TL (Type Language) Primer

Таблица версий документа

07 марта 2024	Добавлена секция про соответствие JSON, мелкие исправления
25 марта 2024	Уточнение про сериализацию false-значений true-полей в JSON

Общие сведения

TL это язык описания данных и формат хранения данных.

Обычно интерес к TL связан с желанием сериализовать данные и делать RPC вызовы к серверам, реализующим TL RPC, а также реализовывать такие сервера, поэтому мы почти не будем касаться этого языка с точки зрения теории типов.

С практической точки зрения, TL описывает структуры данных, в том числе RPC запросы и ответы на них, используя синтаксис, почерпнутый из функциональных языков программирования.

Формат TL отличается компактностью и высокой эффективностью, однако имеет некоторые проблемы совместимости, которые мы рассмотрим в финальном разделе документа. В сравнении с JSON или MessagePack, формат сериализации TL не может быть разобран сам по себе, без схемы, так как большая часть структуры объектов никак не отражена в сериализованном представлении.

Поэтому обычно TL-файл даётся кодогенератору для того, чтобы тот сгенерировал определения структур данных для какого-либо языка программирования, и функции (де)сериализации этих данных, а также код RPC клиентов и серверов, обменивающихся сообщениями, сериализованными в соответствующем формате.

TL-файл разделён на секции types и functions, типы это просто структуры данных, которые сериализуются в определённом формате, а функции определяют пары запрос-ответ RPC, запрос в плане сериализации ничем не отличается от типа, а сериализация ответа может зависеть от полей запроса.

Комбинаторы это выражения вида A = B, файл TL состоит из комбинаторов. Порядок комбинаторов не влияет на формат данных (но может влиять на результат кодогенерации, например на порядок объявления полей в объединении).

Комбинаторы-типы

Сначала рассмотрим комбинаторы типов, находящиеся в секции типов.

Целые типы и простые комбинаторы

Встроенный тип int это 32-битное знаковое целое, которое сохраняется в формате 4 байта Little Endian, встроенный тип long это 64-битное знаковое целое, которое сохраняется в формате 8 байта Little Endian.

При сериализации int 5 получится

```
[05 00 00 00]
```

При сериализации long 5 получится

```
[05 00 00 00] [00 00 00 00]
```

Рассмотрим следующий комбинатор

```
point x:int y:int = Point;
```

Он описывает тип Point с конструктором point и полями x:int y:int

При сериализации point значения полей сохраняются просто одно за другим. Имена полей не влияют на формат и не сохраняются.

При сериализации point 5 0 получится

```
[05 00 00 00] [00 00 00 00]
```

Продолжим комбинировать

```
rectangle a:point b:point = Rectangle;
```

При сериализации rectangle (point 5 0) (point 1 3) получится

```
[05 00 00 00] [00 00 00 00] [01 00 00 00] [03 00 00 00]
```

Видно, что при сериализации тела rectangle просто сохранилось 2 point подряд.

Ничто в байтах не указывает на то, какой именно тип был сериализован.

В примере выше point 5 0 и long 5 сохранились в одни и те же байты.

Тэги, голые и коробочные представления типа

Каждый конструктор имеет тэг (также могут использоваться термины ярлык, magic), который либо задаётся явно, либо принимается равным хэшу текста комбинатора в некотором каноническом виде (рекомендуется задавать тэги явно, иначе они будет неожиданно меняться при изменении текста комбинатора).

```
point x:int y:int = Point;
```

эквивалентно

```
point#e3fe70f4 x:int y:int = Point;
```

Тэг имеет встроенный тип #, это 32-битное беззнаковое целое, которое сохраняется в формате 4 байта Little Endian.

При сериализации **Point** 5 0 получится

```
[f4 70 fe e3] [05 00 00 00] [00 00 00 00]
```

Отличие от голого представления в том, что перед полями сохраняется тэг.

Встроенные типы обычно имеют коробочные обёртки

При сериализации **Long** 5 получится

```
[ba 6c 07 22] [05 00 00 00] [00 00 00 00]
```

При сериализации **Int** 5 получится

```
[da 9b 50 a8] [05 00 00 00]
```

Генератор кода TL следит за тем, чтобы тэги были уникальны, так что если мы используем коробочное представление, то по тэгу всегда можно понять, какой именно конструктор был сохранён.

При определении полей можно использовать как голое, так и коробочное представление типа. Имя с большой буквы (**Point**, **Long**) ссылается на коробочное, а с маленькой буквы (point, long) - на голое представление. Мнемоническое правило - предмет в коробке больше самого предмета.

Поддерживается также синтаксис, когда знак % перед именем с большой буквы (%**Point**, %**Long**) означает голое представление. Кажется, что символы % снижают читаемость кода, и лучше не использовать эту форму, впрочем некоторые считают наоборот.

```
point#e3fe70f4 x:int y:int = Point;
```

эквивалентна

```
point#e3fe70f4 x:%Int y:%Int = Point;
```

В этом документе мы используем **Жирный** шрифт для имён с большой буквы, которые являются коробочными представлениями, чтобы привлечь внимание к важности использования правильного представления.

Объединения

Типы могут иметь более одного конструктора

```
resultOk#d0fa5d20              = Result;
resultError#dd4526fd code:int = Result;
```

Тогда они называются объединениями, а типы с одним конструктором называются простыми.

При сериализации объединения сохранение тэга обязательно, поэтому объединения не имеют голого представления и могут быть использованы другими типами только в коробочном виде.

При сериализации **Result** resultOk получим

```
[20 5d fa d0]
```

При сериализации **Result** resultError 404 получим

```
[fd 26 45 dd] [94 01 00 00]
```

При чтении объединения сначала читается тэг, если он принадлежит одному из конструкторов, то читается тело соответствующего конструктора, иначе происходит ошибка.

Объединения можно расширять, добавляя конструкторы, например

```
resultErrorLine#dd4526fd code:int line:int = Result;
```

Старые клиенты будут продолжать читать значения Result, пока им не встретится неизвестный тэг нового конструктора, тогда произойдёт ошибка чтения.

Коробочное представление обычно не должно применяться для полей простых типов. Рассмотрим такое определение PointB

```
pointB#e3fe70f5 x:Int y:Int = PointB;
```

При сериализации **PointB** 5 0 получим

```
[f5 70 fe e3] [da 9b 50 a8] [05 00 00 00] [da 9b 50 a8] [00 00 00 00]
```

Тэг конструктора pointB однозначно указывает, что это именно он, а тип **Int** имеет и будет иметь в обозримом будущем ровно 1 конструктор, так что дополнительные тэги для каждого поля **Int** избыточны и просто приводят к сохранению лишней информации.

