# C++

# 1. Compilation of contract（C/C++ version）

C/C++'s contract compilation tool chain is under following library:

https://github.com/bottos-project/contract-tool-cpp.git

This tool is used for compiling C/C++ contracts, generating wasm/wast files, and scanning and generating the abi file. We will describe it in following 2 samples from the library.

## 1. Sample of testHelloWorld

Put the contract code file put into a directory, and put that directory into the directory of ""“contract-tool-cpp", like sample of testHelloWorld.

As you need to compile the contract, go through into the directory of contract, like come into testHelloWorld's directory, then use the follwing command to compile the contract, while after then the 'testHelloWorld.wast' and 'testHelloWorld.wasm' will be generated under the same directory:

```
python ../gentool.py wasm testHelloWorld.cpp
```

Since this sample does not related to abi, so we will describe what does the abi file be generated like in following cases.

## 2. Sample of testRegUser

Like above case, compile the contract as coming into the directory of 'testRegUser' and compiling the contract via using following command:

```
python ../gentool.py wasm testRegUser.cpp
```

Now we mainly focus on how does the contract abi file be generated. By now, let us introduce what is abi file composed of:

- structs ：The struct descriptions from scanning, which will be used later.
- actions：The description of contract methods, in which the action_name stands for the name of the function, type stands for the parameters that called by contract；
- tables ：The contract persistence data drovider description, where table_name is the table's name, index_type is the type of index, key_names and Key_types are the name and type of the key value, and type is the structure definition of the content.

The ABI file is generated by scanning the hpp file, in which it tells the scanner specific definition through comments.

- "//@abi action reguser" :

  A method reguser is defined, and the corresponding entry parameter is defined as UserInfo;

- "//@abi table userinfo:[index_type:string， "key_names:userName， key_types:string] " :

  A table is defined, and the structure of the table content is defined as Userbaseinfo.

Under the Testreguser folder, you can scan the ABI file for the hpp file by using the following command:

```
python ../gentool testRegUser.hpp
```

# 2. contract's editing ( C/C++ version )

## 2.1. A simplest contract

The entry function of the contract is the start function, and we create a simple contract, that is, when the contract is tuned, print "Hello World in Start":

```
#include "contractcomm.hpp"

int start(char* method)
{
    myprints("hello world in start");

    return 0;
}
```

The "Contractcomm.hpp" is a common basic interface declaration file. Call this contract, and the "Hello World in Start" will be printed in the log of the node:

```
2018-08-06 16:05:20 [INF] vm/wasm/exec/env_func.go:390 prints(): VM: func prints: hello
world in start
```

## 2.2. Gets the method that invokes the contract

As calling a contract, we need to specify a specific method for calling the contract, that is, the following "method" parameter "Test_method":

```
{"version":1,
"cursor_num":28,"cursor_label":3745260307,"lifetime":15270819998,"sender":"example",
"contract":"example", "method":"test_method", "param":"", "sig_alg":1, "signature":""}
```

This method is passed into the contract through the entry of the start function, that is, the following method parameter, then we add the statement to print the parameter in the contract:

```
#include "contractcomm.hpp"

int start(char* method)
{
    myprints("hello world in start");
    myprints(method);

    return 0;
}
```

We construct a transaction that sets method to a specific string, the following "Test_method": (The exact value of the signature is omitted here, the same below)

```
{"version":1,
"cursor_num":28,"cursor_label":3745260307,"lifetime":15270819998,"sender":"example",
"contract":"example", "method":"test_method", "param":"", "sig_alg":1, "signature":""}
```

From calling this contract, there will be the following log output: Print the set "method" parameter, that is, "Test_method":

```
2018-08-07 14:40:22 [INF] vm/wasm/exec/env_func.go:412 prints(): VM: func prints: hello
word in start

2018-08-07 14:40:22 [INF] vm/wasm/exec/env_func.go:412 prints(): VM: func prints:
test_method
```

