N-Storage

Руководство разработчика

Email: lrndlrnd@mail.ru
URL: www.Sergey-L-Gladkiy.narod.ru

Содержание

Введение

Библиотека N-Storage предназначена для сохранения и чтения (сериализации и десериализации) данных объектов программ NET. Данные сохраняются в текстовом формате в виде объектных структур, легко воспринимаемых человеком. Библиотека разработана для использования с минимальным написанием кода при максимальных возможностях. Библиотека автоматически реализует поддержку совместимости версий (как прямую, так и обратную) при изменениях сохраняемых данных (удалении или добавлении полей). Система предусматривает диагностику ошибок при чтении данных с прямым указанием, в каких элементах произошли ошибки, а также причин этих ошибок.

ПРЕИМУЩЕСТВА библиотеки N-Storage:

1. 100% код C#.
2. Простой текстовый формат файлов, воспринимаемый человеком.
3. Простота использования в программе с минимальным написанием кода.
4. Автоматическая поддержка большинства типов NET (в том числе generic).
5. Прямая и обратная совместимость версий при изменении данных объектов.
6. Диагностика ошибок при чтении данных.
7. Возможность чтения данных при наличии некритических ошибок.
8. Любой тип автоматически сериализуем (без пометки Serializable).
9. Отсутствие проблем с сериализацией событий.
10. Отсутствие проблем с взаимными перекрестными ссылками данных.
11. Сериализация потоков и memory-mapped файлов.

Принципы сериализации

Сериализация это процесс сохранения состояния объектов (данных программы) путем преобразования их в поток данных с возможностью полного восстановления исходного состояния (десериализации).

Основные принципы сериализации, используемые в системе N-Storage, имеют целью реализации системы хранения данных, обеспечивающей простоту использования и поддержку прямой и обратной совместимости при изменении исходных данных системы.

- ***Сериализация должна сохранять ВСЕ данные, как есть.*** Поскольку сериализация должна выполнять сохранение состояния объекта, то должны сохраняться все данные объекта как есть – это гарантирует полное восстановление состояния. Таким образом, не нужно помечать каждый класс атрибутом Serializable.
- ***Десериализация должны восстанавливаться данные, а не выполняться действия.*** Кроме содержащихся данных, объекты так же включают свойства. В некоторых случаях свойства скрывают под простым интерфейсом доступа к значению сложную логику функционирования объекта. В некоторых случаях данная логика зависит от значений других данных и не может выполняться до полного восстановления состояния объекта. Поэтому сериализация сложных структур НЕ ДОЛЖНА основываться на свойствах, она ДОЛЖНА быть основана на данных.
- ***Возможность частичного восстановления данных.*** При десериализации должны восстанавливаться ВСЕ данные, восстановление которых возможно. Если нарушена только часть данных – это не должно влиять на возможность восстановления другой части данных (если формат файла не нарушен).
- ***Автоматическая поддержка прямой и обратной совместимости.*** При изменении внутреннего представления данных в программе, поддержка прямой и обратной совместимости версий должна АВТОМАТИЧЕСКИ поддерживаться системой хранения в тех случаях, если это может быть сделано без знаний об этих изменениях. Например, если в новой версии программы к сериализуемым данным были добавлены новые, то их не нужно помечать специальных атрибутом для системы хранения. Система хранения имеет всю информацию, достаточную для обработки такой ситуации. Если новая версия программы читает старые данные – отсутствие некоторой части данных не должно приводить к генерации ошибки системой хранения. Если старая версия программы читает новые данные, то наличие ‘лишних данных’ не должно представлять сложностей для системы хранения – эти данные просто не нужны старой программе для правильного функционирования. Программист не должен помечать такие поля атрибутом специальным атрибутом или писать специальный код.

1. Формат файлов

Формат NET Object Data Format (NODF) (формат хранения объектных данных NET) предназначен для хранения данных, соответствующим типам NET Framework, включая фундаментальные типы, массивы, обобщенные типы, перечисления, структуры и классы.

