Написание UDR Firebird на Pascal

2 июля 2019 — 0.65 Beta

Спонсоры документации:
Platinum Sponsor

МОСКОВСКАЯ БИРЖА

Gold Sponsor

IBSurgeon

Написание UDR Firebird на Pascal

Над документом работали:

Денис Симонов

Редактор:

Денис Симонов

Содержание

Введение

В Firebird уже достаточно давно существует возможность расширения возможностей языка PSQL с помощью написания внешних функций — UDF (User Defined Functions). UDF можно писать практически на любом компилируемом языке программирования.

В Firebird 3.0 была введена плагинная архитектура для расширения возможностей Firebird. Одним из таких плагинов является External Engine (внешние движки). Механизм UDR (User Defined Routines — определяемые пользователем подпрограммы) добавляет слой поверх интерфейса движка FirebirdExternal. UDR имеют следующие преимущества по сравнению с UDF:

- можно писать не только функции возвращающие скалярный результат, но и хранимые процедуры (как выполняемые, так и селективные), а так же триггеры;
- улучшенный контроль входных и выходных параметров. В ряде случаев (передача по дескриптору) типы и другие свойства входных параметров вообще не контролировались, однако вы могли получить эти свойства внутри UDF. UDR предоставляют более унифицированный способ объявления входных и выходных параметров, так как это делается в случае с обычными PSQL функциями и процедурами;
- UDR доступен контекст текущего соединения или транзакции, что позволяет выполнять некоторые манипуляции с текущей базой данных в этом контексте;
- внешние процедуры и функции (UDR) можно группировать в PSQL пакетах;
- UDR могут быть написаны на любом языке программирования (необязательно компилируемые в объектные коды), для этого необходимо чтобы был написан соответствующий External Engine плагин. Например, существуют плагины для написания внешних модулей на Java или на любом из .NET языков.

> **Примечание**
>
> Текущая реализация UDR использует PSQL заглушку. Например, она используется для проверки параметров и возвращаемых значений на соответствие ограничениям. Заглушка была использована из-за негибкости для прямого вызова внутренних функций. Результаты теста по сравнению производительности UDR и UDF показывает, что UDR примерно в 2.5 раза медленнее на примере простейшей функции сложения двух аргументов. Скорость UDR приблизительно равна скорости обычной PSQL функции. Возможно в будущем этот момент будет оптимизирован. В более сложных функциях эти накладные расходы могут стать незаметными.

В данном руководстве мы расскажем как объявлять UDR, о их внутренних механизмах, возможностях и приведём примеры написания UDR на языке Pascal. Кроме того, будут затронуты некоторые аспекты использования нового объектно-ориентированного API.

Далее в различных главах этого руководства при употреблении терминов внешняя процедура, функция или триггер мы будем иметь ввиду именно UDR (а не UDF).

> **Примечание**
>
> Все наши примеры работают на Delphi 2009 и старше, а так же на Free Pascal. Все примеры могут быть скомпилированы как в Delphi, так и в Free Pascal, если это не оговорено отдельно.

Глава 1

Firebird API

Для написания внешних процедур, функций или триггеров на компилируемых языках программирования нам потребуются знания о новом объектно ориентированном API Firebird. Данное руководство не включает полного описания Firebird API. Вы можете ознакомится с ним в каталоге документации, распространяемой вместе с Firebird (`doc/Using_OO_API.html`). Для русскоязычных пользователей существует перевод данного документа доступный по адресу https://github.com/sim1984/fbapi30/releases/download/0.5/fbapi.pdf.

Подключаемые файлы для различных языков программирования, содержащие интерфейсы API, не распространяются в составе дистрибутива Firebird под Windows, однако вы можете извлечь их из распространяемых под Linux сжатых tarbar файлов (путь внутри архива `/opt/firebird/include/firebird/Firebird.pas`).

CLOOP

CLOOP — Cross Language Object Oriented Programming. Этот инструмент не входит в поставку Firebird. Его можно найти в исходных кодах https://github.com/FirebirdSQL/firebird/tree/B3_0_Release/extern/cloop. После того как инструмент будет собран, можно на основе файла описания интерфейсов `include/firebird/FirebirdInterface.idl` сгенерировать API для вашего языка программирования (`IdlFbInterfaces.h` или `Firebird.pas`).

Для Object pascal это делается следующей командой:

```
cloop FirebirdInterface.idl pascal Firebird.pas Firebird --uses SysUtils \
  --interfaceFile Pascal.interface.pas \
  --implementationFile Pascal.implementation.pas \
  --exceptionClass FbException --prefix I \
  --functionsFile fb_get_master_interface.pas
```

Файлы `Pascal.interface.pas`, `Pascal.implementation.pas` и `fb_get_master_interface.pas` можно найти по адресу https://github.com/FirebirdSQL/firebird/tree/B3_0_Release/src/misc/pascal.

> **Примечание**
>
> В данном случае для интерфейсов Firebird API будет добавлен префикс I, так как это принято в Object Pascal.

Константы

В полученном файле `Firebird.pas` отсутствуют `isc_*` константы. Эти константы для языков C/C++ можно найти под адресу https://github.com/FirebirdSQL/firebird/blob/B3_0_Release/src/include/consts_pub.h. Для получения констант для языка Pascal воспользуемся AWK скриптом

для преобразование синтаксиса. В Windows вам потребуется установить Gawk for Windows или воспользоваться Windows Subsystem for Linux (доступно в Windows 10). Это делается следующей командой:

```
awk -f Pascal.Constants.awk consts_pub.h > const.pas
```

Содержимое полученного файла необходимо скопировать в пустую секцию const файла `Firebird.pas` сразу после implementation. Файл `Pascal.Constants.awk`, можно найти по адресу https://github.com/FirebirdSQL/firebird/tree/B3_0_Release/src/misc/pascal.

Управление временем жизни

Интерфейсы Firebird не основываются на спецификации COM, поэтому управление их временем жизни осуществляется иначе.

В Firebird существует два интерфейса, имеющих дело с управлением временем жизни: `IDisposable` и `IReferenceCounted`. Последний особенно активен при создании других интерфейсов: `IPlugin` подсчитывает ссылки, как и многие другие интерфейсы, используемые подключаемыми модулями. К ним относятся интерфейсы, которые описывают соединение с базой данных, управление транзакциями и операторы SQL.

Не всегда нужны дополнительные издержки интерфейса с подсчетом ссылок. Например, `IMaster`, основной интерфейс, который вызывает функции, доступные для остальной части API, имеет неограниченное время жизни по определению. Для других интерфейсов API время жизни строго определяется временем жизни родительского интерфейса; интерфейс `IStatus` не является многопоточным. Для интерфейсов с ограниченным временем жизни полезно иметь простой способ их уничтожения, то есть функцию `dispose()`.

> **Подсказка**
>
> Если вы не знаете, как уничтожается объект, посмотрите его иерархию, если в ней есть интерфейс `IReferenceCounted`. Для интерфейсов с подсчётом ссылок, по завершению работы с объектом необходимо уменьшить счётчик ссылок вызовом метода `release()`.

Глава 2

Объявления UDR

UDR могут быть добавлены или удалены из базы данных с помощью DDL команд подобно тому, как вы добавляете или удаляете обычные PSQL процедуры, функции или триггеры. В этом случае вместо тела триггера указывается место его расположения во внешнем модуле с помощью предложения EXTERNAL NAME.

Рассмотрим синтаксис этого предложения, он будет общим для внешних процедур, функций и триггеров.

Синтаксис:

```
EXTERNAL NAME '<extname>' ENGINE <engine>
[AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

Аргументом этого предложения EXTERNAL NAME является строка, указывающая на расположение функции во внешнем модуле. Для внешних модулей, использующих движок UDR, в этой строке через разделитель указано имя внешнего модуля, имя функции внутри модуля и определённая пользователем информация. В качестве разделителя используется восклицательный знак (!).

В предложении ENGINE указывается имя движка для обработки подключения внешних модулей. В Firebird для работы с внешними модулями написанных на компилируемых языках (C, C++, Pascal) используется движок UDR. Для внешних функциях написанных на Java требуется движок Java.

После ключевого слова AS может быть указан строковый литерал — "тело" внешнего модуля (процедуры. функции или триггера), оно может быть использовано внешним модулем для различных целей. Например, может быть указан SQL запрос для доступа к внешней БД или текст на некотором языке для интерпретации вашей функцией.

Внешние функции

Синтаксис:

```
{CREATE [OR ALTER] | RECREATE} FUNCTION funcname [(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
EXTERNAL NAME <extname> ENGINE <engine>
[AS <extbody>]


<inparam> ::= <param_decl> [{= | DEFAULT} <value>]
```

```
<value> ::=  {literal | NULL | context_var}

<param_decl> ::= paramname <type> [NOT NULL] [COLLATE collation]

<extname> ::= '<module name>!<routine name>[!<misc info>]'

<type> ::= <datatype> | [TYPE OF] domain | TYPE OF COLUMN rel.col

<datatype> ::=
    {SMALLINT | INT[EGER] | BIGINT}
  | BOOLEAN
  | {FLOAT | DOUBLE PRECISION}
  | {DATE | TIME | TIMESTAMP}
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(size)]
    [CHARACTER SET charset]
  | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(size)]
  | BLOB [SUB_TYPE {subtype_num | subtype_name}]
    [SEGMENT SIZE seglen] [CHARACTER SET charset]
  | BLOB [(seglen [, subtype_num])]
```

Все параметры внешней функции можно изменить с помощью оператора ALTER FUNCTION.

Синтаксис:

```
ALTER FUNCTION funcname [(<inparam> [, <inparam> ...])]
RETURNS <type> [COLLATE collation] [DETERMINISTIC]
EXTERNAL NAME <extname> ENGINE <engine>
[AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

Удалить внешнюю функцию можно с помощью оператора DROP FUNCTION.

Синтаксис:

```
DROP FUNCTION funcname
```

Таблица 2.1. Некоторые параметры внешней функции

Параметр	Описание
funcname	Имя хранимой функции. Может содержать до 31 байта.
inparam	Описание входного параметра.
module name	Имя внешнего модуля, в котором расположена функция.
routine name	Внутреннее имя функции внутри внешнего модуля.
misc info	Определяемая пользователем информация для передачи в функцию внешнего модуля.

Параметр	Описание
engine	Имя движка для использования внешних функций. Обычно указывается имя UDR.
extbody	Тело внешней функции. Строковый литерал который может использоваться UDR для различных целей.

Здесь мы не будем описывать синтаксис входных параметров и выходного результата. Он полностью соответствует синтаксису для обычных PSQL функций, который подробно описан в "Руководстве по языку SQL". Вместо этого приведём примеры объявления внешних функций с пояснениями.

Пример 2.1. Функция сложения трёх аргументов

```
create function sum_args (
    n1 integer,
    n2 integer,
    n3 integer
) returns integer
    external name 'udrcpp_example!sum_args'
    engine udr;
```

Реализация функции находится в модуле udrcpp_example. Внутри этого модуля функция зарегистрирована под именем sum_args. Для работы внешней функции используется движок UDR.

Пример 2.2. Функция на языке Java

```
create or alter function regex_replace (
  regex varchar(60),
  str varchar(60),
  replacement varchar(60)
) returns varchar(60)
  external name 'org.firebirdsql.fbjava.examples.fbjava_example.FbRegex.replace(
      String, String, String)'
  engine java;
```

Реализация функции находится в статической функции `replace` класса `org.firebirdsql.fbjava.examples.fbjava_example.FbRegex`. Для работы внешней функции используется движок Java.

Внешние процедуры

Синтаксис:

```
{CREATE [OR ALTER] | RECREATE} PROCEDURE procname [(<inparam> [, <inparam> ...])]
RETURNS (<outparam> [, <outparam> ...])
EXTERNAL NAME <extname> ENGINE <engine>
```

```
[AS <extbody>]

<inparam> ::= <param_decl> [{= | DEFAULT} <value>]

<outparam>  ::=  <param_decl>

<value> ::=  {literal | NULL | context_var}

<param_decl> ::= paramname <type> [NOT NULL] [COLLATE collation]

<extname> ::= '<module name>!<routine name>[!<misc info>]'

<type> ::= <datatype> | [TYPE OF] domain | TYPE OF COLUMN rel.col

<datatype> ::=
    {SMALLINT | INT[EGER] | BIGINT}
  | BOOLEAN
  | {FLOAT | DOUBLE PRECISION}
  | {DATE | TIME | TIMESTAMP}
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(size)]
    [CHARACTER SET charset]
  | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(size)]
  | BLOB [SUB_TYPE {subtype_num | subtype_name}]
    [SEGMENT SIZE seglen] [CHARACTER SET charset]
  | BLOB [(seglen [, subtype_num])]
```

Все параметры внешней процедуры можно изменить с помощью оператора ALTER PROCEDURE.

Синтаксис:

```
ALTER PROCEDURE procname [(<inparam> [, <inparam> ...])]
RETURNS (<outparam> [, <outparam> ...])
EXTERNAL NAME <extname> ENGINE <engine>
[AS <extbody>]
```

Удалить внешнюю процедуру можно с помощью оператора DROP PROCEDURE.

Синтаксис:

```
DROP PROCEDURE procname
```

Таблица 2.2. Некоторые параметры внешней процедуры

Параметр	Описание
procname	Имя хранимой процедуры. Может содержать до 31 байта.
inparam	Описание входного параметра.
outparam	Описание выходного параметра.
module name	Имя внешнего модуля, в котором расположена процедура.

Параметр	Описание
routine name	Внутреннее имя процедуры внутри внешнего модуля.
misc info	Определяемая пользователем информация для передачи в процедуру внешнего модуля.
engine	Имя движка для использования внешних порцедур. Обычно указывается имя UDR.
extbody	Тело внешней процедуры. Строковый литерал который может использоваться UDR для различных целей.

Здесь мы не будем описывать синтаксис входных и выходных параметров. Он полностью соответствует синтаксису для обычных PSQL процедур, который подробно описан в "Руководстве по языку SQL". Вместо этого приведём примеры объявления внешних процедур с пояснениями.

Пример 2.3. Процедура генерации строк результата

```
create procedure gen_rows_pascal (
    start_n integer not null,
    end_n integer not null
) returns (
    result integer not null
)
    external name 'pascaludr!gen_rows'
    engine udr;
```

Реализация функции находится в модуле pascaludr. Внутри этого модуля процедура зарегистрирована под именем gen_rows. Для работы внешней процедуры используется движок UDR.

Пример 2.4. Процедура сохранения произвольных сообщений в лог

```
create or alter procedure write_log (
  message varchar(100)
)
  external name 'pascaludr!write_log'
  engine udr;
```

Реализация функции находится в модуле pascaludr. Внутри этого модуля процедура зарегистрирована под именем write_log. Для работы внешней процедуры используется движок UDR.

Пример 2.5. Выборка записей из таблицы employee внешней базы данных

```
create or alter procedure employee_pgsql (
  -- Firebird 3.0.0 has a bug with external procedures without parameters
  dummy integer = 1
) returns (
  id type of column employee.id,
```

```
  name type of column employee.name
)
  external name 'org.firebirdsql.fbjava.examples.fbjava_example.FbJdbc
    .executeQuery()!jdbc:postgresql:employee|postgres|postgres'
  engine java
  as 'select * from employee';
```

Реализация функции находится в статической функции `executeQuery` класса `org.firebirdsql.fbjava.examples.fbjava_example.FbJdbc`. После восклицательного знака (!) располагаются сведения для подключения к внешней базе данных через JDBC. Для работы внешней функции используется движок Java. Здесь в качестве "тела" внешней процедуру передаётся SQL запрос для извлечения данных.

> **Примечание**
>
> В этой процедуре используется заглушка, в которой передаётся неиспользуемый параметр. Это связано с тем, что в Firebird 3.0 присутствует баг с обработкой внешних процедур без параметров.

Размещение внешних процедур и функций внутри пакетов

Группу взаимосвязанных процедур и функций удобно размещать в PSQL пакетах. В пакетах могут быть расположены как внешние, так и обычные PSQL процедуры и функции.

Синтаксис:

```
{CREATE [OR ALTER] | RECREATE} PACKAGE package_name
AS
BEGIN
  [<package_item> ...]
END

{CREATE | RECREATE} PACKAGE BODY package_name
AS
BEGIN
  [<package_item> ...]
  [<package_body_item> ...]
END

<package_item> ::=
    <function_decl>;
  | <procedure_decl>;

<function_decl> ::=
  FUNCTION func_name [(<in_params>)]
  RETURNS <type> [COLLATE collation]
  [DETERMINISTIC]

<procedure_decl> ::=
  PROCEDURE proc_name [(<in_params>)]
```

```
  [RETURNS (<out_params>)]

<package_body_item> ::=
    <function_impl>
  | <procedure_impl>

<function_impl> ::=
  FUNCTION func_name [(<in_impl_params>)]
  RETURNS <type> [COLLATE collation]
  [DETERMINISTIC]
  <routine body>

<procedure_impl> ::=
  PROCEDURE proc_name [(<in_impl_params>)]
  [RETURNS (<out_params>)]
  <routine body>

<routine body> ::= <sql routine body> | <external body reference>

<sql routine body> ::=
  AS
    [<declarations>]
  BEGIN
    [<PSQL_statements>]
  END

<declarations> ::= <declare_item> [<declare_item> ...]

<declare_item> ::=
    <declare_var>;
  | <declare_cursor>;
  | <subroutine declaration>;
  | <subroutine implimentation>

<subroutine declaration> ::= <subfunc_decl> | <subproc_decl>

<subroutine implimentation> ::= <subfunc_impl> | <subproc_impl>

<external body reference> ::=
  EXTERNAL NAME <extname> ENGINE <engine> [AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

Для внешних процедур и функций в заголовке пакета указываются имя, входные параметры, их типы, значения по умолчанию, и выходные параметры, а в теле пакета всё тоже самое, кроме значений по умолчанию, а также место расположения во внешнем модуле (предложение EXTERNAL NAME), имя движка, и возможно "тело" процедуры/функции.

Пример 2.6. Расположение внешних процедур и функций в пакете

Предположим вы написали UDR для работы с регулярными выражениями, которая расположена во внешнем модуле (динамической библиотеке) PCRE, и у вас есть ещё несколько UDR выполняющих другие задачи. Если бы мы не использовали PSQL пакеты, то все наши внешние процедуры и функции были бы перемешаны как друг с другом, так и с обычными PSQL процедурами и функциями. Это усложняет поиск зависимостей и внесение изменений во внешние модули, а кроме того создаёт путаницу, и заставляет как минимум использовать

префиксы для группировки процедур и функций. PSQL пакеты значительно облегчают нам эту задачу.

