제20회 한국정보처리학회 추계학술발표대회 논문집 제10권 제2호 (2003. 11)

XML을 위한 Java 기반 객체 정보 저장소[1]

박상원
한국외국어대학교 컴퓨터및정보통신공학부
e-mail : swpark@hufs.ac.kr

Java Based Object Repository for XML

Sangwon Park
Computer Science & Information Communication Engineering Div.
Hankuk University of Foreign Studies

요 약

Java 의 사용 영역이 일반 응용 프로그램 뿐 아니라 가정용 기기에까지 널리 사용되고 있다. 이러한 환경에서 다량의 XML 데이터를 다룰 경우 제한된 메모리를 사용을 보장하면서 객체의 지속성(persistency)를 보장하는 시스템이 필요하다. 이에 본 논문에서는 순수 Java 객체의 지속성을 보장하는 시스템인 XFS 를 구현하였다. XFS 는 쓰레드 환경에서 뛰어난 동시성을 보장하며 임의의 Java 객체의 지속성을 보장하므로 확장성이 뛰어나다. 또한 물리적 모델과 논리적 모델을 분리하여 응용 프로그램을 독립성을 증가시켰다.

Key Words: XML, Object Repository, Java

1. 서론

Java 는 현재 일반 프로그래밍 뿐만 아니라 가전, 핸드폰과 같은 이동식 기계 등에 널리 사용되고 있는 객체지향 언어이다. 핸드폰에 Java 가상 기계를 설치하고 게임이나 스케줄 관리 등과 같은 응용 프로그램이 Java 로 작성되고 있다. 또한 VOD 에서 셋탑 박스(set top box)를 이용하여 주문형 비디오를 시청할 때 전달되는 데이터는 MPEG7 과 같이 XML 형태로 데이터가 전달되게 될 것이다. 이와 같이 Java 를 사용하게 될 환경은 앞으로 지속적으로 증대될 것으로 생각된다.

셋탑박스나 핸드폰, 디지탈 TV 와 같은 환경에서 Java 의 사용이 증대되고 다양한 응용 프로그램이 제공될 것이다. 이때 Java 객체의 지속성(persistency)을 제공할 필요가 있다. 스트림으로 제공되는 MPEG7 과 같은 XML 데이터는 한번만 사용되는 것이 아니라 비디오 시청이 끝날 때까지 필요하다. 그러므로 이러한 데이터를 유지할 필요가 있다. 하지만 이를 메모리에 유지하자면 너무 많은 메모리가 필요하게 된다. 예를 들어 XML 응용 프로그램을 작성하려면 DOM[1]을 이용해야 한다. 하지만 XML 을 파싱하여 DOM 으로 변환하게 되면 DOM 자료구조가 메모리에 모두 적재된다. 셋탑 박스나 가전도구와 등은 메모리가 한정되어 있으므로 한정된 메모리 환경에서 임의의 크기의 지속성 객체를 다룰 수 있는 환경이 필요하다.

이러한 문제를 해결하기 위하여 본 논문에서는 XFS(*XWEET File System*)을 제안하였다. XFS 는 순수 Java 를 이용하여 만든 객체 정보 저장소로서 임의의 Java 객체를 저장소에 저장하고 객체 캐쉬를 통하여 성능을 보장한다. 순수 Java 로 구현되었기 때문에 이식성이 뛰어나며 기존 데이터베이스와 다르게 light weight 한 특징을 가지므로 이동형 기계나 셋탑 박스 등에 적용하기가 용이하다. 객체 캐쉬는 캐쉬를 이용하여 디스크에 저장되어 있는 객체를 캐쉬의 크기만큼 메모리에 올렸다가 내리는 역할을 수행하기 때문에 객체 캐쉬를 이용하여 메모리 크기를 한정하는 문제를 해결할 수 있다.

하지만 Java 는 메모리 할당은 사용자가 원하는 순간에 할 수 있으나 메모리에서 객체를 제거하는 것은 백그라운드로 수행되고 있는 Java garbage collector 가 이를 수행한다. 그러데 Java garbage collector 가 언제 객체를 메모리에서 제거할 지는 알 수 없게 되므로 일정한 메모리 환경을 보장할 수 없게 된다. 본 논문에서는 물리적 객체와 논리적 객체를 통하여 이러한

[1] 본 연구는 2003 학년도 한국외국어대학교 교원연구비의 지원에 의하여 이루어진 것임

문제를 해결한다.

