目录

Doc

概括

JUC

AbstractExecutorService

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;
import java.util.*;

/**
 * Provides default implementations of {@link ExecutorService}
 * execution methods. This class implements the {@code submit},
 * {@code invokeAny} and {@code invokeAll} methods using a
 * {@link RunnableFuture} returned by {@code newTaskFor}, which defaults
 * to the {@link FutureTask} class provided in this package.  For example,
```

```
 * the implementation of {@code submit(Runnable)} creates an
 * associated {@code RunnableFuture} that is executed and
 * returned. Subclasses may override the {@code newTaskFor} methods
 * to return {@code RunnableFuture} implementations other than
 * {@code FutureTask}.
 * 提供ExecutorService执行方法的默认实现。
 * 该类通过使用RunnableFuture返回的newTaskFor（newTaskFor默认提供的返回对象为该包下的FutureTask类实例）实现了submit、invokeAny和invokeAll方法，
 * 例如，实现的submit(Runnable)方法，就创建一个关联的RunnableFuture对象，用于执行任务与返回。
 * 子类可以覆盖newTaskFor方法，以返回RunnableFuture的实现，而不是FutureTask的实现。
 *
 * <p><b>Extension example</b>. Here is a sketch of a class
 * that customizes {@link ThreadPoolExecutor} to use
 * a {@code CustomTask} class instead of the default {@code FutureTask}:
 * 扩展样例。
 * 这有一个草图（sketch），一个自定义的ThreadPoolExecutor使用CustomTask类替代默认的FutureTask类：
 *
 *  <pre> {@code
 * public class CustomThreadPoolExecutor extends ThreadPoolExecutor {
 *
 *   static class CustomTask<V> implements RunnableFuture<V> {...}
 *
 *   protected <V> RunnableFuture<V> newTaskFor(Callable<V> c) {
 *       return new CustomTask<V>(c); // AbstractExecutorService这里是返回一个new FutureTask，该自定义类返回了一个new CustomTask，只要自定义的类实现了RunnableFuture接口就行。
 *   }
 *   protected <V> RunnableFuture<V> newTaskFor(Runnable r, V v) {
 *       return new CustomTask<V>(r, v);
 *   }
 *   // ... add constructors, etc.
 * }}</pre>
 *
 * @since 1.5
 * @author Doug Lea
 */
public abstract class AbstractExecutorService implements ExecutorService {

    /**
     * Returns a {@code RunnableFuture} for the given runnable and default
     * value.
     * 返回RunnableFture，通过给定的runnable与默认返回值构建。
     *
     * @param runnable the runnable task being wrapped
     *        runnable 被包装的runnable任务
     * @param value the default value for the returned future
     *        value 默认的future返回值
     * @param <T> the type of the given value
```

```
     *        <T> 给定返回值的类型
     * @return a {@code RunnableFuture} which, when run, will run the
     * underlying runnable and which, as a {@code Future}, will yield
     * the given value as its result and provide for cancellation of
     * the underlying task
     *         返回一个RunnableFuture，在运行时会执行底层的runnable，
     *         并且作为Future，将使用给定的值作为返回结果，并提供底层任务的取消（方法）
     * @since 1.6
     */
    protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
        return new FutureTask<T>(runnable, value); // FutureTask会调用Executors#callable方法，将runnable+value转化为callable。如果runnable为null，抛出空指针异常
    }

    /**
     * Returns a {@code RunnableFuture} for the given callable task.
     * 返回Runnable，通过给定的callable任务构建。
     *
     * @param callable the callable task being wrapped
     * @param <T> the type of the callable's result
     * @return a {@code RunnableFuture} which, when run, will call the
     * underlying callable and which, as a {@code Future}, will yield
     * the callable's result as its result and provide for
     * cancellation of the underlying task
     * @since 1.6
     */
    protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
        return new FutureTask<T>(callable); // 如果callable为null，抛出空指针异常。否则创建FutureTask，callable=callable，state=NEW
    }

    /**
     * 提交返回值为null的Runnable任务
     * @throws RejectedExecutionException {@inheritDoc}
     * @throws NullPointerException       {@inheritDoc}
     */
    public Future<?> submit(Runnable task) {
        if (task == null) throw new NullPointerException(); // 如果提交的任务为null，抛出异常
        RunnableFuture<Void> ftask = newTaskFor(task, null); // 将任务转化为FutureTask
        execute(ftask);                                        // 这里的execute是Executor接口类定义的方法，具体的调用是实现类做的，比如ThreadPoolExecutor#execute(Runnable command)
        return ftask;                                          // 这里将提交的RunnableFuture对象返回了（是想拿这个Future来获取结果）
    }

    /**
     * 提交带有指定返回值的Runnable任务
     * @throws RejectedExecutionException {@inheritDoc}
```

```
     * @throws NullPointerException       {@inheritDoc}
     */
    public <T> Future<T> submit(Runnable task, T result) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task, result);
        execute(ftask);                                        // 这里是交给实现类做的，比如ThreadPoolExecutor#execute
        return ftask;
    }

    /**
     * 提交callable任务
     * @throws RejectedExecutionException {@inheritDoc}
     * @throws NullPointerException       {@inheritDoc}
     */
    public <T> Future<T> submit(Callable<T> task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task);
        execute(ftask);                                        // 这里是交给实现类做的，比如ThreadPoolExecutor#execute
        return ftask;
    }

    /**
     * the main mechanics of invokeAny.
     * invokeAny的主要机制
     * （执行给定的任务集合，返回其中一个成功完成的任务结果（即没有抛出异常），如果有的话。）
     */
    private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks,
                              boolean timed, long nanos) // tasks为要提交的任务集合，限时执行（该限时为拿到所有执行结果的总限时）
        throws InterruptedException, ExecutionException, TimeoutException {
        if (tasks == null)
            throw new NullPointerException();
        int ntasks = tasks.size();
        if (ntasks == 0)
            throw new IllegalArgumentException(); // 如果任务集合里没有任务，抛出IllegalArgumentException
        ArrayList<Future<T>> futures = new ArrayList<Future<T>>(ntasks); // 创建初识容量为任务数长度的空列表。这里使用future是为了能够方便操作任务，比如cancel等操作。
        ExecutorCompletionService<T> ecs =
            new ExecutorCompletionService<T>(this); // 用AbstractExecutorService创建ExecutorCompletionService（ecs），用于执行任务集，并将完成的任务保存在完成队列中

        // For efficiency, especially in executors with limited
        // parallelism, check to see if previously submitted tasks are
        // done before submitting more of them. This interleaving
        // plus the exception mechanics account for messiness of main
        // loop.
```

```
        // 为提高效率（efficiency），尤其是在并行性（parallelism）有限的执行器中（executors），
        // 在更多的任务提交之前检查之前提交的任务是否已完成。
        // 这种交错（interleaving 交叉）加上异常机制解释了（account for）主循环的混乱。

        try {
            // Record exceptions so that if we fail to obtain any
            // result, we can throw the last exception we got.
            // 记录异常，如果无法获取任何结果，可以抛出获取到的最后一个异常。
            ExecutionException ee = null;
            final long deadline = timed ? System.nanoTime() + nanos : 0L;  // 计算限时截止时间
            Iterator<? extends Callable<T>> it = tasks.iterator();

            // Start one task for sure; the rest incrementally
            // 确定开始的任务，其余逐渐增加
            futures.add(ecs.submit(it.next()));                   // ecs.submit将任务提交到Executor中执行，返回的RunnableFuture，是通过AES（也就是本类）的newTaskFor方法创建的。
            --ntasks;                                             // 执行了一个任务，等待任务数-1
            int active = 1;                                       // 活跃的任务数置为1。活跃任务=正在执行的任务

            for (;;) {
                Future<T> f = ecs.poll();                         // 获取完成队列队首元素，如果没有则返回null（不会阻塞，take或者poll(时限)才会阻塞）
                if (f == null) {                                  // f == null表示没有拿到队首元素，说明当前没有任务完成结果
                    if (ntasks > 0) {
                        --ntasks;
                        futures.add(ecs.submit(it.next()));       // 如果任务没执行完，那么将下一个任务加入ecs中执行（注意使用迭代器实现获取下一个任务）。任务数-1，活跃任务数+1
                        ++active;
                    }
                    else if (active == 0)
                        break;                                    // 如果活跃任务数为0，表示所有任务执行完毕，退出
                    else if (timed) {
                        f = ecs.poll(nanos, TimeUnit.NANOSECONDS);  // 到了这里，尚未执行的任务=0，活跃的任务>0，只需要等任务完成了。如果限时等待，就用ecs.poll(时限)的方法等，等不到抛异常
                        if (f == null)
                            throw new TimeoutException();
                        nanos = deadline - System.nanoTime();       // 注意这里，如果等到了一个，剩余等待时间需要减去当前已用时间
                    }
                    else
                        f = ecs.take();                           // 如果不限时，那就一直等了
```

```
                }                                                   // 到了这里可以看到
，最多有两个任务在并行执行
                if (f != null) {                                    // 如果有任务完成，拿
到完成结果。（注意这个f可以是在f==null分支里面执行的结果）
                    --active;                                       // 活跃任务数-1
                    try {
                        return f.get();                             // 返回执行结果
                    } catch (ExecutionException eex) {
                        ee = eex;                                   // ee用来记录上次的
异常结果
                    } catch (RuntimeException rex) {
                        ee = new ExecutionException(rex);
                    }
                }
            }

            if (ee == null)
                ee = new ExecutionException();                      // 执行到这里，说明
退出了for循环，但并没有执行结果的值，也没有异常信息，那么就抛出执行异常。
            throw ee;

        } finally {
            for (int i = 0, size = futures.size(); i < size; i++)
                futures.get(i).cancel(true);                        // 因为是invokeAny
，有一个执行完了，其他所有未完成的任务都取消。（只有FutureTask NEW->CANCELLED或者是NEW->INTE
RRUPTING->INTERRUPTED）
        }
    }

    // 执行给定的任务集合，返回其中一个成功完成的任务结果（即没有抛出异常），如果有的话。（重载
ExecutorService方法）
    // 不限时的invokeAny
    public <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException {
        try {
            return doInvokeAny(tasks, false, 0); // 不限时
        } catch (TimeoutException cannotHappen) {
            assert false; // 这个有意思，一个从来不会发生的异常如果进来了，直接断言为失败，
不return结果？？？（不知道有啥用）
            return null;
        }
    }

    // 执行给定的任务集合，返回其中一个成功完成的任务结果（即没有抛出异常），如果有的话。（重载
ExecutorService方法）
    // 限时的invokeAny（该限时为拿到所有执行结果的总限时）
    public <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                           long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException {
```

```
        return doInvokeAny(tasks, true, unit.toNanos(timeout));
    }

    // 执行给定的任务集合，返回Future列表持有当所有任务都执行完成后的状态和结果。（该方法会等待wait，直到所有任务都完成）（重载ExecutorService方法）
    // 不限时的invokeAll
    public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException {
        if (tasks == null)
            throw new NullPointerException();
        ArrayList<Future<T>> futures = new ArrayList<Future<T>>(tasks.size()); // 创建初识容量为任务数长度的空列表。这里使用future是为了能够方便操作任务，比如cancel等操作。
        boolean done = false;                                       // 所有任务完成标识
        try {
            for (Callable<T> t : tasks) {
                RunnableFuture<T> f = newTaskFor(t);
                futures.add(f);                                     // 将包装后的任务加入到futures
                execute(f);                                         // 调用子类的execute方法执行任务（像ThreadPoolExecutor可能只是提交了任务，任务需要排队执行）
            }
            for (int i = 0, size = futures.size(); i < size; i++) {
                Future<T> f = futures.get(i);                       // 遍历futures，拿每个任务的执行future
                if (!f.isDone()) {                                  // 判断该任务是否执行完成（state!=NEW）
                    try {
                        f.get();                                    // 如果没完成，通过FutureTask#get()来FutureTask#awaitDone(false, 0L)，等待完成
                    } catch (CancellationException ignore) {
                    } catch (ExecutionException ignore) {
                    }
                }
            }
            done = true;                                            // 执行到这里说明所有任务都完成了
            return futures;                                         // 返回结果是所有future集合
        } finally {
            if (!done)                                              // 如果任务没完成而到了这一步，说明可能该方法被中断，需要取消所有未完成的任务
                for (int i = 0, size = futures.size(); i < size; i++)
                    futures.get(i).cancel(true);                    // 取消任务（可取消的任务state == NEW），可通过中断runner来结束
        }
    }

    // 执行给定的任务集合，返回Future列表持有当所有任务都执行完成后的状态和结果。（该方法会等待wait，直到所有任务都完成）（重载ExecutorService方法）
```

```
    // 限时的invokeAll（该限时为拿到所有执行结果的总限时）
    public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                         long timeout, TimeUnit unit)
        throws InterruptedException {
        if (tasks == null)
            throw new NullPointerException();
        long nanos = unit.toNanos(timeout);
        ArrayList<Future<T>> futures = new ArrayList<Future<T>>(tasks.size());
        boolean done = false;
        try {
            for (Callable<T> t : tasks)
                futures.add(newTaskFor(t));                          // 这里跟不限时的invokeAll有区别，并没有立即将任务放到Executor（具体的实现类）#execute执行

            final long deadline = System.nanoTime() + nanos;
            final int size = futures.size();

            // Interleave time checks and calls to execute in case
            // executor doesn't have any/much parallelism.
            // 交错时间检查和调用执行，以防止executor没有任何/很多并行性
            // （逐个任务提交执行（不并行处理，实际由Executor实现子类来决定），记录执行时间，确保限时功能）
            for (int i = 0; i < size; i++) {
                execute((Runnable)futures.get(i));                   // 逐个提交执行，计算剩余时间，如果剩余时间<=0L，返回（在finally中停止尚未执行的任务）
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L)
                    return futures;
            }

            for (int i = 0; i < size; i++) {
                Future<T> f = futures.get(i);
                if (!f.isDone()) {
                    if (nanos <= 0L)
                        return futures;
                    try {
                        f.get(nanos, TimeUnit.NANOSECONDS);          // 用剩余时间等待任务完成，如果任务执行超过剩余等待时间，或者部分任务执行时间已经超过等待时间，返回（在finally中停止尚未执行的任务）
                    } catch (CancellationException ignore) {
                    } catch (ExecutionException ignore) {
                    } catch (TimeoutException toe) {
                        return futures;
                    }
                    nanos = deadline - System.nanoTime();
                }
            }
            done = true;
            return futures;
```

```
        } finally {
            if (!done)
                for (int i = 0, size = futures.size(); i < size; i++)
                    futures.get(i).cancel(true);                      // 取消任务（可取消的任务state == NEW），可通过中断runner来结束
        }
    }

}
```

Callable

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;

/**
 * A task that returns a result and may throw an exception.
 * Implementors define a single method with no arguments called
 * {@code call}.
 * 任务，能够返回结果，也可能抛出异常。
 * 实现者（需要）定义一个没有入参的call方法
 *
```

```
 * <p>The {@code Callable} interface is similar to {@link
 * java.lang.Runnable}, in that both are designed for classes whose
 * instances are potentially executed by another thread.  A
 * {@code Runnable}, however, does not return a result and cannot
 * throw a checked exception.
 * Callable接口与java.lang.Runnable相似，都是为了某些潜在的（potentially）想要运行在其他线程中的类实例设计的。
 * 不过，Runnable不会返回执行结果，也不会抛出检查型异常（这个是值得关注的差异点）
 *
 * <p>The {@link Executors} class contains utility methods to
 * convert from other common forms to {@code Callable} classes.
 * Executors类包含一些实用（utility）方法将其他常见形式的类转化为Callable类。
 *
 * @see Executor
 * @since 1.5
 * @author Doug Lea
 * @param <V> the result type of method {@code call}
 */
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     * 计算结果，如果无法计算则抛出异常（Exception）。
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

Executor

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;

/**
 * An object that executes submitted {@link Runnable} tasks. This
 * interface provides a way of decoupling task submission from the
 * mechanics of how each task will be run, including details of thread
 * use, scheduling, etc.  An {@code Executor} is normally used
 * instead of explicitly creating threads. For example, rather than
 * invoking {@code new Thread(new(RunnableTask())).start()} for each
```

```
 * of a set of tasks, you might use:
 * 一个执行提交的Runnable任务的对象。
 * 该接口提供了一种将任务提交与每个任务将如何运行的机制分离的方法，包含线程使用、调度等细节。
 * Executor通常用于替代显式（explicitly）创建线程。
 * 例如：不想为了task集合中的的每一个都调用new Thread(new(RunnableTask())).start()，可以使
用下面的方法：
 *
 * <pre>
 * Executor executor = <em>anExecutor</em>;
 * executor.execute(new RunnableTask1());
 * executor.execute(new RunnableTask2());
 * ...
 * </pre>
 *
 * However, the {@code Executor} interface does not strictly
 * require that execution be asynchronous. In the simplest case, an
 * executor can run the submitted task immediately in the caller's
 * thread:
 * 但是，Executor接口并不严格（strictly）要求执行是异步的。
 * 举一个简单的例子：executor可以在调用者的线程里立即执行task：
 *
 *  <pre> {@code
 * class DirectExecutor implements Executor {
 *   public void execute(Runnable r) {
 *     r.run(); // 没有通过传入线程对象执行，那么run就作为普通方法执行了
 *   }
 * }}</pre>
 *
 * More typically, tasks are executed in some thread other
 * than the caller's thread.  The executor below spawns a new thread
 * for each task.
 * 通常情况，task在调用者线程之外，启动线程执行。
 * 下面的执行程序为每个任务生成一个新线程：
 *
 *  <pre> {@code
 * class ThreadPerTaskExecutor implements Executor {
 *   public void execute(Runnable r) {
 *     new Thread(r).start(); // 每个run方法都会在不同的Thread中执行
 *   }
 * }}</pre>
 *
 * Many {@code Executor} implementations impose some sort of
 * limitation on how and when tasks are scheduled.  The executor below
 * serializes the submission of tasks to a second executor,
 * illustrating a composite executor.
 * 许多Executor的实现，对task调度的方式和时间施加（impose 强加）了一些限制。
 * 下面的executor将任务的提交序列化到了第二个executor，说明（illustrating）了一个复合（compo
site）executor
 *
```

```
 *  <pre> {@code
 * class SerialExecutor implements Executor {
 *   final Queue<Runnable> tasks = new ArrayDeque<Runnable>();
 *   final Executor executor; // 内部有一个Executor
 *   Runnable active;
 *
 *   SerialExecutor(Executor executor) {
 *     this.executor = executor;
 *   }
 *
 *   public synchronized void execute(final Runnable r) {
 *     tasks.offer(new Runnable() {
 *       public void run() {
 *         try {
 *           r.run();
 *         } finally {
 *           scheduleNext();
 *         }
 *       }
 *     });
 *     if (active == null) {
 *       scheduleNext();
 *     }
 *   }
 *
 *   protected synchronized void scheduleNext() {
 *     if ((active = tasks.poll()) != null) {
 *       executor.execute(active); // 本类的execute方法执行完后，继续执行下一个executor的e
xecute
 *     }
 *   }
 * }}</pre>
 *
 * The {@code Executor} implementations provided in this package
 * implement {@link ExecutorService}, which is a more extensive
 * interface.  The {@link ThreadPoolExecutor} class provides an
 * extensible thread pool implementation. The {@link Executors} class
 * provides convenient factory methods for these Executors.
 * Executor在包里提供了一个ExecutorService实现类，该类是一个更广泛的接口。
 * ThreadPoolExecutor类提供了一个可扩展的线程池实现。
 * Executors类为这些Executor提供了方便的（convenient）的工厂方法。
 *
 * <p>Memory consistency effects: Actions in a thread prior to
 * submitting a {@code Runnable} object to an {@code Executor}
 * <a href="package-summary.html#MemoryVisibility"><i>happen-before</i></a>
 * its execution begins, perhaps in another thread.
 * 内存一致性影响：在线程提交Runnable对象给Executor之前，内存可见性happen-before执行之前，可
能在其它线程（不知道啥意思）
 *
```

```
 * @since 1.5
 * @author Doug Lea
 */
public interface Executor {

    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the {@code Executor} implementation.
     * 执行给定的命令（实现Runnable接口的）在未来的某个时间。
     * 该命令可能在一个新的线程中执行，或者在一个线程池执行，或者在调用者的线程里执行，执行方式由Executor的实现来决定。
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}
```

ExecutorCompletionService

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;

/**
 * A {@link CompletionService} that uses a supplied {@link Executor}
 * to execute tasks.  This class arranges that submitted tasks are,
 * upon completion, placed on a queue accessible using {@code take}.
 * The class is lightweight enough to be suitable for transient use
 * when processing groups of tasks.
 * CompletionService，使用提供的Executor来执行任务。
```

```
 * 该类安排提交的任务完成后，通过take放到可访问的队列中。
 * 该类足够轻量级，适合在处理任务组时临时使用。
 *
 * <p>
 *
 * <b>Usage Examples.</b>
 * 用例
 *
 * Suppose you have a set of solvers for a certain problem, each
 * returning a value of some type {@code Result}, and would like to
 * run them concurrently, processing the results of each of them that
 * return a non-null value, in some method {@code use(Result r)}. You
 * could write this as:
 * 假设你有一组对某个核心问题的求解器，每个求解器都返回某种Result类型的值，并且希望并发运行他们，
 * 在某些方法中使用use(Result r)处理他们返回的每一个非null结果。
 * 你可以这样写：
 *
 * <pre> {@code
 * void solve(Executor e,
 *            Collection<Callable<Result>> solvers)
 *     throws InterruptedException, ExecutionException {
 *     CompletionService<Result> ecs
 *         = new ExecutorCompletionService<Result>(e); // 用给定的executor创建CompletionService
 *     for (Callable<Result> s : solvers)
 *         ecs.submit(s);                // 提交任务，将完成结果写入完成队列汇总
 *     int n = solvers.size(); // 这里遍历的是solvers的数量
 *     for (int i = 0; i < n; ++i) {
 *         Result r = ecs.take().get(); // 遍历完成队列，获取执行完成的结果
 *         if (r != null)
 *             use(r);                  // 自定义处理执行结果
 *     }
 * }}</pre>
 *
 * Suppose instead that you would like to use the first non-null result
 * of the set of tasks, ignoring any that encounter exceptions,
 * and cancelling all other tasks when the first one is ready:
 * 假设你想使用任务集里的第一个非空结果，忽略任何遇到（encounter）的异常结果，并且在第一个已完成后取消所有其他线程
 *
 * <pre> {@code
 * void solve(Executor e,
 *            Collection<Callable<Result>> solvers)
 *     throws InterruptedException {
 *     CompletionService<Result> ecs
 *         = new ExecutorCompletionService<Result>(e);
 *     int n = solvers.size();
 *     List<Future<Result>> futures
```

```
 *         = new ArrayList<Future<Result>>(n); // 这里用Future，是为了可以cancel任务
 *     Result result = null;
 *     try {
 *         for (Callable<Result> s : solvers)
 *             futures.add(ecs.submit(s));
 *         for (int i = 0; i < n; ++i) {
 *             try {
 *                 Result r = ecs.take().get(); // 从队首开始，找第一个非空结果
 *                 if (r != null) {
 *                     result = r;
 *                     break;
 *                 }
 *             } catch (ExecutionException ignore) {}
 *         }
 *     }
 *     finally {                               // 使用future将未完成的任务取消
 *         for (Future<Result> f : futures)
 *             f.cancel(true);
 *     }
 *
 *     if (result != null)
 *         use(result);                       // 如果有非空结果，进行自定义处理
 * }}</pre>
 */
public class ExecutorCompletionService<V> implements CompletionService<V> {
    private final Executor executor;                        // 保存给定的executor
    private final AbstractExecutorService aes;              // 如果给定的executor是aes，保存该对象
    private final BlockingQueue<Future<V>> completionQueue; // 保存完成的任务

    /**
     * FutureTask extension to enqueue upon completion
     * 对FutureTask的扩展，在任务完成时入队（入完成队列）
     */
    private class QueueingFuture extends FutureTask<Void> {
        QueueingFuture(RunnableFuture<V> task) {
            super(task, null);  // 调用FutureTask的FutureTask(Runnable runnable, V result)构造函数，这里设置返回结果为null
            this.task = task;    // 本类中新增的属性，用于记录提交的任务。（FutureTask里需要执行的任务用callable来引用的）
        }
        protected void done() { completionQueue.add(task); } // 这里是重点，对FutureTask的done方法进行重载，将完成的任务加入ExecutorCompletionService的完成队列中
        private final Future<V> task;
    }

    // 将callable任务转化为RunnableFuture
    // 如果aes为null，表示当前executor不是AbstractExecutorService实现的，创建FutureTask对象返回
```

```
    // 如果aes不为null，那么直接使用AbstractExecutorService#newTaskFor方法创建RunnableFut
ure结果
    // 看AbstractExecutorService#newTaskFor方法，也是用的new FutureTask，
    // 推测是为了方便如果aes的实现类重载了newTaskFor方法，那么该方法就可以调用aes的实现类方法
    private RunnableFuture<V> newTaskFor(Callable<V> task) {
        if (aes == null)
            return new FutureTask<V>(task);
        else
            return aes.newTaskFor(task);
    }

    // 与上面的方法类似
    private RunnableFuture<V> newTaskFor(Runnable task, V result) {
        if (aes == null)
            return new FutureTask<V>(task, result);
        else
            return aes.newTaskFor(task, result);
    }

    /**
     * Creates an ExecutorCompletionService using the supplied
     * executor for base task execution and a
     * {@link LinkedBlockingQueue} as a completion queue.
     * 创建一个ExecutorCompletionService，用提供的执行器（executor）执行基础任务，
     * 和创建一个LinkedBlockingQueue作为完成队列。
     *
     * @param executor the executor to use
     * @throws NullPointerException if executor is {@code null}
     */
    public ExecutorCompletionService(Executor executor) {
        if (executor == null)
            throw new NullPointerException();                          // 如果给定的执行
器为null，抛出空指针异常
        this.executor = executor;                                      // 设置执行器
        this.aes = (executor instanceof AbstractExecutorService) ?    // 判断如果给定的执
行器是AbstractExecutorService实例，那么设置aes为给定执行器，否则为null。这里设置aes是为了后
续使用aes现有的方法
            (AbstractExecutorService) executor : null;
        this.completionQueue = new LinkedBlockingQueue<Future<V>>();  // 设置completion
Queue为新的LinkedBlockingQueue，元素为Future<V>
    }

    /**
     * Creates an ExecutorCompletionService using the supplied
     * executor for base task execution and the supplied queue as its
     * completion queue.
     * 创建一个ExecutorCompletionService，用提供的执行器（executor）执行基础任务，
     * 和使用提供的队列作为完成队列
     *
```

```
     * @param executor the executor to use
     * @param completionQueue the queue to use as the completion queue
     *        normally one dedicated for use by this service. This
     *        queue is treated as unbounded -- failed attempted
     *        {@code Queue.add} operations for completed tasks cause
     *        them not to be retrievable.
     *        给定的completionQueue通常专用于（delicated）该服务作为完成队列。
     *        该队列被视为（treated 对待的）无界的 -- 已完成任务尝试Queue.add操作失败的话，
会导致他们不可取回。
     *
     * @throws NullPointerException if executor or completionQueue are {@code null}
     */
    public ExecutorCompletionService(Executor executor,
                                     BlockingQueue<Future<V>> completionQueue) {
        if (executor == null || completionQueue == null)
            throw new NullPointerException();
        this.executor = executor;
        this.aes = (executor instanceof AbstractExecutorService) ?
            (AbstractExecutorService) executor : null;
        this.completionQueue = completionQueue;                        // 使用给定的等待
队列
    }

    // 提交任务
    public Future<V> submit(Callable<V> task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<V> f = newTaskFor(task);        // 将callable任务转化为RunnableFu
ture
        executor.execute(new QueueingFuture(f));       //
        return f;
    }

    public Future<V> submit(Runnable task, V result) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<V> f = newTaskFor(task, result);
        executor.execute(new QueueingFuture(f));
        return f;
    }

    // 获取队首元素，必要时需等待（如果队列为空，就需阻塞等待）
    public Future<V> take() throws InterruptedException {
        return completionQueue.take(); // 调用的是BlockingQueue#take方法
    }

    // 获取队首元素，如果队列为空返回null（这个不会阻塞）
    public Future<V> poll() {
        return completionQueue.poll(); // 调用的是Queue#poll方法
    }
```

```
    // 获取队首元素，必要时在有限时间内等待队首，如果仍等不到返回null
    // 与take的不同就是这个是有限时间等待
    public Future<V> poll(long timeout, TimeUnit unit)
            throws InterruptedException {
        return completionQueue.poll(timeout, unit); // 调用的是BlockingQueue#poll
    }