```
SET TERM ^;

CREATE OR ALTER PACKAGE REGEXP
AS
BEGIN
  PROCEDURE preg_match(
      APattern VARCHAR(8192), ASubject VARCHAR(8192))
    RETURNS (Matches VARCHAR(8192));

  FUNCTION preg_is_match(
      APattern VARCHAR(8192), ASubject VARCHAR(8192))
    RETURNS BOOLEAN;

  FUNCTION preg_replace(
      APattern VARCHAR(8192),
      AReplacement VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS VARCHAR(8192);

  PROCEDURE preg_split(
      APattern VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS (Lines VARCHAR(8192));

  FUNCTION preg_quote(
      AStr VARCHAR(8192),
      ADelimiter CHAR(10) DEFAULT NULL)
    RETURNS VARCHAR(8192);
END^

RECREATE PACKAGE BODY REGEXP
AS
BEGIN
  PROCEDURE preg_match(
      APattern VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS (Matches VARCHAR(8192))
    EXTERNAL NAME 'PCRE!preg_match' ENGINE UDR;

  FUNCTION preg_is_match(
      APattern VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS BOOLEAN
  AS
  BEGIN
    RETURN EXISTS(
      SELECT * FROM preg_match(:APattern, :ASubject));
  END

  FUNCTION preg_replace(
      APattern VARCHAR(8192),
      AReplacement VARCHAR(8192),
      ASubject VARCHAR(8192))
```

```
    RETURNS VARCHAR(8192)
    EXTERNAL NAME 'PCRE!preg_replace' ENGINE UDR;

  PROCEDURE preg_split(
      APattern VARCHAR(8192),
      ASubject VARCHAR(8192))
    RETURNS (Lines VARCHAR(8192))
    EXTERNAL NAME 'PCRE!preg_split' ENGINE UDR;

  FUNCTION preg_quote(
      AStr VARCHAR(8192),
      ADelimiter CHAR(10))
    RETURNS VARCHAR(8192)
    EXTERNAL NAME 'PCRE!preg_quote' ENGINE UDR;
END^

SET TERM ;^
```

Внешние триггеры

Синтаксис:

```
{CREATE [OR ALTER] | RECREATE} TRIGGER trigname
{
    <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger>
  | <ddl_trigger>
}
<external-body>

<external-body> ::=
  EXTERNAL NAME <extname> ENGINE <engine>
  [AS <extbody>]

<relation_trigger_legacy> ::=
  FOR {tablename | viewname}
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <mutation_list>
  [POSITION number]

<relation_trigger_sql2003> ::=
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <mutation_list>
  [POSITION number]
  ON {tablename | viewname}

<database_trigger> ::=
  [ACTIVE | INACTIVE]
  ON db_event
  [POSITION number]

<ddl_trigger> ::=
```

```
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <ddl_events>
  [POSITION number]

<mutation_list> ::= <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= INSERT | UPDATE | DELETE

<db_event> ::=
    CONNECT
  | DISCONNECT
  | TRANSACTION START
  | TRANSACTION COMMIT
  | TRANSACTION ROLLBACK


<ddl_events> ::=
    ANY DDL STATEMENT
  | <ddl_event_item> [{OR <ddl_event_item>} ...]

<ddl_event_item> ::=
    CREATE TABLE | ALTER TABLE | DROP TABLE
  | CREATE PROCEDURE | ALTER PROCEDURE | DROP PROCEDURE
  | CREATE FUNCTION | ALTER FUNCTION | DROP FUNCTION
  | CREATE TRIGGER | ALTER TRIGGER | DROP TRIGGER
  | CREATE EXCEPTION | ALTER EXCEPTION | DROP EXCEPTION
  | CREATE VIEW | ALTER VIEW | DROP VIEW
  | CREATE DOMAIN | ALTER DOMAIN | DROP DOMAIN
  | CREATE ROLE | ALTER ROLE | DROP ROLE
  | CREATE SEQUENCE | ALTER SEQUENCE | DROP SEQUENCE
  | CREATE USER | ALTER USER | DROP USER
  | CREATE INDEX | ALTER INDEX | DROP INDEX
  | CREATE COLLATION | DROP COLLATION
  | ALTER CHARACTER SET
  | CREATE PACKAGE | ALTER PACKAGE | DROP PACKAGE
  | CREATE PACKAGE BODY | DROP PACKAGE BODY
  | CREATE MAPPING | ALTER MAPPING | DROP MAPPING
```

Внешний триггер можно изменить с помощью оператора ALTER TRIGGER.

Синтаксис:

```
ALTER TRIGGER trigname {
[ACTIVE | INACTIVE]
[
    {BEFORE | AFTER} {<mutation_list> | <ddl_events>}
  | ON db_event
]
[POSITION number]
[<external-body>]

<external-body> ::=
  EXTERNAL NAME <extname> ENGINE <engine>
  [AS <extbody>]

<extname> ::= '<module name>!<routine name>[!<misc info>]'
```

```
<mutation_list> ::= <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= { INSERT | UPDATE | DELETE }
```

Удалить внешний триггер можно с помощью оператора DROP TRIGGER.

Синтаксис:

```
DROP TRIGGER trigname
```

Таблица 2.3. Некоторые параметры внешнего триггера

Параметр	Описание
trigname	Имя триггера. Может содержать до 31 байта.
relation_trigger_legacy	Объявление табличного триггера (унаследованное).
relation_trigger_sql2003	Объявление табличного триггера согласно стандарту SQL-2003.
database_trigger	Объявление триггера базы данных.
ddl_trigger	Объявление DDL триггера.
tablename	Имя таблицы.
viewname	Имя представления.
mutation_list	Список событий таблицы.
mutation	Одно из событий таблицы.
db_event	Событие соединения или транзакции.
ddl_events	Список событий изменения метаданных.
ddl_event_item	Одно из событий изменения метаданных.
number	Порядок срабатывания триггера. От 0 до 32767.
extbody	Тело внешнего триггера. Строковый литерал который может использоваться UDR для различных целей.
module name	Имя внешнего модуля, в котором расположен триггер.
routine name	Внутреннее имя триггера внутри внешнего модуля.
misc info	Определяемая пользователем информация для передачи в триггер внешнего модуля.
engine	Имя движка для использования внешних триггеров. Обычно указывается имя UDR.

Приведём примеры объявления внешних триггеров с пояснениями.

Пример 2.7. Внешний триггер репликации в другую базу данных

```
create database 'c:\temp\slave.fdb';

create table persons (
    id integer not null,
    name varchar(60) not null,
    address varchar(60),
    info blob sub_type text
);

commit;

create database 'c:\temp\master.fdb';

create table persons (
    id integer not null,
    name varchar(60) not null,
    address varchar(60),
    info blob sub_type text
);

create table replicate_config (
    name varchar(31) not null,
    data_source varchar(255) not null
);

insert into replicate_config (name, data_source)
   values ('ds1', 'c:\temp\slave.fdb');

create trigger persons_replicate
    after insert on persons
    external name 'udrcpp_example!replicate!ds1'
    engine udr;
```

Реализация триггера находится в модуле udrcpp_example. Внутри этого модуля триггер зарегистрирован под именем replicate. Для работы внешнего триггера используется движок UDR.

В ссылке на внешний модуль используется дополнительный параметр `ds1`, по которому внутри внешнего триггера из таблицы *replicate_config* читается конфигурация для связи с внешней базой данных.

Глава 3

Структура UDR

Мы будем описывать структуру UDR на языке Pascal. Для объяснения минимальной структуры для построения UDR будем использовать стандартные примеры из `examples/udr/` переведённых на Pascal.

Создайте новый проект новой динамической библиотеки,который назовём MyUdr. В результате у вас должен получиться файл `MyUdr.dpr` (если вы создавали проект в Delphi) или файл `MyUdr.lpr` (если вы создали проект в Lazarus). Теперь изменим главный файл проекта так чтобы он выглядел следующим образом:

```
library MyUdr;

{$IFDEF FPC}
  {$MODE DELPHI}{$H+}
{$ENDIF}

uses
{$IFDEF unix}
    cthreads,
    // the c memory manager is on some systems much faster for multi-threading
    cmem,
{$ENDIF}
  UdrInit in 'UdrInit.pas',
  SumArgsFunc in 'SumArgsFunc.pas';

exports firebird_udr_plugin;

end.
```

В данном случае необходимо экспортировать всего одну функцию `firebird_udr_plugin`, которая является точкой входа для плагина внешних модулей UDR. Реализация этой функции будет находится в модуле UdrInit.

Особенности Free Pascal

Если вы разрабатываете вашу UDR в Free Pascal, то вам потребуются дополнительные директивы. Директива `{$mode objfpc}` требуется для включения режима Object Pascal. Вместо неё вы можете использовать директиву `{$mode delphi}` для обеспечения совместимости с Delphi. Поскольку мои примеры должны успешно компилироваться как в FPC, так и в Delphi я выбираю режим `{$mode delphi}`.

Директива `{$H+}` включает поддержку длинных строк. Это необходимо если вы будете пользоваться типы string, ansistring, а не только нуль-терминированные строки PChar, PAnsiChar, PWideChar.

Кроме того, нам потребуется подключить отдельные модули для поддержки многопоточности в Linux и других Unix-подобных операционных системах.

Регистрация функций

Теперь добавим модуль UdrInit, он должен выглядеть следующим образом:

```
unit UdrInit;

{$IFDEF FPC}
  {$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird;

// точка входа для External Engine модуля UDR
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;

implementation

uses
  SumArgsFunc;

var
  myUnloadFlag: Boolean;
  theirUnloadFlag: BooleanPtr;

function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // регистрируем наши функции
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // регистрируем наши процедуры
  //AUdrPlugin.registerProcedure(AStatus, 'sum_args_proc',
  //  TSumArgsProcedureFactory.Create());
  //AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
```

```
  // регистрируем наши триггеры
  //AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
  //  TMyTriggerFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;

initialization

myUnloadFlag := false;

finalization

if ((theirUnloadFlag <> nil) and not myUnloadFlag) then
  theirUnloadFlag^ := true;

end.
```

В функции `firebird_udr_plugin` необходимо зарегистрировать фабрики наших внешних процедур, функций и триггеров. Для каждой функции, процедуры или триггера необходимо написать свою фабрику. Это делается с помощью методов интерфейса `IUdrPlugin`:

- `registerFunction` - регистрирует внешнюю функцию;
- `registerProcedure` - регистрирует внешнюю процедуру;
- `registerTrigger` - регистрирует внешний триггер.

Первым аргументом этих функций является указатель на статус вектор, далее следует внутреннее имя функции (процедуры или триггера). Внутреннее имя будет использоваться при создании процедуры/функции/триггера на SQL. Третьим аргументом передаётся экземпляр фабрики для создания функции (процедуры или триггера).

Фабрика функций

Теперь необходимо написать фабрику и саму функцию. Они будут расположены в модуле SumArgsFunc. Примеры для написания процедур и триггеров будут представлены позже.

```
unit SumArgsFunc;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird;

// ****************************************************
//    create function sum_args (
//      n1 integer,
```

```
//      n2 integer,
//      n3 integer
//    ) returns integer
//    external name 'myudr!sum_args'
//    engine udr;
// ******************************************************

type
  // структура на которое будет отображено входное сообщение
  TSumArgsInMsg = record
    n1: Integer;
    n1Null: WordBool;
    n2: Integer;
    n2Null: WordBool;
    n3: Integer;
    n3Null: WordBool;
  end;
  PSumArgsInMsg = ^TSumArgsInMsg;

  // структура на которое будет отображено выходное сообщение
  TSumArgsOutMsg = record
    result: Integer;
    resultNull: WordBool;
  end;
  PSumArgsOutMsg = ^TSumArgsOutMsg;

  // Фабрика для создания экземпляра внешней функции TSumArgsFunction
  TSumArgsFunctionFactory = class(IUdrFunctionFactoryImpl)
    // Вызывается при уничтожении фабрики
    procedure dispose(); override;

    { Выполняется каждый раз при загрузке внешней функции в кеш метаданных.
      Используется для изменения формата входного и выходного сообщения.

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AMetadata Метаданные внешней функции)
      @param(AInBuilder Построитель сообщения для входных метаданных)
      @param(AOutBuilder Построитель сообщения для выходных метаданных)
    }
    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    { Создание нового экземпляра внешней функции TSumArgsFunction

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AMetadata Метаданные внешней функции)
      @returns(Экземпляр внешней функции)
    }
    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;

  // Внешняя функция TSumArgsFunction.
  TSumArgsFunction = class(IExternalFunctionImpl)
    // Вызывается при уничтожении экземпляра функции
```

```
    procedure dispose(); override;

    { Этот метод вызывается непосредственно перед execute и сообщает
      ядру наш запрошенный набор символов для обмена данными внутри
      этого метода. Во время этого вызова контекст использует набор символов,
      полученный из ExternalEngine::getCharSet.

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AName Имя набора символов)
      @param(AName Длина имени набора символов)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { Выполнение внешней функции

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AInMsg Указатель на входное сообщение)
      @param(AOutMsg Указатель на выходное сообщение)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;

implementation

{ TSumArgsFunctionFactory }

procedure TSumArgsFunctionFactory.dispose;
begin
  Destroy;
end;

function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := TSumArgsFunction.Create();
end;

procedure TSumArgsFunctionFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TSumArgsFunction }

procedure TSumArgsFunction.dispose;
begin
  Destroy;
end;

procedure TSumArgsFunction.execute(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer);
var
```

```
  xInput: PSumArgsInMsg;
  xOutput: PSumArgsOutMsg;
begin
  // преобразовываем указатели на вход и выход к типизированным
  xInput := PSumArgsInMsg(AInMsg);
  xOutput := PSumArgsOutMsg(AOutMsg);
  // если один из аргументов NULL значит и результат NULL
  xOutput^.resultNull := xInput^.n1Null or xInput^.n2Null or xInput^.n3Null;
  xOutput^.result := xInput^.n1 + xInput^.n2 + xInput^.n3;
end;

procedure TSumArgsFunction.getCharSet(AStatus: IStatus;
  AContext: IExternalContext; AName: PAnsiChar; ANameSize: Cardinal);
begin
end;

end.
```

Фабрика внешней функции должна реализовать интерфейс `IUdrFunctionFactory`. Для упрощения просто наследуем класс `IUdrFunctionFactoryImpl`. Для каждой внешней функции нужна своя фабрика. Впрочем, если фабрики не имеют специфики для создания некоторой функции, то можно написать обобщённую фабрику с помощью дженериков. Позже мы приведём пример как это сделать.

Метод `dispose` вызывается при уничтожении фабрики, в нём мы должны освободить ранее выделенные ресурсы. В данном случае просто вызываем деструктор.

Метод `setup` выполняется каждый раз при загрузке внешней функции в кеш метаданных. В нём можно делать различные действия которые необходимы перед созданием экземпляра функции, например изменить формат для входных и выходных сообщений. Более подробно поговорим о нём позже.

Метод `newItem` вызывается для создания экземпляра внешней функции. В этот метод передаётся указатель на статус вектор, контекст внешней функции и метаданные внешней функции. С помощью IRoutineMetadata вы можете получить формат входного и выходного сообщения, тело внешней функции и другие метаданные. В этом методе вы можете создавать различные экземпляры внешней функции в зависимости от её объявления в PSQL. Метаданные можно передать в созданный экземпляр внешней функции если это необходимо. В нашем случае мы просто создаём экземпляр внешней функции `TSumArgsFunction`.

Экземпляр функции

Внешняя функция должна реализовать интерфейс `IExternalFunction`. Для упрощения просто наследуем класс `IExternalFunctionImpl`.

Метод `dispose` вызывается при уничтожении экземпляра функции, в нём мы должны освободить ранее выделенные ресурсы. В данном случае просто вызываем деструктор.

Метод `getCharSet` используется для того, чтобы сообщить внешней функции набор символов используемый при подключении к текущей базе данных. В большинстве случаев в этом нет необходимости, так как набор символов для входных и выходных переменных описан в метаданных при создании функции.

Метод `execute` обрабатывает непосредственно сам вызов функции. В этот метод передаётся указатель на статус вектор, указатель на контекст внешней функции, указатели на входное и выходное сообщение.

Контекст внешней функции может потребоваться нам для получения контекста текущего соединения или транзакции. Даже если вы не будете использовать запросы к базе данных в текущем соединении, то эти контексты всё равно могут потребоваться вам, особенно при работе с типом BLOB. Примеры работы с типом BLOB, а также использование контекстов соединения и транзакции будут показаны позже.

Входные и выходные сообщения имеют фиксированную ширину, которая зависит от типов данных декларируемых для входных и выходных переменных соответственно. Это позволяет использовать типизированные указатели на структуры фиксированный ширины, члены который должны соответствовать типам данных. Из примера видно, что для каждой переменной в структуре указывается член соответствующего типа, после чего идёт член, который является признаком специального значения NULL (далее Null флаг). Помимо работы с буферами входных и выходных сообщений через структуры, существует ещё один способ с использованием адресной арифметики на указателях с использованием смещениях, значения которых можно получить из интерфейса `IMessageMetadata`. Подробнее о работе с сообщениями мы поговорим далее, а сейчас просто поясним что делалось в методе `execute`.

Первым делом мы преобразовываем не типизированные указатели к типизированным. Для выходного значения устанавливаем Null флаг равный логическому объединению Null флагов у всех входных аргументов, если ни один из входных аргументов не равен NULL, то выходное значение будет равно сумме значений аргументов.

Регистрация процедур

Пришло время добавить в наш UDR модуль хранимую процедуру. Как известно хранимые процедуры бывают двух видов: выполняемые хранимые процедуры и хранимые процедуры для выборки данных. Сначала добавим выполняемую хранимую процедуру, т.е. такую хранимую процедуру которая может быть вызвана с помощью оператора EXECUTE PROCEDURE и может вернуть не более одной записи.

Вернитесь в модуль UdrInit и измените функцию `firebird_udr_plugin` так чтобы она выглядела следующим образом.

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // регистрируем наши функции
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // регистрируем наши процедуры
  AUdrPlugin.registerProcedure(AStatus, 'sum_args_proc',
    TSumArgsProcedureFactory.Create());
  //AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
  // регистрируем наши триггеры
  //AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
  //  TMyTriggerFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
```

```
  Result := @myUnloadFlag;
end;
```

> **Примечание**
>
> Не забудьте добавить с список uses модуль SumArgsProc, в котором и будет расположена наша процедура.

