Java ???

- Java □□□□

Java ????

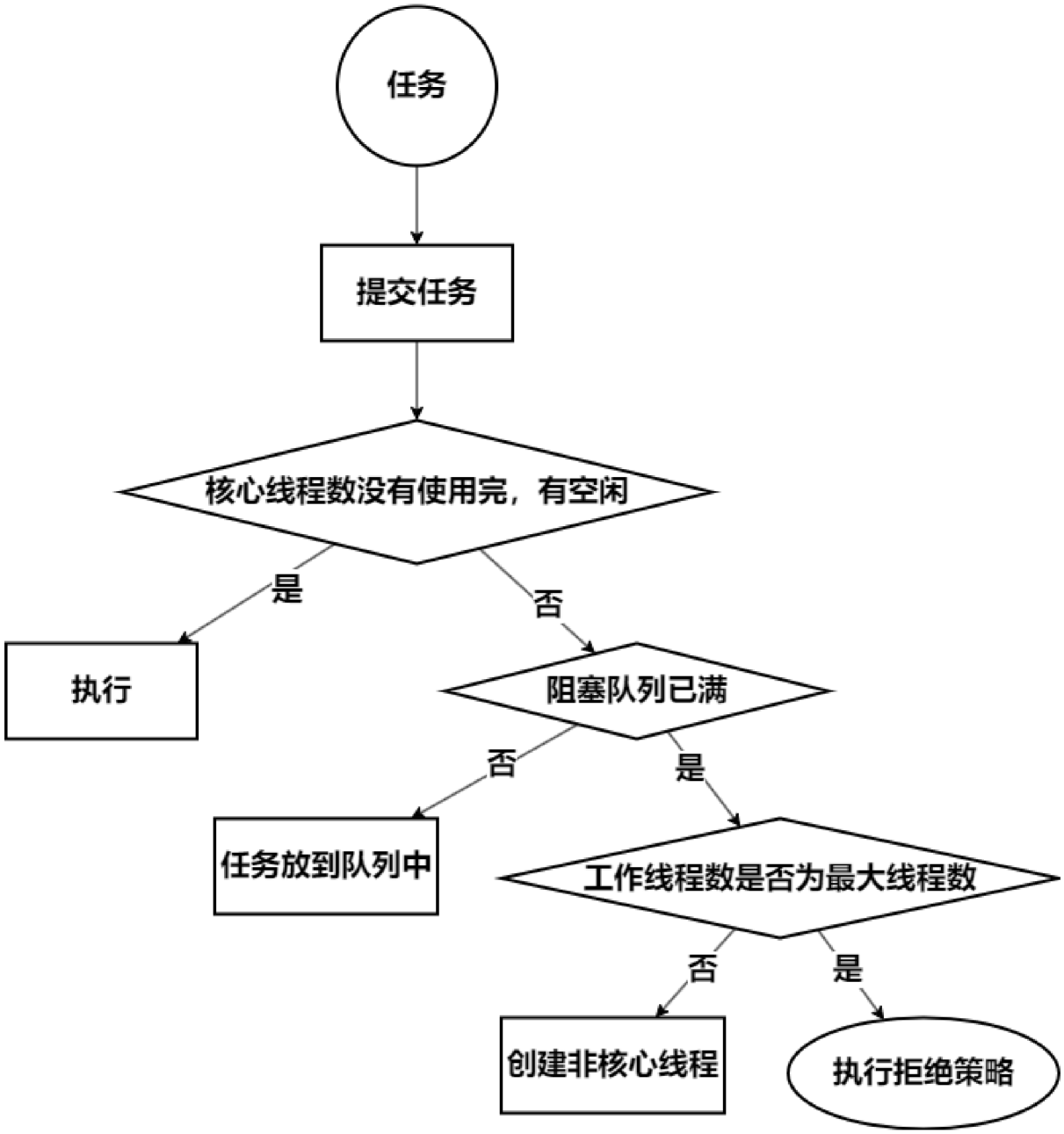
- ????????