C++ 클래스로 정의된 객체를 디스크에 저장하는 객체지향 데이터베이스에서는 관계형 데이터베이스에서 지원하는 물리적 수준과 개념적 수준을 분리하는 것과 같은 기능을 제공하지 못하는 경향이 있다[2]. 왜냐 하면 C++ 클래스는 public 부분의

함수와 private 부분의 변수가 같은 클래스에 존재하게 된다. 여기서 public 부분의 함수는 그 클래스를 접근하는 인터페이스라는 의미가 있고 private 부분의 변수는 그 객체의 구현의 의미가 있다. 그런데 클래스는 의미적으로 인터페이스와 그 인터페이스를 구현하는 코드가 한 곳에 같이 있게되므로 물리적, 논리적 개념을 분리하기가 힘들게 된다. 하지만 XFS 에서는 물리적 객체와 논리적 객체를 이용하여 물리적 모델과 논리적 모델을 분리할 수 있는 장점을 얻을 수 있다.

2. 객체 캐쉬

XFS 에서 Java 객체는 정보 저장소에 저장되며 이러한 객체의 구분은 그 객체의 OID[3]로 구분된다. OID 는 객체가 저장된 물리적인 위치 정보를 이용하여 만드는 물리적 OID 를 사용하였다. 객체에 대한 참조 연산은 객체 캐쉬의 인터페이스를 이용하여 수행된다. 본 장에서는 객체 캐쉬의 디자인에 대하여 살펴본다.

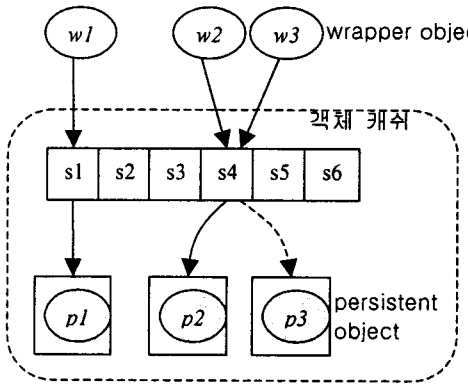

그림 1: 객체 캐쉬의 구조

그림 1 은 객체 캐쉬에 대한 것으로 p1, p2, p3 는 지속성 객체를 가리킨다. 이러한 지속성 객체들은 디스크에 저장되어 있던 것을 객체 캐쉬가 메모리로 읽어들인 경우이다. 이때 이러한 지속성 객체의 갯수를 객체 캐쉬의 캐쉬 엔트리 갯수만큼 유지하여 제한된 메모리 환경에서 다량의 지속성 객체를 처리할 수 있다.

하지만 Java 를 사용하는 환경에서는 이러한 시스템을 구축하기가 쉽지 않다. Java 의 경우 메모리 할당은 사용자가 마음대로 할 수 있으나 객체를 메모리에서 제거하는 것은 마음대로 할 수 없다. 객체를 메모리에서 제거하는 일은 백그라운드로 수행되는 Java garbage collector 가 담당하며 언제 메모리 제거가 일어나는지 알 수 없다.

```
a = p2;
```

위 코드는 사용자 변수 a 가 지속성 객체 p1 을 가리키는 경우이다. 이때 객체 캐쉬가 새로운 지속성 객체를 메모리에 적재하기 위하여 LRU 방법으로 p2 객체를 victim 으로 선택하였을 경우 객체 엔트리 s4 는 더 이상 지속성 객체 p2 를 가리키지 않고 p3 를 가리키게 된다. 하지만 이 객체는 garbage 가 되지 않는다. 왜냐하면 사용자 변수 a 가 여전히 p2 객체를 가리키기 때문이다. 또한 이러한 경우는 논리적 모순이 발생한다. 객체 캐쉬의 입장에서는 객체 p2 가 더 이상 객체 캐쉬 내에 존재하지 않기 때문에 또 다시 객체 p2 를 접근하고자 할 경우 디스크에서 p2 객체를 다시 읽어와 캐쉬 엔트리가 p2 객체를 가리키게 된다. 이러한 경우 메모리에는 두개의 p2 객체가 존재하게 된다. 객체 엔트리가 가리키는 p2 객체와 사용자 변수 a 가 가리키는 p2 객체로 인하여 이상 현상이 발생하게 되는 것이다.