We can provide different contract implementations in the contract according to method, that is, by comparing the method parameters and then going to different branch branches, such as the "add" and "Del" methods provided in the following contracts:

```
#include "contractcomm.hpp"
#include "string.hpp"

int start(char* method)
{
    myprints("hello word in start");
    myprints(method );

    if (0 == strcmp("add", method))
    {
        myprints("it is method add");
    }
    else if (0 == strcmp("del", method))
    {
        myprints("it is method del");
    }
```

```
    return 0;
}
```

## 2.3. Gets the call contract parameter

When calling a contract, in addition to specifying the method of calling the contract, you also need to bring the corresponding parameters, that is, the following "param" parameter:

```
{"version":1,
"cursor_num":28,"cursor_label":3745260307,"lifetime":15270819998,"sender":"example",
"contract":"example", "method":"test_method", "param":"", "sig_alg":1, "signature":""}
```

This parameter can be obtained through the "Getparam" interface in the contract, for example:

```
#include "contractcomm.hpp"
#define ERROR_PARAM (-1)

int start(char* method)
{
    char param[PARAM_MAX_LEN];
    uint32_t paramLen = 0;

    paramLen = getParam(param, PARAM_MAX_LEN);

    if (0 == paramLen)
    {
        myprints("paramLen is 0, error");
        return ERROR_PARAM;
    }

    myprints("paramLen is:");
    printi(paramLen);
    myprints("param detail:");
    for(int i = 0;i<paramLen;i++)
    {
        printi(param[i]);
    }

    return 0;
}
```

We construct a transaction: The parameter is a value that is serialized by Messagepack, for example, we need to pass a serialized parameter to [97,98,99], convert its serialized value into a 16-binary string: "616263", and then fill in the Param:

```
{"version":1,
"cursor_num":28,"cursor_label":3745260307,"lifetime":15270819998,"sender":"example",
"contract":"example", "method":"test_method", "param":"616263", "sig_alg":1,
"signature":""}
```

By calling this contract, there will be the following log output, that is, you can get the value after serialization to [97,98,99] in the contract:

```
2018-08-08 17:25:26 [INF] vm/wasm/exec/env_func.go:431 prints(): VM: func prints:
paramLen is:

2018-08-08 17:25:26 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 3

2018-08-08 17:25:26 [INF] vm/wasm/exec/env_func.go:431 prints(): VM: func prints: param
detail:

2018-08-08 17:25:26 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 97

2018-08-08 17:25:26 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 98

2018-08-08 17:25:26 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 99
```

## 2.4. Storage and reading of contract data

In order to facilitate unified access to data, we require that the data to be saved by the contract be serialized with Messagepack, and that the read data be deserialized accordingly to obtain the original data.

Here we define a structure for testing:

```
struct TestStruct {
    uint32_t valueA;
    uint32_t valueB;
};
```

We pass param to a parameter of type teststruct, where the values of the fields are filled in 1 and 2 respectively, first serialized, obtained [220,0,2,206,0,0,0,1,206,0,0,0,2], and converted into a 16-binary string: " Dc0002ce00000001ce00000002 ", we fill in the Param in the transaction with this value:

```
{"version":1,
"cursor_num":28,"cursor_label":3745260307,"lifetime":15270819998,"sender":"example",
"contract":"example", "method":"test_method", "param":"dc0002ce00000001ce00000002",
"sig_alg":1, "signature":""}
```

