**J.U.C源码分析**

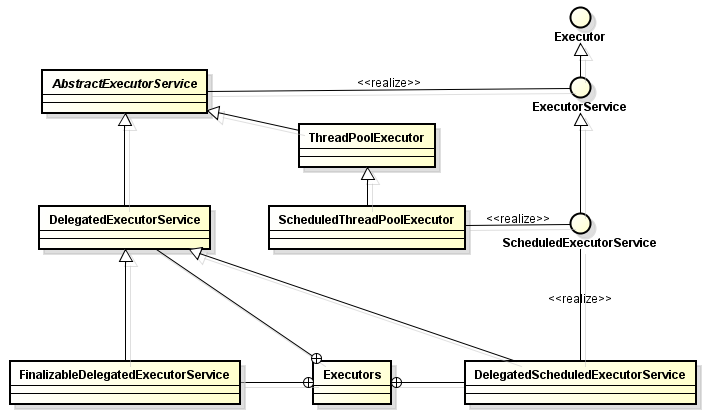
**八两俊**

**2014.12**

# Executor

**Executor** 是一个简单的标准化接口，用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。根据所使用的具体 Executor 类的不同，可能在新创建的线程中，现有的任务执行线程中，或者调用 execute() 的线程中执行任务，并且可能顺序或并发执行。**ExecutorService** 提供了多个完整的异步任务执行框架。ExecutorService 管理任务的排队和安排，并允许受控制的关闭。**ScheduledExecutorService**子接口及相关的接口添加了对延迟的和定期任务执行的支持。ExecutorService 提供了安排异步执行的方法，可执行由 [Callable](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Callable.html) 表示的任何函数，结果类似于 [Runnable](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Runnable.html)。[**Future**](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Future.html) 返回函数的结果，允许确定执行是否完成，并提供取消执行的方法。**RunnableFuture** 是拥有 run 方法的 Future，run 方法执行时将设置其结果。

实现。类 ThreadPoolExecutor 和 ScheduledThreadPoolExecutor 提供可调的、灵活的线程池。**Executors** 类提供大多数 Executor 的常见类型和配置的工厂方法，以及使用它们的几种实用工具方法。其他基于 Executor 的实用工具包括具体类 FutureTask，它提供 Future 的常见可扩展实现，以及**ExecutorCompletionService**，它有助于协调对异步任务组的处理。



Executor类图

## ExecutorService

Executor 提供了管理终止的方法，以及可为跟踪一个或多个异步任务执行状况而生成 [Future](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Future.html)的方法。

可以关闭 ExecutorService，这将导致其拒绝新任务。提供两个方法来关闭 ExecutorService。**shutdown()**方法在终止前允许执行以前提交的任务，而**shutdownNow()** 方法阻止等待任务启动并试图停止当前正在执行的任务。在终止时，执行程序没有任务在执行，也没有任务在等待执行，并且无法提交新任务。应该关闭未使用的 ExecutorService 以允许回收其资源。

通过创建并返回一个可用于取消执行和/或等待完成的 Future，方法 submit 扩展了基本方法 Executor.execute(java.lang.Runnable)。方法 invokeAny 和invokeAll 是批量执行的最常用形式，它们执行任务 collection，然后等待至少一个，或全部任务完成（可使用 ExecutorCompletionService类来编写这些方法的自定义变体）。

内存一致性效果：线程中向 ExecutorService 提交 Runnable 或 Callable 任务之前的操作 [happen-before](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\package-summary.html#MemoryVisibility) 由该任务所提取的所有操作，后者依次happen-before 通过 Future.get() 获取的结果。

## AbstractExecutorService

提供 ExecutorService 执行方法的默认实现。此类使用 newTaskFor 返回的 RunnableFuture 实现 submit、invokeAny 和 invokeAll 方法，默认情况下，[RunnableFuture](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\RunnableFuture.html) 是此包中提供的 [FutureTask](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\FutureTask.html) 类.

### submit(Runnable)

**public** Future<?> submit(Runnable task) {

**if** (task == **null**) **throw** **new** NullPointerException();

RunnableFuture<Object> ftask = newTaskFor(task, **null**);

execute(ftask);

**eturn** ftask;

}

调用newTaskFor方法将Runnable封装成RunnableFuture(采用FutureTask)后将任务的执行交给execute。

对于submit(Callable)和submit(Runnable,T)内部实现也是将Runnable/Callable封装成RunnableFuture对象，然后将任务执行交给execute。execute的具体实现由AbstractExecutorService的实现类(如ThreadPoolExecutor)实现。

**protected** <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {

**return** **new** FutureTask<T>(runnable, value);

}

**protected** <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {

**return** **new** FutureTask<T>(callable);

}

submit(Runnable)执行流程：

1、首先根据Runnable创建FutureTask，创建FutureTask其实就是创建一个[Sync](#_Sync)。在构建Sync的时候，会调用Executors. callable(Runnable task, T result)方法，此方法会创建RunnableAdapter[Executors内部类]，此类实现了Callable接口。

**static** **final** **class** RunnableAdapter<T> **implements** Callable<T> {

**final** Runnable task;

**final** T result;

RunnableAdapter(Runnable task, T result) {

**this**.task = task;

**this**.result = result;

}

**public** T call() {

task.run();

**return** result;

}

}

2、创建完FutureTask后，将FutureTask交给执行程序Executor的execute(Runnable)方法执行。处理逻辑见[execute(Runnable)](#_execute(Runnable))方法。

3、因为FutureTask也是一个Runnable，所以工作线程执行的时候会执行其run方法，执行run方其实就是执行[Sync](#_Sync)中的[innerRun()](#_innerRun())方法。在innerRunn方法中会调用Callable中的call()方法。从上面的处理来看，这里其实调用的是RunnableAdapter的call()方法，在RunnableAdapter中的call()方法中会调用Runnable的run()方法执行任务，并返回预定的结果。

### invokeAny(Collection<? extends Callable<T>> )

**public** <T> T invokeAny(Collection<? **extends** Callable<T>> tasks) **throws** InterruptedException, ExecutionException {

**try** {

**return** doInvokeAny(tasks, **false**, 0);

} **catch** (TimeoutException cannotHappen) {

**assert** **false**;

**return** **null**;

}

}

invokeAny方法所有的逻辑都问题doInvokeAny方法实现，如下：

**private** <T> T doInvokeAny(Collection<? **extends** Callable<T>> tasks, **boolean** timed, **long** nanos) **throws** InterruptedException, ExecutionException, TimeoutException {

**if** (tasks == **null**)

**throw** **new** NullPointerException();

**int** ntasks = tasks.size();

**if** (ntasks == 0)

**throw** **new** IllegalArgumentException();

List<Future<T>> futures= **new** ArrayList<Future<T>>(ntasks);

ExecutorCompletionService<T> ecs = **new** ExecutorCompletionService<T>(**this**);

**try** {

ExecutionException ee = **null**;

**long** lastTime = (timed)? System.*nanoTime*() : 0;

Iterator<? **extends** Callable<T>> it = tasks.iterator();

// Start one task for sure; the rest incrementally

futures.add(ecs.submit(it.next()));

--ntasks;

**int** active = 1;

**for** (;;) {

Future<T> f = ecs.poll();

**if** (f == **null**) {

**if** (ntasks > 0) {

--ntasks;

futures.add(ecs.submit(it.next()));

++active;

} **else** **if** (active == 0)

**break**;

**else** **if** (timed) {

f = ecs.poll(nanos, TimeUnit.*NANOSECONDS*);

**if** (f == **null**)

**throw** **new** TimeoutException();

**long** now = System.*nanoTime*();

nanos -= now - lastTime;

lastTime = now;

} **else**

f = ecs.take();

}

**if** (f != **null**) {

--active;

**try** {

**return** f.get();

} **catch** (InterruptedException ie) {

**throw** ie;

} **catch** (ExecutionException eex) {

ee = eex;

} **catch** (RuntimeException rex) {

ee = **new** ExecutionException(rex);

}

}

}

**if** (ee == **null**)

ee = **new** ExecutionException();

**throw** ee;

} **finally** {

**for** (Future<T> f : futures)

f.cancel(**true**);

}

}

1、创建ExecutorCompletionService实例，来承载Callable任务的执行工作。

2、首先将Collection第一个元素对应的Callable提取出来，调用submit加入执行。

3、采用for (;;) {}循环的方式调用ExecutorCompletionService的poll()方法获取已完成任务的Future，如果存在结果在调用Futrue.get()方法获取结果返回；否则调用ExecutorCompletionService.submit()执行任务。

4、对于设置了超时的，如果在规定时间内仍然没有完成的任务则抛出TimeoutException。对于没有设置超时的，则调用ExecutorCompletionService.take()等待一个任务完成并返回。

### invokeAll(Collection<? extends Callable<T>>)

**public** <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks) **throws** InterruptedException {

**if** (tasks == **null**)

**throw** **new** NullPointerException();

List<Future<T>> futures = **new** ArrayList<Future<T>>(tasks.size());

**boolean** done = **false**;

**try** {

**for** (Callable<T> t : tasks) {

RunnableFuture<T> f = newTaskFor(t);

futures.add(f);

execute(f);

}

**for** (Future<T> f : futures) {

**if** (!f.isDone()) { // Future.isDone()标识任务是否已完成，如果完成返回true。

**try** {// 此处表示任务未完成

f.get();// 等待计算完成,然后其结果。会阻塞线程已达到获取所有Future结果

} **catch** (CancellationException ignore) {

} **catch** (ExecutionException ignore) {

}

}

}

done = **true**;

**return** futures;

} **finally** {

**if** (!done) // 如果由于某种原因(Executor终止或异常终止)，采用Future.cacncel()取消任务执行。

**for** (Future<T> f : futures)

f.cancel(**true**);

}

}

## ThreadPoolExecutor

一个 ExecutorService，它使用可能的几个池线程之一执行每个提交的任务，通常使用 Executors 工厂方法配置。

线程池可以解决两个不同问题：由于减少了每个任务调用的开销，它们通常可以在执行大量异步任务时提供增强的性能，并且还可以提供绑定和管理资源（包括执行任务集时使用的线程）的方法。每个 ThreadPoolExecutor 还维护着一些基本的统计数据，如完成的任务数。

为了便于大量上下文使用，此类提供了很多可调整的参数和**扩展钩子 (hook)**。但是，强烈建议程序员使用较为方便的 Executors 工厂方法Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）、Executors.newFixedThreadPool(int)（固定大小线程池）和Executors.newSingleThreadExecutor()（单个后台线程），它们均为大多数使用场景预定义了设置。否则，在手动配置和调整此类时，使用以下指导：

A、核心和最大池大小

ThreadPoolExecutor 将根据corePoolSize（参见 [getCorePoolSize()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html#getCorePoolSize())）和maximumPoolSize（参见 [getMaximumPoolSize()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html#getMaximumPoolSize())）设置的边界自动调整池大小。当新任务在方法 execute(java.lang.Runnable) 中提交时，如果运行的线程少于 corePoolSize，则创建新线程来处理请求，即使其他辅助线程是空闲的。如果运行的线程多于 corePoolSize 而少于 maximumPoolSize，则仅当队列满时才创建新线程。如果设置的 corePoolSize 和 maximumPoolSize 相同，则创建了固定大小的线程池。如果将 maximumPoolSize 设置为基本的无界值（如 Integer.MAX\_VALUE），则允许池适应任意数量的并发任务。在大多数情况下，核心和最大池大小仅基于构造来设置，不过也可以使用 setCorePoolSize(int) 和 setMaximumPoolSize(int)进行动态更改。

B、按需构造

默认情况下，即使核心线程最初只是在新任务到达时才创建和启动的，也可以使用方法 prestartCoreThread() 或 prestartAllCoreThreads() 对其进行动态重写。如果构造带有非空队列的池，则可能希望预先启动线程。

C、创建新线程

使用 ThreadFactory 创建新线程。如果没有另外说明，则在同一个 ThreadGroup 中一律使用 [Executors.defaultThreadFactory()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Executors.html#defaultThreadFactory()) 创建线程，并且这些线程具有相同的 NORM\_PRIORITY 优先级和非守护进程状态。通过提供不同的 ThreadFactory，可以改变线程的名称、线程组、优先级、守护进程状态，等等。如果从 newThread 返回 null 时 ThreadFactory 未能创建线程，则执行程序将继续运行，但不能执行任何任务。

D、保持活动时间

如果池中当前有多于 corePoolSize 的线程，则这些**多出的线程**在空闲时间超过 keepAliveTime 时将会终止（参见getKeepAliveTime(java.util.concurrent.TimeUnit)）。这提供了当池处于非活动状态时减少资源消耗的方法。如果池后来变得更为活动，则可以创建新的线程。也可以使用方法 [setKeepAliveTime(long, java.util.concurrent.TimeUnit)](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html#setKeepAliveTime(long, java.util.concurrent.TimeUnit)) 动态地更改此参数。使用 Long.MAX\_VALUE TimeUnit.NANOSECONDS 的值在关闭前有效地从以前的终止状态禁用空闲线程。默认情况下，保持活动策略只在有多于 corePoolSizeThreads 的线程时应用。但是只要 keepAliveTime 值非 0，allowCoreThreadTimeOut(boolean) 方法也可将此超时策略应用于核心线程。

E、排队

所有BlockingQueue 都可用于传输和保持提交的任务。可以使用此队列与池大小进行交互：

如果运行的线程少于corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。

如果运行的线程等于或多于corePoolSize，则Executor始终首选将请求加入队列，而不添加新的线程。

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。

排队有三种通用策略：

E.1 直接提交。工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求**无界 maximumPoolSizes** 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

E.2 无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web 页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

E.3 有界队列。当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，**但是可能较难调整和控制**。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O 边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

F、被拒绝的任务

当Executor已经关闭，并且 Executor 将有限边界用于最大线程和工作队列容量，且已经饱和时，在方法 execute(java.lang.Runnable) 中提交的新任务将被拒绝。在以上两种情况下，execute 方法都将调用其**RejectedExecutionHandler** 的[RejectedExecutionHandler.rejectedExecution(java.lang.Runnable, java.util.concurrent.ThreadPoolExecutor)](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\RejectedExecutionHandler.html#rejectedExecution(java.lang.Runnable, java.util.concurrent.ThreadPoolExecutor)) 方法。下面提供了四种预定义的处理程序策略：

在默认的 ThreadPoolExecutor.AbortPolicy 中，处理程序遭到拒绝将抛出运行时 RejectedExecutionException。

在 ThreadPoolExecutor.CallerRunsPolicy 中，线程调用运行该任务的 execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。

在 ThreadPoolExecutor.DiscardPolicy 中，不能执行的任务将被删除。

在 ThreadPoolExecutor.DiscardOldestPolicy 中，如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）。

定义和使用其他种类的 [RejectedExecutionHandler](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\RejectedExecutionHandler.html) 类也是可能的，但这样做需要非常小心，尤其是当策略仅用于特定容量或排队策略时。

G、钩子 (hook) 方法

此类提供 protected 可重写的 beforeExecute(java.lang.Thread, java.lang.Runnable) 和 afterExecute(java.lang.Runnable, java.lang.Throwable) 方法，这两种方法分别在执行每个任务之前和之后调用。它们可用于操纵执行环境；例如，重新初始化 ThreadLocal、搜集统计信息或添加日志条目。此外，还可以重写方法 terminated() 来执行 Executor 完全终止后需要完成的所有特殊处理。

如果钩子 (hook) 或回调方法抛出异常，则内部辅助线程将依次失败并突然终止。

H、队列维护

方法**getQueue()**允许出于监控和调试目的而访问工作队列。强烈反对出于其他任何目的而使用此方法。remove(java.lang.Runnable) 和 [purge()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html#purge()) 这两种方法可用于在取消大量已排队任务时帮助进行存储回收。

I、终止

程序 AND 不再引用的池没有剩余线程会自动 shutdown。如果希望确保回收取消引用的池（即使用户忘记调用 [shutdown()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html#shutdown())），则必须安排未使用的线程最终终止：设置适当保持活动时间，使用 0 核心线程的下边界和/或设置 [allowCoreThreadTimeOut(boolean)](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html#allowCoreThreadTimeOut(boolean))。

