SAN_Protobuffer v.1.0

Содержимое

Введение

Библиотека **SAN_Protobuffer v.1.0** реализована для сериализации и десериализации данных описанных на языке Protocol Buffers.

Исходный код написан на Delphi и поддерживает все версии начиная с Delphi XE2 (Windows 32/64).
(Тестировал на Delphi XE2, Delphi 10.1 Berlin, Delphi 11.3)

Реализация выполнена в виде основного файла `semin64.protobuf.pas` и одного вспомогательного `semin64.memory.pas`. Библиотеку не нужно устанавливать, здесь нет dpk файлов для создания и установки пакета. Вы просто подключаете эти файлы к вашему проекту или указываете путь поиска в настройках.

Что реализовано

Можно создавать объекты в соответствии с описанными типами на языке `Protocol Buffers` (Proto2, Proto3) для сериализации и десериализации данных. Объекты можно описывать вручную или можно использовать программу pbpgui.exe, которая автоматизирует этот процесс, разбирая **.proto* файлы и затем создает файл с описанием типов.

Что не реализовано

- option `allow_alias` в `enum`
- описание поля с устаревшей конструкцией `group`
- описание поля с конструкцией `oneof`
- некоторые ключевые слова синтаксиса proto файлов, которые не используются в сериализации бинарных proto пакетов

`extend`, `reserved`, `service`, и т. д.

Установка

Установка как таковая не требуется, достаточно клонировать данный проект или просто скопировать папки проекта в один каталог.

- DOCS - документация
- SOURCE - исходный код библиотеки
- EXAMPLES - примеры
- TOOLS - утилита `pbpui.exe`

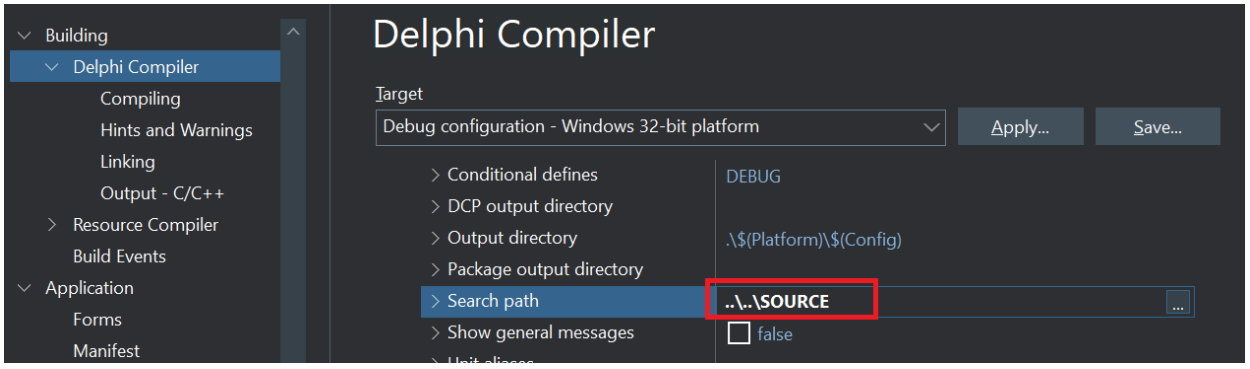
Примеры

Проект снабжен примерами, которые находятся в папке EXAMPLES.
Каждый пример снабжен подробным описанием. Последний пример демонстрирует получение данных из системы "Контур Диадок"

Example01 - generic

Первый пример - это просто программа-шаблон для всех следующих примеров.
Это будет простая форма с двумя кнопками и одним текстовым окном для вывода данных.

1. Мы создаем приложение File -> New -> Windows VCL Application Delphi
2. Прописываем в настройках проекта путь к нашей библиотеке
   Project -> Options



3. В разделе uses добавляем `semin64.protobuf`

```
uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Vcl.StdCtrls, semin64.protobuf;
```

4. На форму бросим две кнопки и один TMemo

Структура кода будет такой:

В событии формы `OnCreate` мы будем описывать тип `Protocol Buffrer` с которым будем работать

В событии `OnClick` кнопки **Save** мы будем формировать данные в соответствии с нашим типом и затем сериализуем их в бинарный файл.

В событии `OnClick` кнопки **Load** мы загрузим ранее сохраненный бинарный файл, десериализуем его и прочитанные данные отобразим в компоненте TMemo.

Полный код доступен в `EXAMPLES\Example01_generic`

Example02 - первый message

В этом примере мы опишем простой **message** и выполним сериализацию и десериализацию.

Описание типа

Перед тем, как начать работать с данными, описанными на языке `Protocol buffers`, мы должны описать эти типы с помощью объектов класса `TsanPBMessageType`. Каждый `message` должен иметь свой `TsanPBMessageType` объект, в котором указываются поля данного типа.

Возьмём простой тип `Team`

```
message Team {
  optional int32 team_id = 1;
  repeated string members = 2;
}
```

Для этого типа мы создадим соответствующий объект `FTeamType: TsanPBMessageType` и укажем состав полей в точном соответствии с описанием типа `Team`.

```
FTeamType:= TsanPBMessageType.Create(nil, 'Team');
FTeamType.AddFieldDef(ftoOptional, ftInt32,  nil, 'team_id', 1);
FTeamType.AddFieldDef(ftoRepeated, ftString, nil, 'members', 2);
```

Остановимся здесь по подробнее. Класс `TsanPBMessageType` имеет конструктор:

```
constructor Create(OwnerA: TsanPBCustomType; TypeNameA: string);
```

`OwnerA` - родительский тип и указывается в случае, если наш тип является подтипом родителя, в нашем случае значение будет `nil`

`TypeNameA` - наименование типа

Для описания полей используется функция:

```
function AddFieldDef(Option: TsanPBFieldTypeOption;
                     FieldType:  TsanPBFieldType;
                         CustomType: TsanPBCustomType;
                     FieldName: string;
                     StoreIndex: integer): integer;
```

`Option` - метка поля, может принимать значения:

```
TsanPBFieldTypeOption = (ftoRequired, ftoOptional, ftoRepeated);
```

> В документации еще есть метка *map* - для работы с такими полями ниже будет отдельный пример

`FieldType` - тип поля, может принимать значения:

```
  TsanPBFieldType = (ftInt32, ftSint32, ftFixed32, ftSfixed32, ftUInt32,
                     ftInt64, ftSint64, ftFixed64, ftSfixed64, ftUInt64,
                     ftFloat, ftDouble, ftBoolean, ftString, ftBytes,
                     ftMessage, ftEnum);
```

Типы `ftInt32 .. ftBytes` соответствуют скалярным типам описанных в документации `Protocol Buffers`. Тип `ftMessage` указывается для поля `message`, а тип `ftEnum` соответственно для `enum`.
Ниже, будут примеры как работать с `ftMessage` и `ftEnum`.