Формат файла

Файл формата NODF представляет собой текстовый файл в кодировке Unicode. Формат данных, текстовое представление которых зависит от языка (с плавающей точкой, дата и т.д.) соответствует формату Invariant Culture.

Структура файла

Файл NODF разбит на блоки. Каждый блок имеет следующий формат

[**BLOCK**:block_name]
	Block Data
[**BLOCKEND**:block_name]

BLOCK, **BLOCKEND** – служебные слова, block_name – имя блока, Block data – данные этого блока.

Файл NODF должен содержать три обязательных блока:
1. FORMAT – информация о формате и версии данных в файле;
2. INFORMATION – информация о данных (количество данных, количество типов и др.);
3. OBJECTS – сохраненные объектные данные.

Так же файл может содержать необязательный блок TYPES, который следует за блоком INFORMATION и содержит словарь типов данных.

Блоки FORMAT и INFORMATION

В блоках FORMAT и INFORMATION данные сохраняются в простейшем формате – последовательность записей вида

NAME=VALUE

где NAME – имя свойства, VALUE – его значение (преобразованное в строку в формате Invariant Culture). Для данных блоков допускается представление каждого свойства только в виде однострочной записи (многострочное представление недопустимо). Количество свойств может меняться в разных версиях. Запись и чтение свойств выполняется с помощью простейшего сериализатора – сохраняются данные public свойств.

Блок типов TYPES

В блоке типов TYPES содержаться записи следующего формата:

TYPE~ID64=TYPENAME

где **TYPE** – служебное слово;
ID64 – 64-битное беззнаковое целое число (номер типа, являющийся уникальным идентификатором);
TYPENAME – описание типа.

Описание типа TYPENAME имеет следующий синтаксис:

Name<Parameter1:Parameter2:…:ParameterN>

Name – имя типа;
ParameterX – параметры типа, зависят от описываемого типа и являются необязательными.

Для обобщений (generic) все параметры также являются описаниями типов, и их формат полностью совпадает с описанием типа TYPENAME.
Для описания имен массивов используется следующая схема: имя состоит из имени базового типа, за которым следуют квадратные скобки ‘[]’. Массивы могут быть регулярными и нерегулярными. Формат скобок соответствует формату при описании массивов в C# (см. приложение).

Блок объектных данных OBJECTS

Блок объектных данных содержит последовательность объектных записей (object record). Каждая объектная запись содержит данные одного объекта и имеет следующий формат:

OBJECT~ID64:TYPE
{
	Object data
}

OBJECT – служебное слово;
ID64 – 64-битное беззнаковое целое число (номер данного объекта в списке);
TYPE – тип объекта.
Object Data – данные объекта.

Тип объекта может быть представлен 2-мя способами. Если в файле присутствует блок типов TYPES, то тип объекта представлен как ссылка на тип в списке и имеет следующий формат:

TYPE~ID64

где **TYPE** – служебное слово;
ID64 – 64-битное беззнаковое целое число – идентификатор типа.

Если в файле отсутствует блок типов, то тип объекта совпадает с описанием типа TYPENAME.

Данные объекта состоят из последовательности элементных записей (member record). Исключение составляют объекты типа String и объекты значимых типов (см. далее). Каждая элементная запись имеет следующий формат:

```
MEMBERID Name:TYPE
/
        Member Value
\
```

MEMBERID – идентификатор элемента объектных данных, может иметь одно из следующих значений **DATA** (данные), **PROPERTY** (свойство);
Name – имя элемента;
TYPE – тип элемента (полностью совпадает с типом объектной записи);
Member value – значение элемента данных.

Значение элемента данных имеет следующий синтаксис:

FORMATID<PARAMETERS>=value

FORMATID – идентификатор формата значения;
PARAMETERS – параметры данных, являются необязательными и зависят от формата;
value – значение (возможно многострочное), формат которого зависит от идентификатора формата.

