线程池

[b站]https://www.bilibili.com/video/BV1BK4y157Ro

线程池的创建

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


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

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

ThreadPoolExecutor

- 线程的复用
- 线程数量的限制

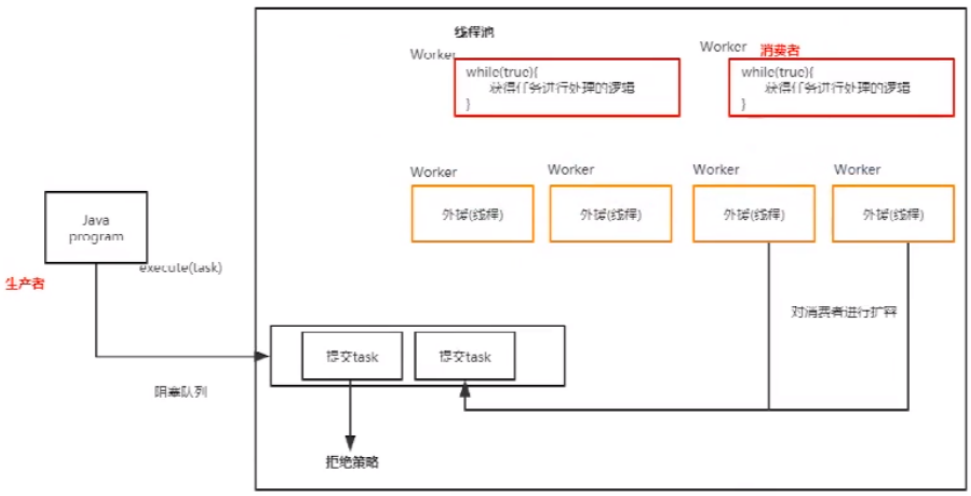
```
public ThreadPoolExecutor(int corePoolSize, //核心线程数（当前线程池里面能够长期运行的线程）
                          int maximumPoolSize,//最大线程数量（当前线程池中最多能创建的线程）
                          long keepAliveTime, //线程存活单位
                          TimeUnit unit,//存活单位
                          BlockingQueue<Runnable> workQueue,//队列
                          ThreadFactory threadFactory,//线程工厂
                          RejectedExecutionHandler handler) {//拒绝策略
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.acc = System.getSecurityManager() == null ?
                null :
                AccessController.getContext();
        this.corePoolSize = corePoolSize;
```

```
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

如何实现线程的复用

- 阻塞队列实现生产者消费者
- 通过`while()`避免线程结束

```
 public static class MyThread extends Thread{
     //在`java`里面只能控制`run`方法
     //线程执行的是run方法
     //如果run方法执行结束，那么当前线程就会被销毁
        public void run(){
            // 不让run方法执行结束
            while (true){
                //如果死循环，有任务来就执行，没有任务就阻塞
                //sychronized(condition){}
                //JVM wait/notify -> java.util.concurrent
condition.await/signal
                //
            }
        }
    }
    public static void main(String[] args){
       new Thread(new MyThread()).start();
    }
```



通信的数据结构：阻塞队列`BlockingDeque`实现生产者消费者

- 如果队列满了，生产者继续往队列添加数据，生产者会阻塞
- 如果队列为空，生产者继续获取数据，消费者会阻塞

实现线程池

- 配置线程的数量
- 配置队列（存储任务的）
- 拒绝策略（队列放不下了）

`ThreadPoolExecutor`的`execute`方法源码分析

```
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        /*
         * Proceed in 3 steps:
         *
         * 1. If fewer than corePoolSize threads are running, try to
         * start a new thread with the given command as its first
         * task.  The call to addWorker atomically checks runState and
         * workerCount, and so prevents false alarms that would add
         * threads when it shouldn't, by returning false.
         *
         * 2. If a task can be successfully queued, then we still need
         * to double-check whether we should have added a thread
         * (because existing ones died since last checking) or that
         * the pool shut down since entry into this method. So we
         * recheck state and if necessary roll back the enqueuing if
         * stopped, or start a new thread if there are none.
         *
         * 3. If we cannot queue task, then we try to add a new
         * thread.  If it fails, we know we are shut down or saturated
         * and so reject the task.
         */
        int c = ctl.get();
        //工作线程数量小于核心线程数量 执行的时候未必初始化
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))//创建工作线程（核心）
                return;
            c = ctl.get();
        }
        //任务直接丢队列里面
        //offer返回添加结果 非阻塞
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);//添加工作线程
        }
        //直接添加一个非核心线程 false表示非核心
        else if (!addWorker(command, false)) //添加失败（外援）
            reject(command);//拒绝
    }
```

addWorker源码

- 线程池里面每一个线程叫一个worker
- 每个worker循环从队列里面拿任务
- 队列为空的情况下`getTask`会阻塞

```
public void run() {
            runWorker(this);
        }
```

```
final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
        try {
            while (task != null || (task = getTask()) != null) {
                w.lock();
                // If pool is stopping, ensure thread is interrupted;
                // if not, ensure thread is not interrupted.  This
                // requires a recheck in second case to deal with
                // shutdownNow race while clearing interrupt
             if ((runStateAtLeast(ctl.get(), STOP) ||
                     (Thread.interrupted() &&
                      runStateAtLeast(ctl.get(), STOP))) &&
                    !wt.isInterrupted())
                    wt.interrupt();
                try {
                    beforeExecute(wt, task);
                    Throwable thrown = null;
                    try {
                        //调用线程类的实例方法
                        //Runnable是接口
                        task.run();
                    } catch (RuntimeException x) {
                        thrown = x; throw x;
                    } catch (Error x) {
                        thrown = x; throw x;
                    } catch (Throwable x) {
                        thrown = x; throw new Error(x);
                    } finally {
                        afterExecute(task, thrown);
                    }
                } finally {
                    task = null;
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            processWorkerExit(w, completedAbruptly);
```

```
        }
    }
```

线程的回收

- 当超时的时候返回 `null` 因此 `runWorker` 里面 `while` 循环就会结束，线程就会回收

```
if ((wc > maximumPoolSize || (timed && timedOut))
                && (wc > 1 || workQueue.isEmpty())) {
                if (compareAndDecrementWorkerCount(c))
                    return null;
                continue;
            }

            try {
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();
                if (r != null)
                    return r;
                timedOut = true;
            } catch (InterruptedException retry) {
                timedOut = false;
            }


        // null的话线程就结束了
    while (task != null || (task = getTask()) != null) {
```