}
```

ExecutorService

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;
import java.util.List;
import java.util.Collection;

/**
 * An {@link Executor} that provides methods to manage termination and
 * methods that can produce a {@link Future} for tracking progress of
 * one or more asynchronous tasks.
 * 一个Executor提供管理终止的方法，并且可以生成Future来跟踪一个或多个异步任务的执行进度的方法
```

```
。
 *
 * <p>An {@code ExecutorService} can be shut down, which will cause
 * it to reject new tasks.  Two different methods are provided for
 * shutting down an {@code ExecutorService}. The {@link #shutdown}
 * method will allow previously submitted tasks to execute before
 * terminating, while the {@link #shutdownNow} method prevents waiting
 * tasks from starting and attempts to stop currently executing tasks.
 * Upon termination, an executor has no tasks actively executing, no
 * tasks awaiting execution, and no new tasks can be submitted.  An
 * unused {@code ExecutorService} should be shut down to allow
 * reclamation of its resources.
 * 可以关闭（shut down）ExecutorService，将它导致拒绝新任务。
 * 提供了两种不同的方法来shutdownExecutorService。
 * 1、shutdown方法，将允许之前提交的任务在终止之前继续执行。
 * 2、shutdownNow方法，防止（prevent）等待任务开始，并且尝试结束当前正在执行的任务。
 * 终止时，executor没有正在执行的任务，没有等待执行的任务，没有新任务可以提交。
 * 应关闭未使用的ExecutorService，以回收资源。
 *
 * <p>Method {@code submit} extends base method {@link
 * Executor#execute(Runnable)} by creating and returning a {@link Future}
 * that can be used to cancel execution and/or wait for completion.
 * Methods {@code invokeAny} and {@code invokeAll} perform the most
 * commonly useful forms of bulk execution, executing a collection of
 * tasks and then waiting for at least one, or all, to
 * complete. (Class {@link ExecutorCompletionService} can be used to
 * write customized variants of these methods.)
 * submit方法基于Executor#execute(Runnable)方法扩展，通过创建和返回Future对象，Future可以用来取消执行的任务，并且/或等待任务完成。
 * invokeAny方法和invokeAll方法最常用于批量（bulk）任务执行，执行一组任务，然后等待至少一个或者所有执行完成。
 *（ExecutorCompletionService类可用于编写这些方法的自定义变体（variants 变种））
 *
 * <p>The {@link Executors} class provides factory methods for the
 * executor services provided in this package.
 * 在本包里的Executors类为executor服务提供了工厂方法
 *
 * <h3>Usage Examples</h3>
 * 用例
 *
 * Here is a sketch of a network service in which threads in a thread
 * pool service incoming requests. It uses the preconfigured {@link
 * Executors#newFixedThreadPool} factory method:
 * 这是一个网络服务的草图，在线程池里的线程服务进来的请求。
 * 使用预配置的Executors#newFixedThreadPool工厂方法：
 *
 *  <pre> {@code
 * class NetworkService implements Runnable { // 为什么要实现Runnable接口，为了表明这是个要执行的类？
```

```
*   private final ServerSocket serverSocket;
*   private final ExecutorService pool;
*
*   public NetworkService(int port, int poolSize)
*       throws IOException {
*     serverSocket = new ServerSocket(port);
*     pool = Executors.newFixedThreadPool(poolSize);
*   }
*
*   public void run() { // run the service
*     try {
*       for (;;) {
*         pool.execute(new Handler(serverSocket.accept()));
*       }
*     } catch (IOException ex) {
*       pool.shutdown();        // 这里给手工shutdown了
*     }
*   }
* }
*
* class Handler implements Runnable {
*   private final Socket socket;
*   Handler(Socket socket) { this.socket = socket; }
*   public void run() {
*     // read and service request on socket
*   }
* }}</pre>
*
* The following method shuts down an {@code ExecutorService} in two phases,
* first by calling {@code shutdown} to reject incoming tasks, and then
* calling {@code shutdownNow}, if necessary, to cancel any lingering tasks:
* 接下来的方法从两个阶段停止ExecutorService，
* 1、首先调用shutdown方法拒绝传入任务（拒绝新增任务）
* 2、然后在有必要的时候，调用shutdownNow方法，取消任何延迟（linger）任务
*
*  <pre> {@code
* void shutdownAndAwaitTermination(ExecutorService pool) {
*   pool.shutdown(); // Disable new tasks from being submitted
*   try {
*     // Wait a while for existing tasks to terminate
*     if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
*       pool.shutdownNow(); // Cancel currently executing tasks
*       // Wait a while for tasks to respond to being cancelled
*       if (!pool.awaitTermination(60, TimeUnit.SECONDS))
*           System.err.println("Pool did not terminate");
*     }
*   } catch (InterruptedException ie) {
*     // (Re-)Cancel if current thread also interrupted
*     pool.shutdownNow();
```

```
 *     // Preserve interrupt status
 *     Thread.currentThread().interrupt();
 *   }
 * }}</pre>
 *
 * <p>Memory consistency effects: Actions in a thread prior to the
 * submission of a {@code Runnable} or {@code Callable} task to an
 * {@code ExecutorService}
 * <a href="package-summary.html#MemoryVisibility"><i>happen-before</i></a>
 * any actions taken by that task, which in turn <i>happen-before</i> the
 * result is retrieved via {@code Future.get()}.
 * 内存一致性影响：在将Runnable或者Callable任务提交ExecutorService之前，MemoryVisibility发生在该任务采取的任何操作之前（happen-before）
 * 而后者又发生在通过Future.get()检索结果之前。
 *
 * @since 1.5
 * @author Doug Lea
 */
public interface ExecutorService extends Executor {

    /**
     * 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.
     * 启动有序的关闭，之前提交的任务将继续执行，但不会接受新的任务。
     * 如果已经shutdown关闭了，多次调用该方法不会有任何额外效果（或者说影响）。
     *
     * <p>This method does not wait for previously submitted tasks to
     * complete execution.  Use {@link #awaitTermination awaitTermination}
     * to do that.
     * 该方法不会等之前提交的任务执行完毕。
     * 想要做到这一点的话，使用awaitTermination方法
     *
     * @throws SecurityException if a security manager exists and
     *         shutting down this ExecutorService may manipulate
     *         threads that the caller is not permitted to modify
     *         because it does not hold {@link
     *         java.lang.RuntimePermission}{@code ("modifyThread")},
     *         or the security manager's {@code checkAccess} method
     *         denies access.
     *         如果安全管理器存在，并且关闭这个ExecutorService可能会操纵不允许调用者修改的线程，因为它没有持有修改线程的执行权限（RuntimePermission），
     *         或者安全管理器的checkAccess方法拒绝访问。
     */
    void shutdown();

    /**
     * Attempts to stop all actively executing tasks, halts the
     * processing of waiting tasks, and returns a list of the tasks
```

```
     * that were awaiting execution.
     * 尝试停止全部正在执行的活跃任务，停止（halt）等待任务的处理，并返回等待执行的任务列表。
     *
     * <p>This method does not wait for actively executing tasks to
     * terminate.  Use {@link #awaitTermination awaitTermination} to
     * do that.
     * 该方法不会等待正在执行的任务停止。（停止而不是完成）
     * 如果想做到这一点，使用awaitTermination方法
     *
     * <p>There are no guarantees beyond best-effort attempts to stop
     * processing actively executing tasks.  For example, typical
     * implementations will cancel via {@link Thread#interrupt}, so any
     * task that fails to respond to interrupts may never terminate.
     * 不保证，除了尽力尝试停止正在执行的任务处理。例如，典型的实现方式为通过interrupt来取消（正在执行的线程），因此任何未能响应中断的任务可能永远不会终止。
     *（意思就是调用了shutdownNow方法，只是会尝试去取消正在执行的任务，通常是用interrupt来实现，如果任务不响应中断，那任务还是会正常执行）
     *
     * @return list of tasks that never commenced execution
     *         从未开始（commenced）执行的任务列表（等待任务列表）
     * @throws SecurityException if a security manager exists and
     *         shutting down this ExecutorService may manipulate
     *         threads that the caller is not permitted to modify
     *         because it does not hold {@link
     *         java.lang.RuntimePermission}{@code ("modifyThread")},
     *         or the security manager's {@code checkAccess} method
     *         denies access.
     */
    List<Runnable> shutdownNow();

    /**
     * Returns {@code true} if this executor has been shut down.
     * 如果该executor被关闭了，返回true
     *
     * @return {@code true} if this executor has been shut down
     */
    boolean isShutdown();

    /**
     * Returns {@code true} if all tasks have completed following shut down.
     * Note that {@code isTerminated} is never {@code true} unless
     * either {@code shutdown} or {@code shutdownNow} was called first.
     * 如果所有任务伴随着shutdown都结束了，返回true
     * 注意，除非shutdown()或者shutdownNow()方法首先被调用，否则直接调用该方法并不会返回true
。
     *
     * @return {@code true} if all tasks have completed following shut down
     */
    boolean isTerminated();
```

```
    /**
     * Blocks until all tasks have completed execution after a shutdown
     * request, or the timeout occurs, or the current thread is
     * interrupted, whichever happens first.
     * 在shutdown请求之后进行阻塞，直到所有任务完成处理，或者处理超时，或者当前线程被中断，以先发生者为准。
     *
     * @param timeout the maximum time to wait
     *        timeout参数，最大等待时间
     * @param unit the time unit of the timeout argument
     * @return {@code true} if this executor terminated and
     *         {@code false} if the timeout elapsed before termination
     *         true  如果executor终止
     *         false 如果在终止前超时
     * @throws InterruptedException if interrupted while waiting
     *         抛出InterruptedException，如果在等待时被中断。
     */
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;

    /**
     * Submits a value-returning task for execution and returns a
     * Future representing the pending results of the task. The
     * Future's {@code get} method will return the task's result upon
     * successful completion.
     * 提交一个有返回值的任务执行，并且返回一个代表等待task执行结果的Future。
     * Future的get方法将在task执行成功后返回执行结果。
     *
     * <p>
     * If you would like to immediately block waiting
     * for a task, you can use constructions of the form
     * {@code result = exec.submit(aCallable).get();}
     * 如果想立即阻塞等待任务完成，可以使用如下结构形式
     * result=exec.submit(aCallable).get(); // 提交任务，直接get等待
     *
     * <p>Note: The {@link Executors} class includes a set of methods
     * that can convert some other common closure-like objects,
     * for example, {@link java.security.PrivilegedAction} to
     * {@link Callable} form so they can be submitted.
     * 注意：Executors类包含了一系列的方法，可以将一些其他常见的类似闭包（closure-like）的对象（例如PrivilegedAction）转化为Callable形式，
     * 以便他们可以被提交
     *
     * @param task the task to submit
     *        task 提交的任务
     * @param <T> the type of the task's result
     *        <T> 泛型T代表任务的执行结果类型
     * @return a Future representing pending completion of the task
```

```
     *         Future代表等待task的执行
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     *         RejectedExecutionException 如果该任务无法被execution安排执行（调度）
     * @throws NullPointerException if the task is null
     */
    <T> Future<T> submit(Callable<T> task);

    /**
     * Submits a Runnable task for execution and returns a Future
     * representing that task. The Future's {@code get} method will
     * return the given result upon successful completion.
     * 提交一个Runnable任务执行并返回一个表示该任务的Future。
     * Future的get方法将在任务执行完成后返回给定的result（通过参数传入的result）
     *
     * @param task the task to submit
     * @param result the result to return
     *        result 任务执行结束后的返回值
     * @param <T> the type of the result
     * @return a Future representing pending completion of the task
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     * @throws NullPointerException if the task is null
     */
    <T> Future<T> submit(Runnable task, T result);

    /**
     * Submits a Runnable task for execution and returns a Future
     * representing that task. The Future's {@code get} method will
     * return {@code null} upon <em>successful</em> completion.
     * 提交一个Runnable任务执行并返回一个表示该任务的Future。
     * Future的get方法在任务成功执行完成将返回null。
     *
     * @param task the task to submit
     * @return a Future representing pending completion of the task
     * @throws RejectedExecutionException if the task cannot be
     *         scheduled for execution
     * @throws NullPointerException if the task is null
     */
    Future<?> submit(Runnable task);

    /**
     * Executes the given tasks, returning a list of Futures holding
     * their status and results when all complete.
     * {@link Future#isDone} is {@code true} for each
     * element of the returned list.
     * Note that a <em>completed</em> task could have
     * terminated either normally or by throwing an exception.
     * The results of this method are undefined if the given
```

```
     * collection is modified while this operation is in progress.
     * 执行给定的任务集合，返回Future列表持有当所有任务都执行完成后的状态和结果。（该方法会等待wait，直到所有任务都完成）
     * Future的isDone方法对每一个返回列表元素都是true。
     * 注意，已完成的任务可能是正常终止或者是抛出了异常。
     * 如果给定的集合在此操作期间被修改，该方法的结果是未定义。
     *
     * @param tasks the collection of tasks
     * @param <T> the type of the values returned from the tasks
     * @return a list of Futures representing the tasks, in the same
     *         sequential order as produced by the iterator for the
     *         given task list, each of which has completed
     * 返回值  代表任务的Future列表，通过给定的task列表迭代器生成相同顺序的序列，每个任务都已完成。（因为这个方法会等待所有集合里的任务都执行完成）
     * @throws InterruptedException if interrupted while waiting, in
     *         which case unfinished tasks are cancelled
     *         InterruptedException，如果在等待时中断，在这种情况下未完成的任务将取消。
     * @throws NullPointerException if tasks or any of its elements are {@code null}
     * @throws RejectedExecutionException if any task cannot be
     *         scheduled for execution
     */
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;

    /**
     * Executes the given tasks, returning a list of Futures holding
     * their status and results
     * when all complete or the timeout expires, whichever happens first.
     * {@link Future#isDone} is {@code true} for each
     * element of the returned list.
     * Upon return, tasks that have not completed are cancelled.
     * Note that a <em>completed</em> task could have
     * terminated either normally or by throwing an exception.
     * The results of this method are undefined if the given
     * collection is modified while this operation is in progress.
     * 执行给定的任务集合，返回一个Future列表，持有当所有task完成，或者发生超时（以先发生的为准）后的状态与结果。
     * Future的Done方法对每一个返回列表中的元素都是true。
     * 返回后，未完成的任务将被取消。（在运行超时时，对于还没完成的任务就直接取消）
     * 注意，已完成的任务可能是正常终止或者是抛出了异常。
     * 如果给定的集合在此操作期间被修改，该方法的结果是未定义
     *
     * @param tasks the collection of tasks
     * @param timeout the maximum time to wait
     * @param unit the time unit of the timeout argument
     * @param <T> the type of the values returned from the tasks
     * @return a list of Futures representing the tasks, in the same
     *         sequential order as produced by the iterator for the
     *         given task list. If the operation did not time out,
```

```
     *         each task will have completed. If it did time out, some
     *         of these tasks will not have completed.
     *         如果没有超时操作，每个任务都将完成。
     *         如果有超时，一些任务将无法完成。
     * @throws InterruptedException if interrupted while waiting, in
     *         which case unfinished tasks are cancelled
     * @throws NullPointerException if tasks, any of its elements, or
     *         unit are {@code null}
     * @throws RejectedExecutionException if any task cannot be scheduled
     *         for execution
     */
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                  long timeout, TimeUnit unit)
        throws InterruptedException;

    /**
     * Executes the given tasks, returning the result
     * of one that has completed successfully (i.e., without throwing
     * an exception), if any do. Upon normal or exceptional return,
     * tasks that have not completed are cancelled.
     * The results of this method are undefined if the given
     * collection is modified while this operation is in progress.
     * 执行给定的任务集合，返回其中一个成功完成的任务结果（即没有抛出异常），如果有的话。
     * 在正常或者异常返回时，未完成的task将被取消。（意思是所有任务有一个正常执行完，其他任务
就都取消？？？这大概就是invokeAny（执行任意一个）的含义）
     * 如果给定的集合在此操作期间被修改，该方法的结果是未定义
     *
     * @param tasks the collection of tasks
     * @param <T> the type of the values returned from the tasks
     * @return the result returned by one of the tasks
     * 返回值  返回其中一个任务的执行结果
     * @throws InterruptedException if interrupted while waiting
     * @throws NullPointerException if tasks or any element task
     *         subject to execution is {@code null}
     * @throws IllegalArgumentException if tasks is empty
     * @throws ExecutionException if no task successfully completes
     * @throws RejectedExecutionException if tasks cannot be scheduled
     *         for execution
     */
    <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;

    /**
     * Executes the given tasks, returning the result
     * of one that has completed successfully (i.e., without throwing
     * an exception), if any do before the given timeout elapses.
     * Upon normal or exceptional return, tasks that have not
     * completed are cancelled.
     * The results of this method are undefined if the given
```

```
     * collection is modified while this operation is in progress.
     * 执行给定的任务集合，返回其中一个成功完成的任务结果（即没有抛出异常），如果在超时之前有的话。
     * 在正常或者异常返回时，未完成的task将被取消。（意思是所有任务有一个正常执行完，其他任务就都取消？？？）
     * 如果给定的集合在此操作期间被修改，该方法的结果是未定义
     *
     * @param tasks the collection of tasks
     * @param timeout the maximum time to wait
     * @param unit the time unit of the timeout argument
     * @param <T> the type of the values returned from the tasks
     * @return the result returned by one of the tasks
     * @throws InterruptedException if interrupted while waiting
     * @throws NullPointerException if tasks, or unit, or any element
     *         task subject to execution is {@code null}
     * @throws TimeoutException if the given timeout elapses before
     *         any task successfully completes
     * @throws ExecutionException if no task successfully completes
     * @throws RejectedExecutionException if tasks cannot be scheduled
     *         for execution
     */
    <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                    long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

Future

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;

/**
 * A {@code Future} represents the result of an asynchronous
 * computation.  Methods are provided to check if the computation is
 * complete, to wait for its completion, and to retrieve the result of
 * the computation.  The result can only be retrieved using method
 * {@code get} when the computation has completed, blocking if
 * necessary until it is ready.  Cancellation is performed by the
```

```
 * {@code cancel} method.  Additional methods are provided to
 * determine if the task completed normally or was cancelled. Once a
 * computation has completed, the computation cannot be cancelled.
 * If you would like to use a {@code Future} for the sake
 * of cancellability but not provide a usable result, you can
 * declare types of the form {@code Future<?>} and
 * return {@code null} as a result of the underlying task.
 * Future表示一个异步计算的结果。
 * 提供的方法可以检查计算是否完成，等待计算完成，取回（retrieve）计算结果。
 * 只能使用get方法在计算完成时取回结果，必要时阻塞，直到计算完成（结果准备好）。
 * 通过cancel方法执行（performed）取消。
 * 提供了额外的（additional）方法支持判断任务是正常完成还是被取消了。
 * 一旦计算完成，那么计算就不能被取消了。
 * 如果只是为了可取消性而想使用Future，不想提供可用的结果，可以声明Future<?>这种类型的形式，返回null作为底层任务执行结果
 *
 * <p>
 * <b>Sample Usage</b> (Note that the following classes are all
 * made-up.)
 * 简单用例（下面的类都是虚构的（made-up））
 * <pre> {@code
 * interface ArchiveSearcher { String search(String target); }
 * class App {
 *   ExecutorService executor = ...
 *   ArchiveSearcher searcher = ...
 *   void showSearch(final String target)
 *       throws InterruptedException {
 *     Future<String> future                                          // -- 带有submit的方法start
 *       = executor.submit(new Callable<String>() {
 *         public String call() {
 *             return searcher.search(target);
 *         }});                                                       // -- 带有submit的方法end
 *     displayOtherThings(); // do other things while searching
 *     try {
 *       displayText(future.get()); // use future
 *     } catch (ExecutionException ex) { cleanup(); return; }
 *   }
 * }}</pre>
 *
 * The {@link FutureTask} class is an implementation of {@code Future} that
 * implements {@code Runnable}, and so may be executed by an {@code Executor}.
 * FutureTask类是Future的实现类，实现了Runnable接口，可以被Executor执行。
 * （因为Executor接口只有一个execute方法，执行传入的Runnable类型的对象）
 *
 * For example, the above construction with {@code submit} could be replaced by:
 * 例如，上面带有submit结构的代码，可以改成这个样子（这个样例看不出来改FutureTask的必要）
 *  <pre> {@code
```

```
 * FutureTask<String> future =
 *   new FutureTask<String>(new Callable<String>() {
 *     public String call() {
 *       return searcher.search(target);
 *   }});
 * executor.execute(future);}</pre>
 *
 * <p>Memory consistency effects: Actions taken by the asynchronous computation
 * <a href="package-summary.html#MemoryVisibility"> <i>happen-before</i></a>
 * actions following the corresponding {@code Future.get()} in another thread.
 * 内存一致性的影响
 * （跟以前的一样，不太明白这块的含义，这次不直译了）
 *
 * @see FutureTask
 * @see Executor
 * @since 1.5
 * @author Doug Lea
 * @param <V> The result type returned by this Future's {@code get} method
 */
public interface Future<V> {

    /**
     * Attempts to cancel execution of this task.  This attempt will
     * fail if the task has already completed, has already been cancelled,
     * or could not be cancelled for some other reason. If successful,
     * and this task has not started when {@code cancel} is called,
     * this task should never run.  If the task has already started,
     * then the {@code mayInterruptIfRunning} parameter determines
     * whether the thread executing this task should be interrupted in
     * an attempt to stop the task.
     * 尝试取消该任务执行。
     * 如果有以下情况任意一种发生，该尝试会失败：
     * 1、任务已经执行完毕
     * 2、任务已经被取消
     * 3、由于其他原因任务不能被取消
     * 如果取消成功，若任务在被调用cancel取消之前尚未启动，那么该任务将永远不会启动了（任务还没开始执行就被取消了，那任务就不会启动了）
     * 如果任务已经开始执行了，那么通过mayInterruptIfRunning参数决定，是否通过中断执行该任务的线程来尝试停止任务执行。
     *
     * <p>After this method returns, subsequent calls to {@link #isDone} will
     * always return {@code true}.  Subsequent calls to {@link #isCancelled}
     * will always return {@code true} if this method returned {@code true}.
     * 在该方法返回后，后续调用isDone方法将总是返回true。
     * 如果该方法返回true，后续调用isCancelled方法也总是返回true
     *
     * @param mayInterruptIfRunning {@code true} if the thread executing this
     * task should be interrupted; otherwise, in-progress tasks are allowed
     * to complete
```

```
     *        mayInterruptIfRunning参数为true时，执行该任务的线程应该被中断；
     *        除此之外，正在执行的任务被运行执行完毕。
     * @return {@code false} if the task could not be cancelled,
     * typically because it has already completed normally;
     *        返回false，如果任务不能被取消，通常是由于该任务已经正常执行完毕了。
     * {@code true} otherwise
     */
    boolean cancel(boolean mayInterruptIfRunning);

    /**
     * Returns {@code true} if this task was cancelled before it completed
     * normally.
     * 如果该任务在正常完成前被取消，返回true
     *
     * @return {@code true} if this task was cancelled before it completed
     */
    boolean isCancelled();

    /**
     * Returns {@code true} if this task completed.
     *
     * Completion may be due to normal termination, an exception, or
     * cancellation -- in all of these cases, this method will return
     * {@code true}.
     * 完成有以下几种情况：
     * 1、正常结束
     * 2、发生异常
     * 3、被取消
     * 所有这些场景，该方法都会返回true
     *
     * @return {@code true} if this task completed
     */
    boolean isDone();

    /**
     * Waits if necessary for the computation to complete, and then
     * retrieves its result.
     * 必要时等待计算完成，然后返回执行结果
     *
     * @return the computed result
     * @throws CancellationException if the computation was cancelled
     *                               如果在等待时发生取消，抛出CancellationException
     * @throws ExecutionException if the computation threw an
     * exception
     * @throws InterruptedException if the current thread was interrupted
     *                              如果在等待时执行该任务的线程发生中断，抛出InterruptedE
xception
     * while waiting
     */
```

```
    V get() throws InterruptedException, ExecutionException;

    /**
     * Waits if necessary for at most the given time for the computation
     * to complete, and then retrieves its result, if available.
     * 必要时等待，如果在给定的时间内能够计算完成，那么返回结果。
     *
     * @param timeout the maximum time to wait
     * @param unit the time unit of the timeout argument
     * @return the computed result
     * @throws CancellationException if the computation was cancelled
     * @throws ExecutionException if the computation threw an
     * exception
     * @throws InterruptedException if the current thread was interrupted
     * while waiting
     * @throws TimeoutException if the wait timed out
     */
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

FutureTask

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;
import java.util.concurrent.locks.LockSupport;

/**
 * A cancellable asynchronous computation.  This class provides a base
 * implementation of {@link Future}, with methods to start and cancel
 * a computation, query to see if the computation is complete, and
 * retrieve the result of the computation.  The result can only be
 * retrieved when the computation has completed; the {@code get}
```

```
 * methods will block if the computation has not yet completed.  Once
 * the computation has completed, the computation cannot be restarted
 * or cancelled (unless the computation is invoked using
 * {@link #runAndReset}).
 * 支持取消的异步计算。
 * 该类基于Future实现了一些方法，支持启动和取消计算，查询计算是否完成，返回计算结果。
 * 只有在计算完成时，才能返回结果；
 * get方法将阻塞直到计算完成。
 * 一旦计算完成，计算不能重新启动或者取消（除非该计算使用runAndReset调用）
 *
 * <p>A {@code FutureTask} can be used to wrap a {@link Callable} or
 * {@link Runnable} object.  Because {@code FutureTask} implements
 * {@code Runnable}, a {@code FutureTask} can be submitted to an
 * {@link Executor} for execution.
 * FutureTask可以用来包装（wrap）Callable或者Runnable对象。
 * 因为FutureTask实现了Runnable接口，可以用于提交到Executor中执行（通过execute方法执行）。
 *
 * <p>In addition to serving as a standalone class, this class provides
 * {@code protected} functionality that may be useful when creating
 * customized task classes.
 * 除了作为独立使用的类外，该类提供的受保护的方法，在创建自定义任务类时可能很有用。
 *
 * @since 1.5
 * @author Doug Lea
 * @param <V> The result type returned by this FutureTask's {@code get} methods
 * V表示get方法的返回值类型
 */
public class FutureTask<V> implements RunnableFuture<V> {
    /*
     * Revision notes: This differs from previous versions of this
     * class that relied on AbstractQueuedSynchronizer, mainly to
     * avoid surprising users about retaining interrupt status during
     * cancellation races. Sync control in the current design relies
     * on a "state" field updated via CAS to track completion, along
     * with a simple Treiber stack to hold waiting threads.
     * 修订（revision）说明：与之前依赖AQS的版本不同，主要为了避免用户对于在取消竞争时仍保留中断状态而感到惊讶。
     * 在当前设计中同步器依赖通过CAS更新的state字段来跟踪完成状态，以及简单的Treiber栈来保存等待线程。
     * （Treiber Stack Algorithm是一个可扩展的无锁栈，利用细粒度的并发原语CAS来实现的）
     *
     * Style note: As usual, we bypass overhead of using
     * AtomicXFieldUpdaters and instead directly use Unsafe intrinsics.
     * 样式说明：像往常一样，绕过使用AtomicXFieldUpdaters的开销，直接使用Unsafe内部函数。
     *
     */

    /**
     * The run state of this task, initially NEW.  The run state
```

```
     * transitions to a terminal state only in methods set,
     * setException, and cancel.  During completion, state may take on
     * transient values of COMPLETING (while outcome is being set) or
     * INTERRUPTING (only while interrupting the runner to satisfy a
     * cancel(true)). Transitions from these intermediate to final
     * states use cheaper ordered/lazy writes because values are unique
     * and cannot be further modified.
     * 任务的执行状态（state），初始化是NEW。
     * 运行状态仅在set、setException、cancel方法中会转变为终止（terminal）状态。
     * 在完成期间，状态可能采用COMPLIETING（在设置结果时）或者INTERRUPTING（仅在中断运行程序以满足取消为true时）这种瞬时态。
     * 从这些中间态到最终态的转化，使用cheaper有序/懒惰写入，因为值是唯一的并且无法进一步（further）修改。
     *
     * Possible state transitions:
     * 可能的状态转化
     *
     * NEW -> COMPLETING -> NORMAL
     * NEW -> COMPLETING -> EXCEPTIONAL
     * NEW -> CANCELLED
     * NEW -> INTERRUPTING -> INTERRUPTED
     */
    private volatile int state;
    private static final int NEW          = 0;
    private static final int COMPLETING   = 1;
    private static final int NORMAL       = 2;
    private static final int EXCEPTIONAL  = 3;
    private static final int CANCELLED    = 4;
    private static final int INTERRUPTING = 5;
    private static final int INTERRUPTED  = 6;

    /** The underlying callable; nulled out after running */
    // 底层的callable；执行完成后置为null。
    private Callable<V> callable;   // 是需要执行的任务。
    /** The result to return or exception to throw from get() */
    // 通过get()返回的结果或者抛出的异常
    private Object outcome; // non-volatile, protected by state reads/writes // 非volatile，受state的读/写保护
    /** The thread running the callable; CASed during run() */
    // 执行callable的线程；在run()期间CAS控制
    private volatile Thread runner;   // 执行任务的线程
    /** Treiber stack of waiting threads */
    // Treiber栈保存的等待线程
    private volatile WaitNode waiters; // 用来指向第一个等待线程WaitNode，没有的话为null。这里的等待线程，是指的等待获取结果的线程，而不是执行任务的线程

    /**
     * Returns result or throws exception for completed task.
     * 对于执行完成的任务，返回结果或者抛出异常
```

```
     *
     * @param s completed state value
     */
    @SuppressWarnings("unchecked")
    private V report(int s) throws ExecutionException {
        Object x = outcome;
        if (s == NORMAL) // 当state处于NORMAL状态，直接返回执行结果
            return (V)x;
        if (s >= CANCELLED) // 当state处于CANCELLED、INTERRUPTING、INTERRUPTED状态，抛出已取消异常。
            throw new CancellationException();
        throw new ExecutionException((Throwable)x); // 能到这里就剩EXCEPTIONAL了，因为只有state>COMPLETING的才能进入该方法，需要注意，这里把执行结果封装成了Exception给抛出了
                                                    // 构造该异常的流程为：public ExecutionException(Throwable cause)->public Exception(Throwable cause)->public Throwable(Throwable cause)
    }

    /**
     * Creates a {@code FutureTask} that will, upon running, execute the
     * given {@code Callable}.
     * FutureTask构造函数，将在运行时执行给定的Callable。
     *
     * @param  callable the callable task
     * @throws NullPointerException if the callable is null
     */
    public FutureTask(Callable<V> callable) {
        if (callable == null)
            throw new NullPointerException();
        this.callable = callable;
        this.state = NEW;       // ensure visibility of callable
    }

    /**
     * Creates a {@code FutureTask} that will, upon running, execute the
     * given {@code Runnable}, and arrange that {@code get} will return the
     * given result on successful completion.
     * FutureTask构造函数，将在运行时执行给定的Runnable，并且安排（arrange）在成功完成后，get方法返回给定的result。
     *
     * @param runnable the runnable task
     * @param result the result to return on successful completion. If
     * you don't need a particular result, consider using
     * constructions of the form:
     * {@code Future<?> f = new FutureTask<Void>(runnable, null)}
     * result参数表示成功执行后的返回值，如果不需要特定的result，考虑使用这样形式的结构：
     * Future<?> f = new FutureTask<Void>(runnable, null)
     * Void是java.lang.Void，是一个不可实例化的占位符类，用于对void关键字的引用。在这里表示无返回值。
```

```
     *
     * @throws NullPointerException if the runnable is null
     */
    public FutureTask(Runnable runnable, V result) {
        this.callable = Executors.callable(runnable, result); // 使用Executors工具类将Runnable适配为Callable（Executors.callable->Executors#RunnableAdapter类）
        this.state = NEW;       // ensure visibility of callable
    }

    public boolean isCancelled() {
        return state >= CANCELLED; // 包括CANCELLED、INTERRUPTING、INTERRUPTED
    }

    public boolean isDone() {
        return state != NEW; // 不是NEW就算结束（包含瞬时态与其他结束态）
    }

    // 除了将NEW状态转化为INTERRUPTING/CANCELLED
    // 或者对于支持mayInterruptedIfRunning的，将非NEW状态转化为INTERRUPTED
    public boolean cancel(boolean mayInterruptIfRunning) {
          // 如果state=NEW，尝试直接设置state=INTERRUPTING（如果这样设置的话），否则设置state=CANCELLED来表示任务取消。（这里只是设置了状态，具体取消在后面）
        if (!(state == NEW &&
              UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
                  mayInterruptIfRunning ? INTERRUPTING : CANCELLED))) // 如果mayInterruptIfRunning为true，表示通过中断执行该任务的线程来尝试停止任务执行，这里直接设置state=INTERRUPTING
            return false;
        try {    // in case call to interrupt throws exception // 如果调用中断会抛出异常
            if (mayInterruptIfRunning) { // 如果允许通过中断执行该任务的线程来停止任务执行
                try {
                    Thread t = runner; // runner是执行该callable的线程
                    if (t != null)
                        t.interrupt(); // 中断该线程
                } finally { // final state
                    UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED); // 最后，将state置为最终态INTERRUPTED（根据类开始的状态转化关系，只能是从瞬时态INTERRUPTING转化为INTERRUPTED）
                }
            }
        } finally {
            finishCompletion(); // 调用该方法，将所有的等待线程都唤醒与移除。这里runner线程与waitnode线程没有关系，runner是执行任务（callable）的线程，waitnode里的线程是想获取结果的线程

        }
        return true;
    }

    /**
```

```
     * @throws CancellationException {@inheritDoc}
     */
    public V get() throws InterruptedException, ExecutionException {
        int s = state;                    // 1、获取当前执行的状态state
        if (s <= COMPLETING)              // 2、如果state<=COMPLETING，表示当前状态为NEW或者COMPLETING时
            s = awaitDone(false, 0L);     // 3、拿当前的state值，如果当前线程被中断了，直接抛出异常，如果处于完成态（>COMPLETING）返回state值（如果本次加入了waiterNode，需要删除），如果=COMPLETING，那么让出CPU时间等待完成，如果不是完成态，那么park等待
        return report(s);                 // 4、state表示正常结束就返回实际结果outcome，如果是CANCELLED或者两个INTERRUPT，抛出取消异常，如果是EXCEPTIONAL，抛出对应的异常。
    }

    /**
     * @throws CancellationException {@inheritDoc}
     * 带有限时等待的get方法
     */
    public V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException {
        if (unit == null)                 // 如果限时为null，直接抛出异常
            throw new NullPointerException();
        int s = state;
        if (s <= COMPLETING &&
            (s = awaitDone(true, unit.toNanos(timeout))) <= COMPLETING) // 调用等待完成的方法，如果超时返回的state仍是未完成状态，那么就抛出异常
            throw new TimeoutException();
        return report(s);                 // 否则返回对应的执行结果
    }

    /**
     * Protected method invoked when this task transitions to state
     * {@code isDone} (whether normally or via cancellation). The
     * default implementation does nothing.  Subclasses may override
     * this method to invoke completion callbacks or perform
     * bookkeeping. Note that you can query status inside the
     * implementation of this method to determine whether this task
     * has been cancelled.
     * 受保护的方法，当任务转化为isDone状态（state!=NEW）时调用（无论是正常结束还是通过取消）。
     * 默认实现什么都不做。
     * 子类可以覆盖此方法来调用完成时回调方法（callback）或者执行簿记（可能是记录日志的意思？）。
     * 注意，可以在该方法的实现中查询状态，以确定该任务是否已经取消。
     * 这个可以参照ExecutorCompletionService子类，里面有对done方法进行重载，记录完成的task列表
     *
     */
    protected void done() { }
```

```
    /**
     * Sets the result of this future to the given value unless
     * this future has already been set or has been cancelled.
     * 设置该future的结果为给定的值，除非该future已经被设置过或者已经被取消。
     *
     * <p>This method is invoked internally by the {@link #run} method
     * upon successful completion of the computation.
     * 当计算成功结束的时候，该方法由run方法内部调用。
     *
     * @param v the value
     */
    protected void set(V v) {
        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) { // 将state
从NEW更新为COMPLETING（如果已经设置过了，那么就不会重复设置）
            outcome = v;
            UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state // 用NORMAL
覆盖瞬时态
            finishCompletion();                                                    // 释放所
有阻塞等待执行结果的等待线程（基本都在awaitDone方法上阻塞了），好让他们拿到结果返回。
        }
    }

    /**
     * Causes this future to report an {@link ExecutionException}
     * with the given throwable as its cause, unless this future has
     * already been set or has been cancelled.
     * 该future使用给定的throwable作为原因上报ExecutionException，除非该future已经被设置过
或者被取消。
     *
     * <p>This method is invoked internally by the {@link #run} method
     * upon failure of the computation.
     * 当计算失败的时候，该方法由run方法内部调用。
     *
     * @param t the cause of failure // t表示失败原因
     */
    protected void setException(Throwable t) {
        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) { // 将state
从NEW更新为COMPLETING（如果已经设置过了，那么就不会重复设置）
            outcome = t;                                                           // 设置结
果为异常原因
            UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); // final state // 用E
XCEPTIONAL覆盖瞬时态
            finishCompletion();                                                    // 释放所
有阻塞等待执行结果的等待线程
        }
    }

    // run()是不返回结果的，结果需要通过get()方法获取
    public void run() {
```

```
        if (state != NEW ||
            !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                         null, Thread.currentThread())) // 如果state不是NEW或者设置runner为当前线程失败，直接返回（有其他线程抢着做runner）
            return;
        try {
            Callable<V> c = callable;
            if (c != null && state == NEW) { // 再次判断state
                V result;
                boolean ran;
                try {
                    result = c.call(); // 等待执行完成
                    ran = true;
                } catch (Throwable ex) {
                    result = null;
                    ran = false;
                    setException(ex); // 将outcome设置为对应的异常
                }
                if (ran)
                    set(result); // 将outcome设置为执行结果
            }
        } finally {
            // runner must be non-null until state is settled to
            // prevent concurrent calls to run()
            // runner必须不为null，直到state稳定（settled 固定的），以防止（prevent）并发调用run()方法。（是null的话run方法就可被并发调用执行）
            runner = null;
            // state must be re-read after nulling runner to prevent
            // leaked interrupts
            // 将runner重置为null后，必须重新读取state，以防止泄露中断。（就是执行了上面那一步之后必须重新读取state）
            int s = state;
            if (s >= INTERRUPTING)
                handlePossibleCancellationInterrupt(s); // 如果进finally是因为cancel(true)引起的中断，那么等待中断完成。
        }
    }

    /**
     * Executes the computation without setting its result, and then
     * resets this future to initial state, failing to do so if the
     * computation encounters an exception or is cancelled.  This is
     * designed for use with tasks that intrinsically execute more
     * than once.
     * 执行计算多次但不设置结果，然后重置future为初识状态，
     * 如果计算遇到（encounter）异常或者被取消，则无法这么做。（无法重置为初始值）
     * 设计用于本质上（intrinsically）执行多次的任务。
     *
     * @return {@code true} if successfully run and reset
```

```
     */
    protected boolean runAndReset() {
        if (state != NEW ||
            !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                         null, Thread.currentThread())) // 如果state不是NEW或者设置runner为当前线程失败，直接返回（有其他线程抢着做runner）
            return false;
        boolean ran = false;
        int s = state;
        try {
            Callable<V> c = callable;
            if (c != null && s == NEW) {
                try {
                    c.call(); // don't set result // 不设置结果
                    ran = true;
                } catch (Throwable ex) {
                    setException(ex);
                }
            }
        } 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
            s = state;
            if (s >= INTERRUPTING)
                handlePossibleCancellationInterrupt(s);
        }
        return ran && s == NEW; // 如果执行成功，返回true（完成且没发生异常，异常有两个方面，一个是执行c.call的时候发生异常，一个是当前runner被cancel的异常）
    }

    /**
     * Ensures that any interrupt from a possible cancel(true) is only
     * delivered to a task while in run or runAndReset.
     * 确保可能来自cancel(true)的任何中断，仅在任务处于run或者runAndReset时传递给任务。
     * 只有一个地方会设置state为INTERRUPTING状态，就是调用cancel(true)的时候。
     * 调用cancel(true)，如果满足state==NEW，那么就设置为INTERRUPTING，直到runner.interrupt()执行完毕才设置INTERRUPTING状态为INTERRUPTED状态。
     *
     */
    private void handlePossibleCancellationInterrupt(int s) {
        // It is possible for our interrupter to stall before getting a
        // chance to interrupt us.  Let's spin-wait patiently.
        // 可能我们的interrupter在获取机会中断我们之前会停止，只需要耐心的（patiently）自旋等待。
        //
        if (s == INTERRUPTING)
```

```
            while (state == INTERRUPTING)
                Thread.yield(); // wait out pending interrupt

        // assert state == INTERRUPTED;

        // We want to clear any interrupt we may have received from
        // cancel(true).  However, it is permissible to use interrupts
        // as an independent mechanism for a task to communicate with
        // its caller, and there is no way to clear only the
        // cancellation interrupt.
        // 我们想清除通过cancel(true)可能获得的任何中断。
        // 然而，允许使用中断作为任务与其调用者之间的通信（communicate）的独立机制（java的基础机制），
        // 所以没办法仅取消cancel中断。
        //
        // Thread.interrupted();
    }

    /**
     * Simple linked list nodes to record waiting threads in a Treiber
     * stack.  See other classes such as Phaser and SynchronousQueue
     * for more detailed explanation.
     * 简单的链表节点，用于记录Treiber stack里等待线程。
     * 有关更详细的说明，可以参阅其他类，例如Phaser与SynchronousQueue
     *
     */
    static final class WaitNode {
        volatile Thread thread; // 保存当前线程（当前线程是一个等待获取执行结果的线程，不是执行任务的线程）
        volatile WaitNode next; // 用于指向下一个等待线程WaitNode
        WaitNode() { thread = Thread.currentThread(); } // 将线程封装成WaitNode
    }

    /**
     * Removes and signals all waiting threads, invokes done(), and
     * nulls out callable.
     * 移除和唤醒所有等待（想要拿到执行结果的）线程，调用done()方法，并将调用对象设置为null。
     *
     */
    private void finishCompletion() {
        // assert state > COMPLETING; // 这个assert本来就注释了
        for (WaitNode q; (q = waiters) != null;) { // 遍历Treiber stack结构的等待线程队列（其实是个栈）（这里是不断的从第一个WaitNode（头）开始遍历）
            if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) { // 将q置为null
                for (;;) {
                    Thread t = q.thread;
                    if (t != null) {
                        q.thread = null;
```

```
                        LockSupport.unpark(t); // 唤醒当前WaitNode的线程
                    }
                    WaitNode next = q.next;
                    if (next == null) // 如果该节点的next为null，退出本层循环（说明这一次对等待线程的遍历完成了，已唤醒所有等待线程）
                        break;
                    q.next = null; // unlink to help gc // 将next打断，方便GC
                    q = next; // 让q成为下一个waitNode，继续遍历
                }
                break;
            }
        }

        done(); // 本类是空，子类可以根据需要自定义

        callable = null;        // to reduce footprint // 减少足迹？？？
    }

    /**
     * Awaits completion or aborts on interrupt or timeout.
     * 等待完成，或者由于中断或者超时导致的终止。
     *
     * @param timed true if use timed waits
     *        timed true 如果使用限时等待
     * @param nanos time to wait, if timed
     *        nanos 时间值 如果需要限时等待
     * @return state upon completion
     *         完成（或者超时）后返回状态（中断抛出异常）
     */
    private int awaitDone(boolean timed, long nanos)
        throws InterruptedException {
        final long deadline = timed ? System.nanoTime() + nanos : 0L;       // 1、如果需要限时等待，计算截止时间
        WaitNode q = null;                                                  // 2、声明一个等待线程节点，等待执行结果
        boolean queued = false;                                             // queued表示当前等待节点是否已在Treiber stack等待队列（其实是个栈）里
        for (;;) {
            if (Thread.interrupted()) {                                     // 如果当前线程发生中断，移除等待结果的线程节点，抛出异常
                removeWaiter(q);                                            // 为什么要做这个呢？因为q可能在等待栈里，要从等待栈里给移除
                throw new InterruptedException();
            }

            int s = state;
            if (s > COMPLETING) {                                           // 如果state状态成为完成态（包含完成、中断、取消）
                if (q != null)                                              //     如果
```

```
等待节点不为null，将等待节点的线程置为null（相当于打个标记，为以后removeWaiter()的时候可以删除，当然本次是不会去调用removeWaiter()了）
                    q.thread = null;
                return s;                                                    // 返回执行状态
            }
            else if (s == COMPLETING) // cannot time out yet
                Thread.yield();                                              // 处在完成的瞬时态，提示调度器当前线程可以暂时放弃CPU调度，等再被调度时不会走进park分支，直接判断state是否为完成态。
            else if (q == null)
                q = new WaitNode();                                          // 第一轮没有拿到完成态，创建一个等待节点
            else if (!queued)                                                // 如果等待节点没在等待队列（其实是个栈）中，尝试入队
                queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                     q.next = waiters, q);     // 新入栈的在栈顶，让当前节点next指向原栈顶元素，然后将当前节点入栈
            else if (timed) {                                                // 如果需要限时，判断是否等待超时，超时的等待将从等待栈中移除
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L) {
                    removeWaiter(q);
                    return state;
                }
                LockSupport.parkNanos(this, nanos);                          // 需要限时的如果没超时，设置park时间开始阻塞
            }
            else
                LockSupport.park(this);                                      // 这一遍检查发现任务没有完成，开始park阻塞（什么时候会唤醒呢？？？）
        }
    }

    /**
     * Tries to unlink a timed-out or interrupted wait node to avoid
     * accumulating garbage.  Internal nodes are simply unspliced
     * without CAS since it is harmless if they are traversed anyway
     * by releasers.  To avoid effects of unsplicing from already
     * removed nodes, the list is retraversed in case of an apparent
     * race.  This is slow when there are a lot of nodes, but we don't
     * expect lists to be long enough to outweigh higher-overhead
     * schemes.
     * 尝试取消超时或者中断的等待节点的链接，避免垃圾积累。
     * 内部节点在没有CAS的情况下只是（simply 只是，简单的）拆开，因为无论发布者怎么遍历他们都是无害的。
     * 为了避免从已经移除的节点拆开的影响，如果有明显竞争的情况下进行列表回溯。
     * 当有太多节点的时候会很慢，但不希望列表足够长以超过更高开销的方案。
     * （就是通过取消无用的节点，来缩短列表长度，同时避免由于已移除的节点导致连接断了，虽然缩短
```

```
的过程中也会带来时间消耗）
     */
    private void removeWaiter(WaitNode node) {
        if (node != null) {
            node.thread = null;
            retry:
            for (;;) {          // restart on removeWaiter race // 在有竞争时重新开始
                for (WaitNode pred = null, q = waiters, s; q != null; q = s) { // 开始时q是Treiber等待栈的头元素
                    s = q.next;
                    if (q.thread != null) // 该node的等待线程不为null，那么移动pred指向该node（表示该node不用删除）
                        pred = q;
                    else if (pred != null) { // 如果该node的等待线程为null，那么删除该node（具体操作为让该node的前驱.next指向该node的后继）
                        pred.next = s;
                        if (pred.thread == null) // check for race // 如果发现前驱的thread也是null了，说明前驱也应该删除了，那么就重新遍历等待栈。
                            continue retry;
                    }
                    // 到这里的条件是q.thread == null && pred == null，这表示pred就没找到个不是null的并且q是头节点（其实q不是开始的头结点，是不断通过下面的CAS设置的头结点（原来的头结点就顺带删了））
                    else if (!UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                          q, s)) // 用头结点的next替换头结点，如果替换失败了，说明可能有竞争入队的，就从头重新遍历
                        continue retry;
                }
                break;
            }
        }
    }

    // Unsafe mechanics
    // 一堆使用Unsafe实现的CAS
    private static final sun.misc.Unsafe UNSAFE;
    private static final long stateOffset;
    private static final long runnerOffset;
    private static final long waitersOffset;
    static {
        try {
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class<?> k = FutureTask.class;
            stateOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("state"));
            runnerOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("runner"));
            waitersOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("waiters"));
```

```
        } catch (Exception e) {
            throw new Error(e);
        }
    }

}
```

RunnableFuture

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;

/**
 * A {@link Future} that is {@link Runnable}. Successful execution of
 * the {@code run} method causes completion of the {@code Future}
 * and allows access to its results.
 * Runnable形式的Future。
 * run方法的成功执行会导致Future完成，并允许通过Future访问执行结果。
 *
```