### ThreadPoolExecutor(int,int,long,TimeUnit,BlockingQueue<Runnable>,ThreadFactory,RejectedExecutionHandler)

第一个参数[corePoolSize]：池中所保存的线程数，包括空闲线程。

第二个参数[maximumPoolSize]:：池中允许的最大线程数。

第三个参数[keepAliveTime]：**当线程数大于核心时**，此为终止前多余的空闲线程等待新任务的最长时间。

第四个参数[unit]：keepAliveTime 参数的时间单位。

第五个参数[workQueue]：执行前用于保持任务的队列。此队列仅保持由 execute 方法提交的 Runnable 任务。

第六个参数[threadFactory]：执行程序创建新线程时使用的工厂。

第七个参数[handler]：由于超出线程范围和队列容量而使执行被阻塞时所使用的**处理程序**。

1、如果corePoolSize < 0或maxinumPoolSize <= 0或maximumPoolSize < corePoolSize 或 keepAliveTime < 0，将会抛出IllegalArgumentException异常。

2、如果workQueue == null 或 threadFactory == null 或 handler == null，将会抛出NullPointerException异常。

ThreadPoolExecutor还提供了另外三个构造器，如下：

1、ThreadPoolExecutor(**int** corePoolSize, **int** maximumPoolSize, **long** keepAliveTime, TimeUnit unit,BlockingQueue<Runnable> workQueue)

2、ThreadPoolExecutor(**int** corePoolSize,**int** maximumPoolSize,**long** keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory)

3、ThreadPoolExecutor(**int** corePoolSize,**int** maximumPoolSize,**long** keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler)

在这三个构造器的实现上，其最终还是调用当前构造器，只是对某些字段采用了默认的值，如下：

1、ThreadFactory：Executors.defaultThreadFactory() 🡪 Executors.DefaultThreadFactory

2、RejectedExecutionHandler：new AbortPolicy() 🡪ThreadPoolExecutor.AbortPolicy

### execute(Runnable)

在进行此方法源码解读前，需要明白ctl字段的含义，如下：

ctl是主池(main pool)控制状态，它是一个原子类型(AtomicInteger),包括一下两个概念：

a、工作线程数量(workerCount)：表明主池中有效线程的数量。

为了将workerCount包装成一个int，我们限制工作线程最大数量为2^29-1(about 500 million)而不是(2^31)-1(2 billion)，如果在未来这个值不够，我们可以将这个变量改变为AtomicLong。

这个值可能是暂时性的价值不同于实际的活动线程的数量，

b、运行状态(runState)：表明”执行程序”是否运行，关闭等。提供了主要的生命周期控制。

RUNNING: Accept new tasks and process queued tasks

SHUTDOWN: Don't accept new tasks, but process queued tasks

STOP: Don't accept new tasks, don't process queued tasks, and interrupt in-progress tasks

TIDYING: All tasks have terminated, workerCount is zero, the thread transitioning to state TIDYING will run the terminated() hook method

TERMINATED: terminated() has completed

状态转移：

RUNNING -> SHUTDOWN ： On invocation of shutdown(), perhaps implicitly in finalize()

(RUNNING or SHUTDOWN) -> STOP ： On invocation of shutdownNow()

SHUTDOWN -> TIDYING ： When both queue and pool are empty

STOP -> TIDYING ： When pool is empty

TIDYING -> TERMINATED ： When the terminated() hook method has completed

Threads waiting in awaitTermination() will return when the state reaches TERMINATED.

在将来某个时间执行给定任务。可以在新线程中或者在现有池线程中执行该任务。 如果无法将任务提交执行，或者因为此执行程序已关闭，或者因为已达到其容量，则该任务由当前 RejectedExecutionHandler 处理。

**public** **void** execute(Runnable command) {

**if** (command == **null**)

**throw** **new** NullPointerException();

**int** c = ctl.get();

**if** (*workerCountOf*(c) < corePoolSize) {

**if** (addWorker(command, **true**))

**return**;

c = ctl.get();

}

**if** (*isRunning*(c) && workQueue.offer(command)) {

**int** recheck = ctl.get();

**if** (! *isRunning*(recheck) && remove(command))

reject(command);

**else** **if** (*workerCountOf*(recheck) == 0)

addWorker(**null**, **false**);

}

**else** **if** (!addWorker(command, **false**))

reject(command);

}

1、判断预执行的任务是否为空，如果为空则抛出NullPointerException，不进行下一步了。

2、通过workerCountOf方法获取正在工作线程的数量，如果正在工作线程的数量小于核心线程，则任务交给[addWorker(Runnable firstTask, boolean core)](#_addWorker(Runnable_firstTask,_boole)[参数值为(command,true)]方法处理。如果addWorker方法返回true，则返回；否则获取ctl值，进行下一步。

3、根据isRunning方法判断当前执行程序的状态是否为RUNNING，如果处于RUNNING状态则将当前任务加入工作队列中workQueue队列中。

3.1 获取当前执行程序的ctl字段值，根据isRunning方法判断当前执行程序的状态是否为RUNNING，如果不处于RUNNING状态，则调用remove将当前任务从workQueue队列中移除，并尝试中断当前执行程序。

3.2 如果3.1返回false并且执行程序的状态为SHUTDOWN，则调用[addWorker(Runnable firstTask, boolean core)](#_addWorker(Runnable_firstTask,_boole)[参数值为(null,false)]方法处理。

private static boolean isRunning(int c) {

return c < *SHUTDOWN*;

}

CAPACITY:536870911

RUNNING:-536870912

SHUTDOWN:0

STOP:536870912

TIDYING:1073741824

TERMINATED:1610612736

4、如果不满足步骤3的条件判断，则进行这一步。如果调用[addWorker](#_addWorker(Runnable_firstTask,_boole)[参数值为(command,false)]方法返回false，则调用[reject(Runnable command)](#_reject(Runnable_command))方法。

#### addWorker(Runnable firstTask, boolean core)

**private** **boolean** addWorker(Runnable firstTask, **boolean** core) {

retry:

**for** (;;) {

**int** c = ctl.get();

**int** rs = *runStateOf*(c);

// Check if queue empty only if necessary.

**if** (rs >= *SHUTDOWN* && ! (rs == *SHUTDOWN* && firstTask == **null** && ! workQueue.isEmpty()))

**return** **false**;

**for** (;;) {

**int** wc = *workerCountOf*(c);

**if** (wc >= *CAPACITY* || wc >= (core ? corePoolSize : maximumPoolSize))

**return** **false**;

**if** (compareAndIncrementWorkerCount(c))

**break** retry;

c = ctl.get(); // Re-read ctl

**if** (*runStateOf*(c) != rs)

**continue** retry;

// else CAS failed due to workerCount change; retry inner loop

}

}

**boolean** workerStarted = **false**;

**boolean** workerAdded = **false**;

Worker w = **null**;

**try** {

**final** ReentrantLock mainLock = **this**.mainLock;

w = **new** Worker(firstTask); // 将任务交给ThreadPoolExecutor的工作任务,详细见[Worker](#_Worker)

**final** Thread t = w.thread;

**if** (t != **null**) {

mainLock.lock();

**try** {

// Recheck while holding lock.

// Back out on ThreadFactory failure or if

// shut down before lock acquired.

**int** c = ctl.get();

**int** rs = *runStateOf*(c);

**if** (rs < *SHUTDOWN* || (rs == *SHUTDOWN* && firstTask == **null**)) {

**if** (t.isAlive()) // precheck that t is startable

**throw** **new** IllegalThreadStateException();

workers.add(w); // 加入工作任务列表

**int** s = workers.size();

**if** (s > largestPoolSize)

largestPoolSize = s;

workerAdded = **true**;

}

} **finally** {

mainLock.unlock();

}

**if** (workerAdded) {

// 开始执行工作线程，也就是执行[Worker](#_Worker)中的run方法，

// 因为Worker是一个Runnable，并且在创建工作线程t的时候已经将Worker赋给了t。

// 在Worker的run方法中调用了[runWorker(Worker)](#_runWorker(Worker))方法。

t.start();

workerStarted = **true**;

}

}

} **finally** {

**if** (! workerStarted)

addWorkerFailed(w);

}

**return** workerStarted;

}

#### runWorker(Worker)

**final** **void** runWorker(Worker w) {

Thread wt = Thread.*currentThread*();

Runnable task = w.firstTask;

w.firstTask = **null**;

w.unlock(); // 允许中断

**boolean** completedAbruptly = **true**;

**try** {

// 如果预执行的任务为空，则去任务队列workQueue中取。

**while** (task != **null** || (task = getTask()) != **null**) {

w.lock();

// If pool is stopping, ensure thread is interrupted;

// if not, ensure thread is not interrupted. This

// requires a recheck in second case to deal with

// shutdownNow race while clearing interrupt

**if** ((*runStateAtLeast*(ctl.get(), *STOP*) || (Thread.*interrupted*() && *runStateAtLeast*(ctl.get(), *STOP*))) && !wt.isInterrupted())

wt.interrupt();

**try** {

beforeExecute(wt, task);

Throwable thrown = **null**;

**try** {

task.run(); // 执行任务

} **catch** (RuntimeException x) {

thrown = x; **throw** x;

} **catch** (Error x) {

thrown = x; **throw** x;

} **catch** (Throwable x) {

thrown = x; **throw** **new** Error(x);

} **finally** {

afterExecute(task, thrown);

}

} **finally** {

task = **null**;

w.completedTasks++;

w.unlock();

}

}

completedAbruptly = **false**;

} **finally** {

// 1、将执行完的任务移出工作任务workers，并对完成completedTaskCount累加操作。

// 2、调用tryTerminate去判断当前执行程序的状态，并进行相关处理，详细见tryTerminate方法

// 3、判断当前执行程序的状态是否为RUNNING或SHUTDOWN，则判断当前执行任务的数量是否大于corePoolSize,如果大于则返回；否则从任务列表中获取一个线程加入执行任务，并执行。

// addWorker 🡪 runWorker 🡪 addWorker 直到所有任务都执行完成或执行程序终止。

processWorkerExit(w, completedAbruptly);

}

}

#### reject(Runnable command)

**final** **void** reject(Runnable command) {

handler.rejectedExecution(command, **this**);

}

将RejectedExecutionHandler的rejectedExecution方法完成后续处理，详细见[RejectedExecutionHandler](#_RejectedExecutionHandler)。

#### remove(Runnable task)

**public** **boolean** remove(Runnable task) {

**boolean** removed = workQueue.remove(task);

tryTerminate(); // In case SHUTDOWN and now empty

**return** removed;

}

将任务从workQueue工作队列中移除，并调用[tryTerminate()](#_tryTerminate())尝试中断当前执行程序。

#### tryTerminate()

**final** **void** tryTerminate() {

**for** (;;) {

**int** c = ctl.get();

**if** (*isRunning*(c) || *runStateAtLeast*(c, *TIDYING*) || (*runStateOf*(c) == *SHUTDOWN* && ! workQueue.isEmpty()))

**return**;

**if** (*workerCountOf*(c) != 0) { // Eligible to terminate

interruptIdleWorkers(*ONLY\_ONE*);

**return**;

}

**final** ReentrantLock mainLock = **this**.mainLock;

mainLock.lock();

**try** {

**if** (ctl.compareAndSet(c, *ctlOf*(*TIDYING*, 0))) {

**try** {

terminated();

} **finally** {

ctl.set(*ctlOf*(*TERMINATED*, 0));

termination.signalAll();

}

**return**;

}

} **finally** {

mainLock.unlock();

}

// else retry on failed CAS

}

}

1、如果当前执行线程的状态为RUNNING或者TIDYING或者SHUTDOWN[任务队列workerQueue不为空]则返回。当状态为STOP、SHUTDOWN(不存在任务队列workerQueue)和TERMINATED进入步骤2。

2、如果存在工作线程，则中断工作线程队列workers中的第一个未中断的线程，并返回。

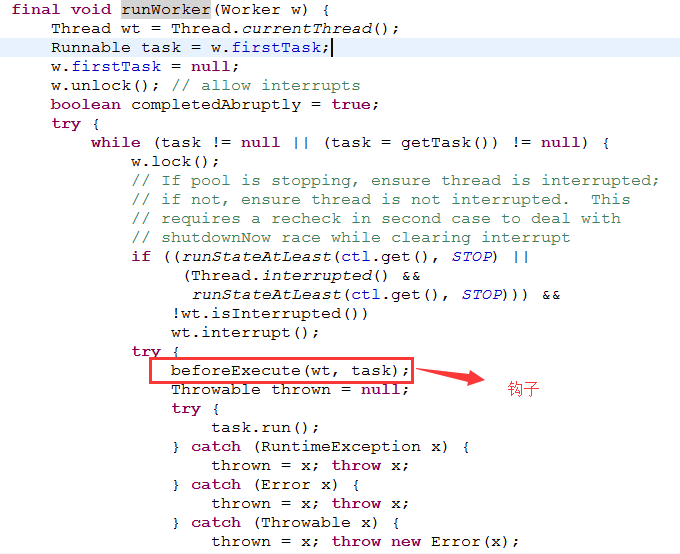
3、将状态设置为TIDYING，调用钩子方法terminated(),执行完terminated()方法后，将状态设置为TERMINATED，唤醒mainLock上所有阻塞的线程。

mainLock在[awaitTermination](#_awaitTermination(long_timeout,_Time)方法中也被使用到了。

### beforeExecute(Thread t, Runnable r)

在执行给定线程中的给定Runnable之前调用的方法。此方法由将执行任务r的线程t调用，并且可用于重新初始化ThreadLocals或者执行日志记录。

此实现不执行任何操作，但可在子类中定制。注：为了正确嵌套多个重写操作，此方法结束时，子类通常应该调用 super.beforeExecute。



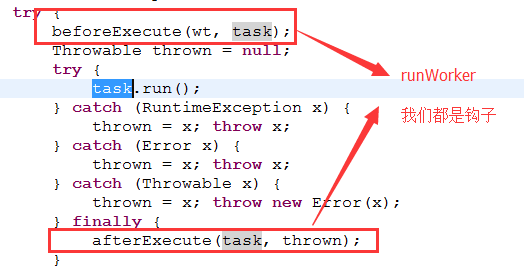
### afterExecute(Runnable r, Throwable t)

基于完成执行给定Runnable所调用的方法。此方法由执行任务的线程调用。如果非null，则Throwable是导致执行突然终止的未捕获 RuntimeException或Error。

注：当操作显式地或者通过submit之类的方法包含在任务内时（如 FutureTask），这些任务对象捕获和维护计算异常，因此它们不会导致突然终止，内部异常不会传递给此方法。

此实现不执行任何操作，但可在子类中定制。注：为了正确嵌套多个重写操作，此方法开始时，子类通常应该调用 super.afterExecute。

详细可以查看ThreadPoolExecutor的runWorker方法



### awaitTermination(long timeout, TimeUnit unit)

**public** **boolean** awaitTermination(**long** timeout, TimeUnit unit) **throws** InterruptedException {

**long** nanos = unit.toNanos(timeout);

**final** ReentrantLock mainLock = **this**.mainLock;

mainLock.lock();

**try** {

**for** (;;) {

**if** (*runStateAtLeast*(ctl.get(), *TERMINATED*))

**return** **true**;

**if** (nanos <= 0)

**return** **false**;

//造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。

nanos = termination.awaitNanos(nanos);

}

} **finally** {

mainLock.unlock();

}

}

1、从for(;;)中的两个条件判断可以看出，如果等待时间期满则返回false，如果执行程序终止则返回true。

2、这个方法用到了mainLock锁，这将导致请求关闭、发生超时或当前线程中断都会被阻塞。

3、termination是mainLock上的一个Condition，用来进行时间等待控制。

### remove(Runnable)

**public** **boolean** remove(Runnable task) {

**boolean** removed = workQueue.remove(task);

tryTerminate(); // In case SHUTDOWN and now empty

**return** removed;

}

从执行程序的内部队列中移除此任务（如果存在），从而如果尚未开始，则其不再运行。

此方法可用作取消方案的一部分。它可能无法移除在放置到内部队列之前已经转换为其他形式的任务。例如，使用 submit 输入的任务可能被转换为维护 Future 状态的形式。但是，在此情况下，[purge()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html#purge()) 方法可用于移除那些已被取消的 Future。

### purge()

**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())

it.remove();

}

} **catch** (ConcurrentModificationException fallThrough) {

**for** (Object r : q.toArray())

**if** (r **instanceof** Future<?> && ((Future<?>)r).isCancelled())

q.remove(r);

}

tryTerminate(); // In case SHUTDOWN and now empty

}

尝试从工作队列移除所有**已取消的**[**Future**](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Future.html) 任务。此方法可用作存储回收操作，它对功能没有任何影响。取消的任务不会再次执行，但是它们可能在工作队列中累积，直到 worker 线程主动将其移除。调用此方法将试图立即移除它们。但是，如果出现其他线程的干预，那么此方法移除任务将失败。

