The Database Times

vol. 1

Hotchpotch Society

まえがき

The Database Times vol.1 をお手にとっていただきありがとうございます。

本書は，データベースシステムを中心として，様々な情報技術に関する話題を読者のみなさんにお届けすることを目的としています。みなさんは「データベースシステム」というものにどのようなイメージをお持ちでしょうか。SQL を投げるとデータを返してくれる，なんだかよくわからないけれど枯れた時代遅れの技術だというイメージを持っている人が多いのではないかと思います。

しかしながら，実際のデータベースシステムは古の技術を土台として，その時代の最先端の技術が詰め込まれた非常に魅力的な技術の集大成なのです。世界中で生み出されるデータ量がどのくらいかご存知でしょうか。2010 年までに生み出されたデータ量はなんと 1,227 ペタバイト。しかも，2020 年には 7,910,000 ペタバイトになるであろうと予測されています。人類が産み出そうとしているデータ量は，まさに爆発的な勢いで増え続けています。このデータの膨張を支える縁の下の力持ちがデータベースシステムです。加速するデータ量の増加に伴って，データベースシステムも日々進歩を続けているのです。

そんなデータベースシステムがどのようにして生まれ，そして最近ではどんなことが話題になっているのか，本書を通してその一端に触れ，楽しんで頂ければ幸いです。

2012 年 8 月

Hotchpotch Society

お品書き

■ データベースシステムの夜明け

今の世の中，データベースシステムといえばリレーショナルデータベースシステムです。そのリレーショナルデータベースシステムが，一体どのようにして生み出されたのか，そしてどのように発展を遂げていったのか，その歴史を辿ります。

■ PostgreSQL カンファレンス 2012 レポート

オープンソース DBMS 代表格の一つである PostgreSQL は，最近は業務でも用いられることがかなり増えてきているようです。そんな PostgreSQL の最新技術事情に触れられる PostgreSQL カンファレンスの参加レポートをお届けします。

■ とある世界で一番高速な Brainfuck インタプリタ

近年では SAP HANA や MemSQL などの主記憶データベースが話題となっているように，膨大するデータ量を捌ききるための「速さ」が強く求められています。データベースシステムは SQL というプログラミング言語の処理系という側面も持ちあわせており，言語処理系の高速な実装はデータベースシステムには必要不可欠です。言語処理系をいかに高速にするのか，その技術について世界最速 Brainfuck インタプリタの実装者が解説します。

■ クラウド時代の DNS

■ IPv4 がこの先生きのこるには

最近では「クラウド」という言葉が話題となっていますが，その背景にはもはや単一のコンピュータだけでシステムを構築してすべてがうまくいく時代は終焉を迎え，ネットワークによりコンピュータを有機的に結合することが必須となりつつある現代の技術トレンドがよみとれます。ネットワーク技術では今一体何が起きているのか，その最前線に迫ります。

■ 電算機技能者的同人誌執筆環境構築概論

本書は LaTeX で執筆されています。プログラマが LaTeX で本を書くというのはどういうことなのか，その開発 (執筆) 環境を紹介します。

目次

1

データベースシステムの夜明け

はやみず

データベースシステム無しでは，今日の社会は成り立たないと言って良いでしょう。社会が高度に情報化された現在，世界中で大量のデジタルデータが日々生み出され，飛び交い，消費され，そして蓄積されてゆきます。一方で，膨大なデジタルデータは，単に生み出され，蓄積されてゆくだけではゴミも同然です。必要なときに必要なデータが取り出せるよう，適切に管理してゆかなければなりません。そのための根幹たる存在がデータベースシステムなのです。

今日においては，「データベースシステム」と呼ばれているその殆どが「リレーショナルデータベースシステム」です。みなさんが馴染みのあるであろう MySQL や PostgreSQL，あるいは Oracle といったデータベースシステムは，すべてリレーショナルデータベースシステムですね。リレーショナルデータベースシステムの歴史を辿ると，その起源はある一篇の論文に遡ることができます。

その論文こそが，Edgar F. Codd により 1970 年に発表された “A Relational Model of Data for Large Shared Data Banks” です。この論文は，リレーショナルデータベースの最も重要な基礎となる**リレーショナルモデル**を提唱したもので，いわばデータベースシステム分野における金字塔です。古典力学を Newton が拓き，相対性理論を Einstein が拓いたとするならば，データベースの一大分野であるリレーショナルデータベースを拓いたのは Edgar F. Codd その人といって間違いないでしょう。

データモデルとは，どのような形式でデータを格納して，どのようにデータにアクセスするかということを取り決める枠組みです。データを蓄積・管理することを目的とするデータベースシステムにおいては，データモデルはその心臓部とも言える存在なのです。

そして，Codd によるリレーショナルモデル提唱から数年の後に，UNIX で動作する世界初のリレーショナルデータベースシステムの開発プロジェクトが立ち上がります。Michael Stonebraker 率いる **INGRES プロジェクト**です。Codd により確立されたリレーショナルデータベースシステムの基礎理論を，実際に動くソフトウェアとして実現し，そしてそれを世に広めたのが INGRES なのです。

本稿では，Codd によるリレーショナルモデルの提唱から，INGRES プロジェクトの黎明期の記録を辿り，現代の社会を支えるデータベースシステムがいかにして創り上げられ

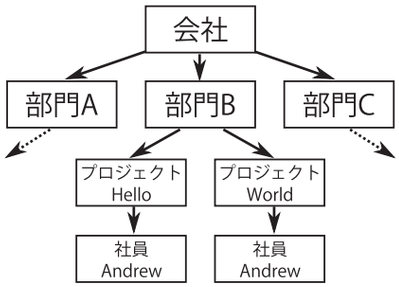

図 1.1　階層型モデル

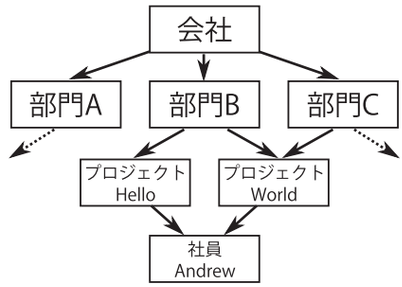

図 1.2　ネットワーク型モデル

たのか，その歴史を紐解いてみようと思います。

§1　リレーショナルモデルの登場

リレーショナルモデルが登場する以前，つまり 1960 年代までのデータベースシステムは殆どが階層型モデルやネットワーク型モデルというデータモデルに基づいて構築されていました。

階層型モデルというのは，木構造を用いてデータを組織化したデータモデルです。たとえば会社組織を階層型モデルで表すとしましょう。この会社では，会社の中に複数の部門があります。各部門の中では複数のプロジェクトが運営されています。そして各プロジェクトに従業員が属しています。そうすると，図 1.1 のような階層構造でデータを組織化することができます。

ここで，例えば 2 つのプロジェクト Hello project と World project に属する 1 人の社員 Andrew がいたとしたらどうでしょう。階層型モデルにおいて，1 つのデータの実体 (社員やプロジェクト) が複数の親データに属することができません。つまり，1 人の社員が複数のプロジェクトに属することを直接表現することができないのです。無理やりこれを表現しようとすると，プロジェクトごとに Andrew のデータを持つことになり，データの重複が生じてしまいます。また，部門 B と部門 C が共同でプロジェクト World を運営していることを表現しようと思うと，プロジェクトのデータに加えてその下にぶら下がっている社員のデータも併せてコピーしなければなりません。階層型モデルは非常にシンプルで性能が出しやすいというメリットもあるのですが，このようにデータ同士の関係を柔軟に記述することができないというデメリットもあるのです。

このような階層型モデルの欠点を克服したのがネットワーク型モデルです (図 1.2)。ネットワーク型モデルでは，木構造ではなくグラフ構造によりデータの組織化を行います。グラフ構造であれば Andrew が複数プロジェクトに属することを自然に表現できる

ために，前述のようなデータ重複の問題は発生しません。

1960 年代には，IBM によって IMS(Information Management System) という階層型モデルにもとづくデータベースシステムが開発されており，企業におけるデータ管理は IMS の独壇場ともいえる状況でした。1960 年代の終盤には，ネットワーク型モデルの標準化団体として CODASYL が立ち上がり，CODASYL データモデルが標準規格として確立されます。

そんな中，IBM 社内でこの動きに疑問を持つ一人の研究者がいました。この人こそ，後にリレーショナルモデルの提唱者となる Edgar F. Codd その人でした。

階層型モデルやネットワーク型モデルに基づくデータベースは，データのアクセスに一つ大きな問題を抱えていたのです。これらのデータベースにおいてデータの問い合わせを行う際には，データの構造がどのようになっているかを把握し，そしてどのような手順でデータを取得するかをプログラマ (データベースシステムのユーザ) が知っている必要がありました。例えば図 1.2 のデータベースにおいて社員 Andrew のデータを取得するためには，「会社のデータを読みとり，部門 B のデータを読み取り，プロジェクト Hello のデータを読み取り，社員 Andrew のデータを読み取る」という手順を指定してあげなければなりません。このように，欲しいデータにアクセスするために，プログラマがデータの案内 (navigation) をしなければならないという意味で，階層型モデルやネットワーク型モデルは**ナビゲーショナルモデル**，またそれにもとづくデータベースシステムは**ナビゲーショナルデータベースシステム**と呼ばれます。

もしもナビゲーショナルデータベースシステムにおいて，図 1.2 の会社で組織改変があり，各部門の下には課が設置され，その下にプロジェクトが属するという構造にデータベースが修正されたとしたらどうなるでしょう。これまで社員のデータにアクセスするアプリケーションは会社→部門→プロジェクト→社員とたどっていましたが，会社→部門→課→プロジェクト→社員とたどるように修正しなければなりません。このようにナビゲーショナルデータベースシステムでは，その上位構造に変化があった時にはデータにアクセスする方法を再構成しなければなりませんでした。

当時の IBM 社内では，データベースの仕様変更が生じるたびに，プログラマ達がデータアクセスのコード書き換えに追われていました。

この状況を打破するべく，Edgar F. Codd は 1970 年[*1] にリレーショナルモデルを提唱します。

リレーショナルモデルでは，データの論理的な構造と物理的な構造を独立させることができるように考慮されています。そのため，データ構造を変更するたびにアプリケーショ

[*1] 正確にはリレーショナルモデルの論文は 1969 年に IBM の社内技術報に掲載され，翌 1970 年に米国コンピュータ学会の論文誌に掲載されます。

ンを書き直す必要がありません。またユーザは「どんなデータが欲しいか」を記述するだけで，具体的なアクセス方法はデータベース側が判断してデータを取得することができます。つまり，ナビゲーショナルデータベースシステムでは “how” を与えなければデータのアクセスが行えなかったのですが，リレーショナルデータベースは “what” を与えるだけでデータのアクセスが可能となります。

また，リレーショナルモデルは集合論にもとづく数学的基礎の上に成り立っており，後のデータベースシステム研究のための大きな理論的基盤となりました。これまでプログラマによる職人芸の上に成り立っていたデータベースシステムは，リレーショナルモデルの登場によって科学の領域へと押し上げられたのです。

§2 INGRES プロジェクト始動

1970 年にリレーショナルモデルが提唱されてから，しばらく大きな動きはありませんでした。Codd がいた IBM においても，発表当初はあまり注目を集めることはなかったようです。

