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 并行和并发有什么区别？

> 难易程度：☆
>
> 出现频率：☆

参考回答：

- 并行：多个任务在计算机中同时执行
- 并发：多个任务在计算机中交替执行

1.2 线程和进程的区别？

> 难易程度：☆
>
> 出现频率：☆☆

进程是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至少有一个线程，但**一个进程一般有多个线程**。

进程在运行过程中，需要拥有独立的内存单元，否则如果申请不到，就会挂起。而多个线程能共享内存资源，这样就能降低运行的门槛，从而效率更高。

线程是cpu调度和分派的基本单位，在实际开发过程中，一般是考虑多线程并发。

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 MyRunnable 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 MyRunnable()) ;

        // 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
2. Runnable接口run方法只能抛出运行时异常，且无法捕获处理；Callable接口call方法允许抛出异常，可以获取异常信息
3. 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。

1.5 线程包括哪些状态，状态之间是如何变化的

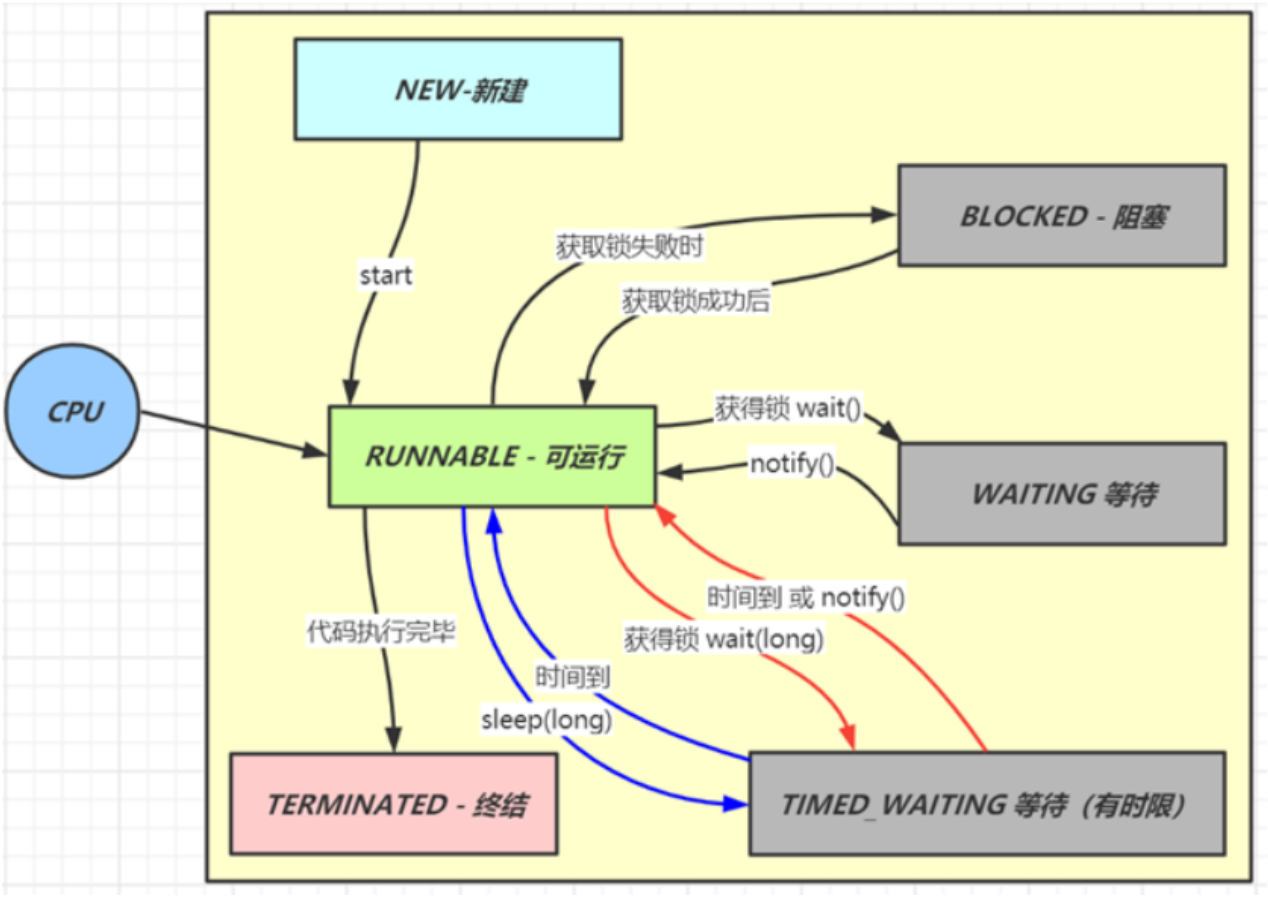
> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

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

1.6 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）

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

1.9 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。

1.10 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 MyThread 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
        MyThread t1 = new MyThread() ;
        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 MyThread 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
        MyThread t1 = new MyThread() ;
        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。

```
public class MyThread 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
        MyThread t1 = new MyThread() ;
        t1.start();
```

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

        // 调用interrupt方法
        t1.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：☆☆☆

如下加锁的代码

```
public class SynchronizedDemo {

    public void method() {

        synchronized (this) {
            System.out.println("synchronized 代码块");
        }

    }

}
```

通过javap查看字节码文件信息，如下所示：

通过javap(javap -v xxx.class)查看字节码文件信息

```
PS E:\idea-edu-workplace\interview\out\production\interview\com\itheima\interview\syn\demo01> javap -c .\SynchronizedDemo.class
Compiled from "SynchronizedDemo.java"
public class com.itheima.interview.syn.demo01.SynchronizedDemo {
  public com.itheima.interview.syn.demo01.SynchronizedDemo();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void method();
    Code:
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String synchronized 代码块
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
      21: athrow
      22: return
    Exception table:
       from    to  target type
           4    14    17   any
          17    20    17   any

  public static void main(java.lang.String[]);
    Code:
       0: return
}
```

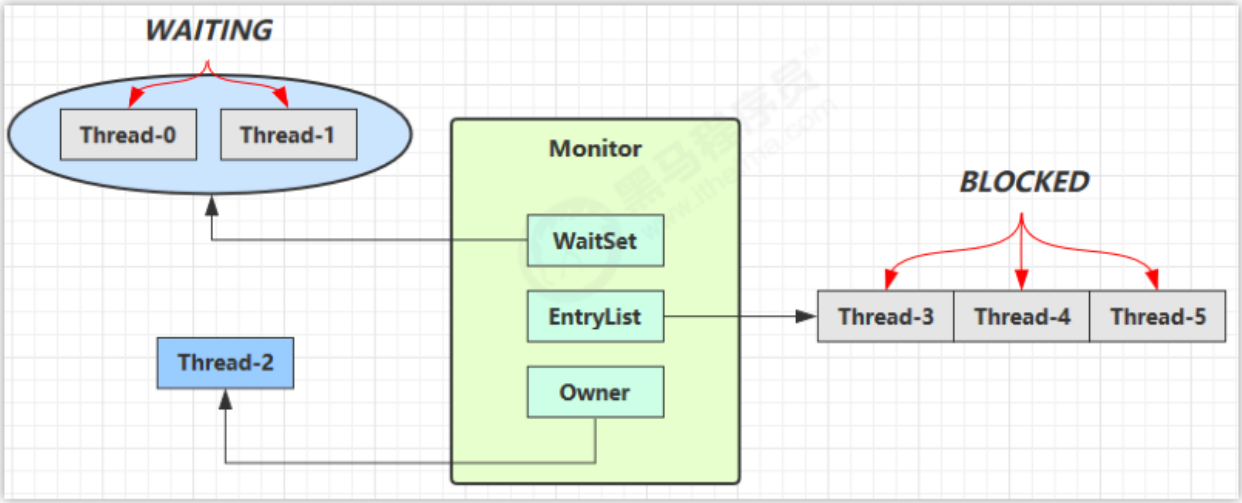
从上面我们可以看出：

synchronized 同步语句块的实现使用的是 **monitorenter** 和 **monitorexit** 指令

- monitorenter 指令指向同步代码块的开始位置
- monitorexit 指令则指明同步代码块的结束位置

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的结构和执行流程如下：



> 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

（1）刚开始 Monitor 中 Owner 为 null

（2）当 Thread-2 执行 synchronized(obj) 就会将 Monitor 的所有者 Owner 置为 Thread-2，Monitor中只能有一个 Owner

