

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4回 プラン処理(1)
~ オプティマイザとクエリの整形


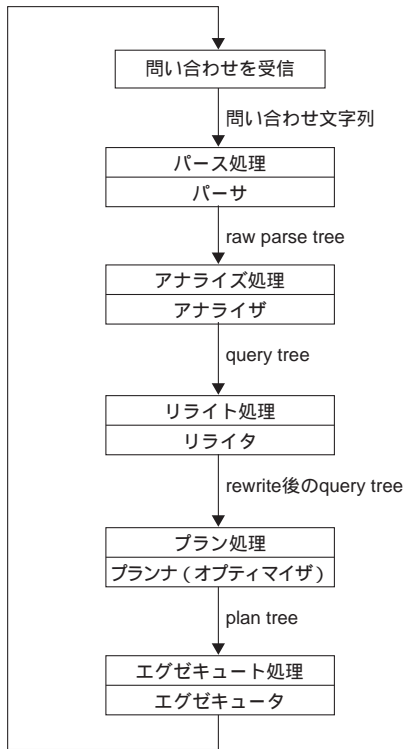

 今回はプラン処理 = 問い合わせオプティマイザの解説
PostgreSQL の内部構造を解説する連載の第4回目は、はいよいよ「プラン処理」です(図1)。
プラン処理はユーザからの問い合わせを実行する方法を考案し、その中でもっとも実行コストが低い(すなわちもっとも速く実行できる)実行プランを生成し


図1 問い合わせ処理の流れ




ます。言い換えると最適と思われるプランを生成するのがプラン処理ということになり、その意味で(問い合わせ)「オプティマイザ(optimizer)」とも呼ばれます。

 オプティマイザの重要性

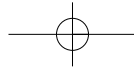
データベース処理はディスクをアクセスする処理が多いので、問い合わせプランの出来によっては数百倍も実行時間が違うことがあります。したがって、プラン処理 = オプティマイザの品質は極めて重要で、PostgreSQL においてもオプティマイザの改良には多くの努力が注ぎ込まれており、バージョンごとにどんどんオプティマイザは進化しています。本稿では PostgreSQL 8.0.3 のオプティマイザを解説しますが、もしかしたら PostgreSQL のバージョンに依存した話になるかもしれないので、そのあたりはあらかじめご了承ください。

 オプティマイザの種類

世の中のデータベース製品はほとんどのものが何らかの問い合わせオプティマイズ機能を持っています。オプティマイザには大きく分けて2つのタイプがあります。

 ルールベース

ルールベースとは、たとえばインデックスがあったら必ずそれを使ってアクセスするなど、あらかじめ決まった「ルール」によって問い合わせプランを決める



プラン処理 (1) 第4回

~ オプティマイザとクエリの整形

方式です。ルールベースオプティマイザは単純で高速ですが、どんな場合にも最適なプランを選ぶわけではありません。たとえば、1ページ^{注1)}に収まる小さなテーブルでは、インデックスを使わなければ1回のディスクアクセスでデータを取得できるのに対し、インデックスを使うと少なくとも2回はディスクアクセスが必要になります。インデックスがあったら必ず使うことにしているルールベースオプティマイザでは、かえって遅いプランを選択してしまいます。



コストベース

ルールベースオプティマイザにはこのような問題があるので、PostgreSQL も含めて近代的なデータベースのほとんどはコストベースのオプティマイザを採用しています。コストベースオプティマイザでは、可能な問い合わせプランをすべて生成し、おののプランを実行するのに必要な時間(コスト)を見積もります。そしてその中でコストが最小のものをプランとして採用します。このとき、コストはテーブルの構造やインデックスの有無といった「静的」な情報だけではなく、データ量やデータの分布など、「動的」な情報も参考にして算出されます^{注2)}。

コストベースオプティマイザは、データ量が変わっても最適なプランを選択できますが、反面オプティマイザの出来によっては最適でないプランを選択してしまうことがあります。また、生成する問い合わせプランの数が膨大になって、オプティマイザが考える時間のほうが問い合わせの実行時間よりもよっぽど長い、などという本末転倒なことが起こることがあります。)

このように、コストベースオプティマイザは理想的である反面、その出来具合によってデータベースとしての性能が左右されると言っても過言ではありません。したがって、各データベース製品はオプティマイザの改良にしのぎを削っています。

