文档说明

> 在文档中对所有的面试题都进行了**难易程度**和**出现频率**的等级说明
>
> 星数越多代表权重越大，最多五颗星（☆☆☆☆☆） 最少一颗星（☆）

Java多线程相关面试题

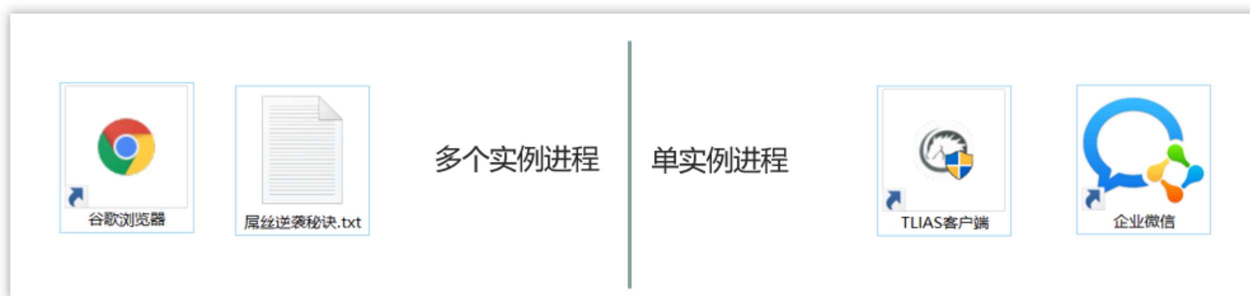
1.线程的基础知识

1.1 线程和进程的区别？

> 难易程度：☆☆
>
> 出现频率：☆☆☆

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。

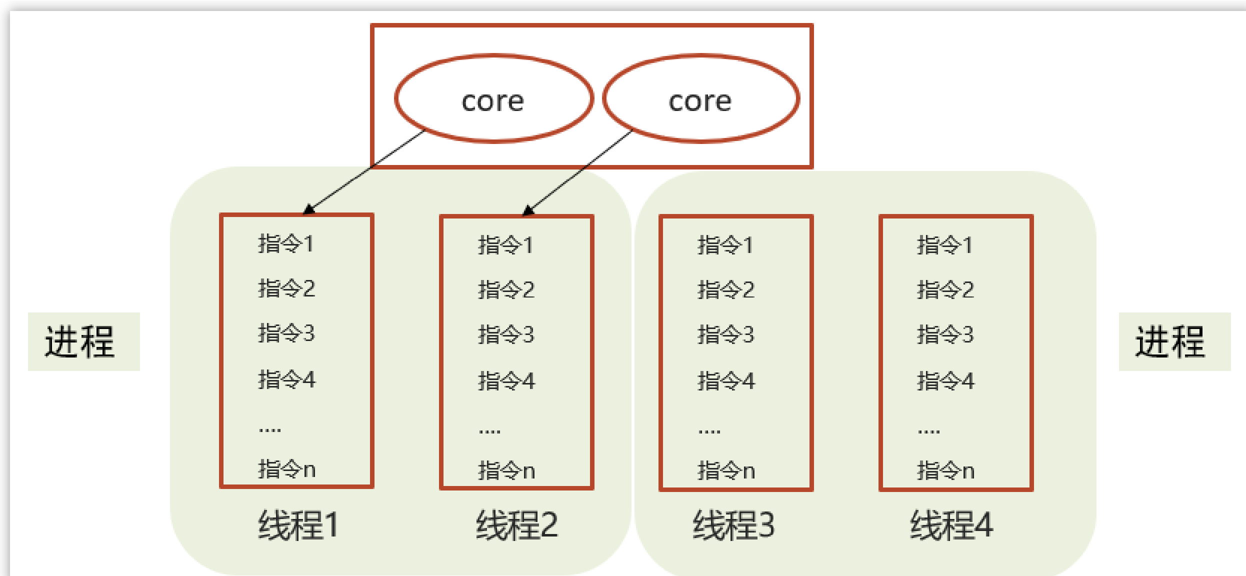
当一个程序被运行，从磁盘加载这个程序的代码至内存，这时就开启了一个进程。



一个进程之内可以分为一到多个线程。

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

Java 中，线程作为最小调度单位，进程作为资源分配的最小单位。在 windows 中进程是不活动的，只是作为线程的容器



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)

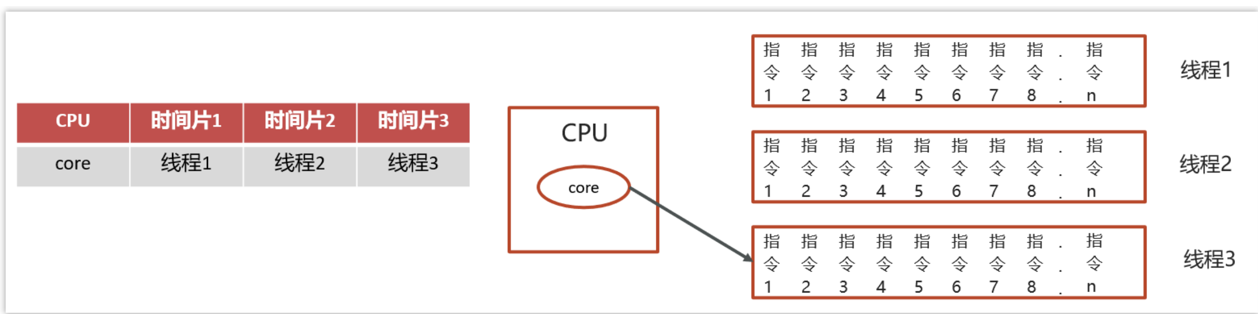
1.2 并行和并发有什么区别？

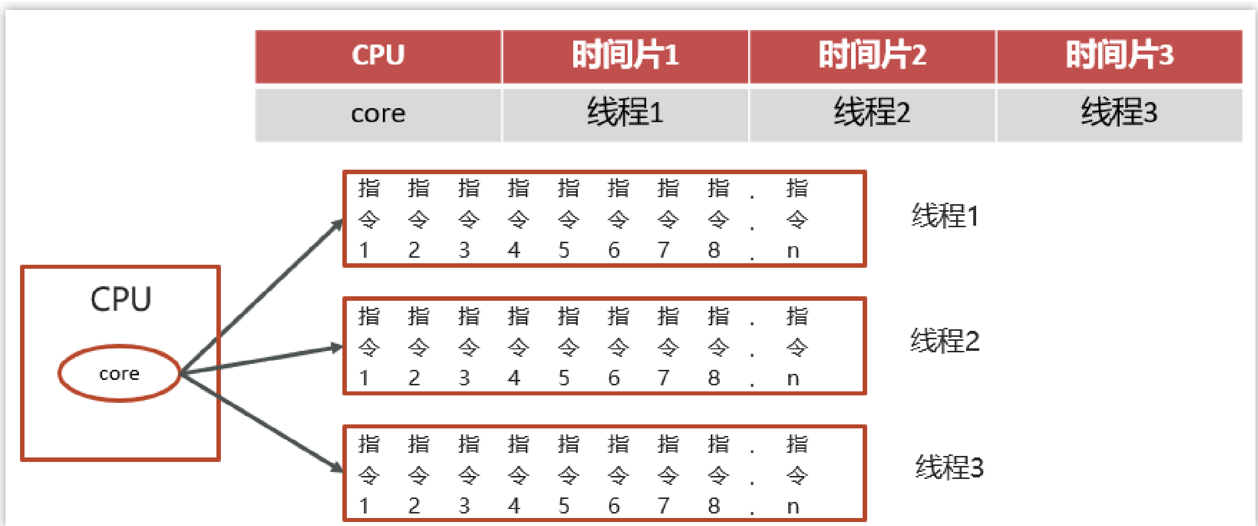
> 难易程度：☆
>
> 出现频率：☆

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

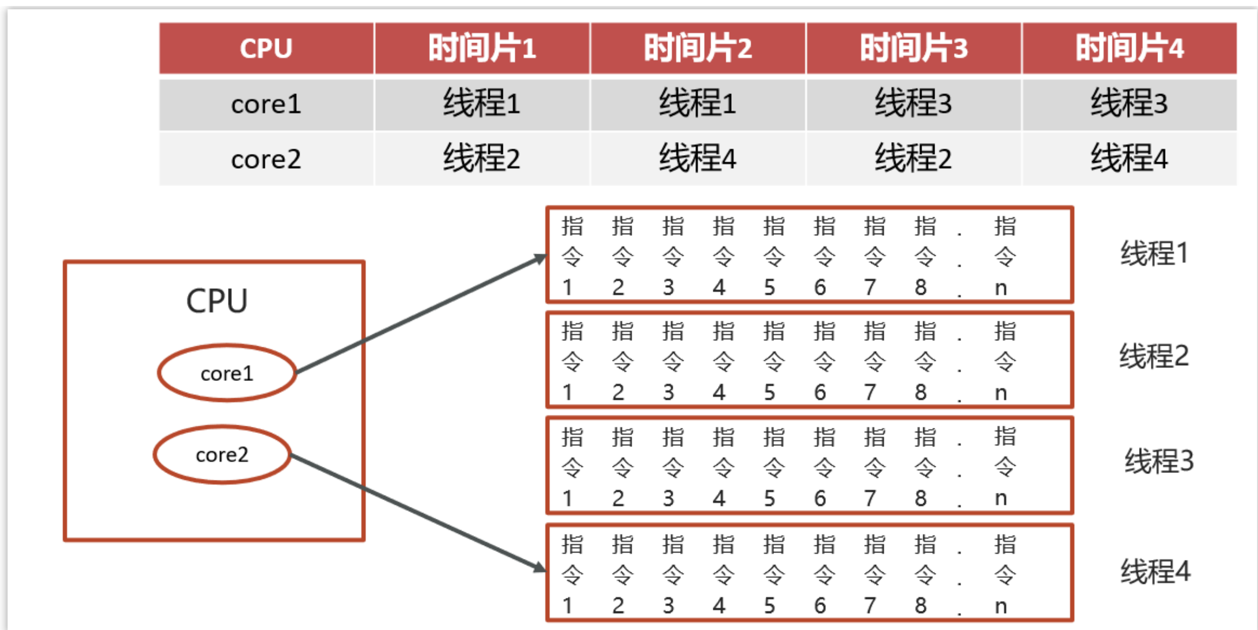
一般会将这种线程轮流使用CPU的做法称为并发（concurrent）





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。



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

> 举例：
>
> - 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

1.3 创建线程的四种方式

> 难易程度：☆☆
>
> 出现频率：☆☆☆☆

参考回答：

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

详细创建方式参考下面代码：

① 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();

    }

}
```

② 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();

    }

}
```

③ 实现Callable接口

```
public class MyCallable implements Callable<String> {

    @Override
    public String call() throws Exception {
        System.out.println("MyCallable...call...");
        return "OK";
```

```
    }

    public static void main(String[] args) throws
ExecutionException, InterruptedException {

        // 创建MyCallable对象
        MyCallable mc = new MyCallable() ;

        // 创建F
        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);

    }

}
```

④ 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
```

```java
        ExecutorService threadPool =
Executors.newFixedThreadPool(3);
        threadPool.submit(new MyExecutors()) ;

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

    }

}
```

1.4 runnable 和 callable 有什么区别

> 难易程度：☆☆
>
> 出现频率：☆☆☆

参考回答：

1. Runnable 接口run方法没有返回值；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
2. Callalbe接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。
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

1.5 线程的 run()和 start()有什么区别？

> 难易程度：☆☆
>
> 出现频率：☆☆

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。

1.6 线程包括哪些状态，状态之间是如何变化的

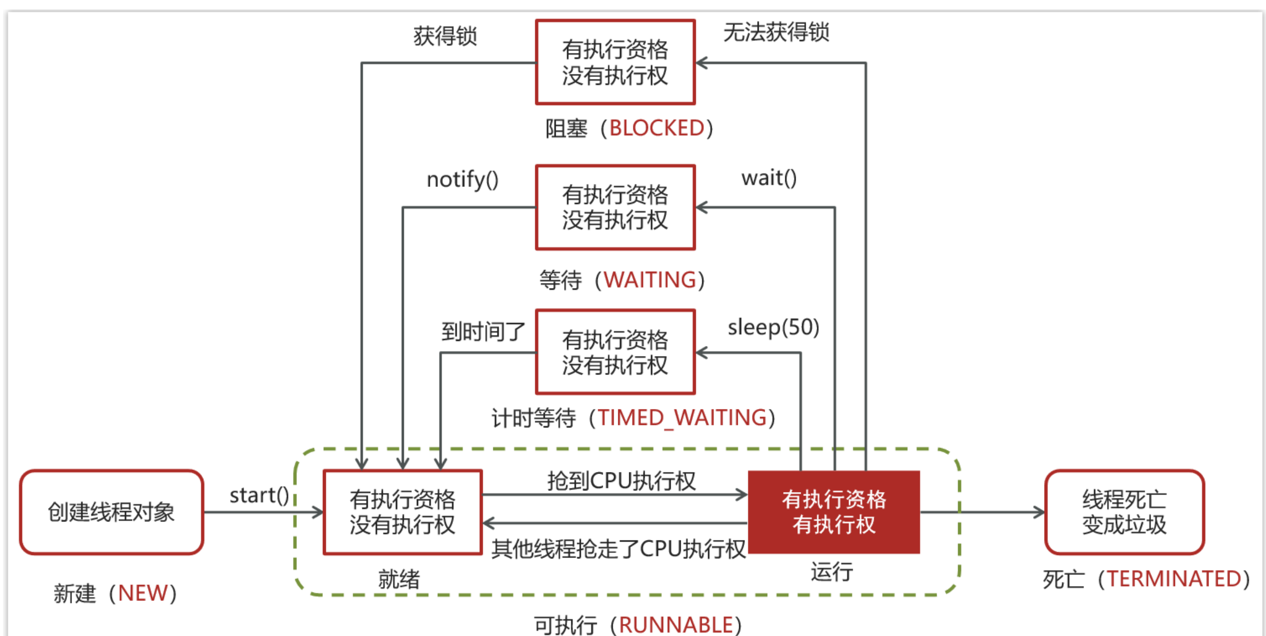
> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

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。处于可运行状态的线程正在 Java 虚拟机中执行，但它可能正在等待来自         * 操作系统的其他资源，例如处理器。
         */
        RUNNABLE,

        /**
         * 线程阻塞等待监视器锁的线程状态。处于阻塞状态的线程正在等待监视器锁进入同步块/方法或在调          * 用Object.wait后重新进入同步块/方法。
         */
        BLOCKED,

        /**
         * 等待线程的线程状态。由于调用以下方法之一，线程处于等待状态：
        * Object.wait没有超时
         * 没有超时的Thread.join
         * LockSupport.park
         * 处于等待状态的线程正在等待另一个线程执行特定操作。
         * 例如，一个对对象调用Object.wait()的线程正在等待另一个线程对该对象调用Object.notify()         * 或Object.notifyAll() 。已调用Thread.join()的线程正在等待指定线程终止。
         */
        WAITING,

        /**
```

```
         * 具有指定等待时间的等待线程的线程状态。由于以指定的正等待时间调用以下方法之一，线程处于定         * 时等待状态：
        * Thread.sleep
        * Object.wait超时
        * Thread.join超时
        * LockSupport.parkNanos
        * LockSupport.parkUntil
         * </ul>
         */
        TIMED_WAITING,

        /**
         * 已终止线程的线程状态。线程已完成执行
         */
        TERMINATED;
    }
```

