Firebird 3.0 OO API

3 мая 2017 — 0.55 Alpha

Спонсоры документации:
Platinum Sponsor

Gold Sponsor

Firebird 3.0 OO API

Над документом работали:

Александр Пешков

Редактор:

Денис Симонов

Содержание

Глава 1

Firebird interfaces

Примечание переводчика

Описываемые интерфейсы генерируются с помощью CLOOP на основе `include/firebird/FirebirdInterface.idl`.

Объектно-оринтированный API Firebird (далее OO API) базируется на использовании интерфейсов. Эти интерфейсы, хотя и рассматриваются с точки зрения сравнения с OLE2-интерфейсами (некоторые из них имеют методы `addRef()` и `release()`), являются нестандартными и имеют функции, отсутствующие в других широко используемых типах интерфейсов. Прежде всего, интерфейсы Firebird **не зависят от языка** — это означает, что для их определения/использования им не нужно использовать конструкции, специфичные для языка, такие как класс в C++, интерфейс может быть определен с использованием любого языка, имеющего понятие массива и указателя на процедуру/функцию. Кроме того, интерфейсы **версионны** — то есть мы поддерживаем разные версии одного и того же интерфейса. Бинарная компоновка интерфейсов предназначеная для поддержки этих функций очень эффективна (нет необходимости в дополнительных виртуальных вызовах как, например, в OLE2/COM с QueryInterface), но это не удобно для непосредственного использования в большинстве языков. Поэтому для упрощения использования API лучше использовать языково-ориентированные обертки для разных языков. В настоящее время у нас есть обертки для C++ и Pascal, скоро появится Java. Для конечного пользователя вызовы POV из C++ и Pascal нет никакой разницы, хотя в Pascal отсутствуют некоторые дополнительные языковые функции, присутствующие в C++ (например, возможность отключить автоматическую проверку статуса после вызовов API).

Обычно API базы данных используется для доступа к данным, хранящимся в базе данных. Firebird OO API, безусловно, выполняет эту задачу, кроме того оно поддерживает создание собственных плагинов — модулей, которые позволяют расширить возможности Firebird в соответствии с вашими потребностями. Поэтому этот документ содержит две большие части — доступ к базам данных и написание плагинов. Некоторые интерфейсы (например, вектор состояния) используются в обеих частях API, они будут обсуждаться в части доступа к данным, позже при обсуждении плагинов мы будем свободно ссылаться на них. Поэтому, даже если вы планируете написать какой-то плагин, лучше начать с первой части этого документа. Кроме того, многим плагинам необходимо обращаться к самим базам данных, и для этого необходим API доступа к данным.

Пакет установки Firebird содержит ряд живых примеров использования OO API — они находятся в каталогах `examples/interfaces` (доступ к базе данных) и `examples/dbcrypt` (плагин, выполняющий фиктивное шифрование базы данных). Предполагается, что читатель знаком с ISC API, используемым в Firebird, со времён Interbase.

Данный документ не претендует на полноту документации Firebird 3 — он просто описывает новый объектно-ориентированный API, и читатель должен быть знаком с основными концепциями Firebird, кроме того очень приветствуются знания о ISC API. Например, при описании работы со службами нет объяснения, что такое служба и для чего она необходима, только описание того, как получить интерфейс IService и как его использовать. Кроме того,

обратите внимание, на то что примеры кода не используют множество мощных возможностей C++. Не используются указатели с подсчетом ссылок, не используются другие хранители RAII, не используются шаблоны (кроме одного в публичных заголовках firebird) и т. д. Основная цель — сделать этот текст применимым не только для программистов C++, потому что наш API ориентирован на поддержку не только C++ но для других, более простых языков тоже.

Доступ к базам данных

Созданеи базы данных и соединение с существующей базой данных

Прежде всего нам нужно получить доступ к интерфейсу IMaster. `IMaster` — это основной интерфейс Firebird, необходимый для доступа ко всем остальным интерфейсам. Поэтому существует особый способ доступа к нему — единственное, что нужно это использование простой функции OO API, называемой `fb_get_master_interface()`. Эта функция не имеет параметров и всегда преуспевает. Существует один и только один экземпляр `IMaster` для каждой клиентской библиотеки Firebird, поэтому не нужно заботиться об освобождении памяти, используемой мастер-интерфейсом. Самый простой способ получить к нему доступ из вашей программы — иметь соответствующую глобальную или статическую переменную:

```
static IMaster* master = fb_get_master_interface();
```

Для многих методов, используемых в API Firebird, первым параметром является интерфейс IStatus. Это логичная замена ISC_STATUS_ARRAY, но работает отдельно с ошибками и предупреждениями (не смешивая их в одном массиве), может содержать неограниченное количество ошибок внутри и (это важно, если вы планируете реализовать `IStatus` самостоятельно) всегда сохраняет строки на которые он ссылается внутри интерфейса. Обычно для вызова других методов требуется хотя бы один экземпляр `IStatus`. Вы получаете его от `IMaster`:

```
IStatus* st = master->getStatus();
```

Если по какой-либо причине метод `getStatus()` не работает (OOM для примера), то он возврщает NULL — в этом случае очевидно, что мы не можем использовать общий метод для сообщений об ошибке, основанный на `IStatus`.

Теперь мы рассмотрим первый интерфейс, напрямую связанный с вызовами базы данных. Это IProvider-интерфейс, называемый таким образом, потому что именно этот интерфейс должен быть реализован любым поставщиком в Firebird. В клиентской библиотеке Firebird есть собственная реализация `IProvider`, которая должна использоваться для запуска любой активности базы данных. Чтобы получить его, мы вызываем метод `IMaster`:

```
IProvider* prov = master->getDispatcher();
```

При подключении к существующей базе данных или создании новой часто требуется передать множество дополнительных параметров (логин/пароль, размер страницы для новой базы данных и т.д.) вызову API. Наличие отдельных параметров на уровне языка мало реально — нам придётся слишком часто менять вызов при добавлении новых параметров, и их число будет слишком большим, независимо от того, что многие из них обычно можно пропускать. Поэтому для передачи дополнительных параметров используется специальная структура данных в памяти, называемая блок параметров базы данных (database parameters block или DPB). Её формат чётко определён, и это даёт возможность построить DPB байт за байтом. Однако гораздо проще испольовать специальный интефейс IXpbBuilder, который упрощает создание различных блоков параметров. Чтобы получить экземпляр `IXpbBuilder`, необходимо знать об ещё одном универсальном интерфейсе Firebird API — IUtil. Это своего рода заполнитель для вызовов, котрые плохо подходят в других местах. Итак мы делаем сделующее

```
IUtil* utl = master->getUtilInterface();
IXpbBuilder* dpb = utl->getXpbBuilder(&status, IXpbBuilder::DPB, NULL, 0);
```

Этоn код создает пустой конструктор блоков параметров типа DPB. Теперь добавление необходимого параметра в него тривиально:

```
dpb->insertInt(&status, isc_dpb_page_size, 4 * 1024);
```

будет создавать базу данных с размером страницы 4 Кб и значениями

```
dpb->insertString(&status, isc_dpb_user_name, "sysdba");
dpb->insertString(&status, isc_dpb_password, "masterkey");
```

смысл которых понятен.

Следующее приведено для C++: мы почти готовы вызвать метода `createDatabase()` экземпляра `IProvider`, но перед этим необходимо сказать несколько слов о концепции Status Wrapper (обёртка над статусом). Status Wrapper не я вляется интерфейсом, это очень тонка обёртка над интефейсом `IStatus`. Она позволяет найстраивать поведение C++ API (изменяет способ обработки ошибок, возвращаемых в интефейсе `IStatus`). Первое время мы рекомендуем использовать `ThrowStatusWrapper`, который вызывает исключение C++ каждый раз, когда в `IStatus` возвращается ошибка.

```
ThrowStatusWrapper status(st);
```

Теперь мы можем создать новую пустую базу данных:

```
IAttachment* att = prov->createDatabase(&status, "fbtests.fdb",
    dpb->getBufferLength(&status), dpb->getBuffer(&status));
printf("Database fbtests.fdb created\n");
```

Обратите внимание, мы не проверяем статус после вызова `createDatabase()`, потому что в случае ошибки будет возникать исключение C++ или Pascal (поэтому в вашей программе очень полезно иметь try/catch/except синтаксис). Мы также используем две новые функции из `IXpbBuilder` - `getBufferLength()` и `getBuffer()`, которые извлекают данные из интерфейса в формате DPB. Как видите, нет необходимости явно проверять статус функций, возвращая промежуточные результаты.

Отсоединение от только что созданной базы данных тривиально:

```
att->detach(&status);
```

Теперь осталось окружить все операторы блоком try и написать обработчик в блоке catch. При использовании `ThrowStatusWrapper`, вы всегда должны обрабатывать (catch) исключение класса `FbException`, определённого в C++ API, в Pascal вы также должны работать с классом `FbException`. Блок обработки исключений в простейшем случае выглядит так:

```
catch (const FbException& error)
{
    char buf[256];
    utl->formatStatus(buf, sizeof(buf), error.getStatus());
    fprintf(stderr, "%s\n", buf);
}
```

Обратите внимание, здесь мы используем ещё одну функцию из IUtil — `formatStatus()`. Она возвращает буфер с текстом, описывающем ошибку (предупреждение), сохранённую в параметре `IStatus`.

Чтобы подключиться к существующей базе данных используйте метод `attachDatabase()` интефейса `IProvider` вместо `createDatabase()`. Все параметры одинаковы для обоих методов.

```
att = prov->attachDatabase(&status, "fbtests.fdb", 0, NULL);
```

Данный пример не использует никаких дополнительных параметров DPB. Учтите, что без логина/пароля любое удалённое подключение будет неудачно, если не настроена

доверительная аутетнтификация. Конечно информация о пользователе может быть предоставлена окружением (в переменных ISC_USER и ISC_PASSWORD), так же как это было раньше.

Папка `examples` содержит завершённые примеры, в том числе и примеры создания базы данных - `01.create.cpp` и `01.create.pas`. При чтении данного документа, полезно построить (build) примеры и попытаться запустить их.

Работа с транзакциями

Только создание пустых баз данных определенно недостаточно для работы с РСУБД. Мы хотим иметь возможность создавать в базе данных различные объекты (например, таблицы и т. д.) и вставлять данные в эти таблицы. В Firebird любая операция с базой данных выполняется под управлением транзакций. Поэтому прежде всего мы должны научиться старовать транзакцию. Здесь мы не обсуждаем распределенные транзакции (поддерживаемые интерфейсом IDtc), чтобы избежать ненужных для большинства пользователей сложностей. Запуск нераспределенной транзакции очень прост и выполняется через интерфейс подключения:

```
ITransaction* tra = att->startTransaction(&status, 0, NULL);
```

В этом примере используются параметры транзакции по умолчанию — TPB не передается методу `startTransaction()`. Если вам нужна транзакция с параметрами отличными от параметров по умолчанию, вы можете создать соответствующий IXpbBuilder и добавить к нему необходимые элементы:

```
IXpbBuilder* tpb = utl->getXpbBuilder(&status, IXpbBuilder::TPB, NULL, 0);
tpb->insertTag(&status, isc_tpb_read_committed);
```

и передать готовый TPB в `startTransaction()`:

```
ITransaction* tra = att->startTransaction(&status, tpb->getBufferLength(&status),
    tpb->getBuffer(&status));
```

Интерфейс транзакции используется как параметр во множестве других вызовов API, но сам он не выполняет никаких действий, кроме фиксации/откат (commit/rollback) транзакции, может быть сохранением (retaining):

```
tra->commit(&status);
```

Вы можете посмотреть, как начать и подтверждать транзакцию в примерах `01.create.cpp` и `01.create.pas`.

Выполнение оператора SQL без входных параметров и возвращаемых строк

После старта транзакции мы говтоы выполнять наши первые SQL операторы. Используемый для этого метод `execute()` в IAttachment является довольно универсальным, и может также использоваться для выполнения операторов SQL с входными и выходными параметрами (что типично для инструкции EXECUTE PROCEDURE), но сейчас мы будем использовать наиболее простую его форму. Могут быть выполнены как DDL, так и DML операторы:

```
att->execute(&status, tra, 0, "create table dates_table (d1 date)",
    SQL_DIALECT_V6, NULL, NULL, NULL, NULL);
tra->commitRetaining(&status);
att->execute(&status, tra, 0, "insert into dates_table values (CURRENT_DATE)",
    SQL_DIALECT_V6, NULL, NULL, NULL, NULL);
```

Как вы видите, интерфейс транзакции является обязательным параметром для метода `execute()` (должен быть NULL, только если вы выполняете инструкцию START TRANSACTION). Следующим параметром следует длина SQL оператора (может быть равна нулю, в этом случае используются правила C для определения длины строки), потом текст оператора и диалект SQL, который должен использоваться для него. Далее следует несколько NULL которые подставляются для описания метаданных, и буферов входных параметров и выходных данных. Полное описание этого метода представлено в интерфейсе IAttachment.

Выполнение SQL операторов с входными параметрами

Существует два способа выполнения оператора с входными параметрами. Выбор правильного метода зависит от того, нужно ли вам выполнять его более одного раза, и знаете ли вы заранее формат параметров. Когда этот формат известен, и оператор нужно запускать только один раз, тогда вы можете воспользоваться одиночным вызовом `IAttachment::execute()`. В противном случае сначала необходимо подготовить SQL-запрос, после чего его можно выполнять многократно с различными параметрами.

