Java Concurrency Tutorial – Callable, Future

javacodegeeks.com Byron Kiourtzoglou September 17th, 2011 view original

One of the beautiful things about Java from its very first release was the ease with which we could write multi-threaded programs and introduce asynchronous processing into our designs. The **Thread** class and **Runnable** interface combined with Java's memory management model meant for straightforward thread programming. But as discussed in Part 3, neither the **Thread** class nor the **Runnable** interface allowed for thrown **Exceptions** or returned values. The lack of returned values was mildly annoying.
The lack of thrown checked exceptions was a little more serious. The contract was **public void run()** which meant you had to catch checked exceptions and do something with them. Even if you were careful and you stored these for later verification, you couldn't force all uses of the class to check the exception. You could go through all your getters and throw the **Exception** if it existed on each one. Besides being cumbersome, even that wasn't foolproof. You couldn't enforce calls to any of these. Thread programmers would correctly call **join()** to wait for it complete and may then have gone on their merry way.

Not to worry though, after many years, this was finally addressed in the 1.5 release. With the introduction of the **Callable** and **Future** interfaces and their support in the thread pools discussed in our last post, both of these issues have been addressed quite elegantly.

Callable

The **Callable** interface declares **public T call() throws Exception**. Now we can return a result, have it strongly typed as declared in our implementation and even throw **Exceptions**. While there are some utility methods in the **Executors** class to convert your **Runnable** instances as discussed in Part 3, you would do well to review your current implementations of **Runnable** or subclasses of **Thread**. Why bother? Primarily to double check and remove the workaround you may have implemented to address the lack of support for thrown **Exceptions**. At the same time, you may wish to make use of the ability to return results right in the execution method eliminating any need to cast to retrieve values.

Future

Here's where the combined power of the thread pools and **Callable** come together. **Future** is another new interface introduced in 1.5. When you submit a **Callable** to one of the thread pools, you are provided an instance of **Future** that is typed to the **Callable** you passed in. This object substitutes for an actual **Thread** instance that you would have used prior to 1.5. Whereas you previously had to do **Thread.join()** or **Thread.join(long millis)**, now you may use them as in this example.

```
public class ServerAcceptingRequestsVerifier implements
Callable {
        /**
         * @return Boolean.TRUE is server is accepting
requests
         * Boolean.FALSE otherwise
         */
        public Boolean call() throws Exception {
                Boolean isAcceptingRequests = null;
                ... ask server about taking requests here
                return isAcceptingRequests;
        }
}
public Boolean isServerTakingRequests(String server)
                        throws UnresponsiveException,
InterruptedException {
        ServerAcceptingRequestsVerifier
acceptingRequestsVerifier =
                new ServerAcceptingRequestsVerifier();
        Future future =

THREAD_POOL.submit(acceptingRequestsVerifier);
        try {
                Boolean isAcceptingRequests =
future.get();
                //waits for the thread to complete, even
```

```
    if it hasn't started
                    return isAcceptingRequests;
            } catch (ExecutionException e) {
                    throw new
    UnresponsiveException(e.getCause());
            }

    }
```

It's also nice that we now have explicit **TimeoutException** if we decide to limit how long we're willing to wait for completion.

```
    try {
            Boolean isAcceptingRequests = future.get(5,
    TimeUnit.SECONDS);
            //this waits for 5 seconds, throwing
    TimeoutException if not done
            return isAcceptingRequests;
    } catch (TimeoutException e) {
            LOGGER.warn("Timed out waiting for server check
    thread." +
                    "We'll try to interrupt it.");
            future.cancel(true);
            return Boolean.FALSE;
    } catch (ExecutionException e) {
            throw new UnresponsiveException(e.getCause());
    }
```

In our next post, we'll get into some of the new interfaces/classes that are used to make the thread pools work that are available for our use too.

Reference: Java Concurrency Part 4 – Callable, Future from our JCG partners at the Carfey Software blog.