PowerPoint

NEAR中NFT开发实践分享

There are moments in life when you miss someone so much that you just want to pick them from your dreams and hug them for real! Dream

web3games.org

目录

CONTENTS

1

NEP171 全解析

NEP71全解

NEP171 是在near开发NFT项目必须要搞明白的NFT标准。

首先Web3Games选择Rust在Near上进行开发的原因：
https://docs.near.org/docs/concepts/data-storage#vector

其次在Near上的NFT标准链接
https://github.com/near/NEPs/tree/master/specs/Standards/NonFungibleToken

整体结构

NEP171 标准 目前有几个部分组成
主要包含协议概念，统一的方法名，以及规定统一的数据结构体
1-ApprovalManagement 批准管理模块
2-Core 核心代码实现
3-Enumeration 枚举查询模块
4 Metadata 元数据《信息》结构
5-Payout 拓展版税支付相关

下面则是学习示例和将具体实现的代码放到了标准库里作为统一依赖

代码示例：
https://github.com/near/near-sdk-rs/tree/master/examples/non-fungible-token

实现标准库：
https://github.com/near/near-sdk-rs/tree/master/near-contract-standards/src/non_fungible_token

整体介绍完有一个初步认识后，接下来就是局部细节的讲解。

```
use near_contract_standards::non_fungible_token::metadata::{
    NFTContractMetadata, NonFungibleTokenMetadataProvider, TokenMetadata, NFT_METADATA_SPEC,
};
use near_contract_standards::non_fungible_token::{Token, TokenId};
use near_contract_standards::non_fungible_token::NonFungibleToken;
use near_sdk::borsh::{self, BorshDeserialize, BorshSerialize};
use near_sdk::collections::LazyOption;
use near_sdk::json_types::ValidAccountId;
use near_sdk::{
    env, near_bindgen, AccountId, BorshStorageKey, PanicOnDefault, Promise, PromiseOrValue,
};
```

在Near上开发跟正常Rust工程流程相似，首先我们要规定好我们需要引入的依赖。

在实际开发中我们常常会在主文件中放置好所有需要用到的依赖项，只需要在子模块中 使用use crate::*;跟传统RUST开发还是差不多的。

首先，我们需要定义好我们合约数据存储的结构

```rust
#[near_bindgen]
#[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]
pub struct Contract {
    tokens: NonFungibleToken,
    metadata: LazyOption<NFTContractMetadata>,
}

const DATA_IMAGE_SVG_NEAR_ICON: &str = "data:image/svg+xml,%3Csvg xmlns='http://www.w3.org/2000/svg' viewBox='0

#[derive(BorshSerialize, BorshStorageKey)]
enum StorageKey {
    NonFungibleToken,
    Metadata,
    TokenMetadata,
    Enumeration,
    Approval,
}
```

其次，我们需要编写合约初始化部分实现之前设计的合约存储结构

```rust
#[near_bindgen]
impl Contract {
    /// Initializes the contract owned by `owner_id` with
    /// default metadata (for example purposes only).
    #[init]
    pub fn new_default_meta(owner_id: ValidAccountId) -> Self {
        Self::new(
            owner_id,
            NFTContractMetadata {
                spec: NFT_METADATA_SPEC.to_string(),
                name: "Example NEAR non-fungible token".to_string(),
                symbol: "EXAMPLE".to_string(),
                icon: Some(DATA_IMAGE_SVG_NEAR_ICON.to_string()),
                base_uri: None,
                reference: None,
                reference_hash: None,
            },
        )
    }

    #[init]
    pub fn new(owner_id: ValidAccountId, metadata: NFTContractMetadata) -> Self {
        assert!(!env::state_exists(), "Already initialized");
        metadata.assert_valid();
        Self {
            tokens: NonFungibleToken::new(
                StorageKey::NonFungibleToken,
                owner_id,
                Some(StorageKey::TokenMetadata),
                Some(StorageKey::Enumeration),
                Some(StorageKey::Approval),
            ),
            metadata: LazyOption::new(StorageKey::Metadata, Some(&metadata)),
        }
    }
```

