavyweight_monitor : 30				10	重量级锁
				11	标记为GC

- hashcode : 25位的对象标识Hash码
- age : 对象分代年龄占4位
- biased_lock : 偏向锁标识，占1位，0表示没有开始偏向锁，1表示开启了偏向锁
- thread : 持有偏向锁的线程ID，占23位
- epoch : 偏向时间戳，占2位
- ptr_to_lock_record : 轻量级锁状态下，指向栈中锁记录的指针，占30位
- ptr_to_heavyweight_monitor : 重量级锁状态下，指向对象监视器Monitor的指针，占30位

lock标识，占2位

Monitor重量级锁

每个 Java 对象都可以关联一个 Monitor 对象，如果使用 synchronized 给对象上锁（重量级）之后，该对象头的Mark Word 中就被设置指向 Monitor 对象的指针



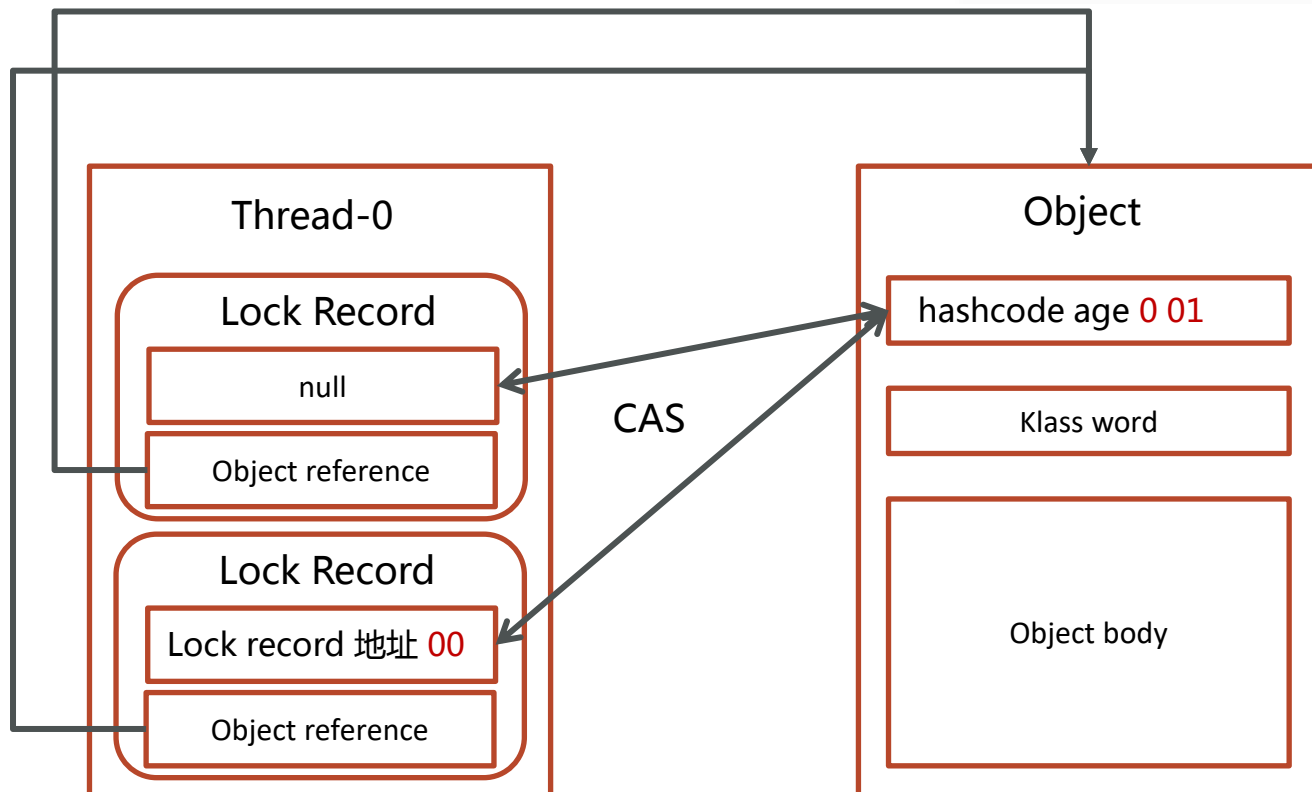
轻量级锁

在很多的情况下，在Java程序运行时，同步块中的代码都是不存在竞争的，不同的线程交替的执行同步块中的代码。这种情况下，用重量级锁是没必要的。因此JVM引入了轻量级锁的概念。

```
static final Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块 A
        method2();
    }
}
public static void method2() {
    synchronized( obj ) {
        // 同步块 B
    }
}
```

轻量级锁

Mark Word (32 bits)					state
hashCode : 25		age : 4	biased_lock : 0	01	无锁
thread : 23	epoch : 2	age : 4	biased_lock : 1	01	偏向锁
ptr_to_lock_record : 30				00	轻量级锁
ptr_to_heavyweight_monitor : 30				10	重量级锁
				11	标记为GC



```
static final Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块 A
        method2();
    }
}
public static void method2() {
    synchronized( obj ) {
        // 同步块 B
    }
}
```


轻量级锁

加锁流程

- 1.在线程栈中创建一个Lock Record，将其obj字段指向锁对象。
- 2.通过CAS指令将Lock Record的地址存储在对象头的mark word中，如果对象处于无锁状态则修改成功，代表该线程获得了轻量级锁。
- 3.如果是当前线程已经持有该锁了，代表这是一次锁重入。设置Lock Record第一部分为null，起到了一个重入计数器的作用。
- 4.如果CAS修改失败，说明发生了竞争，需要膨胀为重量级锁。

解锁过程

- 1.遍历线程栈,找到所有obj字段等于当前锁对象的Lock Record。
- 2.如果Lock Record的Mark Word为null，代表这是一次重入，将obj设置为null后continue。
- 3.如果Lock Record的 Mark Word不为null，则利用CAS指令将对象头的mark word恢复成为无锁状态。如果失败则膨胀为重量级锁。

偏向锁

轻量级锁在没有竞争时（就自己这个线程），每次重入仍然需要执行 CAS 操作。

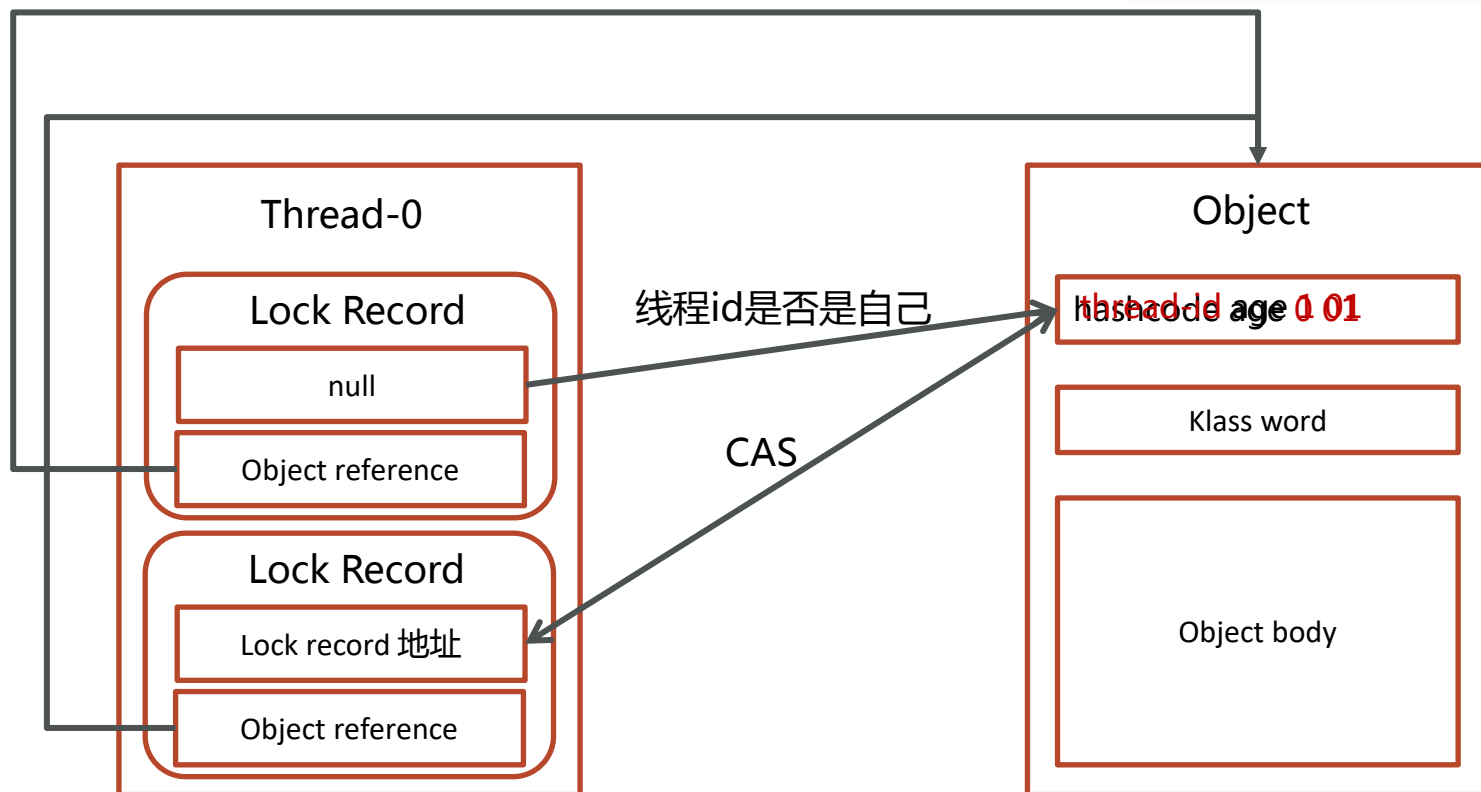
Java 6 中引入了偏向锁来做进一步优化：只有第一次使用 CAS 将线程 ID 设置到对象的 Mark Word 头，之后发现这个线程 ID 是自己的就表示没有竞争，不用重新 CAS。以后只要不发生竞争，这个对象就归该线程所有

```
static final Object obj = new Object();
public static void m1 () {
    synchronized (obj) {
        // 同步块 A
        m2();
    }
}
public static void m2 () {
    synchronized (obj) {
        // 同步块 B
        m3();
    }
}
public static void m3 () {
    synchronized (obj) {

    }
}
```

偏向锁

Mark Word (32 bits)				state
hashCode : 25	age : 4	biased_lock : 0	01	无锁
thread : 23	epoch : 2	age : 4	biased_lock : 1	偏向锁
ptr_to_lock_record : 30			00	轻量级锁
ptr_to_heavyweight_monitor : 30			10	重量级锁
			11	标记为GC



```
static final Object obj = new Object();
public static void m1 () {
    synchronized (obj) {
        // 同步块 A
        m2();
    }
}
public static void m2 () {
    synchronized (obj) {
        // 同步块 B
        m3();
    }
}
public static void m3 () {
    synchronized (obj) {

    }
}
```

Monitor实现的锁属于重量级锁，你了解过锁升级吗？

Java中的synchronized有偏向锁、轻量级锁、重量级锁三种形式，分别对应了锁只被一个线程持有、不同线程交替持有锁、多线程竞争锁三种情况。

	描述
重量级锁	底层使用的Monitor实现，里面涉及到了用户态和内核态的切换、进程的上下文切换，成本较高，性能比较低。
轻量级锁	线程加锁的时间是错开的（也就是没有竞争），可以使用轻量级锁来优化。轻量级修改了对象头的锁标志，相对重量级锁性能提升很多。每次修改都是CAS操作，保证原子性
偏向锁	一段很长的时间内都只被一个线程使用锁，可以使用了偏向锁，在第一次获得锁时，会有一个CAS操作，之后该线程再获取锁，只需要判断mark word中是否是自己的线程id即可，而不是开销相对较大的CAS命令

一旦锁发生了竞争，都会升级为重量级锁

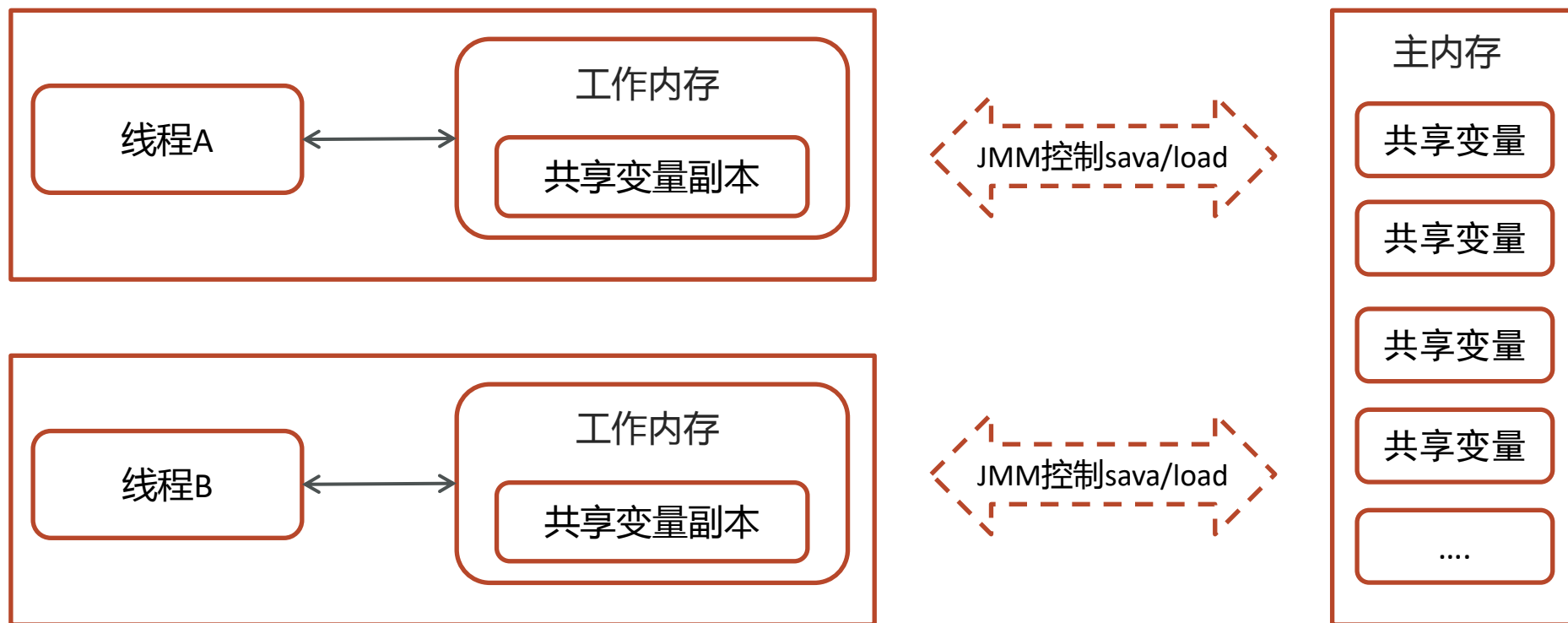
你谈谈 JMM (Java内存模型)