Синтаксис параметров данных: PARAMETERS представляют собой перечисленные через ':' значения, зависящие от формата данных. Для бинарных данных (потоков) это 2 беззнаковых 64-х битных целых – размер данных в байтах и текущая позиция. Для массива указываются его размеры. Поддерживаются регулярные (прямоугольные) массивы и нерегулярные. Для регулярных количество параметров равно количеству измерений, а значения соответствуют их длинам. Для нерегулярных – параметры внутренних измерений представляют собой параметры соответствующих массивов для каждого элемента данных (см. приложение).

Идентификатор формата может принимать следующие значения:

- **VALUE** – значение value является строковым представлением значения типа TYPE (в формате Invariant Culture). Используется для всех примитивных типов – значимых типов value types (int, double, кроме struct), перечислений enum и т.д.
- **REF** (reference - ссылка) – значение value является 64-битным беззнаковым целым – ссылкой на номер объекта в списке данных, или значением **NULL**. Применяется для всех ссылочных типов reference types (данные любых классов), данных типа struct и строковых (string) значений.
- **LIST** – value есть список разделенных пробелами значений примитивных типов (преобразованных в строку в формате Invariant Culture), применяется для хранения массивов примитивных типов.
- **REFLIST** – список разделенных пробелами 64-битных беззнаковых целых (идентификаторов в списке данных) или значений **NULL**. Применяется для хранения массивов значений ссылочных типов.
- **BINARY** – value есть поток бинарных данных. Применяется для хранения потоковых данных.
- **EXTERNAL** – value есть имя внешнего файла, в котором хранятся данные (внешний файл должен находится в директории, которая расположена там же, где находится данный файл и ее имя должно совпадать с именем данного файла).

Данные объекта Object Data для типа String и объектов примитивных типов (значимые, кроме struct) имеют следующий формат:

VALUE<PARAMETERS>= data value

PARAMETERS – параметры данных, являются необязательными и зависят от типа;
data value – значение данных.
Для типа String параметры есть длина строки (количество символов), значение data value начинается со следующей строки.
Для примитивных типов data value есть строковое представление значения (в формате Invariant Culture).

2. Реализация N-Storage

Реализация.

1. Формат файла соответствует описанию NODF.
2. Сохраняются и читаются только элементы данных **DATA** (поля объектов).
3. Реализованы форматы значений **VALUE**, **REF**, **LIST**, **REFLIST**, **BINARY**.
4. Поддерживаемые типы данных: классы, структуры, generics, значимые типы, строки, nullables, массивы (регулярные и нерегулярные), интерфейсы, потоки, memory-mapped файлы.

Ограничения.

1. **НЕ** поддерживаются смешанные массивы – нерегулярные, содержащие регулярные.
2. **НЕ** поддерживаются массивы потоков.
3. **НЕ** поддерживаются потоки, не допускающие изменение текущей позиции.
4. **НЕ** поддерживаются виртуальные поля.

Примечания.

1. Для авто-свойств (свойств, у которых реализованы неявные методы доступа get и set), создаются скрытые поля для хранения данных свойства. При сохранении эти поля идентифицируются по имени свойства.
2. При преобразовании вещественных чисел в строку может происходить потеря точности и при последующем обратном преобразовании исходное и полученное значения могут быть не равны.

3. Использование

Сохранение и чтение данных

Основной класс, используемый для сериализации и десериализации - FileSerializer. Данный класс имеет 2 конструктора:
Для сериализации:

```
FileSerializer(object data, FormatAttributes fa, FileStoreOptions opt, string fileName)
```

Для десериализации:

```
FileSerializer(FormatAttributes fa, string fileName)
```

object data – объект для сериализации;
FormatAttributes fa – информация о формате и версии сериализуемых данных;
FileStoreOptions opt – опции сохранения данных;
string filename – имя файла для сериализации данных.

