优质博文

更好的使用 JAVA 线程池

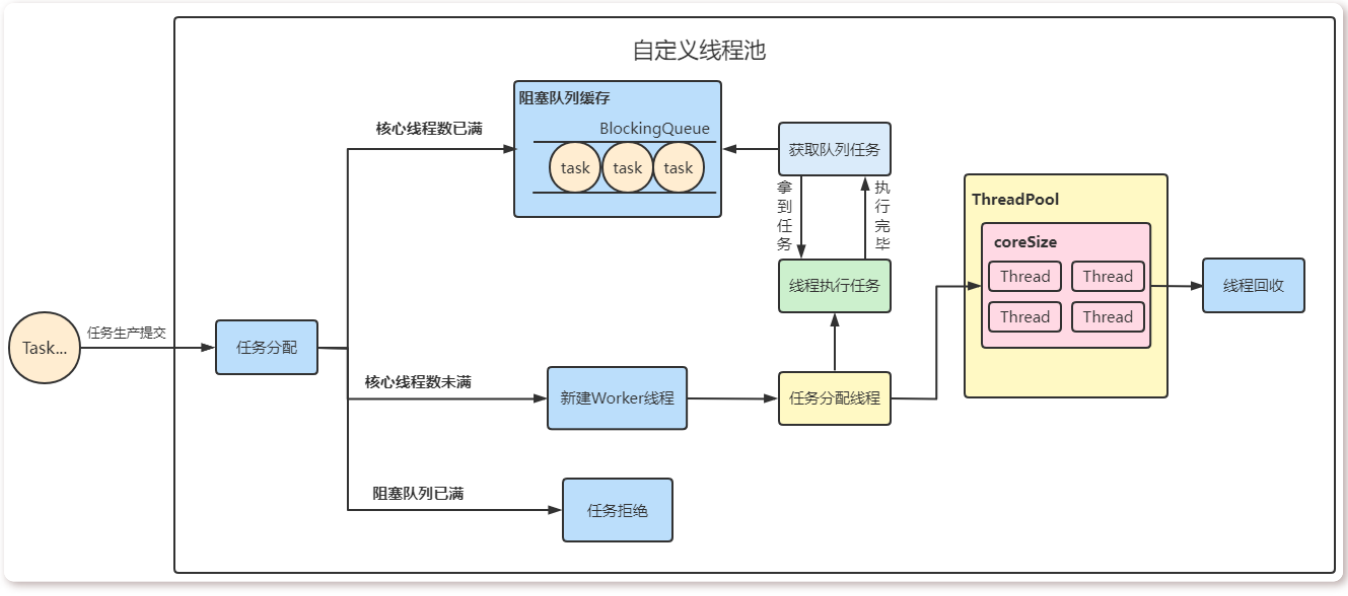
深入理解Java线程池：ThreadPoolExecutor

Java线程池实现原理及其在美团业务中的实践

1、概念图

核心部分：

- 阻塞队列 *BlockingQueue*：暂存线程池中无法处理的任务
- 线程池 *ThreadPool*：自定义的线程池，内部最多包含 *coreSize* 个工作线程执行任务
- 工作线程 *WorkerThread*：执行传递过来的任务
- 拒绝策略 *Rejectpolicy*：当阻塞队列已满时采用指定的策略拒绝任务



2、流程分析

根据上面的概念图，进一步模拟一遍整个线程池执行的流程：

1. **初始化线程池**，指定线程池的参数如**核心线程数、阻塞队列容量、超时时间、拒绝策略**；
2. 并发生产**任务压入线程池**执行；
    1. 工作线程数**未达到**设定的核心线程数。**新建工作线程**执行任务，并将工作线程加入到线程池中的**线程集合**中；
    2. 工作线程数**达到了**设定的核心线程数。**尝试往阻塞队列中暂存任务**，当阻塞队列**已满**无法添加时，采用指定的**拒绝策略**对任务进行拒绝。
3. 工作线程执行完当前任务时，**循环从阻塞队列中获取任务**并执行直到消费完阻塞队列中的任务；
4. 当无任务时，将工作**线程回收**销毁。

