Contract Mutability

Contract state mutability is handled automatically based on howself is used in the function parameters. Depending on which is used, the#[near_bindgen] macro will generate the respective code to load/deserialize state for any function which usesself and serialize/store state only for when&mut self is used.

The following semantics are consistent for allpublic methods .

Read-Only Functions

To access state immutably, where the existing state is not overwritten at the end of the transaction, you can use&self orself as a parameter. Both of these will generate the same code to load and deserialize the state into the structure and call the function, but the difference is that&self will just pass a reference to this variable into the function whereself will move the variable into the function.

For more information about&self versusself seethis section in the Rust book .

Here are some examples of using each:

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, Default)]

pub

struct

MyContractStructure

{ integer :

u64 , message :

String , }

[near_bindgen]

impl

MyContractStructure

{ pub

fn

get_values (self)

->

(u64 ,

String)

{ (self . integer ,

self . message) } pub

fn

log_state_string (& self)

{ near_sdk :: env :: log (self . message . as_bytes ( )) ; } } There is no simple guideline that works for every case, but here are some core reasons on when to use each:

self (owned value)

Moving the owned value into the function can be useful ifself itself or its fields are moved within the function, as it will remove the need toClone /Copy the data.

Example:

/// View method. More efficient, but can't be reused internally, because it consumes self. pub

fn

get_owner_id (self)

->

AccountId

{ self . owner_id }

&self (immutable reference)

This should be used when the contract state is only read or the function is re-used by other methods which do not haveownership of the variable. This can also be useful if the struct uses a lot of memory, to avoid moving a large amount of data into the function scope rather than just referencing it.

Example:

/// View method. Requires cloning the account id. pub

fn

get_owner_id (& self)

->

AccountId

{ self . owner_id . clone ( ) }

Returning derived data

Some less common cases may intend to use read-only methods to return objects that are derived from modified objects stored in state. Below is a demonstration of this concept:

/// View method that "modifies" state, for code structure or computational /// efficiency reasons. Changes state in-memory, but does NOT save the new /// state. If called internally by a change method, WILL result in updated /// contract state. pub

fn

update_stats (& self , account_id :

AccountId , score :

U64)

->

Account

{ let account =

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

env :: panic_str ("ERR_ACCT_NOT_FOUND")) ; account . total += score ; account }

Mutable Functions

Mutable functions allow for loading the existing state, modifying it, then rewriting the modified state at the end of the function call. This should be used for any transaction which modifies the contract state. Note that the serialized contract data is stored in persistent storage under the keySTATE .

An example of a mutable function is as follows:

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, Default)]

pub

struct

MyContractStructure

{ integer :

u64 , }

[near_bindgen]

impl

MyContractStructure

{ pub

fn

modify_value (& mut

self , new_value :

u64)

{ self . integer = new_value ; } pub

fn

increment_value (& mut

self)

{ self . integer +=

1 ; } }

Pure Functions

These functions do not useself at all, and will not read or write the contract state from storage. Using public pure functions will be very rare but can be useful if returning data embedded in the contract code or executing some static shared logic that doesn't depend on state.

Some examples of pure functions are as follows:

const

SOME_VALUE :

u64

=

8 ;

[near_bindgen]

impl

MyContractStructure

{ pub

fn

log_message (/ *Parameters here* /)

{ near_sdk :: log! ("inside log message") ; } pub

fn

log_u64 (value :

u64)

{ near_sdk :: log! ("{}" , value) ; } pub

fn

return_static_u64 ( )

->

u64

{ SOME_VALUE } }Edit this page Last updatedonAug 24, 2022 byDamián Parrino Was this page helpful? Yes No

Previous Public Method Types Next Private Methods