ThreadLocal

源码

```java
package java.lang;
import jdk.internal.misc.TerminatingThreadLocal;

import java.lang.ref.WeakReference;
import java.util.Objects;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.Supplier;
public class ThreadLocal<T> {
    private final int threadLocalHashCode = nextHashCode();
    private static AtomicInteger nextHashCode = new AtomicInteger();
    private static final int HASH_INCREMENT = 0x61c88647;
    private static int nextHashCode() {
        return nextHashCode.getAndAdd(HASH_INCREMENT);
    }
    /*
        初始化方法
    */
    protected T initialValue() {
        return null;
    }
    /*
        初始化方法，提供一个有返回值 没有入参的supplier
    */
    public static <S> ThreadLocal<S> withInitial(Supplier<? extends S>
supplier) {
        return new SuppliedThreadLocal<>(supplier);
    }

    public ThreadLocal() {
    }
    /**
        1.getMap(t)方法获取当前线程的ThreadLocalMap
        2.如果map为空，执行setInitialValue初始化方法
        3.不为空，就根据当前对象(threadLocal对象) 获取Entry的value
        key是threadLocal变量(对象)，value是存的对象的值// 这里要区分 值 跟
threadLocal对象不是一个东西，看下面的使用范例就知道了，value是别的自己真实使用的对象。
因此创建ThreadLocal对象的时候，从不用new ThreadLocal()
    */
    public T get() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this);
            if (e != null) {
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }

    boolean isPresent() {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        return map != null && map.getEntry(this) != null;
    }

    /*
        1.执行initialValue方法，获取value,该方法默认返回null，需要用户重写
        2.获取当前线程的ThreadLocalMap对象
        3.如果map不为空，就重新覆盖value，否则创建map，用当前的value
    */
    private T setInitialValue() {
        T value = initialValue();
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            map.set(this, value);
        } else {
            createMap(t, value);
        }
        if (this instanceof TerminatingThreadLocal) {
            TerminatingThreadLocal.register((TerminatingThreadLocal<?>)
this);
        }
        return value;
    }

    /**
        跟setInitialValue差不多
        如果map不为空，就重新覆盖value，否则创建map，用当前的value
     */
    public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null) {
            map.set(this, value);
        } else {
            createMap(t, value);
        }
    }
    /*

    */
     public void remove() {
         ThreadLocalMap m = getMap(Thread.currentThread());
         if (m != null) {
             m.remove(this);
         }
     }

    /*
```

```java
        返回当前线程的threadLocals
    */
    ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }

    /*
        构造方法创建一个ThreadLocalMap 并给Thread的threadLocals赋值
    */
    void createMap(Thread t, T firstValue) {
        t.threadLocals = new ThreadLocalMap(this, firstValue);
    }


    static ThreadLocalMap createInheritedMap(ThreadLocalMap parentMap) {
        return new ThreadLocalMap(parentMap);
    }


    T childValue(T parentValue) {
        throw new UnsupportedOperationException();
    }


    static final class SuppliedThreadLocal<T> extends ThreadLocal<T> {

        private final Supplier<? extends T> supplier;

        SuppliedThreadLocal(Supplier<? extends T> supplier) {
            this.supplier = Objects.requireNonNull(supplier);
        }

        @Override
        protected T initialValue() {
            return supplier.get();
        }
    }

    //  静态内部类ThreadLocalMap
    static class ThreadLocalMap {
        //  静态内部类Entry,   value是存放的值，key是ThreadLocal对象，所以一个线程可以有多个threadLocal变量(对象)
        static class Entry extends WeakReference<ThreadLocal<?>> {
            /** The value associated with this ThreadLocal. */
            Object value;
            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }
        private static final int INITIAL_CAPACITY = 16;
        private Entry[] table;
        private int size = 0;
        private int threshold; // Default to 0
        private void setThreshold(int len) {
            threshold = len * 2 / 3;
        }
        private static int nextIndex(int i, int len) {
            return ((i + 1 < len) ? i + 1 : 0);
        }
        private static int prevIndex(int i, int len) {
            return ((i - 1 >= 0) ? i - 1 : len - 1);
        }
        ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
            table = new Entry[INITIAL_CAPACITY];
            int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
            table[i] = new Entry(firstKey, firstValue);
            size = 1;
            setThreshold(INITIAL_CAPACITY);
        }
        private ThreadLocalMap(ThreadLocalMap parentMap) {
            Entry[] parentTable = parentMap.table;
            int len = parentTable.length;
            setThreshold(len);
            table = new Entry[len];

            for (Entry e : parentTable) {
                if (e != null) {
                    @SuppressWarnings("unchecked")
