第7卷第2期 软件学报 Vol. 7, No. 2
1996年2月 JOURNAL OF SOFTWARE Feb. 1996

C++对象的持久化中的问题和解决方案*

陶 伟 麦中凡

(北京航空航天大学计算机科学与工程系 北京 100083)

摘要 多媒体智能数据库系统MIDS(multimedia intelligent database system)是一个对象数据库管理系统.它的数据库编程语言是P++,P++语言是一种基于C++的语言.作者在P++的实现过程中遇到了以下问题:首先,C++的指针有二义性,它无法在语义上区分成员指针和引用指针,以及易失性指针和持久性指针,从而给事务管理中的对象加锁及其它方面带来问题.其次,具有虚拟函数或虚拟基类的对象中含有指向内存中的指针,而这些指针不是由程序员定义的.如果C++对象被持久化,这些指针在不同的程序调用中有可能无效.最后,如果作者用文件系统调用来存储对象,那么必须要设计复杂的Cache系统和做大量对象的格式转化工作,这需要大量的空间和时间,所以他们采用了另外的一种方法——基于虚拟内存空间映射的存储方案.

关键词 面向对象数据库,持久性,虚拟内存映射.

P++[1]是基于C++的数据库编程语言,是C++语言的一个超集.P++语言提供了对象模式定义、持久化、迭代、查询、版本、约束、触发器、规则和知识表示等重要的语言功能,是我们正在开发的多媒体智能数据库(MIDS/BUAA)[2]的重要的组成部分.

现在我们已经完成了P++的原型开发工作.该原型系统可以提供持久性对象的创建和操作,对象的簇聚(clustering)、查询和约束等功能.P++程序由P++编译器翻译成纯的C++程序语言,这些C++程序通过调用系统的对象管理库来实现P++的语义,然后通过C++编译进行编译和链接,以形成最终的可执行代码(见图1).

从P++程序员的角度,持久性对象的指针和易失性对象的指针在语义上是基本一致的,持久性对象的读取对用户透明,持久性对象和易失性对象无"语义断层"(sematic gap),同时P++语言在程序设计语言和数据库之间建立了一个统一的类型系统,使两种语言之间没有明显的断层,大大缓解了"阻抗失配"(impedance mismatch).[3]为了达到这些目的,C++对象持久化的实现者必须解决2个核心问题,即C++对象指针的二义性和C++对象内部的隐含指针(hidden pointer)和显式指针(explicit pointer)的重定位.

所谓的C++指针的二义性是指C++的指针在语义上不能区分成员指针(member pointer)和引用指针(reference pointer),以及易失性指针(volatile pointer)和持久性指针

---

* 作者陶伟,1970年生,博士,主要研究领域为面向对象数据库,面向对象程序语言.麦中凡,1935年生,教授,主要研究领域为程序设计语言,软件工程,人工智能,数据库.

本文通讯联系人:陶伟,北京 100083,北京航空航天大学计算机科学与工程系

本文1994-12-29收到修改稿

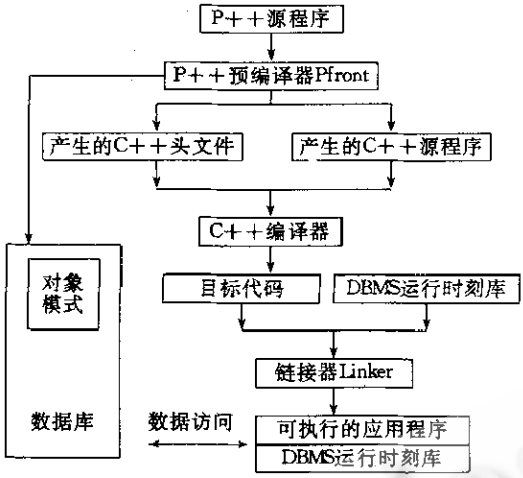

图1 MIDS P++语言编译过程

(persistant pointer). 所谓C++对象内部的"隐含指针"是指具有虚函数(virtual function)和虚基类(virtual base class)的对象中指向内存空间的指针(memory pointer). 之所以称之为"隐含指针",是因为这些指针在P++和C++中对用户是不可见的,是C++编译器在编译过程中加进去的. 在有虚函数的情况下,隐含指针指向一个虚拟函数表VFT (virtual function table)[4],用于动态地确定对象的方法调用束定到哪一个函数上. 在有虚基类的情况下,隐含指针用于共享基类. 所谓的显式指针是指用户定义的指针.

