Executors框架

Executor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);
```

统筹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Runable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

```java
/**
 * 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();
```

```java
/**
 * 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();
```

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

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

```java
/**
 * 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;
```

```java
/**
 * 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);
```

```java
/**
 * 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);
```

```java
/**
 * 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);
```

```java
/**
 * 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;
```

```java
/**
 * 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;
```

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

```java
/**
 * 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

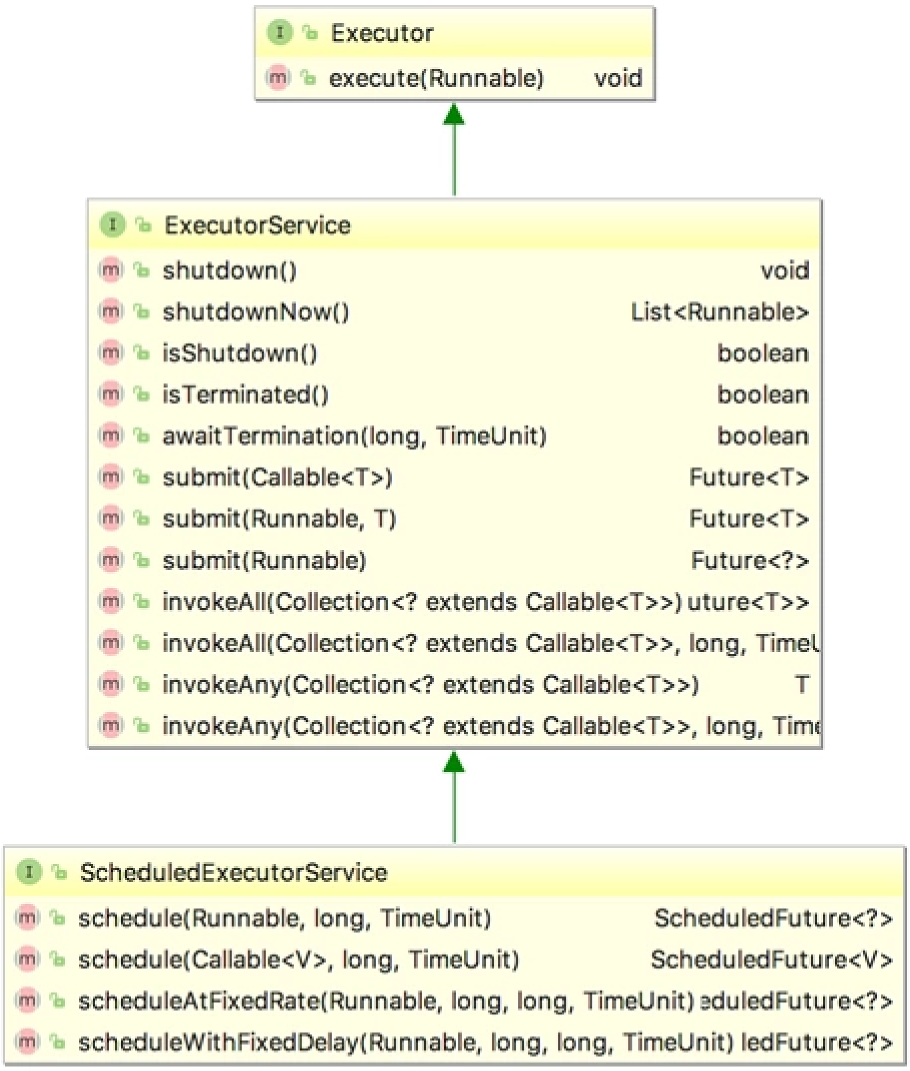
方法

```java
/**
 * 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);
```

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

```java
/**
 * 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=now()+initialDelay
 * 二次执行时间：F+period
 * N次执行时间：F+(N-1)*period
 */
ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                  long initialDelay,
                                                  long period,
                                                  TimeUnit unit);
```

```java
/**
 * 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=now()+initialDelay
 * 二次执行时间：F2=F+delay+F(首次执行时间)
 * N次执行时间：F2+delay+F(N-1执行时间)
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                     long initialDelay,
                                                     long delay,
                                                     TimeUnit unit);
```



