

第9章 Java中的线程池

Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。在开发过程中，合理地使用线程池能够带来3个好处。

第一：**降低资源消耗**。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：**提高响应速度**。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：**提高线程的可管理性**。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。但是，要做到合理利用线程池，必须对其实现原理了如指掌。

9.1 线程池的实现原理

当向线程池提交一个任务之后，线程池是如何处理这个任务的呢？本节来看一下线程池的主要处理流程，处理流程图如图9-1所示。

从图中可以看出，当提交一个新任务到线程池时，线程池的处理流程如下。

1) 线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行任务。如果核心线程池里的线程都在执行任务，则进入下个流程。

2) 线程池判断工作队列是否已经满。如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3) 线程池判断线程池的线程是否都处于工作状态。如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

ThreadPoolExecutor执行execute()方法的示意图，如图9-2所示。

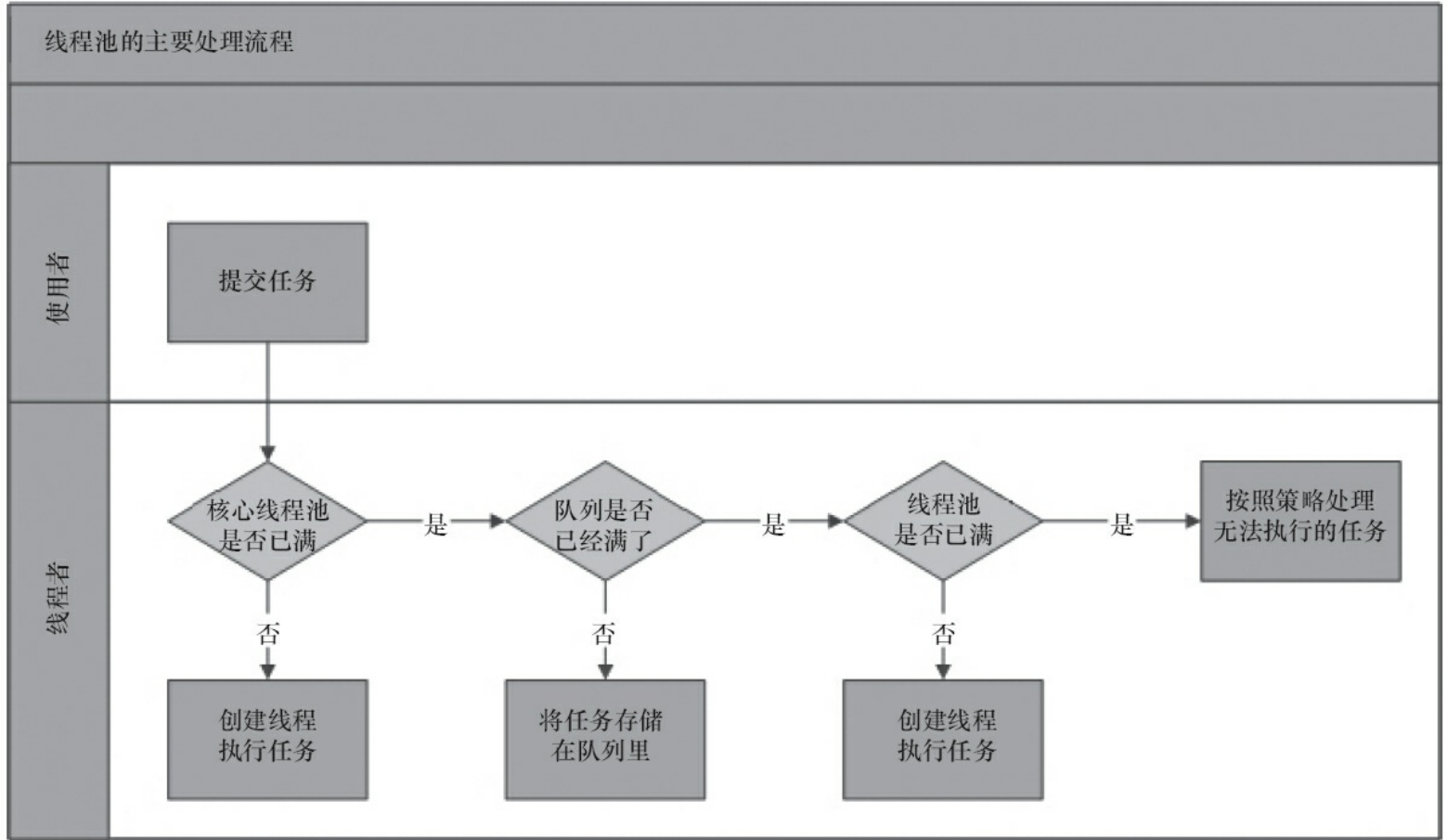


图9-1 线程池的主要处理流程

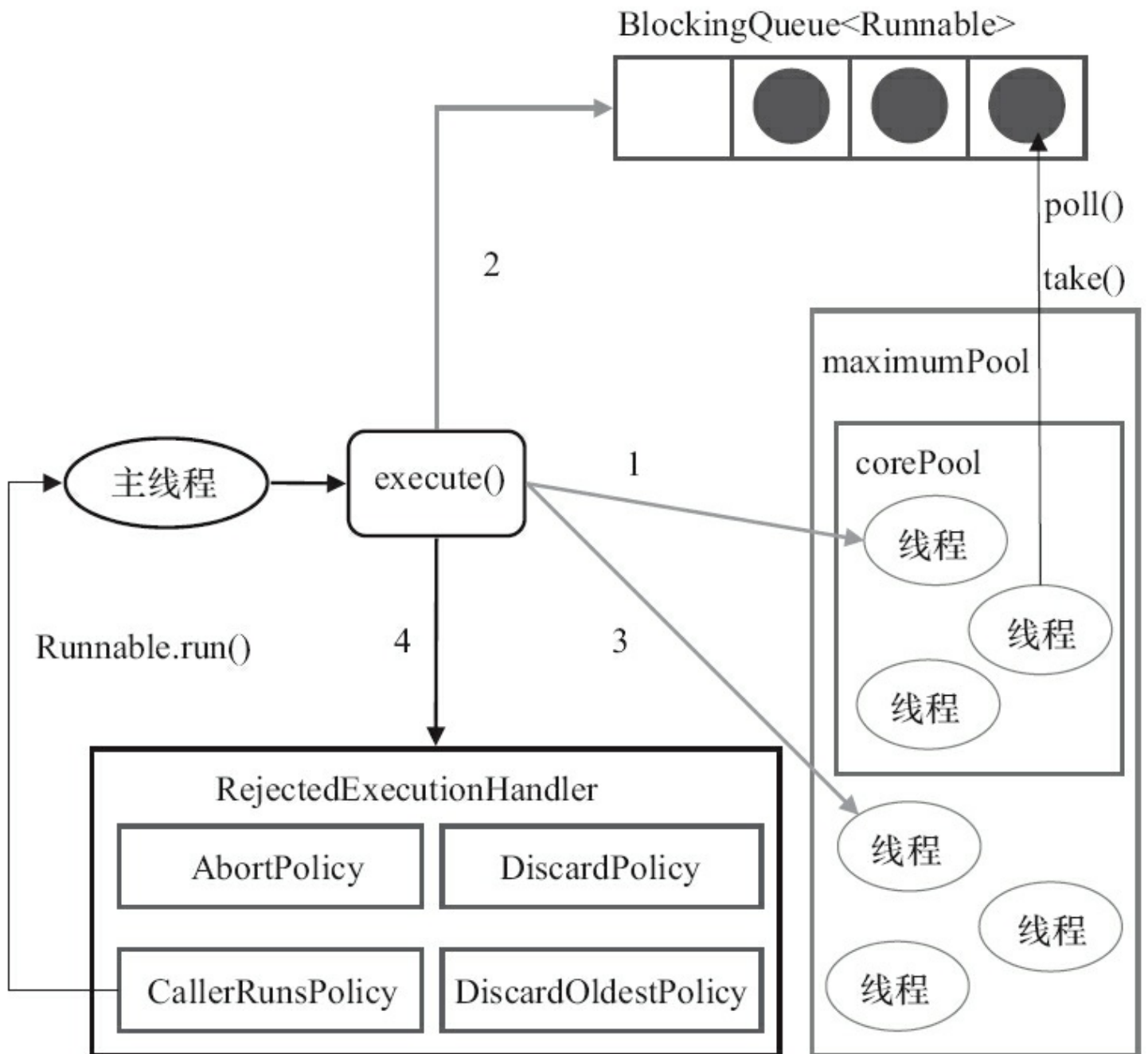


图9-2 ThreadPoolExecutor执行示意图

ThreadPoolExecutor执行execute方法分下面4种情况。

- 1) 如果当前运行的线程少于corePoolSize, 则创建新线程来执行任务(注意, 执行这一步骤需要获取全局锁)。
- 2) 如果运行的线程等于或多于corePoolSize, 则将任务加入BlockingQueue。
- 3) 如果无法将任务加入BlockingQueue(队列已满), 则创建新的线程来处理任务(注意, 执

行这一步骤需要获取全局锁)。

4) 如果创建新线程将使当前运行的线程超出maximumPoolSize, 任务将被拒绝, 并调用RejectedExecutionHandler.rejectedExecution()方法。

ThreadPoolExecutor采取上述步骤的总体设计思路, 是为了在执行execute()方法时, 尽可能地避免获取全局锁(那将会是一个严重的可伸缩瓶颈)。在ThreadPoolExecutor完成预热之后(当前运行的线程数大于等于corePoolSize), 几乎所有的execute()方法调用都是执行步骤2, 而步骤2不需要获取全局锁。

源码分析: 上面的流程分析让我们很直观地了解了线程池的工作原理, 让我们再通过源代码来看看是如何实现的, 线程池执行任务的方法如下。

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    // 如果线程数小于基本线程数, 则创建线程并执行当前任务
    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
        // 如线程数大于等于基本线程数或线程创建失败, 则将当前任务放到工作队列中。
        if (runState == RUNNING && workQueue.offer(command)) {
            if (runState != RUNNING || poolSize == 0)
                ensureQueuedTaskHandled(command);
        }
        // 如果线程池不处于运行中或任务无法放入队列, 并且当前线程数量小于最大允许的线程数量,
        // 则创建一个线程执行任务。
    } else if (!addIfUnderMaximumPoolSize(command)) {
        // 抛出RejectedExecutionException异常
        reject(command); // is shutdown or saturated
    }
}
```

工作线程: 线程池创建线程时, 会将线程封装成工作线程Worker, Worker在执行完任务后, 还会循环获取工作队列里的任务来执行。我们可以从Worker类的run()方法里看到这点。

```
public void run() {
    try {
        Runnable task = firstTask;
        firstTask = null;
        while (task != null || (task = getTask()) != null) {
            runTask(task);
        }
    }
}
```

```

        task = null;
    }
    } finally {
        workerDone(this);
    }
}

```

ThreadPoolExecutor中线程执行任务的示意图如图9-3所示。

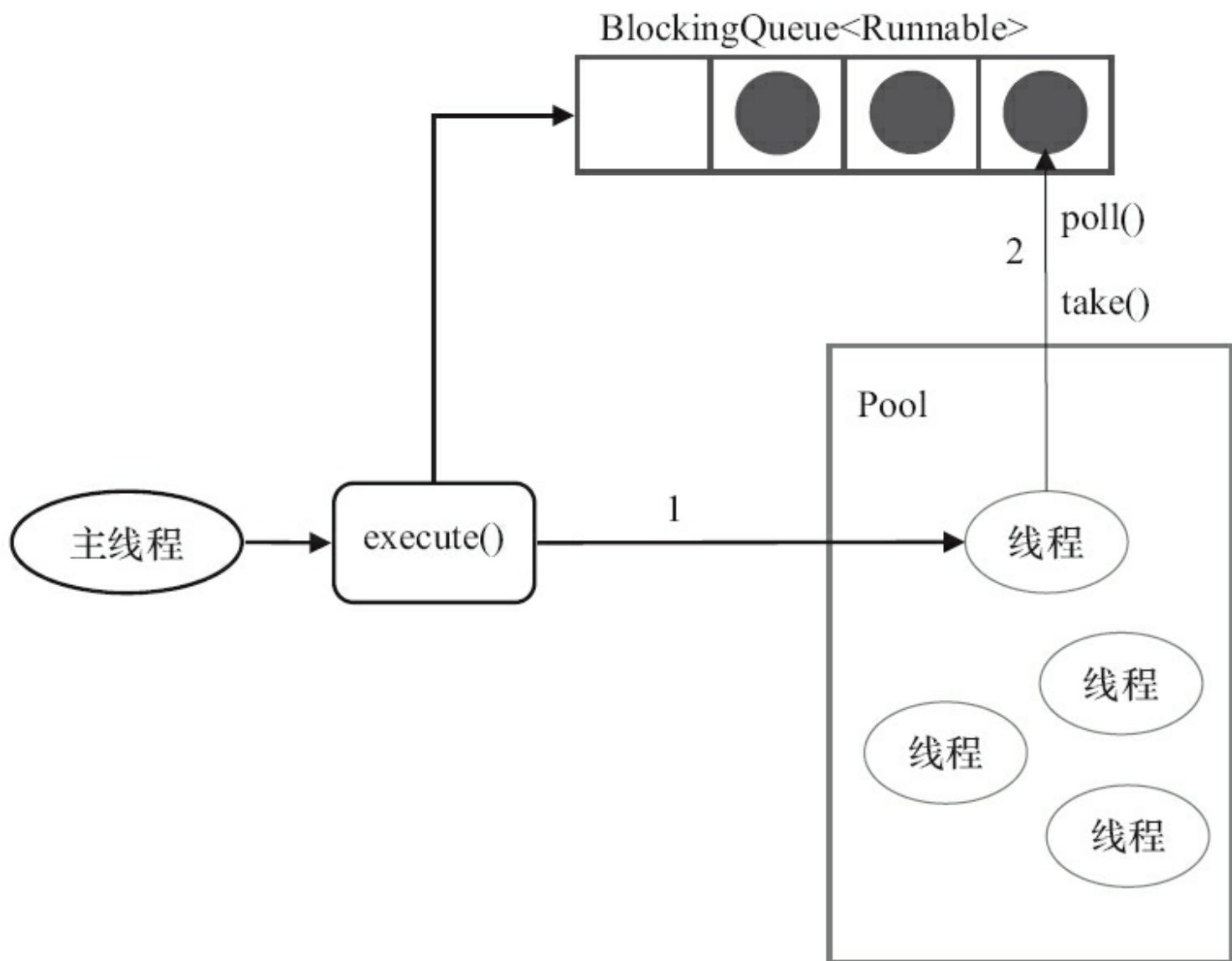


图9-3 ThreadPoolExecutor执行任务示意图

线程池中的线程执行任务分两种情况，如下。

1) 在`execute()`方法中创建一个线程时，会让这个线程执行当前任务。

2) 这个线程执行完上图中1的任务后, 会反复从BlockingQueue获取任务来执行。

9.2 线程池的使用

9.2.1 线程池的创建

我们可以通过ThreadPoolExecutor来创建一个线程池。