Фабрика процедур

Фабрика внешней процедуры должна реализовать интерфейс `IUdrProcedureFactory`. Для упрощения просто наследуем класс `IUdrProcedureFactoryImpl`. Для каждой внешней процедуры нужна своя фабрика. Впрочем, если фабрики не имеют специфики для создания некоторой процедуры, то можно написать обобщённую фабрику с помощью дженериков. Позже мы приведём пример как это сделать.

Метод `dispose` вызывается при уничтожении фабрики, в нём мы должны освободить ранее выделенные ресурсы. В данном случае просто вызываем деструктор.

Метод `setup` выполняется каждый раз при загрузке внешней процедуры в кеш метаданных. В нём можно делать различные действия которые необходимы перед созданием экземпляра процедуры, например изменение формата для входных и выходных сообщений. Более подробно поговорим о нём позже.

Метод `newItem` вызывается для создания экземпляра внешней процедуры. В этот метод передаётся указатель на статус вектор, контекст внешней процедуры и метаданные внешней процедуры. С помощью IRoutineMetadata вы можете получить формат входного и выходного сообщения, тело внешней функции и другие метаданные. В этом методе вы можете создавать различные экземпляры внешней функции в зависимости от её объявления в PSQL. Метаданные можно передать в созданный экземпляр внешней процедуры если это необходимо. В нашем случае мы просто создаём экземпляр внешней процедуры `TSumArgsProcedure`.

Фабрику процедуры а также саму процедуру расположим в модуле SumArgsProc.

```
unit SumArgsProc;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird;

  { ******************************************************

    create procedure sp_sum_args (
      n1 integer,
```

```
      n2 integer,
      n3 integer
    ) returns (result integer)
    external name 'myudr!sum_args_proc'
    engine udr;

    ****************************************************** }
type
  // структура на которое будет отображено входное сообщение
  TSumArgsInMsg = record
    n1: Integer;
    n1Null: WordBool;
    n2: Integer;
    n2Null: WordBool;
    n3: Integer;
    n3Null: WordBool;
  end;
  PSumArgsInMsg = ^TSumArgsInMsg;

  // структура на которое будет отображено выходное сообщение
  TSumArgsOutMsg = record
    result: Integer;
    resultNull: WordBool;
  end;
  PSumArgsOutMsg = ^TSumArgsOutMsg;

  // Фабрика для создания экземпляра внешней процедуры TSumArgsProcedure
  TSumArgsProcedureFactory = class(IUdrProcedureFactoryImpl)
    // Вызывается при уничтожении фабрики
    procedure dispose(); override;

    { Выполняется каждый раз при загрузке внешней процедуры в кеш метаданных
      Используется для изменения формата входного и выходного сообщения.

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней процедуры)
      @param(AMetadata Метаданные внешней процедуры)
      @param(AInBuilder Построитель сообщения для входных метаданных)
      @param(AOutBuilder Построитель сообщения для выходных метаданных)
    }
    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    { Создание нового экземпляра внешней процедуры TSumArgsProcedure

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней процедуры)
      @param(AMetadata Метаданные внешней процедуры)
      @returns(Экземпляр внешней процедуры)
    }
    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalProcedure; override;
  end;

  TSumArgsProcedure = class(IExternalProcedureImpl)
  public
    // Вызывается при уничтожении экземпляра процедуры
```

```
    procedure dispose(); override;

    { Этот метод вызывается непосредственно перед open и сообщает
      ядру наш запрошенный набор символов для обмена данными внутри
      этого метода. Во время этого вызова контекст использует набор символов,
      полученный из ExternalEngine::getCharSet.

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AName Имя набора символов)
      @param(AName Длина имени набора символов)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { Выполнение внешней процедуры

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AInMsg Указатель на входное сообщение)
      @param(AOutMsg Указатель на выходное сообщение)
      @returns(Набор данных для селективной процедуры или
               nil для процедур выполнения)
    }
    function open(AStatus: IStatus; AContext: IExternalContext; AInMsg: Pointer;
      AOutMsg: Pointer): IExternalResultSet; override;
  end;

implementation

{ TSumArgsProcedureFactory }

procedure TSumArgsProcedureFactory.dispose;
begin
  Destroy;
end;

function TSumArgsProcedureFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalProcedure;
begin
  Result := TSumArgsProcedure.create;
end;

procedure TSumArgsProcedureFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata; AInBuilder,
  AOutBuilder: IMetadataBuilder);
begin

end;

{ TSumArgsProcedure }

procedure TSumArgsProcedure.dispose;
begin
  Destroy;
end;

procedure TSumArgsProcedure.getCharSet(AStatus: IStatus;
```

```
  AContext: IExternalContext; AName: PAnsiChar; ANameSize: Cardinal);
begin

end;

function TSumArgsProcedure.open(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer): IExternalResultSet;
var
  xInput: PSumArgsInMsg;
  xOutput: PSumArgsOutMsg;
begin
  Result := nil;
  // преобразовываем указатели на вход и выход к типизированным
  xInput := PSumArgsInMsg(AInMsg);
  xOutput := PSumArgsOutMsg(AOutMsg);
  // если один из аргументов NULL значит и результат NULL
  xOutput^.resultNull := xInput^.n1Null or xInput^.n2Null or xInput^.n3Null;
  xOutput^.result := xInput^.n1 + xInput^.n2 + xInput^.n3;
end;

end.
```

Экземпляр процедуры

Внешняя процедура должна реализовать интерфейс `IExternalProcedure`. Для упрощения просто наследуем класс `IExternalProcedureImpl`.

Метод `dispose` вызывается при уничтожении экземпляра процедуры, в нём мы должны освободить ранее выделенные ресурсы. В данном случае просто вызываем деструктор.

Метод `getCharSet` используется для того чтобы сообщить внешней процедуре набор символов используемый при подключении к текущей базе данных. В большинстве случаев в этом нет необходимости, так как набор символов для входных и выходных переменных описан в метаданных при создании процедуры.

Метод `open` обрабатывает непосредственно сам вызов процедуры. В этот метод передаётся указатель на статус вектор, указатель на контекст внешней функции, указатели на входное и выходное сообщение. Если у вас выполняемая процедура, то метод должен вернуть значение nil, в противном случае должен вернуться экземпляр набора выходных данных для процедуры. В данном случае нам не нужно создавать экземпляр набора данных. Просто переносим логику из метода `TSumArgsFunction.execute`.

Хранимая процедура выбора

Теперь добавим в наш UDR модуль простую процедуру выбора. Для этого изменим функцию регистрации `firebird_udr_plugin`.

```

function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
```

```
begin
  // регистрируем наши функции
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // регистрируем наши процедуры
  AUdrPlugin.registerProcedure(AStatus, 'sum_args_proc',
    TSumArgsProcedureFactory.Create());
  AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
  // регистрируем наши триггеры
  //AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
  //  TMyTriggerFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Примечание

Не забудьте добавить с список uses модуль GenRowsProc, в котором и будет расположена наша процедура.

Фабрика процедур полностью идентична как для случая с выполняемой хранимой процедурой. Методы экземпляра процедуры тоже идентичны, за исключением метода `open`, который разберём чуть подробнее.

```
unit GenRowsProc;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses
  Firebird, SysUtils;

type
  { ******************************************************

    create procedure gen_rows (
      start  integer,
      finish integer
    ) returns (n integer)
    external name 'myudr!gen_rows'
    engine udr;

    ****************************************************** }

  TInput = record
    start: Integer;
    startNull: WordBool;
    finish: Integer;
    finishNull: WordBool;
```

```
end;
PInput = ^TInput;

TOutput = record
  n: Integer;
  nNull: WordBool;
end;
POutput = ^TOutput;

// Фабрика для создания экземпляра внешней процедуры TGenRowsProcedure
TGenRowsFactory = class(IUdrProcedureFactoryImpl)
  // Вызывается при уничтожении фабрики
  procedure dispose(); override;

  { Выполняется каждый раз при загрузке внешней функции в кеш метаданных.
    Используется для изменения формата входного и выходного сообщения.

    @param(AStatus Статус вектор)
    @param(AContext Контекст выполнения внешней функции)
    @param(AMetadata Метаданные внешней функции)
    @param(AInBuilder Построитель сообщения для входных метаданных)
    @param(AOutBuilder Построитель сообщения для выходных метаданных)
  }
  procedure setup(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
    AOutBuilder: IMetadataBuilder); override;

  { Создание нового экземпляра внешней процедуры TGenRowsProcedure

    @param(AStatus Статус вектор)
    @param(AContext Контекст выполнения внешней функции)
    @param(AMetadata Метаданные внешней функции)
    @returns(Экземпляр внешней функции)
  }
  function newItem(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata): IExternalProcedure; override;
end;

// Внешняя процедура TGenRowsProcedure.
TGenRowsProcedure = class(IExternalProcedureImpl)
public
  // Вызывается при уничтожении экземпляра процедуры
  procedure dispose(); override;

  { Этот метод вызывается непосредственно перед open и сообщает
    ядру наш запрошенный набор символов для обмена данными внутри
    этого метода. Во время этого вызова контекст использует набор символов,
    полученный из ExternalEngine::getCharSet.

    @param(AStatus Статус вектор)
    @param(AContext Контекст выполнения внешней функции)
    @param(AName Имя набора символов)
    @param(AName Длина имени набора символов)
  }
  procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
    AName: PAnsiChar; ANameSize: Cardinal); override;

  { Выполнение внешней процедуры
```

```
      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AInMsg Указатель на входное сообщение)
      @param(AOutMsg Указатель на выходное сообщение)
      @returns(Набор данных для селективной процедуры или
               nil для процедур выполнения)
    }
    function open(AStatus: IStatus; AContext: IExternalContext; AInMsg: Pointer;
      AOutMsg: Pointer): IExternalResultSet; override;
  end;

  // Выходной набор данных для процедуры TGenRowsProcedure
  TGenRowsResultSet = class(IExternalResultSetImpl)
    Input: PInput;
    Output: POutput;

    // Вызывается при уничтожении экземпляра набора данных
    procedure dispose(); override;

    { Извлечение очередной записи из набора данных.
      В некотором роде аналог SUSPEND. В этом методе должна
      подготавливаться очередная запись из набора данных.

      @param(AStatus Статус вектор)
      @returns(True если в наборе данных есть запись для извлечения,
               False если записи закончились)
    }
    function fetch(AStatus: IStatus): Boolean; override;
  end;

implementation

{ TGenRowsFactory }

procedure TGenRowsFactory.dispose;
begin
  Destroy;
end;

function TGenRowsFactory.newItem(AStatus: IStatus; AContext: IExternalContext;
  AMetadata: IRoutineMetadata): IExternalProcedure;
begin
  Result := TGenRowsProcedure.create;
end;

procedure TGenRowsFactory.setup(AStatus: IStatus; AContext: IExternalContext;
  AMetadata: IRoutineMetadata; AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TGenRowsProcedure }

procedure TGenRowsProcedure.dispose;
begin
  Destroy;
end;
```

```
procedure TGenRowsProcedure.getCharSet(AStatus: IStatus;
  AContext: IExternalContext; AName: PAnsiChar; ANameSize: Cardinal);
begin

end;

function TGenRowsProcedure.open(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer): IExternalResultSet;
begin
  // если один из входных аргументов NULL ничего не возвращаем
  if PInput(AInMsg).startNull or PInput(AInMsg).finishNull then
  begin
    POutput(AOutMsg).nNull := True;
    Result := nil;
    exit;
  end;
  // проверки
  if PInput(AInMsg).start > PInput(AInMsg).finish then
    raise Exception.Create('First parameter greater then second parameter.');

  Result := TGenRowsResultSet.create;
  with TGenRowsResultSet(Result) do
  begin
    Input := AInMsg;
    Output := AOutMsg;
    // начальное значение
    Output.nNull := False;
    Output.n := Input.start - 1;
  end;
end;

{ TGenRowsResultSet }

procedure TGenRowsResultSet.dispose;
begin
  Destroy;
end;

// Если возвращает True то извлекается очередная запись из набора данных.
// Если возвращает False то записи в наборе данных закончились
// новые значения в выходном векторе вычисляются каждый раз
// при вызове этого метода
function TGenRowsResultSet.fetch(AStatus: IStatus): Boolean;
begin
  Inc(Output.n);
  Result := (Output.n <= Input.finish);
end;

end.
```

В методе `open` экземпляра процедуры `TGenRowsProcedure` проверяем первый и второй входной аргумент на значение NULL, если один из аргументов равен NULL, то и выходной аргумент равен NULL, кроме того процедура не должна вернуть ни одной строки при выборке через оператор SELECT, поэтому результатом этого метода будет nil.

Кроме того мы проверяем, чтобы первый аргумент не превышал значение второго, в противном случае бросаем исключение. Не волнуйтесь это исключение будет перехвачено в подсистеме UDR и преобразовано к исключению Firebird. Это одно из преимуществ новых UDR перед Legacy UDF.

Поскольку мы создаём процедуру выбора, то метод `open` должен возвращать экземпляр набора данных, который реализует интерфейс `IExternalResultSet`. Для упрощения унаследуем свой набор данных от класса `IExternalResultSetImpl`.

Метод `dispose` предназначен для освобождения выделенных ресурсов. В нём мы просто вызываем деструктор.

Метод `fetch` вызывается при извлечении очередной записи оператором SELECT. Этот метод по сути является аналогом оператора SUSPEND используемый в обычных PSQL хранимых процедурах. Каждый раз когда он вызывается, в нём подготавливаются новые значения для выходного сообщения. Метод возвращает true, если запись должна быть возвращена вызывающей стороне, и `false`, если данных для извлечения больше нет. В нашем случае мы просто инкрементируем текущее значение выходной переменной до тех пор, пока оно не больше максимальной границы.

> **Примечание**
>
> В Delphi нет поддержки оператора yeild, таким образом у вас не получится написать код вроде
>
> ```
> while(...) do {
>   ...
>   yield result;
> }
> ```
>
> Вы можете использовать любой класс коллекции, заполнить его в методе `open`, хранимой процедуры, и затем поэлементно возвращать значения из этой коллекции в `fetch`. Однако в этом случае вы лишаетесь возможности досрочно прервать выполнение процедуры (неполный фетч в SELECT или ограничители FIRST/ROWS/FETCH FIRST в операторе SELECT.)

Регистрация триггеров

Теперь добавим в наш UDR модуль внешний триггер.

> **Примечание**
>
> В оригинальных примерах на C++ триггер копирует запись в другую внешнюю базу данных. Я считаю, что такой пример излишне сложен для первого ознакомления с внешними триггерами. Работа с подключениями к внешним базам данных будет рассмотрен позже.

Вернитесь в модуль UdrInit и измените функцию `firebird_udr_plugin` так чтобы она выглядела следующим образом.

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
```

```
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // регистрируем наши функции
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // регистрируем наши процедуры
  AUdrPlugin.registerProcedure(AStatus, 'sum_args_proc',
    TSumArgsProcedureFactory.Create());
  AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
  // регистрируем наши триггеры
  AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
    TMyTriggerFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Примечание

Не забудьте добавить с список uses модуль TestTrigger, в котором и будет расположен наш триггер.

Фабрика триггеров

Фабрика внешнего триггера должна реализовать интерфейс `IUdrTriggerFactory`. Для упрощения просто наследуем класс `IUdrTriggerFactoryImpl`. Для каждого внешнего триггера нужна своя фабрика.

Метод `dispose` вызывается при уничтожении фабрики, в нём мы должны освободить ранее выделенные ресурсы. В данном случае просто вызываем деструктор.

Метод `setup` выполняется каждый раз при загрузке внешнего триггера в кеш метаданных. В нём можно делать различные действия которые необходимы перед созданием экземпляра триггера, например для изменения формата сообщений для полей таблицы. Более подробно поговорим о нём позже.

Метод `newItem` вызывается для создания экземпляра внешнего триггера. В этот метод передаётся указатель на статус вектор, контекст внешнего триггера и метаданные внешнего триггера. С помощью IRoutineMetadata вы можете получить формат сообщения для новых и старых значений полей, тело внешнего триггера и другие метаданные. В этом методе вы можете создавать различные экземпляры внешнего триггера в зависимости от его объявления в PSQL. Метаданные можно передать в созданный экземпляр внешнего триггера если это необходимо. В нашем случае мы просто создаём экземпляр внешнего триггера `TMyTrigger`.

Фабрику триггера а также сам триггер расположим в модуле TestTrigger.

