Сериализация как она есть

Последнее изменение: 19 апреля 2011г.

На первый взгляд, сериализация кажется тривиальным процессом. Действительно, что может быть проще? Объявил класс реализующим интерфейс `java.io.Serializable` – и все дела. Можно сериализовать класс без проблем.

Теоретически это действительно так. Практически же – есть очень много тонкостей. Они связаны с производительностью, с десериализацией, с безопасностью класса. И еще с очень многими аспектами. О таких тонкостях и пойдет разговор.

Статью эту можно разделить на следующие части:

- Тонкости механизмов
- Зачем нужен `Externalizable`
- Производительность
- Обратная сторона медали
- Безопасность данных
- Сериализация объектов Singleton

Приступим к первой части –

Тонкости механизмов

Прежде всего, вопрос на засыпку. А сколько существует способов сделать объект сериализуемым? Практика показывает, что более 90% разработчиков отвечают на этот вопрос приблизительно одинаково (с точностью до формулировки) – такой способ один. Между тем, их **два**. Про второй вспоминают далеко не все, не говоря уж о том, чтобы сказать что-то внятное о его особенностях.

Итак, каковы же эти способы? Про первый помнят все. Это уже упомянутая реализация `java.io.Serializable`, не требующая никаких усилий. Второй способ – это тоже реализация интерфейса, но уже другого: `java.io.Externalizable`. В отличие от `java.io.Serializable`, он содержит два метода, которые необходимо реализовать – `writeExternal(ObjectOutput)` и `readExternal(ObjectInput)`. В этих методах как раз и находится логика сериализации/десериализации.

Замечание. В дальнейшем сериализацию с реализацией `Serializable` я буду

иногда называть стандартной, а реализацию `Externalizable` – расширенной.

Еще одно замечание. Я намеренно не затрагиваю сейчас такие возможности управления стандартной сериализацией, как определение `readObject` и `writeObject`, т.к. считаю эти способы в некоторой степени некорректными. Эти методы не определены в интерфейсе `Serializable` и являются, фактически, подпорками для обхода ограничений и придания стандартной сериализации гибкости. В `Externalizable` же методы, обеспечивающие гибкость, заложены изначально.

Зададимся еще одним вопросом. А как, собственно, работает стандартная сериализация, с использованием `java.io.Serializable`? А работает она через Reflection API. Т.е. класс разбирается как набор полей, каждое из которых пишется в выходной поток. Думаю, понятно, что операция эта неоптимальна по производительности. Насколько именно – выясним позднее.

Между упомянутыми двумя способами сериализации существует еще одно серьезное отличие. А именно – в механизме десериализации. При использовании `Serializable` десериализация происходит так: под объект выделяется память, после чего его поля заполняются значениями из потока. Конструктор объекта при этом *не вызывается*.

Тут надо еще отдельно рассмотреть такую ситуацию. Хорошо, наш класс сериализуемый. А его родитель? Совершенно необязательно! Более того, если наследовать класс от `Object` – родитель уж точно НЕсериализуемый. И пусть о полях `Object` мы ничего не знаем, но в наших собственных родительских классах они вполне могут быть. Что будет с ними? В поток сериализации они не попадут. Какие значения они примут при десериализации?

Посмотрим на этот пример:

```
package ru.skipy.tests.io;

import java.io.*;


public class ParentDeserializationTest {

    public static void main(String[] args){
        try {
            System.out.println("Creating...");
            Child c = new Child(1);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(baos);
```

```
            c.field = 10;
            System.out.println("Serializing...");
            oos.writeObject(c);
            oos.flush();
            baos.flush();
            oos.close();
            baos.close();
            ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
            ObjectInputStream ois = new ObjectInputStream(bais);
            System.out.println("Deserializing...");
            Child c1 = (Child)ois.readObject();
            System.out.println("c1.i="+c1.getI());
            System.out.println("c1.field="+c1.getField());
        } catch (IOException ex){
            ex.printStackTrace();
        } catch (ClassNotFoundException ex){
            ex.printStackTrace();
        }
    }

    public static class Parent {
        protected int field;
        protected Parent(){
            field = 5;
            System.out.println("Parent::Constructor");
        }
        public int getField() {
            return field;
        }
    }

    public static class Child extends Parent implements Serializable{
        protected int i;
        public Child(int i){
            this.i = i;
            System.out.println("Child::Constructor");
        }
        public int getI() {
            return i;
        }
    }
}
```

