Java 객체 직렬화에 대해 당신이 모르고 있던 5 가지

직렬화된 데이터가 안전하다고? 다시 생각해보라.

저자: Ted Neward, Principal, Neward & Associates
작성일: 2010 년 4 월 26 일
대상: 초급 개발자
번역: 김형석 수석
번역일: 2013 년 03 년 23 일
원문 주소: http://www.ibm.com/developerworks/java/library/j-5things1/index.html

몇 년 전에 Java 언어로 어플리케이션을 만드는 팀에 속한 적이 있었는데, 그 때 보통 개발자들이라면 잘 알지 못했을 Java 객체 직렬화의 멋진 기능들을 경험해 본 적이 있다. 그 뒤로 작년쯤인가, Java 어플리케이션을 만드는 팀에서 한 개발자가, 어플리케이션 환경설정 내용을 Hashtable에 저장한 뒤, 그 내용을 직렬화하여 디스크에 파일로 저장하기로 했다는 얘기를 들었다. 물론 환경설정 내용을 바꾸고 저장하면 그 내용이 파일에 다시 저장될 것이었다.

이렇게 하는 것이 처음에는 매우 세련되고 개방된 환경설정 처리 방식으로 보였지만, Hashtable 말고 Java 컬렉션 라이브러리인 HashMap[1]을 사용하기로 개발팀에서 결정한 뒤에는 이러한 생각이 산산이 부서져 버렸다.

문제는 파일로 저장된 Hashtable의 형식이 HashMap과 호환되지 않는다는 생각 때문이었다. 이 때문에 결국 데이터를 변환하는 유틸리티를 만들어서 사용하게 되었는데(정말 기념비적인 작업이었다), 어쨌든 이 어플리케이션의 환경설정 파일은 평생 Hashtable을 이용할 수 밖에 없을 것처럼 보였다.

개발자들은 막막한 느낌을 받았었지만 사실 그것은 Java 객체 직렬화에 대해 중요한(그리고 약간 잘 알려지지 않은) 부분에 대해 모르고 있어서 그랬던 것이다. 즉, Java 객체

---

[1] 예전에 Hashtable은 Java 컬렉션 프레임워크(Java Collection Framework, JCF)에 포함되지 않았었다. 정확히 말하면, JCF가 JavaSDK에 포함된 것은 1.2부터의 일이고 Hashtable이 JCF에 편입된 것은 JDK1.3 때의 일이다. (Hashtable은 JDK1.1때부터 존재했다.)

직렬화가 직렬화된 유형(Type)의 변경을 허용한다는 점이었다. 필자가 그 방법을 알려준 뒤로 HashMap으로 전환하는 작업은 계획되었던 대로 진행될 수 있었다.

이 문서는 Java에서 사소하지만 유용한 내용을 알려주는 시리즈[2]의 첫 번째 문서이다. Java로 프로그래밍을 하는 과정에서 마주치게 되는 문제를 해결하는 데 도움이 되는 잘 알려지지 않은 것들을 다룬다는 말이다.

Java 객체 직렬화는 Java의 시작, JDK1.1부터 제공된 멋진 API이다. 이제 소개될 객체 직렬화에 대한 다섯 가지 내용을 배우고 나면 기본 Java API들을 다시 한 번 살펴보아야겠다는 생각이 들게 될 것이다.

Java Serialization 101[3]

Java 객체 직렬화는 JDK1.1 때부터 제공된 엄청난 기능으로, Java 인스턴스를 디스크에 저장하거나 네트워크로 전송하기 위해 바이트 배열로 전환하고, 또 역으로 그렇게 저장/전송된 바이트 배열을 다시 Java 인스턴스로 전환하는 기술이다.[4]

본질적으로, 직렬화라는 개념은 객체 그래프를 얼린(freeze) 뒤, 디스크나 네트워크 같은 매체로 이동하고, 이동된 정보를 다시 객체 그래프로 해동(thaw)하는 과정을 의미한다. 이 모든 과정은 ObjectInputStream/ObjectOutputStream과, 신뢰할 수 있는 메타데이터, 그리고 직렬화하려는 클래스에 Serializable 인터페이스를 구현하도록 한 프로그래머의 의지에 의해 마술같이 처리된다.

코드1에서 Serializable 인터페이스를 구현한 Person 클래스를 확인할 수 있다.

