baidu

BIDL Specification

百度接口定义语言规范

liuxupeng
2012/10/17

目录

BIDL 规范

1 介绍

1.1 背景

百度接口定义语言（Baidu Interface Definition Language，以下简称为 BIDL），是一种规范性语言，采用类 C/C++语法来定义 BGCC 客户程序与服务器程序之间的通信接口。BIDL 提供了一套通用的基本数据类型，并以这些基本数据类型来定义更为复杂的数据类型，剥离了语言与平台的差异性，使得客户程序与服务程序均不感知接口背后的实现细节。特别地，分离了业务逻辑与通信逻辑，大大提升了客户程序与服务程序的运行可靠性，并降低了编写客户程序与服务程序的复杂性。

1.2 内容

本手册涉及的内容包括 BIDL 语言规范和 BIDL 文件示例。

2 术语

- BIDL: Baidu Interface Definition Language。百度接口定义语言。
- BGCC: Baidu General Communication Component。 百度通用通讯组件。
- bidl2sl: BIDL to Special Language。BIDL 语言解析器，将接口定义翻译成特定语言的源代码。

3 文件结构

BIDL 文件以“.bidl”为文件名后缀，包含接口定义、类型定义和常量定义。BIDL 语言的编译器 bidl2sl 负责解析 BIDL 文件，将在 BIDL 中定义的接口、类型和常量翻译为指定语言的源代码。此源代码被应用于采用指定语言编定的客户端和服务器端的通信过程中。每个 BIDL 文件均由可选的“头”和“体”两部分构成。严格的 BIDL 文件结构可参见 BIDL BNF。

3.1 文件头

“头”包含 0 个或多个 include 语句。Include 语句用来包含并引用其他 BIDL 文件的内容。

3.2 文件体

“体”包含 0 个或多个常量定义语句、类型定义语句或接口定义语句。

常量定义语句用于定义常量，可定义的常量类型有布尔型(boolean)，8 位(int8)、16 位(int16)、32 位(int32)和 64 位(int64)有符号整数类型，浮点型(float)，字符串(string)，二进制(binary)，以及容器类型。容器类型包括序列(sequence)，集合(set)，映射(map)。

类型定义语句可定义结构体(struct)和枚举(enum)。

接口定义语句用于定义具体的通信接口(class)。每个通信接口包含 1 个或多个服务，BIDL 对服务的名称，传入参数、传出参数及返回值都做了严格的定义。

4 BIDL 保留字

注：保留字不允许在 BIDL 文件中出现。

__CLASS__	__DIR__	__FILE__	__LINE__	__METHOD__	__NAMESPACE__	abstract
alias	alignas	alignof	and	and_eq	args	as
asm	assert	auto	BEGIN	bitand	bitor	bool
break	byte	case	catch	char	char16_t	char32_t
clone	compl	const_cast	constexpr	continue	cpp	cpp_type
declare	decltype	def	default	del	delete	do
double	dynamic	dynamic_cast	elif	else	elseif	elsif
END	enddeclare	endfor	endforeach	endif	endswitch	endwhile
ensure	except	exec	explicit	export	extends	extern
final	finally	for	foreach	friend	function	global
goto	if	implements	import	inline	instanceof	int
interface	is	java	lambda	long	module	mutable
native	new	next	nil	noexcept	not	not_eq
null	nullptr	operator	optional	or	or_eq	package
pass	print	private	protected	protocol	public	raise
redo	register	reinterpret_cast	request	request_len	required	rescue
retry	return	self	short	signed	sizeof	static

static_assert	static_cast	strictfp	super	switch	synchronized	template
then	this	thread_local	throw	throws	transient	try
typeid	typename	undef	union	unless	unsigned	until
use	using	var	virtual	volatile	wchar_t	when
while	with	xor	xor_eq	yield		

5 BIDL 关键字

bidl 关键字的使用详见“6 语法组成”小节。

include	namespace	void	boolean	int8	int16	int32
int64	float	string	binary	map	sequence	set
typedef	struct	enum	const	in	out	all
true	TRUE	false	FALSE	class		

