

FIBPlus 6.9.6 Руководство разработчика

Оглавление

Соединение с базой данных.....	9
Параметры соединения.....	9
Создание и удаление БД.....	11
Кэширование метаданных.....	11
Кэширование блоб-полей.....	12
Обработка потери соединения.....	13
Другие полезные методы TrFIBDatabase.....	14
Выполнение простых SQL-запросов.....	15
Получение значений генераторов.....	15
Получение информации о таблицах и полях.....	15
Получение информации о версии сервера.....	15
Получение информации о состоянии коннекта.....	15
Работа с контекстными переменными Firebird 2.....	16
Прерывание выполнения запросов в Firebird начиная с версии 2.1.....	16
Прерывание выполнения запросов в Firebird начиная с версии 2.5.....	16
Свойство TDesignDBOptions.....	17
Работа с транзакциями.....	18
Настройка параметров транзакции.....	18
Планирование использования транзакций в приложениях.....	18
Использование SavePoints.....	19
Свойство TimeOut.....	19
Двухфазный коммит.....	20
Немедленное исполнение запроса (ExecSQLImmediate).....	20
Получение информации о состоянии транзакции.....	20
Выполнение SQL- запросов (компонент - TrFIBQuery).....	21
Методы и свойства компонента - TrFIBQuery.....	22
Свойства доступные под дизайном (published).....	22
Обработчики событий.....	23
Публичные методы и свойства.....	23
Работа с текстом SQL.....	28
SQL - секции.....	28
Макросы.....	29
Условия.....	30
Пакетная обработка.....	31
Выполнение DDL операторов.....	32
Повторное использование запросов.....	33
Работа с наборами данных.....	34
Базовые принципы работы с наборами данных.....	34
Автоматическая генерация обновляющих запросов.....	36
Master-detail.....	38
Локальная сортировка.....	39
Сортировка национальных символов.....	40
Локальная фильтрация.....	40
Поиск данных.....	41
Пессимистическая блокировка.....	42
Работа в режиме ограниченного кэша.....	43
Работа с внутренним кэшем набора данных.....	44
Использование кэшированных изменений.....	45
Использование уникальных типов полей FIBPlus.....	47

TFIBLargeIntField.....	47
TFIBBCDField.....	47
TFIBWideStringField.....	47
TFIBBooleanField.....	47
TFIBGuidField.....	47
Работа с полями-массивами.....	48
Использование контейнеров TDataSetsContainer.....	49
Дополнительные действия при модификации данных TrFIBUpdateObject.....	49
Работа в режиме разделенных транзакций.....	50
Пакетная обработка.....	51
Перечень свойств TrFIBDataSet.....	51
Работа с блоб-полями в клиентских приложениях InterBase и Firebird на основе компонентов FIBPlus.....	54
Введение.....	54
Использование TrFIBDataSet при работе с BLOB-полями.....	54
Использование TrFIBQuery при работе с BLOB-полями.....	58
Уникальные возможности FIBPlus: Client BLOB-filters. "Прозрачное" сжатие BLOB-полей.....	59
Централизованная обработка ошибок – TrFIBErrorHandler.....	62
Исполнение скриптов TrFIBScripter.....	63
Получение событий TFIBSibEventAlerter.....	65
Отладка приложений FIBPlus.....	65
Мониторинг SQL—запросов.....	65
Регистрация выполняемых запросов.....	65
Репозитории FIBPlus.....	66
Репозиторий наборов данных.....	66
Репозиторий полей.....	67
Репозиторий ошибок.....	68
Поддержка FB2.X.....	69
Дополнительные возможности.....	70
Полная поддержка UNICODE_FSS и UTF8.....	70
Опция компиляции NO_GUI.....	70
Использование SynEdit в редакторах.....	70
Уникальное расширение FIBPlusTools.....	71
Preferences.....	71
SQL Navigator.....	72
Работа с сервисами.....	74
Получение информации о сервере.....	74
Управление пользователями сервера.....	76
Обслуживание базы данных.....	78
Получение статистической информации о состоянии БД.....	79
TrFIBDatabase.....	80
Свойства.....	80
AliasName.....	80
BlobSwapSupport.....	80
CacheShemaOptions.....	80
Connected.....	80
ConnectParams.....	80
DBName.....	81
DBParams.....	81
Дополнительные параметры подключения можно посмотреть в документации к	

серверу (APIGuide.pdf, DevGuide.pdf).....	81
DefaultTransaction.....	81
DefaultUpdateTransaction.....	81
DesignDBOptions.....	81
LibraryName.....	83
SaveAliasParamsAfterConnect.....	83
SQLDialect.....	83
SQLLogger.....	83
SynchronizeTime.....	83
TimeOut.....	83
UpperOldNames.....	83
UseLoginPrompt.....	83
UseRepositories.....	83
WaitForRestoreConnect.....	84
UseBlrToTextFilter.....	84
Описание событий.....	84
AfterConnect.....	84
AfterDisconnect.....	84
AfterEndTransaction.....	84
AfterLoadBlobFromSwap.....	84
AfterRestoreConnect.....	84
AfterSaveBlobToSwap.....	84
AfterStartTransaction.....	84
BeforeConnect.....	84
BeforeDisconnect.....	85
BeforeEndTransaction.....	85
BeforeLoadBlobFromSwap.....	85
BeforeSaveBlobToSwap.....	85
BeforeStartTransaction.....	85
OnAcceptCacheSchema.....	85
OnErrorRestoreConnect.....	85
OnLostConnect.....	85
Public свойства.....	86
Методы.....	89
ТрFIBTransaction.....	93
Свойства.....	93
DefaultDatabase.....	93
Timeout Changed.....	93
TimeoutAction.....	93
TPBMode.....	93
TRParams.....	93
UserKindTransaction.....	93
TransactionID.....	93
State.....	93
События.....	93
AfterEnd Changed.....	94
AfterSQLExecute.....	94
AfterStart.....	94
BeforeEnd.....	94
BeforeSQLExecute.....	94
BeforeStart.....	94

OnTimeout.....	94
OnIdleConnect.....	94
Методы.....	94
ТрFIBQuery.....	96
Свойства.....	96
Conditions.....	96
Database.....	96
GoToFirstRecordOnExecute.....	96
Options.....	96
ParamsCheck.....	97
SQL.....	97
Transaction.....	97
Свойства.....	97
AfterExecute.....	97
BeforeExecute.....	97
OnBatchError.....	97
OnBatching.....	97
OnExecuteError.....	97
OnSQLChanging.....	97
TransactionEnded.....	97
TransactionEnding.....	97
Методы.....	97
Public свойства.....	101
ТрFIBStoredProc.....	103
TFIBXSQLDA.....	103
Public свойства.....	103
Public методы.....	103
TFIBXSQLVAR.....	104
Public свойства.....	104
Public методы.....	105
ТрFIBDataset.....	106
Свойства.....	106
Active.....	106
Filter.....	106
FilterOptions.....	106
AllowedUpdateKinds.....	106
AutoCalcFields.....	106
AutoCommit.....	106
AutoUpdateOptions.....	106
CacheModelOptions.....	108
Conditions.....	108
Container.....	108
Database.....	108
DataSet_ID.....	108
Description.....	109
DefaultFormats.....	109
DetailConditions.....	109
FieldOriginalRule.....	109
Options.....	109
PrepareOptions.....	110
RefreshTransactionKind.....	111

SQLs.....	111
SQLScreenCursor.....	111
Transaction.....	112
UniDirectional.....	112
UpdateRecordTypes.....	112
UpdateTransaction.....	112
События.....	112
AfterEndTransaction.....	112
AfterEndUpdateTransaction.....	112
AfterFetchRecord.....	112
AfterStartTransaction.....	112
AfterStartUpdateTransaction.....	112
BeforeEndTransaction.....	112
BeforeEndUpdateTransaction.....	112
BeforeFetchRecord.....	112
BeforeStartTransaction.....	112
BeforeStartUpdateTransaction.....	112
DatabaseDisconnected.....	113
DatabaseDisconnecting.....	113
DatabaseFree.....	113
OnApplyDefaultValue.....	113
OnAskRecordCount.....	113
OnCompareFieldValues.....	113
OnFieldChange.....	113
OnFillClientBlob.....	113
OnReadBlobField, OnWriteBlobField.....	113
OnLockError.....	114
OnLockSQLText.....	114
TransactionEnded.....	114
TransactionEnding.....	114
TransactionFree.....	114
Методы.....	114
ТрFIBUpdateObject.....	122
Свойства.....	122
Conditions.....	122
DataSet.....	122
ExecuteOrder.....	122
KindUpdate.....	122
OrderInList.....	123
TDatasetContainer.....	123
Свойства.....	123
Active.....	123
MasterContainer.....	123
ISGlobal.....	123
События.....	123
OnApplyDefaultValue.....	123
OnApplyFieldRepository.....	123
OnCompareFieldValues.....	123
OnDataSetError.....	123
OnDataSetEvent.....	124
OnUserEvent.....	124

Методы.....	124
TSIBfibEventAlerter.....	124
Свойства.....	124
AutoRegister.....	124
Database.....	124
Events.....	124
События.....	124
TrFIBErrorHandler.....	125
Свойства.....	125
Options.....	125
События.....	125
TrFIBClientDataSet.....	125
Методы.....	125
TrFIBDataSetProvider.....	126
TrFIBScripter.....	126
Свойства.....	126
Методы.....	126
TFIBSQLMonitor.....	127
Свойства.....	127
TraceFlags.....	127
События.....	127
OnSQL.....	127
TFIBSQLLogger.....	128
Свойства.....	128
Database.....	128
ActiveStatistics.....	128
ActiveLogging.....	128
ApplicationID.....	128
LogFileName.....	128
StatisticsParams.....	128
LogFlags.....	128
ForceSaveLog.....	128
Методы.....	128
TrFIBCustomService.....	129
Свойства.....	129
Handle: TISC_SVC_HANDLE.....	129
ServiceParamBySPB.....	129
Active.....	129
ServerName.....	129
Protocol.....	129
Params.....	129
LoginPrompt.....	129
LibraryName.....	129
SQLLogger.....	129
События.....	129
OnAttach.....	129
OnLogin.....	129
Методы.....	130
TrFIBServerProperties.....	130
Свойства.....	130
Option.....	130

DatabaseInfo.....	130
LicenseInfo.....	130
LicenseMaskInfo.....	130
VersionInfo.....	130
ConfigParams.....	130
Методы.....	131
ТрFIBSecurityService.....	131
Свойства.....	131
SecurityAction.....	131
UserName.....	131
Password.....	131
SQLRole.....	131
FirstName, MiddleName, LastName.....	131
UserID.....	131
GroupID.....	131
UserInfo.....	131
UserInfoCount.....	132
Методы.....	132
ТрFIBBackupService.....	132
Свойства.....	132
BackupFile.....	132
DatabaseName.....	132
Option.....	133

Соединение с базой данных

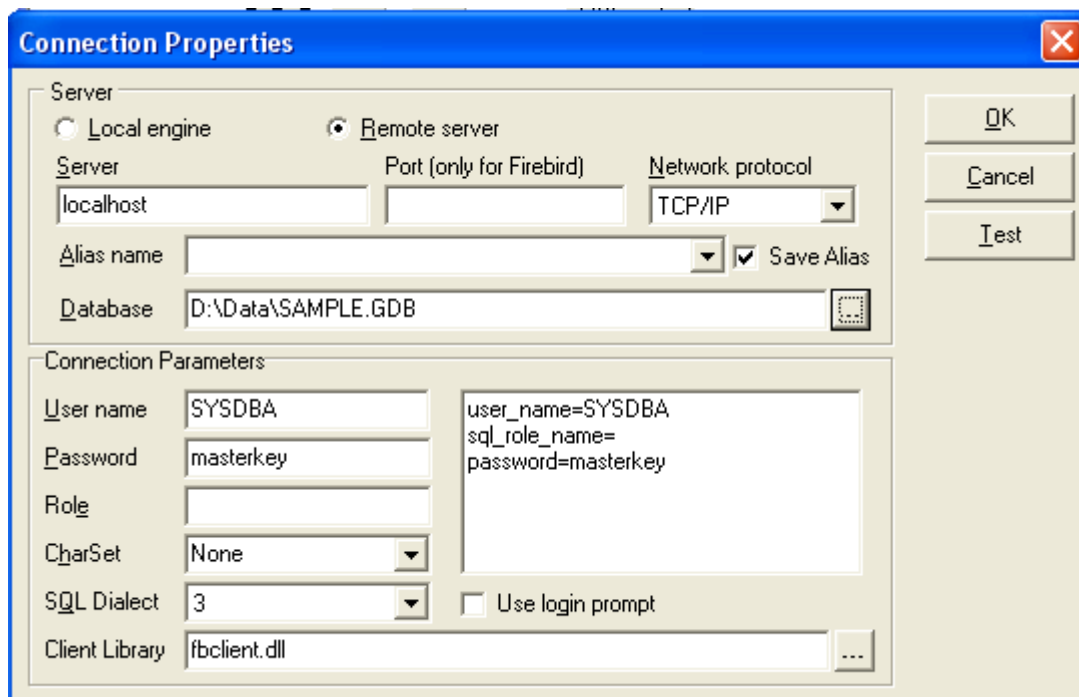
Для соединения с базой данных (БД) используйте компонент TrFIBDatabase. Подробное описание всех свойств и методов компонента можно прочитать в Приложении.

Параметры соединения

Параметры соединения являются типичными для сервера InterBase/Firebird:

- путь к файлу базы данных;
- имя пользователя и пароль;
- роль пользователя;
- набор символов;
- диалект;
- клиентская библиотека (gds32.dll для InterBase и fbclient.dll для Firebird).
- порт используемый сервером при соединении по TCP/IP (только для Firebird)

Для задания всех свойств сразу удобно пользоваться встроенным диалогом настройки подключения, представленном на рисунке 1.



Диалог «Database Editor» можно вызывать из контекстного меню компонента в design-time. Здесь вы можете указать необходимые параметры, взять параметры из данных псевдонима (Alias) или сохранить их в псевдониме.

Замечание: Под псевдонимом подразумевается клиентский псевдоним. По аналогии с технологией псевдонимов под БДЕ. Не надо путать с псевдонимами на стороне сервера.

Также можно проверить правильность параметров, попытавшись произвести тестовое подключение.

Все, что можно сделать в описанном выше диалоге, можно также сделать непосредственно из кода приложения.

Для заполнения параметров соединения, FIBPlus предлагает использовать на выбор два свойства компонента TрFIBDatabase.

А) DBParams:TDBParams - фактически это обыкновенный TStrings со всеми его методами.

Б) ConnectParams:TConnectParams – с внутренними свойствами

```
property UserName : string;  
property RoleName : string;  
property Password : string;  
property CharSet : string ;
```

Использование свойства ConnectParams дает более простой и понятный способ заполнения параметров соединения, но при этом обрабатывает лишь самые необходимые параметры. Для заполнения всех остальных параметров, все равно придется использовать доступ к DBParams напрямую.

Для соединения с БД нужно вызвать метод Open, либо установить свойство Connected в True. Приведем пример кода для подключения к базе данных, использующий свойство ConnectParams:

```
function Login(DataBase: TрFIBDatabase; dbpath, uname, upass, urole: string):  
Boolean;  
begin  
  if DataBase.Connected then DataBase.Connected := False;  
  with FDataBase.ConnectParams do begin  
    UserName := uname;  
    Password := upass;  
    RoleName := urole;  
  end;  
  DataBase.DBName := dbpath;  
  try DataBase.Connected := True;  
  except  
    on e: Exception do  
      ShowMessage(e.Message) ;  
  end;  
  Result := DataBase.Connected;  
end;
```

Для завершения подключения необходимо вызвать метод Close, либо установить свойство Connected в False. При этом все транзакции и запросы, которые к этому моменту были открытыми, будут закрыты автоматически самой библиотекой.

Замечание: Если транзакция закрывается автоматически, то способ ее закрытия зависит от свойства TFIBTransaction.TimeoutAction. Т.е. если у транзакции TimeoutAction равен TACCommit или TACCommitRetaining, то она будет закрыта методом Commit. В остальных случаях методом Rollback

Создание и удаление БД

Создать новую базу данных очень просто. Для этого вы должны задать параметры создаваемой БД и вызвать метод `CreateDatabase`:

Delphi

```
with Database1 do begin
  DBParams.Clear;
  DBParams.Add('USER 'SYSDBA' PASSWORD 'masterkey');
  DBParams.Add('PAGE_SIZE = 2048');
  DBParams.Add('DEFAULT CHARACTER SET WIN1251');
  DBName := 'SERV_DB:C:\DB\TEST.IB';
  SQLDialect := 3;
end;

try
  Database1.CreateDataBase;
except
  // Error handling
end;
```

C++

```
Database1->DBParams->Clear();
Database1->DBParams->Add("USER 'SYSDBA' PASSWORD 'masterkey'");
Database1->DBParams->Add("PAGE_SIZE = 2048");
Database1->DBParams->Add("DEFAULT CHARACTER SET WIN1251");
Database1->DBName = "SERV_DB:C:\\DB\\TEST.GDB";
Database1->SQLDialect = 3;

try
{ Database1->CreateDatabase(); }
catch (...)
{ // Error
}
```

Замечание: Для создания БД свойство `ConnectParams` использовать нельзя. Нужно работать с `DBParams` напрямую.

Для удаления БД используйте метод `DropDatabase`. В момент использования этого метода, вы должны быть подключены к БД.

Кэширование метаданных

FIBPlus предоставляет нам возможность автоматически получать системную информацию о полях таблиц, самостоятельно настраивая такие свойства полей в `TrFIBDataSet` как `Required` (для NOT NULL полей), `ReadOnly` (для вычисляемых полей) и `DefaultExpression` (для полей, у которых в базе данных задано значение по умолчанию). Это удобно и для программиста, и для пользователя, поскольку первому не нужно заботиться о ручной настройке указанных свойств во время разработки клиентского приложения, а второй получает более осмысленные сообщения при работе с программой. В частности, если какое-то поле описано в базе данных как NOT NULL, то при попытке оставить его пустым пользователь получит сообщение «Field '...' must have a value.», что гораздо проще для понимания, чем системная ошибка `InterBase/Firebird` о нарушении PRIMARY KEY. То же самое касается вычисляемых полей, поскольку очевидно, что такие поля нельзя редактировать. FIBPlus автоматически задаст у таких полей свойство `ReadOnly` равное `True`, и пользователь не будет мучаться из-за непонятных ошибок при попытке исправить значения этих полей в `TDBGrid`.

Однако данная возможность FIBPlus имеет и свой недостаток, который очевидным образом проявляется при работе с низкоскоростными каналами связи. Чтобы получить

информацию о полях, компоненты FIBPlus выполняют дополнительные «внутренние» запросы, обращаясь к системным таблицам InterBase/Firebird. Разумеется, при большом количестве таблиц в приложении, а также при большом количестве полей в этих таблицах, работа приложения может замедлиться, а трафик возрасти. Особенно это видно на стадии первоначальных открытий запросов, так как каждый из них сопровождается серией дополнительных. Потом, в процессе работы программы, при повторных запросах, FIBPlus использует уже полученную ранее информацию, однако при старте приложения можно обратить внимание на некоторое замедление работы.

TpFIBDatabase позволяет сохранить информацию о метаданных на клиентском компьютере и использовать ее при следующих сеансах работы. Отвечает за это свойство *TCacheSchemaOptions*.

```
TCacheSchemaOptions = class(TPersistent)
  property LocalCacheFile: string;
  property AutoSaveToFile: Boolean .. default False;
  property AutoLoadFromFile: Boolean .. default False;
  property ValidateAfterLoad: Boolean .. default True;
end;
```

Свойство *LocalCacheFile* позволяет задать имя файла, в котором будет сохраняться эта информация. *AutoSaveToFile* отвечает за автоматическую запись кеша в файл при закрытии приложения. *AutoLoadFromFile* отвечает за загрузку кеша из файла. И, наконец, *ValidateAfterLoad* указывает на то, стоит ли проверять сохраненный кеш после загрузки. В дополнение к этим свойствам добавляется событие *OnAcceptCacheSchema*, где вы можете указать для каких объектов не нужно загружать сохраненную информацию.

Использовать это свойство также очень просто.

```
with pFIBDatabase1.CacheSchemaOptions do begin
  LocalCacheFile := 'fibplus.cache';
  AutoSaveToFile := True;
  AutoLoadFromFile := True;
  ValidateAfterLoad:= True;
end;
```

Изменение метаданных в процессе работы клиентских приложений может привести к ошибкам в работе клиентов. Чтобы избежать появления этих ошибок можно например сделать триггер на таблицы репозитариев, который выдавал бы определенное событие. По этому событию можно чистить кэш метаданных во время исполнения программы.

В модуле pFIBDataInfo содержатся классовые переменные на каждый тип кэшируемой информации. Для того чтобы очистить интересующую вас информацию нужно выполнить метод Clear. Также некоторые компоненты имеют методы для точечной очистки кеша, например для определенной таблицы или определенного TpFIBDataSet или даже по коду DataSet_ID из репозитория.

```
ListTableInfo :TpFIBTableInfoCollect;
ListDataSetInfo:TpDataSetInfoCollect;
ListSPInfo :TpStoredProcCollect;
ListErrorMessages:TpErrorMessagesCollect;
```

Кэширование блов-полей

Кэширование блов-полей на клиентской стороне – это еще одна уникальная особенность FIBPlus. При включении *BlobSwapSupport.Active := True*, FIBPlus будет автоматически сохранять полученные BLOB-поля в указанном каталоге (свойство *SwapDir*). По умолчанию

свойство `SwapDir` принимает значение равное `{APP_PATH}`, то есть, равное каталогу, в котором находится исполняемое приложение. При необходимости, можно указать также подкаталог, в котором будут сохраняться BLOB-поля. Например, `SwapDir := '{APP_PATH}' + '\BLOB_FILES'`

В компоненте `TrFIBDatabase` для поддержки работы с данным свойством существуют 4 события:

```
property BeforeSaveBlobToSwap: TBeforeSaveBlobToSwap;
property AfterSaveBlobToSwap: TAfterSaveLoadBlobSwap;
property AfterLoadBlobFromSwap: TAfterSaveLoadBlobSwap;
property BeforeLoadBlobFromSwap: TBeforeLoadBlobFromSwap;
```

где

```
TBeforeSaveBlobToSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; Stream: TStream; var FileName: string; var
CanSave: boolean) of object;
TAfterSaveLoadBlobSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; const FileName: string) of object;
TBeforeLoadBlobFromSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; var FileName: string; var CanLoad: boolean)
of object;
```

Обработчики служат для более гибкого управления процессом сохранения и чтения BLOB-полей с диска. В частности, в обработчиках перед сохранением BLOB-поля можно запретить сохранение конкретного BLOB-поля в зависимости от имени таблицы и поля, значений других полей записи, свободного места на диске и т.д. Регулировать процесс сохранения можно и при помощи свойства `MinBlobSizeToSwap`, в котором можно задать минимальный размер BLOB-полей, которые будут сохраняться на диске.

Технология имеет ряд ограничений:

1. Таблица **должна иметь первичный ключ**.
2. Чтение BLOB-полей должна производиться компонентом `TrFIBDataSet`.
3. Приложение само должно следить за свободным местом на диске. В частности, такую функцию можно реализовать в обработчике события `BeforeSaveBlobToSwap`.
4. К сожалению, после `backup/restore` базы данных внутри базы данных происходит замена всех `BLOB_ID`, а потому локальный кэш теряет актуальность и автоматически вычищается. Необходимо пояснить, что при очередном подключении вашего приложения к базе данных, FIBPlus автоматически запускает специальный поток, который в отдельном подключении проверяет весь кэш на диске на наличие соответствующих BLOB-полей в базе данных. Для отсутствующих BLOB-полей файлы сразу же удаляются.

Обработка потери соединения

FIBPlus предоставляет своим пользователям уникальную возможность обработки потери соединения. Для обработки потери соединения используется сам компонент `TrFIBDatabase` и компонент централизованной обработки всех ошибок библиотеки `TrFIBErrorHandler`.

Пример использования этой функциональности представлен в примере `ConnectionLost`. Коротко опишем, как он устроен. Компонент `TrFIBDatabase` реализует три специальных события:

AfterRestoreConnect – возникает при успешном восстановлении соединения.

OnLostConnect – возникает при потере соединения. Событие возникает в момент очередного обращения к БД, которое завершается ошибкой. Здесь вы можете задать одно из трех возможных действий, которые можно предпринять в этой ситуации (смотрите описание *TOnLostConnectActions*) - закрыть приложение, закрыть соединение, проигнорировать, попытаться восстановить соединение.

OnErrorRestoreConnect – возникает при очередной ошибке попытки восстановления соединения.

При потере соединения пользователю предоставляется выбор, какое из действий предпринять. В случае успеха выдается сообщение, что соединение восстановлено. При очередной ошибке восстановления соединения мы можем посчитать попытки в нашем коде и предпринимать какие-то иные действия в случае необходимости.

На компоненте *TrFIBErrorHandler* мы остановимся в отдельном разделе, сейчас же скажем, что обработчик ошибок при возникновении потери соединения просто подавляет стандартную реакцию на ошибку.

```

procedure TForm1.dbAfterRestoreConnect(Database: TFIBDatabase);
begin
  MessageDlg('Connection restored', mtInformation, [mbOk], 0);
end;

procedure TForm1.dbErrorRestoreConnect(Database: TFIBDatabase;
  E: EFIBError; var Actions: TOnLostConnectActions);
begin
  Inc(AttemptRest);
  Label4.Caption:=IntToStr(AttemptRest);
  Label4.Refresh
end;

procedure TForm1.dbLostConnect(Database: TFIBDatabase; E: EFIBError;
  var Actions: TOnLostConnectActions);
begin
  case cmbKindOnLost.ItemIndex of
    0: begin
      Actions := laCloseConnect;
      MessageDlg('Connection lost. TrFIBDatabase will be closed!',
        mtInformation, [mbOk], 0);
      end;
    1:begin
      Actions := laTerminateApp;
      MessageDlg('Connection lost. Application will be closed!',
        mtInformation, [mbOk], 0
      );
      end;
    2:Actions := laWaitRestore;
  end;
end;

procedure TForm1.pFibErrorHandler1FIBErrorEvent(Sender: TObject;
  ErrorValue: EFIBError; KindIBError: TKindIBError; var DoRaise: Boolean);
begin
  if KindIBError = keLostConnect then begin
    DoRaise := false;
  end;
end;

```

Другие полезные методы TrFIBDatabase

Компонент TrFIBDatabase реализует множество полезных методов. Перечислим некоторые из них, которые, на наш взгляд, используются наиболее часто.

Выполнение простых SQL-запросов

Если Вам нужно выполнить какой-либо простой SQL-запрос для получения или установки каких-либо параметров необходимых для работы приложения можно воспользоваться следующими методами:

```
function Execute(const SQL: string): boolean;
```

- выполняет SQL-запрос, который передали в параметре SQL, и возвращает истину в случае успеха.

```
function QueryValue(const aSQL: string; FieldNo: integer; ParamValues: array of variant; aTransaction: TFIBTransaction=nil): Variant; overload;
```

- получает значение поля с индексом FieldNo как результат выполнения aSQL в транзакции aTransaction. Транзакцию можно не указывать, тогда будет использована транзакция DefaultTransaction. Можно передать в запрос параметры. Значение возвращается в виде переменной типа Variant. Используя похожий метод QueryValueAsStr можно получить значение в виде строки, а при помощи QueryValues получить вариантный массив значений. Запомните, что ваш SQL в этом случае должен возвращать не более одной строки.

Получение значений генераторов

Используйте метод для получения значение генератора

```
function Gen_Id(const GeneratorName: string; Step: Int64; aTransaction: TFIBTransaction = nil): Int64;
```

Получение информации о таблицах и полях

```
procedure GetTableNames(TableNames: TStrings; WithSystem: Boolean);
```

```
procedure GetFieldNames(const TableName: string; FieldNames: TStrings; WithComputedFields: Boolean = True);
```

Первый метод получает имена всех таблиц и заполняет ими список TableNames. Параметр WithSystem указывает, извлекать ли имена системных таблиц.

Второй метод получает имена полей таблицы TableName и заполняет ими список FieldNames. Параметр WithComputedFields указывает, извлекать ли вычисляемые поля, описанные в БД как «COMPUTED BY».

Получение информации о версии сервера

```
property ServerMajorVersion: integer;
```

```
property ServerMinorVersion: integer;
```

```
property ServerBuild: integer;
```

```
property ServerRelease: integer;
```

Названия свойств говорят сами за себя.

Получение информации о состоянии коннекта

```
property AttachmentID: Long;
```

Идентификатор коннекта. В Firebird начиная с версии 1.5 его можно так же получить из контекстной серверной переменной CURRENT_CONNECTION.

```
property Busy:boolean;
```

Возвращает true, если в рамках данного соединения в данный момент выполняется вызов IB API. Имеет смысл анализировать только из параллельного треда.

Работа с контекстными переменными Firebird 2.

```
function GetContextVariable (ContextSpace:TFBContextSpace;  
  const VarName:string;aTransaction:TFIBTransaction=nil  
):Variant;  
  
procedure SetContextVariable (ContextSpace:TFBContextSpace;  
  const VarName,VarValue:string;aTransaction:TFIBTransaction=nil  
);
```

Через вышеуказанные методы можно работать с пользовательскими и системными контекстными переменными начиная с версии Firebird2. Подробнее о этих переменных смотрите в документации к серверу, файл «**README.context_variables2.txt**».

Пример использования:

```
procedure TForm1.Button3Click(Sender: TObject);  
begin  
  ShowMessage (pFIBDatabase1.GetContextVariable (csSystem, 'CLIENT_ADDRESS'));  
  pFIBDatabase1.SetContextVariable (csSession, 'TAG', 'Tag=1');  
  ShowMessage (pFIBDatabase1.GetContextVariable (csSession, 'TAG'));  
end;
```

Прерывание выполнения запросов в Firebird начиная с версии 2.1.

Для прекращения выполнения «долгоиграющих» запросов можно воспользоваться методом TFIBDatabase:

```
procedure CancelOperationFB21 (ConnectForCancel:TFIBDatabase=nil);
```

Если сам запрос выполняется в параллельном треде, то метод должен быть вызван из главного. Если же прерываемый запрос выполняется в главном, то метод вызывается из параллельного. Метод использует системную таблицу «MON\$STATEMENTS» и выполняется в параллельном соединении, которое ему передается в качестве входного параметра. Если входного параметра нет, то дополнительное соединение будет автоматически создано и закрыто после выполнения метода. Использование этой технологии можно посмотреть в демонстрационном примере BackgroundQuery.

Прерывание выполнения запросов в Firebird начиная с версии 2.5.

Для решения той же самой задачи, разработчики сервера ввели дополнительную возможность на уровне API сервера, начиная с версии Firebird 2.5. Подробнее смотрите в документации к серверу файл «**README.fb_cancel_operation.txt**».

В FIBPlus эта возможность реализуется набором методов TFIBDatabase

```
procedure RaiseCancelOperations;  
procedure EnableCancelOperations;  
procedure DisableCancelOperations;
```

Свойство TDesignDBOptions.

Служит для управления поведением компонента под дизайном. Возможный набор опций: (ddoIsDefaultDatabase, ddoStoreConnected, ddoNotSavePassword)

Значения опций:

`ddoIsDefaultDatabase` – данный компонент будет под дизайном использоваться как Database по умолчанию. Выражается это в том, что для вновь создаваемых компонент типа: TFIBTransaction, TFIBQuery, TFIBDataset свойства DefaultDatabase и Database будут сразу же заполнены ссылкой на данный компонент.

`ddoStoreConnected` – Указывает надо ли сохранять свойство Connected установленное под дизайном. Если эта опция выключена, то свойство Connected сохранено не будет и при запуске откомпилированной программы не будет происходить попытки автоматического соединения

`ddoNotSavePassword` – указывает нужно ли сохранять заполненный под дизайном пароль в ресурсных файлах. Если опция включена, то пароль сохранен не будет.

Работа с транзакциями

Транзакция - это операция перевода БД из одного непротиворечивого состояния в другое непротиворечивое состояние.

Все действия, выполняемые при работе с БД (получение, либо изменение данных или метаданных) осуществляются в контексте некоторой транзакции. И понимание работы `TrFIBTransaction` является ключевым моментом для понимания тонкостей работы FIBPlus. Поэтому, мы настоятельно рекомендуем обратиться к разделу «Working with Transaction» `ApiGuide.pdf` документации InterBase.

Все изменения, выполненные в контексте транзакции, можно либо подтвердить - `Commit` (при отсутствии ошибок), либо отменить - `Rollback`. В дополнение к этим базовым методам в `TrFIBTransaction` реализованы аналогичные методы с сохранением контекста - `CommitRetaining` и `RollbackRetaining`.

Для запуска транзакции нужно вызвать метод `StartTransaction` компонента или установить свойство `Active` в `True`. Для подтверждения транзакции нужно вызывать метод `Commit/CommitRetaining`, а для отмены - метод `Rollback/RollbackRetaining`. Такие компоненты как `TrFIBQuery` и `TrFIBDataSet` имеют свои свойства, которые позволяют не заботиться об управлении транзакциями. В частности, свойство `TrFIBDataSet.AutoCommit`, параметр `poStartTransaction` свойства `TrFIBDataSet.Options`, а также параметры `qoStartTransaction` и `qoCommitTransaction` в свойстве `TrFIBQuery.Options`.

Настройка параметров транзакции

Параметры транзакций очень серьезная тема, выходящая за рамки данного документа. Мы настоятельно рекомендуем ознакомиться с документацией по InterBase. Тем не менее, в большинстве ситуаций нет необходимости вникать во все тонкости управления транзакциями на уровне API, поэтому FIBPlus реализует ряд механизмов для упрощения работы программиста. В частности, `TrFIBTransaction` реализует три встроенных типа транзакции `tpbDefault`, `tpbReadCommitted`, `tpbRepeatableRead`. Вы также можете создать свои специфические типы в `design-time` в редакторе компонента `TrFIBTransaction` (рисунок 2) и использовать их таким же образом, как и встроенные. Указав тип транзакции, вы задаете ее параметры:

tpbDefault – параметры должны быть объявлены в `TRParams`

tpbReadCommitted – уровень изоляции `ReadCommitted`

tpbRepeatableRead – уровень изоляции `RepeatableRead`

Планирование использования транзакций в приложениях

Эффективность работы InterBase в большой степени достигается правильным использованием транзакций в приложениях. Версионная архитектура такова, что обновляющие транзакции удерживают версии записей. Поэтому в общем случае нужно стараться, чтобы ваши обновляющие транзакции были как можно короче. Транзакции только для чтения могут оставаться открытыми сколько угодно, поскольку они версий не удерживают.

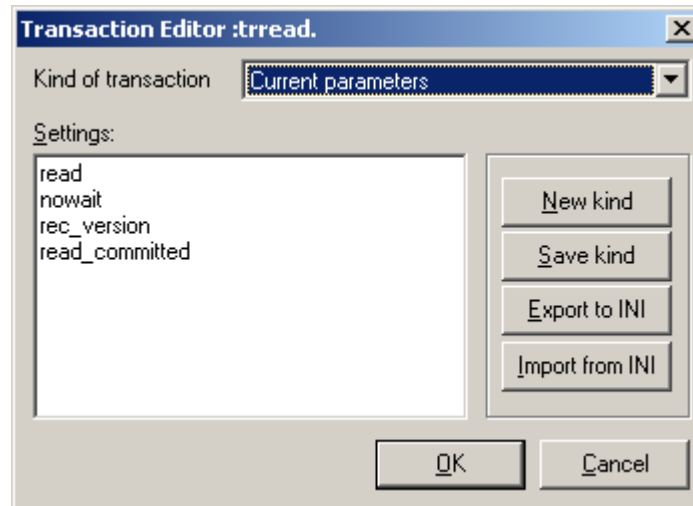


Рисунок 2. Диалог Transaction Editor

Использование SavePoints

Сервера InterBase и их клоны не поддерживают вложенных транзакций. Но InterBase 7.X и Firebird 1.5 поддерживают средства создания контрольных точек сохранения SavePoints и возврата к любой из точек. FIBPlus поддерживает эту функциональность при помощи трех методов:

```

procedure SetSavePoint(const SavePointName:string) ;
procedure RollBackToSavePoint(const SavePointName:string) ;
procedure ReleaseSavePoint(const SavePointName:string) ;

```

Первый метод устанавливает точку восстановления с именем SavePointName. Вторым откатывает транзакцию до состояния в точке SavePointName. Третий освобождает ресурсы сервера, связанные с точкой восстановления SavePointName.

Свойство Timeout

```

property Timeout: Cardinal;
property TimeoutAction: TtransactionAction;

```

Если значение свойства **Timeout** отлично от нуля, то оно определяет время допустимого простоя в миллисекундах. Если за указанный промежуток времени через транзакцию не прошло ни одного действия (запроса, фетча данных, фетча блока), то она будет автоматически закрыта. Способ закрытия такой транзакции указывается в свойстве **TimeoutAction**. По умолчанию свойству **Timeout** присвоено значение 0, т.е. по умолчанию транзакция не закрывается автоматически при простое.