（3）在 Thread-2 上锁的过程中，如果 Thread-3，Thread-4，Thread-5 也来执行 synchronized(obj)，就会进入EntryList BLOCKED

（4）Thread-2 执行完同步代码块的内容，然后唤醒 EntryList 中等待的线程来竞争锁，竞争的时是非公平的图中 WaitSet 中的 Thread-0，Thread-1 是之前获得过锁，但条件不满足进入 WAITING 状态的线程

总结:

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。

----------进阶回答----------

在JDK6之前synchronized锁都属于重量级锁，因为底层都是使用Monitor 来实现的。但是在JDK6之后对synchronized做了升级优化，里面主要体现在：CAS 自旋、锁消除、锁膨胀、轻量级锁、偏向锁等。

在讲锁升级的过程之前，我们必须要知道一个对象布局，这个跟锁升级的过程息息相关。

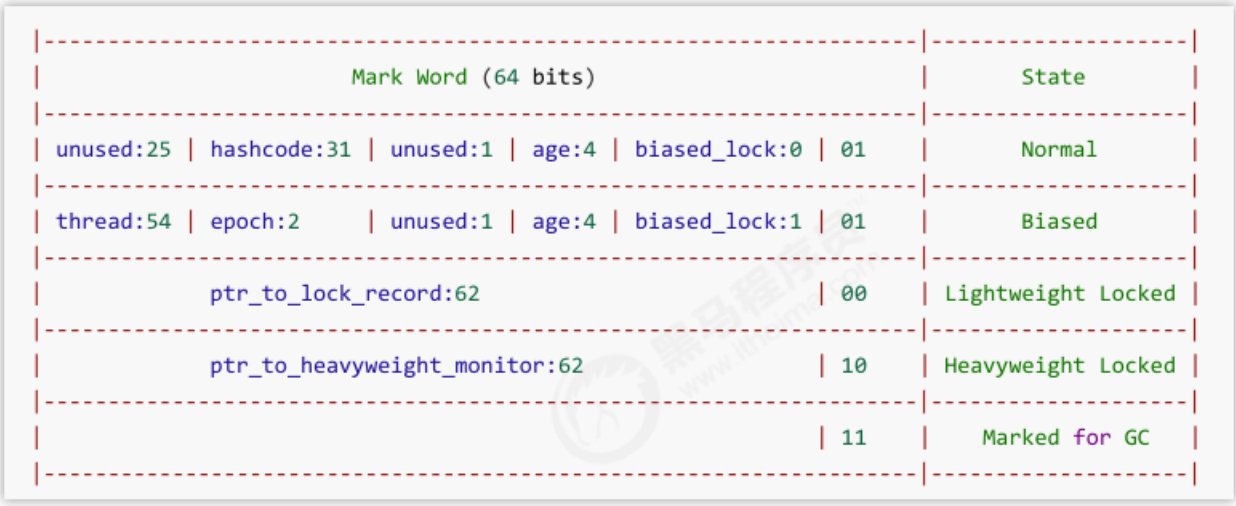
对象布局

在 Java 中，任何的对象实例的内存布局都分为**对象头**、**对象实例数据**和**对齐填充数据**三个部分，其中对象头又包括 **MarkWord** 和 **类型指针**。

组成部分	描述

组成部分	描述
对象头	对象头由Mark Word 和 一个指向一个类对象的指针组成。
对象实例数据	存放这个实例的一些属性信息，比如有的属性是基本类型，那就直接存储值；如果是对象类型，存放的就是一个指向对象的内存地址
对齐填充数据	主要是补齐作用，JVM对象的大小比如是8字节的整数倍，如果 （对象头 + 实例变量 ）不是8的整数倍，则通过对齐填充来补齐

对象头，如下：



1) 由上图得知**最后两位**，也就是**锁标志位**，分别标识处于不同的锁模式；**倒数第3位**是**偏向锁标志**

2）当我的**偏向锁标志是0**，**锁标志位是01**，也就是**最后3位是001**的时候，我表示无锁模式。作为**Mark Word**的我就是记录的数据就是对象的**hashcode** 和 **GC** 的年龄

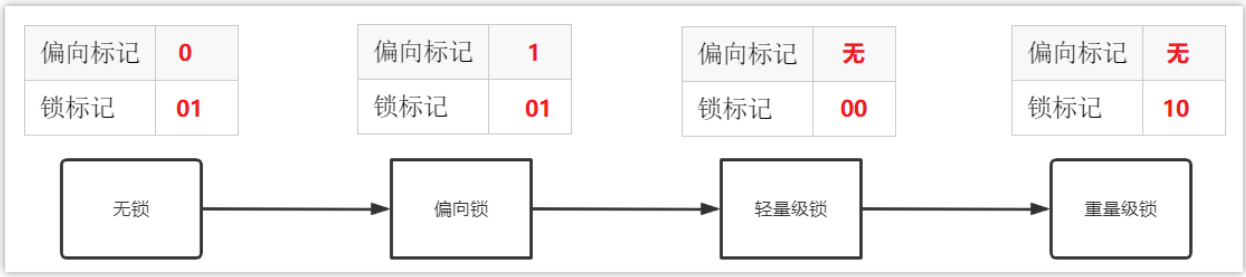
3） 当我的**偏向锁标志是1**，**锁标志是01**，也就是**最后三位是101**的时候，处于**偏向锁模式**，我作为**Mark Word**这个时候记录的数据就是**获取偏向锁的线程ID**、**Epoch**（偏向时间戳）、对象**GC**年龄

4）当我的**锁标志位是00**的时候，表示处于轻量级锁模式。我会把**锁记录放在加锁的线程的虚拟机栈空间中**，所以**这种情况下**，锁记录在哪个线程虚拟机栈中，就表示**所在线程就获取到了锁**。

5）**锁标志位是10**的时候，表示处于**重量级锁**模式，这个时候就说明竞争激烈了，处于重量级锁模式了，由于使用重量级加锁不是java的职责范围，是底层c++的monitor的职责，前面则保存monitor的地址。

这个是我作为Mark Word 记录的数据就是monitor的地址，有加锁的需求直接根据记录的这个地址找到monitor，找它加锁就好了。

总结如下：



锁的具体升级过程(通常情况下):

线程A向锁对象加偏向锁，线程B来了

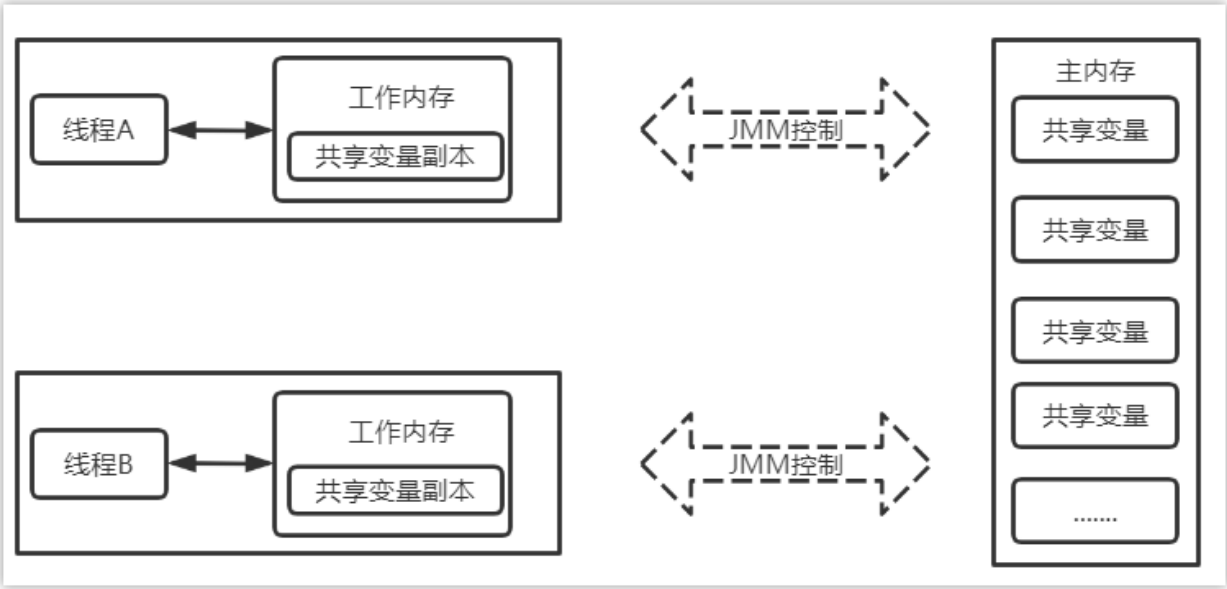
情况1：线程A已经不存活，线程B 获得锁时是轻量级
情况2：线程A存活，线程B 尝试获取锁，结果 A 拿着不放，这时锁升级为重量级
情况3：线程A存活，线程B 尝试获取锁，A在这之前释放了锁，二者没有交替，此时B获得锁，锁升级为轻量级
情况4：线程A不存活，并已经过了批量重偏向阈值，这时 B获得锁，锁还是偏向锁，偏向于B