```
new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime,
    milliseconds, runnableTaskQueue, handler);
```

创建一个线程池时需要输入几个参数, 如下。

1) corePoolSize(线程池的基本大小): 当提交一个任务到线程池时, 线程池会创建一个线程来执行任务, 即使其他空闲的基本线程能够执行新任务也会创建线程, 等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads()方法, 线程池会提前创建并启动所有基本线程。

2) runnableTaskQueue(任务队列): 用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。

- ArrayBlockingQueue: 是一个基于数组结构的有界阻塞队列, 此队列按FIFO(先进先出)原则对元素进行排序。

- LinkedBlockingQueue: 一个基于链表结构的阻塞队列, 此队列按FIFO排序元素, 吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。

- SynchronousQueue: 一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作, 否则插入操作一直处于阻塞状态, 吞吐量通常要高于Linked-BlockingQueue, 静态工厂方法Executors.newCachedThreadPool使用了这个队列。

- PriorityBlockingQueue: 一个具有优先级的无限阻塞队列。

3) `maximumPoolSize`(线程池最大数量): 线程池允许创建的最大线程数。如果队列满了, 并且已创建的线程数小于最大线程数, 则线程池会再创建新的线程执行任务。值得注意的是, 如果使用了无界的任务队列这个参数就没什么效果。

4) `ThreadFactory`: 用于设置创建线程的工厂, 可以通过线程工厂给每个创建出来的线程设置更有意义的名字。使用开源框架 `guava` 提供的 `ThreadFactoryBuilder` 可以快速给线程池里的线程设置有意义的名字, 代码如下。

```
new ThreadFactoryBuilder().setNameFormat("XX-task-%d").build();
```

5) `RejectedExecutionHandler`(饱和策略): 当队列和线程池都满了, 说明线程池处于饱和状态, 那么必须采取一种策略处理提交的新任务。这个策略默认情况下是 `AbortPolicy`, 表示无法处理新任务时抛出异常。在 `JDK 1.5` 中 `Java` 线程池框架提供了以下4种策略。

- `AbortPolicy`: 直接抛出异常。

- `CallerRunsPolicy`: 只用调用者所在线程来运行任务。

- `DiscardOldestPolicy`: 丢弃队列里最近的一个任务, 并执行当前任务。

- `DiscardPolicy`: 不处理, 丢弃掉。

当然, 也可以根据应用场景需要来实现 `RejectedExecutionHandler` 接口自定义策略。如记录日志或持久化存储不能处理的任务。

· `keepAliveTime`(线程活动保持时间): 线程池的工作线程空闲后, 保持存活的时间。所以, 如果任务很多, 并且每个任务执行的时间比较短, 可以调大时间, 提高线程的利用率。

· `TimeUnit`(线程活动保持时间的单位): 可选的单位有天(`DAYS`)、小时(`HOURS`)、分钟(`MINUTES`)、毫秒(`MILLISECONDS`)、微秒(`MICROSECONDS`, 千分之一毫秒)和纳秒

(NANOSECONDS, 千分之一微秒)。

9.2.2 向线程池提交任务

可以使用两个方法向线程池提交任务，分别为`execute()`和`submit()`方法。

`execute()`方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功。

通过以下代码可知`execute()`方法输入的任务是一个`Runnable`类的实例。

```
threadsPool.execute(new Runnable() {
    @Override
    public void run() {
        // TODO Auto-generated method stub
    }
});
```

`submit()`方法用于提交需要返回值的任务。线程池会返回一个`future`类型的对象，通过这个`future`对象可以判断任务是否执行成功，并且可以通过`future`的`get()`方法来获取返回值，`get()`方法会阻塞当前线程直到任务完成，而使用`get(long timeout, TimeUnit unit)`方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

```
Future<Object> future = executor.submit(harReturnValuetask);
try {
    Object s = future.get();
} catch (InterruptedException e) {
    // 处理中断异常
} catch (ExecutionException e) {
    // 处理无法执行任务异常
} finally {
    // 关闭线程池
    executor.shutdown();
}
```

9.2.3 关闭线程池

可以通过调用线程池的shutdown或shutdownNow方法来关闭线程池。它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的区别，shutdownNow首先将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而shutdown只是将线程池的状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法中的任意一个，isShutdown方法就会返回true。当所有的任务都已关闭后，才表示线程池关闭成功，这时调用isTerminated方法会返回true。至于应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用shutdown方法来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow方法。


9.2.4 合理地配置线程池

要想合理地配置线程池，就必须首先分析任务特性，可以从以下几个角度来分析。

- 任务的性质: CPU密集型任务、IO密集型任务和混合型任务。
- 任务的优先级: 高、中和低。
- 任务的执行时间: 长、中和短。
- 任务的依赖性: 是否依赖其他系统资源，如数据库连接。

性质不同的任务可以用不同规模的线程池分开处理。CPU密集型任务应配置尽可能小的线程，如配置 $N_{\text{cpu}}+1$ 个线程的线程池。由于IO密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如 $2*N_{\text{cpu}}$ 。混合型的任务，如果可以拆分，将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。可以通过`Runtime.getRuntime().availableProcessors()`方法获得当前设备的CPU个数。

优先级不同的任务可以使用优先级队列`PriorityBlockingQueue`来处理。它可以让优先级高的任务先执行。

 **注意** 如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者可以使用优先级队列，让执行时间短的任务先执行。

依赖数据库连接池的任务，因为线程提交SQL后需要等待数据库返回结果，等待的时间越长，则CPU空闲时间就越长，那么线程数应该设置得越大，这样才能更好地利用CPU。

建议使用有界队列。有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点儿，比如几千。有一次，我们系统里后台任务线程池的队列和线程池全满了，不断抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行SQL变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞，任务积压在线程池里。如果当时我们设置成无界队列，那么线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然，我们的系统所有的任务是用单独的服务器部署的，我们使用不同规模的线程池完成不同类型的任务，但是出现这样问题时也会影响到其他任务。

9.2.5 线程池的监控

如果在系统中大量使用线程池, 则有必要对线程池进行监控, 方便在出现问题时, 可以根据线程池的使用状况快速定位问题。可以通过线程池提供的参数进行监控, 在监控线程池的时候可以使用以下属性。

- taskCount: 线程池需要执行的任务数量。

- completedTaskCount: 线程池在运行过程中已完成的任务数量, 小于或等于taskCount。

- largestPoolSize: 线程池里曾经创建过的最大线程数量。通过这个数据可以知道线程池是否曾经满过。如该数值等于线程池的最大大小, 则表示线程池曾经满过。

- getPoolSize: 线程池的线程数量。如果线程池不销毁的话, 线程池里的线程不会自动销毁, 所以这个大小只增不减。

- getActiveCount: 获取活动的线程数。

通过扩展线程池进行监控。可以通过继承线程池来自定义线程池, 重写线程池的beforeExecute、afterExecute和terminated方法, 也可以在任务执行前、执行后和线程池关闭前执行一些代码来进行监控。例如, 监控任务的平均执行时间、最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。