状态之间是如何变化的



分别是

- 新建
    - 当一个线程对象被创建，但还未调用 start 方法时处于**新建**状态
    - 此时未与操作系统底层线程关联
- 可运行
    - 调用了 start 方法，就会由**新建**进入**可运行**

  - 此时与底层线程关联，由操作系统调度执行

- 终结
  - 线程内代码已经执行完毕，由**可运行**进入**终结**
  - 此时会取消与底层线程关联

- 阻塞
  - 当获取锁失败后，由**可运行**进入 Monitor 的阻塞队列**阻塞**，此时不占用 cpu 时间
  - 当持锁线程释放锁时，会按照一定规则唤醒阻塞队列中的**阻塞**线程，唤醒后的线程进入**可运行**状态

- 等待
  - 当获取锁成功后，但由于条件不满足，调用了 wait() 方法，此时从**可运行**状态释放锁进入 Monitor 等待集合**等待**，同样不占用 cpu 时间
  - 当其它持锁线程调用 notify() 或 notifyAll() 方法，会按照一定规则唤醒等待集合中的**等待**线程，恢复为**可运行**状态

- 有时限等待
  - 当获取锁成功后，但由于条件不满足，调用了 wait(long) 方法，此时从**可运行**状态释放锁进入 Monitor 等待集合进行**有时限等待**，同样不占用 cpu 时间
  - 当其它持锁线程调用 notify() 或 notifyAll() 方法，会按照一定规则唤醒等待集合中的**有时限等待**线程，恢复为**可运行**状态，并重新去竞争锁
  - 如果等待超时，也会从**有时限等待**状态恢复为**可运行**状态，并重新去竞争锁
  - 还有一种情况是调用 sleep(long) 方法也会从**可运行**状态进入**有时限等待**状态，但与 Monitor 无关，不需要主动唤醒，超时时间到自然恢复为**可运行**状态

1.7 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：☆☆☆

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的**join()**方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。

代码举例：

为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成

```
public class JoinTest {

    public static void main(String[] args) {

        // 创建线程对象
        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();

    }

}
```

1.8 notify()和 notifyAll()有什么区别？

> 难易程度：☆☆
>
> 出现频率：☆☆

notifyAll：唤醒所有wait的线程

notify：只随机唤醒一个 wait 线程

```
package com.itheima.basic;

public class WaitNotify {

    static boolean flag = false;
    static Object lock = new Object();

    public static void main(String[] args) {

        Thread t1 = new Thread(() -> {
            synchronized (lock){
                while (!flag){
```

```
 System.out.println(Thread.currentThread().getName()+"...wating...")
;
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

 System.out.println(Thread.currentThread().getName()+"...flag is
true");
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (lock){
                while (!flag){

 System.out.println(Thread.currentThread().getName()+"...wating...")
;
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

 System.out.println(Thread.currentThread().getName()+"...flag is
true");
            }
        });

        Thread t3 = new Thread(() -> {
            synchronized (lock) {
                System.out.println(Thread.currentThread().getName()
+ " hold lock");
                lock.notifyAll();
                flag = true;
                try {
                    Thread.sleep(2000);
```

```java
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t1.start();
        t2.start();
        t3.start();

    }

}
```

1.9 在 java 中 wait 和 sleep 方法的不同？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

参考回答：

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

- 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
- 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
- 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）

代码示例：

```
public class WaitSleepCase {

    static final Object LOCK = new Object();

    public static void main(String[] args) throws
InterruptedException {
        sleeping();
    }

    private static void illegalWait() throws InterruptedException {
        LOCK.wait();
    }

    private static void waiting() throws InterruptedException {
        Thread t1 = new Thread(() -> {
            synchronized (LOCK) {
                try {
                    get("t").debug("waiting...");
                    LOCK.wait(5000L);
                } catch (InterruptedException e) {
                    get("t").debug("interrupted...");
                    e.printStackTrace();
                }
            }
        }, "t1");
        t1.start();

        Thread.sleep(100);
        synchronized (LOCK) {
            main.debug("other...");
        }

    }
```

```java
    private static void sleeping() throws InterruptedException {
        Thread t1 = new Thread(() -> {
            synchronized (LOCK) {
                try {
                    get("t").debug("sleeping...");
                    Thread.sleep(5000L);
                } catch (InterruptedException e) {
                    get("t").debug("interrupted...");
                    e.printStackTrace();
                }
            }
        }, "t1");
        t1.start();

        Thread.sleep(100);
        synchronized (LOCK) {
            main.debug("other...");
        }
    }
}
```

1.10 如何停止一个正在运行的线程？

> 难易程度：☆☆
>
> 出现频率：☆☆

参考回答：

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

代码参考如下：

① 使用退出标志，使线程正常退出。

```
public class MyInterrupt1 extends Thread {

    volatile boolean flag = false ;     // 线程执行的退出标记

    @Override
    public void run() {
        while(!flag) {
            System.out.println("MyThread...run...");
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

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

        // 创建MyThread对象
        MyInterrupt1 t1 = new MyInterrupt1() ;
        t1.start();

        // 主线程休眠6秒
        Thread.sleep(6000);

        // 更改标记为true
        t1.flag = true ;

    }
}
```

② 使用**stop**方法强行终止

```
public class MyInterrupt2 extends Thread {

    volatile boolean flag = false ;     // 线程执行的退出标记

    @Override
```

```java
    public void run() {
        while(!flag) {
            System.out.println("MyThread...run...");
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

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

        // 创建MyThread对象
        MyInterrupt2 t1 = new MyInterrupt2() ;
        t1.start();

        // 主线程休眠2秒
        Thread.sleep(6000);

        // 调用stop方法
        t1.stop();

    }
}
```

③ 使用**interrupt**方法中断线程。

```java
package com.itheima.basic;

public class MyInterrupt3 {

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

        //1.打断阻塞的线程
        /*Thread t1 = new Thread(()->{
            System.out.println("t1 正在运行...");
            try {
                Thread.sleep(5000);
```

```java
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "t1");
        t1.start();
        Thread.sleep(500);
        t1.interrupt();
        System.out.println(t1.isInterrupted());*/


        //2.打断正常的线程
        Thread t2 = new Thread(()->{
            while(true) {
                Thread current = Thread.currentThread();
                boolean interrupted = current.isInterrupted();
                if(interrupted) {
                    System.out.println("打断状态: "+interrupted);
                    break;
                }
            }
        }, "t2");
        t2.start();
        Thread.sleep(500);
//        t2.interrupt();

    }
}
```

2.线程中并发锁

2.1 讲一下synchronized关键字的底层原理？

> 难易程度：☆☆☆☆☆
>
> 出现频率：☆☆☆

2.1.1 基本使用

如下抢票的代码，如果不加锁，就会出现超卖或者一张票卖给多个人

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

```
public class TicketDemo {

    static Object lock = new Object();
    int ticketNum = 10;


    public synchronized void getTicket() {
        synchronized (this) {
            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();
        }
    }



}
```

2.1.2 Monitor

Monitor 被翻译为监视器，是由jvm提供，c++语言实现

在代码中想要体现monitor需要借助javap命令查看clsss的字节码，比如以下代码：

```
public class SyncTest {

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

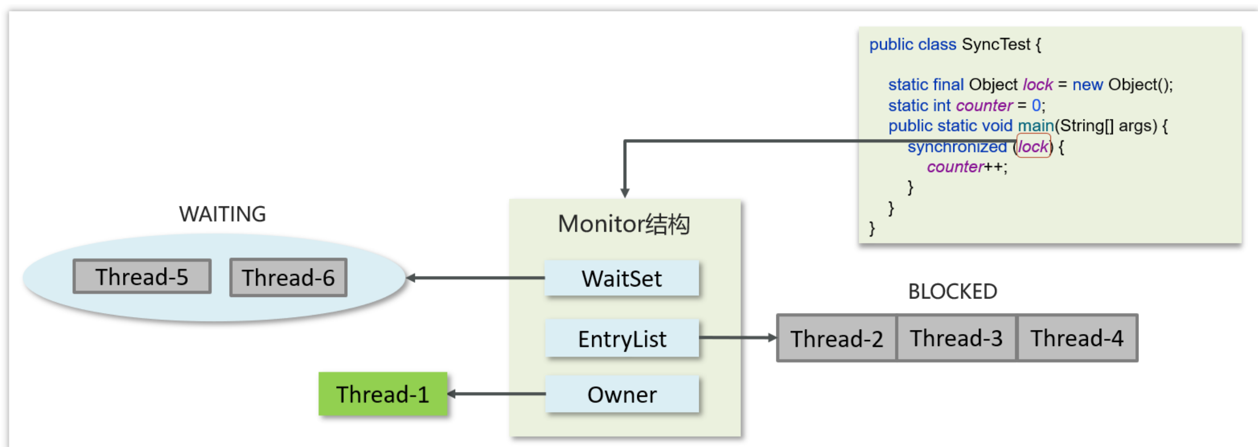
找到这个类的class文件，在class文件目录下执行 `javap -v SyncTest.class`，反编译效果如下：

```
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
```

- monitorenter　上锁开始的地方
- monitorexit　　解锁的地方
- 其中被monitorenter和monitorexit包围住的指令就是上锁的代码
- 有两个monitorexit的原因，第二个monitorexit是为了防止锁住的代码抛异常后不能及时释放锁

在使用了synchornized代码块时需要指定一个对象，所以synchornized也被称为对象锁

monitor主要就是跟这个对象产生关联，如下图



Monitor内部具体的存储结构：

- 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

具体的流程：

- 代码进入synchorized代码块，先让lock（对象锁）关联的monitor，然后判断Owner是否有线程持有
- 如果没有线程持有，则让当前线程持有，表示该线程获取锁成功
- 如果有线程持有，则让当前线程进入entryList进行阻塞，如果Owner持有的线程已经释放了锁，在EntryList中的线程去竞争锁的持有权（非公平）
- 如果代码块中调用了wait()方法，则会进去WaitSet中进行等待

参考回答：

- 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

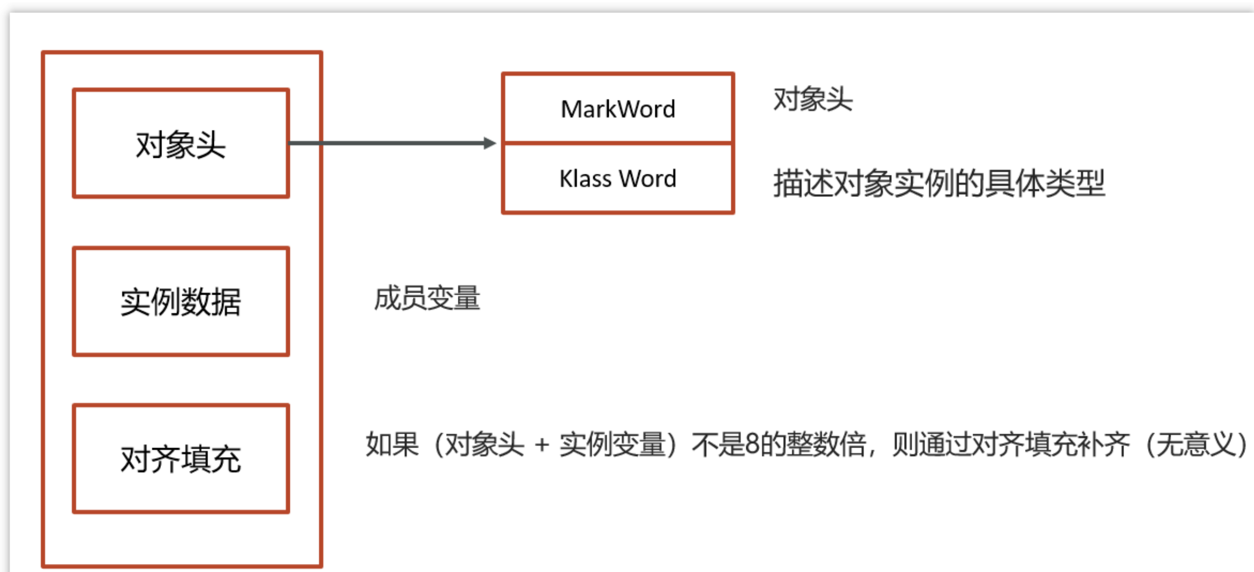
2.2 synchronized关键字的底层原理-进阶

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。

2.2.1 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



我们需要重点分析MarkWord对象头

2.2.2 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

lock标识，占2位

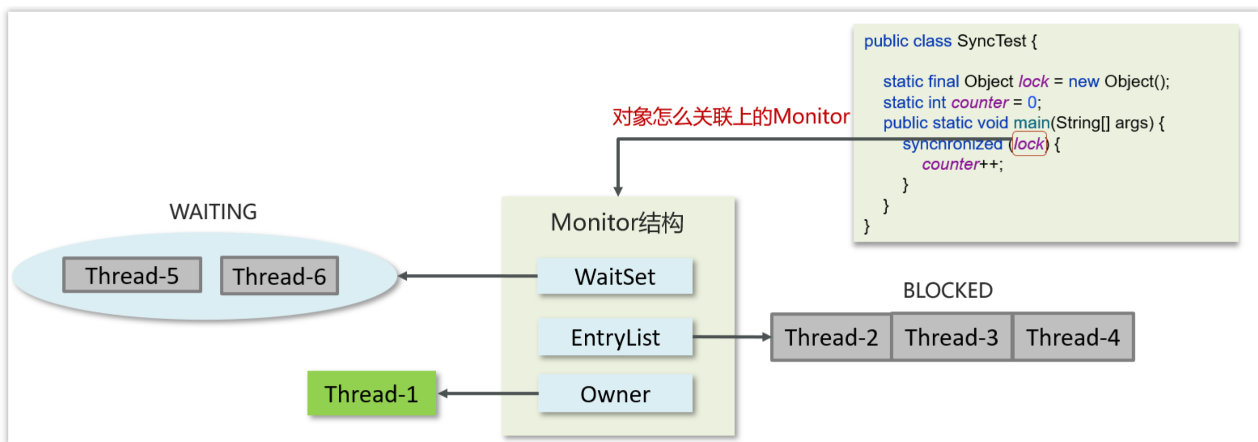
- hashcode：25位的对象标识Hash码
- age：对象分代年龄占4位
- biased_lock：偏向锁标识，占1位 ，0表示没有开始偏向锁，1表示开启了偏向锁
- thread：持有偏向锁的线程ID，占23位
- epoch：偏向时间戳，占2位
- ptr_to_lock_record：轻量级锁状态下，指向栈中锁记录的指针，占30位
- ptr_to_heavyweight_monitor：重量级锁状态下，指向对象监视器Monitor的指针，占30位

我们可以通过lock的标识，来判断是哪一种锁的等级

- 后三位是001表示无锁
- 后三位是101表示偏向锁
- 后两位是00表示轻量级锁
- 后两位是10表示重量级锁

2.2.3 再说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**



简单说就是：每个对象的对象头都可以设置monoitor的指针，让对象与monitor产生关联

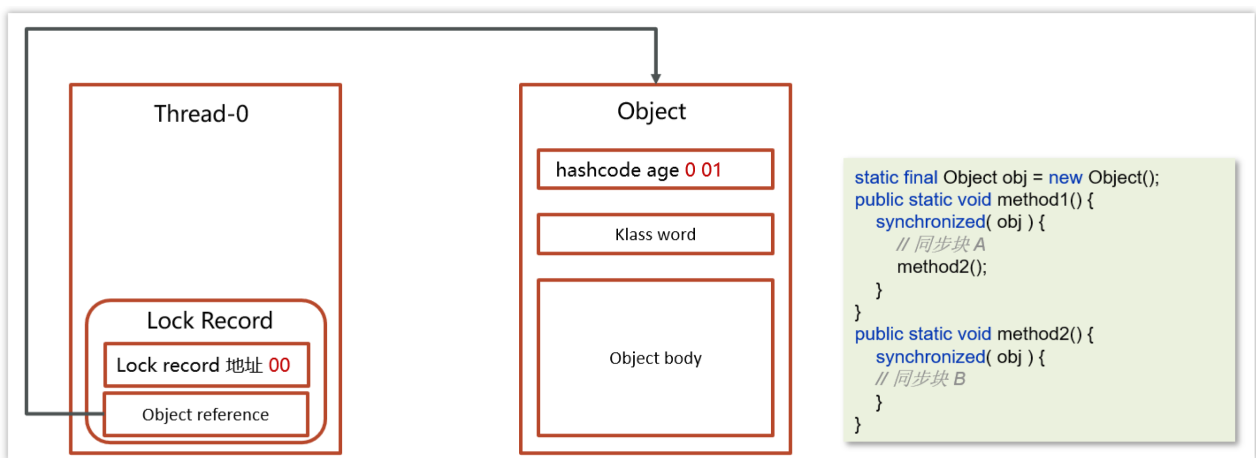
2.2.4 轻量级锁

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。

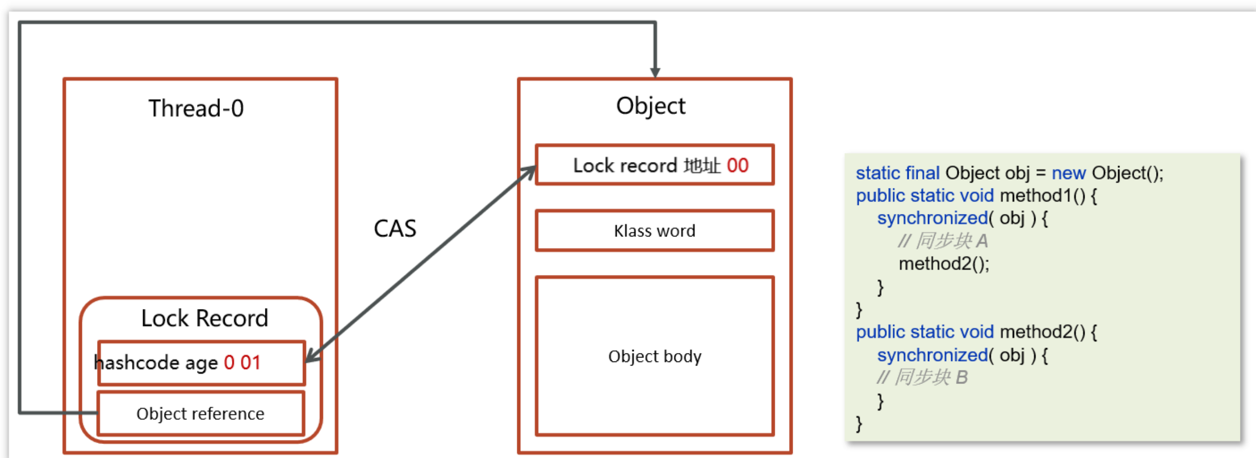
```
static final Object obj = new Object();

