C++Builder ADO Programming (6) - Dive to TADOCommand

우리의 레밍은 지난 강의 까지 ADO에 대한 개괄적인 사항과 Connection 객체에 대해서 알아보았다. 앞으로의 진행에 기본적인 내용이긴 하지만 그래도 열심히 공부를 한 녀석이라면 "에이, 이게 뭐야 너무 약해~!"라고 지껄일 것이다. 그것이야말로 좋은 태도라고 생각한다. 사실 그것은 앞으로 다가올 온갖 고난과 시련에 대비하는 든든한 마지막 비상식량일 것이다. 자 잡설은 그만하고 Command 객체에 Dive 하자.

Connection 객체의 장에서 Connection 객체만으로도 데이터원본에 대해 명령을 수행할 수 있었다는 것을 기억하길 바란다. 예를 들면 Connection 객체만으로도 저장 프로시저를 호출할 수 있었으며 저장 프로시저들과 인자들을 주고 받을 수도 있었다. 뿐만 아니라 테이블과 View를 통해 데이터 셋을 열어 가져올 수도 있었다. 그러나 Command 객체는 '명령수행'에 특화된 객체이기 때문에 다른 객체들보다 훨씬 더 융통성 있고 확장된 방식으로 명령을 수행하게 해준다. 여기서의 명령이란 데이터원본에 대해 직접 SQL 질의문이나 저장 프로시저를 수행하거나 테이블과 View와 같은 데이터베이스 서버 객체를 이용하는 것일 수도 있다. ADO 객체모델의 문서에서든 VCL화 되어 있는 TADOCommand를 살펴보더라도 Command 객체는 Connection 객체보다 메소드나 속성이 적다는 점을 알 수 있다. Event는 아예 없다. 이는 Command 객체가 하나의 작업 --- 데이터 원본에 대해 어떠한 명령만을 수행하는 작업 --- 만을 위한 것이기 때문이다.

미리 맛보기를 계속 하자면 저장 프로시저의 경우 Connection 객체에서의 저장 프로시저 호출과 비교해서 --- 이전의 강의에서 인자가 필요한 저장 프로시저를 조잡한 스트링 조작으로 호출한 것을 기억하라! 물론 인자가 필요 없는 저장 프로시저는 그냥 호출하면 되겠지만 --- Command 객체를 이용하면 동적으로 인자들을 얻을 수 있으며 그것들을 클라이언트에게 보낼 수 있다. 이를 통해서 클라이언트가 그 인자들을 동적으로 --- 인자들의 개수나 데이터 형을 미리 알아야 할 필요가 없다 --- 표시하고, 사용자로부터 값들을 받아서 그 값들을 인자에 넣는 등의 작업이 가능하다. 또 Command 객체에 연결된 Connection 객체의 메소드 형태로 어떤 명령이나 저장 프로시저들도 호출할 수 있다. 이번 Command 객체의 강의에서 다룰 내용은 다음과 같다.

★ Command 객체와 사용법 ← 오늘 먹을 것… =ㅅ=;;
★ Parameters 컬렉션.
★ Parameter 객체
★ 다양한 저장 프로시저와 Command 객체에서의 사용법

역시 Connection 객체 때와 마찬가지로 각각의 내용을 다루는 장에서 그에 관련된 메소드와 속성들을 함께 살펴보자.

Command 객체

말 그대로이다. 어떠한 명령을 뜻하는 객체이다. Command 객체는 데이터 원본에 대해 어떠한 명령을 수행하거나 데이터 셋이나 데이터 원본이 반환하는 값을 받을 때 사용한다. 간단히 말해서 이 객체는 데이터 원본에 대해 지시할 수 있는 어떠한 하나의 명령을 대표한다고 할 수 있다. 단순하게 명령 수행의 측면에서만 본다면 Command 객체는 Connection 객체가 할 수 있는 일을 모두 할 수 있다. Connection 객체가 Transaction을 관리하고 Errors 컬렉션을 가지고 있는 반면 Command 객체는 Parameters 컬렉션을 가지고 있다. --- 첫번째 강의에서 객체모델에 대한 그림을 기억하길 바란다. 이 Parameters 컬렉션은 다음 강의에서 살펴 보겠지만 데이터 원본에 있는 저장 프로시저를 사용할 때 매우 큰 역할을 한다. 저장 프로시저가 출력인자를 사용하여 클라이언트에게 값들을 돌려주는 경우 Connection 객체로는 그 값들을 얻을 수 없다. 그러나 Command 객체를 사용하면 Parameters 컬렉션을 통해서 변경된 인자들의 값을 주고 받을 수 있으며 또한 저장 프로시저의 반환 값도 받을 수 있다. 미리 짧게 언급하는 내용이지만 나중에 이 강의의 한