We are in the contract to reverse the incoming parameters:

```
#include "contractcomm.hpp"
#define ERROR_PARAM (-1)
#define ERROR_UNPACK (-2)
```

```
struct TestStruct {
    uint32_t valueA;
    uint32_t valueB;
};


static bool unpack_struct(MsgPackCtx *ctx, TestStruct *info)
{
    uint32_t size = 0;

    if (!unpack_array(ctx, &size)) return false;
    if (2 != size) return false;

    if (!unpack_u32(ctx, &info->valueA)) return false;
    if (!unpack_u32(ctx, &info->valueB)) return false;

    return true;
}

static bool pack_struct(MsgPackCtx *ctx, TestStruct *info)
{
    if (!pack_array16(ctx, 2)) return false;

    if (!pack_u32(ctx, info->valueA)) return false;
    if (!pack_u32(ctx, info->valueB)) return false;

    return true;
}

int start(char* method)
{
    char param[PARAM_MAX_LEN];
    uint32_t paramLen = 0;

    paramLen = getParam(param, PARAM_MAX_LEN);

    if (0 == paramLen)
    {
        myprints("paramLen is 0, error");
        return ERROR_PARAM;
    }

    MsgPackCtx ctx;
    msgpack_init(&ctx, (char*)param, paramLen);

    TestStruct testStruct;
    bool suc = unpack_struct(&ctx, &testStruct);
    if (!suc)
    {
        myprints("unpack struct error");
        return ERROR_UNPACK;
    }
```

```
    myprints("data from input param:");
    printi(testStruct.valueA);
    printi(testStruct.valueB);

    return 0;
}
```

By calling this contract, we can see that the structural body inside correctly gets the parameters set when called, 1 and 2, respectively:

```
2018-08-08 18:20:00 [INF] vm/wasm/exec/env_func.go:431 prints(): VM: func prints: data
from input param:

2018-08-08 18:20:00 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 1

2018-08-08 18:20:00 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 2
```

We rewrite this data and save it through the Setbinvalue interface, then read it out through Getbinvalue, and then deserialize to get the real data:

```
#include "contractcomm.hpp"
#define ERROR_PARAM (-1)
#define ERROR_UNPACK (-2)
#define ERROR_SAVE_DB (-3)
#define ERROR_READ_DB (-4)
#define ERROR_CONTRACT_NAME (-5)

struct TestStruct {
    uint32_t valueA;
    uint32_t valueB;
};

static bool unpack_struct(MsgPackCtx *ctx, TestStruct *info)
{
    uint32_t size = 0;

    if (!unpack_array(ctx, &size)) return false;
    if (2 != size) return false;

    if (!unpack_u32(ctx, &info->valueA)) return false;
    if (!unpack_u32(ctx, &info->valueB)) return false;

    return true;
}

static bool pack_struct(MsgPackCtx *ctx, TestStruct *info)
{
    if (!pack_array16(ctx, 2)) return false;

    if (!pack_u32(ctx, info->valueA)) return false;
```

```
    if (!pack_u32(ctx, info->valueB)) return false;

    return true;
}

int start(char* method)
{
    char param[PARAM_MAX_LEN];
    uint32_t paramLen = 0;

    paramLen = getParam(param, PARAM_MAX_LEN);
    if (0 == paramLen)
    {
        myprints("paramLen is 0, error");
        return ERROR_PARAM;
    }

    /* Reverse the param of the order */
    MsgPackCtx ctx;
    msgpack_init(&ctx, (char*)param, paramLen);

    TestStruct testStruct;
    bool suc = unpack_struct(&ctx, &testStruct);
    if (!suc)
    {
        myprints("unpack struct error");

        return ERROR_UNPACK;
    }

    myprints("data from input param:");
    printi(testStruct.valueA);
    printi(testStruct.valueB);

    /* Rewrite fields to 3, 4 */
    testStruct.valueA = 3;
    testStruct.valueB = 4;

    /* Serialization of operations */
    msgpack_init(&ctx, (char*)param, PARAM_MAX_LEN);
    suc = pack_struct(&ctx, &testStruct);
    if (!suc)
    {
        myprints("pack struct error");
        return ERROR_UNPACK;
    }

    char tableName[] = "testTableName";
    char keyName[] = "testKeyName";

    /* Saves the calling interface */
    uint32_t handleResult = setBinValue(tableName, strlen(tableName), keyName,
strlen(keyName), ctx.buf, ctx.pos);
```

```
    if (0 == handleResult)
    {
        myprints("save to db error");
        return ERROR_SAVE_DB;
    }

    char contractname[STR_ARRAY_LEN(USER_NAME_MAX_LEN)];
    uint32_t contractNameLen = getCtxName(contractname,
STR_ARRAY_LEN(USER_NAME_MAX_LEN));
    if (0 == contractNameLen)
    {
        myprints("contract name error");
        return ERROR_CONTRACT_NAME;
    }

    /* The calling interface gets the value that was just saved */
    handleResult = getBinValue(contractname, contractNameLen, tableName,
strlen(tableName), keyName, strlen(keyName), param, PARAM_MAX_LEN);
    if (0 == handleResult)
    {
        myprints("read from db error");
        return ERROR_READ_DB;
    }

    /* Reverse serialization of operations */
    msgpack_init(&ctx, (char*)param, handleResult);
    suc = unpack_struct(&ctx, &testStruct);
    if (!suc)
    {
        myprints("unpack struct error");
        return ERROR_UNPACK;
    }

    myprints("data from db:");
    printi(testStruct.valueA);
    printi(testStruct.valueB);

    return 0;
}
```

