データベース設計と制約

森田 亙昭

平成 13 年 6 月 21 日

1　制約

制約とは、使用可能な値及び他のテーブルとの関係を定義する決まりのことです。リレーショナルデータベースにおける制約には、以下のようなものがあります。

- 必須制約
- 一意制約
- ドメイン制約
- 参照制約

これらの制約は、リレーショナルデータベース内では常に満たされていなければいけません。もし満たしていないようなときはエラーが起こり、その時に行っていた処理は中断され、実行前の状態に戻されます。図 1、2、3 のようなテーブルを用いて、制約について説明します。

Customer(顧客)		
customerNo (顧客番号)	customerName (顧客名)	customerAddress (顧客住所)
a001a	morita	fukuoka
t430c	mine	nagasaki
n503i	kubo	fukuoka
i400u	ryu	kumamoto

図 1: Customer

1.1　必須制約

必須制約とは、ある指定した属性値に NIL(空値) を許さないという決まりです。たとえば主キーは、必ず空値以外のデータを持っていなければならないという点で、常に必須制約を満たしていると言えます。顧客を新たに追加しようとしたとき、必須制約のついていない項目 (顧客名) は何も指定しなくてもかまいませんが、主キーである顧客番号を指定せずに追加を行おうとするとエラーとなります。

Commodity (商品)		
commodityNo (商品番号)	commodityName (商品名)	unitPrice (単価)
A005	video	18000
B003	PS2	39500
C009	telephone	20000
D001	printer	17500
E002	stereo	50000

図 2: Commodity

Sales (売上)				
salesNo (売上番号)	soleCommodityNo (商品番号)	purchaserNo (顧客番号)	quantity (数量)	amountSold (金額)
0001	A005	a001a	1	18000
0002	B003	t430c	1	39500
0003	E002	t430c	1	50000
0004	B003	n503i	1	39500
0005	C009	n503i	1	20000
0006	D001	i400u	2	35000

図 3: Sales

1.2 一意制約

一意制約とは、複数のインスタンスが同一の属性値を持つことを許さないという決まりです。これも主キーに対して言える制約で、インスタンスを一意に識別するために、各テーブルにおける主キーである番号は、かならず他とは異なる値でないといけません。売上データを新たに追加した際、もし追加したデータの主キーである売上番号がすでに存在するならば、その追加作業は中止され、エラーが表示されます。

1.3 ドメイン制約

ドメインとは、領域とか範囲とか言う意味です。つまり、ドメイン制約とは、ある属性値がとりうる範囲に関する決まりです。例えば、商品テーブルにおける単価は、常識から考えても 1 円以上となるはずです。商品を新たに追加したり、変更するときに、単価を 0 円に設定したりするとエラーが起こり、その作業は実行されません。

1.4 参照制約

参照制約は、双方向の参照の一貫性に関する制約です。つまり、異なるテーブル間で、互いにデータを参照しあっているような項目 (売上の顧客番号と顧客の顧客番号などのような関係) において、矛盾を生じさせてはいけないというものです。例えば、売上の顧客番号は顧客の顧客番号を参照しています。つまり、顧客の顧客番号に変更や削除が行われた場合は、それを参照していた売上のほうの顧客番号にも同じ処理が行われないと行けません。また、新たに売上を追加しようとする際、顧客番号は必ず顧客に登録されているもでないといけません。これらの制約が存在しなければ、顧客が削除されたのに売上はそのままで、顧客が誰だか分からない売上記録が残ってしまったり、売上に対する顧客を間違えてしまったりと言った深刻な問題がおこります。

2 手続き属性

Jasmine のようなオブジェクトデータベースの他との違いとして、データの格納、検索だけでなく格納されたデータへの処理もデータベースによって行うことができるということがあります。これは手続き属性と呼ばれるデータベース内に定義される関数によって行います。手続き属性を使用することで以下のようなことが行えます。