その一方で，1973 年が終わろうとする頃，その論文が一人の若き研究者の目に留まります。その人こそ，INGRES プロジェクト，そして後のデータベース業界全体を率いる存在となる Michael Stonebraker です。Stonebraker は 1971 年にミシガン大学で博士号を取得し，UC Berkeley で研究職についたばかりでした。テニュアトラック[*2] を走り始めたばかりの Stonebraker にとって，リレーショナルモデルは恰好の研究対象だったのでしょう[*3]。リレーショナルモデルに出会った Stonebraker は，同僚の Eugene Wong と共にリレーショナルデータベースシステムの実装に乗り出します。こうして INGRES プロジェクトは始まりました。

INGRES という名前は，**IN**teractive **G**raphic and **RE**trieval **S**ystem の頭文字をとったもので，本来は UC Berkeley の経済学の研究チームのために，地理情報をグラフィカルに表示するためのシステム研究として予算を獲得していたプロジェクトでした。そこは本音と建前で，まずはデータを効率的に取得するためのデータベースシステムが必要だ，ということで Stonebraker 達はデータベースシステムの開発にのめりこんでゆきます。

[*2] テニュアとは大学における終身雇用資格のことです。若手研究者は任期の限られた研究職につき，テニュア獲得を目指して研究業績を積み上げてゆく，というのは米国における一般的な研究者のキャリアパスです。このテニュア獲得のための若手研究者が通る道をテニュアトラックと呼びます。テニュアトラックでいかに論文を “量産” できる研究ネタを選ぶかというのは，若手研究者にとっては人生に関わる重要な決断なのです。

[*3] ちなみに Stonebraker の博士論文は “The Reduction of Large Scale Markov Models for Random Chains” というタイトルで，もともとは数学寄りの研究をしていたようです。

こうして走り始めた INGRES プロジェクトですが，データベースシステムを開発するためのコンピュータはありませんし，そもそも Stonebraker や Wong は大きなプログラム開発の経験すらありませんでした。リレーショナルデータベースシステムの先駆けである INGRES は，まさにゼロからのスタートだったのです。

何はなくともデータベースシステムの開発環境がなければ話になりません。INGRES プロジェクトが始まって Stonebraker が最初に取り組んだのは，コンピュータの調達と開発環境作りでした。当時といえば，データベースシステムを構築するためには高価なメインフレームを導入して，COBOL でプログラムを書くという時代でした。リレーショナルデータベースシステムのように，まだ世の中に存在すらしていないソフトウェアのために，研究職になりたての Stonebraker がメインフレームを手に入れることができるわけもありません。

そんな Stonebraker の目に留まったのが，当時新たな潮流として注目されつつあったミニコンピュータでした。ミニコンピュータとは，いわゆる現在の PC の先駆けとなる安価なコンピュータシステムのことです。1960 年代に生まれ，そして 1970 年代にはそのビジネス活用の動きが高まり，小型ビジネスコンピュータ市場が生まれました。ミニコンピュータ市場の勃興に呼応するように，ベル研究所では Ken Thompson 達によってミニコンピュータ上で動作する汎用オペレーティングシステム UNIX の開発プロジェクトが始まりました。UNIX はもともとアセンブリで実装されていましたが，Dennis Ritchie 達によって UNIX 開発のために C 言語が開発され，1973 年には C 言語に移植されました。そう，まさに INGRES プロジェクトが始まったその年です。

ミニコンピュータ，そして UNIX に出会った Stonebraker は，これを開発環境として使うことを決意します。開発マシンとして選ばれたのが，DEC(Digital Equipment Corporation) の PDP-11/70 でした。ちなみに，UNIX が最初にベル研究所以外の場所でインストールされたのがこの INGRES プロジェクトであり，インストール時には Ken Thompson と Dennis Ritchie が 5MB のディスクを抱えてやってきたそうです。

こうしてリレーショナルデータベースシステムの開発環境も整い，INGRES プロジェクトは動き始めました。

§3　INGRES の進化

INGRES プロジェクトでは，リレーショナルデータベースシステムの実装における重要な技術が多数生み出されてゆきました。当時の INGRES がどのように設計されていたのか，1976 年に発表された論文 “The design and implementation of INGRES” をもとに，その技術的な側面を概観してゆきます。

以降の説明で使われる用語
♣ **リレーション** · · · リレーショナルモデルにおける，いわゆるテーブルのこと
♣ **タプル** · · · リレーショナルモデルにおける，いわゆるレコードのこと

§3.1 クエリ言語 QUEL

リレーショナルデータベースでは，それ以前のナビゲーショナルデータベースとは異なり，どんなデータが欲しいか “what” を記述するだけで，具体的にどのようなデータ構造やアルゴリズムが実装されているかを一切知る必要なくデータを取得することができる，というのが大きなウリの一つです。その “what” を記述するための言語がクエリ言語です。皆さんのよく知っている SQL もクエリ言語の一つです。

INGRES プロジェクトでは，クエリ言語として QUEL が開発されます。リレーショナルモデルに併せて Codd は DSL/APLHA というクエリ言語を 1971 年に発表しているのですが，DSL/ALPHA は数学的な表記[*4]に基づいておりやや一般的にわかりやすいものではありませんでした。QUEL は DSL/ALPHA をベースとして，数学的な表記のない “やさしい” クエリ言語として設計されました。

QUEL がどんな言語なのか，簡単なサンプルコード[*5]をみてみましょう。

```
1 RANGE OF C IS CITY
2 RETRIEVE INTO W(C.CNAME,
3                 DENSITY = C.POPULATION / C.AREA)
4       WHERE C.STATE = 'California'
5         AND C.POPULATION > 50000
```

このサンプルコードでは，カリフォルニア州にある人口 5 万人以上の市について，市の名前と人口密度の一覧を取得するという処理を行っています。一行目の RANGE 文では，処理の対象として CITY リレーションを選択して，CITY リレーションのタプルを表す変数名 C を宣言しています。二行目の RETRIEVE 文は，SQL でいうところの SELECT 文に相当します。処理の結果は，一時テーブルとして作成される W に保存されます。細かい表記の違いを別にすると，現在使われている SQL とそれほど大きな違いがないことがわかると思います。

[*4] DSL/ALPHA は関係演算 (relational calculus) をベースとして，量化子 $\exists, \forall$ や，論理演算の記号 $\neg, \wedge, \vee$ などがほぼそのまま用いられていました
[*5] 1975 年の INGRES の論文より引用

§3.2 クエリ処理

1975 年に発表された時点で，INGRES は複数のリレーションの結合を含むクエリをサポートしています。

データベースシステムをある程度使い込んだことがある人ならば，「ネステッドループ結合 (ジョイン)」や「ハッシュ結合 (ジョイン)」という言葉を聞いたことがあると思います。データベースの性能チューニングをやったことがある人は，クエリの実行プランを眺めてハッシュジョインが走るようにパラメタを調整して…なんていう記憶もあるのではないでしょうか。

当時の INGRES においては，今で言うネステッドループ結合が実装されていました。INGRES におけるネステッドループ結合は，実は現在のデータベースシステムにおけるネステッドループ結合とアルゴリズムとしてはほとんど同じです。逆に言うと，リレーション結合のコアな処理というのは未だに 30 年以上前の技術から大きく変化していないということでもあります。色々と複雑で高度なことをやっているように見えるデータベースシステムも，細かい技術要素ごとに分解して眺めてみると，実は古の技術がそのまま生き残っていることもあるのです[*6]。

では，INGRES の具体的なクエリ処理エンジンの構造を概観してみましょう。INGRES のクエリ処理エンジンは “One Variable Query Processor (OVQP)” と呼ばれています。日本語にすると「1 変数クエリ処理エンジン」ということになります。前節のクエリの例で **RANGE** 文でリレーションに対して変数を宣言していましたが，「1 変数クエリ処理エンジン」の「変数」とはこのリレーションを表す変数のことを指します。つまり，「1 変数クエリ処理エンジン」とは同時にひとつのリレーションのみを処理することができるクエリ処理エンジンということです。

それじゃあ複数のリレーションの結合ができないじゃないか，と思われるかもしれません。複数のリレーションの結合とは，つまり複数の変数が組み合わさったクエリになるからです。このようなクエリを処理するために，INGRES では事前に 1 つの複雑なクエリを，複数の 1 変数クエリに分解して，OVQP に放り込んで処理してゆきます。例を追いながらその処理を追いかけてみましょう。

[*6] もちろんハッシュ結合やソートマージ結合など別の結合方式も後から生み出されていますし，並列制御なんかは結構複雑で高度なことをやっています

```
1 RANGE OF E, M IS EMPLOYEE
2 RANGE OF D IS DEPT
3 RETRIEVE (E.NAME)
4     WHERE E.SALARY > M.SALARY AND
5           E.MANAGER = M.NAME AND
6           E.DEPT = D.DEPT AND
7           D.FLOOR# = 1 AND
8           E.AGE > 40
```

このクエリでは，EMPLOYEE(従業員) リレーションと DEPT(部門) リレーションが定義されているデータベースから，1 階で働いていて (*D.FLOOR#* = 1)，年齢が 40 歳より上で (*E.AGE* > 40)，マネージャーより給料が高い (*E.SALARY* > *M.SALARY* **AND** *E.MANAGER* = *M.NAME*) 従業員のデータを抜き出す，という問い合わせを行っています。

■Step 1.　クエリが 1 変数かどうかを判定→ 3 変数なので分解する

■Step 2.　次の 2 つのクエリを実行

```
1 RANGE OF D IS DEPT
2 RETRIEVE INTO T1(D.DEPT)                                  (1)
3     WHERE D.FLOOR# = 1
```

```
1 RANGE OF E IS EMPLOYEE
2 RETRIEVE INTO T2(E.NAME, E.SALARY, E.MANAGER, E.DEPT)     (2)
3     WHERE E.AGE > 40
```

こうすると，一時リレーション *T1*，*T2* を使って最初のクエリは次のようになります。

```
1 RANGE OF D IS T1
2 RANGE OF E IS T2
3 RANGE OF M IS EMPLOYEE
4 RETRIEVE (E.NAME)
5     WHERE E.SALARY > M.SALARY AND
6           E.MANAGER = M.NAME AND
7           E.DEPT = D.DEPT
```

■Step 3. データ数の少ない一時テーブル，ここでは *T1* を変数置換用のリレーションとして選択

■Step 4.

