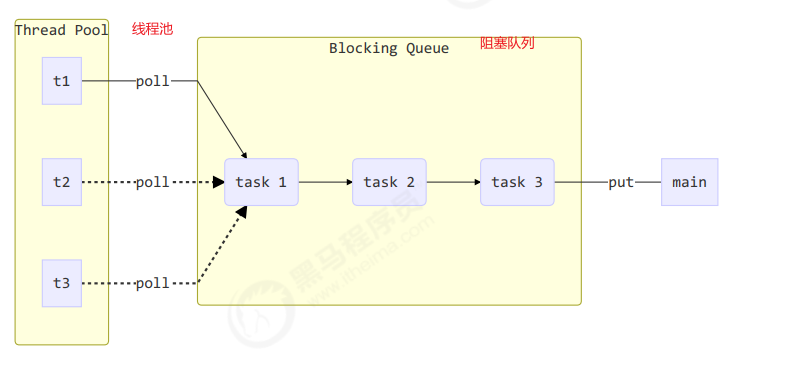
线程池

1、自定义线程池

充分利用已有的线程，享元模式的应用。

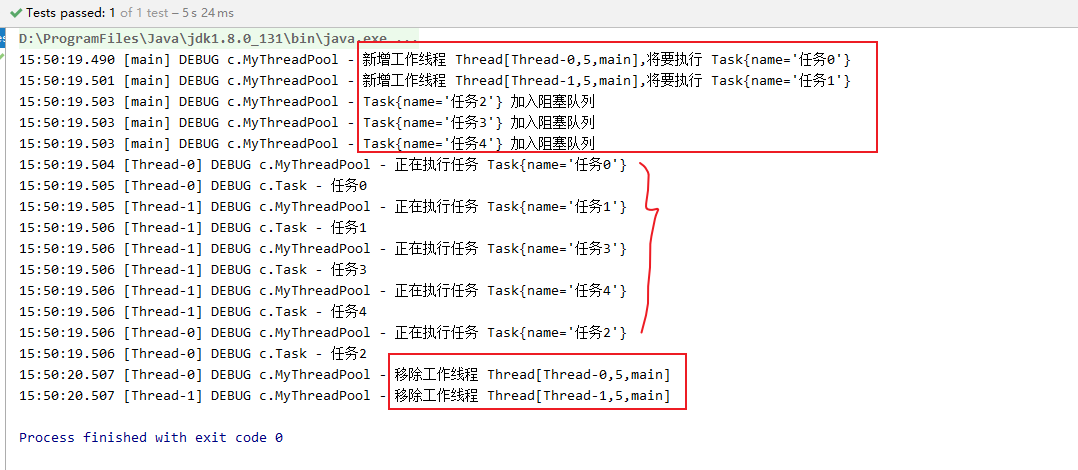


生产者消费者模式,任务放在阻塞队列，线程池中的线程处理阻塞队列中的任务。

1.1、take死等/poll超时等待代码

**package** com.concurrent.p9;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.sql.Time;  
**import** java.util.ArrayDeque;  
**import** java.util.Deque;  
**import** java.util.HashSet;  
**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Condition;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
*/\*\*  
 \* 自定义线程池  
 \*/*@Slf4j(topic = **"c.Test\_MyThreadPool"**)  
**public class** Test\_MyThreadPool {  
  
 @Test  
 **public void** test\_MyThreadPool() {  
 MyThreadPool<Task> myThreadPool =  
 **new** MyThreadPool<>(2, 1000, TimeUnit.***MILLISECONDS***, 5);  
 **for** (**int** i = 0; i < 5; i++) {  
 **int** j = i;  
 myThreadPool.execute(**new** Task(**"任务"** + j));  
 }  
 **try** {  
 Thread.*sleep*(5000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}  
  
*/\*\*  
 \* 线程池实现  
 \*/*@Slf4j(topic = **"c.MyThreadPool"**)  
**class** MyThreadPool<T> {  
 *//阻塞队列* **private** BlockQueue<Task> **taskQueue**;  
 *//线程集合* **private** HashSet<Worker> **workers** = **new** HashSet<>();  
 *//核心线程数* **private int coreSize**;  
 *//获取任务的超时时间* **private long timeout**;  
 *//时间单位* **private** TimeUnit **unit**;  
  
 **public** MyThreadPool(**int** coreSize, **long** timeout, TimeUnit unit, **int** capacity) {  
 **this**.**coreSize** = coreSize;  
 **this**.**timeout** = timeout;  
 **this**.**unit** = unit;  
 **taskQueue** = **new** BlockQueue<>(capacity);  
 }  
  