6 语法组成

6.1 注释

BIDL 文件支持两种注释格式：一是单行注释；一是多行注释。

```
/* 多行注释 */
//单选注释
```

6.2 include

BIDL 文件支持嵌套包含。在 BIDL 文件中可以通过使用 include 语句来包含其他 BIDL 文件，以引用已定义的类型来定义新的类型。如在 address.bidl 中定义有 Address 结构体。如下：

```
struct Address {
    string City;
    string Street;
}
```

在 person.bidl 中定义 Person 结构体时，可通过使用 include 语句来包含 address.bidl，然后再引用 Address 结构体。如下：

```
include "address.bidl"
struct Person {
    string name;
    Address address;
}
```

6.3 namespace

C++和 Java 编译语言均有名称空间的概念。为了能将 BIDL 文件经 bidl2sl 编译后生成的源码放置于指定的名称空间中，BIDL 文件引入了 namespace 关键字。namespace 关键字的另一个作用是在 BIDL 文件内进行类型引用时，对引用类型进行分类管理。如下：

```
namespace ims {
        struct Request {
                int32 seq_no;
        }
}
namespace acd {
        class Api {
                void send_request(ims.Request req);
        }
}
```

6.4 typedef

在 BIDL 文件中，可以使用 typedef 语句来为已有类型取别名，以简化使用。在使用复杂类型时，此语句非常有效。如下：

```
typedef int8 age_t;
typedef map<int32, string> Books;
struct Library {
        Books books;
}
```

6.5 void

void 是 BIDL 基本数据类型之一。通常被用来指定某个服务没有返回值。如下：

```
class Api {
        void ping();
}
```

6.6 boolean

boolean 是 BIDL 基本数据类型之一，表示布尔值类型。通常被用于 const 语句，typedef 语句及服务定义语句中。如下：

```
typedef boolean b_t;
const boolean used = true;
class Api {
        boolean use_boolean([in] boolean i_value, [out] boolean o_value, [all] boolean a_value);
}
```

6.7 int8

int8 是 BIDL 基本数据类型之一，用于表示 8 位整数类型。通常被用于 const 语句，typedef 语句及服务定义语句中。如下：

```
typedef int8 i8_t;
const i8_t i8 = 8;
class Api {
        int8 use_int8([in] int8 i_value, [out] int8 o_value, [all] int8 a_value);
}
```

6.8 int16

int16 是 BIDL 基本数据类型之一，用于表示 16 位整数类型。通常被用于 const 语句，typedef 语句及服务定义语句中。如下：

```
typedef int16 i16_t;
const i16_t i16 = 16;
class Api {
        int16 use_int16([in] int16 i_value, [out] int16 o_value, [all] int16 a_value);
}
```

6.9 int32

int32 是 BIDL 基本数据类型之一，用于表示 32 位整数类型。通常被用于 const 语句，typedef 语句及服务定义语句中。如下：

```
typedef int32 i32_t;
const i32_t i32 = 32;
class Api {
        int32 use_int32([in] int32 i_value, [out] int32 o_value, [all] int32 a_value);
}
```

6.10 int64

int64 是 BIDL 基本数据类型之一，用于表示 64 位整数类型。通常被用于 const 语句，typedef 语句及服务定义语句中。如下：

```
typedef int64 i64_t;
const i64_t i64 = 64;
class Api {
        int64 use_int64([in] int64 i_value, [out] int64 o_value, [all] int64 a_value);
}
```

6.11 float

float 是 BIDL 基本数据类型之一，用于表示浮点数类型。通常被用于 const 语句，typedef 语句和服务定义语句中。如下：

```
typedef float f_t;
const f_t f = 3.3;
class Api {
        float use_float([in] float i_value, [out] float o_value, [all] float a_value);
}
```

6.12 string

string 是 BIDL 基本数据类型之一，用于表示字符串类型。通常被用于 const 语句，typedef 语句和服务定义语句中。如下：

```
typedef string str_t;
const str_t name = "jack";
class Api {
        string use_string([in] string i_value, [out] string o_value, [all] string a_value);
}
```