                    ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();
                    if (key != null) {
                        Object value = key.childValue(e.value);
                        Entry c = new Entry(key, value);
                        int h = key.threadLocalHashCode & (len - 1);
                        while (table[h] != null)
                            h = nextIndex(h, len);
                        table[h] = c;
                        size++;
                    }
                }
            }
        }
        //get方法
        private Entry getEntry(ThreadLocal<?> key) {
            int i = key.threadLocalHashCode & (table.length - 1);
            Entry e = table[i];
            if (e != null && e.refersTo(key))
                return e;
            else
                return getEntryAfterMiss(key, i, e);
        }
        private Entry getEntryAfterMiss(ThreadLocal<?> key, int i, Entry e) {
            Entry[] tab = table;
            int len = tab.length;

            while (e != null) {
                if (e.refersTo(key))
                    return e;
                if (e.refersTo(null))
                    expungeStaleEntry(i);
                else
                    i = nextIndex(i, len);
                e = tab[i];
            }
            return null;
        }
        //set方法
        private void set(ThreadLocal<?> key, Object value) {
            Entry[] tab = table;
            int len = tab.length;
            int i = key.threadLocalHashCode & (len-1);
            for (Entry e = tab[i];
                 e != null;
                 e = tab[i = nextIndex(i, len)]) {
                if (e.refersTo(key)) {
                    e.value = value;
                    return;
                }

                if (e.refersTo(null)) {
                    replaceStaleEntry(key, value, i);
                    return;
                }
            }

            tab[i] = new Entry(key, value);
            int sz = ++size;
            if (!cleanSomeSlots(i, sz) && sz >= threshold)
                rehash();
        }
        //删除
        private void remove(ThreadLocal<?> key) {
            Entry[] tab = table;
            int len = tab.length;
            int i = key.threadLocalHashCode & (len-1);
            for (Entry e = tab[i];
                 e != null;
                 e = tab[i = nextIndex(i, len)]) {
                if (e.refersTo(key)) {
                    e.clear();
                    expungeStaleEntry(i);
                    return;
                }
            }
        }

        private void replaceStaleEntry(ThreadLocal<?> key, Object value,
                                       int staleSlot) {
            Entry[] tab = table;
            int len = tab.length;
            Entry e;
            int slotToExpunge = staleSlot;
            for (int i = prevIndex(staleSlot, len);
                 (e = tab[i]) != null;
                 i = prevIndex(i, len))
                if (e.refersTo(null))
                    slotToExpunge = i;
            for (int i = nextIndex(staleSlot, len);
                 (e = tab[i]) != null;
                 i = nextIndex(i, len)) {
                if (e.refersTo(key)) {
                    e.value = value;

                    tab[i] = tab[staleSlot];
                    tab[staleSlot] = e;
                    if (slotToExpunge == staleSlot)
                        slotToExpunge = i;
                    cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
                    return;
                }

                if (e.refersTo(null) && slotToExpunge == staleSlot)
                    slotToExpunge = i;
            }

            // If key not found, put new entry in stale slot
            tab[staleSlot].value = null;
            tab[staleSlot] = new Entry(key, value);

            // If there are any other stale entries in run, expunge them
            if (slotToExpunge != staleSlot)
                cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
        }
        private int expungeStaleEntry(int staleSlot) {
            Entry[] tab = table;
            int len = tab.length;

            // expunge entry at staleSlot
            tab[staleSlot].value = null;
            tab[staleSlot] = null;
            size--;

            // Rehash until we encounter null
            Entry e;
            int i;
            for (i = nextIndex(staleSlot, len);
                 (e = tab[i]) != null;
                 i = nextIndex(i, len)) {
                ThreadLocal<?> k = e.get();
                if (k == null) {
                    e.value = null;
                    tab[i] = null;
                    size--;
                } else {
                    int h = k.threadLocalHashCode & (len - 1);
                    if (h != i) {
                        tab[i] = null;