### shutdown()

按过去执行已提交任务的顺序发起一个有序的关闭，但是不接受新任务。如果已经关闭，则调用没有其他作用。

**public** **void** shutdown() {

**final** ReentrantLock mainLock = **this**.mainLock;

mainLock.lock();

**try** {

checkShutdownAccess();

advanceRunState(*SHUTDOWN*);

interruptIdleWorkers();

onShutdown(); // hook for ScheduledThreadPoolExecutor

} **finally** {

mainLock.unlock();

}

tryTerminate();

}

1、checkShutdownAccess方法的目的是为了让调用者具有shutdown线程的权限。

2、advanceRunState方法的目的是将执行程序的状态(ctl)设置为SHUTDOWN。

3、interruptIdleWorkers的目的是为了中断所有工作线程(works),在此方法中使用到了mainLock锁。

4、调用钩子onShutdown()，在ScheduledThreadPoolExecutor中对此方法中有实现。

5、调用tryTerminate()方法试图中断当前执行程序。详细见[tryTerminate()](#_tryTerminate())。

### shutdownNow()

尝试停止所有的活动执行任务、暂停等待任务的处理，并返回等待执行的任务列表。在从此方法返回的任务队列中排空（移除）这些任务。

并不保证能够停止正在处理的活动执行任务，但是会尽力尝试。 此实现通过 [Thread.interrupt()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt()) 取消任务，所以无法响应中断的任何任务可能永远无法终止。

### Worker

Work主要维护线程运行任务的中断控制状态，连同其他次要的记录。它继承了AQS，这样在任务执行时可以能够获得和释放相关的锁。

private final class Worker extends AbstractQueuedSynchronizer implements Runnable {

private static final long *serialVersionUID* = 6138294804551838833L;

/\*\* Thread this worker is running in. Null if factory fails. \*/

final Thread thread;

/\*\* Initial task to run. Possibly null. \*/

Runnable firstTask;

/\*\* Per-thread task counter \*/

volatile long completedTasks;

Worker(Runnable firstTask) {

setState(-1); // inhibit interrupts until runWorker

this.firstTask = firstTask;

this.thread = getThreadFactory().newThread(this);// 将Worker交给工作线程

}

/\*\* Delegates main run loop to outer runWorker \*/

public void run() {

runWorker(this);

}

protected boolean isHeldExclusively() {// 判断锁是否被占用

return getState() != 0;

}

protected boolean tryAcquire(int unused) {

if (compareAndSetState(0, 1)) {

setExclusiveOwnerThread(Thread.*currentThread*()); // 当前线程获得锁

return true;

}

return false;

}

protected boolean tryRelease(int unused) { // 释放锁

setExclusiveOwnerThread(null);

setState(0);

return true;

}

public void lock() { acquire(1); }

public boolean tryLock() { return tryAcquire(1); }

public void unlock() { release(1); }

public boolean isLocked() { return isHeldExclusively(); }

void interruptIfStarted() { // 中断执行线程

Thread t;

if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {

try {

t.interrupt();

} catch (SecurityException ignore) {

}

}

}

}

## ScheduledExecutorService

一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

schedule 方法使用各种延迟创建任务，并返回一个可用于取消或检查执行的任务对象。scheduleAtFixedRate 和 scheduleWithFixedDelay 方法创建并执行某些在取消前一直定期运行的任务。

用 Executor.execute(java.lang.Runnable) 和 ExecutorService 的 submit 方法所提交的命令，通过所请求的 0 延迟进行安排。schedule 方法中允许出现 0 和负数延迟（但不是周期），并将这些视为一种立即执行的请求。

所有的 schedule 方法都接受相对延迟和周期作为参数，而不是绝对的时间或日期。将以 [Date](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Date.html) 所表示的绝对时间转换成要求的形式很容易。例如，要安排在某个以后的 Date 运行，可以使用：schedule(task, date.getTime() - System.currentTimeMillis(), TimeUnit.MILLISECONDS)。但是要注意，由于网络时间同步协议、时钟漂移或其他因素的存在，因此相对延迟的期满日期不必与启用任务的当前 Date 相符。 [Executors](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Executors.html) 类为此包中所提供的 ScheduledExecutorService实现提供了便捷的工厂方法。

## ScheduledThreadPoolExecutor

ThreadPoolExecutor，它可另行安排在给定的延迟后运行命令，或者定期执行命令。需要多个辅助线程时，或者要求 ThreadPoolExecutor 具有额外的灵活性或功能时，此类要优于 [Timer](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Timer.html)。

一旦启用已延迟的任务就执行它，**但是有关何时启用，启用后何时执行则没有任何实时保证**。按照提交的先进先出 (FIFO) 顺序来启用那些被安排在同一执行时间的任务。

虽然此类继承自 [ThreadPoolExecutor](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html)，但是几个继承的调整方法对此类并无作用。特别是，因为它作为一个**使用 corePoolSize 线程**和**一个无界队列的固定大小的池**，所以调整 maximumPoolSize 没有什么效果。

扩展注意事项：此类重写 AbstractExecutorService 的 submit 方法，以生成内部对象控制每个任务的延迟和调度。若要保留功能性，子类中任何进一步重写的这些方法都必须调用超类版本，超类版本有效地禁用附加任务的定制。但是，此类提供替代受保护的扩展方法 decorateTask（为 Runnable 和Callable 各提供一种版本），可定制用于通过 execute、submit、schedule、scheduleAtFixedRate 和 scheduleWithFixedDelay 进入的执行命令的具体任务类型。默认情况下，ScheduledThreadPoolExecutor 使用一个扩展 [FutureTask](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\FutureTask.html) 的任务类型。但是，可以使用下列形式的子类修改或替换该类型。

public class CustomScheduledExecutor extends ScheduledThreadPoolExecutor {

static class CustomTask<V> implements RunnableScheduledFuture<V> { ... }

protected <V> RunnableScheduledFuture<V> decorateTask(Runnable r, RunnableScheduledFuture<V> task) {

return new CustomTask<V>(r, task);

}

protected <V> RunnableScheduledFuture<V> decorateTask(Callable<V> c, RunnableScheduledFuture<V> task) {

return new CustomTask<V>(c, task);

}

// ... add constructors, etc.

}

### ScheduledThreadPoolExecutor(int corePoolSize)

使用给定核心池大小创建一个新ScheduledThreadPoolExecutor。

**public** ScheduledThreadPoolExecutor(**int** corePoolSize) {

**super**(corePoolSize, Integer.*MAX\_VALUE*, 0, TimeUnit.*NANOSECONDS*, **new** DelayedWorkQueue());

}

DelayedWorkQueue是ScheduledThreadPoolExecutor中的一个私有匿名类，它是一个无界队列，所以最大线程数设置将在这里失效。

**private** **static** **class** DelayedWorkQueue **extends** AbstractCollection<Runnable>

**implements** BlockingQueue<Runnable> {

// …

}

### ScheduledThreadPoolExecutor(int ,ThreadFactory )

**public** ScheduledThreadPoolExecutor(**int** corePoolSize, ThreadFactory threadFactory) {

**super**(corePoolSize, Integer.*MAX\_VALUE*, 0, TimeUnit.*NANOSECONDS*, **new** DelayedWorkQueue(), threadFactory);

}

### ScheduledThreadPoolExecutor(int, RejectedExecutionHandler)

**public** ScheduledThreadPoolExecutor(**int** corePoolSize, RejectedExecutionHandler handler) {

**super**(corePoolSize, Integer.*MAX\_VALUE*, 0, TimeUnit.*NANOSECONDS*, **new** DelayedWorkQueue(), handler);

}

### ScheduledThreadPoolExecutor(int,ThreadFactory,RejectedExecutionHandler)

**public** ScheduledThreadPoolExecutor(**int** corePoolSize, ThreadFactory threadFactory, RejectedExecutionHandler handler) {

**super**(corePoolSize, Integer.*MAX\_VALUE*, 0, TimeUnit.*NANOSECONDS*, **new** DelayedWorkQueue(), threadFactory, handler);

}

### execute(Runnable command)

使用所要求的零延迟执行命令。这在效果上等同于调用schedule(command, 0, anyUnit)。注意，对由 **shutdownNow**所返回的队列和列表的检查将访问零延迟的**ScheduledFuture**，而不是command本身。

**public** **void** execute(Runnable command) {

schedule(command, 0, TimeUnit.*NANOSECONDS*);

}

方法将转调[schedule(Runnable command, long delay, TimeUnit unit)](#_schedule(Runnable_command,_long)完成任务调度。

### submit(Callable)

提交一个返回值的任务用于执行，返回一个表示任务的未决结果的Future。该Future 的 get 方法在成功完成时将会返回该任务的结果。

如果想立即阻塞任务的等待，则可以使用 result = exec.submit(aCallable).**get()**; 形式的构造。

注：Executors 类包括了一组方法，可以转换某些其他常见的类似于闭包的对象，例如，将 PrivilegedAction 转换为 [Callable](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Callable.html) 形式，这样就可以提交它们了。

**public** <T> Future<T> submit(Callable<T> task) {

**return** schedule(task, 0, TimeUnit.*NANOSECONDS*);

}

可以看出此方法提交的任务将被立即执行，详细见[schedule(Callable<V> callable, long delay, TimeUnit unit)](#_schedule(Callable<V>_callable,_long)

### submit(Runnable,T)

提交一个 Runnable 任务用于执行，并返回一个表示该任务的Future。该Future 的 get 方法在成功完成时将会返回给定的结果。

**public** <T> Future<T> submit(Runnable task, T result) {

**return** schedule(Executors.*callable*(task, result), 0, TimeUnit.*NANOSECONDS*);

}

可以看出，这里调用Executors.callable(Runnable,T)接口将Runnable封装成Callable[RunnableAdapter]对象，然后转调[schedule(Callable<V> callable, long delay, TimeUnit unit)](#_schedule(Callable<V>_callable,_long)方法。

### submit(Runnable)

提交一个Runnable任务用于执行，并返回一个表示该任务的Future。该 Future 的 get 方法在成功完成时将会返回 null。

**public** Future<?> submit(Runnable task) {

**return** schedule(task, 0, TimeUnit.*NANOSECONDS*);

}

将调用[schedule(Runnable command, long delay, TimeUnit unit)](#_schedule(Runnable_command,_long)处理任务的调度

### schedule(Callable<V> callable, long delay, TimeUnit unit)

创建并执行在给定延迟后启用的ScheduledFuture。

**public** <V> ScheduledFuture<V> schedule(Callable<V> callable, **long** delay, TimeUnit unit) {

**if** (callable == **null** || unit == **null**)

**throw** **new** NullPointerException();

**if** (delay < 0) delay = 0;

**long** triggerTime = now() + unit.toNanos(delay); // 触发时间

RunnableScheduledFuture<V> t = decorateTask(callable, **new** ScheduledFutureTask<V>(callable, triggerTime));

delayedExecute(t);

**return** t;

}

1、对Callable和TimeUnit进行非空校验

2、对delay[从现在开始延迟执行的时间]进行校验，如果小于0则将值设置为0.并计算任务的触发时间。

3、装饰任务[Callable]，将任务封装成[ScheduledFutureTask](#_ScheduledFutureTask)，ScheduledFutureTask继承了FutureTask并实现了RunnableScheduledFuture接口。schedule方法中实例化RunnableScheduledFuture时，period等于0，此值表示任务是否周期执行。

4、decorateTask是ScheduledThreadPoolExecutor提供的一个受保护方法，此方法主要是” 修改或替换用于执行 callable 的任务。此方法可重写用于管理内部任务的具体类。默认实现返回给定任务。”，可以在ScheduledThreadPoolExecutor子类中重写此方法逻辑。

5、调用[delayedExecute(RunnableScheduledFuture<?> task)](#_delayedExecute(RunnableScheduledFut)方法执行任务。

### schedule(Runnable command, long delay, TimeUnit unit)

创建并执行在给定延迟后启用的一次性操作。

**public** ScheduledFuture<?> schedule(Runnable command, **long** delay, TimeUnit unit) {

**if** (command == **null** || unit == **null**)

**throw** **new** NullPointerException();

**if** (delay < 0) delay = 0;

**long** triggerTime = now() + unit.toNanos(delay);

RunnableScheduledFuture<?> t = decorateTask(command, **new** ScheduledFutureTask<Void>(command, **null**, triggerTime));

delayedExecute(t);

**return** t;

}

与[schedule(Callable<V> callable, long delay, TimeUnit unit)](#_schedule(Callable<V>_callable,_long)类似。

### scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

initialDelay：首次执行的延迟时间

period ： 连续执行之间的周期

创建并执行一个在给定初始延迟后首次启用的定期操作，后续操作具有给定的周期；也就是将在 initialDelay 后开始执行，然后在initialDelay+period 后执行，接着在 initialDelay + 2 \* period 后执行，依此类推。如果任务的任何一个执行遇到异常，则后续执行都会被取消。否则，只能通过执行程序的取消或终止方法来终止该任务。**如果此任务的任何一个执行要花费比其周期更长的时间，则将推迟后续执行，但不会同时执行**。

**public** ScheduledFuture<?> scheduleAtFixedRate(Runnable command, **long** initialDelay, **long** period, TimeUnit unit) {

**if** (command == **null** || unit == **null**)

**throw** **new** NullPointerException();

**if** (period <= 0)

**throw** **new** IllegalArgumentException();

**if** (initialDelay < 0) initialDelay = 0;

**long** triggerTime = now() + unit.toNanos(initialDelay);

ScheduledFutureTask<Void> sft = **new** ScheduledFutureTask<Void>(command,**null**,triggerTime, unit.toNanos(period));

RunnableScheduledFuture<Void> t = decorateTask(command, sft);

sft.outerTask = t;

delayedExecute(t);

**return** t;

}

### scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)

initialDelay ：首次执行的延迟时间

delay ：一次执行终止和下一次执行开始之间的延迟

创建并执行一个在给定初始延迟后首次启用的定期操作，随后，在**每一次执行终止**和**下一次执行开始之间**都存在给定的延迟。**如果任务的任一执行遇到异常，就会取消后续执行**。否则，只能通过执行程序的取消或终止方法来终止该任务。

**public** ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, **long** initialDelay, **long** delay, TimeUnit unit) {

**if** (command == **null** || unit == **null**)

**throw** **new** NullPointerException();

**if** (delay <= 0)

**throw** **new** IllegalArgumentException();

**if** (initialDelay < 0) initialDelay = 0;

**long** triggerTime = now() + unit.toNanos(initialDelay);

ScheduledFutureTask<Void> sft = **new** ScheduledFutureTask<Void>(command,**null**,triggerTime, unit.toNanos(-delay));

RunnableScheduledFuture<Void> t = decorateTask(command, sft);

sft.outerTask = t;

delayedExecute(t);

**return** t;

}

### delayedExecute(RunnableScheduledFuture<?> task)

**private** **void** delayedExecute(RunnableScheduledFuture<?> task) {

**if** (isShutdown())

reject(task);

**else** {

**super**.getQueue().add(task);

**if** (isShutdown() && !canRunInCurrentRunState(task.isPeriodic()) && remove(task))

task.cancel(**false**);

**else**

prestartCoreThread();

}

}

1、如果当前执行程序已经关闭(SHUTDOWN、STOP、TIDYING和TERMINATED状态)，则调用reject()方法，在前面提到过reject方法内部其实就是调用了[RejectedExecutionHandler](#_RejectedExecutionHandler)的rejectedExecution()方法。具体那个实现类，就要看初始化ScheduledThreadPoolExecutor的设值了，当然在初始化之后调用ThreadPoolExecutor的setRejectedExecutionHandler动态设值。

2、如果执行程序状态为READY或RUNNING，则将任务加入任务队列workQueue中。

3、判断任务是否要取消，同时满足一下三个条件则取消任务。

3.1 调用isShutDown()方法，判断当前执行程序是否关闭。如果关闭则判断3.2逻辑，否则返回。

3.2 调用canRunInCurrentRunState判断任务关闭之后是否可以继续执行任务，task.isPeriodic()的值根据ScheduledFutureTask中period参数决定。如果canRunInCurrentRunState为true则返回，否则继续3.3逻辑。

**boolean** canRunInCurrentRunState(**boolean** periodic) {

**return** isRunningOrShutdown(periodic ? continueExistingPeriodicTasksAfterShutdown : executeExistingDelayedTasksAfterShutdown);

}

**final** **boolean** isRunningOrShutdown(**boolean** shutdownOK) {

**int** rs = *runStateOf*(ctl.get());

**return** rs == *RUNNING* || (rs == *SHUTDOWN* && shutdownOK);

}

continueExistingPeriodicTasksAfterShutdown：设置有关在此执行程序已 shutdown 的情况下是否继续执行现有**定期任务**的策略。在这种情况下，仅在执行 shutdownNow 时，或者在执行程序已关闭、将策略设置为 false 后才终止这些任务。此值默认为 false。可以通过**setContinueExistingPeriodicTasksAfterShutdownPolicy(boolean)**动态设置。

**public** **void** setContinueExistingPeriodicTasksAfterShutdownPolicy(**boolean** value) {

continueExistingPeriodicTasksAfterShutdown = value;

**if** (!value && isShutdown())

onShutdown();

}

executeExistingDelayedTasksAfterShutdown: 设置有关在此执行程序已 shutdown 的情况下是否继续执行现有**延迟任务**的策略。在这种情况下，仅在执行 shutdownNow 时，或者在执行程序已关闭、将策略设置为 false 后才终止这些任务。此值默认为 true。可以通过**setExecuteExistingDelayedTasksAfterShutdownPolicy(boolean)**动态设置。

**public** **void** setExecuteExistingDelayedTasksAfterShutdownPolicy(**boolean** value) {

executeExistingDelayedTasksAfterShutdown = value;

**if** (!value && isShutdown())

onShutdown(); // 当执行程序关闭后，取消和清空任务队列中所有的任务。

}

3.3 从任务队列wokeQueue中移除任务。

如果满足上述3个条件，则调用FutureTask的cancel(boolean)取消任务，但是传入cancel方法的值为false，代表执行程序不会去中断执行任务。

4、如果3中的三个条件”与运算”返回false，则进入本步骤。调用prestartCoreThread方法执行任务。

**public** **boolean** prestartCoreThread() {

**return** *workerCountOf*(ctl.get()) < corePoolSize && addWorker(**null**, **true**);

}

如果工作线程数少于核心线程数corePoolSize，则调用[addWorker(Runnable firstTask, boolean core)](#_addWorker(Runnable_firstTask,_boole)。从前面可以知道工作任务Woker会触发FutureTask的run()方法的执行，所以这里我们重点需要关注的是ScheduledFutureTask的[run()](#_run())方法逻辑。

需要注意的是，调用addWorker时，传入的Runnable是空，在启动工作任务Worker时，会触发runWorker方法的调用，在runWorker方法中存在一段逻辑，如下：

**final** **void** runWorker(Worker w) {

Thread wt = Thread.*currentThread*();

Runnable task = w.firstTask; // task为空

// ..

