MoonBit-Pearls-Choreographic-Programming-with-Moonchor

传统的分布式程序设计是非常痛苦的，其中一个重要的因素是很多整体的逻辑需要拆散到各个分布式节点中实现，分散的实现使得程序难以调试、难以理解，并且无法享用编程语言提供的类型检查能力。Choreographic Programming，即协同式编程，提供了一种整体的视角，允许开发者编写需要多个参与者协同工作的单一程序，然后将这个整体程序分别投射到各个参与者，最终实现协同工作的效果。

协同式编程通过两种不同的方式实现：其一是作为一种全新的编程语言，例如 Choral，开发者编写 Choral 程序，然后用编译器将这个单体程序编译到各个参与者专属的 Java 程序；其二是作为一个库，例如 HasChor，直接利用 Haskell 的类型系统就能实现协同式编程的静态性质，并且完美兼容 Haskell 的生态。MoonBit 的函数式编程特性和强大的类型系统使得它很适合用于构建协同式编程的库。

本文旨在使用 MoonBit 语言的协同式编程库 moonchor，用多个例子阐释协同式编程的核心思想和基本用法。

导览：书店应用

让我们考察一个书店应用。应用包含两个角色：买家和卖家。应用的核心逻辑如下：

1. 买家向卖家发送想要购买的书的标题；
2. 卖家通过查询数据库告诉买家书的价格；
3. 买家决定是否购买书籍；
4. 如果买家决定购买，卖家从数据库中扣除书籍的库存并发送预期送达日期给买家；
5. 否则，交互中止。

传统实现

我们在此不关心实现细节，只关心核心逻辑，使用 `send` 和 `recv` 函数来表示发送和接收消息。按照传统的实现方式，我们需要为买家和卖家分别开发两个应用。在表示这些应用之前，我们假设已经存在一些函数和类型：

```
fn get_title() -> String {
  "Homotopy Type Theory"
}

fn get_price(title : String) -> Int {
  50
}
```

```
fn get_budget() -> Int {
  100
}

fn get_delivery_date(title : String) -> String {
  "2025-10-01"
}

enum Role {
  Buyer
  Seller
}

async fn[T] send(msg : T, target : Role) -> Unit {
  ...
}

async fn[T] recv(source : Role) -> T {
  ...
}
```

买家的应用如下：

```
async fn book_buyer() -> Unit {
  let title = get_title()
  send(title, Seller)
  let price = recv(Seller)
  if price ≤ get_budget() {
    send(true, Seller)
    let delivery_date = recv(Seller)
    println("The book will be delivered on: \{delivery_date}")
  } else {
    send(false, Seller)
  }
}
```

卖家的应用如下：

```
async fn book_seller() -> Unit {
  let title = recv(Buyer)
  let price = get_price(title)
  send(price, Buyer)
  let decision = recv(Buyer)
  if decision {
```

```
    let delivery_date = get_delivery_date(title)
    send(delivery_date, Buyer)
  }
}
```

这两个应用至少有以下几个问题：

1. 无法保证类型安全：注意到 `send` 和 `recv` 都是泛型函数，只有当发送和接收的类型一致时，才能保证类型安全；否则，可能会在序列化、反序列化过程发生运行时错误。而编译期无法检查这种类型安全性，因为编译器无法知道每个 `send` 对应哪个 `recv`，只能寄希望于开发者不会写错。
2. 可能导致死锁：万一买家程序的某个 `send` 语句漏写了，买家和卖家可能会同时等待对方的消息；或者在网络交互时，某个买家连接暂时断开了，卖家也会一直等待买家的消息。上述两种情况都导致死锁。
3. 需要显式同步：买家为了向卖家传达是否要购买的决定，必须显式地发送一个 `Bool` 类型的消息。后续的协同过程需要保证买家和卖家在 `if price ⩽ get_budget()` 和 `if decision` 这两个位置走进相同的分支，而这一特点也是无法在编译期保证的。

导致这些问题的根本原因是我们将一个整体的协同逻辑按照实现的需求拆成了两个独立的部分。接下来，我们看看使用协同式编程如何解决上述问题。

moonchor 实现

使用协同式编程，我们可以将买家和卖家的逻辑写在同一个函数中，然后让它根据调用该函数时不同的参数表现出不同的行为。我们使用 moonchor 中的 API 来定义买家和卖家的角色。在 moonchor 中，角色被定义为 `trait Location`。为了提供更好的静态性质，角色不仅是值，同时还是一个独特的类型，该类型需要实现 `Location` 这个 trait。

```
struct Buyer {} derive(Eq, Show, Hash)

impl @moonchor.Location for Buyer with name(_) {
  "buyer"
}

struct Seller {} derive(Eq, Show, Hash)

impl @moonchor.Location for Seller with name(_) {
  "seller"
}

let buyer : Buyer = Buyer::{  }

let seller : Seller = Seller::{  }
```

