ScaleBit

OpenBuild

Tact语法概述

Jason

大纲

- Tact语法精讲
- 总结
- 作业

Tact基本结构

- 可以从其他合约 import 导入现有合约。
- 新的合约以 contract 关键字开始。
- 可以使用 with 关键字从其他 trait 继承。
- 有私有变量，可存储合约状态。
- 有初始化函数 init 用于初始化合约状态。
- 可以 self 关键字访问当前合约的状态和方法。
- 可定义方法函数。

sources > 1.tact

```
1   // this trait has to be imported
2   import "@stdlib/deploy";
3
4   // The Deployable trait adds a default
5   // receiver for the "Deploy" message.
6   contract Counter with Deployable {
7
8       val: Int as uint32;
9
10      init() {
11          self.val = 0;
12      }
13
14      receive("increment") {
15          self.val = self.val + 1;
16      }
17
18      get fun value(): Int {
19          return self.val;
20      }
21  }
```

原始类型

- Int - Tact 中的所有整数都是 257 位有符号整数。
- Bool - 具有真/假值的经典布尔值。
- Address 标准地址。
- Slice、Cell、Builder - TON VM 的低级原始类型。
- String - 表示 TON VM 中文本字符串的类型。
- StringBuilder - String工具类型，可以高效的方式连接字符串。

```
contract Counter with Deployable {
    val: Int as uint32;
    init() {
        self.val = 0;
    }
    receive("increment") {
        self.val = self.val + 1;
        let value: Int = 123;
        let b: Bool = value == self.val;
        let some: String = value.toString();

        let ctx: Context = context();
        let addr: Address = ctx.sender;
        let body: Cell = "Hello, World!".asComment();
    }
```

结构体和消息

- struct 关键字定义结构体类型。
- message 关键字定义消息类型。
- 结构体和消息几乎是相同的东西，唯一的区别是消息在其序列化中具有标头，因此可以用作接收器。

```
struct Point {
    x: Int;
    y: Int;
}

message SetValue {
    key: Int;
    value: Int?; // Optional
}
```

字典

- 类型 map<k, v> 用于将数据与相应的键关联起来。
- Possible key types: 可能的key类型：
  - Int 整数
  - Address 地址
- Possible value types: 可能的value类型：
  - Int 整数
  - Bool 布尔值
  - Cell Cell（TON特有的数据类型，最多1023bit，以及最多4个其他Cell的引用）
  - Address 地址
  - Struct/Message 结构/消息

```
contract HelloWorld {
  counters: map<Int, Int>;
}
```

Contract 合约

- 合约是 TON 区块链上智能合约的主要入口。它包含合约的所有状态、函数、获取者和接收者。

sources > 1.tact

```tact
// this trait has to be imported
import "@stdlib/deploy";

// The Deployable trait adds a default
// receiver for the "Deploy" message.
contract Counter with Deployable {

    val: Int as uint32;

    init() {
        self.val = 0;
    }

    receive("increment") {
        self.val = self.val + 1;
    }

    get fun value(): Int {
        return self.val;
    }
}
```

Trait 特征

- Tact 不支持经典的类继承，而是引入了trait的概念，合约可以从Trait继承。
- Trait 定义了函数、接收者和必填字段。
- Trait 就像抽象类，但它没有定义字段的存储方式和位置。
  - Trait的所有字段都必须在合约本身中明确声明。
  - Trait本身也没有构造函数，所有初始字段初始化也必须在主合约中完成。

```
trait Ownable {
    owner: Address;

    fun requireOwner() {
        nativeThrowUnless(132, context().sender == self.owner);
    }

    get fun owner(): Address {
        return self.owner;
    }
}
```

```
contract Treasure with Ownable {
  owner: Address; // Field from trait MUST be defined in contract itself

  // Here we init the way we need, trait can't specify how you must init owner field
  init(owner: Address) {
    self.owner = owner;
  }
}
```

运算符

- ! 逻辑反转，仅为 Bool 类型定义。
- / 、* 、% 除法、乘法和取模运算，仅为 Int 类型定义。
- - 、+ 算术运算，仅为 Int 类型定义。
- != 、== 相等运算。
- > 、< 、>= 、<= 比较操作，仅为 Int 类型定义。
- && 、|| 逻辑 AND 和 OR。
- !! 后缀运算符 - 强制执行非空值，仅为可空类型定义。大多数原始类型、结构体和消息都可以为空。
- initOf 计算合约的初始状态。

```
message MsgOpts {
    ma: Int?;
}
receive(msg: MsgOpts) {
    let i: Int = 12;
    if (msg.ma != null) {
        i = i + msg.ma!!; // !! te
        self.ca = i;
    }
}
```

```
let init: StateInit = initOf JettonDefaultWallet(self.master, msg.destination);
```