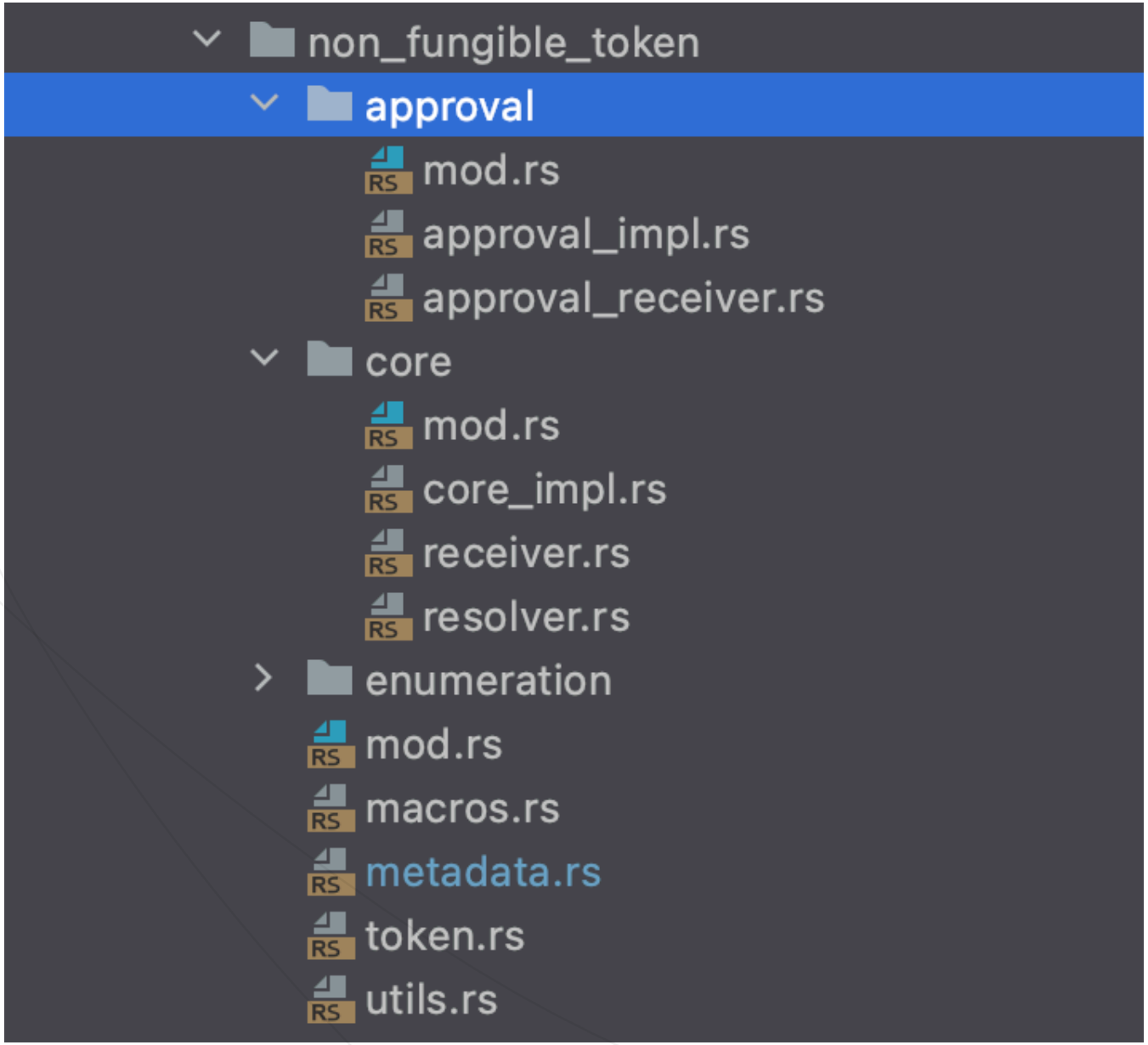
方法规定以及实现部分：

因为大部分的细节被依赖的宏隐藏，所以具体实现我们需要去到标准实现库里翻看公开的方法和实现的流程。

```
    #[payable]
    pub fn nft_mint(
        &mut self,
        token_id: TokenId,
        token_owner_id: ValidAccountId,
        token_metadata: TokenMetadata,
    ) -> Token {
        self.tokens.mint(token_id, token_owner_id, Some(token_metadata))
    }
```

```
near_contract_standards::impl_non_fungible_token_core!(Contract, tokens);
near_contract_standards::impl_non_fungible_token_approval!(Contract, tokens);
near_contract_standards::impl_non_fungible_token_enumeration!(Contract, tokens);
```

```
#[near_bindgen]
impl NonFungibleTokenMetadataProvider for Contract {
    fn nft_metadata(&self) -> NFTContractMetadata { self.metadata.get().unwrap() }
}
```


non_fungible_token
approval
mod.rs
approval_impl.rs
approval_receiver.rs
core
mod.rs
core_impl.rs
receiver.rs
resolver.rs
enumeration
mod.rs
macros.rs
metadata.rs
token.rs
utils.rs

