线程基础

线程和进程区别？

- 程序运行开启一个进程，进程是正在运行程序的实例，包含多个线程；
- 不同进程使用不同的内存空间，当前进行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比进程上下文低（上下文切换指从一个线程切换到另一个线程）

并发和并行

并发是多个线程轮流使用CPU，单核下微观串行，宏观并行；

并行是多核同一时间做多件事，4核CPU同时执行4个线程

创建线程的四种方式

继承Thread类，实现runnable接口，实现Callable接口，线程池创建线程

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() {
            @Override
            public void run() {
            log.debug("hello");
            }
        };
        Runnable task2 = () -> log.debug("hello");
        // 创建Thread对象
        Thread t1 = new Thread(mr) ;
        Thread t2 = new Thread(task2,"t2") ;

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

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

    }

}
```

runnable 和 callable 有什么区别

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. **Callable接口的call()方法允许抛出异常；**而Runnable接口的run()方法的异常只能在内部消化，不能继续上抛
3. 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。

线程的 run()和 start()有什么区别？

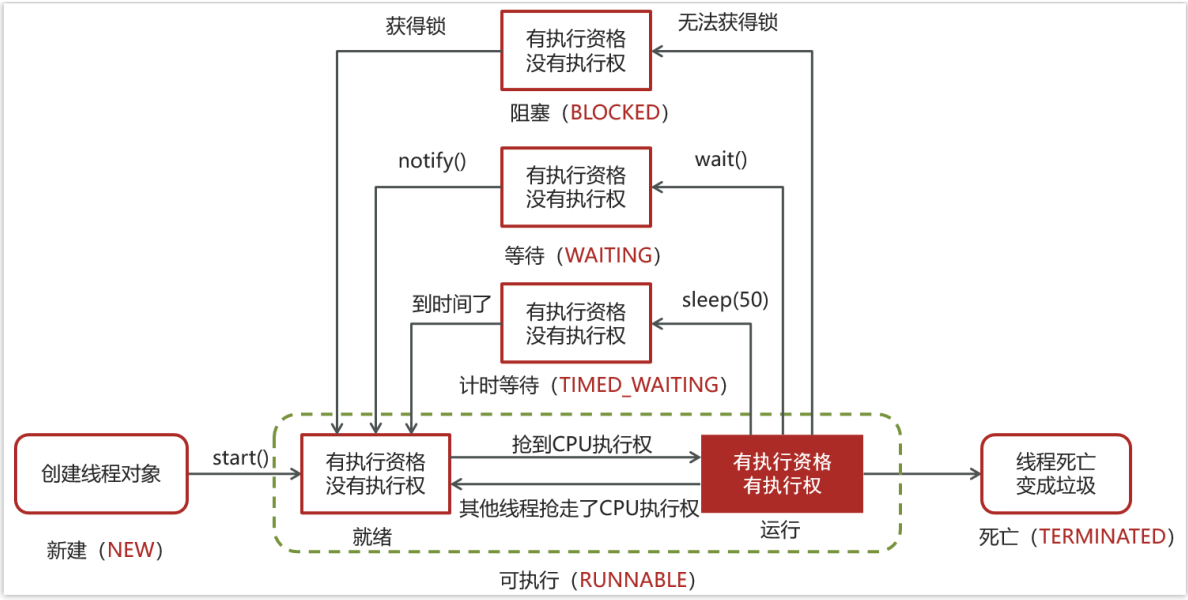
start(): 用来启动线程，通过该线程调用run方法执行run方法中所定义的逻辑代码。

start方法只能被调用一次。run封装了要执行的代码，可以调用多次。

线程包括哪些状态，状态之间是如何变化的

> 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



State是枚举类

- new 新建：线程对象被创建但没调用start()时
- RUNNABLE 可运行：调用了start()，等待系统资源如CPU时间片准备执行；
- TERMINATED 终结：线程内代码执行完毕
- BLOCKED 阻塞：处于阻塞状态的线程正在等待监视器锁进入同步块/方法
  - 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
- WAITING 等待：如Thread.join()正等待指定线程终止
  - 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
- TIMED_WAITING 有限时等待：
  - 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**
  - 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
  - sleep(long) 方法会从可运行状态进入有限时等待，到时间自动恢复可运行状态，不需要主动唤醒

新建 T1、T2、T3 三个线程，如何保证它们按顺序执行？

可以执行join()方法，在T2中调用T1，在T3中调用T2来保证T3最后执行；

notify()和 notifyAll()有什么区别？

notifyAll：唤醒所有wait的线程

notify：只随机唤醒一个 wait 线程

在 java 中 wait 和 sleep 方法的不同?

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；wait() 是Object的成员方法，每个对象都有
- 醒来时机不同：
  - 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
- 锁特性不同
  - wait 方法的调用必须先获取 wait 对象的锁，配合synchronized使用wait，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）

锁举例：

```
private static void waiting() throws InterruptedException {
    Thread t1 = new Thread(() -> {
      synchronized (LOCK) {
        try {
          get("t").debug("waiting...");
          LOCK.wait(5000L);  //wait需要在synchronized代码块中执行，执行后会释放LOCK对象锁
        } catch (InterruptedException e) {
          get("t").debug("interrupted...");
          e.printStackTrace();
        }
      }
    }, "t1");
    t1.start();

    Thread.sleep(100);
    synchronized (LOCK) {  //wait执行后就可以拿到锁饼运行代码块
      main.debug("other...");
    }

  }

  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) {  //不能立即拿到锁因为还没被释放，必须等t1里的sleep自动唤醒后释放才能拿到锁
            main.debug("other...");
        }
    }
```

如何停止一个正在运行的线程？

- 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

```
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);
            } 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();