4.1

T1 の各タプルについて，元のクエリ中の変数 *D* をタプルの値で置き換える

```
1 RANGE OF E IS T2
2 RANGE OF M IS EMPLOYEE
3 RETRIEVE (E.NAME)
4     WHERE E.SALARY > M.SALARY AND
5           E.MANAGER = M.NAME AND
6           E.DEPT = [value].DEPT
```

これで元は 3 変数だったクエリが 2 変数になりました。

4.2

変数を値で置き換えたクエリが 1 変数でない場合には，Step 1. にもどって同じ手順を繰り返し，変数を減らしてゆく

4.3

変数を値で置き換えて実行した各クエリの結果を集約

このようにして，INGRES では複数リレーションの結合を含む複雑なクエリを処理していました。現在のデータベースシステムにおいても，本質的な処理の流れは同じように実装されています。

§3.3 アクセスメソッドとデータ構造

アクセスメソッドとは，その名の通りデータベースに保存されているデータにアクセスするための方法です。例えば，とある会社の従業員のデータが EMP リレーションに 1 万人分保存されていたとします。この中からある従業員 1 人分のタプルを取り出すには，どうすればよいでしょうか。もっともナイーブには，EMP リレーションを頭から全部スキャンして，目的の従業員のタプルを探すという方法が考えられます。これも立派なアクセスメソッドの一つですが，1 人分のデータを探すにはちょっと効率が悪すぎます。従業員番号でタプルを並べておいて，二分探索するというやり方も簡単に思いつく方法です。これもアクセスメソッドの一つといえます。

アクセスメソッドには様々な実装方法があり，それぞれが得意とするデータアクセス，苦手とするデータアクセスのパターンがあります。データベースシステムのように総合的なデータの管理を行うためのシステムでは，複数のアクセスメソッドを利用できる必要があります。

アクセスメソッドインターフェース

INGRES には，複数のアクセスメソッドを利用するための共通インターフェースが定められています。INGRES で標準的に提供されるアクセスメソッドはすべてこのインターフェースを介して提供されており，またこのインターフェースに従って実装することで独自のアクセスメソッドを追加することができるように設計されています。

INGRES で定められているアクセスメソッドインターフェースは次の通りです:

1. **OPENR**(*descriptor*, *mode*, *relation_name*)
2. **GET**(*descriptor*, *tid*, *limit_tid*, *tuple*, *next_flag*)
3. **FIND**(*descriptor*, *key*, *tid*, *key_type*)
4. **PARAMD**(*descriptor*, *access_characteristics_structure*)
5. **PARAMI**(*index_descriptor*, *access_characteristics_structure*)
6. **INSERT**(*descriptor*, *tuple*)
7. **REPLACE**(*descriptor*, *tid*, *new_tuple*)
8. **DELETE**(*descriptor*, *tid*)
9. **CLOSER**(*descriptor*)

OPENR と **CLOSER** はプログラマにとってはファイル操作等でお馴染みのインターフェースでしょう。**OPENR** で指定したリレーションを開き，リレーションのディスクリプタを取得します。以降，このリレーションに対するアクセスはこのディスクリプタを

以って行われます。一連のアクセスが完了した後は，**CLOSER** によってリレーションを閉じます。

GET は，タプル ID(tid) をキーとしてタプルのデータを取得するための関数です。*tid* でタプル ID の下限を，*limit_tid* でタプル ID の上限を指定することができ，**GET** が呼び出されるとこの条件にマッチする最初のタプルを見つけてきて，そのデータを *tuple* に書き込みます。また，*next_flag* に TRUE を指定して **GET** を呼び出した場合には，次にマッチするタプルの ID を *tid* に書きこむため，連続してタプルを読みだしてゆくことができます。

FIND は，*key* で指定した検索キーにマッチするタプル ID を取得するための関数です。*key_type* で完全一致検索，範囲検索などの検索モードを指定することができます。

PARAMD，**PARAMI** は **FIND** において利用可能な検索キーの種類や，検索モードを調べるための関数です。それぞれ，データのリレーションと，索引のリレーションに利用されます。

INSERT はタプルを挿入するための関数，**REPLACE** は既に存在するタプルを更新するための関数，**DELETE** はタプルを削除するための関数です。

データ構造

INGRES では，**ヒープ**，**ハッシュ**，**ISAM**，**圧縮ハッシュ**，**圧縮 ISAM** の 5 種類のアクセスメソッドが提供されています。

ヒープは単純にタプルを並べてゆくだけの最も単純な構造で，タプルを探すときは毎回線形探索を行います。非常にタプル数の少ないリレーションや，クエリの実行結果を格納する一時的なリレーションに適しています。

ハッシュと **ISAM** は，検索キーを用いて少量のタプルに対するアクセスを高速に行うことを可能とする構造です。**ハッシュ**はハッシュテーブルを使って，**ISAM** は探索木を使って，検索キーとタプル ID の関連付けを行うことで高速なアクセスを実現しています。INGRES では，タプル ID は (ページ番号, ページ内ライン番号) という値のペアによって表現されているので，タプル ID さえわかれば目的のデータの在り処がわかります。

ページの構造は図 1.3 のようになっています。ページの先頭にはヘッダ情報があり，またページの末尾にはラインテーブルと呼ばれるデータが格納されています。ラインテーブルの各要素は，タプルのデータがページ内のどの位置にあるかのオフセット情報を保持しています。タプル ID に含まれているページ内ライン番号は，このラインテーブル内のインデックス番号となっているので，タプル ID さえわかればタプルのデータがどこにあるのかがわかるようになっています。この構造には，タプルの物理的な位置が変化しても，ラインテーブルを更新するだけで，タプル ID は変化することがないという利点があ

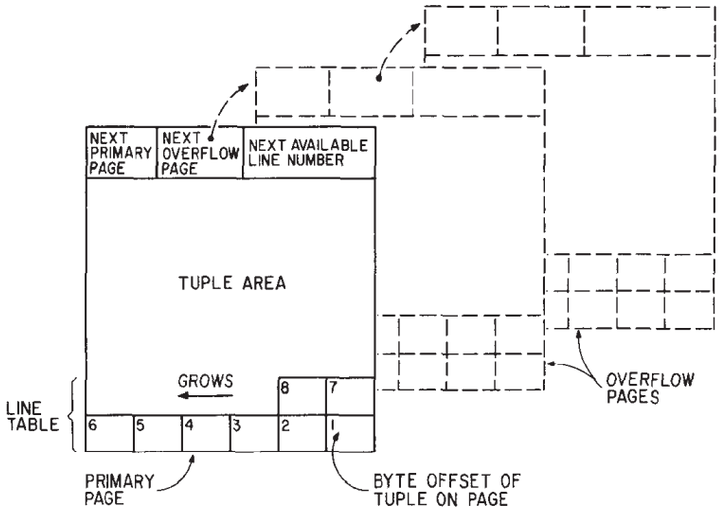

図 1.3　INGRES のページ構造

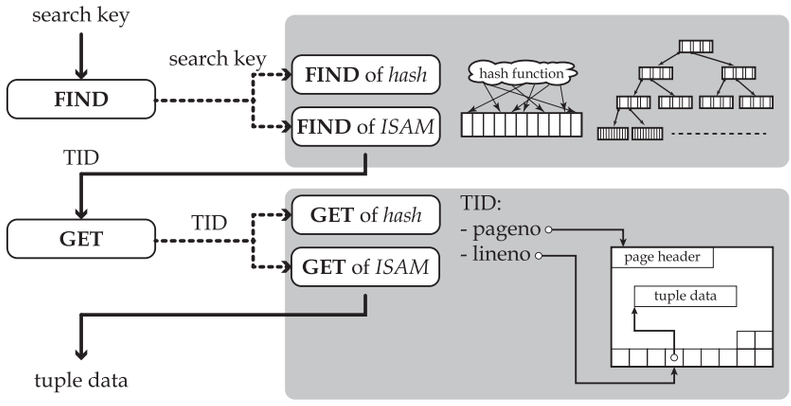

図 1.4　検索キーを用いたアクセスメソッドによるタプルデータの取得

ります。

アクセスメソッドまとめ

検索キーが与えられてから，タプルのデータが得られるまでの一連の流れを図 1.4 にまとめました。まずは検索キーが与えられると，**FIND** を呼び出してタプル ID を取得します。**FIND** の中では，ハッシュテーブルや木構造を用いて効率的にタプル ID の検索が行われます。そして，得られたタプル ID を引数として **GET** を呼び出すことで，ページを

読み出し，タプルのデータが取得されます。

この基本的なデータアクセスの枠組みは，INGRES の頃に確立され，現在でも殆ど変わること無く引き継がれています。

ただ 1980 年頃まで，INGRES には B^+-tree を用いたアクセスメソッドは実装されていませんでした。ISAM と B^+-tree はよく似ているのですが，根本的な違いとして ISAM は静的なアクセスメソッドであるのに対し，B^+-tree は動的なアクセスメソッドであるということです。どちらも多分木構造を索引として用いていますが，ISAM ではデータの挿入や削除のたびに木構造を変更することはなく，適当なタイミングでまとめて木構造を作り直します。そのため，頻繁にデータの挿入，削除が行われる場合にはどんどん構造劣化が進み，アクセス性能が落ちてしまいますが，参照が非常に多い場合には良い性能を発揮します。一方，B^+-tree はデータの挿入，削除のたびに木構造を動的にバランスさせるために，ISAM で起きるような構造劣化は起きません。しかし，1 回の処理あたりのオーバーヘッドは ISAM よりも大きくなってしまいます。

INGRES の開発において，何度も B^+-tree の導入が検討され，その度に採用が見送られてきたようです。しかしながら，1980 年の INGRES 開発を振り返る論文において，Stonebraker は B^+-tree の採用を見送ってきたのは “major mistakes” の一つであり，次にアクセスメソッドを再検討する際には B^+-tree も実装するであろうと述べています。

また，INGRES のアクセスメソッドは UNIX のファイルシステムの上に構築されていたのですが，UNIX のファイルシステムによるオーバーヘッドが当初予測していたよりも深刻なものであり，独自のファイルシステムを実装するべきであった，とも同論文では述べられています。実際に，3 年後に発表された論文において Stonebraker 達は独自に実装したファイルシステムを使った場合と性能比較を行っており，最大で 70% 程度性能が向上することを確認しています。このように INGRES の開発過程で得られた知見は，UNIX の開発にフィードバックされ，その発展に大きく貢献しました。

§3.4 人材育成の場としての INGRES

INGRES プロジェクトは，初期のリレーショナルデータベース実装における技術を確立するという非常に重要な功績を成し遂げました。しかし，INGRES プロジェクトの功績はそれだけにとどまりません。後のデータベース業界を牽引する人材を輩出したという点においても，その価値が評価されています。

INGRES プロジェクトの最初のチーフプログラマを務めた Gerald Held は Tandem Computers において NonStop SQL の開発に携わります。また別の時期にチーフプログラマを務めた Robert Epstein は Sybase 社を設立して Sybase の開発を行います。また Peter Krepps は INGRES 商用化に際して立ち上げられた Relational Technology 社で

プログラマとして活躍しました。

§4 System R

INGRES プロジェクトからすこし遅れて，IBM でもリレーショナルデータベースシステム System R の研究開発が始まります。リレーショナルモデルの父である Codd が IBM で働いていたにも関わらず，歴史を振り返ると IBM はリレーショナルデータベースシステム開発において常に後塵を拝しています。

当時の IBM では IMS が圧倒的な商業的成功を収めていました。そのため，IMS の開発のために多くの投資を行っている IBM としては，よくわからない数学で記述されたリレーショナルモデルは注目に値しないものだと思われたのでしょう。そのため，リレーショナルモデルにもとづくデータベースシステムとして Sysmte R の研究開発が始まっても，スタッフとしては 10 人程度しか人員が配置されなかったようです。

IMS がベースとする階層型モデルはナビゲーショナルなので，プログラマがデータアクセスの方法を記述しなければなりませんでした。しかし見方を変えると，データベースのことを熟知しているプログラマにとっては，最善の方法でデータアクセスを行うことができるということでもあります。一方，リレーショナルモデルではプログラマが明示的にデータアクセスの方法を制御できません。究極的なパフォーマンスの観点からすると，ナビゲーショナルモデルが優位なのは確かです。実際に，究極の性能が求められるミッションクリティカルなシステムでは，未だにナビゲーショナルデータベースシステムが用いられているようです[*7]。