难易程度： ★★★★★

出现频率： ★★★★★

Java 内存模型

JMM(Java Memory Model)Java内存模型，定义了共享内存中多线程程序读写操作的行为规范，通过这些规则来规范对内存的读写操作从而保证指令的正确性





总结

你谈谈 JMM (Java内存模型)

- JMM(Java Memory Model)Java内存模型，定义了共享内存中多线程程序读写操作的行为规范，通过这些规则来规范对内存的读写操作从而保证指令的正确性
- JMM把内存分为两块，一块是私有线程的工作区域（工作内存），一块是所有线程的共享区域（主内存）
- 线程跟线程之间是相互隔离，线程跟线程交互需要通过主内存

CAS 你知道吗？

难易程度： ★★★★★

出现频率： ★★★★★

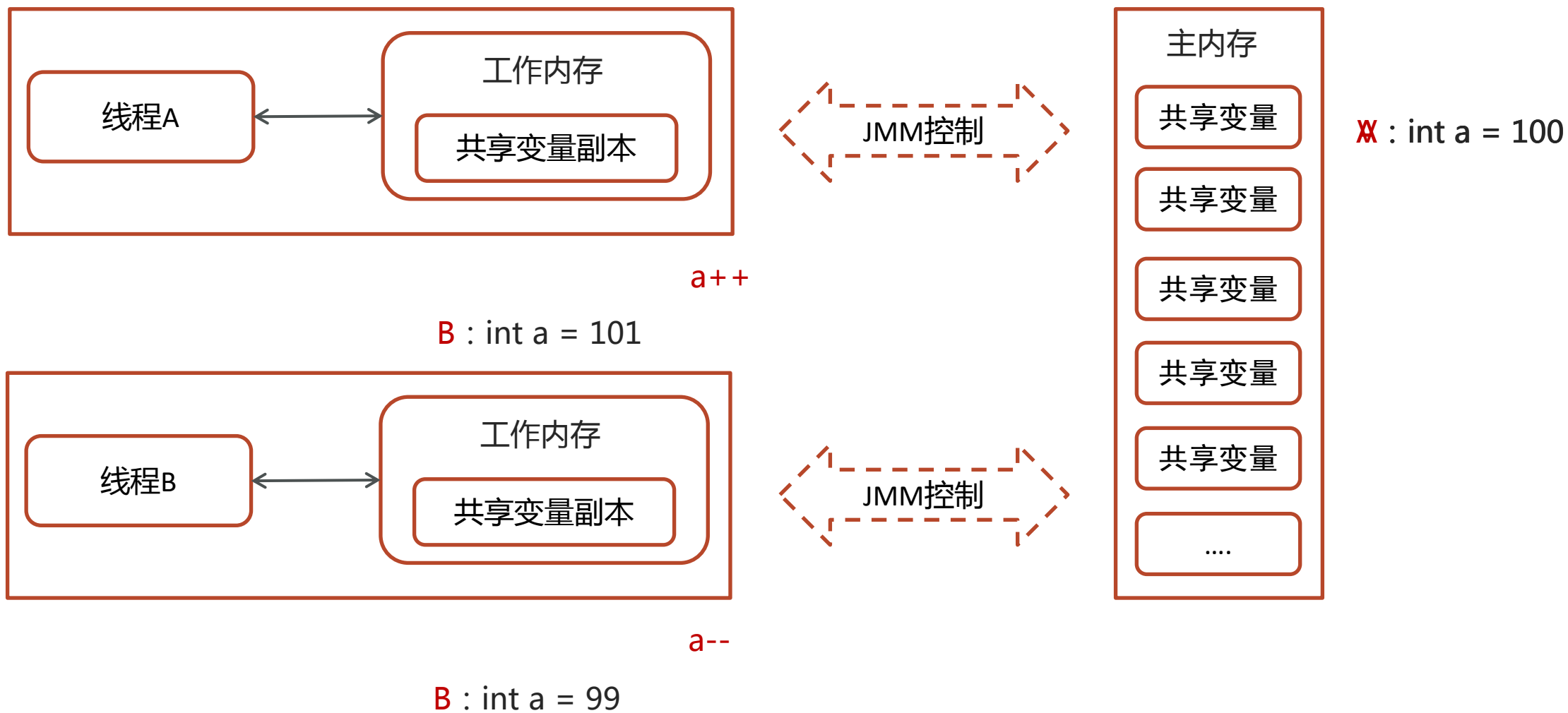
CAS

CAS的全称是： Compare And Swap(比较再交换)，它体现的一种乐观锁的思想，在无锁情况下保证线程操作共享数据的原子性。

在JUC (`java.util.concurrent`) 包下实现的很多类都用到了CAS操作

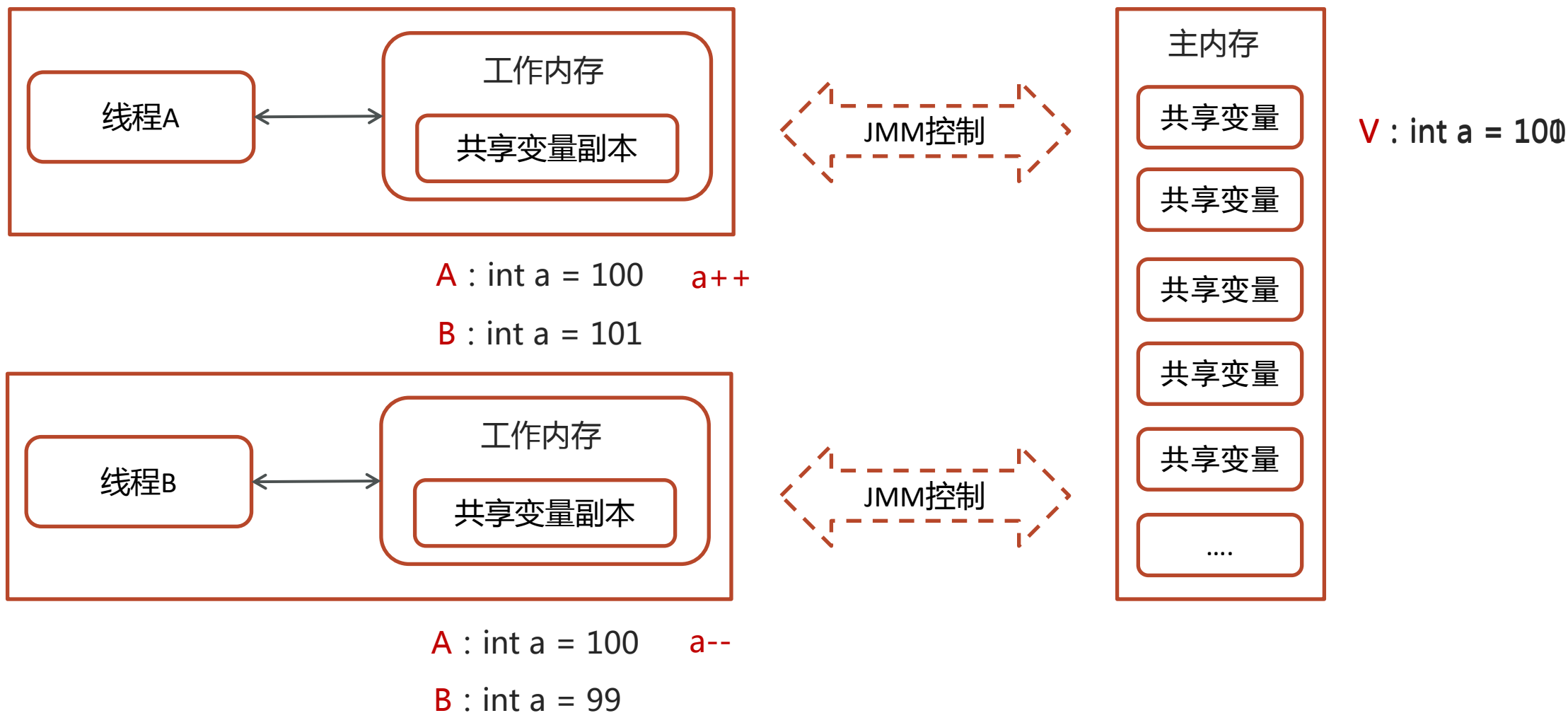
- AbstractQueuedSynchronizer (AQS框架)
- AtomicXXX类

CAS数据交换流程



CAS数据交换流程

一个当前内存值V、旧的预期值A、即将更新的值B，当且仅当旧的预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做，并返回false。如果CAS操作失败，通过自旋的方式等待并再次尝试，直到成功



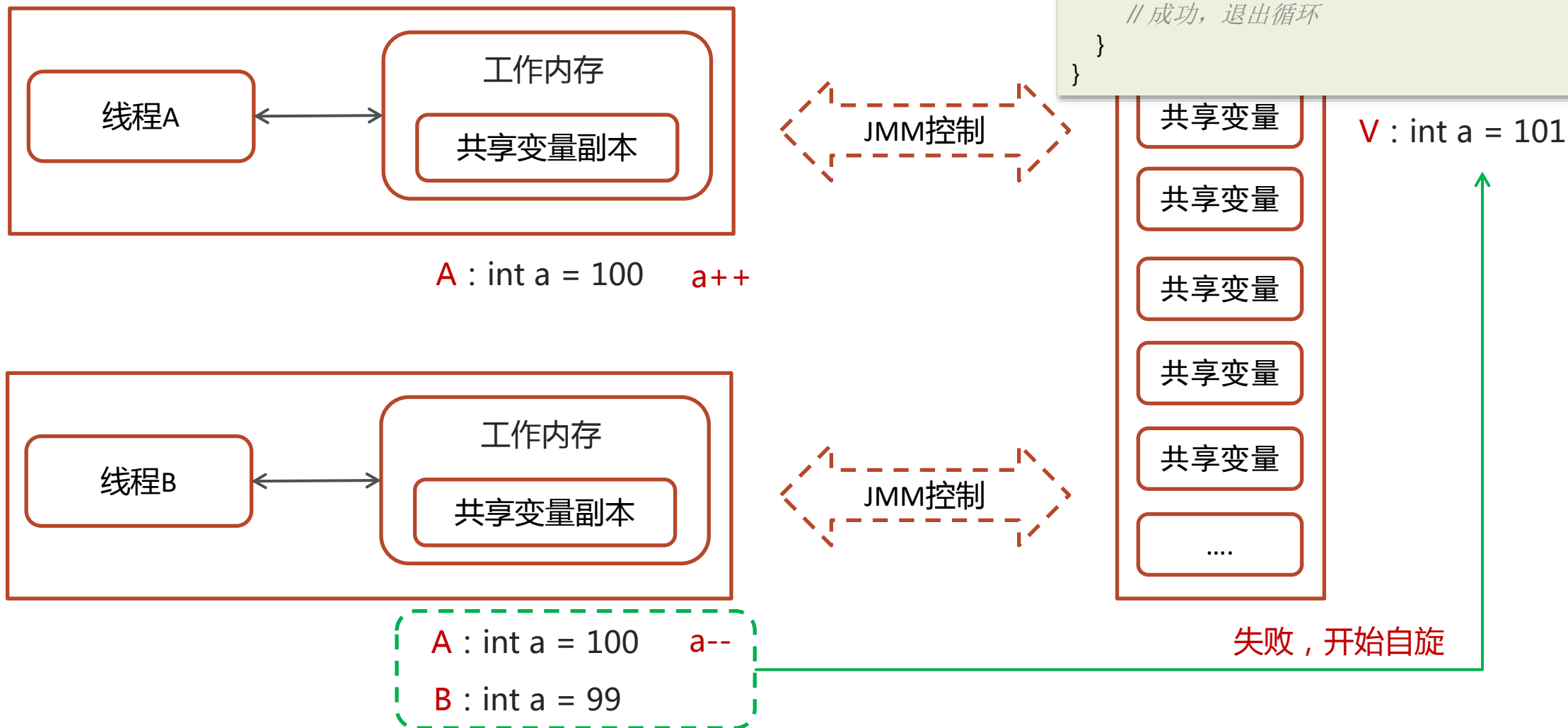
CAS数据交换流程

- 因为没有加锁，所以线程不会陷入阻塞，效率较高
- 如果竞争激烈，重试频繁发生，效率会受影响

// 需要不断尝试

```
while(true){  
    int 旧值A = 共享变量V;  
    int 结果B = 旧值 + 1;  
    if (compareAndSwap(旧值, 结果)) {  
        // 成功，退出循环  
    }  
}
```