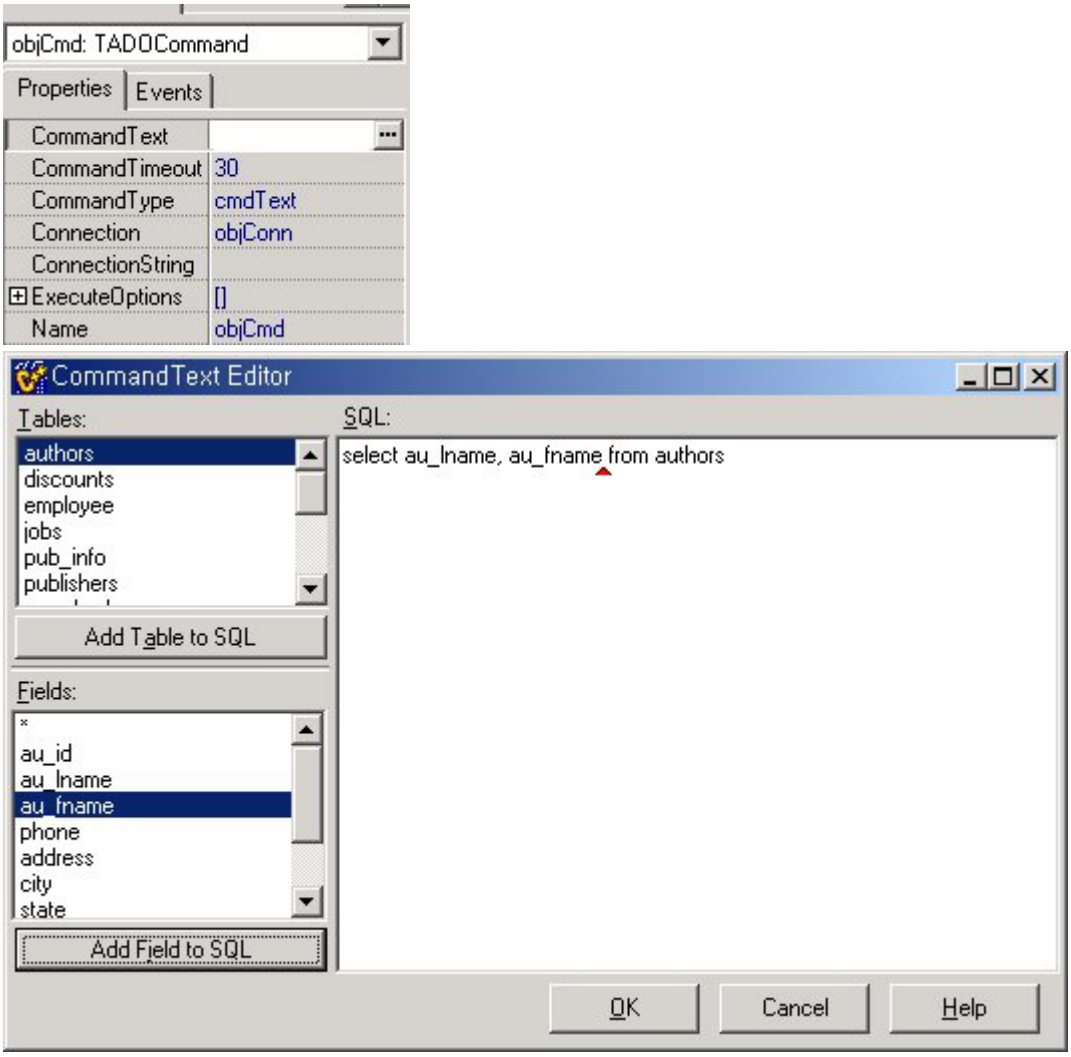
Chapter를 차지할 만큼 중요한 단절된 레코드 셋이란 개념이 있는데 이 단절된 레코드 셋을 생성하는데 Command 객체가 사용될 수 있다. 소개는 이 정도로 마치고 Main Dishes인 Command 객체의 여러 속성들과 메소드들을 맛있게 먹어 주기로 하자~ 냠냠냠~

명령의 수행

Command 객체의 명령의 수행에 직접적인 관련이 있는 속성엔 CommandText, CommandType, CommandTimeout, Prepared 속성과 Execute 메소드가 있다. 자 하나씩 해치우자.