public static void method1() {
    synchronized (obj) {
        // 同步块 A
        method2();
    }
}

public static void method2() {
    synchronized (obj) {
        // 同步块 B
    }
}
```

加锁的流程

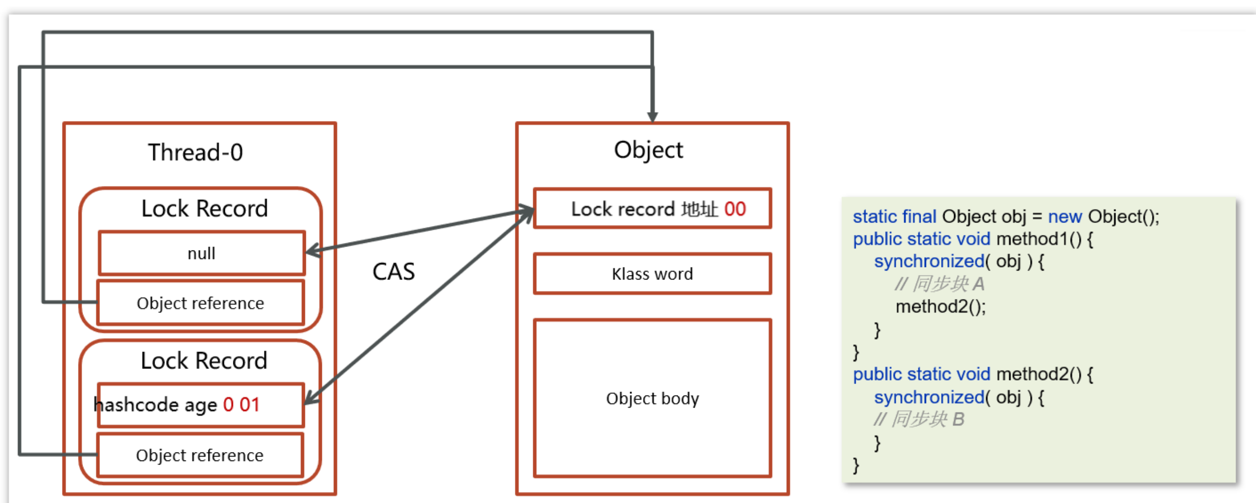
1.在线程栈中创建一个Lock Record，将其obj字段指向锁对象。



2.通过CAS指令将Lock Record的地址存储在对象头的mark word中（数据进行交换），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。

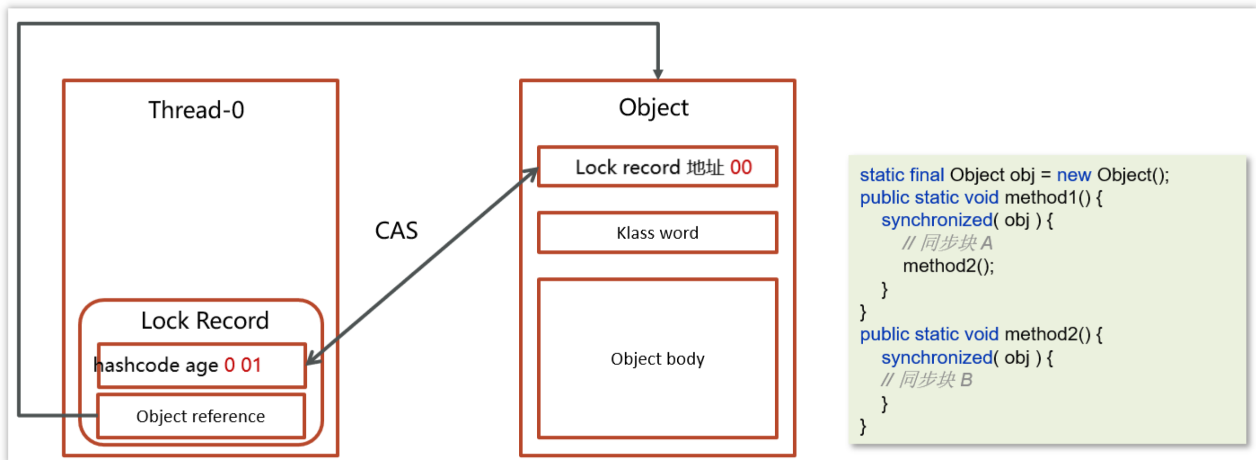


4.如果CAS修改失败，说明发生了竞争，需要膨胀为重量级锁。

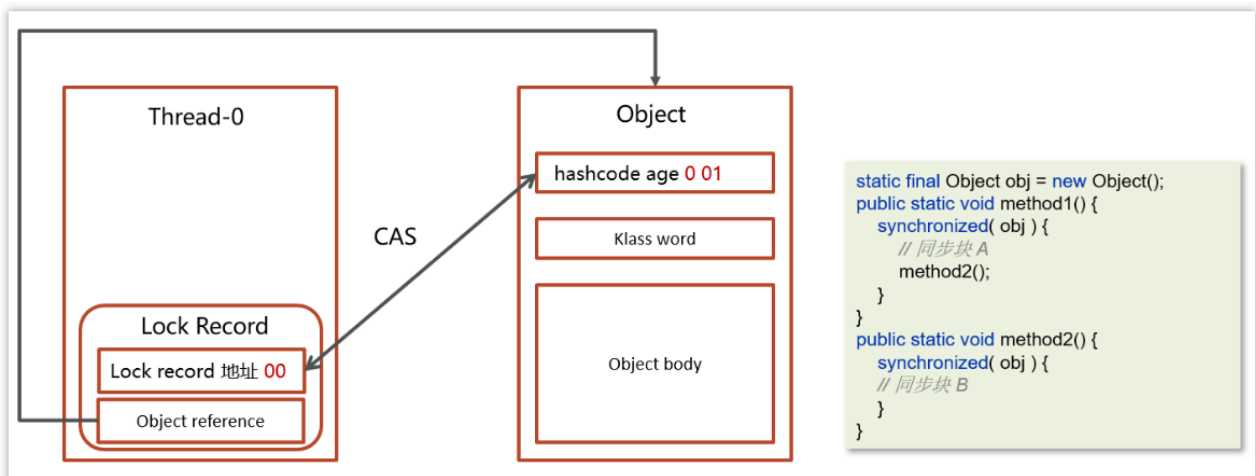
解锁过程

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。



2.2.5 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

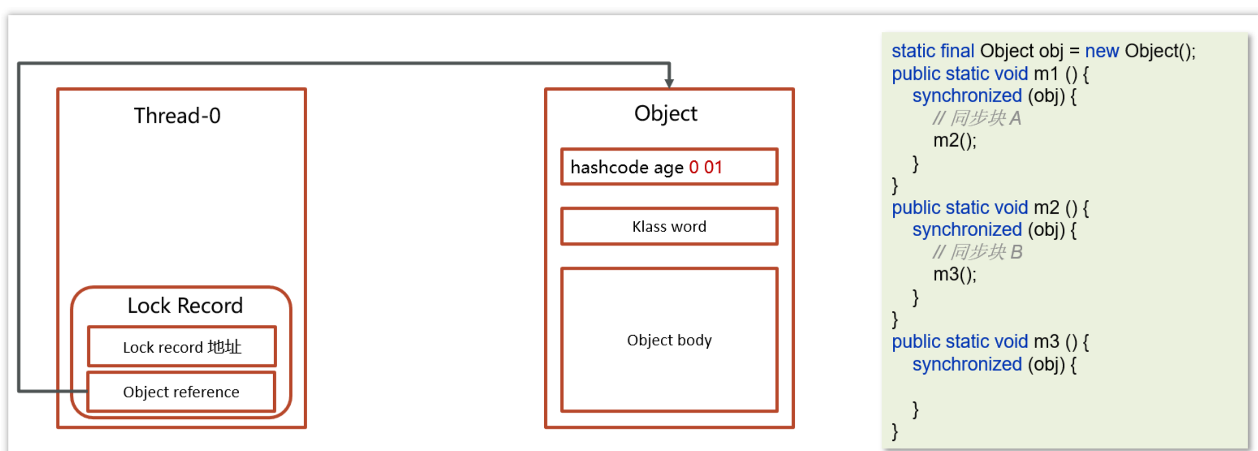
这个线程 ID 是自己的就表示没有竞争，不用重新 CAS。以后只要不发生竞争，这个对象就归该线程所有

```
static final Object obj = new Object();
```

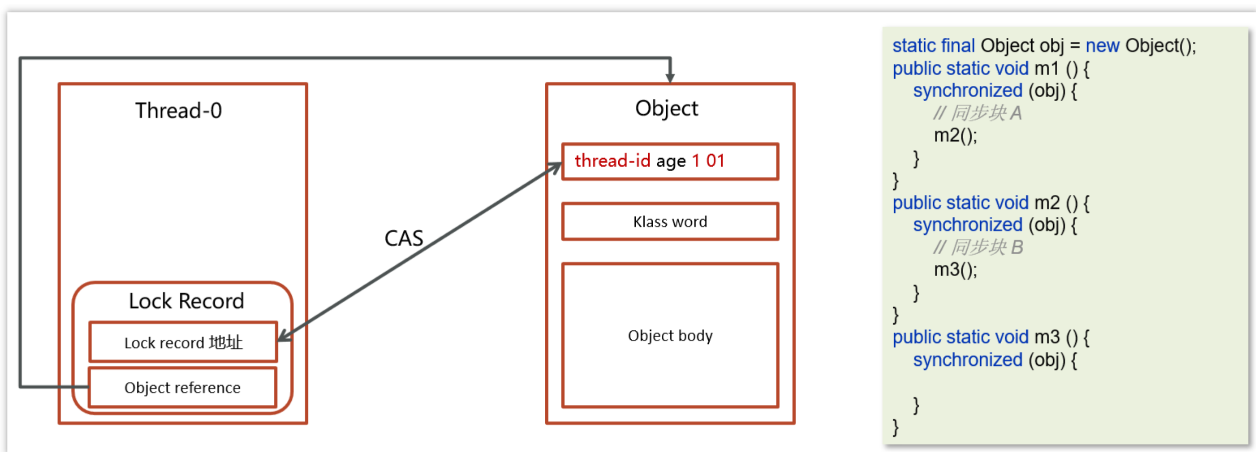
```
public static void m1() {
    synchronized (obj) {
        // 同步块 A
        m2();
    }
}

public static void m2() {
    synchronized (obj) {
        // 同步块 B
        m3();
    }
}

