Active Objects Pattern Futures with .NET Framework Task Parallel Library

Posted by **Jeffrey Juday** on **June 20th, 2011**

Active Object is a common pattern for hiding access to concurrent data structures and simplifying an object's interface. Task Parallel Library (TPL) includes all the structures needed to build an Active Object.

Of course concurrency cannot always be completely hidden from a consuming client. For example, how does a method on an Active Object handle a return value to the client when the value is being generated somewhere in a Threadpool? The Active Object Pattern guidelines recommend using a Future. In TPL a Future is implemented using the Task class. (See Understanding Tasks in .NET Framework 4.0 Task Parallel Library).

There are multiple TPL approaches to implementing the Active Object Pattern's Future. A few of those approaches will be examined in this article.

Active Object Overview

The Active Object Pattern has the following conventions:

- It has a Proxy supporting client interaction.
- It includes Request messages that encapsulate the desired Proxy invocation.
- A Scheduler receives the Request messages from the Proxy and maintains a Request message Queue. The Scheduler runs on a Thread separate from the client containing the Proxy and handles execution scheduling.
- A Servant performs the execution according to the Scheduler's Request message.
- The Active Object can return a value using a callback mechanism like a Future. This scenario is the focus of this article.

The graphic below depicts the interaction between the components above.

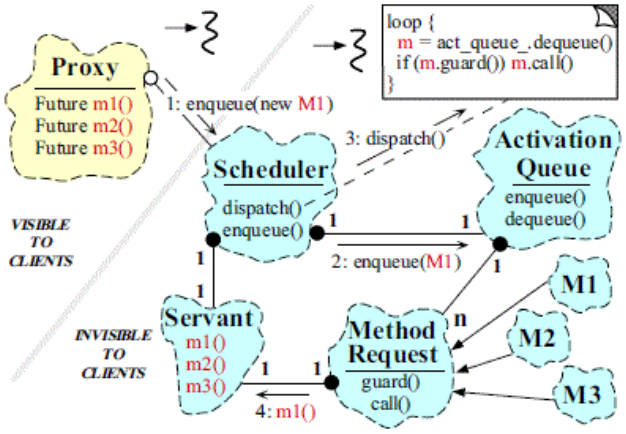

Figure 1: Source: "Active Object An Object Behavioral Pattern for Concurrent Programming" http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf

As stated earlier the article will focus on different approaches to implementing the Future return value in TPL.

Wait in the Proxy

The first approach is to leverage a Task inside of the Active Object Proxy. A client using the application blocks until the operation completes. A timeout on the Task.Wait ensures that the client will not block forever. Sample code for the approach appears below.

```
1.              var d = proxy.Print_Synchronous("Some data 1");
2.              Console.WriteLine("Return Synchronously " + d);
3.
4.          public string Print_Synchronous(string data)
5.          {
6.              var t = ProcessScheduler.ScheduleCompletion(new ExecuteRequest(Guid.NewGuid().ToString(), data));
7.
8.              t.Wait(1000);
```

```
 9.
10.             return t.Result;
11.         }
```

A benefit of this approach is the client never deals with concurrency unless the Task.Wait times out. The Proxy must choose how to handle the generated Exception. Task Exceptions are packaged in an AggregateException. A complete review of the AggregateException is beyond the scope of this article.

The approach works well if the operation is expected to have a maximum duration. To handle a variable timeout; durations can be configured in a configuration file or passed into the method.

On the downside; the operation blocks the client. If blocking is not desirable a developer can return a Task class to the client.

Task

A complete introduction to the Task class is beyond the scope of this article, but there is an introduction here: Understanding Tasks in .NET Framework 4.0 Task Parallel Library**.** The Task sample appears below.

```
 1.             var t2 = proxy.Print_Task(Guid.NewGuid().ToString(),"Some data 3");
 2.
 3.             Task.WaitAll(new Task[2] { t1, t2 });
 4.
 5.
 6.         public Task<string> Print_Task(string jobId, string data)
 7.         {
 8.             return ProcessScheduler.ScheduleTask(new ExecuteRequest(jobId, data));
 9.         }
10.
11.         public static Task<string> ScheduleTask(ExecuteRequest request)
12.         {
13.             var job = new JobTask(request, _servant);
14.
15.             return job.Run();
16.         }
17.
18.     internal class JobTask
19.     {
20.         private ExecuteRequest _request = null;
21.         private Servant _servant = null;
22.
23.         public JobTask(ExecuteRequest request, Servant servant)
24.         {
25.             _request = request;
26.             _servant = servant;
27.         }
28.
29.         public Task<string> Run()
30.         {
31.             return Task.Factory.StartNew<string>(() =>
32.             {
33.                 return _servant.Execute(_request);
34.             }
35.             );
36.
37.         }
38.     }
```

Print_Task returns a Task<string> class. A consuming client can Wait on the Task class or invoke a Continuation in response to a Task class. Unlike the synchronous example; the client can make multiple invocations and wait on completion of all the invoked Tasks.

