并发编程4——线程池

线程池简介

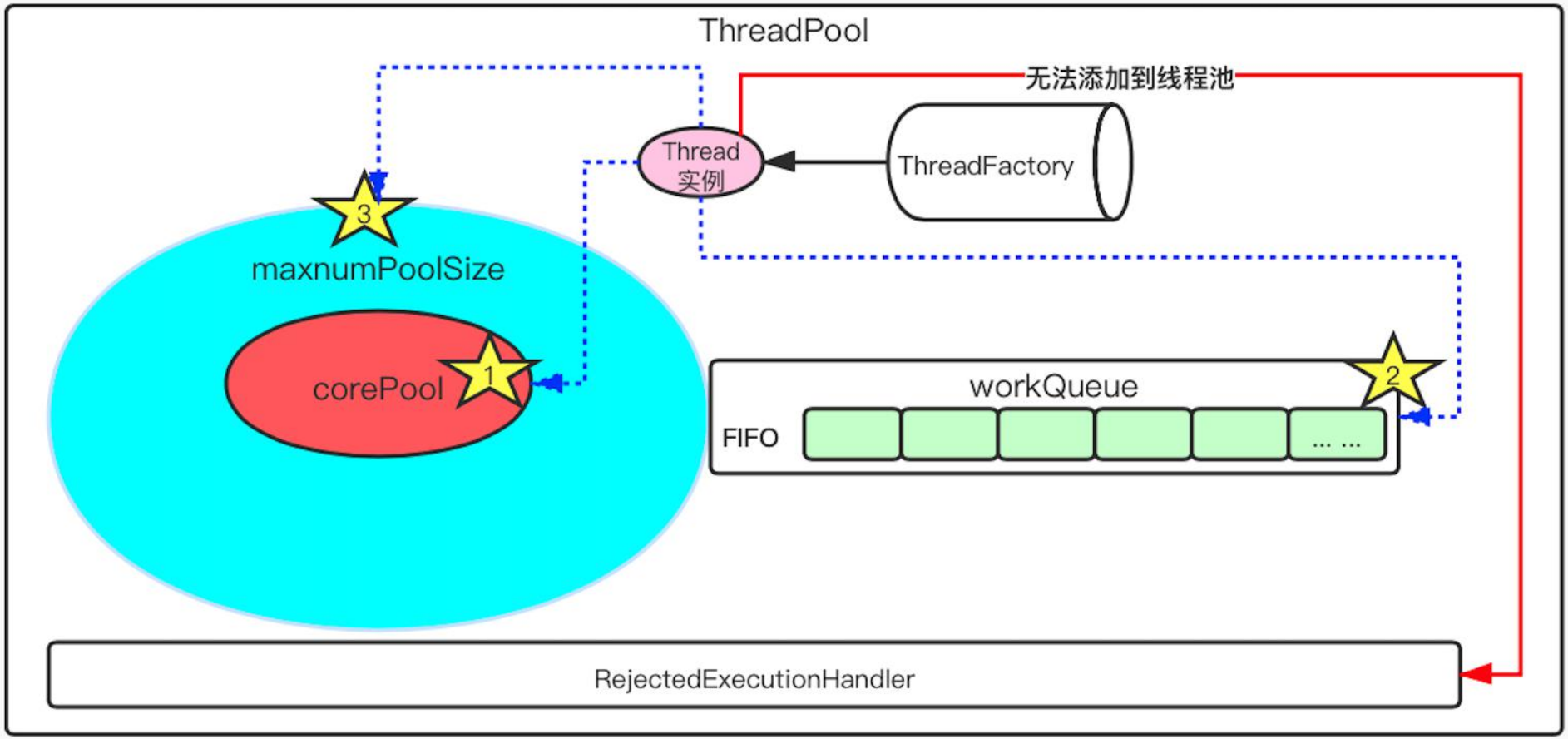
什么是线程池

- 为了避免系统频繁地创建和销毁线程，我们可以让创建的线程复用。

- 在线程池中，总有那么几个活跃线程，当你需要使用线程时，可以从池子中随便拿一个空闲线程，当完成工作时，并不急着关闭线程，而是将这个线程退回到线程池中，方便其他人使用。

- 简而言之，在使用线程池后，创建线程变成了从线程池获得空闲线程，关闭线程变成了向线程池归还线程，

线程池工作原理概览


ThreadPool
无法添加到线程池
Thread
实例
ThreadFactory
3
maxnumPoolSize
corePool
1
2
workQueue
FIFO
... ...
RejectedExecutionHandler

