计算机科学1996 Vol.23 №.3
48-51

在C++中实现多版本永久类

张柳青 戚成功 胡美琛 楼荣生
(复旦大学计算机科学系 上海200433)

摘 要 A Multi-version persist class is realized in C++. It can be persist stored and recovered, and so can all its subclasses. By this way we realize a main function of OODB.

关键词 Object-oriented database, Persist class, Inheritance, Aggregation, Multi-version.

实现面向对象数据库(OODB)的一个方法是在现有的关系数据库中,加入一个对象服务器,使其具有面向对象的特征。目前,很多商业化的OODB通过扩充某种OOPL而实现对对象的永久存储与检索。例如:GemStone与AllTalk扩充了Smalltalk;ORION与Statice扩充了LISP;Vbase,O++扩充C++等等。我们提出了另一种对象永久存储方法,并不扩充C++,而是在C++上实现了一种*PersistClass*类,它和所有它的子类的对象能永久保存和恢复。同时,利用C++语言重载能力和关系数据库便于集合操作的特性,将PersistClass类扩展为多版本类和多版本属性类,实现了OODB的版本功能。这样做的优点是:便于系统移植;便于C++程序员掌握;一定程度上克服了OODB与C++的阻抗失配,实现OODB和C++的无缝连接。

1 PersistClass类的定义与实现

1.1 PersistClass定义

PersistClass类是一个用C++实现的可以永久保存和恢复的类,其所有子类也有保存和恢复的功能。

PersistClass类有两个属性提供对象标识。公有属性Name是字符串型的对象唯一标识,比较直观和便于记忆,供用户使用。保护属性oid是整数型的对象唯一标识,系统对对象操作使用oid属性,可以提高速度。PersistClass有三个公有方法进行对象的永久保存与恢复,它们分别是:

1. Save:按用户指定的名字将对象加入数据库永久保存。当库中已存在用户指定的名字时,则更新库中的对象。若Save成功,返回1,否则返回0。

2. Recover:将指定名字的对象恢复回内存。若恢复成功,返回1。若不成功,或指定的名字不存在,返回0。

3. Remove:将当前对象从库中删除。若删除成功,返回1,若不成功,或当前对象不在库中,返回0。

上述三个方法实现了OODB的一个重要的功能,即对象的永久保存与恢复。PersistClass的定义如下:

```
class PersistClass{
protected:
  int oid;
  void AddAddress( );
  void AddValue( );
  void virtual DeclareValue( );
  void virtual DeclareAddress( );
public:
  char Name [MAX_OBJNAME_LEN];
  int Save(char * Name);
  int Recover(char * Name);
  int Remove(char * Name);
};
```

当保存对象时,系统需知道内存中对象的属性值,相反,当恢复对象时,系统需知道内存中对象的地址。保护方法AddValue( ),AddAddress( )提供这方面的功能,它们的实现对用户是透明的。

保护方法DeclareValue( )与DeclareAddress( )调用AddValue( )与AddAddress( ),将PersistClass的属性值或地址告诉系统。这两个方法是虚拟的,当其它对象继承了PersistClass时,需覆盖DeclareValue( )与DeclareAddress( )声明自己的属性值或地址。

方法DeclareValue( )与DeclareAddress( )非常简单,复杂的细节已尽量封装在AddValue( )与AddAddress( )中了。它们分别如下:

```
PersistClass::DeclareValue( ){
  AddValue(O_INT,oid);
  AddValue(O_STRING,Name);
}
```

```
PersistClass::DeclareAddress( ){
  AddAddress(O_INT,&oid);
  AddAddress(O_STRING,Name);
}
```

方法 Save 需将内存中的对象放入库中，因此将调用 DeclareValue 方法。方法 Recover 需将库中的对象放入内存中，因此将调用 DeclareAddress 方法。

```
PersistClass::Save(char *Name){
  if(库中有相同 Name 的对象)
    Remove(Name);
  插入准备工作;
  DeclareValue ( );
  插入收尾工作;
}
PersistClass::Recover(char *Name);
  if(库中无此 Name 的对象)
    return 0;
  恢复准备工作;
  DeclareAddress ( );
  恢复收尾工作;
}
```

方法 Remove 比较简单，仅删除库中名为 Name 的对象。

当其它类继承了 PersistClass 时，可以使用 PersistClass 公用的三个方法 Save，Recover，Remove 将自己类的对象永久保存与恢复。