```
 * @see FutureTask
 * @see Executor
 * @since 1.6
 * @author Doug Lea
 * @param <V> The result type returned by this Future's {@code get} method
 */
public interface RunnableFuture<V> extends Runnable, Future<V> {
    /**
     * Sets this Future to the result of its computation
     * unless it has been cancelled.
     * 设置该Future为计算结果，除非被取消
     */
    void run();
}
```

ThreadPoolExecutor

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent;
import java.util.concurrent.locks.AbstractQueuedSynchronizer;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.*;

/**
 * An {@link ExecutorService} that executes each submitted task using
```

```
 * one of possibly several pooled threads, normally configured
 * using {@link Executors} factory methods.
 * 一个ExecutorService，使用可能的几个线程池之一执行提交的任务，
 * 通常使用Executors工厂方法来配置。
 * （通常用Executors工厂方法来生成该类实例，比如给定线程池、线程数等，不过现在不推荐使用）
 *
 * <p>Thread pools address two different problems: they usually
 * provide improved performance when executing large numbers of
 * asynchronous tasks, due to reduced per-task invocation overhead,
 * and they provide a means of bounding and managing the resources,
 * including threads, consumed when executing a collection of tasks.
 * Each {@code ThreadPoolExecutor} also maintains some basic
 * statistics, such as the number of completed tasks.
 * 线程池解决（address）两个不同的问题：
 * 1、通过减少每个任务的调用开销，通常在执行大量异步任务时提供高效（improved）性能（performan
ce）。
 * 2、提供限制（bounding）和管理资源的方法，包括执行任务集合时消耗的线程。
 * 每个ThreadPoolExecutor也维护（maintain）一些基本的统计信息，例如已完成任务的数量。
 *
 * <p>To be useful across a wide range of contexts, this class
 * provides many adjustable parameters and extensibility
 * hooks. However, programmers are urged to use the more convenient
 * {@link Executors} factory methods {@link
 * Executors#newCachedThreadPool} (unbounded thread pool, with
 * automatic thread reclamation), {@link Executors#newFixedThreadPool}
 * (fixed size thread pool) and {@link
 * Executors#newSingleThreadExecutor} (single background thread), that
 * preconfigure settings for the most common usage
 * scenarios. Otherwise, use the following guide when manually
 * configuring and tuning this class:
 * 为了在广泛的上下文中有用，该类提供了许多可调整的参数和可扩展的钩子方法（hooks）。
 * 但是，强烈建议（urge 敦促）程序员使用更方便的（convenient）Executors提供的工厂方法：
 * Executors#newCachedThreadPool：无界线程池，具有自动线程回收（reclamation）
 * Executors#newFixedThreadPool：固定大小线程池
 * Executors#newSingleThreadExecutor：单个后台线程
 * 他们预先配置了最常见的使用场景（scenarios）。
 * 否则，在手工（manually）配置与调整（tuning）此类时，使用如下指南：
 *
 * <dl>
 *
 * <dt>Core and maximum pool sizes</dt>
 * 核心和最大（池中）线程数
 *
 * <dd>A {@code ThreadPoolExecutor} will automatically adjust the
 * pool size (see {@link #getPoolSize})
 * according to the bounds set by
 * corePoolSize (see {@link #getCorePoolSize}) and
 * maximumPoolSize (see {@link #getMaximumPoolSize}).
 * ThreadPoolExecutor会依据corePoolSize（getCorePoolSize）与maximumPoolSize（getMaximumP
```

```
oolSize）设置的边界，
 * 自动调整池大小（用getPoolSize查询）
 *
 * When a new task is submitted in method {@link #execute(Runnable)},
 * and fewer than corePoolSize threads are running, a new thread is
 * created to handle the request, even if other worker threads are
 * idle.  If there are more than corePoolSize but less than
 * maximumPoolSize threads running, a new thread will be created only
 * if the queue is full.  By setting corePoolSize and maximumPoolSize
 * the same, you create a fixed-size thread pool. By setting
 * maximumPoolSize to an essentially unbounded value such as {@code
 * Integer.MAX_VALUE}, you allow the pool to accommodate an arbitrary
 * number of concurrent tasks. Most typically, core and maximum pool
 * sizes are set only upon construction, but they may also be changed
 * dynamically using {@link #setCorePoolSize} and {@link
 * #setMaximumPoolSize}. </dd>
 * 当使用execute(Runnable)提交任务，
 * 1、如果当前运行的线程数小于corePoolSize时，新的线程被创建来处理该请求（任务），即使其他线程空闲。
 * 2、如果当前运行的线程数大于corePoolSize但小于maximumPoolSize时，只有在队列（等待处理的队列）满的时候才会创建新的线程。
 * 通过设置corePoolSize等于maximumPoolSize，就创建了一个固定大小的线程池。
 * 通过设置maximumPoolSize设置为一个本质上（essentially）无界值（例如Integer.MAX_VALUE），可以允许线程池适应（accommodate 容纳）任意（arbitrary）数量的并发线程。
 * 最典型的是，核心和最大线程池大小只在构造时设置，但也可以视同setCorePoolSize与setMaximumPoolSize来动态更改。
 *
 * <dt>On-demand construction</dt>
 * 按需构建
 *
 * <dd>By default, even core threads are initially created and
 * started only when new tasks arrive, but this can be overridden
 * dynamically using method {@link #prestartCoreThread} or {@link
 * #prestartAllCoreThreads}.  You probably want to prestart threads if
 * you construct the pool with a non-empty queue. </dd>
 * 默认情况下，即使核心线程也仅在新的任务到达时才初始化创建与启动，但可以使用prestartCoreThread或者prestartAllCoreThreads方法动态覆盖（该模式）
 * 如果使用非空队列构建线程池时，可能想预启动线程。
 *
 * <dt>Creating new threads</dt>
 * 创建新线程
 *
 * <dd>New threads are created using a {@link ThreadFactory}.  If not
 * otherwise specified, a {@link Executors#defaultThreadFactory} is
 * used, that creates threads to all be in the same {@link
 * ThreadGroup} and with the same {@code NORM_PRIORITY} priority and
 * non-daemon status. By supplying a different ThreadFactory, you can
 * alter the thread's name, thread group, priority, daemon status,
 * etc. If a {@code ThreadFactory} fails to create a thread when asked
```

```
 * by returning null from {@code newThread}, the executor will
 * continue, but might not be able to execute any tasks. Threads
 * should possess the "modifyThread" {@code RuntimePermission}. If
 * worker threads or other threads using the pool do not possess this
 * permission, service may be degraded: configuration changes may not
 * take effect in a timely manner, and a shutdown pool may remain in a
 * state in which termination is possible but not completed.</dd>
 * 使用ThreadFactory创建新的线程。
 * 如果没有另外指定，使用Executors#defaultThreadFactory，所有创建的线程使用相同的ThreadGrou
p、相同的NORM_PRIORITY优先级和非守护的进程状态。
 * 通过提供不同的ThreadFactory，可以更改线程名、线程组、优先级、守护进程状态等。
 * 如果在调用ThreadFactory的newThread方法返回null时，则创建线程失败，executor将继续运行，但
可能无法执行任何任务。
 * 线程应该拥有modifyThread的RuntimePermission（运行时权限）。
 * 如果使用线程池的worker线程或者其他线程没有拥有该权限，则服务可能会被降级（degrade）：
 * 1、参数修改可能无法及时生效
 * 2、关闭线程池可能停留在可以终止但未完成状态
 *
 * <dt>Keep-alive times</dt>
 * 保持活跃的时间
 *
 * <dd>If the pool currently has more than corePoolSize threads,
 * excess threads will be terminated if they have been idle for more
 * than the keepAliveTime (see {@link #getKeepAliveTime(TimeUnit)}).
 * This provides a means of reducing resource consumption when the
 * pool is not being actively used. If the pool becomes more active
 * later, new threads will be constructed. This parameter can also be
 * changed dynamically using method {@link #setKeepAliveTime(long,
 * TimeUnit)}.  Using a value of {@code Long.MAX_VALUE} {@link
 * TimeUnit#NANOSECONDS} effectively disables idle threads from ever
 * terminating prior to shut down. By default, the keep-alive policy
 * applies only when there are more than corePoolSize threads. But
 * method {@link #allowCoreThreadTimeOut(boolean)} can be used to
 * apply this time-out policy to core threads as well, so long as the
 * keepAliveTime value is non-zero. </dd>
 * 如果当前线程池中线程数超过corePoolSize，多余的线程将在空闲时间超过KeepAliveTime（见getKee
pAliveTime(TimeUnit)）时被终止。
 * 这提供了一种方法，在线程池未充分利用的情况下减少资源的消耗（consumption）。
 * 如果线程池稍后变得活跃（使用频率变高），则将构建新的线程。
 * 该参数可以使用setKeepAliveTime(long, TimeUnit)方法动态修改。
 * 使用Long.MAX_VALUE TimeUnit#NANOSECONDS值可以有效禁止空闲线程在线程池关闭之前被终止。
 * 默认情况下，keep-alive策略仅在有线程超过corePoolSize的时候才会适用。
 * 但是allowCoreThreadTimeOut(boolean)方法，可以用于核心线程的time-out策略，前提是keepAliv
eTime值不为0。
 *（就是默认情况下，keepAliveTime仅能控制非核心线程的存活时间，allowCoreThreadTimeOut方法可
以控制将keepAliveTime用于核心线程）
 *
 * <dt>Queuing</dt>
 * 队列
```

```
 *
 * <dd>Any {@link BlockingQueue} may be used to transfer and hold
 * submitted tasks.  The use of this queue interacts with pool sizing:
 * 任意BlockingQueue都可以用于传递和保存提交的任务。
 * 使用该队列与线程池大小交互如下：
 *
 * <ul>
 *
 * <li> If fewer than corePoolSize threads are running, the Executor
 * always prefers adding a new thread
 * rather than queuing.</li>
 * 如果运行线程数 < corePoolSize，那么Executor总是喜欢（prefer）添加新线程而不是入队。
 *
 * <li> If corePoolSize or more threads are running, the Executor
 * always prefers queuing a request rather than adding a new
 * thread.</li>
 *  如果运行线程数 >= coolPoolSize，Executor总是喜欢将请求入队而不是添加新线程。
 *
 * <li> If a request cannot be queued, a new thread is created unless
 * this would exceed maximumPoolSize, in which case, the task will be
 * rejected.</li>
 * 如果请求无法入队，创建新线程，除非线程数将超过（exceed）maximumPoolSize，在这种情况下，该任务将被拒绝（reject）
 *
 * </ul>
 *
 * There are three general strategies for queuing:
 * 对于队列的三种常见策略：
 *
 * <ol>
 *
 * <li> <em> Direct handoffs.</em> A good default choice for a work
 * queue is a {@link SynchronousQueue} that hands off tasks to threads
 * without otherwise holding them. Here, an attempt to queue a task
 * will fail if no threads are immediately available to run it, so a
 * new thread will be constructed. This policy avoids lockups when
 * handling sets of requests that might have internal dependencies.
 * Direct handoffs generally require unbounded maximumPoolSizes to
 * avoid rejection of new submitted tasks. This in turn admits the
 * possibility of unbounded thread growth when commands continue to
 * arrive on average faster than they can be processed.  </li>
 * Direct handoffs（直接交接/直接握手）。
 * 工作队列一个好的默认选择是SynchronousQueue，将任务提交（hand off）给线程而不用其他方式保留任务。
 * 在这里，如果没有线程可立即执行任务，尝试将任务入队会失败，所以将创建新的线程。
 * 该策略避免锁定，当处理的请求集可能含有内部依赖的时候。
 * Direct handoffs通常需要maximumPoolSize是无界的，以避免新提交的任务被拒绝。
 * 这反过来（turn）又承认（admit）了存在这种可能性：当commands持续到达的平均速度比处理速度快时，线程将无限增长。
```

```
 *
 * <li><em> Unbounded queues.</em> Using an unbounded queue (for
 * example a {@link LinkedBlockingQueue} without a predefined
 * capacity) will cause new tasks to wait in the queue when all
 * corePoolSize threads are busy. Thus, no more than corePoolSize
 * threads will ever be created. (And the value of the maximumPoolSize
 * therefore doesn't have any effect.)  This may be appropriate when
 * each task is completely independent of others, so tasks cannot
 * affect each others execution; for example, in a web page server.
 * While this style of queuing can be useful in smoothing out
 * transient bursts of requests, it admits the possibility of
 * unbounded work queue growth when commands continue to arrive on
 * average faster than they can be processed.  </li>
 * Unbounded queues（无界队列）
 * 使用无界队列（例如没有预先设置容量的LinkedBlockingQueue）将导致在所有corePoolSize的线程都忙的时候，新任务在队列中等待。
 * 因此，创建的线程数不会超过corePoolSize。（并且，maximumPoolSize值因此不会有任何影响（没啥用））
 * 这可能适用于：每个任务完全互相独立，因此每个任务不会影响彼此的执行；
 * 例如，在网页服务中，尽管这种风格的队列对于平滑处理突发请求很有用，但在commands持续到达平均速度超过服务处理速度时，会导致工作队列无限增长。
 *
 * <li><em>Bounded queues.</em> A bounded queue (for example, an
 * {@link ArrayBlockingQueue}) helps prevent resource exhaustion when
 * used with finite maximumPoolSizes, but can be more difficult to
 * tune and control.  Queue sizes and maximum pool sizes may be traded
 * off for each other: Using large queues and small pools minimizes
 * CPU usage, OS resources, and context-switching overhead, but can
 * lead to artificially low throughput.  If tasks frequently block (for
 * example if they are I/O bound), a system may be able to schedule
 * time for more threads than you otherwise allow. Use of small queues
 * generally requires larger pool sizes, which keeps CPUs busier but
 * may encounter unacceptable scheduling overhead, which also
 * decreases throughput.  </li>
 * Bounded queues（有界队列）
 * 有界队列（例如ArrayBlockingQueue）当与有限的（finite）maximumPoolSize一起使用时，有助于避免资源耗尽（exhaustion），但可能难以调整和控制。
 * 队列大小和最大线程池大小可能互相影响（traded off 折中）：
 * 使用大队列和小线程池数将最大限度的减少CPU使用率、OS资源和上下文切换开销，但是会认为的（artificially）降低吞吐量。
 * 如果任务频繁（frequently）阻塞（例如受限于I/O），跟你允许的线程数比，系统可能能够安排时间给更多的线程。
 * （就是对于I/O密集型的任务，那么系统可以设置更多的线程来使用CPU等资源，如果设置的线程池数过小，就会导致吞吐量降低）
 * 使用小队列通常需要大线程池数，用于保持CPU繁忙，但是可能会遇到不可接受的调度开销，这也会降低吞吐量。
 * （就是对于CPU密集型任务，如果线程过多，会频繁发生线程调用上下文切换，额外的调度开销）
 *
 * </ol>
```

```
 *
 * </dd>
 *
 * <dt>Rejected tasks</dt>
 * 拒绝任务
 *
 * <dd>New tasks submitted in method {@link #execute(Runnable)} will be
 * <em>rejected</em> when the Executor has been shut down, and also when
 * the Executor uses finite bounds for both maximum threads and work queue
 * capacity, and is saturated.  In either case, the {@code execute} method
 * invokes the {@link
 * RejectedExecutionHandler#rejectedExecution(Runnable, ThreadPoolExecutor)}
 * method of its {@link RejectedExecutionHandler}.  Four predefined handler
 * policies are provided:
 * 通过execute(Runnable)方法提交新任务可能被拒绝，1、当Executor已经被关闭，2、或者当Executor
在最大线程数与工作队列容量使用有界限制达到饱和（saturated）。
 * 无论哪种场景，execute方法都会调用RejectedExecutionHandler的RejectedExecutionHandle#rej
ectedException(Runnable, ThreadPoolExecutor)方法。
 * 支持四种预定义的handler策略：
 *
 * <ol>
 *
 * <li> In the default {@link ThreadPoolExecutor.AbortPolicy}, the
 * handler throws a runtime {@link RejectedExecutionException} upon
 * rejection. </li>
 * （本任务丢弃策略，抛出异常）
 * 默认策略为ThreadPoolExecutor.AbortPolicy，该处理器在拒绝时抛出运行时RejectedExecutionEx
ception
 *
 * <li> In {@link ThreadPoolExecutor.CallerRunsPolicy}, the thread
 * that invokes {@code execute} itself runs the task. This provides a
 * simple feedback control mechanism that will slow down the rate that
 * new tasks are submitted. </li>
 * （调用者执行策略）
 * 在ThreadPoolExecutor.CallerRunsPolicy，调用execute的线程自己执行该任务（被线程池拒绝的任
务）。
 * 这提供了一种简单的反馈（feedback）控制机制，可以减慢提交新任务的速度（rate）。
 *
 * <li> In {@link ThreadPoolExecutor.DiscardPolicy}, a task that
 * cannot be executed is simply dropped.  </li>
 * （本任务直接丢弃策略）
 * 在ThreadPoolExecutor.DiscardPolicy，无法执行的任务只是简单的丢弃。（不执行也不抛出异常）
 * （discard 丢弃）
 *
 * <li>In {@link ThreadPoolExecutor.DiscardOldestPolicy}, if the
 * executor is not shut down, the task at the head of the work queue
 * is dropped, and then execution is retried (which can fail again,
 * causing this to be repeated.) </li>
 * （最老任务丢弃策略）
```

```
 * 在ThreadPoolExecutor.DiscardOldestPolicy，如果执行器没有被关闭，在工作队列队首的任务被抛弃，然后重试（retried）执行（有可能再次失败，导致该操作不断重复执行）
 *
 * </ol>
 *
 * It is possible to define and use other kinds of {@link
 * RejectedExecutionHandler} classes. Doing so requires some care
 * especially when policies are designed to work only under particular
 * capacity or queuing policies. </dd>
 * 可以定义和使用其他类型的RejectedExceptionHandler类。
 * 这样做需要谨慎，尤其是当策略设计仅在特定（particular）容量或者排队策略下工作时。
 *
 * <dt>Hook methods</dt>
 * 钩子函数
 *
 * <dd>This class provides {@code protected} overridable
 * {@link #beforeExecute(Thread, Runnable)} and
 * {@link #afterExecute(Runnable, Throwable)} methods that are called
 * before and after execution of each task.  These can be used to
 * manipulate the execution environment; for example, reinitializing
 * ThreadLocals, gathering statistics, or adding log entries.
 * Additionally, method {@link #terminated} can be overridden to perform
 * any special processing that needs to be done once the Executor has
 * fully terminated.
 * 该类提供受保护的可覆盖的beforeExecute(Thread, Runnable)和afterExecute(Runnable, Throwable)方法，在每个任务执行之前与之后被调用。
 * 这些可以用于操作（manipulate）执行环境；例如，重新初始化Threadlocals、收集（gather）统计信息、或者添加日志条目（entries）。
 * 此外，可以覆盖terminated方法，在Executor完全（fully）终止后，执行（perform）需要做的特殊处理。
 * （提及三个钩子方法：beforeExecute、afterExecute、terminated）
 *
 * <p>If hook or callback methods throw exceptions, internal worker
 * threads may in turn fail and abruptly terminate.</dd>
 * 如果hook或者回调（callback）方法抛出异常，内部worker线程可能反过来失败并且突然（abruptly）终止。
 * （就是钩子方法或者回调方法抛出异常的话，会影响该worker线程失败与终止）
 *
 * <dt>Queue maintenance</dt>
 * 队列维护
 *
 * <dd>Method {@link #getQueue()} allows access to the work queue
 * for purposes of monitoring and debugging.  Use of this method for
 * any other purpose is strongly discouraged.  Two supplied methods,
 * {@link #remove(Runnable)} and {@link #purge} are available to
 * assist in storage reclamation when large numbers of queued tasks
 * become cancelled.</dd>
 * getQueue()方法允许访问该线程队列，用于监控和debug目的。
 * 强烈建议不要（discouraged 不建议）将该方法用于其他目的。
```

```
 * 提供的两个方法remove(Runnable)和purge，在大量入队的任务被取消时，可用于协助存储回收（reclamation）。
 *
 * <dt>Finalization</dt>
 * 定稿？？？
 *
 * <dd>A pool that is no longer referenced in a program <em>AND</em>
 * has no remaining threads will be {@code shutdown} automatically. If
 * you would like to ensure that unreferenced pools are reclaimed even
 * if users forget to call {@link #shutdown}, then you must arrange
 * that unused threads eventually die, by setting appropriate
 * keep-alive times, using a lower bound of zero core threads and/or
 * setting {@link #allowCoreThreadTimeOut(boolean)}.  </dd>
 * 程序中不再被引用并且没有剩余（remain）线程的线程池将被自动关闭。
 * 如果你想确保未引用的线程池被回收，即使使用者忘记调用shutdown，
 * 那么你必须安排未使用的线程最终（eventually）死亡，通过设置适当的（appropriate）keep-alive时间、使用0核心线程数的下限并且/或者设置allowCoreThreadTimeOut(boolean)。
 * （corePoolSize为0时，allowCoreThreadTimeOut这个方法可设可不设，corePoolSize不为0时，必须设）
 * （如果线程池中没有存活线程，并且线程池没有被引用，那么线程池对象就会被自动回收）
 *
 * </dl>
 *
 * <p><b>Extension example</b>. Most extensions of this class
 * override one or more of the protected hook methods. For example,
 * here is a subclass that adds a simple pause/resume feature:
 * 扩展样例。大多数该类的扩展类都会覆盖一个或多个受保护的hook方法。
 * 例如，下面这个子类增加了一个简单的暂停/恢复功能（feature 特性）。
 *
 *  <pre> {@code
 * class PausableThreadPoolExecutor extends ThreadPoolExecutor {
 *   private boolean isPaused;
 *   private ReentrantLock pauseLock = new ReentrantLock();  // 看着陌生的话可以先看看Lock、Condition、AQS、ReentrantLock部分源码
 *   private Condition unpaused = pauseLock.newCondition();
 *
 *   public PausableThreadPoolExecutor(...) { super(...); }
 *
 *   protected void beforeExecute(Thread t, Runnable r) {
 *     super.beforeExecute(t, r);
 *     pauseLock.lock();
 *     try {
 *       while (isPaused) unpaused.await();
 *     } catch (InterruptedException ie) {
 *       t.interrupt();
 *     } finally {
 *       pauseLock.unlock();
 *     }
 *   }
```

```
 *
 *   public void pause() {
 *     pauseLock.lock();
 *     try {
 *       isPaused = true;
 *     } finally {
 *       pauseLock.unlock();
 *     }
 *   }
 *
 *   public void resume() {
 *     pauseLock.lock();
 *     try {
 *       isPaused = false;
 *       unpaused.signalAll();
 *     } finally {
 *       pauseLock.unlock();
 *     }
 *   }
 * }}</pre>
 *
 * @since 1.5
 * @author Doug Lea
 */
public class ThreadPoolExecutor extends AbstractExecutorService {
    /**
     * The main pool control state, ctl, is an atomic integer packing
     * two conceptual fields
     *   workerCount, indicating the effective number of threads
     *   runState,    indicating whether running, shutting down etc
     * ctl，线程池主要的控制状态，是一个原子integer，包含两个概念（conceptual）字段
     *   workerCount，代表有效的线程数
     *   runState，    代表线程池状态是否为运行、关闭等
     *
     * In order to pack them into one int, we limit workerCount to
     * (2^29)-1 (about 500 million) threads rather than (2^31)-1 (2
     * billion) otherwise representable. If this is ever an issue in
     * the future, the variable can be changed to be an AtomicLong,
     * and the shift/mask constants below adjusted. But until the need
     * arises, this code is a bit faster and simpler using an int.
     * 为了将这两个字段打包成一个int，将workerCount限制为(2^29)-1个线程，而不是(2^31)-1个其他可表示的线程。
     * 如果这在将来成为问题，该变量（variable）ctl可以改为AtomicLong类型，并调整下面的移位/掩码常量。（常量指的COUNT_BITS）
     * 但是在需要之前（arises 出现），这段代码使用int会更快更简单。
     *
     * The workerCount is the number of workers that have been
     * permitted to start and not permitted to stop.  The value may be
     * transiently different from the actual number of live threads,
```

```
     * for example when a ThreadFactory fails to create a thread when
     * asked, and when exiting threads are still performing
     * bookkeeping before terminating. The user-visible pool size is
     * reported as the current size of the workers set.
     * workerCount表示已经被允许启动但未允许停止的worker数量。
     * 这个值可能和实际活动线程数暂时（transiently）不一致，（不保证一致是因为workerCount-1
操作与workers.remove(w)不是一个Lock锁下的同步操作）
     * 例如，当ThreadFactory创建线程失败，当退出线程在终止前仍旧执行bookkeeping。
     * 用户可见的线程池大小代表worker集当前大小。
     *
     * The runState provides the main lifecycle control, taking on values:
     * runState提供主要的生命周期控制，取值如下：
     *
     *   RUNNING:  Accept new tasks and process queued tasks
     *             接受新任务并处理队列中的任务
     *   SHUTDOWN: Don't accept new tasks, but process queued tasks
     *             不接受新任务，但处理队列中的任务
     *   STOP:     Don't accept new tasks, don't process queued tasks,
     *             and interrupt in-progress tasks
     *             不接受新任务，不处理队列中的任务
     *             并且中断正在处理的任务
     *   TIDYING:  All tasks have terminated, workerCount is zero,
     *             the thread transitioning to state TIDYING
     *             will run the terminated() hook method
     *             所有任务已经终止，workerCount为0，
     *             线程转化为TIDYING状态（TIDYING 整理）
     *             将执行terminated()钩子方法
     *   TERMINATED: terminated() has completed
     *             terminated()方法执行完毕
     *
     *
     * The numerical order among these values matters, to allow
     * ordered comparisons. The runState monotonically increases over
     * time, but need not hit each state. The transitions are:
     * 该值（指的runState）之间的数字（numerical）顺序很重要（matter），以允许进行有序比较。
     * runState跟随时间单调（monotonically）递增，但不需要命中每个状态（就是不需要按顺序递增
，比如不需要-1、0、1、2、3这样增，可以跳为-1、1这样，只要单调递增就行）
     * 过渡是：
     *
     * RUNNING -> SHUTDOWN
     *    On invocation of shutdown(), perhaps implicitly in finalize()
     *    从RUNNING转化为SHUTDOWN：在调用shutdown()时，可能（perhaps）隐藏（implicitly）在
finalize()方法中（finalize()中会执行shutdown()）
     * (RUNNING or SHUTDOWN) -> STOP
     *    On invocation of shutdownNow()
     *    （RUNNING或者SHUTDOWN）转化为STOP：在调用shutdownNow()的时候（转化为STOP状态会re
move掉队列里的所有任务？？？）
     * SHUTDOWN -> TIDYING
     *    When both queue and pool are empty
```

```
     *    SHUTDOWN转化为TIDYING：当队列与线程池都为空时
     * STOP -> TIDYING
     *    When pool is empty
     *    STOP转化为TIDYING：当线程池为空时
     * TIDYING -> TERMINATED
     *    When the terminated() hook method has completed
     *    TIDYING转化为TERMINATED：当terminated()钩子方法执行完毕
     *
     * Threads waiting in awaitTermination() will return when the
     * state reaches TERMINATED.
     * 在awaitTermination()方法上等待的线程，将在状态到达TERMINATED时返回。
     *
     * Detecting the transition from SHUTDOWN to TIDYING is less
     * straightforward than you'd like because the queue may become
     * empty after non-empty and vice versa during SHUTDOWN state, but
     * we can only terminate if, after seeing that it is empty, we see
     * that workerCount is 0 (which sometimes entails a recheck -- see
     * below).
     * 检测（detecting）从SHUTDOWN到TIDYING的转变并不像你想要的那么直接（straightforward）
,
     * 因为在SHUTDOWN状态期间，队列可能从非空变为空，反之亦然（vice versa），（就是不能保证队
列是否一定为空，因为会变化）
     * 但是我们只能在看到队列为空之后，再检查workerCount也是0时，才终止（有些时候需要（entail
 包含）复查 -- 见下文）
     *
     */
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0)); // 线程池主
要控制状态，初始化为runState=RUNNING，workerCount=0
    private static final int COUNT_BITS = Integer.SIZE - 3;                 // 总位数
    private static final int CAPACITY   = (1 << COUNT_BITS) - 1;            // worker线
程最大容量

    // runState is stored in the high-order bits // runState存储在高位，从RUNNING到TERMI
NATED，值逐渐递增
    private static final int RUNNING    = -1 << COUNT_BITS; // -536870912
    private static final int SHUTDOWN   =  0 << COUNT_BITS; // 0
    private static final int STOP       =  1 << COUNT_BITS; // 536870912
    private static final int TIDYING    =  2 << COUNT_BITS; // 1073741824
    private static final int TERMINATED =  3 << COUNT_BITS; // 1610612736

    // Packing and unpacking ctl // 打包和解包ctl
    private static int runStateOf(int c)     { return c & ~CAPACITY; } // 解析线程池运行
状态
    private static int workerCountOf(int c)  { return c & CAPACITY; }  // worker数量
    private static int ctlOf(int rs, int wc) { return rs | wc; }       // 将runState与w
orkerCount打包成ctl

    /*
     * Bit field accessors that don't require unpacking ctl.
```

```
     * These depend on the bit layout and on workerCount being never negative.
     * 不需要解包ctl的位域访问器
     * 这取决于位布局和workerCount永远不会为负值
     */

    private static boolean runStateLessThan(int c, int s) {
        return c < s;
    }

    private static boolean runStateAtLeast(int c, int s) {
        return c >= s;
    }

    // 判断线程池状态是否为running（c < SHUTDOWN）
    private static boolean isRunning(int c) {
        return c < SHUTDOWN;
    }

    /**
     * Attempts to CAS-increment the workerCount field of ctl.
     * 尝试通过CAS递增ctl里的workerCount值
     *
     */
    private boolean compareAndIncrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect + 1);
    }

    /**
     * Attempts to CAS-decrement the workerCount field of ctl.
     * 尝试通过CAS递减ctl里的workerCount值
     *
     */
    private boolean compareAndDecrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect - 1);
    }

    /**
     * Decrements the workerCount field of ctl. This is called only on
     * abrupt termination of a thread (see processWorkerExit). Other
     * decrements are performed within getTask.
     * 递减ctl里的workerCount值。该方法可以被两个方法调用：
     * 1、processWorkerExit方法，调用条件为：仅在线程突然（abrupt）终止时调用
     * 2、getTask方法：其他递减在getTask时执行。（比如线程池有超过corePoolSize的线程空闲等）
     *
     */
    private void decrementWorkerCount() {
        do {} while (! compareAndDecrementWorkerCount(ctl.get()));
    }
```

```
    /**
     * The queue used for holding tasks and handing off to worker
     * threads.  We do not require that workQueue.poll() returning
     * null necessarily means that workQueue.isEmpty(), so rely
     * solely on isEmpty to see if the queue is empty (which we must
     * do for example when deciding whether to transition from
     * SHUTDOWN to TIDYING).  This accommodates special-purpose
     * queues such as DelayQueues for which poll() is allowed to
     * return null even if it may later return non-null when delays
     * expire.
     * 该队列用于保存任务和将任务移交给worker线程。
     * 不要求workQueue.poll()返回null一定意味着workQueue.isEmpty()，
     * 所以只能（solely）依赖（rely on）isEmpty来检查队列是否为空（例如，当我们需要决定是否将线程池状态从SHUTDOWN转化为TIDYING）。
     * 这样能适应特殊目的的队列，例如DelayQueue，允许poll()返回null（可能没元素，或者元素还没到延迟时间），在稍后延迟到期时返回非null值。
     *
     */
    private final BlockingQueue<Runnable> workQueue;

    /**
     * Lock held on access to workers set and related bookkeeping.
     * While we could use a concurrent set of some sort, it turns out
     * to be generally preferable to use a lock. Among the reasons is
     * that this serializes interruptIdleWorkers, which avoids
     * unnecessary interrupt storms, especially during shutdown.
     * Otherwise exiting threads would concurrently interrupt those
     * that have not yet interrupted. It also simplifies some of the
     * associated statistics bookkeeping of largestPoolSize etc. We
     * also hold mainLock on shutdown and shutdownNow, for the sake of
     * ensuring workers set is stable while separately checking
     * permission to interrupt and actually interrupting.
     * 对设置（操作）workers与关联bookkeeping进行加锁访问。
     * 尽管可以使用某种（some sort）并发集，结果（turn out）证明通常最好使用锁。
     * 其中一个原因是它序列化了interruptIdleWorkers，避免了不必要的中断风暴，尤其是在线程池shutdown的时候。
     * 否则退出线程将同时中断那些尚未中断的线程。（可能n个中断线程并发去中断1个尚未中断的线程）
     * 它还简化了最大线程池数等关联统计bookkeeping。
     * 也在shutdown与shutdownNow的时候持有mainLock，为了确保worker集合在分别检查中断权限和实际中断时保持稳定
     * （几乎所有涉及到对workers这个集合的操作，都要加mainLock锁）
     *
     */
    private final ReentrantLock mainLock = new ReentrantLock();

    /**
     * Set containing all worker threads in pool. Accessed only when
     * holding mainLock.
```

```
     * 集合包含线程池里所有的worker线程。
     * 只有在持有mainLock锁时才能访问。
     *
     */
    private final HashSet<Worker> workers = new HashSet<Worker>();

    /**
     * Wait condition to support awaitTermination
     * 在条件上等待，支持awaitTermination
     * （不好翻译，可以先看看Condition接口源码）
     *
     */
    private final Condition termination = mainLock.newCondition();

    /**
     * Tracks largest attained pool size. Accessed only under
     * mainLock.
     * 跟踪线程池到达的最大线程数。
     * 只有持有mainLock时才能访问。
     *
     */
    private int largestPoolSize;

    /**
     * Counter for completed tasks. Updated only on termination of
     * worker threads. Accessed only under mainLock.
     * 统计已完成任务数。
     * 仅在worker线程结束后更新该值。（可见processWorkerExit方法）
     * 只有持有mainLock才能访问。
     *
     */
    private long completedTaskCount;

    /*
     * All user control parameters are declared as volatiles so that
     * ongoing actions are based on freshest values, but without need
     * for locking, since no internal invariants depend on them
     * changing synchronously with respect to other actions.
     * 所有用户控制的参数都被声明为volatile，以便在进行的动作都基于最新的值，
     * 但不需要加锁，因为没有内部不变量（invariants）依赖其他动作进行同步修改。
     *
     */

    /**
     * Factory for new threads. All threads are created using this
     * factory (via method addWorker).  All callers must be prepared
     * for addWorker to fail, which may reflect a system or user's
     * policy limiting the number of threads.  Even though it is not
     * treated as an error, failure to create threads may result in
```

```
     * new tasks being rejected or existing ones remaining stuck in
     * the queue.
     * 新线程的工厂。所有线程都通过该factory来创建（通过addWorker方法）。
     * 所有调用者都必须准备好对addWorker失败的处理，这可能反映了系统或者使用者限制线程数的策略
。
     * 即使它没有被作为（treat 对待）error，创建线程失败可能导致新的任务被拒绝或者现有任务卡在
队列中。
     *
     * We go further and preserve pool invariants even in the face of
     * errors such as OutOfMemoryError, that might be thrown while
     * trying to create threads.  Such errors are rather common due to
     * the need to allocate a native stack in Thread.start, and users
     * will want to perform clean pool shutdown to clean up.  There
     * will likely be enough memory available for the cleanup code to
     * complete without encountering yet another OutOfMemoryError.
     * 我们更进一步，即使在面临异常例如在创建线程时可能抛出OOM异常，也保留线程池的不变量。
     * 由于在Thread.start时需要从本地栈中分配，因此这类error相当常见，并且用户希望干净的shut
down线程池来进行清理。（清理从栈上分配的线程空间？？？）
     * 可能有足够的内存可用于清理代码完成，而不会遇到其他OOM异常。
     *
     */
    private volatile ThreadFactory threadFactory;

    /**
     * Handler called when saturated or shutdown in execute.
     * 当饱和（saturated 饱和的）或者shutdown时执行的handler。（用于处理那些线程池处理不了的
task）
     *
     */
    private volatile RejectedExecutionHandler handler;

    /**
     * Timeout in nanoseconds for idle threads waiting for work.
     * Threads use this timeout when there are more than corePoolSize
     * present or if allowCoreThreadTimeOut. Otherwise they wait
     * forever for new work.
     * 空闲线程最大等待作业时间。（单位：纳秒）
     * 线程使用此超时：
     * 1、当前线程数超过corePoolSize（核心线程数）
     * 2、允许核心线程超时退出（allowCoreThreadTimeOut）
     * 否则会永久等待新的作业。
     *
     */
    private volatile long keepAliveTime;

    /**
     * If false (default), core threads stay alive even when idle.
     * If true, core threads use keepAliveTime to time out waiting
     * for work.
```

```
     * 如果为false（也是默认值），核心线程即使空闲也会保持活跃（alive）。
     * 如果为true，核心线程在等待作业时使用keepAliveTime设置的限时时间。（超时退出）
     *
     */
    private volatile boolean allowCoreThreadTimeOut;

    /**
     * Core pool size is the minimum number of workers to keep alive
     * (and not allow to time out etc) unless allowCoreThreadTimeOut
     * is set, in which case the minimum is zero.
     * 核心线程池大小是保持活跃（并且不允许超时等）的worker最小值，除非设置了allowCoreThread
TimeOut（设置allowCoreThreadTimeOut的场景下，线程数最小值为0）
     * （注意：0<=corePoolSize<=CAPACITY）
     */
    private volatile int corePoolSize;

    /**
     * Maximum pool size. Note that the actual maximum is internally
     * bounded by CAPACITY.
     * 最大线程池大小。注意，实际上的最大值受CAPACITY内部限制。（maximumCoreSize不能超过CAPA
CITY）
     * （注意：corePoolSize<=maximumPoolSize<=CAPACITY）
     */
    private volatile int maximumPoolSize;

    /**
     * The default rejected execution handler
     * 默认的拒绝执行的处理程序（handler）
     * （当任务被拒绝时，默认采用的拒绝策略是AbortPolicy()，拒绝执行并抛出RejectedExecutionE
xception（extends RuntimeException））
     *
     */
    private static final RejectedExecutionHandler defaultHandler =
        new AbortPolicy();

    /**
     * Permission required for callers of shutdown and shutdownNow.
     * We additionally require (see checkShutdownAccess) that callers
     * have permission to actually interrupt threads in the worker set
     * (as governed by Thread.interrupt, which relies on
     * ThreadGroup.checkAccess, which in turn relies on
     * SecurityManager.checkAccess). Shutdowns are attempted only if
     * these checks pass.
     * 调用者执行shutdown与shutdownNow时需要的权限。
     * 我们还要求（见checkShutdonwAccess）调用者具有实际能够 中断在worker集里的线程 的权限。
     * （由Thread.interrupt管理（governed），它依赖ThreadGroup.checkAccess，后者又依赖于Se
curityManager.checkAccess）。
     * 只有该检查通过，shutdown系列方法才会尝试执行。（尝试关闭）
     *
```

```
     * All actual invocations of Thread.interrupt (see
     * interruptIdleWorkers and interruptWorkers) ignore
     * SecurityExceptions, meaning that the attempted interrupts
     * silently fail. In the case of shutdown, they should not fail
     * unless the SecurityManager has inconsistent policies, sometimes
     * allowing access to a thread and sometimes not. In such cases,
     * failure to actually interrupt threads may disable or delay full
     * termination. Other uses of interruptIdleWorkers are advisory,
     * and failure to actually interrupt will merely delay response to
     * configuration changes so is not handled exceptionally.
     * 所有实际调用Thread.interrupte（见interruptIdleWorkers和interruptWorkers方法）都忽略SecurityExceptions，
     * 这意味着尝试的中断失败时都是静默的。（silently fail反义词可以看做throw Exception）
     * 在shutdown场景下，除非SecurityManage有非一致性策略（有时允许访问线程有时不允许），否则不应该失败。
     * 在这种情况下，未能实际中断线程可能导致不可用或者延迟完全终止。（啥不可用，啥延迟完全终止？？？）
     * interruptIdleWorkers的其他用法是建议性的（advisory），未能实际中断只会（merely）延迟对参数修改的响应，所以不会被异常处理。
     *
     */
    private static final RuntimePermission shutdownPerm =
        new RuntimePermission("modifyThread"); // 修改线程，例如通过调用线程的 interrupt、stop、suspend、resume、setDaemon、setPriority、setName 和 setUncaughtExceptionHandler 方法

    /**
     * Class Worker mainly maintains interrupt control state for
     * threads running tasks, along with other minor bookkeeping.
     * This class opportunistically extends AbstractQueuedSynchronizer
     * to simplify acquiring and releasing a lock surrounding each
     * task execution.  This protects against interrupts that are
     * intended to wake up a worker thread waiting for a task from
     * instead interrupting a task being run.  We implement a simple
     * non-reentrant mutual exclusion lock rather than use
     * ReentrantLock because we do not want worker tasks to be able to
     * reacquire the lock when they invoke pool control methods like
     * setCorePoolSize.  Additionally, to suppress interrupts until
     * the thread actually starts running tasks, we initialize lock
     * state to a negative value, and clear it upon start (in
     * runWorker).
     * Worker类主要用于保存线程运行的任务中断控制状态，和其他次要bookkeeping。
     * 该类取巧继承AQS，来实现围绕每个任务执行获取和释放锁。
     * 可以防止中断用于唤醒等待任务的worker线程，而不是中断正在执行的任务。（是想用中断来唤醒等待的线程而不是中断正在执行的线程？？？）
     * 我们实现了一个简单的不可重入的互斥锁，而不是使用ReentrantLock，
     * 因为我们不想worker任务能够在调用线程池控制方法（像setCorePoolSize）时能够重新获取锁（再次加锁）。
     * 此外，为了在线程实际开始执行任务之前抑制（suppress）中断，我们初始化锁的状态为负值，在
```

```
开始（在runWorker）之后清除它。（可以先看runWorker方法）
     *
     */
    private final class Worker
        extends AbstractQueuedSynchronizer  // 继承AQS
        implements Runnable                 // 实现Runnable
    {
        /**
         * This class will never be serialized, but we provide a
         * serialVersionUID to suppress a javac warning.
         * 该类永远不会被序列化，但是提供了serialVersionUID来抑制javac警告。
         *
         */
        private static final long serialVersionUID = 6138294804551838833L;

        /** Thread this worker is running in.  Null if factory fails. */
        // 运行该worker的线程，如果工厂失败则为null。？？？
        final Thread thread;
        /** Initial task to run.  Possibly null. */
        // 初始化的待运行任务，可能为null（比如观察到任务队列有任务但线程池里没worker了）
        Runnable firstTask;
        /** Per-thread task counter */
        // 每个线程任务计数器
        volatile long completedTasks;

        /**
         * Creates with given first task and thread from ThreadFactory.
         * 构造方法，用给定的任务和从ThreadFactory生成的线程来创建。
         * @param firstTask the first task (null if none)
         */
        Worker(Runnable firstTask) {
            setState(-1); // inhibit interrupts until runWorker // 禁止（inhibit）中断，
直到runWorker。设置的state是AQS的state
            this.firstTask = firstTask;
            this.thread = getThreadFactory().newThread(this); // 将该worker作为线程的执
行对象
        }

        /** Delegates main run loop to outer runWorker  */
        // 主运行循环委托给外部的runWorker（run的时候咋执行由外部的runWorker决定）
        public void run() {
            runWorker(this);
        }

        // Lock methods
        // 锁相关方法
        //
        // The value 0 represents the unlocked state.
        // The value 1 represents the locked state.
```

```
        // 值为0代表解锁（未加锁）状态
        // 值为1代表加所状态

        // isHeldExclusively、tryAcquire、tryRelease是使用AQS实现独占锁需要自己实现的三个方法

        // 当前线程是否持有独占锁
        protected boolean isHeldExclusively() {
            return getState() != 0; // getState()方法是AQS定义的，用来获取当前state
        }

        // 尝试加锁
        protected boolean tryAcquire(int unused) {  // 入参没用到
            if (compareAndSetState(0, 1)) { // 调用AQS#compareAndSetState方法，设置state值（类比ReentrantLock，Worker的state只有0、1两个值）
                setExclusiveOwnerThread(Thread.currentThread()); // 如果加锁成功，更新独占锁的持有者为当前线程
                return true;
            }
            return false;
        }

        // 尝试解锁（解锁必定成功）
        protected boolean tryRelease(int unused) {
            setExclusiveOwnerThread(null);
            setState(0); // 这里没有通过CAS来设置值，直接置为0（需要看为什么如此自信）
            return true;
        }

        public void lock()        { acquire(1); } // AQS#acquire方法，尝试获取锁，如果失败入等待队列阻塞
        public boolean tryLock()  { return tryAcquire(1); } // Worker#tryAcquire
        public void unlock()      { release(1); } // AQS#release方法，释放锁，并尝试从AQS等待队列的head开始唤醒一个可用后继。这个方法会调用tryRelease，所以调用完state变为0了。
        public boolean isLocked() { return isHeldExclusively(); } // Worker#isHeldExclusively

        void interruptIfStarted() {
            Thread t;
            if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) { // 判断执行当前worker的线程是否已启动。state有值，thread不为null，thread没有中断
                try {
                    t.interrupt();
                } catch (SecurityException ignore) { // 捕获异常并忽略
                }
            }
        }
    }
```

```
    /*
     * Methods for setting control state
     * 设置控制状态的方法
     */

    /**
     * Transitions runState to given target, or leaves it alone if
     * already at least the given target.
     * 将runState转化为给定的目标值，如果runState的值已经>=给定的目标值，则不做任何操作。
     *
     * @param targetState the desired state, either SHUTDOWN or STOP
     *        (but not TIDYING or TERMINATED -- use tryTerminate for that)
     *        targetState，所需的状态，只能为SHUTDOWN或者STOP
     *        如果想修改runState为TIDYING或者TERMINATED，那么使用tryTerminate方法
     */
    private void advanceRunState(int targetState) {
        for (;;) {
            int c = ctl.get();
            if (runStateAtLeast(c, targetState) ||
                ctl.compareAndSet(c, ctlOf(targetState, workerCountOf(c))))
                break;
        }
    }

    /**
     * Transitions to TERMINATED state if either (SHUTDOWN and pool
     * and queue empty) or (STOP and pool empty).  If otherwise
     * eligible to terminate but workerCount is nonzero, interrupts an
     * idle worker to ensure that shutdown signals propagate. This
     * method must be called following any action that might make
     * termination possible -- reducing worker count or removing tasks
     * from the queue during shutdown. The method is non-private to
     * allow access from ScheduledThreadPoolExecutor.
     * runState值转化为TERMINATED，当以下两种情况任意一种发生：
     * 1、runState=SHUTDOWN，并且线程池与任务队列都为空
     * 2、runState=STOP，并且线程池为空
     * 如果有资格（eligible）终止但是workerCount不为0，则中断空闲的worker保证关闭信号传播。
     * 必须在可能会导致线程池终止的操作发生后调用该方法，可能的操作包括：
     * 1、在shutdown期间减少worker数量
     * 2、在shutdown期间从任务队列移除任务
     * （就是涉及上面两种操作的方法A可能会导致线程池关闭，那就在A方法之后调用该方法更新runStat
e值为TERMINATED）
     * 该方法是非私有的，允许从ScheduledThreadPoolExecutor来访问。
     *
     */
    final void tryTerminate() { // 要注意该方法目标是要把线程池的runState更新为TERMINATED
        for (;;) {
            int c = ctl.get();
            if (isRunning(c) ||                                          // 如果线程池还在
```

```
运行，
                runStateAtLeast(c, TIDYING) ||                          // 或者状态已经>=TIDYING（包含TIDYING、TERMINATED）（TIDYING状态是钩子方法还没执行完，执行完就成TERMINATED）
                (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty())) // 或者状态=SHUTDOWN但是还有任务没完成
                return;                                                 // 则不能设置runState为TERMINATED
            if (workerCountOf(c) != 0) { // Eligible to terminate      // 走到这一步，runState=SHUTDOWN或者STOP，此时任务队列为空，如果worker数不为0（还有存活的worker线程）
                interruptIdleWorkers(ONLY_ONE);                         // 尝试中断空闲的worker（只中断一个）。（因为现在没任务了并且线程池处于即将关闭状态，空闲线程留着也没用）
                return;
            }

            final ReentrantLock mainLock = this.mainLock;               // 到此runState=SHUTDOWN或者STOP，任务队列与worker都为empty，开始修改runState值
            mainLock.lock();                                            // 获取mainLock
            try {
                if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {          // 尝试将ctl转化成runState=TIDYING，workerCount=0
                    try {
                        terminated();                                   // 调用钩子方法ThreadPoolExecutor#terminated()，本类该方法为空，子类可以按照自己的需求实现
                    } finally {
                        ctl.set(ctlOf(TERMINATED, 0));                  // terminated()钩子方法执行完毕，ctl转化为runState=TERMINATED，workerCount=0
                        termination.signalAll();                        // 唤醒所有在Condition上等待的线程（还没有见到Condition的用处）
                    }
                    return;
                }
            } finally {
                mainLock.unlock();
            }
            // else retry on failed CAS
        }
    }

    /*
     * Methods for controlling interrupts to worker threads.
     * 控制worker线程中断的方法
     *
     */

    /**
     * If there is a security manager, makes sure caller has
     * permission to shut down threads in general (see shutdownPerm).
     * If this passes, additionally makes sure the caller is allowed
     * to interrupt each worker thread. This might not be true even if
```

```
     * first check passed, if the SecurityManager treats some threads
     * specially.
     * 如果有线程管理器，通常需要确保调用者拥有关闭线程的权限（见shutdownPrem）。
     * 如果上面的通过了，另外需要确保允许调用者中断所有worker线程。
     * 即使第一个检查通过了，如果SecurityManager特殊对待某些线程，这也不一定成立。
     *
     */
    private void checkShutdownAccess() {
        SecurityManager security = System.getSecurityManager(); // 如果当前应用创建了安全管理器，那么返回该管理器，否则返回null
        if (security != null) {
            security.checkPermission(shutdownPerm);             // 检查是否有shutdown权限，如果没有抛出基于SecurityException的异常
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();                                    // 加锁是为了防止在遍历workers的时候，worker集合有变化
            try {
                for (Worker w : workers)
                    security.checkAccess(w.thread);             // 逐个检查是否有对每个worker线程的访问权限（具体是MODIFY_THREAD_PERMISSION权限），如果没有抛出基于SecurityException的异常
            } finally {
                mainLock.unlock();
            }
        }
    }

    /**
     * Interrupts all threads, even if active. Ignores SecurityExceptions
     * (in which case some threads may remain uninterrupted).
     * 中断所有线程，即使线程是活跃的。忽略各种SecurityException
     * （在某些情况下，某些线程可能会保持不中断（不响应中断））
     *
     */
    private void interruptWorkers() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers)
                w.interruptIfStarted();                         // 中断处于运行状态的worker线程
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Interrupts threads that might be waiting for tasks (as
     * indicated by not being locked) so they can check for
```

```
     * termination or configuration changes. Ignores
     * SecurityExceptions (in which case some threads may remain
     * uninterrupted).
     * 中断可能正在等待任务的线程（通过没有被lock来表明（如果该worker没有内部加锁，那么就判断为正在等待任务）），
     * 以便他们（至正在等待任务的worker）可以检查终止或者配置变化。
     * 忽略SecurityException（在某些情况下，某些线程可能会保持不中断（不响应中断））
     *
     * @param onlyOne If true, interrupt at most one worker. This is
     * called only from tryTerminate when termination is otherwise
     * enabled but there are still other workers.  In this case, at
     * most one waiting worker is interrupted to propagate shutdown
     * signals in case all threads are currently waiting.
     * Interrupting any arbitrary thread ensures that newly arriving
     * workers since shutdown began will also eventually exit.
     * To guarantee eventual termination, it suffices to always
     * interrupt only one idle worker, but shutdown() interrupts all
     * idle workers so that redundant workers exit promptly, not
     * waiting for a straggler task to finish.
     * 参数onlyOne如果为true，则最多中断一个worker。
     * 只会通过tryTerminate进行这种调用（当其他都可以终止但是仍有worker的时候）（就是除了有worker，其他条件都满足可以终止的时候，才会这样调用该方法）
     * 在这种情况下，所有worker线程当前都处于等待状态，最多一个等待worker被中断来广播shutdown信号。
     * 中断任意线程可以保证自shutdown开始以来，新到达的worker也将最终退出。
     * 为保证最终全部退出，每次只中断一个空闲worker就足够了，（通过多次调用来中断所有worker）
     * 但是shutdown()中断所有空闲线程，以便冗余（redundant）的worker立即（promptly）退出，而不是等待一个落后的（straggler）任务完成。
     *
     */
    private void interruptIdleWorkers(boolean onlyOne) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();                                    // 所有workers操作需要加锁mainLock
        try {
            for (Worker w : workers) {
                Thread t = w.thread;
                if (!t.isInterrupted() && w.tryLock()) {
                    try {
                        t.interrupt();
                    } catch (SecurityException ignore) {
                    } finally {
                        w.unlock();
                    }
                }
                if (onlyOne)
                    break;
            }
        } finally {
```

```
            mainLock.unlock();
        }
    }

    /**
     * Common form of interruptIdleWorkers, to avoid having to
     * remember what the boolean argument means.
     * interruptIdleWorkers的通用形式，避免需要记住入参的实际含义
     * （默认无入参的会中断所有空闲worker）
     *
     */
    private void interruptIdleWorkers() {
        interruptIdleWorkers(false);
    }

    // 用于interruptIdleWorkers的常量
    private static final boolean ONLY_ONE = true;

    /*
     * Misc utilities, most of which are also exported to
     * ScheduledThreadPoolExecutor
     * 其他方法，大多数暴露给ScheduledThreadPoolExecutor
     *
     */

    /**
     * Invokes the rejected execution handler for the given command.
     * Package-protected for use by ScheduledThreadPoolExecutor.
     * 对给定的命令调用拒绝执行处理器
     * protected权限，供ScheduledThreadPoolExecutor使用。
     *
     */
    final void reject(Runnable command) {
        handler.rejectedExecution(command, this);
    }

    /**
     * Performs any further cleanup following run state transition on
     * invocation of shutdown.  A no-op here, but used by
     * ScheduledThreadPoolExecutor to cancel delayed tasks.
     * 在调用shutdown运行状态转化后，执行进一步的清理。
     * 此处无操作，被用于ScheduledThreadPoolExecutor去取消延迟任务。
     *
     */
    void onShutdown() {
    }

    /**
     * State check needed by ScheduledThreadPoolExecutor to
```

```
     * enable running tasks during shutdown.
     * 用于ScheduledThreadPoolExecutor进行状态检查，确保shutdown期间能够运行任务。
     *
     * @param shutdownOK true if should return true if SHUTDOWN
     */
    final boolean isRunningOrShutdown(boolean shutdownOK) {
        int rs = runStateOf(ctl.get());
        return rs == RUNNING || (rs == SHUTDOWN && shutdownOK);
    }

    /**
     * Drains the task queue into a new list, normally using
     * drainTo. But if the queue is a DelayQueue or any other kind of
     * queue for which poll or drainTo may fail to remove some
     * elements, it deletes them one by one.
     * 将任务队列转移到新的列表中，通常使用drainTo。
     * 但如果队列是DelayQueue或任何其他类型的queue，对于他们的poll或者drainTo可能无法移除某
些元素，它会一个一个的删除他们
     *
     */
    private List<Runnable> drainQueue() {
        BlockingQueue<Runnable> q = workQueue;
        ArrayList<Runnable> taskList = new ArrayList<Runnable>();
        q.drainTo(taskList);  // 从q中移除所有元素，并转移到给定的taskList里
        if (!q.isEmpty()) {
            for (Runnable r : q.toArray(new Runnable[0])) {
                if (q.remove(r))
                    taskList.add(r);
            }
        }
        return taskList;
    }

    /*
     * Methods for creating, running and cleaning up after workers
     * 用于创建、执行、清理worker的方法
     */

    /**
     * Checks if a new worker can be added with respect to current
     * pool state and the given bound (either core or maximum). If so,
     * the worker count is adjusted accordingly, and, if possible, a
     * new worker is created and started, running firstTask as its
     * first task. This method returns false if the pool is stopped or
     * eligible to shut down. It also returns false if the thread
     * factory fails to create a thread when asked.  If the thread
     * creation fails, either due to the thread factory returning
     * null, or due to an exception (typically OutOfMemoryError in
     * Thread.start()), we roll back cleanly.
```

```
     * 检查根据当前线程池状态和给定的边界（线程池的核心线程数与最大线程数），判断是否可以增加新的worker。
     * 如果可以，相应（accordingly）调整worker数量，并且，如何可能的话，创建新worker并启动，执行firstTask作为它的第一个任务。
     * 如果线程池已经停止或者有资格（eligible）关闭，该方法返回false。
     * 如果调用线程的工厂方法时创建线程失败，该方法返回false。
     * 如果该线程创建失败，可能有两种情况：
     * 1、由于线程工厂返回null
     * 2、由于异常（典型的如在Thread.start()时发生OOM）
     * 那我们会干净利落的进行回滚。
     *
     * @param firstTask the task the new thread should run first (or
     * null if none). Workers are created with an initial first task
     * (in method execute()) to bypass queuing when there are fewer
     * than corePoolSize threads (in which case we always start one),
     * or when the queue is full (in which case we must bypass queue).
     * Initially idle threads are usually created via
     * prestartCoreThread or to replace other dying workers.
     * firstTask参数，是新线程应该首先执行的任务（如果没有则为null）。
     * 有以下两种情况之一时，使用初始的第一个任务（通过execute()方法传入）创建新的Worker：
     * 1、当线程数少于corePoolSize（这种情况下总是启动一个新线程）
     * 2、或者当任务队列已满（这种情况下必须要绕过（bypass）queue）
     * 最初的空闲进程通常通过preStartCoreThread创建，或者替换其他将要挂掉的worker。
     * （上面说的这个替换，是指的原来将要退出的worker还没remove掉，就进行了wc+1，然后add新的worker了？）
     *
     * @param core if true use corePoolSize as bound, else
     * maximumPoolSize. (A boolean indicator is used here rather than a
     * value to ensure reads of fresh values after checking other pool
     * state).
     * core参数，如果为true则使用corePoolSize作为边界，否则使用maximumPoolSize。
     * （这里使用boolean指示符而不是具体的线程数value，是确保在检查完其他线程池状态后重新读取最新的线程数value）
     *
     * @return true if successful
     */
    private boolean addWorker(Runnable firstTask, boolean core) {           // addWorker里增加workerCount与创建启动worker不是同步的
        retry:
        for (;;) {
            int c = ctl.get();
            int rs = runStateOf(c);

            // Check if queue empty only if necessary.                      // 根据线程池状态与工作队列，有的情况下是不允许创建新的worker的
            if (rs >= SHUTDOWN &&                                           // 1、runState为STOP、TIDYING、TERMINATED，此时线程池处于关闭状态，工作队列中已没有任务
                ! (rs == SHUTDOWN &&                                        // 2、runState为SHUTDOWN时，firstTask不为null（表示在SHUTDOWN状态提交的新任务，这种的直接拒绝）
```

```
                    firstTask == null &&                                    // 3、ru
nState为SHUTDOWN时，工作队列为空（这种的也不能创建新worker）
                    ! workQueue.isEmpty()))                                 // （2、
3）的判断是为了允许如果线程池处于SHUTDOWN状态，且工作队列不为空，那么允许创建firstTask为null
的worker处理积压的工作队列任务
                return false;

            for (;;) {
                int wc = workerCountOf(c);
                if (wc >= CAPACITY ||
                    wc >= (core ? corePoolSize : maximumPoolSize))          // 判断
当前workerCount是否大于边界（根据入参core判断corePoolSize或者maximumPoolSize）
                    return false;
                if (compareAndIncrementWorkerCount(c))                      // 如果
增加workerCount成功，则跳出这两个for循环，进入该方法的新worker创建与启动部分
                    break retry;
                c = ctl.get();  // Re-read ctl
                if (runStateOf(c) != rs)                                    // 如果
增加workerCount失败，并且runState发生了变化，那么重新判断是否允许创建新的worker（从第一个for
循环开始分析）
                    continue retry;
                // else CAS failed due to workerCount change; retry inner loop // 如果
增加workerCount失败，并且runState没有变化，那么可能是并发导致workerCount发生了变化，重新CAS
来增加workerCount
            }
        }

        boolean workerStarted = false;
        boolean workerAdded = false;
        Worker w = null;
        try {
            w = new Worker(firstTask);                                      // 创建新
的worker
            final Thread t = w.thread;                                      // 获取当
前worker的线程t（这个线程t是用于执行该worker里面的任务的，这个线程t是通过线程工厂创建的）
            if (t != null) {                                                // t != n
ull，表示线程工厂成功创建线程。
                final ReentrantLock mainLock = this.mainLock;               // 对于wo
rkers的操作都需要加mainLock锁
                mainLock.lock();
                try {
                    // Recheck while holding lock.
                    // Back out on ThreadFactory failure or if
                    // shut down before lock acquired.                      // 保持锁
的状态下重新检查，在ThreadFactory失败或者在获取锁之前线程池shutdown，则退出？？？
                    int rs = runStateOf(ctl.get());

                    if (rs < SHUTDOWN ||
                        (rs == SHUTDOWN && firstTask == null)) {            // 重新判
```

```
断线程池状态（runState），只有满足条件时才将新worker加入workers集合，并运行新的worker
                        if (t.isAlive()) // precheck that t is startable      // 不知道
这个alive()检查的是啥
                            throw new IllegalThreadStateException();
                        workers.add(w);                                       // 将新wo
rker加入的workers集合
                        int s = workers.size();
                        if (s > largestPoolSize)
                            largestPoolSize = s;                              // 如果当
前workers集合数量大于largestPoolSize，那么更新largestPoolSize为当前值。这个值仅用于跟踪线程
池达到的最大值
                        workerAdded = true;                                   // 新work
er加入workers集合成功，允许新的worker启动
                    }
                } finally {
                    mainLock.unlock();
                }
                if (workerAdded) {
                    t.start();                                                // 执行t.s
tart()，会调用worker.run()方法（具体见Thread.start()）
                    workerStarted = true;                                     // 到这一
步表示worker启动没有异常（如果有OOM等异常，就走不到这一步了）
                }
            }
        } finally {
            if (! workerStarted)                                              // 如果新w
orker启动失败，那么需要从workers集合中删除该worker（当然也有可能worker就没加入到workers集合
，不过没影响）
                addWorkerFailed(w);
        }
        return workerStarted;
    }

    /**
     * Rolls back the worker thread creation.
     * - removes worker from workers, if present
     * - decrements worker count
     * - rechecks for termination, in case the existence of this
     *   worker was holding up termination
     * 回滚worker线程的创建
     * 1、如果存在的话，从workers集合中移除该worker
     * 2、worker数量减一
     * 3、重新检查终止，以防止该worker的存在耽误了线程池终止。
     *
     */
    private void addWorkerFailed(Worker w) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();                // 对workers的操作都要加mainLock锁
        try {
```

```
            if (w != null)
                workers.remove(w);       // 从workers集合中移除该worker（如果集合中没有该w
orker，会返回false）
            decrementWorkerCount();      // 如果走到了增加worker这一步（不管worker有没有创
建加入成功），那workerCount必须减一
            tryTerminate();              // 尝试终止线程池，为的是避免由于该worker的存在阻
碍了原本终止线程池操作
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Performs cleanup and bookkeeping for a dying worker. Called
     * only from worker threads. Unless completedAbruptly is set,
     * assumes that workerCount has already been adjusted to account
     * for exit.  This method removes thread from worker set, and
     * possibly terminates the pool or replaces the worker if either
     * it exited due to user task exception or if fewer than
     * corePoolSize workers are running or queue is non-empty but
     * there are no workers.
     * 清理将死worker并进行相关bookkeeping。（将死worker为即将退出worker序列的线程，不再执行
提交给线程池的任务）
     * 只能由worker线程调用。
     * 除非设置了completedAbruptly（突然完成），否则假定workerCount已经调整为认定退出。（就
是非completedAbruptly情况下，已经从workerCount中减去了该worker（wc-1））。
     * 该方法从worker集合中移除对应的线程，并尝试终止线程池或者替换该worker，替换worker的场景
为：
     * 1、用户的任务异常导致该worker线程异常退出（此时completedAbruptly为true）
     * 2、运行的worker线程数小于corePoolSize
     * 3、工作队列非空但是没有存活的worker
     *
     * @param w the worker
     * @param completedAbruptly if the worker died due to user exception
     *        completedAbruptly参数，如果worker由于用户任务异常导致挂掉的情况下为true
     */
    private void processWorkerExit(Worker w, boolean completedAbruptly) {
        if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted  // 如果
突然中断，workerCount没有来得及调整
            decrementWorkerCount();                                             // wc - 1


        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();                                                        // 任何
对worker集合的操作都要加mainLock锁
        try {
            completedTaskCount += w.completedTasks;                             // 更新
已完成任务数，注意这里用的w.completedTasks。
            workers.remove(w);                                                  // 从wo
```

```
rker集合中移除该worker
        } finally {
            mainLock.unlock();
        }

        tryTerminate();                                                    // 因为
有worker退出，防止该worker的存在影响线程池正常终止，例行公事尝试终止线程池

        int c = ctl.get();
        if (runStateLessThan(c, STOP)) {                                   // 判断
当前线程池运行状态是否<STOP
            if (!completedAbruptly) {                                      // 如果
不是异常退出，那么需要分析当前线程是否小于核心线程出，或者工作队列有任务但是没alive worker
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                if (min == 0 && ! workQueue.isEmpty())
                    min = 1;
                if (workerCountOf(c) >= min)
                    return; // replacement not needed                      // 如果
不满足上面的条件，就不用替换worker
            }
            addWorker(null, false);                                        // 如果
是异常退出或者满足上面的条件，那么需要新增一个初始任务为null的worker来消费工作队列中的任务
        }
    }

    /**
     * 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.
     * 阻塞或者限时等待获取任务，选取哪种方式取决于当前参数设置，
     * 如果该worker由于以下原因必须退出，则会返回null：
     * 1、当前worker数量超过maximumPoolSize（由于调用了setMaximumPoolSize重新设置了最大线程
池数）
     * 2、线程池被停止（runState>=STOP）
     * 3、线程池被关闭（runState=SHUTDOWN），并且工作队列为空
     * 4、该worker等待任务超时，并且超时worker在限时等待之前与之后都会终止（即allowCoreThrea
dTimeOut || workerCount > corePoolSize），如果工作队列非空，那么该worker不会是线程池里最后
一个线程。
     * （第4条这个，不明白注释说的啥，具体看代码吧）
     *
```

```
     * @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? // 最近poll是否超时（poll进行限时阻塞获取）

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

            // Check if queue empty only if necessary.
            if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {  // 想不出来这个判断怎么写才会更简洁
                decrementWorkerCount();                                   // 这里decrementWorkerCount，是因为在线程池状态为关闭并且任务队列没任务时，可以清理所有worker（不用管并发下清理的worker太多）
                return null;                                              // wc-1之后直接返回null，让worker自己去执行退出
            }

            int wc = workerCountOf(c);

            // Are workers subject to culling?
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize; // 判断是否允许超时退出

            if ((wc > maximumPoolSize || (timed && timedOut))            // 当前wc超过最大线程池数或者允许超时退出情况下上次获取已超时
                && (wc > 1 || workQueue.isEmpty())) {                    // 还需注意工作队列非空时，最后一个worker线程不能退出
                if (compareAndDecrementWorkerCount(c))                   // 这里compareAndDecrementWorkerCount(c)，要拿当前的worker数-1，是防止并发下清理的worker过多（比如现在一共有俩worker，都等待超时了，任务队列还不为空，结果并发都走到了这一步，如果直接-1.这俩就都被清理了）
                    return null;
                continue;
            }

            try {
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();                                    // 如果worker可以超时退出，那么使用限时阻塞的poll方法，否则使用take方法
                if (r != null)
                    return r;
                timedOut = true;                                         // 如果限时拿不到任务，则记录上次获取已超时
            } catch (InterruptedException retry) {
```

```
                timedOut = false;                                                    // 如果在从工
作队列中拿任务时被中断退出，那么不算上次获取超时
            }
        }
    }

    /**
     * Main worker run loop.  Repeatedly gets tasks from queue and
     * executes them, while coping with a number of issues:
     * 主要的worker执行循环。
     * 重复从工作队列中获取任务然后执行他们，同时处理（cope）以下这些问题：
     *
     * 1. We may start out with an initial task, in which case we
     * don't need to get the first one. Otherwise, as long as pool is
     * running, we get tasks from getTask. If it returns null then the
     * worker exits due to changed pool state or configuration
     * parameters.  Other exits result from exception throws in
     * external code, in which case completedAbruptly holds, which
     * usually leads processWorkerExit to replace this thread.
     * 1、我们可以从一个初始任务开始执行，在这种情况下我们不需要获取第一个任务。
     * 否则，只要线程池在运行，我们就通过getTask方法获取任务。
     * 如果由于线程池状态改变或者参数配置发生变化，getTask返回null，那么该worker退出。
     * 其他退出是由于外部代码抛出异常引起的，在这种情况下completedAbruptly成立（满足complete
dAbruptly条件），
     * 这通常会导致processWorkerExit来替换该worker线程。
     *
     * 2. Before running any task, the lock is acquired to prevent
     * other pool interrupts while the task is executing, and then we
     * ensure that unless pool is stopping, this thread does not have
     * its interrupt set.
     * 2、在运行任何任务之前，要获取lock来防止运行任务时发生其他线程池中断，
     * 并且确保除非线程池被stopping，否则该线程不会设置中断。
     *
     * 3. Each task run is preceded by a call to beforeExecute, which
     * might throw an exception, in which case we cause thread to die
     * (breaking loop with completedAbruptly true) without processing
     * the task.
     * 3、每个任务执行之前都会调用beforeExecute方法，这可能会抛出异常，
     * 在这种情况下会导致线程挂掉（设置completedAbruptly为true然后跳出循环）而不处理任务。
     *
     * 4. Assuming beforeExecute completes normally, we run the task,
     * gathering any of its thrown exceptions to send to afterExecute.
     * We separately handle RuntimeException, Error (both of which the
     * specs guarantee that we trap) and arbitrary Throwables.
     * Because we cannot rethrow Throwables within Runnable.run, we
     * wrap them within Errors on the way out (to the thread's
     * UncaughtExceptionHandler).  Any thrown exception also
     * conservatively causes thread to die.
     * 4、假设beforeExecute正常完成，然后执行task，收集执行任务期间抛出的任何异常发送给after
```

```
Execute方法。
     * 分别处理RuntimeException、Error（规范（spec）保证我们可以捕获到这两个），和任意Throwables。
     * 由于在Runnable.run方法内部不能重新抛出Throwables。所以将Throwables包装到Errors中输出（到线程的UncaughtExceptionHandler）。
     * 任何抛出的异常也会保守的（conservatively）导致线程挂掉。
     *
     * 5. After task.run completes, we call afterExecute, which may
     * also throw an exception, which will also cause thread to
     * die. According to JLS Sec 14.20, this exception is the one that
     * will be in effect even if task.run throws.
     * 5、当task.run方法执行完成，调用afterExecute方法，可能也会抛出异常，也会导致该worker线程挂掉。
     * 根据JLS Sec 14.20，该异常即使是task.run抛出，也会生效。（？？？）
     *
     * The net effect of the exception mechanics is that afterExecute
     * and the thread's UncaughtExceptionHandler have as accurate
     * information as we can provide about any problems encountered by
     * user code.
     * 这种异常机制的最终效果是：afterExecute与线程的UncaughtExceptionHandler，具有我们可以提供的关于用户代码遇到的任何问题的准确（accurate）信息。
     *
     * @param w the worker
     */
    final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();                             // 因为是thread调用的worker#run，当前线程为worker线程，该线程也就是worker的thread，通过线程工厂创建出来的。
        Runnable task = w.firstTask;                                    // 拿到给定worker的firstTask，然后将worker的firstTask置为null
        w.firstTask = null;
        w.unlock(); // allow interrupts                                 // unlock->AQS#release->tryRelease，state变为0，表示该worker进入运行阶段，允许interruptIfStarted。
        boolean completedAbruptly = true;                               // 突然完成的标识，用于判断是否要替换该worker
        try {
            while (task != null || (task = getTask()) != null) {        // 如果有需要执行的任务，就获取任务
                w.lock();                                               // 上来加锁，表示该worker在执行任务。防止被interruptIdleWorkers方法认为是空闲线程给中断了
                // If pool is stopping, ensure thread is interrupted; // 如果线程池被停止，确保线程被中断
                // if not, ensure thread is not interrupted.  This     // 如果线程池没有停止，确保线程没有被中断。
                // requires a recheck in second case to deal with      // 这需要在第二种情况下重新检查线程中断状态，以在清除中断的同时处理shutdownNow的竞争。
                // shutdownNow race while clearing interrupt
                if ((runStateAtLeast(ctl.get(), STOP) ||                // 这个if有两个功能：1、判断出该线程是否应该中断，runState>=STOP的就直接中断；2、如果该线程不应该中断，那么清理中断状态；
```

```
                    (Thread.interrupted() &&                            // 在满足runState
<STOP的前提下，Thread.interrupted()会清理当前线程的中断状态，如果被中断了，该方法会返回true
                     runStateAtLeast(ctl.get(), STOP))) &&              // 如果worker线程
发生过中断，那么再重新检查runState是否仍<STOP，如果是，则该if结束；如果不是的话，可能在if期间
发生了shutdownNow，可能需要重新设置worker线程中断
                    !wt.isInterrupted())                                // 如果worker线程
已经中断了，那么就不用重新设置中断状态了，否则就设置中断状态。wt.isInterrupted用于检查中断状态
，不会清除中断状态。
                    wt.interrupt();                                     // 这里会执行，表
示runState>=STOP，并且此时线程未设置中断状态，需要worker线程中断并退出
                try {
                    beforeExecute(wt, task);                            // 任务执行之前先
调用beforeExecute
                    Throwable thrown = null;
                    try {
                        task.run();                                     // 执行任务。注意
这里不会再开新线程来执行了，只是调用了Runnable的run方法（可能是RunnableFuture实现的Runnable）

                    } catch (RuntimeException x) {
                        thrown = x; throw x;
                    } catch (Error x) {
                        thrown = x; throw x;
                    } catch (Throwable x) {
                        thrown = x; throw new Error(x);
                    } finally {                                         // 注意：这一块异常
try-catch-finally的逻辑，内部捕获的异常会走两部分，第一部分是继续throw抛出，第二部分是传入aft
erExecute。第一部分抛出的异常在外边的try给忽略了。
                        afterExecute(task, thrown);                     // 任务执行之后调
用afterExecute，传入执行过程中捕获到的异常
                    }
                } finally {
                    task = null;
                    w.completedTasks++;                                 // 完成任务数+1
                    w.unlock();                                         // 解锁，作为空闲
线程等着了
                }
            }
            completedAbruptly = false;                                  // 如果getTask返回
了null，就表示该worker是准备正常退出了
        } finally {                                                     // 注意：内部循环如
果有抛出异常，直接忽略
            processWorkerExit(w, completedAbruptly);                    // 带着completedA
bruptly去尝试本worker进行退出与终止线程池
        }
    }

    // Public constructors and methods
    // 公共构造函数和方法
```

```
    /**
     * Creates a new {@code ThreadPoolExecutor} with the given initial
     * parameters and default thread factory and rejected execution handler.
     * It may be more convenient to use one of the {@link Executors} factory
     * methods instead of this general purpose constructor.
     * 见调用
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     *
     * @param maximumPoolSize the maximum number of threads to allow in the
     *        pool
     *
     * @param keepAliveTime when the number of threads is greater than
     *        the core, this is the maximum time that excess idle threads
     *        will wait for new tasks before terminating.
     *
     * @param unit the time unit for the {@code keepAliveTime} argument
     *
     * @param workQueue the queue to use for holding tasks before they are
     *        executed.  This queue will hold only the {@code Runnable}
     *        tasks submitted by the {@code execute} method.
     *
     * @throws IllegalArgumentException if one of the following holds:<br>
     *         {@code corePoolSize < 0}<br>
     *         {@code keepAliveTime < 0}<br>
     *         {@code maximumPoolSize <= 0}<br>
     *         {@code maximumPoolSize < corePoolSize}
     * @throws NullPointerException if {@code workQueue} is null
     */
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), defaultHandler);
    }

    /**
     * Creates a new {@code ThreadPoolExecutor} with the given initial
     * parameters and default rejected execution handler.
     * 同上
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     * @param maximumPoolSize the maximum number of threads to allow in the
     *        pool
     * @param keepAliveTime when the number of threads is greater than
```

```
     *        the core, this is the maximum time that excess idle threads
     *        will wait for new tasks before terminating.
     * @param unit the time unit for the {@code keepAliveTime} argument
     * @param workQueue the queue to use for holding tasks before they are
     *        executed.  This queue will hold only the {@code Runnable}
     *        tasks submitted by the {@code execute} method.
     * @param threadFactory the factory to use when the executor
     *        creates a new thread
     * @throws IllegalArgumentException if one of the following holds:<br>
     *         {@code corePoolSize < 0}<br>
     *         {@code keepAliveTime < 0}<br>
     *         {@code maximumPoolSize <= 0}<br>
     *         {@code maximumPoolSize < corePoolSize}
     * @throws NullPointerException if {@code workQueue}
     *         or {@code threadFactory} is null
     */
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              ThreadFactory threadFactory) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             threadFactory, defaultHandler);
    }

    /**
     * Creates a new {@code ThreadPoolExecutor} with the given initial
     * parameters and default thread factory.
     * 同上
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     * @param maximumPoolSize the maximum number of threads to allow in the
     *        pool
     * @param keepAliveTime when the number of threads is greater than
     *        the core, this is the maximum time that excess idle threads
     *        will wait for new tasks before terminating.
     * @param unit the time unit for the {@code keepAliveTime} argument
     * @param workQueue the queue to use for holding tasks before they are
     *        executed.  This queue will hold only the {@code Runnable}
     *        tasks submitted by the {@code execute} method.
     * @param handler the handler to use when execution is blocked
     *        because the thread bounds and queue capacities are reached
     * @throws IllegalArgumentException if one of the following holds:<br>
     *         {@code corePoolSize < 0}<br>
     *         {@code keepAliveTime < 0}<br>
     *         {@code maximumPoolSize <= 0}<br>
     *         {@code maximumPoolSize < corePoolSize}
```

```
     * @throws NullPointerException if {@code workQueue}
     *         or {@code handler} is null
     */
    public ThreadPoolExecutor(int corePoolSize,
                              int maximumPoolSize,
                              long keepAliveTime,
                              TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                              RejectedExecutionHandler handler) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
             Executors.defaultThreadFactory(), handler);
    }

    /**
     * Creates a new {@code ThreadPoolExecutor} with the given initial
     * parameters.
     * 创建新的ThreadPoolExecutor，通过给定的初始化参数和默认的线程工厂（ThreadFactory）和默认的拒绝执行处理器（RejectedExecutionHandler）。
     * 可能更方便的（convenient）是通过调用Executors的工厂方法，而不是此公共构造方法。（然而实际上代码风格检测通常更建议直接调用ThreadPoolExecutor的构造方法）
     *
     * @param corePoolSize the number of threads to keep in the pool, even
     *        if they are idle, unless {@code allowCoreThreadTimeOut} is set
     *        corePoolSize表示线程池保持的线程数，即使这些线程是空闲的，除非设置了allowCoreThreadTimeOut为true
     * @param maximumPoolSize the maximum number of threads to allow in the
     *        pool
     *        maximumPoolSize表示线程池允许的最大线程数
     * @param keepAliveTime when the number of threads is greater than
     *        the core, this is the maximum time that excess idle threads
     *        will wait for new tasks before terminating.
     *        keepAliveTime表示当线程数超过核心线程数时，多余的（excess）空闲线程在终止前用于等待新任务的最大等待时间
     * @param unit the time unit for the {@code keepAliveTime} argument
     *        unit表示keepAliveTime的时间单位
     * @param workQueue the queue to use for holding tasks before they are
     *        executed.  This queue will hold only the {@code Runnable}
     *        tasks submitted by the {@code execute} method.
     *        workQueue用于保存还未执行任务的队列。该队列将只保存通过execute方法提交的实现Runnable接口的任务。
     * @param threadFactory the factory to use when the executor
     *        creates a new thread
     *        threadFactor用于当executor创建新线程的时候
     * @param handler the handler to use when execution is blocked
     *        because the thread bounds and queue capacities are reached
     *        handler用于当线程池线程边界与工作队列容量饱和时导致执行阻塞，需要执行的相关拒绝策略
     * @throws IllegalArgumentException if one of the following holds:<br>
     *         {@code corePoolSize < 0}<br>
```

```
     *         {@code keepAliveTime < 0}<br>
     *         {@code maximumPoolSize <= 0}<br>   // maximum不能<=0
     *         {@code maximumPoolSize < corePoolSize}
     * @throws NullPointerException if {@code workQueue}
     *         or {@code threadFactory} or {@code handler} is null
     */
    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)
            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); // 转化为纳秒
        this.threadFactory = threadFactory;
        this.handler = handler;
    }

    /**
     * Executes the given task sometime in the future.  The task
     * may execute in a new thread or in an existing pooled thread.
     * 在未来的某个时间执行给定的任务。
     * 该任务可能在一个新的线程或者一个已经存在的线程池线程中执行。
     *
     * If the task cannot be submitted for execution, either because this
     * executor has been shutdown or because its capacity has been reached,
     * the task is handled by the current {@code RejectedExecutionHandler}.
     * 如果任务不能提交执行，可能是由于执行器已经shutdown或者它的容量已经饱和，
     * 这种情况下该任务由当前RejectedExecutionHandler来处理。
     *
     * @param command the task to execute
     * @throws RejectedExecutionException at discretion of
     *         {@code RejectedExecutionHandler}, if the task
     *         cannot be accepted for execution
     * @throws NullPointerException if {@code command} is null
     */
    public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
```

```
        /*
         * Proceed in 3 steps:
         * 分三步进行
         *
         * 1. If fewer than corePoolSize threads are running, try to
         * start a new thread with the given command as its first
         * task.  The call to addWorker atomically checks runState and
         * workerCount, and so prevents false alarms that would add
         * threads when it shouldn't, by returning false.
         * 1、如果运行线程数<corePoolSize，尝试创建新的线程，并将给定的command作为它的第一个任务。
         * 调用addWorker将原子性的检查runState和workerCount，如果不应该创建新的worker线程，addWorker通过返回false以防止添加线程时的误报。
         *
         * 2. If a task can be successfully queued, then we still need
         * to double-check whether we should have added a thread
         * (because existing ones died since last checking) or that
         * the pool shut down since entry into this method. So we
         * recheck state and if necessary roll back the enqueuing if
         * stopped, or start a new thread if there are none.
         * 2、如果任务能够成功入队，我们仍需要再次检查我们是否应该添加一个线程（因为上次检查之后，有线程挂掉了）（如果线程池线程数不满足要求，那么需要创建新的线程）
         * 或者在进入该方法后线程池已shutdown。
         * 所以我们重新检查状态，有必要的话在停止时回滚入队操作，或者在没有线程的情况下启动一个新线程。
         *
         * 3. If we cannot queue task, then we try to add a new
         * thread.  If it fails, we know we are shut down or saturated
         * and so reject the task.
         * 3、如果无法将task入队，那么尝试添加一个新的线程。
         * 如果添加失败，则知道线程池已经shutdown或者饱和，因此需要拒绝该任务。
         *
         */
        int c = ctl.get();
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))                    // 1、如果当前线程数<corePoolSize，那么启动新worker线程，将该任务作为firstTask
                return;
            c = ctl.get();                                   // 2、如果添加新worker失败，重新查询当前线程池信息
        }
        if (isRunning(c) && workQueue.offer(command)) {      // 在线程池处于RUNNING的前提下，将该command入队（offer为非阻塞入队，失败返回false）
            int recheck = ctl.get();                         // 重新检查线程池状态（这个recheck感觉是个保障，确保提交的任务一定会被处理：包括有worker来执行，或者在线程池不处理任务队列的情况下移除该任务）
            if (! isRunning(recheck) && remove(command))     // 如果此时线程池处于关闭状态（>=SHUTDOWN），则移除该command
                reject(command);                             // 拒绝该任务（相当于作为新
```

```
提交的任务来拒绝的，而SHUTDOWN状态还是会处理工作队列中的任务）
            else if (workerCountOf(recheck) == 0)             // 如果线程池还处于RUNNING
时，worker线程为0，则需要添加worker线程来处理任务
                addWorker(null, false);
        }
        else if (!addWorker(command, false))                 // 3、如果线程数>corePoolSi
ze，并且入队失败（工作队列满了），那么就新建worker线程处理该任务（创建corePoolSize到maximumP
oolSize中间的worker线程）
            reject(command);                                  // 如果新建worker失败（比
如当前rs>=SHUTDOWN），则拒绝该任务
    }

    /**
     * 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.
     * 启动有序关闭，其中之前已提交的任务将执行，不会再接受新任务。
     * 如果已经处于SHUTDOWN，重复调用不会有额外的效果。
     *
     * <p>This method does not wait for previously submitted tasks to
     * complete execution.  Use {@link #awaitTermination awaitTermination}
     * to do that.
     * 该方法不会等待之前已提交的任务完成执行。（意思就是该方法会通知线程池去SHUTDOWN，但不会
等待所有worker线程都完成退出与线程池终止）
     * 使用awaitTermination来做这些事儿（限时等待检测任务完成与线程池终止）。
     * 常见用法：
     * threadPoolExecutor.shutdown();
     * while(!threadPoolExecutor.awaitTermination(...)) { // 循环 限时等待worker线程完成
退出与线程池状态成为TERMINATED};
     *
     * @throws SecurityException {@inheritDoc}
     *
     * 注意，该方法正常退出时仅表示已正确设置了线程池状态为SHUTDOWN，但不保证所有worker线程已
完成，也不保证线程池状态到达TERMINATED状态
     *
     */
    public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();                                      // 涉及worker集
合的操作加mainLock（因为要逐步清除worker线程）
        try {
            checkShutdownAccess();                            // 检查shutdown
权限
            advanceRunState(SHUTDOWN);                        // 更新线程池状态
为SHUTDOWN（如果rs<SHUTDOWN）
            interruptIdleWorkers();                           // 中断所有*空闲*
worker线程
            onShutdown(); // hook for ScheduledThreadPoolExecutor      // 调用onShutDow
n钩子方法，通常是针对ScheduledThreadPoolExecutor
```

```
        } finally {
            mainLock.unlock();                                          // 记得解锁
        }
        tryTerminate();                                                 // 尝试终止线程池，如果不满足TERMINATED条件，会通过中断空闲线程来等待线程池自己终止
    }

    /**
     * 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.
     * 尝试停止所有正在执行的任务，停止（halt）处理等待任务，返回等待执行的任务列表。
     * 从该方法返回时，将这些等待执行的任务从任务队列排出（移除）。
     *
     * <p>This method does not wait for actively executing tasks to
     * terminate.  Use {@link #awaitTermination awaitTermination} to
     * do that.
     * 该方法不会等待正在执行的任务执行完成。（意思就是该方法会通知线程池去STOP，但不会等待所有worker线程完成退出与线程池终止）
     * 使用awaitTermination来做到这一点。（用于限时等待检测线程池里的任务是否已执行完毕，线程池已关闭）
     *
     * <p>There are no guarantees beyond best-effort attempts to stop
     * processing actively executing tasks.  This implementation
     * cancels tasks via {@link Thread#interrupt}, so any task that
     * fails to respond to interrupts may never terminate.
     * 除了尽力尝试停止运行正在执行的任务外，没有任何保证。
     * 通过Thread#interrupt实现取消任务，所以任何未能响应中断的的任务可能永远不会终止。
     * 常见用法：
     * threadPoolExecutor.shutdown();
     * while(!threadPoolExecutor.awaitTermination(...)) { // 循环 限时等待worker线程完成退出与线程池状态成为TERMINATED};
     *
     * @throws SecurityException {@inheritDoc}
     *
     * 注意，该方法正常退出时仅表示已正确设置了线程池状态为SHUTDOWN，但不保证所有worker线程已完成，也不保证线程池状态到达TERMINATED状态
     *
     */
    public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();                                                // 涉及worker集合的操作加mainLock（因为要逐步清除worker线程）
        try {
            checkShutdownAccess();                                      // 检查shutdown权限
            advanceRunState(STOP);                                      // 更新线程池状态
```

```
为STOP（如果rs<STOP）
            interruptWorkers();                                    // 中断所有线程（
与shutdown只中断空闲线程不同）
            tasks = drainQueue();                                  // 将工作队列中等
待任务排出，并生成到一个新的List中
        } finally {
            mainLock.unlock();
        }
        tryTerminate();                                            // 尝试终止线程池
        return tasks;                                             // 返回工作队列中
尚未执行的任务列表
    }

    // 判断线程池状态是否停止（rs>RUNNING）
    public boolean isShutdown() {
        return ! isRunning(ctl.get());
    }

    /**
     * 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.
     * 如果executor在shutdown或者shutdownNow之后正在终止但尚未完全终止，该方法返回true。
     * 该方法可能对debug有用。如果在shutdown之后足够（sufficient）长的时间里该方法仍返回true
,
     * 可能表明提交的任务已经忽略或者抑制了中断（不响应中断），导致executor无法正常终止。
     *
     * @return {@code true} if terminating but not yet terminated
     *          正在终止但未完成终止，返回true
     */
    public boolean isTerminating() {
        int c = ctl.get();
        return ! isRunning(c) && runStateLessThan(c, TERMINATED);
    }

    // 检测线程池是否已终止
    public boolean isTerminated() {
        return runStateAtLeast(ctl.get(), TERMINATED);
    }

    // 可以在shutdown()与shutdonwNow()调用之后，调用该方法，等待检测线程池是否已关闭
    public boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException {
        long nanos = unit.toNanos(timeout);
        final ReentrantLock mainLock = this.mainLock;
```

```
            mainLock.lock();                                    // 通过加锁来防止其他对
worker的操作（包括清理空闲/所有worker）
            try {
                for (;;) {
                    if (runStateAtLeast(ctl.get(), TERMINATED))       // 如果线程池状态已到达
TERMINATED，说明现在线程池已关闭（没有在运行的worker线程了），返回true
                        return true;
                    if (nanos <= 0)
                        return false;
                    nanos = termination.awaitNanos(nanos);            // 通过Condition来限时
等待（如果线程池终止了，tryTerminate会通过调用termination.signalAll来唤醒）。
                }
            } finally {
                mainLock.unlock();
            }
        }

        /**
         * Invokes {@code shutdown} when this executor is no longer
         * referenced and it has no threads.
         * 当不再引用此executor并且没有线程时执行关闭
         *
         */
        protected void finalize() {
            shutdown();
        }

        /**
         * Sets the thread factory used to create new threads.
         * 设置线程工厂用于创建新线程
         *
         * @param threadFactory the new thread factory
         * @throws NullPointerException if threadFactory is null
         * @see #getThreadFactory
         */
        public void setThreadFactory(ThreadFactory threadFactory) {
            if (threadFactory == null)
                throw new NullPointerException();
            this.threadFactory = threadFactory;     // 通过volatile保持可见性
        }

        /**
         * Returns the thread factory used to create new threads.
         * 返回用于创建线程的线程工厂
         *
         * @return the current thread factory
         * @see #setThreadFactory(ThreadFactory)
         */
        public ThreadFactory getThreadFactory() {
```

```
        return threadFactory;
    }

    /**
     * Sets a new handler for unexecutable tasks.
     * 设置新的handler用于无法执行的任务
     *
     * @param handler the new handler
     * @throws NullPointerException if handler is null
     * @see #getRejectedExecutionHandler
     */
    public void setRejectedExecutionHandler(RejectedExecutionHandler handler) {
        if (handler == null)
            throw new NullPointerException();
        this.handler = handler;              // 通过volatile保持可见性
    }

    /**
     * Returns the current handler for unexecutable tasks.
     * 返回当前用于处理无法执行任务的handler
     *
     * @return the current handler
     * @see #setRejectedExecutionHandler(RejectedExecutionHandler)
     */
    public RejectedExecutionHandler getRejectedExecutionHandler() {
        return handler;
    }

    /**
     * Sets the core number of threads.  This overrides any value set
     * in the constructor.  If the new value is smaller than the
     * current value, excess existing threads will be terminated when
     * they next become idle.  If larger, new threads will, if needed,
     * be started to execute any queued tasks.
     * 设置核心线程数。
     * 这会覆盖构造函数中设置的任意值。（就是不管构造函数中原值是否大于/小于/等于目标值，都会
设置为目标值）
     * 如果新值<当前值（原值），多余的现有线程会在下一次空闲时终止。
     * 如果新值>当前值（原值），如果需要的话，将启动新线程来执行任意排队任务。
     *
     * @param corePoolSize the new core size
     * @throws IllegalArgumentException if {@code corePoolSize < 0} // 注意不能<0
     * @see #getCorePoolSize
     */
    public void setCorePoolSize(int corePoolSize) {
        if (corePoolSize < 0)                                          // 核心线程数不能<0

            throw new IllegalArgumentException();
        int delta = corePoolSize - this.corePoolSize;
```

```
        this.corePoolSize = corePoolSize;
        if (workerCountOf(ctl.get()) > corePoolSize)
            interruptIdleWorkers();                                    // 如果当前线程数>
更新后的核心线程数，尝试中断空闲worker线程
        else if (delta > 0) {                                          // 如果更新后核心线
程数>原值，需要考虑是否是需要是为了增加worker线程而特意设置了较大的核心线程数。
            // We don't really know how many new threads are "needed".
            // As a heuristic, prestart enough new workers (up to new
            // core size) to handle the current number of tasks in
            // queue, but stop if queue becomes empty while doing so.
            // 我们不知道需要启动多少新线程。
            // 作为一种启发式（heuristic）方法，预先启动足够多的新worker线程（最多达到新的核
心线程数）来处理队列中的当前任务
            // 但如果在执行此操作过程中队列变空，则停止启动新的worker线程（原已启动的worker线
程会继续存活（如果没有设置allowCoreThreadTimeOut））
            //
            int k = Math.min(delta, workQueue.size());             // 新启动的线程数取
核心线程数与当前队列任务数中的最小值
            while (k-- > 0 && addWorker(null, true)) {             // 避免启的新线程太多
，有不必要的浪费
                if (workQueue.isEmpty())                           // 如果任务队列为空
了，那么不管当前worker线程数是否达到了核心线程数，都停止创建新的线程
                    break;
            }
        }
    }

    /**
     * Returns the core number of threads.
     * 返回核心线程数
     *
     * @return the core number of threads
     * @see #setCorePoolSize
     */
    public int getCorePoolSize() {
        return corePoolSize;
    }

    /**
     * Starts a core thread, causing it to idly wait for work. This
     * overrides the default policy of starting core threads only when
     * new tasks are executed. This method will return {@code false}
     * if all core threads have already been started.
     * 启动核心线程，让它空闲着等待任务。
     * 该方法覆盖了默认只有在执行新任务的时候才启动核心线程的策略（默认如果没任务就不启动线程）
     * 该方法会返回false，如果所有的核心线程在此之前都已经启动了（在进入该方法前，worker线程数
已达到核心线程数）
     * （预启动一个核心线程，在任务提交之前准备好线程）
     *
```

```
     * @return {@code true} if a thread was started
     */
    public boolean prestartCoreThread() {
        return workerCountOf(ctl.get()) < corePoolSize &&
            addWorker(null, true);                          // 只启动一个核心线程
    }

    /**
     * Same as prestartCoreThread except arranges that at least one
     * thread is started even if corePoolSize is 0.
     * 与prestartCoreThread相同，安排至少一个线程启动，即使设置的corePoolSize为0
     * （保证线程池中至少有一个worker线程，即使设置了线程池的corePoolSize为0）
     *
     */
    void ensurePrestart() {
        int wc = workerCountOf(ctl.get());
        if (wc < corePoolSize)
            addWorker(null, true);
        else if (wc == 0)
            addWorker(null, false);
    }

    /**
     * Starts all core threads, causing them to idly wait for work. This
     * overrides the default policy of starting core threads only when
     * new tasks are executed.
     * 启动所有核心线程，让他们空闲等待任务到来。
     * 该方法覆盖了默认只有在执行新任务的时候才启动核心线程的策略（默认如果没任务就不启动线程）
     *
     * @return the number of threads started
     *         返回启动的线程数
     */
    public int prestartAllCoreThreads() {
        int n = 0;
        while (addWorker(null, true))     // 传给addWorker()方法中，true参数表示需要启动
核心线程，如果当前worker线程数>corePoolSize，返回false
            ++n;
        return n;
    }

    /**
     * Returns true if this pool allows core threads to time out and
     * terminate if no tasks arrive within the keepAlive time, being
     * replaced if needed when new tasks arrive. When true, the same
     * keep-alive policy applying to non-core threads applies also to
     * core threads. When false (the default), core threads are never
     * terminated due to lack of incoming tasks.
     * 如果该线程池允许核心线程在keepAlive时间内没有任务到达时超时与终止，则返回true，
     * 并在新任务到达时根据需要进行替换。（不知道这个替换跟addWorker()的替换又有什么关系？？？
```

```
)
     * 当返回true时，应用于非核心线程的keep-alive策略也会应用于核心线程。
     * 当返回false时（也就是allowCoreThreadTimeOut的默认值），核心线程永远不会由于缺失传入任务而终止。
     *
     * @return {@code true} if core threads are allowed to time out,
     *         else {@code false}
     *
     * @since 1.6
     */
    public boolean allowsCoreThreadTimeOut() {
        return allowCoreThreadTimeOut;
    }

    /**
     * Sets the policy governing whether core threads may time out and
     * terminate if no tasks arrive within the keep-alive time, being
     * replaced if needed when new tasks arrive. When false, core
     * threads are never terminated due to lack of incoming
     * tasks. When true, the same keep-alive policy applying to
     * non-core threads applies also to core threads. To avoid
     * continual thread replacement, the keep-alive time must be
     * greater than zero when setting {@code true}. This method
     * should in general be called before the pool is actively used.
     * 设置策略，主要用于控制该线程池核心线程在keepAlive时间内没有任务到达时超时与终止，
     * 并在新任务到达时根据需要进行替换。
     * 如果设置为false，核心线程永远不会由于缺失传入任务而终止。
     * 如果设置为true，应用于非核心线程的keep-alive策略也会应用于核心线程。
     * 为避免不断更换线程，当设置为true时，keep-alive的值必须大于0。
     * 通常在主动使用线程池之前调用此方法。
     *
     * @param value {@code true} if should time out, else {@code false}
     * @throws IllegalArgumentException if value is {@code true}
     *         and the current keep-alive time is not greater than zero
     *
     * @since 1.6
     */
    public void allowCoreThreadTimeOut(boolean value) {
        if (value && keepAliveTime <= 0)
            throw new IllegalArgumentException("Core threads must have nonzero keep alive times");
        if (value != allowCoreThreadTimeOut) {     // 如果新值不等于原来的allowCoreThreadTimeOut，那么就设置新值
            allowCoreThreadTimeOut = value;
            if (value)
                interruptIdleWorkers();            // 如果原来不是true，现在为true了，则中断空闲的线程
        }
    }
```

```
    /**
     * Sets the maximum allowed number of threads. This overrides any
     * value set in the constructor. If the new value is smaller than
     * the current value, excess existing threads will be
     * terminated when they next become idle.
     * 设置允许的最大线程数。
     * 该方法覆盖构造方法中设置的任意值。
     * 如果新值<当前值（原值），存在的多余（excess）线程将在他们下一次空闲时被终止
     *
     * @param maximumPoolSize the new maximum
     * @throws IllegalArgumentException if the new maximum is
     *         less than or equal to zero, or
     *         less than the {@linkplain #getCorePoolSize core pool size}
     *         要保证maximumPoolSize>=corePoolSize>0
     * @see #getMaximumPoolSize
     */
    public void setMaximumPoolSize(int maximumPoolSize) {
        if (maximumPoolSize <= 0 || maximumPoolSize < corePoolSize)
            throw new IllegalArgumentException();
        this.maximumPoolSize = maximumPoolSize;
        if (workerCountOf(ctl.get()) > maximumPoolSize) // 如果当前worker线程数>新值maxi
mumPoolSize，需要将空闲线程终止
            interruptIdleWorkers();
    }

    /**
     * Returns the maximum allowed number of threads.
     * 返回允许的最大线程数
     *
     * @return the maximum allowed number of threads
     * @see #setMaximumPoolSize
     */
    public int getMaximumPoolSize() {
        return maximumPoolSize;
    }

    /**
     * Sets the time limit for which threads may remain idle before
     * being terminated.  If there are more than the core number of
     * threads currently in the pool, after waiting this amount of
     * time without processing a task, excess threads will be
     * terminated.  This overrides any value set in the constructor.
     * 设置线程在终止前可以保持空闲的时间限制。（空闲多少时长后终止）
     * 如果在线程池中的当前线程数超过核心线程数，并在这段等待时间内没有执行任务，多余的线程将被
终止。
     * 这会覆盖构造函数中设置的任意值。
     *
     * @param time the time to wait.  A time value of zero will cause
```

```
     *        excess threads to terminate immediately after executing tasks.
     *        time参数，等待时间。如果time值为0，将导致执行额外的线程（超过核心线程的那部分线程）在执行完任务后会立即终止
     *
     * @param unit the time unit of the {@code time} argument
     * @throws IllegalArgumentException if {@code time} less than zero or
     *         if {@code time} is zero and {@code allowsCoreThreadTimeOut}
     *         注意，必须保证time>=0，
     *               如果设置了allowCoreThreadTimeOut，必须保证time>0（要不然留不住核心线程）
     * @see #getKeepAliveTime(TimeUnit)
     */
    public void setKeepAliveTime(long time, TimeUnit unit) {
        if (time < 0)
            throw new IllegalArgumentException();
        if (time == 0 && allowsCoreThreadTimeOut())
            throw new IllegalArgumentException("Core threads must have nonzero keep alive times");
        long keepAliveTime = unit.toNanos(time);
        long delta = keepAliveTime - this.keepAliveTime;
        this.keepAliveTime = keepAliveTime;
        if (delta < 0)              // 如果keepAliveTime时间调小，则需要尝试终止已超时的空闲线程
            interruptIdleWorkers();
    }

    /**
     * Returns the thread keep-alive time, which is the amount of time
     * that threads in excess of the core pool size may remain
     * idle before being terminated.
     * 返回线程keep-alive时长，这是超过线程池核心线程数的线程在终止之前可能保持空闲的时长。
     *
     * @param unit the desired time unit of the result
     * @return the time limit
     * @see #setKeepAliveTime(long, TimeUnit)
     */
    public long getKeepAliveTime(TimeUnit unit) {
        return unit.convert(keepAliveTime, TimeUnit.NANOSECONDS); // 转化为入参的时间单位返回
    }

    /* User-level queue utilities */
    // 用户级的队列操作

    /**
     * Returns the task queue used by this executor. Access to the
     * task queue is intended primarily for debugging and monitoring.
     * This queue may be in active use.  Retrieving the task queue
     * does not prevent queued tasks from executing.
```

```
     * 返回线程池使用的任务队列。
     * 访问任务队列主要用于debug与监控。
     * 该队列可能正在使用（动态变化）
     * 遍历任务队列不会阻止入队的任务执行。
     *
     * @return the task queue
     */
    public BlockingQueue<Runnable> getQueue() {
        return workQueue;
    }

    /**
     * Removes this task from the executor's internal queue if it is
     * present, thus causing it not to be run if it has not already
     * started.
     * 如果该任务存在（present），从executor的内部队列中移除该任务，这会导致尚未启动的任务不会去运行。
     *
     * <p>This method may be useful as one part of a cancellation
     * scheme.  It may fail to remove tasks that have been converted
     * into other forms before being placed on the internal queue. For
     * example, a task entered using {@code submit} might be
     * converted into a form that maintains {@code Future} status.
     * However, in such cases, method {@link #purge} may be used to
     * remove those Futures that have been cancelled.
     * 该方法可作为取消方案的一部分。
     * 它可能无法删除那些在放入内部队列之前已转化为其他形式的任务。
     * 例如，使用submit方法入队的任务可能转化为了含有Future状态的形式。
     * （例如Runnable通过AbstractExecutorService#submit()转成了FutureTask对象（这个过程叫形式转化），如果还用Runnable对象来remove，是删除不了的）
     * 但是，在这种情况下，purge方法可用于删除这部分已经取消的Future。
     *
     * @param task the task to remove
     * @return {@code true} if the task was removed
     */
    public boolean remove(Runnable task) {
        boolean removed = workQueue.remove(task);
        tryTerminate(); // In case SHUTDOWN and now empty // 对于rs=SHUTDOWN并且任务队列非空的情况下，删除了一个任务队列任务，需要尝试终止线程池。
        return removed;
    }

    /**
     * Tries to remove from the work queue all {@link Future}
     * tasks that have been cancelled. This method can be useful as a
     * storage reclamation operation, that has no other impact on
     * functionality. Cancelled tasks are never executed, but may
     * accumulate in work queues until worker threads can actively
     * remove them. Invoking this method instead tries to remove them now.
```

```
     * However, this method may fail to remove tasks in
     * the presence of interference by other threads.
     * 尝试从工作队列中删除所有已取消的Future任务。
     * 该方法可用作存储（storage）回收（reclamation）操作，对功能没有其他影响。
     * 取消的任务永远不会执行，但可能会在工作队列中累积，直到worker线程主动删除他们。（worker会调用任务的run方法，在FutureTask中，如果run的Callable的state不为NEW，则直接结束）
     * 现在调用该方法会尝试删除他们。
     * 然而，这种方法可能会在（presence 存在）其他线程干扰（interference）下无法删除任务。
     *
     */
    public void purge() {
        final BlockingQueue<Runnable> q = workQueue;
        try {
            Iterator<Runnable> it = q.iterator();
            while (it.hasNext()) {
                Runnable r = it.next();
                if (r instanceof Future<?> && ((Future<?>)r).isCancelled()) // 判断任务是否实现了Future接口，并判断该Future任务是否已取消
                    it.remove();
            }
        } catch (ConcurrentModificationException fallThrough) {
            // Take slow path if we encounter interference during traversal.
            // Make copy for traversal and call remove for cancelled entries.
            // The slow path is more likely to be O(N*N).
            // 发生并发修改异常
            // 如果在遍历过程中遇到干扰，采取slow path。
            // 为遍历创建副本，并调用remove来取消entries（就是遍历对象）
            // slow path的时间复杂度很大的可能为O(N*N)
            //
            for (Object r : q.toArray())
                if (r instanceof Future<?> && ((Future<?>)r).isCancelled())
                    q.remove(r);
        }

        tryTerminate(); // In case SHUTDOWN and now empty // 对于rs=SHUTDOWN并且任务队列非空的情况下，删除了一个任务队列任务，需要尝试终止线程池。
    }

    /* Statistics */
    // 统计数据

    /**
     * Returns the current number of threads in the pool.
     * 返回当前线程池线程数
     *
     * @return the number of threads
     */
    public int getPoolSize() {
        final ReentrantLock mainLock = this.mainLock;
```

```
        mainLock.lock();                                    // 对于worker集合的操作都
要加mainLock
        try {
            // Remove rare and surprising possibility of
            // isTerminated() && getPoolSize() > 0
            // 一般来说，当执行到isTerminated()方法时，线程池里应该是没有worker线程了
            // 但是为了防止罕见与惊讶的可能性发生，对这种情况进行单独判断
            return runStateAtLeast(ctl.get(), TIDYING) ? 0 // 如果rs=TIDYING，直接返回0
，不再去检查worker集合的数量
                : workers.size();
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns the approximate number of threads that are actively
     * executing tasks.
     * 返回正在执行任务的线程大致（approximate）数量。（不包含空闲的线程）
     *
     * @return the number of threads
     */
    public int getActiveCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();                // 对worker集合的操作都需要加mainLock
        try {
            int n = 0;
            for (Worker w : workers)
                if (w.isLocked())       // 判断线程是否正在执行任务的依据是看该worker是否
有加锁
                    ++n;
            return n;
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns the largest number of threads that have ever
     * simultaneously been in the pool.
     * 返回同时（simultaneously）进入线程池的历史最大线程数。
     *
     * @return the number of threads
     */
    public int getLargestPoolSize() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();            // 对worker集合的操作都需要加mainLock，因为largestPoo
lSize依赖与workers.size()，所以也加了锁
        try {
```

```
            return largestPoolSize;
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns the approximate total number of tasks that have ever been
     * scheduled for execution. Because the states of tasks and
     * threads may change dynamically during computation, the returned
     * value is only an approximation.
     * 返回已安排执行的所有任务的预估值。（包含已执行完成、正在执行、入队等待执行的所有任务数）
     * 由于在计算过程中任务状态与线程状态可能动态变化，因此返回值只是一个近似值。
     *
     * @return the number of tasks
     */
    public long getTaskCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();                    // 对worker集合的操作都需要加mainLock
        try {
            long n = completedTaskCount;
            for (Worker w : workers) {
                n += w.completedTasks;      // 任务数累加已执行完成任务数
                if (w.isLocked())
                    ++n;                    // 任务数累加当前worker正在执行任务
            }
            return n + workQueue.size();    // 任务数累加工作队列中尚未执行的任务数
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns the approximate total number of tasks that have
     * completed execution. Because the states of tasks and threads
     * may change dynamically during computation, the returned value
     * is only an approximation, but one that does not ever decrease
     * across successive calls.
     * 返回已执行完成的所有数的预估值。
     * 由于在计算过程中任务状态与线程状态可能动态变化，因此返回值只是一个近似值，
     * 但是在连续调用本方法时该值永远不会减少。
     *
     * @return the number of tasks
     */
    public long getCompletedTaskCount() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();                    // 对worker集合的操作都需要加mainLock
        try {
            long n = completedTaskCount;
```

```
            for (Worker w : workers)
                n += w.completedTasks;      // 任务数累加已执行完成任务数
            return n;
        } finally {
            mainLock.unlock();
        }
    }

    /**
     * Returns a string identifying this pool, as well as its state,
     * including indications of run state and estimated worker and
     * task counts.
     * 返回标识此线程池及其状态的字符串，包括运行状态、预估worker数量、任务数量
     *
     * @return a string identifying this pool, as well as its state
     */
    public String toString() {
        long ncompleted;        // 完成任务数
        int nworkers, nactive;  // worker线程数、活跃worker线程数（活跃=非空闲）
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            ncompleted = completedTaskCount;
            nactive = 0;
            nworkers = workers.size();
            for (Worker w : workers) {
                ncompleted += w.completedTasks;
                if (w.isLocked())
                    ++nactive;
            }
        } finally {
            mainLock.unlock();
        }
        int c = ctl.get();
        String rs = (runStateLessThan(c, SHUTDOWN) ? "Running" :
                     (runStateAtLeast(c, TERMINATED) ? "Terminated" :
                      "Shutting down"));
        return super.toString() +
            "[" + rs +                                  // 运行状态
            ", pool size = " + nworkers +               // 线程池大小=worker线程数
            ", active threads = " + nactive +           // 活跃线程数=非空闲线程数
            ", queued tasks = " + workQueue.size() +    // 排队等待任务数=工作队列大小
            ", completed tasks = " + ncompleted +       // 完成任务数
            "]";
    }

    /* Extension hooks */
    // 扩展的钩子方法
```

```
    /**
     * 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.
     * 在给定的线程中执行给定的Runnable任务时，首先执行该方法。
     * 在线程t将要执行任务r的时候调用该方法，可用于重新初始化ThreadLocals，或者记录日志。
     *
     * <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.
     * 本类该方法没有具体实现，但在子类中可能会自定义实现
     * 注意：为了正确的嵌套覆盖，子类通常应该在该方法的末尾调用super.beforeExecute
     *
     * @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) { }

    /**
     * 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.
     * 在执行完给定的Runnable任务后调用该方法。
     * 由执行任务的线程调用该方法。
     * 如果Throwable非空，表示导致执行突然中断并且未被捕获的RunntimeException或Error
     * （这个未被捕获，表示未被Runnable内部catch，但被runWorker给catch到了，再通过调用after
Execute进行处理）
     *
     * <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.
     * 本类该方法没有具体实现，但在子类中可能会自定义实现
     * 注意：为了正确的嵌套覆盖，子类通常应该在该方法前面调用super.afterExecute（跟beforeExe
cute顺序相反）
     *
     * <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:
     * 注意：当动作被明确的或者通过submit方法包含到任务中（例如FutureTask），该任务对象捕获和
```

```
维护计算异常，
     * 所以它不会导致突然终止，并且内部的异常不会传递给此方法。
     * （回顾一下FutureTask的run方法，该方法执行时不会抛出异常，而是将异常给set进了outcome里
，只有通过get方法拿执行结果outcome时，才能知道运行结果是否有异常）
     * （这个不会传递意思是，这部分异常不会通过Throwable t入参传递，而是task内部的异常。task
内部维护的异常不会抛出，在runWorker中也捕获不到）
     * 如果想要在该方法中捕获这两种失败，可以进一步探测此种情况，
     * 例如下面的样例子类里，如果任务已中止（abort），将打印直接原因或者底层异常。
     *
     *  <pre> {@code
     * class ExtendedExecutor extends ThreadPoolExecutor {
     *   // ...
     *   protected void afterExecute(Runnable r, Throwable t) {
     *     super.afterExecute(r, t);                                    // 先调用父类afterE
xecute
     *     if (t == null && r instanceof Future<?>) {                   // 如果runWorker捕
获到的异常为null，并且Runnable r为Future类型（存在异常由Future维护，而没有throw给runWorker
）
     *       try {
     *         Object result = ((Future<?>) r).get();                   // 调用Future的get
时，如果执行过程中发生异常，那么该方法会直接抛出由它维护的相关异常
     *       } catch (CancellationException ce) {                       // 捕获Future维护的
异常
     *           t = ce;
     *       } catch (ExecutionException ee) {
     *           t = ee.getCause();
     *       } catch (InterruptedException ie) {
     *           Thread.currentThread().interrupt(); // ignore/reset  // 捕获到中断异常
，重新设置中断
     *       }
     *     }
     *     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.
     * 当executor终止时调用该方法。
     * 默认本类该方法没有具体实现。
```

```
     * 注意：为了正确的嵌套(nest 巢 mutiple 多个，嵌套多个)覆盖，子类通常应该在该方法中调用super.terminated。
     *
     */
    protected void terminated() { }

    /* Predefined RejectedExecutionHandlers */
    /* 预定义的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.
     * 处理拒绝任务的handler，将直接在execute方法的调用线程中执行被拒绝的任务。
     * 除非executor已经关闭，在这种情况下该任务将被丢弃。
     *
     */
    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.
         * 在caller线程中执行任务r，除非executor已经关闭，这种情况下该任务将被抛弃。
         *
         * @param r the runnable task requested to be executed    r是需要执行的任务
         * @param e the executor attempting to execute this task  e是尝试执行该任务的executor（用于判断该executor线程池是否被关闭）
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                r.run();
            }
        }
    }

    /**
     * A handler for rejected tasks that throws a
     * {@code RejectedExecutionException}.
     * 处理拒绝任务的handler，抛出RejectedExecutionException
     *
     */
    public static class AbortPolicy implements RejectedExecutionHandler {
        /**
         * Creates an {@code AbortPolicy}.
```

```
         */
        public AbortPolicy() { }

        /**
         * Always throws RejectedExecutionException.
         * 总是抛出RejectedExecutionException
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         * @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.
     * 处理拒绝任务的handler，静默抛弃拒绝的任务
     *
     */
    public static class DiscardPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code DiscardPolicy}.
         */
        public DiscardPolicy() { }

        /**
         * Does nothing, which has the effect of discarding task r.
         * 啥也不做，就会静默丢弃任务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.
     * 处理拒绝任务的handler，丢弃最早未处理的请求（任务），然后对当前拒绝的任务重试execute，
     * 除非executor关闭，这种情况下该任务将被抛弃。
     *
     */
```

```
    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.
         * 获取并忽略executor将要执行的下一个任务，
         * 如果一个任务立即可用，那么重试任务r的执行，
         * 除非executor关闭，这种情况下该任务r将被抛弃。
         *
         * @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(); // poll 非阻塞，没有返回null
                e.execute(r);
            }
        }
    }
}
```

