基于Liquid的定点数实现方案2.0

一、设计目标

本方案的主要目标是提供Liquid智能合约定点数的使用，实现对包含定点数定义的Liquid合约编译和使用。

用户能通过console，实现带定点数参数智能合约的deploy、call、sendTransaction等功能，同时Java SDK端通过与编译端的编解码方案共识，实现通过ABI对wasm字节码的定点数解析。

二、设计背景

对于智能合约的定点数支持能够完善整个合约的功能，满足用户的使用需求，给予良好的用户体验。

本方案实施的背景是基于Fisco-bcos 3.0、Java SDK 3.1.0和console 3.1.0实现对带有定点数使用的Liquid智能合约相关功能实现，使用户能够使用定点数参数，实现基本的运算功能等。首先我们对目前已有的一些定点数相关实现进行调研汇总。

2.1 Solidity定点数

https://docs.soliditylang.org/en/latest/types.html?highlight=fixed#fixed-point-numbers

Fixed Point Numbers

> **Warning**
>
> **Fixed** point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from.

Soldiity 0.8.0还没有完全支持定长浮点型，可以**声明**定长浮点型的变量，但不能给它们**赋值**或把它们赋值给其他变量（**不能实际使用**）。

`fixed/ufixed`表示各种大小的有符号和无符号的定长浮点型，关键字`ufixedMxN`和`fixedMxN`，`M`表示这个定点数类型所占的bit位总长度，`N`表示可表示的小数位数（例如`ufixed128x18`表示该类型用128个bit位表示，小数位数支持到18位）。

2.1.1 编码方式：

对于`fixedMxN=X`，类型总位数M最多为256位，`8 < M <=256, M % 8 ==0, 0 < N <=80`：

编码函数`enc(X) = enc(X*10**N)`,其中`X*10**N`被视为`int256`类型，同理`ufixedMxN`。

在Solidity编码方案中，所有的数据类型都占32个字节长度，所以存在很大的空间浪费。

例如：

对`于uint32`类型的数据69，Solidity编码方式如下：

`0x0000000000000000000000000000000000000000000000000000000000000045`

2.1.2 编码实例

对于一个`baz(uint32,bool)`方法，假设传入的参数为`69`和`true`

`0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000000000000000000000000000000000001`

占用的字节总长为68，其中方法签名4字节，两个数据类型分别占用32字节，可以看出存在大量未使用的字节空间。

基于该Solidity定点数的设计方案，提出两种待讨论Liquid智能合约的定点数设计方案。

目前基于Solidity的定点数支持开源项目：

1.prb-math

支持18位小数和59/60位整数（十进制）

2.abdkmath

2^{-N}表示有精度损失

2.2 Scale编码

SCALE编码：

> https://docs.substrate.io/v3/advanced/scale-codec/

SCALE现在不支持对定点数的支持

2.3 Rust crate

> https://docs.rs/fixed/1.10.0/fixed/struct.FixedU64.html

This crate does *not* provide decimal fixed-point numbers. For example 0.001 cannot be represented exactly, as it is $1/10^3$. It is binary fractions like $1/2^4$ (0.0625) that can be represented exactly, provided there are enough fractional bits.

二进制精确，对于可以由2^{-N}累积求和得到的数字而言，存在无损表示；而其他数字则可能存在精度损失

2.4 其他调研结果

针对目前主流的平台和DSL语言进行定点数相关实现的调研结果如下：

智能合约语言	平台	是否对定点数支持	相关方案
Ink!	Parity	否	
Solidity	Ethereum	支持声明不支持赋值	fixedMxN = v; v = v/10**N; 转化为int256编码
Vyper	Ethereum	同Solidity	同Solidity
Move	Libra	否	
无	EOS	否	
无	Fabric	否	
无	Solana	支持浮点数	

Solana官方文档相关补充：

> Recent results show the float operations take more instructions compared to integers equivalents. Fixed point implementations may vary but will also be less than the float equivalents:

