C++Builder ADO Programming (5) - ADO Transaction, Errors Collections, Connection Events

지난번 강의에서 우리의 레밍은 TADOConnection의 여러 가지 속성들과 메소드들을 익히고 그것을 사용해서 SQL 문도 실행시키고 저장 프로시저도 호출해 보았다. 그것은 그것 나름대로의 한 방법이며 예제일 것이다. 필자는 개인적으로 아케이드 격투 게임을 상당히 좋아하는데 --- The King of Fighters를 좋아하는데 (철권은 못한다. 그러나 3D 격투 게임 중에 Dead or Alive는 잘한다. ^ㅅ^) C++ Builder와 더불어 필자의 훌륭한 장난감 중에 하나다. (장난감 2호, Builder는 장난감 3호. 아직 Psycho Soldier Team을 나보다 잘하는 사람을 만나지 못했다. 집에 플레이스테이션이 있지만 때때로 오락실에 가서 초, 중, 고, 대딩들을 혼내주며 꼭 엔딩을 본다~ =ㅅ=;; 음 KOF2000을 마치고 SNK가 망해서 2001을 국내에서 제작했다고 하는데 금년에도 엄청 기다려진다.) 1편이 나온 94년부터 해서 근 8년을 해 오고 있는데 C++ Builder와 마찬가지로 질리지가 않는다. =ㅅ=;; --- 거기엔 연속기와 필살기란 것이 있어서 단번에 적을 제압하는 기술이 있다. 하지만 이 무적의 연속기나 필살기도 알고 보면 버튼 하나를 누르는 작은 동작들이 모여서 되는 것으로 우리의 레밍도 이런 한 예제, 한 방법씩 익혀가면서 연속 필살기를 익혀서 ADO Programming을 끝냈으면 한다. 오늘은 잡설이 너무 기네~ =ㅅ=;;

이번 강의에서는 지난 강의에서 말한 대로 ADO 트랜잭션, Connection 객체의 중요 이벤트, Errors 컬렉션과 Error 처리를 알아보고 간단한 요약을 통해서 길었던 Connection 객체의 장을 마치도록 하자.

(음 빨리 Kensu와 Bao가 Psycho Ball을 자유롭게 쓸 수 있어야 되는데…. =ㅅ=;;)

ADO 트랜잭션

트랜잭션이란 의미는 추상적인 용어로 '원소성(Atomicity)'에 대한 것인데 이것은 어떤 것이 하나의 단위로서 작동해야 한다는 개념이다. 데이터베이스 적인 관점에서 원소성은 전부 아니면 전무로 간주되는 가장 작은 명령문 집합을 의미한다.

Connection 객체에는 트랜잭션을 지원하기 위한 여러 가지 메소드와 속성들이 포함되어 있다. 이 말은 Connection 객체에 대한 트랜잭션을 시작하려면 데이터 공급자 자체가 트랜잭션을 지원해야 한다는 의미이다. 간략하게 알아본다면 BeginTrans 메소드로 트랜잭션을 시작하고 데이터 공급자에 따라 다르지만 하나의 트랜잭션이 끝나기 전에 또 다른 트랜잭션을 시작하는 방식으로 트랜잭션들을 내포시킬 수도 있다. 각각의 트랜잭션들은 RollbackTrans 메소드에 의해 철회될 수도 있고 CommitTrans에 의해 완수될 수도 있다. 그리고 Attributes 속성에 설정된 값에 따라서 앞의 두 메소드에 의해 트랜잭션이 끝났을 때 또 다른 트랜잭션이 자동적으로 시작될 수도 있다. 또한 IsolationLevel 속성을 통해 트랜잭션들이 상호 작용하는 방식도 설정할 수 있다. 어떤 데이터 공급자들은 트랜잭션을 아예 지원하지 않으며 또 어떤 것들은 트랜잭션을 지원하긴 하지만 ODBC처럼 내포된 트랜잭션을 지원하지 않고 SQL 7.0 같은 공급자는 내포된 트랜잭션은 지원하지만 가장 외부의 트랜잭션이 호출되기 전 까지는 내부 트랜잭션의 완수를 허용하지 않는다. 자 그럼 트랜잭션에 관련된 메소드들과 속성들을 살펴보자.

InTransaction 속성

bool 값 읽기 전용 속성으로 트랜잭션이 진행 중인지를 나타낸다. 이 속성으로 실행 중에 TADOConnection Component에 의해 현재 트랜잭션이 활성화 되었는지를 결정할 수 있다.
BeginTrans 메소드가 호출되면 true로 되고 CommitTrans, RollbackTrans 메소드가 호출된 후 false로 바뀐다.

BeginTrans 메소드

BeginTrans 메소드는 Connection 객체에 대한 트랜잭션을 시작한다. 그 연결을 사용하는 데이터에 대한 모든 변경들은 트랜잭션이 완수되거나 철회 되기 전까지 버퍼에 저장된다. 이 메소드가 성공적으로 수행되면 OnBeginTransComplete Event가 발생하며 InTransaction 속성이 true로 설정된다. 위의 사항들은 반드시 연결이 열려있는 상태에서 일어난다는 것은 당연한 이야기이다.

RollbackTrans 메소드

RollbackTrans 메소드는 트랜잭션을 철회하며 그렇게 하면 현재 트랜잭션 도중에 생겼던 변경들이 모두 폐기되며 현재 트랜잭션이 끝난다. 이 메소드가 성공적으로 수행되면 OnRollbackTransComplete Event가 발생하며 InTransaction 속성이 false로 설정된다.

CommitTrans 메소드

CommitTrans 메소드는 트랜잭션을 완수하며 그렇게 하면 현재 트랜잭션 내에서 생긴 변경들이 실제로 저장되며 현재 트랜잭션이 끝난다. 이 메소드가 성공적으로 수행되면 OnCommitTransComplete Event가 발생하며 InTransaction 속성이 false로 설정된다.

