PostgreSQL 9.4

Глава 36. Триггеры

В этой главе содержится общая информация о разработке триггерных функций. Триггерные функции могут быть написаны на большинстве доступных процедурных языков, включая PL/pgSQL (Гл. 40), PL/Tcl (Гл. 41), PL/Perl (Гл. 42), and PL/Python (Гл. 43). После прочтения этого раздела, следует обратиться к разделу, посвящённому любимому процедурному языку, чтобы узнать специфические для него детали разработки триггеров.

Триггерные функции можно писать и на C, хотя большинство людей находит, что проще использовать один из процедурных языков. В настоящее время невозможно написать триггерную функцию на чистом SQL.

36.1. Обзор механизма работы триггеров

Триггер является указанием, что база данных должна автоматически выполнить заданную функцию, всякий раз когда выполнен определённый тип операции. Триггеры можно использовать с таблицами, с представлениями и с внешними таблицами.

Для таблиц можно определять триггеры, которые будут срабатывать до или после любой из команд `INSERT`, `UPDATE` или `DELETE`; либо один раз на каждую модифицируемую строку, либо один раз на SQL оператор. Кроме того, для триггеров на `UPDATE` можно задать, чтобы они срабатывали только в том случае, когда определённые столбцы указаны во фразе `SET` оператора `UPDATE`. Триггеры также могут срабатывать для операторов `TRUNCATE`. Если происходит событие триггера, триггерная функция вызывается в соответствующее время (до, после) для обработки события. Внешние таблицы не поддерживают оператор `TRUNCATE`.

Для представлений триггеры могут быть определены вместо команд `INSERT`, `UPDATE` или `DELETE`. Триггеры `INSTEAD OF` запускаются один раз для каждой строки, которую необходимо изменить в представлении. Именно триггерная функция отвечает за выполнение необходимых изменений в базовых таблицах и, где это уместно, возвращает измененную строку в том виде, как она будет отображаться в представлении. Триггеры на представления также можно определять для срабатывания только один раз на SQL оператор, до или после команд `INSERT`, `UPDATE` или `DELETE`.

Триггерная функция должна быть создана до триггера. Она должна быть объявлена без аргументов и возвращать тип `trigger`. (Триггерная функция получает данные на вход посредством специально переданной структуры `TriggerData`, а не в форме обычных аргументов.)

После создания триггерной функции создается триггер с помощью CREATE TRIGGER. Одна

и та же триггерная функция может быть использована для нескольких триггеров.

PostgreSQL предлагает как строчные триггеры (*per-row*), так и операторные триггеры (*per-statement*). В случае строчного триггера, триггерная функция вызывается один раз для каждой строки, затронутой оператором, запустившим триггер. В противоположность этому, операторный триггер вызывается только один раз при выполнении соответствующего оператора, независимо от количества строк, которые затрагивает. В частности оператор, который вообще не затрагивает строк, все равно приведет к срабатыванию операторного триггера. Эти два типа триггеров иногда называют триггеры уровня строк (*row-level*) и триггеры уровня оператора (*statement-level*) соответственно. Триггеры на `TRUNCATE` могут быть определены только на уровне оператора. Триггеры на представления, срабатывающие до или после, могут быть определены только уровне оператора, в то время как триггеры, срабатывающие вместо команд `INSERT`, `UPDATE` или `DELETE`, могут быть определены только на уровне строк.

Триггеры также классифицируются в соответствии с тем, срабатывают ли они *до*, *после* или *вместо* операции. Они называются `BEFORE` триггеры, `AFTER` триггеры и `INSTEAD OF` триггеры соответственно. Триггеры `BEFORE` уровня оператора срабатывают до того как оператор начинает делать что-либо, в то время как триггеры `AFTER` уровня оператора срабатывают в самом конце работы оператора. Эти типы триггеров могут быть определены для таблиц или представлений. Триггеры `BEFORE` уровня строки срабатывают непосредственно перед обработкой конкретной строки, в то время как триггеры `AFTER` уровня строки срабатывают в конце работы всего оператора (но до любого из триггеров `AFTER` уровня оператора). Эти типы триггеров могут определяться только для таблиц и внешних таблиц. Триггеры `INSTEAD OF` уровня строки могут определяться только для представлений и срабатывают для каждой строки, сразу после того как строка представления идентифицирована как нуждающаяся в обработке.

Триггерные функции, вызываемые триггерами оператора должны всегда возвращать `NULL`. Триггерные функции, вызываемые триггерами строк могут вернуть строку таблицы (значение типа `HeapTuple`). У триггера уровня строки, срабатывающего до операции, есть следующий выбор:

- Можно вернуть `NULL`, чтобы пропустить операцию для текущей строки. Это указывает исполнителю запросов, что не нужно выполнять операцию со строкой вызвавшей триггер (вставку, изменение или удаление конкретной строки в таблице).
- Возвращаемая строка для триггеров `INSERT` или `UPDATE` будет именно той, которая будет вставлена или обновлена в таблице. Это позволяет триггерной функции изменять вставляемую или обновляемую строку.