↑关系图↑

Future

FutureTask的由来

- 思考

定义一个被执行的任务，可以实现**Runnable**

```
public class Task implements Runnable {
    @Override
    public void run() {
        // do something
    }
}
```

这种方式***不能获得***执行结果，run方法并没有返回值。

```
public interface Runnable {
    public abstract void run();
}
```

- 进阶

Callable定义了一个具有返回值的任务

```
public interface Callable<V> {
    V call() throws Exception;
}
```

所以，如果需要返回结果，定义的任务实现Callable接口，但是任务并不能直接交由Thread执行

- Future模式

J.U.C定义了RunnableFuture接口，是Runnable和Future的结合体，用于定义一个可以被线程或线程池执行，并且有返回值的任务

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}
```

Future作为一个异步任务的执行凭证，提供了对任务的控制及结果的获取方法

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

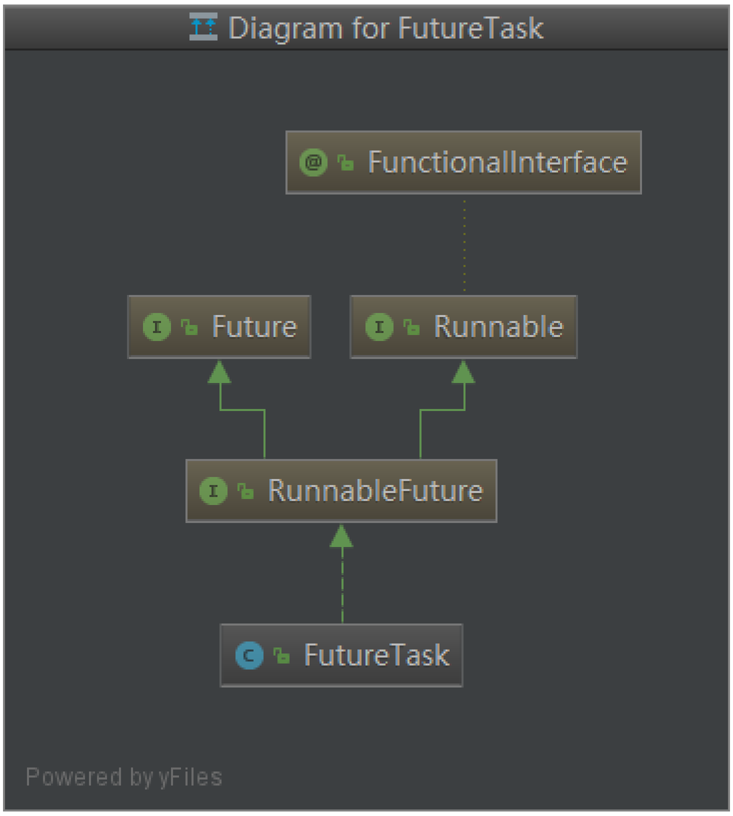
至此，已经有了RunnableFuture接口，完成了对思考问题的抽象。FutureTask作为实现，典型的构造器定义如下

```
public class FutureTask<V> implements RunnableFuture<V> {

    public FutureTask(Callable<V> callable) {
        if (callable == null)
            throw new NullPointerException();
        this.callable = callable;
        this.state = NEW;       // ensure visibility of callable
    }

    public FutureTask(Runnable runnable, V result) {
        // 将runnable包装为callable（RunnableAdapter···>Callable）
        this.callable = Executors.callable(runnable, result);
        this.state = NEW;       // ensure visibility of callable
    }
}
```

从两个构造方法看，最终都是将Runnable任务转换为Callable任务，多了一层RunnableAdapter用于适配

```
    // Executors#
    public static <T> Callable<T> callable(Runnable task, T result) {
        if (task == null)
```

