

内部を知って業務に活かす

PostgreSQL 研究所

石井 達夫 ISHII Tatsuo
ishii@postgresql.org



第3回 バックエンドの中身を解析(2) ~リライト処理

はじめに

PostgreSQLの内部構造について解説する本連載も第3回目となりました。ここで、ここまでにお話ししたことを振り返ってみましょう。

第1回目では、

- PostgreSQLのプロセス構造
- ソースツリー
- デバッガを使って実行の流れを追う方法
- tagsの使い方

といった話題を取り上げ、PostgreSQLのソースコードを追う準備をしました。

第2回目は、問い合わせ処理の流れ(図1)に沿って、

- SQLのパーズ処理
- パース結果のアナライズ処理

を解説しました。

第3回目の今回は、アナライズ処理に続く「リライト処理」について解説することにします。

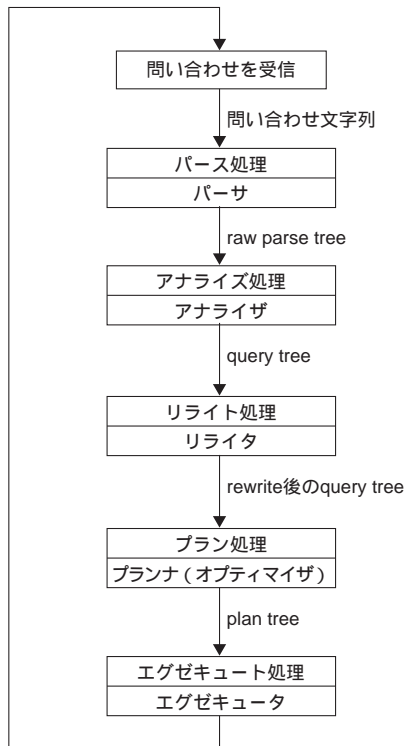
今回から PostgreSQL 8.0.3 がターゲット

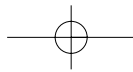
前はPostgreSQL 8.0.2を前提にお話ししましたが、今回からは5月にリリースされたPostgreSQL 8.0.3のソースを解析の対象にします。8.0.3は8.0.2に対するバグ修正版という位置付けであり、本質的な部分に変更はないので前回までの記事内容を修正する必要はないはずです。

ちょっと寄り道

PostgreSQLの内部の処理は図1に示したとおりですが、各処理から共通に呼ばれるモジュールがあります。これらのモジュールは縁の下の力持ちのような存在で、

図1 問い合わせ処理の流れ





バックエンドの中身を解析 (2)

第3回

~リライト処理



目立たないながらも重要な役割を果たしていますし、これらのモジュールを理解しておけばソースコードを読み解く際の助けにもなります。

今回はそれらのモジュールのうち、メモリマネージャについてお話しします。他のモジュールについても順次連載の中で取り上げていきます。



メモリマネージャとは

C言語で書かれたプログラムでは、標準ライブラリが提供する malloc/free などの関数を使ってメモリを管理します。しかし malloc/free をそのまま使うのは、PostgreSQL ではいろいろ問題があります。

- malloc で獲得したメモリは必ず free で解放しなければなりません。それを忘れるとメモリリークを起こす。逆に2回以上 free してしまうとプログラムが異常終了するなどの問題を起こす
- 小さなメモリを多数獲得、解放を繰り返すと効率が悪い

そこで PostgreSQL では、malloc/free の上にメモリマネージャというレイヤを設け、これらの問題を解決しています^{注1}。



メモリマネージャのインタフェース

PostgreSQL のバックエンドの中では、ほとんどの場合、malloc や free を直接呼び出しません。その代わりに、palloc や pfree という関数（実際にはマクロにな

っています）を呼び出します。引数は malloc, free とまったく同じですので、もともと PostgreSQL 用に作られていなかったプログラムを移植してバックエンドに取り込むことも容易です。palloc, pfree 以外にも各種の関数が用意されています。それらを表1に示します。