その後，リレーショナル陣営とナビゲーショナル陣営は激しい論争を繰り広げてゆきますが，次第に世の中の流れはリレーショナルモデルに傾いてゆきました。歴史を振り返ると，今やアセンブラを直接書くことが殆どないように，コンピュータの性能が良くなるにつれて技術のトレンドが変化するというのはままあることです。データベースシステムにおいても，最初は性能が低くて受け入れられないと思われていたリレーショナルモデルが，その利便性が故に主流となってゆきました。

§5 エピローグ

1970 年代，Codd によるリレーショナルモデルの提唱に始まり，INGRES や System R の研究によってリレーショナルデータベースシステムが実用に耐えうるものであること

[*7] JAL のシステムにも IMS が使われているとか

が証明されました。1981 年には，Codd はその功績を認められ計算機科学分野において最も権威あるチューリング賞を受賞します。

そして 1970 年代の終盤から 1980 年代は，商用リレーショナルデータベースシステムが次々と登場し，熾烈な戦いを繰り広げてゆきます。

1977 年，現在の Oracle の前身である Software Development Laboratories が Larry Ellison と Robert Miner によって設立され，1979 年には世界初の SQL ベースの商用リレーショナルデータベースシステム Oracle RDBMS をリリースします。

1979 年，Jack Shemer，Philip Neches，Walter Muir，Jerold Modes，William Worth，Carroll Reed によって Teradata が設立され，1984 年に世界初の並列データベースシステム DBC 1012[*8]をリリースします。

1980 年，Lawrence Rowe，Michael Stonebraker，Eugene Wong，Gary Morgenthaler によって INGRES 商用化のために Relational Technology (後の Ingres Corp.) が設立され，翌 1981 年に商用 INGRES をリリースします。一方，UC Berkeley においても INGRES プロジェクトは継続しており，こちらのほうは University INGRES と呼ばれるようになります。University INGRES のソースコードは現在 UC Berkeley のウェブサイトからダウンロード可能です[*9]。

時を同じくして，Roger Sippl と Laura King によって 1980 年に Relational Database Systems (後の Informix Software) が設立され，翌 1981 年に Informix がリリースされます。

IBM は 1981 年に SQL/DS を，さらに 1983 年には DB2 をリリースしてリレーショナルデータベース市場に乗り出します。エンタープライズ市場の覇者である IBM のこの動きは，旧来のナビゲーショナルデータベースシステムに対する新興勢力リレーショナルデータベースシステムの勝利を決定的なものにします。

1984 年，Mark Hoffman，Bob Epstein，Jane Doughty，Tom Haggin によって Sybase が設立され，1986 年にリレーショナルデータベースシステム Sybase をリリースします。後に Sybase は Microsoft と提携して Sybase SQL Server を開発しました。その設計は現在の Microsoft SQL Server にも受け継がれています。

同 1984 年，ミニコンピュータ市場を牽引する DEC が OpenVMS オペレーティングシステムにおける機能として DEC Rdb を発表します。

1986 年，無停止コンピュータシステムを主な事業として手がける Tandem Computers が NonStop SQL をリリースします。

そして 1990 年代には更にプレイヤーが増えつつもその統廃合が進み，オブジェクト指

[*8] ちなみに 1012 というのは $10^{12} = 1\text{TB}$ にちなんでつけられた名前です

[*9] `http://s2k-ftp.cs.berkeley.edu/ingres/`

向データベースシステムなど新興技術の登場，オープンソースのデータベースシステム隆盛の兆候など，データベースシステムは新たな時代へと突入してゆきます。

参考文献

- E. F. Codd, “A Relational Model of Data for Large Shared Data Banks”, *Communications of the ACM*, Volume 13, Number 6, 1970
- E. F. Codd, “A data base sublanguage founded on the relational calculus”, *Proceedings of the 1971 ACM SIGFIDET workshop on Data Description, Access and Control*, 1971
- C. J. Date, E. F. Codd, “The relational and network approaches: Comparison of the application programming interfaces”, *Proceedings of the 1974 ACM SIGFIDET workshop on Data Description, Access and Control*, 1975
- G. Held, M. Stonebraker, E. Wong, “INGRES–A relational data base management system”, *Proceedings of AFIPS 1975 NCC*, Volume 44, 1975
- M. Stonebraker, G. Held, E. Wong, P. Kreps, “The design and implementation of INGRES”, *ACM Transactions on Database Systems*, Volume 1, Issue 3, 1976
- G. Held, M. Stonebraker, “B-trees re-examined”, *Communications of the ACM*, Volume 21, Issue 2, 1978
- M. Stonebraker, “Retrospection on a database system”, *ACM Transactions on Database Systems*, Volume 5, Issue 2, 1980
- D. Chamberlin, M. Astrahan, M. Blasgen, J. Gray, W. F. King, B. Lindsay, R. Lorie, J. Mehl, T. Price, F. Putzolu, P. Selinger, M. Schkolnick, D. Slutz, I. Traiger, B. Wade, R. Yost, “A history and evaluation of System R”, *Communications of the ACM*, Volume 24, Issue 10, 1981
- M. Stonebraker, J. Woodfill, J. Ranstrom, M. Murphy, M. Meyer, E. Allman, “Performance enhancements to a relational database system”, *ACM Transactions on Database Systems*, Volume 8, Issue 2, 1983
- Relational Technology Inc., “INGRES The Distributed SQL Relational Database System Press Kit”, 1987
- National Research Council, “Funding a Revolution: Government Support for Computing Research”, Natl Academy Pr, 1999
- M. Stonebraker, J. M. Hellerstein, “What Goes Around Comes Around”, *Readings in Database Systems, Fourth Edition*, Morgan Kaufmann, 2005
- A. Ances, “RDBMS Workshop: Ingres and Sybase”, Computer History Museum, 2007

and so many online resources

2

PostgreSQL カンファレンス 2012 レポート

はやみず

2012 年 3 月 24 日に日本 PostgreSQL ユーザ会主催で開催された PostgreSQL カンファレンス 2012 に参加してきました。この記事は，PostgreSQL カンファレンス全体の概観についてレポートを記したものです。

このカンファレンスは技術的な話題を中心として，PostgreSQL に関する導入事例や技術的な話題を提供するためのカンファレンスです。 昨年度の参加者約 180 名に対して今年度は 275 名 (関係者含む) ということで，日本における PostgreSQL のユーザ層の広がりを感じさせます。 特にこの数年はビジネスとしてこのカンファレンスに参加する方も増えているそうで，業務用途としての PostgreSQL の利用が広まっていることを示しているのではないかと思われます。

Linux の急速な普及をきっかけとして，今やオープンソースソフトウェアを業務において利用することは全く珍しくなくなっています。 PostgreSQL もその例に漏れず，といったところでしょうか。本年度のカンファレンスのプログラムも，業務用途における PostgreSQL の利用ということが一つの大きな軸として捉えられているように見えます。例えば，午後のプログラムは 3 トラック構成となっていたのですが，そのうちの 1 トラックは「マイグレーショントラック」と題して他の DBMS から PostgreSQL へ移行することを主眼としたものであり，「商用 DB から PostgreSQL への移行」「PostgreSQL がより使いやすく進化した Postgres Plus Advanced Server の実力とは」などは明らかに商用 DB を意識しています。また，技術トラックや Lightning Talk において高可用・高性能な PostgreSQL クラスタに関する発表がありました。業務用 DBMS としてはクラスタソリューションがほしい。MySQL には MySQL Cluster が，Oracle には Oracle RAC が，じゃあ PostgreSQL はどうなの？というポイントに答えを出すのが，今回のカンファレンスの一つの見所だったのではないでしょうか。

もちろん，ビジネス的な側面だけではなく，技術の深い話もできるのがこのカンファレンスの魅力だと思います。午前中の基調講演では，PostgreSQL の大型移行案件の事例紹介であるフランス社会保障機構の講演に加えて，PostgreSQL のコア開発者でありコミュニティの中でも若手のエースと目されている EnterpriseDB 社の Robert Haas 氏による PostgreSQL 9.2 の新機能に関する講演がありました。こちらの講演では，メニーコア環

境におけるスケーラビリティ向上の取り組みや，Index-only Scan の導入，消費電力削減のための実装努力，レプリケーションの新機能，その他細々とした新機能などが紹介されました。加えて，バージョン 9.2 より後にどんな技術的な課題に取り組んでいくべきか，といった方向性についても触れられていました。質疑応答も非常に活発に行われていたのが印象的です。一つ残念だったのは，こちらの講演は発表・質疑応答ともに逐次翻訳が行われていたため，実際に話せる時間が半分になってしまっていたことです。英語のみで進行してくれたほうが個人的には中身がもっと詰まって面白かったのではないかと思いますが，難しいところですね。

もう 1 件技術的な講演として面白かったのは藤井雅雄氏，松尾隆利氏による「PostgreSQL 9.1 同期レプリケーションと Pacemaker による高可用クラスタ化の紹介」です。こちらの発表では，まず藤井氏から PostgreSQL におけるレプリケーションの詳細について紹介されていました。非同期レプリケーション，同期レプリケーションの動作や，何ができるのか，何ができないのかについて丁寧にまとめられており，レプリケーション機能の全体像を理解できる発表でした。またそれに加えて，9.2 で導入される予定のレプリケーション関連の機能紹介や，周辺ツールに関する紹介もありました。 その後，松尾氏により同期レプリケーション機能と Pacemaker を組み合わせることによる，障害発生時にもフェイルオーバ可能なクラスタ構成法についての発表が行われました。Pacemaker については名前くらいしか知らなかったのですが，障害発生からフェールオーバまでの流れがわかりやすく図解されており，Pacemaker を知るという観点からも有用な発表であったように思います。

本会議のほうについてはこんなところで，懇親会にも参加したのでの話をひとつ。PostgreSQL のお役立ち情報源として参照している人も多いかと思われる Let's Postgres という読み物サイトがあるのですが，こちらの運営をしている方とお話をさせて頂きました。 この手の読み物サイトでは，最も需要の高い入門系記事や，企業の提灯記事が並ぶというのがよくあるパターンなのですが，Let's Postgres には PostgreSQL の内部構造についてやたら詳しく書かれた記事がたくさんあり，これは一体どういったことだろうと思っていたのでした。 そのことを聞いてみたところ，内部構造の多くは記事 PostgreSQL 本体にコミットしている開発者の方々が好きで寄稿しているということだそうです。PostgreSQL 自体が内部構造まで含めた子細なドキュメントが充実していますが，新機能の実装詳解がこれほどまで精力的に行われているソフトウェアはなかなかないのではないかとおもいます。これらの記事は目的別ガイド：内部解析編としてまとめられています。内部の詳解に加えて，開発プロセスへの参加方法についても紹介されており，非常に充実した内容となっています。PostgreSQL の中身に興味がある人は必見ですね。

以上，簡単ではありますが参加レポートでした。

3

とある世界で一番高速な Brainfuck インタプリタ

hogelog

§1 まえがき

世の中にはたくさんのプログラミング言語が存在します。Dennis Ritchie は Unix を記述するために C 言語を作り[*1]，Tecgraf チームはブラジルの貿易障壁の影響下で Lua を[*2]，そして Urban Müller はチューリング完全な最小限の言語として Brainfuck を作成しました[*3]。