`CustomType` - объект, описывающий тип поля (указывается, только когда `FieldType` равен `ftMessage` или `ftEnum`)

`FieldName` - имя поля

`StoreIndex` - индекс поля

Заполнение данными и сериализация

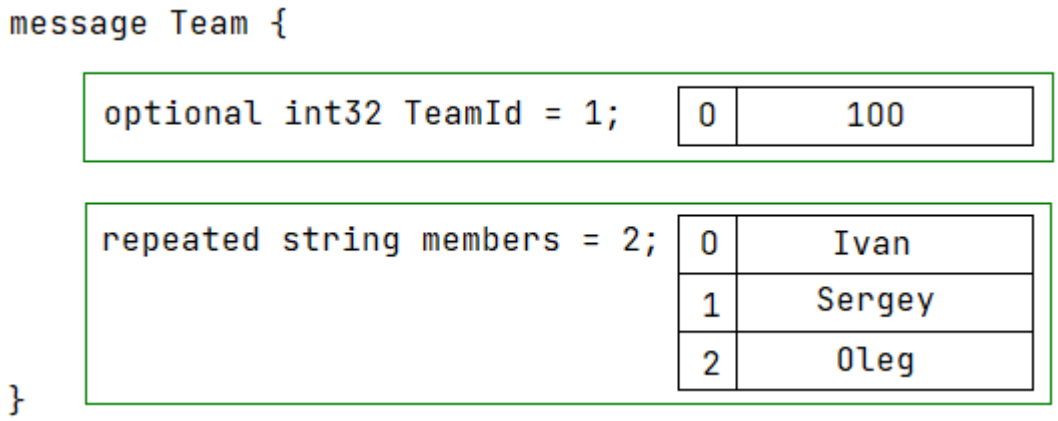
После того как мы определили тип `Team` с помощью класса `TsanPBMessageType`, мы можем начать работать с данными. Однако, поскольку класс `TsanPBMessageType` создан только для описания типа, он не может самостоятельно работать с данными. Вместо этого, он может создать объект класса `TsanPBMessage`, который специализируется именно для работы с данными.

```
var
  TeamMsg: TsanPBMessage;
begin
  TeamMsg:= FTeamType.CreateInstance;
```

Класс `TsanPBMessage` позволяет хранить данные в памяти в соответствии с описанием типа класса `TsanPBMessageType`. Более того, таким образом мы можем создать сколько угодно объектов `TsanPBMessage` от одного типа `TsanPBMessageType`, и все они будут иметь независимые данные друг от друга.

Данные в объекте `TsanPBMessage` распределены по полям. Сам объект `TsanPBMessage` содержит набор полей, а каждое поле представляет собой набор данных. Данные в каждом поле могут быть представлены списком значений. В некоторых случаях в списке может быть не более одного значения (`optional`) или строго одно значение (`required`).

Ниже на рисунке это показано более наглядно.



Каждое поле представляет собой объект класса `TsanPBField`, который отвечает за свои данные и обеспечивает нас набором функций для добавления или чтения данных. Прежде чем начать работать с полем, нам необходимо получить его функцией `FieldByName`.

```
TeamId:= TeamMsg.FieldByName('team_id');
```

> Если мы укажем несуществующее имя поля, то функция вызовет исключение.

Как только мы получили объект `TsanPBField` мы можем добавлять данные. Для этого предназначены функции `AppendValueAs ...` .

```
function AppendValueAsInt32(Value: integer): integer;
function AppendValueAsUInt32(Value: cardinal): integer;
function AppendValueAsInt64(Value: Int64): integer;
function AppendValueAsUInt64(Value: UInt64): integer;
function AppendValueAsFloat(Value: single): integer;
function AppendValueAsDouble(Value: double): integer;
function AppendValueAsBoolean(Value: Boolean): integer;
function AppendValueAsString(Value: string): integer;
function AppendValueAsBytes(Value: TBytes): integer;
function AppendValueFromStream(Stream: TStream): integer;
```

Мы должны использовать ту функцию, которая соответствует нашему типу поля. Код будет таким:

```
TeamIdField:= TeamMsg.FieldByName('team_id');
TeamIdField.AppendValueAsInt32(100);
```

После того, как мы заполним данными все поля нашего `TsanPBMessage` мы можем сериализовать данные в бинарный файл.

Для этого мы можем использовать два метода класса `TsanPBMessage`

```
procedure SaveToStream(Stream: TStream);
procedure SaveToFile(FileName: string);
```

Полный код заполнения и сериализации представлен ниже:

```
procedure TForm1.btnSaveClick(Sender: TObject);
var
  TeamMsg: TsanPBMessage;
begin

  TeamMsg:= FTeamType.CreateInstance;

  try

    TeamMsg.FieldByName('team_id').AppendValueAsInt32(100);

    TeamMsg.FieldByName('members').AppendValueAsString('Ivan');
    TeamMsg.FieldByName('members').AppendValueAsString('Sergey');
    TeamMsg.FieldByName('members').AppendValueAsString('Oleg');

    TeamMsg.SaveToFile('team.bin');

  finally
    TeamMsg.Free;
  end;

end;
```