3、设计思路及实现

整体的设计思路应该由广到细，整体到局部。前面的概念图以及流程分析其实就算是一个整体的设计了，接下来便是局部的设计了。首先先列举一下需要的部分，分别为：

- 线程池
- 工作线程
- 阻塞队列
- 拒绝策略

结合上面一二点的描述我们可以得出线程池中用到了工作线程和阻塞队列，而当阻塞队列满时需要根据拒绝策略进行任务拒绝，因此我们采取自下而上的方式逐一设计需要的几大主体。

3.1、拒绝策略

其实拒绝策略就是一段逻辑，通过调用者告知使用哪种方式进行任务拒绝。根据 *OOP思想* ，这一段逻辑我们可以封装成不同的方法，通过传入不同的标识选用不同的方法即可。这里使用了 *Java1.8* 出现的函数式编程进行设计，将这一个逻辑封装成一个函数式接口，调用者可直接使用Lambda表达式指定需要的拒绝策略，也可将逻辑封装成一个枚举类，直接传入对应的方法即可，这符合设计模式中的开闭原则，可维护性更高。

```java
@FunctionalInterface
public interface RejectPolicy<T> {
    /**
     * @description 拒绝策略中的拒绝方法，可自定义设置适合的拒绝策略
     * @author xbaozi
     * @date 2022/11/18 22:34
     * @param queue 阻塞队列
     * @param task  需要拒绝的任务
     **/
    void reject(BlockingQueue<T> queue, T task);
}
```

3.2、阻塞队列

在该阻塞队列中采取了`公平的FIFO形式`，避免任务一直得不到消费出现**饿死情况**，因此内部需要维护一个**双向队列**。出于内存层面考虑，我们需要维护一个队列**最大容量**变量，用于判断队列是否已满，避免`OOM问题`出现。同时由于阻塞队列为多线程下的共享资源，我们需要对其上锁保证在并发消费下的原子性。最后为了阻塞队列的拓展性，队列中存放的内容采取泛型设计。

- **双向队列Deque**：Java内置的双向队列接口，实现类采用ArrayDeque，在大部分情况下会比LinkList性能要好一点；
- **最大容量capacity**：基本整形变量，用于判断队列是否已满；
- **锁对象LOCK**：采用可重入锁ReentrantLock实现，并分别设置消费者条件变量与生产者条件变量，对队列为空与队列已满两种情况进行隔离。

在变量设计完成之后，我们还需要对队列中的**方法**进行设计。显而易见的是队列的**核心任务为存和取**，重点的是怎么存和取。比较容易想到的是**超时与无超时限制**的存取，但是这样子的话并没有用到我们的拒绝策略，因此应该还有一个方法是**尝试将任务存入队列**中，当队列满时采用指定的拒绝策略即可。