Двухфазный коммит

Одной из особенностей IB/FB серверов является возможность выполнения одной транзакции в нескольких соединениях одновременно. Эта возможность реализуется механизмом двухфазного коммита (2PC). Мы не будем здесь подробно останавливаться на серверной части реализации этой особенности, остановимся на том как эту возможность использовать с помощью FIBPlus. Рекомендуемым способом является использование методов TPFIBTransaction:

```
function AddDatabase(db: TFIBDatabase):integer;  
function AddDatabase(db: TFIBDatabase; const aTRParams: string);
```

Пример использования:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  pFIBTransaction1.AddDatabase(pFIBDatabase1);  
  pFIBTransaction1.AddDatabase(pFIBDatabase2, 'write'#13#10'nowait'#13#10+  
    'concurrency'  
  );  
  pFIBTransaction1.StartTransaction;  
end;
```

В первой строке этого кода мы указали транзакции, что она будет выполняться в контексте соединения pFIBDatabase1. Поскольку параметров транзакции мы не указали, они будут браться из свойства pFIBTransaction1.TRParams. Во второй строке мы указали что транзакция будет также выполняться и в контексте соединения pFIBDatabase2, но параметры для этого соединения указали явно. При выполнении следующей строки

```
pFIBTransaction1.StartTransaction;
```

транзакция стартует в обоих указанных соединениях, причем в каждом из них имеет свой собственный набор параметров.

Немедленное исполнение запроса (ExecSQLImmediate)

Метод TFIBTransaction.ExecSQLImmediate служит для немедленного выполнения запроса, не разбивая это выполнение на стадии препарации, выполнения и освобождения хендла запроса. Служит оберткой вокруг API функции isc_dsqli_execute_immediate. Имеет смысл использовать для однократно выполняемых запросов. Например, DDL запросов.

Получение информации о состоянии транзакции

```
property Active: Boolean;  
property InTransaction: Boolean;
```

Свойства показывают активна ли транзакция.

```
property TransactionID: integer;
```

Идентификатор транзакции. В **Firebird** начиная с версии **1.5** его можно так же получить из контекстной серверной переменной `CURRENT_TRANSACTION`.

```
property State: TTransactionState;
```

Где

```
TTransactionState =
(tsActive, tsClosed, tsDoRollback, tsDoRollbackRetaining, tsDoCommit,
tsDoCommitRetaining);;
```

Свойство позволяет определить не только активность транзакции, но и находится ли она в состоянии завершения. Анализировать это состояние можно в обработчиках, которые вызываются при завершении транзакции. Например :

```
procedure TForm1.pFIBDataSet1TransactionEnded(Sender: TObject);
begin
  with pFIBDataSet1 do
    if UpdateTransaction.State in [tsDoRollBack, tsDoRollBackRetaining] then
      if HaveRollbackedChanges then
        CloseOpen(False)    ;
end;
```

В этом примере, мы анализируем чем завершилась `UpdateTransaction`. Если она завершилась откатом, мы анализируем имеются ли в датасете изменения, которые были отменены на сервере. Если имеются – переоткрываем запрос.

Выполнение SQL- запросов (компонент - TrFIBQuery)

Взаимодействие приложения с базой данных заключается в выполнении SQL-операторов. Они используются для получения и модификации данных и метаданных. В FIBPlus за выполнение SQL-операторов отвечает компонент `TrFIBQuery`. Это быстрый, легкий и вместе с тем мощный компонент для выполнения любых действий над БД.

Использовать компонент очень просто. Достаточно заполнить свойства `Database` и `Transaction`, после чего заполнить свойство `SQL` и вызвать один из методов `ExecQuery` (`ExecQueryWP`, `ExecQueryWPS`). Следующий пример кода демонстрирует, как создать компонент `TrFIBQuery` динамически в `run-time` и получить с его помощью данные о клиентах.

```
var sql: TrFIBQuery;

sql := TrFIBQuery.Create(nil);
with sql do
  try
    Database := db;
    Transaction := db.DefaultTransaction;
    SQL.Text := 'select first_name, last_name from customer';
```

```

ExecQuery;
while not Eof do
begin
    Mem1.Lines.Add(
        FldByName['FIRST_NAME'].AsString+' '+
        FldByName['LASTST_NAME'].AsString);
    Next;
end;
sql.Close;
finally
    sql.Free;
end;

```

Методы и свойства компонента - TrFIBQuery

Свойства доступные под дизайном (published)

property Conditions: TConditions;

Подробно описано ниже.

property GotoToFirstRecordOnExecute:boolean

Указывает фетчить ли первую запись сразу же после выполнения запроса.

property ParamCheck:boolean

Указывает производить парсинг текста запроса, для создания списка параметров, или нет. Если запрос не содержит параметров, то отключение этого свойства позволит сэкономить немного времени. Если же запрос является DDL запросом, то крайне желательно это свойство установить в false.

property Options: TrFIBQueryOptions;

где TrFIBQueryOptions является множеством следующих опций:

qoStartTransaction - если включена, то перед выполнением запроса, если транзакция неактивна, то она будет стартовать автоматически.

qoAutoCommit - если включено, то сразу же после выполнения запроса транзакция, в рамках которой он был выполнен, будет завершена методом Commit. Внимание, если запрос селективный, то он тоже будет сразу же закрыт, и вы не сможете получить доступ к следующим записям.

qoTrimCharFields - определяет способ работы с CHAR-VARCHAR полями. Если опция включена, то методы Fields.asString, Fields.asWideString будут возвращать значения без хвостовых пробелов.

qoFreeHandleAfterExecute- указывает, что Handle запроса должен быть немедленно освобожден после выполнения в случае если запрос не селективный. Если запрос селективный, то Handle будет освобожден либо после выполнения метода Close, либо после того как все записи будут выбраны методом Next

qoNoForceIsNull- определяет надо ли преобразовывать текст SQL в случае использования параметров с NULL значениями. Например, допустим есть такой запрос:

```
Delete from table1 where Field1=:Field1
```

Если параметр Field1 принимает значение отличное от NULL, то этот запрос удалит все записи из таблицы, которые подпадают под условие запроса. Если же параметр принимает значение NULL, то запрос не удалит ни одной записи, так как для NULL значений условие всегда будет возвращать false. Для того чтоб удалить записи в

которых поле Field1 принимает значение NULL требуется переписать запрос как

```
Delete from table1 where Field1 IS NULL.
```

Т.е. разработчику необходимо отслеживать этот момент и менять текст запроса, в зависимости от значения параметра. FIBPlus – упрощает эту работу, и изменяет текст запроса автоматически, в зависимости от текущего значения параметра. Если эта возможность разработчику не нужна, он может ее отключить вышеупомянутой опцией. Т.е. включение опции `qoNoForceIsNull` в `Options` компонента `TrFIBQuery` отключит данное преобразование.

Обработчики событий

```
property BeforeExecute:TNotifyEvent;
property AfterExecute :TNotifyEvent;
```

Вызываются до и после выполнения запроса, соответственно. В обработчике `BeforeExecute` удобно изменять текст SQL, или заполнять значения параметров. В обработчике `AfterExecute` – анализировать результат выполнения запроса.

```
property OnExecuteError:TFIBQueryErrorEvent;
```

Вызывается в случае, если выполнение запроса завершилось ошибкой. В нем можно проанализировать ошибку и подменить стандартный диалог об ошибке, какими-то другими действиями.

```
property TransactionEnding:TNotifyEvent;
```

Вызывается непосредственно перед тем как транзакция, в рамках которой выполнялся запрос, будет завершена.

```
property TransactionEnded:TNotifyEvent;
```

Вызывается сразу же после того как транзакция, в рамках которой выполнялся запрос, завершилась.

```
property OnBatching :TOnBatching;
```

```
property OnBatchError :TOnBatchError;
```

Вызываются только если `rFIBQuery` использовались для пакетных операций. Подробнее рассмотрим эти события, при рассмотрении методов

```
function BatchInput(InputObject: TFIBBatchInputStream) :boolean;
function BatchOutput(OutputObject: TFIBBatchOutputStream):boolean;
procedure BatchToQuery(ToQuery:TFIBQuery;Mappings:TStrings);
```

Публичные методы и свойства

Подготовка и выполнение запроса.

```
procedure Prepare;
```

Производит подготовку запроса к выполнению. Если этот метод не вызван явно, то он будет вызван автоматически во время выполнения запроса.

```
procedure ExecQuery;
```

Запускает выполнение подготовленного(отпрепарированного) запроса.

```
procedure ExecuteImmediate;
```

Запускает выполнение запроса без предварительной препарации. Запросы такого рода, не имеют своего Handle и не требуют его последующего освобождения. Этим методом можно выполнять только запросы не имеющие параметров.

```
procedure ExecProcedure(const ProcName:string);
```

Выполняет сохраненную процедуру по имени ProcName без параметров. Т.е. автоматически формируется текст запроса вида :

```
'Execute Procedure '+ ProcName
```

и он запускается на выполнение.

```
procedure ExecProcedure(const ProcName:string;const InputParams:array of variant);
```

Выполняет сохраненную процедуру по имени ProcName с параметрами InputParams.

```
procedure Close;
```

Закрывает селективный запрос.

```
procedure FreeHandle;
```

Освобождает ресурсы сервера ассоциированные с данным запросом.

```
function Next: TFIBXSQLDA;
```

Производит фетч следующей записи для уже выполненного селективного запроса. Если следующей записи нет, то метод возвращает значение **nil**.

Работа с полями запроса

```
property Fields: TFIBXSQLDA;
```

Возвращает результирующие поля запроса.

```
property Fields[const Idx: Integer]: TFIBXSQLVAR;
```

Возвращает ссылку на поле запроса. Параметр Idx указывает на индекс поля в коллекции. 0 – первое поле, 1- второе и т.д.

```
function FieldCount:integer;
```

Возвращает количество полей в запросе.

```
function FieldByName(const FieldName: string): TFIBXSQLVAR;
```

Возвращает ссылку на поле запроса по имени `FieldName`. Если поля с таким именем в коллекции полей запроса отсутствует, то генерируется ошибка.

```
function FindField(const FieldName: string): TFIBXSQLVAR;
```

Возвращает ссылку на поле запроса по имени `FieldName`. Если поля с таким именем в коллекции полей запроса отсутствует, то возвращается значение **nil**.

```
function FieldByOrigin(const TableName, FieldName: string): TFIBXSQLVAR;
```

Возвращает ссылку на поле запроса по имени таблицы, которой оно принадлежит (`TableName`) и имени поля в таблице (`FieldName`). Если такого поля в коллекции полей запроса отсутствует, то возвращается значение **nil**. Следует так же обратить внимание, что для диалекта 3, все имена таблиц и полей регистрочувствительные. Как следствие, в этот метод надо передавать имена таблицы и поля в том регистре, в котором они созданы в БД.

Рассмотрим вышеупомянутые функции на примере нижеследующего кода:

```
var
  F: TFIBXSQLVAR
begin
  pFIBQuery1.SQL.Text:='Select  Field1 F1 from Table1 T';
  pFIBQuery1.ExecQuery;
  F:= pFIBQuery1.Fields[0];
  F:= pFIBQuery1.FieldByName('F1');
  F:= pFIBQuery1.FindField ('F1');
  F:= pFIBQuery1.FieldByOrigin('TABLE1', 'FIELD1');
end;
```

Во всех случаях, в переменную `F` попадает ссылка на поле `F1` запроса. Обратите внимание, что в последнем вызове мы передавали имя поля и таблицы в верхнем регистре.

Работа с параметрами запроса

```
property Params: TFIBXSQLDA;
```

Возвращает параметры запроса.

```
function ParamCount: integer;
```

Возвращает количество параметров в запросе.

```
property Params[const Idx: Integer]: TFIBXSQLVAR;
```

Возвращает ссылку на параметр запроса. Параметр `Idx` указывает на индекс параметра в коллекции. 0 – первый параметр, 1- второй и т.д.

```
function ParamByName(const ParamName: string): TFIBXSQLVAR;
```

Возвращает ссылку на параметр запроса по имени `ParamName`. Если параметра с таким именем в коллекции параметров запроса отсутствует, то генерируется ошибка.

```
function FindParam(const ParamName: string): TFIBXSQLVAR;
```

Возвращает ссылку на параметр запроса по имени `ParamName`. Если параметра с таким именем в коллекции параметров запроса отсутствует, то возвращается значение **nil**.

Заполнение параметров запроса.

Перед тем как запрос будет выполнен, необходимо заполнить параметры запроса актуальными значениями. Для этого есть несколько возможностей.

Возможность 1: Можно получить ссылку на параметр, и заполнить значение методами TFIBXSQLVAR.

```
var
P:TFIBXSQLVAR
begin
  pFIBQuery1.SQL.Text:='Select Field1 F1 from Table1 T where F2=:F2';
  P:= pFIBQuery1.Params[0];
  P:= pFIBQuery1.ParamByName('F2');
  P:= pFIBQuery1.FindParam('F2');
  P.Value:=1;
  pFIBQuery1.ExecQuery;
end;
```

Все три присвоения в переменную P в этом примере, дадут один и тот же результат.

Возможность 2: Можно так же воспользоваться методами TrFIBQuery

```
procedure SetParamValues(const ParamValues: array of Variant);
procedure SetParamValues(const ParamNames: string;ParamValues: array of
Variant);
```

Например :

```
begin
  pFIBQuery1.SQL.Text:='Select Field1 F1 from Table1 T where F2=:F2 and F3=:F3';
  //1 способ
  pFIBQuery1.SetParamValues([1,2]);
  pFIBQuery1.ExecQuery;
  //2 способ
  pFIBQuery1.SetParamValues(['F1', 'F2'], [1,2]);
  pFIBQuery1.ExecQuery;
end;
```

Возможность 3: Можно так же воспользоваться методами TrFIBQuery

```
procedure ExecWP(const ParamValues: array of Variant);
procedure ExecWP(const ParamNames: string;ParamValues: array of Variant);
```

В отличие от метода SetParamValues метод ExecWP не только заполняет параметры, но и вызывает исполнение запроса.

Возможность 4: Можно так же воспользоваться методами TrFIBQuery

```
procedure ExecWPS(ParamSource:ISQLObject; AllRecords:boolean=True);
procedure ExecWPS(const ParamSources: array of ISQLObject);
```

При использовании этих методов, в качестве источников для значений полей выступают объекты – носители интерфейса ISQLObject. Носителями этого интерфейса являются компоненты pFIBDataSet, pFIBQuery, pFIBClientDataSet. Сам интерфейс описан в юните pFIBInterfaces, и в принципе он может быть имплементирован в любых сторонних компонентах. Рассмотрим применение этих методов на примере. Допустим нам нужно загрузить данные из xml файла и все их вставить в таблицу БД.

```

begin
  pFIBClientDataSet1.LoadFromFile('D:\File1.xml');
  pFIBQuery1.SQL.Text:='Insert into Table1 (ID,Name) Values (:ID, :NAME)';
  pFIBQuery1.ExecWPS (pFIBClientDataSet1);
end;

```

В первой строчке этого примера, мы загружаем xml файл в pFIBClientDataSet1. Во второй, формируем запрос на вставку записей в таблицу. Третьей строкой мы отправляем все данные в таблицу. При этом для каждой записи будет происходить следующее: Значение поля ID из pFIBClientDataSet1 будет присвоено в параметр ID компонента pFIBQuery1, аналогично поле NAME передаст свое значение параметру NAME. После выполнения запроса на вставку, происходит переход на следующую запись и действия повторяются. В случае возникновения ошибки при операции вставки, вызовется событие OnBatchError, в котором можно эту ошибку обработать и принять решение о продолжении или прекращении дальнейшего выполнения процедуры.

Получение дополнительной информации о запросе.

```
property SQLType: TFIBSQLTypes;
```

где

```

TFIBSQLTypes = (SQLUnknown, SQLSelect, SQLInsert,
                SQLUpdate, SQLDelete, SQLDDL,
                SQLGetSegment, SQLPutSegment,
                SQLExecProcedure, SQLStartTransaction,
                SQLCommit, SQLRollback,
                SQLSelectForUpdate, SQLSetGenerator, SQLSavePointOperation
                );

```

Свойство возвращает тип запроса. Внимание: запрос должен быть уже подготовлен методом Prepare.

```
property SQLKind: TSQLKind;
```

где

```

TSQLKind=(skUnknown, skSelect, skUpdate, skInsert, skDelete, skExecuteProc, skDDL,
skExecuteBlock);

```

Еще одно свойство возвращающее тип запроса. В отличие от SQLType, оно доступно сразу же при присвоении текста SQL.

```
property Prepared: Boolean;
```

Возвращает true, если запрос уже подготовлен методом Prepare;

```
property Plan: string;
```

Возвращает текст плана, по которому будет выполняться, или уже выполнен запрос.

(Запрос должен быть уже подготовлен методом Prepare)

```
property RecordCount: Integer;
```

Возвращает количество отфетченных на данный момент записей. (Только для селективных запросов)

```
property RowsAffected: Integer;
```

Возвращает количество записей на которые воздействовал запрос. (Только для модифицирующих запросов)

```
property AllRowsAffected: TAllRowsAffected;
```

где

```
TAllRowsAffected =
record
  Updates: integer;
  Deletes: integer;
  Selects: integer;
  Inserts: integer;
end;
```

Возвращает количество записей на которые воздействовал запрос, с подразбивкой на операции. (Только для модифицирующих запросов)

```
property Bof: Boolean;
```

Показывает были ли уже фетчи записей из запроса.

```
property Eof: Boolean;
```

Возвращает true если достигнут конец набора записей и все записи отфетчены.

```
property CallTime : Cardinal;
```

Возвращает количество времени в тиках, за которое выполнялся запрос. (Время фетча сюда не входит)

Работа с текстом SQL

```
property SQL : TStrings;
```

Хранит текст SQL заданный разработчиком.

```
procedure BeginModifySQLText;
procedure EndModifySQLText;
function CountModifySQLText: integer;
```

Следует отметить, что если включено свойство ParamCheck, то как только разработчик меняет текст SQL, то FIBPlus немедленно пытается отпарсить текст, для того чтоб создать список параметров. Вышеописанные методы позволяют отложить процесс парсинга если текст SQL сформировывается не одноразовой операцией, а несколькими. Например:

```
begin
  pFIBQuery1.BeginModifySQLText;
  try
    pFIBQuery1.SQL.Clear;
    pFIBQuery1.SQL.Add('Insert into Table1 (ID,Name)');
    pFIBQuery1.SQL.Add('Values (:ID, :NAME)');
  finally
    pFIBQuery1.EndModifySQLText;
  end
end;
```

В этом примере парсинг запроса произойдет один раз, после вызова pFIBQuery1.EndModifySQLText. Если бы мы не вызывали бы метод BeginModifySQLText перед изменением текста, то парсинг происходил бы после каждой операции.

SQL - секции

FIBPlus предоставляет вам широкие возможности по управлению текстом SQL в ваших запросах. Прежде всего, это секции SQL – список полей, условий, порядка группировки и

сортировки, план выполнения запроса.

Это простые строковые свойства доступные как для чтения, так и для записи:

FieldsClause – содержит список полей;

MainWhereClause – содержит основную секцию WHERE (подробности в разделе ниже)

OrderClause – секция «order by»

GroupByClause – секция «group by»

PlanClause – план.

Рассмотрим применение этих свойств на примере.

```
begin
  sql.SQL.Text := 'select first_name, last_name from customer';
  sql.MainWhereClause := 'first_name starting with :first_name';
  sql.ExecWP(['A'])

  sql.MainWhereClause := 'last_name starting with :last_name';
  sql.ExecWP(['A'])
end;
```

В первых двух строках этого примера формируется текст запроса, который в итоге будет выглядеть как:

```
select first_name, last_name from customer
where first_name starting with :first_name
```

После выполнения запроса, мы присваиваем в MainWhereClause новое условие запроса и FIBPlus преобразует текст запроса в

```
select first_name, last_name from customer
where last_name starting with :last_name
```

Также в FIBPlus реализованы такие уникальные возможности, как макросы и conditions - расширенный механизм работы с секцией where. Остановимся на них подробно.

Макросы

Механизм макросов позволяет вам управлять вариационной частью ваших запросов. Вы можете менять и уточнять их смысл без переписывания текста запроса.

Макрос выглядит следующим образом @@<MACROS_NAME>[%<DEFAULT_VALUE>|#]@

Т.е., макрос - это специфическая последовательность символов между знаками @@ и @. После @@ следует обязательный параметр – имя макроса. Можно также задать значение макроса по умолчанию, которое отделяется от имени символом %. Есть возможность задать необязательный параметр #, который будет говорить FIBPlus, что значение макроса нужно заключить в кавычки.

Использование макросов очень похоже на использование параметров. Продемонстрируем это:

```
Sql.SQL.Text := 'select * from @@table_clause@ where @@where_clause% 1=1@';
Sql.ExecWP(['CUSTOMER', 'FIRST_NAME STARTING WITH 'A']);
```

Для того чтобы установить значение макроса в значение по умолчанию, нужно вызвать метод SetDefaultMacroValue объекта-параметра. Например, команда

```
Sql.Params[1].SetDefaultMacroValue;
```

присвоит макросу 'where_clause' значение '1=1'.

Макрос может также содержать параметр. В этом случае для поиска параметра необходимо использовать функцию FindParam, а для установки параметра использовать методом ParamByName:

```
Sql.SQL.Text := 'select * from @@table_clause@ where @@where_clause% 1=1@';
Sql.Params[0].AsString := 'CUSTOMER';
Sql.Params[1].AsString := 'CUST_NO = :CUST_NO';
if Assigned(Sql.FindParam('CUST_NO')) then
    Sql.ParamByName('CUST_NO').AsInteger := 1001;
Sql.ExecQuery;
```

Использование макросов для TrFIBDataSet показано в демонстрационном примере ServerFilterMarcoses.

Условия

Механизм условий (conditions) - это еще одна возможность для изменения вариативной части ваших SQL-запросов.

Для любого SQL в design-time (рисунок 3) можно задать один или несколько параметров-условий. Для этого удобно пользоваться встроенным диалогом, приведенным на рисунке

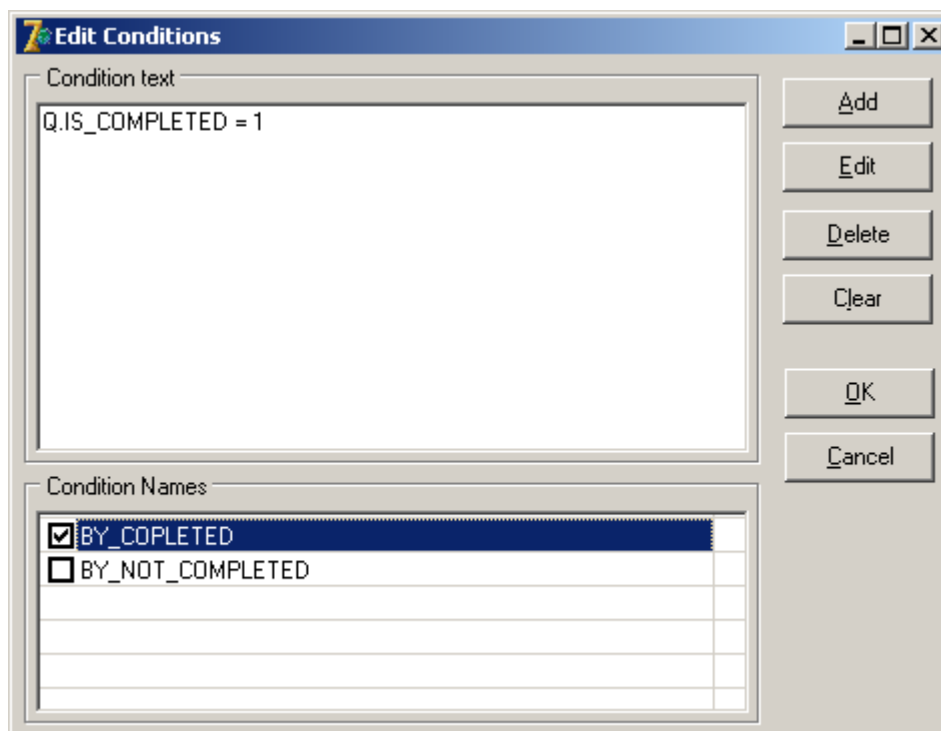


Рисунок 3. Диалог Edit Conditions.

Чтобы

включить условие достаточно установить его свойство Active в True.

```
pFIBQuery1.Conditions[0].Active := True;
```

или

```
pFIBQuery1.Conditions.By_name('by_customer').Active := True;
```

Код для работы с Conditions мог бы выглядеть следующим образом:

```
if pFIBQuery1.Open then pFIBQuery1.Close;
pFIBQuery1.Conditions.CancelApply;
pFIBQuery1.Conditions.Clear;
if byCustomerFlag then
  pFIBQuery1.Conditions.AddCondition('by_customer', 'cust_no = 1001', True);
pFIBQuery1.Conditions.Apply;
pFIBQuery1.Open;
```

В TrFIBDataSet реализованы два дополнительных метода – CancelConditions и ApplyConditions, которые вызывают, соответственно, Conditions.Cancel и Conditions.Apply. Код для TrFIBDataSet выглядит немного проще.

```
with pFIBDataSet1 do begin
  if Active then Close;
  CancelConditions;
  Conditions.Clear;
  if byCustomerFlag then Conditions.AddCondition('by_customer', 'cust_no =
1001', True);
  ApplyConditions;
  Open;
end;
```

Использование условий для TrFIBDataSet продемонстрировано в примере ServerFilterConditions

Пакетная обработка

FIBPlus содержит встроенные методы для пакетной обработки данных, так называемые Batch-методы. Эти методы могут пригодиться для репликации между базами данных, и при операциях импорта и экспорта.

```
function BatchInput(InputObject: TFIBBatchInputStream) :boolean;
function BatchOutput(OutputObject: TFIBBatchOutputStream):boolean;
procedure BatchInputRawFile(const FileName:string);
procedure BatchOutputRawFile(const FileName:string;Version:integer=1);
procedure BatchToQuery(ToQuery:TFIBQuery; Mappings:TStrings);
```

Параметр Version введен для совместимости форматов со старыми версиями файлов, выгруженных FIBPlus при помощи метода BatchOutputXXX. Если Version = 1, то используется старый принцип, при котором во внешний файл выводятся только данные в порядке, который определяется полями SQL-запроса. Предполагается, что при чтении данных, сохраненных методом BatchInputRawFile, в читающем SQL параметры будут расположены в том же порядке. Количество полей TrFIBQuery, данные которого записывались, и количество параметров в TrFIBQuery, который потом будет считывать данные, должны совпадать. Для строковых полей необходимо, чтобы длина записываемого поля и читающего параметра также совпадали, однако совпадения имен не требуется. Если Version = 2, то используется новый принцип записи. Помимо данных в файл записывается служебная информация о полях (имя, тип и длина). При последующем чтении, данные будут читаться по принципу совпадения имен. Порядок и количество полей в записывающем TrFIBQuery может не совпадать с порядком и количеством параметров в читающем TrFIBQuery. Типы и длина тоже могут не совпадать. Требуется лишь совпадение имен.

Работать с batch-методами очень просто. Покажем это на простом примере. Код,

приведенный ниже, состоит из трех частей. Первая часть сохраняет данные о клиентах во внешний файл, а вторая загружает эти данные в БД. Третья часть показывает, как можно сделать преобразование данных.

```
{ I }
pFIBQuery1.SQL := 'select EMP_NO, FIRST_NAME, LAST_NAME from CUSTOMER';
pFIBQuery1.BatchOutputRawFile('employee_buffer.fibplus', 1);

{ II }
pFIBQuery1.SQL := 'insert into employees(EMP_NO, FIRST_NAME, LAST_NAME)'+
  ' values(:EMP_NO, :FIRST_NAME, :LAST_NAME)';
pFIBQuery1.BatchInputRawFile('employee_buffer.fibplus');

{ III }
pFIBQuery1.SQL := 'select EMP_NO, FIRST_NAME, LAST_NAME from CUSTOMER';
pFIBQuery2.SQL := 'insert into tmp_employees(EMP_NO, FIRST_NAME, LAST_NAME)'+
  ' values(:EMP_NO, :FIRST_NAME, :LAST_NAME)';

mapStrings.Add('EMP_NO=EMP_NO');
mapStrings.Add('FIRST_NAME=FIRST_NAME');
mapStrings.Add('LAST_NAME=LAST_NAME');

pFIBQuery1.BatchToQuery(pFIBQuery2, mapStrings);
```

Мы еще вернемся к пакетной обработке при рассмотрении `TrFIBDataSet`, который также содержит несколько методов пакетной обработки.

При возникновении ошибки выполнения обработки возникает событие `OnBatchError`. В коде этого события, используя параметр `BatchErrorAction` (`TBatchErrorAction = (beFail, beAbort, beRetry, beIgnore)`) можно принять решение о том, что делать в случае ошибки.

Выполнение хранимых процедур

Выполнение процедур, практически не отличается от выполнения запросов. Вы просто пишете в тексте SQL `'execute procedure some_proc(:proc_param)'` либо, для селективных процедур, `'select * from some_proc(:proc_param)'`.

Если неселективная процедура возвращает какие-либо результаты, их можно получить после выполнения запроса через уже известное нам свойство `Fields`.

```
Sql.SQL.Text := 'execute procedure some_proc(:proc_param)';
Sql.ExecWP([25]);
ResultField1 := Sql.Fields[0].AsInteger;
```

Это немного отличается от BDE, ADO и других компонент доступа, где выходные данные доступны также через параметры.

Кроме того, в FIBPlus хранимые процедуры можно выполнять при помощи компонента `TrFIBStoredProc`. Он является прямым наследником `TrFIBQuery`, у которого есть свойство `StoredProcName` и его рекомендуется использовать для выполнения неселективных процедур.

Выполнение DDL операторов.

Кроме выполнения SQL-операторов, компонент `TrFIBQuery` позволяет выполнять DDL операторы. Для этого необходимо установить свойство `ParamsCheck` в `False`. DDL- операции выполняются по одной и никаких разделителей не ставится. В новых версиях для DDL поддержки

ваются макросы.

Повторное использование запросов

Для того чтобы подготовить запрос к выполнению, все клиентские библиотеки, включая FIBPlus, передают на сервер полный текст запроса. Для выполнения же подготовленного запроса достаточно передавать на сервер только Handle и значения параметров. Если в вашей программе часто используются одинаковые запросы, то вы вполне можете организовать их повторное использование при помощи методов модуля из pFIBCacheQueries.pas:

```
function GetQueryForUse (aTransaction: TFIBTransaction; const SQLText: string):  
TpFIBQuery;  
procedure FreeQueryForUse (aFIBQuery: TpFIBQuery);
```

Вам не придется создавать экземпляры TpFIBQuery - процедура GetQueryForUse самостоятельно создаст его при первом вызове, а потом будет возвращать ссылку на существующий компонент, если вы будете выполнять этот же запрос снова и снова. Очевидно, что при каждом последующем вызове будет использоваться уже подготовленный к выполнению запрос, а, следовательно, передача текста запроса на сервер будет выполнена только один раз. После получения результатов запроса из компонента TpFIBQuery необходимо вызвать метод FreeQueryForUse. Данный механизм уже используется для внутренних целей в компонентах FIBPlus, в частности, при вызове генераторов для получения значений первичных ключей. Вы также можете воспользоваться этими методами в своих приложениях для оптимизации трафика.

Работа с наборами данных

И, наконец, мы подошли к работе с наборами данных. Это компонент `TpFIBDataSet`. Он работает на основе компонента `TpFIBQuery` и позволяет кэшировать результаты выборки. `TpFIBDataSet` является наследником `TDataSet` и полностью поддерживает все его свойства, события и методы. Для получения более подробной информации о `TDataSet` смотрите встроенную справку Delphi/C++Builder.

Базовые принципы работы с наборами данных

`TpFIBDataSet` позволяет получать, вставлять, изменять и удалять данные. Все операции реализованы при помощи соответствующих компонентов `TpFIBQuery` в его составе.

Для того чтобы выбрать данные, необходимо заполнить свойство `SelectSQL`. Это равнозначно заполнению свойства `SQL` компонента `QSelect` типа `TpFIBQuery`. Для вставки данных требуется заполнить свойство `InsertSQL.Text`. Аналогично задаются запросы для модификации (`UpdateSQL`), удаления (`DeleteSQL`) и обновления текущей записи (`RefreshSQL`).

Для примера возьмем демонстрационную базу данных `employee.gdb` (`fdb`, если это Firebird). Напишем `SelectSQL` для получения всех служащих, укажем запросы в `InsertSQL` и т.д.

```
with pFIBDataSet1 do
begin
  if Active then Close;
  SelectSQL.Text :=
    'select CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST from CUSTOMER';
  InsertSQL.Text :=
    'insert into CUSTOMER(CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST )'+
    ' values (:CUST_NO, :CUSTOMER, :CONTACT_FIRST, :CONTACT_LAST)';
  UpdateSQL.Text :=
    'update CUSTOMER set CUSTOMER = :CUSTOMER, '+
    'CONTACT_FIRST = :CONTACT_FIRST, CONTACT_LAST = :CONTACT_LAST '+
    'where CUST_NO = :CUST_NO';
  DeleteSQL.Text := 'delete from CUSTOMER where CUST_NO = :CUST_NO';
  RefreshSQL.Text :=
    'select CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST ' +
    'from CUSTOMER where CUST_NO = :CUST_NO';

  Open;
end;
```

Для того, чтобы открыть `TpFIBDataSet`, необходимо выполнить метод `Open` или `OpenWP`, или установить свойство `Active` в `True`. Чтобы закрыть `TpFIBDataSet`, вызовите метод `Close`.

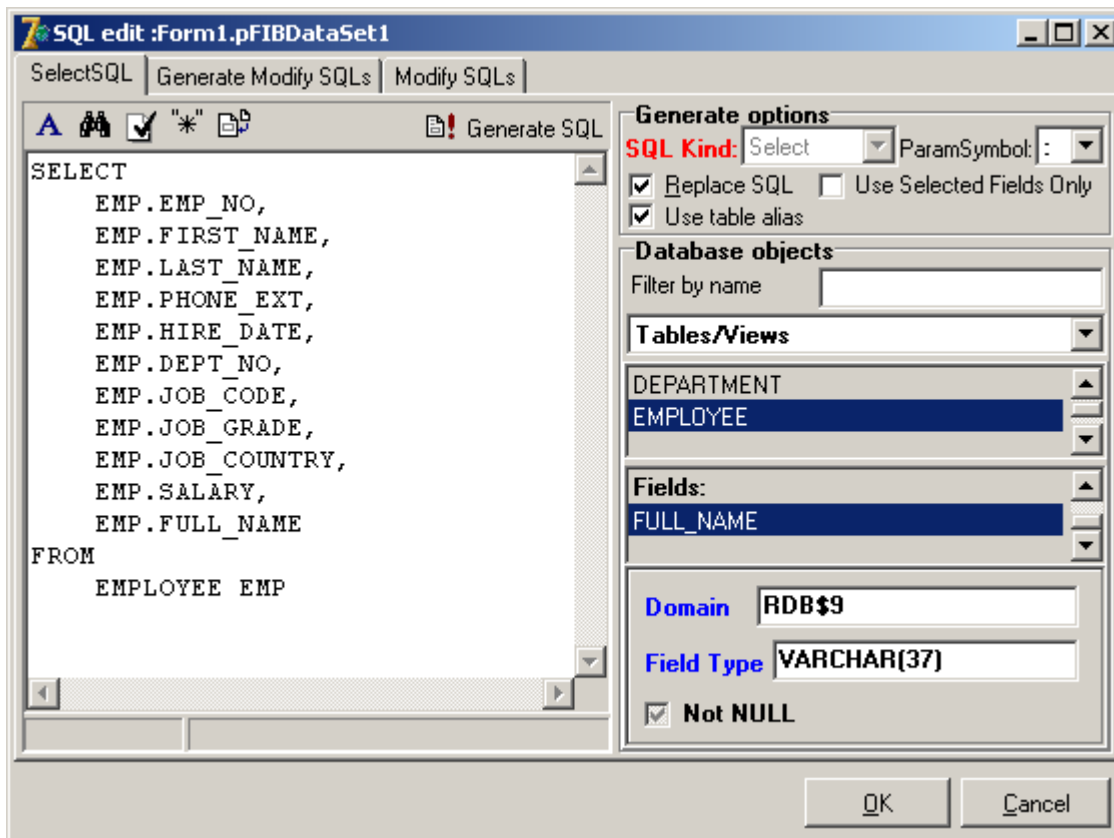


Рисунок 4. Диалог SQL Edit TpFIBDataSet

Пусть вас не пугает количество написанного текста, все эти запросы сделает за вас редактор TpFIBDataSet, который можно вызвать через контекстное меню компонента. Диалог показан на рисунке 4.

Простое приложение, демонстрирующее использование редактируемого TpFIBDataSet, представлено в примере DataSetBasic.

Автоматическая генерация обновляющих запросов

Кроме использования SQL-редактора TpFIBDataSet, FIBPlus может построить все обновляющие запросы в режиме исполнения программы, и сделать это возможно даже эффективнее, чем при генерации запросов в design-time.

Необходимо использовать свойство AutoUpdateOptions. Это очень важная группа настроек, которое можно значительно упростить вам жизнь.

