Apache Thrift - Interface Description Language (IDL)

Thrift interface description language

For Thrift version 0.15.0.

The Thrift interface definition language (IDL) allows for the definition of Thrift Types. A Thrift IDL file is processed by the Thrift code generator to produce code for the various target languages to support the defined structs and services in the IDL file.

Description

Here is a description of the Thrift IDL.

Document

Every Thrift document contains 0 or more headers followed by 0 or more definitions.

```
[1]  Document        ::=  Header* Definition*
```

Header

A header is either a Thrift include, a C++ include, or a namespace declaration.

```
[2]  Header          ::=  Include | CppInclude | Namespace
```

Thrift Include

An include makes all the symbols from another file visible (with a prefix) and adds corresponding include statements into the code generated for this Thrift document.

```
[3]  Include         ::=  'include' Literal
```

C++ Include

A C++ include adds a custom C++ include to the output of the C++ code generator for this Thrift document.

```
[4]  CppInclude      ::=  'cpp_include' Literal
```

Namespace

A namespace declares which namespaces/package/module/etc. the type definitions in this file will be declared in for the target languages. The namespace scope indicates which language the namespace applies to; a scope of ‘*’ indicates that the namespace applies to all target languages.

```
[5]  Namespace       ::=  ( 'namespace' ( NamespaceScope Identifier ) )

[6]  NamespaceScope  ::=  '*' | 'c_glib' | 'cpp' | 'delphi' | 'haxe' | 'go' | 'java' | 'js' |
'lua' | 'netstd' | 'perl' | 'php' | 'py' | 'py.twisted' | 'rb' | 'st' | 'xsd'
```

Definition

```
[7]  Definition      ::=  Const | Typedef | Enum | Senum | Struct | Union | Exception | Service
```

Const

```
[8]  Const           ::=  'const' FieldType Identifier '=' ConstValue ListSeparator?
```

Typedef

A typedef creates an alternate name for a type.

```
[9]  Typedef         ::=  'typedef' DefinitionType Identifier
```

Enum

An enum creates an enumerated type, with named values. If no constant value is supplied, the value is either 0 for the first element, or one greater than the preceding value for any subsequent element. Any constant value that is supplied must be non-negative.

```
[10] Enum            ::=  'enum' Identifier '{' (Identifier ('=' IntConstant)? ListSeparator?)*
'}'
```

Senum

Senum (and Slist) are now deprecated and should both be replaced with String.

```
[11] Senum           ::=  'senum' Identifier '{' (Literal ListSeparator?)* '}'
```

Struct

Structs are the fundamental compositional type in Thrift. The name of each field must be unique within the struct.

```
[12] Struct          ::=  'struct' Identifier 'xsd_all'? '{' Field* '}'
```

N.B.: The `xsd_all` keyword has some purpose internal to Facebook but serves no purpose in Thrift itself. Use of this feature is strongly discouraged

Union

Unions are similar to structs, except that they provide a means to transport exactly one field of a possible set of fields, just like union {} in C++. Consequently, union members are implicitly considered optional (see requiredness).

```
[13] Union          ::=  'union' Identifier 'xsd_all'? '{' Field* '}'
```

N.B.: The `xsd_all` keyword has some purpose internal to Facebook but serves no purpose in Thrift itself. Use of this feature is strongly discouraged

Exception

Exceptions are similar to structs except that they are intended to integrate with the native exception handling mechanisms in the target languages. The name of each field must be unique within the exception.

```
[14] Exception      ::=  'exception' Identifier '{' Field* '}'
```

Service

A service provides the interface for a set of functionality provided by a Thrift server. The interface is simply a list of functions. A service can extend another service, which simply means that it provides the functions of the extended service in addition to its own.

```
[15] Service        ::=  'service' Identifier ( 'extends' Identifier )? '{' Function* '}'
```

Field

```
[16] Field          ::=  FieldID? FieldReq? FieldType Identifier ('=' ConstValue)?
XsdFieldOptions ListSeparator?
```

Field ID

```
[17] FieldID        ::=  IntConstant ':'
```

Field Requiredness

There are two explicit requiredness values, and a third one that is applied implicitly if neither *required* nor *optional* are given: *default* requiredness.

```
[18] FieldReq       ::=  'required' | 'optional'
```

The general rules for requiredness are as follows:

required

- Write: Required fields are always written and are expected to be set.
- Read: Required fields are always read and are expected to be contained in the input stream.
- Defaults values: are always written

If a required field is missing during read, the expected behaviour is to indicate an unsuccessful read operation to the caller, e.g. by throwing an exception or returning an error.