Десериализация и чтение данных

Когда мы выполним сериализацию, описанную в предыдущем разделе, на диске у нас должен быть сформирован бинарный файл `team.bin`, и теперь наша задача - загрузить его и прочитать данные. Для этого нам понадобится, так же как и при заполнении данными, объект класса `TsanPBMessage` и его метод `LoadFromFile`.

```
var
  TeamMsg: TsanPBMessage;
begin

  TeamMsg:= FTeamType.CreateInstance;
  try
    TeamMsg.LoadFromFile('team.bin');
        // ...
        //читаем данные
        // ...
  finally
    TeamMsg.Free;
  end;

end;
```

После того, как файл загружен мы можем прочитать значения полей с помощью функций класса `TsanPBField`, выбрав ту функцию, которая соответствует типу поля.

```
function GetValueAsInt32(RecordIndex: integer = 0): integer;
function GetValueAsUInt32(RecordIndex: integer = 0): cardinal;
function GetValueAsInt64(RecordIndex: integer = 0): Int64;
function GetValueAsUInt64(RecordIndex: integer = 0): UInt64;
function GetValueAsFloat(RecordIndex: integer = 0): single;
function GetValueAsDouble(RecordIndex: integer = 0): double;
function GetValueAsBoolean(RecordIndex: integer = 0): Boolean;
function GetValueAsString(RecordIndex: integer = 0): string;
function GetValueAsBytes(RecordIndex: integer = 0): TBytes;
```

Так как данные поля - это список значений, то передаваемый параметр `RecordIndex` указывает на индекс значения из этого списка. Число значений в списке можно определить через свойство `RecordCount`.

```
// MembersField: TsanPBField;
// MemberName: string
MembersField:= TeamMsg.FieldByName('members');

for I := 1 to MembersField.RecordCount do begin
    MemberName:= MembersField.GetValueAsString(I-1);

end;
```

Справедливо будет заметить, что такой способ чтения нужен только для полей с меткой `repeated`. Для других меток полей у нас будет список из одного значения или вообще пустой список. В этих случаях нам нужно передать `RecordIndex` равный `0` или вообще вызвать функцию без параметров. Если список значений будет пуст, то

функции `GetValueAs ...` возвратит значение по умолчанию. Подробнее об этом будет написано в Example03 - packed, default

```
TeamId:= TeamMsg.FieldByName('team_id').GetValueAsInt32;
```

Полный код загрузки файла **team.bin** и чтения значений с отображением их в **TMemo** представлен ниже:

```
procedure TForm1.btnLoadClick(Sender: TObject);
var
  TeamMsg: TsanPBMessage;
  TeamId: integer;
  MembersField: TsanPBField;
  MemberName: string;
  I: integer;
begin

  TeamMsg:= FTeamType.CreateInstance;

  try

    mmContents.Clear;
        TeamMsg.LoadFromFile('team.bin');

    TeamId:= TeamMsg.FieldByName('team_id').GetValueAsInt32;
    mmContents.Lines.Add('team_id: ' + IntToStr(TeamId));

    MembersField:= TeamMsg.FieldByName('members');
    for I := 1 to MembersField.RecordCount do begin
      MemberName:= MembersField.GetValueAsString(I-1);
      mmContents.Lines.Add('member: ' + MemberName);
    end;

  finally
    TeamMsg.Free;
  end;

end;
```

Полный код доступен в `EXAMPLES\Example02_first_message`

Example03 - packed, default

Рассмотрим такое сообщение:

```
message MyMessage {
  optional int32 per_page = 1 [default = 10];
```

```
  repeated int32 samples = 2 [packed = true];
}
```

Здесь `default` и `packed` являются дополнительными опциями, которые указываются при объявлении полей `per_page` и `samples` соответственно.

Default

Опция default указывается только для полей со скалярными типами значений и с меткой **optional**. Эта опция определяет значение поля в том случае, если при сериализации значение поля не было указано. Если данная опция не указана, то для числовых типов значение по умолчанию будет 0, а для **ftString** — пустая строка.

Packed

Опция Packed указывается только для полей со скалярными числовыми типами значений и с меткой **repeated**. Данная опция позволяет уменьшить размер бинарного файла при сериализации.

Когда мы описываем тип с помощью `TsanPBMessageType`, мы с помощью функции `AddFieldDef` добавляем определение поля, создавая тем самым объект класса `TsanPBFieldDef`. Все основные характеристики поля мы передаем через параметры `AddFieldDef`, но если нам нужно получить доступ к дополнительным свойствам, то нам нужен сам объект `TsanPBFieldDef`, который мы можем получить через функцию `FieldDefByName` и затем установить дополнительные свойства для него.

```
PerPageFieldDef:= FMyMessageType.FieldDefByName('per_page');
PerPageFieldDef.DefaultValue:= 10;
```

Ниже представлен код, касательно нашего примера, который описывает тип `MyMessage` с указанием дополнительных опций:

```
procedure TForm1.FormCreate(Sender: TObject);
begin

  FMyMessageType:= TsanPBMessageType.Create(nil, 'MyMessage');

  FMyMessageType.AddFieldDef(ftoOptional, ftInt32,  nil, 'per_page', 1);
  FMyMessageType.FieldDefByName('per_page').DefaultValue:= 10;

  FMyMessageType.AddFieldDef(ftoRepeated, ftInt32, nil, 'samples', 2);
  FMyMessageType.FieldDefByName('samples').DataPacked:= True;

end;
```

Полный код доступен в `EXAMPLES\Example03_packed_default`

Example04 - поле с типом message

В этом примере мы разберем работу с полем, которое имеет тип message. Рассмотрим два типа:

```
message Team {
  optional int32 TeamId = 1;
  repeated string members = 2;
}

message Tournament {
        repeated Team teams = 1;
}
```

