限制

JUC中提供了几个比较常用的并发工具类，比如CountDownLatch、CyclicBarrier、Semaphore。 其实在以前我们课堂的演示代码中，或多或少都有用到过这样一些api，接下来我们会带大家去深入研究一些常用的api。

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毕再执行。从命名可以解读到countdown是倒数的意思，类似于我们倒计时的概念。

countdownlatch提供了两个方法，一个是countDown，一个是await， countdownlatch初始化的时候需要传入一个整数，在这个整数倒数到0之前，调用了await方法的程序都必须要等待，然后通过countDown来倒数。

使用案例

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

    CountDownLatch countDownLatch=new CountDownLatch(3);

    new Thread(()->{

        countDownLatch.countDown();


    },"t1").start();


    new Thread(()->{

        countDownLatch.countDown();


    },"t2").start();


    new Thread(()->{

        countDownLatch.countDown();


    },"t3").start();
```

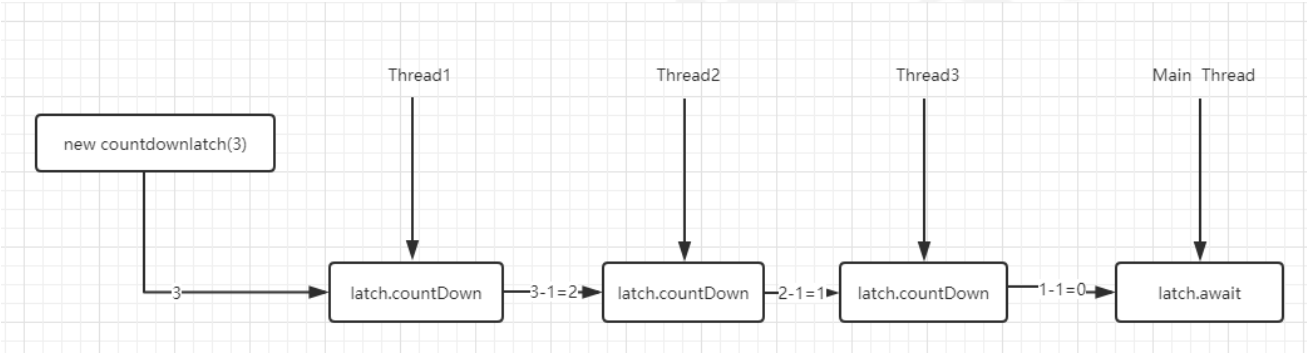
```
        countDownLatch.await();

        System.out.println("所有线程执行完毕");

    }
```

从代码的实现来看，有点类似join的功能，但是比join更加灵活。CountDownLatch构造函数会接收一个int类型的参数作为计数器的初始值，当调用CountDownLatch的countDown方法时，这个计数器就会减一。

通过await方法去阻塞去阻塞主流程



使用场景

\1. 通过countdownlatch实现最大的并行请求，也就是可以让N个线程同时执行，这个我也是在课堂上写得比较多的

\2. 比如应用程序启动之前，需要确保相应的服务已经启动，比如我们之前在讲zookeeper的时候，通过原生api连接的地方有用到countDownLatch

源码分析

CountDownLatch类存在一个内部类Sync，上节课我们讲过，它是一个同步工具，一定继承了AbstractQueuedSynchronizer。很显然，CountDownLatch实际上是是使得线程阻塞了，既然涉及到阻塞，就一定涉及到AQS队列

await

await函数会使得当前线程在countdownlatch倒计时到0之前一直等待，除非线程别中断；从源码中可以得知await方法会转发到Sync的acquireSharedInterruptibly

方法

public void await() **throws** InterruptedException { **sync**.acquireSharedInterruptibly(1); }

acquireSharedInterruptibly

这块代码主要是判断当前线程是否获取到了共享锁; 上一节课提到过，AQS有两种锁类型，一种是共享锁，一种是独占锁，在这里用的是共享锁； 为什么要用共享锁，因为CountDownLatch可以多个线程同时通过。