6.13 binary

binary 是 BIDL 基本数据类型之一，用于表示二进制类型。通常被用于 const 语句，typedef 语句和服务定义语句中。如下：

```
typedef binary bin_t;
const bin_t data = "1234";
class Api {
        binary use_binary([in] binary i_value, [out] binary o_value, [all] binary a_value);
}
```

6.14 enum

enum 在 BIDL 中用于枚举值定义，构造复杂类型。通常被用于结构体定义和服务定义语句中。如下：

```
enum Color {
        RED,
        GREEN,
        BLUE
}
struct Person {
        string name;
        Color skin_color;
}
```

```
class Api {
        Color use_color([in] Color i_value, [out] Color o_value, [all] Color a_value);
}
```

说明：enum 的用法与 C 语法类似。在默认情况下，枚举值从 0 开始且依次加 1。枚举值也可以手动指定。如下：

```
enum Color {
        RED = -255,
        GREEN = 0,
        BLUE = 255
}
```

6.15 struct

struct 在 BIDL 中用于结构体定义，构造复杂类型。通常被用于结构体定义和服务定义语句中。如下：

```
struct Address {
        string city;
        string street;
}
struct Person {
        string name;
        Address address;
}
class Api {
        Person use_person([in] Person i_value, [out] Person o_value, [all] Person a_value);
}
```

6.16 sequence

sequence 是 BIDL 容器类型之一，用于定义值序列。通常被用于 const 定义语句，struct 定义语句以及服务定义语句。如下:

```
const sequence<string> names = ["jack", "mery"];
typedef sequence<int32> numbers_t;
struct Person {
        string name;
        numbers_t numbers;
}
class Api {
        sequence<string> use_sequence([in] sequence<string> i_value, [out] sequence<string> o_value, [all]
        sequence<string> a_value);
}
```

6.17 set

set 是 BIDL 容器类型之一，用于定义值序列。通常被用于 const 定义语句，struct 定义语句以及服务定义语句。如下:

```
const set<int32> numbers = <1, 2, 3, 4>;
typedef set<string> names_t;
struct Person {
        string name;
        names_t names;
}
class Api {
        set<string> use_names([in] set<string> i_value, [out] set<string> o_value, [all] set<string> a_value);
}
```

6.18 map

map 是 BIDL 容器类型之一，用于定义值序列。通常被用于 const 定义语句，struct 定义语句以及服务定义语句。如下:

```
const map<int32, string> n2s = {1: "first", 2: "second"};
typedef map<string, float> course2score_t;
struct Person {
        string name;
        course2score_t course2score;
}
class Api {
        map<string, float> use_map([in] map<string, float> i_value, [out] map<string, float>
        o_value, [all] map<string, float> a_value);
}
```

6.19 in

in 是 BIDL 服务参数传递修饰符之一，表示参数是传入参数。在未指定传递修饰符的情况下参数默认为传入参数。用法如下：

```
class Api {
        void send_name_and_age(string name, [in] int32 age);
}
```

6.20 out

out 是 BIDL 服务参数传递修饰符之一，表示参数是传出参数。用法如下：

```
class Api {
        void get_name([out] string name);
```

```
}
```

6.21 all

all 是 BIDL 服务参数传递修饰符之一，表示参数是传入传出参数。用法如下：

```
class Api {
        void round([all] int32 number);
}
```

6.22 const

const 关键字在 BIDL 中用于定义常量，包括基本类型常量和容器类型常量。

基本类型常量定义如下所示：

```
const boolean used = true;
const int8 i8 = 8;
const int16 i16 = 16;
const int32 i32 = 32;
const int64 i64 = 64;
const float flt = 3.3;
const string name = "jack";
const binary data = "123";
```

容器类型常量如下所示：

```
const sequence<int32> numbers = [1, 2, 3, 4, 5];
const set<string> names = <"jack", "mery">;
const map<string, float> course2score = {"Math": 90.3, "English": 88.3};
```

