并发工具类及线程池

一、CyclicBarrier简介

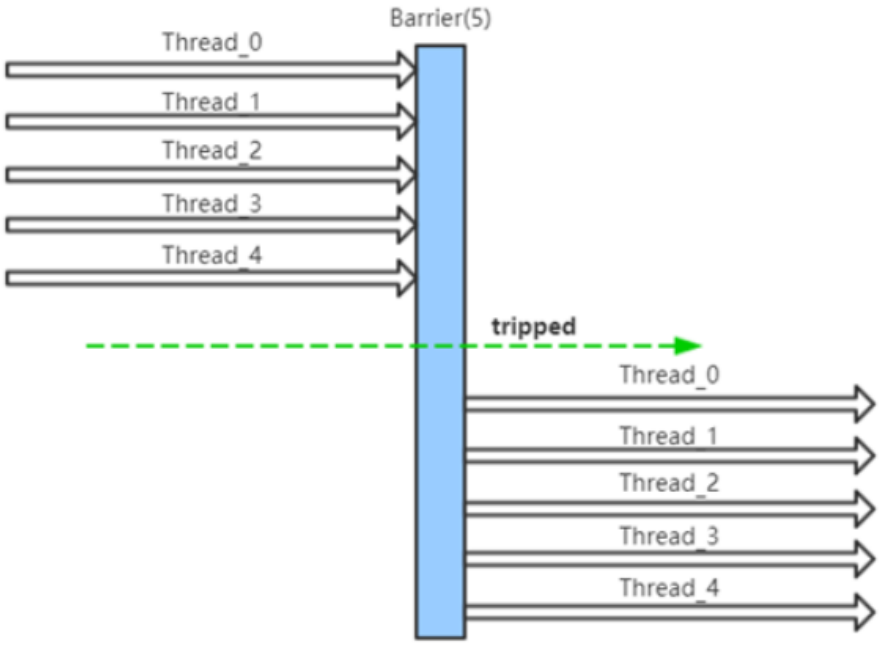
1、简介

CyclicBarrier是一个同步的辅助类，允许一组线程相互之间等待，达到一个共同点，再继续执行。
CyclicBarrier（循环屏障） 直译为可循环使用（Cyclic）的屏障（Barrier）。它可以让一组线程到达一个屏障（同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续工作。
JDK中的描述：

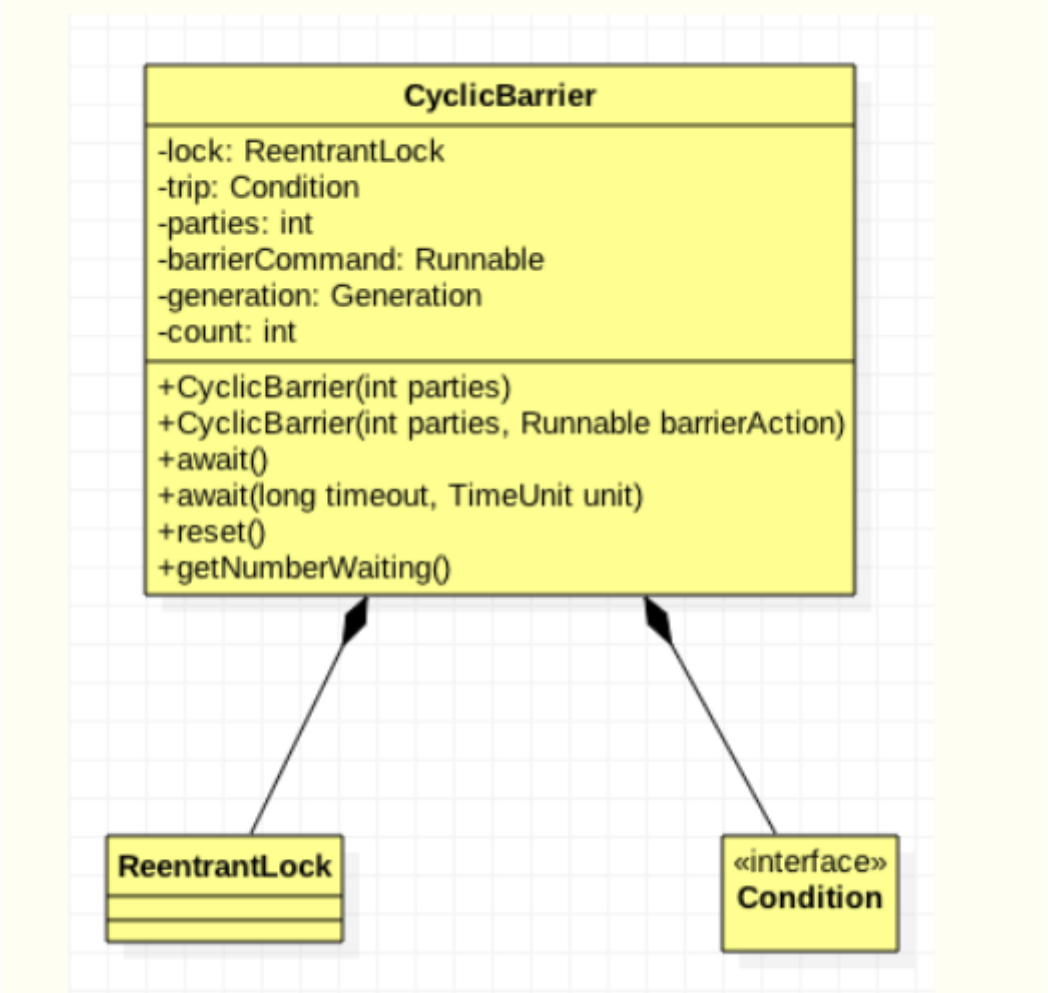
```
A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.
CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other.
The barrier is called cyclic because it can be re-used after the waiting threads are released.
```

CyclicBarrier是一个同步辅助类，它允许一组线程相互等待直到所有线程都到达一个公共的屏障点。
在程序中有固定数量的线程，这些线程有时候必须等待彼此，这种情况下，使用CyclicBarrier很有帮助。
这个屏障之所以用循环修饰，是因为在所有的线程释放彼此之后，这个屏障是可以重新使用的。

2、运行机制



二、CyclicBarrier结构图



三、CyclicBarrier方法说明

1、CyclicBarrier(parties)

初始化相互等待的线程数量的构造方法

2、CyclicBarrier(parties,Runnable barrierAction)

初始化相互等待的线程数量的构造方法以及屏障线程的构造方法
屏障线程的运行时机：等待的线程数量 = parties，CyclicBarrier打开屏障之前
举例：在分组计算中，每个线程负责一部分计算，最终这些线程计算结束之后，交由屏障线程进行汇总计算

3、getParties（）

获取CyclicBarrier打开屏障的线程数量，也成为方数

4、getNumberWaiting()

获取真在CyclicBarrier上等待的线程数量

5、await()