本文将紧紧围绕上述两个问题给出深入的分析和有效的解决方案. 此外我们还将讨论C++对象持久化的另一个问题:即C++中允许基类的指针指向派生类对象[5],那么在面向对象数据库中一个指向持久的基类对象的指针指向一个持久的派生类的对象指针,系统必须做相应的处理. 本文最后讨论基于虚拟空间映射(virtual memory mapping)的对象的存储方法.

1 C++指针的二义性和增强的C++对象模型

C++对象指针二义性是C++对象持久化中较难解决的问题之一. C++指针是指向内存空间的地址,仅在程序的运行时期内有效,持久性对象调入内存后,其内部指针的值必须要做调整. C++的对象指针有二义性,这表现在无法区分引用指针和成员指针,以及易失性指针和持久性指针. 作为成员指针指向的对象无OID(object identifier),它是复杂对象的组成部分,而引用指针反映的是对象实体之间的引用关系(被引用对象不属于引用对象,删除一个对象不能删除其引用对象),数据库系统必须维护其引用关系的完整性. 作为持久性指针,系统必须负责对象的读写,以实现其持久性语义.

由于C++指针的二义性给面向对象数据库带来了3个方面的问题,第1:持久性对象的删除,由于系统无法区分成员指针和引用指针,从而无法自动地、在较高抽象层次上删除对象,用户必须自己区分引用指针和成员指针,并提供相应的删除程序;第2:无法实现基于对象粒度的并发控制,在Client/Server体系结构下,当Client进程向Server进程申请对象时,由于指针有二义性,Server是不可能把对象拆成一维线性的网络数据报,传送到Client进程中,再将对象重构出来. 此外,在对象粒度上加锁,也要区分成员对象和引用对象,否则要么加锁不足,要么加锁过度,影响事务的并行处理. 第3:不区分易失性指针和持久性指针,我们就无法做到所谓持久性三原则[18]的第3条准则:持久性对象和易失性对象的操作一致性,即要求声明不同,但操作相同.

针对C++指针的二义性,大多数ODBMS系统都采取了相应的补救措施,例如:ONTOS、ObjectStore、GemStore等.

ONTOS[6]把对持久性对象的引用分为直接引用(direct reference)和抽象引用(abstract reference).直接引用实际上是C++的指针,引用完整性完全由程序员自己负责;抽象引用是ONTOS提供的一种更可靠、功能更强的引用机制,它可以跨越数据库,对象的激活和读入内存均对用户透明,它以一个Reference类的形式提供给用户,但它不是类型化的.ONTOS的抽象引用实际上要完成以下工作[17]:持久性对象的激活(activated)读入内存,所有对该持久性对象的引用均需调整成内存格式引用,当持久性对象被写回数据库(deactivated),则需要把内存格式引用转换成UID.可见其效率较低.

ObjectStore[7,11],为了解决对象引用,一方面,引入reference_base类,用它取代C++的指针,它比C++的指针的尺寸要大,通常是8～24字节,它可以跨跃数据库和事务进行对象引用;另一方面,引入两元关系的定义,这种关系是用集合实现的,该集合中的成员是指针,可以维护逆向数据成员的完整性.由于ObjectStore是采用基于虚拟内存空间映射的存储方案,其持久性指针和易失性指针的完全一样,因此,它的所谓reference主要是完成对象引用的完整性维护,它并没有解决引用指针和成员指针的区分问题.