世の中にはたくさんのプログラミング言語処理系が存在します。例えば Ruby 言語にはまつもとゆきひろによるオリジナル実装を元にした MRI[*4]，JVM 上で動作する JRuby[*5]，JIT コンパイラや世代別 GC などが組み込まれた Rubinius[*6]，組み込みスクリプトエンジン向けとして作られた mruby[*7] など様々な処理系があります。例えば大規模分散処理のため Hadoop フレームワークを利用したければ JRuby を使うこともあるでしょう。エディタアプリケーションにスクリプト機能を組み込むためには mruby を使うかもしれません。言語処理系にとって重要な機能は使う局面によって変わってきますが，処理系が高速であればだいたいの人は嬉しいでしょう？

Brainfuck という言語は `+ - > < , . [ ]` という 8 個の命令しか存在しない，処理系の実装が容易な言語です[*8]。以下では Brainfuck インタプリタの高速化を通じて**インタプリタの高速化**技術について親しんでいきましょう。とりあえずはこの記事のタイトル通りに**世界で一番高速な Brainfuck 処理系**を作ってみましょう。

なお以下で示すサンプルコードは筆者の github リポジトリ[*9] から適宜参照できます。

[*1] The Development of the C Language* http://cm.bell-labs.com/cm/cs/who/dmr/chist.html
[*2] The Evolution of Lua http://www.lua.org/doc/hopl.pdf
[*3] http://www.muppetlabs.com/~breadbox/bf/
[*4] http://www.ruby-lang.org/ja/
[*5] http://jruby.org/
[*6] http://rubini.us/
[*7] https://github.com/mruby/mruby/
[*8] http://www.kmonos.net/alang/etc/brainfuck.php とかわかりやすい解説だと思います
[*9] https://github.com/hogelog/sample-code/tree/master/dbtimes-vol01

§2 普通の処理系

https://github.com/hogelog/sample-code/blob/master/dbtimes-vol01/bf.cpp

さて，まずはなんの高速化も施されていない Brainfuck 処理系を実装します。(図 3.1) たったこれだけの記述で一応ちゃんとした言語処理系になるんだから楽で良いですね。

```
#include <stdio.h>
#include <string>
int main() {
  std::string code;
  for (int ch;(ch=getchar())!=EOF;)
    code.push_back(ch);
  static int membuf[30000];
  int *mem = membuf;
  int depth = 0;
  for (size_t pc=0;pc<code.size();++pc) {
    switch(code[pc]) {
      case '+': ++*mem; break;
      case '-': --*mem; break;
      case '>': ++mem; break;
      case '<': --mem; break;
      case ',': *mem = getchar(); break;
      case '.': putchar(*mem); break;
      case '[':
        if (*mem == 0) {
          depth = 1;
          while (depth != 0) {
            switch(code[++pc]) {
              case '[': ++depth; break;
              case ']': --depth; break;
            }}}
        break;
      case ']':
        depth = -1;
        while (depth != 0) {
          switch(code[--pc]) {
            case '[': ++depth; break;
            case ']': --depth; break;
          }}
        --pc;
        break;
    }}
  return 0;
}
```

図 3.1 普通の Brainfuck インタプリタ

§3 インタプリタ高速化

上述したインタプリタは簡潔ですしちゃんと動くので良いのですがあまり高速ではないので，いくつかの高速化を施して世界最速インタプリタにしてみましょう。

§3.1 VM 化

https://github.com/hogelog/sample-code/blob/master/dbtimes-vol01/bf-vm.cpp

プログラムの文字列を逐次解釈するインタプリタを高速化する手法として，プログラムを VM[*10]命令列へ変換する部分と VM 命令列のインタプリタ部分に分ける方法がよく知られています。Ruby インタプリタの MRI はバージョン 1.9 から独自 VM を内蔵するようになり，大幅な高速化に成功しました。ここでは Brainfuck インタプリタの VM 化を試してみましょう。

図 3.1 のプログラムは [ ] 命令を処理する部分に while 文があるのが気になります。

[*10] Virtual Machine の略

[と対応する] は一度調べればわかるのに，毎回 1 文字ずつ探すのは効率がよくありません。[ ] 命令を対応する VM 命令の位置を保持する VM 命令へと変換することで，[ ] 命令の処理部分から while 文が消えます。(図 3.3)

またプログラムを実行する for ループで毎回 pc<code.size() とチェックするのも無駄です。処理する VM 命令列の最後に「処理を終了する」命令を一つ追加することで，終了判定も switch 文の中に混ぜることができました。(図 3.4)

```
struct Instruction {
    char op;
    int jmp;
};
```

図 3.2　VM 命令構造

```
case '[':
    if (*mem == 0) {
        pc = code.jmp;
    }
    break;
case ']':
    pc = code.jmp;
    break;
```

図 3.3　[] 命令処理部分

```
for (size_t pc=0;;++pc) {
    Instruction code = codes[pc];
    switch(code.op) {
        ...
        case '\0':
            return;
```

図 3.4　終了処理

direct threaded code

https://github.com/hogelog/sample-code/blob/master/dbtimes-vol01/bf-vm2.cpp

VM の高速化手法の一つに direct threaded code と呼ばれる手法があります。詳細は Ruby 1.9 の VM 化を主導した笹田耕一さんによる記事 YARV Maniacs [*11] に任せますが，この手法により命令処理部分に存在した大きな switch 文による分岐コストを大幅に軽減できます。

§3.2　JIT コンパイル

https://github.com/hogelog/sample-code/blob/master/dbtimes-vol01/bf-jit.cpp

前の章では VM 化としてプログラムを VM 命令に変換する部分と VM 命令を処理する部分に分けることによる最適化について説明しました。しかしどうせ変換するならプログラムを実際の機械語列に変換して，その機械語列をそのまま CPU に実行してもらえば良いのです。この手法は JIT コンパイルと呼ばれています。最近では各種ブラウザベンダが「うちのブラウザの JavaScript エンジンは JIT コンパイラを内蔵してるから速いんだ！」と主張しあってたりすることから耳にする機会も多いでしょう。

Xbyak[*12]，asmjit[*13] などの JIT アセンブラを用いると，各命令処理をアセンブラ相当の記述をするだけで JIT コンパイラができますし，慣れてしまえば簡単ですね？

[*11] http://jp.rubyist.net/magazine/?0008-YarvManiacs
[*12] http://homepage1.nifty.com/herumi/soft/xbyak.html
[*13] http://code.google.com/p/asmjit/

§3.3 最適化

```
+++++++++++++[->++>>>+++++>++>+<<<<<<]>>>>>++++++>--->>>>>>>>>>+++++++++++++++[[
>>>>>>>>>]+[<<<<<<<<<]>>>>>>>>>-]+[>>>>>>>>[-]>]<<<<<<<<<[<<<<<<<<<]>>>>>>>>[-]+
<<<<<<<+++++[-[->>>>>>>>>+<<<<<<<<<]>>>>>>>>>]>>>>>>>+>>>>>>>>>>>>>>>>>>>>>>>>>>
>+<<<<<<<<<<<<<<<<<[<<<<<<<<<]>>>[-]+[>>>>>[>>>>>>>[-]>>]<<<<<<<<<[<<<<<<<<<]>>
>>>>>[-]+<<<<<<++++[-[->>>>>>>>>+<<<<<<<<<]>>>>>>>>>]>>>>>>+<<<<<<+++++++[-[->>>
>>>>>>+<<<<<<<<<]>>>>>>>>>]>>>>>>+<<<<<<<<<<<<<<<<[<<<<<<<<<]>>>[[-]>>>>>>[>>>>>
>>[-<<<<<<+>>>>>>]<<<<<<[->>>>>>+<<+<<<+<]>>>>>>>>]<<<<<<<<<[<<<<<<<<<]>>>>>>>>>
[>>>>>>>>[-<<<<<<<+>>>>>>>]<<<<<<<[->>>>>>>+<<+<<<+<<]>>>>>>>>]<<<<<<<<<[<<<<<<<
<<]>>>>>>>[-<<<<<<<+>>>>>>>]<<<<<<<[->>>>>>>+<<+<<<<<]>>>>>>>>>+++++++++++++++[[
>>>>>>>>>]+>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]>[-]<<<<<<<<<[<<<<<<<<<]>>>>>>>>>-]+[
```

図 3.5 とある巨大な Brainfuck プログラム（抜粋）

```
c[->cmc>c>+m]mc>cmc[[m]+[m]m-]+[mz>]m[m]mz+mc[-[-m+m]m]m+m+m[m]mz+[m[mzm]m[m]mz+
mc[-[-m+m]m]m+mc[-[-m+m]m]m+m[m]m[zm[m[-m+m]m[-m+m+m+<]m]m[m]m[m[-m+m]m[-m+m+m+m
]m]m[m]m[-m+m]m[-m+m+m]mc[[m]+>z>z>z>z>z>z>z>z>zm[m]m-]+[
```

図 3.6 とある巨大な Brainfuck プログラム（抜粋）を最適化

https://github.com/hogelog/sample-code/blob/master/dbtimes-vol01/bf-vm-opt.cpp

https://github.com/hogelog/sample-code/blob/master/dbtimes-vol01/bf-jit-opt.cpp

ここまでは対象言語を問わない汎用的な高速化手法を紹介しました。ここからは Brainfuck 特有の性質に着目した最適化を施してみましょう。

図 3.5 のプログラムをしばらく眺めてみると，いくつかの無駄が存在することに気づきます。まずは `++++`，`----`，`>>>>`，`<<<<` のような同じ命令の繰り返し。n 個の + 命令の繰り返しは n 加算する `c` 命令，n 個の − 命令は $-n$ 加算する `c` 命令に置き換えられます。同様に > と < も `m` 命令に置き換えられます。

またよく出てくる命令の組み合わせに `[-]` があります。これは「メモリの値が 0 になるまで − 命令を繰り返す」という処理，つまり「メモリの値を 0 にする」という `z` 命令に置き換えてしまいましょう。

ここまでの最適化で図 3.5 のプログラムが図 3.6 までに圧縮されます。特に `z` 命令の導入により，存在していたループ処理のいくつかが消えています。

このように，あとはひたすら「こういう組み合わせよく出てくるからこういう特定の命令に置き換えよう」という作業を続けるだけで，なんと世界で最速の Brainfuck インタプリタが完成したのでした[*14]。

「世界最速の処理系だ」と主張しているのに，まだ誰も「こっちの方が速い」と言ってこないので，たぶん世界最速なんじゃないかな。たぶん。きっと。おそらく。

[*14] https://github.com/hogelog/fast-bf

4

クラウド時代の DNS

suu-g (@suu_g)

インターネットが始まってから 20 年，ARPANET から数えると 40 年以上。インターネットはここ 10 年ほどで一気に生活に不可欠のものとなりました。そんなインターネットを支える基礎技術の一つが DNS です。

その DNS を取り巻く状況ですが，今年は随分と熱い一年間でした。普通のソフトウェア屋さんにとって DNS とは使うものであってそれ以外ではないのでしょうから，こちらの世界のニュースについてはあまり確認されていないことと思います。今年は，どんな問題が起きたのでしょうか？振り返ってみましょう。