Перечисления и Bool

Если все конструкторы типа пустые, то он называется перечислением, и кодогенератор создаст для него c++ enum или аналогичный тип в других языках. При сериализации enum сохраняется только тэг конструктора (так как все поля во всех конструкторах отсутствуют). При добавлении нового непустого конструктора перечисление превратится в обычное объединение с сохранением обычной совместимости для объединений.

Кодогенератор использует имя перечисления **Bool** как аннотацию и генерирует для этого перечисления поле типа bool в языках, которые его поддерживают. С точки зрения TL тип **Bool** ничем не отличается от остальных объединений.

Маски полей

При добавлении новых полей в конструктор простого типа возникает вопрос, как читать значения, сохранённые до добавления новых полей.

Если исходные определения вот такие

```
point#e3fe70f4 x:int y:int = Point;
rectangle a:point b:point  = Rectangle;
```

То при сериализации rectangle (point 5 0) (point 1 3) получалось

```
[05 00 00 00] [00 00 00 00]
[01 00 00 00] [03 00 00 00]
```

А после добавления поля z

```
point#e3fe70f4 x:int y:int z:int = Point;
```

```
rectangle a:point b:point         = Rectangle;
```

При сериализации rectangle (point 5 0 2) point(1 3 2) получится

```
[05 00 00 00] [00 00 00 00] [02 00 00 00]
[01 00 00 00] [03 00 00 00] [02 00 00 00]
```

По байтам невозможно понять, какая версия Point используется

Можно было бы использовать коробочное представление **Point**, тогда при изменении списка полей можно было бы добавить новый конструктор.

```
pointV1#e3fe70f4 x:int y:int       = Point;
pointV2#7f42a5be x:int y:int z:int = Point;
rectangle2 a:Point b:Point         = Rectangle2;
```

При сериализации rectangle2 (**Point** pointV1 5 0) (**Point** pointV1 1 3) получится

```
[f4 70 fe e3] [05 00 00 00] [00 00 00 00]
[f4 70 fe e3] [01 00 00 00] [03 00 00 00]
```

При сериализации rectangle2 (**Point** pointV2 5 0 2) (**Point** pointV2 1 3 2) получится

```
[be a5 42 7f] [05 00 00 00] [00 00 00 00] [02 00 00 00]
[be a5 42 7f] [01 00 00 00] [03 00 00 00] [02 00 00 00]
```

Это рабочий подход, но компактность хранения и скорость чтения будет страдать. Также большим неудобством является то, что кодогенератор (в случае C++) при таком определение создаст по классу для каждого конструктора, и что-то типа std::variant, так что в каждом месте кода, где используется **Point** придётся делать выбор между конструкторами.

Маски полей позволяют решить этот вопрос по-другому,

```
point fields_mask:# x:fields_mask.0?int y:fields_mask.1?int = Point;
rectangle a:point b:point = Rectangle;
```

Маска полей сама по себе является обычным полем типа #, любое поле определённое после неё может ссылаться на её бит. Если бит установлен - поле будет прочитано, а если сброшен - поле будет пропущено и установлено в default-значение, например 0 для целых. Маска полей это обычное поле и его нужно не забывать устанавливать в правильное значение перед сохранением объекта, иначе часть полей не будет сохранены.

При сериализации rectangle (point 3 5 0) point(3 1 3) получится

```
[03 00 00 00] [05 00 00 00] [00 00 00 00]
[03 00 00 00] [01 00 00 00] [03 00 00 00]
```

Сначала читается маска полей (3), затем, так как оба бита установлены в 1, читается поля x и y.

Если мы расширим тип point

```
point fields_mask:# x:fields_mask.0?int
      y:fields_mask.1?int z:fields_mask.2?int = Point;
rectangle a:point b:point                     = Rectangle;
```

То при сохранении значения rectangle (point 7 5 0 2) point(7 1 3 2) запишется

```
[07 00 00 00] [05 00 00 00] [00 00 00 00] [02 00 00 00]
[07 00 00 00] [01 00 00 00] [03 00 00 00] [02 00 00 00]
```

Кажется, что используется столько же байтов, сколько и при использование коробочного представления Point, однако есть плюсы - тип Point в языке наподобие C++ вместо std::variant может остаться простой структурой, и появляется возможность более компактного хранения, если часть полей не нужны. Например в случае Point можно было бы перед сохранением поля со значением 0 просто убрать соответствующий ему бит маски полей.

Например, при сохранении значения rectangle (point 1 5 0 0) point(0 0 0 0) запишется

```
[01 00 00 00] [05 00 00 00] [00 00 00 00]
```

Маски полей являются обычными полями, масок может быть определено сколько угодно и в любом порядке, другие поля могут ссылаться на любой бит любой маски, и маски могут ссылаться на другие маски. Также несколько полей могут ссылаться на один бит одной маски, тогда если бит не установлен, то просто будут пропущены все поля.

```
funnyMasks x:int k:# a:int b:k.0?int m:k.1?# c:k.0?int
           d:m.31?int e:int g:m.31?int = FunnyMasks;
```

Однако самое интересное свойство маски полей это то, она может передаваться, как параметр снаружи, а не храниться в каждом экземпляре объекта в памяти и в сериализованных байтах.

#-параметры (Nat-параметры)

Рассмотрим следующие определение точки

```
point {F:#} x:F.0?int y:F.1?int = Point F;
```

Конструктор точки теперь имеет #-параметр F. Параметр не хранится в точке, а передаётся при сохранении и чтении. Так что теперь точку можно использовать только если передать ей параметр. Имена и количество параметров слева и справа должны совпадать.

Вот новое определение прямоугольника

```
rectangle fields_mask:# a:(point fields_mask)
          b:(point fields_mask) = Rectangle;
```

Поле с именем fields_mask хранится в rectangle обычным образом, и передаётся в каждый point при чтении и записи.

При сериализации значения rectangle 3 (point 5 0) point(1 3) запишется

```
[03 00 00 00] [05 00 00 00] [00 00 00 00] [01 00 00 00] [03 00 00 00]
```

Теперь при расширении точки новым полем достаточно установить правильную маску не в каждой точке, а в каждом прямоугольнике.

```
point {F:#} x:F.0?int y:F.1?int z:F.2?int = Point F;
```

Например,при сохранении значения rectangle 7 (point 5 0 2) point(1 3 2) запишется

```
[07 00 00 00] [05 00 00 00] [00 00 00 00] [02 00 00 00]
              [01 00 00 00] [03 00 00 00] [02 00 00 00]
```

А при сохранении значения rectangle 3 (point 5 0 2) point(1 3 2) запишется

```
[03 00 00 00] [05 00 00 00] [00 00 00 00] [01 00 00 00] [03 00 00 00]
```

То есть тот же формат, что был до добавления поля. Устанавливая маску полей типа прямоугольник мы теперь управляем форматом сохранения полей точек.

