# *第２章　【PostgreSQLの特徴】*

当章では PostgreSQL の特徴の説明を行う。

## 2－1　特徴

### 1）　データ型

PostgreSQL はデータベースの標準言語である SQL92 をサポートしている。
主キー、サブクエリー等 SQL92 の主な機能はほとんどサポートしている。

データ型に関しても同様にそのほとんどをサポートしている。更に、PostgreSQL 独自に拡張したデータ型も存在する。PostgreSQL の型のうちいくつか特徴的なものを紹介する。

- シリアル型

  自動連番、いわゆる MS_ACCESS のオートナンバー型のようなものである。

  重複しない値が割り振られることが必ず保証されるので連番を割り振りたい項目等には非常に好都合だといえる。しかし、例えロールバックが発生したとしても、一度取得した値は欠番となることを認識しておく必要がある。

- TEXT 型

  可変長文字列。データ長が存在せず、常に可変長でデータを扱うことが可能である。

  しかし、PostgreSQL7.0 以前までは一行の大きさが 8000 バイトまでという制約があったために、この型の恩恵は十分には得られなかった。が、7.1 以降に TOAST（The Oversized-Attribute Storage Technique）という機能が拡張され、一行あたり最大 1G バイトのデータが格納できるようになり、実質一行あたりのデータサイズの制限は取り払われている。

- 幾何データ型

  ２次元データを表現できる型。POINT、CIRCLE 等２次元を表すデータを扱うことが可能である。地図の座標データ等を格納するのに適したデータ型といえる。

  ＳＱＬを用いてのデータの挿入、削除、更新も楽に行えることが特徴である。

以下の表２-１に上述の型も含めた PostgreSQL のデータ型の一覧を掲載する。

| データ型名 | 別名 | 説明 |
|---|---|---|
| Bigint | int8 | 8バイト符号付整数 |
| Bit | | 固定長ビット列 |
| Bit varying(*n*) | varbit(*n*) | 可変長ビット列 |
| Boolean | bool | 真偽 |
| Box | | 矩形 (左下の座標、右上の座標) |
| character(*n*) | char(*n*) | 固定長文字列 |
| character varying(*n*) | varchar(*n*) | 可変長文字列 |
| Cidr | | IP ネットワークアドレス |
| Circle | | 円 (中心点の座標、半径) |
| Date | | 日付 (年月日) |
| double precision | float8 | 8バイト浮動小数点 |
| Inet | | IP ホストアドレス |
| Integer | int, int4 | 4バイト符号付整数 |
| Interval | | 日付/時刻の差 |
| Line | | 直線 (線上の1点、線上のもう1点) |
| Lseg | | 線分 (始点の座標、終点の座標) |
| macaddr | | MAC アドレス |
| money | | 米国通貨 |
| numeric(*p*, *s*) | decimal(*p*, *s*) | 精度の高い整数と小数 |
| Oid | | オブジェクト識別子 |
| Path | | 経路 (点1、点2、...) |
| point | | 点 (x,y) |
| polygon | | 多角形 (点1...) |
| Real | float4 | 4バイト浮動小数点 |
| smallint | int2 | 2バイト符号付整数 |
| serial | | 4バイト自動増加整数 |
| Text | | 可変長テキスト (無制限) |
| Time [ without time zone ] | | 時刻 (時分秒) タイムゾーンなし |
| Time with time zone | | 時刻 (時分秒) タイムゾーン付き |
| timestamp [ with time zone ] | | 日付と時刻 |

**表 2-1　PostgreSQL のデータ型**

## ２）　テーブル継承

一般的なリレーショナルデータベースモデルにおいては集合の汎化、特化という概念が存在する。俗にスーパーセット、サブセットと言われているものである。

顧客というスーパーセットに対し、そのサブセットとして個人顧客、法人顧客といった具合である。または、学生というスーパーセットに対する大学院生、大学生といったもの等である。では PostgreSQL においてはどうだろうか？