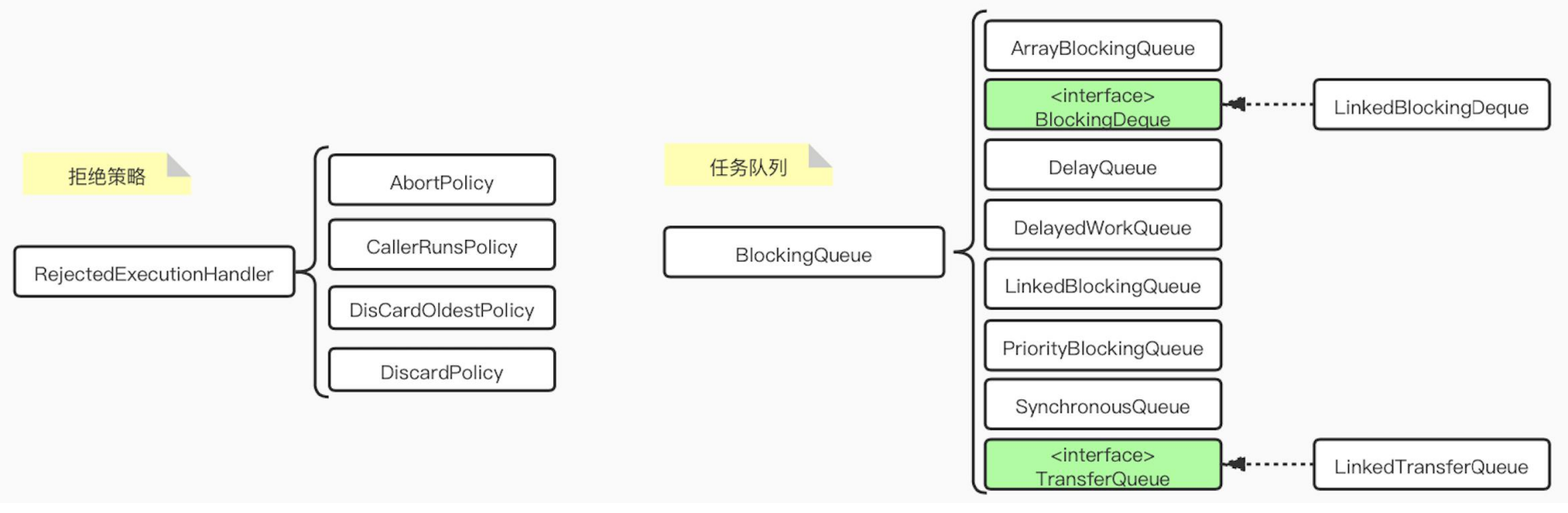
开篇的第一段源码——线程池构造函数

```
public ThreadPoolExecutor(int corePoolSize,     // 核心线程数
                          int maximumPoolSize, // 最大线程数
                          long keepAliveTime,   // 当线程数量大于核心线程数时，空闲线程最大的生存期为keepAliveTime
                          TimeUnit unit,         //  keepAliveTime的时间单位
                          BlockingQueue<Runnable> workQueue,// 待执行的任务队列
                          ThreadFactory threadFactory,        // 用于创建新线程的工厂类
                          RejectedExecutionHandler handler)   // 由于达到线程边界和队列容量而无法执行时使用的拒绝策略实现类
```