위에서 말한 내용을 간단하게 의사 코드를 통해 알아보도록 하자.

```
// 트랜잭션에 대한 속성들을 설정한다.
if ( !objConn->InTransaction )
  objConn->BeginTrans();
else
{
  objConn->CommitTrans();
}
try
{
  // 데이타 작업을 수행한다.
  objConn->CommitTrans();
}
catch (...)
{
  objConn->RollbackTrans();
}
```

위의 의사 코드를 통해 볼 수 있듯이 BeginTrans 메소드는 반드시 그에 해당하는 CommitTrans나 RollbackTrans 메소드로 끝을 맺는다. VCL과 C++ Builder는 단일 트랜잭션만을 지원하며 이것이 일반적인 방법이다. 때때로 공급자 중에서 트랜잭션 내포를 지원하는 공급자들이 존재하긴 하지만 트랜잭션 내포가 일반적으로 쓰이지는 않는다.

Attributes 속성

이 속성은 한 트랜잭션이 완수되거나 철회된 후에 자동적으로 다른 트랜잭션을 시작할 것인지의 여부를 결정한다. 이 속성은 Default 값으로 어떤 값도 가지지 않는데 TXactAttribute 열거형 값을 가질 수 있다.

TXactAttribute Options	의미
xaCommitRetaining	CommitTrans 메소드에 의해 트랜잭션이 완수된 후 자동적으로

	새로운 트랜잭션을 시작한다.
xaAbortRetaining	RollbackTrans 메소드에 의해 트랜잭션이 철회된 후 자동적으로 새로운 트랜잭션을 시작한다.

그러나 모든 공급자들이 이 기능을 지원하지는 않는다는 점을 주의해야 한다. 또한 SQL 서버와 같은 공급자는 이미 백엔드 쪽에서 자체적으로 이러한 형태의 자동적인 트랜잭션 시작기능을 사용하고 있을 수도 있다. 이 속성의 값을 바꾸면 서버쪽에 미리 지정되어 있던 설정들을 덮어쓰게 된다. 필자의 경험으로 볼 때 이런 자동적이고 암묵적인 트랜잭션 보다는 위의 의사코드에서 보여지는 명시적이고 구체적인 트랜잭션을 사용하는 것을 추천한다.

IsolationLevel 속성

이 속성은 주어진 연결에 대한 트랜잭션 격리 수준을 뜻한다. 트랜잭션 격리 수준은 트랜잭션들이 서로 상호작용하는 방식을 결정한다. 이 속성의 값들은 TIsolationLevel 열거형 값들을 가지며 나중에 TADODataSet을 다룰 때 TADOLockType 열거형과 비슷하다. 이 값들은 아래의 표와 같다.

Isolation Level	의미
ilUnspecified	격리수준을 결정할 수 없거나 지정한 수준을 사용할 수 없을 때 이 값이 설정된다.
ilChaos	다른 사용자가 시작한 트랜잭션에 의해 생긴 변경을 덮어 쓰지 못하게 한다.
ilReadUncommitted	데이터에 대한 완수되지 않은 변경들을 볼 수 있게 한다. 다른 말로 하면 한 트랜잭션에 의해 독점적으로 잠겨진 데이터를 그 트랜잭션이 진행중인 상태에서도 다른 트랜잭션에서 볼 수 있다는 뜻이다. 이처럼 아직 완수되지도 않은 데이터를 다른 곳에서 읽는 것을 Dirty Read라고 한다.
ilBrowse	ilReadUncommitted와 동일
ilCursorStability	한 트랜잭션이 독점적으로 데이터를 잠근 경우 다른 트랜잭션들은 잠금이 해제되기 전까지는 그 데이터에 접근할 수 없게 한다.
ilReadCommitted	ilCursorStability와 동일
ilRepeatableRead	데이터가 재질의 되기 전까지는 다른 트랜잭션들에 의해 생긴 변경들을 볼 수 없다.
ilIsolated	트랜잭션이 다른 트랜잭션들과 완전하게 격리된다. 한 트랜잭션의 범위 동안에는 초기 트랜잭션이 사용한 잠금의 종류에 상관 없이 어떠한 종류의 잠금이라도 다른 트랜잭션들이 데이터에 접근하지 못하도록 막는다.
ilSeriallizable	ilIsolated와 동일

트랜잭션의 상호 작용하는 방식인 IsolationLevel 속성은 Recordset의 3대 속성 --- CursorType, CursorLocation, LockType를 말하는 것으로 데이터베이스 프로그래밍에 가장 중요한 기본 개념중의 하나로 이후의 TADODataSet을 다룬 강의에서 자세히 알아볼 것이다. --- 중에 하나인 LockType 속성과 밀접한 관계가 있다는 것만 일단 알아두자..

예를 들어 현재 잠금 상태에 상관 없이 어떤 데이터를 읽을 수 있어야 한다고 하면 IsolationLevel을 다음과 같이 설정하면 된다.

```
String dogString, sql;
dogString = "개 농장 디비에 연결하는 커넥스트링";

objConn->ConnectionString = WideString(dogString);
objConn->CursorLocation = clUseClient;
objConn->IsolationLevel = ilReadUncommitted;
// 기타 다른 설정을 한다.....

objConn->Open();

objConn->DefaultDatabase = "DogFarm";

sql = "select * from dogs where alias = '파트라슈'";

objRsDogs->Recordset = objConn->Execute(WideString(sql),
                                        cmdText,
                                        TExecuteOptions());
while ( !objRsDogs->Eof )
{
  DogsList->Items->Add(objRsDogs->FieldByName("NAME")->AsString);
  // 파트라슈를 데리고 지지고 볶고 데리고 논다.... =ㅅ=;;
  objRsDogs->Next();
}
```