Если в `BEFORE` триггере уровня строки не планируется использовать любой из этих вариантов, то нужно аккуратно вернуть в качестве результата ту же строку, которая была передана на вход (то есть строку `NEW` для триггеров `INSERT` и `UPDATE`, или строку `OLD` для

триггеров DELETE).

INSTEAD OF триггер уровня строки должен вернуть либо NULL, чтобы указать, что он не модифицирует базовые таблицы представления, либо он должен вернуть строку представления, полученную на входе (строку NEW для операций INSERT и UPDATE или строку OLD для операций DELETE). Отличное от NULL возвращаемое значение сигнализирует, что триггер выполнил необходимые изменения данных в представлении. Это приведёт к увеличению счётчика количества строк, затронутых командой. Для операций INSERT и UPDATE триггер может изменить строку NEW перед тем как её вернуть. Это изменит данные, возвращаемые INSERT RETURNING или UPDATE RETURNING, и полезно для того, чтобы не показывать уже не актуальные первоначальные данные.

Возвращаемое значение игнорируется для триггеров уровня строки, вызываемых после операции, поэтому они могут возвращать NULL.

Если есть несколько триггеров на одно и то же событие для одной и той же таблицы, то они будут вызываться в алфавитном порядке по имени триггера. Для триггеров BEFORE и INSTEAD OF потенциально изменённая строка, возвращаемая одним триггером, становится входящей строкой для следующего триггера. Если любой из триггеров BEFORE или INSTEAD OF возвращает NULL, операция для этой строки прекращается и последующие триггеры (для этой строки) не срабатывают.

В определении триггера можно указать логическое условие WHEN, которое будет проверяться, чтобы посмотреть, нужно ли запускать триггер. В триггерах уровня строки в условии WHEN можно проверять старые и/или новые значения столбцов строки. (В триггерах уровня оператора также можно использовать условие WHEN, хотя в этом случае это не так полезно.) В триггерах BEFORE условие WHEN вычисляется непосредственно перед тем, как триггерная функция будет выполнена, поэтому использование WHEN существенно не отличается от выполнения той же проверки в самом начале триггерной функции. Однако, в триггерах AFTER условие WHEN вычисляется сразу после обновления строки и от этого зависит будет ли поставлено в очередь событие запуска триггера в конце оператора или нет. Поэтому, когда условие WHEN в триггере AFTER не возвращает истину, не требуется ни постановка события в очередь, ни повторная выборка этой строки в конце оператора. Это может существенно ускорить работу операторов, изменяющих большое количество строк, с триггером, который должен сработать только для нескольких. В триггерах INSTEAD OF не поддерживается использование условий WHEN.

Как правило, триггеры BEFORE уровня строки используются для проверки или модификации данных, которые будут вставлены или изменены. Например, триггер BEFORE можно использовать для вставки текущего времени в столбец timestamp или проверки, что два элемента строки согласованы между собой. Триггеры AFTER уровня строки наиболее разумно использовать для каскадного обновления данных в других таблицах или проверки согласованности сделанных изменений с данными в других таблицах. Причина для такого

разделения работы в том, что триггер AFTER видит окончательное значение строки, в то время как для триггера BEFORE это не так, ведь могут быть другие триггеры BEFORE, которые сработают позже. Если нет особых причин для выбора между триггерами BEFORE или AFTER, то триггер BEFORE предпочтительнее, так как не требует сохранения информации об операции до конца работы оператора.

Если триггерная функция выполняет команды SQL, эти команды могут заново запускать триггеры. Это известно как каскадные триггеры. Прямых ограничений на количество каскадных уровней не существует. Вполне возможно, что каскадные вызовы приведут к рекурсивному срабатыванию одного и того же триггера. Например, в триггере INSERT может выполняться команда, которая добавляет строку в эту же таблицу, тем самым опять вызывая триггер на INSERT. Обязанность программиста не допускать бесконечную рекурсию в таких случаях.

При определении триггера можно указывать аргументы. Цель включения аргументов в определение триггера в том, чтобы позволить разным триггерам с аналогичными требованиями вызывать одну и ту же функцию. В качестве примера можно создать обобщенную триггерную функцию, которая принимает два аргумента с именами столбцов и записывает текущего пользователя в первый аргумент и текущий штамп времени во второй. При правильном написании такая триггерная функция будет независима от конкретной таблицы, для которой она будет запускаться. Таким образом, одна и та же функция может использоваться при выполнении INSERT в любую таблицу с соответствующими столбцами, чтобы, например, автоматически отслеживать создание записей в транзакционной таблице. Для триггеров UPDATE аргументы также могут использоваться для отслеживания последних сделанных изменений.