Однако можно пойти ещё дальше, установить маску поля точек один раз для самого внешнего обьекта, и передать её рекурсивно во все типы, которые в ней нуждаются.

```
point {F:#} x:F.0?int y:F.1?int z:F.2?int = Point F;
rectangle {F:#} a:(point F) b:(point F)   = Rectangle F;
picture point_fields_mask:#
        r:(Rectangle point_fields_mask)   = Picture;
```

Такой подход сочетает несколько плюсов - форматом хранения легко управлять, установив маску в небольшом количестве мест, большинство объектов не хранит лишних полей, и при сохранении достигается компактность, особенно если сохраняется много одинаковых объектов.

Nat-параметры-константы можно задавать в десятичном виде, например

```
rectangle2D r:(rectangle 3) = Rectangle2D;
rectangle3D r:(rectangle 7) = Rectangle3D;
```

Для битовых масок рекомендуется использовать оператор +

```
rectangle2D r:(rectangle (1 + 2)) = Rectangle2D;
rectangle3D r:(rectangle (1 + 2 + 4)) = Rectangle3D;
```

Встроенные массивы

Рассмотрим следующее определение

```
point x:int y:int = Point;
triangle color:int a:3*[point]  = Triangle;
```

Поле a является встроенным массивом (далее в документе просто массивом) из 3 точек, 3 называется множителем.

При сериализации triangle 127 [(point 5 0) (point 1 3) (point 6 4)] получится

```
[7f 00 00 00] [05 00 00 00] [00 00 00 00]
              [01 00 00 00] [03 00 00 00] [06 00 00 00] [04 00 00 00]
```

Как мы можем заметить, никакой информации о размере в байтах не записывается. При чтении треугольника мы знаем, что хранится именно 3 именно точки.

Если размер массива не фиксированный, например нам нужно определить многоугольник, то размер придётся сохранить в поле типа # и использовать, как множитель.

```
point x:int y:int = Point;
polygon color:int n:# a:n*[point] = Polygon;
```

При сериализации polygon 127 2 [(point 5 0) (point 1 3)] получится

```
[7f 00 00 00] [02 00 00 00] [05 00 00 00] [00 00 00 00]
                           [01 00 00 00] [03 00 00 00]
```

Здесь подобно маскам полей n является обыкновенным полем типа #, значение которого при чтении передаётся массиву, чтобы он прочитал ровно столько элементов, сколько было записано.

Например, мы бы могли захотеть назначить каждой точке многоугольника вес

```
polygon color:int n:# a:n*[point] weight:n*[int] = Polygon;
```

Здесь оба массива при чтении получат один и тот же размер n, так что каждый прочитает ровно n элементов. При сохранении (если для хранения массивов в языке кодогенерации используются динамические массивы, например std::vector) размеры массивов должны быть одинаковыми, иначе произойдёт ошибка записи.

Также подобно маске полей размер массива можно получать извне в #-параметре. Рассмотрим следующее определение точки в многомерном пространстве

```
point {dim:#} x:dim*[int] = Point dim;
```

При сериализации point{0}, point {1} 5, point {2} 5 0 и point {3} 5 0 2 получится

```
[ ]
[05 00 00 00]
[05 00 00 00] [00 00 00 00]
[05 00 00 00] [00 00 00 00] [02 00 00 00]
```

Параметр не сериализуется вместе с точкой, сериализуются только элементы массива.

Тогда определение многоугольника в многомерном пространстве будет

```
polygon {dim:#} color:int n:# a:n*[(point dim)] = Polygon dim;
```

Параметр можно либо зафиксировать во внешнем типе

```
picture2d n:# polygons:n*[(polygon 2)] = Picture2d;
```

Либо передать в него какое-то поле типа #.

```
pictureXd dim:# n:# polygons:n*[(polygon dim)] = PictureXd;
```

Для языков, где есть отдельные коллекции фиксированного и динамического размера, кодогенератор в случае фиксированных и нефиксированных размеров генерирует разные типы. В случае C++, к примеру, это std::vector<T> или std::array<T, N>, в случае golang это слайс []T или массив [N]T и так далее.

Встроенные массивы являются единственным типом для коллекций в TL, остальные коллекции являются определяемыми пользователем обёртками. Кодогенератор для конкретного языка может использовать аннотации для генерации, например, ассоциативных массивов вместо обычных для некоторых типов.

Рассмотрим два популярных типа-обёртки для массивов. Это **Tuple** и **Vector**.

Type-параметры

Рассмотрим определение

```
vector#1cb5c415 {t:Type} n:# a:n*[t] = Vector t;
```

Как нетрудно догадаться, оно определяет шаблонный конструктор, способный хранить размер массива и сам массив элементов.

Теперь вместо

```
picture2d n:# polygons:n*[(polygon 2)] = Picture2d;
```

Можем писать

```
picture2d polygons:(vector (polygon 2)) = Picture2d;
```

Без изменения формата сериализации. Тип **Vector**, в отличие от встроенного массива, имеет коробочное представление. Чаще всего оно не применяется, так как тип **Vector** имеет один конструктор и вряд ли будет расширен новыми конструкторами в обозримом будущем. Поэтому обычно не стоит использовать коробочное представление вектора.

```
picture2d polygons:(Vector (polygon 2)) = Picture2d; // ПЛОХО
```

Сам параметр типа вектора тоже может иметь коробочное представление, тогда при сохранении каждого элемента будет сохранён тэг конструктора элемента. В подавляющем большинстве случаев такое использование уменьшает компактность без каких-либо плюсов.

```
picture2d polygons:(vector (Polygon 2)) = Picture2d; // ПЛОХО
```

Сравним форматы сериализации 4 вариантов (коробочный-голый) векторов целых.

vector {int} [5, 0]

```
[02 00 00 00] [05 00 00 00] [00 00 00 00]
```

Vector {int} [5, 0]

```
[15 c4 b5 1c] [02 00 00 00] [05 00 00 00] [00 00 00 00]
```

vector {**Int**} [5, 0]

```
[02 00 00 00] [da 9b 50 a8] [05 00 00 00] [da 9b 50 a8] [00 00 00 00]
```

И, наконец, **Vector** {**Int**} [5, 0]

```
[15 c4 b5 1c] [02 00 00 00] [da 9b 50 a8] [05 00 00 00]
                            [da 9b 50 a8] [00 00 00 00]
```

Рассмотрим другой тип-обёртку массива

```
tuple#9770768a {t:Type} {n:#} a:n*[t] = Tuple t n;
```

Такое определение кортежа полностью эквивалентно встроенному массиву из n элементов типа t, но позволяет использовать массив там, где требуется коробочное представление, например при желании возвратить встроенный массив из функции.