GemStore[8]把指针分为GPointer和DPointer(direct pointer),GPointer是对象的标识,通过hash表进行追踪(dereference).DPointer直接指向虚拟空间中的对象的地址.GPointer使用时需转换成DPointer.GemStore的方案和ONTOS类似.

总之,一般的解决方案是:在利用对象标识(OID)作为引用对象手段的同时,引入一种所谓的Reference类,重载(overload)其>、*、=、==等操作,使之和易失性指针有相同的特性,但内部却隐含着持久性对象的读入和写出,以及引用的完整性维护.

以上方案只能是权宜之计,要从根本上解决问题,还需从对象的模型入手,采用一种增强的C++对象模型,这种增强模型的最大特点是把关系(Relationship)引入对象模型,其大致的形式如图2所示.

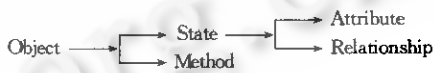

图2 增强的C++对象模型(Object Model)

- 基本的建模单位只有对象(object).
- 对象可归于类(class).给定类的所有对象均有共同的方法(method)和共同的状态(state).
- 对象的行为可由一组方法(method)定义.
- 对象的状态是由对象的属性(attributes)以及该对象和其它对象的关系(relationships)定义,属性可以是值(value)或对象(object).

我们的增强的C++对象模型和对象数据库管理组织ODMG(object database management group)在ODMG－93中给出的对象模型(object model)[9]有类似之处,ODMG－93的对象模型实际上也是对OMG(object management group)对象模型[10]做了扩展,引入了关系的概念,但它的属性(attribute)只能是值(value)不能是对象(object).

关系实际上是对象之间的一种联系,是指针的一种表现形式.在C++中实现关系时要引入一个inverse子句用于定义了一组相关对象的类型和用于引用相关对象的跨连路径(traversal paths),关系本身是通过嵌入对象内部的引用(1:1关系)或集合实现的,见图3中的示例.关系可以是1:1,1:n,n:m 3种.例如图3中的例子:Professor对Department是1:1,Professor对学生是1:n,学生对课程是n:m.数据库系统要维护这些关系的引用的完整性.如果关系中的一个对象被删除,则逆关系中的对该对象的引用将导致异常,但是有时

为了满足C++编程的灵活性,也允许一种降低形式的关系,即单向引用,不过在这种引用中,逆向子句(inverse clause)将省略,关系的完整性由程序员自己维护.

```
class Professor {
      public:
          Reference<Department> dept
            inverse Department::professors;
          Set<Student> advisees
            inverse Student::advisor;
}
class Course {
      public:
         Set<Student> students—enrolled
               inverse Student::classes ;
}
```

```
class Student {
      public:
          Reference<Professor> advisor
          inverse Professor::advisees;
        Set<Course> classes
          inverse Course::students—enrooled;
}
```

图3 增强C++对象模型中关系定义的示例

我们在C++这个层次上,为了实现P++的语义,在对象模型上采用了增强的C++对象模型,加入了关系扩充,对成员指针和引用指针做了区分,凡是作为关系的指针即为引用指针,其余的指针均为成员指针.引用指针在许多方面和C++指针是类似的,但它有保证持久性对象引用的完整性的机制.我们用一个模板类template <class T> class Reference实现了引用指针,它对*、&、[]、=、==等操作符做了重载,其外观用法和C++指针几乎一样.为了向ODMG-93靠拢,参照ODMG-93中的定义,该类可以设计成图4中的模式.

```
template <class T> class Reference {
   public :
     Reference<T>(const T * ) ;      // constructor
     Reference<T>(const Reference<T> &) ;      // constructor
     Reference<T> & operator =(const T * ) ;      // the T won't change
     Reference<T> & operator =(const Reference<T>&) ;      // rvalue doesn't change
     T * operator ->() ;      // dereference the reference
     const T * operator->() ;      // pointer you get back is a const
     operator T * () ;      // applies to something that is non—const
     operator const T * () ;      // applies to something that is a const
     ......
     boolean operator == (const Reference<T>&) const ;
     boolean operator == (consst T * ) const ;
     const T&operator * () const ;
     T&operator[](int) ;      // array behavior
     virtual ~Reference<T> ;      // destructor
     ......
   }
```