```


unit TestTrigger;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
```

```
{$ENDIF}

interface

uses
  Firebird, SysUtils;

type
  { *******************************************************
    create table test (
      id int generated by default as identity,
      a int,
      b int,
      name varchar(100),
      constraint pk_test primary key(id)
    );

    create or alter trigger tr_test_biu for test
    active before insert or update position 0
    external name 'myudr!test_trigger'
    engine udr;
  }

  // структура для отображения сообщений NEW.* и OLD.*
  // должна соответствовать набору полей таблицы test
  TFieldsMessage = record
    Id: Integer;
    IdNull: WordBool;
    A: Integer;
    ANull: WordBool;
    B: Integer;
    BNull: WordBool;
    Name: record
      Length: Word;
      Value: array [0 .. 399] of AnsiChar;
    end;
    NameNull: WordBool;
  end;

  PFieldsMessage = ^TFieldsMessage;

  // Фабрика для создания экземпляра внешнего триггера TMyTrigger
  TMyTriggerFactory = class(IUdrTriggerFactoryImpl)
    // Вызывается при уничтожении фабрики
    procedure dispose(); override;

    { Выполняется каждый раз при загрузке внешнего триггера в кеш метаданных.
      Используется для изменения формата сообщений для полей.

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешнего триггера)
      @param(AMetadata Метаданные внешнего триггера)
      @param(AFieldsBuilder Построитель сообщения для полей таблицы)
    }
    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder); override;

    { Создание нового экземпляра внешнего триггера TMyTrigger
```

```
      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешнего триггера)
      @param(AMetadata Метаданные внешнего триггера)
      @returns(Экземпляр внешнего триггера)
    }
    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalTrigger; override;
  end;

  TMyTrigger = class(IExternalTriggerImpl)
    // Вызывается при уничтожении триггера
    procedure dispose(); override;

    { Этот метод вызывается непосредственно перед execute и сообщает
      ядру наш запрошенный набор символов для обмена данными внутри
      этого метода. Во время этого вызова контекст использует набор символов,
      полученный из ExternalEngine::getCharSet.

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешнего триггера)
      @param(AName Имя набора символов)
      @param(AName Длина имени набора символов)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;

      AName: PAnsiChar; ANameSize: Cardinal); override;

    { выполнение триггера TMyTrigger

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешнего триггера)
      @param(AAction Действие (текущее событие) триггера)
      @param(AOldMsg Сообщение для старых значение полей :OLD.*)
      @param(ANewMsg Сообщение для новых значение полей :NEW.*)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AAction: Cardinal; AOldMsg: Pointer; ANewMsg: Pointer); override;
  end;

implementation

{ TMyTriggerFactory }

procedure TMyTriggerFactory.dispose;
begin
  Destroy;
end;

function TMyTriggerFactory.newItem(AStatus: IStatus; AContext: IExternalContext;
  AMetadata: IRoutineMetadata): IExternalTrigger;
begin
  Result := TMyTrigger.create;
end;

procedure TMyTriggerFactory.setup(AStatus: IStatus; AContext: IExternalContext;
  AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder);
begin
```

```
end;

{ TMyTrigger }

procedure TMyTrigger.dispose;
begin
  Destroy;
end;

procedure TMyTrigger.execute(AStatus: IStatus; AContext: IExternalContext;
  AAction: Cardinal; AOldMsg, ANewMsg: Pointer);
var
  xOld, xNew: PFieldsMessage;
begin
  // xOld := PFieldsMessage(AOldMsg);
  xNew := PFieldsMessage(ANewMsg);
  case AAction of
    IExternalTrigger.ACTION_INSERT:
      begin
        if xNew.BNull and not xNew.ANull then
        begin
          xNew.B := xNew.A + 1;
          xNew.BNull := False;
        end;
      end;

    IExternalTrigger.ACTION_UPDATE:
      begin
        if xNew.BNull and not xNew.ANull then
        begin
          xNew.B := xNew.A + 1;
          xNew.BNull := False;
        end;
      end;

    IExternalTrigger.ACTION_DELETE:
      begin

      end;
  end;
end;

procedure TMyTrigger.getCharSet(AStatus: IStatus; AContext: IExternalContext;
  AName: PAnsiChar; ANameSize: Cardinal);
begin

end;

end.
```

Экземпляр триггера

Внешний триггер должна реализовать интерфейс `IExternalTrigger`. Для упрощения просто наследуем класс `IExternalTriggerImpl`.

Метод `dispose` вызывается при уничтожении экземпляра триггера, в нём мы должны освободить ранее выделенные ресурсы. В данном случае просто вызываем деструктор.

Метод `getCharSet` используется для того чтобы сообщить внешнему триггеру набор символов используемый при подключении к текущей базе данных. В большинстве случаев в этом нет необходимости, так как набор символов для полей таблицы описан в метаданных таблицы.

Метод `execute` вызывается при выполнении триггера на одно из событий для которого создан триггер. В этот метод передаётся указатель на статус вектор, указатель на контекст внешнего триггера, действие (событие) которое вызвало срабатывание триггера и указатели на сообщения для старых и новых значений полей. Возможные действия (события) триггера перечислены константами в интерфейсе `IExternalTrigger`. Такие константы начинаются с префикса `ACTION_`. Знания о текущем действие необходимо, поскольку в Firebird существуют триггеры созданные для нескольких событий сразу. Сообщения необходимы только для триггеров на действия таблицы, для DDL триггеров, а для триггеров на события подключения, отключения от базы данных и триггеров на события старта, завершения и отката транзакции указатели на сообщения будут инициализированы значением nil. В отличие от процедур и функций сообщения триггеров строятся для полей таблицы на события которой создан триггер. Статические структуры для таких сообщений строятся по тем же принципам, что и структуры сообщений для входных и выходных параметров процедуры, только вместо переменных берутся поля таблицы.

> **Примечание**
>
> Обратите внимание, что если вы используете отображение сообщений на структуры, то ваши триггеры могут сломаться после изменения состава полей таблицы и их типов. Чтобы этого не произошло используйте работу с сообщение через смещения получаемые из `IMessageMetadata`. Это не так актуально для процедур и функций, поскольку входные и выходные параметры меняются не так уж часто. Или хотя бы вы делаете это явно, что может натолкнуть вас на мысль, что необходимо переделать и внешнюю процедуру/функцию.

В нашем простейшем триггере мы определяем тип события, и в теле триггера выполняем следующий PSQL аналог

```
...
  if (:new.B IS NULL) THEN
    :new.B = :new.A + 1;
...
```

Глава 4

Сообщения

Под сообщением в UDR понимается область памяти фиксированного размера для передачи в процедуру или функцию входных аргументов, или возврата выходных аргументов. Для внешних триггеров на события записи таблицы сообщения используются для получения и возврата данных в NEW и OLD.

Для доступа к отдельным переменным или полям таблицы, необходимо знать как минимум тип этой переменной, и смещение от начала буфера сообщений. Как уже упоминалось ранее для этого существует два способа:

- преобразование указателя на буфер сообщения к указателю на статическую структуру (в Delphi это запись, т.е. record);
- получение смещений с помощью экземпляра класса реализующего интерфейс `IMessageMetadata`, и чтение/запись из буфера данных, размером соответствующим типу переменной или поля.

Первый способ является наиболее быстрым, второй — более гибким, так как в ряде случаев позволяет изменять типы и размеры для входных и выходных переменных или полей таблицы без перекомпиляции динамической библиотеки содержащей UDR.

Работа с буфером сообщения с использованием структуры

Как говорилось выше мы можем работать с буфером сообщений через указатель на структуру. Такая структура выглядит следующим образом:

```
TMyStruct = record
  <var_1>: <type_1>;
  <nullIndicator_1>: WordBool;
  <var_2>: <type_1>;
  <nullIndicator_2>: WordBool;
  ...
  <var_N>: <type_1>;
  <nullIndicator_N>: WordBool;
end;
PMyStruct = ^TMyStruct;
```

Типы членов данных должны соответствовать типам входных/выходных переменных или полей (для триггеров). Null-индикатор должен быть после каждой переменной/поля, даже если у них есть ограничение NOT NULL. Null-индикатор занимает 2 байта. Значение -1 обозначает

что переменная/поле имеют значение NULL. Поскольку на данный момент в NULL-индикатор пишется только признак NULL, то удобно отразить его на 2-х байтный логический тип. Типы данных SQL отображаются в структуре следующим образом:

Таблица 4.1. Отображение типов SQL на типы Delphi

SQL тип	Delphi тип	Замечание
BOOLEAN	Boolean, ByteBool	
SMALLINT	Smallint	
INTEGER	Integer	
BIGINT	Int64	
FLOAT	Single	
DOUBLE PRECISION	Double	
NUMERIC(*N*, *M*)	Тип данных зависит от точности и диалекта: • 1-4 — Smallint; • 5-9 — Integer; • 10-18 (3 диалект) — Int64; • 10-15 (1 диалект) — Double.	В качестве значения в сообщение будет передано число умноженное на 10^M.
DECIMAL(*N*, *M*)	Тип данных зависит от точности и диалекта: • 1-4 — Integer; • 5-9 — Integer; • 10-18 (3 диалект) — Int64; • 10-15 (1 диалект) — Double.	В качестве значения в сообщение будет передано число умноженное на 10^M.
CHAR(*N*)	array[0 .. *M*] of AnsiChar	*M* вычисляется по формуле M = N * BytesPerChar - 1, где *BytesPerChar* - количество байт на символ, зависит от кодировки переменной/поля. Например для UTF-8 - это 4 байт/символ, для WIN1251 - 1 байт/символ.

VARCHAR(*N*)	record
	  Length: Smallint;
	  Data: array[0 .. *M*] of AnsiChar;
	end
		M вычисляется по формуле M = N * BytesPerChar - 1, где *BytesPerChar* - количество байт на символ, зависит от кодировки переменной/поля. Например для UTF-8 - это 4 байт/символ, для WIN1251 - 1 байт/символ.

SQL тип	Delphi тип	Замечание
		В *Length* передаётся реальная длина строки в символах.
DATE	ISC_DATE	
TIME	ISC_TIME	

TIMESTAMP	ISC_TIMESTAMP	Тип ISC_TIMESTAMP не определён в Firebird.pas, вы можете определить его сами следующим образом:
		`ISC_TIMESTAMP = record`
		`  date: ISC_DATE;`
		`  time: ISC_TIME;`
		`end;`

BLOB	ISC_QUAD	Содержимое BLOB никогда не передаётся непосредственно, вместо него передаётся BlobId. Как работать с типом BLOB будет рассказано в главе Работа с типом BLOB.

Теперь рассмотрим несколько примеров того как составлять структуры сообщений по декларациям процедур, функций или триггеров.

Пример 4.1. Структуры сообщений для функции

Предположим у нас есть внешняя функция объявленная следующим образом:

```
function SUM_ARGS(A SMALLINT, B INTEGER) RETURNS BIGINT
....
```

В этом случае структуры для входных и выходных сообщений будут выглядеть так:

```
TInput = record
  A: Smallint;
  ANull: WordBool;
  B: Integer;
  BNull: WordBool;
end;
PInput = ^TInput;

TOutput = record
  Value: Int64;
  Null: WordBool;
```

```
end;
POutput = ^TOutput;
```

Если та же самая функция определена с другими типами (в 3 диалекте):

```
function SUM_ARGS(A NUMERIC(4, 2), B NUMERIC(9, 3)) RETURNS NUMERIC(18, 6)
....
```

В этом случае структуры для входных и выходных сообщений будут выглядеть так:

```
TInput = record
  A: Smallint;
  ANull: WordBool;
  B: Integer;
  BNull: WordBool;
end;
PInput = ^TInput;

TOutput = record
  Value: Int64;
  Null: WordBool;
end;
POutput = ^TOutput;
```

Пример 4.2. Структуры сообщений для строковых параметров

Предположим у нас есть внешняя процедура объявленная следующим образом:

```
procedure SOME_PROC(A CHAR(3) CHARACTER SET WIN1251, B VARCHAR(10) CHARACTER SET UTF8)
....
```

В этом случае структуры для входного сообщений будет выглядеть так:

```
TInput = record
  A: array[0..2] of AnsiChar;
  ANull: WordBool;
  B: record
    Length: Smallint;
    Value: array[0..39] of AnsiChar;
  end;
  BNull: WordBool;
end;
```

```
PInput = ^TInput;
```

Работа с буфером сообщений с помощью IMessageMetadata

Как было описано выше с буфером сообщений можно работать с использованием экземпляра объекта реализующего интерфейс `IMessageMetadata`. Этот интерфейс позволяет узнать о переменной/поле следующие сведения:

- имя переменной/поля;
- тип данных;
- набор символов для строковых данных;
- подтип для типа данных BLOB;
- размер буфера в байтах под переменную/поле;
- может ли переменная/поле принимать значение NULL;
- смещение в буфере сообщений для данных;
- смещение в буфере сообщений для NULL-индикатора.

Методы интерфейса IMessageMetadata

1. getCount

```
unsigned getCount(StatusType* status)
```

   возвращает количество полей/параметров в сообщении. Во всех вызовах, содержащих индексный параметр, это значение должно быть: 0 <= index < getCount().

2. getField

```
const char* getField(StatusType* status, unsigned index)
```

   возвращает имя поля.

3. getRelation

```
```

```
const char* getRelation(StatusType* status, unsigned index)
```

возвращает имя отношения (из которого выбрано данное поле).

4. getOwner

```
const char* getOwner(StatusType* status, unsigned index)
```

возвращает имя владельца отношения.

5. getAlias

```
const char* getAlias(StatusType* status, unsigned index)
```

возвращает псевдоним поля.

6. getType

```
unsigned getType(StatusType* status, unsigned index)
```

возвращает SQL тип поля.

7. isNullable

```
FB_BOOLEAN isNullable(StatusType* status, unsigned index)
```

возвращает true, если поле может принимать значение NULL.

8. getSubType

```
int getSubType(StatusType* status, unsigned index)
```

возвращает подтип поля BLOB (0 - двоичный, 1 - текст и т. д.).

9. getLength

```
unsigned getLength(StatusType* status, unsigned index)
```

возвращает максимальную длину поля в байтах.

10 getScale

```
int getScale(StatusType* status, unsigned index)
```

возвращает масштаб для числового поля.

11 getCharSet

```
unsigned getCharSet(StatusType* status, unsigned index)
```

возвращает набор символов для символьных полей и текстового BLOB.

12 getOffset

```
unsigned getOffset(StatusType* status, unsigned index)
```

возвращает смещение данных поля в буфере сообщений (используйте его для доступа к данным в буфере сообщений).

13 getNullOffset

```
unsigned getNullOffset(StatusType* status, unsigned index)
```

возвращает смещение NULL индикатора для поля в буфере сообщений.

14 getBuilder

```
IMetadataBuilder* getBuilder(StatusType* status)
```

возвращает интерфейс `IMetadataBuilder`, инициализированный метаданными этого сообщения.

15 getMessageLength

```
unsigned getMessageLength(StatusType* status)
```

возвращает длину буфера сообщения (используйте его для выделения памяти под буфер).

Получение и использование IMessageMetadata

Экземпляры объектов реализующих интерфейс `IMessageMetadata` для входных и выходных переменных можно получить из интерфейса `IRoutineMetadata`. Он не передаётся непосредственно в экземпляр процедуры, функции или триггера. Это необходимо делать явно в фабрике соответствующего типа. Например:

```
// Фабрика для создания экземпляра внешней функции TSumArgsFunction
TSumArgsFunctionFactory = class(IUdrFunctionFactoryImpl)
  // Вызывается при уничтожении фабрики
  procedure dispose(); override;

  { Выполняется каждый раз при загрузке внешней функции в кеш метаданных

    @param(AStatus Статус вектор)
    @param(AContext Контекст выполнения внешней функции)
    @param(AMetadata Метаданные внешней функции)
    @param(AInBuilder Построитель сообщения для входных метаданных)
    @param(AOutBuilder Построитель сообщения для выходных метаданных)
  }
  procedure setup(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
    AOutBuilder: IMetadataBuilder); override;

  { Создание нового экземпляра внешней функции TSumArgsFunction

    @param(AStatus Статус вектор)
    @param(AContext Контекст выполнения внешней функции)
    @param(AMetadata Метаданные внешней функции)
    @returns(Экземпляр внешней функции)
  }
  function newItem(AStatus: IStatus; AContext: IExternalContext;
    AMetadata: IRoutineMetadata): IExternalFunction; override;
end;

// Внешняя функция TSumArgsFunction.
TSumArgsFunction = class(IExternalFunctionImpl)
private
  FMetadata: IRoutineMetadata;
public
  property Metadata: IRoutineMetadata read FMetadata write FMetadata;
public
  // Вызывается при уничтожении экземпляра функции
  procedure dispose(); override;

  { Этот метод вызывается непосредственно перед execute и сообщает
    ядру наш запрошенный набор символов для обмена данными внутри
    этого метода. Во время этого вызова контекст использует набор символов,
    полученный из ExternalEngine::getCharSet.

    @param(AStatus Статус вектор)
    @param(AContext Контекст выполнения внешней функции)
```

```
      @param(AName Имя набора символов)
      @param(AName Длина имени набора символов)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { Выполнение внешней функции

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AInMsg Указатель на входное сообщение)
      @param(AOutMsg Указатель на выходное сообщение)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;
.......................