今回本稿を書くにあたってPostgreSQL のオプティマイザのソースコードを詳細に調査しましたが、本当によく考えられており、改めてその完成度の高さに驚

かされました。ベースになっているpostgresのオプティマイザの出来がよかったせいもあると思いますが、やはり長い時間かけて改良されてきた成果の現れではないかと思いました。



プランタイプ

問い合わせの実行プランは、あらかじめ用意された「プランタイプ」の組み合わせでできています。プランタイプは、実際に問い合わせを実行するエグゼキュータのモジュールと対応しています。

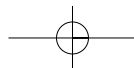
オプティマイザの役割は、これらのプランタイプをどう組み合わせたらトータルでもっとも実行時間が短くなるかを考えることにあります。逆に言うと、存在しないプランタイプはいかに優れたオプティマイザといえども使いようがありません。PostgreSQL にはリリースと共に新しいプランタイプが追加されることがあり、そのたびに飛躍的に問い合わせ性能が向上します。

表1 PostgreSQL 8.0 のプランタイプ

プランタイプ	説明
Append	下位のプランが生成したデータを追加します
Group	GROUP BY を実行します
Function Scan	行を生成する関数の実行結果を取出します
Hash Join	ハッシュ結合を実行します
Hash Aggregate	ハッシュ集約を実行します
Index Scan	インデックスを使ってテーブルをスキャンします
Limit	LIMIT 処理を実行します
Material	中間結果をメモリ上に保持します
Merge Join	マージ結合を行います
Nested Loop	ネストループ結合を実行します
Sequential Scan	テーブルを順スキャンします
SetOp Except	EXCEPT を実行します
SetOp Intersect	INTERSECT を実行します
Sort	ソート処理を実行します
Subquery Scan	副問い合わせの結果を保持します
Tid Scan	TID (Tuple ID) を使ってテーブルをスキャンします
Unique	ユニーク (重複の削除) を実行します

注1) PostgreSQL がディスクI/Oをする単位。通常8192バイト。

注2) PostgreSQL では、動的な情報を収集するのはVACUUMもしくはANALYZE コマンドの役割になっています。



PostgreSQL 研究所

内部を知って業務に活かす

ここで本稿執筆時点の最新安定版である PostgreSQL 8.0 が実装しているプランタイプを紹介いたします。表1をご覧ください。読者の方には見慣れない用語も含まれていると思いますが、今後エグゼキュータを解説するときに説明します。



プラン処理の流れ

プラン処理のメイン関数はpg_plan_query という関

数です。pg_plan_query はパース/アナライズ結果を格納したQuery 構造体^{注3}を要素に持つリストを受け取り、問い合わせ実行プランを格納したPlan 構造体のリストを出力します。

pg_plan_query 自体はたいした処理は行わず、プラン処理の実質的なメイン関数はpg_plan_query から呼び出されるplanner です。planner は単独で呼び出すこともできる独立性の高い関数で、実際EXPLAIN コマンドはplanner を呼び出して問い合わせプランを作

注3) Query 構造体については、連載第2回目(本誌Vol.27に掲載)で説明したのでそちらをご覧ください。

■ PostgreSQL 8.1 で追加された新しいプランタイプ

コラム

現在開発中のPostgreSQL 8.1 では、「ビットマップスキャン(bitmap scan)」という新しいプランタイプが追加されました。これは、メモリ中にビットマップを動的に生成し、ある種の検索条件を効率良く処理できるものです。

たとえばリストAのような問い合わせで「bar BETWEEN 1 AND 10000」を満たす行が10,000件、「buz BETWEEN 1 AND 100」を満たす行が1,000件あります。しかし、両方の条件を満たすものは100件しかないとします。インデックスはbar とbuzの両方に張ってあるものとします。

PostgreSQL 8.0 では、まず「bar BETWEEN 1 AND 10000」を満たす行を取り出し、次の中で「buz BETWEEN 1 AND 100」を満たすものを探すという動作をするので、foo から10,000行を抜き出す処理を避けることができません。

それに対して8.1では、bar, buzのインデックスを調べて条件を満たすものをそれぞれメモリ上のビットマップでオンにしたのち、ビット操作のANDを行います。これで100件のデータが該当することがわかるので、後は100件だけfooから行を取り出せば良いわけです。10,000