1. 继承 Thread
2. 实现 Runable 接口 无返回值 无参数
3. 实现 Callable 接口 有返回值 无参数
4. 线程池

- ????????

1. corePoolSize: 核心线程池的大小
2. maximumPoolSize: 最大线程池的大小
3. keepAliveTime: 当线程池中线程数大于 corePoolSize, 并且没有可执行任务时大于 corePoolSize那部分线程的存活时间
4. unit: keepAliveTime的时间单位
5. workQueue: 用来暂时保存任务的工作队列
6. threadFactory: 线程工厂提供线程的创建方式，默认使用 Executors.defaultThreadFactory()
7. handler
   : 当线程池所处理的任务数超过其承载容量或关闭后继续有任务提交时，所调用的拒绝策略

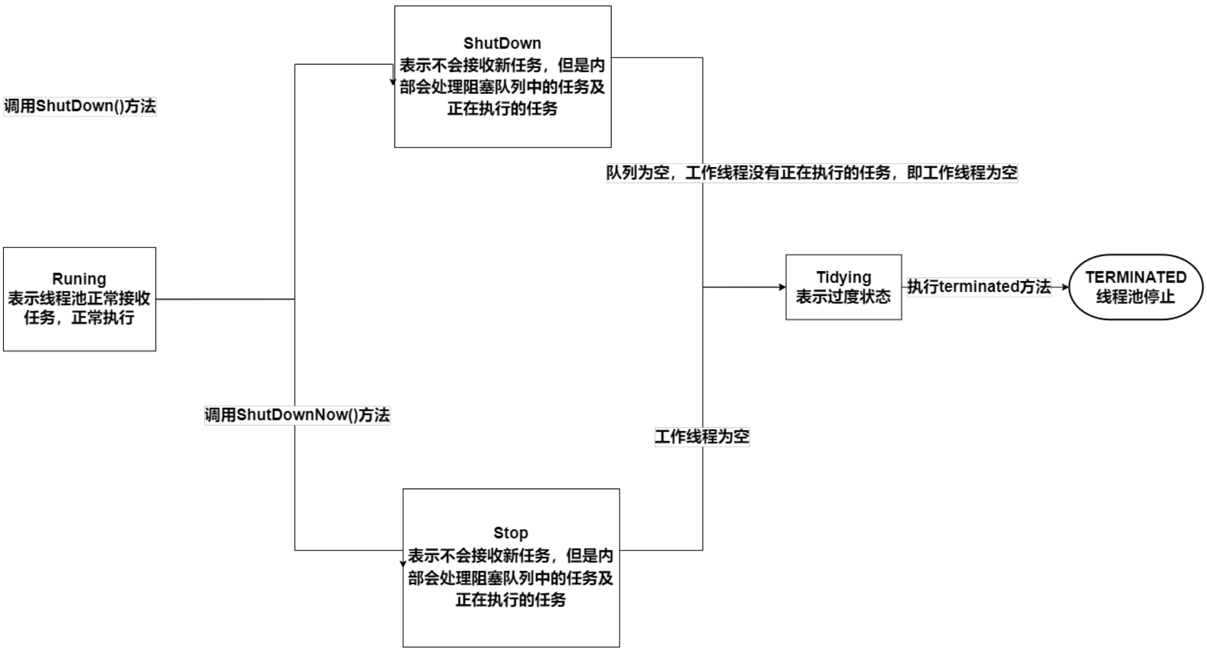
- ??????????



