

Table of Contents

1	Option
2	Usage
2.1	Create
2.2	Extracting values
2.3	Transforming values
2.4	Monadic operations

Option

The Option type is a built-in type in MoonBit that represents an optional value. The type annotation Option[A] can also be written as A?.

It is an enum with two variants: Some(T), which represents a value of type T, and None, representing no value.

Note that some methods of the Option are defined in the core/builtin package.

Usage

Create

You can create an Option value using the Some and None constructors, remember to give proper type annotations.

```
1
2  test {
3    let some : Int? = Some(42)
4    let none : String? = None
5    inspect(some, content="Some(42)")
6    inspect(none, content="None")
7  }
```

Extracting values

You can extract the value from an Option using the match expression (Pattern Matching).

```
1
2  test {
3    let i = Some(42)
4    let j = match i {
5      Some(value) => value
6      None => abort("unreachable")
7    }
8    assert_eq(j, 42)
9  }
```

Or using the unwrap method, which will panic if the result is None and return the value if it is Some.

```
1
2  test {
3    let some : Int? = Some(42)
4    let value = some.unwrap()
5    assert_eq(value, 42)
6  }
```

A safer alternative to `unwrap` is the `or` method, which returns the value if it is `Some`, otherwise, it returns the default value.

```
1
2  test {
3      let none : Int? = None
4      let value = none.unwrap_or(0)
5      assert_eq(value, 0)
6  }
```

There is also the `or_else` method, which returns the value if it is `Some`, otherwise, it returns the result of the provided function.

```
1
2  test {
3      let none : Int? = None
4      let value = none.unwrap_or_else(() => 0)
5      assert_eq(value, 0)
6  }
```

Transforming values

You can transform the value of an `Option` using the `map` method. It applies the provided function to the value if it is `Some`, otherwise, it returns `None`.

```
1
2  test {
3      let some : Int? = Some(42)
4      let new_some = some.map((value : Int) => value + 1)
5      assert_eq(new_some, Some(43))
6  }
```

There is a `filter` method that applies a predicate to the value if it is `Some`, otherwise, it returns `None`.

```
1
2  test {
3      let some : Int? = Some(42)
4      let new_some = some.filter((value : Int) => value > 40)
5      let none = some.filter((value : Int) => value > 50)
6      assert_eq(new_some, Some(42))
7      assert_eq(none, None)
8  }
```

Monadic operations

You can chain multiple operations that return `Option` using the `bind` method, which applies a function to the value if it is `Some`, otherwise, it returns `None`. Different from `map`, the function in argument returns an `Option`.

```
1
2  test {
3    let some : Int? = Some(42)
4    let new_some = some.bind((value : Int) => Some(value + 1))
5    assert_eq(new_some, Some(43))
6  }
```

Sometimes we want to reduce the nested Option values into a single Option, you can use the flatten method to achieve this. It transforms Some(Some(value)) into Some(value), and None otherwise.

```
1
2  test {
3    let some : Int?? = Some(Some(42))
4    let new_some = some.flatten()
5    assert_eq(new_some, Some(42))
6    let none : Int?? = Some(None)
7    let new_none = none.flatten()
8    assert_eq(new_none, None)
9  }
```