```

```java
                        // Unlike Knuth 6.4 Algorithm R, we must scan until
                        // null because multiple entries could have been stale.
                        while (tab[h] != null)
                            h = nextIndex(h, len);
                        tab[h] = e;
                    }
                }
            }
            return i;
        }

        private boolean cleanSomeSlots(int i, int n) {
            boolean removed = false;
            Entry[] tab = table;
            int len = tab.length;
            do {
                i = nextIndex(i, len);
                Entry e = tab[i];
                if (e != null && e.refersTo(null)) {
                    n = len;
                    removed = true;
                    i = expungeStaleEntry(i);
                }
            } while ( (n >>>= 1) != 0);
            return removed;
        }

        private void rehash() {
            expungeStaleEntries();
            if (size >= threshold - threshold / 4)
                resize();
        }

        //扩容，2倍
        private void resize() {
            Entry[] oldTab = table;
            int oldLen = oldTab.length;
            int newLen = oldLen * 2;
            Entry[] newTab = new Entry[newLen];
            int count = 0;

            for (Entry e : oldTab) {
                if (e != null) {
                    ThreadLocal<?> k = e.get();
                    if (k == null) {
                        e.value = null; // Help the GC
                    } else {
                        int h = k.threadLocalHashCode & (newLen - 1);
                        while (newTab[h] != null)
                            h = nextIndex(h, newLen);
                        newTab[h] = e;
                        count++;
                    }
                }
            }

            setThreshold(newLen);
            size = count;
            table = newTab;
        }
        private void expungeStaleEntries() {
            Entry[] tab = table;
            int len = tab.length;
            for (int j = 0; j < len; j++) {
                Entry e = tab[j];
                if (e != null && e.refersTo(null))
                    expungeStaleEntry(j);
            }
        }
    }
}

```

范例

```
public static final ThreadLocal<DateFormat> threadLocal = new
ThreadLocal<DateFormat>(){
    @Override
    protected DateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};
```

```
public class TestThreadLocal implements Runnable{
    // SimpleDateFormat 不是线程安全的，所以每个线程都要有自己独立的副本
    /**
     * formatter就是我们竞争的资源，这里每个线程都有它的副本
     */
    private static final ThreadLocal<SimpleDateFormat> formatter =
ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd HHmm"));
    public static void main(String[] args) throws InterruptedException {
        TestThreadLocal obj = new TestThreadLocal();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }
    @Override
    public void run() {
        System.out.println("Thread Name=
"+Thread.currentThread().getName()+" default Formatter =
"+formatter.get().toPattern());
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

ThreadPoolExecutor

要将task和thread区分开，task任务，thread 线程

常量

```
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));//记录workCount
    private static final int COUNT_BITS = Integer.SIZE - 3;
    private static final int COUNT_MASK = (1 << COUNT_BITS) - 1;
    // 线程池的运行状态
    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 & ~COUNT_MASK; }
    private static int workerCountOf(int c)  { return c & COUNT_MASK; }
    private static int ctlOf(int rs, int wc) { return rs | wc; }

    private static boolean runStateLessThan(int c, int s) {
        return c < s;
    }
    private static boolean runStateAtLeast(int c, int s) {
        return c >= s;
    }
    private static boolean isRunning(int c) {
        return c < SHUTDOWN;
    }
    /**
     * Attempts to CAS-increment the workerCount field of ctl.
     */
    private boolean compareAndIncrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect + 1);
    }

    /**
     * Attempts to CAS-decrement the workerCount field of ctl.
     */
    private boolean compareAndDecrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect - 1);
    }
```