- （日本）児童ポルノに関する DNS ブロッキング立法・ISP における運用が開始
- SOPA に Go Daddy が賛成を表明するも 1 日で撤回
- SOPA に反対を表明するため Wikipedia が 24 時間閉鎖する
- SOPA に反対を表明して Anonymous が Operation Blackout を表明
- v6-v4 フォールバック問題の対策法として AAAA フィルタリングが検討される
- Google, AAAA フィルタを導入している可能性のある DNS サーバに対して AAAA レコードを回答しない運用を宣言
- Ghost Domain 問題の発表
- ニフティクラウドの DNS 障害問題
- GMO クラウドの DNS で謎の障害がある
- さくらの DNS に（サブ）ドメインジャッキングが可能な脆弱性が発覚
- JPRS があらためてコンテンツサーバとキャッシュサーバの兼用の危険性を告知
- BIND, 長さ 0 の RDATA により異常終了する脆弱性（重複をお許しください）
- お名前.com，忍者ツールズの NS レコードを規約に基づき変更
- 今年の夏の DNS 祭りは BIND ではなく NSD

問題に限定しなければ他にも，DNSSEC.jp が活動を終了したり，gTLD の申請があったりと，今年は本当に色々なことがありました。

DNS の最初の RFC 1034, 1035 が出たのは 1987 年でしたが，それから 25 年経過し，「クラウド」時代と言われる今でもなお起こり続ける DNS 障害。その原因とは一体，何な

のでしょうか？この記事では，DNS を一種の分散システムと考えて，それをソフトウェアっぽい視点から見ていくことで，その問題の一端を解き明かします。解き明かすと思う。解き明かすんじゃないかな。まあ，ちょっと覚悟してください。

§1　DNS の正常系

§1.1　DNS とぼくらの世界

```
Query

{
  Header: {
    ID,
    is_query,
    Flags: {
      AA: 1 or 0,
      TC: 1 or 0,
      RD: 1 or 0,
      RA: 1 or 0,
    }
  },
  Query: [{
    Name: "www.example.jp.",
    Type: "AAAA",
    Class: "IN",
  }],
  Answer: [],
  Authoritative: [],
  Additional: []
}
```

```
Answer

{
  Header: { ... },
  Query: [{
    Name: "www.example.jp.",
    Type: "AAAA",
    Class: "IN",
  }],
  Answer: [{
    Name: "www.example.jp.",
    Type: "AAAA",
    Class: "IN",
    TTL: 86400,
    RData: "2001:DB8::163"
  }],
  Authoritative: [],
  Additional: []
}
```

```
Answer

{
  Header: { ... },
  Query: [{ ... }],
  Answer: [],
  Authoritative: [],
    Name: "example.jp.",
    Type: "NS",
    Class: "IN",
    TTL: 86400,
    RData: "ns.example.jp"
  Additional: []
    Name: "ns.example.jp",
    Type: "AAAA",
    Class: "IN",
    TTL: 86400,
    RData: "2001:DB8::160"
}
```

図 4.1　DNS パケットの JSON 風表記

DNS のパケットフォーマットは，RFC1034 [*1] ,1035 [*2] に載っている通りです。4 つのセクションがあり，それぞれ Query, Answer, Authority, それから Additional と呼ばれています。これを JSON 風に書くと，図 4.1 のようになってます。（正確には，DNSSEC や EDNS0 対応等でフラグはもう幾つかある）読者の皆様におかれましてはこのパケットの意味を判断しようとしてくれると期待しますが，勘の良い人だと，このプロトコルのもつ怪しさをなんとなく感じ取ったかも知れません。

さて，このようなパケットを元にして，DNS というシステムは成り立っています。このパケットを用いて Paul Vixie らが作ろうとしたのが，巨大な名前空間の木です。そして，その木をデータベースとしてユーザから参照できるようにした。それが DNS というシステムです。あ，ご存知ですよね。すいません。でも大事なことなんで。

ここで note しておきたいのが，DNS プロトコルが作られた当時の時代情勢です。DNS

[*1] http://www.ietf.org/rfc/rfc1034.txt Section 3-7

[*2] http://www.ietf.org/rfc/rfc1034.txt Section 4-1

は，/etc/hosts からの移行のひとつの可能性として生まれていたわけですが，その時代が今とは違っていて，例えば DNS でちょっと間違いが起こったとしても，それは手動で直していれば良かった。今じゃ警察沙汰だもん。

もっとも，DNS に関しては今でも法的にきちんとしているとは言い難いところです。DNS でのブロッキングの効果を法律で認めているわりには，JPRS が電気通信事業者というわけでもありませんし。このあたりは，そろそろ本格的に議論されるべき話ですよね。

さて話は戻ります。いま当時の時代情勢を取り上げたのは，DNS という分散システムが，設計段階から「異なる管理者同士で」ひとつの巨大な名前空間を管理するように作られている，ということを再認識したかったからです。各管理者が責任を持つ名前空間の範囲をゾーンと呼び，その空間の管理を他の人に任せることを委譲と呼んでいる。そういう信頼関係となっています。それらを前提として，内容に入って行きましょう。

§1.2　基本動作

DNS の正常系は，先に示しました 4 つのセクションと 5 つのフラグを使い，おおよそ次の 5 種類のパケットを用います。

1. Query (recursive)
2. Query (not recursive)
3. Response (Authoritative, with glue record in additional section)
4. Response (with Answer)
5. Response (against recursive query)

ユーザがキャッシュサーバに対して問い合わせを行い，そのキャッシュサーバがユーザ

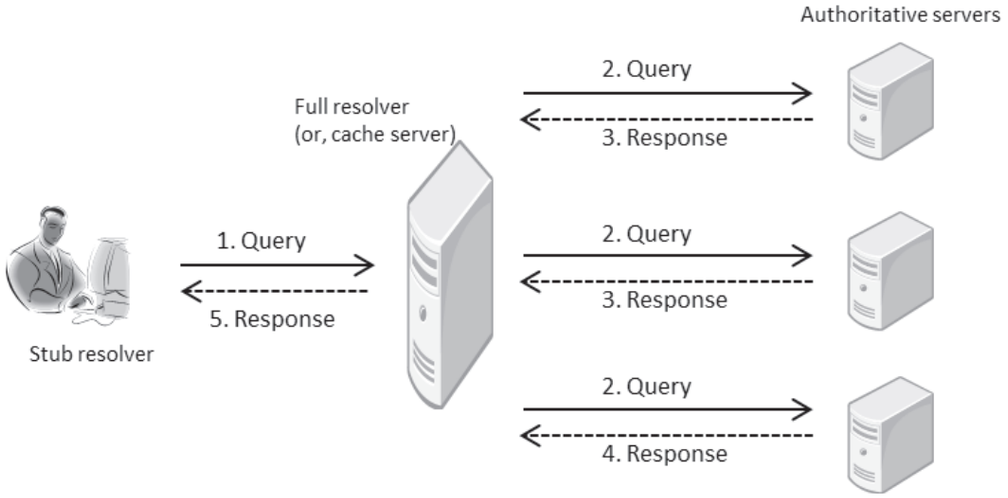

図 4.2　DNS においてやりとりされるパケット

の代わりにイテラティブにクエリを発行するときのパケットは，それぞれ図 4.2 のようになります。キャッシュサーバがキャッシュを持ったあとは，1 と 5 のパケットのみのやりとりとなります。ここまでは皆様ご存じですねー。また，このキャッシュサーバがコンテンツサーバも兼ねている場合，キャッシュサーバから直接 4 の，権威あるレスポンスが返ってくることになります。なお，権威のある回答は通常のキャッシュと比較して優先されるべきものということになっています[*3] が，スタブリゾルバではどちらも信用するので同じものとして扱われます。

正常系としては，こんなところでしょう。この決め事によって，ルートサーバを頂点とする綺麗なツリー構造が世界に作れます。

§2　DNS の異常系

以上のような DNS ですが，問題が山積みです。これはどう考えてもプロトコルが悪いのです。DNS のプロトコルがこんなに狂っているはずがない。

§2.1　Lame Delegation

これは基本ですね。 要は，上位のサーバでの正しくない設定です。構造的には， Visa ドメイン問題[*4] と言われているものもこれのうちのひとつですね。Lame delegation のあるドメインは，最悪の場合は即日乗っ取りが可能です。基本ったら最強ね。

§2.2　浸透さんって幽霊じゃね？

DNS の設定変更や引っ越しをしたときに，その結果が「浸透」するのを待たなければいけない…というのは有名な嘘です。DNS はキャッシュを持つだけなのだから，そんな現象は存在しない。…そう言われていたのですが，キャッシュに許容された以上の間，キャッシュが保持されてしまう問題が実在したことがわかりました。これが ″Ghost Domain″ [*5] 問題（日本語で「幽霊ドメイン」問題）です。その原因は，複数のサーバソフトウェアに同様に存在した実装不具合でした。

詳細な問題内容については，DNSOps の資料[*6] や「Geek なぺーじ」[*7] に詳しいのでそちらをご覧ください。

ソフトウェア的に考えたときに問題なのは，古いキャッシュが残存して昔のサーバへ問

[*3] http://www.ietf.org/rfc/rfc2181.txt
[*4] http://www.e-ontap.com/summary/
[*5] www.isc.org/files/imce/ghostdomain_camera.pdf
[*6] http://dnsops.jp/bof/20120425/20120425-DNSOPS.jp-BoF_Ghost_Domain_Names_v02.pdf
[*7] http://www.geekpage.jp/blog/?id=2012/3/21/1

い合わせを続けるということ，それから，問い合わせを続けていると古いキャッシュが expire しないという実装バグがあったということ。そして最後に，本来の DNS のツリーから外れたコンテンツサーバが，レコードを消さずに残しているために，昔のレコードをいつまでも返し続ける場合があるということ。この三点です。

本来の信頼関係から外れたところのデータが有効性を持ってしまい，おまけにその状態を意図的に作り出せる，という状態ですね。このような問題が今年までバグとして存在し続けていたわけです。

システムとして考えると，キャッシュを TTL 分保持することは至極もっともなようですが，信頼関係のツリーまで含めてキャッシュするということは，非常に責任が大きい行為でした。今回のように，「なぜかキャッシュが消えない」という異常事態になったとき，影響が大きくなります。当然，負荷とのトレードオフですけどね。

§2.3 サブドメインジャッキング問題

2011 年 6 月，さくらの DNS サービスにおいて，一時的にサブドメインジャッキングが可能になっていた問題が発覚しました。これは， example.jp ドメインが登録されていたとしても，別のユーザによって自由に www.example.jp ドメインを作成できてしまった，という問題です[*8]。これは，DNS の本質的な問題を幾つか激しく突いています。

ひとつには，DNS の作成しているツリーが理想的なものではないこと。理想的には権威サーバはすべて異なるサーバであるべきですが，現実的には相乗りが発生します。同じ階層のドメインやゾーンの相乗りならさほど問題はありませんが，親子ゾーン・孫ゾーンが同じサーバ上に乗ると，サーバ同士の依存関係が崩れてしまいます。結果，「正しい依存関係」を順番に辿って回答にたどり着く，という DNS の基本動作から外れることになり，鼻から悪魔が出ます。

