

Executors框架

接口

Executor

接口，解耦任务逻辑和任务调度。

方法

```
/**
 * 在以后的某个时刻，执行给定的command
 * 可以是一个线程池中的新线程，也可以是在调用线程，由实现决定
 */
void execute(Runnable command);
```

通过线程执行一个任务的代码

```
new Thread(()->{
    // command
    System.out.println("hello task");
}).start();
```

上面代码分3段逻辑

1. 一个实现Runnable接口的command
2. 创建一个线程，用于执行这个command
3. 启动线程，等待调度

如果，用Executor框架，选择合适的执行器，将command提交给执行器即可；

```
Executor executor = ...;
executor.execute(command);
```

相比于传统任务的执行，优点是

让开发人员专注于任务逻辑代码(command)的实现，而不需要关注任务的执行，如：线程的创建、任务调度等

ExecutorService

接口，继承于Executor，增强了对线程池的管理、异步任务、批量任务的支持

方法

```
/**
 * 关闭执行器
 * 1.已经提交的任务继续执行，不接受新任务；
 * 2.如果已经shutdown，调用没有副作用
 * 3.不会等待之前已提交的任务执行完成（不阻塞调用线程），如果需要可以使用#awaitTermination
 */
void shutdown();
```

```
/**
 * 立即关闭执行器
 * 1.终止正在执行的任务，但不能保证停止成功
 * 2.暂停处理正在等待的任务
 * 3.返回（已提交）等待执行的任务列表
 */
List<Runnable> shutdownNow();
```

```
/**
 * 执行器是否被关闭
 */
boolean isShutdown();
```

```
/**
 * 执行器是否完成
 * 1.shutdown后所有任务都已完成，返回true
 * 2.如果先前没有调用shutdown或shutdownNow，则永不返回true
 */
boolean isTerminated();
```

```
/**
 * 阻塞调用线程，等待执行器完成
 * 1.阻塞调用线程，至所有任务都已完成（前提是shutdown后）
 * 2.有超时设置
 * 3.等待时可被中断
 */
boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException;
```

```
/**
 * 提交一个有返回值的任务[Callable]用于执行（异步执行）
 * Future维护着任务的返回结果，Future#get()在任务成功完成后返回任务的结果
 */
<T> Future<T> submit(Callable<T> task);
```

```
/**
 * 提交一个可执行的任务[Runnable]用于执行（异步执行）
 * 任务完成后，Future#get()返回给定的result
 */
<T> Future<T> submit(Runnable task, T result);
```

```
/**
 * 提交一个可执行的任务[Runnable]用于执行（异步执行）
 * 任务完成后，Future#get()返回null
 */
Future<?> submit(Runnable task);
```

```
/**
 * 执行给定的tasks
 * 所有任务执行完成后，返回包含了状态和结果的future集合，与task列表顺序相同
 * 1.同步
 * 2.返回结果列表中每个的Future#isDone()都为true
 */
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
    throws InterruptedException;
```

```
/**
 * 超时版本
 * 任务完成或者超时先达，未完成任务被取消
 */
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
    long timeout, TimeUnit unit)
    throws InterruptedException;
```

```
/**
 * 其中一个任务成功完成（无异常）后立即返回
 * 未完成任务被取消
 */
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
    throws InterruptedException, ExecutionException;
```

```
/**
 * 超时版本
 */
<T> T invokeAny(Collection<? extends Callable<T>> tasks,
    long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException;
```