```java

    private void decrementWorkerCount() {
        ctl.addAndGet(-1);
    }

    //阻塞队列
    private final BlockingQueue<Runnable> workQueue;
    //可重入锁
    private final ReentrantLock mainLock = new ReentrantLock();

    /**
     * 线程池中所有的worker,只有获取到锁之后，才能操作
     */
    private final HashSet<java.util.concurrent.ThreadPoolExecutor.Worker> workers = new HashSet<>();

    /**
     * Wait condition to support awaitTermination.
     */
    private final Condition termination = mainLock.newCondition();

    /**
     * Tracks largest attained pool size. Accessed only under
     * mainLock.
     */
    private int largestPoolSize;
    private long completedTaskCount;//计数 所有完成的任务，只有worker停止时候更新，获取mainLock时候 才能写

    /**
     * 创建线程的工厂，通过addWorker方法
     */
    private volatile ThreadFactory threadFactory;
    private volatile RejectedExecutionHandler handler;
    private volatile long keepAliveTime;//线程存活时间
    private volatile boolean allowCoreThreadTimeOut;
    private volatile int corePoolSize;//核心线程数
    private volatile int maximumPoolSize;//最大线程数
    private static final RejectedExecutionHandler defaultHandler =  //默认的拒绝策略
            new java.util.concurrent.ThreadPoolExecutor.AbortPolicy();
    private static final RuntimePermission shutdownPerm =
            new RuntimePermission("modifyThread");
```

Worker内部类

```java
//继承了AbstractQueuedSynchronizer， 就有了state变量和阻塞队列
private final class Worker extends AbstractQueuedSynchronizer implements Runnable{
    private static final long serialVersionUID = 6138294804551838833L;

    //聚合Thread对象
    final Thread thread;
    //Runnable类型的task
    Runnable firstTask;
    /** Per-thread task counter */
    volatile long completedTasks;
    Worker(Runnable firstTask) {
        setState(-1); // inhibit interrupts until runWorker
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);//这里很重要：this指向自己，thread就是由自己创建的，这样在start的时候，调用的就是自己的run方法
    }
    //自己的run方法
    public void run() {
        runWorker(this);
    }
    //是否独占
    protected boolean isHeldExclusively() {
        return getState() != 0;
    }
    //获取锁
    protected boolean tryAcquire(int unused) {
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }
    //释放锁
    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) {
            }
        }
    }
}
```

execute方法:整体流程

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {如果运行中的线程少于核心线程数，就开启新线程
        if (addWorker(command, true))//true 表示核心线程
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {//加入到阻塞队列
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))//判断线程是否是运行状态，如果否，就移除线程
            reject(command);//拒绝任务
        else if (workerCountOf(recheck) == 0)//如果任务数为0，就添加一个空转
            addWorker(null, false);//false表示最大线程
    }
    else if (!addWorker(command, false))//添加线程(占用最大线程数)
        reject(command);
}