Чтобы подготовить SQL оператор для выполнения, используйте метод `prepare()` интерфейса IAttachment:

```
IStatement* stmt = att->prepare(&status, tra, 0,
    "UPDATE department SET budget = ? * budget + budget WHERE dept_no = ?",
    SQL_DIALECT_V6, IStatement::PREPARE_PREFETCH_METADATA);
```

Если вы не собираетесь использовать описание параметров из Firebird (т.е. вы можете предоставить эту информацию самостоятельно), используйте `IStatement::PREPARE_PREFETCH_NONE` вместо `IStatement::PREPARE_PREFETCH_METADATA` — это немного снизит клиет/серверный трафик и сохранит ресурсы.

В ISC API используется структура XSQLDA для описания фолрмата параметров оператора. Новый API не использует XSQLDA — вместо этого используется интерфейс IMessageMetadata. Набор входных параметров (а также запись, взятая из курсора) описывается в Firebird API

таким же образом, далее называемый сообщением. `IMessageMetadata` передаётся в качестве параметра в методы обмена сообщениями между программой и движком базы данных. Существует много способов получить экземпляр `IMessageMetadata`, вот некоторые из них:

- получить из IStatement;
- построить используя IMetadataBuilder интерфейс;
- иметь собственную реализацию этого интерфейса.

Полученить метаданые из подговтоленного запроса очень просто — метод `getInputMetadata()` возвращает интерфейс, описывающий входное сообщение (т.е. параметры оператора), интерфейс, возвращаемый `getOutputMetadata()`, описывает выходное сообщение (т.е. строку выбранных данных или значения возвращаемые процедурой). В нашем случае мы можем сделать так:

```
IMessageMetadata* meta = stmt->getInputMetadata(&status);
```

Или мы можем построить сообщение метаданных самостоятельно. Для этого прежде всего нам необходимо получить интерфейс построителя:

```
IMetadataBuilder* builder = master->getMetadataBuilder(&status, 2);
```

Второй параметр — это ожидамеое количество полей в сообщении, его можно изменить позже, т.е. он необходим только для оптимизации.

Теперь необходимо задать индивидуальные характеристики полей в построителе. Минимально необходимыми являются типы полей и длина для строковых полей:

```
builder->setType(&status, 0, SQL_DOUBLE + 1);

builder->setType(&status, 1, SQL_TEXT + 1);
builder->setLength(&status, 1, 3);
```

Новый API использует старые константы для типов SQL, наименьший бит, как и раньше, испольуется возможности принимать null значение. В некоторых случаях имеет смысл установить подтип (для BLOB), набор символов (для текстовых полей) или масштаб (для числовых полей). Наконец, пришло время получить экземпляр `IMessageMetadata`:

```
IMessageMetadata* meta = builder->getMetadata(&status);
```

Здесь мы не обсуждаем собственную реализацию `IMessageMetadata`. Если вам это интересено, то вы можете посмотреть пример `05.user_metadata.cpp`.

Итак, мы получили экземпляр описания метаданных входных параметров. Но для работы с сообщением нам также необходим буфер. Размер буфера является одной из основных

характеристик сообщений метаданных и возвращается методом `getMessageLength()` из `IMessageMetadata`:

```
char* buffer = new char[meta->getMessageLength(&status)];
```

Чтобы иметь дело с отдельными значениями внутри буфера, смещение к ним должно быть принято в расчёт. IMessageMetadata знает о смещениях для всех значений в сообщении, используя его, мы можем создавать указатели на них:

```
double* percent_inc = (double*) &buffer[meta->getOffset(&status, 0)];
char* dept_no = &buffer[meta->getOffset(&status, 1)];
```

Кроме того, не забывайте установить NULL флаги:

```
short* flag = (short*)&buffer[meta->getNullOffset(&status, 0)];
*flag = 0;

flag = (short*) &buffer[meta->getNullOffset(&status, 1)];
*flag = 0;
```

Полсе завершения манипуляция со смещениями, мы готовы получить значения параметров:

```
getInputValues(dept_no, percent_inc);
```

и выполнить подготовленный оператор:

```
stmt->execute(&status, tra, meta, buffer, NULL, NULL);
```

Два последних NULL в параметрах предназначены для выходных сообщений и обычно используются для оператора EXECUTE PROCEDURE.

Если вам не нужно получать метаданные из оператора и вы планируете выполнить его только один раз, то вы можете выбрать более простой способ — используйте метод `execute()` из интерфейса IAttachment:

```
att->execute(&status, tra, 0,
    "UPDATE department SET budget = ? * budget + budget WHERE dept_no = ?",
    SQL_DIALECT_V6, meta, buffer, NULL, NULL);
```

В этом случае вам вообще не нужно использовать IStatement.

Пример того, как выполнить оператор UPDATE с параметрами, присутствует в `02.update.cpp`, вы также увидите, как возбужденное исключение в триггере/процедуре может быть перехвачено в программе на C ++.

Открытие курсора и извлечение данных из него

Единственный способ получить строки данных, возвращаемых оператором SELECT в OO API — это использовать интерфейс IResultSet. Этот интерфейс возвращается методом `openCursor()` как в `IAttachment`, так и в `IStatement`. `openCursor()` в большинстве аспектов похож на `execute()`, и выбор каким образом открыть курсор (с использованием подготовленного оператора или непосредственно из интерфейса подключения) тот же. В примерах `03.select.cpp` и `04.print_table.cpp` используются оба способа. Обратите внимание на одно отличие метода `openCursor()` по сравнению с `execute()` — никто не передает буфер для выходного сообщения в `openCursor()`, он будет передан позже, когда данные будут извлечены из курсора. Это позволяет открывать курсор с неизвестным форматом выходного сообщения (NULL передается вместо выходных метаданных). В этом случае Firebird использует формат сообщения по умолчанию, который может запрашен через интерфейс `IResultSet`:

```
const char* sql = "select * from ..."; // some select statement

IResultSet* curs = att->openCursor(&status, tra, 0, sql, SQL_DIALECT_V6,
    NULL, NULL, NULL, NULL, 0);

IMessageMetadata* meta = curs->getMetadata(&status);
```

Позже эти метаданные могут использоваться для выделения буфера для данных и разбора извлечённых строк.

В качестве альтернативы можно сначала подготовить оператор, получить метаданные из подготовленного оператора и после этого открыть курсор. Это предпочтительный способ, если вы предполагаете, что курсор будет открыт более одного раза.

```
IStatement* stmt = att->prepare(&status, tra, 0, sql, SQL_DIALECT_V6,
    IStatement::PREPARE_PREFETCH_METADATA);

IMessageMetadata* meta = stmt->getOutputMetadata(&status);

IResultSet* curs = stmt->openCursor(&status, tra, NULL, NULL, NULL, 0);
```

Мы получили (тем или иным способом) экземпляр описания метаданных выходных полей (строк в наборе). Для работы с сообщением нам также нужен буфер:

```
unsigned char* buffer = new unsigned char[meta->getMessageLength(&status)];
```

В `IResultSet` есть много различных методов выборки, но когда курсор открыт не с параметром SCROLL, то работает только `fetchNext()`, то есть можно перемещаться по

записям только вперед. В дополнение к ошибкам и предупреждениям в статусе метод `fetchNext()` возвращает код завершения, который может иметь значения RESULT_OK (когда буфер заполняется значениями для следующей строки) или RESULT_NO_DATA (когда в курсоре больше строк не осталось). RESULT_NO_DATA не является состоянием ошибки, это нормальное состояние после завершения метода, которое сигнализирует, что данных в курсоре больше нет. Если используется оболочка статуса (Status Wrapper), то исключение не бросается в случае возврата ошибки. Может быть возвращено еще одно значение — RESULT_ERROR — оно означает отсутствие данных в буфере и ошибки в статусе векторе. Метод `fetchNext()` обычно вызывается в цикле:

```
while (curs->fetchNext(&status, buffer) == IStatus::RESULT_OK)
{
    // row processing
}
```

То что происходит при обработке строк, зависит от ваших потребностей. Для получения доступа к определённому полю следует использовать смещенеи поля:

```
unsigned char* field_N_ptr = buffer + meta->getOffset(&status, n);
```

где n - номер поля в сообщении. Этот указатель должен быть присвоен соответствующему типу, в зависимости от типа поля. Например, для поля VARCHAR, следует использовать приведение к структуре vary:

```
vary* v_ptr = (vary*) (buffer + meta->getOffset(&status, n));
```

Теперь мы можем напечатать значение поля:

```
printf("field %s value is %*.*s\n",
       meta->getField(&status, n),
       v_ptr->vary_length,
       v_ptr->vary_length,
       v_ptr->vary_string);
```

Если вам нужна максимальная производительность, будет полезно кэшировать необходимые значения метаданных, как это сделано в наших примерах `03.select.cpp` и `04.print_table.cpp`.

Использование макросов FB_MESSAGE для статических сообщений

Работа с данными с использованием смещений довольно эффективна, но требует написания большого количества кода. В C ++ эту проблему можно решить с помощью шаблонов, но

даже по сравнению с ними наиболее удобным способом работы с сообщением является представление его в родном (для заданного языка) форме — структуре в C/C ++, записи в Pascal и т. д. Конечно это работает только в том случае, если формат сообщения известен заранее. Для создания таких структур в C ++ в Firebird существует специальный макрос FB_MESSAGE.

FB_MESSAGE имеет 3 аргумента: имя сообщения (структуры), тип обёртки статуса (status wrapper) и список полей. Использование первого и второго аргумента очевидно, список полей содержит пары `(field_type, field_name)`, где `field_type` является одним из следующих:

- `FB_BIGINT`
- `FB_BLOB`
- `FB_BOOLEAN`
- `FB_CHAR(len)`
- `FB_DATE`
- `FB_DOUBLE`
- `FB_FLOAT`
- `FB_INTEGER`
- `FB_INTL_CHAR(len, charSet)`
- `FB_INTL_VARCHAR(len, charSet)`
- `FB_SCALED_BIGINT(x)`
- `FB_SCALED_INTEGER(x)`
- `FB_SCALED_SMALLINT(x)`
- `FB_SMALLINT`
- `FB_TIME`
- `FB_TIMESTAMP`
- `FB_VARCHAR(len)`

В сгенерированной предпроцессором структуре типы integer и float сопоставляются с соответсвующими типами C, типы date и time — с классами FbDate и FbTime (все упомянутые здесь классы находятся в пространстве имен Firebird), тип timestamp — с классом FbTimestamp, содержащим два публичных члена данных дату и время соответсвующих классов, тип char — со структурой FbChar и varchar — со структурой FbVarChar. Для каждого поля предпроцессор создаст два члена данных с сообщении — `name` для значения поля/параметра и `nameNull` для индикатора NULL. Конструктор сообщений имеет 2 параметра — указатель на оболочку статуса (status wrapper) и главный интерефейс (master interface):

```
FB_MESSAGE(Output, ThrowStatusWrapper,
```

```
    (FB_SMALLINT, relationId)
    (FB_CHAR(31), relationName)
    (FB_VARCHAR(100), description)
) output(&status, master);
```

Для статических сообщений использование FB_MESSAGE является самым лучшим выбором, в то же время они легко могут быть переданы в методы `execute`, `openCursor` и `fetch`:

```
rs = att->openCursor(&status, tra, 0, sqlText,
      SQL_DIALECT_V6, NULL, NULL, output.getMetadata(), NULL, 0);
```

и используется для работы со значениями отдельных полей:

```
while (rs->fetchNext(&status, output.getData()) == IStatus::RESULT_OK)
{
  printf("%4d %31.31s %*.*s\n", output->relationId, output->relationName.str,
    output->descriptionNull ? 0 : output->description.length,
    output->descriptionNull ? 0 : output->description.length,
    output->description.str);
}
```

Пример использования макроса FB_MESSAGE для работы с сообщениями приведен в примере `06.fb_message.cpp`.

Работа с BLOB

Для BLOBs Firebird хранит в буфере сообщения идентификатор BLOB — 8 байтовый объект, который должен быть выравнен по 4-байтовый границе. Идентификатор имеет тип ISC_QUAD. Интерфейс IAttachment имеет 2 метода работы с BLOB — `openBlob()` и `createBlob()`, возвращающие интерфейс IBlob и имеющие одинаковый набор параметров, но выполняющие несколько разные действия: `openBlob()` принимает BLOB идентификатор из сообщения и подготавливает BLOB для чтения, а `createBlob()` создает новый BLOB, помещает его идентификатор в сообщение и подготавливает BLOB для записи.

Для работы с BLOBs прежде всего необходимо включить в сообщение их BLOB-идентификаторы. Если вы получите метаданные из поля движка Firebird соответствующего типа, то этот идентификатор уже будет присутствовать. В этом случае вы просто используете его смещение (при условии, что переменная blobFieldNumber содержит номер поля BLOB) (и соответствующее NULL смещение для проверки NULL или установки NULL флага) для получения указателя в буфере сообщений:

```
ISC_QUAD* blobPtr =
  (ISC_QUAD*) &buffer[metadata->getOffset(&status, blobFieldNumber)];
ISC_SHORT* blobNullPtr =
  (ISC_SHORT*) &buffer[metadata->getNullOffset(&status, blobFieldNumber)];
```

Если вы используете статические сообщенияи макрос FB_MESSAGE, то поле BLOB будет объявлено как тип FB_BLOB:

```
FB_MESSAGE(Msg, ThrowStatusWrapper,
    (FB_BLOB, b)
) message(&status, master);

ISC_QUAD* blobPtr = &message->b;
ISC_SHORT* blobNullPtr = &message->bNull;
```

Для создания нового BLOB, вызовите метод `createBlob()`:

```
IBlob* blob = att->createBlob(status, tra, blobPtr, 0, NULL);
```

Последние два параметра требуются только в том случае, если вы хотите использовать blob-фильтры или blob-поток, который не рассмариваются здесь.