```java
            throw new NullPointerException();
        return new RunnableAdapter<T>(task, result);
    }
    /**
     * A callable that runs given task and returns given result
     */
    static final class RunnableAdapter<T> implements Callable<T> {
        final Runnable task;
        final T result;
        RunnableAdapter(Runnable task, T result) {
            this.task = task;
            this.result = result;
        }
        public T call() {
            task.run();
            return result;
        }
    }
```

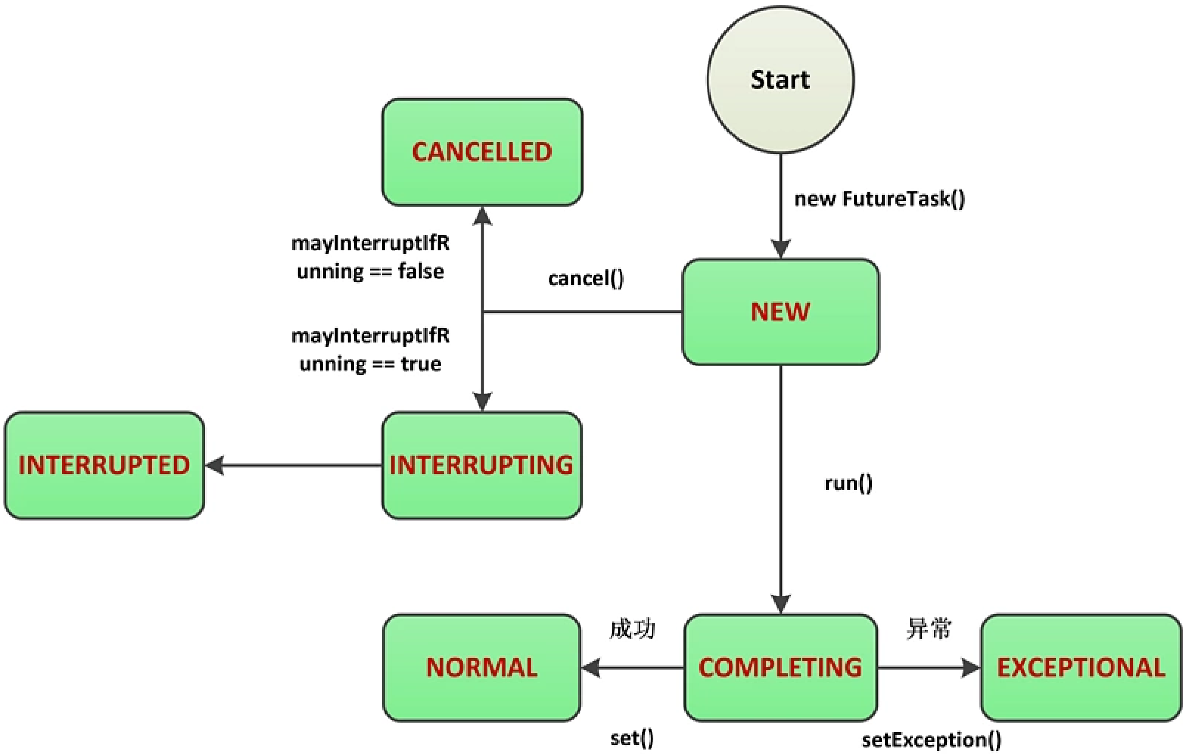


FutureTask的定义

- 状态定义

> 7 种
>
> - *NEW*：表示任务的初始化状态；
> - *COMPLETING*：表示任务已执行完成（正常完成或异常完成），但任务结果或异常原因还未设置完成，属于中间状态；
> - *NORMAL*：表示任务已经执行完成（正常完成），且任务结果已设置完成，属于最终状态；
> - *EXCEPTIONAL*：表示任务已经执行完成（异常完成），且任务异常已设置完成，属于最终状态；
> - *CANCELLED*：表示任务还没开始执行就被取消（非中断方式），属于最终状态；

> - *INTERRUPTING：* 表示任务还没开始执行就被取消（中断方式），正式被中断前的过渡状态，属于中间状态；
> - *INTERRUPTED：* 表示任务还没开始执行就被取消（中断方式），且已被中断，属于最终状态。
>
> 1. 仅在NEW时才可cancel
> 2. run后必定会经过 COMPLETING中间状态



- 其它字段