위의 소스는 이전 강의의 내용도 다루고 있는데 특히 Connection 객체로 Recordset을 생성할 수 있는 예제이다. SQL문이 텍스트 형태의 명령이므로 Execute 메소드의 2번째 인자로 cmdText가 사용되었으며 특히 주의 깊게 볼 것은 TExecuteOptions 열거형에 아무것도 지정되지 않았다는 것이다. --- select 질의 문으로 데이터를 가져오는 명령이기 때문에 eoExecuteNoRecords 가 사용되지 않았고 eoAsyncExecute를 사용할 경우 비동기적으로 데이터 셋을 가져오기 때문에 에러를 일으킬 수도 있다. 그리고 Connection 객체의 CursorLocation 속성이 clUseClient이므로 이 객체를 통해 생성되는 Recordset 객체의 CursorLocation 속성도 clUseClient를 따르게 된다. 아무튼 여기서 중요한 것은 분리 수준을 위와 같이 지정하면 읽고자 하는 데이터의 상태에 상관 없이 그 데이터의 값을 가져올 수 있게 된다. 만일 어떤 개 한 마리가 몸무게가 70Kg이 넘어 파트라슈 급이 되어서 개 사육 마녀가 이 개의 구분을 '벤지' 에서 '파트라슈' 라고 바꾸는 도중에 있다면 우리의 레밍이 위의 코드로 얻은 레코드 셋에는 변경된 값인 '파트라슈' 가 들어가게 된다. 그러나 개 사육 마녀가 몸무게를 잘 못 재서 이후에 변경을 철회하게 된다면 우리의 레밍이 얻은 레코드셋에는 잘못된 값이 들어 있는 결과가 된다.

잠깐 여기서 필자의 개발 경험에 비추어서 각 상황에 어떤 트랜잭션을 쓸 것인지에 대한 문제에 대한 해결책을 추천하고 넘어가고자 한다. 일반적으로 트랜잭션을 사용할 때 생기는 2 가지 문제를 살펴보자. 그 2가지 문제란 다음과 같다.

★ 비반복적 읽기 (NonRepeatable Read) – 트랜잭션 a 가 레코드 하나를 읽는다. 트랜잭션 b 가 그 레코드를 갱신하거나 삭제하고 트랜잭션을 완수한다. 트랜잭션 a 가 그 레코드를 다시 읽으려고 하면 이미 삭제되었거나 변경되었기 때문에 올바른 결과를 얻을 수 없다.

★ 팬텀 (phantom – 허깨비, 유령) – 트랜잭션 a 가 어떤 조건을 만족하는 레코드 셋을 얻는다. 트랜잭션 b 가 동일한 조건을 만족하는 레코드 셋에 레코드를 하나 삽입한다. 트랜잭션 a 가 레코드들을 읽는 명령문을 다시 수행하면 이전과는 다른 --- 즉 허깨비 레코드가 끼어든 --- 레코드셋을 얻게 된다.

TADOConnection의 기본적인 격리 수준은 ilCursorStability로 --- 위의 표에서 ilReadCommitted와 같다고 설명했다. --- 이것은 BDE를 사용할 때 TDataBase Component의 기본적인 트랜잭션 격리 수준과 같다. 단지 이름만 다를 뿐이다. (TransIsolation 속성이며 값은 tiReadCommitted이다.) 이 기본 값에 대한 설명은 표

에 나와 있으며 다르게 말하면 완수되었으면 읽기 가능이라는 것이다. 이 수준에서 데이터를 읽는 클라이언트는 다른 모든 완수된 트랜잭션에 의해서 일어난 변경 모두를 볼 수 있다. 이러한 수준의 격리는 속도가 매우 빠르긴 하지만 위에서 말한 2가지 문제를 방지할 수 없다는 단점을 가지고 있다. 따라서 이 수준은 읽기가 많고 쓰기가 적은 상황에서는 적합하지 않다. 그보다 느리기는 하지만 비교적 안전한 격리 수준은 ilRepeatableRead – 반복적 읽기 – 이다. 이 수준에서 일어나는 한 트랜잭션이 이미 읽은 데이터를 다른 트랜잭션들이 변경했다고 해도 그 트랜잭션의 데이터는 변경되지 않는다. 따라서 첫번째 문제인 비반복적 읽기 문제는 발생하지 않는다. 그러나 허깨비 유령 데이터는 발생할 수 있다. 가장 느린 반면 가장 안전한 격리 수준은 ilSerializable --- 일렬화 기능, ilisolated와 동일하다 --- 이다. 이 수준에서는 모든 병행적인 트랜잭션들이 하나씩 차례대로 수행되는 것과 같은 효과가 발생한다. 즉 완전히 격리되는 것이다. 따라서 위의 2가지 문제는 발생하지 않는다.

ilRepeatableRead와 ilSerializable은 잠금을 이용해서 트랜잭션들을 격리시킨다. 이런 잠금 방식은 낙관적인 방식과 비관적인 방식이 있는데 이것은 앞에서 말했듯이 TADODataSet --- Recordset --- 을 다룬 강의에서 자세하게 살펴보도록 하자.