なお、palloc などのメモリアロケーションルーチンは、メモリが獲得できなかったときに NULL を返しません。それどころか、呼び出し元にも戻りません。その代わりに、内部で longjmp を呼び出して現在の処理を打ち切り、トランザクションをアボートして再び次の問い合わせの入力待ちに戻ります。ですから、palloc などの戻り値を呼び出し元でチェックする必要はないのです。このため、malloc などに付きものの面倒なエラー処理をする必要もありません。このあたりは PostgreSQL は実に良くできていると思います^{注2}。

このような挙動は、palloc に限らず PostgreSQL の他の部分でも同じで、リスト1やリスト2のように、エラー処理関数（実際にはマクロ）の elog や ereport が第1引数に ERROR または FATAL で呼び出されると、

リスト1 elog の使用例

```
if (nnames < 2) /* parser messed up */
    elog(ERROR, "must specify relation and attribute");
```

リスト2 ereport の使用例

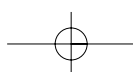
```
if (def->arg == NULL)
    ereport(ERROR,
            (errcode(ERRCODE_SYNTAX_ERROR),
             errmsg("%s requires a numeric value",
                    def->defname)));
```

表1 PostgreSQL のメモリ管理関数

関数 (マクロ) 名	引数	説明
palloc	バイト数	メモリを指定バイト数獲得する
pfree	アドレス	palloc で獲得したメモリを解放する
palloc0	バイト数	メモリを指定バイト数獲得し、0クリアする
repalloc	アドレス, バイト数	realloc の代替
palloc0fast	バイト数	palloc0 の高速版。引数が定数の場合のみ使用する
pstrdup	アドレス	strdup の代替

注1) メモリマネージャのソースコードは、src/backend/utills/mmgr/以下にあります。関連するヘッダファイルはsrc/include/utills/以下にあります。また、メモリマネージャに関する詳細な解説はsrc/backend/utills/mmgr/READMEにあります。

注2) もし palloc の戻り値をいちいちチェックしてエラー処理しなければならないとしたら、ソースコード量は今の1.5倍くらいになっていたかもしれません。



PostgreSQL 研究所

内部を知って業務に活かす

そこから下のソースコードは実行されず、次のトランザクションの入力待ちまで一気に戻ります^{注3}。ソースコードを読む上で混乱しやすいポイントなので注意しましょう。



メモリアネージャによるメモリ獲得

メモリアネージャは多数の小さなメモリの獲得を効率化するために「ブロック」という単位でmallocを使ってメモリを獲得します。ブロックサイズは8Kバイトです。8Kバイトに満たない小さなメモリの獲得要求は、メモリアネージャが内部的に管理します。



メモリコンテキスト

PostgreSQLのメモリアネージャには「メモリコンテキスト (memory context)」という概念があります。メモリコンテキストには、メモリアネージャが管理するメモリが所属します。メモリコンテキストは自由に作るができます。

「カレントメモリコンテキスト (current memory context)」は特に現在のメモリコンテキストを指します。pallocなどで獲得したメモリは、暗黙の内にカレントメモリコンテキストに所属します。

PostgreSQLがトランザクションを実行中は、専用のメモリコンテキストが作られます。トランザクション処理は複雑な作業なので、たくさんのメモリが獲得されます。トランザクションが終了すれば、それらのメモリはもはや必要ありませんので、そのメモリコンテキストに所属するメモリをすべて解放することによって、メモリリークを起こすことなく安全にメモリを解放できます。

メモリコンテキストは、さらに子供のメモリコンテキストを作ることができます。大きな処理の中の一部の処理を子供のメモリコンテキストに対応するようにしておけば、その処理が終わったときに子供のメモリコンテキストを解放することによって、大きな処理が終わるまでにシステムのメモリを全部使い果たしてしまうようなこともなくなります。親のメモリコンテキストを解放すると、子供のメモリコンテキストに所属するメモリも自動的に解放されるので、処理が簡単に