Теперь Blob интерфейс готов принять данные в BLOB. Используйте метод `putSegment()` для отправки данных в движок:

```
void* segmentData;
unsigned segmentLength;
while (userFunctionProvidingBlobData(&segmentData, &segmentLength))
    blob->putSegment(&status, segmentLength, segmentData);
```

После отправки некоторых данных в BLOB не забудьте закрыть blob-интерфейс:

```
blob->close(&status);
```

Убедитесь, что null флаг не установлен (не требуется, если вы сбросили весь буфер сообщений перед созданием BLOB):

```
*blobNullPtr = 0;
```

и сообщение, содержащее BLOB, может использоваться в операторе вставки или обновления. После выполнения этого операторе новый BLOB будет сохранен в базе данных.

Чтобы прочитать blob, необходимо получить содержащего его идентификатора в сообщении от двигателя firebird. Это можно сделать с помощью методов `fetch()` или `execute()`. После используйте метод `openBlob()`:

```
IBlob* blob = att->openBlob(status, tra, blobPtr, 0, NULL);
```

Blob интерфейс готов предоставить данные BLOB. Используйте метод `getSegment()` для получения данных от движка:

```
char buffer[BUFSIZE];
unsigned actualLength;

for(;;)
{
  switch (blob->getSegment(&status, sizeof(buffer), buffer, &actualLength))
  {
    case IStatus::RESULT_OK:
      userFunctionAcceptingBlobData(buffer, actualLength, true);
      continue;

    case IStatus::RESULT_SEGMENT:
      userFunctionAcceptingBlobData(buffer, actualLength, false);
      continue;

    default:
      break;
  }
}
```

Последний параметр в `userFunctionAcceptingBlobData()` — это флаг достижения конца сегмента — когда `getSegment()` возвращает код завершения RESULT_SEGMENT, о котором будет функция уведомлена (в последний параметр передан `false`), то есть этот сегмент прочитан не полностью, и продолжение ожидается при следующем вызове.

Закончив работать с BLOB, не забудьте закрыть его:

```
blob->close(&status);
```

Работа с событиями

Интерфейс событий не был завершен в FIrebird 3.0, мы ожидаем, что в следующей версии будет что-то более интересное. Минимальная существующая поддержка выглядит следующим образом: IAttachment содержит метод `queEvents()`, который выполняет почти те же функции, что и вызов `isc_que_events()`. Вместо пары параметров `FPTR_EVENT_CALLBACK ast` и `void* arg`, необходимых для вызова кода пользователя, когда в Firebird происходит событие, используется интерфейс обратного вызова `IEventCallback`. Это традиционный подход, который помогает избежать небезопасных бросков из void* в пользовательской функции. Другое важное различие заключается в том, что вместо идентификатора события (вида обработчика) эта функция возвращает ссылку на интерфейс IEvents, имеющий метод `cancel()`, используемый остановки ожидании события. В отличие от идентификатора, который уничтожается автоматически при поступлении события, интерфейс не может быть уничтожен

автоматически, если событие получено непосредственно перед вызовом метода `cancel()` для, то это вызовет segfault из-за того, что интерфейс уже будет уничтожен. Поэтому после получения события интерфейс IEvents должен быть явно освобождён. Это может быть сделано, например, прямо перед запросом события из очереди в следующий раз:

```
events->release();
events = NULL;

events = attachment->queEvents(&status, this, eveLen, eveBuffer);
```

Установка указателя интерфейса в NULL полезна в случае возникновения исключения в queEvents. В других аспектах обработка событий не изменилась по сравнению с ISC API. Для получения дополнительной информации используйте наш пример `08.events.cpp`.

Использование сервисов

Чтобы начать пользоваться сервисами (службами), прежде всего необходимо подключиться к менеджеру сервиса. Это делается с помощью метода `attachServiceManager()` интерфейса IProvider. Этот метод возвращает интерфейс IService, который позже используется для связи с сервисом. Чтобы подготовить SPB для подключения к диспетчеру сервиса, вы можете использовать `IXpbBuilder`:

```
IXpbBuilder* spb1 = utl->getXpbBuilder(&status, IXpbBuilder::SPB_ATTACH, NULL, 0);

spb1->insertString(&status, isc_spb_user_name, "sysdba");
spb1->insertString(&status, isc_spb_password, "masterkey");
```

и подключится:

```
IService* svc = prov->attachServiceManager(&status, "service_mgr",
    spb1->getBufferLength(&status), spb1->getBuffer(&status));
```

Используя `IService`, вы можете выполнять как доступные для служб действия — запускать службы, так и запрашивать различную информацию о запущенных утилитах и сервере в целом. При запросе информации, есть одно ограничение — формат блока параметров, используемые методом `query()`, в Firebird 3 не поддерживается `IXpbBuilder`. Вероятно, поддержка будет добавлена в более поздних версиях, в Firebird 3 вам придется создавать и анализировать этот блок вручную. Формат этого блока повторяет старый формат (используемый в ISC API) один в один.

Чтобы стартовать сервис, необходимо прежде всего создать соответствующий SPB:

```
IXpbBuilder* spb2 = utl->getXpbBuilder(&status, IXpbBuilder::SPB_START, NULL, 0);
```

и добавить к нему необходимые элементы. Например, для печати статистики шифрования для базы данных `employee` в SPB следует поместить следующее:

```
spb2->insertTag(&status, isc_action_svc_db_stats);
spb2->insertString(&status, isc_spb_dbname, "employee");
spb2->insertInt(&status, isc_spb_options, isc_spb_sts_encryption);
```

После этого сервис можно запустить с использованием метода `start()` интерфейса `IService`:

```
svc->start(&status, spb2->getBufferLength(&status), spb2->getBuffer(&status));
```

Многие запущенные службы (включая упомянутый здесь gstat) во время выполнения возвращают текстовую информацию. Чтобы отобразить её, необходимо запросить эту информацию у запущенного сервис построчно. Это делается с помощью вызова метода `query()` интерфейса IService с соответствующими блоками параметров для приёма и отправки. Блок отправки может содержать различную вспомогательную информацию (например, тайм-аут запроса у службы) или информацию, которая должна быть передана в служебную программу stdin, или может быть пустым в простейшем случае. Блок приема должен содержать список тегов, которые вы хотите получать из службы. Для большинства утилит это единственный `isc_info_svc_line`:

```
const unsigned char receiveItems1[] = {isc_info_svc_line};
```

Кроме того, для запроса этой информации для неё необходим буфер:

```
unsigned char results[1024];
```

После этих предварительных шагов мы готовы запросить информацию из сервиса в цикле (каждая строка возвращается в одном вызове `query()`):

```
do
{
    svc->query(&status, 0, NULL,
               sizeof(receiveItems1), receiveItems1,
               sizeof(results), results);
} while (printInfo(results, sizeof(results)));
```

В этом примере мы предполагаем, что функция `printInfo()` возвращает TRUE, пока сервис возвращает блок результатов, содержащий следующую выходную строку (то есть до конца потока данных из сервис). Формат блока результатов варьируется от сервис к сервису, а некоторые сервисы, такие как gsec, создают исторические форматы, которые не являются

тривиальными для синтаксического анализа, но это за рамки данной главы. Минимальный рабочий пример `printInfo()` присутствует в примере `09.service.cpp`.

Тот же метод запроса используется для извлечения информации о сервере, но в этом случае функция запроса не вызывается в цикле, т. е. буфер должен быть достаточно большим, чтобы сразу вместить всю информацию. Это не слишком сложно, так как обычно такие вызовы не возвращают много данных. Как и в предыдущем случае, необходимо начать с того, чтобы разместить в блоке приема необходимые элементы — в нашем примере это isc_info_svc_server_version:

```
const unsigned char receiveItems2[] = {isc_info_svc_server_version};
```

Существующий буфера результатов из предыдущего вызова может быть использован повторно. В данном случае цикл не требуется:

```
svc->query(&status, 0, NULL,
           sizeof(receiveItems2), receiveItems2,
           sizeof(results), results);

printInfo(results, sizeof(results));
```

После завершения сервисных задач не забудьте отключить сервис:

```
svc->detach(&status);
```

Написание плагинов

Чтобы написать плагин, нужно реализовать некоторые интерфейсы и поместить вашу реализацию в динамическую библиотеку (`.dll` в Windows или `.so` в Linux), которую называют модулем плагина или просто модулем. В большинстве случаев одиночный плагин размещается в динамической библиотеке, но не обязательно. Один из этих интерфейсов — IPluginModule — является модульным (как более или менее ясно из его имени), другие отвечают за плагин. Также каждый модуль плагина должен содержать специальную экспортированную точку входа `firebird_plugin()`, имя которой указано в файле `include/firebird/Interfaces.h` как FB_PLUGIN_ENTRY_POINT.

В предыдущей части мы в основном описывали, как использовать существующие интерфейсы, здесь основное внимание будет уделено самостоятельной реализации интерфейсов. Разумеется, для этого можно и нужно использовать уже существующие интерфейсы, общие для доступа к базам данных firebird (уже описанные) и некоторые дополнительные интерфейсы, специально предназначенные для плагинов.

Далее активно используется пример плагина шифрования базы данных `examples/dbcrypt/DbCrypt.cpp`. Будет хорошей идеей собрать этот пример самостоятельно и изучить его при чтении позже.

Реализация модуля плагина

Плагины активно взаимодействуют со специальным компонентом firebird, называемым диспетчером плагинов. В частности, менеджер плагинов должен знать, какие модули плагина были загружены и должен быть уведомлен, если операционная система пытается выгрузить один из этих модулей без явной команды диспетчера плагина (это может произойти прежде всего при использовании встроенного сервера (embedded) - когда в программе вызывается `exit()` или основная библиотека firebird fbclient выгружается). Основная задача интерфейса `IPluginModule` — это уведомление. Прежде всего, нужно решить — как определить, что модуль будет выгружен? Когда динамическая библиотека выгружается по какой-либо причине, выполняется множество зависимых от ОС действий, и некоторые из этих действий могут использоваться для обнаружения этого факта в программе. При написании плагинов, распространяемых вместе с firebird, мы всегда используем вызов деструктора глобальной переменной. Большой «плюс» этого метода заключается в том, что он независим от ОС (хотя что-то вроде функции `exit()`, возможно, также успешно используется). Но использование деструктора позволяет легко сконцентрировать почти все, что связано с обнаружением выгрузки в одном классе, реализуя в то же время интерфейс IPluginModule.

Минимальная реализация выглядит следующим образом:

```
class PluginModule : public IPluginModuleImpl<PluginModule, CheckStatusWrapper>
{

private:
  IPluginManager* pluginManager;

public:
  PluginModule()
    : pluginManager(NULL)
  { }


  ~PluginModule()
  {
    if (pluginManager)
    {
      pluginManager->unregisterModule(this);
      doClean();
    }
  }

  void registerMe(IPluginManager* m)
  {
    pluginManager = m;
    pluginManager->registerModule(this);
  }

  void doClean()
  {
    pluginManager = NULL;
  }

};
```

Единственным членом данных является интерфейс диспетчера плагинов IPluginManager. Он передается функции `registerModule()` и сохраняется в приватной переменной, в то же время модуль регистрируется в диспетчере плагинов методом `callModule()` с собственным адресом в качестве единственного параметра. Переменная `pluginManager` не только сохраняет указатель на интерфейс, но одновременно служит в качестве флага, что модуль зарегистрирован. Когда вызывается деструктор зарегистрированного модуля, он уведомляет диспетчер плагинов о неожиданной выгрузке с помощью вызова `unregisterModule()`, передающим указатель на себя. Когда диспетчер плагинов будет регулярно выгружать модуль, то в первую очередь вызов метода `doClean()` меняет состояние модуля на незарегистрированное, и это позволяет избежать вызова `unregisterModule()`, когда ОС выполняет фактическую выгрузку.

Реализовав интерфейс плагина `IPluginModule`, мы встретились с первым интерфейсом, необходимым для реализации плагинов — `IPluginManager`. Он будет активно использоваться позже, остальные члены этого класса вряд ли потребуются вам после копирования в вашу программу. Просто не забудьте объявить глобальную переменную этого типа и вызвать функцию `registerMe()` из FB_PLUGIN_ENTRY_POINT.

Основной интерфейс любого плагина

Мы получили (тем или иным способом) экземпляр описания метаданных выходных полей (строки в наборе данных). Для работы с сообщением нам также необходим буфер: приступим к реализации самого плагина. Тип основного интерфейса зависит от типа плагина (что очевидно), но все они основаны на общем интерфейсе `IPluginBase` с подсчётом ссылок, который выполняет общие для всех плагинов (и очень простые) задачи. Каждый плагин имеет некоторый (тоже с подсчётом ссылок) объект, которому принадлежит этот плагин. Чтобы выполнять интеллектуальное управление жизненным циклом плагинов, любой плагин должен иметь возможность хранить информацию о владельце и сообщать об её диспетчеру плагинов по запросу. Это означает, что каждый плагин должен реализовывать два тривиальных метода `setOwner()` и `getOwner()`, содержащиеся в интерфейсе `IPluginBase`. Зависимые от типа плагина методы, безусловно, более интересны — они обсуждаются в части описания интерфейсов.