    }
}
```

> 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flag，flag为true则退出

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

线程中的并发锁

synchronized关键字的底层原理？

Synchronized【对象锁】采用互斥的方式让同一时刻至多只有一个线程能持有【对象锁】，其它线程再想获取这个【对象锁】时就会阻塞住，是悲观锁；底层是JVM中的monitor实现的，在字节码中会对代码块用monitorenter上锁，monitorexit解锁，解两次是防止代码抛异常不能及时释放锁；

具体说下monitor

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

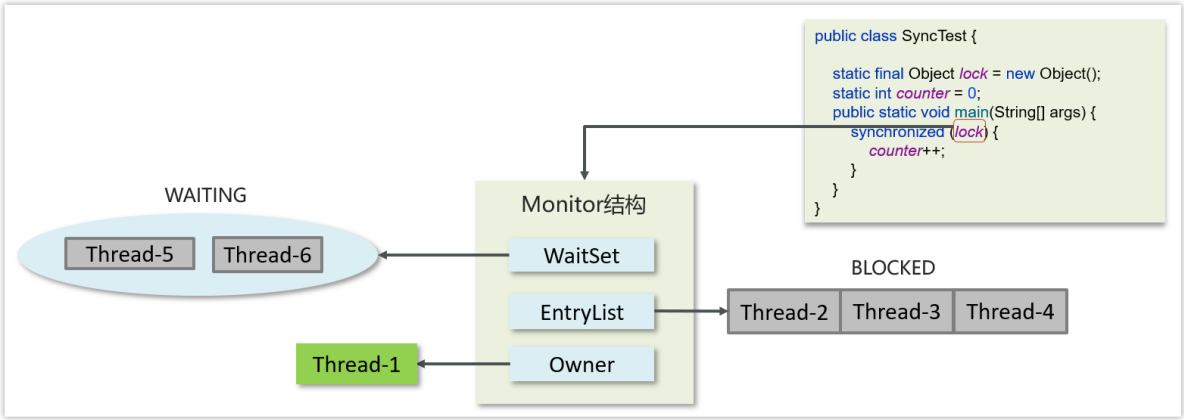
monitor内部维护了三个变量

- 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。

monitor结构是：



- 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

流程：

- 代码进入synchorized代码块，先让对象锁关联monitor，判断monitor中的owner是否由线程持有
- 没有则让当前线程持有
- 有则进入entryList阻塞等待，当owner释放锁再让entryList中的线程去竞争owner的持有权
- 如果调用了wait()，会去waiSet中等待

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

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？

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

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

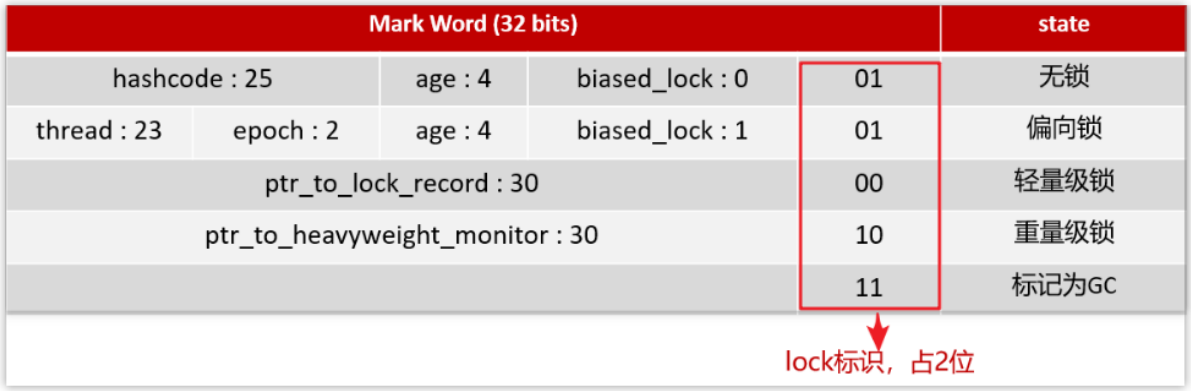