なるとともに、不注意によるメモリリークを防ぐことができます。



リライト処理とは

それでは今回の本題であるリライト処理を見ていきましょう。

PostgreSQLでは、クエリを内部的に書き換えることによってVIEWやRULEを実装しています。この処理のことをリライト処理と呼びます。



リライト処理のソースコード

リライト処理のソースコードは、src/backend/rewrite/以下にあります。また、関連するヘッダファイルはsrc/include/rewrite/以下にあります。

なお、リライト処理に関しては、PostgreSQL付属のマニュアル(第33章ルールシステム)にもわりと詳しく説明されています。



ルールの使用例

リライトルールは、CREATE RULEコマンドで定義します。ルールを使ったことのある方は意外と少ないと思いますので、ついでにここでちょっと実用的なルールの使い方の例を紹介しましょう。

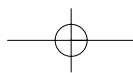
継承を使ったテーブルのパーティショニング
商用データベースには、大きなテーブルを複数のテーブルに分けることによってパフォーマンスを向上させる「テーブルパーティショニング」という機能が備わっているものがあります。テーブルパーティショニングを使うと、テーブルが複数に分かれていることを意識する必要がなくなるので便利です。

PostgreSQLはテーブルパーティショニングを直接にはサポートしていませんが、継承を使って同等のことが可能です。このときにルールが役に立ちます。

早速テーブルパーティショニングに挑戦してみましょう。

まず、「親」となるテーブルを作ります。

注3) elogは古いインタフェースで、現在ではereportの利用が推奨されています。詳細はsrc/backend/utils/error/をご覧ください。



バックエンドの中身を解析 (2)

第3回

~リライト処理



```
CREATE TABLE t1(i INTEGER PRIMARY KEY,
j INTEGER);
```

このテーブルがアプリケーションから検索や更新の対象となるテーブルです。しかし、実際にはこれはダミーで、データの実体は「子」のテーブルに格納します。それを定義しましょう(図2)。

```
CREATE TABLE t11() INHERITS(t1);
ALTER TABLE t11 ADD PRIMARY KEY(i);
CREATE TABLE t12() INHERITS(t1);
ALTER TABLE t12 ADD PRIMARY KEY(i);
```

ここではt11とt12の2つの子テーブルを定義しました。t11とt12へのデータの分担の仕方は必要に応じて適当に決めれば良いのですが、ここではiが100未満ならt11, 100以上ならt12ということにしました。

ところでt11とt12の列定義が見当たりませんが、これで問題ありません。親テーブルから継承されるからです。ただし、インデックスや主キーといった属性は継承されないため、ALTER TABLE コマンドで後から主キーを追加しています。

これで一応準備ができました。検索や更新、削除はt1テーブルに対して行えばよく、後はPostgreSQLがt11とt12に自動的に振り分けてくれます^{注4}。

なお、t1に対するSELECTでは、FOR UPDATEが使用できないので注意してください。FOR UPDATEを使用する必要がある場合は、直接子テーブルに対して実行してください。

INSERTでルールを活用

問題はINSERTです。t1にINSERTすると、そのままt1にデータが入ってしまいます。PostgreSQLは「iが100未満ならt11, 100以上ならt12」ということなど知りませんから当然です。

ルールを使えばこの問題に対処できます。

まず、t11にデータを振り分けるルールを作ります

(リスト3)。詳細はCREATE RULEのマニュアルを参照していただくとして、簡単に解説しておきます。

t1rule1はルールの名前です。AS ON以下でどのようなアクションでルールが起動されるかを指定します。ここでは当然「AS ON INSERT TO t1」ということで、t1にINSERTされたときとします。WHERE句はルールが起動される条件です。「NEW」はCREATE RULE固有の予約語で、新規に登録される行を表す仮想的なテーブルです。ここではご覧のようにiが100未満という条件を付けています。「DO INSTEAD ...」は、指定されたINSERT文の代わりに、INTO t11 VALUES(NEW.i, NEW.j)を実行することを示します。