1.2 继承

任何类只要继承了 PersistClass 或其子类，就具有永久保存与恢复的能力，但直正实现永久保存和恢复，用户还需做一些简单的工作。

下面以一个 Person 类为例，说明如何利用 PersistClass 将 Person 类对象永久保存与恢复。

```
class Person:public PersistClass{
public:
  char Name[10];
  int Sex;
};
```

Person 中有属性 Name，PersistClass 中也有属性 Name，这两个 Name 在 C++中是不同的，需分别保存。

首先，用户必需覆盖 PersistClass 的 DeclareValue( )与 DeclareAddress( )方法。任何类的 DeclareValue( )与 DeclareAddress( )都是相同的，即先调用其直接祖先的 DeclareValue( )或 DeclareAddress( )，然后根据自己的属性调用 AddValue( )或 AddAddress( )。

```
Person::DeclareValue( ){
  PersistClass::DeclareValue( );
  AddValue(O_STRING,Name);
  AddValue(O_STRING,Sex);
}
Person::DeclareAddress( ){
  PersistClass::DeclareAddress( );
  AddAddress(O_STRING,Name);
  AddAddress(O_STRING,&Sex);
}
```

第二，若 Person 类从未向系统注册过，需向系统注册，让系统准备相应的表空间存放对象。系统有函数 RegisterClass( )供用户调用。

1.3 数组，结构，联合

数组，结构和联合是 C++的重要数据类型，在关系库中没有相应的类型与之对应。仍以 Person 类为例：

```
class Person:public PersistClass{
protected:
  void virtual DeclareValue( );
  void virtual DeclareAddress( );
public:
  char Name[20];
  int Sex;
  struct PARENT{
    char Father[20];
    char Mother[20];
  }Parent;
};
```

这里，系统提供了两种永久保存 Person 类的 Parent 结构的方法。第一种是将 Parent 结构拆开存放，这时 Person 类的 DeclareValue( )与 DeclareAddress( )写法如下：

```
Person::DeclareValue( ){
  PersistClass::DeclareValue( );
  AddValue(O_STRING,Name);
  AddValue(O_STRING,Sex);
  AddValue(O_STRING,Parent.Father);
  AddValue(O_STRING,Parent.Mother);
}
Person::DeclareAddress( ){
  PersistClass::DeclareAddress( );
  AddAddress(O_STRING,Name);
  AddAddress(O_STRING,&Sex);
  AddAddress(O_STRING,Parent.Father);
  AddAddress(O_STRING,Parent.Mother);
}
```

同时，系统还提供了将数组、结构或联合作为一个二进制块存放的第二种方法。对于 Person 类的 Parent 结构，Person 类的 DeclareValue( )与 DeclareAddress( )写法如下：

```
Person::DeclareValue( ){
  PersistClass::DeclareValue( );
  AddValue(O_STRING,Name);
  AddValue(O_STRING,Sex);
  AddValue(O_BIN,&Parent,sizeof(Parent));
}
Person::DeclareAddress( ){
  PersistClass::DeclareAddress( );
  AddAddress(O_STRING,Name);
  AddAddress(O_STRING,&Sex);
  AddAddress(O_BIN,&Parent,sizeof(Parent));
}
```

拆开存放的方式比较适合于结构，而作为二进制块存放的方式适合于数组和结构。

1.4 聚合

对象的属性还可能是一个对象，这称为聚合。例如 Person 类的 Parent 中的 Father 和 Mother 就应是一个嵌入对象。本系统中，嵌入对象存放的是对象的 Name，因此，需引用嵌入对象时，要根据嵌入对象的 Name 从数据库中恢复一次对象。例如，需引用某

Person 的 Father 的属性时，在程序中应这样写：

```
...
Person person1,person2;
person1. Recover("somebody");
person2. Recover(person1. Parent. Father);
/ * now person2 is person1's father. * /
```

这样做的优点是只有当需引用对象的嵌入对象时，才将嵌入对象引入内存，从而节省了空间，时间。缺点是不能使用路径表达式，不符合一般引用嵌入对象的习惯。

2 多版本的实现

通常我们有下面三种版本要求：一种是历史版本，用来保存对象的历史信息；另一种是候选版本，保存对象在不同要求下不同侧重点的不同信息；另外还有一种叫修订版本，它考虑到对象记录本身和客观现实之间的差别，由物理的记录和记录的更改所形成的版本，其中最常遇见的是历史版本，本文以历史版本的实现为例说明，不难推及其余版本。

