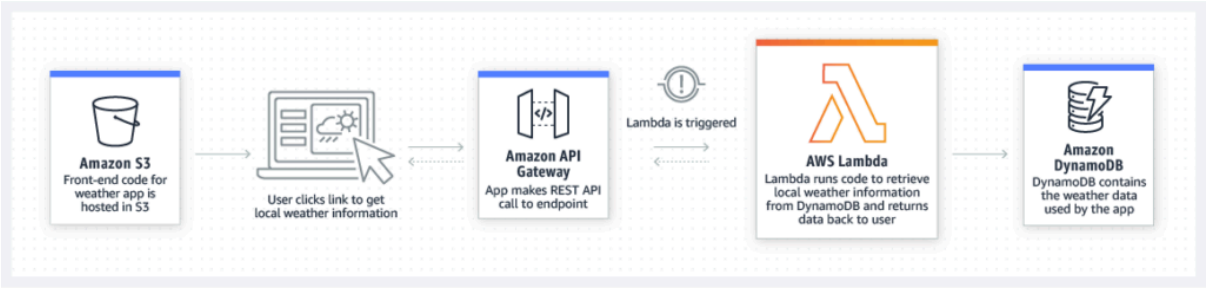
Desenvolvimento de um contrato inteligente no Near usando Rust 2ª parte

birchmd.near (20 nL)

11 min read

2 nL 4 nL 8 nL Tip



Tweet +4nLEARNs

Esta é a segunda parte de uma série de postagens sobre como criar um aplicativo de bate-papo com Rust na blockchain Near. Você pode encontrar o primeiro post da série aqui.

Neste post, vamos nos concentrar no contrato inteligente em si. Veremos a biblioteca near-sdk que faz nosso código funcionar no Near. Também veremos os padrões de acesso do estado próximo e os princípios de desenvolvimento de contratos inteligentes em ação, revisando o código deste contrato inteligente. Você pode encontrar o repositório completo com todo o código que discutiremos hoje no meu GitHub.

SDK Rust de contrato inteligente da Near

Em seu núcleo, ao tempo de execução do contrato inteligente no Near é usado WebAssembly (Wasm). Wasm é um formato de bytecode bem estabelecido que também é usado fora da blockchain, como em aplicativos da web. Isso é bom para a Near porque seu tempo de execução pode se beneficiar do trabalho que é feito na comunidade Wasm mais ampla.

O compilador Rust faz um bom trabalho ao gerar a saída Wasm, mas precisa haver algum andaime em torno dele para que o bytecode Wasm funcione corretamente com seu "host" (o tempo de execução do Near em nosso caso, ou o mecanismo JavaScript de um navegador da web no caso de um aplicativo da web). Este andaime pode ser gerado automaticamente usando bibliotecas Rust convenientes: wasm-bindgen no caso de integração do navegador, e near-sdk no caso do Near. O contrato inteligente com o qual estamos trabalhando hoje é escrito usando near-sdk.

Ambas bibliotecas usam Rust procedural macros (proc macros). Este é um tipo de metaprogramming (meta-desenvolvimento) onde a biblioteca define pequenas anotações que podemos usar para acionar o código Rust para ser gerado automaticamente para nós. As proc macros do Rust são usadas para reduzir a quantidade de boilerplate code (códigos clichês) que o desenvolvedor precisa escrever para fazer sua lógica de negócios funcionar. Por exemplo, o derive proc macro é o núcleo da linguagem Rust. Ele pode definir automaticamente a funcionalidade comum em novos tipos de dados que você criar. Você pode ver isso usado no simples fragmento de código seguinte do contrato inteligente de bate-papo:

```
#[derive(
Debug, BorshDeserialize, BorshSerialize, Serialize,
Deserialize, Clone, Copy, PartialEq, Eq,
```

```
)]
#[serde(crate = "near_sdk::serde")]
pub enum MessageStatus {
Read,
Unread,
}
```

Você pode ver muitas características listadas na anotação do `derive`. Para chamar alguns específicos: `Debug` significa o tipo de `MessageStatus` que pode ser convertido em uma string para ajudar na depuração do código; `Clone` significa que é possível criar uma instância idêntica à do atual `MessageStatus`, e `Copy` significa que a operaçao `Clone` é barata; `PartialEq` e `Eq` significa que você pode comparar duas instâncias de MessageStatus para ver se são iguais. Os traços `Serialize` e o `Deserialize` vêm da biblioteca serde, que é onipresente no ecossistema Rust para codificar/decodificar dados de formatos como JSON ou CBOR. Voltaremos aos traços de Borsh mais tarde.