코드 1. Serializable Person

```
package com.tedneward;

public class Person
   implements java.io.Serializable
```

---

[2] IBM DeveloperWorks에는 "5 things that you didn't know about ....." 문서가 시리즈로 몇 개 있다. 이 문서는 시리즈 중 첫 번째로 작성된 문서이다.

[3] 101(one-o-one)은 1장 1절을 의미한다. 가장 기초적인 내용을 다룰 때 101이라는 표현을 사용한다.

[4] 직렬화(Serialization)와 역직렬화(Deserialization)는 항상 쌍으로 생각되어야 한다.

```
{
   public Person(String fn, String ln, int a)
   {
      this.firstName = fn; this.lastName = ln; this.age = a;
   }

   public String getFirstName() { return firstName; }
   public String getLastName() { return lastName; }
   public int getAge() { return age; }
   public Person getSpouse() { return spouse; }

   public void setFirstName(String value) { firstName = value; }
   public void setLastName(String value) { lastName = value; }
   public void setAge(int value) { age = value; }
   public void setSpouse(Person value) { spouse = value; }

   public String toString()
   {
      return "[Person: firstName=" + firstName +
         " lastName=" + lastName +
         " age=" + age +
         " spouse=" + spouse.getFirstName() +
         "]";
   }

   private String firstName;
   private String lastName;
   private int age;
   private Person spouse;

}
```

Person 인스턴스를 직렬화하였다면 아주 간단한 작업으로 디스크에 쓰거나 디스크에서 다시 읽어올 수 있다. 코드2에서 이에 대한 JUnit4 유닛 테스트 코드를 확인할 수 있다.

코드 2. Deserializing Person

```
public class SerTest
{
   @Test public void serializeToDisk()
   {
      try
      {
         com.tedneward.Person ted = new com.tedneward.Person("Ted", "Neward", 39);
         com.tedneward.Person charl = new com.tedneward.Person("Charlotte",
            "Neward", 38);

         ted.setSpouse(charl); charl.setSpouse(ted);

         FileOutputStream fos = new FileOutputStream("tempdata.ser");
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(ted);
         oos.close();
      }
      catch (Exception ex)
      {
         fail("Exception thrown during test: " + ex.toString());
      }

      try
      {
         FileInputStream fis = new FileInputStream("tempdata.ser");
         ObjectInputStream ois = new ObjectInputStream(fis);
         com.tedneward.Person ted = (com.tedneward.Person) ois.readObject();
         ois.close();

         assertEquals(ted.getFirstName(), "Ted");
         assertEquals(ted.getSpouse().getFirstName(), "Charlotte");

         // Clean up the file
         new File("tempdata.ser").delete();
      }
```

```
        catch (Exception ex)
        {
            fail("Exception thrown during test: " + ex.toString());
        }
    }
}
```

정말 기초적인 내용인 관계로, 지금까지는 어떤 것도 새롭거나 대단할 것이 없어 보일 것이다. 이 내용을 먼저 말하는 이유는, 독자들이 '아마도' 몰랐을 다섯 가지 내용을 다루는 데에 이 Person 클래스를 이용할 것이기 때문이다.

1. 직렬화는 리팩토링[5]을 허용한다.

직렬화는 클래스가 어느 정도 변경되거나, 심지어 리팩토링이 되어도 괜찮은데, 웬만하면 ObjectInputStream이 적절하게 처리해 주기 때문이다.

Java 객체 직렬화 규격이 자동적으로 처리할 수 있는 변경사항들은 다음과 같다.

- 클래스에 새로운 필드 추가
- static 필드를 non-static 필드로 변경하기
- transient 필드를 non-transient 필드로 변경하기

이것 외의 변경, 즉 non-static을 static으로 바꾸거나 non-transient를 transient로 바꾸거나, 존재하던 필드를 제거하는 등의 작업은 하위 호환성을 위해 추가적인 작업을 수행해야 한다.

직렬화된 클래스를 리팩토링하기

직렬화가 리팩토링을 지원한다고 하였으니, Person 클래스에 새로운 필드를 추가하기로 했을 때 어떤 일이 일어나는지 확인해보도록 하자.

코드3에서 확인할 수 있겠지만, Person 버전2는 원래 클래스에 gender라고 하는 필드를 한 것이다.