```
TAutoUpdateOptions= class (TPersistent)
  property AutoParamsToFields: Boolean .. default False;
  property AutoRewriteSqls: Boolean .. default False;
  property CanChangeSQLs: Boolean .. default False;
  property GeneratorName: string;
  property GeneratorStep: Integer .. default 1;
  property KeyFields: string;
  property ParamsToFieldsLinks: TStrings;
  property SeparateBlobUpdate: Boolean .. default False;
  property UpdateOnlyModifiedFields: Boolean .. default False;
  property UpdateTableName: string;
  property WhenGetGenID: TWhenGetGenID .. default wgNever;
  property UpdateTableName: string;
  property WhenGetGenID: TWhenGetGenID .. default wgNever;
  property UseExecuteBlock :boolean .. default False;
```

```
property UseReturningFields: TSetReturningFields .. default []  
end;
```

```
TWhenGetGenID= (wgNever, wgOnNewRecord, wgBeforePost);
```

AutoRewriteSQLs при наличии пустых SQLText для InsertSQL, UpdateSQL, DeleteSQL, RefreshSQL будет автоматически производится их генерация на основе свойств SelectSQL, KeyFields и UpdateTableName.

CanChangeSQLs говорит о том, что разрешена перезапись непустых запросов.

GeneratorName и *GeneratorStep* задают, соответственно, имя и шаг генератора.

KeyFields содержит список ключевых полей.

SeparateBlobUpdate управляет записью блоб-полей в базу данных. Если опция установлена, то будет сначала производится запись строки без блоб-поля, и, в случае успеха, будет записываться и само блоб-поле.

При установленной опции *UpdateOnlyModifiedFields*, если установлена также опция *CanChangeSQLs*, для каждой операции обновления будет автоматически формироваться новый SQL-запрос, в котором будут представлены только те поля, которые реально были изменены.

UpdateTableName должна содержать имя обновляемой таблицы

WhenGetGenId позволяет задать режим использования генератора для формирования первичного ключа – не генерировать ключ, генерировать при добавлении новой записи, генерировать непосредственно перед операцией Post.

ParamsToFieldsLinks - представляет собой «карту соответствия» между полями запроса и параметрами. Если это свойство заполнено, то при вставке новой записи, автоматически указанные поля будут заполнены значениями соответствующих параметров. Особенно актуально это свойство для деталь-датасета в режиме мастер деталь.

AutoParamsToFields - если содержит значение True то ParamsToFieldsLinks будет заполнен автоматически.

Последние два свойства поясним на примере. Допустим есть такой запрос:

```
Select id, id_parent, name from Table1 where id_parent=:mas_id
```

Если свойство AutoParamsToFields равняется True, то после открытия этого запроса, ParamsToFieldsLinks заполнится одной строкой 'id_parent=mas_id'. В дальнейшем, если к этому датасету будет применен метод Insert, то поле id_parent сразу же будет заполнено значением параметра mas_id

FIBPlus, при использовании настроек AutoUpdateOptions позволяет избавиться от генерации модифицирующих SQL в режиме дизайна и переложить эту задачу на время исполнения программы. Для этого Вам, фактически, нужно лишь прописать имя обновляемой таблицы и ключевое поле.

Все это можно как настраивать как в режиме дизайна, так и в режиме выполнения. Код взят из примера AutoUpdateOptions:

```
pFIBDataSet1.SelectSQL.Text := 'SELECT * FROM EMPLOYEE';  
pFIBDataSet1.AutoUpdateOptions.AutoRewriteSqls := True;  
pFIBDataSet1.AutoUpdateOptions.CanChangeSQLs := True;  
pFIBDataSet1.AutoUpdateOptions.UpdateOnlyModifiedFields := True;
```

```

pFIBDataSet1.AutoUpdateOptions.UpdateTableName := 'EMPLOYEE';
pFIBDataSet1.AutoUpdateOptions.KeyFields      := 'EMP_NO';
pFIBDataSet1.AutoUpdateOptions.GeneratorName  := 'EMP_NO_GEN';
pFIBDataSet1.AutoUpdateOptions.WhenGetGenID   := wgBeforePost;
pFIBDataSet1.Open;

```

Master-detail

Реализация мастер-деталь связей в FIBPlus делается точно так же, как в BDE на компонентах TQuery. Т.е. у детального датасета, должно быть заполнено свойство DataSource, причем сам DataSource должен быть ассоциирован с мастер-датасетом. Поясним на примере:

Допустим в базе есть 2 таблицы - клиенты и заказы. Нам нужно создать два набора данных которые должны быть соединены в связь мастер-деталь. Клиенты - master, заказы - detail.

Мастер-датасет назовем MDS, деталь-датасет назовем DDS. И пусть с мастер-датасетом связан DataSource по имени MasterSrc, а с детальным - DataSource по имени DetailSrc.

- 1) Прописываем у MDS в SelectSQL запрос
select id, name from clients
- 2) Прописываем у DDS в SelectSQL запрос
select * from orders where client_id = :mas_id.
- 3) У детального датасета DDS присваиваем в свойство DataSource ссылку на MasterSrc.

Собственно все. Связь мастер-деталь создана. Теперь при изменении текущей записи в датасете MDS, подчиненный датасет DDS будет автоматически переоткрываться, причем значение параметра 'mas_id', он будет брать из поля 'id' главного датасета. Вообще следует заметить, что при переоткрытии Select запроса детального датасета, FIBPlus заполняет его параметры, значениями соответствующих полей главного датасета. При этом руководствуясь принципом совпадения имени параметра детального датасета с именем поля главного. Либо если параметр имеет необязательный префикс «mas_», то этот префикс сначала отсекается, а затем ищется совпадение по усеченному имени. Если же у детального датасета выполняется не Select запрос, а Update, Delete или Refresh, то параметры этих запросов заполняются из полей **детального** датасета, кроме параметров начинающихся с префикса «mas_». Параметры, начинающиеся с этого префикса, всегда заполняются значениями полей главного датасета.

Для более тонкой настройки поведения детального датасета существует свойство TDetailConditions. Рассмотрим возможные значения опций этого свойства по отдельности.

- 1) *dcForceOpen* – если включено, то детальный датасет автоматически откроется при первом открытии главного.
- 2) *dcIgnoreMasterClose* – если включено, то детальный датасет не будет автоматически закрываться, даже если главный датасет уже закрылся
- 3) *dcForceMasterRefresh* – форсирует автоматический рефреш главного датасета, в случае если записи детального подверглись модификации.
- 4) *dcWaitEndMasterScroll* – если включено, то по мере перемещения по главному датасету, детальный датасет будет переоткрываться с задержкой. Если за время этой задержки, опять произошло перемещение по главному датасету, то

детальный отреагирует, только на последнее перемещение. Само время задержки регулируется через свойство датасета `WaitEndMasterInterval:integer` и по умолчанию равно 300 тиков. Этот режим позволяет уменьшить количество запросов к серверу, за счет уменьшения переоткрытий детального датасета.

Локальная сортировка

FIBPlus содержит методы локальной сортировки, которые позволяют пересортировать кэш `TrFIBDataSet` в любом интересующем вас порядке, а также запомнить и восстановить сортировку после переоткрытия `TrFIBDataSet`. Кроме того, FIBPlus поддерживает режим, при котором вновь вставленные и измененные записи будут помещаться в правильную позицию буфера в соответствии с порядком сортировки.

Для того чтобы отсортировать буфер `TrFIBDataSet` вызовите один из трех следующих методов.

```
procedure DoSort(Fields: array of const; Ordering: array of Boolean); virtual;
procedure DoSortEx(Fields: array of integer; Ordering: array of Boolean);
overload;
procedure DoSortEx(Fields: TStrings; Ordering: array of Boolean); overload;
```

Первый параметр - это список полей, а второй - массив направлений сортировки, `True` означает, что сортировка будет по возрастанию, `False` – по убыванию.

Так, например, следующие два примера кода выполнят одну и ту же сортировку по двум полям в порядке возрастания:

```
pFIBDataSet1.DoSort(['FIRST_NAME', 'LAST_NAME'], [True, True]);
или
pFIBDataSet1.DoSortEx([1, 2], [True, True]);
```

Текущую сортировку можно получить, используя свойства

```
function SortFieldsCount: integer;
    возвращает количество полей сортировки

function SortFieldInfo(OrderIndex:integer): TSortFieldInfo -
    возвращает информацию о сортировке в позиции OrderIndex.

function SortedFields:string
    возвращает строку с полями сортировки, перечисленными через ','

TSortFieldInfo = record
    FieldName: string;           //имя поля
    InDataSetIndex: Integer;     //входит ли в индекс
    InOrderIndex: Integer;      //содержится ...
    Asc: Boolean;               //Истина если по возрастанию
    NullsFirst: Boolean;       //если Null значение перед остальными
end;
```

Для того, чтобы записи при вставке и редактировании попадали в правильную позицию буфера, установите свойство `roKeepSorting` в `True`. Если вы хотите, чтобы `TrFIBDataSet` автоматически использовал тот же порядок локальной сортировки, который указан в запросе в выражении `ORDER BY`, используйте опцию `psGetOrderInfo`.

Для того чтобы не терять порядок сортировки при переоткрытии `TrFIBDataSet`, установите свойство `roPersistentSorting`. Но будьте осторожны, при больших выборках это приведет к тому, что в буфер в памяти будут получены все записи с сервера, и только потом

весь буфер будет отсортирован.

Сортировка национальных символов

На правильную сортировку национальных символов, как правило, влияет два параметра - набор символов (CHARSET) и порядок сортировки (COLLATION). И даже при правильно выставленных этих параметрах на уровне БД и запросов, вы можете обнаружить, что TrFIBDataSet сортирует их неправильно. Дело в том, что при локальной сортировке по умолчанию используется простой метод сортировки без сравнения национальных кодировок. Так, как если бы был указан неопределенный набор символов (NONE).

В компоненте TrFIBDataSet существует возможность сортировки в соответствии с национальной кодировкой. Установить альтернативный метод сравнения строковых полей можно при помощи свойства OnCompareFieldValues. Существуют три готовых метода для Ansi- сортировки:

```
function CompareFieldValues(Field:TField;const S1,S2:variant):integer;
```

```
function AnsiCompareString(Field:TField;const val1, val2: variant): Integer;
```

```
function StdAnsiCompareString(Field:TField;const S1, S2: variant): Integer;
```

AnsiCompareString будет учитывать регистр символов, а StdAnsiCompareString будет, наоборот, игнорировать регистр.

```
pFIBDataSet1.OnCompareFieldValues := pFIBDataSet1.AnsiCompareString;
```

```
pFIBDataSet1.OnCompareFieldValues := pFIBDataSet1.StdAnsiCompareString;
```

По умолчанию для CompareFieldValues вы можете установить один из стандартных методов, либо написать свой, если ни один из готовых вам не подходит.

Локальная фильтрация

TrFIBDataSet в отличие от IBX полностью поддерживает локальную фильтрацию. Поддерживается свойство Filter и Filtered, а также событие OnFilterRecord.

В таблице X перечислены дополнительные операторы, которые могут быть использованы в строке Filter.

<i>Операция</i>	<i>Описание</i>
<	Less than.
>	Greater than.
>=	Greater than or equal.
<=	Less than or equal.
=	Equal to
<>	Not equal to
AND	Logical AND
NOT	Logical NOT
OR	Logical OR
IS NULL	Tests that a field value is null
IS NOT NULL	Tests that a field value is not null
+	Adds numbers, concatenates strings, adds number to date/time values

<i>Операция</i>	<i>Описание</i>
–	Subtracts numbers, subtracts dates, or subtracts a number from a date
*	Multiplies two numbers
/	Divides two numbers
Upper	Upper-cases a string
Lower	Lower-cases a string
Substring	Returns the substring starting at a specified position
Trim	Trims spaces or a specified character from front and back of a string
TrimLeft	Trims spaces or a specified character from front of a string
TrimRight	Trims spaces or a specified character from back of a string
Year	Returns the year from a date/time value
Month	Returns the month from a date/time value
Day	Returns the day from a date/time value
Hour	Returns the hour from a time value
Minute	Returns the minute from a time value
Second	Returns the seconds from a time value
GetDate	Returns the current date
Date	Returns the date part of a date/time value
Time	Returns the time part of a date/time value
Like	Provides pattern matching in string comparisons
In	Tests for set inclusion
*	Wildcard for partial comparisons.

При установленном и неактивном свойстве `Filter`, можно воспользоваться следующими методами для перемещения по записям, которые удовлетворяют условию фильтрации:

FindFirst

FindLast

FindNext

FindPrior

На больших объемах данных рекомендуется использовать серверную фильтрацию. Как уже было сказано выше, ее можно осуществить при помощи механизма макросов и условий.

Для ознакомления же с локальной фильтрацией рекомендуется изучить пример `LocalFiltering`.

Важно: для того, чтобы получить количество записей, которые попадают под условие фильтрации, используйте функцию `VisibleRecordCount` вместо `RecordCount`.

Поиск данных

После фильтрации логично перейти к поиску данных. `TpFIBDataSet` поддерживает методы `Locate`, `LocateNext`, `LocatePrior`, описание которых можно получить в стандартной справке Delphi/C++Builder.

В дополнение к этим методам добавлены аналогичные методы, которые учитывают

специфику FIBPlus и имеют некоторые преимущества.

```
function ExtLocate(const KeyFields: String; const KeyValues: Variant; Options: TExtLocateOptions): Boolean;
```

```
function ExtLocateNext(const KeyFields: String; const KeyValues: Variant; Options: TExtLocateOptions): Boolean;
```

```
function ExtLocatePrior(const KeyFields: String; const KeyValues: Variant; Options: TExtLocateOptions): Boolean;
```

TExtLocateOptions = (eloCaseInsensitive, eloPartialKey, eloWildCards, eloInSortedDS, eloNearest, eloInFetchedRecords)

eloCaseInsensitive игнорировать регистр ;

eloPartialKey частичное совпадение

eloWildCards поиск по маске (подобно оператору LIKE);

eloInSortedDS поиск в отсортированном наборе данных (влияет на скорость);

eloNearest только в комбинации с eloInSortedDS. Позиционируется на место где "должно было быть";

eloInFetchedRecords поиск только в уже полученных записях.

Пессимистическая блокировка

Стандартное поведение при обновлении записей серверами семейства InterBase заключается в оптимистической блокировке. Если одна запись одновременно редактируется двумя или более пользователями, то последнее обновление запишется в БД, без учета факта других модификаций с момента начала редактирования записи.

Как правило, если нужна пессимистическая блокировка, разработчики приложений для InterBase/Firebird используют так называемый «холостой update». Это означает, что перед редактированием записи выполняется холостое обновление записи, например, по первичному ключу:

```
update customer set cust_no = cust_no where cust_no = :cust_no
```

Под Firebird 2 и выше, текст лолирующего запроса может быть и другой.

```
Select 0 from customer where cust_no = :cust_no for update with lock.
```

Этот вариант более щадящий, поскольку не производится даже холостого update и как следствие не срабатывают триггера на update. Для того чтоб FIBPlus создавал именно такой вид лолирующего запроса, необходимо в датасете включить опцию poUseSelectForLock в свойство Options. Ну и кроме того, разработчик может вообще взять на себя генерацию лолирующего запроса, создав обработчик OnLockSQLText.

После того, как лолирующий запрос выполнен, с сервера автоматически запрашивается последняя актуальная версия записи. Такое поведение гарантирует, что запись нельзя будет обновить из другой транзакции, пока не завершится транзакция, выполнившая холостой update. Важен еще один момент, блокирование записи нельзя отменить, не завершив модифицирующую транзакцию. На уровне сервера нет способа отменить блокировку одной единственной записи.

FIBPlus автоматизирует это поведение - вам достаточно включить опцию poProtectedEdit, либо вы можете использовать метод TrFIBDataSet.LockRecord.

Демонстрационный пример ProtectedEditing демонстрирует работу этой опции.

Работа в режиме ограниченного кэша

Режим ограниченного кэша впервые был предложен в компонентах `gb_Datasets` Сергея Спирина. Начиная с версии FIBPlus 6.0, аналогичная функциональность была реализована непосредственно в коде FIBPlus. Особенность данного режима состоит в том, что при навигации по `TrFIBDataSet` не происходит полное скачивание всех записей, которые возвращаются запросом. Фактически, реализуется имитация непосредственного доступа к произвольным записям за счет выполнения дополнительных запросов. Технология накладывает ряд требований на запрос. В частности, необходимым требованием является использование `ORDER BY` в `SelectSQL`. Причем, комбинация значений полей, входящих в выражение `ORDER BY`, должна быть уникальной. Во-вторых, для реального быстрого действия желательно иметь два индекса - по возрастанию и по убыванию - для данной комбинации полей.

Технологию можно пояснить следующим простым примером. Пусть задан запрос в `SelectSQL`:

```
SELECT * FROM TABLE  
ORDER BY FIELD
```

Мы можем получить несколько первых записей, делая последовательный `fetch`. Если мы хотим сразу посмотреть последние записи, то вместо полного выкачивания на клиента всех записей данного запроса, можно выполнить дополнительный запрос с обратной сортировкой:

```
SELECT * FROM TABLE  
ORDER BY FIELD DESC
```

Очевидно, что последовательный `fetch` нескольких записей вернет нам последние записи относительно первоначального запроса. Аналогичные запросы выполняются для точного позиционирования на произвольную запись, а также на записи выше и ниже текущей:

```
SELECT * FROM TABLE  
WHERE (FIELD = x)
```

```
SELECT * FROM TABLE  
WHERE (FIELD < x)  
ORDER BY FIELD DESC
```

```
SELECT * FROM TABLE  
WHERE (FIELD > x)  
ORDER BY FIELD
```

Для реализации этой технологии в `TrFIBDataSet` добавлено свойство:

property `CacheModelOptions:TCacheModelOptions`, где

```
TCacheModelOptions = class(TPersistent)  
  property BufferChunks: Integer ;  
  property CacheModelKind: TCacheModelKind ;  
  property PlanForDescSQLs: string ;  
end;
```

`BufferChunks` в данном случае заменяет существующее свойство `BufferChunks` у `TrFIBDataSet`. Тип `TCacheModelKind` может принимать значение `cmkStandard` для использования стандартной работы локального буфера, и `cmkLimitedBufferSize` для использования новой технологии ограниченного локального буфера. Размер буфера - это количество записей, указанных в свойстве `BufferChunks`. Свойство `PlanForDescSQLs` позволяет указать отдельный план для запросов с обратной сортировкой.

Необходимо отметить, что при использовании технологии ограниченного локального буфера существует ряд ограничений на другую функциональность TrFIBDataSet:

- a) нельзя включать режим CachedUpdate;
- b) свойство ResNo будет возвращать неправильные значения;
- c) локальная фильтрация не поддерживается;
- d) работа с BLOB-полями в текущей версии не гарантируется;
- e) в PrepareOptions необходимо включать опцию psGetOrderInfo.

Работа с внутренним кэшем набора данных

TrFIBDataSet предоставляет вам несколько специальных методов для работы с внутренним буфером, в котором хранятся записи. В общем-то, данные методы превращают TrFIBDataSet в аналог TClientDataSet, ориентированный на InterBase. Единственное отличие состоит в том, что для работы с TrFIBDataSet должно существовать соединение с БД и в SelectSQL должен быть прописан правильный запрос. Несмотря на это ограничение, механизм является достаточно гибким для реализации множества «нестандартных» вещей. Например, следующий запрос создаст выборку из одного целочисленного поля и одного строкового:

```
select cast(0 as integer) some_id, cast(" as varchar(255)) some_name
from RDB$DATABASE.
```

Открыть такой TrFIBDataSet можно при помощи метода CacheOpen. После этого становятся работоспособны методы:

```
procedure CacheModify(aFields: array of integer; Values: array of Variant;
KindModify: byte );
procedure CacheEdit(aFields: array of integer; Values: array of Variant);
procedure CacheAppend(aFields: array of integer; Values: array of Variant);
overload;
procedure CacheAppend(Value: Variant; DoRefresh: boolean = False); overload;
procedure CacheInsert(aFields: array of integer; Values: array of Variant);
overload;
procedure CacheInsert(Value: Variant; DoRefresh: boolean = False); overload;
procedure CacheRefresh(FromDataSet: TDataSet; Kind: TCachRefreshKind ; FieldMap:
Tstrings);
procedure CacheRefreshByArrMap( FromDataSet: TDataSet; Kind: TCachRefreshKind;
const SourceFields, DestFields: array of string )
```

Так, чтобы добавить запись мы должны выполнить

```
pFIBDataSet1.CacheInsert([0,1],[255, 'string1'])
```

чтобы модифицировать

```
pFIBDataSet1.CacheModify([0,1],[255, 'string1'])
```

чтобы удалить из кеша, просто вызовите CacheDelete;

Методы CacheRefresh и CacheRefreshByArrMap позволяют обновить запись на основе данных из другого TrFIBDataSet.

Помните, что все эти операции не изменяют БД – все действия производятся в буфере TrFIBDataSet.

Такую технику иногда можно использовать и в обычном режиме. Например, иногда

возникает необходимость вставить запись при помощи какой-либо сложной хранимой процедуры, которая возвратит кодвставленной записи, и отобразить его в TrFIBDataSet. Для этого можно, в частности, вставить код и потом вызвать метод Refresh:

```
id := SomeInsertByProc;  
pFIBDataSet1.CacheInsert([0], [1]);  
pFIBDataSet1.Refresh;
```

Вы также можете, наоборот, удалить некоторые несуществующие записи из кеша.

Использование кэшированных изменений.

Кэширование изменений заключается в том, что все корректировки данных не применяются сразу же к базе данных, а сохраняются в локальной копии данных в буфере (кэше). Внесенные изменения могут быть впоследствии физически перенесены в БД, или может быть произведен отказ от запоминания ("откат"). Преимуществом такого способа работы является то, что все изменения передаются на сервер в одной короткой транзакции.

Для того чтоб датасет работал в режиме кэшированных изменений необходимо установить свойство `CachedUpdates` в `true`. Установка этого свойства может происходить только при закрытом наборе данных. Для работы в режиме кэшированных изменений существует несколько специальных свойств, методов и обработчиков.

Свойство:

```
property UpdateRecordTypes: TFIBUpdateRecordTypes;
```

Набор опций которые указывают на то, какие из записей должны быть "видны" в датасете.

```
cusUnmodified - не изменявшиеся;  
cusModified   - измененные;  
cusDeleted    - удаленные;  
cusInserted   - вставленные;
```

```
property UpdatesPending:boolean;
```

Показывает есть ли в датасете непримененные изменения. Если свойство возвращает `true` – значит в наборе данных присутствуют вставленные, измененные или удаленные записи, изменения по которым еще не применялись к базе данных.

```
procedure CancelUpdates;
```

Отменяет изменения в кэше датасета. Отменяются только непримененные к этому времени изменения.

```
procedure ApplyUpdates;
```

Применяет изменения из датасета к базе данных. При этом со всех записей, которые были успешно применены, снимается пометка об их модифицированности. Поэтому метод `CancelUpdates` изменения по этим записям уже отменить не сможет.

procedure ApplyUpdToBase;

Применяет изменения из датасета к базе данных, но отметка о модифицированности с записей не снимается. Поэтому метод `CancelUpdates` может отменить эти изменения.

procedure CommitUpdToCach;

Снимает отметки об изменениях со всех записей. Применяется обычно после успешного применения метода `ApplyUpdToBase`.

Принцип работы обоих методов `ApplyUpdates` и `ApplyUpdToBase` заключается в том, что совершается проход по буферу датасета и для каждой модифицированной записи применяется `InsertSQL`, `UpdateSQL` или `DeleteSQL` в зависимости от того как именно была модифицирована запись. Кроме того, для каждой записи прежде чем будет вызван соответствующий SQL запрос, сначала вызывается обработчик `OnUpdateRecord`.

property OnUpdateRecord: TFIBUpdateRecordEvent; где

```
TFIBUpdateRecordEvent=procedure (DataSet: TDataSet; UpdateKind: TUpdateKind;
    var UpdateAction: TFIBUpdateAction) of object;
```

В этом обработчике вы можете проанализировав запись, изменения по которой сейчас отправляются в базу данных, совершить некие специальные операции и принять решение что с ней делать дальше.

Если в ходе работы методов `ApplyXXX`, произошла ошибка, то для конкретной записи вызовется обработчик

property OnUpdateRecord: TFIBUpdateRecordEvent; где

```
TFIBUpdateErrorEvent=procedure (DataSet: TDataSet; E: EFIBError;
    UpdateKind: TUpdateKind; var UpdateAction: TFIBUpdateAction) of object;
```

В этом обработчике, вы можете проанализировать возникшую ошибку и состояние текущей записи совершить специальные операции по обработке ошибки и принять решение что с записью делать дальше.

В обоих вышеописанных обработчиках в качестве результирующего параметра применяется параметр типа `TFIBUpdateAction`. Вот его описание:

```
TFIBUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApply, uaApplied);
```

```
uaFail      - прекратить исполнение метода ApplyXXX с сообщением об ошибке
uaAbort     - прекратить исполнение метода ApplyXXX без сообщения об ошибке
uaSkip      - пропустить применение этой записи
uaApply     - применить изменения этой записи
uaApplied   - пометить эту запись как уже примененную и никаких действий не делать
uaRetry     - повторить применение этой записи, предполагается что на момент
повторения попытки, значения полей скорректированы программным способом.
```

Кроме того следует упомянуть, что в обоих обработчиках вы можете узнать не только текущее состояние записи, но и состояние бывшее до изменений. Для этого надо воспользоваться свойством объекта `TField OldValue`.

Отдельным вопросом мы рассмотрим работу методов ApplyXXX в сочетании с автогенерируемыми запросами. Если у датасета свойство AutoUpdateOptions.UseExecuteBlock установлено в true, то для всего пакета изменений будет сформирован запрос типа EXECUTE BLOCK. Т.о. сетевой трафик и скорость применения изменений будет существенно оптимизированы. Работает эта опция только для версий сервера начиная с Firebird 2, кроме того, она не может быть применена к датасетам содержащим блоб поля.

В заключение темы о кэшированных изменениях приведем пример кода применяющего изменения к базе данных.

```

if pFIBDataSet1.UpdatesPending then
begin
  pFIBDataSet1.UpdateTransaction.StartTransaction;
  try
    pFIBDataSet1.ApplyUpdToBase;
    pFIBDataSet1.UpdateTransaction.Commit;
    pFIBDataSet1.CommitUpdToCach;
  except
    pFIBDataSet1.UpdateTransaction.Rollback;
  end;
end;

```

Использование уникальных типов полей FIBPlus

FIBPlus имеет несколько уникальных полей. Это TFIBLargeIntField, TFIBWideStringField, TFIBBooleanField, TFIBGuidField.

TFIBLargeIntField

представляет собой поле типа BIGINT в Interbase/ Firebird.

TFIBBCDField

Предназначено для полей типа NUMERIC(x,y) диалекта 3. Т.е. для полей, которые хранятся в 64 битном целом. По умолчанию в TFIBBCDField маппируются только поля, у которых scale меньше 5. Если включить опцию psSQLINT64ToBCD в PrepareOptions, то и поля с любой scale будут представлены именно BCD полем. Следует заметить, что BCD поле единственное решение, для тех кто хочет не терять точность при отображении и манипулировании данными из 64битных полей.

TFIBWideStringField

Предназначено для строковых полей в кодировке UNICODE_FSS и UTF8.

TFIBBooleanField

Эмулирует логическое поле. Стандартного логического поля в Interbase/Firebird нет, но с FIBPlus вы можете его довольно просто эмулировать. Для этого вы должны создать домен (INTEGER или SMALLINT), в имени которого будет содержаться подстрока BOOLEAN. В PrepareOptions TrFIBDataSet должно быть включено свойство psUseBooleanField. Тогда при создании объектов полей FIBPlus будет проверять имя домена, и если там содержится BOOLEAN, то для таких полей будут создаваться экземпляры TFIBBooleanField.

```

CREATE DOMAIN FIB$BOOLEAN AS SMALLINT
DEFAULT 1 NOT NULL CHECK (VALUE IN (0,1));

```

TFIBGuidField

Работает аналогично схеме работы TFIBBooleanField – т.е. поле должно быть объявлено

через домен в имени которого должно присутствовать GUID. Должна быть включена опция psUseGuidIdField. Пример объявления домена приведен ниже.

```
CREATE DOMAIN FIB$GUID AS CHAR(16) CHARACTER SET OCTETS;
```

Если AutoGenerateValue у поля выставлено в True, то при вставке значение поля будет сформировано автоматически.

Работа с полями-массивами

InterBase с самых ранних версий позволял описывать в таблицах многомерные поля массивы, делая хранение специализированных данных максимально удобным. Array-поля - это расширение InterBase, которое стандартом SQL не поддерживается, и работа с такими полями на уровне SQL-запросов крайне затруднена. Фактически, вы можете использовать массивы только поэлементно и только в операциях чтения. Чтобы изменить значения array-поля, необходимо использовать специальные команды InterBase API. FIBPlus позволяет обойтись без подобных сложностей, взяв на себя всю рутину, связанную с array-полями.

Пример работы с array-полями можно найти в демонстрационной базе EMPLOYEE из поставки сервера. Поле LANGUAGE_REQ таблицы JOB является массивом (LANGUAGE_REQ VARCHAR(15) [1:5])

Первый способ дает возможность редактировать array-поле при помощи специальных методов TrFIBDataSet.ArrayFieldValue и SetArrayValue, а также методов GetArrayValues, SetArrayValue и AsQuad класса TFIBXSQLDA. Методы позволяют работать с таким полем, как с единой структурой. TFIBXSQLDA.GetArrayElement позволяет получить значение элемента массива по индексу.

Методы TrFIBDataSet.ArrayFieldValue и TFIBXSQLDA.GetArrayValues позволяют получить вариантный массив из значений поля-массива. Например, чтобы получить отдельные элементы, можно использовать следующий код:

```
var v: Variant;
with ArrayDataSet do begin
    v := ArrayFieldValue(FieldByName('LANGUAGE_REQ'));
    Edit1.Text := VarToStr(v[1]);
end;
```

Метод TrFIBDataSet.SetArrayValue и TFIBXSQLDA.SetArrayValue позволяют задать все элементы поля в виде массива вариантов:

```
with ArrayDataSet do
    SetArrayValue(FBN('LANGUAGE_REQ'), VarArrayOf([Edit1.Text, ...]));
```

В новых версиях FIBPlus это можно сделать еще проще, присваивая вариантный массив напрямую:

```
FBN('LANGUAGE_REQ').Value := VarArrayOf([Edit1.Text, ...]);
```

Наиболее удачным местом для заполнения поля является событие BeforePost. В случае неудачной операции Update или Insert необходимо восстанавливать внутренний идентификатор массива у редактируемой записи. Для этого достаточно обновить текущую

запись, вызвав метод Refresh. Это правило диктуется функциями InterBase API, и мы вынуждены его придерживаться. Таким образом, обработчик ошибок является важным моментом при работе с полями-массивами. Его можно поместить в OnPostError.

```
procedure ArrayDataSetPostError(DataSet: TDataSet; E: EDatabaseError; var
Action: TDataAction);
begin
  Action := daAbort;
  MessageDlg('Error!', mtError, [mbOk], 0);
  ArrayDataSet.Refresh;
end;
```

Функция TFIBXSQLEDA.AsQuad для полей массивов и блоб-полей содержит BLOB_ID данного поля.

Работу с полями-массивами демонстрирует примеры ArrayFields1 и ArrayFields2. В первом из них для отображения поля используется техника извлечения массива из поля, во втором - прямая выборка в SelectSQL. И в том, и в другом случае используется одинаковая техника записи поля в БД свернутым полем-массивом.

Релизы Firebird 1.5.X содержат ошибку, из-за которой для строковых полей возвращается неправильная длина. Поэтому вы можете увидеть странный последний символ «|» в строковых элементах массива.

Использование контейнеров TDataSetContainer

Компонент TDataSetContainer позволяет централизованно обрабатывать события от разных компонентов TrFIBDataSet, а также посылать им сообщения, при получении которых они также могут производить какие-то дополнительные действия. Например, вы могли бы перед открытием всех TrFIBDataSet стандартным образом настраивать параметры отображение полей, сохранять и восстанавливать сортировку и много многое другое. Казалось бы, что подобного результат можно достичь, попросту присвоив один и тот же обработчик нескольким экземплярам TrFIBDataSet. Но удобство использования TDataSetContainer состоит, в частности, в том, что его, а, следовательно, и обработчики, можно размещать на отдельном TDataModule, чтобы вынести общий код приложения за пределы визуальных форм.

Кроме того, он может быть использован для задания общей функции локальной сортировки для присоединенных к нему TrFIBDataSet.

```
TKindDataSetEvent = (deBeforeOpen, deAfterOpen, deBeforeClose,
deAfterClose, deBeforeInsert, deAfterInsert, deBeforeEdit, deAfterEdit,
deBeforePost, deAfterPost, deBeforeCancel, deAfterCancel, deBeforeDelete,
deAfterDelete, deBeforeScroll, deAfterScroll, deOnNewRecord, deOnCalcFields,
deBeforeRefresh, deAfterRefresh)
```

Начиная с версии FIBPlus 6.4.2, контейнер может быть глобальным для всех экземпляров TrFIBDataSet. Для этого достаточно установить свойство IsGlobal в Истину.

Существует возможность строить целые цепочки контейнеров. Для этого используется свойство MasterContainer. Таким образом, можно дополнять поведение контейнеров путем «наследования» поведения MasterContainer.

Дополнительные действия при модификации данных TrFIBUpdateObject

В дополнение к стандартным модифицирующим запросам `TrFIBDataSet` можно задавать сколь угодно много дополнительных действий. Это осуществляется при помощи объектов `TrFIBUpdateObject`. Это объект наследник `TrFIBQuery`, своеобразный клиентский триггер, который позволяет выполнить дополнительные действия для `TrFIBDataSet` до или после вставки, модификации и удаления. Наследуемые свойства и методы `TrFIBQuery` читайте в соответствующем разделе Приложения.

Например, у вас в программе есть связка мастер-деталь – таблицы `master(id, name)` и `detail(id, name, master_id)` и при удалении записей из таблицы `master` вы хотели бы, чтобы автоматически удалялись зависимые данные в таблице `detail`. Пример немного надуманный, но использование компонента демонстрирует. Итак, чтобы реализовать необходимое поведение добавим `rFIBUpdateObject`. Заполним его свойство `SQL` `'delete from deatail where master_id = :id'`, установим свойство `DataSet` в датасет для таблицы `master`, установим `KindUpdate` в `kuDelete` и, наконец, скажем, что выполнять его нужно перед оператором основного `DataSet` – `ExecuteOrder` `oeBeforeDefault`. Вот и все – теперь перед удалением записи из `MasterDataSet` будет производиться оператор из ассоциированного `rFIBUpdateObject`.

Работа в режиме разделенных транзакций

Как было сказано в разделе [«Работа с транзакциями»](#) нужно стремиться делать обновляющие транзакции как можно короче. `TrFIBDataSet` имеет уникальную возможность работать сразу в контексте двух транзакций: `Transaction` и `UpdateTransaction`. Мы рекомендуем этот способ, как наиболее правильный для работы с `InterBase/Firebird`. Запускается одна длинная транзакция, в которой данные только читаются, и другая короткая транзакция, в контексте которой выполняются все модифицирующие запросы.

При этом читающая транзакция (`Transaction`), как правило, `ReadCommitted` и только для чтения (чтобы не удерживать версии записей). Рекомендуемые параметры для нее: `read`, `nawait`, `rec_version`, `read_committed`. Пишущая транзакция (`UpdateTransaction`) короткая и конкурентная, рекомендуемые параметры `write`, `nawait`, `concurrency`.

В этом случае, при использовании свойства `AutoCommit = True`, каждое изменение в `TrFIBDataSet` будет записано в БД немедленно и станет доступным для других пользователей. Вообще для режима разделенных транзакций, крайне рекомендуется включать свойство `AutoCommit`. Иначе во многом теряется смысл разделения транзакций. Кроме того если не использовать `AutoCommit`, то возникает вопрос, в какой из транзакций можно корректно производить рефреш записи. Дело в том, что в датасете, после модификации записи возникает ситуация, когда часть записей соответствует контексту читающей транзакции, а только что модифицированная соответствует контексту пишущей, которая еще не завершена. Конечно, можно делать рефреш записи в контексте пишущей транзакции чтоб видеть изменения, но... полное переоткрытие датасета все эти изменения скроет, поскольку они не видны в контексте читающей транзакции. Для управления этой ситуацией существует свойство `RefreshTransactionKind: TTransactionKind`, которое и определяет в контексте какой транзакции будет осуществлен `Refresh`. Но нужно понимать, что это полумера и единственной гарантией правильного отображения данных может служить только быстрое завершение пишущей транзакции.

Вы должны хорошо представлять себе, в какой момент запускаются и закрываются транзакции в вашем приложении во избежание «непонятных» ситуаций.

Пакетная обработка

У компонента TrFIBDataSet есть несколько методов для выполнения пакетных операций. Это методы BatchRecordToQuery и BatchAllRecordToQuery, которые выполняют SQL-запрос, предварительно настроенный в TrFIBQuery, который передается как параметр.

Использование этих методов показано в демонстрационном примере DatasetBatching.

Перечень свойств TrFIBDataSet

property DefaultFormats: TFormatFields;

DateTimeDisplayFormat: формат для DateTime полей
NumericDisplayFormat : формат для показа числовых полей
NumericEditFormat : формат для редактирования числовых полей
DisplayFormatDate : формат для показа Date полей;
DisplayFormatTime : формат для показа Time полей;

Работает это свойство следующим образом. После открытия датасета, после того как созданы все поля, каждому из полей в свойства DisplayFormat и EditFormat присваивается соответствующий формат из DefaultFormats. В принципе, в обработчике AfterOpen вы можете для отдельных полей переприсвоить формат на любой другой. Для Numeric полей обработка несколько сложнее. Для каждого из них FIBPlus выясняет scale поля, и преобразует формат таким образом, чтоб число знаков после запятой в формате соответствовало scale. Приведем несколько примеров.

Допустим у нас есть поле NUMERIC(18,2), посмотрим как оно будет отформатировано при различных значениях NumericDisplayFormat.

- 1) DefaultFormats.NumericDisplayFormat:='#,###.';
В этом случае в Field.DisplayFormat попадет строка '#,###.00'.
- 2) DefaultFormats.NumericDisplayFormat:='#,###.0';
В этом случае в Field.DisplayFormat тоже попадет строка '#,###.00'.
- 3) DefaultFormats.NumericDisplayFormat:='#,###.#';
В этом случае в Field.DisplayFormat попадет строка '#,###.###'.
- 4) DefaultFormats.NumericDisplayFormat:='#,###';
В этом случае в Field.DisplayFormat попадет строка '#,###'.

Замечание: Действие свойства можно отключить выключив опцию poAutoFormatFields в свойстве Options.

property DetailConditions:TDetailConditions;

Подробно рассмотрено в теме про Master-Detail

property FieldOriginRule:TFieldOriginRule;

Возможные значения свойства:

forNoRule, forTableAndFieldName, forClientFieldName, forTableAliasAndFieldName

Свойство влияет на результат возвращаемый методом `GetFieldOrigin`, и соответственно на значение свойств `Field.Origin`. Рассмотрим на примере. Допустим у нас есть запрос