Определения

```
combinator ... a:n*[t] ... = Combinator;
```

И

```
combinator ... a:(tuple t n) ... = Combinator;
```

эквивалентны, какое использовать - дело вкуса. Встроенный массив выглядит предпочтительней.

Анонимные типы

При определении массивов в квадратных скобках можно указывать не только тип, но и список полей.

```
triangle a:3*[a:int b:int] = Triangle;
```

В таком случае кодогенератор придумает имя для нового типа, и трансформирует определение в

```
triangleElem a:int b:int = TriangleElem;
```

```
triangle a:3*[triangleElem] = Triangle;
```

Такие элементы не имеют тэга, и не могут быть коробочными. Если имя, придуманное кодогенератором, неудачно (часто это так), можно сделать трансформацию вручную.

Определения полей внутри скобок могут ссылаться на параметры типа и #-параметры, а также на предшествующие поля, так что трансформация может быть довольно нетривиальной.

```
funnyAnon {n:#} {X:Type} k:# a:k*[b:n.0?k*[(Pair int X)]
                         c:k.0?n*[int]] = FunnyAnon n X;
```

Анонимные поля и множители

Если размер встроенного массива не указан явно, считается, что размером является последний аргумент конструктора, если это первое поле.

```
replace1 {n:#} a:[int] = Replace1 n;
```

или предыдущее поле в остальных случаях

```
replace2 n:# a:[int] m:# b:[int] = Replace2;
```

Имена полей для множителей могут быть опущены, так как они не участвуют в сериализации, например

```
replace6 # a:[int] = Replace6;
replace7 #   [int] = Replace7;
```

Существует договорённость, что если у конструктора простого типа X одно безымянное поле типа Y, то кодогенератор постарается везде, где упоминается тип X, использовать вместо этого тип Y (с учётом того, что коробочное представление должно по-прежнему содержать тэг типа X).

Во всех остальных случаях нужно указывать явно имена всех полей, в противном случае кодогенератор вернёт ошибку.

Вскоре мы будем требовать двоеточие перед типом безымянного поля, например

```
replace6 :# a:[int] = Replace6;
replace7 :#  :[int] = Replace7;
```

Это значительно упрощает грамматику, делая её контекстно независимой.

Комбинаторы-функции

Секция функции

TL разделён на секции types и functions, пока мы рассматривали комбинаторы типов, находящиеся в секции типов, которые определяли структуры данных и их формат сериализации.

Теперь рассмотрим комбинаторы в секции функций.

Рассмотрим комбинатор

```
getWeights#f53ad7be user_id:int count:int = (Vector int);
```

Этот комбинатор определяет функцию RPC с именем getWeights, которая получает аргументы user_id и count и возвращает вектор целых.

Фактически определяется формат запроса (левая часть) и ответа (правая часть). Формат запроса ничем не отличается от конструктора типа, сохраняется тэг, и затем все поля.

При вызове getWeights 127 5 будут отправлены следующие байты

```
[be d7 3a f5] [7f 00 00 00] [05 00 00 00]
```

А в ответ придёт, например, такой ответ (**Vector** int [5, 0])

```
[15 c4 b5 1c] [02 00 00 00] [05 00 00 00] [00 00 00 00]
```

И левая и правая часть функции всегда при сериализации имеет коробочное преставление, левая, так как сервер при получении запроса должен правильно интерпретировать его, для чего используется тэг, а правая просто по историческим причинам.

Здесь надо сказать, что для шаблонов с Type-параметрами тэга недостаточно, чтобы полностью восстановить тип обьекта, поскольку для **Vector** int, **Vector** long и любого другого вектора будет сохранён лишь тэг конструктора вектора, а для **Vector Int**, **Vector Long** тэги будут сохранены в каждом обьекте, если их больше 0, и не будут сохранены вовсе если длина вектора 0.

Предположительно при сохранении вектора с коробочным аргументом (и других шаблонов) тэг аргумента должен был сохраняться не для каждого элемента, а после тэга вектора. Однако это не было реализовано, так что часто делают тип-обёртку, например

getWeightsResult и возвращают его. В случае чего в этот тип можно добавить конструкторы, расширив набор возможных ответов функции.

Вскоре мы будем поддерживать новый синтаксис для объявления функций

```
getWeights#f53ad7be ... => (Vector int);
```

Который позволяет объявлять функции в любой секции. Сами же секции становятся ненужными.

Шаблонные параметры функций

Все используемые шаблоны с type-параметрами должны использовать конкретные типы аргументов, так как запрос и ответ является самым внешним объектом и ничем не параметризуется.

Левая часть комбинатора сама не может иметь никаких шаблонных параметров, а в правой части разрешено ссылаться на поля левой части. Это логично, ведь запрос сериализуется сам по себе, а ответ сериализуется для конкретного запроса.

Поэтому имея тип polygon, рассмотренный выше

```
polygon {dim:#} color:int n:# a:n*[(point dim)] = Polygon dim;
```

Можно определить запрос, возвращаемый многоугольники разной размерности

```
@read getPolygons dim:# user_id:int = (Polygon dim);
```

При сериализации ответа сервером и десериализации клиентом будет использована ровно та размерность, которая передана клиентом. Этот же приём используется для возврата дополнительных полей в ответе и организации обратной совместимости.

Клиент устанавливает маску известных ему полей, и сервер возвращает только их. Поля, неизвестные клиенту пропускаются при сериализации

```
@read getUser fields_mask:# user_id:int = (User fields_mask);
```

Маски полей ответа

Рассмотрим этот механизм подробнее, представим, что изначально пользователя содержал поля id и name.

```
user id:int name:string = User;
```

Теперь расширим пользователя, добавив в него поле рост под маской полей.

```
user {fields_mask:#} id:int name:string height:fields_mask.0?int
     = User fields_mask;
```

Старые клиенты вызывают getUser с нулевой маской полей, это значит, что при сохранении ответа новое поле не будет сериализовано.

Новые клиент вызывают getUser с выставленным битом маски полей, а значит при сохранении ответа новое поле будет сериализовано.

Это важно, так как сериализованный TL не содержит никакой структуры, и пропустить поле, если оно добавлено, невозможно, так как неизвестен его тип и размер, сколько байтов пропускать.

