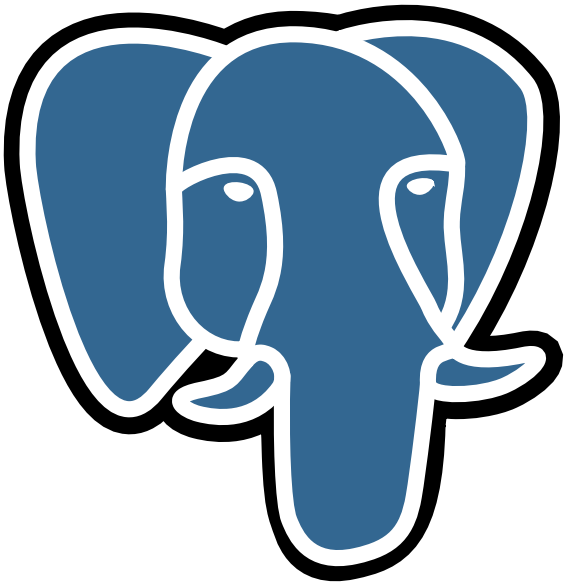

История о PostgreSQL

© Е.М. Балдин*

Эта статья была опубликована в февральском номере русскоязычного журнала Linux Format (`http://www.linuxformat.ru`) за 2007 год. Статья размещена с разрешения редакции журнала на сайте `http://www.inp.nsk.su/~baldin/` и до июля месяца все вопросы с размещением статьи в других местах следует решать с редакцией Linux Format. Затем все права на текст возвращаются ко мне.

Текст, представленный здесь, не является точной копией статьи в журнале. Текущий текст в отличии от журнального варианта корректор не просматривал. Все вопросы по содержанию, а так же замечания и предложения следует задавать мне по электронной почте `mailto:E.M.Baldin@inp.nsk.su`.

Текст на текущий момент является просто *текстом*, а не книгой. Поэтому результирующая доводка в целях улучшения восприятия текста не проводилась.

*e-mail: E.M.Baldin@inp.nsk.su

Слон взят с сайта `http://pgfoundry.org/projects/graphics/`. Изображение предоставляется под лицензией BSD.

Оглавление

4 Интерфейсы **1**
4.1 libpq 1
4.1.1 Открытие и закрытие соединения 2
4.1.2 SQL запросы 4
4.1.3 Большие объекты 7
4.2 ECPG 7
4.3 Всё остальное 9
4.4 Послесловие 11

В институте им очень дорожили, так как попутно он использовался для некоторых уникальных экспериментов и как переводчик при общении со Змеем Горынычем.

АБС о Кощее Бессмертном

Глава 4

Интерфейсы

Понятно, что в принципе любую базу данных можно заполнить в ручную, правда некоторые придётся заполнять очень долго. СУБД — это просто хранилище, а для заполнения и доступа к хранилищу необходима инфраструктура, и эту инфраструктуру надо создавать. Вот такая она — жизнь.

Родной библиотекой для доступа к PostgreSQL является библиотека libpq. Написана эта библиотека на чистом C, что и не удивительно, так как это основной язык *родной* системы. Все остальные языки важны, но безусловно вторичны. В любом случае мы выбираем их вовсе не поэтому.

4.1 libpq

Чтобы общаться с базой данных много функций не надо: одна функция для открытия соединения, одна для посылки запроса, одна для получения ответа и одна для закрытия соединения. В реальности всё немного сложнее, но суть остаётся.

К вопросу о переносимости Библиотека libpq написана на чистом C, поэтому практически везде, где есть `gcc`, можно организовать связь с PostgreSQL. Мне както пришлось это делать для VAX/VMS — всё решилось методом тыка, даже думать почти не потребовалось. Все данные — текст, поэтому вопрос бинарной совместимости платформ попросту отсутствует.