万件に対して100件ですから、効率が非常に良くなるのがわかります。

この例ではbar + buzのマルチカラムインデックスを追加することによって、8.0でも検索効率を上げることができますが、インデックスを追加すると更新処理の負荷が大きくなってしまいます。ビットマップスキャンはその心配もありません。

ビットマップスキャンはOR 検索にも威力を発揮

リストBのような問い合わせを考えます。この問い合わせでは、2つのBETWEEN条件の両方にマッチする行が5,000行あり、非常に重複が多くなっています。このような問い合わせでは従来インデックスを使うことができませんでした。

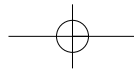
8.1では「bar BETWEEN 1 AND 10000」と「bar BETWEEN 5000 AND 15000」のそれぞれの条件に該当するデータ位置をインデックスを走査しながらメモリ上のビットマップに記録し、最後にビットマップ上でORを取って検索を実施します。インデックスが活用できるため、高速な検索ができるわけです。

リストA 問い合わせの例(1)

```
SELECT * FROM foo WHERE bar BETWEEN 1 AND 10000 AND buz BETWEEN 1 AND 100;
```

リストB 問い合わせの例(2)

```
SELECT * FROM foo WHERE bar BETWEEN 1 AND 10000 OR bar BETWEEN 5000 AND 15000;
```



プラン処理 (1) 第 4 回 ~ オプティマイザとクエリの整形

成, 表示しています。

また, planner は再帰呼び出しされることがあります。たとえばSQL 関数を実行する場合, planner は SELECT 文の実行プランを作成し, SQL 関数内部に記述されたSELECT 文を実行するために自分自身を呼び出します。

実質的なプラン処理は, planner から呼び出される subquery_planner で行います。subquery_planner は副問い合わせを見つけると, 自分自身を再帰呼び出してプラン処理を行います。



プラン処理のソースツリー

プラン処理のソースコードはsrc/backend/optimizer 以下に格納されています。optimizer 以下はさらに表2 のサブディレクトリに分かれています。

これらのソースコード行数はコメントもいれて約3 万ステップほどです。難しいことをやっている割にはコンパクトにまとまっています。これはモジュール化の徹底と, 再帰呼び出しを多用したコーディング技術の成果と言えるのではないかと思います。



subquery_planner での処理概要

subquery_planner で行われるプラン処理は大まかに言って以下のようなステップに分かれます。

ステップ 1 : 前処理

問い合わせの中には, そのまま処理するよりも変形させたほうがよいものがあります。ここではクエリツリーをより効率の良い形に書き換えます。

ステップ 2 : path 生成

同じクエリツリーに対して複数の実行方法が考えられます。たとえば, あるテーブルを検索するにも順スキャンを行う方法とインデックススキャンを行う方法が考えられるでしょう。ここでは, 考えられる問い合わせの実行方法 (path) を可能な限り生成し, プランツリーの形で出力します。

ステップ 3 : 最適な path の選択

ステップ2 で生成したpath の中からもっともコストが低い (= 実行時間の短い) ものを選択してオプティマイザの出力とします。



ステップ 1 : 前処理

それではまずステップ1 の前処理を見ていきましょう。ソースコードで言うと, src/backend/optimizer/plan/planner.c に定義されているsubquery_planner を中心に見ていくことになります。subquery_planner には, 引数としてパースツリーが渡されています^{注4}。



IN 副問い合わせの「引き上げ」

SELECT * FROM ... WHERE foo IN (SELECT...) あるいは SELECT * FROM ... foo = ANY (SELECT...) の形の一部の副問い合わせは, 副問い合わせをFROM 句に追加し, あたかもJOIN のように扱うことによって効率を高めることができます^{注5}。

結果として内側のSELECT を外側に「引っ張りあげる」ため, これを「引き上げ (pull up)」と呼びます。この処理は, pull_up_IN_clauses という関数を実行します。pull_up_IN_clauses の引数はパースツリーと, パースツリーの中のjointree->quals です。

ここから先の説明には具体例があったほうがわかりやすいので, リスト1 のような問い合わせを処理するものとします。

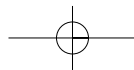
history とaccounts はPostgreSQL に付属のベンチマークツールpgbench で作成したもので, 図2 のような

