

Лекция 13. Сериализация и десериализация

Илья Макаров

итмо јв

14 декабря 2021 Санкт-Петербург



Сериализация и десериализация

- **Сериализация** процесс перевода структуры данных в последовательность байтов.
- Десериализация создание структуры данных из битовой последовательности (обратная операция).



Как можно сериализовать?

- С помощью текстового представления. Например, в форматы: **XML**, **JSON**,
- В бинарные форматы.
 Например, в специализированные форматы для хранения больших объемов данных: HDF, netCDF или более старый GRIB.



Зачем это может быть нужно?

- Для передачи данных по сетевым протоколам.
- Для сохранения информации на файловую систему.
- Для удаленного вызова процедур RPC.
- ...



Как это можно сделать?

Путь инженера:

- Выбираем подходящий текстовый формат.
- Выбираем подходящую библиотеку для чтения/записи нашего формата.
- Реализуем запись/чтение файла с помощью это библиотеки.

Путь джедая:

- Придумываем и описываем наш собственный бинарный формат.
- Пишем библиотеку/утилиту для работы с нашим форматом.
- Реализуем запись/чтение с помощью нашей библиотеки/утилиты.
- Чиним баги...



Библиотеки для работы с текстовым представлением

Для XML:

• pugixml, tinyxml, rapidxml, ...

Для JSON:

• jsoncpp, rapidjson, ...

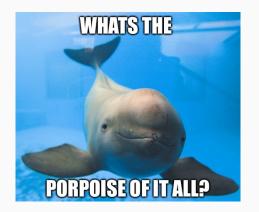
Для YML:

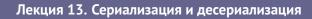
• yaml-cpp, rapidyaml, ...

Другие форматы: ...



Про путь джедая







Про путь джедая





Проблемы при разработке формата

- Высокая сложность в разработке и отладке.
- Сложные оптимизация для оптимального хранения данных.
- Всегда много багов, требуется хорошее тестовое покрытие.
- В перспективе, такой код тяжело поддерживать.
- Возникают проблемы с версионированием.
- ...



Путь инженера-джедая

- Для сериализации/десериализации будем использовать готовые и гибкие решения.
- Примеров таких решений: protobuf, flatbuffers, ...
- Позволяют избежать сложностей связанных с разработкой и сконцентрироваться на самом формате (на описании структур данных).

Далее в качестве примера рассмотрим protobuf.



Protocol buffers

- Протокол сериализации (передачи) структурированных данных, предложенный Google как эффективная бинарная альтернатива текстовому формату XML.
- Проще, компактнее и быстрее, чем XML, поскольку осуществляется передача бинарных данных, оптимизированных под минимальный размер сообщения.
- Интерфейс поддерживает несколько популярных языков: **C++**, **Java**, **Kotlin**, **C**, **Go**, ...
- Последняя актуальная версия версия 3.19.1 (октябрь 2021).
- https://developers.google.com/ protocol-buffers/docs/cpptutorial



Алгоритм работы с protobuf

- Описываем наши структуры данных в **proto-файле**.
- Компилируем с помощью protoc.
- Подключаем сгенерированные заголовки к нашему проекту.
- Работаем с бинарным представлением.
 Читаем/записываем/модифицируем.
- Возможно расширяем наш proto-формат.



Описание формата

```
syntax = "proto3";
package tutorial;
message Person {
  string name = 1;
  int32 id = 2;
  string email = 3:
  enum PhoneType {
    MOBILE = 0;
   HOME = 1:
    WORK = 2;
  message PhoneNumber {
    string number = 1;
    PhoneType type = 2 [default = HOME];
  repeated PhoneNumber phones = 4;
message AddressBook {
  repeated Person people = 1;
```



Лекция 13. Сериализация и десериализация

Генерация кода

Запускаем компиляцию вручную:

```
protoc
   -I=$SRC_DIR
   --cpp_out=$DST_DIR
   $SRC_DIR/addressbook.proto
```



Генерация кода

Используем вспомогательные функции из cmake:

```
include(FindProtobuf)
find_package(Protobuf REQUIRED)
include_directories(${PROTOBUF_INCLUDE_DIR})

# to find *.pb.h files
include_directories(${CMAKE_CURRENT_BINARY_DIR})

protobuf_generate_cpp(PROTO_SRC PROTO_HEADER src/proto/addressbook.proto)
add_library(proto ${PROTO_HEADER} ${PROTO_SRC})
add_executable(main main.cpp)
target link libraries(main proto ${PROTOBUF_LIBRARY})
```



Результат генерации

Сгенерировали 2 файла: addressbook.pb.h и addressbook.pb.cc.

```
// name
inline bool has_name() const;
inline void clear_name();
inline const ::std::string& name() const;
inline void set_name(const ::std::string& value);
inline void set_name(const char* value);
inline ::std::string* mutable_name();

// id
inline bool has_id() const;
inline void clear_id();
inline int32_t id() const;
inline void set_id(int32_t value);
...
```



Результат генерации

Появились дополнительные методы, нужные прежде всего для отладки.

```
// checks if all the required fields have been set.
bool IsInitialized() const;
// returns a human-readable representation of the message,
// particularly useful for debugging.
string DebugString() const;
// overwrites the message with the given message's values.
void CopyFrom(const Person& from);
// clears all the elements back to the empty state.
void Clear();
```

И методы для чтения/записи:

```
// serializes the message and stores the bytes in the given string.
bool SerializeToString(string* output) const;
// parses a message from the given string.
bool ParseFromString(const string& data);
// writes the message to the given C++ ostream.
bool SerializeToOstream(ostream* output) const;
// parses a message from the given C++ istream.
bool ParseFromIstream(istream* input);
```