最后合约写完需要编写一下单元测试，测试用例部分。

```rust
#[cfg(all(test, not(target_arch = "wasm32")))]
mod tests {
    use near_sdk::test_utils::{accounts, VMContextBuilder};
    use near_sdk::testing_env;

    use super::*;

    const MINT_STORAGE_COST: u128 = 5870000000000000000000;

    fn get_context(predecessor_account_id: ValidAccountId) -> VMContextBuilder {
        let mut builder = VMContextBuilder::new();
        builder
            .current_account_id(accounts(0))
            .signer_account_id(predecessor_account_id.clone())
            .predecessor_account_id(predecessor_account_id);
        builder
    }

    fn sample_token_metadata() -> TokenMetadata {
        TokenMetadata {
            title: Some("Olympus Mons".into()),
            description: Some("The tallest mountain in the charted solar system".into()),
            media: None,
            media_hash: None,
            copies: Some(1u64),
            issued_at: None,
            expires_at: None,
            starts_at: None,
            updated_at: None,
            extra: None,
            reference: None,
            reference_hash: None,
        }
    }
```

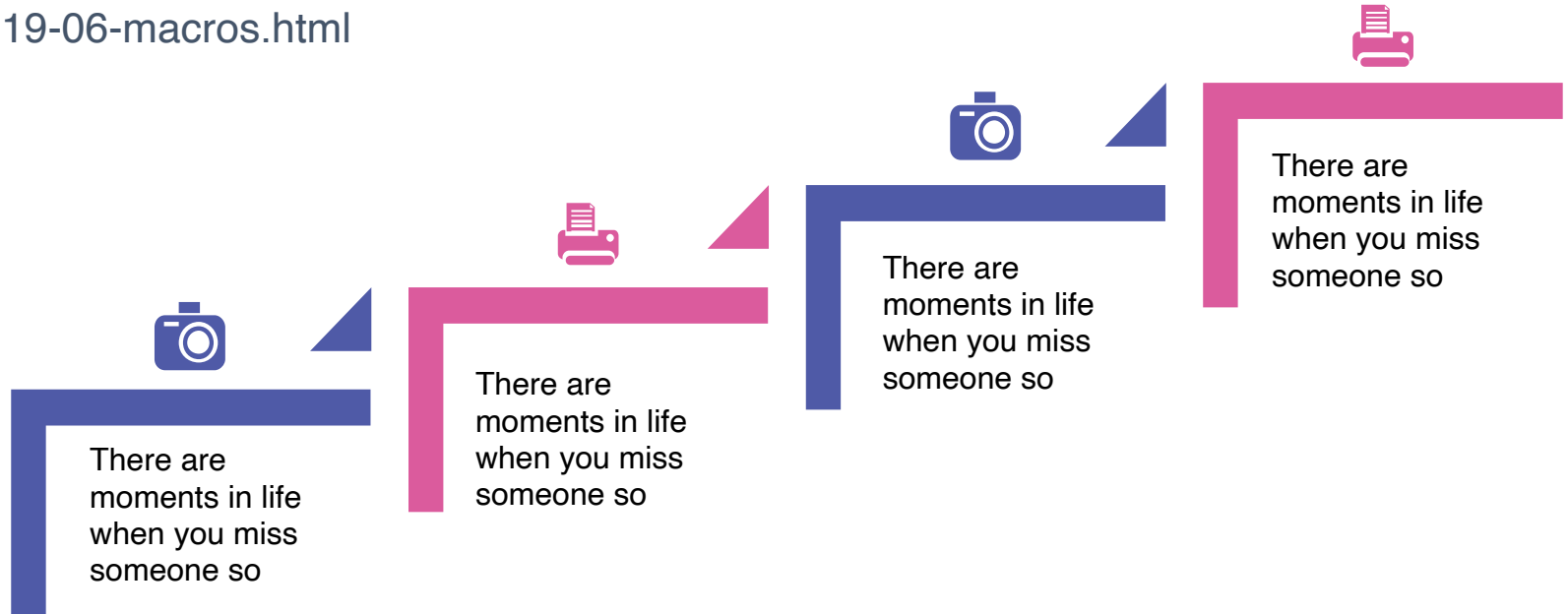
下来我们具体深入到RUST宏《元编程部分》和方法实现部分

c/c++的include头文件展开也是在预编译阶段。但Rust宏的展开不是发生在预编译期，而是编译期。于是Rust能够获得更复杂的更详细的编译器的信息。这里有两个概念Token Tree和AST。

由于是中文课程加上RUST开发以及宏这一部分可能比较麻烦 所以大家对RUST和宏不太了解的同学可以看如下连接

https://rust-lang.budshome.com

https://kaisery.github.io/trpl-zh-cn/ch19-06-macros.html



我们可以看到，在标准库中 很多的细节都放在了 声明宏中macro_rules! 的声明宏用于通用元编程

https://github.com/web3gamesofficial/web3games-near-nftdrop

```rust
