올해 3월 닷넷 정식 버전이 발표되면서 C#이 새로운 언어로 떠오르고 있는데, 특히 네트워크 부분에서 기존의 IOCP(IO Completion Port) 기능을 손쉽게 사용할 수 있도록 만들어 놓았다는 점에서 주목할만 하다.   
  
기존에는 이 기능을 이용하려면 Win sock 2 API를 직접 호출해야 했지만, C#에서는 이 기능이 BCL (Base Class Library) 안에 포함되어 있어 손쉽게 사용할 수 있다. C#에서는 기본적으로 비동기 통신을 하면 자동으로 IOCP를 이용한다. 이는 C# 뿐만 아니라 닷넷의 기본 기능인 것이다. 또한 플래시는 이번에 MX 버전이 출시되면서 많은 기능의 개선이 있었다. 플래시 5부터 XML 소켓을 지원해 지속적으로 연결된 상태에서 네트워크 통신이 가능해졌으며 온라인 게임으로까지 영역을 넓힐 수 있게 됐다.   
  
앞으로 총 4회의 연재를 통하여 온라인 게임 서버로서의 C#의 가능성을 알아보고, 게임 클라이언트로서 플래시의 가능성에 대해 알아볼 것이다. 기존 온라인 게임의 경우 프로그램을 다운받아 플레이해야 했으나 플래시로 온라인 게임을 만들 경우, 스트리밍 방식을 이용하여 별도의 다운로드없이 실시간으로 데이터를 주고받음으로써 즉시 플레이가 가능하다. 초보자도 해당 홈페이지에 접속하기만 하면 바로 플레이할 수 있기 때문에 누구나 쉽게 게임을 시작할 수 있다.   
  
필자는 이러한 플래시와 C#의 특징에 주목하여 그 가능성을 테스트한다는 의미에서 포트리스와 비슷한 게임인 ‘심플 포트리스(Simple Fortress)’를 만들어 보았다. 별도의 다운로드 없이 URL 주소만 입력하면 플레이할 수 있으며, 웹 브라우저 내에서 실행되므로 게임을 하면서도 다른 작업창을 실행할 수 있다는 이점이 있다. 본격적인 설명에 들어가기 전에, 이 게임은 필자가 닷넷과 플래시에 대한 테스트용으로 만든 것으로 상업적으로 사용할 의도가 없으며, 이 게임의 거의 모든 이미지와 사운드 파일은 [포트리스 2 공식 홈페이지](http://fortress2.x2game.com/)에서 다운받아 사용한 것임을 미리 밝혀둔다.   
  
**심플 포트리스 미리보기**   
앞으로 우리가 만들 게임이 어떤 게임인지 한 번 보도록 하자. ‘이달의 디스켓’의 압축을 풀면 Server와 Client 두 개의 폴더가 있을 것이다. 이중 Server 폴더에서 FortressServer.exe를 실행하면 서버가 작동한다(이 서버 프로그램은 닷넷 기반 하에서만 작동하기 때문에 최소한 닷넷 프레임워크는 설치되어 있어야 한다). 그 다음 Client 폴더의 fortress.html 파일을 실행시킨다. <화면 1>과 <화면 2>는 서버와 클라이언트의 작동 화면이다.

|  |  |  |
| --- | --- | --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h1.jpg |  | http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h2.jpg |
| <화면 1> FortessServer.exe 실행화면 |  | <화면 2> Fortess.html 실행화면 |

웹 브라우저 화면에서 원하는 ID를 입력하고 들어간가면 <화면 3>과 같은 대기실 화면이 나온다. 이 곳에서 탱크 종류와 팀을 선택할 수 있다. 이 상태에서 또 한 번 Fortress.html 파일을 실행해서 새로운 ID를 입력하고 들어오면 두 명의 게이머가 대기실에 들어온 상태가 된다.

|  |  |  |
| --- | --- | --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h3.jpg |  | http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h4.jpg |
| <화면 3> 대기실 화면 |  | <화면 4> 게임 시작 화면 |

이때 두 개의 웹 브라우저에서 동시에 배경음악이 나오므로 약간 혼란스러울 수도 있다. 서로 다른 팀을 고른 후, 처음에 들어왔던 사람이 START 버튼을 누르면 게임이 시작된다(<화면 4>). 게임 방법은 <표 1>과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | http://images.zdnet.co.kr/img/edit_table/table8_1.gif | <표 1> 게임방법 | http://images.zdnet.co.kr/img/edit_table/table8_2.gif |  | http://images.zdnet.co.kr/img/edit_table/table8_7.gif | |
| |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  | | --- | --- | | **기능** | **키** | | 탱크의 이동 | 화살표 좌우 키 | | 탱크의 각도 조정 | 화살표 상하 키 | | 대포 발사 | 스페이스 바를 눌러서 파워를 조절 후 발사한다. | | |  | |
| |  |  |  | | --- | --- | --- | | http://images.zdnet.co.kr/img/edit_table/table8_bot_left.gif |  | http://images.zdnet.co.kr/img/edit_table/table8_bot_right.gif | |

먼저 자신의 차례가 되면 자신의 탱크 위에 ‘READY!’라는 글자가 깜빡거린다. 그 상태에서 각도 조정이나 이동을 하면서 조절한 후 스페이스 바를 길게 눌렀다가 떼면 대포가 발사된다(<화면 5>). 서로 번갈아 가면서 대포를 발사하는 방식으로 게임을 쉽게 하기 위하여 폭발의 파편만 닿아도 생명치를 줄게 해 놓았다. 채팅도 지원하므로 대화를 입력해도 된다(<화면 6>). 이제 어떤 게임인지 살펴봤으니 본격적으로 게임 제작에 착수해 보자. 먼저 서버부터 만들어 볼 것이다.

|  |  |  |
| --- | --- | --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h5.jpg |  | http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h6.jpg |
| <화면 5> 대포를 발사한 장면 |  | <화면 6> 대화를 나누는 장면 |

**비동기 호출의 기본은 델리게이트**   
서버 제작에서 중요한 점은 다수의 사용자를 처리해야 하는 데 있다. 한 명이 아니라 여러 명이 동시에 접속하므로 그들의 요구를 동시에 처리해 줘야만 한다. 그런데 일반적으로 프로그래밍하다 보면 한 명을 처리하기 위해 그 대답을 기다리다가 다른 사람의 요구를 못 들어주게 된다.   
  
즉 블럭이 돼 버려서 다수의 사용자를 처리할 수 없게 된다. 이 때의 해결책이 바로 쓰레드이다. 닷넷에서는 이러한 쓰레드를 이용하여 비동기 호출을 지원하는데, 이를 이용하여 많은 사용자들의 요구를 처리할 수 있다. 이는 함수를 호출할 때 동기적으로 그 함수의 호출이 끝날 때까지 기다리는 것이 아니라 함수를 호출하면 그 함수는 새로운 쓰레드 안에서 돌아가고 호출자 또한 기존 쓰레드 내에서 돌아가므로 동시에 일을 처리할 수 있게 된다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-1.jpg |
| <그림 1> 동기 호출과 비동기 호출 |

<그림 1>은 소켓의 Accept문을 예로 들어 동기 호출과 비동기 호출의 차이점을 설명한 그림이다. 이 비동기 호출의 핵심 개념에는 바로 델리게이트(delegate)라는 것이 자리잡고 있다. 그러므로 델리게이트의 본질부터 파악하는 것이 비동기 호출의 원리를 이해하는 방법일 것이다.   
  
**C# 세계의 브로커, 델리게이트**   
C#을 배우는 사람들에게 있어 델리게이트는 생소한 개념이 아닐 수 없다. 델리게이트는 C 언어의 함수 포인터에서부터 유래되었다. C 언어에서는 함수 포인터를 잘 안 썼으므로 생소할 수도 있다. 먼저 델리게이트의 사전적인 의미를 살펴보면 ‘대리자’ 또는 ‘위임형’ 등으로 정의하고 있다. 델리게이트라는 것이 어떤 함수를 대신해서 호출되기 때문에 그렇게 이름을 붙인 듯하다. 이해를 돕기 위해 다음과 같은 기상청 시나리오를 살펴보자.   
  
기상청에서는 기상정보를 수집해 그 정보를 필요로 하는 곳에 전달한다. 그런데 누가 언제 그러한 정보를 필요로 할지 미리 알수 없기 때문에 그런 보고 시스템을 미리 구축해 놓을 수 없었다. 그래서 대신 기상 정보 브로커를 고용해 그에게 정보를 주면, 그가 자신에게 연결된 기상 정보를 필요로 하는 사람들에게 그 정보를 주기로 했다. 그렇게 해서 보고 시스템을 완성하게 됐다. 이에 신문사가 제일 처음 그 정보를 달라고 브로커에게 요청을 했다. 브로커는 그 요청을 받아들이고 기상청으로부터 기상 정보를 받는대로 신문사에게 전해 주기로 했다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-2.jpg |
| <그림 2> 기상 브로커 시나리오 |

이를 그림으로 나타내면 <그림 2>와 같다. 그러면 이를 코딩으로 나타내 보자. 미래의 일은 예측할 수 없기 때문에 이미 모든 계획은 다 세웠지만 누가 그 계획에 참여할지는 모를 때가 있다. 프로그래밍의 세계에서도 누가 그 일에 참여하게 될지 모르는 상황이 종종 생긴다. 이럴 때 델리게이트를 사용하는 것이다. 차후에 델리게이트를 통해서 그 일을 할 메쏘드만 연결시켜주면 된다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 1> 기상 보고 시스템 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | namespace MeteorologicalSystem  {         // 기상 정보 브로커      public delegate void Information(int temparature, int humidity,          string nephanalysis);         // 기상청      class MeteorologicalOffice      {          // 기상 예보          public static void Report(Information broker)          {              broker(25,60,”구름 없음”);          }      }         // 신문사      class NewspaperCompany      {          // 신문사에서 신문을 발간          public static void Publish(int temparature, int humidity,              string nephanalysis)          {              Console.WriteLine(“[신문사 출판] 온도:{0}, 습도:{1}, 구름분포:{2}”,                  temparature, humidity, nephanalysis);          }          }          // <summary>          // Class1에 대한 요약 설명          // </summary>          class Class1          {              // 해당 응용 프로그램의 주 진입점              [STAThread]              static void Main(string[] args)              {                  // TODO: 여기에 응용 프로그램을 시작하는 코드를 추가                  // 첫번째 예제                  Console.WriteLine(“=== 첫번째 예제 ===”);                   Information broker = new Information                   (NewspaperCompany.Publish);                  // 신문사 등록                  MeteorologicalOffice.Report(broker); // 기상정보 보고 시스템 가동              }          }      } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

**델리게이트의 비밀을 파헤치자**   
<리스트 1>을 보면 한 가지 궁금증이 떠오를 수도 있다. 마지막 줄을 다시 보자.

Information broker =   
    new Information(NewspaperCompany.Publish);

여기서 왜 new라는 키워드를 썼는지 궁금증이 일어날 것이다. new라는 것은 새로운 Object를 할당할 때에만 쓰는 키워드인데, 여기서 사용했다는 것은 마치 클래스를 할당하는 것과 비슷하다고 생각할 수 있다. 만약 그렇게 생각했다면 맞다. 델리게이트라는 것이 바로 클래스이기 때문이다. 델리게이트를 ‘위임[형]’이라고 번역하듯이 델리게이트는 클래스 타입이다. 그러면 클래스 바디는 어디에 있는 걸까? 클래스라면 다음과 같이 되어 있어야 한다.

Class Information   
{   
      
}

하지만 클래스의 정의가 다음과 같이 한 줄로 되어 있다.

delegate void Information(int temparature, int humidity, string nephanalysis);

도대체 바디는 어디에 있는가? 사실 이 한 줄에는 바디를 포함하고 있다(<그림 3>). 즉 리턴형과 인자형에 대한 정보가 클래스 바디가 되는 것이다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-3.jpg |
| <그림 3> DeleBang의 클래스 바디 |

델리게이트가 정말 클래스인지 확인해 보기 위하여 중간 코드로 확인해 보자. 닷넷을 설치한 폴더에 ILDA SM.exe 파일이 있다. 이는 IL Disassembler의 약자로 말 그대로 중간 코드를 disassemble해 준다. 이를 통해 앞에서 컴파일한 Meteorological System.exe 파일을 열어 보면 <화면 7>이 나타난다. <화면 7>을 보면 글자 옆에 아이콘들이 있는데, 이들이 무엇을 의미하는지는 <화면 8>을 보면 알 수 있다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h7.jpg |
| <화면 7> MeteorologicalSystem.exe를 Disassemble한 화면 |

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h8.jpg |
| <화면 8> 아이콘 도움말 |

이를 통해서 보면 Information은 클래스라는 것과 .ctor, BeginInvoke, EndInvoke, Invoke라는 네 개의 메쏘드를 가지고 있음을 알 수 있다. 또한 Information은 System.Muticastdele gate에서 상속받았다는 정보까지 갖고 있다. 여기에 나오는 4개의 메쏘드중 .ctor은 생성자를 의미한다. BeginInvoke와 EndInvoke는 비동기 호출에 쓰이며, Invoke는 동기 호출에 쓰이는 메쏘드이다. 이들에 대한 코드는 <리스트 2>와 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 2> Information 클래스의 코드 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | public class Information : System.Multicastdelegate  {      // 생성자      public Information (object target, int32 methodPtr);         public void virtual Invoke( int temparature, int humidity,          string nephanalysis );      public virtual IAsyncResult BeginInvoke( int temparature,          int humidity, string nephanalysis,          AsyncCallback callback, Object object);         public virtual void EndInvoke( IAsyncResult result);  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | http://images.zdnet.co.kr/img/edit_table/table8_1.gif | <표 2> 델리게이트의 Private 필드 | http://images.zdnet.co.kr/img/edit_table/table8_2.gif |  | http://images.zdnet.co.kr/img/edit_table/table8_7.gif | |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  | | --- | --- | --- | | **필드** | **타입** | **설 명** | | \_target | System.Object | 인스턴스 메쏘드에 쓰이는 것으로, 콜백메쏘드가 호출될때 참조하는 Object이다. | | \_methodPtr | System.Int32 | CLR에서 사용되는데, 콜백될 메쏘드를 가리키는 integer 값 | | \_prev | System.Multicastdelegate | 다른 델리게이트를 가리키는 값 | | |  | |
| |  |  |  | | --- | --- | --- | | http://images.zdnet.co.kr/img/edit_table/table8_bot_left.gif |  | http://images.zdnet.co.kr/img/edit_table/table8_bot_right.gif | |

<리스트 2>를 보면 한 가지 이상한 점을 발견할 수 있을 것이다. 바로 Information의 생성자인데, 우리는 <리스트 1>에서 생성자로 NespaperCompany.Publish를 넘겨줬다. 그런데 <리스트 2>를 보면 생성자는 두 개의 인자가 필요하다. 분명 에러를 발생해야 하는데 잘 되는 것을 보면 이상이 없는 것이다. 여기서 컴파일러는 원본 소스를 컴파일할 때, 앞의 생성자에 맞도록 파싱을 해주기 때문에 에러가 안 나는 것이다.   
  
