**Секреты сериализации в Java**

**Java Serialization API**

**Сериализацию не использовать (Джошуа Блох)!!!**

**Использовать вместо нее механизмы GSON или protobuf!!!**

**Т.к. при десериализации пролазят вирусы!!!**

**Если всеже на старом коде придется использовать сериализацию/десериализацию, то использовать ObjectInputFilter и в фильтр передать список разрешенных к открытию классов («белый список»).**

*Java Serialization API используется множеством других Java API (например, RMI и JavaBeans) для сохранения объектов за пределами жизненного цикла виртуальной машины. Вы также можете самостоятельно использовать Java Serialization API  для сохранения объектов в собственных целях. Несмотря на простоту основ сериализации Java, в использовании API существуют некоторые сложные моменты. В этой статье Тодд Гриньер (Todd Greanier) откроет вам секреты использования Java Serialization API.*

Все мы знаем о том, что Java позволяет создавать в памяти объекты для многократного использования. Однако все эти объекты существуют лишь до тех пор, пока выполняется создавшая их виртуальная машина. Было бы неплохо, если бы создаваемые нами объекты могли бы существовать и вне пределов жизненного цикла виртуальной машины, не так ли? Что же, используя сериализацию объектов вы можете разложить свои объекты на байты и затем использовать их наиболее эффективным образом.

Сериализация объектов - это процесс сохранения состояния объектов в виде последовательности байтов, а также процесс восстановления в дальнейшем из этих байтов "живых" объектов. Java Serialization API предоставляет разработчикам Java стандартный механизм управления сериализацией объектов. API мал и легок в применении, а его классы и методы просты для понимания.

В этой статье мы увидим каким образом можно сохранять Java объекты и, начав с основ, закончим знакомством с более сложными концепциями. Мы рассмотрим три различных способа выполнения сериализации (используя протокол по умолчанию, изменяя протокол по умолчанию и создавая свой собственный протокол) и познакомимся с ситуациями, возникающими при использовании этих схем сохранения, такими как кэширование объектов, контроль версий и проблемы с быстродействием.

Прочитав статью вы получите полное представление об этом мощном, но зачастую плохо понимаемом Java API.

**Начнем с начала: Механизм используемый по умолчанию**

Давайте начнем с основ. Для сохранения объекта в Java мы должны иметь объект, нуждающийся в сохранении и этот объект должен быть отмечен как сериализуемый. Это осуществляется путем реализации объектом интерфейса *java.io.Serializable*, что является для API знаком того, что объект может быть разложен на байты, а затем вновь восстановлен.

Давайте взглянем на сохраняемый класс, используемый в нашей статье для демонстрации механизма сериализации:

*10 import java.io.Serializable;*

*20 import java.util.Date;*

*30 import java.util.Calendar;*

*40* ***public class PersistentTime implements Serializable***

*50 {*

*60 private Date time;*

*70*

*80 public PersistentTime()*

*90 {*

*100 time = Calendar.getInstance().getTime();*

*110 }*

*120*

*130 public Date getTime()*

*140 {*

*150 return time;*

*160 }*

*170 }*

Как вы видите, единственное чем этот класс отличается от обычного класса - то, что он реализует интерфейс *java.io.Serializable* в 40-й строке. Будучи совершенно пустым, интерфейс *Serializable* является лишь *маркерным интерфейсом* - он позволяет механизму сериализации определить, может ли данный класс быть сохранен. Итак, мы можем сформулировать первое правило сериализации:

**Правило №1: Сохраняемый объект должен реализовать интерфейс *Serializable* или унаследовать эту реализацию от вышестоящего по иерархии объекта.**

Следующим шагом является, собственно, сохранение объекта. Оно выполняется при помощи класса *java.io.ObjectOutputStream*. Этот класс является *фильтрующим потоком (filter stream)* - он окружает низкоуровневый поток байтов (называемый *узловым потоком (node stream)*) и предоставляет нам поток сериализации. Узловые потоки могут быть использованы для записи в файловую систему, или даже в сокеты. Это означает, что мы с легкостью можем передавать разложенные на байты объекты по сети и затем восстанавливать их на других компьютерах!