总结：

在目前的主流平台中，针对类似Liquid智能合约语言的其他语言，只有Solidity和Vyper实现了定点数，但由于其针对ether特性，小数点后实现18位的精度对于Liquid意义并不大。

所以针对Solidity和Vyper对定点数表示的具体实现细节，基于Liquid采取Scale编码方式的特点，我们提出了以下方案。

三、方案设计

3.1 方案

定点数需要实现编解码和简单的加减乘除运算。根据计算机组成原理对于定点数的支持，实现方式为2^{-N}（N=1..)的累积求和结果，由于小数部分位数限制，采用此种方式对于某些常见数字也会存在精度损失。

如采用`16bits`表示小数时，小数部分最大值为`0.99998`，且在范围`[0.999978, 0.999992]`均会被处理为`0.99998`

```
#[allow(unused_imports)]
use fixed::{types::extra::U16, FixedU128};

fn main() {
        let _test1 = FixedU128::<U16> ::checked_from_num(0.999992).unwrap();
        let _test2=FixedU128::<U16>::checked_from_num(0.999978).unwrap();
        assert_eq!(_test1.to_string(), "0.99998");
        assert_eq!(_test2.to_string(), "0.99998")
}
```

为实现无损精度损失，现在设计如下方案

	存储	计算	编解码	特点
方案	struct { signal: bool, wb_int: bytes#M, wb_frac: bytes#N}	转化为u64/u128/u256计算，计算结果在转化为定点数	大端法编码 按照1位符号，m位整数位，n位小数位进行编码	优点：存储效率高，精度表示高，意义明确 缺点：liquid中存储空间浪费

```
方案具体描述：
方案可实现多种表示，形式类似于Fixed<M>x<N> 其中M表示浮点数的总位数，N表示转化为10进制时小数的位数
如Fixed128x6，表示总共使用128bits表示小数，转化为10进制时含有6位小数
struct {
  signal: bool,
  wb_int: u128,
  wb_frac: u32
}
计算：转化为u128计算
编码：大端法编码，按照1位符号位，107位整数位，20位小数位进行编码
```

3.2 Liquid编解码方案

liquid实现对合约中FixedPoint类型编解码的支持

3.2.1 Encoder

1.声明Vec作为编码结果的容器

2.通过位操作将FixedPoint转化为u256

3.对u256转化为u8数组(大端序)

4.传入容器成为Vec

简单实现

```
impl scale::Encode for FixedPoint128x6 {
    fn encode(&self) -> Vec<u8> {
        let mut buf:Vec<u8> = Vec::with_capacity(16 as usize);
        let mut ans: u128 = self.wb_frac.into();
        ans |= self.wb_int << 20;
        if self.signal == true {
            ans |= 0 << 127;
        } else {
            ans |= 1 << 127;
        }
        buf.extend(ans.to_be_bytes());
        buf
```

```
    }
}
```

3.2.2 Decoder

1.读取成为u256

2.通过位运算解码FixedPoint各部分

3.2 SDK编解码方案

Java SDK端主要对ABI和Wasm文件进行解析实现字节码与Java定点数对象之间的转化，从而实现基于Java SDK的console提供用户传输参数与数据交互等功能，Liquid编解码同SDK。

3.2.1 Encoder

功能：通过ABI参数类型的判断，解析出具体的定点数对应的类型，并按照类型定义编码为对应字节码

对于示例：`fixed128x6("1.1")`（总长度`128`bits，做多表示`6`位（0.000001）小数）

1.`new byte[16]`作为整个`fixed128x6`类型的字节长度

2.判断符号位，从而确定`bit[0]`的值，`1.2`符号位为0

3.整数部分`1`转化为`byteArray[]`，小数部分`2`转化为`byteArray[]`

4.顺序拼接符号位、整数位、小数位：`byte[16]`:`[0 0 0 0 0 0 0 0 0 0 0 0 0 0 16 0 1]`

针对上例子，如果传入的值位`-1.2`，则结果为`byte[16]`：`[1 0 0 0 0 0 0 0 0 0 0 0 0 0 16 0 1]`

3.2.2 Decoder

功能：读入定点数参数对应的字节码，并根据类型特点获取实际的参数值，创建定点数对象

对于`fixed128x6("1.1")`

1.根据类型名读取类型总长`M=128`和小数位数`N=6`

2.按顺序`readByteArray`分别读符号位（0）、整数部分、小数部分，对于`byte[length-3]`需要通过移位操作处理该字节存储的整数部分和小数部分（因为小数部分用20bits表示）