```
protected void beforeExecute(Thread t, Runnable r) { }
```

9.3 本章小结

在工作中我经常发现, 很多人因为不了解线程池的实现原理, 把线程池配置错误, 从而导致了各种问题。本章介绍了为什么要使用线程池、如何使用线程池和线程池的使用原理, 相信阅读完本章之后, 读者能更准确、更有效地使用线程池。

第10章 Executor框架

在Java中，使用线程来异步执行任务。Java线程的创建与销毁需要一定的开销，如果我们为每一个任务创建一个新线程来执行，这些线程的创建与销毁将消耗大量的计算资源。同时，为每一个任务创建一个新线程来执行，这种策略可能会使处于高负荷状态的应用最终崩溃。

Java的线程既是工作单元，也是执行机制。从JDK 5开始，把工作单元与执行机制分离开来。工作单元包括Runnable和Callable，而执行机制由Executor框架提供。

10.1 Executor框架简介

10.1.1 Executor框架的两级调度模型

在HotSpot VM的线程模型中, Java线程(`java.lang.Thread`)被一对一映射为本地操作系统线程。Java线程启动时会创建一个本地操作系统线程;当该Java线程终止时, 这个操作系统线程也会被回收。操作系统会调度所有线程并将它们分配给可用的CPU。

在上层, Java多线程程序通常把应用分解为若干个任务, 然后使用用户级的调度器(Executor框架)将这些任务映射为固定数量的线程;在底层, 操作系统内核将这些线程映射到硬件处理器上。这种两级调度模型的示意图如图10-1所示。

从图中可以看出, 应用程序通过Executor框架控制上层的调度;而下层的调度由操作系统内核控制, 下层的调度不受应用程序的控制。

10.1.2 Executor框架的结构与成员

本文将分两部分来介绍Executor: Executor的结构和Executor框架包含的成员组件。

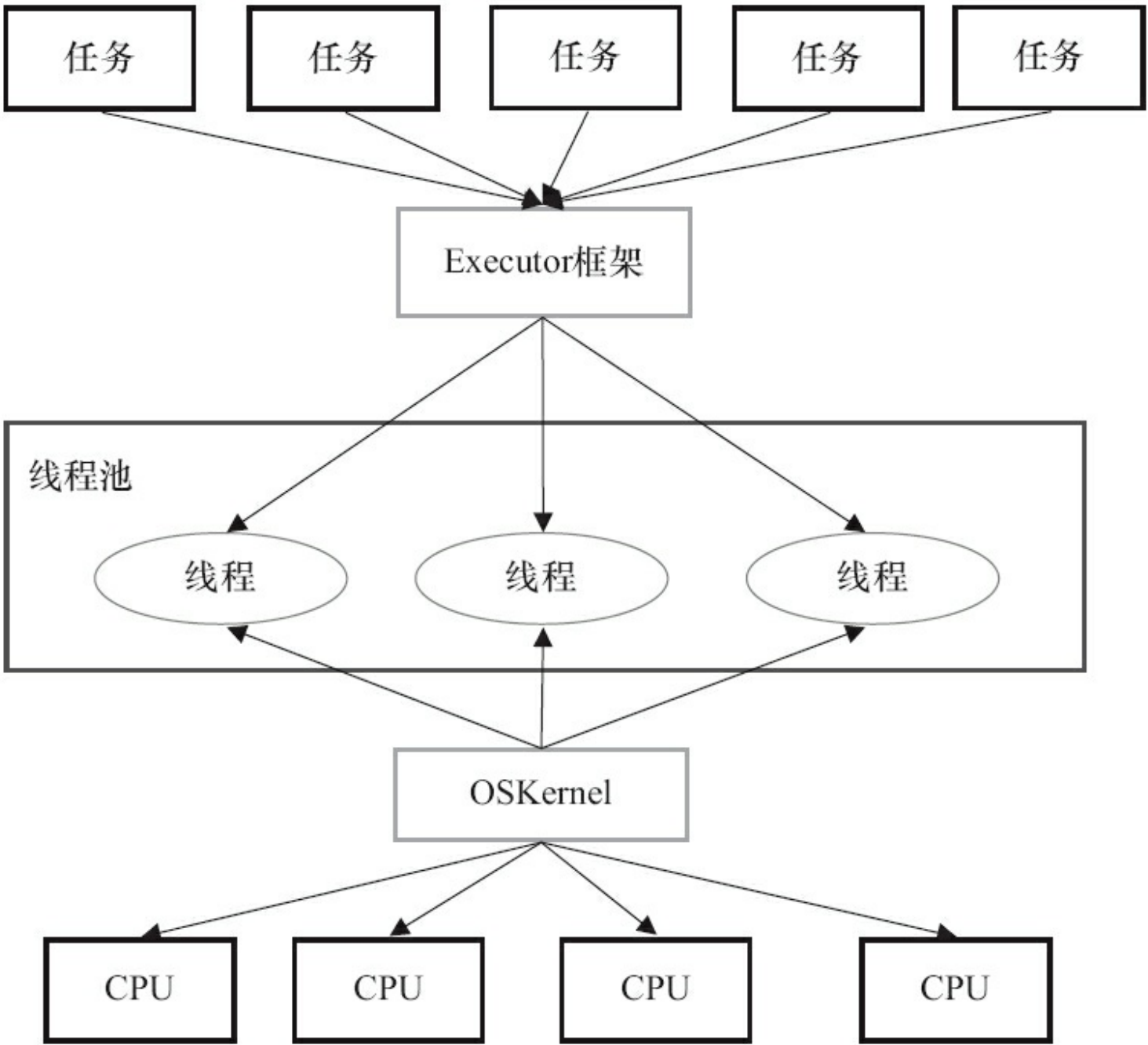


图10-1 任务的两级调度模型

1.Executor框架的结构

Executor框架主要由3大部分组成如下。

- 任务。包括被执行任务需要实现的接口:Runnable接口或Callable接口。

- 任务的执行。包括任务执行机制的核心接口Executor, 以及继承自Executor的ExecutorService接口。Executor框架有两个关键类实现了ExecutorService接口(ThreadPoolExecutor和ScheduledThreadPoolExecutor)。

- 异步计算的结果。包括接口Future和实现Future接口的FutureTask类。

Executor框架包含的主要的类与接口如图10-2所示。

下面是这些类和接口的简介。

- Executor是一个接口, 它是Executor框架的基础, 它将任务的提交与任务的执行分离开来。

- ThreadPoolExecutor是线程池的核心实现类, 用来执行被提交的任务。

- ScheduledThreadPoolExecutor是一个实现类, 可以在给定的延迟后运行命令, 或者定期执行命令。ScheduledThreadPoolExecutor比Timer更灵活, 功能更强大。

- Future接口和实现Future接口的FutureTask类, 代表异步计算的结果。

- Runnable接口和Callable接口的实现类, 都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。

Executor框架的使用示意图如图10-3所示。

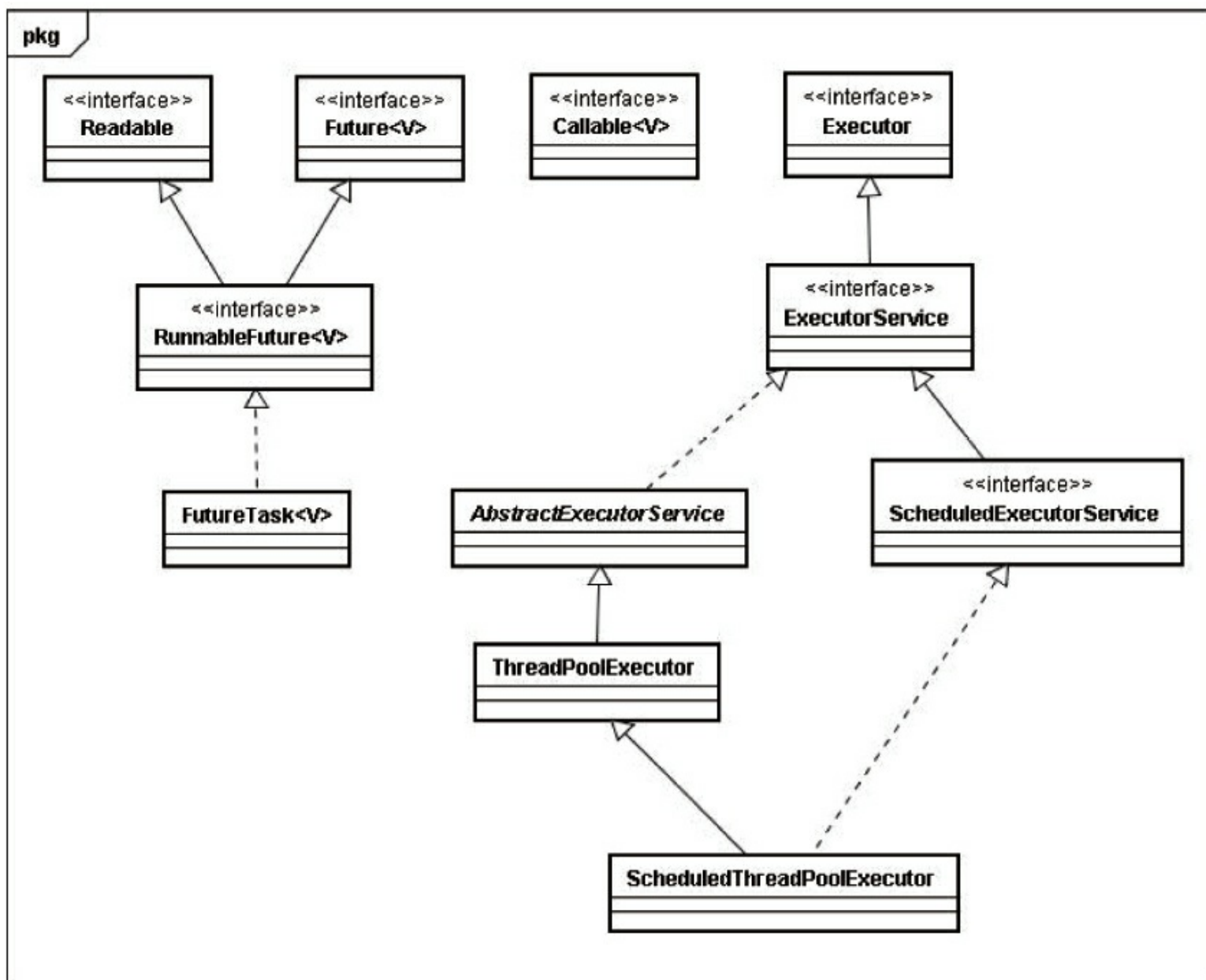


图10-2 Executor框架的类与接口

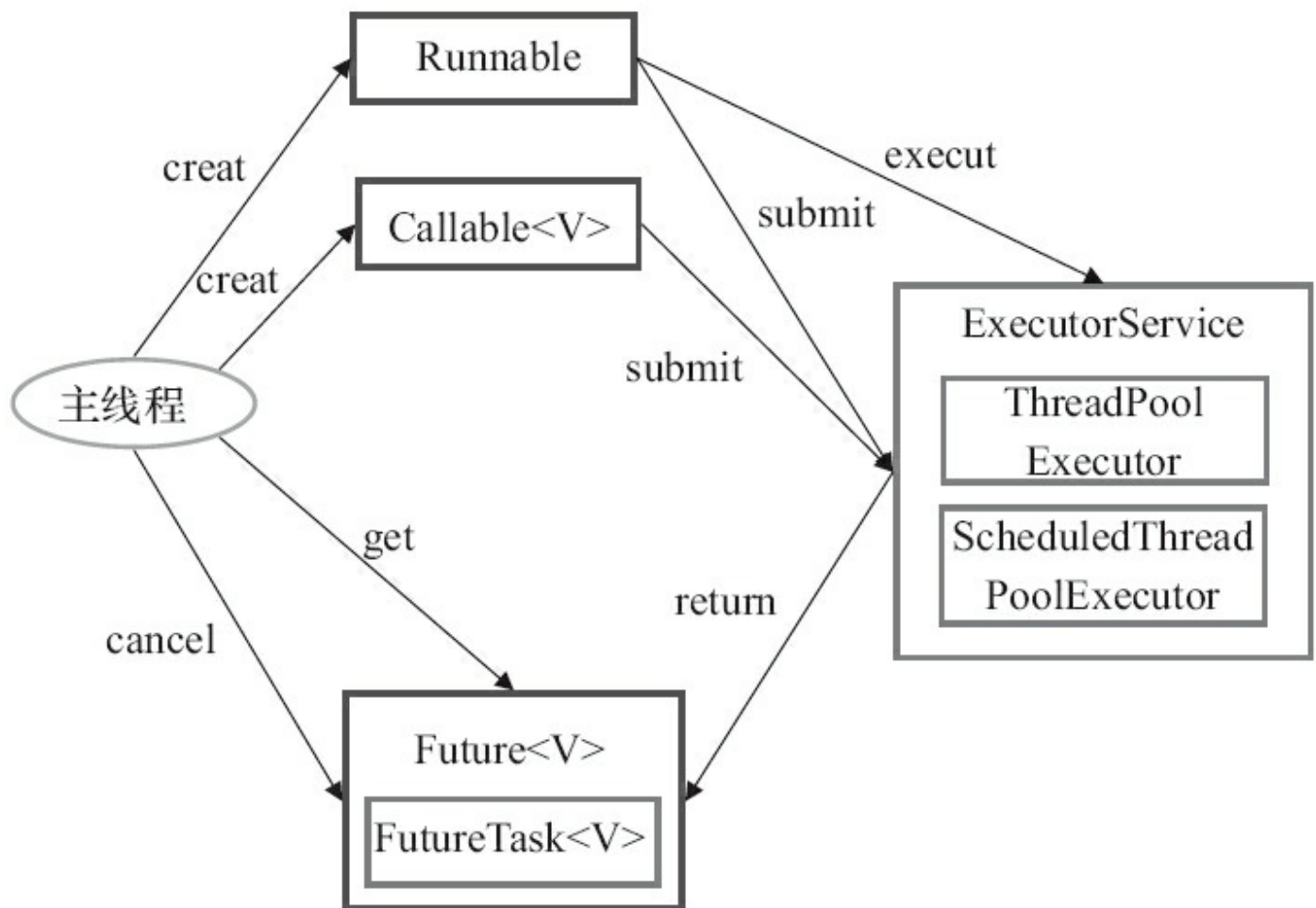


图10-3 Executor框架的使用示意图

主线程首先要创建实现Runnable或者Callable接口的任务对象。工具类Executors可以把一个Runnable对象封装为一个Callable对象(Executors.callable(Runnable task)或Executors.callable(Runnable task, Object result))。

然后可以把Runnable对象直接交给ExecutorService执行(ExecutorService.execute(Runnable command));或者也可以把Runnable对象或Callable对象提交给ExecutorService执行(ExecutorService.submit(Runnable task)或ExecutorService.submit(Callable<T>task))。

如果执行ExecutorService.submit(...), ExecutorService将返回一个实现Future接口的对象(到目前为止的JDK中, 返回的是FutureTask对象)。由于FutureTask实现了Runnable, 程序员也可以创建FutureTask, 然后直接交给ExecutorService执行。

最后, 主线程可以执行FutureTask.get()方法来等待任务执行完成。主线程也可以执行

FutureTask.cancel (boolean mayInterruptIfRunning) 来取消此任务的执行。

2.Executor框架的成员

本节将介绍Executor框架的主要成员: ThreadPoolExecutor、ScheduledThreadPoolExecutor、Future接口、Runnable接口、Callable接口和Executors。

(1) ThreadPoolExecutor

ThreadPoolExecutor通常使用工厂类Executors来创建。Executors可以创建3种类型的ThreadPoolExecutor: SingleThreadExecutor、FixedThreadPool和CachedThreadPool。

下面分别介绍这3种ThreadPoolExecutor。

1) FixedThreadPool。下面是Executors提供的, 创建使用固定线程数的FixedThreadPool的API。

```
public static ExecutorService newFixedThreadPool(int nThreads)
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFact
```

FixedThreadPool适用于为了满足资源管理的需求, 而需要限制当前线程数量的应用场景, 它适用于负载比较重的服务器。

2) SingleThreadExecutor。下面是Executors提供的, 创建使用单个线程的SingleThread-Executor的API。

```
public static ExecutorService newSingleThreadExecutor()
public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)
```

SingleThreadExecutor适用于需要保证顺序地执行各个任务; 并且在任意时间点, 不会有多个线程是活动的应用场景。

3) `CachedThreadPool`。下面是Executors提供的, 创建一个会根据需要创建新线程的`CachedThreadPool`的API。

```
public static ExecutorService newCachedThreadPool()  
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)
```

`CachedThreadPool`是大小无界的线程池, 适用于执行很多的短期异步任务的小程序, 或者是负载较轻的服务器。

(2) `ScheduledThreadPoolExecutor`

`ScheduledThreadPoolExecutor`通常使用工厂类Executors来创建。Executors可以创建2种类型的`ScheduledThreadPoolExecutor`, 如下。

- `ScheduledThreadPoolExecutor`。包含若干个线程的`ScheduledThreadPoolExecutor`。
- `SingleThreadScheduledExecutor`。只包含一个线程的`ScheduledThreadPoolExecutor`。

下面分别介绍这两种`ScheduledThreadPoolExecutor`。

下面是工厂类Executors提供的, 创建固定个数线程的`ScheduledThreadPoolExecutor`的API。

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)  
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadF
```

`ScheduledThreadPoolExecutor`适用于需要多个后台线程执行周期任务, 同时为了满足资源管理的需求而需要限制后台线程的数量的应用场景。下面是Executors提供的, 创建单个线程的`SingleThreadScheduledExecutor`的API。

```
public static ScheduledExecutorService newSingleThreadScheduledExecutor()  
public static ScheduledExecutorService newSingleThreadScheduledExecutor  
(ThreadFactory threadFactory)
```

SingleThreadScheduledExecutor适用于需要单个后台线程执行周期任务，同时需要保证顺序地执行各个任务的应用场景。

(3)Future接口

Future接口和实现Future接口的FutureTask类用来表示异步计算的结果。当我们把Runnable接口或Callable接口的实现类提交(submit)给ThreadPoolExecutor或ScheduledThreadPoolExecutor时，ThreadPoolExecutor或ScheduledThreadPoolExecutor会向我们返回一个FutureTask对象。下面是对应的API。

```
<T> Future<T> submit(Callable<T> task)
<T> Future<T> submit(Runnable task, T result)
Future<> submit(Runnable task)
```

有一点需要读者注意，到目前最新的JDK 8为止，Java通过上述API返回的是一个FutureTask对象。但从API可以看到，Java仅仅保证返回的是一个实现了Future接口的对象。在未来的JDK实现中，返回的可能不一定是FutureTask。

(4)Runnable接口和Callable接口

Runnable接口和Callable接口的实现类，都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。它们之间的区别是Runnable不会返回结果，而Callable可以返回结果。

除了可以自己创建实现Callable接口的对象外，还可以使用工厂类Executors来把一个Runnable包装成一个Callable。

下面是Executors提供的，把一个Runnable包装成一个Callable的API。

```
public static Callable<Object> callable(Runnable task) // 假设返回对象Callable1
```