앞의 두 인자중 target은 메쏘드가 있는 오브젝트를 가리키는데 만약 메쏘드가 static이면 null 값을 넘겨준다. methodPtr은 callback 메쏘드를 가리키는 CLR 내부에서 쓰이는 레퍼런스 값이다. 이들 생성자에서 받은 두 개의 값을 Information 클래스는 따로 Private 필드에 저장해 두는데 그 Private 필드는 <표 2>와 같다.   
  
이중 \_prev 값은 나중에 Muticatedelegate에서 설명할 것이다. 그럼 이제 우리가 생성자에게 넘겨준 그 값들을 직접 눈으로 확인해 보자. <리스트 1>에서 main 메쏘드에 다음과 같은 코드를 추가하자.

// 생성자에 넘겨준 값을 보자.   
if ( broker.Target == null )   
{   
   Console.WriteLine(“null”);   
}   
else   
{   
   Console.WriteLine(broker.Target);   
}   
Console.WriteLine(broker.Method);

이를 실행하면 다음과 같은 결과를 볼 수 있다.

=== 첫번째 예제 ===   
[신문사 출판] 온도 : 25, 습도 : 60, 구름분포 : 구름 없음   
=== 두번째 예제 ===   
null   
Void Publish(Int32, Int32, System.String)

우리가 넘겨준 메쏘드가 Static이기 때문에 Target에는 null 값이 들어갔고, 메쏘드에는 대리자에 등록된 메쏘드의 형식이 나왔다. 만약 여기에서 instance 메쏘드를 넘겨주면 어떤 값이 나올까? 앞에서 Kill 메쏘드에서 static을 빼고, NewspaperCompany 클래스를 새로 생성해서 실행해 보면 다음과 같은 결과가 나온다. 즉 메쏘드의 Object를 넘겨주는 것이다.

=== 첫번째 예제 ===   
[신문사 출판] 온도 : 25, 습도 : 60, 구름분포 : 구름 없음   
=== 두번째 예제 ===   
MeteorologicalSystem.NewspaperCompany   
Void Publish(Int32, Int32, System.String)

이제 생성자에 대한 비밀은 풀었으나 아직 Information 클래스의 메쏘드에 대한 비밀이 남아 있다. <리스트 2>에서 Invoke 메쏘드가 있는데 이것이 실제 실행하는 메쏘드이다. 그런데 우리는 그 메쏘드를 호출한 적이 없다. 그러면 컴파일러가 알아서 호출해 주는 것일까? 그렇게 생각했다면 정답이다. 우리는 <리스트 1>에서 다음과 같이 호출했다.

broker(25,60,”구름 없음”);

컴파일러는 이 코드를 보고 다음과 같이 번역한다.

broker.Invoke(25,60,”구름 없음”);

그런데 정말 이렇게 번역하는 것일까? 이것 역시 ILDASM을 이용해서 확인해 보자. <화면 9>를 보면 컴파일러가 만들어 준 Invoke 메쏘드를 볼 수 있을 것이다.

|  |  |  |
| --- | --- | --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h9.jpg |  | http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h9.jpg |
| <화면 9> Invoke가 호출된 부분 |  | <화면 10> += 연산자가 나타내는 메쏘드 |

**너에게 임무를 추가한다!**   
이제 델리게이트에 대해 어느 정도 비밀을 풀었다. 그런데 여기서 한 가지 의문이 남아 있다. <표 2>를 보면 \_prev라는 필드가 있는데 이 필드의 용도가 무엇이냐 하는 것이다. 이를 위해 다음과 같은 시나리오를 보자.   
  
어느 날 방송사에서도 그 기상 정보를 달라는 요청이 들어왔다. 이미 기상청에서는 브로커에게 그 일을 일임했으므로 방송사는 브로커와 거래를 해 등록함으로써 브로커를 통해 기상청의 정보를 제공받게 된다. 이를 코드로 나타내면 다음과 같다. 먼저 Broadcasting Company 클래스를 다음과 같이 새로 만든다.

// 방송사   
class BroadcastingCompany   
{   
   // 방송사에서 방송 보도   
   public static void Broadcast( int temparature,   
      int humidity, string nephanalysis)   
   {   
      Console.WriteLine(“[방송 보도] 온도:{0}, 습도:{1}, 구름분포:{2}”,   
      temparature, humidity, nephanalysis);   
   }   
}

그 다음 브로커에 다음과 같이 추가하면 된다.

broker += new Information(BroadcastingCompany.Broadcast);

결과는 다음과 같다.

=== 세번째 예제 ===   
[신문사 출판] 온도 : 25, 습도 : 60, 구름분포 : 구름 없음   
[방송 보도] 온도 : 25, 습도 : 60, 구름분포 : 구름 없음

여기서는 단순히 += 연산자를 이용했다. C#에서는 연산자 오버로딩을 지원하기 때문에 += 연산자는 실질적으로 메쏘드를 호출하는 것이다. 그러면 그 메쏘드가 무엇인지 ILDASM을 통해 확인해 보자. <화면 10>을 보면 Combine이라는 메쏘드가 호출됨을 볼 수 있다.   
  
기본적으로 델리게이트 타입은 Muticastdelegate를 상속받으므로 하나의 callback 메쏘드가 아닌 다수의 callback 메쏘드를 가질 수 있다. 앞의 시나리오에서 보듯이 어떤 사건에 의해 다수가 그 영향을 받는 경우가 있기 때문에 이러한 기능을 지원하는 것이다. 델리게이트 내부적으로는 이것을 linked-list로 유지를 한다. linked-list로 유지하기 때문에 앞의 링크를 가리키는 \_prev 필드가 필요한 것이다. 이를 그림으로 나타내면 <그림 4>와 같다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-4.jpg |
| <그림 4> 브로커의 linked-list |

이 그림을 보면 한 가지 의문점이 들 것이다. 우선 브로커가 처음에 등록했던 신문사를 가리키는 것이 아니고 나중에 등록한 방송사를 가리킨다는 것과 일반적인 linked-list에서는 \_next 필드로 다음 오브젝트를 가리키는 데 비해 여기는 \_prev필드를 써서 앞의 오브젝트를 가리킨다는 것이다. 그 이유는 리턴 값 때문에 그렇다. 만약 callback 메쏘드에 리턴 값이 있다면 어떻게 할 것인가? 예를 들어 다음과 같은 경우이다.

delegate int Information(int temparature, int humidity,   
   string nephanalysis);

이와 같이 리턴 값이 있는 경우 브로커에 등록된 callback 메쏘드가 한 개라면 별 문제가 없지만, 여러 개라면 어떻게 할까? 먼저 C#에서는 여러 개의 callback 메쏘드중 단 한개의 리턴 값만 받아 올 수 있다. 그 여러 개의 리턴 값을 다 받아오려면 다른 방법을 취해야 하는데 그 방법은 나중에 소개할 것이다. 일단 일반적인 상황에서는 리턴 값을 하나만 취한다.   
  
그러면 어떤 리턴 값을 취할 것인가? 상식적으로 생각해 보면 가장 최근에 호출된 callback 메쏘드의 리턴 값이 가장 가치있다고 생각될 것이다. 그래서 가장 나중에 호출된 리턴 값이 필요하기 때문에 \_next 필드를 안 쓰고 위로 거슬러 올라가서 호출하는 것이다. \_next를 쓴 경우와 \_prev를 쓴 경우를 그림으로 비교해 보자. 예를 들어 f1, f2, f3를 브로커에 등록했다고 하면 <그림 5>와 같은 호출 과정을 볼 수 있다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-5.jpg |
| <그림 5> \_prev와 \_next의 차이점 |

그러면 직접 호출을 담당하는 Invoke 메쏘드에 대한 가상 코드를 만들어 보자.

class Information : Multicastdelegate   
{   
   public int virtual Invoke(int temparature, int humidity,   
      string nephanalysis)   
   {   
      // 앞으로 거슬러 올라간다.   
      if ( \_prev != null ) \_prev.Invoke(temparature,   
      humidity,   
      nephanalysis);   
  
      // 결국 맨 나중에 호출된 callback 메쏘드의 리턴 값이 리턴된다.   
      return \_target.methodPtr ( temparature, humidity,   
      nephanalysis);   
   }   
}

그럼 이제 \_prev를 살펴봤으니, Combine 메쏘드가 내부적으로 어떻게 이들 연결을 만드는지 보자. <리스트 3>은 델리게이트 클래스의 Combine 메쏘드들이다. Combine 메쏘드는 두 개의 델리게이트를 인자로 받는데, 먼저 delegate는 한 번 생성되면 immutable하기 때문에 \_prev 필드를 마음대로 변경할 수가 없다. 그래서 Combine을 할 때에는 second와 같은 새로운 델리게이트 오브젝트를 생성하고, 이때 \_prev 필드 값을 first로 설정해 준다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 3> 델리게이트의 메쏘드 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class System.delegate  {      // first와 second를 연결한 후에 second를 리턴한다.      public static delegate Combine(delegate first, delegate second);            // 배열에 의한 델리게이트를 연결시켜 준다.      public static delegate Combine(delegate[] delegateArray);         // 델리게이트를 chain에서 제거      public static Remove(delegate source, delegate value);  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

그렇다면 이를 이용해서 Combine 메쏘드가 정말 새로운 오브젝트를 생성해서 리턴하는지 다음과 같은 코드를 보자. 다음 결과를 보면 False가 나온다. Combine 메쏘드가 새로운 델리게이트를 새로 생성해서 리턴하기 때문이다.

Information broker1 =   
   new Information(NewspaperCompany.Publish);   
Information broker2 = (Information) delegate.Combine(broker1,   
   broker1);   
  
Console.WriteLine( (object) broker1 == (object) broker2 );

델리게이트에 Combine시키는 방법이 있으니 Remove시키는 방법도 있을 것이다. 다음과 같은 경우를 보자.

broker -= new Information(NewspaperCompany.Publish);   
MeteorologicalOffice.Report(broker);

앞에서 브로커에게 신문사와 방송사를 다 등록시켰는데 이번에는 신문사를 제거해 봤다. 이 -= 연산자 또한 실제로는 Remove 메쏘드를 나타낸다. 이는 <리스트 3>에서 보듯이 두 개의 인자를 취하는데, 첫 번째는 linked-list를 이루고 있는 델리게이트의 헤드를 가리키며, 두 번째는 삭제할 델리게이트를 가리킨다. 그런데 왜 지우는데 오브젝트를 새로 생성할까? linked-list에서 원하는 것을 찾아서 지워야 하는데, 이를 비교하는 방법에 그 원인이 있다.   
  
우리가 정확히 찾고자 하는 것은 NewspaperCompany. Publish로 이를 비교해야만 하는 것이다. 그런데 앞서서 델리게이트 클래스에서 생성자로 넘겨주는 것으로 \_target과 \_method Ptr이 있었다. 즉 instance/static이냐 하는 것과, 리턴 값과 인자형에 따라 클래스를 구분할 수 있는 것이다. 그렇기 때문에 델리게이트에서는 동등 비교를 하는 데 있어 \_target과 \_methodPtr을 이용한다. 다을 실행하면 TRUE를 리턴하는 것을 볼 수 있다.

Information broker3 =   
   new Information(NewspaperCompany.Publish);   
Information broker4 =   
   new Information(NewspaperCompany.Publish);   
  
Console.WriteLine(broker3.Equals(broker4));

이제 델리게이트에 다른 델리게이트를 쉽게 추가/삭제할 수 있게 됐다. 그러나 앞서 얘기했듯이 델리게이트의 linked-list 호출 구조는 한 가지 단점을 지니고 있다. 중간의 리턴 값들을 무시한다는 것이다. 게다가 만에 하나 리스트들 중에서 exception이 일어나든가 블러킹(blocking)되기라도 하면, 뒤에 딸려 있는 리스트들은 모두 호출되지 못하고 멈춰버린다.   
  