 *//执行任务* **public void** execute(Task task) {  
 **synchronized** (**workers**) {  
 **if** (**workers**.size() < **coreSize**) { *//如果任务数小于核心数，则直接执行* Worker worker = **new** Worker(task);  
 ***log***.debug(**"新增工作线程 {},将要执行 {}"**, worker, task);  
 **workers**.add(worker);  
 worker.start();  
 } **else** { *//如果任务书大于核心数，则放入阻塞队列* ***log***.debug(**"{} 加入阻塞队列"**, task);  
 **taskQueue**.put(task);  
 }  
 }  
 }  
  
 *//Work线程对象* **class** Worker **extends** Thread {  
 **private** Task **task**;  
  
 **public** Worker(Task task) {  
 **this**.**task** = task;  
 }  
  
 @Override  
 **public void** run() {  
 **while** (**task** != **null** || (**task** = **taskQueue**.poll(1000, **unit**)) != **null**) {  
 **try** {  
 ***log***.debug(**"正在执行任务 {}"**, **task**);  
 **task**.run();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **task** = **null**;  
 }  
 }  
 **synchronized** (**workers**) {  
 ***log***.debug(**"移除工作线程 {}"**, **this**);  
 *//执行完任务，将工作线程移除* **workers**.remove(**this**);  
 }  
 }  
 }  
  
  
}  
  
@Slf4j(topic = **"c.Task"**)  
**class** Task **implements** Runnable {  
 **private** String **name**;  
  
 **public** Task(String name) {  
 **this**.**name** = name;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Task{"** +  
 **"name='"** + **name** + **'\''** +  
 **'}'**;  
 }  
  
 @Override  
 **public void** run() {  
 ***log***.debug(**"{}"**, **name**);  
 }  
}  
  
*/\*  
 阻塞队列实现  
 \*/*@Slf4j(topic = **"c.BlockQueue"**)  
**class** BlockQueue<T> {  
 *//1.队列对象* **private** Deque<T> **queue** = **new** ArrayDeque<>();  
 *//2.阻塞队列容量* **private int capacity**;  
 *//3.锁* **private** ReentrantLock **lock** = **new** ReentrantLock();  
 *//4.阻塞队列为空的条件变量* **private** Condition **emptyWaitSet** = **lock**.newCondition();  
 *//5.阻塞队列为满的条件变量* **private** Condition **fullWaitSet** = **lock**.newCondition();  
  
 **public** BlockQueue(**int** capacity) {  
 **this**.**capacity** = capacity;  
 }  
  
 *//阻塞获取* **public** T take() {  
 **try** {  
 **lock**.lock();  
 *//如果阻塞队列为空，则等待* **while** (**queue**.size() == 0) {  
 **try** {  
 **emptyWaitSet**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 *//阻塞队列不为空，取出取出一个对象后唤醒生产者线程* T t = **queue**.removeFirst();  
 **fullWaitSet**.signal();  
 **return** t;  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *//带超时阻塞获取* **public** T poll(**long** timeout, TimeUnit unit) {  
 **try** {  
 **lock**.lock();  
 *//纳秒* **long** nano = unit.toNanos(timeout);  
 **while** (**queue**.size() == 0) {  
 **try** {  
 **if** (nano <= 0) {  
 **return null**;  
 }  
 *//awaitNanos方法返回超时时间-经历时间，将返回值再次赋值给nano，可解决虚假唤醒问题* nano = **emptyWaitSet**.awaitNanos(nano);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 T t = **queue**.removeFirst();  
 **fullWaitSet**.signal();  
 **return** t;  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *//阻塞添加* **public void** put(T task) {  
 **try** {  
 **lock**.lock();  
 *//如果阻塞队列满，则等待* **while** (**queue**.size() == **capacity**) {  
 **try** {  
 **fullWaitSet**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 *//阻塞队列不满，添加后唤醒消费者线程* **queue**.addLast(task);  
 **emptyWaitSet**.signal();  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *//阻塞队列大小* **public int** size() {  
 **try** {  
 **lock**.lock();  
 **return queue**.size();  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
}