同様に、t12のためのルールも設定します(リスト4)。

以上で、t1にiが100未満のデータが挿入されたらt11, 100以上ならt12に挿入するという、ルールによるパーティショニングができました。

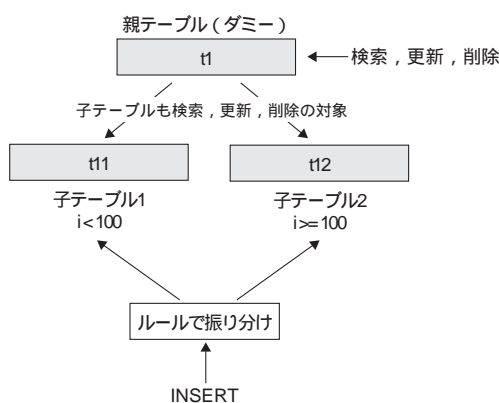


リライトルールの格納場所

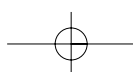
さて、CREATE RULEで作成されたルールはどこに格納されるのでしょうか？システムカタログという特別なテーブルに格納されます^{注5}。

システムカタログもテーブルですから、図3のように、その中身をSELECT文で確認することができます。

図2 問い合わせ処理の流れ



注4) 実際には、すべての子テーブルに検索や更新、削除が実施されてしまいます。ただ、WHERE句の条件にマッチしないテーブルでは処理が実行されないために、あたかも該当のデータを保持している子テーブルだけが処理の対象になっているように見えるだけです。このため、とくに検索処理において不要な子テーブルまで検索されてしまうことがパフォーマンス上の問題になります。そこでPostgreSQL 8.1では、Constraint Exclusionという技術を使って不要な子テーブルが検索対象にならないようにしており、パフォーマンスの向上が期待できます。



PostgreSQL 研究所

内部を知って業務に活かす

す。

rulename はルール名です。ev_class はテーブル定義を格納するシステムカタログ pg_class の対応行の OID (オブジェクトID) です (この値は環境によって変わります)。さらに図4のようにして pg_class の該当行を検索することができます。relname 列が t1 であることから、このルールが t1 に適用されるべきものであることがわかります。

図3に戻り、ev_attr は通常-1です。

ev_type は適用されるアクションの種類で、3は INSERT を表しています。

is_instead が t1 なら、DO INSTEAD ルールであることを示します。

ev_qual は、CREATE RULE のときに指定した

WHERE 句のパースツリーを文字列で表したものです。ev_qual をもう少し見やすくインデントを付けると図5のようになります。OPEXPR は比較演算子「<」を表現しています。この演算子の引数2つが「args」以下の VAR と CONST です。VAR のほうは「NEW」に対応し、CONST のほうは「100」に対応します。定数の型が int4 (consttype = 23)、長さが4バイト (constlen = 4)、定数の値が100 (constvalue 4 [100 0 0 0]) ということになります。とりあえず WHERE NEW.i < 100 を表現している雰囲気を感じ取っていただければここでは充分です。

ev_action は、「INSERT INTO t11 VALUES(NEW.i, NEW.j)」のパースツリーの文字列表現です。

このように、CREATE RULE で定義したルールはシ

リスト3 t11 にデータを振り分けるルール

```
CREATE RULE t1rule1 AS ON INSERT TO t1
WHERE NEW.i < 100 DO INSTEAD INSERT INTO t11 VALUES(NEW.i, NEW.j);
```

リスト4 t12 にデータを振り分けるルール

```
CREATE RULE t1rule2 AS ON INSERT TO t1
WHERE NEW.i >= 100 DO INSTEAD INSERT INTO t12 VALUES(NEW.i, NEW.j);
```