可以看见，我们定义的 `Buyer` 和 `Seller` 类型不包含任何字段。实现 `Location` trait 的类型只需要提供一个 `name` 方法，返回一个字符串作为角色的名称。这个 `name` 方法非常重要，它标识着角色的身份属性，并在类型检查无法保证类型安全时，提供终端检查手段。不要为不同的角色设置相同的名称，否则会导致意外的运行时错误。我们将在后文了解到类型如何保证一定程度的安全性，以及为什么仅依靠类型是不够的。

接下来，我们定义书店应用的核心逻辑，它被称作一个 choreography：

```
async fn bookshop(ctx : @moonchor.ChoreoContext) -> Unit {
  let title_at_buyer = ctx.locally(buyer, _unwrapper => get_title())
  let title_at_seller = ctx.comm(buyer, seller, title_at_buyer)
  let price_at_seller = ctx.locally(seller, fn(unwrapper) {
    let title = unwrapper.unwrap(title_at_seller)
    get_price(title)
  })
  let price_at_buyer = ctx.comm(seller, buyer, price_at_seller)
  let decision_at_buyer = ctx.locally(buyer, fn(unwrapper) {
    let price = unwrapper.unwrap(price_at_buyer)
    price < get_budget()
  })
  if ctx.broadcast(buyer, decision_at_buyer) {
    let delivery_date_at_seller = ctx.locally(seller, unwrapper =>
get_delivery_date(
      unwrapper.unwrap(title_at_seller),
    ))
    let delivery_date_at_buyer = ctx.comm(
      seller, buyer, delivery_date_at_seller,
    )
    ctx.locally(buyer, fn(unwrapper) {
      let delivery_date = unwrapper.unwrap(delivery_date_at_buyer)
      println("The book will be delivered on \{delivery_date}")
    })
    ▷ ignore
  }
}
```

这个程序稍微有点长，我们先逐行分析一下。

函数的参数 `ctx: @moonchor.ChoreoContext` 是 moonchor 给应用提供的上下文对象，它包含了协同式编程在应用侧的所有接口。首先，我们使用 `ctx.locally` 执行一个仅在买家角色处需要执行的操作 `get_title()`。`ctx.locally` 的第一个参数是角色，第二个参数是一个闭包，闭包的内容就是需要执行的参数，返回值被包装后作为 `ctx.locally` 的返回值。在这里，`get_title()` 的返回值是 `String` 类型，而 `title_at_buyer` 的类型是 `@moonchor.Located[String, Buyer]`，表示这个值位于买家这个角色，无法被其它角色使

用。当你试图在卖家角色中使用 `title_at_buyer` 时，编译器会报错，告诉你 Buyer 和 Seller 不是同一个类型。

接下来，买家需要将书名发送给卖家，我们使用 `ctx.comm` 来实现这个操作。`ctx.comm` 的第一个参数是发送者角色，第二个参数是接收者角色，第三个参数是发送的内容。在这里，`ctx.comm` 的返回值 `title_at_seller` 的类型是 `@moonchor.Located[String, Seller]`，表示这个值位于卖家角色。你已经猜到了，`ctx.comm` 对应的操作正是 `send` 和 `recv`。但这里，类型得到了保障：`ctx.comm` 是一个泛型函数，它保证1）发送和接受的消息是同一个类型；2）发送者和接收者的角色对应为参数类型和返回值类型的类型参数，即 `@moonchor.Located[T, Sender]` 和 `@moonchor.Located[T, Receiver]`。

再往下，卖家开始通过查询数据库获取书的价格。在这一步我们用到了 `ctx.locally` 传递给闭包的参数 `unwrapper`。这个参数是一个用于为 Located 类型解包的对象，它的类型签名中也包含一个角色类型参数，我们通过 `Unwrapper::unwrap` 方法的签名即可看懂它是如何工作的：`fn[T, L] Unwrapper::unwrap(_ : Unwrapper[L], v : Located[T, L]) -> T`。也就是说，`ctx.locally(buyer, unwrapper => ...)` 中的 `unwrapper` 的类型是 `Unwrapper[Buyer]`，而 `title_at_seller` 的类型是 `Located[String, Seller]`，因此 `unwrapper.unwrap(title_at_seller)` 的结果类型是 `String`。这就是为什么我们可以在闭包中使用 `title_at_seller` 而不能使用 `title_at_buyer` 的原因。