Давайте рассмотрим типичную часть реализации любого плагина (здесь я специально использую несуществующий тип `SomePlugin`):

```
class MyPlugin : public ISomePluginImpl<MyPlugin, CheckStatusWrapper>
{
public:
  explicit MyPlugin(IPluginConfig* cnf) throw()
     : config(cnf), refCounter(0), owner(NULL)
  {
    config->addRef();
  }
  ...
```

Конструктор получает в качестве параметра интерфейс конфигурации плагина. Если вы собираетесь конфигурировать плагин каким-то образом, то рекомендуется сохранить этот интерфейс в вашем плагине и использовать его позже. Это позволит вам использовать общий стиль конфигурации firebird, позволяя пользователям иметь привычную конфигурацию и свести к минимуму написание кода. Конечно, при сохранении какого-либо ссылочного интерфейса

лучше не забывать добавлять ссылку на него. Также установите счетчик ссылок в 0 и владельца плагина в NULL.

```
  ~MyPlugin()
  {
    config->release();
  }
```

Деструктор освобождает конфигурационный интерфейс. Обратите внимание: мы не меняем счетчик ссылок нашего владельца, потому что он принадлежит нам, а не мы принадлежим ему.

```
  // IRefCounted implementation
  int release()
  {
    if (--refCounter == 0)
    {
      delete this;
      return 0;
    }
    return 1;
  }


  void addRef()
  {
    ++refCounter;
  }
```

Абсолютно типичная реализация объекта с одсчётом ссылок.

```
  // IPluginBase implementation
  void setOwner(IReferenceCounted* o)
  {
    owner = o;
  }

  IReferenceCounted* getOwner()
  {
    return owner;
  }
```

Как и было обещано, реализация `IPluginBase` тривиальна.

```
  // ISomePlugin implementation
  // … here go various methods required for particular plugin type
private:
  IPluginConfig* config;
  FbSampleAtomic refCounter;
  IReferenceCounted* owner;
```

```
};
```

В этом примере формальная часть реализации основного интерфейса плагина завершена. После добавления специфичных для типа методов (и, возможно, написания кода, чтобы сделать их полезным), интерфейс готов.

Фабрика плагинов

Еще один интерфейс, необходимый для работы плагина — IPluginFactory. Фабрика создает экземпляры плагина и возвращает их в диспетчер плагинов. Фабрика обычно выглядит так:

```
class Factory : public IPluginFactoryImpl<Factory, CheckStatusWrapper>
{
public:
  IPluginBase* createPlugin(CheckStatusWrapper* status,
                            IPluginConfig* factoryParameter)
  {
    MyPlugin* p = new MyPlugin(factoryParameter);
    p->addRef();
    return p;
  }
};
```

Здесь внимание следует уделить тому факту, что даже в случае, когда код в функции может генерировать исключения (оператор new может бросать в случае, когда память исчерпана), то не обязательно всегда вручную определять блок try/catch — реализация интерфейсов Firebird делает это работа за вас, в реализации `IPluginFactory` эта обработка происходит в шаблоне `IPluginFactoryImpl`. Обратите внимание, что обертки статуса по умолчанию выполняют полноценную обработку только для `FbException`. Но если вы (что определенно имеет смысл, если вы работаете над каким-то крупным проектом), определите свою собственную оболочку, вы можете обрабатывать любой тип исключения C++ и передавать полезную информацию об этом из своего плагина.

Точка инициализации модуля плагина

Когда диспетчер плагинов загружает модуль плагина, он вызывает процедуру инициализации модуля — единственную экспортируемую функцию плагина FB_PLUGIN_ENTRY_POINT. Для написания кода ей понадобятся две глобальные переменные — модуль плагина и фабрика плагинов. В нашем случае это:

```
PluginModule module;

Factory factory;
```

Если модуль содержит более одного плагина, вам понадобится фабрика для каждого плагина.

Для FB_PLUGIN_ENTRY_POINT мы не должны забывать, что она должна быть экспортирована из модуля плагина, для этого требуется учет некоторых особенностей ОС. Мы

делаем это, используя макрос FB_DLL_EXPORT, определенный в `examples/interfaces/ifaceExamples.h`. Если вы уверены, что используете плагин только для некоторых конкретных ОС, то вы можете сделать это место немного проще. В минимальном случае функция должна регистрировать модуль и все фабрики в диспетчере плагинов:

```
extern "C" void FB_DLL_EXPORT FB_PLUGIN_ENTRY_POINT(IMaster* master)
{
  IPluginManager* pluginManager = master->getPluginManager();
  module.registerMe(pluginManager);
  pluginManager->registerPluginFactory(IPluginManager::TYPE_DB_CRYPT,
                                       "DbCrypt_example",
                                       &factory);
}
```

Прежде всего, мы вызываем недавно написанную нами функцию `PluginModule::registerMe()`, которая сохраняет `IPluginManager` для дальнейшего использования и регистрирует наш модуль плагина. Затем региструем фабрику (или фабрики если в одном модуле будет несколько плагинов). Мы должны передать правильный тип плагина (допустимые типы перечислены в интерфейсе `IPluginManager`) и имя, под которым будет зарегистрирован плагин. В простейшем случае он должен совпадать с именем динамической библиотеки модуля плагина. Это правило поможет вам не настраивать плагин вручную в `plugins.conf`.

Обратите внимание — в отличие от приложений плагины не должны использовать `fb_get_master_interface()` для получения `IMaster`. Вместо этого следует использовать экземпляр, переданный в FB_PLUGIN_ENTRY_POINT. Если вам нужен мастер-интерфейс в вашем плагине, позаботьтесь об его сохранении в этой функции.

Интерфейсы от A до Z

В этом глоссарии мы не перечисляем интерфейсы, которые не используются активно (например, `IRequest`, необходимые в первую очередь для поддержки устаревших запросов API ISC). Та же ссылка может быть получен из некоторых методов (например, `compileRequest()` в `IAttachment`). Для интерфейсов/методов, имеющих прямой аналог в старом API, этот аналог будет указан.

Общие интерфейсы

IAttachment

Назначение: Интерфейс `IAttachment` заменяет `isc_db_handle`.

1.

```
void getInfo(StatusType* status,
             unsigned itemsLength,
             const unsigned char* items,
```

```
            unsigned bufferLength,
            unsigned char* buffer)
```

заменяет `isc_database_info()`.

2.

```
ITransaction* startTransaction(StatusType* status,
                               unsigned tpbLength,
                               const unsigned char* tpb)
```

частично заменяет `isc_start_multiple()`, использует координатор, чтобы запустить более одной распределённой тразакции. Позволяет объединить 2 транзакции в одну распределённую.

3.

```
ITransaction* reconnectTransaction(StatusType* status,
                                   unsigned length,
                                   const unsigned char* id)
```

позволяет подключиться к транзакции в состоянии limbo. Параметр Id содержит номер транзакции в сетевом формате заданной длины.

4.

```
IRequest* compileRequest(StatusType* status,
                         unsigned blrLength,
                         const unsigned char* blr)
```

поддержка ISC API.

5.

```
void transactRequest(StatusType* status,
                     ITransaction* transaction,
                     unsigned blrLength,
                     const unsigned char* blr,
                     unsigned inMsgLength,
                     const unsigned char* inMsg,
                     unsigned outMsgLength,
                     unsigned char* outMsg)
```

поддержка ISC API.

6.

```
IBlob* createBlob(StatusType* status,
                  ITransaction* transaction,
                  ISC_QUAD* id,
                  unsigned bpbLength,
```

```
                  const unsigned char* bpb)
```

создает новый blob, сохраняет его идентификатор в id, заменяет `isc_create_blob2()`.

7.

```
IBlob* openBlob(StatusType* status,
                ITransaction* transaction,
                ISC_QUAD* id,
                unsigned bpbLength,
                const unsigned char* bpb)
```

открывает существующий blob, заменяет `isc_open_blob2()`.

8.

```
int getSlice(StatusType* status,
             ITransaction* transaction,
             ISC_QUAD* id,
             unsigned sdlLength,
             const unsigned char* sdl,
             unsigned paramLength,
             const unsigned char* param,
             int sliceLength,
             unsigned char* slice)
```

поддержка ISC API.

9.

```
void putSlice(StatusType* status,
              ITransaction* transaction,
              ISC_QUAD* id,
              unsigned sdlLength,
              const unsigned char* sdl,
              unsigned paramLength,
              const unsigned char* param,
              int sliceLength,
              unsigned char* slice)
```

поддержка ISC API.

10.

```
void executeDyn(StatusType* status,
                ITransaction* transaction,
                unsigned length,
                const unsigned char* dyn)
```

поддержка ISC API.

11.

```
IStatement* prepare(StatusType* status,
                    ITransaction* tra,
                    unsigned stmtLength,
                    const char* sqlStmt,
                    unsigned dialect,
                    unsigned flags)
```

заменяет `isc_dsql_prepare()`. Дополнительный параметр flags позволяют контролировать, какая информация будет предварительно загружена из движка сразу (т.е. в одном сетевом пакете для удаленного операций).

12.

```
ITransaction* execute(StatusType* status,
                      ITransaction* transaction,
                      unsigned stmtLength,
                      const char* sqlStmt,
                      unsigned dialect,
                      IMessageMetadata* inMetadata,
                      void* inBuffer,
                      IMessageMetadata* outMetadata,
                      void* outBuffer)
```

выполняет любую инструкцию SQL, за исключением возврата нескольких строк данных. Частичный аналог `isc_dsql_execute2()` — вход и выход XSLQDA заменены на входные и выходные сообщения с соответствующими буферами.

13.

```
IResultSet* openCursor(StatusType* status,
                       ITransaction* transaction,
                       unsigned stmtLength,
                       const char* sqlStmt,
                       unsigned dialect,
                       IMessageMetadata* inMetadata,
                       void* inBuffer,
                       IMessageMetadata* outMetadata,
                       const char* cursorName,
                       unsigned cursorFlags)
```

выполняет оператор SQL, потенциально возвращающий несколько строк данных. Возвращает интерфейс IResultSet, который используется для извлечения этих данных. Формат выходных данных определяется параметром *`outMetadata`*, при задании NULL ипользуется формат по умолчанию. Параметр cursorName указывает имя открытого курсора (аналог `isc_dsql_set_cursor_name()`). Параметр *`cursorFlags`* необходим, чтобы открыть двунаправленный указатель курсора, для его значения значение `IStatement::CURSOR_TYPE_SCROLLABLE`.

14.

```
IEvents* queEvents(StatusType* status,
                   IEventCallback* callback,
```

```
                    unsigned length,
                    const unsigned char* events)
```

заменяет вызов `isc_que_events()`. Вместо функции обратного вызова с void* параметром используется интерфейс обратного вызова.

15.

```
void cancelOperation(StatusType* status, int option)
```

замена `fb_cancel_operation()`.

16.

```
void ping(StatusType* status)
```

проверка состояния соединения. Если тест не удаётся, то единственная возможная операция с подключением — закрыть его.

17.

```
void detach(StatusType* status)
```

заменяет `isc_detach_database()`. В случае успеха освобождает интерфейс.

18.

```
void dropDatabase(StatusType* status)
```

заменяет `isc_drop_database()`. В случае успеха освобождает интерфейс.

IBlob

Назначение: Интерфейс `IBlob` заменяет `isc_blob_handle`.

1.

```
void getInfo(StatusType* status,
             unsigned itemsLength,
             const unsigned char* items,
             unsigned bufferLength,
             unsigned char* buffer)
```

заменяет `isc_blob_info()`.

2.

```
int getSegment(StatusType* status,
               unsigned bufferLength,
```

```
                void* buffer,
                unsigned* segmentLength)
```

заменяет `isc_get_segment()`. В отличие от него никогда не возвращаются ошибки `isc_segstr_eof` и `isc_segment` (которые на самом деле не являются ошибками), вместо этого возвращаются коды завершения `IStatus::RESULT_NO_DATA` и `IStatus::RESULT_SEGMENT`, обычно возвращает `IStatus::RESULT_OK`.

3.

```
void putSegment(StatusType* status,
                unsigned length,
                const void* buffer)
```

заменяет `isc_put_segment()`.

4.

```
void cancel(StatusType* status)
```

заменяет `isc_cancel_blob()`. В случае успеха освобождает интерфейс.

5.

```
void close(StatusType* status)
```

заменяет `isc_close_blob()`. В случае успеха освобождает интерфейс.

6.

```
int seek(StatusType* status,
         int mode,
         int offset)
```

заменяет `isc_seek_blob()`.

IConfig

Назначение: Интерфейс `IConfig` — общий интерфейс файла конфигурации.

1.

```
IConfigEntry* find(StatusType* status, const char* name)
```

находит запись по имени.

2.

```
IConfigEntry* findValue(StatusType* status, const char* name, const char* value)
```

находит запись по имени и значению.

3.

```
IConfigEntry* findPos(StatusType* status, const char* name, unsigned pos)
```

находит запись по имени и позиции. Если файл конфигурации содержит строки:

```
Db=DBA
Db=DBB
Db=DBC
```

вызов `findPos(status, "Db", 2)` вернет запись со значением DBB.

IConfigManager

Назначение: Интерфейс `IConfigManager` — общий интерфейс для доступа к различным объектам конфигурации.

1.

```
const char* getDirectory(unsigned code)
```

возвращает местоположение соответствующего каталога в текущем экземпляре firebird. См. коды каталогов для этого вызова ниже.

2.

```
IFirebirdConf* getFirebirdConf()
```

возвращает интерфейс для доступа к значениям конфигурации по умолчанию (из `firebird.conf`).

3.

```
IFirebirdConf* getDatabaseConf(const char* dbName)
```

возвращает интерфейс для доступа к конфигурации, специфичной для базы данных (берёт в расчёт `firebird.conf` и соответствующую часть `database.conf`).