- `void put(T task)`：添加任务，这是一个**阻塞添加无超时**的方法，即这个方式在队列已满时会一直等待直到队列中出现闲余空间；
- `boolean put(T task, long timeout, TimeUnit timeUnit)`：带超时限制的添加任务，当等待了指定时间后队列仍然无空间时，则会放弃当前任务退出等待；
- `void tryPut(T task, RejectPolicy<T> rejectPolicy)`：**尝试添加任务**，如果任务队列满了会根据传入的**拒绝策略**对任务进行处理；
- `T take()`：获取任务，这是一个**阻塞获取无超时**的方法，即队列为空时会一直等待直到队列中出现任务；
- `T take(long timeout, TimeUnit unit)`：带**超时时间限制**返回获取到的任务，当等待了指定时间后队列仍然为空时，则会放弃获取退出等待。

```java
/**
 * @author xbaozi
 * @version 1.0
```

```
 * @classname BlockingQueue
 * @date 2022-11-17  16:35
 * @description 阻塞队列，使用泛型增加拓展性
 */
@Slf4j(topic = "xbaoziplus.BlockingQueue")
public class BlockingQueue<T> {
    // 任务队列，使用双向链表尾进头出
    private final Deque<T> TASK_QUEUE = new ArrayDeque<>();

    // 队列最大容量
    private int capacity;

    // 锁对象
    private final ReentrantLock LOCK = new ReentrantLock();

    // 生产者条件变量，当队列中满了的话生产者线程需要进入该condition进行等待
    private final Condition PRODUCER_WAIT_CONDITION = LOCK.newCondition();

    // 消费者条件变量，当队列中为空时消费者线程需要进入该condition进行等待
    private final Condition CONSUMER_WAIT_CONDITION = LOCK.newCondition();

    // 有参构造器，初始化队列最大容量
    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    /**
     * @param task 生产者产生的任务
     * @description 添加任务，这是一个阻塞添加的方法
     * @author xbaozi
     * @date 2022/11/17 16:57
     **/
    public void put(T task) {
        // 因为获取大小和添加任务不是原子操作，因此需要上锁保证原子性
        LOCK.lock();
        try {
            // 自旋判断队列是否已满，避免虚假唤醒
            while (TASK_QUEUE.size() >= capacity) {
                try {
                    log.error("队列已满，生产者等待将任务加入任务队列中……");
                    // 进入生产者条件变量中等待
                    PRODUCER_WAIT_CONDITION.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
            }
            // 任务队列出现闲余空间时，将任务采用尾插法添加至任务队列中
            log.info("任务队列存在闲余空间，{}加入队列", task);
            TASK_QUEUE.addLast(task);
            // 唤醒消费者条件变量中线程，提示队列不为空，可以进行任务消费移除
            CONSUMER_WAIT_CONDITION.signal();
        } finally {
            LOCK.unlock();
        }
    }

    /**
     * @param task     生产者产生的任务
     * @param timeout  超时时间
     * @param timeUnit 时间单位
     * @description 带超时限制的添加任务
     * @author xbaozi
     * @date 2022/11/18 21:38
     **/
    public boolean put(T task, long timeout, TimeUnit timeUnit) {
        LOCK.lock();
        try {
            // 将超时时间转换成纳秒
            long nanos = timeUnit.toNanos(timeout);
            while (TASK_QUEUE.size() >= capacity) {
                try {
                    // 判断是否超时
                    if (nanos <= 0) {
                        // 超时返回失败标识
                        log.error("添加任务{}超时", task);
                        return false;
                    }
                    // 进入生产者条件变量中等待nanos秒或等待被唤醒
                    nanos = PRODUCER_WAIT_CONDITION.awaitNanos(nanos);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            log.info("不超时，{}加入任务队列成功", task);
            // 尾插法插入任务
            TASK_QUEUE.addLast(task);
            // 唤醒消费者条件变量中线程，提示队列不为空，可以进行任务消费移除
            CONSUMER_WAIT_CONDITION.signal();
            return true;
        } finally {
```

```
94              LOCK.unlock();
95          }
96      }
97
98      /**