ScheduledExecutorService

接口，继承于ExecutorService，定时或周期性调度

方法

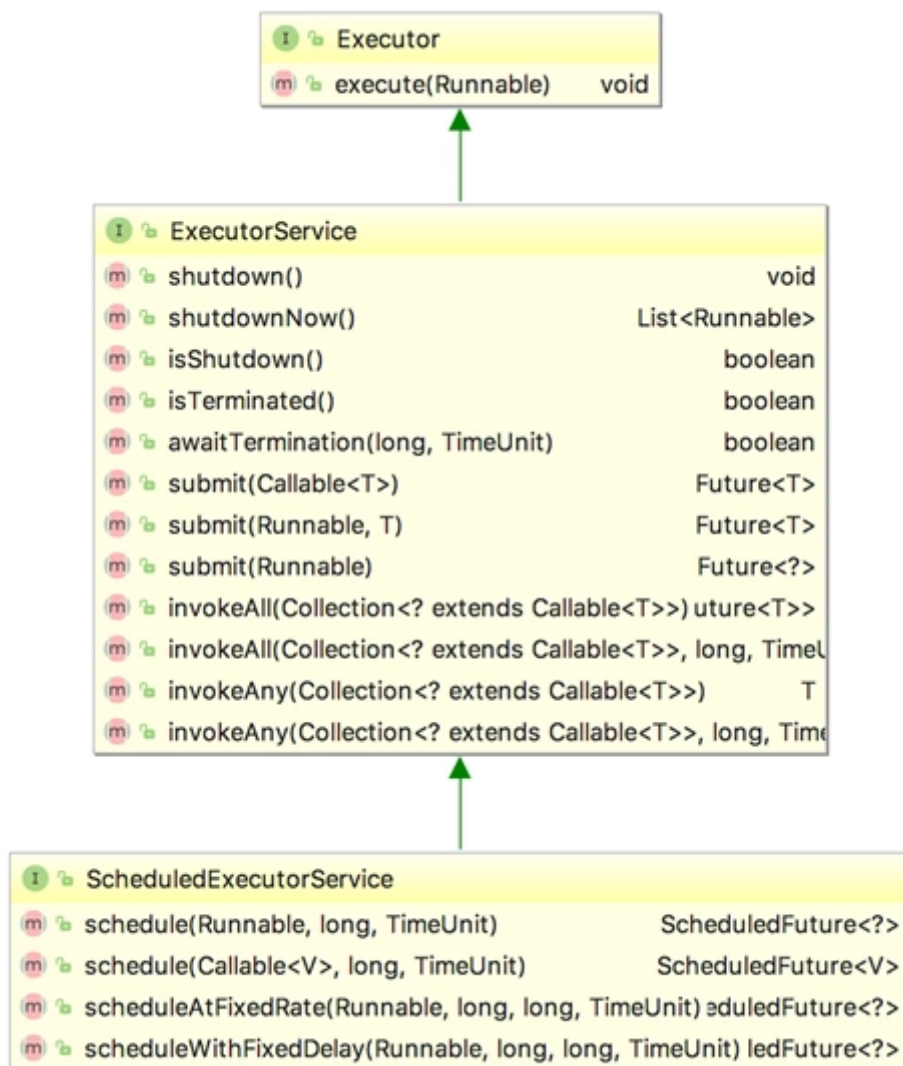
```
/**
 * 在给定的延迟之后，执行任务（仅一次性的）
 * 任务完成之后，Future#get()返回null
 */
ScheduledFuture<?> schedule(Runnable command,
    long delay, TimeUnit unit);
```

```
/**
 * callable版本
 */
<V> ScheduledFuture<V> schedule(Callable<V> callable,
                                long delay, TimeUnit unit);
```

```
/**
 * 初始延迟之后，以固定周期执行任务，不受任务执行时间影响
 * 首次执行时间:  $F = \text{now()} + \text{initialDelay}$ 
 * 二次执行时间:  $F + \text{period}$ 
 * N次执行时间:  $F + (N-1) * \text{period}$ 
 */
ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                        long initialDelay,
                                        long period,
                                        TimeUnit unit);
```

```
/**
 * 初始延迟之后，以固定延迟执行任务（任务执行完后再以delay延时执行下个任务），受执行时间影响
 * 首次执行时间:  $F = \text{now()} + \text{initialDelay}$ 
 * 二次执行时间:  $F_2 = F + \text{delay} + F$  (首次执行时间)
 * N次执行时间:  $F_2 + \text{delay} + F(N-1 \text{ 执行时间})$ 
 */
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                           long initialDelay,
                                           long delay,
                                           TimeUnit unit);
```

