JSON Tools for Pascal

(перевод на русский Zoltanleo aka Док)

Ветка обсуждения на форуме Лазаруса.

Мы решили написать небольшой, но способный парсер JSON и конструктор JSON. Парсер JSON преобразует простой текст в формате объектной нотации Javascript в набор узлов, которые можно запрашивать. Конструктор JSON позволяет динамически создавать узлы и управлять ими, а затем выводить эти узлы как допустимый текст JSON.

Обновления

Обновление для JsonTools было размещено в его репозитории на github. Это обновление устраняет проблему с экранированными двойными кавычками `"\""` и добавляет некоторые удобства. Наиболее значительным удобством является добавление метода `Force`.

Метод `Force` позволяет пользователям добавлять или изменять цепочку объектов, даже если их не существует. `Force` будет искать указанный путь и возвращать узел. Если какой-либо части пути не существует, он будет создан для вас.

```
{ Заставляет ряд узлов появиться и вернуть конечный узел }
function Force(const Path: string): TJsonNode;
```

Рассмотрим следующий пример:

```
var
  N, C: TJsonNode;
begin
  N := TJsonNode.Create;
  N.Force('customer/first').AsString := 'James';
  N.Force('customer/last').AsString := 'Milligan';
  N.Force('address/street').AsString := '123 Skippy Lane';
  N.Force('address/state').AsString := 'FL';
  C :=  N.Force('cart').Add;
  C.Force('name').AsString := 'shorts';
  C.Force('price').AsNumber := 25.12;
  N.Force('cart').Add.Add('name', 'gloves').Parent.Add('price', 30);
  N.Force('total').AsNumber := 55.12;
  WriteLn(N.Value);
  N.Free;
end;
```

Это приведет к следующему выводу:

```
{
    "customer": {
        "first": "James",
        "last": "Milligan"
```

```
    },
    "address": {
        "street": "123 Skippy Lane",
        "state": "FL"
    },
    "cart": [
        {
            "name": "shorts",
            "price": 25.12
        },
        {
            "name": "gloves",
            "price": 30
        }
    ],
    "total": 55.12
}
```

Еще одно дополнение - метод `Exists`. `Exist` проверит путь и вернет `True`, если найден узел, соответствующий заданному пути.

```
{ Ищет узел, используя строку Path, и возвращает True, если узел существует }
function Exists(const Path: string): Boolean;
```

Добавлена перегрузка метода `Add` без параметров. Перегруженный `Add` преобразует узел в массив, если это еще не сделано, и возвращает новый пустой узел объекта, принадлежащий массиву.

```
{ Конвертирует в массив и добавляет элементы }
function Add: TJsonNode; overload;
```

Наконец, была добавлена перегрузка метода `Find`, которая помещает найденный узел в выходной параметр и возвращает True, если узел, соответствующий пути, был найден.

```
{ Ищет указанный узел в строке Path и возвращает true, если он существует }
function Find(const Path: string; out Node: TJsonNode): Boolean; overload;
```

Зачем использовать JSON?

JSON - это стандартизированный формат текстовых данных, который становится все более распространенным. Он работает на любой компьютерной системе, он портативный, краткий, читаемый и редактируемый. Многие вызовы веб-API требуют либо содержимого JSON в качестве данных публикации, либо возврата содержимого JSON в качестве результатов. Многие настольные системы переходят на JSON в качестве формата файла для конфигураций, настроек, макетов или других данных, сохраняемых на диске.

Зачем писать парсер JSON, если он уже существует?

Побудительным мотивом к написанию этого парсера и конструктора стали две основные причины:

1. Нам нужен был очень простой интерфейс для работы с JSON в Паскале, который работает так, как мы привыкли. Подробнее об этом ниже.
2. Мы хотели написать довольно небольшое, но полное решение, чисто реализованное в одном классе без каких-либо зависимостей, кроме модулей `SysUtils` и `Classes`.

Как наш парсер JSON упрощает работу с JSON?

