# JUC线程池: ThreadPoolExecutor详解

## 面试问题去理解

- 为什么要有线程池?
- Java是实现和管理线程池有哪些方式? 请简单举例如何使用。
- 为什么很多公司不允许使用Executors去创建线程池? 那么推荐怎么使用呢?
- ThreadPoolExecutor有哪些核心的配置参数? 请简要说明
- ThreadPoolExecutor可以创建哪是哪三种线程池呢?
- 当队列满了并且worker的数量达到maxSize的时候，会怎么样?
- 说说ThreadPoolExecutor有哪些RejectedExecutionHandler策略? 默认是什么策略?
- 简要说下线程池的任务执行机制? execute –> addWorker –>runworker (getTask)
- 线程池中任务是如何提交的?
- 线程池中任务是如何关闭的?
- 在配置线程池的时候需要考虑哪些配置因素?
- 如何监控线程池的状态?

## 为什么要有线程池

线程池能够对线程进行统一分配，调优和监控:

- 降低资源消耗(线程无限制地创建，然后使用完毕后销毁)
- 提高响应速度(无须创建线程)
- 提高线程的可管理性

## ThreadPoolExecutor例子

Java是如何实现和管理线程池的?

从JDK 5开始，把工作单元与执行机制分离开来，工作单元包括Runnable和Callable，而执行机制由Executor框架提供。

- WorkerThread

```
public class WorkerThread implements Runnable {

    private String command;

    public WorkerThread(String s){
        this.command=s;
    }

    @Override
    public void run() {
```

```
        System.out.println(Thread.currentThread().getName()+" Start. Command = "+command);
        processCommand();
        System.out.println(Thread.currentThread().getName()+" End.");
    }

    private void processCommand() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String toString(){
        return this.command;
    }
}
```

- SimpleThreadPool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SimpleThreadPool {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker);
          }
        executor.shutdown(); // This will make the executor accept no new threads and finish all existing threads in the queue
        while (!executor.isTerminated()) { // Wait until all threads are finish,and also you can use "executor.awaitTermination();" to wait
        }
        System.out.println("Finished all threads");
    }

}
```

程序中我们创建了固定大小为五个工作线程的线程池。然后分配给线程池十个工作，因为线程池大小为五，它将启动五个工作线程先处理五个工作，其他的工作则处于等待状态，一旦有工作完成，空闲下来工作线程就会捡取等待队列里的其他工作进行执行。

这里是以上程序的输出。

```
pool-1-thread-2 Start. Command = 1
pool-1-thread-4 Start. Command = 3
pool-1-thread-1 Start. Command = 0
pool-1-thread-3 Start. Command = 2
pool-1-thread-5 Start. Command = 4
pool-1-thread-4 End.
pool-1-thread-5 End.
pool-1-thread-1 End.
pool-1-thread-3 End.
pool-1-thread-3 Start. Command = 8
pool-1-thread-2 End.
```

```
pool-1-thread-2 Start. Command = 9
pool-1-thread-1 Start. Command = 7
pool-1-thread-5 Start. Command = 6
pool-1-thread-4 Start. Command = 5
pool-1-thread-2 End.
pool-1-thread-4 End.
pool-1-thread-3 End.
pool-1-thread-5 End.
pool-1-thread-1 End.
Finished all threads
```

输出表明线程池中至始至终只有五个名为 "pool-1-thread-1" 到 "pool-1-thread-5" 的五个线程，这五个线程不随着工作的完成而消亡，会一直存在，并负责执行分配给线程池的任务，直到线程池消亡。

Executors 类提供了使用了 ThreadPoolExecutor 的简单的 ExecutorService 实现，但是 ThreadPoolExecutor 提供的功能远不止于此。我们可以在创建 ThreadPoolExecutor 实例时指定活动线程的数量，我们也可以限制线程池的大小并且创建我们自己的 RejectedExecutionHandler 实现来处理不能适应工作队列的工作。

这里是我们自定义的 RejectedExecutionHandler 接口的实现。

- RejectedExecutionHandlerImpl.java

```
import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadPoolExecutor;

public class RejectedExecutionHandlerImpl implements RejectedExecutionHandler {

    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        System.out.println(r.toString() + " is rejected");
    }

}
```

ThreadPoolExecutor 提供了一些方法，我们可以使用这些方法来查询 executor 的当前状态，线程池大小，活动线程数量以及任务数量。因此我是用来一个监控线程在特定的时间间隔内打印 executor 信息。

- MyMonitorThread.java