表 2 optimizer ディレクトリ以下の構成

ディレクトリ	説明
plan	プラン処理の主要関数群
path	問い合わせの実行方法 (path) を生成する関数群
prep	前処理関数群
util	ユーティリティ関数群
geqo	遺伝的アルゴリズムを使ったオプティマイザ

注4) パースツリーについては本連載第2回で解説しました。ここで言っている「パースツリー」は, パースツリーにアナライズ処理の情報が追加された「クエリツリー」のことを指しています。できれば, 本誌 Vol.27 の235 ページ図3 のクエリツリーを見ながら本稿を読んでいただくとうわかりやすいでしょう。

注5) PostgreSQL 内部ではIN と=ANY は同じものとして扱われます。



PostgreSQL 研究所

内部を知って業務に活かす

構造になっています。

pull_up_IN_clauses の処理は以下になります
(実際にはほとんどの処理はconvert_IN_to_join (plan/subselect.c) で行われます)。

- ① 検索条件がINであることを確認。この例では検索条件はWHERE aid IN ...となっているのでOKです。
- ② 副問い合わせ (SELECT aid FROM accounts) で、上位レベルのSELECT文の中の変数を参照していないことを確認。これもOKです。なおこの確認作業では、対象となる副問い合わせがさらに副問い合わせを含んでいる場合はそのチェックも必要と

```

図2 history と accounts の構造
test=# \d history
                Table "public.history"
  Column |          Type          | Modifiers
-----+-----+-----
  tid    | integer                |
  bid    | integer                |
  aid    | integer                |
  delta  | integer                |
  mtime  | timestamp without time zone |
  filler | character(22)          |

test=# \d accounts
                Table "public.accounts"
  Column |          Type          | Modifiers
-----+-----+-----
  aid    | integer                | not null
  bid    | integer                |
  abalance | integer                |
  filler | character(84)          |

Indexes:
  "accounts_pkey" PRIMARY KEY, btree (aid)
    
```

なります。

- ③ 左項 (left hand side) が揮発性 (volatile) 関数かどうかをチェック。ここでは左項はaidですので、もちろんvolatile関数ではないので問題ありません^{注6}。
- ④ 無事チェックが終わったので、副問い合わせに対応するRTE (Range Table Entry) をaddRangeTableEntryForSubquery (parser/parse_relation.c) を呼び出して作成し、パースツリーに記録されているRTEリスト (構造体メンバ名: rtable) に追加します。
- ⑤ IN clause info を作成します。このようにしてJOINに変換されたINに関してはエグゼキュータ実行時に追加の情報が必要になります。それをInClauseInfo構造体を作成し、パースツリーのin_info_listにセットします。InClauseInfo構造体の定義をリスト2に示します。
- ⑥ WHERE 句の検索条件を作ります。
- ⑦ 元のWHERE 句の検索条件を⑥で置き換えます。

以上の結果、例題の問い合わせは、リスト1からリスト3へと書き換えられました (厳密に言うとちょっと違います)。

```

リスト1 問い合わせの例
SELECT * FROM history WHERE aid IN (SELECT aid from accounts);
    
```

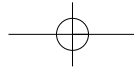
```

リスト2 InClauseInfo 構造体の定義
typedef struct InClauseInfo
{
    NodeTag type; /* T_InClauseInfo 固定 */
    Relids lefthand; /* 左項に含まれるテーブル */
    Relids righthand; /* 右項 (副問い合わせ) に含まれるテーブル */
    List *sub_targetlist; /* 元の副問い合わせのターゲットリスト */
} InClauseInfo;
    
```

```

リスト3 リスト1 の書き換え結果
SELECT history.* FROM history, (SELECT aid FROM accounts) AS IN_subquery WHERE history.aid =
IN_subquery.aid;
    
```

注6) volatile関数の概念については、PostgreSQL 付属ドキュメントのリファレンスマニュアル、CREATE FUNCTIONの項をご覧ください。



プラン処理 (1) 第4回

~ オプティマイザとクエリの整形

実際、書き換え前のSQL文と書き換え後のSQL文の実行プランをEXPLAINで見ると、非常によく似ているのがわかります(図3)。違うのは「Nested Loop IN Join」と「Nested Loop」のところだけです。