```
public final void acquireSharedInterruptibly(int arg)

        throws InterruptedException {
```

```
    if (Thread.interrupted()) //判断线程是否中断
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0) //如果等于0则返回1，否则返回-1，返回-1表示需要阻塞
        doAcquireSharedInterruptibly(arg);
}
```

在这里，state的意义是count，如果计数器为0，表示不需要阻塞，否则，只有在满足条件的情况下才会被唤醒

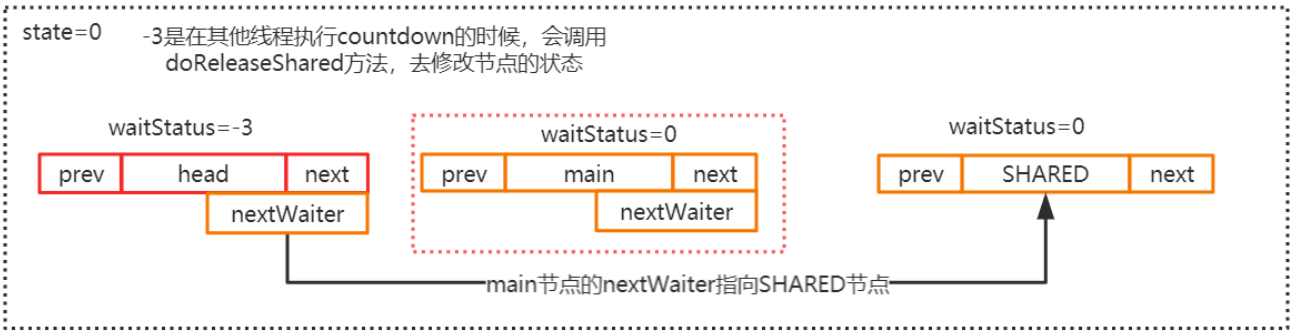
doAcquireSharedInterruptibly

获取共享锁

```
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.SHARED); //创建一个共享模式的节点添加到队列中
    boolean failed = true;
    try {
        for (;;) { //自旋等待共享锁释放，也就是等待计数器等于0。
            final Node p = node.predecessor(); //获得当前节点的前一个节点
            if (p == head) {
                int r = tryAcquireShared(arg);//就判断尝试获取锁
                if (r >= 0) {//r>=0表示计数器已经归零了，则释放当前的共享锁
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
            }
//当前节点不是头节点，则尝试让当前线程阻塞，第一个方法是判断是否需要阻塞，第二个方法是阻塞
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
```

```
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

setHeadAndPropagate



PROPAGATE：值为-3，表示releaseShared需要被传播给后续节点

```
private void setHeadAndPropagate(Node node, int propagate) {
    Node h = head; // 记录头节点
    setHead(node); //设置当前节点为头节点
//前面传过来的propagate是1，所以会进入下面的代码
    if (propagate > 0 || h == null || h.waitStatus < 0 ||
        (h = head) == null || h.waitStatus < 0) {
        Node s = node.next; //获得当前节点的下一个节点，如果下一个节点是空表示当前节点为最后一个节点，或者下一个节点是share节点
        if (s == null || s.isShared())
            doReleaseShared(); //唤醒下一个共享节点
    }
}
```

doReleaseShared

释放共享锁，通知后面的节点

```
private void doReleaseShared() {
      for (;;) {
        Node h = head; //获得头节点
        if (h != null && h != tail) { //如果头节点不为空且不等于tail节点
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) { //头节点状态为SIGNAL,
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0)) //修改当前头节点的状态为0,
避免下次再进入到这个里面
                    continue;            // loop to recheck cases
                unparkSuccessor(h); //释放后续节点
            }
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue;                // loop on failed CAS
        }
        if (h == head)                   // loop if head changed
            break;
    }
}
```

countdown

以共享模式释放锁，并且会调用tryReleaseShared函数，根据判断条件也可能会调用doReleaseShared函数

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) { //如果为true，表示计数器已归0了
        doReleaseShared(); //唤醒处于阻塞的线程
        return true;
    }
    return false;
}
```

tryReleaseShared

这里主要是对state做原子递减，其实就是我们构造的CountDownLatch的计数器，如果等于0返回true，否则返回false