При расширении пользователя, чтобы связь запроса и ответа была очевидна, настоятельно рекомендуется добавить в запрос поле типа true, соответствующее новому полю ответа.

```
@read getUser fields_mask:# user_id:int
              result_user_height:fields_mask.0?true
    = (User fields_mask);
```

Это позволит в вызывающем коде удобно проставить соответствующий бит запроса. В скором будущем TL компилятор будет генерировать предупреждение, если такое поле в запросе не проставлено.

Рассмотрим более сложный случай, запрос, возвращающий пользователя и точку. Пользователя расширили полем рост, а точку координатой z.

```
user {fields_mask:#} id:int name:string height:fields_mask.0?int
     = User fields_mask;
point {fields_mask:#} x:int y:int z:fields_mask.0?int
     = Point fields_mask;
```

Оба типа имеют маску полей, причём бит 0 независимо используется для расширения обоих структур данных. Тогда в запросе ответе нужно будет сделать 2 маски полей, для каждого возвращаемого типа.

```
@read getUser user_fields_mask:#
              point_fields_mask:#
              user_id:int
              result_user_height:user_fields_mask.0?true
```

```
                point_z:point_fields_mask.0?true
                = GetUserResult user_fields_mask point_fields_mask;
getUserResult {user_fields_mask:#} {point_fields_mask:#}
                u:(user user_fields_mask)
                p:(point point_fields_mask)
                = GetUserResult user_fields_mask point_fields_mask;
```

Из примера понятно, что маска полей запроса и маска полей ответа в общем случае не совпадают, и наиболее корректный подход - использовать маску полей запроса только для полей запроса и ответа, а для всех возвращаемых объектов, имеющих маски полей, добавить отдельную маску в запросе.

Рассмотрим трансформацию API, если мы начинали с определения

```
user id:int name:string = User fields_mask;
getUserResult {fields_mask:#} u:user = GetUserResult fields_mask;
@read getUser fields_mask:# = GetUserResult fields_mask;
```

То затем, когда пользователя расширили полем height, оно без потери совместимости старых клиентов трансформируется в

```
user {fields_mask:#} id:int name:string height:fields_mask.0?int
     = User fields_mask;
getUserResult
     {fields_mask:#} {user_fields_mask:#}
     u:(user user_fields_mask)
     = GetUserResult fields_mask user_fields_mask;
@read getUser fields_mask:#
     user_fields_mask:fields_mask.0?#
     result_user_height:user_fields_mask.0?true
     = GetUserResult fields_mask user_fields_mask;
```

Маска полей пользователя, поскольку её не было в оригинальном запросе, добавлена сама под битом маски полей запроса. А уже установкой её бита контролируется, будет ли сериализовано новое поле пользователя.

Затем, когда в ответ добавляется точка, определения станут

```
user {fields_mask:#} id:int name:string height:fields_mask.0?int
     = User fields_mask;
point x:int y:int = Point fields_mask;
getUserResult
     {fields_mask:#} {user_fields_mask:#}
     u:(user user_fields_mask)
     p:fields_mask.1?point
```

```
      = GetUserResult fields_mask user_fields_mask;
@read getUser fields_mask:#
      user_fields_mask:fields_mask.0?#
      result_point:fields_mask.1?true
      result_user_height:user_fields_mask.0?true
      = GetUserResult fields_mask user_fields_mask;
```

Бит 1 маски полей запроса контролирует, будет ли в ответе точка.

И затем, когда в точку добавляется координата z, определения станут

```
user {fields_mask:#} id:int name:string height:fields_mask.0?int
      = User fields_mask;
point {fields_mask:#} x:int y:int z:fields_mask.0?int
      = Point fields_mask;
getUserResult
      {fields_mask:#} {user_fields_mask:#} {point_fields_mask:#}
      u:(user user_fields_mask)
      p:fields_mask.1?(point point_fields_mask)
      = GetUserResult
           fields_mask user_fields_mask point_fields_mask;
@read getUser fields_mask:#
      user_fields_mask:fields_mask.0?#
      result_point:fields_mask.1?true
      point_fields_mask:fields_mask.2?#
      result_user_height:user_fields_mask.0?true
      result_point_z:point_fields_mask.0?true
      = GetUserResult
           fields_mask user_fields_mask point_fields_mask;
```

Мы снова добавили новое поле в запрос - маску полей точки, используя маску полей запроса.

Итак, вызывающий независимо контролирует, будет ли в ответе точка, и будет ли у точки и пользователя каждое дополнительное поле.

Не всегда такая дополнительная сложность стоит сохранения совместимости со старыми клиентами, разобраться в хитросплетениях масок выше можно только имея определённый опыт, и то не сразу . В какой-то момент, когда API перестало значительно меняться, можно добавить функцию getUser2, и перевести всех клиентов на неё, а оригинальное определение getUser удалить.

Пространства имён

Как типы, так и функции могут группироваться по пространству имён, например

```
@read notify.getWeights user_id:int
                        count:int = (Vector int);
@write notify.setWeights user_id:int
                         weights:(vector int) = notify.Result;
```

При генерации определений типов кодогенератор помещает их в соответствующее пространство имён (в этом случае notify), если это поддерживает язык генерации.

А также для каждого пространства имён генерируется код RPC-сервера, реализующий вызовы для каждой функции в этом пространстве имён.

Все типы могут свободно ссылаться на типы из любых пространств имён, инкапсуляции не происходит. Тэги также должны быть уникальны глобально, уникальности в пространстве имён недостаточно.

Пространства имён не могут быть вложенными, и их имена должны начинаться с маленькой буквы.

Аннотации

Функции и остальные комбинаторы могут иметь аннотации

```
@read      getWeights user_id:int count:int = (Vector int);
@write     setWeights user_id:int weights:(vector int) = Result;
@readwrite resetWeights user_id:int = Result;
```

Которые доступны в сгенерированном коде, например, помогают прокси определить, можно ли такой вызов направлять на readonly реплики, и можно ли повторять вызов при возникновении ошибок. Рекомендуется всегда устанавливать аннотацию чтения-записи для функций.

Генератор кода TL интерпретирует часть аннотаций, однако поддерживаются любые аннотации.

Ссылка на аннотации во внутренней документации VK confluence (internal link)

Первые 4 атрибута являются взаимоисключающими, они соответствуют 2 битам во флагах.