下面是Executors提供的, 把一个Runnable和一个待返回的结果包装成一个Callable的API。

```
public static <T> Callable<T> callable(Runnable task, T result)           // 假设返回对象Call
```

前面讲过, 当我们把一个Callable对象(比如上面的Callable1或Callable2)提交给ThreadPoolExecutor或ScheduledThreadPoolExecutor执行时, submit(...)会向我们返回一个FutureTask对象。我们可以执行FutureTask.get()方法来等待任务执行完成。当任务成功完成后FutureTask.get()将返回该任务的结果。例如, 如果提交的是对象Callable1, FutureTask.get()方法将返回null; 如果提交的是对象Callable2, FutureTask.get()方法将返回result对象。

10.2 ThreadPoolExecutor详解

Executor框架最核心的类是ThreadPoolExecutor, 它是线程池的实现类, 主要由下列4个组件构成。

- corePool: 核心线程池的大小。
- maximumPool: 最大线程池的大小。
- BlockingQueue: 用来暂时保存任务的工作队列。
- RejectedExecutionHandler: 当ThreadPoolExecutor已经关闭或ThreadPoolExecutor已经饱和时(达到了最大线程池大小且工作队列已满), execute()方法将要调用的Handler。
- 通过Executor框架的工具类Executors, 可以创建3种类型的ThreadPoolExecutor。
 - FixedThreadPool。
 - SingleThreadExecutor。
 - CachedThreadPool。

下面将分别介绍这3种ThreadPoolExecutor。

10.2.1 FixedThreadPool详解

FixedThreadPool被称为可重用固定线程数的线程池。下面是FixedThreadPool的源代码实现。

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

FixedThreadPool的corePoolSize和maximumPoolSize都被设置为创建FixedThreadPool时指定的参数nThreads。

当线程池中的线程数大于corePoolSize时, keepAliveTime为多余的空闲线程等待新任务的最长时间, 超过这个时间后多余的线程将被终止。这里把keepAliveTime设置为0L, 意味着多余的空闲线程会被立即终止。

FixedThreadPool的execute()方法的运行示意图如图10-4所示。

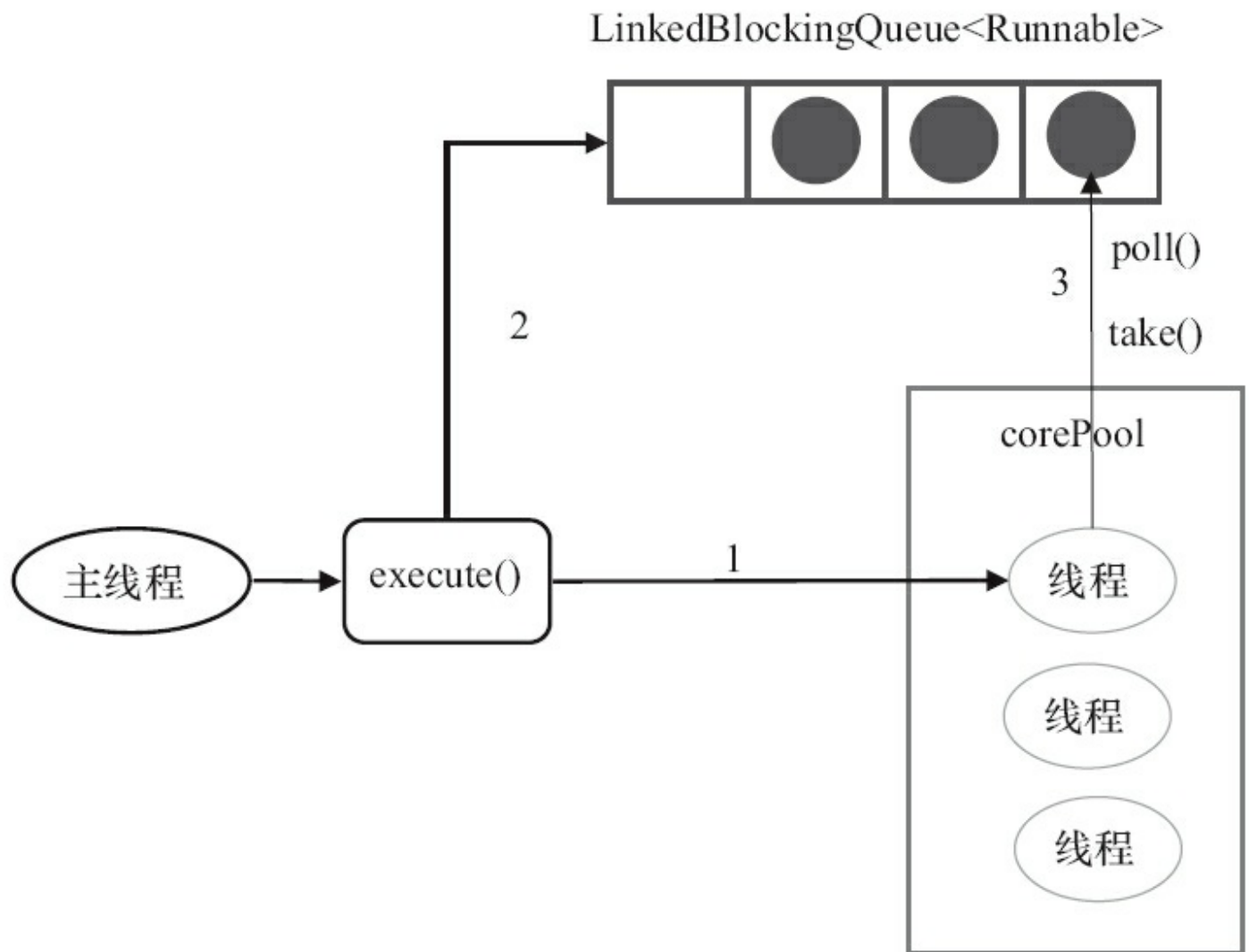


图10-4 FixedThreadPool的execute()的运行示意图

对图10-4的说明如下。

1) 如果当前运行的线程数少于`corePoolSize`, 则创建新线程来执行任务。

2) 在线程池完成预热之后(当前运行的线程数等于`corePoolSize`), 将任务加入`LinkedListBlockingQueue`。

3) 线程执行完1中的任务后, 会在循环中反复从`LinkedListBlockingQueue`获取任务来执行。

`FixedThreadPool`使用无界队列`LinkedListBlockingQueue`作为线程池的工作队列(队列的容量为`Integer.MAX_VALUE`)。使用无界队列作为工作队列会对线程池带来如下影响。

1) 当线程池中的线程数达到`corePoolSize`后, 新任务将在无界队列中等待, 因此线程池中的线程数不会超过`corePoolSize`。

2) 由于1, 使用无界队列时`maximumPoolSize`将是一个无效参数。

3) 由于1和2, 使用无界队列时`keepAliveTime`将是一个无效参数。

4) 由于使用无界队列, 运行中的`FixedThreadPool`(未执行方法`shutdown()`或`shutdownNow()`)不会拒绝任务(不会调用`RejectedExecutionHandler.rejectedExecution`方法)。

10.2.2 SingleThreadExecutor详解

SingleThreadExecutor是使用单个worker线程的Executor。下面是SingleThreadExecutor的源代码实现。

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

SingleThreadExecutor的corePoolSize和maximumPoolSize被设置为1。其他参数与FixedThreadPool相同。SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列(队列的容量为Integer.MAX_VALUE)。SingleThreadExecutor使用无界队列作为工作队列对线程池带来的影响与FixedThreadPool相同, 这里就不赘述了。

SingleThreadExecutor的运行示意图如图10-5所示。

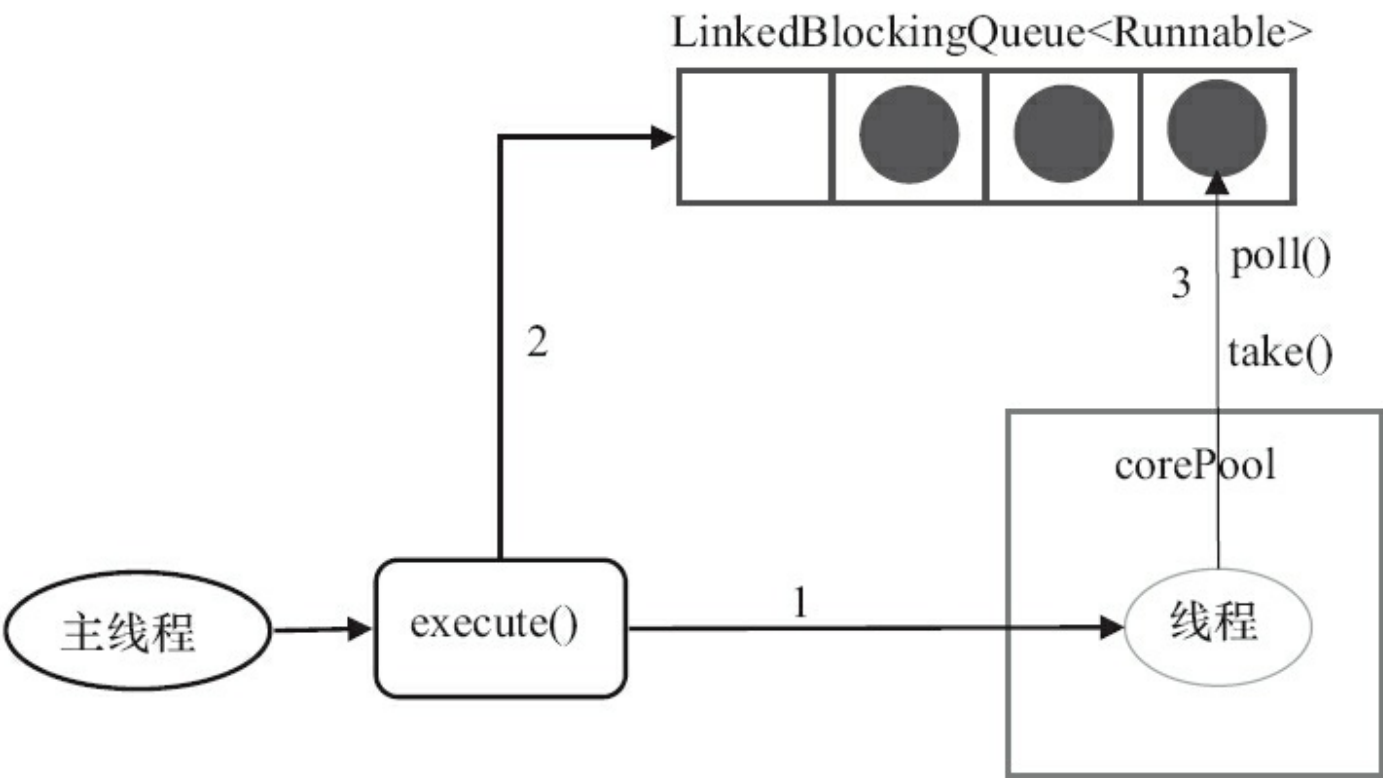


图10-5 SingleThreadExecutor的execute()的运行示意图

对图10-5的说明如下。

- 1) 如果当前运行的线程数少于corePoolSize(即线程池中无运行的线程), 则创建一个新线程来执行任务。
- 2) 在线程池完成预热之后(当前线程池中有一个运行的线程), 将任务加入Linked-BlockingQueue。
- 3) 线程执行完1中的任务后, 会在一个无限循环中反复从LinkedBlockingQueue获取任务来执行。

10.2.3 CachedThreadPool详解

CachedThreadPool是一个会根据需要创建新线程的线程池。下面是创建CachedThreadPool的源代码。

```
public static ExecutorService newCachedThreadPool() {  
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,  
                                   60L, TimeUnit.SECONDS,  
                                   new SynchronousQueue<Runnable>());  
}
```

CachedThreadPool的corePoolSize被设置为0, 即corePool为空; maximumPoolSize被设置为Integer.MAX_VALUE, 即maximumPool是无界的。这里把keepAliveTime设置为60L, 意味着CachedThreadPool中的空闲线程等待新任务的最长时间为60秒, 空闲线程超过60秒后将会被终止。

FixedThreadPool和SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列。CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列, 但CachedThreadPool的maximumPool是无界的。这意味着, 如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时, CachedThreadPool会不断创建新线程。极端情况下, CachedThreadPool会因为创建过多线程而耗尽CPU和内存资源。