음 만약 여러분이 데이터베이스 서버 프로그래밍에 익숙하다면 개별 명령을 기반으로 격리 수준을 설정할 수도 있다. SQL Server 나라에 다음과 같은 데이터베이스 객체인 View가 있다고 하자. --- 이 View는 멍멍멍 View로 멍멍멍 테이블에서 오늘 개 농장에 들어온 개새끼들의 필수 자료들을 조회한다.

```
CREATE VIEW vwMungMungMung

AS

SELECT dogID,
       dogName,
       dogAlias,
       dogMaster

FROM tbMungMungMung (NOLOCK)

WHERE dogInDate = convert(varchar(10), getdate(), 120)

GO
```

그리고 다음은 위의 멍멍멍 View를 Connection 객체를 통해 레코드 셋에 들이붓는 소스코드중의 일부이다.

```
objConn->ConnectionString = "개 농장 연결 스트링";
objConn->CursorLocation = clUseClient;
objConn->IsolationLevel = ilSerializable;

// 기타 다른 설정을 한다.....

objConn->Open();

objConn->DefaultDatabase = "DogFarm";

objRsTodayDogs->Recordset =
                objConn->Execute(WideString("vwMungMungMung"),
                                 cmdTable,
                                 TExecuteOptions());
while ( !objRsTodayDogs->Eof )
{
  Ls->Items->Add(objRsTodayDogs->FieldByName("dogName")->AsString);
  // 추출된 개를 지지고 볶는다.... =ㅅ=;;
  objRsTodayDogs->Next();
}
```

Connection 객체를 통해 SQL 문을 날려 데이터 셋을 얻는 이전 예제와 잘 비교해 보기 바란다. 차이점은 View는 데이터 베이스 객체라는 점이며 문법 체크와 컴파일이 이미 완료된 상태에서 메모리 캐시 내에 저장되어 있을 수 있기 때문에 직접 날리는 SQL 문보다 수행 속도가 빠르다. Execute 메소드의 CommandText 인자가 View 이름으로 바뀐 것과 2 번째 인자인 CommandType 인자가 cmdTable로 바뀐 것도 봐두길 바란다. 이것은 View를 Table로 볼 수 있다는 말이며 그냥 단순하게 테이블 하나씩을 불러오거나 TADOTable Component를 사용하는 경우 보다 View를 사용하는 이런 방식의 이점은

1. 원하는 필드만을 가져올 수 있다. 테이블을 불러올 경우 원하지 않는 필드까지 불러와야 한다. 이것은 추가적인 네트웍 부담의 문제이다. 또 최종 사용자에게 보여지는 데이터베이스의 복잡성을 줄인다. 비슷하게 작업에 필요한 데이터에 대한 접근을 허용하면서도 민감한 정보를 담은 열들이 선택되지 않도록 숨긴다.
2. 2 개 이상의 테이블을 조인하는 View를 만들 수 있다. --- 단 조건을 입력 받거나 그 조건이 가변 조건일 경우는 예외이다. 조건을 입력 받는 경우는 레코드 셋을 반환 하는 저장 프로시저를 사용할 수 있다. 이것은 이후의 강의에서 살펴본다 --- 이 경우 달랑 테이블 하나보다 자료를 처리하는 데 더 많은 융통성과 운용성을 제공하며 조인의 과정이 미리 선행되어 있으므로 해서 성능 상의 이점이 생긴다.

또 TADOQuery Component 상에서 SQL문을 사용해서 직접 레코드 셋을 얻을 때 보다 얻는 장점은 SQL 문이 문법체크와 컴파일이 이미 완료된 상태이므로 속도나 성능상의 이점을 가질 수 있다는 것이다.

다시 본론으로 돌아와 위의 View를 보면 nolock option이 있는 것을 볼 수 있을 것이다. 만일 여러분이 Database Server Programming(업계에선 PL/SQL이라 통하는데 아마 필자의 말이 정식용어로 더 맞을 것이다)에 능숙하다면 무엇인지 알 것이다. 잠깐 설명하자면 nolock option은 SQL 문을 질의할 때 어떤 잠금도 하지 않겠다는 최적화 지시자이다. 그런데 가만 뒤의 예제 소스에서 Connection 객체의 격리 수준은 ilSerializable로 격리 수준과 잠금 수준이 서로 다르게 된다. 이때의 nolock option은 Connection 객체에 설정된 격리수준 보다 우선한다. 이러한 명령별 격리 수준 기능은 사실 데이터 원본의 고유한 기능이다. 기존의 트랜잭션 격리 수준을 덮어쓰려면 위의 예처럼 최적화 지시자를 명시적으로 지정해 주어야 한다. 그렇지 않으면 IsolationLevel 속성의 값에 해당하는 최적화 지시자가 묵시적으로 사용된다 라는 것을 의미한다. 예를 들면 위의 예처럼 만약 ilSerializable에 해당하는 격리 수준을 얻으려면 nolock 대신 holdback 최적화 지시자를 써야 한다. 따라서 위의 예의 경우 nolock이 없다면 가장 안전한 격리 수준인 ilSerializable이 적용되겠지만 실제로 적용되는 격리 수준은 그것이 아니므로 우리의 레밍이 일단 View로 얻은 데이터를 보고 있는 동안에 마녀가 새로운 개새끼들의 정보를 고치고 있거나 늦게 들어온 개새끼들을 추가했거나 하는 완수되

지 않은 데이터를 읽을 수 있다

★ 위의 트랜잭션에 대한 내용들은 근본적으로 Multi-tier 층을 가지는 엔터프라이즈 어플리케이션에 적합한 사항이 아님을 명심하기 바란다. 위의 제반 설명들은 C/S 방식의 어플리케이션이나 솔루션 내에서 다루는 트랜잭션에 대한 사항이다. 필자의 경험상 multi-tier 층에서는 기반 클라이언트 어플리케이션(프리젠테이션 층을 의미함)에서가 아니라 중간의 비즈니스 층(흔히 말하는 어플리케이션 서버 층이다)이나 각종 데이터베이스 층 상에서 트랜잭션을 처리하는 것이 성능상이나 유지 보수적인 각종 비용에 있어서 훨씬 낫다고 생각한다. --- COM 이나 MTS 상에서(Windows 2000에서 이 2가지는 COM+로 통합되었다) ADO 데이터 층을 구축하는 것이 골자이다. 필자는 그에 대한 내용도 앞으로의 강의에서 다룰 것이다.

트랜잭션 이벤트들

이 이벤트들은 트랜잭션이 시작되거나 완수되거나 철회되면 발생한다.