99       * @description 添加任务，如果任务队列满了会根据传入的拒绝策略对任务进行处理
100      * @author xbaozi
101      * @date 2022/11/18 23:15
102      * @param task  任务
103      * @param rejectPolicy  拒绝策略
104      **/
105     public void tryPut(T task, RejectPolicy<T> rejectPolicy) {
106         LOCK.lock();
107         try {
108             // 判断任务队列是否已满
109             if (TASK_QUEUE.size() >= capacity) {
110                 // 任务队列已满，采取设定的拒绝策略进行处理
111                 rejectPolicy.reject(this, task);
112             } else {
113                 // 任务队列未满，添加至任务队列中
114                 log.info("{}加入任务队列成功", task);
115                 TASK_QUEUE.addLast(task);
116                 // 唤醒消费者条件变量中线程，提示队列不为空，可以进行任务消费移除
117                 CONSUMER_WAIT_CONDITION.signal();
118             }
119         } finally {
120             LOCK.unlock();
121         }
122     }
123
124     /**
125      * @return T    返回获取到的任务
126      * @description 获取任务，这是一个阻塞获取的方法
127      * @author xbaozi
128      * @date 2022/11/17 17:05
129      **/
130     public T take() {
131         // 因为判断队列是否为空和弹出任务不是原子操作，因此需要上锁保证原子性
132         LOCK.lock();
133         try {
134             // 自旋判断队列是否为空，避免虚假唤醒
135             while (TASK_QUEUE.isEmpty()) {
136                 try {
137                     log.error("队列为空，消费者等待任务到来进行消费……");
138                     // 进入消费者条件变量中等待
```

```
                    CONSUMER_WAIT_CONDITION.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            // 任务队列中产生了新任务时，从队列头部获取
            T task = TASK_QUEUE.removeFirst();
            log.info("队列中有任务{}取出消费", task);
            // 唤醒生产者条件变量中线程，提示队列已出现闲余空间，可以进行任务生产添加
            PRODUCER_WAIT_CONDITION.signal();
            // 返回任务
            return task;
        } finally {
            LOCK.unlock();
        }
    }

    /**
     * @description 带超时时间限制返回获取到的任务
     * @author xbaozi
     * @date 2022/11/18 22:40
     * @param timeout    超时时间
     * @param unit       超时时间单位
     **/
    public T take(long timeout, TimeUnit unit) {
        LOCK.lock();
        try {
            long nanos = unit.toNanos(timeout);
            // 判断任务队列中是否有任务可拿
            while (TASK_QUEUE.isEmpty()) {
                if (nanos <= 0) {
                    return null;
                }
                try {
                    log.error("队列为空，消费者等待任务到来进行消费……");
                    // 进入消费者条件变量中等待
                    nanos = CONSUMER_WAIT_CONDITION.awaitNanos(nanos);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            // 任务队列中有任务可拿时，从队列头部获取任务
            T task = TASK_QUEUE.removeFirst();
            log.info("队列中有任务{}取出消费", task);
            // 唤醒生产者条件变量中线程，提示队列已出现闲余空间，可以进行任务生产添加
```

```
                PRODUCER_WAIT_CONDITION.signal();
                // 返回任务
                return task;

            } finally {
                LOCK.unlock();
            }
        }

        /**
         * @description 获取阻塞队列中的任务数
         * @author xbaozi
         * @date 2022/11/18 22:19
         **/
        public int size() {
            LOCK.lock();
            try {
                return TASK_QUEUE.size();
            } finally {
                LOCK.unlock();
            }
        }
    }
```