이러한 문제를 해결하기 위하여 XFS 에서는 두 종류의 객체인 지속성 객체와 랩퍼 객체(wrapper object)를 두었다. 지속성 객체는 실제 디스크에 저장되어야 할 객체로서 객체 캐쉬가 관리하게 된다. 이러한 지속성 객체를 사용자 변수가 직접 가리키게 되면 garbage collection 대상이 되지 못하기 때문에 일정한 메모리 사용을 보장할 수 없을 뿐만 아니라 앞에서 언급한 바와 같이 이상 현상이 발생하게 된다. 그러므로 사용자 변수는 지속성 객체를 직접 가리키는 것이 아니라 랩퍼 객체를 가리키게 된다.

```
a = w2;
```

위 코드와 같이 사용자 변수는 랩퍼 객체 w3 를 가리키며 이러한 랩퍼 객체는 내부에 지속성 객체에 대한 OID 를 가지고 있다. 지속성 객체는 OID 로서 각 객체를 구분하게 되는데 랩퍼 객체가 가리키고 있는 지속성 객체가 어떠한 것인지를 알기 위하여 랩퍼 객체는 단순히 자신이 랩핑하고 있는 지속성 객체의 OID 만 가지고 있는 것이다.

만약 지속성 객체 p2 가 객체 캐쉬에서 제거될 경우 p2 를 가리키는 변수는 객체 엔트리 밖에 없으므로 자연스럽게 p2 객체는 garbage 가 되어 garbage collection 의 대상이 된다. 이때 랩퍼 객체 w2 를 통하여 지속성 객체를 접근하게 되면 w2 가 가리키는 엔트리 s4 에 p2 객체가 없다는 것을 알 수 있게 되어 객체 캐쉬에게 새로이 p2 객체를 요구하도록 하여 이상 현상을 막는다. 이때 랩퍼 객체를 사용자 변수가 직접 가리키게 되므로 랩퍼 객체는 객체 캐쉬에서 통제할 수 없게 된다. 이러한 랩퍼 객체를 garbage 로 만들어 메모리 사용량을 적절히 조절하는 책임은 프로그래머에게 있다. 하지만 랩퍼 객체는 그 자체에 지속성 객체를 가리키는 OID 밖에 없으므로 랩퍼 객체는

전체 메모리 사용량을 증가시키는데 지속성 객체에 비하여 크게 영향을 미치지 않는다고 볼 수 있다.

3. PObject 와 Wrapper

XFS 에는 지속성 객체를 위하여 PObject 클래스와 랩퍼 객체를 위하여 Wrapper 클래스가 있다. 만약 사용자가 Person 클래스를 정의하고 이를 지속성을 가지도록 하고 싶으면 Person 클래스의 상위 클래스를 PObject 로 하면 된다. 즉 PObject 하위 클래스로 정의하면 그 클래스의 인스턴스는 지속성 객체가 되어 디스크에 자동으로 저장된다. 사용자는 지속성 클래스를 정의하면 반드시 그 지속성 클래스에 대응하는 랩퍼 클래스를 정의해야 한다. 이때 랩퍼 클래스의 이름은 지속성 클래스의 이름에 Wrapper 를 덧붙인 것이 된다. 즉 PersonWrapper 클래스가 Person 클래스의 랩퍼 클래스가 된다. 이것은 객체 캐쉬에 대하여 랩퍼 객체를 요청하면 객체 캐쉬가 지속성 객체의 타입 정보를 실행시에 알아내어 지속성 객체의 랩퍼 타입의 객체를 생성하여 그 객체를 반환하기 때문이다. 다음은 지속성 객체를 생성하고 이를 얻어내는 코드이다.

```
public class Person extends PObject  {
  // persistent code
}

public class WPerson extends Wrapper  {
     // logic code
}

ObjectCache oc =
        ObjectCache.getObjectCache();
PObject p = new Person();
int oid = objectCache.createObject(p);
WPerson w = (WPerson)w.getWrapper(oid);
```

위에서 보는 바와 같이 지속성 객체를 생성한 다음 이를 디스크에 저장하고, 이러한 지속성 객체의 랩퍼 객체를 얻어 변수 w 가 이를 가리키는 것을 볼 수 있다. 이때 지속성 클래스에 정의할 수 있는 변수는 기본 타입(primitive type) 혹은 String 타입이거나 이들의 배열이어야 한다. 이와 다르게 객체를 내부 변수로 선언하고 싶으면 그 객체도 지속성 객체로 정의하고 이 객체의 OID 만 내부 변수로 가지고 있어야 한다.