Параметр FileStoreOptions opt позволяет задавать параметры сериализации. Например, одной из опций является использование блока типов TYPES при сохранении данных.

Класс FormatAttributes содержит набор public свойств, которые сохраняются в блок FORMAT. Можно создавать наследников данного класса и добавлять в них дополнительные public свойства, которые будут автоматически сохраняться при сериализации.

Сериализация.
Сериализация выполняется в 2 этапа (вызовом 2-х методов):
1. Вызов метода
   bool PrepareSerialization()
   Данный метод создает структуру сериализуемых данных. Возвращает true, если структура создана.
2. Вызов метода
   bool Serialize()
   Данный метод сохраняет данные в файл. Возвращает true, если данные сохранены.

Десериализация.
Десериализация выполняется в 2 этапа (вызовом 2-х методов):
1. Вызов метода
   bool PrepareDeserialization()
   Данный метод открывает файл и читает заголовочные данные – данные версии и атрибуты (количество объектов в файле и т.д). Возвращает true, если заголовочные данные прочитаны успешно.
2. Вызов метода
   bool Deserialize()
   Данный метод читает данные из файла и выполняет реконструкцию данных. Возвращает true, если хотя бы одно данное прочитано и не произошло критических ошибок, приводящих к невозможности дальнейшего чтения данных.

При десериализации могут произойти некритические ошибки (не приводящие к невозможности продолжения процесса), такие как ‘лишние данные’, невозможность создать объекты некоторых типов и т.д. После десериализации информация о данных ошибках может быть получена вызовом метода

```
List<BaseObjectDataItem> GetErrorInformation()
```


Данный метод возвращает список объектов, которые были прочитаны с ошибками.

Сериализация специальных данных

Ограничения на сериализуемые данные обусловлены текущей программной реализацией.

Для десериализации все классы должны иметь 'public' конструктор без параметров. Если некоторый класс не имеет такого конструктора, объекты этого класса не будут созданы и в структуре данных будут отмечены ошибочными.

Примечание: Почему нужен именно 'public' конструктор. Если класс не имеет 'public' конструктора, его разработчик подразумевает, что объекты данного типа не могут быть созданы внешними объектами стандартным способом, а должны создаваться специальным методом. Типичным примером являются объекты одиночки 'singleton'.

Если некоторое поле не является необходимым для сохранения, его можно отметить атрибутом NonSerialized.

Если для некоторого класса невозможно создать public конструктор без параметров (например, для класса внешней библиотеки), то для создания объектов этого класса можно зарегистрировать генерирующий объект класса Creator. Данный класс имеет следующий конструктор

```
Creator(Type type, Create func)
```

Type type – тип создаваемых объектов;
Create func – делегат для создания объектов.
Генерирующий объект должен быть зарегистрирован вызовом метода *SetCreator* класса Instantiator.

Если некоторый класс не может быть сериализован по каким-либо причинам, то для него может быть зарегистрирован класс-хранитель – наследник класса Storer

Данный класс применяется для общего случая замены данных одного класса данными другого класса. Для реализации класса-хранителя необходимо создать класс – наследник Storer и переопределить два его абстрактных метода:

```
void FromObject(object wrapped)
void ToObject(ref object wrapped)
```

первый метод "копирует" данные из объекта в хранитель (перед сериализацией), второй – из хранителя в объект (после десериализации).
Класс должен реализовывать два конструктора:

```
Storer(object stored)
Storer()
```

первый – для сериализации, второй – для десериализации.

Для регистрации класса-хранителя необходимо создать класс – наследник StorerReg и переопределить два его абстрактных метода:

```
Type GetStoredClass()
Type GetDataClass()
```

Первый должен возвращать тип заменяемого класса, второй – тип класса-хранителя. Наследник StorerReg должен быть зарегистрирован методом SetStorer класса TypeRepository.