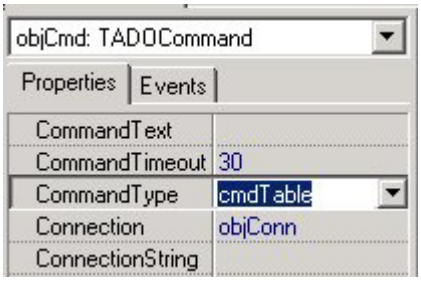
CommandText 속성

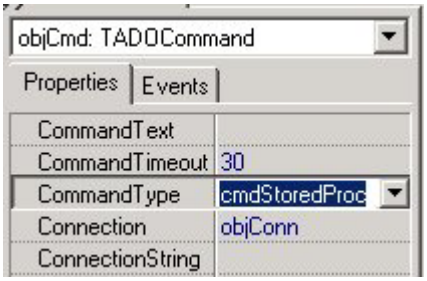
이 속성은 Connection 객체를 다룰 때 접한 속성이다. Command 객체에도 그와 동일한 속성이 있으며 수행할 명령을 뜻하는 문자열을 담는다. 그 문자열은 역시 SQL 문장일 수도 있고 저장 프로시저 이름이나 테이블, View의 이름일 수도 있으며 아니면 데이터 공급자만의 어떠한 명령어 일수도 있다. 기본값은 정해져 있지 않으며 이 속성에 담긴 명령의 종류는 CommandType 속성에 반영된다. Object Inspector의 … 버튼을 누르면 다음과 같은 CommandText Editor 창이 뜬다.

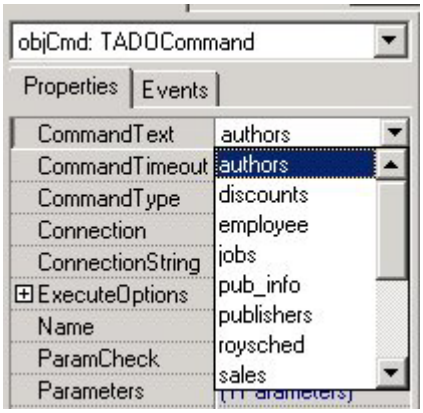


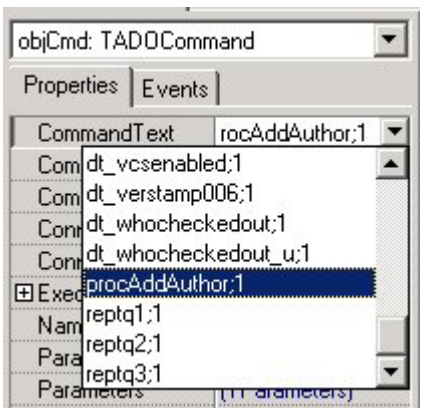
이 창이 뜨는 이유는 이 다음에 설명할 CommandType 속성의 기본값이 cmdText로 설정 되어 있기 때문이다. 그리고 그전에 Connection 객체의 인스턴스를 하나 생성하고 --- 폼이나 데이터 모듈에 TADOConnection 객체를 떨어뜨려서 --- 연결하고자 하는 데이터 원본에 대한 정보를 가진 ConnectionString을 생성한 후 Command 객체의 인스턴스의 Connection 속성에 연결하는 것이 선행되어야

한다. (Command 객체 자체의 ConnectionString 속성으로 연결 정보를 생성해도 되지만 별로 추천하고 싶지 않다) 연결정보가 맞다면 위의 그림처럼 연결 정보에 해당하는 스키마에 속하는 테이블들과 그 테이블에 해당하는 필드들이 보이고 각각 창의 아래에 있는 버튼 클릭으로 오른 쪽 SQL 편집 창에 더하거나 직접 SQL 문을 작성, 편집할 수 있다. 만약 연결 정보가 없거나 활성 연결이 없고 또 연결 정보가 잘못된 경우, 테이블들과 필드들 창에 아무 것도 나타나지 않는다. 단 SQL 문은 입력 가능하다. 이 CommandText Editor 창은 CommandType 속성의 값이 cmdText 이거나 cmdUnknown일 때 뜨며 아래의 그림과 같이 cmdTable일 때는 이 창 대신 테이블들의 리스트를 cmdStoredProc일 때는 저장 프로시저들의 리스트를 보여준다.









★ 앞으로 CommandType 속성에서 자세히 알아보겠지만 이 값 중에 cmdTableDirect와 cmdFile는 될 수 있으면 Command 객체와 함께 사용하지 말기를 추천한다. 에러를 일으키는 수도 있다. cmdTableDirect는 모든 공급자들이 지원하지는 않는다. cmdTableDirect대신 cmdTable을 사용하길 추천한다. 위 값들은 Command 객체보다 TCustomADODataSet 의 후손 클래스인 ADO DataSet들을 위한 값이다.

