Collections Nesting

Traditional approach for unique prefixes

Hardcoded prefixes in the constructor using a short one letter prefix that was converted to a vector of bytes. When using nested collection, the prefix must be constructed manually.

use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: collections :: { UnorderedMap ,

UnorderedSet } ; use

near_sdk :: { near_bindgen ,

AccountId } ;

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{ pub accounts :

UnorderedMap < AccountId ,

UnorderedSet < String

, }

impl

Default

for

Contract

{ fn

default ( )

->

Self

{ Self

{ accounts :

UnorderedMap :: new (b"t") , } } }

[near_bindgen]

impl

Contract

{ pub

fn

get_tokens (& self , account_id :

& AccountId)

->

Vec < String

{ let tokens =

self . accounts . get (account_id) . unwrap_or_else (| |

{ // Constructing a unique prefix for a nested UnorderedSet from a concatenation // of a prefix and a hash of the account id. let prefix :

Vec < u8

=

[b"s" . as_slice ( ) , & near_sdk :: env :: sha256_array (account_id . as_bytes ( )) ,] . concat ( ) ; UnorderedSet :: new (prefix) }) ; tokens . to_vec ( ) } }

Generating unique prefixes for persistent collections

Read more about persistent collectionsfrom this documentation or fromthe Rust docs .

Every instance of a persistent collection requires a unique storage prefix. The prefix is used to generate internal keys to store data in persistent storage. These internal keys need to be unique to avoid collisions (including collisions with keySTATE).

When a contract gets complicated, there may be multiple different collections that are not all part of the main structure, but instead part of a sub-structure or nested collections. They all need to have unique prefixes.

We can introduce anenum for tracking storage prefixes and keys. And then use borsh serialization to construct a unique prefix for every collection. It's as efficient as manually constructing them, because with Borsh serialization, an enum only takes one byte.

use

near_sdk :: borsh :: { self ,

BorshDeserialize ,

BorshSerialize } ; use

near_sdk :: collections :: { UnorderedMap ,

UnorderedSet } ; use

near_sdk :: { env , near_bindgen ,

AccountId ,

BorshStorageKey ,

CryptoHash } ;

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize)]

pub

struct

Contract

{ pub accounts :

UnorderedMap < AccountId ,

UnorderedSet < String

, }

impl

Default

for

Contract

{ fn

default ( )

->

Self

{ Self

{ accounts :

UnorderedMap :: new (StorageKeys :: Accounts) , } } }

[derive(BorshStorageKey, BorshSerialize)]

pub

enum

StorageKeys

{ Accounts , SubAccount

{ account_hash :

CryptoHash

} , }

[near_bindgen]

impl

Contract

{ pub

fn

get_tokens (& self , account_id :

& AccountId)

->

Vec < String

{ let tokens =

self . accounts . get (account_id) . unwrap_or_else (| |

{ UnorderedSet :: new (StorageKeys :: SubAccount

{ account_hash :

env :: sha256_array (account_id . as_bytes ( )) , }) }) ; tokens . to_vec ( ) } }

Error prone patterns

By extension of the error-prone patterns to avoid mentioned in the collections section , it is important to keep in mind how these bugs can easily be introduced into a contract when using nested collections.

Some issues for more context:

- https://github.com/near/near-sdk-rs/issues/560
- https://github.com/near/near-sdk-rs/issues/703

The following cases are the most commonly encountered bugs that cannot be restricted at the type level:

use

near_sdk :: borsh :: { self ,

BorshSerialize } ; use

near_sdk :: collections :: { LookupMap ,

UnorderedSet } ; use

near_sdk :: BorshStorageKey ;

[derive(BorshStorageKey, BorshSerialize)]

pub

enum

StorageKey

{ Root , Nested (u8) , }

// Bug 1: Nested collection is removed without clearing it's own state. let

mut root :

LookupMap < u8 ,

UnorderedSet < String

=

LookupMap :: new (StorageKey :: Root) ; let

mut nested =

UnorderedSet :: new (StorageKey :: Nested (1)) ; nested . insert (& "test" . to_string ( )) ; root . insert (& 1 ,

& nested) ;

// Remove inserted collection without clearing it's sub-state. let

mut _removed = root . remove (& 1) . unwrap ( ) ;

// This line would fix the bug: // _removed.clear();

// This collection will now be in an inconsistent state if an empty UnorderedSet is put // in the same entry of root. root . insert (& 1 ,

& UnorderedSet :: new (StorageKey :: Nested (1))) ; let n = root . get (& 1) . unwrap ( ) ; assert! (n . is_empty ( )) ; assert! (n . contains (& "test" . to_string ( ))) ;

// Bug 2 (only relevant for near_sdk::collections, not near_sdk::store): Nested // collection is modified without updating the collection itself in the outer collection. // // This is fixed at the type level in near_sdk::store because the values are modified // in-place and guarded by regular Rust borrow-checker rules. root . insert (& 2 ,

& UnorderedSet :: new (StorageKey :: Nested (2))) ;

let

mut nested = root . get (& 2) . unwrap ( ) ; nested . insert (& "some value" . to_string ( )) ;

// This line would fix the bug: // root.insert(&2, &nested);

let n = root . get (& 2) . unwrap ( ) ; assert! (n . is_empty ( )) ; assert! (n . contains (& "some value" . to_string ( ))) ; Edit this page Last updatedonAug 3, 2023 bycornflower Was this page helpful? Yes No

Previous Collections Next Public Method Types