이럴 때에는 알아서 호출하게 놔두지 말고 사용자가 직접 하나하나 체크해 가면서 호출하면 된다. C#에서는 이와 같은 문제를 해결하기 위해 GetInvoca tionList()라는 함수를 제공하고 있다. 이를 이용하면 linked-list의 각 구성원을 똑같이 복사한 배열로 리턴받을 수 있다. 단 이때 \_prev 필드는 필요없기 때문에 null로 셋팅이 된다.

delegate[] arraydelegates = broker.GetInvocationList();   
foreach(Information agent in arraydelegates)   
{   
   Console.WriteLine(agent.Method);   
}

**이벤트 핸들러로 임명합니다~**   
이제 마지막으로 이벤트에 대해 알아 보자. 이벤트는 한 오브젝트에서 어떤 일이 일어나서 그 일을 다른 오브젝트에게 알려줄 때 이용한다. 이는 델리게이트와도 많이 유사한데, 실제로도 델리게이트를 이용하므로 이벤트는 델리게이트의 특별한 용도라고 생각하면 된다. 예를 들어 앞서 정의한 기상정보 시스템을 이벤트로 만들어 보자. 이때 이벤트라는 의미에 좀더 충실하기 위해 기상 특보를 기상청에서 발령한다고 가상해 보았다. 이때 기상청에서는 자신의 이벤트에 등록된 신문사에게 통지해 준다. <리스트 4>를 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 4> 이벤트 예제 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | namespace MeteorologicalSystem2  {      // 기상청      class MeteorologicalOffice      {      // 이벤트 인자 정의      public class SpecialReportEventArgs : EventArgs      {          public SpecialReportEventArgs(string nephanalysis)          {              this.nephanalysis = nephanalysis;          }             // 이벤트 인자 내에서 쓸 목록          public readonly string nephanalysis;      }         // 위임형 선언      public delegate void SpecialReportEventHandler( object sender ,          SpecialReportEventArgs args);             // 이벤트 정의          public event SpecialReportEventHandler SpecialReport;             // 이벤트를 발생시키는 함수          protected virtual void OnSpecialReport(SpecialReportEventArgs e)          {              if ( SpecialReport != null )              {                  SpecialReport(this,e);          }      }             // 이벤트 발생을 위해 테스트용으로 만든 함수          public void SimulateEvent(string nephanalysis)          {              SpecialReportEventArgs e = new SpecialReportEventArgs              (nephanalysis);              OnSpecialReport(e);          }      }         // 신문사      class NewspaperCompany      {          public NewspaperCompany( MeteorologicalOffice mm)          {              mm.SpecialReport += new MeteorologicalOffice.                  SpecialReportEventHandler(Publish);          }             // 신문사에서 신문을 발간          public static void Publish( object sender , MeteorologicalOffice.              SpecialReportEventArgs e)          {              Console.WriteLine(“[신문사 특보] 구름분포:{0}”, e.nephanalysis);          }      }         // Class2에 대한 요약 설명      class Class2      {          // 해당 응용 프로그램의 주 진입점          [STAThread]          static void Main(string[] args)          {              // TODO: 여기에 응용 프로그램을 시작하는 코드를 추가              MeteorologicalOffice office = new MeteorologicalOffice();              NewspaperCompany company = new NewspaperCompany(office);              office.SimulateEvent(“태풍 북상”); // 이벤트를 넣어줌          }      }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

이벤트 이용에는 몇 가지 관례가 있다. 먼저 일반적인 델리게이트에서는 인자에 제한이 없지만 이벤트에서는 두 개의 인자를 사용한다. 그 두 개는 보내는 이가 누구인지 나타내는 object형과 System.EventArgs에서 상속받은 클래스를 인자로 받는 것이 있다. 먼저 받는 사람이 여러 사람의 이벤트에 등록해 두면 누가 보냈는지 알 수 없으므로 누가 보냈는지 알기 위해 첫 번째 인자로 object형 인자를 받는다.   
  
예를 들면 신문사는 정보를 기상청으로부터 들을 수도 있지만 소식이 들어오는 경로는 여러 군데일 것이다. 누가 그 정보를 보냈는지 알아야 할 때가 있기 때문에 이런 방법을 사용하는 것이다. 두 번째 인자는 EventArgs를 상속받은 클래스인데 이를 사용면 좀더 깔끔하게 인자 관리를 할 수 있다(그러나 여기서는 편의를 위해서 단순하게 하나의 인자만 썼다).   
  
이벤트에서는 이름을 정하는 데 있어서도 몇 가지 관례가 있다. 먼저 System.EventArgs를 상속받는 클래스는 그 이름 끝에 EventArgs를 붙여준다. 또한 델리게이트를 선언시에도 이름 뒤에 EventHandler를 붙여준다. 마지막으로 이벤트를 발생시키는 메쏘드는 이름 앞에 On을 붙여준다. <리스트 4>를 보면 그냥 이벤트 키워드없이 관례만 따라주면 이벤트가 되지 않느냐고 물을 수 있는데 사실 이벤트는 내부적으로 또 다른 일을 하고 있다. 무슨 일을 내부적으로 꾸미는지 알기 위해 ILDASM으로 확인해 보자.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h11.jpg |
| <화면 11> 이벤트가 들어간 클래스 |

<화면 11>을 보면 생성하지 않았던 두 개의 함수가 추가되어 있는 것을 볼 수 있다. 게다가 우리는 SpecialReport를 public으로 선언을 했는데 private으로 되어 있다. 이를 가상 코드로 나타내 보자.

private SpecialReportEventHandler SpecialReport = null;   
  
// 이벤트 등록 메쏘드   
[MethodImpAttribute(MethodImplOption.Synchronized)]   
public void add\_SpecialReport(SpecialReportEventHandler handler)   
{   
   SpecialReport = (SpecialReportEventHandler)   
      delegate.Combine   
      (SpecialReport, handler);   
}   
  
// 이벤트 등록해제 메쏘드   
[MethodImpAttribute(MethodImplOption.Synchronized)]   
public void remove\_SpecialReport(SpecialReportEventHandler handler)   
{   
   SpecialReport = (SpecialReportEventHandler)   
      delegate.Remove   
      (SpecialReport, handler);   
}

먼저 우리가 public으로 선언한 SpecialReport가 private으로 되어 있으면서 null로 초기화되어 있다. 이는 이벤트라는 것을 외부에서 함부로 접근하지 못하게 막기 위함이다. 예를 들어 태풍이 오는 그런 급박한 상황에서만 이벤트가 발생해야 하는데 이를 public으로 둘 경우 외부에서 마음대로 통제하거나, 바람이 불어도 기상 특보를 발령하는 우를 범할 수가 있기 때문이다. 그래서 이벤트는 그 이벤트를 소유한 클래스 내에서만 발생시킬 수 있게 하기 위해 private 필드로 두는 것이다.   
  
또한 add\_＊와 remove\_\*라는 두개의 메쏘드가 추가됐는데, 이는 메쏘드명에서 알 수 있듯이 이벤트에 등록자(listener)들을 등록/해제하는 역할을 한다. +=와 -=연산자를 쓰면 add\_\*나 remove\_\*로 컴파일러가 바꿔 준다. Combine이나 Remove를 쓴 경우와 다르지 않게 보일 수 있으나, 자세히 보면 메쏘드 위에 애트리뷰트가 있다.   
  
이는 메쏘드를 동기화시켜서 쓰레드에 대한 안정성을 보장해 준다. 예를 들면 두 개의 리스너가 동시에 이벤트에 등록/해제해도 linked-list가 깨지지 않고 올바로 유지된다(<화면 12>). 결론적으로 이벤트는 결국 델리게이트의 특별한 케이스인데 보다 보안과 안정성에 중점을 둔 케이스라고 할 수 있다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129782-h12.jpg |
| <화면 12> 동기화된 메쏘드 |

**다음 글에서는 비동기 프로그래밍을**   
이번 글에서는 서버 구축하는 데 있어 필수인 비동기 소켓 프로그래밍에 앞서, 비동기 프로그래밍의 기본인 델리게이트에 대하 알아봤고, 델리게이트의 특별한 케이스인 이벤트에 대해서도 알아봤다. 다음 연재에서는 비동기 프로그래밍의 원리와 사용 방법에 대해 알아 볼 것이다. @

비동기 프로그래밍이라는 것은 한 가지 일을 할 때 그 일이 끝날 때까지 기다리는 것이 아니라 그 일은 그 일 나름대로 진행하면서 동시에 자신의 일을 계속 할 수 있는 것을 말한다. 이러한 기능이 가능하게 하려면 결국 쓰레드를 써야만 한다. 이 쓰레드를 이용해 비동기 프로그래밍 기법을 흉내내 보자.   
  
**쓰레드를 이용한 비동기 프로그래밍**   
<리스트 1>은 쓰레드를 이용해 다른 작업을 동시에 하는 것을 보여준 예이다. 0부터 9까지 출력하는 프로그램으로 별개의 쓰레드를 하나 더 돌려서 이들을 동시에 처리하고 있다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 1> 0부터 9까지 출력하는 프로그램 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class Class1  {                  public void DoSubWork()                  {                                  // 서브 작업                                  for(int i =0 ; i < 10 ; i++)                                  {                                                  // 많은 계산을 요구하는 작업                                                  for(int j=0; j < 10000000 ; j++) {}                                                  Console.WriteLine(“SubWork :{0} “,i);                                  }                                                  Console.WriteLine(“부 작업 완료”);                  }                   // 해당 응용 프로그램의 주 진입점이다.                  [STAThread]                  static void Main(string[] args)                  {                                  // TODO: 여기에 응용 프로그램 시작 코드를 추가                                  Thread t = new Thread                                  (new ThreadStart(Class1.DoSubWork)); t.Start();                                   // 메인 작업                                  for(int i =0 ; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“MainWork:{0} “,i);                                  }                                  Console.WriteLine(“메인 작업 완료”);                  }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

이 프로그램의 결과는 컴퓨터 사양에 따라 차이가 나고, 할 때마다 다른 결과가 나올 수 있으나 대략 다음의 결과와 비슷할 것이다.

MainWork : 0   
SubWork : 0   
SubWork : 1   
MainWork : 1   
SubWork : 2   
MainWork : 2   
MainWork : 3   
SubWork : 3   
MainWork : 4   
SubWork : 4   
MainWork : 5   
SubWork : 5   
MainWork : 6   
SubWork : 6   
SubWork : 7   
MainWork : 7   
SubWork : 8   
MainWork : 8   
MainWork : 9   
메인 작업 완료   
SubWork : 9   
부 작업 완료

즉 쓰레드를 이용하면 이처럼 두 가지 작업을 동시에 처리할 수 있다. 그런데 쓰레드의 생성자를 보면 ThreadStart라는 델리게이트를 취하고 있음을 볼 수 있을 것이다. 이 ThreadStart 델리게이트의 형식을 보면 다음과 같다.

public delegate void ThreadStart();

리턴형은 void이고, 인자로는 아무런 값을 받지 않는 델리게이트이다. 따라서 우리가 쓰레드를 사용해 다른 함수를 가동할 때는 인자도 없고 리턴 값도 없는 함수만을 사용해야 된다는 얘기다. 그런데 세상사라는 것이 그리 간단하지 않은 것이, 앞의 경우만 보더라도 0부터가 아닌 임의의 숫자로 시작하고 싶어서 그 숫자를 인자로 넘기고 싶을 때가 있을 것이다. 이럴 때에는 어떻게 해야 할 것인가? <리스트 2>의 예제를 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 2> 원하는 수부터 9까지 출력하는 프로그램 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class CSubWork  {                  private int start;                  public CSubWork(int i) { start = i; }                  public void DoSubWork2()                  {                                  // 서브 작업                                  for(int i =start ; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“SubWork :{0} “,i);                                  }                                  Console.WriteLine(“부 작업 완료”);                  }  }   class Class1  {                  [STAThread]                  static void Main(string[] args)                  {                                  CSubWork = new CSubWork(5);                                  Thread t = new Thread ( new ThreadStart                                  (c.DoSubWork2) ); t.Start();                                   // 메인 작업                                  for(int i =0 ; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“MainWork:{0} “,i);                                  }                                  Console.WriteLine(“메인 작업 완료”);                  }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

즉 비동기적으로 함수를 랩핑할 수 있는 클래스를 새로 만들어서 데이터 관리를 하면 되는 것이다. 그러므로 따로 인자를 넘기지 않더라도 클래스에 그 값을 줘서 해결할 수 있다. 다음은 <리스트 2>의 결과이다.

MainWork : 0   
SubWork : 5   
MainWork : 1   
SubWork : 6   
MainWork : 2   
SubWork : 7   
MainWork : 3   
SubWork : 8   
MainWork : 4   
SubWork : 9   
부 작업 완료   
MainWork : 5   
MainWork : 6   
MainWork : 7   
MainWork : 8   
MainWork : 9   
메인 작업 완료

인자가 필요할 때에는 이렇게 해서 문제를 해결했는데, 이제 남은 또 다른 가능성은 리턴 값이 필요할 때이다. 되돌려 받는 값이 없다면 caller 측에서는 호출하고 잊어버리면 그만이다. 이를 ‘fire-and-forget style programming’이라고 한다. 그런데 되돌려 받는 값이 있다면 문제가 되기 시작한다. 상대방에게 비동기적으로 수행하라고 일을 시켜 놓았으니, 언제 끝날지 모르기 때문이다.   
  