★ 여태껏 줄기차게 설명하였으나 이런 방법보다는 다음의 방법을 추천하고 싶다. 뭐 괜찮다, 허무의 극복은 인생의 일반적이면서도 지극히 개인적이고 고유한, 숙명적인 과제 중에 하나다. =ㅅ=;; 여러분은 이전의 ConnectionString을 다룬 강의에서 무대뽀 통박 레밍을 기억할 것이다. 통박으로 들이미는 아주 무서운 녀석이었는데 앞의 내용도 이 녀석이 한 대로 Connection 객체에 Connection 속성만 연결하고 나머지는 코드를 통해 들이미는 것이 낫다. 손가락이 아프다고? Code Complete 기능이 느리다고? 필자와 레밍이 만세를 부르고 박수를 쳐줄 것이다! =ㅅ=;;

아래의 소스는 위에서 설명한 예이다. 이 속성을 사용하는 구문으로 SQL 문, 그리고 저장 프로시저를 집어넣고 그것이 각각 SQL 문과 저장 프로시저 임을 데이터 원본에 알려주는 예이다.

```
objCmd->CommandText = WideString("SELECT * FROM AUTHORS");
objCmd->CommandType = cmdText;
objRsAuthors->Recordset = objCmd->Execute();

objCmd->CommandText = WideString("authors");
objCmd->CommandType = cmdTable;
objRsAuthors->Recordset = objCmd->Execute();

objCmd->CommandText = WideString("procAddAuthor");
objCmd->CommandType = cmdStoredProc;

// 여기서 인자들을 생성 추가하는 코드가 들어가야 되겠죠.
// 이에 관한 예제는 나중에 Parameters 컬렉션을 다룰때

objCmd->ExecuteOptions = TExecuteOptions() << eoExecuteNoRecords;
objCmd->Execute();
```

위의 소스에서 보는 것 같이 CommandText 속성과 CopmmandType 속성을 이용하여 Command 객체를 실행시킴으로 데이터 셋을 직접 가져올 수 있고 저장 프로시저를 실행시킬 수 도 있다. 자 그럼 CommandType 속성에 대해 알아보자.

CommandType 속성

앞에서 잠깐 설명한대로 이 속성은 데이터 원본에게 보낼 명령의 종류를 뜻한다. TCommandType의 열거형 값으로 다음의 값을 가진다. Connection 객체의 Recordset을 반환하는 Execute 메소드의 2 번째 인자형이기도 하다.

CommandType	의미
cmdText	기본값으로 데이터 원본에 대해 한 문장을 실행한다. 이 값이 설정되면 CommandText 속성엔 주로 SQL 문이 사용되며 인자를 가지지 않는 저장 프로시저나 View등도 사용할 수 있다.
cmdTable	CommandText 값이 테이블의 이름이 된다. 그리고 그 테이블을 연다.
cmdStoredProc	CommandText 값이 저장 프로시저의 이름이 된다. 그리고 그 저장 프로시저를 수행한다.
cmdUnknown	CommandText 값에 있는 명령 종류를 알 수 없다.
cmdTableDirect	CommandText 값이 테이블의 이름이 된다. 그 테이블을 직접 연다. 단 모든 공급자들이 지원하지는 않는다. 주로 이 값은 Command 객체를 위한 것이기 보다 ADODataSet들을 위한 CommandType 이다.
cmdFile	CommandText 값이 저장된 레코드 셋 파일 이름이 된다. 역시 위의 cmdTableDirect와 마찬 가지로 Command 객체에서 사용되기 보다 ADODataSet들을 위한 CommandType 이다.

Execute 메소드

Connection 객체의 Execute 메소드와 마찬가지로 Command 객체의 Execute 메소드 역시 데이터 원본에 대해 명령을 수행하고 데이터를 얻는 수단을 제공한다. 그러나 Connection 객체와는 달리 Command 객체를 이용하면 데이터 원본에 인자들을 전달할 수 있다. 다음은 Execute 메소드의 원형이며 3가지의 다른 원형이 있는 것을 볼 수 있다.

```
_di__Recordset __fastcall Execute();

_di__Recordset __fastcall Execute(const OleVariant &Parameters);

_di__Recordset __fastcall Execute(int &RecordsAffected,
                                  const OleVariant &Parameters);
```

첫번째 아무 인자도 가지지 않는 메소드는 위의 예제에서 보았듯이 단순한 SQL 문과 테이블들, 넘겨줄 인자가 없거나 Parameters 컬렉션으로 인자 설정이 끝난 저장 프로시저의 호출등 일반적으로 사용되는 형태이다. 나머지 메소드는 인자들을 OleVariant형의 배열 형태로 넘겨줄 때 사용된다. 2번째와 3번째 메소드의 차이는 명령의 수행결과 그 명령이 적용된 레코드의 개수를 정수형의 RecordAffected 인자에게 돌려 주는지의 여부이다.

```
CREATE PROC procAddAuthor

@au_id char(11),
@au_lname varchar(40),
@au_fname varchar(20),
@phone char(14),
@contract bit

AS

INSERT authors (au_id, au_lname, au_fname, phone, contract)
VALUES (@au_id, @au_lname, @au_fname, @phone, @contract)

GO
```