Здесь мы видим, что тип `Tournament` содержит внутри поле с типом `Team`

Описание типа

Описание типа `Team` было подробно рассмотрено в Example02 - первый message, а вот описание типа `Tournament`, а именно описание поля `teams`, несколько отличается. Поле описывается все той же функцией `AddFieldDef`, но во втором параметре мы указываем тип `ftMessage` и обязательно в третьем параметре указываем объект, который описывает наш тип `Team`.

```
  private
    FTournamentType: TsanPBMessageType;
    FTeamType: TsanPBMessageType;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin

  FTeamType:= TsanPBMessageType.Create(nil, 'Team');
  FTeamType.AddFieldDef(ftoOptional, ftInt32,  nil, 'team_id', 1);
  FTeamType.AddFieldDef(ftoRepeated, ftString, nil, 'members', 2);

  FTournamentType:= TsanPBMessageType.Create(nil, 'Tournament');
  FTournamentType.AddFieldDef(ftoRepeated, ftMessage, FTeamType, 'teams',
1);

end;
```

Заполнение данными и сериализация

Здесь приведен код для заполнения и сериализации данных. Сам код мы разберем ниже.

```
var
  TournamentMsg: TsanPBMessage;
  TeamMsg: TsanPBMessage;
begin

  TournamentMsg:= FTournamentType.CreateInstance;

  try

    TeamMsg:= TournamentMsg.MessageByName('teams'); // 1

    TeamMsg.Append; // 2
    TeamMsg.FieldByName('team_id').AppendValueAsInt32(1);
    TeamMsg.FieldByName('members').AppendValueAsString('Ivan');
    TeamMsg.FieldByName('members').AppendValueAsString('Sergey');
    TeamMsg.FieldByName('members').AppendValueAsString('Oleg');

    TeamMsg.Append;
    TeamMsg.FieldByName('team_id').AppendValueAsInt32(2);
    TeamMsg.FieldByName('members').AppendValueAsString('Dmitriy');
    TeamMsg.FieldByName('members').AppendValueAsString('Alexey');
    TeamMsg.FieldByName('members').AppendValueAsString('Andrey');

    TournamentMsg.SaveToFile('tournament.bin');

    ShowMessage('OK');

  finally
    TournamentMsg.Free;
  end;

end;
```

Посмотрим на строчку кода помеченную `//1`
Раньше для получения поля `TsanPBField` для работы с данными мы использовали такой подход:

```
var
  Field: TsanPBField;
begin
  Field:= TournamentMsg.FieldByName('teams');
```

В нашем случае, `Field` является объектом класса `TsanPBMessage` и чтобы использовать его методы нам нужно сделать приведение типа:

```
var
  TeamMsg: TsanPBMessage;
```

```
begin
  TeamMsg:= TsanPBMessage(TournamentMsg.FieldByName('teams'));
```

или мы можем использовать функцию `MessageByName` которая делает тоже самое, но выглядит короче и не требует приведения типа:

```
TeamMsg:= TournamentMsg.MessageByName('teams');
```

Теперь посмотрим на строчку кода `//2`
Здесь все просто, метод `Append` имеет тот же смысл, что и методы `AppendValueAs ...` , которые применяются для полей с простыми (скалярными типами)

Десериализация и чтение данных

```
procedure TForm1.btnLoadClick(Sender: TObject);
var
  TournamentMsg: TsanPBMessage;
  TeamMsg: TsanPBMessage;
  I: integer;
begin

  mmContents.Clear;

  TournamentMsg:= FTournamentType.CreateInstance;

  try

    TournamentMsg.LoadFromFile('tournament.bin');

    TeamMsg:= TournamentMsg.MessageByName('teams');

    for I := 1 to TeamMsg.RecordCount do begin
      TeamMsg.MoveTo(I-1);
      PrintTeamInfo(TeamMsg);
    end;

  finally
    TournamentMsg.Free;
  end;

end;
```

Здесь стоит рассмотреть код чтения данных:

```
for I := 1 to TeamMsg.RecordCount do begin
        TeamMsg.MoveTo(I-1);
```

```
    PrintTeamInfo(TeamMsg);
end;
```

Для простых полей, например, с типом `ftInt32`, мы могли считывать значения с помощью функции `GetValueAsInt32(RecordIndex: integer)`, где в параметре мы передавали индекс записи. Для полей типа `message` методы чтения несколько отличаются. Здесь мы должны сначала указать индекс записи с помощью процедуры `MoveTo(Index: integer)` и только потом мы можем обращаться к полям нашего типа и считывать их значения.
Вместо процедуры `MoveTo` можно использовать свойство `RecordIndex`, с помощью которого можно установить или получить индекс текущей записи.

Если наш тип не помечен меткой `repeated`, то вызывать процедуру `MoveTo(0)` не обязательно.
Индекс текущей записи по умолчанию всегда будет `0`. Только перед чтением нужно убедиться, что наш **message** не пустой. Проверить это можно с помощью свойства **RecordCount** или **IsEmpty**. В противном случае при попытке получить объект поля с помощью вызова функции **FieldByName** будет сгенерировано исключение, что такое поле не найдено.

```
if Not TeamMsg.IsEmpty then begin
        TeamId:= TeamMsg.FieldByName('team_id').GetValueAsInt32;
end;
```

Полный код доступен в `EXAMPLES\Example04_field_message`

Example05 - поле с типом enum

Рассмотрим тип `Shirt`, который использует перечисляемый тип `Color`.

```
enum Color {
        COLOR_WHITE = 0;
        COLOR_RED   = 1;
        COLOR_GREEN = 2;
}

message Shirt {
        optional float price = 1;
        optional Color color = 2;
}
```

В целом, можно использовать тип данных Int32 вместо перечисляемого типа и записывать/читать значения, используя только целочисленные идентификаторы. В этом случае тип `Shirt` можно представить так:

```
message Shirt {
        optional float price = 1;
        optional Int32 color = 2;
}
```

Однако, использование перечисляемого типа делает данные более удобными и понятными для чтения, а также уменьшает вероятность ошибки из-за использования неопределенного значения.