이를 확인하는 방법은 크게 두 가지가 있는데, caller 측에서 비동기 호출이 끝났는지 확인해서 끝났다면 리턴 값을 받아오는 방법과 caller 측에서 델리게이트를 넘겨주고 호출된 쪽에서 연산이 다 끝나면 그 델리게이트를 호출해줘서 리턴 값을 받아오는 방법이 있다. 이를 그림으로 나타내면 <그림 1>과 같다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129829-h1.jpg |
| <그림 1> 비동기 호출로부터 결과값을 받는 방법 |

이번에는 비동기 부분에서 넘겨온 리턴 값을 가지고, 메인 부분에서 그 부분을 시작 값으로 하여 출력하는 프로그램을 만들 것이다. 지금까지 쓰레드 부분을 메인 부분에서 만들어 줬는데, 이번에는 메인 부분의 코드를 간결하게 하기 위하여 새로운 랩핑 클래스를 만들고 그곳에서 쓰레드를 담당하게 할 것이다. 쓰레드를 시작하는 함수는 Begin×××라는 이름을 붙여주고 결과 값을 받아 오는 함수는 End×××라고 붙여주자. <리스트 3>을 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 3> 리턴 값을 위한 비동기 클래스 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | // 콜백 함수 델리게이트  public delegate void CallBack(int result);   // 리턴 값을 처리하기 위한 클래스  class CSubWork2  {                  private int start; // 시작 값                  private CallBack callback; // 콜백 함수 델리게이트                  public bool isCompleted;                  // 연산이 끝났는가? caller 측에서 물어볼 때 대답해 주기 위해서                  public int ret; // 결과 값                   public CSubWork2(int i, CallBack d)                  {                                  start = i;                                  callback = d;                                  isCompleted = false                  }                                   public void DoSubWork2()                  {                                  ret = Calc(start);                                  isCompleted = true                                  if ( callback != null ) callback(ret );                  }                   protected int Calc(int s)                  {                                  // 서브 작업                                  for(int i =s ; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“SubWork :{0} “,i);                                  }                                  Console.WriteLine(“부 작업 완료”);                                  return s+1;                  }  }   class CAsync  {                  CSubWork2 w;                  public void BeginWork(int i, CallBack d)                  {                                  w = new CSubWork2(i, d);                                  Thread t = new Thread(new ThreadStart                                  ( w.DoSubWork2 )); t.Start();                  }                   public int EndWork()                  {                                  // 결과 값이 나올 때까지 블럭                                  do                                  {                                                  if ( w.isCompleted == true ) break                                  }while(true);                                  return w.ret;                  }                   public bool IsCompleted()                  {                                  return w.isCompleted;                  }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

리턴 값이 준비돼 있는지 안 돼 있는지 확인하기 위해 IsCim pleted하는 메쏘드를 준비했다. 또한 콜백(callback) 형식으로 델리게이트를 넘길 때를 위해 그에 대한 델리게이트도 마련해뒀다. CAsync 클래스의 BeginWork 함수에서 인자와 콜백 함수를 넘겨주게 되는데 이때 만약 콜백 함수가 필요없다면 null을 넘겨주면 된다. 그러면 대신 IsCompleted하는 함수로 비동기 연산의 종료 여부를 확인할 수 있다.   
  
한편 EndWork 함수에서는 비동기 연산이 종료되지도 않았는데 이 함수를 호출하면 준비되지 않는 결과 값을 가져가는 오류를 미연에 방지하기 위해 결과 값이 나올 때까지 블럭되게 한 후, 결과 값이 나오면 리턴하도록 했다. 그럼 먼저 caller 측에서 비동기 연산의 종료를 확인하는 예제를 보자(<리스트 4>).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 4> 비동기 연산 종료 여부 확인하기 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | CAsync a = new CAsync();  a.BeginWork(4,null);   while(!a.IsCompleted())  {                  // 메인 작업                  Console.Write(“.”);   }  for(int i = a.EndWork() ; i < 10 ; i++)  {                  for(int j=0; j < 10000000 ; j++) {} // 많은 계산을 요구하는 작업                  Console.WriteLine(“MainWork:{0} “,i);  }  Console.WriteLine(“메인 작업 완료”); | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

이번 예제에서는 콜백 함수가 필요없으므로 BeginWork 함수에서 null을 넘겨줬다. 그리고 대신 IsCompleted 함수를 이용해 종료 여부를 확인하는 동안 메인에서는 자신의 일을 할 수 있도록 했다. 종료된 후에는 EndWork 함수를 이용하여 결과 값을 가져와서 메인 부분의 일을 처리했다. 이 프로그램의 결과는 다음과 같다.

................................................................   
................................................................   
................................................SubWork : 4   
SubWork : 5   
SubWork : 6   
SubWork : 7   
.................................................................   
.................................................................   
..................................SubWork : 8   
SubWork : 9   
부 작업 완료   
MainWork : 5   
MainWork : 6   
MainWork : 7   
MainWork : 8   
MainWork : 9   
메인 작업 완료

이번에는 콜백 함수를 이용해 결과 값을 받아 오는 예제를 살펴보자(<리스트 5>). 콜백 함수에서는 비동기 함수의 리턴 값을 인자로 받아 와서 그 일을 하고 수행하고 있다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 5> 델리게이트로 결과값 받아오기 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class Class1  {                  // 비동기 함수가 대신 호출해 줄 콜백 함수                  public static void CallMe(int ret)                  {                                  // 메인 작업                                  for(int i = ret; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“MainWork:{0} “,i);                                  }                  Console.WriteLine(“메인 작업 완료”);                  }                   [STAThread]                  static void Main(string[] args)                  {                                  CAsync b = new CAsync();                                  b.BeginWork(4,new CallBack(CallMe));                                   for(int i=0; i<400; i++)                                  {                                                  // 아무 작업 수행                                                  Console.Write(“.”);                                  }                  }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

이 방법을 쓰면 caller 측에서 일일이 확인하지 않아도, 호출된 비동기 함수 부분에서 다 끝났다고 알려주는 격이 된다. 결과는 다음과 같다.

................................................................................   
................................................................................   
....................................................SubWork : 4   
SubWork : 5   
SubWork : 6   
SubWork : 7   
................................................................................   
................................................................................   
............................SubWork : 8   
SubWork : 9   
부 작업 완료   
MainWork : 5   
MainWork : 6   
MainWork : 7   
MainWork : 8   
MainWork : 9   
메인 작업 완료

이상이 비동기 프로그래밍의 대략적인 내부 구현이다. 실제 닷넷 프레임워크 내부적으로는 이보다 훨씬 더 복잡하게 진행이 되지만 대략적인 것은 이와 비슷하다. 그럼 이제는 실제 델리게이트를 이용해 비동기 프로그래밍을 해 보자.   
  
**델리게이트를 이용한 비동기 프로그래밍의 실제**   
이번 예제는 앞서 우리가 만든 프로그램과 비슷하다. 시작 숫자를 인자로 넘겨주면 그 숫자부터 프린트하는 프로그램이다. 이를 비동기 호출로 하기 위해 그에 대한 델리게이트를 다음과 같이 선언했다.

public delegate void SubWork(int i);

반환형을 다루는 예제는 조금 뒤에 다룰 것이므로, 지금은 반환형이 없는 델리게이트를 이용하자. 인자로는 시작 숫자를 넘겨줬다. 우리가 이렇게 델리게이트를 만들면 지난 시간에 소개했듯이 컴파일러는 이를 바탕으로 하여 다음과 같은 클래스를 만들게 된다.

public class SubWork : System.MulticastDelegate   
{   
   // 생성자   
   public SubWork (object target, int32 methodPtr);   
  
   public void virtual Invoke( int i );   
  
   public virtual IAsyncResult BeginInvoke( int i,   
      AsyncCallback callback, Object object);   
  
   public virtual void EndInvoke( IAsyncResult result);   
}

BeginInvoke는 앞에서 BeginWork와 비슷한 역할을 한다. 먼저 델리게이트가 받을 인자가 오고, 그 다음에 콜백 함수가 오고, 추가로 상태를 지정할 수 있는 인자를 쓸 수 있다. 이는 추가 정보를 넘겨줄 필요가 있을 때에만 쓰는 것이므로, 필요없다면 안 써도 된다. 그럼 이를 실제로 테스트해 보자(<리스트 6>).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 6> 델리게이트를 이용한 비동기 호출 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | public delegate void SubWork(int i);  class Class1  {                  public static void DoSubWork(int start)                  {                                   for(int i=start; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“SubWork :{0}”,i);                                  }                  }                   [STAThread]                  static void Main(string[] args)                  {                                  SubWork d = new SubWork(DoSubWork);                                  d.BeginInvoke(3,null,null);                                   for(int i=0; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“MainWork:{0}”,i);                                  }                  }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

앞서 우리가 한 예제와 동일한 기능을 하는 예제이다. 델리게이트에는 기본적으로 BeginInvoke라는 메쏘드가 있어 프로그래머가 손수 쓰레드 관련 코딩을 하지 않고도 손쉽게 비동기 호출을 할 수 있게 해 준다. 결과는 다음과 같다.

MainWork : 0   
SubWork : 3   
MainWork : 1   
SubWork : 4   
SubWork : 5   
MainWork : 2   
MainWork : 3   
SubWork : 6   
MainWork : 4   
SubWork : 7   
MainWork : 5   
SubWork : 8   
MainWork : 6   
SubWork : 9   
MainWork : 7   
MainWork : 8   
MainWork : 9

이와 같이 fire-and-forget형 프로그래밍의 경우 간단하지만 만약 반환 값을 다뤄야 할 경우는 약간 복잡해진다. 닷넷 플랫폼에서는 반환 값을 다루기 위한 4가지 스타일의 프로그래밍 기법을 제공한다.

1. Use Callbacks : 콜백 델리게이트를 이용해 비동기 부분에서 연산이 다 끝나면 델리게이트를 호출해 주는 방식   
2. Poll Completed : caller 부분에서 연산이 다 끝났는지 IsCompleted라는 속성을 이용하여 계속 확인해 보는 방식   
3. Begin Invoke, End Invoke : caller 측에서 결과 값을 받기 위하여 블러킹돼 기다리는 방식   
4. Begin Invoke, Wait Handle, End Invoke : 앞의 방식과 비슷하나 wait handle에서 제한 시간을 설정해 줌으로써 계속 블러킹되는 것을 방지할 수 있다.

그럼 이들에 대한 예제를 하나씩 살펴보도록 하자. 이 예제는 SubWork에서 반환 값을 주는데, 역시 앞의 예제와 비슷하게 main 부분에서는 시작 값으로 사용될 값을 넘겨주게 된다. 먼저 <리스트 7>을 통해 1번의 경우부터 보도록 하자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 7> Use Callbacks 방식 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | public delegate int SubWork2(int i);  class Class1  {                  public static int DoSubWork2(int start)                  {                                  for(int i=start; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“SubWork :{0}”,i);                                  }                                  return start + 1;                  }                   public static void CallMe(IAsyncResult ar)                  {                                  SubWork2 d = (SubWork2) ((AsyncResult)ar)                                  .AsyncDelegate;                                  int result = d.EndInvoke(ar);                                  for(int i=result; i < 10 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  Console.WriteLine(“MainWork:{0}”,i);                                  }                  }                   [STAThread]                  static void Main(string[] args)                  {                                  // 여섯번째 예제                                  Console.WriteLine(“\*\*\* 여섯번째 예제 \*\*\*”);                                  SubWork2 d2 = new SubWork2(DoSubWork2);                                  d2.BeginInvoke(3,new AsyncCallback(CallMe),null);                                   // 아무런 작업                                  for(int i =0; i<300;i++) Console.Write(“.”);                                   // 그냥 끝나면 안되므로 키입력 까지 대기                                  Console.Read ();                  }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

이번에는 콜백 함수를 이용해서 함께 넘겨주고 있다. 이 콜백 델리게이트의 형식을 보면 다음과 같다.

public delegate void AsyncCallback(IAsyncResult ar)

즉 리턴 값은 없고, 인자로 IAsyncResult라는 것을 받고 있다. 이는 닷넷 프레임워크에서 콜백 함수를 호출할 때 그 인자를 자동으로 넘겨주므로 걱정하지 않아도 된다. 이때 넘어오는 인자에서 AsyncDelegate라는 속성을 이용하면 해당 델리게이트를 받아올 수 있다. 따라서 이를 이용해 EndInvoke를 호출하여 결과 값을 받아오는 것이다. 결과는 다음과 같다.

SubWork : 3   
SubWork : 4   
SubWork : 5   
................................................................................   
................................................................................   
.............................SubWork : 6   
SubWork : 7   
SubWork : 8   
SubWork : 9   
................................................................................   
...............................MainWork : 4   
MainWork : 5   
MainWork : 6   
MainWork : 7   
MainWork : 8   
MainWork : 9

<리스트 8>은 2의 경우로서 caller 측에서 계속 폴링(polling)하면서 연산이 다 끝났는지 확인하는 방법이다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 8> Poll Completed | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | SubWork2 d3 = new SubWork2(DoSubWork2);  IAsyncResult ar = d3.BeginInvoke(3,null,null);  while( !ar.IsCompleted )  {                  Console.Write(“.”);  }   int ret = d3.EndInvoke(ar);   for(int i= ret; i < 10 ; i++)  {                  for(int j=0; j < 10000000 ; j++) {} // 많은 계산을 요구하는 작업                  Console.WriteLine(“MainWork:{0}”,i);  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

caller 측에서 IAsyncResult의 IsCompleted 속성을 이용하여 계속적으로 연산이 끝났는지 안 끝났는지 확인하고 있다. 확인하는 동안 caller 측에서는 계속 다른 작업을 할 수 있다. 결과는 다음과 같다.