在CyclicBarrier上进行阻塞等待，直到发生以下情形之一：
a.在CyclicBarrier上等待的线程数量达到parties，则所有线程被释放，继续执行。
b.当前线程被中断，则抛出InterruptedException异常，并停止等待，继续执行。
c.其他等待的线程被中断，则当前线程抛出BrokenBarrierException异常，并停止等待，继续执行。
d.其他等待的线程超时，则当前线程抛出BrokenBarrierException异常，并停止等待，继续执行。
e.其他线程调用CyclicBarrier.reset()方法，则当前线程抛出BrokenBarrierException异常，并停止等待，继续执行。

6、await(timeout,TimeUnit)

在CyclicBarrier上进行限时的阻塞等待，直到发生以下情形之一：
a.在CyclicBarrier上等待的线程数量达到parties，则所有线程被释放，继续执行。
b.当前线程被中断，则抛出InterruptedException异常，并停止等待，继续执行。
c.当前线程等待超时，则抛出TimeoutException异常，并停止等待，继续执行。
d.其他等待的线程被中断，则当前线程抛出BrokenBarrierException异常，并停止等待，继续执行。
e.其他等待的线程超时，则当前线程抛出BrokenBarrierException异常，并停止等待，继续执行。
f.其他线程调用CyclicBarrier.reset()方法，则当前线程抛出BrokenBarrierException异常，并停止等待，继续执行。

7、isBroken

获取是否破损标志位broken的值，此值有以下几种情况：
a.CyclicBarrier初始化时，broken=false，表示屏障未破损。
b.如果正在等待的线程被中断，则broken=true，表示屏障破损。
c.如果正在等待的线程超时，则broken=true，表示屏障破损。
d.如果有线程调用CyclicBarrier.reset()方法，则broken=false，表示屏障回到未破损状态。

8、reset

使得CyclicBarrier回归初始状态，直观来看它做了两件事：
a.如果有正在等待的线程，则会抛出BrokenBarrierException异常，且这些线程停止等待，继续执行。
b.将是否破损标志位broken置为false。

四、源码分析

首先看一下CyclicBarrier内部声明的一些属性

```
/**用于保护屏障入口的锁*/
private final ReentrantLock lock = new ReentrantLock();
/**线程等待条件 */
private final Condition trip = lock.newCondition();
/** 记录等待的线程数 */
private final int parties;
/**所有线程到达屏障点后，首先执行的命令 */
private final Runnable barrierCommand;
private Generation generation = new Generation();
/**实际中仍在等待的线程数，每当有一个线程到达屏障点，count值就会减一；当一次新的运算开始后，
count的值被重置为parties*/
private int count;
```

其中，Generation是CyclicBarrier的一个静态内部类，它只有一个boolean类型的属性，具体代码如下：

```
private static class Generation {
    Generation() {}                 // prevent access constructor creation
    boolean broken;                 // initially false
}
```

当使用构造方法创建CyclicBarrier实例的时候，就是给上面这些属性赋值

```
//创建一个CyclicBarrier实例，parties指定参与相互等待的线程数，
//barrierAction指定当所有线程到达屏障点之后，首先执行的操作，该操作由最后一个进入屏障点的线程执行。
public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    this.parties = parties;
    this.count = parties;
    this.barrierCommand = barrierAction;
}
//创建一个CyclicBarrier实例，parties指定参与相互等待的线程数
public CyclicBarrier(int parties) {
    this(parties, null);
}
```

CyclicBarrier.await方法调用CyclicBarrier.dowait()，每次调用await()都会使计数器-1，当减少到0时就会唤醒所有的线程，当调用await()方法时，当前线程已经到达屏障点，当前线程阻塞进入休眠状态。

```
 //该方法被调用时表示当前线程已经到达屏障点，当前线程阻塞进入休眠状态
 //直到所有线程都到达屏障点，当前线程才会被唤醒
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
```

当前线程已经到达屏障点，当前线程阻塞进入休眠状态

```
//该方法被调用时表示当前线程已经到达屏障点，当前线程阻塞进入休眠状态
//在timeout指定的超时时间内，等待其他参与线程到达屏障点
//如果超出指定的等待时间，则抛出TimeoutException异常，如果该时间小于等于零，则此方法根本不会等待
public int await(long timeout, TimeUnit unit)
    throws InterruptedException,
           BrokenBarrierException,
           TimeoutException {
    return dowait(true, unit.toNanos(timeout));
}
```

dowait（）方法

```
private int dowait(boolean timed, long nanos)
```

```
    throws InterruptedException, BrokenBarrierException,
           TimeoutException {
    //使用独占资源锁控制多线程并发进入这段代码
    final ReentrantLock lock = this.lock;
    //独占锁控制线程并发访问
    lock.lock();
    try {
        final Generation g = generation;

        if (g.broken)
            throw new BrokenBarrierException();
        //检查当前线程是否被中断
        if (Thread.interrupted()) {
            //如果当前线程被中断会做以下三件事
            //1.打翻当前栅栏
            //2.唤醒拦截的所有线程
            //3.抛出中断异常
            breakBarrier();
            throw new InterruptedException();
        }
         //每调用一次await()方法，计数器就减一
        int index = --count;
        //计数器的值减为0则需唤醒所有线程并转换到下一代
        if (index == 0) {  // tripped
            boolean ranAction = false;
            try {
                //如果在创建CyclicBarrier实例时设置了barrierAction，则先执行
barrierAction
                final Runnable command = barrierCommand;
                if (command != null)
                    command.run();
                ranAction = true;
                //当所有参与的线程都到达屏障点，为唤醒所有处于休眠状态的线程做准备工作
                //需要注意的是，唤醒所有阻塞线程不是在这里
                nextGeneration();
                return 0;
            } finally {
                //确保在任务未成功执行时能将所有线程唤醒
                if (!ranAction)
                    breakBarrier();
            }
        }

        // loop until tripped, broken, interrupted, or timed out
        //如果计数器不为0则执行此循环
        for (;;) {
            try {
                //根据传入的参数来决定是定时等待还是非定时等待
                if (!timed)
                    //让当前执行的线程阻塞，处于休眠状态
                    trip.await();
                else if (nanos > 0L)
                    //让当前执行的线程阻塞，在超时时间内处于休眠状态
                    nanos = trip.awaitNanos(nanos);
            } catch (InterruptedException ie) {
                //若当前线程在等待期间被中断则打翻栅栏唤醒其他线程
                if (g == generation && ! g.broken) {
                    breakBarrier();
```

```
                    throw ie;
                } else {
                    // We're about to finish waiting even if we had not
                    // been interrupted, so this interrupt is deemed to
                    // "belong" to subsequent execution.
                    Thread.currentThread().interrupt();
                }
            }
            //如果线程因为打翻栅栏操作而被唤醒则抛出异常
            if (g.broken)
                throw new BrokenBarrierException();

            if (g != generation)
                return index;
             //如果线程因为时间到了而被唤醒则打翻栅栏并抛出异常
            if (timed && nanos <= 0L) {
                breakBarrier();
                throw new TimeoutException();
            }
        }
    } finally {
        lock.unlock();
    }
}
```

每次调用await方法都会使内部的计数器临时变量-1，当减少到0时，就会调用nextGeneration方法

```
private void nextGeneration() {
    // signal completion of last generation
    trip.signalAll();
    // set up next generation
    count = parties;
    generation = new Generation();
}
```

