**Название статьи:** Сравнительный анализ эффективности популярных форматов сериализации данных.

**Цель статьи:** Представить читателю информацию об эффективности упаковки разнородных данных популярными методами сериализации в чистом виде, а также с применением zip архивации. Иными словами, мы оцениваем объем объектов в упакованном виде и их способность к дополнительному сжатию.

**Уровень сложности:** Для начинающих программистов.

Итак, начнем с того что же такое эта сериализация. Для тех, кто возможно не знает сериализация в программировании это - процесс перевода какой-либо структуры данных в последовательность битов. Ее основное назначение — сохранить состояние объекта для того, чтобы иметь возможность воссоздать его при необходимости. В общем и целом, при сериализации открытые поля и свойства объекта сериализуются в некое строковое представление, соответствующее определенному стандарту. Я выбрал 4 наиболее популярных, на сегодняшний день, формата:

1. XML (*e****X****tensible****M****arkup****L****anguage*) – всем известный расширяемый язык разметки. Объект сериализуется в XML-строку в соответствии с особым документом, составленным на языке XSD (язык определения схемы XML). Один из первых, широко распространенных, DOM форматов;
2. JSON (**J**ava**S**cript **O**bject **N**otation) — текстовый формат обмена данными, основанный на JavaScript и обычно используемый именно с этим языком. Как и многие другие текстовые форматы, JSON легко читается людьми;
3. Protobuf (**Proto**col **Buf**fers) - язык описания сообщений и данных, предложенный Google, как эффективная двоичная альтернатива текстовому формату XML;
4. Простая двоичная сериализация – стандартный, встроенный в .Net бинарный сериализатор.

Для тестирования мне потребовалось написать небольшую консольную программу (исходный код можно получить по [ссылке](https://github.com/merkator2160/SerializersCompare.git)). В качестве основы я выбрал платформу .Net Framework от Microsoft, язык программирования С#. На помощь мне пришли уже готовые библиотеками сериализации, полученные мною из Nugget. Например, для JSON я применял Json.Net версии 8.0.3 от Newtonsoft. Он очень популярен в наше время в среде веб разработчиков. Для XML в дело пошел популярный подход LINQ To XML. Работать же с Protobuf мне помогала библиотека protobuf-net версии 2.0.0.688. Для простой двоичной сериализацией я применил встроенный в среду .Net штатный сериализатор BinaryFormatter. Для того чтобы произвести сжатие я ограничился стандартным программным механизмом создания Zip архивов уже имеющимся в .Net.

Теперь поговорим немного о данных, т.к. это важно. Большинство типов данных обрабатываются каждым сериализатором по-своему и от того конечный результат получается разный. В моем случае набором данных для сериализации явилась коллекция объектов, генерируемая программно таким образом, что значения свойств у объектов коллекции случайны. Я ограничился размером коллекции в 100000 экземпляров. Для проведения теста этого достаточно, а также во время работы с такой коллекцией не выделяется слишком много памяти.

При помощи класса StopWatch я замерял время выполнения нужных мне участков кода. В написанной мною программе сериализация в начале производится в память, а уже лишь за тем данные пишутся на диск. Это позволяет получать чистое время без влияния задержек дисковой системы. В иной ситуации можно писать поток прямо на диск, заменив MemoryStream на файловый или любой другой. По правде сказать, для чистоты эксперимента нужно было бы еще прогнать алгоритм раз 100 и получить усредненное значение временных результатов, но я этого делать не стал. Колебания 2-5 миллисекунд меня не особенно интересовали, главное было приблизительно сравнить качество работы.

Результат работы программы.

