The Move Book

Option is a type that represents an optional value which may or may not exist. The concept of Option in Move is borrowed from Rust, and it is a very useful primitive in Move. Option is defined in the Standard Library , and is defined as follows:

The 'std::option' module is implicitly imported in every module, so you don't need to add an explicit import.

The Option type is a generic type with an Element type parameter. It contains a single field, vec , which is a vector of Element . The vector can have a length of 0 or 1, representing the absence or presence of a value, respectively.

Note: You might be surprised that Option is a struct containing a vector instead of an enum . This is for historical reasons: Option was added to Move before it had support for enums.

The Option type has two variants: Some and None . The Some variant contains a value, while the None variant represents the absence of a value. The Option type is used to represent the absence of a value in a type-safe way, avoiding the need for empty or undefined values.

To showcase why the Option type is necessary, let's look at an example. Consider an application which takes a user input and stores it in a variable. Some fields are required, and some are optional. For example, a user's middle name is optional. While we could use an empty string to represent the absence of a middle name, it would require extra checks to differentiate between an empty string and a missing middle name. Instead, we can use the Option type to represent the middle name.

In the previous example, the middle_name field is of type Option . This means that the middle_name field can either contain a String value, wrapped in Some, or be explicitly empty, represented by None. Using the Option type makes the optional nature of the field clear, avoiding ambiguity and the need for extra checks to differentiate between an empty string and a missing middle name.

The Option type, along with the std::option module, is implicitly imported in Move. This means you can use the Option type directly without needing a use statement.

To create a value of the Option type, you can use the option::some or option::none methods. Option values also support several operations (borrowing will be discussed in the references chapter):

In Practice

To showcase why the Option type is necessary, let's look at an example. Consider an application which takes a user input and stores it in a variable. Some fields are required, and some are optional. For example, a user's middle name is optional. While we could use an empty string to represent the absence of a middle name, it would require extra checks to differentiate between an empty string and a missing middle name. Instead, we can use the Option type to represent the middle name.

```bash module book::user_registry;

use std::string::String;

/// A struct representing a user record. public struct User has drop { first_name: String, middle_name: Option, last_name: String, }

/// Create a new `User` struct with the given fields. public fun register( first_name: String, middle_name: Option, last_name: String, ): User { User { first_name, middle_name, last_name } } ```

In the previous example, the middle_name field is of type Option . This means that the middle_name field can either contain a String value, wrapped in Some, or be explicitly empty, represented by None. Using the Option type makes the optional nature of the field clear, avoiding ambiguity and the need for extra checks to differentiate between an empty string and a missing middle name.

The Option type, along with the std::option module, is implicitly imported in Move. This means you can use the Option type directly without needing a use statement.

To create a value of the Option type, you can use the option::some or option::none methods. Option values also support several operations (borrowing will be discussed in the references chapter):

```bash //option::somecreates anOption` value with a value. let mut opt = option::some(b"Alice");

// `option::none` creates an `Option` without a value. We need to specify the // type since it can't be inferred from context. let empty : Option = option::none();

// `option.is_some()` returns true if option contains a value. assert!(opt.is_some()); assert!(empty.is_none());

// internal value can be `borrow`ed and `borrow_mut`ed. assert!(opt.borrow() == &b"Alice");

// `option.extract` takes the value out of the option, leaving the option empty. let inner = opt.extract();

// `option.is_none()` returns true if option is None. assert!(opt.is_none()); ```

## Creating and using Option values

The Option type, along with the std::option module, is implicitly imported in Move. This means you can use the Option type directly without needing a use statement.

To create a value of the Option type, you can use the option::some or option::none methods. Option values also support several operations (borrowing will be discussed in the references chapter):

```bash //option::some`creates an`Option` value with a value. let mut opt = option::some(b"Alice");

// `option::none` creates an `Option` without a value. We need to specify the // type since it can't be inferred from context. let empty : Option = option::none();

// `option.is_some()` returns true if option contains a value. assert!(opt.is_some()); assert!(empty.is_none());

// internal value can be `borrow`ed and `borrow_mut`ed. assert!(opt.borrow() == &b"Alice");

// `option.extract` takes the value out of the option, leaving the option empty. let inner = opt.extract();

// `option.is_none()` returns true if option is None. assert!(opt.is_none()); ```

## References
```