Liquid

发布 *1.0.0-rc3*

2023 年 03 月 06 日

快速开始

1 关键特性 **3**

2 合作共建 **5**

2.1 环境配置 5
2.2 Hello World! 9
2.3 合约模块 13
2.4 状态变量与容器 14
2.5 合约方法 25
2.6 事件 32
2.7 类型 34
2.8 环境与内置方法 41
2.9 外部合约调用 42
2.10 开发指南 47
2.11 编译选项 53
2.12 单元测试专用 API 53
2.13 外部合约 Mock 56
2.14 基于宏的元编程 59
2.15 Liquid 架构设计 61
2.16 FISCO BCOS 环境接口规范 62
2.17 FISCO BCOS Wasm 合约接口规范 68
2.18 微众银行区块链开源生态 70

不断多样化、复杂化的应用场景为智能合约编程语言带来了全新挑战：分布式、不可篡改的执行环境要求智能合约具备更强的隐私安全性与鲁棒性；日渐扩大的服务规模要求智能合约能够更加高效运行；智能合约开发过程需要对开发者更加友好；对于跨链协同等不断涌现的新型计算范式，也需要能够提供原生抽象。在上述背景下，微众银行区块链团队提出了 **SPEC** 设计规范，即智能合约编程语言应当涵盖安全（Security）、性能（Performance）、体验（Experience）及可定制（Customization）四大要旨。

微众银行区块链团队结合对智能合约技术的理解与掌握，选择以 Rust 语言为载体对 **SPEC** 设计规范进行工程化实现，即 Liquid 项目。Liquid 对 **SPEC** 设计规范中的技术要旨提供了全方位支持，能够用来编写运行于区块链底层平台 FISCO BCOS 的智能合约。

CHAPTER 1

关键特性

CHAPTER 2

合作共建

微众银行区块链团队秉承多方参与、资源共享、友好协作和价值整合的理念，将 Liquid 项目完全向公众开源，并专设有智能合约编译技术专项兴趣小组（CTSC-SIG），欢迎广大企业及技术爱好者踊跃参与 Liquid 项目共建。

- GitHub 主页
- Gitee 主页
- 公众号
- CTSC-SIG

2.1 环境配置

注意

受限于网络情况及机器性能，本小节中部分依赖项的安装过程可能较为耗时，请耐心等待。必要时可能需要配置网络代理。

推荐切换 Rustup 官方镜像源为国内镜像源，请参考 Rustup 镜像安装帮助 更换镜像源。

2.1.1 部署 Rust 编译环境

Liquid 智能合约的构建过程主要依赖 Rust 语言编译器 `rustc` 及代码组织管理工具 `cargo`，且均要求版本号大于或等与 1.50.0。如果此前从未安装过 `rustc` 及 `cargo`，可参考下列步骤进行安装：

- 对于 Mac 或 Linux 用户，请在终端中执行以下命令；

```
# 此命令将会自动安装 rustup，rustup 会自动安装 rustc 及 cargo
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

- 对于 32 位 Windows 用户，请从此处下载安装 32 位版本安装程序。
- 对于 64 位 Windows 用户，请从此处下载安装 64 位版本安装程序。

如果此前安装过 `rustc` 及 `cargo`，但是未能最低版本要求，则可在终端中执行以下命令进行更新：

```
rustup update
```

安装完毕后，分别执行以下命令验证已安装正确版本的 `rustc` 及 `cargo`：

```
rustc --version
cargo --version
```

此外需要安装以下工具链组件：

```
rustup toolchain install nightly-2023-01-03 --component rust-src rustc-dev llvm-tools-
↪preview
rustup default nightly-2023-01-03
rustup target add wasm32-unknown-unknown
```

注意

由于 Liquid 使用了少量目前尚不稳定的 Rust 语言特性，因此在构建时需要依赖 `nightly` 版本的 `rustc` 。但是这些特性目前已经被广泛应用在 Rust 项目中，因此其可靠性值得信赖。随着 Rust 语言迭代演进，这些特性终将变为稳定特性。

注意

所有可执行程序都会被安装于 `$HOME/cargo/bin` 目录下，包括 `rustc` 、`cargo` 及 `rustup` 等。为方便使用，需要将 `$HOME/cargo/bin` 目录加入到操作系统的 `PATH` 环境变量中。在安装过程中，`rustup` 会尝试自动配置 `PATH` 环境变量，但是由于权限等原因，该过程可能会失败。当发现 `rustc` 或 `cargo` 无法正常执行时，可能需要手动配置该环境变量。

注意

如果当前网络无法访问 Rustup 官方镜像，请参考 Rustup 镜像安装帮助 更换镜像源。

构建 Liquid 智能合约的过程中需要下载大量第三方依赖，若当前网络无法正常访问 crates.io 官方镜像源，则按照以下步骤为 `cargo` 更换镜像源：

```
# 编辑 cargo 配置文件，若没有则新建
vim $HOME/.cargo/config
```

并在配置文件中添加以下内容：

```
[source.crates-io]
registry = "https://github.com/rust-lang/crates.io-index"
replace-with = 'ustc'
[source.ustc]
registry = "git://mirrors.ustc.edu.cn/crates.io-index"
```

2.1.2 安装其他依赖

请确保配置 `cmake` 环境，Linux 可以通过以下命令安装：

```
# Ubuntu 请执行下面的命令
sudo apt install cmake build-essential
# CentOS 请执行下面的命令
sudo yum install cmake3
```

Mac 下可以直接通过 `homebrew` 安装：

```
brew install cmake
```

如果 CentOS 的 yum 资源无 cmake3（如 CentOS 7），则需要手动下载 cmake3 进行配置

以下载 `cmake 3.21.3` 版本为例，到cmake 官网下载包后，解压到`/data/home/software` 目录，得到了 `cmake-3.21.3-linux-x86_64` 目录，并修改`/etc/profile` 以设置 cmake 环境变量

```
vi /etc/profile
export CMAKE3_HOME=/data/home/software/cmake-3.21.3-linux-x86_64
export PATH=$CMAKE3_HOME/bin:$PATH

# 环境变量生效
source /etc/profile
```

2.1.3 安装 cargo-liquid

cargo-liquid 是用于辅助开发 Liquid 智能合约的命令行工具，在终端中执行以下命令安装：

```
cargo install --git https://github.com/WeBankBlockchain/cargo-liquid --branch main --
↪locked --force
```

注意

若无法正常访问 GitHub，则请执行 `cargo install --git https://gitee.com/WeBankBlockchain/cargo-liquid --branch main --locked --force` 命令进行安装。

如果使用的是 CentOS 系统，安装前，确保按此Issue 检查依赖

- 确保 cmake 版本大于 3.12
- 安装 devtoolset 并启用
- 安装 rust 工具链

```
# 请确保 cmake 版本大于 3.12
# 请参考下述命令使用 gcc7
sudo yum install -y epel-release centos-release-scl
sudo yum install -y devtoolset-7
source /opt/rh/devtoolset-7/enable
# 请参考下述命令使用要求版本的 rust 工具链
rustup toolchain install nightly-2023-01-03 --component rust-src rustc-dev llvm-tools-
↪preview
rustup default nightly-2023-01-03
```

确保上述工具版本符合要求后，再次重新执行 cargo install 命令安装

开始安装后，以 gitee 为例，会输出类似的日志：

```
Updating git repository `https://gitee.com/WeBankBlockchain/cargo-liquid`
  Installing cargo-liquid v1.0.0-rc3 (https://gitee.com/WeBankBlockchain/cargo-liquid?
↪tag=v1.0.0-rc3#5da4da65)
Updating `git://mirrors.ustc.edu.cn/crates.io-index` index
  Fetch [=======>                 ]  34.20%, 5.92MiB/s
```

如果下载 crates 包失败，可重新执行 cargo install 命令重试下载

安装成功后，会输出如下日志：

```
Compiling wabt v0.10.0
    Finished release [optimized] target(s) in 1m 33s
  Installing /data/home/webase/.cargo/bin/cargo-liquid
  Installing /data/home/webase/.cargo/bin/liquid-analy
   Installed package `cargo-liquid v1.0.0-rc3 (https://gitee.com/WeBankBlockchain/cargo-
↪liquid?tag=v1.0.0-rc3#5da4da65)` (executables `cargo-liquid`, `liquid-analy`)
```

2.1.4 推荐安装 Binaryen（可选）

推荐安装 Binaryen 以优化编译过程

Binaryen 项目中包含了一系列 Wasm 字节码分析及优化工具，其中如 `wasm-opt` 等工具会在 Liquid 智能合约的构建过程中使用。请参考其官方文档。

除根据官方文档的编译安装方式外

- Ubuntu 下可通过 `sudo apt install binaryen` 下载安装（如使用 Ubuntu，则系统版本不低于 20.04
- Mac 下可直接通过 `brew install binaryen` 下载安装 binaryen
- 其他操作系统可参照此处查看是否可直接通过包管理工具安装）
  - 如 CentOS 则参考

```
# 下载 binaryen 的 rpm 包
wget https://download-ib01.fedoraproject.org/pub/epel/7/x86_64/Packages/b/binaryen-
↪104-1.el7.x86_64.rpm
# 安装 rpm 包
sudo rpm -ivh binaryen-104-1.el7.x86_64.rpm
```

2.2 Hello World!

提示： 为了能够更好地使用 Liquid 进行智能合约开发，我们强烈建议提前参考 Rust 语言官方教程 ，掌握 Rust 语言的基础知识，尤其借用、生命周期、属性等关键概念。

本节将以简单的 HelloWorld 合约为示例，帮助读者快速建立对 Liquid 合约的直观认识。

```rust
#![cfg_attr(not(feature = "std"), no_std)]

use liquid::storage;
use liquid_lang as liquid;
```

（下页继续）

(续上页)

```
#[liquid::contract]
mod hello_world {
    use super::*;

    #[liquid(storage)]
    struct HelloWorld {
        name: storage::Value<String>,
    }

    #[liquid(methods)]
    impl HelloWorld {
        pub fn new(&mut self) {
            self.name.initialize(String::from("Alice"));
        }

        pub fn get(&self) -> String {
            self.name.clone()
        }

        pub fn set(&mut self, name: String) {
            self.name.set(name)
        }
    }

    #[cfg(test)]
    mod tests {
        use super::*;

        #[test]
        fn get_works() {
            let contract = HelloWorld::new();
            assert_eq!(contract.get(), "Alice");
        }

        #[test]
        fn set_works() {
            let mut contract = HelloWorld::new();

            let new_name = String::from("Bob");
```

(下页继续)

(续上页)

```
45            contract.set(new_name.clone());
46            assert_eq!(contract.get(), "Bob");
47        }
48    }
49 }
```

上述智能合约代码中所使用的各种语法的详细说明可参阅“**普通合约**”一章，在本节中我们先进行初步的认识：

- 第 1 行：

```
1 #![cfg_attr(not(feature = "std"), no_std)]
```

  cfg_attr 是 Rust 语言中的内置属性之一。此行代码用于向编译器告知，若编译时没有启用 std 特性，则在全局范围内启用 no_std 属性，所有 Liquid 智能合约项目都需要以此行代码为首行。当在本地运行单元测试用例时，Liquid 会自动启用 std 特性；反之，当构建为可在区块链底层平台部署及运行的 Wasm 格式字节码时，std 特性将被关闭，此时 no_std 特性将被自动启用。

  由于 Wasm 虚拟机的运行时环境较为特殊，对 Rust 语言标准库的支持并不完整，因此需要启用 no_std 特性以保证智能合约代码能够被 Wasm 虚拟机执行。相反的，当在本地运行单元测试用例时，Liquid 并不生成 Wasm 格式字节码，而是生成可在本地直接运行的可执行二进制文件，因此并不受前述限制。

- 第 2~3 行：

```
2 use liquid::storage;
3 use liquid_lang as liquid;
```

  上述代码用于导入 liquid_lang 库并将其重命名为 liquid，同时一并导入 liquid_lang 库中的 storage 模块。liquid_lang 库是 Liquid 的核心组成部分，Liquid 中的诸多特性均由该库实现，而 storage 模块对区块链状态读写接口进行了封装，是定义智能合约状态变量所必须依赖的模块。

- 第 10~13 行：

```
10 #[liquid(storage)]
11 struct HelloWorld {
12     name: storage::Value<String>,
13 }
```

  上述代码用于定义 HelloWorld 合约中的状态变量，状态变量中的内容会在区块链底层存储中永久保存。可以看出，HelloWorld 合约中只包含一个名为“name”的状态变量，且其类型为字符串类型 String。但是注意到在声明状态变量类型时并没有直接写为 String，而是将其置于单值容器 storage::Value 中，更多关于容器的说明可参阅状态变量与容器一节。

- 第 15~28 行：

```
15 #[liquid(methods)]
16 impl HelloWorld {
17     pub fn new(&mut self) {
18         self.name.initialize(String::from("Alice"));
19     }
20
21     pub fn get(&self) -> String {
22         self.name.clone()
23     }
24
25     pub fn set(&mut self, name: String) {
26         self.name.set(name)
27     }
28 }
```

上述代码用于定义 HelloWorld 合约的合约方法。示例中的合约方法均为外部方法，即可被外界直接调用，其中：

- `new` 方法为 HelloWorld 合约的构造函数，构造函数会在合约部署时自动执行。示例中 `new` 方法会在初始时将状态变量 `name` 的内容初始化为字符串 “Alice”；
- `get` 方法用于将状态变量 `name` 中的内容返回至调用者
- `set` 方法要求调用者向其传递一个字符串参数，并将状态变量 `name` 的内容修改为该参数。

- 第 30~48 行：

```
30 #[cfg(test)]
31 mod tests {
32     use super::*;
33
34     #[test]
35     fn get_works() {
36         let contract = HelloWorld::new();
37         assert_eq!(contract.get(), "Alice");
38     }
39
40     #[test]
41     fn set_works() {
42         let mut contract = HelloWorld::new();
43
44         let new_name = String::from("Bob");
45         contract.set(new_name.clone());
```

(下页继续)

(续上页)