图4 Reference类的实现

对于P++程序中的关系,P++预编译器在预编译时要加入保持一致性的控制代码,这些代码的实质工作是维护表达关系的双向指针.当程序中有删除对象操作时,P++预编译器也会加入一些代码,以删除成员指针所指的对象.

2 C++对象的虚拟函数和虚拟函数表

C++的对象和其编译后的相应的对象在布局(layout)上是不同的,特别是具有继承关系的类对象,C++编译器在编译过程中会加入虚拟函数表和虚拟基类的指针.当调用虚拟函数时将通过使用vtbl指针来间接访问在虚拟函数表中的函数入口.当程序引用虚基类的

部分则必须使用虚基类指针(vbase pointer).我们把 vtbl 和 vbase 指针称为"隐含指针",因为它们是C++为了实现其面向对象范型而加入的信息,对用户是不可见的.C++程序员甚至不知道它们的存在.

由于隐含指针是易变指针,它们仅在创建它们的程序运行时有效.当把一个含有隐含指针的对象存入数据库,而另一个程序把该对象读出时,对象中的所有隐含指针均无效(实际上还包括引用指针和成员指针).这就要求面向对象数据库系统维护这些隐含指针的有效性.

为了更直观地分析C++对象的隐含指针 vtbl 和 vbase,我们引入4个类作为例子,分别是:Person,Student,Assistant,StudentAssistant,其中 Person 是基类,Student 和 Assistant 分别是 Person 的派生类,StudentAssitant 是 Student 和 Assistant 的派生类,它们的定义见图5.

```
class Person {
        public:
          char Name[32];
          char Sex;
          int Age;
          virtual void print (void) ;
};
class Assistant: virtual public Person {
        public:
          char Course[32];
          int CourseNumber;
          virtual void print(void);
};
void Person::print( void ){ ..... }
void Assistant::print( void ){ ..... }
```

```
class Student: virtual public Person {
        public:
          char Department[32];
          virtual void print (void);
};

class StudentAssistant: public Student,public Assistant {
        public:
            int Salary ;
            virtual void print(void);
};
void Student::print( void ) { ..... }
void StudentAssistant::print( void ) { ..... }
```