Até agora, tudo isso foi um Rust padrão que você encontrará em qualquer projeto. A proc macro, especificamente Near, é a `near_bindgen` que você pode ver sendo usado no seguinte fragmento de código:

```
#[near_bindgen]
#[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]
pub struct MessengerContract {
accounts: LookupMap<AccountId, AccountStatus>,
messages: LookupMap<MessageId, Message>,
unread_messages: UnorderedSet<MessageId>,
read_messages: UnorderedSet<MessageId>,
last_received_message: LookupMap<AccountId, MessageId>,
pending_contacts: UnorderedSet<AccountId>,
owner: AccountId,
}
```

A `near_bindgen` proc macro gera automaticamente o código extra que precisamos para que, quando compilarmos para o Wasm, obtenhamos uma saída que o tempo de execução da Near saiba como usar. Ele é usado em vários locais onde esse glue code (código cola) é necessário. Aqui ele marca a estrutura `MessengerContract` como tendo o estado necessário para executar os métodos do contrato. Uma instância da estrutura `MessengerContract` será criada cada vez que chamarmos um método em nosso contrato inteligente. Mais tarde discutiremos para quê são usados alguns desses campos.

A macro `near_bindgen` também é usada sobre o bloco impl para a estrutura `MessengerContract`:

```
#[near_bindgen]
impl MessengerContract {
// …
}
```

Aqui significa que as funções definidas neste bloco são os métodos que queremos expor em nosso contrato inteligente. Ele permite que os usuários da blockchain Near enviem transações chamando essas funções pelo nome. Por exemplo, o método para enviar mensagem está definido neste bloco. Veremos alguns outros métodos deste bloco com mais detalhes abaixo.

Em resumo, a biblioteca rust da near-sdk fornece uma macro proc chamada `near_bindgen` para gerar automaticamente o glue code que faz a saída Wasm funcionar com o tempo de execução Near. Essa macro pode ser usada em uma estrutura para definir o estado do seu contrato e no bloco impl dessa estrutura para definir os métodos públicos em seu contrato.Near-sdk também fornece outras funções e estruturas úteis, que veremos nas seções subsequentes.

Estado do contrato inteligente

Essencialmente, todos os contratos inteligentes não triviais requerem algum estado para operar corretamente. Por exemplo, um contrato de token precisa manter os saldos dos vários detentores de token. Nosso contrato de chat não é diferente. Vimos na seção anterior que a estrutura do `MessengerContract` continha muitos campos. Nesta seção, discutimos alguns recursos gerais do estado no tempo de execução da Near, bem como alguns detalhes de como ela é usada no exemplo de contrato inteligente.

A coisa mais importante a saber sobre o estado do contrato inteligente no Near é que ele é um armazenamento de valor-chave simples. Você pode ver isso nas funções storage_read e storage_write de baixo nível expostas por near-sdk. No entanto, você pode criar algumas estruturas de dados mais sofisticadas sobre essa base simples, e o near-sdk fornece algumas delas em seu collections module. Por esse motivo, nosso contrato de exemplo não usa o armazenamento de valor-chave diretamente; em vez disso, ele faz uso das coleções de nível superior oferecidas pelo near-sdk.

Por exemplo, o contrato inteligente acompanha o status das contas que conhece (quais são contratos, para quem enviamos uma solicitação de contato, etc.). O campo de contas do `MessengerContract` é uma estrutura `LookupMap` da Near-sdk. Isso é muito próximo ao uso direto do armazenamento de valor-chave, pois o mapa também é simplesmente uma maneira de procurar um valor de uma chave, mas o `LookupMap` faz duas coisas importantes acima da interface bruta de armazenamento de valor-chave. Primeiro, ele tem um prefixo que inclui todas as chaves de armazenamento relacionadas a este mapa. O uso de um prefixo evita a mistura de chaves deste mapa com chaves de outro (por exemplo, o mapa `last_received_message` que também é codificado em `AccountId`). Segundo, o `LookupMap` nos permite trabalhar com tipos Rust de nível superior, enquanto a interface de armazenamento bruto funciona apenas com bytes. Isso é obtido ao usar a serialização Borsh Isso é obtido usando a serialização Borsh para converter os tipos de/para strings

binárias. Borsh é um formato de serialização projetado pela Near para ser útil especificamente em aplicativos blockchain. Esse uso de Borsh é o motivo pelo qual você vê `BorshDeserialize` e `BorshSerialize` derivados de muitos tipos em todo o código.