................................................................................   
................................................................................   
...........................................MainWork : 5   
MainWork : 6   
MainWork : 7   
MainWork : 8   
................................................................................   
................................................................................   
......................................MainWork : 9   
SubWork : 3   
SubWork : 4   
SubWork : 5   
................................................................................   
................................................................................   
......................................SubWork : 6   
SubWork : 7   
SubWork : 8   
SubWork : 9   
MainWork : 4   
MainWork : 5   
MainWork : 6   
MainWork : 7   
MainWork : 8   
MainWork : 9

<리스트 9>는 3의 경우인데, 이는 결국 caller 측에서 비동기 호출된 부분이 끝날 때까지 블러킹되고 있으므로, 별로 권장하는 방법은 아니다. 이렇게 되면 비동기 호출을 할 의미가 없기 때문이다. 그러나 이도 결과 값을 받는 방법의 하나이므로 알아두자. 이번 예제는 콘솔 프로그램이 아닌 윈도우용 프로그램으로 만들었다. EndInvoke를 실행했을 때 caller 측이 블러킹된다는 것을 보여주기 위해서 윈도우용 프로그램으로 만들었다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 9> Begin Invoke, End Invoke | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | public class Form1 : System.Windows.Forms.Form  {                  private System.Windows.Forms.TextBox textBox1;                  private System.Windows.Forms.Button button1;                  public delegate void SubWork();                  public void DoSubWork()                  {                                  // 서브 작업                                  string str;                                  for(int i =0 ; i < 100 ; i++)                                  {                                                  for(int j=0; j < 10000000 ; j++) {}                                                  // 많은 계산을 요구하는 작업                                                  str = String.Format(“SubWork :{0}\r\n”,i);                                                  textBox1.Text += str;                  }  }   [STAThread]  static void Main()                  {                                  Application.Run(new Form1());                  }                   private void button1\_Click(object sender, System.EventArgs e)                  {                                  SubWork d = new SubWork(DoSubWork);                                  IAsyncResult r = d.BeginInvoke(null,null);                                  d.EndInvoke(r);                  }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

이 프로그램을 실행하고 ‘비동기 연산 시작’ 버튼을 누르면, 비동기 연산을 시작한다. 그러나 연산 결과가 바로 나오지도 않을 것이며 윈도우가 움직이지도 않을 것이다. 모든 연산이 끝난 후에 한꺼번에 나올 것이다. 이는 EndInvoke라는 함수 때문에 블러킹돼버려서 그런 것이다. 그래서 연산 결과를 표시하기 위해 윈도우 화면 갱신도 못하고 마치 다운된 것처럼 멈췄다가 결과가 나오게 된다. <화면 1>이 그 결과 화면이다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129829-1.jpg |
| <화면 1> 윈도우에서 비동기 연산을 한 결과 화면 |

이번에는 마지막 방법인 4에 대해 알아보자. 이 방법은 3번 방법과 비슷한데 한 가지 다른 점은 time out을 정해줄 수 있어서 마냥 블러킹되는 것이 아니라 일정 시간이 넘어버리면 끊어버릴 수 있는 기능을 제공한다. 이번에는 비동기 연산을 0.1초 안에 해내지 못하면 블러킹을 해제하고 main 부분을 실행하는 프로그램을 만들어 본다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 10> Begin Invoke, Wait Handle, End Invoke | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | SubWork2 d4 = new SubWork2(DoSubWork2);  IAsyncResult ar2 = d4.BeginInvoke(3,null,null);   if ( ar2.AsyncWaitHandle.WaitOne(100,false) == false )                  {                                  ret = 4;                                  Console.WriteLine(“도중 차단”);                  }                  else                  {                                  ret = d4.EndInvoke(ar2);                  }   for(int i= ret; i < 10 ; i++)                  {                                  for(int j=0; j < 10000000 ; j++) {}                                  // 많은 계산을 요구하는 작업                                  Console.WriteLine(“MainWork:{0}”,i);                  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

<리스트 10>은 caller 측에서 결과 값을 받기 위해서 0.1초간만 기다리고 그 안에 결과 값을 받지 못하면 블러킹을 중지하고 자신의 코드를 수행하는 예제이다. 결과는 다음과 같다. 결과에서 확인할 수 있는 것처럼 비동기 호출이 0.1초 안에 연산을 끝내지 못했기 때문에 main은 블러킹을 멈추고 자신이 할 일을 하고 있다.

SubWork : 3   
SubWork : 4   
SubWork : 5   
SubWork : 6   
SubWork : 7   
도중 차단   
MainWork : 4   
MainWork : 5   
MainWork : 6   
SubWork : 8   
SubWork : 9   
MainWork : 7   
MainWork : 8   
MainWork : 9

**리턴 값을 이용하지 않을 경우**   
이상으로 비동기 호출에서 결과 값을 얻기 위한 4가지 방법을 소개했다. 그러나 비동기 호출에서 결과 값을 받기 위해 꼭 리턴 값을 이용해야 하는 것은 아니다. 우리가 인자로 넘겨줄 때 ref형이나, out형으로 넘겨주면 꼭 리턴 값을 이용하지 않더라도 그 결과 값을 받는 방법이 된다. 그러나 여기서 한 가지 주의할 것은 인자(parameter) 값의 업데이트도 비동기적으로 일어난다는 것이다. <리스트 11>을 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 11> Ref 파라미터를 사용한 예 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | public delegate void SubWork3(ref int i);   public static void DoSubWork3(ref int i)  {                  i \*= 2;  }    [STAThread]  static void Main(string[] args)  {                  Console.WriteLine(“\*\*\* 열번째 예제\*\*\*”);                  int v = 42;                  SubWork3 d5 = new SubWork3(DoSubWork3);                  IAsyncResult ar3 = d5.BeginInvoke(ref v,null,null);                  ar3.AsyncWaitHandle.WaitOne();                  Console.WriteLine(“EndInvoke를 호출하기 이전의 값 :{0}”,v);                  d5.EndInvoke(ref v,ar3);                  Console.WriteLine(“EndInvoke를 호출하고 난 후의 값:{0}”,v);  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

이번에는 값을 ref형으로 넘기고 있다. 만약 비동기 호출이 아닌 동기 호출이라면, 처음에 결과 값이 84가 나와야 할 것이다. 그러나 비동기 호출이기 때문에, 파라미터값의 업데이트도 비동기적으로 하므로 EndInvoke를 호출해야만 값의 업데이트가 일어난다. 그 결과는 다음과 같다.

EndInvoke를 호출하기 이전의 값 : 42   
EndInvoke를 호출하고 난 후의 값 : 84

그런데 여기서 만약 reference type을 파라미터로 넘기게 되면 비동기 호출 부분에서는 이 값을 실시간적으로 업데이트를 한다. 즉 EndInvoke를 호출할 필요없이, 자신이 연산을 하면서 값을 지속적으로 갱신을 하는 것이다. 이번에는 reference type인 byte형 배열을 파라미터로 넘겨서 그 값을 확인해 보겠다. <리스트 12>를 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 12> reference type을 파라미터로 넘긴 예 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | public delegate void SubWork4(byte[] b);    public static void DoSubWork4(byte[] b)  {                  for(byte i=0; i < b.Length; i++)                  {                                  for(int j=0; j < 10000000 ; j++) {}                                  // 많은 계산을 요구하는 작업                                  b[i] = (byte)(i\*i);                  }  }   [STAThread]  static void Main(string[] args)  {                  Console.WriteLine(“\*\*\* 열한번째 예제\*\*\*”);                  byte[] b = new byte[10];                  SubWork4 d6 = new SubWork4(DoSubWork4);                  IAsyncResult ar4 = d6.BeginInvoke(b, null, null);                  // ar4.AsyncWaitHandle.WaitOne();                  for(int i=0; i< b.Length; i++)                                  Console.WriteLine(“b[{0}]={1}”,i,b[i]);  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

이 예제는 byte형 배열을 만든 후(이때 모든 배열의 값은 0으로 자동 초기화된다) 이를 비동기 호출로 넘겨주면, 비동기 연산 부분에서는 이 배열의 값을 넣어주는 작업을 하는 예제이다. <리스트 12>에서 결과 값이 다 나오도록 기다리는 부분을 주석 처리했는데, 그렇게 한 이유는 비동기 호출이 값을 실시간으로 고치는지 확인해 보기 위해서이다.   
  
그 주석 처리한 부분의 주석을 없애 버리면 연산이 다 끝날 때까지 기다리기 때문에 언제나 정확한 값을 얻을 수 있을 것이다. 그러나 우리는 과연 비동기 호출이 값을 실시간으로 바꾸는지 확인하기 위한 것이므로 주석 처리를 했다. 결과 값은 컴퓨터 사양에 따라 다르고 할 때마다 다른 값이 나온다. 다음은 필자 컴퓨터에서 실행한 결과이다.   
  
결과를 보면 5번 방까지는 제대로 들어갔는데, 그 이후로 값이 안 들어와 있다. 이는 caller 측에서 비동기 연산이 아직 다 끝나지 않았는데, 이는 그 값을 꺼내봤기 때문이다.

b[0]=0   
b[1]=1   
b[2]=4   
b[3]=9   
b[4]=16   
b[5]=25   
b[6]=0   
b[7]=0   
b[8]=0   
b[9]=0

**쓰레드 풀을 이용하자**   
이번 호에서는 쓰레드를 이용한 비동기 프로그래밍의 원리와 4가지 구현 방법에 대해 알아봤다. 그런데 여기서 아직 해결하지 못한 문제점이 남아 있다. 닷넷 프레임워크는 우리가 매번 비동기 호출을 할 때마다 쓰레드를 새로 만들어서 하는 것일까?   
  
만약 그렇다면, 쓰레드라는 것이 적당한 수가 유지된다면 문제가 안 되지만 과도한 쓰레드의 생성은 오히려 쓰레드를 교체하면서 생기는 컨텍스트 체인지 오버헤드(context change overhead)가 있을 것이다. 닷넷 프레임워크에서는 이 문제를 쓰레드 풀(pool)을 이용해서 해결하고 있다. 다음 연재에서는 이 쓰레드 풀을 이용한 비동기 호출에 대해 알아보고, 최종적으로 게임 서버를 완성할 것이다. @

지난 시간에는 쓰레드를 통한 비동기 프로그래밍의 원리와 구현에 대해 알아봤다. 이번 시간에는 쓰레드를 효과적으로 관리하기 위한 쓰레드 풀과 함께 서버 제작에 필요한 네트워크 기술을 설명한다. 쓰레드 풀을 이용한 비동기 프로그래밍과 소켓의 개념에 대해 알아 보고 최종적으로 게임 서버를 완성해 보자.   
  
비동기 프로그래밍은 내부적으로 쓰레드를 이용한다. 그런데 비동기 호출을 할 때마다 새로운 쓰레드를 생성해서 작업을 하게 되면, 많은 비동기 호출이 일어날 때에는 쓰레드의 수가 너무 많아져서 오히려 컨텍스트 스위칭(context switching)하는 데 시간이 더 걸리는 현상이 일어난다. 이러한 현상을 해결하기 위해서는 적절한 쓰레드 수를 유지하는 것이 해결 방법이라 할 수 있을 것이다. 닷넷에서는 그러한 관리 방법으로 쓰레드 풀이라는 것을 이용한다. 이를 이용해 시스템의 CPU 사용량에 따라 항상 적절한 쓰레드 수를 유지시켜 준다.   
  
**쓰레드 풀이란**   
먼저 풀(pool)의 사전적인 의미는 스위밍 풀(swimming pool)처럼 물 웅덩이, 저수지라는 뜻이 있다. 다른 뜻으로는 카 풀(car pool)처럼 공동으로 이용하는 것이라는 뜻이 있다. 여기서는 두 번째의 공동으로 이용한다는 의미이다. 카 풀이라는 것이 에너지 절약을 위해서 이웃끼리 통근 시간 같은 때에 차를 같이 이용하는 것을 말한다. 쓰레드 풀도 이와 비슷한 것으로 쓰레드들이 시스템의 효율성을 높이기 위하여 집합적으로 모여 있는 것을 쓰레드 풀이라고 부른다.   
  