С чего начать Для того чтобы воспользоваться вызовами libpq, необходимо для начала установить её. В Debian (Etch) для этого надо установить пакет `postgresql-dev`:

```
> sudo apt-get install postgresql-dev
```

Для доступа к функциям libpq необходимо включить в исходник include-файл:

```
#include "libpq-fe.h"
```

Скрипт pg_config (man pg_config) позволяет получить информацию куда помещаются include-файлы, библиотеки и тому подобное:

```
> #сборка программы
> gcc  -o "бинарник" "исходник".c -I`pg_config --includedir` \
 -lpq  `pg_config --libs`
```

4.1.1 Открытие и закрытие соединения

Даже открывать соединение с PostgreSQL можно двумя способами:

```
//открыть соединение
PGconn *PQconnectdb(const char *conninfo);
//то же, но не блокируя программу
PGconn *PQconnectStart(const char *conninfo);
//проверка статуса соединения (после PQconnectStart)
PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

PQconnectdb — обычная функция, где на вход подаём текстовую строку conninfo с параметрами для соединения с сервером, а на выходе получаем структуру типа PGconn с информацией о сделанном соединении и на сколько операция по соединению прошла удачно. В дальнейшем при передача данных эта переменная будет использоваться в качестве параметра.

Передача информации о серевере в качестве строки (conninfo) позволяет в случае появления дополнительных параметров не менять внешний интерфейс вызова и легко добавлять дополнительные опции. Пример открытия соединения:

```
const char *conninfo= "dbname = test host=localhost";
PGconn *conn=PQconnectdb(conninfo);
if (PQstatus(conn) != CONNECTION_OK) {
   fprintf(stderr, "Не удалось соединиться с базой данных: %s",
                   PQerrorMessage(conn));
   /*завершаем работу*/ ...}
```

Параметры передаются в форме «ключевое слово» = «значение». Пары разделяются обычным пробелом. Пробелы вокруг знака равенства можно опустить. Если необходимо передать значение с пробелами, то его необходимо заключить в одинарные кавычки '«составное» «значение»'. Для передачи одинарной кавычки её необходимо экранировать с помощью обратной косой черты \'. При отсутствии какого-либо параметра в строке conninfo его значение берётся из соответствующей переменной окружения, если такая определена. Если нет, то при открытии соединения используется значение по умолчанию.

Функции открытия соединения распознают следующие параметры и переменные окружения (кое-какие особенности опущены):

host TCP/IP имя узла на котором находится сервер PostgreSQL. Соответствует переменной окружения `PGHOST`. Значение по умолчанию `localhost`.

hostaddr Числовой адрес узла на котором находится PostgreSQL (альтернатива host). Соответствует переменной окружения `PGHOSTADDR`. Значение по умолчанию эквивалентно `localhost`.

port Номер порта, который «слушает» POSTMASTER. Соответствует переменной окружения `PGPORT`. Значение по умолчанию обычно 5432.

dbname Имя базы данных. Соответствует переменной окружения `PGDATABASE`. Значение по умолчанию совпадает с системной учётной записью пользователя.

user Имя пользователя базы данных. Соответствует переменной окружения `PGUSER`. Значение по умолчанию совпадает с системной учётной записью пользователя.

password Поле пароля, если для аутентификации требуется пароль. Соответствует переменной окружения `PGPASSWORD`. Если аутентификация требуется, а поле неопределенно, то для доступа используется информация в файле ~/.pgpass. Переменная окружения `PGPASSFILE` может указать другой файл для проведения аутентификации.

connect_timeout Устанавливает максимальное время ожидания соединения в секундах. С сервером и сетью всякое может случиться. Соответствует переменной окружения `PGCONNECT_TIMEOUT`. Значение по умолчанию равно 0, что означает что время ожидания равно бесконечности. Не рекомендуется устанавливать значение на ожидание меньше 2 секунд.

options Опции, посылаемые непосредственно серверу, коли такое потребуется. Соответствует переменной окружения `PGOPTIONS`.

sslmode Определяет порядок действий при SSL-соединении. Принимает четыре возможных значения:

disable — без шифрации,

allow — сначала попробовать соединиться без шифрации, а в случае неудачи постараться установить защищённое соединение,

prefer — сначала попробовать установить защищённое соединения, а в случае неудачи повторить соединение без шифрации,

require — выполнять только защищённое соединение.

Соответствует переменной окружения `PGSSLMODE`. Значение по умолчанию `prefer`.

krbsrvname Имя Kerberos-сервиса. используется для аутентификации с помощью Kerberos-5[1]. Это совершенно отдельная тема для беседы. Соответствует переменной окружения `PGKRBSRVNAME`.

[1] Kerberos — промышленный стандарт для аутентификации и взаимодействия в условиях незащищённого окружения. Алгоритмы Kerberos основаны на шифровании с использованием симметричного криптографического ключа и требует наличие доверенного агента.

PGDATESTYLE Переменная окружения, позволяющая установить представление времени и даты по умолчанию. Соответствует SQL-команде SET datestyle TO...

PGTZ Переменная окружения, позволяющая установить текущий часовой пояс. Соответствует SQL-команде SET timezone TO...

PGCLIENTENCODING Переменная окружения, позволяющая установить «кодировку» клиента. Соответствует SQL-команде SET client_encoding TO...

Существует целый класс функций, которые позволяют получить информацию о соединении. Для подробностей следует обратиться к документации. Для начала полезно знать о двух из них:

```
ConnStatusType PQstatus(const PGconn *conn);
char *PQerrorMessage(const PGconn *conn);
```

PQstatus возвращает информацию о том, как прошло соединение. Интересны состояния `CONNECTION_OK` — всё хорошо и `CONNECTION_BAD` — ничего не вышло. Функция PQerrorMessage позволяет получить текстовую строку с описанием последней возникшей проблемы.

Для того чтобы разорвать соединение используется функция:

```
void PQfinish(PGconn *conn);
```

Внимание: Всегда следует закрывать соединения, когда в них отпадает нужда. Число соединений которые поддерживает POSTMASTER ограничено — очень легко парализовать работу базы данных только открывая новые соединения.

4.1.2 SQL запросы

Что ж, до сервера уже «дозвонились», теперь пора с ним поговорить.

Посылка запросов Простейший способ выполнить SQL-запрос, это воспользоваться функцией PQexec:

```
PGresult *PQexec(PGconn *conn, const char *command);
```

В качестве параметров передаётся структура соединения `conn` и строка с SQL-командой `command`. Возвращается указатель на структуру типа PGresult, где сохраняется информация полученная от СУБД в ответ на запрос. При желании можно в одном запросе отсылать сразу несколько SQL-команд, разделённых точкой с запятой «;», но в этом случае информация сохранённая в структуре PGresult будет относиться только к последнему запросу.

По умолчанию каждый PQexec считается за отдельную транзакцию, если явно не начать транзакцию с помощью команды BEGIN, которая будет продолжаться либо до COMMIT, либо до ROLLBACK.

Есть более сложный вызов PQexecParams, который позволяет передавать вызов и параметры к этому вызову раздельно. Таким образом исчезает необходимость

самостоятельно формировать строку SQL-команды и заботиться об экранировании данных, что важно в случае сохранения бинарных последовательностей. В качестве платы из соображения безопасности PQexecParams может послать не более одной команды за раз.

В некоторых случаях для увеличения скорости выполнения часто встречающихся запросов полезно обратить внимание на парочку PQprepare и PQexecPrepared. Эти команды эквивалентны своим SQL-аналогам PREPARE и EXECUTE. Идея оптимизации состоит в том, что прежде чем выполнить запрос, PostgreSQL сначала анализирует его, затем планирует порядок действий и только потом, собственно, выполняет запрос. Первые два этапа для похожих запросов с разными условиями отбора можно выполнить заранее с помощью команды PREPARE. Затем, с помощью команды EXECUTE, можно выполнять подобные уже подготовленные (prepared) запросы.

Все упомянутые выше команды работают с сервером БД синхронным образом, то есть посылают запрос и ждут ответа. Клиентское приложение на это время «засыпает» (suspended). В libpq предусмотрен целый класс функций предназначенный для асинхронных операций, не блокирующих клиентское приложение. Их применение усложняет код и логику программы, хотя всё в пределах допустимого. С моей точки зрения лучше организовать всё так, чтобы время использованное на ожидание результатов запроса фатально не влияло на внешние процессы и обеспечить бесперебойную работу сети и сервера базы данных.

Информация о состоянии запроса После выполнения запроса всегда интересно узнать каково его состояние:

```
ExecStatusType PQresultStatus(const PGresult *res);
```

На вход подаётся структура PGresult, создаваемая в результате работы PQexec-подобных функций, а на выходе получаем информацию о состоянии в виде числа, значение которого можно сравнить со следующими константами:

PGRES_COMMAND_OK — всё прошло хорошо (для запросов, которые не возвращают данные, например, INSERT),

PGRES_TUPLES_OK — всё прошло хорошо, плюс получены данные в ответ на запрос (для запросов типа SELECT или SHOW),

PGRES_EMPTY_QUERY — строка запроса была почему-то пустой,

PGRES_COPY_OUT — идёт передача данных *от* сервера,

PGRES_COPY_IN — идёт передача данных *на* сервер,

PGRES_BAD_RESPONSE — ошибка, ответ сервера не разборчив,

PGRES_NONFATAL_ERROR — ошибка, не смертельно: предупреждение (notice) или информация к сведению (warning),

PGRES_FATAL_ERROR — при выполнении запроса произошла серьёзная ошибка.

Для получения более подробной информации об ошибке следует воспользоваться функцией

```
char *PQresultErrorMessage(const PGresult *res);
```

При вызове этой функции в качестве результата будет сформирована строка с информацией об ошибке или пустая строка если всё прошло хорошо.

Получение данных При получении данных предполагается, что статус запроса соответствует **PGRES_TUPLES_OK**. Теперь, если примерно известно что хочется получить в результате запроса, то для получения данных достаточно четырёх функций:

```
int PQntuples(const PGresult *res);
int PQnfields(const PGresult *res);
char *PQgetvalue(const PGresult *res,
                 int row_number, int column_number);
int PQgetisnull(const PGresult *res,
                 int row_number, int column_number);
```

Первые две функции являются информационными и позволяют узнать сколько в результате запроса получено строк (PQntuples) и сколько колонок в каждой такой строке (PQnfields). Возьмите на заметку, что 0 строк — это тоже хороший результат.

Функция PQgetvalue позволяет получить доступ к полученным данным. В качестве параметров кроме структуры соединения (res) передаётся номер строки (column_number) и номер колонки (column_number). Все данные возвращаются так же в виде текстовой строки, как и посылаются, то есть, эти данные необходимо перевести в привычный формат. Например, в случае целых чисел можно воспользоваться функцией atoi.

Следует помнить, что данные SQL могут иметь неопределённое значение (NULL). Если подобная возможность существует, то перед получением значения проверить, а определено ли оно. PQgetisnull позволяет разобраться с этой проблемой. По передаваемым параметрам эта функция эквивалентна PQgetvalue, а в качестве результата возвращает 1, если значение *не* определено и 0, если опеределено.

Кроме упомянутых существует целый ряд функций, позволяющих получить информацию о полученных данных, как то: имя колонки (PQfname), размер передаваемых данных в байтах (PQgetlength) и тому подобное. Для экранирования специальных символов при операции с бинарными или текстовыми данными есть набор сервисных функций PQescape*.

COPY SQL команда COPY является расширением специфичным для PostgreSQL. Основное преимущество SQL — *«всё есть понятный текст»*, в некоторых случаях, когда надо передавать большие объёмы данных, оборачивается недостатком. Функции PQputCopyData и PQgetCopyData позволяют под час значительно ускорить передачу данных между сервером и клиентом.

Асинхронные сигналы Стандартный SQL не предполагает взаимодействия разных пользователей, кроме как через изменение данных в таблицах. PostgreSQL позволяет посылать асинхронные сигналы с помощью команд LISTEN и NOTIFY. LISTEN **"имя␣сигнала"** передаётся серверу как обычная SQL-команда. Если статус запроса становится равным **PGRES_COMMAND_OK**, то это означает, что ранее был выполнен запрос NOTIFY **"имя␣сигнала"**. Если же инициализация сигнала (NOTIFY) ожидается позже регистрации (LISTEN), то функция PQnotifies позволяет после любого запроса проверить наличие сигнала вновь.

Сборка «мусора» «Мусор» убирать придётся руками. Каждая функция типа PQexec создаёт объект типа PGresult. После того как вся необходимая информация о результатах запроса получена, следует освободить память, занимаемую этим объектом с помощью команды:

```
void PQclear(PGresult *res);
```

Если утечки памяти Вас не волнуют, то можно этого и не делать. В этом случае следует побеспокоиться о том: «Почему Вас не беспокоят утечки памяти?» ☺

4.1.3 Большие объекты

Ещё один способ сохранять неструктурированные данные в PostgreSQL — это «пихать» их как большие объекты (Large Objects). PostgreSQL предоставляет интерфейс схожим с файловым интерфейсом Unix: open (lo_open), read (lo_read), write (lo_write), lseek (lo_lseek) и так далее. Все lo_* команды работают со значениями полученными из колонки с типом `oid`. `oid` — это специальный тип данных, который является ссылкой на объект произвольного типа. То есть последовательность работы с большим объектом следующая: создаётся большой объект (lo_create). Далее возвращаемый lo_create указатель Oid используется для записи данных в большой объект (lo_import/lo_write), а затем этот указатель вставляется в таблицу с помощью стандартных SQL операторов. Чтение происходит в обратном порядке (lo_export/lo_read). Все операции с большими объектами должны происходить внутри транзакции.

P.S. Необходимость интерфейса больших объектов на текущий момент не так уж и очевидна. Стандартными средствами в PostgreSQL можно сохранять бинарные данные размером вплоть до 1 Гб, что вполне может соперничать с максимальным размером для большого объекта в 2 Гб.

4.2 ECPG

Чтобы не отставать от коммерческих баз данных PostgreSQL имеет свой собственный вариант «встроенного SQL». Эта технология позволяет смешивать обычный язык C с SQL-структурами, примерно следующим образом:

```
// файл test.pgc
#include <stdio.h>
#include <stdlib.h>
// структура для обработки ошибок
EXEC SQL include sqlca;
// реакция в случае ошибки/предупреждения
EXEC SQL whenever sqlwarning sqlprint;
EXEC SQL whenever sqlerror do ExitForError();
void  ExitForError() {
  fprintf(stderr,"Всё, конец - это фатально.\n");
  sqlprint();
  exit(1);
}

int main(int argc, char **argv)
{
// определение переменных, чтобы их можно было использовать
// инструкциях ECPG
  EXEC SQL BEGIN DECLARE SECTION;
   const char *dbname = "test";
   const char *user = "baldin";
   VARCHAR FIO[128];
   VARCHAR NUMBER[128];
  EXEC SQL END DECLARE SECTION;
// соединение с базой данных
// внешние переменные предваряются двоеточием
  EXEC SQL CONNECT TO :dbname USER :user;
// определение курсора через SELECT
  EXEC SQL DECLARE mycursor  CURSOR FOR
    SELECT fio, number FROM fiodata,phonedata
                       WHERE fiodata.id=phonedata.id;
  EXEC SQL open mycursor;
// чтение данных из курсора
  EXEC SQL FETCH NEXT FROM mycursor INTO :FIO,:NUMBER;
  while (sqlca.sqlcode == 0) { // не 0, если данные больше нет
     printf("ФИО: %s номер: %s\n",FIO.arr,NUMBER.arr);
    EXEC SQL FETCH NEXT FROM mycursor INTO :FIO, :NUMBER;
  }
// разъединение с базой данных
 EXEC SQL DISCONNECT;
}
```

Все SQL-команды начинаются с метки EXEC SQL. Эта метка позволяет затем препроцессору `ecpg` обработать и произвести C-исходник. Внутри SQL-команд можно использовать C-переменные. Для этого переменным в начале добавляется двоеточие «:».

Для компиляции выше процитированного исходника (файл `test1.pgc`) необходимо выполнить следующие действия:

```
> # установить ecpg
> sudo apt-get install libecpg-dev
> # запустить препроцессор
> ecpg test1.pgc
> # скомпилировать получившийся исходник
> gcc  -o test1 test1.c -I`pg_config --includedir` -lecpg
> # проверка работоспособности программы
> ./test1
ФИО: Иванов И.П. номер: 555-32-23
ФИО: Балдин Е.М. номер: 555-41-37
ФИО: Балдин Е.М. номер: (+7)5559323919
```

Удобно это или нет — решайте сами.

4.3 Всё остальное

Статья называется «Интерфейсы», а большая часть *посвящена* только одному из них. Дело в том, что этот «один» является родным и наиболее полным, а всё остальное лишь подмножество. В простейшем случае все интерфейсы одинаковы: открыл соединение, послал запрос, обработал результаты запроса, закрыл соединение. Так же заметна энергосберегающая тенденция везде делать ровно один интерфейс на все типы СУБД.

bash Да, да к bash тоже есть свой интерфейс, правда для этого надо патчить его исходники. Возни, конечно, не мало — зато прямо в shell-скриптах можно обращать к базе данных ☺.

Страничка проекта: `http://www.psn.co.jp/PostgreSQL/pgbash/index-e.html`.

Java Совершенно ожидаемо, что Java общается с PostgreSQL стандартным образом, а именно через JDBC. Поэтому если знаком с Java, то достаточно добыть драйвер JDBC для PostgreSQL, например отсюда: `http://jdbc.postgresql.org/` или в Debian (Sarge) набрать

```
> sudo apt-get install libpgjava
```

и, прочитав README к пакету, приступить к работе.

lisp Точнее Common Lisp. Скорее всего эти драйвера подойдут и для других диалектов:

```
> sudo apt-get install cl-pg
#или
> sudo apt-get install cl-sql-postgresql
```

Второй вариант является драйвером для единого интерфейса доступа к SQL-базам данных из Common Lisp CLSQL (`http://clsql.b9.com/`).

perl Интерфейс для связи с PostgreSQL DBD-Pg используется в perl через DBI[2]. Все подробности на CTAN: `http://search.cpan.org/~dbdpg/DBD-Pg/Pg.pm`.

```
> libdbd-pg-perl
```

DBD-Pg охватывает фактически все имеющиеся на сегодня возможности PostgreSQL от больших объектов (large objects), до точек сохранения (savepoints).

PHP О том как использовать PostgreSQL в PHP-проектах можно прочитать здесь: `http://www.php.net/manual/en/ref.pgsql.php`. Установить драйвер можно, например, так:

```
> sudo apt-get install php5-pgsql
```

Говорят, почти единственной причиной, по которой PHP-разработчики предпочитают MySQL является то, что раньше не было «родной» версии PostgreSQL под альтернативную операционную систему. С версии 8.0 PostgreSQL конкретно *этот* довод «против» уже не работает.

Python Модуль для Python существует уже больше десяти лет. Подробности выясняются здесь: `http://www.druid.net/pygresql/`. Установка модуля:

```
> sudo apt-get install python-pygresql
```

Более «молодая» и по утверждениям пользователей более стабильная библиотека для связи с PostgreSQL psycopg2 (`http://initd.org/projects/psycopg2`) так же устанавливается из коробки:

```
>  sudo apt-get install python-psycopg2
```

Ruby Что-то есть здесь: `http://ruby.scripting.ca/postgres/`. Установка, как обычно:

```
> sudo apt-get install libdbd-pg-ruby
```

[2]DBI — унифицированный интерфейс для доступа к данным. Подробнее об этом пакете можно посмотреть на CTAN: `http://search.cpan.org/~timb/DBI-1.52/DBI.pm`

ODBC Разработка драйвера идёт на pgFoundry, аскетичная страничка проекта здесь: `http://pgfoundry.org/projects/psqlodbc/`. Установка:

```
> sudo apt-get install odbc-postgresql
```

4.4 Послесловие

Очевидно, что есть много чего ещё. При желании можно самому написать, благо родной C-интерфейс уже знаком, а логика достаточно прозрачна: открыл соединение, обменялся SQL-запросами и обязательно закрыл соединение. С другой стороны не стоит изобретать велосипеда и лучше для начала посмотреть, что было уже сделано, например, здесь: `http://techdocs.postgresql.org/oresources.php`.

Врезка про bond

Лень писать всё самому? Но не лень изучать XML? Тогда BOND — это программа для Вас. Сайт проекта `http://www.treshna.com/bond/`.

Рабочей частью пакета является исполняемый файл `bondfrontend`, который осуществляет связь с базой данных и может «прикинуться» любой формой. Описание формы хранится в обычном xml-файле. Используемый диалект xml подробно описан в документации.

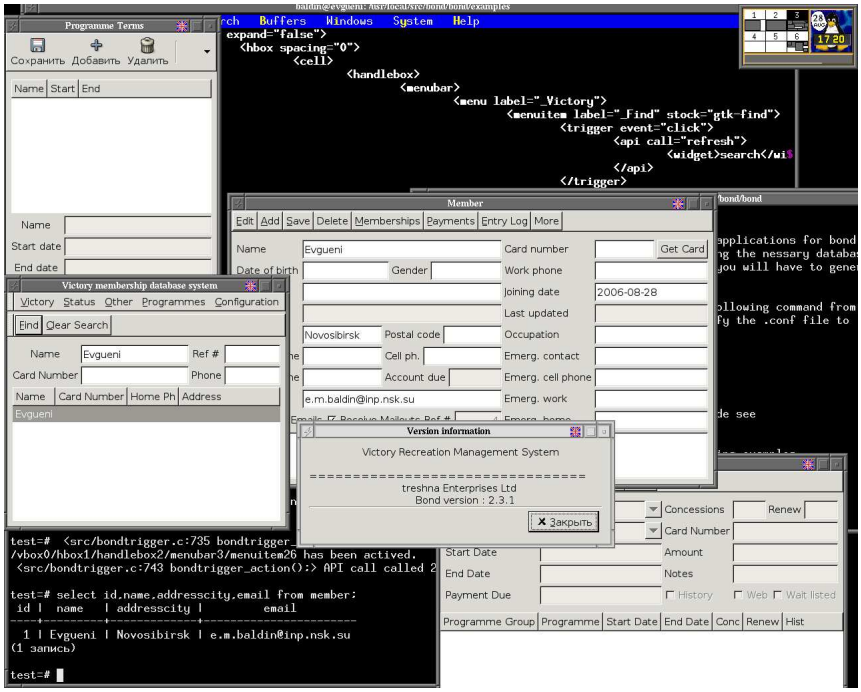

Рис. 4.1. Формочки, XML (правда на заднем фоне и без подсветки) и связь с базой данных — это bond

История пакета насчитывает уже пять лет. Программа доступна под GPL, то есть исходники производных продуктов надо открывать, а если хочется пожадничать, то есть версия и для такого случая, но за деньги. Доступна версия и под win32.

Внимательно читаем README и устанавливаем всё что там перечисляется. Сборка осуществляется с помощью `scons`, который позиционируется как замена `make` с функциональность `automake`/`autoconf` и синтаксисом от Python. Установка по умолчанию производится `/usr/local/`. Для установки (scons install) необходимы привилегии системного администратора. Далее читаем документацию и изучаем директорию `examples`.