偏向锁和轻量级锁，都是没有竞争。
偏向锁是只有一个线程使用锁，轻量级锁是两个线程交替使用锁。一旦有竞争，就会升级为重量级锁，升级重量级锁的过程中，会首先尝试自旋，自旋过程中，对方释放锁，这样就避免线程进入阻塞，如果自旋试了几次人家还持有锁，那么线程进入 entrylist 等待

2.2 JMM（Java 内存模型） 你谈谈

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：

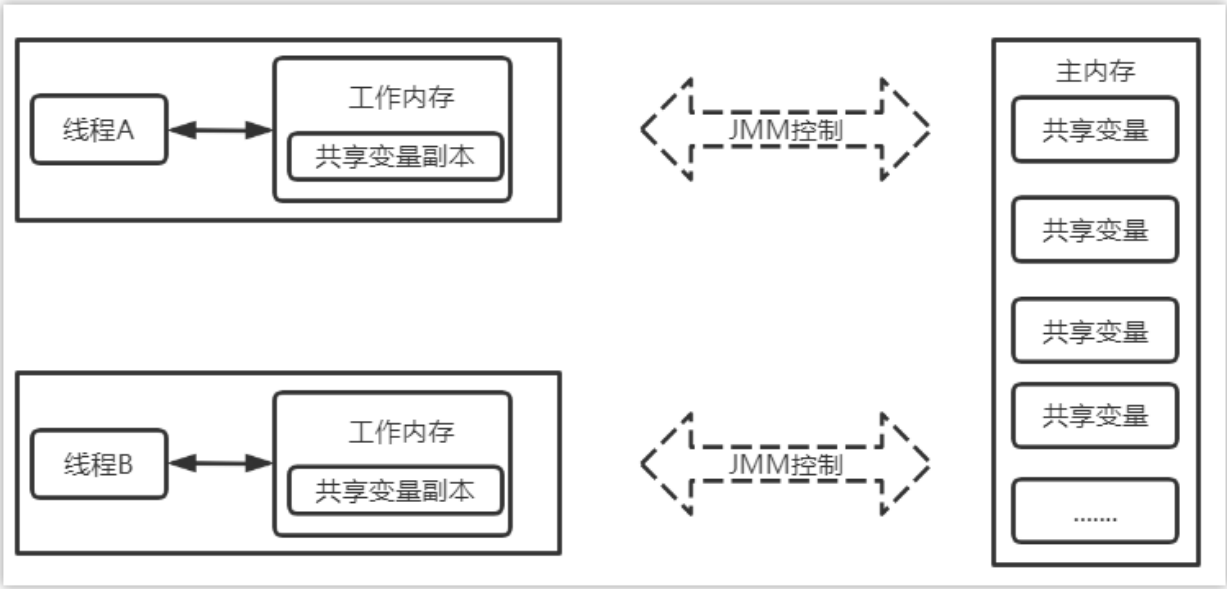
1. 所有的共享变量都存储于主内存(计算机的RAM)这里所说的变量指的是实例变量和类变量。不包含局部变量，因为局部变量是线程私有的，因此不存在竞争问题。
2. 每一个线程还存在自己的工作内存，线程的工作内存，保留了被线程使用的变量的工作副本。
3. 线程对变量的所有的操作(读，写)都必须在工作内存中完成，而不能直接读写主内存中的变量，不同线程之间也不能直接访问对方工作内存中的变量，线程间变量的值的传递需要通过主内存完成。

2.3 CAS 你知道吗？

> 难易程度：☆☆☆
>
> 出现频率：☆☆

CAS的全称是： Compare And Swap(比较再交换); 是现代CPU广泛支持的一种对内存中的共享数据进行操作的一种特殊指令。CAS可以将read-modify-write转换为原子操作，这个原子操作直接由CPU保证。

CAS有3个操作数：**内存值V，旧的预期值A，要修改的新值B**。当且仅当旧预期值A和内存值V相同时，将内存值V修改为B并返回true，否则什么都不做，并返回false。



例：

①：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

②：线程1操作：V：int a = 100，A：int a = 100，B：修改后的值：int a = 90(100-10)

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90更新到主内存中

③：线程2操作：V：int a = 100，A：int a = 100，B：修改后的值：int a = 90(100-10)

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90)

不相等，则线程2更新失败

> ④：自旋操作
>
> 线程2再次读取主内存的内容：V：int a = 90，A：int a = 90，B：修改后的值：int a = 80(90-10)
>
> 线程2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80更新到主内存中

2.4 什么是AQS？

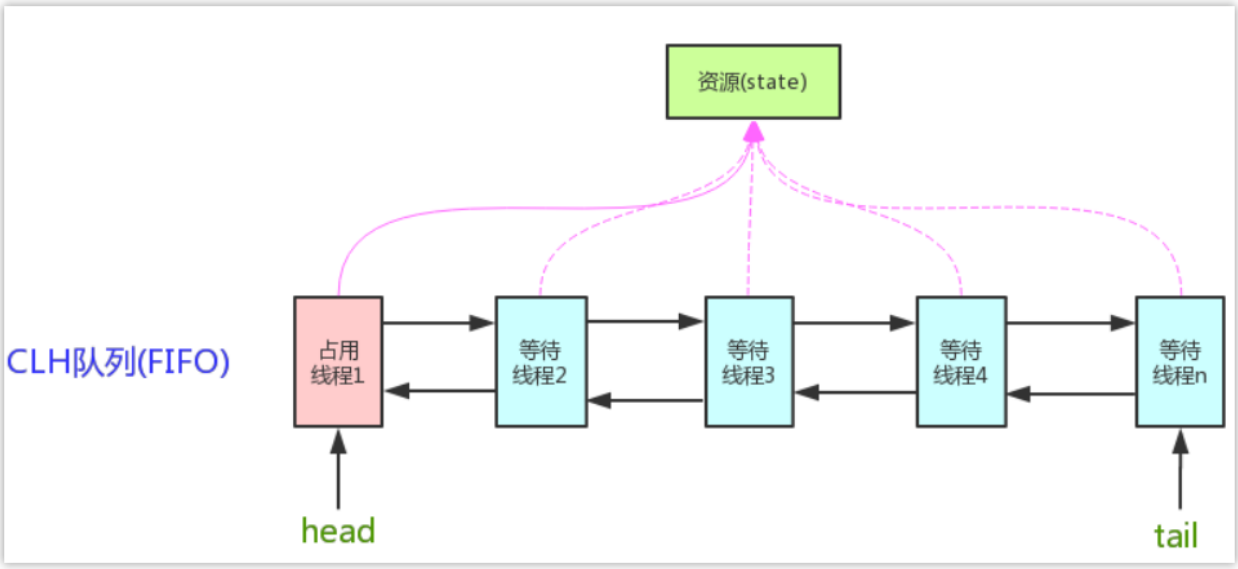
> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