```
        assert_eq!(contract.get(), "Bob");
    }
}
```

上述代码用于编写 HelloWorld 合约的单元测试用例。首行 `#[cfg(test)]` 用于告知编译器，只有启用 `test` 编译标志时，才编译其后跟随的模块，否则直接从代码中剔除。当将 Liquid 智能合约编译为 Wasm 格式字节码时，不会启用 `test` 编译标志，因此最终的字节码中不会包含任何与测试相关的代码。代码中的剩余部分则是包含了单元测试用例的具体实现，示例中的用例分别用于测试 `get` 方法及 `set` 方法的逻辑正确性，其中每一个测试用例均由 `#[test]` 属性进行标注。

2.3 合约模块

为开发 Liquid 合约，首先需要在代码中通过 `use` 关键字引入 `liquid_lang` 库，`liquid_lang` 库包含智能合约解析功能的实现：

```
use liquid_lang as liquid;
```

上述代码使用 `as` 关键字将 `liquid_lang` 库重命名为 `liquid`，此后便可以通过这个较短的名字使用 `liquid_lang` 库提供的所有功能。

Liquid 使用 Rust 语言中的模块（`mod`）语法创建合约，在 `mod` 关键字之后是合约模块名。合约模块名能够自定义，但是建议按照 Rust 语言代码风格为其命名（即小写加下划线形式），以防编译器发出风格警告。合约模块需要使用 `#[liquid::contract]` 属性进行标注，以向 Liquid 告知该该模块中包含有智能合约各个组成部分的定义，从而引导 Liquid 解析该合约：

```
#[liquid::contract]
mod hello_world {
    ...
}
```

Rust 语言中支持为模块声明可见性（如 `pub`、`pub(crate)` 等），可用于控制当前模块能否被其他模块使用。然而对于 Liquid 而言，由于所有合约都会对外部可见，因此模块的可见性声明并无实际意义。为避免引发歧义，Liquid 禁止为合约模块添加任何可见性声明。例如，下列试图将合约模块的可见性声明为 `pub` 的代码会引发编译时报错：

```
#[liquid::contract]
pub mod hello_world {
    ...
}
```

除此之外，合约模块必须是内联的，即智能合约各个组成部分的定义都必须放置于合约模块名后、由花体括号 `{}` 括起的代码块中，从而保证 Liquid 能够完整解析智能合约。非内联形式的模块声明是非法的，例如：

```
#[liquid::contract]
mod hello_world;
```

合约模块创建完成后，便能够继续在其中定义状态变量、合约方法及事件。

2.4 状态变量与容器

状态变量用于在区块链存储上永久存储状态值，是 Liquid 合约重要的组成部分。在HelloWorld 合约中，我们已经初步接触了状态变量的定义方式及容器的使用方式。在 Liquid 合约中，状态变量与容器的关系极为密切，我们将在本节中分别对两者进行介绍。

2.4.1 状态变量

Liquid 中使用结构体语法（`struct`）对状态变量定义进行封装，并且该结构体需要使用 `liquid(storage)` 属性进行标注，以告知 Liquid 该结构体中包含了状态变量的定义，例如：

```
#[liquid(storage)]
struct HelloWorld {
    name: storage::Value<String>,
}
```

在上述代码中可以看出，结构体中每个成员各自对应一个状态变量的定义。状态变量的名称位于冒号：的左侧，而类型位于右侧，状态变量定义之间使用英语逗号，分隔。虽然在合约的设计上，状态变量 `name` 的实际类型应当为 `String`，但是在定义时需要实际类型包裹于容器类型 `storage::Value` 中。之所以要使用容器类型，是因为状态变量实际上是区块链存储系统某一存储位置的引用，对状态变量的读取、写入都需要转化为对区块链存储系统的读取、写入，这是状态变量区别于其他普通变量最重要的差异。容器是连接智能合约与区块链底层平台的桥梁，Liquid 通过容器替封装了区块链存储系统的访问细节，使得能够像使用普通变量一般使用状态变量。若没有使用容器封装状态变量的实际类型，将会引发编译时报错，例如：

```
#[liquid(storage)]
struct HelloWorld {
    name: String,
}
```

所有容器的定义均位于 `liquid_lang` 库的 `storage` 模块中，需要预先引入该模块：

```
use liquid_lang as liquid;
use liquid::storage;
```

用于封装状态变量定义的结构体在合约中能且仅能出现一次，因此不能将状态变量定义分散在不同的、用 `#[liquid(storage)]` 属性标注的结构体中：

```
#[liquid::contract]
mod hello_world {
    #[liquid(storage)]
    struct HelloWorld {
        ...
    }

    #[liquid(storage)]
    struct Another {
        ...
    }
}
```

被 #[liquid(storage)] 属性标注的结构体中至少需要一个状态变量定义，因此不能将其定义为unit 类型；同时，由于每个状态变量均需要一个有效的名称，也不能将其定义为元组类型。此外，不能为被 #[liquid(storage)] 属性标的结构体声明任何模板参数，即不能在该结构体中使用泛型，也不能为其添加任何可见性声明。下列代码了展示部分错误的使用方式：

```
// Unit is not allowed.
#[liquid(storage)]
struct HelloWorld();

// Tuple is not allowed.
#[liquid(storage)]
struct HelloWorld(u8, u32);

// Generic is not allowed.
#[liquid(storage)]
struct HelloWorld<T, E> {
    ...
}

// Visibility is not allowed.
#[liquid(storage)]
pub struct HelloWorld {
    ...
}
```

但是可以在状态变量的定义之前添加 pub 可见性声明：

```
#[liquid(storage)]
```

（下页继续）

(续上页)

```
pub struct HelloWorld {
    pub name: storage::Value<String>,
}
```

pub 可见性代表外界可以直接访问该状态变量，Liquid 会自动为此类状态变量生成一个公开的访问器。关于访问器的更多细节可参考合约方法一节。但是除了 pub 可见性以外，其他种类的可见性声明均不能使用。

2.4.2 容器

Liquid 中的容器包括单值容器（Value）、向量容器（Vec）、映射容器（Mapping）及可迭代映射容器（IterableMapping）。

注意

Liquid 中所有容器均没有实现拷贝语义，因此无法拷贝容器。同时，Liquid 限制了您不能移动容器的所有权，因此在合约中只能通过引用的方式使用容器。

单值容器

单值容器的类型定义为 Value<T>。当状态变量类型定义为单值容器时，可以像使用普通变量一般使用该状态变量。使用单值容器时需要通过模板参数传入状态变量的实际类型，如 Value<bool>、Value<String> 等。基本容器提供下列方法：

```
pub fn initialize(&mut self, input: T)
```

用于在合约构造函数中使用提供的初始值初始化单值容器。此方法应当只在构造函数中使用，且只使用一次。若状态变量初始化后再次调用 initialize 方法将不会产生任何效果。

```
pub fn set(&mut self, new_val: T)
```

用一个新的值更新状态变量的值。

```
pub fn mutate_with<F>(&mut self, f: F) -> &T
where
    F: FnOnce(&mut T),
```

允许传入一个用于修改状态变量的函数，在修改完状态变量后，返回状态变量的引用。当状态变量未初始化时，调用 mutate_with 会引发运行时异常并导致交易回滚。

```
pub fn get(&self) -> &T
```

返回状态变量的只读引用。

```
pub fn get_mut(&mut self) -> &mut T
```

返回状态变量的可变引用，可以通过该可变引用直接修改状态变量的值。

除了上述基本接口外，单值容器还通过实现 `core::ops` 中的运算符 trait，对大量的运算符进行了重载，从而能够直接使用容器进行运算。单值容器重载的运算符包括：

运算符	trait	功能	备注
*	Deref	解引用	通过容器的只读引用返回 `&T` ，借助 Rust 语言的 解引用强制多态 ，可以像操作普通变量那样操单值容器。例如：若状态变量 `name` 的类型为 `Value<String>`，如需获取 `name` 的长度，则可以直接使用 `name.len()`
*	DerefMut	解引用	通过容器的可变引用返回 `&mut T`
+	Add	加	需要 `T` 自身支持双目 `+` 运算，例如：若状态变量 `len` 的类型为 `Value<u8>`，则可以直接使用 `len + 1`
+=	AddAssign	加并赋值	需要 `T` 自身支持 `+=` 运算
-	Sub	减	需要 `T` 自身支持双目 `-` 运算
-=	SubAssign	减并赋值	需要 `T` 自身支持 `-=` 运算
*	Mul	乘	需要 `T` 自身支持 `*` 运算
*=	MulAssign	乘并赋值	需要 `T` 自身支持 `*=` 运算
/	Div	除	需要 `T` 自身支持 `/` 运算
/=	DivAssign	除并赋值	需要 `T` 自身支持 `/=` 运算
%	Rem	求模	需要 `T` 自身支持 `%` 运算
%=	RemAssign	求模并赋值	需要 `T` 自身支持 `%=` 运算
&	BitAnd	按位与	需要 `T` 自身支持 `&` 运算
&=	BitAndAssign	按位与并赋值	需要 `T` 自身支持 `&=` 运算
|	BitOr	按位或	需要 `T` 自身支持 `|` 运算
|=	BitOrAssign	按位或并赋值	需要 `T` 自身支持 `|=` 运算
^	BitXor	按位异或	需要 `T` 自身支持 `^` 运算
^=	BitXorAssign	按位异或并赋值	需要 `T` 自身支持 `^=` 运算
<<	Shl	左移	需要 `T` 自身支持 `<<` 运算
<<=	ShlAssign	左移并赋值	需要 `T` 自身支持 `<<=` 运算
>>	Shr	右移	需要 `T` 自身支持 `>>` 运算
>>=	ShrAssign	右移并赋值	需要 `T` 自身支持 `>>=` 运算
-	Neg	取负	需要 `T` 自身支持单目 `-` 运算
!	Not	取反	需要 `T` 自身支持 `!` 运算
[]	Index	下标运算	需要 `T` 自身支持按下标进行索引
[]	IndexMut	下标运算	同上，但是用于可变引用上下文中
==、!=、>、>=、<、<=	PartialEq、PartialOrd、Ord	比较运算	需要 `T` 自身支持相应的比较运算

向量容器

向量容器的类型定义为 Vec<T>。当状态变量类型为向量容器时，可以像使用动态数组一般的方式使用该状态变量。在向量容器中，所有元素按照严格的线性顺序排序，可以通过元素在序列中的位置访问对应的元素。使用向量容器时需要通过模板参数传入元素的实际类型，如 Vec<bool>、Vec<String> 等。向量容器提供下列方法：

```
pub fn initialize(&mut self)
```

用于在构造函数中初始化向量容器。若向量容器初始化后再调用 initialize 接口，则不会产生任何效果。

```
pub fn len(&self) -> u32
```

返回向量容器中元素的个数。

```
pub fn is_empty(&self) -> bool
```

检查向量容器是否为空。

```
pub fn get(&self, n: u32) -> Option<&T>
```

返回向量容器中第 n 个元素的只读引用。若 n 越界，则返回 None。

```
pub fn get_mut(&mut self, n: u32) -> Option<&mut T>
```

返回向量容器中第 n 个元素的可变引用。若 n 越界，则返回 None。

```
pub fn mutate_with<F>(&mut self, n: u32, f: F) -> Option<&T>
where
    F: FnOnce(&mut T),
```

允许传入一个用于修改向量容器中第 n 个元素的值的函数，在修改完毕后，返回该元素的只读引用。若 n 越界，则返回 None。

```
pub fn push(&mut self, val: T)
```

向向量容器的尾部插入一个新元素。当插入前向量容器的长度等于 232 - 1 时，引发 panic。

```
pub fn pop(&mut self) -> Option<T>
```

移除向量容器的最后一个元素并将其返回。若向量容器为空，则返回 None。

```
pub fn swap(&mut self, a: u32, b: u32)
```

交换向量容器中第 a 个及第 b 个元素。若 a 或 b 越界，则引发 panic。

```
pub fn swap_remove(&mut self, n: u32) -> Option<T>
```

从向量容器中移除第 n 个元素，并将最后一个元素移动至第 n 个元素所在的位置，随后返回被移除元素的引用。若 n 越界，则返回 None；若第 n 个元素就是向量容器中的最后一个元素，则效果等同于 pop 接口。

同时，向量容器实现了以下 trait：

```
impl<T> Extend<T> for Vec<T>
{
    fn extend<I>(&mut self, iter: I)
    where
        I: IntoIterator<Item = T>;
}
```

按顺序遍历迭代器，并将迭代器所访问的元素依次插入到向量容器的尾部。

```
impl<'a, T> Extend<&'a T> for Vec<T>
where
    T: Copy + 'a,
{
    fn extend<I>(&mut self, iter: I)
    where
        I: IntoIterator<Item = &'a T>;
}
```

按顺序遍历迭代器，并将迭代器所访问的元素依次插入到向量容器的尾部。

```
impl<T> core::ops::Index<u32> for Vec<T>
{
    type Output = T;

    fn index(&self, index: u32) -> &Self::Output;
}
```

使用下标对序列中的任意元素进行快速直接访问，下标的类型为 u32，返回元素的只读引用。若下标越界，则会引发运行时异常并导致交易回滚。

```
impl<T> core::ops::IndexMut<u32> for Vec<T>
{
    fn index_mut(&mut self, index: u32) -> &mut Self::Output;
}
```

使用下标对序列中的任意元素进行快速直接访问，下标的类型为 u32，返回元素的可变引用。若下标越界，

则会引发运行时异常并导致交易回滚。

向量容器支持迭代。在迭代时，需要先调用向量容器的 `iter` 方法生成迭代器，并配合 `for ... in ...` 等语法完成对向量容器的迭代，如下列代码所示：

```
#[liquid(storage)]
struct Sum {
    value: storage::Vec<u32>,
}

...

pub fn sum(&self) -> u32 {
    let mut ret = 0u32;
    for elem in self.value.iter() {
        ret += elem;
    }
    ret
}
```

注意

向量容器的长度并不能无限增长，其上限为 $2^{32} - 1$（4294967295，约为 42 亿）。

映射容器

映射容器的类型定义为 `Mapping<K, V>`。映射容器是键值对集合，当状态变量类型为映射容器时，能够通过键来获取一个值。使用映射容器时需要通过模板参数传入键和值的实际类型，如 `Mapping<u8, bool>`、`Mapping<String, u32>` 等。映射容器提供下列方法：