図3 システムカタログの中身

```
SELECT * FROM pg_rewrite WHERE rulename = 't1rule1';
[略]
rulename | t1rule1
ev_class | 5565177
ev_attr  | -1
ev_type  | 3
is_instead | t
ev_qual  | {OPEXPR :opno 97 :opfuncid 0 :opresulttype 16 :opretset false :args ({VAR :varno 2 :varattno 1 :vartype 23 :vartypmod -1 :varlevelsup 0 :varnoold 2 :varoattno 1} {CONST :consttype 23 :constlen 4 :constbyval true :constisnull false :constvalue 4 [ 100 0 0 0 ]})}
ev_action | ({QUERY :commandType 3 :querySource 0 :canSetTag true :utilityStmt <> :resultRelation 3 :into <> :hasAggs false :hasSubLinks false :rtable ({RTE :alias {ALIAS :aliasname *OLD* :colnames <>} :eref {ALIAS :aliasname *OLD* :colnames ("i" "j")} :rtekind 0 :relid 5565177 :inh false :inFromCl false :requiredPerms 0 :checkAsUser 1} {RTE :alias {ALIAS :aliasname *NEW* :colnames <>} :eref {ALIAS :aliasname *NEW* :colnames ("i" "j")} :rtekind 0 :relid 5565177 :inh false :inFromCl false :requiredPerms 2 :checkAsUser 1} {RTE :alias <> :eref {ALIAS :aliasname t11 :colnames ("i" "j")} :rtekind 0 :relid 5565181 :inh false :inFromCl false :requiredPerms 1 :checkAsUser 1}) :jointree {FROMEXPR :fromList <> :quals <>} :rowMarks <> :targetList ({TARGETENTRY :resdom {RESDOM :resno 1 :restype 23 :restypmod -1 :resname i :ressortgroupref 0 :resorigtbl 0 :resorigcol 0 :resjunk false} :expr {VAR :varno 2 :varattno 1 :vartype 23 :vartypmod -1 :varlevelsup 0 :varnoold 2 :varoattno 1}} {TARGETENTRY :resdom {RESDOM :resno 2 :restype 23 :restypmod -1 :resname j :ressortgroupref 0 :resorigtbl 0 :resorigcol 0 :resjunk false} :expr {VAR :varno 2 :varattno 2 :vartype 23 :vartypmod -1 :varlevelsup 0 :varnoold 2 :varoattno 2}}) :groupClause <> :havingQual <> :distinctClause <> :sortClause <> :limitOffset <> :limitCount <> :setOperations <> :resultRelations <>})
```

注5) システムカタログに関するドキュメントは、PostgreSQL 付属ドキュメントの「第41章 System Catalogs」にあります。システムカタログのソースコード上の定義は src/include/catalog/以下にあり、すべてCのヘッダファイルの形式になっています。ヘッダファイルの名前がシステムカタログのテーブル名と一致しています。

バックエンドの中身を解析 (2) 第3回 ~リライト処理



ステムカタログに格納され、リライト処理で利用されます。

リライト処理の概要

リライト処理のメインプログラムはQueryRewrite という関数です (src/backend/rewrite/rewriteHandler.c)。この関数はパース処理が作成したパースツリーを引数に取り、書き換えたパースツリーを返します。

QueryRewrite の処理は以下のような3つのステージに分かれています。

- ① SELECT 以外のルールを適用する
- ② SELECT のルール (実際にはVIEWに関するルール) を適用する
- ③ コマンドリザルトタグ (command result tag) を調整する^{注6}

それではステージごとに処理を見ていきましょう。

図4 pg_class の該当行の検索

```
SELECT * FROM pg_class WHERE oid = 5565177;
-[ RECORD 1 ]-----
relname      | t1
relnamespace | 2200
reltype      | 5565178
relowner     | 1
relam        | 0
relfilenode  | 5565177
reltablespace | 0
relpages     | 0
reltuples    | 0
reltoastrelid | 0
reltoastidxid | 0
relhasindex  | t
relisshared  | f
relkind      | r
relnatts     | 2
relchecks    | 0
reltriggers  | 0
relukeys     | 0
relfkeys     | 0
relrefs      | 0
relhasoids   | t
relhaspkey   | t
relhasrules  | t
relhassubclass | t
relacl       |
```

① SELECT 以外のルールを適用する

この部分は、RewriteQuery (src/backend/rewrite/rewriteHandler.c) という関数で処理されます。処理の対象となるのは、UPDATE、INSERT、DELETE です。