```

添加和执行worker

addworker

runworker

```
    final void reject(Runnable command) {
        handler.rejectedExecution(command, this);
    }
    void onShutdown() {
    }
    private List<Runnable> drainQueue() {
        BlockingQueue<Runnable> q = workQueue;
        ArrayList<Runnable> taskList = new ArrayList<>();
        q.drainTo(taskList);
        if (!q.isEmpty()) {
            for (Runnable r : q.toArray(new Runnable[0])) {
                if (q.remove(r))
                    taskList.add(r);
            }
        }
        return taskList;
    }
    //核心方法
    private boolean addWorker(Runnable firstTask, boolean core) {
        retry:
        for (int c = ctl.get();;) {
            // Check if queue empty only if necessary.
            if (runStateAtLeast(c, SHUTDOWN)
                && (runStateAtLeast(c, STOP)
                    || firstTask != null
                    || workQueue.isEmpty()))
```

```java
                return false;

            for (;;) {
                if (workerCountOf(c)
                    >= ((core ? corePoolSize : maximumPoolSize) & COUNT_MASK))
                    return false;//如果是核心线程又大于核心线程数，或者不是核心线程，又大于最大线程，就返回false
                if (compareAndIncrementWorkerCount(c))//计数workcount增加成功，就跳出循环
                    break retry;
                c = ctl.get();  // Re-read ctl
                if (runStateAtLeast(c, SHUTDOWN))
                    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);//创建一个worker
            final Thread t = w.thread;//这个threa是threadFactory创建的
            if (t != null) {
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {
                    int c = ctl.get();

                    if (isRunning(c) ||
                        (runStateLessThan(c, STOP) && firstTask == null)) {
                        if (t.getState() != Thread.State.NEW)
                            throw new IllegalThreadStateException();
                        workers.add(w);//添加worker
                        workerAdded = true;
                        int s = workers.size();
                        if (s > largestPoolSize)
                            largestPoolSize = s;
                    }
                } finally {
                    mainLock.unlock();
                }
                if (workerAdded) {
                    t.start();//执行线程,worker中的thread是可以直接start的
                    workerStarted = true;
                }
            }
        } finally {
            if (! workerStarted)
                addWorkerFailed(w);
        }
        return workerStarted;
    }

    private void addWorkerFailed(Worker w) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            if (w != null)
                workers.remove(w);
            decrementWorkerCount();
            tryTerminate();
        } finally {
            mainLock.unlock();
        }
    }
    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();
        }

        tryTerminate();

        int c = ctl.get();
        if (runStateLessThan(c, STOP)) {
            if (!completedAbruptly) {
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                if (min == 0 && ! workQueue.isEmpty())
                    min = 1;
                if (workerCountOf(c) >= min)
                    return; // replacement not needed
            }
            addWorker(null, false);
        }
    }

    /**
     * Performs blocking or timed wait for a task, depending on
     * current configuration settings, or returns null if this worker
     * must exit because of any of:
     * 1. There are more than maximumPoolSize workers (due to
     *    a call to setMaximumPoolSize).
     * 2. The pool is stopped.
     * 3. The pool is shutdown and the queue is empty.
     * 4. This worker timed out waiting for a task, and timed-out
     *    workers are subject to termination (that is,
     *    {@code allowCoreThreadTimeOut || workerCount > corePoolSize})
     *    both before and after the timed wait, and if the queue is
     *    non-empty, this worker is not the last thread in the pool.
     *
     * @return task, or null if the worker must exit, in which case
     *         workerCount is decremented
     */
    private Runnable getTask() {
        boolean timedOut = false; // Did the last poll() time out?

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

            // Check if queue empty only if necessary.
            if (runStateAtLeast(c, SHUTDOWN)
                && (runStateAtLeast(c, 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;
            }
        }
    }

    final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;//置空firstTask。
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
       try {
            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);//扩展前置方法
                    try {
                        task.run();//调用的其实是task自己的run方法
                        afterExecute(task, null);//扩展后置方法
                    } catch (Throwable ex) {
                        afterExecute(task, ex);
                        throw ex;
                    }
                } finally {
                    task = null;
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            processWorkerExit(w, completedAbruptly);
        }
    }
```