위는 이전의 Connection 객체의 강의 때 다루었던 저자들을 추가하는 저장 프로시저다. 역시 동일하게 데이터 원본인 SQL 서버에 위와 같은 저장 프로시저가 있을 때 Command 객체의 Execute 메소드로 프로시저를 호출하는 방법을 알아보자. 이전의 Connection 객체로 인자를 가지고 있는 저장 프로시저를 호출할 때는 CommandType 값이 cmdStoredProc 이 아닌 cmdText 값으로 설정한 뒤 CommandText 속성의 값인 명령문 스트링과 인자들을 스트링 조작으로 건네주고 Query Analyzer나 iSQL 같은 SQL 문 실행 에디터에서 저장 프로시저를 호출 하는 식으로 저장 프로시저 앞에 'EXEC' 이라는 명령어를 붙여 실행시켰음을 보았을 것이다. 사실 인자가 필요 없는 저장 프로시저는 거의 쓸모가 없으므로 --- 그리고 대부분의 어플리케이션은 조건과 인자를 갖춘 데이터 조작이 많이 쓰인다 --- 인자가 필요한 저장 프로시저를 두 객체를 통해 호출하는 방법을 각각 비교해 보고 알아두자. --- 사실 Command 객체가 필요 없는 경우 어플리케이션 내에서 실행파일의 크기도 크기거니와 꼭 쓸 필요가 없다. 양쪽을 알아두면 그런 경우에 유용할 것이다. 같은 맥락에서 ADO 탭에는 저장 프로시저만을 전문적으로 다루는 TCustomADODataSet의 후손 클래스인 TADOSoredProc Component가 있는데 나중에 살펴보도록 하자. 그러나 필자의 경우 Command 객체를 선호한다.

```
objCmd->Connection = objConn;

objCmd->CommandText = WideString("procAddAuthor");
objCmd->CommandType = cmdStoredProc;
objCmd->ExecuteOptions = TExecuteOptions() << eoExecuteNoRecords;

int recCount, lpBound[2] = {0,4};

Variant Params = VarArrayCreate(lpBound, 1, varVariant);

Params.PutElement("564-45-6457", 0);
Params.PutElement("Reisdorph", 1);
Params.PutElement("Kent", 2);
Params.PutElement("02-991-1988", 3);
Params.PutElement(1, 4);

objCmd->Execute(recCount, Params);

ShowMessage(IntToStr(recCount) + "명이 등록되었습니다!");
```

Execute 메소드 첫번째 형은 CommandText 속성과 CommandType 속성에서 알아보았으므로 3번째 메소드 형을 다루어 보았다. 2번째 메소드는 3번째 메소드의 첫번째 인자를 생략하면 되므로 3번째 메소드와 거의 같다고 생각하면 된다. 굳이 적용된 레코드의 개수를 알고싶지 않으면 인자가 있고 그것을 간단한 방식으로 넘겨줄 경우의 저장 프로시저를 호출할 경우 2번째 메소드 형을 사용하는 것을 추천한다. 여기서 중요하게 봐두어야 하는 것은 2번째와 3번째 Execute 메소드의 OleVariant 형의 인자인 Parameters이다. 이 인자는 데이터 원본에 있는 저장 프로시저를 호출할 때 인자들을 건네주는 역할을 하는데 저장 프로시저가 원하는 인자들의 타입이 각각 다르기 때문에 항상 Variant 형의 배열이 되어야 한다. 다행히도 VCL의 Variant 형과 클래스는 아주 막강하므로 간단히 해결할 수 있다. 위의 예제 소스도 그것을 보여주고 있는데 만약 인자가 필요하지 않다면 --- 그럴 경우는 거의 없을 것이다 --- Connection 객체의 OpenSchema 메소드를 다룰 때처럼 EmptyParam을 넘겨준다. 인자들의 Variant 배열을 생성해서 값을 채우고 2, 3 번째 Execute 메소드를 호출하는 방법은 위에서도 언급했듯이 Command 객체의 Parameters 컬렉션을 이용해서 인자들을 설정한 뒤 첫번째 Execute 메소드를 호출하는 방법과 같다. 그러나 Parameters 인자가 Parameters 컬렉션보다 우선한다. 즉 Parameters 인자를 지정한 경우 인자에 지정된 값들은 Parameters 컬렉션에 있는 인자 값들보다 우선한다.