一旦锁发生了竞争，都会升级为重量级锁

还有偏向锁和轻量级锁，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。

对象的内存结构

对象在内存中主要存对象头、实例数据（成员变量）和对齐填充（补齐前两个实现8的整数倍）

其中对象头有MarkWord和KlassWord（描述对象实例的具体类型）



- **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
- **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

可以通过后三位判断锁的等级：

- 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

Monitor重量级锁

使用 synchronized 给对象上锁（重量级）之后，**该对象头的Mark Word 中就被设置指向 Monitor 对象的指针**

轻量级锁

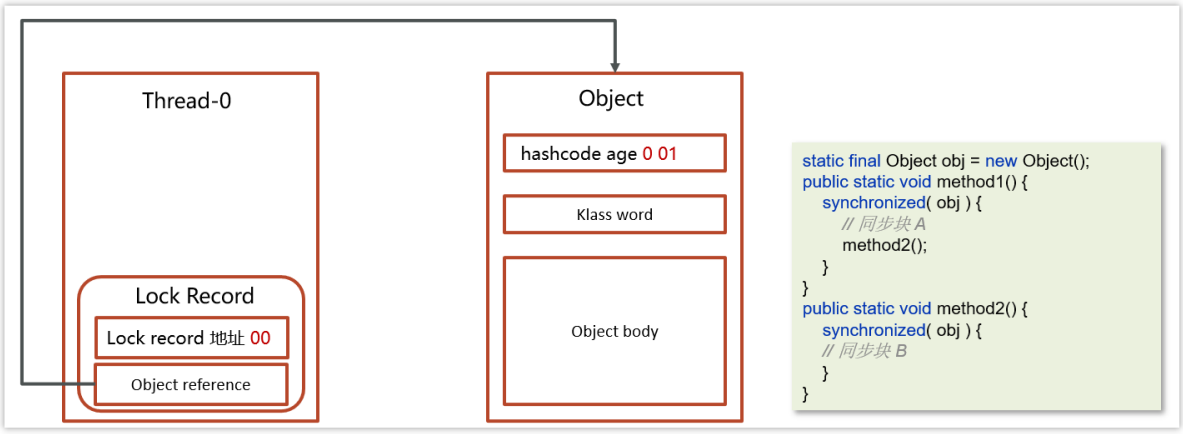
在同步块中的代码是不存在竞争的，就没必要使用重量级锁过大的开销。轻量级锁对使用者是透明的，即语法仍然是synchronized。

加锁时会在栈帧中创建LockRecord，将其中的obj字段指向锁对象，通过CAS指令将Lockrecord地址存在对象头中，第一部分存markword的值；

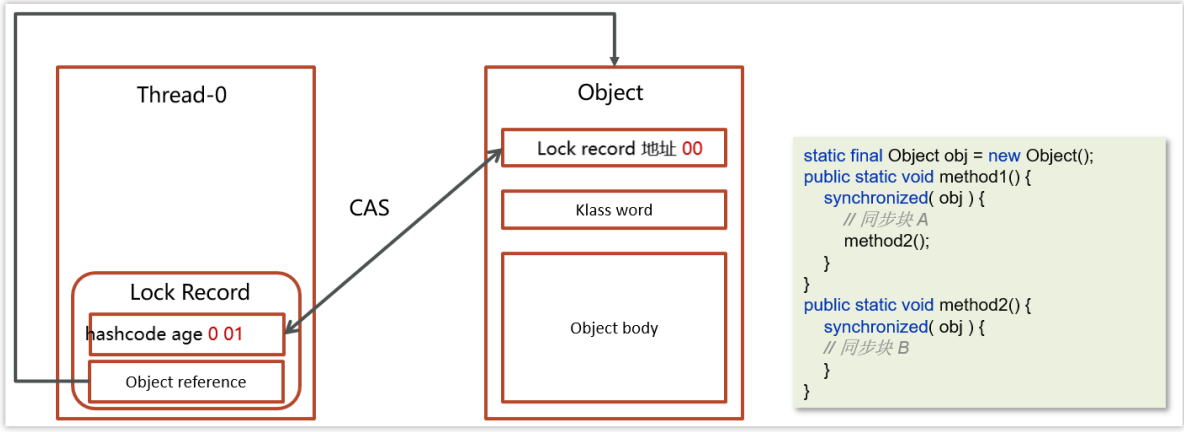
- CAS数据交换成功加锁成功，说明是无锁状态；
- CAS数据交换失败：说明有锁，是其他线程的锁则升级为重量锁；如果是当前线程持有的锁表示锁重入，则创建的LockRecord第一部分为null，重入计数+1，当释放锁后，如果有null值的锁记录则重置所忌入

加锁的流程

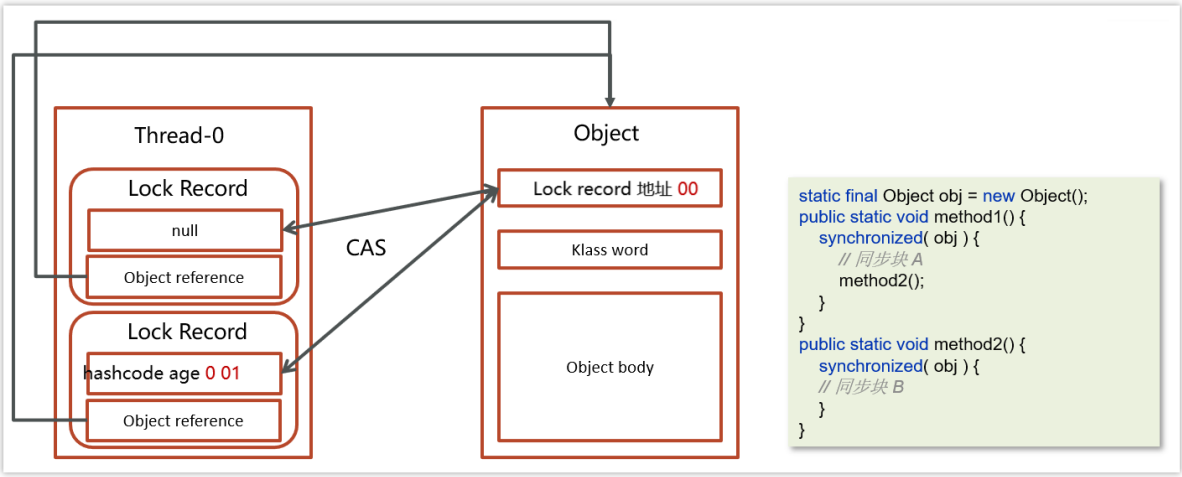
1.在线程栈中创建一个Lock Record，将其obj字段指向锁对象。



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。

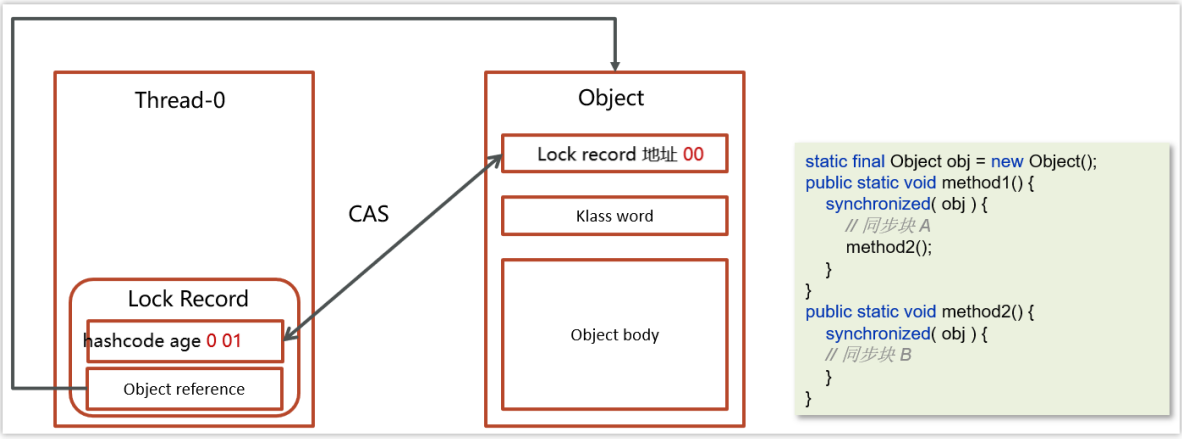


