

(Де)сериализация

Protocol Buffers



Илья Кокорин

kokorin.ilya.1998@gmail.com

Постановка задачи

- Сериализация - процесс преобразования объекта в массив байт
 - Чтобы можно было переслать объект по сети
 - Или записать на диск
- Десериализация - обратный процесс



Пример объекта

- Описание библиотеки

```
babylonLibrary := Library{
  Address: "Babylon",
  Books: []Book{
    {
      Title:    "In Stahlgewittern",
      Language: "German",
      Author: Author{
        Name:    "Ernst Jünger",
        BirthDate: Date{Year: 1895, Month: March, Day: 29},
      },
    },
    {
      Title:    "Опавшие листья",
      Language: "Russian",
      Author: Author{
        Name:    "Василий Розанов",
        BirthDate: Date{Year: 1856, Month: May, Day: 2},
      },
    },
    {
      Title:    "Rigodon",
      Language: "French",
      Author: Author{
        Name:    "Louis-Ferdinand Céline",
        BirthDate: Date{Year: 1894, Month: May, Day: 27},
      },
    },
  },
}
```

Сериализация через XML

- Выглядит очень громоздко

- Множество символов не несёт смысла с точки зрения содержимого

- <
- </
- />
- ``

- Неэффективно передаём числа

```
<Library Address = "Babylon">
  <Books>
    <Book Title = "In Stahlgewittern" Language = "German">
      <Author Name = "Ernst Jünger">
        <BirthDate Year = "1895" Month = "March" Day = "29"/>
      </Author>
    </Book>
    <Book Title = "Опавшие листья" Language = "Russian">
      <Author Name = "Василий Розанов">
        <BirthDate Year = "1856" Month = "May" Day = "2"/>
      </Author>
    </Book>
    <Book Title = "Rigodon" Language = "French">
      <Author Name = "Louis-Ferdinand Céline">
        <BirthDate Year = "1894" Month = "May" Day = "27"/>
      </Author>
    </Book>
  </Books>
</Library>
```

Сериализация через XML

- Можно при сериализации использовать атрибуты тегов

```
<Book Title = "Опавшие листья" Language = "Russian">  
  <Author Name = "Василий Розанов">  
    <BirthDate Year = "1856" Month = "May" Day = "2"/>  
  </Author>  
</Book>
```

- А можно не использовать

- Десериализатор должен это учитывать

```
<Book>  
  <Title>  
    Опавшие листья  
  </Title>  
  <Language>  
    Russian  
  </Language>  
  <Author>  
    <Name>  
      Василий Розанов  
    </Name>  
    <BirthDate>  
      <Year>  
        1856  
      </Year>  
      <Month>  
        May  
      </Month>  
      <Day>  
        2  
      </Day>  
    </BirthDate>  
  </Author>  
</Book>
```

Сериализация через JSON

- Выглядит более привычно
- Похоже на описание объекта на каком-то языке программирования
- На JavaScript
- JSON =
JavaScript
Object Notation
- Неэффективный
 - Те же проблемы, что у XML

```
{  
  "address": "Babylon",  
  "books": [  
    {  
      "title": "In Stahlgewittern", "language": "German",  
      "author": {  
        "name": "Ernst Jünger", "birth_date": {"year": 1895, "month": "March", "day": 29}  
      }  
    },  
    {  
      "title": "Опавшие листья", "language": "Russian",  
      "author": {  
        "name": "Василий Розанов", "birth_date": {"year": 1856, "month": "May", "day": 2}  
      }  
    },  
    {  
      "title": "Rigodon", "language": "French",  
      "author": {  
        "name": "Louis-Ferdinand Céline", "birth_date": {"year": 1894, "month": "May", "day": 27}  
      }  
    }  
  ]  
}
```