Давайте взглянем на код, используемый для сохранения объекта *PersistentTime*:

*10 import java.io.ObjectOutputStream;*

*20 import java.io.FileOutputStream;*

*30 import java.io.IOException;*

*40 public class FlattenTime*

*50 {*

*60 public static void main(String [] args)*

*70 {*

*80 String filename = "time.ser";*

*90 if(args.length > 0)*

*100 {*

*110 filename = args[0];*

*120 }*

*130 PersistentTime time = new PersistentTime();*

*140 FileOutputStream fos = null;*

*150 ObjectOutputStream out = null;*

*160 try*

*170 {*

*180 fos = new FileOutputStream(filename);*

*190 out = new ObjectOutputStream(fos);*

*200* ***out.writeObject(time)****;*

*210 out.close();*

*220 }*

*230 catch(IOException ex)*

*240 {*

*250 ex.printStackTrace();*

*260 }*

*270 }*

*280 }*

Реальная работа выполняется в 200-й строке, когда мы вызываем метод *ObjectOutputStream.writeObject()*, который запускает механизм сериализации и объект разлагается на байты (в данном случае в файл).

Для восстановления объекта из файла можно использовать следующий код:

*10 import java.io.ObjectInputStream;*

*20 import java.io.FileInputStream;*

*30 import java.io.IOException;*

*40 import java.util.Calendar;*

*50 public class InflateTime*

*60 {*

*70 public static void main(String [] args)*

*80 {*

*90 String filename = "time.ser";*

*100 if(args.length > 0)*

*110 {*

*120 filename = args[0];*

*130 }*

*140 PersistentTime time = null;*

*150 FileInputStream fis = null;*

*160 ObjectInputStream in = null;*

*170 try*

*180 {*

*190 fis = new FileInputStream(filename);*

*200 in = new ObjectInputStream(fis);*

*210 time = (PersistentTime)in.readObject();*

*220 in.close();*

*230 }*

*240 catch(IOException ex)*

*250 {*

*260 ex.printStackTrace();*

*270 }*

*280 catch(ClassNotFoundException ex)*

*290 {*

*300 ex.printStackTrace();*

*310 }*

*320 // распечатать восстановленное время*

*330 System.out.println("Время разложения: " + time.getTime());*

*340 System.out.println();*

*350 // распечатать текущее время*

*360 System.out.println("Текущее время: " + Calendar.getInstance().getTime());*

*370 }*

*380}*

В этом коде в 210-й строке происходит восстановление объекта при помощи вызова метода *ObjectInputStream.readObject()*. Метод считывает последовательность байтов, которую мы перед этим сохранили в файле, и создает "живой" объект, полностью повторяющий оригинал. Поскольку *readObject()* может считывать любой сериализуемый объект, необходимо его присвоение соответствующему типу. Таким образом, из системы, в которой происходит восстановление объекта, должен быть доступен файл класса. Другими словами, при сериализации не сохраняется ни файл класса объекта, ни его методы, сохраняется лишь *состояние* объекта.

Затем, в 360-й строке, мы просто вызываем метод *getTime()*, чтобы получить время у разложенного объекта. Время разложенного объекта сравнивается с текущим временем, дабы показать что механизм действительно работает так, как мы ожидаем.

**Несериализуемые объекты**

Основной механизм сериализации объектов Java прост для применения, но есть еще кое-что, что вам необходимо знать. Как упоминалось ранее, сохраняться могут лишь объекты, помеченные как *Serializable*. Класс *java.lang.Object* не реализует этот интерфейс, поэтому не все объекты Java могут быть автоматически сохранены. Хорошая новость заключается в том, что большая часть из них, включая *AWT* и компоненты *Swing GUI*, строки и массивы - сериализуемые.