Наш парсер очень прост в использовании. Просто добавьте модуль `JsonTools` в свой раздел `uses`, после чего вы сможете проанализировать действительный документ JSON следующим образом:

```
N := TJsonNode.Create;
N.Parse(YourJsonText);
```

Когда вы закончите работать со своим `TJsonNode`, просто напишите `Free`. Вам нужно только освободить `TJsonNode`, если вы создали его напрямую.

```
N.Free;
```

Помимо вызова `Parse`, вы также можете использовать эти методы для анализа документа JSON.

```
N.Value := YourJsonText;
```

Или, если JSON находится в файле, вы можете написать:

```
N.LoadFromFile(YourJsonFileName);
```

Эти методы делают то же самое, что и `Parse`. Они строят дерево узлов JSON путем анализа данных JSON. Чтобы получить текстовые данные JSON из любого узла, вы можете просто написать:

```
WriteLn(N.Value);
```

И узлы конвертируются обратно в красиво отформатированный документ JSON. Если вы хотите распечатать каждый узел, то можно расписать весь документ:

```
N.Value := '[ "Hello", "World!" ]';
for C in N do
  WriteLn(C.Value);
WriteLn(N.Value);
```

В результате получится следующий результат:

```
"Hello"
"World!"
[
    "Hello",
    "World!"
]
```

Если вы не уверены, что ваш исходный текст JSON валиден(корректен), вы можете написать этот фрагмент кода, чтобы проверить его правильность:

```
S := '{ это не валидный json }';
if N.TryParse(S) then
  // Делаем что-нибудь со своими данными
else
  // Ваши входные данные не содержали валидного текста JSON
```

Другие возможности JSON

Теперь, когда мы знаем, как этот объект JSON может анализировать текст, давайте рассмотрим пример того, как он может перемещаться и управлять узлами JSON. Вот немного текстовых данных JSON, которые мы будем использовать в нескольких следующих примерах.

```
{
    "name": "Alice Brown",
    "sku": "54321",
    "price": 199.95,
    "shipTo": {
        "name": "Bob Brown",
        "address": "456 Oak Lane",
        "city": "Pretendville",
        "state": "HI",
        "zip": "98999"
    },
    "billTo": {
        "name": "Alice Brown",
        "address": "456 Oak Lane",
        "city": "Pretendville",
        "state": "HI",
        "zip": "98999"
    }
}
```

Если бы эти данные были в файле с именем `orders.json`, мы могли бы использовать их так:

```
N.LoadFromFile('orders.json');
WriteLn('The price is ', N.Find('price').Value);
WriteLn('The customer is ', N.Find('billTo/name').Value);
```

Результатом будет:

```
The price is 199.95
The customer is "Alice Brown"
```

Если вы хотите изменить "shipTo" в соответствии с "billTo", вы можете просто написать:

```
N.Find('shipTo').Value := N.Find('billTo').Value;
```

И все узлы из-под "billTo" будут скопированы поверх "shipTo". Это означает, что мой объект может анализировать текст на уровне узла, что позволяет вам составлять документ JSON из нескольких источников или легко перемещать и копировать данные. Вы также можете загружать и сохранять различные дочерние узлы из отдельных файлов или потоков. Ну не круто?

Если вы просто хотите изменить имя клиента, вы можете написать:

```
N.Find('billTo/name').AsString := 'Joe Schmoe';
```

Обратите внимание, что мы используем `AsString` выше вместо `Value`. Это связано с характером свойства `Value`. Всякий раз, когда вы задаете значение `Value`, текущий узел будет пытаться проанализировать входящее значение, как фрагмент JSON. Введенное значение `'Joe Schmoe'` невалидно в формате JSON. Чтобы сообщить нашему объекту, что мы собираемся установить строковое значение, мы можем использовать свойство `AsString`.