```
/** The underlying callable; nulled out after running */
// 真正的任务
private Callable<V> callable;
/** The result to return or exception to throw from get() */
private Object outcome; // non-volatile, protected by state reads/writes
/** The thread running the callable; CASed during run() */
private volatile Thread runner;
/** Treiber stack（无锁栈） of waiting threads */
// 将调用线程（等待结果的线程）包装成WaitNode
private volatile WaitNode waiters;

static final class WaitNode {
    volatile Thread thread;
    volatile WaitNode next;
    WaitNode() { thread = Thread.currentThread(); }
}
```

FutureTask的执行

- run()

先不谈Executor是如何创建线程并如何执行任务的，仅从FutureTask看起，因为FutureTask是实现了Runnable，所以，执行入口也是run()

```java
    public void run() {
        // 必须是NEW状态，且runner设置成功
        if (state != NEW ||
            !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                         null, Thread.currentThread()))
            return;
        try {
            Callable<V> c = callable;
            // 被执行的任务非空且状态为NEW
            if (c != null && state == NEW) {
                V result;
                boolean ran;
                try {
                    // [根本之处]调用到Callable的call方法
                    result = c.call();
                    ran = true;
                } catch (Throwable ex) {
                    result = null;
                    ran = false;
                    // 执行异常
                    setException(ex);
                }
                if (ran)
                    // 正常执行
                    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)
                // 被中断
                handlePossibleCancellationInterrupt(s);
        }
    }
```

略去set及setException方法，仅设置COMPLETING-->EXCEPTIONAL/NORMAL状态，最后调入finishCompletion()处理：**唤醒等待线程**

```java
    /**
     * Removes and signals all waiting threads, invokes done(), and
     * nulls out callable.
     */
    private void finishCompletion() {
        // assert state > COMPLETING;
        // 遍历等待线程
        for (WaitNode q; (q = waiters) != null;) {
            if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {
                for (;;) {
                    Thread t = q.thread;
                    if (t != null) {
                        q.thread = null;
                        // 唤醒
                        LockSupport.unpark(t);
```

```
                }
                // 其后线程
                WaitNode next = q.next;
                if (next == null)
                    break;
                q.next = null; // unlink to help gc
                q = next;
            }
            break;
        }
    }

    done(); // 钩子

    callable = null;        // to reduce footprint
}
```

- cancel()