В то же время, некоторые системные классы, такие как *Thread*, *OutputStream* и его подклассы, и *Socket* - не сериализуемые На самом деле даже если бы они были сериализуемыми, ничего бы не изменилось. К примеру, поток, запущенный в моей виртуальной машине, использует системную память. Его сохранение и последующее восстановление в вашей виртуальной машине ни к чему не приведет. Другой важный момент, вытекающий из того, что *java.lang.Object* не реализует интерфейс *Serializable*, заключается в том, что любой созданный вами класс, который расширяет только *Object* (и больше никакие сериализуемые классы) не может быть сериализован до тех пор, пока вы сами не реализуете этот интерфейс (как было показано в предыдущем примере).

Такая ситуация вызывает проблему: что если у нас есть класс, который содержит экземпляр *Thread*? Можем ли мы в этом случае сохранить объект такого типа? Ответ положительный, поскольку мы имеем возможность сообщить механизму сериализации о своих намерениях, пометив объект *Thread* нашего класса как *нерезидентный (transient)*.

Предположим, нам нужно создать класс, выполняющий анимацию. В своем примере я не стал приводить код анимации, ограничившись только общим описанием класса:

*10 import java.io.Serializable;*

*20 public class PersistentAnimation implements Serializable, Runnable*

*30 {*

*40 transient private Thread animator;*

*50 private int animationSpeed;*

*60 public PersistentAnimation(int animationSpeed)*

*70 {*

*80 this.animationSpeed = animationSpeed;*

*90 animator = new Thread(this);*

*100 animator.start();*

*110 }*

*120 public void run()*

*130 {*

*140 while(true)*

*150 {*

*160 // выполнение анимации*

*170 }*

*180 }*

*190 }*

При создании экземпляра класса *PersistentAnimation* создается и запускается поток *animator*. В 40-й строке мы пометили этот поток как *transient*, дабы сообщить механизму сериализации о том, что поле не должно сохраняться вместе с остальными состояниями этого объекта (в нашем случае, полем *speed*). Резюме: вы должны помечать как *transient* все поля, которые либо не могут быть сериализованы, либо те, которые вы не хотите сериализовать. Сериализация не заботится о модификаторах доступа, таких как *private*. Все резидентные поля рассматриваются как части состояния сохраняемого объекта, предназначенные для сохранения.

Следовательно нам нужно добавить еще одно правило. Итак, вот оба правила относительно сохраняемых объектов:

* **Правило №1: Сохраняемый объект должен реализовать интерфейс *Serializable* или унаследовать эту реализацию от вышестоящего по иерархии объекта.**
* **Правило №2: Сохраняемый объект должен пометить все свои несериализуемые поля как *transient*.**

**Изменение протокола по умолчанию**

Давайте перейдем ко второму способу реализации сериализации: изменение протокола по умолчанию. Хотя в анимационном коде, рассмотренном выше, был показан способ использования потока с объектом, обеспечив при этом его сериализацию, для того чтобы понять суть проблемы нужно разобраться в том, каким образом Java создает объекты. Задумайтесь, когда мы создаем объект при помощи ключевого слова *new*, конструктор объекта вызывается только при создании нового экземпляра объекта. Запомним этот факт и вновь взглянем на наш анимационный код. Сначала мы создаем экземпляр объекта *PersistentAnimation*, который запускает поток анимации. Затем мы сериализуем его при помощи кода:

PersistentAnimation animation = new PersistentAnimation(10);

FileOutputStream fos = ...

ObjectOutputStream out = new ObjectOutputStream(fos);

out.**writeObject(animation);**

Все кажется в порядке, но только до тех пор, пока мы не прочитаем объект используя вызов метода *readObject()*. Помните, конструктор вызывается только при создании нового экземпляра объекта. Здесь же мы не создаем нового экземпляра, мы просто восстанавливаем сохраненный объект. В результате анимационный объект отработает лишь однажды, при первом создании экземпляра этого объекта, что делает процесс его сохранения бессмысленным, не так ли?

