

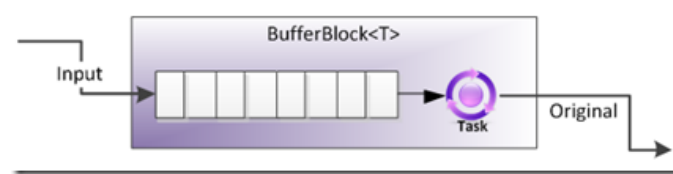
TPL DataFlow初探（一）

属性TPL Dataflow是微软面向高并发应用而推出的一个类库。借助于异步消息传递与管道，它可以提供比线程池更好的控制，也比手工线程方式具备更好的性能。我们常常可以消息传递，生产-消费模式或Actor-Agent模式中使用。在TDF是构建于Task Parallel Library (TPL)之上的，它是我们开发高性能，高并发的应用程序的又一利器。您可以在NuGet中下载使用，目前最新的版本只支持.net framework 4.5。最早支持.net framework 4.0 是作为Microsoft Visual Studio Async CTP中的一部分发布的，你可以在[这里](#)下载到。

TDP的主要作用就是Buffering Data和Processing Data，在TDF中，有两个非常重要的接口，ISourceBlock<T> 和ITargetBlock<T>接口。继承于ISourceBlock<T>的对象时作为提供数据的数据源对象-生产者，而继承于ITargetBlock<T>接口类主要是扮演目标对象-消费者。在这个类库中，System.Threading.Tasks.Dataflow名称空间下，提供了很多以Block名字结尾的类，ActionBlock，BufferBlock，TransformBlock，BroadcastBlock等9个Block，我们在开发中通常使用单个或多个Block组合的方式来实现一些功能。下面我们逐个来简单介绍一下。

BufferBlock

BufferBlock是TDF中最基础的Block。BufferBlock提供了一个有界限或没有界限的Buffer，该Buffer中存储T。该Block很像BlockingCollection<T>。可以用过Post往里面添加数据，也可以通过Receive方法阻塞或异步的获取数据，数据处理的顺序是FIFO的。它也可以通过Link向其他Block输出数据。



简单的同步的生产者消费者代码示例：

```
private static BufferBlock<int> m_buffer = new BufferBlock<int>();

// Producer
private static void Producer()
{
    while(true)
    {
        int item = Produce();
        m_buffer.Post(item);
    }
}

// Consumer
private static void Consumer()
{
    while(true)
    {
        int item = m_buffer.Receive();
        Process(item);
    }
}

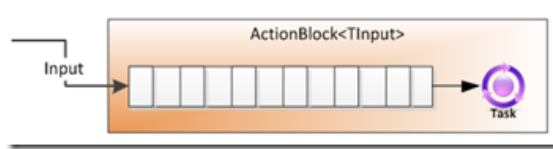
// Main
public static void Main()
```

```
{
    var p = Task.Factory.StartNew(Producer);
    var c = Task.Factory.StartNew(Consumer);
    Task.WaitAll(p,c);
}
```



ActionBlock

ActionBlock实现ITargetBlock，说明它是消费数据的，也就是对输入的一些数据进行处理。它在构造函数中，允许输入一个委托，来对每一个进来的数据进行一些操作。如果使用Action(T)委托，那说明每一个数据的处理完成需要等待这个委托方法结束，如果使用了Func<TInput, Task>来构造的话，那么数据的结束将不是委托的返回，而是Task的结束。默认情况下，ActionBlock会FIFO的处理每一个数据，而且一次只能处理一个数据，一个处理完了再处理第二个，但也可以通过配置来并行的执行多个数据。