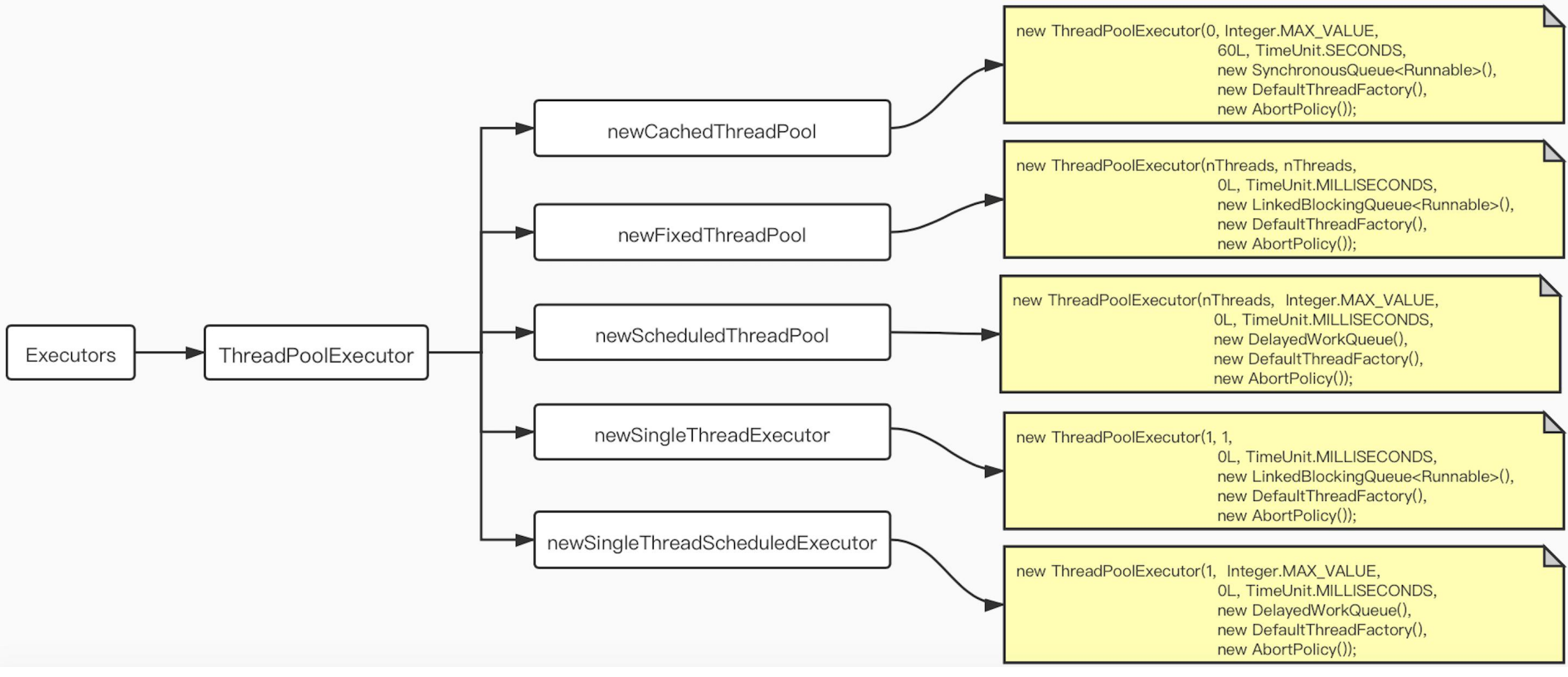
下面要介绍如下几个接口的实现：

- BlockingQueue：阻塞队列接口
- RejectedExecutionHandler ：拒绝策略接口

部分接口的实现


拒绝策略
RejectedExecutionHandler
AbortPolicy
CallerRunsPolicy
DisCardOldestPolicy
DiscardPolicy
任务队列
BlockingQueue
ArrayBlockingQueue
<interface>
BlockingDeque
LinkedBlockingDeque
DelayQueue
DelayedWorkQueue
LinkedBlockingQueue
PriorityBlockingQueue
SynchronousQueue
<interface>
TransferQueue
LinkedTransferQueue