关系图



实现

ThreadPoolExecutor

继承自 **AbstractExecutorService**，提供了 **ExecutorService** 接口的默认实现。

构造器

[illegible]

```

        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }

```

源码分析

线程池状态

ThreadPoolExecutor内部定义了一个AtomicInteger变量——**ctl**，通过按位划分的方式，在一个变量中记录线程池状态和工作线程数——**低29位保存线程数，高3位保存线程池状态**：

```

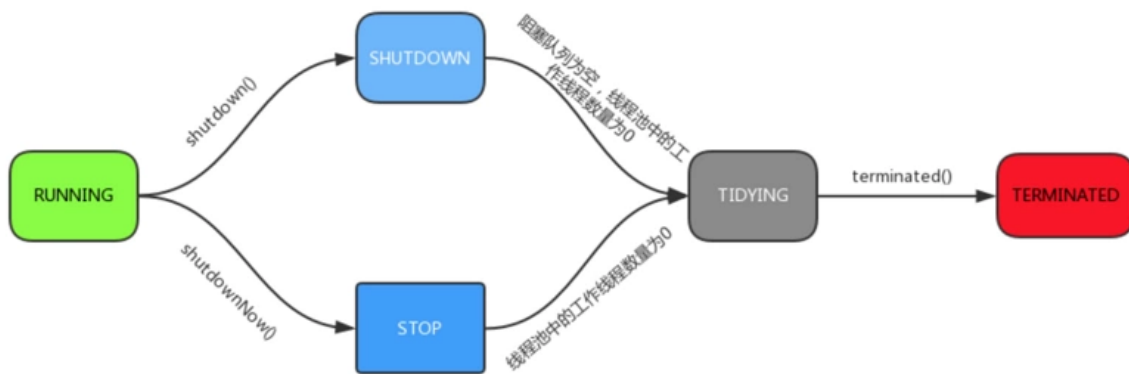
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
// 低29位用于记录工作线程数
private static final int COUNT_BITS = Integer.SIZE - 3;
// 最大线程数 2^29-1
private static final int CAPACITY = (1 << COUNT_BITS) - 1; // 00011111
11111111 11111111 11111111

// runState is stored in the high-order bits
// 高3位标识线程池状态
private static final int RUNNING = -1 << COUNT_BITS; // 11100000 00000000
00000000 00000000
private static final int SHUTDOWN = 0 << COUNT_BITS; // 00000000 00000000
00000000 00000000
private static final int STOP = 1 << COUNT_BITS; // 00100000 00000000
00000000 00000000
private static final int TIDYING = 2 << COUNT_BITS; // 01000000 00000000
00000000 00000000
private static final int TERMINATED = 3 << COUNT_BITS; // 01100000 00000000
00000000 00000000

```

定义了5种状态

- **RUNNING** : 接受新任务, 且处理已经进入阻塞队列的任务
- **SHUTDOWN** : 不接受新任务, 但处理已经进入阻塞队列的任务
- **STOP** : 不接受新任务, 且不处理已经进入阻塞队列的任务, 同时中断正在运行的任务
- **TIDYING** : 所有任务都已终止, 工作线程数为0, 线程转化为TIDYING状态并准备调用terminated方法
- **TERMINATED** : terminated方法已经执行完成



工作线程

Work是ThreadPoolExecutor的内部类，实现了AQS；ThreadPoolExecutor以HashSet保存工作线程

```

/**
 * Set containing all worker threads in pool. Accessed only when
 * holding mainLock.
 */
private final HashSet<Worker> workers = new HashSet<Worker>();

```

Work的定义

```

private final class Worker
    extends AbstractQueuedSynchronizer
    implements Runnable
{
    /**
     * This class will never be serialized, but we provide a
     * serialVersionUID to suppress a javac warning.
     */
    private static final long serialVersionUID = 6138294804551838833L;

    /** Thread this worker is running in. Null if factory fails. */
    // 每个worker都有一个线程，除非线程工厂类创建失败
    final Thread thread;
    /** Initial task to run. Possibly null. */
    Runnable firstTask;
    /** Per-thread task counter */
    volatile long completedTasks;

    /**
     * Creates with given first task and thread from ThreadFactory.
     * @param firstTask the first task (null if none)
     */
    Worker(Runnable firstTask) {
        setState(-1); // inhibit interrupts until runWorker -1 初始状态
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);
    }

    /** Delegates main run loop to outer runWorker */
    // 【重点】执行任务