/// The core methods for a basic non-fungible token. Extension standards may be
/// added in addition to this macro.
#[macro_export]
macro_rules! impl_non_fungible_token_core {
    ($contract: ident, $token: ident) => {
        use std::collections::HashMap;
        use $crate::non_fungible_token::core::NonFungibleTokenCore;
        use $crate::non_fungible_token::core::NonFungibleTokenResolver;

        #[near_bindgen]
        impl NonFungibleTokenCore for $contract {
            #[payable]
            fn nft_transfer(
                &mut self,
                receiver_id: ValidAccountId,
                token_id: TokenId,
                approval_id: Option<u64>,
                memo: Option<String>,
            ) {
                self.$token.nft_transfer(receiver_id, token_id, approval_id, memo)
            }

            #[payable]
            fn nft_transfer_call(
                &mut self,
                receiver_id: ValidAccountId,
                token_id: TokenId,
                approval_id: Option<u64>,
                memo: Option<String>,
                msg: String,
            ) -> PromiseOrValue<bool> {
                self.$token.nft_transfer_call(receiver_id, token_id, approval_id, memo, msg)
            }

            fn nft_token(self, token_id: TokenId) -> Option<Token> {
                self.$token.nft_token(token_id)
            }

            fn mint(
                &mut self,
                token_id: TokenId,
                token_owner_id: ValidAccountId,
                token_metadata: Option<TokenMetadata>,
            ) -> Token {
```