У каждого языка программирования, поддерживающего триггеры, есть свой собственный метод доступа из триггерной функции к входным данным триггера. Входные данные триггера включают в себя тип события (например, INSERT или UPDATE), а также любые аргументы, перечисленные в CREATE TRIGGER. Для триггеров уровня строки входные данные также включают строку NEW для триггеров INSERT и UPDATE, и/или строку OLD для триггеров UPDATE и DELETE. Триггеры уровня оператора в настоящее время не имеют возможностей для проверки отдельных строк, модифицированных оператором.

36.2. Видимость изменений в данных

Если в триггерной функции выполняются SQL команды и эти команды обращаются к таблице, на которую создан триггер, то необходимо знать правила видимости данных, потому что они определяют будут ли видеть эти SQL команды изменения в данных, для которых сработал триггер. Кратко:

- Триггеры уровня оператора следуют простым правилам видимости: никакие из изменений, сделанных оператором, не видны в триггерах BEFORE, тогда как в

триггерах AFTER видны все изменения.

- Изменение данных (вставка, обновление или удаление), заставляющее сработать триггер, *не видно* для команд SQL, выполняемых в триггере BEFORE уровня строки, потому что это изменение ещё не произошло.
- Тем не менее, команды SQL, выполняемые в триггере BEFORE уровня строки, *будут* видеть изменения данных в строках, которые уже были обработаны в этом операторе. Это требует осторожности, так как порядок обработки строк в целом непредсказуемый; команда SQL, обрабатывающая множество строк, может делать это в любом порядке.
- Аналогично, INSTEAD OF триггер уровня строки увидит изменения данных, сделанные предыдущими срабатываниями триггера INSTEAD OF в этом же операторе.
- Когда срабатывает триггер AFTER уровня строки, все изменения сделанные оператором уже выполнены и видны в вызываемой триггерной функции.

Если триггерная функция написана на одном из стандартных процедурных языков, вышеприведённые утверждения применимы, только если функция объявлена как VOLATILE. Функции объявленные как STABLE или IMMUTABLE в любом случае не будут видеть изменений, сделанных вызывающим оператором.

Дополнительную информацию о правилах видимости данных можно найти в Разд. 44.4. Пример в Разд. 36.4 содержит демонстрацию этих правил.

36.3. Триггерные функции на языке C

Этот раздел описывает низкоуровневые детали интерфейса для триггерной функции. Эта информация необходима только при разработке триггерных функций на C. При использовании языка более высокого уровня эти детали обрабатываются автоматически. В большинстве случаев необходимо рассмотреть использование процедурного языка, прежде чем начать разрабатывать триггеры на C. В документации по каждому процедурному языку объясняется как создавать триггеры на этом языке.

Триггерные функции должны использовать "version 1" интерфейса диспетчера функций.

Когда функция вызывается диспетчером триггеров, ей не передаются обычные аргументы, но передается указатель "context", ссылающийся на структуру TriggerData. Функции на C могут проверить вызваны ли они диспетчером триггеров или нет выполнив макрос:

```
CALLED_AS_TRIGGER(fcinfo)
```

который разворачивается в:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Если возвращается истина, то `fcinfo->context` можно безопасно привести к типу `TriggerData *` и использовать указатель на структуру `TriggerData`. Функция *не* должна изменять структуру `TriggerData` или любые данные, которые на неё указывают.

`struct TriggerData` определяется в `commands/trigger.h`:

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
    Buffer        tg_trigtuplebuf;
    Buffer        tg_newtuplebuf;
} TriggerData;
```

где элементы определяются следующим образом:

`type`

Всегда `T_TriggerData`.

`tg_event`

Описывает событие, для которого вызывается функция. Можно использовать следующие макросы для получения информации о `tg_event`:

`TRIGGER_FIRED_BEFORE(tg_event)`

Возвращает истину, если триггер сработал до операции.

`TRIGGER_FIRED_AFTER(tg_event)`

Возвращает истину, если триггер сработал после операции.

`TRIGGER_FIRED_INSTEAD(tg_event)`

Возвращает истину, если триггер сработал вместо операции.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Возвращает истину, если триггер сработал на уровне строки.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Возвращает истину, если триггер сработал на уровне оператора.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Возвращает истину, если триггер сработал для операции `INSERT`.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Возвращает истину, если триггер сработал для операции `UPDATE`.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Возвращает истину, если триггер сработал для операции `DELETE`.

`TRIGGER_FIRED_BY_TRUNCATE(tg_event)`

Возвращает истину, если триггер сработал для операции `TRUNCATE`.

`tg_relation`

Указатель на структуру, описывающую таблицу, для которой сработал триггер. Подробнее об этой структуре в `utils/rel.h`. Самое интересное здесь это `tg_relation->rd_att` (дескриптор записей таблицы) и `tg_relation->rd_rel->relname` (имя таблицы; имеет тип `NameData`, а не `char*`; используйте `SPI_getrelname(tg_relation)` чтобы получить тип `char*` если потребуется копия имени).

`tg_trigtuple`

Указатель на строку, для которой сработал триггер. Это строка, которая вставляется, обновляется или удаляется. При срабатывании триггера для `INSERT` или `DELETE` это значение нужно вернуть из функции, только если не планируется изменять строку (в случае `INSERT`) или пропускать операцию для этой строки.

`tg_newtuple`

Для триггера на `UPDATE` это указатель на новую версию строки либо `NULL`, если триггер на `INSERT` или `DELETE`. Это значение нужно вернуть из функции в случае `UPDATE`, если не планируется изменять строку или пропускать операцию для этой строки.

`tg_trigger`

Указатель на структуру с типом `Trigger`, определенную в `utils/reltrigger.h`:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
```

```
        bool        tginitdeferred;
        int16       tgnargs;
        int16       tgnattr;
        int16      *tgattr;
        char      **tgargs;
        char       *tgqual;
    } Trigger;
```