常量

- 简单常量，它是在编译时定义的值，无法更改。可以在 顶层或contract/trait内定义常量。
- 虚拟常量是可以在trait中定义但在contract中更改的常量。当您需要在 编译时配置某些trait时，它非常有用。
- 抽象常量是可以在trait中定义但不指定值的常量。

```
const MY_CONSTANT: Int = 42;

trait MyTrait {
    const MY_CONSTANT: Int = 42;
}

contract MyContract {
    const MY_CONSTANT: Int = 42;
}
```

```
trait MyTrait {
    virtual const MY_FEE: Int = ton("1.0");
}

trait MyAbstractTrait {
    abstract const MY_DEVFEE: Int;
}
```

```
contract MyContract with MyTrait, MyAbstractTrait {
    override const MY_FEE: Int = ton("0.5");
    override const MY_DEVFEE: Int = ton("1000");
}
```

If 语句

- If
- If/else
- If/else if/else

```
if (condition) {
  doSomething();
}
```

```
if (condition) {
  doSomething();
} else {
  doSomething2();
}
```

```
if (condition) {
  doSomething();
} else if (condition2) {
  doSomething2();
} else {
  doSomething3();
}
```

Repeat 循环

- 重复循环执行一段代码指定的次数。
  - 重复数必须是32位int，否则会抛出超出范围的异常。负值将被忽略。

```
let a: Int = 1;
repeat(10) {
  a = a * a;
}
```

While 循环

- 只要给定条件为真，While 循环就会继续执行代码块。

```
let x: Int = 10;
while(x > 0) {
  x = x - 1;
}
```

Until 循环

- Until 循环是一个测试后循环，它至少执行一次代码块，然后继续执行直到给定条件成立。

```
let x: Int = 10;
do {
  x = x - 1;  # do something no matter at least one time
} until (x <= 0);
```

函数及其类型

Tact 中的函数可以用不同的方式定 义:

- Global static function 全局静态函数
- Extension functions 扩展功能
- Mutable functions 可变函数
- Native functions 原生函数
- Receiver functions 接收器功能
- Getter functions 读取函数

Global static functions 全局静态函数

- 可以在程序中的任何位置定 义全局函数:

```
fun pow(a: Int, c: Int): Int {
  let res: Int = 1;
  repeat(c) {
    res = res * a;
  }
  return res;
}
```

Extension function 扩展函数

- 扩展函数由关键字 extends 修饰，允许对任何类型进行扩展。
  - 第一个参数的名称必须命名为 self，并且此参数的类型是要扩展的类型。

```
extends fun pow(self: Int, c: Int) {
  let res: Int = 1;
  repeat(c) {
    res = res * self;
  }
  return res;
}
```

```
let some: String = 95.toString(); // toString() is a stdlib function
```

Mutable function 可变函数

- 可变函数是指可以改 变self值的扩展函数。

```
extends mutates fun pow(self: Int, c: Int) {
  let res: Int = 1;
  repeat(c) {
    res = res * self;
  }
  self = res;
}
```

Native functions 原生函数

- 原生函数是指直接 绑定到 func的函数。
- 原生函数也可以是可 变和扩展的。

```
@name(store_uint)
native storeUint(s: Builder, value: Int, bits: Int): Builder;

@name(load_int)
extends mutates native loadInt(self: Slice, l: Int): Int;
```

Receiver functions 接收器函数

- 接收器函数是负责接收合约中消息的特殊函数，只能在contract或trait中定义。

```
contract Treasure {

  // ...

  // This means that this contract can receive comment "Increment"
  receive("Increment") {
    self.counter = self.counter + 1;
  }
}
```

Getter Functions 读取函数

- Getter 函数在智能合约上定义 getter，并且只能在contract或trait中定义。

```
contract Treasure {

  // ...

  get fun counter(): Int {
    return self.counter;
  }
}
```

消息类型

- TON 合约之间的通信是通过发送和接收消息来完成的。
- 消息分为两种
  - 内部消息：最常见的消息类型是内部消息 - 从一个合约（或钱包）发送到另一个合约的消息
  - 外部消息：外部消息是与链下系统集成的，一般开发中较少遇到，本教程中不做深入学习。

接收内部消息

一个合约允许定义有多个接收器函数。所有接收器函数都按照下面列出的顺序进行处理：

- receive() - 当向合约发送空消息时调用
- receive("message") - 当带有特定文本的消息发送到合约时调用
- receive(str: String) - 当任意文本消息发送到合约时调用
- receive(msg: MyMessage) - 当 MyMessage 类型的二进制消息发送到合约时调用
- receive(msg: Slice) - 当未知类型的二进制消息发送到合约时调用

```
message MyMessage {
    value: Int;
}

contract MyContract {
    receive() {
        // ...
    }
    receive("message") {
        // ...
    }
    receive(str: String) {
        // ...
    }
    receive(msg: MyMessage) {
        // ...
    }
    receive(msg: Slice) {
        // ...
    }
}
```

