第一期IC课程内容回顾及作业

廖师虎

2022-01-19

Content

- 课程回顾
- 作业点评
- 优秀作业

内容要点1

值、类型、类型推断、类型检查

- 值域 vs. 类型域
- 类型代表了静态语义
- 类型检查让代码更安全
- 类型标注可以帮助类型推断

```
import Blob "mo:base/Blob";
type Blob = Blob.Blob;
```

duplicate definition for Blob in block Motoko

mo:base/Blob

View Problem No quick fixes available

```
let Blob = "Blob";
```

```
var seed : [var Nat8] = [var 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
```

内容要点2

基础类型

- 布尔型 Bool
- 自然数 Nat, Nat8, Nat16, Nat32, Nat64
- 整数 Int, Int8, Int16, Int32, Int64
- 浮点数 Float
- 字符串 Text
- 字符 Char
- Principal
- Blob
- None
- Error

内容要点3

Record (记录结构) vs. Variant (枚举)

```
let person = {
  name = "Jacky Chan";
  age = 67;
};

func f() : {name: Text; age: Nat} {
  person
}
```

```
public type Message = {
    time : Time.Time;
    text : Text;
};
public type Blog = {
    text: Text;
    time: Time.Time;
};
```

```
let Mesg : Message ={
        text = "abc";
        time = Time.now();
};
let blog: Blog = Mesg;
```

元组 (tuple) 是记录结构 (record) 的特殊形式

```
let x : (Int, Bool) = (10, false);
let y : Bool = x.1;
```

```
type Gender = {
  #male;
  #female;
};

let person = {
  name = "Jacky Chan";
  age = 67;
  gender = #male;
};

func f() : {name: Text; age: Nat} {
  person
};
```

```
field gender does not exist in type
  {age : Nat; name : Text} Motoko
View Problem    No quick fixes available
```

```
let g = f().gender;
```

内容要点4

模式匹配 (Pattern match)

```
type Person = {
  name: Text;
  age: Nat;
  gender: Gender;
};

func retired(person: Person) : Bool {
  switch (person.gender) {
    case (#male) (person.age >= 60);
    case (#female) (person.age >= 55);
  }
};
```

```
type Gender = {
  #male;
  #female;
  #unspecified: {retire_age: Nat};
};

func retired(person: Person) : Bool {
  switch (person.gender) {
    case (#male) (person.age >= 60);
    case (#female) (person.age >= 55);
    case (#unspecified({retire_age})) (person.age >= retire_age);
  }
};
```

#unspecified : Nat

#unspecified retire_age

内容要点5

Option 和 Result

```
func retired(person: Person) : ?Bool {
  switch (person.gender) {
    case (#male) ?(person.age >= 60);
    case (#female) ?(person.age >= 55);
    case (#unspecified) null;
  }
};
```

- Option 类型: ?Bool, ?Nat, ...
- Option 值: null, ?true, ?12, ...

- Result 类型: Result<R, E>
- Result 值: #ok(true), #err(“Unknown”)

```
type Result<Ok, Err> = {
  #ok : Ok;
  #err : Err;
};
//import Result "mo:base/Result";
//type Result<R, E> = Result.Result<R, E>;

func retired(person: Person) : Result<Bool, Text> {
  switch (person.gender) {
    case (#male) #ok(person.age >= 60);
    case (#female) #ok(person.age >= 55);
    case (#unspecified) #err("Unknown");
  }
};
```

内容要点6

函数

- 函数: 从定义域 (Domain) 到值域 (Range) 的映射关系
- 类型: () -> Result<Bool, Text>, () -> (), ...
- 函数定义

```
func dec(a: Int) : Int { a - 1 };
func inc(a: Nat) : Nat { a + 1 };
```

- 匿名函数

```
let dec : Int -> Int = func (a) { a - 1 };
let inc = func (a: Nat) : Nat { a + 1 };
```

内容要点7

高阶函数

From “mo:base/Array”:

```
/// Initialize a mutable array with `size` copies of the initial value.
public func init<A>(size : Nat,  initVal : A) : [var A] {
  Prim.Array_init<A>(size, initVal);
};
/// Initialize an immutable array of the given size, and use the `gen` function to produce the initial value for every index.
public func tabulate<A>(size : Nat,  gen : Nat -> A) : [A] {
  Prim.Array_tabulate<A>(size, gen);
};

// arr = [var 42, 42, 42, 42, 42] : [var Int]
let arr = Array.init<Int>(5, 42);

// brr = [0, 1, 2, ..., 99] : [Nat]
let brr = Array.tabulate<Nat>(100, func (i) { i });

// crr = [0, 2, 4, ..., 198] : [Int]
let crr = Array.tabulate<Int>(100, func (i) { i * 2 });
```