Knowledge of Choice

在后续的流程中，如何解决显式同步问题是一个关键点，以至于我们要单独用一个小节来说明。在协同式编程中，这个问题被称作 Knowledge of Choice（选择知识）。在上面的例子中，买家需要知道是否购买书籍，而卖家需要知道买家是否购买书籍。我们使用 `ctx.broadcast` 来实现这个功能。

`ctx.broadcast` 的第一个参数是发送者的角色，第二个参数是需要共享给所有其它角色的消息。在这个例子中，买家和卖家都需要知道买家是否购买书籍，因此买家要将这一决定 `decision_at_buyer` 通过 `ctx.broadcast` 发送给所有参与者（在这里只有卖家）。有趣的是，这个 `broadcast` 的返回值是一个普通类型而非 `Located` 类型，这意味着它可以被所有角色使用，并且直接在顶层使用而不需要 `unwrapper` 解包。因此，我们能够利用 MoonBit 本身的 `if` 条件语句来编写后续流程，从而保证买家和卖家在 `if` 分支中走入相同的分支。

从名字可以看出，`ctx.broadcast` 的作用是在整个 choreography 中广播一个值。它不仅可以广播一个 `Bool` 类型，也可以广播任意其它类型。它的结果不仅可以应用于 `if` 条件语句，也可以用于 `while` 循环或者任何其它需要公共知识的地方。

启动代码

这样一个 choreography 怎样运行呢？moonchor 提供了 `run_choreo` 函数来启动一个 choreography。目前，由于 MoonBit 的多后端特性，提供稳定的、可移植的 TCP 服务器和跨进

程通信接口是一项挑战，因此我们将使用管道和协程来探寻 choreography 的真正运行过程。完整的启动代码如下：

```
test "Blog: bookshop" {
  let backend = @moonchor.make_local_backend([buyer, seller])
  @toolkit.run_async(() => @moonchor.run_choreo(backend, bookshop, buyer)
)
  @toolkit.run_async(() => @moonchor.run_choreo(backend, bookshop, seller)
)
}
```

上述代码启动了两个协程，分别在买家和卖家处执行同一个 choreography。也可以理解为，`bookshop` 这个函数被投射成（也被称为 EPP，端点投射）了「买家版」和「卖家版」两个完全不同的版本。在上面的例子中，`run_choreo` 的第一个参数是一个 `Backend` 类型的对象，它提供了协同式编程所需的底层通信机制。我们使用 `make_local_backend` 函数创建了一个本地后端（不要和刚刚提到的 MoonBit 多后端混淆），这个后端可以在本地进程中运行，使用 `peter-jerry-ye/async/channel` 提供的管道 API 作为通信基础。在未来，moonchor 还会提供更多的后端实现，例如 HTTP。

API 和部分原理

我们已经对协同式编程和 moonchor 有了初步的了解。接下来，我们正式引入刚刚用到的 API 以及一些没有用到的 API，并且介绍它们的部分原理。

角色

在 moonchor 中，我们通过实现 `Location` 这个 trait 来定义角色。该 trait 的声明如下：

```
pub(open) trait Location: Show + Hash {
  name(Self) -> String
}
```

`Location` 的 trait object 实现了 `Eq`：

```
impl Eq for &Location with op_equal(self, other) {
  self.name() == other.name()
}
```

如果两个角色的 `name` 方法返回相同的字符串，那么它们被认为是同一个角色，否则就不是。在判断某个值是否是某个角色时，`name` 方法是最终决定者。也就是说，可以存在类型相同但实际上不是同一角色的值。这个特性在处理动态生成的角色时是尤其重要的。比如在书店例子中，买

家有可能不止一个，卖家需要同时处理多个买家请求，并且根据服务器接收到的连接来动态生成买家角色。此时，买家的类型定义如下：

```
struct DynamicBuyer {
  id : String
} derive(Eq, Show, Hash)

impl @moonchor.Location for DynamicBuyer with name(self) {
  "buyer-\{self.id}"
}
```

Located Values

因为 choreography 中会同时出现位于不同角色的值，因此我们需要某种手段来区分每个值都是位于哪个角色之处的。在 moonchor 中，这个用 `Located[T, L]` 这个类型表示位于角色 `L` 处的类型为 `T` 的值。

```
type Located[T, L]

type Unwrapper[L]
```

构建一个 `Located` 值的方式是通过 `ChoreoContext::locally` 或 `ChoreoContext::comm`。这两个函数都会返回一个 `Located` 值。