public static void m3() {
    synchronized (obj) {

    }
}
```

加锁的流程

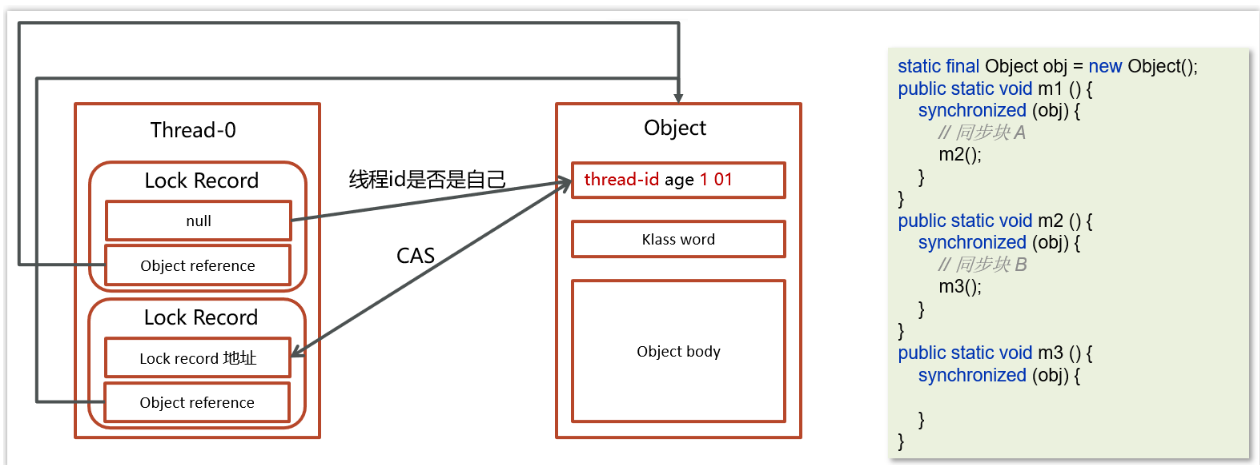
1.在线程栈中创建一个Lock Record，将其obj字段指向锁对象。



2.通过CAS指令将Lock Record的**线程id**存储在对象头的mark word中，同时也设置偏向锁的标识为101，如果对象处于无锁状态则修改成功，代表该线程获得了偏向锁。



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。与轻量级锁不同的时，这里不会再次进行cas操作，只是判断对象头中的线程id是否是自己，因为缺少了cas操作，性能相对轻量级锁更好一些



解锁流程参考轻量级锁

2.2.6 参考回答

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。

	描述
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

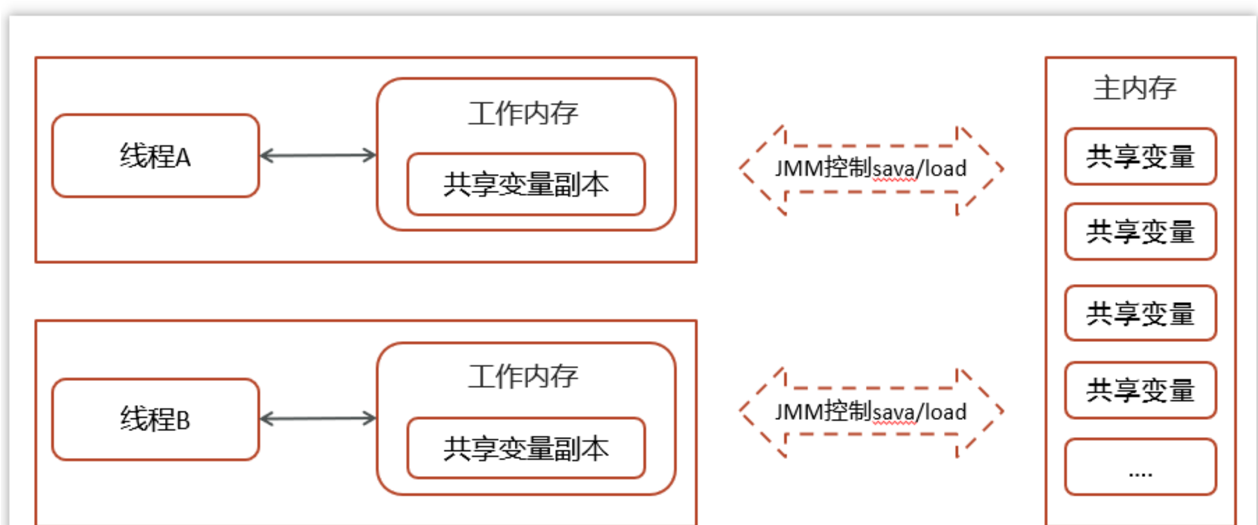
一旦锁发生了竞争，都会升级为重量级锁

2.3你谈谈 JMM（Java 内存模型）

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

JMM(Java Memory Model)Java内存模型,是java虚拟机规范中所定义的一种内存模型。

Java内存模型(Java Memory Model)描述了Java程序中各种变量(线程共享变量)的访问规则，以及在JVM中将变量存储到内存和从内存中读取变量这样的底层细节。



特点：

1. 所有的共享变量都存储于主内存(计算机的RAM)这里所说的变量指的是实例变量和类变量。不包含局部变量，因为局部变量是线程私有的，因此不存在竞争问题。
2. 每一个线程还存在自己的工作内存，线程的工作内存，保留了被线程使用的变量的工作副本。
3. 线程对变量的所有的操作(读，写)都必须在工作内存中完成，而不能直接读写主内存中的变量，不同线程之间也不能直接访问对方工作内存中的变量，线程间变量的值的传递需要通过主内存完成。

2.4 CAS 你知道吗？

> 难易程度：☆☆☆
>
> 出现频率：☆☆

2.4.1 概述及基本工作流程

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。

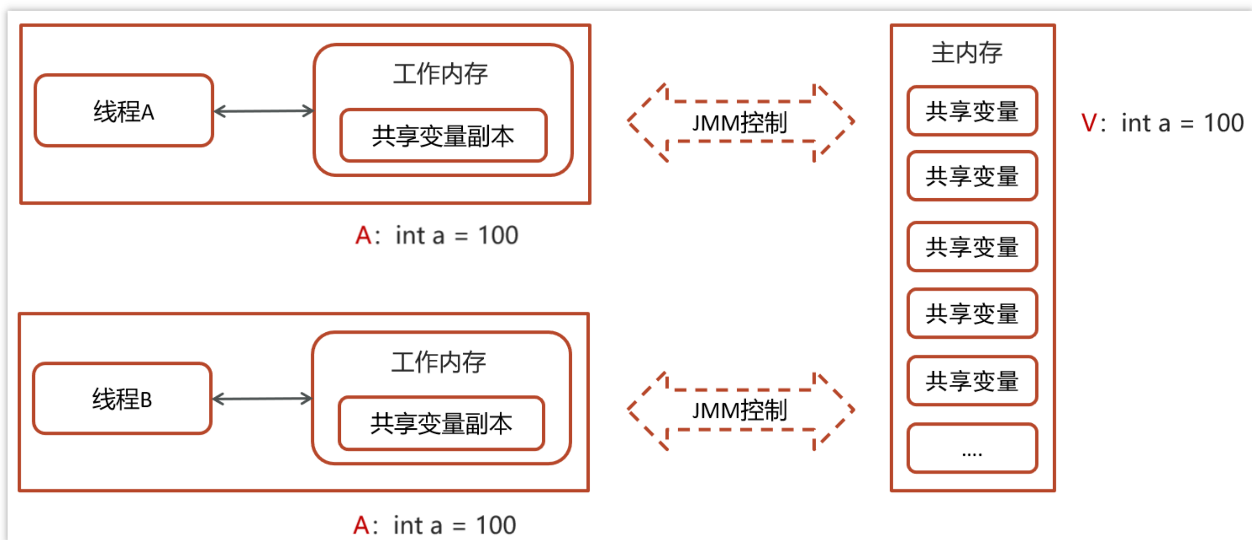
在JUC（ java.util.concurrent ）包下实现的很多类都用到了CAS操作

- AbstractQueuedSynchronizer（AQS框架）
- AtomicXXX类

例子：

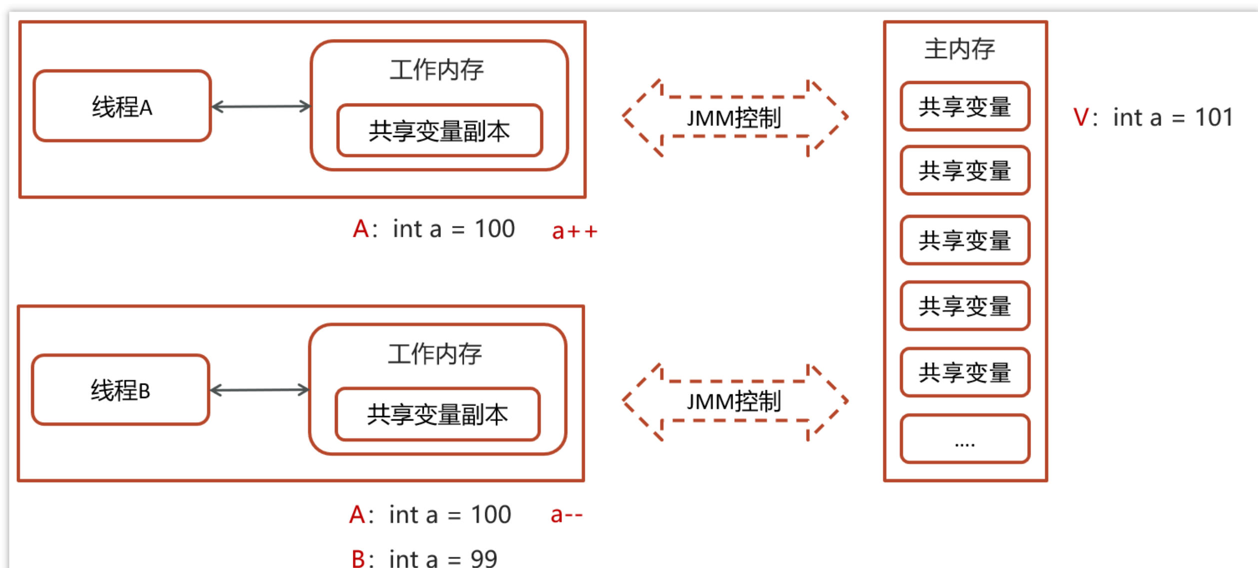
我们还是基于刚才学习过的JMM内存模型进行说明

- 线程1与线程2都从主内存中获取变量int a = 100,同时放到各个线程的工作内存中

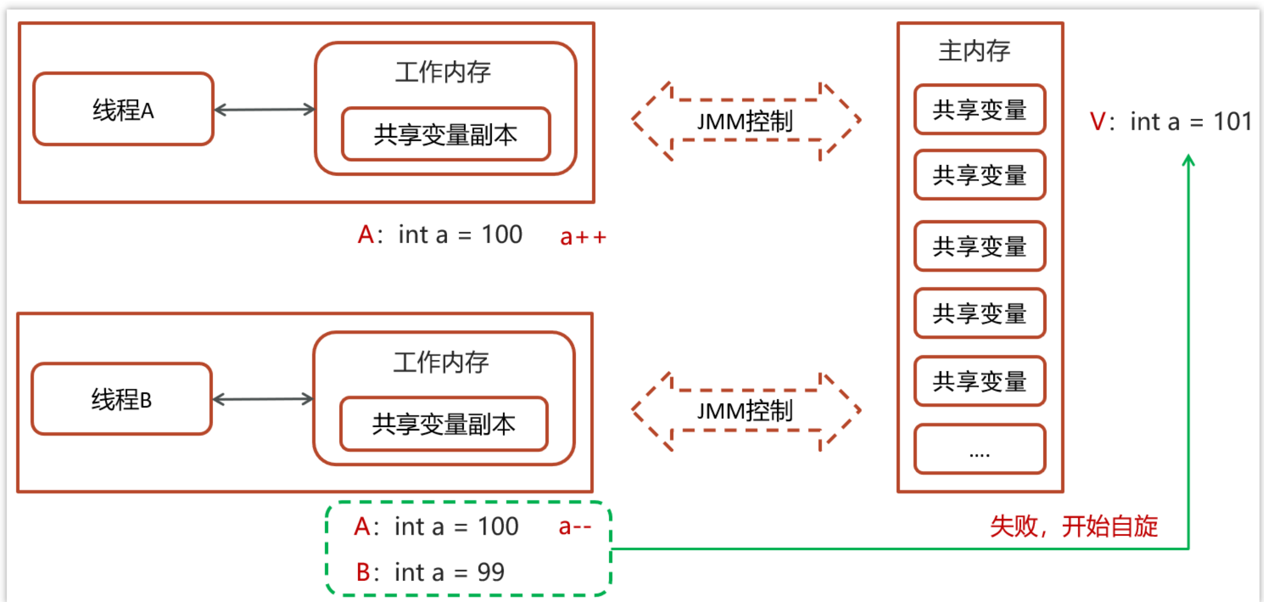


> 一个当前内存值V、旧的预期值A、即将更新的值B，当且仅当旧的预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做，并返回false。如果CAS操作失败，通过自旋的方式等待并再次尝试，直到成功

- 线程1操作：V：int a = 100，A：int a = 100，B：修改后的值：int a = 101 (a++)
  - 线程1拿A的值与主内存V的值进行比较，判断是否相等
  - 如果相等，则把B的值101更新到主内存中



- 线程2操作：V：int a = 100，A：int a = 100，B：修改后的值：int a = 99(a--)
  - 线程2拿A的值与主内存V的值进行比较，判断是否相等(目前不相等，因为线程1已更新V的值99)
  - 不相等，则线程2更新失败



- 自旋锁操作
  - 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
    }
}
```

需要不断尝试获取共享内存V中最新的值，然后再在新的值的基础上进行更新操作，如果失败就继续尝试获取新的值，直到更新成功

2.4.2 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

```java
public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);

public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);

public final native boolean compareAndSwapLong(Object var1, long var2, long var4, long var6);
```

都是native修饰的方法，由系统提供的接口执行，并非java代码实现，一般的思路也都是自旋锁实现

```
// 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
    }
}
```

在java中比较常见使用有很多，比如ReentrantLock和Atomic开头的线程安全类，都调用了Unsafe中的方法

- 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

2.4.3 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。

2.5 请谈谈你对 volatile 的理解

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

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：

2.5.1 保证线程间的可见性

保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的,volatile关键字会强制将修改的值立即写入主存。

一个典型的例子：永不停止的循环

```java
package com.itheima.basic;


// 可见性例子
// -Xint
public class ForeverLoop {
    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("modify stop to true...");
        }).start();
        foo();
    }

    static void foo() {
        int i = 0;
        while (!stop) {
            i++;
        }
        System.out.println("stopped... c:"+ i);
    }
}
```

当执行上述代码的时候，发现foo()方法中的循环是结束不了的，也就说读取不到共享变量的值结束循环。

主要是因为在JVM虚拟机中有一个JIT（即时编辑器）给代码做了优化。

> 上述代码
>
> ```
> while (!stop) {
> i++;
> }
> ```
>
> 在很短的时间内，这个代码执行的次数太多了，当达到了一个阈值，JIT就会优化此代码，如下：
>
> ```
> while (true) {
> i++;
> }
> ```
>
> 当把代码优化成这样子以后，及时 `stop` 变量改变为了 `false` 也依然停止不了循环

解决方案：

第一：

在程序运行的时候加入vm参数 `-Xint` 表示禁用即时编辑器，不推荐，得不偿失（其他程序还要使用）

第二：

在修饰 `stop` 变量的时候加上 `volatile` ,表示当前代码禁用了即时编辑器，问题就可以解决，代码如下：

```
static volatile boolean stop = false;
```

2.5.2 禁止进行指令重排序

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

```java
int x;
int y;

@Actor
public void actor1() {
    x = 1;
    y = 1;
}

@Actor
public void actor2(II_Result r) {
    r.r1 = y;
    r.r2 = x;
}
```

在去获取上面的结果的时候，有可能会出现4种情况

情况一：先执行actor2获取结果--->0,0(正常)

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(正常)

情况三：先执行actor1中所有代码，然后执行actor2获取结果--->1,1(正常)

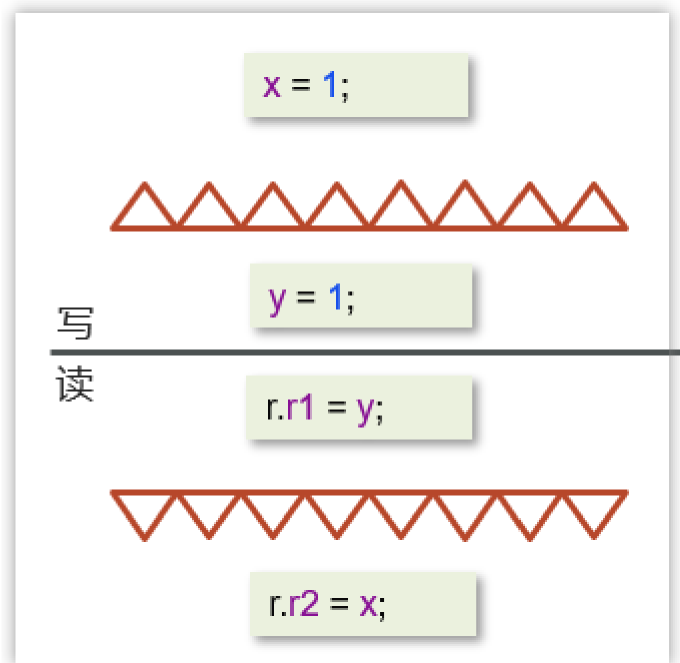
情况四：先执行actor1中第二行代码，然后执行actor2获取结果--->1,0(发生了指令重排序，影响结果)