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

特点：

- 用 state 属性来表示资源的状态（分独占模式和共享模式），子类需要定义如何维护这个状态，控制如何获取锁和释放锁
  - getState - 获取 state 状态
  - setState - 设置 state 状态
  - compareAndSetState - cas 机制设置 state 状态
  - 独占模式是只有一个线程能够访问资源，而共享模式可以允许多个线程访问资源
- 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

AQS内部维护着一个FIFO队列，该队列就是CLH同步队列，遵循FIFO原则（ First Input First Output先进先出）。CLH同步队列是一个FIFO双向队列，AQS依赖它来完成同步状态的管理。



当前线程如果获取同步状态失败时，AQS则会将当前线程已经等待状态等信息构造成一个节点（Node）并将其加入到CLH同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点唤醒（公平锁），使其再次尝试获取同步状态。

- tail 指向队列最后一个元素
- head 指向队列中最久的一个元素

2.5 ReentrantLock的实现原理

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

重入锁: 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，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，比如递归调用

（1）lock锁用法

```
package com.itheima.basic;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockTest {

    static Lock  lock  = new ReentrantLock();
```

```
    public static void method1(){
        try{
            lock.lock();
            //进入method2方法
            method2();
            //可能会出现线程安全的操作
            System.out.println("method1...");
        }finally{
            //尽量在finally中释放锁
            lock.unlock();
        }
    }

    public static void method2(){
        try{
            lock.lock();
            //可能会出现线程安全的操作
            System.out.println("method2...");
        }finally{
            //尽量在finally中释放锁
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        method1();
    }
}
```

- lock()用来获取锁
- unlock()释放锁，最好在finally块中释放
- Lock 是 java.util.concurrent.locks.lock 包下的，是 api层面的锁

（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。

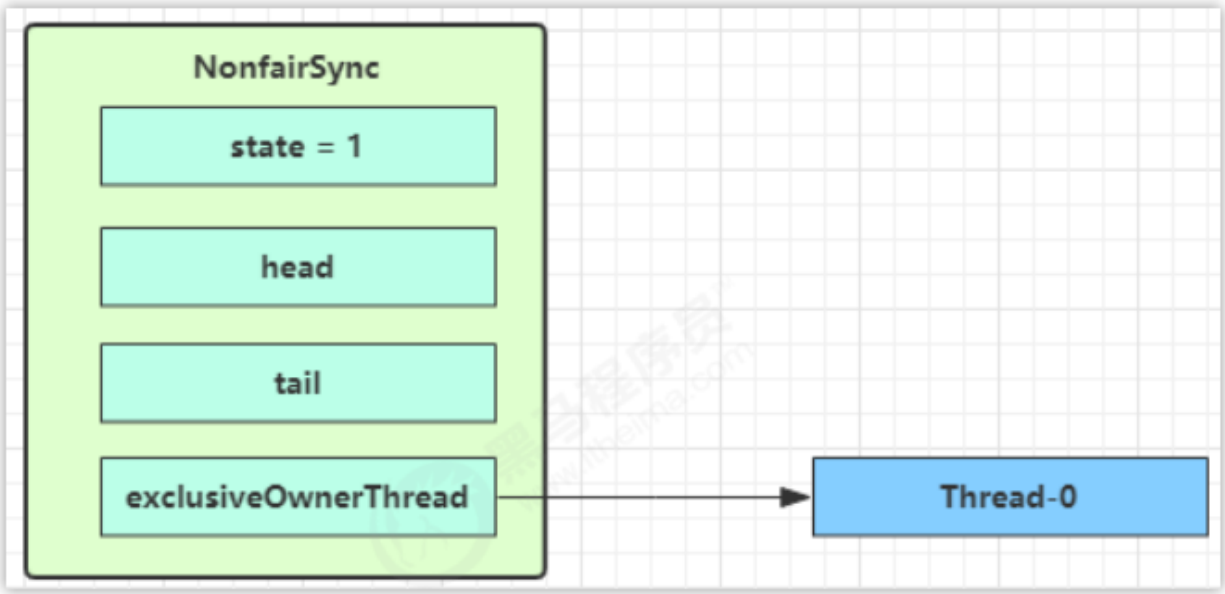
查看ReentrantLock源码中的构造方法：

```
public ReentrantLock() {
    //非公平锁
    sync = new NonfairSync();
}

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

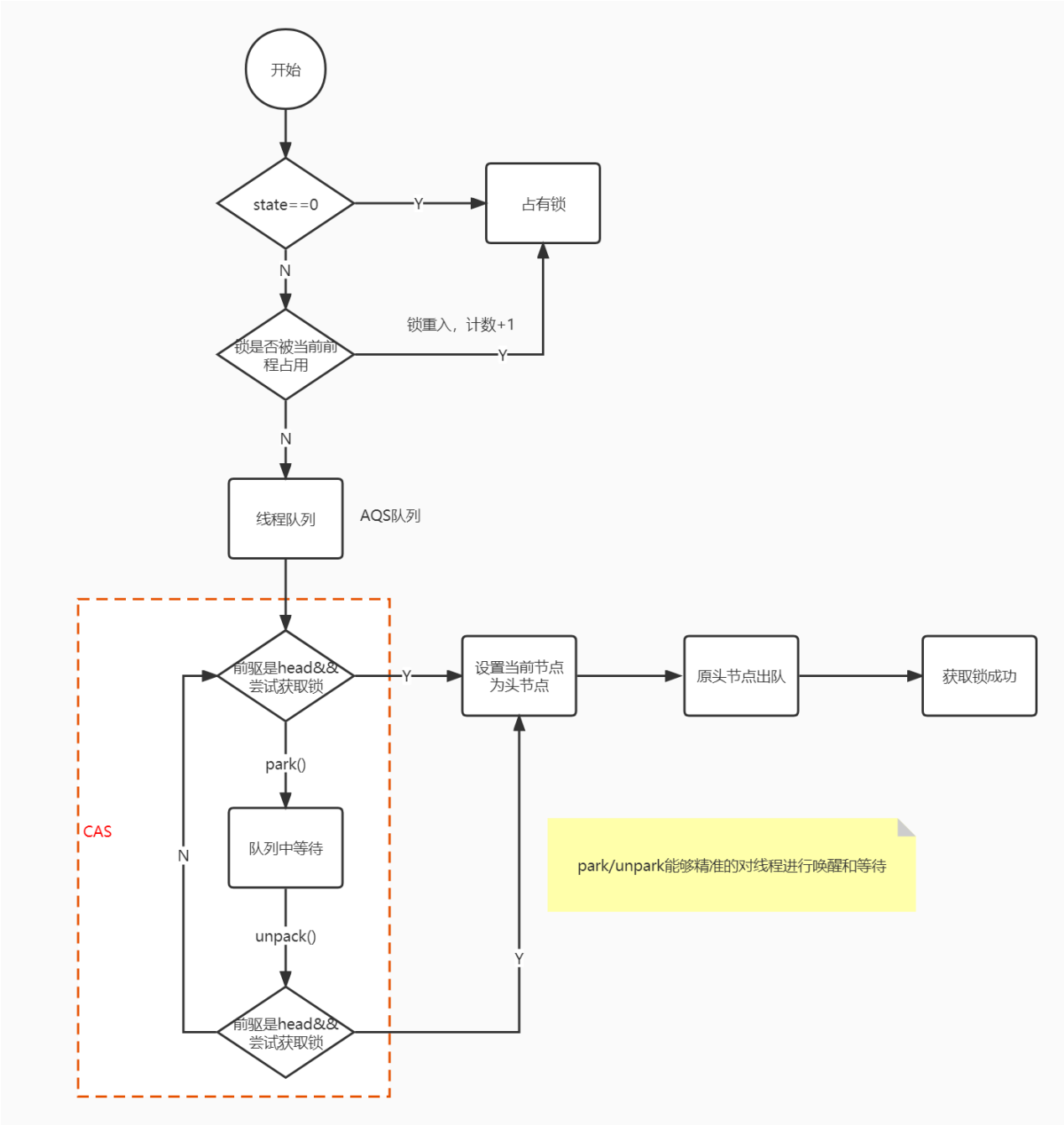
加锁解锁流程(非公平)

先从构造器开始看，默认为非公平锁实现