- データの加工
- 冗長なデータの削除
- 外部データへのアクセス
- 内部的なデータ表現の隠蔽

このような手続き属性を使用することにより、複雑な制約などを Jasmine で実現することが可能となります。

2.1 手続き属性の実現

手続き属性を使用するには以下の手順が必要になります

1. 手続き属性の定義
2. 手続きのコーディング
3. コンパイル

2.1.1 手続き属性の定義

手続き属性の定義は、以前説明したクラスへのデータ定義と同様に、ODQL を用いてクラス内に定義します。例えば、Customer クラスに新たなオブジェクトを生成する addCustomer という手続きを定義したい場合は、クラス定義ファイルを以下のように記述します

```
defineClass Customer
super:Composite
```

```
 description:"Class Customer "
{
 instance:
  Integer customerNo;
  String customerName;
  String  customerAddress;
  Void addCustomer(Integer i,String n,String a); /*手続き属性*/
 };
```

Void addCustomer(Integer i,String n,String a) が手続き属性の定義になります。ここでは手続き addCustomer は引数として i,n,a の 3 つを受け取り、値を返さない (void 型) ものとして定義されています。

2.1.2 手続きのコーディング

クラスに定義した手続きが、実際に存在しないことには意味がありません。というわけで、定義した手続き属性を作成します。addCustomer の内容は以下のようになっています

```
defineProcedure Void Customer::instance:addCustomer(Integer i,String n,String a)
language :"c"
{
  $defaultCF moritaCF;
  $Customer Cu;
  $Customer.new(customerNo:= i,customerName:= n,customerAddress := a);
  $return;
};
```

defineProcedure という ODQL コマンドが、手続きの内容を定義するコマンドで、それ以降が実際の処理内容になります。ここでは、手続きの引数として与えられた値を受け取り、その値を new() に渡すことで新たなオブジェクトを生成しています。このファイルを execFile コマンドで実行することにより、手続き addCustomer の処理内容がクラス Customer に定義されます

2.1.3 コンパイル

定義した手続きはコンパイルしないと使えません。コンパイルには compileProcedure を使います compileProcedure はクラス単位でのコンパイルを行います。例えばクラス Customer に addCustomer,deleteCustomer という 2 つの手続きが定義されている時、compileProcedure Customer; と入力することでこの 2 つのコンパイルが行われます。

2.1.4 手続き使用

コンパイルが通れば、無事に手続きが使用出来るようになります Customer に定義した addCustomer を使用し、新たなデータを登録するときにはインタプリタ上で以下のように入力します

```
Customer cu;
```

```
cu.addCustomer(1,morita,Fukuoka)
```

すると、customerNo が 1、customerName が morita、customerAddress が Fukuoka という新たな Customer オブジェクトが生成されるはずです。

3 制約の実現

先ほどあげた制約は、基本的にリレーショナルデータベースについてのものなので、オブジェクトデータベースである Jasmine でこれらの制約を実現するためには多少の工夫が必要になります。

3.1 必須制約の実現

必須制約を Jasmine で実現するには、データ定義の際に「mandatory」という記述を付け加えます。たとえば、Commodity の CommodityNo に必須制約を適用したい場合は定義ファイルにおいて

```
instance:
Integer       commodityNo     mandatory:;
```

のように、書けば OK です。こうしておけば、CommodityNo に何も値が指定されなかったときに、データベース側がエラーを通知してくるようになります。

3.2 一意制約の実現

一意制約も、必須制約と同じようにデータ定義ファイルに宣言を追加することで実現します。一意制約を適用したい場合は

```
instance:
Integer       commodityNo     unique:;
```

のようにします。必須制約と一意制約は、同時に適用させることも可能です

```
instance:
Integer       commodityNo     unique: mandatory:;
```

これにより、CommodityNo は必須であり一意である、まさに主キーとしての性質を持つことが可能になります。

3.3 ドメイン制約の実現

ドメイン制約を実現するためには、手続き属性を使用します以下に示した、Commodity にデータを追加する手続きの中でに、unitPrice が 1 円以上というドメイン制約を実現するためのコードが記述してあります。

```
defineProcedure Void Commodity::instance:addCommodity(Integer i,String n,Integer u,String a )
language :"c"
```

```
{
  $defaultCF moritaCF;

 /*Domain 制約*/
 /*u(unitPrice<1 なら、エラーを返す)*/
  $if(u<1)
 {
  odbGenerateUserMtdError("moritaCF","Commodity","addCommodity","Domain error wrong unitPrice");
  ODB_ERRORRETURN(-10);
  };
 /*それ以外なら、データを追加*/
  $Commodity.new(commodityNo:= i,commodityName:= n,unitPrice := u,commodityAddress := a);
  $return;
};
```

