عنوان: **Long Polling در WCF**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۱/۱۷ ۱۴:۱۵
آدرس: www.dotnettips.info
گروه‌ها: WCF, Long Polling

به صورت پیش فرض سرویس‌های WCF به صورت Sync اجرا خواهند شد، یعنی هر گاه درخواستی از سمت کلاینت به سرور ارسال شود سرور بعد از پردازش درخواست پاسخ مورد نظر را به کلاینت باز می‌گرداند. اما حالتی را در نظر بگیرید که بعد از دریافت Request از کلاینت بنا به دلایلی امکان پاسخ گویی سمت سرور در آن لحظه وجود ندارد. خوب چه اتفاقی خواهد افتاد؟ در این حالت thread جاری سمت کلاینت نیز در حالت wait است و برنامه سمت کلاینت از کار می‌افتد تا زمانی که پاسخ از سرور دریافت نماید. اما در WCF به صورت پیش فرض هر درخواست ارسالی باید در طی یک دقیقه در اختیار سرور قرار گیرد و سرور نیز باید در طی یک دقیقه پاسخ مورد نظر را برگرداند(مقادیر خواص SendTimeout و ReceiveTimeout برای مدیریت این موارد به کار می‌روند). افزایش مقادیر این خواص کمک خاصی به این حالت نمی‌کند زیرا هم چنان کلاینت در حالت wait است و سرور نیز پاسخ خاصی ارسال نمی‌کند. حتی اگر کل عملیات را به صورت Async پیاده سازی نماییم باز ممکن است بعد از منقضی شدن زمان پردازش با یک TimeoutException برنامه از کار بیفتد. برای حل اینگونه موارد پیاده سازی سرویس‌ها به صورت Long Polling به ما کمک خوبی خواهد کرد.

حال سناریو زیر را در نظر بگیرید:

سمت سرور:

»یک درخواست دریافت می‌شود؛

»سرور در حالت wait (البته توسط یک thread دیگر) منتظر تامین منابع برای پاسخ به کلاینت است؛

»در نهایت پاسخ مورد نظر ارسال خواهد شد.

سمت کلاینت:

»درخواست مورد نظر به سرور ارسال می‌شود؛

»کلاینت منتظر پاسخ از سمت سرور است(البته توسط یک Thread دیگر)؛

»اگر در حین انتظار برای پاسخ از سمت سرور، با یک TimeoutException روبرو شدیم به جای توقف برنامه و نمایش پیغام خطای Server is not available، باید عملیات به صورت خودکار restart شود.

»در نهایت پاسخ مورد نظر دریافت خواهد شد.