3.3、工作线程

你可能在想着为什么还要自定义一个工作线程，直接用Thread不行吗？其实还真不行.

因为在工作线程中我们需要对task任务进行消费，而run方法并不支持传参，因此我们需要自定义一个**WorkerThread继承Thread**，并拓展一个成员变量task，通过构造器传参实现对任务的消费。

另外需要注意的是这个工作线程在设计的时候将其设定为线程池的内部类，因此代码在线程池中再一起贴出来

3.4、线程池

我们先根据需求来判断我们需要哪些成员变量。

首先我们需要核心线程来工作执行消费任务，因此需要一个**核心线程数**变量，并且需要一个**容纳线程的集合**存放工作线程；

其次我们在工作线程达到核心线程数时，需要将任务暂时存入阻塞队列中，因此需要一个**阻塞队列**的变量，值得注意的是在构造器初始化时应该传入队列容量在构造器中进行实例化，而不是传入一个阻塞队列对象，提供使用而不暴露实现；

紧接着的就是**超时时间**了，也可以将其忽略在执行方法时将其当做参数进行方法传参，这里放在成员变量中便于统一管理；

最后便是**拒绝策略**，在初始化线程池时就应该指定线程池的拒绝策略，在阻塞队列满时对任务进行拒绝。

```
@Slf4j(topic = "xbaoziplus.MyThreadPool")
public class ThreadPool {
    // 任务阻塞队列
    private final BlockingQueue<Runnable> BLOCKING_QUEUE;

    // 线程集合
    private final Set<WorkerThread> workers = new HashSet<>();

    // 核心线程数
    private int coreSize;

    // 获取任务的超时时间
    private long timeout;

    // 获取任务超时时间的时间单位
    private TimeUnit unit;

    // 拒绝策略
    RejectPolicy<Runnable> rejectPolicy;

    public ThreadPool(int queueCapacity, int coreSize, int timeout,
TimeUnit unit, RejectPolicy<Runnable> rejectPolicy) {
        this.BLOCKING_QUEUE = new BlockingQueue<>(queueCapacity);
        this.coreSize = coreSize;
        this.timeout = timeout;
        this.unit = unit;
        this.rejectPolicy = rejectPolicy;
    }

    /**
     * @param task 需要执行的任务
     * @description 线程池接收任务执行
     * @author xbaozi
     * @date 2022/11/17 17:52
     **/
    public void execute(Runnable task) {
        synchronized (workers) {
            // 判断工作线程数是否达到了核心线程数
            if (workers.size() < coreSize) {
```

```
                // 工作线程数未达到核心线程数，新建一个工作线程
                WorkerThread workerThread = new WorkerThread(task, workers.size() + "号工作线程");
                log.info("未达到核心线程数，新建工作线程{}", workerThread.getName());
                // 将工作线程添加到线程集合中
                workers.add(workerThread);
                // 启动线程执行任务
                workerThread.start();
            } else {
                // 工作线程已达到核心线程数，将任务放入任务队列中暂存
                // log.info("工作线程已达到核心线程数，{}进入任务队列暂存", task);
                // 无超时阻塞添加任务
                // BLOCKING_QUEUE.put(task);
                // 设置超时时间添加任务
                // BLOCKING_QUEUE.put(task, timeout, unit);
                // 尝试添加任务，任务添加失败时选择自定义的拒绝策略
                log.info("工作线程已达到核心线程数，尝试添加任务{}到任务队列中暂存", task);
                BLOCKING_QUEUE.tryPut(task, rejectPolicy);
            }
        }
    }

    /**
     * @author xbaozi
     * @description 线程池中工作线程
     * @date 2022/11/17 17:26
     **/
    //@Slf4j(topic = "xbaoziplus.MyThreadPool.WorkerThread")
    class WorkerThread extends Thread {
        // 需要执行的任务
        private Runnable task;

        public WorkerThread(Runnable task, String name) {
            super(name);
            this.task = task;
        }

        @Override
        public void run() {
            // 自旋判断当前任务是否为空，不为空执行任务，为空时获取下一个任务接着执行
            // while (task != null || (task = BLOCKING_QUEUE.take()) != null) {  // 无超时限制的等待获取任务
            // 有超时限制的等待获取任务
            while (task != null || (task = BLOCKING_QUEUE.take(timeout, unit)) != null) {
```

```
                try {
                    // 执行任务
                    log.info("{}正在执行……", task);
                    task.run();
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    // 不能在这里赋值task = BLOCKING_QUEUE.take()，否则容易产生线程饥饿
                    task = null;
                }
            }
            // 执行完毕时，将当前工作线程移除，实现线程销毁效果
            synchronized (workers) {
                log.info("任务执行完毕，线程{}已销毁", this.getName());
                workers.remove(this);
            }
        }
    }
}
```