消息组成

TON 区块链是一种基于消息的区 块链，可与您需要 发送消息的其他合 约进行通信。
消息本身由如下部分 组成：

- value in TON - 您想要与消息一起 发送的 TON 数量。该值用于支付接收方的 Gas 费。
- bounce - 如果设置为 true（默认），那么 如果接收者合约不存在或无法 处理消息，消息将被退回 给发送者。
- code 和 data - init 包，对于部署很有用。
- body - 消息正文为 Cell。
- mode - 配置如何 发送消息的 8 位标志。

使用reply发送简单消息

最简单的消息是对传入消息的回复，并返回消息的所有多TON代币余值：

```
self.reply("Hello, World!".asComment()); // asComment converts string to a Cell with a comment
```

使用send发送消息

- 如果需要自定义发送消息参数，可以使用 send 函数。
- 如下代码表示向 to 地址发送一条消息，其中 value 为 1 TON，body 为字符串“Hello, World!”的文本。SendIgnoreErrors 意味着即使在消息发送过程中发生错误，下一条消息仍然会被发送。

```
let to: Address = ...;
let value: Int = ton("1");
send(SendParameters{
    to: to,
    value: value,
    mode: SendIgnoreErrors,
    bounce: true,
    body: "Hello, World!".asComment()
});
```

发送自定义结构体消息

要发送二进制类型的消息，您可以使用以下代 码：

```
let to: Address = ...;
let value: Int = ton("1");
send(SendParameters{
    to: to,
    value: value,
    mode: SendIgnoreErrors,
    bounce: true,
    body: SomeMessage{arg1: 123, arg2: 1234}.toCell()
});
```

发送部署合约消息

要部署合约，您需要计算它的地址和初始化包，然后将其与初始消息一起发送。您始终可以发送带有消息的 init 包，它会被忽略，但比没有 init 包的消息花费更多。

```
let init: StateInit = initOf SecondContract(arg1, arg2);
let address: Address = contractAddress(init);
let value: Int = ton("1");
send(SendParameters{
    to: address,
    value: value,
    mode: SendIgnoreErrors,
    bounce: true,
    code: init.code,
    data: init.data,
    body: "Hello, World!".asComment()
});
```

处理bounced消息

- 当合约发送一条消息，并将反弹标志(bounce)设置为true时，如果该消息未正确处理，它将反弹回发送者。发送者可以做一些补救操作，比如回滚。
- bounced消息有些限制，比如不支持文本消息，且消息大小目前不超过224bits。
- bounced消息处理函数定义。

```
contract MyContract {
    bounced(src: bounced<MyMessage>) {
        // ...
    }
}
```

```
contract MyContract {
    bounced(src: Slice) {
        // ...
    }
}
```

内置函数

主要有Common、Strings、Random、Math、Cells、Builders、Slices，下面是一些示例

- fun sender(): Address;
- fun require(condition: Bool, error: String);
- fun now(): Int
- fun myBalance(): Int;
- fun myAddress(): Address;
- fun ton(value: String): Int;
- fun context(): Context;

Refer https://docs.tact-lang.org/language/ref/common

标准库

标准库也封装了一些traits，合约直接继承可用

Refer https://docs.tact-lang.org/language/libs/deploy

```
import "@stdlib/deploy";

contract ExampleContract with Deployable {
    // ...
}
```

```
import "@stdlib/ownable";

contract ExampleContract with Ownable {

    owner: Address;

    init(owner: Address) {
        self.owner = owner;
    }
}
```

```
import "@stdlib/stoppable";

contract MyContract with Stoppable {

    owner: Address;
    stopped: Bool;

    init(owner: Address) {
        self.owner = owner;
        self.stopped = false;
    }
}
```

总结

- Tact语言比较简洁，类似于JavaScript的语法，能够快速上手。
- 专为TON区块链设计，可编译为FunC语言。

作业

从这里下载代码 https://tact-by-example.org/03-receivers，完成如下作业：
1 增加 Multiply/Divide 消息，并实现对应的接收处理方法；
2 通过上一课的ts工程，将合约部署到链上，并向合约随机发送加减乘除指令，在区块浏览器查看交易信息。

Thanks

Contact us:

- Twitter: @scalebit_
- Email: contact@scalebit.xyz

ScaleBit

More information : www.scalebit.xyz