4.如果CAS修改失败，说明发生了竞争，需要膨胀为重量级锁。

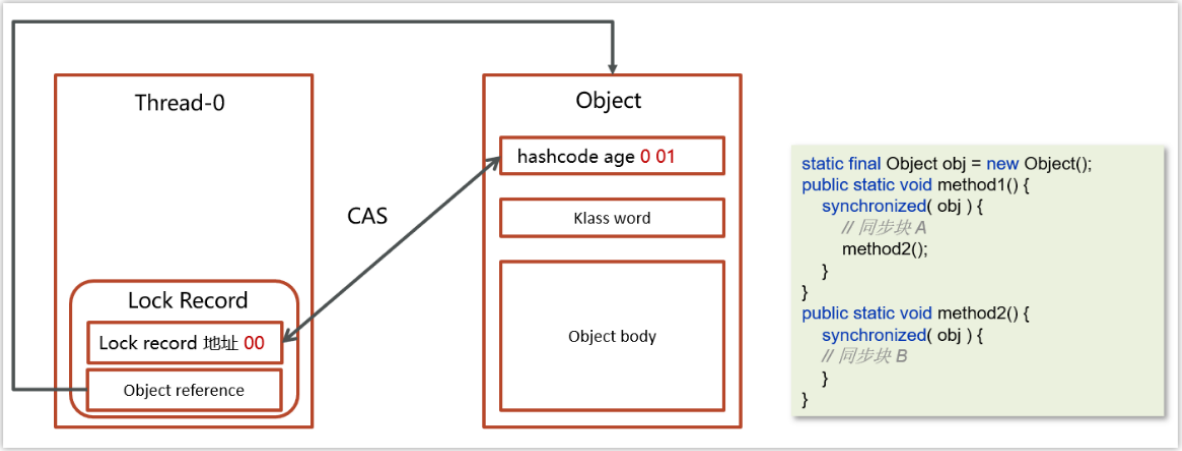
解锁过程

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。

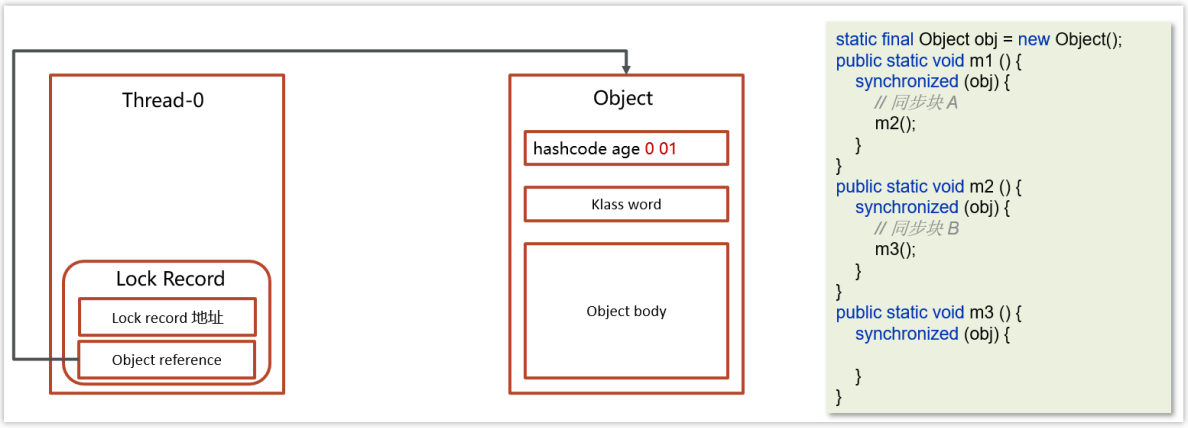


偏向锁

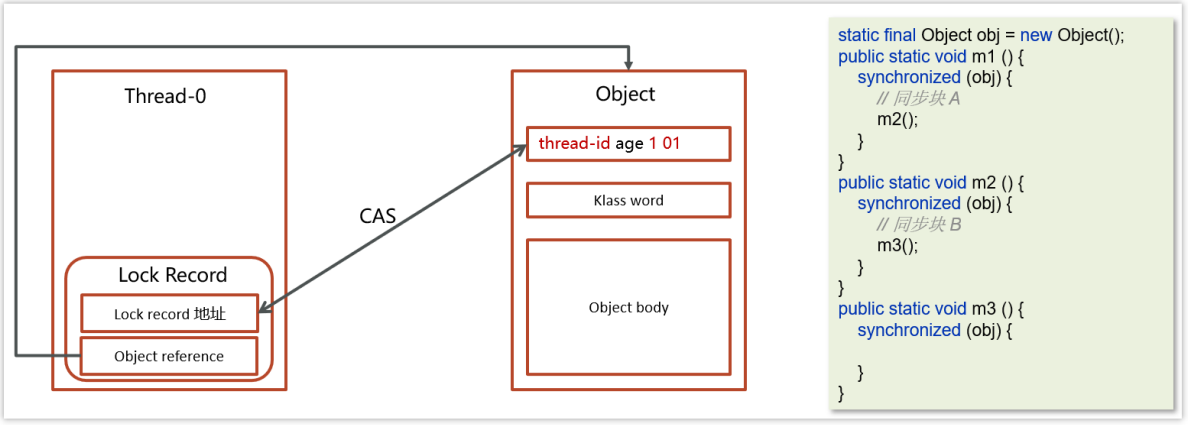
轻量级锁在没有竞争时（就自己这个线程），每次重入仍然需要执行 CAS 操作。可以将线程ID设置到markWord头中，如果线程id是自己表示不用CAS

加锁的流程

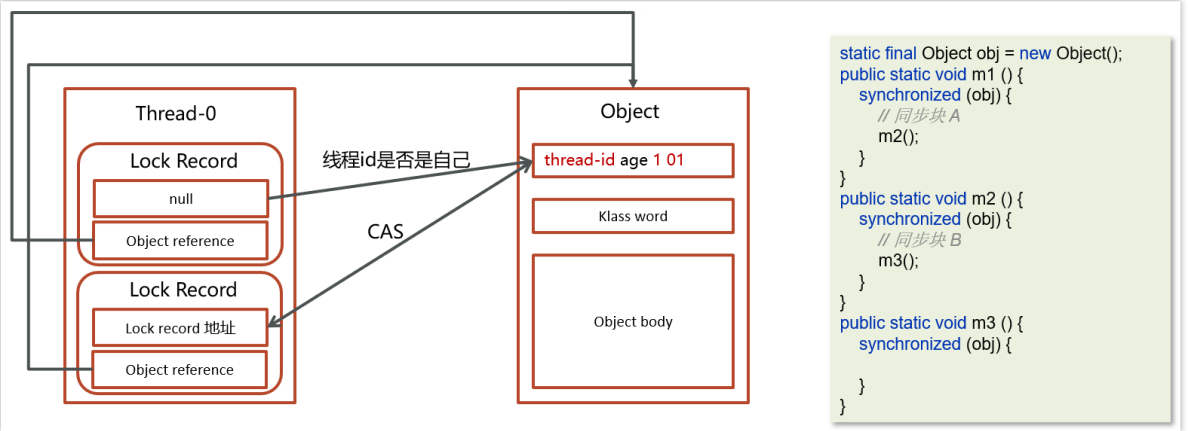
1.在线程栈中创建一个Lock Record，将其obj字段指向锁对象。



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

谈谈 JMM（Java 内存模型）

线程不能直接读写主内存中的变量，在自己的工作内存中保留共享变量副本进行读写操作，线程间的变量传递需要通过主内存完成。

CAS 你知道吗？

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**。

> 旧值A要更新为B，需要和内存值V作比较，A和V即旧值和内存值相同的时候才能修改内存值为B；如果CAS失败，则获取共享内存中V值的最新值，通过自旋方式等待尝试修改，直到成功。

CAS乐观锁的思想：不怕别的线程来修改共享变量，大不了自旋重试

synchronized悲观锁思想：防止其他线程来修改，我上锁后等修改完再解锁，其他线程阻塞等待

对 volatile 的理解

> 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

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，volatile关键字会强制将修改的值立即写入主存。

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。

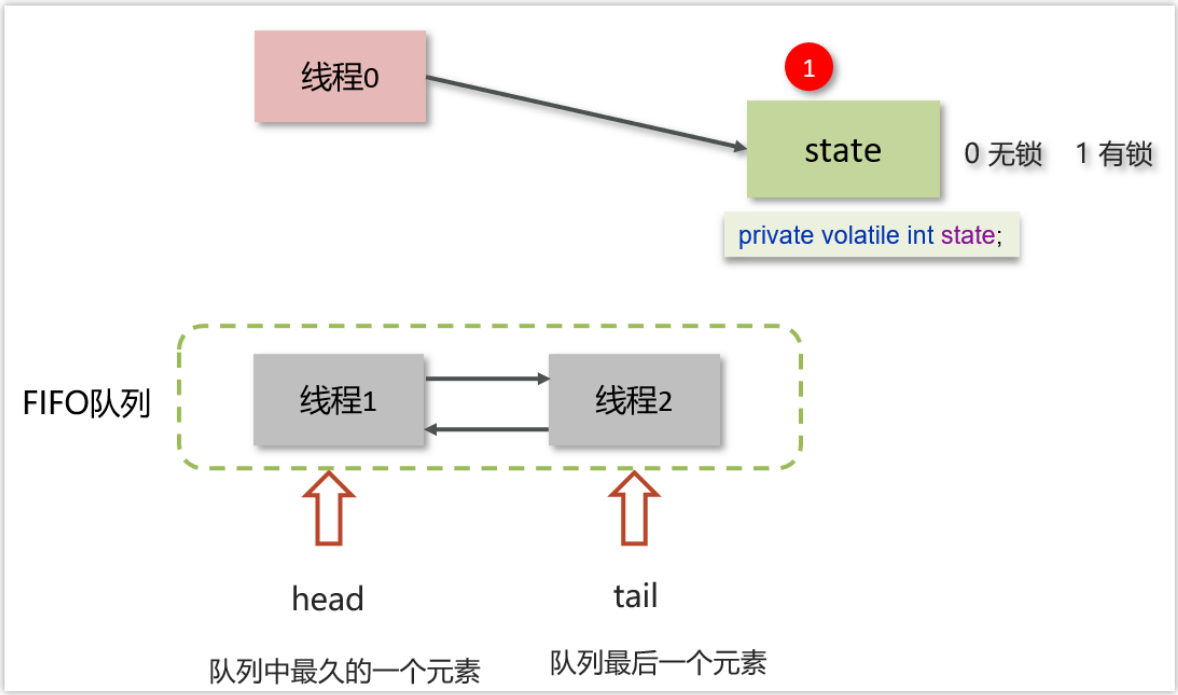