在这里唤醒所有阻塞的线程

提醒：在声明CyclicBarrier的时候还可以传一个Runnable的实现类，当计数器减少到0时，会执行该实现类

到这里CyclicBarrier的实现原理基本已经都清楚了，下面来深入源码分析一下线程阻塞代码trip.await()和线程唤醒trip.signalAll()的实现。

```
        //await()是AQS内部类ConditionObject中的方法
        public final void await() throws InterruptedException {
            //如果线程中断抛异常
            if (Thread.interrupted())
                throw new InterruptedException();
            //新建Node节点，并将新节点加入到Condition等待队列中
            //Condition等待队列是AQS内部类ConditionObject实现的，ConditionObject有两个属性，分别
                        是firstWaiter和lastWaiter，都是Node类型
            //firstWaiter和lastWaiter分别用于代表Condition等待队列的头结点和尾节点
            Node node = addConditionWaiter();
            //释放独占锁，让其它线程可以获取到dowait()方法中的独占锁
            int savedState = fullyRelease(node);
```

```
            int interruptMode = 0;
            //检测此节点是否在资源等待队列(AQS同步队列)中，
            //如果不在，说明此线程还没有竞争资源锁的权利，此线程继续阻塞，直到检测到此节点在资源等待队列上(AQS同步队列)中
            //这里出现了两个等待队列，分别是Condition等待队列和AQS资源锁等待队列(或者说是同步队列)
            //Condition等待队列是等待被唤醒的线程队列，AQS资源锁等待队列是等待获取资源锁的队列
            while (!isOnSyncQueue(node)) {
                //阻塞当前线程，当前线程进入休眠状态，可以看到这里使用LockSupport.park阻塞当前线程
                LockSupport.park(this);
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null) // clean up if cancelled
                unlinkCancelledWaiters();
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode);
        }

        //addConditionWaiter()是AQS内部类ConditionObject中的方法
        private Node addConditionWaiter() {
            Node t = lastWaiter;
            // 将condition等待队列中，节点状态不是CONDITION的节点，从condition等待队列中移除
            if (t != null && t.waitStatus != Node.CONDITION) {
                unlinkCancelledWaiters();
                t = lastWaiter;
            }
            //以下操作是用此线程构造一个节点，并将之加入到condition等待队列尾部
            Node node = new Node(Thread.currentThread(), Node.CONDITION);
            if (t == null)
                firstWaiter = node;
            else
                t.nextWaiter = node;
            lastWaiter = node;
            return node;
        }

        //signalAll是AQS内部类ConditionObject中的方法
        public final void signalAll() {
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            //Condition等待队列的头结点
            Node first = firstWaiter;
            if (first != null)
                doSignalAll(first);
        }

        private void doSignalAll(Node first) {
            lastWaiter = firstWaiter = null;
            do {
                Node next = first.nextWaiter;
                first.nextWaiter = null;
                //将Condition等待队列中的Node节点按之前顺序都转移到了AQS同步队列中
```

```
                transferForSignal(first);
                first = next;
            } while (first != null);
        }

        final boolean transferForSignal(Node node) {
            if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
                return false;
            //这里将Condition等待队列中的Node节点插入到AQS同步队列的尾部
            Node p = enq(node);
            int ws = p.waitStatus;
            if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
                LockSupport.unpark(node.thread);
            return true;
        }

        //ReentrantLock#unlock()方法
        public void unlock() {
            //Sync是ReentrantLock的内部类，继承自AbstractQueuedSynchronizer，它是
ReentrantLock中公平锁和非公平锁的基础实现
            sync.release(1);
        }

        public final boolean release(int arg) {
            //释放锁
            if (tryRelease(arg)) {
                //AQS同步队列头结点
                Node h = head;
                if (h != null && h.waitStatus != 0)
                    //唤醒节点中的线程
                    unparkSuccessor(h);
                return true;
            }
            return false;
        }

        private void unparkSuccessor(Node node) {
            int ws = node.waitStatus;
            if (ws < 0)
                compareAndSetWaitStatus(node, ws, 0);
            Node s = node.next;
            if (s == null || s.waitStatus > 0) {
                s = null;
                for (Node t = tail; t != null && t != node; t = t.prev)
                    if (t.waitStatus <= 0)
                        s = t;
            }
            if (s != null)
                //唤醒阻塞线程
                LockSupport.unpark(s.thread);
    }
```

案例：

```
/**
 * 旅行线程
 */
```

```
public class TravelTask implements Runnable{

    private CyclicBarrier cyclicBarrier;
    private String name;
    private int arriveTime;//赶到的时间

    public TravelTask(CyclicBarrier cyclicBarrier,String name,int arriveTime){
        this.cyclicBarrier = cyclicBarrier;
        this.name = name;
        this.arriveTime = arriveTime;
    }

    @Override
    public void run() {
        try {
            //模拟达到需要花的时间
            Thread.sleep(arriveTime * 1000);
            System.out.println(name +"到达集合点");
            cyclicBarrier.await();
            System.out.println(name +"开始旅行啦～～");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

```
/**
 * 导游线程，都到达目的地时，发放护照和签证
 *
 */
public class TourGuideTask implements Runnable{

    @Override
    public void run() {
        System.out.println("****导游分发护照签证****");
        try {
            //模拟发护照签证需要2秒
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class Client {

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

        CyclicBarrier cyclicBarrier = new CyclicBarrier(3,new TourGuideTask());
        Executor executor = Executors.newFixedThreadPool(3);
        //登哥最大牌，到的最晚
        executor.execute(new TravelTask(cyclicBarrier,"哈登",5));
        executor.execute(new TravelTask(cyclicBarrier,"保罗",3));
        executor.execute(new TravelTask(cyclicBarrier,"戈登",1));
    }
}
```

五、CountDownLatch

CountDownLatch是一个同步工具类，它允许一个或多个线程一直等待，直到其他线程的操作执行完后再执行。在Java并发中，countdownlatch的概念是一个常见的面试题，所以一定要确保你很好的理解了它

案例：一个例子：有三个线程解析表格中的数据，主线程需要等到表格解析完成后才执行后面的操作

```
package com.yuandengta.chap23;

import java.util.concurrent.CountDownLatch;

/**
 * @Author:Hardy
 * @QQ:2937270766
 * @官网：http://www.yuandengta.com
 */
class TaskThread extends Thread{
    private CountDownLatch latch ;

    public TaskThread(CountDownLatch latch){
        this.latch = latch;
    }
    public void run(){

        try {
            System.out.println("子线程"+Thread.currentThread().getName()+"正在执
行");
            Thread.sleep(3000);
            System.out.println("子线程"+Thread.currentThread().getName()+"执行完
毕");
            latch.countDown();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class Test {
    public static void main(String[] args) {
        CountDownLatch latch = new CountDownLatch(2);
        for(int i = 0;i < 2;i++){
```

```
            new TaskThread(latch).start();
        }
        System.out.println("等待两个子线程执行完毕");
        try {
            latch.await();
            System.out.println("两个子线程执行完毕");
            System.out.println("继续执行主线程");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

两者区别：

- CountDownLatch：一个或者多个线程，等待其它多个线程完成某件事情之后才能执行
- CyclicBarrier：多个线程互相等待，直到到达同一个同步点，再继续一次执行
- 对于CountDownLatch来说，重点是"一个线程（多个线程）等待"，而其它N个线程完成某件事情之后，可以终止，也可以等待。
- 而对于CyclicBarrier，重点是多个线程，在任意一个线程没有完成，所有线程都必须互相等待，然后继续执行。
- CountDownLatch是计数器，线程完成一个记录一个，只不过计数不是递增而是递减
- Cyclic更像是一个阀门，需要所有线程都到达，阀门才能打开，然后继续执行。

六、Semaphore

Semaphore能做什么?