AbstractOwnableSynchronizer

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent.locks;

/**
 * A synchronizer that may be exclusively owned by a thread.  This
 * class provides a basis for creating locks and related synchronizers
 * that may entail a notion of ownership.  The
 * {@code AbstractOwnableSynchronizer} class itself does not manage or
 * use this information. However, subclasses and tools may use
 * appropriately maintained values to help control and monitor access
```

```
 * and provide diagnostics.
 * 同步器可能被一个线程独占
 * 该类为创建可能需要所有权概念（notion）的锁和相关同步器提供了基础。
 * AbstractOwnableSynchronizer类本身不管理或使用这些信息。
 * 但是子类和工具可以使用适当（appropriately）维护（maintain）的值来帮助控制、访问控制器、提供诊断。
 *
 * @since 1.6
 * @author Doug Lea
 */
public abstract class AbstractOwnableSynchronizer
    implements java.io.Serializable {

    /** Use serial ID even though all fields transient. */
    private static final long serialVersionUID = 3737899427754241961L;

    /**
     * Empty constructor for use by subclasses.
     */
    protected AbstractOwnableSynchronizer() { }

    /**
     * The current owner of exclusive mode synchronization.
     * 独占锁当前的拥有者
     * transient 在序列化过程中，用transient修饰的属性不会被序列化，也就是在序列化之后该属性无法被访问
     * 一旦变量被transient修饰，变量将不再是对象持久化的一部分
     */
    private transient Thread exclusiveOwnerThread;

    /**
     * Sets the thread that currently owns exclusive access.
     * A {@code null} argument indicates that no thread owns access.
     * This method does not otherwise impose any synchronization or
     * {@code volatile} field accesses.
     * @param thread the owner thread
     * 设置线程为当前独占访问的拥有者。
     * 参数为null表示没有线程拥有访问。也就是没有被占用。
     * 此方法不会以其他方式强加任何同步或者volatile字段访问？？？
     */
    protected final void setExclusiveOwnerThread(Thread thread) {
        exclusiveOwnerThread = thread;
    }

    /**
     * Returns the thread last set by {@code setExclusiveOwnerThread},
     * or {@code null} if never set.  This method does not otherwise
     * impose any synchronization or {@code volatile} field accesses.
     * @return the owner thread
```