```
// mayInterruptIfRunning 是否中断正在执行的任务，false则只变更状态为CANCELLED
public boolean cancel(boolean mayInterruptIfRunning) {
    // 必须是NEW状态才可以取消任务
    if (!(state == NEW &&
          UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
                                   mayInterruptIfRunning ? INTERRUPTING :
CANCELLED)))
        return false;
    try {    // in case call to interrupt throws exception
        if (mayInterruptIfRunning) {
            try {
                Thread t = runner;
                if (t != null)
                    t.interrupt();
            } finally { // final state
                UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
            }
        }
    } finally {
        finishCompletion();
    }
    return true;
}
```

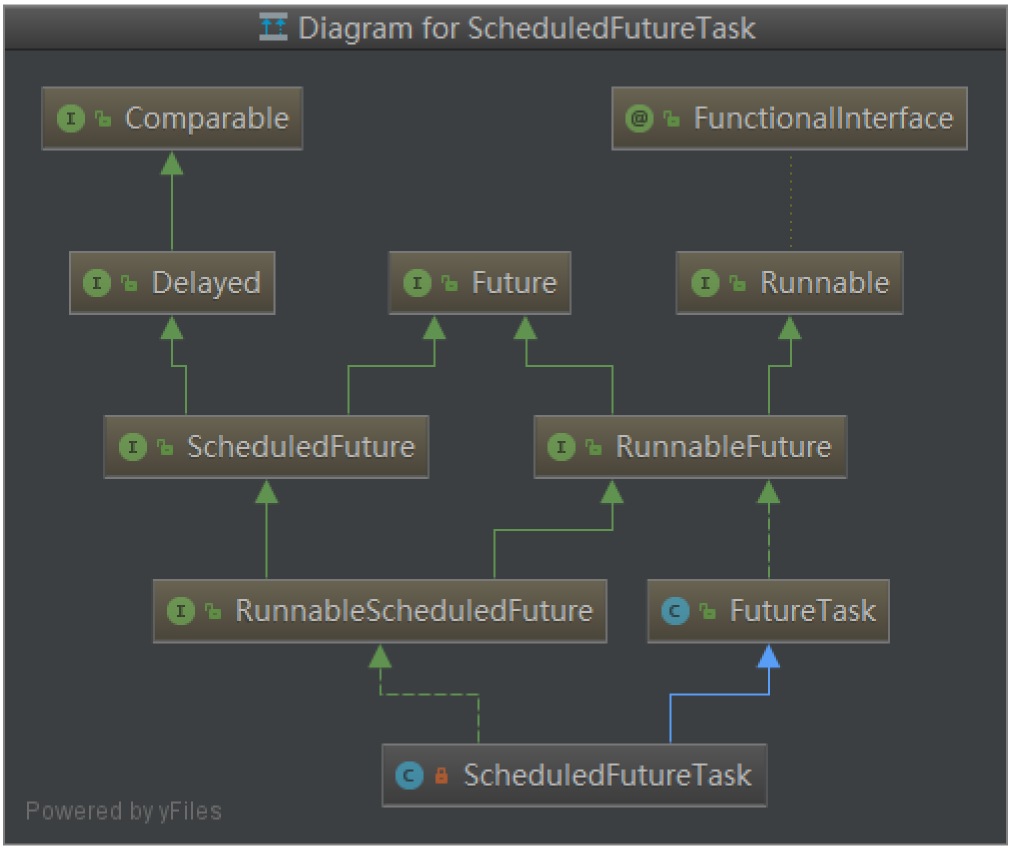
FutureTask获取结果

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

public V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException {
    if (unit == null)
```

```java
        throw new NullPointerException();
    int s = state;
    // 等待时间到后还未执行完成
    if (s <= COMPLETING &&
        (s = awaitDone(true, unit.toNanos(timeout))) <= COMPLETING)
        throw new TimeoutException();
    return report(s);
}

private int awaitDone(boolean timed, long nanos)
    throws InterruptedException {
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    WaitNode q = null;
    boolean queued = false;
    for (;;) {
        // 等待结果的线程被中断
        if (Thread.interrupted()) {
            removeWaiter(q);
            throw new InterruptedException();
        }

        int s = state;
        if (s > COMPLETING) { // 已完成或异常
            if (q != null)
                q.thread = null;
            return s;
        }
        // 临界状态-刚完成 让出线程资源
        else if (s == COMPLETING) // cannot time out yet
            Thread.yield();
        else if (q == null)
            // 还在执行 创建等待节点（还未入栈）
            q = new WaitNode();
        else if (!queued)
            // 接上 自旋后入栈等待
            queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                 q.next = waiters, q);
        else if (timed) { // 设置了超时时间
            nanos = deadline - System.nanoTime();
            if (nanos <= 0L) { // 已超时
                removeWaiter(q);
                return state;
            }
            LockSupport.parkNanos(this, nanos);
        }
        else
            // 在finishCompletion唤醒
            LockSupport.park(this);
    }
}
```

ScheduledFutureTask

继承FutureTask和Delayed接口来实现周期/延迟功能



ScheduledFutureTask的**字段**及**构造器**定义

```
private class ScheduledFutureTask<V>
    extends FutureTask<V> implements RunnableScheduledFuture<V> {

    /** Sequence number to break ties FIFO */
    private final long sequenceNumber;

    /** The time the task is enabled to execute in nanoTime units */
    // 允许被执行的时间-首次执行时间
    private long time;

    /**
     * 正数：fixed-rate
     * 负数：fixed-delay
     * 0：非周期任务
     */
    private final long period;

    /** The actual task to be re-enqueued by reExecutePeriodic */
    RunnableScheduledFuture<V> outerTask = this;

    /**
     * Index into delay queue, to support faster cancellation.
     */
    // 在延迟队列的索引  支持快速取消
    int heapIndex;

    /**
     * Creates a one-shot action with given nanoTime-based trigger time.
     */
    ScheduledFutureTask(Runnable r, V result, long ns) {
        super(r, result);
```

```java
        this.time = ns;
        this.period = 0;
        this.sequenceNumber = sequencer.getAndIncrement();
    }

    /**
     * Creates a periodic action with given nano time and period.
     */
    ScheduledFutureTask(Runnable r, V result, long ns, long period) {
        super(r, result);
        this.time = ns;
        this.period = period;
        this.sequenceNumber = sequencer.getAndIncrement();
    }

    /**
     * Creates a one-shot action with given nanoTime-based trigger time.
     */
    ScheduledFutureTask(Callable<V> callable, long ns) {
        super(callable);
        this.time = ns;
        this.period = 0;
        this.sequenceNumber = sequencer.getAndIncrement();
    }
    ...
}
```