Because of this behaviour, required fields drastically limit the options with regard to soft versioning. Because they must be present on read, the fields cannot be deprecated. If a required field would be removed (or changed to optional), the data are no longer compatible between versions.

optional

- Write: Optional fields are only written when they are set
- Read: Optional fields may, or may not be part of the input stream.
- Default values: are written when the isset flag is set

Most language implementations use the recommended practice of so-called “isset” flags to indicate whether a particular optional field is set or not. Only fields with this flag set are written, and conversely the flag is only set when a field value has been read from the input stream.

default requiredness (implicit)

- Write: In theory, the fields are always written. There are some exceptions to that rule, see below.
- Read: Like optional, the field may, or may not be part of the input stream.
- Default values: may not be written (see next section)

Default requiredness is a good starting point. The desired behaviour is a mix of optional and required, hence the internal name “opt-in, req-out”. Although in theory these fields are supposed to be written (“req-out”), in reality unset fields are not always written. This is especially the case, when the field contains a value, which by definition cannot be transported through thrift. The only way to achieve this is by not writing that field at all, and that's what most languages do.

Semantics of Default Values

There are ongoing discussions about that topic, see JIRA for details. Not all implementations treat default values in the very same way, but the current status quo is more or less that default fields are typically set at initialization time. Therefore, a value that equals the default may not be written, because the read end will set the value implicitly. On the other hand, an implementation is free to write the default value anyways, as there is no hard restriction that prevents this.

The major point to keep in mind here is the fact, that any unwritten default value implicitly becomes part of the interface version. If that default is changed, the interface changes. If, in contrast, the default value is written into the output data, the default in the IDL can change at any time without affecting serialized data.

XSD Options

N.B.: These have some internal purpose at Facebook but serve no current purpose in Thrift. The use of these options is strongly discouraged.

```
[19] XsdFieldOptions ::=  'xsd_optional'? 'xsd_nillable'? XsdAttrs?

[20] XsdAttrs        ::=  'xsd_attrs' '{' Field* '}'
```

Functions

```
[21] Function        ::=  'oneway'? FunctionType Identifier '(' Field* ')' Throws?
ListSeparator?

[22] FunctionType    ::=  FieldType | 'void'

[23] Throws          ::=  'throws' '(' Field* ')'
```

Types

```
[24] FieldType       ::=  Identifier | BaseType | ContainerType

[25] DefinitionType  ::=  BaseType | ContainerType

[26] BaseType        ::=  'bool' | 'byte' | 'i8' | 'i16' | 'i32' | 'i64' | 'double' | 'string'
| 'binary' | 'slist'

[27] ContainerType   ::=  MapType | SetType | ListType

[28] MapType         ::=  'map' CppType? '<' FieldType ',' FieldType '>'

[29] SetType         ::=  'set' CppType? '<' FieldType '>'

[30] ListType        ::=  'list' '<' FieldType '>' CppType?

[31] CppType         ::=  'cpp_type' Literal
```

Constant Values

```
[32] ConstValue      ::=  IntConstant | DoubleConstant | Literal | Identifier | ConstList |
ConstMap

[33] IntConstant     ::=  ('+' | '-')? Digit+

[34] DoubleConstant  ::=  ('+' | '-')? Digit* ('.' Digit+)? ( ('E' | 'e') IntConstant )?

[35] ConstList       ::=  '[' (ConstValue ListSeparator?)* ']'

[36] ConstMap        ::=  '{' (ConstValue ':' ConstValue ListSeparator?)* '}'
```

Basic Definitions

Literal

```
[37] Literal          ::=  ('"' [^"]* '"') | ("'" [^']* "'")
```

Identifier

```
[38] Identifier       ::=  ( Letter | '_' ) ( Letter | Digit | '.' | '_' )*

[39] STIdentifier     ::=  ( Letter | '_' ) ( Letter | Digit | '.' | '_' | '-' )*
```

List Separator

```
[40] ListSeparator    ::=  ',' | ';'
```

Letters and Digits

```
[41] Letter           ::=  ['A'-'Z'] | ['a'-'z']

[42] Digit            ::=  ['0'-'9']
```

Reserved keywords

```
"BEGIN", "END", "__CLASS__", "__DIR__", "__FILE__", "__FUNCTION__",
"__LINE__", "__METHOD__", "__NAMESPACE__", "abstract", "alias", "and", "args", "as",
"assert", "begin", "break", "case", "catch", "class", "clone", "continue", "declare",
"def", "default", "del", "delete", "do", "dynamic", "elif", "else", "elseif", "elsif",
"end", "enddeclare", "endfor", "endforeach", "endif", "endswitch", "endwhile", "ensure",
"except", "exec", "finally", "float", "for", "foreach", "from", "function", "global",
"goto", "if", "implements", "import", "in", "inline", "instanceof", "interface", "is",
"lambda", "module", "native", "new", "next", "nil", "not", "or", "package", "pass",
"public", "print", "private", "protected", "raise", "redo", "rescue", "retry", "register",
"return", "self", "sizeof", "static", "super", "switch", "synchronized", "then", "this",
"throw", "transient", "try", "undef", "unless", "unsigned", "until", "use", "var",
"virtual", "volatile", "when", "while", "with", "xor", "yield"
```

Examples

Here are some examples of Thrift definitions, using the Thrift IDL:

- ThriftTest.thrift used by the Thrift TestFramework
- Thrift tutorial