图5 四个示例类的定义

当我们使用 Sun SPARC C++ 3.1编译上述示例程序,我们可以得到编译器生成的中间.C 文件,如图6所示.从这个例子我们可以看出,任何一个具有虚拟函数的类的对象都含有一个隐含指针指向一个虚拟函数表(vtbl),例如:——vptr—6Person.vtbl 中含有虚拟函数的地址,例如:print—6PersonFv,同时它还含有一个偏移量即 struct ——mptr 中的 short d,该偏移量用于定位一个派生类对象中的基类子对象的地址.在图7的例子中,当把 AssisPtr赋给 PersonPtr 时,PersonPtr 被指向 Assistant 对象中的 Person 子对象;当 PersonPtr->print()被调用时,C++将通过 Person 子对象中的 vtbl 指针,间接地调用 Assistant::print().然而 Assistant::print()需要 Assistant 对象的地址作为其参数,该地址是从 PersonPtr 中减去 Assistant 和 Person 之间的偏移量获得的.,具体的实现过程见图8.

```
typedef int ( * ——vptp)(void);
struct ——mptr {short d; short i; ——vptp f; };
......
/* sizeof Person == 44 */
struct Person {
        char Name———6Person [32];
        char Sex———6Person ;
        int Age———6Person ;
        /* vtbl pointer */
        struct ——mptr * ——vptr——6Person;
};
```

```
/* sizeof Student == 84 */
struct Student {
    char Department——7Student [32];
  /* vtbl pointer */
  struct ——mptr * ——vptr———7Student;
/* vbase pointer */
  struct Person *PPerson;
    struct Person OPerson;
```

```
    /* sizeof Assistant==88 */
    struct Assistant {
            char Course——9Assistant [32];
            int CourseNumber——9Assistant;
            /* vtbl pointer */
            struct——mptr * ——vptr——9Assistant;
            /* vbase pointer */
            struct Person *PPerson;
            struct Person OPerson;
    };
```

```
};
/* sizeof StudentAssistant == 132 */
struct StudentAssistant {
    char Department——7Student [32];
  /* vtbl pointer */
  struct ——mptr * ——vptr——7Student ;
/* vbase pointer */
  struct Person *PPerson;
    struct Assistant OAssistant ;
    int Salary——16StudentAssistant;
};
```

```
……
void print——6PersonFv (register struct Person * ——0this ) { …… }
void print——7StudentFv (register struct Student * ——0this ) { …… }
void print——9AssistantFv (register struct Assistant * ——0this ) { …… }
void print——16StudentAssistantFv (register struct StudentAssistant * ——0this ) { …… }
struct ——mptr ——vtbl ——6Person ——16StudentAssistant [ ] = { 0,0,0, -84,0,(——vptp)print——16StudentAssistantFv,0,0,0} ;
struct ——mptr ——vtbl ——9Assistant ——16StudentAssistant [ ] = { 0,0,0, -40,0,(——vptp)print——16StudentAssistantFv,0,0,0};
struct ——mptr ——vtbl——16StudentAssistant[]={ 0,0,0, 0,0,(——vptp)print——16StudentAssistantFv,0,0,0};
struct ——mptr ——vtbl——6Person——9Assistant[]={ 0,0,0, -44,0,(——vptp)print——9AssistantFv,0,0,0};
struct ——mptr ——vtbl——9Assistant[]={ 0,0,0, 0,0,(——vptp)print——9AssistantFv,0,0,0};
struct ——mptr ——vtbl——6Person——7Student[]={ 0,0,0, -40,0,(——vptp)print——7StudentFv,0,0,0};
struct ——mptr ——vtbl——7Student[]={ 0,0,0, 0,0,(——vptp)print——7StudentFv,0,0,0};
struct ——mptr ——vtbl——6Person[]={ 0,0,0, 0,0,(——vptp)print——6PersonFv,0,0,0};
```

图6 四个示例类的定义在C语言中的形式

```
Person *PersonPtr = new Person ;
Assistant *AssisPtr = new Assistant ;
……
PersonPtr->print() ;
AssisPtr->print() ;
……
PersonPtr = AssisPtr ;
PersonPtr->print() ;
```

图7 派生类指针赋给基类的指针

```
struct Person *——1PersonPtr ;
struct Assistant *——1AssisPtr ;
……
((*(((void (*)(struct Person *——0this ))(——1PersonPtr ->——vptr——6Person [1]).f))))((struct Person *)((((char *)——1PersonPtr ))+ (——1PersonPtr ->——vptr——6Person [1]).d))) ;
((*(((void (*)(struct Assistant *——0this ))(——1AssisPtr ->——vptr——9Assistant [1]).f))))((struct Assistant *)((((char *)——1AssisPtr ))+ (——1AssisPtr ->——vptr——9Assistant [1]).d))) ;
——1PersonPtr = (((struct Person *)(——1AssisPtr ?——1AssisPtr ->PPerson:0 )));
((*(((void (*)(struct Person *——0this ))(——1PersonPtr ->——vptr——6Person [1]).f))))((struct Person *)((((char *)——1PersonPtr ))+ (——1PersonPtr ->——vptr——6Person [1]).d))) ;
```