@write	Запрос пишущий. Его не может выполнить движок отмеченный как r в прокси, или запущенный с опцией -r.
@read	Запрос читающий. Его не может выполнить движок отмеченный как w (не rw!) в прокси, или запущенный с опцией -w (такое бывает например с логами или news2).
@readwrite	Запрос и пишущий и читающий. Его может выполнить только движок отмеченный как rw в проксе. Типичным примером такого запроса является memcache.append, которому нужно и уметь писать бинлог, и честно знать состояние. В целом, различать его от @write когда движки не умеют в write-only-mode не очень важно, но для консистентности лучше различать Если у движка в будущем не планируется поднятие реплик/write-only движков, потому что движок в целом к этому не готов, то все запросы, по котором не понятно, что они точно @read , нужно помечать как @readwrite.
@any	Запрос может быть выполнен любым движком, вне зависимости от read/write режимов. Типичным примером является запрос engine.stat и подобные.
@internal	Запрос внутренний, и не должен посылаться из php в разумных ситуациях. Это в будущем приведет к не генерации типизированных php классов для таких запросов, и позволит например более просто проверять, что запрос пришел из разрешенного места, не трекая списки вручную.
@kphp	Is function processed by kphp rpc server

Некоторые детали

Тип True

```
true#3fedd339 = True;
```

Традиционно коробочное представление типа **True** используется для возврата пустых значений из функций.

```
@readwrite resetWeights#261f6898 user_id:int = True;
```

Голое представление типа true а также любого другого типа без полей не занимает места при сериализации. Этот факт можно использовать для оптимизации булевых полей.

Рассмотрим наивное использование **Bool**

```
getPoint fields_mask:# option0:Bool
                       option1:Bool
                       option2:Bool = Point;
```

При сериализации getPoint 0 boolTrue boolTrue boolFalse получится

```
[00 00 00 00] [b5 75 72 99] [b5 75 72 99] [37 97 79 bc]
```

Поскольку **Bool** это перечисление, для каждого значения сохраняется тэг конструктора boolTrue#997275b5 или boolFalse#bc799737.

Использование маски полей вместе с конструктором true позволяет использовать бит маски вместо **Bool**

```
getPoint fields_mask:# option0:fields_mask.0?true
                       option1:fields_mask.1?true
                       option2:fields_mask.2?true = Point;
```

При сериализации getPoint 3 (маска полей option0, `option1`) получится

```
[03 00 00 00]
```

Сохранилась только маска полей, потому, что размер значений true равен 0. Также при добавлении новых полей типа true формат сериализации не меняется, что позволяет убирать и добавлять их в количестве, ограниченном только числом свободных бит в маске полей.

Использование при такой оптимизации коробочного представления **True** вместе с маской полей, либо случайное использование типа **Bool** вместо true является частой ошибкой. Например.

```
getPoint fields_mask:# option0:fields_mask.0?True
                       option1:fields_mask.1?True
                       option2:fields_mask.2?True = Point;
```

При сериализации getPoint 3 (маска полей option0, `option1`) получится

```
[03 00 00 00] [3f ed d3 39] [3f ed d3 39]
```

Здесь кроме маски полей для каждого установленного бита сохранился тэг конструктора **True**

При использовании же типа **Bool**

```
getPoint fields_mask:# option0:fields_mask.0?Bool
                       option1:fields_mask.1?Bool
                       option2:fields_mask.2?Bool = Point;
```

Для каждого установленного бита маски полей ещё и запишется тэг одного из конструкторов **Bool**, так что получится тройная логика)

Все встроенные типы

Все значения сохраняются с порядком байтов Little Endian

Имя TL	Формат	Пример значения
# (nat)	32-бит целое беззнаковое	`[15 c4 b5 1c]`
int	32-бит целое со знаком	`[05 00 00 00]`
long	64-бит целое со знаком	`[05 00 00 00] [00 00 00 00]`
float	32-бит IEEE float	`[40 49 0f db]`
double	64-бит IEEE float	`[40 09 21 fb] [54 44 2d 18]`
string	Если len < 0xfe, то первый байт длина, дальше байты строки, затем padding до 32 битной границы.	`[04 k e y ] [s 00 00 00]`

	Если 0xfe <= len < 2^24, то первый байт 0xfe, затем 3 байта длина, дальше байты строки, затем padding до 32 битной границы.	`[fe ff 00 00] [t h i s ]...`
		
		`В сумме 255 байтов строки`
		
		`...[i n g 00]`

	Если 2^24 <= len < 2^56, то первый байт 0xff, затем 7 байтов длина, дальше байты строки, затем padding до 32 битной границы.
	Этот вариант пока реализован не во всех библиотеках VK
		`[ff 01 00 00] [00 01 00 00]`
		`[t h i s ] [i s _ a ]...`
		
		`В сумме 2^32 + 1 байтов строки`
		
		`...[g 00 00 00]`

	Строки с 2^56 <= len не поддерживаются)	
n*[T]	n элементов T подряд без префиксов, суффиксов или разделителей	

Поскольку размеры всех встроенные типов кратны 4 байтам, размер любого сериализованного объекта в TL кратен 4 байтам.

Набор встроенных типов, в принципе, легко расширяются при необходимости, если нужен какой-то новый встроенный тип, поскольку встроенные типы не имеют тэгов.

Содержимое строки не интерпретируется, это могут быть как строки в любых кодировках, так и какие-то бинарные данные.

Коробочные обёртки, наделяющие встроенные типы тэгами, с точки зрения TL ничем не отличаются от остальных определённых пользователем типов.

```
int#a8509bda ? = Int;
long#22076cba ? = Long;
```

Синтаксис с вопросительным знаком является устаревшим и эквивалентен (овый синтаксис поддерживается только новым tlgen)

```
int#a8509bda int = Int;
long#22076cba long = Long;
```

Обёрткам встроенных типов разрешается определять конструктор с совпадающим именем, так как голое представление обёртки не отличается от голого представления её содержимого.

Если нужно, можно создать сколько угодно обёрток (поддерживается только новым tlgen)

```
int32#7934e71f int = Int32;
int64#c96607df long = Int64;
```

Взаимное соответствие с JSON

Для распечатки, анализа а также взаимодействия с подсистемами, не требующими высокой скорости удобно иметь стандартное представление объектов TL в каком-либо текстовом формате. JSON идеально подходит для этой задачи.

Общие принцип

При сохранении по возможности делается оптимизация, когда пустые значения (нулевые числа, пустые строки, false, пустые массивы) не записываются.

При чтении поля если оно отсутствует в JSON, то ему присваивается пустое значение (0 для числовых типов, пустая строка для строк, пустой массив для массивов, объект со всеми полями установленными в пустое значение для структур, первый конструктор объединения и т.д.)