Um exemplo mais interessante de uma coleção usada aqui é o `UnorderedSet` usado nocampo `unread_messages`. Isso é usado pelo contrato para rastrear quais mensagens ainda não foram lidas. O `UnorderedSet` ainda é construído no armazenamento de valor-chave subjacente, mas efetivamente usa apenas as chaves, pois só nos importamos que um elemento esteja no conjunto ou não. A estrutura também mantém metadados sobre quais chaves ela está usando para nos permitir iterar sobre todas as chaves do conjunto.

Verificando o ambiente e chamando outros contratos

Nesta seção, discutimos os recursos gerais do ambiente de tempo de execução do Near e as chamadas entre contratos. Para nos manter fundamentados, isso é feito no contexto de como os usuários adicionam uns aos outros como contatos em nosso aplicativo de bate-papo. Vamos dar uma olhada no `add_contact` function definition (essa definição está no bloco impl do `MessengerContact`, com a anotaçao `near_bindgen` antes mencionada, porque é um ponto de entrada principal para nosso contrato).

```
#[payable]
pub fn add_contact(&mut self, account: AccountId) -> Promise
{
self.require_owner_only();
let deposit = env::attached_deposit();
require!(deposit >= ADD_CONTACT_DEPOSIT, "Insufficient
deposit");
```

```
let this = env::current_account_id();
Self::ext(account.clone())
.with_attached_deposit(deposit)
.ext_add_contact()
.then(Self::ext(this).add_contact_callback(account))
}
```

Há muito o que descompactar nessas poucas linhas de código. Como enquadramento adicional para orientar nossa discussão, lembre-se dos três princípios do desenvolvimento de contratos inteligentes descritos no postagem anterior:

1. Uma mentalidade adversária,
2. Economia,
3. Garantir invariáveis antes de fazer chamadas entre contratos.

Volte e revise a primeira postagem se precisar de uma atualização sobre o que eram esses princípios. Cada um desses princípios aparecem nessa função.

Uma mentalidade adversária

Todos os métodos de contrato inteligente são públicos e devemos aplicar o controle de acesso quando o método faz uma ação delicada, sensível, caso contrário, alguém fará uso indevido da funcionalidade. Nesse caso, não queremos que ninguém possa adicionar contatos em nome do proprietário; apenas o proprietário deve poder decidir com quem se conectar (se alguém quiser fazer contatos na rede de bate-papo, pode implantar este contrato em sua própria conta!). Por tal motivo temos a função de chamada `require_owner_only()` bem no topo do corpo da função. A implementação dessa função é bem simples:

```
fn require_owner_only(&self) -> AccountId {
let predecessor_account = env::predecessor_account_id();
require!(
self.owner == predecessor_account,
“Only the owner can use this method!”
);
predecessor_account
}
```

Ela faz uso da função `predecessor_account_id` do env module da Near-sdk. O módulo env contém muitas funções úteis para consultar aspectos do ambiente de tempo de execução Near em que nosso contrato está sendo executado. Por exemplo, aqui estamos verificando qual conta fez a chamada para o nosso contrato. O módulo env contém outras funções úteis, como verificar a ID da conta do nosso próprio contrato e quantos tokens Near foram anexados a esta chamada. Recomendo a leitura da documentação do módulo para ver todas as funções disponíveis.

Por razões de eficácia a função `require_owner_only` também retorna a conta predecessora (para evitar múltiplas chamadas para o `env::predecessor_account_id()` caso uma função exclusiva do proprietário também precise da conta predecessora por algum outro motivo).

Economia

Na primeira linha do fragmento de código `add_contact` acima inclui o atributo a pagar `(payable)`. O uso dessa anotação é ativado pela função que está sendo definida como parte de um bloco impl da `near_bindgen`. Isso significa que este método aceitará tokens Near dos usuários que o chamarem. Esses tokens são necessários porque tomamos a decisão de

que os usuários estão pagando por ações como a criação de estado on-chain (na cadeia). Uma vez que adicionar outra conta como contato cria um estado em seu contrato, assim como no nosso (precisamos avisá-los de que queremos nos conectar), devemos garantir que o usuário que inicia essa conexão esteja pagando por esse armazenamento. O depósito associado a essa função pagável é usado para cobrir esse custo de armazenamento.