4.

```
IConfig* getPluginConfig(const char* configuredPlugin)
```

возвращает интерфейс для доступа к конфигурации именованного плагина.

5.

```
const char* getInstallDirectory()
```

возвращает каталог, в котором установлен firebird.

6.

```
const char* getRootDirectory()
```

возвращает корневой каталог текущего экземпляра, в случае с единственным экземпляром обычно совпадает с каталогом установки.

Коды каталогов:

- DIR_BIN — `bin` (утилиты наподобие isql, gbak, gstat);
- DIR_SBIN — sbin (fbguard и firebird сервер);
- DIR_CONF — каталог файлов конфигурации (`firebird.conf`, `databases.conf`, `plugins.conf`);
- DIR_LIB — `lib` (`fbclient`, `ib_util`);
- DIR_INC — `include` (`ibase.h`, `firebird/Interfaces.h`);
- DIR_DOC — каталог документации;
- DIR_UDF — UDF (`ib_udf`, `fbudf`);
- DIR_SAMPLE — каталог примеров;
- DIR_SAMPLEDB — каталог где расположена база данных примеров (`employee.fdb`);
- DIR_HELP — qli help (`help.fdb`);
- DIR_INTL — каталог библиотек интернационализаци (`fbintl`);
- DIR_MISC — различные файлы (как манифест деинсталлятор и другое);
- DIR_SECDB — каталог где хранится база данных безопасности (`securityN.fdb`);
- DIR_MSG — каталог где хранится файл сообщений (`firebird.msg`);
- DIR_LOG — каталог где хранится лог файл (`firebird.log`);
- DIR_GUARD — каталог где хранится блокировка хранителя (fb_guard);
- DIR_PLUGINS — директория плагинов ([lib]Engine12.{dll|so}).

IConfigEntry

Назначение: Интерфейс `IConfigEntry` — представляет запись (Key = Values с возможными подзаголовками (подзаписями)) в файле конфигурации firebird.

1.

```
const char* getName()
```

возвращает имя ключа.

2.

```
const char* getValue()
```

возвращает значение в качестве символьной строки.

3.

```
ISC_INT64 getIntValue()
```

обрабатывает значение как целое и возвращает его.

4.

```
FB_BOOLEAN getBoolValue()
```

обрабатывает значение как boolean и возвращает его.

5.

```
IConfig* getSubConfig(StatusType* status)
```

рассматривает подзаголовки как отдельный файл конфигурации и возвращает интерфейс IConfig для него.

IDtc

Назначение: Интерфейс `IDtc` — координатор распределенных транзакций. Используется для запуска распределенной (работает с двумя или более подключениями) транзакции. В отличие от пред-FB3-подхода, когда распределенная транзакция должна запускаться таким образом с самого начала, координатор распределенных транзакций FB3 позволяет также присоединить уже запущенные транзакции в единую распределенную транзакцию.

1.

```
ITransaction* join(StatusType* status, ITransaction* one, ITransaction* two)
```

объединяет 2 независимых транзакции в распределенную транзакцию. При успешном выполнении обе транзакции, переданные в `join()`, освобождаются, а указатели на них больше не должны использоваться.

2.

```
IDtcStart* startBuilder(StatusType* status)
```

возвращает IDtcStart интерфейс.

IDtcStart

Назначение: Интерфейс `IDtcStart` — заменяет массив структур TEB (передан `isc_start_multiple()` в ISC API). Этот интерфейс собирает подключения (и, вероятно, соответствующие TPB), для которых должна быть запущена распределённая транзакция.

1.

```
void addAttachment(StatusType* status, IAttachment* att)
```

добавляет подключение, транзакция для него будет запущена с TPB по умолчанию.

2.

```
void addWithTpb(StatusType* status, IAttachment* att, unsigned length, const unsigned cha
```

добавляет подключение и TPB, которые будут использоваться для запуска транзакции для этого подключения.

3.

```
ITransaction* start(StatusType* status)
```

начинает распределенную транзакцию для собранных подключений. При успехе возвращает интерфейс `IDtcStart`.

IEventCallback

Назначение: Интерфейс `IEventCallback` — заменяет функцию обратного вызова, используемую в вызове `isc_que_events()`. Должен быть реализован пользователем для отслеживания событий с помощью метода `IAttachment::queEvents()`.

1.

```
void eventCallbackFunction(unsigned length, const unsigned char* events)
```

вызывается каждый раз, когда происходит событие.

IEvents

Назначение: Интерфейс `IEvents` — заменяет идентификатор события при работе с мониторингом событий.

1.

```
void cancel(StatusType* status)
```

отменяет мониторинг событий, начатый в `IAttachment::queEvents()`.

IFirebirdConf

Назначение: Интерфейс `IFirebirdConf` — доступ к основной конфигурации firebird. Используется как для конфигурации по умолчанию, заданной конфигурацией `firebird.conf`, так и для каждой базы данных, скорректированной с помощью `database.conf`. Чтобы ускорить доступ к значениям конфигурации, вызовы, обращающиеся к фактическим значениям, используют целочисленный ключ вместо символьного имени параметра. Ключ стабилен во время работы сервера (т. е. плагин может получить его один раз и использовать для получения значения параметров конфигурации для разных баз данных).

1.
```
unsigned getKey(const char* name)
```
возвращает ключ для заданного имени параметра. ~0 (все биты равны 1) возвращается в случае когда такого параметра нет.

2.
```
ISC_INT64 asInteger(unsigned key)
```
возвращает значение целочисленного параметра.

3.
```
const char* asString(unsigned key)
```
возвращает значение строкового парамера

4.
```
FB_BOOLEAN asBoolean(unsigned key)
```
возвращает значение логичекого параметра. Стандартные аббревиатуры (1/true/t/yes/y) рассматриваются как true, все остальные — как false.

IMaster

Назначение: `IMaster` — основной интерфейс, с которого начинаются все операции с API-интерфейсом Firebird.

1.
```
IStatus* getStatus()
```

возвращает экземпляр интерфейса IStatus.

2.
```
IProvider* getDispatcher()
```
возвращает экземпляр интерфейса IProvider, реализованный YValve (основной экземпляр поставщика).

3.
```
IPluginManager* getPluginManager()
```
возвращает экземпляр интерфейса IPluginManager.

4.
```
ITimerControl* getTimerControl()
```
возвращает экземпляр интерфейса ITimerControl.

5.
```
IDtc* getDtc()
```
возвращает экземпляр интерфейса IDtc.

6.
```
IUtil* getUtilInterface()
```
возвращает экземпляр интерфейса IUtil.

7.
```
IConfigManager* getConfigManager()
```
возвращает экземпляр интерфейса IConfigManager.

IMessageMetadata

Назначение: Интерфейс `MessageMetadata` — частичный аналог XSQLDA (не содержит данных сообщений, присутствует только информация о формате сообщения). Используется в вызовах, связанных с выполнением операторов SQL.

1.
```
unsigned getCount(StatusType* status)
```

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

2.

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

возвращает имя поля.

3.

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

возвращает имя отношения (из которого выбрано данное поле).

4.

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

возвращает имя владельца отношения.

5.

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

возвращает псевдоним поля.

6.

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

вовзаращет SQL тип поля.

7.

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

возвращает true, если поле может принимать значение NULL.

8.

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

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

9.

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

возвращает максимальную длину поля.

10.

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

возвращает масштаб для числового поля.

11.

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

возвращает набор символов для символьных полей и текстового BLOB.

12.

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

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

13.

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

возвращает смещение NULL индикатора для поля в буфере сообщений.

14.

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

возвращает интерфейс IMetadataBuilder, инициализированный метаданными этого сообщения.

15.

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

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

IMetadataBuilder

Назначение: Интерфейс `IMetadataBuilder` — позволяет описать типы данных для существующих сообщений или создавать метаданные с самого начала.

1.

```
void setType(StatusType* status, unsigned index, unsigned type)
```

устанавливает SQL тип поля.

2.
```
void setSubType(StatusType* status, unsigned index, int subType)
```
устанавливает подтип BLOB поля.

3.
```
void setLength(StatusType* status, unsigned index, unsigned length)
```
установливает максимальную длину символьного поля.

4.
```
void setCharSet(StatusType* status, unsigned index, unsigned charSet)
```
устанавливает набор символов для символьного поля и текстового BLOB.

5.
```
void setScale(StatusType* status, unsigned index, unsigned scale)
```
устанавливает масштаб для числовых полей.

6.
```
void truncate(StatusType* status, unsigned count)
```
обрезает сообщение чтобы оно содержало не более *`count`* полей.

7.
```
void moveNameToIndex(StatusType* status, const char* name, unsigned index)
```
реорганизует поля в сообщении — перемещает поле с именем *`name`* в заданное положение.

8.
```
void remove(StatusType* status, unsigned index)
```
удаляет поле.

9.
```
unsigned addField(StatusType* status)
```

добавляет поле.

10.

```
IMessageMetadata* getMetadata(StatusType* status)
```

возвращает интерфейс IMessageMetadata, построенный этим построителем.

IOffsetsCallback

Назначение: Интерфейс IOffsetsCallback

1.

```
void setOffset(StatusType* status, unsigned index, unsigned offset, unsigned nullOffset)
```

уведомляет, что должны быть установлены смещения для поля/параметра с индексом *index*. Должен быть реализован пользователем при реализации интерфейса MessageMetadata и с использованием `IUtil::setOffsets()`.

IPluginConfig

Назначение: Интерфейс `IPluginConfig` — передается фабрике плагинов при создании экземпляра плагина (с конкретной конфигурацией).

1.

```
const char* getConfigFileName()
```

возвращает рекомендованное имя файла, в котором ожидается сохранение конфигурации для плагина.

2.

```
IConfig* getDefaultConfig(StatusType* status)
```

конфигурация плагина загруженная по стандартным правилам.

3.

```
IFirebirdConf* getFirebirdConf(StatusType* status)
```

возвращает главную конфигурацию Firebird с учётом настроек для базы данных, с которой будет работать новый экземпляр плагина.

4.

```
void setReleaseDelay(StatusType* status, ISC_UINT64 microSeconds)
```

используемые плагином для настройки рекомендуемой задержки, в течение которой модуль плагина не будет выгружен менеджером плагинов после освобождения последнего экземпляра плагина из этого модуля.

IPluginFactory

Назначение: Интерфейс `IPluginFactory` — должен быть реализован автором плагина при написании плагина.

1.
```
IPluginBase* createPlugin(StatusType* status, IPluginConfig* factoryParameter)
```
создает новый экземпляр плагина с переданной рекомендуемой конфигурацией.

IPluginManager

Назначение: Интерфейс `IPluginManager` — API диспетчера плагинов.

1.
```
void registerPluginFactory(unsigned pluginType,
                           const char* defaultName,
                           IPluginFactory* factory)
```
регистрирует именованную файбрику плагинов данного типа.

2.
```
void registerModule(IPluginModule* cleanup)
```
регистрирует модуль плагина.

3.
```
void unregisterModule(IPluginModule* cleanup)
```
разрегистрирует модуль плагина.

4.
```
IPluginSet* getPlugins(StatusType* status,
                       unsigned pluginType,
                       const char* namesList,
                       IFirebirdConf* firebirdConf)
```
возвращает интерфейс `IPluginSet`, предоставляющий доступ к списку плагинов данного типа. Имена включенных плагинов берутся из *`namesList`*, если отсутствует (NULL), то из настроек конфигурации для данного типа *`pluginType`*. Если указан параметр

firebirdConf, то он используется для всех целей конфигурации (включая получение списка плагинов и переход к методу `PluginFactory::createPlugin()`), если отсутствует (NULL), то используется настройка по умолчанию (из `firebird.conf`).

5.

```
IConfig* getConfig(StatusType* status, const char* filename)
```

возвращает интерфейс `IConfig` для заданного имени файла конфигурации. Может использоваться плагинами для доступа к файлам конфигурации со стандартным форматом, но не с именем по умолчанию.

6.

```
void releasePlugin(IPluginBase* plugin)
```

освобождение данного плагина. Должен использоваться для плагинов вместо простой `release()` из-за необходимости выполнять дополнительные действия с владельцем плагина до фактического выпуска.

Константы, определенные интерфейсом IPluginManager (типы плагинов):

- TYPE_PROVIDER
- TYPE_AUTH_SERVER
- TYPE_AUTH_CLIENT
- TYPE_AUTH_USER_MANAGEMENT
- TYPE_EXTERNAL_ENGINE
- TYPE_TRACE
- TYPE_WIRE_CRYPT
- TYPE_DB_CRYPT
- TYPE_KEY_HOLDER

IPluginModule

Назначение: Интерфейс `IPluginModule` — представляет модуль плагина (динамическая библиотека). Должен быть реализован автором плагина в каждом модуле плагина (по одному экземпляру на модуль).

1.

```
void doClean()
```

вызывается диспетчером плагинов перед нормальной выгрузкой модуля плагина.

IPluginSet

Назначение: Интерфейс `IPluginSet` — представляет собой набор плагинов данного типа. Обычно используется внутренним кодом Firebird, но рекомендуется для использования в плагинах, загружающих другие плагины.

1.

```
const char* getName()
```

возвращает имя текущего плагина в наборе.

2.

```
const char* getModuleName()
```

возвращает имя модуля текущего плагина в наборе (в простом случае совпадает с именем плагина).

3.

```
IPluginBase* getPlugin(StatusType* status)
```

возвращает экземпляр текущего плагина, возвращенный интерфейс должен быть преобразован в основной интерфейс плагина запрошенного типа в методе `IPluginManager::getPlugins()`. Возвращает NULL, если в наборе больше нет плагинов. Счётчик ссылок плагина, возвращаемого этой функцией, увеличивается при возврате — не забудьте использовать метод `releasePlugin()` интерфейса IPluginManager для плагинов, возвращаемых этим методом.