自旋锁



CAS 底层实现

CAS 底层依赖于一个 Unsafe 类来直接调用操作系统底层的 CAS 指令

ReentrantLock中的一段CAS代码

```
protected final boolean compareAndSetState(int expect, int update) {  
    return STATE.compareAndSet(this, expect, update);  
}
```

当前值

期望的值

更新后的值

```
// 需要不断尝试  
while(true){  
    int 旧值A = 共享变量V;  
    int 结果B = 旧值 + 1;  
    if (compareAndSwap(旧值, 结果)) {  
        // 成功, 退出循环  
    }  
}
```

乐观锁和悲观锁

- CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
- synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。

```
// 需要不断尝试
while(true){
    int 旧值A = 共享变量V;
    int 结果B = 旧值 + 1;
    if (compareAndSwap(旧值, 结果)) {
        // 成功, 退出循环
    }
}
```



总结

CAS 你知道吗？

- CAS的全称是：Compare And Swap(比较再交换);它体现的一种乐观锁的思想，在无锁状态下保证线程操作数据的原子性。
- CAS使用到的地方很多：AQS框架、AtomicXXX类
- 在操作共享变量的时候使用的自旋锁，效率上更高一些
- CAS的底层是调用的Unsafe类中的方法，都是操作系统提供的，其他语言实现

乐观锁和悲观锁的区别

- CAS 是基于乐观锁的思想：最乐观的估计，不怕别的线程来修改共享变量，就算改了也没关系，我吃亏点再重试呗。
- synchronized 是基于悲观锁的思想：最悲观的估计，得防着其它线程来修改共享变量，我上了锁你们都别想改，我改完了解开锁，你们才有机会。

请谈谈你对 volatile 的理解

难易程度： ★★★★★

出现频率： ★★★★★

请谈谈你对 volatile 的理解

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

- ① 保证线程间的可见性
- ② 禁止进行指令重排序

保证线程间的可见性

用 volatile 修饰共享变量，能够防止编译器等优化发生，让一个线程对共享变量的修改对另一个线程可见

```
static boolean stop = false;
public static void main(String[] args) {
    new Thread() -> {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        stop = true;
        System.out.println(Thread.currentThread().getName()+" : modify stop to true...");
    }, "t1").start();

    new Thread() -> {
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+" : "+stop);
    }, "t2").start();

    new Thread() -> {
        int i = 0;
        while (!stop) {
            i++;
        }
        System.out.println("stopped... c:"+ i);
    }, "t3").start();
}
```

请谈谈你对 volatile 的理解

问题分析：主要是因为 JVM 虚拟机中有一个 JIT（即时编译器）给代码做了优化。

```
while (!stop) {  
    i++;  
}
```

优化

```
while (true) {  
    i++;  
}
```

解决方案一：在程序运行的时候加入 vm 参数 -Xint 表示禁用即时编译器，不推荐，得不偿失（其他程序还要使用）

解决方案二：在修饰 stop 变量的时候加上 volatile，当前告诉 jit，不要对 volatile 修饰的变量做优化

volatile禁止指令重排序

用 volatile 修饰共享变量会在读、写共享变量时加入不同的屏障，阻止其他读写操作越过屏障，从而达到阻止重排序的效果

```
int x;  
int y;  
  
@Actor  
public void actor1() {  
    x = 1;  
    y = 1;  
}  
  
@Actor  
public void actor2(Il_Result r) {  
    r.r1 = y;  
    r.r2 = x;  
}
```

情况一：先执行actor2获取结果



0,0

情况二：先执行actor1中的第一行代码，然后执行actor2获取结果



0,1

情况三：先执行actor1中所有代码，然后执行actor2获取结果



1,1

情况四：先执行actor1中第二行代码，然后执行actor2获取结果



1,0

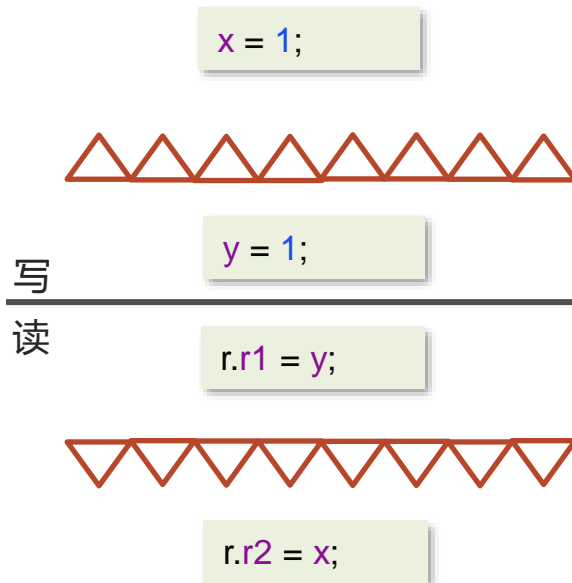
已经发生了指令重排序

注解@Actor保证方法内的代码在同一个线程下执行

volatile禁止指令重排序

在变量上添加volatile，禁止指令重排序，则可以解决问题

```
int x;  
volatile int y;  
  
@Actor  
public void actor1() {  
    x = 1;  
    y = 1;  
}  
  
@Actor  
public void actor2(Il_Result r) {  
    r.r1 = y;  
    r.r2 = x;  
}
```



写操作加的屏障是阻止上方其它写操作越过屏障排到volatile变量写之下

读操作加的屏障是阻止下方其它读操作越过屏障排到volatile变量读之上

volatile禁止指令重排序

在变量上添加volatile，禁止指令重排序，则可以解决问题

```
volatile int x;  
int y;  
  
@Actor  
public void actor1() {  
    x = 1;  
    y = 1;  
}  
  
@Actor  
public void actor2(Il_Result r) {  
    r.r1 = y;  
    r.r2 = x;  
}
```



x = 1;

y = 1;

写操作加的屏障是阻止上方其它写操作越过屏障排到volatile变量写之下

r.r1 = y;

r.r2 = x;



读操作加的屏障是阻止下方其它读操作越过屏障排到volatile变量读之上

volatile使用技巧：

- 写变量让volatile修饰的变量的在代码最后位置
- 读变量让volatile修饰的变量的在代码最开始位置

总结

1. 请谈谈你对 volatile 的理解

① 保证线程间的可见性

用 volatile 修饰共享变量，能够防止编译器等优化发生，让一个线程对共享变量的修改对另一个线程可见

② 禁止进行指令重排序

指令重排：用 volatile 修饰共享变量会在读、写共享变量时加入不同的屏障，阻止其他读写操作越过屏障，从而达到阻止重排序的效果

什么是AQS ?

难易程度： ★★★★★

出现频率： ★★★★★

什么是AQS ?

全称是 **AbstractQueuedSynchronizer**，即抽象队列同步器。它是构建锁或者其他同步组件的**基础框架**

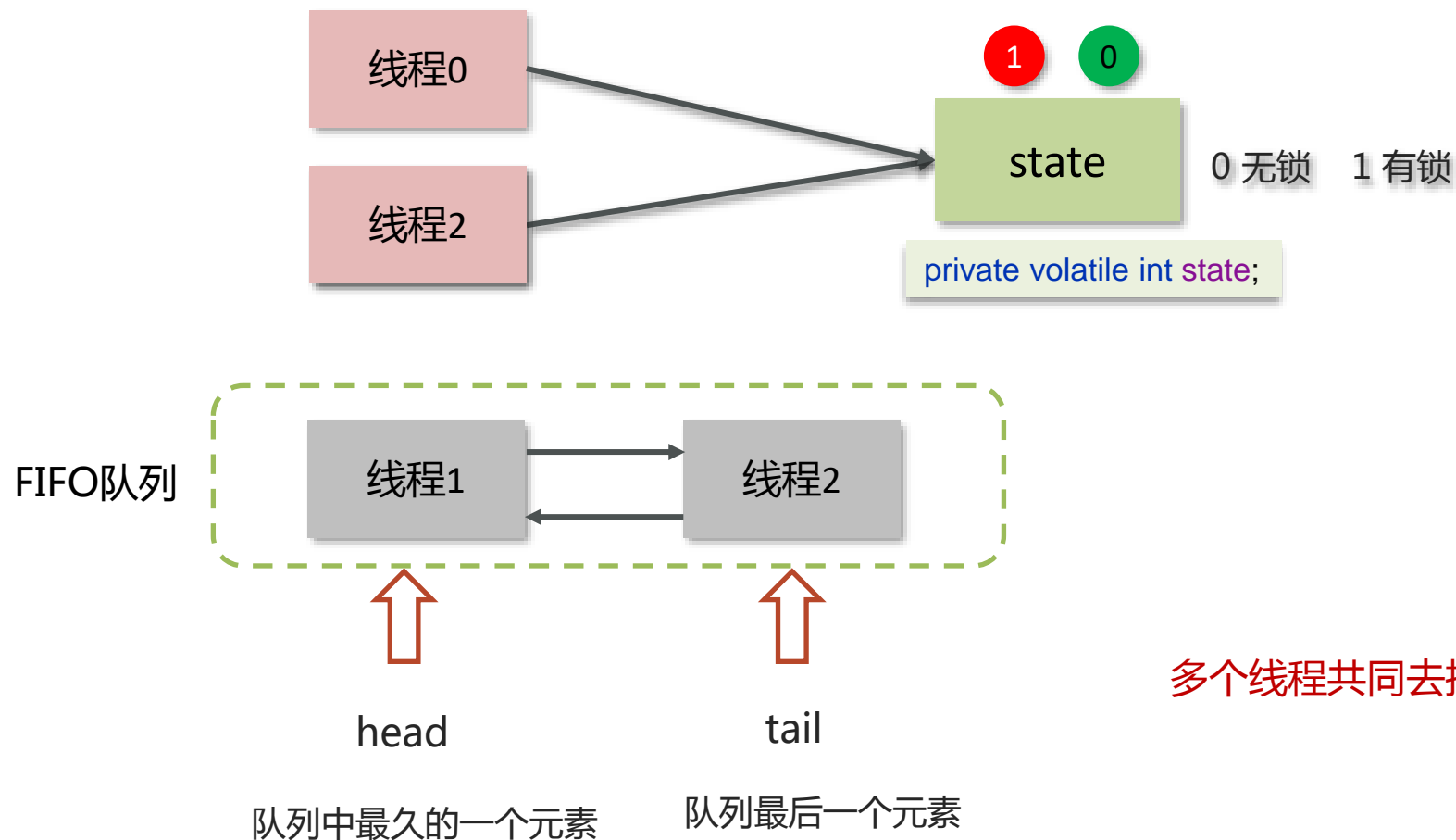
AQS与Synchronized的区别

synchronized	AQS
关键字，c++ 语言实现	java 语言实现
悲观锁，自动释放锁	悲观锁，手动开启和关闭
锁竞争激烈都是重量级锁，性能差	锁竞争激烈的情况下，提供了多种解决方案

AQS常见的实现类

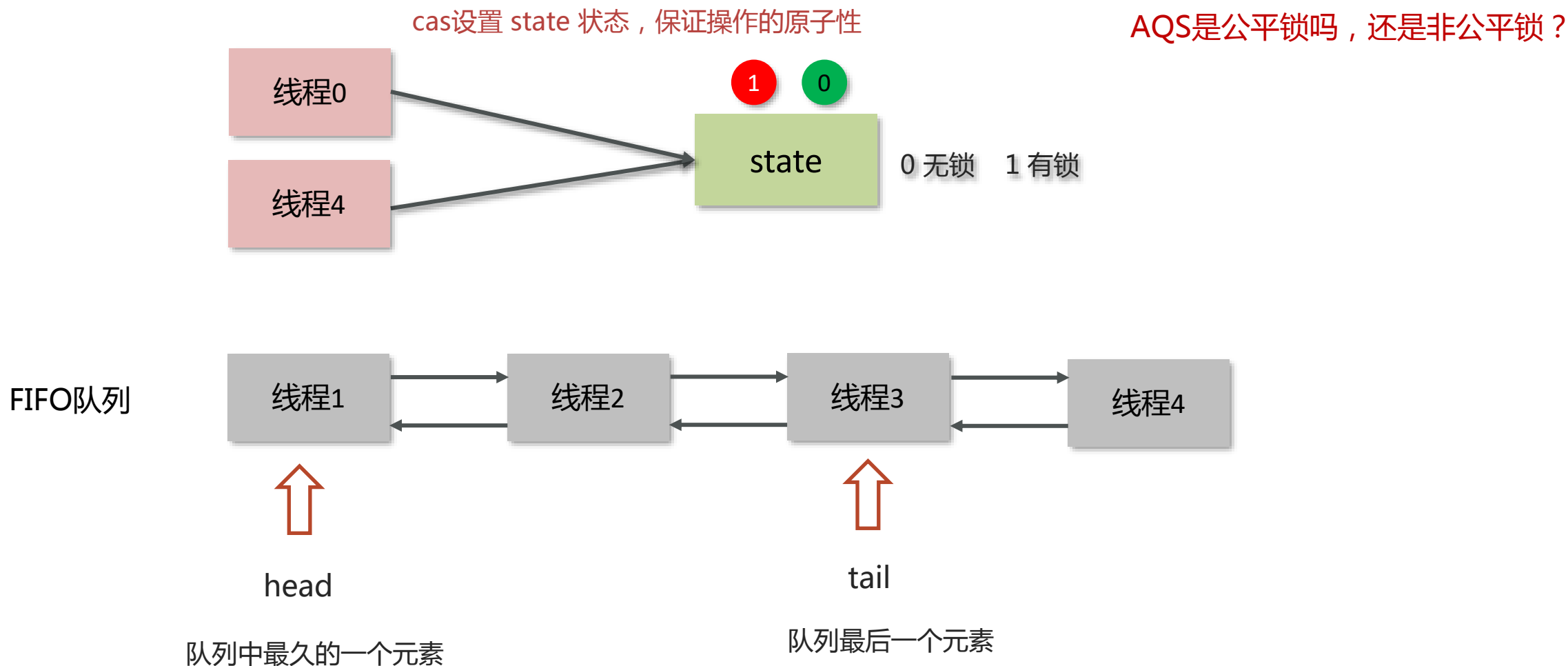
- ReentrantLock 阻塞式锁
- Semaphore 信号量
- CountdownLatch 倒计时锁