实现对象的多版本，有两种方法：一是多版本对象法，对象的每个版本都是一个全新的完整对象，但是共用同一个 OID，由版本属性标识其版本。二是多版属性法，对象只有一份，但是其基本类型的属性是多版本类的。第一种方法反复存放一些不变的属性，而第二种方法要求对象的每个属性都有一套版本标识和对象标识，开销也相当大，两者各有优劣。下面将分别说明这两种方法的实现。

2.1 时间戳方法

这里的历史多版本实现，类似于数据库并发控制中使用的多版本时间戳方法。首先为对象的每个版本数据扣上一个时间戳记，这个戳记是个随时间永远增加的数值，并且和时间一一对应。这样，既可以反应出历史版本之间的时间顺序关系，又可以根据给定的时间(时刻)，确定版本的可见度。版本的可见度是指：当对象的一个版本在某时间以前存在，并且是这些存在版本中最后的版本，也就是在该版本生成时刻和某时刻之间没有生成该对象的其它版本，则该版本相对这一时刻是可见的，否则不可见。

这里的时间戳是一个整型值(32位)，是通过调用 time( )获得的。为了处理系统的时间戳和直观的时间，我们定义下列全局函数：Now( )，TimeStamp( )等。作好了时间戳的准备工作，下面介绍两种具体的实现。

2.2 多版本对象

多版本对象的处理方法，要继承 PersistClass 类定义 MVPersistClass，首先增加一个时间戳属性 ts，使所有永久对象成为带有时间戳的版本对象。其中 Save( )根据时间戳参数，考虑对象的 OID 适当地生成对象的新版本，Remove( )删除指定版本的对象，Recover( )取出相对时间戳的可见版本。

2.3 多版本属性

采用多版本属性处理方法，要为 C++ 中对象的几种基本属性创建多版本属性类，比如：int，char *，等等。

2.3.1 多版本属性类。这里以 int 类型的多版本属性类 MV_int 的实现为代表，来说明怎样构造多版本属性类。程序如下：

```
class MV_int{
    int  value;
    int  oid;
    int  time_stamp;//以上三值用于内存操作
  public:
    SetOID(int);//设置 oid 值，以便下标操作时使用
    SetVersion(int time_stamp);
    int GetVersion( );//提供时间戳的保护性操作
    Save ( );//保存三属性到 mv_int 表中，用于更新和生
      成新版本
    int &operator[ ](int);//提供访问多版本整型的重要
      手段
    void Remove( );//以 oid 和 time_stamp 为依据删除
      一整型版本
}
```

对于这样一个多版本属性类，我们在底层数据库建一个 mv_int 表，存放 OID(oid)、TS(time_stamp)、Value(value)三个域，所有用 MV_int 说明的属性，全部放在该表中。对于用 MV_int 类说明属性的永久类，在进行永久类注册时，作为聚集换名保存其 oid，在永久对象恢复时，在 mv_int 表选出正确的对象。永久类进行恢复时，首先恢复非多版本属性，再依次调用 SetOID( )恢复多版本属性的 oid，使以后多版本引用时能找准对象。

2.3.2 重载下标操作符。下面整型引用的下标操作符重载提供访问多版本整型的重要手段，使得作为右值的下标引用能够从永久存储对象的多个版本中，选择一个合适的版本恢复到内存对象中。返回整型引用使其可以作为左值，应用时如同在操作一个整型数组的属性。

```
int  &MV_int :: operator[ ](int time_stamp)
{
  从 mv_int 表中查找 OID=oid & TS<=time_stamp 的
  最大 TS 的 Value 值，
  赋给 value;
  取 TS 值赋与 time_stamp;
  return value;
}
```

看起来，重载下标操作符以后，引用对象的多版本属性相当方便自然，但是特殊情况下，这样是不够的。当同一对象多版本属性的不同版本引用同时分别出现在一个赋值语句的左值和右值的时候，会引