PostgreSQL はテーブルの継承という概念が存在する。PostgreSQL がオブジェクトリレーショナルデータベースとして、その特徴を最も表わしているのがこのテーブルの“継承

（INHERITS）”という機能だといえるだろう。基本となる親テーブルがあり、そのテーブルよりテーブル構造とデータを継承した子テーブルを作成するのである。

例えば、顧客情報を考えてみよう。共通の情報として以下のものがあるとする。

顧客情報：　　顧客コード、初回購入日、最終購入日

これは、個人顧客であっても、法人顧客であっても共通する基本的な情報である。
この情報より、個人顧客と法人顧客を区別するためにそれぞれ必要な情報を考えると
以下のようになる。

個人顧客情報：　　顧客名、自宅電話番号、Email
法人顧客情報：　　法人名、代表者氏名、担当部署名、担当部署電話番号、代表 Email

個人顧客情報と法人顧客情報は、属性はそれぞれ異なるが顧客情報という意味では共通に扱うことが可能である。よってここに顧客情報と個人／法人顧客情報との間に継承が成立することになるのである。（図２-１参照）

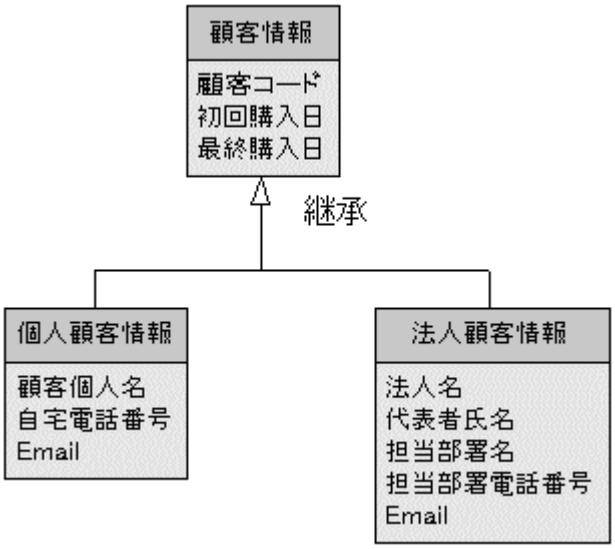


**図 2-1　テーブルの継承**

では上記のような継承関係にあるテーブルを PostgreSQL で作成する為にはどうすればよいのだろうか。このとき必要となるのが INHERITS というオプションである。

```
CREATE TABLE  テーブル名(  カラム定義  ) INHERITS(  継承するテーブル名  );
```

つまり、個人顧客情報、および法人顧客情報テーブルは以下のような SQL となる。

```
CREATE TABLE  個人顧客情報(  顧客個人名、自宅電話番号、Email    )
INHERITS(  顧客情報  );
CREATE TABLE  法人顧客情報(  法人名、代表者氏名、担当部署名、
担当部署電話番号、Email    ) INHERITS(  顧客情報  );
```

では、継承関係にあるテーブルにはどのように値が格納されるのだろうか？

```
INSERT　INTO 個人顧客情報 VALUES( 1、’20030101’、 ’ ’、 ’田中’、
’097-XXX-XXXX’、 ’AAA@BBB.JP’) ;
INSERT　INTO 法人顧客情報 VALUES( 2、’20030110’、 ’ ’、 ’○×商事’、
’ 佐藤’ 、’ 営業部’、 ’03-YYYY-YYYY’、 ’CCC@DDD.JP’) ;
```

まず、継承関係にあるテーブルの子テーブルに対し、レコードを挿入する際は親のテーブルの項目に対しても、必須項目に対してはレコードを挿入する必要性があることに注意されたい。ただし SQL での挿入先はあくまでも子となるテーブルとなる。