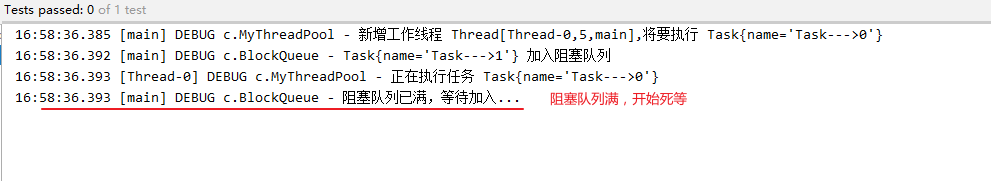
运行结果：



1.2、当任务队列已满时，要设置拒绝策略

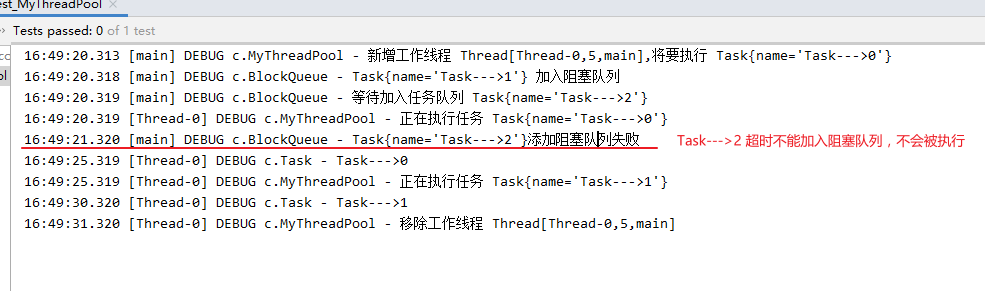
策略1：死等

MyThreadPool<Task> myThreadPool =  
 **new** MyThreadPool<>(1, 1000, TimeUnit.***MILLISECONDS***, 1,  
 ((queue, task) -> queue.put(task)));



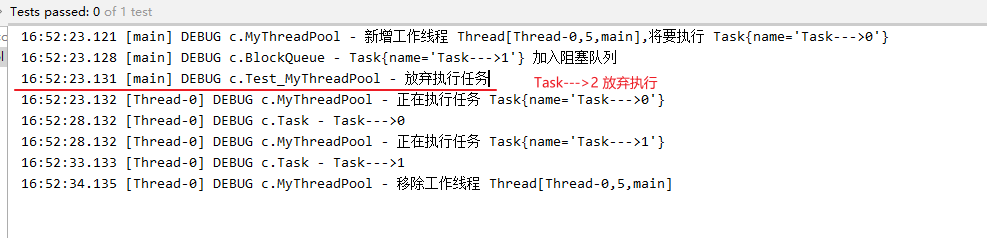
策略2：带超时等待

MyThreadPool<Task> myThreadPool =  
 **new** MyThreadPool<>(1, 1000, TimeUnit.***MILLISECONDS***, 1,  
 ((queue, task) -> queue.offer(task, 1000, TimeUnit.***MILLISECONDS***)));



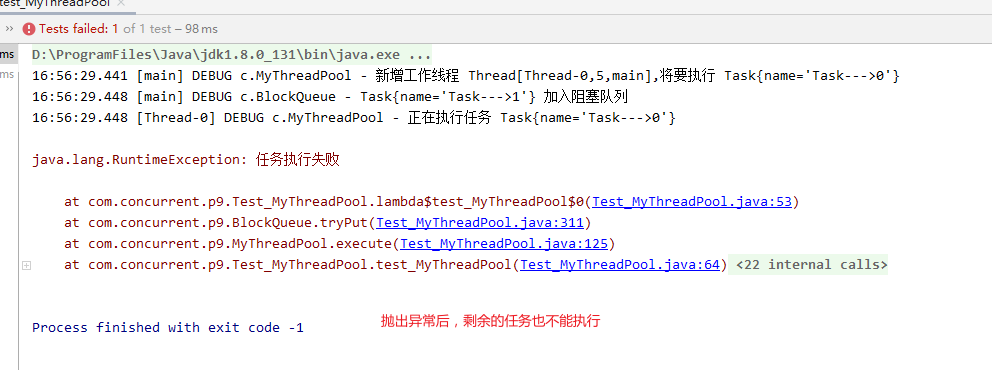
策略3：阻塞队列满时，放弃任务

MyThreadPool<Task> myThreadPool =  
 **new** MyThreadPool<>(1, 1000, TimeUnit.***MILLISECONDS***, 1,  
 ((queue, task) -> ***log***.debug(**"放弃执行任务"**)));



策略4：抛出异常

MyThreadPool<Task> myThreadPool =  
 **new** MyThreadPool<>(1, 1000, TimeUnit.***MILLISECONDS***, 1,  
 ((queue, task) -> {  
 **throw new** RuntimeException(**"任务执行失败"**);  
 }));



策略5：任务本身调用

完整代码

**package** com.concurrent.p9;  
  
**import** lombok.extern.slf4j.Slf4j;  
**import** org.junit.Test;  
  
**import** java.util.ArrayDeque;  
**import** java.util.Deque;  
**import** java.util.HashSet;  
**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Condition;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
*/\*\*  
 \* 自定义线程池  
 \*/*@Slf4j(topic = **"c.Test\_MyThreadPool"**)  
**public class** Test\_MyThreadPool {  
  