この手続きでは、引数として受け取った追加する unitPrice の値 u が 1 より小さい場合にエラーを発生させ、それ以外の場合はデータの追加を行います。このような手続きを使用することで、データ追加の際にドメイン制約を常に満たすように監視するということができるようになります。

3.4 参照制約の実現

参照制約の実現にも、手続き属性を使用します。例えば以下の手続きでは,指定された CustomerNo を持つオブジェクトを削除し、そのオブジェクトを参照していた Sales のオブジェクトを削除することで参照制約を満たします。

```
defineProcedure Void Customer::instance:deleteCustomer(Integer dn)
language :"c"
{
  $defaultCF moritaCF;
  $Bag<Customer> cus;
  $Customer cu;
  $Bag<Sales> sas;
  $Sales sa;
  $cus = Customer from Customer where Customer.customerNo == dn;
  /*入力された CustomerNo のオブジェクトが存在しなければ、エラーを返す*/
  $if (cus.count() == 0){
   odbGenerateUserMtdError("moritaCF","Customer","deleteCustomer","customerNo NOT FOUND");
   ODB_ERRORRETURN(-10);
  };
  /*オブジェクトが存在すれば、それを削除*/
  $cu = Customer from Customer where Customer.customerNo == dn;
  $cu.delete();

  /*削除されたオブジェクトを参照していた Sales の削除*/
```

```
 $sas = Sales from Sales where Sales.soldCommodityNo == dn;
 $if (sas.count() == 1){
  $sa = Sales from Sales where Sales.soldCommodityNo == dn;
  $sa.delete();
 };

 $return;
};
```

4 課題

以下の手順に従って、各制約を持つテーブルを作成し、実際に手続きを使用してみましょう。

1. 定義ファイルを持ってくる

   ```
   cp -r /u/morita/DBfile ~
   ```

2. ファイルの修正

   ```
   cd ~/DBfile
   ```
   DBfile にある add******.odql,delete******.odql(6 つ) を
   エディタで開き、defaultcf moritaCF; の moritaCF を自分の CF に修正

3. インタプリタ起動

   ```
   ssh jasmine
   cd  ~/DBfile
   bash
   source /jasmine/jasmine2/.profile
   codqlie
   ```

4. CF の変更

   ```
   jasmine(systemCF) > defaultCF xxxxCF;
   ```
   (自分の CF に変更)

5. クラス定義

   ```
   jasmine(xxxxCF) > execFile setClasses.odql;
   ```

6. クラス構築

   ```
   jasmine(xxxxCF) > buildClass Customer Commodity Sales;
   ```

7. 手続き定義各制約を含んだデータ追加、削除の手続きを定義する。

```
jasmine(xxxxCF) > execFile addCustomer.odql;
jasmine(xxxxCF) > execFile deleteCustomer.odql;
jasmine(xxxxCF) > execFile addCommodity.odql;
jasmine(xxxxCF) > execFile deleteCommodity.odql;
jasmine(xxxxCF) > execFile addSales.odql;
jasmine(xxxxCF) > execFile deleteSales.odql;
```

8. コンパイル

```
 compileProcedure Customer Commodity Sales;
```

9. 使用例

Commodity の追加、削除

```
jasmine(moritaCF) > Commodity co;
jasmine(moritaCF) > Bag<Commodity> cc;
jasmine(moritaCF) > co.addCommodity(1,"video",19000);
jasmine(moritaCF) > co.addCommodity(2,"TV",35000);
jasmine(moritaCF) > cc = Commodity from Commodity;
jasmine(moritaCF) > cc.print();
Bag{
    moritaCF::Commodity::2 {
        commodityNo = 1,
        commodityName = "video",
        unitPrice = 19000
    },
    moritaCF::Commodity::3 {
        commodityNo = 2,
        commodityName = "TV",
        unitPrice = 35000
    }
}
jasmine(moritaCF) > co.deleteCommodity(1);
jasmine(moritaCF) > cc.print();
Bag{
    <<<deleted-instance>>>,
    moritaCF::Commodity::3 {
        commodityNo = 2,
        commodityName = "TV",
        unitPrice = 35000
    }
}
```

手続きの引数になにを入れたらいいか分からないときは手続きのソースを見るか、インタプリタ上でクラス名.print() とするとクラスの定義内容が見られるのでそこで調べましょう。