INSERT 文や UPDATE 文のターゲットリストが省略されていたらそれを補う
たとえば、

```
CREATE TABLE t1(i INTEGER,
j INTEGER DEFAULT 100);
```

のようなテーブルがあったとして、

```
INSERT INTO t1 VALUES(1, 2);
```

のようなINSERT 文を実行すると、ターゲットリストを拡張して、

```
INSERT INTO t1(i, j) VALUES(1, 2);
```

のようにします。また、

図5 ev_qual

```
{OPEXPR
:opno 97
:opfuncid 0
:opresulttype 16
:opretset false
:args
(
{VAR
:varno 2
:varattno 1
:vartype 23
:vartypmod -1
:varlevelsup 0
:varnoold 2
:varoattno 1}
{CONST
:consttype 23
:constlen 4
:constbyval true
:constisnull false
:constvalue 4 [ 100 0 0 0 ]}
)
}
```

注6) コマンドリザルトタグとは、psql から UPDATE などの SQL を実行したときに表示される「UPDATE」のような文字列のことです。ルールを適用すると、INSERT を実行しても実際にはUPDATE が実行されたりすることがあるので、このような処理が必要になります。

PostgreSQL 研究所

内部を知って業務に活かす

```
INSERT INTO t1 DEFAULT VALUES;
```

や,

```
INSERT INTO t1 VALUES(1, DEFAULT);
```

のようにデフォルト値が指定されているものは,

```
INSERT INTO t1(i, j) VALUES(NULL, 100);
```

あるいは,

```
INSERT INTO t1(i, j) VALUES(1, 100);
```

のように、決められたデフォルト値を設定します。UPDATE の場合も同様です。

ルールを適用する

実際にルールを適用するのはfireRIRules という内部関数で、引数としてオリジナルパースツリーと適用すべきルールなどを取り、順次ルールを適用していきます。ルールはすでにパースツリーの形でシステムカタログに保存されていますから、ルールの適用処理は思っているほど難しいものではありません。

すなわち、

- DO ALSO ならパースツリーにルールで指定されたアクションを追加
- DO INSTEAD ならパースツリーをルールで指定されたアクションに置き換え

というのが基本です。もちろん細かな調整はたくさんありますが、ここではそのうちの一つを紹介しましょう。

先ほどのテーブルパーティショニングにおけるルールの例では、

```
WHERE NEW.i < 100 DO INSTEAD INSERT INTO t11
VALUES(NEW.i, NEW.j);
```

```
WHERE NEW.i >= 100 DO INSTEAD INSERT INTO t12
VALUES(NEW.i, NEW.j);
```

のように必ずどちらかのルールが適用されるようになっていましたが、もし仮にWHERE NEW.i < 100 DO INSTEAD INSERT INTO t11 VALUES(NEW.i, NEW.j) だけが定義していないとすると、i が100以上の場合

にはオリジナルの問い合わせを実行する必要があります。このような場合には、PostgreSQL NEW.i < 100 の否定の条件文 NEW.i >= 100 を作り、そのときにオリジナルの問い合わせを実行するようにすればよいわけです。たとえば、オリジナルの問い合わせが、

```
INSERT INTO t1 VALUES(1,2);
```

ならば、

```
INSERT INTO t1 VALUES(1,2) WHERE 1 >= 100;
```

が追加実行すべき問い合わせとしてfireRIRules から返却されることとなります (INSERT 文にWHERE 句が付くのはおかしいのですが、PostgreSQL のエンジンはこうしたSQL を内部で実行することができます)。つまり、最終的には、

```
INSERT INTO t11 VALUES(1,2) WHERE 1 < 100;
```

```
INSERT INTO t1 VALUES(1,2) WHERE 1 >= 100;
```