// task为空，则从workQueue获取，因为这里的workQueue是DelayedWorkQueue类型，并且DelayedWorkQueue

// 中操作的元素全部都是ScheduledFutureTask类型，所以，延迟时间的判断由ScheduledFutureTask的[getDelay(TimeUnit)](#_getDelay(TimeUnit))决定。

**while** (task != **null** || (task = getTask()) != **null**) { }

}

### ScheduledFutureTask

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

/\*\* Sequence number to break ties FIFO \*/

**private** **final** **long** sequenceNumber;

/\*\* The time the task is enabled to execute in nanoTime units \*/

**private** **long** time;

/\*\*

\* Period in nanoseconds for repeating tasks. A positive value indicates fixed-rate execution.

\* A negative value indicates fixed-delay execution. A value of 0 indicates a non-repeating task.

\*/

**private** **final** **long** period;

/\*\* The actual task to be re-enqueued by reExecutePeriodic \*/

RunnableScheduledFuture<V> outerTask = **this**;

/\*\*

\* Creates a one-shot action with given nanoTime-based trigger time.

\*/

ScheduledFutureTask(Runnable r, V result, **long** ns) {

**super**(r, result);

**this**.time = ns;

**this**.period = 0;

**this**.sequenceNumber = *sequencer*.getAndIncrement();

}

/\*\*

\* Creates a periodic action with given nano time and period.

\*/

ScheduledFutureTask(Runnable r, V result, **long** ns, **long** period) {

**super**(r, result);

**this**.time = ns;

**this**.period = period;

**this**.sequenceNumber = *sequencer*.getAndIncrement();

}

/\*\*

\* Creates a one-shot action with given nanoTime-based trigger.

\*/

ScheduledFutureTask(Callable<V> callable, **long** ns) {

**super**(callable);

**this**.time = ns;

**this**.period = 0;

**this**.sequenceNumber = *sequencer*.getAndIncrement();

}

**// …**

}

1、因为ScheduledFutureTask继承了FutureTask，所以在ScheduledFutureTask的所有构造器中可以看出，它们都调用了父类的构造器，详细见[FutureTask](#_FutureTask)。也就是说，不管是Runnable还是Callable都会给封装成FutureTask中的Sync对象。

2、因为Callable只能被计算一次，ScheduledFutureTask没有定义可以周期执行的Callable，但是定义了可以周期执行的Runnable。

ScheduledFutureTask(Runnable r, V result, **long** ns, **long** period) {

**super**(r, result);

**this**.time = ns;

**this**.period = period;

**this**.sequenceNumber = *sequencer*.getAndIncrement();

}

#### run()

public void run() {

boolean periodic = isPeriodic();

if (!canRunInCurrentRunState(periodic))

cancel(false);

else if (!periodic)

ScheduledFutureTask.super.run();

else if (ScheduledFutureTask.super.runAndReset()) {

setNextRunTime();

reExecutePeriodic(outerTask);

}

}

这个方法三段逻辑

1、判断当前任务是否可以执行，如果不可执行则取消任务。

2、如果当前调度任务为非周期执行，则调用ScheduledFutureTask的父类[FutureTask](#_FutureTask)的[run()](#_run()_1)方法执行任务。

3、然后执行目标任务，并重置任务状态、设置下次执行时间，并将任务从新加入workQueue队列中。

**private** **void** setNextRunTime() { // 设置下一次执行的时间

**long** p = period;

**if** (p > 0)

time += p; // [scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)](#_scheduleAtFixedRate(Runnable_comman)

**else**

time = now() - p; //[scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)](#_scheduleWithFixedDelay(Runnable_com)

}

#### getDelay(TimeUnit)

**public** **long** getDelay(TimeUnit unit) {

**long** d = unit.convert(time - now(), TimeUnit.*NANOSECONDS*);

**return** d;

}

#### compareTo(Delayed)

**public** **int** compareTo(Delayed other) {

**if** (other == **this**) // compare zero ONLY if same object

**return** 0;

**if** (other **instanceof** ScheduledFutureTask) {

ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;

**long** diff = time - x.time;

**if** (diff < 0)

**return** -1;

**else** **if** (diff > 0)

**return** 1;

**else** **if** (sequenceNumber < x.sequenceNumber)

**return** -1;

**else**

**return** 1;

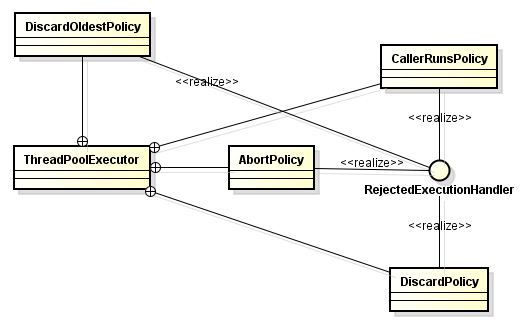
}

**long** d = (getDelay(TimeUnit.*NANOSECONDS*) - other.getDelay(TimeUnit.*NANOSECONDS*));

**return** (d == 0) ? 0 : ((d < 0) ? -1 : 1);

}

# RejectedExecutionHandler



RejectedExecutionHandler类图

无法由 [ThreadPoolExecutor](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ThreadPoolExecutor.html) 执行的任务的处理程序。

ThreadPoolExecutor.AbortPolicy：用于被拒绝任务的处理程序，它将抛出 RejectedExecutionException.

ThreadPoolExecutor.CallerRunsPolicy：用于被拒绝任务的处理程序，它直接在 execute 方法的调用线程中运行被拒绝的任务；如果执行程序已关闭，则会丢弃该任务。

**public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {

**if** (!e.isShutdown()) {// 如果执行程序未关闭，则直接执行任务。

r.run();

}

}

ThreadPoolExecutor.DiscardOldestPolicy：用于被拒绝任务的处理程序，它放弃最旧的未处理请求，然后重试 execute；如果执行程序已关闭，则会丢弃该任务。

**public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {

**if** (!e.isShutdown()) {

e.getQueue().poll();// 放弃执行程序任务队列中等待最久的任务

e.execute(r);// 尝试执行任务。

}

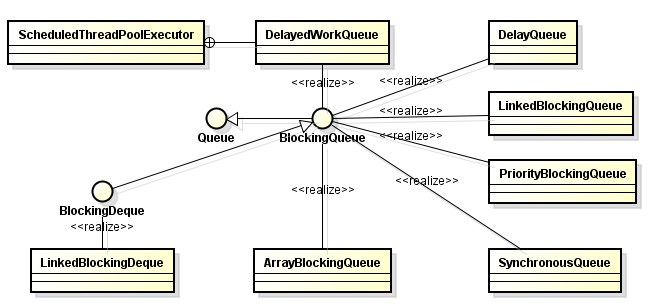
}

ThreadPoolExecutor.DiscardPolicy：用于被拒绝任务的处理程序，默认情况下它将丢弃被拒绝的任务

**public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor e) {

}

# BlockingQueue



BlockingQueue类图

支持两个附加操作的 Queue，这两个操作是：获取元素时等待队列变为非空，以及存储元素时等待空间变得可用。

BlockingQueue 方法以四种形式出现，对于不能立即满足但可能在将来某一时刻可以满足的操作，这四种形式的处理方式不同：

第一种是抛出一个异常。

第二种是返回一个特殊值（null 或 false，具体取决于操作）。

第三种是在操作可以成功前，无限期地阻塞当前线程。

第四种是在放弃前只在给定的最大时间限制内阻塞。下表中总结了这些方法：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 抛出异常 | 特殊值 | 阻塞 | 超时 |
|  |  |  |  |  |
| 插入 | add(e) | offer(e) | put(e) | offer(e, time, unit) |
| 移除 | remove() | poll() | take() | poll(time, unit) |
| 检查 | element() | peek() | 不可用 | 不可用 |

**BlockingQueue** 不接受 **null** 元素。试图 add、put 或 offer 一个 null 元素时，某些实现会抛出 NullPointerException。null 被用作指示 poll 操作失败的警戒值。

BlockingQueue 可以是限定容量的。它在任意给定时间都可以有一个 remainingCapacity，超出此容量，便无法无阻塞地 put 附加元素。没有任何内部容量约束的 BlockingQueue 总是报告 Integer.MAX\_VALUE 的剩余容量。

BlockingQueue 实现主要用于生产者-使用者队列，但它另外还支持 [Collection](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Collection.html) 接口。因此，举例来说，使用 remove(x) 从队列中移除任意一个元素是有可能的。然而，这种操作通常不会有效执行，只能有计划地偶尔使用，比如在取消排队信息时。

**BlockingQueue 实现**是**线程安全**的。**所有排队方法**都可以使用**内部锁**或**其他形式的并发控制**来自动达到它们的目的。然而，大量的 Collection 操作（addAll、containsAll、retainAll 和 removeAll）没有必要自动执行，除非在实现中特别说明。因此，举例来说，在只添加了 c 中的一些元素后，addAll(c) 有可能失败（抛出一个异常）。

BlockingQueue 实质上不支持使用任何一种“close”或“shutdown”操作来指示不再添加任何项。这种功能的需求和使用有依赖于实现的倾向。例如，一种常用的策略是：对于生产者，插入特殊的 end-of-stream 或 poison 对象，并根据使用者获取这些对象的时间来对它们进行解释。

内存一致性效果：当存在其他并发 collection 时，将对象放入 BlockingQueue 之前的线程中的操作 [happen-before](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\package-summary.html#MemoryVisibility) 随后通过另一线程从BlockingQueue 中访问或移除该元素的操作。

## ArrayBlockingQueue

一个由数组支持的**有界**[**阻塞队列**](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\BlockingQueue.html)。此队列按FIFO（先进先出）原则对元素进行排序。队列的头部是在队列中存在时间最长的元素。队列的尾部是在队列中存在时间最短的元素。新元素插入到队列的尾部，队列获取操作则是从队列头部开始获得元素。

这是一个典型的“有界缓存区”，固定大小的数组在其中保持生产者插入的元素和使用者提取的元素。一旦创建了这样的缓存区，就不能再增加其容量。试图向已满队列中放入元素会导致操作受阻塞；试图从空队列中提取元素将导致类似阻塞。

此类支持对等待的生产者线程和使用者线程**进行排序的可选公平策略**。默认情况下，不保证是这种排序。然而，通过将公平性 (fairness) 设置为true 而构造的队列允许按照FIFO顺序访问线程。公平性通常会降低吞吐量，但也减少了可变性和避免了“不平衡性”。

此类及其迭代器实现了Collection 和 Iterator 接口的所有可选方法。

/\*\* The queued items \*/

**private** **final** E[] items;

/\*\* items index for next take, poll or remove \*/

**private** **int** takeIndex;

/\*\* items index for next put, offer, or add. \*/

**private** **int** putIndex;

/\*\* Number of items in the queue \*/

**private** **int** count;

/\* Concurrency control uses the classic two-condition algorithm found in any textbook. \*/

/\*\* Main lock guarding all access \*/

**private** **final** ReentrantLock lock;

/\*\* Condition for waiting takes \*/

**private** **final** Condition notEmpty;

/\*\* Condition for waiting puts \*/

**private** **final** Condition notFull;

### ArrayBlockingQueue(int capacity)

**public** ArrayBlockingQueue(**int** capacity) {

**this**(capacity, **false**);

}

创建一个带有给定的（固定）容量和默认访问策略的 ArrayBlockingQueue。

这个默认策略就是创建一个非公平策略的可重入的互斥锁ReentrantLock.

详细见[ArrayBlockingQueue(int capacity, boolean fair)](#_ArrayBlockingQueue(int_capacity,_bo)。

### ArrayBlockingQueue(int capacity, boolean fair)

**public** ArrayBlockingQueue(**int** capacity, **boolean** fair) {

**if** (capacity <= 0)

**throw** **new** IllegalArgumentException();

**this**.items = (E[]) **new** Object[capacity];

lock = **new** ReentrantLock(fair);

notEmpty = lock.newCondition();

notFull = lock.newCondition();

}

创建一个具有给定的（固定）容量和指定访问策略的ArrayBlockingQueue。

fair ：如果为 true，则按照 **FIFO顺序访问**插入或移除时受阻塞线程的队列；如果为 false，则**访问顺序**是不确定的。

### ArrayBlockingQueue(int, boolean, Collection<? extends E>)

**public** ArrayBlockingQueue(**int** capacity, **boolean** fair, Collection<? **extends** E> c) {

**this**(capacity, fair);

**if** (capacity < c.size())

**throw** **new** IllegalArgumentException();

**for** (Iterator<? **extends** E> it = c.iterator(); it.hasNext();)

add(it.next());

}

创建一个具有给定的（固定）容量和指定访问策略的 ArrayBlockingQueue，它最初包含给定 collection 的元素，并以 collection 迭代器的遍历顺序添加元素。

### add(E)

将指定的元素插入到此队列的尾部（如果立即可行且不会超过该队列的容量），在成功时返回 true，如果此队列已满，则抛出IllegalStateException。

public boolean add(E e) {

return super.add(e);

}

调用父类AbstractQueue的add(E)方法，如下：

public boolean add(E e) {

if (offer(e))

return true;

else

throw new IllegalStateException("Queue full");

}

AbstractQueue的add方法会调用offer(E)方法将元素添加到队列中，offer(E)在不同的子类中有不同的实现，ArrayBlockingQueue的实现见[offer(E)](#_offer(E))。如果添加元素失败[可能队列已满]，则抛出IllegalStateException异常。

### offer(E)

将指定的元素插入到此队列的尾部（如果立即可行且不会超过该队列的容量），在成功时返回 true，如果此队列已满，则返回 false。此方法通常要优于 [add(E)](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ArrayBlockingQueue.html#add(E)) 方法，后者可能无法插入元素，而只是抛出一个异常。

**public** **boolean** offer(E e) {

**if** (e == **null**) **throw** **new** NullPointerException();

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**if** (count == items.length) // 队列已满

**return** **false**;

**else** { // 将元素插入队列末端

insert(e);

**return** **true**;

}

} **finally** {

lock.unlock();