With multiple Tasks a consuming client may have difficulty correlating the completed Task back to the originating invocation. A Guid, int, or a string all make good correlating identifiers. Unlike the sample, however, it may be more helpful to wrap the results and the identifier in another data structure.

A variation on the Task is to use a TaskCompletionSource inside the Active Object.

TaskCompletionSource

Like the Task example, the TaskCompletionSource sample returns a Task class. The TaskCompletionSource sample appears below.

```
 1.             var t1 = proxy.Print_Completion(Guid.NewGuid().ToString(),"Some data 2");
 2.
 3.             Task.WaitAll(new Task[2] { t1, t2 });
 4.
 5.         public static Task<string> ScheduleCompletion(ExecuteRequest request)
 6.         {
 7.             var job = new JobCompletion(request, _servant);
 8.
```

```
            job.Run();

            return job.Completion.Task;
        }

    internal class JobCompletion
    {
        public TaskCompletionSource<string> Completion = new TaskCompletionSource<string>();

        private ExecuteRequest _request = null;
        private Servant _servant = null;

        public JobCompletion(ExecuteRequest request, Servant servant)
        {
            _request = request;
            _servant = servant;
        }

        public void Run()
        {
            Task.Factory.StartNew(() =>
            {
                _servant.Execute(_request, Completion);
            }
            );

        }
    }

public void Execute(ExecuteRequest request, TaskCompletionSource<string> completion)
{
    Console.WriteLine("Starting print " + request.Context + "...");
    Thread.Sleep(750);


    Console.WriteLine("Done printing " + request.Context + " " + request.Data + "...");

    completion.TrySetResult(request.Context);//only use this with the Async methods

}
```

TaskCompletionSource allows a developer to control the completion and result of a Task class; rather than tying the Task result to the result of an Action delegate.

TaskCompletionSource is best used for the Asynchronous Programming Model (APM) or another **.NET** operation that either behaves asynchronously or executes in its own Thread. Rather than tying up a Thread in the Threadpool; an APM operation includes the TaskCompletionSource in the operation state.

Cancellation

No TPL article would be complete without mentioning Cancellation. Aborting running operations require a CancellationToken and an Action delegate checking for the cancellation status. There are many ways a consuming client could cancel the work. For example: the Active Object could include an additional Cancel method that accepts the Task object and maps the Task back to a CancellationToken.

A complete review of Cancellations is beyond the scope of this article; but for a good resource, see Understanding .NET Framework Task Parallel Library Cancellations.

Conclusion

Active Object is a common pattern for hiding access to concurrent data structures and simplifying an object's interface. Active Object implementation difficulties arise when a method must return a value back to the client via a Future or Callback. Task Parallel Library is equipped with Tasks and TaskCompletionSource object to ease the difficulty.

Resources

"Active Object: An Object Behavioral Pattern for Concurrent Programming"

"Know When to Use an Active Object Instead of a Mutex"

"Prefer Using Futures or Callbacks to Communicate Asynchronous Results"

About the Author

Jeffrey Juday

Jeff is a software developer specializing in enterprise application integration solutions utilizing BizTalk, SharePoint, WCF, WF, and SQL Server. Jeff has been developing software with Microsoft tools for more than 15 years in a variety of industries including: military, manufacturing, financial services, management consulting, and computer security. Jeff is a Microsoft BizTalk MVP. Jeff spends his spare time with his wife Sherrill and daughter Alexandra.

Related Articles

- Understanding .NET Framework Task Parallel Library Cancellations
- .NET Framework Task Parallel Library and the Active Objects Pattern
- Microsoft .NET Reactive Extensions and .NET Framework Task Parallel Library
- Quick Guide to .NET Framework Task Parallel Library Visual Studio 2010 Debugger