注意：sequence 使用方括号；set 使用尖括号;map 使用花括号，且键与值以冒号分隔。对于 sequence,set,map 而言，各元素之间以逗号分隔。

6.23 class

class 在 BIDL 文件中提供了服务器与客户端进行功能调用的一种约定。class 包含 1 个或多个服务。服务可有返回值，返回值类型为基本数据类型或复杂数据类型或容器数据类型。当服务无返回值时返回类型记作 void。服务可有 0 个或多个参数。参数类型为基本数据类型或复杂数据类型或容器数据类型。如下所示：

```
struct Person {
        string name;
        int32 age;
}
```

```
class Api {
        void ping();
        int32 add(int32 a, int32 b);
        int32 sub(int32 a, int32 b);
        void shuffle([all] sequence<int32> numbers);
        int32 send_person(Person person);
}
```

7 示例

本节给出了两个 BIDL 文件示例，以供参考。BIDL 文件的编译方法请参见《bidl2sl 工具手册》。

address.bidl

```
namespace math {
 struct Address {
        string city;
        string road;
        int32 number;
 }
}
```

math.bidl

```
include "address.bidl"

namespace math {

 typedef boolean b_t;

 const sequence<boolean> ss1 = [true, false, false];
 const sequence<int32> ss2 = [1, 2, 3];
 const sequence<string> ss3 = ["first", "second"];
 const sequence<float> ss4 = [1.1, 2.2, 3.3];

 const sequence<set<int32>> tt1 = [<1>, <2>, <3>];

 const set<boolean> sk1 = <true, false, false>;
 const set<int32> sk2 = <1, 2, 3>;
 const set<string> sk3 = <"first", "second">;
 const set<float> sk4 = <1.1, 2.2, 3.3>;

 const set<sequence<string>> tt2 = <["first"], ["second", "third"]>;
```

```
const map<int32, string> sm1 = {1: "first"};

const map<sequence<int32>, set<string>> tt3 =
        {
                [1]: <"first">,
                [2]: <"second">
        };


const b_t be_used = false;
const boolean be_health = true;

typedef int8 i8_t;
const i8_t    i8    = 8;

typedef int16 i16_t;
const i16_t i16 = 16;

typedef int32 i32_t;
const i32_t i32 = 32;

typedef int64 i64_t;
const i64_t i64 = 64;

typedef string s_t;
const s_t str = "hello";

typedef binary bin_t;
const bin_t bin = "data";

typedef float f_t;
const f_t f = 3.3;

struct Person {
        string name;
        int32 age;
}

typedef Person MyPerson;

enum Color {
        RED,
        GREEN,
        BLUE
```

```
}

typedef Color MyColor;

enum Gender {
        MALE,
        FEMALE
}

typedef sequence<string> Children;

struct Citizen {
        string name;
        int32 age;
        Gender gender;
        Address address;
        Children children;
        set<i32_t> xxx;
        map<int8, boolean> kkk;
}

        class Math {
        Citizen test_citizen(Citizen ins, [out] Citizen outs, [all] Citizen alls);
        void ping();
                void add(int32 a, int32 b, [out] int32 sum);

        boolean get_boolean();
        int8 get_int8();
        int16 get_int16();
        int32 get_number();
        int64 get_int64();
        float get_float();
        string get_string();
        binary get_binary();
        Color get_color();

        sequence<int32> get_sequence();
        set<string> get_string_set();
        map<int64, string> get_int_str_map();
        sequence<set<boolean>> get_bool_set_sequence();
        Person get_person();
        sequence<Person> get_persons();
        set<Person> get_persons2();
        map<Person, Person> get_persons3();
```

```
sequence<set<Person>> get_persons4();

void send_bool(b_t value);
void send_int8(i8_t value);
void send_int16(i16_t value);
        void send_int32(i32_t value);
void send_int64(i64_t value);
void send_float(f_t value);
void send_string(s_t value);
void send_binary(bin_t value);
void send_color(Color color);
void send_person(MyPerson person);
void send_int3_seqence(sequence<int32> numbers);
void send_persons(sequence<Person> persons);
void send_persons2(set<Person> persons);
void send_int32_set_seq(sequence<set<int32>> xx);
void send_persons3(map<Person, Person> persons);
void send_person_vec_set(set<sequence<Person>> xxx);

void out_boolean([out] boolean value);
void out_int8([out] int8 value);
void out_int16([out] int16 value);
void out_int32([out] int32 value);
void out_int64([out] int64 value);
void out_string([out] string value);
void out_binary([out] binary value);
void out_float([out] float value);
void out_seq([out] sequence<int32> xxx);
void out_string_set([out] set<string> xxx);
void out_int16_float_map([out] map<int16, float> xxx);
void out_int_vec_set([out] set<sequence<int32>> xxx);

void all_boolean([all] boolean value);
void all_int8([all] int8 value);
void all_int16([all] int16 value);
void all_int32([all] int32 value);
void all_int64([all] int64 value);
void all_string([all] string value);
void all_binary([all] binary value);
void all_float([all] float value);
void all_int32_seq([all] sequence<int32> value);
void all_person([all] Person value);
void all_int32_set([all] set<int32> value);
void all_person_seq([all] sequence<Person> value);
```

```
        void all_int32_person_map([all] map<int32, Person> value);


        void color_out_all([all] MyColor color1, [out] MyColor color2);

        sequence<MyColor> color_xx(
            [in]    set<MyColor> color1,
            [out] sequence<MyColor> color2,
            [all] map<MyColor, MyColor> color3);
    }
}
```