Call this contract again with the following trades (that is, input structures, fields 1, 2, respectively):

```
{"version":1,
"cursor_num":28,"cursor_label":3745260307,"lifetime":15270819998,"sender":"example",
"contract":"example", "method":"test_method", "param":"dc0002ce00000001ce00000002",
"sig_alg":1, "signature":""}
```

As you can see from the log, the fields obtained to the input are 1, 2, which are overwritten to 3, 4, and then saved, and the expected values can still be obtained after the last read is deserialized, that is, the fields are 3 and 4, respectively:

```
2018-08-08 18:38:34 [INF] vm/wasm/exec/env_func.go:431 prints(): VM: func prints: data
from input param:

2018-08-08 18:38:34 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 1

2018-08-08 18:38:34 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 2

......

2018-08-08 18:38:34 [INF] vm/wasm/exec/env_func.go:431 prints(): VM: func prints: data
from db:

2018-08-08 18:38:34 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 3

2018-08-08 18:38:34 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:example, method:test_method, func printi: 4
```

## 2.5. Invoking other contracts

The following is a compute contract that currently implements the Add method, which calculates the and of two parameters:

```
#include "contractcomm.hpp"
#include "string.hpp"

#define ERROR_PARAM (-1)
#define ERROR_UNPACK (-2)
#define ERROR_INVALID_METHOD (-3)

struct TestStruct {
    uint32_t valueA;
    uint32_t valueB;
};

static bool unpack_struct(MsgPackCtx *ctx, TestStruct *info)
{
    uint32_t size = 0;

    if (!unpack_array(ctx, &size)) return false;
    if (2 != size) return false;

    if (!unpack_u32(ctx, &info->valueA)) return false;
    if (!unpack_u32(ctx, &info->valueB)) return false;

    return true;
}
```

```
static bool pack_struct(MsgPackCtx *ctx, TestStruct *info)
{
    if (!pack_array16(ctx, 2)) return false;

    if (!pack_u32(ctx, info->valueA)) return false;
    if (!pack_u32(ctx, info->valueB)) return false;

    return true;
}

int start(char* method)
{
    if (0 == strcmp("add", method))
    {
        char param[PARAM_MAX_LEN];
        uint32_t paramLen = 0;

        /* Get the parameters from input */
        paramLen = getParam(param, PARAM_MAX_LEN);

        if (0 == paramLen)
        {
            myprints("paramLen is 0, error");
            return ERROR_PARAM;
        }

        /* Deserialize parameters to get raw data */
        MsgPackCtx ctx;
        msgpack_init(&ctx, (char*)param, paramLen);
        TestStruct testStruct;
        bool suc = unpack_struct(&ctx, &testStruct);
        if (!suc)
        {
            myprints("unpack struct error");

            return ERROR_UNPACK;
        }

        printi(testStruct.valueA);
        printi(testStruct.valueB);

        /* Print the and of two parameters */
        myprints("add result is:");
        printi(testStruct.valueA + testStruct.valueB);
    }
    else
    {
        myprints("invalid method");

        return ERROR_INVALID_METHOD;
    }
```

```
    return 0;
}
```