}

}

**private** **void** insert(E x) { // 务必在保持锁的线程中操作

items[putIndex] = x;

putIndex = inc(putIndex); // 将待插入元素的下标加1,如果加1后达到最大数(item.length)，则为0

++count;// 总数加1

notEmpty.signal(); // 唤醒非空Condition

}

**final** **int** inc(**int** i) {

**return** (++i == items.length)? 0 : i;

}

插入的过程是不可中断的，详细见ReentrantLock的lock方法。

### put(E)

将指定的元素插入此队列的尾部，如果该队列已满，则等待可用的空间。

**public** **void** put(E e) **throws** InterruptedException {

**if** (e == **null**) **throw** **new** NullPointerException();

**final** E[] items = **this**.items;

**final** ReentrantLock lock = **this**.lock;

lock.lockInterruptibly(); // 获得锁的过程是可中断的。

**try** {

**try** {

**while** (count == items.length)

notFull.await(); // 等待队列不为空

} **catch** (InterruptedException ie) {

notFull.signal(); // 唤醒等待队列不为空的Condition

**throw** ie;

}

insert(e);// 插入元素

} **finally** {

lock.unlock();

}

}

### offer(E e, long timeout, TimeUnit unit)

**public** **boolean** offer(E e, **long** timeout, TimeUnit unit) **throws** InterruptedException {

**if** (e == **null**) **throw** **new** NullPointerException();

**long** nanos = unit.toNanos(timeout);

**final** ReentrantLock lock = **this**.lock;

lock.lockInterruptibly(); // 获取锁的过程可中断

**try** {

**for** (;;) {

**if** (count != items.length) {

insert(e);

**return** **true**;

}

**if** (nanos <= 0) // 超时，返回false

**return** **false**;

**try** {

nanos = notFull.awaitNanos(nanos); // 等待队列有空位

} **catch** (InterruptedException ie) {

notFull.signal(); // propagate to non-interrupted thread

**throw** ie;

}

}

} **finally** {

lock.unlock();

}

}

### poll()

获取并移除此队列的头，如果此队列为空，则返回 null。

**public** E poll() {

**final** ReentrantLock lock = **this**.lock;

lock.lock(); // 不可中断

**try** {

**if** (count == 0)

**return** **null**;

E x = extract();

**return** x;

} **finally** {

lock.unlock();

}

}

**private** E extract() {

**final** E[] items = **this**.items;

E x = items[takeIndex];

items[takeIndex] = **null**;

takeIndex = inc(takeIndex);

--count;

notFull.signal();

**return** x;

}

## SynchronousQueue

一种**阻塞队列**，其中每个插入操作必须等待另一个线程的对应移除操作，反之亦然。同步队列没有任何内部容量，甚至连一个队列的容量都没有。不能在同步队列上进行 peek，因为仅在试图要移除元素时，该元素才存在；除非另一个线程试图移除某个元素，否则也不能（使用任何方法）插入元素；也不能迭代队列，因为其中没有元素可用于迭代。队列的头是尝试添加到队列中的首个已排队插入线程的元素；如果没有这样的已排队线程，则没有可用于移除的元素并且 poll() 将会返回 null。对于其他 Collection 方法（例如 contains），SynchronousQueue 作为一个空 collection。此队列不允许 null 元素。

同步队列类似于CSP和Ada中使用的 rendezvous 信道。它**非常适合于传递性设计**，在这种设计中，在一个线程中运行的对象要将某些信息、事件或任务传递给在另一个线程中运行的对象，它就必须与该对象同步。

对于正在等待的生产者和使用者线程而言，此类支持可选的公平排序策略。默认情况下不保证这种排序。但是，使用公平设置为 true 所构造的队列可保证线程以FIFO的顺序进行访问。

此类及其迭代器实现 [Collection](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Collection.html) 和 [Iterator](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Iterator.html) 接口的所有可选方法。

### SynchronousQueue(boolean fair)

**public** SynchronousQueue(**boolean** fair) {

transferer = (fair)? **new** TransferQueue() : **new** TransferStack();

}

 创建一个具有指定公平策略的 SynchronousQueue.

TransferQueue、TransferStack和Transferer都是SynchronousQueue中的静态内部类，TransferQueue、TransferStack都继承了Transferer抽象类，Transferer中定义了transfer方法由实现类实现。

### Transferer

static abstract class Transferer {

// 执行一个put或take操作

abstract Object transfer(Object e, boolean timed, long nanos);

}

### TransferQueue

TransferQueue中定义了一个静态内部类QNode,用于承载队列中的节点功能

static final class QNode {

volatile QNode next; // 节点的下一个节点

volatile Object item;// 节点中的元素

volatile Thread waiter; // 控制阻塞/唤醒

final boolean isData;

QNode(Object item, boolean isData) {

this.item = item;

this.isData = isData;

}

static final AtomicReferenceFieldUpdater<QNode, QNode> *nextUpdater* = AtomicReferenceFieldUpdater.*newUpdater* (QNode.class, QNode.class, "next"); // QNode类的next原子字段更新器

boolean casNext(QNode cmp, QNode val) { // 原子更新next字段

return (next == cmp && *nextUpdater*.compareAndSet(this, cmp, val));

}

static final AtomicReferenceFieldUpdater<QNode, Object> *itemUpdater* = AtomicReferenceFieldUpdater.*newUpdater* (QNode.class, Object.class, "item");// QNode类的item原子字段更新器

boolean casItem(Object cmp, Object val) {// 原子更新item字段

return (item == cmp && *itemUpdater*.compareAndSet(this, cmp, val));

}

void tryCancel(Object cmp) {

*itemUpdater*.compareAndSet(this, cmp, this);

}

boolean isCancelled() {

return item == this;

}

boolean isOffList() {

return next == this;

}

}

在TransferQueue定义了队列的头节点head、尾节点tail等

/\*\* Head of queue \*/

**transient** **volatile** QNode head;

/\*\* Tail of queue \*/

**transient** **volatile** QNode tail;

/\*\* Reference to a cancelled node that might not yet have been unlinked from queue because it was the last inserted node when \* it cancelled. \*/

**transient** **volatile** QNode cleanMe;

TransferQueue() {

QNode h = **new** QNode(**null**, **false**); // 初始化一个傀儡(dummy)节点.

head = h;

tail = h;

}

#### transfer()

Object transfer(Object e, **boolean** timed, **long** nanos) {

QNode s = **null**; // constructed/reused as needed

**boolean** isData = (e != **null**);

**for** (;;) {

QNode t = tail;

QNode h = head;

**if** (t == **null** || h == **null**)// 判断TransferQueue是否初始化完毕

**continue**; // 如果没有初始化完毕，则自旋。

**if** (h == t || t.isData == isData) { // 队列为空

QNode tn = t.next;

**if** (t != tail) // inconsistent read

**continue**;

**if** (tn != **null**) { // lagging tail

advanceTail(t, tn);

**continue**;

}

**if** (timed && nanos <= 0) // can't wait

**return** **null**;

**if** (s == **null**)

s = **new** QNode(e, isData);

**if** (!t.casNext(**null**, s)) // failed to link in

**continue**;

advanceTail(t, s); // swing tail and wait

Object x = awaitFulfill(s, e, timed, nanos);

**if** (x == s) { // wait was cancelled

clean(t, s);

**return** **null**;

}

**if** (!s.isOffList()) { // not already unlinked

advanceHead(t, s); // unlink if head

**if** (x != **null**) // and forget fields

s.item = s;

s.waiter = **null**;

}

**return** (x != **null**)? x : e;

} **else** { // complementary-mode

QNode m = h.next; // node to fulfill

**if** (t != tail || m == **null** || h != head)

**continue**; // inconsistent read

Object x = m.item;

**if** (isData == (x != **null**) || // m already fulfilled

x == m || // m cancelled

!m.casItem(x, e)) { // lost CAS

advanceHead(h, m); // dequeue and retry

**continue**;

}

advanceHead(h, m); // successfully fulfilled

LockSupport.*unpark*(m.waiter);

**return** (x != **null**)? x : e;

}

}

}

## LinkedBlockingQueue

一个基于已链接节点的、范围任意的BlockingQueue。此队列按 FIFO（先进先出）排序元素。队列的头部是在队列中时间最长的元素。队列的尾部是在队列中时间最短的元素。新元素插入到队列的尾部，并且队列获取操作会获得位于队列头部的元素。链接队列的吞吐量通常要高于基于数组的队列，但是在大多数并发应用程序中，其可预知的性能要低。

可选的容量范围构造方法参数作为防止队列过度扩展的一种方法。如果未指定容量，则它等于 Integer.MAX\_VALUE。除非插入节点会使队列超出容量，否则每次插入后会动态地创建链接节点。

此类及其迭代器实现 [Collection](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Collection.html) 和 [Iterator](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Iterator.html) 接口的所有可选 方法。

### LinkedBlockingQueue(int capacity)

**public** LinkedBlockingQueue(**int** capacity) {

**if** (capacity <= 0) **throw** **new** IllegalArgumentException();

**this**.capacity = capacity;

last = head = **new** Node<E>(**null**);

}

**static** **class** Node<E> {

/\*\* The item, volatile to ensure barrier separating write and read \*/

**volatile** E item;

Node<E> next;

Node(E x) { item = x; }

}

### offer(E e)

**public** **boolean** offer(E e) { // 此方法不会因为队列已满而造成阻塞

**if** (e == **null**) **throw** **new** NullPointerException();

**final** AtomicInteger count = **this**.count; // 链表中元素的个数

**if** (count.get() == capacity) // 如果链表已满，返回false

**return** **false**;

**int** c = -1;

**final** ReentrantLock putLock = **this**.putLock;

putLock.lock(); // 不可中断的锁

**try** {

**if** (count.get() < capacity) {

insert(e);

c = count.getAndIncrement();

**if** (c + 1 < capacity)

notFull.signal(); // 如果还可以插入元素则唤醒putLock 锁的notFull条件

}

} **finally** {

putLock.unlock();

}

**if** (c == 0) // 队列中已经元素了，唤醒takeLock上的notEmpty Condition阻塞

signalNotEmpty();

**return** c >= 0;

}

**private** **void** insert(E x) {

last = **last.next** = **new** Node<E>(x);

}

先建立最后一个节点与新节点的关系。然后将新节点替换最后一个节点。

### take()

**public** E take() **throws** InterruptedException { //在元素变得可用之前一直等待

E x;

**int** c = -1;

**final** AtomicInteger count = **this**.count;

**final** ReentrantLock takeLock = **this**.takeLock;

takeLock.lockInterruptibly();

**try** {

**try** {

**while** (count.get() == 0) // 队列为空，等待队列不为空

notEmpty.await();

} **catch** (InterruptedException ie) {

notEmpty.signal(); // propagate to a non-interrupted thread

**throw** ie;

}

x = extract(); // 从队列中获取元素

c = count.getAndDecrement(); // 数量减1

**if** (c > 1)

notEmpty.signal();

} **finally** {

takeLock.unlock();

}

**if** (c == capacity)

signalNotFull();// 唤醒putLock上的notFull Condition

**return** x;

}

**private** E extract() {

Node<E> first = head.next; // 取出头节点的下一个节点

head = first;// 将头节点的下一个节点赋值为新头节点

E x = first.item;// 取头节点的元素

first.item = **null**;// Help GC

**return** x;

}

### drainTo(Collection<? super E> c)

**public** **int** drainTo(Collection<? **super** E> c) {

**if** (c == **null**)

**throw** **new** NullPointerException();

**if** (c == **this**)

**throw** **new** IllegalArgumentException();

Node<E> first;

fullyLock(); // 锁住takeLock和putLock锁

**try** {

first = head.next; // 获取第一个节点

head.next = **null**; // 将头节点与first节点及后面的节点脱离

**assert** head.item == **null**; // 断言，如果为true则继续运行，否则抛出AssertionError错误

last = head;// 将last指针指向head节点

**if** (count.getAndSet(0) == capacity) // 将节点数量计数值设置为0

notFull.signalAll();

} **finally** {

fullyUnlock();

}

**int** n = 0;

**for** (Node<E> p = first; p != **null**; p = p.next) { // 迭代所有元素

c.add(p.item);

p.item = **null**;// help GC

++n;

}

**return** n;

}

### drainTo(Collection<? super E> c, int maxElements)

**public** **int** drainTo(Collection<? **super** E> c, **int** maxElements) {

**if** (c == **null**)

**throw** **new** NullPointerException();

**if** (c == **this**)

**throw** **new** IllegalArgumentException();

fullyLock();

**try** {

**int** n = 0;

Node<E> p = head.next;

**while** (p != **null** && n < maxElements) {

c.add(p.item);

p.item = **null**;

p = p.next;

++n;

}

**if** (n != 0) {

head.next = p;

**assert** head.item == **null**;

**if** (p == **null**)

last = head;

**if** (count.getAndAdd(-n) == capacity)

notFull.signalAll();

}

**return** n;

} **finally** {

fullyUnlock();

}

}

## PriorityBlockingQueue

一个**无界阻塞队列**，它使用与类 PriorityQueue 相同的顺序规则，并且提供了阻塞获取操作。虽然此队列逻辑上是无界的，但是资源被耗尽时试图执行 add 操作也将失败（导致 OutOfMemoryError）。此类不允许使用 null 元素。依赖自然顺序的优先级队列也不允许插入不可比较的对象（这样做会导致抛出 ClassCastException）。

此类及其迭代器可以实现 [Collection](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Collection.html) 和 [Iterator](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Iterator.html) 接口的所有可选 方法。[iterator()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\PriorityBlockingQueue.html#iterator()) 方法中提供的迭代器并不保证以特定的顺序遍历 PriorityBlockingQueue 的元素。如果需要有序地进行遍历，则应考虑使用 **Arrays.sort(pq.toArray())**。此外，可以使用方法 drainTo 按优先级顺序移除 全部或部分元素，并将它们放在另一个 collection 中。

在此类上进行的操作不保证具有**同等优先级**的元素的顺序。如果需要实施某一排序，那么可以定义自定义类或者比较器，比较器可使用修改键断开主优先级值之间的联系。

### PriorityBlockingQueue()

用默认的初始容量 (**11**) 创建一个 PriorityBlockingQueue，并根据元素的**自然顺序**对其元素进行排序。

**public** PriorityBlockingQueue() {

q = **new** PriorityQueue<E>();

}

其实在PriorityQueue中创建的是一个Object[]的数组。

**public** PriorityQueue(**int** initialCapacity, Comparator<? **super** E> comparator) {

**if** (initialCapacity < 1)

**throw** **new** IllegalArgumentException();

**this**.queue = **new** Object[initialCapacity];

**this**.comparator = comparator;

}

### PriorityBlockingQueue(int initialCapacity)

使用指定的初始容量创建一个 PriorityBlockingQueue，并根据元素的**自然顺序**对其元素进行**排序**。

**public** PriorityBlockingQueue(**int** initialCapacity) {

q = **new** PriorityQueue<E>(initialCapacity, **null**);

}

### PriorityBlockingQueue(int,Comparator<? super [E](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\PriorityBlockingQueue.html)> )

使用指定的初始容量创建一个 PriorityBlockingQueue，并根据**指定的比较器**对其元素进行排序。

### PriorityBlockingQueue(Collection<? extends E> c)

创建一个包含指定 collection 元素的 PriorityBlockingQueue。如果指定collection是一个 [SortedSet](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\SortedSet.html) 或 [PriorityQueue](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\PriorityQueue.html)，则此优先级队列将按照相同顺序进行排序。否则，此优先级队列将根据此元素的[自然顺序](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Comparable.html)进行排序。

**public** PriorityBlockingQueue(Collection<? **extends** E> c) {

q = **new** PriorityQueue<E>(c);

}

**public** PriorityQueue(Collection<? **extends** E> c) {

initFromCollection(c); 初始化队列

**if** (c **instanceof** SortedSet)

comparator = (Comparator<? **super** E>) ((SortedSet<? **extends** E>)c).comparator();

**else** **if** (c **instanceof** PriorityQueue)

comparator = (Comparator<? **super** E>) ((PriorityQueue<? **extends** E>)c).comparator();

**else** { // 自然排序

comparator = **null**;

heapify(); // 此段方法值得考究

}

}

**private** **void** heapify() {

**for** (**int** i = (size >>> 1) - 1; i >= 0; i--)

siftDown(i, (E) queue[i]);

}

**private** **void** siftDown(**int** k, E x) {

**if** (comparator != **null**)

siftDownUsingComparator(k, x);

**else**

siftDownComparable(k, x);

}

**private** **void** siftDownComparable(**int** k, E x) {

Comparable<? **super** E> key = (Comparable<? **super** E>)x; // 使用元素本身的进行排序

**int** half = size >>> 1; // loop while a non-leaf

**while** (k < half) {

**int** child = (k << 1) + 1; // assume left child is least

Object c = queue[child];