图8 派生类指针赋给基类的指针的实现方法

3 持久性对象的隐含指针的重定位

隐含指针问题在ObjectStore[7],Vbase[12],E[13]语言的实现中都曾经是它们面临的重要问题之一.在ObjectStore中,它有一个模式获取工具Schema Generator[11],对用户的源程序进行扫描,并直接生成所有类型对象的内部指针布局模式,该布局模式存入两个地方,一

处是用户定义的 obj 文件,它将和应用程序链接;另一处是 ObjectStore 的数据库.由于对象内部的物理布局和具体的 C++编译器有关,因此 ObjectStore 或是知道它所支持的 C++编译所产生的对象的布局,或是它直接从 obj 文件中提取,否则别无它法.在读入对象时,ObjectStore 根据对象的物理模式信息,以递归的方式重定位对象的 vtbl 指针,并把 vbase 指针指向当前虚基类子对象的相应部分的地址.ObjectStore 的解决方案的特点是:直接、高效.缺点是依赖于特定 C++编译器产生的代码.Vbase 采用的方法是创建一个 vtbl 持久对象,E 语言的编译器 efront 则把虚基类的指针用一个偏移量取代,对于有虚函数的持久性对象则做上标记,根据标记产生 Hash 表,由系统运行时动态调用虚拟函数.

为了解决隐含指针的重定位问题我们尝试了2种解决方案.方案1:利用 C++编译器产生的.C 中间文件,直接提取对象的物理模式,形成相应的物理模式文件,一份存入数据库中,另一份连同指针重定位程序链接到应用程序中去.方案2:不直接访问对象中的隐含指针,而采用 C++本身的机制来恢复隐含指针的合法值.

采用方案1,我们必须分析 C++的编译器的编译结果,必须分析 C++转换成 C 的命名规则,然后用 LEX 和 YACC 生成对象的物理模式提取工具(实际上是从如图6的文件中提取).对象的物理模式提取工具生成两个结果,一是对象内部的指针布局表,另一个是指针的递归重定位程序,然后调用 C++编译器对其编译生成 obj 文件,链接到源应用程序中去.采用这种方案的优点是直接面对问题,解决矛盾.缺点是只能针对某一特定的 C++编译器(如我们使用的 Sun SPARC C++ 3.01).还有我们无法保证穷举出所有可能的C++编译结果,特别是在有优化的情况下,只能不断地在使用中发现一个问题,解决一个问题.因此,我们认为在以后的 C++编译器中应该提供对象的物理布局文件,以供用户使用.

采用方案2,我们将面临两大障碍:一是 C++不允许对一个已存在的 C++对象调用构造函数,二是我们无法直接调用用户定义的构造函数来重新初始化持久对象中的隐含指针,因为这些构造函数有可能修改对象的属性值.为了解决上述2个障碍我们必须做2件事:其一,重载一个新 new,但这个 new 不分配空间,仅作为调用构造函数来重定位隐含指针的一种手段,其定义如图9所示.

```
class —Nothing { } ;
void * operator new( size—t, —Nothing * p )
{ return (void * )p }
```

图9 重载 new 用以重定位隐含指针

其二,用上述 new 调用构造函数时,构造函数不能修改任何属性值,因此我们必须在 pfront 预处理时,修改用户定义的构造函数,使其在重定位调用时,不修改属性.解决方案是引入一个全程变量 Is—FixPointer—Now,当它为0是为正常调用,非0时为重定位调用,不修改属性.此外,还要为每一个类生成一个宏,用于持久性对象的指针重定位调用(图10).

```
extern int Is—FixPointer—Now = 0 ;
Person::Person() {
  if( Is—FixPointer—Now == 0 ) {
      ..... // Attribute initialize
  }
}
```

```
#define FixPointer## #Person( void * p ) \
  Is—FixPointer—Now = 1 ; \
  new( (—Nothing * )p) Person ; \
    Is—FixPointer—Now = 0 ; \
```

图10 隐含指针的重定位

4 基于虚拟内存映射的存储方法的关键问题

持久性对象的存储方法一般有3种.第1种,对象翻译法.数据库对象以磁盘格式存储,调入内存后翻译成内存格式,写回数据库再翻译成磁盘格式.该方法需要双缓冲区,即页面缓冲区和对象缓冲区.此方法的特点是实现技术简单,但系统复杂,其空间效率和时间效率较低,ORION[14]就采用此法.