```
Select T.ID as F from table1 T
```

Посмотрим что будет происходить со свойством `Origin` поля 'F' в зависимости от значения свойства `FieldOriginRule`

- 1) `forNoRule` - `Origin` останется пустым.
- 2) `forTableAndFieldName` - `Origin` будет иметь значение `'TABLE1.ID'`
- 3) `forClientFieldName`-`Origin` будет совпадать с клиентским именем поля. Т.е. «F».
- 4) `forTableAliasAndFieldName`- `Origin` будет иметь значение `'T.ID'`

Замечание: Не для любого поля любого запроса, можно получить `Origin`. Действительно если посмотреть, например, на запрос `Select 1 from RDB$DATABASE`, то становится ясно, что для единственного поля этого запроса, никакого `Origin` существовать не может, потому что нет физического поля в базе данных

property `RefreshTransactionKind: TTransactionKind;`

Рассмотрено в теме о режиме разделенных транзакций.

property `UniDirectional: Boolean;`

При `False` все выбираемые записи кэшируются в буфере `FIBDataSet`, что позволяет после выборки всех записей перемещаться от начала до конца выборки и обратно, не обращаясь к серверу. При `True` размер буфера ограничен, просмотр записей "вверх" на определенном этапе будет невозможен. Значение `True` имеет смысл выставлять, если датасет участвует в каком-нибудь однократном процессе, например формировании отчета на основе данных выборки.

property `Options: TpFIBDsOptions;`

Набор опций регулирующий разные аспекты поведения датасета. Рассмотрим их по отдельности.

`poTrimCharFields`- если включено то в полях типа `char` будут обрезаться хвостовые пробелы.

`poRefreshAfterPost` - Определяет нужно ли делать автоматический `Refresh` после `Post`

`poRefreshAfterDelete`- Определяет нужно ли делать автоматический `Refresh` после `Delete`

`poRefreshDeletedRecord`- Определяет что делать с записью, если `Refresh` записи обнаружил ее отсутствие в базе данных. Т.е. запись вероятней всего была удалена, либо перестала попадать в условие выборки. Если этот флаг включен, то в этом случае запись будет вычеркнута из буфера датасета.

`poStartTransaction` - Определяет нужно ли автоматически стартовать транзакцию при попытке открытия датасета.

poAutoFormatFields - Включает действие свойства *DefaultFormats*.

poProtectedEdit - Включает режим пессимистической блокировки.

poUseSelectForLock - Описано в теме о пессимистической блокировке.

poKeepSorting - Описано в теме о локальной сортировке.

poPersistentSorting - Описано в теме о локальной сортировке.

poVisibleRecno - Определяет значение возврата метода *Recno* в случае если он применяется к отфильтрованному датасету. Если установлен флаг *poVisibleRecno*, то метод *Recno* будет возвращать не номер записи в полном буфере, а номер записи в уже отфильтрованной его части.

poNoForceIsNull - Действие аналогично опции *qoNoForceIsNull* в *TrFIBQuery*.

poFetchAll - Определяет фетчить ли все записи сразу же после открытия датасета.

poFreeHandlesAfterClose - Действие аналогично опции *qoFreeHandleAfterExecute* в *TrFIBQuery*.

poCacheCalcFields - Оптимизирует внутреннюю работу с *Calc* полями.

property *PrepareOptions*: *TrPrepareOptions*;

Набор опций регулирующий разные аспекты поведения датасета. Рассмотрим их по отдельности.

pfSetRequiredFields - если включено то в *Fields* датасета будет автоматически заполняться свойство *Required*. Для NOT NULL полей оно будет принимать значение *True*, для остальных *False*. (Эта опция не вызывает дополнительные запросы к базе)

pfSetReadOnlyFields - Если включено, то те поля датасета, которые не участвуют в модифицирующих запросах, автоматически становятся *ReadOnly*. Т.е. их нельзя будет изменять даже в буфере датасета. Кроме того становятся *ReadOnly* те поля, которые являются калькулируемыми на сервере. (Эта опция вызывает дополнительные запросы к базе, с целью выяснить какие поля являются *server-calculated*)

pfImportDefaultValues - Если включено, то те поля датасета, которые в базе имеют значение по умолчанию, автоматически получают соответствующее значение в свойство *DefaultExpression*. Это свойство используется при *Insert/Append* новой записи. (Эта опция вызывает дополнительные запросы к базе, с целью получения текста значения по умолчанию)

pfImportDefaultValues - Если включено, то те поля датасета, которые в базе имеют значение по умолчанию, автоматически получают соответствующее значение в свойство *DefaultExpression*. Это свойство используется при *Insert/Append* новой записи. (Эта опция вызывает дополнительные запросы к базе, с целью получения текста значения по умолчанию)

psUseBooleanField - Если включено, то датасет сможет использовать *Boolean* поля. Подробнее см. в теме «Использование уникальных типов полей». (Эта опция вызывает дополнительные запросы к базе, с целью получения домена поля)

psUseGuidField - Если включено, то датасет сможет использовать *GUID* поля. Подробнее см. в теме «Использование уникальных типов полей» (Эта опция вызывает дополнительные запросы к базе, с целью получения домена поля)

psSQLINT64ToBCD - Если включено, то датасет будет использовать *TBCDField* для полей типа *NUMERIC(x,y)*, где *y* больше 4. Подробнее см. в теме «Использование уникальных типов полей» (Эта опция не вызывает дополнительные запросы к базе)

psApplyRepository - Если включено, то датасет будет использовать репозиторий полей, для заполнения таких свойств полей, как *DisplayLabel*, *DisplayFormat*, *EditFormat* и т.д. Подробнее см в теме «Репозитории FIBPlus»

psGetOrderInfo - Если включено, то при открытии датасета автоматически заполняется свойство датасета *SortFields*, которое в дальнейшем используется в режимах ограниченного кэша и режимах «удерживания» сортировки при операциях вставки модификации записей. Подробнее см. в теме «Локальная сортировка». (Эта опция не вызывает дополнительные запросы к базе)

psAskRecordCount - Если включено, то перед открытием датасета будет послан запрос к серверу, с целью выяснить количество записей, которые попадают под условия запроса. После этого свойство *RecordCount* будет возвращать не количество отфетченных записей, а количество попавших под условия запроса.

psSetEmptyStrToNull - Если включено, то если строковое поле содержит пустую строку, и эта запись модифицирована, то при применении изменений к базе, для этого поля будет отправлено Null значение. Необходимость этой опции связана с тем, что стандартные *DBControls* не могут показать, или принять Null значение. Оно в них отображается именно как пустая строка.

psUseLargeIntField - Если включено, то поля типа *BIGINT*, *NUMERIC(18,0)* будут представлены классом *TFIBLargeField*. Если выключено, то они будут представлены классом *TFIBBCDField*.

Работа с блоб-полями в клиентских приложениях InterBase и Firebird на основе компонентов FIBPlus

Введение

Достаточно часто желательно хранить в базе данных разнообразные неструктурированные данные: изображения, OLE-объекты, звук и т.д. Специально для этих целей существует специальный тип данных - BLOB. Прежде чем рассматривать работу FIBPlus с полями этого типа на примерах, вспомним о том, как сам сервер реализовывает работу с ними. Важно знать и помнить следующее: В отличие от всех других полей, данные BLOB поля не хранятся непосредственно в записи таблицы. В записи таблицы хранится лишь идентификатор BLOB (*BLOB_ID*), а само тело BLOB хранится на отдельных страницах базы данных и доступ к ним осуществляется специальными функциями IB API. Эта особенность позволяет хранить в BLOB полях данные нефиксированного размера. FIBPlus максимально скрывает эти нюансы непосредственно от разработчика, от вас не требуется самостоятельно вызывать вышеупомянутые функции IB API, но, тем не менее, полезно знать, что происходит «за кулисами» .

Итак, продемонстрируем использование BLOB-полей на примере следующей таблицы:

```
CREATE TABLE BIOLIFE (  
  ID INTEGER NOT NULL,  
  CATEGORY VARCHAR (15) character set WIN1251 collate WIN1251,  
  COMMON_NAME VARCHAR (30) character set WIN1251 collate WIN1251,  
  SPECIES_NAME VARCHAR (40) character set WIN1251 collate WIN1251,  
  LENGTH_CM DOUBLE PRECISION,  
  LENGTH_IN DOUBLE PRECISION,  
  NOTES BLOB sub_type 1 segment size 80,  
  GRAPHIC BLOB sub_type 0 segment size 80);
```

Использование *TrFIBDataSet* при работе с BLOB-полями

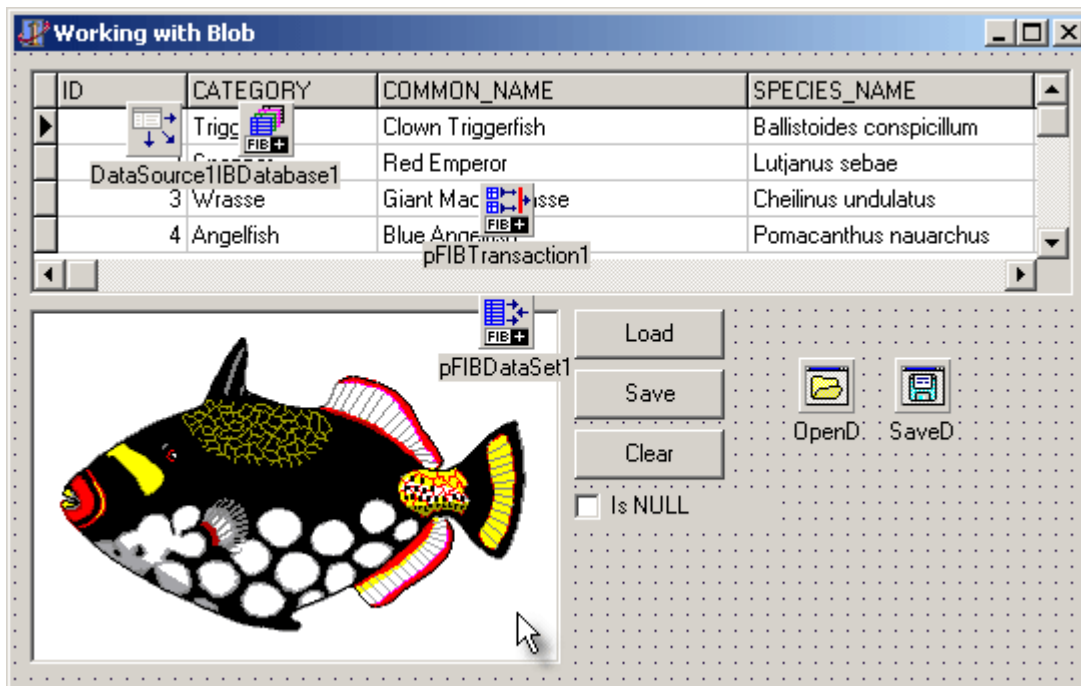


Рис. 1. Внешний вид формы приложения для работы с BLOB-полями.

Для вывода изображений, сохраненных в поле GRAPHIC, мы будем использовать стандартный компонент DBImage1: TDBImage. Очевидно также, что запросы при работе с BLOB-полями внешне ничем не отличаются от запросов со стандартными типами полей:

SelectSQL:

```
SELECT * FROM BIOLIFE
```

UpdateSQL:

```
UPDATE BIOLIFE Set
  ID=?NEW_ID,
  CATEGORY=?NEW_CATEGORY,
  COMMON_NAME=?NEW_COMMON_NAME,
  SPECIES_NAME=?NEW_SPECIES_NAME,
  LENGTH_CM=?NEW_LENGTH_CM,
  LENGTH_IN=?NEW_LENGTH_IN,
  NOTES=?NEW_NOTES,
  GRAPHIC=?NEW_GRAPHIC
WHERE ID=?OLD_ID
```

InsertSQL:

```
INSERT INTO BIOLIFE (
  ID,
  CATEGORY,
  COMMON_NAME,
  SPECIES_NAME,
  LENGTH_CM,
  LENGTH_IN,
  NOTES,
  GRAPHIC
)
VALUES (
  ?NEW_ID,
  ?NEW_CATEGORY,
  ?NEW_COMMON_NAME,
  ?NEW_SPECIES_NAME,
  ?NEW_LENGTH_CM,
  ?NEW_LENGTH_IN,
```

```

        ?NEW_NOTES,
        ?NEW_GRAPHIC
    )

```

```

DeleteSQL:
DELETE FROM BIOLIFE
WHERE ID=?OLD_ID

```

```

RefreshSQL:
SELECT * FROM BIOLIFE
WHERE
ID=?OLD_ID

```

Нюансы при чтении:

Сразу же акцентируем внимание на первом скрытом нюансе. Выполнение «SELECT * FROM BIOLIFE» не вычитывает данные из BLOB полей на клиента. В поле «GRAPHIC» возвращается идентификатор BLOB'a. Далее «за кулисами» происходит следующее: Компонент DBImage1 хочет отобразить содержимое поля из первой записи. Он обращается к pFIBDataSet1 за этим содержимым, и тот «втайне» от нас обращается к серверу через специальные функции IB API непосредственно за телом BLOB поля, пользуясь Blob_ID поля из ПЕРВОЙ записи. Таким образом, мы должны понимать, что в отображенной на иллюстрации ситуации мы «вытащили» на клиента содержимое BLOB поля только первой записи. По мере скроллинга по записям в TrFIBDataSet, DBImage1 будет обращаться за данными других записей, и эти обращения будут транслироваться к серверу. За телом блоба FIBPlus обращается всего один раз. После того как содержимое блоба один раз прочитано, оно сохраняется в специальном кэше, для повторного использования. Т.е. если мы в нашем примере, перешли на следующую запись, а потом вернулись к первой, то содержимое поля «GRAPHIC» затребовано с сервера уже не будет. Кэш блобов будет очищен после закрытия pFIBDataSet1. Понятно, что если таблица большая, и мы ее скроллируем всю, то кэш блобов будет занимать достаточно много памяти. Чтоб поруководить этим процессом, разработчику дается возможность ограничить количество кэшируемых блоб-значений. Опция BlobCacheLimit в свойстве CacheModelOptions определяет это количество. Если это значение этой опции отлично от нуля, то в кэш блоб полей датасета будет ограничен именно этим значением.

Нюансы при модификации:

BLOB- поля в TFIBDataSet представлены потомками от TBlobField, и как следствие, наследуют четыре специальных метода модификации таких полей: методы LoadFromFile, LoadFromStream, SaveToFile и SaveToStream.

Первый метод (LoadFromFile) используется для сохранения в поле данных из внешнего файла, второй (LoadFromStream) - для сохранения из любого объекта типа TStream.

Например, если мы хотим сохранить в нашем поле изображение из внешнего файла, то мы можем написать следующий обработчик нажатия на кнопку:

```

procedure TMainForm.OpenBClick(Sender: TObject);
begin
    if not OpenD.Execute then exit;
    with pFIBDataSet1 do
        begin
            Edit;
            TBlobField(FieldByName('GRAPHIC')).LoadFromFile(OpenD.FileName);
            Post;
        end

```

```

    end
end;
```

Обратите внимание на важный момент: перед тем, как присваивать значение BLOB-полю, необходимо перевести `pFIBDataSet` в состояние редактирования данных. В данном случае это делается безусловным вызовом метода `Edit`. После загрузки данных остается только сохранить изменения вызовом метода `Post`.

Второй важный момент - необходимо прописать явное приведение поля к типу `TBlobField`. Дело в том, что `FieldByName` возвращает объект типа `TField`, а он не имеет нужных нам методов.

Помимо специальных методов `LoadFromXXX`, для модификации BLOB полей можно использовать и простые методы типа `FieldByName(...).AsString:='asfdsafsadsad'`;

Методами `SaveToFile`, `SaveToStream` мы можем сохранить значение BLOB-поля в некоторый внешний файл или `TStream`.

Пример сохранения в файл:

```

procedure TMainForm.SaveBClick(Sender: TObject);
begin
    if not Saved.Execute then exit;
    with pFIBDataSet1 do
        begin
            if not FieldByName('GRAPHIC').IsNull then
                begin
                    TBlobField(FieldByName('GRAPHIC')).SaveToFile(Saved.FileName);
                end;
        end;
end;
```

Пример очистки поля.

```

procedure TMainForm.Button1Click(Sender: TObject);
begin
    with pFIBDataSet1 do
        begin
            Edit;
            FieldByName('GRAPHIC').Clear;
            Post;
        end
end;
```

Иногда также нужно знать, является ли BLOB-поле пустым или нет. При использовании визуальных компонентов типа `TDBImage` мы не можем быть в этом уверены. Согласитесь, что никто не мешает нам «нарисовать» пустую картинку и сохранить ее в BLOB-поле. В этом случае, мы не сможем отличить «на глаз»: есть ли какое-то изображение в BLOB-поле, или нет. Однако мы можем написать обработчик события `OnDataChange` компонента `DataSource1: TDataSource`:

```

procedure TMainForm.DataSource1DataChange(Sender: TObject; Field:
    TField);
begin
    CheckBox1.Checked := pFIBDataSet1.FieldByName('GRAPHIC').IsNull;
end;
```

Это событие вызывается, в частности, при навигации по `DBGrid1`, таким образом, мы всегда можем узнать, является ли текущее поле пустым или нет.

Итак, с внешней стороны дела мы разобрались, давайте теперь заглянем «за кулисы». Что происходит при модификациях записи, содержащей BLOB поле?

Вариант 1. Если BLOB поле не изменялось в процессе редактирования, то в соответствующий параметр UPDATE SQL попадет прежний BLOB_ID. Само содержимое BLOB поля на сервер не передается.

Вариант 2. BLOB поле подверглось модификации. Фактически для записи нового содержимого производится несколько действий. Во-первых, с помощью IB API функций `isc_create_blob2`, `isc_put_segment`, `isc_close_blob` в базе данных сохраняется тело НОВОГО BLOB. Так же при этом действии клиент узнает и запоминает BLOB_ID для нового поля. Во-вторых, в UPDATE SQL передается этот новый BLOB_ID, и UPDATE запускается на выполнение. В-третьих (**ОЧЕНЬ НЕОЧЕВИДНЫЙ НЮАНС**), сервер при фиксации изменений записи ПРЕОБРАЗУЕТ полученный BLOB_ID. То есть, тот BLOB_ID, который передавался клиентом, становится негодным к повторному использованию.

Из упомянутых нюансов, можно сделать несколько практических выводов.

- 1) Для `TrDataSet`, в которых вы будете модифицировать BLOB поля, просто необходимо использовать опцию `poRefreshAfterPost`. (Альтернативой этой опции под Firebird 2 может являться использование в `UpdateSQL`, `InsertSQL` секций RETURNING возвращающих на клиента блоб поле. Синтаксис описан в документации к серверу в файле `README.returning.txt`.) В этом случае, сразу же после выполнения модифицирующего запроса, FIBPlus получит с сервера преобразованный BLOB_ID и подменит им тот, который уже стал невалидным.
- 2) Мы видим, что тело BLOB поля передается на сервер ДО выполнения модификации записи. Если по каким-либо причинам последующая модификация записи будет отвергнута сервером (например, через `constraints`), то при повторной попытке модификации нам придется передавать тело BLOB поля заново. Это может быть накладно и с точки зрения сетевого трафика, и с точки зрения «разбухания» базы. Поэтому иногда имеет смысл разделить два процесса: отдельным запросом делать модификации всех не BLOB полей, а после успешного завершения этих модификаций отдельно отправлять изменения BLOB полей. (для `TrFIBDataSet` с автогенерацией модифицирующих запросов в FIBPlus есть специальная опция, регулирующая такое поведение: `AutoUpdateOptions. SeparateBlobUpdate`).

Использование `TrFIBQuery` при работе с BLOB-полями

Если вы используете `TrFIBQuery` для работы с BLOB-полями, то общий принцип остается тем же - можно использовать либо файлы, либо методы работы с `TStream`. Например, мы можем написать следующую процедуру для сохранения всех изображений из нашей таблицы в файлы:

```
pFIBQuery.SQL: SELECT * FROM BIOLIFE
```

```
procedure TMainForm.Button2Click(Sender: TObject);
var
  Index: Integer;
begin
  with pFIBQuery1 do begin
    ExecQuery;
    Index := 1;
    while not Eof do begin
      FN('GRAPHIC').SaveToFile(IntToStr(Index) + '.bmp');
      Next;
      inc(Index);
    end;
  Close;
  end;
end;
```

Примечание: Метод FN является аналогом метода FieldByName.

Смысл кода, приведенного выше, совершенно очевиден: мы получаем все записи из таблицы BIOLIFE, в цикле берем от сервера очередную запись из запроса, сохраняем в файл значение поля GRAPHIC при помощи метода SaveToFile и запрашиваем следующую запись при помощи метода Next. Аналогичным образом мы могли бы присваивать значение BLOB-параметру:

```
pFIBQuery.SQL: INSERT INTO BIOLIFE (GRAPHIC) VALUES (?GRAPHIC)
```

```
procedure TMainForm.Button2Click(Sender: TObject);
var
  Index: Integer;
begin
  with pFIBQuery1 do begin
    Prepare;
    for Index := 1 to 3 do begin
      Params[0].LoadFromFile(IntToStr(Index) + '.bmp');
      ExecQuery;
    end;
    Transaction.Commit;
  end;
end;
```

Данный пример вставляет три новые записи в таблицу BIOLIFE и сохраняет в них изображения из некоторых файлов “1.bmp”, “2.bmp” и “3.bmp”.

Примечание: поскольку в данном примере для сохранения изменений использовался метод Commit, то необходимо перезапустить приложение, чтобы увидеть вставленные записи в DBGrid1.

Уникальные возможности FIBPlus: Client BLOB-filters. "Прозрачное" сжатие BLOB-полей..

Многие из вас знают о технологии BLOB фильтров в Firebird/InterBase. Эти пользовательские функции позволяют обрабатывать (т.е. кодировать/декодировать, сжимать и т.д.) BLOB поля на сервере прозрачно для клиентского приложения. Особенно это полезно, если вам нужно заархивировать BLOB поля в базе данных, так как для этого не нужно вносить изменения в клиентскую программу. Но, используя такой подход, вы не сможете снизить сетевой трафик, потому что сервер и приложение в любом случае будут обмениваться несжатыми полями.

В FIBPlus есть механизм клиентских BLOB фильтров, очень схожий с аналогичным механизмом на сервере. Но преимущество локальных BLOB фильтров FIBPlus в том, что они значительно снижают сетевой трафик приложения, если BLOB поле сжимается до отправки на клиентское приложение и распаковывается после того, как оно получено на клиенте. Вы можете сделать это путем регистрации двух процедур: чтения и записи BLOB полей в TrFIBDatabase. После этого FIBPlus будет автоматически использовать эти процедуры для того, чтобы обрабатывать все BLOB поля заданного типа во всех TrFIBDataSet'ах, используя один экземпляр TrFIBDatabase. Проиллюстрируем этот механизм примером:

Сначала мы создадим таблицу с BLOB полями и триггер для генерации уникальных значений первичного ключа:

```
CREATE TABLE "BlobTable" (
  "Id" INTEGER NOT NULL,
  "BlobText" BLOB sub_type -15 segment size 1);

ALTER TABLE "BlobTable" ADD CONSTRAINT "PK_BlobTable" PRIMARY KEY ("Id");
```

Обратите внимание, что sub_type должен иметь отрицательное значение!

Примечание: «Существует несколько predefined подтипов BLOB, которые встроены в InterBase. Все эти подтипы имеют неотрицательные номера, например subtype 0 - это данные неопределенного типа, subtype 1 - текст, subtype 2 - BLR (Binary Language Representation, см. глоссарий и главу "Структура базы данных InterBase") и т. д. Пользователь также может определять свои подтипы BLOB, которые могут иметь отрицательные значения.». Мир InterBase, А. Ковязин, С. Востриков.

Теперь положите следующие компоненты на форму:

```
pFIBDataSet1: TpFIBDataSet;
pFIBTransaction1: TpFIBTransaction;
pFIBDatabase1: TpFIBDatabase;
DataSource1: TDataSource;
DBGrid1: TDBGrid;
DBMemo1: TDBMemo;
Button1: TButton;
OpenDialog1: TOpenDialog;
```

Свяжите компоненты FIBPlus и сгенерируйте запросы для pFIBDataSet1 (только для таблицы "BlobTable") с SQL Generator. Получится следующая форма:

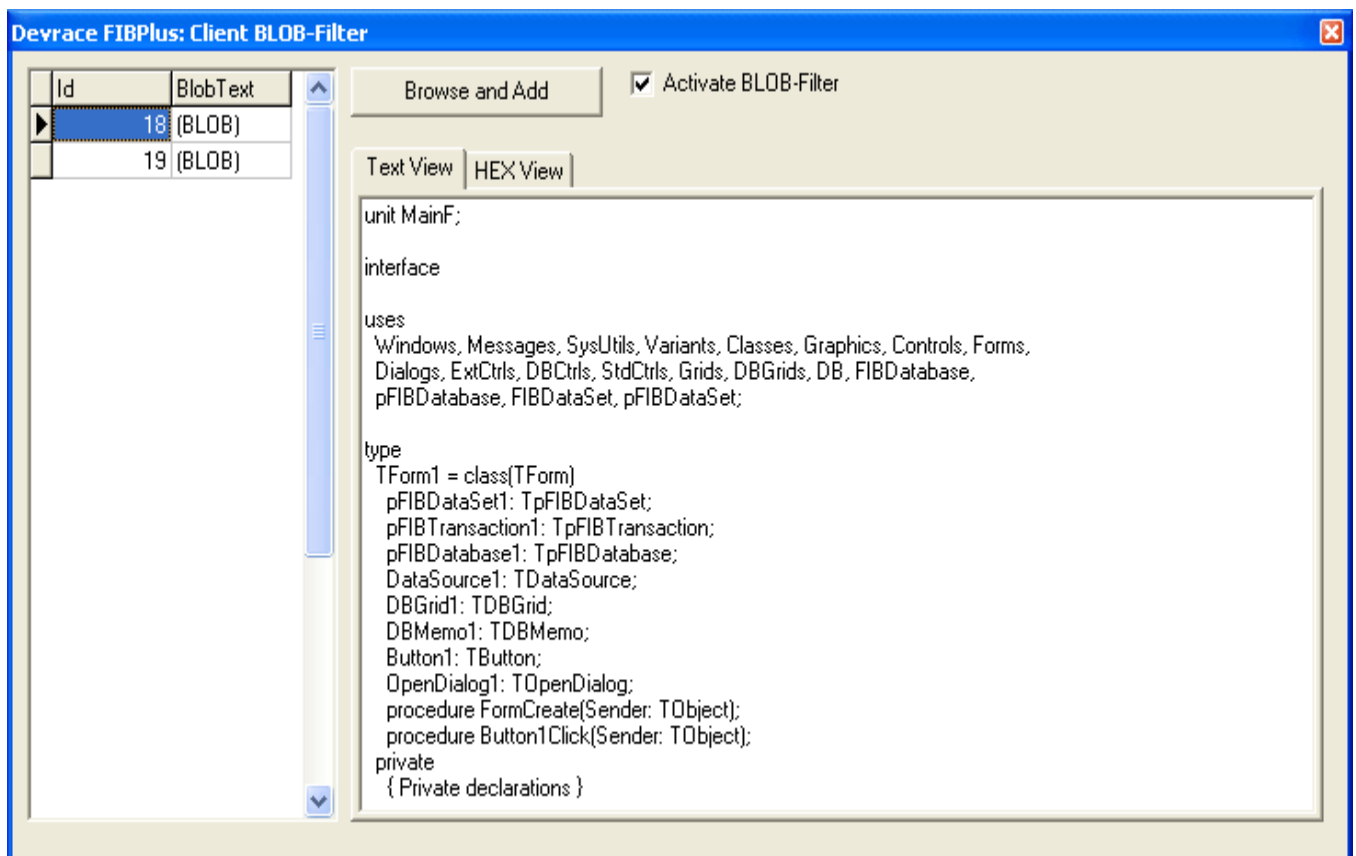


Рис.2. Приложение с использованием BLOB-фильтров FIBPlus

Напишем обработчик нажатия на кнопку:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if not OpenDialog1.Execute then exit;
    pFIBDataSet1.Append;
    TBlobField(pFIBDataSet1.FieldByName('BlobText')).LoadFromFile(OpenDialog1.FileName);
end;
```

```
pFIBDataSet1.Post;
```

```
end;
```

Теперь создадим функции сжатия/распаковки BLOB полей:

```
procedure PackBuffer(var Buffer: PChar; var BufSize: LongInt);
var srcStream, dstStream: TStream;
begin
  srcStream := TMemoryStream.Create;
  dstStream := TMemoryStream.Create;
  try
    srcStream.WriteBuffer(Buffer^, BufSize);
    srcStream.Position := 0;
    GZipStream(srcStream, dstStream, 6);
    srcStream.Free;
    srcStream := nil;
    BufSize := dstStream.Size;
    dstStream.Position := 0;
    ReallocMem(Buffer, BufSize);
    dstStream.ReadBuffer(Buffer^, BufSize);
  finally
    if Assigned(srcStream) then srcStream.Free;
    dstStream.Free;
  end;
end;
```

```
procedure UnpackBuffer(var Buffer: PChar; var BufSize: LongInt);
var srcStream, dstStream: TStream;
begin
  srcStream := TMemoryStream.Create;
  dstStream := TMemoryStream.Create;
  try
    srcStream.WriteBuffer(Buffer^, BufSize);
    srcStream.Position := 0;
    GunZipStream(srcStream, dstStream);
    srcStream.Free;
    srcStream:=nil;
    BufSize := dstStream.Size;
    dstStream.Position := 0;
    ReallocMem(Buffer, BufSize);
    dstStream.ReadBuffer(Buffer^, BufSize);
  finally
    if assigned(srcStream) then srcStream.Free;
    dstStream.Free;
  end;
end;
```

Не забудьте добавить два модуля в секцию **uses**: **zStream**, **IBBlobFilter**. Первый модель предназначен для создания архива с данными, второй входит в FIBPlus и отвечает за контроль над BLOB фильтрами. Теперь нам остается только зарегистрировать BLOB фильтры. Это можно сделать путем вызова функции **RegisterBlobFilter**. Значение первого параметра - это тип BLOB поля (в нашем случае оно равно -15). Второй и третий параметры - это функции кодирования и декодирования BLOB поля:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  pFIBDatabase1.RegisterBlobFilter(-15, @PackBuffer, @UnpackBuffer);
  pFIBDatabase1.Connected := True;
  pFIBDataSet1.Active := True;
end;
```

Запустите приложение, удалите записи, которые уже содержались в нем и добавьте новые. Вы не увидите никакой разницы, но если заглянуть на то, что реально было сохранено в BLOB поле, станет ясно, что данные были заархивированы:

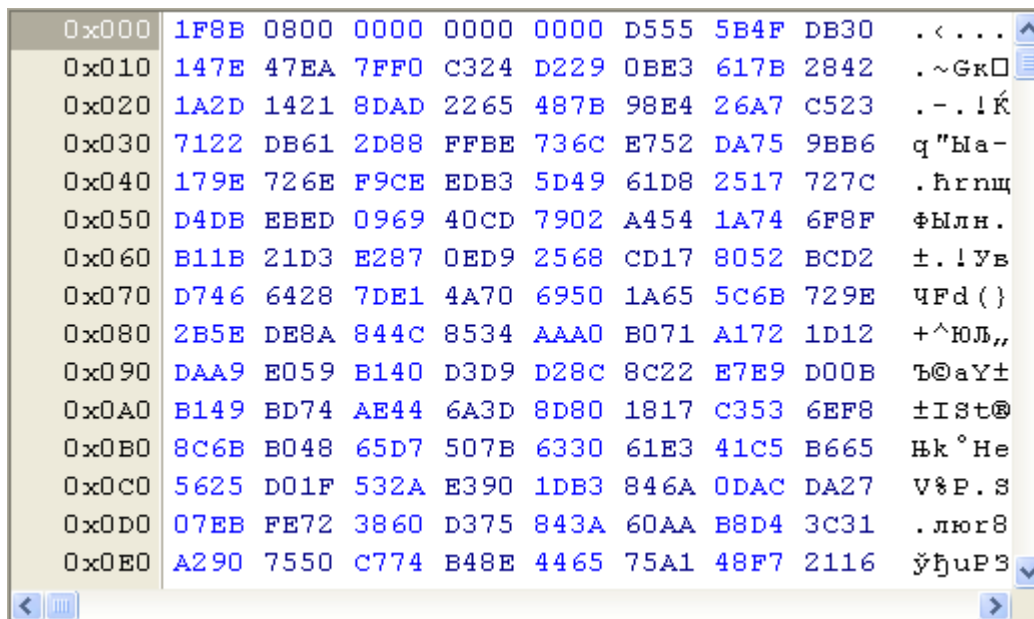


Рис. 3. Данные в BLOB-поле, упакованные при помощи локального фильтра FIBPlus.

Итак, приложение отсылает на сервер (и получает с сервера) уже заархивированные BLOB поля, и, таким образом, сетевой трафик значительно снижается! Конечно, вы можете упаковывать BLOB поля, не используя вышеописанный механизм BLOB фильтров. Например, вы можете сжимать поле в процедуре `Button1Click` перед его сохранением, а затем распаковывать его в обработчике `AfterScroll` (или делать что-то подобное). Но, во-первых, предлагаемый централизованный механизм значительно упрощает ваш код (так как BLOB поля обрабатываются независимо от других частей программы), а, во-вторых, он помогает избежать типичных ошибок (например, когда вы пакуете BLOB поля в одной части программы и не делаете то же самое в другой).

Централизованная обработка ошибок – `TrFIBErrorHandler`

FIBPlus позволяет централизованно обрабатывать ошибки и исключения, которые возникают при работе «своих» компонентов. Для этого существует компонент `TrFIBErrorHandler` с единственным событием `OnFIBErrorEvent`:

```
TOnFIBErrorEvent = procedure (Sender: TObject; ErrorValue: EFIBError;
  KindIBError: TKindIBError; var DoRaise: boolean) of object;
```

где

```
TKindIBError = (keNoError, keException, keForeignKey, keLostConnect,
  keSecurity, keCheck, keUniqueViolation, keOther);
```

```
EFIBError = class (EDatabaseError)
  //..
  property SQLCode : Long read FSQLCode ;
  property IBErrorCode: Long read FIBErrorCode ;
  property SQLMessage : string read FSQLMessage;
  property IBMessage : string read FIBMessage;
end;
```

Опция `DoRaise` управляет дальнейшим поведением библиотеки после выполнения

обработчика, то есть, указывает, будет ли генерироваться стандартное исключение или нет.

Опция может быть использована для стандартной обработки различных типов ошибок перечисленных в `KindIBError`.

Начиная с версии FIBPlus 6.4.2, в компонент были добавлены новые свойства, которые позволяют корректно работать с локализованными сообщениями сервера. Вы должны в свойстве `ErrorLexems` указать такие строковые подсвойства как `Index`, `Constraint`, `Exception` и `At`.

Исполнение скриптов `TrFIBScripter`

Данный компонент предназначен для анализа и исполнения скриптов. Если скрипт планируется исполнять в рамках конкретного соединения, то в компоненте необходимо установить свойство `Database`. Если в рамках конкретной транзакции, то необходимо установить и свойство `Transaction`. Важно то, что если свойство `Database` не установлено, то в скрипте обязан существовать стейтмент `CREATE DATABASE` или `CONNECT DATABASE`. Еще один важный момент, все соединения по умолчанию делаются с диалектом равным 3. Для того чтоб указать другой диалект, необходимо в начале скрипта прописать стейтмент

```
SET SQL DIALECT x
```

где `x` – номер диалекта.

Для выполнения скрипта компонент имеет два метода:

```
procedure ExecuteScript (FromStmt:integer=1);
procedure ExecuteFromFile(const FileName: string;Terminator:Char=';') ;
```

Первый метод выполняет скрипт, текст которого уже находится в памяти, и расположен в свойстве `Script:TStrings`. Скрипт выполняется начиная со стейтмента указанного во входном параметре `FromStmt`.

Второй метод выполняет скрипт, текст которого находится в файле. Важно знать, что `ExecuteFromFile` не загружает весь файл в память, а считывает файл построчно. Это с одной стороны позволяет обрабатывать очень большие скрипты (например скрипт содержащий не метаданные, а команды на заполнение базы). С другой стороны этот подход может несколько замедлять процесс исполнения, поэтому если у вас скрипты не сильно большие, то имеет смысл вместо `ExecuteFromFile` использовать следующий код

```
pFIBScripter1.Script.LoadFromFile(FileName);
pFIBScripter1.ExecuteScript;
```

Во время выполнения скрипта, вы можете отслеживать прогресс исполнения в обработчиках

```
property BeforeStatementExecute:TOnStatementExecute;
property AfterStatementExecute:TOnStatementExecute;
```

где

```
TOnStatementExecute = procedure (Sender: TObject;Line:Integer; StatementNo:
Integer; Desc:TStatementDesc; Statement: TStrings) of object;
```

Пример использования

```
procedure TForm1.pFIBScripter1BeforeStatementExecute(Sender: TObject; Line,
StatementNo: Integer; Desc: TStatementDesc; Statement: TStrings);
begin
StatusBar1.Panels[0].Text:='Execute Statement No '+IntToStr(StatementNo);
ProgressBar1.Max:=pFIBScripter1.StatementsCount;
```

```

ProgressBar1.Position:=StatementNo;
Application.ProcessMessages
end;

```

Приостановить выполнение скрипта можно используя свойство

```
property Paused: Boolean read FPaused write FPaused;
```

Ошибки исполнения конкретных стейтментов можно обрабатывать в обработчике

```

property OnExecuteError:TOnSQLScriptExecError;
TOnSQLScriptExecError = procedure(Sender: TObject; StatementNo: Integer;
    Line:Integer; Statement: TStrings; SQLCode: Integer; const Msg: string;
    var doRollBack:boolean; var Stop: Boolean) of object;

```

В нем вы можете проанализировать ошибку, сообщение об ошибке, узнать строку скрипта с которой начинается ошибочный стейтмент, проанализировать сам стейтмент и принять решение о завершении транзакции и завершении выполнения скрипта.

Кроме выполнения скрипта целиком, существует несколько методов для его анализа и выборочного выполнения некоторых стейтментов.