OnBeginTransComplete

Connection 객체의 BeginTrans 메소드가 성공적으로 수행된 후에 발생하는 것으로 ADO에게 트랜잭션이 시작되었으며 이제부터 만들어지는 모든 변경들은 버퍼에 저장되어야 함을 알려주는 역할을 한다. 이벤트 핸들러는 다음과 같다.

```
typedef void __fastcall (__closure *TBeginTransCompleteEvent)
                        (TADOConnection* Connection,
                         int TransactionLevel,
                         const _di_Error Error,
                         TEventStatus &EventStatus);

__property TBeginTransCompleteEvent OnBeginTransComplete;

void __fastcall ~::OnBeginTransComplete(TADOConnection *Connection,
                                        int TransactionLevel,
                                        const Error *Error,
                                        TEventStatus &EventStatus)
```

TransactionLevel 인자는 트랜잭션의 수준, 즉 현재 Connection 객체에서 현재 트랜잭션이 내포된 깊이를 알려준다. 하나의 Connection 객체에 대해 BeginTrans 메소드가 처음으로 수행되면 이 인자의 값은 1이 된다. 3번째 Error 인자는 연결에 대해 발생했을 수 있는 에러에 대한 정보를 담은 Error 객체에 대한 참조를 돌려준다. 앞으로 설명할 여러 이벤트에서 인자로 사용되는 4번째 인자인 EventStatus 인자는 보는 것과 같이 TEventStatus 열거형 값을 가지는데 다음 표의 여러 값들 중 하나를 가진다. 만약 이 인자의 값이 esErrorsOccured라면 에러가 발생한 것이다. 첫번째 인자인 Connection은 이 이벤트를 발생시킨 Connection 객체를 돌려준다.

값	의미
esOK	어떤 에러나 문제없이 조작이 성공적으로 실행됨
esErrorsOccured	조작 실행 중에 에러가 발생했다.
esCantDeny	연결 이벤트에만 있는 것으로 계류 중인 연결이 취소될 수 없다.
esCancel	연결 이벤트에만 있는 것으로 계류 중인 연결이 활성화 되기 전에 취소 되었다.
esUnwantedEvent	다음에 이어지는 이벤트의 통지를 막았다.

OnCommitTransComplete

CommitTrans 메소드가 성공적으로 수행되고 난 후에 발생하는 것으로 ADO에게 현재트랜잭션의 범위 안에

서 생긴 변경들을 데이터 공급자들에게 출력하라고 알려주는 역할을 한다. 다음은 이벤트 핸들러이다. 대부분의 인자의 내용은 BeginTransComplete 이벤트의 해당 인자들과 동일하다.

```
typedef void __fastcall (__closure *TConnectErrorEvent)
                        (TADOConnection* Connection,
                         const _di_Error Error,
                         TEventStatus &EventStatus);

__property TConnectErrorEvent OnCommitTransComplete;

void __fastcall ~::OnCommitTransComplete(TADOConnection *Connection,
                                         const Error *Error,
                                         TEventStatus &EventStatus)
```

OnRollbackTransComplete

RollbackTrans 메소드가 성공적으로 수행되고 난 후에 발생하는 것으로 ADO에게 현재 트랜잭션의 범위 안에서 생긴 변경들을 폐기하고 데이터 공급자에게 출력하지 말라고 알려주는 역할을 한다. 다음은 이 이벤트의 이벤트 핸들러 부분으로 대부분의 인자의 내용은 BeginTransComplete 이벤트의 해당 인자들과 동일하다.

```
typedef void __fastcall (__closure *TConnectErrorEvent)
                        (TADOConnection* Connection,
                         const _di_Error Error,
                         TEventStatus &EventStatus);

__property TConnectErrorEvent OnRollbackTransComplete;

void __fastcall ~::OnRollbackTransComplete(TADOConnection *Connection,
                                           const Error *Error,
                                           TEventStatus &EventStatus)
```

연결/설정 관련 이벤트들

이 이벤트들은 연결이 초기화 되는 중이거나 성공적으로 연결되거나 아니면 성공적으로 종료될 때 발생한다.

OnWillConnect

Connection 객체의 Open 메소드가 호출된 즉시 발생하는 것으로 ADO에게 연결을 열라고 알려주는 역할을 한다. 이 이벤트의 이벤트 핸들러는 다음과 같다.

```
typedef void __fastcall (__closure *TWillConnectEvent)
                        (TADOConnection* Connection,
                         WideString &ConnectionString,
                         WideString &UserID,
                         WideString &Password,
                         TConnectOption &ConnectOptions,
                         TEventStatus &EventStatus);

__property TWillConnectEvent OnWillConnect;

void __fastcall ~::OnWillConnect(TADOConnection *Connection,
                                 WideString &ConnectionString,
                                 WideString &UserID,
                                 WideString &Password,
                                 TConnectOption &ConnectOptions,
                                 TEventStatus &EventStatus)
```

ConnectionString 인자는 데이터 원본에 연결하는데 쓰일 연결 문자열을 돌려준다. UserID와 Password 인자들은 지정된 데이터 원본에 로그인 하는데 필요한 사용자 인증 정보들이다. ConnectOptions 인자는 해당

Connection 객체의 동일한 프로퍼티에서 설정한 값을 담고 있으며 첫번째와 마지막 인자들은 위의 이벤트들에서 설명한 내용들과 동일하다.

OnConnectComplete

Connection 객체가 데이터 소스에 대한 연결을 마친 직후 발생한다. 비동기적으로 연결을 여는 경우, 이 이벤트를 통해서 연결이 실제로 만들어진 시점을 알아낼 수 있다. 다음은 이 이벤트들의 이벤트 핸들러이다.

```
typedef void __fastcall (__closure *TConnectErrorEvent)
                        (TADOConnection* Connection,
                         const _di_Error Error,
                         TEventStatus &EventStatus);

__property TConnectErrorEvent OnConnectComplete;

void __fastcall ~::OnConnectComplete(TADOConnection *Connection,
                                     const Error *Error,
                                     TEventStatus &EventStatus)
```

이 이벤트의 인자들의 의미는 위의 WillConnect 이벤트의 해당 인자들과 동일하다.