内容要点8

Object (对象)

```
object counter {
  var count = 0;
  public func inc() { count += 1 };
  public func read() : Nat { count };
  public func bump() : Nat {
    inc();
    read()
  };
};
```

```
let counter : Counter = do {
  var count = 0;
  let inc = func () { count += 1; };
  let read = func () : Nat { count };
  {
    inc = inc;
    read = read;
    bump = func () : Nat { inc(); read() };
  }
};
```

```
type Counter = {
  inc: () -> ();
  read: () -> Nat;
  bump: () -> Nat;
};

let counter : Counter = object {
  var count = 0;
  public func inc() { count += 1 };
  public func read() : Nat { count };
  public func bump() : Nat {
    inc();
    read()
  };
};
```

内容要点9

Actor

```
actor Counter {

  var count = 0;

  public shared func inc() : async () { count += 1 };

  public shared func read() : async Nat { count };

  public shared func bump() : async Nat {
    count += 1;
    count;
  };
};
```

```
actor Counter {

  var count = 0;

  public shared func inc() : async () { count += 1 };

  public shared query func read() : async Nat { count };

  public shared func bump() : async Nat {
    await inc();
    await read();
  };
};
```

```
type Counter = actor {
  inc  : shared () -> async ();
  read : shared () -> async Nat;
  bump : shared () -> async Nat;
};
```

```
type Counter = actor {
  inc  : shared () -> async ();
  read : shared query () -> async Nat;
  bump : shared () -> async Nat;
};
```

内容要点10

实例 - Microblog

```
public type Message = Text;

public type Microblog = actor {
    follow: shared(Principal) -> async (); // 添加关注对象
    follows: shared query () -> async [Principal]; // 返回关注列表
    post: shared (Text) -> async (); // 发布新消息
    posts : shared query () -> async [Message]; // 返回所有发布的消息
    timeline : shared () -> async [Message]; // 返回所有关注对象发布的消息
};
```

一个（极简的）去中心化的社交网络应用

- 每个 canister 代表一个用户
- Canister 可以通过 canister id 相互关注

作业点评:

stable var: Trie, Array, List
var HashMap, Buffer
preupgrade postupgrade

重复follow TrieSet, HashSet

Principal

a) canister id dfx deploy

b) account id dfx deploy –no-wallet

shared ({caller = owner}) actor class MicroblogActor () {}

思考题

优秀作业:

https://github.com/jinhuaio/icp-study/tree/main/course4/icblog

如何提高 timeline 查询效率的方案

总体方案思路：对timeline方法调用canister.posts()方法的查询结果使用缓存机制，以便在下一次的查询中从当前canister的本地缓存中获取，以提高查询效率。

详细实现的关键步骤： 1、在调用关注微博方法 follow(id: Principal) 中，同时调用对方canister的订阅方法 详细见 doSubscribe(tagId: Principal) 2、在调用对方canister的订阅方法 await tagCanister.subscribe(0) 时，对方将返回其微博的所有历史消息内容，此时获取到所有历史消息内容后，对消息进行缓存至本地变量中var messagesCache 3、若此时对方的canister发布新的微博消息，此时在func post(text: Text) 方法中，将会同时调用 func broadcastToSubscribers(msg: Message) 分发给订阅者新消息 4、订阅者将在func receiveSubscribe(msg: Message) 方法中获取到推送过来的新消息，此时同样将新消息缓存在本地变量 var messagesCache 中。 5、此时，若通过 func timeline(since: Time.Time) 方法查询本canister关注的消息列表，若本地已有缓存，则优先从本地缓存 var messagesCache 中查找；若本地尚未有对该canister进行过缓存（通过var initMsgCache变量来标记是否有记录过缓存），则先通过对方的canister.posts(0)获取所有消息，然后再次缓存至本地变量 var messagesCache 6、因 var messagesCache 没有做成持久化，因此在canister升级时，缓存数据会丢失，此时会在首次调用 timeline 方法时，会重新获取一次所有信息，并对其再次缓存在本地。

思考：该方案也存在一些缺陷，若我有太多的粉丝关注了我的微博，那么在我发微博消息时 func post(text: Text)，会因需要广播给所有的粉丝，因此同样会存在性能瓶颈，这里是否有办法能在canister中执行多线程的异步后台任务？

备注：关于上述方案可以参考本项目 src/icblog/main.mo 的代码实现。

Thanks