다음 필자의 개 농장 소스의 일부를 보면 저장 프로시저가 아니라 CommandType 속성을 cmdText로 설정한 후 CommandText 속성에 참 친근하고도 즐겁고 익숙한 TQuery Component의 SQL 속성에서 쓰던 인자 달린 SQL 문을 날리는 것을 볼 수 있다. 그러나 필자는 이런 방식을 좋아하지는 않으며 여러분은 가능성만을 보기 바란다.

```
TStringList *SalSaengBu = new TStringList();
TStringList *GogiGeunSu = new TStringList();
// 살생부 리스트를 얻는다. 개고기 근수를 잰다.

String GaeDosal;
GaeDosal  = "delete from dogs where dogName = :SalSaengBu ";
GaeDosal += "and dogWeight = :GogiGeunSu";

objCmd->Connection = objConn; // 개농장 커넥션
objCmd->CommandText = WideString(GaeDosal);
objCmd->CommandType = cmdText;
objCmd->ExecuteOptions = TExecuteOptions() << eoExecuteNoRecords;
objCmd->Prepared = true;

int lpBound[2] = {0,1};
Variant Params = VarArrayCreate(lpBound, 1, varVariant);

for ( int i = 0 ; i < SalSaengBu->Count ; i++ )
{
  Params.PutElement(SalSaengBu->Strings[i], 0);
  Params.PutElement(GogiGeunSu->Strings[i].ToInt(), 1);
  objCmd->Execute(Params);
  // .... 이하 마리수와 개고기 근수 합계를 잰다. =ㅅ=;;
  Params.Clear();
}

ShowMessage("총 몇 마리를 도축해서 개고기 얼마를 얻었습니다! ^0^n");

delete SalSaenBu, GogiGeunSu;
```

특별한 것은 없었다. 단지 CommandType 속성이 cmdText로 바뀌고 CommandText 속성에 인자 달린 SQL 문이 들어가 있을 뿐이다. 인자 달린 SQL 문도 Execute 메소드를 이용하여 인자 값을 전달해서 명령을 실행시킬 수 있다는 것 만 알아두자.

View 역시 Command 객체로 관련 RecordSet들을 가져올 수 있다. SQL Server 나라에 다음과 같은 View가 있다고 하자. 이 View는 이름 그대로 작가 당 책이 몇 권이나 팔렸는지에 대한 총합을 알려주며 데이터를 불러들여 스트링 조작을 하는 고통스럽고 비 효율적이며 쓸데없는 코드가 들어가는 일을 덜어준다.

```
CREATE VIEW vwSalesCountPerAuthor

AS

SELECT au.au_id,
       au.au_lname + ' ' + au.au_fname as au_name,
       au.address as address1,
       au.city + ' ' + au.state + ' ' + au.zip as address2,
       SUM(sale.qty) as SalesCount

FROM authors as au

JOIN titleauthor as title ON au.au_id = title.au_id
JOIN sales as sale ON title.title_id = sale.title_id

GROUP BY au.au_id,
         au.au_lname + ' ' + au.au_fname,
         au.address,
         au.city + ' ' + au.state + ' ' + au.zip
GO
```

자 그럼 이 View를 Command 객체로 가져오자. Connection 객체 때와 마찬가지로 비교적 간단하다.

```
objCmd->Connection = objConn;

objCmd->CommandText = WideString("vwSalesCountPerAuthor");
objCmd->CommandType = cmdTable;

objRsAuthors->Recordset = objCmd->Execute();
```

★ View 의 단점중의 하나는 조건을 지정할 수 없다는 것이다. 위의 View도 조건이 없는데 --- Where 절이 없는 것을 볼 수 있다 --- 가령 책이 총 50권 이상 팔린 저자들만을 조회하는 경우가 그러하다. 이런 경우 조건에 해당하는 인자 값을 건네 받고 데이터 셋을 돌려주는 저장 프로시저나 사용자 정의 함수를 사용할 수 있다. 그리고 그 조건은 언제나 상황에 맞게 변화하므로 인자 값으로 건네 주는 것이 해결책이다. 나머지 자세한 사항은 Command 객체의 Parameters 컬렉션에 대해서 다룰 때 알아보자.