Он прозрачен – у нас есть несериализуемый родительский класс и сериализуемый дочерний. И вот что получается:

```
Creating...
Parent::Constructor
Child::Constructor
Serializing...
Deserializing...
Parent::Constructor
```

```
c1.i=1
c1.field=5
```

То есть **при десериализации вызывается конструктор без параметров родительского НЕсериализуемого класса**. И если такого конструктора не будет – при десериализации возникнет ошибка. Конструктор же дочернего объекта, того, который мы десериализуем, не вызывается, как и было сказано выше.

Так ведут себя стандартные механизмы при использовании `Serializable`. При использовании же `Externalizable` ситуация иная. Сначала *вызывается конструктор без параметров*, а потом уже на созданном объекте вызывается метод `readExternal`, который и вычитывает, собственно, все свои данные. Потому **– любой реализующий интерфейс `Externalizable` класс обязан иметь `public` конструктор без параметров!** Более того, поскольку все наследники такого класса тоже будут считаться реализующими интерфейс `Externalizable`, у них тоже должен быть конструктор без параметров!

Пойдем дальше. Существует такой модификатор поля как `transient`. Он означает, что это поле *не должно* быть сериализовано. Однако, как вы сами понимаете, указание это действует только на стандартный механизм сериализации. При использовании `Externalizable` никто не мешает сериализовать это поле, равно как и вычитать его. Если поле объявлено `transient`, то при десериализации объекта оно принимает значение по умолчанию.

Еще один достаточно тонкий момент. При стандартной сериализации поля, имеющие модификатор `static`, *не сериализуются*. Соответственно, после десериализации это поле значения не меняет. Разумеется, при реализации `Externalizable` сериализовать и десериализовать это поле никто не мешает, однако я крайне не рекомендую этого делать, т.к. это может привести к трудноуловимым ошибкам.

Поля с модификатором `final` сериализуются как и обычные. За одним исключением – их невозможно десериализовать при использовании `Externalizable`. Ибо `final`-поля должны быть инициализированы в конструкторе, а после этого в `readExternal` изменить значение этого поля будет невозможно. Соответственно – если вам необходимо сериализовать объект, имеющий `final`-поле, вам придется использовать только стандартную сериализацию.

Еще один момент, который многие не знают. При стандартной сериализации учитывается порядок объявления полей в классе. Во всяком случае, так было в

ранних версиях, в JVM версии 1.6 реализации Oracle уже порядок неважен, важны тип и имя поля. Состав же методов с очень большой вероятностью повлияет на стандартный механизм, при том, что поля могут вообще остаться теми же. Чтобы этого избежать, есть следующий механизм. В каждый класс, реализующий интерфейс `Serializable`, на стадии компиляции добавляется еще одно поле – **`private static final long`** `serialVersionUID`. Это поле содержит уникальный идентификатор версии сериализованного класса. Оно вычисляется по содержимому класса – полям, их порядку объявления, методам, их порядку объявления. Соответственно, при любом изменении в классе это поле поменяет свое значение.

Это поле записывается в поток при сериализации класса. Кстати, это, пожалуй, единственный известный мне случай, когда `static`-поле сериализуется. При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают – инициируется исключение наподобие этого:

```
java.io.InvalidClassException: test.ser2.ChildExt;
    local class incompatible: stream classdesc serialVersionUID = 8218484765288926197,
                                  local class serialVersionUID = 1465687698753363969
```

Есть, однако, способ эту проверку если не обойти, то обмануть. Это может оказаться полезным, если набор полей класса и их порядок уже определен, а методы класса могут меняться. В этом случае сериализации ничего не угрожает, однако стандартный механизм не даст десериализовать данные с использованием байткода измененого класса. Но, как я уже сказал, его можно обмануть. А именно – вручную в классе определить поле **`private static final long`** `serialVersionUID`. В принципе, значение этого поля может быть абсолютно любым. Некоторые предпочитают ставить его равным дате модификации кода. Некоторые вообще используют `1L`. Для получения стандартного значения (того, которое вычисляется внутренним механизмом) можно использовать утилиту `serialver`, входящую в поставку SDK. После такого определения значение поля будет фиксировано, следовательно, десериализация всегда будет разрешена.