 @Test  
 **public void** test\_MyThreadPool() {  
 */\*\*  
 \* 策略1：死等  
 \* MyThreadPool<Task> myThreadPool =  
 \* new MyThreadPool<>(1, 1000, TimeUnit.MILLISECONDS, 1,  
 \* ((queue, task) -> queue.put(task)));  
 \*/* MyThreadPool<Task> myThreadPool =  
 **new** MyThreadPool<>(1, 1000, TimeUnit.***MILLISECONDS***, 1,  
 ((queue, task) -> queue.put(task)));  
  
 */\*\*  
 \* 策略2：带超时等待  
 \* MyThreadPool<Task> myThreadPool =  
 \* new MyThreadPool<>(1, 1000, TimeUnit.MILLISECONDS, 1,  
 \* ((queue, task) -> queue.offer(task, 1000, TimeUnit.MILLISECONDS)));  
 \*/  
  
 /\*\*  
 \* 策略3：放弃任务执行  
 \*  
 \* MyThreadPool<Task> myThreadPool =  
 \* new MyThreadPool<>(1, 1000, TimeUnit.MILLISECONDS, 1,  
 \* ((queue, task) -> log.debug("放弃执行任务")));  
 \*/  
  
  
 /\*\*  
 \* 策略4：抛出异常，终止执行  
 \*MyThreadPool<Task> myThreadPool =  
 \* new MyThreadPool<>(1, 1000, TimeUnit.MILLISECONDS, 1,  
 \* ((queue, task) -> {  
 \* throw new RuntimeException("任务执行失败");  
 \* }));  
 \*/  
  
  
 /\*\*  
 \* 策略5：调用者自己执行  
 \* MyThreadPool<Task> myThreadPool =  
 \* new MyThreadPool<>(1, 1000, TimeUnit.MILLISECONDS, 1,  
 \* ((queue, task) -> {  
 \* task.run();  
 \* }));  
 \*/* **for** (**int** i = 0; i < 3; i++) { *//3个任务* myThreadPool.execute(**new** Task(**"Task--->"** + i));  
 }  
 **try** {  
 Thread.*sleep*(50000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}  
  
*/\*\*  
 \* 拒绝策略接口  
 \*/*@FunctionalInterface  
**interface** rejectPolicy<T> {  
 **void** reject(BlockQueue<T> queue, T task);  
}  
  
*/\*\*  
 \* 线程池实现  
 \*/*@Slf4j(topic = **"c.MyThreadPool"**)  
**class** MyThreadPool<T> {  
 *//阻塞队列* **private** BlockQueue<Task> **taskQueue**;  
 *//线程集合* **private** HashSet<Worker> **workers** = **new** HashSet<>();  
 *//核心线程数* **private int coreSize**;  
 *//获取任务的超时时间* **private long timeout**;  
 *//时间单位* **private** TimeUnit **unit**;  
 *//线程池的拒绝策略* **private** rejectPolicy **rejectPolicy**;  
  
  
 **public** MyThreadPool(**int** coreSize, **long** timeout, TimeUnit unit, **int** capacity, rejectPolicy rejectPolicy) {  
 **this**.**coreSize** = coreSize;  
 **this**.**timeout** = timeout;  
 **this**.**unit** = unit;  
 **taskQueue** = **new** BlockQueue<>(capacity);  
 *//初始化拒绝策略* **this**.**rejectPolicy** = rejectPolicy;  
 }  
  
 *//执行任务* **public void** execute(Task task) {  
 **synchronized** (**workers**) {  
 **if** (**workers**.size() < **coreSize**) { *//如果任务数小于核心数，则直接执行* Worker worker = **new** Worker(task);  
 ***log***.debug(**"新增工作线程 {},将要执行 {}"**, worker, task);  
 **workers**.add(worker);  
 worker.start();  
 } **else** {  
 *//策略模式，具体操作由调用者实现  
 //（1）死等  
 //（2）带超时等待  
 //（3）放弃任务执行  
 //（4）抛出异常  
 //（5）让调用者自己执行任务* **taskQueue**.tryPut(**rejectPolicy**, task);  
 }  
 }  
 }  
  
 *//Work线程对象* **class** Worker **extends** Thread {  
 **private** Task **task**;  
  