ただのJOINだと、たとえば、あるhistory.aidにマッチするIN_subquery.aidが複数あると、同じhistory.aidを持つ行が結果として複数生成されてしまいます。そこで、PostgreSQL内部ではこのJOINがINを書き換えたことによって生成されたものであることを認識していて、それがNested Loop IN Joinとなって表現されているのです。



FROM 句内の副問い合わせの「引き上げ」

先ほど例に示したリスト3のような問い合わせは、よく考えると副問い合わせを使わなくても実行できます。つまり、リスト4でよいわけです。このような書き換えを行うのがpull_up_subqueriesです。対象となる副問い合わせがさらに副問い合わせを含んでいる場合は、自分自身を呼び出して処理を続けます。

逆にこのような書き換えができないのは以下のような条件を1つでも満たすものです。

- INTERSECT/EXCEPT を使っているもの
- GROUP BY, HAVING, ORDER BY, DISTINCT, LIMIT を使っているもの

- 行を返す関数を呼び出しているもの
- FROM 句がないもの



ターゲットリストを正規化

ターゲットリストの正規化を行います。

eval_const_expressions を呼び出して定数を含む式を可能な限り事前に計算、評価しておきます。そのようなものとしては、以下のようなものがあります。

```
SELECT 2+2 → SELECT 4
SELECT foo OR TRUE → SELECT TRUE
```



検索条件を正規化

検索条件の正規化を行います。ここでは無駄なAND/OR条件をもっと高速化しやすい形に変形します。



AND/OR を「平坦化」

PostgreSQLのパーサはANDやORなどの論理演算子は2項演算子であると仮定しているため、たとえば、

```
(A = 1) OR (A = 2) OR (A = 3)
```

のような検索条件から、

```
(OR (A = 1) (OR (A = 2) (OR (A = 3))))
```

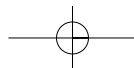
図3 実行プランの比較

```
test=# EXPLAIN SELECT * FROM history WHERE aid IN (SELECT aid from accounts);
          QUERY PLAN
-----
Nested Loop IN Join (cost=0.00..3811.96 rows=630 width=94)
-> Seq Scan on history (cost=0.00..16.30 rows=630 width=94)
-> Index Scan using accounts_pkey on accounts (cost=0.00..6.01 rows=1 width=4)
    Index Cond: ("outer".aid = accounts.aid)
(4 rows)
```

```
test=# EXPLAIN SELECT history.* FROM history, (SELECT aid FROM accounts) AS IN_subquery WHERE history.aid = IN_subquery.aid;
          QUERY PLAN
-----
Nested Loop (cost=0.00..3811.96 rows=630 width=94)
-> Seq Scan on history (cost=0.00..16.30 rows=630 width=94)
-> Index Scan using accounts_pkey on accounts (cost=0.00..6.01 rows=1 width=4)
    Index Cond: ("outer".aid = accounts.aid)
(4 rows)
```

リスト4 副問い合わせを使わない形

```
SELECT history.* FROM history, accounts WHERE history.aid = accounts.aid;
```



PostgreSQL 研究所

内部を知って業務に活かす

のようなパースツリーを作ります。ちょっとわかりにくいので図にすると、図4のようになります。

しかし、プランナやエグゼキュータは実際にはn項論理演算子を扱うことができ、木構造にしておくのは無駄なので、

```
(OR (A = 1) (A = 2) (A = 3))
```

という形に変形します。つまり、木になっていたのを「平坦化」したわけです。これも図にしておくと、図5のようになります。



NOT の最適化

以下のような変形を行います。

```
(NOT (< A B)) => (>= A B)
(NOT (AND A B)) => (OR (NOT A) (NOT B))
(NOT (OR A B)) => (AND (NOT A) (NOT B))
(NOT (NOT A)) => (A)
```



OR を含む論理式の正規化

以下のような変形を行います。

```
((A AND B) OR (A AND C)) => (A AND (B OR C))
(A AND B) OR (A) => A
```



実行可能プランに変換

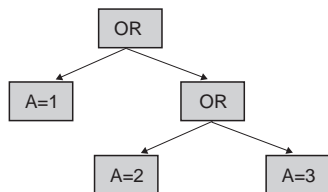
検索条件に副問い合わせが含まれていたなら、そのための実行プランの構造体であるSubPlanを作ります。