Then we write a contract in which the above calculation contract is called, which is passed to the above calculation contract two parameters, in the calculation contract to calculate the and of these two parameters: (assuming that the above calculation contract has been deployed on the Calccontract account)

```
#include "contractcomm.hpp"
#define ERROR_UNPACK  (-1)
#define ERROR_CALLTRX (-2)

struct TestStruct {
    uint32_t valueA;
    uint32_t valueB;
};

static bool unpack_struct(MsgPackCtx *ctx, TestStruct *info)
{
    uint32_t size = 0;

    if (!unpack_array(ctx, &size)) return false;
    if (2 != size) return false;

    if (!unpack_u32(ctx, &info->valueA)) return false;
    if (!unpack_u32(ctx, &info->valueB)) return false;

    return true;
}

static bool pack_struct(MsgPackCtx *ctx, TestStruct *info)
{
    if (!pack_array16(ctx, 2)) return false;

    if (!pack_u32(ctx, info->valueA)) return false;
    if (!pack_u32(ctx, info->valueB)) return false;

    return true;
}

int start(char* method)
{
    char param[PARAM_MAX_LEN];
    TestStruct testStruct;

    testStruct.valueA = 3;
    testStruct.valueB = 4;

    /* Serialization of operations */
    MsgPackCtx ctx;
    msgpack_init(&ctx, (char*)param, PARAM_MAX_LEN);
    bool suc = pack_struct(&ctx, &testStruct);
    if (!suc)
    {
```

```
        myprints("pack struct error");
        return ERROR_UNPACK;
    }

    /* Define the name of the contract to call, and the method name */
    char *callContractName = "calccontract";
    char *callMethod = "add";

    /* Invoke the contract */
    uint32_t callResult = callTrx(callContractName , strlen(callContractName),
callMethod, strlen(callMethod), ctx.buf , ctx.pos);
    if (0 != callResult)
    {
        myprints("call trx error");
        return ERROR_CALLTRX;
    }

    myprints("call trx succed");

    return 0;
}
```

Let's call this contract above and there will be the following log:

```
2018-08-09 12:00:34 [INF] vm/wasm/exec/env_func.go:431 prints(): VM: func prints: call
trx succed

2018-08-09 12:00:34 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:calccontract, method:add, func printi: 3

2018-08-09 12:00:34 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:calccontract, method:add, func printi: 4

2018-08-09 12:00:34 [INF] vm/wasm/exec/env_func.go:431 prints(): VM: func prints: add
result is:

2018-08-09 12:00:34 [INF] vm/wasm/exec/env_func.go:405 printi(): VM: from
contract:calccontract, method:add, func printi: 7
```

## 2.6. Attachment 1: interface function for shared invocation

- void prints(char *str, uint32_t len);

  Function: Print string

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| str | char* | Character Pointer to print |
| len | uint32_t | The length of the string to print |

- void printi(uint64_t param);

| Parameter | Type | Description |
| --- | --- | --- |
| param | uint64_t | The integer to print |

- uint32_t setBinValue(char* object, uint32_t objLen, char* key, uint32_t keyLen, char *value, uint32_t valLen);

  Function: Save data to the chain

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| object | char* | The table name corresponding to the saved dat |
| objLen | uint32_t | The length of the table name corresponding to the saved data |
| key | char* | Key value name of the saved data |
| keyLen | uint32_t | The key value name length of the saved data |
| value | char* | Data to be saved |
| valLen | uint32_t | Length of data to be saved |

  Return value: Saved data length

- uint32_t getBinValue(char* contract, uint32_t contractLen, char* object, uint32_t objLen, char* key, uint32_t keyLen, char *valueBuf, uint32_t valueBufLen);