Что же, есть и хорошая новость. Мы можем заставить наш объект работать так, как нам хочется, перезапуская анимацию при восстановлении объекта. Чтобы сделать это мы можем, например, создать вспомогательный метод *startAnimation()*, выполняющий те же функции, что и наш конструктор. Затем мы можем вызывать этот метод из конструктора, после каждой загрузки объекта. Неплохо, но несколько сложно. Теперь все, кто захочет использовать анимационный объект, должны знать о необходимости вызова этого метода после обычного процесса десериализации, что никак не вписывается в тот единообразный механизм, который Java Serialization API обещает разработчикам.

Однако, существует другое странное и хитрое решение. Используя встроенную возможность механизма сериализации, разработчики могут реализовать нормальный процесс поместив в свои файлы классов два метода:

* *private void writeObject(ObjectOutputStream out) throws IOException;*
* *private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;*

Обратите внимание, что оба метода (совершенно справедливо), объявлены как *private*, поскольку это гарантирует что методы не будут переопределены или перезагружены. Весь фокус в том, что виртуальная машина при вызове соответствующего метода автоматически проверяет, не были ли они объявлены в классе объекта. Виртуальная машина в любое время может вызвать *private* методы вашего класса, но другие объекты этого сделать не смогут. Таким образом обеспечивается целостность класса и нормальная работа протокол сериализации. Протокол сериализации всегда используется одинаково, путем вызова *ObjectOutputStream.writeObject()* или *ObjectInputStream.readObject()*. Таким образом, даже если в классе присутствуют эти специализированные *private* методы, сериализация объектов будет работать так же, как и для любых других вызываемых объектов.

Учитывая это, давайте взглянем на исправленную версию *PersistentAnimation*, в которую включены эти *private* методы для контроля над процессом десериализации через псевдо-конструктор:

10 import java.io.Serializable;

20 public class PersistentAnimation implements Serializable, Runnable

30 {

40 **transient** private Thread animator;

50 private int animationSpeed;

60 public PersistentAnimation(int animationSpeed)

70 {

80 this.animationSpeed = animationSpeed;

90 startAnimation();

100 }

110 public void run()

120 {

130 while(true)

140 {

150 // do animation here

160 }

170 }

180 private void writeObject(ObjectOutputStream out) throws IOException

190 {

200 out.defaultWriteObject();

220 }

230 private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException

240 {

250 // наш "псевдо-конструктор"

260 in.defaultReadObject();

270 // теперь мы вновь получили "живой" объект, поэтому давайте перестроим и запустим его

280 startAnimation();

290

300 }

310 private void startAnimation()

320 {

330 animator = new Thread(this);

340 animator.start();

350 }

360 }

Обратите внимание на первые строки новых *private* методов. Эти вызовы выполняют операции, созвучные их названию - они выполняют по умолчанию запись и чтение разложенных объектов, что важно, поскольку мы не заменяем нормальный процесс, а лишь дополняем его. Эти методы работают, потому что вызов *ObjectOutputStream.writeObject()* соответствует протоколу сериализации. Сначала объект проверяется на реализацию *Serializable*, а затем проверяется на наличие этих *private* методов. Если они есть, им в качестве параметра передается класс потока, через использование которого осуществляется управление кодом.

Эти *private* методы могут использоваться для внесения любого рода изменений в процесс сериализации. Например, для вывода объектов в поток может быть использована шифровка, а для ввода - дешифровка (при записи и чтении байтов данные записываются даже без применения технологии запутывания (obfuscation)). Методы могут использоваться также  для сохранения в потоке дополнительных данных, например кода версии. Ваши возможности поистине не ограничены.