Приложение A. Примеры данных NODF

Блок FORMAT

```
[BLOCK:FORMAT]
Format=TEST
Version=1.0
[BLOCKEND:FORMAT]
```

Блок INFORMATION

```
[BLOCK:INFORMATION]
Root=TestClasses.General
Types=39
Objects=45
[BLOCKEND:INFORMATION]
```

Блок TYPES

```
[BLOCK:TYPES]
TYPE~0=TestClasses.Structured
TYPE~1=String
TYPE~2=DateTime
TYPE~3=UInt64
TYPE~4=TestClasses.Struct
TYPE~5=Double
TYPE~6=TestClasses.Enumerated
TYPE~7=Object
TYPE~8=TestClasses.Primitive
TYPE~9=Int32
[BLOCKEND:TYPES]
```

Описания типов TYPENAME

Простые

```
Int32
Double
String
DateTime
TestClasses.Enumerated
```

Массивы

```
Object[,]
Object[][]
Double[,]
String[][]
```

Обобщенные типы

```
Nullable<Double>
System.Collections.Generic.List<Double>
System.Collections.Generic.Dictionary<String:Object>
```

Значения данных

Значения VALUE

```
VALUE=1
VALUE=2.2
VALUE=Three
```


Ссылки REF

```
REF=3
REF=NULL
```

Списки LIST

```
LIST<4:5>=
0.564041984064524 -0.524071000294793 0.917046885898824 0.742994267373809 -0.255815873507325
-0.874795514100602 -0.672671720233127 -0.0310939024347318 0.862441429804285 -0.246964344404156
0.0394569481906746 -0.644531836567694 0.47184890204661 0.581974272421549 0.762016043887481
0.0081122987941431 -0.925377672503413 0.76113086648338 -0.000737712253228606 -0.20026077386004

LIST<5:<3:1:4:NULL:4>:<<3:2:3>:<2>:<1:2:3:4>:<1:2:NULL:NULL>>>=
1.11 1.12 1.13
1.21 1.22
1.31 1.32 1.33
2.11 2.12
3.11
3.21 3.22
3.31 3.32 3.33
3.41 3.42 3.43 3.44
5.11
5.21 5.22

LIST<3:<2:NULL:3>>=
One Two
One Two Last
```

Списки ссылок REFLIST

```
REFLIST<3:2>=
12 NULL
NULL 13
14 16

REFLIST<4:<3:1:NULL:4>:<<3:2:3>:<2>:<1:2:NULL:NULL>>>=
17 NULL 18
19 20
NULL NULL NULL
21 22
23
24 25
```

Двоичные BINARY

```
BINARY=NULL

BINARY<0:0>=

BINARY<722:123>=
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789
01234567890123456789012345678901234567890123456789

01234567890123456789

01234567890123456789012345678901234567890123456789012345678901234567890123456789

0123456789
```

Элементные записи (member record)

```
DATA _size:Int32
/
VALUE=3
\

DATA strValue:String
```



```
/
REF=4
\

DATA regularDoubles:Double[,]
/
LIST<3:4>=
1.1 1.2 1.3 1.4
2.1 2.2 2.3 2.4
3.1 3.2 3.3 3.4
\

DATA dateData:TYPE~3
/
VALUE=634609728000000000
\

DATA doubleField:TYPE~5
/
VALUE=1.1
\

DATA data:TYPE~11
/
REFLIST<4:5>=
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
\
```

Объектные записи (object record)

```
OBJECT~14:TestClasses.Simple
{
DATA intValue:Int32
/
VALUE=1
\
DATA strValue:String
/
REF=15
\
DATA doubleValue:Double
/
VALUE=3.3
\
}

OBJECT~15:String
{
VALUE<1>=
2
}

OBJECT~16:DateTime
{
DATA dateData:UInt64
/
VALUE=32
\
}

OBJECT~6:TYPE~8
{
DATA privateInt:TYPE~9
/
VALUE=1
\
DATA privateDouble:TYPE~5
/
VALUE=2.2
\
}
```

