Subject: Flatbuffers package Posted by Xemuth on Tue, 15 Dec 2020 11:48:36 GMT

View Forum Message <> Reply to Message

Hello,

I had to work with Google Flatbuffers and for my test I have made a TCP/IP client with Upp (in order to connect to server (which work with flatbuffer)).

That's why I'm sharing here my Flatbuffers package and a quick client/server exemple.

What is Flatbuffers?

FlatBuffers is an efficient cross platform serialization library for C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust and Swift. It was originally created at Google for game development and other performance-critical applications.

How Flatbuffers work?

FlatBuffers provide an executable (named flatc.exe)(present in bin folder of my package, Windows only, I plan to add debian/linux executable too) capable of converting fbs file to header file representing data. Let me show a simple example:

```
Command.fbs:

// My command fbs

namespace TcplpServer;

table Command {
   id:int;
   name:string;
}
```

root_type Command;

In this simple fbs file we can quickly remarque a namespace, a table (wich can basicly be interpreted as structure or object) and a root_type. This file describe layout of our data.

- -The schema starts with a namespace declaration. This determines the corresponding package/namespace for the generated code. In our example, we have the TcpIpServer namespace
- -The Command table is the main object in our FlatBuffer. This will be used as the template to store all our defined command.
- -The last part of the schema is the root_type. The root type declares what will be the root table for the serialized data. In our case, the root type is our Command table

You can find a far more precise description of FBS file here: Flatbuffers tutorial

By passing this fbs file to flatc.exe as follow: ~/flatc.exe --cpp command.fbs it gonna generate an header file (in our case named command_generated.h) contening a huge implementation of our data structure (you can find this file here if you are curious!) this header file will be inserted in our project as follow: #include <Core/Core.h> #include "command generated.h" using namespace Upp; Right now flatbuffers is inserted in our project and can be used to Serialize/Deserialize command object. Lets quickly see how it's done: //a simple Command struct struct Command{ public: int id: String name; Command(int _id, String _name){id = _id; name = _name;} String ToString()const{ String toStr; toStr << "Id: " << id << ", Name: " << name << "\n"; return toStr: **}**; struct CommandSerializer{ public:

CommandSerializer(const Command& cmd){

flatbuffers::Offset<flatbuffers::String> str1 = builder.CreateString(cmd.name.ToStd()); //First, we create a special flatbuffers object that handle string

TcplpServer::CommandBuilder commandBuilder(builder); //Then we create a commandBuilder (a special object that will construct our Command (the fbs file))

commandBuilder.add_id(cmd.id); //We define value of each parameters of our command commandBuilder.add_name(str1); //We define value of each parameters of our command flatbuffers::Offset<TcplpServer::Command> freshCommand = commandBuilder.Finish(); //we retrieve our command by telling our commandBuilder we are done with it

builder.Finish(freshCommand); //we end the builder of our newly command (named freshCommand)

int GetSize(){return builder.GetSize();} //Return a ptr to our serialized object

```
uint8_t* GetDatasPtr(){return builder.GetBufferPointer();} //Return size of our serialized object private:
flatbuffers::FlatBufferBuilder builder; //Builder is used to create everything we want with flatbuffers
};

Command CommandDeserializer(uint8_t* datas, int size){
flatbuffers::Verifier verfier(datas,size); //Verifier is used to verify our buffer of data is a valid command
if(TcplpServer::VerifyCommandBuffer(verfier)){
    const TcplpServer::Command* serializedCmd = TcplpServer::GetCommand(datas); //If our buffer is valid then we retrieve a ptr to the command within this buffer
    return Command(serializedCmd->id(), String(serializedCmd->name()->c_str())); //then we use our buffer to build our command
}
throw Exc("Invalide command");
}
```

the ptr to the binary serialized data can be used to sent data over the network for example. Lets see in the next post a simple Client/Server flatbuffer exchange

File Attachments

1) Flatbuffers.7z, downloaded 324 times