- 写操作加的屏障是阻止上方其它写操作越过屏障排到volatile变量写的下方，一般修饰的变量放在最下方防止其他写操作越位
- 读操作加的屏障是阻止下方其它读操作越过屏障排到volatile变量读的上方，一般修饰的变量放在最上方防止其他读操作越位

什么是AQS？

> 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

阻塞式锁和相关的同步器工具的框架，常见的实现类有reentrantLock可重入锁；悲观锁，手动开启释放。

- 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

多个线程共同去抢这个state资源是如何保证原子性的呢？

用CAS自旋锁保证原子性，失败线程的去等待队列

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

ReentrantLock的实现原理

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。

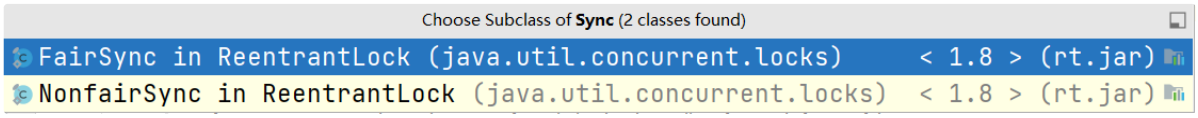
ReentrantLock是属于juc报下的类，属于api层面的锁，跟synchronized一样，都是悲观锁。通过lock()用来获取锁，unlock()释放锁。

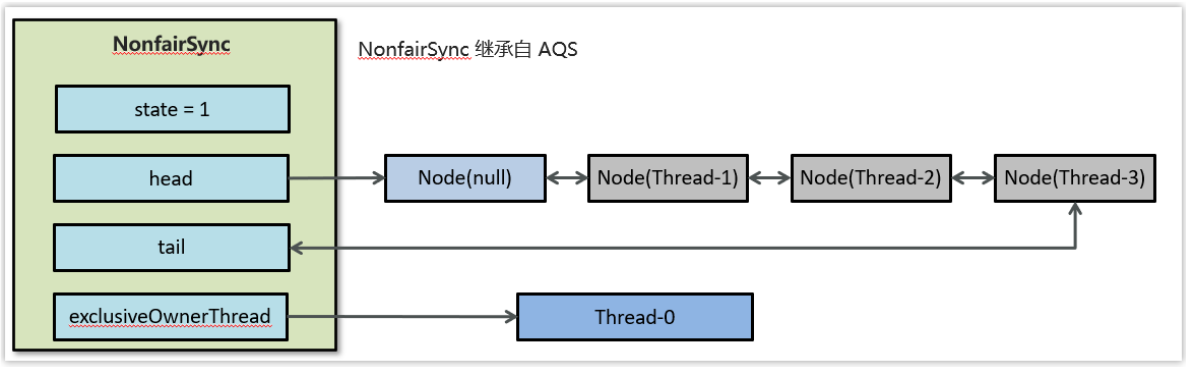
它的底层实现原理主要利用**CAS+AQS队列**来实现。它支持公平锁和非公平锁，两者的实现类似

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。

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

主要利用CAS+AQS队列来实现。它支持公平锁和非公平锁，带参数公平锁，不带参数非公平锁





- 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

synchronized和Lock有什么区别？

- 语法层面
  - synchronized 是关键字，源码在jvm中，C++实现；Lock由jdk提供Java实现