{ TSumArgsFunctionFactory }

procedure TSumArgsFunctionFactory.dispose;
begin
  Destroy;
end;

function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := TSumArgsFunction.Create();
  with Result as TSumArgsFunction do
  begin
    Metadata := AMetadata;
  end;
end;

procedure TSumArgsFunctionFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;
```

Экземпляры `IMessageMetadata` для входных и выходных переменных можно получить с помощью методов `getInputMetadata` и `getOutputMetadata` из `IRoutineMetadata`. Метаданные для полей таблицы, на которую написан триггер, можно получить с помощью метода `getTriggerMetadata`.

> **Важно**
>
> Обратите внимание, жизненный цикл объектов интерфейса `IMessageMetadata` управляется с помощью подсчёта ссылок. Он наследует интерфейс `IReferenceCounted`. Методы `getInputMetadata` и `getOutputMetadata` увеличивают счётчик ссылок на 1 для возвращаемых объектов, поэтому после окончания использования этих объектов необходимо уменьшить счётчик ссылок для переменных `xInputMetadata` и `xOutputMetadata` вызывав метод `release`.

Для получения значения соответствующего входного аргумента нам необходимо воспользоваться адресной арифметикой. Для этого получаем смещение из `IMessageMetadata` с помощью метода `getOffset` и прибавляем к адресу буфера для входного сообщения. После чего полученный результат приводим к соответствующему типизированному указателю. Примерна такая же схема работы для получения null индикаторов аргументов, только для получения смещений используется метод `getNullOffset`..

```
.......................

procedure TSumArgsFunction.execute(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer);
var
  n1, n2, n3: Integer;
  n1Null, n2Null, n3Null: WordBool;
  Result: Integer;
  resultNull: WordBool;
  xInputMetadata, xOutputMetadata: IMessageMetadata;
begin
  xInputMetadata := FMetadata.getInputMetadata(AStatus);
  xOutputMetadata := FMetadata.getOutputMetadata(AStatus);
  try
    // получаем значения входных аргументов по их смещениям
    n1 := PInteger(PByte(AInMsg) + xInputMetadata.getOffset(AStatus, 0))^;
    n2 := PInteger(PByte(AInMsg) + xInputMetadata.getOffset(AStatus, 1))^;
    n3 := PInteger(PByte(AInMsg) + xInputMetadata.getOffset(AStatus, 2))^;
    // получаем значения null-индикаторов входных аргументов по их смещениям
    n1Null := PWordBool(PByte(AInMsg) +
      xInputMetadata.getNullOffset(AStatus, 0))^;
    n2Null := PWordBool(PByte(AInMsg) +
      xInputMetadata.getNullOffset(AStatus, 1))^;
    n3Null := PWordBool(PByte(AInMsg) +
      xInputMetadata.getNullOffset(AStatus, 2))^;
    // по умолчанию выходной аргемент = NULL, а потому выставляем ему nullFlag
    resultNull := True;
    Result := 0;
    // если один из аргументов NULL значит и результат NULL
    // в противном случае считаем сумму аргументов
    if not(n1Null or n2Null or n3Null) then
    begin
      Result := n1 + n2 + n3;
      // раз есть результат, то сбрасываем NULL флаг
      resultNull := False;
    end;
    PWordBool(PByte(AInMsg) + xOutputMetadata.getNullOffset(AStatus, 0))^ :=
      resultNull;
    PInteger(PByte(AInMsg) + xOutputMetadata.getOffset(AStatus, 0))^ := Result;
  finally
    xInputMetadata.release;
    xOutputMetadata.release;
  end;
end;
```

Примечание

В главе Контекст соединения и транзакции приведён большой пример для работы с различными SQL типами с использованием интерфейса `IMessageMetadata`.

Глава 5

Фабрики

Вы уже сталкивались с фабриками ранее. Настало время рассмотреть их более подробно.

Фабрики предназначены для создания экземпляров процедур, функций или триггеров. Класс фабрики должен быть наследником одного из интерфейсов `IUdrProcedureFactory`, `IUdrFunctionFactory` или `IUdrTriggerFactory` в зависимости от типа UDR. Их экземпляры должны быть зарегистрированы в качестве точки входа UDR в функции `firebird_udr_plugin`.

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // регистрируем нашу функцию
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TSumArgsFunctionFactory.Create());
  // регистрируем нашу процедуру
  AUdrPlugin.registerProcedure(AStatus, 'gen_rows', TGenRowsFactory.Create());
  // регистрируем наш триггер
  AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
    TMyTriggerFactory.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

В данном примере класс `TSumArgsFunctionFactory` наследует интерфейс `IUdrFunctionFactory`, `TGenRowsFactory` наследует интерфейс `IUdrProcedureFactory`, а `TMyTriggerFactory` наследует интерфейс `IUdrTriggerFactory`.

Экземпляры фабрик создаются и привязываются к точкам входа в момент первой загрузки внешней процедуры, функции или триггера. Это происходит один раз при создании каждого процесса Firebird. Таким образом, для архитектуры SuperServer для всех соединений будет ровно один экземпляр фабрики связанный с каждой точкой входа, для Classic это количество экземпляров будет умножено на количество соединений.

При написании классов фабрик вам необходимо реализовать методы `setup` и `newItem` из интерфейсов `IUdrProcedureFactory`, `IUdrFunctionFactory` или `IUdrTriggerFactory`.

```
  IUdrFunctionFactory = class(IDisposable)
 const VERSION = 3;

 procedure setup(status: IStatus; context: IExternalContext;
   metadata: IRoutineMetadata; inBuilder: IMetadataBuilder;
        outBuilder: IMetadataBuilder);

 function newItem(status: IStatus; context: IExternalContext;
```

```
   metadata: IRoutineMetadata): IExternalFunction;
 end;

 IUdrProcedureFactory = class(IDisposable)
const VERSION = 3;

procedure setup(status: IStatus; context: IExternalContext;
  metadata: IRoutineMetadata; inBuilder: IMetadataBuilder;
       outBuilder: IMetadataBuilder);

function newItem(status: IStatus; context: IExternalContext;
  metadata: IRoutineMetadata): IExternalProcedure;
 end;

 IUdrTriggerFactory = class(IDisposable)
const VERSION = 3;

procedure setup(status: IStatus; context: IExternalContext;
  metadata: IRoutineMetadata; fieldsBuilder: IMetadataBuilder);

function newItem(status: IStatus; context: IExternalContext;
  metadata: IRoutineMetadata): IExternalTrigger;
 end;
```

Кроме того, поскольку эти интерфейсы наследуют интерфейс `IDisposable`, то необходимо так же реализовать метод `dispose`. Это обозначает что Firebird сам выгрузит фабрику, когда это будет необходимо. В методе `dispose` необходимо разместить код, который освобождает ресурсы, при уничтожении экземпляра фабрики. Для упрощения реализации методов интерфейсов удобно воспользоваться классами `IUdrProcedureFactoryImpl`, `IUdrFunctionFactoryImpl`, `IUdrTriggerFactoryImpl`. Рассмотрим каждый из методов более подробно.

Метод newItem

Метод `newItem` вызывается для создания экземпляра внешней процедуры, функции или триггера. Создание экземпляров UDR происходит в момент её загрузки в кэш метаданных, т.е. при первом вызове процедуры, функции или триггера. В настоящий момент кэш метаданных раздельный для каждого соединения для всех архитектур сервера.

Кэш метаданных процедур и функция связан с их именами в базе данных. Например, две внешние функции с разными именами, но одинаковыми точками входа, будут разными экземплярами `IUdrFunctionFactory`. Точка входа состоит из имени внешнего модуля и имени под которым зарегистрирована фабрика. Как это можно использовать покажем позже.

В метод `newItem` передаётся указатель на статус вектор, контекст выполнения UDR и метаданные UDR.

В простейшем случае реализация этого метода тривиальна

```
function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
```

```
  // создаём экземпляр внешней функции
  Result := TSumArgsFunction.Create();
end;
```

С помощью `IRoutineMetadata` вы можете получить формат входного и выходного сообщения, тело UDR и другие метаданные. Метаданные можно передать в созданный экземпляр UDR. В этом случае в экземпляр класса реализующего вашу UDR необходимо добавить поле для хранение метаданных.

```
  // Внешняя функция TSumArgsFunction.
  TSumArgsFunction = class(IExternalFunctionImpl)
  private
    FMetadata: IRoutineMetadata;
  public
    property Metadata: IRoutineMetadata read FMetadata write FMetadata;
  public
  ...
  end;
```

В этом случае реализация метода `newItem` выглядит следующим образом:

```
function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := TSumArgsFunction.Create();
  with Result as TSumArgsFunction do
  begin
    Metadata := AMetadata;
  end;
end;
```

Создание экземпляров UDR в зависимости от их объявления

В методе `newItem` вы можете создавать различные экземпляры внешней процедуры или функции в зависимости от её объявления в PSQL. Для этого можно использовать информацию полученную из `IMessageMetadata`.

Допустим мы хотим реализовать PSQL пакет с однотипным набором внешних функций для возведения числа в квадрат под различные типы данных и единой точкой входа.

```


SET TERM ^ ;

CREATE OR ALTER PACKAGE MYUDR2
AS
begin
  function SqrSmallint(AInput SMALLINT) RETURNS INTEGER;
  function SqrInteger(AInput INTEGER) RETURNS BIGINT;
  function SqrBigint(AInput BIGINT) RETURNS BIGINT;
```

```
  function SqrFloat(AInput FLOAT) RETURNS DOUBLE PRECISION;
  function SqrDouble(AInput DOUBLE PRECISION) RETURNS DOUBLE PRECISION;
end^

RECREATE PACKAGE BODY MYUDR2
AS
begin
  function SqrSmallint(AInput SMALLINT) RETURNS INTEGER
  external name 'myudr2!sqrt_func'
  engine udr;

  function SqrInteger(AInput INTEGER) RETURNS BIGINT
  external name 'myudr2!sqrt_func'
  engine udr;

  function SqrBigint(AInput BIGINT) RETURNS BIGINT
  external name 'myudr2!sqrt_func'
  engine udr;

  function SqrFloat(AInput FLOAT) RETURNS DOUBLE PRECISION
  external name 'myudr2!sqrt_func'
  engine udr;

  function SqrDouble(AInput DOUBLE PRECISION) RETURNS DOUBLE PRECISION
  external name 'myudr2!sqrt_func'
  engine udr;

end
^

SET TERM ; ^
```

Для проверки функций будем использовать следующий запрос

```
select
  myudr2.SqrSmallint(1) as n1,
  myudr2.SqrInteger(2) as n2,
  myudr2.SqrBigint(3) as n3,
  myudr2.SqrFloat(3.1) as n4,
  myudr2.SqrDouble(3.2) as n5
from rdb$database
```

Для упрощения работы с `IMessageMetadata` и буферами можно написать удобную обёртку или попробовать совместно использовать `IMessageMetadata` и структуры для отображения сообщений. Здесь мы покажем использование второго способа.

Реализация такой идея достаточно проста: в фабрике функций мы будем создавать различные экземпляры функций в зависимости от типа входного аргумента. В современных версиях Delphi вы можете использовать дженерики для обобщения кода.

```
.......................
```

```
type
  // структура на которое будет отображено входное сообщение
  TSqrInMsg<T> = record
    n1: T;
    n1Null: WordBool;
  end;

  // структура на которое будет отображено выходное сообщение
  TSqrOutMsg<T> = record
    result: T;
    resultNull: WordBool;
  end;

  // Фабрика для создания экземпляра внешней функции TSqrFunction
  TSqrFunctionFactory = class(IUdrFunctionFactoryImpl)
    // Вызывается при уничтожении фабрики
    procedure dispose(); override;

    { Выполняется каждый раз при загрузке внешней функции в кеш метаданных.
      Используется для изменения формата входного и выходного сообщения.

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AMetadata Метаданные внешней функции)
      @param(AInBuilder Построитель сообщения для входных метаданных)
      @param(AOutBuilder Построитель сообщения для выходных метаданных)
    }
    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    { Создание нового экземпляра внешней функции TSqrFunction

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AMetadata Метаданные внешней функции)
      @returns(Экземпляр внешней функции)
    }
    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;


  // Внешняя функция TSqrFunction.
  TSqrFunction<TIn, TOut> = class(IExternalFunctionImpl)
  private
    function sqrExec(AIn: TIn): TOut; virtual; abstract;
  public
    type
      TInput = TSqrInMsg<TIn>;
      TOutput = TSqrOutMsg<TOut>;
      PInput = ^TInput;
      POutput = ^TOutput;
    // Вызывается при уничтожении экземпляра функции
    procedure dispose(); override;

    { Этот метод вызывается непосредственно перед execute и сообщает
      ядру наш запрошенный набор символов для обмена данными внутри
```

```
      этого метода. Во время этого вызова контекст использует набор символов,
      полученный из ExternalEngine::getCharSet.

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AName Имя набора символов)
      @param(AName Длина имени набора символов)
    }
    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { Выполнение внешней функции

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AInMsg Указатель на входное сообщение)
      @param(AOutMsg Указатель на выходное сообщение)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;

  TSqrExecSmallint = class(TSqrFunction<Smallint, Integer>)
  public
    function sqrExec(AIn: Smallint): Integer; override;
  end;

  TSqrExecInteger = class(TSqrFunction<Integer, Int64>)
  public
    function sqrExec(AIn: Integer): Int64; override;
  end;

  TSqrExecInt64 = class(TSqrFunction<Int64, Int64>)
  public
    function sqrExec(AIn: Int64): Int64; override;
  end;

  TSqrExecFloat = class(TSqrFunction<Single, Double>)
  public
    function sqrExec(AIn: Single): Double; override;
  end;

  TSqrExecDouble = class(TSqrFunction<Double, Double>)
  public
    function sqrExec(AIn: Double): Double; override;
  end;

implementation

uses
  SysUtils, FbTypes, System.TypInfo;

{ TSqrFunctionFactory }

procedure TSqrFunctionFactory.dispose;
begin
  Destroy;
```

```
end;

function TSqrFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
var
  xInputMetadata: IMessageMetadata;
  xInputType: TFBType;
begin
  // получаем тип входного аргумента
  xInputMetadata := AMetadata.getInputMetadata(AStatus);
  xInputType := TFBType(xInputMetadata.getType(AStatus, 0));
  xInputMetadata.release;
  // создаём экземпляр функции в зависимости от типа
  case xInputType of
    SQL_SHORT:
      result := TSqrExecSmallint.Create();

    SQL_LONG:
      result := TSqrExecInteger.Create();
    SQL_INT64:
      result := TSqrExecInt64.Create();

    SQL_FLOAT:
      result := TSqrExecFloat.Create();
    SQL_DOUBLE, SQL_D_FLOAT:
      result := TSqrExecDouble.Create();
  else
    result := TSqrExecInt64.Create();
  end;

end;

procedure TSqrFunctionFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TSqrFunction }

procedure TSqrFunction<TIn, TOut>.dispose;
begin
  Destroy;
end;

procedure TSqrFunction<TIn, TOut>.execute(AStatus: IStatus;
  AContext: IExternalContext; AInMsg, AOutMsg: Pointer);
var
  xInput: PInput;
  xOutput: POutput;
begin
  xInput := PInput(AInMsg);
  xOutput := POutput(AOutMsg);
  xOutput.resultNull := True;
  if (not xInput.n1Null) then
  begin
    xOutput.resultNull := False;
```

```
    xOutput.result := Self.sqrExec(xInput.n1);
  end;
end;

procedure TSqrFunction<TIn, TOut>.getCharSet(AStatus: IStatus;
  AContext: IExternalContext; AName: PAnsiChar; ANameSize: Cardinal);
begin
end;


{ TSqrtExecSmallint }

function TSqrExecSmallint.sqrExec(AIn: Smallint): Integer;
begin
  Result := AIn * AIn;
end;

{ TSqrExecInteger }

function TSqrExecInteger.sqrExec(AIn: Integer): Int64;
begin
  Result := AIn * AIn;
end;

{ TSqrExecInt64 }

function TSqrExecInt64.sqrExec(AIn: Int64): Int64;
begin
  Result := AIn * AIn;
end;

{ TSqrExecFloat }

function TSqrExecFloat.sqrExec(AIn: Single): Double;
begin
  Result := AIn * AIn;
end;

{ TSqrExecDouble }

function TSqrExecDouble.sqrExec(AIn: Double): Double;
begin
  Result := AIn * AIn;
end;

.................
```

Метод setup

Метод `setup` позволяет изменить типы входных параметров и выходных переменных для внешних процедур и функций или полей для триггеров. Для этого используется интерфейс `IMetadataBuilder`, который позволяет построить входные и выходные сообщения с заданными типами, размерностью и набором символов. Входные сообщения будут

перестроены в формат заданный в методе `setup`, а выходные перестроены из формата заданного в методе `setup` в формат сообщения определенного в DLL процедуры, функции или триггера. Типы полей или параметров должны быть совместимы для преобразования.

Данный метод позволяет упростить создание обобщённых для разных типов параметров процедур и функций путём их приведения в наиболее общий тип. Более сложный и полезный пример будет рассмотрен позже, а пока немного изменим уже существующий пример внешней функции SumArgs.

Наша функция будет работать с сообщениями, которые описываются следующей структурой

```
type
  // структура на которое будет отображено входное сообщение
  TSumArgsInMsg = record
    n1: Integer;
    n1Null: WordBool;
    n2: Integer;
    n2Null: WordBool;
    n3: Integer;
    n3Null: WordBool;
  end;

  PSumArgsInMsg = ^TSumArgsInMsg;

  // структура на которое будет отображено выходное сообщение
  TSumArgsOutMsg = record
    result: Integer;
    resultNull: WordBool;
  end;

  PSumArgsOutMsg = ^TSumArgsOutMsg;
```

Теперь создадим фабрику функций, в методе setup которой зададим формат сообщений, которые соответствуют выше приведённым структурам.