```
pub fn initialize(&mut self)
```

用于在构造函数中初始化映射容器。若映射容器初始化后再调用 `initialize` 接口，则不会产生任何效果。

```
pub fn len(&self) -> u32
```

返回映射容器中元素的个数。

```
pub fn is_empty(&self) -> bool
```

检查映射容器是否为空。

```
pub fn insert<Q>(&mut self, key: &Q, val: V) -> Option<V>
```

向映射容器中插入一个由 key、val 组成的键值对，注意 key 的类型为一个引用。当 key 在之前的映射容器中不存在时，返回 None；否则返回之前的 key 所对应的值。

```
pub fn mutate_with<Q, F>(&mut self, key: &Q, f: F) -> Option<&V>
where
    K: Borrow<Q>,
    F: FnOnce(&mut V),
```

允许传入一个用于修改映射容器中 key 所对应的值的函数，在修改完毕后，返回值的只读引用。若 key 在映射容器中不存在，则返回 None。

```
pub fn remove<Q>(&mut self, key: &Q) -> Option<V>
where
    K: Borrow<Q>,
```

从映射容器中移除 key 及对应的值，并返回被移除的值。若 key 在映射容器中不存在，则返回 None。

```
pub fn get<Q>(&self, key: &Q) -> Option<&V>
```

返回映射容器中 key 所对应的值的只读引用。若 key 在映射容器中不存在，则返回 None。

```
pub fn get_mut<Q>(&mut self, key: &Q) -> Option<&mut V>
```

返回映射容器中 key 所对应的值的可变引用。若 key 在映射容器中不存在，则返回 None。

```
pub fn contains_key<Q>(&self, key: &Q) -> bool
```

检查 key 在映射容器中是否存在。

同时，映射容器实现了以下 trait：

```
impl<K, V> Extend<(K, V)> for Mapping<K, V>
{
    fn extend<I>(&mut self, iter: I)
    where
        I: IntoIterator<Item = (K, V)>;
}
```

按顺序遍历迭代器，并将迭代器所访问的键值对依次插入到映射容器中。

```
impl<'a, K, V> Extend<(&'a K, &'a V)> for Mapping<K, V>
where
```

（下页继续）

(续上页)

```
    K: Copy,
    V: Copy,
{
    fn extend<I>(&mut self, iter: I)
    where
        I: IntoIterator<Item = (&'a K, &'a V)>;
}
```

按顺序遍历迭代器，并将迭代器所访问的键值对依次插入到映射容器中。

```
impl<'a, K, Q, V> core::ops::Index<&'a Q> for Mapping<K, V>
where
    K: Borrow<Q>,
{
    type Output = V;

    fn index(&self, index: &'a Q) -> &Self::Output;
}
```

以键为索引访问映射容器中对应的值，索引类型为 `&Q`，返回值的只读引用。若索引不存在，则会引发运行时异常并导致交易回滚。

```
impl<'a, K, Q, V> core::ops::IndexMut<&'a Q> for Mapping<K, V>
where
    K: Borrow<Q>,
{
    fn index_mut(&mut self, index: &'a Q) -> &mut Self::Output;
}
```

以键为索引访问映射容器中对应的值，索引类型为 `&Q`，返回值的可变引用。若索引不存在，则会引发运行时异常并导致交易回滚。

注意

映射容器的容量大小并不能无限增长，其上限为 $2^{32} - 1$（4294967295，约为 42 亿）。

注意

映射容器不支持迭代，如果需要迭代映射容器，请使用可迭代映射容器。

可迭代映射容器

可迭代映射容器的类型定义为 IterableMapping<K, V>，其功能与映射容器基本类似，但是提供了迭代功能。使用可迭代映射容器时需要通过模板参数传入键和值的实际类型，如 IterableMapping<u8, bool>、IterableMapping<String, u32> 等。

可迭代映射容器支持迭代。在迭代时，需要先调用可迭代映射容器的 iter 方法生成迭代器，迭代器在迭代时会不断返回由键的引用及对应的值的引用组成的元组，可配合 for ... in ... 等语法完成对可迭代映射容器的迭代，如下列代码所示：

```
#[liquid(storage)]
struct Sum {
    values: storage::IterableMapping<String, u32>,
}

...

pub fn sum(&self) -> u32 {
    let mut ret = 0u32;
    for (_, v) in self.values.iter() {
        ret += v;
    }
    ret
}
```

此外，可迭代映射容器的 insert 方法与映射容器略有不同，其描述如下：

```
pub fn insert(&mut self, key: K, val: V) -> Option<V>
```

其功能与映射容器的 insert 方法相同，但是其参数并不 K 类型的引用。

注意

可迭代映射容器的容量大小并不能无限增长，其上限为 $2^{32} - 1$（4294967295，约为 42 亿）。

注意

为实现迭代功能，可迭代映射容器在内部存储了所有的键，且受限于区块链特性，这些键不会被删除。因此，可迭代容器的性能及存储开销较大，请根据应用需求谨慎选择使用。

2.5 合约方法

合约方法可以用于访问合约的状态变量，并向调用者返回调用结果。在定义了合约状态变量后，我们可以通过为被 `#[liquid(storage)]` 属性标注的结构体实现成员方法来定义合约方法，其语法如下所示：

```
#[liquid(storage)]
struct Foo {
    ...
}

#[liquid(methods)]
impl Foo {
    ...
}
```

在上述代码中，状态变量定义位于 `Foo` 结构体类型中，因此需要通过为 `Foo` 结构体类型实现成员方法来定义合约方法时，所有合约方法的定义放置于 `impl` 代码块，请注意 `struct` 代码块与 `impl` 代码块中的类型名称必须要一致，同时需要使用 `#[liquid(methods)]` 属性标注 `impl` 代码块，以告知 Liquid 该代码块中包含合约方法的定义。

虽然在 Liquid 合约中只能将状态变量的定义集中至一处中，但是合约方法的定义并不存在这个限制，您可以将合约方法的定义分散在多个 `impl` 代码块中。Liquid 在解析合约时，会自动组合这些分散的 `impl` 代码块。但是对于简单的合约，我们一般不推荐这样做，这样会使得合约代码看起来较为凌乱。例如，HelloWorld 合约也可以写成如下形式：

```
#[liquid(storage)]
struct HelloWorld {
    ...
}

#[liquid(methods)]
impl HelloWorld {
    pub fn new(&mut self) {
        ...
    }
}

#[liquid(methods)]
impl HelloWorld {
    pub fn get(&self) -> String {
        ...
    }
```

(下页继续)

(续上页)

```
}

#[liquid(methods)]
impl HelloWorld {
    pub fn set(&mut self, name: String) {
        ...
    }
}
```

注意

定义合约方法时，请不要在 `impl` 关键字前添加 `default` 关键字或任何可见性声明。

2.5.1 方法签名

Liquid 中合约方法的签名由可见性声明、合约方法名、接收器（Receiver）、参数及返回值组成。除此之外，不允许为合约方法添加 `const`、`async`、`unsafe` 或 `extern "C"` 等修饰符，也不能使用模板参数或者可变参数。

可见性声明

可见性声明只能为 `pub` 或者为空，当可见性为 `pub` 时，表示该合约方法是公开方法，可供外部用户或其他合约调用；反之，若可见性声明为空，则表示该合约方法是私有方法，只能在合约内部调用：

```
// Public method.
pub fn plus(&self, x: u8, y: u8) -> u8 {
    self.plus_impl(x, y)
}

// Private method.
fn plus_impl(&self, x: u8, y: u8) -> u8 {
    x + y
}
```

接收器

合约方法的接收器只能为 `&self` 或 `&mut self`。Liquid 在执行合约方法时会自动生成一个合约对象，`&self` 即表示该合约对象的只读引用，而 `&mut self` 则表示该合约对象的可变引用。只有通过接收器才能够访问合约中的状态变量及方法，即只能通过 `self.foo` 或 `self.bar()` 之类形式访问合约状态变量或合约方法。

当接收器为 `&self` 时，表明该合约方法是一个只读方法（类似于 Solidity 语言中的 `view` 或 `pure` 修饰符的功能），此时无法在方法中改变任何状态变量的值，也无法调用任何能够改变合约状态的其他方法。调用只读方法时，不会生成交易，即相关操作记录无需区块链节点共识，也不会以交易的形式记录于区块链上；当接收器为 `&mut self` 时，表明该合约方法是一个可写方法，即能够修改状态变量的值，也能够调用合约中其他任何可写或只读方法。调用可写方法时，区块链节点间会就对应交易进行共识并将相关交易记录于区块链上：

```
#[liquid(storage)]
struct Foo {
    value: storage::Value<u8>,
}

#[liquid(methods)]
impl Foo {
    pub fn read_only(&self) -> u8 {
        // Compile error, can't modify state in an read-only method.
        self.x += 1;
        self.x
    }

    pub fn writable(&mut self) -> u8 {
       // Pass
       self.x += 1;
       self.x
    }
}
```

参数

Liquid 强制要求合约方法的第一个参数必须为接收器，合约方法要使用到的其他参数需要跟在接收器之后。为在编译期确定合约参数的解码方式，Liquid 限制合约方法的参数（不包括接收器在内）个数不能超过 16 个。与 Solidity，当前 Liquid 只限制合约方法的参数个数不能超过 16 个，但是对局部变量的个数没有限制，未来可能会放宽这一限制。

当前，为兼容 Solidity，只有在 Solidity 中有对应类型的数据类型才能用作公开方法的参数类型（如 `u8`、`String` 等），未来可能会放宽这一限制，关于类型的更多信息请参考类型一节。私有方法的参数类型则没有该限制，可以使用包括引用、`Option<T>` 及 `Result<T, E>` 在内的任意数据类型作为私有方法的参数类型。

```
// Compile error, for now `Option<u8>` is not supported in public method
pub fn foo(&self, x: Option<u8>) {
    ...
}
```

（下页继续）

(续上页)

```

// Pass.
fn bar(&self, y: Option<u8>) {
    ...
}
```

返回值

当合约方法没有返回值时，可以不写返回值类型或令返回值类型为 unit 类型（即 `()`）：

```
pub fn foo(&self) {
    ...
}

pub fn bar(&self) -> () {
    ...
}
```

当合约方法有一个返回值时，直接将返回值的类型置于`->` 后即可：

```
pub fn foo(&self) -> String {
    ...
}
```

当合约方法有多个返回值时，需要将返回值类型写为元组的形式，元组中每个元素即是一个返回值类型：

```
pub fn foo(&self) -> (String, bool, u8) {
    (String::from("hello"), false, 0u8)
}
```

与参数类型的限制类似，为在编译期确定合约返回值的编码方法，Liquid 限制合约方法的返回值个数不能超过 16 个，未来可能会放宽这一限制。同时，只有在 Solidity 中有对应类型的数据类型才能用作公开方法的返回值类型，私有方法的返回值类型则没有这个限制。

2.5.2 构造函数

构造函数是一种特殊的合约方法，用于在部署合约时自动执行，且不能被用户或外部合约调用。Liquid 合约中，构造函数名字必须为 `new`，合约中必须有且只有一个构造函数，因此在 Liquid 合约中无法定义同名的其他合约方法。此外，构造函数的可见性必须为 `pub`、接收器必须为 `&mut self` 且不能有返回值，合法的构造函数形式如下列代码所示：

```
pub fn new(&mut self, ...) {
    ...
}
```

构造函数对于 Liquid 合约极其重要，因为 Liquid 并不会主动为状态变量分配默认值，因此要求在使用状态变量之前务必先初始化状态变量，否则会引发运行时异常，而构造函数则是最适合用于执行状态变量初始化。尽管也可以在其他合约方法中初始化状态变量，但是并不推荐这样做，因为外部用户或其他合约可能跳过该合约方法的调用，但是构造函数在部署时一定会被执行。因此，请尽量将所有状态变量初始化的工作放置于构造函数中，例如：

```
#[liquid(storage)]
struct Foo {
    b: storage::Vec<bool>,
    i: storage::Value<i32>,
}

#[liquid(methods)]
impl Foo {
    pub fn new(&mut self) {
        self.b.initialize();
        self.i.initialize(0);
    }
}
```

不要忘记初始化状态变量。

请将上面这句话默读三遍，然后喝杯咖啡，接着再读一遍。

2.5.3 访问器

在状态变量与容器一节中，我们提到可以将状态变量的可见性声明为 `pub`，Liquid 将会自动为该状态变量生成一个访问器以用于外界直接读取该状态变量的值。访问器是一个与状态变量同名的公开方法，假设有状态变量的定义如下：

```
#[liquid(storage)]
struct Foo {
    pub b: storage::Value<bool>,
}
```

当将状态变量 `b` 的可见性声明为 `pub` 时，可以理解为 Liquid 会在合约中自动插入以下代码：

```
#[liquid(methods)]
impl Foo {
    pub fn b(&self) -> bool {
        self.b.get()
    }
}
```

因此，当指定要为某个状态变量生成访问器时，合约中将不能再定义一个同名的合约方法，否则编译器会报重复定义错误，例如：

```
#[liquid(storage)]
struct Foo {
    pub b: storage::Value<bool>,
}

#[liquid(methods)]
impl Foo {
    // Compile error, attempt to redefine `b`.
    pub fn b(&self) {
        ...
    }
}
```

不同容器类型所生成访问器并不相同，其区别见下表（表中我们假定状态变量的名字为 `foo`）：

2.5.4 杂注

Liquid 规定合约模块中所有的 `impl` 代码块都需要被 `#[liquid(methods)]` 属性标注，即合约模块中的 `impl` 代码块只能用于定义合约方法。当在合约模块中试图为另外某个类型实现成员或静态方法时将导致编译报错，例如：

```
#[liquid::contract(version = "0.1.0")]
mod foo {
    #[liquid(storage)]
    struct Foo {
        bar: String,
    }

    // 合约方法
    impl Foo {
        // ...
```

（下页继续）

(续上页)

```
    }

    // 另外一个普通结构体的定义
    struct Ace {
        // ...
    }

    // 编译错误，存在多个 impl 代码块
    impl Ace {
        // ...
    }
}
```

```
#[liquid::contract]
mod foo {
    #[liquid(storage)]
    struct Foo {
        ...
    }