使用一个 `Located` 值的方式是通过 `Unwrapper` 对象的 `unwrap` 方法。这些内容在上面的书店应用中已经展示过了，不作赘述。

局部计算

我们在例子中见到的最常见的 API 即为 `ChoreoContext::locally`，它用于在某个角色处执行一个局部计算动作。其签名如下：

```
type ChoreoContext

fn[T, L : Location] locally(
  self : ChoreoContext,
  location : L,
  computation : (Unwrapper[L]) -> T
) -> Located[T, L] {
  ...
}
```

该 API 表示会在 `location` 这个角色处执行 `computation` 这个闭包，并将计算结果包装成一个 Located Value。`computation` 闭包的唯一参数是一个解包器对象，类型为 `Unwrapper[L]`，它在闭包中用于将 `Located[T, L]` 类型的值解包成 `T` 类型。这个 API 的作用是将计算的结果绑定到某个角色上，确保该值只能在该角色处使用。如果试图在其它角色处使用这个值，或用这个解包器处理其它角色的值，编译器会报错。

通信

`ChoreoContext::comm` API 用于将一个值从一个角色发送到另一个角色。其签名如下：

```
trait Message: ToJson + @json.FromJson {}

async fn[T : Message, From : Location, To : Location] comm(
  self : ChoreoContext,
  from : From,
  to : To,
  value : Located[T, From]
) -> Located[T, To] {
  ...
}
```

发送和接收通常意味着需要序列化和反序列化过程。在 moonchor 目前的实现中，使用 Json 而非字节流作为消息的物理载体。这也是一个不需要过度关注的细节，在未来可能会有更高效的实现。

`ChoreoContext::comm` 有三个类型参数，除了要发送的消息类型，还有发送方和接收方的角色类型 `From` 和 `To`。这两个类型刚好对应了该方法的 `from` 参数、`to` 参数，以及 `value` 参数和返回值的类型。这保证了发送方和接收方在该消息序列化、反序列化的类型安全性，并且保证发送和接收行为必然会配对，不会疏忽导致死锁。

广播

当需要在多个角色之间共享一个值时，我们使用 `ChoreoContext::broadcast` API 让某个角色将一个值广播给所有其它角色。其签名如下：

```
async fn[T : Message, L : Location] ChoreoContext::broadcast(
  self : ChoreoContext,
  loc : L,
  value : Located[T, L]
) -> T {
  ...
}
```

广播和通信的 API 很相似，除了两点不同：

1. 广播不需要指明接收方的角色，默认是该 choreography 中的所有角色；
2. 广播的返回值并非 Located Value，而是消息本身的类型。

这两个特点揭示了广播的目的：所有角色都能访问到同一个值，从而在 choreography 的顶层对该值进行操作而不是局限在 `ChoreoContext::locally` 方法内部。例如在书店例子中，买家和卖家需要对「是否购买」这一决定达成共识，以确保后续的流程仍然保持一致。

后端和运行

运行一个 choreography 的 API 如下：

```
type Backend

typealias async (ChoreoContext) -> T as Choreo[T]

async fn[T, L : Location] run_choreo(
  backend : Backend,
  choreography : Choreo[T],
  role : L
) -> T {
  ...
}
```

它接收三个参数：一个后端、一个用户编写的 choreography 和一个待运行的角色。后端包含了通信机制的具体实现，待运行的角色则是指定这个 choreography 要在哪个位置执行。比如之前的例子中，买家的程序需要在此处传递一个 `Buyer` 类型的值，而卖家需要传递 `Seller` 类型的值。

moonchor 提供了一个基于协程和通道的本地后端：

```
fn make_local_backend(locations : Array[&Location]) -> Backend {
  ...
}
```

这个函数为参数中的所有角色之间构建通信通道，提供具体的通信实现，即 `send` 和 `recv` 方法。尽管本地后端只能用于单体并发程序而非真正的分布式应用程序，但它的实现是可插拔的。只要拥有了基于稳定的网络通信 API 实现的其它后端，moonchor 就能轻松用于构建分布式程序了。

（可选阅读）案例研究：多副本 KVStore

在本节中，我们将探讨一个更复杂的案例，使用 moonchor 实现多副本的 KVStore。我们依然只使用 moonchor 的核心 API，但会充分利用 MoonBit 的泛型和一等公民函数来探索使用

moonchor 进行协同式编程的丰富表达能力。

基本实现

首先做一些准备工作，定义客户端 Client 和服务器 Server 两个角色：

```
struct Server {} derive(Eq, Hash, Show)

struct Client {} derive(Eq, Hash, Show)

impl @moonchor.Location for Server with name(_) {
  "server"
}

impl @moonchor.Location for Client with name(_) {
  "client"
}

let server : Server = Server::{  }

let client : Client = Client::{  }
```