  Function: Get data from the chain

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| contract | char* | The contract name corresponding to the obtained data |
| contractLen | uint32_t | The length of the contract name corresponding to the obtained data |
| object | char* | Gets the table name corresponding to the data |
| objLen | uint32_t | Gets the length of the table name corresponding to the data |
| key | char* | Gets the key value name of the data |
| keyLen | uint32_t | Gets the length of the key value name of the data |
| valueBuf | char* | Buffers used to store data |
| valueBufLen | uint32_t | The length of the buffer used to store the data |

Return value: The length of the obtained data

- uint32_t removeBinValue(char* object, uint32_t objLen, char* key, uint32_t keyLen);

  Function: Remove data from the chain

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| object | char* | The table name corresponding to the deleted data |
| objLen | uint32_t | The length of the table name corresponding to the deleted dat |
| key | char* | The key value name of the deleted data |
| keyLen | uint32_t | The length of the key value name of the deleted data |

  Return value: The length of the deleted data

- uint32_t getParam(char *param, uint32_t bufLen);

  Function: Gets the parameter when the contract is invoked

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| param | char* | Buffer for storing parameter |
| bufLen | uint32_t | Buffer for storing parameter |

  Return value: The length of the parameter obtained

- bool callTrx(char *contract , uint32_t contractLen, char *method , uint32_t methodLen, char *buf , uint32_t bufLen );

  Function: Invoke other contracts (asynchronous mode)

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| contract | char* | The name of the contract to invoke |
| contractLen | uint32_t | The length of the name of the contract to invoke |
| method | char* | The method name to invoke the contract |
| methodLen | uint32_t | The length of the method name to invoke the contract |
| buf | char* | Call the contract incoming parameter |
| bufLen | uint32_t | The length of the parameter passed by the calling contract |

Return value：true: success，false: failure

- uint32_t getCtxName(char *str , uint32_t len);

  Function: Get contract name

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| str | char* | Buffer for storing contract name |
| len | uint32_t | The length of the buffer that stores the contract name |

  Return value: The length of the account name obtained

- uint32_t getSender(char *str , uint32_t len);

  Function: Gets the name of the account that invokes the current contract

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| str | char* | Buffer for storing account names |
| len | uint32_t | The length of the buffer that stores the account name |

  Return value: The length of the account name obtained

- bool isAccountExist(char *name， uint32_t nameLen);

  function: Check if the account exists

  Parameter description：

| Parameter | Type | Description |
| --- | --- | --- |
| name | char* | The account you want to check |
| nameLen | uint32_t | The length of the account name to check |

  Return value：true: exists，false: not exists

# 2.7. Serialization of the attached 2:messagepack

## 2.7.1 Overview

In order to facilitate the parameter transmission when the contract is invoked, and to read the persistent data of the contract, we have selected Messagepack this lightweight codec method, detailed specification reference:

https://msgpack.org/

https://github.com/msgpack/msgpack/blob/master/spec.md

In addition, we have made some tailoring to the characteristics of the contract data.

1. Basic Type：Supports uint8、uint16、uint32、uint64、array16、bin16、str16Type；
2. Variable length data: For example strtype,messagepack the original specification according to the length of the string fill in the length of bytes, there are 1, 2, 4 byte length of the difference, after the transformation of the default use of 2 bytes (str16), Bintype and ArrayType is also, that is, only support Bin16, ARRAY16 this type;
3. Structure: The structure body is encapsulated in the form of array, the ARRAY16 head is written first, and then the fields are encoded in turn.

