The Move Book

Collection types are a fundamental part of any programming language. They are used to store a collection of data, such as a list of items. The vector type has already been covered in the vector section , and in this chapter we will cover the vector-based collection types offered by the Sui Framework .

While we have previously covered the vector type in the vector section , it is worth going over it again in a new context. This time we will cover the usage of the vector type in objects and how it can be used in an application.

VecSet is a collection type that stores a set of unique items. It is similar to a vector , but it does not allow duplicate items. This makes it useful for storing a collection of unique items, such as a list of unique IDs or addresses.

VecSet will fail on attempt to insert an item that already exists in the set.

VecMap is a collection type that stores a map of key-value pairs. It is similar to a VecSet , but it allows you to associate a value with each item in the set. This makes it useful for storing a collection of key-value pairs, such as a list of addresses and their balances, or a list of user IDs and their associated data.

Keys in a VecMap are unique, and each key can only be associated with a single value. If you try to insert a key-value pair with a key that already exists in the map, the old value will be replaced with the new value.

Standard collection types are a great way to store typed data with guaranteed safety and consistency. However, they are limited by the type of data they can store - the type system won't allow you to store a wrong type in a collection; and they're limited in size - by the object size limit. They will work for relatively small-sized sets and lists, but for larger collections you may need to use a different approach.

Another limitations on collection types is inability to compare them. Because the order of insertion is not guaranteed, an attempt to compare a VecSet to another VecSet may not yield the expected results.

This behavior is caught by the linter and will emit a warning: Comparing collections of type 'sui::vec_set::VecSet' may yield unexpected result

In the example above, the comparison will fail because the order of insertion is not guaranteed, and the two VecSet instances may have different orders of elements. And the comparison will fail even if the two VecSet instances contain the same elements.

In the next section we will cover the Wrapper Type Pattern - a design pattern often used with collection types to extend or restrict their behavior.

Vector

While we have previously covered the vector type in the vector section , it is worth going over it again in a new context. This time we will cover the usage of the vector type in objects and how it can be used in an application.

```bash module book::collections_vector;

use std::string::String;

/// The Book that can be sold by a `BookStore` public struct Book has key, store { id: UID, name: String }

/// The BookStore that sells `Book`s public struct BookStore has key, store { id: UID, books: vector } ```

VecSet is a collection type that stores a set of unique items. It is similar to a vector , but it does not allow duplicate items. This makes it useful for storing a collection of unique items, such as a list of unique IDs or addresses.

```bash module book::collections_vec_set;

use sui::vec_set::{Self, VecSet};

public struct App has drop { /// `VecSet` used in the struct definition subscribers: VecSet

*}*

# [test]

*fun vec_set_playground() { let set = vec_set::empty(); // create an empty set let mut set = vec_set::singleton(1); // create a set with a single item*

```
set.insert(2); // add an item to the set
set.insert(3);

assert!(set.contains(&1)); // check if an item is in the set
assert!(set.size() == 3); // get the number of items in the set
assert!(!set.is_empty()); // check if the set is empty

set.remove(&2); // remove an item from the set
```

} ```

VecSet will fail on attempt to insert an item that already exists in the set.

VecMap is a collection type that stores a map of key-value pairs. It is similar to a VecSet , but it allows you to associate a value with each item in the set. This makes it useful for storing a collection of key-value pairs, such as a list of addresses and their balances, or a list of user IDs and their associated data.

Keys in a VecMap are unique, and each key can only be associated with a single value. If you try to insert a key-value pair with a key that already exists in the map, the old value will be replaced with the new value.

*```bash module book::collections_vec_map;*

use std::string::String; use sui::vec_map::{Self, VecMap};

public struct Metadata has drop { name: String, /// `VecMap` used in the struct definition attributes: VecMap }

[test]

fun vec_map_playground() { let mut map = vec_map::empty(); // create an empty map

```
map.insert(2, b"two".to_string()); // add a key-value pair to the map
map.insert(3, b"three".to_string());

assert!(map.contains(&2)); // check if a key is in the map

map.remove(&2); // remove a key-value pair from the map
```

} ```

Standard collection types are a great way to store typed data with guaranteed safety and consistency. However, they are limited by the type of data they can store - the type system won't allow you to store a wrong type in a collection; and they're limited in size - by the object size limit. They will work for relatively small-sized sets and lists, but for larger collections you may need to use a different approach.

Another limitations on collection types is inability to compare them. Because the order of insertion is not guaranteed, an attempt to compare a VecSet to another VecSet may not yield the expected results.

This behavior is caught by the linter and will emit a warning: Comparing collections of type 'sui::vec_set::VecSet' may yield unexpected result

*```bash let mut set1 = vec_set::empty(); set1.insert(1); set1.insert(2);*

let mut set2 = vec_set::empty(); set2.insert(2); set2.insert(1);

assert!(set1 == set2); // aborts! ```

In the example above, the comparison will fail because the order of insertion is not guaranteed, and the two VecSet instances may have different orders of elements. And the comparison will fail even if the two VecSet instances contain the same elements.

In the next section we will cover the [Wrapper Type Pattern]() - a design pattern often used with collection types to extend or restrict their behavior.

VecSet

VecSet is a collection type that stores a set of unique items. It is similar to a vector , but it does not allow duplicate items. This makes it useful for storing a collection of unique items, such as a list of unique IDs or addresses.

*```bash module book::collections_vec_set;*

use sui::vec_set::{Self, VecSet};

public struct App has drop { /// `VecSet` used in the struct definition subscribers: VecSet

}

[test]

fun vec_set_playground() { let set = vec_set::empty(); // create an empty set let mut set = vec_set::singleton(1); // create a set with a single item

```
set.insert(2); // add an item to the set
set.insert(3);

assert!(set.contains(&1)); // check if an item is in the set
assert!(set.size() == 3); // get the number of items in the set
assert!(!set.is_empty()); // check if the set is empty

set.remove(&2); // remove an item from the set
```

} ```

VecSet will fail on attempt to insert an item that already exists in the set.