起引用的混乱，因此，我们又重载了类的（ ）操作符。该重载类似于上述的下标重载，只是不用查找属性值，直接返回 value 的引用，可以在复杂的语句中使用作为左值。

2.4 两种多版本方法的综合

上述的两种实现多版本的方法，分别适用于不同特性的对象。比如，股票的价格这样的对象适合用多版本对象的方法，而病员（包括体温记录）这样的对象则更适合使用多版本属性的方法。

事实上两种方法在实现时是同时并存于一个系统中的，两者并没有抵触，也就是说，系统同时提供 MVPersistClass 和 PersistClass 以及各种基本属性的多版本类，应用可以按照自己的需要来选用，要注意的是，两种多版本的永久对象在具体的操作上是要区别对待的。

3 应用示例

假使我们要多次记录病人的体温，以备查询。先构造一病员类如下：

```
class    Patient:public Person{
    protected:
        void virtual DeclareValue( );
        void virtual DeclareAddress( );
    public:
        int No;
        MV_int Temp;
        Patient(char * str,int s,int n,int t){
          No=n;
          this->Temp.SetOID(OID(Temp));
          Temp[NULL]=t;
        }
    }
```

在完成 DeclareValue( ),DeclareAddress( )和 RegisterClass( )之后，下面一段 C++的应用完成记录"wang"病员体温的工作。

```
Patient wang("wang",0,17,37);
int     year,month,date,hour,t;
wang.Save( );
wang.Temp.Save( );
cout≪"请登记测量体温的时间(年  月  日  时):";
cin≫year≫month≫date≫hour;
for(;hour<=24;)
{
    cout≪"请输入"≪wang.name≪"的体温:";
    cin≫t;
    wang.Temp[TimeStamp(year,month,date,hour,0,
        0)]=t;
    wang.Temp.Save( );
    cout≪"请登记测量体温的时间(年  月  日  时):";
    cin≫year≫month≫date≫hour;
}
```

下面的 C++过程用来查询"wang"病人的体温情况：

```
cout≪"您需要"≪wang.name≪"什么时间(年  月  日
  时)的体温:";
cin≫year≫month≫date≫hour;
for(;hour<=24;){
  cout≪wang.name;

  int i=wang.time_stamp;
  cout≪"在"≪Year(i)≪"年"≪Month(i)≪"月"
      ≪Date(i)≪"日"≪Hour(i)≪"点的体温是:"
  cout≪wang.Temp[TimeStamp(year,month,date,hour,
    0,0)];
  cout≪"您还需要"≪wang.name≪"什么时间(年  月
    日  时)的体温:";
  cin≫year≫month≫date≫hour;
}
```

本文讨论了在 C++中实现多版本 OODB 的核心，无需任何元级机制来支持，使多版本 OODB 的操纵和 C++语言融为一体。在多版本永久类或者多版本属性类中增加一些时态处理方法，就可以使系统具有时态数据库的特征和功能，这样的功能设施，既是版本 OODB 应用编程的界面，也是进一步完成面向对象查询语言时需要的元级机制。

参 考 文 献

[1] W. Kim, Introduction to Object-Oriented Database, the MIT Press, London, England

[2] Edward Sciore, Using Annotations to Support *Multiple Kinds of Versioning in an Object-Oriented* Database System, ACM Trans. Database System Vol 16, No. 3. 1991

[3] 张成洪、周傲英、徐涌、施伯乐，面向对象数据库系统 FOODB 的数据模型，第十一届全国数据库学术会议论文，1993年9月

（上接第61页）

所示，在图2中同时列出了国外一 GIS 软件在同等条件下的刷新时间。从图2可以看出，在查询窗口小于64%时，增强型 BANG 文件的刷新时间比 GIS 软件快，特别是在小窗口查询时，刷新时间要快得多，这是由于增强型 BANG 文件所需的磁盘 I/O 次数少，但大窗口刷新时，性能相差不大。

从上面还可以得出：数据桶小，则点查询时间短，小窗口刷新时间快，但数据的冗余高，所占用栅格多，随着查询窗口的增大，由于磁盘 I/O 次数增加，其性能变差，数据桶大的增强型 BANG 文件，数据冗余相对小，中等查询窗口的性能好。

参 考 文 献

[1] 詹舒波、张其善，电子地图数据库存储文件的设计，计算机科学（本期）

[2] mapinfo 参考手册

[3] 李文通等，Borland C++3.1，北京航空航天出版社