    // Pass, definition of contract methods is allowed.
    impl Foo {
        ...
    }

    // The definition of another type.
    struct Ace;

    // Compile error, `impl` blocks in contract should be tagged with `#[liquid(methods)]`.
    impl Ace {
        ...
    }
}
```

但如果的确有类似的需求，可以将该类型成员或静态方法的实现挪出合约模块的，然后再在合约模块内引用相关符号，例如：

```
// The definition of another type.
struct Ace;
```

(下页继续)

(续上页)

```

// Implementations...
impl Ace {
    ...
}

#[liquid::contract]
mod foo {
    // Reference outer symbols
    use super::Ace;
}
```

2.6 事件

事件是区块链底层虚拟机日志基础设施提供的一个便利接口。当触发事件时，事件中的参数存储到交易收据的日志字段中，日志是一种特殊的数据结构，这些日志与合约地址相关联，并随交易收据记录到区块链中。每条交易收据中可以包含 0 条或多条日志记录。在分布式应用中，如果监听了某事件，则当该事件发生时，便会触发应用相应的回调。

2.6.1 创建事件

Liquid 中使用结构体（`struct`）语法定义事件。结构体中的每个成员都是事件的参数，为向 Liquid 告知该结构体用于定义事件，需要使用 `#[liquid(event)]` 属性标注该结构体，例如：

```
#[liquid(event)]
struct Foo {
    s: String,
    i: i32,
}
```

上述代码中，我们定义了一个名为 `Foo` 的事件，事件中包含两个参数，分别为 `String` 及 `i32`。更进一步，还可以使用 `#[liquid(indexed)]` 属性将事件参数标注为可被索引：

```
#[liquid(event)]
struct Foo {
    #[liquid(indexed)]
    s: String,
    i: i32,
}
```

被索引的参数本身不会被保存，但是分布式应用可以通过被索引参数的值来对事件进行检索。在 Liquid 中，一个事件最多有四个参数可被用于被索引，但是第一个索引恒定为事件签名（事件名及其参数类型）的哈希值，因此在事件定义中，最多可以使用 `#[liquid(indexed)]` 标注三个参数。

与状态变量定义类似，不能为定义事件的结构体添加可见性声明或模板参数。但和状态变量定义不同的是，其内部每个成员也不允许添加可见性声明。当前为与 Solidity 兼容，事件参数及索引参数的类型均需要在 Solidity 中存在相应的类型，具体的限制可参考类型一节，未来可能会放宽这一限制。

2.6.2 触发事件

在 Liquid 中，通过环境对象触发事件。环境对象由 Liquid 自动生成，可以在合约方法中通过调用 `self.env()` 来获取环境对象。获取环境对象后，可以通过调用环境对象的 `emit` 方法来触发我们之前定义的事件，例如：

```
self.env().emit(Foo {
    s: String::from("hello"),
    i: 42,
})
```

上述代码中，`emit` 方法以事件对象为参数，事件对象可通过结构体初始化语法直接进行构造。提供给 `emit` 方法的参数类型一定需要是有效的事件类型（即被 `#[liquid(event)]` 属性标注的结构体类型），否则会报出类型不匹配的编译错误。事件被出发后，对应交易的回执中会多出一条日志记录，例如：

```
"logs": [
    {
        "address": "0x6119432a43a2a5da27f31fa4912f1c43400b1690",
        "data": "0x000000000000000000000000000000000000000000000000000000000000002a",
        "topics": [
            "0x1be2d150ed559c350b05f7dfa5a74669ec8d2ce63bb14c134730ffa02d2d111c",
            "0x1c8aff950685c2ed4bc3174f3472287b56d9517b9c948127319a09a7a36deac8"
        ]
    }
]
```

日志记录，`address` 字段是合约地址；`data` 字段中保存了非索引参数的ABI 编码，此处因为我们只有一个非索引参数 `i`，因此 `data` 字段中只保存了它的值 42；`topics` 字段包含了两个可用于索引该事件的值，其中第一个是事件签名的哈希值，第二个则是事件中索引参数 `s` 的值的哈希值。对于 `String` 这类动态对象，Liquid 会将它们的哈希值作为事件索引，以提高检索效率并减少存储空间占用。因此若需要在应用中按照字符串检索事件，则需要在本地预先计算待检索字符串的哈希值。

注意

Liquid 目前支持将合约编译为国密版本或非国密版本，两种版本的合约在计算哈希值时采用的哈希算法并不

相同，分别为 sm3 和 keccak256 。如果需要使用动态对象索引事件，则请务必确保所使用的哈希算法与产生日志的 Liquid 合约一致。

2.7 类型

由于 Liquid 以 Rust 语言为宿主语言，因此合约中能够使用 Rust 语言支持的所有数据类型。为方便合约编写，Liquid 也提供了一系列内置数据类型。此外，受编解码机制的限制，在状态变量、合约方法及事件的定义中能够使用的数据类型会受到一定限制。本节将会对这些知识要点进行逐一介绍。

2.7.1 地址类型

地址类型（Address）是字符串类型 String 的别名，可用于表示账户及合约地址，其定义为：

```
pub struct Address(String);
```

Liquid 为 Address 实现了用于与 String 类型相互转换的 trait，因此其使用方式与 String 基本一致：

```
let addr: Address = String::from("/usr/bin/").into();
assert_eq!(addr.as_bytes(), addr_str.as_bytes());
```

2.7.2 动态字节数组类型

容纳字节数据的数组，其类型名称为 bytes，是 Vec<u8> 类型的封装，数组长度运行时动态可变。bytes 类型提供以下方法：

```
pub fn new() -> Self
```

构造一个空字节数组

同时，bytes 类型还实现了以下 trait：

```
impl core::ops::Deref for Bytes {
    type Target = Vec<u8>;

    fn deref(&self) -> &Self::Target;
}
```

```
impl core::ops::DerefMut for Bytes {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

通过内部 Vec<u8> 数组的只读引用或可变引用，通过 Rust 语言的解引用强制多态，可以直接在 bytes 类型对象上使用 Vec<u8> 提供的成员方法，例如：

```
let mut b1 = Bytes::new();
b1.push(1);
assert_eq!(b1.len(), 1);
assert_eq!(b1[0], 1);
```

```
impl From<&[u8]> for Bytes {
    fn from(origin: &[u8]) -> Self;
}

impl<const N: usize> From<[u8; N]> for Bytes {
    fn from(origin: [u8; N]) -> Self;
}

impl<const N: usize> From<&[u8; N]> for Bytes {
    fn from(origin: &[u8; N]) -> Self;
}

impl From<Vec<u8>> for Bytes {
    fn from(origin: Vec<u8>) -> Self;
}
```

用于将 u8 类型的切片、数组及动态数组转换为 bytes 类型对象。

注解： bytes 是 Bytes 的类型别名。

2.7.3 定长数组类型

容纳字节数据的数组，但其数组长度在编译期长度就已经确定，是对应长度 u8 数组类型的封装。Liquid 提供 bytes1、bytes2、…、bytes32 共 32 种类型，分别代表长度为 1、2、…、32 的定长字节数组类型。bytes#N 类型实现了以下 trait：

```
// Same for Bytes2, Bytes3...
impl core::ops::Shl<usize> for Bytes1 {
    type Output = Self;

    fn shl(mut self, mid: usize) -> Self::Output;
}
```

(下页继续)

(续上页)

```
// Same for Bytes2, Bytes3...
impl core::ops::Shr<usize> for Bytes1 {
    type Output = Self;

    fn shr(mut self, mid: usize) -> Self::Output;
}
```

左移及右移运算。注意 bytes#N 类型的移位是按位进行，而不是按字节，因此例如有类型为 bytes1 的变量 b，其内容为 0b01010101，则执行 b << 1 后所得结果为 0b10101010u8。另外 bytes#N 类型的移位运算**不是**循环移位，移出的左（右）端的位将会被直接丢弃，同时在右（左）端补零。

```
// Same for Bytes2, Bytes3...
impl core::ops::BitAnd for Bytes1 {
    type Output = Self;

    fn bitand(self, rhs: Self) -> Self::Output;
}

// Same for Bytes2, Bytes3...
impl core::ops::BitOr for Bytes1 {
    type Output = Self;

    fn bitor(self, rhs: Self) -> Self::Output;
}

// Same for Bytes2, Bytes3...
impl core::ops::BitXor for Bytes1 {
    type Output = Self;

    fn bitxor(self, rhs: Self) -> Self::Output;
}
```

按位与、或及异或运算。

```
// Same for Bytes2, Bytes3...
impl FromStr for Bytes1 {
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

将一个字符串转换为 bytes#N 类型对象，转换时会直接将字符串的原始字节数组填入 bytes#N 类型对象

中。要求字符串的原始字节数组长度必须要小于或等于定长字节数组的长度，若长度将会在左端补零。由于str类型为实现了 FromStr trait 的类型自动实现了 parse 方法，因此可以在代码中使用如下方式将符合要求的字符串转换为 bytes#N 类型对象：

```
// Due to that string in Rust using UTF-8 encoding,
// `b` equals to [0xe4, 0xbd, 0xa0, 0xe5, 0xa5, 0xbd]
let b: bytes6 = "你好".parse().unwrap();
```

```
// Same for Bytes2, Bytes3...
impl core::ops::Index<usize> for Bytes1 {
    type Output = u8;

    fn index(&self, index: usize) -> &Self::Output;
}

impl core::ops::IndexMut<usize> for Bytes1 {
    fn index_mut(&mut self, index: usize) -> &mut Self::Output;
}
```

支持通过下标对字节数组中的值进行随机访问，返回对应字节的只读或可变引用。下标的类型为 usize。

bytes#N 类型实现了整数类型到 bytes#N 类型、bytes#N 类型到 bytes#N 类型的转换，所有转换都是通过实现相应的From trait 实现。整数类型转换到 bytes#N 类型时，要求整数类型的存储大小不得超过目标定长字节数组的长度；bytes#N 类型到 bytes#N 类型时，要求原始字节数组的长度不得超过目标定长字节数组的长度，例如：

```
let b1: bytes1 = 0b10101010u8.into();
let b2: bytes32 = b1.into();
```

此外，bytes#N 类型还实现了 Copy、Clone、PartialEq、Eq、PartialOrd、Ord trait，因此可以直接对长度相同的 bytes#N 类型对象使用值拷贝，或在长度相同的 bytes#N 类型对象间进行大小比较。

注解： bytes1 、bytes2 、…、bytes32 分别是 Bytes1 、Bytes2 、…、Bytes32 的类型别名。

2.7.4 大整数类型

Liquid 中的大整数类型包括 u256 及 i256，分别对应无符号 256 位整数及有符号 256 位整数。u256、i256 的使用方式与 Rust 语言中的原生整数类型类似，支持**同类型之间**的加、减、乘、除、大小比较等运算，其中 i256 还支持取负运算。

u256 类型及 i256 类型提供的方法及构造方式类似，只是由于 i256 能够表示负数，因此其数值表示范围与 u256 不相同。与在此仅对 u256 类型进行详细介绍，i256 同理类推即可。u256 类型实现了以下 trait：

```
impl FromStr for u256 {
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

基于 10 进制或 16 进制字符串构造 u256 类型对象，其中 16 进制字符串必须以 0x 或 0X 开头。当字符串中包含非法字符时会引发运行时异常。

```
#[cfg(feature = "std")]
impl fmt::Display for u256 {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result;
}

#[cfg(feature = "std")]
impl fmt::Debug for u256 {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result;
}
```

用于将 u256 类型对象转换为格式化字符串。需要注意的是，上述实现仅在进行合约单元测试时提供，在正式的合约代码中不允许使用上述实现。

此外，u256 类型还实现了各种整数类型（包括有符号整数类型）到 u256 类型的转换。支持有符号整数类型转换到 u256 类型的原因是为了方便开发者书写如下代码：

```
let u: u256 = 1024.into();
```

由于 Rust 语言编译器在做类型推断时会将表示范围内的整数自动推导为有符号整数类型，例如上述代码中 1024 会被推导为 i32 类型，若没有实现有符号整数类型转换到 u256 类型的转换，开发者将不得不将上述代码改写为：

```
let u: u256 = 1024u32.into();
```

但是若尝试将一个负数转换为 u256 类型对象，会导致引发运行时异常。i256 类型则没有这个问题。

2.7.5 类型限制

为节省链上存储空间及提高编解码效率，Liquid 使用了紧凑的二进制编码格式SCALE来对状态变量进行编解码。因此只要能够被 SCALE 编解码器编解码的类型，就都能够用于定义状态变量、合约方法参数、合约方法返回值及事件参数的实际类型，这些类型包括：

- 基本类型
  - bool
  - u8，u16，u32，u64，u128，u256

  - i8, i16, i32, i64, i128, i256
  - String
  - Address
  - bytes
  - bytes1, bytes2, …, bytes32
  - Option
  - Result
- 复合类型
  - 元组类型
  - 数组类型
  - 动态数组类型（Vec<T>）
  - 结构体类型
  - 枚举类型，但最多能够有 256 个枚举变体（variants）

当使用复合类型时，Liquid 要求它们的各个成员或元素类型也同样能够被 SCALE 编解码器编解码，特别的，复合类型能够嵌套复合类型，如 Vec<[(u8, Address); 5]>。对于结构体类型，若需要用于定义状态变量的类型，则必须要在结构体定义前 derive InOut 属性，否则会引发编译报错，其中 InOut 属性的定义位于 liquid_lang 中，需要在合约代码中提前导入：

```
use liquid_lang::InOut;

 #[derive(InOut)]
pub struct Baz {
    b: bool,
    i: Baz,
}

#[derive(InOut)]
pub struct Foo {
    b: bool,
    i: Baz,
}
...
#[liquid(storage)]
struct Bar {
    foo: storage::Value<Foo>,
}
```

需要注意的是，尽管此处的动态数组（Vec<T>）与容器中的向量容器（storage::Vec<T>）名称上类似，但两者是完全不一样的概念。向量容器能以类似动态数组的方式访问区块链底层存储，而动态数组的实现则是由 Rust 语言的标准库提供，表示内存中一段连续的存储空间。两者的区别主要体现在：