쓰레드 풀은 쓰레드 생성 요청이 있을 때마다 그 쓰레드를 바로 생성하는 것이라 일단 큐에 그 쓰레드를 넣어 두었다가 쓰레드 풀이 그 요청을 처리할 수 있는 여유가 있을 때 큐에서 하나씩 꺼내서 처리를 한다. 닷넷 환경에서는 기본적으로 쓰레드 풀 안에서의 최대 25개의 쓰레드를 넣어 둘 수 있다. 이를 그림으로 나타내면 <그림 1>과 같다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-1.jpg |
| <그림 1> 쓰레드 풀 |

**쓰레드 풀의 사용 방법**   
닷넷에서 쓰레드 풀을 이용하기 위해서는 쓰레드 풀 클래스를 이용하면 된다. <리스트 1>은 쓰레드 풀 클래스의 메쏘드들이다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 1> 쓰레드 풀 클래스 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | {       // Constructors       // Methods       public static bool BindHandle(IntPtr osHandle);       public virtual bool Equals(object obj);       public static void GetAvailableThreads(ref Int32 workerThreads,            ref Int32 completionPortThreads);       public virtual int GetHashCode();       public static void GetMaxThreads(ref Int32 workerThreads,            ref Int32 completionPortThreads);       public Type GetType();       public static bool QueueUserWorkItem(            System.Threading.WaitCallback callBack);       public static bool QueueUserWorkItem(            System.Threading.WaitCallback callBack, object state);       public static System.Threading.RegisteredWaitHandle            RegisterWaitForSingleObject(            System.Threading.WaitHandle waitObject,            System.Threading.WaitOrTimerCallback callBack,            object state, int millisecondsTimeOutInterval,            bool executeOnlyOnce);       public static System.Threading.RegisteredWaitHandle            RegisterWaitForSingleObject(            System.Threading.WaitHandle waitObject,            System.Threading.WaitOrTimerCallback callBack,            object state, UInt32 millisecondsTimeOutInterval,            bool executeOnlyOnce);       public static System.Threading.RegisteredWaitHandle            RegisterWaitForSingleObject(            System.Threading.WaitHandle waitObject,            System.Threading.WaitOrTimerCallback callBack,            object state, long millisecondsTimeOutInterval,            bool executeOnlyOnce);            public static System.Threading.RegisteredWaitHandle            RegisterWaitForSingleObject(            System.Threading.WaitHandle waitObject,            System.Threading.WaitOrTimerCallback callBack,            object state, TimeSpan timeout, bool executeOnlyOnce);       public virtual string ToString();       public static bool UnsafeQueueUserWorkItem(            System.Threading.WaitCallback callBack, object state);       public static System.Threading.RegisteredWaitHandle            UnsafeRegisterWaitForSingleObject(            System.Threading.WaitHandle waitObject,            System.Threading.WaitOrTimerCallback callBack,            object state, int millisecondsTimeOutInterval,            bool executeOnlyOnce);       public static System.Threading.RegisteredWaitHandle            UnsafeRegisterWaitForSingleObject(            System.Threading.WaitHandle waitObject,            System.Threading.WaitOrTimerCallback callBack,            object state, UInt32 millisecondsTimeOutInterval,            bool executeOnlyOnce);       public static System.Threading.RegisteredWaitHandle            UnsafeRegisterWaitForSingleObject(            System.Threading.WaitHandle waitObject,            System.Threading.WaitOrTimerCallback callBack,            object state, long millisecondsTimeOutInterval,            bool executeOnlyOnce);       public static System.Threading.RegisteredWaitHandle            UnsafeRegisterWaitForSingleObject(            System.Threading.WaitHandle waitObject,            System.Threading.WaitOrTimerCallback callBack,            object state, TimeSpan timeout, bool executeOnlyOnce);       } // end of System.Threading.ThreadPool | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

<리스트 1>을 보면 거의 모든 멤버가 static이고 public constructor가 없음을 볼 수 있을 것이다. 이는 닷넷에서는 하나의 프로세스당 한 개의 풀만을 허용하기 때문이다. 즉 모든 비동기 호출은 같은 하나의 풀을 통해 이뤄진다. 따라서 제 3자의 컴포넌트가 새로운 풀을 만들어서 기존의 풀과 함께 돌아감으로써 생기는 오버헤드를 줄일 수 있는 것이다. 쓰레드 풀의 큐에 새로운 쓰레드를 추가시키려면 다음과 같은 메쏘드를 이용한다.

public static bool QueueUserWotkItem ( WaitCallBack callBack ,object state);

우선 WaitCallBack이라는 대리자를 이용하여 처리할 함수를 등록하고, state를 이용하여 함께 넘길 파라미터를 지정해 준다.

public delegate void WaitCallBack( object state );

WaitCallBack 대리자의 형식이 반환 값은 없고, 인자로는 state 하나만을 받는 형식이다. 따라서 쓰레드를 사용할 함수는 이와 같은 signature를 가져야 한다. 이를 이용하여 0부터 3까지 출력하는 3개의 작업을 만들어 보자. 이를 실행하면 다음과 같이 보통 일반 쓰레드를 이용하는 것과 비슷한 결과 화면을 볼 수 있다. 3개의 작업이 동시에 이뤄지고 있다.

1번 작업 : 0   
2번 작업 : 0   
1번 작업 : 1   
3번 작업 : 0   
2번 작업 : 1   
1번 작업 : 2   
3번 작업 : 1   
2번 작업 : 2   
1번 작업 : 3   
3번 작업 : 2   
2번 작업 : 3   
1번 작업 끝   
3번 작업 : 3   
2번 작업 끝   
3번 작업 끝

이번에는 Thread.IsThreadPoolThread이라는 속성을 이용하여 정말 쓰레드 풀을 이용하고 있는지와, 현재 쓰레드의 고유 번호를 나타내주는 메쏘드인 GetHashCode를 이용하여 그 값을 확인해 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 2> 0부터 3까지 출력하는 4개의 작업 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class Class1  {       [STAThread]       static void Main(string[] args)       {            WaitCallback callBack;             callBack = new WaitCallback(Calc);            ThreadPool.QueueUserWorkItem(callBack,1);            ThreadPool.QueueUserWorkItem(callBack,2);            ThreadPool.QueueUserWorkItem(callBack,3);                  Console.ReadLine();       }       static void Calc(object state)       {            for(int i= 0; i < 4; i++)       {            Console.WriteLine(“{0}번 작업: {1}”,state,i);            Thread.Sleep(1000);       }            Console.WriteLine(“{0}번 작업 끝”,state);       }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

<리스트 2>에 <리스트 3>과 같은 코드를 추가한다. 결과는 다음과 같다.

Main thread. Is Pool thread:False, Hash : 2   
1번 작업 thread. Is Pool thread:True, Hash : 7   
1번 작업 : 0   
2번 작업 thread. Is Pool thread:True, Hash : 8   
2번 작업 : 0   
1번 작업 : 1   
3번 작업 thread. Is Pool thread:True, Hash : 9   
3번 작업 : 0   
2번 작업 : 1   
1번 작업 : 2   
3번 작업 : 1   
2번 작업 : 2   
1번 작업 : 3   
3번 작업 : 2   
2번 작업 : 3   
1번 작업 끝   
3번 작업 : 3   
2번 작업 끝   
3번 작업 끝

즉 메인 쓰레드는 쓰레드 풀에서 하는 작업이 아니며, 나머지는 쓰레드 풀 내에서 작업하고 있음을 볼 수 있을 것이다. 그리고 각자 다른 해시코드를 가지고 있으므로 각자 새로운 쓰레드를 생성해서 작업하고 있는 것이다. 이는 현재 CPU 사용량에 여유가 있었기 때문에 각자 하나씩의 쓰레드를 생성해서 작업을 한 것이다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 3> 쓰레드 풀 확인 방법 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | // 메인 부분에 추가  Console.WriteLine(“Main thread. Is Pool thread:{0}, Hash: {1}”,  Thread.CurrentThread.IsThreadPoolThread,  Thread.CurrentThread.GetHashCode());   // 쓰레드 작업 부분에 추가  Console.WriteLine(“{0}번 작업 thread. Is Pool thread:{1}, Hash: {2}”,  state,  Thread.CurrentThread.IsThreadPoolThread,  Thread.CurrentThread.GetHashCode()); | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

만약 CPU 사용량이 많아져서 컨텍스트 스위칭 시간이 더 걸릴거라 판단되면, 다른 쓰레드들은 큐에서 대기하다가 기존 작업이 끝나고 그 쓰레드를 재사용해서 작업을 하게 된다. 이에 대한 예를 보자. CPU 사용량을 높이려면 다음과 같은 함수를 추가한다.

int ticks = Environment.TickCount;   
while( Environment.TickCount - ticks < 500 );

Environment.TickCount 속성은 마지막 리부팅한 후부터의 시간을 millisecond 단위로 리턴해 준다. Thread.Sleep(1000)이라는 부분 대신 이 함수를 넣고 실행해 보면 <화면 1>과 비슷한 결과를 볼 수 있다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-h1.jpg |
| <화면 1> CPU 사용량을 높인 후의 쓰레드 푸 작동 화면 |

<화면 1>을 보면 CPU 사용량이 100%임을 확인할 수 있다. 그리고 결과를 보면 3번 작업이 1번 작업과 같은 해시코드를 사용하고 있다. 즉 같은 쓰레드를 재사용하고 있는 것이다. 그래서 1번 작업이 끝난 후에, 1번 작업이 쓰던 쓰레드를 3번 작업이 다시 사용하고 있는 것이다. 이처럼 쓰레드 풀이라는 것은 현재 시스템의 상황에 따라 적절히 쓰레드 개수를 유지시켜 줌으로써 효율성을 높이고 있다. 그럼 이제 정말 비동기 호출이 쓰레드 풀을 이용하는지 확인해 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 4> 비동기 호출 확인하기 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class Class1  {       public static void Calc()       {             Console.WriteLine(“Is pool:{0}”, Thread.CurrentThread.                 IsThreadPoolThread);            for(int i=1; i < 10 ; i++)       {                 for(int j=0; j < 10000000 ; j++) {} // 많은 계산을 요구하는 작업                 Console.WriteLine(“SubWork :{0}”,i);       }  }   [STAThread]  static void Main(string[] args)  {       SubWork d = new SubWork(Calc);       d.BeginInvoke(null,null);             for(int i=0; i < 10 ; i++)       {            for(int j=0; j < 10000000 ; j++) {} // 많은 계산을 요구하는 작업            Console.WriteLine(“MainWork:{0}”,i);       }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

<리스트 4>는 지난 시간에 했던 예제이다. 그곳에 단지 쓰레드 풀임을 확인할 수 있는 문장을 하나 추가했을 뿐이다. 결과는 다음과 같다.

Is pool:True   
SubWork : 1   
MainWork : 0   
SubWork : 2   
MainWork : 1   
SubWork : 3   
MainWork : 2   
SubWork : 4   
MainWork : 3   
MainWork : 4   
SubWork : 5   
SubWork : 6   
MainWork : 5   
SubWork : 7   
MainWork : 6   
SubWork : 8   
MainWork : 7   
SubWork : 9   
MainWork : 8   
MainWork : 9

이제 비동기 호출이 쓰레드 풀을 이용하는 것이 확실해졌다. 그러면 모든 비동기 호출은 이와 같이 델리게이트를 만들어서 해야 할까? 그것은 아니다. 우리가 매번 비동기 호출을 위해서 델리게이트를 만들어야 한다면 그것 또한 귀찮은 일일 것이다.   
  
그래서 닷넷에서는 미리 비동기 호출을 위한 함수들을 마련해 두고 있다. Begin×××, End×××로 표기되는 메쏘드들이 비동기 호출을 위해 미리 만들어 둔 함수들이다. 델리게이트의 BeginInvoke도 이와 같은 연장선에 보면 될 것이다. 우리는 소켓을 이용한 비동기 통신 방법에 대해 알아볼 것이므로 소켓과 관련된 비동기 함수들을 살펴볼 것이다. 그전에 소켓의 기본 개념부터 설명하겠다.   
  
**소켓이란?**   
일반적인 의미로 소켓이란, 전구의 소켓처럼 꽂는 구멍을 말한다. 즉 다른 것과 연결시켜 주는 구멍이다. 컴퓨터에서의 소켓도 이와 비슷한 의미이다. 네트워크의 다른 대상과 정보를 교환하기 위한 구멍인 것이다. 일종의 네트워크 자료 교환을 위한 파이프라인(pipeline)으로 생각하면 된다.   
  
일반적으로 네트워크에서 정보를 주고받기 위한 주소로 IP 어드레스라는 것을 사용한다. 그런데 이 주소는 대개 하나의 컴퓨터에 한 개의 주소가 할당된다. 그런데 네트워크 정보 교환은 하나의 컴퓨터뿐만 아니라 여러 컴퓨터와 정보를 주고받아야 하므로 하나의 IP 주소로는 이 정보를 어디로 보내야 하는지 구분할 수 없다.   
  
그래서 포트(Port)라는 개념을 쓴다. 이는 항구라는 뜻으로 각 네트워크 정보들이 통신하는 입구인 것이다. 일반적으로 HTTP는 80 포트를 사용하고, FTP는 21 포트를 사용한다. 그래서 어떤 한 컴퓨터에 네트워크 데이터를 보내더라도 포트 번호가 다르므로, HTTP용 데이터와 FTP용 데이터가 각각 제 자리를 찾아가는 것이다. 이를 그림으로 나타내면 <그림 2>와 같다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-2.jpg |
| <그림 2> 포트의 개념 |

일반적으로 포트 번호는 0∼65535까지 쓸 수 있지만 0∼1024번까지는 80번이나 21번처럼 미리 정해진 포트 번호를 사용하므로 사용자가 임의의 포트 번호를 사용하려면 그 이상의 번호를 사용하면 된다.   
  
<그림 2>를 보면 포트에 소켓이 연결되어 있음을 볼 수 있을 것이다. 특히 서버쪽을 보면 하나의 포트에 여러 개의 소켓이 달려있음을 볼 수 있을 것이다. 이는 다중의 클라이언트가 하나의 포트로 접속하기 때문이다. 각 클라이언트마다 이들의 데이터를 맡아서 중개해주는 파이프라인(소켓)이 따로 있어야 하기 때문에 하나의 포트에 여러 개의 소켓이 달려 있는 것이다.   
  
그런데 여기서 한 가지 의문점이 있을 수 있다. 하나의 포트에 여러 개의 네트워크 데이터들이 몰려들어 올텐데 서버는 이를 어떻게 구분해서 각자의 전담 파이프라인(소켓)으로 보내주는 것일까? 이는 TCP/IP의 헤더를 보면 쉽게 해결이 된다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-p1.jpg |
| <표 1> TCP/IP 헤더 |

<표 1>을 보면 IP 프로토콜의 헤더에는 보내는 곳과 받는 곳의 IP 주소가 들어 있다. 한편 TCP 헤더에는 보내는 곳과 받는 곳의 포트 번호가 들어 있다. 이들 4가지의 정보는 서로의 데이터를 확실히 구분하는 기준이 되므로, 서버측에서는 이 헤더를 보고 각자에 맞는 소켓으로 데이터를 보내주는 것이다.   
  