解决方案

在变量上添加volatile，禁止指令重排序，则可以解决问题

```java
int x;
volatile int y;

@Actor
public void actor1() {
    x = 1;
    y = 1;
}

@Actor
public void actor2(II_Result r) {
    r.r1 = y;
    r.r2 = x;
}
```

屏障添加的示意图



- 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

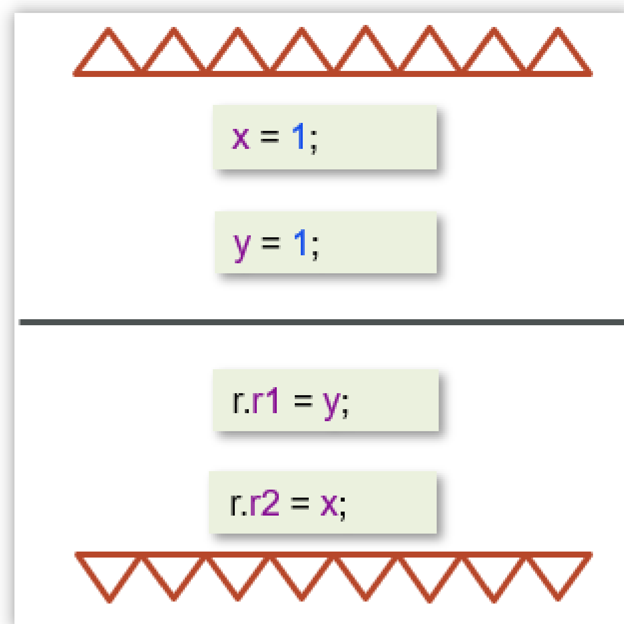
其他补充

我们上面的解决方案是把volatile加在了int y这个变量上，我们能不能把它加在int x这个变量上呢？

下面代码使用volatile修饰了x变量

```
volatile int x;
int y;

@Actor
public void actor1() {
    x = 1;
    y = 1;
}

@Actor
public void actor2(II_Result r) {
    r.r1 = y;
    r.r2 = x;
}
```

屏障添加的示意图



这样显然是不行的，主要是因为下面两个原则：

- 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

所以，现在我们就可以总结一个volatile使用的小妙招：

- 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

2.6 什么是AQS？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

2.6.1 概述

全称是 AbstractQueuedSynchronizer，是阻塞式锁和相关的同步器工具的框架，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

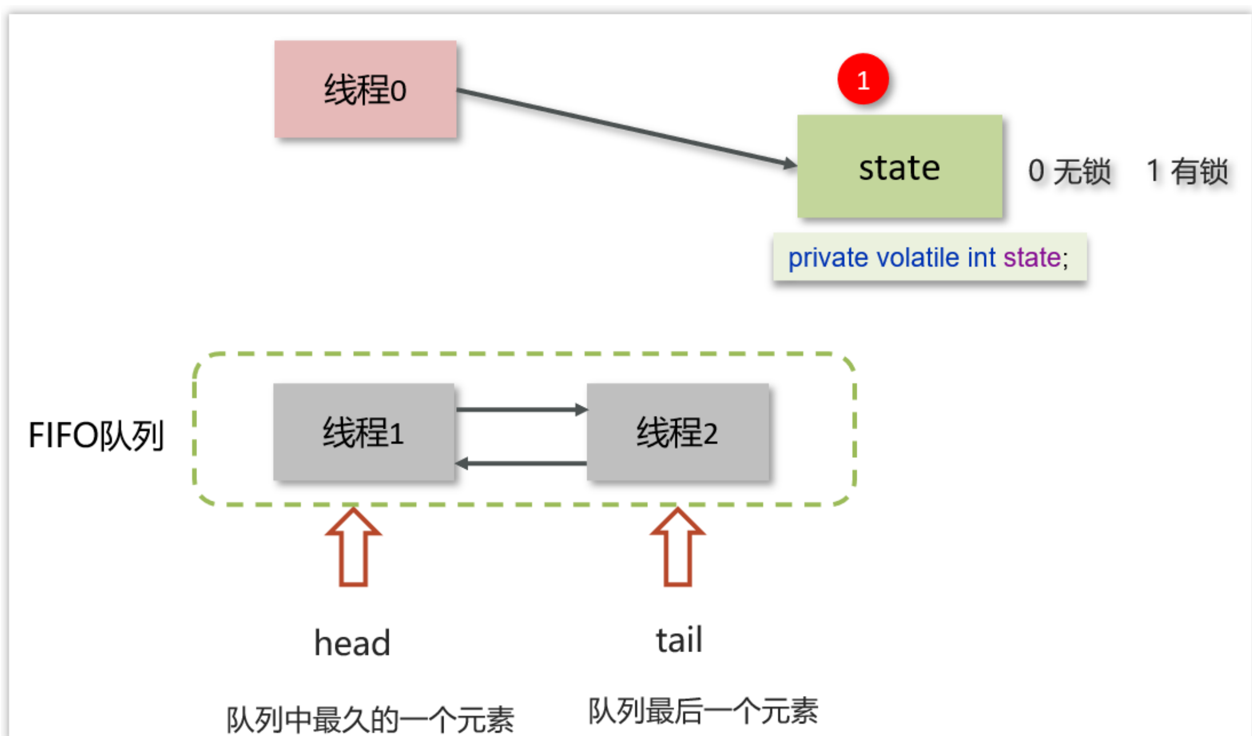
synchronized	AQS
关键字，c++ 语言实现	java 语言实现
悲观锁，自动释放锁	悲观锁，手动开启和关闭
锁竞争激烈都是重量级锁，性能差	锁竞争激烈的情况下，提供了多种解决方案

AQS常见的实现类

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

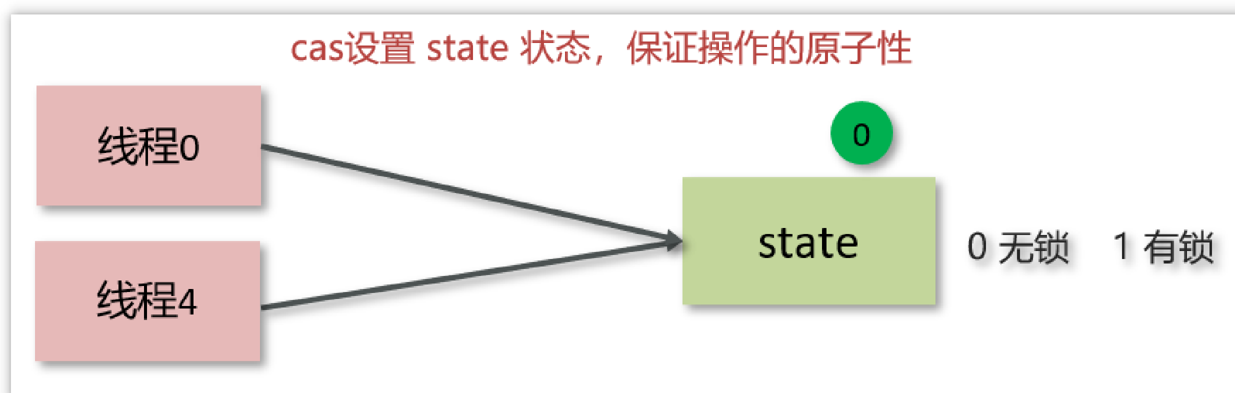
2.6.2 工作机制

- 在AQS中维护了一个使用了volatile修饰的state属性来表示资源的状态，0表示无锁，1表示有锁
- 提供了基于 FIFO 的等待队列，类似于 Monitor 的 EntryList
- 条件变量来实现等待、唤醒机制，支持多个条件变量，类似于 Monitor 的 WaitSet



- 线程0来了以后，去尝试修改state属性，如果发现state属性是0，就修改state状态为1，表示线程0抢锁成功
- 线程1和线程2也会先尝试修改state属性，发现state的值已经是1了，有其他线程持有锁，它们都会到FIFO队列中进行等待，
- FIFO是一个双向队列，head属性表示头结点，tail表示尾结点

如果多个线程共同去抢这个资源是如何保证原子性的呢？



在去修改state状态的时候，使用的cas自旋锁来保证原子性，确保只能有一个线程修改成功，修改失败的线程将会进入FIFO队列中等待

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

> 比较典型的AQS实现类ReentrantLock，它默认就是非公平锁，新的线程与队列中的线程共同来抢资源

2.5 ReentrantLock的实现原理

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

2.5.1 概述

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();
}
```

2.5.2 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高，在许多线程访问的情况下，公平锁表现出较低的吞吐量。

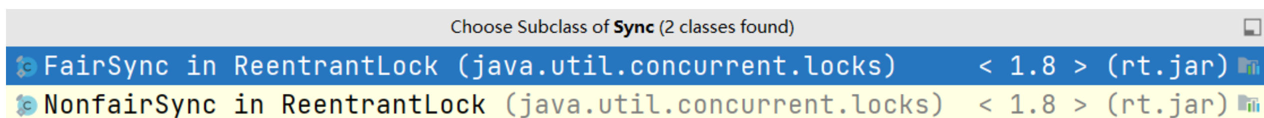
查看ReentrantLock源码中的构造方法：

```
public ReentrantLock() {
    sync = new NonfairSync();
}

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

提供了两个构造方法，不带参数的默认为非公平

如果使用带参数的构造函数，并且传的值为true，则是公平锁

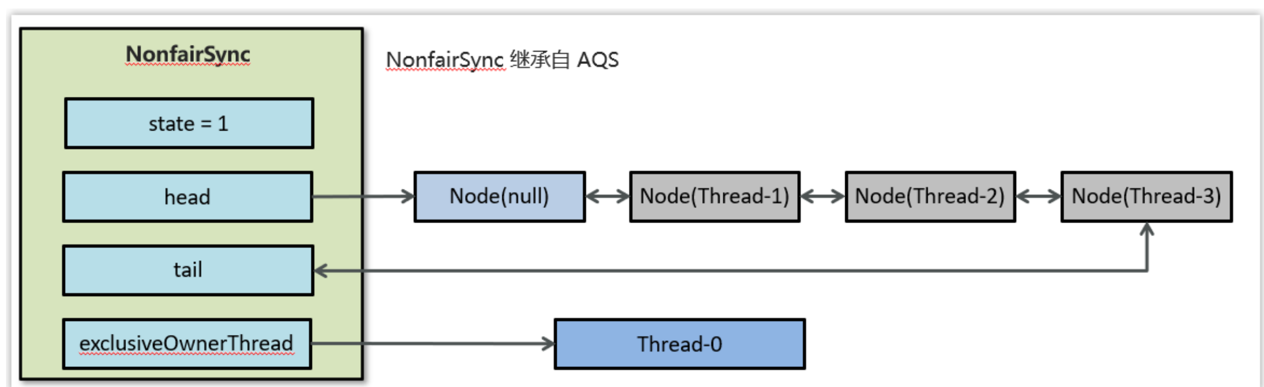
其中NonfairSync和FairSync这两个类父类都是Sync



而Sync的父类是AQS，所以可以得出ReentrantLock底层主要实现就是基于AQS来实现的

```
abstract static class Sync extends AbstractQueuedSynchronizer {

}
```

工作流程



- 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

2.6 synchronized和Lock有什么区别？

> 难易程度：☆☆☆☆

> 出现频率：☆☆☆☆

参考回答

- 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，例如获取等待状态、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
- 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

2.7 死锁产生的条件是什么？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

死锁：一个线程需要同时获取多把锁，这时就容易发生死锁

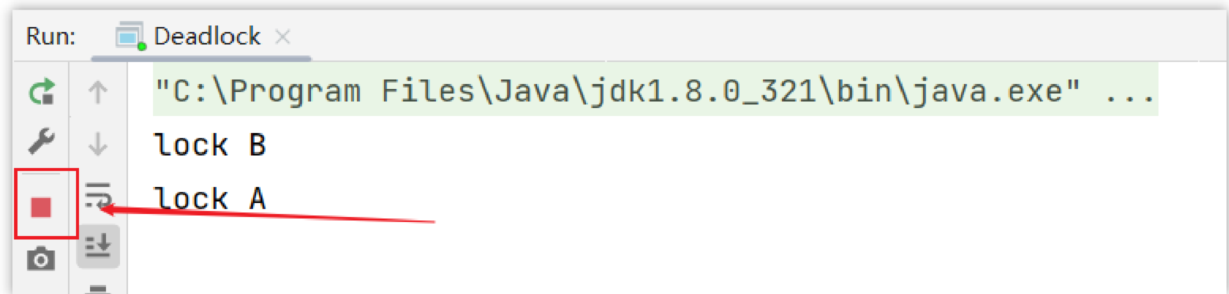
> 例如：
>
> t1 线程获得A对象锁，接下来想获取B对象的锁
>
> t2 线程获得B对象锁，接下来想获取A对象的锁

代码如下：

```
package com.itheima.basic;

import static java.lang.Thread.sleep;

public class Deadlock {

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

控制台输出结果



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。

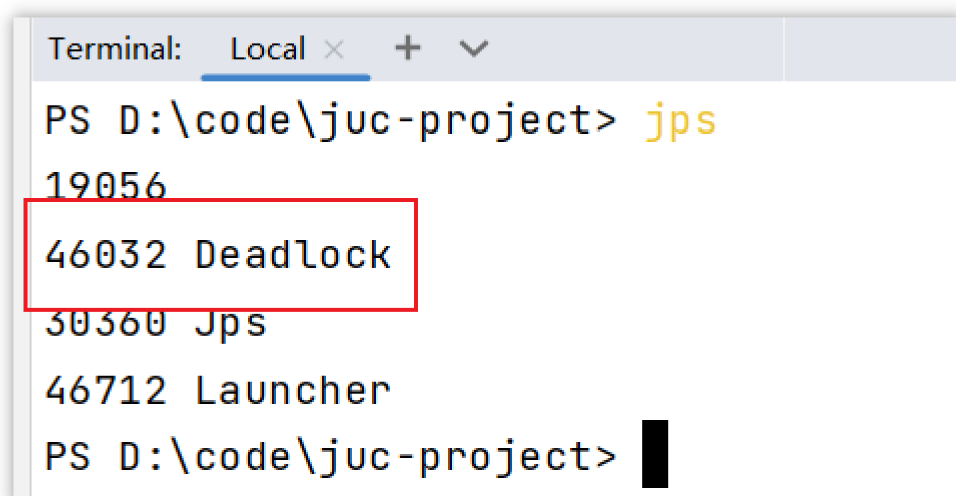
2.8 如何进行死锁诊断？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

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

步骤如下：

第一：查看运行的线程



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 0x000001705cfd6bc8 (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
```