- 动态数组中相邻元素在内存中的位置也是相邻的，但向量容器中相邻元素在区块链底层存储中的位置并不一定是相邻的；
- 动态数组支持在任意位置插入或删除元素，但向量容器只能在尾部插入及删除元素；
- 动态数组能够直接使用 for ... in ... 语法进行迭代，但向量容器在使用 for ... in ... 语法进行迭代前必须要先调用 iter() 方法生成迭代器；
- 动态数组能够整体作为一个状态变量的值存入区块链存储中，但是向量容器无法做到这一点。例如，下列代码展示了在单值容器中存放动态数组：

```
#[liquid(storage)]
struct Foo {
    foo: storage::Value<Vec<u8>>,
}
```

但是不能将状态变量定义为：

```
#[liquid(storage)]
struct Foo {
    foo: storage::Value<storage::Vec<u8>>,
}
```

注意

上述示例中形如 storage::Value<Vec<u8>> 的容器使用方式并不为我们所推荐。因为这种情况下，每次初次读取该状态变量时，都需要从区块链底层存储读入所有元素的编码并从中解码出完整的动态数组；当更新该状态变量后、需要写回至区块链底层存储时，同样需要对动态数组的所有元素进行编码然后再写回至区块链存储中。当动态数组中的元素个数较多时，编解码过程中将会带来极大的计算开销。正确的方式应该是使用向量容器 storage::Vec<u8> 。

事件参数定义中的类型限制与上述规则一致，但当某个参数被设置为可索引时，该参数的定义中能够使用的类型进一步收窄为：

- bool
- u8，u16，u32，u64，u128，u256
- i8，i16，i32，i64，i128，i256
- String
- Address

2.8 环境与内置方法

2.8.1 环境

环境能够用于在合约代码中访问某些区块链执行上下文中的信息。以获取合约调用者的账户地址为例，可以通过如下形式在合约方法中借助环境取得该信息：

```
self.env().get_caller();
```

其中 `self` 是执行合约方法时当前合约对象的引用。在构建合约时，Liquid 会自动在合约实现一个名为 `env` 的私有方法。`env` 方法不接受任何参数，但会返回一个环境访问器。获得环境访问器后，便可以通过环境访问器调用所需方法，能且仅能通过环境访问器获取区块链执行上下文信息。目前环境访问器提供了以下方法：

```
pub fn get_caller(self) -> Address
```

获取合约调用者的账户地址。

```
pub fn get_tx_origin(self) -> Address
```

获取整个合约调用链中，最开始发起调用的调用方的账户地址，此时获得的账户地址一定是一个外部账户地址。

```
pub fn now(self) -> timestamp
```

获取当前区块的时间戳，以 13 位时间戳的形式表示。其中 `timestamp` 为 `u64` 类型的别名。

```
pub fn get_address(self) -> Address
```

```
pub fn is_contract(self, account: &Address) -> bool
```

```
 pub fn emit<E>(self, event: E)
```

触发事件，要求模板参数 `E` 必须为被 `#[liquid(event)]` 属性标注的结构体类型。

每次调用环境相关的的接口时，都需要消耗一个环境访问器，因此不能通过如下方式复用环境访问器：

```
let env_access = &self.env();
let caller = env_access.get_caller();
// Compile error, due to that `env_access` had been consumed already.
env_access.emit(some_event);
```

正确的方式是每次调用环境相关的接口时都调用 `self.env()` 创建一个环境访问器对象。环境访问器极为轻量，因此无需担心创建时的性能开销：

```
let caller = self.env().get_caller();
self.env().emit(some_event);
```

2.8.2 内置函数

Liquid 提供了一些基本的内置函数。在构建合约时，Liquid 会自动导入这些函数，因此在合约代码中可以直接使用这些函数。内置函数包括：

```
pub fn require<Q>(expr: bool, msg: Q)
where
    Q: AsRef<str>,
```

断言函数，用于判断布尔类型的断言表达式 `expr` 是否成立。若断言成立，则合约代码继续向下执行；若不成立，则直接终止合约代码的运行并引发交易回滚，然后将异常信息 `msg` 放入交易回执中一并返回至合约的调用方。

2.9 外部合约调用

2.9.1 外部合约声明

当需要调用外部合约时，需要首先在代码中声明外部合约所包含的公开方法。同合约模块类似，外部合约声明也需要使用模块（`mod`）模块语法，并需要在模块定义前使用 `#[liquid::interface]` 属性进行标注，例如：

```
#[liquid::interface(name = auto)]
mod entry {
    ...
}
```

在用于声明外部合约的模块中，可以使用以下语法元素：

- 符号引入：使用 `use ... as ...` 语法，以将在模块外部定义的符号引入至当前模块中，例如：

```
#[liquid::interface(name = auto)]
mod kv_table {
    use super::entry::*;
    ...
}
```

- 结构体类型定义：使用 `struct` 结构体语法定义新的结构体类型，该结构体类型之后可用于定义外部合约公开方法的参数或返回值的类型，例如：

```
#[liquid::interface(name = auto)]
mod kv_table {
    struct Result {
        success: bool,
        value: Entry,
    }
    ...
}
```

所定义的结构体类型中，不允许为成员指定可见性。同时，由于外部合约声明中的结构体类型定义一般会用于定义外部合约方法的参数或返回值的类型，因此 Liquid 会自动为这些结构体类型添加 `#[derive(liquid_lang::InOut)]` 属性，请勿重复标注该属性。

- 合约方法声明：所有外部合约公开方法的声明都需要封装于 `extern` 关键字后、由花体括号 `{}` 括起的代码块中，例如：

```
#[liquid::interface(name = auto)]
mod entry {
    extern "liquid" {
        fn getInt(&self, key: String) -> i256;
        ...
    }
}
```

不允许为外部合约方法的声明添加任何可见性声明，因为这些方法必定都是公开的。由于在执行外部合约调用时需要计算目标方法的选择器，因此需要所声明的方法签名（包括方法名称及参数类型）与目标方法的实际签名完全一致，即使方法名称可能并不满足 Rust 语言编程规范中关于“方法名必须使用 snake_case 式命名”的要求。为避免 Rust 编译器报出代码风格警告，Liquid 会自动为所有外部合约方法的声明添加 `#[allow(non_snake_case)]` 属性。

在用于外部合约声明的模块中，必须有且只能有一个 `extern` 代码块，且该代码块中需要有至少一个合约方法的声明。`extern` 代码块中，只能包含外部合约公开方法的签名，而不能包含其实现。每个外部合约公开方法的签名中，第一个参数必须为接收器，可以为 `&self` 或 `&mut self`，用于表示该方法是否为只读方法。所声明的的只读性必须要和目标方法的只读性一致，否则可能会导致调用失败。外部合约公开方法的声明中无需包含构造函数的声明。

外部合约声明的描述对象与合约模块相同，两者均是对智能合约行为的描述，只是外部合约声明中并不包含其行为的具体实现，因此两者在语义上属于同等地位。当需要声明外部合约时，较好的代码组织方式是将外部合约声明与合约模块放置于同级的命名空间中，而不是在合约模块内部放置外部合约声明：

```
// Good programming practice (•_•).
#[liquid::interface(name = auto)]
mod entry {
```

（下页继续）

(续上页)

```
    ...
}

#[liquid::contract]
mod kv_table_test {
    ...
}

// Bad programming practice (× F ×).
#[liquid::contract]
mod kv_table_test {
    #[liquid::interface(name = auto)]
    mod entry {
        ...
    }
    ...
}
```

2.9.2 调用外部合约

构建合约时，Liquid 会在声明外部合约的模块中自动生成一个代表外部合约的类型，在后文中，我们称这个类型为外部合约类型。外部合约类型可以用于构造外部合约对象，通过外部合约对象便可调用外部合约公开方法。

尽管我们始终没有解释，但从前面的示例可以观察到，在声明外部合约时，所使用的 `#[liquid::interface]` 属性中包含了一个名为 `name` 的参数。`name` 参数用于指定所生成的外部合约类型的名字，其参数可以为 `auto` 或一个字符串常量。当指定 `name` 参数为 `auto` 时，Liquid 会将声明外部合约所使用的模块名的“CamelCase”式命名作为外部合约类型的名字。例如在上述名为 `kv_table` 的外部合约声明中，由于 `name` 参数被指定为 `auto`，因此所声明的外部合约类型名为 `KvTable`；当 `name` 参数为一个字符串常量时，则外部合约类型的名字是参数所指定的名称。例如，若将上述外部合约声明改写为：

```
#[liquid::interface(name = "Foo")]
mod kv_table {
    ...
}
```

此时外部合约类型的名称便是 `Foo`。

注意

请注意 `name = auto` 与 `name = "auto"` 的区别：前者的 `auto` 没有双引号，用于指示 Liquid 按照驼峰规则

自动生成外部合约类型名称；后者的 auto 带有有双引号，用于指示 Liquid 生成一个名为 auto 的外部合约类型。

外部合约类型可以用在合约模块或外部合约声明中的任何位置。外部合约类型既能够用于定义合约方法参数或返回值的类型，也可以用于定义状态变量或临时变量的类型。使用外部合约类型时，需要先将其符号导入，导入方式如下列代码所示：

```
#[liquid::interface(name = auto)]
mod kv_table_factory {
    extern "liquid" {
        fn openTable(&self, name: String) -> KvTable;
        ...
    }
}

#[liquid::contract]
mod kv_table_test {
    use super::{kv_table_factory::*};

    #[liquid(storage)]
    struct KvTableTest {
       table_factory: storage::Value<KvTableFactory>,
    }
    ...
}
```

需要先通过外部合约类型构造出外部合约对象后，才能通过外部合约对象调用外部合约公开方法。可使用下列两种方式构造外部合约对象：

- 外部合约类型所提供的 at 方法。at 是一个静态方法，其接受一个 Address 类型的参数，其使用方式如下：

```
let entry = Entry::at("0x1001".parse().unwrap());
```

- 外部合约类型实现了 From<Address> trait，因此可以通过显式的类型转换将一个 Address 类型对象转换为外部合约对象。同时，外部合约类型也实现了 Into<Address> trait，因此外部合约类型可以和地址类型相互转换。类型转换的使用方式如下：

```
let addr_1: Address = "0x1001".parse().unwrap();
let entry: Entry = addr_1.into();
let addr_2: Address = entry.into();
assert_eq!(addr_1, addr_2);
```

构造出外部合约对象后，便能够通过成员方法的形式调用外部合约公开方法，如下列代码所示：

```
let entry = Entry::at("0x1001".parse().unwrap());
let i = entry.getInt().unwrap();
```

注意在上述代码中，当声明某个外部合约方法的返回值类型为 T 时，Liquid 会在构建合约时自动将该外部合约方法的返回值变换为 Option<T>。当外部合约方法因为某些原因（如权限等）调用失败时，此时则会返回 None，否则返回包含实际返回值的 Some。基于这一机制，可以根据返回值的内容判断外部合约调用是否成功，从而当外部合约方法调用失败时，继续执行指定的错误处理逻辑。

外部合约中重载方法的调用方式较为特殊，Liquid 会为重载方法生成一个特殊的、与重载方法同名的**成员**（注意不是成员方法）。该成员的类型也经过特殊处理，自动实现了 Fn、FnOnce、FnMut 等 trait。相应地，也需要使用如下的特殊方式调用重载方法：

```
(entry.set)(String::from("id"), id.clone());
(entry.set)(String::from("item_price"), item_price);
(entry.set)(String::from("item_name"), item_name);
```

注意到上述代码中，entry.set 的两边都是用括号 () 括起。若不使用该方式调用外部合约的重载方法，例如去掉 entry.set 两边的括号，则会导致编译时报错如下：

```
error[E0599]: no method named `set` found for struct `entry::__liquid_private::Entry` in␣
↪the current scope
  --> $DIR/13-interface.rs:94:19
   |
6  | mod entry {
   | --------- method `set` not found for this
...
94 |             entry.set(String::from("id"), id.clone());
   |                   ^^^ field, not a method
   |
help: to call the function stored in `set`, surround the field access with parentheses
   |
94 |             (entry.set)(String::from("id"), id.clone());
   |             ^         ^
```

特别地，在上述示例中，由于 set 是 Entry 类型的一个成员，因而在代码中可以先获取 set 成员的引用，然后再进行调用：

```
let set = &entry.set;
set(String::from("id"), id.clone());
```

（下页继续）

(续上页)

```
set(String::from("item_price"), item_price);
set(String::from("item_name"), item_name);
```

2.10 开发指南

本节将以HelloWorld 合约为例介绍 Liquid 智能合约的开发步骤，将会涵盖智能合约的创建、测试、构建、部署及调用等步骤。

2.10.1 创建

在终端中执行以下命令创建 Liquid 智能合约项目：

```
cargo liquid new contract hello_world
```

注解： `cargo liquid` 是调用命令行工具 `cargo-liquid` 的另一种写法，这种写法使得 `liquid` 看上去似乎是 `cargo` 的子命令。

上述命令将会在当前目录下创建一个名为 hello_world 的智能合约项目，此时会观察到当前目录下新建了一个名为“hello_world”的目录：

```
cd ./hello_world
```

hello_world 目录内的文件结构如下所示：

```
hello_world/
   .gitignore
   .liquid
       abi_gen
           Cargo.toml
           main.rs
   Cargo.toml
   src
      lib.rs
```

其中各文件的功能如下：

- `.gitignore`：隐藏文件，用于告诉版本管理软件Git哪些文件或目录不需要被添加到版本管理中。Liquid 会默认将某些不重要的问题件（如编译过程中生成的临时文件）排除在版本管理之外，如果不需要使用 Git 管理对项目版本进行管理，可以忽略该文件；

- `.liquid/`：隐藏目录，用于实现 Liquid 智能合的内部功能，其中 `abi_gen` 子目录下包含了 ABI 生成器的实现，该目录下的编译配置及代码逻辑是固定的，如果被修改可能会造成无法正常生成 ABI；
- `Cargo.toml`：项目配置清单，主要包括项目信息、外部库依赖、编译配置等，一般而言无需修改该文件，除非有特殊的需求（如引用额外的第三方库、调整优化等级等）；
- `src/lib.rs`：Liquid 智能合约项目根文件，合约代码存放于此文件中。智能合约项目创建完毕后，`lib.rs` 文件中会自动填充部分样板代码，我们可以基于这些样板代码做进一步的开发。