**Остановите сериализацию!**

О'кей, мы уже кое-что узнали о процессе сериализации, а теперь давайте двигаться дальше. Что если вы создали класс, чей суперкласс сериализуемый, но при этом вы не хотите чтобы ваш класс был сериализуемым? Вы не можете "разреализовать" интерфейс, поэтому если суперкласс реализует *Serializable*, то и созданный вами новый класс также будет реализовать его (в соответствии с двумя рассмотренными выше правилами). Чтобы остановить автоматическую сериализацию вы можете снова применить *private* методы для создания исключительной ситуации *NotSerializableException*. Вот как это можно сделать:

*10 private void writeObject(ObjectOutputStream out) throws IOException*

*20 {*

*30 throw new NotSerializableException("Не сегодня!");*

*40 }*

*50 private void readObject(ObjectInputStream in) throws IOException*

*60 {*

*70 throw new NotSerializableException("Не сегодня!");*

*80 }*

Любая попытка записать или прочитать этот объект теперь приведет к возникновению исключительной ситуации. Запомните, если методы объявлены как *private*, никто не сможет модифицировать ваш код не изменяя исходный код класса. Java не позволяет переопределять такие методы.

**Создание своего собственного протокола: интерфейс Externalizable**

Наше обсуждение было бы неполным без упоминания третьей возможности сериализации: создания собственного протокола с интерфейсом *Externalizable*. Вместо реализации интерфейса *Serializable*, вы можете реализовать интерфейс *Externalizable*, который содержит два метода:

* *public void writeExternal(ObjectOutput out) throws IOException;*
* *public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;*

Для создания собственного протокола нужно просто переопределить эти два метода. В отличие от двух рассмотренных ранее вариантов сериализации, здесь ничего не делается автоматически. Протокол полностью в ваших руках. Хотя это и наболее сложный способ, при этом он наиболее контролируемый. Возьмем, к примеру, ситуацию с альтернативным типом сериализации: запись и чтение PDF файлов Java приложением. Если вы знаете как читать и записывать PDF файлы (требуется определенная последовательность байт), вы можете создать протокол с учетом специфики PDF используя методы *writeExternal* и *readExternal*.

Так же, как и в рассмотренных случаях, нет никакой разницы в том, как используется класс, реализующий *Externalizable*. Вы просто вызываете методы *writeObject()* или *readObject()* и, вуаля, эти расширяемые методы будут вызываться автоматически.

**Нюансы**

Существует несколько моментов, имеющих отношение к протоколу сериализации, которые могут показаться очень странными для непосвященного разработчика. Разумеется, цель этой статьи посвятить вас в них! Поэтому давайте обсудим некоторые нюансы, попробуем понять почему они появились и как с ними обращаться.

**Кэширование объектов в потоке**

Во-первых, рассмотрим ситуацию, когда объект однажды уже записанный в поток, спустя какое-то время записывается в него снова. По умолчанию, *ObjectOutputStream* сохраняет ссылки на объекты, которые в него записываются. Это означает, что если состояние записываемого объекта, который уже был записан, будет записано снова, новое состояние не сохраняется! Следующий фрагмент кода демонстрирует эту проблему:

*10 ObjectOutputStream out = new ObjectOutputStream(...);*

*20 MyObject obj = new MyObject(); // должен быть Serializable*

*30 obj.setState(100);*

*40 out.writeObject(obj); // сохраняет объект с состоянием = 100*

*50 obj.setState(200);*

*60 out.writeObject(obj); // не сохраняет новое состояние объекта*

Существует два способа взять ситуацию под контроль. Во-первых, вы можете каждый раз после вызова метода записи убеждаться в том, что поток закрыт. Во-вторых, вы можете вызвать метод *ObjectOutputStream.reset()*, который сигнализирует потоку о том, что необходимо освободить кэш от ссылок, которые он хранит, чтобы новые вызовы методов записи действительно записывали данные. Будьте осторожны, *reset* очищает весь кэш объекта, поэтому все ранее записанные объекты могут быть перезаписаны заново.

**Контроль версий**

Для второго случая представим что мы создали класс, затем создали его экземпляр, который записали в поток объекта. Этот разложенный на байты объект какое-то время находился в файловой системе. Тем временем вы обновляете файл класса, например, добавив в него новое поле. Что произойдет если затем вы попробуете прочитать разложенный объект?