Executors中提供的线程池实现

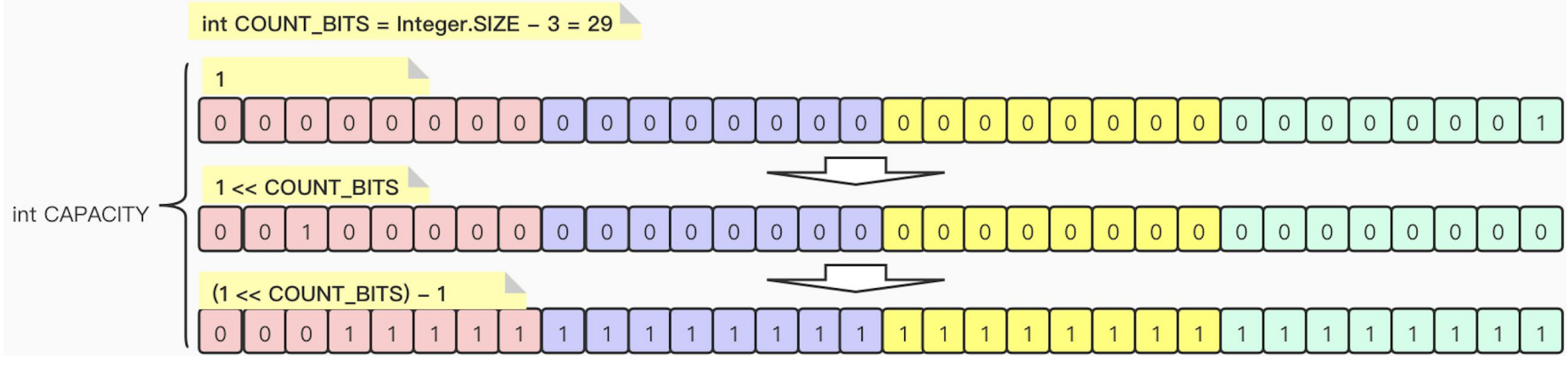


线程池源码解析

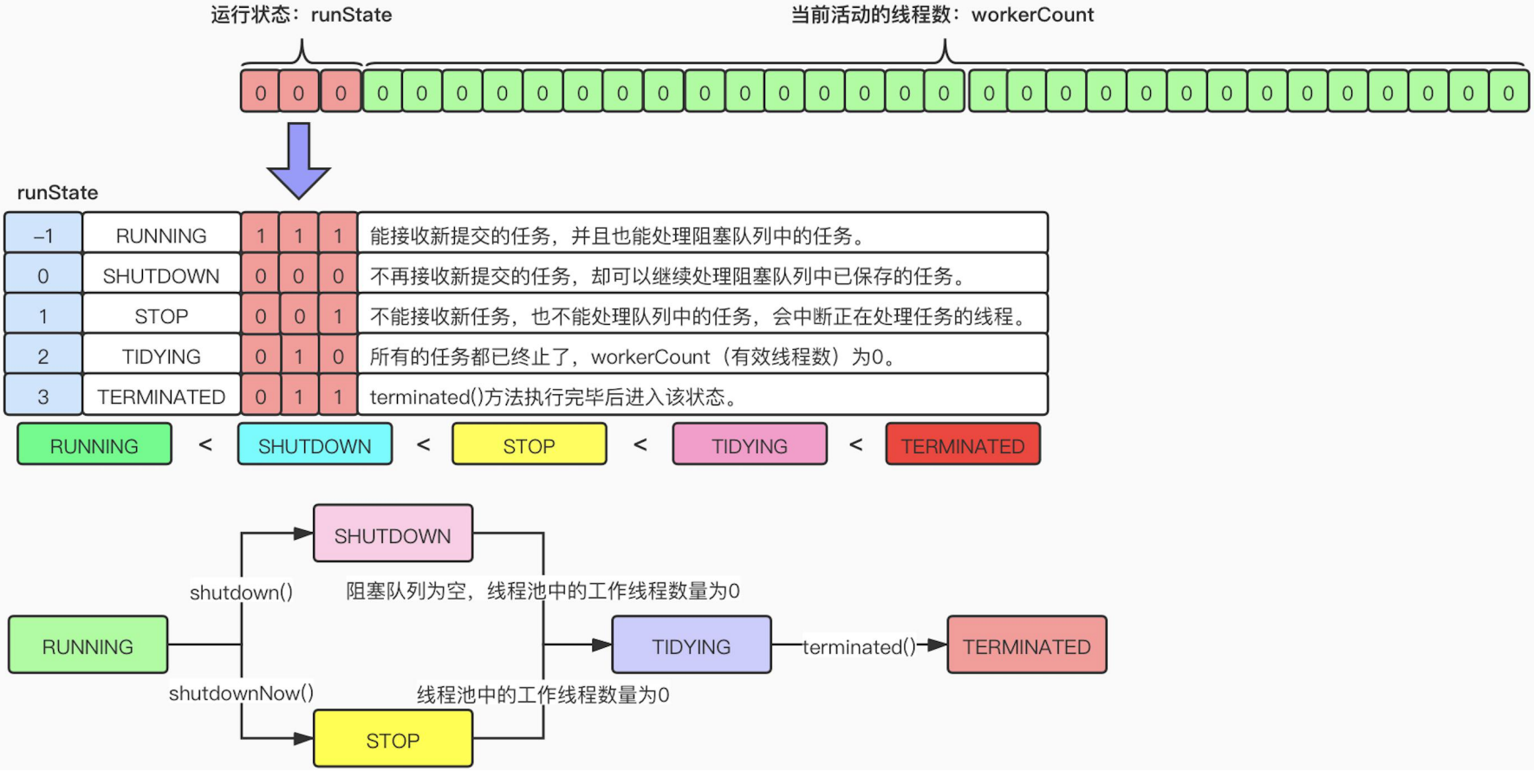
从这里开始吧——CAPACITY的初始化

```
@Native public static final int SIZE = 32;
private static final int COUNT_BITS = Integer.SIZE - 3;
private static final int CAPACITY = (1 << COUNT_BITS) - 1;

private static final int RUNNING = -1 << COUNT_BITS;    // 1110 0000 0000 0000 0000 0000 0000 0000
private static final int SHUTDOWN = 0 << COUNT_BITS;    // 0000 0000 0000 0000 0000 0000 0000 0000
private static final int STOP = 1 << COUNT_BITS;        // 0010 0000 0000 0000 0000 0000 0000 0000
private static final int TIDYING = 2 << COUNT_BITS;     // 0100 0000 0000 0000 0000 0000 0000 0000
private static final int TERMINATED = 3 << COUNT_BITS; // 0110 0000 0000 0000 0000 0000 0000 0000
```