Сериализация: Protobuf

- Бинарный формат

```
Marshaled library: [10 7 66 97 98 121 108 111 110 18 53 10 17 73 110 32 83 116
 97 104 108 103 101 119 105 116 116 101 114 110 18 6 71 101 114 109 97 110 26 24
 10 13 69 114 110 115 116 32 74 195 188 110 103 101 114 18 7 8 231 14 16 3 24 29
 18 80 10 27 208 158 208 191 208 176 208 178 209 136 208 184 208 181 32 208 187
208 184 209 129 209 130 209 140 209 143 18 7 82 117 115 115 105 97 110 26 40 10
29 208 146 208 176 209 129 208 184 208 187 208 184 208 185 32 208 160 208 190 20
8 183 208 176 208 189 208 190 208 178 18 7 8 192 14 16 5 24 2 18 53 10 7 82 105
103 111 100 111 110 18 6 70 114 101 110 99 104 26 34 10 23 76 111 117 105 115 45
 70 101 114 100 105 110 97 110 100 32 67 195 169 108 105 110 101 18 7 8 230 14 1
6 5 24 27]
Marshaled library size: 201 bytes
```

- Компактный

```
> wc -c < library.json
629
> wc -c < library-attributes.xml
732
> wc -c < library-children.xml
1936
```

Protobuf: язык описания

- Описываем структуру
- У структуры есть поля
 - У каждого поля есть название
 - Тип
 - И числовой уникальный идентификатор
 - Уникальность - в пределах одной структуры

```
message MyStruct {  
    string string_field = 1;  
    int32 int_field = 2;  
    uint64 uint_field = 3;  
}
```


Protobuf: язык описания

- Структуры могут быть вложенными
 - Вложенность произвольная

```
enum Type {  
    TYPE_A = 0;  
    TYPE_B = 1;  
    TYPE_C = 2;  
}
```

```
message MyStruct {  
    string name = 1;  
    Type type = 2;  
}
```

- Есть перечисления
 - Работают так, как вы от них ожидаете

```
message InnerStruct {  
    string key = 1;  
    string value = 2;  
}
```

```
message MyStruct {  
    string name = 1;  
    InnerStruct inner = 2;  
}
```

Protobuf: язык описания

- Любое поле может быть помечено как `repeated`
- Структура будет хранить произвольное количество (от нуля до бесконечности) значений такого поля

```
enum Type {  
    TYPE_A = 0;  
    TYPE_B = 1;  
}  
  
message InnerStruct {  
    string key = 1;  
    repeated string value = 2;  
}  
  
message MyStruct {  
    repeated string name = 1;  
    repeated Type type = 2;  
    repeated InnerStruct inner = 3;  
}
```

Protobuf: генерация кода

- Генерируется код для сериализации и десериализации
 - На вашем любимом ЯП
- Код страшный
 - Читать его не надо
 - Редактировать - тем более

```
func (x *Person) Reset() {
    *x = Person{}
    if protoimpl.UnsafeEnabled {
        mi := &file__1_simple_person_proto_msgTypes[0]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Person) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Person) ProtoMessage() {}

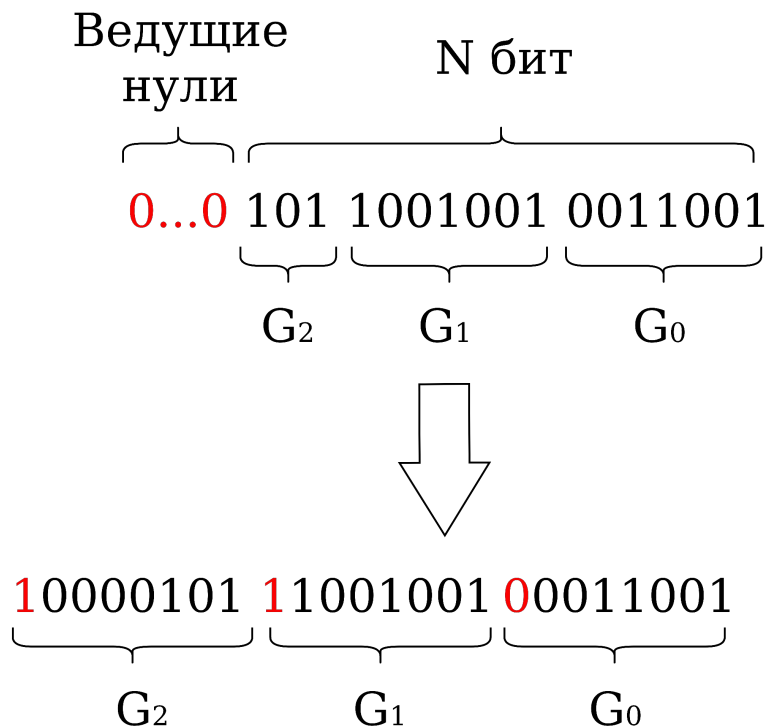
func (x *Person) ProtoReflect() protoreflect.Message {
    mi := &file__1_simple_person_proto_msgTypes[0]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}
```