**int** right = child + 1;

**if** (right < size && ((Comparable<? **super** E>) c).compareTo((E) queue[right]) > 0)

c = queue[child = right];

**if** (key.compareTo((E) c) <= 0)

**break**;

queue[k] = c;

k = child;

}

queue[k] = key;

}

**private** **void** siftDownUsingComparator(**int** k, E x) { // 使用初始化时的Collection容器进行排序

**int** half = size >>> 1;

**while** (k < half) {

**int** child = (k << 1) + 1;

Object c = queue[child];

**int** right = child + 1;

**if** (right < size && comparator.compare((E) c, (E) queue[right]) > 0)

c = queue[child = right];

**if** (comparator.compare(x, (E) c) <= 0)

**break**;

queue[k] = c;

k = child;

}

queue[k] = x;

}

### offer(E)

将指定元素插入此优先级队列。

**public** **boolean** offer(E e) {

**final** ReentrantLock lock = **this**.lock;

lock.lock();

**try** {

**boolean** ok = q.offer(e);

**assert** ok;// 断言

notEmpty.signal();

**return** **true**;

} **finally** {

lock.unlock();

}

}

从方法中可以看出，内部offer的操作交给了其父类PriorityQueue实现：

**public** **boolean** offer(E e) {

**if** (e == **null**)

**throw** **new** NullPointerException();

modCount++;

**int** i = size;

**if** (i >= queue.length) // 如果当前队列已满，则扩容

grow(i + 1);

size = i + 1;

**if** (i == 0)

queue[0] = e;

**else**

siftUp(i, e);// 排序

**return** **true**;

}

**private** **void** siftUp(**int** k, E x) {

**if** (comparator != **null**)

siftUpUsingComparator(k, x);

**else**

siftUpComparable(k, x);

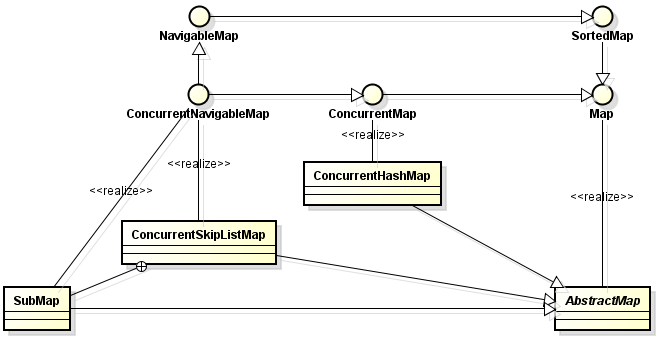
}

# BlockingDeque

# ConcurrentMap

提供其他原子 putIfAbsent、remove、replace 方法的 Map。

内存一致性效果：当存在其他并发collection时，将对象放入 ConcurrentMap 之前的线程中的操作 [happen-before](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\package-summary.html#MemoryVisibility) 随后通过另一线程从ConcurrentMap 中访问或移除该元素的操作。



ConcurrentMap类图

## ConcurrentNavigableMap

支持 NavigableMap 操作，且以递归方式支持其可导航子映射的 ConcurrentMap。

## ConcurrentHashMap

支持获取的完全并发和更新的所期望可调整并发的哈希表。此类遵守与 Hashtable 相同的功能规范，并且包括对应于 Hashtable 的每个方法的方法版本。不过，**尽管所有操作都是线程安全的**，但**获取操作不必锁定**，并且不支持以某种防止所有访问的方式锁定整个表。此类可以通过程序完全与Hashtable进行互操作，**这取决于其线程安全**，而与其同步细节无关。

获取操作（包括get）通常不会受阻塞，因此，可能与更新操作交迭（包括put和remove）。获取会影响最近完成的更新操作的结果。对于一些聚合操作，比如putAll和clear，并发获取可能只影响某些条目的插入和移除。类似地，在创建迭代器/枚举时或自此之后，Iterators和Enumerations 返回在某一时间点上影响哈希表状态的元素。它们不会抛出ConcurrentModificationException。不过，迭代器被设计成每次仅由一个线程使用。

这允许通过可选的**concurrencyLevel** 构造方法参数（默认值为 16）来引导更新操作之间的并发，该**参数用作内部调整大小的一个提示**。表是在内部进行分区的，试图允许指示无争用并发更新的数量。因为哈希表中的位置基本上是随意的，所以实际的并发将各不相同。理想情况下，应该选择一个尽可能多地容纳并发修改该表的线程的值。使用一个比所需要的值高很多的值可能会浪费空间和时间，而使用一个显然低很多的值可能导致线程争用。对数量级估计过高或估计过低通常都会带来非常显著的影响。当仅有一个线程将执行修改操作，而其他所有线程都只是执行读取操作时，才认为某个值是合适的。此外，重新调整此类或其他任何种类哈希表的大小都是一个相对较慢的操作，因此，在可能的时候，提供构造方法中期望表大小的估计值是一个好主意。

此类及其视图和迭代器实现了[Map](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Map.html)和[Iterator](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\Iterator.html)接口的所有可选方法。

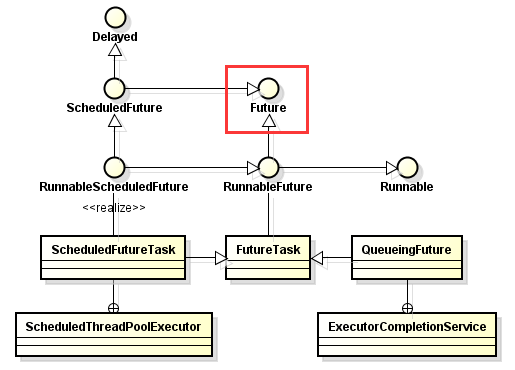
# Callable

返回结果并且可能抛出异常的任务。实现者定义了一个不带任何参数的叫做 call 的方法。

Callable 接口类似于 [Runnable](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Runnable.html)，两者都是为那些其实例可能被另一个线程执行的类设计的。但是**Runnable 不会返回结果**，并且**无法抛出经过检查的异常**。

[Executors](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Executors.html) 类包含一些从其他普通形式转换成 Callable 类的实用方法。

# Future



Future类图

Future 表示**异步计算的结果**。它**提供了检查计算是否完成的方法**，以等待计算的完成，并获取计算的结果。计算完成后只能使用 get 方法来获取结果，如有必要，计算完成前可以**阻塞此方法**。取消则由 cancel 方法来执行。还提供了其他方法，以确定任务是正常完成还是被取消了。一旦计算完成，就不能再取消计算。如果为了可取消性而使用 Future 但又不提供可用的结果，则可以声明 Future<?> 形式类型、并返回 null 作为底层任务的结果。

Futrue下面定义了五个方法，如下：

cancel()：试图取消对此任务的执行。

get()： 如有必要，等待计算完成，然后获取其结果。

get(long,TimeUnit)： 如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。

isCancelled()：如果在任务正常完成前将其取消，则返回 true。

isDone()：如果任务已完成，则返回 true。

这五个方法的详细时间见FutureTask。

## FutureTask

可取消的异步计算。利用开始和取消计算的方法、查询计算是否完成的方法和获取计算结果的方法，此类提供了对 Future 的基本实现。仅在计算完成时才能获取结果；如果计算尚未完成，则阻塞 get 方法。一旦计算完成，就不能再重新开始或取消计算。

可使用 FutureTask 包装 [**Callable**](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Callable.html) 或 [**Runnable**](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Runnable.html) 对象。因为 FutureTask 实现了 Runnable，所以可将 FutureTask 提交给 [Executor](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Executor.html) 执行。

除了作为一个独立的类外，此类还提供了protected 功能，这在创建自定义任务类时可能很有用。

FutureTask提供了两个构造函数，分别是：

方式一：

**public** FutureTask(Callable<V> callable) {

**if** (callable == **null**)

**throw** **new** NullPointerException();

sync = **new** Sync(callable);

}

方式二：

**public** FutureTask(Runnable runnable, V result) {

sync = **new** Sync(Executors.*callable*(runnable, result));

}

当采用方式二使用Runnable，FutureTask会调用Executors.callable(Runnable,V)将Runnable封装成Callable。

但是最终还是创建了[Sync](#_Sync)对象。

### run()

除非已将此 Future 取消，否则将其设置为其计算的结果。

public void run() {

sync.innerRun();

}

从逻辑上可以看出，run方法将方法的处理交给了Sync的[innerRun](#_innerRun())()

### cancel(boolean)

**public** **boolean** cancel(**boolean** mayInterruptIfRunning) {

**return** sync.innerCancel(mayInterruptIfRunning);

}

试图取消对此任务的执行。如果任务已完成、或已取消，或者由于某些其他原因而无法取消，则此尝试将失败。当调用 cancel 时，如果调用成功，而此任务尚未启动，则此任务将永不运行。如果任务已经启动，则 mayInterruptIfRunning 参数确定**是否应该以试图停止任务的方式来中断执行此任务的线程**。

此方法返回后，对 [Future.isDone()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Future.html#isDone()) 的后续调用将始终返回 true。如果此方法返回 true，则对 [Future.isCancelled()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Future.html#isCancelled()) 的后续调用将始终返回true。

详细见[innerCancel(boolean mayInterruptIfRunning)](#_innerCancel(boolean_mayInterruptIfR)

### get()

**public** V get() **throws** InterruptedException, ExecutionException {

**return** sync.innerGet();

}

如有必要，等待计算完成，然后获取其结果。详细见[innerGet()](#_innerGet())。支持中断。

### get(long timeout, TimeUnit unit)

**public** V get(**long** timeout, TimeUnit unit) **throws** InterruptedException, ExecutionException, TimeoutException {

**return** sync.innerGet(unit.toNanos(timeout));

}

如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。

详细见[innerGet(long nanosTimeout)](#_innerGet(long_nanosTimeout))。

### isCancelled()

 如果在任务正常完成前将其取消，则返回 true。

### isDone()

如果任务已完成，则返回 true。 可能由于正常终止、异常或取消而完成，在所有这些情况中，此方法都将返回 true。

### runAndReset()

**protected** **boolean** runAndReset() {

**return** sync.innerRunAndReset();

}

### Sync

Sync实现了AbstractQueuedSynchronizer抽象类，用于FutureTask的同步控制，使用了AQS同步状态来代表FutureTask的运行状态。

Sync下定义了一下几种状态

/\*\* State value representing that task is ready to run \*/

**private** **static** **final** **int** *READY* = 0;

/\*\* State value representing that task is running \*/

**private** **static** **final** **int** *RUNNING* = 1;

/\*\* State value representing that task ran \*/

**private** **static** **final** **int** *RAN* = 2;

/\*\* State value representing that task was cancelled \*/

**private** **static** **final** **int** *CANCELLED* = 4;

#### innerRun()

**void** innerRun() {

**if** (!compareAndSetState(*READY*, *RUNNING*)) // 将Sync的状态设置为RUNNING

**return**;

runner = Thread.*currentThread*();

**if** (getState() == *RUNNING*) { // 如果当前状态为RUNNING

V result;

**try** {

result = callable.call(); // 获取结果

} **catch** (Throwable ex) {

setException(ex);

**return**;

}

set(result); // 详细见[innerSet(V v)](#_innerSet(V_v))

} **else** {

// 释放共享模式，详细见AbstractQueuedSynchronizer的doReleaseShared方法

releaseShared(0);

}

}

#### innerSet(V v)

**void** innerSet(V v) {

**for** (;;) {

**int** s = getState();

**if** (s == *RAN*) // 运行完毕

**return**;

**if** (s == *CANCELLED*) { // 取消

releaseShared(0);

**return**;

}

**if** (compareAndSetState(s, *RAN*)) {

result = v;

releaseShared(0);

done(); // 钩子，在ExecutorCompletionService中的内部类QueueingFuture中有实现。

**return**;

}

}

}

#### innerCancel(boolean mayInterruptIfRunning)

**boolean** innerCancel(**boolean** mayInterruptIfRunning) {

**for** (;;) {

**int** s = getState();

**if** (ranOrCancelled(s))

**return** **false**;

**if** (compareAndSetState(s, *CANCELLED*))

**break**;

}

**if** (mayInterruptIfRunning) {

Thread r = runner;

**if** (r != **null**)

r.interrupt();

}

releaseShared(0);

done();

**return** **true**;

}

1、调用ranOrCancelled方法判断当前任务的状态是运行完毕还是已经取消，如果运行完毕或已经取消，返回false。

**private** **boolean** ranOrCancelled(**int** state) {

**return** (state & (*RAN* | *CANCELLED*)) != 0;// RAN = 2,CANCELLED = 4

}

2、将任务的状态设置为CANCELLED[4]

3、如果传入的参数mayInterruptIfRunning为true，这中断当前执行线程。

4、释放共享模式，调用done()方法。

#### innerGet()

V innerGet() **throws** InterruptedException, ExecutionException {

acquireSharedInterruptibly(0); //

**if** (getState() == *CANCELLED*)

**throw** **new** CancellationException();

**if** (exception != **null**)

**throw** **new** ExecutionException(exception);

**return** result;

}

1、调用acquireSharedInterruptibly方法可能会阻塞当前想获得结果的线程，并将当前线程加入到共享模式中等到共享锁的释放。

**public** **final** **void** acquireSharedInterruptibly(**int** arg) **throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

// 调用FutureTask的tryAcquireShared方法，判断任务的状态

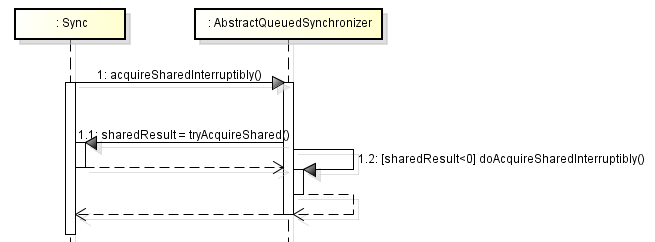
// 小于0表示状态为READY或RUNNING

**if** (tryAcquireShared(arg) < 0)

doAcquireSharedInterruptibly(arg); // 将当前线程加入共享模式等待队列，在innerRun中涉及了共享模式锁释放

}

2、当共享锁释放后，判断当前执行任务的状态，如果状态为CANCELLED(取消)，则抛出异常；如果任务执行过程中抛出了异常，则将异常包装成ExecutionException异常抛出；如果无异常情况，则返回计算结果



FutureTask.Sync.innerGet()

#### innerGet(long nanosTimeout)

V innerGet(**long** nanosTimeout) **throws** InterruptedException, ExecutionException, TimeoutException {

**if** (!tryAcquireSharedNanos(0, nanosTimeout))

**throw** **new** TimeoutException();

**if** (getState() == *CANCELLED*)

**throw** **new** CancellationException();

**if** (exception != **null**)

**throw** **new** ExecutionException(exception);

**return** result;

}

在等待超时仍未获得结果时，将抛出TimeoutException异常。

**public** **final** **boolean** tryAcquireSharedNanos(**int** arg, **long** nanosTimeout) **throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

**return** tryAcquireShared(arg) >= 0 || doAcquireSharedNanos(arg, nanosTimeout);// 任务状态为READY或RUNNING

}

#### innerRunAndReset()

**boolean** innerRunAndReset() {

**if** (!compareAndSetState(*READY*, *RUNNING*))

**return** **false**;

**try** {

runner = Thread.*currentThread*();

**if** (getState() == *RUNNING*)

callable.call(); // don't set result

runner = **null**;

**return** compareAndSetState(*RUNNING*, *READY*);

} **catch** (Throwable ex) {

setException(ex);

**return** **false**;

}

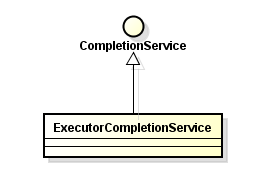
}

# CompletionService

将生产新的异步任务与使用已完成任务的结果**分离开来**的服务。生产者 submit 执行的任务。使用者 take 已完成的任务，并按照完成这些任务的顺序处理它们的结果。例如，CompletionService 可以用来管理异步 IO ，执行读操作的任务作为程序或系统的一部分提交，然后，当完成读操作时，会在程序的不同部分执行其他操作，执行操作的顺序可能与所请求的顺序不同。

通常，CompletionService 依赖于一个单独的 [Executor](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Executor.html) 来实际执行任务，在这种情况下，CompletionService 只管理一个内部完成队列。[ExecutorCompletionService](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\ExecutorCompletionService.html) 类提供了此方法的一个实现。