Sample(C)：

```
struct user_login {
    char user_name[USER_NAME_MAX_LEN];
    uint32_t random_num;
};

user_login login;
strcpy(login.user_name, "testuser");
login.random_num = 99;

pack_array16(&ctx, 2);
pack_str16(&ctx, login->user_name, strlen(login->user_name));
pack_u32(&ctx, login->random_num);
```

Encoding result：

```
0xdc, 0x00, 0x02, 0xda, 0x00, 0x08, 0x74, 0x65, 0x73, 0x74, 0x75, 0x73, 0x65, 0x72,
0xce, 0x00, 0x00, 0x00, 0x63
```

## 2.7.2 Coding specification

**Types**

- **Integer** represents an integer

- Raw
  - **String** extending Raw type represents a UTF-8 string
  - **Binary** extending Raw type represents a byte array
- **Array** represents a sequence of objects

#### Format

| format name | first byte (in binary) | first byte (in hex) |
|---|---|---|
| bin16 | 11000101 | 0xc5 |
| uint8 | 11001100 | 0xcc |
| uint16 | 11001101 | 0xcd |
| uint32 | 11001110 | 0xce |
| uint64 | 11001111 | 0xcf |
| str16 | 11011010 | 0xda |
| array16 | 11011100 | 0xdc |

#### Notation in diagrams

```
one byte:
+--------+
|        |
+--------+

a variable number of bytes:
+========+
|        |
+========+

variable number of objects stored in MessagePack format:
+~~~~~~~~~~~~~~~~~+
|                 |
+~~~~~~~~~~~~~~~~~+
```

#### int format family

Int format family stores an integer in 2, 3, 5, or 9 bytes.

```
uint 32 stores a 32-bit big-endian unsigned integer
+--------+--------+--------+--------+--------+
|  0xce  |ZZZZZZZZ|ZZZZZZZZ|ZZZZZZZZ|ZZZZZZZZ|
+--------+--------+--------+--------+--------+

uint 64 stores a 64-bit big-endian unsigned integer
+--------+--------+--------+--------+--------+--------+--------+--------+--------+
|  0xcf  |ZZZZZZZZ|ZZZZZZZZ|ZZZZZZZZ|ZZZZZZZZ|ZZZZZZZZ|ZZZZZZZZ|ZZZZZZZZ|ZZZZZZZZ|
+--------+--------+--------+--------+--------+--------+--------+--------+--------+
```

**str format family**

Str format family stores a byte array in 3 bytes of extra bytes in addition to the size of the byte array.

```
str 16 stores a byte array whose length is upto (2^16)-1 bytes:
+--------+--------+--------+========+
|  0xda  |ZZZZZZZZ|ZZZZZZZZ|  data  |
+--------+--------+--------+========+

where
* ZZZZZZZZ_ZZZZZZZZ is a 16-bit big-endian unsigned integer which represents N
* N is the length of data
```

**bin format family**

Bin format family stores an byte array in 3 bytes of extra bytes in addition to the size of the byte array.

```
bin 16 stores a byte array whose length is upto (2^16)-1 bytes:
+--------+--------+--------+========+
|  0xc5  |YYYYYYYY|YYYYYYYY|  data  |
+--------+--------+--------+========+

where
* YYYYYYYY_YYYYYYYY is a 16-bit big-endian unsigned integer which represents N
* N is the length of data
```

**array format family**

Array format family stores a sequence of elements in 3 bytes of extra bytes in addition to the elements.

```
array 16 stores an array whose length is upto (2^16)-1 elements:
+--------+--------+--------+~~~~~~~~~~~~~~~~~+
|  0xdc  |YYYYYYYY|YYYYYYYY|    N objects    |
+--------+--------+--------+~~~~~~~~~~~~~~~~~+

where
* YYYYYYYY_YYYYYYYY is a 16-bit big-endian unsigned integer which represents N
    N is the size of a array
```

**Serialization: type to format conversion**

MessagePack serializers convert MessagePack types into formats as following:

| source types | output format |
| --- | --- |
| Integer | int format family (positive fixint uint 8/16/32/64) |
| String | str format family (str16) |
| Binary | bin format family (bin16) |
| Array | array format family (array16) |

**Deserialization: format to type conversion**

MessagePack deserializers convert MessagePack formats into types as following:

| source formats | output type |
| --- | --- |
| positive fixint,uint 8/16/32/64 | Integer |
| str16 | String |
| bin16 | Binary |
| array16 | Array |