Protobuf: сериализация чисел

- Целые числа любой битности сериализуются одним и тем же способом
 - Отводить по 8 байт на все целые числа - очень расточительно
 - Числа из диапазона $[0..127]$ занимают 1 байт, а не 8
- Применим varint encoding
- Если число может быть представлено 7 битами - записываем его как есть
- 01011011

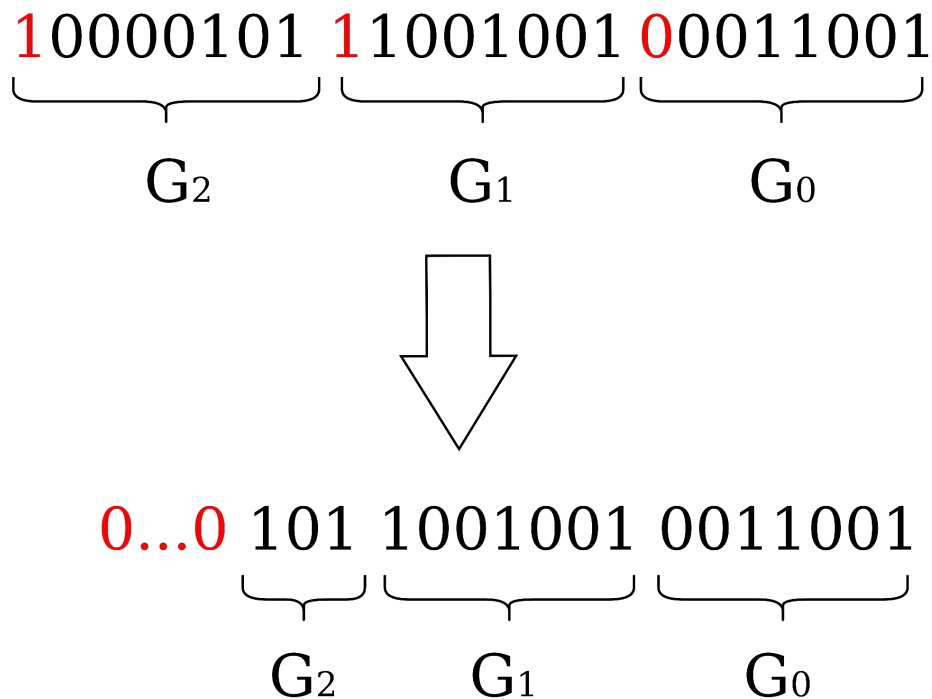
Varint: сериализация

- Иначе отбрасываем все незначащие нули
- Делим число на группы по 7 бит
 - Групп будет $\left\lceil \frac{N}{7} \right\rceil$
- Всем группам, кроме последней, ставим старший (восьмой) бит = 1
- Последней группе ставим старший (восьмой) бит = 0



Varint: десериализация

- Читаем байты пока старший бит не 0
- Отбрасываем старшие биты у каждого прочитанного байта
 - Получаем несколько семибайтных групп
- Конкатенируем их



Varint: порядок байт

- Можно хранить группы в порядке big-endian
 - $G_3 G_2 G_1 G_0$
 - От старшей группы к младшей
- Можно хранить группы в порядке little-endian
 - $G_0 G_1 G_2 G_3$
 - От младшей группы к старшей
 - Protobuf хранит так