Если вы не хотите использовать `AsString`, вам нужно будет вместо этого написать этот оператор, что делает `'Joe Schmoe'` допустимой строкой JSON:

```
N.Find('billTo/name').Value := '"Joe Schmoe"';
```

Обратите внимание, что мы заключили новое имя в символы `"` двойных кавычек. Двойные кавычки необходимы для создания строк JSON. Поскольку мы добавили символы двойных кавычек, второй оператор теперь передает допустимое значение в наше свойство `Value`.

Вот несколько примеров допустимых значений, которые вы можете использовать при установке свойства `Value` узла.

```
N.Value := 'true';   // узел становится узлом типа bool
N.Value := '3.1415'; // узел становится узлом типа number
N.Value := 'null';   // узел становится узлом типа null
N.Value := '[]';     // узел становится узлом типа array
N.Value := '{}';     // узел становится узлом типа object
```

В приведенных выше примерах `N` заменяется на другой тип узла с каждым оператором. Если `N` является корневым узлом, первые три оператора завершатся ошибкой, поскольку корневой узел должен быть либо объектом, либо массивом.

В качестве альтернативы, пять приведенных ниже операторов делают то же самое, что и пять приведенных выше операторов, но с небольшой дополнительной безопасностью типов:

```
N.AsBoolean := True;
N.AsNumber := 3.1415;
N.AsNull;
N.AsArray;
N.AsObject;
```

Обратите внимание, что `AsNull`, `AsArray` и `AsObject` не имеют оператора присваивания `:=`. Это потому, что значение каждого из этих типов фиксировано. `Null`-евой узел равен нулю, узел массива - это массив, а узел объекта - это объект. Эти три оператора способны к преобразованию типа узла и сброса его значения.

Переход на другой узел

Как отмечалось в последнем разделе, попытка установить для корневого узла значение, не являющееся объектом или массивом, является ошибкой. Стандарт JSON требует, чтобы корневой уровень всех объектов JSON был либо объектом, либо массивом.

Чтобы переключиться на другой узел, вы можете написать:

```
N := N.Find('shipTo/name');
N.Add('first', 'Joe');
N.Add('last', 'Schmoe');
```

Если `shipTo/name` был строковым, он переключится на тип объекта при добавлении узла. После временного переключения узлов, чтобы получить ссылку на наш корень, мы могли бы написать:

```
WriteLn(N.Value);
N := N.Root;
```

И теперь мы можем быть уверены, что `N` снова относится к корневому узлу. Результатом приведенного выше фрагмента было бы:

```
{
    "first": "Joe",
    "last": "Schmoe"
}
```

Добавление и удаление узлов

Узлы можно добавлять или удалять программно, используя специальные методы. Если вы хотите добавить значение к объекту (или массиву), вы можете написать:

```
N.Add('note', 'Customer requested we change the ship to address');
```

Это добавит к нашему объекту JSON следующее:

```
"note": "Customer requested we change the ship to address"
```

В нашем коде на паскале, если `N` является массивом, первый параметр будет проигнорирован, поскольку массивы в JSON не имеют именованных значений для своих дочерних узлов.

Также обратите внимание, когда мы используем метод `Add`, нам не нужно использовать строки JSON. Мы можем просто использовать обычные строки, и они будут внутренне преобразованы в формат JSON. Это внутреннее форматирование позволяет нам добавлять строки с несколькими строками или другим форматом кодировки, и внутри они будут сохранены как строки, совместимые с JSON.

Метод `Add` является перегруженным (overload), что позволяет добавлять логические значения, числа, строки и многое другое:

```
N.Add('discount', -8.50);   // добавляет узел типа number
N.Add('verified', True);    // добавляет узел типа bool
N.Add('wishlist', nkArray); // добавляет узел типа array
```

Особенностью JSON является то, что он разрешает только один узел с заданным именем на определенном уровне. Это означает, что вы снова добавляете то же имя, тогда будет возвращен существующий узел, и его значение будет изменено, чтобы соответствовать второму аргументу `Add`.

