Gem #130: Базы данных API, безопасные по отношению к типам - Часть 2.

Автор: Emmanuel Briot — AdaCore

Краткое содержание: В первом Gem этой серии обсуждалось написание правильных с синтаксической точки зрения и безопасных по отношению к типам запросов SQL. Теперь нам нужно запустить эти запросы на выбранной нами СУБД и получить результаты. В данном Gem поясняется, как использовать с этой целью независимый от типа СУБД API в `GNATColl`.

Давайте начнём…

Первый Gem в этой серии обсуждал, как писать синтаксически правильные и безопасные для типов SQL-запросы. Теперь нам нужно выполнить эти запросы в выбранной СУБД и получить результаты. Этот Gem объясняет, как использовать независимый от СУБД API в `GNATColl` для этого.

В настоящее время `GNATColl` обеспечивает поддержку двух СУБД: `PostgreSQL` и `SQLite`. Он может быть расширен для других СУБД путем переопределения определенных примитивных операций (дополнительную информацию см. В документации по `gnatcoll-sql-exec.ads`).

Когда вы компилируете `GNATColl`, он автоматически определяет, установлена ли какая-либо из поддерживаемых СУБД в вашей системе, и при необходимости компилирует их поддержку. Но, конечно, когда вы связываете свое приложение с `GNATColl`, вы не хотите систематически зависеть от каждой СУБД, распознаваемой `GNATColl` (например, `PostgreSQL` и `SQLite`), только той, которая вам действительно нужна. Используя файлы проекта, вы можете сообщить компоновщику, какая СУБД вам нужна. Вот пример файла проекта, который добавляет поддержку `SQLite`:

```
with "gnatcoll_sqlite";
project Default is
   for Main use ("main.adb");
end Default;
```

В коде Ada нам теперь нужно подключиться к реальной СУБД. Это единственное место в коде, где будет явно указано, какую СУБД мы используем. Поэтому перенос вашего приложения из `SQLite` в `PostgreSQL` - это всего лишь вопрос изменения этого кода. На остальную часть приложения это не влияет.

```
with GNATCOLL.SQL.Sqlite;    -- or PostgreSQL
declare
   DB_Descr : GNATCOLL.SQL.Exec.Database_Description :=
      new GNATCOLL.SQL.Sqlite.Setup ("dbname.db");
   DB : GNATCOLL.SQL.Exec.Database_Connection;
begin
   ...
end;
```

Параметры для настройки зависят от СУБД. Например, `PostgreSQL` позволяет вам указать хост, имя пользователя и пароль.

`DB_Descr` - это просто описание будущих соединений. Чтобы получить реальное соединение, есть ряд возможностей:

1. Если ваше приложение не является многозадачным, вы можете использовать следующее:

```
DB := DB_Descr.Build_Connection;
```

2. Однако, при использовании многозадачности могло бы быть лучше всегда получить то же соединение из данной задачи. Например, веб-сервер обычно использует одну задачу на HTTP-запрос, и мы можем получить определенное для задачи соединение с:

```
DB := GNATCOLL.SQL.Exec.Get_Task_Connection (DB_Descr);
```

3. Наконец, можно иметь пул таких соединений, что будет объяснено в третьей части этой серии Gem.

На данный момент мы еще не подключились к СУБД, но это делается автоматически при первом выполнении запроса. Если по какой-либо причине соединение было разорвано (например, из-за ошибки сети), `GNATColl` автоматически пытается повторно подключиться несколько раз, прежде чем сдаться.

Теперь выполним наш первый запрос:

```
declare
   Q : constant SQL_Query := ... ;   -- See first Gem in the series
   R : Forward_Cursor;
begin
   R.Fetch (Connection => DB, Query => Q);
end;
```

`R` будет содержать фактический результат. Для `R` существует два возможных типа: `Forward_Cursor` может повторяться только по одной строке за раз; при переходе к следующей строке предыдущая строка теряется навсегда. В обмен на это ограничение, `GNATCOLL.SQL` не должен извлекать все результаты в памяти сразу, что может быть более эффективным, если вы намерены остановить итерацию на ранней стадии. Второй тип - это `Direct_Cursor`, который извлекает все строки сразу, сохраняет их в памяти и позволяет перемещаться по ним вперед или назад или даже напрямую переходить к определенной строке. Это более гибко, но требует больше памяти в приложении.