- ??????????????????????????
  - 最大化资源利用
  - 举个例子
    - 饭店（线程池） ----> 厨子（线程） -------> 人多先排队（阻塞队列）【天天客满，忙不过来】 -----> 招厨子 ---> 今日客满（拒绝策略）
- ???????
  -

```
// □□□□□□□□□          □□□□□          □□□□□□          29□□□□□□□□□□
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
// □□   29 □□□□
private static final int COUNT_BITS = Integer.SIZE - 3;

// □□□□□□□
private static final int CAPACITY = (1 << COUNT_BITS) - 1;

// □□□□□
// RUNNING  111 □□□□□□□□□          □□□□□□□
private static final int RUNNING = -1 << COUNT_BITS;

// 000 □□□□□□          SHUTDOWN □□   □□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
private static final int SHUTDOWN = 0 << COUNT_BITS;

// 001 □□□□□□          stop □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

private static final int STOP = 1 << COUNT_BITS;

// 010 □□□□□□          TIDYING □□□□□□□□□□□□□□          terminated()□□  , □□□□□□□□□
private static final int TIDYING = 2 << COUNT_BITS;

// 011 □□   TERMINATED □□□□□□□□□□□□
private static final int TERMINATED = 3 << COUNT_BITS;

// □□□□□□□□
private static int runStateOf(int c) { return c & ~CAPACITY; }

// □□□□□□□□□□□□□□
private static int workerCountOf(int c) { return c & CAPACITY; }
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

- ??????



- Execute ????

```
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        /*
          按3个步骤进行:


        *
       1。如果运行的线程少于corePoolSize，请尝试
        *以给定的命令作为第一个线程启动一个新线程
        *任务。调用addWorker会自动检查runState和
        * workerCount，这样可以防止误报警
        *在不应该执行线程时返回false。
        *
        * 2。如果一个任务可以成功排队，那么我们仍然需要
        *检查我们是否应该添加一个线程
        *(因为自上次检查以来现有的已死亡)或其他
        进入此方法后池关闭。所以我们
        *重新检查状态，必要时回滚队列
        * stopped，或者在没有线程时启动一个新线程。
        *
        * 3。如果我们不能排队任务，那么我们尝试添加一个新的
        *线程。如果它失败了，我们知道我们被关闭或饱和了
        *，因此拒绝该任务。
         */
        // 线程池信息 有两个作用 高三位代表 线程池状态，低29位代表线程池中的线程数量
        int c = ctl.get();

        // 判断线程池中的工作线程数量是否小于核心线程数
        if (workerCountOf(c) < corePoolSize) {
            // 创建核心线程 执行任务
            if (addWorker(command, true))
                return;
            // 创建核心线程数失败，说明核心线程执行的任务已满 再次获取线程池信息
            c = ctl.get();
        }

        // 判断线程池是否是Running状态且可以把任务放到队列中
```

```
        if (isRunning(c) && workQueue.offer(command)) {

            // 再次获取线程池信息
            int recheck = ctl.get();
            // 判断程池是否是非Running状态，是的话移除任务
            if (! isRunning(recheck) && remove(command))
                // 执行拒绝策略
                reject(command);
            // 如果线程池状态为Running状态，但是线程池中工作线程数量为0
            else if (workerCountOf(recheck) == 0)
                // 阻塞队列中有任务，但是没有工作线程处理，则创建一个任务为null的工作线程去处理阻塞队列中的任务
                addWorker(null, false);
        }
       // 阻塞队列已满，创建非核心线程处理任务
        else if (!addWorker(command, false))
          // 失败则执行拒绝策略
            reject(command);
    }
```

- addWorker ????

```
private boolean addWorker(Runnable firstTask, boolean core) {
        retry:
        // 通过大量的循环和判断，目的是让工作线程数加1
        for (;;) {
            // 线程池信息 有两个作用 高三位代表 线程池状态，低29位代表线程池中的线程数量
            int c = ctl.get();

            // 获取线程池状态
            int rs = runStateOf(c);

            //
            if (
              // 判断 线程池状态 (除了 Running 都有可能)
              // runState is stored in the high-order bits
              // RUNNING    = -1 << COUNT_BITS;
              // SHUTDOWN   =  0 << COUNT_BITS;
              // STOP       =  1 << COUNT_BITS;
              // TIDYING    =  2 << COUNT_BITS;
```