我们将HelloWorld 合约中的代码复制至 `lib.rs` 文件中后，便可进行后续步骤。

2.10.2 测试

在正式部署之前，在本地对智能合约进行详尽的单元测试是一种良好的开发习惯。Liquid 内置了对区块链链上环境的模拟，因此即使不将智能合约部署上链，也能够在本地方便地执行单元测试。在 hello_world 项目根目录下执行以下命令即可执行我们预先编写好的单元测试用例：

```
cargo test
```

注意

上述命令与创建合约项目时的命令有所不同：

1. 命令中并不包含 `liquid` 子命令，因为 Liquid 可以使用标准 cargo 单元测试框架来执行单元测试，因此并不需要调用 `cargo-liquid` 。

命令执行结束后，显示如下内容：

```
running 2 tests
test hello_world::tests::get_works ... ok
test hello_world::tests::set_works ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in␣
↪0.00s

   Doc-tests hello_world

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in␣
↪0.00s
```

从结果中可以看出，所有用例均通过了测试，因此可以有信心认为智能合约中的逻辑实现是正确无误的 。我们接下来将开始着手构建 HelloWorld 智能合约，并把它部署至真正的区块链上。

2.10.3 构建

在 hello_world 项目根目录下执行以下命令即可开始进行构建：

```
cargo liquid build
```

该命令会引导 Rust 语言编译器以 `wasm32-unknown-unknown` 为目标对智能合约代码进行编译，最终生成 Wasm 格式字节码及 ABI。cargo-liquid 会在编译过程中对合约代码做冲突字段分析，并将分析结果放在 abi 文件中，底层根据冲突信息自动并行执行无冲突的合约调用。命令执行完成后，会显示如下形式的内容：

```
:-) Done in 9 seconds, your project is ready now:
Binary: C:/Users/liche/hello_world/target/hello_world.wasm
ABI: C:/Users/liche/hello_world/target/hello_world.abi
```

其中，“Binary:” 后为生成的字节码文件的绝对路径，“ABI:” 后为生成的 ABI 文件的绝对路径。为尽量简化 FISCO BCOS 各语言 SDK 的适配工作，Liquid 采用了与 Solidity ABI 规范兼容的 ABI 格式，HelloWorld 智能合约的 ABI 文件内容如下所示：

```
[
    {
        "inputs": [],
        "type": "constructor"
    },
    {
        "constant": true,
        "inputs": [],
        "name": "get",
        "outputs": [
            {
                "internalType": "string",
                "type": "string"
            }
        ],
        "type": "function"
    },
    {
        "conflictFields": [
            {
                "kind": 0,
                "path": [],
                "read_only": false,
                "slot": 0
```

(下页继续)

(续上页)

```
            }
        ],
        "constant": false,
        "inputs": [
            {
                "internalType": "string",
                "name": "name",
                "type": "string"
            }
        ],
        "name": "set",
        "outputs": [],
        "type": "function"
    }
]
```

提示: 构建过程中会从 GitHub 拉取 Liquid 的相关依赖包，若无法正常访问 GitHub，则请在项目中将 `git = "https://github.com/WeBankBlockchain/liquid"` **全局**替换为 `git = "https://gitee.com/WeBankBlockchain/liquid"` 。

提示: 如果希望构建出能够在国密版 FISCO BCOS 区块链底层平台上运行的智能合约，请在执行构建命令时添加-g 选项，例如：`cargo liquid build -g`。

2.10.4 部署

搭建 FISCO BCOS 区块链

当前，FISCO BCOS 3.0 已经支持 wasm 模式，请按照以下步骤手动搭建 FISCO BCOS 区块链：

1. 根据依赖项说明中的要求安装依赖项；
2. 下载建链工具 build_chain.sh：

```
cd ~ && mkdir -p fisco && cd fisco
curl -#LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v3.0.0-rc1/
↪build_chain.sh && chmod u+x build_chain.sh && chmod u+x build_chain.sh
```

提示: 若无法访问 GitHub，则请执行 `curl -#LO https://osp-1257653870.cos.ap-guangzhou.`

myqcloud.com/FISCO-BCOS/FISCO-BCOS/releases/v3.0.0-rc1/build_chain.sh 命令下载 build_chain.sh。

3. 使用 build_chain.sh 在本地搭建一条单群组 4 节点的 FISCO BCOS 区块链并运行。更多 build_chain.sh 的使用方法可参考其使用文档：

```
bash build_chain.sh -l 127.0.0.1:4 -p 30300,20200 -w
bash nodes/127.0.0.1/start_all.sh
```

配置和使用 console

请参考这里安装依赖，下文是安装 Java 之后的 console 下载和配置步骤。

```
cd ~/fisco && curl -LO https://github.com/FISCO-BCOS/console/releases/download/v3.0.0-rc1/download_console.sh && bash download_console.sh
cp -n console/conf/config-example.toml console/conf/config.toml
cp -r nodes/127.0.0.1/sdk/* console/conf/
cd console && bash start.sh
```

提示： 若无法访问 GitHub，则请执行 `curl -#LO https://gitee.com/FISCO-BCOS/console/releases/download/v3.0.0-rc1/download_console.sh` 命令克隆 console。

将合约部署至区块链

使用 console 提供的 `deploy` 子命令，我们可以将 Hello World 合约构建生成的 Wasm 格式字节码部署至区块链上，`deploy` 子命令的使用说明如下：

```
Usage:
deploy liquid bin abi path parameters...
* bin -- The path of binary file after contract being compiled via cargo-liquid.
* abi -- The path of ABI file after contract being compiled via cargo-liquid.
* path -- The path where the contract will be located at.
* parameters -- Parameters will be passed to constructor when deploying the contract.
```

执行该命令时需要传入**字节码 (wasm) 文件的路径、abi 文件路径、合约部署路径及构造函数的参数**。可以使用以下命令部署 HelloWorld 智能合约。由于合约中的构造函数不接受任何参数，因此无需在部署时提供参数：

```
deploy C:/Users/liche/hello_world/target/hello_world.wasm C:/Users/liche/hello_world/target/hello_world.abi /helloworld
```

部署成功后，返回如下形式的结果，其中包含状态码、合约地址及交易哈希：

```
transaction hash: 0x08d4b696c02b107e7d4fff122f621d1eeefb81e1764d5d74fd5ae07c4b774a54
contract address: /hello_world
currentAccount: 0x0929dcf8268561c573092985a5d2086b03873c40
```

2.10.5 调用

使用 console 提供的 `call` 子命令，我们可以调用已被部署到链上的智能合约，`call` 子命令的使用方式如下：

```
Call a contract by a function and parameters.
Usage:
call path function parameters
* path -- The path where the contract located at, when set to "latest", the path of␣
↪latest contract deployment will be used.
* function -- The function of a contract.
* parameters -- The parameters(splited by a space) of a function.
```

执行该命令时需要传入合约名、合约地址、要调用的合约方法名及传递给该合约方法的参数。以调用 HelloWorld 智能合约中的 `get` 方法为例，可以使用以下命令调用该方法。由于 `get` 方法不接受任何参数，因此无需在调用时提供参数：

```
[group]: /> call /hello_world get
```

调用成功后，返回如下形式结果：

```
----------------------------------------------------------------------------------------
↪----
Return code: 0
description: transaction executed successfully
Return message: Success
----------------------------------------------------------------------------------------
↪----
Return value size:1
Return types: (string)
Return values:(Alice)
----------------------------------------------------------------------------------------
↪----
```

其中 `Return values` 字段中包含了 `get` 方法的返回值。可以看到，`get` 方法返回了字符串“Alice”。

2.11 编译选项

Liquid 项目根目录下的 `Cargo.toml` 配置文件中有一个特殊的名为 `[profile.release]` 的 section，此 section 中用于配置合约的编译及优化选项，其内容如下所示：

```
[profile.release]
panic = "abort"
lto = true
opt-level = "z"
overflow-checks = true
```

其中：

- `panic = "abort"`，当发生 `panic` 时，Rust 程序的默认行为是执行堆栈解退（Stack Unwinding），此时会依次执行各个栈上对象的析构函数以释放资源。此配置项用于更改此默认行为，使得当 `panic` 发生时，合约直接终止且不执行堆栈解退，从而有助于减少字节码的体积。由于合约在虚拟机中执行，当合约执行终止时，所有的资源都将会被宿主环境直接回收，因此无需担心资源泄露的问题；
- `lto = true`，此配置项用于开启链接时优化（Link Time Optimization）。开启 LTO 后链接器将会对整个项目进行分析并进行跨模块优化，有助于减少合约字节码的体积；
- `opt-level = "z"`，此配置项用于指定编译器的优化等级，`z` 级别的优化将会在优化性能的同时专注于缩小字节码的体积；
- `overflow-checks = true`，此配置项用于开启运行时算数溢出检查。开启后，Rust 语言编译器将会项目中每一处执行算数运算的代码后插入溢出检查代码。当运算过程中出现算数溢出时，会直接引发 `panic`。关闭该选项能够获得更快的执行速度和更小的字节码体积，但是也会削弱合约的安全性。

可以根据自身的需求调整这些编译配置项，但是调整之前务必对可能造成的后果做到心知肚明。一般而言，默认的编译选项已经足够应付大部分场景的需求。

2.12 单元测试专用 API

Liquid 的特色功能之一是能够直接在合约中编写单元测试用例并在本地执行测试。但是在单元测试的过程中，除了需要对合约方法的输出、状态变量的内容等进行测试外，有时还需要对区块链的状态进行测试，甚至需要改变区块链状态来观察对合约方法执行流程的影响。为此，Liquid 提供了一组测试专用的 API，使得在本地执行合约单元测试时，能够基于这些 API 获取或改变本地模拟区块链环境中的状态，从而使单元测试的过程更为灵活。

注意

本节所述的 API 仅能够在单元测试用例中使用，请不要在合约方法中使用这些 API，否则会引发编译错误。

2.12.1 使用方式

使用单元测试专用 API 之前，首先需要导入位于 `liquid_lang:env` 模块中的 `test` 子模块，所有的测试专用 API 的实现均位于 `test` 子模块中：

```
#[cfg(test)]
mod tests {
    use super::*;
    use liquid::env::test;

    #[test]
    fn foo() {
        let events = test::get_events();
        ...
    }
    ...
```

2.12.2 API 列表

```
pub fn set_caller(caller: Address)
```

将参数中的账户地址压入合约调用栈的栈顶，通过该 API 可以设置合约的调用者，即能够影响环境访问器的 `get_caller` 方法的返回值。使用完毕后需要配合 `push_execution_context` 方法将合约调用者还原。

```
pub fn set_caller_callee(caller: Address, callee: Address)
```

// FIXME: 添加函数的相关说明

```
pub fn pop_execution_context()
```

将合约调用栈栈顶的环境信息弹出。

```
pub fn default_accounts() -> DefaultAccounts
```

合约的测试过程中需要经常使用一些虚拟的账户地址，该 API 可以返回一组固定的账户地址常量，从而免去每次手工创建账户地址的麻烦，并使得单元测试用例拥有更好的可读性。返回值中 `DefaultAccounts` 类型的定义如下：

```
pub struct DefaultAccounts {
    pub alice: Address,
    pub bob: Address,
    pub charlie: Address,
```

(下页继续)

(续上页)

```
    pub david: Address,
    pub eva: Address,
    pub frank: Address,
}
```

可以通过如下方式使用这些虚拟地址：

```
let accounts = test::default_accounts();
let alice = accounts.alice;
```

```
pub fn get_events() -> Vec<Event>
```

单元测试开始执行后，本地模拟区块链环境中会维护一个事件记录器。每当合约方法触发事件时，事件记录器中便会增加一条事件记录，可以通过该 API 获取这些事件记录以测试合约方法是否触发了正确的事件。返回值中表示事件的 `Event` 类型的定义是：

```
pub struct Event {
    pub data: Vec<u8>,
    pub topics: Vec<Hash>,
}
```

其中，`data` 为经过编码后的事件数据，可以通过调用 `Event` 类型的 `decode_data` 方法对数据进行解码，`decode_data` 方法的签名为：

```
pub fn decode_data<R>(&self) -> R
```

其中泛型参数 `R` 是事件定义中各个非索引字段的类型所组成的元组类型，可以使用如下方式调用该方法：

```
#[liquid(event)]
struct foo {
    x: u128,
    y: bool,
}
...
let event = test::get_events()[0];
let (x, y) = event.decode_data<(u128, bool)>();
```

除 `data` 外，`Event` 类型中还包括一个由事件索引组成的数组成员 `topics`，每个索引的类型为 `Hash`。`Hash` 类型内部是一个长度为 32 的字节数组，能够方便与字节数组、字符串互相转换，其提供的方法与地址类型类似。

2.13 外部合约 Mock

当合约中包含外部合约声明时，由于合约的所有单元测试均在本地执行，因此执行单元测试时，Liquid 无法获知实际外部合约的具体实现逻辑。为能够对包含有外部合约调用的合约进行单元测试，Liquid 使用模拟对象机制对外部合约进行模拟，从而使得即使不将合约部署至链上，也可以对包含有外部合约调用的合约进行测试。具体而言，可以为外部合约方法设置**期望**，并通过期望指定在测试时，外部合约方法应当以何种方式进行工作。

为了能够给某一外部合约方法设置期望，首先需要获取该合约方法的模拟上下文。在测试合约时，Liquid 会通过外部合约类型为每一个外部合约方法生成一个静态的模拟上下文获取方法。模拟上下文获取方法的名称形如 `<method_name>_context`，其中 `<method_name>` 为对应外部合约方法的名称。模拟上下文获取方法不接受任何参数。通过调用模拟上下文获取方法便可以获得对应外部合约方法的模拟上下文，例如：

```
#[liquid::interface(name = auto)]
mod kv_table {
    use super::entry::*;

    extern "liquid" {
        fn get(&self, primary_key: String) -> (bool, Entry);
        fn set(&mut self, primary_key: String, entry: Entry) -> u8;
        ...
    }
}

#[test]
fn get_works() {
    let get_ctx = KvTable::get_context();
}
```