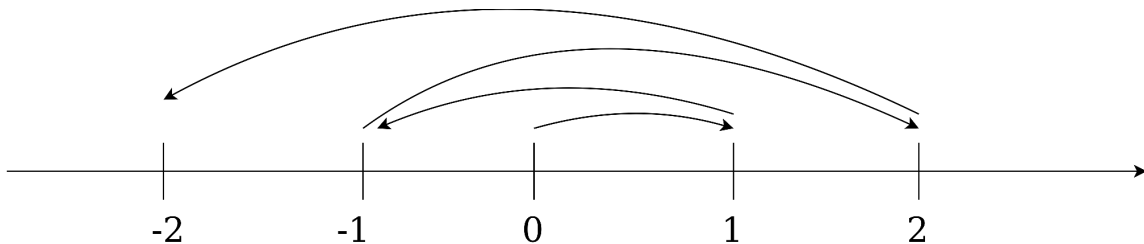
Varint: отрицательные числа

- Как выглядит число $-1 :: \text{int64}$?
 - $1\ b_{62}\ b_{61}\ b_{60} \dots b_1\ b_0$
- Применим zig-zag encoding

0	0
1	1
-1	2
2	3
-2	4

```
1 fun to_zigzag(v):  
2   if v ≤ 0:  
3     return 2 · | v |  
4   else:  
5     return 2 · (v - 1)
```

```
1 fun from_zigzag(z):  
2   if z % 2 = 0:  
3     return - z // 2  
4   else:  
5     return (z + 1) // 2
```



Zigzag Encoding

- Преимущества:
 - Маленькие отрицательные числа кодируются небольшим числом байт
- Недостатки:
 - `to_szigzag(126) = 250`
 - 2 байта на представление вместо одного
- Пользователь может включать и отключать Zigzag Encoding
 - `sint32/sint64` - включает
 - `int32/int64` - отключает

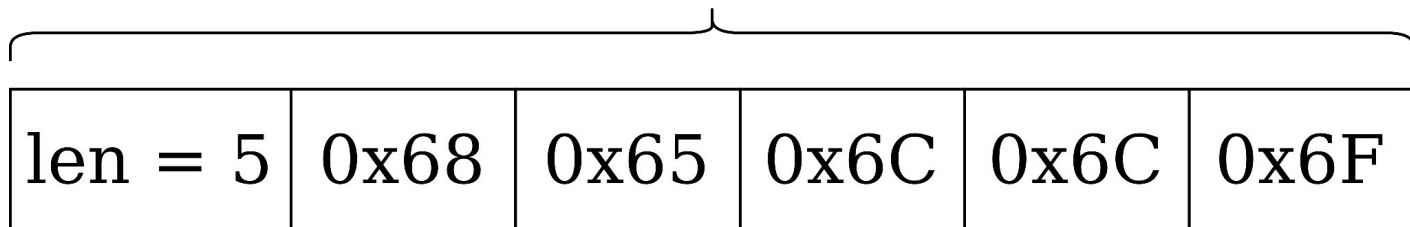
Protobuf: сериализация структур

- $id_1 : value_1; id_2 : value_2; \dots id_n : value_n$
- Формат сериализации: список key-value пар
 - Ключ - идентификатор поля
 - Значение - сериализованное значение поля
- Имя поля в записи отсутствует
 - Без схемы нельзя десериализовать
- Идентификатор - целое число
 - Используем varint encoding

Protobuf: сериализация полей

- Целые числа - varint
- Перечисления - как целые числа
- Числа с плавающей точкой - как есть
 - Тратим всегда 4/8 байт
- Строки - длина в байтах + содержимое
 - Массивы байт так же
 - Структуры сериализуем, потом так же

"hello"



Protobuf: repeated поля

- `rep_id : value1; rep_id : value2; ... rep_id : valuen;`
 - `=> rep_field : [value1, value2, ... valuen]`
 - Просто повторяем key-value пару для каждого элемента списка
 - Можно чередовать с остальными полями

Protobuf: поля без значения

- Что происходит, если полю не было задано значение перед сериализацией?
 - Этого поля не будет в байтовом представлении
- Что произойдёт при десериализации?
 - Числовое поле получит значение 0 (`false` для `bool`)
 - Строковое - пустую строку
 - Структурное - зависит от языка
 - *For message fields, its exact value is language-dependent. See the generated code guide for details.*
- Поля со значением по умолчанию опускаются при сериализации

Оптимизации Protobuf: идентификаторы

- Представим себе такую структуру

```
message MyStruct {  
    string rare_field = 1;  
    // ... other fields  
    repeated string frequent_field = 3000;  
}
```