```

```

public void run() {
    runWorker(this);
}

// Lock methods
// The value 0 represents the unlocked state.
// The value 1 represents the locked state.
protected boolean isHeldExclusively() {
    return getState() != 0;
}

protected boolean tryAcquire(int unused) {
    if (compareAndSetState(0, 1)) {
        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    }
    return false;
}

protected boolean tryRelease(int unused) {
    setExclusiveOwnerThread(null);
    setState(0);
    return true;
}

public void lock()      { acquire(1); }
public boolean tryLock() { return tryAcquire(1); }
public void unlock()    { release(1); }
public boolean isLocked() { return isHeldExclusively(); }

void interruptIfStarted() {
    Thread t;
    if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
        try {
            t.interrupt();
        } catch (SecurityException ignore) {
        }
    }
}
}

```

任务提交

任务提交是调用执行器的submit方法，AbstractExecutorService是ExecutorService的默认实现，主要实现了submit、invokeAny、invokeAll这三类方法

```

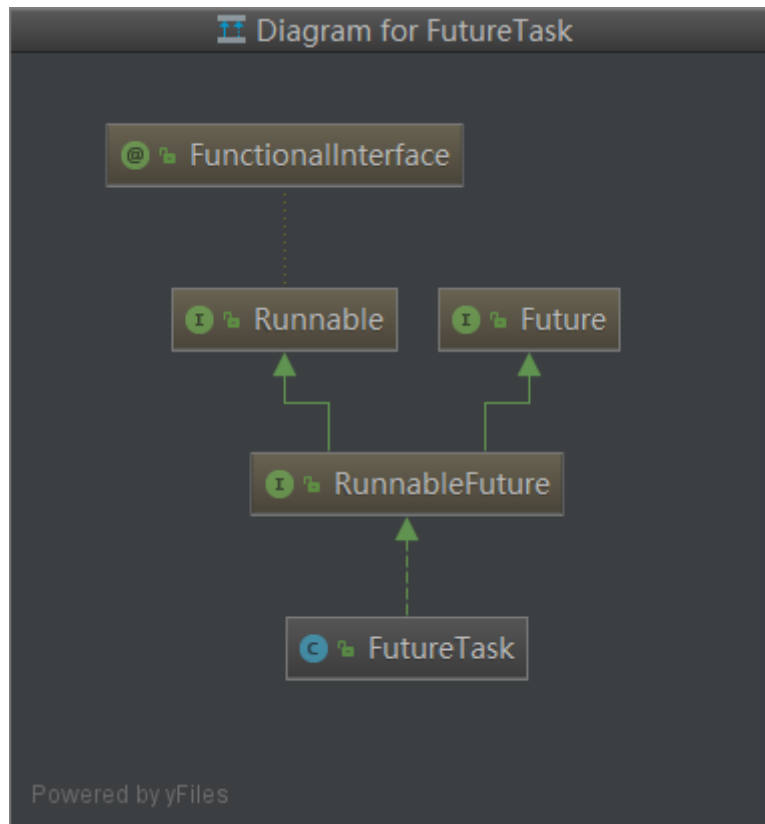
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    // 对task和result封装
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    // 模板方法，子类实现
    execute(ftask);
    return ftask;
}

```



```
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
    return new FutureTask<T>(runnable, value);
}
```

FutureTask就是对Future的实现，RunnableFuture同时继承了Runnable和Future，即：支持异步处理任务



任务执行

由上见，任务submit后，具体执行是在子类execute方法中，ThreadPoolExecutor#execute(...)