```
protected boolean tryReleaseShared(int releases) {
    // Decrement count; signal when transition to zero
    for (;;) {
        int c = getState();
        if (c == 0)
            return false;
        int nextc = c-1;
        if (compareAndSetState(c, nextc))
            return nextc == 0;
    }
}
```

Semaphore

semaphore也就是我们常说的信号灯，semaphore可以控制同时访问的线程个数，通过acquire获取一个许可，如果没有就等待，通过release释放一个许可。有点类似限流的作用。叫信号灯的原因也和他的用处有关，比如某商场就5个停车位，每个停车位只能停一辆车，如果这个时候来了10辆车，必须要等前面有空的车位才能进入。

使用案例

```
public class Test {
```

```
    public static void main(String[] args) {
        Semaphore semaphore=new Semaphore(5);
        for(int i=0;i<10;i++){
            new Car(i,semaphore).start();
        }
    }
    static class Car extends Thread{
        private int num;
        private Semaphore semaphore;

        public Car(int num, Semaphore semaphore) {
            this.num = num;
            this.semaphore = semaphore;
        }
        public void run(){
            try {
                semaphore.acquire();//获取一个许可
                System.out.println("第"+num+"占用一个停车位");
                TimeUnit.SECONDS.sleep(2);
                System.out.println("第"+num+"俩车走喽");
                semaphore.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}
```

使用场景

可以实现对某些接口访问的限流

源码分析

semaphore也是基于AQS来实现的，内部使用state表示许可数量；它的实现方式和CountDownLatch的差异点在于acquireSharedInterruptibly中的tryAcquireShared方法的实现，这个方法是在Semaphore方法中重写的

acquireSharedInterruptibly

```
public final void acquireSharedInterruptibly(int arg)

        throws InterruptedException {

    if (Thread.interrupted())

        throw new InterruptedException();

    if (tryAcquireShared(arg) < 0)

        doAcquireSharedInterruptibly(arg);

}
```

tryAcquireShared

在semaphore中存在公平和非公平的方式，和重入锁是一样的，如果通过FairSync表示公平的信号量、NonFairSync表示非公平的信号量；公平和非公平取决于是否按照FIFO队列中的顺序去分配Semaphore所维护的许可，我们来看非公平锁的实现

nonfairTryAcquireShared

自旋去获得一个许可，如果许可获取失败，也就是remaining<0的情况下，让当前线程阻塞

```
final int nonfairTryAcquireShared(int acquires) {

    for (;;) {

        int available = getState();

        int remaining = available - acquires;

        if (remaining < 0 ||

            compareAndSetState(available, remaining))

            return remaining;

    }
```

```
}
```

releaseShared

releaseShared方法的逻辑也很简单，就是通过线程安全的方式去增加一个许可，如果增加成功，则触发释放一个共享锁，也就是让之前处于阻塞的线程重新运行

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

增加令牌数

```
protected final boolean tryReleaseShared(int releases) {
    for (;;) {
        int current = getState();
        int next = current + releases;
        if (next < current) // overflow
            throw new Error("Maximum permit count exceeded");
        if (compareAndSetState(current, next))
            return true;
    }
}
```

原子操作

当在多线程情况下，同时更新一个共享变量，由于我们前面讲过的原子性问题，可能得不到预期的结果。如果要达到期望的结果，可以通过synchronized来加锁解决，因为synchronized会保证多线程对共享变量的访问进行排队。

在Java5以后，提供了原子操作类，这些原子操作类提供了一种简单、高效以及线程安全的更新操作。而由于变量的类型很多，所以Atomic一共提供了12个类分别对应四种类型的原子更新操作，基本类型、数组类型、引用类型、属性类型

基本类型对应：AtomicBoolean、AtomicInteger、AtomicLong

数组类型对应：AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray

引用类型对应：AtomicReference、AtomicReferenceFieldUpdater、AtomicMarkableReference

字段类型对应：AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicStampedReference

Atomic原子操作的使用

```
private static AtomicInteger count=new AtomicInteger(0);


public static synchronized void inc() {
    try {
        Thread.sleep(1);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    count.getAndIncrement();
}
public static void main(String[] args) throws InterruptedException {
    for(int i=0;i<1000;i++){
        new Thread(()-> {
            SafeDemo.inc();
        }).start();
    }
    Thread.sleep(4000);
    System.out.println(count.get());
}
```

AtomicInteger实现原理

由于所有的原子操作类都是大同小异的，所以我们只分析其中一个原子操作类