```
{ TSumArgsFunctionFactory }

procedure TSumArgsFunctionFactory.dispose;
begin
  Destroy;
end;

function TSumArgsFunctionFactory.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := TSumArgsFunction.Create();
end;

procedure TSumArgsFunctionFactory.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
```

```
begin
  // строим сообщение для входных параметров
  AInBuilder.setType(AStatus, 0, SQL_LONG + 1);
  AInBuilder.setLength(AStatus, 0, sizeof(Int32));
  AInBuilder.setType(AStatus, 1, SQL_LONG + 1);
  AInBuilder.setLength(AStatus, 1, sizeof(Int32));
  AInBuilder.setType(AStatus, 2, SQL_LONG + 1);
  AInBuilder.setLength(AStatus, 2, sizeof(Int32));
  // строим сообщение для выходных параметров
  AOutBuilder.setType(AStatus, 0, SQL_LONG + 1);
  AOutBuilder.setLength(AStatus, 0, sizeof(Int32));
end;
```

Реализация функции тривиальна

```
procedure TSumArgsFunction.execute(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer);
var
  xInput: PSumArgsInMsg;
  xOutput: PSumArgsOutMsg;
begin
  // преобразовываем указатели на вход и выход к типизированным
  xInput := PSumArgsInMsg(AInMsg);
  xOutput := PSumArgsOutMsg(AOutMsg);
  // по умолчанию выходной аргемент = NULL, а потому выставляем ему nullFlag
  xOutput^.resultNull := True;
  // если один из аргументов NULL значит и результат NULL
  // в противном случае считаем сумму аргументов
  with xInput^ do
  begin
    if not(n1Null or n2Null or n3Null) then
    begin
      xOutput^.result := n1 + n2 + n3;
      // раз есть результат, то сбрасываем NULL флаг
      xOutput^.resultNull := False;
    end;
  end;
end;
```

Теперь даже если мы объявим функции следующим образом, она всё равно сохранит свою работоспособность, поскольку входные и выходные сообщения будут преобразованы к тому формату. что мы задали в методе setup.

```
create or alter function FN_SUM_ARGS (
    N1 varchar(15),
    N2 varchar(15),
    N3 varchar(15))
returns varchar(15)
EXTERNAL NAME 'MyUdrSetup!sum_args'
ENGINE UDR;
```

Вы можете проверить вышеприведённое утверждение выполнив следующий запрос

```
select FN_SUM_ARGS('15', '21', '35') from rdb$database
```

Обобщённые фабрики

В процессе разработки UDR необходимо под каждую внешнюю процедуру, функцию или триггер писать свою фабрику создающую экземпляр это UDR. Эту задачу можно упростить написав обобщённые фабрики с помощью так называемых дженериков. Они доступны начиная с Delphi 2009, в Free Pascal начиная с версии FPC 2.2.

> **Примечание**
>
> В Free Pascal синтаксис создания обобщённых типов отличается от Delphi. Начиная с версии FPC 2.6.0 декларируется совместимый с Delphi синтаксис.

Рассмотрим два основных случая для которых будут написаны обобщённые фабрики:

- экземплярам внешних процедур, функций и триггеров не требуются какие либо сведения о метаданных, не нужны специальные действия в логике создания экземпляров UDR, для работы с сообщениями используются фиксированные структуры;
- экземплярам внешних процедур, функций и триггеров требуются сведения о метаданных, не нужны специальные действия в логике создания экземпляров UDR, для работы с сообщениями используются экземпляры интерфейсов `IMessageMetadata`.

В первом случае достаточно просто создавать нужный экземпляр класса в методе newItem без дополнительных действий. для этого воспользуемся ограничением конструктора в классах потомках классов `IUdrFunctionFactoryImpl`, `IUdrProcedureFactoryImpl`, `IUdrTriggerFactoryImpl`. Объявления таких фабрик выглядит следующим образом:

```
unit UdrFactories;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses SysUtils, Firebird;

type

  // Простая фабрика внешних функций
```

```
  TFunctionSimpleFactory<T: IExternalFunctionImpl, constructor> = class
    (IUdrFunctionFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;

  // Простая фабрика внешних процедур
  TProcedureSimpleFactory<T: IExternalProcedureImpl, constructor> = class
    (IUdrProcedureFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalProcedure; override;
  end;

  // Простая фабрика внешних триггеров
  TTriggerSimpleFactory<T: IExternalTriggerImpl, constructor> = class
    (IUdrTriggerFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalTrigger; override;
  end;
```

В секции реализации тело метода `setup` можно оставить пустым, в них ничего не делается, в теле метода `dispose` просто вызвать деструктор. А в теле метода `newItem` необходимо просто вызвать конструктор по умолчанию для подстановочного типа `T`.

```
implementation

{ TProcedureSimpleFactory<T> }

procedure TProcedureSimpleFactory<T>.dispose;
begin
  Destroy;
end;

function TProcedureSimpleFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalProcedure;
begin
```

```
  Result := T.Create;
end;

procedure TProcedureSimpleFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;


{ TFunctionFactory<T> }

procedure TFunctionSimpleFactory<T>.dispose;
begin
  Destroy;
end;

function TFunctionSimpleFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := T.Create;
end;

procedure TFunctionSimpleFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TTriggerSimpleFactory<T> }

procedure TTriggerSimpleFactory<T>.dispose;
begin
  Destroy;
end;

function TTriggerSimpleFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalTrigger;
begin
  Result := T.Create;
end;

procedure TTriggerSimpleFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AFieldsBuilder: IMetadataBuilder);
begin

end;
```

Теперь для случая 1 можно не писать фабрики под каждую процедуру, функцию или триггер. Вместо этого регистрировать их с помощью обобщённых фабрик следующим образом:

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // регистрируем нашу функцию
  AUdrPlugin.registerFunction(AStatus, 'sum_args',
    TFunctionSimpleFactory<TSumArgsFunction>.Create());
  // регистрируем нашу процедуру
  AUdrPlugin.registerProcedure(AStatus, 'gen_rows',
    TProcedureSimpleFactory<TGenRowsProcedure>.Create());
  // регистрируем наш триггер
  AUdrPlugin.registerTrigger(AStatus, 'test_trigger',
    TTriggerSimpleFactory<TMyTrigger>.Create());

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Второй случай более сложный. По умолчанию сведения о метаданных не передаются в экземпляры процедур, функций и триггеров. Однако метаданных передаются в качестве параметра в методе `newItem` фабрик. Метаданные UDR имеют тип `IRoutineMetadata`, жизненный цикл которого контролируется самим движком Firebird, поэтому его можно смело передавать в экземпляры UDR. Из него можно получить экземпляры интерфейсов для входного и выходного сообщения, метаданные и тип триггера, имя UDR, пакета, точки входа и тело UDR. Сами классы для реализаций внешних процедур, функций и триггеров не имеют полей для хранения метаданных, поэтому нам придётся сделать своих наследников.

```
unit UdrFactories;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses SysUtils, Firebird;

type
...

  // Внешняя функция с метаданными
  TExternalFunction = class(IExternalFunctionImpl)
    Metadata: IRoutineMetadata;
  end;

  // Внешняя процедура с метаданными
  TExternalProcedure = class(IExternalProcedureImpl)
    Metadata: IRoutineMetadata;
  end;

  // Внешний триггер с метаданными
  TExternalTrigger = class(IExternalTriggerImpl)
    Metadata: IRoutineMetadata;
  end;
```

В этом случае ваши собственные хранимые процедуры, функции и триггеры должны быть унаследованы от новых классов с метаданными.

Теперь объявим фабрики которые будут создавать UDR и инициализировать метаданные.

```
unit UdrFactories;

{$IFDEF FPC}
{$MODE DELPHI}{$H+}
{$ENDIF}

interface

uses SysUtils, Firebird;

type
...

  // Фабрика внешних функций с метаданными
  TFunctionFactory<T: TExternalFunction, constructor> = class
    (IUdrFunctionFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;

  // Фабрика внешних процедур с метаданными
  TProcedureFactory<T: TExternalProcedure, constructor> = class
    (IUdrProcedureFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalProcedure; override;
  end;

  // Фабрика внешних триггеров с метаданными
  TTriggerFactory<T: TExternalTrigger, constructor> = class
    (IUdrTriggerFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalTrigger; override;
```

```
  end;
```

Реализация метода `newItem` тривиальна и похожа на первый случай, за исключением того, что необходимо инициализировать поле с метаданными.

```
implementation
...

{ TFunctionFactory<T> }

procedure TFunctionFactory<T>.dispose;
begin
  Destroy;
end;

function TFunctionFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalFunction;
begin
  Result := T.Create;
  (Result as T).Metadata := AMetadata;
end;

procedure TFunctionFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TProcedureFactory<T> }

procedure TProcedureFactory<T>.dispose;
begin
  Destroy;
end;

function TProcedureFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalProcedure;
begin
  Result := T.Create;
  (Result as T).Metadata := AMetadata;
end;

procedure TProcedureFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata;
  AInBuilder, AOutBuilder: IMetadataBuilder);
begin

end;

{ TTriggerFactory<T> }

procedure TTriggerFactory<T>.dispose;
```

```
begin
  Destroy;
end;

function TTriggerFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalTrigger;
begin
  Result := T.Create;
  (Result as T).Metadata := AMetadata;
end;

procedure TTriggerFactory<T>.setup(AStatus: IStatus; AContext: IExternalContext;
  AMetadata: IRoutineMetadata; AFieldsBuilder: IMetadataBuilder);
begin

end;
```

Готовый модуль с обобщёнными фабриками можно скачать по адресу https://github.com/sim1984/udr-book/blob/master/examples/Common/UdrFactories.pas.

Глава 6

Работа с типом BLOB

В отличие от других типов данных BLOB передаются по ссылке (идентификатор BLOB), а не по значению. Это логично, BLOB могут быть огромных размеров, а потому поместить их в буфер фиксированный ширины невозможно. Вместо этого в буфер сообщений помещается так называемый BLOB идентификатор. а работа с данными типа BLOB осуществляются через интерфейс `IBlob`.

Ещё одной важной особенностью типа BLOB является то, что BLOB является не изменяемым типом, вы не можете изменить содержимое BLOB с заданным идентификатором, вместо этого нужно создать BLOB с новым содержимым и идентификатором.

Поскольку размер данных типа BLOB может быть очень большим, то данные BLOB читаются и пишутся порциями (сегментами), максимальный размер сегмента равен 64 Кб. Чтение сегмента осуществляется методом `getSegment` интерфейса `IBlob`. Запись сегмента осуществляется методом `putSegment` интерфейса `IBlob`.

Чтение данных из BLOB

В качестве примера чтения BLOB рассмотрим процедуру которая разбивает строку по разделителю (обратная процедура для встроенной агрегатной функции LIST). Она объявлена следующим образом

```
create procedure split (
    txt blob sub_type text character set utf8,
    delimiter char(1) character set utf8 = ','
) returns (
    id integer
)
external name 'myudr!split'
engine udr;
```

Зарегистрируем фабрику нашей процедуры:

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // регистрируем нашу процедуру
  AUdrPlugin.registerProcedure(AStatus, 'split', TProcedureSimpleFactory<TSplitProcedure>.C

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Здесь я применил обобщённую фабрику процедур для простых случаев, когда фабрика просто создаёт экземпляр процедуры без использования метаданных. Такая фабрика объявлена следующим образом:

```
...
interface

uses SysUtils, Firebird;

type

  TProcedureSimpleFactory<T: IExternalProcedureImpl, constructor> =
  class(IUdrProcedureFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalProcedure; override;
  end;

...

implementation

{ TProcedureSimpleFactory<T> }

procedure TProcedureSimpleFactory<T>.dispose;
begin
  Destroy;
end;

function TProcedureSimpleFactory<T>.newItem(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata): IExternalProcedure;
begin
  Result := T.Create;
end;

procedure TProcedureSimpleFactory<T>.setup(AStatus: IStatus;
  AContext: IExternalContext; AMetadata: IRoutineMetadata; AInBuilder,
  AOutBuilder: IMetadataBuilder);
begin

...
```

Теперь перейдём к реализации процедуры. Сначала объявим структуры для входного и выходного сообщения.

```
  TInput = record
    txt: ISC_QUAD;
    txtNull: WordBool;
    delimiter: array [0 .. 3] of AnsiChar;
    delimiterNull: WordBool;
  end;

  TInputPtr = ^TInput;

  TOutput = record
    Id: Integer;
    Null: WordBool;
  end;

  TOutputPtr = ^TOutput;
```

Как видите вместо значения BLOB передаётся идентификатор BLOB, который описывается структурой ISC_QUAD.

Теперь опишем класс процедуры и возвращаемого набора данных:

```
  TSplitProcedure = class(IExternalProcedureImpl)
  private
    procedure SaveBlobToStream(AStatus: IStatus; AContext: IExternalContext;
      ABlobId: ISC_QUADPtr; AStream: TStream);
    function readBlob(AStatus: IStatus; AContext: IExternalContext;
      ABlobId: ISC_QUADPtr): string;
  public
    // Вызывается при уничтожении экземпляра процедуры
    procedure dispose(); override;

    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    function open(AStatus: IStatus; AContext: IExternalContext; AInMsg: Pointer;
      AOutMsg: Pointer): IExternalResultSet; override;
  end;

  TSplitResultSet = class(IExternalResultSetImpl)
{$IFDEF FPC}
    OutputArray: TStringArray;
{$ELSE}
    OutputArray: TArray<string>;
{$ENDIF}
    Counter: Integer;
    Output: TOutputPtr;

    procedure dispose(); override;
    function fetch(AStatus: IStatus): Boolean; override;
  end;
```

Дополнительные методы `SaveBlobToStream` и `readBlob` предназначены для чтения BLOB. Первая читает BLOB в поток, вторая — основана на первой и выполняет преобразование прочтённого потока в строку Delphi. В набор данных передаётся массив строк OutputArray и счётчик возвращённых записей Counter.

В методе open читается BLOB и преобразуется в строку. Полученная строка разбивается по разделителю с помощью встроенного метода Split из хелпера для строк. Полученный массив строк передаётся в результирующий набор данных.

```
function TSplitProcedure.open(AStatus: IStatus; AContext: IExternalContext;
  AInMsg, AOutMsg: Pointer): IExternalResultSet;
var
  xInput: TInputPtr;
  xText: string;
  xDelimiter: string;
begin
  xInput := AInMsg;
  if xInput.txtNull or xInput.delimiterNull then
  begin
    Result := nil;
    Exit;
  end;

  xText := readBlob(AStatus, AContext, @xInput.txt);
  xDelimiter := TFBCharSet.CS_UTF8.GetString(TBytes(@xInput.delimiter), 0, 4);
  // автоматически не правильно определяется потому что строки
  // не завершены нулём
  // ставим кол-во байт/4
  SetLength(xDelimiter, 1);

  Result := TSplitResultSet.Create;
  with TSplitResultSet(Result) do
  begin
    Output := AOutMsg;
    OutputArray := xText.Split([xDelimiter], TStringSplitOptions.ExcludeEmpty);
    Counter := 0;
  end;
end;
```

> **Примечание**
>
> Тип перечисление TFBCharSet не входит в Firebird.pas. Он написан мною для облегчения работы с кодировками Firebird. В данном случае считаем что все наши строки приходят в кодировке UTF-8.

Теперь опишем процедуру чтения данных из BLOB в поток. Для того чтобы прочитать данные из BLOB необходимо его открыть. Это можно сделать вызвав метод `openBlob` интерфейса `IAttachment`. Поскольку мы читаем BLOB из своей базы данных, то будем открывать его в контексте текущего подключения. Контекст текущего подключения и контекст текущей транзакции мы можем получить из контекста внешней процедуры, функции или триггера (интерфейс `IExternalContext`).

BLOB читается порциями (сегментами), максимальный размер сегмента равен 64 Кб. Чтение сегмента осуществляется методом `getSegment` интерфейса `IBlob`.

```
procedure TSplitProcedure.SaveBlobToStream(AStatus: IStatus;
  AContext: IExternalContext; ABlobId: ISC_QUADPtr; AStream: TStream);
var
  att: IAttachment;
  trx: ITransaction;
  blob: IBlob;
  buffer: array [0 .. 32767] of AnsiChar;
  l: Integer;
begin
  try
    att := AContext.getAttachment(AStatus);
    trx := AContext.getTransaction(AStatus);
    blob := att.openBlob(AStatus, trx, ABlobId, 0, nil);
    while True do
    begin
      case blob.getSegment(AStatus, SizeOf(buffer), @buffer, @l) of
        IStatus.RESULT_OK:
          AStream.WriteBuffer(buffer, l);
        IStatus.RESULT_SEGMENT:
          AStream.WriteBuffer(buffer, l);
      else
        break;
      end;
    end;
    AStream.Position := 0;
    blob.close(AStatus);
  finally
    if Assigned(att) then
      att.release;
    if Assigned(trx) then
      trx.release;
    if Assigned(blob) then
      blob.release;
  end;
end;
```

Примечание

Обратите внимание, интерфейсы `IAttachment`, `ITransaction` и `IBlob` наследуют интерфейс `IReferenceCounted`, а значит это объекты с подсчётом ссылок. Методы возвращающие объекты этих интерфейсов устанавливают счётчик ссылок в 1. По завершению работы с такими объектами нужно уменьшить счётчик ссылок с помощью метода `release`.

На основе метода `SaveBlobToStream` написана процедура чтения BLOB в строку:

```
function TSplitProcedure.readBlob(AStatus: IStatus; AContext: IExternalContext;
  ABlobId: ISC_QUADPtr): string;
var
```

```
{$IFDEF FPC}
  xStream: TBytesStream;
{$ELSE}
  xStream: TStringStream;
{$ENDIF}
begin
{$IFDEF FPC}
  xStream := TBytesStream.Create(nil);
{$ELSE}
  xStream := TStringStream.Create('', 65001);
{$ENDIF}
  try
    SaveBlobToStream(AStatus, AContext, ABlobId, xStream);
{$IFDEF FPC}
    Result := TEncoding.UTF8.GetString(xStream.Bytes, 0, xStream.Size);
{$ELSE}
    Result := xStream.DataString;
{$ENDIF}
  finally
    xStream.Free;
  end;
end;
```

Примечание

К сожалению Free Pascal не обеспечивает полную обратную совместимость с Delphi для класса `TStringStream`. В версии для FPC нельзя указать кодировку с которой будет работать поток, а потому приходится обрабатывать для него преобразование в строку особым образом.

Метод `fetch` выходного набора данных извлекает из массива строк элемент с индексом Counter и увеличивает его до тех пор, пока не будет извлечён последний элемент массива. Каждая извлечённая строка преобразуется к целому. Если это невозможно сделать то будет возбуждено исключение с кодом `isc_convert_error`.