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();

    int c = ctl.get();
    // CASE1: 工作线程小于核心线程数
    if (workerCountOf(c) < corePoolSize) {
        // 增加核心线程数并执行
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // CASE2: (工作线程创建失败或者工作线程数>=核心线程数) 校验线程池running状态并添加到队列
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        // 重新校验状态, 非running状态 移出队列
        if (! isRunning(recheck) && remove(command))
            reject(command);
        // 工作线程=0
        else if (workerCountOf(recheck) == 0)
            // 创建一个空任务线程
    }
}
```

```

        addworker(null, false);
    }
    // CASE3: 线程池不是running或者添加队列失败
    else if (!addworker(command, false))
        // 执行拒绝策略
        reject(command);
}

```

提交任务-->执行任务时添加工作线程或加入队列

```

/**
 * 添加新的worker
 * @param firstTask 创建一个线程执行这个任务
 * @param core true: 核心线程 false: 非核心线程
 */
private boolean addworker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();
        // 运行状态
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        /**
         * 线程池状态的判断
         *
         * 分解条件
         * 1. rs >= SHUTDOWN 首先状态必须是SHUTDOWN、STOP 或 TIDYING 或 TERMINATED
         * 2. && 之后是对SHUTDOWN状态的例外判断
         *
         * [NOT]: STOP 或 TIDYING 或 TERMINATED状态下，不运行创建worker
         *         >= SHUTDOWN且firstTask != null不再接受新任务的提交
         *         >= SHUTDOWN且workQueue为空，没有任务要执行
         */
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
                firstTask == null &&
                ! workQueue.isEmpty()))
            return false;

        for (;;) {
            // 工作线程数
            int wc = workerCountOf(c);
            // [NOT] 超过CAPACITY或者超过corePoolSize或maximumPoolSize
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            // [OK] 工作线程+1
            if (compareAndIncrementWorkerCount(c))
                break retry; // 终止跳出自旋
            c = ctl.get(); // Re-read ctl
            if (runStateOf(c) != rs) // 线程池状态发生变化
                continue retry; // 重复自旋
            // else CAS failed due to workerCount change; retry inner loop
        }
    }
}

```

```

boolean workerStarted = false;
boolean workerAdded = false;
worker w = null;
try {
    // 构造worker, 注意线程的target是worker实例
    w = new worker(firstTask);
    final Thread t = w.thread;
    if (t != null) {
        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 IllegalStateException();
                // 加入工作线程
                workers.add(w);
                int s = workers.size();
                if (s > largestPoolSize)
                    largestPoolSize = s;
                workerAdded = true;
            }
        } finally {
            mainLock.unlock();
        }
        if (workerAdded) {
            // 【重点】
            t.start(); // 工作线程创建后, 执行任务, 执行的是worker的run(), 最终回到
task.run()

            workerStarted = true;
        }
    }
} finally {
    if (! workerStarted)
        // 添加工作线程失败 执行回滚
        addWorkerFailed(w);
}
return workerStarted;
}

```

工作线程运行

ThreadPoolExecutor#runWorker(...)

```

final void runWorker(worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    // 相当于worker复位: 线程置null, state=0
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {

```

```

while (task != null || (task = getTask()) != null) {
    // 持有锁: worker正在执行任务
    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
    /**
     *
     * 1. 如果线程状态是STOP/TIDYING/TERMINATED, 工作线程wt必须是中断状态, 否则
    将其中断
     *
     * 2. 如果线程是中断状态, 线程池状态必须是STOP/TIDYING/TERMINATED,
     *    即: 保证RUNNING/SHUTDOWN线程状态是正常的
     */
    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 {
    // 任务执行完或者异常
    processWorkerExit(w, completedAbruptly);
}
}

```

工作线程获取任务

通过自旋, 不断的从阻塞队列中获取任务, 获取不到的可能有

- 线程池状态STOP/TIDYING/TERMINATED
- 线程池状态为SHUTDOWN且状态队列为空

前提: 工作线程>1 或者 (工作线程=1且队列为空)

- 工作线程数> maximumPoolSize
- 回旋时工作线程未在线程存活期内获取到任务

ThreadPoolExecutor#getTask()

```

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.
        /**
         * 不执行任务的条件
         *
         * 线程池状态为STOP/TIDYING/TERMINATED
         * 线程池状态为SHUTDOWN且状态队列为空
         */
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            // 工作线程数-1
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        // Are workers subject to culling?
        /**
         * 判断是否执行超时策略，满足其一
         *
         * 设置了allowCoreThreadTimeOut=true
         * 工作线程超corePoolSize
         */
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