先看一个例子：



```
public ActionBlock<int> abSync = new ActionBlock<int>((i) =>
{
    Thread.Sleep(1000);
    Console.WriteLine(i + " ThreadId:" +
Thread.CurrentThread.ManagedThreadId + " Execute Time:" + DateTime.Now);
});

public void TestSync()
{
    for (int i = 0; i < 10; i++)
    {
        abSync.Post(i);
    }

    Console.WriteLine("Post finished");
}
```



```
Post finished
0 ThreadId:10 Execute Time:2013/2/25 10:30:25
1 ThreadId:10 Execute Time:2013/2/25 10:30:26
2 ThreadId:10 Execute Time:2013/2/25 10:30:27
3 ThreadId:10 Execute Time:2013/2/25 10:30:28
4 ThreadId:10 Execute Time:2013/2/25 10:30:29
5 ThreadId:10 Execute Time:2013/2/25 10:30:30
6 ThreadId:10 Execute Time:2013/2/25 10:30:31
7 ThreadId:10 Execute Time:2013/2/25 10:30:32
8 ThreadId:10 Execute Time:2013/2/25 10:30:33
9 ThreadId:10 Execute Time:2013/2/25 10:30:34
```

可见，ActionBlock是顺序处理数据的，这也是ActionBlock一大特性之一。主线程在往ActionBlock中Post数据以后马上返回，具体数据的处理是另外一个线程来做的。数据是异步处理的，但处理本身是同步的，这样在一定程度上保证数据处理的准确性。下面的例子是使用async和await。

```
public ActionBlock<int> abSync2 = new ActionBlock<int>(async (i) =>
{
    await Task.Delay(1000);
    Console.WriteLine(i + " ThreadId:" + Thread.CurrentThread.ManagedThreadId +
" Execute Time:" + DateTime.Now);
})
```

```
Post finished
0 ThreadId:11 Execute Time:2013/2/25 14:08:13
1 ThreadId:14 Execute Time:2013/2/25 14:08:14
2 ThreadId:13 Execute Time:2013/2/25 14:08:15
3 ThreadId:13 Execute Time:2013/2/25 14:08:16
4 ThreadId:14 Execute Time:2013/2/25 14:08:17
5 ThreadId:14 Execute Time:2013/2/25 14:08:18
6 ThreadId:11 Execute Time:2013/2/25 14:08:19
7 ThreadId:14 Execute Time:2013/2/25 14:08:20
8 ThreadId:13 Execute Time:2013/2/25 14:08:21
9 ThreadId:13 Execute Time:2013/2/25 14:08:22
```

虽然还是1秒钟处理一个数据，但是处理数据的线程会有不同。

如果你想异步处理多个消息的话，ActionBlock也提供了一些接口，让你轻松实现。在ActionBlock的构造函数中，可以提供一个ExecutionDataflowBlockOptions的类型，让你定义ActionBlock的执行选项，在下面例子中，我们定义了MaxDegreeOfParallelism选项，设置为3。目的让ActionBlock中的Item最多可以3个并行处理。

```
public ActionBlock<int> abAsync = new ActionBlock<int>((i) =>
{
    Thread.Sleep(1000);
    Console.WriteLine(i + " ThreadId:" + Thread.CurrentThread.ManagedThreadId +
" Execute Time:" + DateTime.Now);
}, new ExecutionDataflowBlockOptions() { MaxDegreeOfParallelism = 3 });

public void TestAsync()
{
    for (int i = 0; i < 10; i++)
    {
        abAsync.Post(i);
    }
    Console.WriteLine("Post finished");
}
```

```
Post finished
0 ThreadId:10 Execute Time:2013/2/25 10:29:32
2 ThreadId:11 Execute Time:2013/2/25 10:29:32
1 ThreadId:12 Execute Time:2013/2/25 10:29:32
4 ThreadId:11 Execute Time:2013/2/25 10:29:33
5 ThreadId:12 Execute Time:2013/2/25 10:29:33
3 ThreadId:10 Execute Time:2013/2/25 10:29:33
6 ThreadId:11 Execute Time:2013/2/25 10:29:34
8 ThreadId:10 Execute Time:2013/2/25 10:29:34
7 ThreadId:12 Execute Time:2013/2/25 10:29:34
9 ThreadId:11 Execute Time:2013/2/25 10:29:35
```