```
import java.util.concurrent.ThreadPoolExecutor;

public class MyMonitorThread implements Runnable
{
    private ThreadPoolExecutor executor;

    private int seconds;

    private boolean run=true;

    public MyMonitorThread(ThreadPoolExecutor executor, int delay)
    {
        this.executor = executor;
        this.seconds=delay;
    }

    public void shutdown(){
        this.run=false;
    }
```

```
    @Override
    public void run()
    {
        while(run){
                System.out.println(
                    String.format("[monitor] [%d/%d] Active: %d, Completed: %d, Task: %d,
isShutdown: %s, isTerminated: %s",
                        this.executor.getPoolSize(),
                        this.executor.getCorePoolSize(),
                        this.executor.getActiveCount(),
                        this.executor.getCompletedTaskCount(),
                        this.executor.getTaskCount(),
                        this.executor.isShutdown(),
                        this.executor.isTerminated()));
                try {
                    Thread.sleep(seconds*1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }

    }
}
```

这里是使用 ThreadPoolExecutor 的线程池实现例子。

- WorkerPool.java

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class WorkerPool {

    public static void main(String args[]) throws InterruptedException{
        //RejectedExecutionHandler implementation
        RejectedExecutionHandlerImpl rejectionHandler = new RejectedExecutionHandlerImpl();
        //Get the ThreadFactory implementation to use
        ThreadFactory threadFactory = Executors.defaultThreadFactory();
        //creating the ThreadPoolExecutor
        ThreadPoolExecutor executorPool = new ThreadPoolExecutor(2, 4, 10, TimeUnit.SECONDS, new
ArrayBlockingQueue<Runnable>(2), threadFactory, rejectionHandler);
        //start the monitoring thread
        MyMonitorThread monitor = new MyMonitorThread(executorPool, 3);
        Thread monitorThread = new Thread(monitor);
        monitorThread.start();
        //submit work to the thread pool
        for(int i=0; i<10; i++){
            executorPool.execute(new WorkerThread("cmd"+i));
        }

        Thread.sleep(30000);
        //shut down the pool
        executorPool.shutdown();
        //shut down the monitor thread
        Thread.sleep(5000);
        monitor.shutdown();
```

```
    }
}
```

注意在初始化 ThreadPoolExecutor 时，我们保持初始池大小为 2，最大池大小为 4 而工作队列大小为 2。因此如果已经有四个正在执行的任务而此时分配来更多任务的话，工作队列将仅仅保留他们(新任务)中的两个，其他的将会被 RejectedExecutionHandlerImpl 处理。

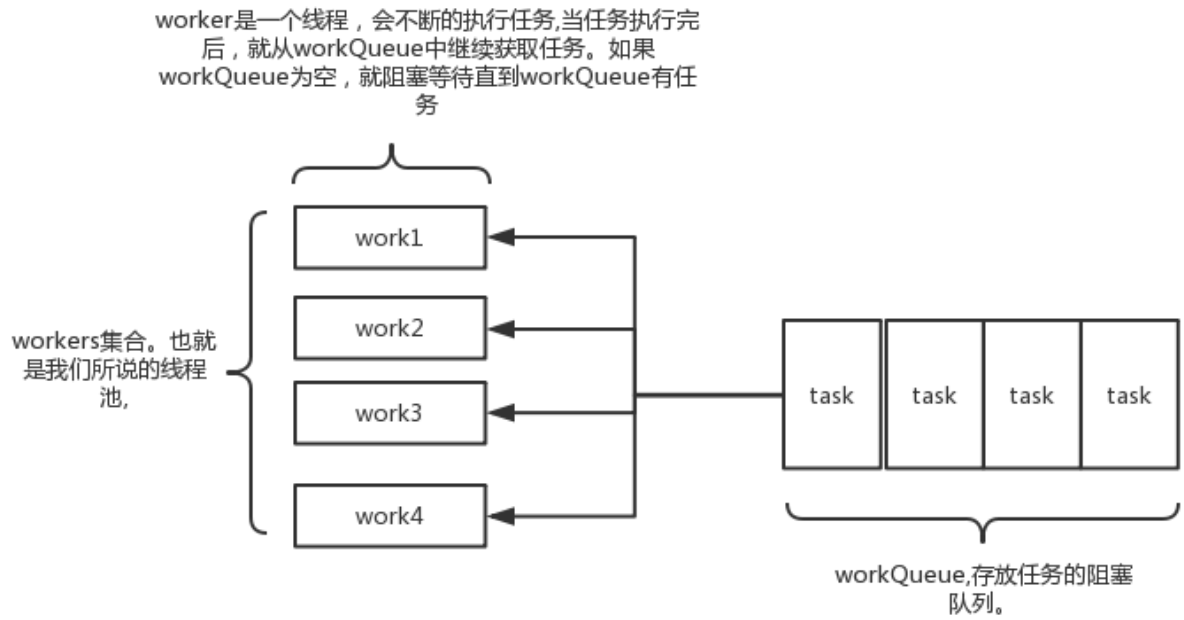
上面程序的输出可以证实以上观点。

```
pool-1-thread-1 Start. Command = cmd0
pool-1-thread-4 Start. Command = cmd5
cmd6 is rejected
pool-1-thread-3 Start. Command = cmd4
pool-1-thread-2 Start. Command = cmd1
cmd7 is rejected
cmd8 is rejected
cmd9 is rejected
[monitor] [0/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false
[monitor] [4/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false
pool-1-thread-4 End.
pool-1-thread-1 End.
pool-1-thread-2 End.
pool-1-thread-3 End.
pool-1-thread-1 Start. Command = cmd3
pool-1-thread-4 Start. Command = cmd2
[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false
[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false
pool-1-thread-1 End.
pool-1-thread-4 End.
[monitor] [4/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false
[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true
[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true
```