内存一致性效果：线程中向 CompletionService 提交任务之前的操作 [happen-before](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\package-summary.html#MemoryVisibility) 该任务执行的操作，后者依次 happen-before 紧跟在从对应take() 成功返回的操作。



ComletionService类图

# ExecutorCompletionService

使用提供的 [**Executor**](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Executor.html) 来执行任务的 [CompletionService](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CompletionService.html)。此类将安排那些完成时提交的任务，把它们放置在可使用 take 访问的队列上。该类非常轻便，适合于在执行几组任务时临时使用。

## ExecutorCompletionService(Executor)

**public** ExecutorCompletionService(Executor executor) {

**if** (executor == **null**)

**throw** **new** NullPointerException();

**this**.executor = executor;

**this**.aes = (executor **instanceof** AbstractExecutorService) ? (AbstractExecutorService) executor : **null**;

**this**.completionQueue = **new** LinkedBlockingQueue<Future<V>>();

}

使用为执行基本任务而提供的执行程序创建一个 ExecutorCompletionService，并将 [LinkedBlockingQueue](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\LinkedBlockingQueue.html) 作为完成队列。

## ExecutorCompletionService(Executor,BlockingQueue<Future<V>>)

**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;

}

## submit(…)

ExecutorCompletionService提供了两种submit的实现，如下：

方式一

**public** Future<V> submit(Callable<V> task) {

**if** (task == **null**) **throw** **new** NullPointerException();

RunnableFuture<V> f = newTaskFor(task);

executor.execute(**new** QueueingFuture(f));

**return** f;

}

方式二

**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;

}

提交要执行的Runnable任务，并返回一个表示任务完成的Future。

不管是哪种方式，最后都会将Callable/Runnable封装成RunnableFuture，然后再将RunnableFuture封装成QueueingFuture，QueueingFuture是ExecutorCompletionService中的一个私有内部类,其继承了FutureTask。

**private** **class** QueueingFuture **extends** FutureTask<Void> {

QueueingFuture(RunnableFuture<V> task) {

**super**(task, **null**);

**this**.task = task;

}

**protected** **void** done() { completionQueue.add(task); }

**private** **final** Future<V> task;

}

其中done()方法将在FutureTask中的run()方法被调用，将任务执行的结果保存到完成队列completionQueue中。

ExecutorCompletionService中的poll()、pool(long,TimeUnit)和take()实现其实就是调用了BlockingQueue对象方法的实现，详细见BlockingQueue部分的讲解。

# Executors

# CountDownLatch

一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

用**给定的计数初始化CountDownLatch**。由于调用了countDown()方法，**所以在当前计数到达零之前，await方法会一直受阻塞**。**之后**，会释放所有等待的线程，await的所有后续调用都将立即返回。这种现象只出现一次——**计数无法被重置**。如果需要重置计数，请考虑使用**CyclicBarrier**。

CountDownLatch是一个通用同步工具，它有很多用途。将计数1初始化的CountDownLatch用作一个简单的开/关锁存器，或入口：在通过调用countDown()的线程打开入口前，所有调用 await 的线程都一直在入口处等待。用N初始化的CountDownLatch可以使一个线程在N个线程完成某项操作之前一直等待，或者使其在某项操作完成N次之前一直等待。

CountDownLatch的一个有用特性是，它不要求调用countDown方法的线程等到计数到达零时才继续，而在所有线程都能通过之前，它只是阻止任何线程继续通过一个 await。

示例用法： 下面给出了两个类，其中一组worker线程使用了两个倒计数锁存器：

第一个类是一个启动信号，在driver为继续执行worker做好准备之前，它会阻止所有的worker继续执行。

第二个类是一个完成信号，它允许driver在完成所有worker之前一直等待。

class Driver { // ...

void main() throws InterruptedException {

CountDownLatch startSignal = new CountDownLatch(1);

CountDownLatch doneSignal = new CountDownLatch(N);

for (int i = 0; i < N; ++i) // create and start threads

new Thread(new Worker(startSignal, doneSignal)).start();

doSomethingElse();// 做一些准备工作

startSignal.countDown();// 解除每个任务run()方法中await(),任务线程开始真正的执行任务

doSomethingElse();// 做主线程(调用线程自己的事情)

doneSignal.await();// 等待所有任务线程执行完各自的任务，在各任务执行完，会countDown计数值

}

}

class Worker implements Runnable {

private final CountDownLatch startSignal;

private final CountDownLatch doneSignal;

Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {

this.startSignal = startSignal;

this.doneSignal = doneSignal;

}

public void run() {

try {

startSignal.await();// 等待所有其他任务线程准备工作完成，主线程countDown计数值

doWork();

doneSignal.countDown();// 发出一个信号量，表名此任务已经完成。

} catch (InterruptedException ex) {} // return;

}

void doWork() { ... }

}

## CountDownLatch

**public** CountDownLatch(**int** count) {

**if** (count < 0) **throw** **new** IllegalArgumentException("count < 0");

**this**.sync = **new** Sync(count);

}

**private** **static** **final** **class** Sync **extends** AbstractQueuedSynchronizer {

**private** **static** **final** **long** *serialVersionUID* = 4982264981922014374L;

Sync(**int** count) {

setState(count);

}

**int** getCount() {

**return** getState();

}

**protected** **int** tryAcquireShared(**int** acquires) {

**return** getState() == 0? 1 : -1;

}

**protected** **boolean** tryReleaseShared(**int** releases) {

**for** (;;) {

**int** c = getState();

**if** (c == 0)

**return** **false**;

**int** nextc = c-1;

**if** (compareAndSetState(c, nextc))

**return** nextc == 0;

}

}

}

从CountDownLatch中可以看出，其实其内部核心还是AbstractQueuedSynchronizer，内部采用共享锁模式。

构造器直接调用AbstractQueuedSynchronizer的setState(int)方法设置共享锁的同步状态。

## await()

**public** **void** await() **throws** InterruptedException {

sync.acquireSharedInterruptibly(1);

}

使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。

如果当前计数为零，则此方法立即返回。

如果当前计数大于零，则出于线程调度目的，将禁用当前线程，且在发生以下两种情况之一前，该线程将一直处于休眠状态：

a、由于调用 countDown() 方法，计数到达零；或者

b、其他某个线程中断当前线程。

如果当前线程：

a、在进入此方法时已经设置了该线程的中断状态；

b、在等待时被中断，

则抛出 InterruptedException，并且清除当前线程的已中断状态。

我们还是继续看看AbstractQueuedSynchronizer中acquireSharedInterruptibly(int)方法的实现。

**public** **final** **void** acquireSharedInterruptibly(**int** arg) **throws** InterruptedException {

**if** (Thread.*interrupted*()) // 当前线程已经被中断

**throw** **new** InterruptedException();

// 回调CountDownLatch内部类Sync的tryAcquireShared(int)方法

**if** (tryAcquireShared(arg) < 0) // 小于0表示锁未释放

doAcquireSharedInterruptibly(arg);// 阻塞当前线程，直至共享锁释放

}

**CountDownLatch.Sync.tryAcquireShared(int)方法实现**

**protected** **int** tryAcquireShared(**int** acquires) {

**return** getState() == 0? 1 : -1;// 同步状态为0(锁已释放)返回1，否则返回-1

}

**AbstractQueuedSynchronizer.doAcquireSharedInterruptibly(int)方法实现。**

**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); // 同步状态为0(锁已释放)返回1，否则返回-1

**if** (r >= 0) {

setHeadAndPropagate(node, r); // 释放锁

p.next = **null**; // help GC

failed = **false**;

**return**;

}

}

// 挂起当前线程，支持中断

**if** (*shouldParkAfterFailedAcquire*(p, node) && parkAndCheckInterrupt())

**throw** **new** InterruptedException();

}

} **finally** {

**if** (failed)

cancelAcquire(node);

}

}

## await(long timeout,TimeUnit unit)

使当前线程在**锁存器倒计数至零之前一直等待**，除非线程被中断或超出了指定的等待时间。

如果当前计数为零，则此方法立刻返回true值。

如果当前计数大于零，则出于线程调度目的，将禁用当前线程，且在发生以下三种情况之一前，该线程将一直处于休眠状态：

a、由于调用 countDown() 方法，计数到达零；

b、其他某个线程中断当前线程；

c、已超出指定的等待时间。

如果计数到达零，则该方法返回**true**值。

如果当前线程：

a、在进入此方法时已经设置了该线程的中断状态；

b、在等待时被中断，

则抛出InterruptedException，并且清除当前线程的已中断状态。

如果超出了指定的等待时间，则返回值为 false。如果该时间小于等于零，则此方法根本不会等待。

**public** **boolean** await(**long** timeout, TimeUnit unit) **throws** InterruptedException {

**return** sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));

}

await(…)方法的内部实现还是由AbstractQueuedSynchronizer.tryAcquireSharedNanos(int,long)方法实现。

**public** **final** **boolean** tryAcquireSharedNanos(**int** arg, **long** nanosTimeout) **throws** InterruptedException {

**if** (Thread.*interrupted*()) // 线程已被中断，抛出异常，并恢复中断状态

**throw** **new** InterruptedException();

// 调用CountDownLatch.Sync.tryAcquireShared()方法，判断锁是否释放，如果释放立即返回true，

// 否则调用AbstractQueuedSynchronizer.doAcquireSharedNanos(int,long)获取锁。

**return** tryAcquireShared(arg) >= 0 || doAcquireSharedNanos(arg, nanosTimeout);

}

AbstractQueuedSynchronizer.doAcquireSharedNanos(int,long)实现

**private** **boolean** doAcquireSharedNanos(**int** arg, **long** nanosTimeout) **throws** InterruptedException {

**long** lastTime = System.*nanoTime*();

**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**;

}

}

**if** (nanosTimeout <= 0) // 超时

**return** **false**;

**if** (*shouldParkAfterFailedAcquire*(p, node) && nanosTimeout > *spinForTimeoutThreshold*)

LockSupport.*parkNanos*(**this**, nanosTimeout); // 指定时间内挂起线程

**long** now = System.*nanoTime*();

nanosTimeout -= now - lastTime;

lastTime = now;

**if** (Thread.*interrupted*()) // 支持中断

**throw** **new** InterruptedException();

}

} **finally** {

**if** (failed)

cancelAcquire(node);

}

}

## countDown()

递减锁存器的计数，如果计数到达零，则释放所有等待的线程。

如果当前计数大于零，则将计数减少。如果新的计数为零，出于线程调度目的，将重新启用所有的等待线程。

如果当前计数等于零，则不发生任何操作。

**public** **void** countDown() {

sync.releaseShared(1);

}

内部调用CountDownLatch.Sync.releaseShared(int)完成锁存器计数值的递减工作。

**public** **final** **boolean** releaseShared(**int** arg) {

// 如果锁存器的值为0，则返回false，表示不发生任何操作。

// 如果锁存器的值不为0，则将锁存器的值减一，返回锁存器值是否等于0

**if** (tryReleaseShared(arg)) { // 如果锁存器的值为0

// 调用AbstractQueuedSynchronizer.doReleaseShared()重启所有等待线程。

doReleaseShared();

**return** **true**;

}

**return** **false**;

}

AbstractQueuedSynchronizer.doReleaseShared()实现。

**private** **void** doReleaseShared() {

**for** (;;) {

Node h = head;

**if** (h != **null** && h != tail) {

**int** ws = h.waitStatus;

**if** (ws == Node.*SIGNAL*) {// 表示当前节点的猴急节点线程需要被唤醒

**if** (!*compareAndSetWaitStatus*(h, Node.*SIGNAL*, 0))

**continue**;

unparkSuccessor(h); // 唤醒挂起的线程

} **else** **if** (ws == 0 && !*compareAndSetWaitStatus*(h, 0, Node.*PROPAGATE*))

**continue**;

}

**if** (h == head)

**break**;

}

}

# CyclicBarrier

一个同步辅助类，它允许一组线程互相等待，直到到达某个**公共屏障点 (common barrier point)**。在涉及**一组固定大小的线程**的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环的 barrier。

CyclicBarrier支持一个可选的Runnable命令，在一组线程中的最后一个线程到达之后（但在释放所有线程之前），该命令只在每个屏障点运行一次。若在继续所有参与线程之前更新共享状态，此屏障操作很有用。

如果屏障操作在执行时不依赖于正挂起的线程，则线程组中的任何线程在获得释放时都能执行该操作。为方便此操作，每次调用 [await()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#await()) 都将返回能到达屏障处的线程的索引。然后，您可以选择哪个线程应该执行屏障操作，例如：

if (barrier.await() == 0) {

// log the completion of this iteration

}

对于失败的同步尝试，CyclicBarrier使用了一种**要么全部 要么全不 (all-or-none)** 的破坏模式：如果因为中断、失败或者超时等原因，导致线程过早地离开了屏障点，那么在该屏障点等待的其他所有线程也将通过 BrokenBarrierException（如果它们几乎同时被中断，则用 [InterruptedException](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\InterruptedException.html)）以反常的方式离开。

内存一致性效果：线程中调用await()之前的操作 [happen-before](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\package-summary.html#MemoryVisibility)那些是屏障操作的一部份的操作，后者依次 happen-before 紧跟在从另一个线程中对应 await() 成功返回的操作。

## CyclicBarrier(int parties, Runnable barrierAction)

**public** CyclicBarrier(**int** parties, Runnable barrierAction) {

**if** (parties <= 0) **throw** **new** IllegalArgumentException();

**this**.parties = parties;

**this**.count = parties;

**this**.barrierCommand = barrierAction;

}

创建一个新的CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，并在启动barrier 时执行给定的**屏障操作**，**该操作由最后一个进入barrier的线程执行**。

参数：

parties - 在启动 barrier 前必须调用 [await()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#await()) 的线程数

barrierAction - 在启动 barrier 时执行的命令；如果不执行任何操作，则该参数为 null

## await()

在所有参与者都已经在此barrier上调用await方法之前，将一直等待。

如果当前线程不是将到达的最后一个线程，出于调度目的，将禁用它，且在发生以下情况之一前，该线程将一直处于休眠状态：

a、最后一个线程到达；

b、其他某个线程[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())当前线程；

c、其他某个线程[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())另一个等待线程；

d、其他某个线程在等待 barrier 时超时；

e、其他某个线程在此 barrier 上调用 [reset()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#reset())。

如果当前线程：

a、在进入此方法时已经设置了该线程的中断状态；

b、在等待时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())

则抛出 [InterruptedException](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\InterruptedException.html)，并且清除当前线程的已中断状态。

如果在线程处于等待状态时barrier被[reset()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#reset())，或者在调用await时barrier[被损坏](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#isBroken())，抑或任意一个线程正处于等待状态，则抛出BrokenBarrierException异常。

如果任何线程在等待时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())，则其他所有等待线程都将抛出[BrokenBarrierException](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\BrokenBarrierException.html) 异常，并将barrier置于损坏状态。

如果当前线程是**最后一个将要到达的线程**，并且构造方法中提供了一个非空的**屏障操作**[Runnable]，则在允许其他线程继续运行之前，当前线程将**运行该操作[屏障操作]**。如果在执行屏障操作过程中发生异常，则该异常将传播到当前线程中，并将 barrier 置于损坏状态。

返回：

到达的当前线程的索引，其中，索引getParties()-1指示将到达的第一个线程，零指示最后一个到达的线程。

抛出：

InterruptedException - 如果当前线程在等待时被中断

BrokenBarrierException - 如果另一个 线程在当前线程等待时被中断或超时，或者重置了barrier，或者在调用 await 时 barrier 被损坏，抑或由于异常而导致屏障操作（如果存在）失败。

**public** **int** await() **throws** InterruptedException, BrokenBarrierException {

**try** {

**return** dowait(**false**, 0L);

} **catch** (TimeoutException toe) {

**throw** **new** Error(toe); // cannot happen;

}

}

等待逻辑由[dowait(boolean timed, long nanos)](#_dowait(boolean_timed,_long)，由于不支持等待，这里会catch掉TimeoutException异常，并将其转换为系统级别的错误。

## dowait(boolean timed, long nanos)

**private** **int** dowait(**boolean** timed, **long** nanos) **throws** InterruptedException, BrokenBarrierException, TimeoutException {

**final** ReentrantLock lock = **this**.lock; // 使用可重入的互斥锁

lock.lock();

**try** {

**final** Generation g = generation;

**if** (g.broken) // generation标注此屏障是否处于受损状态

**throw** **new** BrokenBarrierException();

**if** (Thread.*interrupted*()) {// 线程中断

// 打破屏障。重置计数器，唤醒等待线程，将generation.broken设置为true。

breakBarrier();