Описание типа

```
private
        FShirtType: TsanPBMessageType;
        FColorType: TsanPBEnumType;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin

  FColorType:= TsanPBEnumType.Create(nil, 'Color');
  FColorType.AddEnumItem(0, 'COLOR_WHITE');
  FColorType.AddEnumItem(1, 'COLOR_RED');
  FColorType.AddEnumItem(2, 'COLOR_GREEN');

  FShirtType:= TsanPBMessageType.Create(nil, 'Shirt');
  FShirtType.AddFieldDef(ftoOptional, ftFloat, nil, 'price', 1);
  FShirtType.AddFieldDef(ftoOptional, ftEnum, FColorType, 'color', 2);

end;
```

Для описания типа **enum** используется класс `TsanPBEnumType`, который просто хранит список перечисляемых значений и предоставляет методы для доступа к этому списку. Рассмотрим эти методы:

```
procedure AddEnumItem(Value: integer; Descr: string);
```

Эта процедура добавляет перечисляемый элемент в наш `enum` тип.
`Value` - целочисленный идентификатор перечисляемого значения
`Descr` - наименование перечисляемого значения

```
function EnumToString(Value: integer): string;
```

Функция возвращает наименование указанного перечисляемого значения по его числовому идентификатору. Возвращаемое значение будет содержать наименование и числовой идентификатор. Например, применимо к нашему описанию, вызов

`EnumToString(2)` вернет строку: `COLOR_GREEN (2)` В случае, если мы передадим неверное значение, то функция вернет строку вида: `Unknown enum value (%d) for type %s`

```
function StringToEnum(Value: string): integer;
```

Функция позволяет получить числовой идентификатор по наименованию.
Например, вызов `StringToEnum('COLOR_GREEN')` вернет значение `2`.
В случае, если мы передадим несуществующее наименование, будет вызвано исключение.

Заполнение данными и сериализация

```
procedure TForm1.btnSaveClick(Sender: TObject);
var
  ShirtMsg: TsanPBMessage;
begin

  ShirtMsg:= FShirtType.CreateInstance;

  try
    ShirtMsg.FieldByName('price').AppendValueAsFloat(10.5);

    ShirtMsg.FieldByName('color').AppendValueAsString('COLOR_RED');
    // или
    // ShirtMsg.FieldByName('color').AppendValueAsInt32(1);

    ShirtMsg.SaveToFile('shirt.bin');
  finally
    ShirtMsg.Free;
  end;

end;
```

Когда мы добавляем данные **enum** типа, мы можем использовать два метода:
AppendValueAsString - указываем перечисляемое значение по наименованию
AppendValueAsIn32 - указываем перечисляемое значение по числовому идентификатору

Десериализация и чтение данных

```
procedure TForm1.btnLoadClick(Sender: TObject);
var
  ShirtMsg: TsanPBMessage;
  Price: single;
  ColorEnumNum: integer;
```

```
  ColorEnumName: string;
begin

  mmContents.Clear;

  ShirtMsg:= FShirtType.CreateInstance;

  try

    ShirtMsg.LoadFromFile('shirt.bin');

    Price:= ShirtMsg.FieldByName('price').GetValueAsFloat;
    ColorEnumNum:= ShirtMsg.FieldByName('color').GetValueAsInt32;
    ColorEnumName:= ShirtMsg.FieldByName('color').GetValueAsString;

    mmContents.Lines.Add('price: ' + FloatToStr(price));
    mmContents.Lines.Add('color: ' + IntToStr(ColorEnumNum));
    mmContents.Lines.Add('color: ' + ColorEnumName);

  finally
    ShirtMsg.Free;
  end;

end;
```

Как и с добавлением данных при чтении мы также можем использовать две функции:
`GetValueAsInt32` - получить числовой идентификатор
`GetValueAsString` - получить наименование и числовой идентификатор в виде строки.

Полный код доступен в `EXAMPLES\Example05_field_enum`

Example06 - поле с типом map

Поле с типом **map** позволяет работать со списком ключ-значение.
Ниже показан пример из документации Protocol Buffers.

```
message Test6 {
  map<string, int32> g = 7;
}
```

Данный тип можно представить в эквивалентном виде:

```
message Test6 {
  message g_Entry {
    optional string key = 1;
    optional int32 value = 2;
  }
```

```
  repeated g_Entry g = 7;
}
```

В этом примере будет показано, как описать, сериализовать и десериализовать такой тип.

Описание типа

```
private
        FTest6Type: TsanPBMessageType;
```

```
procedure TForm1.FormCreate(Sender: TObject);
var
  EntryType: TsanPBMessageType;
begin

  FTest6Type:= TsanPBMessageType.Create(nil, 'Test6');
  EntryType:= TsanPBMessageType.Create(FTest6Type, 'g_Entry');

  EntryType.AddFieldDef(ftoOptional, ftString, nil, 'key', 1);
  EntryType.AddFieldDef(ftoOptional, ftInt32, nil, 'value', 2);

  FTest6Type.AddFieldDef(ftoRepeated, ftMessage, EntryType, 'g', 7);

end;
```

Здесь мы описываемый вложенный тип `g_Entry`, это реализуется просто указав в `FTest6Type` в качестве `Owner`.

```
EntryType:= TsanPBMessageType.Create(FTest6Type, 'g_Entry');
```

В этом случае, за освобождение объекта `EntryType` будет отвечать "хозяин" `FTest6Type`.

Заполнение данными и сериализация

```
procedure TForm1.btnSaveClick(Sender: TObject);
var
  Test6Msg: TsanPBMessage;
  EntryMsg: TsanPBMessage;
begin

  Test6Msg:= FTest6Type.CreateInstance;
  try
```

```
    EntryMsg:= Test6Msg.MessageByName('g');

    EntryMsg.Append;
    EntryMsg.FieldByName('key').AppendValueAsString('A001');
    EntryMsg.FieldByName('value').AppendValueAsInt32(100);

    EntryMsg.Append;
    EntryMsg.FieldByName('key').AppendValueAsString('A002');
    EntryMsg.FieldByName('value').AppendValueAsInt32(200);

    EntryMsg.Append;
    EntryMsg.FieldByName('key').AppendValueAsString('A003');
    EntryMsg.FieldByName('value').AppendValueAsInt32(300);

    Test6Msg.SaveToFile('test6.bin');

    ShowMessage('Ok');

  finally
    Test6Msg.Free;
  end;

end;
```