操作线程池的方法

```java
/**
 * Initiates an orderly shutdown in which previously submitted
 * tasks are executed, but no new tasks will be accepted.
 * Invocation has no additional effect if already shut down.
 *
 * <p>This method does not wait for previously submitted tasks to
 * complete execution.  Use {@link #awaitTermination awaitTermination}
 * to do that.
 *
 * @throws SecurityException {@inheritDoc}
 */
public void shutdown() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();
        advanceRunState(SHUTDOWN);
        interruptIdleWorkers();
        onShutdown(); // hook for ScheduledThreadPoolExecutor
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
}

/**
```

```java
 * Attempts to stop all actively executing tasks, halts the
 * processing of waiting tasks, and returns a list of the tasks
 * that were awaiting execution. These tasks are drained (removed)
 * from the task queue upon return from this method.
 *
 * <p>This method does not wait for actively executing tasks to
 * terminate.  Use {@link #awaitTermination awaitTermination} to
 * do that.
 *
 * <p>There are no guarantees beyond best-effort attempts to stop
 * processing actively executing tasks.  This implementation
 * interrupts tasks via {@link Thread#interrupt}; any task that
 * fails to respond to interrupts may never terminate.
 *
 * @throws SecurityException {@inheritDoc}
 */
public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();
        advanceRunState(STOP);
        interruptWorkers();
        tasks = drainQueue();
    } finally {
        mainLock.unlock();
    }
    tryTerminate();
    return tasks;
}

public boolean isShutdown() {
    return runStateAtLeast(ctl.get(), SHUTDOWN);
}

/** Used by ScheduledThreadPoolExecutor. */
boolean isStopped() {
    return runStateAtLeast(ctl.get(), STOP);
}

/**
 * Returns true if this executor is in the process of terminating
 * after {@link #shutdown} or {@link #shutdownNow} but has not
 * completely terminated.  This method may be useful for
 * debugging. A return of {@code true} reported a sufficient
 * period after shutdown may indicate that submitted tasks have
 * ignored or suppressed interruption, causing this executor not
 * to properly terminate.
 *
 * @return {@code true} if terminating but not yet terminated
 */
public boolean isTerminating() {
    int c = ctl.get();
    return runStateAtLeast(c, SHUTDOWN) && runStateLessThan(c, TERMINATED);
}

public boolean isTerminated() {
    return runStateAtLeast(ctl.get(), TERMINATED);
}

public boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException {
    long nanos = unit.toNanos(timeout);
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        while (runStateLessThan(ctl.get(), TERMINATED)) {
            if (nanos <= 0L)
                return false;
            nanos = termination.awaitNanos(nanos);
        }
        return true;
    } finally {
        mainLock.unlock();
    }
}
```

获取线程池统计信息

```java
//正在执行的任务数
public int getActiveCount() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        int n = 0;
        for (Worker w : workers)
            if (w.isLocked())
                ++n;
        return n;
    } finally {
        mainLock.unlock();
    }
}

//曾经达到的最大的线程数
public int getLargestPoolSize() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        return largestPoolSize;
    } finally {
        mainLock.unlock();
    }
}
//所有处理过的任务数
```

```java
public long getTaskCount() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        long n = completedTaskCount;
        for (Worker w : workers) {
            n += w.completedTasks;
            if (w.isLocked())
                ++n;
        }
        return n + workQueue.size();
    } finally {
        mainLock.unlock();
    }
}

//成功处理过的任务
public long getCompletedTaskCount() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        long n = completedTaskCount;
        for (Worker w : workers)
            n += w.completedTasks;
        return n;
    } finally {
        mainLock.unlock();
    }
}

```

扩展钩子

```java

/* Extension hooks */

/**
 * Method invoked prior to executing the given Runnable in the
 * given thread.  This method is invoked by thread {@code t} that
 * will execute task {@code r}, and may be used to re-initialize
 * ThreadLocals, or to perform logging.
 *
 * <p>This implementation does nothing, but may be customized in
 * subclasses. Note: To properly nest multiple overridings, subclasses
 * should generally invoke {@code super.beforeExecute} at the end of
 * this method.
 *
 * @param t the thread that will run task {@code r}
 * @param r the task that will be executed
 */
protected void beforeExecute(Thread t, Runnable r) { }

/**
```

```java
 * Method invoked upon completion of execution of the given Runnable.
 * This method is invoked by the thread that executed the task. If
 * non-null, the Throwable is the uncaught {@code RuntimeException}
 * or {@code Error} that caused execution to terminate abruptly.
 *
 * <p>This implementation does nothing, but may be customized in
 * subclasses. Note: To properly nest multiple overridings, subclasses
 * should generally invoke {@code super.afterExecute} at the
 * beginning of this method.
 *
 * <p><b>Note:</b> When actions are enclosed in tasks (such as
 * {@link FutureTask}) either explicitly or via methods such as
 * {@code submit}, these task objects catch and maintain
 * computational exceptions, and so they do not cause abrupt
 * termination, and the internal exceptions are <em>not</em>
 * passed to this method. If you would like to trap both kinds of
 * failures in this method, you can further probe for such cases,
 * as in this sample subclass that prints either the direct cause
 * or the underlying exception if a task has been aborted:
 *
 * <pre> {@code
 * class ExtendedExecutor extends ThreadPoolExecutor {
 *   // ...
 *   protected void afterExecute(Runnable r, Throwable t) {
 *     super.afterExecute(r, t);
 *     if (t == null
 *         && r instanceof Future<?>
 *         && ((Future<?>)r).isDone()) {
 *       try {
 *         Object result = ((Future<?>) r).get();
 *       } catch (CancellationException ce) {
 *         t = ce;
 *       } catch (ExecutionException ee) {
 *         t = ee.getCause();
 *       } catch (InterruptedException ie) {
 *         // ignore/reset
 *         Thread.currentThread().interrupt();
 *       }
 *     }
 *     if (t != null)
 *       System.out.println(t);
 *   }
 * }}</pre>
 *
 * @param r the runnable that has completed
 * @param t the exception that caused termination, or null if
 * execution completed normally
 */
protected void afterExecute(Runnable r, Throwable t) { }

/**
 * Method invoked when the Executor has terminated.  Default
 * implementation does nothing. Note: To properly nest multiple
 * overridings, subclasses should generally invoke
 * {@code super.terminated} within this method.
 */
protected void terminated() { }


