Wait until any of Future<T> is done

Asked 16 years, 3 months ago Modified 4 years, 1 month ago Viewed 39k times

58

I have few asynchronous tasks running and I need to wait until at least one of them is finished (in the future probably I'll need to wait util M out of N tasks are finished). Currently they are presented as Future, so I need something like

```
/**
 * Blocks current thread until one of specified futures is done and returns it.
 */
public static <T> Future<T> waitForAny(Collection<Future<T>> futures)
        throws AllFuturesFailedException
```

Is there anything like this? Or anything similar, not necessary for Future. Currently I loop through collection of futures, check if one is finished, then sleep for some time and check again. This looks like not the best solution, because if I sleep for long period then unwanted delay is added, if I sleep for short period then it can affect performance.

I could try using

```
new CountDownLatch(1)
```

and decrease countdown when task is complete and do

```
countdown.await()
```

, but I found it possible only if I control Future creation. It is possible, but requires system redesign, because currently logic of tasks creation (sending Callable to ExecutorService) is separated from decision to wait for which Future. I could also override

```
<T> RunnableFuture<T> AbstractExecutorService.newTaskFor(Callable<T> callable)
```

and create custom implementation of RunnableFuture with ability to attach listener to be notified when task is finished, then attach such listener to needed tasks and use CountDownLatch, but that means I have to override newTaskFor for every ExecutorService I use - and potentially there will be implementation which do not extend AbstractExecutorService. I could also try wrapping given ExecutorService for same purpose, but then I have to decorate all methods producing Futures.

All these solutions may work but seem very unnatural. It looks like I'm missing something simple, like

```
WaitHandle.WaitAny(WaitHandle[] waitHandles)
```

in c#. Are there any well known solutions for such kind of problem?

UPDATE:

Originally I did not have access to Future creation at all, so there were no elegant solution. After redesigning system I got access to Future creation and was able to add countDownLatch.countdown() to execution process, then I can countDownLatch.await() and everything works fine. Thanks for other answers, I did not know about ExecutorCompletionService and it indeed can be helpful in similar tasks, but in this particular case it could not be used because some Futures are created without any executor - actual task is sent to another server via network, completes remotely and completion notification is received.

java multithreading concurrency

Share
Improve this question
Follow

edited Sep 23, 2008 at 7:13

asked Sep 22, 2008 at 21:13
Pavel Feldman
4,719 ● 7 ● 33 ● 32

4 Many people who come to this question will probably prefer looking below the long accepted answer to the higher-voted answers that refer to `ExecutorCompletionService` and `ExecutorService.invokeAny()` . – Robert Tupelo-Schneck Nov 6, 2013 at 19:52

8 Answers

Sorted by: Highest score (default)

55

simple, check out ExecutorCompletionService.

Share
Improve this answer
Follow

edited Nov 2, 2012 at 11:11
pingw33n
12.5k ● 2 ● 38 ● 38

answered Sep 22, 2008 at 23:42
james
699 ● 4 ● 3

6 The documentation for this class, including an example of how to cancel all other tasks after the first one completes (if that is what you want to do) can be found at

java.sun.com/javase/6/docs/api/java/util/concurrent/… – Bill Michell Sep 23, 2008 at 9:36

4 ExecutorCompletionService can't accept Futures, AFAICT this does not answer the original question. – Charlie May 14, 2016 at 22:45

Curiously, I was always punished for single-word answers in the past. I guess, it is because I am not a decent human. – Little Alien Sep 21, 2016 at 18:00

9

ExecutorService.invokeAny

Share Improve this answer Follow

answered Sep 23, 2008 at 3:45
ykaganovich
14.9k ● 8 ● 62 ● 98

4 Almost. It takes a `Collection<Callable>` rather than a `Collection<Future>` – finnw Jul 17, 2010 at 12:24

7

Why not just create a results queue and wait on the queue? Or more simply, use a CompletionService since that's what it is: an ExecutorService + result queue.

Share Improve this answer Follow

answered Sep 23, 2008 at 1:42
Alex Miller
70.1k ● 25 ● 124 ● 168

6

This is actually pretty easy with wait() and notifyAll().

First, define a lock object. (You can use any class for this, but I like to be explicit):

```
package com.javadude.sample;

public class Lock {}
```

Next, define your worker thread. He must notify that lock object when he's finished with his processing. Note that the notify must be in a synchronized block locking on the lock object.

```
package com.javadude.sample;

public class Worker extends Thread {
    private Lock lock_;
    private long timeToSleep_;
    private String name_;
    public Worker(Lock lock, String name, long timeToSleep) {
        lock_ = lock;
        timeToSleep_ = timeToSleep;
        name_ = name;
    }
    @Override
    public void run() {
        // do real work -- using a sleep here to simulate work
        try {
            sleep(timeToSleep_);
        } catch (InterruptedException e) {
            interrupt();
        }
        System.out.println(name_ + " is done... notifying");
        // notify whoever is waiting, in this case, the client
        synchronized (lock_) {
            lock_.notify();
        }
    }
}
```