ExecuteOptions 속성

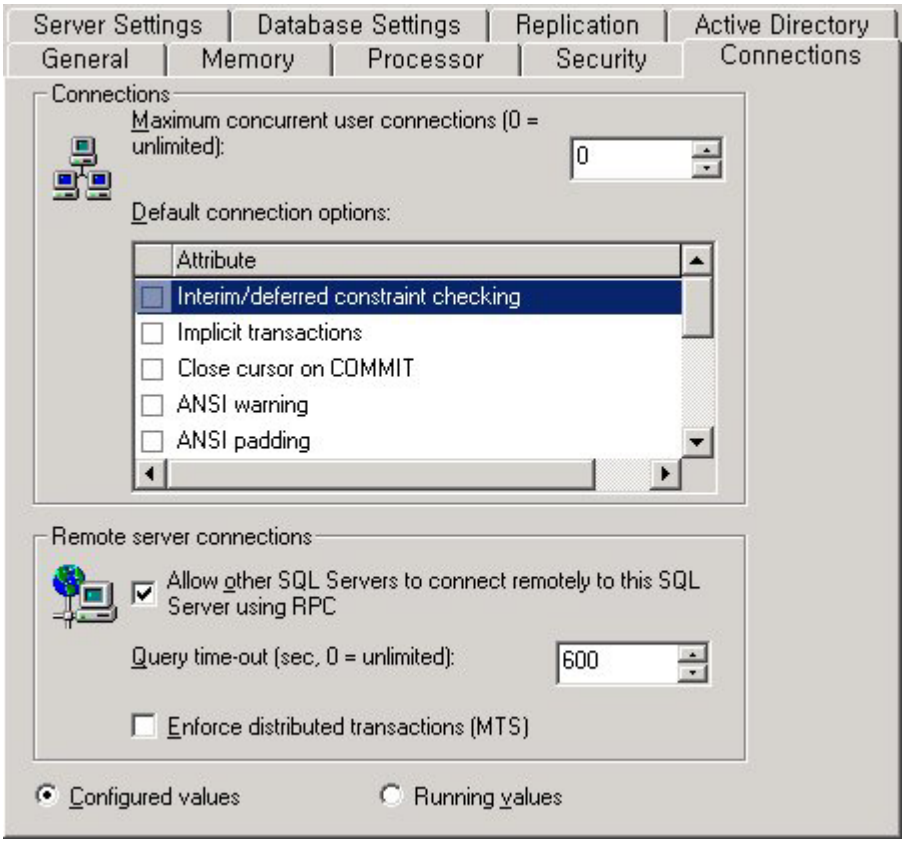
이 속성은 Connection 객체의 Execute 메소드를 다룰 때 접해 본 속성이다. TExecuteOptions 열거형으로 아래의 값을 가진다. Command 객체가 수행하는 명령의 실행 속성을 정의한다. 항상 Execute 메소드를 실행하기 전에 설정한다.

Execute Options	의미
eoAsyncExecute	명령이 비동기적으로 실행된다.
eoAsyncFetch	데이터를 비동기적으로 가져온다.
eoAsyncFetchNonBlocking	데이터를 비동기적, 비블로킹 방식으로 가져온다.
eoExecuteNoRecords	데이터를 돌려주지 않는 명령에 쓰인다.

전반적인 내용은 Connection 객체의 Execute 메소드를 살펴볼 때 다루었던 내용과 동일하다. ADO가 골 때리는 것 중 하나가 명령을 수행할 때 마다 Recordset을 생성하는데 --- 데이터 질의가 아닌 명령에 대해서는 빈 데이터 셋을 만든다 --- 이것은 서버가 비효율적이고 쓸데없는 것에 시간과 자원을 소비하는 결과를 낳는다는 것이다. 데이터 셋을 돌려 받을 필요가 없는 명령들에 한해서 반드시 이 속성에 4번째 인자인 eoExecuteNoRecords 값을 추가해 줘야 한다. 위에서 설명한 소스들을 참고하길 바란다.

CommandTimeOut 속성

이 속성 역시 Connection 객체의 해당 속성과 마찬 가지로 데이터 원본에 보낼 하나의 명령이 수행될 수 있는 제한 시간을 뜻한다. 기본 값은 30 초 이다. 이 속성은 해당 Command 객체와 연결된 Connection 객체의 ConnectionTimeOut 속성을 물려받지 않으며 그 값을 덮어 쓰지도 않는다. Connection 객체의 CommandTimeOut 속성은 그 Connection 객체의 Execute 메소드에만 적용되는 것이며 Command 객체의 CommandTimeOut 속성은 Command 객체 자체의 Execute 메소드에만 적용되는 것일 뿐이다. 그리고 이 속성을 0으로 설정하는 것은 명령을 수행할 수 있는 시간 제한을 두지 않는다는 것이다. 그러나 이 속성의 설정과는 다르게 명령의 시간 제한이 데이터 원본 자체의 설정으로 되어 있는 경우를 볼 수 있다. 다음은 SQL Server를 데이터 원본으로 할 때 명령의 시간제한을 설정하는 창이다.



데이터 베이스 이름에서 마우스 오른쪽 클릭하여 등록정보를 선택하면 된다. 이와 같이 백엔드 자체의 설정이 있을 때는 그것에 따른다. 그러나 필자의 경험으로 제한 시간을 두는 것이 낫다. 계류중인 작업이나 많은 시간을 잡아먹는 작업은 분명 서버에 부하를 주기 때문이다.

Prepared 속성

이 속성은 BDE를 사용한 데이터베이스 프로그래밍에서 많이 접해본 아주 친근하고 즐거운 속성이다. ADO의 Command 객체도 이 속성을 가지고 있는데 이 속성은 데이터 원본에 대해 어떠한 명령을 수행하기 전에 그 명령을 미리 컴파일 해 둘 것인지의 여부를 결정한다. CommandType이 cmdStoredProc이 아닌 경우 --- Command 객체가 수행하는 명령이 저장 프로시저를 호출하는 명령이 아닐 경우 --- 명령을 미리 컴파일 해 두는 것이 수행성능에 도움이 될 수 있다. 물론 명령이 처음 수행될 때는 추가적인 부담이 생기지만 그 이후부터 컴파일된 버전의 명령이 수행되는 것이므로 첫번째 보다 훨씬 더 빠르게 수행된다. 이는 위의 인자가 달린 SQL 문을 Command 객체를 통해 날리는 것을 다룬 필자의 소스에서처럼 동일한 명령을 여러 번 수행하는 경우 특히 더 유용하다.