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

2.10 ConcurrentHashMap

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

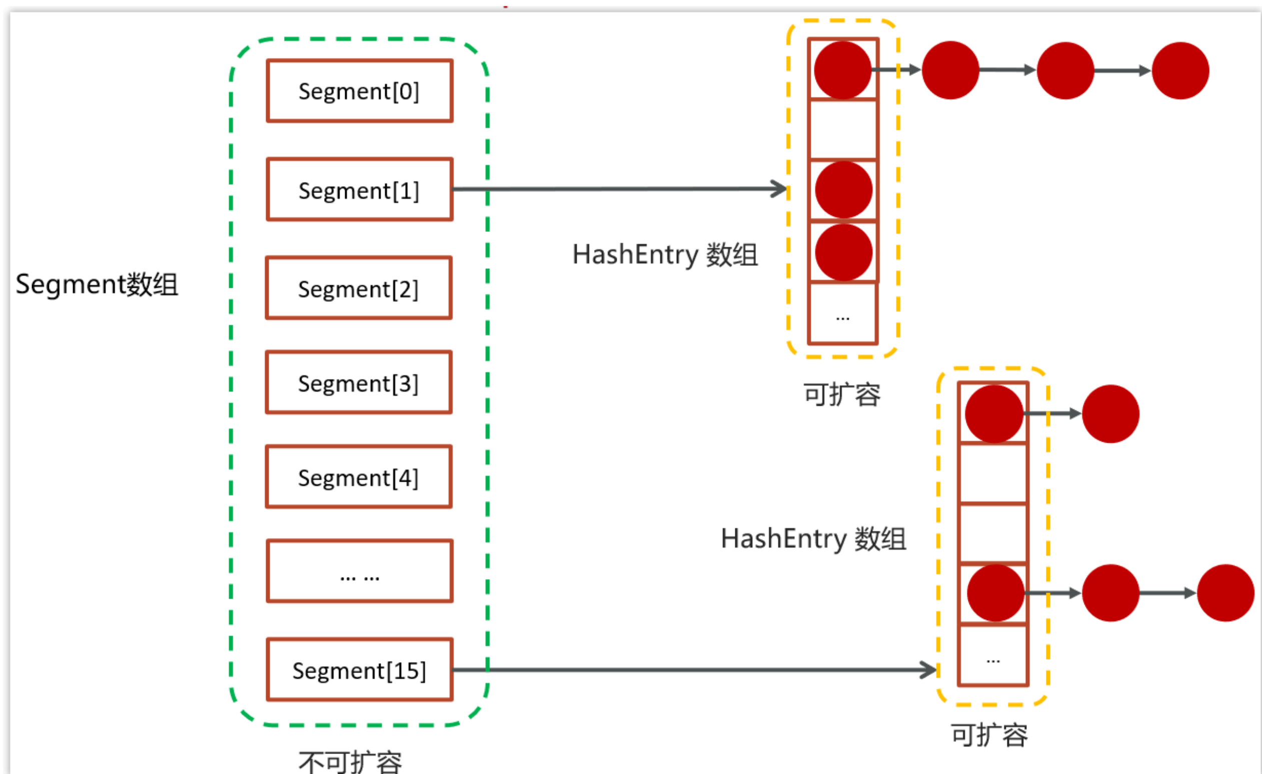
ConcurrentHashMap 是一种线程安全的高效Map集合

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。

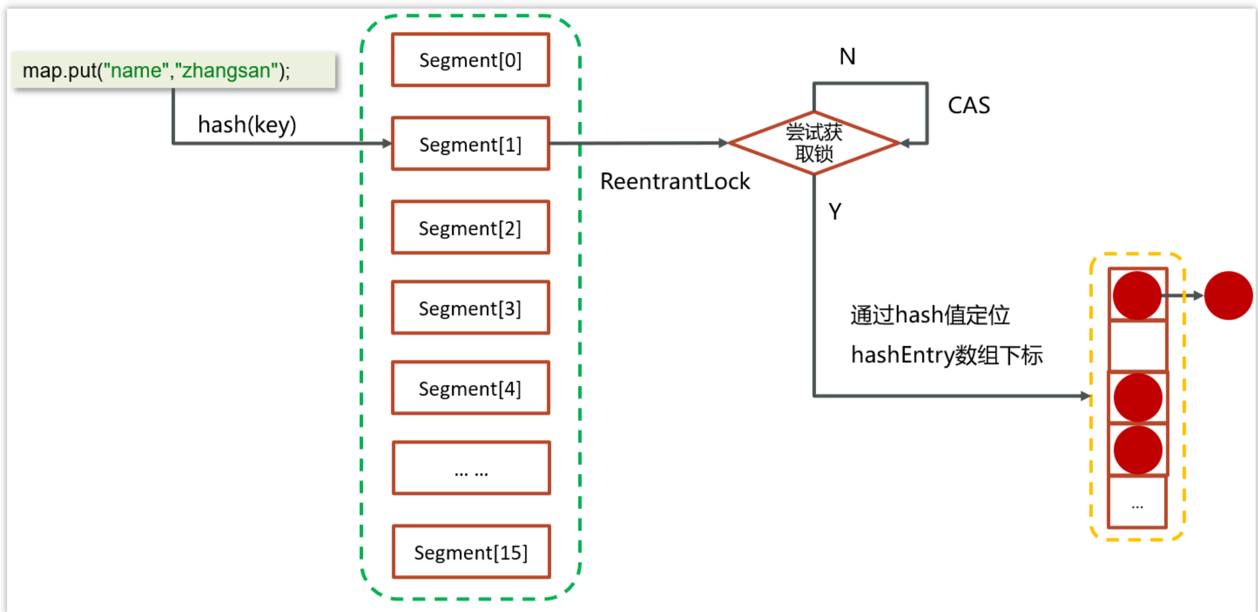
（1） JDK1.7中concurrentHashMap

数据结构



> - 提供了一个segment数组，在初始化ConcurrentHashMap 的时候可以指定数组的长度，默认是16，一旦初始化之后中间不可扩容
> - 在每个segment中都可以挂一个HashEntry数组，数组里面可以存储具体的元素，HashEntry数组是可以扩容的
> - 在HashEntry存储的数组中存储的元素，如果发生冲突，则可以挂单向链表

存储流程



- 先去计算key的hash值，然后确定segment数组下标
- 再通过hash值确定hashEntry数组中的下标存储数据
- 在进行操作数据的之前，会先判断当前segment对应下标位置是否有线程进行操作，为了线程安全使用的是ReentrantLock进行加锁，如果获取锁是被会使用cas自旋锁进行尝试

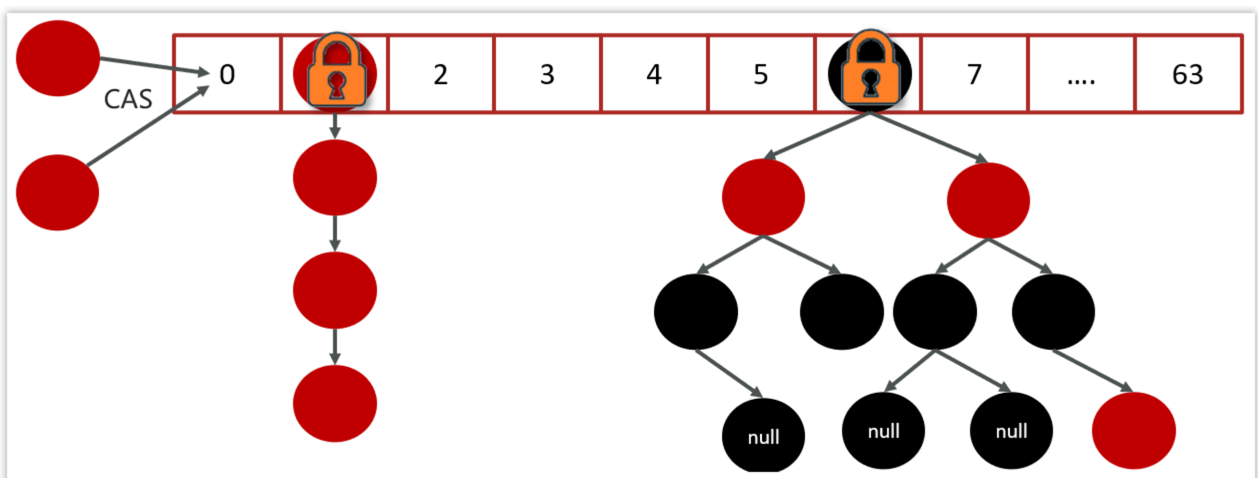
（2） JDK1.8中concurrentHashMap

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题，效率得到提升



2.11 导致并发程序出现问题的根本原因是什么

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

Java并发编程三大特性

- 原子性
- 可见性
- 有序性

（1）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

比如，如下代码能保证原子性吗？

```
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();
    }
}
```

以上代码会出现超卖或者是一张票卖给同一个人，执行并不是原子性的

解决方案：

1.synchronized：同步加锁

2.JUC里面的lock：加锁

```java
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();
    }
}
```

（3）内存可见性

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

比如，以下代码不能保证内存可见性

```java
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（推荐）
- LOCK

（3）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

还是之前的例子，如下代码：

```
int x;
int y;

@Actor
public void actor1() {
    x = 1;
    y = 1;
}

@Actor
public void actor2(II_Result r) {
    r.r1 = y;
    r.r2 = x;
}
```

解决方案：

- volatile

3.线程池

3.1 说一下线程池的核心参数（线程池的执行原理知道嘛）

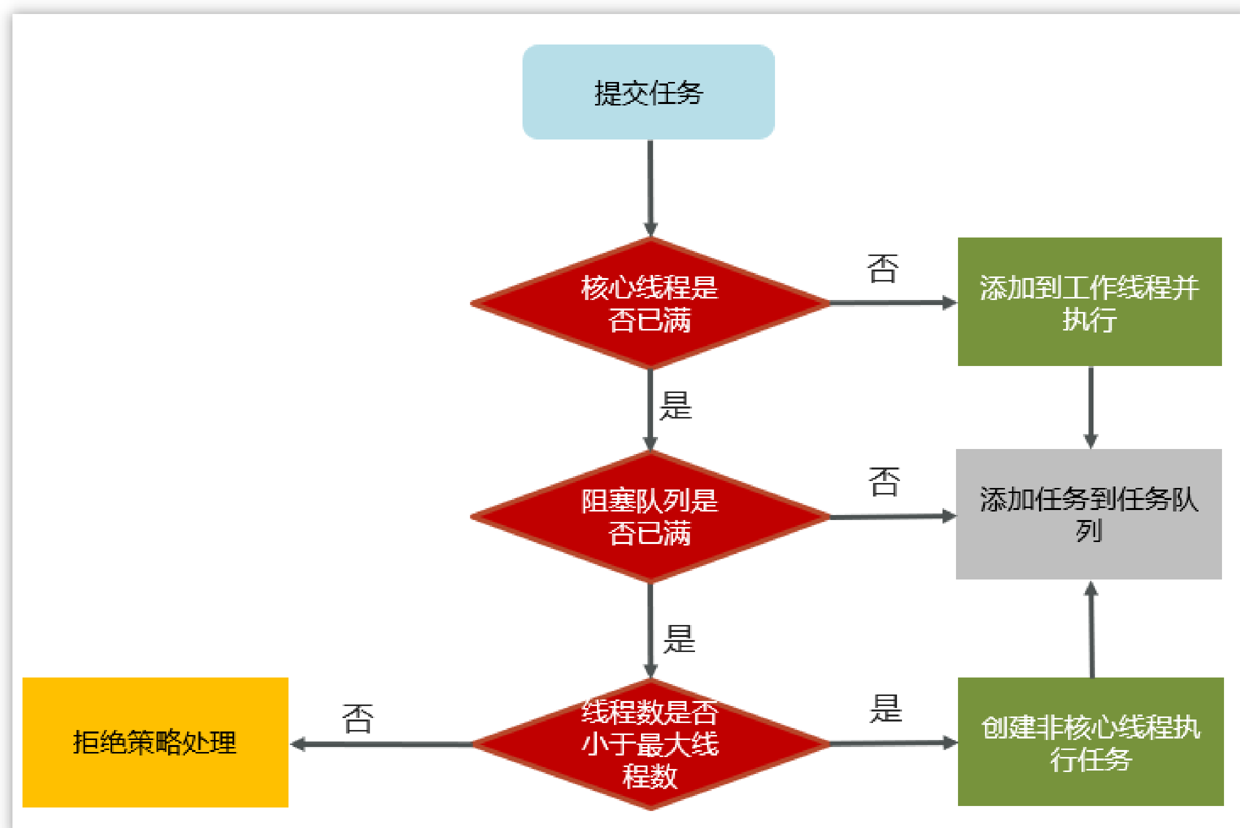
> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

线程池核心参数主要参考ThreadPoolExecutor这个类的7个参数的构造函数

```java
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

工作流程



> 1，任务在提交的时候，首先判断核心线程数是否已满，如果没有满则直接添加到工作线程执行
>
> 2，如果核心线程数满了，则判断阻塞队列是否已满，如果没有满，当前任务存入阻塞队列
>
> 3，如果阻塞队列也满了，则判断线程数是否小于最大线程数，如果满足条件，则使用临时线程执行任务
>
> 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
>
> 4，如果所有线程都在忙着（核心线程+临时线程），则走拒绝策略

拒绝策略：

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；

参考代码：

```
public class TestThreadPoolExecutor {

    static class MyTask implements Runnable {
        private final String name;
        private final long duration;

        public MyTask(String name) {
            this(name, 0);
        }

        public MyTask(String name, long duration) {
            this.name = name;
            this.duration = duration;
        }

```

```java
        @Override
        public void run() {
            try {
                LoggerUtils.get("myThread").debug("running..." +
this);
                Thread.sleep(duration);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        @Override
        public String toString() {
            return "MyTask(" + name + ")";
        }
    }

    public static void main(String[] args) throws
InterruptedException {
        AtomicInteger c = new AtomicInteger(1);
        ArrayBlockingQueue<Runnable> queue = new
ArrayBlockingQueue<>(2);
        ThreadPoolExecutor threadPool = new ThreadPoolExecutor(
                2,
                3,
                0,
                TimeUnit.MILLISECONDS,
                queue,
                r -> new Thread(r, "myThread" +
c.getAndIncrement()),
                new ThreadPoolExecutor.AbortPolicy());
        showState(queue, threadPool);
        threadPool.submit(new MyTask("1", 3600000));
        showState(queue, threadPool);
        threadPool.submit(new MyTask("2", 3600000));
        showState(queue, threadPool);
        threadPool.submit(new MyTask("3"));
        showState(queue, threadPool);
        threadPool.submit(new MyTask("4"));
        showState(queue, threadPool);
        threadPool.submit(new MyTask("5",3600000));
```

```java
        showState(queue, threadPool);
        threadPool.submit(new MyTask("6"));
        showState(queue, threadPool);
    }

    private static void showState(ArrayBlockingQueue<Runnable>
queue, ThreadPoolExecutor threadPool) {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        List<Object> tasks = new ArrayList<>();
        for (Runnable runnable : queue) {
            try {
                Field callable =
FutureTask.class.getDeclaredField("callable");
                callable.setAccessible(true);
                Object adapter = callable.get(runnable);
                Class<?> clazz =
Class.forName("java.util.concurrent.Executors$RunnableAdapter");
                Field task = clazz.getDeclaredField("task");
                task.setAccessible(true);
                Object o = task.get(adapter);
                tasks.add(o);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        LoggerUtils.main.debug("pool size: {}, queue: {}",
threadPool.getPoolSize(), tasks);
    }

}
```