が実行すべき問い合わせとなりますが、もちろん2番目のWHERE 句は成立しませんから、実際に実行されるのは最初のINSERT 文だけになります。



② SELECT のルールを適用する

ここでは、実際にはVIEW を展開する処理が行われます。これも先ほど出てきたfireRIRules で処理されます。

RTE の取得とチェック

レンジテーブルエントリ (Range Table Entries : RTE), すなわちSELECT * FROM ...のFROM 句の部分がVIEW の展開処理の対象となるので、それを取ります。ただしRTE がテーブルでない場合 (関数など) は無視します。また、RTE には含まれているものの、実際に問い合わせで使われていないテーブルも無視します。

RTE がサブクエリの場合は、再帰的にサブクエリの内側を処理していきます。

VIEW の展開

ApplyRetrieveRule という関数で実際にON SELECT

バックエンドの中身を解析(2)

~リライト処理

第3回



ルールの適用, すなわちVIEWの展開を行います^{注7}.

まず, VIEWにさらにVIEWが含まれている可能性があるので, fireRIRrules を呼び出して再帰的に展開します.

次にVIEWをルールアクションで置き換え, VIEWを展開します. これは, VIEWをルールアクションのサブクエリ呼び出しで置き換えることによって実現します. たとえば,

```
CREATE VIEW myview AS SELECT * FROM t1;
```

というVIEWがあったとして,

```
SELECT * FROM myview;
```

という呼び出しは,

```
SELECT * FROM (SELECT * FROM t1);
```

と展開されます.

最後に, VIEW自体にはSELECT ... FOR UPDATE

が適用されないようにし, その代わりに, 元のテーブルにSELECT ... FOR UPDATE が適用されるようにします.



③ コマンドリザルトタグの調整

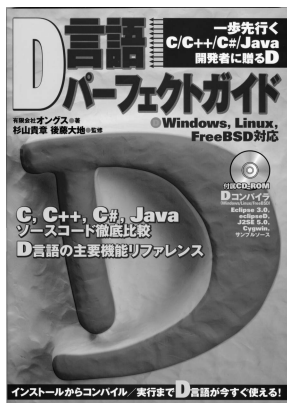
問い合わせ書き換えの結果, もしオリジナルの問い合わせが残っていればその問い合わせのコマンドリザルトタグを採用します. そうでなければ, INSTEADで指定されたアクションのうち, 最後の問い合わせのコマンドリザルトタグを採用します.



次回は?

今回はリライト処理について解説しました. 次回はよいよプラン処理を解説します. プラン処理は別名オプティマイザとも呼ばれ, DBエンジンの心臓部の一つとも言える重要な部分です. ご期待ください. **Web**

注7) ここで「Retrieve」という単語が使われていますが, これはPostgreSQLの先祖であるPOSTGRESが使用していた問い合わせ言語PostQuelで, SELECTに相当するコマンドがRetrieveであったころの名残です.



著者: オングス監修: 杉山貴章, 後藤大地
判型: B5判 256ページ
ISBN: 4-7741-2208-4
発売日: 2004/12/22
価格: 1974円 (1880円)



CD-ROM収録

Dコンパイラ
Eclipse 3.0, eclipseD, J2SE 5.0
Cygwin, サンプルソース

D言語 パーフェクトガイド

一歩先行く
C/C++/C#/Java
プログラマに贈る

最近ではJavaやC#などがアプリケーション開発用のプログラミング言語として定着してきましたが, 誰にでも使い易くという目的が優先されたその仕様に対する不満の声も随所で聞こえており, こうした理由からJavaやC#などを使わずにC/C++を使い続けているユーザも多く, 特に古くからのベテランプログラマにその傾向は強いようです. そこでC/C++に代わるプログラミング言語として一部で注目を集めているのがD言語です. D言語は位置づけとしてはCやC++の後継とされており, 多くのプログラマが有用と考える機能をこれらの言語に付け加えたようなものになっています. 既存の言語のいい部分を残し不要な部分は削るといったコンセプトの元に設計されたこの言語はプログラマの間に徐々に浸透しつつあります. 本書ではこのD言語の全貌をC/C++/Java/C#のソースコードと比較しながら, 詳細に解説します.

**インストールからコンパイル/実行まで
D言語が今すぐ使える!**

技術評論社