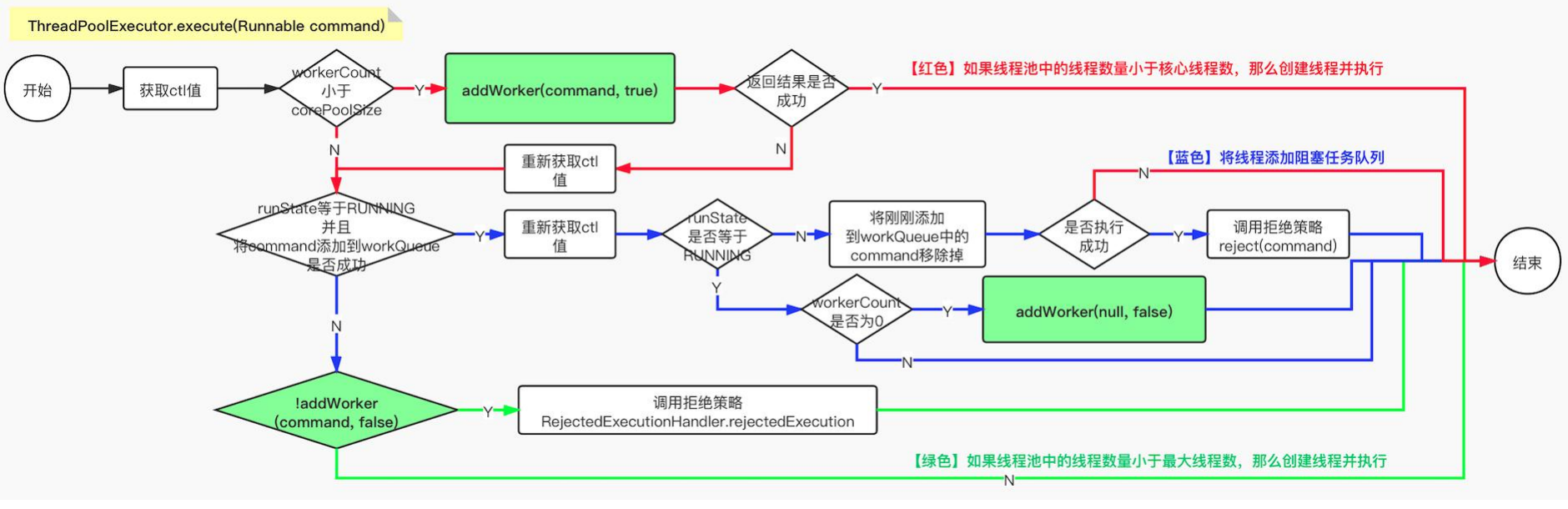
```
/**
 * 获取运行状态 RUNNING/SHUTDOWN/STOP/TIDYING/TERMINATED
 */
private static int runStateOf(int c) {
    // ~CAPACITY = 11100000000000000000000000000000
    return c & ~CAPACITY;
}

/**
 * 取出低位29位的值，表示获得当前活动的线程数
 */
private static int workerCountOf(int c) {
    // CAPACITY = 00011111111111111111111111111111
    return c & CAPACITY;
}
```