Semaphore[ˈseməfɔ:(r)]（信号量）是用来控制同时访问特定资源的线程数量，通过协调各个线程以保证合理地使用公共资源。Semaphore可以用作流量控制，特别是公共资源有限的应用场景，比如数据库的连接。

有些情况下需要控制并发的线程数量，这一点是synchronized做不到的.Semaphore从字面上理解是信号量等含义,它所提供的功能完全是synchronized关键字的升级版。这个类的主要作用是控制线程并发的数量

Semaphore如何使用

Semaphore主要用于管理信号量，同样在创建Semaphore对象实例的时候通过传入构造参数设定可供管理的信号量的数值。简单说，信号量管理的信号就好比令牌，构造时传入令牌数量，也就是Semaphore控制并发的数量。线程在执行并发的代码前要先获取信号（通过aquire函数获取信号许可），执行后归还信号（通过release方法归还），每次acquire信号成功后，Semaphore可用的信号量就会减一，同样release成功之后，Semaphore可用信号量的数目会加一，如果信号量的数量减为0，acquire调用就会阻塞，直到release调用释放信号后，aquire才会获得信号返回。

信号量上定义两种操作:

- acquire(获取):当一个线程调用acquire操作时,它要么成功获取到信号量(信号量减1),要么一直等下去,直到有线程释放信号量,或超时,Semaphore内部会维护一个等待队列用于存储这些被暂停的线程.
- release(释放)实际上会将信号量的值+1,然后唤醒相应Sepmaphore实例的等待队列中的一个任意等待线程.

应用场景

信号量主要用于两个目的:

- 用于多个共享资源的互斥使用.
- 用于并发线程数的控制.

案例

```
/**
 * @Author:Hardy
 * @QQ:2937270766
 * @官网：http://www.yuandengta.com
 */
public class SemaphoreTest {
    //读取文件的线程数量
    private static final int THREAD_COUNT = 10;
    private static ExecutorService exec =
Executors.newFixedThreadPool(THREAD_COUNT);

    private static Semaphore semaphore = new Semaphore(3);
    public static void main(String[] args) {
        //创建线程读取数据，并尝试获取数据连接，将数据存储到数据库中
        for(int i = 0;i < THREAD_COUNT;i++){
            final int index = i;
            Runnable task = new Runnable() {
                @Override
                public void run() {
                    try {
                        //从远程读取数据
                        System.out.println("thread-" + index + "开始远程读取文件数
据");
                        //通过acquire函数获取数据库连接，如果成功将数据存储到数据库
                        semaphore.acquire();
                        System.out.println("thread-"+ index + "保存数据成功");
                        Thread.sleep(1000);
                    }catch(Exception e){
                        e.printStackTrace();
                    }finally {
                        semaphore.release();
                    }
                }
            };
            exec.execute(task);
        }
    }
}
```

也就是说当一个线程调用acquire方法获取许可的时候，它会首先获取许可，如果没有可用许可则进入共享模式，并且将当前线程加入等待队列去获取许可，直到获取成功才返回退出循环。

从执行结果不难发现，线程将数据读到内存后，先只能有三个线程先获取到数据库连接，完成数据存储后释放了信号量后，其他的线程才能获取数据库连接。

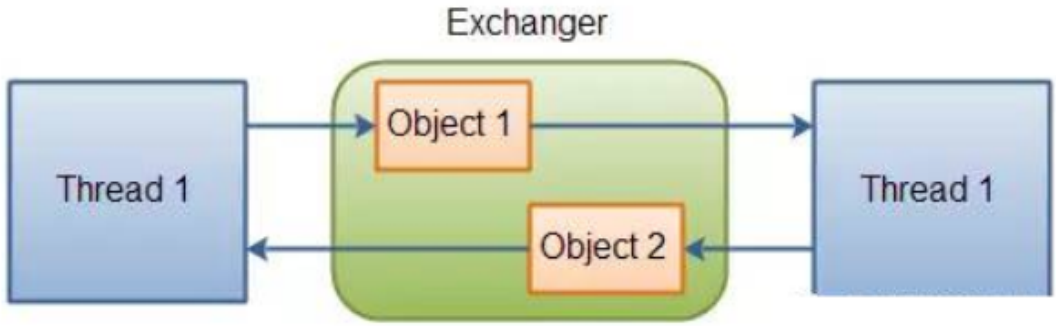
注意事项

- Semaphore.acquire()和Semaphore.release()总是配对使用的,这点需要由应用代码自身保证.
- Semaphore.release()调用应该放在finally块中,已避免应用代码出现异常的情况下,当前线程所获得的信号量无法返还.

- 如果Semaphore构造器中的参数permits值设置为1,所创建的Semaphore相当于一个互斥锁.与其他互斥锁不同的是,这种互斥锁允许一个线程释放另外一个线程所持有的锁.因为一个线程可以在未执行过Semaphore.acquire()的情况下执行相应的Semaphore.release().
- 默认情况下,Semaphore采用的是非公平性调度策略

七、Exchanger

Exchanger用于进行两个线程之间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。这两个线程通过exchange()方法交换数据，当一个线程先执行exchange()方法后，它会一直等待第二个线程也执行exchange()方法，当这两个线程到达同步点时，这两个线程就可以交换数据了。当线程 A 调用 Exchange 对象的 exchange 方法后，它会陷入阻塞状态，直到线程B也调用了 exchange 方法，然后以线程安全的方式交换数据，之后线程A和B继续运行。



exchange 方法有两个重载实现，在交换数据的时候还可以设置超时时间。如果一个线程在超时时间内没有其他线程与之交换数据，就会抛出 TimeoutException 超时异常。如果没有设置超时时间，则会一直等待。

```
//交换数据，并设置超时时间
public V exchange(V x, long timeout, TimeUnit unit)
throws InterruptedException, TimeoutException
//交换数据
public V exchange(V x) throws InterruptedException
```

案例

```
import java.util.concurrent.Exchanger;

/**
 * @Author:Hardy
 * @QQ:2937270766
 * @官网：http://www.yuandengta.com
 */
public class ExChangerDemo {
    static Exchanger<String> ec = new Exchanger<>();
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                String ownData = "毒品";
                System.out.println(Thread.currentThread().getName()+"准备交易出
去:"+ownData);
                try {
                    String changeData = ec.exchange(ownData);
                    Thread.sleep(2000);
```

```
                    System.out.println(Thread.currentThread().getName()+"交易回
来:"+changeData);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"贩毒者").start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                String ownData = "美元";
                System.out.println(Thread.currentThread().getName()+"准备交易出
去:"+ownData);
                try {
                    String changeData = ec.exchange(ownData);
                    Thread.sleep(3000);
                    System.out.println(Thread.currentThread().getName()+"交易回
来: "+changeData);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },"吸毒者").start();
    }
}
```