Finally, you can write your client:

```
package com.javadude.sample;

public class Client {
    public static void main(String[] args) {
        Lock lock = new Lock();
        Worker worker1 = new Worker(lock, "worker1", 15000);
        Worker worker2 = new Worker(lock, "worker2", 10000);
        Worker worker3 = new Worker(lock, "worker3", 5000);
        Worker worker4 = new Worker(lock, "worker4", 20000);

        boolean started = false;
        int numNotifies = 0;
        while (true) {
            synchronized (lock) {
                try {
                    if (!started) {
                        // need to do the start here so we grab the lock, just
                        //   in case one of the threads is fast -- if we had
done the
                        //   starts outside the synchronized block, a fast
thread could
                        //   get to its notification *before* the client is
waiting for it
                        worker1.start();
                        worker2.start();
                        worker3.start();
                        worker4.start();
                        started = true;
                    }
                    lock.wait();
                } catch (InterruptedException e) {
                    break;
                }
                numNotifies++;
                if (numNotifies == 4) {
                    break;
                }
                System.out.println("Notified!");
            }
        }
        System.out.println("Everyone has notified me... I'm done");
    }
}
```

Share Improve this answer Follow

answered Sep 22, 2008 at 23:08

Scott Stanchfield
30.6k ● 9 ● 51 ● 67

Your solution is what I was thinking of. It seems clean and easier to understand. +1
– Jacob Schoen Sep 23, 2008 at 0:18

4

As far as I know, Java has no analogous structure to the `WaitHandle.WaitAny` method.

It seems to me that this could be achieved through a "WaitableFuture" decorator:

```
public WaitableFuture<T>
    extends Future<T>
{
    private CountDownLatch countDownLatch;

    WaitableFuture(CountDownLatch countDownLatch)
    {
        super();

        this.countDownLatch = countDownLatch;
    }

    void doTask()
    {
        super.doTask();

        this.countDownLatch.countDown();
    }
}
```

Though this would only work if it can be inserted before the execution code, since otherwise the execution code would not have the new `doTask()` method. But I really see no way of doing this without polling if you cannot somehow gain control of the Future object before execution.

Or if the future always runs in its own thread, and you can somehow get that thread. Then you could spawn a new thread to join each other thread, then handle the waiting mechanism after the join returns... This would be really ugly and would induce a lot of overhead though. And if some Future objects don't finish, you could have a lot of blocked threads depending on dead threads. If you're not careful, this could leak memory and system resources.

```
/**
 * Extremely ugly way of implementing WaitHandle.WaitAny for Thread.Join().
 */
public static joinAny(Collection<Thread> threads, int numberToWaitFor)
{
    CountDownLatch countDownLatch = new CountDownLatch(numberToWaitFor);

    foreach(Thread thread in threads)
    {
        (new Thread(new JoinThreadHelper(thread, countDownLatch))).start();
    }

    countDownLatch.await();
}

class JoinThreadHelper
    implements Runnable
{
    Thread thread;
    CountDownLatch countDownLatch;

    JoinThreadHelper(Thread thread, CountDownLatch countDownLatch)
    {
        this.thread = thread;
        this.countDownLatch = countDownLatch;
    }

    void run()
    {
        this.thread.join();
        this.countDownLatch.countDown();
    }
}
```

Share

Improve this answer

Follow

edited Sep 22, 2008 at 21:36

answered Sep 22, 2008 at 21:18

jdmichal

11.1k ● 4 ● 46 ● 42

2

If you can use `CompletableFuture`s instead then there is `CompletableFuture.anyOf` that does what you want, just call join on the result:

```
CompletableFuture.anyOf(futures).join()
```

You can use `CompletableFuture`s with executors by calling the `CompletableFuture.supplyAsync` or `runAsync` methods.

Share Improve this answer Follow

answered Oct 28, 2020 at 8:03

Alex - GlassEditor.com
15.5k ● 6 ● 50 ● 49

0

Since you don't care which one finishes, why not just have a single WaitHandle for all threads and wait on that? Whichever one finishes first can set the handle.

Share Improve this answer Follow

answered Sep 22, 2008 at 21:18

1800 INFORMATION
135k ● 30 ● 163 ● 242

1 I do not control Futures creation. And do not always have separate thread per Future. Some tasks are dispatched via network to be executed on separate JVM, after notification of completion is received, Future is marked as done. I'm redesigning it now to control Future creation to add countDownLatch – Pavel Feldman Sep 22, 2008 at 21:55

-1

See this option:

```
public class WaitForAnyRedux {

private static final int POOL_SIZE = 10;

public static <T> T waitForAny(Collection<T> collection) throws
InterruptedException, ExecutionException {

    List<Callable<T>> callables = new ArrayList<Callable<T>>();
    for (final T t : collection) {
        Callable<T> callable = Executors.callable(new Thread() {

            @Override
            public void run() {
                synchronized (t) {
                    try {
                        t.wait();
                    } catch (InterruptedException e) {
                    }
                }
            }
        }, t);
        callables.add(callable);
    }

    BlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>
(POOL_SIZE);
```

```
    ExecutorService executorService = new ThreadPoolExecutor(POOL_SIZE,
POOL_SIZE, 0, TimeUnit.SECONDS, queue);
    return executorService.invokeAny(callables);
}

static public void main(String[] args) throws InterruptedException,
ExecutionException {

    final List<Integer> integers = new ArrayList<Integer>();
    for (int i = 0; i < POOL_SIZE; i++) {
        integers.add(i);
    }

    (new Thread() {
        public void run() {
            Integer notified = null;
            try {
                notified = waitForAny(integers);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
            System.out.println("notified=" + notified);
        }

    }).start();


    synchronized (integers) {
        integers.wait(3000);
    }


    Integer randomInt = integers.get((new Random()).nextInt(POOL_SIZE));
    System.out.println("Waking up " + randomInt);
    synchronized (randomInt) {
        randomInt.notify();
    }
  }
}
```

Share

Improve this answer

Follow

edited Sep 3, 2013 at 12:06

Crowie
3,272 ● 7 ● 30 ● 50

answered Jan 8, 2009 at 10:02

fabrice armisen