        /**
         * 回收工作线程
         *
         * 1.回收过多的：工作线程数> maximumPoolSize
         *   （可能是setMaximumPoolSize重新设置过最大线程数）
         * 2.回收超时的：timed && timedOut 回旋时工作线程获取任务超时
         *
         * 前提：工作线程>1 或者 （工作线程=1且队列为空）
         */
        if ((wc > maximumPoolSize || (timed && timedOut))
            && (wc > 1 || workQueue.isEmpty())) { // 工作线程>1 或者 （工作线程=1且队
            列为空），wc不会为0
                if (compareAndDecrementWorkerCount(c))
                    return null;
                continue;
            }

        try {
            Runnable r = timed ?
                // 以队列超时阻塞的方式实现线程超时回收
                workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                workQueue.take(); // 一直阻塞
            if (r != null)
                return r;
            timedOut = true;
        } catch (InterruptedException retry) { // poll or take超时前发生了中断
    }
}

```

```

        timedOut = false;
    }
}
}

```

工作线程退出

Worker线程会不停的从队列获取任务执行，如果队列中没有任务或者执行时发生异常，则执行

ThreadPoolExecutor#processWorkerExit(...)

```

private void processWorkerExit(Worker w, boolean completedAbruptly) {
    if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
        decrementWorkerCount();

    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        completedTaskCount += w.completedTasks;
        workers.remove(w);
    } finally {
        mainLock.unlock();
    }

    // 判断是否终止线程池
    tryTerminate();

    int c = ctl.get();
    if (runStateLessThan(c, STOP)) { // RUNNING/SHUTDOWN : 还可以执行任务的2种状态
        /**
         * 工作线程补充
         *
         * 1、正常退出
         * 2、异常退出
         */
        if (!completedAbruptly) { // worker是正常退出
            // 核心线程保留的最小数目
            int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
            // 如果设置了允许核心线程回收，任务队列非空，则保留1个线程
            if (min == 0 && !workQueue.isEmpty())
                min = 1;
            // 工作线程 >= min
            if (workerCountOf(c) >= min)
                return; // replacement not needed
        }
        // 异常退出或者工作线程不足，，则创建一个【非核心工作线程】
        addWorker(null, false);
    }
}

```

线程池关闭

```

public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();
        advanceRunState(SHUTDOWN); // RUNNING->SHUTDOWN
        interruptIdleworkers(); // 中断空闲线程
        onShutdown(); // hook for ScheduledThreadPoolExecutor
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
}