CachedThreadPool的execute()方法的执行示意图如图10-6所示。

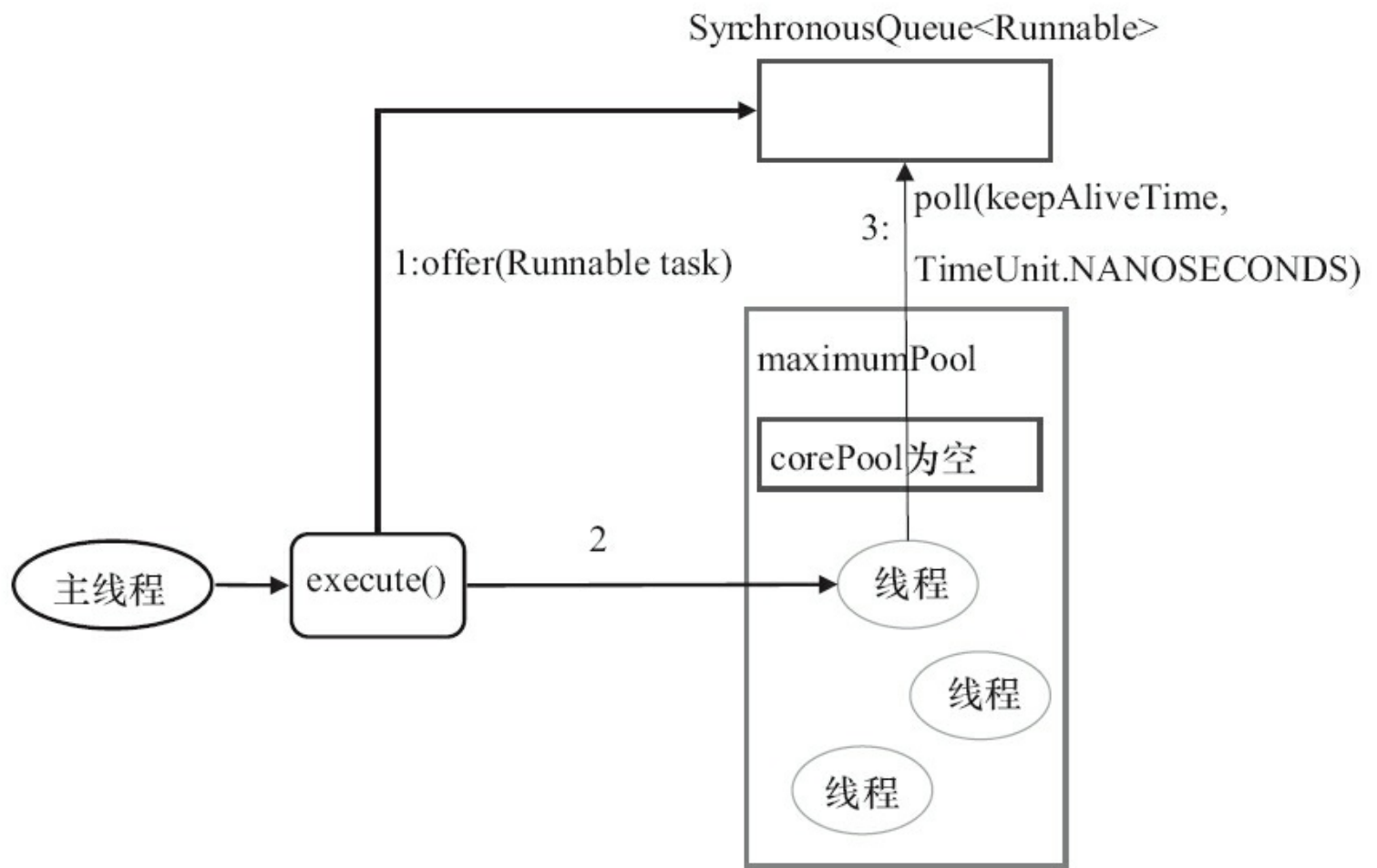


图10-6 CachedThreadPool的execute()的运行示意图

对图10-6的说明如下。

1) 首先执行SynchronousQueue.offer(Runnable task)。如果当前maximumPool中有空闲线程正在执行SynchronousQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)，那么主线程执行offer操作与空闲线程执行的poll操作配对成功，主线程把任务交给空闲线程执行，execute()方法执行完成；否则执行下面的步骤2)。

2) 当初始maximumPool为空，或者maximumPool中当前没有空闲线程时，将没有线程执行SynchronousQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)。这种情况下，步骤1)将失败。此时CachedThreadPool会创建一个新线程执行任务，execute()方法执行完成。

3) 在步骤2)中新创建的线程将任务执行完后，会执行SynchronousQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)。这个poll操作会让空闲线

程最多在SynchronousQueue中等待60秒钟。如果60秒钟内主线程提交了一个新任务(主线程执行步骤1)), 那么这个空闲线程将执行主线程提交的新任务;否则, 这个空闲线程将终止。由于空闲60秒的空闲线程会被终止, 因此长时间保持空闲的CachedThreadPool不会使用任何资源。

前面提到过, SynchronousQueue是一个没有容量的阻塞队列。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。CachedThreadPool使用SynchronousQueue, 把主线程提交的任务传递给空闲线程执行。CachedThreadPool中任务传递的示意图如图10-7所示。

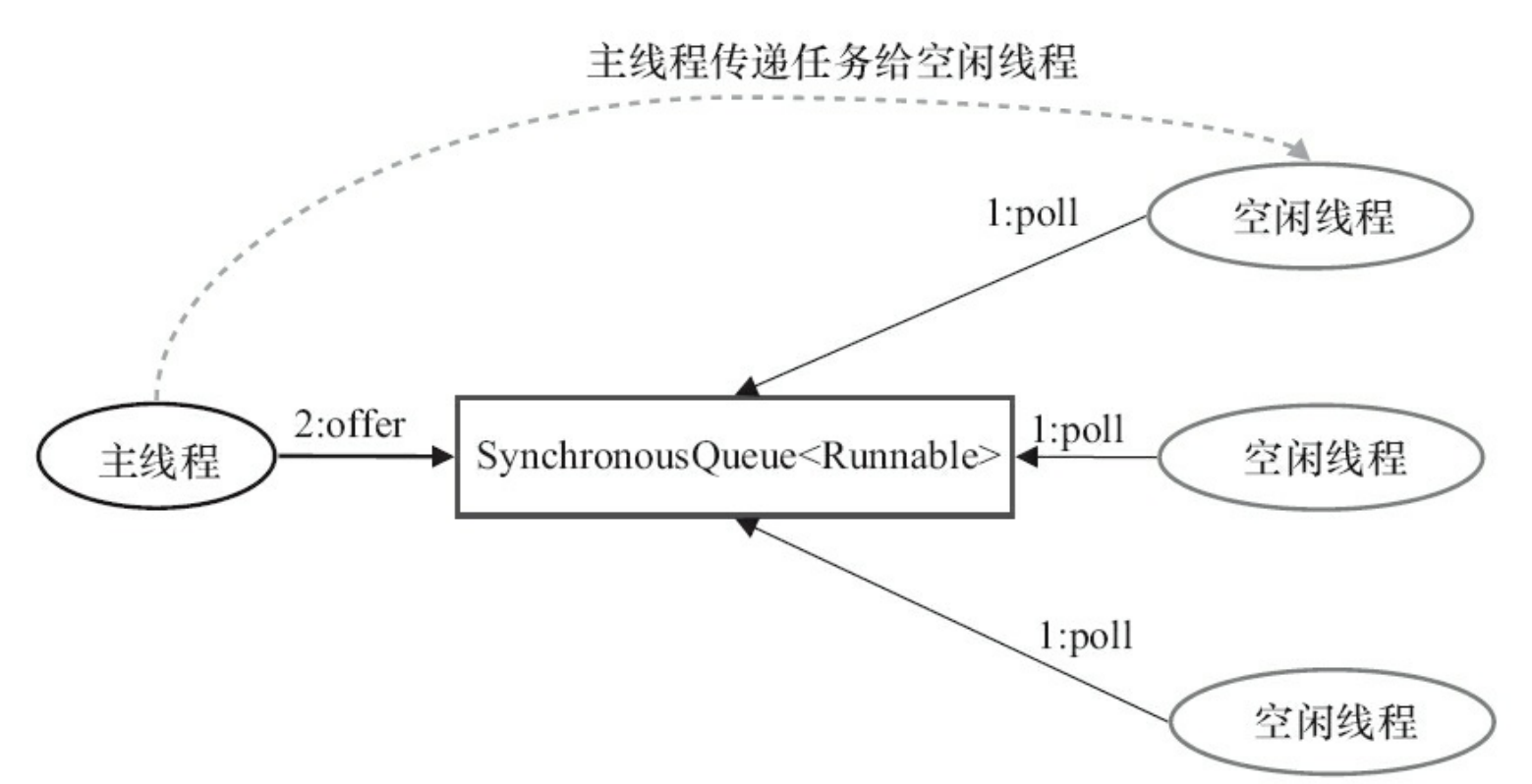


图10-7 CachedThreadPool的任务传递示意图

10.3 ScheduledThreadPoolExecutor详解

ScheduledThreadPoolExecutor继承自ThreadPoolExecutor。它主要用来在给定的延迟之后运行任务，或者定期执行任务。ScheduledThreadPoolExecutor的功能与Timer类似，但ScheduledThreadPoolExecutor功能更强大、更灵活。Timer对应的是单个后台线程，而ScheduledThreadPoolExecutor可以在构造函数中指定多个对应的后台线程数。

10.3.1 ScheduledThreadPoolExecutor的运行机制

ScheduledThreadPoolExecutor的执行示意图(本文基于JDK 6)如图10-8所示。

DelayQueue是一个无界队列, 所以ThreadPoolExecutor的maximumPoolSize在ScheduledThreadPoolExecutor中没有什么意义(设置maximumPoolSize的大小没有什么效果)。

ScheduledThreadPoolExecutor的执行主要分为两大部分。

1) 当调用ScheduledThreadPoolExecutor的scheduleAtFixedRate()方法或者scheduleWithFixedDelay()方法时, 会向ScheduledThreadPoolExecutor的DelayQueue添加一个实现了RunnableScheduledFuture接口的ScheduledFutureTask。

2) 线程池中的线程从DelayQueue中获取ScheduledFutureTask, 然后执行任务。

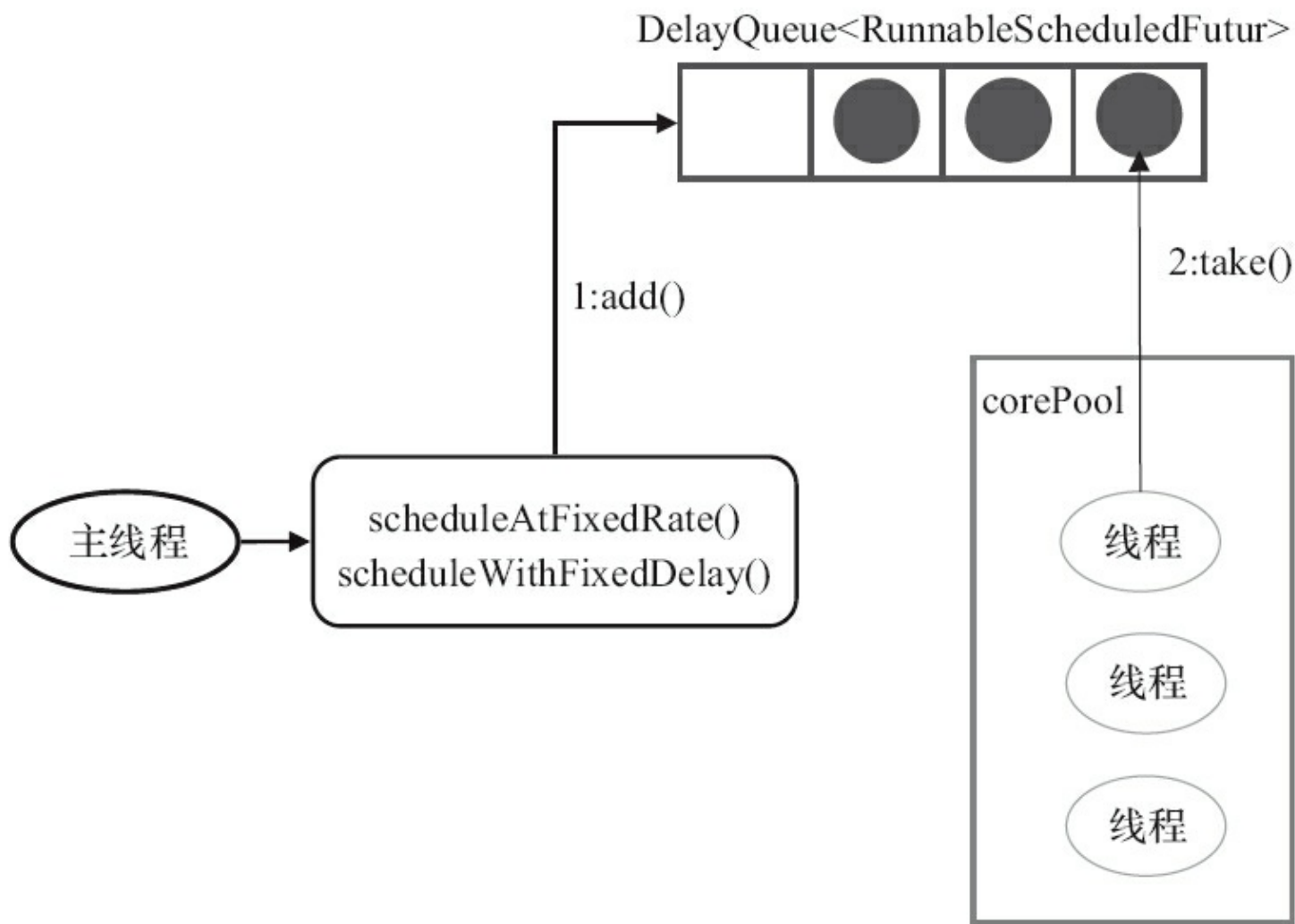


图10-8 `ScheduledThreadPoolExecutor`的任务传递示意图

`ScheduledThreadPoolExecutor`为了实现周期性的执行任务，对`ThreadPoolExecutor`做了如下的修改。

- 使用`DelayQueue`作为任务队列。
- 获取任务的方式不同(后文会说明)。
- 执行周期任务后，增加了额外的处理(后文会说明)。

10.3.2 ScheduledThreadPoolExecutor的实现

前面我们提到过，ScheduledThreadPoolExecutor会把待调度的任务(ScheduledFutureTask)放到一个DelayQueue中。

ScheduledFutureTask主要包含3个成员变量，如下。

- long型成员变量time，表示这个任务将要被执行的具体时间。

- long型成员变量sequenceNumber，表示这个任务被添加到ScheduledThreadPoolExecutor中的序号。

- long型成员变量period，表示任务执行的间隔周期。

DelayQueue封装了一个PriorityQueue，这个PriorityQueue会对队列中的ScheduledFutureTask进行排序。排序时，time小的排在前面(时间早的任务将被先执行)。如果两个ScheduledFutureTask的time相同，就比较sequenceNumber，sequenceNumber小的排在前面(也就是说，如果两个任务的执行时间相同，那么先提交的任务将被先执行)。