8 BNF

本节采用 BNF 描述 BIDL 语法范式。

```
/* for flex */
TOKEN_IN ::= 'in'
TOKEN_ALL ::= 'all'
TOKEN_OUT ::= 'out'
TOKEN_NAMESPACE ::= 'namespace'
TOKEN_INCLUDE ::= 'include'
TOKEN_CONST ::= 'const'
TOKEN_IDENTIFIER ::= [a-zA-Z_][\.a-zA-Z_0-9]*
TOKEN_EQ ::= '='
TOKEN_SEMICOLON ::= ';'
TOKEN_BOOLEAN ::= 'boolean'
TOKEN_INT8 ::= 'int8'
TOKEN_INT16 ::= 'int16'
TOKEN_INT32 ::= 'int32'
TOKEN_INT64 ::= 'int64'
TOKEN_FLOAT ::= 'float'
TOKEN_DOUBLE ::= 'double'
TOKEN_STRING ::= 'string'
TOKEN_BINARY ::= 'binary'
TOKEN_TYPEDEF ::= 'typedef'
TOKEN_MAP ::= 'map'
TOKEN_SET ::= 'set'
TOKEN_SEQUENCE ::= 'sequence'
TOKEN_ENUM ::= 'enum'
TOKEN_STRUCT ::= 'struct'
TOKEN_CLASS ::= 'class'
TOKEN_VOID ::= 'void'
```

```
TOKEN_LESS ::= '<'
TOKEN_MORE ::= '>'
TOKEN_COMMA ::= ','
TOKEN_LEFT_CURLY_BRACKET ::= '{'
TOKEN_RIGHT_CURLY_BRACKET ::= '}'
TOKEN_LEFT_SQUARE_BRACKET ::= '['
TOKEN_RIGHT_SQUARE_BRACKET ::= ']'
TOKEN_LEFT_BRACKET ::= '('
TOKEN_RIGHT_BRACKET ::= ')'

TOKEN_INTEGER ::= [+|-]?[0-9]+
TOKEN_DECIMAL ::= [+|-]?[0-9]+(\.[0-9]+)?((E|e)[+|-]?[0-9]+)?

TOKEN_BOOLVAL ::=
        'true'
        | 'TRUE'
        | 'false'
        | 'FALSE'

TOKEN_LITERAL ::= (""[^"]*"")|("'"[^']*"'")

/* for bison */
document ::=
        include_clause_list definition_clause_list

include_clause_list ::=
        /* 空 */
        | include_clause_list include_clause

include_clause ::=
        TOKEN_INCLUDE TOKEN_LITERAL

definition_clause_list ::=
        /* 空 */
        | definition_clause_list definition_clause

definition_clause ::=
        const_clause
        | typedef_clause
        | enum_clause
        | struct_clause
        | class_clause
        | namespace_clause
```

```
const_clause ::=
        TOKEN_CONST const_type TOKEN_IDENTIFIER TOKEN_EQ const_value TOKEN_SEMICOLON

const_type ::=
        TOKEN_IDENTIFIER
        | basic_type


basic_type ::=
        TOKEN_BOOLEAN
        | TOKEN_INT8
        | TOKEN_INT16
        | TOKEN_INT32
        | TOKEN_INT64
        | TOKEN_FLOAT
        | TOKEN_DOUBLE
        | TOKEN_STRING
        | TOKEN_BINARY

const_value ::=
        TOKEN_BOOLVAL
        | TOKEN_INTEGER
        | TOKEN_DECIMAL