Listing 3. Adding a new field to serialized Person

```
enum Gender
{
    MALE, FEMALE
}

public class Person
```

[5] 리팩토링이란 코드의 기능은 변경하지 않고 코드의 구조를 변경하는 작업을 말한다. 일반적으로는 좀 더 바람직한 방향으로 변경하는 작업, 즉 추상화 구조를 더한다던가, 메소드를 작게 만든다던가, 클래스를 분할한다던가 하는 등의 작업을 의미하긴 하지만, 파일명이나 변수명을 바꾸는 단순한 변경 작업을 의미하기도 한다.

```
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a, Gender g)
    {
        this.firstName = fn; this.lastName = ln; this.age = a; this.gender = g;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public Gender getGender() { return gender; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setGender(Gender value) { gender = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " gender=" + gender +
            " age=" + age +
            " spouse=" + spouse.getFirstName() +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
    private Gender gender;
}
```

직렬화는 메소드 명, 필드명, 필드 유형, 접근지시자, 파일명 등 직렬화하려는 소스 파일에 대한 모든 내용에 기반하여 계산된 해시(hash)값을 이용하고, 이 해시값과 직렬화된 스트림의 해시값을 비교하여 처리한다.

Java 런타임이 두 가지 클래스의 유형[6]이 결과적으로 동일한지 알려면, 이 두 번째 (PersonV2) 혹은 그 이후 버전을 Person 클래스들이 원래의 Person 클래스와 동일한 직렬화 버전 해시값이 적용되어 있어야 한다. 그리고 이 값은 클래스 내에 private static final long serialVersionUID 필드에 저장되도록 되어 있다. 그러므로 우리가 해야 할 일은, serialVersionUID 필드를 선언하고 거기에 특정 해시값을 입력하는 것인데, 이 값은 JDK의 serialver 명령어를 이용하여 얻을 수 있다.[7]

serialVersionUID 값을 얻었고 그 값을 클래스에 적용하였다면, 기존 Person 객체가 직렬화되어 있다고 하더라도 PersonV2 객체로 역직렬화를 수행할 수 있을 뿐만 아니라 반대

---

[6] 직렬화되어 있는 클래스와 역직렬화할 클래스

[7] 요즘은 이클립스 IDE를 이용하여 자동 생성할 수 있다. serialver 명령어에 대한 상세한 내용은 http://docs.oracle.com/javase/1.4.2/docs/tooldocs/solaris/serialver.html를 참고하라.

로 직렬화된 PersonV2 객체에서 Person 객체로 역직렬화를 수행할 수도 있게 된다.

2. 직렬화는 안전하지 않다.

가끔 Java 개발자들이 직렬화된 이진(binary) 형식의 데이터가 완벽하게 문서화되어 있고, 그것을 바탕으로 온전한 객체를 만들 수 있다는 것에 깜짝 놀라거나 불편해하는 것을 본 적이 있다. 실제로, 직렬화된 이진 스트림을 콘솔창에 출력하는 것 만으로 클래스의 형태나 저장되어 있는 데이터의 형태를 확인할 수 있다.

이 말은 (직렬화된 데이터가) 보안적으로 문제가 있을 수 있음을 의미한다. RMI[8]를 이용하여 원격의 메소드를 호출할 경우, private으로 지정된 필드 내의 값이 평범한 텍스트 형태 그대로 소켓 스트립에 전송되는 것을 볼 수 있는데, 이는 분명히 가장 간단한 보안 규정을 위반한 하는 것이다.

다행히도 Java 객체 직렬화 기능은 직렬화 과정을 가로채어[9] 직렬화하거나 역직렬화하는 과정에서 특정 필드를 암호화할 수 있도록 하는 기능을 제공한다. 아래에서 설명하겠지만, 직렬화할 객체에 writeObject 메소드를 추가함으로써 이 처리를 수행할 수 있다.

직렬화된 데이터 감추기

우리의 Person 클래스에서 매우 민감한 데이터가 age 필드에 저장된다고 가정해보자. 뭐 어쨌든 어떤 여성도 자신의 나이를 함부로 말하지 않고 어떤 남자도 함부로 물어보지 않으니 그냥 그렇다고 하자. 우리는 이 값을 직렬화하기 전에 간단한 비트 연산을 이용하여 감출 수 있는데, 나중에 역직렬화할 때에는 비트연산을 거꾸로 하면 된다. (물론 정말 보안을 위한다면 제대로 된 암호화 알고리즘을 이용해야 하겠지만, 여기서는 그냥 하나의 예일 뿐이라고 생각하자.)

직렬화 과정을 가로채기 위해, writeObject 메소드를, 그리고 역직렬화 과정을 가로채려면 readObject 메소드를 각각 구현할 것이다. 이 메소드를 구현할 때 지정된 방식대로 구현하는 것이 중요한데, 접근지시자, 파라미터, 메소드명 등이 다를 경우에는 정상적으로 처리되지 않을 것이고 그럴 경우 age 필드의 값이 예전과 같이 그대로 노출될 것이다.