运行程序，我们看见，每3个数据几乎同时处理，并且他们的线程ID也是不一样的。

ActionBlock也有自己的生命周期，所有继承IDataflowBlock的类型都有Completion属性和Complete方法。调用Complete方法是让ActionBlock停止接收数据，而Completion属性则是一个Task，是在ActionBlock处理完所有数据时候会执行的任务，我们可以使用Completion.Wait()方法来等待ActionBlock完成所有的任务，Completion属性只有在设置了Complete方法后才会有效。

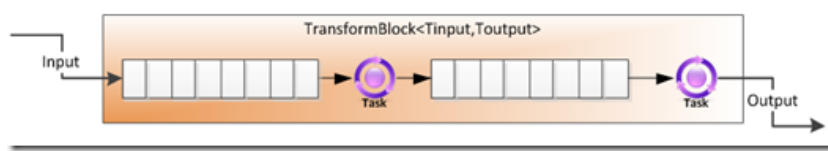
```
public void TestAsync()
{
    for (int i = 0; i < 10; i++)
    {
        abAsync.Post(i);
    }
    abAsync.Complete();
    Console.WriteLine("Post finished");
    abAsync.Completion.Wait();
    Console.WriteLine("Process finished");
}
```

```
Post finished
1 ThreadId:12 Execute Time:2013/2/25 11:17:37
0 ThreadId:11 Execute Time:2013/2/25 11:17:37
2 ThreadId:10 Execute Time:2013/2/25 11:17:37
3 ThreadId:12 Execute Time:2013/2/25 11:17:38
5 ThreadId:10 Execute Time:2013/2/25 11:17:38
4 ThreadId:11 Execute Time:2013/2/25 11:17:38
6 ThreadId:12 Execute Time:2013/2/25 11:17:39
7 ThreadId:10 Execute Time:2013/2/25 11:17:39
8 ThreadId:11 Execute Time:2013/2/25 11:17:39
9 ThreadId:12 Execute Time:2013/2/25 11:17:40
Process finished
```

TransformBlock

TransformBlock是TDF提供的另一种Block，顾名思义它常常在数据流中充当数据转换处理的功能。在TransformBlock内部维护了2个Queue，一个InputQueue，一个OutputQueue。InputQueue存储输入的数据，而通过Transform处理以后的数据则放在OutputQueue，OutputQueue就好像是一个BufferBlock。最终

我们可以通过Receive方法来阻塞的一个一个获取OutputQueue中的数据。TransformBlock的Completion.Wait()方法只有在OutputQueue中的数据为0的时候才会返回。



举个例子，我们有一组网址的URL，我们需要对每个URL下载它的HTML数据并存储。那我们通过如下的代码来完成：



```
public TransformBlock<string, string> tbUrl = new TransformBlock<string, string>((url)
=>
{
    WebClient webClient = new WebClient();
    return webClient.DownloadString(new Uri(url));
}

public void TestDownloadHTML()
{
    tbUrl.Post("www.baidu.com");
    tbUrl.Post("www.sina.com.cn");

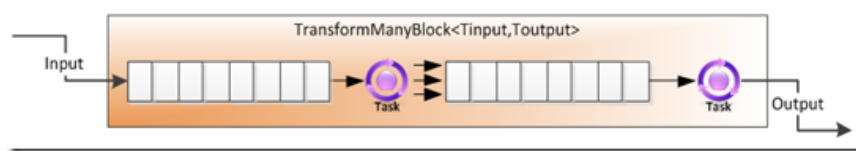
    string baiduHTML = tbUrl.Receive();
    string sinaHTML = tbUrl.Receive();
}
```



当然，Post操作和Receive操作可以在不同的线程中进行，Receive操作同样也是阻塞操作，在OutputQueue中有可用的数据时，才会返回。

TransformManyBlock

TransformManyBlock和TransformBlock非常类似，关键的不同点是，TransformBlock对应于一个输入数据只有一个输出数据，而TransformManyBlock可以有多个，及可以从InputQueue中取一个数据出来，然后放多个数据放入到OutputQueue中。



```
TransformManyBlock<int, int> tmb = new TransformManyBlock<int, int>((i) => { return new
int[] { i, i + 1 }; });

ActionBlock<int> ab = new ActionBlock<int>((i) => Console.WriteLine(i));

public void TestSync()
{
    tmb.LinkTo(ab);

    for (int i = 0; i < 4; i++)
    {
        tmb.Post(i);
    }
}
```

```

        Console.WriteLine("Finished post");
    }
}