注意 executor 的活动任务、完成任务以及所有完成任务，这些数量上的变化。我们可以调用 shutdown() 方法来结束所有提交的任务并终止线程池。

# ThreadPoolExecutor使用详解

其实java线程池的实现原理很简单，说白了就是一个线程集合workerSet和一个阻塞队列workQueue。当用户向线程池提交一个任务(也就是线程)时，线程池会先将任务放入workQueue中。workerSet中的线程会不断的从workQueue中获取线程然后执行。当workQueue中没有任务的时候，worker就会阻塞，直到队列中有任务了就取出来继续执行。




## *Execute*原理

当一个任务提交至线程池之后:

1. 线程池首先当前运行的线程数量是否少于corePoolSize。如果是，则创建一个新的工作线程来执行任务。如果都在执行任务，则进入2.
2. 判断BlockingQueue是否已经满了，倘若还没有满，则将线程放入BlockingQueue。否则进入3.
3. 如果创建一个新的工作线程将使当前运行的线程数量超过maximumPoolSize，则交给RejectedExecutionHandler来处理任务。

当ThreadPoolExecutor创建新线程时，通过CAS来更新线程池的状态ctl.

## 参数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler)
```

- `corePoolSize` 线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize, 即使有其他空闲线程能够执行新来的任务, 也会继续创建线程；如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。
- `workQueue` 用来保存等待被执行的任务的阻塞队列. 在JDK中提供了如下阻塞队列:

- `ArrayBlockingQueue`: 基于数组结构的有界阻塞队列，按FIFO排序任务；
- `LinkedBlockingQuene`: 基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于ArrayBlockingQuene；
- `SynchronousQuene`: 一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene；
- `PriorityBlockingQuene`: 具有优先级的无界阻塞队列；

`LinkedBlockingQueue`比`ArrayBlockingQueue`在插入删除节点性能方面更优，但是二者在`put()`, `take()`任务的时均需要加锁， `SynchronousQueue`使用无锁算法，根据节点的状态判断执行，而不需要用到锁，其核心是`Transfer.transfer()`.

- `maximumPoolSize` 线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize；当阻塞队列是无界队列, 则maximumPoolSize则不起作用, 因为无法提交至核心线程池的线程会一直持续地放入workQueue.
- `keepAliveTime` 线程空闲时的存活时间，即当线程没有任务执行时，该线程继续存活的时间；默认情况下，该参数只在线程数大于corePoolSize时才有用, 超过这个时间的空闲线程将被终止；
- `unit` keepAliveTime的单位
- `threadFactory` 创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名。默认为DefaultThreadFactory
- `handler` 线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略:
  - `AbortPolicy`: 直接抛出异常，默认策略；
  - `CallerRunsPolicy`: 用调用者所在的线程来执行任务；
  - `DiscardOldestPolicy`: 丢弃阻塞队列中靠最前的任务，并执行当前任务；
  - `DiscardPolicy`: 直接丢弃任务；

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

# 三种类型

## newFixedThreadPool

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

线程池的线程数量达corePoolSize后，即使线程池没有可执行任务时，也不会释放线程。

FixedThreadPool的工作队列为无界队列LinkedBlockingQueue(队列容量为Integer.MAX_VALUE), 这会导致以下问题:

- 线程池里的线程数量不超过corePoolSize,这导致了maximumPoolSize和keepAliveTime将会是个无用参数
- 由于使用了无界队列, 所以FixedThreadPool永远不会拒绝, 即饱和策略失效

## newSingleThreadExecutor

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

初始化的线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行.

由于使用了无界队列, 所以SingleThreadPool永远不会拒绝, 即饱和策略失效

### newCachedThreadPool

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

线程池的线程数可达到Integer.MAX_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列； 和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销； 执行过程与前两种稍微不同:

- 主线程调用SynchronousQueue的offer()方法放入task, 倘若此时线程池中有空闲的线程尝试读取 SynchronousQueue的task, 即调用了SynchronousQueue的poll(), 那么主线程将该task交给空闲线程. 否则执行(2)
- 当线程池为空或者没有空闲的线程, 则创建新的线程执行任务.
- 执行完任务的线程倘若在60s内仍空闲, 则会被终止. 因此长时间空闲的CachedThreadPool不会持有任何线程资源.

## 关闭线程池

遍历线程池中的所有线程，然后逐个调用线程的interrupt方法来中断线程.

### 关闭方式 - shutdown

将线程池里的线程状态设置成SHUTDOWN状态, 然后中断所有没有正在执行任务的线程.

### 关闭方式 - shutdownNow

将线程池里的线程状态设置成STOP状态, 然后停止所有正在执行或暂停任务的线程. 只要调用这两个关闭方法中的任意一个, isShutDown() 返回true. 当所有任务都成功关闭了, isTerminated()返回true.

## ThreadPoolExecutor源码详解

# 几个关键属性