  - synchronized自动释放锁，Lock需要手动unLock
- Lock比synchronized更灵活，提供了更多的场景，实现ReentrantLock之类的
- 性能上竞争不激烈下synchronized由偏向锁轻量锁性能不差；竞争激烈Lock更好

死锁产生的条件是什么？

一个线程需要同时获取多把锁，这时就容易发生死锁，但是t1线程持有A锁等待获取B锁，t2线程获取B锁等待A锁；

如何进行死锁诊断？

jdk自带的工具：jps和 jstack或者一些可视化工具监控线程

> 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。

ConcurrentHashMap

HashMap1.8的结构一样，数组+链表/红黑二叉树。在计算完key的hash值确认存储的下标后，会先判断当前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

> 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

*导致并发程序出现问题的根本原因是什么

Java并发编程三大特性：

- 原子性：一个线程在CPU中的操作要么都执行完成，要么不执行：synchronized：同步加锁/lock：加锁
- 可见性：让一个线程对共享变量的修改对另一个线程可见：volatile（推荐）
- 有序性：指令重排，处理器为了提高程序运行效率，不保证程序中的语句执行顺序一致，但是保证程序运行结果一致。

线程池

说一下线程池的核心参数（线程池的执行原理知道嘛）

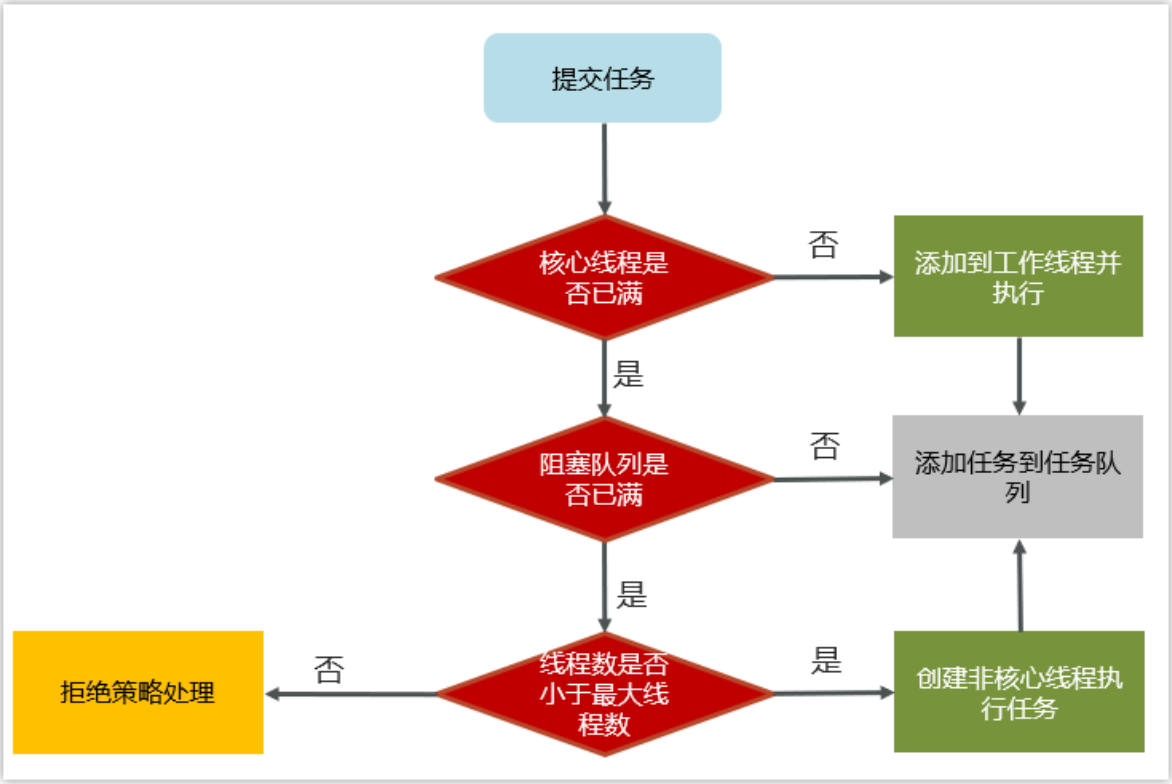
> 首先判断线程池里的核心线程是否都在执行任务，如果不是则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队 列里。如果工作队列满了，则判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任 务。如果已经满了，则交给拒绝策略来处理这个任务。

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