---

[8] Remote Method Invocation. Java가 객체 직렬화를 이용하여 내/외부 시스템 간 연동을 수행할 수 있도록 제안된 연동방식이다. EJB도 내부적으로 RMI를 사용하고 있다.

[9] Hooking은, 일반적으로 수행되는 프로세스를 강제로 가로채어 프로세스가 동작하는 방식을 변경하는 것을 의미한다. 어떤 의미에서 볼 때, AOP도 hooking의 일종이라고 할 수 있다.

코드 4. Obscuring serialized data

```
public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    private void writeObject(java.io.ObjectOutputStream stream)
        throws java.io.IOException
    {
        // "Encrypt"/obscure the sensitive data
        age = age >> 2;
        stream.defaultWriteObject();
    }

    private void readObject(java.io.ObjectInputStream stream)
        throws java.io.IOException, ClassNotFoundException
    {
        stream.defaultReadObject();

        // "Decrypt"/de-obscure the sensitive data
        age = age << 2;
    }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " age=" + age +
            " spouse=" + (spouse!=null ? spouse.getFirstName() : "[null]") +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
}
```

변경된 데이터를 확인하고 싶다면 파일 등에 저장된 직렬화된 데이터를 확인하면 된다. 그리고 직렬화 데이터의 포맷은 완전히 문서화되어 공개되어 있으므로, 클래스 소스 없이도 직렬화된 스트림 데이터로부터 원하는 내용을 읽을 수 있다.

3. 직렬화된 데이터를 서명하거나 봉인할 수 있다.

위의 팁에서는 직렬화된 데이터를 암호화하거나 데이터가 변조를 막기 위해 어떤 작업을 하지는 않았고, 단지 노출되는 것을 막기만 하였다. 물론 writeObject나 readObject 메소드를 이용할 때 암호화 방식이나 서명관리 방식을 이용하여 데이터를 감출 수도 있겠지만, 이보다 더 좋은 방법이 있다.

만약 전체 객체를 암호화하거나 서명하기를 원한다면 그 객체를

javax.crypto.SealedObject 래퍼 클래스나 java.security.SignedObject 래퍼 클래스 안에 집어넣기만 하면 된다. 두 클래스 모두 직렬화가 가능하기 때문에 SealedObject 클래스를 이용하여 감싸는 것 만으로 원래 객체를 "선물상자" 안에 넣는 것 같은 효과를 얻을 수 있다. 이 과정에서 암호화 처리를 하기 위해 대칭키가 필요한데, 이 키는 각각 독립적으로 관리되어야 한다. 비슷한 방식으로 SignedObject를 이용하면 데이터에 대한 검증을 수행할 수 있는데, 마찬가지로 대칭키가 별개로 관리되어야 한다.

이 두 클래스를 같이 사용하면 암호화하거나 봉인하기 위해 큰 스트레스를 받지 않고 한꺼번에 데이터를 봉인하고 서명할 수도 있다. 괜찮지 않나?

4. Serialization can put a proxy in your stream

가끔은 클래스의 주요 정보들로부터 나머지 부가 정보들이 얻어질 수 있는 경우도 있다. 이럴 경우에는 클래스 전체를 직렬화할 필요가 없다. 물론 이런 부가 정보들을 transient로 처리할 수도 있지만, 그렇다고 하더라도 각각의 필드에 접근하는 메소드들은 그 필드에 접근할 때마다 해당 필드가 적절하게 초기화되었는지 명시적으로 확인하는 코드가 추가되어야 한다.

지금 얘기하고 있는 주제가 직렬화에 대한 것이므로, 이런 경우에는 경량 클래스(lightweight)나 프록시(proxy)를 이용하여 직렬화하는 것이 더 바람직할 수 있다. 원래 Person 클래스에 writeReplace 메소드를 구현함으로써 다른 종류의 클래스가 직렬화될 수 있도록 할 수 있다. 역으로, 역직렬화 과정에서 readResolve 메소드가 구현되어 있는 것이 확인되면, 이 메소드가 원래의 클래스 형태로 리턴되도록 할 것이다.

프록시를 이용하여 직렬화/역직렬화하기

위에서 설명한 대로, writeReplace 메소드와 readResolve 메소드를 같이 이용하면 Person 클래스의 데이터 중의 일부를 PersonProxy 클래스에 입력하여 직렬화하고, 나중에 역직렬화할 때 다시 되돌이킬 수 있다.