```
     * 返回最后一次通过setExclusiveOwnerThread设置的线程，如果从来没有设置，返回null。
     * 此方法不会以其他方式强加任何同步或者volatile字段访问？？？
     */
    protected final Thread getExclusiveOwnerThread() {
        return exclusiveOwnerThread;
    }
}
```

AbstractQueuedSynchronizer

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166 // assistance 协助
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent.locks;
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import sun.misc.Unsafe;

// 这里引入了一个Unsafe，是以前没看过的，等看完这个再看
```

```
/**
 * Provides a framework for implementing blocking locks and related
 * synchronizers (semaphores, events, etc) that rely on
 * first-in-first-out (FIFO) wait queues.  This class is designed to
 * be a useful basis for most kinds of synchronizers that rely on a
 * single atomic {@code int} value to represent state. Subclasses
 * must define the protected methods that change this state, and which
 * define what that state means in terms of this object being acquired
 * or released.  Given these, the other methods in this class carry
 * out all queuing and blocking mechanics. Subclasses can maintain
 * other state fields, but only the atomically updated {@code int}
 * value manipulated using methods {@link #getState}, {@link
 * #setState} and {@link #compareAndSetState} is tracked with respect
 * to synchronization.
 * 提供一个框架，用于实现依赖FIFO等待队列的阻塞锁和相关（related）同步器（semaphores、events等）。
 * 这个类被设计成一个有用的基本类，对于大多数依赖单个原子int值代表状态的同步器。
 * 子类必须定义用来改变状态的受保护方法，并定义该状态在获取/释放这个对象时的含义。
 * 鉴于这些，此类中的其他方法执行（carry out）所有排队和阻塞机制。
 * 子类可以维护（maintain）其他状态字段，但是只有使用getState、setState和compareAndSetState方法来操纵（manipulated）原子性的更新int值，才会在同步方面进行跟踪
 *
 * <p>Subclasses should be defined as non-public internal helper
 * classes that are used to implement the synchronization properties
 * of their enclosing class.  Class
 * {@code AbstractQueuedSynchronizer} does not implement any
 * synchronization interface.  Instead it defines methods such as
 * {@link #acquireInterruptibly} that can be invoked as
 * appropriate by concrete locks and related synchronizers to
 * implement their public methods.
 * 子类应该被定义为非公共的内部帮助类，用于实现其封闭类的同步属性。（就是AQS的子类都应该作为[想要FIFO同步属性的]类的内部类使用，就像ReentrantLock里面的Sync类）
 * AbstractQueuedSynchronizer类没有实现任何同步接口。
 * 相反，它定义了例如acquireInterruptibly等方法，可以被有关（concrete）锁与相关同步器适当调用来实现他们的公共方法。
 *
 * <p>This class supports either or both a default <em>exclusive</em>
 * mode and a <em>shared</em> mode. When acquired in exclusive mode,
 * attempted acquires by other threads cannot succeed. Shared mode
 * acquires by multiple threads may (but need not) succeed. This class
 * does not &amp;quot;understand&amp;quot; these differences except in the
 * mechanical sense that when a shared mode acquire succeeds, the next
 * waiting thread (if one exists) must also determine whether it can
 * acquire as well. Threads waiting in the different modes share the
 * same FIFO queue. Usually, implementation subclasses support only
 * one of these modes, but both can come into play for example in a
 * {@link ReadWriteLock}. Subclasses that support only exclusive or
 * only shared modes need not define the methods supporting the unused mode.
 * 这个类支持独占（exclusive）模式与共享（shared）模式中的一种或者两种。
```

```
 * 在独占模式中获取（可以理解为加锁）时，其他线程获取不会成功。
 * 在共享模式中多个线程获取可能（但是不一定）成功。
 * 该类不理解这些不同点，除了在机械意义上说，当共享模式获取成功时，下一个等待线程（如果有一个的话）必须确定它是否也可以获取。
 * 在不同模式下等待的线程们共享同一个FIFO队列。（意味着一个AQS可以同时实现两种模式，就开头第一句话）。
 * 通常，子类实现时只支持其中一种模式，但是两种都可以起到作用，例如在ReadWriteLock。
 * 仅支持独占或者共享模式的子线程不需要实现支持未使用模式的方法（就是两种模式会有两套方法，如果只实现一种模式，只选择一套方法实现就行）（这里也可以解释为什么两套方法都不是abstract方法）。
 *
 * <p>This class defines a nested {@link ConditionObject} class that
 * can be used as a {@link Condition} implementation by subclasses
 * supporting exclusive mode for which method {@link
 * #isHeldExclusively} reports whether synchronization is exclusively
 * held with respect to the current thread, method {@link #release}
 * invoked with the current {@link #getState} value fully releases
 * this object, and {@link #acquire}, given this saved state value,
 * eventually restores this object to its previous acquired state.  No
 * {@code AbstractQueuedSynchronizer} method otherwise creates such a
 * condition, so if this constraint cannot be met, do not use it.  The
 * behavior of {@link ConditionObject} depends of course on the
 * semantics of its synchronizer implementation.
 * 该类定义了一个嵌套（nest）的ConditionObject对象，可以被支持独占模式的子类用来作为Condition的实现，其中：
 * isHeldExclusively方法报告是否针对当前线程独占同步，
 * 使用当前getState值调用release方法完全释放这个对象？？？
 * 和acquire，给定保存状态值，最终将这个对象恢复到它之前获取的状态。（看不懂，等下看源码再看看是啥意思）
 * 没有AbstractQueuedSynchronizer方法否则创建这样的condition，所以如果这些约束不能被满足，不要使用它。
 * ConditionObject的行为当然取决于其同步器实现的语义。
 *
 * <p>This class provides inspection, instrumentation, and monitoring
 * methods for the internal queue, as well as similar methods for
 * condition objects. These can be exported as desired into classes
 * using an {@code AbstractQueuedSynchronizer} for their
 * synchronization mechanics.
 * 该类提供对内部队列的检查、监测和监控的方法，对condition对象也有同样的方法。
 * 可以根据需要导出到类中，使用AQS作为他们的同步机制。
 *
 * <p>Serialization of this class stores only the underlying atomic
 * integer maintaining state, so deserialized objects have empty
 * thread queues. Typical subclasses requiring serializability will
 * define a {@code readObject} method that restores this to a known
 * initial state upon deserialization.
 * 该类的序列化只存储底层的用原子interger维护的状态，所以反序列化对象有空的线程队列（或者翻译为线程队列为空）。
 * 需要序列化的典型子类将定义readObject方法，在反序列化时将其恢复（restores）到已知（known）的初始化状态。
```

```
 *
 * <h3>Usage</h3>
 * 用法
 *
 * <p>To use this class as the basis of a synchronizer, redefine the
 * following methods, as applicable, by inspecting and/or modifying
 * the synchronization state using {@link #getState}, {@link
 * #setState} and/or {@link #compareAndSetState}:
 * 使用该类作为同步器基础，根据适应性（applicable）情况重新定义下面几个方法（5个方法），可以使用getState、setState与compareAndSetState来检查（inspect）和修改同步器状态。
 *
 * 下面这几个方法很关键，是实现类唯一能改的5个方法
 * <ul>
 * <li> {@link #tryAcquire}
 * <li> {@link #tryRelease}
 * <li> {@link #tryAcquireShared}
 * <li> {@link #tryReleaseShared}
 * <li> {@link #isHeldExclusively}
 * </ul>
 *
 * Each of these methods by default throws {@link
 * UnsupportedOperationException}.  Implementations of these methods
 * must be internally thread-safe, and should in general be short and
 * not block. Defining these methods is the <em>only</em> supported
 * means of using this class. All other methods are declared
 * {@code final} because they cannot be independently varied.
 * 上面的每个方法默认抛出UnsupportedOperationException异常。
 * 这些方法的实现必须是内部线程安全的，通常应该是简短并且不会阻塞的。
 * 定义这些方法，是使用该类唯一支持的方法（means）（mean是意味着）
 * 所有的其他方法都声明为final，因为他们不能独立变化（varied）
 * （就是除了上面这五个方法可以重新定义，其他方法都改不了）
 *
 * <p>You may also find the inherited methods from {@link
 * AbstractOwnableSynchronizer} useful to keep track of the thread
 * owning an exclusive synchronizer.  You are encouraged to use them
 * -- this enables monitoring and diagnostic tools to assist users in
 * determining which threads hold locks.
 * 从AbstractOwnableSynchronizer继承的方法对追踪独占同步器的线程很有用。
 * 鼓励使用这些方法 -- 监控和诊断工具能够去帮助使用者确定哪些线程持有锁
 *
 * <p>Even though this class is based on an internal FIFO queue, it
 * does not automatically enforce FIFO acquisition policies.  The core
 * of exclusive synchronization takes the form:
 * 即使该类基于内部的FIFO队列，它也不会自动执行FIFO获取策略。
 * 独占同步的核心形式为：
 *
 * <pre>
 * Acquire:
 *     while (!tryAcquire(arg)) { // 获取失败会循环
```

```
 *        <em>enqueue thread if it is not already queued</em>; // 如果没有排队，则线程排队
 *        <em>possibly block current thread</em>; // 可能阻塞当前线程
 *     }
 *
 * Release:
 *     if (tryRelease(arg)) // 释放成功
 *        <em>unblock the first queued thread</em>; // 第一个队列线程被唤醒
 * </pre>
 *
 * (Shared mode is similar but may involve cascading signals.)
 * 共享模式类似，不过可能会涉及级联信号
 *
 * 非公平锁与公平锁的大致实现
 * <p id="barging">Because checks in acquire are invoked before
 * enqueuing, a newly acquiring thread may <em>barge</em> ahead of
 * others that are blocked and queued.  However, you can, if desired,
 * define {@code tryAcquire} and/or {@code tryAcquireShared} to
 * disable barging by internally invoking one or more of the inspection
 * methods, thereby providing a <em>fair</em> FIFO acquisition order.
 * In particular, most fair synchronizers can define {@code tryAcquire}
 * to return {@code false} if {@link #hasQueuedPredecessors} (a method
 * specifically designed to be used by fair synchronizers) returns
 * {@code true}.  Other variations are possible.
 * 因为在入队之前会调用acquire进行检查，新的发起acquire的线程可能抢（barge ahead of抢先）在其它阻塞和在队列里的线程之前（拿到锁）。
 * 但是，如果需要，定义tryAcquire和/或tryAcquireShared以通过内部调用一个或多个检查方式来禁止抢占（插队），从而提供公平（fair）的FIFO获取顺序。
 * 特别的，如果hasQueuePredecessors（一个在公平同步器中使用的特定设计的方法）返回true，大多数公平同步器可以定义tryAcquire返回false。（就是等待队列不为空，就不允许插队）
 * 其他变化（variations）也是可能的。
 *
 * <p>Throughput and scalability are generally highest for the
 * default barging (also known as <em>greedy</em>,
 * <em>renouncement</em>, and <em>convoy-avoidance</em>) strategy.
 * While this is not guaranteed to be fair or starvation-free, earlier
 * queued threads are allowed to recontend before later queued
 * threads, and each recontention has an unbiased chance to succeed
 * against incoming threads.  Also, while acquires do not
 * &quot;spin&quot; in the usual sense, they may perform multiple
 * invocations of {@code tryAcquire} interspersed with other
 * computations before blocking.  This gives most of the benefits of
 * spins when exclusive synchronization is only briefly held, without
 * most of the liabilities when it isn't. If so desired, you can
 * augment this by preceding calls to acquire methods with
 * "fast-path" checks, possibly prechecking {@link #hasContended}
 * and/or {@link #hasQueuedThreads} to only do so if the synchronizer
 * is likely not to be contended.
 * 在默认抢占策略（又称greedy，renouncement和convoy-avoidance）下，吞吐量（throughput）和可
```

扩展性（scalability）最高。
 * 虽然这样不能保证公平或者没有饥饿，但是允许较早的排队进程在较晚的排队线程之前重新竞争（recontend），并且每次重新竞争都有平等的机会成功对抗新来的线程。
 *（意思就是排在队首的线程一定在队列里的其他线程之前进行锁竞争，并且队首线程与新来的还未入队的线程竞争锁具有平等的概率）
 * 虽然acquires在通常意义上不会自旋（自旋就是重复操作直到某个状态退出，类似于while(cond){...}），但他们可能会在阻塞之前执行多次调用tryAcquire并穿插其他计算。
 * 在独占同步只短暂持有时，提供的这种自旋的方式具有很大的好处；如果不是，也没有多大的坏处。（如果锁占有时间短暂，可能在自旋过程中就能拿到锁，减少了阻塞再唤醒的消耗）
 * 如果有需求，你可以通过使用“快速路径”检查来预先调用acquire方法以增强这一点，可能预先检查hasContended 和/或hasQueuedThreads方法，以仅在如果同步器很可能没有竞争时才这样做。
 * （预先检查有没有竞争的情况，如果有不竞争的可能性，就通过自旋的方式来尝试获取锁）
 *
 * <p>This class provides an efficient and scalable basis for
 * synchronization in part by specializing its range of use to
 * synchronizers that can rely on {@code int} state, acquire, and
 * release parameters, and an internal FIFO wait queue. When this does
 * not suffice, you can build synchronizers from a lower level using
 * {@link java.util.concurrent.atomic atomic} classes, your own custom
 * {@link java.util.Queue} classes, and {@link LockSupport} blocking
 * support.
 * 该类对于部分是通过将其使用范围专门用于依赖int状态、获取和释放参数以及内部FIFO等待队列的同步器，从而为同步提供有效（efficient）和可扩展的基础。
 * 如果这不能满足（suffice），可以通过使用atomic类、自定义Queue类和LockSupprot阻塞支持从较低级别构建同步器
 *
 * <h3>Usage Examples</h3>
 * 使用样例
 *
 * <p>Here is a non-reentrant mutual exclusion lock class that uses
 * the value zero to represent the unlocked state, and one to
 * represent the locked state. While a non-reentrant lock
 * does not strictly require recording of the current owner
 * thread, this class does so anyway to make usage easier to monitor.
 * It also supports conditions and exposes
 * one of the instrumentation methods:
 * 这里是一个不可重入的互斥锁class，用0代表解锁状态，用1代表加锁状态。
 * 虽然不可重入锁不严格要求记录当前用有锁的线程，但这个类无论如何这样做是为了让使用更容易监控。
 * 它还支持条件并公开一种检测方法：
 *
 * <pre> {@code
 * class Mutex implements Lock, java.io.Serializable { // mutex 是信号量
 *
 *   // Our internal helper class
 *   private static class Sync extends AbstractQueuedSynchronizer {
 *   // 继承AQS的，独占锁，得自己实现两个try方法（tryAcquire与tryRelease）与isHeldExclusively方法，因为AQS没有定义这仨实际操作。！！！！！！
 *   // Reports whether in locked state
 *   protected boolean isHeldExclusively() { // 当前同步器是否在独占模式下被线程占用，一

```
般该方法表示是否被当前线程所独占
 *       return getState() == 1;
 *     }
 *
 *     // Acquires the lock if state is zero
 *     public boolean tryAcquire(int acquires) {
 *       assert acquires == 1; // Otherwise unused // assert是java的关键字--断言，如果表
达式成立，则继续执行，否则抛出AssertionError，并终止执行。
 *       if (compareAndSetState(0, 1)) { // 使用CAS来设置加锁
 *         setExclusiveOwnerThread(Thread.currentThread());
 *         return true;
 *       }
 *       return false;
 *     }
 *
 *     // Releases the lock by setting state to zero
 *     protected boolean tryRelease(int releases) {
 *       assert releases == 1; // Otherwise unused
 *       if (getState() == 0) throw new IllegalMonitorStateException();
 *       setExclusiveOwnerThread(null);
 *       setState(0); // 不使用CAS，直接解锁
 *       return true;
 *     }
 *
 *     // Provides a Condition
 *     // 需要用到Conditionobject的，得自己写newCondition方法，AQS不提供这个方法，只提供Con
ditionObject这个内部类
 *     Condition newCondition() { return new ConditionObject(); }
 *
 *     // Deserializes properly // 正确反序列化
 *     private void readObject(ObjectInputStream s)
 *         throws IOException, ClassNotFoundException {
 *       s.defaultReadObject();
 *       setState(0); // reset to unlocked state // 重置为不加锁状态
 *     }
 *   }
 *
 *   // The sync object does all the hard work. We just forward to it.
 *   private final Sync sync = new Sync();
 *
 *   // 通过Sync实现Lock接口的一些功能
 *   public void lock()                { sync.acquire(1); }
 *   public boolean tryLock()          { return sync.tryAcquire(1); }
 *   public void unlock()              { sync.release(1); }
 *   public Condition newCondition()   { return sync.newCondition(); }
 *   public boolean isLocked()         { return sync.isHeldExclusively(); }
 *   public boolean hasQueuedThreads() { return sync.hasQueuedThreads(); }
 *   public void lockInterruptibly() throws InterruptedException {
 *     sync.acquireInterruptibly(1);
```

```
 *   }
 *   public boolean tryLock(long timeout, TimeUnit unit)
 *       throws InterruptedException {
 *     return sync.tryAcquireNanos(1, unit.toNanos(timeout));
 *   }
 * }}</pre>
 *
 * <p>Here is a latch class that is like a
 * {@link java.util.concurrent.CountDownLatch CountDownLatch}
 * except that it only requires a single {@code signal} to
 * fire. Because a latch is non-exclusive, it uses the {@code shared}
 * acquire and release methods.
 * 这里是一个类似于CountDownLatch的闩锁类，只需要单信号量就可以触发。
 * 因为latch是个非独占的，它使用共享的acquire与release方法
 *
 *  <pre> {@code
 * class BooleanLatch {
 *
 *   private static class Sync extends AbstractQueuedSynchronizer {
 *     // 用AQS实现共享锁，得自己实现tryAcquireShared和tryReleaseShared方法
 *     boolean isSignalled() { return getState() != 0; } // 是否有信号，state初始化为0
 *
 *     protected int tryAcquireShared(int ignore) {
 *       return isSignalled() ? 1 : -1;
 *     }
 *
 *     protected boolean tryReleaseShared(int ignore) {
 *       setState(1);
 *       return true;
 *     }
 *   }
 *
 *   private final Sync sync = new Sync();
 *   public boolean isSignalled() { return sync.isSignalled(); }
 *   public void signal()         { sync.releaseShared(1); } //
 *   public void await() throws InterruptedException {  // 如果上来就调用await，那么因为state=0不满足条件，当前线程进入等待队列。如果现在state=1了，那么就不会阻塞直接执行
 *     sync.acquireSharedInterruptibly(1);
 *   }
 * }}</pre>
 *
 * @since 1.5
 * @author Doug Lea
 */
public abstract class AbstractQueuedSynchronizer
    extends AbstractOwnableSynchronizer // 用来记录当前独占锁的拥有者（拥有者是个Thread对象）
    implements java.io.Serializable {
```

```
    private static final long serialVersionUID = 7373984972572414691L;

    /**
     * Creates a new {@code AbstractQueuedSynchronizer} instance
     * with initial synchronization state of zero.
     * 初始化时，表示状态的state为0
     */
    protected AbstractQueuedSynchronizer() { }

    /**
     * Wait queue node class.
     * 等待队列节点类
     *
     * <p>The wait queue is a variant of a "CLH" (Craig, Landin, and
     * Hagersten) lock queue. CLH locks are normally used for
     * spinlocks.  We instead use them for blocking synchronizers, but
     * use the same basic tactic of holding some of the control
     * information about a thread in the predecessor of its node.  A
     * "status" field in each node keeps track of whether a thread
     * should block.  A node is signalled when its predecessor
     * releases.  Each node of the queue otherwise serves as a
     * specific-notification-style monitor holding a single waiting
     * thread. The status field does NOT control whether threads are
     * granted locks etc though.  A thread may try to acquire if it is
     * first in the queue. But being first does not guarantee success;
     * it only gives the right to contend.  So the currently released
     * contender thread may need to rewait.
     * 等待队列是CLH锁队列的变种（variant）。CLH锁通常用于自旋锁（spinlock）。
     * 我们改为将它用于阻塞同步器，但也用相同的基本策略（tactic），即在node的前驱（predecessor）中保存有关线程的一些控制信息
     * 每个节点中的“state”字段保持跟踪线程是否应该被阻塞。
     * 节点在其前驱解锁（releases）的时候收到信号（理解为唤醒）。
     * 队列中的每个node都充当一个特定通知式监视器，持有一个等待线程。（一个node里面包含了一个waiting线程对象）
     * 尽管status属性不会控制线程是否被授予锁。（status属性只是用来表名可以去竞争锁，不管会不会加锁成功）
     * 如果线程是队列里的第一个，它可能尝试去加锁（acquire）
     * 但是作为第一个不会保证一定能加锁成功；它只是被给予了去竞争的权利。（在unfair非公平锁里，队列的第一个线程要跟尚未入队的竞争线程一起竞争锁）
     * 所以当前释放的竞争者线程（也就是被唤醒的线程或者队列里的第一个线程）可能需要重新等待。
     *
     * <p>To enqueue into a CLH lock, you atomically splice it in as new
     * tail. To dequeue, you just set the head field.
     * CLH锁进队，需要原子性的将它拼接为新的尾部（tail）。
     * 出队，只需要设置它的头部（head）字段。
     * <pre>
     *      +------+  prev +-----+       +-----+
     * head |      | <---- |     | <---- |     |  tail
     *      +------+       +-----+       +-----+
```

```
     * </pre>
     *
     * <p>Insertion into a CLH queue requires only a single atomic
     * operation on "tail", so there is a simple atomic point of
     * demarcation from unqueued to queued. Similarly, dequeuing
     * involves only updating the "head". However, it takes a bit
     * more work for nodes to determine who their successors are,
     * in part to deal with possible cancellation due to timeouts
     * and interrupts.
     * 插入CLH队列只需要对“tail”进行一次原子性操作，所以从未入队到入队有一个简单的原子分界点。（入队仅经过一个原子性操作）
     * 类似的，出队涉及只更新“head”。
     * 然而，节点需要做更多的工作来确定他们的后继（successors）是谁，部分是为了处理由于超时和中断可能导致的取消。
     *
     * 下面会讲取消的问题
     * <p>The "prev" links (not used in original CLH locks), are mainly
     * needed to handle cancellation. If a node is cancelled, its
     * successor is (normally) relinked to a non-cancelled
     * predecessor. For explanation of similar mechanics in the case
     * of spin locks, see the papers by Scott and Scherer at
     * http://www.cs.rochester.edu/u/scott/synchronization/
     * “prev”连接（在原始CLH锁中未使用）主要用于处理取消。
     * 如果一个node被取消，它的后继（通常）会重新连接到一个未取消的前驱。（就是一个节点被取消了，那么这个节点应该从CLH链上删除，这时候就需要它的后继去重新找到未取消的前驱）
     * 有关自旋锁情况下的类似机制解释，请参阅链接的论文
     *
     * <p>We also use "next" links to implement blocking mechanics.
     * The thread id for each node is kept in its own node, so a
     * predecessor signals the next node to wake up by traversing
     * next link to determine which thread it is.  Determination of
     * successor must avoid races with newly queued nodes to set
     * the "next" fields of their predecessors.  This is solved
     * when necessary by checking backwards from the atomically
     * updated "tail" when a node's successor appears to be null.
     * (Or, said differently, the next-links are an optimization
     * so that we don't usually need a backward scan.)
     * 我们还使用“next”连接来实现阻塞机制。
     * 每个节点的线程ID保存在他们自己的node里，因此前驱信号是通过遍历next连接来确定它是哪个线程来通知唤醒下一个节点。？？？
     * 确定后驱节点必须避免与新排队节点竞争以设置其前驱节点的“next”字段。（就是设置其前驱节点的next值时不要与新入队的节点发生冲突）
     * 当一个节点的后驱出现空时，如果必要，从原子更新的“tail”向后检查来解决。（换句话说，next连接是一种优化（optimization），所以我们通常不需要向后扫描）？？？
     *
     * <p>Cancellation introduces some conservatism to the basic
     * algorithms.  Since we must poll for cancellation of other
     * nodes, we can miss noticing whether a cancelled node is
     * ahead or behind us. This is dealt with by always unparking
```

```
     * successors upon cancellation, allowing them to stabilize on
     * a new predecessor, unless we can identify an uncancelled
     * predecessor who will carry this responsibility.
     * 取消（Canellation）为基础算法引入了些保守性。由于我们必须轮询（poll）其他节点的取消，因此我们可能无法注意到取消的node是在我们之前还是之后。
     * 通过在取消时总是解除后继来处理的，允许他们稳定在一个新的前驱上，除非我们可以确定一个未取消的前驱将承担这个责任。
     *
     * <p>CLH queues need a dummy header node to get started. But
     * we don't create them on construction, because it would be wasted
     * effort if there is never contention. Instead, the node
     * is constructed and head and tail pointers are set upon first
     * contention.
     * CLH队列需要一个虚拟（dummy）头结点来启动。但是我们不会在构建时创建他们，因为如果没有从来没有竞争，这个虚拟头就是浪费。
     * 取而代之的是，在第一次竞争的时候，这个虚拟头节点将创建并设置head跟tail指针
     *
     * <p>Threads waiting on Conditions use the same nodes, but
     * use an additional link. Conditions only need to link nodes
     * in simple (non-concurrent) linked queues because they are
     * only accessed when exclusively held.  Upon await, a node is
     * inserted into a condition queue.  Upon signal, the node is
     * transferred to the main queue.  A special value of status
     * field is used to mark which queue a node is on.
     * 在Condition上等待的线程使用相同的node，不过使用额外的连接（就是说有一个是CLH主链（主链=主队列），还有一些是针对不同的Condition建立的不同的链）
     * Condition只需要连接在简单的（非并发）链接队列上的node，因为他们仅在独占时才会被访问。
     * 根据信号，node被转移到主队列上。（这个信号就是能够满足独占需求的信号，会将对应的Condition链转移到主队列上）
     * status字段的特殊值用来标记node所在的队列。
     *
     * <p>Thanks go to Dave Dice, Mark Moir, Victor Luchangco, Bill
     * Scherer and Michael Scott, along with members of JSR-166
     * expert group, for helpful ideas, discussions, and critiques
     * on the design of this class.
     * 大佬感谢大佬们的时间，不看了
     */
    static final class Node {
        /** Marker to indicate a node is waiting in shared mode */
        // 表明在共享模式下等待的node
        static final Node SHARED = new Node();
        /** Marker to indicate a node is waiting in exclusive mode */
        // 表明在独占模式下等待的node
        static final Node EXCLUSIVE = null;

        /** waitStatus value to indicate thread has cancelled */
        // 等待status值=1 表示线程取消
        static final int CANCELLED =  1;
        /** waitStatus value to indicate successor's thread needs unparking */
```

```
        // 等待status值=-1 表示后继的线程需要唤醒
        static final int SIGNAL    = -1;
        /** waitStatus value to indicate thread is waiting on condition */
        // 等待status值=-2 表示线程在等待条件（或者说线程在condition队列上）
        static final int CONDITION = -2;
        /**
         * waitStatus value to indicate the next acquireShared should
         * unconditionally propagate
         */
         // waitStatus值=-3 表示下一个acquireShared应该无条件传播
        static final int PROPAGATE = -3;

        /**
         * Status field, taking on only the values:
         *   SIGNAL:     The successor of this node is (or will soon be)
         *               blocked (via park), so the current node must
         *               unpark its successor when it releases or
         *               cancels. To avoid races, acquire methods must
         *               first indicate they need a signal,
         *               then retry the atomic acquire, and then,
         *               on failure, block.
         *               当前节点的后继节点是被阻塞的，所以当前节点在释放或者取消的时候，必须unpark他的后继节点。
         *               为了避免竞争，加锁方法必须首先声明他们需要一个信号，然后重试原子操作的加锁，然后在失败时阻塞。
         *   CANCELLED:  This node is cancelled due to timeout or interrupt.
         *               Nodes never leave this state. In particular,
         *               a thread with cancelled node never again blocks.
         *               当前节点由于超时或者中断被取消。
         *               节点从来不会离开这个状态。特别是，取消节点的线程永远不会再阻塞。
         *   CONDITION:  This node is currently on a condition queue.
         *               It will not be used as a sync queue node
         *               until transferred, at which time the status
         *               will be set to 0. (Use of this value here has
         *               nothing to do with the other uses of the
         *               field, but simplifies mechanics.)
         *               当前节点在条件队列中。
         *               它在传输之前不会用作同步节点，到那个时候status将被设置为0。（在这里使用这个值和这个字段的其他用法无关，但是简化了机制）
         *   PROPAGATE:  A releaseShared should be propagated to other
         *               nodes. This is set (for head node only) in
         *               doReleaseShared to ensure propagation
         *               continues, even if other operations have
         *               since intervened.
         *               共享锁的释放（releaseShared）应当传播到其他节点。
         *               这在doReleaseShared中设置（仅适用于头节点）以确保传播继续，即使其他的操作已经介入。
         *   0:          None of the above
         *               不处于以上情况的status值就是0
```

```
         *
         * The values are arranged numerically to simplify use.
         * Non-negative values mean that a node doesn't need to
         * signal. So, most code doesn't need to check for particular
         * values, just for sign.
         * 值按数字化排列以简化使用。非负值意味着节点不需要发出信号。
         * 所以，大多数代码不需要检查特定值，只需要检查符号。
         *
         * The field is initialized to 0 for normal sync nodes, and
         * CONDITION for condition nodes.  It is modified using CAS
         * (or when possible, unconditional volatile writes).
         * 该字段对于普通的同步节点初始化为0，对于condition节点初识化为CONDIITON。使用CAS进
行修改（或者可以的话，使用无条件的volatile写入）。
         */
        volatile int waitStatus;

        /**
         * Link to predecessor node that current node/thread relies on
         * for checking waitStatus. Assigned during enqueuing, and nulled
         * out (for sake of GC) only upon dequeuing.  Also, upon
         * cancellation of a predecessor, we short-circuit while
         * finding a non-cancelled one, which will always exist
         * because the head node is never cancelled: A node becomes
         * head only as a result of successful acquire. A
         * cancelled thread never succeeds in acquiring, and a thread only
         * cancels itself, not any other node.
         * 连接到当前线程/节点依赖检查waitStatus的前驱节点。
         * 在入队期间分配，并仅在出队时取消（为了GC）。
         * 同样，在前驱cancel时，当找到一个未取消的node之前进行短路，未取消的node始终存在，因
为头节点从来不会cancel：一个节点要成为头结点，只有成功获取到结果。
         * 取消的线程在加锁时永远不会成功，并且线程只能取消自己，不能取消其他node。
         */
        volatile Node prev;

        /**
         * Link to the successor node that the current node/thread
         * unparks upon release. Assigned during enqueuing, adjusted
         * when bypassing cancelled predecessors, and nulled out (for
         * sake of GC) when dequeued.  The enq operation does not
         * assign next field of a predecessor until after attachment,
         * so seeing a null next field does not necessarily mean that
         * node is at end of queue. However, if a next field appears
         * to be null, we can scan prev's from the tail to
         * double-check.  The next field of cancelled nodes is set to
         * point to the node itself instead of null, to make life
         * easier for isOnSyncQueue.
         * 连接到当前节点/线程在解锁时需要unpark的后继节点。
         * 在入队时分配，在绕过取消的前驱时调整，在出队时取消（置为null）（为了GC）。
         * enq操作直到连接后才分配前驱的next字段，因此看到next字段为null时不一定意味着节点是
```

```
尾结点。
         * 然而，如果next字段为null，可以从tail开始扫描上一个字段以进行二次检查。
         * 取消节点的next字段指向该节点自己而不是null，以使isOnSyncQueue的工作更轻松。
         */
        volatile Node next;

        /**
         * The thread that enqueued this node.  Initialized on
         * construction and nulled out after use.
         * 入队node的线程。
         * 在构造时初始化，在使用后置为null。
         */
        volatile Thread thread;

        /**
         * Link to next node waiting on condition, or the special
         * value SHARED.  Because condition queues are accessed only
         * when holding in exclusive mode, we just need a simple
         * linked queue to hold nodes while they are waiting on
         * conditions. They are then transferred to the queue to
         * re-acquire. And because conditions can only be exclusive,
         * we save a field by using special value to indicate shared
         * mode.
         * 连接到下一个等待条件（在条件上等待）的节点，或者特殊值SHARED。
         * 因为条件队列（condition queues）只有在独占模式下才会被访问，所以当他们正在等待条件
时，我们只需要一个简单的链接队列来保存节点。
         * 然后将他们转移到主队列上来重新加锁。因为条件只能是独占的，所以我们通过使用特殊值来
表名共享模式来保存字段。
         */
        Node nextWaiter;

        /**
         * Returns true if node is waiting in shared mode.
         */
        final boolean isShared() {
            return nextWaiter == SHARED;
        }

        /**
         * Returns previous node, or throws NullPointerException if null.
         * Use when predecessor cannot be null.  The null check could
         * be elided, but is present to help the VM.
         * 返回前驱节点，如果前驱为空，抛出NullPointerException
         * 当前驱不能为空时使用。
         * 空检查可以省略，但是对VM有帮助。
         *
         * @return the predecessor of this node
         */
        final Node predecessor() throws NullPointerException {
```

```
            Node p = prev;
            if (p == null)
                throw new NullPointerException();
            else
                return p;
        }

        Node() {    // Used to establish initial head or SHARED marker // 独占的可以用来初始化头结点，共享的可以建立SHARED标记
        }

        Node(Thread thread, Node mode) {     // Used by addWaiter
            this.nextWaiter = mode;
            this.thread = thread;
        }

        Node(Thread thread, int waitStatus) { // Used by Condition
            this.waitStatus = waitStatus;
            this.thread = thread;
        }
    }

    /**
     * Head of the wait queue, lazily initialized.  Except for
     * initialization, it is modified only via method setHead.  Note:
     * If head exists, its waitStatus is guaranteed not to be
     * CANCELLED.
     * 等待队列的头，延迟初始化。
     * 除了初始化，只能使用setHead方法来修改
     * 提示：如果head存在，保证waitStatus不是CANCELLED
     */
    private transient volatile Node head;

    /**
     * Tail of the wait queue, lazily initialized.  Modified only via
     * method enq to add new wait node.
     * 等待队列的尾部，懒初始化。
     * 只能通过enq方法，在增加新的等待node时修改。
     */
    private transient volatile Node tail;

    /**
     * The synchronization state.
     * 同步状态（获取锁的状态）（用volatile修饰，内存可见）
     */
    private volatile int state;

    /**
     * Returns the current value of synchronization state.
```

```
     * This operation has memory semantics of a {@code volatile} read.
     * state由volatile修饰，这个操作有内存读的语义
     * @return current state value
     */
    protected final int getState() {
        return state;
    }

    /**
     * Sets the value of synchronization state.
     * This operation has memory semantics of a {@code volatile} write.
     * 通过volatile实现的内存语义，如果不安全，可以用下面的compareAndSetState方法。
     * （这个方法通常被实现AQS的子类来调用，由子类决定如何更新state值）
     *
     * @param newState the new state value
     */
    protected final void setState(int newState) {
        state = newState;
    }

    /**
     * Atomically sets synchronization state to the given updated
     * value if the current state value equals the expected value.
     * This operation has memory semantics of a {@code volatile} read
     * and write.
     * 用原子操作来设定state值
     * （这个方法通常被实现AQS的子类来调用，由子类决定如何更新state值）
     *
     * @param expect the expected value
     * @param update the new value
     * @return {@code true} if successful. False return indicates that the actual
     *         value was not equal to the expected value.
     */
    protected final boolean compareAndSetState(int expect, int update) {
        // See below for intrinsics setup to support this
        return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
    }

    // Queuing utilities
    // 队列相关有益操作

    /**
     * The number of nanoseconds for which it is faster to spin
     * rather than to use timed park. A rough estimate suffices
     * to improve responsiveness with very short timeouts.
     * 自旋比使用定时park具有更快的纳秒数。粗略估计满足在非常短的超时时间内提高响应。
     * （自旋的时间，单位：纳秒）
     */
    static final long spinForTimeoutThreshold = 1000L;
```

```
    /**
     * Inserts node into queue, initializing if necessary. See picture above.
     * node入队，必要的话初始化。上面有图。
     * @param node the node to insert
     * @return node's predecessor
     */
    private Node enq(final Node node) {
        for (;;) {  // 这个循环是为了，如果CAS加锁失败，通过循环来重新加锁或者执行其他操作
            Node t = tail;
            if (t == null) { // Must initialize // 队尾为null，说明现在队列里啥也没有，需要初始化队列（主要就是初始化队列Head与Tail，用来指向队列头尾）
                if (compareAndSetHead(new Node())) // 如果Head为null，则设置为New Node()
                    tail = head; // 新增一个空节点，头和尾都指向同一个节点
            } else {
                node.prev = t; // 当前节点的prev指向尾指针指向的节点
                if (compareAndSetTail(t, node)) { // CAS修改tail指向的节点为当前节点
                    t.next = node; // 原来的最后一个节点next指向这个node
                    return t;
                }
            }
        }
    }

    /**
     * Creates and enqueues node for current thread and given mode.
     * 为当前线程和给定模式创建和入队node
     * （用当前线程创建Node，根据指定的模式入队）
     *
     * @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared
     * @return the new node
     */
    private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode); // addWaiter的构造方法
        // Try the fast path of enq; backup to full enq on failure
        // 尝试简化enq过程；
        Node pred = tail; // 前驱=tail（注意不是head）
        if (pred != null) {  // 有tail，就简化enq(...)方法
            node.prev = pred; // node.prev->tail
            if (compareAndSetTail(pred, node)) { // tail -> node
                pred.next = node; // 原来最后一个节点的next指向这个node，这个node的next是null
                return node;
            }
        }
        enq(node); // 上面的不行再走enq方法
        return node;
    }
```

```
    /**
     * Sets head of queue to be node, thus dequeuing. Called only by
     * acquire methods.  Also nulls out unused fields for sake of GC
     * and to suppress unnecessary signals and traversals.
     * 设置队列的head为该节点，从而出队。只能通过加锁方法调用。
     * 将不使用的字段设置为null，为了GC与抑制不必要的信号与遍历。
     * （相当于在该节点加锁成功时（就是成功获取到了锁，不需要再排队了），把当前节点设置为了原来的虚拟节点作为head）
     *
     * @param node the node
     */
    private void setHead(Node node) {
        head = node;
        node.thread = null;
        node.prev = null;
    }

    /**
     * Wakes up node's successor, if one exists.
     * 如果存在，唤醒node的后继
     *
     * @param node the node
     */
    private void unparkSuccessor(Node node) {
        /*
         * If status is negative (i.e., possibly needing signal) try
         * to clear in anticipation of signalling.  It is OK if this
         * fails or if status is changed by waiting thread.
         * 如果状态为负值（可能需要信号量）尝试清楚以期待信号。
         * 如果清除状态失败或者状态已经被等待线程修改了，也没问题。
         *
         */
        int ws = node.waitStatus;
        if (ws < 0) // 只有cancelled是1，其他的signal、condition、propagate都是负值
            compareAndSetWaitStatus(node, ws, 0); // 0就是正常等待的node的waitStatus值，表示该节点正在被操作？？？其他关于该节点的操作可以等等

        /*
         * Thread to unpark is held in successor, which is normally
         * just the next node.  But if cancelled or apparently null,
         * traverse backwards from tail to find the actual
         * non-cancelled successor.
         * unpark线程被保存在后继节点中，通常是next指向的下一个节点。
         * 如果next节点被取消或者明显为null，从tail回溯找到实际上没有取消的后继者（为什么要从tail回溯？因为next可能是null，没法从前遍历）
         */
        Node s = node.next;
        if (s == null || s.waitStatus > 0) { // 后继为null或者被取消
            s = null;
```

```
            for (Node t = tail; t != null && t != node; t = t.prev) // 找到离当前node最近的未取消的非空（后继）node （当前node的prev是null，所以找到当前node之后就会终止）
                if (t.waitStatus <= 0)
                    s = t;
        }
        if (s != null)
            LockSupport.unpark(s.thread); // 唤醒后继，竞争锁
    }

    /**
     * Release action for shared mode -- signals successor and ensures
     * propagation. (Note: For exclusive mode, release just amounts
     * to calling unparkSuccessor of head if it needs signal.)
     * 共享模式的释放（解锁）动作 -- 信号通知后继者并且确保广播。
     * （提示：对于独占模式，如果需要信号量，解锁只是相当于唤醒head的后继者）
     * 这个解锁动作，实际上是将节点从park状态唤醒（调用unpark），而不是释放竞争锁
     */
    private void doReleaseShared() {
        /*
         * Ensure that a release propagates, even if there are other
         * in-progress acquires/releases.  This proceeds in the usual
         * way of trying to unparkSuccessor of head if it needs
         * signal. But if it does not, status is set to PROPAGATE to
         * ensure that upon release, propagation continues.
         * Additionally, we must loop in case a new node is added
         * while we are doing this. Also, unlike other uses of
         * unparkSuccessor, we need to know if CAS to reset status
         * fails, if so rechecking.
         * 即使有其他线程正在加锁/解锁，也要确保release广播。
         * 通常尝试去unpark head的后继，如果它需要信号的话。如果不是这样，设置head的后继的status为PROPAGATE，以确保release时传播继续。
         * 此外，必须loop循环以防止在我们这样做时有新的节点加入。
         * 此外，不同于其他的unparkSuccessor，必须知道使用CAS重置status是否失败，如果失败则重新检查。
         */
        for (;;) {
            Node h = head;
            if (h != null && h != tail) {
                int ws = h.waitStatus;
                if (ws == Node.SIGNAL) { // 当前node的ws为SIGNAL，则表示后继节点需要信号（也就是需要唤醒的信号）
                    if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0)) // 将head节点的waitStatus由SIGNAL更新为0
                        continue;            // loop to recheck cases // 循环，再次从头开始设置，直到head由SIGNAL状态转为0（因为当前队列可能会有新的节点出队/入队）
                    unparkSuccessor(h); // 唤醒后继节点
                }
                else if (ws == 0 &&
                         !compareAndSetWaitStatus(h, 0, Node.PROPAGATE)) // 后继节点不需
```

```
要信号量，那么直接设置ws为PROPAGATE，确保release的时候传播继续
                    continue;                // loop on failed CAS
                }
                if (h == head)                   // loop if head changed // 如果head没变，就
退出
                    break;
            }
        }

        /**
         * Sets head of queue, and checks if successor may be waiting
         * in shared mode, if so propagating if either propagate > 0 or
         * PROPAGATE status was set.
         * 设置队列头，并且检查它的后继者是否在共享模式下等待。
         * 如果propagate > 0，或者设置了PROPAGATE状态，则进行传播。
         *
         * @param node the node
         * @param propagate the return value from a tryAcquireShared
         * propagate入参值是tryAcquireShared方法的返回值。
         */
        private void setHeadAndPropagate(Node node, int propagate) {
            Node h = head; // Record old head for check below
            setHead(node); // 设置head指向node，并清除thread、prev的值为null（因为当前线程一定
是成功获取到锁了，所以直接置为head，表示线程已执行，变成了虚拟head）
            /*
             * Try to signal next queued node if:
             *   Propagation was indicated by caller,
             *     or was recorded (as h.waitStatus either before
             *     or after setHead) by a previous operation
             *     (note: this uses sign-check of waitStatus because
             *      PROPAGATE status may transition to SIGNAL.)
             * and
             *   The next node is waiting in shared mode,
             *     or we don't know, because it appears null
             *
             * The conservatism in both of these checks may cause
             * unnecessary wake-ups, but only when there are multiple
             * racing acquires/releases, so most need signals now or soon
             * anyway.
             * 尝试向下一个队列里的node发出信号，如果：
             * 1. 调用者明确指示广播（Propagation），或者被前一个操作记录（作为h.waitStataus，在
setHead之前或者之后）。（注意：waitStatus上的信号检查，因为PROPAGATE状态可能会转化为SIGNAL状
态）
             * 2. 下一个节点在共享模式下等待，或者我们不知道，因为它看起来是null。
             * 这个两项检查的保守性可能会导致不必要的唤醒，但是仅当有多个竞争加锁/解锁时，大多数很
快就会需要信号。
             */

            // h == null 的判断是防止空指针异常。
```

```
        // h.waitStatus < 0，表示 h 处于 SIGNAL 或 PROPAGATE 状态，一般情况下是 PROPAGATE
 状态，因为在 doReleaseShared 方法中 h 状态变化是 SIGNAL -> 0 -> PROPAGATE。
        // 那么为什么 SIGNAL 状态也要唤醒呢？这是因为在 doAcquireShared 中，第一次没有获得足
够的资源时，shouldParkAfterFailedAcquire 将 PROPAGATE 状态转换成 SIGNAL，准备阻塞线程，
        // 但是第二次进入本方法时发现资源刚好够，而此时 h 的状态是 SIGNAL 状态
        // (h = head) == null 是再次检查
        if (propagate > 0 || h == null || h.waitStatus < 0 ||
            (h = head) == null || h.waitStatus < 0) { // 重新指向head，再判断
            Node s = node.next;
            if (s == null || s.isShared()) // 后继为null或者后继为共享模式
                doReleaseShared(); // 共享模式的解锁
        }
    }

    // Utilities for various versions of acquire
    // 各种版本的加锁

    /**
     * Cancels an ongoing attempt to acquire.
     * 取消正在进行的加锁尝试
     *
     * @param node the node
     */
    private void cancelAcquire(Node node) {
        // Ignore if node doesn't exist
        if (node == null)
            return;

        node.thread = null;

        // Skip cancelled predecessors
        // 跳过已取消的前驱（一个要取消加锁的node为啥还有前驱，可以从中间取消么？？？）
        Node pred = node.prev;
        while (pred.waitStatus > 0)
            node.prev = pred = pred.prev; // 不断修改当前节点的前驱

        // predNext is the apparent node to unsplice. CASes below will
        // fail if not, in which case, we lost race vs another cancel
        // or signal, so no further action is necessary.
        // predNext是要取消拼接的明显节点（就是这个节点要退出队列，不在队列链上）。
        // 如果没有，下面的CAS将失败，在这种场景下，在与另一个cancel或者signal竞争中输了，所
以不需要采取后续操作。
        Node predNext = pred.next;

        // Can use unconditional write instead of CAS here.
        // After this atomic step, other Nodes can skip past us.
        // Before, we are free of interference from other threads.
        // 在这里可使用无条件的写而不是用CAS
        // 在这个原子步骤之后，其他节点可以跳过我们
```

```
        // 之前，不受其他线程的干扰（waitStatus由volatile提供内存可见性）
        node.waitStatus = Node.CANCELLED;

        // If we are the tail, remove ourselves.
        // 如果待取消的节点是在队尾，直接移除
        if (node == tail && compareAndSetTail(node, pred)) { // 设置tail为该节点的前驱节点

            compareAndSetNext(pred, predNext, null); // 将该节点前驱节点的next设置为null（断开与该节点的关联，为了GC与其他操作）
        } else {
            // If successor needs signal, try to set pred's next-link
            // so it will get one. Otherwise wake it up to propagate.
            // 如果待取消的节点不在队尾（在队列中间（不能为队首，因为队首是个虚拟节点））
            // 如果后继需要signal，尝试设置前驱节点的下一个连接，这样就会得到一个。
            // 否则唤醒它进行广播（propagate）
            int ws;
            if (pred != head && // 如果该节点前面是队首，说明它前面没有等待的节点了
                ((ws = pred.waitStatus) == Node.SIGNAL || // 前驱节点的ws本来就是SIGNAL
                 (ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) && // 前驱节点状态非默认与取消，并且更新前驱的ws为SIGNAL成功
                pred.thread != null) { // 前驱节点里线程不为null
                Node next = node.next;
                if (next != null && next.waitStatus <= 0) // 当前节点的后继节点不是空，并且没有取消
                    compareAndSetNext(pred, predNext, next); // 设置前驱节点的next指向该节点的后继节点
            } else {
                unparkSuccessor(node); // 这个否则很灵性，如果该节点是head之后的第一个节点，（或者是上面的条件不满足）那么它取消之后就得直接唤醒后继节点了。
            }

            node.next = node; // help GC // 打断当前节点与其他节点的关联，方便GC
        }
    }

    /**
     * Checks and updates status for a node that failed to acquire.
     * Returns true if thread should block. This is the main signal
     * control in all acquire loops.  Requires that pred == node.prev.
     * 检查和更新加锁失败的节点状态。
     * 如果线程需要阻塞，返回true。这是在所有加锁循环中主要的信号控制。
     * 要求pred == node.prev
     *
     * @param pred node's predecessor holding status // 节点的前驱持有状态
     * @param node the node // 当前节点
     * @return {@code true} if thread should block
     */
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
```

```
        if (ws == Node.SIGNAL)
            /*
             * This node has already set status asking a release
             * to signal it, so it can safely park.
             * 该node已经设置了status，要求解锁时通知它（这个翻译不一定准，等看具体调用），所以它可以安全park。
             */
            return true;
        if (ws > 0) {
            /*
             * Predecessor was cancelled. Skip over predecessors and
             * indicate retry.
             * 前驱节点已经取消了。跳过取消节点重试找到未取消的。
             */
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
            /*
             * waitStatus must be 0 or PROPAGATE.  Indicate that we
             * need a signal, but don't park yet.  Caller will need to
             * retry to make sure it cannot acquire before parking.
             * 到了这里waitStatus一定是0或者PROPAGATE（-3）。声明需要一个signal，但park还没有执行。？？？
             * 调用者应该重试以确保在park前无法加锁？？？
             */
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }

    /**
     * Convenience method to interrupt current thread.
     * 中断当前线程的便捷（convenience）方法
     */
    static void selfInterrupt() {
        Thread.currentThread().interrupt();
    }

    /**
     * Convenience method to park and then check if interrupted
     * park和之后检查是否中断的便捷方法
     *
     * @return {@code true} if interrupted
     */
    private final boolean parkAndCheckInterrupt() {
        LockSupport.park(this);  // park会响应中断，中断发生时会设置interrupted值，不会抛出异常
```

```
        return Thread.interrupted(); // 返回检查到的线程中断状态，并清除中断状态
    }

    /*
     * Various flavors of acquire, varying in exclusive/shared and
     * control modes.  Each is mostly the same, but annoyingly
     * different.  Only a little bit of factoring is possible due to
     * interactions of exception mechanics (including ensuring that we
     * cancel if tryAcquire throws exception) and other control, at
     * least not without hurting performance too much.
     * 不同形式的加锁，在独占/共享和控制模式下各有不同。
     * 每一个都大致相同，但总有不同（annoyingly 恼人的）。
     * 由于异常机制（包括确保我们在tryAcquire抛出的异常时cancel）和其他控制的相互作用，只能进行一点点分解，至少在不会过多损耗性能的前提下进行。
     *
     */

    /**
     * Acquires in exclusive uninterruptible mode for thread already in
     * queue. Used by condition wait methods as well as acquire.
     * 对于已经在队列里的线程，在独占非中断模式下加锁。
     * 由条件等待方法使用也能加锁？？？
     *
     * @param node the node
     * @param arg the acquire argument
     * @return {@code true} if interrupted while waiting // 如果在等待时中断，返回true
     */
    final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) { // 如果加锁失败，这个for啥时候会退出呢？？？
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) { // 如果前驱是head，并且尝试获取独占锁成功
                    setHead(node);  // 将当前node转移为head
                    p.next = null; // help GC // 原来head的next指向null，断开head的连接，准备回收原head
                    failed = false; // 加锁失败状态为false（表示获取锁成功）
                    return interrupted; // 获取锁成功了，返回当前线程中断状态
                }
                if (shouldParkAfterFailedAcquire(p, node) && // 判断在获取锁失败后是否需要park
                    parkAndCheckInterrupt())                 // 如果需要park，进行park并且检查中断状态（如果线程为中断状态，返回true）（因为park能响应中断，中断时会退出park）
                    interrupted = true;                      // 能进到这里说明已经设置了park（阻塞），并且在park等待时发生了中断，当前线程中断状态为true。（这里true也不会直接抛出异常，而是继续去尝试获取锁）
            }
```

```
        } finally {
            if (failed)
                cancelAcquire(node); // 啥时候会走到这里呢？？？
        }
    }

    /**
     * Acquires in exclusive interruptible mode.
     * 在独占可中断模式下加锁
     * @param arg the acquire argument
     */
    private void doAcquireInterruptibly(int arg)
        throws InterruptedException {
        final Node node = addWaiter(Node.EXCLUSIVE); // 返回用当前线程封装的node
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) { // tryAcquire()方法需要自己实现，来决定如何实现尝试获取锁的语义，在AQS里没有阻塞/非阻塞的概念
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    throw new InterruptedException(); // 如果检查到线程中断，抛出异常
            }
        } finally {
            if (failed)
                cancelAcquire(node); // 啥时候会走到这里呢？？？
        }
    }

    /**
     * Acquires in exclusive timed mode.
     * 在独占限时模式下加锁
     *
     * @param arg the acquire argument
     * @param nanosTimeout max wait time // 最大等待时间
     * @return {@code true} if acquired // 成功加锁返回true
     */
    private boolean doAcquireNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (nanosTimeout <= 0L)
            return false; // 对于等待时间<=0的，直接返回false
        final long deadline = System.nanoTime() + nanosTimeout; // 生成截止时间
        final Node node = addWaiter(Node.EXCLUSIVE);
```

```
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) {
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return true;
                }
                nanosTimeout = deadline - System.nanoTime(); // 检查剩余的等待时间
                if (nanosTimeout <= 0L)
                    return false; // 等待时间不够，返回false
                if (shouldParkAfterFailedAcquire(p, node) &&
                    nanosTimeout > spinForTimeoutThreshold) // 如果等待时间（相当于预估的还需要等待时间）> 自旋的时间阈值，就进入park（如果预估时间小于自旋的阈值，可以通过自旋继续等待）。
                    // 这里表明，如果给一个较小的等待时间，就可以不断的通过自旋来加锁（当然也会因为自旋加锁失败，需要不断调用加锁的消耗）
                    LockSupport.parkNanos(this, nanosTimeout);
                if (Thread.interrupted())
                    throw new InterruptedException();
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }

    /**
     * Acquires in shared uninterruptible mode.
     * 在共享非中断模式下加锁
     * @param arg the acquire argument
     */
    private void doAcquireShared(int arg) {
        final Node node = addWaiter(Node.SHARED);
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {
                final Node p = node.predecessor();
                if (p == head) {
                    int r = tryAcquireShared(arg); // 需要自己实现的5大方法之一，在共享模式下尝试加锁。负值表示失败，0表示当前成功，但后继的线程们加锁可能会失败，正值表示当前成功，后继的线程们加锁也可能成功
                    if (r >= 0) {
                        setHeadAndPropagate(node, r); // 将当前节点设置为头结点（表示当前节点已成功获取到锁），如果其他后继节点也能获取到锁（毕竟是个共享锁），也会被从park唤醒
                        p.next = null; // help GC
```

```
                        if (interrupted)
                            selfInterrupt(); // 如果检测到线程中断，调用中断方法（只是写了个中断标记，没有抛出异常）
                        failed = false;
                        return;
                    }
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt()) // 检测线程是否被中断
                    interrupted = true; // 如果判断在加锁失败时需要阻塞（park），并且阻塞后检测到线程被中断，更新当前中断标记为true
            }
        } finally { // 不知道什么时候会走到这里，可能是中断的时候？？？
            if (failed)
                cancelAcquire(node);
        }
    }

    /**
     * Acquires in shared interruptible mode.
     * 在共享可中断模式下加锁
     * @param arg the acquire argument
     */
    private void doAcquireSharedInterruptibly(int arg)
        throws InterruptedException {
        final Node node = addWaiter(Node.SHARED);
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head) {
                    int r = tryAcquireShared(arg);
                    if (r >= 0) {
                        setHeadAndPropagate(node, r);
                        p.next = null; // help GC
                        failed = false;
                        return;
                    }
                }
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    throw new InterruptedException(); // 与上面不同的就在这里，如果检测到了中断，直接抛出中断异常
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