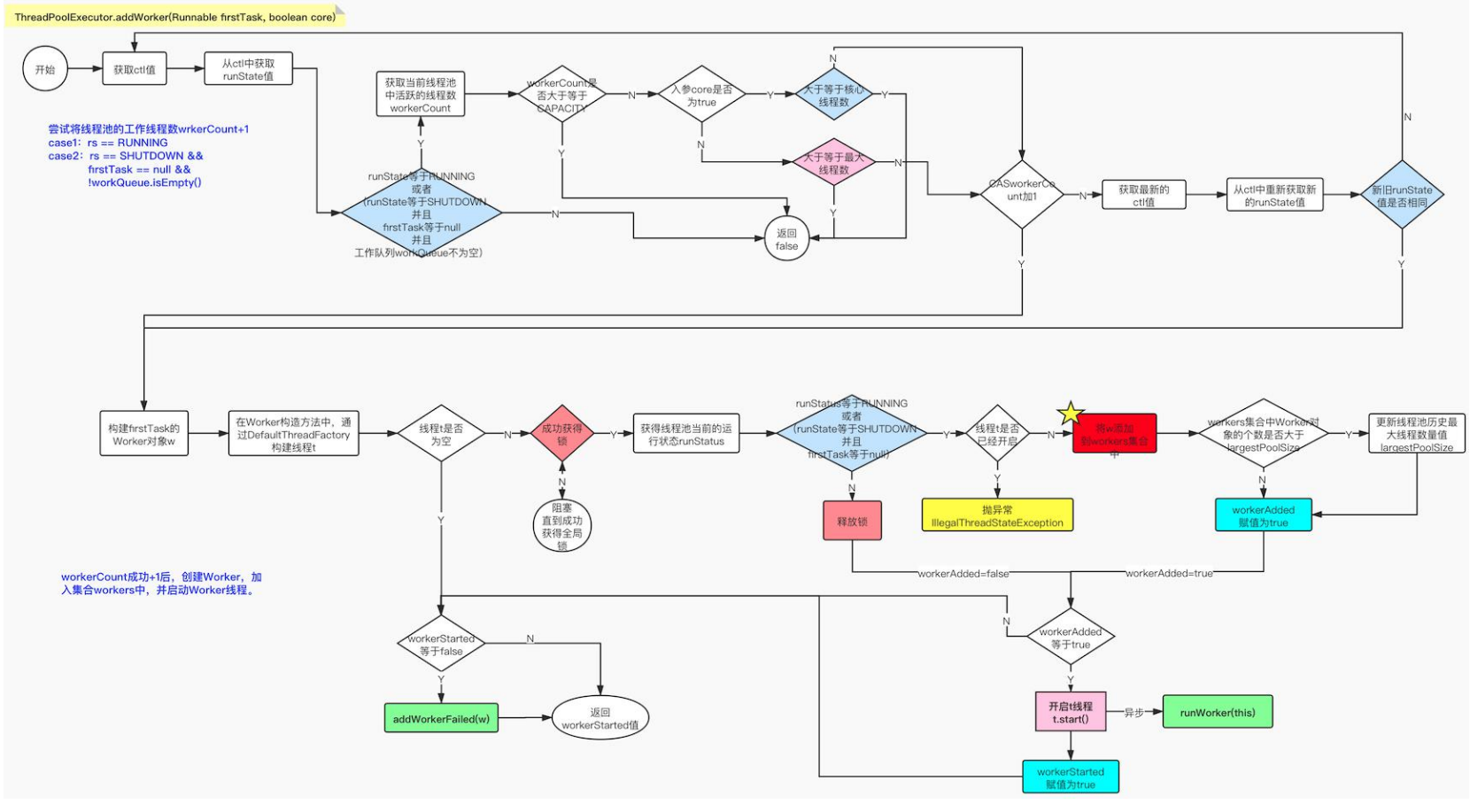
线程池运行状态和活动线程数



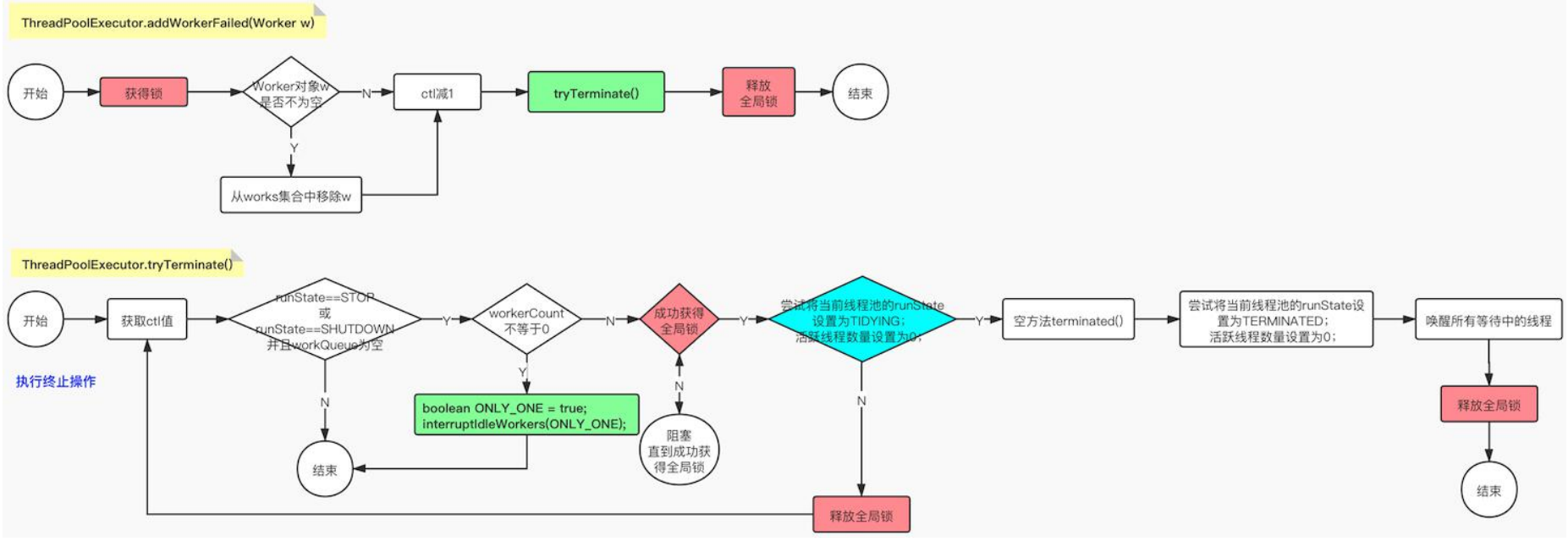
线程池源码——execute