다시 <그림 2>를 보면 서버측의 포트 번호는 지정되어 있는 반면에 클라이언트측의 포트 번호는 일관성 없이 중구난방으로 아무 번호나 할당되어 있음을 볼 수 있을 것이다. 그 이유는 클라이언트 입장에서는 데이터를 보내야 하는 서버측의 포트 번호는 알아야 하지만 자신의 포트 번호는 그냥 비어있는 아무 번호나 써도 상관없다. 굳이 자신의 포트 번호를 미리 정하지 않아도 되는 것이다. 그래서 클라이언트가 서버로 연결할 때, 자신의 남는 포트 번호 중 아무나 한 개를 할당해서 소켓과 연결시켜 주는 것이다.   
  
**소켓의 구현 과정**   
소켓은 <그림 3>과 같은 일련의 과정을 거쳐 작업이 진행된다. 먼저 서버측에서는 소켓을 생성하고 그 소켓을 특정 포트 번호와 연결(bind)시킨다. 그리고 상대방으로 연결이 오기를 허락하는 듣기(listen) 작업을 수행한다. 그러다가 클라이언트가 접속을 하게 되면 서버는 이를 받아들이고(accept) 새로운 소켓을 만들어서 그 새로운 소켓이 계속 통신을 담당하게 하고 자신은 다시 듣기(lisetn)상태로 들어간다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-3.jpg |
| <그림 3> 소켓의 구현 과정 |

그런데 이 때 한 가지 주의할 것이 있다. 하나의 포트에는 한 개의 소켓만 bind할 수 있다는 것이다. 여기서 조심해야 할 것이 bind라는 말이다. 하나의 포트에 여러 개의 소켓이 있을 수는 있지만 bind는 오직 한 개만 된다. 하나의 포트에 두 개의 소켓을 bind하려 하면 에러가 나면서 bind가 실패하게 된다.   
  
그럼 왜 bind는 하나만 되는 것일까? 그 이유는 앞에서 보았듯이 데이터를 구분할 방법이 없기 때문이다. 데이터를 구분할 때 TCP/IP 헤더를 보고 구분한다고 했다. 그런데 하나의 포트에 두 개 이상의 소켓이 bind되면 이들 데이터를 구분할 방법이 없는 것이다.   
  
예를 들어 <그림 2>에서 80번 포트에 HTTP와 FTP용 소켓 두 개를 bind시켰다고 해보자. 그러면 서버는 포트로 들어오는 패킷의 TCP/IP 헤더 정보를 보고 데이터를 구분하는데 그 헤더에는 IP와 포트 번호밖에 없다. 그래서 이 패킷이 HTTP용인지 FTP용인지 구분할 방법이 없는 것이다. 그래서 하나의 포트번호에는 하나의 소켓만 bind할 수 있다. 그러면 이제 실제로 간단한 소켓 통신 프로그램을 만들어 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 5> 서버소켓 예제 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | [STAThread]  static void Main(string[] args)  {       Socket listeningSocket = new Socket(AddressFamily.InterNetwork,            SocketType.Stream, ProtocolType.Tcp);        IPAddress hostadd = Dns.Resolve(“localhost”).AddressList[0];       IPEndPoint EPhost = new IPEndPoint(hostadd, 7000);        listeningSocket.Bind(EPhost);        listeningSocket.Listen( 10 );       Console.WriteLine(listeningSocket.LocalEndPoint +            “에서 접속을 listening하고 있습니다.”);       Socket newSocket;        while(true)       {            newSocket = listeningSocket.Accept(); // blocking                       Console.WriteLine(newSocket.RemoteEndPoint.ToString() +            “에서 접속하였습니다.”);       byte[] msg = Encoding.Default.GetBytes(“접속해 주셔서 감사합니다.”);       int i = newSocket.Send(msg);       }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

**간단한 소켓 예제**   
<리스트 5>는 서버 소켓 예제이다. 먼저 TCP 방식의 소켓을 생성하고 7000번 포트에 bind한 후 listen하고 있다. 그러다가 클라이언트가 접속을 하게 되면, 클라이언트의 주소를 표시해 주고 메시지를 전송해 주고 있다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 6> 클라이언트 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | [STAThread]  static void Main(string[] args)  {  Socket s = new Socket(AddressFamily.InterNetwork,  SocketType.Stream, ProtocolType.Tcp);  IPAddress hostadd = Dns.Resolve(“localhost”).AddressList[0];  IPEndPoint EPhost = new IPEndPoint(hostadd, 7000);  s.Connect(EPhost); // blocking  if ( s.Connected == true)  {  byte[] bytes = new byte[1024];  s.Receive(bytes); // blocking  Console.WriteLine(Encoding.Default.GetString(bytes));  s.Shutdown(SocketShutdown.Both);  s.Close();  }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

<리스트 6>은 클라이언트의 코드이다. 클라이언트는 특정 포트와 bind할 필요가 없으므로 connect할 때 자동으로 임의의 포트가 할당된다. 서버로의 접속이 성공하면 메시지를 받아서 화면에 표시해 준다. 그럼 이제 이들의 결과 화면을 보자.

◆ 서버 화면   
127.0.0.1:7000에서 접속을 listening하고 있습니다.   
127.0.0.1:3912에서 접속하였습니다.   
  
◆ 클라이언트 화면   
접속해 주셔서 감사합니다.

서버 화면을 보면 클라이언트 측에서는 임의의 포트 번호에 소켓을 할당해서 접속하고 있다는 것을 확인할 수 있을 것이다. 그러면 이제 정말 하나의 포트에 여러 개의 소켓이 존재하는지 보자. 먼저 클라이언트를 두 개 실행시켜 서버에 접속하도록 하자. 다음은 서버 화면이다.

127.0.0.1:7000에서 접속을 listening하고 있습니다.   
127.0.0.1:3916에서 접속하였습니다.   
127.0.0.1:3917에서 접속하였습니다.

두 개의 클라이언트를 실행시켜서 7000번 포트에 두 개의 클라이언트가 접속을 했다. 이제 netstat -a라는 명령어를 ‘명령프롬프트’창에서 입력해 네트워크 상태를 확인해 보자.

C:\>netstat -a   
  
Active Connections   
  
Proto Local Address Foreign Address State   
TCP 한용희:7000 한용희:0 LISTENING   
TCP 한용희:7000 한용희:3916 CLOSE\_WAIT   
TCP 한용희:7000 한용희:3917 CLOSE\_WAIT

앞의 화면에서 다른 부분은 생략하고, 우리가 보기를 원하는 화면만 표시를 했다. 현재 로컬 컴퓨터의 7000번 포트의 상태를 보면 listening하는 상태가 있고, 이미 연결된 두 개의 정보가 나온다. 모두 같은 7000번 포트에 연결된 것들이다. 이로써 하나의 포트에 여러 개의 소켓이 있을 수 있다는 것을 확인할 수 있을 것이다.   
  
이제 소켓에 대한 궁금증을 풀었다. 그런데 앞의 예제를 응용해서 게임 서버로 만들기에는 무리가 있다. 왜냐하면 accept할 때나 receive할 때 블러킹이 걸려서 다른 일을 하지 못하기 때문이다. 그러므로 우리가 지금껏 익혀온 비동기 호출을 이용해서 이 문제를 해결해 보자.   
  
**비동기 소켓 통신을 이용해 블러킹 해결**   
앞서 닷넷에서는 델리게이트를 따로 이용하지 않고서도 미리 준비된 Begin×××와 End×××를 이용해서 비동기 프로그래밍을 할 수 있다고 했다. 이를 이용해 앞서 만든 예제에 적용해 보자(<리스트 7>).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 7> 비동기 통신으로 작성한 서버 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class Class1  {       static void AcceptCallBack(IAsyncResult ar)       {            Socket listener = (Socket)ar.AsyncState;            Socket newSocket = listener.EndAccept( ar );                       Console.WriteLine(newSocket.RemoteEndPoint.ToString() +                 “에서 접속하였습니다.”);            byte[] msg = Encoding.Default.GetBytes(“접속해 주셔서 감사합니다.”);            int i = newSocket.Send(msg);        }  [STAThread]  static void Main(string[] args)  {       Socket listeningSocket = new Socket(AddressFamily.InterNetwork,            SocketType.Stream, ProtocolType.Tcp);        IPAddress hostadd = Dns.Resolve(“localhost”).AddressList[0];       IPEndPoint EPhost = new IPEndPoint(hostadd, 7000);       listeningSocket.Bind(EPhost);        listeningSocket.Listen( 10 );       Console.WriteLine(listeningSocket.LocalEndPoint +            “에서 접속을 listening하고 있습니다.”);        while(true)       {            IAsyncResult ar = listeningSocket.BeginAccept (                 new AsyncCallback(AcceptCallBack), listeningSocket);            // non-blocking            ar.AsyncWaitHandle.WaitOne();            }       }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

먼저 예제에서 블러킹이 되었던 accept 부분을 비동기 함수인 BeginAccept로 바꾸었을 뿐 결과는 동일하다. 만약 이 프로그램을 윈도우폼으로 만들었다면 accept할 때 윈도우가 움직이는 것을 보면 확실히 블러킹되지 않았다는 것을 확인할 수 있을 것이다. 그러나 여기서는 간결한 예제를 위해서 콘솔 프로그램으로 만들었다.   
  
<리스트 8>은 클라이언트를 비동기 방식으로 수정한 것이다. 이번에는 connect와 receive 두 개를 비동기 방식으로 만들었다. 결과는 먼저 예제와 동일하다. 이 예제들은 간단하기 때문에, 별 어려움이 없을 것이라 생각한다. 그러면 이 비동기 통신이 쓰레드 풀을 이용하는지 직접 확인해 보고 쓰레드 풀에 남아 있는 쓰레드의 갯수에 대해 알아보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 8>비동기 통신을 이용한 클라이언트 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class Class1  {       static byte[] bytes = new byte[1024];       static void ConnectCallBack(IAsyncResult ar)       {            Socket s = (Socket)ar.AsyncState;        if ( s.Connected == true)       {            s.BeginReceive(bytes, 0, bytes.Length, SocketFlags.None,                 new AsyncCallback( ReceiveCallBack) , s);            // non-blocking       }  }  static void ReceiveCallBack(IAsyncResult ar)  {       Socket s = (Socket)ar.AsyncState;        int nLength = s.EndReceive(ar);       if ( nLength > 0 ) // 0보다 작다면 접속이 끊어진 것이다.       {            Console.WriteLine(Encoding.Default.GetString( bytes ) );       }  }   [STAThread]  static void Main(string[] args)  {       Socket s = new Socket(AddressFamily.InterNetwork,            SocketType.Stream, ProtocolType.Tcp);             IPAddress hostadd = Dns.Resolve(“localhost”).AddressList[0];       IPEndPoint EPhost = new IPEndPoint(hostadd, 7000);        s.BeginConnect(EPhost, new AsyncCallback(ConnectCallBack) , s);       // non-blocking        Console.ReadLine();       s.Shutdown(SocketShutdown.Both);       s.Close();       }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

I/O completion ports   
<리스트 8>에 다음과 같은 코드를 추가해서 <리스트 9>와 같이 현재 쓰레드의 상태에 대해 알아보자. ShowThreadInfo()라는 함수를 만들었다. 이는 현재 쓰레드의 해시코드, 쓰레드 풀인지 여부, 그리고 남아있는 쓰레드 풀의 여분 갯수를 표시한다. 앞서 쓰레드 풀은 시스템에 따라 적절한 쓰레드 갯수를 유지시켜 준다고 했다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 9> 쓰레드의 상태를 알아보기 위한 코드 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | static void ShowThreadsInfo()  {       int workerThreads, completionPortThreads;             Console.WriteLine(“Thread HashCode: {0}”,            Thread.CurrentThread.GetHashCode());       Console.WriteLine(“Is Thread Pool? : {0}”,            Thread.CurrentThread.IsThreadPoolThread);        ThreadPool.GetAvailableThreads(out workerThreads,            out completionPortThreads);       Console.WriteLine(“Available Threads”);       Console.WriteLine(“WorkerThreads: {0}, CompletionPortThreads: {1}”,            workerThreads, completionPortThreads);       Console.WriteLine();  }   static void ConnectCallBack(IAsyncResult ar)  {       Socket s = (Socket)ar.AsyncState;       ShowThreadsInfo();       if ( s.Connected == true)       {       s.BeginReceive(bytes, 0, bytes.Length, SocketFlags.None,            new AsyncCallback( ReceiveCallBack) , s);            // non-blocking       }       Thread.Sleep(2000);  }   static void ReceiveCallBack(IAsyncResult ar)  {       Socket s = (Socket)ar.AsyncState;        ShowThreadsInfo();        int nLength = s.EndReceive(ar);       if ( nLength > 0 ) // 0보다 적다면 접속이 끊어진 것이다.       {            Console.WriteLine(Encoding.Default.GetString( bytes ) );       }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

기본적으로 25개가 최대인데 쓰레드 풀에서 쓰레드를 하나씩 돌릴 때마다 이 최대 수치는 줄어들게 된다. 이를 표시해 주는 함수가 GetAvailableThreads라는 함수이다. 앞에서 Connect의 콜백 함수의 경우 비동기 호출 후 바로 끝나는 것을 막기 위해서 2초간 잠시 잠을 재웠다. 어떤 결과가 나올 것인가? 그냥 생각하기로는 connect에서 쓰레드 하나 쓰고 receive에서 쓰레드 하나 쓰니 남아있는 쓰레드 갯수는 23개가 돼야 할 것이다. 과연 그럴까?