 **public** Worker(Task task) {  
 **this**.**task** = task;  
 }  
  
 @Override  
 **public void** run() {  
 **while** (**task** != **null** || (**task** = **taskQueue**.poll(1000, **unit**)) != **null**) {  
 **try** {  
 ***log***.debug(**"正在执行任务 {}"**, **task**);  
 Thread.*sleep*(5000); *//故意设置长等待时间* **task**.run();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **task** = **null**;  
 }  
 }  
 **synchronized** (**workers**) {  
 ***log***.debug(**"移除工作线程 {}"**, **this**);  
 **workers**.remove(**this**);  
 }  
 }  
 }  
}  
  
*/\*\*  
 \* 任务类  
 \*/*@Slf4j(topic = **"c.Task"**)  
**class** Task **implements** Runnable {  
 **private** String **name**;  
  
 **public** Task(String name) {  
 **this**.**name** = name;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Task{"** +  
 **"name='"** + **name** + **'\''** +  
 **'}'**;  
 }  
  
 @Override  
 **public void** run() {  
 ***log***.debug(**"{}"**, **name**);  
 }  
}  
  
*/\*  
 阻塞队列实现  
 \*/*@Slf4j(topic = **"c.BlockQueue"**)  
**class** BlockQueue<T> {  
 *//1.队列对象* **private** Deque<T> **queue** = **new** ArrayDeque<>();  
 *//2.阻塞队列容量* **private int capacity**;  
 *//3.锁* **private** ReentrantLock **lock** = **new** ReentrantLock();  
 *//4.阻塞队列为空的条件变量* **private** Condition **emptyWaitSet** = **lock**.newCondition();  
 *//5.阻塞队列为满的条件变量* **private** Condition **fullWaitSet** = **lock**.newCondition();  
  
 **public** BlockQueue(**int** capacity) {  
 **this**.**capacity** = capacity;  
 }  
  
 *//阻塞获取* **public** T take() {  
 **try** {  
 **lock**.lock();  
 *//如果阻塞队列为空，则等待* **while** (**queue**.size() == 0) {  
 **try** {  
 **emptyWaitSet**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 *//阻塞队列不为空，取出取出一个对象后唤醒生产者线程* T t = **queue**.removeFirst();  
 **fullWaitSet**.signal();  
 **return** t;  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *//带超时阻塞获取* **public** T poll(**long** timeout, TimeUnit unit) {  
 **try** {  
 **lock**.lock();  
 *//纳秒* **long** nano = unit.toNanos(timeout);  
 **while** (**queue**.size() == 0) {  
 **try** {  
 **if** (nano <= 0) {  
 **return null**;  
 }  
 *//awaitNanos方法返回超时时间-经历时间，将返回值再次赋值给nano，可解决虚假唤醒问题* nano = **emptyWaitSet**.awaitNanos(nano);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 T t = **queue**.removeFirst();  
 **fullWaitSet**.signal();  
 **return** t;  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *//阻塞添加* **public void** put(T task) {  
 **try** {  
 **lock**.lock();  
 *//如果阻塞队列满，则等待* **while** (**queue**.size() == **capacity**) {  
 **try** {  
 ***log***.debug(**"阻塞队列已满，等待加入..."**);  
 **fullWaitSet**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 *//阻塞队列不满，添加后唤醒消费者线程* **queue**.addLast(task);  
 **emptyWaitSet**.signal();  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *//带超时阻塞添加* **public boolean** offer(T task, **long** timeout, TimeUnit unit) {  
 **try** {  
 **lock**.lock();  
 **long** nano = unit.toNanos(timeout);  
 **while** (**queue**.size() == **capacity**) {  
 **try** {  
 **if** (nano <= 0) {  
 ***log***.debug(**"{}添加阻塞队列失败"**, task);  
 **return false**;  
 }  
 ***log***.debug(**"等待加入任务队列 {}"**, task);  
 nano = **fullWaitSet**.awaitNanos(nano);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 ***log***.debug(**"{} 加入阻塞队列"**, task);  
 **queue**.addLast(task);  
 **emptyWaitSet**.signal();  
 **return true**;  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *//阻塞队列大小* **public int** size() {  
 **try** {  
 **lock**.lock();  
 **return queue**.size();  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 **public void** tryPut(rejectPolicy<T> rejectPolicy, T task) {  
 **lock**.lock();  
 **try** {  
 *//判断队列是否满* **if** (**queue**.size() == **capacity**) {  
 rejectPolicy.reject(**this**, task); *//队列满时的策略* } **else** { *//有空闲将任务加入阻塞队列* ***log***.debug(**"{} 加入阻塞队列"**, task);  
 **queue**.addLast(task);  
 **emptyWaitSet**.signal();  
 }  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
}

1. ThreadPoolExecutor