随后，通过调用模拟上下文的 `expect` 方法，便能够创建一个对该外部合约方法的期望，随后能够通过期望指定该合约方法的工作方式，例如：

```
#[test]
fn get_works() {
    let get_ctx = KvTable::get_context();
    get_ctx
        .expect()
        .when(predicate::eq(String::from("cat")))
        .returns((true, Entry::at(Default::default())));
}
```

上述代码示例中，`when` 方法的语义是“若参数满足…条件时…”，其参数个数与外部合约方法参数数量相同，且每个参数都是一个关于对应外部合约方法参数的谓词，用于判断调用外部合约方法时传入的参数是否满足

谓词的要求。returns 方法的语义是“返回一个固定值”，其参数为期望外部合约方法返回的固定值，且要求该固定值的类型与该外部合约方法的返回值类型一致。综上所述，我们对 get 方法创建的期望是：调用该方法时，若第一个参数为"cat"，则返回的固定值 (true, Entry::at(Default::default()))。

除了 when 方法，还可以使用 when_fn 方法，其作用与 when 方法类似，只是其参数为一个闭包，闭包的参数数量及类型与外部合约方法一致，且在闭包中能够实现更加复杂的谓词逻辑。类似地，除 returns 方法外，还可以使用 returns_fn 方法，其参数同样为一个闭包，且闭包的参数数量及类型、返回值数量及类型与外部合约方法一致。上述示例也可以改写为如下等价的形式：

```
#[test]
fn get_works() {
    let get_ctx = KvTable::get_context();
    get_ctx
        .expect()
        .when_fn(|key| key == String::from("cat"))
        .returns_fn(|_| (true, Entry::at(Default::default())));
}
```

在创建期望后可以不调用 when 或 when_fn 方法，此时表示的语义时“对于任意参数…”。例如在下面的示例中，创建的期望为：对于任意的参数，get 方法均返回 (true, Entry::at(Default::default))：

```
#[test]
fn get_works() {
    let get_ctx = KvTable::get_context();
    get_ctx
        .expect()
        .returns_fn(|_| (true, Entry::at(Default::default())));
}
```

类似地，在创建期望后也可以不调用 returns 或 returns_fn 方法，此时表示的语义是“返回一个默认值”。这种使用方式需要外部合约方法返回值的类型实现了Default trait。例如在下面的示例中，创建的期望为：当第一个参数等于"cat" 时，set 方法返回 u8 类型的默认值，即 0：

```
#[test]
fn set_works() {
    let set_ctx = KvTable::set_context();
    set_ctx
        .expect()
        .when_fn(|key| key == String::from("cat"))
}
```

甚至，可以既不调用 when 或 when_fn 方法，也不调用 returns 或 returns_fn 方法，此时表示的语义为“对任意参数，均返回默认值”。例如在下面的示例中，创建的期望为：对于任意的参数，set 方法均回 u8 类

型的默认值，即 0：

```
#[test]
fn set_works() {
    let set_ctx = KvTable::set_context();
    set_ctx.expect();
}
```

除了 returns 或 returns_fn 方法外，还可以调用 throws 方法，用于模拟外部合约方法调用失败时的场景。例如在下面的例子中，创建的期望为：当参数等于"dog" 时，get 方法调用失败：

```
#[test]
fn get_works() {
    let get_ctx = KvTable::get_context();
    get_ctx
        .expect()
        .when_fn(|primary_key| primary_key == "dog")
        .throws();
}
```

当外部合约中存在重载函数时，需要在调用 expect 方法时传入类型参数以指定为哪一个重载方法创建期望。类型参数为一个元组，元组中依次排列对应重载方法的全部参数类型。除此之外，使用方式与普通合约方法一致，如下列代码所示：

```
let entry_set_ctx = Entry::set_context();
entry_set_ctx
    .expect::<(String, String)>();
```

每当执行完一个单元测试用例，所有外部合约方法的期望均会被清空，因此不同单元测试用例之间的期望互不影响。

可以为同一个外部合约方法创建多个期望。当执行单元测试用例时，会按照先入先出的顺序使用期望中的参数谓词对参数进行匹配，并执行第一个匹配成功的期望所指定的行为。若没有任何期望与参数成功匹配，则会引发 panic，其提示信息如下所示：

```
thread 'kv_table_test::tests::set_works' panicked at 'no matched expectation is found␣
↪for `getString(&self, key: String)` in `Entry`'
```

在少部分情况下，外部合约声明中可能恰好包含一个与模拟上下文获取方法同名的外部合约方法，如下列代码所示：

```
#[liquid::interface(name = auto)]
mod foo {
```

（下页继续）

(续上页)

```
    extern {
        fn foo();
        // Oops, what a coincidence...
        fn foo_context();
    }
}
```

此时若 Liquid 再生成一个同名的 `foo_context` 方法，则会导致编译器报告重复定义的错误。为避免这种情况发生，Liquid 允许为外部合约方法标注名为 `#[liquid(mock_context_getter)]` 的属性，其参数为一个字符串常量，用于告知 Liquid 在为该合约方法生成模拟上下文获取方法时，使用属性中指定的方法名。基于这一机制，上述示例可以改写为如下形式：

```
#[liquid::interface(name = auto)]
mod foo {
    extern {
        fn foo();
        // The compiler will be happy.
        #[liquid(mock_context_getter = "liquid_is_fun")]
        fn foo_context();
    }
}
```

此时，若需要在单元测试用例中获取 `foo_context` 方法的模拟上下文，则可以通过调用 `liquid_is_fun` 函数：

```
let foo_ctx = Foo::liquid_is_fun();
```

2.14 基于宏的元编程

为理解 Liquid 的实现原理，我们需要简单了解元编程与宏的概念。在维基百科中，元编程被描述成一种计算机程序可以将代码看待成数据的能力，使用元编程技术编写的程序能够像普通程序在运行时更新、替换变量那样操作更新、替换代码。宏在 Rust 语言中是一种功能，能够在编译实际代码之前按照自定义的规则展开原始代码，从而能够达到修改原始代码的目的。从元编程的角度理解，宏就是“生成代码的代码”，因而 Rust 语言中的元编程能力主要来自于宏系统。通过 Rust 语言的宏系统，不仅能够实现 C/C++ 语言的宏系统所提供的模式替换功能，甚至还能够控制编译器的行为、设计自己的语法从而实现 eDSL，而 Liquid 正是基于 Rust 语言的宏系统实现的 eDSL。我们接下来将对 Liquid 的工作机制进行简要介绍。

Rust 源代码文件编译需要经过下列阶段（图中省略了优化等步骤，因为它们并不涉及我们所讨论的主题）：

1. 编译器在获得源代码文件后，会先进行词法分析，即把源代码字符序列转换为标记（Token）序列。标记是单独的语法单元，在 Rust 语言中，关键字、标识符都能够构成标记。词法分析还会将标记与标记

的关系也记录下来，从而的生成标记树（Token tree），以一条简单的程序语句为例：

```
a + b + (c + d[0])  + e
```

其标记树如下图所示：

注意

与 C/C++ 中宏处理（导入 `#include` 头文件、替换 `#define` 符号等）是发生在预编译阶段不同，Rust 语言并没有预编译阶段，其宏展开是发生在的完成语法分析后。也正是因为如此，Rust 宏能够获得更详细、更复杂的编译期信息，从而提供极为强大的元编程能力。

---

2. 随即，编译器启动语法分析流程，将词法分析生成的标记树翻译为 AST（Abstract Syntax Tree，抽象语法树）。在计算机科学中，AST 是源代码语法结构的一种抽象表示，能够方便地被编译器处理。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。上述第 1 步中生成的样例标记树会被翻译为如下图所示的 AST：

1. 然后，编译器开始分析 AST 并执行宏展开过程。此阶段是是最为重要的阶段，因为 Liquid 主要工作在这个阶段。以HelloWorld 合约为例，编译器构造出 HelloWorld 合约的 AST 后，当扫描至 AST 中表示 `#[liquid::contract]"` 语句的语法树节点时，编译器能够知道，此处正在调用属性宏（Rust 中一种特殊的宏），因此会开始寻找 `contract` 属性宏的定义并尝试进行展开。在 Liquid 中，`contract` 属性宏的定义如下：

```rust
#[proc_macro_attribute]
pub fn contract(attr: TokenStream, item: TokenStream) -> TokenStream {
    contract::generate(attr.into(), item.into()).into()
}
```

属性宏以函数形式定义，其输入是两个标记序列（TokenStream），其输出也是一个标记序列。事实上，在 Rust 语言中，宏可以理解为将某一个 AST 变换到另外一个 AST 的函数。Rust 编译器并不会向属性宏直接传递 AST，而且会将其调用位置所在的语法树节点转换为标记序列传递给属性宏，由属性宏的编写者自行决定如何处理这段标记序列。无论如何处理，属性宏都需要返回一段标记序列，Rust 编译器接收到这段标记序列后，会将其重新编译为 AST 并插入到宏的调用位置，从而完成代码的编译期修改。具体到 Liquid 的 `contract` 属性宏，当编译器进行展开时，`contract` 属性宏会获取到自身及其后跟随的 `mod` 代码块（即我们用来定义合约状态及合约方法的模块）的标记序列，并将其解析为一棵 AST。随后，`contract` 属性宏会自顶向下扫描这棵 AST，当遇到使用 `#[liquid(storage)]` 属性标注的 `struct` 代码块时，会进行语法检查及代码变换，将对结构体成员的读写操作变换为对区块链链上状态读写接口的调用。同理，当合约代码中出现 `#[liquid(methods)]` 属性标注的 `impl` 代码块时，也会经历相似的代码变换过程，只是变换及桥接到区块链底层平台的方式不尽相同。

2. 编译器将经过宏展开之后的 AST 编译为可执行文件：若是需要在本地运行单元测试，则会将 AST 编译为本地操作系统及 CPU 所能识别的可执行文件；若是需要能够在链上部署运行，则会将 AST 编译为 Wasm 格式字节码。至此，合约的基本构建流程结束。

从上述实现原理中可以看出，Liquid 可以理解为是一种以 Rust 语言目标语言的编程语言。在编译器的广义定义中，编译器是一种能够将以某种编程语言（原始语言）书写的源代码转换成另一种编程语言（目标语言）的计算机程序，因此 Liquid 在一定程度上扮演了编译器的角色。通过屏蔽区块链的底层实现细节，智能合约的开发过程能够更加便利及自然。HelloWorld 合约的完全展开形态已放置于Rust Playground，供有兴趣的读者参考学习。

2.15 Liquid 架构设计

Liquid 及周边开发工具的整体架构如下图所示：

在整体架构中，`cargo-liquid` 是面向开发者的命令行辅助工具，帮助开发者创建及构建 Liquid 项目。在项目创建阶段，`cargo-liquid` 能够根据用户选定的项目类型根据模板自动配置编译选项及外部依赖，并生成 ABI 生成器等辅助代码；在项目构建阶段，`cargo-liquid` 负责收集编译元信息并进行跨平台构建，将 Liquid 项目编译为 Wasm 格式字节码。基本构建完成后，`cargo-liquid` 还会使用 Tree-Shaking 算法及 wasm-opt 等工具对生成的字节码进行效率和体积上的进一步优化。

Lang 组件主要包括开发者在合约开发过程中所使用到的 `contract` 过程宏（用于以 `mod` 语法声明智能合约）、`InOut` 派生宏（用于以 `struct` 语法定义结构体参数类型）等，这些宏均由 **macro** 模块定义并导出。当构建 Liquid 项目时，Rust 语言编译器会对这些宏进行模式匹配并展开。在宏的展开过程中，IR 模块会解析开发者的代码并重新生成 AST，以对部分 Rust 语法进行重新诠释。随后，**code-gen** 模块会依据 IR 模块生成的 AST 生成调用 Core 模块中的区块链底层接口封装，展开后的代码对开发者完全透明。

Core 组件包含了开发者能够使用的区块链底层功能的实现。以自底向上的视角来看，**engine** 模块是 Liquid 智能合约的执行引擎，为合约运行提供了最为坚实的基础。对于上层，**engine** 模块提供了一系列基础 API，包括用于读取链上存储的 `get_storage` 接口、用于写入链上存储的 `set_storage` 接口、用于获取当前区块时间戳的 `now` 接口等。对于这些接口，**engine** 有两种版本的实现：**off-chain** 版本用于在本机执行智能合约的单元测试时使用，其内部模拟了区块链特性（键值对存储、事件记录器等）并提供了测试专用的接口，用于开发者在正式部署合约前测试合约逻辑是否正确；**on-chain** 版本用于智能合约在真正地区块链环境中执行时使用，其实现相对较为简单，因为具体实现是由区块链底层平台完成，**on-chain** 中只负责对这些接口进行声明并适配即可。

区块链底层接口的规范（名称、参数类型、返回值类型等）由区块链底层平台给出，对于 FISCO BCOS，这个规范称为 FISCO BCOS 环境接口规范（FISCO BCOS Environment Interface，FBEI）。理论上，只要接口规范确定且底层能够提供对应的支持，Liquid 也能够对接其他区块链平台，从而做到“一处编译，处处运行”。

Core 组件中的 **types** 模块提供了智能合约中基本数据类型的定义，如地址（`Address`）、字符串（`String`）等。**types** 模块与 **engine** 模块一同构成了智能合约的执行环境，即 **env** 模块。**storage** 模块基于 **env** 模块提供接口，对链上状态的访问方式进行了进一步的抽象。智能合约需要通过 **storage** 模块提供的容器类型读写链上状态。若要访问简单合约状态，则可以使用常规容器 `Value`；若要以下标的形式序列式地访问合约状态，则可以使用向量容器 `Vec`；若要以键值对的形式访问合约状态，则可以使用映射容器 `Mapping`；若需要在 `Mapping` 的基础上根据键对合约状态进行迭代访问，则可以使用可迭代映射容器 `IterableMapping`。