HAVING 節の正規化

HAVING があれば、そこに含まれる論理式を正規

図4 (A = 1) OR (A = 2) OR (A = 3) に対応するパースツリー



化します。その手法は「検索条件を正規化」と同じです。



LIMIT 節の正規化

「ターゲットリストの正規化」と同じ処理をLIMIT/OFFSET 節にも適用します。



HAVING 節の変換

GROUP BY 節と集約関数と関連づけられない HAVING 節はWHERE 節と同じなので、WHERE 節に変換します。



OUTER JOIN の変換

ある種の OUTER JOIN は一定の条件のもとで INNER JOIN に変換できます。INNER JOIN のほうが最適化しやすいので、この変換は有用です。

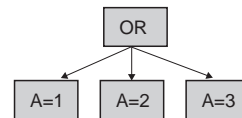
例を挙げます。図6のようなテーブルaとbがあったとします。LEFT OUTER JOIN は図7のような結果になります。結果の2行目はx = 2にマッチする行がないので、b.x とb.yはNULLになっています。

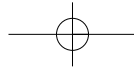
ここで、WHERE b.y = 42 という条件をこの問い合わせに追加すると、図8の結果が得られます。先ほどの検索結果の2行目のyがNULLになっているため、WHERE 句が成立せず、削除されました。

このようにWHERE 句に右側のテーブル(ここではb)に関する検索条件を持つLEFT OUTER JOINでは、その検索条件がIS NULLでない限り、OUTER JOINの結果、右側のテーブルの列にNULLが埋め込まれた行(先ほどの検索結果の2行目)は必ず削除されます。ということは、これはINNER JOINと同じになるわけです。

そこで、図7のSELECT文はリスト5のようなINNER JOINに置き換えることができます。

図5 プランナが「平坦化」した結果





プラン処理 (1) 第4回
~ オプティマイザとクエリの整形

JOIN 句を使わずに、リスト6のように書くこともできます。

もちろんRIGHT OUTER JOIN に関しても同じような変換が可能です。FULL OUTER JOIN では、左右のOUTER JOIN に対して同じ変換が可能です。



次回は

プランナの前処理まで説明したところで誌面が尽きてしまいました。次回はいよいよ「ステップ2: path 生成」から解説します。 **Web**

図6 テーブルの例

```
CREATE TABLE a(x INTEGER, y INTEGER);
CREATE TABLE b(x INTEGER, y INTEGER);

test=# SELECT * FROM a;
 x | y
---+---
 1 | 10
 2 | 20
(2 rows)

test=# SELECT * FROM b;
 x | y
---+---
 1 | 42
 3 | 42
(2 rows)
```

図7 LEFT OUTER JOIN の結果

```
test=# SELECT * FROM a LEFT OUTER JOIN b ON
(a.x = b.x);
 x | y | x | y
---+---+---+---
 1 | 10 | 1 | 42
 2 | 20 |   |
(2 rows)
```

図8 条件を追加した結果

```
test=# SELECT * FROM a LEFT OUTER JOIN b ON
(a.x = b.x) WHERE b.y = 42;
 x | y | x | y
---+---+---+---
 1 | 10 | 1 | 42
(1 row)
```

リスト5 INNER JOIN に置き換え

```
SELECT * FROM a INNER JOIN b ON (a.x = b.x)
WHERE b.y = 42;
```

リスト6 JOIN 句を使わない置き換え

```
SELECT * FROM a, b WHERE a.x = b.x AND b.y =
42;
```

書いて書いて、身体で覚える《書き込み式ドリル》

すらすらと手が動くようになる

SQL 書き方ドリル



羽生章洋 著

B5判 / 304ページ
ISBN4-7741-2299-8
価格2180円+税

新しい知識は、一度読んだだけで習得することはできません。同じレベルの問題を何度も繰り返して、身体で覚えることで、初めて自分のものにすることができます。

本書は、繰り返し同じレベルのSQLを実際を書くことで、自然に、SQLを書くときの書き順と思考プロセスを身につけることができるようになっています。もちろん、PCから入力して学習することも可能です。

開発や運用の現場でも、顧客のさまざまな要求に応じたSQLを、自在に書き分ける力がつく一冊!

東京都品川区上大崎3-1-1
<http://www.gihyo.co.jp/>

技術評論社