```
          // TERMINATED =  3 << COUNT_BITS;
          rs >= SHUTDOWN
            &&
            ! (
              // 线程池状态为Shutdown，如果状态不是Shutdown 则可能是 STOP等更高的状态，则现在不需要添加线程处理任务
              rs == SHUTDOWN
               &&
              // 任务为null, 如果任务为空 线程池状态又不为Running 则不需要添加线程处理任务
               firstTask == null
               &&
              // 阻塞队列不为空，取反后为空，代表阻塞队列为空则不需要处理
               ! workQueue.isEmpty()
              ))
           // 创建线程失败
            return false;

        for (;;) {
            // 获取工作线程数量
            int wc = workerCountOf(c);

            if (
              // 如果工作线程数量大于线程池最大线程容量，就不创建线程了
              wc >= CAPACITY ||
              // 或者 工作线程数量大于核心线程数量或最大线程数量，就不创建线程了
                wc >= (core ? corePoolSize : maximumPoolSize))
                // 创建线程失败
                return false;
            // 工作线程数量 +1，利用CAS锁
            if (compareAndIncrementWorkerCount(c))
                // 跳出外层循环
                break retry;
            // 再次 线程池信息
            c = ctl.get();
            //  判断线程池状态是否有变化，没有变化，则执行内层循环
            if (runStateOf(c) != rs)
                // 有变化，执行外层循环
                continue retry;

        }
```

```
        }

        boolean workerStarted = false;
        boolean workerAdded = false;
        Worker w = null;
        try {
            w = new Worker(firstTask);
            final Thread t = w.thread;
            if (t != null) {
                // 加锁，原因，创建任务线程时，避免外部操作停止了线程池，销毁线程池是需要拿到锁的
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {
                    // Recheck while holding lock.
                    // Back out on ThreadFactory failure or if
                    // shut down before lock acquired.
                    // 重新获取线程池运行状态
                    int rs = runStateOf(ctl.get());

                    if (
                      // rs < SHUTDOWN 只有一种情况 rs = Running
                      rs < SHUTDOWN ||
                        // 或者 rs 的状态为 SHUTDOWN 且 firstTask 为null时，创建工作线程，处理阻塞队列中的任务
                        (rs == SHUTDOWN && firstTask == null)) {
                        // 线程是否已启动
                        if (t.isAlive())
                            // 因为状态没有改变就启动了，说明外部干扰线程池工作，抛出异常
                            throw new IllegalThreadStateException();
                        // 添加工作线程
                        workers.add(w);
                        // 调整之前记录的最大线程数数量
                        int s = workers.size();
                        if (s > largestPoolSize)
                            largestPoolSize = s;
                        // 修改状态
                        workerAdded = true;
                    }
                } finally {
                    mainLock.unlock();
```

```
                }
                if (workerAdded) {
                    // 启动线程
                    t.start();
                    workerStarted = true;
                }
            }
        } finally {
            if (! workerStarted)
                addWorkerFailed(w);
        }
        return workerStarted;
    }
```

- Worker ???

```
final void runWorker(Worker w) {
      // 获取当前线程
      Thread wt = Thread.currentThread();
      // 拿到当前任务
      Runnable task = w.firstTask;
      w.firstTask = null;
      w.unlock(); // allow interrupts
      boolean completedAbruptly = true;
      try {
          // 如果线程不等于空 或 为空的时候从阻塞队列中获取任务
          while (task != null || (task = getTask()) != null) {
              // 加锁，线程池shutdown了也不影响我继续执行
              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
              // 如果 线程池状态大于等于 STOP，代表线程池已经不是Running状态了，即出现问题了，则中断当前线程
              if ((runStateAtLeast(ctl.get(), STOP) ||
                   (Thread.interrupted() &&
                    runStateAtLeast(ctl.get(), STOP))) &&
                  !wt.isInterrupted())
                  wt.interrupt();
```

```
                try {
                    // 任务执行前置处理
                    beforeExecute(wt, task);
                    Throwable thrown = null;
                    try {
                        // 执行
                        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 {
                       // 任务执行后置处理
                        afterExecute(task, thrown);
                    }
                } finally {
                    task = null;
                    // 完成任务数加1
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            // woker 退出后续处理
            processWorkerExit(w, completedAbruptly);
        }
}
```

- getTask??