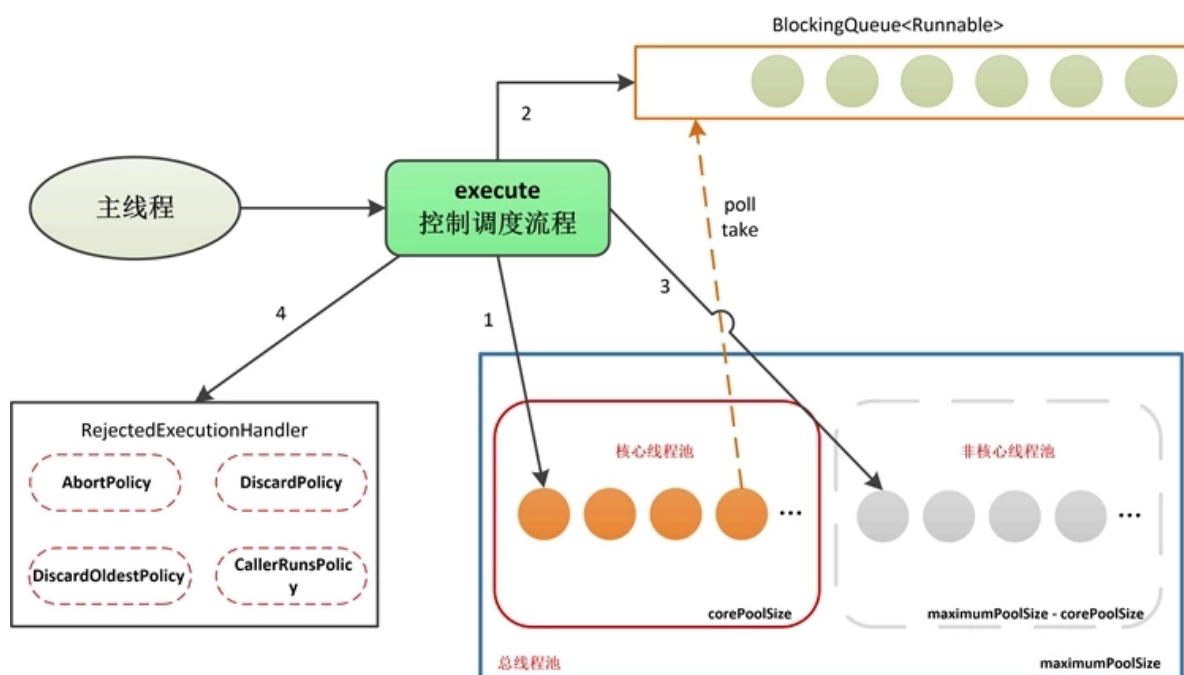
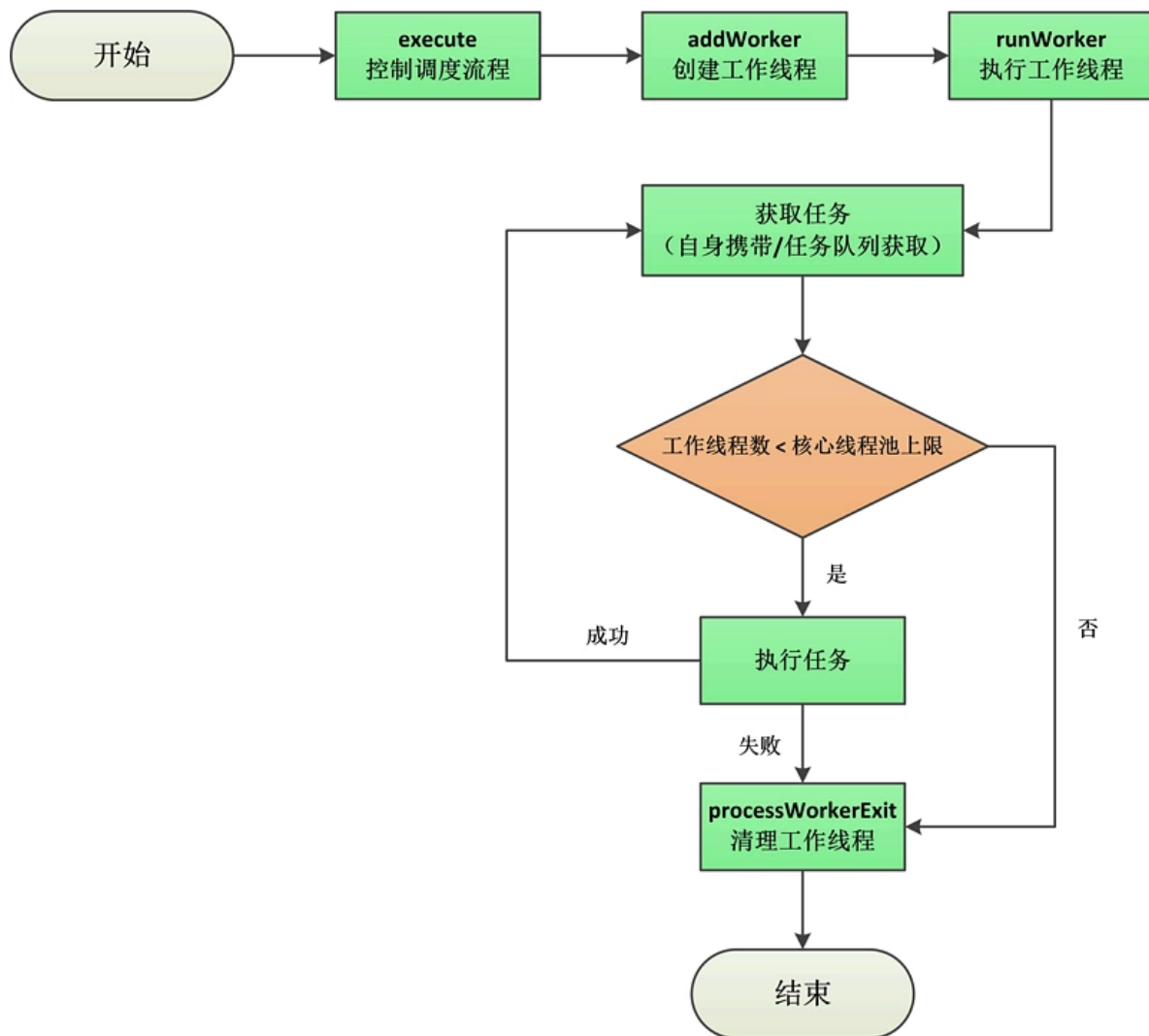
```