- Идентификатор часто встречающегося поля занимает 2 байта
- А редко встречающегося - 1 байт
- Лучше сделать наоборот

Оптимизации Protobuf: repeated поля

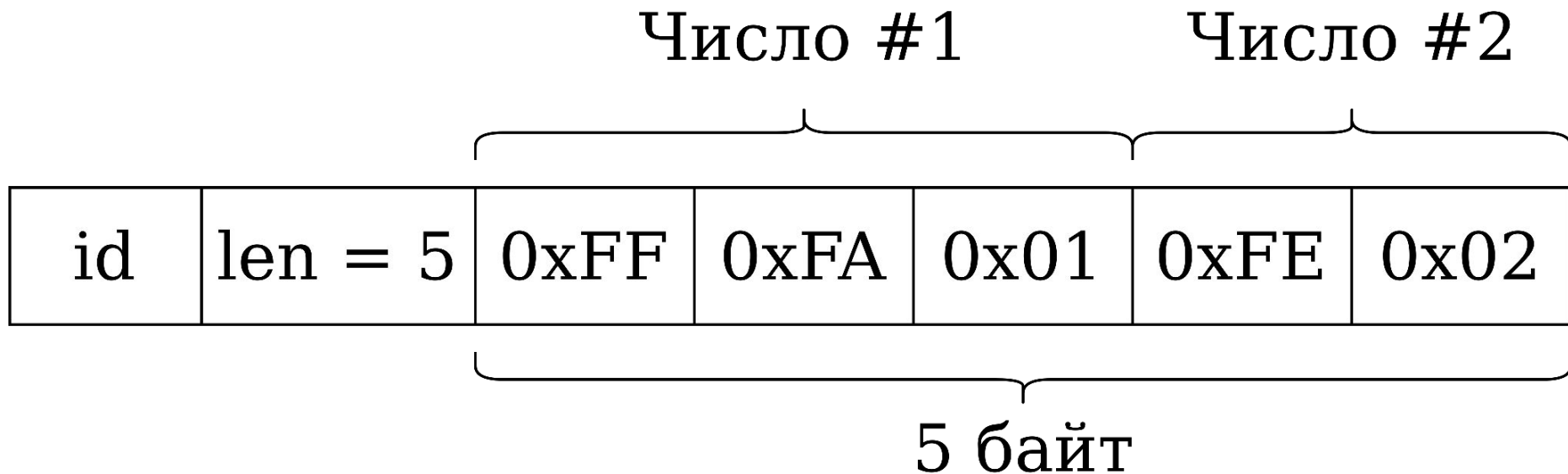
- Представим себе такую структуру

```
message MyStruct {  
    // ... other fields  
    repeated uint32 ids = 2000;  
}
```

- Формат сериализации:
 - 2000 : 1; 2000: 2; ... 2000 : 7;
 - На каждый байт значения приходится два байта идентификатора

Packed repeated fields

- Используем один идентификатор на множество значений
- Оптимизация не применяется для строк, структур и массивов байт



Protobuf: Backward compatibility

- Обратная совместимость
 - Сериализуем объект
 - Записываем его на диск
 - Меняем .proto-файл
 - Перекомпилируем
 - Читаем с помощью новой версии объект, созданный старой версией кода
 - Ничего не должно сломаться

Protobuf: Backward compatibility

- Удаляем поле F
 - { ..., F_Id : value; ... }
 - Значение будет проигнорировано
- Добавляем поле F
 - { A_Id : 5, B_Id : Hello }
 - Поле F получит значение по умолчанию
- Делаем поле F repeated
 - { F_Id : 5 }
 - Массив значений будет состоять из одного значения

Protobuf: Backward compatibility

- Делаем поле F не repeated
 - { F_Id : 5, F_Id : 6, F_Id : 42 }
- Для скалярных значений - возьмём последнее
 - 42
- Структуры смёржим

$$\left\{ \begin{array}{l} x : x_1, \\ y : y_1 \end{array} \right\} \oplus \left\{ \begin{array}{l} z : z_2, \\ y : y_2 \end{array} \right\} = \left\{ \begin{array}{l} x : x_1 \\ z : z_2, \\ y : y_2 \end{array} \right\}$$