```
procedure TSplitResultSet.dispose;
begin
  SetLength(OutputArray, 0);
  Destroy;
end;

function TSplitResultSet.fetch(AStatus: IStatus): Boolean;
var
  statusVector: array [0 .. 4] of NativeIntPtr;
begin
  if Counter <= High(OutputArray) then
  begin
    Output.Null := False;
    // исключение будут перехвачены в любом случае с кодом isc_random
    // здесь же мы будем выбрасывать стандартную для Firebird
    // ошибку isc_convert_error
    try
      Output.Id := OutputArray[Counter].ToInteger();
    except
      on e: EConvertError do
```

```
      begin

        statusVector[0] := NativeIntPtr(isc_arg_gds);
        statusVector[1] := NativeIntPtr(isc_convert_error);
        statusVector[2] := NativeIntPtr(isc_arg_string);
        statusVector[3] := NativeIntPtr(PAnsiChar('Cannot convert string to integer'));
        statusVector[4] := NativeIntPtr(isc_arg_end);

        AStatus.setErrors(@statusVector);
      end;
    end;
    inc(Counter);
    Result := True;
  end
  else
    Result := False;
end;
```

Примечание

На самом деле обработка любых ошибок кроме isc_random не очень удобна, для упрощения можно написать свою обёртку.

Работоспособность процедуры можно проверить следующим образом:

```
SELECT ids.ID
FROM SPLIT((SELECT LIST(ID) FROM MYTABLE), ',') ids
```

Примечание

Главным недостатком такой реализации состоит в том, что BLOB будет всегда прочитан целиком, даже если вы хотите досрочно прервать извлечение записей из процедуры. При желании вы можете изменить код процедуры таким образом, чтобы разбиение на подстроки осуществлялось более маленькими порциями. Для этого чтение этих порций необходимо осуществлять в методе `fetch` по мере извлечения строк результата.

Запись данных в BLOB

В качестве примера записи BLOB рассмотрим функцию читающую содержимое BLOB из файла.

Примечание

Этот пример является адаптированной версией UDF функций для чтения и записи BLOB из/в файл. Оригинальная UDF доступна по адресу blobsaveload.zip

Утилиты для чтения и записи BLOB из/в файл оформлены в виде пакета

```
CREATE PACKAGE BlobFileUtils
```

```
AS
BEGIN
  PROCEDURE SaveBlobToFile(ABlob BLOB, AFileName VARCHAR(255) CHARACTER SET UTF8);

  FUNCTION LoadBlobFromFile(AFileName VARCHAR(255) CHARACTER SET UTF8) RETURNS BLOB;
END^

CREATE PACKAGE BODY BlobFileUtils
AS
BEGIN
  PROCEDURE SaveBlobToFile(ABlob BLOB, AFileName VARCHAR(255) CHARACTER SET UTF8)
  EXTERNAL NAME 'BlobFileUtils!SaveBlobToFile'
  ENGINE UDR;

  FUNCTION LoadBlobFromFile(AFileName VARCHAR(255) CHARACTER SET UTF8) RETURNS BLOB
  EXTERNAL NAME 'BlobFileUtils!LoadBlobFromFile'
  ENGINE UDR;
END^
```

Зарегистрируем фабрики наших процедур и функций:

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // регистрируем
  AUdrPlugin.registerProcedure(AStatus, 'SaveBlobToFile', TSaveBlobToFileProcFactory.Create
  AUdrPlugin.registerFunction(AStatus, 'LoadBlobFromFile', TLoadBlobFromFileFuncFactory.Cre

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

В данном случае приведём пример только для функции считывающий BLOB из файла, полный пример UDR вы можете скачать по адресу 06.BlobSaveLoad. Интерфейсная часть модуля с описанием функции LoadBlobFromFile выглядит следующим образом:

```
interface

uses
  Firebird, Classes, SysUtils;

type

  // входное сообщений функции
  TInput = record
    filename: record
      len: Smallint;
      str: array [0 .. 1019] of AnsiChar;
    end;
```

```
    filenameNull: WordBool;
  end;
  TInputPtr = ^TInput;

  // выходное сообщение функции
  TOutput = record
    blobData: ISC_QUAD;
    blobDataNull: WordBool;
  end;
  TOutputPtr = ^TOutput;

  // реализация функции LoadBlobFromFile
  TLoadBlobFromFileFunc = class(IExternalFunctionImpl)
  public
    procedure dispose(); override;

    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;

  // Фабрика для создания экземпляра внешней функции LoadBlobFromFile
  TLoadBlobFromFileFuncFactory = class(IUdrFunctionFactoryImpl)
    procedure dispose(); override;

    procedure setup(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata; AInBuilder: IMetadataBuilder;
      AOutBuilder: IMetadataBuilder); override;

    function newItem(AStatus: IStatus; AContext: IExternalContext;
      AMetadata: IRoutineMetadata): IExternalFunction; override;
  end;
```

Приведём только реализацию основного метода `execute` класса `TLoadBlobFromFile`, остальные методы классов элементарны.

```
procedure TLoadBlobFromFileFunc.execute(AStatus: IStatus;
  AContext: IExternalContext; AInMsg: Pointer; AOutMsg: Pointer);
const
  MaxBufSize = 16384;
var
  xInput: TInputPtr;
  xOutput: TOutputPtr;
  xFileName: string;
  xStream: TFileStream;
  att: IAttachment;
  trx: ITransaction;
  blob: IBlob;
  buffer: array [0 .. 32767] of Byte;
  xStreamSize: Integer;
  xBufferSize: Integer;
```

```
  xReadLength: Integer;
begin
  xInput := AInMsg;
  xOutput := AOutMsg;
  if xInput.filenameNull then
  begin
    xOutput.blobDataNull := True;
    Exit;
  end;
  xOutput.blobDataNull := False;
  // получаем имя файла
  xFileName := TEncoding.UTF8.GetString(TBytes(@xInput.filename.str), 0,
    xInput.filename.len * 4);
  SetLength(xFileName, xInput.filename.len);
  // читаем файл в поток
  xStream := TFileStream.Create(xFileName, fmOpenRead or fmShareDenyNone);
  att := AContext.getAttachment(AStatus);
  trx := AContext.getTransaction(AStatus);
  blob := nil;
  try
    xStreamSize := xStream.Size;
    // определём максимальный размер буфера (сегмента)
    if xStreamSize > MaxBufSize then
      xBufferSize := MaxBufSize
    else
      xBufferSize := xStreamSize;
    // создаём новый blob
    blob := att.createBlob(AStatus, trx, @xOutput.blobData, 0, nil);
    // читаем содержимое потока и пишем его в BLOB по сегментно
    while xStreamSize <> 0 do
    begin
      if xStreamSize > xBufferSize then
        xReadLength := xBufferSize
      else
        xReadLength := xStreamSize;
      xStream.ReadBuffer(buffer, xReadLength);

      blob.putSegment(AStatus, xReadLength, @buffer[0]);

      Dec(xStreamSize, xReadLength);
    end;
    // закрываем BLOB
    blob.close(AStatus);
  finally
    if Assigned(blob) then
      blob.release;
    att.release;
    trx.release;
    xStream.Free;
  end;
end;
```

Первым делом необходимо создать новый BLOB и привязать его в blobId выхода с помощью метода `createBlob` интерфейса `IAttachment`. Поскольку мы пишем пусть и временный BLOB для своей базы данных, то будем создавать его в контексте текущего подключения. Контекст

текущего подключения и контекст текущей транзакции мы можем получить из контекста внешней процедуры, функции или триггера (интерфейс `IExternalContext`).

Так же как и в случае с чтением данных из BLOB, запись ведётся по сегментно с помощью метода `putSegment` интерфейса `IBlob` до тех пор, пока данные в потоке файла не закончатся. По завершению записи данных в блоб необходимо закрыть его с помощью метода `close`.

Хелпер для работы с типом BLOB

В выше описанных примерах мы использовали сохранение содержимого BLOB в поток, а также загрузку содержимого BLOB в поток. Это довольно частая операция при работе с типом BLOB, поэтому было бы хорошо написать специальный набор утилит для повторного использования кода.

Современные версии Delphi и Free Pascal позволяют расширять существующие классы и типы без наследования с помощью так называемых хэлперов. Добавим методы в интерфейс `IBlob` для сохранения и загрузки содержимого потока из/в Blob.

Создадим специальный модуль FbBlob, где будет размещён наш хэлпер.

```
unit FbBlob;

interface

uses Classes, SysUtils, Firebird;

const
  MAX_SEGMENT_SIZE = $7FFF;

type
  TFbBlobHelper = class helper for IBlob
    { Загружает в BLOB содержимое потока

      @param(AStatus Статус вектор)
      @param(AStream Поток)
    }
    procedure LoadFromStream(AStatus: IStatus; AStream: TStream);
    { Загружает в поток содержимое BLOB

      @param(AStatus Статус вектор)
      @param(AStream Поток)
    }
    procedure SaveToStream(AStatus: IStatus; AStream: TStream);
  end;

implementation

uses Math;

procedure TFbBlobHelper.LoadFromStream(AStatus: IStatus; AStream: TStream);
var
  xStreamSize: Integer;
```

```
  xReadLength: Integer;
  xBuffer: array [0 .. MAX_SEGMENT_SIZE] of Byte;
begin
  xStreamSize := AStream.Size;
  AStream.Position := 0;
  while xStreamSize <> 0 do
  begin
    xReadLength := Min(xStreamSize, MAX_SEGMENT_SIZE);
    AStream.ReadBuffer(xBuffer, xReadLength);
    Self.putSegment(AStatus, xReadLength, @xBuffer[0]);
    Dec(xStreamSize, xReadLength);
  end;
end;

procedure TFbBlobHelper.SaveToStream(AStatus: IStatus; AStream: TStream);
var
  xInfo: TFbBlobInfo;
  Buffer: array [0 .. MAX_SEGMENT_SIZE] of Byte;
  xBytesRead: Cardinal;
  xBufferSize: Cardinal;
begin
  AStream.Position := 0;
  xBufferSize := Min(SizeOf(Buffer), MAX_SEGMENT_SIZE);
  while True do
  begin
    case Self.getSegment(AStatus, xBufferSize, @Buffer[0], @xBytesRead) of
      IStatus.RESULT_OK:
        AStream.WriteBuffer(Buffer, xBytesRead);
      IStatus.RESULT_SEGMENT:
        AStream.WriteBuffer(Buffer, xBytesRead);
    else
      break;
    end;
  end;
end;

end.
```

Теперь вы можете значительно упростить операции с типом BLOB, например вышеприведенная функция сохранения BLOB в файл может быть переписана так:

```
procedure TLoadBlobFromFileFunc.execute(AStatus: IStatus;
  AContext: IExternalContext; AInMsg: Pointer; AOutMsg: Pointer);
var
  xInput: TInputPtr;
  xOutput: TOutputPtr;
  xFileName: string;
  xStream: TFileStream;
  att: IAttachment;
  trx: ITransaction;
  blob: IBlob;
begin
  xInput := AInMsg;
  xOutput := AOutMsg;
```

```
  if xInput.filenameNull then
  begin
    xOutput.blobDataNull := True;
    Exit;
  end;
  xOutput.blobDataNull := False;
  // получаем имя файла
  xFileName := TEncoding.UTF8.GetString(TBytes(@xInput.filename.str), 0,
    xInput.filename.len * 4);
  SetLength(xFileName, xInput.filename.len);
  // читаем файл в поток
  xStream := TFileStream.Create(xFileName, fmOpenRead or fmShareDenyNone);
  att := AContext.getAttachment(AStatus);
  trx := AContext.getTransaction(AStatus);
  blob := nil;
  try
    // создаём новый blob
    blob := att.createBlob(AStatus, trx, @xOutput.blobData, 0, nil);
    // загружаем содержимое потока в BLOB
    blob.LoadFromStream(AStatus, xStream);
    // закрываем BLOB
    blob.close(AStatus);
  finally
    if Assigned(blob) then
      blob.release;
    att.release;
    trx.release;
    xStream.Free;
  end;
end;
```

Глава 7

Контекст соединения и транзакции

Если ваша внешняя процедура, функция или триггер должна получать данные из собственной базы данных не через входные аргументы, а например через запрос, то вам потребуется получать контекст текущего соединения и/или транзакции. Кроме того, контекст соединения и транзакции необходим если вы будете работать с типом BLOB.

Контекст выполнения текущей процедуры, функции или триггера передаётся в качестве параметра с типом `IExternalContext` в метод `execute` триггера или функции, или в метод `open` процедуры. Интерфейс `IExternalContext` позволяет получить текущее соединение с помощью метода `getAttachment`, и текущую транзакцию с помощью метода `getTransaction`. Это даёт большую гибкость вашим UDR, например вы можете выполнять запросы к текущей базе данных с сохранением текущего сессионного окружения, в той же транзакции или в новой транзакции, созданной с помощью метода `startTransaction` интерфейса `IExternalContext`. В последнем случае запрос будет выполнен так как будто он выполняется в автономной транзакции. Кроме того, вы можете выполнить запрос к внешней базе данных с использованием транзакции присоединённой к текущей транзакции, т.е. транзакции с двухфазным подтверждением (2PC).

В качестве примера работы с контекстом выполнения функции напишем функцию, которая будет сериализовать результат выполнения SELECT запроса в формате JSON. Она объявлена следующим образом:

```
create function GetJson (
    sql_text blob sub_type text character set utf8,
    sql_dialect smallint not null default 3
) returns returns blob sub_type text character set utf8
external name 'JsonUtils!getJson'
engine udr;
```

Поскольку мы позволяем выполнять произвольный SQL запрос, то мы не знаем заранее формат выходных полей, и мы не сможем использовать структуру с фиксированными полями. В этом случае нам придётся работать с интерфейсом `IMessageMetadata`. Мы уже сталкивались с ним ранее, но на этот раз придётся работать с ним более основательно, поскольку мы должны обрабатывать все существующие типы Firebird.

> **Примечание**
>
> В JSON можно закодировать практически любые типы данных кроме бинарных. Для кодирования типов CHAR, VARCHAR с OCTETS NONE и BLOB SUB_TYPE BINARY будем кодировать бинарное содержимое с помощью кодирования base64, которое уже можно размещать в JSON.

Зарегистрируем фабрику нашей функции:

```
function firebird_udr_plugin(AStatus: IStatus; AUnloadFlagLocal: BooleanPtr;
  AUdrPlugin: IUdrPlugin): BooleanPtr; cdecl;
begin
  // регистрируем функцию
  AUdrPlugin.registerFunction(AStatus, 'getJson', TFunctionSimpleFactory<TJsonFunction>.Cre

  theirUnloadFlag := AUnloadFlagLocal;
  Result := @myUnloadFlag;
end;
```

Теперь объявим структуры для входного и выходного сообщения, а так же интерфейсную часть нашей функции:

```
unit JsonFunc;

{$IFDEF FPC}
{$MODE objfpc}{$H+}
{$DEFINE DEBUGFPC}
{$ENDIF}

interface

uses
  Firebird,
  UdrFactories,
  FbTypes,
  FbCharsets,
  SysUtils,
  System.NetEncoding,
  System.Json;

// ******************************************************
// create function GetJson (
//   sql_text blob sub_type text,
//   sql_dialect smallint not null default 3
// ) returns blob sub_type text character set utf8
// external name 'JsonUtils!getJson'
// engine udr;
// ******************************************************

type

  TInput = record
    SqlText: ISC_QUAD;
    SqlNull: WordBool;
    SqlDialect: Smallint;
    SqlDialectNull: WordBool;
  end;

  InputPtr = ^TInput;
```

```
  TOutput = record
    Json: ISC_QUAD;
    NullFlag: WordBool;
  end;

  OutputPtr = ^TOutput;

  // Внешняя функция TSumArgsFunction.
  TJsonFunction = class(IExternalFunctionImpl)
  public
    procedure dispose(); override;

    procedure getCharSet(AStatus: IStatus; AContext: IExternalContext;
      AName: PAnsiChar; ANameSize: Cardinal); override;

    { Преобразует целое в строку в соответствии с масштабом

      @param(AValue Значение)
      @param(Scale Масштаб)
      @returns(Строковое представление масштабированного целого)
    }
    function MakeScaleInteger(AValue: Int64; Scale: Smallint): string;

    { Добавляет закодированную запись в массив объектов Json

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AJson Массив объектов Json)
      @param(ABuffer Буфер записи)
      @param(AMeta Метаданные курсора)
      @param(AFormatSetting Установки формата даты и времени)
    }
    procedure writeJson(AStatus: IStatus; AContext: IExternalContext;
      AJson: TJsonArray; ABuffer: PByte; AMeta: IMessageMetadata;
      AFormatSettings: TFormatSettings);

    { Выполнение внешней функции

      @param(AStatus Статус вектор)
      @param(AContext Контекст выполнения внешней функции)
      @param(AInMsg Указатель на входное сообщение)
      @param(AOutMsg Указатель на выходное сообщение)
    }
    procedure execute(AStatus: IStatus; AContext: IExternalContext;
      AInMsg: Pointer; AOutMsg: Pointer); override;
  end;
```

Дополнительный метод `MakeScaleInteger` предназначен для преобразования масштабируемых чисел в строку, метод `writeJson` кодирует очередную запись выбранную из курсора в Json объект и добавляет его в массив таких объектов. Эти методы мы опишем позже, а пока приведём основной метод `execute` для выполнения внешней функции.