```
public ReentrantLock() {
    //非公平锁
    sync = new NonfairSync();
}
```



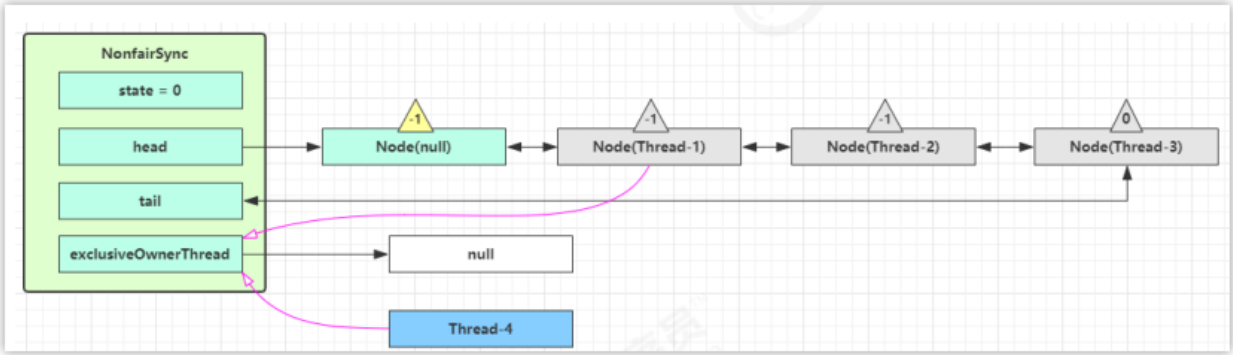
NonfairSync继承自 AQS，也就说底层主要实现也是基于AQS



非公平锁体现

当某一个线程释放锁之后，会从AQS维护的双向队列中激活一个线程去抢锁（下图中的Thread-1），这个时候同时有一个新的线程(下图中的Thread-4)抢锁，如果不巧又被 Thread-4 占了先

- Thread-4 被设置为 exclusiveOwnerThread(锁拥有者)，state = 1
- Thread-1 再次进入 acquireQueued 流程，获取锁失败，重新进入 park 阻塞



公平锁体现

上图的情况就变为，Thread-4会乖乖去队列中等待被激活。Thread-1会抢到锁

2.6 synchronized和Lock有什么区别？加锁的方式有哪些？

> 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

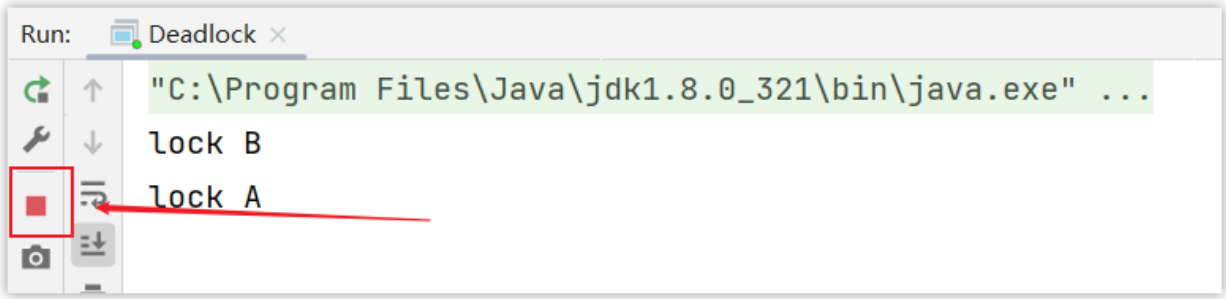
> 例如：
>
> t1 线程获得A对象锁，接下来想获取B对象的锁
>
> t2 线程获得B对象锁，接下来想获取A对象的锁

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。

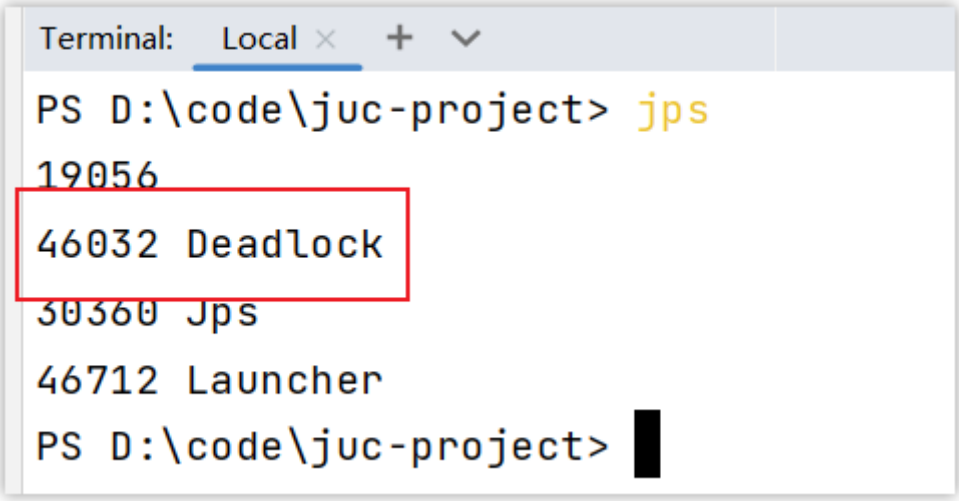
2.8 如何进行死锁诊断？

> 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
```

2.9 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：

① 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

```
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();

        new Thread(() -> {
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(stop);
        }).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 禁止进行指令重排序，可以保证有序性。

指令重排：计算机在执行程序时，为了提高性能，编译器和处理器常常会对指令重排。处理器在进行重排序时，必须要考虑指令之间的**数据依赖性**。如下代码：

```
public void mySort() {
    int x = 11;
    int y = 12;
    x = x + 5;
    y = x * x;
}
```

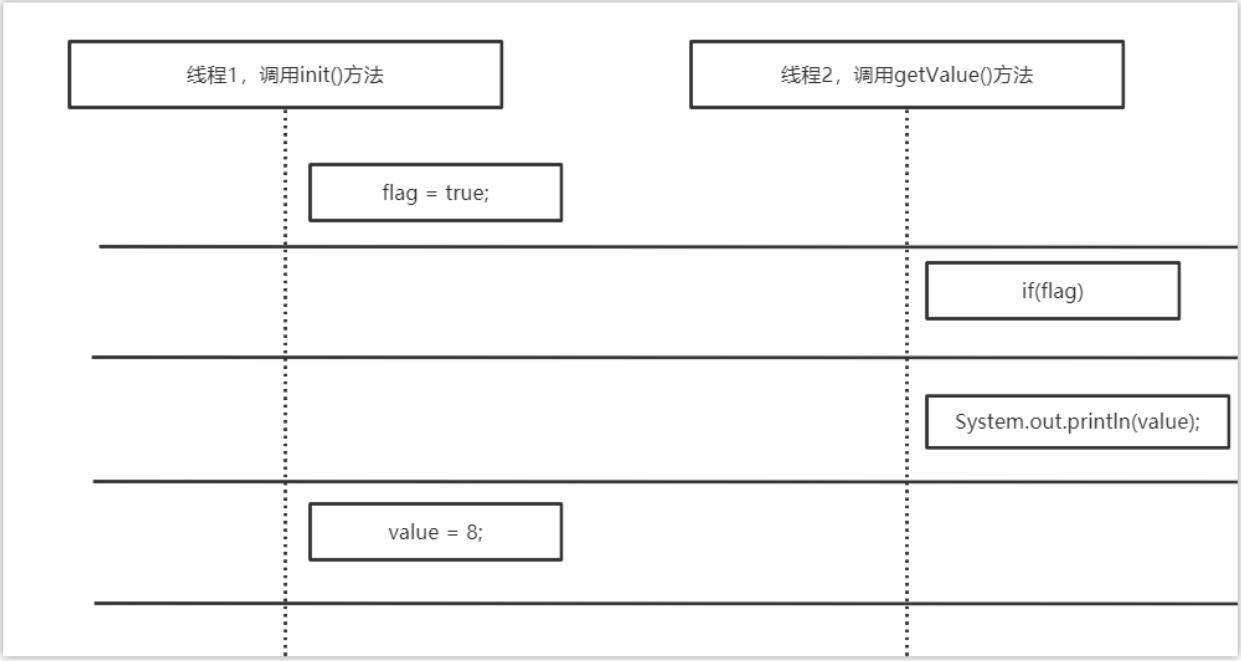
按照正常的顺序进行执行，那么执行顺序应该是：1 2 3 4 。但是如果发生了指令重排，那么此时的执行顺序可能是：① 1 3 2 4 ② 2 1 3 4 但是肯定不会出现：4 3 2 1这种顺序，因为处理器在进行重排时候，必须考虑到指令之间的数据依赖性。多线程环境里编译器和CPU指令优化根本无法识别多个线程之间存在的数据依赖性，比如说下面的程序代码如果两个方法在两个不同的线程里面调用就可能出现问题。