Более того, в версии 5.0 в документации появилось приблизительно следующее: *крайне рекомендуется* всем сериализуемым классам декларировать это поле в явном виде, ибо вычисление по умолчанию очень чувствительно к деталям структуры класса, которые могут различаться в зависимости от реализации компилятора, и вызывать таким образом неожиданные `InvalidClassException` при десериализации. Объявлять это поле лучше как `private`, т.к. оно относится исключительно к классу, в котором объявляется. Хотя в спецификации

модификатор не оговорен.

Рассмотрим теперь вот какой аспект. Пусть у нас есть такая структура классов:

```
public class A{
    public int iPublic;
    protected int iProtected;
    int iPackage;
    private int iPrivate;
}

public class B extends A implements Serializable{}
```

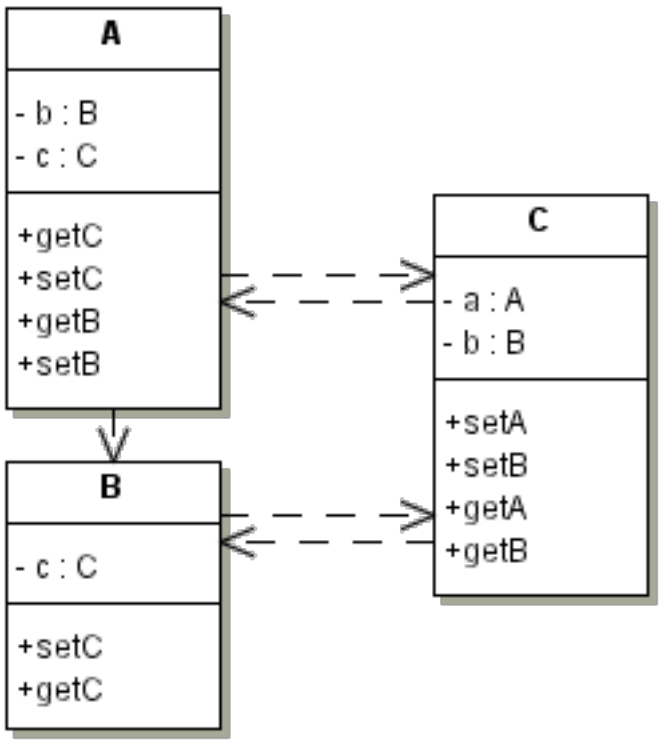
Иначе говоря, у нас есть класс, унаследованный от несериализуемого родителя. Можно ли сериализовать этот класс, и что для этого надо? Что будет с переменными родительского класса?

Ответ такой. Да, сериализовать экземпляр класса `B` можно. Что для этого нужно? А нужно, чтобы у класса `A` был конструктор без параметров, `public` либо `protected`. Тогда при десериализации все переменные класса `A` будут инициализированы с помощью этого конструктора. Переменные класса `B` будут инициализированы значениями из потока сериализованных данных.

Теоретически можно в классе B определить методы, о которых я говорил в начале – `readObject` и `writeObject`, – в начале которых производить (де-)сериализацию переменных класса `B` через `in.defaultReadObject`/`out.defaultWriteObject`, а потом – (де-)сериализацию доступных переменных из класса `A` (в нашем случае это `iPublic`, `iProtected` и `iPackage`, если `B` находится с том же пакете, что и `A`). Однако, на мой взгляд, для этого лучше использовать расширенную сериализацию.

Следующий момент, которого я хотел бы коснуться – сериализация нескольких объектов. Пусть у нас есть следующая структура классов:

```
public class A implements Serializable{
    private C c;
    private B b;
    public void setC(C c) {this.c = c;}
    public void setB(B b) {this.b = b;}
    public C getC() {return c;}
    public B getB() {return b;}
}
public class B implements Serializable{
    private C c;
    public void setC(C c) {this.c = c;}
```




```
    public C getC() {return c;}
}
public class C implements Serializable{
    private A a;
    private B b;
    public void setA(A a) {this.a = a;}
    public void setB(B b) {this.b = b;}
    public B getB() {return b;}
    public A getA() {return a;}
}
```