Thread HashCode : 30   
Is Thread Pool? : True   
Available Threads   
WorkerThreads : 24, CompletionPortThreads : 25   
  
Thread HashCode : 33   
Is Thread Pool? : True   
Available Threads   
WorkerThreads : 24, CompletionPortThreads : 24

쓰레드 풀인 것은 확인이 됐고 문제는 남아있는 쓰레드 개수이다. 비동기 호출인 receive를 했는데도 WorkerThread 개수가 변함이 없다. 대신 completionPortThread에서 숫자가 하나 줄었다. 왜 이런 현상이 일어나는 것일까?   
그것은 또 다른 쓰레드 풀을 사용했기 때문이다.   
  
앞에서 프로세스당 하나의 쓰레드 풀이 존재한다고 했는데 사실 하나가 더 존재한다. 그것은 바로 I/O 전용으로 또 하나의 쓰레드 풀, 즉 I/O completion 포트용 쓰레드 풀이다. 이는 I/O 작업 전용의 쓰레드 풀로서 I/O 작업을 완료했는지 안 했는지에 대한 체크를 담당하게 된다. 그럼 왜 I/O 전용 쓰레드 풀을 사용하는 것일까? 이것을 쓰는 것이 성능이 더 좋기 때문이다.   
  
그러나 이 기능을 사용하려면 Winsock에서 이 기능을 지원해야만 한다. 그래서 앞 프로그램을 윈도우 95나 윈도우 98에서 실행하면 이들 운영체제의 Winsock에는 이 기능이 없기 때문에 닷넷에서는 자동으로 I/O completion 포트용 쓰레드 풀 대신에 workerThread를 이용해서 처리를 하게 해 준다.   
  
그러나 윈도우 NT/2000/XP의 경우 Winsock2가 설치돼 있는데 이 Winsock2가 IOCP 기능을 지원하므로 별도의 IOCP용 쓰레드 풀을 가동해서 일을 처리하게 된다. 과거 비주얼 C++로 IOCP를 구현하려면 복잡하게 코딩을 해야 했으나 닷넷에서는 손쉽게 비동기 호출 중, 네트워크 I/O 관련 함수를 호출하면 자동으로 IOCP를 이용하게 되어 있어 보다 손쉽게 코딩을 할 수 있다.   
  
**Deadlocks**   
비동기 함수를 이용하는 데 있어 한 가지 주의 사항이 있다. <리스트 10>을 보자.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트 10> Deadlocks | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class ConnectionSocket  {       public void Connect()       {            IPHostEntry ipHostEntry = Dns.Resolve( “localhost”);            IPEndPoint ipEndPoint = new IPEndPoint(                 ipHostEntry.AddressList[0], 7000 );             Socket s= new Socket( ipEndPoint.AddressFamily,                 SocketType.Stream, ProtocolType.Tcp );            IAsyncResult ar = s.BeginConnect(ipEndPoint, null, null);            s.EndConnect(ar);            Console.WriteLine(“비동기 호출 완료”);       }  }   class Class1  {  [STAThread]  static void Main(string[] args)  {       for(int i=0; i < 30 ; i++)       {            ThreadPool.QueueUserWorkItem(new WaitCallback(PoolFunc) );       }       Console.WriteLine(“ThreadPool큐에 30개 적재”);       Console.ReadLine();  }  static void ShowThreadsInfo()  {       int workerThreads, completionPortThreads;        Console.WriteLine(“Thread HashCode: {0}”            ,Thread.CurrentThread.GetHashCode());       ThreadPool.GetAvailableThreads(out workerThreads,       out completionPortThreads);            Console.WriteLine(“WorkerThreads: {0},            CompletionPortThreads: {1}”,       workerThreads, completionPortThreads);       Console.WriteLine();  }   static void PoolFunc(object state)  {       ShowThreadsInfo();             ConnectionSocket connection = new ConnectionSocket();       connection.Connect();       }  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

먼저 비동기 호출을 하는 connectionSocket이라는 클래스를 만들었다고 하자. 그런데 어떤 사람이 이 클래스를 쓰면서 이를 쓰레드 풀 내에서 호출하기로 했다. 그래서 그는 30개 쓰레드를 연속으로 만들고 이를 쓰레드 풀에 적재하였다. 그리고 각 쓰레드가 비동기 호출을 하는 이 클래스를 사용하였다. 서버는 먼저 만든 서버를 그대로 이용하기로 하자. 어떤 결과가 나올 것인가? 다음 결과를 보자.

ThreadPool큐에 30개 적재   
Thread HashCode : 3   
WorkerThreads : 24, CompletionPortThreads : 25   
Thread HashCode : 18   
WorkerThreads : 23, CompletionPortThreads : 25   
  
Thread HashCode : 19   
WorkerThreads : 22, CompletionPortThreads : 25   
  
Thread HashCode : 1   
WorkerThreads : 21, CompletionPortThreads : 25   
  
Thread HashCode : 20   
WorkerThreads : 20, CompletionPortThreads : 25   
  
Thread HashCode : 21   
WorkerThreads : 19, CompletionPortThreads : 25   
Thread HashCode : 22   
WorkerThreads : 18, CompletionPortThreads : 25   
  
Thread HashCode : 23   
WorkerThreads : 17, CompletionPortThreads : 25   
  
Thread HashCode : 24   
WorkerThreads : 16, CompletionPortThreads : 25   
Thread HashCode : 25   
WorkerThreads : 15, CompletionPortThreads : 25   
  
Thread HashCode : 26   
WorkerThreads : 14, CompletionPortThreads : 25   
  
Thread HashCode : 29   
WorkerThreads : 11, CompletionPortThreads : 25   
  
Thread HashCode : 30   
WorkerThreads : 10, CompletionPortThreads : 25   
  
Thread HashCode : 31   
WorkerThreads : 9, CompletionPortThreads : 25   
  
Thread HashCode : 32   
WorkerThreads : 8, CompletionPortThreads : 25   
  
Thread HashCode : 33   
WorkerThreads : 7, CompletionPortThreads : 25   
Thread HashCode : 34   
WorkerThreads : 6, CompletionPortThreads : 25   
Thread HashCode : 35   
WorkerThreads : 5, CompletionPortThreads : 25   
  
Thread HashCode : 36   
WorkerThreads : 4, CompletionPortThreads : 25   
  
Thread HashCode : 17   
WorkerThreads : 3, CompletionPortThreads : 25   
  
Thread HashCode : 4   
WorkerThreads : 2, CompletionPortThreads : 25   
  
Thread HashCode : 5   
WorkerThreads : 1, CompletionPortThreads : 25   
  
Thread HashCode : 7   
WorkerThreads : 0, CompletionPortThreads : 25

실행을 해 보면 프로그램이 멈춰버릴 것이다. WorkerThread가 0이 되면서 프로그램이 더 이상 작동 안 하는 데드록(deadlock) 현상이 일어난다. 왜 이런 현상이 일어나는 것일까? 먼저 어떤 사용자가 쓰레드 풀을 사용하면서 30개의 쓰레드를 쓰레드 풀에 적재를 했다. 쓰레드 풀의 기본적인 최대치는 25인데 한꺼번에 30개의 쓰레드를 적재해 버린 것이다.   
  
그래서 이미 쓰레드 풀은 포화 상태가 되었다. 그런데 비동기 호출인 BeginConnect를 하려고 큐에 적재를 했는데, 이미 차지하고 있는 쓰레드 풀에서 빈 공간이 나올 기미가 안 보이는 것이다. 이미 connect 함수에서는 EndConnect 함수를 이용해 비동기 호출이 끝나기를 블럭되면서 기다리고 있는데 끝나질 않으니 한없이 기다리게 된다. 그렇다고 끝날 수도 없다. 이미 쓰레드 풀은 포화 상태이기 때문에 더 이상의 비동기 호출이 끼어들 자리가 없기 때문이다.   
  
이 문제를 해결하기 위해서는 BeginConnect라는 비동기 호출을 동기 호출 함수로 바꿔주거나 처음 쓰레드 30개를 적재할 때, 한꺼번에 적재하지 말고 비동기 함수가 실행될 여지를 남겨주기 위해서 앞의 for문에서 Thread.Sleep(1000);이라는 문장을 주어 잠시 기다려 주면 비동기 호출이 실행될 여지가 있어서 데드록이 발생하지 않는다. 이러한 현상은 일반적으로 쓰레드 풀 내의 쓰레드가 비동기 호출이 끝나기를 기다릴 때 발생한다. 그러므로 쓰레드 풀과 비동기 호출을 같이 쓸 때는 주의해야 한다.   
  
**게임 서버 소개**   
지금까지 소개한 내용을 가지고 본격적으로 온라인 게임 서버를 만들어 보겠다.   
  
네트워크 데이터 통신 방법   
플래시와 소켓 통신을 하는데, 데이터 통신 방법은 단순하게 문자열로 보내고 받는 방법을 택하였다. 원래 플래시에는 XMLSocket이라는 것을 제공한다. 이는 XML 데이터를 위한 소켓으로 데이터를 XML 방식으로 보내야만 하는 것이다. 그러나 게임과 같이 속도가 중요한 프로그램에서는 XML로 데이터를 처리하면 이를 파싱하는 데 오버헤드가 있어 바람직하지 않다. 그래서 플래시의 XMLSocket을 이용하기는 하지만 이를 파싱하지 않고 데이터를 콤마로 구분한 문자열로 보내서 쓰는 방법을 택하였다.   
  
이때 주의할 것은 플래시의 XMLSocket은 맨 마지막에 문자열의 끝임을 나타내주는 ‘\0’ 표시가 있어야 제대로 받아들인다. 그래서 서버에서 데이터를 전송할 때 데이터의 끝에 ‘\0’을 추가해 주었다. 서버에서 네트워크 데이터를 보낼 때는 보통 서버에 접속한 모든 사용자에게 데이터를 전송하는데 자기 자신을 포함하는 경우가 있고, 자기 자신을 제외한 나머지에게 데이터를 전송할 경우가 있어 브로드캐스트(broadcast) 함수를 두 가지로 만들었다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-4.jpg |
| <그림 4> 로그인 부분 |

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-5.jpg |
| <그림 5> 대기실 부분 |

사용자 처리 방법   
각 사용자마다 이를 담당하는 user 클래스를 따로 만들었다. 이 클래스의 멤버는 <리스트 11>과 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  | <리스트11> User 클래스 | | http://images.zdnet.co.kr/img/edit_table/table9_1.gif | |
| |  | | --- | | class User  {      private Socket m\_sock; // Connection to the user      private byte[] m\_byBuff = new byte[50]; // Receive data buffer      public string m\_sID; // ID 이름      public string m\_sTank; // 탱크 종류      public string m\_sTeam; // 팀 종류      public int m\_nLocation; // 방에서의 자신의 위치      public Point m\_point; // 자신의 위치  } | |
| http://images.zdnet.co.kr/img/edit_table/table9_2.gif |

각 사용자마다 자신의 네트워크 데이터를 처리할 소켓을 가지고 있고, 자신의 각종 정보를 가지고 있다. 메인에서는 이들을 arraylist로 유지해 새로운 사용자가 들어올 때마다 리스트에 추가해 준다.   
  
방 관리   
본 게임 서버에는 방이 하나밖에 없다. 최초에 들어온 사람이 방장이 되는 것이다. 이렇게 만든 이유는 간단하게 만들기 위해서이다. 본 게임 서버를 소개하는 목적이 소스를 이해하는 데 있으므로 가능한 최소한의 기능만 구현하여 소스 코드 크기를 줄였다. 아마 이 소스를 분석해 보면 쉽게 여러 개의 방도 만들 수 있을 것이다. 방이 하나밖에 없으므로 이미 게임중이면 다른 사용자가 들어오지 못하게 하였다.

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-6.jpg |
| <그림 6> 게임 시작전 초기화 부분 |

|  |
| --- |
| http://www.zdnet.co.kr/images/stories/etc/2004/08/39129847-7.jpg |
| <그림 7> 게임중 부분 |

패킷 정보   
서버와 클라이언트가 정보를 주고 받기 위해서는 서로 약속된 정보를 주고 받아야 한다. 이때 패킷의 처음 부분에는 이 패킷의 종류를 나타내는 정보를 담고, 나머지에 데이터를 담았다. 게임의 상태를 크게 세 부분으로 나눌 수 있는데 처음 로그인 부분, 대기실 부분, 게임 시작 전 초기화 부분, 게임 중 부분으로 나눌 수 있다. 이 세 부분에서 주고받는 패킷 정보는 <그림 4~7>과 같다.   
  
**플래시 MX를 이용한 게임 완성**   
이번 연재에서는 쓰레드 풀을 이용한 비동기 프로그래밍과 소켓의 개념에 대해 알아 보고 최종적으로 게임 서버를 완성했다. 크게 네 가지 주제로 다시 한 번 정리하자면, 첫 번째는 쓰레드 풀에 대한 개념이다. 쓰레드 풀은 다수의 쓰레드를 운영하는 데 있어, 많은 쓰레드를 교환하는 데 있어 생기는 오버헤드를 해결할 수 있는 것이다. 이 쓰레드 풀의 핵심 개념은 바로 재사용이다.   
  
두 번째 소켓은 한 마디로 네트워크 프로그래밍을 하는 데 있어 마치 파일을 조작하듯 좀더 쉽게 접근할 수 있도록 도와주는 도구라고 할 수 있다. 세 번째는 IO completion 포트에 대해 알아 봤다. 이는 Winsock2에서 제기된 기능으로써, 대량의 네트워크 접속을 처리하는 데 있어 쓰레드 풀을 이용하여 각각의 네트워크 접속을 효과적으로 처리하는 것을 말한다. 마지막으로는 간단한 게임 서버를 완성해 보았다. 다음 호에서는 플래시 MX를 이용해 게임의 클라이언트 부분을 완성해 볼 것이다. @