ふたつめとして，ゾーンと言う概念が，プロトコルや実装とかけ離れてしまっているという点があります。ドメインのデリミタはドットですが，ドットがあったからと言ってゾーンの切れ目となっているかどうかは不明です。example.jp ゾーンを他へ委譲したサーバ上で www.example.jp の A レコードの登録は（ソフトウェア的に）できますし，www.example.jp を問い合わせたときにどちらが返って来るかは実装依存になってしまっています。

最後に，フルリゾルバがどのコンテンツサーバにも同じクエリを投げる，というプロトコルの怪しさがあります。これは，どこでゾーンが分かれるかが問い合わせを行うまではキャッシュサーバには知ることができない，ということとは符合します。しかし，キャッ

[*8] さくらはたまたま注目を浴びただけで，問題を抱えたサービスプロバイダは他にもたくさんある…が，その話はまた今度

シュサーバはゾーンの別れ方について，知らないでは済まされない。問い合わせを行ったキャッシュサーバは，即座にその結果をキャッシュします。このキャッシュを正しく行うためには，「現在問い合わせている先がどのゾーンの Authority を持っているか」という情報が必須です。したがってキャッシュサーバは問い合わせの際に状態を持っています。状態を持っているのに，その状態を利用して問い合わせ内容を変更しなかったこと。これが大罪です。もしも問い合わせがゾーンごとに行われていれば，子ゾーンについて問い合わせているところで孫ゾーン情報が返ってきたり，あるいは A レコード問い合わせに対して謎の NS レコードをキャッシュさせられたりすることは無かったでしょう。どう見てもプロトコル上の脆弱性です。本当にありがとうございました。

これらの問題点を合わせて起こった問題が，このサブドメインジャッキング問題です。「DNS サーバの相乗りという業務形態自体が，プロトコルとの相性が悪いのではないか」と言った方がいましたが， その意味が分かるのではないでしょうか。

§2.4　キャッシュ兼用サーバ

コンテンツ・キャッシュ兼用サーバはしばしば問題と言われます。先に示したような，自由に偽ドメインを入れられるコンテンツサーバをフルリゾルバとして使用するのは，自分のホストの /etc/hosts の書き換え権を他人に委ねているようなものだ，というのはお分かりいただけるかと思います。

さて，ソフトウェア的に見た時に，キャッシュ兼用サーバに潜む DNS の問題はというと…，これは優先順位の問題があります。Authoritative なサーバからの Answer セクションは，他のどのレコードよりも優先されるべきだと RFC2181 にあります。が，毒入れの可能性を考えたときにもっとも危険性が高いのも，この Authoritative な Answer セクションです。本来信頼性が最高レベルのものが一番信用できなくなるのは，プロトコルとして崩壊していますね。わぁい崩壊。

§3　まとめ

以上に挙げた問題は，この「クラウド」な一年間で実際に「クラウド」な業者のもとで起き，社会的に影響を与えたものです。DNS の問題の多くはそのプロトコルへの不理解によるものですが，そういうヒューマンエラーが起きやすいような仕組みが，DNS には備わっています。皆様方におかれましては，この DNS という「想定外に利用されてしまった」プロトコルの問題を他山の石とし，これからのクラウドちっくな分散システムの開発にぜひ活かしていただければ，この記事のタイトルが救われるというものです。

以上，タイトル詐欺の記事でした。

5

IPv4 がこの先生きのこるには

yuyarin

§1 はじめに

インターネットの主要技術である IPv4 というプロトコル。その中で個体識別子として使われる IPv4 アドレスは 32bit で表現され数は約 42 億個である。かつては「無限アドレスだぜヒャッハー」と湯水のように消費されていたのだが，70 億もの人間が社会インフラとしてインターネットを必要としている今となっては，このアドレスの数が全く足りなくなってしまっているのだ。レジストリから事業者への IPv4 アドレスの新規割り振りはほぼ終了しており，IPv4 アドレスは「枯渇」したと表現されている。

この枯渇を見越して作られた次世代プロトコルが IPv6 である。IPv4 の失敗を踏まえて色々な改良は施されているが，なんといっても 128bit のアドレス空間が魅力である。国内主要 ISP のバックボーンは既に IPv6 に対応できており，一部の ISP では利用可能になっているが，中小 ISP や地方 CATV，ICP[*1]ではまだ対応できていない。

1998 年に RFC2460 にて IPv6 の仕様が発表されてから既に 14 年が経っているのだが，未だにほんの一部にしか普及していないのだ。ISP は通信相手の ICP が IPv6 に対応していなければ対応する必要性は薄いし，ICP もほんの一部の ISP の一部のユーザしか IPv6 を使えないのに対応するのは金がかかるだけなのだ。デッドロックに陥っているのだ。

さすがにこのままではダメだということで，2012 年 6 月 6 日には World IPv6 Launch として，Google を筆頭に世界中の主要サイトが IPv6 を恒久的に有効にするという取り組みが行われた。こうして徐々にデッドロックが解けて IPv6 への対応が進んでいき，IPv4 アドレスの枯渇問題がそれを更に加速させるだろう。そして IPv6 が主要のプロトコルになり IPv4 は古の技術として扱われるようになる未来が来るだろう。

だがしかし，すべての通信が IPv6 に対応できるまで，我々はアドレスが枯渇している IPv4 を必死に「延命」させなければならないのだ。

前置きが長くなったが，この章ではそんな「IPv4 延命技術」を紹介しようと思う。

[*1] Internet Content Provider

§2 IPv4 延命技術概観

枯渇によって IPv4 アドレスの割り振りを受けられなかった ISP において，顧客に割り当てる IPv4 グローバルアドレスが無くなってしまうというシナリオが，今後実際に起こるだろう。というか起きてる。この時に必要になるのは**顧客間で IPv4 グローバルアドレスを共有する**方法である。IPv4 アドレスの共有には NAT[*2]が使われる。

主要な延命技術を NAT の場所と IPv4 パケットの運び方で分類した表を表 5.1 に示す。

	IPv4 ネイティブ	IPv6 トランスポート	
		カプセル化	トランスレーション
CPE で NAT	いまここ	MAP-E(旧 4rd)	MAP-T(旧 dIVI)
ISP で NAT	CGN	DS-Lite	464XLAT

表 5.1 NAT の場所とトランスポート方法による IPv4 延命技術の分類

§2.1 IPv4 ネイティブ方式

現在の IPv4 ネットワークでは CPE[*3]，いわゆるご家庭のルータで NAT をしている。これも 1 家庭 (CE[*4]) で 1 アドレスを共有するという 1 つのアドレス節約術であり，実は既にこの延命戦は始まっていたのだ。これに更に ISP 側で NAT を重ねることで複数の家庭で 1 つのグローバルアドレスを共有するのが CGN(Carrier Grade NAT) である[*5]。

§2.2 IPv6 トランスポート方式

それに対して CPE に IPv6 アドレスを割り当て，その IPv6 を利用して ISP 網内の IPv4 バックボーンまで IPv4 パケットを通す方法がある。これは IPv6 の使い方によってカプセル化とトランスレーションに分けられる。カプセル化はいわゆるトンネルのことで，CPE と ISP の間で IPv4 over IPv6 トンネルを張り，IPv4 パケットを IPv6 パケットのデータとして運ぶ。トランスレーションでは IPv4 パケットを IPv6 区間を通る間だけ IPv6 パケットに変換する。

これらは更に NAT をする場所によって分類できる。

[*2] 実際には NAPT だが以降 NAT と呼ぶことにする
[*3] Customer Premises Equipment
[*4] Customer Edge
[*5] 表では ISP で NAT としているが，実際には CPE と ISP の両方で NAT を行うことになる

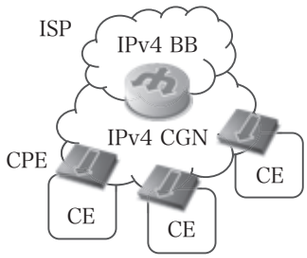

図 5.1　IPv4 ネイティブ

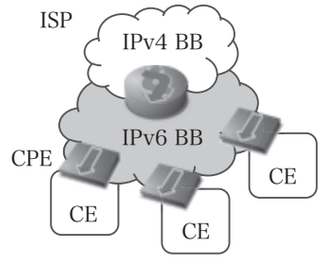

図 5.2　IPv6 トランスポート

CPE で特殊な NAT をして，プライベートアドレスからグローバルアドレスへ変換した後，IPv6 でカプセル化するのが MAP-E(旧 4rd)，IPv6 へトランスレーションするのが MAP-T(旧 dIVI) である。MAP における NAT の特殊性は外部送信元ポートとして利用されるポート番号が各 CPE で限定されることである。

プライベートアドレスの IPv4 パケットを IPv6 でカプセル化して ISP 網内へ運んだ後，ISP 側で NAT(CGN) してグローバルアドレスへ変換するのが DS-Lite(Dual Stack Lite) であり，プライベートアドレスの IPv4 パケットを IPv6 パケットにトランスレーション (NAT46) して，ISP 側で NAT(NAT64) するのが 464XLAT である。

これらの技術においては複数の CE で 1 つの IPv4 グローバルアドレスを共有することになる。いくつの CE で共有するかが設計のパラメータとなり，目安としては 256 程度である。

§3　ステート管理のお話

§3.1　CGN の NAT の問題

NAT の根底にある設計思想はアドレスだけでは識別子が足りないのでポート番号も識別子に使ってしまおうというものである。NAT では (内部アドレス，内部送信元ポート，外部アドレス，外部送信元ポート) というバインディングをセッション情報として一定期間保持している。そうしないと外から戻ってきたパケットを内側の誰に送って良いのかわからなくなるからである。

CGN では何千という家庭の通信を NAT しなくてはならないので，NAT のセッションデータベースは巨大なものになり生成頻度も高くなる。ISP はユーザの通信を最低限 90 日間は保存しなくてはならないのだが，この時に NAT のデータベースも併せて保存しておかないと通信元の顧客が特定できなくなってしまう。

これらの処理はソフトウェアで行う必要があるため，ネットワーク機器からすれば意外

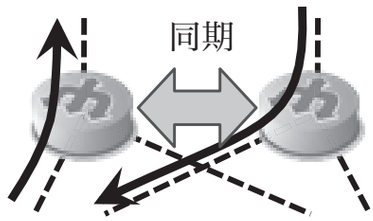

図 5.3　行きと帰りで違う箱を通るパケットのために必死で DB を同期する NAT 箱達

と重い処理なのだ。顧客数を n とすると定数項が大きい O(n) の仕組みなのだがネットワーク業界ではこれはスケールすると言わなかったりする。

§3.2　IPv6 トランスポート方式での CPE 特定の問題

DS-Lite 以外の IPv6 トランスポート方式では ISP から CPE へ IPv4 パケットを送ろうとした時に，どの IPv6 アドレスの CPE に対してパケットを送るのかを特定しなければならない。NAT に非常によく似た状況であるのだが，ここで IPv6 のアドレスの長さを活かした工夫がされている。

MAP では IPv4 アドレスとポート番号が決まれば IPv6 アドレスが相互に一意に定まるような「ルール」を決めてアドレス変換を行う，Stateless Address Mapping(SAM) という方式を採用している。MAP で送信元ポート番号が特定の範囲に制限されるのはこのためである。ルールさえ各機器で共有してしまえば CPE を特定するためのデータベース (ステート) の管理をする必要がないのだ。464XLAT では RFC6145 の Stateless IP Translation を行なっており，こちらもステートを持たない。