OnDisconnect

Connection 객체의 Close 메소드의 호출과 같은 코드를 통해 명시적으로 Connection 객체가 닫히거나 아니면 공급자에 의해 또는 네트웍 오류등 어떠한 문제로 인해 묵시적으로 연결이 끊긴 경우 발생한다. 비동기적으로 연결을 여는 경우 이 이벤트를 통해서 연결이 닫힌 시점을 알아낼 수 있다. 다음은 이 이벤트의 이벤트 핸들러이다.

```
typedef void __fastcall (__closure *TDisconnectEvent)
                        (TADOConnection* Connection,
                         TEventStatus &EventStatus);

__property TDisconnectEvent OnDisconnect;

void __fastcall ~::OnDisconnect(TADOConnection *Connection,
                                TEventStatus &EventStatus)
```

이 이벤트의 인자들의 의미 또한 위의 WillConnect 이벤트의 해당 인자들과 동일하다.

명령/수행 관련 이벤트들

이 이벤트들은 연결이 어떠한 명령을 수행하려고 할 때 또는 명령을 마쳤을 때 발생한다.

OnWillExecute

Connection 객체가 어떠한 명령을 수행하기 직전에 발생한다. 이 이벤트의 이벤트 핸들러는 다음과 같다.

```
typedef void __fastcall (__closure *TWillExecuteEvent)
                        (TADOConnection* Connection,
                         WideString &CommandText,
                         TCursorType &CursorType,
                         TADOLockType &LockType,
                         TCommandType &CommandType,
                         TExecuteOptions &ExecuteOptions,
                         TEventStatus &EventStatus,
                         const _di__Command Command,
                         const _di__Recordset Recordset);

__property TWillExecuteEvent OnWillExecute;

void __fastcall ~::OnWillExecute(TADOConnection *Connection,
                                 WideString &CommandText,
                                 TCursorType &CursorType,
                                 TADOLockType &LockType,
                                 TCommandType &CommandType,
                                 TExecuteOptions &ExecuteOptions,
                                 TEventStatus &EventStatus,
                                 const _Command *Command,
                                 const _Recordset *Recordset)
```

CommandText 인자는 데이터 원본에 수행될 명령 문자열을 돌려준다. CursorType 인자는 사용될 커서의 종류를 뜻한다. LockType 인자는 사용될 잠금의 종류를 뜻한다. 그 밖의 CommandType 인자나 ExecuteOptions 인자는 Execute 메소드를 사용한 다른 예제들에서 보았던 그대로이고 이벤트는 그것들 각각의 맞는 열거형의 값들로 돌려준다. EventStatus 인자 역시 위에서 살펴 보았다. 첫번째 인자인 Connection 역시 이벤트를 발생시킨 Connection 객체를 가리키며 Command 인자는 이 연결을 ActiveConnection 속성으로 가지며 실행된 Command 객체를 가리키고 마지막 Recordset 인자는 이 이벤트의 수행에 의해서 반환될 Recordset 객체를 가리킨다.

OnExecuteComplete

Connection 객체가 어떠한 명령을 수행한 직후에 발생한다. 이 이벤트의 이벤트 핸들러는 다음과 같다.

```
typedef void __fastcall (__closure *TExecuteCompleteEvent)
                        (TADOConnection* Connection,
                         int RecordsAffected,
                         const _di_Error Error,
                         TEventStatus &EventStatus,
                         const _di__Command Command,
                         const _di__Recordset Recordset);

__property TExecuteCompleteEvent OnExecuteComplete;

void __fastcall ~::OnExecuteComplete(TADOConnection *Connection,
                                     int RecordsAffected,
                                     const Error *Error,
                                     TEventStatus &EventStatus,
                                     const _Command *Command,
                                     const _Recordset *Recordset)
```

RecordsAffected 인자는 데이터 원본에 대한 명령의 수행에 관련된 레코드들의 개수를 뜻한다. EventStatus 값이 esErrorsOccured 이면 Error 인자로 Error에 대한 정보를 담은 Error 객체에 대한 참조를 돌려준다. Error가 발생하지 않았다면 이 참조는 무시해도 된다. Command 인자는 실행된 명령을 뜻하는 Command 객체(존재하는 경우, 지금은 없지만 앞으로 상당히 많을 것이다)를 가리킨다. 역시 위의 이벤트의 인자와 동일한 마지막 인자인 Recordset 인자는 명령의 수행에 의해 반환되는 Recordset 객체를 가리킨다. 첫번째 인

자인 Connection 역시 이 이벤트를 발생시킨 Connection 객체를 가리킨다.

경고 이벤트

이 이벤트는 어떠한 주의나 경고가 생겼을 때 발생한다.

OnInfoMessage

Connection 객체에 대해 어떠한 경고가 생겼을 때 발생한다. 이 이벤트의 이벤트 핸들러는 다음과 같다.

```
typedef void __fastcall (__closure *TInfoMessageEvent)
                        (TADOConnection* Connection,
                         const _di_Error Error,
                         TEventStatus &EventStatus);

__property TInfoMessageEvent OnInfoMessage;

void __fastcall ~::OnInfoMessage(TADOConnection *Connection,
                                 const Error *Error,
                                 TEventStatus &EventStatus)
```

2번째 인자인 Error는 Connection 객체에 대해 발생했을 수 있는 에러에 대한 정보를 담은 Error 객체에 대한 참조를 돌려준다. 익숙하게 EventStatus 인자로 Connection 객체의 상태를 점검해도 될 것이다. 마지막으로 Connection 인자는 이벤트를 발생시킨 Connection 객체를 돌려준다.