```
private static int value;
private static volatile boolean flag;

public static void init(){
    value=8;     //语句1
    flag=true;   //语句2
}

public static void getValue(){
    if(flag){
        System.out.println(value);
    }
}
```

根据上面代码，如果程序代码运行都是按顺序的，那么getValue() 中打印的value值必定是等于8的，不过如果init()方法经过了指令重排序，那么结果就不一定了。进行重排序后代码执行顺序可能如下。

```
flag=true;  //语句2
value=8;    //语句1
```

如果init()方法经过了指令重排序后，这个时候两个线程分别调用 init()和getValue()方法，那么就有可能出现下图的情况，导致最终打印出来的value数据等于0。



解决方案：使用volatile修饰flag，禁止指令重排。

原理说明：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

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

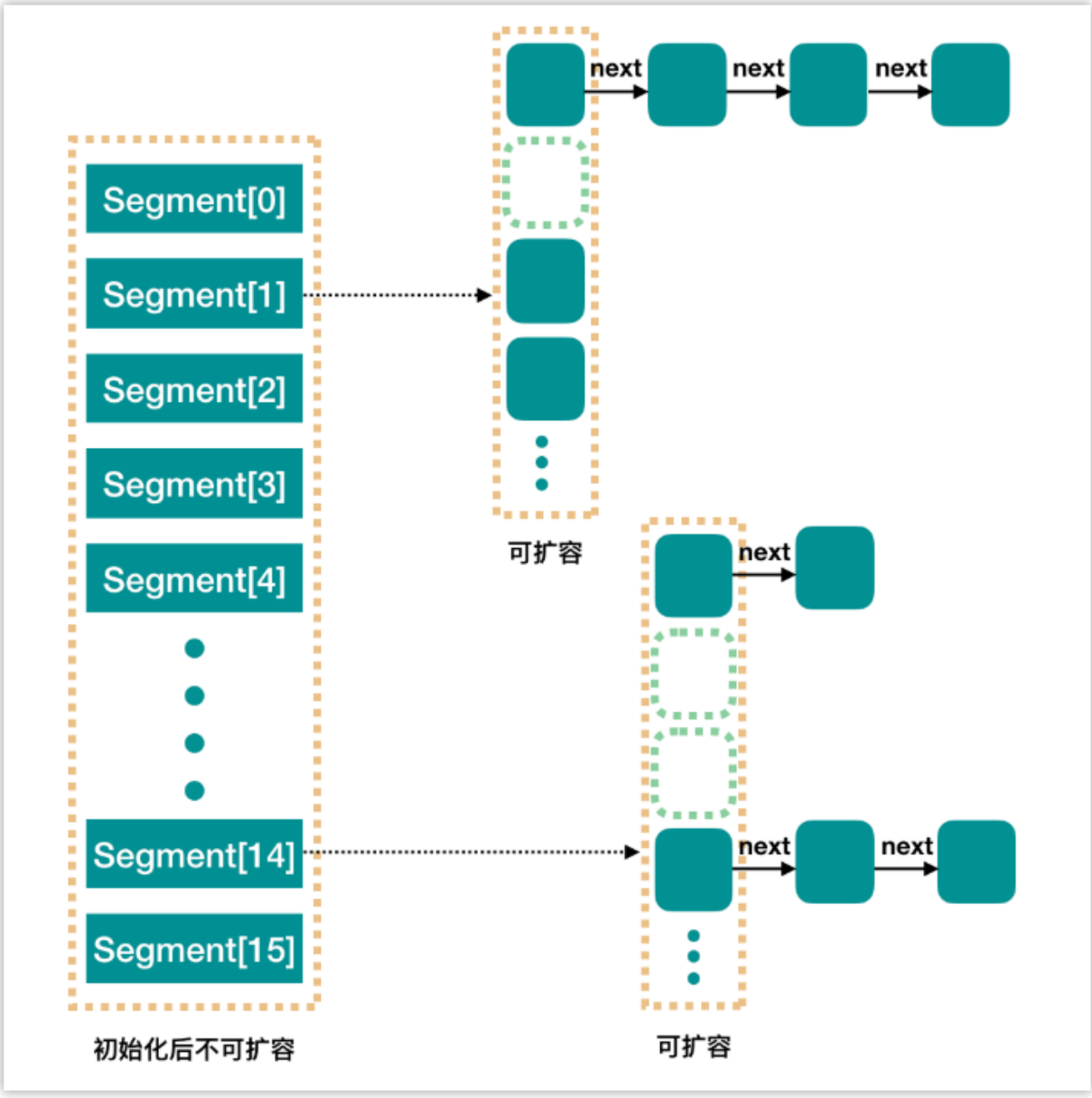
底层数据结构：

- JDK1.7的 ConcurrentHashMap 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

首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段 数据时，其他段的数据也能被其他线程访问。

在JDK1.7中，ConcurrentHashMap采用Segment + HashEntry的方式进行实现

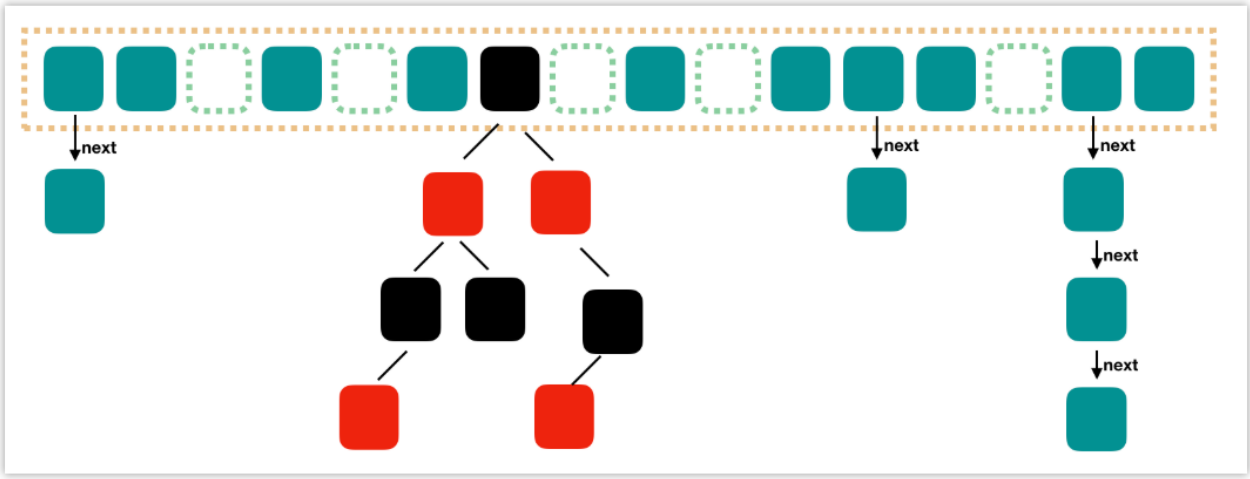


一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和HashMap类似，是一 种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构 的元素，每个 Segment 守护着一个HashEntry数组里的元素，当对 HashEntry 数组的数据进行修 改时，必须首先获得对应的 Segment的锁。

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。

JDK1.8

在JDK1.8中，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



3.线程池

3.1 线程池的种类

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

参考回答：

1. 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。
2. 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。
3. newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
4. 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。

3.2 线程池的核心参数

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

1. corePoolSize

线程池中核心线程的数量（也称为线程池的基本大小）。当提交一个任务时，线程池会新建一个线程来执行任务，直到当前线程数等于corePoolSize。如果调用了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有基本线程。

2. maximumPoolSize

线程池中允许的最大线程数。线程池的阻塞队列满了之后，如果还有任务提交，如果当前的线程数小于maximumPoolSize，则会新建线程来执行任务。注意，如果使用的是无界队列，该参数也就没有什么效果了。

3. keepAliveTime

线程空闲的时间。线程的创建和销毁是需要代价的。线程执行完任务后不会立即销毁，而是继续存活一段时间：keepAliveTime。默认情况下，该参数只有在线程数大于corePoolSize时才会生效。

4. unit

keepAliveTime的单位。TimeUnit

5. workQueue

用来保存等待执行的任务的BlockQueue阻塞队列，等待的任务必须实现Runnable接口。选择如下：

> ArrayBlockingQueue：基于数组结构的有界阻塞队列，FIFO。
> LinkedBlockingQueue：基于链表结构的有界阻塞队列，FIFO。
> PriorityBlockingQueue：具有优先级别的阻塞队列。
>
> 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。

6. threadFactory

用于设置创建线程的工厂。ThreadFactory的作用就是提供创建线程的功能的线程工厂。他是通过newThread()方法提供创建线程的功能，newThread()方法创建的线程都是“非守护线程”而且“线程优先级都是默认优先级”。

7. handler

RejectedExecutionHandler，线程池的拒绝策略。所谓**拒绝策略**，是指将任务添加到线程池中时，线程池拒绝该任务所采取的相应策略。当向线程池中提交任务时，如果此时线程池中的线程已经饱和了，而且阻塞队列也已经满了，则线程池会选择一种拒绝策略来处理该任务。

线程池提供了四种拒绝策略：

> AbortPolicy：直接抛出异常，默认策略；
>
> CallerRunsPolicy：用调用者所在的线程来执行任务；
>
> DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
> DiscardPolicy：直接丢弃任务；
>
> 当然我们也可以实现自己的拒绝策略，例如记录日志等等，实现RejectedExecutionHandler接口即可。

3.3 如何确定核心线程数

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

参考回答：

① 高并发、任务执行时间短的业务，线程池程数可以设置为CPU核数+1，减少线程上下文的切换

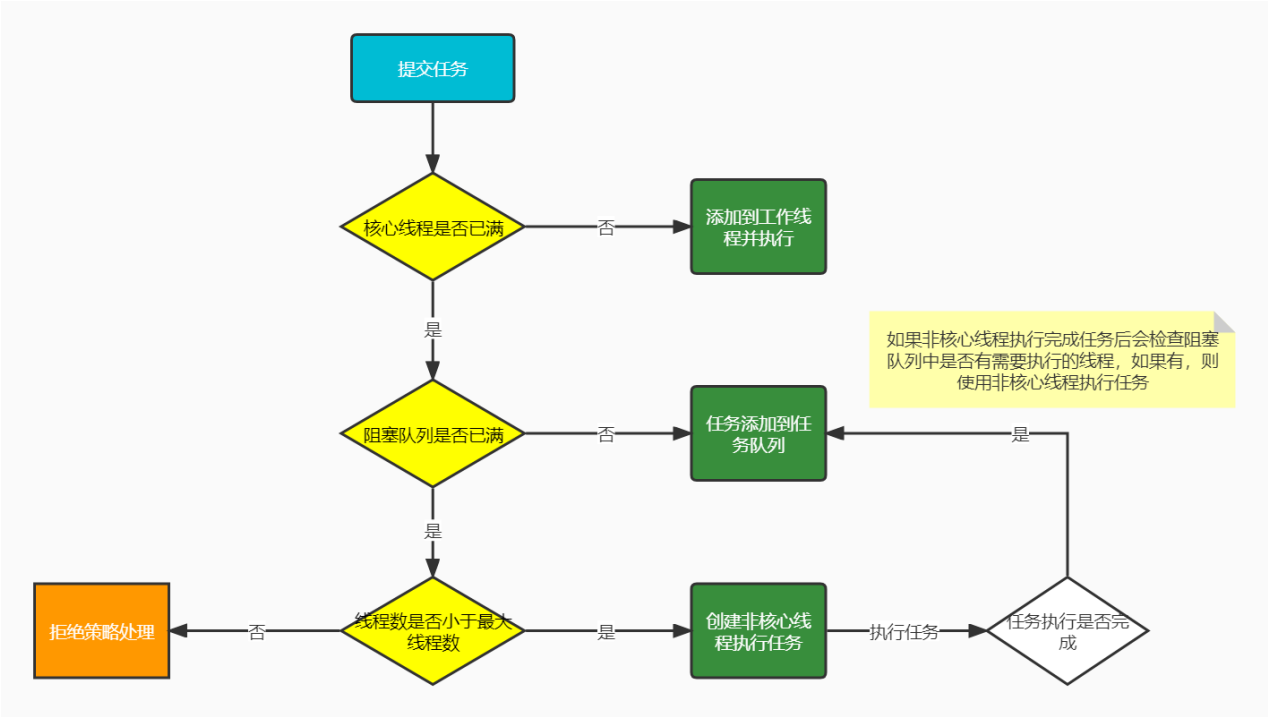
② 并发不高、任务执行时间长的业务要区分开看

- 假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务
- 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

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）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

3.4 线程池的执行原理知道嘛

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆



提交一个任务到线程池中，线程池的处理流程如下：

1. 判断线程池里的核心线程是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）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，则进入下个 流程。
2. 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，则进入下个流程。
3. 判断线程数是否小于最大线程数，如果是则创建临时线程直接执行任务，临时线程执行完任务后会检查阻塞队列中是否有等待的线程，如果有，则使用非核心线程最色队列中的任务；
4. 如果线程数大于了最大线程数，则走拒绝策略逻辑进行处理

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 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两个重要的方法就是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

线程任务类：