第2种是基于磁盘的方法.对象在库中和内存中均以磁盘格式,但程序设计语言的对象既可以是内存格式,也可以是磁盘格式,但二者必须隔离.该方法和方案1类似,同样需要二级存储.采用这种方案的系统有Exodus,ODE,O2等.[15]

第3种方法是基于虚拟空间(virtual memory),所有的对象均以内存格式表达,对象中的指针亦是虚存地址.采用这种单一的存储结构,系统复杂性小,速度快,空间利用率高.越来越多的系统开始采用这种方法,如ObjectStore[7],Bubba[15],EPVM2.0[16]等.

我们采用的是第3种方法.我们重载了C++的运算符new和delete,通过重载的new和delete进行基于虚存空间映射的持久性对象的创建和删除.在Client/Server体系结构的背景下,采用这种方法所要解决的主要问题是:指针的重定位和如何发现缺页.因为数据库中的每个页面,在不同的应用程序中虚拟空间的地址可能不相同,则页面中的对象的指针,以及引用该页面的指针均需调整.此外,当指针追踪时,引用了Client进程尚未获得的页,则需要向Server发出请求,以获取所需页.

解决该问题的基本思路是:Client进程每获取一个页面,则重定位该页面中的所有指针.在用户实际使用某一页面之前就分配给其虚拟空间的地址,当程序第一次试图访问它时,操作系统发出缺页中断信号,而我们可以截取该信号,把所需要的页从Server进程取来,然后返回引起缺页中断的语句.

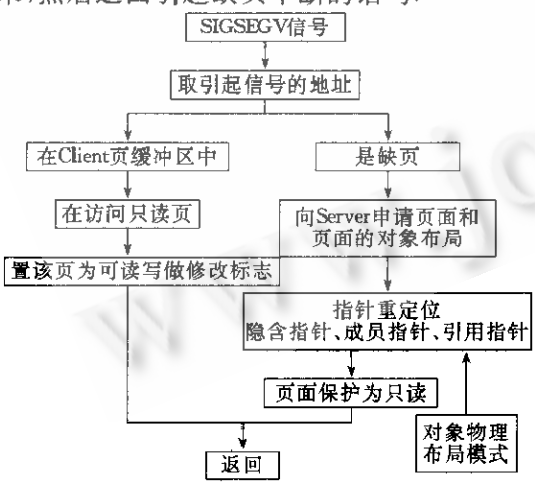

图11 SIGSEGV信号处理程序

在实现过程中有以下的关键点:

(1)数据库应分为若干段,每个段有若干页面构成,预分配地址,以段为单位.当页面内的指针不跨越该段时,只需将指针的值加一个偏移量;当跨越到另一个段时,如果这个段尚未分配给虚拟空间地址,则为该段分别虚拟空间地址,保护模式置为no access,然后在调整跨越段的指针值.

(2)页面内的指针重定位.我们必须要知道页面中含有那些对象以及各类对象的物理布局模式,然后才能对对象中隐含指针,成员指针,引用指针进行重定位.

(3)截取SIGSEGV信号以发现缺页中断和处理读写保护.SIGSEGV信号是UNIX提供的一个信号,当程序引用了不可读或不可访问的地址时便会发生.同时它会向信号处理程序提供引起该信号的地址引用.我们截取该信号主要做2个工作:其一,把原先只读页,置为可读写,为被修改的页做标记;其二,如果发现是缺页中断,则向Server发出页申请,获得页

后再做指针重定位工作,见图11.