```
//这个属性是用来存放 当前运行的worker数量以及线程池状态的
//int是32位的，这里把int的高3位拿来充当线程池状态的标志位,后29位拿来充当当前运行worker的数量
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
//存放任务的阻塞队列
private final BlockingQueue<Runnable> workQueue;
//worker的集合,用set来存放
private final HashSet<Worker> workers = new HashSet<Worker>();
//历史达到的worker数最大值
private int largestPoolSize;
//当队列满了并且worker的数量达到maxSize的时候,执行具体的拒绝策略
private volatile RejectedExecutionHandler handler;
//超出coreSize的worker的生存时间
private volatile long keepAliveTime;
//常驻worker的数量
private volatile int corePoolSize;
//最大worker的数量,一般当workQueue满了才会用到这个参数
private volatile int maximumPoolSize;
```

# 内部状态

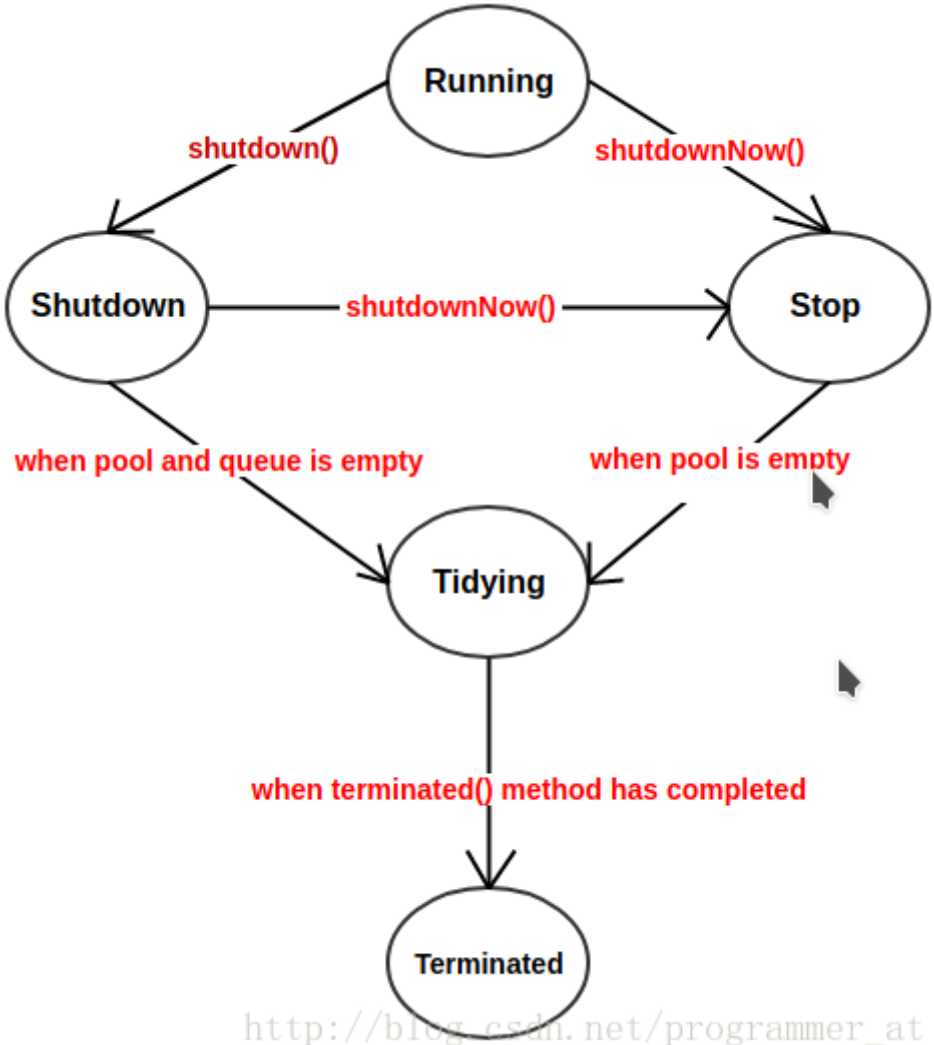
```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
private static final int COUNT_BITS = Integer.SIZE - 3;
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