要实现一个 KVStore，例如 Redis，我们需要实现最基本的两个接口：get 和 put（对应 Redis 的 set）。最简单的实现就是用一个 Map 数据结构来存储键值对：

```
struct ServerState {
  db : Map[String, Int]
}

fn ServerState::new() -> ServerState {
  { db: {} }
}
```

对于 KVStore 而言，get 和 put 请求是客户端通过网络发送过来的，在接收到请求前，我们并不知道具体的请求是什么。所以我们需要定义一个请求类型 `Request`，它包含了请求的类型和参数：

```
enum Request {
  Get(String)
  Put(String, Int)
} derive(ToJson, FromJson)
```

为了方便，我们的 KVStore 只支持 `String` 类型的键和 `Int` 类型的值。接下来，我们定义一个 `Response` 类型，用于表示服务器对请求的响应：

```
typealias Int? as Response
```

响应是一个可选的整数。当请求是 `Put` 时，响应是 `None`；当请求是 `Get` 时，响应是对应键的值，如果键不存在，则响应为 `None`。

```
fn handle_request(state : ServerState, request : Request) -> Response {
  match request {
    Request::Get(key) => state.db.get(key)
    Request::Put(key, value) => {
      state.db[key] = value
      None
    }
  }
}
```

我们的目标是定义两个函数 `put` 和 `get` 模拟客户端发起请求的过程。它们要做的事情分别是：

1. 在 Client 处生成请求，包装键值对；
2. 将请求发送给 Server；
3. Server 使用 `handle_request` 函数处理请求；
4. 将响应发送回 Client。

可以看到，`put` 和 `get` 函数的逻辑是相似的，我们可以把 2、3、4 三个过程抽象成一个函数，叫作 `access_server`。

```
async fn put_v1(
  ctx : @moonchor.ChoreoContext,
  state_at_server : @moonchor.Located[ServerState, Server],
  key : String,
  value : Int
) -> Unit {
  let request = ctx.locally(client, _unwrapper => Request::Put(key,
value))
  access_server_v1(ctx, request, state_at_server) |> ignore
}

async fn get_v1(
  ctx : @moonchor.ChoreoContext,
  state_at_server : @moonchor.Located[ServerState, Server],
  key : String
```

```
) -> @moonchor.Located[Response, Client] {
  let request = ctx.locally(client, _unwrapper => Request::Get(key))
  access_server_v1(ctx, request, state_at_server)
}

async fn access_server_v1(
  ctx : @moonchor.ChoreoContext,
  request : @moonchor.Located[Request, Client],
  state_at_server : @moonchor.Located[ServerState, Server]
) -> @moonchor.Located[Response, Client] {
  let request_at_server = ctx.comm(client, server, request)
  let response = ctx.locally(server, fn(unwrapper) {
    let request = unwrapper.unwrap(request_at_server)
    let state = unwrapper.unwrap(state_at_server)
    handle_request(state, request)
  })
  ctx.comm(server, client, response)
}
```

这样我们的 KVStore 就完成了。我们可以写一个简单的 choreography 来测试它：

```
async fn kvstore_v1(ctx : @moonchor.ChoreoContext) -> Unit {
  let state_at_server = ctx.locally(server, _unwrapper =>
ServerState::new())
  put_v1(ctx, state_at_server, "key1", 42)
  put_v1(ctx, state_at_server, "key2", 41)
  let v1_at_client = get_v1(ctx, state_at_server, "key1")
  let v2_at_client = get_v1(ctx, state_at_server, "key2")
  ctx.locally(client, fn(unwrapper) {
    let v1 = unwrapper.unwrap(v1_at_client).unwrap()
    let v2 = unwrapper.unwrap(v2_at_client).unwrap()
    if v1 + v2 == 83 {
      println("The server is working correctly")
    } else {
      panic()
    }
  })
  |> ignore
}

test "kvstore v1" {
  let backend = @moonchor.make_local_backend([server, client])
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v1,
server))
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v1,
```

```
client))
}
```

这个程序的含义是，分别在 "key1" 和 "key2" 存储两个数字 42 和 41，然后从服务器获取这两个值并检查它们的和是否等于 83。如果有任何一个请求返回 None 或者计算结果不是 83，程序就会 panic。

双副本

现在，考虑为 KVStore 增加容错功能。最简单的容错就是构建一个从副本，它与主副本存有相同的数据，并在处理 `Get` 请求时检查主从数据的一致性。

我们为从副本构建一个新的角色：

```
struct Backup {} derive(Eq, Hash, Show)

impl @moonchor.Location for Backup with name(_) {
  "backup"
}

let backup : Backup = Backup::{  }
```