因为篇幅的限制和为了说明问题更方便,我们把并发控制等方面的处理省略了.

5 结　论

C++对象的持久化的核心问题是隐含指针、成员指针、引用指针的重定位问题.解决的方案既可以直接提取对象的物理布局模式,对各种指针进行递归重定位,也可以在C++语言这一层次上,间接地重定位指针.但是最有效的方法是直接在C++编译中直接完成持久化工作,把C++演进成数据库编程语言,或者C++编译能够提供一定的对象的实现信息,当然这只能在未来完成.基于虚拟空间映射的存储方法有较高的时间和空间效率,实现的复杂性相对小,是ODBMS的较理想可选存储方案.

参考文献

1 车敦仁,麦中凡.MIDS/BUAA语言的设计与实现.西安:第11届全国数据库学术会议论文集,1993.

2 Che Dunren, Mai Zhongfan. The overall design of MIDS/BUAA: a multimedia intelligent database system. ICYCS'93, July 1993.

3 Copehand G, Maier D *et al*. Making smalltalk a database system. Proc. of the ACM SIGMOD'89, 1989.

4 Ellis M A, Stroustrup B. The annotated C++ reference manual. Addison—Wesley, 1990.

5 Stroustrup B. The C++ programming language, 2nd edition. Addison—Wesley, 1991.

6 ONTOS DB 2.2 First Time User's Guide. ONTOS, INC. February 1992.

7 Lamb C *et al*. The object store database system. Communications of the ACM, Oct. 1991, **34(10)**:50.

8 Butterworth Paul *et al*. The gemstone object database management system. Communications of the ACM, Oct. 1991, **34(10)**.

9 Cattel R G G. The object database standard: ODMG—93. Morgan Kanfmann Publishers, 1994.

10 OMG(Object Management Group). Object Management Architecture Guide 1.0. 1990.

11 ObjectStore user guide release 1.2.4 for microsoft windows™3.1. Object Design, Inc. 1992.

12 Vbase technical notes. ONTOS Inc, Burlington, MA, 1987.

13 Richardson J E, Carey M J. Persistence in the E language: issues and implementation. Software—Practice and Experience 19, 1989.

14 Kim Won, Ballou Nat *et al*. Integrating an object—oriented programming system with a database system. OOPSLA'88 proceedings, 1988.

15 Copeland George, Franklin Michael, Weikum Gerhard. Uniform object management. LNCS416:EDBT'90, 1990.

16 Seth J W, David J D. A performance study of alternative object faulting and pointer swizzling strategies. Proceedings of the 18th VLDB Conference, 1992.

17 ONTOS DB 2.2 Developer's Guide. ONTOS, Inc. July 1993.

18 Atkinson M P P *et al*. An approach to persistent programming. The Computer Journal, Nov. 1983.

MAKING C++ OBJECTS PERSISTENT: PROBLEMS AND SOLUTIONS

Tao Wei　Mai Zhongfan

(*Department of Computer Science and Engineering　Beijing University of Aeronautics and Astronautics　Beijing*　100083)

Abstract MIDS(mutilmedia intelligent database system) is an object database management system. Its database programming language is P++, which is based on C++. In the implementation of P++ the authors encountered the following problems: First, C++ pointer is ambiguous, not only can member pointer and reference pointer not be distinguished in semantics, but also volatile pointer and persistent pointer. This pointer ambiguity results in object locking problem in transaction management and others. Second, C++ objects that have virtual functions or virtual base classes contain memory pointers, which were not specified by the programmer. If such C++ objects are made persistent, then these pointers may be invalid across program invocations. Last, If the authors store object using file system call services, they should design complicated cache subsystem and a large amount of object format transformation, which demand a lot of memory and time. So they must find another solution——storage scheme based on virtual memory mapping.

Key words Object—oriented database, persistence, virtual memory mapping.