この状態で顧客情報テーブルはどのようになっているのだろうか？
顧客情報テーブルに対し SELECT 文を発行してみた。

```
SELECT　*　FROM 顧客情報;

顧客コード　初回購入日　最終購入日
　1　　20030101
　2　　20030110
```

顧客情報テーブルに対し、INSERT 文は一度も発行していないにも関わらず顧客情報テーブルに対しレコードが追加されていることが確認できた。
では個人顧客情報テーブルはどうであろうか？

```
SELECT　顧客コード、顧客個人名、自宅電話番号　FROM 個人顧客情報;

顧客コード　顧客個人名　自宅電話番号
　1　　田中　　097-XXX-XXXX
```

結果は INSERT 文で挿入した１件のみであった。法人顧客情報テーブルに対しても、SQL を発行したが結果は INSERT 文で挿入した１件のみであった。

次に更新処理を行う。

```
UPDATE　個人顧客情報
SET 最終購入日=’20030131’ 、自宅電話番号=’ 097-ZZZ-ZZZZ’
WHERE 顧客コード =１;
```

結果を確認する為に個人顧客情報テーブルに対し SELECT 文を発行する。

```
SELECT　顧客コード、最終購入日、顧客個人名、自宅電話番号
FROM 個人顧客情報;

顧客コード　最終購入日　顧客個人名　自宅電話番号
  1          20030131        田中       097- ZZZ-ZZZZ
```

正しく更新されていることが確認できる。
続いて顧客情報テーブルに対して SELECT 文を発行する。

```
SELECT　*　FROM 顧客情報;
顧客コード　顧客個人名　最終購入日
  1          20030101      20030131
  2          20030110
```

個人顧客情報テーブルへの更新が親テーブルである顧客情報テーブルへ正しく反映されていることが確認できた。

以上の動きからも分かるように、継承されたテーブルに対してのデータの登録、更新は継承元のテーブルにも反映されるということである。

つまり継承関係を表現できるものであれば、非常にオブジェクティブなデータモデルを PostgreSQL は構築できるのである。

しかし、（これはスーパーセット、サブセットの導入にも言えることだが）その継承関係を的確にとらえることができてこその概念、機能であるが故に、当機能を使う場合には物理設計段階におけるオブジェクトの洗い出しには細心の注意を払う必要性があるといえるだろう。

### 3）　トランザクション

リレーショナルデータベースを用いてシステムを構築する際に重要なファクターとなるもの、それがトランザクションである。PostgreSQL のトランザクション管理機能は十分なものなのであろうか？

PostgreSQL にはトランザクションレベルのうちのリードコミティッドがデフォルトで設定されている。これはつまり【ダーティリード】（コミットされていないトランザクションが書き込んだデータを別のトランザクションが読み込むこと）がないということである。

デフォルトで設定されているがため、たとえ無意識でも【ダーティリード】が発生することはありえないものとなっている。

また、あるトランザクションが過去に読み込んだデータをもう一度読み込もうとしたとき、あ

るトランザクションによって別の値に書き換えられていてその値を読み込んでしまう、【Non-repeatable read】に対してはどうであろうか？

この状況に対して、PostgreSQL はトランザクションレベルの中でも最もレベルの高いシリアライザブルというトランザクションレベルを実装している。

これにより参照トランザクション上でもデータの不整合を発生させないようになっている。

### ４）　バックアップ

データベースの運用となると考えなければならなくなるのがバックアップである。

データは代替の効かないものであるという性質は PostgreSQL も当然その例外ではない。

PostgreSQL にはバックアップ機能として【pg_dump】というものが用意されている。

【pg_dump】はホットバックアップが可能となっている。ホットバックアップとはデータベースを稼働させたままバックアップを取ることである。

バックアップ中にデータが更新、削除が発生する可能性はあるが、PostgreSQL のホットバックアップはバックアップを開始した時点のデータを対象にコピーを実行するのでバックアップデータに不整合が発生することはない。しかし、データの更新、削除等には対応しているが、テーブルの作成や削除には対応していないため注意が必要である。また差分のみのバックアップ等も行えないため、バックアップ時間やバックアップデータのサイズに関しては商用データベースより劣ることも認識しておかなければならない。