定义一个函数用于检查一致性：这个函数会检查所有副本的响应是否一致，如果不一致，则 panic。

```
fn check_consistency(responses : Array[Response]) -> Unit {
  match responses.pop() {
    None => return
    Some(f) =>
      for res in responses {
        if res != f {
          panic()
        }
      }
  }
}
```

其余的大部分内容都不需要修改，只要在 `access_server` 函数中增加对副本的处理即可。新的 `access_server_v2` 的逻辑是，Server 接收到请求后，将请求转发给 Backup；然后 Server 和 Backup 分别处理请求；Backup 处理完请求后发回给 Server，Server 对两个结果进行一致性检验。

```
async fn put_v2(
  ctx : @moonchor.ChoreoContext,
  state_at_server : @moonchor.Located[ServerState, Server],
  state_at_backup : @moonchor.Located[ServerState, Backup],
  key : String,
  value : Int
) → Unit {
  let request = ctx.locally(client, _unwrapper ⇒ Request::Put(key,
value))
  access_server_v2(ctx, request, state_at_server, state_at_backup) ▷
ignore
}

async fn get_v2(
  ctx : @moonchor.ChoreoContext,
  state_at_server : @moonchor.Located[ServerState, Server],
  state_at_backup : @moonchor.Located[ServerState, Backup],
  key : String
) → @moonchor.Located[Response, Client] {
  let request = ctx.locally(client, _unwrapper ⇒ Request::Get(key))
  access_server_v2(ctx, request, state_at_server, state_at_backup)
}

async fn access_server_v2(
  ctx : @moonchor.ChoreoContext,
  request : @moonchor.Located[Request, Client],
  state_at_server : @moonchor.Located[ServerState, Server],
  state_at_backup : @moonchor.Located[ServerState, Backup]
) → @moonchor.Located[Response, Client] {
  let request_at_server = ctx.comm(client, server, request)
  let request_at_backup = ctx.comm(server, backup, request_at_server)
  let response_at_backup = ctx.locally(backup, fn(unwrapper) {
    let request = unwrapper.unwrap(request_at_backup)
    let state = unwrapper.unwrap(state_at_backup)
    handle_request(state, request)
  })
  let backup_response_at_server = ctx.comm(backup, server,
response_at_backup)
  let response_at_server = ctx.locally(server, fn(unwrapper) {
    let request = unwrapper.unwrap(request_at_server)
    let state = unwrapper.unwrap(state_at_server)
    let response = handle_request(state, request)
    let backup_response = unwrapper.unwrap(backup_response_at_server)
    check_consistency([response, backup_response])
    response
  })
```

```
  ctx.comm(server, client, response_at_server)
}
```

和刚才一样，我们可以写一个简单的 choreography 来测试它：

```
async fn kvstore_v2(ctx : @moonchor.ChoreoContext) -> Unit {
  let state_at_server = ctx.locally(server, _unwrapper =>
ServerState::new())
  let state_at_backup = ctx.locally(backup, _unwrapper =>
ServerState::new())
  put_v2(ctx, state_at_server, state_at_backup, "key1", 42)
  put_v2(ctx, state_at_server, state_at_backup, "key2", 41)
  let v1_at_client = get_v2(ctx, state_at_server, state_at_backup, "key1")
  let v2_at_client = get_v2(ctx, state_at_server, state_at_backup, "key2")
  ctx.locally(client, fn(unwrapper) {
    let v1 = unwrapper.unwrap(v1_at_client).unwrap()
    let v2 = unwrapper.unwrap(v2_at_client).unwrap()
    if v1 + v2 == 83 {
      println("The server is working correctly")
    } else {
      panic()
    }
  })
  ▷ ignore
}

test "kvstore 2.0" {
  let backend = @moonchor.make_local_backend([server, client, backup])
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v2,
server) )
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v2,
client) )
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v2,
backup) )
}
```

利用高阶函数抽象复制策略

在双副本实现过程中，出现了一些耦合的代码：Server 处理请求、备份请求、检查结果一致性的代码放在了一起。

利用 MoonBit 的高阶函数特性，我们可以把复制策略从具体处理过程中抽象出来。我们分析一下，什么是复制策略？它应该包含一个过程，即服务器拿到请求后如何利用各个副本处理它的方式。关键在于，构建一个复制策略的过程本身是和请求无关的。这样的话，我们就能让复制策略