Дочерний узел можно удалить либо с помощью метода `Delete`, либо с помощью метода `Clear`.

```
N.Delete('note');   // удаляет узел 'note'
N.Delete('shipTo'); // удаляет узел 'shipTo'
```

Чтобы удалить элементы в массиве, просто используйте строковое представление индекса массива:

```
N.ToArray.Add('note', 'Please call the customer');
N.Delete('0');
```

В приведенном выше примере наш `N` преобразуется в массив и добавляется элемент. Поскольку массивы не используют имена для индексации своих дочерних узлов, первый аргумент `'note'` отбрасывается. Вторая строка в примере удаляет элемент 0 (массивы отсчитываются с 0). Если `N` не был массивом перед первой строкой, то его значения отбрасываются, и он будет содержать только нашу строку `'Please call the customer'`.

Чтобы удалить все дочерние узлы объекта или массива, просто введите:

```
N.Clear;
```

Это удалит каждый узел ниже текущего узла. `Clear` не влияет на `null`-евые, логические, числовые или строковые узлы.

Основные свойства каждого узла JSON

Каждый узел в своей основе хранит следующую информацию:

```
property Root: TJsonNode     // корень вашего документа JSON (только для чтения)
property Parent: TJsonNode   // родительский элемент вашего узла JSON (только для
чтения)
property Kind: TJsonNodeKind // тип узла, такие как объект, массив, строка и т.д.
(чтение/запись)
property Name: string        // Имя узла (чтение/запись)
property Value: string       // Значение узла в форме JSON (чтение/запись)
```

Свойство `Kind` использует перечислимый тип, определенный как одно из следующих возможных значений:

```
nkObject,  nkArray, nkBool, nkNull, nkNumber, nkString
```

Когда для `Kind` установлено значение, отличное от текущего, узел сбрасывается. Для типов `object` и `array` это означает, что все дочерние элементы будут удалены. Для типов `bool`, `null`, `number` и `string` значения по умолчанию после сброса - `false`, `null`, `0` и `''` соответственно.

Если задано повторяющееся `Name`, то существующий узел с таким же именем удаляется и заменяется текущим узлом. Вы можете задать имя узла только в том случае, если родитель является объектом, в противном случае любые попытки изменить имя игнорируются.

Когда задано `Value`, внутренне переданное вами значение анализируется как JSON, и вся часть значения узла заменяется, потенциально изменяя тип узла в процессе. Важно отметить, что значение корневого узла должно быть допустимым объектом или массивом JSON. Иного быть не может.

Вот несколько примеров того, как можно использовать свойство `Value`.

```
N.Child('sku').Value := '[ "54321", "98765" ]'; // Sku становится массивом значений
N.Find('shipTo/name').Value := '{ "id": 5028, "primary": true }'; // name становится
типом object
```

Помимо возможности использовать перечислитель `for ... in` на любом узле, вы также можете использовать эти три свойства для перечисления дочерних узлов:

```
function Child(Index: Integer): TJsonNode;     // получает дочерний узел по индексу
function Child(const Name: string): TJsonNode; // получает дочерний узел по имени
property Count: Integer;                       // возвращает количество дочерних узлов
```

Обратите внимание, что `Child` - это не то же самое, что и ранее упомянутый метод `Find`. Метод `Find` ищет путь, а метод `Child` ищет точное совпадение имени непосредственно под текущим узлом.