AQS-基本工作机制



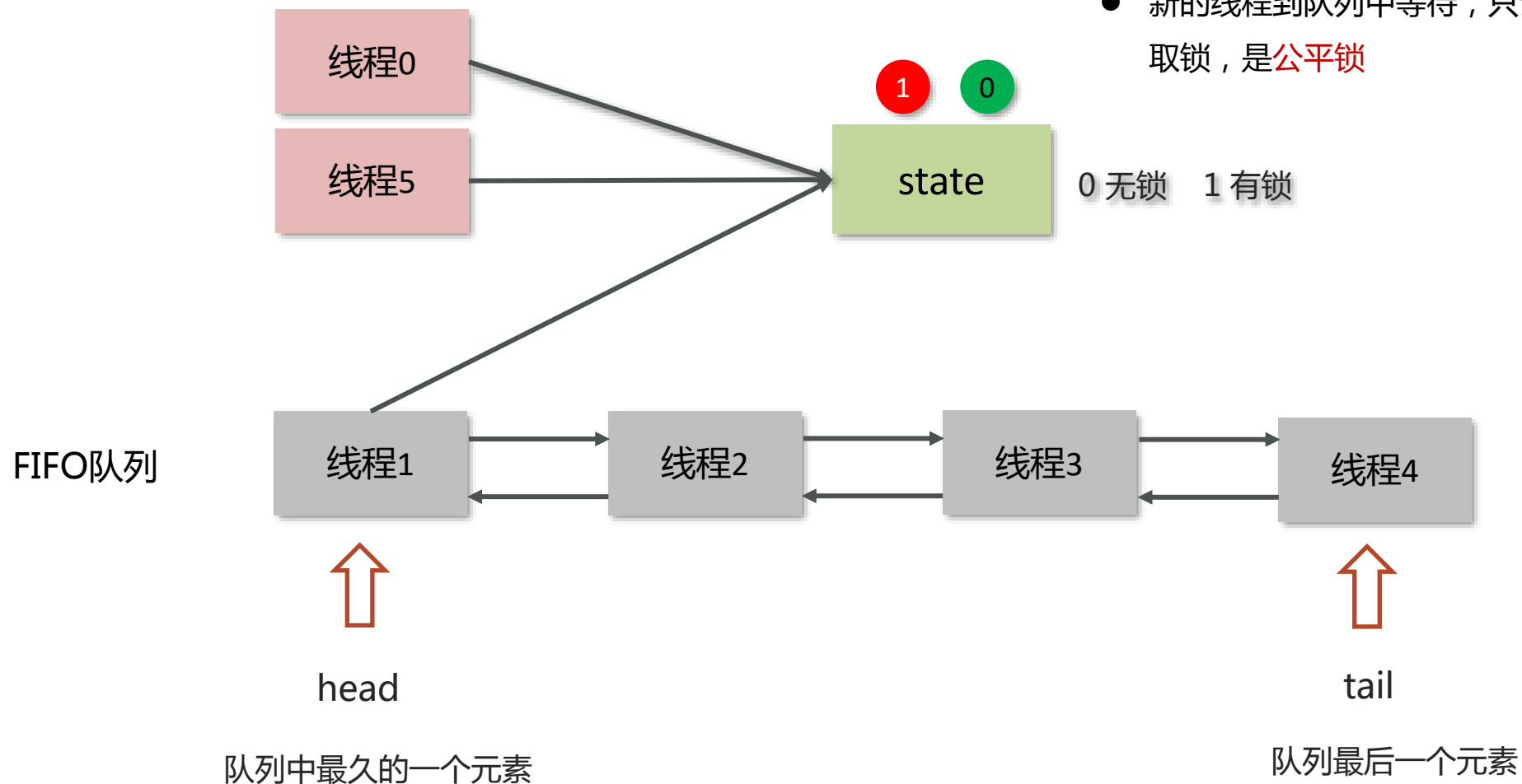
多个线程共同去抢这个资源是如何保证原子性的呢？

AQS-多个线程共同去抢这个资源是如何保证原子性的呢



AQS是公平锁吗，还是非公平锁？

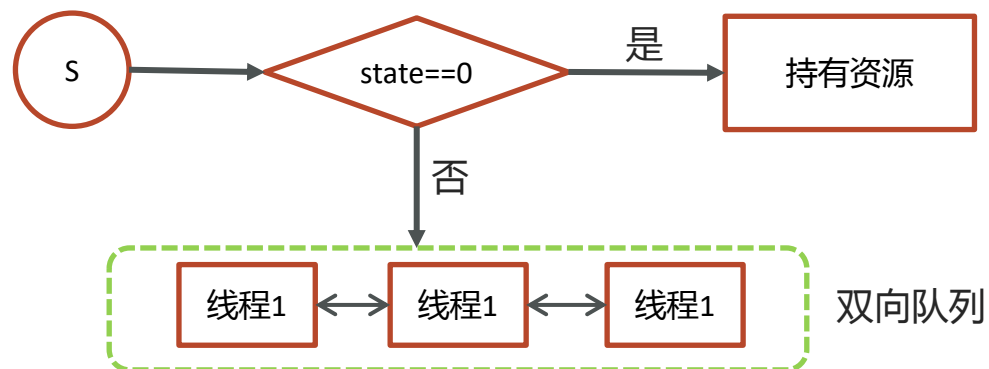
- 新的线程与队列中的线程共同来抢资源，是**非公平锁**
- 新的线程到队列中等待，只让队列中的head线程获取锁，是**公平锁**



总结

什么是AQS ?

- 是多线程中的队列同步器。是一种锁机制，它是做为一个基础框架使用的，像ReentrantLock、Semaphore都是基于AQS实现的
- AQS内部维护了一个先进先出的双向队列，队列中存储的排队的线程
- 在AQS内部还有一个属性state，这个state就相当于是一个资源，默认是0（无锁状态），如果队列中的有一个线程修改成功了state为1，则当前线程就相等于获取了资源
- 在对state修改的时候使用的cas操作，保证多个线程修改的情况下原子性



ReentrantLock的实现原理

难易程度： ★★★★★

出现频率： ★★★★★

ReentrantLock的实现原理

ReentrantLock翻译过来是可重入锁，相对于synchronized它具备以下特点：

- 可中断
- 可以设置超时时间
- 可以设置公平锁
- 支持多个条件变量
- 与synchronized一样，都支持重入

```
//创建锁对象
ReentrantLock lock = new ReentrantLock();
try {
    // 获取锁
    lock.lock();
} finally {
    // 释放锁
    lock.unlock();
}
```

ReentrantLock的实现原理

ReentrantLock主要利用CAS+AQS队列来实现。它支持公平锁和非公平锁，两者的实现类似

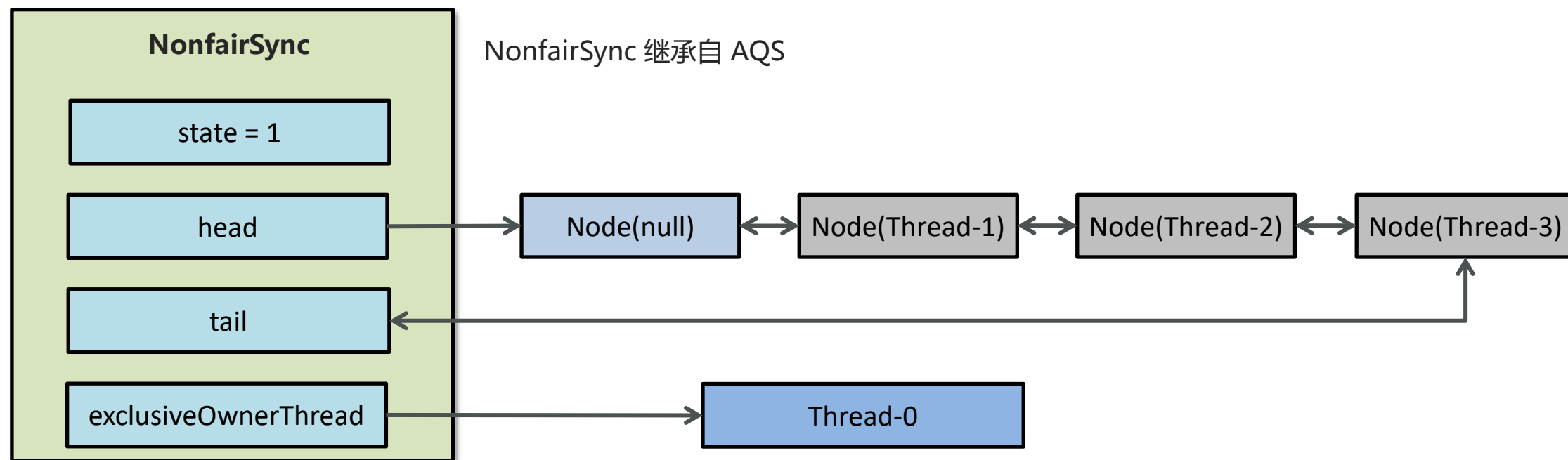
构造方法接受一个可选的公平参数（默认非公平锁），当设置为true时，表示公平锁，否则为非公平锁。公平锁的效率往往没有非公平锁的效率，在许多线程访问的情况下，公平锁表现出较低的吞吐量。

查看ReentrantLock源码中的构造方法：

```
public ReentrantLock() {  
    sync = new NonfairSync();  
}  
  
public ReentrantLock(boolean fair) {  
    sync = fair ? new FairSync() : new NonfairSync();  
}
```

```
abstract static class Sync extends AbstractQueuedSynchronizer {  
  
}
```


ReentrantLock的实现原理



- 线程来抢锁后使用cas的方式修改state状态，修改状态成功为1，则让exclusiveOwnerThread属性指向当前线程，获取锁成功
- 假如修改状态失败，则会进入双向队列中等待，head指向双向队列头部，tail指向双向队列尾部
- 当exclusiveOwnerThread为null的时候，则会唤醒在双向队列中等待的线程
- 公平锁则体现在按照先后顺序获取锁，非公平体现在不在排队的线程也可以抢锁



总结

ReentrantLock的实现原理

- ReentrantLock表示支持重新进入的锁，调用 lock 方法获取了锁之后，再次调用 lock，是不会再阻塞
- ReentrantLock主要利用CAS+AQS队列来实现
- 支持公平锁和非公平锁，在提供的构造器的中无参默认是非公平锁，也可以传参设置为公平锁

synchronized和Lock有什么区别？

难易程度： ★★★★★

出现频率： ★★★★★

synchronized和Lock有什么区别？

- 语法层面

synchronized 是关键字，源码在 jvm 中，用 c++ 语言实现

Lock 是接口，源码由 jdk 提供，用 java 语言实现

使用 synchronized 时，退出同步代码块锁会自动释放，而使用 Lock 时，需要手动调用 unlock 方法释放锁

- 功能层面

二者均属于悲观锁、都具备基本的互斥、同步、锁重入功能

Lock 提供了许多 synchronized 不具备的功能，例如公平锁、可打断、可超时、多条件变量

Lock 有适合不同场景的实现，如 ReentrantLock，ReentrantReadWriteLock(读写锁)

- 性能层面

在没有竞争时，synchronized 做了很多优化，如偏向锁、轻量级锁，性能不赖

在竞争激烈时，Lock 的实现通常会提供更好的性能

死锁产生的条件是什么？

难易程度： ★★★★★

出现频率： ★★★★★

死锁产生的条件是什么？

死锁：一个线程需要同时获取多把锁，这时就容易发生死锁



此时程序并没有结束，这种现象就是死锁现象...线程t1持有A的锁等待获取B锁，线程t2持有B的锁等待获取A的锁。

```
Object A = new Object();
Object B = new Object();
Thread t1 = new Thread() -> {
    synchronized (A) {
        System.out.println("lock A");
        try {
            sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        synchronized (B) {
            System.out.println("lock B");
            System.out.println("操作...");
        }
    }
}, "t1");

Thread t2 = new Thread() -> {
    synchronized (B) {
        System.out.println("lock B");
        try {
            sleep(500);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        synchronized (A) {
            System.out.println("lock A");
            System.out.println("操作...");
        }
    }
}, "t2");
t1.start();
t2.start();
```

如何进行死锁诊断？

当程序出现了死锁现象，我们可以使用jdk自带的工具：`jps`和 `jstack`

- `jps`：输出JVM中运行的进程状态信息
- `jstack`：查看java进程内线程的堆栈信息

如何进行死锁诊断？

解决步骤如下

第一：查看运行的线程

```
Terminal: Local x + v
PS D:\code\juc-project> jps
19056
46032 Deadlock
30360 Jps
46712 Launcher
PS D:\code\juc-project>
```

第二，使用jstack查看线程运行的情况，下图是截图的关键信息

运行命令：`jstack -l 46032`

```
Found one Java-level deadlock:
-----
"t2":
  waiting to lock monitor 0x0000001705cfd6bc8 (object 0x0000000716b5c5e8, a java.lang.Object),
  which is held by "t1"
    - waiting to lock <0x0000000716b5c5e8> (a java.lang.Object) 等待锁: 0x0000000716b5c5e8
    - locked <0x0000000716b5c5f8> (a java.lang.Object) 已经拥有的锁: 0x0000000716b5c5f8
    at com.itheima.basic.Deadlock$$Lambda$2/990368553.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:750)

"t1":
  at com.itheima.basic.Deadlock.lambda$main$0(Deadlock.java:19)
    - waiting to lock <0x0000000716b5c5f8> (a java.lang.Object) 等待锁: 0x0000000716b5c5f8
    - locked <0x0000000716b5c5e8> (a java.lang.Object) 已经拥有的锁: 0x0000000716b5c5e8
    at com.itheima.basic.Deadlock$$Lambda$1/2003749087.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:750)

Found 1 deadlock. 发现了一个死锁
```