```
public final int getAndIncrement() {

    return unsafe.getAndAddInt(this, valueOffset, 1);

}
```

大家又会发现一些熟悉的东西，就是unsafe。调用unsafe类中的getAndAddInt方法，这个方法如下

```
public final int getAndAddInt(Object var1, long var2, int var4) {

    int var5;

    do {

        var5 = this.getIntVolatile(var1, var2);// 方法获取对象中offset偏移地址对应的整型
field的值

    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;

}
```

通过循环以及cas的方式实现原子更新，从而达到在多线程情况下仍然能够保证原子性的目的。大家会发现，只要基本的东西搞明白以后，剩下的都比较容易理解了

线程池

Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。线程池就像数据库连接池的作用类似，只是线程池是用来重复管理线程避免创建大量线程增加开销。所以合理的使用线程池可以

\1. 降低创建线程和销毁线程的性能开销

\2. 合理的设置线程池大小可以避免因为线程数超出硬件资源瓶颈带来的问题，类似起到了限流的作用；线程是稀缺资源，如果无线创建，会造成系统稳定性问题

线程池的使用

JDK 为我们内置了几种常见线程池的实现，均可以使用 Executors 工厂类创建

为了更好的控制多线程，JDK提供了一套线程框架Executor，帮助开发人员有效的进行线程控制。它们都在java.util.concurrent包中，是JDK并发包的核心。

其中有一个比较重要的类:Executors，他扮演着线程工厂的角色，我们通过Executors可以创建特定功能的线程池

newFixedThreadPool：**该方法返回一个固定数量的线程池，线程数不变，当有一个任务提交时，若线程池中空闲，则立即执行，若没有，则会被暂缓在一个任务队列中，等待有空闲的线程去执行。

newSingleThreadExecutor: 创建一个线程的线程池，若空闲则执行，若没有空闲线程则暂缓在任务队列中。

newCachedThreadPool：**返回一个可根据实际情况调整线程个数的线程池，不限制最大线程数量，若用空闲的线程则执行任务，若无任务则不创建线程。并且每一个空闲线程会在60秒后自动回收

newScheduledThreadPool: 创建一个可以指定线程的数量的线程池，但是这个线程池还带有延迟和周期性执行任务的功能，类似定时器。

```
public class Test implements Runnable{


    @Override
    public void run() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName());
    }


    static ExecutorService service=Executors.newFixedThreadPool(3);
    public static void main(String[] args) {
        for(int i=0;i<100;i++) {
            service.execute(new Test());
        }
        service.shutdown();
    }
}
```

设置了3个固定线程大小的线程池来跑100

submit和execute的区别

执行一个任务，可以使用submit和execute，这两者有什么区别呢?

\1. execute只能接受Runnable类型的任务

\2. submit不管是Runnable还是Callable类型的任务都可以接受，但是Runnable返回值均为void，所以使用Future的get()获得的还是null

ThreadpoolExecutor

前面说的四种线程池构建工具，都是基于ThreadPoolExecutor 类，它的构造函数参数

```
public ThreadPoolExecutor(int corePoolSize, //核心线程数量

                          int maximumPoolSize, //最大线程数

                          long keepAliveTime, //超时时间,超出核心线程数量以外的线程空余存活时间

                          TimeUnit unit, //存活时间单位

                          BlockingQueue<Runnable> workQueue, //保存执行任务的队列
                      ThreadFactory threadFactory,//创建新线程使用的工厂

RejectedExecutionHandler handler //当任务无法执行的时候的处理方式
) {

    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,

         Executors.defaultThreadFactory(), defaultHandler);

}
```

分别看一下前面提到的几个初始化工具类的构造以及原理

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {

    return new ThreadPoolExecutor(nThreads, nThreads,

                                  0L, TimeUnit.MILLISECONDS,

                                  new LinkedBlockingQueue<Runnable>());

}
```

FixedThreadPool 的核心线程数和最大线程数都是指定值，也就是说当线程池中的线程数超过核心线程数后，任务都会被放到阻塞队列中。另外 keepAliveTime 为 0，也就是超出核心线程数量以外的线程空余存活时间