// runState is stored in the high-order bits
private static final int RUNNING    = -1 << COUNT_BITS;
private static final int SHUTDOWN   =  0 << COUNT_BITS;
private static final int STOP       =  1 << COUNT_BITS;
private static final int TIDYING    =  2 << COUNT_BITS;
private static final int TERMINATED =  3 << COUNT_BITS;

// Packing and unpacking ctl
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; }
```

其中AtomicInteger变量ctl的功能非常强大: 利用低29位表示线程池中线程数，通过高3位表示线程池的运行状态:

- RUNNING: -1 << COUNT_BITS，即高3位为111，该状态的线程池会接收新任务，并处理阻塞队列中的任务;
- SHUTDOWN: 0 << COUNT_BITS，即高3位为000，该状态的线程池不会接收新任务，但会处理阻塞队列中的任务;
- STOP : 1 << COUNT_BITS，即高3位为001，该状态的线程不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务;
- TIDYING : 2 << COUNT_BITS，即高3位为010, 所有的任务都已经终止;
- TERMINATED: 3 << COUNT_BITS，即高3位为011, terminated()方法已经执行完成




# 任务的执行

> execute –> addWorker –>runworker (getTask)

线程池的工作线程通过Woker类实现，在ReentrantLock锁的保证下，把Woker实例插入到HashSet后，并启动Woker中的线程。 从Woker类的构造方法实现可以发现: 线程工厂在创建线程thread时，将Woker实例本身this作为参数传入，当执行start方法启动线程thread时，本质是执行了Worker的runWorker方法。 firstTask执行完成之后，通过getTask方法从阻塞队列中获取等待的任务，如果队列中没有任务，getTask方法会被阻塞并挂起，不会占用cpu资源；

## execute()方法

ThreadPoolExecutor.execute(task)实现了Executor.execute(task)

```
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();
    if (workerCountOf(c) < corePoolSize) {
    //workerCountOf获取线程池的当前线程数；小于corePoolSize，执行addWorker创建新线程执行command任务
       if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // double check: c, recheck
    // 线程池处于RUNNING状态，把提交的任务成功放入阻塞队列中
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        // recheck and if necessary 回滚到入队操作前，即倘若线程池shutdown状态，就remove(command)
        //如果线程池没有RUNNING，成功从阻塞队列中删除任务，执行reject方法处理任务
        if (! isRunning(recheck) && remove(command))
            reject(command);
        //线程池处于running状态，但是没有线程，则创建线程
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    // 往线程池中创建新的线程失败，则reject任务
    else if (!addWorker(command, false))
        reject(command);
}
```

- 为什么需要double check线程池的状态?

在多线程环境下，线程池的状态时刻在变化，而ctl.get()是非原子操作，很有可能刚获取了线程池状态后线程池状态就改变了。判断是否将command加入workque是线程池之前的状态。倘若没有double check，万一线程池处于非running状态(在多线程环境下很有可能发生)，那么command永远不会执行。

## addWorker方法

从方法execute的实现可以看出: addWorker主要负责创建新的线程并执行任务 线程池创建新线程执行任务时，需要获取全局锁:

```
private final ReentrantLock mainLock = new ReentrantLock();
```

```
private boolean addWorker(Runnable firstTask, boolean core) {
    // CAS更新线程池数量
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
```

```
                firstTask == null &&
                ! workQueue.isEmpty()))
            return false;

        for (;;) {
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get();  // Re-read ctl
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }

    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
            // 线程池重入锁
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                int rs = runStateOf(ctl.get());

                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive()) // precheck that t is startable
                        throw new IllegalThreadStateException();
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    workerAdded = true;
                }
            } finally {
                mainLock.unlock();
            }
            if (workerAdded) {
                t.start();  // 线程启动，执行任务(Worker.thread(firstTask).start());
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

## Worker类的runworker方法

```
private final class Worker extends AbstractQueuedSynchronizer implements Runnable{
    Worker(Runnable firstTask) {
        setState(-1); // inhibit interrupts until runWorker
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this); // 创建线程
    }
    /** Delegates main run loop to outer runWorker  */
    public void run() {
        runWorker(this);
    }
    // ...
}
```