3.2 线程池中有哪些常见的阻塞队列

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

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

比较常见的有4个，用的最多是ArrayBlockingQueue和LinkedBlockingQueue

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

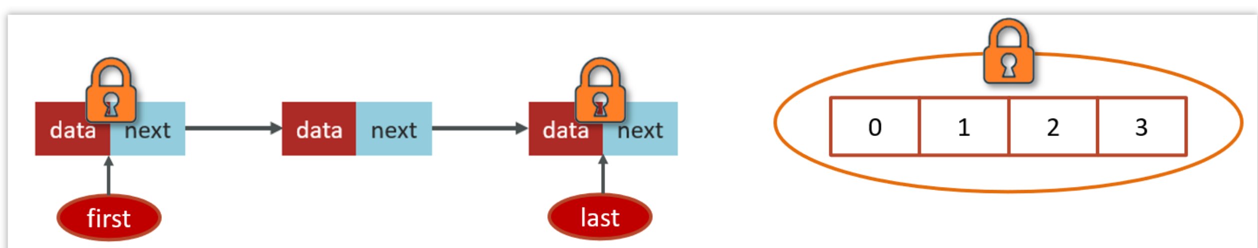
4.SynchronousQueue：不存储元素的阻塞队列，每个插入操作都必须等待一个移出操作。

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

左边是LinkedBlockingQueue加锁的方式，右边是ArrayBlockingQueue加锁的方式

- LinkedBlockingQueue读和写各有一把锁，性能相对较好
- ArrayBlockingQueue只有一把锁，读和写公用，性能相对于LinkedBlockingQueue差一些



3.3 如何确定核心线程数

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

在设置核心线程数之前，需要先熟悉一些执行线程池执行任务的类型

- IO密集型任务

一般来说：文件读写、DB读写、网络请求等

推荐：核心线程数大小设置为2N+1　（N为计算机的CPU核数）

- CPU密集型任务

一般来说：计算型代码、Bitmap转换、Gson转换等

推荐：核心线程数大小设置为N+1　（N为计算机的CPU核数）

java代码查看CPU核数

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

参考回答：

① 高并发、任务执行时间短 -->（ 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 --> (CPU核数 * 2 + 1)
- 计算密集型任务 -->（ 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）

3.4 线程池的种类有哪些

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

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
- 适用场景：适用于任务量已知，相对耗时的任务
- 案例：

```java
public class FixedThreadPoolCase {

    static class FixedThreadDemo implements Runnable{
        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            for (int i = 0; i < 2; i++) {
                System.out.println(name + ":" + i);
            }
        }
    }

```

```java
    public static void main(String[] args) throws
InterruptedException {
        //创建一个固定大小的线程池，核心线程数和最大线程数都是3
        ExecutorService executorService =
Executors.newFixedThreadPool(3);

        for (int i = 0; i < 5; i++) {
            executorService.submit(new FixedThreadDemo());
            Thread.sleep(10);
        }

        executorService.shutdown();
    }

}
```

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

```java
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
- 适用场景：适用于按照顺序执行的任务
- 案例：

```java
public class NewSingleThreadCase {

    static int count = 0;

    static class Demo implements Runnable {
        @Override
        public void run() {
            count++;
```

```
 System.out.println(Thread.currentThread().getName() + ":" +
count);
        }
    }

    public static void main(String[] args) throws
InterruptedException {
        //单个线程池，核心线程数和最大线程数都是1
        ExecutorService exec =
Executors.newSingleThreadExecutor();

        for (int i = 0; i < 10; i++) {
            exec.execute(new Demo());
            Thread.sleep(5);
        }
        exec.shutdown();
    }

}
```

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。
- 适用场景：适合任务数比较密集，但每个任务执行时间较短的情况
- 案例：

```
public class CachedThreadPoolCase {

    static class Demo implements Runnable {
        @Override
```

```java
        public void run() {
            String name = Thread.currentThread().getName();
            try {
                //修改睡眠时间，模拟线程执行需要花费的时间
                Thread.sleep(100);

                System.out.println(name + "执行完了");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) throws
InterruptedException {
        //创建一个缓存的线程，没有核心线程数，最大线程数为
Integer.MAX_VALUE
        ExecutorService exec =
Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
            exec.execute(new Demo());
            Thread.sleep(1);
        }
        exec.shutdown();
    }

}
```

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

```java
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);
}
```

- 适用场景：有定时和延迟执行的任务

- 案例：

```
public class ScheduledThreadPoolCase {

    static class Task implements Runnable {
        @Override
        public void run() {
            try {
                String name =
Thread.currentThread().getName();

                System.out.println(name + ", 开始：" + new
Date());
                Thread.sleep(1000);
                System.out.println(name + ", 结束：" + new
Date());

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

    public static void main(String[] args) throws
InterruptedException {
        //按照周期执行的线程池，核心线程数为2，最大线程数为
Integer.MAX_VALUE
        ScheduledExecutorService scheduledThreadPool =
Executors.newScheduledThreadPool(2);
        System.out.println("程序开始：" + new Date());

        /**
         * schedule 提交任务到线程池中
         * 第一个参数：提交的任务
         * 第二个参数：任务执行的延迟时间
         * 第三个参数：时间单位
         */
        scheduledThreadPool.schedule(new Task(), 0,
TimeUnit.SECONDS);
        scheduledThreadPool.schedule(new Task(), 1,
TimeUnit.SECONDS);
```

```
        scheduledThreadPool.schedule(new Task(), 5,
TimeUnit.SECONDS);

        Thread.sleep(5000);

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

    }

}
```

3.5 为什么不建议用Executors创建线程池

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

参考阿里开发手册《Java开发手册-嵩山版》

4. 【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。

4.线程使用场景问题

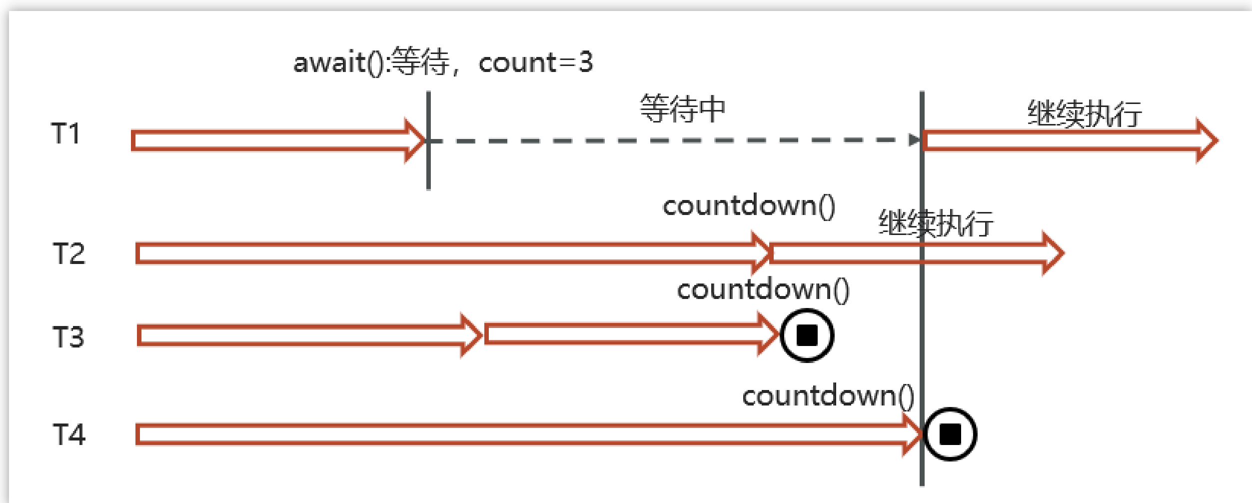
4.1 线程池使用场景CountDownLatch、Future（你们项目哪里用到了多线程）

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

4.1.1 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



案例代码：

```
public class CountDownLatchDemo {

    public static void main(String[] args) throws
InterruptedException {
        //初始化了一个倒计时锁 参数为 3
        CountDownLatch latch = new CountDownLatch(3);

        new Thread(() -> {
            System.out.println(Thread.currentThread().getName()+"-
begin...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            //count--
            latch.countDown();
```

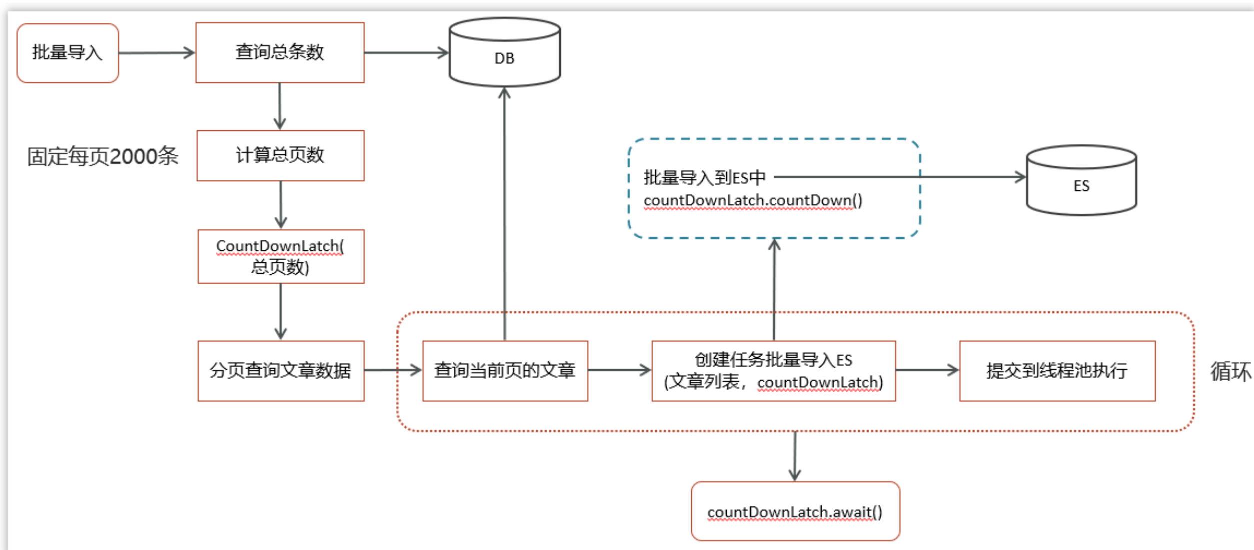
```
            System.out.println(Thread.currentThread().getName()+"-
end..." +latch.getCount());
        }).start();
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName()+"-
begin...");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            //count--
            latch.countDown();
            System.out.println(Thread.currentThread().getName()+"-
end..." +latch.getCount());
        }).start();
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName()+"-
begin...");
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            //count--
            latch.countDown();
            System.out.println(Thread.currentThread().getName()+"-
end..." +latch.getCount());
        }).start();
        String name = Thread.currentThread().getName();
        System.out.println(name + "-waiting...");
        //等待其他线程完成
        latch.await();
        System.out.println(name + "-wait end...");
    }

}
```

4.1.2 案例一（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

整体流程就是通过CountDownLatch+线程池配合去执行

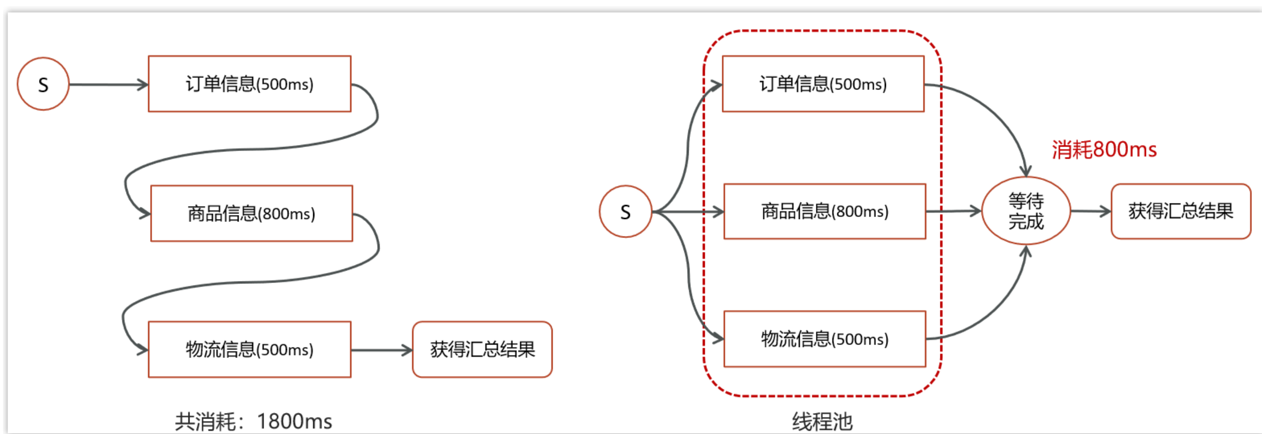
详细实现流程：



> 详细实现代码，请查看当天代码

4.1.3 案例二（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？



> 详细实现代码，请查看当天代码

- 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

4.1.4 案例二（异步调用）

在进行搜索的时候，需要保存用户的搜索记录，而搜索记录不能影响用户的正常搜索，我们通常会开启一个线程去执行历史记录的保存，在新开启的线程在执行的过程中，可以利用线程提交任务

4.1 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：☆☆

Semaphore ['sɛməˌfɔr] 信号量，是JUC包下的一个工具类，我们可以通过其限制执行的线程数量，达到限流的效果

当一个线程执行时先通过其方法进行获取许可操作，获取到许可的线程继续执行业务逻辑，当线程执行完成后进行释放许可操作，未获取达到许可的线程进行等待或者直接结束。

Semaphore两个重要的方法

lsemaphore.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）

lsemaphore.release()：释放一个信号量，此时信号量个数+1

线程任务类：

```java
public class SemaphoreCase {
    public static void main(String[] args) {
        // 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();
        }
    }

}
```

5.其他

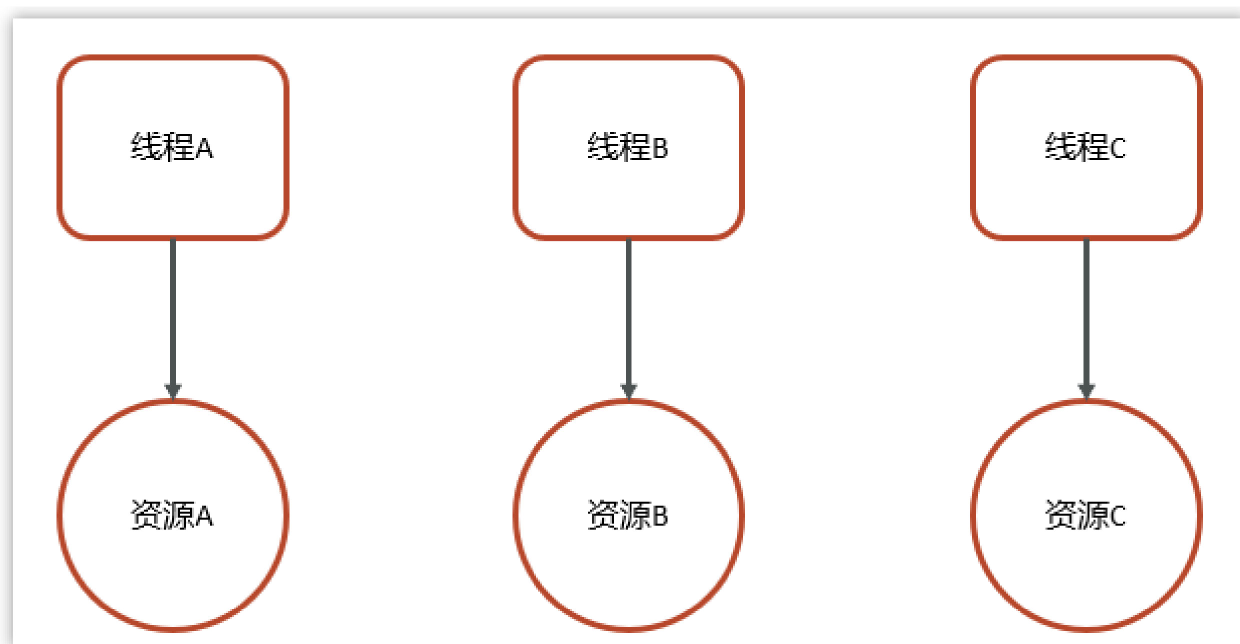
5.1 谈谈你对ThreadLocal的理解

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

5.1.1 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。



5.1.2 ThreadLocal基本使用

三个主要方法：

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