Печать результатов можно выполнить с помощью знакомой идиомы цикла на основе курсора:

```
   while Has_Row (R) loop
      Put_Line (Integer'Image (Integer_Value (R, 0))  -- First field
                & ' ' & Value (R, 1));  -- Second field, as a string
      Next (R);
   end loop;
```

Мы должны знать индекс каждого поля в запросе, и должны ли мы получить его как целое число, последовательность, булевскую переменную, и т.д. Третий Gem в этой серии, покажет решение, которое более безопасно и приводит к ошибке времени компиляции, когда мы используем неправильные типы.

Вот две дополнительных возможности, обеспеченные `GNATCOLL.SQL.Exec`, которые полностью описаны в документации:

**** Автоматические транзакции***

Когда команда SQL изменяет базу данных (например, с помощью `INSERT` или `UPDATE`), `GNATColl` автоматически запускает транзакцию в СУБД, так что все такие изменения группируются и либо выполняются, либо отбрасываются. Для этого он предоставляет две примитивные операции над соединением: `Commit` и `Rollback`.

**** Подготовленные и параметризованные запросы***

Как мы уже упоминали в первом Gem, создание запроса в первую очередь менее эффективно (но безопаснее), чем использование строки. Решение подготовить запрос. Подготовка может происходить либо на стороне клиента (так `GNATColl` преобразует его в строку только один раз, а затем использует ее каждый раз, когда вы посылаете один и тот же запрос), или непосредственно на сервере СУБД (которая будет потом разобрать строку и подготовить ее выполнение, так что несколько исполнений запросов на стороне сервера гораздо быстрее - оптимизируются затраты на передачу запросов по каналам связи). Конечно, не часто требуется выполнять один и тот же запрос, поэтому последний полезен только тогда, когда часть запроса может быть заменена с помощью параметров. Вот пример такого запроса:

```
declare
   Q : constant SQL_Query := SQL_Select
      (Fields => Table1.Field2 & Table2.Field3,
       From => Table1 & Table2,
       Where => Table1.Field1 = Table2.Field3
          and Table1.Field1 = Integer_Param (1));
   Prepared_Q : constant Prepared_Statement :=
      Prepare (Q, On_Server => True);
begin
   ...
end;
```

Здесь многое происходит: `Q` сам по себе указывает на то, что ему понадобится один параметр (индекс `1`), который является целым числом; `Prepare_Q` тогда такой же, как `Q`, но будет подготовлен на сервере. В момент объявления соединения нет, поэтому очевидно, что запрос еще не подготовлен на сервере. Более того, `GNATColl` должен подготовить его для каждого подключения к базе данных, а не только один раз. Таким образом, эта подготовка будет происходить при первом выполнении запроса для определенного соединения.

При выполнении `Prepared_Q` необходимо указать значение параметра, например:

```
R.Fetch (DB, Prepared_Q, Params => (1 => +23)); -- first execution
R.Fetch (DB, Prepared_Q, Params => (1 => +34)); -- second execution
```

Второе исполнение будет намного быстрее первого. Для сравнения: при прохождении `SQL_Query` непосредственно и с помощью `Direct_Cursor`, он затратил 4.05 s, чтобы выполнить тот же запрос 100_000 раз с `SQLite`; один и тот же запрос подготовлен на клиент затратил 2.50 с; наконец, тот же запрос, подготовленный на сервере, как описано выше, затратил только 0.55 сек. Подготовка запросов, таким образом, может иметь очень значительное влияние на производительность вашего приложения.

Продолжение следует в части 3 ...

Связанный со статьёй текст программы

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Автор: Emmanuel Briot

AdaCore

Эммануэль Брио работает в AdaCore с 1998 года. Он принимал участие в различных проектах, в частности, ориентированных на графические пользовательские интерфейсы, включая GtkAda, GPS, XML / Ada, GnatTracker и нашу внутреннюю CRM. Он получил степень инженера в Национальной школе телекоммуникаций - Брест, Франция (Ecole Nationale des Telecommunications - Brest, France).

Last Updated: 10/13/2017
Posted on: 7/23/2012

Обсуждение…