- 继承了AQS类，可以方便的实现工作线程的中止操作；
- 实现了Runnable接口，可以将自身作为一个任务在工作线程中执行；
- 当前提交的任务firstTask作为参数传入Worker的构造方法；

一些属性还有构造方法:

```
//运行的线程,前面addWorker方法中就是直接通过启动这个线程来启动这个worker
final Thread thread;
//当一个worker刚创建的时候,就先尝试执行这个任务
Runnable firstTask;
//记录完成任务的数量
volatile long completedTasks;

Worker(Runnable firstTask) {
    setState(-1); // inhibit interrupts until runWorker
    this.firstTask = firstTask;
    //创建一个Thread,将自己设置给他,后面这个thread启动的时候,也就是执行worker的run方法
    this.thread = getThreadFactory().newThread(this);
}
```

runWorker方法是线程池的核心:

- 线程启动之后，通过unlock方法释放锁，设置AQS的state为0，表示运行可中断；
- Worker执行firstTask或从workQueue中获取任务:
  - 进行加锁操作，保证thread不被其他线程中断(除非线程池被中断)
  - 检查线程池状态，倘若线程池处于中断状态，当前线程将中断。
  - 执行beforeExecute
  - 执行任务的run方法
  - 执行afterExecute方法
  - 解锁操作

> 通过getTask方法从阻塞队列中获取等待的任务，如果队列中没有任务，getTask方法会被阻塞并挂起，不会占用cpu资源；

```
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        // 先执行firstTask，再从workerQueue中取task(getTask())
```

```
        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 {
                    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);
    }
}
```

## getTask方法

下面来看一下getTask()方法，这里面涉及到keepAliveTime的使用，从这个方法我们可以看出先吃池是怎么让超过corePoolSize的那部分worker销毁的。

```
private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        // Are workers subject to culling?
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
```

```
            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;
            }
        }
    }
```

注意这里一段代码是keepAliveTime起作用的关键:

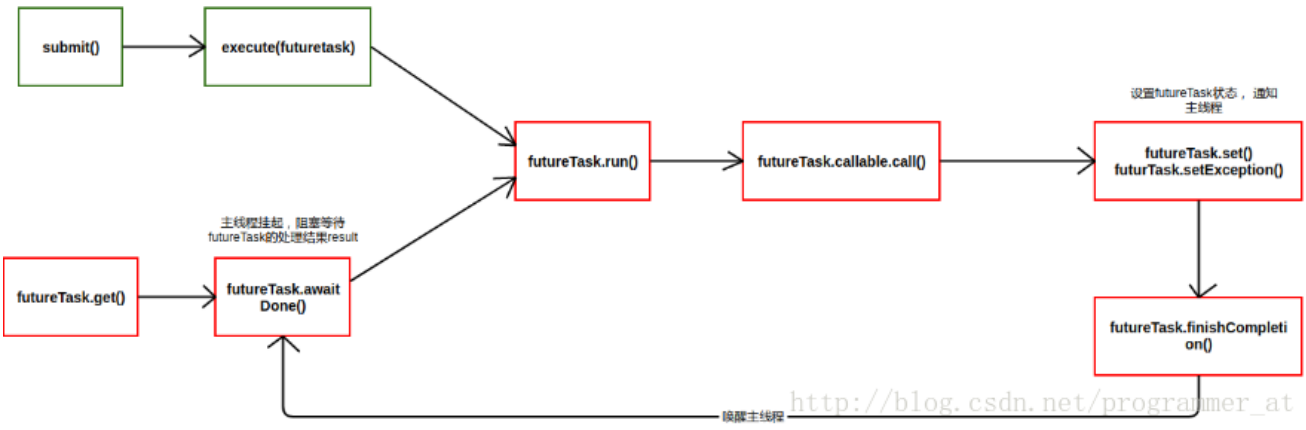
```
boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
Runnable r = timed ?
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take();
```

allowCoreThreadTimeOut为false，线程即使空闲也不会被销毁；倘若为ture，在keepAliveTime内仍空闲则会被销毁。

如果线程允许空闲等待而不被销毁timed == false， workQueue.take任务: 如果阻塞队列为空，当前线程会被挂起等待；当队列中有任务加入时，线程被唤醒，take方法返回任务，并执行；

如果线程不允许无休止空闲timed == true, workQueue.poll任务: 如果在keepAliveTime时间内，阻塞队列还是没有任务，则返回null；