```
/**
 * Acquires in shared timed mode.
 * 在共享限时模式下加锁
 *
 * @param arg the acquire argument
 * @param nanosTimeout max wait time
 * @return {@code true} if acquired
 */
private boolean doAcquireSharedNanos(int arg, long nanosTimeout)
        throws InterruptedException {
    if (nanosTimeout <= 0L)
        return false;
    final long deadline = System.nanoTime() + nanosTimeout;
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return true;
                }
            }
            nanosTimeout = deadline - System.nanoTime();
            if (nanosTimeout <= 0L)
                return false;
            if (shouldParkAfterFailedAcquire(p, node) &&
                nanosTimeout > spinForTimeoutThreshold)
                LockSupport.parkNanos(this, nanosTimeout);
            if (Thread.interrupted())
                throw new InterruptedException(); // 同样的，也会抛出中断异常
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}

// Main exported methods
// 主要对外方法（上面那些都是private的方法）

// 下面这5个protected方法需要子类来实现
/**
 * Attempts to acquire in exclusive mode. This method should query
```

```
     * if the state of the object permits it to be acquired in the
     * exclusive mode, and if so to acquire it.
     * 尝试以独占模式加锁。这个方法应当检查对象的状态是否允许在独占模式下加锁，如果允许则加锁
     *
     * <p>This method is always invoked by the thread performing
     * acquire.  If this method reports failure, the acquire method
     * may queue the thread, if it is not already queued, until it is
     * signalled by a release from some other thread. This can be used
     * to implement method {@link Lock#tryLock()}.
     * 这个方法总是被执行加锁（acquire）的线程调用。
     * 如果此方法报告失败，这个acquire方法可能会将线程入队（如果该线程还没有入队），直到它被其他线程发出release（释放）信号。
     * 可以通过tryLock()方法实现。
     *
     * <p>The default
     * implementation throws {@link UnsupportedOperationException}.
     *
     * @param arg the acquire argument. This value is always the one
     *        passed to an acquire method, or is the value saved on entry
     *        to a condition wait.  The value is otherwise uninterpreted
     *        and can represent anything you like.
     *        加锁参数。该值始终是传递给acquire方法的值，或者是进入条件等待时保存的值。
     *        该值是未经解释的，可以表示你喜欢的任何内容
     * @return {@code true} if successful. Upon success, this object has
     *         been acquired.
     * @throws IllegalMonitorStateException if acquiring would place this
     *         synchronizer in an illegal state. This exception must be
     *         thrown in a consistent fashion for synchronization to work
     *         correctly.
     *         如果加锁会导致同步器进入非法状态则会抛出IllegalMonitorStateException。
     *         必须以一致的方式抛出此异常，同步器才能正常工作（也是个runtimeException）
     * @throws UnsupportedOperationException if exclusive mode is not supported // 如果不支持独占模式，抛出UnsupportedOperationException（这是个runtimeException）
     */
    protected boolean tryAcquire(int arg) {
        throw new UnsupportedOperationException();
    }

    /**
     * Attempts to set the state to reflect a release in exclusive
     * mode.
     * 尝试设置状态来反映独占模式下的解锁/释放（release）
     *
     * <p>This method is always invoked by the thread performing release.
     * 该方法总是被执行解锁（release）的线程调用
     *
     * <p>The default implementation throws
     * {@link UnsupportedOperationException}.
     *
```

```
     * @param arg the release argument. This value is always the one
     *        passed to a release method, or the current state value upon
     *        entry to a condition wait.  The value is otherwise
     *        uninterpreted and can represent anything you like.
     * @return {@code true} if this object is now in a fully released
     *         state, so that any waiting threads may attempt to acquire;
     *         and {@code false} otherwise.
     *         true：表示此对象处于完全释放状态，任何等待线程都可以尝试获取
     * @throws IllegalMonitorStateException if releasing would place this
     *         synchronizer in an illegal state. This exception must be
     *         thrown in a consistent fashion for synchronization to work
     *         correctly.
     * @throws UnsupportedOperationException if exclusive mode is not supported
     */
    protected boolean tryRelease(int arg) {
        throw new UnsupportedOperationException();
    }

    /**
     * Attempts to acquire in shared mode. This method should query if
     * the state of the object permits it to be acquired in the shared
     * mode, and if so to acquire it.
     * 尝试在共享模式下加锁。这个方法应当检查对象的状态是否允许在共享模式下加锁，如果允许则加锁
     *
     * <p>This method is always invoked by the thread performing
     * acquire.  If this method reports failure, the acquire method
     * may queue the thread, if it is not already queued, until it is
     * signalled by a release from some other thread.
     * 这个方法总是被想加锁的线程调用。
     * 如果这个方法报告失败，这个方法可能会将还没在等待队列里的线程入队，直到其他线程释放锁时来
唤醒它。
     *
     * <p>The default implementation throws {@link
     * UnsupportedOperationException}.
     *
     * @param arg the acquire argument. This value is always the one
     *        passed to an acquire method, or is the value saved on entry
     *        to a condition wait.  The value is otherwise uninterpreted
     *        and can represent anything you like.
     * @return a negative value on failure; zero if acquisition in shared
     *         mode succeeded but no subsequent shared-mode acquire can
     *         succeed; and a positive value if acquisition in shared
     *         mode succeeded and subsequent shared-mode acquires might
     *         also succeed, in which case a subsequent waiting thread
     *         must check availability. (Support for three different
     *         return values enables this method to be used in contexts
     *         where acquires only sometimes act exclusively.)  Upon
     *         success, this object has been acquired.
     *         负值表示失败；
```

```
     *          0表示在共享模式下加锁成功但是在随后的共享模式加锁中没有成功；（就是当前线程能够获取到共享锁，没有剩余的共享锁可以被获取）
     *          正值表示在共享模式下加锁成功但是在随后的共享模式加锁中也可能成功（就是当前线程能够获取到共享锁，还有剩余的共享锁可以被获取），在这种情况下，后续等待线程必须检查可用性。
     *          （支持三种不同的返回值，能够保证这个方法能够在仅执行独占行为的上下文中使用）
     *          成功值表示该对象加锁成功。
     *
     * @throws IllegalMonitorStateException if acquiring would place this
     *         synchronizer in an illegal state. This exception must be
     *         thrown in a consistent fashion for synchronization to work
     *         correctly.
     * @throws UnsupportedOperationException if shared mode is not supported
     */
    protected int tryAcquireShared(int arg) {
        throw new UnsupportedOperationException();
    }

    /**
     * Attempts to set the state to reflect a release in shared mode.
     * 尝试设置状态来反映在共享模式下解锁/释放（release）
     *
     * <p>This method is always invoked by the thread performing release.
     *
     * <p>The default implementation throws
     * {@link UnsupportedOperationException}.
     *
     * @param arg the release argument. This value is always the one
     *        passed to a release method, or the current state value upon
     *        entry to a condition wait.  The value is otherwise
     *        uninterpreted and can represent anything you like.
     * @return {@code true} if this release of shared mode may permit a
     *         waiting acquire (shared or exclusive) to succeed; and
     *         {@code false} otherwise
     *         true：表示共享模式下的release可以允许等待获取（共享/独占）操作成功
     * @throws IllegalMonitorStateException if releasing would place this
     *         synchronizer in an illegal state. This exception must be
     *         thrown in a consistent fashion for synchronization to work
     *         correctly.
     * @throws UnsupportedOperationException if shared mode is not supported
     */
    protected boolean tryReleaseShared(int arg) {
        throw new UnsupportedOperationException();
    }

    /**
     * Returns {@code true} if synchronization is held exclusively with
     * respect to the current (calling) thread.  This method is invoked
     * upon each call to a non-waiting {@link ConditionObject} method.
     * (Waiting methods instead invoke {@link #release}.)
```

```
     * 如果当前（调用）线程持有独占的同步锁，将返回true。
     * 这个方法被每个非等待的ConditionObject方法调用。
     * （等待方法替换成调用release）
     *
     * <p>The default implementation throws {@link
     * UnsupportedOperationException}. This method is invoked
     * internally only within {@link ConditionObject} methods, so need
     * not be defined if conditions are not used.
     * 这个方法仅在ConditionObject的方法内部调用，如果不使用Condition就不用定义这个方法。
     *
     * @return {@code true} if synchronization is held exclusively;
     *         {@code false} otherwise
     * @throws UnsupportedOperationException if conditions are not supported
     */
    protected boolean isHeldExclusively() {
        throw new UnsupportedOperationException();
    }

        // 下面是完全对外的public方法
    /**
     * Acquires in exclusive mode, ignoring interrupts.  Implemented
     * by invoking at least once {@link #tryAcquire},
     * returning on success.  Otherwise the thread is queued, possibly
     * repeatedly blocking and unblocking, invoking {@link
     * #tryAcquire} until success.  This method can be used
     * to implement method {@link Lock#lock}.
     * 在独占模式下获取，忽略中断。通过至少调用一次tryAcquire来实现，在成功时返回。
     * 否则线程会排队，可能会反复阻塞与解除阻塞，调用tryAcquire直到成功。
     * 这个方法可以用来实现lock方法。
     * （拿不到锁就入队排队，等不到就阻塞等待唤醒）
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
     */
    public final void acquire(int arg) {
        if (!tryAcquire(arg) && // 尝试获取失败
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            // addWaiter(Node.EXCLUSIVE), arg)将当前线程按独占模式创建node，加入到队列中
            // acquireQueued如果当前node的前驱是head，那么尝试获取，如果不是，分析是否阻塞等
待与阻塞唤醒后检测中断信号
            selfInterrupt();
    }

    /**
     * Acquires in exclusive mode, aborting if interrupted.
     * Implemented by first checking interrupt status, then invoking
     * at least once {@link #tryAcquire}, returning on
     * success.  Otherwise the thread is queued, possibly repeatedly
```

```
     * blocking and unblocking, invoking {@link #tryAcquire}
     * until success or the thread is interrupted.  This method can be
     * used to implement method {@link Lock#lockInterruptibly}.
     * 在独占模式下获取，如果中断了就终止。
     * 通过首先检查中断状态，然后至少调用一次tryAcquire实现，成功时返回。
     * 否则线程入队，可能反复阻塞与取消阻塞，调用tryAcquire直到成功或者线程被中断。
     * 这个方法可以用来实现lockInterruptibly方法。
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
     * @throws InterruptedException if the current thread is interrupted
     */
    public final void acquireInterruptibly(int arg)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        if (!tryAcquire(arg))
            doAcquireInterruptibly(arg); // 如果获取失败，调用doAcquireInterruptibly方法，跟acquireQueued方法类似，只是会在检测到线程中断后，直接抛出中断异常，而不是继续尝试获取锁
    }

    /**
     * Attempts to acquire in exclusive mode, aborting if interrupted,
     * and failing if the given timeout elapses.  Implemented by first
     * checking interrupt status, then invoking at least once {@link
     * #tryAcquire}, returning on success.  Otherwise, the thread is
     * queued, possibly repeatedly blocking and unblocking, invoking
     * {@link #tryAcquire} until success or the thread is interrupted
     * or the timeout elapses.  This method can be used to implement
     * method {@link Lock#tryLock(long, TimeUnit)}.
     * 在独占模式下获取，如果中断或者超时就终止或者失败。
     * 通过首先检查中断状态，然后至少一次调用tryAcquire来实现，成功时返回。
     * 否则当前线程入队，可能反复阻塞与取消阻塞，调用tryAcquire直到获取成功，或者被中断，或者超时。
     * 这个方法可以用来实现tryLock(long, TimeUnit)方法
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquire} but is otherwise uninterpreted and
     *        can represent anything you like.
     * @param nanosTimeout the maximum number of nanoseconds to wait
     * @return {@code true} if acquired; {@code false} if timed out
     * @throws InterruptedException if the current thread is interrupted
     */
    public final boolean tryAcquireNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        return tryAcquire(arg) ||
```

```
            doAcquireNanos(arg, nanosTimeout);
    }

    /**
     * Releases in exclusive mode.  Implemented by unblocking one or
     * more threads if {@link #tryRelease} returns true.
     * This method can be used to implement method {@link Lock#unlock}.
     * 在独占模式下释放。如果tryRelease返回true，通过解除阻塞一个或多个线程来实现。
     * 这个方法可以用来实现unlock方法
     *
     * @param arg the release argument.  This value is conveyed to
     *        {@link #tryRelease} but is otherwise uninterpreted and
     *        can represent anything you like.
     * @return the value returned from {@link #tryRelease}
     */
    public final boolean release(int arg) {
        if (tryRelease(arg)) {
            Node h = head;
            if (h != null && h.waitStatus != 0) // 如果有头结点，并且ws不是默认值0
                unparkSuccessor(h); // 唤醒一个后继节点，这个方法会把ws置为0，表示正在操作
? ? ?
            return true;
        }
        return false;
    }

    /**
     * Acquires in shared mode, ignoring interrupts.  Implemented by
     * first invoking at least once {@link #tryAcquireShared},
     * returning on success.  Otherwise the thread is queued, possibly
     * repeatedly blocking and unblocking, invoking {@link
     * #tryAcquireShared} until success.
     * 共享模式下获取，忽略中断。
     * 通过首先调用至少一次tryAcquireShared方法来实现，成功时返回。
     * 否则该线程入队，可能反复阻塞或者解除阻塞，调用tryAcquireShared直到成功。
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquireShared} but is otherwise uninterpreted
     *        and can represent anything you like.
     */
    public final void acquireShared(int arg) {
        if (tryAcquireShared(arg) < 0) // 尝试获取失败（这是个需要自己实现的方法）
            doAcquireShared(arg); // 将该线程封装成共享模式的node然后入队，循环调用tryAcqu
ireShared来获取，如果tryAcquireShared返回值>=0，会调用setHeadAndPropagate方法，先把当前nod
e设置为head，然后尝试释放后继的node；如果tryAcquireShared返回<0，会判断是否需要park
    }

    /**
     * Acquires in shared mode, aborting if interrupted.  Implemented
```

```
     * by first checking interrupt status, then invoking at least once
     * {@link #tryAcquireShared}, returning on success.  Otherwise the
     * thread is queued, possibly repeatedly blocking and unblocking,
     * invoking {@link #tryAcquireShared} until success or the thread
     * is interrupted.
     * 在共享模式下获取，在中断时终止。
     * 通过首先检查中断状态，然后至少调用一次tryAcquireShared方法实现，成功时返回。
     * 否则当前线程入队，可能反复阻塞与解除阻塞，调用tryAcquireShared方法，直到成功或者线程中
断。
     *
     * @param arg the acquire argument.
     * This value is conveyed to {@link #tryAcquireShared} but is
     * otherwise uninterpreted and can represent anything
     * you like.
     * @throws InterruptedException if the current thread is interrupted
     */
    public final void acquireSharedInterruptibly(int arg)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        if (tryAcquireShared(arg) < 0)
            doAcquireSharedInterruptibly(arg); // 在中断时抛出中断异常
    }

    /**
     * Attempts to acquire in shared mode, aborting if interrupted, and
     * failing if the given timeout elapses.  Implemented by first
     * checking interrupt status, then invoking at least once {@link
     * #tryAcquireShared}, returning on success.  Otherwise, the
     * thread is queued, possibly repeatedly blocking and unblocking,
     * invoking {@link #tryAcquireShared} until success or the thread
     * is interrupted or the timeout elapses.
     * 尝试在共享模式下获取，如果中断则终止，如果超时则返回失败。
     * 通过首先检查中断状态，然后至少调用一次tryAcquireShared方法实现，成功时返回。
     * 否则，当前线程入队，可能反复多次阻塞与解除阻塞，调用tryAcquireShared方法直到成功，或者
线程中断，或者超时
     *
     * @param arg the acquire argument.  This value is conveyed to
     *        {@link #tryAcquireShared} but is otherwise uninterpreted
     *        and can represent anything you like.
     * @param nanosTimeout the maximum number of nanoseconds to wait
     * @return {@code true} if acquired; {@code false} if timed out
     * @throws InterruptedException if the current thread is interrupted
     */
    public final boolean tryAcquireSharedNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        return tryAcquireShared(arg) >= 0 ||
```

```
            doAcquireSharedNanos(arg, nanosTimeout); // 正常逻辑操作，没啥说的
    }

    /**
     * Releases in shared mode.  Implemented by unblocking one or more
     * threads if {@link #tryReleaseShared} returns true.
     * 共享模式下释放。
     * 如果tryReleaseShared返回true，通过解除阻塞一个或多个线程来实现。
     *
     * @param arg the release argument.  This value is conveyed to
     *        {@link #tryReleaseShared} but is otherwise uninterpreted
     *        and can represent anything you like.
     * @return the value returned from {@link #tryReleaseShared}
     */
    public final boolean releaseShared(int arg) {
        if (tryReleaseShared(arg)) {
            doReleaseShared();
            return true;
        }
        return false;
    }

    // Queue inspection methods
    // 队列检查方法

    /**
     * Queries whether any threads are waiting to acquire. Note that
     * because cancellations due to interrupts and timeouts may occur
     * at any time, a {@code true} return does not guarantee that any
     * other thread will ever acquire.
     * 查询是否有线程正在等待获取。
     * 注意：由于可能在任意时间发生中断或者超时，会导致线程取消，所以返回true也不能保证有线程永远在等待获取。
     *
     * <p>In this implementation, this operation returns in
     * constant time.
     *
     * @return {@code true} if there may be other threads waiting to acquire
     */
    public final boolean hasQueuedThreads() {
        return head != tail;
    }

    /**
     * Queries whether any threads have ever contended to acquire this
     * synchronizer; that is if an acquire method has ever blocked.
     * 查询是否有线程竞争过获取这个同步器；
     * 也就是说，曾经有线程调用acquire方法阻塞过。
     *
```

```
     * <p>In this implementation, this operation returns in
     * constant time.
     *
     * @return {@code true} if there has ever been contention
     */
    public final boolean hasContended() {
        return head != null;
    }

    /**
     * Returns the first (longest-waiting) thread in the queue, or
     * {@code null} if no threads are currently queued.
     * 返回队列里的第一个线程（等待时间最长的），如果没有线程在当前队列，返回null
     *
     * <p>In this implementation, this operation normally returns in
     * constant time, but may iterate upon contention if other threads are
     * concurrently modifying the queue.
     *
     * @return the first (longest-waiting) thread in the queue, or
     *         {@code null} if no threads are currently queued
     */
    public final Thread getFirstQueuedThread() {
        // handle only fast path, else relay
        return (head == tail) ? null : fullGetFirstQueuedThread();
    }

    /**
     * Version of getFirstQueuedThread called when fastpath fails
     * 快速路径失败时调用的版本
     */
    private Thread fullGetFirstQueuedThread() {
        /*
         * The first node is normally head.next. Try to get its
         * thread field, ensuring consistent reads: If thread
         * field is nulled out or s.prev is no longer head, then
         * some other thread(s) concurrently performed setHead in
         * between some of our reads. We try this twice before
         * resorting to traversal.
         * 确保读一致性，如果线程字段被置null，或者s.prev不再是head，然后其他线程并发的在我们
读中间setHead。
         * 尝试两次
         */
        Node h, s;
        Thread st;
        if (((h = head) != null && (s = h.next) != null &&
             s.prev == head && (st = s.thread) != null) ||
            ((h = head) != null && (s = h.next) != null &&
             s.prev == head && (st = s.thread) != null))
            return st;
```

```
        /*
         * Head's next field might not have been set yet, or may have
         * been unset after setHead. So we must check to see if tail
         * is actually first node. If not, we continue on, safely
         * traversing from tail back to head to find first,
         * guaranteeing termination.
         * head的next可能还没有设置，或者可能在setHead之后未设置。所以应该检查tail是否实际上是第一个节点。
         * 如果不是，继续安全的从tail向head查找第一个，保证终止。
         */

        Node t = tail;
        Thread firstThread = null;
        while (t != null && t != head) {
            Thread tt = t.thread;
            if (tt != null)
                firstThread = tt;
            t = t.prev;
        }
        return firstThread;
    }

    /**
     * Returns true if the given thread is currently queued.
     * 如果当前线程在排队，返回true
     *
     * <p>This implementation traverses the queue to determine
     * presence of the given thread.
     *
     * @param thread the thread
     * @return {@code true} if the given thread is on the queue
     * @throws NullPointerException if the thread is null
     */
    public final boolean isQueued(Thread thread) {
        if (thread == null)
            throw new NullPointerException();
        for (Node p = tail; p != null; p = p.prev)
            if (p.thread == thread)
                return true;
        return false;
    }

    /**
     * Returns {@code true} if the apparent first queued thread, if one
     * exists, is waiting in exclusive mode.  If this method returns
     * {@code true}, and the current thread is attempting to acquire in
     * shared mode (that is, this method is invoked from {@link
     * #tryAcquireShared}) then it is guaranteed that the current thread
```

```
     * is not the first queued thread.  Used only as a heuristic in
     * ReentrantReadWriteLock.
     * 如果明显的第一个线程（如果有的话）在独占模式下等待，返回true。
     * 如果该方法返回true，并且当前线程试图在共享模式下获取（这意味着，这个方式是通过tryAcquireShared方法调用的），则可以保证当前线程不是第一个排队的线程。
     * 仅用于ReentrantReadWriteLcok的启发式方法
     *
     */
    final boolean apparentlyFirstQueuedIsExclusive() {
        Node h, s;
        return (h = head) != null &&
            (s = h.next)  != null &&
            !s.isShared()         &&
            s.thread != null;
    }

    /**
     * Queries whether any threads have been waiting to acquire longer
     * than the current thread.
     * 查询是否有比当前线程等待时间更久的线程。
     *
     * <p>An invocation of this method is equivalent to (but may be
     * more efficient than):
     * 调用此方法等效于（但是可能更高效）：
     *  <pre> {@code
     * getFirstQueuedThread() != Thread.currentThread() &&
     * hasQueuedThreads()}</pre>
     * 当前线程不是队列里的第一个，并且队列里有线程
     *
     * <p>Note that because cancellations due to interrupts and
     * timeouts may occur at any time, a {@code true} return does not
     * guarantee that some other thread will acquire before the current
     * thread.  Likewise, it is possible for another thread to win a
     * race to enqueue after this method has returned {@code false},
     * due to the queue being empty.
     * 这个方法不能保证这个线程前面一定有老线程，或者这个线程一定是等待时间最长的线程
     * 原因1、可能在这个方法返回true后，前面的线程由于中断或者超时导致退出了等待
     * 原因2、可能在这个方法返回false后，在队列为空时，有新的线程在与该线程入队竞争时获胜，比该线程更早入队
     *
     * <p>This method is designed to be used by a fair synchronizer to
     * avoid <a href="AbstractQueuedSynchronizer#barging">barging</a>.
     * Such a synchronizer's {@link #tryAcquire} method should return
     * {@code false}, and its {@link #tryAcquireShared} method should
     * return a negative value, if this method returns {@code true}
     * (unless this is a reentrant acquire).  For example, the {@code
     * tryAcquire} method for a fair, reentrant, exclusive mode
     * synchronizer might look like this:
     * 该方法是为了公平同步器设计的，避免插队情况。
```

```
     * 如果这个方法返回true，这种同步器的tryAcquire方法应该返回false，它的tryAcquireShared
方法应该返回负值，除非这是一个可重入的获取过程。
     * 例如：公平、可重入、独占模式下的同步器tryAcquire方法可能如下所示：
     *
     *  <pre> {@code
     * protected boolean tryAcquire(int arg) {
     *   if (isHeldExclusively()) { // 先看是不是当前线程独占
     *     // A reentrant acquire; increment hold count
     *     return true;
     *   } else if (hasQueuedPredecessors()) { // 再看前面有没有排队的（现在线程可能不在队
列，也可能在队列里，不影响判断）
     *     return false;
     *   } else { // 都没有就准备竞争
     *     // try to acquire normally
     *   }
     * }}</pre>
     *
     * @return {@code true} if there is a queued thread preceding the
     *         current thread, and {@code false} if the current thread
     *         is at the head of the queue or the queue is empty
     * @since 1.7
     */
    public final boolean hasQueuedPredecessors() {
        // The correctness of this depends on head being initialized
        // before tail and on head.next being accurate if the current
        // thread is first in queue.
        // 正确性取决于head在tail之前被初始化，并且如果当前线程是在队列的第一个，那么head.nex
t是准确的
        Node t = tail; // Read fields in reverse initialization order
        Node h = head;
        Node s;
        // 方法返回true表示在当前线程之前还有等待的线程，false表示没有
        // head != tail排除了两种情况：
        // 1、head = tail = null（此时队列还未有阶段入过队）
        // 2、head = tail = node（此时队列里曾经有入过队的，但都已出队）
        // (s = h.next) == null（可能存在一种情况，设置了head，但是next还没来得及设置为（除
了head之外）第一个等待的线程节点，这种情况下，next不准确，因为不知道next是当前线程还是其他线程
，为了保险起见，返回true）
        // s.thread != Thread.currentThread() （这个简单了，判断第一个等待节点是不是该线程
的节点，不是的话返回true）
        return h != t &&
            ((s = h.next) == null || s.thread != Thread.currentThread());
    }


    // Instrumentation and monitoring methods
    // 仪表盘和监控方法

    /**
```

```
     * Returns an estimate of the number of threads waiting to
     * acquire.  The value is only an estimate because the number of
     * threads may change dynamically while this method traverses
     * internal data structures.  This method is designed for use in
     * monitoring system state, not for synchronization
     * control.
     * 返回预估的等待获取的线程数。
     * 这个返回值只是个预估值，因为在方法遍历内部数据结构时，线程是可能动态变化的。
     * 该方法是为了监控系统状态设计的，不是为了同步。
     *
     * @return the estimated number of threads waiting to acquire
     */
    public final int getQueueLength() {
        int n = 0;
        for (Node p = tail; p != null; p = p.prev) { // 再次理解一下，说是queue，其实没有维护一个队列长度的属性
            if (p.thread != null)
                ++n;
        }
        return n;
    }

    /**
     * Returns a collection containing threads that may be waiting to
     * acquire.  Because the actual set of threads may change
     * dynamically while constructing this result, the returned
     * collection is only a best-effort estimate.  The elements of the
     * returned collection are in no particular order.  This method is
     * designed to facilitate construction of subclasses that provide
     * more extensive monitoring facilities.
     * 返回一个包含正在等待获取的线程集合。
     * 因为在构造该集合结果时，实际线程可能在动态改变，所以这个返回的集合只是一个尽力的预估。
     * 返回集合里的线程没有特定顺序。
     * 该方法是为了促进子类构建而设计，以提高更广泛的监控设备
     *
     * @return the collection of threads
     */
    public final Collection<Thread> getQueuedThreads() {
        ArrayList<Thread> list = new ArrayList<Thread>();
        for (Node p = tail; p != null; p = p.prev) {
            Thread t = p.thread;
            if (t != null)
                list.add(t);
        }
        return list;
    }

    /**
     * Returns a collection containing threads that may be waiting to
```

```
     * acquire in exclusive mode. This has the same properties
     * as {@link #getQueuedThreads} except that it only returns
     * those threads waiting due to an exclusive acquire.
     * 返回一个包含可能正在独占模式下等待的线程集合。
     * 跟上面的getQueueThreads方法类似，除了只返回独占获取的等待线程
     *
     * @return the collection of threads
     */
    public final Collection<Thread> getExclusiveQueuedThreads() {
        ArrayList<Thread> list = new ArrayList<Thread>();
        for (Node p = tail; p != null; p = p.prev) {
            if (!p.isShared()) { // 去掉共享模式下的等待线程（就是说一个queue里面可能既有独占，又有共享的等待线程）
                Thread t = p.thread;
                if (t != null)
                    list.add(t);
            }
        }
        return list;
    }

    /**
     * Returns a collection containing threads that may be waiting to
     * acquire in shared mode. This has the same properties
     * as {@link #getQueuedThreads} except that it only returns
     * those threads waiting due to a shared acquire.
     * 返回一个包含在共享模式下等待的线程集合。
     *
     * @return the collection of threads
     */
    public final Collection<Thread> getSharedQueuedThreads() {
        ArrayList<Thread> list = new ArrayList<Thread>();
        for (Node p = tail; p != null; p = p.prev) {
            if (p.isShared()) { // 只要共享模式下的
                Thread t = p.thread;
                if (t != null)
                    list.add(t);
            }
        }
        return list;
    }

    /**
     * Returns a string identifying this synchronizer, as well as its state.
     * The state, in brackets, includes the String {@code "State ="}
     * followed by the current value of {@link #getState}, and either
     * {@code "nonempty"} or {@code "empty"} depending on whether the
     * queue is empty.
     * 返回标识此同步器和state的字符串（state由子类实现，在AQS中没使用）
```

```
     * （bracket 括号）
     * @return a string identifying this synchronizer, as well as its state
     */
    public String toString() {
        int s = getState();
        String q  = hasQueuedThreads() ? "non" : "";
        return super.toString() +
            "[State = " + s + ", " + q + "empty queue]";
    }


    // Internal support methods for Conditions
    // 内部支持Condition的方法

    /**
     * Returns true if a node, always one that was initially placed on
     * a condition queue, is now waiting to reacquire on sync queue.
     * 如果一个节点始终是最初放在条件队列中的节点，现在正在等待重新获取sync队列，则返回true
     *
     * @param node the node
     * @return true if is reacquiring
     */
    final boolean isOnSyncQueue(Node node) {
          // 如果节点的等待状态是CONDITION，说明在condition队列中（不在AQS主队列）；
          // 如果prev是null，并且是AQS，也是已获取到锁的head节点，也不在AQS主队列中等待
        if (node.waitStatus == Node.CONDITION || node.prev == null)
            return false;
        if (node.next != null) // If has successor, it must be on queue // 意思是只有AQS
主队列才有next跟prev关系，如果next不为空，一定在AQS sync主队列里
            return true;
        /*
         * node.prev can be non-null, but not yet on queue because
         * the CAS to place it on queue can fail. So we have to
         * traverse from tail to make sure it actually made it.  It
         * will always be near the tail in calls to this method, and
         * unless the CAS failed (which is unlikely), it will be
         * there, so we hardly ever traverse much.
         * node.prev如果不为null，不能保证一定在队列里，因为通过CAS操作入队时会失败。（可以看
addWaiter方法，先设置node.prev=tail，然后去做的CAS，只有CAS成功了才算成功入队）
         * 所以从tail向前遍历，确保它确实入队了。
         * 除非CAS失败（基本不太可能），否则在调用这个方法时它总是靠近尾部，所以我们不会遍历太
多。
         */
        return findNodeFromTail(node);
    }

    /**
     * Returns true if node is on sync queue by searching backwards from tail.
     * Called only when needed by isOnSyncQueue.
```

```
     * 从sync队列的tail向前遍历，如果找到该节点，就返回true
     *
     * @return true if present
     */
    private boolean findNodeFromTail(Node node) {
        Node t = tail;
        for (;;) {
            if (t == node)
                return true;
            if (t == null) // 要么tail为null，要么找到了head的prev，也是null，可以看setHead

                return false;
            t = t.prev;
        }
    }

    /**
     * Transfers a node from a condition queue onto sync queue.
     * Returns true if successful.
     * 将node从condition队列转移到sync队列。
     * 成功转移返回true
     *
     * @param node the node
     * @return true if successfully transferred (else the node was
     * cancelled before signal)
     */
    final boolean transferForSignal(Node node) {
        /*
         * If cannot change waitStatus, the node has been cancelled.
         * 如果不能更改ws，说明这个node已经被取消了。（在condition队列上的node一定是CONDITIO
N状态，如果node存活，一定可以CAS改变ws）
         */
        if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
            return false;

        /*
         * Splice onto queue and try to set waitStatus of predecessor to
         * indicate that thread is (probably) waiting. If cancelled or
         * attempt to set waitStatus fails, wake up to resync (in which
         * case the waitStatus can be transiently and harmlessly wrong).
         * 拼接到队列，尝试设置前驱的ws值，表名当前线程在（可能）等待。
         * 如果前驱被取消或者试图设置前驱的ws失败，则唤醒以重新同步（在这种情况下，waitStatus
不匹配可能是暂时且无害的错误）
         *
         */
        Node p = enq(node); // 加入到sync队列中，并返回前驱节点
        int ws = p.waitStatus;
        if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL)) // 如果前驱的ws>0（
说明前驱节点被cancel了），或者设置前驱的ws为SIGNAL失败，那么直接唤醒当前这个node去竞争资源（竞
```

```
争不到咋办？？？由具体的实现去做，比如重新入队）
            LockSupport.unpark(node.thread);
        return true;
    }

    /**
     * Transfers node, if necessary, to sync queue after a cancelled wait.
     * Returns true if thread was cancelled before being signalled.
     * 如果必要的话，在取消等待后，转移node到sync队列。（这个取消等待，不是在condition队列wait时被CANCEL了，而是wait的条件满足，或者等待时间超时，被唤醒了，才调用的这个方法）
     * 如果线程在收到信号前取消等待了，返回true
     *
     * @param node the node
     * @return true if cancelled before the node was signalled
     */
    final boolean transferAfterCancelledWait(Node node) {
        if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) { // 如果当前状态为CONDITION，则进行入sync队列操作
            enq(node);
            return true;
        }
        // 不为CONDITION状态的话，有两种情况
        // 1、CANCEL 说明当前节点被取消了，但不知道是在队列里被取消还是没在队列里被取消
        // 2、其他状态 说明当前节点在队列里
        // 上面那句话不对，其实不为CONDITION只有一种情况，那就是该node已经进了sync队列，并且ws发生了变化。在condition队列里ws是不会变的。这句话也不对，condition里面状态可以变。
        /*
         * If we lost out to a signal(), then we can't proceed
         * until it finishes its enq().  Cancelling during an
         * incomplete transfer is both rare and transient, so just
         * spin.
         * 如果我们输给了signal方法，那么在它完成enq()之前不能继续做别的。
         * 在未完成转移时取消，是罕见又短暂的，因此只需要自旋。
         */
        while (!isOnSyncQueue(node))
            Thread.yield();
        return false;
    }

    /**
     * Invokes release with current state value; returns saved state.
     * Cancels node and throws exception on failure.
     * 在当前state值下调用release；
     * 返回保存的state值
     * 在失败时取消节点并抛出异常
     *
     * @param node the condition node for this wait
     * @return previous sync state
     */
```

```
    final int fullyRelease(Node node) {
        boolean failed = true;
        try {
            int savedState = getState();
            if (release(savedState)) { // 用当前state值作为release(arg)的入参，如果自定义的tryRelease返回true，从head开始唤醒后继节点
                failed = false;
                return savedState;
            } else {
                throw new IllegalMonitorStateException();
            }
        } finally {
            if (failed)
                node.waitStatus = Node.CANCELLED; // 如果唤醒失败，就取消当前节点（这是个什么操作？）
        }
    }

    // Instrumentation methods for conditions
    // condition的检测方法

    /**
     * Queries whether the given ConditionObject
     * uses this synchronizer as its lock.
     * 查询给定的ConditionObject是否用该同步器作为它的锁
     *
     * @param condition the condition
     * @return {@code true} if owned
     * @throws NullPointerException if the condition is null
     */
    public final boolean owns(ConditionObject condition) {
        return condition.isOwnedBy(this);
    }

    /**
     * Queries whether any threads are waiting on the given condition
     * associated with this synchronizer. Note that because timeouts
     * and interrupts may occur at any time, a {@code true} return
     * does not guarantee that a future {@code signal} will awaken
     * any threads.  This method is designed primarily for use in
     * monitoring of the system state.
     * 查询是否有线程在给定的condition关联的同步器上等待。
     * 注意，由于超时和中断可能在任意时刻发生，返回结果true不保证未来signal可以唤醒线程。
     * 该方法设计的意图是为了在监控系统中使用。
     *
     * @param condition the condition
     * @return {@code true} if there are any waiting threads
     * @throws IllegalMonitorStateException if exclusive synchronization
     *         is not held
```

```
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this synchronizer
     * @throws NullPointerException if the condition is null
     */
    public final boolean hasWaiters(ConditionObject condition) {
        if (!owns(condition)) // 如果condition没有关联到本AbstractQueuedSynchronizer（sync)
            throw new IllegalArgumentException("Not owner");
        return condition.hasWaiters(); // 如果有在CONDITION上等待条件的，返回true
    }

    /**
     * Returns an estimate of the number of threads waiting on the
     * given condition associated with this synchronizer. Note that
     * because timeouts and interrupts may occur at any time, the
     * estimate serves only as an upper bound on the actual number of
     * waiters.  This method is designed for use in monitoring of the
     * system state, not for synchronization control.
     * 返回预估的在给定的condition关联的同步器上等待的线程数。
     * 注意，由于超时与中断可能在任意时刻发生，预估的服务数是实际服务数的上限。
     * 该方法只是设计用作监控系统的，不是为了同步控制。
     *
     * @param condition the condition
     * @return the estimated number of waiting threads
     * @throws IllegalMonitorStateException if exclusive synchronization
     *         is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this synchronizer
     * @throws NullPointerException if the condition is null
     */
    public final int getWaitQueueLength(ConditionObject condition) {
        if (!owns(condition))
            throw new IllegalArgumentException("Not owner");
        return condition.getWaitQueueLength();
    }

    /**
     * Returns a collection containing those threads that may be
     * waiting on the given condition associated with this
     * synchronizer.  Because the actual set of threads may change
     * dynamically while constructing this result, the returned
     * collection is only a best-effort estimate. The elements of the
     * returned collection are in no particular order.
     * 返回包含可能在给定condition关联的同步器上等待的线程集合。
     * 由于在结构化生成结果过程中，实际上线程是动态变化的，这个返回结果稽核只是一个尽力预估值。
     * 返回的稽核元素没有特定的顺序。
     *
     * @param condition the condition
     * @return the collection of threads
```

```
     * @throws IllegalMonitorStateException if exclusive synchronization
     *         is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this synchronizer
     * @throws NullPointerException if the condition is null
     */
    public final Collection<Thread> getWaitingThreads(ConditionObject condition) {
        if (!owns(condition))
            throw new IllegalArgumentException("Not owner");
        return condition.getWaitingThreads();
    }

    /**
     * Condition implementation for a {@link
     * AbstractQueuedSynchronizer} serving as the basis of a {@link
     * Lock} implementation.
     * 作为AQS服务的条件实现，作为Lock实现的基础。
     *
     * <p>Method documentation for this class describes mechanics,
     * not behavioral specifications from the point of view of Lock
     * and Condition users. Exported versions of this class will in
     * general need to be accompanied by documentation describing
     * condition semantics that rely on those of the associated
     * {@code AbstractQueuedSynchronizer}.
     * 该类的方法说明从使用者的角度描述Lock与Condition的机制，而不是行为规范。
     * 该类的导出版本通常需要跟依赖关联的AQS的条件语义文档一起看。
     *
     * <p>This class is Serializable, but all fields are transient,
     * so deserialized conditions have no waiters.
     * class是可序列化的，不过所有的字段都是transient（暂时的），所以反序列化的condition没有任何waiter。
     *
     */
    public class ConditionObject implements Condition, java.io.Serializable {
        private static final long serialVersionUID = 1173984872572414699L; // 用来验证版本一致性。根据包名，类名，继承关系，非私有的方法和属性，以及参数，返回值等诸多因子计算得出的，极度复杂生成的一个64位的哈希字段。基本上计算出来的这个值是唯一的。默认是1L。
        /** First node of condition queue. */
        private transient Node firstWaiter; // firstWaiter是个Node类型，具有prev、next、thread等属性，还有nextWaiter这种属性
        /** Last node of condition queue. */
        private transient Node lastWaiter; // lastWaiter是个Node类型

        /**
         * Creates a new {@code ConditionObject} instance.
         */
        public ConditionObject() { }

        // Internal methods
```

```
        // 内部方法

        /**
         * Adds a new waiter to wait queue.
         * @return its new wait node
         */
        private Node addConditionWaiter() {
            Node t = lastWaiter;
            // If lastWaiter is cancelled, clean out. // 如果最后一个节点取消了，清理掉。
            if (t != null && t.waitStatus != Node.CONDITION) {
                unlinkCancelledWaiters(); // 从头开始进行完整遍历，留下ws=CONDITION的节点
                t = lastWaiter; // unlink会更新lastWaiter，重新指向lastWaiter
            }
            Node node = new Node(Thread.currentThread(), Node.CONDITION); // 封装当前线
程为Node，默认ws为CONDITION
            if (t == null) // 没有lastWaiter，说明当前condition队列为空，让firstWaiter指
向该节点
                firstWaiter = node;
            else
                t.nextWaiter = node; // 否则让当前lastWaiter的next指向该节点
            lastWaiter = node; // 设置当前节点为lastWaiter
            return node;
        }

        /**
         * Removes and transfers nodes until hit non-cancelled one or
         * null. Split out from signal in part to encourage compilers
         * to inline the case of no waiters.
         * 转移和移除节点，直到命中未取消的节点或者遍历完没有非null节点。
         * 从signal中分离出来，一部分是为了鼓励编译器内联没有waiters的情况。
         *
         * @param first (non-null) the first node on condition queue
         */
        private void doSignal(Node first) {
              // 目标就是将firstWaiter节点转移到sync队列里，然后移除该节点。first为非空的fi
rstWaiter。（如果是null，会导致transferForSignal(first)报错）
            do {
                if ( (firstWaiter = first.nextWaiter) == null) // 1、让firstWaiter指向f
irstWaiter的下一个节点
                    lastWaiter = null;                          // 2、如果下一个节点为空，
说明condition队列里没有等待节点了，lastWaiter也置为空
                first.nextWaiter = null;                        // 3、断开即将移入sync队
列的节点next引用
            } while (!transferForSignal(first) &&               // 4、如果该节点状态非CON
DITION，表示该节点已CANCEL，返回false，表示无法入队，否则将该节点enq入队，如果入队失败，会unp
ark该节点直接竞争锁
                     (first = firstWaiter) != null);            // 5、让first指向firstWa
iter，如果现在第一个节点不为null，并且上一个节点已经被取消了，那么尝试释放下一个节点。
        }
```

```
        /**
         * Removes and transfers all nodes.
         * 转移和移除所有节点
         *
         * @param first (non-null) the first node on condition queue
         */
        private void doSignalAll(Node first) {
            lastWaiter = firstWaiter = null; // 反正都要移除了，啥也不管直接都置为null
            do {
                Node next = first.nextWaiter; // 遍历非null的节点，有一个算一个，都给扔到transferForSignal(first)方法中去转移到sync队列里，然后移除。
                first.nextWaiter = null;
                transferForSignal(first);
                first = next;
            } while (first != null);
        }

        /**
         * Unlinks cancelled waiter nodes from condition queue.
         * Called only while holding lock. This is called when
         * cancellation occurred during condition wait, and upon
         * insertion of a new waiter when lastWaiter is seen to have
         * been cancelled. This method is needed to avoid garbage
         * retention in the absence of signals. So even though it may
         * require a full traversal, it comes into play only when
         * timeouts or cancellations occur in the absence of
         * signals. It traverses all nodes rather than stopping at a
         * particular target to unlink all pointers to garbage nodes
         * without requiring many re-traversals during cancellation
         * storms.
         * 从条件队列中取消连接的cancel waiter节点。
         * 只有在持有锁的时候才会被调用。
         * 当在条件等待时发生取消，在插入新的waiter时发现最后的waiter已经被取消时，会调用到该方法。（发现最后一个被取消时会调用，遍历则是从头开始向后遍历）
         * 需要该方法在没有信号（absence 缺席）的情况下避免垃圾保留。
         * 因此，即使它可能需要全部遍历，它也仅在没有信号的情况下发生超时或者取消时才起作用。
         * 它遍历所有节点而不是在特定节点处停止，以取消所有指向垃圾节点的指针，而不需要在取消风暴中多次重新遍历。
         *
         */
        private void unlinkCancelledWaiters() {
            Node t = firstWaiter; // 从第一个节点开始向后遍历
            Node trail = null; // trail保留最后一个未取消的节点引用（每当发现一个后续未取消节点，这个trail就变为指向该节点）
            while (t != null) {
                Node next = t.nextWaiter;
                if (t.waitStatus != Node.CONDITION) { // 如果当前节点的ws不是CONDITION，说明当前节点不再等待了（就是取消了），需要取消连接
```

```
                    t.nextWaiter = null; // 断开该节点与下一个节点的关联，
                    if (trail == null) // 如果现在剩余节点link为空，说明现在还没有节点留下来

                        firstWaiter = next; // 把第一个留下来的节点作为firstWaiter
                    else
                        trail.nextWaiter = next; // 否则就把当前节点（准备清理的节点）的下一个节点加入到剩余节点link中（通过trail保留截止到目前最后一个节点的引用，使用next来构建link）
                    if (next == null) // 如果没有后续waiter了
                        lastWaiter = trail; // lastWaiter指向剩余节点link的最后一个节点
                }
                else
                    trail = t; // 当前节点不用取消，link没变化，直接让trail指向当前未取消的节点，继续向后遍历
                t = next; // 准备下一个节点的遍历
            }
        }

        // public methods
        // 公共方法

        /**
         * Moves the longest-waiting thread, if one exists, from the
         * wait queue for this condition to the wait queue for the
         * owning lock.
         * 如果有等待线程的话，将等待时间最长的线程从等待condition的队列转移到等待lock的队列
         *
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        public final void signal() {
            if (!isHeldExclusively()) // 首先要当前线程得持有锁（自己实现的方法返回true）
                throw new IllegalMonitorStateException();
            Node first = firstWaiter; // 排在第一个的线程节点，就是等待时间最长的（因为先来先入队先等待）
            if (first != null) // 如果上来就是null，就不转移了
                doSignal(first);
        }

        /**
         * Moves all threads from the wait queue for this condition to
         * the wait queue for the owning lock.
         * 把所有的线程Node从等待condition的队列转移到等待lock的队列。
         *
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        public final void signalAll() {
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
```

```
            Node first = firstWaiter;
            if (first != null)
                doSignalAll(first);
        }

        /**
         * Implements uninterruptible condition wait.
         * 实现非中断的condition等待
         *
         * <ol>
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled.
         * <li> Reacquire by invoking specialized version of
         *      {@link #acquire} with saved state as argument.
         * </ol>
         */
        public final void awaitUninterruptibly() {
            Node node = addConditionWaiter(); // 将当前线程封装成Node添加到等待condition
队列中
            int savedState = fullyRelease(node); // 这里为什么要去释放呢？释放不成功还会直
接cancel  这里结合场景说一下，如果当前线程想要在condition上做await操作，那么它一定是已经获取
到锁了，这是第一。第二，已获取到锁的线程需要await，那么它一定要释放锁，把资源交出去，直到它被唤
醒进行竞争。
            boolean interrupted = false;
            while (!isOnSyncQueue(node)) { // 判断当前node是否已在sync队列里（就是等待lock
的队列），如果没有的话，就自己阻塞了
                LockSupport.park(this);
                if (Thread.interrupted()) // 即使有中断，也只是记录状态，不响应
                    interrupted = true;
            }
            if (acquireQueued(node, savedState) || interrupted) // 能到这里，说明已经在sy
nc队列里了，尝试获取锁，获取失败也阻塞。如果在加入sync队列时发生了中断，或者在sync获取锁的时候
发生中断，都会重新中断。
                selfInterrupt();
        }

        /*
         * For interruptible waits, we need to track whether to throw
         * InterruptedException, if interrupted while blocked on
         * condition, versus reinterrupt current thread, if
         * interrupted while blocked waiting to re-acquire.
         * 对于可中断的waits，需要跟踪是否抛出InterruptedException，
         * 如果在condition阻塞过程中发生中断，
         * 相对的重新中断当前线程
         * 如果在等待重新获取的阻塞时发生中断
         */
```

```
        /** Mode meaning to reinterrupt on exit from wait */
        // 在退出时重新中断
        private static final int REINTERRUPT =  1;
        /** Mode meaning to throw InterruptedException on exit from wait */
        // 在退出时抛出中断异常
        private static final int THROW_IE    = -1;

        /**
         * Checks for interrupt, returning THROW_IE if interrupted
         * before signalled, REINTERRUPT if after signalled, or
         * 0 if not interrupted.
         * 检查中断
         * 如果在获取到signal之前发生中断，返回THROW_IE
         * 如果在获取到signal之后发生中断，返回REINTERRUPT
         * 没有中断发生，返回0
         */
        private int checkInterruptWhileWaiting(Node node) {
            return Thread.interrupted() ?
                (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) : // transf
erAfterCancelledWait如果能成功将ws从CONDITION更新成0，会尝试将node加入到sync队列，返回true
表示被signal之前被cancel
                0;
        }

        /**
         * Throws InterruptedException, reinterrupts current thread, or
         * does nothing, depending on mode.
         * 根据模式，抛出中断异常，或者重新中断当前线程，或者啥也不做
         *
         */
        private void reportInterruptAfterWait(int interruptMode)
            throws InterruptedException {
            if (interruptMode == THROW_IE)
                throw new InterruptedException();
            else if (interruptMode == REINTERRUPT)
                selfInterrupt();
        }