```

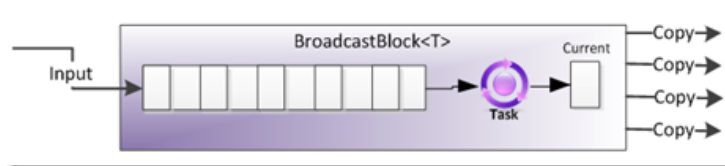
```

Finished post
0
1
1
2
2
3
3
4

```

BroadcastBlock

BroadcastBlock的作用不像BufferBlock，它是使命是让所有和它相联的目标Block都收到数据的副本，这点从它的命名上面就可以看出来。还有一点不同的是，BroadcastBlock并不保存数据，在每一个数据被发送到所有接收者以后，这条数据就会被后面最新的一条数据所覆盖。如没有目标Block和BroadcastBlock相连的话，数据将被丢弃。但BroadcastBlock总会保存最后一个数据，不管这个数据是不是被发出去过，如果有一个新的目标Block连上来，那么这个Block将收到这个最后一个数据。



```

BroadcastBlock<int> bb = new BroadcastBlock<int>((i) => { return i; });

ActionBlock<int> displayBlock = new ActionBlock<int>((i) =>
    Console.WriteLine("Displayed " + i));

ActionBlock<int> saveBlock = new ActionBlock<int>((i) =>
    Console.WriteLine("Saved " + i));

ActionBlock<int> sendBlock = new ActionBlock<int>((i) => Console.WriteLine("Sent
" + i));

public void TestSync()
{
    bb.LinkTo(displayBlock);
    bb.LinkTo(saveBlock);
    bb.LinkTo(sendBlock);

    for (int i = 0; i < 4; i++)
    {
        bb.Post(i);
    }

    Console.WriteLine("Post finished");
}

```

```
Post finished
Displayed
Displayed
Displayed
Displayed
Sent 0
Sent 1
Sent 2
Sent 3
Saved 0
Saved 1
Saved 2
Saved 3
```

如果我们在Post以后再添加连接Block的话，那些Block就只会收到最后一个数据了。



```
public void TestSync()
{
    for (int i = 0; i < 4; i++)
    {
        bb.Post(i);
    }

    Thread.Sleep(5000);

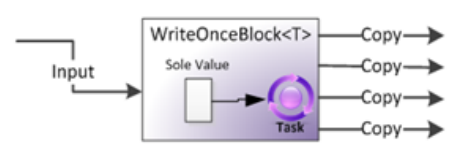
    bb.LinkTo(displayBlock);
    bb.LinkTo(saveBlock);
    bb.LinkTo(sendBlock);
    Console.WriteLine("Post finished");
}
```



```
Displayed
Saved 3
Post finished
Sent 3
```

WriteOnceBlock

如果说BufferBlock是最基本的Block，那么WriteOnceBlock则是最最简单的Block。它最多只能存储一个数据，一旦这个数据被发送出去以后，这个数据还是会留在Block中，但不会被删除或被新来的数据替换，同样所有的接收者都会收到这个数据的备份。



和BroadcastBlock同样的代码，但是结果不一样：



```
WriteOnceBlock<int> bb = new WriteOnceBlock<int>((i) => { return i; });

    ActionBlock<int> displayBlock = new ActionBlock<int>((i) =>
    Console.WriteLine("Displayed " + i));
```

```

        ActionBlock<int> saveBlock = new ActionBlock<int>((i) =>
        Console.WriteLine("Saved " + i));

        ActionBlock<int> sendBlock = new ActionBlock<int>((i) => Console.WriteLine("Sent
        " + i));

        public void TestSync()
        {
            bb.LinkTo(displayBlock);
            bb.LinkTo(saveBlock);
            bb.LinkTo(sendBlock);
            for (int i = 0; i < 4; i++)
            {
                bb.Post(i);
            }

            Console.WriteLine("Post finished");
        }
    }