```
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；

线程池中有哪些常见的阻塞队列

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

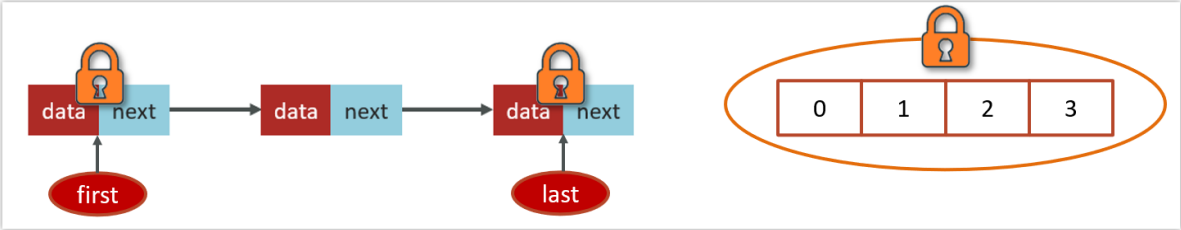
用的最多是ArrayBlockingQueue和LinkedBlockingQueue

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



如何确定核心线程数

> 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。

高并发、任务执行时间短 -->（ CPU核数+1 ），减少线程上下文的切换

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 ）

线程池的种类有哪些

> 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。

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

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

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

- 核心线程数和最大线程数都是1
- 阻塞队列是LinkedBlockingQueue，最大容量为Integer.MAX_VALUE
- 适用场景：适用于按照顺序执行的任务

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

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

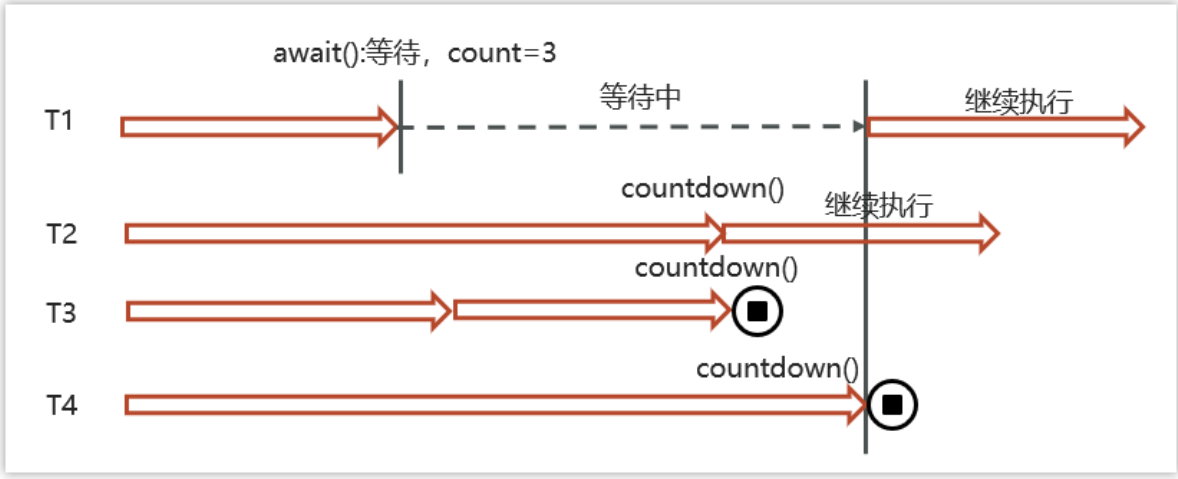
```
public ScheduledThreadPoolExecutor(int corePoolSize) {
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,new DelayedWorkQueue());
}
public ScheduledThreadPoolExecutor(int corePoolSize,
                                   ThreadFactory threadFactory) {
    super(corePoolSize, Integer.MAX_VALUE,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

- 适用场景：有定时和延迟执行的任务

线程使用场景问题

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

CountDownLatch



用于进行线程同步协作的，等待所有线程完成倒计时；