成为可替换的部分，从而日后能轻易地在不同的复制策略之间进行切换，或者实现新的复制策略。

当然，真实世界的复制策略是非常复杂的，往往很难清晰地从处理流程中剥离出来。但是在这个例子中，我们为了简化问题，体现 moonchor 的编程能力，直接将复制策略定义为 Server 在接收到请求后决定如何处理请求的函数。我们可以用一个类型别名来定义它：

```
typealias async (@moonchor.ChoreoContext, @moonchor.Located[Request,
Server]) → @moonchor.Located[
  Response,
  Server,
] as ReplicationStrategy
```

接下来，我们就可以简化 `access_server` 的实现了。我们将策略作为参数传递进去：

```
async fn access_server_v3(
  ctx : @moonchor.ChoreoContext,
  request : @moonchor.Located[Request, Client],
  strategy : ReplicationStrategy
) → @moonchor.Located[Response, Client] {
  let request_at_server = ctx.comm(client, server, request)
  let response = strategy(ctx, request_at_server)
  ctx.comm(server, client, response)
}

async fn put_v3(
  ctx : @moonchor.ChoreoContext,
  strategy : ReplicationStrategy,
  key : String,
  value : Int
) → Unit {
  let request = ctx.locally(client, _unwrapper ⇒ Request::Put(key,
value))
  access_server_v3(ctx, request, strategy) ▷ ignore
}

async fn get_v3(
  ctx : @moonchor.ChoreoContext,
  strategy : ReplicationStrategy,
  key : String
) → @moonchor.Located[Response, Client] {
  let request = ctx.locally(client, _unwrapper ⇒ Request::Get(key))
  access_server_v3(ctx, request, strategy)
}
```

这样一来，复制策略被成功从处理请求的逻辑中抽象出来了。下面，我们重新实现一遍双副本的复制策略：

```
async fn double_replication_strategy(
  state_at_server : @moonchor.Located[ServerState, Server],
  state_at_backup : @moonchor.Located[ServerState, Backup],
) → ReplicationStrategy {
  fn(
    ctx : @moonchor.ChoreoContext,
    request_at_server : @moonchor.Located[Request, Server]
  ) {
    let request_at_backup = ctx.comm(server, backup, request_at_server)
    let response_at_backup = ctx.locally(backup, fn(unwrapper) {
      let request = unwrapper.unwrap(request_at_backup)
      let state = unwrapper.unwrap(state_at_backup)
      handle_request(state, request)
    })
    let backup_response = ctx.comm(backup, server, response_at_backup)
    ctx.locally(server, fn(unwrapper) {
      let request = unwrapper.unwrap(request_at_server)
      let state = unwrapper.unwrap(state_at_server)
      let res = handle_request(state, request)
      check_consistency([unwrapper.unwrap(backup_response), res])
      res
    })
  }
}
```

其中，`do_backup` 函数负责执行从副本处理请求的逻辑。注意看 `double_replication_strategy` 的函数签名，它返回一个 `ReplicationStrategy` 类型的函数。只要提供两个参数，`double_replication_strategy` 就能构造出一个新的复制策略。至此，我们成功利用高阶函数抽象出了复制策略，这个特性在协同式编程中叫作高阶 choreography。

同样的，我们可以写一个简单的 choreography 来测试它：

```
async fn kvstore_v3(ctx : @moonchor.ChoreoContext) → Unit {
  let state_at_server = ctx.locally(server, _unwrapper ⇒
ServerState::new())
  let state_at_backup = ctx.locally(backup, _unwrapper ⇒
ServerState::new())
  let strategy = double_replication_strategy(state_at_server,
state_at_backup)
  put_v3(ctx, strategy, "key1", 42)
```

```
  put_v3(ctx, strategy, "key2", 41)
  let v1_at_client = get_v3(ctx, strategy, "key1")
  let v2_at_client = get_v3(ctx, strategy, "key2")
  ctx.locally(client, fn(unwrapper) {
    let v1 = unwrapper.unwrap(v1_at_client).unwrap()
    let v2 = unwrapper.unwrap(v2_at_client).unwrap()
    if v1 + v2 == 83 {
      println("The server is working correctly")
    } else {
      panic()
    }
  })
  ▷ ignore
}

test "kvstore 3.0" {
  let backend = @moonchor.make_local_backend([server, client, backup])
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v2,
server))
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v2,
client))
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v2,
backup))
}
```

利用参数化多态实现角色多态

如果要进一步实现新的复制策略，例如三副本，我们需要定义两个新的 Backup 类型以做区分：

```
struct Backup1 {} derive(Eq, Hash, Show)

impl @moonchor.Location for Backup1 with name(_) {
  "backup1"
}

let backup1 : Backup1 = Backup1::{  }

struct Backup2 {} derive(Eq, Hash, Show)

impl @moonchor.Location for Backup2 with name(_) {
  "backup2"
}

let backup2 : Backup2 = Backup2::{  }
```