Десериализация и чтение данных

```
procedure TForm1.btnLoadClick(Sender: TObject);
var
  Test6Msg: TsanPBMessage;
  EntryMsg: TsanPBMessage;
  I: integer;
begin

  mmContents.Clear;

  Test6Msg:= FTest6Type.CreateInstance;
  try

    Test6Msg.LoadFromFile('test6.bin');

    EntryMsg:= Test6Msg.MessageByName('g');

    for I := 1 to EntryMsg.RecordCount do begin
      EntryMsg.RecordIndex:= I-1;
      mmContents.Lines.Add('key: ' +
EntryMsg.FieldByName('key').GetValueAsString);
      mmContents.Lines.Add('value: ' +
                           EntryMsg.FieldByName('value').GetValueAsString)
```

```
    end;

  finally
    Test6Msg.Free;
  end;

end;
```

Полный код доступен в `EXAMPLES\Example06_field_map`

Example07 - автоматизируем описание типов

Иногда возникает ситуация, когда у нас есть большое количество proto-файлов на языке `Protocol Buffers`. Описывать типы вручную с помощью класса `TsanPBMessageType`, как мы делали в предыдущих примерах, не очень удобно и может привести к ошибкам. Однако, эту работу можно автоматизировать с помощью программы `pbpgui.exe`. Ее можно найти в папке `TOOLS` или скачать с GitHub.

В этом примере у нас есть папка `proto`, где находятся три proto-файла:

`EXAMPLES\Examples07_parser\proto\firm_list.proto'

```
syntax = "proto2";

import "firm.proto";

message FirmList {
        repeated Firm firms = 1;
}
```

`EXAMPLES\Examples07_parser\proto\firm.proto'

```
syntax = "proto2";

import "address.proto";

message Firm {
        required string frm_name = 1;
        optional Address address = 2;
        optional FirmType frm_type = 3 [Default=FirmType.BUYER];
}

enum FirmType {
        SELLER = 0;
        BUYER = 1;
        PREMIUM_BUYER = 2;
}
```

`EXAMPLES\Examples07_parser\proto\address.proto'

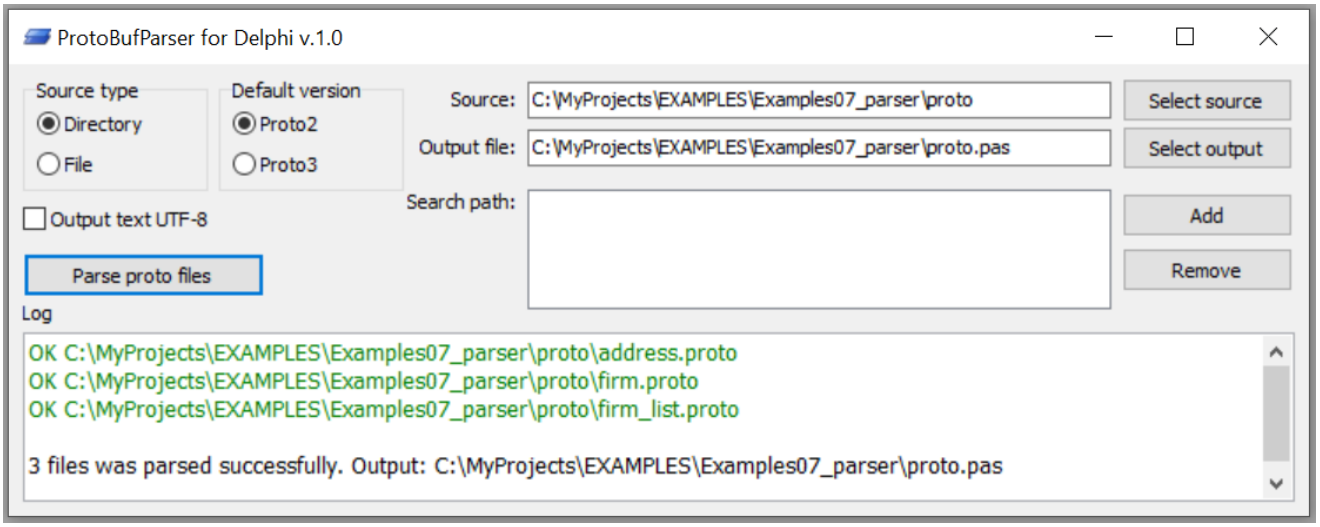
```
syntax = "proto2";

message Address {
        optional string zip_code = 1;
        optional string region = 2;
        optional string city = 3;
        optional string street = 4;
        optional string building = 5;
}
```

Теперь запустим программу `pbpgui.exe` и укажем необходимые ей данные:

1. `Source type` - параметр указывает, что мы будем обрабатывать: один proto-файл или папку с proto-файлами.
2. `Default version` - данный параметр указывает версию языка Protocol Buffers по умолчанию. Действует, если в описании отсутствует явное указание версии.
3. `Source` - источник proto-файлов, которые нужно описать. В зависимости от параметра `Source type` может быть папкой или одним файлом.
4. `Output file` - файл, который будет содержать описание типов на языке Delphi
5. `Output text UTF8` - кодировка `Output file`. Если параметр указан, кодировка будет UTF-8, в противном случае ANSI в кодовой странице операционной системы.
6. `Search path` - дополнительные пути для поиска proto-файлов.

Затем, нажимаем кнопку `Parse proto files` и если все правильно, результат будет таким:



Посмотрим теперь на созданный файл `proto.pas`

В начале файла у нас есть комментарий:

```
{=============================================================================
==
  This file was generated automatically by the ProtoBufParser for Delphi
v.1.0
  Date: 26.03.2024 20:44:56

  Files:
        C:\MyProjects\EXAMPLES\Examples07_parser\proto\address.proto
        C:\MyProjects\EXAMPLES\Examples07_parser\proto\firm.proto
        C:\MyProjects\EXAMPLES\Examples07_parser\proto\firm_list.proto

  Available types:
        Address
        Firm
        FirmList



=============================================================================
=}
```

Здесь есть два раздела:

- `Files` - список proto-файлов, которые были прочитаны
- `Available types` - список типов, которые можно использовать

Далее, объявлена одна единственная функция:

```
function CreateProtoInstance(ProtoName: string): TsanPBMessage;
```

С помощью нее можно создать объект `TsanPBMessage` для указанного типа из списка, который был показан в разделе `Available types`.