Utils 组件则涵盖了其他基础功能。主要包括用于实现合约方法参数及返回值编解码的 **abi-codec** 模块——

此模块是 Liquid 与 Solidity 合约进行通信的关键——以及用于生成 ABI 的 **abi-gen** 模块及用于内存分配的 **alloc**。其中，**alloc** 模块用于为合约注册为全局内存分配器，合约内所有的内存分配操作（动态数组、字符串等）都会通过 **alloc** 模块进行。

2.16 FISCO BCOS 环境接口规范

FISCO BCOS 环境接口（FISCO BCOS Environment Interface，FBEI）规范中包含区块链底层平台FISCO BCOS向 Wasm 虚拟机公开的应用程序接口（Application Programming Interface，API）。FBEI 规范中所有的 API 均由 FISCO BCOS 负责实现，运行于 Wasm 虚拟机中的程序能够直接访问这些 API 以获取区块链的环境及状态。

2.16.1 数据类型

在 FBEI 规范中，API 参数及返回值的数据类型会使用 `i32`、`i32ptr` 及 `i64` 三种类型标记，其定义如下：

类型标记	定义
i32	32 位整数，与 Wasm 中 i32 类型的定义一致
i32ptr	32 位整数，其存储方式与 Wasm 中 i32 类型一致，但是用于表示虚拟机中的内存偏移量
i64	64 位整数，与 Wasm 中 i64 类型的定义一致

2.16.2 API 列表

setStorage

描述

将键值对数据写入至区块链底层存储中以实现持久化存储。使用时需要先将表示键及值的字节序列存储在虚拟机内存中。

参数

参数名	类型	描述
keyOffset	i32ptr	键在虚拟机内存中的存储位置的起始地址
keyLength	i32	键的长度
valueOffset	i32ptr	值在虚拟机内存中的存储位置的起始地址
valueLength	i32	值的长度

返回值

无。

注解： 调用 setStorage 时，若提供的 valueLength 参数为 0，则表示从区块链底层存储中删除键所对应的数据。在这种情况下，API 的实现将直接跳过值的读取，因此 valueOffset 参数不用赋予有效值，一般直接置为 0 即可。

getStorage

描述

根据所提供的键，将区块链底层存储中对应的值读取至虚拟机内存中。使用时需要先将表示键的字节序列存储在虚拟机内存中，并提前分配好存储值的内存区域。

参数

参数名	类型	描述
keyOffset	i32ptr	键在虚拟机内存中的存储位置的起始地址
keyLength	i32	键的长度
valueOffset	i32ptr	用于存放值的虚拟机内存起始地址

返回值

类型	描述
i32	值的长度

getCallData

描述

将当前交易的输入数据拷贝至虚拟机内存中，使用时需要提前分配好存储交易输入数据的内存区域。

参数

参数名	类型	描述
resultOffset	i32ptr	用于存放当前交易输入数据的虚拟机内存起始地址

返回值

无。

getCallDataSize

描述

获取当前交易输入数据的长度。

参数

无。

返回值

类型	描述
i32	当前交易输入数据的长度

getCaller

描述

获取发起合约调用的调用方地址，使用时需要提前分配好存储调用方地址的内存区域。

参数

参数名	类型	描述
resultOffset	i32ptr	用于存放调用方地址的虚拟机内存起始地址

返回值

无。

finish

描述

将表示返回值的字节序列传递至宿主环境并结束执行流程，宿主环境会将该其作为交易回执的一部分返回至调用方。

参数

参数名	类型	描述
dataOffset	i32ptr	用于存放返回值的虚拟机内存起始地址
dataLength	i32	返回值的长度

返回值

无。

revert

描述

将表示异常信息的字节序列抛出至宿主环境，宿主环境会将其作为交易回执的一部分返回至调用者。调用此接口后，交易回执中的状态将会被标记为“已回滚”。

参数

参数名	类型	描述
dataOffset	i32ptr	异常信息在虚拟机内存中的存储位置的起始地址
dataLength	i32	异常信息的长度

返回值

无。

注解： 异常信息需要为人类可读的字符串，以方便快速定位异常原因。

---

log

描述

创建一条交易日志。可以至多为该日志创建 4 个日志索引。使用时需要先将表示日志数据及其索引的字节序列存储在虚拟机内存中。

参数

参数名	类型	描述
dataOffset	i32ptr	日志数据在虚拟机内存中的存储位置的起始地址
dataLength	i32	日志数据的长度
topic1	i32ptr	第 1 个日志索引的虚拟机内存起始地址，没有时置 0
topic2	i32ptr	第 2 个日志索引的虚拟机内存起始地址，没有时置 0
topic3	i32ptr	第 3 个日志索引的虚拟机内存起始地址，没有时置 0
topic4	i32ptr	第 4 个日志索引的虚拟机内存起始地址，没有时置 0

返回值

无。

注解： 日志索引的长度需要为恰好为 32 字节。

---

getTxOrigin

描述

获取调用链中最开始发起合约调用的调用方地址，使用时需要提前分配好存储调用方地址的内存区域。与 `getCaller` 接口不同，本接口获取到的调用方地址一定为外部账户地址。

参数

参数名	类型	描述
resultOffset	i32ptr	用于存放调用方地址的虚拟机内存起始地址

返回值

无。

getBlockNumber

描述

获取当前块高。

参数

无。

返回值

类型	描述
i64	当前块高

getBlockTimestamp

描述

获取当前块的时间戳。

参数

无。

返回值

类型	描述
i64	当前块的时间戳

call

描述

发起外部合约调用，使用时需要先将表示调用参数的字节序列存储在虚拟机内存中。调用此接口后执行流程会陷入阻塞，直至外部合约调用结束或发生异常。

参数

参数名	类型	描述
addressOffset	i32ptr	被调用合约地址在虚拟机内存中的存储位置的起始地址
dataOffset	i32ptr	调用参数在虚拟机内存中的存储位置的起始地址
dataLength	i32	调用参数的长度

返回值

类型	描述
i32	调用状态，0 表示成功，否则表示失败

getReturnDataSize

描述

获取外部合约调用的返回值长度，此接口仅能在外部合约调用成功后调用。

参数

无。

类型	描述
i32	外部合约调用的返回值长度

getReturnData

获取外部合约调用的返回值，使用时需要根据 `getReturnDataSize` 的返回结果提前分配好存储返回值内存区域。

参数

参数名	类型	描述
resultOffset	i32ptr	用于存放返回值的虚拟机内存起始地址

返回值

无。

2.17 FISCO BCOS Wasm 合约接口规范

FISCO BCOS Wasm 合约接口（FISCO BCOS Wasm Contract Interface，FBWCI）规范中包含关于合约文件格式及内容的约定。符合 FBWCI 规范要求合约文件能够在区块链底层平台FISCO BCOS内置的 Wasm 虚拟机中运行。

2.17.1 传输格式

所有的合约件必须以WebAssembly 二进制编码格式保存及传输。

2.17.2 符号导入

合约文件仅能导入在BCOS 环境接口规范中规定的接口，所有的接口都需要从名为 `bcos` 的命名空间中导入，且签名必须与 BCOS 环境接口规范中所声明的接口签名保持一致。除 `bcos` 命令空间外，还有一个名为 `debug` 的特殊命名空间。`debug` 命名空间中所声明的函数的主要用于虚拟机的调试模式，在正式的生产环境中该命名空间不会被启用，详情请参考调试模式。

2.17.3 符号导出

合约文件必须恰好导出下列 3 个符号：

2.17.4 调试模式

调试模式是一种用于调试虚拟机的特殊模式，通过 `debug` 命名空间为合约提供了一组额外调试接口。但是在正式的生产环境中，若合约字节码尝试从 `debug` 命名空间中导入符号，则会被拒绝部署。`debug` 命名空间中可用的接口如下所示，所有接口均没有返回值：

print32

描述

在区块链底层的日志中输出一个 32 位整数值。

参数

参数名	类型	描述
value	i32	32 位整数值

print64

描述

在区块链底层的日志中输出一个 64 位整数值。

参数

参数名	类型	描述
value	i64	64 位整数值

printMem

描述

以可打印字符的形式在区块链底层的日志中输出一段虚拟机内存中的内容。

参数

参数名	类型	描述
offset	i32	内存区域的起始地址
len	i32	内存区域的长度

printMemHex

以 16 进制字符串的形式在区块链底层的日志中输出一段虚拟机内存中的内容。

参数

参数名	类型	描述
offset	i32	内存区域的起始地址
len	i32	内存区域的长度

2.17.5 Start function

Start function 会在虚拟机载入合约字节码时自动执行，而此时宿主环境尚无法获得虚拟机提供的共享内存的访问权限，因而可能会导致引发运行时异常，因此 FBWCI 规范规定合约文件中不允许存在 start function。

2.18 微众银行区块链开源生态

2.18.1 FISCO-BCOS

适用于金融行业的区块链底层平台

git 地址：https://github.com/FISCO-BCOS

gitee 地址：https://gitee.com/FISCO-BCOS

文档地址：https://fisco-bcos-doc.readthedocs.io/zh_CN/latest/

2.18.2 WeIdentity

基于区块链的实体身份标识及可信数据交换解决方案

git 地址：https://github.com/WeBankFinTech/WeIdentity

gitee 地址：https://gitee.com/WeBank/WeIdentity

文档地址：https://weidentity.readthedocs.io/

2.18.3 WeEvent

基于区块链的分布式事件驱动架构

git 地址：https://github.com/WeBankFinTech/WeEvent

gitee 地址：https://gitee.com/WeBank/WeEvent

文档地址：https://weevent.readthedocs.io/

2.18.4 WeBase

区块链中间件平台

git 地址：https://github.com/WeBankFinTech/WeBASE

gitee 地址：https://gitee.com/WeBank/WeBASE

文档地址：https://webasedoc.readthedocs.io/

2.18.5 WeCross

区块链跨链协作平台

git 地址：https://github.com/WeBankBlockchain/WeCross

gitee 地址：https://gitee.com/WeBank/WeCross

文档地址：https://wecross.readthedocs.io/

2.18.6 WeDPR

即时可用，场景式隐私保护高效解决方案套件和服务

git 地址：https://github.com/WeBankBlockchain/WeDPR-Lab-Core

文档地址：https://wedpr-lab.readthedocs.io/

2.18.7 Data-Stash

数据仓库组件

git 地址：https://github.com/WeBankBlockchain/Data-Stash

gitee 地址：https://gitee.com/WeBankBlockchain/Data-Stash

文档地址：https://data-doc.readthedocs.io/zh_CN/stable/docs/WeBankBlockchain-Data-Stash/index.html

2.18.8 Data-Export

数据导出组件

git 地址：https://github.com/WeBankBlockchain/Data-Export

gitee 地址：https://gitee.com/WeBankBlockchain/Data-Export

文档地址：https://data-doc.readthedocs.io/zh_CN/stable/docs/WeBankBlockchain-Data-Export/index.html

2.18.9 Data-Reconcile

数据对账组件

git 地址：https://github.com/WeBankBlockchain/Data-Reconcile

gitee 地址：https://gitee.com/WeBankBlockchain/Data-Reconcile

文档地址：https://data-doc.readthedocs.io/zh_CN/stable/docs/WeBankBlockchain-Data-Reconcile/index.html

2.18.10 Liquid

智能合约编程语言

git 地址：https://github.com/WeBankBlockchain/liquid

gitee 地址：https://gitee.com/WeBankBlockchain/liquid

文档地址：https://liquid-doc.readthedocs.io/

2.18.11 cargo-liquid

Liquid 智能合约辅助开发工具

git 地址：https://github.com/WeBankBlockchain/liquid

gitee 地址：https://gitee.com/WeBankBlockchain/cargo-liquid

2.18.12 Governance-Account

账户治理组件

git 地址：https://github.com/WeBankBlockchain/Governance-Account

gitee 地址：https://gitee.com/WeBankBlockchain/Governance-Account

文档地址：https://governance-doc.readthedocs.io/zh_CN/latest/docs/WeBankBlockchain-Governance-Acct/index.html

2.18.13 Governance-Authority

权限治理组件

git 地址：https://github.com/WeBankBlockchain/Governance-Authority

gitee 地址：https://gitee.com/WeBankBlockchain/Governance-Authority

文档地址：https://governance-doc.readthedocs.io/zh_CN/latest/docs/WeBankBlockchain-Governance-Auth/index.html

2.18.14 Governance-Key

私钥管理组件

git 地址：https://github.com/WeBankBlockchain/Governance-Key

gitee 地址：https://gitee.com/WeBankBlockchain/Governance-Key

文档地址：https://governance-doc.readthedocs.io/zh_CN/latest/docs/WeBankBlockchain-Governance-Key/index.html

2.18.15 Governance-Cert

证书管理组件

git 地址：https://github.com/WeBankBlockchain/Governance-Cert

gitee 地址：https://gitee.com/WeBankBlockchain/Governance-Cert

文档地址：https://governance-doc.readthedocs.io/zh_CN/latest/docs/WeBankBlockchain-Governance-Cert/index.html

2.18.16 Truora

可信预言机服务

git 地址：https://github.com/WeBankBlockchain/Truora

gitee 地址：https://gitee.com/WeBankBlockchain/Truora

文档地址：https://truora.readthedocs.io/

2.18.17 SmartDev-Contract

智能合约库组件

git 地址：https://github.com/WeBankBlockchain/SmartDev-Contract

gitee 地址：https://gitee.com/WeBankBlockchain/SmartDev-Contract

文档地址：https://smartdev-doc.readthedocs.io/zh_CN/latest/docs/WeBankBlockchain-SmartDev-Contract/index.html

2.18.18 SmartDev-SCGP

合约编译插件

git 地址：https://github.com/WeBankBlockchain/SmartDev-SCGP

gitee 地址：https://gitee.com/WeBankBlockchain/SmartDev-SCGP

文档地址：https://smartdev-doc.readthedocs.io/zh_CN/latest/docs/WeBankBlockchain-SmartDev-SCGP/index.html

2.18.19 SmartDev-Scaffold

应用开发脚手架

git 地址：https://github.com/WeBankBlockchain/SmartDev-Scaffold

gitee 地址：https://gitee.com/WeBankBlockchain/SmartDev-Scaffold

文 档 地 址：https://smartdev-doc.readthedocs.io/zh_CN/latest/docs/WeBankBlockchain-SmartDev-Scaffold/index.html