4.

```
void next(StatusType* status)
```

устанавливает переключатель для перехода к следующему плагину из списка.

5.

```
void set(StatusType* status, const char* list)
```

сбрасывает интерфейс: заставить его работать со списком плагинов, предоставляемых параметром списка. Тип плагинов остается неизменным.

IProvider

Назначение: Интерфейс `IPluginModule` — основной интерфейс для начала доступа к базе данных/сервису.

1.

```
IAttachment* attachDatabase(StatusType* status,
                            const char* fileName,
                            unsigned dpbLength,
                            const unsigned char* dpb)
```

заменяет `isc_attach_database()`.

2.

```
IAttachment* createDatabase(StatusType* status,
                            const char* fileName,
                            unsigned dpbLength,
                            const unsigned char* dpb)
```

заменяет `isc_create_database()`.

3.

```
IService* attachServiceManager(StatusType* status,
                               const char* service,
                               unsigned spbLength,
                               const unsigned char* spb)
```

заменяет `isc_service_attach()`.

4.

```
void shutdown(StatusType* status, unsigned timeout, const int reason)
```

заменяет `fb_shutdown()`.

5.

```
void setDbCryptCallback(IStatus* status, ICryptKeyCallback* cryptCallback)
```

устанавливает интерфейс обратного вызова шифрования базы данных, который будет использоваться для последующих подключений к базе данных и сервисам. См. ... для подробностей.

IResultSet

Назначение: Интерфейс `IResultSet` — заменяет (с расширенной функциональностью) некоторые функции `isc_stmt_handle`. Этот интерфейс возвращается вызовом `openCursor()` из IAttachment или IStatement. Все вызовы `fetch...`, кроме `fetchNext()`, работают только для двунаправленного (открытого с флагом `CURSOR_TYPE_SCROLLABLE`) курсора.

1.

```
int fetchNext(StatusType* status, void* message)
```

выбирает следующую запись, заменяет `isc_dsql_fetch()`. Этот метод (и другие методы выборки) возвращает код завершения `Status::RESULT_NO_DATA` при достижении EOF, и статус `Status::RESULT_OK` при успешном завершении.

2.

```
int fetchPrior(StatusType* status, void* message)
```

выбирает предыдущую запись.

3.

```
int fetchFirst(StatusType* status, void* message)
```

выбирает первую запись.

4.

```
int fetchLast(StatusType* status, void* message)
```

выбирает последнюю запись.

5.

```
int fetchAbsolute(StatusType* status, int position, void* message)
```

получает запись по абсолютной позиции в наборе результатов.

6.

```
int fetchRelative(StatusType* status, int offset, void* message)
```

извлекает запись по положению относительно текущей.

7.

```
FB_BOOLEAN isEof(StatusType* status)
```

проверка EOF.

8.

```
FB_BOOLEAN isBof(StatusType* status)
```

проверка BOF.

9.

```
IMessageMetadata* getMetadata(StatusType* status)
```

возвращает метаданные для сообщений в наборе результатов, особенно полезно, когда набор результатов открывается вызовом `IAttachment::openCursor()` с параметром

формата вывода метаданных равным NULL (это единственный способ получить формат сообщения в данном случае).

10.

```
void close(IStatus* status)
```

закрывает набор результатов, освобождает интерфейс в случае успеха.

IService

Назначение: Интерфейс `IService` — заменяет `isc_svc_handle`.

1.

```
void detach(StatusType* status)
```

закрывает подключение к менеджеру сервисов, при успехе освобождает интерфейс. Заменяет `isc_service_detach()`.

2.

```
void query(StatusType* status,
           unsigned sendLength,
           const unsigned char* sendItems,
           unsigned receiveLength,
           const unsigned char* receiveItems,
           unsigned bufferLength,
           unsigned char* buffer)
```

отправляет и запрашивает информацию в/из службы, при этом *`receiveItems`* могут использоваться как для запущенных служб, так и для получения различной информации по всему серверу. Заменяет `isc_service_query()`.

3.

```
void start(StatusType* status,
           unsigned spbLength,
           const unsigned char* spb)
```

запускает утилиту в диспетчере служб. Заменяет `isc_service_start()`.

IStatement

Назначение: Интерфейс `IStatement` — заменяет (частично) `isc_stmt_handle`.

1.

```
void getInfo(StatusType* status,
             unsigned itemsLength,
```

```
            const unsigned char* items,
            unsigned bufferLength,
            unsigned char* buffer)
```

заменяет `isc_dsql_sql_info()`.

2.

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

тип оператора, в настоящее время можно найти только в источниках firebird в `dsql/dsql.h`.

3.

```
const char* getPlan(StatusType* status, FB_BOOLEAN detailed)
```

возвращает план выполнения оператора.

4.

```
ISC_UINT64 getAffectedRecords(StatusType* status)
```

возвращает количество записей, которые затронуты оператором.

5.

```
IMessageMetadata* getInputMetadata(StatusType* status)
```

возвращает метаданные параметров.

6.

```
IMessageMetadata* getOutputMetadata(StatusType* status)
```

возвращает метаданные значений выходных данных.

7.

```
ITransaction* execute(StatusType* status,
                      ITransaction* transaction,
                      IMessageMetadata* inMetadata,
                      void* inBuffer,
                      IMessageMetadata* outMetadata,
                      void* outBuffer)
```

выполняет любую инструкцию SQL, за исключением тех, что возвращают нескольких строк данных. Частичный аналог `isc_dsql_execute2()` — вход и выход XSLQDA заменены на входные и выходные сообщения с соответствующими буферами.

8.

```
IResultSet* openCursor(StatusType* status,
                       ITransaction* transaction,
                       IMessageMetadata* inMetadata,
                       void* inBuffer,
                       IMessageMetadata* outMetadata,
                       unsigned flags)
```

выполняет оператор SQL, потенциально возвращающий несколько строк данных. Возвращает интерфейс `IResultSet`, который должен использоваться для извлечения этих данных. Формат выходных данных определяется параметром *`outMetadata`*, если указано NULL, то будет использоваться формат по умолчанию.

9.

```
void setCursorName(StatusType* status, const char* name)
```

заменяет `isc_dsql_set_cursor_name()`.

10.

```
void free(StatusType* status)
```

уничтожает оператор, освобождает интерфейс в случае успеха.

11.

```
unsigned getFlags(StatusType* status)
```

возвращает флаги, описывающие, как должен выполняться этот оператор, упрощенная замена метода `getType()`.

Константы, определенные интерфейсом IStatement

Флаги `IAttachment::prepare()`:

- `PREPARE_PREFETCH_NONE` — константа, чтобы пропускать флаги, значение 0.

Следующие флаги могут быть объеденены с помощью `OR` для полчение желаемого эффекта:

1. `PREPARE_PREFETCH_TYPE`
2. `PREPARE_PREFETCH_INPUT_PARAMETERS`
3. `PREPARE_PREFETCH_OUTPUT_PARAMETERS`
4. `PREPARE_PREFETCH_LEGACY_PLAN`
5. `PREPARE_PREFETCH_DETAILED_PLAN`
6. `PREPARE_PREFETCH_AFFECTED_RECORDS`
7. `PREPARE_PREFETCH_FLAGS` (флаги возвращаемые методом `getFlags()`)

Для наиболее часто испольуемых комбинаций флагов можно использовать константы:

1. `PREPARE_PREFETCH_METADATA`
2. `PREPARE_PREFETCH_ALL`

Значения возвращаемые методом `getFlags()`:

1. `FLAG_HAS_CURSOR` — используйте `openCursor()` для выполнения этого оператора, а не `execute()`
2. `FLAG_REPEAT_EXECUTE` — когда подготовленный оператор может выполняться много раз с разными параметрами.

Флаги передаваемые в `openCursor()`:

- `CURSOR_TYPE_SCROLLABLE` — открывается двунаправленный курсор.

IStatus

Назначение: Интерфейс `IStatus` — заменяет `ISC_STATUS_ARRAY`. Функциональность расширена — статус имеет отдельный доступ к векторам ошибок и предупреждений, может содержать векторы неограниченной длины, самостоятельно хранит строки, используемые в векторах, не имеет необходимости в кольцевом буфере строк. В C++ `IStatus` всегда используется в оболочке состояния, C++ API предоставляет две разные оболочки, имеющие различное поведение, когда ошибка возвращается из вызова API. Интерфейс сведен к минимуму (методы, такие как преобразование его в текст, перемещены в интерфейс IUtil), чтобы упростить его реализацию пользователями при необходимости.

1. ```
   void init()
   ```
   очищает интерфейс, установливая его в исходное состояние.
2. ```
   unsigned getState()
   ```
   возвращает текущее состояние интерфейса, возвращаемые флаги могут быть объеденены с помощью `OR`.
3. ```
   void setErrors2(unsigned length, const intptr_t* value)
   ```
   установливает содержимое вектора ошибок с длиной, явно указанной в вызове.
4. ```
   void setWarnings2(unsigned length, const intptr_t* value)
   ```
   установливает содержимое вектора предупреждений с длиной, явно указанной в вызове.
5. ```
   void setErrors(const intptr_t* value)
   ```
   ```

установливает содержимое вектора ошибок, длина определяется контекстом значения.

6.

```
void setWarnings(const intptr_t* value)
```

установливает содержимое вектора предупреждений, длина определяется контекстом значения.

7.

```
const intptr_t* getErrors()
```

возвращает вектор ошибок.

8.

```
const intptr_t* getWarnings()
```

возвращает вектор предупреждений.

9.

```
IStatus* clone()
```

создаёт клон текущего интерфейса.

Константы определённые в IStatus

Флаги, установленные в значение, возвращаемые методом `getState()`:

- `STATE_WARNINGS`
- `STATE_ERRORS`

Коды завершения:

- `RESULT_ERROR`
- `RESULT_OK`
- `RESULT_NO_DATA`
- `RESULT_SEGMENT`

ITimer

Назначение: Интерфейс `ITimer` — пользовательский таймер. Интерфейс обратного вызова, который должен быть реализован пользователем для использования таймера Firebird.

1.

```
void handler()
```

метод вызывается, когда таймер звонит (или когда сервер выключается).

ITimerControl

Назначение: Интерфейс `ITimerControl` — очень простая и не слишком точная реализация таймера. Прибыл сюда, потому что существующие таймеры очень зависимы от ОС и могут использоваться в программах, которые требуют переносимости и не требуют действительно высокоточного таймера. В частности, выполнение данного таймера может быть отложено, если другой не был завершен в тот момент, когда данный таймер должен сигнализировать.

1.
```
void start(StatusType* status, ITimer* timer, ISC_UINT64 microSeconds)
```
запуск ITimer после сигнала (в микросекундах, 10^{-6} секунд). Таймер будет разбужен только один раз после этого вызова.

2.
```
void stop(StatusType* status, ITimer* timer)
```
остановка ITimer. Не стоит останавливать не запущенный таймер, что позволит избежать проблем с гонками между `stop()` и сигналом таймером.

ITransaction

Назначение: Интерфейс `ITransaction` — заменяет `isc_tr_handle`.

1.
```
void getInfo(StatusType* status,
             unsigned itemsLength,
             const unsigned char* items,
             unsigned bufferLength,
             unsigned char* buffer)
```
заменяет `isc_transaction_info()`.

2.
```
void prepare(StatusType* status,
             unsigned msgLength,
             const unsigned char* message)
```
заменяет `isc_prepare_transaction2()`.

3.

```
void commit(StatusType* status)
```

заменяет `isc_commit_transaction()`.

4.

```
void commitRetaining(StatusType* status)
```

заменяет `isc_commit_retaining()`.

5.

```
void rollback(StatusType* status)
```

заменяет `isc_rollback_transaction()`.

6.

```
void rollbackRetaining(StatusType* status)
```

заменяет `isc_rollback_retaining()`.

7.

```
void disconnect(StatusType* status)
```

заменяет `fb_disconnect_transaction()`.

8.

```
ITransaction* join(StatusType* status, ITransaction* transaction)
```

соединяет текущую транзакцию и транзакцию переданную как параметр в единую распределённую транзакцию (с импользованием Dtc). При успешном выполнении текущая транзакция и тразакция переданная в качестве параметра освобождаются и больше не должны использоваться.

9.

```
ITransaction* validate(StatusType* status, IAttachment* attachment)
```

этот метод используется для поддержки координатора распределенных транзакций.

10.

```
ITransaction* enterDtc(StatusType* status)
```

этот метод используется для поддержки координатора распределенных транзакций.

IVersionCallback

Назначение: Интерфейс `IVersionCallback` — обратный вызов для `IUtil::getFbVersion()`.

1.
```
void callback(StatusType* status, const char* text)
```
вызвается движком firebird для каждой строки в многострочной версии отчета. Позволяет печатать эти строки одина за другой, помещать их в поле сообщения в любом графическом интерфейсе и т. д.

IUtil

Назначение: Интерфейс `IUtil` — различные вспомогательные методы, требуемые здесь или там.

1.
```
void getFbVersion(StatusType* status,
                  IAttachment* att,
                  IVersionCallback* callback)
```
производят длинный и красивый отчет о версии для firebird. Это можно увидеть в ISQL при вызове с ключом `-Z`.

2.
```
void loadBlob(StatusType* status,
              ISC_QUAD* blobId,
              IAttachment* att,
              ITransaction* tra,
              const char* file,
              FB_BOOLEAN txt)
```
загрузка BLOB из файла.