Для того, чтобы использовать созданный файл `proto.pas` в нашем примере, его нужно добавить в проект и прописать в раздел `uses`.

```
uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Vcl.StdCtrls, semin64.protobuf, proto;
```

Заполнение данными и сериализация

```
procedure TForm1.btnSaveClick(Sender: TObject);
var
  FirmList: TsanPBMessage;
begin
```

```
  FirmList:= CreateProtoInstance('FirmList');

  try

    LoadFirmList(FirmList);
    FirmList.SaveToFile('firm_list.bin');

  finally
    FirmList.Free;
  end;

end;
```

Я не стал здесь загромождать код заполнением данных, выделив его в отдельную процедуру `LoadFirmList`.

```
procedure TForm1.LoadFirmList(FirmList: TsanPBMessage);
var
  Firm, Address: TsanPBMessage;
begin

  Firm:= FirmList.MessageByName('firms');

  Firm.Append;
  Firm.FieldByName('frm_name').AppendValueAsString('Olimp');
  Firm.FieldByName('frm_type').AppendValueAsString('PREMIUM_BUYER');

  Address:= Firm.MessageByName('address');

  Address.Append;
  Address.FieldByName('zip_code').AppendValueAsString('123456');
  Address.FieldByName('city').AppendValueAsString('Moscow');
  Address.FieldByName('street').AppendValueAsString('Ryasansky pr-t');
  Address.FieldByName('building').AppendValueAsString('55/1');

  Firm.Append;
  Firm.FieldByName('frm_name').AppendValueAsString('Chamomile');
  Firm.FieldByName('frm_type').AppendValueAsString('BUYER');
  Address.Append;
  Address.FieldByName('zip_code').AppendValueAsString('142184');
  Address.FieldByName('region').AppendValueAsString('MO');
  Address.FieldByName('city').AppendValueAsString('Klimovsk');
  Address.FieldByName('street').AppendValueAsString('Prospect oktyabrya');
  Address.FieldByName('building').AppendValueAsString('9');

end;
```

Десериализация и чтение данных

Если в предыдущих примерах мы читали данные, обращаясь к полям напрямую, то здесь я хочу продемонстрировать процедуру `DbgMessageToStrings`, которая позволяет вывести все содержимое объекта класса TsanPBMessage в объект TStrings. Это может быть полезно при отладке, например, после десериализации мы можем сразу увидеть все содержимое сообщения.

```
procedure TForm1.btnLoadClick(Sender: TObject);
var
  FirmList: TsanPBMessage;
begin

  FirmList:= CreateProtoInstance('FirmList');

  try

    FirmList.LoadFromFile('firm_list.bin');
    FirmList.DbgMessageToStrings(mmContents.Lines);

  finally
    FirmList.Free;
  end;

end;
```

Результат вывода выглядит так:

```
FirmList
  firms (Firm)
    frm_name (String): Olimp
    address (Address)
      zip_code (String): 123456
      city (String): Moscow
      street (String): Ryasansky pr-t
      building (String): 55/1
    frm_type (FirmType): PREMIUM_BUYER (2)
  firms (Firm)
    frm_name (String): Chamomile
    address (Address)
      zip_code (String): 142184
      region (String): MO
      city (String): Klimovsk
      street (String): Prospect oktyabrya
```

```
        building (String): 9
      frm_type (FirmType): BUYER (1)
```

Полный код доступен в `EXAMPLES\Example07_proto_parser`

Example08 - Контур Диадок

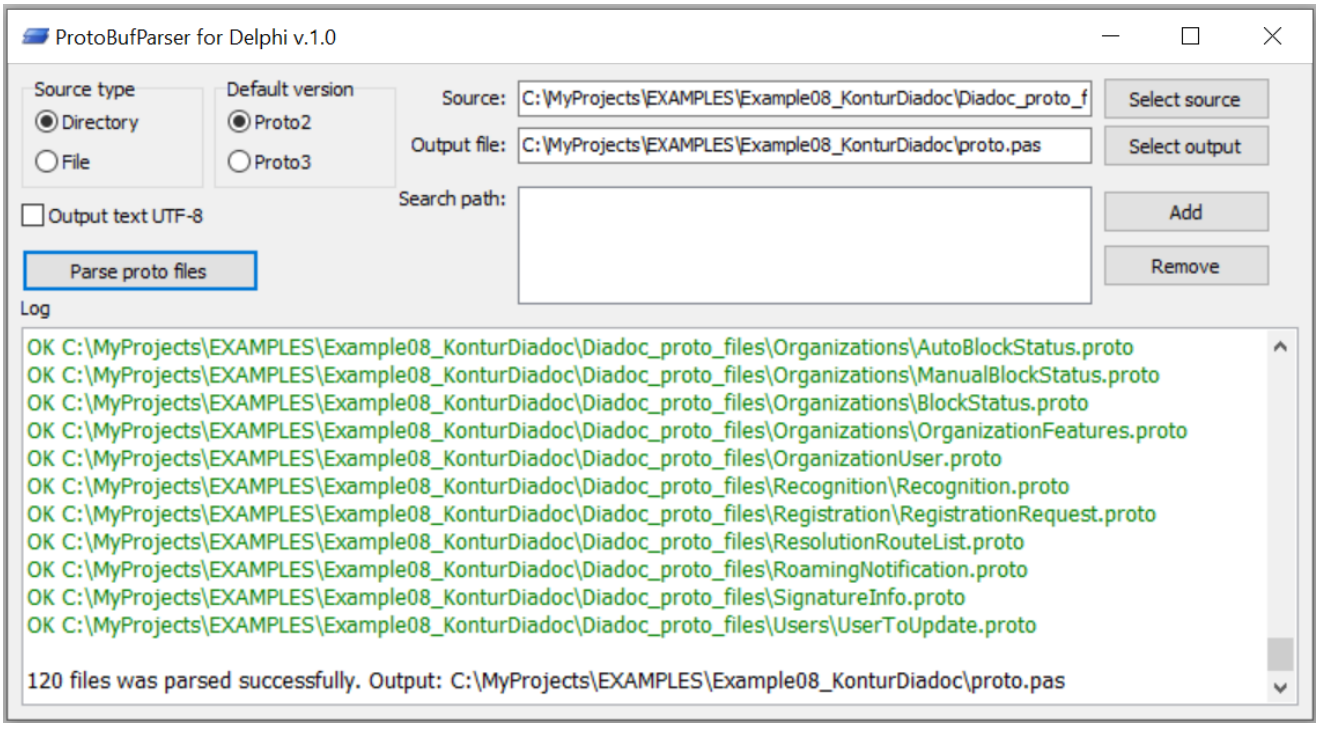
Так как данная библиотека была разработана в первую очередь для работы с Контур Диадок, то в последнем примере будет показано взаимодействие с этой системой по API с использованием Protocol Buffers для получения информации.
В этом примере мы опишем все proto-типы для взаимодействия с «Контур Диадок» и создадим приложение, которое подключится и загрузит список ящиков, к которым пользователь имеет доступ. (GetMyOrganizations).