где `tgname` - имя триггера, `tgnargs` - количество аргументов в `tgargs`, и `tgargs` - массив указателей на аргументы, указанные в команде `CREATE TRIGGER`. Остальные члены структуры предназначены для внутреннего использования.

`tg_trigtuplebuf`

Буфер, содержащий `tg_trigtuple`, или содержащий `InvalidBuffer` - если нет такой строки или она не хранится в дисковом буфере.

`tg_newtuplebuf`

Буфер, содержащий `tg_newtuple`, или содержащий `InvalidBuffer` - если нет такой строки или она не хранится в дисковом буфере.

Триггерная функция должна возвращать указатель `HeapTuple` или указатель `NULL`(но *не* SQL значение `null`, то есть не нужно устанавливать `isNull` в истину). Не забудьте, что если не планируете менять обрабатываемую триггером строку, то нужно вернуть либо `tg_trigtuple`, либо `tg_newtuple`.

36.4. Полный пример триггера

Вот очень простой пример триггерной функции, написанной на C. (Примеры триггеров для процедурных языков могут быть найдены в документации на процедурные языки.)

Функция `trigf` сообщает количество строк в таблице `ttest` и пропускает операцию для строки при попытке вставить пустое значение в столбец `x`. (Таким образом, триггер действует как ограничение `NOT NULL`, но не прерывает транзакцию.)

Вначале определение таблицы:

```
CREATE TABLE ttest (
    x integer
);
```

Теперь исходный код триггерной функции:

```
#include "postgres.h"
#include "executor/spi.h"       /* это нужно для работы с SPI */
#include "commands/trigger.h"   /* ... с триггерами ... */
#include "utils/rel.h"          /* ... и с таблицами */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
```

```
#endif

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* Убедимся, что функция вызвана триггером */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* Строка, которую будем возвращать */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* Проверяем на пустые значения */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* Подключаемся к менеджеру SPI */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (сработал %s): SPI_connect вернула %d", when, ret);

    /* Получаем число строк в таблице */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (сработал %s): SPI_exec вернула %d", when, ret);

    /* count(*) возвращает int8, требуется конвертация */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (сработал %s): в таблице ttest %d строк", when, i);

    SPI_finish();

    if (checknull)
    {
```

```
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

После компиляции исходного кода (см. Разд. 35.9.6) объявляем функцию и триггеры:

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

Теперь можно проверить работу триггера:

```
=> INSERT INTO ttest VALUES (NULL);

INFO:  trigf (сработал before): в таблице ttest 0 строк
INSERT 0 0

-- Вставка записи пропущена (NULL значение), поэтому AFTER триггер не сработал

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (сработал before): в таблице ttest 0 строк
INFO:  trigf (сработал after ): в таблице ttest 1 строк
                                                ^^^^^^^
                                    вспомним, что говорили о видимости
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (сработал before): в таблице ttest 1 строк
INFO:  trigf (сработал after ): в таблице ttest 2 строк
                                                ^^^^^^^
                                    вспомним, что говорили о видимости
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (сработал before): в таблице ttest 2 строк
```

```
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (сработал before): в таблице ttest 2 строк
INFO:  trigf (сработал after ): в таблице ttest 2 строк
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (сработал before): в таблице ttest 2 строк
INFO:  trigf (сработал before): в таблице ttest 1 строк
INFO:  trigf (сработал after ): в таблице ttest 0 строк
INFO:  trigf (сработал after ): в таблице ttest 0 строк
                                                ^^^^^^^
                                    вспомним, что говорили о видимости
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

Более сложные примеры можно найти в `src/test/regress/regress.c` и в spi.