八、线程池详解

ThreadPoolExecutor是JUC提供的一类线程池工具，也是Java语言中应用场景最多的并发框架，可以说，几乎所有需要异步或者并发执行的，都可以使用Java线程池。java的线程池支持主要通过ThreadPoolExecutor来实现，我们使用的ExecutorService的各种线程池策略都是基于ThreadPoolExecutor实现的，所以ThreadPoolExecutor十分重要。

开发者的困境

1）**线程管理**：线程的创建、启动、销毁等工作；

2）**线程复用**：线程的创建是会给服务器带来一定开销的，如何减少频繁重复创建线程的开销；

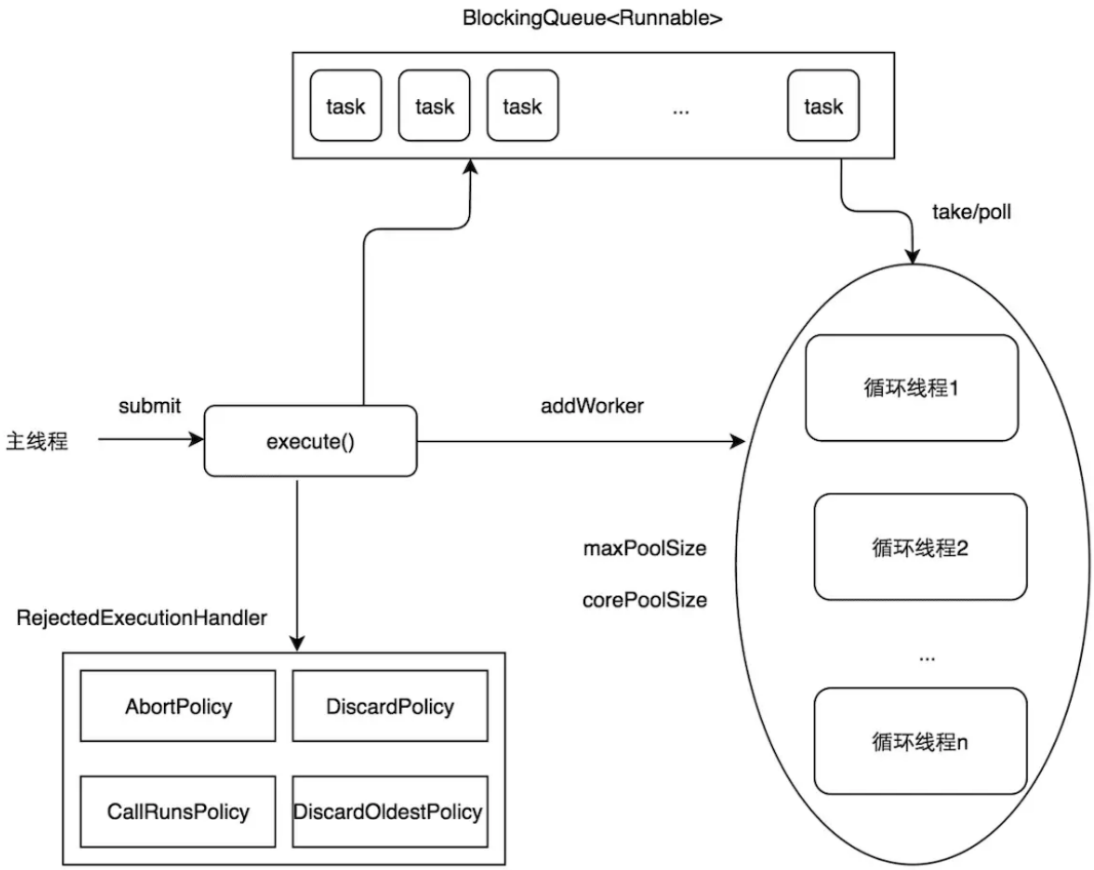
3）**弹性伸缩**：服务器通常有高峰期也有低峰期，线程池是否可以弹性伸缩，比如线程创建成功后长时间不使用是否可以回收，以减少系统资源的浪费，或者线程池的容量是否可以随时增长；

4）**拒绝策略**：线程数量有限而需要处理的任务很多，超出系统承载范围的任务是拒绝还是阻塞等待；

5）**异常处理**：线程在执行过程中可能遇到异常或者错误，开发者如何正确应对这些异常或者错误；

6）**任务分配**：任务的分配是基于先入先出还是基于某种优先级的策略。

线程池工作流程



1. 如果正在运行的线程数量小于corePoolSize，那么马上会创建线程运行这个任务
2. 如果正在运行的线程数量大于或等于corePoolSize，那么将这个任务放入队列
3. 如果这个时候队列满了，而且正在运行的线程数量小于maximumPoolSize，那么还是会创建非核心线程立刻运行这个任务
4. 如果队列满了，而且正在运行的线程数量大于或等于maximumPoolSize，那么线程池会按照拒绝策略处理。
5. 当一个线程完成任务时，它会从队列中取下一个任务来执行。

   当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

1）**AbortPolicy（中止）**：该策略将会丢弃任务，直接抛出RejectedExecutionException异常，调用者将会获得异常；

2）**DiscardPolicy（抛弃）**：使用该策略，线程池将会悄悄地丢弃这个任务而不被调用者知道；

3）**CallerRunsPolicy（调用者运行）**：该策略既不会抛弃任务也不会抛出异常，而是将这个任务退回给调用者，从而降低新任务的流量；

4）**DiscardOldestPolicy（抛弃最旧的）**：该策略将会抛弃下一个即将轮到执行的任务，那么“抛弃最旧”的将导致抛弃优先级最高的任务，因此最好不要把“抛弃最旧的”饱和策略和优先级队列放在一起使用； 这里，代码实现我们将只展示**CallerRunsPolicy（调用者运行）**策略： （如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序，如果再次失败，则重复此过程）

参数解释

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
```

```
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory,
                              RejectedExecutionHandler handler) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

int corePoolSize：该线程池中核心线程数最大值 。线程池新建线程的时候，如果当前线程总数小于corePoolSize，则新建的是核心线程，如果超过corePoolSize，则新建的是非核心线程。核心线程默认情况下会一直存活在线程池中，即使这个核心线程啥也不干(闲置状态)。