线程池Executor

ThreadPoolExecutor

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

构造器

```java
/**
 * @param corePoolSize 核心线程数，在线程池中一直保持存在，即便是空闲，除非设置了
 *        allowCoreThreadTimeOut
 * @param maximumPoolSize 允许创建线程的最大数量
 * @param keepAliveTime 线程数大于corePoolSize时，多余空闲线程的存活时间
 * @param workQueue 任务队列，保存已提交还未执行的任务
 * @param handler 拒绝策略
 */
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
```

```java
            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

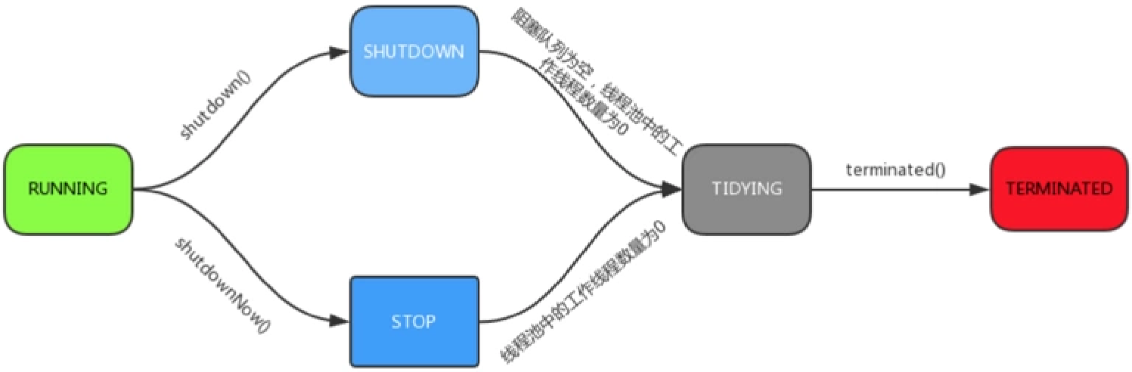
线程池状态

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**：

```java
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) {
            }
        }
    }
}
```

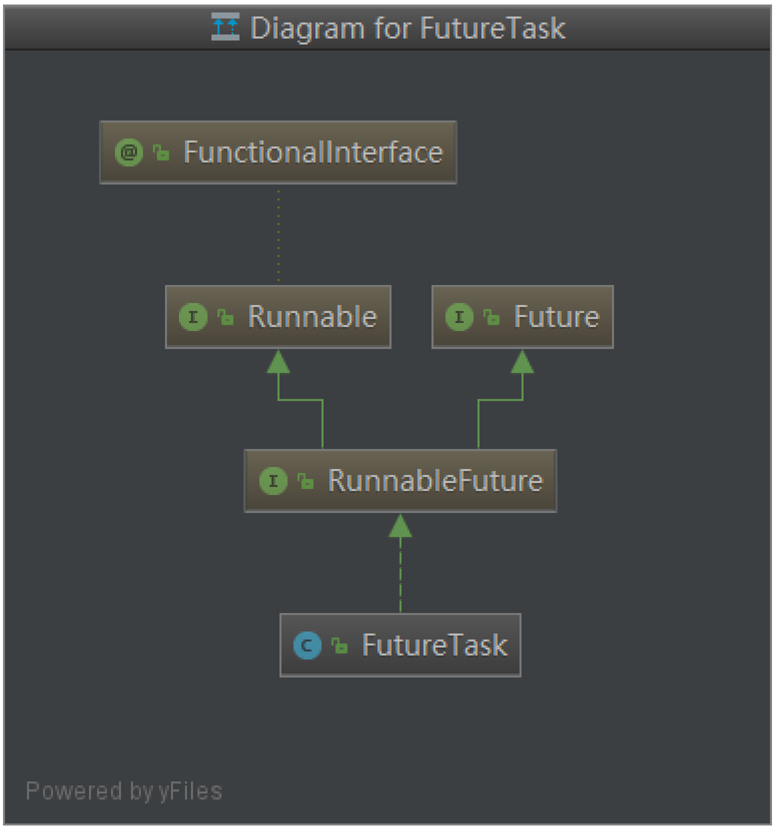
任务提交

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
        }
    }
```

```java
    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 IllegalThreadStateException();
                    // 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(...)