코드 5. 프록시를 이용한 생성과 변경

```
class PersonProxy
    implements java.io.Serializable
{
    public PersonProxy(Person orig)
    {
        data = orig.getFirstName() + "," + orig.getLastName() + "," + orig.getAge();
        if (orig.getSpouse() != null)
        {
            Person spouse = orig.getSpouse();
            data = data + "," + spouse.getFirstName() + "," + spouse.getLastName() + ","
              + spouse.getAge();
        }
    }

    public String data;
```

```
    private Object readResolve()
        throws java.io.ObjectStreamException
    {
        String[] pieces = data.split(",");
        Person result = new Person(pieces[0], pieces[1], Integer.parseInt(pieces[2]));
        if (pieces.length > 3)
        {
            result.setSpouse(new Person(pieces[3], pieces[4], Integer.parseInt
              (pieces[5])));
            result.getSpouse().setSpouse(result);
        }
        return result;
    }
}

public class Person
    implements java.io.Serializable
{
    public Person(String fn, String ln, int a)
    {
        this.firstName = fn; this.lastName = ln; this.age = a;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getAge() { return age; }
    public Person getSpouse() { return spouse; }

    private Object writeReplace()
        throws java.io.ObjectStreamException
    {
        return new PersonProxy(this);
    }

    public void setFirstName(String value) { firstName = value; }
    public void setLastName(String value) { lastName = value; }
    public void setAge(int value) { age = value; }
    public void setSpouse(Person value) { spouse = value; }

    public String toString()
    {
        return "[Person: firstName=" + firstName +
            " lastName=" + lastName +
            " age=" + age +
            " spouse=" + spouse.getFirstName() +
            "]";
    }

    private String firstName;
    private String lastName;
    private int age;
    private Person spouse;
}
```

위에서 PersonProxy 클래스가 Person 클래스의 모든 데이터를 추적하고 있다는 것을 확인할 수 있을 것이다. 이 말은 프록시들이 private 필드들에 접근하기 위해 Person 클래스의 내부 클래스이어야 할 필요가 있다는 말이다.[10] 또, Person의 spouse 정보를 얻기 위해 다른 클래스의 객체 참조를 확인하고 이를 직렬화해야 할 필요도 있다.

이 방법은 읽기와 쓰기를 동시에 사용할 필요가 없는 경우에 사용할 수 있는 방법 중 하나다. 예를 들어, 아예 다른 클래스로 리팩토링된 클래스에 readResolve 메소드를 이용하여 새로운 클래스 유형으로 조용히 직렬화가 되도록 처리할 수도 있다. 비슷하게 오래된

[10] Person 클래스가 모든 private 필드들에 대한 public 인터페이스를 제공한다면 반드시 그럴 필요는 없지만, 그렇다면 캡슐화의 의미가 없다.

클래스에 writeReplace 메소드를 구현하여 그 클래스가 새로운 클래스 버전으로 직렬화되도록 처리할 수도 있다.

5. 믿더라도 검증해보라[11]

직렬화되어 이진 형식으로 전달된 데이터가 최초에 직렬화될 때의 스트림 형식 그대로일 것이라고 가정하는 것이 나쁜 것은 아니지만, 전직 미국 대통령이 지적했듯이, "믿더라도 검증해" 보아야 한다.

직렬화된 객체에 대해 역직렬화한 후에는 반드시 해당 값이 적절한 범위 내에 있는지 확인해야 한다는 말이다. 이를 위해 ObjectInputValidation 인터페이스를 구현하고 validateObject 메소드를 재정의하면 된다. 뭔가 데이터가 잘못된 것처럼 보이면 InvalidObjectException을 던지면 된다.

결론

Java 객체 직렬화는 보통 개발자들에 의해 생각되는 것보다 훨씬 유연하며, 처리하기 어려운 문제들도 생각보다 쉽게 해결할 수 있도록 해 준다. 다행히도, 이와 같은 주옥 같은 코드들이 JVM 내에 여기저기 산재해 있는데, 중요한 것은 이것들에 대해 공부하고, 머리가 깨질 것 같은 어려운 문제들을 해결하기 위해 이들을 적절히 활용해야 할 것이다.

---

[11] Trust, but verify. 미국 40대 대통령 로널드 레이건이 소련과의 관계에 대해 자주 사용했던 말이다.