        /**
         * Implements interruptible condition wait.
         * 实现可中断condition等待
         *
         * <ol>
         * <li> If current thread is interrupted, throw InterruptedException.
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled or interrupted.
         * <li> Reacquire by invoking specialized version of
```

```
         *      {@link #acquire} with saved state as argument.
         * <li> If interrupted while blocked in step 4, throw InterruptedException.
         * 1、如果当前线程被中断，抛出异常。
         * 2、保存由getState返回的锁的state值。
         * 3、调用release方法，将保存的state值作为参数，如果release失败，抛出IllegalMonitor
StateException异常。（说明该线程的操作非法，类似于Ojbect上的wait与notify）
         * 4、阻塞，直到被唤醒或者被中断
         * 5、通过使用保存的state值，调用特殊版本的acquire方法，来重新获取。
         * 6、如果在第4步阻塞时被中断，抛出中断异常。
         *
         * </ol>
         */
        public final void await() throws InterruptedException {
            if (Thread.interrupted())                                       // 0、
上来先看中断状态，如果已经中断了，直接抛出异常。
                throw new InterruptedException();
            Node node = addConditionWaiter();                               // 1、
将当前线程加入到Condition队列中。与不可中断的await一样
            int savedState = fullyRelease(node);                            // 2、
当前线程放弃对锁的竞争，释放资源唤醒后继进行锁获取
            int interruptMode = 0;                                          //
            while (!isOnSyncQueue(node)) {                                  // 3、
判断当前线程node是否在sync队列里
                LockSupport.park(this);                                     // 4、
如果不在sync队列，说明没有满足的Condition，进行park
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)    // 5、
检查在wait时发生的中断，如果没有中断，返回0，当返回值!=0时，跳出循环等待
                    break;
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)    // 6、
调用不可中断的acquireQueued方法竞争，竞争成功返回中断状态。如果竞争时发生中断，并且中断模式为T
HROW_IE（在获取到信号之前就被中断了）
                interruptMode = REINTERRUPT;                                 // 7、
中断模式改为REINTERRUPT（在获取信号之后被中断）
            if (node.nextWaiter != null) // clean up if cancelled               // 8、
如果当前节点的nextWaiter不为null，从CONDITION队列的firstWaiter开始清理一遍非CONDITION状态的
节点
                unlinkCancelledWaiters();
            if (interruptMode != 0)                                         // 9、
如果中断模式不是0（意味着发生过中断），按照中断模式，调用reportInterruptAfterWait方法抛出异常
或者恢复中断状态（就是重新将中断标识位置为中断）
                reportInterruptAfterWait(interruptMode);
        }

        /**
         * Implements timed condition wait.
         * 实现超时condition等待
         * <ol>
         * <li> If current thread is interrupted, throw InterruptedException.
```

```
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled, interrupted, or timed out.
         * <li> Reacquire by invoking specialized version of
         *      {@link #acquire} with saved state as argument.
         * <li> If interrupted while blocked in step 4, throw InterruptedException.
         * 1、如果当前线程被中断，抛出InterruptedException。
         * 2、保存getState方法返回的state值。
         * 3、调用release方法，使用保存的state值作为参数。如果调用失败，抛出IllegalMonitorS
tateException。
         * 4、阻塞，直到被唤醒，或者被中断，或者超时。
         * 5、通过带着保存的state值调用特殊版本的acquire方法，来重新获取。
         * 6、如果在步骤4阻塞时被中断，抛出中断异常。
         *
         * </ol>
         */
        public final long awaitNanos(long nanosTimeout)
                throws InterruptedException {
            if (Thread.interrupted())
                throw new InterruptedException();
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            final long deadline = System.nanoTime() + nanosTimeout;
            int interruptMode = 0;
            while (!isOnSyncQueue(node)) {
                if (nanosTimeout <= 0L) {
                    transferAfterCancelledWait(node); // 如果超时时间<=0，调用transferAf
terCancelledWait方法尝试进行该node入sync队列。
                    break;
                }
                if (nanosTimeout >= spinForTimeoutThreshold) // 如果剩余的超时时间比自旋
的等待时间阈值高，那么直接park
                    LockSupport.parkNanos(this, nanosTimeout);
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0) // 检查是
否发生过中断，如果有，跳出循环
                    break;
                nanosTimeout = deadline - System.nanoTime();
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null)
                unlinkCancelledWaiters();
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode);
            return deadline - System.nanoTime(); // 返回剩余的等待纳秒数
        }

        /**
```

```
         * Implements absolute timed condition wait.
         * 实现绝对定时condition等待。
         * <ol>
         * <li> If current thread is interrupted, throw InterruptedException.
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled, interrupted, or timed out.
         * <li> Reacquire by invoking specialized version of
         *      {@link #acquire} with saved state as argument.
         * <li> If interrupted while blocked in step 4, throw InterruptedException.
         * <li> If timed out while blocked in step 4, return false, else true.
         * 1、如果当前线程被中断，抛出InterruptedException。
         * 2、保存getState方法返回的state值。
         * 3、调用release方法，使用保存的state值作为参数。如果调用失败，抛出IllegalMonitorS
tateException。
         * 4、阻塞，直到被唤醒，或者被中断，或者超时。
         * 5、通过带着保存的state值调用特殊版本的acquire方法，来重新获取。
         * 6、如果在步骤4阻塞时被中断，抛出中断异常。
         * 7、如果在步骤4超时了，返回false，否则返回true。
         * </ol>
         */
        public final boolean awaitUntil(Date deadline)
                throws InterruptedException {
            long abstime = deadline.getTime();
            if (Thread.interrupted())
                throw new InterruptedException();
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            boolean timedout = false;
            int interruptMode = 0;
            while (!isOnSyncQueue(node)) {
                if (System.currentTimeMillis() > abstime) {
                    timedout = transferAfterCancelledWait(node); // transferAfterCancel
ledWait如果在信号之前取消了wait，返回true（超时了返回true）
                    break;
                }
                LockSupport.parkUntil(this, abstime);
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null)
                unlinkCancelledWaiters();
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode);
            return !timedout; // 超时了返回false，否则返回true
        }
```

```
        /**
         * Implements timed condition wait.
         * 实现超时condition等待
         * <ol>
         * <li> If current thread is interrupted, throw InterruptedException.
         * <li> Save lock state returned by {@link #getState}.
         * <li> Invoke {@link #release} with saved state as argument,
         *      throwing IllegalMonitorStateException if it fails.
         * <li> Block until signalled, interrupted, or timed out.
         * <li> Reacquire by invoking specialized version of
         *      {@link #acquire} with saved state as argument.
         * <li> If interrupted while blocked in step 4, throw InterruptedException.
         * <li> If timed out while blocked in step 4, return false, else true.
         * 1、如果当前线程被中断，抛出InterruptedException。
         * 2、保存getState方法返回的state值。
         * 3、调用release方法，使用保存的state值作为参数。如果调用失败，抛出IllegalMonitorS
tateException。
         * 4、阻塞，直到被唤醒，或者被中断，或者超时。
         * 5、通过带着保存的state值调用特殊版本的acquire方法，来重新获取。
         * 6、如果在步骤4阻塞时被中断，抛出中断异常。
         * 7、如果在步骤4超时了，返回false，否则返回true。
         * </ol>
         */
        public final boolean await(long time, TimeUnit unit)
                throws InterruptedException {
            long nanosTimeout = unit.toNanos(time);
            if (Thread.interrupted())
                throw new InterruptedException();
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            final long deadline = System.nanoTime() + nanosTimeout;
            boolean timedout = false;
            int interruptMode = 0;
            while (!isOnSyncQueue(node)) {
                if (nanosTimeout <= 0L) {
                    timedout = transferAfterCancelledWait(node);
                    break;
                }
                if (nanosTimeout >= spinForTimeoutThreshold)
                    LockSupport.parkNanos(this, nanosTimeout);
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
                nanosTimeout = deadline - System.nanoTime();
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null)
                unlinkCancelledWaiters();
```

```
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode);
            return !timedout;
        }

        //  support for instrumentation

        /**
         * Returns true if this condition was created by the given
         * synchronization object.
         * 如果该condition是通过给定的同步器创建的，返回true
         *
         * @return {@code true} if owned
         */
        final boolean isOwnedBy(AbstractQueuedSynchronizer sync) {
            return sync == AbstractQueuedSynchronizer.this;
        }

        /**
         * Queries whether any threads are waiting on this condition.
         * Implements {@link AbstractQueuedSynchronizer#hasWaiters(ConditionObject)}.
         * 查询是否在该condition上有线程在等待。
         *
         * @return {@code true} if there are any waiting threads
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        protected final boolean hasWaiters() {
            if (!isHeldExclusively()) // 如果当前线程没有独占锁，抛出异常，这个方法是需要独占锁自己实现的三个方法之一。
                throw new IllegalMonitorStateException();
            for (Node w = firstWaiter; w != null; w = w.nextWaiter) { // 从第一个等待节点开始找，如果有waitStatus是CONDITION的，表示在等待条件，返回true
                if (w.waitStatus == Node.CONDITION) // （在CONDITION队列上的，ws只有CONDITION跟CANCEL状态么？）
                    return true;
            }
            return false;
        }

        /**
         * Returns an estimate of the number of threads waiting on
         * this condition.
         * Implements {@link AbstractQueuedSynchronizer#getWaitQueueLength(ConditionObject)}.
         * 返回预估的在该condition上等待的线程数
         * AbstractQueuedSynchronizer的getWaitQueueLength(ConditionObject)方法会调用
         *
         * @return the estimated number of waiting threads
```

```
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        protected final int getWaitQueueLength() {
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            int n = 0;
            for (Node w = firstWaiter; w != null; w = w.nextWaiter) {
                if (w.waitStatus == Node.CONDITION)
                    ++n;
            }
            return n;
        }

        /**
         * Returns a collection containing those threads that may be
         * waiting on this Condition.
         * Implements {@link AbstractQueuedSynchronizer#getWaitingThreads(ConditionObje
ct)}.
         * 返回包含可能在该Condition上等待的线程集合。
         *
         * @return the collection of threads
         * @throws IllegalMonitorStateException if {@link #isHeldExclusively}
         *         returns {@code false}
         */
        protected final Collection<Thread> getWaitingThreads() {
            if (!isHeldExclusively())
                throw new IllegalMonitorStateException();
            ArrayList<Thread> list = new ArrayList<Thread>();
            for (Node w = firstWaiter; w != null; w = w.nextWaiter) {
                if (w.waitStatus == Node.CONDITION) {
                    Thread t = w.thread;
                    if (t != null)
                        list.add(t);
                }
            }
            return list;
        }
    }

    /**
     * Setup to support compareAndSet. We need to natively implement
     * this here: For the sake of permitting future enhancements, we
     * cannot explicitly subclass AtomicInteger, which would be
     * efficient and useful otherwise. So, as the lesser of evils, we
     * natively implement using hotspot intrinsics API. And while we
     * are at it, we do the same for other CASable fields (which could
     * otherwise be done with atomic field updaters).
     * 设置以支持CAS。
```

```
     * 需要在这里进行本地实现：
     * 为了允许未来增强，我们不能显式集成AtomicInteger类，否则就是有效和有用的。（如果不是为了增强，就可以用AtomicXXX操作了）
     * 所以，作为较小的弊端，本地实现使用hotspot内在函数API。
     * 当我们这样做时，对其他可以使用CAS字段也这样做（否则可以使用原子字段更新程序来完成）
     */
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long stateOffset;
    private static final long headOffset;
    private static final long tailOffset;
    private static final long waitStatusOffset;
    private static final long nextOffset;

    static {
        try {
            stateOffset = unsafe.objectFieldOffset
                (AbstractQueuedSynchronizer.class.getDeclaredField("state"));
            headOffset = unsafe.objectFieldOffset
                (AbstractQueuedSynchronizer.class.getDeclaredField("head"));
            tailOffset = unsafe.objectFieldOffset
                (AbstractQueuedSynchronizer.class.getDeclaredField("tail"));
            waitStatusOffset = unsafe.objectFieldOffset
                (Node.class.getDeclaredField("waitStatus"));
            nextOffset = unsafe.objectFieldOffset
                (Node.class.getDeclaredField("next"));

        } catch (Exception ex) { throw new Error(ex); }
    }

    /**
     * CAS head field. Used only by enq.
     */
    private final boolean compareAndSetHead(Node update) {
        return unsafe.compareAndSwapObject(this, headOffset, null, update);
    }

    /**
     * CAS tail field. Used only by enq.
     */
    private final boolean compareAndSetTail(Node expect, Node update) {
        return unsafe.compareAndSwapObject(this, tailOffset, expect, update);
    }

    /**
     * CAS waitStatus field of a node.
     */
    private static final boolean compareAndSetWaitStatus(Node node,
                                                         int expect,
                                                         int update) {
```

```
        return unsafe.compareAndSwapInt(node, waitStatusOffset,
                                        expect, update);
    }

    /**
     * CAS next field of a node.
     */
    private static final boolean compareAndSetNext(Node node,
                                                   Node expect,
                                                   Node update) {
        return unsafe.compareAndSwapObject(node, nextOffset, expect, update);
    }
}
```

Condition

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent.locks;
import java.util.concurrent.TimeUnit;
import java.util.Date;

/**
 * {@code Condition} factors out the {@code Object} monitor
 * methods ({@link Object#wait() wait}, {@link Object#notify notify}
 * and {@link Object#notifyAll notifyAll}) into distinct objects to
 * give the effect of having multiple wait-sets per object, by
 * combining them with the use of arbitrary {@link Lock} implementations.
 * Where a {@code Lock} replaces the use of {@code synchronized} methods
 * and statements, a {@code Condition} replaces the use of the Object
 * monitor methods.
 * Condition将Object监视器方法（wait、notify、notifyAll）分解到不同的对象，通过将他们与使用任意的Lock实现进行组合，以产生每个对象具有多个等待集的效果。
 *在使用Lock替换synchronized使用的地方，用Condition替换Object监视器使用（Lock与Condition配对，synchronized与ObjectMonitor配对（就是Object.wait/notify那一套））
 *
 * <p>Conditions (also known as <em>condition queues</em> or
 * <em>condition variables</em>) provide a means for one thread to
 * suspend execution (to &quot;wait&quot;) until notified by another
 * thread that some state condition may now be true.  Because access
 * to this shared state information occurs in different threads, it
 * must be protected, so a lock of some form is associated with the
 * condition. The key property that waiting for a condition provides
 * is that it <em>atomically</em> releases the associated lock and
 * suspends the current thread, just like {@code Object.wait}.
 * Conditions（也叫条件队列或者条件变量）提供了一种一个线程暂停执行（等待）直到一些状态条件可能为true时被另一个线程唤醒的能力。
 * 因为访问共享变量信息发生在不同的线程，它必须受保护，所以需要某种形式的锁与条件相关联。
 * 等待条件提供的关键属性是原子地释放关联的锁并且暂停当前线程，就像Object.wait一样。
 *
 * <p>A {@code Condition} instance is intrinsically bound to a lock.
 * To obtain a {@code Condition} instance for a particular {@link Lock}
 * instance use its {@link Lock#newCondition newCondition()} method.
 * Condition实例本质上（intrinsically）绑定到了一个锁。
 * 获取（obtain）特定Lock实例的Condition实例使用Lock的newCondition()方法。
```

```
 *
 * <p>As an example, suppose we have a bounded buffer which supports
 * {@code put} and {@code take} methods.  If a
 * {@code take} is attempted on an empty buffer, then the thread will block
 * until an item becomes available; if a {@code put} is attempted on a
 * full buffer, then the thread will block until a space becomes available.
 * We would like to keep waiting {@code put} threads and {@code take}
 * threads in separate wait-sets so that we can use the optimization of
 * only notifying a single thread at a time when items or spaces become
 * available in the buffer. This can be achieved using two
 * {@link Condition} instances.
 * 举例来说，假设我们有一个有界的buffer，支持put和take方法。
 * 如果有线程对空buffer尝试take，当前线程会阻塞直到有东西变得可用（就是buffer里有东西）
 * 如果有线程对满buffer尝试put，当前线程会阻塞直到有空地方可用
 * 我们希望将等待的多个put线程与多个take线程保持在独立的等待集，这样我们就可以优化（optimizat
ion）成当buffer里有东西或者有空间可用时，每次只唤醒一个线程。
 * 可以通过使用两个Condition实例来实现
 *
 * <pre>
 * class BoundedBuffer {
 *   <b>final Lock lock = new ReentrantLock();</b> // 等下再返回看ReentrantLock实现
 *   final Condition notFull  = <b>lock.newCondition(); </b> // 同一个lock创建两个Condit
ion实例
 *   final Condition notEmpty = <b>lock.newCondition(); </b>
 *
 *   final Object[] items = new Object[100];
 *   int putptr, takeptr, count;
 *
 *   public void put(Object x) throws InterruptedException {
 *     <b>lock.lock();
 *     try {</b>
 *       // 注意这里的while循环，特别有用，因为线程被唤醒后，可能仍然没有满足的条件（比如条件
被别的线程抢了），这时候就得重新进入挂起-唤醒的循环
 *       //（这其实就是虚假唤醒的语义，下面会解释）
 *       while (count == items.length)
 *         <b>notFull.await();</b> // 挂起，并释放锁
 *       items[putptr] = x;
 *       if (++putptr == items.length) putptr = 0;
 *       ++count;
 *       <b>notEmpty.signal();</b> // buffer里有东西了，唤醒一个挂起的take
 *     <b>} finally {
 *       lock.unlock(); // 记得释放锁
 *     }</b>
 *   }
 *
 *   public Object take() throws InterruptedException {
 *     <b>lock.lock();
 *     try {</b>
 *       while (count == 0)
```

```
 *         <b>notEmpty.await();</b>
 *       Object x = items[takeptr];
 *       if (++takeptr == items.length) takeptr = 0;
 *       --count;
 *       <b>notFull.signal();</b>
 *       return x;
 *     <b>} finally {
 *       lock.unlock();
 *     }</b>
 *   }
 * }
 * </pre>
 *
 * (The {@link java.util.concurrent.ArrayBlockingQueue} class provides
 * this functionality, so there is no reason to implement this
 * sample usage class.)
 * ArrayBlockingQueue类提供了这些方法，所有没有必要去实现这个相同用处的类
 *
 * <p>A {@code Condition} implementation can provide behavior and semantics
 * that is
 * different from that of the {@code Object} monitor methods, such as
 * guaranteed ordering for notifications, or not requiring a lock to be held
 * when performing notifications.
 * If an implementation provides such specialized semantics then the
 * implementation must document those semantics.
 * Condition的实现可以提供与Object监视器不同的行为与语义，例如保证通知顺序，或者执行通知时不要求持有锁。
 * 如果实现提供这些专门的语义，那么实现必须记录下来。
 *
 * <p>Note that {@code Condition} instances are just normal objects and can
 * themselves be used as the target in a {@code synchronized} statement,
 * and can have their own monitor {@link Object#wait wait} and
 * {@link Object#notify notification} methods invoked.
 * Acquiring the monitor lock of a {@code Condition} instance, or using its
 * monitor methods, has no specified relationship with acquiring the
 * {@link Lock} associated with that {@code Condition} or the use of its
 * {@linkplain #await waiting} and {@linkplain #signal signalling} methods.
 * It is recommended that to avoid confusion you never use {@code Condition}
 * instances in this way, except perhaps within their own implementation.
 * 注意，Condition的实例就是普通的object，并且他自身可以用于synchronized statement，并且也拥有自己的监视器wait和notify方法调用。
 * 获取Condition实例的监视器锁，或者使用它的监视器方法，跟获取该实例关联的Lock，或者使用await、signal方法没有任何特定的关系。
 * 为了避免混淆，建议不要以这种方式使用Condition的实例，除非在他们自己的实现中
 * （跟Lock的建议一样）
 *
 * <p>Except where noted, passing a {@code null} value for any parameter
 * will result in a {@link NullPointerException} being thrown.
 * 除非另有说明，传参时传个null会抛出NullPointerException异常。
```

```
 *
 * <h3>Implementation Considerations</h3>
 * 注意事项
 * <p>When waiting upon a {@code Condition}, a &quot;<em>spurious
 * wakeup</em>&quot; is permitted to occur, in
 * general, as a concession to the underlying platform semantics.
 * This has little practical impact on most application programs as a
 * {@code Condition} should always be waited upon in a loop, testing
 * the state predicate that is being waited for.  An implementation is
 * free to remove the possibility of spurious wakeups but it is
 * recommended that applications programmers always assume that they can
 * occur and so always wait in a loop.
 * 当等待Condition时（就是调用await后等待唤醒时），允许虚假唤醒（spurious wakeup）发生，通常，作为对底层平台语义的让步（concession）。
 * 这对大多数应用几乎没有实际影响，因为应用始终在循环中等待Condition，在循环中检测等待的状态谓词。
 * 实现可以自由的消除虚假唤醒的可能性，不过建议应用程序员始终假设他们可能发生，并且始终在循环中等待（与检测）。
 *
 * <p>The three forms of condition waiting
 * (interruptible, non-interruptible, and timed) may differ in their ease of
 * implementation on some platforms and in their performance characteristics.
 * In particular, it may be difficult to provide these features and maintain
 * specific semantics such as ordering guarantees.
 * Further, the ability to interrupt the actual suspension of the thread may
 * not always be feasible to implement on all platforms.
 * 条件等待的三种形式（可中断、不可中断、定时）在某些平台上实现的难易程度和性能特征方面可能有所不同。
 * 特别是，可能很难提供这些功能并维护特定的语义，例如排序保证。
 * 此外，在所有平台上实现中断线程实际挂起的能力并不总是可行。
 *
 * <p>Consequently, an implementation is not required to define exactly the
 * same guarantees or semantics for all three forms of waiting, nor is it
 * required to support interruption of the actual suspension of the thread.
 * 因此，不需要一个实现针对这三种形式定义完全（exactly 确切）相同的保证或者语义，也不需要支持中断线程实际挂起的。
 *
 * <p>An implementation is required to
 * clearly document the semantics and guarantees provided by each of the
 * waiting methods, and when an implementation does support interruption of
 * thread suspension then it must obey the interruption semantics as defined
 * in this interface.
 * 要求实现明确记录下每个等待方法提供的语义和保证，并且当实现支持中断线程挂起，那么它必须遵循这个接口中定义的中断语义
 *
 * <p>As interruption generally implies cancellation, and checks for
 * interruption are often infrequent, an implementation can favor responding
 * to an interrupt over normal method return. This is true even if it can be
 * shown that the interrupt occurred after another action that may have
```

```
 * unblocked the thread. An implementation should document this behavior.
 * 中断通常意味着（implies）取消，并且中断检查通常很少发生，因此实现类可以偏向于响应中断而不是正常的方法返回。
 * 即使可以证明 中断发生在 另一个动作已经解除了本线程阻塞之后，也是如此。实现类中应该记录下该行为
 *
 * @since 1.5
 * @author Doug Lea
 */
public interface Condition {

    /**
     * Causes the current thread to wait until it is signalled or
     * {@linkplain Thread#interrupt interrupted}.
     * 使当前线程等待，直到收到信号或者被中断
     *
     * <p>The lock associated with this {@code Condition} is atomically
     * released and the current thread becomes disabled for thread scheduling
     * purposes and lies dormant until <em>one</em> of four things happens:
     * 原子操作释放Condition关联的锁，当前线程转变为不可用的线程调度目标，并且挂起，直到以下四种情况之一发生：
     *
     * <ul>
     * <li>Some other thread invokes the {@link #signal} method for this
     * {@code Condition} and the current thread happens to be chosen as the
     * thread to be awakened; or
     * <li>Some other thread invokes the {@link #signalAll} method for this
     * {@code Condition}; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts} the
     * current thread, and interruption of thread suspension is supported; or
     * <li>A &amp;quot;<em>spurious wakeup</em>&amp;quot; occurs.
     * </ul>
     * 1.其他一些线程调用了这个Condition的signal方法，并且当前线程被选中为唤醒线程
     * 2.其他一些线程调用了这个Condition的signalAll方法
     * 3.其他一些线程中断当前线程，并且当前线程支持中断线程挂起
     * 4.发生虚假唤醒
     *
     * <p>In all cases, before this method can return the current thread must
     * re-acquire the lock associated with this condition. When the
     * thread returns it is <em>guaranteed</em> to hold this lock.
     * 在所有情况下，在此方法可以返回到当前线程（继续执行）之前，当前线程必须重新获取到这个condition关联的锁。
     * 当线程返回时，它保证持有这个锁
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while waiting
     * and interruption of thread suspension is supported,
```

```
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared. It is not specified, in the first
     * case, whether or not the test for interruption occurs before the lock
     * is released.
     * 如果当前线程：
     * 1. 在进入该方法时设置了中断状态
     * 2. 或者在等待时被中断，并且当前线程支持中断挂起
     * 那么会抛出InterruptedException异常并且清理当前线程的中断状态。
     * 对于第一种情况，没有规定要检测在中断发生前锁是否被释放。
     *
     * <p><b>Implementation Considerations</b>
     * 注意事项
     * <p>The current thread is assumed to hold the lock associated with this
     * {@code Condition} when this method is called.
     * It is up to the implementation to determine if this is
     * the case and if not, how to respond. Typically, an exception will be
     * thrown (such as {@link IllegalMonitorStateException}) and the
     * implementation must document that fact.
     * 在调用此方法时，假定当前线程拥有该Condition关联的锁。
     * 由实现去决定如果不满足上述假设时该如何响应。
     * 通常会抛出IllegalMonitorStateException异常，并且实现必须记录下该事实（写明处理规则）
     *
     * <p>An implementation can favor responding to an interrupt over normal
     * method return in response to a signal. In that case the implementation
     * must ensure that the signal is redirected to another waiting thread, if
     * there is one.
     * 实现响应信号时可以倾向于响应中断而不是正常方法返回。
     * 实现必须确保信号量被重定向到另一个等待线程，如果还有等待线程的话。（就是如果当前线程唤醒
失败，需要把唤醒信号量传递给其他等待线程）
     *
     * @throws InterruptedException if the current thread is interrupted
     *         (and interruption of thread suspension is supported)
     */
    void await() throws InterruptedException;

    /**
     * Causes the current thread to wait until it is signalled.
     * (这些都与await()方法一样）
     * <p>The lock associated with this condition is atomically
     * released and the current thread becomes disabled for thread scheduling
     * purposes and lies dormant until <em>one</em> of three things happens:
     * <ul>
     * <li>Some other thread invokes the {@link #signal} method for this
     * {@code Condition} and the current thread happens to be chosen as the
     * thread to be awakened; or
     * <li>Some other thread invokes the {@link #signalAll} method for this
     * {@code Condition}; or
     * <li>A &amp;quot;<em>spurious wakeup</em>&amp;quot; occurs.
```

```
     * </ul>
     * 这里少了一种打断wait的方法，那就是少了中断
     *
     * <p>In all cases, before this method can return the current thread must
     * re-acquire the lock associated with this condition. When the
     * thread returns it is <em>guaranteed</em> to hold this lock.
     *
     * <p>If the current thread's interrupted status is set when it enters
     * this method, or it is {@linkplain Thread#interrupt interrupted}
     * while waiting, it will continue to wait until signalled. When it finally
     * returns from this method its interrupted status will still
     * be set.
     * 注意这里是不一样的：
     * 如果当前线程在进入方法之前被设置了中断，或者在等待时被中断，它将仍然处于等待状态直到收到唤醒信号。
     * 当它最终返回的时候，它的中断状态仍旧被设置。（不是清除中断状态）
     * 所以，它不会抛出异常
     *
     * <p><b>Implementation Considerations</b>
     *
     * <p>The current thread is assumed to hold the lock associated with this
     * {@code Condition} when this method is called.
     * It is up to the implementation to determine if this is
     * the case and if not, how to respond. Typically, an exception will be
     * thrown (such as {@link IllegalMonitorStateException}) and the
     * implementation must document that fact.
     */
    void awaitUninterruptibly();

    /**
     * Causes the current thread to wait until it is signalled or interrupted,
     * or the specified waiting time elapses.
     * 当前线程等待直到收到唤醒信号或者被中断，或者是指定的等待时间超时。
     *
     * <p>The lock associated with this condition is atomically
     * released and the current thread becomes disabled for thread scheduling
     * purposes and lies dormant until <em>one</em> of five things happens:
     * <ul>
     * <li>Some other thread invokes the {@link #signal} method for this
     * {@code Condition} and the current thread happens to be chosen as the
     * thread to be awakened; or
     * <li>Some other thread invokes the {@link #signalAll} method for this
     * {@code Condition}; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts} the
     * current thread, and interruption of thread suspension is supported; or
     * <li>The specified waiting time elapses; or
     * <li>A &amp;quot;<em>spurious wakeup</em>&amp;quot; occurs.
     * </ul>
     * 这里有五种方法可以打断线程的挂起：
```

```
     * 1. signal并且当前线程被选中
     * 2. signalAll
     * 3. interrupted
     * 4. 等待超时（相较await()，多了个这个）
     * 5. 虚假唤醒
     *
     * <p>In all cases, before this method can return the current thread must
     * re-acquire the lock associated with this condition. When the
     * thread returns it is <em>guaranteed</em> to hold this lock.
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while waiting
     * and interruption of thread suspension is supported,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared. It is not specified, in the first
     * case, whether or not the test for interruption occurs before the lock
     * is released.
     *
     * <p>The method returns an estimate of the number of nanoseconds
     * remaining to wait given the supplied {@code nanosTimeout}
     * value upon return, or a value less than or equal to zero if it
     * timed out. This value can be used to determine whether and how
     * long to re-wait in cases where the wait returns but an awaited
     * condition still does not hold. Typical uses of this method take
     * the following form:
     * 这个方法返回给定超时纳秒数的剩余纳秒数预估值（就是如果在给定的超时纳秒数之前收到了唤醒信号，就返回剩余的等待时间），如果超时了，返回一个小于等于0的值。
     * 传入的超时纳秒数M，线程等待了N秒，返回值为M-N
     * 这个返回值（剩余等待时间）可以用来确定当等待返回了但是等待条件仍未满足时，是否重新等待与重新等待的时间。
     * 此方法的典型用法如下：
     *
     *  <pre> {@code
     * boolean aMethod(long timeout, TimeUnit unit) {
     *   long nanos = unit.toNanos(timeout);
     *   lock.lock();
     *   try {
     *     while (!conditionBeingWaitedFor()) { // 判断是否重新等待
     *       if (nanos <= 0L)
     *         return false;
     *       nanos = theCondition.awaitNanos(nanos); // 等待指定纳秒数
     *     }
     *     // ...
     *   } finally {
     *     lock.unlock();
     *   }
```

```
     * }}</pre>
     *
     * <p>Design note: This method requires a nanosecond argument so
     * as to avoid truncation errors in reporting remaining times.
     * Such precision loss would make it difficult for programmers to
     * ensure that total waiting times are not systematically shorter
     * than specified when re-waits occur.
     * 这个方法要求传入的参数为纳秒值，以避免报告剩余时间时出现截断错误。
     * 这种精度损失会导致程序员难以保证在重新等待发生时，总等待时间（系统性的）不短于指定时间。
     * 理解上总等待时间要>=指定时间（对应超时时，返回值为<=0），如果发生精度缺失，会导致总等待时间<指定时间的可能。
     *
     * <p><b>Implementation Considerations</b>
     *
     * <p>The current thread is assumed to hold the lock associated with this
     * {@code Condition} when this method is called.
     * It is up to the implementation to determine if this is
     * the case and if not, how to respond. Typically, an exception will be
     * thrown (such as {@link IllegalMonitorStateException}) and the
     * implementation must document that fact.
     *
     * <p>An implementation can favor responding to an interrupt over normal
     * method return in response to a signal, or over indicating the elapse
     * of the specified waiting time. In either case the implementation
     * must ensure that the signal is redirected to another waiting thread, if
     * there is one.
     * （记得发生中断后要把唤醒信号重定向到另一个等待线程）
     *
     * @param nanosTimeout the maximum time to wait, in nanoseconds
     * @return an estimate of the {@code nanosTimeout} value minus
     *         the time spent waiting upon return from this method.
     *         A positive value may be used as the argument to a
     *         subsequent call to this method to finish waiting out
     *         the desired time.  A value less than or equal to zero
     *         indicates that no time remains.
     * indicates 表示，remains 剩余
     * @throws InterruptedException if the current thread is interrupted
     *         (and interruption of thread suspension is supported)
     */
    long awaitNanos(long nanosTimeout) throws InterruptedException;

    /**
     * Causes the current thread to wait until it is signalled or interrupted,
     * or the specified waiting time elapses. This method is behaviorally
     * equivalent to:
     *  <pre> {@code awaitNanos(unit.toNanos(time)) > 0}</pre>
     * 这个方法等效于awaitNanos(unit.toNanos(time)) > 0，就是如果在等待时间内被唤醒，返回true，等待超时返回false
     * 这里有个问题，0算等待超时么？可以看具体实现，比如AQS里面的ConditionObject
```

```
     *
     * @param time the maximum time to wait
     * @param unit the time unit of the {@code time} argument
     * @return {@code false} if the waiting time detectably elapsed
     *         before return from the method, else {@code true} // 如果在返回之前可检测到
等待时间超时，返回true，否则返回false
     * @throws InterruptedException if the current thread is interrupted
     *         (and interruption of thread suspension is supported)
     */
    boolean await(long time, TimeUnit unit) throws InterruptedException;

    /**
     * Causes the current thread to wait until it is signalled or interrupted,
     * or the specified deadline elapses.
     * 当前线程等待直到收到信号、被中断，或者超过指定截止日期。
     *
     * <p>The lock associated with this condition is atomically
     * released and the current thread becomes disabled for thread scheduling
     * purposes and lies dormant until <em>one</em> of five things happens:
     * <ul>
     * <li>Some other thread invokes the {@link #signal} method for this
     * {@code Condition} and the current thread happens to be chosen as the
     * thread to be awakened; or
     * <li>Some other thread invokes the {@link #signalAll} method for this
     * {@code Condition}; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts} the
     * current thread, and interruption of thread suspension is supported; or
     * <li>The specified deadline elapses; or
     * <li>A &quot;<em>spurious wakeup</em>&quot; occurs.
     * </ul>
     * 有五种方式可以打断线程的挂起：
     * 1. signal并且当先线程被选中
     * 2. signalAll
     * 3. interrupted
     * 4. 超过指定的截止日期
     * 5. 虚假唤醒
     *
     * <p>In all cases, before this method can return the current thread must
     * re-acquire the lock associated with this condition. When the
     * thread returns it is <em>guaranteed</em> to hold this lock.
     * （源码里多了个空行，不严谨了-.-）
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while waiting
     * and interruption of thread suspension is supported,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
```

```
     * interrupted status is cleared. It is not specified, in the first
     * case, whether or not the test for interruption occurs before the lock
     * is released.
     *
     *
     * <p>The return value indicates whether the deadline has elapsed,
     * which can be used as follows:
     * 返回值表示当前方法是否已经超过了截止日期
     *
     *  <pre> {@code
     * boolean aMethod(Date deadline) {
     *   boolean stillWaiting = true;
     *   lock.lock();
     *   try {
     *     while (!conditionBeingWaitedFor()) {
     *       if (!stillWaiting)
     *         return false;
     *       stillWaiting = theCondition.awaitUntil(deadline); // 继续等待到原来的截止日
期（awaitNanos是等待到剩余时间）
     *     }
     *     // ...
     *   } finally {
     *     lock.unlock();
     *   }
     * }}</pre>
     *
     * <p><b>Implementation Considerations</b>
     *
     * <p>The current thread is assumed to hold the lock associated with this
     * {@code Condition} when this method is called.
     * It is up to the implementation to determine if this is
     * the case and if not, how to respond. Typically, an exception will be
     * thrown (such as {@link IllegalMonitorStateException}) and the
     * implementation must document that fact.
     *
     * <p>An implementation can favor responding to an interrupt over normal
     * method return in response to a signal, or over indicating the passing
     * of the specified deadline. In either case the implementation
     * must ensure that the signal is redirected to another waiting thread, if
     * there is one.
     *
     * @param deadline the absolute time to wait until
     * @return {@code false} if the deadline has elapsed upon return, else // （upon 在
...之上）
     *         {@code true}
     * @throws InterruptedException if the current thread is interrupted
     *         (and interruption of thread suspension is supported)
     */
    boolean awaitUntil(Date deadline) throws InterruptedException;
```

```
    /**
     * Wakes up one waiting thread.
     * 唤醒一个等待线程
     *
     * <p>If any threads are waiting on this condition then one
     * is selected for waking up. That thread must then re-acquire the
     * lock before returning from {@code await}.
     * 如果在这个condition下有许多线程在等待，那么会选择一个唤醒。
     * 选中的线程必须在await返回之前重新获取锁。
     *
     * <p><b>Implementation Considerations</b>
     *
     * <p>An implementation may (and typically does) require that the
     * current thread hold the lock associated with this {@code
     * Condition} when this method is called. Implementations must
     * document this precondition and any actions taken if the lock is
     * not held. Typically, an exception such as {@link
     * IllegalMonitorStateException} will be thrown.
     * 实现可能（基本是必须）要求调用该方法的线程必须持有该Condition的关联锁，实现必须记录下先决条件和如果没有持有锁下的任何操作。
     * 如果没有持有锁就调用，通常会抛出IllegalMonitorStateException（这是个RuntimeException）。
     */
    void signal();

    /**
     * Wakes up all waiting threads.
     * 唤醒所有等待线程
     *
     * <p>If any threads are waiting on this condition then they are
     * all woken up. Each thread must re-acquire the lock before it can
     * return from {@code await}.
     * 唤醒所有在这个condition上等待的线程，并且每一个线程在从wait返回前必须重新获取到锁
     *（如果是共享锁，所有线程都可以从await转为正式运行，如果是排它锁，就会出现唤醒了一堆，但只有一个会运行，造成部分开销）
     *
     * <p><b>Implementation Considerations</b>
     *
     * <p>An implementation may (and typically does) require that the
     * current thread hold the lock associated with this {@code
     * Condition} when this method is called. Implementations must
     * document this precondition and any actions taken if the lock is
     * not held. Typically, an exception such as {@link
     * IllegalMonitorStateException} will be thrown.
     */
    void signalAll();
}
```

Lock

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent.locks;
import java.util.concurrent.TimeUnit;

/**
 * {@code Lock} implementations provide more extensive locking
 * operations than can be obtained using {@code synchronized} methods
 * and statements.  They allow more flexible structuring, may have
 * quite different properties, and may support multiple associated
 * {@link Condition} objects.
```

```
 * 相比使用synchronized的方法和语句获得的锁操作，Lock的实现提供了更广泛的锁操作。
 * 它允许更灵活的结构，可能具有完全不同的属性，并且可能支持多个关联Condition对象
 * 简单说一下，synchronized methods 指的是 public synchronized getV()这个样子的
 * synchronized statements指的是 public getV() { synchronized(this) { ... }} 这个样子的
 *
 * <p>A lock is a tool for controlling access to a shared resource by
 * multiple threads. Commonly, a lock provides exclusive access to a
 * shared resource: only one thread at a time can acquire the lock and
 * all access to the shared resource requires that the lock be
 * acquired first. However, some locks may allow concurrent access to
 * a shared resource, such as the read lock of a {@link ReadWriteLock}.
 * Lock是一个工具，用来在多线程场景下对共享资源访问（访问包含读写）的限制。
 * 通常来说，lock提供对共享资源的独占访问：在同一时间只能有一个线程可以获得锁，并且所有对共享资源的访问都必须先获取到锁
 * 然而，一些lock（的实现）可能支持并发访问共享资源，例如读锁（ReadWriteLock）
 *
 * 下面开始说明synchronized的特性与不足：
 * <p>The use of {@code synchronized} methods or statements provides
 * access to the implicit monitor lock associated with every object, but
 * forces all lock acquisition and release to occur in a block-structured way:
 * when multiple locks are acquired they must be released in the opposite
 * order, and all locks must be released in the same lexical scope in which
 * they were acquired.
 * 使用synchronized方法或者语句提供对关联对象的隐式监视器锁访问，但是强制所有锁的获取与释放都按块结构（block-structured）方式进行：
 * 当获取多个锁时，必须按相反的顺序进行释放，并且必须在与获取锁同一个词法范围内释放锁。
 *
 * <p>While the scoping mechanism for {@code synchronized} methods
 * and statements makes it much easier to program with monitor locks,
 * and helps avoid many common programming errors involving locks,
 * there are occasions where you need to work with locks in a more
 * flexible way. For example, some algorithms for traversing
 * concurrently accessed data structures require the use of
 * &amp;quot;hand-over-hand&amp;quot; or &amp;quot;chain locking&amp;quot;: you
 * acquire the lock of node A, then node B, then release A and acquire
 * C, then release B and acquire D and so on.  Implementations of the
 * {@code Lock} interface enable the use of such techniques by
 * allowing a lock to be acquired and released in different scopes,
 * and allowing multiple locks to be acquired and released in any
 * order.
 * 尽管synchronized方法与语句的作用域机制使得使用监视器锁变成变得容易，
 * 并且可以帮助避免许多涉及到锁的常见编程错误，但在某些情况（occasion）下，需要更灵活的锁方式。
 * 例如一些遍历（traverse）并发访问数据结构的算法需要使用交换锁（hand-over-hand）或者链锁（chain locking），向下面这样：
 * 按这样的顺序操作锁：加锁节点A-->加锁节点B-->解锁节点A-->加锁节点C-->解锁节点B-->加锁节点D这样
 * Lock接口的实现，确保可以使用如下技术：允许在不同范围内获取和释放锁，并允许以任意顺序获取和释放多个锁
```

```
 *
 * <p>With this increased flexibility comes additional
 * responsibility. The absence of block-structured locking removes the
 * automatic release of locks that occurs with {@code synchronized}
 * methods and statements. In most cases, the following idiom
 * should be used:
 * 随着灵活性的提高，责任（responsibility）也随之（comes带来）增加。
 * 块结构锁的缺失消除了synchronized方法与语句中发生的锁自动释放（Lock接口的实现不会自动释放锁
)
 * 在大多数情况下，应使用以下习惯用法
 *
 *  <pre> {@code
 * Lock l = ...;
 * l.lock();
 * try {
 *   // access the resource protected by this lock
 * } finally {
 *   l.unlock(); // 手工释放
 * }}</pre>
 *
 * When locking and unlocking occur in different scopes, care must be
 * taken to ensure that all code that is executed while the lock is
 * held is protected by try-finally or try-catch to ensure that the
 * lock is released when necessary.
 * 当在不同的作用域获得锁与释放锁时，必须注意确保所有在持有锁期间执行的方法必须被try-finally或
者try-catch保护（包裹），确保在必要时刻可以释放锁
 *
 * <p>{@code Lock} implementations provide additional functionality
 * over the use of {@code synchronized} methods and statements by
 * providing a non-blocking attempt to acquire a lock ({@link
 * #tryLock()}), an attempt to acquire the lock that can be
 * interrupted ({@link #lockInterruptibly}, and an attempt to acquire
 * the lock that can timeout ({@link #tryLock(long, TimeUnit)}).
 * Lock的实现，相较于synchronized方法与语句提供了额外的方法，例如：
 * tryLock()，非阻塞前提下尝试获取锁
 * lockInterruptibly，尝试获取可以中断的锁
 * tryLock(long, TimeUnit)，有最大等待时间的尝试获取锁
 *
 * <p>A {@code Lock} class can also provide behavior and semantics
 * that is quite different from that of the implicit monitor lock,
 * such as guaranteed ordering, non-reentrant usage, or deadlock
 * detection. If an implementation provides such specialized semantics
 * then the implementation must document those semantics.
 * Lock类还可以提供与隐式监视器锁完全不同的行为与语义，例如保证排序，不可重入使用，或者死锁检测
。
 * 如果实现类提供这些专门的语义，那么实现类必须在文档中记录这些语义。（看起来是对代码说明的限制
)
 *
 * <p>Note that {@code Lock} instances are just normal objects and can
```

```
 * themselves be used as the target in a {@code synchronized} statement.
 * Acquiring the
 * monitor lock of a {@code Lock} instance has no specified relationship
 * with invoking any of the {@link #lock} methods of that instance.
 * It is recommended that to avoid confusion you never use {@code Lock}
 * instances in this way, except within their own implementation.
 * 请注意，Lock的实例只是个普通的对象，并且他们自身也可以用来做synchronized statement的目标对象（synchronized(lock1){...}这个样子）。
 * 获取Lock实例的监视器锁与调用lock实例的任何锁方法没有特定的关系。
 * 为避免混淆，建议不要以这种方式使用Lock实例，除非在他们自己的实现里。
 *
 * <p>Except where noted, passing a {@code null} value for any
 * parameter will result in a {@link NullPointerException} being
 * thrown.
 * 除非另有说明，传递null值给任何参数都会导致抛出NullPointException异常
 *
 * <h3>Memory Synchronization</h3>
 * 内存同步
 *
 * <p>All {@code Lock} implementations <em>must</em> enforce the same
 * memory synchronization semantics as provided by the built-in monitor
 * lock, as described in
 * <a href="https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4">
 * The Java Language Specification (17.4 Memory Model)</a>:
 * 强制所有Lock的实现类都必须与内置监视器锁提供的具有相同的内存锁语义，如这个链接里的所述
 *
 * <ul>
 * <li>A successful {@code lock} operation has the same memory
 * synchronization effects as a successful <em>Lock</em> action.
 * <li>A successful {@code unlock} operation has the same
 * memory synchronization effects as a successful <em>Unlock</em> action.
 * </ul>
 * （不知道这句话实际含义）
 * 成功的Lock动作与成功的lock操作具有相同的内存同步效果
 * 成功的Unlock动作与成功的unlock操作具有相同的内存同步效果
 *
 * Unsuccessful locking and unlocking operations, and reentrant
 * locking/unlocking operations, do not require any memory
 * synchronization effects.
 * 不成功的加锁/解锁操作，和可重入的加锁/解锁操作，不要求任何内存同步效果
 *
 * <h3>Implementation Considerations</h3>
 * 实现的注意事项
 *
 * <p>The three forms of lock acquisition (interruptible,
 * non-interruptible, and timed) may differ in their performance
 * characteristics, ordering guarantees, or other implementation
 * qualities.  Further, the ability to interrupt the <em>ongoing</em>
 * acquisition of a lock may not be available in a given {@code Lock}
```

```
 * class.  Consequently, an implementation is not required to define
 * exactly the same guarantees or semantics for all three forms of
 * lock acquisition, nor is it required to support interruption of an
 * ongoing lock acquisition.  An implementation is required to clearly
 * document the semantics and guarantees provided by each of the
 * locking methods. It must also obey the interruption semantics as
 * defined in this interface, to the extent that interruption of lock
 * acquisition is supported: which is either totally, or only on
 * method entry.
 * 锁获取（acquisition）的三种形式（可中断，不可中断，和有时限的）可能有着不同的行为特征，顺序保证，与其他的实现质量。
 * 此外，中断正在获取锁的操作的能力，在给定的Lock实现中可能不支持。
 * 因此，并不要求实现类为三种形式的锁获取（lock acquisition）定义完全相同的保证或者语义，也不要求支持中断正在获取锁的操作。
 * 要求实现类清楚的记录提供的每个locking方法对应的语义和保证。
 * 同时，实现类也需要遵循在Lock这个接口类中定义的中断语义，以支持锁获取的中断：要么完全支持，要么只在方法入口
 *
 * <p>As interruption generally implies cancellation, and checks for
 * interruption are often infrequent, an implementation can favor responding
 * to an interrupt over normal method return. This is true even if it can be
 * shown that the interrupt occurred after another action may have unblocked
 * the thread. An implementation should document this behavior.
 * 中断通常意味着（implies）取消，并且中断检查通常很少发生，因此实现类可以偏向于响应中断而不是正常的方法返回。
 * 即使可以证明 中断发生在 另一个动作已经解除了本线程阻塞之后，也是如此。实现类中应该记录下该行为
 *
 * @see ReentrantLock
 * @see Condition
 * @see ReadWriteLock
 *
 * @since 1.5
 * @author Doug Lea
 */
public interface Lock {

    /**
     * Acquires the lock.
     *
     * <p>If the lock is not available then the current thread becomes
     * disabled for thread scheduling purposes and lies dormant until the
     * lock has been acquired.
     * 如果不能获取到锁，那么当前线程将被禁止用于线程调度目的并处于（lie）休眠（dormant蛰伏）状态，直到能够获取到锁
     *
     * <p><b>Implementation Considerations</b>
     * 实现注意事项
     *
```

```
     * <p>A {@code Lock} implementation may be able to detect erroneous use
     * of the lock, such as an invocation that would cause deadlock, and
     * may throw an (unchecked) exception in such circumstances.  The
     * circumstances and the exception type must be documented by that
     * {@code Lock} implementation.
     * Lock的实现可能需要能够检测到锁的错误使用，比如导致死锁的调用，和在这些情况（circumstan
ces）下可能抛出（未检查）异常。
     * 在Lock的实现中必须记录下这些情况和异常类型。
     */
    void lock();

    /**
     * Acquires the lock unless the current thread is
     * {@linkplain Thread#interrupt interrupted}.
     * 除非当前线程被中断（interrupted），否则获取锁
     *
     * <p>Acquires the lock if it is available and returns immediately.
     * 如果可用，则获取锁并立即返回。
     *
     * <p>If the lock is not available then the current thread becomes
     * disabled for thread scheduling purposes and lies dormant until
     * one of two things happens:
     * 如果锁不可用，那么当前线程将被禁止用户线程调度目的并处于休眠状态，直到以下两种情况之一发
生：（这两种情况就是1.要么获取到了锁，2.要么在获取锁的时候被中断了，而正好支持中断这个获取锁操
作）
     *
     * <ul>
     * <li>The lock is acquired by the current thread; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts} the
     * current thread, and interruption of lock acquisition is supported.
     * </ul>
     * 1.锁被当前线程获取；
     * 2.或者其他的一些线程中断了当前线程，并且支持在获取锁的过程中被中断。
     *
     * 下面解释如果获取锁过程中被中断会发生什么
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while acquiring the
     * lock, and interruption of lock acquisition is supported,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     * 如果当前线程：
     * 1. 在进入这个方法时设置了中断状态
     * 2. 或者在获取锁时被中断，并且支持获取锁的过程被中断。
     * 那么将抛出InterruptedException，并且清除当前线程中断状态
     *
     * <p><b>Implementation Considerations</b>
```