Что произойдет, если сериализовать экземпляр класса `A`? Он потащит за собой экземпляр класса `B`, который, в свою очередь, потащит экземпляр `C`, который имеет ссылку на экземпляр `A`, тот же самый, с которого все начиналось. Замкнутый круг и бесконечная рекурсия? К счастью, нет. Посмотрим на следующий тестовый код:

```
A a = new A();
B b = new B();
C c = new C();

a.setB(b);
a.setC(c);
b.setC(c);
c.setA(a);
c.setB(b);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(a);
oos.writeObject(b);
oos.writeObject(c);
oos.flush();
oos.close();

ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(baos.toByteArray(
A a1 = (A)ois.readObject();
B b1 = (B)ois.readObject();
C c1 = (C)ois.readObject();

System.out.println("a==a1: "+(a==a1));
System.out.println("b==b1: "+(b==b1));
System.out.println("c==c1: "+(c==c1));
System.out.println("a1.getB()==b1: "+(a1.getB()==b1));
System.out.println("a1.getC()==c1: "+(a1.getC()==c1));
System.out.println("b1.getC()==c1: "+(b1.getC()==c1));
System.out.println("c1.getA()==a1: "+(c1.getA()==a1));
```

```
System.out.println("c1.getB()==b1: "+(c1.getB()==b1));
```

Что мы делаем? Мы создаем по экземпляру классов `A`, `B` и `C`, ставим им ссылки друг на друга, после чего сериализуем каждый из них. Потом мы десериализуем их обратно и проводим серию проверок. Что получится в результате:

```
a==a1: false
b==b1: false
c==c1: false
a1.getB()==b1: true
a1.getC()==c1: true
b1.getC()==c1: true
c1.getA()==a1: true
c1.getB()==b1: true
```

Итак, что можно извлечь из этого теста. Первое. Ссылки на объекты после десериализации отличаются от ссылок до нее. Иначе говоря, при сериализации/десериализации объект *был скопирован*. Этот метод используется иногда для клонирования объектов.

Второй вывод, более существенный. При сериализации/десериализации нескольких объектов, имеющих перекрестные ссылки, эти ссылки остаются действительными после десериализации. Иначе говоря, если до сериализации они указывали на один объект, то после десериализации они тоже будут указывать на один объект.

Еще один небольшой тест в подтверждение этого:

```
B b = new B();
C c = new C();
b.setC(c);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(b);
oos.writeObject(c);
oos.writeObject(c);
oos.writeObject(c);
oos.flush();
oos.close();
ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(baos.toByteArray(
B b1 = (B)ois.readObject();
C c1 = (C)ois.readObject();
C c2 = (C)ois.readObject();
C c3 = (C)ois.readObject();
System.out.println("b1.getC()==c1: "+(b1.getC()==c1));
System.out.println("c1==c2: "+(c1==c2));
System.out.println("c1==c3: "+(c1==c3));
```

Объект класса `B` имеет ссылку на объект класса `C`. При сериализации `b` сериализуется вместе с экземпляром класса `C`, после чего тот же экземпляр `c` сериализуется трижды. Что получается после десериализации?

```
b1.getC()==c1: true
c1==c2: true
c1==c3: true
```

Как видим, все четыре десериализованных объекта на самом деле представляют собой один объект – ссылки на него равны. Ровно как это и было до сериализации.

Еще один интересный момент – что будет, если одновременно реализовать у класса `Externalizable` и `Serializable`? Как в том вопросе – слон против кита – кто кого поборет?

Поборет `Externalizable`. Механизм сериализации сначала проверяет его наличие, а уж потом – наличие `Serializable` Так что если класс `B`, реализующий `Serializable`, наследуется от класса `A`, реализующего `Externalizable`, поля класса `B` сериализованы не будут.

Последний момент – наследование. При наследовании от класса, реализующего `Serializable`, никаких дополнительных действий предпринимать не надо. Сериализация будет распространяться и на дочерний класс. При наследовании от класса, реализующего `Externalizable`, необходимо переопределить методы родительского класса `readExternal` и `writeExternal`. Иначе поля дочернего класса сериализованы не будут. В этом случае надо бы не забыть вызвать родительские методы, иначе не сериализованы будут уже родительские поля.