また、商用データベースのトランザクションログに相当する機能として、WAL（Write Ahead Logging）というものが PostgreSQL には存在するが重大な障害からの復旧や、消失したデータの復旧は現時点では無理である。

現状ではＯＳのバックアップ機能（tar、cpio 等）との併用が良いと思われる。

## 2―2　削除を行ってもサイズが小さくならない？

### １）　PostgreSQL のデータ管理

PostgreSQL はデータを削除してもデータファイルは小さくならない。何故か？

それは PostgreSQL、最大の特徴の一つといえるデータ管理の方法にその答えを見つけることができる。

PostgreSQL のデータ管理は追記型である。つまりデータを削除しても（DELETE 文を発行しても）データに対しフラグをたて無効にしているだけなのである。

更新の際も同様である。古いデータに対してはフラグをたて無効化し、新たなデータを加えている。つまり、データファイルのサイズが増えていくのは PostgreSQL の世界では必然の話なのである。

ではその仕組みはどうなっているのであろうか？

PostgreSQL のすべてのオブジェクトには OID というユニークな値が割り振られている。

PostgreSQL ではデータ行もオブジェクトとして考えられている。つまりレコードがテーブルに対して新たに挿入されるとその都度ユニークな値が行に対し割り振られるのだ。またその際

XID という連番も行に対し割り振られている。

ある行に対し更新処理を行うとその行の古いレコードには削除フラグが立つ。

そして、その古い行と同じ OID を持つ新たなレコードが挿入される。このとき新たに追加された行の XID は古い行の XID とは異なる値を持つことになる。

これにより更新される前のレコードは論理的にテーブルより削除され新たなレコードが更新結果として残るわけである。

顧客テーブルの顧客番号 001 のレコードに対して更新処理を行った結果は以下のようになる。XID　001 のレコードは論理的に削除され、新たに XID　002 のデータが追加される。

顧客テーブル

| OID | XID | 顧客番号 | 顧客氏名 | 削除フラグ |
|---|---|---|---|---|
| 5001 | 001 | 001 | 鈴木一郎 | ON |
| 5001 | 002 | 001 | 鈴木次郎 | OFF |

**表 2-2　顧客テーブル**

追記型であるが故に、レコードの更新、削除は速くなる。がしかし、削除されデータはあくまでも論理的に削除されたのみで物理的には残りつづけるため、PostgreSQL ではデータを削除してもデータファイルは増え続けるのである。

### 2）　増え続けるデータファイルをどうする？

このような追記型でデータを管理する PostgreSQL だが、当然そうなると問題になってくるのはその増え続けていくデータファイルだ。

本番運用とともに間違いなくデータベースは**虫食い状態**、いわゆる断片化に陥る。

（注　本論文では以下、断片化を**虫食い状態**と表現する。）

放っておけば間違いなく膨大なサイズのデータファイルがディスク領域を占めるようになり、データベースのレスポンスにも重大な影響を与えることになるのは必至である。

更新が頻繁に発生するシステムならば、この問題は尚更深刻である。

だが、PostgreSQL には【vacuum】というコマンドが用意されている。

使わなくなった領域を再利用したり、削除フラグがたち論理的に削除状態になっているレコードに対し実削除を行うことが可能となっている。

また、PostgreSQL7.1 までは【vacuum】を実行するとその対象となるテーブルがロック状態になってしまい、その間処理が待たされていたが、PostgreSQL7.2 以降になると【concurrent vacuum】という機能が追加されたことによりロックを行わず【vacuum】の実行が可能となった。（注　だがこの【concurrent vacuum】は空き領域 を再利用するだけで，データファイルそのものは小さくならない）

【vacuum】機能の実行、未実行によるレスポンスの違いは第３章を参照していただきたい。