int maximumPoolSize：

该线程池中线程总数最大值，线程总数 = 核心线程数 + 非核心线程数。

long keepAliveTime

该线程池中非核心线程闲置超时时长。一个非核心线程，如果不干活(闲置状态)的时长超过这个参数所设定的时长，就会被销毁掉。

BlockingQueue workQueue

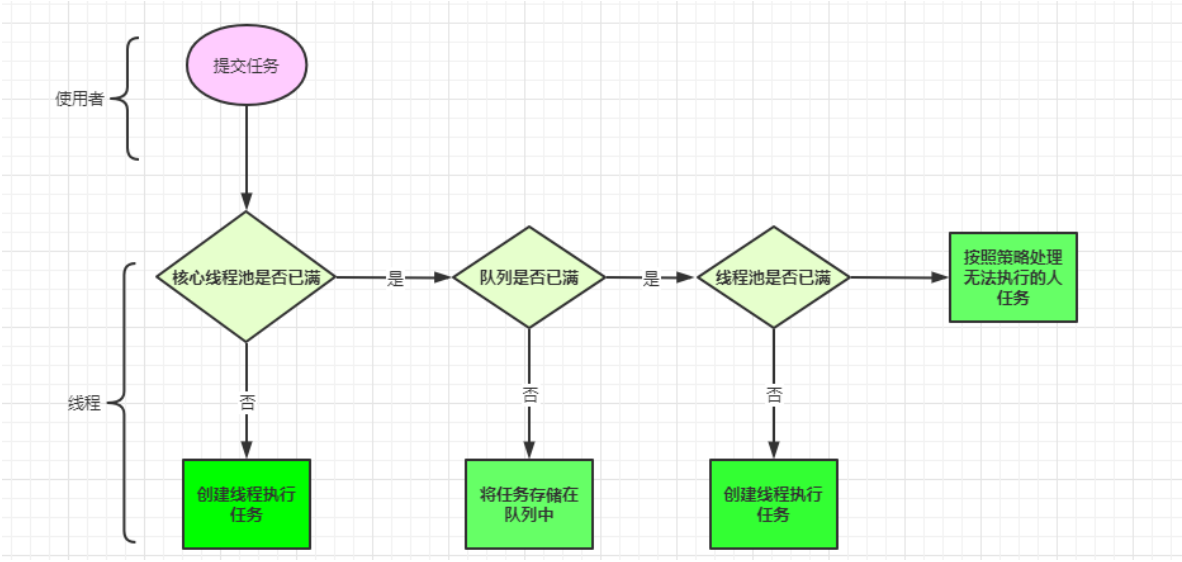
该线程池中的任务队列。维护着等待执行的Runnable对象当所有的核心线程都在干活时，新添加的任务会被添加到这个队列中等待处理，如果队列满了，则新建非核心线程执行任务。

unit：时间单位。为 keepAliveTime 指定时间单位。

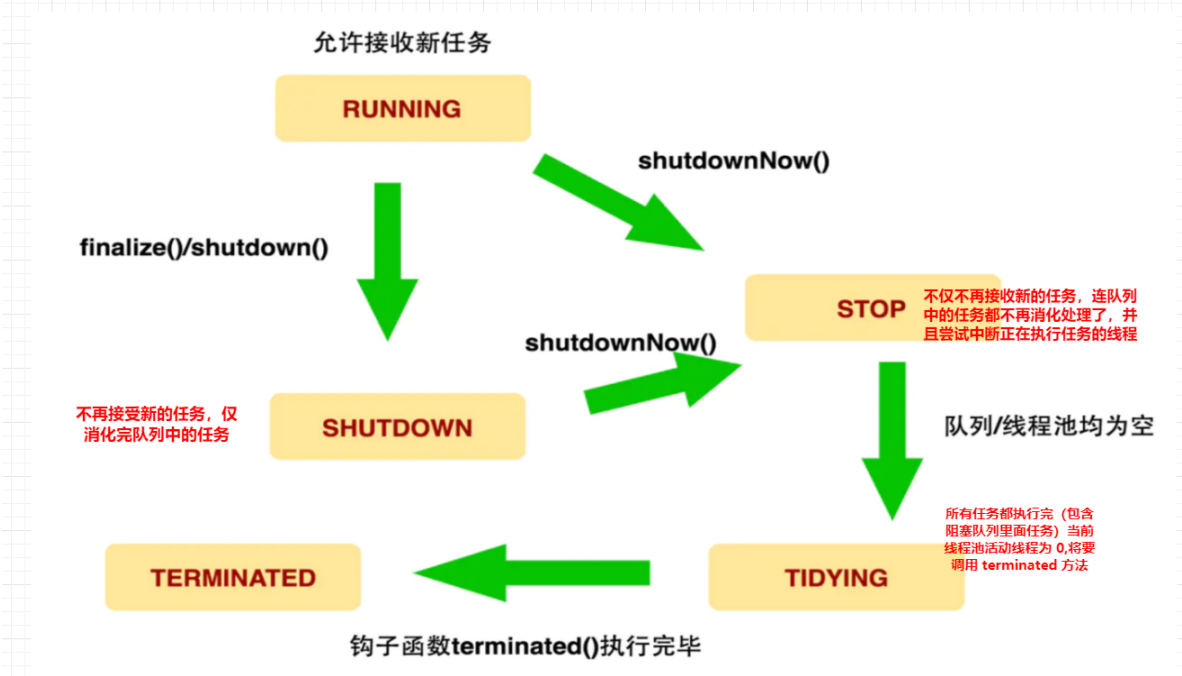
threadFactory 创建线程的工程类。可以通过指定线程工厂为每个创建出来的线程设置更有意义的名字，如果出现并发问题，也方便查找问题原因。

handler 执行拒绝策略的对象。当线程池的阻塞队列已满和指定的线程都已经开启，说明当前线程池已经处于饱和状态了，那么就需要采用一种策略来处理这种情况

逻辑图



线程池状态



1. ***RUNNING***:
    1. 状态说明：线程池处于RUNNING状态时，能够接受新的任务，以及对已添加的任务进行处理
    2. 状态切换：线程池的初始化状态时RUNNING。换句话说，线程一旦被阻塞，就处于RUNNING状态，并且线程池中任务数为0
2. ***SHUTDOWN***
    1. 状态说明：线程池处于shutdown状态时，不接受新的任务，但能处理已添加的任务
    2. 状态切换：调用线程池的shutdown()时，线程由RUNNING ---> SHUTDOWN
3. ***STOP***
    1. 状态说明：线程池处于stop状态时，不接受新的任务，不处理已添加的任务，并且会中断正在处理的任务
    2. 状态切换：调用线程池的shutdownNow时，线程池由RUNNING or SHUTDOWN ---> STOP
4. ***TIDYING***
    1. 状态说明：当所有的任务已终止，ctl记录的任务数量为0，线程池会变为tidying状态。当线程池变为TIDYING状态时，会执行钩子函数terminated()，terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行响应的处理，可以通过重载terminated函数来实现

2. 状态切换：当线程池在shutdown状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由shutdown ---> tidying。当线程池为stop状态下，线程池中执行的任务为空时，就会由stop -> tidying

5. ***TERMINATED***

   1. 状态说明：线程池彻底终止，就变成terminated状态
   2. 状态切换：线程池处在TIDYING状态时，执行完terminated之后，就会由TIDYING -> TERMINATED。

Executor接口

```
 /**
 * @since 1.5
 * @author Doug Lea
 */
public interface Executor {
    void execute(Runnable command);
}
```

当然了，Executor 这个接口只有提交任务的功能，太简单了，我们想要更丰富的功能，比如我们想知道执行结果、我们想知道当前线程池有多少个线程活着、已经完成了多少任务等等，这些都是这个接口的不足的地方。接下来我们要介绍的是继承自 `Executor` 接口的 `ExecutorService` 接口，这个接口提供了比较丰富的功能，也是我们最常使用到的接口。