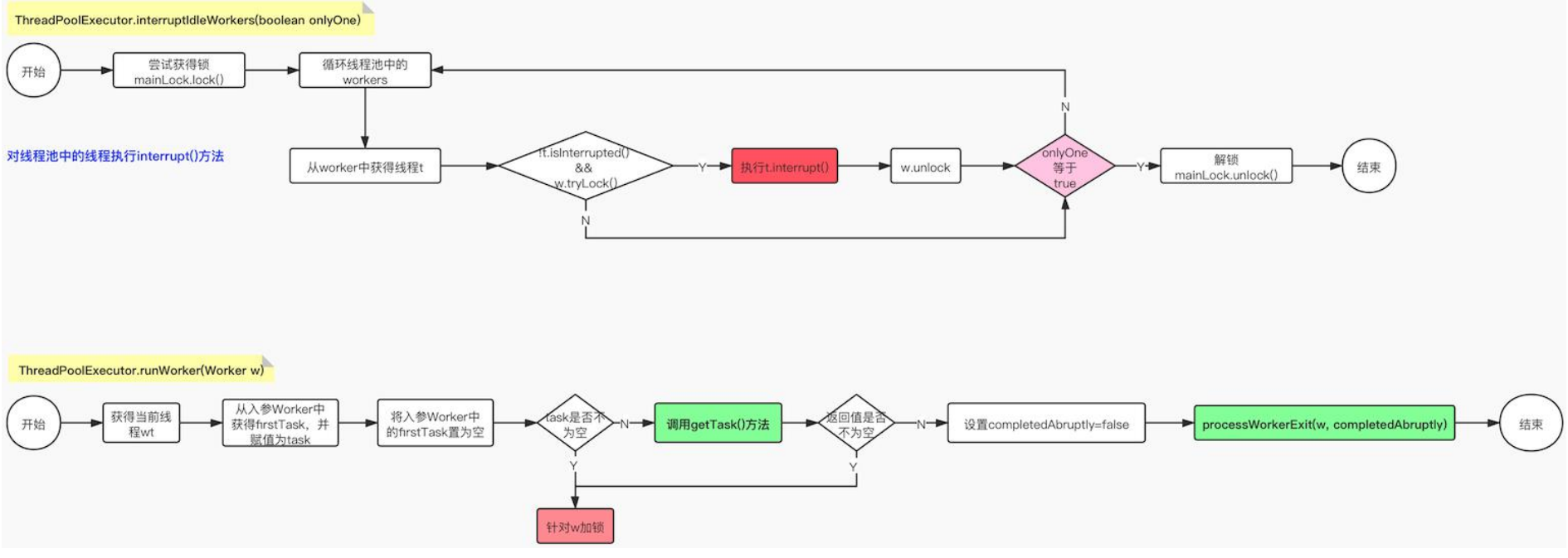
线程池源码——addWorker



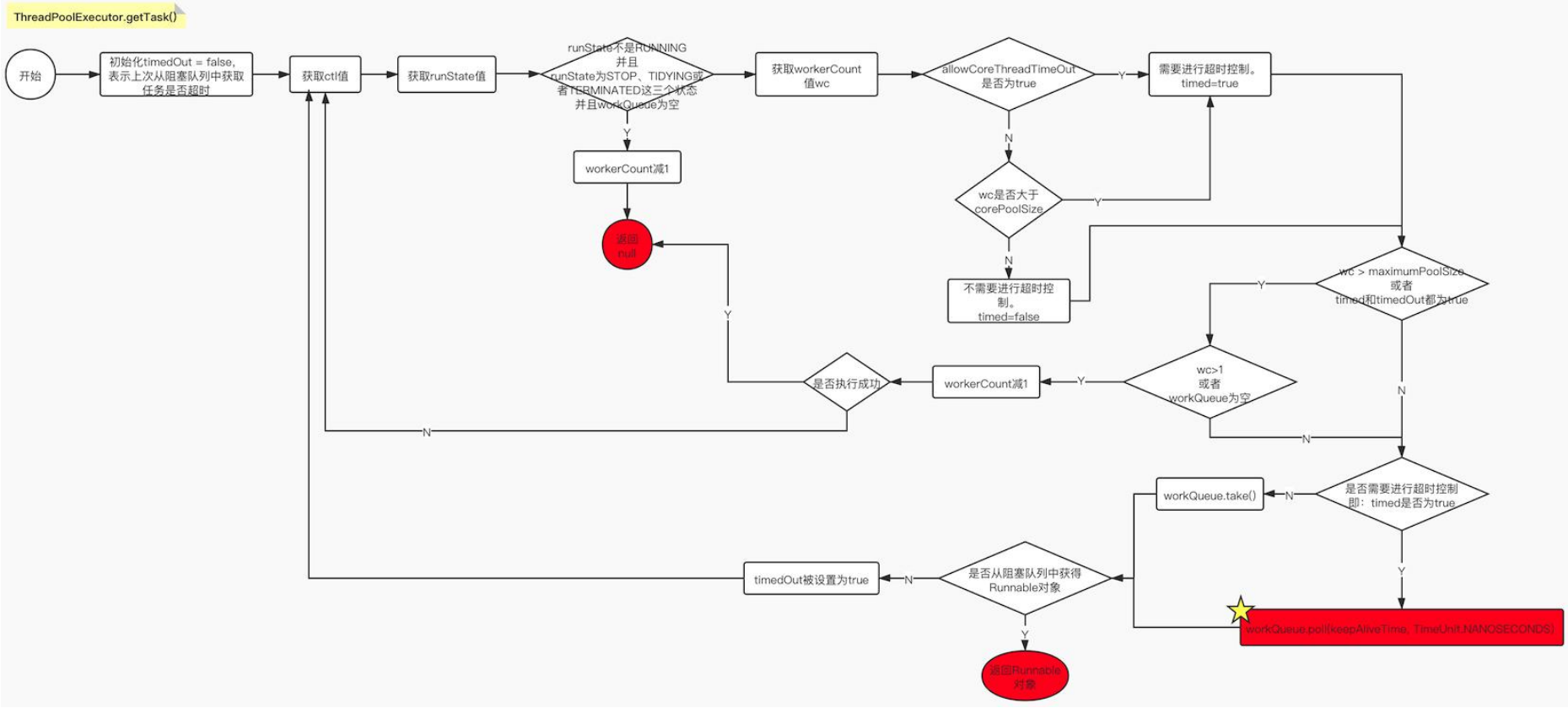
线程池源码——addWorkerFailed&tryTerminate



线程池源码——interruptIdelWorkers&runWorker



线程池源码——getTask



线程池源码——processWorkerExit