3.将整数和小数数值算出相加`add`创建`fixed128x6`类型实例，`type.getConstructor(BigDecimal.class).newInstance(result)`

四、数据结构

4.1 Rust数据结构：

```
可实现的方案：
Fixed64x4, Fixed128x6, Fixed256x10;
形式可写为Fixed<M>x<N> 其中M表示定点数的总位数，N表示转化为10进制时小数的位数

具体实现举例：
```

```
Fixed128x6，表示总共使用128bits表示小数，转化为10进制时含有6位小数
struct Fixed128x6{
  signal: bool,
  wb_int: u128,
  wb_frac: u32
}
计算：转化为u128计算，先将小数和整数部分分别转化为字符串，在利用字符串进行拼接后转化为u128计算
编码：大端法编码，按照1位符号位，107位整数位，20位小数位进行编码
模块声明：fixed_point_simple
模块导出：pub use fixed_point_simple::Fixed128x6
liquid内合法化：impl_basic_trait! {FixedPointU64F16
}
ABI中命名：Primitive_type_to_string!(Fixed128x6 => Fixed128x6);
```

举例：

```
数值： 1.1
Fixed128x6 {
  signal: 0,
  wb_int: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
  wb_frac: 00 00 00 01
}
编码：
SCALE字节
00000000000000000000000000100001
[00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 10, 00, 01]

数值： 12345678.876549
Fixed128x6 {
  signal: 0,
  wb_int: 123,
  wb_frac: 267,
}
编码：
Scale字节：0000000000000000000000007b0010b
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 176, 1, 11]

```

4.2 SDK数据结构

```
public class Fixed128x6 extends FixedPointNumType {
    public static final Fixed64x16 Default = new Fixed128x6(BigDecimal.ZERO);
    public static final String TYPE_NAME = "fixed";

    public Fixed128x6(BigDecimal value) {
        this(128, 20, value);
    }

```

```
    public Fixed128x6(String value) {
        this(new BigDecimal(value));
    }

    protected Fixed128x6(int mBitSize, int nBitSize, BigDecimal value) {
        super(TYPE_NAME, mBitSize, nBitSize, value);
    }

    protected Fixed128x6(int mBitSize, int nBitSize, String value) {
        super(TYPE_NAME, mBitSize, nBitSize, value);
    }
}
```

```
public class FixedPointNumType extends FixedType {
    static final int DEFAULT_BIT_LENGTH = MAX_BIT_LENGTH >> 1;

    public FixedPointNumType(String typePrefix, int mBitSize, int nBitSize,
BigDecimal value) {
        super(typePrefix + mBitSize + "x" + nBitSize, value, mBitSize, nBitSize);
        if (!valid(mBitSize, nBitSize, value)) {
            throw new UnsupportedOperationException(
                    "Bitsize must be 8 bit aligned, and in range 0 < bitSize <=
256");
        }
    }

    public FixedPointNumType(String typePrefix, int mBitSize, int nBitSize, String
value) {
        super(typePrefix + mBitSize + "x" + nBitSize, value, mBitSize, nBitSize);
    }

    boolean valid(int mBitSize, int nBitSize, BigDecimal value) {
        return isValidBitSize(mBitSize, nBitSize);
    }

    static boolean isValidBitSize(int mBitSize, int nBitSize) {
        return mBitSize % 8 == 0
                && nBitSize % 8 == 0
                && mBitSize * nBitSize > 0
                && mBitSize <= MAX_BIT_LENGTH;
    }
}
```