而这里选用的阻塞队列是 LinkedBlockingQueue，使用的是默认容量 Integer.MAX_VALUE，相当于没有上限

这个线程池执行任务的流程如下：

\1. 线程数少于核心线程数，也就是设置的线程数时，新建线程执行任务

\2. 线程数等于核心线程数后，将任务加入阻塞队列

\3. 由于队列容量非常大，可以一直添加

\4. 执行完任务的线程反复去队列中取任务执行

用途：FixedThreadPool 用于负载比较大的服务器，为了资源的合理利用，需要限制当前线程数量

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool() {

    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,

                                  60L, TimeUnit.SECONDS,

                                  new SynchronousQueue<Runnable>());

}
```

CachedThreadPool 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; 并且没有核心线程，非核心线程数无上限，但是每个空闲的时间只有 60 秒，超过后就会被回收。

它的执行流程如下：

\1. 没有核心线程，直接向 SynchronousQueue 中提交任务

\2. 如果有空闲线程，就去取出任务执行；如果没有空闲线程，就新建一个

\3. 执行完任务的线程有 60 秒生存时间，如果在这个时间内可以接到新任务，就可以继续活下去，否则就被回收

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

线程池的源码分析

ThreadPoolExecutor是线程池的核心，提供了线程池的实现。ScheduledThreadPoolExecutor继承了ThreadPoolExecutor，并另外提供一些调度方法以支持定时和周期任务。Executers是工具类，主要用来创建线程池对象

我们把一个任务提交给线程池去处理的时候，线程池的处理过程是什么样的呢？首先直接来看看定义

线程数量和线程池状态管理

线程池用一个AtomicInteger来保存 [线程数量] 和 [线程池状态] ,一个int数值一共有32位,高3位用于保存运行状态,低29位用于保存线程数量

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0)); //一个原子操作类
```

```
private static final int COUNT_BITS = Integer.SIZE - 3; //32-3

private static final int CAPACITY   = (1 << COUNT_BITS) - 1; //将1的二进制向右位移29位,再减1表示最大线程容量


//运行状态保存在int值的高3位（所有数值左移29位）

private static final int RUNNING    = -1 << COUNT_BITS;// 接收新任务,并执行队列中的任务

private static final int SHUTDOWN   =  0 << COUNT_BITS;// 不接收新任务,但是执行队列中的任务

private static final int STOP       =  1 << COUNT_BITS;// 不接收新任务,不执行队列中的任务,中断正在执行中的任务

private static final int TIDYING    =  2 << COUNT_BITS; //所有的任务都已结束,线程数量为0,处于该状态的线程池即将调用terminated()方法

private static final int TERMINATED =  3 << COUNT_BITS;// terminated()方法执行完成



// Packing and unpacking ctl

private static int runStateOf(int c)     { return c & ~CAPACITY; } //获取运行状态

private static int workerCountOf(int c)  { return c & CAPACITY; } //获取线程数量
```

execute

通过线程池的核心方法了解线程池中这些参数的含义

```
public void execute(Runnable command) {

    if (command == null)

        throw new NullPointerException();

       int c = ctl.get();

    if (workerCountOf(c) < corePoolSize) {//1.当前池中线程比核心数少，新建一个线程执行任务

        if (addWorker(command, true))

            return;

        c = ctl.get();

    }
```

```
    if (isRunning(c) && workQueue.offer(command)) {//2.核心池已满，但任务队列未满，添加到队列中

        int recheck = ctl.get();
//任务成功添加到队列以后，再次检查是否需要添加新的线程，因为已存在的线程可能被销毁了

        if (! isRunning(recheck) && remove(command))

            reject(command);//如果线程池处于非运行状态，并且把当前的任务从任务队列中移除成功，则拒绝该任务

        else if (workerCountOf(recheck) == 0)//如果之前的线程已被销毁完，新建一个线程

            addWorker(null, false);

    }

    else if (!addWorker(command, false)) //3.核心池已满，队列已满，试着创建一个新线程

        reject(command);  //如果创建新线程失败了，说明线程池被关闭或者线程池完全满了，拒绝任务

}
```

流程图