* * *

С деталями, пожалуй, закончили. Однако есть один вопрос, который мы не затронули, глобального характера. А именно –

Зачем нужен Externalizable

Зачем вообще нужна расширенная сериализация? Ответ прост. Во-первых, она дает гораздо большую гибкость. Во-вторых, зачастую она может дать немалый выигрыш по объему сериализованных данных. В-третьих, существует такой аспект как производительность, о котором мы поговорим ниже.

С гибкостью вроде как понятно всё. Действительно, мы можем управлять

процессами сериализации и десериализации как хотим, что делает нас независимыми от любых изменений в классе (как я говорил чуть выше, изменения в классе способны сильно повлиять на десериализацию). Потому хочу сказать пару слов о выигрыше по объему.

Допустим, у нас есть следующий класс:

```
public class DateAndTime{

  private short year;
  private byte month;
  private byte day;
  private byte hours;
  private byte minutes;
  private byte seconds;

}
```

Остальное несущественно. Поля можно было бы сделать типа `int`, но это лишь усилило бы эффект примера. Хотя в реальности поля могут быть типа `int` по соображениям производительности. В любом случае, суть понятна. Класс представляет собой дату и время. Нам он интересен прежде всего с точки зрения сериализации.

Возможно, проще всего было бы хранить простейший timestamp. Он имеет тип `long`, т.е. при сериализации он занял бы 8 байт. Кроме того, этот подход требует методов преобразования компонент в одно значение и обратно, т.е. – потеря в производительности. Плюс такого подхода – совершенно сумасшедшая дата, которая может поместиться в 64 бита. Это огромный запас прочности, чаще всего в реальности не нужный. Класс же, приведенный выше, займет 2 + 5*1 = 7 байт. Плюс служебные издержки на класс и 6 полей.

Можно ли как-нибудь ужать эти данные? Наверняка. Секунды и минуты лежат в интервале 0-59, т.е. для их представления достаточно 6 бит вместо 8. Часы – 0-23 (5 бит), дни – 0-30 (5 бит), месяцы – 0-11 (4 бита). Итого, всё без учета года – 26 бит. До размера `int` еще остается 6 бит. Теоретически, в некоторых случаях этого может хватить для года. Если нет – добавление еще одного байта увеличивает размер поля данных до 14 бит, что дает промежуток 0-16383. Этого более чем достаточно в реальных приложениях. Итого – мы ужали размер данных, необходимых для хранения нужной информации, до 5 байт. Если не до 4.

Недостаток тот же, что и в предыдущем случае – если хранить дату упакованной, то нужны методы преобразования. А хочется так – хранить в отдельных полях, а

сериализовать в упакованном виде. Вот тут как раз целесообразно использовать `Externalizable`:

```
public void writeExternal(ObjectOutput out){
    int packed = 0;
    packed += ((int)hours) << 27;
    packed += ((int)minutes) << 21;
    packed += ((int)seconds) << 15;
    packed += ((int)day) << 10;
    packed += ((int)month) << 6;
    packed += (((int)year) >> 8) & 0x3F;
    out.writeInt(packed);
    out.writeByte((byte)year);
}

public void readExternal(ObjectInput in){
    int packed = in.readInt();
    year = in.readByte() & 0xFF;
    year += (packed & 0x3F) << 8;
    month = (packed >> 6) & 0x0F;
    day = (packed >> 10) & 0x1F;
    seconds = (packed >> 15) & 0x3F;
    minutes = (packed >> 21) & 0x3F;
    hours = (packed >> 27);
}
```

Собственно, это все. После сериализации мы получаем служебные издержки на класс, два поля (вместо 6) и 5 байт данных. Что уже существенно лучше. Дальшейшую упаковку можно оставить специализированным библиотекам.

Приведенный пример весьма прост. Его основное предназначение – показать, как можно применять расширенную сериализацию. Хотя возможный выигрыш в объеме сериализованных данных – далеко не основное преимущество, на мой взгляд. Основное же преимущество, помимо гибкости... (плавно переходим к следующему разделу...)

Производительность

Как я уже говорил, стандартная сериализация работает через Reflection API. Что означает, что для сериализации берется класс сериализуемого объекта, у него берется список полей, по всем полям в цикле проверяются различные условия (`transient` или нет, если объект, то Externalizable или Serializable), значения пишутся в поток, причем достаются из полей тоже через reflection... В общем,

ситуация ясна. В противоположность этому методу, вся процедура при использовании расширенной сериализации контролируется самим разработчиком. Осталось выяснить, какие преимущества это дает по скорости.