```
public class CarThreadRunnable implements Runnable {

    // 创建一个Semaphore对象,限制只允许2个线程获取到许可证
    private Semaphore semaphore = new Semaphore(2) ;

    @Override
    public void run() {                          // 这个run只允许2个线
程同时执行

        try {

            // 获取许可证
            semaphore.acquire();
```

```
            System.out.println(Thread.currentThread().getName() + "-
---->>正在经过十字路口");

            // 模拟车辆经过十字路口所需要的时间
            Random random = new Random();
            int nextInt = random.nextInt(7);
            TimeUnit.SECONDS.sleep(nextInt);

            System.out.println(Thread.currentThread().getName() + "-
---->>驶出十字路口");

            // 释放许可证
            semaphore.release();

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

    }

}
```

测试类：

```
public class SemaphoreDemo01 {

    public static void main(String[] args) {

        // 创建线程任务类对象
        CarThreadRunnable carThreadRunnable = new
CarThreadRunnable() ;

        // 创建5个线程对象，并启动。
        for(int x = 0 ; x < 5 ; x++) {
            new Thread(carThreadRunnable).start();
        }

    }

}
```

4.2 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：☆☆☆

CPU、内存、IO 设备的读写速度差异巨大，表现为 **CPU 的速度 > 内存的速度 > IO 设备**的速度。
程序的性能瓶颈在于速度最慢的 IO 设备的读写，也就是说当涉及到 IO 设备的读写，再怎么提升CPU 和内存的速度也是起不到提升性能的作用。

为了更好地利用 CPU 的高性能计算机体系结构，给 CPU 增加了缓存，均衡 CPU 和内存的速度差异操作系统，增加了进程与线程，分时复用 CPU，均衡 CPU 和 IO 设备的速度差异编译器，增加了指令执行重排序，更好地利用缓存，提高程序的执行速度

基于以上原因：

1、 CPU 缓存，在多核 CPU 的情况下，带来了可见性问题

可见性：一个线程对共享变量的修改，另一个线程能够立刻看到修改后的值

2、操作系统对当前执行线程的切换，带来了原子性问题

原子性：一个或多个指令在 CPU 执行的过程中不被中断的特性

3、编译器指令重排优化，带来了有序性问题

有序性：程序按照代码执行的先后顺序

4.3 Java程序中怎么保证多线程的执行安全

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

线程的安全性问题体现在：

- 原子性：一个或者多个操作在 CPU 执行的过程中不被中断的特性
- 可见性：一个线程对共享变量的修改，另外一个线程能够立刻看到
- 有序性：程序执行的顺序按照代码的先后顺序执行

导致原因：

- 缓存导致的可见性问题
- 线程切换带来的原子性问题
- 编译优化带来的有序性问题

解决办法：

- 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

4.4 线程池使用场景（CountDownLatch、Future等）

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

用来进行线程同步协作，等待所有线程完成倒计时。
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