پیاده سازی این سناریو در WCF کار پیچیده ای نیست. بدین منظور می‌توانید از کلاس زیر استفاده کنید(لینک دانلود). سورس آن به صورت زیر است:

```
public abstract class LongPollingAsyncResult<TResult> : IAsyncResult where TResult : class
    {
        #region - Fields -

        private AsyncCallback _callback;
        private TimeSpan _timoutSpan;
        private TimeSpan _intervalWaitSpan;

        #endregion

        #region - Properties -
        public Exception Exception { get; private set; }

        public TResult Result { get; private set; }

        public object SyncRoot { get; private set; }

        #endregion

        #region - Ctor -

        public LongPollingAsyncResult(AsyncCallback callback, object asyncState, int timeoutSeconds =
300, int intervalWaitMilliseconds = 500)
        {
            SyncRoot = new object();
            _callback = callback;
            AsyncState = asyncState;
            AsyncWaitHandle = new ManualResetEvent(IsCompleted);
            _timoutSpan = TimeSpan.FromSeconds(timeoutSeconds);
            _intervalWaitSpan = TimeSpan.FromMilliseconds(intervalWaitMilliseconds);
```

```
            ThreadPool.QueueUserWorkItem(new WaitCallback(LoopWithIntervalAndTimeout));
        }

        #endregion

        #region - Private Helper Methods -

        private void LoopWithIntervalAndTimeout(object input)
        {
            try
            {
                Stopwatch stopwatch = new Stopwatch();
                stopwatch.Start();
                while (!IsCompleted)
                {
                    if (stopwatch.Elapsed > _timoutSpan)
                        throw new TimeoutException();

                    DoWork();

                    if (!IsCompleted)
                        Thread.Sleep(_intervalWaitSpan);
                }
            }
            catch (Exception e)
            {
                Complete(null, e);
            }
        }

        #endregion

        #region - Protected/Abstract Methods -
        protected void Complete(TResult result, Exception e = null, bool completedSynchronously =
false)
        {
            lock (SyncRoot)
            {
                CompletedSynchronously = completedSynchronously;
                Result = result;
                Exception = e;
                IsCompleted = true;

                if (_callback != null)
                    _callback(this);
            }
        }

        protected abstract void DoWork();

        #endregion

        #region - Public Methods -

        public TResult WaitForResult()
        {
            if (!IsCompleted)
                AsyncWaitHandle.WaitOne();

            if (Exception != null)
            {
                if (Exception is TimeoutException && WebOperationContext.Current != null)
                    WebOperationContext.Current.OutgoingResponse.StatusCode =
HttpStatusCode.RequestTimeout;

                throw Exception;
            }

            return Result;
        }

        #endregion

        #region - IAsyncResult Implementation -

        public object AsyncState { get; private set; }

        public WaitHandle AsyncWaitHandle { get; private set; }

        public bool CompletedSynchronously { get; private set; }
```

```
        public bool IsCompleted { get; private set; }

        #endregion
    }
```

در این حالت شما می‌توانید حداکثر زمان مورد نیاز برای درخواست را به عنوان پارامتر از طریق سازنده کلاس بالا تعیین نمایید. اگر این زمان بیش از زمان تعیین شده در خواص SendTimeout و ReceiveTimeout بود بعد از منقضی شدن زمان پردازش درخواست، به جای دریافت TimeoutException عملیات پردازش به کار خود ادامه خواهد داد.
برای استفاده از کلاس تهیه شده ابتدا باید عملیات خود را به صورت Async پیاده سازی نمایید که در این مقاله به صورت کامل شرح داده شده است.

یک مثال

قصد داریم Operation زیر را به صورت Long Polling پیاده سازی نماییم:

```
[OperationContract]
public string GetNotification();
```

ابتدا متد زیر باید به صورت Async تبدیل شود:

```
[OperationContract(AsyncPattern = true)]
public IAsyncResult BeginWaitNotification(AsyncCallback callback, object state);

public string EndWaitNotification(IAsyncResult result);
```

حال نوع بازگشتی سرویس مورد نظر را با استفاده از کلاس LongPollingAsyncResult به صورت زیر ایجاد خواهیم کرد:

```
public class MyNotificationResult : LongPollingAsyncResult<string>
{
   protected override DoWork()
    {
        کدهای مورد نظر را اینجا قرار دهید
        base.Complete(...)
    }
}
```

در نهایت پیاده سازی متدهای Begin و End همانند ذیل خواهد بود:

```
public IAsyncResult BeginWaitNotification(AsyncCallback callback, object state)
{
    return new MyNotificationResult(callback, state);
}

public string EndWaitNotification(IAsyncResult result)
{
    MyNotificationResult myResult = result as MyNotificationResult;
    if(myResult == null)
        throw new ArgumentException("result was of the wrong type!");

    myResult.WaitForResult();
    return myResult.Result;
}
```

در این حالت کلاینت می‌تواند یک درخواست به صورت LongPolling به سرور ارسال نماید و البته مدیریت این درخواست در یک thread دیگر انجام می‌گیرد که نتیجه آن از عدم تداخل پردازش این درخواست با سایر قسمت‌های برنامه است.