```java
public class ThreadLocalTest {
    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();
    }

}
```

5.1.3 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
  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
```

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

在ThreadLocal中有一个内部类叫做ThreadLocalMap，类似于HashMap

ThreadLocalMap中有一个属性table数组，这个是真正存储数据的位置

set方法

```java
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);
}
```

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

```java
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);
}
```

get方法/remove方法

```java
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();
}
```

```java
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);
}
```

5.1.4 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，就会回收到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);
```

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

在使用ThreadLocal的时候，强烈建议：**务必手动remove**

6 真实面试还原

6.1 线程的基础知识

> **面试官**：聊一下并行和并发有什么区别？
>
> **候选人**：
>
> 是这样的~~
>
> 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
>
> ---
>
> **面试官**：说一下线程和进程的区别？
>
> **候选人**：
>
> 嗯，好~
>
> - 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)
>
> ---
>
> **面试官**：如果在java中创建线程有哪些方式？
>
> **候选人**：
>
> 在java中一共有四种常见的创建方式，分别是：继承Thread类、实现runnable接口、实现Callable接口、线程池创建线程。通常情况下，我们项目中都会采用线程池的方式创建线程。
>
> **面试官**：好的，刚才你说的runnable 和 callable 两个接口创建线程有什么不同呢？
>
> **候选人**：
>
> 是这样的~
>
> 最主要的两个线程一个是有返回值，一个是没有返回值的。

Runnable 接口run方法无返回值；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

还有一个就是，他们异常处理也不一样。Runnable接口run方法只能抛出运行时异常，也无法捕获处理；Callable接口call方法允许抛出异常，可以获取异常信息

在实际开发中，如果需要拿到执行的结果，需要使用Callalbe接口创建线程，调用FutureTask.get()得到可以得到返回值，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。

面试官：线程包括哪些状态，状态之间是如何变化的？

候选人：

在JDK中的Thread类中的枚举State里面定义了6中线程的状态分别是：新建、可运行、终结、阻塞、等待和有时限等待六种。

关于线程的状态切换情况比较多。我分别介绍一下

当一个线程对象被创建，但还未调用 start 方法时处于**新建**状态，调用了 start 方法，就会由**新建**进入**可运行**状态。如果线程内代码已经执行完毕，由**可运行**进入**终结**状态。当然这些是一个线程正常执行情况。

如果线程获取锁失败后，由**可运行**进入 Monitor 的阻塞队列**阻塞**，只有当持锁线程释放锁时，会按照一定规则唤醒阻塞队列中的**阻塞**线程，唤醒后的线程进入**可运行**状态

如果线程获取锁成功后，但由于条件不满足，调用了 wait() 方法，此时从**可运行**状态释放锁**等待**状态，当其它持锁线程调用 notify() 或 notifyAll() 方法，会恢复为**可运行**状态

还有一种情况是调用 sleep(long) 方法也会从**可运行**状态进入**有时限等待**状态，不需要主动唤醒，超时时间到自然恢复为**可运行**状态

面试官：嗯，好的，刚才你说的线程中的 wait 和 sleep方法有什么不同呢？

候选人：

它们两个的相同点是都可以让当前线程暂时放弃 CPU 的使用权，进入阻塞状态。

不同点主要有三个方面：

第一：方法归属不同

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

第二：线程醒来时机不同

线程执行 sleep(long) 会在等待相应毫秒后醒来，而 wait() 需要被 notify 唤醒，wait() 如果不唤醒就一直等下去

第三：锁特性不同

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（相当于我放弃 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（相当于我放弃 cpu，你们也用不了）

面试官：好的，我现在举一个场景，你来分析一下怎么做，新建 T1、T2、T3 三个线程，如何保证它们按顺序执行？

候选人：

嗯~~，我思考一下 （适当的思考或想一下属于正常情况，脱口而出反而太假[背诵痕迹]）

可以这么做，在多线程中有多种方法让线程按特定顺序执行，可以用线程类的**join()**方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。

比如说：

使用join方法，T3调用T2，T2调用T1，这样就能确保T1就会先完成而T3最后完成

面试官：在我们使用线程的过程中，有两个方法。线程的 run()和 start()有什么区别？

候选人：

start方法用来启动线程，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方法封装了要被线程执行的代码，可以被调用多次。

面试官：那如何停止一个正在运行的线程呢？

候选人：

有三种方式可以停止线程

第一：可以使用退出标志，使线程正常退出，也就是当run方法完成后线程终止，一般我们加一个标记

第二：可以使用线程的stop方法强行终止，不过一般不推荐，这个方法已作废

第三：可以使用线程的interrupt方法中断线程，内部其实也是使用中断标志来中断线程

我们项目中使用的话，建议使用第一种或第三种方式中断线程

6.2 线程中并发锁

面试官：讲一下synchronized关键字的底层原理？

候选人：

嗯~~好的，

synchronized 底层使用的JVM级别中的Monitor 来决定当前线程是否获得了锁，如果某一个线程获得了锁，在没有释放锁之前，其他线程是不能或得到锁的。synchronized 属于悲观锁。

synchronized 因为需要依赖于JVM级别的Monitor ，相对性能也比较低。

面试官：好的，你能具体说下Monitor 吗？

候选人：

monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因

monitor内部维护了三个变量

- WaitSet：保存处于Waiting状态的线程
- EntryList：保存处于Blocked状态的线程
- Owner：持有锁的线程

只有一个线程获取到的标志就是在monitor中设置成功了Owner，一个monitor中只能有一个Owner

在上锁的过程中，如果有其他线程也来抢锁，则进入EntryList 进行阻塞，当获得锁的线程执行完了，释放了锁，就会唤醒EntryList 中等待的线程竞争锁，竞争的时候是非公平的。

面试官：好的，那关于synchronized 的锁升级的情况了解吗？

候选人：

嗯，知道一些（要谦虚）

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。

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

面试官：好的，刚才你说了synchronized它在高并发量的情况下，性能不高，在项目该如何控制使用锁呢？

候选人：

嗯，其实，在高并发下，我们可以采用ReentrantLock来加锁。

面试官：嗯，那你说下ReentrantLock的使用方式和底层原理？

候选人：

好的，

ReentrantLock是一个可重入锁:，调用 lock 方 法获取了锁之后，再次调用lock，是不会再阻塞，内部直接增加重入次数 就行了，标识这个线程已经重复获取一把锁而不需要等待锁的释放。

ReentrantLock是属于juc报下的类，属于api层面的锁，跟synchronized一样，都是悲观锁。通过lock()用来获取锁，unlock()释放锁。

它的底层实现原理主要利用**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高。

面试官：好的，刚才你说了CAS和AQS，你能介绍一下吗？

候选人：

好的。

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

AQS的话，其实就一个jdk提供的类AbstractQueuedSynchronizer，是阻塞式锁和相关的同步器工具的框架。

内部有一个属性 state 属性来表示资源的状态，默认state等于0，表示没有获取锁，state等于1的时候才标明获取到了锁。通过cas 机制设置 state 状态

在它的内部还提供了基于 FIFO 的等待队列，是一个双向列表，其中

- tail 指向队列最后一个元素
- head 指向队列中最久的一个元素

其中我们刚刚聊的ReentrantLock底层的实现就是一个AQS。

面试官：synchronized和Lock有什么区别？

候选人：

嗯~~，好的，主要有三个方面不太一样

第一，语法层面

- synchronized 是关键字，源码在 jvm 中，用 c++ 语言实现，退出同步代码块锁会自动释放
- Lock 是接口，源码由 jdk 提供，用 java 语言实现，需要手动调用 unlock 方法释放锁

第二，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，例如获取等待状态、公平锁、可打断、可超时、多条件变量，同时Lock 可以实现不同的场景，如 ReentrantLock，ReentrantReadWriteLock

第三，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

统合来看，需要根据不同的场景来选择不同的锁的使用。

面试官：死锁产生的条件是什么？

候选人：

嗯，是这样的，一个线程需要同时获取多把锁，这时就容易发生死锁，举个例子来说：

t1 线程获得A对象锁，接下来想获取B对象的锁

t2 线程获得B对象锁，接下来想获取A对象的锁

这个时候t1线程和t2线程都在互相等待对方的锁，就产生了死锁

面试官：那如果产出了这样的，如何进行死锁诊断？

候选人：

这个也很容易，我们只需要通过jdk自动的工具就能搞定

我们可以先通过jps来查看当前java程序运行的进程id

然后通过jstack来查看这个进程id，就能展示出来死锁的问题，并且，可以定位代码的具体行号范围，我们再去找到对应的代码进行排查就行了。

面试官：请谈谈你对 volatile 的理解

候选人：

嗯~~

volatile 是一个关键字，可以修饰类的成员变量、类的静态成员变量，主要有两个功能

第一：保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的,volatile关键字会强制将修改的值立即写入主存。

第二： 禁止进行指令重排序，可以保证代码执行有序性。底层实现原理是，添加了一个**内存屏障**，通过插入内存屏障禁止在内存屏障**前后**的指令执行重排序优化

本文作者：接《集合相关面试题》

面试官：那你能聊一下ConcurrentHashMap的原理吗？

候选人：

嗯好的，

ConcurrentHashMap 是一种线程安全的高效Map集合，jdk1.7和1.8也做了很多调整。

- JDK1.7的底层采用是**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。

在jdk1.7中 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和HashMap类似，是一 种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构 的元素，每个 Segment 守护着一个HashEntry数组里的元素，当对 HashEntry 数组的数据进行修 改时，必须首先获得对应的 Segment的锁。

Segment 是一种可重入的锁 ReentrantLock，每个 Segment 守护一个 HashEntry 数组里得元 素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 锁

在jdk1.8中的ConcurrentHashMap 做了较大的优化，性能提升了不少。首先是它的数据结构与jdk1.8的hashMap数据结构完全一致。其次是放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保 证并发安全进行实现，synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲 突，就不会产生并发，效率得到提升

6.3 线程池

面试官：线程池的种类有哪些？

候选人：

嗯！是这样

在jdk中默认提供了4中方式创建线程池

第一个是：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。

第二个是：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。

第三个是：newScheduledThreadPool 创建一个定长线程池，支持定时及周期任务执行。

第四个是：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, LIFO, 优先级)执行。

面试官：线程池的核心参数有哪些？

候选人：

在线程池中一共有7个核心参数：

1. corePoolSize 核心线程数目 - 池中会保留的最多线程数
2. maximumPoolSize 最大线程数目 - 核心线程+救急线程的最大数目
3. 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
4. unit 时间单位 - 救急线程的生存时间单位，如秒、毫秒等
5. 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
6. 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
7. 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

在拒绝策略中又有4中拒绝策略

当线程数过多以后，第一种是抛异常、第二种是由调用者执行任务、第三是丢弃当前的任务，第四是丢弃最早排队任务。默认是直接抛异常。

面试官：如何确定核心线程池呢？

候选人：

是这样的，我们公司当时有一些规范，为了减少线程上下文的切换，要根据当时部署的服务器的CPU核数来决定，我们规则是：CPU核数+1就是最终的核心线程数。

面试官：线程池的执行原理知道吗？

候选人：

嗯~，它是这样的

首先判断线程池里的核心线程是否都在执行任务，如果不是则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队 列里。如果工作队列满了，则判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任 务。如果已经满了，则交给拒绝策略来处理这个任务。

面试官：为什么不建议使用Executors创建线程池呢？

> **候选人**：
>
> 好的，其实这个事情在阿里提供的最新开发手册《Java开发手册-嵩山版》中也提到了
>
> 主要原因是如果使用Executors创建线程池的话，它允许的请求队列默认长度是Integer.MAX_VALUE，这样的话，有可能导致堆积大量的请求，从而导致OOM（内存溢出）。
>
> 所以，我们一般推荐使用ThreadPoolExecutor来创建线程池，这样可以明确规定线程池的参数，避免资源的耗尽。

6.4 线程使用场景问题

> **面试官**：如果控制某一个方法允许并发访问线程的数量？
>
> **候选人**：
>
> 嗯~~，我想一下
>
> 在jdk中提供了一个Semaphore[seməfɔːr]类（信号量）
>
> 它提供了两个方法，semaphore.acquire() 请求信号量，可以限制线程的个数，是一个正数，如果信号量是-1,就代表已经用完了信号量，其他线程需要阻塞了
>
> 第二个方法是semaphore.release()，代表是释放一个信号量，此时信号量的个数+1
>
> **面试官**：好的，那该如何保证Java程序在多线程的情况下执行安全呢？
>
> **候选人**：
>
> 嗯，刚才讲过了导致线程安全的原因，如果解决的话，jdk中也提供了很多的类帮助我们解决多线程安全的问题，比如：
>
> - JDK Atomic开头的原子类、synchronized、LOCK，可以解决原子性问题
> - synchronized、volatile、LOCK，可以解决可见性问题
> - Happens-Before 规则可以解决有序性问题

面试官：你在项目中哪里用了多线程？

候选人：

嗯~~，我想一下当时的场景[根据自己简历上的模块设计多线程场景]

参考场景一：

es数据批量导入

在我们项目上线之前，我们需要把数据量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），如果分批执行的话，耗时也太久了。所以，当时我就想到可以使用线程池的方式导入，利用CountDownLatch+Future来控制，就能大大提升导入的时间。

参考场景二：

在我做那个xx电商网站的时候，里面有一个数据汇总的功能，在用户下单之后需要查询订单信息，也需要获得订单中的商品详细信息（可能是多个），还需要查看物流发货信息。因为它们三个对应的分别三个微服务，如果一个一个的操作的话，互相等待的时间比较长。所以，我当时就想到可以使用线程池，让多个线程同时处理，最终再汇总结果就可以了，当然里面需要用到Future来获取每个线程执行之后的结果才行

参考场景三：

《黑马头条》项目中使用的

我当时做了一个文章搜索的功能，用户输入关键字要搜索文章，同时需要保存用户的搜索记录（搜索历史），这块我设计的时候，为了不影响用户的正常搜索，我们采用的异步的方式进行保存的，为了提升性能，我们加入了线程池，也就说在调用异步方法的时候，直接从线程池中获取线程使用

6.5 其他

面试官：谈谈你对ThreadLocal的理解

候选人：

嗯，是这样的~~

ThreadLocal 主要功能有两个，第一个是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，第二个是实现了线程内的资源共享

面试官：好的，那你知道ThreadLocal的底层原理实现吗？

候选人：

嗯，知道一些~

在ThreadLocal内部维护了一个一个 ThreadLocalMap 类型的成员变量，用来存储资源对象

当我们调用 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

当调用 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

当调用 remove 方法，就是以 ThreadLocal 自己作为 key，移除当前线程关联的资源值

面试官：好的，那关于ThreadLocal会导致内存溢出这个事情，了解吗？

候选人：

嗯，我之前看过源码，我想一下~~

是应为ThreadLocalMap 中的 key 被设计为弱引用，它是被动的被GC调用释放key，不过关键的是只有key可以得到内存释放，而value不会，因为value是一个强引用。

在使用ThreadLocal 时都把它作为静态变量（即强引用），因此无法被动依靠 GC 回收，建议主动的remove 释放 key，这样就能避免内存溢出。