- Значения по умолчанию (0, false, "") опять неотличимы от отсутствия значения

Protobuf: Forward compatibility

- Совместимость “в будущее”
 - Сериализуем объект
 - Отправляем его по сети
 - Читаем с помощью старой версии объект, созданный новой версией кода
 - Ничего не должно сломаться
- Обработка такая же, как и в случае Backward compatibility
 - Но в обратную сторону

Protobuf: Forward compatibility

- Добавляем поле F
 - `{ ..., F_Id : value; ... }`
 - Значение будет проигнорировано
- Удаляем поле F
 - `{ A_Id : 5, B_Id : Hello }`
 - Поле F получит значение по умолчанию
- Делаем поле F не repeated
 - `{ F_Id : 5 }`
 - Массив значений будет состоять из одного значения
- Делаем поле F repeated
 - `{ F_Id : 5; F_Id : 6; F_Id : 42 }`
 - Логика нам известна

Protobuf: Удаление полей

- Есть описание структуры

```
message MyStruct {  
    // ... Other fields  
}
```

- Удаляем поле

- Другой разработчик создаёт поле с таким же идентификатором, но другим типом

```
message MyStruct {  
    string value = 1;  
    // ... Other fields  
}
```

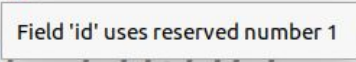
```
message MyStruct {  
    int32 id = 1;  
    // ... Other fields  
}
```

- Всё сломается!
 - Пытаемся распарсить строку как число или наоборот

Protobuf: Удаление полей

- При удалении помечаем поле как `reserved`
- Гарантирует, что никто не создаст поле с таким же идентификатором

```
message MyStruct {  
    reserved 1;  
    int32 id = 1;  
    // ... Other fields  
}
```

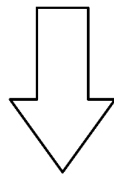
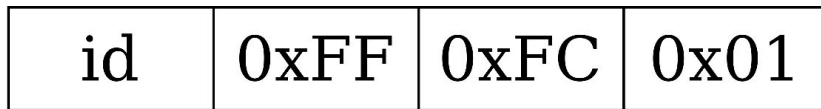


Protobuf: смена типа поля

- Лучше не надо
 - Но есть исключения
- `string <-> bytes`
 - Если в поле записывается правильный UTF-8
- `message <-> bytes`
 - Если в поле записывается закодированное представление этого сообщения
- `int32 <-> int64`
- `sint32 <-> sint64`
- `int32` не совместим с `sint32`
- `int64` не совместим с `sint64`

Compatibility & Packed fields

- Было поле типа `int32`
- Поле сделали `repeated` [`packed = true`]
- Поменялся формат представления поля



└──────────┬──────────┘ └──────────┘
Число #1 Число #2

Compatibility & Packed fields

- Вместо идентификатора поля будем хранить пару `<field_id, wire_type>`

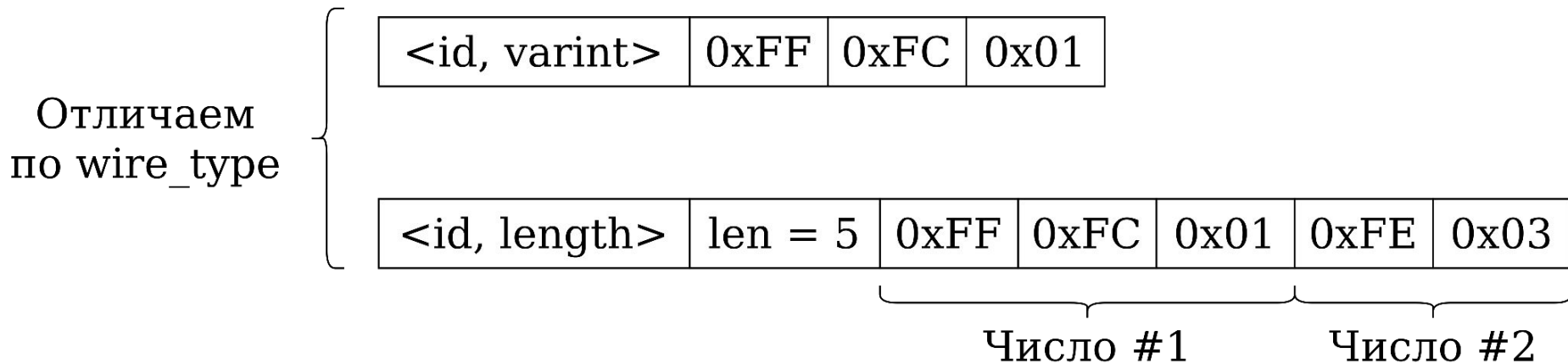
field_id
┌───────────┐
100101001 010
└──────────┘
wire_type

