MoonBit 月兔编程

现代编程思想

接口

Hongbo Zhang

回顾

MoonBit 月兔编程

- 第六课：定义平衡二叉树
  - 我们定义一个更一般的二叉搜索树，允许存放任意类型的数据

```
enum Tree[T] {
  Empty
  Node(T, Tree[T], Tree[T])
}

// 我们需要一个比较函数来比较值的大小以了解顺序
// 负数表示小于，0表示等于，正数表示大于
fn insert[T](self: Tree[T], value: T, compare: (T, T) -> Int) -> Tree[T]
fn delete[T](self: Tree[T], value: T, compare: (T, T) -> Int) -> Tree[T]
```

- 第八课：定义循环队列
  - 我们需要类型的默认值来初始化数组

```
fn make[T](default: T) -> Queue[T] {
  { array: Array::make(5, default), start: 0, end: 0, length: 0 }
}
```

MoonBit 月兔编程

方法

- 我们注意到一些与类型相关联的函数
  - 类型的比较：`fn T::compare(self: T, other: T) -> Int`
  - 类型的默认值：`fn T::default() -> T`
  - 类型的输出：`fn T::to_string(self: T) -> String`
  - ……
- 我们将这类函数称为**方法**

MoonBit 月兔编程

接口 Trait

- 我们通过接口定义一系列方法的实现需求

```
1. trait Compare {
2.   compare(Self, Self) -> Int // Self代表实现该接口的类型
3. }
4. trait Default {
5.   default() -> Self
6. }
```

- 月兔中的接口是结构化的
  - 无需声明为特定的接口实现方法，类型本身实现方法即可

接口 Trait

- 我们可以在泛型的参数上添加接口的要求
  - 限制参数的类型：`<类型参数> : <接口>`
  - 在函数中使用接口定义的方法：`<类型参数>::<方法名>`

```
1. fn make[T: Default]() -> Queue[T] { // 类型参数T应当满足Default接口
2.   {
3.     array: Array::make(5, T::default()), // 我们可以利用接口中的方法，返回类型为Self，即T
4.     start: 0,  end: 0,  length: 0
5.   }
6. }
```

- 接口可以尽早发现使用不存在方法的错误

接口 Trait

```
fn insert[T : Compare](tree : Tree[T], value : T) -> Tree[T] {
  // 类型参数T应当满足比较接口
  match tree {
    Empty => Node(value, Empty, Empty)
    Node(v, left, right) =>
      if T::compare(value, v) == 0 { // 可以使用比较方法
        tree
      } else if T::compare(value, v) < 0 { // 可以使用比较方法
        Node(v, insert(left, value), right)
      } else {
        Node(v, left, insert(right, value))
      }
  }
}
```

方法的定义

- 方法定义以 `<类型>::` 为函数名称起始

```
1. struct BoxedInt { value : Int }
2.
3. fn BoxedInt::default() -> BoxedInt { // 定义方法即实现Default接口
4.   { value : Int::default() } // 使用整数的默认值 0
5. }
6.
7. fn init {
8.   let array: Queue[BoxedInt] = make()
9. }
```

链式调用

- 月兔允许利用 `<变量>.<方法>()` 的形式调用方法
  - 方法的第一个参数为该类型的数值

```
fn BoxedInt::plus_one(b: BoxedInt) -> BoxedInt {
  { value : b.value + 1 }
}
fn plus_two(self: BoxedInt) -> BoxedInt { // 参数名称为self时可省略 <类型>::
  { value : self.value + 2}
}

fn init {
  let _five = { value: 1 }.plus_one().plus_one().plus_two()
  // 无需进行深层嵌套，方便理解
  let _five = plus_two(plus_one(plus_one({value: 1})))
}
```

派生定义

- 简单的接口可以自动生成，在定义最后声明 `derive(<接口>)` 即可

```
1. struct BoxedInt { value : Int } derive(Default, Eq, Compare, Debug)
```

- 需要数据结构内部的数据同样实现接口

MoonBit 月兔编程

表：利用接口实现

- 一个表是键值对的集合
  - 对于每一个 **键** 存在一个对应 **值**
  - 例：`{ 0 -> "a", 5 -> "Hello", 7 -> "a"}`

```
type Map[Key, Value]

// 创建表
fn make[Key, Value]() -> Map[Key, Value]
// 添加键值对，或更新键对应值
fn put[Key, Value](map: Map[Key, Value], key: Key, value: Value) -> Map[Key, Value]
// 获取键对应值
fn get[Key, Value](map: Map[Key, Value], key: Key) -> Option[Value]
```

MoonBit 月兔编程

表：利用接口实现

- 表的简易实现
  - 利用列表+二元组存储键值对
  - 添加/更新时向列表前添加键值对
  - 查询时从列表前开始，找到键即返回
- 简易实现需要判断存储的键值对是否为搜索的键
  - 键应当满足相等接口

```
1. fn get[Key: Eq, Value](map: Map[Key, Value], key: Key) -> Option[Value]
```

表：利用接口实现

- 我们以列表+二元组作为表

```
// 我们定义一个类型Map，其实际值为List[(Key, Value)]
type Map[Key, Value] List[(Key, Value)]

fn make[Key, Value]() -> Map[Key, Value] {
  Map(Nil)
}

fn put[Key, Value](map: Map[Key, Value], key: Key, value: Value) -> Map[Key, Value] {
  let Map(original_map) = map
  Map( Cons( (key, value), original_map ) )
}
```

表：利用接口实现

- 我们以列表+二元组作为表

```
fn get[Key: Eq, Value](map : Map[Key, Value], key : Key) -> Option[Value] {
  fn aux(list : List[(Key, Value)]) -> Option[Value] {
    match list {
      Nil => None
      Cons((k, v), tl) => if k == key { // Key实现了Eq接口，因此可以利用==比较
        Some(v)
      } else {
        aux(tl)
      }
    }
  }

  aux(map.0) // 利用 .0 取出实际的值
}
```

MoonBit 月兔编程

自定义运算符

- 月兔允许自定义部分运算符：比较、加减乘除、取值、设值
- 通过定义特定名称、类型的方法即可实现

```
fn BoxedInt::op_equal(i: BoxedInt, j: BoxedInt) -> Bool {
  i.value == j.value
}
fn BoxedInt::op_add(i: BoxedInt, j: BoxedInt) -> BoxedInt {
  { value: i.value + j.value }
}

fn init {
  let _ = { value: 10 } == { value: 100 } // false
  let _ = { value: 10 } + { value: 100 } // { value: 110 }
}
```

MoonBit 月兔编程

自定义运算符

- 月兔允许自定义部分运算符：比较、加减乘除、取值、设值
- 通过定义特定名称、类型的方法即可实现

```
// 使用：map [ key ]
fn Map::op_get[Key: Eq, Value](map: Map[Key, Value], key: Key) -> Option[Value] {
  get(map, key)
}
// 使用：map [ key ] = value
fn Map::op_set[Key: Eq, Value](map: Map[Key, Value], key: Key, value: Value) -> Map[Key, Value] {
  put(map, key, value)
}

fn init {
  let empty: Map[Int, Int] = make()
  let one = { empty[1] = 1 } // 等价于 let one = Map::op_set(empty, 1, 1)
  let _ = one[1] // 等价于 let _ = Map::op_get(one, 1)
}
```

总结

- 本章节展示了如何在月兔中
  - 定义接口 Trait并修饰类型变量
  - 实现方法及自定义运算符
- 以及简单的表的实现