```

```

Post finished
Displayed 0
Sent 0
Saved 0

```

WriteOnceBlock只会接收一次数据。而且始终保留那个数据。

同样使用Receive方法来获取数据也是一样的结果，获取到的都是第一个数据：

```

public void TestReceive()
{
    for (int i = 0; i < 4; i++)
    {
        bb.Post(i);
    }

    Console.WriteLine("Post finished");

    Console.WriteLine("1st Receive:" + bb.Receive());
    Console.WriteLine("2nd Receive:" + bb.Receive());
    Console.WriteLine("3rd Receive:" + bb.Receive());
}

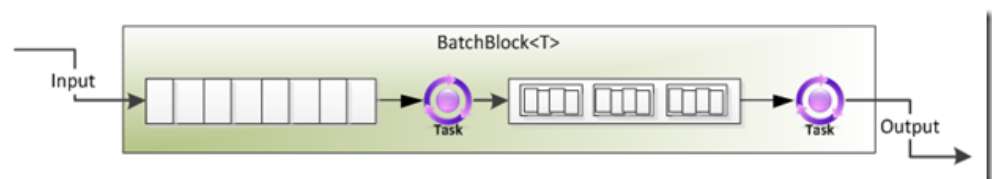
```

```

Post finished
1st Receive:0
2nd Receive:0
3rd Receive:0

```

BatchBlock



BatchBlock提供了能够把多个单个的数据组合起来处理的功能，如上图。应对有些需求需要固定多个数据才能处理的问题。在构造函数中需要制定多少个为一个Batch，一旦它收到了那个数量的数据后，会打包放在它的OutputQueue中。当BatchBlock被调用Complete告知Post数据结束的时候，会把InputQueue中余下的数据打包放入OutputQueue中等待处理，而不管InputQueue中的数据量是不是满足构造函数的数量。



```
BatchBlock<int> bb = new BatchBlock<int>(3);

ActionBlock<int[]> ab = new ActionBlock<int[]>((i) =>
{
    string s = string.Empty;

    foreach (int m in i)
    {
        s += m + " ";
    }
    Console.WriteLine(s);
});

public void TestSync()
{
    bb.LinkTo(ab);

    for (int i = 0; i < 10; i++)
    {
        bb.Post(i);
    }
    bb.Complete();

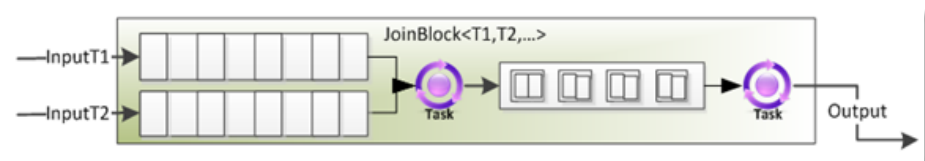
    Console.WriteLine("Finished post");
}
```



BatchBlock执行数据有两种模式：贪婪模式和非贪婪模式。贪婪模式是默认的。贪婪模式是指任何Post到BatchBlock，BatchBlock都接收，并等待个数满了以后处理。非贪婪模式是指BatchBlock需要等到构造函数中设置的BatchSize个数的Source都向BatchBlock发数据，Post数据的时候才会处理。不然都会留在Source的Queue中。也就是说BatchBlock可以使用在每次从N个Source那个收一个数据打包处理或从1个Source那里收N个数据打包处理。这里的Source是指其他的继承ISourceBlock的，用LinkTo连接到这个BatchBlock的Block。

在另一个构造参数中GroupingDataflowBlockOptions，可以通过设置Greedy属性来选择是否贪婪模式和MaxNumberOfGroups来设置最大产生Batch的数量，如果到达了这个数量，BatchBlock将不会再接收数据。