```
procedure Parse(Terminator:Char=';');
```

Выполняет анализ скрипта. С него явно или неявно начинается вся работа с текстом скрипта. Если вы запускаете скрипт на выполнение, то явного вызова метода Parse не требуется. Если же вы хотите анализировать скрипт или выборочно выполнить из него несколько стейтментов, то метод Parse придется вызвать явно.

```
function StatementsCount:integer;
```

Возвращает количество стейтментов в скрипте.

```
function GetStatement(StmtNo:integer;Text:TStrings):PStatementDesc;
```

По номеру стейтмента возвращает указатель на структуру

```

TStatementDesc = record
    smdBegin:TStmtCoord;
    smdEnd :TStmtCoord;
    smtType :TStmtType;
    objType :TObjectType;
    objName :string;
end;

```

По ней вы можете узнать координаты стейтмента в тексте, тип стейтмента и тип объекта который создается этим стейтментом. Кроме того, во входном параметре Text вы можете получить текст стейтмента с указанным номером.

```

procedure ExecuteStatement(StmtTxt:TStrings;stmt:PStatementDesc;StmtNo:integer;
    TmpSQL:TStrings=nil;LineInFile:integer=-1
);

```

Метод позволяет выполнить стейтмент который был предварительно получен методом GetStatement. Приведем пример исполнения скрипта начиная с 3 стейтмента.

```
procedure TForm1.ToolButton1Click(Sender: TObject);
```

```

var i:integer;
    p:PStatementDesc;
begin
    pFIBScripter1.Script:=Memo1.Lines;
    pFIBScripter1.Parse;
    for i:=3 to pFIBScripter1.StatementsCount do
        begin
            p:=pFIBScripter1.GetStatement(i,Memo2.Lines);
            pFIBScripter1.ExecuteStatement(Memo2.Lines,p,i,Memo2.Lines);
        end;
    end;
end;

```

Получение событий TFIBSibEventAlerter

Для получения событий БД используется компонент TFIBSibEventAlerter. Необходимо установить свойство Database, чтобы показать, события какого соединения будут отслеживаться; указать в Events имена отслеживаемых событий; и задать свойство Active равным True, чтобы активировать компонент.

При получении события, на которое подписан компонент, будет выполняться обработчик OnEventAlert, объявленный как:

```
procedure (Sender: TObject; EventName: String; EventCount: Integer);
```

где EventName – имя произошедшего события, EventCount их количество.

Помните, что события будут поступать только в момент подтверждения транзакции, в контексте которой оно наступило. Поэтому к моменту срабатывания ObEventAlert в приложении, могло произойти уже несколько событий.

Пример использование механизма событий смотрите в демонстрационном примере Events

Отладка приложений FIBPlus

FIBPlus предоставляет программисту мощные механизмы контроля и отладки SQL в приложениях. Для этого используются SQL-монитор и возможность автоматической сборки статистики SQL во время выполнения приложения.

Мониторинг SQL—запросов

За мониторинг SQL-запросов отвечает компонент TFIBSQLMonitor. Для его использования достаточно настроить тип интересующей вас информации в свойстве TraceFlags и написать обработчик события OnSQL. Эта возможность показана в демонстрационном примере SQLMonitor.

Регистрация выполняемых запросов

Компонент TFIBSQLLogger позволяет вести лог выполнения SQL запросов, а также вести статистику их выполнения.

Свойства:

property ActiveStatistics:boolean - включен ли сбор статистики.

property ActiveLogging:boolean - включено ли ведения лога

property LogFileName:string - имя файла, куда пишется лог

property StatisticsParams :TFIBStatisticsParams - какие именно

параметры включены в сбор статистики

property LogFlags: TLogFlags - какие операции логируются

property ForceSaveLog:boolean - вести ли запись лога сразу же. Т.е. после каждого запроса немедленно идет запись в файл.

Методы

function ExistStatisticsTable:boolean; - проверяет, существует ли таблица для хранения статистики

procedure CreateStatisticsTable; - создает таблицу для хранения статистики в базе данных

procedure SaveStatisticsToDB(ForMaxExecTime:integer=0); - сохраняет накопленную статистику в таблицу. Параметр указывает, по каким запросам статистика нам интересна, а параметр указывает нижний предел времени выполнения запроса. Т.е. записывается статистика только по тем запросам, которые выполнялись дольше или столько же времени, указанного в ForMaxExecTime.

procedure SaveLog; - запись лога в файл (имеет смысл, если выключен ForceSaveLog).

Использование этого компонента показано в демонстрационном примере SQLLogger.

Репозитории FIBPlus

FIBPlus предоставляет своим пользователям три встроенных репозитория для хранения и использования настроек TрFIBDataSet, входящих в них TFields и сообщений об ошибках. Для использования репозитория нужно позволить их использование – в компоненте TрFIBDatabase есть свойство UseRepositories (urFieldsInfo, urDataSetInfo, urErrorMessagesInfo), где вы должны указать, какие репозитории нужно использовать.

Работа со всеми типами репозитория демонстрируется в примерах DataSetRepository, ErrorMessageRepository, FieldsRepository.

Репозиторий наборов данных

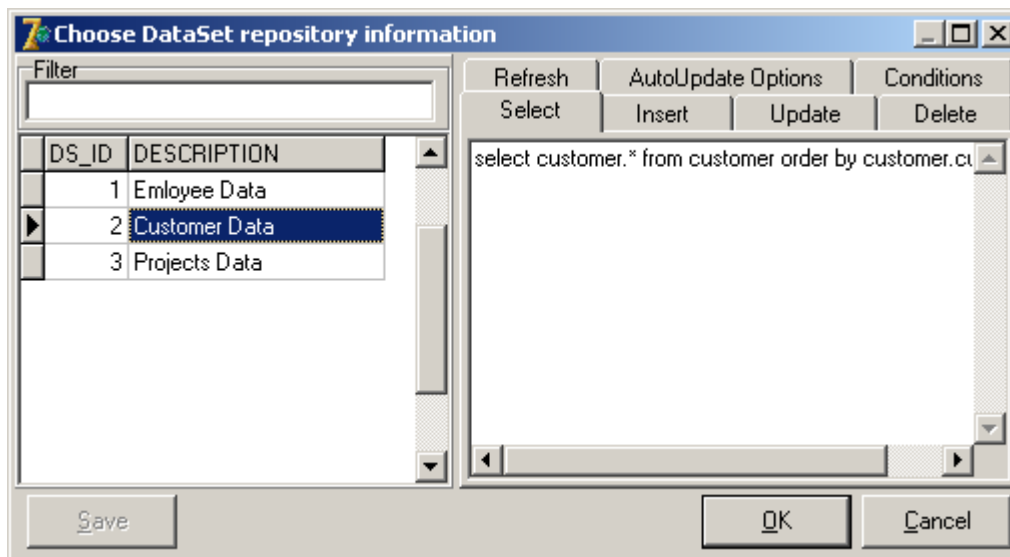


Рисунок 5. Диалог репозитория DataSets

Для использования этого репозитория в контекстном меню компонента `TrFIBDataSet` существует два пункта «Save to DataSets Repository table» и «Choose from dataSets Repository table». Первый позволяет сохранить основные свойства `TrFIBDataSet` в специальную таблицу `FIB$DATASETS_INFO`, а второй - загрузить свойства из ранее сохраненного в репозитории компонента. Если таблицы в БД не существует, вам будет предложено ее создать. Для того чтобы свойства конкретного `TrFIBDataSet` можно было сохранить в БД, вы должны указать значение свойства `DataSet_ID`, отличное от нуля.

Далее, в режиме выполнения программы достаточно просто установить `DataSet_ID` и свойства `TrFIBDataSet` будут загружены из таблицы базы данных.

Как можно увидеть из рисунка 5 сохраняются такие свойства как основные SQL-запросы, `Conditions` и `AutoUpdateOptions`.

Механизм репозитория добавляет новый уровень гибкости в ваши приложения и позволяет изменять его поведение без перекомпиляции – достаточно реплицировать таблицы репозитория.

Репозиторий полей

Доступ к репозиторию полей открывается через контекстное меню компонента `TrFIBDatabase` «Edit field information table».

Для любого поля таблицы, вида (views) и селективной процедуры вы можете задать такие свойства как `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat` и `EditFormat`. Значение `TRIGGERED` позволяет пометить поля, которые будут заполнены в триггере и для которых не требуется ввода значения в приложении, даже если поле помечено как `Required (NOT NULL)`.

Необходимо также установить параметр `psApplyRepository` в `TrFIBDataSet.PrepareOptions`, чтобы при открытии запроса настройки `TField` были взяты из репозитория.

При использовании алиасов для полей в SQL запросах вы можете обнаружить, что для таких полей настройки не применяются. И это правильно, поскольку алиасов нет в физических таблицах. Тем не менее, репозиторий полей позволяет ввести настройки и для таких полей. Для этого достаточно в репозитории вместо имени таблицы написать `ALIAS`.

Начиная с версии FIBPlus 6.4.2 в датасет добавлен стандартное событие:

TonApplyFieldRepository=**procedure**(DataSet:TDataSet;Field:TField;FieldInfo:TpFIBFieldInfo)
of object;

который позволяет разработчику легко использовать свои собственные настройки в репозитории полей. Например, если хочется настраивать свойство EditMask, то добавляем в таблицу репозитория поле EDIT_MASK, в приложении ставим контейнер, делаем его глобальным. В обработчике OnApplyFieldRepository пишем:

```
procedure TForm1.DataSetsContainer1ApplyFieldRepository(DataSet: TDataSet;
  Field: TField; FieldInfo: TpFIBFieldInfo);
begin
  Field.EditMask:=FieldInfo.OtherInfo.Values['EDIT_MASK'];
end;
```

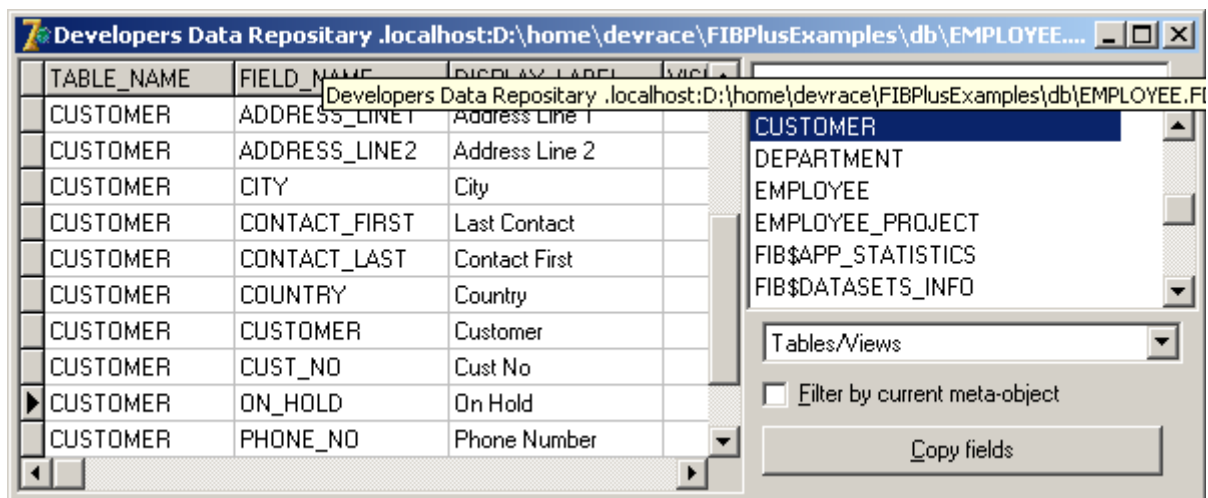


Рисунок 6. Репозиторий полей

Репозиторий ошибок

Доступ к репозиторию открывается через контекстное меню компонента TpFIBDatabase «Edit error messages table». Здесь вы можете задать свои собственные сообщения об ошибках для таких ситуаций как нарушение уникальности РК, ограничений unique, ограничений FK, ограничений checks и уникальных индексов.

Для использования репозитория необходимо поместить на какую-либо форму или модуль проекта компонент TpFIBErrorHandler, и включить использование этого репозитория. Все тексты возникающих ошибок, которые содержатся в репозитории, будут автоматически заменены вашими сообщениями.

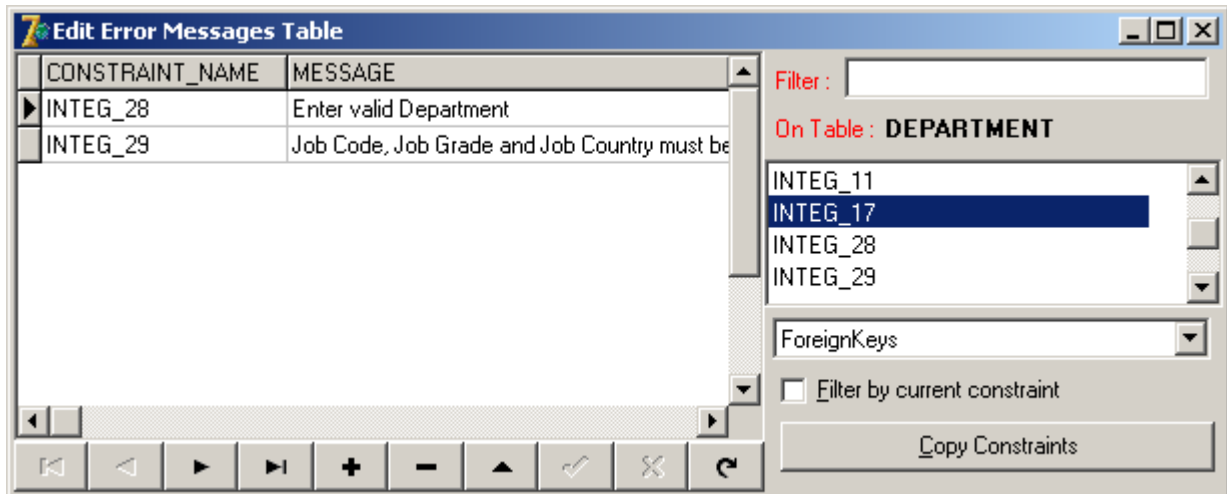


Рисунок 7. Репозиторий ошибок

Поддержка FB2.X

Несмотря на то, что на момент написания этого документа доступна только первая публичная бета-версия Firebird 2.0, FIBPlus полностью совместим с этой версией сервера.

Теперь в SelectSQL можно пользоваться конструкциями Execute block

Опция *poAskRecordCount* теперь во всех случаях работает правильно т.к. используется конструкция `select count from (select – ваша оригинальная выборка)`.

В компонент `TrFIBDatabase` добавлены два метода для поддержки новых команд `RDB$GET_CONTEXT` и `RDB$SET_CONTEXT` введенных в Firebird 2:

```
function GetContextVariable (ContextSpace: TFBCContextSpace; const VarName: string): Variant;
```

```
procedure SetContextVariable (ContextSpace: TFBCContextSpace; const VarName, VarValue: string);
```

FIBPlus поддерживает конструкцию FB2.0 `insert ... into ... returning`. Теперь можно не заботиться о получении с клиента значения генератора, а оставлять его на триггере. Появилась возможность использования `RDB$DB_KEY`. Новые возможные техники работы с `insert returning` и `RDB$DB_KEY` показаны в примере `FB2InsertReturnin`.

Если у вас есть запросы соединения таблицы с самой собой:

```
Select * from Table1 t, Table1 t1
where ....
```

то только в версии Firebird 2.0 для каждого поля FIBPlus может четко «понять», откуда оно взялось: из `t` или `t1`. Эта особенность используется внутри компонентов FIBPlus для правильной генерации модифицирующих запросов.

Дополнительные возможности

Полная поддержка UNICODE_FSS и UTF8

С версии 6.0 FIBPlus поддерживает правильную работу с CHARSET UNICODE_FSS. Версия 6.9.5 поддерживает так же и CHARSET UTF8.

Если вы используете более ранние чем Delphi 2009 версии, то вам для нормальной работы понадобятся визуальные компоненты, поддерживающие юникод, например TntControls.

В связи с поддержкой юникод добавлены новые типы полей для датасета:

TFIBWideStringField = class(TWideStringField) - для полей типа VARCHAR,CHAR;

TFIBMemoField = class(TMemoField, IWideStringField) - для BLOB-полей, где

IWideStringField - интерфейс обрабатываемый визуальными компонентами TNT

(http://tnt.ccci.org/delphi_unicode_controls)

Под версией Delphi 2009/Cbuilder 2009 возможно обойтись и без TFIBWideStringField, для этого служит директива {\$DEFINE UNICODE_TO_STRING_FIELDS} в файле FIBPlus.inc.

При включении этой директивы юникодные поля будут представлены классом

TFIBStringField. Для всех более ранних версий Delphi/CB ситуация довольно запутанная.

Попробуем ее прояснить.

- 1) **Коннект с юникодным чарсетом. (UNICODE_FSS или UTF8).** В этом случае все CHAR/VARCHAR поля, кроме полей с чарсетом NONE и OCTETS, будут представлены классом TFIBWideStringField.
- 2) **Коннект с чарсетом NONE.** В этом случае поля с чарсетом UNICODE_FSS или UTF8 будут представлены классом TFIBWideStringField, остальные поля - классом TFIBStringField
- 3) **Коннект с другими чарсетами.** В этом случае все поля представлены классом TFIBStringField.

Как мы видим, при разработке приложений под более ранними версиями Delphi, выбор чарсета коннекта нельзя сменить без переработки самого приложения. Поэтому этот момент надо учитывать на стадии проектирования.

Опция psSupportUnicodeBlobs в TрFIBDataSet.PrepareOptions позволяет работать с BLOB-полями UNICODE_FSS и UTF8. Если вы используете версию сервера Firebird 2.1 или более позднюю версию Firebird, то эта опция для вас излишня.

В TрFIBDatabase реализован метод: function IsUnicodeCharSet: Boolean, который возвращает True, если подключение использует UNICODE_FSS или UTF8.

Опция компиляции NO_GUI

Начиная с версии 6.0, в FIBPlus введена новая директива компиляции NO_GUI. При ее использовании библиотека не ссылается на стандартные модули, которые содержат визуальные компоненты, что позволяет получать приложения, ориентированные на системные, а не пользовательские задачи. {\$DEFINE NO_GUI} в файле FIBPlus.inc.

Использование SynEdit в редакторах

Если у вас в установлен набор компонентов SynEdit, вы можете скомпилировать пакеты

редакторов с использованием SynEdit. После этого в SQL-редакторе вам будет доступно синтаксическое выделение SQL операторов и инструмент CodeComplete. За использование SynEdit отвечает директива `{SDEFINE USE_SYNEDIT}` в файле `pFIBPropEd.inc`.

Важно: вы не можете изменять директивы компиляции в триальной версии FIBPlus.

Уникальное расширение FIBPlusTools

Кроме компонент библиотека FIBPlus также включает ряд дополнительных инструментов - FIBPlus Tools, которые расширяют возможности среды разработки для более удобного и эффективного использования FIBPlus в design-time.

Preferences

Пункт Preferences позволяет настроить параметры основных компонент по умолчанию. На первой странице диалога вы можете задать значения по умолчанию для свойств Options, PrepareOptions и DetailsConditions для всех компонент класса `TpFIBDataSet`. Вы можете задать определенные ключи для этих свойств. Например, если вы включите флажок `SetRequiredFields` то, когда вы положите новую компоненту `TpFIBDataSet` на вашу форму, ее свойство `PrepareOptions` будет содержать ключ `pfSetRequiredFields`. Наиболее важным является тот факт, что умолчания, заданные в FIBPlus Tools Preferences, действуют во всех приложениях, которые вы будете создавать. Однако необходимо иметь в виду, что это только первоначальные умолчания. То есть, если после помещения компоненты на форму вы измените свойства, то это никак не коснется Preferences. Также изменение Preferences не коснется тех компонент, значения свойств которых уже были заданы.

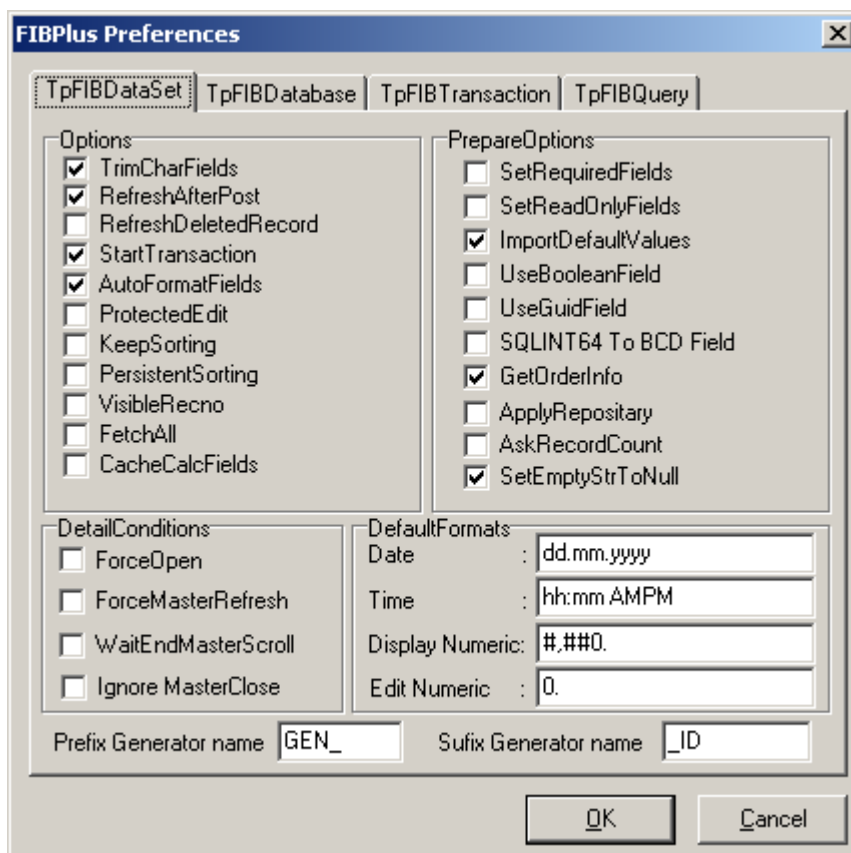


Рисунок 8. Tools Preferences.

Обратите внимание на поля Prefix Generator name и Suffix Generator name. Задав их значения, вы сможете регулировать формирование имен для названий генераторов в свойстве

AutoUpdateOptions у TrFIBDataSet. Имя генератора в AutoUpdateOptions генерируется из названия таблицы (UpdateTable), префикса и суффикса.

Следующие страницы диалога позволяют настраивать ключевые свойства компонентов TrFIBDataBase, TrFIBTransaction и TrFIBQuery. В частности, если вы всегда работаете с новыми версиями InterBase, то есть, с версиями 6 и более (а также Firebird), то мы рекомендуем вам задать значение SQL Dialect на закладке TrFIBDatabase равным 3, чтобы каждый раз не переключать это свойство «вручную».

SQL Navigator

Это наиболее интересная часть FIBPlus Tools, не имеющая аналогов в других продуктах. Фактически, это инструмент централизованной обработки SQL в рамках целого приложения.

SQL Navigator позволяет разработчику получить доступ к свойствам SQL любого компонента приложения и все это в одном месте.

Кнопка «Scan all forms of active project» сканирует все формы приложения и выделяет из них те, которые содержат компоненты FIBPlus для работы с SQL: TrFIBDataSet, TrFIBQuery, TrFIBUpdateObject и TrFIBStoredProc. Нажмите в списке на любую из обнаруженных форм. Список справа будет заполнен компонентами, обнаруженными на этой форме. Нажатие на любой из компонентов позволит нам посмотреть соответствующие свойства, в которых содержится SQL-код. Для компонентов класса TrFIBDataSet будут выведены свойства: SelectSQL, InsertSQL, UpdateSQL, DeleteSQL и RefreshSQL. Для компонент TrFIBQuery, TrFIBUpdateObject и TrFIBStoredProc будет выведено значение свойства SQL.

Вы можете изменить любое свойство напрямую из SQLNavigator, и новое значение будет сохранено. SQLNavigator позволяет делать операции с группами компонент. Для этого достаточно пометить соответствующие компоненты или даже формы.

“Save selected SQLs” сохраняет значения выделенных свойств во внешний файл.

“Check selected SQLs” проверяет корректность выделенных запросов прямо в SQLNavigator. Записанный файл с выделенными запросами можно использовать для дальнейшего анализа при помощи специализированных инструментов.

Вы можете использовать SQLNavigator для поиска текста в SQL в рамках всего проекта.

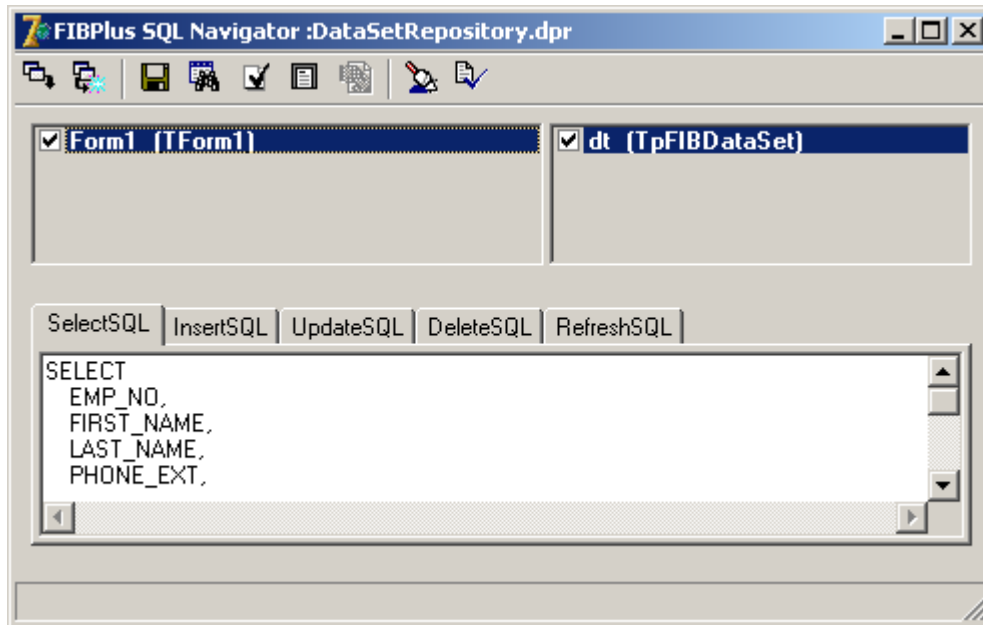


Рисунок 9. Tools SQL Navigator

При помощи двойного нажатия на каждом найденном элементе SQLNavigator выберет компонент и свойство, чтобы разработчик мог редактировать SQL.

Работа с сервисами

В дополнение к основной вкладке палитры компонентов FIBPlus библиотека содержит еще одну вкладку: FIBPlus Services. Собранные здесь компоненты предназначены для получения информации о сервере InterBase/Firebird, управления пользователями, тонкой настройки параметров баз данных и их обслуживания.

Подробную информацию по всем этим вопросам вы можете найти в документации к InterBase (OpGuide.pdf, разделы «Database Security», «Database Configuration and Maintenance», «Database Backup and Restore» и «Database and Server Statistic»).

Вся работа, которая может быть выполнена при помощи сервисов, показана в демонстрационном примере Services. Кроме того, есть хороший пример в поставке Delphi/C++Builder по пути DelphiX\Demos\Db\IBX\Admin\. Несмотря на то, что он написан с использованием IBX, этот пример подойдет и для FIBPlus.

Все компоненты сервисов унаследованы от класса TpfIBCustomService и имеют ряд обязательных свойств и методов. Это, в частности, ServerName - имя сервера, к которому будет производиться подключение, Protocol - сетевой протокол подключения, UserName - имя пользователя, и Password - пароль пользователя.

В общем случае работа с сервисом выглядит следующим образом. Сначала производится установка свойств подключения (сервер, протокол, пользователь и пароль). Потом присоединение к серверу (Attach := True), выполнение необходимых действий и отсоединение (Attach := False). Данная схема представлена в коде ниже и необходима при работе с любым сервисом.

```
with TpfIBXXXService.Create(nil) do
try
  ServerName := <server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  Attach;
  try
    //выполняемая работа
  finally
    Deattach;
  end;
finally
  Free;
end;
```

Получение информации о сервере

Вы можете получить информацию о лицензиях, конфигурации сервера, количестве подключенных пользователей и базах данных, используя компонент TpfIBServerProperties. О доступных свойствах, событиях и методах подробнее смотрите в Приложении. В общем случае вам нужно соединиться с сервером, установить интересующую вас информацию и получить ее при помощи метода Fetch.

Параметры и результаты описаны в модуле IB_Services.pas:

Следующая запись содержит информацию о количестве подключений к серверу, количестве активных БД, обслуживаемых сервером, и именах этих активных баз данных:

```
TDatabaseInfo = record
  NoOfAttachments: Integer;
  NoOfDatabases: Integer;
```

```

    DbName: Variant;
end;

```

Следующие две записи содержат информацию о лицензиях сервера InterBase:

```

TLicenseInfo = record
    Key: Variant;
    Id:      Variant;
    Desc:    Variant;
    LicensedUsers: Integer;
end;

```

```

TLicenseMaskInfo = record
    LicenseMask: Integer;
    CapabilityMask: Integer;
end;

```

```

TConfigFileData = record
    ConfigFileValue: Variant;
    ConfigFileKey: Variant;
end;

```

```

TConfigParams = record
    ConfigFileData: TConfigFileData;
    BaseLocation: string;
    LockFileLocation: string;
    MessageFileLocation: string;
    SecurityDatabaseLocation: string;
end;

```

Следующая запись содержит информацию о версии сервера:

```

TVersionInfo = record
    ServerVersion: String;
    ServerImplementation: string;
    ServiceVersion: Integer;
end;

```

```

TPropertyOption = (Database, License, LicenseMask, ConfigParameters, Version);
TPropertyOptions = set of TPropertyOption;

```

```

TpFIBServerProperties = class (TpFIBCustomService)
    procedure Fetch;
    procedure FetchDatabaseInfo;
    procedure FetchLicenseInfo;
    procedure FetchLicenseMaskInfo;
    procedure FetchConfigParams;
    procedure FetchVersionInfo;
    property DatabaseInfo: TDatabaseInfo
    property LicenseInfo: TLicenseInfo
    property LicenseMaskInfo: TLicenseMaskInfo
    property ConfigParams: TConfigParams
    property Options : TPropertyOptions
end;

```

Для получения лога работы сервера используется компонент TpFIBLogService.

Многие сервисы возвращают текстовую информацию. Такая информация возвращается при помощи события OnTextNotify типа TserviceGetTextNotify. Этот тип описан в модуле IB_Services следующим образом:

```

TServiceGetTextNotify = procedure (Sender: TObject; const Text: string) of

```

object;

TrFIBLogService использует именно такой тип оповещения. Работа с этими сервисами требует установки реакции на событие OnTextNotify старта сервиса и вычитывания информации. Выглядит работа по вычитыванию информации всегда одинаково:

```
ServiceStart;
while not Eof do
  GetNextLine;
```

Т.е. полный код будет выглядеть следующим образом:

```
with TrFIBXXXService.Create(nil) do
try
  ServerName := <server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  Atach;
  try
    ServiceStart;
    while not Eof do
      GetNextLine;
  finally
    Deattach;
  end;
finally
  Free;
end;
```

При работе с некоторыми другими аналогичными сервисами дополнительно требуется установка опций.

Подробности смотрите в примере Services.

Управление пользователями сервера

Работу с пользователями обеспечивает компонент TrFIBSecurityService. Он содержит методы для получения информации о пользователях, а также для добавления, модификация и удаления пользователей. Пользователю этого компонента доступна следующая информация:

Запись, предоставляющая информацию о пользователе на сервере:

```
TUserInfo = class
public
  UserName: string;
  FirstName: string;
  MiddleName: string;
  LastName: string;
  GroupID: Integer;
  UserID: Integer;
end;
```

Доступные свойства и методы компонента:

```
TrFIBSecurityService = class(TrFIBControlAndQueryService)
  procedure DisplayUsers;
  procedure DisplayUser(UserName: string);
  procedure AddUser;
  procedure DeleteUser;
```

```

procedure ModifyUser;
procedure ClearParams;
property UserInfo[Index: Integer]: TUserInfo read GetUserInfo;
property UserInfoCount: Integer read GetUserInfoCount;

property SqlRole : string read FSqlRole write FSqlRole;
property UserName : string read FUserName write FUserName;
property FirstName : string read FFirstName write SetFirstName;
property MiddleName : string read FMiddleName write SetMiddleName;
property LastName : string read FLastName write SetLastName;
property UserID : Integer read FUserID write SetUserID;
property GroupID : Integer read FGroupID write SetGroupID;
property Password : string read FPassword write setPassword;
end;

```

Для того, чтобы получить информацию о пользователе сервера, используется метод `DisplayUser`, параметром которого является имя интересующего нас пользователя. Для того, чтобы получить информацию обо всех пользователях, используется метод `DisplayUsers`. После вызова одного из этих методов свойство `UserInfoCount` будет содержать количество пользователей, а индексное свойство `UserInfo` будет возвращать запись о пользователе по индексу.

Для того, чтобы добавить пользователя, вам нужно заполнить свойства `UserName`, `FirstName`, `MiddleName`, `LasName`, `Password` и выполнить метод `AddUser`. Аналогично `AddUser` работают `DeleteUser` и `ModifyUser`; минимальным необходимым параметром для них будет имя пользователя `UserName`.

Обратите внимание на свойство `Password` – оно не возвращается методами `DisplayUser` и `DisplayUsers`. Также свойство `SQLRole` не может быть использовано для ассоциирования роли с пользователем. Используйте для этих операций выполнение запросов через `TrFIBQuery` - `GRANT/REVOKE`.

Все аспекты работы с пользователями сервера представлены в демонстрационном примере `Services`

Конфигурирование базы данных

`TrFIBConfigService` позволяет осуществлять настройку таких параметров, как:

- интервал в транзакциях до автоматической сборки мусора (`Sweep Interval`);
- установка режима записи изменений на диск (`Async Mode`);
- установка размера страницы БД;
- установка резервируемого пространства для БД;
- установка режима доступа только для чтения;
- активация и деактивация тени

А также позволяет выполнять такие действия, как:

- остановка базы данных (`shutdown`)
- старт базы данных (`online`).

Прежде всего, для работы с сервисом вам нужно установить свойство

```

property DatabaseName: string
Путь к базе данных на сервере

```

Чтобы установить интервал в транзакциях для автоматической сборки мусора, используется метод `SetSweepInterval`, единственным параметром которого нужно выставить интересующий вас интервал (по умолчанию он равен 20000)

```
procedure SetSweepInterval (Value: Integer);
```

Для установки диалекта базы данных используется метод `SetDBSqlDialect`, параметром которого есть диалект БД. Поддерживается всего три параметра 1, 2, 3.

```
procedure SetDBSqlDialect (Value: Integer);
```

Для установки `PageBuffers` используется метод `SetPageBuffers`, параметром которого есть интересующий размер буфера.

```
procedure SetPageBuffers (Value: Integer);
```

Для активации тени используется метод `ActivateShadow`. Параметры для этого метода не требуются.

```
procedure ActivateShadow;
```

Для установки режима асинхронной записи на диск служит метод `SetAsyncMode` с параметром Истина, для снятия режима передайте параметром Ложь

```
procedure SetAsyncMode (Value: Boolean);
```

Для установки режима только для чтения используется метод `SetReadOnly`. Передайте Истину, если хотите установить режим только для чтения и Ложь, если хотите снять. Режим только для чтения используется для подготовки баз для распространения на носителях типа компакт-дисков.

```
procedure SetReadOnly (Value: Boolean);
```

Для установки ... служит метод `SetReserveSpace`

```
procedure SetReserveSpace (Value: Boolean);
```

Для остановки доступа к БД используется метод `ShutdownDatabase`. Возможны три стандартных варианта остановки БД – форсированное, с запретом новых транзакций и с запретом новых соединений. Все вышеперечисленные операции желательно выполнять с монопольным подключением к бд администратором сервера - SYSDBA

```
procedure ShutdownDatabase (Options: TShutdownMode; Wait: Integer);
```

```
TShutdownMode = (Forced, DenyTransaction, DenyAttachment);
```

Для возвращения БД в состояние доступное для подключения используется метод `BringDatabaseOnline`.