接下来需要修改 `access_server` 的核心逻辑。我们立刻发现了问题，为了让 Backup1 和 Backup2 都处理一遍请求并且得到响应，需要将以下几条语句重复：`let request = unwrapper.unwrap(request_at_backup); let state = unwrapper.unwrap(state_at_backup); handle_request(state, request)`。重复代码是坏味道，应当被抽象出来。此时，moonchor 的「角色作为类型」优势就体现出来了，我们可以利用 MoonBit 的参数化多态，将从副本处理逻辑抽象成一个多态函数 `do_backup`，它接收一个角色类型参数 `B`，表示从副本的角色：

```
async fn[B : @moonchor.Location] do_backup(
  ctx : @moonchor.ChoreoContext,
  request_at_server : @moonchor.Located[Request, Server],
  backup : B,
  state_at_backup : @moonchor.Located[ServerState, B]
) → @moonchor.Located[Response, Server] {
  let request_at_backup = ctx.comm(server, backup, request_at_server)
  let response_at_backup = ctx.locally(backup, fn(unwrapper) {
    let request = unwrapper.unwrap(request_at_backup)
    let state = unwrapper.unwrap(state_at_backup)
    handle_request(state, request)
  })
  ctx.comm(backup, server, response_at_backup)
}
```

如此一来，我们就能随心所欲地实现双副本或者三副本的复制策略了。对于三副本策略，只需在 `triple_replication_strategy` 返回的函数内调用 `do_backup` 两次即可：

```
async fn triple_replication_strategy(
  state_at_server : @moonchor.Located[ServerState, Server],
  state_at_backup1 : @moonchor.Located[ServerState, Backup1],
  state_at_backup2 : @moonchor.Located[ServerState, Backup2]
) → ReplicationStrategy {
  fn(
    ctx : @moonchor.ChoreoContext,
    request_at_server : @moonchor.Located[Request, Server]
  ) {
    let backup_response1 = do_backup(
      ctx, request_at_server, backup1, state_at_backup1,
    )
    let backup_response2 = do_backup(
      ctx, request_at_server, backup2, state_at_backup2,
    )
    ctx.locally(server, fn(unwrapper) {
      let request = unwrapper.unwrap(request_at_server)
      let state = unwrapper.unwrap(state_at_server)
```

```
      let res = handle_request(state, request)
      check_consistency([
        unwrapper.unwrap(backup_response1),
        unwrapper.unwrap(backup_response2),
        res,
      ])
      res
    })
  }
}
```

由于我们成功完成了复制策略和访问过程的分离，`access_server`、`put`、`get` 函数不需要任何修改。让我们对最终的 KVStore 进行测试：

```
async fn kvstore_v4(ctx : @moonchor.ChoreoContext) -> Unit {
  let state_at_server = ctx.locally(server, _unwrapper =>
ServerState::new())
  let state_at_backup1 = ctx.locally(backup1, _unwrapper =>
ServerState::new())
  let state_at_backup2 = ctx.locally(backup2, _unwrapper =>
ServerState::new())
  let strategy = triple_replication_strategy(
    state_at_server, state_at_backup1, state_at_backup2,
  )
  put_v3(ctx, strategy, "key1", 42)
  put_v3(ctx, strategy, "key2", 41)
  let v1_at_client = get_v3(ctx, strategy, "key1")
  let v2_at_client = get_v3(ctx, strategy, "key2")
  ctx.locally(client, fn(unwrapper) {
    let v1 = unwrapper.unwrap(v1_at_client).unwrap()
    let v2 = unwrapper.unwrap(v2_at_client).unwrap()
    if v1 + v2 == 83 {
      println("The server is working correctly")
    } else {
      panic()
    }
  })
  |> ignore
}

test "kvstore 4.0" {
  let backend = @moonchor.make_local_backend([server, client, backup1,
backup2])
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v4,
server))
```

```
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v4,
client))
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v4,
backup1))
  @toolkit.run_async(() => @moonchor.run_choreo(backend, kvstore_v4,
backup2))
}
```

结语

真是一趟奇妙的旅程！在这篇文章中，我们借助 moonchor 体验了协同式编程的魅力，还看到了 MoonBit 强大的表达能力为我们带来的无限可能性。关于协同式编程的更多细节，可以参考 Haskell 的库 HasChor 和 Choral 语言 或者 moonchor 的源码。想要自己尝试使用 moonchor，可以通过 `moon add Milky2018/moonchor@0.15.0` 命令安装。

（本文定价5元）