wire_type	тип
0	Varint
1	Fixed64
2	Length
5	Fixed32

- `eid = field_id << 3 || wire_type`

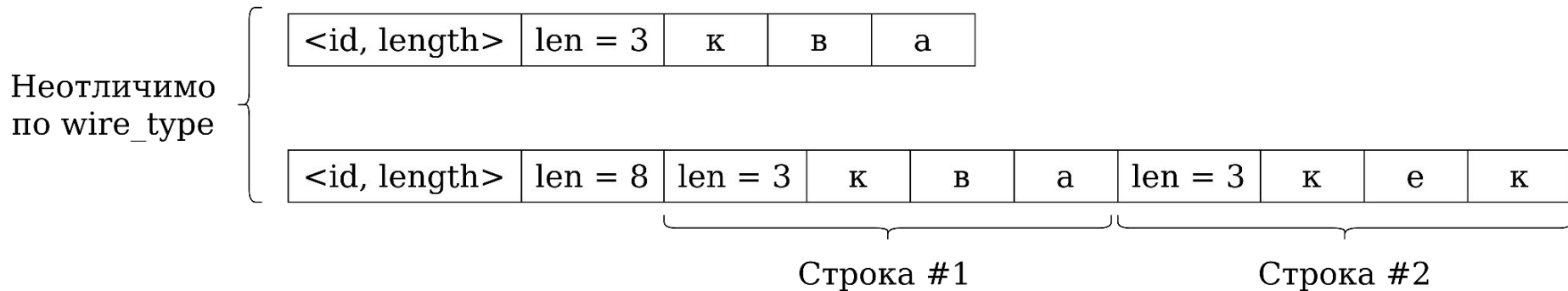
Compatibility & Packed fields

- Вместо идентификатора поля храним пару `<field_id, wire_type>`



Compatibility & Packed fields

- *Packed repeated* оптимизация не применяется для строк, структур и массивов байт
 - Теперь мы можем понять причину



Что ещё есть в Protobuf: Отображения

```
message MyStruct {  
    map<string, OtherStruct> m = 1;  
    // ... Other fields  
}
```

- Преобразуется в конструкцию справа
- Уникальность ключей гарантируется на этапе создания структуры

```
message Entry {  
    string key = 1;  
    OtherStruct value = 2;  
}
```

```
message MyStruct {  
    repeated Entry m = 1;  
    // ... Other fields  
}
```

Что ещё есть в Protobuf: oneof

```
message Response {  
    uint64 request_id = 1;  
    oneof response_body {  
        string successful_response = 2;  
        uint32 error_code = 3;  
    }  
}
```

- Не более одного поля из списка будет присутствовать в структуре
 - Гарантируется на этапе создания структуры
- Сериализация происходит по обычным правилам
 - Список key-value пар

Бинарная сериализация: другие инструменты

- Apache Avro
- Apache Thrift
- MessagePack
- CBOR
- [TL](#)
- Специализированные форматы
 - FITS
 - TIFF
 - PNG
 - WAVE
 - ...

Что почитать

- Protobuf Encoding
 - developers.google.com/protocol-buffers/docs/encoding
- Proto3 language guide
 - developers.google.com/protocol-buffers/docs/proto3
- Protobuf tutorials
 - developers.google.com/protocol-buffers/docs/tutorials
- gogo/Protobuf
 - github.com/gogo/protobuf
- Flatbuffers
 - flatbuffers.dev

Thanks for your attention



my dudes