3.
```
void dumpBlob(StatusType* status,
              ISC_QUAD* blobId,
              IAttachment* att,
              ITransaction* tra,
              const char* file,
              FB_BOOLEAN txt)
```
сохраняет BLOB в файл.

4.

```
void getPerfCounters(StatusType* status,
                     IAttachment* att,
                     const char* countersSet,
                     ISC_INT64* counters)
```

получает статистику для данного подключения.

5.

```
IAttachment* executeCreateDatabase(StatusType* status,
                                   unsigned stmtLength,
                                   const char* creatDBstatement,
                                   unsigned dialect,
                                   FB_BOOLEAN* stmtIsCreateDb)
```

выполняет инструкцию `CREATE DATABASE ...` — трюк ISC с NULL дескриптором оператора не работает с интерфейсами.

6.

```
void decodeDate(ISC_DATE date,
                unsigned* year,
                unsigned* month,
                unsigned* day)
```

заменяет `isc_decode_sql_date()`.

7.

```
void decodeTime(ISC_TIME time,
                unsigned* hours,
                unsigned* minutes,
                unsigned* seconds,
                unsigned* fractions)
```

заменяет `isc_decode_sql_time()`.

8.

```
ISC_DATE encodeDate(unsigned year, unsigned month, unsigned day)
```

заменяет `isc_encode_sql_date()`.

9.

```
ISC_TIME encodeTime(unsigned hours,
                    unsigned minutes,
                    unsigned seconds,
                    unsigned fractions)
```

заменяет `isc_encode_sql_time()`.

10.

```
unsigned formatStatus(char* buffer, unsigned bufferSize, IStatus* status)
```

заменяет `fb_interpret()`. Размер буфера, переданного в этот метод, не должен быть меньше 50 байт.

11.

```
unsigned getClientVersion()
```

возвращает целое число, содержащее основную версию в байте 0 и младшую версию в байте 1.

12.

```
IXpbBuilder* getXpbBuilder(StatusType* status,
                           unsigned kind,
                           const unsigned char* buf,
                           unsigned len)
```

возвращает интерфейс IXpbBuilder. Допустимые *`kind`* перечислены в `IXpbBuilder`.

13.

```
unsigned setOffsets(StatusType* status,
                    IMessageMetadata* metadata,
                    IOffsetsCallback* callback)
```

устанавливает допустимые смещения в IMessageMetadata. Выполняет вызовы для обратного вызова в IOffsetsCallback для каждого поля/параметра.

IXpbBuilder

Назначение: Интерфейс `IXpbBuilder`

1.

```
void clear(StatusType* status)
```

сбрасывает построитель в пустое состояние.

2.

```
void removeCurrent(StatusType* status)
```

удаляет текущий clumplet.

3.

```
void insertInt(StatusType* status, unsigned char tag, int value)
```

вставляет clumplet со значением, представляющим целое число в сетевом формате.

4.

```
void insertBigInt(StatusType* status, unsigned char tag, ISC_INT64 value)
```

вставляет clumplet со значением, представляющим 64-битное целое число в сетевом формате.

5.

```
void insertBytes(StatusType* status, unsigned char tag, const void* bytes, unsigned lengt
```

вставляет clumplet со значением, содержащим переданные байты.

6.

```
void insertTag(StatusType* status, unsigned char tag)
```

вставляет clumplet без значения.

7.

```
FB_BOOLEAN isEof(StatusType* status)
```

проверяет, нет ли текущего clumplet.

8.

```
void moveNext(StatusType* status)
```

переходит к следующему clumplet.

9.

```
void rewind(StatusType* status)
```

переходит к первому clumplet.

10.

```
FB_BOOLEAN findFirst(StatusType* status, unsigned char tag)
```

находит первый clumplet с данным тегом.

11.

```
FB_BOOLEAN findNext(StatusType* status)
```

находит следующий clumplet с заданным тегом.

12.

```
unsigned char getTag(StatusType* status)
```

возвращает тег для текущего clumplet.

13.

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

возвращает длину текущего значения clumplet.

14.

```
int getInt(StatusType* status)
```

возвращает значение текущего clumplet как целое.

15.

```
SC_INT64 getBigInt(StatusType* status)
```

возвращает значение текущего clumplet как 64-битное целое число.

16.

```
const char* getString(StatusType* status)
```

возвращает значение текущего clumplet как указатель на нуль-терминальную строку (указатель действителен до следующего вызова этого метода).

17.

```
const unsigned char* getBytes(StatusType* status)
```

возвращает значение текущего clumplet как указатель на unsigned char.

18.

```
unsigned getBufferLength(StatusType* status)
```

возвращает длину блока параметров.

19.

```
const unsigned char* getBuffer(StatusType* status)
```

возвращает указатель на блок параметров.

Константы, определенные интерфейсом IXpbBuilder

Допустимые типы построителей:

- `DPB`
- `SPB_ATTACH`
- `SPB_START`
- `TPB`

Плагин шифрования данных передаваемых по сети

Алгоритмы, выполняющие шифрование данных для разных целей, хорошо известны на протяжении многих лет. Единственной «маленькой» типичной проблемой остается то, где можно получить секретный ключ, который будет использоваться этим алгоритмом. К счастью для шифрования сетевого трафика есть одно хорошее решение — уникальный ключ шифрования должен быть сгенерирован плагином аутентификации. По крайней мере, по умолчанию плагин SRP может создать такой ключ. Этот ключ устойчив к атакам, в том числе с помощью "человека в середине" (man-in-the-middle). Поэтому для плагина шифрования сетевого трафика был выбран следующий способ предоставления ключей: получать его от плагина проверки подлинности (аутентификации). (В случае, если используемый плагин аутентификации не может предоставить ключ, псевдоплагин может быть добавлен в списки AuthClient и AuthServer для создания ключей, что-то вроде двух асимметричных пар приватного и публичного.)

ICryptKey

Назначение: Интерфейс `ICryptKey` используется для хранения ключа, предоставленного плагином аутентификации, и передает его в плагин шифрования сетевого трафика. Этот интерфейс следует использовать следующим образом: когда плагин аутентификации сервера или клиента готов предоставить ключ, то он запрашивает `IServerBlock` или `IClientBlock` для создания нового интерфейса `ICryptKey` и хранит в нем ключ. Подходящий для `IWireCryptPlugin` тип ключа будет выбран Firebird и передан этому интерфейсу.

1.

```
void setSymmetric(StatusType* status,
                  const char* type,
                  unsigned keyLength,
                  const void* key)
```

сохраняет симметричный ключ заданного типа.

2.

```
void setAsymmetric(StatusType* status,
                   const char* type,
                   unsigned encryptKeyLength,
                   const void* encryptKey,
                   unsigned decryptKeyLength,
                   const void* decryptKey)
```

сохраняет пару асимметричных ключей заданного типа.

3.

```
const void* getEncryptKey(unsigned* length)
```

возвращает ключ для шифрования.

4.

```
const void* getDecryptKey(unsigned* length))
```

возвращает ключ для дешифрования (в случае симметричного ключа получается тот же результат, что и `getEncryptKey()`).

IWireCryptPlugin

Назначение: Интерфейс `IWireCryptPlugin` является основным интерфейсом плагина сетевого шифрования. Как и любой другой такой интерфейс, он должен быть реализован автором плагина.

1.

```
const char* getKnownTypes(StatusType* status)
```

возвращает список допустимых ключей разделённых пробелами/табуляциями/запятыми.

2.

```
void setKey(StatusType* status, ICryptKey* key)
```

плагин должен использовать ключ, переданный ему этим вызовом.

3.

```
void encrypt(StatusType* status,
             unsigned length,
             const void* from,
             void* to)
```

шифрует пакет, который должен быть отправлен по сети.

4.

```
void decrypt(StatusType* status,
             unsigned length,
             const void* from,
             void* to)
```

расшифровывает пакет, полученный из сети.

Плагин аутентификации на серверной стороне

Плагин аутентификации содержит две требуемые части — клиентскую и серверверную, а также может содержать связанную с ним третью часть — менеджер пользователей. В процессе аутентификации клиент Firebird вызывает клиентский плагин и отправляет сгенерированные им данные на сервер, затем сервер вызывает серверный плагин и отправляет сгенерированные им данные клиенту. Этот процесс повторяется до тех пор, пока оба плагина возвращают код `AUTH_MORE_DATA`. `AUTH_SUCCESS`, возвращенный на стороне сервера, означает успешную аутентификацию, `AUTH_FAILED` с любой стороны — немедленное прерывание итеративного процесса и отказ, сообщаемый клиенту, `AUTH_CONTINUE` означает, что должен быть проверен следующий плагин из списка настроенных плагинов проверки подлинности.

Нет выделенных примеров плагинов для аутентификации, но в исходных кодах firebird в каталоге `src/auth` можно найти плагин *AuthDbg*, с помощью которого можно учиться на тривиальном примере (без сложных вычислений как, например, в *Srp*, и без вызовов сумасшедших функций WinAPI, таких как в *AuthSspi*), как клиентская и серверная сторона выполняют аутентификацию (рукопожатие).

IAuth

Назначение: Интерфейс `IAuth` не содержит методов, только некоторые константы, определяющие коды, возвращаются из метода `authenticate()` в `IClient` и `IServer`.

- `AUTH_FAILED`
- `AUTH_SUCCESS`
- `AUTH_MORE_DATA`
- `AUTH_CONTINUE`

IWriter

Назначение: Интерфейс `IWriter` — записывает блок параметров аутентификации.

1.

```
void reset()
```

очищает целевой блок.

2.

```
void add(StatusType* status, const char* name)
```

добавляет имя логина.

3.

```
void setType(StatusType* status, const char* value)
```

устанавливает тип добавленного логина (пользователь, роль, группа и т.д.).

4.

```
void setDb(StatusType* status, const char* value)
```

устаналивает базу данных безопасности, в которой была выполнена аутентификация.

IServerBlock

Назначение: Интерфейс `IServerBlock` используется серверной частью модуля аутентификации для обмена данными с клиентом.

1.

```
const char* getLogin()
```

возвращает имя пользователя, переданное от клиента.

2.

```
const unsigned char* getData(unsigned* length)
```

возвращает данные аутентификации, переданные от клиента.

3.

```
void putData(StatusType* status, unsigned length, const void* data)
```

передаёт данные аутентификации клиенту.

4.

```
ICryptKey* newKey(StatusType* status)
```

создаёт новый ключ шифрования и добавляет его в список доступных для плагинов шифрования сетевого трафика.

IServer

Назначение: Интерфейс `IServer` является основным интерфейсом серверной части плагина аутентификации.

1.

```
int authenticate(StatusType* status,
                 IServerBlock* sBlock,
                 IWriter* writerInterface)
```

выполняет один этап аутентификации. Обмен данными с клиентом осуществляется с использованием интерфейса *sBlock*. Когда создается некоторый элемент аутентификации, его следует добавить в блок аутентификации с помощью *writerInterface*. Возможные значения возврата определяются в интерфейсе IAuth.

Плагин аутентификации на клиентской стороне

IClientBlock

Назначение: Интерфейс IClientBlock используется клиентской стороной модуля аутентификации для обмена данными с сервером.

1.

```
const char* getLogin()
```

возвращает имя пользователя, если оно присутствует в DPB.

2.

```
const char* getPassword()
```

возвращает пароль, если он присутствует в DPB.

3.

```
const unsigned char* getData(unsigned* length)
```

возвращает данные аутентификации, переданные с сервера.

4.

```
void putData(StatusType* status, unsigned length, const void* data)
```

передаёт данные аутентификации на сервер.

5.

```
ICryptKey* newKey(StatusType* status)
```

создаёт новый ключ шифрования и добавляет его в список доступных для плагинов шифрования сетевого трафика.

IClient

Назначение: Интерфейс `IClient` является основным интерфейсом клиентской стороны модуля аутентификации.

1.
```
int authenticate(StatusType* status,
                 IClientBlock* cBlock)
```
выполняет один этап аутентификации. Обмен данными с сервером осуществляется с использованием интерфейса *`cBlock`*. Возможные значения возврата определяются в интерфейсе `IAuth`. `AUTH_SUCCESS` обрабатывается клиентской стороной как `AUTH_MORE_DATA` (т.е. клиент отправляет сгенерированные данные на сервер и ждет ответа от него).

Плагин управления пользователями

Этот плагин активно связан с серверной частью проверки подлинности — он подготавливает список пользователей для плагина аутентификации. Для каждого плагина проверки подлинности требуется менеджер пользователей — некоторые из них могут получить доступ к списку пользователей, созданных с использованием не Firebird программного обеспечения (например, *AuthSspi*). Запись, описывающая пользователя, состоит из нескольких полей, и поддерживать выполнение нескольких операций, таких как добавление пользователя, изменение пользователя, получение списка пользователей и т. д. Плагин должен уметь интерпретировать команды, полученные в интерфейсе `IUser`.

IUserField

Назначение: Интерфейс `IUserField` не используется как автономный интерфейс, он является базовым для `ICharUserField` и `IIntUserField`.

1.
```
int entered()
```
возвращает ненулевое значение, если было введено (присвоено) значение для поля.

2.
```
int specified()
```
возвращает ненулевое значение, если для поля было присвоено значение NULL.

3.
```
void setEntered(StatusType* status, int newValue)
```
устанавливает entered флаг в 0 или ненулевое значение для поля. Нет способа назначить NULL для поля, потому что он никогда не требуется. NULL, если они используются,

назначаются реализациями интерфейсами и, следовательно, имеют полный доступ к их внутренним элементам.