## 任务的提交




1. submit任务，等待线程池execute
2. 执行FutureTask类的get方法时，会把主线程封装成WaitNode节点并保存在waiters链表中， 并阻塞等待运行结果；
3. FutureTask任务执行完成后，通过UNSAFE设置waiters相应的waitNode为null，并通过LockSupport类unpark方法唤醒主线程；

```
public class Test{
    public static void main(String[] args) {

        ExecutorService es = Executors.newCachedThreadPool();
        Future<String> future = es.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                try {
                    TimeUnit.SECONDS.sleep(2);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return "future result";
            }
        });
        try {
            String result = future.get();
            System.out.println(result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在实际业务场景中，Future和Callable基本是成对出现的，Callable负责产生结果，Future负责获取结果。

1. Callable接口类似于Runnable，只是Runnable没有返回值。
2. Callable任务除了返回正常结果之外，如果发生异常，该异常也会被返回，即Future可以拿到异步执行任务各种结果；
3. Future.get方法会导致主线程阻塞，直到Callable任务执行完成；

## submit方法

AbstractExecutorService.submit()实现了ExecutorService.submit() 可以获取执行完的返回值, 而ThreadPoolExecutor 是AbstractExecutorService.submit()的子类，所以submit方法也是ThreadPoolExecutor`的方法。

```
// submit()在ExecutorService中的定义
<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);
```

```
// submit方法在AbstractExecutorService中的实现
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    // 通过submit方法提交的Callable任务会被封装成了一个FutureTask对象。
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}
```

通过submit方法提交的Callable任务会被封装成了一个FutureTask对象。通过Executor.execute方法提交FutureTask到线程池中等待被执行，最终执行的是FutureTask的run方法；

## FutureTask对象

`public class FutureTask<V> implements RunnableFuture<V>` 可以将FutureTask提交至线程池中等待被执行(通过FutureTask的run方法来执行)

- 内部状态

```
/* The run state of this task, initially NEW.
    * ...
    * Possible state transitions:
    * NEW -> COMPLETING -> NORMAL
    * NEW -> COMPLETING -> EXCEPTIONAL
    * NEW -> CANCELLED
    * NEW -> INTERRUPTING -> INTERRUPTED
    */
private volatile int state;
private static final int NEW          = 0;
private static final int COMPLETING   = 1;
private static final int NORMAL       = 2;
private static final int EXCEPTIONAL  = 3;
private static final int CANCELLED    = 4;
private static final int INTERRUPTING = 5;
private static final int INTERRUPTED  = 6;
```

内部状态的修改通过sun.misc.Unsafe修改

- get方法

```
public V get() throws InterruptedException, ExecutionException {
    int s = state;
    if (s <= COMPLETING)
        s = awaitDone(false, 0L);
    return report(s);
}
```

内部通过awaitDone方法对主线程进行阻塞，具体实现如下:

```
private int awaitDone(boolean timed, long nanos)
    throws InterruptedException {
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    WaitNode q = null;
    boolean queued = false;
    for (;;) {
        if (Thread.interrupted()) {
            removeWaiter(q);
            throw new InterruptedException();
        }

        int s = state;
        if (s > COMPLETING) {
            if (q != null)
                q.thread = null;
            return s;
        }
        else if (s == COMPLETING) // cannot time out yet
            Thread.yield();
        else if (q == null)
            q = new WaitNode();
        else if (!queued)
```

```
            queued = UNSAFE.compareAndSwapObject(this, waitersOffset,q.next = waiters, q);
        else if (timed) {
            nanos = deadline - System.nanoTime();
            if (nanos <= 0L) {
                removeWaiter(q);
                return state;
            }
            LockSupport.parkNanos(this, nanos);
        }
        else
            LockSupport.park(this);
    }
}
```

1. 如果主线程被中断，则抛出中断异常；
2. 判断FutureTask当前的state，如果大于COMPLETING，说明任务已经执行完成，则直接返回；
3. 如果当前state等于COMPLETING，说明任务已经执行完，这时主线程只需通过yield方法让出cpu资源，等待state变成NORMAL；
4. 通过WaitNode类封装当前线程，并通过UNSAFE添加到waiters链表；
5. 最终通过LockSupport的park或parkNanos挂起线程；

run方法

```
public void run() {
    if (state != NEW || !UNSAFE.compareAndSwapObject(this, runnerOffset, null,
Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex);
            }
            if (ran)
                set(result);
        }
    } finally {
        // runner must be non-null until state is settled to
        // prevent concurrent calls to run()
        runner = null;
        // state must be re-read after nulling runner to prevent
        // leaked interrupts
        int s = state;
        if (s >= INTERRUPTING)
            handlePossibleCancellationInterrupt(s);
    }
}
```

FutureTask.run方法是在线程池中被执行的，而非主线程

1. 通过执行Callable任务的call方法；
2. 如果call执行成功，则通过set方法保存结果；

3. 如果call执行有异常，则通过setException保存异常；

# 任务的关闭

shutdown方法会将线程池的状态设置为SHUTDOWN,线程池进入这个状态后,就拒绝再接受任务,然后会将剩余的任务全部执行完