```
private Runnable getTask() {
    // 表示（非核心线程可以干掉）
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
      // ====================== 判断线程池状态 ======================================
```

```
        // 获取线程池完整信息
        int c = ctl.get();
        // 获取线程池状态
        int rs = runStateOf(c);

        // 如果线程池状态>=SHUTDOWN,则 rs 可能为 Shutdown，STOP, Tidying, TERMINATED 且（rs >=
STOP，则移除当前工作线程
        // 如果线程池状态>=SHUTDOWN 且队列为空，则移除当前工作线程
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
           // 则 递减ctl的workerCount字段，减少woker 数量，为移除当前工作线程做准备
            decrementWorkerCount();
            // 交付 processWorkerExit 后续处理
            return null;
        }
        // ====================== 判断工作线程数量
=======================================

        // 获取woker 数量
        int wc = workerCountOf(c);

         // 是否计时
        // allowCoreThreadTimeOut：表示线程池核心线程是否可以超时，一般为false
        // 或者工作线程数量已超过线程池核心线程数量
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        if (
          // 如果 wc 大于线程池线程最大数量 或 工作线程已超时
          (wc > maximumPoolSize || (timed && timedOut))
            //  wc > 1 或 队列为空， 尝试停止工作线程工作，因为wc>1 ，停止一个，还有其他的工作线程
            && (wc > 1 || workQueue.isEmpty())) {
            // 基于 CAS 停止工作线程工作， 只有一个可以成功
            if (compareAndDecrementWorkerCount(c))
                // 交付 processWorkerExit 后续处理
                return null;
            continue;
        }

        try {
            // 获取任务，是否计时，
            Runnable r = timed ?
```

```
                // 计时 走这个 从阻塞队列中阻塞一段时间获取任务
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                // 不计时，一直阻塞，直至拿到任务
                workQueue.take();
            if (r != null)
                // 返回任务
                return r;
            //  说明从队列中获取任务已超时，已达到线程最大生存时间，再走一遍
            timedOut = true;
        } catch (InterruptedException retry) {
            timedOut = false;
        }
    }
}
```

processWorkerExit ??

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    // 如果 processWorkerExit 操作 不是 getTask 中引起的，而是异常引起的（一般是异常由钩子函数引起的）
    if (completedAbruptly)
        // 执行方式有问题“不合法” 手动减少工作线程数量
        decrementWorkerCount();

    // 加锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        // 增加任务完成数量
        completedTaskCount += w.completedTasks;
        // 从阻塞队列中移除任务
        workers.remove(w);
    } finally {
        mainLock.unlock();
    }

    // 尝试把工作线程池状态从 过度状态 ----> 销毁状态
    tryTerminate();

    // 获取线程池信息
```

```
    int c = ctl.get();
    // 判断 线程池状态 是否比 STOP 小
    if (runStateLessThan(c, STOP)) {
        // 代表 线程池状态 是 Running 或 Shutdown
        // 若 !completedAbruptly 为true，则说明 执行的任务没有异常
        if (!completedAbruptly) {
            // 最小 工作线程数量
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
            // 若 最小数量为0 且队列是空的
            if (min == 0 && ! workQueue.isEmpty())
                // 则 将min置为 1
                min = 1;
            // 若线程池中的工作线程大于最小线程数量，则说明线程池工作线程还富余
            if (workerCountOf(c) >= min)
                return; // replacement not needed
        }
        // 说明 工作线程 以非正常方式结束，则新添加一个非核心工作线程
        // 若 阻塞队列不为空，且没有工作线程，则新添加一个非核心工作线程
        addWorker(null, false);
    }
}
```