Triggers

4 min. read · View original

A trigger is a fragment of code that you tell Oracle to run before or after a table is modified. A trigger has the power to

- make sure that a column is filled in with default information
- make sure that an audit row is inserted into another table
- after finding that the new information is inconsistent with other stuff in the database, raise an error that will cause the entire transaction to be rolled back

Consider the `general_comments` table:

```
create table general_comments (
        comment_id              integer primary
key,
        on_what_id              integer not null,
        on_which_table          varchar(50),
        user_id                 not null
references users,
        comment_date            date not null,
        ip_address              varchar(50) not
null,
        modified_date   date not null,
        content                 clob,
        -- is the content in HTML or plain text
(the default)
        html_p                  char(1) default
'f' check(html_p in ('t','f')),
        approved_p              char(1) default
't' check(approved_p in ('t','f'))
);
```

Users and administrators are both able to edit comments. We want to make sure that we know when a comment was last modified so that we can offer the administrator a "recently modified comments page". Rather than painstakingly go through all of our Web scripts that insert or update comments, we can specify an invariant in Oracle that "after every time someone touches the `general_comments` table, make sure that the `modified_date` column is set equal to the current date-time." Here's the trigger definition:

```

create trigger general_comments_modified
before insert or update on general_comments
for each row
begin
 :new.modified_date := sysdate;
end;
/
show errors
```

We're using the PL/SQL programming language, discussed in the procedural language chapter. In this case, it is a simple `begin-end` block that sets the `:new` value of `modified_date` to the result of calling the `sysdate` function.

When using SQL*Plus, you have to provide a / character to get the program to evaluate a trigger or PL/SQL function definition. You then have to say "show errors" if you want SQL*Plus to print out what went wrong. Unless you expect to write perfect code all the time, it can be convenient to leave these SQL*Plus incantations in your .sql files.

An Audit Table Example

The canonical trigger example is the stuffing of an audit table. For example, in the data warehouse section of the ArsDigita Community System, we keep a table of user queries. Normally the SQL code for a query is kept in a `query_columns` table. However, sometimes a user might hand edit the generated SQL code, in which case we simply store that in the `query_sql`queries table. The SQL code for a query might be very important to a business and might have taken years to evolve. Even if we have good RDBMS backups, we don't want it getting erased because of a careless mouse click. So we add a `queries_audit` table to keep historical values of the `query_sql` column:

```
create table queries (
        query_id        integer primary key,
        query_name      varchar(100) not null,
        query_owner     not null references
users,
        definition_time date not null,
        -- if this is non-null, we just forget
about all the query_columns
        -- stuff; the user has hand edited the
SQL
        query_sql       varchar(4000)
);

create table queries_audit (
        query_id        integer not null,
        audit_time      date not null,
        query_sql       varchar(4000)
);
```

Note first that `queries_audit` has no primary key. If we were to make `query_id` the primary key, we'd only be able to store one history item per query, which is not our intent.

How to keep this table filled? We could do it by making sure that every Web script that might update the `query_sql` column inserts a row in `queries_audit` when appropriate. But how to enforce this after we've handed off our code to other programmers? Much better to let the RDBMS enforce the auditing:

```
create or replace trigger queries_audit_sql
before update on queries
for each row
when (old.query_sql is not null and
(new.query_sql is null or old.query_sql <>
new.query_sql))
begin
  insert into queries_audit (query_id,
audit_time, query_sql)
  values
  (:old.query_id, sysdate, :old.query_sql);
end;
```

The structure of a row-level trigger is the following:

```
CREATE OR REPLACE TRIGGER ***trigger name***
***when*** ON ***which table***
FOR EACH ROW
***conditions for firing***
begin
  ***stuff to do***
end;
```

Let's go back and look at our trigger:

- It is named `queries_audit_sql`; this is really of no consequence so long as it doesn't conflict with the names of other triggers.
- It will be run `before update`, i.e., only when someone is executing an SQL UPDATE

statement.

- It will be run only when someone is updating the table `queries`.
- It will be run only when the old value of `query_sql` is not null; we don't want to fill our audit table with NULLs.
- It will be run only when the new value of `query_sql` is different from the old value; we don't want to fill our audit table with rows because someone happens to be updating another column in `queries`. Note that SQL's three-valued logic forces us to put in an extra test for `new.query_sql is null` because `old.query_sql <> new.query_sql` will not evaluate to true when `new.query_sql` is NULL (a user wiping out the custom SQL altogether; a very important case to audit).

Reference

Next: views

philg@mit.edu

Add a comment