```
procedure TJsonFunction.execute(AStatus: IStatus; AContext: IExternalContext;
```

```
  AInMsg, AOutMsg: Pointer);
var
  xFormatSettings: TFormatSettings;
  xInput: InputPtr;
  xOutput: OutputPtr;
  att: IAttachment;
  tra: ITransaction;
  stmt: IStatement;
  inBlob, outBlob: IBlob;
  inStream: TBytesStream;
  outStream: TStringStream;
  cursorMetaData: IMessageMetadata;
  rs: IResultSet;
  msgLen: Cardinal;
  msg: Pointer;
  jsonArray: TJsonArray;
begin
  xInput := AInMsg;
  xOutput := AOutMsg;
  // если один из входных аргументов NULL, то и результат NULL
  if xInput.SqlNull or xInput.SqlDialectNull then
  begin
    xOutput.NullFlag := True;
    Exit;
  end;
  xOutput.NullFlag := False;
  // установки форматирования даты и времени
  xFormatSettings := TFormatSettings.Create;
  xFormatSettings.DateSeparator := '-';
  xFormatSettings.TimeSeparator := ':';
  // создаём поток байт для чтения blob
  inStream := TBytesStream.Create(nil);
  outStream := TStringStream.Create('', 65001);
  jsonArray := TJsonArray.Create;
  // получение текущего соединения и транзакции
  att := AContext.getAttachment(AStatus);
  tra := AContext.getTransaction(AStatus);
  stmt := nil;
  inBlob := nil;
  outBlob := nil;
  try
    // читаем BLOB в поток
    inBlob := att.openBlob(AStatus, tra, @xInput.SqlText, 0, nil);
    inBlob.SaveToStream(AStatus, inStream);
    inBlob.close(AStatus);
    // подготавливаем оператор
    stmt := att.prepare(AStatus, tra, inStream.Size, @inStream.Bytes[0],
      xInput.SqlDialect, IStatement.PREPARE_PREFETCH_METADATA);
    // получаем выходные метаданные курсора
    cursorMetaData := stmt.getOutputMetadata(AStatus);
    // открываем курсор
    rs := stmt.openCursor(AStatus, tra, nil, nil, nil, 0);
    // выделяем буфер нужного размера
    msgLen := cursorMetaData.getMessageLength(AStatus);
    msg := AllocMem(msgLen);
    try
      // читаем каждую запись курсора
      while rs.fetchNext(AStatus, msg) = IStatus.RESULT_OK do
```

```
      begin
        // и пишем её в JSON
        writeJson(AStatus, AContext, jsonArray, msg, cursorMetaData, xFormatSettings);
      end;
    finally
      // освобождаем буфер
      FreeMem(msg);
    end;
    // закрываем курсор
    rs.close(AStatus);
    // пишем JSON в поток
    outStream.WriteString(jsonArray.ToJSON);

    // пишем json в выходной blob
    outBlob := att.createBlob(AStatus, tra, @xOutput.Json, 0, nil);
    outBlob.LoadFromStream(AStatus, outStream);
    outBlob.close(AStatus);
  finally
    if Assigned(inBlob) then
      inBlob.release;
    if Assigned(stmt) then
      stmt.release;
    if Assigned(outBlob) then
      outBlob.release;
    tra.release;
    att.release;
    jsonArray.Free;
    inStream.Free;
    outStream.Free;
  end;
end;
```

Первым делом получаем из контекста выполнения функции текущее подключение и текущую транзакцию с помощью методов `getAttachment` и `getTransaction` интерфейса `IExternalContext`. Затем читаем содержимое BLOB для получения текста SQL запроса. Запрос подготавливается с помощью метода `prepare` интерфейса `IAttachment`. Пятым параметром передаётся SQL диалект полученный из входного параметра нашей функции. Шестым параметром передаём флаг `IStatement.PREPARE_PREFETCH_METADATA`, что обозначает что мы хотим получить метаданные курсора вместе с результатом препарирования запроса. Сами выходные метаданные курсора получаем с помощью метода `getOutputMetadata` интерфейса `IStatement`.

> **Примечание**
>
> На самом деле метод `getOutputMetadata` вернёт выходные метаданные в любом случае. Флаг `IStatement.PREPARE_PREFETCH_METADATA` заставит получить метаданные вместе с результатом подготовки запроса за один сетевой пакет. Поскольку мы выполняем запрос в рамках текущего соединение никакого сетевого обмена не будет, и это не принципиально.

Далее открываем курсор с помощью метода `openCursor` в рамках текущей транзакции (параметр 2). Получаем размер выходного буфера под результат курсора с помощью метода `getMessageLength` интерфейса `IMessageMetadata`. Это позволяет выделить память под буфер, которую мы освободим сразу после вычитки последней записи курсора.

Записи курсора читаются с помощью метода `fetchNext` интерфейса `IResultSet`. Этот метод заполняет буфер `msg` значениями полей курсора и возвращает `IStatus.RESULT_OK` до тех пор, пока записи курсора не кончатся. Каждая прочитанная запись передаётся в метод `writeJson`, который добавляет объект типа `TJsonObject` с сериализованной записью курсора в массив `TJsonArray`.

После завершения работы с курсором, закрываем его методом `close`, преобразуем массив Json объектов в строку, пишем её в выходной поток, который записываем в выходной Blob.

Теперь разберём метод `writeJson`. Объект `IUtil` потребуется нам для того, чтобы получать функции для декодирования даты и времени. В этом методе активно задействована работа с метаданными выходных полей курсора с помощью интерфейса `IMessageMetadata`. Первым дело создаём объект тип `TJsonObject` в который будем записывать значения полей текущей записи. В качестве имён ключей будем использовать алиасы полей из курсора. Если установлен NullFlag, то пишем значение null для ключа и переходим к следующему полю, в противном случае анализируем тип поля и пишем его значение в Json.

```
function TJsonFunction.MakeScaleInteger(AValue: Int64; Scale: Smallint): string;
var
  L: Integer;
begin
  Result := AValue.ToString;
  L := Result.Length;
  if (-Scale >= L) then
    Result := '0.' + Result.PadLeft(-Scale, '0')
  else
    Result := Result.Insert(Scale + L, '.');
end;

procedure TJsonFunction.writeJson(AStatus: IStatus; AContext: IExternalContext;
  AJson: TJsonArray; ABuffer: PByte; AMeta: IMessageMetadata;
  AFormatSettings: TFormatSettings);
var
  jsonObject: TJsonObject;
  i: Integer;
  FieldName: string;
  NullFlag: WordBool;
  fieldType: Cardinal;
  pData: PByte;
  util: IUtil;
  metaLength: Integer;
  // типы
  CharBuffer: array [0 .. 35766] of Byte;
  charLength: Smallint;
  charset: TFBCharSet;
  StringValue: string;
  SmallintValue: Smallint;
  IntegerValue: Integer;
  BigintValue: Int64;
  Scale: Smallint;
  SingleValue: Single;
  DoubleValue: Double;
  BooleanValue: Boolean;
  DateValue: ISC_DATE;
  TimeValue: ISC_TIME;
```

```
  TimestampValue: ISC_TIMESTAMP;
  DateTimeValue: TDateTime;
  year, month, day: Cardinal;
  hours, minutes, seconds, fractions: Cardinal;
  blobId: ISC_QUADPtr;
  BlobSubtype: Smallint;
  blob: IBlob;
  att: IAttachment;
  tra: ITransaction;
  textStream: TStringStream;
  binaryStream: TBytesStream;
begin
  // Получаем IUtil
  util := AContext.getMaster().getUtilInterface();
  // Создаём объект TJsonObject в которой будем
  // записывать значение полей записи
  jsonObject := TJsonObject.Create;
  for i := 0 to AMeta.getCount(AStatus) - 1 do
  begin
    // получаем алиас поля в запросе
    FieldName := AMeta.getAlias(AStatus, i);
    NullFlag := PWordBool(ABuffer + AMeta.getNullOffset(AStatus, i))^;
    if NullFlag then
    begin
      // если NULL пишем его в JSON и переходим к следующему полю
      jsonObject.AddPair(FieldName, TJsonNull.Create);
      continue;
    end;
    // получаем указатель на данные поля
    pData := ABuffer + AMeta.getOffset(AStatus, i);
    // аналог AMeta->getType(AStatus, i) & ~1
    fieldType := AMeta.getType(AStatus, i) and not 1;
    case fieldType of
      // VARCHAR
      SQL_VARYING:
        begin
          // размер буфера для VARCHAR
          metaLength := AMeta.getLength(AStatus, i);
          charset := TFBCharSet(AMeta.getCharSet(AStatus, i));
          // Для VARCHAR первые 2 байта - длина
          charLength := PSmallint(pData)^;
          // бинарные данные кодируем в base64
          if charset = CS_BINARY then
            StringValue := TNetEncoding.Base64.EncodeBytesToString((pData + 2),
              charLength)
          else
          begin
            // копируем данные в буфер начиная с 3 байта
            Move((pData + 2)^, CharBuffer, metaLength - 2);
            StringValue := charset.GetString(TBytes(@CharBuffer), 0,
              charLength * charset.GetCharWidth)
            SetLength(StringValue, charLength);
          end;
          jsonObject.AddPair(FieldName, StringValue);
        end;
      // CHAR
      SQL_TEXT:
        begin
```

```
        // размер буфера для CHAR
        metaLength := AMeta.getLength(AStatus, i);
        charset := TFBCharSet(AMeta.getCharSet(AStatus, i));
        // бинарные данные кодируем в base64
        if charset = CS_BINARY then
          StringValue := TNetEncoding.Base64.EncodeBytesToString((pData + 2),
            metaLength)
        else
        begin
          // копируем данные в буфер
          Move(pData^, CharBuffer, metaLength);
          StringValue := charset.GetString(TBytes(@CharBuffer), 0,
            metaLength);
          charLength := metaLength div charset.GetCharWidth;
          SetLength(StringValue, charLength);
        end;
        jsonObject.AddPair(FieldName, StringValue);
      end;
    // FLOAT
    SQL_FLOAT:
      begin
        SingleValue := PSingle(pData)^;
        jsonObject.AddPair(FieldName, TJSONNumber.Create(SingleValue));
      end;
    // DOUBLE PRECISION
    // DECIMAL(p, s), где p = 10..15 в 1 диалекте
    SQL_DOUBLE, SQL_D_FLOAT:
      begin
        DoubleValue := PDouble(pData)^;
        jsonObject.AddPair(FieldName, TJSONNumber.Create(DoubleValue));
      end;
    // INTEGER
    // NUMERIC(p, s), где p = 1..4
    SQL_SHORT:
      begin
        Scale := AMeta.getScale(AStatus, i);
        SmallintValue := PSmallint(pData)^;
        if (Scale = 0) then
        begin
          jsonObject.AddPair(FieldName, TJSONNumber.Create(SmallintValue));
        end
        else
        begin
          StringValue := MakeScaleInteger(SmallintValue, Scale);
          jsonObject.AddPair(FieldName, TJSONNumber.Create(StringValue));
        end;
      end;
    // INTEGER
    // NUMERIC(p, s), где p = 5..9
    // DECIMAL(p, s), где p = 1..9
    SQL_LONG:
      begin
        Scale := AMeta.getScale(AStatus, i);
        IntegerValue := PInteger(pData)^;
        if (Scale = 0) then
        begin
          jsonObject.AddPair(FieldName, TJSONNumber.Create(IntegerValue));
```

```
        end
        else
        begin
          StringValue := MakeScaleInteger(IntegerValue, Scale);
          jsonObject.AddPair(FieldName, TJSONNumber.Create(StringValue));
        end;
      end;
    // BIGINT
    // NUMERIC(p, s), где p = 10..18 в 3 диалекте
    // DECIMAL(p, s), где p = 10..18 в 3 диалекте
    SQL_INT64:
      begin
        Scale := AMeta.getScale(AStatus, i);
        BigintValue := Pint64(pData)^;
        if (Scale = 0) then
        begin
          jsonObject.AddPair(FieldName, TJSONNumber.Create(BigintValue));
        end
        else
        begin
          StringValue := MakeScaleInteger(BigintValue, Scale);
          jsonObject.AddPair(FieldName, TJSONNumber.Create(StringValue));
        end;
      end;
    // TIMESTAMP
    SQL_TIMESTAMP:
      begin
        TimestampValue := PISC_TIMESTAMP(pData)^;
        // получаем составные части даты-времени
        util.decodeDate(TimestampValue.date, @year, @month, @day);
        util.decodeTime(TimestampValue.time, @hours, @minutes, @seconds,
          @fractions);
        // получаем дату-время в родном типе Delphi
        DateTimeValue := EncodeDate(year, month, day) +
          EncodeTime(hours, minutes, seconds, fractions div 10);
        // форматируем дату-время по заданному формату
        StringValue := FormatDateTime('yyyy/mm/dd hh:nn:ss', DateTimeValue,
          AFormatSettings);
        jsonObject.AddPair(FieldName, StringValue);
      end;
    // DATE
    SQL_DATE:
      begin
        DateValue := PISC_DATE(pData)^;
        // получаем составные части даты
        util.decodeDate(DateValue, @year, @month, @day);
        // получаем дату в родном типе Delphi
        DateTimeValue := EncodeDate(year, month, day);
        // форматируем дату по заданному формату
        StringValue := FormatDateTime('yyyy/mm/dd', DateTimeValue,
          AFormatSettings);
        jsonObject.AddPair(FieldName, StringValue);
      end;
    // TIME
    SQL_TIME:
      begin
        TimeValue := PISC_TIME(pData)^;
        // получаем составные части времени
```

```
        util.decodeTime(TimeValue, @hours, @minutes, @seconds, @fractions);
        // получаем время в родном типе Delphi
        DateTimeValue := EncodeTime(hours, minutes, seconds,
          fractions div 10);
        // форматируем время по заданному формату
        StringValue := FormatDateTime('hh:nn:ss', DateTimeValue,
          AFormatSettings);
        jsonObject.AddPair(FieldName, StringValue);
      end;
    // BOOLEAN
    SQL_BOOLEAN:
      begin
        BooleanValue := PBoolean(pData)^;
        jsonObject.AddPair(FieldName, TJsonBool.Create(BooleanValue));
      end;
    // BLOB
    SQL_BLOB, SQL_QUAD:
      begin
        BlobSubtype := AMeta.getSubType(AStatus, i);
        blobId := ISC_QUADPtr(pData);
        att := AContext.getAttachment(AStatus);
        tra := AContext.getTransaction(AStatus);
        blob := att.openBlob(AStatus, tra, blobId, 0, nil);
        if BlobSubtype = 1 then
        begin
          // текст
          charset := TFBCharSet(AMeta.getCharSet(AStatus, i));
          // создаём поток с заданной кодировкой
          textStream := TStringStream.Create('', charset.GetCodePage);
          try
            blob.SaveToStream(AStatus, textStream);
            StringValue := textStream.DataString;
          finally
            textStream.Free;
            blob.release;
            tra.release;
            att.release
          end;
        end
        else
        begin
          // все остальные подтипы считаем бинарными
          binaryStream := TBytesStream.Create;
          try
            blob.SaveToStream(AStatus, binaryStream);
            // кодируем строку в base64
            StringValue := TNetEncoding.Base64.EncodeBytesToString
              (binaryStream.Memory, binaryStream.Size);
          finally
            binaryStream.Free;
            blob.release;
            tra.release;
            att.release
          end;
        end;
        jsonObject.AddPair(FieldName, StringValue);
      end;
```

```
    end;
  end;
  // добавление записи в формате Json в массив
  AJson.AddElement(jsonObject);
end;
```

Примечание

Перечисление типа TFbType отсутствует в стандартном модуле `Firebird.pas`. Однако использовать числовые значения не удобно, поэтому я написал специальный модуль FbTypes в котором разместил некоторые дополнительные типы и константы для удобства.

Перечисление TFBCharSet также отсутствует в модуле `Firebird.pas`. Я написал отдельный модуль FbCharsets в котором размещено это перечисление. Кроме того, для этого типа написан специальный хелпер, в котором размещены функции для получения названия набора символов, кодовой страницы, размера символа в байтах, получение класса `TEncoding` в нужной кодировки, а также функцию для преобразования массива байт в юникодную строку Delphi.

Для строк типа CHAR и VARCHAR проверяем кодировку, если это кодировка OCTETS, то кодируем строку алгоритмом base64, в противном случае преобразуем данные из буфера в строку Delphi. Обратите внимание, что для типа VARCHAR первые 2 байта содержат длину строки в символах.

Типы SMALLINT, INTEGER, BIGINT могут быть как обычными целыми числами, так масштабируемыми. Масштаб числа можно получить методом `getScale` интерфейса `IMessageMetadata`. Если масштаб не равен 0, то требуется специальная обработка числа, которая осуществляет методом `MakeScaleInteger`.

Типы DATE, TIME и TIMESTAMP декодируется на составные части даты и времени с помощью методов `decodeDate` и `decodeTime` интерфейса `IUtil`. Используем части даты и времени для получения даты-времени в стандартном Delphi типе TDateTime.

С типом BLOB работаем через потоки Delphi. Если BLOB бинарный, то создаём поток типа `TBytesStream`. Полученный массив байт кодируем с помощью алгоритма base64. Если BLOB текстовый, то используем специализированный поток `TStringStream` для строк, который позволяет учесть кодовую страницу. Кодовую страницу мы получаем из кодировки BLOB поля.

Алфавитный указатель