```java
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()

```java
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(...)

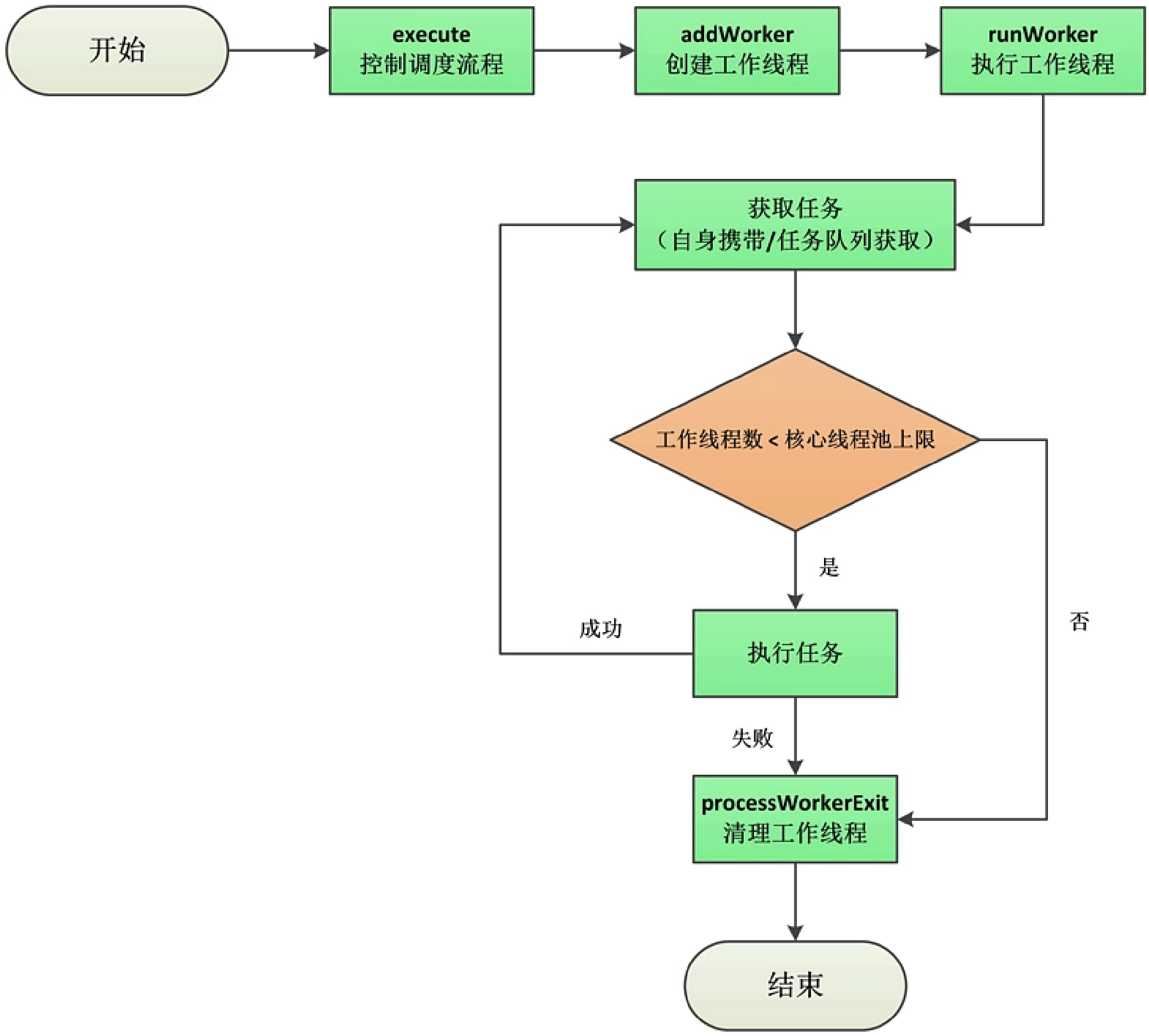
```java
private void processWorkerExit(Worker w, boolean completedAbruptly) {
    if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
        decrementWorkerCount();