```
CountDownLatch latch = new CountDownLatch(3); //表示有3个任务线程
for(int i=0; i<3;i++){
    new Thread(()->{
        //代码
        latch.countDown();
    }).start();
}
//等待其他线程完成
latch.await();
```

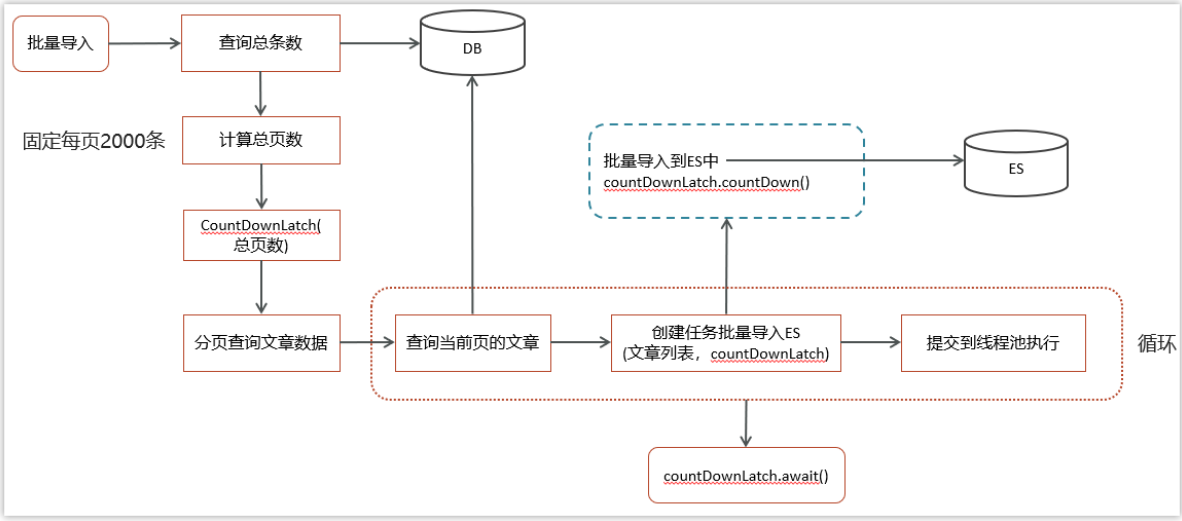
- 用countDownLatch.await() 等待计数归零表示其余线程都执行完了；
- countDown() 用来让计数减一

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：



这个例子就是为了保证在项目启动前es能完成导入，防止搜索服务启动后es还没完成索引导入造成用户体验差；

计算总页数后可以开多个线程批量导入每页的数据。每个线程里将每页的文章数据放入es的bulkRequest中上传，提交到线程池中执行，线程完成用countDown(); 等全部数据页导入索引成功会await()才会放行；

如何控制某个方法允许并发访问线程的数量？

Semaphore [ˈsɛməˌfɔr] 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

> 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

谈谈你对ThreadLocal的理解

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

比如用户登入后，会将用户信息保存到各自的ThreadLocal中，保证每个线程都在各自的用户上进行操作，保存对应用户的行为信息。不同用户之间独立。

ThreadLocal基本使用

三个主要方法：

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

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

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

ThreadLocal-内存泄露问题

> 是因为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。

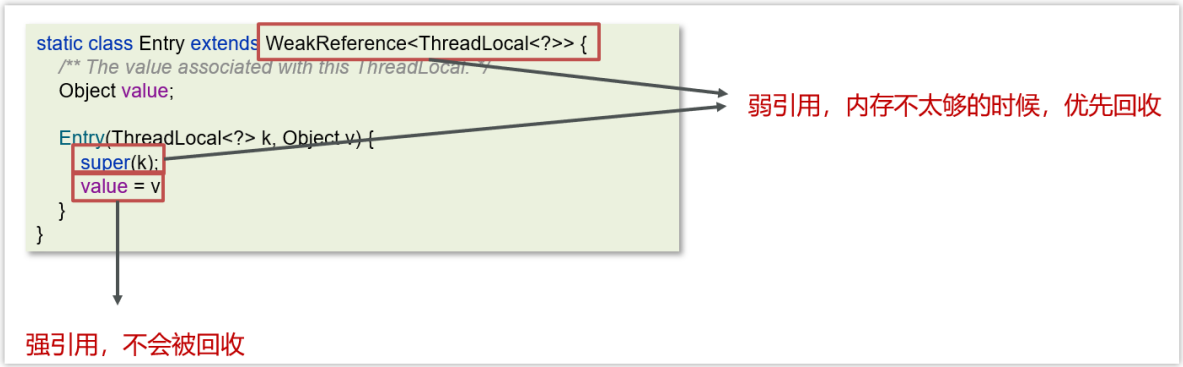
- 强引用：最为普通的引用方式，表示一个对象处于有用且必须的状态，如果一个对象具有强引用，则GC并不会回收它。即便堆中内存不足了，宁可出现OOM，也不会对其进行回收

```
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



在使用ThreadLocal的时候，强烈建议：**务必手动remove**