4. 로직 객체와 지속성 객체의 분리

클래스는 객체지향 데이터베이스의 스키마를 의미한다. 데이터베이스는 논리적 데이터 모델과 물리적인 모델을 구분하여 만들어진다. 그 이유는 물리적 모델이 바뀌더라도 논리적인 모델은 영향을 받지 않아서 응용 프로그램의 수정이 필요없게 되기 때문에 개발하거나 유지, 보수하는데 드는 비용을 줄일 수 있기 때문이다. 그런데 객체지향 데이터베이스에서의 스키마에 해당하는 클래스에는 이러한 논리적인 모델과 물리적인 모델이 하나의 구조 안에 같이 있게 되는 문제점이 있는 것이다.

이러한 문제를 해결하기 위하여 본 논문에서는 클래스를 크게 논리적인 클래스인 Wrapper 와 물리적인 클래스인 PObject 로 구분하였다. 사용자 응용 프로그램은 논리적 클래스를 이용하여 작성하게 되므로 실제 저장하는 것과 관련있는 물리적인 클래스의 내용이 변경되더라도 사용자 응용 프로그램은 이 영향을 받지않게 되는 것이다. 즉 논리적 클래스에는 그 클래스의 논리적인 구현 코드가 있게 된다. 또한 논리적 클래스의 구현조차 Java 의 interface 를 이용하여 논리적 클래스의 구현과 인터페이스를 완벽하게 분리할 수 있다. 물리적 클래스는 실제 저장하는 데이터와 관련있는 것으로 실제 디스크에 저장되는 클래스에 해당한다.

예를 들어 사용자가 클래스를 PObject 로 부터 계승받아 저장 클래스 Person 클래스를 만든 후 객체를 생성하면 XFS 는 이를 디스크에 자동으로 객체를 저장한다. 이때 객체 캐쉬의 인터페이스인 getWrapper 함수를 이용하면 사용자가 만든 클래스의 랩퍼 클래스 PersonWrapper 객체에 대한 레퍼런스 값이 반환된다. 그러므로 사용자는 Person 클래스에는 사람에 대하여 저장할 때 필요한 변수를 선언하고 PersonWrapper 클래스에는 사람에 관한 로직에 해당하는 여러 함수를 만든다. 이렇게 하여 저장 포맷인 Person 과 이에 대한 로직인 PersonWrapper 를 통하여 물리적인 모델과 논리적인 모델을 분리할 수 있다.

5. 확장성

XFS 는 PObject 타입의 객체이면 저장할 수 있기 때문에 확장성이 뛰어나다. XML 데이터는 DOM 으로 표현되는데 이러한 DOM 구현을 XFS 를 이용하여 구현함으로써 DOM 객체를 객체 정보 저장소에 저장하는 시스템인 XDOM[4]을 구현하였다. 또한 XML 은 링크의 시맨틱을 확장하기 위하여 XLink 를 제안하고 있다. 이러한 XLink 를 XDOM 에 구현하여 XFS 의 뛰어난 확장성을 검증하였다. 이러한 확장성이 PDOM[5]과 가장 구별되는 점이다.

5.1 XDOM

XDOM 은 XFS 를 이용하여 DOM 인터페이스를 구현한 클래스 라이브러리이다. DOM 객체에 대하여 지속성을 줌으로써 XML 데이터를 처리할 때 DOM 인터페이스를 이용하여 데이터를 접근하더라도 일정한 메모리 사용량을 보장할 수 있게 해준다. 이러한 DOM 인터페이스를 이용하여 임의의 XML 문서를 저장하며 DOM 인터페이스를 이용한 일반 응용 프로그램과의 연동을 쉽게 해준다.

XDOM 에서 실제 저장되는 부분과 관련있는 것은

지속성 클래스로 정의하였고, DOM 인터페이스의 구현은 랩퍼 클래스를 이용하여 정의하였다. 즉 저장 데이터의 구조가 변경되더라도 이를 접근하는 랩퍼 클래스의 인터페이스는 변함이 없으므로 저장 모델과 로직이 분리되는 효과를 얻을 수 있다. 아래는 이에 대한 예를 보여준다. 이 예에서 함수 getNodeName 은 지속성 객체에서 값을 읽어서 사용자에게 전달해 주는 함수이다.

```
public class Node extends PObject {
   int parentOID;
   String name;
   String value;
}

public class WNode extends Wrapper {
   public String getNodeName() {
      Node node = (Node)pinPObject();
      String str = node.name;
      unpinPObject();
      return str;
   }
}
```

XFS 는 사용자 변수가 가리키는 객체가 랩퍼 객체이고 이 랩퍼 객체는 단지 지속성 객체를 가리키는 OID 만 가지고 있으므로 그 크기가 매우 작다. 그러므로 사용자 변수가 이러한 랩퍼 객체의 레퍼런스 값을 유지하고 있더라도 필요한 힙의 크기는 크게 증가하지 않는다.