        | TOKEN_LITERAL

typedef_clause ::=
        TOKEN_TYPEDEF data_type TOKEN_IDENTIFIER TOKEN_SEMICOLON

data_type ::=
        TOKEN_IDENTIFIER
        | basic_type
        | container_type

container_type ::=
        map_type
        | set_type
        | sequence_type

map_type ::=
        TOKEN_MAP TOKEN_LESS data_type TOKEN_COMMA data_type TOKEN_MORE

set_type ::=
        TOKEN_SET TOKEN_LESS data_type TOKEN_MORE
```

```
sequence_type ::=
        TOKEN_SEQUENCE TOKEN_LESS data_type TOKEN_MORE

enum_clause ::=
        TOKEN_ENUM TOKEN_IDENTIFIER TOKEN_LEFT_CURLY_BRACKET enum_field_list
TOKEN_RIGHT_CURLY_BRACKET

enum_field_list ::=
        TOKEN_IDENTIFIER
        | TOKEN_IDENTIFIER TOKEN_EQ TOKEN_INTEGER
        | enum_field_list TOKEN_COMMA TOKEN_IDENTIFIER
        | enum_field_list TOKEN_COMMA TOKEN_IDENTIFIER TOKEN_EQ TOKEN_INTEGER

struct_clause ::=
        TOKEN_STRUCT TOKEN_IDENTIFIER TOKEN_LEFT_CURLY_BRACKET struct_field_list
TOKEN_RIGHT_CURLY_BRACKET

struct_field_list ::=
        struct_filed
        | struct_field_list struct_filed

struct_filed ::=
        data_type TOKEN_IDENTIFIER TOKEN_SEMICOLON

class_clause ::=
        TOKEN_CLASS TOKEN_IDENTIFIER TOKEN_LEFT_CURLY_BRACKET function_list
TOKEN_RIGHT_CURLY_BRACKET

function_list ::=
        function
        | function_list function

function ::=
        function_type TOKEN_IDENTIFIER TOKEN_LEFT_BRACKET function_field_list TOKEN_RIGHT_BRACKET
TOKEN_SEMICOLON

function_type ::=
        data_type
        | TOKEN_VOID

function_field_list ::=
        /*空*/
        | function_field
        | function_field_list TOKEN_COMMA function_field
```

```
function_field ::=
        data_type TOKEN_IDENTIFIER
        | function_field_direction data_type TOKEN_IDENTIFIER

function_field_direction ::=
        TOKEN_LEFT_SQUARE_BRACKET TOKEN_IN TOKEN_RIGHT_SQUARE_BRACKET
        | TOKEN_LEFT_SQUARE_BRACKET TOKEN_OUT TOKEN_RIGHT_SQUARE_BRACKET
        | TOKEN_LEFT_SQUARE_BRACKET TOKEN_ALL TOKEN_RIGHT_SQUARE_BRACKET

namespace_clause ::=
        TOKEN_NAMESPACE TOKEN_IDENTIFIER TOKEN_LEFT_CURLY_BRACKET definition_clause_list
TOKEN_RIGHT_CURLY_BRACKET
```