JoinBlock



JoinBlock一看名字就知道是需要和两个或两个以上的Source Block相连接的。它的作用就是等待一个数据组合，这个组合需要的数据都到达了，它才会处理数据，

并把这个组合作为一个Tuple传递给目标Block。举个例子，如果定义了JoinBlock<int, string>类型，那么JoinBlock内部会有两个ITargetBlock，一个接收int类型的数据，一个接收string类型的数据。那只有当两个ITargetBlock都收到各自的数据后，才会放到JoinBlock的OutputQueue中，输出。



```
JoinBlock<int, string> jb = new JoinBlock<int, string>();
ActionBlock<Tuple<int, string>> ab = new ActionBlock<Tuple<int, string>>((i) =>
{
    Console.WriteLine(i.Item1 + " " + i.Item2);
});

public void TestSync()
{
    jb.LinkTo(ab);

    for (int i = 0; i < 5; i++)
    {
        jb.Target1.Post(i);
    }

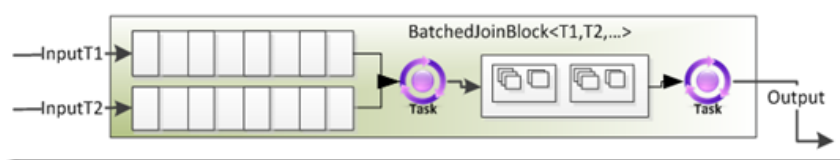
    for (int i = 5; i > 0; i--)
    {
        Thread.Sleep(1000);
        jb.Target2.Post(i.ToString());
    }

    Console.WriteLine("Finished post");
}
```




```
0 5
1 4
2 3
3 2
Finished post
4 1
```

BatchedJoinBlock



BatchedJoinBlock一看就是BatchBlock和JoinBlock的组合。JoinBlock是组合目标队列的一个数据，而BatchedJoinBlock是组合目标队列的N个数据，当然这个N可以在构造函数中配置。如果我们定义的是BatchedJoinBlock<int, string>，那么在最后的OutputQueue中存储的是Tuple<IList<int>, IList<string>>，也就是说最后得到的数据是Tuple<IList<int>, IList<string>>。它的行为是这样的，还是假设上文的定义，BatchedJoinBlock<int, string>，构造BatchSize输入为3。那么在这个BatchedJoinBlock种会有两个ITargetBlock，会接收Post的数据。那什么时候会生成一个Tuple<IList<int>, IList<string>>到

OutputQueue中呢，测试下来并不是我们想的需要有3个int数据和3个string数据，而是只要2个ITargetBlock中的数据个数加起来等于3就可以了。3和0,2和1，1和2或0和3的组合都会生成Tuple<IList<int>, IList<string>>到OutputQueue中。可以参看下面的例子：



```
BatchedJoinBlock<int, string> bjb = new BatchedJoinBlock<int, string>(3);

        ActionBlock<Tuple<IList<int>, IList<string>>> ab = new
ActionBlock<Tuple<IList<int>, IList<string>>>((i) =>
    {
        Console.WriteLine("-----");

        foreach (int m in i.Item1)
        {
            Console.WriteLine(m);
        };

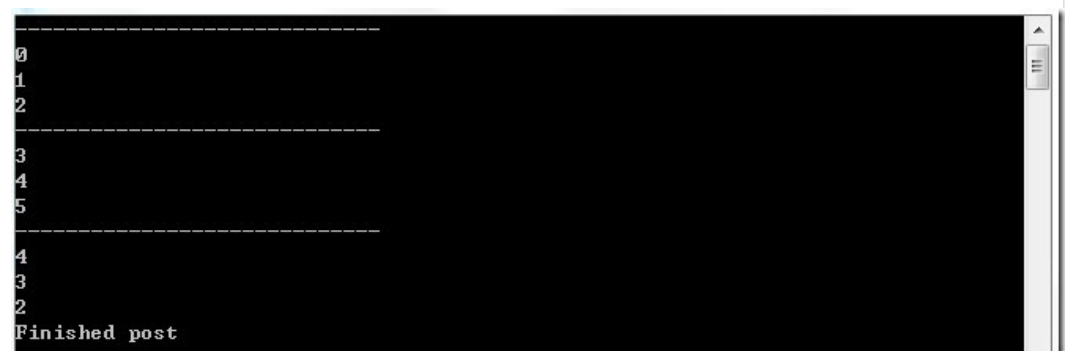

        foreach (string s in i.Item2)
        {
            Console.WriteLine(s);
        };
    });

public void TestSync()
{
    bjb.LinkTo(ab);

    for (int i = 0; i < 5; i++)
    {
        bjb.Target1.Post(i);
    }

    for (int i = 5; i > 0; i--)
    {
        bjb.Target2.Post(i.ToString());
    }

    Console.WriteLine("Finished post");
}
```