Итак, условия теста. Объект произвольной структуры. Два варианта – один `Serializable`, второй `Externalizable`. Некоторое количество объектов обоих вариантов инициализируется произвольными (идентичными для каждой пары объектов) данными, после чего помещается в контейнер. Контейнер тоже в одном случае `Serializable`, в другом `Externalizable`. Далее контейнеры будут сериализованы и десериализованы с замерами времени.

Полный код теста вместе с build-файлом для `ant` можно найти тут – serialization.zip. В тексте я буду приводить только отрывки.

Сериализуемый объект содержит следующий набор полей:

```
private int fieldInt;
private boolean fieldBoolean;
private long fieldLong;
private float fieldFloat;
private double fieldDouble;
private String fieldString;
```

Тест содержит три реализации `Externalizable` контейнеров. Первая из них, `ContainerExt1`, простейшая. Это просто сериализация содержащего объекты `java.util.List`:

```
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(items);
}
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    items = (List<ItemExt>)in.readObject();
}
```

Вторая реализация, `ContainerExt2`, сериализует последовательно все имеющиеся объекты, предваряя их количеством объектов:

```
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(items.size());
    for(Externalizable ext : items)
        out.writeObject(ext);
}
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    int count = in.readInt();
    for(int i=0; i<count; i++){
```

```
        ItemExt ext = (ItemExt)in.readObject();
        items.add(ext);
    }
}
```

Третья реализация, `ContainerExt3`, использует `externalizable`-методы объектов:

```
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(items.size());
    for(Externalizable ext : items)
        ext.writeExternal(out);
}
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    int count = in.readInt();
    for(int i=0; i<count; i++){
        ItemExt ext = new ItemExt();
        ext.readExternal(in);
        items.add(ext);
    }
}
```

Запускается тест с помощью команды `ant` (поскольку задача `run` запускается по умолчанию). В `build`-файле задано количество создаваемых объектов – `100000`. Другое количество может быть задано с помощью параметра командной строки `-Dobjcount=<value>`.

Итак, каковы результаты выполнения теста? На 100000 создаваемых объектов (результаты могут незначительно отличаться от запуска к запуску):

```
Creating 100000 objects
Serializable: written in 3516ms, readed in 3235
Externalizable1: written in 4046ms, readed in 3234
Externalizable2: written in 3875ms, readed in 2985
Externalizable3: written in 235ms, readed in 297
```

И размеры сериализованных данных (размеры файлов на диске):

```
cont.ser        5 547 955
contExt1.ser    5 747 884
contExt2.ser    5 747 846
contExt3.ser    4 871 461
```

Что мы видим? Первый способ реализации `Externalizable` даже несколько хуже стандартной сериализации. Сериализация занимает немного больше времени, десериализация сравнима. Размеры файлов тоже немного в пользу стандартной

сериализации. Вывод – простейшая сериализация контейнера преимуществ не дает: +15% при сериализации, десериализация отличается на доли процента, причем как в одну, так и в другую сторону.

Второй способ реализации `Externalizable` по характеристикам практически идентичен первому. Чуть быстрее сериализация, но все равно проигрывает стандартной, десериализация чуть выигрывает. Размер файла практически идентичен первому способу (разница – 38 байт). Выигрыша по сравнению со стандартной сериализацией нет – +10% при сериализации, -8% при десериализации.

Третий способ реализации `Externalizable`. Вот тут есть на что посмотреть! Сериализация быстрее в 15 раз! Естественно, плюс-минус, но тем не менее – разница на порядок! Десериализация быстрее практически в 11 раз! Разница тоже на порядок! Опять же плюс-минус, но мне не удавалось получить разницу меньше, нежели в 5 раз. Ну и разница в размере файла -13%. Как маленькое, но приятное дополнение.

Думаю, комментарии излишни. Получаемые от **грамотной** реализации `Externalizable` преимущества в скорости с лихвой компенсируют затраты на эту самую реализацию. Грамотной – в смысле, целиком и полностью реализованной самостоятельно, без использования имеющихся механизмов сериализации целых объектов (в основном это методы `writeObject/readObject`). Использование же имеющихся механизмов и/или смешивание со стандартной сериализацией способно свести скоростные преимущества `Externalizable` на нет.