ExecutorService

```
public interface ExecutorService extends Executor {

    /**
     *关闭线程池，已提交的任务继续执行，不接受继续提交新任务
     */
    void shutdown();

    /**
     * 关闭线程池，尝试停止正在执行的所有任务，不接受继续提交新任务
     * 它和前面的方法相比，加了一个单词now，区别在于他会停止当前正在进行的任务
     */
    List<Runnable> shutdownNow();

    /**
     * 线程池是否已经关闭
     */
    boolean isShutdown();

    /**
     * 如果调用了 shutdown() 或 shutdownNow() 方法后，所有任务结束了，那么返回true
     * 这个方法必须在调用shutdown或shutdownNow方法之后调用才会返回true
     */
    boolean isTerminated();

    /**
     *  等待所有任务完成，并设置超时时间
```

```
     我们这么理解，实际应用中是，先调用 shutdown 或 shutdownNow，
     然后再调这个方法等待所有的线程真正地完成，返回值意味着有没有超时
     */
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;

    /**
     * 提交一个 Callable 任务
     */
    <T> Future<T> submit(Callable<T> task);

    /**
     * 提交一个 Runnable 任务，第二个参数将会放到 Future 中，作为返回值
     * 因为 Runnable 的 run 方法本身并不返回任何东西
     */
    <T> Future<T> submit(Runnable task, T result);

    /**
     * 提交一个Runnable任务
     */
    Future<?> submit(Runnable task);

    /**
     * 执行所有任务，返回 Future 类型的一个 list
     */
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;

    /**
     * 也是执行所有任务，但是这里设置了超时时间
     */
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                  long timeout, TimeUnit unit)
        throws InterruptedException;

    /**
     * 只有其中的一个任务结束了，就可以返回，返回执行完的那个任务的结果
     */
    <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;

    /**
     * 同上一个方法，只有其中的一个任务结束了，就可以返回，返回执行完的那个任务的结果，
     * 不过这个带超时，超过指定的时间，抛出 TimeoutException 异常
     */
    <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                    long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

这些方法都很好理解，一个简单的线程池主要就是这些功能，能提交任务，能获取结果，能关闭线程池，这也是为什么我们经常用这个接口的原因。

常见创建线程的方式

1、FixedThreadPool：创建固定长度的线程池，每次提交任务创建一个线程，直到达到线程池的最大数量，线程池的大小不再变化。

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

- FixedThreadPool的corePoolSize和maxiumPoolSize都被设置为创建FixedThreadPool时指定的参数nThreads。
- 0L则表示当线程池中的线程数量操作核心线程的数量时，多余的线程将被立即停止
- 最后一个参数表示FixedThreadPool使用了无界队列LinkedBlockingQueue作为线程池的做工队列，由于是无界的，当线程池的线程数达到corePoolSize后，新任务将在无界队列中等待，因此线程池的线程数量不会超过corePoolSize，同时maxiumPoolSize也就变成了一个无效的参数，并且运行中的线程池并不会拒绝任务

执行过程如下：

1.如果当前工作中的线程数量少于corePool的数量，就创建新的线程来执行任务。

2.当线程池的工作中的线程数量达到了corePool，则将任务加入LinkedBlockingQueue。

3.线程执行完1中的任务后会从队列中去任务。

注意LinkedBlockingQueue是无界队列，所以可以一直添加新任务到线程池。

2、SingleThreadExecutor：SingleThreadExecutor是使用单个worker线程的Executor。特点是使用单个工作线程执行任务。它的构造源码如下：

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

SingleThreadExecutor的corePoolSize和maxiumPoolSize都被设置1。

执行过程如下：

1.如果当前工作中的线程数量少于corePool的数量，就创建一个新的线程来执行任务。

2.当线程池的工作中的线程数量达到了corePool，则将任务加入LinkedBlockingQueue。

3.线程执行完1中的任务后会从队列中去任务。

注意：由于在线程池中只有一个工作线程，所以任务可以按照添加顺序执行。

3、CachedThreadPool

CachedThreadPool是一个"无限"容量的线程池，它会根据需要创建新线程。特点是可以根据需要来创建新的线程执行任务，没有特定的corePool。下面是它的构造方法：

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

CachedThreadPool的corePoolSize被设置为0，即corePool为空；maximumPoolSize被设置为Integer.MAX_VALUE，即maximum是无界的。这里keepAliveTime设置为60秒，意味着空闲的线程最多可以等待任务60秒，否则将被回收。

CachedThreadPool使用没有容量的SynchronousQueue作为主线程池的工作队列，它是一个没有容量的阻塞队列。每个插入操作必须等待另一个线程的对应移除操作。这意味着，如果主线程提交任务的速度高于线程池中处理任务的速度时，CachedThreadPool会不断创建新线程。极端情况下，CachedThreadPool会因为创建过多线程而耗尽CPU资源。

4、newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

```
public class ScheduledExecutorServiceTest {

    public static void main(String[] args) throws Exception {
        ScheduledExecutorService timer =
Executors.newSingleThreadScheduledExecutor();

        TimerTask timerTask = new TimerTask(2000); // 任务需要 2000 ms 才能执行完毕

        System.out.printf("起始时间：%s\n\n", new
SimpleDateFormat("HH:mm:ss").format(new Date()));

        // 延时 1 秒后，按 3 秒的周期执行任务
        timer.scheduleAtFixedRate(timerTask, 1000, 3000, TimeUnit.MILLISECONDS);
    }

    private static class TimerTask implements Runnable {

        private final int sleepTime;
        private final SimpleDateFormat dateFormat;

        public TimerTask(int sleepTime) {
            this.sleepTime = sleepTime;
            dateFormat = new SimpleDateFormat("HH:mm:ss");
        }

        @Override
        public void run() {
            System.out.println("任务开始，当前时间：" + dateFormat.format(new
Date()));

            try {
                System.out.println("模拟任务运行...");
                Thread.sleep(sleepTime);
            } catch (InterruptedException ex) {
                ex.printStackTrace(System.err);
            }
```

```
            System.out.println("任务结束，当前时间：" + dateFormat.format(new
Date()));
            System.out.println();
        }

    }
}
```

FutureTask

Future接口和实现Future接口的FutureTask类，代表异步计算的结果。一个可取消的异步计算。FutureTask提供了对Future的基本实现，可以调用方法去开始和取消一个计算，可以查询计算是否完成并且获取计算结果。

我们知道，Runnable 的 void run() 方法是没有返回值的，所以，通常，如果我们需要的话，会在 submit 中指定第二个参数作为返回值：

```
<T> Future<T> submit(Runnable task, T result);
```

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

`cancel()`:cancel()方法用来取消异步任务的执行。如果异步任务已经完成或者已经被取消，或者由于某些原因不能取消，则会返回false。如果任务还没有被执行，则会返回true并且异步任务不会被执行。如果任务已经开始执行了但是还没有执行完成，若mayInterruptIfRunning为true，则会立即中断执行任务的线程并返回true，若mayInterruptIfRunning为false，则会返回true且不会中断任务执行线程。

`isCanceled()`:判断任务是否被取消，如果任务在结束(正常执行结束或者执行异常结束)前被取消则返回true，否则返回false。

`isDone()`:判断任务是否已经完成，如果完成则返回true，否则返回false。需要注意的是：任务执行过程中发生异常、任务被取消也属于任务已完成，也会返回true。

`get()`:获取任务执行结果，如果任务还没完成则会阻塞等待直到任务执行完成。如果任务被取消则会抛出CancellationException异常，如果任务执行过程发生异常则会抛出ExecutionException异常，如果阻塞等待过程中被中断则会抛出InterruptedException异常。

`get(long timeout,Timeunit unit)`:带超时时间的get()版本，如果阻塞等待过程中超时则会抛出TimeoutException异常