    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

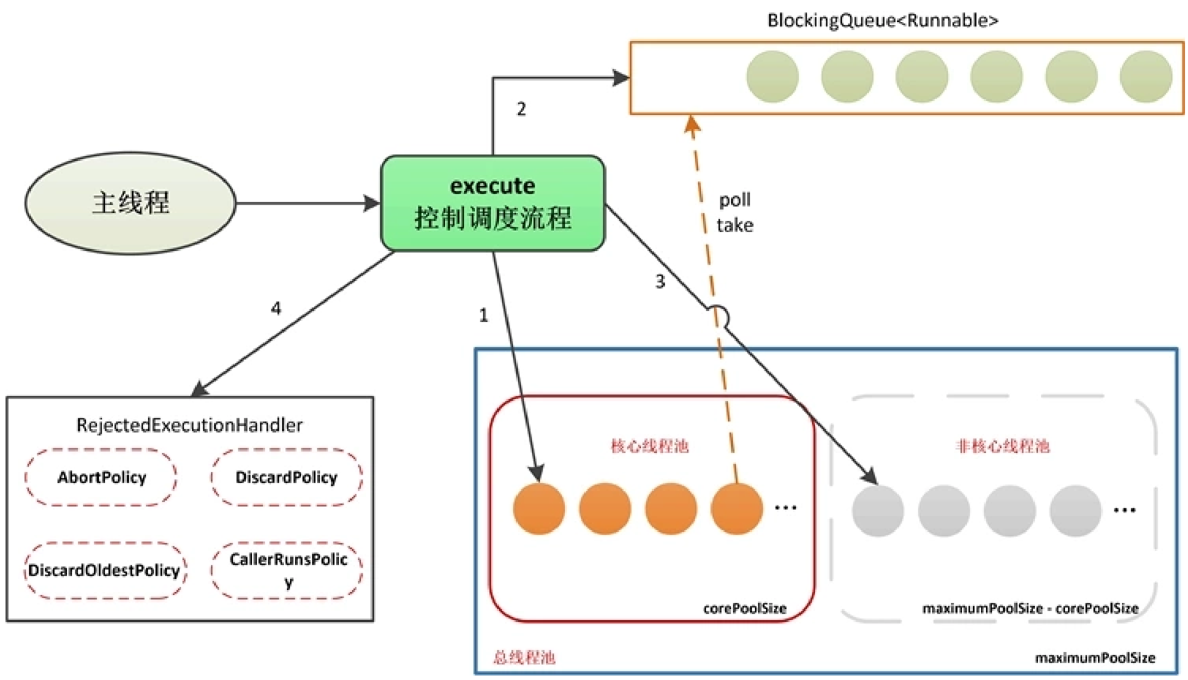
```java
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();
        advanceRunState(SHUTDOWN);// RUNNING->SHUTDOWN
        interruptIdleWorkers(); // 中断空闲线程
        onShutdown(); // hook for ScheduledThreadPoolExecutor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

```java
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



↑ 调度执行流程 ↑



↑ ThreadPoolExecutor核心组件：阻塞队列、核心线程池、拒绝策略，关系图 ↑

执行链

AbstractExecutorService.submit(..) --> ThreadPoolExecutor.execute(FutureTask task) --> addWorker(task) --> worker.start --> worker.run() --> runWorker(worker) --> task.run()

ScheduledThreadPoolExecutor

执行链

ScheduledThreadPoolExecutor.schedule --> delayedExecute(ScheduledFutureTask task) --> workQueue.add(task) --> addWorker(task::null) --> worker.start --> worker.run() --> runWorker(worker) --> task.run() --> FutureTask.run()/runAndReset() --> reExecutePeriodic(task)

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

```java
// 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

```java
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

```java
/**
 * 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);
}
```