この仕組みは顧客数 n に対して定数オーダー O(1) で処理できるために良くスケールする。ISP 側に置く設備 1 つに対して帯域が許す限り多くの CPE を収容することができる。

§3.3　ISP 側 NAT における可用性の問題

ISP では当然ネットワークの冗長性を確保しなくてはならない。ISP 側で NAT する場合は 2 台の巨大 NAT 箱を置かなくてはならないのだ。

Act-Standby 構成の場合，故障した時に即座に Standby 機を起動して故障した機器から NAT のセッションデータベースを受け取らなくてはならない。故障しているのにそのようなことはできるはずがないので，当然ながら Act-Act 構成になる。

IP ネットワークでは往路と復路が必ずしも一致しない。ある NAT 箱を通って出ていったパケットの応答が別の NAT 箱に戻ってくることは，当たり前に起こりうることであ

る。よって 2 台の NAT 箱は常に NAT の巨大なセッションデータベースを同期し続け無くてはならないのだ。もちろん非常に大変な処理なのだが，そこは力技でなんとかする感じである。

§4　結局どの技術がイケてるのか？

§4.1　CGN

スケールもしないし泥臭いので大変な割に技術的にはダサい。でも既存の家庭用ルータには手を入れずに，ISP に設備を置くだけで実現可能なのである。ハイスペックな機器があって，収容する CE の数を上手に設計して，その分お金をかければなんとかなるのかもしれない。そういう意味では現実的な技術である。

§4.2　MAP-E(4rd), MAP-T

MAP は技術的な面では非常にイケている。ステートレスなアドレス変換を行うため定数オーダーでスケールところが非常に素晴らしい。

ただカプセル化をする場合は MTU の問題とフラグメントの問題が発生する。実装面での負荷は高いが既存の MAP(4rd) 実装ではこの点を既にクリアしている。7 月頭に開催された JANOG[*6] 30 Meeting では MAP-E の前身技術である 4rd の相互接続実験が実施された。また，トランスレートする場合は IPv4 の属性が IPv6 に変換される時点で失われてしまう。

イケているのだがデプロイが非常に難しい。NTT のホームゲートウェイや Buffalo や NEC のルータなど家庭用ルータに MAP の複雑な実装を入れなくてはならない。ビジネス的な視点で見るとこれがとてつもなく困難になってしまうのだ。

§4.3　DS-Lite

ISP で NAT するので CGN 同様にスケーラビリティはない。CPE にも機能を入れなくてはいけない。技術的にはあまりイケてないと思っている。

§4.4　464XLAT

ステートフルトランスレーション (RFC6146) とステートレストランスレーション (RFC6145) という既存の標準技術の組合せで作られていて非常に美しい。NAT64 に始まるトランスレーション技術の最終形態とすら思える。

[*6] JApan Network Operators' Group

しかしやはり ISP 側で NAT をしてステートを持ってしまうためスケールしない。また各家庭の CPE に機能を追加する必要が出てくる。今のところ CPE(CLAT) では NEC-AT などの実装がある。ISP 側の機器 (PLAT) は NAT64 のため既に多くのベンダで実装が行われている。

§5 おわりに

技術的には MAP や 464XLAT がイケてるのだけれども，ビジネス的な面では CGN でないとデプロイできないだろう。とはいっても今回の記事で紹介したのはひとつの切り口であって，他にも語れなかった判断要素があるので，他の技術も活躍できる場所が多いにあるのだ。結局は適所適材なのである。

アプリケーションを開発する人には，今後このように家庭間でのアドレス共有ということが ISP で行われることを知ってもらいたい。もはや IP アドレスだけでは誰かを特定できないのだ。1 つの IP アドレスでの BAN が何千世帯に対して影響することが今後起こりうるのだ。2 ちゃんねるでプロバイダごとアク禁を食らったかのようなことが，いとも容易く起きてしまうのだ。

アドレスが枯渇した IPv4 を，我々はこのようなアドレス共有技術を用いてちまちまと延命していかなくてはならない。自律分散システムであるが故に，地デジのように「今日でアナログ放送は終わりです」ということはできない。

IPv4 はこの先生きのこらなければならず，そのためにはこうした技術や仕組みを考えて，標準化して，製品に組み込んで，設計をして，お金をかけて現実の「インターネット」にデプロイしなくてはならないのだ。

とっくの昔から，インターネットが止まると人が死ぬ時代になってしまっている。IPv6 が世界を覆い尽くす日が来るまで IPv4 を生かし続けるための戦いは始まったばかりだ。

6

電算機技能者的同人誌執筆環境構築概論

はやみず

技術者たるもの，同人誌を書く時であってもその技術的ノウハウを積極的に投入して，執筆作業を最大限に効率化しなければならない。そのような信条のもとに，本誌の執筆環境は構築された。

§1 LaTeX

文章執筆は LaTeX に限る。

文章ファイルの保存形式は，やはりプレインテキストでなければならない。プログラマは各自の研ぎ澄まされたテキストエディタ環境を有している。即ち，執筆速度を最大化するためには，各々が最大限能力を発揮できるテキストエディタ環境を利活用することが最善である。そのためには，やはりプレインテキストでなければならない。

そして，我々プログラマは情報を構造化されていない状況に極度のストレスを覚える生き物である。ある程度以上の長さ，内容の高度さをもつ文章を執筆する場合には，図表や章・節の参照，文献管理，目次の作成，整合性のある文章スタイル調整など，文章全体が高度に構造化され，文章内の要素が有機的に結合されていなければならない。これを為すためには，高度かつ柔軟な組版システムを利用することが必要不可欠である。

やはり，文章執筆は LaTeX に限る。

本誌の組版には TEXLive 2011 を用いている。文書のクラスには奥村先生の `jsbook.cls` を利用しており，`tombow` オプションを利用することでトンボ付きの出力を得ている。`tombow` オプションを利用する際には，併せて `papersize` オプションを利用しないと，文書のサイズとしてはトンボ無しで出力されてしまうので注意が必要である。

§2 GitHub

文章執筆は GitHub に限る。

複数名で共同作業を行う場合，最近では Dropbox が用いられる場合が多い。Dropbox

は便利である。一度導入してしまえば，誰でも簡単にファイルを即時に共有することができる。

しかし，我々はプログラマである。プログラマは変更履歴を重んじる生き物である。即ち，バージョン管理システムはプログラマにとって必須である。Dropbox にも履歴管理機能は存在するが，限定的でありインターフェースも非効率なものである。

また，プログラマは編集の競合に注意を払う生き物である。特に本誌のように締め切りのあるようなものの場合，締め切り直前には多数の編集が競合することが予想される。Dropbox では，競合の解決をシステマティックに行うことは不可能である。一方，バージョン管理システムはその基本機能として競合解決のシステマティックな方法を提供する。バージョン管理システムに高いリテラシを有する我々にとって，これを用いないことは有り得ない。

そして今日，GitHub はプログラマの共通基盤である。殆どのプログラマは，各々の SSH 公開鍵を GitHub に登録済みである。即ち，GitHub をバージョン管理システムのホストとして用いることで，最小限の手間で共同作業基盤を構築可能である。

幸いなことに，私は GitHub のプライベートレポジトリを作成できるアカウントを保有している。そのため，原稿はクローズドな形で管理することが可能である。

やはり，文章執筆は GitHub に限る。

§3　継続的インテグレーション

文章執筆は継続的インテグレーションに限る。

我々は重要な事実に目を向けなければならない。LaTeX の基盤たる TeX という言語は，単なるマークアップ言語にあらず。チューリング完全性を有する，完備なるプログラミング言語であり，LaTeX による文章執筆とは即ちプログラミング行為であり，ソフトウェア開発にほかならないのである。

ソフトウェア開発において，今日最も重要視されている開発基盤の一つが継続的インテグレーションである。バージョン管理システムにコードがコミットされる度に，プログラムがビルドされ，テストコードが走り，常に最新の成果物が生成された状態が維持される。一度誤ったコードがコミットされた折には，そのことが開発者に直ちに通知され，更なる状況の悪化を食い止めるフィードバックループを形成する。

本誌の執筆においては，継続的インテグレーションシステム Jenkins を導入し，GitHub のコミットにフックして PDF ファイルの生成を行うことにより，即座に入稿可能な状態の原稿が常に確認可能な環境を構築した。

やはり，文章執筆は継続的インテグレーションに限る。

～雑談～

- (はやみず) もともとこの本を書くきっかけになったのは、去年の冬コミで知り合いが謎の技術系同人誌とか随筆集みたいなのを出していて、ああこういうのもアリかと思ったこと
- (はやみず) 大学で DB 系の研究をしていて、研究自体は諸般の事情で口外できないことが多くて、何でもいいから人に自分の研究に関わる話を聞いてもらいたい欲みたいなものが高まっていたのも相まって、うっかりコミケに応募したら通ってしまった
- (はやみず) そして 7/13 現在、入稿締切があと 10 日後に迫っているにも関わらず、半分も原稿が書き上がっていないという
- (yuyarin) ネットワーク屋なのに DB を冠する同人誌に誘われてほいほいついてきちまったぜ！
- (はやみず) Oracle Exadata とか Teradata とか並列データベースアプライアンスはマシン間の高速インターコネクト込みでシステムが作られてるけど、最近は大規模 Hadoop クラスタとかでネットワークのことも考えないとまともにデータ処理できなかったりするんだよなあ
- (yuyarin) この雑談を書いている今，金曜ロードショーでサマーウォーズがやっているが，締め切り目前の俺達が真のサマーウォーズだぜ！！
- (はやみず) NOSQL の基礎知識を買ってパラパラ眺めてみたら、色々と書きたいことが湧いてきたので冬コミに向けてネタを仕込もう
- (suu-g) 締め切り前日、よーやく字を書いたので初コミットなり。ボリューム感 間違えた…
- (suu-g) 締め切りまであと 2 時間ですがまだ改訂ます。CEP 作ってみたい。簡単なやつ。あと Ruby 用のまともな FUSE バインディングが欲しい。
- (はやみず) 入稿準備おわったー！

著者紹介

■ **はやみず**　某研究施設にて、データベースシステムと戯れる日々を過ごしている。昔は Lisp 系言語に傾倒したり、マルチコア向け並列処理フレームワークの研究をしたりしていた。この本の首謀者。

■ hogelog　明治座という歴史ある建物で日々プログラミングをして過ごしている。

■ suu-g　皇居の近くで雲みたいにフワフワしたものの開発をしている。DNS は趣味。

■ yuyarin　インターネットを守るために日本のインターネットの中心地である大手町で日々戦っている．美味しいビールとスコッチウイスキーが生きる歓び．

■ Eliza (**表紙デザイン**)　人生のほとんどを画像処理と行列演算に費やしている謎の美女。高次元空間に迷い込んだら、会えるかもしれない。

The Database Times vol. 1

タイトル	The Database Times vol.1
発行日	2012 年 8 月 11 日
サークル	Hotchpotch Society
著者	はやみず、hogelog、suu-g、yuyarin
表紙デザイン	Eliza
連絡先	*yuto+c82@hayamiz.com*
ウェブサイト	*http://hayamiz.com/~hotchpotch/*
印刷所	株式会社ポプルス