2

关于淘宝造物节和黄河山先生的NFT艺术品领取展示平台实战

首先这个小产品开发上有以下两部分组成

1-前端部分 **【Next.JS】**
2-合约部分 **【Rust+DSL】**

没有用到后端作为链上数据索引服务 纯前端直接通过RPC封装对应语言实现库-如Near_JS_API 和节点交互。

合约部分：

Link Drop

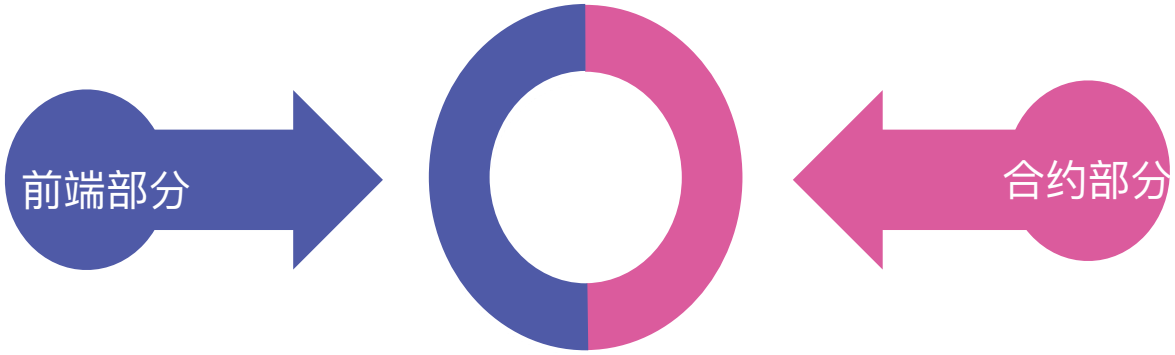
https://github.com/near/near-linkdrop

NFT Drop

https://github.com/web3gamesofficial/web3games-near-nftdrop

NFT

https://github.com/web3gamesofficial/web3games-near-nft



这三个合约组成流程很简单

1- 首先用户通过NFT合约Mint NFT

2- 生成1个NFT就对应生成一对公私钥

3-（在这之前需要先调用linkdrop send将 你想空投的near数量和公钥信息）再将你想要发送的NFT 通过NFT合约 transfer到NftDrop上，并在msg参数内携带对应NFT生成时对应生成的公钥信息.。

4-最后前端功能脚本需要在玩家点击领取时触发构建低级交易，签名和接受者地址都是nftdrop本身，合约账户自己调自己，签名的私钥哪里来呢？ 通过带有私钥的URL连接通过获取用户URL参数，获取私钥。

5-用户没有账号就可以通过这个小产品linkdrop创建账户并且连锁nftdrop领取NFT

有账户的直接可以通过nftdrop领取NFT，并且获得linkdrop的空投Near

前端部分UI就不在这里叙述了，主要讲一下脚本（下方是代码链接）

https://github.com/web3gamesofficial/web3games-near-scripts/tree/main/nftdrop

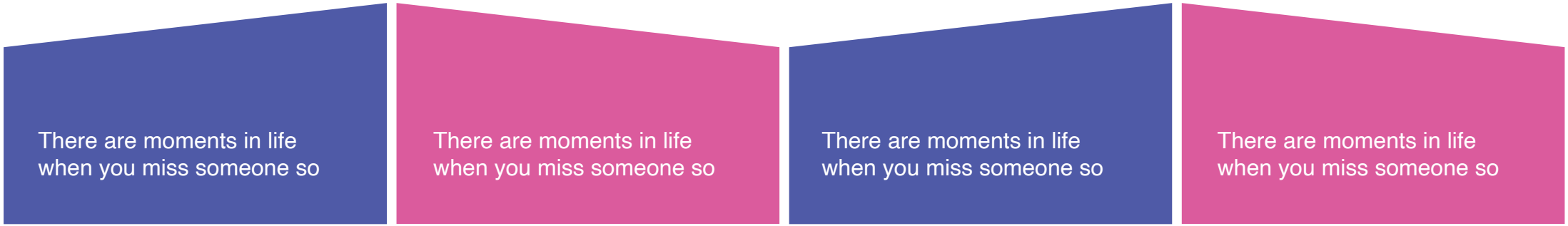
这里有一些官方例子

https://docs.near.org/docs/api/naj-quick-reference

3

NFT Market places実战

NFT Market places实战



有了前面的基础，我想NFT这块应该大家有一个相对的了解了，最后NFT 大型产品 NFT Markte places 我们可以了解一下

Examples

https://github.com/near-apps/nft-market/tree/main/contracts

4

合约测试及注意事项

跨合约调用示例

https://github.com/near/near-sdk-rs/tree/master/near-sdk-sim

https://github.com/near/near-sdk-rs/tree/master/examples/fungible-token

Js脚本使用注意事项

- 当你通过JS调用合约使其改变链上状态时，一般来说单笔交易很难突破上限300TG0.03Near。
  但是如果你在需要在合约的Promise中拿到结果再发Promise再发这种回调层级的加深交易的gas是成倍增加的 所以有时间精力能力的情况下可以通过沙盒和Sim库多编写一些测试用例跑一下大概就知道需要消耗的Gas数量，根据生产环境的经验来讲差距不大。

- index数据的部分
  如果用node+express KOA nest 之类的搭建后端索引服务
  https://github.com/near/nearcore/blob/master/tools/indexer/example/src/main.rs
  对应的Js实现 步骤和model和数据库部分

- https://docs.near.org/docs/api/rpc/block-chunk
  https://docs.near.org/docs/api/rpc/transactions
  即可拿到你想要的 历史信息 并且 按照你的model格式【想要的编排模型】塞进数据库里
  https://docs.near.org/docs/api/rpc （如果历史数据太久远最好准备一个港外服务器 去读历史归档节点）

2021

感谢您的观看

There are moments in life when you miss someone so much that you just want to pick them from your dreams and hug them for real! Dream

web3games.org