```
     * 实现该方法的注意事项
     *
     * <p>The ability to interrupt a lock acquisition in some
     * implementations may not be possible, and if possible may be an
     * expensive operation.  The programmer should be aware that this
     * may be the case. An implementation should document when this is
     * the case.
     * 中断锁的获取的能力在一些实现中是不可能的，如果可能，也会是个昂贵的操作（比较费时等）。
     * 程序员应当意识到这是可能的情况。
     * 实现应当记录下这些情况。
     *
     * <p>An implementation can favor responding to an interrupt over
     * normal method return.
     * 实现可以倾向于响应中断而不是正常的方法返回。
     *
     * <p>A {@code Lock} implementation may be able to detect
     * erroneous use of the lock, such as an invocation that would
     * cause deadlock, and may throw an (unchecked) exception in such
     * circumstances.  The circumstances and the exception type must
     * be documented by that {@code Lock} implementation.
     * （跟上面一样，记得检测可能会发生的死锁等错误使用锁的情况，并进行记录）
     *
     * @throws InterruptedException if the current thread is
     *         interrupted while acquiring the lock (and interruption
     *         of lock acquisition is supported)
     * 抛出InterruptedException，如果当前线程在获取锁的时候被中断
     */
    void lockInterruptibly() throws InterruptedException;

    /**
     * Acquires the lock only if it is free at the time of invocation.
     * 只有在调用时，当前锁空闲才会获取到锁（跟上面两种方式不一样的是，如果在调用时锁被占用了，
那么当前线程不会进入休眠，而是立即返回）
     *
     * <p>Acquires the lock if it is available and returns immediately
     * with the value {@code true}.
     * If the lock is not available then this method will return
     * immediately with the value {@code false}.
     * 如果可用那么会获取锁（加锁）并立即返回true
     * 如果锁不可用，该方法会立即返回false（没有休眠）
     *
     * <p>A typical usage idiom for this method would be: （typical usage idiom典型使用
习惯）
     *  <pre> {@code
     * Lock lock = ...;
     * if (lock.tryLock()) {
     *   try {
     *     // manipulate protected state
     *   } finally {
```

```
     *     lock.unlock(); // tryLock()成功也会加锁，记得显式释放锁
     *   }
     * } else {
     *   // perform alternative actions // 执行替代操作
     * }}</pre>
     *
     * This usage ensures that the lock is unlocked if it was acquired, and
     * doesn't try to unlock if the lock was not acquired.
     * 这种用法确保在获取到锁后正确释放锁，并且不会在获取锁失败后尝试释放锁。
     *
     * @return {@code true} if the lock was acquired and
     *         {@code false} otherwise
     * 这个不会抛出异常
     */
    boolean tryLock();

    /**
     * Acquires the lock if it is free within the given waiting time and the
     * current thread has not been {@linkplain Thread#interrupt interrupted}.
     * 在给定的等待时间里，如果锁空闲并且当前线程没有被中断，则会获取锁（加锁）。
     *
     * <p>If the lock is available this method returns immediately
     * with the value {@code true}.
     * If the lock is not available then
     * the current thread becomes disabled for thread scheduling
     * purposes and lies dormant until one of three things happens:
     * <ul>
     * <li>The lock is acquired by the current thread; or
     * <li>Some other thread {@linkplain Thread#interrupt interrupts} the
     * current thread, and interruption of lock acquisition is supported; or
     * <li>The specified waiting time elapses
     * </ul>
     * 如果能够获取锁，该方法立即返回true
     * 如果锁不可用，那么当前线程将禁止作为线程调用目标并进入休眠，直到以下三种情况之一发生：
     * 1. 当前线程获取到该锁
     * 2. 其他一些线程中断当前线程，并且支持中断获取锁操作
     * 3. 到达了等待时间（等待超时）
     *
     * 下面会针对上面说的三种情况进行说明
     * <p>If the lock is acquired then the value {@code true} is returned.
     * 如果成功获取到锁，会返回true
     *
     * <p>If the current thread:
     * <ul>
     * <li>has its interrupted status set on entry to this method; or
     * <li>is {@linkplain Thread#interrupt interrupted} while acquiring
     * the lock, and interruption of lock acquisition is supported,
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
```

```
     * interrupted status is cleared.
     * 如果当前线程：
     * 1. 在进入这个方法时设置了中断状态
     * 2. 或者在获取锁时被中断，并且支持中断获取锁的操作
     * 那么就会抛出InterruptedException异常，并且当前线程清理中断状态
     *
     * <p>If the specified waiting time elapses then the value {@code false}
     * is returned.
     * If the time is
     * less than or equal to zero, the method will not wait at all.
     * 如果等待时间过去了，那么会返回false
     * 注意：如果等待时间参数<=0，那么该方法根本不会等待（可以传入<=0的等待时间，但是不会等待
)
     *
     * <p><b>Implementation Considerations</b>
     * 注意事项
     * <p>The ability to interrupt a lock acquisition in some implementations
     * may not be possible, and if possible may
     * be an expensive operation.
     * The programmer should be aware that this may be the case. An
     * implementation should document when this is the case.
     * 与lockInterruptibly()的注意事项一样：
     * 中断锁的获取的能力在一些实现中是不可能的，如果可能，也会是个昂贵的操作（比较费时等）。
     * 程序员应当意识到这是可能的情况。
     * 实现应当记录下这些情况。
     *
     * <p>An implementation can favor responding to an interrupt over normal
     * method return, or reporting a timeout.
     * 更倾向于抛出异常而不是正常的方法返回，或者报告超时
     *
     * <p>A {@code Lock} implementation may be able to detect
     * erroneous use of the lock, such as an invocation that would cause
     * deadlock, and may throw an (unchecked) exception in such circumstances.
     * The circumstances and the exception type must be documented by that
     * {@code Lock} implementation.
     * 同样需要检测死锁等错误使用的情况，并且记录下来
     *
     * @param time the maximum time to wait for the lock // 最大等待时间
     * @param unit the time unit of the {@code time} argument // 等待时间的时间单位
     * @return {@code true} if the lock was acquired and {@code false}
     *         if the waiting time elapsed before the lock was acquired
     *
     * @throws InterruptedException if the current thread is interrupted
     *         while acquiring the lock (and interruption of lock
     *         acquisition is supported)
     * 这个方法如果响应中断也会抛出异常
     */
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
```

```
    /**
     * Releases the lock.
     * 释放锁（解锁）
     *
     * <p><b>Implementation Considerations</b>
     * 注意事项
     * <p>A {@code Lock} implementation will usually impose
     * restrictions on which thread can release a lock (typically only the
     * holder of the lock can release it) and may throw
     * an (unchecked) exception if the restriction is violated.
     * Any restrictions and the exception
     * type must be documented by that {@code Lock} implementation.
     * 通常会对可以释放锁的线程加以限制（restrictions）（通常只有锁的持有者（持有该锁的线程）才可以释放它），如果违反限制可能会抛出（未检查）的异常
     * 必须记录下任何限制与异常情况
     */
    void unlock();

    /**
     * Returns a new {@link Condition} instance that is bound to this
     * {@code Lock} instance.
     * 返回绑定到此Lock实例的新Condition实例
     * 这个在AQS中有使用到，到那里在进行详细观察
     *
     * <p>Before waiting on the condition the lock must be held by the
     * current thread.
     * A call to {@link Condition#await()} will atomically release the lock
     * before waiting and re-acquire the lock before the wait returns.
     * 在等待条件之前，必须由当前线程来持有锁。
     * 调用wait()将在等待之前自动释放锁，并且在等待返回之前重新获取锁
     * （这个很关键，当前线程可以调用wait()来主动释放锁并休眠，直到等待结束（比如等待的条件满足了）当前线程会重新获取到锁来继续执行。
     *
     * <p><b>Implementation Considerations</b>
     * 注意事项
     * <p>The exact operation of the {@link Condition} instance depends on
     * the {@code Lock} implementation and must be documented by that
     * implementation.
     * Condition实例的具体操作取决于Lock实现，并且必须由实现记录下来。
     *
     * @return A new {@link Condition} instance for this {@code Lock} instance
     * @throws UnsupportedOperationException if this {@code Lock}
     *         implementation does not support conditions
     * 如果Lock实现不支持conditions，会返回UnsupporttedOperationException异常
     */
    Condition newCondition();
}
```

ReentrantLock

```
/*
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */

/*
 *
 *
 *
 *
 *
 *
 * Written by Doug Lea with assistance from members of JCP JSR-166
 * Expert Group and released to the public domain, as explained at
 * http://creativecommons.org/publicdomain/zero/1.0/
 */

package java.util.concurrent.locks;
import java.util.concurrent.TimeUnit;
import java.util.Collection;

/**
 * A reentrant mutual exclusion {@link Lock} with the same basic
 * behavior and semantics as the implicit monitor lock accessed using
 * {@code synchronized} methods and statements, but with extended
 * capabilities.
```

```
 * 可重入的互斥锁与使用synchronized方法或语句访问的隐式监视器锁具有相同的行为和语义，不过提供了扩展功能
 *
 * <p>A {@code ReentrantLock} is <em>owned</em> by the thread last
 * successfully locking, but not yet unlocking it. A thread invoking
 * {@code lock} will return, successfully acquiring the lock, when
 * the lock is not owned by another thread. The method will return
 * immediately if the current thread already owns the lock. This can
 * be checked using methods {@link #isHeldByCurrentThread}, and {@link
 * #getHoldCount}.
 * ReentrantLock属于上次成功加锁并且还没解锁的线程。当lock不属于其他线程时，调用Lock方法的线程会返回并成功加锁。
 * 如果该线程已拥有该锁，那么Lock方法会立即返回。可以调用isHeldByCurrentThread与getHoldCount方法来检查锁的相关信息。
 *
 * <p>The constructor for this class accepts an optional
 * <em>fairness</em> parameter.  When set {@code true}, under
 * contention, locks favor granting access to the longest-waiting
 * thread.  Otherwise this lock does not guarantee any particular
 * access order.  Programs using fair locks accessed by many threads
 * may display lower overall throughput (i.e., are slower; often much
 * slower) than those using the default setting, but have smaller
 * variances in times to obtain locks and guarantee lack of
 * starvation. Note however, that fairness of locks does not guarantee
 * fairness of thread scheduling. Thus, one of many threads using a
 * fair lock may obtain it multiple times in succession while other
 * active threads are not progressing and not currently holding the
 * lock.
 * Also note that the untimed {@link #tryLock()} method does not
 * honor the fairness setting. It will succeed if the lock
 * is available even if other threads are waiting.
 * ReetrantLock的构造方法可以接收一个公平参数，如果设置该参数为true，在有竞争的情况下，lock偏向于授予等待时间最长的线程。否则这个锁不能保证任何的特定的访问顺序。
 * 通过多线程访问使用公平锁的程序，相较于使用默认（即非公平锁）设置，会显现出较低的吞吐量（即更慢，通常慢很多），但是在获取锁与保证不出现饥饿的时间上具有较小的差异。
 * 然而需要注意的是，锁的公平性并不保证线程调度的公平性。因此，使用公平锁的许多线程中的某一个会连续多次的获得锁（不断重入？），其他active线程并没有进行和持有该锁
 *
 * 另外需要注意的是，无时间的tryLock()方法没有公平性的设置，即使有其他线程在等待，当锁可用时也会返回true
 *
 * <p>It is recommended practice to <em>always</em> immediately
 * follow a call to {@code lock} with a {@code try} block, most
 * typically in a before/after construction such as:
 * 推荐的做法是“总是”在使用lock后马上调用try方法块，最常见的做法是在构造函数之前/之后
 *
 *  <pre> {@code
 * class X {
 *   private final ReentrantLock lock = new ReentrantLock();
```

```
 *   // ...
 *
 *   public void m() {
 *     lock.lock();  // block until condition holds
 *     try {
 *       // ... method body
 *     } finally {
 *       lock.unlock() //  因为不能自己释放锁，所以要确保try final手动释放锁
 *     }
 *   }
 * }}</pre>
 *
 * <p>In addition to implementing the {@link Lock} interface, this
 * class defines a number of {@code public} and {@code protected}
 * methods for inspecting the state of the lock.  Some of these
 * methods are only useful for instrumentation and monitoring.
 * 除了实现Lock接口，该class还定义了一些public和protected方法来检查锁的状态，其中的一些方法只对仪表盘与监控有用
 *
 * <p>Serialization of this class behaves in the same way as built-in
 * locks: a deserialized lock is in the unlocked state, regardless of
 * its state when serialized.
 * 此类的序列化与内置锁的行为方式一致；反序列化的锁处于解锁状态，无论其序列化时的状态如何。？？？
 *
 * <p>This lock supports a maximum of 2147483647 recursive locks by
 * the same thread. Attempts to exceed this limit result in
 * {@link Error} throws from locking methods.
 * 该锁最大支持同一个线程建立2147483647个递归锁，尝试超过此限制会导致抛出Error异常（通过递归的方式重入2^31 - 1次）
 *
 * @since 1.5
 * @author Doug Lea
 */
public class ReentrantLock implements Lock, java.io.Serializable {
    private static final long serialVersionUID = 7373984872572414699L;
    /** Synchronizer providing all implementation mechanics */
    // 提供所有实现机制的同步器
    private final Sync sync; // 用Sync来实现锁，它包含子类公平/非公平锁类的实现，与AQS的默认实现

    /**
     * Base of synchronization control for this lock. Subclassed
     * into fair and nonfair versions below. Uses AQS state to
     * represent the number of holds on the lock.
     * ReentrantLock的同步控制基础（根基）。可实现公平锁与非公平锁版本。使用AQS的state值来表示锁的持有次数。
     */
    abstract static class Sync extends AbstractQueuedSynchronizer {
```

```
        private static final long serialVersionUID = -5179523762034025860L;

        /**
         * Performs {@link Lock#lock}. The main reason for subclassing
         * is to allow fast path for nonfair version.
         * lock方法。子类化（abstract）的主要原因是为了快速支持非公平版本。
         */
        abstract void lock();

        /**
         * Performs non-fair tryLock.  tryAcquire is implemented in
         * subclasses, but both need nonfair try for trylock method.
         * 执行非公平的tryLock。
         * tryAcquire是在子类中实现，但是nofaireTryAcquire与tryAcquire都需要对tryLock方法进行非公平尝试。
         */
        final boolean nonfairTryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) { // 当前AQS没有被占用
                if (compareAndSetState(0, acquires)) { // 设置state状态，这一步很关键，如果多线程同时调用该tryAcquire方法，只有一个线程能CAS成功，成为锁的拥有者，其他线程只能失败
                    setExclusiveOwnerThread(current); // 调用AQS继承的AbstarctOwnableSynchronizer类的方法，设置独占锁的当前拥有者
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) { // state!=0的情况下，如果独占锁当前的拥有者是本线程，表示可重入
                int nextc = c + acquires; // 计算AQS state的新值
                if (nextc < 0) // overflow // 越界了
                    throw new Error("Maximum lock count exceeded");
                setState(nextc); // 设置AQS的state新值，通过volatile保证的内存可见性
                return true;
            }
            return false; // 即不能占有锁，也不能重入，返回false
        }

        // 尝试释放（正常情况下，调用该方法的线程一定为独占锁的拥有者）
        protected final boolean tryRelease(int releases) {
            int c = getState() - releases; // 计算AQS state的新值
            if (Thread.currentThread() != getExclusiveOwnerThread()) // 如果当前线程不是独占锁的拥有者，说明调用的有问题，直接抛出异常
                throw new IllegalMonitorStateException();
            boolean free = false; // free表示该线程是否已完全释放锁（重入次数=0）
            if (c == 0) {
                free = true;
                setExclusiveOwnerThread(null); // 当前线程已完全释放锁，设置独占锁的拥有者为null
```

```
            }
            setState(c); // 设置AQS state的新值
            return free; // 如果该线程完全释放锁，返回true，否则返回false
        }

        protected final boolean isHeldExclusively() {
            // While we must in general read state before owner,
            // we don't need to do so to check if current thread is owner
            // 虽然通常来说，我们在成为（独占锁）拥有者之前一定要读state值，但如果要判断当前线程是否为拥有者的时候，不必那么做。直接调用该方法就行。
            return getExclusiveOwnerThread() == Thread.currentThread();
        }

        // 没实现Lock接口，但是声明了一个final方法，直接生成一个AQS里面的ConditionObject对象。

        final ConditionObject newCondition() {
            return new ConditionObject();
        }

        // Methods relayed from outer class
        // 从外部类继承的方法？还是说这些方法是给外部类用的

        final Thread getOwner() {
            return getState() == 0 ? null : getExclusiveOwnerThread(); // 先判断state再判断exclusiveOwner（exclusiveOwnerThread只保证记录最后一个加锁成功的线程，不保证没有owner时该字段为null）
        }

        final int getHoldCount() {
            return isHeldExclusively() ? getState() : 0;
        }

        final boolean isLocked() {
            return getState() != 0; // 通过state判断是否有加锁
        }

        /**
         * Reconstitutes the instance from a stream (that is, deserializes it).
         * 从流中重构实例（即进行反序列化）
         */
        private void readObject(java.io.ObjectInputStream s)
            throws java.io.IOException, ClassNotFoundException {
            s.defaultReadObject();
            setState(0); // reset to unlocked state
        }
    }

    /**
     * Sync object for non-fair locks
```

```
     * 非公平锁
     */
    static final class NonfairSync extends Sync {
        private static final long serialVersionUID = 7316153563782823691L;

        /**
         * Performs lock.  Try immediate barge, backing up to normal
         * acquire on failure.
         * 加锁。试图立即抢占（barge），在失败时退回到正常获取。
         *
         */
        final void lock() {
            if (compareAndSetState(0, 1))                          // 1、如果state=0，表明现在没有线程抢占到锁，开始CAS抢占。这个方法是AQS里写的，直接设置state值
                setExclusiveOwnerThread(Thread.currentThread());  // 2、如果抢占成功，设置当前独占锁的拥有者为自己
            else
                acquire(1);                                        // 3、如果抢占失败，调用Sync继承的AQS里acquire方法（开始调用NonfaireSync（本类）实现的tryAcquire方法，如果加锁失败，将线程入AQS的sync队列等待）
        }

        protected final boolean tryAcquire(int acquires) {     // 子类实现的AQS里独占模式需要重写的3种方法之一：tryAcquire。（剩下的两种是继承的Sync类里实现了：tryRelease与isHeldExclusively）
            return nonfairTryAcquire(acquires);                // 调用的Sync实现的方法

        }
    }

    /**
     * Sync object for fair locks
     * 公平锁
     */
    static final class FairSync extends Sync {
        private static final long serialVersionUID = -3000897897090466540L;

        final void lock() {
            acquire(1);                                        // 直接调用AQS里的方法（尝试tryAcquire，如果加锁失败，入AQS的sync队列）
        }

        /**
         * Fair version of tryAcquire.  Don't grant access unless
         * recursive call or no waiters or is first.
         * tryAcquire的公平版本。
         * 除非是递归调用（重入）或者是sync等待队列的第一个waiter，否则不授予访问权限
         */
        protected final boolean tryAcquire(int acquires) {
```

```
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {                                    // 1、分析当前state状态，
如果为0，包含两种情况：1、sync队列没有等待线程；2、sync的head在锁释放后还没来得及竞争加锁。
                if (!hasQueuedPredecessors() &&              // 2、分析当前线程之前有没
有在sync队列里等待的线程（调用的AQS里实现的方法，有的话返回true，没有返回false）
                    compareAndSetState(0, acquires)) {       // 3、如果该线程前面没有等
待的线程，CAS来设置state值以抢占锁（这时候也可能有其他线程也在抢占锁）
                    setExclusiveOwnerThread(current);        // 4、抢占成功，设置当前独
占锁的拥有者为当前线程
                    return true;                            // 5、返回true，抢占成功（
不用排队了）
                }
            }
            else if (current == getExclusiveOwnerThread()) {   // 6、如果当前state状态不
为0，再分析当前独占锁的拥有者是否为当前线程
                int nextc = c + acquires;                    // 7、如果当前线程是拥有者
，将state值再次增加，表示重入
                if (nextc < 0)                               // 8、如果递归重入次数过多
(大于2^32 - 1移除)，抛出异常
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);                              // 9、设置state值
                return true;                                 // 10、返回true，重入成功
            }
            return false;                                    // 11、返回false，抢占/重
入失败，需要排队了
        }
    }

    /**
     * Creates an instance of {@code ReentrantLock}.
     * This is equivalent to using {@code ReentrantLock(false)}.
     * 创建ReentrantLock实例（默认是非公平锁实现）
     * 与使用ReentrantLock(false)是等价的
     */
    public ReentrantLock() {
        sync = new NonfairSync();
    }

    /**
     * Creates an instance of {@code ReentrantLock} with the
     * given fairness policy.
     * 创建ReentrantLock实例，使用给定的公平策略。
     *
     * @param fair {@code true} if this lock should use a fair ordering policy
     *             true表示使用公平锁
     */
    public ReentrantLock(boolean fair) {
        sync = fair ? new FairSync() : new NonfairSync(); // 用sync来
```

```
    }


    // 下面这些方法是对Lock接口的实现

    /**
     * Acquires the lock.
     * 获取锁
     *
     * <p>Acquires the lock if it is not held by another thread and returns
     * immediately, setting the lock hold count to one.
     * 如果没有被其他线程持有，则获取锁并立即返回，设置锁的持有数量为1.
     *
     * <p>If the current thread already holds the lock then the hold
     * count is incremented by one and the method returns immediately.
     * 如果当前线程已经持有该锁，那么锁的持有数量加1，并且方法立即返回
     *
     * <p>If the lock is held by another thread then the
     * current thread becomes disabled for thread scheduling
     * purposes and lies dormant until the lock has been acquired,
     * at which time the lock hold count is set to one.
     * 如果该锁已被其他线程持有，那么当前线程被禁止用于线程调度并且挂起（休眠），直到锁能够获取
，到那时设置锁的持有数为1
     *
     */
    public void lock() {
        sync.lock();
    }

    /**
     * Acquires the lock unless the current thread is
     * {@linkplain Thread#interrupt interrupted}.
     * 获取锁，除非当前线程被中断。
     *
     * <p>Acquires the lock if it is not held by another thread and returns
     * immediately, setting the lock hold count to one.
     * 如果锁没有被其他线程持有，加锁并且立即返回，设置锁持有数为1。
     *
     * <p>If the current thread already holds this lock then the hold count
     * is incremented by one and the method returns immediately.
     * 如果当前线程已经持有该锁，那么将持有数加1，并且立即返回。
     *
     * <p>If the lock is held by another thread then the
     * current thread becomes disabled for thread scheduling
     * purposes and lies dormant until one of two things happens:
     * 如果该锁已被其他线程获取，那么当前线程对于线程调度不可用并且挂起（休眠）直到以下两个事件
发生任意一个：
     *
     * <ul>
```

```
     *
     * <li>The lock is acquired by the current thread; or
     * 线程成功获取到锁；
     *
     * <li>Some other thread {@linkplain Thread#interrupt interrupts} the
     * current thread.
     * 其他线程中断了当前线程。
     *
     * </ul>
     *
     * <p>If the lock is acquired by the current thread then the lock hold
     * count is set to one.
     * 如果锁被当前线程成功获取，那么锁的持有数设置为1。
     *
     * <p>If the current thread:
     * 如果当前线程：
     * <ul>
     *
     * <li>has its interrupted status set on entry to this method; or
     * 在进入该方法前被设置了中断状态；
     *
     * <li>is {@linkplain Thread#interrupt interrupted} while acquiring
     * the lock,
     * 在获取锁的过程中被中断，
     *
     * </ul>
     *
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     * 那么抛出中断异常，并且清除当前线程的中断状态（线程的中断status由“中断”变为“无中断”）
     *
     * <p>In this implementation, as this method is an explicit
     * interruption point, preference is given to responding to the
     * interrupt over normal or reentrant acquisition of the lock.
     * 在此实现中，这个方法是个明显的中断点，所以优先（preference）响应中断而不是正常或者可重入的获取锁
     *
     * @throws InterruptedException if the current thread is interrupted
     */
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly(1); // 调用的AQS方法，如果进来时已经被中断，直接抛出异常，否则就调用AQS的doAcquireInterruptibly方法去排队竞争锁，如果排队过程中有中断，也抛出异常并退出等待
    }

    /**
     * Acquires the lock only if it is not held by another thread at the time
     * of invocation.
     * 获取锁，只有在调用时没有被其他线程持有该锁。
```

```
     *
     * <p>Acquires the lock if it is not held by another thread and
     * returns immediately with the value {@code true}, setting the
     * lock hold count to one. Even when this lock has been set to use a
     * fair ordering policy, a call to {@code tryLock()} <em>will</em>
     * immediately acquire the lock if it is available, whether or not
     * other threads are currently waiting for the lock.
     * This &quot;barging&quot; behavior can be useful in certain
     * circumstances, even though it breaks fairness. If you want to honor
     * the fairness setting for this lock, then use
     * {@link #tryLock(long, TimeUnit) tryLock(0, TimeUnit.SECONDS) }
     * which is almost equivalent (it also detects interruption).
     * 如果没有被其他线程持有则获取锁，并立即返回true，设置锁持有数为1。
     * 即使设置使用公平策略加锁，在锁可用的情况下，调用tryLock将立即获取锁，不管是否有其他线程正在sync队列上等待锁。
     * 抢占在某些（certain）情况（circumstances）下是有用的，即使破坏了公平性。
     * 如果想保持公平性，那么使用tryLock(long, TimeUnit)、tryLock(0, TimeUnit.SECONDS)，这俩几乎是等效的。
     *
     * <p>If the current thread already holds this lock then the hold
     * count is incremented by one and the method returns {@code true}.
     * 如果当前线程已经持有锁了，那么持有数加1，并且返回true。
     *
     * <p>If the lock is held by another thread then this method will return
     * immediately with the value {@code false}.
     * 如果锁已经被其他线程持有，那么该方法将立即返回false。
     *
     * @return {@code true} if the lock was free and was acquired by the
     *         current thread, or the lock was already held by the current
     *         thread; and {@code false} otherwise
     */
    public boolean tryLock() {
        return sync.nonfairTryAcquire(1); // 不管公平不公平，直接调用Sync的nonfairTryAcquire方法，试图抢占锁，加锁失败返回false。
    }

    /**
     * Acquires the lock if it is not held by another thread within the given
     * waiting time and the current thread has not been
     * {@linkplain Thread#interrupt interrupted}.
     * 获取锁，如果在等待时间内没有被其他线程持有（在等待时间内有时间点锁有空闲），并且当前线程没有被中断。
     *
     * <p>Acquires the lock if it is not held by another thread and returns
     * immediately with the value {@code true}, setting the lock hold count
     * to one. If this lock has been set to use a fair ordering policy then
     * an available lock <em>will not</em> be acquired if any other threads
     * are waiting for the lock. This is in contrast to the {@link #tryLock()}
     * method. If you want a timed {@code tryLock} that does permit barging on
```

```
     * a fair lock then combine the timed and un-timed forms together:
     * 获取锁，如果在等待时间内没有被其他线程持有，将立即返回true，设置锁的持有数为1。
     * 如果设置使用公平策略加锁，有线程在排队等锁的话，那么即使锁可用也不会成功获取锁。
     * 这与tryLock()方法相反（contrast 比对）。
     * 如果在公平锁下，既想用限时的tryLock()，又想允许抢占，那么联合使用限时与不限时的形式：
     *
     *  <pre> {@code
     * if (lock.tryLock() ||
     *     lock.tryLock(timeout, unit)) { // 先试试抢占，如果抢不到就在等待时间内尝试
     *   ...
     * }}</pre>
     *
     * <p>If the current thread
     * already holds this lock then the hold count is incremented by one and
     * the method returns {@code true}.
     * 如果当前线程已经持有锁，那么持有数加1，并且方法返回true。
     *
     * <p>If the lock is held by another thread then the
     * current thread becomes disabled for thread scheduling
     * purposes and lies dormant until one of three things happens:
     * 如果该锁已被其他线程持有，那么当前线程对线程调度不可用并且挂起（休眠），直到以下三种情况任意一种发生：
     *
     * <ul>
     *
     * <li>The lock is acquired by the current thread; or
     *
     * <li>Some other thread {@linkplain Thread#interrupt interrupts}
     * the current thread; or
     *
     * <li>The specified waiting time elapses
     * 1、当前线程获取到锁；
     * 2、当前线程被其他线程中断；
     * 3、指定的等待时间超时；
     *
     * </ul>
     *
     * <p>If the lock is acquired then the value {@code true} is returned and
     * the lock hold count is set to one.
     *
     * <p>If the current thread:
     *
     * <ul>
     *
     * <li>has its interrupted status set on entry to this method; or
     *
     * <li>is {@linkplain Thread#interrupt interrupted} while
     * acquiring the lock,
     *
```

```
     * </ul>
     * then {@link InterruptedException} is thrown and the current thread's
     * interrupted status is cleared.
     * 如果当前线程：
     * 1、在进入该方法之前已被中断；
     * 2、在等待获取锁时被中断；
     * 那么将抛出InterruptedException，并且当前线程的interrupted status清空。
     *
     * <p>If the specified waiting time elapses then the value {@code false}
     * is returned.  If the time is less than or equal to zero, the method
     * will not wait at all.
     * 如果指定等待时间超时，那么将返回false。
     * 如果指定的等待时间<=0，那么该方法将不会等待。
     *
     * <p>In this implementation, as this method is an explicit
     * interruption point, preference is given to responding to the
     * interrupt over normal or reentrant acquisition of the lock, and
     * over reporting the elapse of the waiting time.
     * 优先响应中断，而不是正常或者重入的获取锁，也不是响应已超时时间。
     *
     * @param timeout the time to wait for the lock
     * @param unit the time unit of the timeout argument
     * @return {@code true} if the lock was free and was acquired by the
     *         current thread, or the lock was already held by the current
     *         thread; and {@code false} if the waiting time elapsed before
     *         the lock could be acquired
     * @throws InterruptedException if the current thread is interrupted
     * @throws NullPointerException if the time unit is null
     */
    public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
        return sync.tryAcquireNanos(1, unit.toNanos(timeout)); // AQS的tryAcquireNanos，先尝试本类(sync)实现的tryAcquire，如果失败，则调用AQS的doAcquireNanos，将该线程入队，等待获取锁。
    }

    /**
     * Attempts to release this lock.
     * 释放锁
     *
     * <p>If the current thread is the holder of this lock then the hold
     * count is decremented.  If the hold count is now zero then the lock
     * is released.  If the current thread is not the holder of this
     * lock then {@link IllegalMonitorStateException} is thrown.
     * 如果当前线程持有该锁，那么持有数递减。
     * 如果锁的持有数成了0，那么当前锁被释放。
     * 如果当前线程不是该锁的持有者，那么抛出IllegalMonitorStateException（大概是非法修改监视器状态异常）
     *
```

```
     * @throws IllegalMonitorStateException if the current thread does not
     *         hold this lock
     */
    public void unlock() {
        sync.release(1); // 调用AQS实现的release方法，先调用本类（sync）实现的tryRelease，如果可以，从等待队列的head处unpark一个后继。
    }

    /**
     * Returns a {@link Condition} instance for use with this
     * {@link Lock} instance.
     * 返回一个与该Lock实例一起使用的Condition实例。
     *
     * <p>The returned {@link Condition} instance supports the same
     * usages as do the {@link Object} monitor methods ({@link
     * Object#wait() wait}, {@link Object#notify notify}, and {@link
     * Object#notifyAll notifyAll}) when used with the built-in
     * monitor lock.
     * 当与内置的监视器锁一起使用时，返回的Condition实例支持与Object监视器相同的使用方法，例如wait、notify和notifyAll。
     *
     * <ul>
     *
     * <li>If this lock is not held when any of the {@link Condition}
     * {@linkplain Condition#await() waiting} or {@linkplain
     * Condition#signal signalling} methods are called, then an {@link
     * IllegalMonitorStateException} is thrown.
     * 如果在调用Condition的waiting、signalling等方法时，关联的该锁没有被当前线程持有，那么将抛出IllegalMonitorStateException。
     *
     * <li>When the condition {@linkplain Condition#await() waiting}
     * methods are called the lock is released and, before they
     * return, the lock is reacquired and the lock hold count restored
     * to what it was when the method was called.
     * 当调用Condition的waiting方法时，持有的锁将释放。
     * 在该方法返回时，将重新持有锁，并且将锁的持有数恢复到调用该方法时的状态。
     * （锁的释放与重新获取）
     *
     * <li>If a thread is {@linkplain Thread#interrupt interrupted}
     * while waiting then the wait will terminate, an {@link
     * InterruptedException} will be thrown, and the thread's
     * interrupted status will be cleared.
     * 如果线程在等待过程中被中断，那么wait将终止，抛出中断异常，并且线程的interrupted state被清理。
     *
     * <li> Waiting threads are signalled in FIFO order.
     * 等待线程按FIFO顺序被唤醒（signalled 接收信号）
     *
     * <li>The ordering of lock reacquisition for threads returning
```

```
     * from waiting methods is the same as for threads initially
     * acquiring the lock, which is in the default case not specified,
     * but for <em>fair</em> locks favors those threads that have been
     * waiting the longest.
     * 从等待方法返回的线程重新获取锁的顺序与最初获取锁的线程（方式）相同，在默认情况下没有指定，但是对于公平锁，那些队列中等待时间最长的线程要比该线程优先。
     *
     * </ul>
     *
     * @return the Condition object
     */
    public Condition newCondition() {
        return sync.newCondition(); // 本类重写的Lock接口方法，其实是调用的内部类Sync里写的方法。
    }

    /**
     * Queries the number of holds on this lock by the current thread.
     * 查询当前线程持有该锁的持有数。
     *
     * <p>A thread has a hold on a lock for each lock action that is not
     * matched by an unlock action.
     * 线程为每一个与解锁操作不匹配的加锁操作持有一个锁（含义可能是，如果加锁了但没解锁，那么锁的数量会加1）
     *
     * <p>The hold count information is typically only used for testing and
     * debugging purposes. For example, if a certain section of code should
     * not be entered with the lock already held then we can assert that
     * fact:
     * 持有数信息通常只用于测试和debug的目的。
     * 例如，如果在已经持有锁的情况下，某段代码不应该被输入，那么我们可以断言（assert）这个事实：
     *
     *  <pre> {@code
     * class X {
     *   ReentrantLock lock = new ReentrantLock();
     *   // ...
     *   public void m() {
     *     assert lock.getHoldCount() == 0; // assert 断言，在AQS里也有，表示如果满足就继续执行，不满足就抛出异常
     *     lock.lock();
     *     try {
     *       // ... method body
     *     } finally {
     *       lock.unlock();
     *     }
     *   }
     * }}</pre>
     * 上面那段代码的意思就是，assert 如果当前线程没有持有锁，那么就加锁，否则就直接跳过加锁。
```

```
     *
     * @return the number of holds on this lock by the current thread,
     *         or zero if this lock is not held by the current thread
     *         如果当前线程没有持有该锁，返回0
     */
    public int getHoldCount() {
        return sync.getHoldCount(); // 调用Sync的，先判断当前线程是否持有锁，如果持有，返回
AQS的state值
    }

    /**
     * Queries if this lock is held by the current thread.
     * 查询当前线程是否持有该锁。
     *
     * <p>Analogous to the {@link Thread#holdsLock(Object)} method for
     * built-in monitor locks, this method is typically used for
     * debugging and testing. For example, a method that should only be
     * called while a lock is held can assert that this is the case:
     * 类似于内置监视器锁的holdsLock(Object)方法，通常只用于测试和debug的目的。
     * 例如，方法应该只在当前线程持有锁的情况下执行，可以这样使用assert：
     *
     *  <pre> {@code
     * class X {
     *   ReentrantLock lock = new ReentrantLock();
     *   // ...
     *
     *   public void m() {
     *       assert lock.isHeldByCurrentThread();
     *       // ... method body
     *   }
     * }}</pre>
     *
     * <p>It can also be used to ensure that a reentrant lock is used
     * in a non-reentrant manner, for example:
     * 也可以用于确保可重入锁以不可重入的方式使用。
     * （把可重入锁当做不可重入锁使用）
     *
     *  <pre> {@code
     * class X {
     *   ReentrantLock lock = new ReentrantLock();
     *   // ...
     *
     *   public void m() {
     *       assert !lock.isHeldByCurrentThread(); // 当前线程是否持有锁，如果持有，抛出异
常
     *       lock.lock();
     *       try {
     *           // ... method body
     *       } finally {
```

```
     *           lock.unlock();
     *       }
     *   }
     * }}</pre>
     *
     * @return {@code true} if current thread holds this lock and
     *         {@code false} otherwise
     */
    public boolean isHeldByCurrentThread() {
        return sync.isHeldExclusively(); // Sync#isHeldExclusively -> AbstractOwnerSync
hronizer#getExclusiveOwnerThread 与当前线程比较
    }

    /**
     * Queries if this lock is held by any thread. This method is
     * designed for use in monitoring of the system state,
     * not for synchronization control.
     * 查询该锁是否被线程持有（不限于当前线程）。
     * 该方法为了监控系统状态设计的，不是为了同步控制。
     *
     * @return {@code true} if any thread holds this lock and
     *         {@code false} otherwise
     */
    public boolean isLocked() {
        return sync.isLocked(); // Sync#isLocked，通过判断AQS的state值是否等于=0
    }

    /**
     * Returns {@code true} if this lock has fairness set true.
     * 使用公平策略的话，返回true。
     *
     * @return {@code true} if this lock has fairness set true
     */
    public final boolean isFair() {
        return sync instanceof FairSync;
    }

    /**
     * Returns the thread that currently owns this lock, or
     * {@code null} if not owned. When this method is called by a
     * thread that is not the owner, the return value reflects a
     * best-effort approximation of current lock status. For example,
     * the owner may be momentarily {@code null} even if there are
     * threads trying to acquire the lock but have not yet done so.
     * This method is designed to facilitate construction of
     * subclasses that provide more extensive lock monitoring
     * facilities.
     * 返回当前拥有该锁的线程，如果没有拥有者，返回null。
     * 当该方法被非拥有者调用时，返回的值尽力反映当前锁的状态。（是个近似值 approximation）
```

```
     * 例如，持有者可能某个时刻是null：有线程正在获取锁，但还没有完成。
     * 该方法是为了促进（facilitate）子类构建而设计，以提高更广泛的监控设备
     *
     * @return the owner, or {@code null} if not owned
     */
    protected Thread getOwner() {
        return sync.getOwner();
    }

    /**
     * Queries whether any threads are waiting to acquire this lock. Note that
     * because cancellations may occur at any time, a {@code true}
     * return does not guarantee that any other thread will ever
     * acquire this lock.  This method is designed primarily for use in
     * monitoring of the system state.
     * 查询是否有线程在等待获取锁。
     * 注意，因为取消（取消可能是中断或者超时）可能在任意时刻发生，返回true不保证任何其他线程将获得锁。（返回true不保证后续一定有线程会成功加锁，因为即使有排队，但排队线程可能会取消）
     * 该方法主要是为了监视系统状态设计的。
     *
     * @return {@code true} if there may be other threads waiting to
     *         acquire the lock
     */
    public final boolean hasQueuedThreads() {
        return sync.hasQueuedThreads(); // AQS的hasQueuedThreads方法，判断head != tail
    }

    /**
     * Queries whether the given thread is waiting to acquire this
     * lock. Note that because cancellations may occur at any time, a
     * {@code true} return does not guarantee that this thread
     * will ever acquire this lock.  This method is designed primarily for use
     * in monitoring of the system state.
     * 查询给定的线程是否在该锁的等待队列上。
     * 注意，因为取消可能在任意时刻发生，返回true不保证该线程将获得锁。（跟上面方法的原因一样）
     * 该方法主要是为了监视系统状态设计的。
     *
     * @param thread the thread
     * @return {@code true} if the given thread is queued waiting for this lock
     * @throws NullPointerException if the thread is null
     */
    public final boolean hasQueuedThread(Thread thread) {
        return sync.isQueued(thread); // AQS的isQueued方法，从tail向前遍历，如果找到该线程，返回true，否则false。（为什么从tail开始遍历，可以看一下AQS的addWaiter或者enq过程）
    }

    /**
     * Returns an estimate of the number of threads waiting to
```

```
     * acquire this lock.  The value is only an estimate because the number of
     * threads may change dynamically while this method traverses
     * internal data structures.  This method is designed for use in
     * monitoring of the system state, not for synchronization
     * control.
     * 返回一个预估的等待获取该锁的线程数。
     * 值只是个预估值，因为在遍历内部数据结构时，线程可能会动态改变。
     * 该方法是为了监视系统状态设计的，不是为了同步。
     *
     * @return the estimated number of threads waiting for this lock
     */
    public final int getQueueLength() {
        return sync.getQueueLength(); // 调用AQS#getQueueLength方法，从tail遍历，累加所有的节点数
    }

    /**
     * Returns a collection containing threads that may be waiting to
     * acquire this lock.  Because the actual set of threads may change
     * dynamically while constructing this result, the returned
     * collection is only a best-effort estimate.  The elements of the
     * returned collection are in no particular order.  This method is
     * designed to facilitate construction of subclasses that provide
     * more extensive monitoring facilities.
     * 返回包含可能在获取该锁等待的线程集合。
     * 由于实际上的线程集合可能在构建结果的过程中动态改变，这个集合的结果只是个尽力预估值。
     * 返回的集合中的元素没有特定顺序。
     * 该方法是为了促进子类构建而设计，以提高更广泛的监控设备。
     *
     * @return the collection of threads
     */
    protected Collection<Thread> getQueuedThreads() {
        return sync.getQueuedThreads(); // 调用AQS#getQueuedThreads
    }

    /**
     * Queries whether any threads are waiting on the given condition
     * associated with this lock. Note that because timeouts and
     * interrupts may occur at any time, a {@code true} return does
     * not guarantee that a future {@code signal} will awaken any
     * threads.  This method is designed primarily for use in
     * monitoring of the system state.
     * 查询是否有线程在给定的与该锁关联的condition上等待。
     * 注意，由于超时和中断可能在任意时刻发生，返回true不保证未来signal将唤醒任意线程。（有可能现在在condition queue上有线程在等待，但是过了一段时间这些线程取消了，再过一段时间调用signal也不会有任何线程响应了）
     * 该方法主要是为了用于监视系统状态设计的。
     *
     * @param condition the condition
```

```
     * @return {@code true} if there are any waiting threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    public boolean hasWaiters(Condition condition) {
        if (condition == null)
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject)) // 判断给定的condition是否为AQS内部实现的ConditionObject
            throw new IllegalArgumentException("not owner");
        return sync.hasWaiters((AbstractQueuedSynchronizer.ConditionObject)condition); // 调用AQS#hasWaiters(ConditionObject condition)->AQS#hasWaiters，要求必须持有该锁。从firstWaiter开始遍历，如果有ws=CONDITION的，返回true
    }

    /**
     * Returns an estimate of the number of threads waiting on the
     * given condition associated with this lock. Note that because
     * timeouts and interrupts may occur at any time, the estimate
     * serves only as an upper bound on the actual number of waiters.
     * This method is designed for use in monitoring of the system
     * state, not for synchronization control.
     * 返回预估的在给定与该锁关联的condition上等待的线程数。
     * 注意，由于超时和中断可能在任意时刻发生，预估的waiter数量>=实际的waiter数量。
     * 该方法主要是为了用于监视系统状态设计的。
     *
     * @param condition the condition
     * @return the estimated number of waiting threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    public int getWaitQueueLength(Condition condition) {
        if (condition == null)
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
            throw new IllegalArgumentException("not owner");
        return sync.getWaitQueueLength((AbstractQueuedSynchronizer.ConditionObject)condition); // 调用AQS#getWaitQueueLength，要求必须持有该锁。
    }

    /**
     * Returns a collection containing those threads that may be
     * waiting on the given condition associated with this lock.
     * Because the actual set of threads may change dynamically while
     * constructing this result, the returned collection is only a
```

```
     * best-effort estimate. The elements of the returned collection
     * are in no particular order.  This method is designed to
     * facilitate construction of subclasses that provide more
     * extensive condition monitoring facilities.
     * 返回在给定的与该锁关联的condition上等待的线程集合。
     * 由于在构造结果的过程中实际的线程集会动态变化，所以返回的集合只是一个尽力预估值。
     * 返回的集合中元素没有特定的顺序。
     * 该方法是为了促进子类构建而设计，以提高更广泛的监控设备。
     *
     * @param condition the condition
     * @return the collection of threads
     * @throws IllegalMonitorStateException if this lock is not held
     * @throws IllegalArgumentException if the given condition is
     *         not associated with this lock
     * @throws NullPointerException if the condition is null
     */
    protected Collection<Thread> getWaitingThreads(Condition condition) {
        if (condition == null)
            throw new NullPointerException();
        if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
            throw new IllegalArgumentException("not owner");
        return sync.getWaitingThreads((AbstractQueuedSynchronizer.ConditionObject)condi
tion);
    }

    /**
     * Returns a string identifying this lock, as well as its lock state.
     * The state, in brackets, includes either the String {@code "Unlocked"}
     * or the String {@code "Locked by"} followed by the
     * {@linkplain Thread#getName name} of the owning thread.
     *
     * @return a string identifying this lock, as well as its lock state
     */
    public String toString() {
        Thread o = sync.getOwner();
        return super.toString() + ((o == null) ?
                                   "[Unlocked]" :
                                   "[Locked by thread " + o.getName() + "]");
    }
}
```

学习进度

```
-- 20210721

1、线程池终于看到最后了，明天收尾了
2、hashset、concurrentHashMap咋搞


-- 20210717

1、workQueue用来存还没分配给worker的任务
2、Worker是用AQS实现的，用state控制是否可以被中断，好像没有涉及到aqs主要sync队列的部分


-- 20210713
昨天把AbstractExecutorService看完了
今天继续线程池了，结果才看了500行，还有1700+

-- 20210711
第36天
玩了两天，看完了FutureTask，准备接着看了
AbstractExecutorService依赖串太长了
ExecutorCompletionService（看完了）->completionService
                                                -> blockingQueue -> queue -> collection
 -> Iterable

明天把AbstractExecutorService收尾，可以看线程池了



-- 20210708
第33天
没想到，原来的短信接口发版了，没办法，只能加班把自己的短信上线了。
差仨方法就看完FutureTask了，明天看完，再加上Abstract那个，就开线程池了
估计明天能看完AbstracExecutorService就完成任务了


-- 20210706
第31天
脑袋疼，记忆力下降严重
本来以为今天能重看完线程池，下午的考试加上其他的事情，还是争取先看完依赖的接口类吧

-- 20210705
今天是读JUC源码的第30天吧，算第30天，没仔细记下来啥时候开始的，大概是6月7号吧。
本来只是为了看一下线程池的实现，结果发现线程池关联了很多的其他JUC接口/类，所以，就把涉及到的重点都看一下
```

今天算是个里程碑吧，看完了ReentrantLock的总体源码，加上前几天通读了一遍的AQS（读完了，但实际上还有些实现点不太吃准），基本上可重入锁这一块差不多了。
前几天看完AQS，觉得所有没超过3000行的源码都是可读的，555。

明天要把线程池的搞完，到本周结束看完AtomicReference、ReadWriteLock、ConcurrentHashMap、CopyOnWriteArrayList、CountDownLatch、DelayQueue

java.util.concurrent总目录：

```
AbstractExecutorService.java
ArrayBlockingQueue.java
BlockingDeque.java
BlockingQueue.java
BrokenBarrierException.java
Callable.java
CancellationException.java
CompletableFuture.java
CompletionException.java
CompletionService.java
CompletionStage.java
ConcurrentHashMap.java
ConcurrentLinkedDeque.java
ConcurrentLinkedQueue.java
ConcurrentMap.java
ConcurrentNavigableMap.java
ConcurrentSkipListMap.java
ConcurrentSkipListSet.java
CopyOnWriteArrayList.java
CopyOnWriteArraySet.java
CountDownLatch.java
CountedCompleter.java
CyclicBarrier.java
Delayed.java
DelayQueue.java
Exchanger.java
ExecutionException.java
Executor.java
ExecutorCompletionService.java
Executors.java
ExecutorService.java
ForkJoinPool.java
ForkJoinTask.java
ForkJoinWorkerThread.java
Future.java
FutureTask.java
LinkedBlockingDeque.java
```

```
LinkedBlockingQueue.java
LinkedTransferQueue.java
package-info.java
Phaser.java
PriorityBlockingQueue.java
RecursiveAction.java
RecursiveTask.java
RejectedExecutionException.java
RejectedExecutionHandler.java
RunnableFuture.java
RunnableScheduledFuture.java
ScheduledExecutorService.java
ScheduledFuture.java
ScheduledThreadPoolExecutor.java
Semaphore.java
SynchronousQueue.java
ThreadFactory.java
ThreadLocalRandom.java
ThreadPoolExecutor.java
TimeoutException.java
TimeUnit.java
TransferQueue.java

locks目录下:
.
..
AbstractOwnableSynchronizer.java
AbstractQueuedLongSynchronizer.java
AbstractQueuedSynchronizer.java
Condition.java
Lock.java
LockSupport.java
package-info.java
ReadWriteLock.java
ReentrantLock.java
ReentrantReadWriteLock.java
StampedLock.java

atomic目录下:
AtomicBoolean.java
AtomicInteger.java
AtomicIntegerArray.java
AtomicIntegerFieldUpdater.java
AtomicLong.java
AtomicLongArray.java
AtomicLongFieldUpdater.java
AtomicMarkableReference.java
AtomicReference.java
AtomicReferenceArray.java
```

```
AtomicReferenceFieldUpdater.java
AtomicStampedReference.java
DoubleAccumulator.java
DoubleAdder.java
LongAccumulator.java
LongAdder.java
package-info.java
Striped64.java
```