Однако есть и ...

Обратная сторона медали

И прежде всего это **нарушение целостности графа**. Поскольку протокол сериализации не используется – контроль целостности остается на самом разработчике. И об этом следует помнить, ибо в некоторых случаях можно легко убить все преимущества. Если, к примеру, необходимо сериализовать очень много экземпляров класса `A`, каждый из которых ссылается на единственный экземпляр класса `B`, то при неумелом использовании `Externalizable` может получиться так, что экземпляр `B` будет сериализован по разу на каждый экземпляр `A`, что даст потерю как в скорости, так и в объеме сериализованных данных. А при десериализации мы вообще получим кучу экземпляров `B` вместо одного! Что намного хуже.

Поэтому, да и не только, `Externalizable` следует использовать обдуманно. Как,

впрочем, и любую другую возможность. Если необходимо сериализовать достаточно сложные графы – пожалуй, лучше все-таки воспользоваться имеющимися механизмами. Если же объемы данных большие, но сложность невелика – можно немного поработать и получить солидный выигрыш в скорости. В любом случае лучше написать небольшой прототип и уже на нем оценивать реальную скорость и сложность реализации целостности.

Перейдем к следующему вопросу, связанному с сериализацией.

Безопасность данных

Есть такое правило: проверять входящие данные (входные параметры функций и т.п.) на "правильность" – соответствие определенным требованиям. Причем это не столько правило хорошего тона, сколько правило выживания приложения. Ибо если этого не сделать, то при передаче неверных параметров в лучшем случае (**действительно** – в **лучшем**!) приложение просто "упадет". В худшем случае оно тихо примет предложенные данные и может нанести значительно больший урон.

Про это правило худо-бедно, но помнят. Однако конструкторы и открытые методы – не единственный способ поставки данных объекту. Точно так же объект может быть создан с помощью десериализации. И вот тут о контроле внутреннего состояния полученного объекта, как правило, забывают. Между тем, создать поток для получения из него объекта с неверным внутренним состоянием не легко, а очень легко.

Пример номер один. Объект с двумя полями типа `java.util.Date`. Одно поле – начало интервала времени, другое – конец. Следовательно, между ними должно существовать определенное соотношение (конец должен быть не раньше начала). Однако любой человек, знающий байткод, сумеет отредактировать сериализованный объект так, что после десериализации конец интервала будет раньше начала. К чему приведет появление в системе такого объекта – предугадать сложно. В любом случае, ничего хорошего ждать не приходится. Потому, примите для себя...

Правило 1. После десериализации объекта необходимо проверить его внутреннее состояние (инварианты) на правильность, точно так же, как и при создании с помощью конструктора. Если объект не прошел такую проверку, необходимо инициировать исключение `java.io.InvalidObjectException`.

Пример номер два. Объект класса `A` содержит в себе `private`-поле типа `java.util.Date`. Для изменения снаружи объекта это поле недоступно. Однако

возможна следующая операция: к потоку дописывается некоторая информация. Потом, после десериализации из этого потока объекта класса `A` производится десериализация еще одного объекта, но уже типа `Date`. Как мы уже видели в примере ранее, можно создать такой поток (в примере он создавался легально), что при десериализации этот второй объект в действительности будет лишь ссылкой на экземпляр `Date`, казалось бы так надежно спрятанный внутри объекта класса `A`. Соответственно, с этим экземпляром можно делать все, что заблагорасудится.

Не буду вдаваться в подробности. Описание этого приема есть в книге Джошуа Блох. Java. Эффективное программирование, в статье 56. Скажу только, что достаточно к потоку дописать 5 байт, чтобы добиться желаемого.

Чтобы этого избежать, необходимо следовать следующему правилу:

Правило 2. Если в составе класса `A` присутствуют объекты, которые не должны быть доступными для изменения извне, то при десериализации экземпляра класса `A` необходимо вместо этих объектов создать и сохранить их копии.

Приведенные выше примеры показывают возможные "дыры" в безопасности. Следование упомянутым правилам, разумеется, не спасает от проблем, но может существенно снизить их количество. Советую по этому поводу почитать книгу Джошуа Блох. Java. Эффективное программирование, статью 56.