Null не используется при записи и не поддерживается при чтении, для пустого массив нужно использовать конструкцию [], для пустого объекта {}, либо вообще не указывать соответствующее поле.

Мы стараемся максимально оградить пользователей JSON от TL-специфичных вещей, например при сохранении объектов сгенерированным кодом маски полей сохраняются как есть, при чтении делается всё возможное для того, чтобы маски можно было не указывать и их значения восстанавливались. В случае, где пользователю приходится работать с масками полей и явными длинами массивов при чтении производится максимальная валидация и во всех сомнительных случаях генерируются ошибки.

Bool

Сохраняется и читается, как JSON Bool

Числа

Сохраняются, как JSON Number. Читаются, как JSON Number или JSON String, содержащий десятичное представление числа.

NaN, +-Inf сохраняются, как строки “NaN”, “+Inf”, “-Inf”, если клиент не ожидает подобную строку вместо числа, произойдет ошибка не его стороне.

Строки

В TL строки могут содержать любые последовательности байтов, а в JSON строки должны содержать только валидные символы UTF-8. Поэтому, при сохранении, если строка содержит только валидные символы, она сохраняется, как строка. В противном же случае вместо строки сохраняется объект с единственным полем “base64”.

Например, для комбинатора

```
foo str:string bin:string = Foo;
```

Если str содержит байты строки “good”, а bin байты строки “bye”, то объект будет записан в JSON, как

```
{
  "str": "good",
  "bin": "bye"
}
```

Если же bin содержит байты 0xf0 0xf1 0xf2 0xf3, то объект будет записан в JSON, как

```
{
  "str": "good",
  "bin": { "base64" : "8PHy8w==" }
}
```

При чтении строки поддерживаются оба формата независимо от содержимого строки.

Простые комбинаторы

Сохраняются в JSON Object, где имена полей соответствуют именам полей в TL, например комбинатор

```
memcache.strvalue value:string flags:int = memcache.Value;
```

Может быть сохранён, как

```
{
  "value":"Hello",
  "flags":1
}
```

Если поле не зависит от маски полей, то оно будет сохранено только если значение непустое, например если флаги содержат 0, то комбинатор выше будет сохранён, как

```
{
  "value":"Hello"
}
```

Если поле зависит от маски полей, то оно будет добавлено в JSON при сохранении строго если установлен бит маски, даже если значение поля пустое.

При чтении же поля, зависящего от маски полей, применяется следующая таблица

Бит маски полей	Поле указано явно	Результат чтения
1	Да	Прочитано
1	Нет	Установлено в пустое значение
0	Да	Если маска полей внутренняя, поле прочитано, а соответствующий бит маски поля установлен в 1. Мотивация - позволяет вызывающему не знать какому биту маски соответствует поле и часто вообще не указывать явно маску полей. Если маска полей внешняя, ошибка чтения. Мотивация - предотвратить игнорирование явно указанного поля в ситуации, когда мы не можем установить в 1 соответствующий бит маски полей..
0	Нет	Установлено в пустое значение

Если при чтении попадается неизвестное поле, происходит ошибка чтения. Это сделано, чтобы не допустить опечаток.

Поля сохраняются в детерминистском порядке, например объявления или алфавитном, как удобно кодогенератору.

Объединения

Превращаются в JSON Object с полями type, value, где поле типа содержит имя конструктора.

Поле value может быть опущено, тогда оно будет установлено в пустое значение.

Например, тип

```
memcache.not_found                    = memcache.Value;
memcache.longvalue x:long flags:int   = memcache.Value;
memcache.strvalue x:string flags:int = memcache.Value;
```

Может быть сохранён, как

```
{
  "type":"memcache.not_found"
}
{
  "type":"memcache.longvalue",
```

```
    "value":{
      "x":5
    }
  }
  {
    "type":"memcache.strvalue",
    "value":{
      "x":"Hello",
      "flags":1
    }
  }
```

Так как для чтения и интерпретации value нужно знать type, лучше указывать type до value иначе чтение будет менее эффективным.

Также можно вместо объекта указать JSON String с типом, например вместо

```
  {
    "type":"memcache.not_found"
  }
```

Можно указать

```
  "memcache.not_found"
```

Это сделано для совместимости с перечислениями.

Перечисления

Фактически являются разновидностью объединения.

Сохраняются в строку, содержащую имя конструктора.
Например, тип

```
  memcache.getQueryType = memcache.QueryType;
  memcache.delQueryType = memcache.QueryType;
```

Может быть сохранён: как

```
  "memcache.getQueryType"
  "memcache.delQueryType"
```

Также можно вместо строки указать объект с полем type либо с полями type, value, например вместо

```
"memcache.getQueryType"
```

Можно указать

```
{
  "type":"memcache.getQueryType"
}
```

Либо

```
{
  "type":"memcache.getQueryType",
  "Value":{}
}
```

Это сделано для совместимости с объединениями.

Встроенные массивы и их производные

Превращаются в JSON Array.

Например, тип

```
engine.status pids:(vector int) time:int = engine.Status;
```

Может быть сохранён, как

```
{
  "pids":[1,5,20],
  "time":100
}
```

Тип

```
engine.stat memory:int bytes:long = engine.Stat;
engine.statData data:(vector engine.stat) = engine.StatData;
```

Может быть сохранён, как

```
{
  "data":[
    {"memory":1,"bytes":2},
```

```
    {"memory":10,"bytes":20},
    {"memory":100,"bytes":200}
  ]
}
```

Тип

```
liked.item counters:(vector (tuple int 8)) flags:# = liked.Item;
```

Может быть сохранён, как

```
{
  "counters":[
    [0,1,2,3,4,5,6,7],
    [10,11,12,13,14,15,16,17]
  ],
  "flags":1
}
```

Если размер массива зависит от Nat-параметра, размер в json должен точно совпадать с параметром.

Например, тип

```
dependent n:# data:n*[int] = Dependent;
```

Может быть сохранён, как

```
{
  "n":5,
  "data":[0,1,2,3,4]
}
```

При этом, если размер массива будет не равен 5, произойдёт ошибка чтения.

Словари

Если массив пар является словарём (имя содержит строку Dictionary, либо (в скором будущем) помечен аннотацией @dictionary), и ключ является строкой или числом (возможно косвенно, через typedef), то такой массив пар будет сохранён в JSON, как отсортированный по ключу словарь, при этом возможные дубликаты пар по ключу будут выброшены. При чтении таких массивов поддерживается как словарь, так и массив пар.