```

四种拒绝策略

```java
/* Predefined RejectedExecutionHandlers */

/**
 * A handler for rejected tasks that runs the rejected task
 * directly in the calling thread of the {@code execute} method,
 * unless the executor has been shut down, in which case the task
 * is discarded.
 */
public static class CallerRunsPolicy implements RejectedExecutionHandler {
    /**
     * Creates a {@code CallerRunsPolicy}.
     */
    public CallerRunsPolicy() { }

    /**
     * Executes task r in the caller's thread, unless the executor
     * has been shut down, in which case the task is discarded.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
     */
    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            r.run();
        }
    }
}

/**
 * A handler for rejected tasks that throws a
 * {@link RejectedExecutionException}.
 *
 * This is the default handler for {@link ThreadPoolExecutor} and
 * {@link ScheduledThreadPoolExecutor}.
 */
public static class AbortPolicy implements RejectedExecutionHandler {
    /**
     * Creates an {@code AbortPolicy}.
     */
    public AbortPolicy() { }

    /**
     * Always throws RejectedExecutionException.
     *
     * @param r the runnable task requested to be executed
     * @param e the executor attempting to execute this task
```

```java
         * @throws RejectedExecutionException always
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            throw new RejectedExecutionException("Task " + r.toString() +
                                                 " rejected from " +
                                                 e.toString());
        }
    }

    /**
     * A handler for rejected tasks that silently discards the
     * rejected task.
     */
    public static class DiscardPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code DiscardPolicy}.
         */
        public DiscardPolicy() { }

        /**
         * Does nothing, which has the effect of discarding task r.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        }
    }

    /**
     * A handler for rejected tasks that discards the oldest unhandled
     * request and then retries {@code execute}, unless the executor
     * is shut down, in which case the task is discarded. This policy is
     * rarely useful in cases where other threads may be waiting for
     * tasks to terminate, or failures must be recorded. Instead consider
     * using a handler of the form:
     * <pre> {@code
     * new RejectedExecutionHandler() {
     *   public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
     *     Runnable dropped = e.getQueue().poll();
     *     if (dropped instanceof Future<?>) {
     *       ((Future<?>)dropped).cancel(false);
     *       // also consider logging the failure
     *     }
     *     e.execute(r);  // retry
     * }}}</pre>
     */
    public static class DiscardOldestPolicy implements RejectedExecutionHandler
    {
        /**
         * Creates a {@code DiscardOldestPolicy} for the given executor.
         */
        public DiscardOldestPolicy() { }

        /**
         * Obtains and ignores the next task that the executor
         * would otherwise execute, if one is immediately available,
         * and then retries execution of task r, unless the executor
         * is shut down, in which case task r is instead discarded.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                e.getQueue().poll();
                e.execute(r);
            }
        }
    }
```