```
procedure BringDatabaseOnline;
```

Обслуживание базы данных

Компоненты `TrFIBBackupService` и `TrFIBRestoreService` открывают доступ к функциям резервирования и восстановления баз данных.

`TrFIBValidationService` позволяет выполнить сборку мусора, проверить базу данных на ошибки, и, в случае поломки, выполнить починку БД.

При резервировании, восстановлении и проверке баз данных большое влияние оказывают опции. Подробное описание всех этих опций содержится в документации к серверу `OpGuide.pdf`.

Работа с этими сервисам крайне проста и мало чем отличается от работы FIBLogService. Единственной сложностью может стать интерпретация результатов работы этих сервисов. Тут есть некоторые особенности.

Если нужно убедиться, что не было ошибок при резервировании или восстановлении БД нужно смотреть лог выполнения операции. При возникновении ошибок лог операции будет содержать строку вида «ГБАК: ERROR»

Если TrFIBValidationService не нашел ошибок, его лог будет состоять из одной пустой строки.

Если возникли ошибки при починке базы данных (опция Mend TrFIBValidationService), то они будут записаны в лог самого сервера.

Важно помнить, что при работе TrFIBBackupService файл бекапа базы данных создается на машине сервере – т.е. резервирование делает сервер и, соответственно, файл создается на сервере. Но резервировать можно файл с другого сервера. Тогда в строке подключения к базе данных нужно писать полный путь к БД. Выглядеть это будет примерно так:

```
with TrFIBXXXService.Create(nil) do
try
  ServerName := <local_backup_server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  DatabaseName := <remote_db_name>;
  BackupFile.Add(<local_backup_name>);
  Atach;
  try
    ServiceStart;
    while not Eof do
      GetNextLine;
  finally
    Deatach;
  end;
finally
  Free;
end;
```

Описанные сервисы могут работать как обычным сервером, так и со встроенным сервером. Единственное условие: не забудьте указать протокол Local при работе с Firebird Embedded Server.

Получение статистической информации о состоянии БД

При помощи компонента TrFIBStatisticalService можно получить важную статистическую информацию о работающей БД. Работа получения статистики ничем не отличается от работы с TrFIBLogService.

Приложение 1. FIBPlus 6.9.6 свойства, события, методы компонентов.

TrFIBDatabase

Компонент, инкапсулирующий соединение с базой данных сервера InterBase/Firebird

СВОЙСТВА

AliasName

Группа опций, которая позволяет задать псевдоним базы данных. Псевдоним хранится в реестре Windows и содержит информацию, необходимую для подключения к БД.

BlobSwapSupport

Группа опций, которая позволяет задать режим кэширования BLOB-полей на клиенте.

```

TBlobSwapSupport = class(TPersistent)
  property Active: Boolean default False;
  property SwapDir: string;
  property MinBlobSizeToSwap: Integer default 0;
end;

```

Подробное описание использования этой опции приведено в разделе [Кэширование блоб-полей на клиенте](#)

CacheShemaOptions

Группа опций, которая позволяет задать режим кэширования информации, получаемой из базы данных и от сервера InterBase, которая появляется при работе приложения: подготовленные запросы, служебную информацию FIBPlus о значениях по умолчанию для полей, и многое другое.

```

TCacheSchemaOptions =class(TPersistent)
  property LocalCacheFile: string;
  property AutoSaveToFile: Boolean .. default False;
  property AutoLoadFromFile: Boolean .. default False;
  property ValidateAfterLoad: Boolean .. default True;
end;

```

FIBPlus позволяет использовать эту информацию многократно, а также не терять ее между сеансами.

Свойство `LocalCacheFile` позволяет задать имя файла, в котором будет сохраняться эта информация. `AutoSaveToFile` отвечает за автоматическую запись кеша в файл при закрытии приложения. `AutoLoadFromFile` отвечает за загрузку кеша из файла. И, наконец, `ValidateAfterLoad` указывает, стоит ли проверять сохраненный кеш после загрузки.

Connected

Группа опций, которая позволяет управлять состоянием соединения с базой данных.

ConnectParams

Группа опций, которая позволяет задать основные параметры подключения.

```

TConnectParams=class(TPersistent)

  property UserName: string;
  property RoleName: string;
  property Password: string;
  property CharSet : string;
end;

```

DBName

Группа опций, которая позволяет задать строку для подключения. Строка подключения может включать имя сервера, протокол и базу данных. Так, например, для локального подключения пишется только имя файла:

```
D:\IB7Book\Program\Base\Bpexampl.ib
```

Для соединения с удаленным сервером по протоколу TCP/IP нужно дописать имя сервера:

```
netserver:D:\IB7Book\Program\Base\Bpexampl.ib
```

Для соединения с удаленным сервером по протоколу TCP/IP и порту 3051 нужно дописать порт

```
netserver/3051:D:\IB7Book\Program\Base\Bpexampl.ib
```

- Для протокола NetBEUI

```
\\netserver\D:\IB7Book\Program\Base\Bpexampl.
```

5. Для IPX/SPX (NetWare servers)

```
netserver@vol1:\IB7Book\Program\Base\Bpexampl.
```

- для UNIX или Linux сервера

```
netserver:/user/IB7Book/Program/Base/Bpexampl.
```

DBParams

Группа опций, которая позволяет задать параметры подключения. Частично пересекается с опциями [ConnectParams](#). Все возможные параметры можно добавлять напрямую сразу в буфер, например:

```
DBParams.Add('user_name=XUSER');
```

```
DBParams.Add('password=XUSER');
```

```
DBParams.Add('lc_ctype=WIN1251');
```

```
DBParams.Add('sql_role_name=XROLE');
```

Дополнительные параметры подключения можно посмотреть в документации к серверу (APIGuide.pdf, DevGuide.pdf).

DefaultTransaction

Группа опций, которая позволяет задать транзакцию по умолчанию. При присоединении компонентов TrFIBDataSet, TrFIBQuery, TrFIBStoredProc к компоненту TrFIBDatabase свойство Transaction этого компонента будет автоматически скопировано в свойство Transaction нового компонента.

DefaultUpdateTransaction

Группа опций, которая позволяет задать транзакцию по умолчанию для обновления в компонентах TrFIBDataSet. Т.е. при заполнении свойства Database компонента TrFIBDataSet будет автоматически установлено свойство UpdateTransaction.

DesignDBOptions

Группа опций, которая позволяет задать поведение компонента TrFIBDatabase в режиме проектирования.

```
TDesignDBOption = (ddoIsDefaultDatabase, ddoStoreConnected, ddoNotSavePassword);
TDesignDBOptions = set of TDesignDBOption;
```

`ddoIsDefaultDatabase` определяет, будет ли данный компонент соединением по умолчанию для добавляемых компонентов `TpFIBDataSet`, `TpFIBQuery`, `TpFIBStoredProc`. Если опция включена, то при добавлении новых компонентов в проект им будет автоматически присваиваться свойство `Database` и, как следствие, `Transaction` и `UpdateTransaction`.

`ddoStoreConnected` определяет, будет ли сохраняться состояние подключения при компиляции приложения.

`ddoNotSavePassword` определяет, будет ли сохраняться пароль для подключения в свойствах компонента. Если опция включена, то пароль сохранен не будет.

LibraryName

Группа опций, которая позволяет задать имя клиентской библиотеки InterBase/Firebird. Данная опция может быть изменена только в зарегистрированной версии FIBPlus.

SaveAliasParamsAfterConnect

Если включена эта опция, то при удачном соединении с базой данных информация будет записана в реестр Windows как клиентский псевдоним (псевдоним задается в свойстве AliasName)

SQLDialect

Эта опция позволяет задать диалект коннекта к базе данных

SQLLogger

Эта опция позволяет задать компонент для логгирования обращений к БД. Подробнее смотрите в описании компонента TSQLLogger.

SynchronizeTime

Если включена эта опция, то при успешном подключении к БД время рабочей станции будет синхронизировано с временем на сервере.

Timeout

Если значение свойства **Timeout** отлично от нуля, то оно определяет время допустимого простоя в миллисекундах. Если за указанный промежуток времени через коннект не прошло ни одного действия (запроса, фетча данных, фетча блоба), то автоматически будет вызвано событие `OnIdleConnect`. Если на это событие нет обработчика, то коннект будет автоматически закрыт.

UpperOldNames

Если установлена эта опция, все имена в SQL-запросах будут переводиться в верхний регистр. Это позволяет избежать ошибок с именами объектов при написании запросов.

UseLoginPrompt

Если установлена эта опция, то, несмотря на пароль, прописанный в информации для подключения, будет вызваться диалог подключения.

UseRepositories

Эта опция позволяет задать режим использования репозиторий:

```
TFIBUseRepository = (urFieldsInfo,urDataSetInfo,urErrorMessagesInfo);
TFIBUseRepositories = set of TFIBUseRepository;
```

`urFieldsInfo` использовать репозиторий информации о полях

`urDataSetInfo` использовать репозиторий информации о датасетах

`urErrorMessagesInfo` использовать репозиторий информации об ошибках

Подробно работа с репозиториями описывается в разделе «Работа с репозиториями». Также Вы можете посмотреть примеры из комплекта FIBPlusExamples:

FIBPlusExamples\src\DataSetRepository\

FIBPlusExamples\src>ErrorMessagesRepository\

FIBPlusExamples\src\FieldsRepository\

WaitForRestoreConnect

Эта опция позволяет задать интервал времени в миллисекундах, по истечении которого будет производиться очередная попытка восстановить соединение. Подробнее об обработке потери соединения Вы можете прочитать в разделе [Обработка потери соединения](#)

UseBlrToTextFilter

Включает и выключает обработку blob полей с подтипами Blr (2), Acl(3). Дело в том что blob поля с подтипом 2 и более используется сервером для внутренних целей. Например в блобах с подтипом 2 хранятся скомпилированные в blr код триггера и процедуры. Если включить опцию UseBlrToTextFilter, то для таких полей будет применен соответствующий фильтр.

Описание событий

AfterConnect

procedure (Sender: TObject);

Событие возникает после соединения с базой данных посредством использования метода Open или установки в истину свойства Connected.

AfterDisconnect

procedure (Sender: TObject);

Событие генерируется после отсоединения от БД методом Close или установкой в False свойства Connected.

AfterEndTransaction

procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force: Boolean);

Событие возникает после завершения транзакции, при этом передаются завершившаяся транзакция и действие, которым она завершилась.

TTransactionAction =(TARollback, TARollbackRetaining, TACCommit, TACCommitRetaining)

AfterLoadBlobFromSwap

procedure (const TableName, FieldName: String; RecordKeyValues: array of Variant; const FileName: String);

Событие возникает после загрузки BLOB-поля с диска.

AfterRestoreConnect

procedure (Database: TFIBDatabase);

Событие возникает после того, как соединение было успешно восстановлено после разрыва. Подробное описание работы смотрите в разделе "Обработка потери соединения" и в демонстрационном примере:

FIBPlusExamples\src\ConnectionLost\

AfterSaveBlobToSwap

procedure (const TableName, FieldName: String; RecordKeyValues: array of Variant; const FileName: String);

Событие возникает после сохранения BLOB-поля на диск

AfterStartTransaction

procedure (Sender: TObject);

Событие возникает после старта транзакции.

BeforeConnect

procedure (Database: TFIBDatabase; LoginParams: TStrings; var DoConnect:

Boolean);

Событие возникает непосредственно перед попыткой соединения с БД.

BeforeDisconnect

procedure (Sender: TObject);

Событие возникает перед отсоединением от БД.

BeforeEndTransaction

procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force: Boolean);

Событие возникает перед завершением транзакции.

BeforeLoadBlobFromSwap

procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force: Boolean);

Событие возникает перед загрузкой BLOB-поля из кеша. Подробнее смотрите раздел [Кэширование блоб-полей на клиенте](#)

BeforeSaveBlobToSwap

procedure (const TableName, FieldName: String; RecordKeyValues: array of Variant; Stream: TStream; var FileName: String; var CanSave: Boolean);

Событие возникает перед сохранением BLOB-поля на диск.

BeforeStartTransaction

procedure (Sender: TObject);

Событие возникает перед стартом транзакции.

OnAcceptCacheSchema

procedure (const ObjName: String; var Accept: Boolean);

Событие возникает в момент принятия решения о том, стоит ли загружать информацию из кеша для объекта ObjName. Здесь вы можете разрешить или запретить загрузку кеша объекта. Подробнее читайте в разделе [Кэширование метаданных](#).

OnErrorRestoreConnect

procedure (Database: TFIBDatabase; E: EFIBError; var Actions: TOnLostConnectActions);

Событие возникает при ошибке очередной попытки восстановления подключения к БД. При этом вы можете проанализировать полученную ошибку и задать действие, которое следует выполнить в этой ситуации. Подробнее читайте в разделе [Обработка потери соединения](#).

TOnLostConnectActions = (laTerminateApp, laCloseConnect, laIgnore, laWaitRestore);

Действие может принимать одно из следующих значений

laTerminateApp	закрыть приложение;
laCloseConnect	закрыть соединение;
laIgnore	игнорировать;
laWaitRestore	продолжить ожидание восстановления соединения.

OnLostConnect

procedure (Database: TFIBDatabase; E: EFIBError; var Actions: TOnLostConnectActions);

Событие возникает в момент потери соединения. Подробное описание работы события смотрите в разделе "Обработка потери соединения" и в демонстрационном примере FIBPlusExamples\src\ConnectionLost\OnTimeout

procedure (Sender: TObject);

Событие возникает в момент превышения времени таймаута соединения или транзакции.

Public свойства

DBParamByDPB[const Idx: Integer]: string;

Свойство позволяет получить строковое значение параметра подключения по индексу

FIBBaseCount: Integer;

Свойство позволяет задать количество объектов сервера БД, TrFIBTransaction, TrFIBDataSet, TrFIBQuery.

FIBBases[Index: Integer]: TFIBBase;

Свойство позволяет получить объект БД по индексу

Handle: TISC_DB_HANDLE;

Дескриптор подключения к базе данных

HandleIsShared: Boolean;

Свойство возвращает True, если текущее подключение «разделяемое». Т.е., если соединение в TrFIBDatabasee получилось путем присвоения свойству Handle значения из другого, уже существующего Handle. Актуально при использовании внутри dll.

TransactionCount: Integer;

Свойство позволяет получить количество транзакций

FirstActiveTransaction: TFIBTransaction;

Указатель на первую активную транзакцию

ActiveTransactionCount: Integer;

Свойство позволяет получить количество активных транзакций

Transactions[Index: Integer]: TFIBTransaction;

Свойство позволяет получить транзакцию по ее индексу

AttachmentID: Long;

Уникальный идентификатор подключения к БД. В **Firebird** начиная с версии **1.5** его можно так же получить из контекстной серверной переменной CURRENT_CONNECTION.

Changed

property Busy:boolean;

Возвращает true, если в рамках данного соединения в данный момент выполняется вызов IB API. Имеет смысл анализировать только из параллельного треда.

Allocation: Long;

Информация от вызова IB API функции isc_database_info с параметром isc_info_allocation. Подробности смотрите в документации к серверу (APIGuide.pdf)

BaseLevel: Long;

Информация от вызова IB API функции `isc_database_info` с параметром `isc_info_base_level`.
Подробности смотрите в документации к серверу (APIGuide.pdf)

DBFileName: string;

Свойство позволяет получить имя файла БД

DBSiteName: string;

Свойство позволяет получить имя хоста (сервера), на котором работает БД.

IsRemoteConnect: boolean;

Свойство возвращает True, если соединение удаленное

DBImplementationNo: Long;

Информация от вызова IB API функции `isc_database_info` с параметром `isc_info_implementation`.
Подробности смотрите в документации к серверу (APIGuide.pdf)

DBImplementationClass: Long;

Информация от вызова IB API функции `isc_database_info` с параметром `isc_info_implementation`.
Подробности смотрите в документации к серверу (APIGuide.pdf)

NoReserve: Long;

Информация от вызова IB API функции `isc_database_info` с параметром `isc_info_no_reserve`.
Подробности смотрите в документации к серверу (APIGuide.pdf)

ODSMajorVersion: Long;

Свойство позволяет получить младшую цифру версии ODS

ODSMajorVersion: Long;

Свойство позволяет получить старшую цифру версии ODS

PageSize: Long;

Свойство позволяет получить размер страницы БД

Version: string;

Свойство позволяет получить строковое сообщение о версии сервера.

FBVersion: string;

Свойство позволяет получить строковое сообщение о версии сервера. Актуально только для клонов Firebird. Если строка пустая, то версия сервера InterBase, иначе - Firebird.

ServerMajorVersion: integer;

Свойство позволяет получить старшую цифру версии сервера

ServerMinorVersion: integer;

Свойство позволяет получить младшую цифру версии сервера

ServerBuild: integer;

Свойство позволяет получить номер сборки сервера

ServerRelease: integer;

Свойство позволяет получить номер релиза сервера

CurrentMemory: Long;

Свойство возвращает True, если текущая память занята сервером.

ForcedWrites: Long;

Свойство позволяет получить значение ForcedWrites для подключенной БД

MaxMemory: Long;

Свойство позволяет получить максимальный размер памяти, который занимал сервер для работы с данной БД.

SweepInterval: Long;

Свойство позволяет получить количество транзакций, после которого стартует автоматическая сборка мусора.

UserNames: TstringList;

Свойство позволяет получить список подключенных пользователей, для Firebird 1.5 - только для SuperServer.

TrFIBDatabase предоставляет доступ к функциям IB API. Подробное описание этих параметров можно найти в документации сервера (APIGuide.pdf, OpGuide.pdf)

NumBuffers: Long;

Fetches: Long;

Marks: Long;

Reads: Long;

Writes: Long;

BackoutCount: TstringList;

DeleteCount: TstringList;

ExpungeCount: TstringList;

InsertCount: TstringList;

PurgeCount: TstringList;

ReadIdxCount: TstringList;

ReadSeqCount: TstringList;

UpdateCount: TstringList;

IndexedReadCount[const TableName:string]: integer;

NonIndexedReadCount[const TableName:string]: integer;

InsertsCount[const TableName:string]: integer;

UpdatesCount[const TableName:string]: integer;

DeletesCount[const TableName:string]: integer;

AllModifications: integer;

LogFile: Long;

CurLogFileName: string;

CurLogPartitionOffset: Long;

Префикс WAL относится к серверам под NewWare.

NumWALBuffers: Long;

WALBufferSize: Long;

WALCheckpointLength: Long;

WALCurCheckpointInterval: Long;

WALPrvCheckpointFilename: string;

WALPrvCheckpointPartOffset: Long;

WALGroupCommitWaitUSecs: Long;

WALNumIO: Long;

WALAverageIOSize: Long;

WALNumCommits: Long;

WALAverageGroupCommitSize: Long;

DBSQLDialect: Word;

Это свойство позволяет получить диалект БД

ReadOnly: Long;

Это свойство возвращает True, если БД только для чтения

DatabaseName: string;

Это свойство позволяет получить имя БД

DifferenceTime: double;

Это свойство позволяет получить дельту времени на серверной машине и на машине клиента.

ServerActiveTransactions: TstringList;

Это свойство позволяет получить список активных транзакций на сервере

OldestTransactionID: Long;

Это свойство позволяет получить ID старейшей транзакции. Подробности читайте в документации по серверу.

OldestActiveTransactionID: Long;

Это свойство позволяет получить ID старейшей заинтересованной транзакции. Подробности читайте в документации по серверу

ClientLibrary: TlibClientLibrary;

Это свойство позволяет получить интерфейсную ссылку на клиентскую библиотеку

SQLStatisticsMaker: ISQLStatMaker;

Это свойство позволяет интерфейсную получить ссылку на счетчик статистики

Методы

procedure RegisterBlobFilter (BlobSubType:integer; EncodeProc, DecodeProc: PIBBlobFilterProc);

Этот метод позволяет зарегистрировать Blob-Filter для подтипа BlobSubType

procedure RemoveBlobFilter (BlobSubType:integer);

Этот метод позволяет удалить Blob-Filter для подтипа BlobSubType

procedure CheckActive;

Этот метод позволяет проверить, подключена ли БД. В том случае, если БД не подключена, метод генерирует исключение.

procedure CheckInactive;

Этот метод противоположен предыдущему методу CheckActive. Он позволяет проверить, не подключена ли БД. В том случае, если БД подключена, метод генерирует исключение.

procedure CheckDatabaseName;

Этот метод проверяет, заполнено ли имя базы данных. Если имя БД не заполнено, он генерирует исключение.

procedure Close;

Этот метод позволяет закрыть соединение.

procedure CreateDatabase;

Этот метод позволяет создать БД. Базовые параметры подключения должны быть заполнены.

procedure DropDatabase;

Этот метод позволяет физически удалить БД.

function FindTransaction(TR: TFIBTransaction): Integer;

Этот метод возвращает индекс транзакции в локальном списке транзакций.

procedure ForceClose;

Этот метод позволяет принудительно закрыть БД.

function IndexOfDBConst(const st: string): Integer;

Этот метод возвращает индекс константы БД

procedure Open; **virtual**;

Этот метод позволяет открыть БД

function TestConnected: Boolean;

Этот метод позволяет тестировать подключение к БД. Он возвращает True в случае удачной попытки подключения.

function GetServerTime:TDateTime;

Этот метод возвращает серверное время.

function ClientVersion: string;

Этот метод возвращает версию клиентской библиотеки (WI-V6.3.2.4731 Firebird 1.5) .

function ClientMajorVersion:integer;

Этот метод возвращает старшую цифру версии клиенткой библиотеки.

function ClientMinorVersion: Integer;

Этот метод возвращает младшую цифру версии клиенткой библиотеки.

function IsFirebirdConnect: Boolean;

Этот метод возвращает значение True, если есть соединение с сервером Firebird.

function IsIB2007Connect: Boolean;

Этот метод возвращает значение True, если есть соединение с сервером Interbase2007.

function NeedUnicodeFieldsTranslation: Boolean;

Это внутренний флаг. Для клиентской библиотеки метод возвращает информацию о том, нужна ли UTF перекодировка для полей UNICODE_FSS. Данная информация нужна не всегда, это зависит от CHAR_SET подключения.

Changed

function IsUnicodeConnect :boolean;

Этот метод возвращает True, если это чарсет соединения UNICODE_FSS или UTF8.

function GetContextVariable (ContextSpace:TFBContextSpace;**const** VarName:string ;
aTransaction: TFIBTransaction=nil): Variant;

TFBContextSpace = (csSystem, csSession, csTransaction);

Этот метод возвращает значение контекстной переменной. Только для Firebird версии 2.0.

```
procedure SetContextVariable( ContextSpace:TFBContextSpace;const
VarName,VarValue:string ;aTransaction:TFIBTransaction = nil);
```

Этот метод позволяет установить значение контекстной переменной. Только для Firebird версии 2.0.

```
procedure StartTransaction;
```

Этот метод позволяет стартовать транзакцию, которая прописана в свойстве DefaultTransaction.

```
procedure Commit;
```

Этот метод позволяет завершить по Commit транзакцию, которая прописана в свойстве DefaultTransaction.

```
procedure Rollback;
```

Этот метод позволяет завершить по Rollback транзакцию, которая прописана в свойстве DefaultTransaction.

```
procedure CommitRetaining;
```

Этот метод позволяет завершить по CommitRetaining транзакцию, которая прописана в свойстве DefaultTransaction.

```
procedure RollbackRetaining;
```

Этот метод позволяет завершить по RollbackRetaining транзакцию, которая прописана в свойстве DefaultTransaction.

```
function Gen_Id(const GeneratorName: string; Step: Int64; aTransaction:
TFIBTransaction = nil): Int64;
```

Этот метод позволяет получить значение генератора с именем, шагом и в определенной транзакции. Если транзакция не задана, то она будет создана автоматически.

```
function Execute(const SQL: string): boolean;
```

Этот метод позволяет выполнить sql-запрос и возвращает True в случае удачного выполнения запроса.

```
procedure CreateGUIDDomain;
```

Этот метод позволяет создать домен FIBGUID char(16) character set octets в БД.

```
function QueryValue(const aSQL: string; FieldNo: integer; aTransaction:
TFIBTransaction=nil): Variant; overload;
```

Этот метод позволяет получить значение поля с номером FieldNo запроса с текстом aSQL. Если транзакция не задана, то она будет создана автоматически.

```
function QueryValue(const aSQL: string;FieldNo:integer; ParamValues:array of
variant;aTransaction:TFIBTransaction=nil ):Variant; overload;
```

Эта функция аналогична функции, описанной выше, но дает возможность передавать параметры.

```
function QueryValues(const aSQL:
string; aTransaction:TFIBTransaction=nil):Variant; overload;
```

Эта функция аналогична функции, описанной выше, но позволяет получить запись целиком в вариантный массив.

```
function QueryValues(const aSQL: string; ParamValues:array of variant;
aTransaction: TFIBTransaction=nil ): Variant; overload;
```

Эта функция аналогична функции, описанной выше, но позволяет получить запись целиком в вариантный массив и, кроме того, передавать параметры.

```
function QueryValueAsStr(const aSQL: string;FieldNo:integer):string; overload;
```

Эта функция возвращает строковое представление поля.

```
function QueryValueAsStr(const aSQL: string;FieldNo:integer; ParamValues:array of variant):string; overload;
```

Эта группа перегруженных методов QueryValueAsStr позволяет получить результатом строковое представление выполнения SQL-запроса первой строки, которую он выполняет.

Внимание: Семь методов, описанные выше, должны возвращать только одну запись!

```
function EasyFormatsStr: Boolean;
```

Этот метод позволяет выяснить, нужно ли обрамлять кавычками имена таблиц и полей или нужно приводить их к верхнему регистру. (Это метод для внутреннего использования)

New Changed

```
procedure CancelOperationFB21 (ConnectForCancel:TFIBDatabase=nil);
```

Позволяет прекратить выполнение «долгоиграющих» запросов. Если сам запрос выполняется в параллельном треде, то метод должен быть вызван из главного. Если же прерываемый запрос выполняется в главном, то метод вызывается из параллельного. Метод использует системную таблицу «MON\$STATEMENTS» и выполняется в параллельном соединении, которое ему передается в качестве входного параметра. Если входного параметра нет, то дополнительное соединение будет автоматически создано и закрыто после выполнения метода. Работает для всех версий Firebird начиная с версии 2.1.

```
procedure RaiseCancelOperations;
```

```
procedure EnableCancelOperations;
```

```
procedure DisableCancelOperations;
```

Позволяет прекратить выполнение «долгоиграющих» запросов другим способом.

Использует возможность на уровне API сервера, которая введена начиная с версии Firebird 2.5. Подробнее смотрите в документации к серверу файл «README.fb_cancel_operation.txt». В FIBPlus эта возможность реализуется набором методов TFIBDatabase

```
procedure .ClearQueryCacheList;
```

Уничтожает все запросы из списка закэшированных. См тему «Повторное использование запросов» из руководства пользователя.

```
function UnicodeCharSets:TIBCharSets;
```

Возвращает множество идентификаторов чарсетов которые представляют уникальные данные. (В разных серверах это множество разное, поэтому метод обрабатывает корректно

только после соединения с БД.)

function BytesInUnicodeChar(CharSetId:integer):Byte;

Возвращает размер указанного уникадного чарсета в байтах.

TrFIBTransaction

Это очень важный компонент, инкапсулирующий транзакцию, без которого невозможно обойтись.

Свойства

DefaultDatabase

Это свойство возвращает базу данных для транзакции

Timeout

Если значение свойства **Timeout** отлично от нуля, то оно определяет время допустимого простоя в миллисекундах. Если за указанный промежуток времени через транзакцию не прошло ни одного действия (запроса, фетча данных, фетча блоба), то автоматически будет выполнен TimeoutAction

TimeoutAction

Это свойство возвращает действие, которое выполнятся при наступлении события Timeout. Оно описано следующим образом:

```
TTransactionAction = (TARollback, TARollbackRetaining, TACCommit,
TACCommitRetaining);
```

TPBMode

Используя это свойство, можно настроить параметры транзакции. Его возможные параметры таковы:

```
TPBMode = (tpbDefault, tpbReadCommitted, tpbRepeatableRead)
```

TRParams

В этом свойстве можно задать свои собственные параметры

UserKindTransaction

Используя это свойство, можно выбрать пользовательский вариант настроек параметров транзакции. Наиболее удобно работать в редакторе «Edit Transaction Parameters», который можно вызвать из контекстного меню компонента.

TransactionID

Возвращает идентификатор транзакции. В **Firebird** начиная с версии **1.5** его можно так же получить из контекстной серверной переменной CURRENT_TRANSACTION.

State

```
property State: TTransactionState;
```

Свойство позволяет определить не только активность транзакции, но и находится ли она в состоянии завершения. Подробнее см. руководство пользователя тему «Получение информации о состоянии транзакции»

События

AfterEnd

```
procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force: Boolean);
```

Это событие генерируется при завершении транзакции

AfterSQLExecute

```
procedure (Query: TFIBQuery; SQLType: TFIBSQLTypes);
```

Это событие генерируется после выполнения SQL.

AfterStart

```
procedure (Sender: TObject);
```

Это событие генерируется после старта транзакции.

BeforeEnd

```
procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force: Boolean);
```

Это событие генерируется после завершения транзакции.

BeforeSQLExecute

```
procedure (Query: TFIBQuery; SQLType: TFIBSQLTypes);
```

Это событие генерируется перед выполнением запроса.

BeforeStart

```
procedure (Sender: TObject);
```

Это событие генерируется перед стартом транзакции

OnTimeout

```
procedure OnTimeOut(Sender: TObject);
```

Это событие вызывается сразу же после закрытия соединения по таймауту.

OnIdleConnect

```
TOnIdleConnect=procedure (Sender:TFIBDatabase; IdleTicks:Cardinal; var Action:TActionOnIdle);
```

```
TActionOnIdle=(aiCloseConnect, aiKeepLiveConnect);
```

Это событие вызывается по таймауту, в нем можно принять решение о закрытии или удержании соединения, а так же произвести любые дополнительные действия.

Параметры обработчика:

IdleTicks:Cardinal – время в миллисекундах прошедшее с последней активности соединения.

Action:TActionOnIdle – действие которое необходимо выполнить по завершению обработчика.

Методы

```
function MainDatabase:TFIBDatabase;
```

Этот метод возвращает основную базу данных.

```
function FindDatabase(db: TFIBDatabase): Integer;
```

Этот метод возвращает индекс базы данных **в рамках которой выполнятся транзакция.**

```
ew
```

```
function AddDatabase(db: TFIBDatabase):integer;
```

```
function AddDatabase(db: TFIBDatabase; const aTRParams: string);
```

Методы позволяют добавить в список коннектов еще один коннект. При старте транзакции она будет стартовать во всех коннектах, которые находятся в ее списке. Отличие этих

методов друг от друга в том, что если коннект был добавлен первым методом, то транзакция в этом коннекте стартует с теми параметрами, которые прописаны у нее в свойстве TRParams. Если же коннект был добавлен вторым методом, то транзакция **в этом коннекте** стартует с параметрами которые были переданы в метод. При этом в других коннектах, она может иметь другие параметры. Подробнее см. GUID тему «Двухфазный коммит»

procedure CheckInTransaction;

Этот метод генерирует исключение, если транзакция активна.

procedure CheckNotInTransaction;

Этот метод генерирует исключение, если транзакция не активна.

procedure StartTransaction; **virtual**;

Этот метод стартует транзакцию.

procedure Commit; **virtual**;

Этот метод подтверждает транзакцию по Commit.

procedure CommitRetaining; **virtual**;

Этот метод подтверждает транзакцию по commitRetaining.

procedure Rollback; **virtual**;

Этот метод откатывает транзакцию по Rollback.

procedure RollbackRetaining; **virtual**;

Этот метод откатывает транзакцию по RollbackRetaining.

procedure ExecSQLImmediate(**const** SQLText:string);

Этот метод выполняет запрос с использованием `isc_dsql_execute_immediate` без подготовки, и, как следствие, без рутинных операций с Handle запросов. Подробнее смотрите в IB API.

procedure SetSavePoint(**const** SavePointName:string);

Этот метод создает точку восстановления (Savepoint).

procedure RollBackToSavePoint(**const** SavePointName:string);

Этот метод отменяет все изменения, сделанные до точки восстановления (Savepoint).

procedure ReleaseSavePoint(**const** SavePointName: string);

Этот метод освобождает все точки восстановления (Savepoint).

procedure CloseAllQueryHandles;

Этот метод закрывает все Handle запросов, ассоциированных с этой транзакцией.

function IsReadCommittedTransaction:boolean;

Этот метод возвращает True, если транзакция имеет параметр ReadCommitted.

procedure StartTransaction; **override**;

Этот метод стартует транзакцию.

function FIBQueryCount: integer;

Этот метод возвращает количество запросов, связанных с транзакцией.

function FIBDataSetsCount:integer;

Этот метод возвращает количество датасетов, связанных с транзакцией.

TrFIBQuery

Свойства

Conditions

Смотрите подробное описание в разделе "Выполнение SQL-запросов. Условия" о TrFIBDataSet Руководства пользователя.

Database

Это свойство возвращает базу данных, для которой будет работать транзакция.

GoToFirstRecordOnExecute

Свойство влияет на выполнение селективных SQL. Если оно установлено в True, то первый fetch будет сделан сразу же после выполнения. Если оно установлено в False, то первый fetch будет сделан только после вызова Next. Опция имеет смысл только для селективных запросов.

Options

Это свойство аналогично свойствам TrFIBDataSet и описывается следующим образом:

```
TrFIBQueryOption = (qoStartTransaction, qoAutoCommit, qoTrimCharFields,
qoNoForceIsNull, qoFreeHandleAfterExecute);
TrFIBQueryOptions=set of TrFIBQueryOption;
```

`qoStartTransaction` - если включена, то перед выполнением запроса, если транзакция неактивна, то она будет стартовать автоматически.

`qoAutoCommit` - если включено, то сразу же после выполнения запроса транзакция, в рамках которой он был выполнен, будет завершена методом Commit. Внимание, если запрос селективный, то он тоже будет сразу же закрыт, и вы не сможете получить доступ к следующим записям.

`qoTrimCharFields` - определяет способ работы с CHAR-VARCHAR полями. Если опция включена, то методы `Fields.asString`, `Fields.asWideString` будут возвращать значения без хвостовых пробелов.

`qoFreeHandleAfterExecute` - указывает, что Handle запроса должен быть немедленно освобожден после выполнения в случае если запрос не селективный. Если запрос селективный, то Handle будет освобожден либо после выполнения метода Close, либо после того как все записи будут выбраны методом Next

`qoNoForceIsNull` - определяет надо ли преобразовывать текст SQL в случае использования параметров с NULL значениями. Например, допустим есть такой запрос:

```
Delete from table1 where Field1=:Field1
```

Если параметр Field1 принимает значение отличное от NULL, то этот запрос удалит все записи из таблицы, которые подпадают под условие запроса. Если же параметр принимает значение NULL, то запрос не удалит ни одной записи, так как для NULL значений условие всегда будет возвращать false. Для того чтоб удалить записи в которых поле Field1 принимает значение NULL требуется переписать запрос как

```
Delete from table1 where Field1 IS NULL.
```

Т.е. разработчику необходимо отслеживать этот момент и менять текст запроса, в зависимости от значения параметра. FIBPlus - упрощает эту работу, и изменяет текст запроса автоматически, в зависимости от текущего значения параметра. Если эта возможность разработчику не нужна, он может ее отключить вышеупомянутой опцией. Т.е. включение опции `qoNoForceIsNull` в Options компонента TrFIBQuery отключит данное преобразование.

ParamsCheck

Указывает производить ли парсинг текста запроса, для создания списка параметров, или нет. Если запрос не содержит параметров, то отключение этого свойства позволит сэкономить немного времени. Если же запрос является DDL запросом, то крайне желательно это свойство установить в false.

SQL

Это свойство возвращает текст выполняемого запроса

Transaction

Это свойство возвращает транзакцию, в контексте которой будет выполнен запрос.

СВОЙСТВА

AfterExecute

```
procedure (Sender: Tobject); TnotifyEvent;
```

Это событие генерируется после выполнения SQL-запроса (вызова TpFIBQuery.ExecQuery, ExecProc, ExecWP).

BeforeExecute

```
procedure (Sender: Tobject);
```

Это событие генерируется перед выполнением.

OnBatchError

```
procedure (E: EFIBError; var BatchErrorAction: TbatchErrorAction);
```

Это событие генерируется при ошибке выполнения пакетной обработки.

OnBatching

```
procedure (BatchOperation: TBatchOperation;  
  RecNumber: Integer; var BatchAction: TbatchAction);
```

Это событие генерируется при очередном выполнении пакетной обработки.

OnExecuteError

```
procedure (pFIBQuery: TpFIBQuery; E: EFIBError; var Action: TdataAction);
```

Это событие генерируется при ошибке выполнения запроса.

Вызывается в случае, если выполнение запроса завершилось ошибкой. В нем можно проанализировать ошибку и подменить стандартный диалог об ошибке, какими-то другими действиями.

OnSQLChanging

```
procedure (Sender: Tobject);
```

Это событие вызывается при изменении текста SQL.

TransactionEnded

```
procedure (Sender: Tobject);
```

Это событие вызывается после завершения транзакции.

TransactionEnding

```
procedure (Sender: Tobject);
```

Это событие вызывается при завершении транзакции.

Методы

```
function TableAliasForField(FieldIndex: integer) : string;  
function TableAliasForField(const aFieldName: string) : string;
```

Эти методы возвращают псевдоним таблицы для поля по индексу поля либо по его имени.

```
function BatchInput(InputObject: TFIBBatchInputStream) :boolean;  
function BatchOutput(OutputObject: TFIBBatchOutputStream) :boolean;  
procedure BatchInputRawFile(const FileName:string);  
procedure BatchOutputRawFile(const FileName:string;Version:integer=1);  
procedure BatchToQuery(ToQuery:TFIBQuery;Mappings:TStrings);
```

Методы пакетной обработки. Подробнее о методах для пакетной обработки читайте в разделе "Выполнение SQL-Запросов. Пакетная обработка" Руководства пользователя.