**throw** **new** InterruptedException();

}

**int** index = --count;

**if** (index == 0) { // 最后一个任务到达，跨越屏障点

**boolean** ranAction = **false**;

**try** {

**final** Runnable command = barrierCommand;

**if** (command != **null**)

command.run(); // 最后一个到达现场执行**屏障操作**

ranAction = **true**;

nextGeneration(); // 进入下一次同步

**return** 0;

} **finally** {

**if** (!ranAction)

breakBarrier();

}

}

// loop until tripped, broken, interrupted, or timed out

**for** (;;) {

**try** {

**if** (!timed)

trip.await(); // 一直等待其他任务

**else** **if** (nanos > 0L)

nanos = trip.awaitNanos(nanos); // 等待指定时间

} **catch** (InterruptedException ie) {

**if** (g == generation && ! g.broken) {

breakBarrier();

**throw** ie;

} **else** {

Thread.*currentThread*().interrupt();

}

}

**if** (g.broken)

**throw** **new** BrokenBarrierException();

**if** (g != generation)

**return** index;

**if** (timed && nanos <= 0L) {

breakBarrier();

**throw** **new** TimeoutException();

}

}

} **finally** {

lock.unlock();

}

}

## await(long timeout, TimeUnit unit)

在所有[参与者](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#getParties())都已经在此屏障上调用await方法之前将一直等待,或者超出了指定的等待时间。

如果当前线程不是将到达的最后一个线程，出于调度目的，将禁用它，且在发生以下情况之一前，该线程将一直处于休眠状态：

a、最后一个线程到达；

b、超出指定的超时时间；

c、其他某个线程[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())当前线程；

d、其他某个线程[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())另一个等待线程；

e、其他某个线程在等待 barrier 时超时；

f、其他某个线程在此 barrier 上调用 [reset()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#reset())。

如果当前线程：

a、在进入此方法时已经设置了该线程的中断状态；

b、在等待时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())

则抛出InterruptedException，并且清除当前线程的已中断状态。

如果超出指定的等待时间，则抛出TimeoutException 异常。如果该时间小于等于零，则此方法根本不会等待。

如果在线程处于等待状态时 barrier 被 [reset()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#reset())，或者在调用 await 时 barrier [被损坏](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\CyclicBarrier.html#isBroken())，亦或任意一个线程正处于等待状态，则抛出BrokenBarrierException 异常。

如果任何线程在等待时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())，则其他所有等待线程都将抛出BrokenBarrierException，并将屏障置于损坏状态。

如果当前线程是最后一个将要到达的线程，并且构造方法中提供了一个非空的屏障操作，则在允许其他线程继续运行之前，当前线程将运行该操作。如果在执行屏障操作过程中发生异常，则该异常将传播到当前线程中，并将barrier置于损坏状态。

参数：

timeout - 等待 barrier 的时间

unit - 超时参数的时间单位

返回：

到达的当前线程的索引，其中，索引getParties()-1 指示第一个将要到达的线程，零指示最后一个到达的线程.

# Semaphore

一个计数信号量。从概念上讲，**信号量**维护了一个**许可集**。**如有必要**，在许可可用前会阻塞每一个acquire()，然后再获取该许可。每个release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，**Semaphore**只对可用许可的号码进行计数，并采取相应的行动。

## Semaphore(int permits)

创建具有给定的许可数和非公平的公平设置的Semaphore。

参数：

permits - 初始的可用许可数目。此值可能为负数，在这种情况下，必须在授予任何获取前进行释放。

**public** Semaphore(**int** permits) {

sync = **new** NonfairSync(permits);

}

NonfairSync是一个非公平的同步，继承了Semaphore.Sync内部类，Semaphore.Sync实现了AbstractQueuedSynchronizer抽象类

final static class NonfairSync extends Sync {

private static final long *serialVersionUID* = -2694183684443567898L;

NonfairSync(int permits) {

super(permits);

}

protected int tryAcquireShared(int acquires) {

return nonfairTryAcquireShared(acquires);

}

}

**abstract** **static** **class** Sync **extends** AbstractQueuedSynchronizer {

**private** **static** **final** **long** *serialVersionUID* = 1192457210091910933L;

Sync(**int** permits) {

setState(permits);

}

**final** **int** getPermits() {

**return** getState();

}

**final** **int** nonfairTryAcquireShared(**int** acquires) {

**for** (;;) {

**int** available = getState();

**int** remaining = available - acquires;

**if** (remaining < 0 || compareAndSetState(available, remaining))

**return** remaining;

}

}

**protected** **final** **boolean** tryReleaseShared(**int** releases) {

**for** (;;) {

**int** p = getState();

**if** (compareAndSetState(p, p + releases))

**return** **true**;

}

}

**final** **void** reducePermits(**int** reductions) {

**for** (;;) {

**int** current = getState();

**int** next = current - reductions;

**if** (compareAndSetState(current, next))

**return**;

}

}

**final** **int** drainPermits() {

**for** (;;) {

**int** current = getState();

**if** (current == 0 || compareAndSetState(current, 0))

**return** current;

}

}

}

## Semaphore(int permits, boolean fair)

创建具有给定的许可数和给定的公平设置的 Semaphore。

参数：

permits - 初始的可用许可数目。此值可能为负数，在这种情况下，必须在授予任何获取前进行释放。

fair - 如果此信号量保证在争用时按先进先出的顺序授予许可，则为 true；否则为 false。

**public** Semaphore(**int** permits, **boolean** fair) {

sync = (fair)? **new** FairSync(permits) : **new** NonfairSync(permits);

}

FairSync实现了Semaphore.Sync内部类进行同步控制。

final static class FairSync extends Sync {

private static final long *serialVersionUID* = 2014338818796000944L;

FairSync(int permits) {

super(permits);

}

protected int tryAcquireShared(int acquires) {

for (;;) {

if (getFirstQueuedThread() != Thread.*currentThread*() && hasQueuedThreads())

return -1;

int available = getState();

int remaining = available - acquires;

if (remaining < 0 || compareAndSetState(available, remaining))

return remaining;

}

}

}

## acquire()

从此信号量获取一个许可，在提供一个许可前一直将线程阻塞，否则线程被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())。

获取一个许可（如果提供了一个）并立即返回，将可用的许可数减 1。

如果没有可用的许可，则在发生以下两种情况之一前，禁止将当前线程用于线程安排目的并使其处于休眠状态：

a、某些其他线程调用此信号量的 [release()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Semaphore.html#release()) 方法，并且当前线程是下一个要被分配许可的线程；

b、其他某些线程[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())当前线程。

如果当前线程：

a、被此方法将其已中断状态设置为on；

b、在等待许可时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())。

则抛出InterruptedException，并且清除当前线程的已中断状态。

抛出：

InterruptedException - 如果当前线程被中断

**public** **void** acquire() **throws** InterruptedException {

sync.acquireSharedInterruptibly(1);

}

从信号量中获取一个许可，这其中还是由AbstractQueuedSynchronizer.acquireSharedInterruptibly()实现。

**public** **final** **void** acquireSharedInterruptibly(**int** arg) **throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

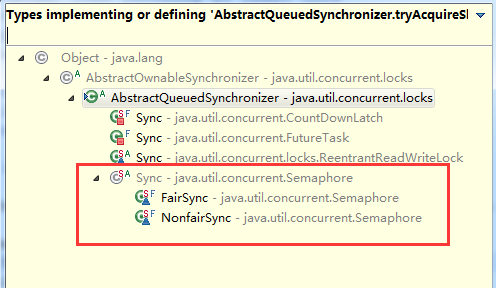
**if** (tryAcquireShared(arg) < 0)

doAcquireSharedInterruptibly(arg);

}

1、如果当前线程已经被中断，则抛出InterruptedException异常。

2、通过tryAcquireShared(int)获取许可。如果没有获取到许可，则调用doAcquireSharedInterruptibly(int)将当前获取请求加入等待队列，并阻塞当前线程，支持可中断。



从上图可以看出，Semaphore在公平机制(FairSync)和非公平机制(NonfairSync)中对tryAcquireShared(int)有着不同的实现，如下：

**公平机制FairSync.tryAcquireShared(int)实现**：

**protected** **int** tryAcquireShared(**int** acquires) {

**for** (;;) {

**if** (getFirstQueuedThread() != Thread.*currentThread*() && hasQueuedThreads())

**return** -1;

**int** available = getState();

**int** remaining = available - acquires;

**if** (remaining < 0 || compareAndSetState(available, remaining))

**return** remaining;

}

}

1、如果当前线程不是AQS CLH队列的头部节点线程，且CLH中还有其他等待队列，则返回-1，也就是在acquireSharedInterruptibly(int)方法中，会触发当前线程被加入CLH等待队列中，且会挂起当前线程。

2、通过getState()方法获取信号量中所有的可用许可。如果需要获得许可数大于可用许可，则返回负数，也就相当于会触发acquireSharedInterruptibly(int)方法；否则通过CVS操作设置可用许可，并返回。

**非公平机制NonfairSync.tryAcquireShared(int)实现**：

**protected** **int** tryAcquireShared(**int** acquires) {

**return** nonfairTryAcquireShared(acquires);

}

**final** **int** nonfairTryAcquireShared(**int** acquires) {

**for** (;;) {

**int** available = getState();

**int** remaining = available - acquires;

**if** (remaining < 0 || compareAndSetState(available, remaining))

**return** remaining;

}

}

相对于公平机制，非公平机制并没有去判断AQS CLH队列中是否存在等待节点，而是直接获取许可。

## acquire(int permits)

从此信号量获取给定数目的许可，在提供这些许可前一直将线程阻塞，或者线程已被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())。

获取给定数目的许可（如果提供了）并立即返回，将可用的许可数减去给定的量。

如果没有足够的可用许可，则在发生以下两种情况之一前，禁止将当前线程用于线程安排目的并使其处于休眠状态：

a、其他某些线程调用此信号量的某个[释放](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Semaphore.html#release())方法，当前线程是下一个被分配允许的线程并且可用许可的数目满足此请求；

b、其他某些线程[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())当前线程。

如果当前线程：

a、被此方法将其已中断状态设置为on；

b、在等待许可时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())。

则抛出InterruptedException，并且清除当前线程的已中断状态。任何原本应该分配给此线程的许可将被分配给其他试图获取许可的线程，就好像已通过调用[release()](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Semaphore.html#release())而使许可可用一样。

参数：

permits - 要获取的许可数

抛出：

InterruptedException - 如果当前线程已被中断

IllegalArgumentException - 如果 permits 为负

**public** **void** acquire(**int** permits) **throws** InterruptedException {

**if** (permits < 0) **throw** **new** IllegalArgumentException();

sync.acquireSharedInterruptibly(permits);

}

如果permits参数小于0，则抛出IllegalArgumentException异常。

后面的方法处理逻辑与[acquire()](#_acquire())一致。

## acquireUninterruptibly()

从此信号量中获取许可，在有可用的许可前将其阻塞。

获取一个许可（如果提供了一个）并立即返回，将可用的允许数减 1。

如果没有可用的许可，则在其他某些线程调用此信号量的release()方法，并且当前线程是下一个要被分配许可的线程前，禁止当前线程用于线程安排目的并使其处于休眠状态。

如果当前线程在等待许可时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())，那么它将继续等待，但是与没有发生中断，其将接收允许的时间相比，为该线程分配许可的时间可能改变。当线程确实从此方法返回后，将设置其中断状态。

**public** **void** acquireUninterruptibly() {

sync.acquireShared(1);

}

AQS对acquireShared(int)的实现。

**public** **final** **void** acquireShared(**int** arg) {

**if** (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

tryAcquireShared(int)与[acquire()](#_acquire())中的一致，包括公平和非公平两种机制的不同实现。这里我们看一下doAcquireShared(int)方法。

**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);

**if** (r >= 0) {

setHeadAndPropagate(node, r);

p.next = **null**; // help GC

**if** (interrupted) // 如果节点线程已中断。

*selfInterrupt*(); // 重新设置中断状态。

failed = **false**;

**return**;

}

}

// shouldParkAfterFailedAcquire()检查并更新等待的状态。

// parkAndCheckInterrupt() 挂起线程，清空中断状态位，并返回中断状态。

**if** (*shouldParkAfterFailedAcquire*(p, node) && parkAndCheckInterrupt())

interrupted = **true**;

}

} **finally** {

**if** (failed)

cancelAcquire(node);

}

}

## acquireUninterruptibly(int permits)

从此信号量获取给定数目的许可，在提供这些许可前一直将线程阻塞。

获取给定数目的许可（如果提供了）并立即返回，将可用的许可数减去给定的量。

如果没有足够的可用许可，则在其他某些线程调用此信号量的某个[释放](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\util\concurrent\Semaphore.html#release())方法，当前线程是下一个要被分配许可的线程，并且可用的许可数目满足此请求前，禁止当前线程用于线程安排目的并使其处于休眠状态。

如果当前的线程在等待许可时被[中断](file:///E:\MCFLY_Library\API%E5%85%A8%E9%9B%86\html\zh_CN\api\java\lang\Thread.html#interrupt())，则它会继续等待并且它在队列中的位置不受影响。当线程确实从此方法返回后，将其设置为中断状态。

参数：

permits - 要获取的许可数

抛出：

IllegalArgumentException - 如果 permits 为负

**public** **void** acquireUninterruptibly(**int** permits) {

**if** (permits < 0) **throw** **new** IllegalArgumentException();

sync.acquireShared(permits);

}  
可以看出，此方法会先判断permits参数，其后续逻辑与[acquireUninterruptibly()](#_acquireUninterruptibly())保持一致。

## availablePermits()

返回此信号量中当前可用的许可数。

此方法通常用于调试和测试目的。

返回：

此信号量中的可用许可数

**public** **int** availablePermits() {

**return** sync.getPermits();

}

**final** **int** getPermits() {

**return** getState();

}

返回state的值。

## drainPermits()

获取并返回立即可用的所有许可。

返回：

获取的许可数

**public** **int** drainPermits() {

**return** sync.drainPermits();

}

具体实现有Semaphore.Sync.drainPermits()实现。

**final** **int** drainPermits() {

**for** (;;) {

**int** current = getState();

**if** (current == 0 || compareAndSetState(current, 0))

**return** current;

}

}

立即返回可用的所有许可，此方法返回后，此信号量中没有可用许可。

## getQueuedThreads()

返回一个collection，包含**可能等待获取的线程**。因为在构造此结果的同时实际的线程set可能动态地变化，所以返回的 collection 仅是尽力的估计值。所返回collection中的元素没有特定的顺序。此方法用于加快子类的构造速度，提供更多的监视设施。

返回：

线程collection

**protected** Collection<Thread> getQueuedThreads() {

**return** sync.getQueuedThreads();

}

**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;

}

## getQueueLength()

返回正在等待获取的线程的估计数目。该值仅是估计的数字，因为在此方法遍历内部数据结构的同时，线程的数目可能动态地变化。此方法用于监视系统状态，不用于同步控制。

返回：

正在等待此锁的线程的估计数目

**public** **final** **int** getQueueLength() {

**return** sync.getQueueLength();

}

**public** **final** **int** getQueueLength() {

**int** n = 0;

**for** (Node p = tail; p != **null**; p = p.prev) {

**if** (p.thread != **null**)

++n;

}

**return** n;

}

## hasQueuedThreads()

查询是否有线程正在等待获取。注意，因为同时可能发生取消，所以返回true并不保证有其他线程等待获取许可。此方法主要用于监视系统状态。

返回：

如果可能有其他线程正在等待获取锁，则返回true

**public** **final** **boolean** hasQueuedThreads() {

**return** sync.hasQueuedThreads();

}

**public** **final** **boolean** hasQueuedThreads() {

**return** head != tail;

}

## isFair()

如果此信号量的公平设置为 true，则返回 true。

返回：

如果此信号量的公平设置为 true，则返回 true

public boolean isFair() {

return sync instanceof FairSync;

}

## reducePermits(int reduction)

根据指定的缩减量减小可用许可的数目。此方法在使用信号量来跟踪那些变为不可用资源的子类中很有用。此方法不同于acquire，在许可变为可用的过程中，它不会阻塞等待。

参数：

reduction - 要移除的许可数

抛出：

IllegalArgumentException - 如果 reduction 是负数

**protected** **void** reducePermits(**int** reduction) {

**if** (reduction < 0) **throw** **new** IllegalArgumentException();

sync.reducePermits(reduction);

}

**final** **void** reducePermits(**int** reductions) {

**for** (;;) {

**int** current = getState();

**int** next = current - reductions;

**if** (compareAndSetState(current, next))

**return**;

}

}