```
public class FutureTaskForMultiCompute {
    public static void main(String[] args) throws Exception {
        System.out.println("使用 Callable 获得返回结果：");

        List<FutureTask<Integer>> futureTasks = new ArrayList<>(10);
        // 新建 3个线程，每个线程分别负责累加 1~100
        for (int i = 0; i < 3; i++) {
            AccumCallable task = new AccumCallable();
```

```
            FutureTask<Integer> futureTask = new FutureTask<>(task);
            futureTasks.add(futureTask);
            Thread worker = new Thread(futureTask, "累加器线程" + i);
            worker.start();
        }

        int total = 0;
        for (FutureTask<Integer> futureTask : futureTasks) {
            total += futureTask.get(); // get() 方法会阻塞直到获得结果
        }
        System.out.println("累加的结果: " + total);
    }

    static final class AccumCallable implements Callable<Integer> {
        @Override
        public Integer call() throws Exception {
            int result = 0;
            for (int i = 0; i <= 100; i++) {
                result += i;
                Thread.sleep(100);
            }
            System.out.printf("(%s) - 运行结束，结果为 %d\n",
                    Thread.currentThread().getName(), result);
            return result;
        }
    }
}
```

源码

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    // 获取线程数
    int c = ctl.get();
    //如果当前线程数少于核心线程数，直接添加一个worker来执行任务
    //创建一个新的线程，并把当前任务command作为这个线程的第一个任务（firstTask）
    if (workerCountOf(c) < corePoolSize) {
        //添加任务成功，那么就结束了。提交任务嘛，线程池已经接受了这个任务，这个方法也可以返回
了
        //至于执行的结果，到时候会包装到FutrueTask中
        // 返回false代表线程池不允许提交任务
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 执行到这里说名：要么当前线程数大于等于核心线程数，要么刚刚addworker失败了
    // 如果线程池处于running状态，把这个任务添加到任务队列中workerQueue
    if (isRunning(c) && workQueue.offer(command)) {
         /* 这里面说的是，如果任务进入了 workQueue，我们是否需要开启新的线程
         * 因为线程数在 [0, corePoolSize) 是无条件开启新的线程
         * 如果线程数已经大于等于 corePoolSize，那么将任务添加到队列中，然后进到这里
         */
        int recheck = ctl.get();
        // 如果线程池已不处于 RUNNING 状态，那么移除已经入队的这个任务，并且执行拒绝策略
        if (! isRunning(recheck) && remove(command))
            reject(command);
```

```
            // 如果线程池还是 RUNNING 的，并且线程数为 0，那么开启新的线程
            // 到这里，我们知道了，这块代码的真正意图是：担心任务提交到队列中了，但是线程都关闭了
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        // 如果 workQueue 队列满了，那么进入到这个分支
        // 以 maximumPoolSize 为界创建新的 worker，
        // 如果失败，说明当前线程数已经达到 maximumPoolSize，执行拒绝策略
        else if (!addWorker(command, false))
            reject(command);
    }
```

```
// 第一个参数是准备提交给这个线程执行的任务，之前说了，可以为 null
// 第二个参数为 true 代表使用核心线程数 corePoolSize 作为创建线程的界线，也就说创建这个线程的时候，
//          如果线程池中的线程总数已经达到 corePoolSize，那么不能响应这次创建线程的请求
//          如果是 false，代表使用最大线程数 maximumPoolSize 作为界线
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

         // 这个非常不好理解
        // 如果线程池已关闭，并满足以下条件之一，那么不创建新的 worker：
        // 1. 线程池状态大于 SHUTDOWN，其实也就是 STOP, TIDYING, 或 TERMINATED
        // 2. firstTask != null
        // 3. workQueue.isEmpty()
        // 简单分析下：
        // 还是状态控制的问题，当线程池处于 SHUTDOWN 的时候，不允许提交任务，但是已有的任务继续执行
        // 当状态大于 SHUTDOWN 时，不允许提交任务，且中断正在执行的任务
        // 多说一句：如果线程池处于 SHUTDOWN，但是 firstTask 为 null，且 workQueue 非空，那么是允许创建 worker 的
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;

        for (;;) {
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            // 如果成功，那么就是所有创建线程前的条件校验都满足了，准备创建线程执行任务了
            // 这里失败的话，说明有其他线程也在尝试往线程池中创建线程
            if (compareAndIncrementWorkerCount(c))
                break retry;
            // 由于有并发，重新再读取一下 ctl
            c = ctl.get();  // Re-read ctl
            // 正常如果是 CAS 失败的话，进到下一个里层的for循环就可以了
            // 可是如果是因为其他线程的操作，导致线程池的状态发生了变更，如有其他线程关闭了这个线程池
            // 那么需要回到外层的for循环
```

```
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }

    /*
     * 到这里，我们认为在当前这个时刻，可以开始创建线程来执行任务了，
     * 因为该校验的都校验了，至于以后会发生什么，那是以后的事，至少当前是满足条件的
     */
     // worker 是否已经启动
    boolean workerStarted = false;
    // 是否已将这个 worker 添加到 workers 这个 HashSet 中
    boolean workerAdded = false;
    Worker w = null;
    try {
        // 把 firstTask 传给 worker 的构造方法
        w = new Worker(firstTask);
        // 取 worker 中的线程对象，之前说了，Worker的构造方法会调用 ThreadFactory 来创建一个新的线
            程
        final Thread t = w.thread;
        if (t != null) {
            // 这个是整个类的全局锁，持有这个锁才能让下面的操作“顺理成章”，
            // 因为关闭一个线程池需要这个锁，至少我持有锁的期间，线程池不会被关闭
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                int rs = runStateOf(ctl.get());

                // 小于 SHUTTDOWN 那就是 RUNNING，这个自不必说，是最正常的情况
                // 如果等于 SHUTDOWN，前面说了，不接受新的任务，但是会继续执行等待队列中的任务
                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    // worker 里面的 thread 可不能是已经启动的
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    // 加到 workers 这个 HashSet 中
                    workers.add(w);
                    int s = workers.size();
                    // largestPoolSize 用于记录 workers 中的个数的最大值
                    // 因为 workers 是不断增加减少的，通过这个值可以知道线程池的大小曾经达到的最大值

                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            // 添加成功的话，启动这个线程
            if (workerAdded) {
                t.start();
                workerStarted = true;
```

```
                }
            }
        } finally {
            // 如果线程没有启动，需要做一些清理工作，如前面 workCount 加了 1，将其减掉
            if (! workerStarted)
                addWorkerFailed(w);
        }
            // 返回线程是否启动成功
        return workerStarted;
    }
```