ICharUserField

Интерфейс `ICharUserField`:

1.
```
const char* get()
```
возвращает значение поля как C-строку (\0 терминальную).

2.
```
void set(StatusType* status, const char* newValue)
```
присваивает значение полю. Устанавливает флаг entered в true.

IIntUserField

Интрефейс `IIntUserField`:

1.
```
int get()
```
возвращает значение поля.

2.
```
void set(StatusType* status, int newValue)
```
присваивает значение полю. Устанавливает флаг entered в true.

IUser

Назначение: Интерфейс `IUser` — это список методов доступа к полям, включенным в запись о пользователе.

1.
```
unsigned operation()
```
код операции (см. список ниже).

2.
```
ICharUserField* userName()
```

имя пользователя.

3.
```
ICharUserField* password()
```
пароль.

4.
```
ICharUserField* firstName()
```
это и 2 следующие компоненты полного имени пользователя.

5.
```
ICharUserField* lastName()
```

6.
```
ICharUserField* middleName()
```

7.
```
ICharUserField* comment()
```
комментарий (из SQL оператора COMMENT ON USER IS …).

8.
```
ICharUserField* attributes()
```
теги в форме `tag1=val1, tag2=val2, …, tagN=valN`. Val может быть пустым, что означает, что тег будет удален.

9.
```
IIntUserField* active()
```
изменяет настройку ACTIVE/INACTIVE для пользователя.

10.
```
IIntUserField* admin()
```

устанавливает/отменяет права администратора для пользователя.

11.

```
void clear(StatusType* status)
```

устанавливает, что все поля не введены и не указаны.

Константы, определенные пользовательским интерфейсом — действующие коды операций.

- `OP_USER_ADD` — добавление пользователя.
- `OP_USER_MODIFY` — редактирование пользователя.
- `OP_USER_DELETE` — удаление пользователя.
- `OP_USER_DISPLAY` — отображение пользователя.
- `OP_USER_SET_MAP` — включение отображения администраторов Windows на роль `RDB$ADMIN`.
- `OP_USER_DROP_MAP` — выключение отображения администраторов Windows на роль `RDB$ADMIN`.

IListUsers

Назначение: Интерфейс `IListUsers` — это обратный вызов, используемый плагином проверки подлинности при запросе списка пользователей. Плагин заполняет интерфейс `IUser` для всех элементов в списке пользователей один за другим и для каждого пользователя вызывает метод `list()` этого интерфейса.

1.

```
void list(StatusType* status, IUser* user)
```

функция обратного вызова. Реализация может делать полученными данными то что хочет. Например, она может поместить данные из пользовательского параметра в выходной поток сервиса или разместить в специальных таблицах SEC$ группы.

ILogonInfo

Назначение: Интерфейс `ILogonInfo` содержит данные, переданные плагину управления пользователями лдя подключения к базе данных безопасности с действительными учётыми данным.

1.

```
const char* name()
```

возвращает имя пользователя текущего подключения.

2.

```
const char* role()
```

возвращает активную роль текущего подключения.

3.

```
const char* networkProtocol()
```

возвращает сетевой протокол текущего подключения. В настоящее время не используется плагинами.

4.

```
const char* remoteAddress()
```

возвращает удаленный адрес текущего подключения. В настоящее время не используется плагинами.

5.

```
const unsigned char* authBlock(unsigned* length)
```

возвращает блок аутентификации текущего подключения. Если не NULL переписывает имя пользователя.

IManagement

Назначение: Интерфейс `IManagement` является основным интерфейсом плагина управления пользователями.

1.

```
void start(StatusType* status, ILogonInfo* logonInfo)
```

запускает плагин, при необходимости он подключается к базе данных безопасности для управления пользователями (использовать это или нет это решение зависящее от плагинов), используя учетные данные из *`logonInfo`*.

2.

```
int execute(StatusType* status, IUser* user, IListUsers* callback)
```

выполняет команду, предоставляемую методом `operation()` параметра *`user`*. При необходимости будет использоваться интерфейс обратного вызова. Параметр *`callback`* может иметь значение NULL для команд, не требующих получения списка пользователей.

3.

```
void commit(StatusType* status)
```

подтверждает изменения, выполненные вызовами метода `execute()`.

4.

```
void rollback(StatusType* status)
```

отменяет изменения, выполненные вызовами метода `execute()`.

Плагин шифрования базы данных

Возможность шифрования базы данных присутствовала в Firebird со времён Interbase, но соответствующие места в коде были закомментированы. Реализация была сомнительной — ключ шифрования всегда отправлялся от клиента в DPB, не было сделано попыток скрыть его от внешнего мира, и не предлагалось путей для шифрования существующих баз данных. Firebird 3.0 решает большинство проблем, за исключением, вероятно, худшей — как управлять ключами шифрования. Мы предлагаем различные типы решений, но они требует усилий в плагинах, т. е. нет красивого способа работы с ключами, например, для плагинов шифрования сетевого трафика.

Перед запуском с собственным плагином шифрования базы данных следует принять во внимание следующее. Мы видим два основных случая для которых использзуется шифрование базы данных — во-первых, может потребоваться избежать утечки данных, если сервер базы данных физически украден, а во-вторых, оно может использоваться для защиты данных в базе данных, которые распространяется вместе со специальным приложением, обращающимся к этим данным. Требования к этим случаям совершенно разные. В первом случае мы можем доверять серверу базы данных, что он не модифицирован, чтобы красть ключи, переданные в плагин безопасности, то есть мы ожидаем, что этот ключ не будет отправлен на неподходящий сервер. Во втором случае сервер может быть каким-то образом модифицирован для кражи ключей (если они передаются из приложения в плагин через код сервера) или даже данных (в качестве последнего места для снятия дампов из кеша, где они находятся в незашифрованном виде). Поэтому ваш плагин должен убедиться, что он работает с не измененными двоичными файлами Firebird и вашим приложением перед отправкой ключа в плагин, например, плагин может потребоваться от них какой-то цифровой подписи.Кроме того, если используется сетевой доступ к серверу, то хорошей идеей является проверка того, что сетевой канал зашифрован (разбор вывода IUtil::getFbVersion()) или используется собственный ключ шифрования. Вся эта работа должна выполняться в плагине (и в приложении, работающим с ним), то есть алгоритм шифрования блока базы данных сам по себе может оказаться наиболее простой частью плагина шифрования базы данных, особенно когда для него используется некоторая стандартная библиотека.

ICryptKeyCallback

Назначение: Интерфейс `ICryptKeyCallback` должен обеспечивать передачу ключа шифрования в плагин шифрования базы данных или плагин хранителя ключа.

1.

```
unsigned callback(unsigned dataLength,
                  const void* data,
```

```
                  unsigned bufferLength,
                  void* buffer)
```

при выполнении обратного вызова информация передается в обоих направлениях. Источник ключа получает *dataLength* байт данных и может отправлять *bufferLength* байт в буфер. Возвращает фактическое количество байтов, помещенных в буфер.

IDbCryptInfo

Назначение: Интерфейс `IDbCryptInfo` передается движку `IDbCryptPlugin`. Плагин может сохранить этот интерфейс и использовать, когда это необходимо, для получения дополнительной информации о базе данных.

1.

```
const char* getDatabaseFullPath(StatusType* status)
```

возвращает полное (включая путь) имя первичного файла базы данных.

IDbCryptPlugin

Назначение: Интерфейс `IDbCryptPlugin` является основным интерфейсом плагина шифрования базы данных.

1.

```
void setKey(StatusType* status,
            unsigned length,
            IKeyHolderPlugin** sources,
            const char* keyName)
```

используется для предоставления информации плагину шифрования базы данных о ключе шифрования. Firebird никогда не передает ключи для этого типа плагина напрямую. Вместо этого массив `IKeyHolderPlugins` заданной длины передается в плагин шифрования, который должен получить от одного из них интерфейс `ICryptKeyCallback` и затем получить ключ, используя его. Параметр *keyName* — это имя ключа, которое было введено в операторе `ALTER DATABASE ENCRYPT ...`.

2.

```
void encrypt(StatusType* status,
             unsigned length,
             const void* from,
             void* to)
```

шифрует данные перед записью блока в файл базы данных

3.

```
void decrypt(StatusType* status,
             unsigned length,
```

```
            const void* from,
            void* to)
```

расшифровывает данные после чтения блока из файла базы данных.

4.

```
void setInfo(StatusType* status,
            IDbCryptInfo* info)
```

в этом методе плагин шифрования обычно сохраняет информационный интерфейс для будущего использования.

Хранитель ключа для плагина шифрования базы данных

Этот тип плагина необходим для разграничения функциональности — плагин шифрования базы данных имеет дело с фактическим шифрованием, держатель ключа решает вопросы, связанные с предоставлением ему ключа безопасным способом. Плагин может получить ключ из приложения или загрузить его каким-либо другим способом (вплоть до использования флеш-устройства, вставленного в сервер при запуске Firebird).

IKeyHolderPlugin

Назначение: Интерфейс `IKeyHolderPlugin` является основным интерфейсом для плагина хранения ключей шифрования.

1.

```
int keyCallback(StatusType* status,
                ICryptKeyCallback* callback)
```

используется для передачи интерфейса `ICryptKeyCallback` в подключение (если он предоставляется пользователем с вызовом `IProvider::setDbCryptCallback()`). Этот вызов всегда выполняется в момент подключения к базе данных, и некоторые держатели ключа могут отклонить подключение, если не был предоставлен удовлетворительный ключ.

2.

```
ICryptKeyCallback* keyHandle(StatusType* status,
                             const char* keyName)
```

предназначен для непосредственного вызова интерфейсом `IDbCryptPlugin` для получения интерфейса обратного вызова для именованного ключа из держателя ключа. Это позволяет использовать код Firebird с открытым исходным кодом так, чтобы никогда не касаться фактических ключей, избегая возможности кражи ключа, изменяющим код Firebird. После получения интерфейса `ICryptKeyCallback` плагин шифрования запускает обмен данными, используя его. Держатель ключа может (например) проверить цифровую подпись плагина шифрования перед отправкой ему ключа, чтобы избежать использования модифицированного плагина шифрования, способного украсть секретный ключ.

3.

```
FB_BOOLEAN useOnlyOwnKeys(StatusType* status)
```

информирует Firebird о том, будет ли использоваться ключ, предоставленный другим держателем ключа, или нет. Имеет смысл только для SuperServer — только он может делиться ключами шифрования базы данных между подключениями. Возвращая `FB_TRUE` из этого метода, принудительно заставляет Firebird убедиться, что этот конкретный держатель ключа (и, следовательно, связанное с ним подключение) предоставляет правильный ключ шифрования, прежде чем позволить ему работать с базой данных.

4.

```
ICryptKeyCallback* chainHandle(StatusType* status)
```

поддержка цепочки держателей ключей. В некоторых случаях ключ должен проходить через более чем один держатель ключа, прежде чем он достигнет плагина шифрования базы данных. Это необходимо (например) для поддержки `EXECUTE STATEMENT` в зашифрованной базе данных. Это всего лишь пример — цепочки также используются в некоторых других случаях. Интерфейс обратного вызова, возвращенный этим методом, может отличаться от возвращаемого функцией `keyHandle()` (см. выше). Как правило, он должен иметь возможность дублировать ключи один в один, полученные из `IKeyHolderPlugin` при вызове функции `keyCallback()`.

Неинтерфейсные объекты, используемые в API

> **Примечание**
>
> Они находятся в специальном заголовоке `Message.h` C++

Следующие 3 класса используются для представления типов DATE, TIME и TIMESTAMP (datetime) при использовании макроса FB_MESSAGE. Члены структуры данных, представляющие статическое сообщение, соответствуют полям типов FB_DATE/FB_TIME/FB_TIMESTAMP, будут иметь тип одного из этих классов. Для получения доступа к полям даты и времени в статических сообщениях необходимо знать методы и члены класса (которые достаточно самоописательны, чтобы не описывать их здесь).

FbDate

Методы класса `FbDate`:

1.

```
void decode(IUtil* util,
            unsigned* year,
            unsigned* month,
```

```
            unsigned* day)
```

2.

```
unsigned getYear(IUtil* util)
```

3.

```
unsigned getMonth(IUtil* util)
```

4.

```
unsigned getDay(IUtil* util)
```

5.

```
void encode(IUtil* util,
            unsigned year,
            unsigned month,
            unsigned day)
```

FbTime

Методы класса `FbTime`:

1.

```
void decode(IUtil* util,
            unsigned* hours,
            unsigned* minutes,
            unsigned* seconds,
            unsigned* fractions)
```

2.

```
unsigned getHours(IUtil* util)
```

3.

```
unsigned getMinutes(IUtil* util)
```

4.

```
unsigned getSeconds(IUtil* util)
```

5.

```
unsigned getFractions(IUtil* util)
```

6.

```
void encode(IUtil* util,
            unsigned hours,
            unsigned minutes,
            unsigned seconds,
            unsigned fractions)
```

FbTimestamp

Члены класса `FbTimestamp` :

1.

```
FbDate date;
```

2.

```
FbTime time;
```

FbChar u FbVarChar

Следующие два шаблона используются в статических сообщениях для представления полей CHAR(*N*) и VARCHAR(*N*).

```
template <unsigned N>
struct FbChar
{
    char str[N];
};
```

```
template <unsigned N>
struct FbVarChar
{
    ISC_USHORT length;
    char str[N];
    void set(const char* s);
};
```

Заключение

В этом документе отсутствуют два типа плагинов — ExternalEngine и Trace. Информация о них будет доступна в следующем выпуске.

Алфавитный указатель

F

I