Você pode ver algumas linhas abaixo onde verificamos se o depósito está realmente presente. Isso faz uso da função `attached_deposit` do módulo env. O fato de estarmos fazendo essa verificação antecipadamente segue perfeitamente o terceiro princípio.

Certifique-se de invariantes antes de fazer chamadas entre contratos

É importante por atenção a assinatura de tipo da função add_contact. Primeiro, os argumentos da função `(&mut self, account: AccountId)` significa que esta é uma chamada mutável (vai mudar o estado do contrato) e leva um argumento chamado “account” que deve ser um Near Account ID. Quando `near_bindgen` faz sua magia, isso significará que os usuários do blockchain Near podem chamar essa função fazendo uma transação que leva um argumento codificado em JSON como `{ "account": "my.account.near" }`. Segundo, o tipo de retorno é `Promise`, o que significa que estamos fazendo uma chamada entre contratos no final desta função. As chamadas entre contratos no Near são assíncronas e non-atomics (não pode ser efetuada entre blockchains separadas), portanto, devemos garantir que tudo esteja correto antes de fazermos a chamada. É por isso que incluímos primeiro o "somente o proprietário" e a "verificação de depósito" no corpo da função. A natureza assíncrona das chamadas entre contratos também significa que não há valor de retorno imediato dessa função A chamada assíncrona será realizada e o resultado só virá mais tarde, depois que essa chamada acontecer.

Você pode ver os detalhes da chamada entre contratos na parte inferior da função. Ela usa a API de alto nível de near-sdk (embora a API de baixo nível

também esteja disponível no módulo env), onde a função `ext` é gerada automaticamente pela `near_bindgen` e retorna uma estrutura de dados para construir a chamada entre contratos. Você pode ver que primeiro usamos `ext(account)` para chamar a conta que queremos adicionar como contato. A chamada inclui nosso depósito via `with_attached_deposit` e está chamando a função `ext_add_contact` (Que é, neste caso, definida no mesmo bloco impl, mas em geral pode ser definida em qualquer lugar). Finalmente, chamamos `then` o que significa incluir uma callback (chamada de volta). A callback é outro `Promise` de por si, então usamos novamente a mesma função `ext`, mas desta vez chamando nossa própria ID de conta. Isso é feito para que nosso contrato saiba qual foi a resposta do contrato que estamos tentando adicionar como contato. Não entrarei aqui, em detalhes sobre as implementações `ext_add_contact` ou `add_contact_callback` (eles apenas manipulam o armazenamento dependendo do status atual da conta), mas encorajo você a lê-los no código-fonte no GitHub caso esteja interessado.

Resumo

Neste post, mergulhamos de cabeça em algo de codificação! Nós vimos como `near_bindgen` é usado para gerar automaticamente o código necessário para executar nosso contrato no tempo de execução do Near, bem como outros recursos do near-sdk para interagir com o armazenamento, o ambiente de tempo de execução e outros contratos. Na próxima postagem, continuaremos nos aprofundando no código, mas mudaremos de assunto para examinar o componente off-chain desse aplicativo. Um contrato inteligente por si só não constitui um dapp, fique atento para ver o porquê!

Se você quiser alguma experiência prática com este código, experimente alguns dos exercícios! Em alguns lugares no código do contrato inteligente, incluí um comentário marcado como `EXERCISE`. Por exemplo, na definição de tipos eu chamo o fato de que um status de usuário `Blocked` (bloqueado) está disponível, mas não há como bloquear alguém atualmente implementado. Adicionar essa funcionalidade para bloquear

outro usuário é um exercício sugerido e muito bom para começar. Todos os exercícios são sugestões de maneiras de estender a funcionalidade do contrato, dando a você a oportunidade de tentar escrever algum código de contrato inteligente por conta própria. Talvez em um futuro post desta série eu discuta algumas soluções para os exercícios.

Se você está gostando desta série de postagens do blog, por favor entre em contato conosco na consultoria Type-Driven. Para nós é um prazer fornecer serviços de desenvolvimento de software para dapps, bem como materiais de treinamento para seus próprios engenheiros.