```
public static void main(String[] args) throws InterruptedException {
    CountDownLatch latch = new CountDownLatch(3);
    new Thread(() -> {
        System.out.println("begin...");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        latch.countDown();
        System.out.println("end..." +latch.getCount());
```

```
        }).start();
        new Thread(() -> {
            System.out.println("begin...");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            latch.countDown();
            System.out.println("end..." +latch.getCount());
        }).start();
        new Thread(() -> {
            System.out.println("begin...");
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            latch.countDown();
            System.out.println("end..." +latch.getCount());
        }).start();
        System.out.println("waiting...");
        latch.await();
        System.out.println("wait end...");
    }
```

Future

Futrue可以监视目标线程调用call的情况，当你调用Future的get()方法以获得结果时，当前线程就开始阻塞，直接call方法结束返回结果

举个例子

在一个电商网站，当用户下单之后，我们一般会有一些后续操作，比如，需要查询订单，需要获得订单中的商品详细信息（可能是多个），需要查看物流发货信息

这三个操作，分别是不同的微服务中实现的，我们正常情况是，先调用查询订单，然后分别调用商品和物流，但这样的性能是不高的，因为接口需要等待所有调用成功之后才能返回数据，我们可以使用多线程解决这个性能问题，想要获取返回值，则需要使用Future

详细代码案例，如下：

```
public static void main(String[] args) throws InterruptedException {
    RestTemplate restTemplate = new RestTemplate();
    log.debug("begin");
    ExecutorService service = Executors.newCachedThreadPool();
    CountDownLatch latch = new CountDownLatch(4);
    Future<Map<String,Object>> f1 = service.submit(() -> {
        Map<String, Object> r =

 restTemplate.getForObject("http://localhost:8080/order/{1}",
Map.class, 1);
        return r;
    });
    Future<Map<String, Object>> f2 = service.submit(() -> {
        Map<String, Object> r =

 restTemplate.getForObject("http://localhost:8080/product/{1}",
Map.class, 1);
        return r;
    });
    Future<Map<String, Object>> f3 = service.submit(() -> {
        Map<String, Object> r =

 restTemplate.getForObject("http://localhost:8080/product/{1}",
Map.class, 2);
        return r;
    });
    Future<Map<String, Object>> f4 = service.submit(() -> {
        Map<String, Object> r =

 restTemplate.getForObject("http://localhost:8080/logistics/{1}",
Map.class, 1);
        return r;
    });
    System.out.println(f1.get());
    System.out.println(f2.get());
    System.out.println(f3.get());
    System.out.println(f4.get());
    log.debug("执行完毕");
    service.shutdown();
```

```
}
```

我们使用线程池同时调用了三类接口，在性能上更好一些，同时可以通过Future获得到接口的返回值。

5.其他

5.1 谈谈你对ThreadLocal的理解

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

作用

- 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
- ThreadLocal 同时实现了线程内的资源共享

原理

每个线程内有一个 ThreadLocalMap 类型的成员变量，用来存储资源对象

- 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
- 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
- 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

----------以下为增强回答----------

弱引用 key

ThreadLocalMap 中的 key 被设计为弱引用，原因如下

- Thread 可能需要长时间运行（如线程池中的线程），如果 key 不再使用，需要在内存不足（GC）时释放其占用的内存

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
         super(k);
         value = v; //强引用，不会被回收
     }
}
```

内存释放时机

- 被动 GC 释放 key
  - 仅是让 key 的内存释放，关联 value 的内存并不会释放
- 懒惰被动释放 value
  - get key 时，发现是 null key，则释放其 value 内存
  - set key 时，会使用启发式扫描，清除临近的 null key 的 value 内存，启发次数与元素个数，是否发现 null key 有关
- 主动 remove 释放 key，value
  - 会同时释放 key，value 的内存，也会清除临近的 null key 的 value 内存
  - 推荐使用它，因为一般使用 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

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~~
>
> 并行是指多个任务在计算机中同时执行，并发是指多个任务在计算机中交替执行。都是同时执行多个任务，一个是同时执行，一个是交替执行。

面试官：说一下线程和进程的区别？

候选人：

嗯，好~

进程是程序运行和资源分配的基本单位，就像我们经常使用的windows操作系统，打开一个软件就相当于执行了一个进程。一般一个进程至少会有一个线程执行，也可能是多个线程在进程中执行。

进程在运行过程中，需要拥有独立的内存单元，否则如果申请不到内存的话就会挂起。而多个线程能共享内存资源，这样就能降低运行的门槛，从而效率更高。

java中的线程是cpu调度和分派的基本单位，在实际开发过程中，一般是考虑多线程并发。

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：

会心一笑（不要表现出来）

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）

在JDK6之前synchronized锁都属于重量级锁，因为底层都是使用Monitor 来实现的。但是在JDK6之后对synchronized做了升级优化，里面主要体现在：CAS 自旋、锁消除、锁膨胀、轻量级锁、偏向锁等。

在控制锁升级的过程中，跟对象实例的内存布局是有关系的。

在 Java 中，任何的对象实例的内存布局都分为**对象头、对象实例数据**和**对齐填充数据**三个部分，其中对象头又包括 **MarkWord** 和 **类型指针**。

在**MarkWord** 中是由32个bit位或64个bit位组成，最后三位是来标志锁的升级情况的。

倒数第三位是偏向锁标志，默认是0，后两位默认是01，也就说最后三位是001，则是无锁状态，这种情况也是性能最优的情况。

当只有一个线程去争抢锁的时候,会先使用偏向锁,就是给一个标识,说明现在这个锁被某一个线程占有。在**MarkWord** 倒数第三位如果是1，后两位默认是01，也就说最后三位是101，则是说明是偏向锁。只需要判断一下是否有偏向锁指向它的 线程ID，无需再进入 Monitor 去竞争对象了

当线程越来越多想要抢占锁，于是将标识去掉,也就是撤销偏向锁,升级为轻量级锁。多个线程通过CAS进行锁的争抢。这个时候在在**MarkWord** 的后两位是00。这种情况可以应付少量线程的并发。

当线程更多的时候，一直在自旋,那这样等着也是干耗费CPU资源,所以就将锁升级为重量级锁,向内核申请资源,直接将等待的线程进行阻塞.这个时候在在**MarkWord** 的后两位是10。这种情况可以应付大量线程的并发。目前走的monitor的逻辑，相对来说，新能不太高。

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，意思是比较再交换，是现代CPU广泛支持的一种对内存中的共享数据进行操作的一种特殊指令。CAS可以将read-modify-write转换为原子操作，这个原子操作直接由CPU保证。在CAS中有3个操作数：**内存值V，旧的预期值A，要修改的新值B**。当且仅当旧预期值A和内存值V相同时，将内存值V修改为B并返回true，否则什么都不做，并返回false。

通常情况下自旋锁都是通过这种方式来完成的。

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，支持定时及周期性任务执行。

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[seməfɔ:r]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：能聊一下导致并发程序出现问题的根本原因是什么?

候选人：

嗯，是这样的，有这几种情况。

第一： CPU 缓存，在多核 CPU 的情况下，带来了可见性问题，一个线程对共享变量的修改，另一个线程能够立刻看到修改后的值

第二：操作系统对当前执行线程的切换，带来了原子性问题，一个或多个指令在 CPU 执行的过程中不被中断的特性

第三：编译器指令重排优化，带来了有序性问题

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。