

现代编程思想

接口

Hongbo Zhang

- 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

```
1. enum Tree[T] {  
2.     Empty  
3.     Node(T, Tree[T], Tree[T])  
4. }  
5.  
6. // 我们需要一个比较函数来比较值的大小以了解顺序  
7. // 负数表示小于，0表示等于，正数表示大于  
8. fn[T] insert(self: Tree[T], value: T, compare: (T, T) -> Int) -> Tree[T] { ... }  
9. fn[T] delete(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

```
1. fn[T] make(default: T) -> Queue[T] {  
2.     { array: Array::make(5, default), start: 0, end: 0, length: 0 }  
3. }
```

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**

接口 Trait

- 我们通过接口定义一系列方法的实现需求

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

- 月兔中的接口是名义上的，即必须声明为特定的接口实现方法

- 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[T : Default] make() -> 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

```
13 struct BoxedInt { value : Int }
14
15 fn init {
16   let a: Queue[BoxedInt] = make()
17 }
```

PROBLEMS 2 OUTPUT TERMINAL Filter (e.g. text, **/*.ts...)

001_hello.mbt 1

✗ Type BoxedInt does not implement trait Default: method default is missing [Ln 16, Col 28]

接口 Trait

```
1. fn[T : Compare] insert(tree : Tree[T], value : T) -> Tree[T] {
2.     // 类型参数T应当满足比较接口
3.     match tree {
4.         Empty => Node(value, Empty, Empty)
5.         Node(v, left, right) =>
6.             if T::compare(value, v) == 0 { // 可以使用比较方法
7.                 tree
8.             } else if T::compare(value, v) < 0 { // 可以使用比较方法
9.                 Node(v, insert(left, value), right)
10.            } else {
11.                Node(v, left, insert(right, value))
12.            }
13.     }
14. }
```

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 {
4.     { value : Int::default() } // 使用整数的默认值 0
5. }
6.
7. impl Default for BoxedInt with default() { // 可以省略类型标注
8.     BoxedInt::default()
9. }
10.
11. test {
12.     let array: Queue[BoxedInt] = make()
13. }
```

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

```
1. fn BoxedInt::plus_one(b: BoxedInt) -> BoxedInt {
2.     { value : b.value + 1 }
3. }
4. fn BoxedInt::plus_two(self: Self) -> Self { // 可以用 Self 代替参数名称
5.     { value : self.value + 2}
6. }
7.
8. fn init {
9.     let _five = { value: 1 }.plus_one().plus_one().plus_two()
10.    // 无需进行深层嵌套，方便理解
11.    let _five = BoxedInt::plus_two(
12.        BoxedInt::plus_one(BoxedInt::plus_one({value: 1}))
13.    )
14. }
```


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, Show)
```

- 需要数据结构内部的数据同样实现接口

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" }`

```
1. type Map[Key, Value]
2.
3. // 创建表
4. fn[Key, Value] make() -> Map[Key, Value] { ... }
5. // 添加键值对, 或更新键对应值
6. fn[Key, Value] put(map: Map[Key, Value], key: Key, value: Value) -> Map[Key, Value] { ... }
7. // 获取键对应值
8. fn[Key, Value] get(map: Map[Key, Value], key: Key) -> Option[Value] { ... }
```

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[Key: Eq, Value] get(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

```
1. // 我们定义一个类型Map, 其实际值为List[(Key, Value)]
2. type Map[Key, Value] = List[(Key, Value)]
3.
4. fn[Key, Value] make() -> Map[Key, Value] {
5.   Map(Nil)
6. }
7.
8. fn[Key, Value] put(map: Map[Key, Value], key: Key, value: Value) -> Map[Key, Value] {
9.   let Map(original_map) = map
10.  Map( Cons( (key, value), original_map ) )
11. }
```

表：利用接口实现

- 我们以列表+二元组作为表

```
1. fn[Key: Eq, Value] get(map : Map[Key, Value], key : Key) -> Option[Value] {
2.   fn aux(list : List[(Key, Value)]) -> Option[Value] {
3.     match list {
4.       Nil => None
5.       Cons((k, v), tl) => if k == key { // Key实现了Eq接口，因此可以利用==比较
6.         Some(v)
7.       } else {
8.         aux(tl)
9.       }
10.    }
11.  }
12.
13.  aux(map.inner()) // 利用 .inner() 取出实际的值
14. }
```

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
- 通过实现特定接口
- 特别的，对于取值、设值，通过定义特定名称、类型的方法即可实现

```
1. impl Eq for BoxedInt with op_equal(i: BoxedInt, j: BoxedInt) -> Bool {
2.   i.value == j.value
3. }
4. impl Add for BoxedInt with op_add(i, j) { // 可以省略
5.   { value: i.value + j.value }
6. }
7.
8. test {
9.   inspect({ value: 10 } == { value: 100 }, content="false")
10.  inspect({ value: 10 } + { value: 100 }, content="{value: 110}")
11. }
```

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
- 通过实现特定接口
- 特别的，对于取值、设值，通过定义特定名称、类型的方法即可实现

```
1. // 使用: map [ key ]
2. fn[Key: Eq, Value] Map::op_get(map: Map[Key, Value], key: Key) -> Option[Value] {
3.     get(map, key)
4. }
5. // 使用: map [ key ] = value
6. fn[Key, Value] Map::op_set(map: Map[Key, Value], key: Key, value: Value) -> Map[Key, Value] {
7.     put(map, key, value)
8. }
9.
10. test {
11.     let empty: Map[Int, Int] = make()
12.     let one = { empty[1] = 1 } // 等价于 let one = Map::op_set(empty, 1, 1)
13.     let _ = one[1] // 等价于 let _ = Map::op_get(one, 1)
14. }
```

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