首先，让我们看看ScheduledThreadPoolExecutor中的线程执行周期任务的过程。图10-9是ScheduledThreadPoolExecutor中的线程1执行某个周期任务的4个步骤。

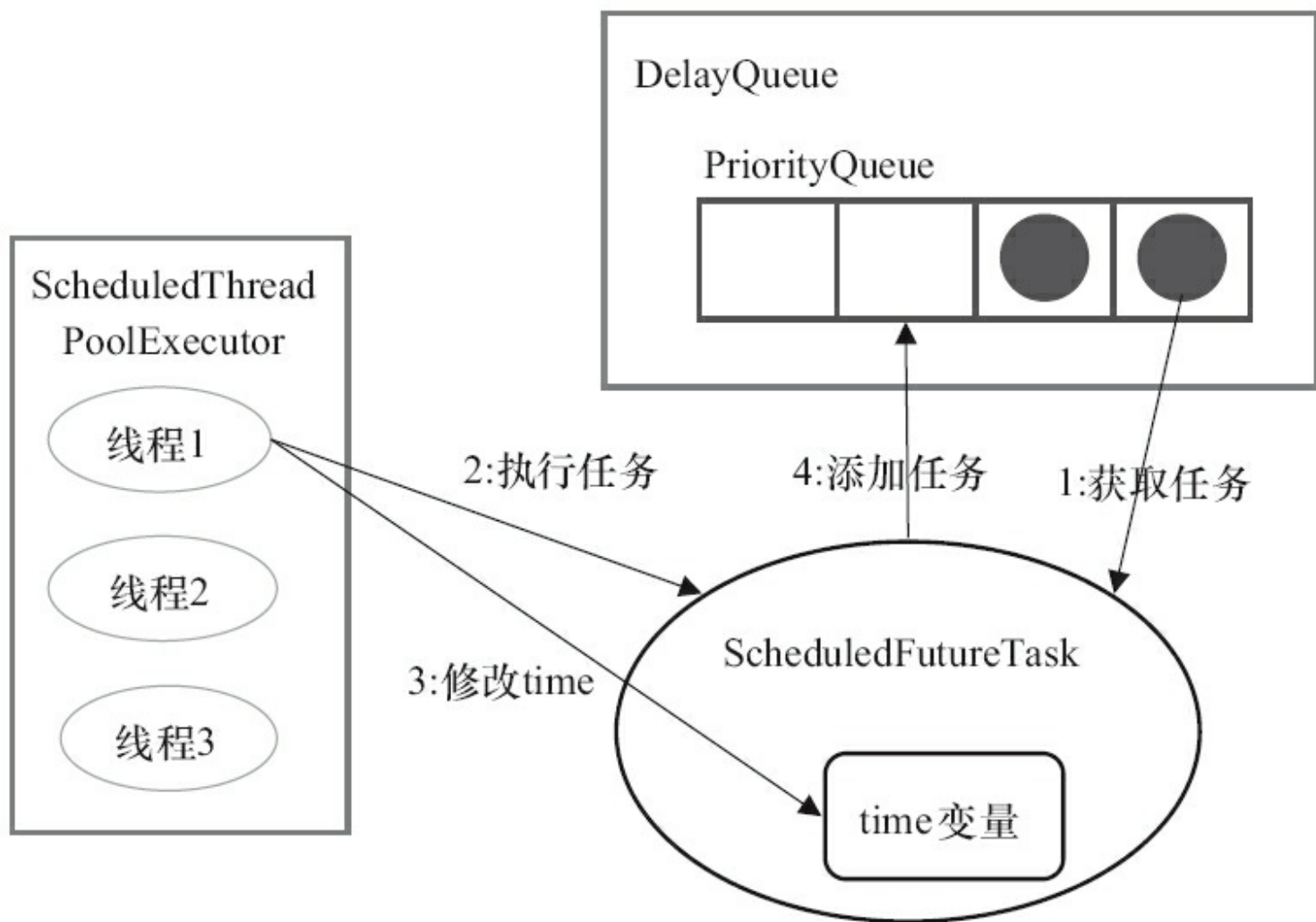


图10-9 ScheduledThreadPoolExecutor的任务执行步骤

下面是对这4个步骤的说明。

- 1) 线程1从DelayQueue中获取已到期的ScheduledFutureTask(`DelayQueue.take()`)。到期任务是指ScheduledFutureTask的time大于等于当前时间。
- 2) 线程1执行这个ScheduledFutureTask。
- 3) 线程1修改ScheduledFutureTask的time变量为下次将要被执行的时间。
- 4) 线程1把这个修改time之后的ScheduledFutureTask放回DelayQueue中(`DelayQueue.add()`)。

接下来, 让我们看看上面的步骤1) 获取任务的过程。下面是DelayQueue.take()方法的源代

码实现。

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();                                // 1
    try {
        for (;;) {
            E first = q.peek();
            if (first == null) {
                available.await();                            // 2.1
            } else {
                long delay = first.getDelay(TimeUnit.NANOSECONDS);
                if (delay > 0) {
                    long tl = available.awaitNanos(delay);    // 2.2
                } else {
                    E x = q.poll();                            // 2.3.1
                    assert x != null;
                    if (q.size() != 0)
                        available.signalAll();                // 2.3.2
                    return x;
                }
            }
        }
    } finally {
        lock.unlock();                                        // 3
    }
}
```

图10-10是DelayQueue.take()的执行示意图。

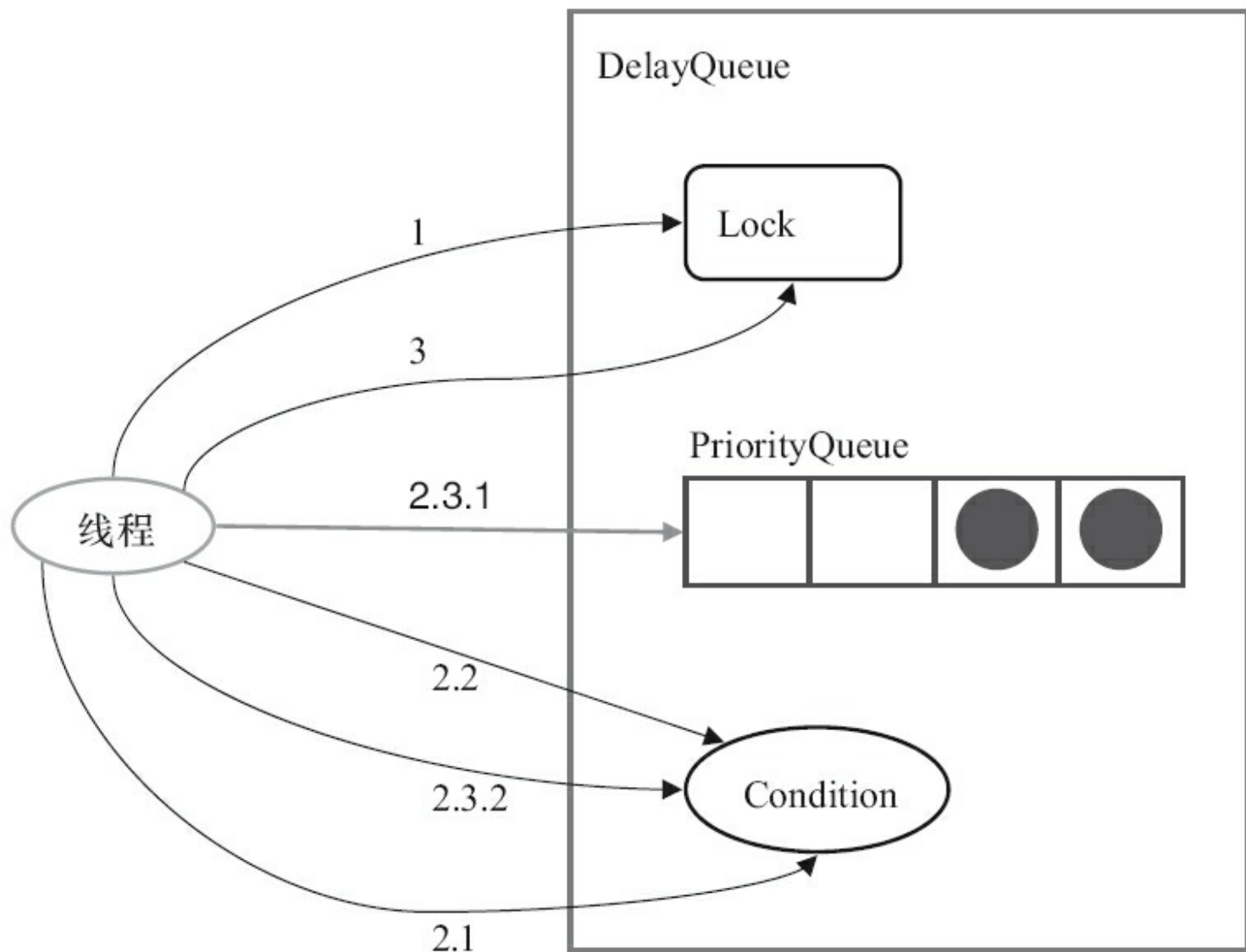


图10-10 ScheduledThreadPoolExecutor获取任务的过程

如图所示，获取任务分为3大步骤。

1) 获取Lock。

2) 获取周期任务。

·如果PriorityQueue为空，当前线程到Condition中等待；否则执行下面的2.2。

·如果PriorityQueue的头元素的time时间比当前时间大，到Condition中等待到time时间；否则执行下面的2.3。

·获取PriorityQueue的头元素(2.3.1)；如果PriorityQueue不为空，则唤醒在Condition中等待

的所有线程(2.3.2)。

3)释放Lock。

ScheduledThreadPoolExecutor在一个循环中执行步骤2, 直到线程从PriorityQueue获取到一个元素之后(执行2.3.1之后), 才会退出无限循环(结束步骤2)。

最后, 让我们看看ScheduledThreadPoolExecutor中的线程执行任务的步骤4, 把ScheduledFutureTask放入DelayQueue中的过程。下面是DelayQueue.add()的源代码实现。