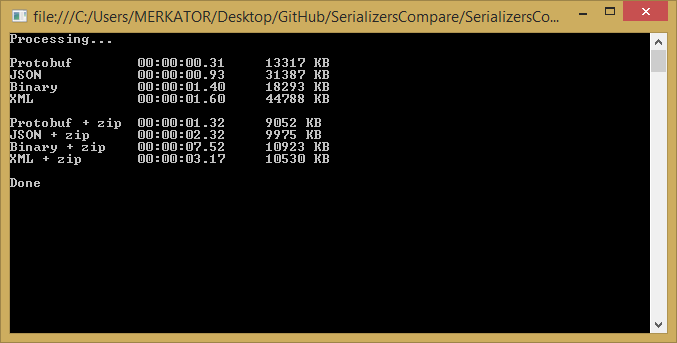


Рис. 1

На рабочем столе появится папка со следующим содержимым:

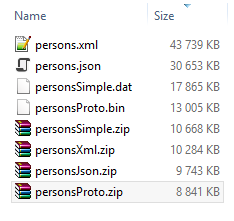


Рис. 2

Приведенные на первом изображении (Рис. 1) временные результаты, при запуске программы на вашей машине, могут численно отличаться, но общая тенденция сохранится. Это происходит из-за различий в “железе”. К слову в моем случае, замеры проводились на немного устаревшем на сегодняшний день Intel Core I7 2600K под разгоном. Установленная память имеет объем 16 Gb и частоту 1866 МГц, пропускная способность 15000 Mb/c.

Общая таблица результатов работы.

|  |  |  |
| --- | --- | --- |
| Тип сериализации | Затраченное время, (сек) | Размер данных на выходе, (МБ) |
| Protobuf | 0,31 | 13,317 |
| JSON | 0,93 | 31,387 |
| Binary | 1,40 | 18,293 |
| XML | 1,60 | 44,788 |
| Protobuf + zip | 1,32 | 9,052 |
| JSON + zip | 2,32 | 9,975 |
| Binary + zip | 7,52 | 10,923 |
| XML + zip | 3,17 | 10,53 |

Таб. 1

Сопоставив все полученные данные, мы имеем бесспорного лидера по скорости работы и наименьшему объему данных на выходе без использования алгоритмов сжатия. Этот лидер Protobuf, а вот его бинарный собрат показал куда более скромные результаты и оказался на третьем месте по скорости и на втором объему данных. Работать программисту с этими сериализаторами легко, но к сожалению бинарный формат не читабелен и ручному редактированию не подлежит. Из читабельных форматов лучше всех себя показал JSON. Втрое и третье место по скорости и объему данных соответственно. В работе использовать его даже еще проще чем первые два. У XML же все плохо он самый объемный и медленный. Сказывается огромное количество служебной разметки (тегов), а также, наверное, выбранная мною технология формирования документа LINQ To XML. Существуют альтернативные технологии, но при их использовании объем написанного программного кода оказывается существенно больше. На сколько я знаю они дают определенный прирост в скорости, но вот в объеме данных увы нет.

Что же касается возможности сжатия одного того же набора данных в разных форматах сериализации, то у JSON и Protobuf результат практически идентичен. Binary и XML также показали сопоставимые результаты, но все же объем данных у них на входе где-то на 10% больше чем у первых двух. Коэффициент сжатия для каждого формата сильно отличается. Вычислять каждый из них в данной статье я не буду, любой может прикинуть это сам, просто взглянув на цифры. Но должен заметить, что XML рекордсмен по подверженности сжатию. После сжатия объем XML файла снизился более чем в 4 раза. Protobuf сжался незначительно, а время, затраченное на его сжатие, в 3 раза превысило время самой сериализации. JSON как и любой друнгой текст сжимается также хорошо, Binary чуть хуже. Но что более всего меня ужаснуло, так это то сколько времени тратится на сжатие обычного бинарного формата, 7,52 секунды — это слишком мнго.

Думаю, что более детально анализировать результат не имеет смысла т.к. каждый формат больше приспособлен под свои задачи. Каждый волен сам решать, исходя из стоящих перед ним задач, в каком случае какой формат будет наиболее удобен. Данная статья дополнительно поможет вам в этом. Считаю, что прочесть данную публикацию стоит многим, и не только начинающим, разработчикам т.к. дискуссии о преимуществе того или иного вида сериализации возникают регулярно.

В конце хочу добавить, что приложение из данной статьи будет полезно для начинающих программистов. Для тех, кто хочет увидеть пример работы с типом данных Stream, программным сжатием потока на лету и записью на диск. Можно также ознакомиться с форматированным выводом информации в консоль и применением технологии LINQ для создания XML документов.

Исходники тут: <https://github.com/merkator2160/SerializersCompare.git>