자 그럼 위의 이벤트를 다룬 것에서 제일 중요한 내용이 되는 에러에 대한 처리에 대해 살펴보자.

Errors 컬렉션

여러분이 이 강의를 잘 보고 있다면 강의의 제일 처음 부분에서 우리의 레밍이 그린 ADO 객체모델 그림에 Connection 객체에 대한 컬렉션의 형태로 Errors 객체가 존재한다는 것을 기억할 것이다. 말 그대로 Errors 컬렉션은 SQL 문장이나 저장 프로시저를 수행할 때 또는 레코드 셋들을 가져올 때 데이터 공급자가 보고한 어떠한 문제들을 알아내기 위한 수단이다. 데이터 공급자에 어떤 문제가 있으며 데이터 소비자에게 그 문제들을 알릴 필요가 있을 때 마다 Errors 컬렉션에 Error 객체들이 동적으로 채워진다. 한가지 아쉬운 것은 ADO 자체의 에러들은 Errors 컬렉션에 포함되지 않으며 어플리케이션 코드의 런타임 에러 형태로 통지될 뿐이다. Errors 컬렉션은 데이터 공급자 자체가 발생시킨 에러나 경고들만을 담는다.
다시 객체 모델적인 입장에서 본다면 Errors 컬렉션은 Connection 객체의 한 속성이다. 이 컬렉션은 델파이 인터페이스 Errors형으로 (_di_Errors) 다음과 같은 메소드와 속성들을 가지고 있다.

Property & Method	의미
Clear	이 메소드는 Errors 컬렉션을 비운다.
Count	이 속성은 Errors 컬렉션이 현재 가지고 있는 Error 객체들의 개수를 뜻한다.
Item	이 속성은 색인 번호를 통해서 컬렉션 안에 담긴 개별 Error 객체들에 접근하는데 쓰인다.
Refresh	이 메소드는 Errors 컬렉션을 갱신한다.

Error 객체에는 다음과 같은 속성들이 있다.

Property	의미
Description	에러에 대한 설명 (문자열)
HelpContext	에러에 대한 도움말 문맥 ID

HelpFile	에러에 대한 도움말 파일 이름
NativeError	공급자 고유의 에러번호
Number	에러번호
Source	에러의 출저 --- 에러를 발생시킨 어플리케이션이나 객체의 이름
SQLState	SQL 에러번호 --- 주어진 에러 객체에 대한 SQL 상태, 다섯 글자 짜리 문자열이다.

다음은 위에서 살펴본 Errors 컬렉션과 Error 객체들의 예로 일반적인 어플리케이션 내에서의 사용 방법이다.