```
public boolean offer(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();                // 1
    try {
        E first = q.peek();
        q.offer(e);              // 2.1
        if (first == null || e.compareTo(first) < 0)
            available.signalAll(); // 2.2
        return true;
    } finally {
        lock.unlock();          // 3
    }
}
```

图10-11是DelayQueue.add()的执行示意图。

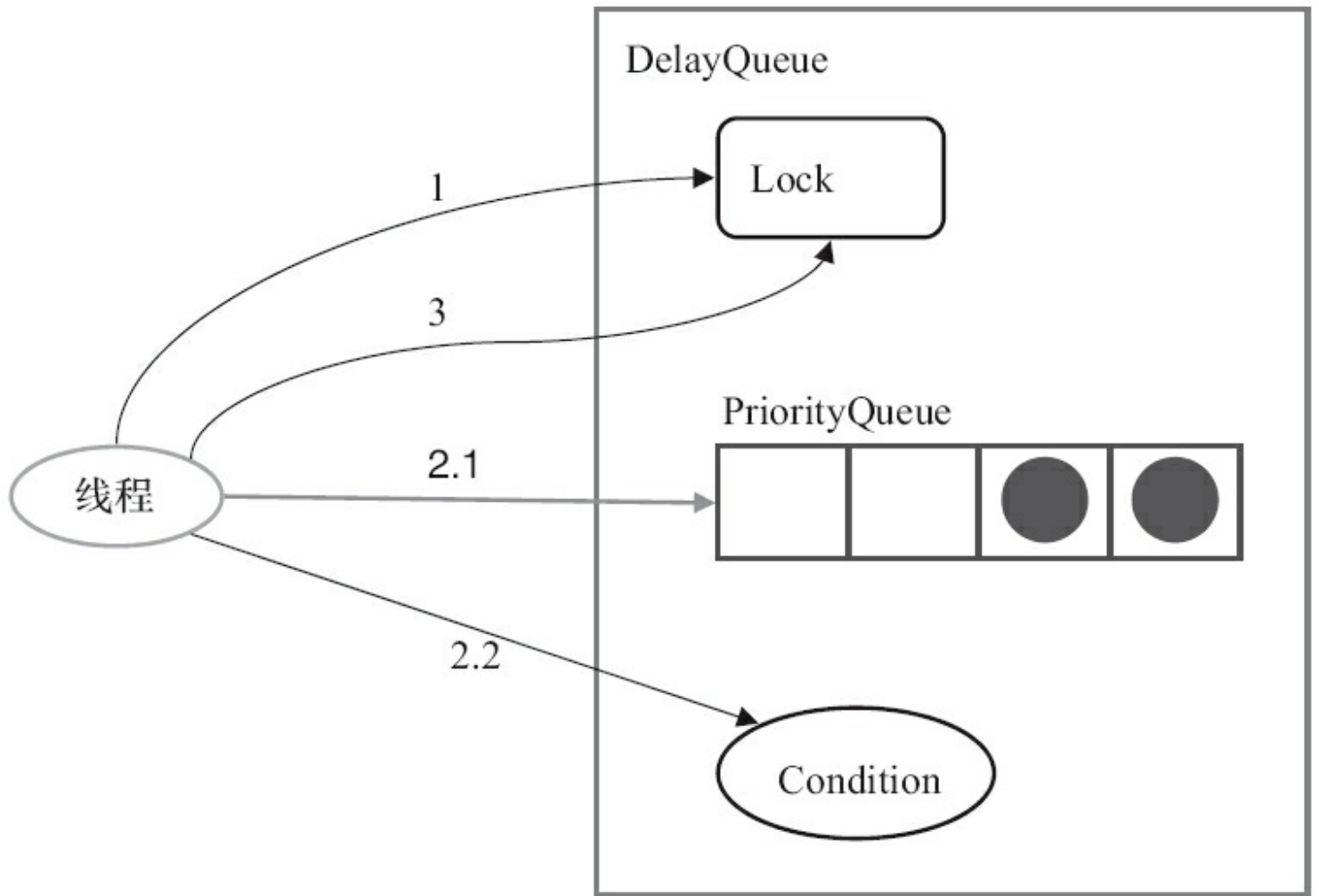


图10-11 `ScheduledThreadPoolExecutor`添加任务的过程

如图所示，添加任务分为3大步骤。

1) 获取Lock。

2) 添加任务。

·向PriorityQueue添加任务。

·如果在上面2.1中添加的任务是PriorityQueue的头元素，唤醒在Condition中等待的所有线程。

3) 释放Lock。

10.4 FutureTask详解

Future接口和实现Future接口的FutureTask类, 代表异步计算的结果。

10.4.1 FutureTask简介

FutureTask除了实现Future接口外，还实现了Runnable接口。因此，FutureTask可以交给Executor执行，也可以由调用线程直接执行(FutureTask.run())。根据FutureTask.run()方法被执行的时机，FutureTask可以处于下面3种状态。

- 1) 未启动。FutureTask.run()方法还没有被执行之前，FutureTask处于未启动状态。当创建一个FutureTask，且没有执行FutureTask.run()方法之前，这个FutureTask处于未启动状态。
- 2) 已启动。FutureTask.run()方法被执行的过程中，FutureTask处于已启动状态。
- 3) 已完成。FutureTask.run()方法执行完后正常结束，或被取消(FutureTask.cancel(...))，或执行FutureTask.run()方法时抛出异常而异常结束，FutureTask处于已完成状态。

图10-12是FutureTask的状态迁移的示意图。

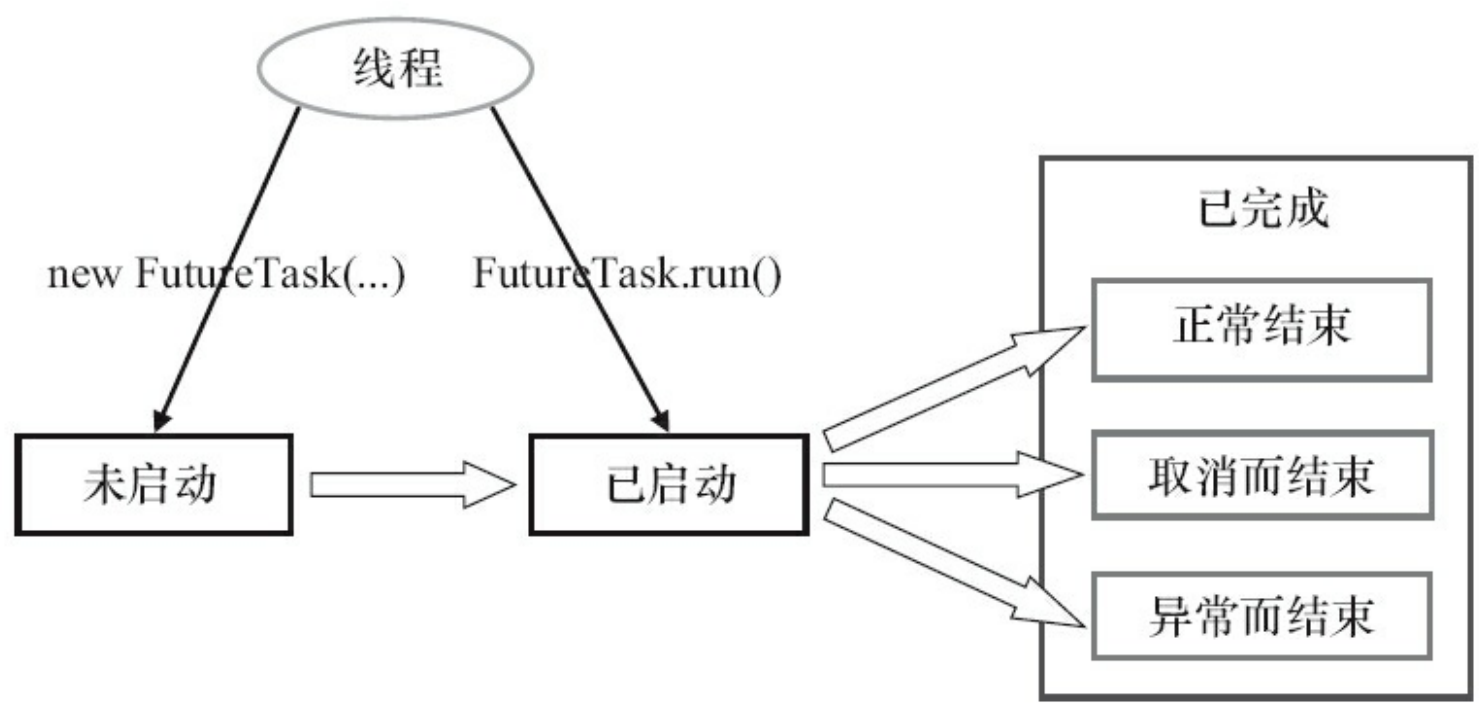


图10-12 FutureTask的状态迁移示意图

当FutureTask处于未启动或已启动状态时，执行FutureTask.get()方法将导致调用线程阻塞；

当FutureTask处于已完成状态时，执行FutureTask.get()方法将导致调用线程立即返回结果或抛出异常。

当FutureTask处于未启动状态时，执行FutureTask.cancel()方法将导致此任务永远不会被执行；当FutureTask处于已启动状态时，执行FutureTask.cancel(true)方法将以中断执行此任务线程的方式来试图停止任务；当FutureTask处于已启动状态时，执行FutureTask.cancel(false)方法将不会对正在执行此任务的线程产生影响（让正在执行的任务运行完成）；当FutureTask处于已完成状态时，执行FutureTask.cancel(...)方法将返回false。

图10-13是get方法和cancel方法的执行示意图。

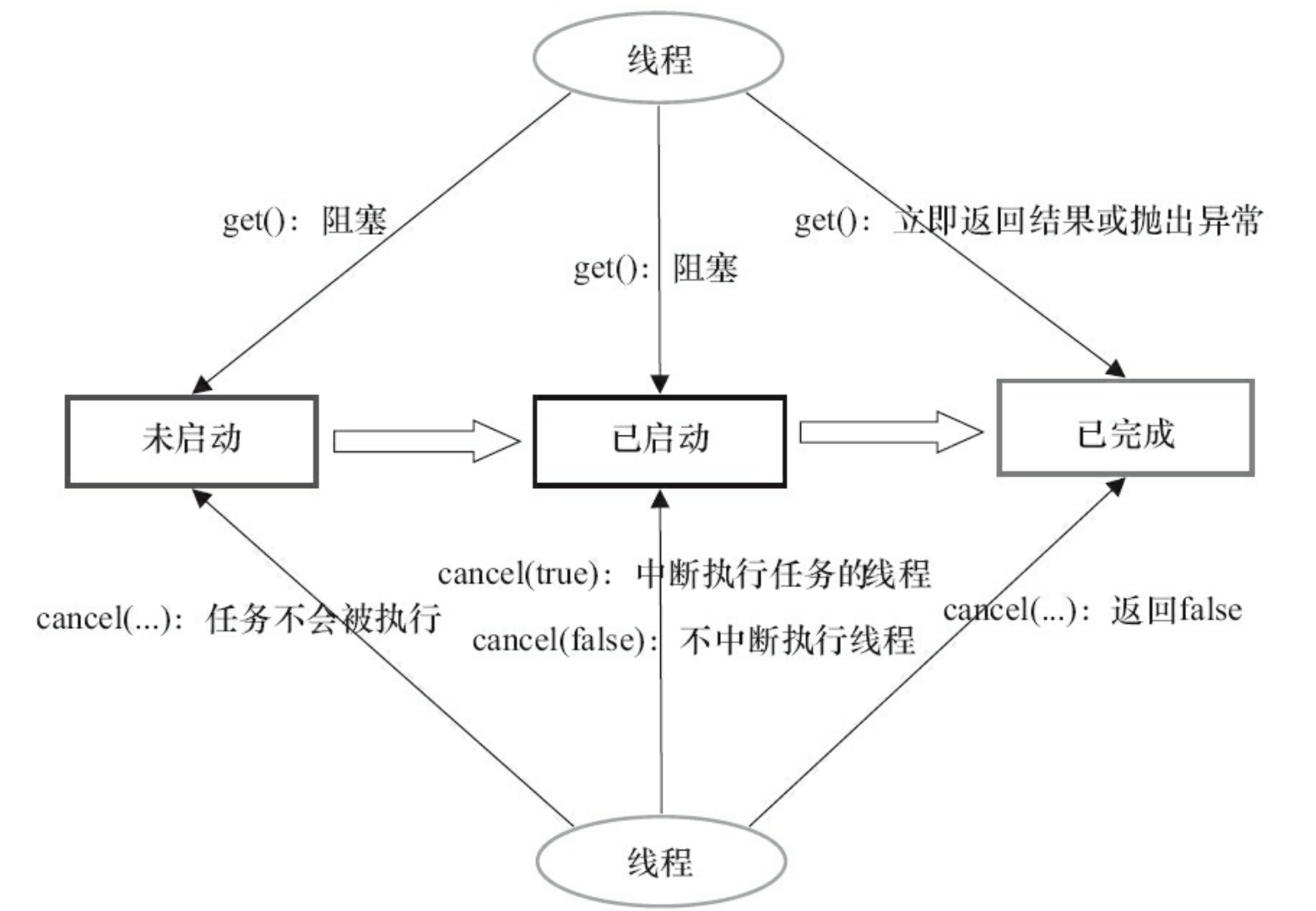


图10-13 FutureTask的get和cancel的执行示意图

10.4.2 FutureTask的使用

可以把FutureTask交给Executor执行;也可以通过ExecutorService.submit(...)方法返回一个FutureTask, 然后执行FutureTask.get()方法或FutureTask.cancel(...)方法。除此以外, 还可以单独使用FutureTask。

当一个线程需要等待另一个线程把某个任务执行完后它才能继续执行, 此时可以使用FutureTask。假设有多个线程执行若干任务, 每个任务最多只能被执行一次。当多个线程试图同时执行同一个任务时, 只允许一个线程执行任务, 其他线程需要等待这个任务执行完后才能继续执行。下面是对应的示例代码。