```

public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();
        advanceRunState(STOP); // RUNNING->STOP
        interruptworkers(); // 中断所有STARTED线程
        tasks = drainQueue(); // drainTo LIST
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
    return tasks;
}

```

执行流程总图



Executors

简单静态工厂，提供了**五类**创建Executor的方法

固定线程数的线程池

2个构造方法，返回ThreadPoolExecutor实例，ThreadPoolExecutor是ExecutorService的实现。

```
// 1.指定线程数，默认ThreadFactory
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}

// 2.指定ThreadFactory
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>(),
                                  threadFactory);
}
```

nThreads

线程数

threadFactory

创建线程的工厂类

ThreadFactory 是一个接口，目的是由外部统一创建线程

```
public interface ThreadFactory {
    /**
     * 构造一个线程，实现类可以初始化priority, name, daemon status, ThreadGroup等
     */
    Thread newThread(Runnable r);
}
```

DefaultThreadFactory 一个默认的thread factory，是Executors中的内部静态类

pool-x-thread-y

单个线程的线程池

2个构造方法，返回FinalizableDelegatedExecutorService，一个包装类对象

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

// 2.指定ThreadFactory
public static ExecutorService newSingleThreadExecutor(ThreadFactory
threadFactory) {
}
```

```

return new FinalizableDelegatedExecutorService
    (new ThreadPoolExecutor(1, 1,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(),
        threadFactory));
}

```

FinalizableDelegatedExecutorService继承自DelegatedExecutorService, DelegatedExecutorService是一个包装类, 实现了ExecutorService的所有方法, 但对方法的实现, 最终还是委托给构造函数代入的ExecutorService

```

static class DelegatedExecutorService extends AbstractExecutorService {
    private final ExecutorService e;
    DelegatedExecutorService(ExecutorService executor) { e = executor; }
    public void execute(Runnable command) { e.execute(command); }
    ...
}

```

包装类的作用是

屏蔽ThreadPoolExecutor中线程池设置方法

可缓存的线程池

设置keepalive参数, 实现线程可回收

```

/**
 * 初始0线程, 最大Integer.MAX_VALUE
 * 1. 适合执行耗时短的异步任务 (many short-lived asynchronous tasks)
 * 2. 60s未被使用, 从线程池中移除
 */
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
        60L, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>());
}
// 指定ThreadFactory
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
        60L, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>(),
        threadFactory);
}

```

可调度的线程池

返回ScheduledThreadPoolExecutor, 继承自ThreadPoolExecutor, 是ScheduledExecutorService接口的实现

```

/**
 * 1. 固定线程数
 * 2. 默认固定延迟执行
 */
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
{
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

public static ScheduledExecutorService newScheduledThreadPool(
    int corePoolSize, ThreadFactory threadFactory) {
    return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
}

```

ForkJoin线程池

返回ForkJoinPool, since 1.8

```

// 指定并行级别
public static ExecutorService newWorkStealingPool(int parallelism) {
    return new ForkJoinPool
        (parallelism,
         ForkJoinPool.defaultForkJoinWorkerThreadFactory,
         null, true);
}

// 默认并行级别=CPU核数
public static ExecutorService newWorkStealingPool() {
    return new ForkJoinPool
        (Runtime.getRuntime().availableProcessors(),
         ForkJoinPool.defaultForkJoinWorkerThreadFactory,
         null, true);
}

```