Например, тип

```
logs.type type:string desc:(dictionary string) = logs.Type;
```

Может быть сохранён, как

```
{
  "type":"internal",
  "desc":{
    "a":"alpha",
    "b":"beta”
  }
}
```

При этом при чтении поддерживается форма

```
{
  "type":"internal",
  "desc":[
    {"key":"a", "value":"alpha"},
    {"key":"b", "value":"beta”}
  ]
}
```

А тип

```
tree_stats.periods
  counters_long:(intKeyDictionary (intKeyDictionary long))
  = tree_stats.Periods;
```

Может быть сохранён, как

```
{
  "counters_long":{
    "1":{"10":100,"11":101},
    "2":{"20":200,"21":201}
  }
}
```

При этом при чтении поддерживается форма

```
{
  "counters_long":{
    "1":[{"key":"10", "value":100},{"key":"11", "value":101}],
```

```
    "2":[{"key":"20", "value":200},{"key":"20", "value":201}]
  }
}
```

Maybe

Превращается в JSON Object. Если значение установлено, объект будет содержать поля value и ok:true. Если значение не установлено, пишется пустой объект.

Поле ok пишется для помощи библиотекам, которым сложно проверить указано ли поле value.

Например, тип

```
memcache.query s:(Maybe string) v:(Maybe int) = memcache.Query;
```

Может быть сохранён, как

```
{
    "s":{
      "ok":true,
      "value":"hello"
     },
     "v":{}
}
```

При чтении все комбинации

`"ok"`	Указано ли явно `value`	Результат
`true`	да	MaybeTrue, value прочитано
`true`	нет	MaybeTrue, value установлено в пустое значение
`false`	да	ошибка
`false`	нет	MaybeFalse
Не указан явно	да	MaybeTrue, value прочитано
Не указан явно	нет	MaybeFalse

True

Чаще всего опускается, сохраняется только в некоторых контекстах, как пустой JSON Object, если поле нельзя опустить, например при возврате из функции.

Если же это поле зависит от маски полей, находящейся в самом объекте, то поле дублирует бит маски и сохраняется, как значение true если соответствующий бит выставлен. Это сделано для удобства интеграции с динамическими языками, где в объектах можно часто вообще не делать маски полей, а просто сделать несколько булевых переменных.

Такое поле можно указывать при чтении со значением true вместо задания значения маски (взаимодействие с битами маски аналогично полям остальных типов). Дополнительно, если такое поле указано со значением false, то проверяется, что соответствующий бит маски равен нулю, в противном случае происходит ошибка. Это сделано для того, чтобы если есть несколько полей, влияющих на значение маски различным образом, в них не было противоречий.

Например, тип

```
lists2.sublist fields_mask:#
  sort_by_date:fields_mask.1?true reverse:fields_mask.2?true
  = lists2.Sublist;
```

Может быть сохранён, как

```
{
  "fields_mask":4,
  "reverse":true
}
```

А при чтении достаточно указать

```
{
  "reverse":true
}
```

Чтобы значение маски полей было установлено в 4.

Следующий пример прочитается корректно, маска полей будет установлена в 0.

```
{
  "reverse":false
}
```

Но при чтении следующего примера произойдёт ошибка, так как значение false противоречит маски полей 4.

```
{
  "fields_mask":4,
  "reverse":false
}
```

Заключение

Сравнение с другими форматами

При сравнении TL с MessagePack и ProtoBuf отметим, что TL выигрывает по эффективности, схож по компактности и проигрывает по обратной совместимости.

В Protobuf и MessagePack небольшие значения целых могут занимать 1 или 2 байта против 4 в TL, однако при этом форматы сохраняют номера или имена полей и размеров объектов, которые не нужно сохранять в TL, так что по размеру сериализованных данных форматы оказываются примерно равны.

Эффективность чтения TL высочайшая, например чтение структуры **Point** из трёх полей int компилируется буквально в 3 инструкции процессора без ветвлений, если нет масок полей, и с несложным ветвлением при использовании маски. Так как нет разделителей, а формат встроенных типов совпадает с представлением в памяти обычного процессора, чтение идёт практически со скоростью чтения бинарных данных.

Для сравнения в Protobuf и MessagePack для чтения каждого целого нужны ветвления или даже мини цикл, а для чтения структуры из 3 целых нужно организовать цикл с ветвлением по номеру поля, так как поля могут идти в любом порядке и любые поля отсутствовать.

Однако эта дополнительная структура и сложность чтения, которой нет в TL, позволяет Protobuf и MessagePack пропускать любые неизвестные поля. MessagePack при пропуске неизвестного поля сложного типа будет парсить всю структуру, а Protobuf пропускает поле любой сложности целиком, так как формат объекта это строка с сериализованным представлением объекта.

Если же неизвестное поле было сохранено в TL, то при практически полном отсутствии структуры чтение поля, и таким образом всего объекта в TL будет невозможно.

Золотой серединой был бы, вероятно, гипотетический формат, который

1. Подобно MessagePack сохранял бы простые типы в компактный формат, не требующий цикла при чтении
2. Подобно Protobuf компактно сохранял тип каждого поля
3. Подобно Protobuf сохранял сложные объекты, как строки с сериализованными байтами этого обьекта, для возможности пропуска.
4. Подобно TL сохранял поля в строгом порядке (по возрастанию ID), чтобы при чтении объекта избегать циклов и многочисленных проверок

Среднесрочные планы по развитию TL

В основном мелочи, которые должны немного облегчить жизнь

1. Добавить явный маркер версии в начало файла, например version=1.1; Этот маркер позволит компилятору не делать предположений о том, какая версия языка используется.
2. Ограничить имена простых комбинаторов, чтобы правая часть была капитализированной версией левой, и оба имени были в одном namespace
3. Ограничить имена объединений, чтобы левая часть имени всегда содержала в качестве префикса правую, минус капитализация и потенциально нахождение в разных namespace (cd,responseEmpty = ab.Response);
4. Убрать секции типов и функций, а использовать для функций синтаксис a => B вместо a = B;
5. Убрать эвристики из кодогенератора, использовать явные аннотации для решения о том, какой код генерировать, например @map для генерации словаря вместо массива, @bool для генерации типа bool вместо enum.
6. Запретить более 1 анонимного поля в структуре, так как кодогенератор всё равно вынужден придумывать для них имена, и делает он это крайне плохо и по-разному.
7. Дискуссионно - поддержать открытые Union, например объединение (Bool | String | MyType)
8. Возможно сделать явный синтаксис для Union
9. Специфицировать и внедрить промежуточный формат TLO2, который будет позволять выразить весь язык TL, и ничего из того, чего нет в TL. Сейчас формат TLO не обладает обоими этими свойствами, что приводит к проблемам для инструментов, использующих формат TLO.