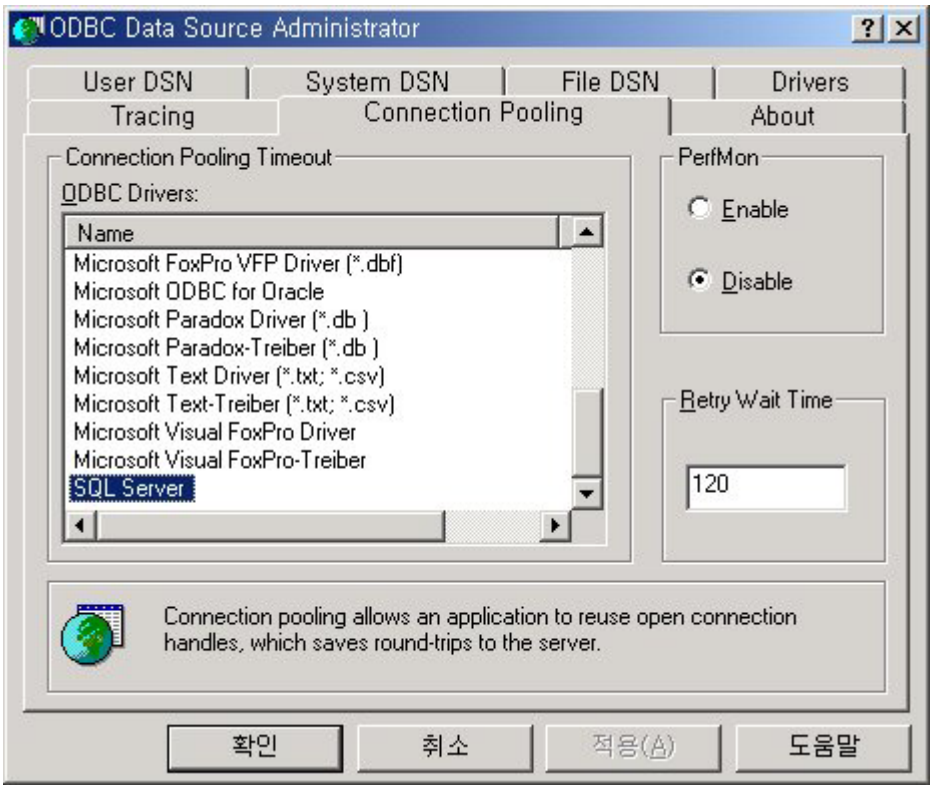
```
try
{
  // Connection 객체에 관계된 각종 작업을 한다.
}
catch(...)
{
  String ErrorMsg = "";

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

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

★ Errors 컬렉션은 MTS 나 COM+ 내의 ActiveX Server 안에서 쓰일 때에는 제대로 작동하지 않는다. 이때는 Errors 컬렉션의 에러들은 다른 어떤 객체나 배열 등 다른 자료형으로 캡슐화 해서 돌려주는 함수를 만들어서 해결한다.

연결 풀링

이전의 강의에서 연결 풀링이란 데이터베이스 인스턴스가 서버 내에서 활성화 될 때 설정에 맞게 생성된 일종의 메모리 구조체 버퍼라고 언급한 적이 있는데 그것은 서버측에서 물리적인 관점에서 본 포괄적인 개념이었다. 이제 자세히 살펴볼 때가 왔다. 간단하게 연결 풀링 이란 코드를 통해서 종료된 연결들을 ADO가 관리하는 데 쓰이는 기법을 말한다. 어떠한 연결이 일정 시간 이상 쓰이지 않아도 연결이 아예 파괴되지는 않으며 단지 닫히기만 할 뿐이다. 그리고 연결은 새로 요청된 연결에 지정되어 있는 정보 --- 사용자 이름과 패스워드도 포함된다 --- 가 풀에 있는 연결과 정확히 동일할 때만 재사용된다. 이러한 기법은 Connection 객체가 물리적으로 생성되는 횟수를 줄여주므로 서버의 부담을 크게 감소시킨다. 코드가 Connection 객체의 Open 메소드를 통해서 어떠한 연결을 요청하면 데이터 공급자는 풀에 있는 기존의 연결들 중 요청된 연결에 일치하는 것이 있는지 알아본다. 있으면 기존의 Connection 객체를 돌려주므로 새로운 Connection 객체를 생성하는데 따르는 자원 낭비를 막을 수 있다. 이 때 기존의 연결이 누구에 의해서 만들어진 것인지는 문제가 되지는 않는다. 누군가가 어떠한 연결을 닫으면 그 연결은 연결 풀에 다시 들어가며 다시 사용되기를 기다린다. 사용되지 않은 연결들은 영원히 연결 풀에 들어가지 않으며 미리 지정된 시간이 지나면 그러한 연결들은 자동적으로 폐기되고 관련 자원들도 해제된다.

ODBC 공급자들을 사용하는 경우 연결 풀링 설정은 아래의 그림에서와 같이 연결 풀링 탭을 통해 설정할 수 있다.



Summary

자 그럼 Connection 객체에 대해 마칠 때가 온 것 같다. 마치기 전에 요약을 해보면 중요 골자는 ADO Connection 객체의 한 인스턴스가 데이터 원본에 대한 하나의 고유한 연결을 뜻한다는 것이다. 그리고 그 인스턴스의 메소드들과 속성들은 연결의 특성이나 작동 방식을 제어할 수 있는 수단을 제공한다. Connection 객체로 할 수 있는 일들에는 다음과 같은 것이 있었다.

★ 연결에 대해 SQL 문을 직접 수행하거나 레코드 셋으로 데이터를 가져온다.
★ 데이터, 트랜잭션과 격리(잠금) 수준을 관리한다.
★ 데이터 공급자로부터 스키마 정보를 얻는다.
★ Errors 컬렉션을 통해서 데이터 공급자의 에러 정보에 접근한다.
★ 자체로 연결 풀링을 제어한다.
★ 저장 프로시저를 수행한다.
★ ADO Connection 객체에 의해서 발생한 이벤트들을 관리한다.

위의 것들은 알아본 것들로서 다시 한번 익혀두기 바란다. Connection 객체에 대해서 앞으로 다뤄 볼 것을 이야기 한다면

★ Command 객체나 저장 프로시저들을 생성하고 그것들을 Connection 객체의 유효한 메소드처럼 직접 수행하는 기능들 --- 필자의 경험으로 볼 때 Command 객체는 저장 프로시저를 위한 것이다. --- Command 객체 편을 기대해도 좋을 것이다.
★ 웹 자원들을 효율 적으로 관리하는 기능들 및 다양한 공급자들에 연결 이용하는 기능들. --- 유심히 강의를 지켜본 독자라면 ConnectionString의 URL 속성을 다루지 않았다는 것을 기억할 것이다. 그리고 RDBMS나 로컬 데이터베이스만이 데이터 원본이 될 수 있다는 것이 아님을 기억하기 바란다. (필자는 웹 파일 구조와 다양한 파일 데이터 원본, Data Shaping 기술이나 COM+의 기능 중 하나인 메모리 내부 데이터베이스도 다룰 것이다.)

Connection 객체를 명시적으로 생성하지 않고도 Recordset을 생성할 수 있고 어떠한 종류의 데이터 원본에도 묵시적인 연결을 만들 수 있으므로 --- ADO 탭에 있는 각각의 Component에는 RDSConnection을 제외하고 ConnectionString 속성이 다 존재한다 --- Connection 객체가 괜히 실행파일 크기만 늘리고 그리 유용하지 않게 보일 수도 있다. 그러나 이후의 강의에서 보게 되겠지만 데이터를 조회하고 저장하는 엔터프라이즈 어플리케이션에 있어서 Connection 객체는 필수적인 요소이다.

이것으로 Connection 객체에 대한 강의는 마치고자 한다. 그러나 진정한 마침은 여러분의 활용과 적용이 숙련될 때이다. 많은 것을 설명했지만 죄다 부분에 그치는 내용들이곤 했다. 정작 중요한 것은 다양한 객체들 간의 상호적인 관계와 어울림 속에서 발휘되는 부분을 넘어서는 어떤 것일 것이다. 여러분이 앞으로 진행될 이 미흡하고 다소 부족한 강의에서 그것들을 얻을 수 있기를 바란다. 다시 각설하고 우리의 레밍은 Connection 객체를 배움으로써 데이터 원본이 제공하는 강력하고도 낯선 힘을 얻을 수 있는 기회를 얻은 것이다. 사실 이 세상의 모든 이야기들과 사건들은 주체라고 불리는 모호한 어떤 존재가 우연으로 가득찬 세계의 순간순간 속에 필연이란 특이점에 연결됨으로 발생하는 것이다. 좌우지간 어딘가에 연결된다는 것은 멋진 출발을 의미한다. 자아, Command 객체를 향해 멋지게 출발하자~

Mortalpain