```
procedure CheckValidStatement;  
Этот метод проверяет SQL на валидность.
```

```
procedure Close;  
Этот метод закрывает запрос.
```

```
function Current: TFIBXSQlda;  
Этот метод возвращает буфер текущей записи.
```

```
procedure ExecQuery; virtual;  
Этот метод выполняет запрос.
```

```
procedure FreeHandle;  
Освобождает ресурсы сервера ассоциированные с данным запросом. Этот метод закрывает запрос с флагом DSQL_drop. При этом Handle запроса на сервере уничтожается и повторно его использовать уже нельзя.
```

```
function Next: TFIBXSQlda;  
Этот метод производит fetch следующей записи.
```

```
procedure Prepare;  
Производит подготовку запроса к выполнению. Если этот метод не вызван явно, то он будет вызван автоматически во время выполнения запроса.
```

Данные функции аналогичны одноименным функциям TpfIBDataSet:

Работа с полями запроса

```
function FieldName(FieldIndex:integer) :string;  
Этот метод возвращает поле по индексу.
```

```
function FieldsCount:integer;  
Этот метод возвращает количество полей.
```

```
function FieldExist(const FieldName:string; var FieldIndex:integer) :boolean;  
Этот метод проверяет существование поля с именем FieldName.
```

```
function FieldCount:integer;  
Этот метод возвращает количество полей запроса.
```

```
function FieldByName(const FieldName: string): TFIBXSQLVAR;
```

Возвращает ссылку на поле запроса по имени FieldName. Если поля с таким именем в коллекции полей запроса отсутствует, то генерируется ошибка.

```
function FindField(const FieldName: string): TFIBXSQLVAR;
```

```
function FN(const FieldName: string): TFIBXSQLVAR;
```

Возвращает ссылку на поле запроса по имени FieldName. Если поля с таким именем в коллекции полей запроса отсутствует, то возвращается значение **nil**.

```
function FieldByOrigin(const TableName,FieldName:string):TFIBXSQLVAR; overload;
```

Возвращает ссылку на поле запроса по имени таблицы, которой оно принадлежит (TableName) и имени поля в таблице (FieldName). Если такого поля в коллекции полей запроса отсутствует, то возвращается значение **nil**. Следует так же обратить внимание, что для диалекта 3, все имена таблиц и полей регистрочувствительные. Как следствие, в этот метод надо передавать имена таблицы и поля в том регистре, в котором они созданы в БД.

Рассмотрим вышеупомянутые функции на примере нижеследующего кода:

```
var
  F:TFIBXSQLVAR
begin
  pFIBQuery1.SQL.Text:='Select  Field1 F1 from Table1 T';
  pFIBQuery1.ExecQuery;
  F:= pFIBQuery1.Fields[0];
  F:= pFIBQuery1.FieldByName('F1');
  F:= pFIBQuery1.FindField ('F1');
  F:= pFIBQuery1.FieldByOrigin('TABLE1','FIELD1');
end;
```

Во всех случаях, в переменную F попадает ссылка на поле F1 запроса. Обратите внимание, что в последнем вызове мы передавали имя поля и таблицы в верхнем регистре.

```
procedure PrepareArrayFields;
```

```
procedure PrepareArraySqlVar( SqlVar:TFIBXSQLVAR;const RelName,SQLName:string;
  IsField:boolean );
```

Подробнее см. тему «Работа с полями-массивами» в руководстве пользователя.

```
procedure SetParamValues(const ParamValues: array of Variant); overload;
```

```
procedure SetParamValues(const ParamNames: string;ParamValues: array of
  Variant); overload;
```

```
procedure ExecWP(const ParamValues: array of Variant); overload;
```

```
procedure ExecWP(const ParamNames: string;ParamValues: array of Variant);
overload;
```

```
procedure ExecWPS(const ParamSources: array of ISQLObject);
```

Методы позволяют выполнить запрос и одновременно передать параметры этого запроса на сервер. Подробнее см. тему «Заполнение параметров» в руководстве пользователя.

```
function IsProc :boolean;
```

Этот метод установлен в True, если текст SQL - это выполнение процедуры (execute

procedure xxx).

procedure RestoreMacroDefaultValues;

Этот метод устанавливает значения макросов в значения по умолчанию.

function ParamByName(**const** ParamName:**string**): TFIBXSQLVAR;

Этот метод возвращает параметр по его имени. Если параметра с таким именем в коллекции параметров запроса отсутствует, то генерируется ошибка.

function FindParam (**const** aParamName: **string**): TFIBXSQLVAR;

Этот метод ищет параметр по имени (в том числе и в макросах), Если параметра с таким именем в коллекции параметров запроса отсутствует, то возвращается значение **nil**.

function ParamCount:integer;

Этот метод возвращает количество параметров.

function ParamName(ParamIndex:integer):**string**;

Этот метод возвращает параметр по индексу.

function ParamExist(**const** ParamName:**string**; **var** ParamIndex:integer):boolean;

Этот метод проверяет существование параметра с именем ParamName.

Подробнее см. тему «Заполнение параметров» в руководстве пользователя.

function FieldValue(**const** FieldName:**string**;Old:boolean):variant; **overload**;

function FieldValue(**const** FieldIndex:integer;Old:boolean):variant; **overload**;

function ParamValue(**const** ParamName:**string**):variant; **overload**;

function ParamValue(**const** ParamIndex:integer):variant; **overload**;

Этот метод возвращает значения полей и параметров

function ReadySQLText (ForChangeExecSQL:boolean=True): **string**;

Этот метод возвращает реальный текст SQL, который уходит на сервер при использовании параметров и макросов.

procedure BeginModifySQLText;

procedure EndModifySQLText;

function CountModifySQLText:integer;

Следует отметить, что если включено свойство ParamCheck, то как только разработчик меняет текст SQL, то FIBPlus немедленно пытается отпарсить текст, для того чтоб создать список параметров. Вышеописанные методы позволяют отложить процесс парсинга если текст SQL сформировывается не одноразовой операцией, а несколькими. Например:

begin

pFIBQuery1.BeginModifySQLText;

try

pFIBQuery1.SQL.Clear;

pFIBQuery1.SQL.Add('Insert into Table1 (ID,Name)');

pFIBQuery1.SQL.Add(' Values (:ID, :NAME)');

finally

```

    pFIBQuery1.EndModifySQLText;
end
end;
```

В этом примере парсинг запроса произойдет один раз, после вызова `pFIBQuery1.EndModifySQLText`. Если бы мы не вызывали бы метод `BeginModifySQLText` перед изменением текста, то парсинг происходил бы после каждой операции.

```
procedure AssignProperties(Source: TFIBQuery);
```

Этот метод копирует все свойства компонента-параметра

NEW

```
procedure ExecuteImmediate;
```

Запускает выполнение запроса без предварительной препарации. Все ресурсы необходимые для выполнения этого запроса будут созданы самим сервером и им же освобождены. Этим методом можно выполнять только запросы не имеющие параметров.

```
procedure ExecProcedure(const ProcName:string);
```

Выполняет сохраненную процедуру по имени `ProcName` без параметров. Т.е. автоматически формируется текст запроса вида :

```
'Execute Procedure '+ ProcName
```

и он запускается на выполнение.

```
procedure ExecProcedure(const ProcName:string;const InputParams:array of variant);
```

Выполняет сохраненную процедуру по имени `ProcName` с параметрами `InputParams`.

```
procedure ExecuteAsBatch;
```

```
procedure ExecuteAsBatch(const SQLs:array of string);
```

Выполняет пакет запросов. Работает только под IB2007 вызывая API функцию `isc_dsqli_batch_execute_immed`

Public свойства

```
property Bof: Boolean read FBOF;
```

Это свойство установлено в True, если достигнуто начало набора данных.

```
property DBHandle: PISC_DB_HANDLE read GetDBHandle;
```

Это свойство возвращает Handle запроса.

```
property Eof: Boolean read GetEOF;
```

Это свойство установлено в True, если достигнут конец выборки

```
property FldByName[const FieldName: string]: TFIBXSQLVAR read FieldByName;
```

Это свойство возвращает значение поля по имени.

property Fields[**const** Idx: Integer]: TFIBXSQLVAR **read** GetFields;

Это свойство возвращает значение поля по индексу.

property FieldIndex[**const** FieldName: **string**]: Integer **read** GetFieldIndex;

Это свойство возвращает индекс поля по имени.

property Open: Boolean **read** Fopen;

Это свойство показывает, активен ли запрос.

property Params: TFIBXSQLDA **read** GetSQLParams;

Это свойство возвращает массив параметров запроса.

property OnlySrvParams: TStringList;

Это свойство возвращает список имен параметров запроса. В список попадают только те параметры которые реально будут использоваться параметры отправляемого запроса. Например для «Select * from @@Table@» свойство вернет пустой список, поскольку макрос не является параметром конечного запроса..

property Plan: **string** **read** GetPlan;

Это свойство возвращает план выполнения запроса.

property Prepared: Boolean **read** Fprepared;

Это свойство установлено в True, если запрос подготовлен на сервере.

property RecordCount: Integer **read** GetRecordCount;

Это свойство возвращает количество отфетченных записей.

property RowsAffected: Integer **read** GetRowsAffected;

Это свойство возвращает количество записей, измененных запросом.

property AllRowsAffected: TAllRowsAffected **read** GetAllRowsAffected;

Это свойство возвращает количество операций, произведенных запросом.

```
TAllRowsAffected =
record
  Updates: integer;
  Deletes: integer;
  Selects: integer;
  Inserts: integer;
end;
```

property SQLType: TFIBSQLTypes **read** FSQLType;

TFIBSQLTypes = (SQLUnknown, SQLSelect, SQLInsert, SQLUpdate, SQLDelete, SQLDDL, SQLGetSegment, SQLPutSegment, SQLExecProcedure, SQLStartTransaction, SQLCommit, SQLRollback, SQLSelectForUpdate, SQLSetGenerator, SQLSavePointOperation)

property TRHandle: PISC_TR_HANDLE **read** GetTRHandle;

Это свойство возвращает Handle – внутренний дескриптор транзакции на сервере, который идентифицирует каждую операцию с БД, сообщая, в рамках какой транзакции приходят запросы. Подробности смотрите в APIGuide.pdf

property ProcExecuted: boolean **read** FProcExecuted **write** FprocExecuted;

Это свойство установлено в True, если процедура в запросе выполнена через execute procedure.

```
property OnSQLFetch: TOnSQLFetch read FOnSQLFetch write FonSQLFetch;
TOnSQLFetch = procedure (RecordNumber:integer; var StopFetching:boolean ) of
object;
```

Это свойство генерируется при fetch очередной записи.

```
property CursorName : string read FCursorName write FcursorName;
```

Это свойство возвращает имя курсора. Необходимо только для запросов типа Select for update.

```
property OrderClause: string read GetOrderString write SetOrderString;
```

Это свойство возвращает секцию order by

```
property GroupByClause: string read GetGroupByString write SetGroupByString;
```

Это свойство возвращает секцию group by

```
property FieldsClause: string read GetFieldsClause write SetFieldsClause;
```

Это свойство возвращает секцию полей

```
property PlanClause: string read GetPlanClause write SetPlanClause;
```

Это свойство позволяет задать план выполнения запроса.

```
property SQLKind: TSQLKind read GetSQLKind;
```

```
TSQLKind = (skUnknown, skSelect, skUpdate, skInsert, skDelete, skExecuteProc, skDDL,
skExecuteBlock);
```

Еще одно свойство возвращающее тип запроса. В отличие от SQLType, оно доступно сразу же при присвоении текста SQL.

TrFIBStoredProc

Это наследник TrFIBQuery. Отличается свойством StoredProcName, которое позволяет задать имя хранимой процедуры.

TFIBXSQLDA

Класс управляющий списком параметров или полей запроса.

Public свойства

```
property Names: string read GetNames;
```

Возвращает строку – список имен параметров.

```
property Count: Integer;
```

Возвращает количество параметров.

```
property Modified: Boolean;
```

Возвращает True – если параметры были модифицированы после последнего выполнения запроса.

Public методы

```
procedure ClearValues;
```

Этот метод очищает значения всех параметров в списке.

procedure AssignValues (SourceSQLDA:TFIBXSQLDA);

Этот метод, по принципу совпадения имен, присваивает параметрам из списка значения параметров SourceSQLDA.

TFIBXSQLVAR

Класс представляющий параметр или поле запроса.

Public свойства

property Name: **string** read GetNames;

Возвращает строку – имя параметра или поля.

property IsMacro: **boolean**;

Возвращает True, если объект представляет макрос.

property DefMacroValue : **string**;

Если объект представляет макрос, то свойство возвращает строку-значение по умолчанию для него.

property InWhereClause: **boolean**;

Возвращает True, если объект представляет параметр находящийся в where условии.

property BeginPosInText: **integer**;

Возвращает начальную позицию параметра в тексте SQL.

property EndPosInText: **integer**;

Возвращает конечную позицию параметра в тексте SQL.

property Modified: **Boolean**;

Возвращает True – если параметр был модифицирован после последнего выполнения запроса.

property IsNull: **Boolean**;

Возвращает True, если параметр или поле содержит Null значение.

property IsNullable: **Boolean**;

Возвращает True, если параметр или поле может содержать Null значение.

property SQLType: **Integer**;

Возвращает тип поля или параметра.

property ServerSQLType: **Integer**;

Возвращает предусмотренный тип поля или параметра. SQLType и ServerSQLType всегда совпадают для полей и могут различаться для параметров.

property SQLSubtype: **Integer**;

Возвращает подтип поля или параметра.

property ServerSubtype: **Integer**;

Возвращает предусмотренный подтип поля или параметра. SQLSubType и ServerSQLSubType – всегда совпадают для полей и могут различаться для параметров.

property Size: **Integer**;

Возвращает размер значения параметра в байтах.

property ServerSizeSize: **Integer**;

Возвращает размер значения параметра в байтах. Size и ServerSizeSize – всегда

совпадают для полей и могут различаться для параметров.

property Value: Variant **read write**

Содержит значение параметра или поля.

property OldValue: Variant **read**

Возвращает значение параметра до последней модификации.

```

property AsExtended: Extended read write
property AsInt64: Int64 read write
property AsBcd: TBcd read write
property AsGuid: TGUID read write
property AsDateTime: TDateTime read write
property AsDate: TDateTime read write
property AsTime: TDateTime read write
property AsTimeStamp: TTimeStamp read write
property AsFloat: Double read write
property AsSingle: Float read write
property AsInteger: Integer read write
property AsLong: Long read write
property AsQuad: TISC_QUAD read write
property AsShort: Short read write
property AsString: string read write
property AsWideString: WideString read write
property AsBoolean: boolean read write

```

Свойства отвечающие за чтение-запись значений параметров, и только чтения значений полей.

Public методы

function IsParam:boolean;

Возвращает True, если объект представляет параметр, и False если он представляет поле.

function IsBlob:boolean;

Возвращает True, если объект представляет блоб-поле или блоб-параметр.

procedure Clear;

Очищает значение параметра. После этого параметр принимает значение Null.

procedure Assign(Source: TFIBXSQLVAR);

Присваивает параметру значение, тип и все прочие характеристики из параметра Source.

procedure SaveToFile(const FileName: string);

procedure SaveToStream(Stream: TStream);

Методы сохраняющие значение блоб-поля или блоб-параметра в файл или Stream.

procedure LoadFromFile(const FileName: string);

procedure LoadFromStream(Stream: TStream);

Методы загружающие значение блоб-параметра из файла или Stream.

function IsDefMacroValue :boolean;

Возвращает True, если объект представляет макрос и на данный момент содержит значение по умолчанию .

procedure SetDefMacroValue;

Если объект представляет макрос, то метод присваивает ему значение по умолчанию.

TrFIBDataset

Является потомком TDataSet и поддерживает все его методы и свойства. Здесь будут перечислены только специфические свойства TrFIBDataSet.

Свойства

Active

Смотрите справку по TDataSet.

Filter

Смотрите описание свойства в справке по Delphi/C++Builder, оно ничем не отличается от свойств в TDataSet.

FilterOptions

Смотрите описание свойства с справке по Delphi/C++Builder, оно ничем не отличается от свойств в TDataSet.

AllowedUpdateKinds

Объявленные как

```
TUpdateKind = (ukModify, ukInsert, ukDelete);
TUpdateKinds = set of TUpdateKind;
```

Это свойство позволяет задать режим обновления датасета. ukModify определяет, будут ли производиться модификации датасета, ukInsert вставки и ukDelete удаления. Если опция выключена, но в момент операции библиотека будет вызвать тихое исключение Abort, то операция не будет выполняться.

AutoCalcFields

Значение опции аналогично значению опции для TDataSet.

AutoCommit

Если опция установлена в True, то, в зависимости от настроек [UpdateTransaction](#), после каждой модифицирующей операции Insert/Update/Delete будет вызываться принудительный Commit/CommitRetaining.

AutoUpdateOptions

Очень важная группа опций. Если понимать, как работает данная группа опций, можно избежать значительного количества проблем.

```
TAutoUpdateOptions= class (TPersistent)
  property AutoParamsToFields: Boolean .. default False;
  property AutoReWriteSqls: Boolean .. default False;
  property CanChangeSQLs: Boolean .. default False;
  property GeneratorName: string;
  property GeneratorStep: Integer .. default 1;
  property KeyFields: string;
  property ParamsToFieldsLinks: TStrings;
  property SeparateBlobUpdate: Boolean .. default False;
  property UpdateOnlyModifiedFields: Boolean .. default False;
  property UpdateTableName: string;
  property WhenGetGenID: TWhenGetGenID .. default wgNever;
  property UseExecuteBlock: Boolean;
  property UseReturningFields: TSetReturningFields;
end;
```

```
TWhenGetGenID= (wgNever, wgOnNewRecord, wgBeforePost) ;
```

Если опция `AutoRewriteSQLs` установлена в `True`, то, при наличии пустых `SQLText` для `InsertSQL`, `UpdateSQL`, `DeleteSQL` и `RefreshSQL`, будет автоматически производиться их генерация на основе `KeyFields`, `UpdateTableName`.

Если опция `CanChangeSQLs` установлена в `True`, то разрешена перезапись непустых `SQL`.

Опции `GeneratorName` и `GeneratorStep` задают, соответственно, имя и шаг генератора

Опция `KeyFields` содержит список ключевых полей

Опция `SeparateBlobUpdate` управляет записью `BLOB` полей в базу данных. Если эта опция установлена в `True`, то сначала будет производиться запись строки без `BLOB` поля, а затем, в случае успеха, будет записываться само `BLOB` поле.

Если опция `UpdateOnlyModifiedFields` установлена в `True` и если также установлены `CanChangeSQLs`, то для каждой операции обновления будет формироваться новый `SQL`-запрос, в котором будут представлены только реально измененные поля.

UpdateTableName должна содержать имя обновляемой таблицы

WhenGetGenId позволяет задать режим использования генератора: никогда, на новую запись, непосредственно перед операцией `Post`.

ParamsToFieldsLinks - представляет собой «карту соответствия» между полями запроса и параметрами. Если это свойство заполнено, то при вставке новой записи, автоматически указанные поля будут заполнены значениями соответствующих параметров. Особенно актуально это свойство для деталь-датасета в режиме мастер-деталь.

AutoParamsToFields - если содержит значение `True` то `ParamsToFieldsLinks` будет заполнен автоматически.

`UseExecuteBlock` : Работает только при `CachedUpdates=True`. Если включено, то при `ApplyUpdates`, `ApplyUpdToBase` будут генериться запросы типа `EXECUTE BLOCK`. Т.е. изменения будут посылаться в базу не для каждой записи отдельно, а пачками по 255 записей.

`UseReturningFields` : Указывает использовать ли для генерации `UpdateSQL`, `InsertSQL` секцию `RETURNING`, что позволяет после выполнения модифицирующего запроса частично «освежить» запись без выполнения метода `Refresh`. (работает начиная с файрберд 2.0)
Имеет три возможных значения.

`rfAll` - включать в секцию `RETURNING` все поля

`rfKeyFields` - включать в секцию `RETURNING` только ключевые поля

`rfBlobFields`- включать в секцию `RETURNING` блоб поля.

Т.е., при использовании настроек `AutoUpdateOptions` FIBPlus позволяет избавиться от генерации SQL в режиме `design time` и переложить эту задачу на время исполнения программы. Для этого нужно лишь приписать имя обновляемой таблицы и ключевое поле.

Данные код взят из примера `AutoUpdateOptions`:

```
pFIBDataSet1.SelectSQL.Text := 'SELECT * FROM EMPLOYEE';
pFIBDataSet1.AutoUpdateOptions.AutoRewriteSqls := True;
pFIBDataSet1.AutoUpdateOptions.CanChangeSQLs := True;
pFIBDataSet1.AutoUpdateOptions.UpdateOnlyModifiedFields := True;
pFIBDataSet1.AutoUpdateOptions.UpdateTableName := 'EMPLOYEE';
pFIBDataSet1.AutoUpdateOptions.KeyFields := 'EMP_NO';
pFIBDataSet1.AutoUpdateOptions.GeneratorName := 'EMP_NO_GEN';
pFIBDataSet1.AutoUpdateOptions.WhenGetGenID := wgBeforePost;
pFIBDataSet1.Open;
```

CachedUpdates

Это свойство соответствует аналогичному свойству компонента `TDataSet`, подробнее смотри руководство пользователя тему «Использование кэшированных изменений».

CacheModelOptions

Это «революционное» новшество FIBPlus. Используя эту опцию, Вы можете выбрать модель хранения кеша датасета. Подробное описание работы в режиме ограниченного кеша описано в разделе ...

```
TCacheModelOptions = class(TPersistent)
  property CacheModelKind: TCacheModelKind default cmkStandard;
  property BufferChunks: Integer default vBufferCacheSize;
  property PlanForDescSQLs: string;
  property BlobCacheLimit: integer;
end;
TCacheModelKind = (cmkStandard, cmkLimitedBufferSize);
```

`CacheModelKind` может принимать значение `cmkStandard` `cmkLimitedBuffersSize`. При установлении этой опции в `cmkLimitedBuffersSize` датасет будет загружать в память приложения столько записей, сколько установлено в свойстве `BuffersChunks`. В дополнении к этому можно указать план для выполнения обратных запросов (быстрое извлечение последних записей выборки). `BlobCacheLimit` – указывает максимальное число блобов удерживаемых в кэше.

Conditions

Подробнее о дополнительных условиях читайте в Руководстве пользователя.

Container

Подробнее об использовании контейнеров смотрите описание компонента `TpFIBDatasetContainer`.

Database

Это свойство используется для подключения датасета к базе данных.

DataSet_ID

Свойство позволяет задать код датасета, хранящегося в репозитории. Если значение свойства отлично от нуля, то при открытии датасета будет произведена выгрузка настроек из репозитория датасетов и заполнено свойство `Description`. Подробнее о использовании репозитория FIBPlus читайте в разделе «Репозитории FIBPlus» а также смотрите демонстрационные примеры `XXXRepository`

Description

Это свойство описание датасета, которое используется в репозитории датасетов.

DefaultFormats

Эта группа опций позволяет задать форматы отображения стандартных полей для этого датасета. Описание выглядит следующим образом:

```
TFormatFields = class(TPersistent)
property DateTimeDisplayFormat: string;
property NumericDisplayFormat: string;
property NumericEditFormat: string;
property DisplayFormatDate: string;
property DisplayFormatTime: string;
end;
```

Т.е., вы можете задать формат отображения для полей TDateTimeField/TDataField/TTimeField, отображения и редактирования TnumericField.

Подробнее см. в руководство пользователя

DetailConditions

Это группа опций облегчает работу в режиме master-detail. Вот ее описание:

```
TDetailCondition=(dcForceOpen, dcIgnoreMasterClose, dcForceMasterRefresh,
dcWaitEndMasterScroll);
```

```
TDetailConditions= set of TDetailCondition;
```

dcForceOpen	если эта опция включена, то детальный датасет будет открываться автоматически при открытии мастера
dcIgnoreMasterClose	опция означает, что детальный датасет не будет закрываться в случае закрытия мастера
dcForceMasterRefresh	при обновлении детального датасета будет производиться обновление мастер-датасета – будет вызываться его RefreshSQL
dcWaitEndMasterScroll	опция означает, что при прокрутке мастера выжидается некоторое время и только потом происходит переоткрытие детали. Опция позволяет избежать лишней работы при прокрутке мастер-датасета

FieldOriginalRule

Это свойство позволяет руководить заполнением свойства TField.Origin и может принимать несколько значений:

```
TfieldOriginRule =(forNoRule, forTableAndFieldName, forClientFieldName,
forTableAliasAndFieldName);
```

Значениями, соответственно, являются: не заполнять, для таблицы и имени поля, просто имя поля на клиенте, для алиаса таблицы и имени поля. Свойство **Origin** заполняется только в режиме выполнения программы.

Options

Эта группа опций - одна из основных и необходимых для понимания тонкостей работы с TpFIBDataSet. Описание группы опций:

```
TpFIBDsOption = (poTrimCharFields, poRefreshAfterPost, poRefreshDeletedRecord,
poStartTransaction, poAutoFormatFields, poProtectedEdit, poKeepSorting,
poPersistentSorting, poVisibleRecno, poNoForceIsNull, poFetchAll,
poFreeHandlesAfterClose, poCacheCalcFields);
```

```
TpFIBDsOptions= set of TpFIBDsOption;
```

poStartTransaction	стартовать транзакцию при открытии датасета, если она не активна
--------------------	--

poTrimCharFields	усекать концевые пробелы для полей типа CHAR/VARCHAR
poRefreshAfterPost	выполнять RefreshSQL, после фиксации изменений в БД, после метода Post;
poRefreshDeletedRecord	удалять из кеша запись после выполнения RefreshSQL, если тот не вернул записи
poAutoFormatFields	использовать автоматическое форматирование для датасета
poProtectedEdit	использовать защищенное редактирование (подробнее описано в разделе Защищенное редактирование)
poKeepSorting	помещать добавленные либо измененные записи в правильную позицию буфера, которая отвечает локальной сортировке датасета. Подробнее работа с локальной сортировкой будет рассмотрена в разделе Локальная сортировка
poPersistentSorting	сохранять сортировку и восстанавливать при следующем открытии датасета.
poVisibleRecno	при включенной опции добавляется поле RecNo
poNoForceIsNull	если опция выключена, то для параметров в случае если они NULL условие where вида FIELD1 = :FIELD1 будет заменяться на where FIELD IS NULL. В некоторых случаях такое поведение может быть нежелательно. Например, если нужно вернуть информацию о типе параметра, следует включить эту опцию.
poFetchAll	при включенной опции будет производиться полный fetch данных, что может потребоваться, например, для справочников
poFreeHandlesAfterClose	эта опция отвечает за то, чтобы автоматически сразу же после закрытия запроса (модифицирующий запрос, либо селективный после fetch всех записей) вызывалось автоматическое освобождения ресурсов, связанных с запросом FreeHandle
poCacheCalcFields	при включенной опции будут кэшироваться вычисляемые и лукап поля.
poUseSelectForLock	при включенной опции, для создания локирующего запроса будет применяться синтаксис SELECT 1 FROM TABLE1 FOR UPDATE WITH LOCK

PrepareOptions

Это также ключевые опции для тонкой настройки работы датасета

```

TpPrepareOption = (pfSetRequiredFields, pfSetReadOnlyFields,
pfImportDefaultValues, psUseBooleanField, psUseGuidField, psSQLINT64ToBCD,
psApplyRepository, psGetOrderInfo, psAskRecordCount, psCanEditComputedFields,
psSetEmptyStrToNull, psSupportUnicodeBlobs, psUseLargeIntField);
TpPrepareOptions=set of TpPrepareOption;

```

pfSetRequiredFields - если включено то в Fields датасета будет автоматически заполняться свойство Required. Для NOT NULL полей оно будет принимать значение True, для остальных False. (Эта опция не вызывает дополнительные запросы к базе)

pfSetReadOnlyFields - Если включено, то те поля датасета, которые не участвуют в модифицирующих запросах, автоматически становятся ReadOnly. Т.е. их нельзя будет изменять даже в буфере датасета. Кроме того становятся ReadOnly те поля, которые являются калькулируемыми на сервере. (Эта опция вызывает дополнительные запросы к базе, с целью выяснить какие поля являются server-calculated)

pfImportDefaultValues - Если включено, то те поля датасета, которые в базе имеют значение по умолчанию, автоматически получают соответствующее значение в свойство DefaultExpression. Это свойство используется при Insert/Append новой

записи. (Эта опция вызывает дополнительные запросы к базе, с целью получения текста значения по умолчанию)

pfImportDefaultValues - Если включено, то те поля датасета, которые в базе имеют значение по умолчанию, автоматически получают соответствующее значение в свойство DefaultExpression. Это свойство используется при Insert/Append новой записи. (Эта опция вызывает дополнительные запросы к базе, с целью получения текста значения по умолчанию)

psUseBooleanField - Если включено, то датасет сможет использовать Boolean поля. Подробнее см. в теме «Использование уникальных типов полей». (Эта опция вызывает дополнительные запросы к базе, с целью получения домена поля)

psUseGuidField - Если включено, то датасет сможет использовать GUID поля. Подробнее см. в теме «Использование уникальных типов полей» (Эта опция вызывает дополнительные запросы к базе, с целью получения домена поля)

psSQLINT64ToBCD - Если включено, то датасет будет использовать TBCDField для полей типа NUMERIC(x,y), где y больше 4. Подробнее см. в теме «Использование уникальных типов полей» (Эта опция не вызывает дополнительные запросы к базе)

psApplyRepository - Если включено, то датасет будет использовать репозиторий полей, для заполнения таких свойств полей, как DisplayLabel, DisplayFormat, EditFormat и т.д. Подробнее см в теме «Репозитории FIBPlus»

psGetOrderInfo - Если включено, то при открытии датасета автоматически заполняется свойство датасета SortFields, которое в дальнейшем используется в режимах ограниченного кэша и режимах «удерживания» сортировки при операциях вставки модификации записей. Подробнее см. в теме «Локальная сортировка». (Эта опция не вызывает дополнительные запросы к базе)

psAskRecordCount - Если включено, то перед открытием датасета будет послан запрос к серверу, с целью выяснить количество записей, которые попадают под условия запроса. После этого свойство RecordCount будет возвращать не количество отфетенных записей, а количество попавших под условия запроса.

psSetEmptyStrToNull - Если включено, то если строковое поле содержит пустую строку, и эта запись модифицирована, то при применении изменений к базе, для этого поля будет отправлено Null значение. Необходимость этой опции связана с тем, что стандартные DBControls не могут показать, или принять Null значение. Оно в них отображается именно как пустая строка.

psUseLargeIntField - Если включено, то поля типа BIGINT, NUMERIC(18,0) будут представлены классом TFIBLargeField. Если выключено, то они будут представлены классом TFIBBCDField.

Также для PrepareOptions и Options есть редакторы, облегчающие комплексную настройку датасета. Помимо этого, в инструменте FIBPlusTools эти опции можно настроить для компонентов TrFIBDataSet, вновь добавляемых в проект.

RefreshTransactionKind

Это свойство позволяет задать, в какой транзакции (читающей или обновляющей) будет вызваться RefreshSQL. Подробнее см. в теме о режиме разделенных транзакций.

SQLs

Это свойство содержит тексты SQL-запросов.

SQLScreenCursor

Это свойство позволяет задать курсор экрана для долгих операций датасета.

Transaction

Это свойство возвращает транзакцию, в которой производится чтение данных.

UniDirectional

Если это свойство установлено в True, ваш датасет станет однонаправленным и не будет кэшировать результаты выполнения. В каждый момент будет доступна лишь одна запись. Это может понадобиться, например, для больших отчетов.

UpdateRecordTypes

Это свойство позволяет задать режим отображения записей в режиме кэшированные обновлений:

```
TCachedUpdateStatus = (cusUnmodified, cusModified, cusInserted, cusDeleted,
  cusUninserted, cusDeletedApplied);
TFIBUpdateRecordTypes = set of TCachedUpdateStatus;
```

Поддерживаются следующие режимы отображения записей: отображение немодифицировавшихся; модифицировавшихся; вставленных; удаленных; удаленных, с уже примененным удалением.

UpdateTransaction

Это свойство задает транзакцию, в которой будут производиться модифицирующие запросы

События

Рассмотрим события, уникальные для TрFIBDataSet. События TDataSet смотрите в справке по Delphi. Многие события понятны без описания, мы же обратим внимание на некоторые из них.

AfterEndTransaction

```
procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force:
Boolean);
```

AfterEndUpdateTransaction

```
procedure TForm1.pFIBDataSet1AfterEndUpdateTransaction(
  EndingTR: TFIBTransaction; Action: TTransactionAction; Force: Boolean);
```

AfterFetchRecord

```
procedure (FromQuery: TFIBQuery; RecordNumber: Integer; var StopFetching:
Boolean);
```

AfterStartTransaction

```
procedure (Sender: Tobject);
```

AfterStartUpdateTransaction

```
procedure TForm1.pFIBDataSet1AfterStartUpdateTransaction(Sender: Tobject);
```

BeforeEndTransaction

```
procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force:
Boolean);
```

BeforeEndUpdateTransaction

```
procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force:
Boolean);
```

BeforeFetchRecord

```
procedure (EndingTR: TFIBTransaction; Action: TTransactionAction; Force:
Boolean);
```

BeforeStartTransaction

```
procedure (Sender: Tobject);
```

BeforeStartUpdateTransaction

```
procedure (Sender: Tobject);
```

DatabaseDisconnected

```
procedure (Sender: TObject);
```

DatabaseDisconnecting

```
procedure (Sender: TObject);
```

DatabaseFree

```
procedure (Sender: TObject);
```

OnApplyDefaultValue

```
procedure (DataSet: TDataSet; Field: TField; var Applied: Boolean);
```

Это событие позволяет переопределить значения полей, выставленных по умолчанию, при вставке новой записи.

OnApplyFieldRepository

Это событие позволяет разработчику легко использовать свои собственные настройки в репозитории полей. Например. Если вам хочется настраивать свойство EditMask, то добавляете в таблицу репозитория поле EDIT_MASK. А в обработчике OnApplyFieldRepository пишете:

```
procedure TForm1.pFIBDataSet1ApplyFieldRepository(DataSet: TDataSet;
  Field: TField; FieldInfo: TpfIBFieldInfo);
begin
  Field.EditMask:=FieldInfo.OtherInfo.Values['EDIT_MASK'];
end;
```

OnAskRecordCount

```
procedure (DataSet: TFIBDataSet; var SQLText: String);
```

Это событие возникает при выставленной опции psAskRecordCount и позволяет изменить SQL по умолчанию для получения количества записей

OnCompareFieldValues

```
function (Field: TField; const S1, S2: Variant): Integer;
```

Это событие позволяет задать свою функцию сравнения для локальной сортировки. Различные кодировки (разные наборы символов) требуют разных методов сортировки. Поэтому для кодировки None и Unicode_FSS подойдет стандартный метод, т.е., ничего переопределять не нужно. Для национальных кодировок нужно ставить ANSI кодировку AnsiCompareString (функция датасета).

OnFieldChange

```
procedure OnFieldChange(Sender: Tfield);
```

Это событие возникает при изменении значения поля.

OnFillClientBlob

```
procedure (DataSet: TFIBCustomDataSet;
  Field: TFIBBlobField; Stream: TFIBBlobStream);
```

Это событие вызывается при заполнении BLOB поля на клиенте, в случае если свойство TFIBBlobField.IsClientField=True

OnReadBlobField, OnWriteBlobField

```
procedure (Field: TBlobField; BlobSize: integer; Progress: integer; var Stop: boolean);
```

Эти события вызываются при чтении-записи BLOB поля из БД. Позволяют отобразить

прогресс процесса чтения-записи, а при чтении и прервать его в случае необходимости

OnLockError

```
procedure (DataSet: TDataSet; LockError: TLockStatus; var ErrorMessage: String;  
  var Action: TDataAction);
```

Это событие возникает при выставленной опции `roProtectedEdit`, если попытка блокирования не была успешной. Смотрите раздел «Пессимистическая блокировка» руководства пользователя.

OnLockSQLText

```
procedure (DataSet: TFIBDataSet; var SQLText: String);
```

Это событие возникает перед генерацией лолирующего запроса, и позволяет разработчику самому задать текст этого запроса, не полагаясь на автогенерацию.

TransactionEnded

```
procedure (Sender: Tobject);
```

Это событие возникает после завершения транзакции.

TransactionEnding

```
procedure (Sender: Tobject);
```

Это событие возникает в момент завершения транзакции.

TransactionFree

```
procedure (Sender: Tobject);
```

Это событие возникает при высвобождении транзакции.

Методы

Данные методы используются для сравнения Ansi-строк:

```
function AnsiCompareString(Field:TField;const val1, val2: variant): Integer;  
function StdAnsiCompareString(Field:TField;const S1, S2: variant): Integer;
```

`StdAnsiCompareString` – задает следующий порядок:

```
a  
A  
б  
Б
```

`AnsiCompareString` – задает следующий порядок:

```
A  
Б  
a  
б
```

Метод соответствует разным сортировкам сервера при различных сочетаниях чарсетов (`charset`) и коллэитов (`collate`).

```
function FN(const FieldName: string): TField;  
function FBN(const FieldName: string): Tfield;
```

Данные функции аналогичны `TdataSet.FieldName`, но более короткие в написании.