如何进行死锁诊断？

其他解决工具，可视化工具

- jconsole

用于对jvm的内存，线程，类的监控，是一个基于 jmx 的 GUI 性能监控工具

打开方式：java 安装目录 bin目录下 直接启动 jconsole.exe 就行

- VisualVM：故障处理工具

能够监控线程，内存情况，查看方法的CPU时间和内存中的对象，已被GC的对象，反向查看分配的堆栈

打开方式：java 安装目录 bin目录下 直接启动 jvisualvm.exe就行



总结

1. 死锁产生的条件是什么？

一个线程需要同时获取多把锁，这时就容易发生死锁

2. 如何进行死锁诊断？

- 当程序出现了死锁现象，我们可以使用jdk自带的工具：jps和jstack
- jps：输出JVM中运行的进程状态信息
- jstack：查看java进程内线程的堆栈信息，查看日志，检查是否有死锁
如果有死锁现象，需要查看具体代码分析后，可修复
- 可视化工具jconsole、VisualVM也可以检查死锁问题

聊一下ConcurrentHashMap

难易程度： ★★★★★

出现频率： ★★★★★

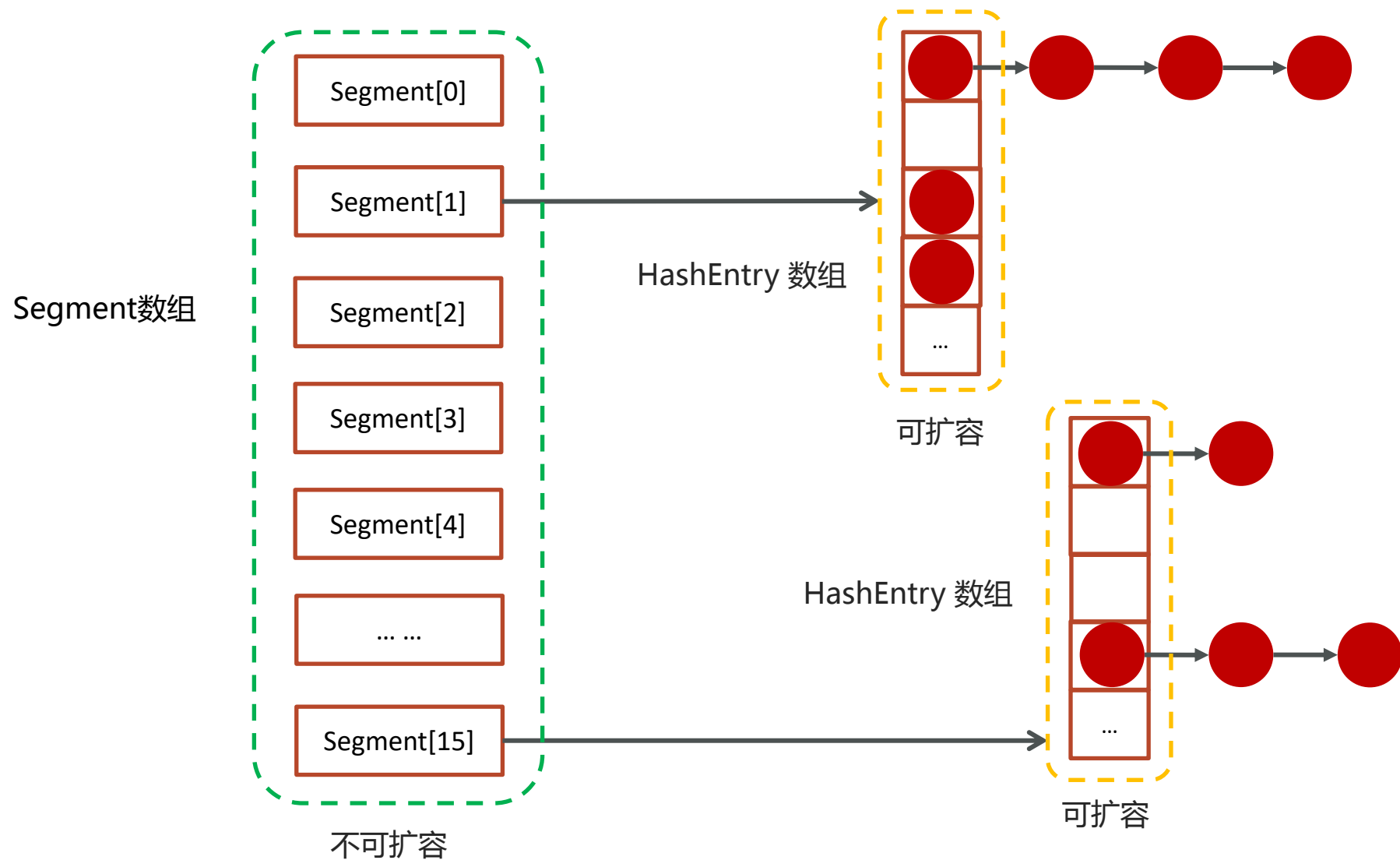
聊一下ConcurrentHashMap

ConcurrentHashMap 是一种线程安全的高效Map集合

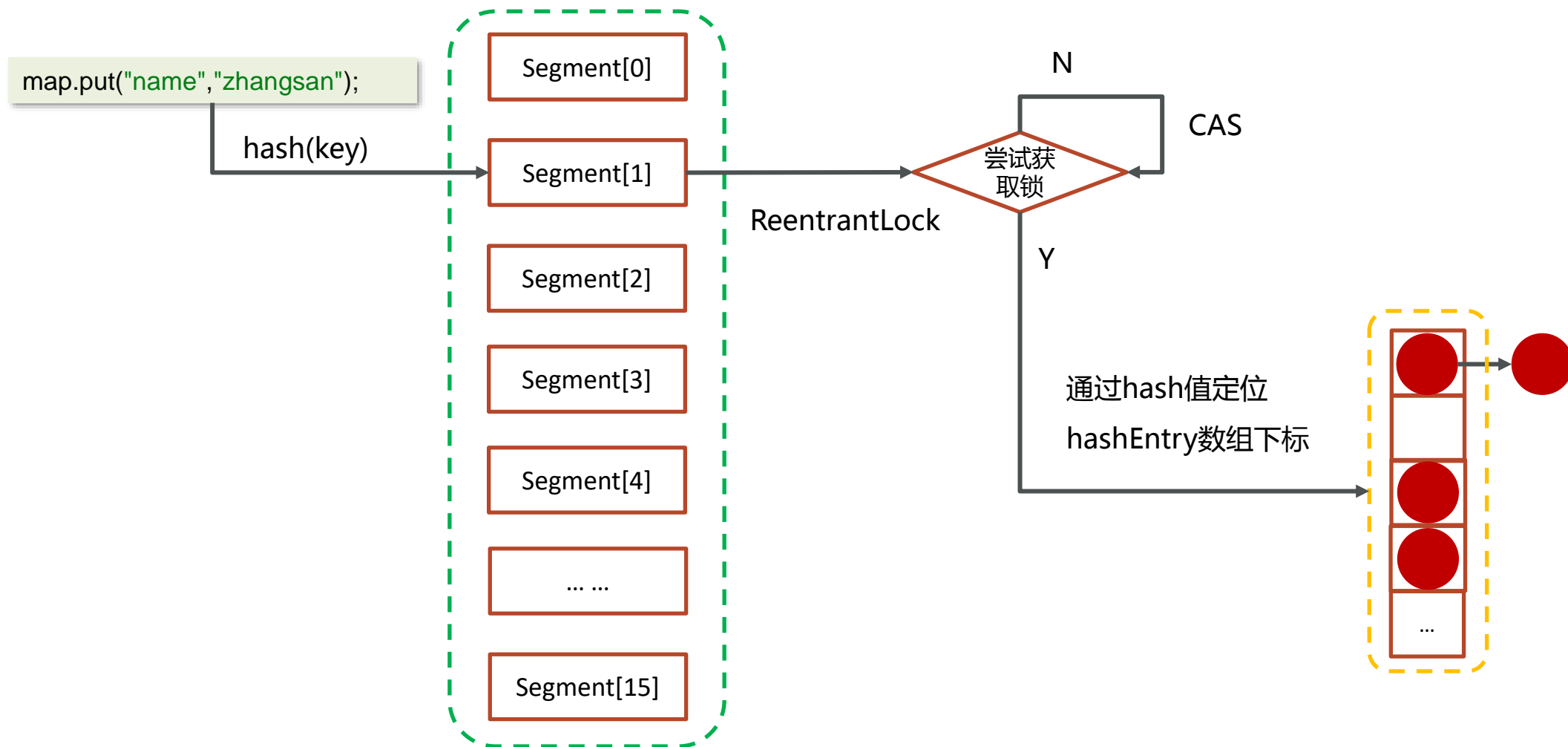
底层数据结构：

- JDK1.7底层采用分段的数组+链表实现
- JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。

JDK1.7中ConcurrentHashMap



JDK1.7中ConcurrentHashMap

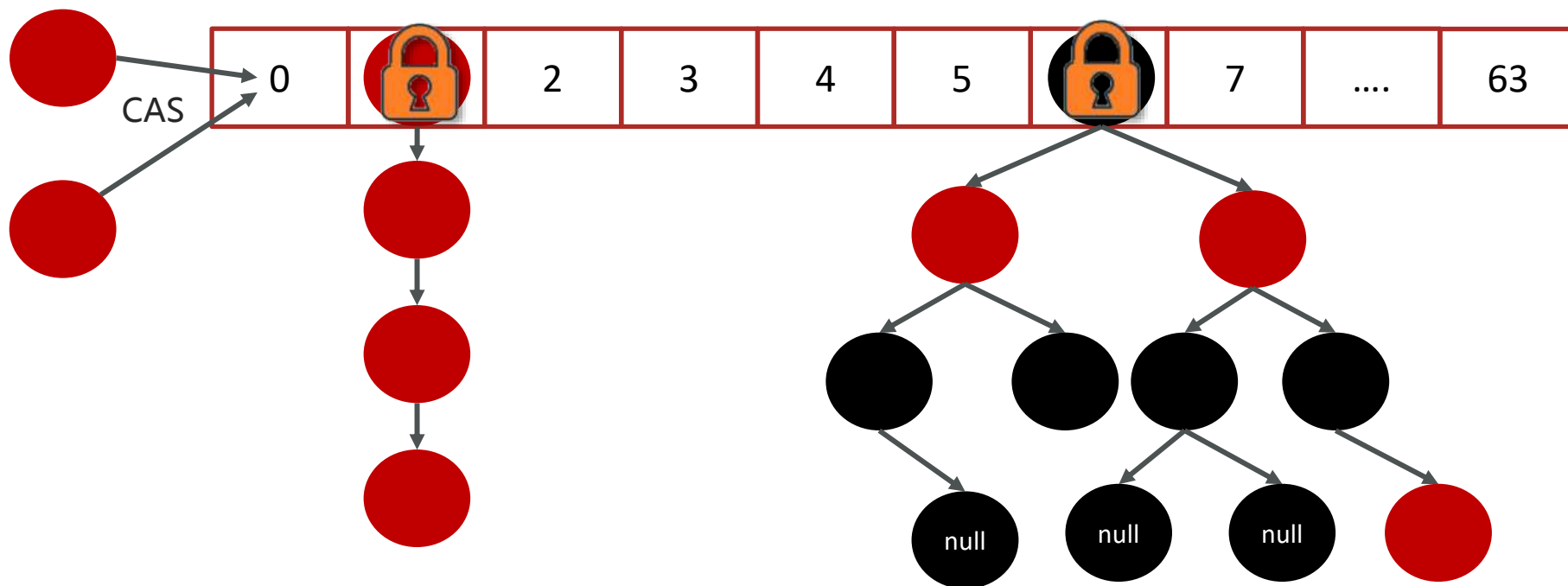


JDK1.8中ConcurrentHashMap

在JDK1.8中，放弃了Segment臃肿的设计，数据结构跟HashMap的数据结构是一样的：数组+红黑树+链表

采用 CAS + Synchronized来保证并发安全进行实现

- CAS控制数组节点的添加
- synchronized只锁定当前链表或红黑二叉树的首节点，只要hash不冲突，就不会产生并发的问題，效率得到提升





总结

聊一下ConcurrentHashMap

1. 底层数据结构：

- JDK1.7底层采用分段的数组+链表实现
- JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树

2. 加锁的方式

- JDK1.7采用Segment分段锁，底层使用的是ReentrantLock
- JDK1.8采用CAS添加新节点，采用synchronized锁定链表或红黑二叉树的首节点，相对Segment分段锁粒度更细，性能更好

导致并发程序出现问题的根本原因是什么

(Java程序中怎么保证多线程的执行安全)

难易程度： ★★★★★

出现频率： ★★★★★

导致并发程序出现问题的根本原因是什么

Java并发编程三大特性

- 原子性
- 可见性
- 有序性

导致并发程序出现问题的根本原因是什么

原子性：一个线程在CPU中操作不可暂停，也不可中断，要不执行完成，要不不执行

```
int ticketNum = 10;
public void getTicket(){
    if(ticketNum <= 0){
        return ;
    }
    System.out.println(Thread.currentThread().getName()+"抢到一张票,剩余:"+ticketNum);
    // 非原子性操作
    ticketNum--;
}

public static void main(String[] args) {
    TicketDemo demo = new TicketDemo();
    for(int i=0;i<20;i++){
        new Thread(demo::getTicket).start();
    }
}
```