4、测试线程池

这里并没有封装一部分的拒绝策略给调用者进行选择，而是完全由调用者编写拒绝策略。这里一共列举了五种拒绝策略，分别为：

- 死等，直到阻塞队列出现空间或工作线程空余；
- 超时等待，等待一定时间后自行结束；
- 直接放弃，当阻塞队列已满时直接对后续的任务进行放弃；
- 抛异常，当阻塞队列已满时抛出异常提示调用者；
- 自行执行任务，让调用者线程自行执行多出来的任务。

```
@Slf4j(topic = "test.TestPool")
public class TestPool {
    public static void main(String[] args) {
        // 新建线程池
        ThreadPool pool = new ThreadPool(2, 2, 2, TimeUnit.SECONDS, (queue, task) -> {
            // 1. 死等
            //queue.put(task);
            // 2) 超时等待
            //queue.put(task, 1500, TimeUnit.MILLISECONDS);
            // 3) 直接放弃
            log.debug("任务队列已满，放弃任务{}", task);
            // 4) 抛出异常
            //try {
```

```
14                //    throw new RuntimeException("任务执行失败 " +
    task);
15                //} catch (RuntimeException e) {
16                //    log.debug("任务队列已满，{}", e.getMessage());
17                //}
18                // 5) 自行执行任务
19                // task.run();
20            });
21            // 模拟五个线程生产任务压入线程池中执行
22            for (int i = 0; i < 5; i++) {
23                int index = i;
24                pool.execute(() -> {
25                    try {
26                        // 模拟任务执行需要1s
27                        Thread.sleep(1000L);
28                    } catch (InterruptedException e) {
29                        e.printStackTrace();
30                    }
31                    log.info("任务{}执行完毕", index);
32                });
33            }
34        }
35    }
```

```
"D:\Program Files\Java\jdk1.8.0_341\bin\java.exe" ...
15:45:01.595 [main] INFO xbaoziplus.MyThreadPool - 未达到核心线程数，新建工作线程0号工作线程
15:45:01.598 [main] INFO xbaoziplus.MyThreadPool - 未达到核心线程数，新建工作线程1号工作线程
15:45:01.598 [main] INFO xbaoziplus.MyThreadPool - 工作线程已达到核心线程数，尝试添加任务top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4cdf35a9到任务队列中暂存
15:45:01.598 [0号工作线程] INFO xbaoziplus.MyThreadPool - top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4ac295e5正在执行……
15:45:01.598 [main] INFO xbaoziplus.BlockingQueue - top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4cdf35a9加入任务队列成功
15:45:01.598 [main] INFO xbaoziplus.MyThreadPool - 工作线程已达到核心线程数，尝试添加任务top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4c98385c到任务队列中暂存
15:45:01.598 [1号工作线程] INFO xbaoziplus.MyThreadPool - top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@16104d06正在执行……
15:45:01.598 [main] INFO xbaoziplus.BlockingQueue - top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4c98385c加入任务队列成功
15:45:01.599 [main] INFO xbaoziplus.MyThreadPool - 工作线程已达到核心线程数，尝试添加任务top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@5fcfe4b2到任务队列中暂存
15:45:01.599 [main] DEBUG test.TestPool - 任务队列已满，放弃任务top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@5fcfe4b2
15:45:02.606 [0号工作线程] INFO test.TestPool - 任务0执行完毕
15:45:02.606 [1号工作线程] INFO test.TestPool - 任务1执行完毕
15:45:02.606 [0号工作线程] INFO xbaoziplus.BlockingQueue - 队列中有任务top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4cdf35a9取出消费
15:45:02.607 [1号工作线程] INFO xbaoziplus.BlockingQueue - 队列中有任务top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4c98385c取出消费
15:45:02.607 [0号工作线程] INFO xbaoziplus.MyThreadPool - top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4cdf35a9正在执行……
15:45:02.607 [1号工作线程] INFO xbaoziplus.MyThreadPool - top.xbaoziplus.juc.pool.TestPool$$Lambda$2/254413710@4c98385c正在执行……
15:45:03.619 [1号工作线程] INFO test.TestPool - 任务3执行完毕
15:45:03.619 [0号工作线程] INFO test.TestPool - 任务2执行完毕
15:45:03.619 [1号工作线程] ERROR xbaoziplus.BlockingQueue - 队列为空，消费者等待任务到来进行消费……
15:45:03.619 [0号工作线程] ERROR xbaoziplus.BlockingQueue - 队列为空，消费者等待任务到来进行消费……
15:45:05.630 [1号工作线程] INFO xbaoziplus.MyThreadPool - 任务执行完毕，线程1号工作线程已销毁
15:45:05.630 [0号工作线程] INFO xbaoziplus.MyThreadPool - 任务执行完毕，线程0号工作线程已销毁
```