```
procedure ApplyConditions(Reopen :boolean = False);
```

Этот метод применяет дополнительные условия. Параметр `Reopen` указывает, нужно ли переоткрывать `TrFIBDataSet`

procedure CancelConditions;

Этот метод отменяет все дополнительные условия.

procedure CloseOpen(**const** DoFetchAll:boolean);

Этот метод переоткрывает TрFIBDataSet. Параметр DoFetchAll указывает, делать ли полный fetch данных

procedure StartTransaction;

procedure BatchInput(InputObject: TFIBBatchInputStream; SQLKind: TpSQLKind =skInsert);

procedure BatchOutput(OutputObject: TFIBBatchOutputStream);

Подробности смотрите в разделе [Пакетные изменения](#) Руководства пользователя.

function CachedUpdateStatus: TcachedUpdateStatus;

Этот метод позволяет получить статус записи при использовании кэшированных обновлений.

procedure CancelUpdates; **virtual**;

Этот метод отменяет все изменения, сделанные в режиме кэшированных обновлений

procedure FetchAll;

Этот метод производит полный fetch датасета

procedure RevertRecord;

При использовании режима CachedUpdates этот метод возвращает запись к начальному состоянию.

procedure Undelete;

Этот метод отменяет удаление записи в режиме CachedUpdates.

procedure DisableScrollEvents;

procedure EnableScrollEvents;

procedure DisableCloseOpenEvents;

procedure EnableCloseOpenEvents;

Блокируют и деблокируют выполнение соответствующих событий.

procedure DisableCalcFields;

procedure EnableCalcFields;

Блокируют и деблокируют выполнение события OnCalcFields.

procedure DisableMasterSource;

procedure EnableMasterSource;

function MasterSourceDisabled:boolean;

Служат для временного отключения режима мастер-деталь. Например

```
DetailDataSet.DisableMasterSource;
try
  MasterDataSet.Edit;
  MasterDataSet.FieldName('ID').asInteger:=NewValue;
  MasterDataSet.Post;
  ChangeDetailDataSetLinkField(NewValue);
finally
  DetailDataSet.EnableMasterSource;
```

end

при этом деталь датасет не переоткрывается.

```
function ArrayFieldValue(Field:TField):Variant;
procedure SetArrayValue(Field:TField;Value:Variant);
function GetElementFromValue( Field:TField; Indexes:array of integer):Variant;
procedure SetArrayElementValue(Field:TField;Value:Variant; Indexes:array of
integer );
```

Для получения подробностей смотрите раздел «Работа с полями-массивами» Руководства пользователя.

```
function GetRelationTableName(Field:TObject):string;
```

Этот метод возвращает имя отношения для поля.

```
function GetRelationFieldName(Field:TObject):string;
```

Этот метод возвращает имя поля.

```
procedure MoveRecord(OldRecno,NewRecno:integer); virtual;
```

Этот метод перемещает поле в кеше с позиции OldRecNo на позицию NewRecNo

```
procedure DoSortEx(Fields: array of integer; Ordering: array of Boolean);
overload;
```

```
procedure DoSortEx(Fields: TStrings; Ordering: array of Boolean); overload;
```

```
procedure DoSort(Fields: array of const; Ordering: array of Boolean); virtual;
```

Данные методы – это методы локальной сортировки.

```
function CreateCalcField(FieldClass:TFieldClass; const
```

```
aName,aFieldName:string;aSize:integer):TField;
```

```
function CreateLookupField(FieldClass:TFieldClass; const
```

```
aName,aFieldName:string;aSize:integer; aLookupDataSet: TDataSet; const
```

```
aKeyFields, aLookupKeyFields, aLookupResultField: string ):TField;
```

Этот метод позволяет создавать в ран-тайме Calc-, Lookup-поля.

```
function GetFieldOrigin(Fld:TField):string;
```

Этот метод возвращает оригинальное имя поля Tfield.Origin

```
function FieldByOrigin(const aOrigin:string):TField; overload;
```

```
function FieldByOrigin(const TableName,FieldName:string):TField; overload;
```

Этот метод получает объект-поле по оригинальному имени

```
function FieldByRelName(const Fname:string):TField;
```

Этот метод возвращает первое поле для 'Select AAA as Name from Table1'

FieldByRelName('AAA').

```
function ReadySelectText:string;
```

Этот метод позволяет увидеть запрос, который в действительности отправляется на сервер. Полезен при работе с дополнительными условиями.

```
function TableAliasForField(const aFieldName:string):string;
```

Этот метод возвращает псевдоним таблицы для поля.

```
function SQLFieldName(const aFieldName:string):string;
```

Этот метод возвращает реальное имя поля.

```
procedure RestoreMacroDefaultValues;
```

Этот метод устанавливает значения макросов в значения по умолчанию (default).

function IsComputedField(Fld:Variant):boolean;

Этот метод возвращает True, если поле вычисляемое.

function DomainForField(Fld:Variant):string;

Этот метод возвращает домен поля.

function SortInfoIsValid:boolean;

Этот метод проверяет информация о сортировке на валидность

function IsSortedField(Field:TField; var FieldSortOrder:TSortFieldInfo):boolean;

Этот метод получает информацию о порядке сортировке для поля. Тип TSortFieldInfo объявлен как:

```
TSortFieldInfo = record
  FieldName: string;           //имя поля
  InDataSetIndex: Integer;    // порядковый номер поля в датасете, т.е., индекс в
                              // коллекции Fields.
  InOrderIndex: Integer;      // порядковый номер поля в order. Т.е., например,
                              // для order by 2,3,1 второе поле датасета будет
                              // первым в ордере. InDataSetIndex=1, а
                              // InOrderIndex=0 (нумерация с нуля)
  Asc: Boolean;               //True, если порядок полей по возрастанию
  NullsFirst: Boolean;       //если установлено в Null значение перед
                              //остальными
end;
```

function SortFieldsCount:integer;

Этот метод возвращает количество полей сортировки

function SortFieldInfo(OrderIndex:integer):TSortFieldInfo;

Этот метод возвращает информацию о сортировке поля в позиции OrderIndex

function SortedFields:string;

Этот метод возвращает строку с полями сортировки, перечисленными через '!':

function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer; **override;**

Этот метод сравнивает две закладки.

function BlobModified(Field: TField): boolean;

Этот метод возвращает True, если BLOB поле Field было модифицировано.

function CreateBlobStream(Field: TField; Mode: TBlobStreamMode): TStream;
override;

Этот метод создает BLOB Stream для поля.

function GetRecordFieldInfo(Field: TField; var TableName,FieldName:string; var
RecordKeyValues:TDynArray):boolean;

Этот метод получает информацию о поле.

function RecordFieldValue(Field:TField;RecNumber:integer):Variant; **overload;**

function RecordFieldValue(Field:TField;aBookmark:TBookmark):Variant; **overload;**

Эти методы возвращают значение поля и значение поля по закладке соответственно.

```

function Locate(const KeyFields: String; const KeyValues: Variant; Options:
TLocateOptions): Boolean; override;
function LocatePrior(const KeyFields: String; const KeyValues: Variant; Options:
TLocateOptions): Boolean;
function LocateNext(const KeyFields: String; const KeyValues: Variant; Options:
TLocateOptions): Boolean;

```

Это стандартная функция датасета: поиск записи, в дополнение к нему поиск следующей записи и поиск предыдущей.

```

function ExtLocate(const KeyFields: String; const KeyValues: Variant; Options:
TExtLocateOptions): Boolean;
function ExtLocateNext(const KeyFields: String; const KeyValues: Variant;
Options: TExtLocateOptions): Boolean;
function ExtLocatePrior(const KeyFields: String; const KeyValues: Variant;
Options: TExtLocateOptions): Boolean;

```

Это уникальные функции FIBPlus, которые работают аналогично предыдущим, но позволяют более гибкое управление.

TExtLocateOptions = (eloCaseInsensitive, eloPartialKey, eloWildCards, eloInSortedDS, eloNearest, eloInFetchedRecords)

eloCaseInsensitive	игнорировать регистр при сравнении строк ;
eloPartialKey	поиск по частичному совпадению;
eloWildCards	поиск будет производиться по маске (подобно тому, как в операторе LIKE);
eloInSortedDS	поиск производится в отсортированном датасете. Если датасет сортирован по этому полю, то поиск будет работать быстрее, чем обычно;
eloNearest	(только в комбинации с eloInSortedDS). В результате операции, если запись не будет найдена, указатель текущей записи будет спозиционирован на то место, где должен был быть результат;
eloInFetchedRecords	поиск производится только в тех записях, которые уже fetched и находятся в буфере датасета.

```

procedure RefreshFilters;

```

Этот метод обновляет фильтр.

При обычной работе с фильтрами нужно делать так:

```

DataSet.Filtered := False;
DataSet.Filter := <строка фильтра>;
DataSet.Filtered := True;

```

Используя FIBPlus, можно написать проще:

```

DataSet.Filter := <строка фильтра>;
DataSet.RefreshFilters;

```

```

procedure CacheDelete;

```

Этот метод удаляет запись из кэша датасета, но реального удаления не происходит.

```

procedure CacheOpen;

```

Этот метод открывает датасет, но не делает fetch данных. Необходимо, чтобы было подключение к БД.

```

procedure RefreshClientFields (ForceCalc:boolean=True);

```

Этот метод пересчитывает Calculated поля без переоткрытия запроса.

```
function CreateCalcFieldAs (Field:TField) :TField;
```

Этот метод создает вычисляемое поле с таким же типом, как у поля-параметра.

```
procedure CopyFieldsStructure (Source:TFIBCustomDataSet;RecreateFields:boolean);
```

Этот метод копирует структуру полей из датасета-источника.

```
procedure CopyFieldsProperties (Source, Destination:TFIBCustomDataSet);
```

Этот метод копирует свойства полей из одного датасета в другой.

```
procedure AssignProperties (Source:TFIBCustomDataSet);
```

Этот метод копирует все свойства датасета-параметра.

```
procedure OpenAsClone (DataSet:TFIBCustomDataSet);
```

Этот метод открывает датасет как копию датасета-параметра.

```
procedure Clone (DataSet:TFIBCustomDataSet; RecreateFields:boolean);
```

Этот метод клонирует данные датасета-параметра.

```
function CanCloneFromDataSet (DataSet:TFIBCustomDataSet) :boolean;
```

Эта функция возвращает True, если может быть создана копия датасета-параметра

```
function PrimaryKeyFields (const TableName: string) : string;
```

Этот метод возвращает имя ключевого поля для таблицы.

```
function FetchNext (FetchCount:Dword) :integer;
```

Этот метод делает fetch следующих записей в количестве, указанном в параметре.

```
procedure ReopenLocate (const LocateFieldNames:string);
```

Этот метод перекрывает TрFIBDataSet с позиционированием на том же месте, где был курсор перед закрытием. Параметр определяет, по каким полям будет сделан последующий Locate. Если полей несколько, их нужно писать через ','

```
function AllFieldValues: variant;
```

Этот метод возвращает вариантный массив – текущую строку датасета.

```
procedure FullRefresh;
```

Этот метод производит перекрытие датасета. При этом отключаются методы, влияющие на отображение: визуальные компоненты данных и прокрутка.

```
function FieldsCount:integer;
```

Этот метод возвращает количество полей.

```
function FieldName (FieldIndex:integer) :string;
```

Этот метод возвращает имя поля по индексу.

```
function FieldExist (const FieldName:string; var FieldIndex:integer) :boolean;
```

Этот метод проверяет существование поля в TDataSet, и в случае успеха (если поле существует) его индекс возвращается в FieldIndex.

```
function ParamExist (const ParamName:string; var ParamIndex:integer) :boolean;
```

Этот метод проверяет существование параметра в TDataSet, и в случае успеха (если параметр существует) его индекс возвращается в FieldIndex.

```
function FieldValue(const FieldName:string; Old:boolean):variant; overload;  
function FieldValue(const FieldIndex:integer;Old:boolean):variant; overload;
```

Этот метод возвращает значение поля по имени или индексу.

```
function ParamValue(const ParamName:string):variant; overload;  
function ParamValue(const ParamIndex:integer):variant; overload;
```

Этот метод возвращает значение параметра по индексу или имени

```
procedure SetParamValue(const ParamIndex:integer; aValue:Variant);
```

Этот метод устанавливает значение параметра

```
function RecordCountFromSrv: integer; dynamic;
```

Этот метод возвращает количество записей на сервере. Используется для получения реального количества записей на сервере, когда fetch выборки выполнен не до конца. Так, например, этот метод используется при опции psAskRecordCount.

```
function VisibleRecordCount: Integer;
```

Этот метод возвращает количество видимых записей, например, в сетке данных.

```
function CanEdit: Boolean; override;
```

Этот метод возвращает True, если датасет поддерживает операцию Edit;

```
function CanInsert: Boolean; override;
```

Этот метод возвращает True, если датасет поддерживает операцию Insert;

```
function CanDelete: Boolean; override;
```

Этот метод возвращает True, если датасет поддерживает операцию Delete.

```
function ExistActiveUO(KindUpdate: TUpdateKind): boolean;
```

```
function AddUpdateObject(Value: TpFIBUpdateObject): integer;
```

```
procedure RemoveUpdateObject(Value: TpFIBUpdateObject);
```

Эти методы проверяют существование дополнительного обработчика TpFIBUpdateObject, а также удаляют или добавляют дополнительный обработчик TpFIBUpdateObject.

```
function ParamByName(const ParamName: string): TFIBXSQLVAR;
```

Этот метод возвращает параметр по имени.

```
function FindParam(const ParamName: string): TFIBXSQLVAR;
```

Этот метод ищет параметр, в том числе, и на уровне макросов. Если макросы содержат параметры, то для их заполнения нужно использовать именно этот метод.

```
function RecordStatus(RecNumber: integer): TupdateStatus;
```

Этот метод возвращает статус для кэшированных обновлений.

```
procedure CloneRecord(SrcRecord: integer; IgnoreFields: array of const);
```

Этот метод копирует запись по индексу SrcRecord. Второй параметр указывает, какие поля игнорировать при клонировании (например, ключевые поля)

```
procedure CloneCurRecord(IgnoreFields: array of const);
```

Этот метод клонирует текущую запись.

```
procedure CommitUpdToCach;
procedure ApplyUpdToBase;
procedure ApplyUpdates;
```

Это методы для кэшированных обновлений в дополнение к стандартным ApplyUpdates, CancelUpdates. Стандартные операции датасета неадекватно работают при фильтрации, поэтому рекомендуем использовать эти дополнительные методы. Подробнее смотри тему в руководстве пользователя «Использование кэшированных изменений»

```
procedure SaveToStream(Stream: TStream; SeekBegin: boolean);
procedure LoadFromStream(Stream: TStream; SeekBegin: boolean);
procedure SaveToFile(const FileName: string);
procedure LoadFromFile(const FileName: string);
```

Эти методы позволяют сохранить, а затем загрузить кеш датасета в файл или в поток. Датасет должен быть подключен к БД.

```
function LockRecord(RaiseErr: boolean= True): TlockStatus;
```

Этот метод позволяет производить пессимистическое блокирование записи.

```
function FieldByFieldNo(FieldNo: Integer): Tfield;
```

Этот метод возвращает поле по числовому параметру FieldNo

```
function ParamNameCount(const aParamName: string): integer;
```

Этот метод возвращает количество уникальных имен параметров, если один и тот же параметр используется несколько раз.

```
function ParamCount: integer;
```

Этот метод возвращает количество параметров.

```
procedure ExecUpdateObjects(KindUpdate: TUpdateKind; Buff: Pointer;
aExecuteOrder: TFIBOrderExecUO);
```

Этот метод выполняет дополнительные обновляющие запросы TpFIBUpdateObject, ассоциированные с датасетом.

```
procedure OpenWP(const ParamValues: array of Variant); overload;
procedure OpenWP(const ParamNames : string; const ParamValues: array of Variant);
overload;
procedure OpenWPS(const ParamSources: array of ISQLObject);
procedure ReOpenWP(const ParamValues: array of Variant); overload;
procedure ReOpenWP(const ParamNames : string; const ParamValues: array of
Variant); overload;
procedure ReOpenWPS(const ParamSources: array of ISQLObject);
```

Это группа перегруженных методов, которые позволяют выполнить открытие и переоткрытие датасета с одновременной передачей параметров.

```
procedure BatchRecordToQuery (ToQuery: TFIBQuery);
procedure BatchAllRecordsToQuery (ToQuery: TFIBQuery);
```

Подробности смотрите в разделе "Выполнение SQL-Запросов. Пакетная обработка" Руководства пользователя.

```
procedure AutoGenerateSQLText (ForState: TDataSetState);
function GenerateSQLText (const TableName, KeyFieldNames: string; SK: TpSQLKind;
IncludeFields: TIncludeFieldsToSQL=ifsAllFields): string;
function GenerateSQLTextWA (const TableName: string; SK: TpSQLKind;
```

```

IncludeFields:TIncludeFieldsToSQL=ifsAllFields): string;
procedure GenerateUpdateBlobsSQL;
procedure GenerateSQLs;
function CanGenerateSQLs: boolean;

```

Это группа методов для автоматической генерации обновляющих SQL-запросов.

```
function KeyField: Tfield;
```

Этот метод возвращает объект ключевого поля

```
function SqlTextGenID: string;
```

Этот метод возвращает текст для получения значения генератора

```
procedure IncGenerator; virtual;
```

Этот метод увеличивает значение генератора.

```
function AllKeyFields(const TableName: string): string;
```

Этот метод возвращает имена ключевых полей.

Следующие методы используются для манипуляции с кешем датасета

```

procedure CacheModify( aFields: array of integer; Values: array of Variant;
KindModify: byte );
procedure CacheEdit(aFields: array of integer; Values: array of Variant);
procedure CacheAppend(aFields: array of integer; Values: array of Variant);
overload;
procedure CacheAppend(Value: Variant; DoRefresh: boolean = False); overload;
procedure CacheInsert(aFields: array of integer; Values: array of Variant);
overload;
procedure CacheInsert(Value: Variant; DoRefresh: boolean = False); overload;
procedure CacheRefresh(FromDataSet: TDataSet; Kind: TCachRefreshKind ; FieldMap:
Tstrings);
procedure CacheRefreshByArrMap( FromDataSet: TDataSet; Kind: TCachRefreshKind;
const SourceFields, DestFields: array of string );

```

TrFIBUpdateObject

Это объект-наследник TrFIBQuery, который позволяет выполнить дополнительные действия для TrFIBDataSet при вставке, модификации или удалении записей. Про наследуемые свойства и методы TrFIBQuery читайте в соответствующем разделе «Выполнение по TrFIBQuery» Руководства пользователя.

Свойства

Conditions

Смотрите соответствующее свойство TrFIBDataSet или TrFIBQuery в разделе "Выполнение SQL-Запросов. Условия" Руководства пользователя.

DataSet

Это свойство возвращает датасет, для которого будет работать дополнительное действие TrFIBUpdateObject.

ExecuteOrder

Это свойство возвращает порядок выполнения действия, которое может быть выполнено до основного действия или после.

KindUpdate

Это свойство возвращает тип объекта обновления: вставка, модификация, удаление.

OrderInList

Это свойство возвращает порядок в списке объектов с одинаковым KindUpdate

TDataSetContainer

Этот объект позволяет задать одинаковое поведение для родственных объектов TрFIBDataSet.

Компонент TDataSetContainer позволяет централизованно обрабатывать события от разных компонентов TрFIBDataSet, а также посылать им сообщения, при получении которых они также могут производить какие-то дополнительные действия. Кроме того, он может использоваться, чтобы задать одинаковую функцию сравнения полей для локальной сортировки

Свойства

Active

Установите это свойство в True, если компонент активен.

MasterContainer

Если свойство установлено в True, компонент может быть подчиненным другому контейнеру

ISGlobal

Если свойство установлено в True, то через этот контейнер будут проходить события всех датасетов, вне зависимости от того привязаны они к нему, или нет. Глобальный контейнер может быть лишь один на все приложение.

СОБЫТИЯ

События повторяют избранные события TрFIBDataSet, поэтому ищите описания в соответствующем разделе о TрFIBDataSet Руководства Пользователя.

OnApplyDefaultValue

```
procedure (DataSet: TDataSet; Field: TField; var Applied: Boolean);
```

Это событие вызывается при применении значения по умолчанию для поля

OnApplyFieldRepository

Это событие вызывается при применении значений репозитория к полю. Подробнее смотри в аналогичную тему по TрFIBDataSet.

OnCompareFieldValues

```
function (Field: TField; const S1, S2: Variant; var Compared: Boolean): Integer;
```

Это событие позволяет задать свою функцию сортировки для функции DoSort/DoSortEx.

Контейнер имеет смысл для всех присоединенных датасетов, и, следовательно, функция сортировки будет применена для всех датасетов.

OnDataSetError

```
procedure (DataSet: TDataSet; Event: TKindDataSetError; E: EDatabaseError; var Action: TdataAction);
```

```
TKindDataSetError = (deOnEditError, deOnPostError, deonDeleteError);
```

Это событие генерируется при ошибке и позволяет выполнить стандартное действие

редактирования (`deOnEditError`), подтверждения (`deOnPostError`) и удаления (`deonDeleteError`).

OnDataSetEvent

```
procedure (DataSet: TDataSet; Event: TkindDataSetEvent);
TKindDataSetEvent = (deBeforeOpen, deAfterOpen, deBeforeClose, deAfterClose,
deBeforeInsert, deAfterInsert, deBeforeEdit, deAfterEdit, deBeforePost, deAfterPost,
deBeforeCancel, deAfterCancel, deBeforeDelete, deAfterDelete, deBeforeScroll, deAfterScroll,
deOnNewRecord, deOnCalcFields, deBeforeRefresh, deAfterRefresh)
```

Это событие генерируется при получении события датасета

OnUserEvent

```
procedure (Sender: TObject; Receiver: TDataSet; const EventName: String; var
Info: String);
```

Это событие генерируется при получении пользовательского события.

Методы

```
procedure AddDataSet (Value:TDataSet);
```

Этот метод позволяет добавить датасет в контейнер.

```
procedure RemoveDataSet (Value:TDataSet);
```

Этот метод позволяет удалить датасет из контейнера.

```
function DataSetCount:integer;
```

Этот метод возвращает количество датасетов в контейнере

```
function DataSet (Index:integer):TDataSet;
```

Этот метод возвращает датасет по индексу

TSIBfibEventAlerter

Этот компонент позволяет получать события базы данных.

Свойства

AutoRegister

Если свойство установлено в `True`, то при создании формы события будут зарегистрированы автоматически, иначе нужно вызвать метод `Register` самостоятельно.

Database

Это свойство задает объект `TrFIBDataBase`, события которого он будет принимать.

Events

Это свойство возвращает `StringList` с событиями, которые будут обрабатываться компонентом `TSIBfibEventAlerter`.

События

Существует только одно событие `OnEventAlert`, объявленное как:

```
procedure (Sender: TObject; EventName: String; EventCount: Integer);
EventName возвращает имя произошедшего события, EventCount – количество произошедших событий.
```

Помните, что события отсылаются только при подтверждении транзакции. Это может

накладывая ограничения на способ использования данных событий.

TrFIBErrorHandler

Этот объект позволяет централизованно обрабатывать все ошибки библиотеки, что, на наш взгляд, довольно удобно.

Свойства

Options

```
TOptionErrorHandler = (oeException, oeForeignKey, oeLostConnect, oeCheck,
    oeUniqueViolation);
TOptionsErrorHandler = set of TOptionErrorHandler;
```

Это свойство отвечает за типы ошибок, которые будут попадать в обработчик

OnFIBErrorEvent

ErrorLexems

В некоторых методах компонента используется анализ текста ошибки. Этот анализ может быть корректен, только если файл с сообщениями об ошибках сервера не локализован. Для локализованных версий необходимо провести настройку компонента, используя свойство ErrorLexems.

События

```
TOnFIBErrorEvent = procedure (Sender: TObject; ErrorValue: EFIBError;
    KindIBError: TKindIBError; var DoRaise: boolean) of object;
TKindIBError = (keNoError, keException, keForeignKey, keLostConnect,
    keSecurity, keCheck, keUniqueViolation, keOther);
```

TrFIBClientDataSet

Это прямой потомок TClientDataset, который введен для корректной работы с BCD полями. Используется совместно с TrFIBDatasetProvider. Из новых методов есть только OpenWP, специфичный для TrFIBDataSet.

Методы

```
procedure Commit;
```

Этот метод позволяет совершить коммит UpdateTransaction у pFIBDataSet, который является источником данных для ClientDataset. Для работоспособности данного метода, необходимо чтоб у DatasetProvider была включена опция poAllowCommandText.

```
procedure RollBack;
```

Этот метод позволяет совершить откат UpdateTransaction у pFIBDataSet, который является источником данных для ClientDataset. Для работоспособности данного метода, необходимо чтоб у DatasetProvider была включена опция poAllowCommandText.

function TransactionIsActive:boolean;

Этот метод позволяет узнать активна ли UpdateTransaction у pFIBDataSet, который является источником данных для ClientDataset. Для работоспособности данного метода, необходимо чтоб у DataSetProvider была включена опция poAllowCommandText.

Остальные подробности читайте во встроенной справке по Delphi.

TpFIBDatasetProvider

Этот компонент осуществляет связь между датасетом и клиентским датасетом. Это прямой потомок TDataSetProvider, поэтому подробности читайте в справке по Delphi. Объект введен для корректной работы с VCD- полями.

TpFIBScripter

Этот компонент позволяет анализировать и выполнять скрипты.

Свойства

property Database :TpFIBDatabase;

Позволяет указать Database в рамках которого будет выполняться скрипт

property Transaction: TpFIBTransaction;

Позволяет указать Transaction в рамках которой будет выполняться скрипт

property Script:TStrings;

Хранит текст скрипта, который будет выполняться.

property Paused: Boolean;

Позволяет приостановить выполнение скрипта, а так же узнать не был ли он приостановлен.

property StopStatementNo:Integer;

Позволяет узнать номер стейтмента на котором был приостановлен скрипт.

property Prepared:boolean;

Позволяет узнать подготовлен ли скрипт к выполнению

property MakeConnectInScript:boolean;

Позволяет узнать существуют ли в скрипте команды CONNECT или CREATE DATABASE

Методы

procedure ExecuteScript (FromStmt:integer=1);

Выполняет скрипт текст которого загружен в свойство Script, начиная со стейтмента указанного в FromStmt.

procedure ExecuteFromFile(const FileName: string; Terminator:Char=';') ;

Выполняет скрипт, текст которого находится в файле. ExecuteFromFile не загружает весь файл в память, а считывает файл построчно.

procedure Parse(Terminator:Char=';') ;

Выполняет анализ скрипта. С него явно или неявно начинается вся работа с текстом скрипта.

Если вы запускаете скрипт на выполнение, то явного вызова метода Parse не требуется. Если же вы хотите анализировать скрипт или выборочно выполнить из него несколько стейтментов, то метод Parse придется вызвать явно.

```
function StatementsCount:integer;
```

Возвращает количество стейтментов в скрипте.

```
function GetStatement(StmtNo:integer;Text:TStrings):PStatementDesc;
```

Позволяет получить информацию о стейтменте по его номеру. Подробнее см. Руководство пользователя.

```
procedure ExecuteStatement(StmtTxt:TStrings;stmt:PStatementDesc;StmtNo:integer;
    TmpSQL:TStrings=nil;LineInFile:integer=-1
);
```

Метод позволяет выполнить стейтмент, который был предварительно получен методом GetStatement. Подробнее см. Руководство пользователя.

TFIBSQLMonitor

Этот объект позволяет осуществить мониторинг всех действий с БД, которые производит приложение, использующее FIBPlus.

Свойства

TraceFlags

Это свойство задает типы событий, которые будет отслеживать компонент. Свойство объявляется следующим образом:

```
TFIBTraceFlag = (tfQPrepare, tfQExecute, tfQFetch, tfConnect, tfTransact,
tfService, tfMisc);
TFIBTraceFlags = set of TFIBTraceFlag;
```

tfQPrepare определяет, будут ли отслеживаться операции Prepare;

tfQExecute отслеживать выполнение;

tfQFetch отслеживать фетч;

tfConnect события соединения;

tfTransact события транзакций старта, завершения транзакций;

tfService работа с сервисами;

tfMisc служебные запросы библиотеки.

События

OnSQL

Это событие будет вызываться каждый раз при выполнении операций, выставленных в TraceFlags. Свойство объявлено следующим образом:

```
TSQLEvent = procedure(EventText: String; EventTime : TDateTime) of object;
```

TFIBSQLLogger

Этот компонент предназначен для ведения статистики работы с БД, а также логирования SQL-запросов. Смотрите демонстрационный пример SQLLogger:

FIBPlusExamples\src\SQLLogger\.

СВОЙСТВА

Database

Это свойство возвращает базу данных, запросы которой отслеживает компонент.

ActiveStatistics

Включите эту опцию, чтобы отслеживать статистику БД

ActiveLogging

Включите эту опцию, чтобы вести лог БД.

ApplicationID

Это свойство возвращает строку, которая будет определять ваше приложение. Например, это свойство пригодится для идентификации приложений при наличии нескольких компонент для записи статистики в БД.

LogFileName

Это свойство возвращает имя файла, в который будет писаться лог БД.

StatisticsParams

Это свойство возвращает параметры, которые будут сохранены в статистике. Вот возможные значения параметров:

```
TFIBStatisticsParam = (fspExecuteCount, fspPrepareCount, fspSumTimeExecute, fspAvgTimeExecute, fspMaxTimeExecute, fspLastTimeExecute);
```

<code>fspExecuteCount</code>	количество выполнений определенного запроса;
<code>fspPrepareCount</code>	количество операций подготовки запроса;
<code>fspSumTimeExecute</code>	суммарное время выполнения запроса;
<code>fspAvgTimeExecute</code>	среднее время выполнения запроса;
<code>fspMaxTimeExecute</code>	максимальное время выполнения запроса;
<code>fspLastTimeExecute</code>	последнее время выполнения запроса.

LogFlags

Это свойство отвечает за то, какие типы операций будут записываться в лог. Повторяют события TFIBSQLMonitor:

```
TLogFlag = (lfQPrepare, lfQExecute, lfQFetch, lfConnect, lfTransact, lfService, lfMisc);
```

ForceSaveLog

Если включена эта опция, то события будут записываться по мере поступления.

Методы

```
procedure Clear;
```

```
procedure SaveStatisticsToFile(const FileName:string);
```

Этот метод позволяет сохранить статистику в файл БД

```
procedure SortStatisticsForPrint(const VarName:string;Ascending:boolean);
```

Этот метод позволяет сортировать статистику БД

```
function ExistStatisticsTable:boolean;
```

Этот метод проверяет, создана ли в базе таблица статистики БД

```
procedure CreateStatisticsTable;
```

Этот метод позволяет создать таблицу для хранения статистики в БД

```
procedure SaveStatisticsToDB(ForMaxExecTime:integer=0);
```

Этот метод позволяет сохранить статистику в БД

```
procedure SaveLog;
```

Этот метод позволяет сохранить лог статистики БД

TrFIBCustomService

Свойства

Handle: TISC_SVC_HANDLE

Это свойство возвращает хендл сервиса

ServiceParamBySPB

Это свойство позволяет получить параметр сервиса по имени

Active

Это логическое свойство управляет состоянием соединения с сервисом

ServerName

Это строковое свойство имя сервера, обязательное для заполнения.

Protocol

Это свойство возвращает протокол, по которому будет осуществляться взаимодействие с сервисом. Тип Tprotocol описан следующим образом:

```
TProtocol = (TCP, SPX, NamedPipe, Local)
```

Params

Это свойство возвращает параметры сервиса

LoginPrompt

Это свойство позволяет определить, следует ли выводить диалог авторизации.

LibraryName

Это свойство возвращает имя клиентской библиотеки

SQLLogger

Это свойство логирует вызовы сервиса.

События

OnAttach

Это событие TNotifyEvent, которое возникает при присоединении к сервису

OnLogin

Это событие возникает при присоединении к сервису

```
TLoginEvent = procedure (Database: TpFIBCustomService; LoginParams: TStrings) of
object;
```

Методы

Существуют два основных метода присоединения и отсоединения от сервиса.

```
procedure Attach;
procedure Detach;
```

TpFIBServerProperties

Это событие позволяет получать информацию о сервере и обслуживаемых базах данных.

Свойства

Option

TPropertyOption задает опции для получаемой информации о БД. Объявляется следующим образом:

```
TPropertyOption = (Database, License, LicenseMask, ConfigParameters, Version)
```

Если опция установлена в True, то после вызова метода Fetch будет заполнена соответствующая запись.

DatabaseInfo

Это свойство содержит информацию о базах данных и заполняется после вызова методов Fetch или FetchDatabaseInfo.

```
TDatabaseInfo = record
  NoOfAttachments: Integer; //количество соединений с сервером
  NoOfDatabases: Integer; //количество соединенных баз данных
  DbName: Variant; //имена соединенных баз данных
end;
```

LicenseInfo

Это свойство содержит информацию о лицензии сервера

```
TLicenseInfo = record
  Key: Variant; //ключ
  Id: Variant; //код
  Desc: Variant; //описание
  LicensedUsers: Integer; //пользователи
end;
```

LicenseMaskInfo

```
TLicenseMaskInfo = record
  LicenseMask: Integer; //маска
  CapabilityMask: Integer; //маска
end;
```

VersionInfo

```
TVersionInfo = record
  ServerVersion: String; //версия сервера
  ServerImplementation: string; //внутренняя информация о сборке
  ServiceVersion: Integer; //версия сервисов
end;
```

ConfigParams

```
TConfigParams = record
  ConfigFileData: TConfigFileData;
  BaseLocation: string;
  LockFileLocation: string;
  MessageFileLocation: string;
  SecurityDatabaseLocation: string;
end;
```

```
TConfigFileData = record
  ConfigFileValue:Variant;
  ConfigFileKey:Variant;
end;
```

Методы

При помощи доступных методов можно получить всю информацию либо по всем опциям сразу, либо по каждой опции отдельно:

```
procedure Fetch;
procedure FetchDatabaseInfo;
procedure FetchLicenseInfo;
procedure FetchLicenseMaskInfo;
procedure FetchConfigParams;
procedure FetchVersionInfo;
```

TrFIBSecurityService

Этот метод используется для управления пользователями сервера.

Свойства

SecurityAction

```
TSecurityAction = (ActionAddUser, ActionDeleteUser, ActionModifyUser,
ActionDisplayUser)
```

Это свойство задает такие возможные действия как: получение информации о пользователях, а также добавление, модификация и удаление пользователей.

UserName

Это свойство возвращает имя пользователей на сервере

Password

Это свойство возвращает пароль на сервере

SQLRole

Это свойство возвращает роль пользователя на сервере

FirstName, MiddleName, LastName

Это свойство возвращает необязательную общую информацию о пользователе на сервере

UserID

Это свойство возвращает код пользователя на сервере

GroupID

Это свойство возвращает код группы. В настоящее время свойство сервером не используется.

UserInfo

Это индексруемое свойство, в котором содержится информация о пользователе в позиции Index. Информация о пользователе представлена объектом TUserInfo:

```
TUserInfo = class(TObject)
public
  UserName: string;
  FirstName: string;
  MiddleName: string;
  LastName: string;
  GroupID: Integer;
```

```
UserID: Integer;
end;
```

UserInfoCount

Это свойство возвращает количество пользователей на сервере, заполняется при выполнении метода DisplayUsers

Методы

```
procedure DisplayUsers;
```

Этот метод возвращает информацию о пользователях в свойства UserInfo

```
procedure DisplayUser (UserName: string);
```

Этот метод возвращает информацию о пользователе с именем, заполненным в свойстве UserName

```
procedure AddUser;
```

Этот метод добавляет пользователя с именем, заполненным в свойстве UserName. Свойства UserName и Password должны быть предварительно заполнены для добавляемого пользователя.

```
procedure DeleteUser;
```

Этот метод удаляет пользователя с именем, заполненным в свойстве UserName

```
procedure ModifyUser;
```

Этот метод модифицирует информацию о пользователе с именем, заполненным в свойстве UserName. Аналогично методам, описанным выше, должны быть заполнены основные свойства UserName и Password.

TpFIBBackupService

Данный компонент позволяет выполнить резервирование базы данных. Он четвертый в иерархии наследования, это добавляет ему целую группу свойств и методов.

TpFIBControlService

```
procedure ServiceStart //запускает сервис
```

```
property IsServiceRunning //показывает, активен ли сервис
```

TpFIBControlAndQueryService

```
function GetNextLine : string; //получает следующую строку из выходного буфера
```

```
property Eof: boolean //если возвращает True, то достигнут конец буфера
```

TpFIBBackupRestoreService

```
property Verbose: Boolean //выводить ли лог работы сервиса
```

```
property OnTextNotify //событие возникает при получении очередной строки
буфера
```

Свойства

BackupFile

Tstrings, в который нужно поместить целевые имена файлы бэкапа.

DatabaseName

Свойство возвращает имя базы данных, для которой будет создана резервная копия

(backup).

Option

Задаёт опции процесса резервирования. Подробное описание опций можно получить в [OpGuide.pdf](#) документации по InterBase.

```

TBackupOption = (
  IgnoreChecksums,      //игнорировать контрольную сумму
  IgnoreLimbo,          //игнорировать лимбо-транзакции
  MetadataOnly,        //резервировать только метаданные
  NoGarbageCollection, //не производить сборку мусора
  OldMetadataDesc,     //совместимость со старыми версиями
  NonTransportable,    //показывает, создавать ли backup, понятный другим
                      //версиями сервера, либо только для использования тем
                      //сервером, который его сделал (если есть два сервера,
                      //например, версии 1.0 и 1.5, то, если сервером 1.0
                      //создать backup с опцией NonTransportable, то для этого
                      //backup нельзя будет сделать restore на версии 1.5)
  ConvertExtTables);   //содержимое внешних таблиц будет включено в backup, при
                      //restore внешние таблицы будут создаваться в основной
                      //базе данных

```

```
TBackupOptions = set of TBackupOption;
```

Работа с сервисом производится следующим образом:

```

//Delphi
BackupService1.Active := True;
BackupService1.Verbose := True;
BackupService1.ServiceStart;
while not BackupService1.Eof do
  Mem1.Lines.Add(BackupService1.GetNextLine);
BackupService1.Active := False;

//C++
BackupService1->Active = true;
BackupService1->Verbose = true;
BackupService1->ServiceStart();
while (!BackupService1->Eof)
  Mem1->Lines->Add(BackupService1->GetNextLine());
BackupService1->Active = false;

```