最后剩下的一个数据1，由于没有满3个，所以一直被保留在Target2中。

TDF中最有用的功能之一就是多个Block之间可以组合应用。ISourceBlock可以连接ITargetBlock，一对一，一对多，或多对多。下面的例子就是一个TransformBlock和一个ActionBlock的组合。TransformBlock用来把数据*2，并转换成字符串，然后把数据扔到ActionBlock中，而ActionBlock则用来最后的处理数据打印结果。



```
public ActionBlock<string> abSync = new ActionBlock<string>((i) =>
{
    Thread.Sleep(1000);
    Console.WriteLine(i + " ThreadId:" + Thread.CurrentThread.ManagedThreadId +
" Execute Time:" + DateTime.Now);
})

);

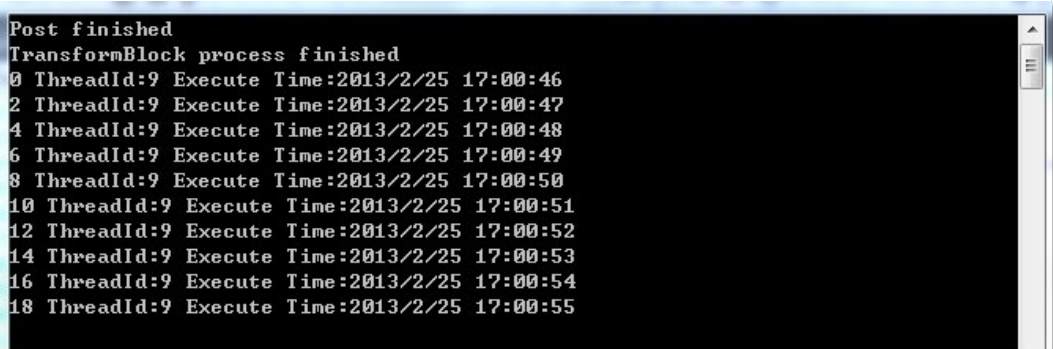
public TransformBlock<int, string> tbSync = new TransformBlock<int, string>((i)
=>
{
    i = i * 2;
    return i.ToString();
})

);

public void TestSync()
{
    tbSync.LinkTo(abSync);

    for (int i = 0; i < 10; i++)
    {
        tbSync.Post(i);
    }
    tbSync.Complete();
    Console.WriteLine("Post finished");

    tbSync.Completion.Wait();
    Console.WriteLine("TransformBlock process finished");
}
```



```
Post finished
TransformBlock process finished
0 ThreadId:9 Execute Time:2013/2/25 17:00:46
2 ThreadId:9 Execute Time:2013/2/25 17:00:47
4 ThreadId:9 Execute Time:2013/2/25 17:00:48
6 ThreadId:9 Execute Time:2013/2/25 17:00:49
8 ThreadId:9 Execute Time:2013/2/25 17:00:50
10 ThreadId:9 Execute Time:2013/2/25 17:00:51
12 ThreadId:9 Execute Time:2013/2/25 17:00:52
14 ThreadId:9 Execute Time:2013/2/25 17:00:53
16 ThreadId:9 Execute Time:2013/2/25 17:00:54
18 ThreadId:9 Execute Time:2013/2/25 17:00:55
```