VecMap is a collection type that stores a map of key-value pairs. It is similar to a VecSet , but it allows you to associate a value with each item in the set. This makes it useful for storing a collection of key-value pairs, such as a list of addresses and their balances, or a list of user IDs and their associated data.

Keys in a VecMap are unique, and each key can only be associated with a single value. If you try to insert a key-value pair with a key that already exists in the map, the old value will be replaced with the new value.

*```bash module book::collections_vec_map;*

use std::string::String; use sui::vec_map::{Self, VecMap};

public struct Metadata has drop { name: String, /// `VecMap` used in the struct definition attributes: VecMap }

[test]

fun vec_map_playground() { let mut map = vec_map::empty(); // create an empty map

```
map.insert(2, b"two".to_string()); // add a key-value pair to the map
map.insert(3, b"three".to_string());

assert!(map.contains(&2)); // check if a key is in the map

map.remove(&2); // remove a key-value pair from the map
```

} ```

Standard collection types are a great way to store typed data with guaranteed safety and consistency. However, they are limited by the type of data they can store - the type system won't allow you to store a wrong type in a collection; and they're limited in size - by the object size limit. They will work for relatively small-sized sets and lists, but for larger collections you may need to use a different approach.

Another limitations on collection types is inability to compare them. Because the order of insertion is not guaranteed, an attempt to compare a VecSet to another VecSet may not yield the expected results.

This behavior is caught by the linter and will emit a warning: Comparing collections of type 'sui::vec_set::VecSet' may yield unexpected result

*```bash let mut set1 = vec_set::empty(); set1.insert(1); set1.insert(2);*

let mut set2 = vec_set::empty(); set2.insert(2); set2.insert(1);

assert!(set1 == set2); // aborts! ```

In the example above, the comparison will fail because the order of insertion is not guaranteed, and the two VecSet instances may have different orders of elements. And the comparison will fail even if the two VecSet instances contain the same elements.

In the next section we will cover the [Wrapper Type Pattern]() - a design pattern often used with collection types to extend or restrict their behavior.

VecMap

VecMap is a collection type that stores a map of key-value pairs. It is similar to a VecSet , but it allows you to associate a value with each item in the set. This makes it useful for storing a collection of key-value pairs, such as a list of addresses and their balances, or a list of user IDs and their associated data.

Keys in a VecMap are unique, and each key can only be associated with a single value. If you try to insert a key-value pair with a key that already exists in the map, the old value will be replaced with the new value.

*```bash module book::collections_vec_map;*

use std::string::String; use sui::vec_map::{Self, VecMap};

public struct Metadata has drop { name: String, /// `VecMap` used in the struct definition attributes: VecMap }

[test]

fun vec_map_playground() { let mut map = vec_map::empty(); // create an empty map

```
map.insert(2, b"two".to_string()); // add a key-value pair to the map
map.insert(3, b"three".to_string());

assert!(map.contains(&2)); // check if a key is in the map

map.remove(&2); // remove a key-value pair from the map
```

} ```

Standard collection types are a great way to store typed data with guaranteed safety and consistency. However, they are limited by the type of data they can store - the type system won't allow you to store a wrong type in a collection; and they're limited in size - by the object size limit. They will work for relatively small-sized sets and lists, but for larger collections you may need to use a different approach.

Another limitations on collection types is inability to compare them. Because the order of insertion is not guaranteed, an attempt to compare a VecSet to another VecSet may not yield the expected results.

This behavior is caught by the linter and will emit a warning: Comparing collections of type 'sui::vec_set::VecSet' may yield unexpected result

*```bash let mut set1 = vec_set::empty(); set1.insert(1); set1.insert(2);*

let mut set2 = vec_set::empty(); set2.insert(2); set2.insert(1);

assert!(set1 == set2); // aborts! ```

In the example above, the comparison will fail because the order of insertion is not guaranteed, and the two VecSet instances may have different orders of elements. And the comparison will fail even if the two VecSet instances contain the same elements.

In the next section we will cover the [Wrapper Type Pattern]() - a design pattern often used with collection types to extend or restrict their behavior.

Limitations

Standard collection types are a great way to store typed data with guaranteed safety and consistency. However, they are limited by the type of data they can store - the type system won't allow you to store a wrong type in a collection; and they're limited in size - by the object size limit. They will work for relatively small-sized sets and lists, but for larger collections you may need to use a different approach.

Another limitations on collection types is inability to compare them. Because the order of insertion is not guaranteed, an attempt to compare a VecSet to another VecSet may not yield the expected results.

This behavior is caught by the linter and will emit a warning: Comparing collections of type 'sui::vec_set::VecSet' may yield unexpected result

```bash let mut set1 = vec_set::empty(); set1.insert(1); set1.insert(2);

let mut set2 = vec_set::empty(); set2.insert(2); set2.insert(1);

assert!(set1 == set2); // aborts! ```

In the example above, the comparison will fail because the order of insertion is not guaranteed, and the two VecSet instances may have different orders of elements. And the comparison will fail even if the two VecSet instances contain the same elements.

In the next section we will cover the [Wrapper Type Pattern]() - a design pattern often used with collection types to extend or restrict their behavior.

## Summary

In the next section we will cover the [Wrapper Type Pattern]() - a design pattern often used with collection types to extend or restrict their behavior.

## Next Steps

In the next section we will cover the [Wrapper Type Pattern]() - a design pattern often used with collection types to extend or restrict their behavior.
```