不是原子操作，怎么保证原子操作呢？

导致并发程序出现问题的根本原因是什么

1.synchronized : 同步加锁

2.JUC里面的lock : 加锁


```
int ticketNum = 10;
public synchronized void getTicket(){
    if(ticketNum <= 0){
        return ;
    }
    System.out.println(Thread.currentThread().getName()+"抢到一张票,剩余:"+ticketNum);
    // 非原子性操作
    ticketNum--;
}

public static void main(String[] args) {
    TicketDemo demo = new TicketDemo();
    for(int i=0;i<20;i++){
        new Thread(demo::getTicket).start();
    }
}
```

导致并发程序出现问题的根本原因是什么

内存可见性：让一个线程对共享变量的修改对另一个线程可见

```
public class VolatileDemo {  
  
    private static boolean flag = false;  
    public static void main(String[] args) throws InterruptedException {  
        new Thread()->{  
            while(!flag){  
            }  
            System.out.println("第一个线程执行完毕...");  
        }.start();  
        Thread.sleep(100);  
        new Thread()->{  
            flag = true;  
            System.out.println("第二线程执行完毕...");  
        }.start();  
    }  
}
```



解决方案


- synchronized
- volatile
- LOCK

导致并发程序出现问题的根本原因是什么

有序性

指令重排：处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的

```
int x;  
int y;  
  
@Actor  
public void actor1() {  
    x = 1;  
    y = 1;  
}  
  
@Actor  
public void actor2(Il_Result r) {  
    r.r1 = y;  
    r.r2 = x;  
}
```



解决方案

volatile



总结

导致并发程序出现问题的根本原因是什么

- 1.原子性 `synchronized`、`lock`
- 2.内存可见性 `volatile`、`synchronized`、`lock`
- 3.有序性 `volatile`

线程的基础知识

完成

线程与进程的区别

并行与并发的区别

线程创建的方式有哪些

Runnable 和 Callable 有什么区别

线程包括哪些状态，状态之间是如何变化的

在Java中wait和sleep方法的不同

新建三个线程，如何保证它们按顺序执行

notify()和 notifyAll()有什么区别

线程的 run()和 start()有什么区别

如何停止一个正在运行的线程

线程中并发安全

完成

synchronized关键字的底层原理

你谈谈 JMM (Java 内存模型)

CAS 你知道吗

什么是AQS

ReentrantLock的实现原理

synchronized和Lock有什么区别

死锁产生的条件是什么

如何进行死锁诊断

请谈谈你对 volatile 的理解

聊一下ConcurrentHashMap

导致并发程序出现问题的根本原因是什么

线程池

说一下线程池的核心参数（线程池的执行原理知道嘛）

线程池中有哪些常见的阻塞队列

如何确定核心线程数

线程池的种类有哪些

为什么不建议用Executors创建线程池

使用场景

线程池使用场景(你们项目中哪里用到了线程池)

如何控制某个方法允许并发访问线程的数量

谈谈你对ThreadLocal的理解

说一下线程池的核心参数

线程池的执行原理知道嘛

难易程度： ★★★★★

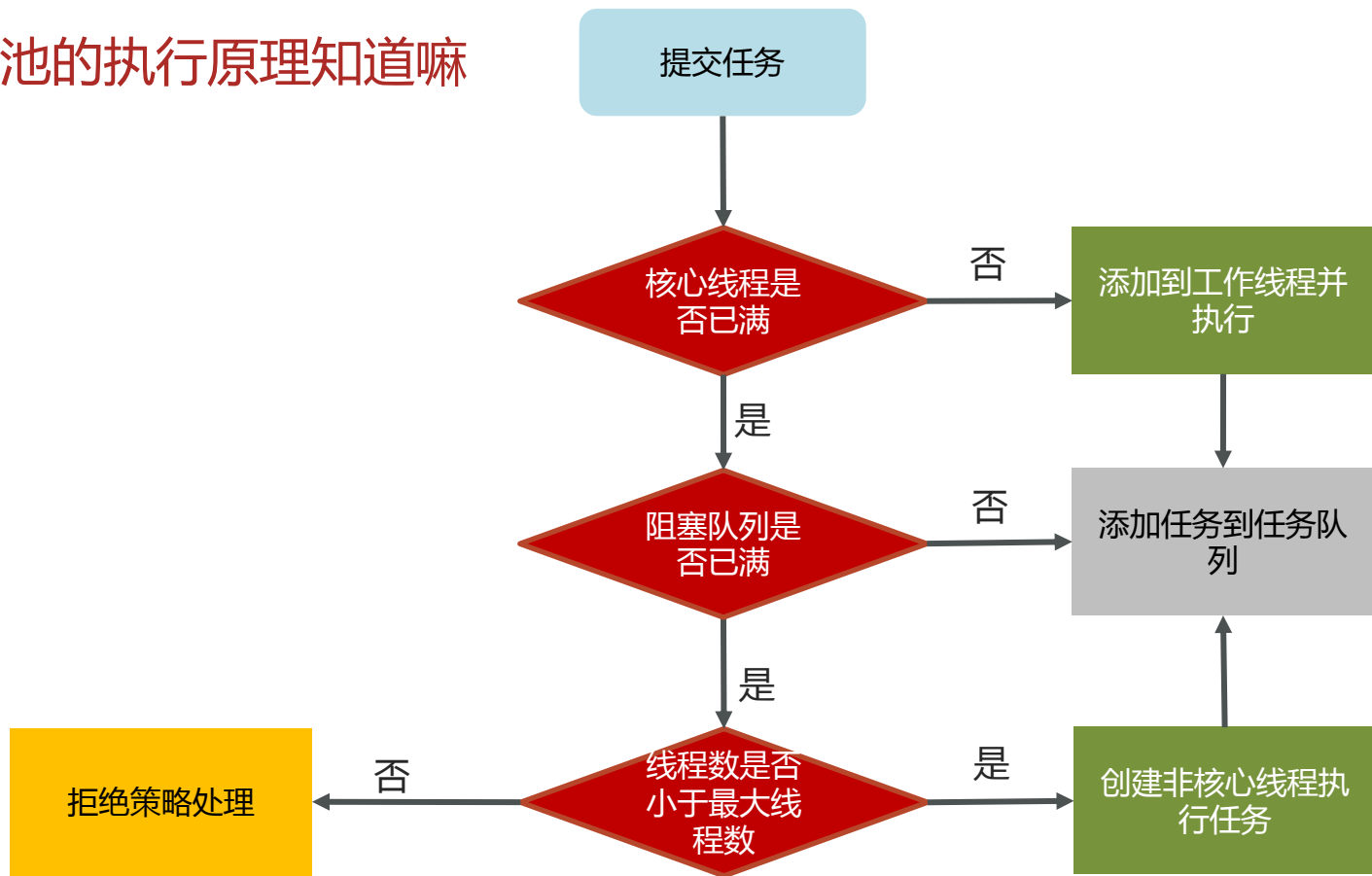
出现频率： ★★★★★

说一下线程池的核心参数

```
public ThreadPoolExecutor(int corePoolSize,  
                           int maximumPoolSize,  
                           long keepAliveTime,  
                           TimeUnit unit,  
                           BlockingQueue<Runnable> workQueue,  
                           ThreadFactory threadFactory,  
                           RejectedExecutionHandler handler)
```

- corePoolSize 核心线程数目
- maximumPoolSize 最大线程数目 = (核心线程+救急线程的最大数目)
- keepAliveTime 生存时间 - 救急线程的生存时间，生存时间内没有新任务，此线程资源会释放
- unit 时间单位 - 救急线程的生存时间单位，如秒、毫秒等
- workQueue - 当没有空闲核心线程时，新来任务会加入到此队列排队，队列满会创建救急线程执行任务
- threadFactory 线程工厂 - 可以定制线程对象的创建，例如设置线程名字、是否是守护线程等
- handler 拒绝策略 - 当所有线程都在繁忙，workQueue 也放满时，会触发拒绝策略

线程池的执行原理知道嘛



如果核心或临时线程执行完成任务后会检查阻塞队列中是否有需要执行的线程，如果有，则使用非核心线程执行任务

- 1.AbortPolicy：直接抛出异常，默认策略；
- 2.CallerRunsPolicy：用调用者所在的线程来执行任务；
- 3.DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
- 4.DiscardPolicy：直接丢弃任务；

线程池中有哪些常见的阻塞队列

难易程度： ★★★★★

出现频率： ★★★★★

线程池中有哪些常见的阻塞队列

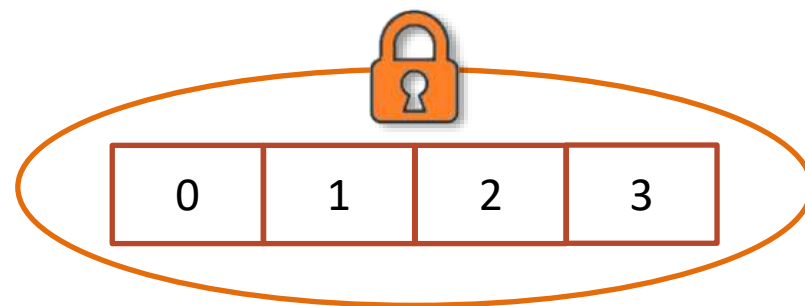
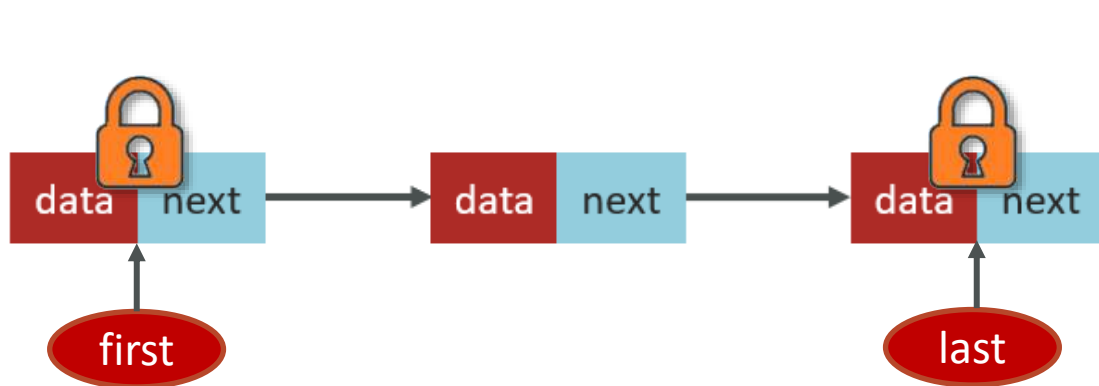
workQueue - 当没有空闲核心线程时，新来任务会加入到此队列排队，队列满会创建救急线程执行任务

1. `ArrayBlockingQueue`：基于数组结构的有界阻塞队列，FIFO。
2. `LinkedBlockingQueue`：基于链表结构的有界阻塞队列，FIFO。
3. `DelayedWorkQueue`：是一个优先级队列，它可以保证每次出队的任务都是当前队列中执行时间最靠前的
4. `SynchronousQueue`：不存储元素的阻塞队列，每个插入操作都必须等待一个移出操作。

线程池中有哪些常见的阻塞队列

ArrayBlockingQueue的LinkedBlockingQueue区别

LinkedBlockingQueue	ArrayBlockingQueue
默认无界，支持有界	强制有界
底层是链表	底层是数组
是懒惰的，创建节点的时候添加数据	提前初始化 Node 数组
入队会生成新 Node	Node需要是提前创建好的
两把锁（头尾）	一把锁



如何确定核心线程数

难易程度： ★★★★★

出现频率： ★★★★★

如何确定核心线程数

- IO密集型任务

一般来说：文件读写、DB读写、网络请求等

核心线程数大小设置为 $2N+1$

- CPU密集型任务

一般来说：计算型代码、Bitmap转换、Gson转换等

核心线程数大小设置为 $N+1$

```
public static void main(String[] args) {  
    //查看机器的CPU核数  
    System.out.println(Runtime.getRuntime().availableProcessors());  
}
```

查看机器的CPU核数

如何确定核心线程数

参考回答：

- ① 高并发、任务执行时间短 \rightarrow (CPU核数+1)，减少线程上下文的切换
- ② 并发不高、任务执行时间长
 - IO密集型的任务 \rightarrow (CPU核数 * 2 + 1)
 - 计算密集型任务 \rightarrow (CPU核数+1)
- ③ 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，[设置参考（2）](#)

线程池的种类有哪些

难易程度： ★★★★★

出现频率： ★★★★★

线程池的种类有哪些

在java.util.concurrent.Executors类中提供了大量创建连接池的静态方法，常见就有四种

1. 创建使用固定线程数的线程池

```
public static ExecutorService newFixedThreadPool(int nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads,  
        0L, TimeUnit.MILLISECONDS,  
        new LinkedBlockingQueue<Runnable>());  
}
```

- 核心线程数与最大线程数一样，没有救急线程
- 阻塞队列是LinkedBlockingQueue，最大容量为Integer.MAX_VALUE

适用于任务量已知，相对耗时的任务

线程池的种类有哪些

2. 单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO)执行

```
public static ExecutorService newSingleThreadExecutor() {  
    return new FinalizableDelegatedExecutorService  
        (new ThreadPoolExecutor(1, 1,  
                                0L, TimeUnit.MILLISECONDS,  
                                new LinkedBlockingQueue<Runnable>()));  
}
```

- 核心线程数和最大线程数都是1
- 阻塞队列是LinkedBlockingQueue，最大容量为Integer.MAX_VALUE

适用于按照顺序执行的任务

线程池的种类有哪些

3. 可缓存线程池

```
public static ExecutorService newCachedThreadPool() {  
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,  
        60L, TimeUnit.SECONDS,  
        new SynchronousQueue<Runnable>());  
}
```

- 核心线程数为0
- 最大线程数是Integer.MAX_VALUE
- 阻塞队列为SynchronousQueue:不存储元素的阻塞队列，每个插入操作都必须等待一个移出操作。