> Внимание! Данный пример можно запустить, только если у вас есть учётная запись в системе «Контур Диадок», а также открыт доступ по API (у вас должен быть идентификатор API Key).

Описание типов

Первое, что мы должны сделать, — это создать файл с описаниями proto-типов. Их можно найти в папке `EXAMPLES\Example08_KonturDiadoc\Diadoc_proto_files`, но имейте в виду, что, скорее всего, они могли устареть. Свежую версию можно найти здесь.

Как и в предыдущем примере, мы воспользуемся утилитой `pbpgui.exe`.



Если мы посмотрим сформированный файл proto.pas, то легко заметить, что имена доступных типов здесь гораздо длиннее.

```
Available types:
        Diadoc.Api.Proto.AcquireCounteragentRequest
        Diadoc.Api.Proto.InvitationDocument
        Diadoc.Api.Proto.AcquireCounteragentResult
 ...
```

Это связано с именами пакетов. Чтобы обеспечить уникальность имени типа, используется имя пакета и через точку указывается имя типа.
Например proto файл `Content.proto`

```
package Diadoc.Api.Proto;

message Content {
        required sfixed32 Size = 1;
        optional bytes Data = 2;
}
```

Здесь имя пакета `Diadoc.Api.Proto`, имя типа `Content`. Полное имя типа: `Diadoc.Api.Proto.Content`

Подключение и получение информации через Diadok API

К сожалению, объяснение процесса подключения выходит за рамки проекта, но, если вкратце, то взаимодействие было реализовано с помощью библиотеки `wininet.dll`, и весь код был упакован в класс `TDiadocApi`.

Класс `TDiadocApi` содержит три метода:

```
    procedure Connect;
```

Данный метод устанавливает подключение к «Контур Диадок»

```
        procedure Disconnect;
```

Метод отключает от «Контур Диадок»

```
        procedure Get(Url: string; Answer: TStream);
```

Метод запрашивает информацию передавая соответствующий *Url* и возвращает бинарные данные `Protocol buffers` в виде результата в поток *Answer*.

В нашем примере мы создаем объект `TDiadocApi` в событии `OnCreate` нашей формы.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FDiadocAPI:= TDiadocApi.Create;
end;
```

При нажатии на кнопку `btnGetMyOrganizations` мы выполним соединение с «Контур Диадок», запросим информацию через метод `GetMyOrganizations`, распакуем ее и выведем полученный результат в текстовое окно mmResult (TMemo).

```
procedure TForm1.btnGetMyOrganizationsClick(Sender: TObject);
var
  Buf: TMemoryStream;
  OrganizationList: TsanPBMessage;
begin

  Buf:= TMemoryStream.Create;
  OrganizationList:=
CreateProtoInstance('Diadoc.Api.Proto.OrganizationList');

  try

    FDiadocAPI.ApiKey:= edtApiKey.Text;
    FDiadocAPI.UserName:= edtUserName.Text;
    FDiadocAPI.Password:= edtPassword.Text;
    FDiadocAPI.Host:= 'diadoc-api.kontur.ru';
    FDiadocAPI.Port:= 443;

    FDiadocAPI.Connect;

    FDiadocAPI.Get('/GetMyOrganizations?autoRegister=false', Buf);
    OrganizationList.LoadFromStream(Buf);
    OrganizationList.DbgMessageToStrings(mmResult.Lines);

    FDiadocAPI.Disconnect;

  finally
    Buf.Free;
    OrganizationList.Free;
  end;

end;
```

Рассмотрим код более подробно.
Сначала мы создадим поток `Buf` в который потом поместим бинарные данные, полученные по API Диадок.

```
Buf:= TMemoryStream.Create;
```

Создадим объект для работы с данными, описанными в proto-типе `OrganizationList`

```
OrganizationList:= CreateProtoInstance('Diadoc.Api.Proto.OrganizationList');
```

Перед подключением, установим параметры соединения

```
FDiadocAPI.ApiKey:= edtApiKey.Text;
FDiadocAPI.UserName:= edtUserName.Text;
FDiadocAPI.Password:= edtPassword.Text;
FDiadocAPI.Host:= 'diadoc-api.kontur.ru';
FDiadocAPI.Port:= 443;
```

Выполняем подключение и вызываем метод API `GetMyOrganizations`, результат помещаем в наш буфер `Buf`.

```
FDiadocAPI.Connect;
FDiadocAPI.Get('/GetMyOrganizations?autoRegister=false', Buf);
```

Полученные данные загружаем в наш объект `OrganizationList` и выводим все содержимое в текстовое окно `mmResult`

```
OrganizationList.LoadFromStream(Buf);
OrganizationList.DbgMessageToStrings(mmResult.Lines);
```

Далее, отсоединяемся и освобождаем созданные объекты.

Полный код доступен в `EXAMPLES\Example08_KonturDiadoc`