Исходный код

Весь исходный код этого синтаксического анализатора находится в свободном доступе по лицензии GPLv3. Он [размещен на github здесь](), а источник интерфейса указан здесь для справки. Не стесняйтесь присылать мне свои комментарии или отзывы.

```
type
  TJsonNode = class
  public
    { A parent node owns all children. Only destroy a node if it has no parent.
      To destroy a child node use Delete or Clear methods instead. }
    destructor Destroy; override;
    { GetEnumerator adds 'for ... in' statement support }
    function GetEnumerator: TJsonNodeEnumerator;
    { Loading and saving methods }
    procedure LoadFromStream(Stream: TStream);
    procedure SaveToStream(Stream: TStream);
```

```
    procedure LoadFromFile(const FileName: string);
    procedure SaveToFile(const FileName: string);
    { Convert a json string into a value or a collection of nodes. If the
      current node is root then the json must be an array or object. }
    procedure Parse(const Json: string);
    { The same as Parse, but returns true if no exception is caught }
    function TryParse(const Json: string): Boolean;
    { Add a child node by node kind. If the current node is an array then the
      name parameter will be discarded. If the current node is not an array or
      object the Add methods will convert the node to an object and discard
      its current value.

      Note: If the current node is an object then adding an existing name will
      overwrite the matching child node instead of adding. }
    function Add(const Name: string; K: TJsonNodeKind = nkObject): TJsonNode; overload;
    function Add(const Name: string; B: Boolean): TJsonNode; overload;
    function Add(const Name: string; const N: Double): TJsonNode; overload;
    function Add(const Name: string; const S: string): TJsonNode; overload;
    { Convert to an array and add an item }
    function Add: TJsonNode; overload;
    { Delete a child node by index or name }
    procedure Delete(Index: Integer); overload;
    procedure Delete(const Name: string); overload;
    { Remove all child nodes }
    procedure Clear;
    { Get a child node by index. EJsonException is raised if node is not an
      array or object or if the index is out of bounds.

      See also: Count }
    function Child(Index: Integer): TJsonNode; overload;
    { Get a child node by name. If no node is found nil will be returned. }
    function Child(const Name: string): TJsonNode; overload;
    { Search for a node using a path string and return true if exists }
    function Exists(const Path: string): Boolean;
    { Search for a node using a path string }
    function Find(const Path: string): TJsonNode; overload;
    { Search for a node using a path string and return true if exists }
    function Find(const Path: string; out Node: TJsonNode): Boolean; overload;
    { Force a series of nodes to exist and return the end node }
    function Force(const Path: string): TJsonNode;
    { Format the node and all its children as json }
    function ToString: string; override;
    { Root node is read only. A node the root when it has no parent. }
    property Root: TJsonNode read GetRoot;
    { Parent node is read only }
    property Parent: TJsonNode read FParent;
    { Kind can also be changed using the As methods.

      Note: Changes to Kind cause Value to be reset to a default value. }
    property Kind: TJsonNodeKind read FKind write SetKind;
    { Name is unique within the scope }
    property Name: string read GetName write SetName;
    { Value of the node in json e.g. '[]', '"hello\nworld!"', 'true', or '1.23e2' }
    property Value: string read GetValue write Parse;
    { The number of child nodes. If node is not an object or array this
      property will return 0. }
    property Count: Integer read GetCount;
    { AsJson is the more efficient version of Value. Text returned from AsJson
```

```
      is the most compact representation of the node in json form.

      Note: If you are writing a services to transmit or receive json data then
      use AsJson. If you want friendly human readable text use Value. }
    property AsJson: string read GetAsJson write Parse;
    { Convert the node to an array }
    property AsArray: TJsonNode read GetAsArray;
    { Convert the node to an object }
    property AsObject: TJsonNode read GetAsObject;
    { Convert the node to null }
    property AsNull: TJsonNode read GetAsNull;
    { Convert the node to a bool }
    property AsBoolean: Boolean read GetAsBoolean write SetAsBoolean;
    { Convert the node to a string }
    property AsString: string read GetAsString write SetAsString;
    { Convert the node to a number }
    property AsNumber: Double read GetAsNumber write SetAsNumber;
  end;
```

Сравнение

Мы добавили новую страницу, чтобы протестировать наш новый синтаксический анализатор по сравнению с несколькими другими синтаксическими анализаторами на основе паскаля. Вы можете прочитать больше об этих тестах здесь.

Результаты показывают, что наш парсер значительно лучше, чем другие парсеры, которые нам удалось найти и протестировать. Лучше и по скорости, и по корректности. Хотя мы считаем, что наши тесты честны, они ни в коем случае не являются исчерпывающими, некоторые, пожалуйста, помните об этом факте при рассмотрении наших результатов.

См.также

- Скорость и корректность парсера JSON(англ)
- Учебный центр Lazarus(англ)