```
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //检查是否可以关闭线程
        checkShutdownAccess();
        //设置线程池状态
        advanceRunState(SHUTDOWN);
        //尝试中断worker
        interruptIdleWorkers();
            //预留方法,留给子类实现
        onShutdown(); // hook for ScheduledThreadPoolExecutor
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
}

private void interruptIdleWorkers() {
    interruptIdleWorkers(false);
}

private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //遍历所有的worker
        for (Worker w : workers) {
            Thread t = w.thread;
            //先尝试调用w.tryLock(),如果获取到锁,就说明worker是空闲的,就可以直接中断它
            //注意的是,worker自己本身实现了AQS同步框架,然后实现的类似锁的功能
            //它实现的锁是不可重入的,所以如果worker在执行任务的时候,会先进行加锁,这里tryLock()就会返回
false
            if (!t.isInterrupted() && w.tryLock()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
    }
}
```

shutdownNow做的比较绝，它先将线程池状态设置为STOP，然后拒绝所有提交的任务。最后中断左右正在运行中的worker,然后清空任务队列。

```
public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();
        //检测权限
        advanceRunState(STOP);
        //中断所有的worker
        interruptWorkers();
        //清空任务队列
        tasks = drainQueue();
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
    return tasks;
}

private void interruptWorkers() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //遍历所有worker，然后调用中断方法
        for (Worker w : workers)
            w.interruptIfStarted();
    } finally {
        mainLock.unlock();
    }
}
```

# 更深入理解

## 为什么线程池不允许使用*Executors*去创建？推荐方式是什么？

线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。 说明：Executors各个方法的弊端：

- newFixedThreadPool和newSingleThreadExecutor:  主要问题是堆积的请求处理队列可能会耗费非常大的内存，甚至OOM。
- newCachedThreadPool和newScheduledThreadPool:  主要问题是线程数最大数是Integer.MAX_VALUE，可能会创建数量非常多的线程，甚至OOM。

### 推荐方式 1

首先引入：commons-lang3包

```
ScheduledExecutorService executorService = new ScheduledThreadPoolExecutor(1,
        new BasicThreadFactory.Builder().namingPattern("example-schedule-pool-
%d").daemon(true).build());
```

## 推荐方式 2

首先引入：com.google.guava包

```
ThreadFactory namedThreadFactory = new ThreadFactoryBuilder().setNameFormat("demo-pool-
%d").build();

//Common Thread Pool
ExecutorService pool = new ThreadPoolExecutor(5, 200, 0L, TimeUnit.MILLISECONDS, new
LinkedBlockingQueue<Runnable>(1024), namedThreadFactory, new ThreadPoolExecutor.AbortPolicy());

// excute
pool.execute(()-> System.out.println(Thread.currentThread().getName()));

 //gracefully shutdown
pool.shutdown();
```

## 推荐方式 3

spring配置线程池方式：自定义线程工厂bean需要实现ThreadFactory，可参考该接口的其它默认实现类，使用方式直接注入bean调用execute(Runnable task)方法即可

```
    <bean id="userThreadPool"
class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
        <property name="corePoolSize" value="10" />
        <property name="maxPoolSize" value="100" />
        <property name="queueCapacity" value="2000" />

    <property name="threadFactory" value= threadFactory />
        <property name="rejectedExecutionHandler">
            <ref local="rejectedExecutionHandler" />
        </property>
    </bean>

    //in code
    userThreadPool.execute(thread);
```

# 配置线程池需要考虑因素

从任务的优先级，任务的执行时间长短，任务的性质(CPU密集/ IO密集)，任务的依赖关系这四个角度来分析。并且近可能地使用有界的工作队列。

性质不同的任务可用使用不同规模的线程池分开处理:

- CPU密集型: 尽可能少的线程，Ncpu+1
- IO密集型: 尽可能多的线程, Ncpu*2，比如数据库连接池
- 混合型: CPU密集型的任务与IO密集型任务的执行时间差别较小，拆分为两个线程池；否则没有必要拆分。

# 监控线程池的状态

可以使用ThreadPoolExecutor以下方法:

- `getTaskCount()` Returns the approximate total number of tasks that have ever been scheduled for execution.
- `getCompletedTaskCount()` Returns the approximate total number of tasks that have completed execution. 返回结果少于getTaskCount()。
- `getLargestPoolSize()` Returns the largest number of threads that have ever simultaneously been in the pool. 返回结果小于等于maximumPoolSize
- `getPoolSize()` Returns the current number of threads in the pool.
- `getActiveCount()` Returns the approximate number of threads that are actively executing tasks