Ну и последняя тема, которой я хотел бы коснуться –

Сериализация объектов Singleton

Тех, кто не в курсе, что такое `Singleton`, отсылаю к отдельной статье.

В чем проблема сериализации `Singleton`-ов? А проблема в уже упомянутом мной факте – после десериализации мы получим другой объект. Это видно в результатах первого из тестов в этой статье – ссылки на исходный и десериализованный объекты не совпадают. Таким образом, сериализация дает возможность создать `Singleton` еще раз, что нам совсем не нужно. Можно, конечно, запретить сериализовать `Singleton`-ы, но это, фактически, уход от проблемы, а не ее решение.

Решение же заключается в следующем. В классе определяется метод со следующей сигнатурой

```
ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException
```

Модификатор доступа может быть `private`, `protected` и по умолчанию (`default`). Можно, наверное, сделать его и `public`, но смысла я в этом не вижу. Назначение этого метода – возвращать замещающий объект вместо объекта, на котором он вызван. Приведу простой пример:

```
public class Answer implements Serializable{

    private static final String STR_YES = "Yes";
    private static final String STR_NO = "No";

    public static final Answer YES = new Answer(STR_YES);
    public static final Answer NO = new Answer(STR_NO);

    private String answer = null;

    private Answer(String answer){
        this.answer = answer;
    }

    private Object readResolve() throws ObjectStreamException{
        if (STR_YES.equals(answer))
            return YES;
        if (STR_NO.equals(answer))
            return NO;
        throw new InvalidObjectException("Unknown value: " + answer);
    }
}
```

Класс, приведенный выше – простейший перечислимый тип. Всего два значения – `Answer.YES` и `Answer.NO`. Соответственно, именно эти два значения и должны фигурировать после десериализации. Что делается в методе `readResolve`? Он вызывается на десериализованном объекте. И возвращать он должен уже существующий экземпляр класса, соответствующий внутреннему состоянию десериализованного объекта. В данном примере – проверяется значение поля `answer`. Если объект, соответствующий внутреннему состоянию, не найден... На мой взгляд, это зависит от ситуации. В приведенном примере стоит инициировать исключение. Возможно, в каких-то ситуациях будет полезно вернуть `this`. Примером этого, например, является реализация `java.util.logging.Level`.

Существует и обратный метод – `writeReplace`, который, как вы, наверное, уже догадались, позволяет выдать замещающий объект вместо текущего, для сериализации. Мне, честно сказать, трудно представить себе ситуации, в

которых это может понадобиться. Хотя в недрах кода Sun он как-то используется.

Оба метода, как `readResolve`, так и `writeReplace`, вызываются при использовании стандартных средств сериализации (методов `readObject` и `writeObject`), вне зависимости от того, объявлен ли сериализуемый класс как `Serializable` или `Externalizable`.

Самое интересное, что, похоже, из этих методов можно возвращать не только экземпляр класса, в котором этот метод определен, но и экземпляр другого класса. Я видел подобные примеры в глубинах библиотек Sun, во всяком случае, для `writeReplace` – точно видел. Но по каким принципам можно это делать – не берусь пока судить. Вообще, советую интересующимся просмотреть исходники J2SE 5.0, причем полные. Они доступны по лицензии JRL. Там есть много интересных примеров использования этих методов. Исходники можно взять тут – http://java.sun.com/j2se/jrl_download.html. Правда, требуется регистрация, но она, естественно, бесплатна.

Отдельно хочу коснуться сериализации перечислений (`enum`), появившихся в Java 5.0. Поскольку при сериализации в поток пишется имя элемента и его порядковый номер в определении в классе, можно было бы ожидать проблем при десериализации в случае изменения порядкового номера (что может случиться очень легко – достаточно поменять элементы местами). Однако, к счастью, таких проблем нет. Десериализация объектов типа `enum` контролируется для обеспечения соответствия десериализуемых экземпляров уже имеющимся у виртуальной машины. Фактически, это то, что делает обычно метод `readResolve`, но реализовано где-то существенно глубже. Сопоставление объектов осуществляется по имени. Разработчикам версии 5.0 – респект!

* * *

Наверное, на текущий момент это все, что я хотел рассказать о сериализации. Думаю, теперь она не кажется такой простой, какой казалась до прочтения этой статьи. И хорошо. Пребывание в блаженном неведении к добру не приводит.