Плохая новость заключается в том, что возникнет исключительная ситуация, а именно *java.io.InvalidClassException*, потому что всем классам, которые могут быть сохранены, присваивается уникальный идентификатор. Если идентификатор класса не совпадает с идентификатором разложенного объекта, возникает исключительная ситуация. Однако, если задуматься над этим, зачем нужны все эти исключительные ситуации, если вы всего лишь добавили новое поле? Разве нельзя установить в поле значение по умолчанию, а после сохранено?

Да, но это потребует легких манипуляций с кодом. Идентификатор, который является частью всех классов, хранится в поле, которое называется *serialVersionUID*. Если вы хотите контролировать версии, вы должны вручную задать поле *serialVersionUID* и убедиться в том, что оно такое же, и не зависит от изменений, внесенных вами в объект. Вы можете использовать утилиту, входящую в состав JDK, которая называется *serialver*, чтобы посмотреть какой код будет присвоен по умолчанию (это просто hash код объекта по умолчанию).

Вот пример использования *serialver* с классом *Baz*:

*> serialver Baz*

*> Baz: static final long serialVersionUID = 10275539472837495L;*

Просто скопируйте возвращенную строку с идентификатором версии и поместите ее в ваш код. (В Windows вы можете запустить эту утилиту с опцией *-show*, чтобы упростить процедуру копирования и вставки). Теперь, если вы внесли какие-либо изменения в файл класса *Baz*, просто убедитесь что указан тот же идентификатор версии и все будет в порядке.

Контроль за версией прекрасно работает до тех пор, пока вносимые изменения совместимы. К совместимым изменениям относят добавление и удаление методов и полей. К несовместимым изменениям относят изменение иерархии объектов или прекращение реализации интерфейса *Serializable*. Полный перечень совместимых и несовместимых изменений приведен в Спецификации сериализации Java (см. [Ссылки](http://www.ccfit.nsu.ru/~deviv/courses/oop/java_ser_rus.html#References)).

**Обсуждение производительности**

И третий момент. Механизм, используемый по умолчанию, несмотря на свою простоту, далеко не самый производительный. Я выполнил запись объект *Data* в файл 1'000 раз и повторил эту процедуру 100 раз. Среднее время записи объекта *Data* было 115 миллисекунд. Затем я вручную записал объект *Data*, используя стандартные инструменты ввода/вывода и повторил эту же операцию. Среднее время - 52 миллисекунды. Почти вдвое меньше! Это весьма обычный компромисс между простотой и производительностью, и сериализация - не исключение. Если скорость имеет для вашего приложения ключевое значение, вы можете выбрать метод создания собственного протокола.

Отдельного обсуждения заслуживает вышеупомянутый факт, что ссылки на объекты кэшируются в поток вывода. Согласно этому, система не может выполнять сбор мусора для записанных в поток объектов если поток не был закрыт. Лучшее решение (как всегда при помощи операций ввода/вывода) - это как можно скорее закрывать потоки после выполнения записи.

**Заключение**

Сериализация в Java проста для понимания и почти настолько же проста в реализации. Знание трех различных способов реализации сериализации должно помочь вам привязке API к вашей задаче. В этой статье вы познакомились с множеством механизмов сериализации и, я надеюсь, это пролило свет на эти вопросы, а не запутало вас еще больше. И, после прочтения, как это бывает в программировании, у вас появилось такое чувство, что вы давно знакомы с этим API. В статье изложены основы  API Java сериализации, но перед использованием спецификации я рекомендую получше изучить ее во всех подробностях

**Об авторе**

Тодд Гриньер (Todd Greanier), технический директор ComTech Training приступил к обучению и разработке на языке Java сразу же, как только тот был представлен широкой общественности. Являясь экспертом по распределенным Java технологиям, он проводил обучение работе с классами для широчайшего спектра тематик, включая JDBC, RMI, CORBA, UML, Swing, сервлеты/JSP, безопасность, JavaBeans, Enterprise Java Beans и многопоточность. Он также организовывал специальные семинары для корпораций с учетом специфики их требований. Тодд живет на севере шатата Нью-Йорк вместе с женой Стэйси (Stacey) и кошкой Бин (Bean).