适合任务数比较密集，但每个任务执行时间较短的情况

线程池的种类有哪些

4. 提供了“延迟”和“周期执行”功能的ThreadPoolExecutor。

```
public ScheduledThreadPoolExecutor(int corePoolSize) {  
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue());  
}  
public ScheduledThreadPoolExecutor(int corePoolSize,  
    ThreadFactory threadFactory) {  
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue(), threadFactory);  
}  
public ScheduledThreadPoolExecutor(int corePoolSize,  
    RejectedExecutionHandler handler) {  
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue(), handler);  
}  
public ScheduledThreadPoolExecutor(int corePoolSize,  
    ThreadFactory threadFactory,  
    RejectedExecutionHandler handler) {  
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new DelayedWorkQueue(), threadFactory, handler);  
}
```

总结

线程池的种类有哪些

- ① newFixedThreadPool：创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待
- ② newSingleThreadExecutor：创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO)执行
- ③ newCachedThreadPool：创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程
- ④ newScheduledThreadPool：可以执行延迟任务的线程池，支持定时及周期性任务执行

为什么不建议用Executors创建线程池

难易程度： ★★★★★

出现频率： ★★★★★

为什么不建议用Executors创建线程池

参考阿里开发手册《Java开发手册-嵩山版》

【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors 返回的线程池对象的弊端如下：

1) **FixedThreadPool** 和 **SingleThreadPool**：

允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。

2) **CachedThreadPool**：

允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

线程的基础知识

完成

线程与进程的区别

并行与并发的区别

线程创建的方式有哪些

Runnable 和 Callable 有什么区别

线程包括哪些状态，状态之间是如何变化的

在Java中wait和sleep方法的不同

新建三个线程，如何保证它们按顺序执行

notify()和 notifyAll()有什么区别

线程的 run()和 start()有什么区别

如何停止一个正在运行的线程

线程中并发安全

完成

synchronized关键字的底层原理

你谈谈 JMM (Java 内存模型)

CAS 你知道吗

什么是AQS

ReentrantLock的实现原理

synchronized和Lock有什么区别

死锁产生的条件是什么

如何进行死锁诊断

请谈谈你对 volatile 的理解

聊一下ConcurrentHashMap

导致并发程序出现问题的根本原因是什么

线程池

完成

说一下线程池的核心参数 (线程池的执行原理知道嘛)

线程池中有哪些常见的阻塞队列

如何确定核心线程数

线程池的种类有哪些

为什么不建议用Executors创建线程池

使用场景

线程池使用场景(你们项目中哪里用到了线程池)

如何控制某个方法允许并发访问线程的数量

谈谈你对ThreadLocal的理解

线程池使用场景（CountDownLatch、Future） （你们项目哪里用到了多线程）

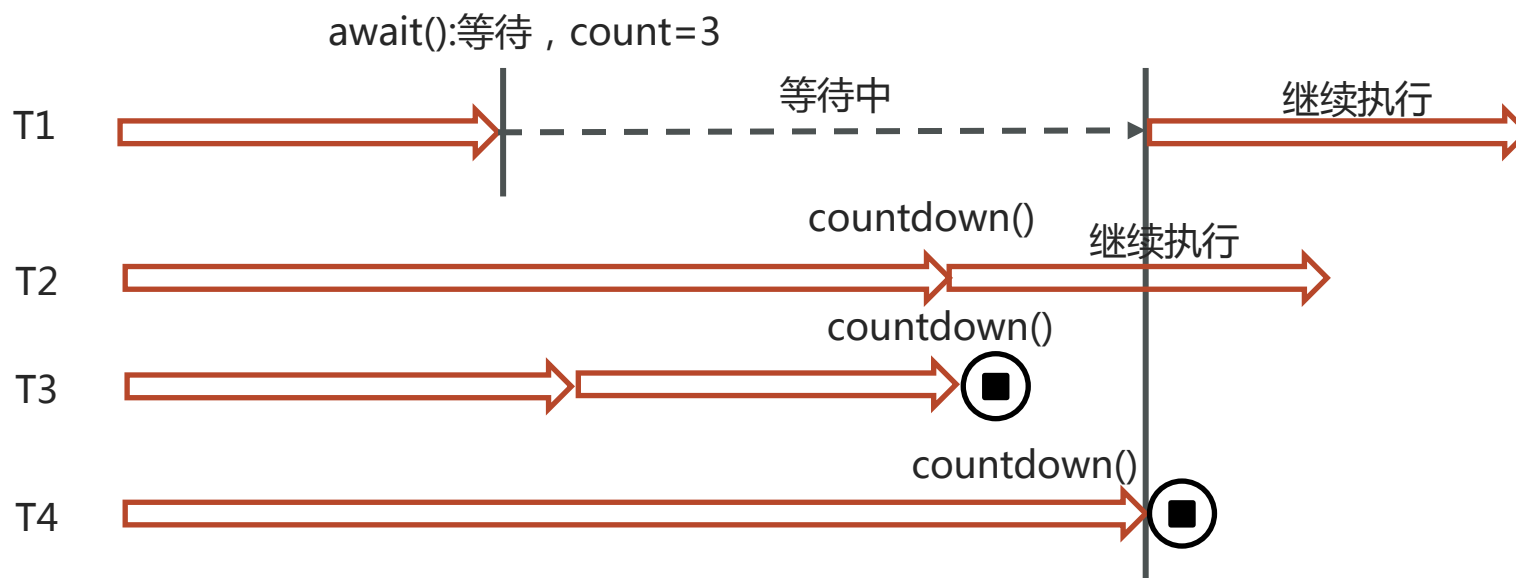
难易程度： ★ ★ ★ ☆ ☆

出现频率： ★ ★ ★ ★ ☆

CountDownLatch

CountDownLatch（闭锁/倒计时锁）用来进行线程同步协作，等待所有线程完成倒计时（一个或者多个线程，等待其他多个线程完成某件事情之后才能执行）

- 其中构造参数用来初始化等待计数值
- `await()` 用来等待计数归零
- `countDown()` 用来让计数减一

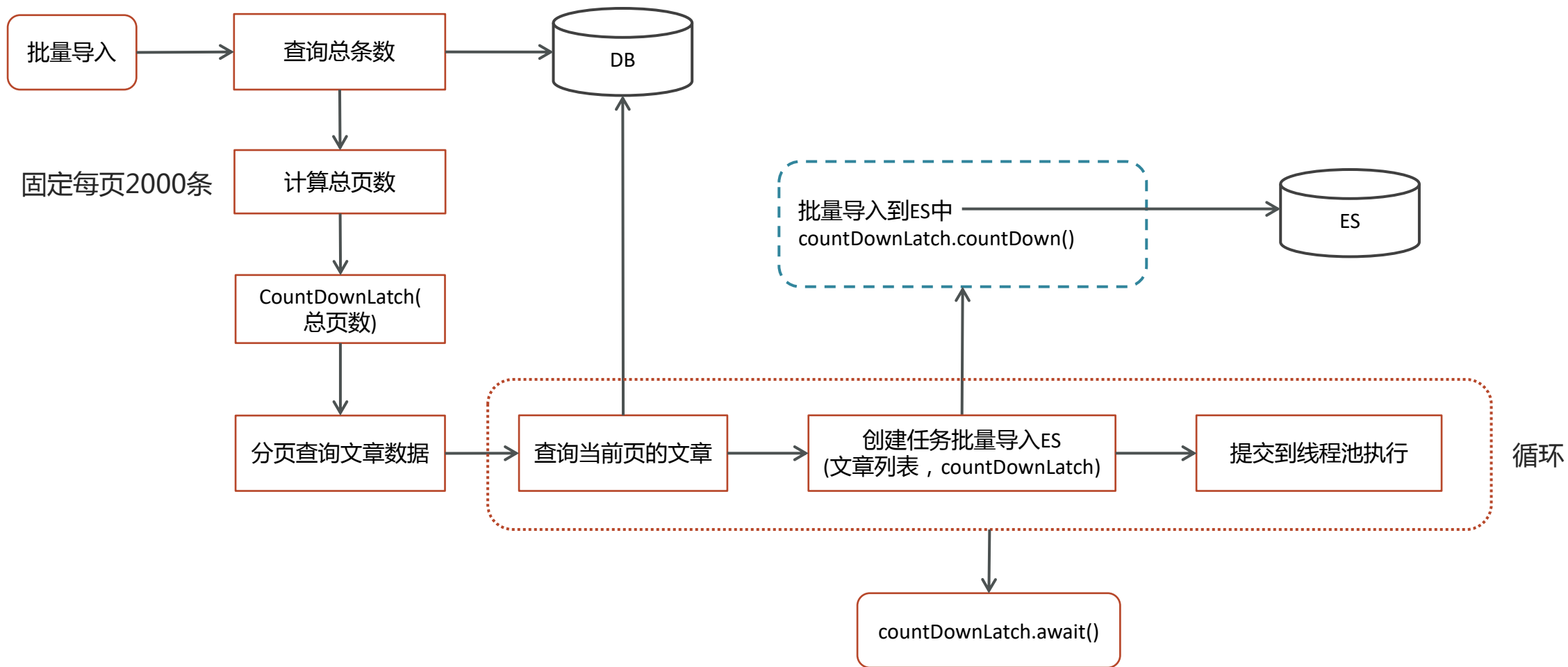


多线程使用场景一（es数据批量导入）

在我们项目上线之前，我们需要把数据库中的数据一次性的同步到es索引库中，但是当时的数据好像是1000万左右，一次性读取数据肯定不行（oom异常），当时我就想到可以使用线程池的方式导入，利用CountDownLatch来控制，就能避免一次性加载过多，防止内存溢出

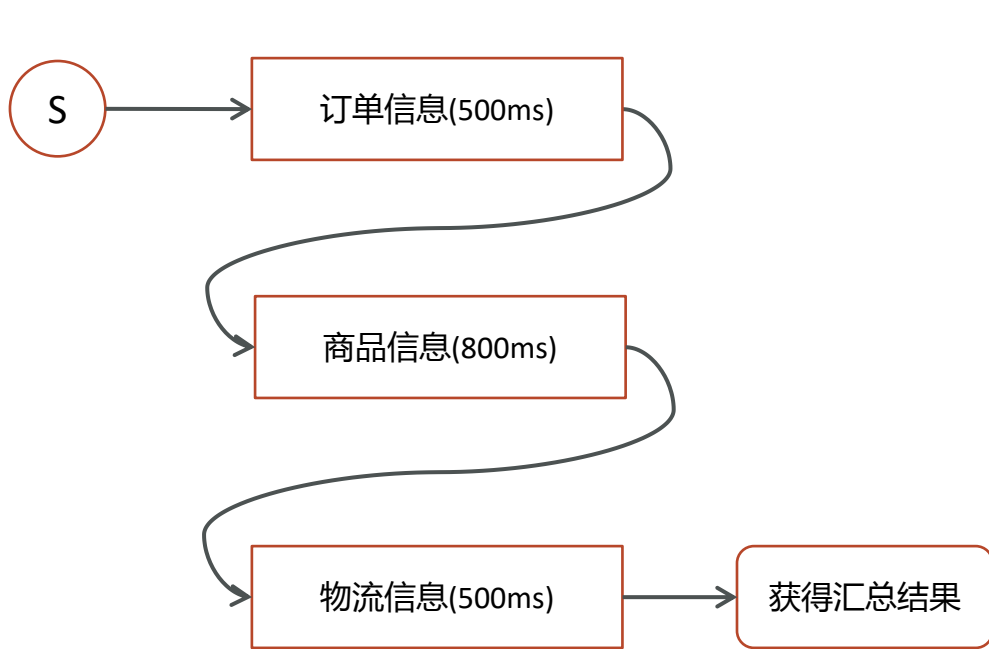


多线程使用场景一（es数据批量导入）

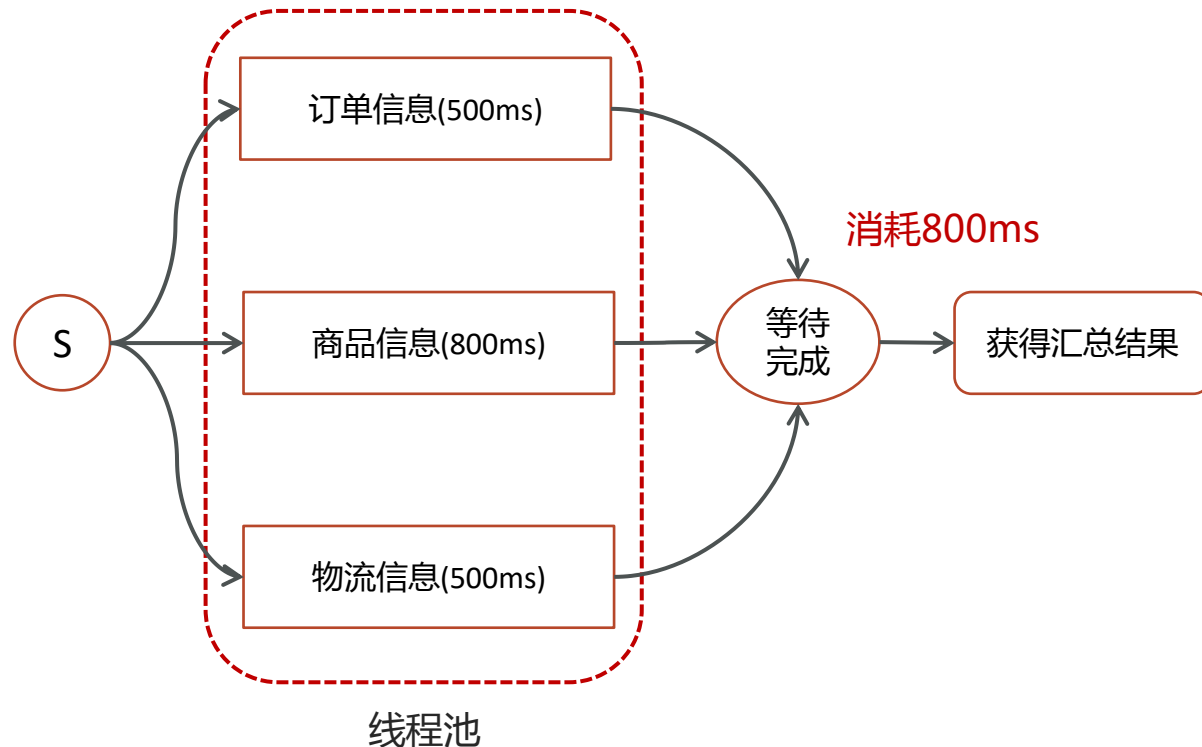


多线程使用场景二（数据汇总）

在一个电商网站中，用户下单之后，需要查询数据，数据包含了三部分：订单信息、包含的商品、物流信息；这三块信息都在不同的微服务中进行实现的，我们如何完成这个业务呢？



共消耗：1800ms



多线程使用场景二（数据汇总）

- 在实际开发的过程中，难免需要调用多个接口来汇总数据，如果所有接口（或部分接口）的没有依赖关系，就可以使用线程池+future来提升性能
- 报表汇总



多线程使用场景三（异步调用）



异步保存

在**线程池**中获取一个
新的线程执行





总结

你们项目哪里用到了多线程

- **批量导入**：使用了线程池+CountDownLatch批量把数据库中的数据导入到了ES(任意)中，避免OOM
- **数据汇总**：调用多个接口来汇总数据，如果所有接口（或部分接口）的没有依赖关系，就可以使用线程池+future来提升性能
- **异步线程（线程池）**：为了避免下一级方法影响上一级方法（性能考虑），可使用异步线程调用下一个方法（不需要下一级方法返回值），可以提升方法响应时间