5.2 XLink

HTML 은 링크의 의미가 다른 문서를 가리키는 것으로 매우 단순하다. 이러한 링크에 많은 의미를 부여하여 링크의 사용을 좀 더 확장한 것이 XML Linking Language[6]이다. 이에 XDOM 에서 XLink 를 지원하도록

확장하였다. 이것은 객체 정보 저장소에서 무결성(integrity)를 정의하고 이를 XML 의 의미와 부합되는 것들로 재 정의한 것이다. 이러한 시스템을 구현하는데 쉽게 구현할 수 있는 이유는 XFS 가 확장성이 뛰어나기 때문이다. 이러한 뛰어난 확장성으로 인하여 여러 가지 응용 프로그램을 작성하여도 각 응용 프로그램에서 만들어진 객체의 지속성을 보장할 수 있게 되는 것이다.

6. 결론

본 논문에서는 순수 Java 언어를 이용하여 객체 정보 저장소를 설계하였다. 이러한 객체 정보 저장소는 임의의 Java 객체를 저장할 수 있기 때문에 확장성이 뛰어나다. 또한 제한된 메모리 사용을 보장하기 위하여 지속성 객체와 랩퍼 객체로 분리하여 객체 캐쉬에서는 지속성 객체를 관리하고 사용자 변수는 랩퍼 객체만 다루도록 하였다. 즉 응용 프로그램에서 접근하는 객체는 랩퍼 객체이며 이러한 랩퍼 객체는 OID 를 통하여 지속성 객체에 대하여 접근하게 된다. 이러한 랩퍼 객체는 시스템에서 그 생명 주기(life time)을 조절할 수 없지만 그 객체의 크기가 OID 의 크기이므로 메모리 내에 다수가 만들어진다 하더라도 사용 메모리의 증가는 그리 크지 않게 된다.

또한 지속성 객체와 랩퍼 객체를 통하여 물리적 모델과 논리적 모델을 분리할 수 있게 되어 응용 프로그램의 확장성을 증가시켰다. 즉 지속성 클래스가 변경되더라도 그 객체의 인터페이스에 해당하는 랩퍼 클래스를 사용하는 응용 프로그램은 변경되지 않아도 된다. 또한 인터페이스를 구성하는 랩퍼 클래스가 변경되더라도 이에 대응하는 지속성 클래스의 내용은 변경되지 않아도 된다. 이러한 구현과 인터페이스의 분리를 통한 물리적 모델과 논리적 모델의 분리는 데이터베이스 응용 프로그램의 생산성을 향상하며 더 나은 디자인 환경을 제공한다.

또한 동시성을 증가시키기 위하여 사용자 모드와 캐쉬 모드를 두어 한 쓰레드가 장기간 객체 캐쉬에 대한 배타적인 락을 획득하고 있지 못하게 하였다. 즉 한 쓰레드가 전역 자료구조인 객체 캐쉬에 대한 접근의 독점을 방지하여 여러 쓰레드가 동시에 동작하는 환경에서도 성능을 각 쓰레드가 균등한 권한을 갖고 동작할 수 있도록 하였다.

현재 시스템에서 사용하고 있는 객체 교체 기법은 LRU 를 기본으로 하고 있다. 하지만 XML 데이터와 같이 특정 모델의 데이터를 처리할 경우 이러한 객체 교체 기법을 모델에 맞게 변경하면 성능을 향상 시킬 수 있을 것이다. 또한 현재는 객체 단위로 압축하여 저장하는데 더 큰 압축 효과를 얻기 위해서는 페이지 단위로 압축하는 것이 필요하다.

참고문헌

[1] W3C, Document Object Model (DOM), *http://www.w3.org/DOM*, 2000

[2] 조은선, 김형주, C++-based OODBPL 에서 클래스의 분리를 지원하기 위한 구현 기법, 한국정보과학회 논문지(B), 26(6), Jun. 1999

[3] Won Kim, *Introduction to Object-Oriented Databases*, The MIT Press, 1990

[4] 오동일, 최일환, 박상원, 김형주, XDOM: 확장성 기반의 경량 XML 객체 정보 저장소, 정보과학회 논문지: 컴퓨팅의 실제, 9(3), Jun. 2003

[5] Gerald Huck, Ingo Macherius, and Peter Fankhauser, PDOM: Lightweight Persistent Support for the Document Object Model, *OOPSLA*, Nov. 1999

[6] W3C, XML Linking Language, *http://www.w3c.org/XML/Linking*, 2001