```
private final ConcurrentMap<Object, Future<String>> taskCache =
    new ConcurrentHashMap<Object, Future<String>>();
private String executionTask(final String taskName)
    throws ExecutionException, InterruptedException {
    while (true) {
        Future<String> future = taskCache.get(taskName);           // 1.1,2.1
        if (future == null) {
            Callable<String> task = new Callable<String>() {
                public String call() throws InterruptedException {
                    return taskName;
                }
            };
            FutureTask<String> futureTask = new FutureTask<String>(task);
            future = taskCache.putIfAbsent(taskName, futureTask); // 1.3
            if (future == null) {
                future = futureTask;
                futureTask.run();                                   // 1.4执行任务
            }
        }
        try {
            return future.get();                                     // 1.5
        } catch (CancellationException e) {
            taskCache.remove(taskName, future);
        }
    }
}
```

上述代码的执行示意图如图10-14所示。

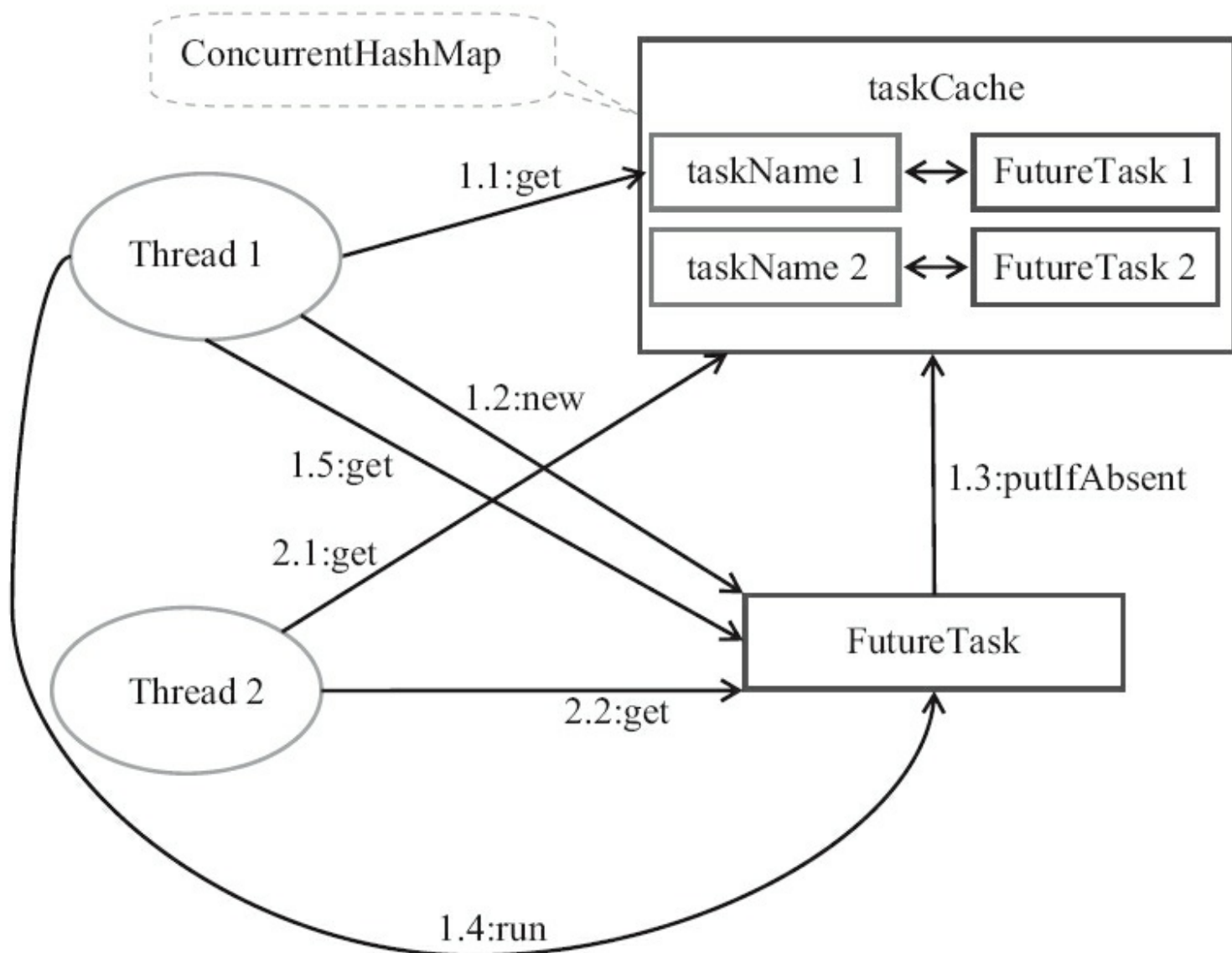


图10-14 代码的执行示意图

当两个线程试图同时执行同一个任务时，如果Thread 1执行1.3后Thread 2执行2.1，那么接下来Thread 2将在2.2等待，直到Thread 1执行完1.4后Thread 2才能从2.2 (FutureTask.get()) 返回。

10.4.3 FutureTask的实现

FutureTask的实现基于AbstractQueuedSynchronizer(以下简称为AQS)。java.util.concurrent中的很多可阻塞类(比如ReentrantLock)都是基于AQS来实现的。AQS是一个同步框架,它提供通用机制来原子性管理同步状态、阻塞和唤醒线程,以及维护被阻塞线程的队列。JDK 6中AQS被广泛使用,基于AQS实现的同步器包括:ReentrantLock、Semaphore、ReentrantReadWriteLock、CountDownLatch和FutureTask。

每一个基于AQS实现的同步器都会包含两种类型的操作,如下。

- 至少一个acquire操作。这个操作阻塞调用线程,除非/直到AQS的状态允许这个线程继续执行。FutureTask的acquire操作为get()/get(long timeout, TimeUnit unit)方法调用。
- 至少一个release操作。这个操作改变AQS的状态,改变后的状态可允许一个或多个阻塞线程被解除阻塞。FutureTask的release操作包括run()方法和cancel(...)方法。

基于“复合优先于继承”的原则,FutureTask声明了一个内部私有的继承于AQS的子类Sync,对FutureTask所有公有方法的调用都会委托给这个内部子类。

AQS被作为“模板方法模式”的基础类提供给FutureTask的内部子类Sync,这个内部子类只需要实现状态检查和状态更新的方法即可,这些方法将控制FutureTask的获取和释放操作。具体来说,Sync实现了AQS的tryAcquireShared(int)方法和tryReleaseShared(int)方法,Sync通过这两个方法来检查和更新同步状态。

FutureTask的设计示意图如图10-15所示。

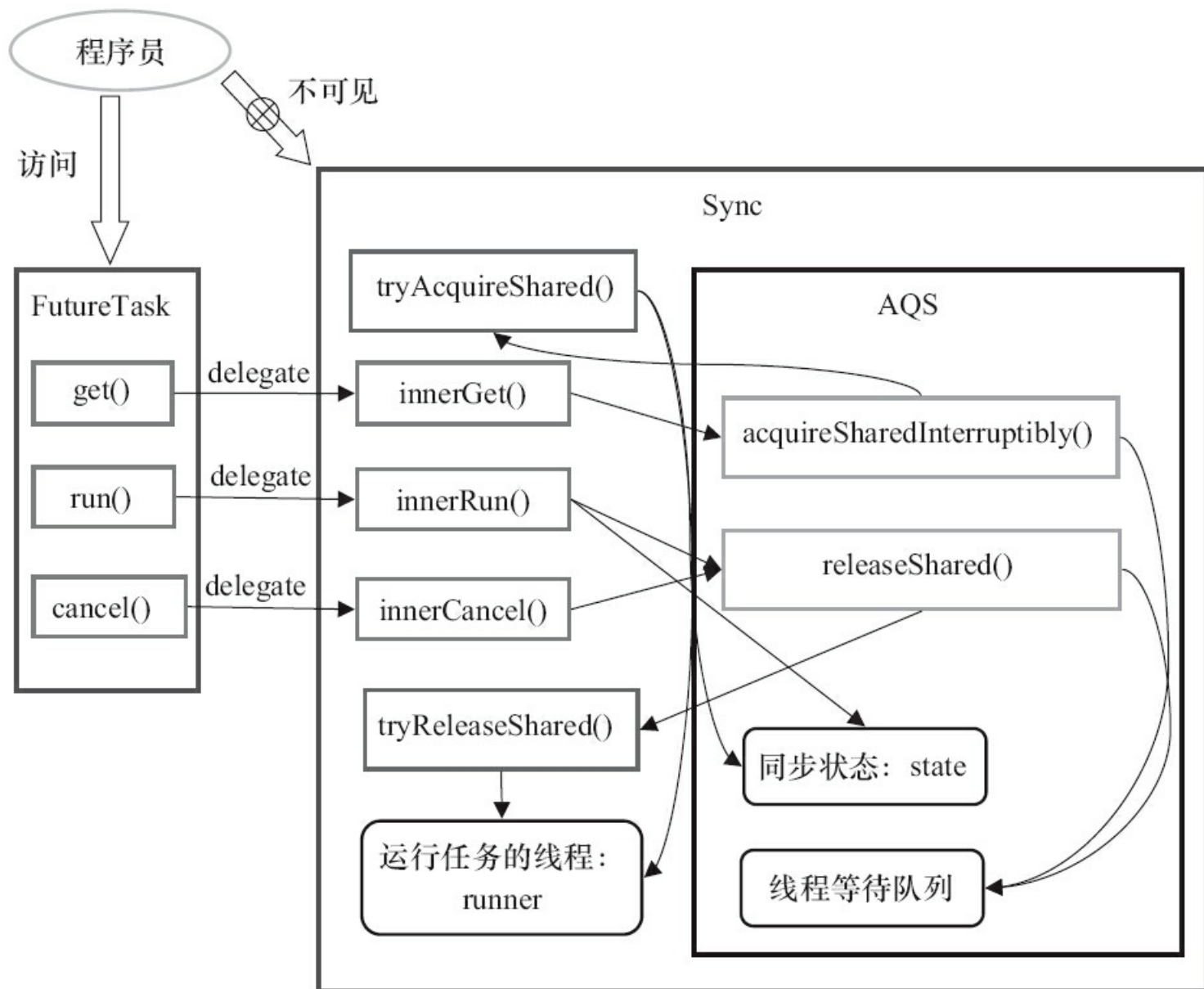


图10-15 FutureTask的设计示意图

如图所示，Sync是FutureTask的内部私有类，它继承自AQS。创建FutureTask时会创建内部私有的成员对象Sync，FutureTask所有的的公有方法都直接委托给了内部私有的Sync。

FutureTask.get()方法会调用AQS.acquireSharedInterruptibly(int arg)方法，这个方法的执行过程如下。

1) 调用AQS.acquireSharedInterruptibly(int arg)方法，这个方法首先会回调在子类Sync中实现的tryAcquireShared()方法来判断acquire操作是否可以成功。acquire操作可以成功的条件为：state为执行完成状态RAN或已取消状态CANCELLED，且runner不为null。

2) 如果成功则`get()`方法立即返回。如果失败则到线程等待队列中去等待其他线程执行`release`操作。

3) 当其他线程执行`release`操作(比如`FutureTask.run()`或`FutureTask.cancel(...)`)唤醒当前线程后, 当前线程再次执行`tryAcquireShared()`将返回正值1, 当前线程将离开线程等待队列并唤醒它的后继线程(这里会产生级联唤醒的效果, 后面会介绍)。

4) 最后返回计算的结果或抛出异常。

`FutureTask.run()`的执行过程如下。

1) 执行在构造函数中指定的任务(`Callable.call()`)。

2) 以原子方式来更新同步状态(调用`AQS.compareAndSetState(int expect, int update)`, 设置`state`为执行完成状态`RAN`)。如果这个原子操作成功, 就设置代表计算结果的变量`result`的值为`Callable.call()`的返回值, 然后调用`AQS.releaseShared(int arg)`。

3) `AQS.releaseShared(int arg)`首先会回调在子类`Sync`中实现的`tryReleaseShared(arg)`来执行`release`操作(设置运行任务的线程`runner`为`null`, 然会返回`true`); `AQS.releaseShared(int arg)`, 然后唤醒线程等待队列中的第一个线程。

4) 调用`FutureTask.done()`。

当执行`FutureTask.get()`方法时, 如果`FutureTask`不是处于执行完成状态`RAN`或已取消状态`CANCELLED`, 当前执行线程将到`AQS`的线程等待队列中等待(见下图的线程A、B、C和D)。当某个线程执行`FutureTask.run()`方法或`FutureTask.cancel(...)`方法时, 会唤醒线程等待队列的第一个线程(见图10-16所示的线程E唤醒线程A)。

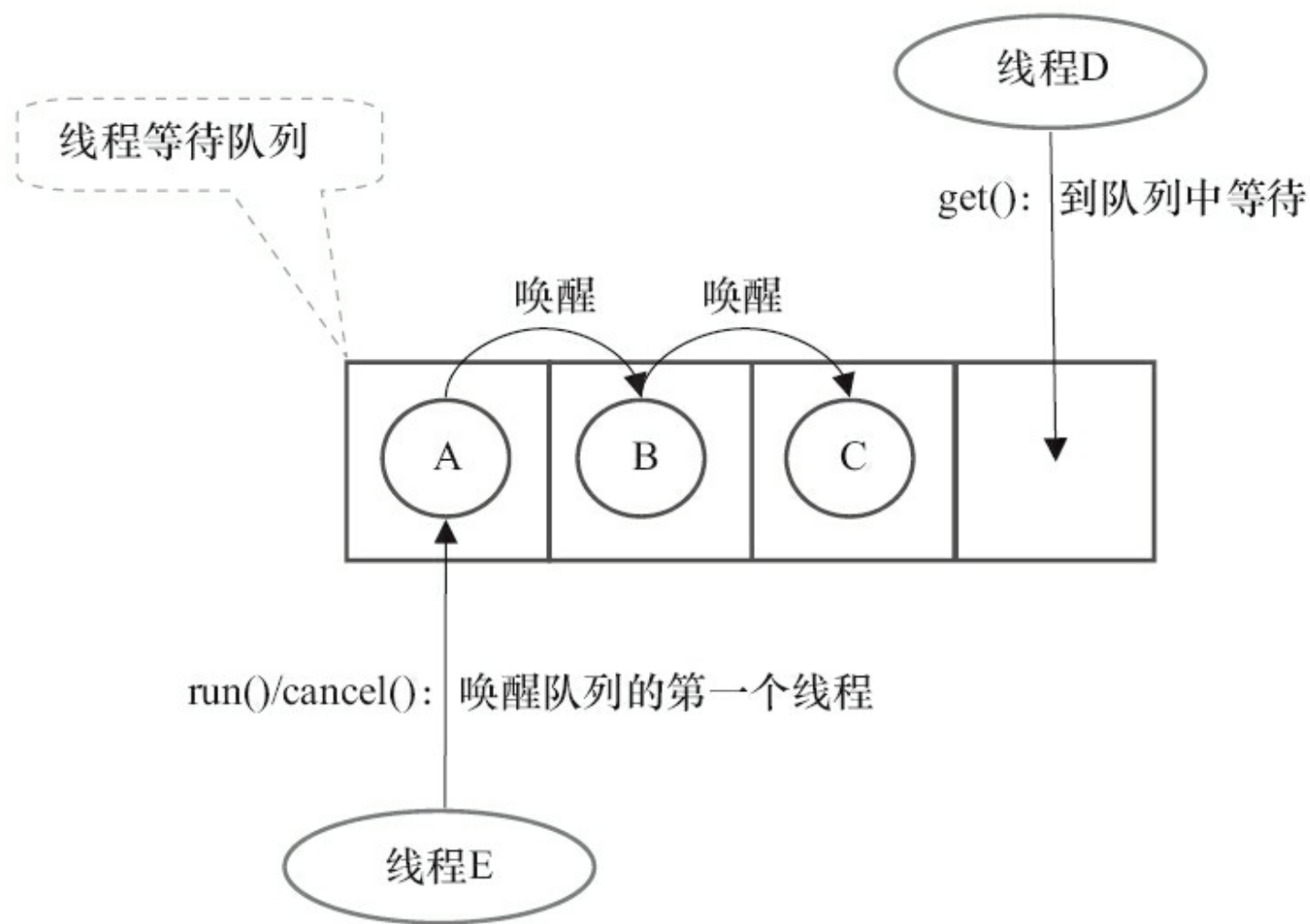


图10-16 FutureTask的级联唤醒示意图

假设开始时FutureTask处于未启动状态或已启动状态，等待队列中已经有3个线程(A、B和C)在等待。此时，线程D执行get()方法将导致线程D也到等待队列中去等待。

当线程E执行run()方法时，会唤醒队列中的第一个线程A。线程A被唤醒后，首先把自己从队列中删除，然后唤醒它的后继线程B，最后线程A从get()方法返回。线程B、C和D重复A线程的处理流程。最终，在队列中等待的所有线程都被级联唤醒并从get()方法返回。

10.5 本章小结

本章介绍了Executor框架的整体结构和成员组件。希望读者阅读本章之后，能够对Executor框架有一个比较深入的理解，同时也希望本章内容有助于读者更熟练地使用Executor框架。