States 속성

이 속성은 읽기 전용으로 Command 객체의 현재 상태를 뜻한다. 이 속성이 돌려줄 수 있는 값들은 Connection 객체의 State 속성과 마찬가지로 TobjectStates 열거형으로 아래의 값을 가진다.

stClosed	Command 객체가 닫히고 데이터 원본과의 연결이 종료 되었다.
StOpen	Command 객체가 데이터 원본과 연결되어 있음.
stConnecting	Command 객체가 연결중임.
stExecuting	Command 객체가 명령을 실행중임.
stFetching	Command 객체가 데이터를 가져오는 중임.

마지막 두 값인 stExecuting 과 stFetching 은 비동기적 명령이 수행되는 도중에만 나타난다. 이 속성을 이용

하면 Command 객체가 데이터 원본에 아직 연결되어 있는지를 검사할 수 있다. 만일 연결이 끊어 졌다면 Connection 속성을 다시 설정한 후 작업을 개시해야 할 것이다.

Cancel 메소드

Cancel 메소드는 Command 객체의 비동기적 작업에 관련된 것이다. 일반적으로 비 동기적 작업이란 명령이나 행위, 조작 등을 수행한 후에 그에 대한 반응이 즉시 돌아오는 동기적 작업과 달리 그 반응이 언제 원래 수행자에게 돌아올지 모르는 작업을 말한다. 진행중인 비동기적 작업이 없는 상황에서 이 메소드를 호출하면 런타임 에러가 발생한다. Connection 객체의 Cancel 메소드와 마찬가지로 명령을 비동기적으로 수행시킨 명령이나 조작들을 States 속성을 이용하여 취소 시킬 때 주로 사용할 수 있다. 다음의 의사 코드를 보자.

```
if (!objCmd->States.Contains(stOpen))
{
  objCmd->Connection = objConn;
}

try
{
  // 명령및 인자들을 설정한다. =ㅅ=;;

  objCmd->ExecuteOptions = TExecuteOptions()
                           << eoAsyncExecute << eoExecuteNoRecords;
  objCmd->Execute();
}
catch(...)
{
  String ErrorMsg = "";

  for ( int i = 0 ; i < objConn->Errors->Count ; i ++ )
  {
    ErrorMsg += "에러번호: ";
    ErrorMsg += IntToStr(objConn->Errors->Item[i]->Number);
    ErrorMsg += "₩n에러내용: ";
    ErrorMsg += objConn->Errors->Item[i]->Description + "₩n₩n";
  }

  MessageDlg(ErrorMsg, mtError, TMsgDlgButtons() << mbOK, 0);
}
```

이 코드는 명령을 비 동기적으로 실행시킬 경우 그에 해당하는 이벤트 핸들러에 해당하는 내용일 것이다. 다음은 비동기적으로 실행중인 Command 객체의 상태를 알아내 Cancel 메소드를 사용하여 그 명령을 취소하는 코드이다.

```
if ( objCmd->States.Contains(stExecuting) )
{
  objCmd->Cancel();
}
```

★ 필자의 경험으로 볼 때 될 수 있으면 데이터를 조회하는 명령은 비 동기적으로 실행시키지 말 것을 추천한다. 주 어플리케이션과 별도의 스레드에서 데이터조작이나 조회등과 거리가 먼 모니터링이나 관리 명령 등을 날려볼 것을 추천한다.

★ 비 동기적 작업은 상태 유지가 편한 클라이언트/서버 어플리케이션에서 쓰이는 것이 일반적이다. 웹 기반 어플리케이션의 경우 비 동기적 작업이 그다지 실용적이지는 않다. 왜냐하면 여러분도 다 아시다시피 웹은 근본적으로 상태가 없는 환경이기 때문이다.

이번 강의는 여기 까지 이다. Dive를 해 보았는데 뭐 처음은 시원하고 재미있었지만 나중은 Connection 객체를 다룰 때의 내용과 비슷한 부분이 많아서 재미 없었다. Command 객체의 속성과 메소드는 이것으로 마치고 다음 강의에서 Parameters 컬렉션과 Parameter 객체에 대해 알아보도록 하자. 다음에는 해수욕을 그만하고 잠수도 한번 해보자. 아직도 우리를 기다리는 예쁜 물고기와 조개들은 많이 있다. =ㅅ=;;

Mortalpain