如何控制某个方法允许并发访问线程的数量

难易程度： ★★★★★

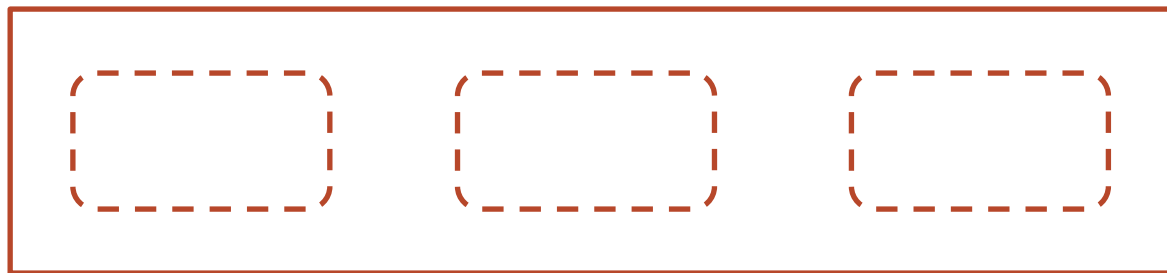
出现频率： ★★★★★

如何控制某个方法允许并发访问线程的数量

Semaphore ['semə,for] 信号量，是JUC包下的一个工具类，底层是AQS，我们可以通过其限制执行的线程数量

使用场景：

通常用于那些资源有明确访问数量限制的场景，常用于限流。



停车场共3个车位

比亚迪

如何控制某个方法允许并发访问线程的数量

Semaphore使用步骤

- 创建Semaphore对象，可以给一个容量
- semaphore.acquire()：请求一个信号量，这时候的信号量个数-1（一旦没有可使用的信号量，也即信号量个数变为负数时，再次请求的时候就会阻塞，直到其他线程释放了信号量）
- semaphore.release()：释放一个信号量，此时信号量个数+1

```
// 1. 创建 semaphore 对象
Semaphore semaphore = new Semaphore(3);
// 2. 10个线程同时运行
for (int i = 0; i < 10; i++) {
    new Thread() -> {
        try {
            // 3. 获取许可
            semaphore.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
            System.out.println("running...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("end...");
        } finally {
            // 4. 释放许可
            semaphore.release();
        }
    }.start();
}
```



总结

如何控制某个方法允许并发访问线程的数量

在多线程中提供了一个工具类Semaphore，信号量。在并发的情况下，可以控制方法的访问量

1. 创建Semaphore对象，可以给一个容量
2. acquire()可以请求一个信号量，这时候的信号量个数-1
3. release()释放一个信号量，此时信号量个数+1

谈谈你对ThreadLocal的理解

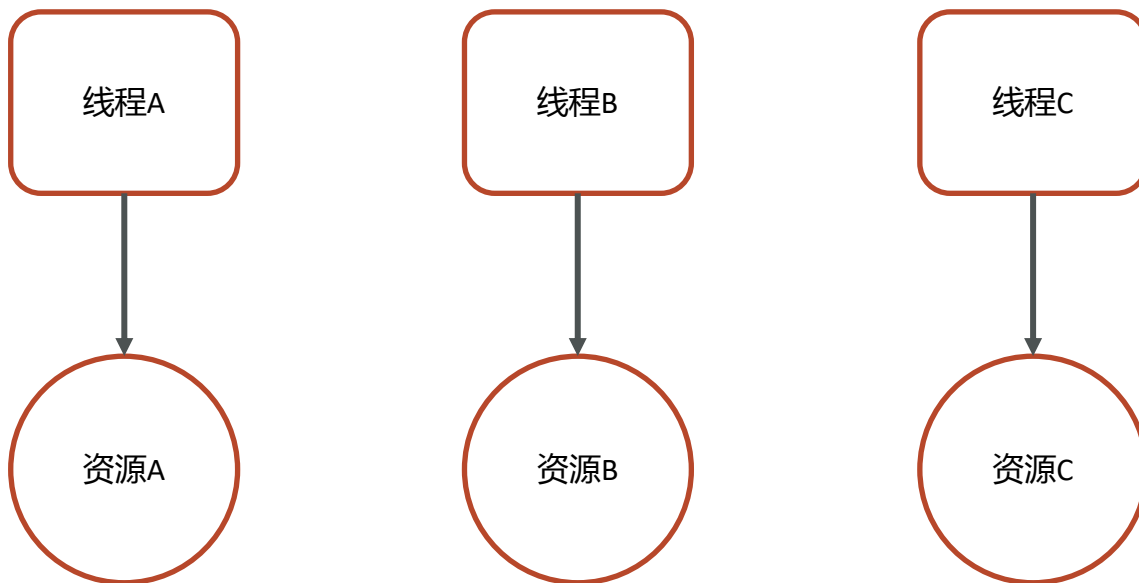
难易程度： ★★★★★

出现频率： ★★★★★

ThreadLocal概述

ThreadLocal是多线程中对于解决线程安全的一个操作类，它会为每个线程都分配一个独立的线程副本从而解决了变量并发访问冲突的问题。ThreadLocal 同时实现了线程内的资源共享

案例：使用JDBC操作数据库时，会将每一个线程的Connection放入各自的ThreadLocal中，从而保证每个线程都在各自的 Connection 上进行数据库的操作，避免A线程关闭了B线程的连接。



ThreadLocal基本使用

- set(value) 设置值
- get() 获取值
- remove() 清除值

```
static ThreadLocal<String> threadLocal = new ThreadLocal<>();

public static void main(String[] args) {
    new Thread(() -> {
        String name = Thread.currentThread().getName();
        threadLocal.set("itcast");
        print(name);
        System.out.println(name + "-after remove : " + threadLocal.get());
    }, "t1").start();
    new Thread(() -> {
        String name = Thread.currentThread().getName();
        threadLocal.set("itheima");
        print(name);
        System.out.println(name + "-after remove : " + threadLocal.get());
    }, "t2").start();
}

static void print(String str) {
    //打印当前线程中本地内存中本地变量的值
    System.out.println(str + " : " + threadLocal.get());
    //清除本地内存中的本地变量
    threadLocal.remove();
}
```

ThreadLocal的实现原理&源码解析

ThreadLocal本质来说就是一个线程内部存储类，从而让多个线程只操作自己内部的值，从而实现线程数据隔离

```
ThreadLocal
> SuppliedThreadLocal
> ThreadLocalMap
  ThreadLocal()
  nextHashCode(): int
  initialValue(): T
  withInitial(Supplier<? extends S>): ThreadLocal<S>
  get(): T
  isPresent(): boolean
  setInitialValue(): T
  set(T): void
  remove(): void
  getMap(Thread): ThreadLocalMap
  createMap(Thread, T): void
  createInheritedMap(ThreadLocalMap): ThreadLocalMap
  childValue(T): T
  threadLocalHashCode: int = nextHashCode()
  nextHashCode: AtomicInteger = new AtomicInteger()
  HASH_INCREMENT: int = 0x61c88647
```

每个线程持有一个
ThreadLocalMap对象

ThreadLocalMap
中为每一个线程都
维护了一个数组
table(存储数据)

```
ThreadLocalMap
> Entry
  ThreadLocalMap(ThreadLocal<?>, Object)
  ThreadLocalMap(ThreadLocalMap)
  setThreshold(int): void
  nextIndex(int, int): int
  prevIndex(int, int): int
  getEntry(ThreadLocal<?>): Entry
  getEntryAfterMiss(ThreadLocal<?>, int, Entry): Entry
  set(ThreadLocal<?>, Object): void
  remove(ThreadLocal<?>): void
  replaceStaleEntry(ThreadLocal<?>, Object, int): void
  expungeStaleEntry(int): int
  cleanSomeSlots(int, int): boolean
  rehash(): void
  resize(): void
  expungeStaleEntries(): void
  INITIAL_CAPACITY: int = 16
  table: Entry[]
  size: int = 0
  threshold: int
```

ThreadLocal的实现原理&源码解析

set方法

```
public void set(T value) {  
    //获取当前线程对象  
    Thread t = Thread.currentThread();  
    //根据当前线程对象，获取ThreadLocal中的ThreadLocalMap  
    ThreadLocalMap map = getMap(t);  
    //如果map存在  
    if (map != null)  
        //执行map中的set方法，进行数据存储  
        map.set(this, value);  
    else  
        //否则创建ThreadLocalMap，并存值  
        createMap(t, value);  
}
```

```
void createMap(Thread t, T firstValue) {  
    t.threadLocals = new ThreadLocalMap(this, firstValue);  
}
```



```
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {  
  
    //内部成员数组，INITIAL_CAPACITY值为16的常量  
    table = new Entry[INITIAL_CAPACITY];  
  
    //位运算，结果与取模相同，计算出需要存放的位置  
    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);  
    table[i] = new Entry(firstKey, firstValue);  
    size = 1;  
    setThreshold(INITIAL_CAPACITY);  
}
```

ThreadLocal的实现原理&源码解析

get方法/remove方法

```
public T get() {  
    Thread t = Thread.currentThread();  
    //根据线程对象，获取对应的ThreadLocalMap  
    ThreadLocalMap map = getMap(t);  
    if (map != null) {  
        //获取ThreadLocalMap中对应的Entry对象  
        ThreadLocalMap.Entry e = map.getEntry(this);  
        if (e != null) {  
            @SuppressWarnings("unchecked")  
            //获取Entry中的value  
            T result = (T)e.value;  
            return result;  
        }  
    }  
    return setInitialValue();  
}
```



```
private Entry getEntry(ThreadLocal<?> key) {  
    //确定数组下标位置  
    int i = key.threadLocalHashCode & (table.length - 1);  
    //得到该位置上的Entry  
    Entry e = table[i];  
    if (e != null && e.get() == key)  
        return e;  
    else  
        return getEntryAfterMiss(key, i, e);  
}
```

面试官：你对ThreadLocal理解的挺深的，你知道ThreadLocal的内存泄露问题吗？

ThreadLocal-内存泄露问题

Java对象中的四种引用类型：强引用、软引用、弱引用、虚引用

- 强引用：最为普通的引用方式，表示一个对象处于**有用且必须**的状态，如果一个对象具有强引用，则GC并不会回收它。即便堆中内存不足了，宁可出现OOM，也不会对其进行回收

```
User user = new User();
```

- 弱引用：表示一个对象处于**可能有用且非必须**的状态。在GC线程扫描内存区域时，一旦发现弱引用，就会回收弱引用相关联的对象。对于弱引用的回收，无关内存区域是否足够，一旦发现则会被回收

```
User user = new User();  
WeakReference weakReference = new WeakReference(user);
```

ThreadLocal-内存泄露问题

每一个Thread维护一个ThreadLocalMap，在ThreadLocalMap中的Entry对象继承了WeakReference。其中key为使用弱引用的ThreadLocal实例，value为线程变量的副本

```
static class Entry extends WeakReference<ThreadLocal<?>> {  
    /** The value associated with this ThreadLocal. */  
    Object value;  
  
    Entry(ThreadLocal<?> k, Object v) {  
        super(k);  
        value = v;  
    }  
}
```

弱引用，内存不太够的时候，优先回收

强引用，不会被回收

内存
泄漏

防止内存泄漏：务必remove

总结

谈谈你对ThreadLocal的理解

1. ThreadLocal 可以实现【资源对象】的线程隔离，让每个线程各用各的【资源对象】，避免争用引发的线程安全问题
2. ThreadLocal 同时实现了线程内的资源共享
3. 每个线程内有一个 ThreadLocalMap 类型的成员变量，用来存储资源对象
 - a)调用 set 方法，就是以 ThreadLocal 自己作为 key，资源对象作为 value，放入当前线程的 ThreadLocalMap 集合中
 - b)调用 get 方法，就是以 ThreadLocal 自己作为 key，到当前线程中查找关联的资源值
 - c)调用 remove 方法，就是以 ThreadLocal 自己作为 key，移除当前线程关联的资源值
4. ThreadLocal内存泄漏问题
ThreadLocalMap 中的 key 是弱引用，值为强引用；key 会被GC 释放内存，关联 value 的内存并不会释放。建议主动 remove 释放 key，value



传智教育旗下高端IT教育品牌