QCon 全球软件开发大会
INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

BEIJING 2017

单页应用的数据流方案探索

徐飞

主办方 Geekbang. 极客邦科技 InfoQ

关于我

- 徐飞
- 民工精髓V
- 民工叔
- 蚂蚁金服体验技术部

现代前端框架

- 组件化
- **MDV**
- 语法糖
- 工具体系

Model Driven View

给定一个数据模型，可以得到对应的的视图

$$V = f(M)$$

当数据模型产生变化的时候，其对应的视图也会随之变化

$$V + \Delta V = f(M + \Delta M)$$

数据模型不是无缘无故变化的，它是由某个操作引起

$$\Delta M = perform(action)$$

整个应用等于初始状态叠加之后所有的操作结果

```
state := actions.reduce(reducers, initState)
```

F & RP 简介

传统编程模式

在传统的编程实践中，我们可以：

- 复用一种数据
- 复用一个函数
- 复用一组数据和函数的集合

但是，很难做到：

提供一种会持续变化的数据让其他模块复用。

可观察的数据封装

基于Reactive Programming的库可以提供一种能力，把数据包装成可持续变更、可观测的类型，供后续使用，这种库包括：RxJS，xstream，most.js等等

```
const a$ = xs.of(1)
const arr$ = xs.from([1, 2, 3])
const interval$ = xs.periodic(1000)
```

订阅这种数据，就可以持续得到每次变更的响应：

```
interval$.subscribe(console.log)
```

数据的流式处理

经过这样封装的数据，可以被视为管道。如果在管道上附加后续操作，可以作用到流过的所有数据。

```
const interval$ = xs.periodic(1000)
const result$ = interval$
   .filter(num => num % 3)
   .map(num => num * 2)
```

管道可被连续拼接，并形成新的管道。

数据管道的组合

可以把若干个数据管道用不同方式组合起来，得到新的数据管道：

```
const priv$ = xs.combine(user$, article$)
  .map(arr => {
    const [user, article] = arr
    return user.isAdmin || article.creator === user.id
  })
```

从这个关系中可以看出，当user$或task$中的数据发生变更的时候，priv$都会自动计算出最新结果。

更高层次的抽象

可以使用数据管道对业务进行更高层次的抽象：

```
const data$ = xs.fromPromise(service(params))
  .map(data => ({ loading: false, data }))
  .replaceError(error => xs.of({ loading: false, error }))
  .startWith({
    loading: true,
    error: null,
  })
```

这段代码处理了请求过程中的loading、正常数据和异常。

数据来源和变更的抽象

初始数据状态的来源

- 应用的默认配置
- HTTP请求
- 本地存储
- ...等等

不管来自哪里，都可以合并在一起

对初始状态的归纳

```
const fromInitState$ = xs.of(todo)
const fromLocalStorage$ = xs.of(getTodoFromLS())

// initState
const init$ = xs
  .merge(
    fromInitState$,
    fromLocalStorage$
  )
  .filter(todo => !todo)
  .startWith({})
```

状态变更的来源

- 用户从视图发起的操作
- 来自WebSocket的推送消息
- 来自Worker的处理消息
- 来自其它窗体的postMessage调用
- ...等等

不管从哪里发起的，只要修改的是同样的数据，都可以归纳到一起

对变更来源的归纳

```
const changeFromHTTP$ = xs.fromPromise(getTodo())
  .map(result => result.data)
const changeFromDOMEvent$ = xs.fromEvent($('.button'), 'click')
  .map(evt => evt.data)
const changeFromWebSocket$ = xs.fromEvent(ws, 'message')
  .map(evt => evt.data)

// 所有变更统一到这里
const changes$ = xs
  .merge(
    changeFromHTTP$,
    changeFromDOMEvent$,
    changeFromWebSocket$
  )
```

对变更结果的消费

```
// 简单的使用
changes$.subscribe(({ payload }) => {
  xxx.setState({ todo: payload })
})

// 类似Redux的使用方式
const changeActions$ = changes$
  .map(todo => ({type: 'updateTodo', payload: todo}))

const todo$ = changeActions$
  .fold((acc, val) => {
    const { payload } = val
    return {...acc, ...payload}
  }, initState)
```

组件的状态管理

组件间的通讯

整个系统组件化之后，会产生一棵组件树。

- 上级组件可以给下级组件设置数据（`props`）
- 上级组件可以给下级组件设置回调
- 可以在多个层级之间逐级传递

距离较远的组件之间通讯，传递过程很繁琐，因此，一般会引入一些全局转发机制。

外置的状态

组件树需要依赖于外部机制做数据中转，并且有如下事实：

- 转发器在组件树之外
- 部分数据在组件树之外
- 对这部分数据的修改过程在组件树之外
- 修改完数据之后，通知组件树更新

产生如下推论：

- 组件可以通过中转器修改其他组件的状态
- 组件可以通过中转器修改自身的状态
- 组件可以通过中转器修改全局的其他状态

是否应当外置所有状态?

两种组件状态管理思路

蚁群：依赖于强大的主控，个体弱小
一切状态外置，组件不管理自己状态，比较理想化。
优势：整个应用的状态可预测
劣势：抽象成本高；不必要的数据进入全局，让store变得很复杂

人群：个体自主权较大
部分内置，由组件自己管理，另外一些由全局store管理，现实的妥协
优势：土洋结合，容易开发
劣势：部分状态不受控，热替换存在缺陷

Local State is Fine.

- Dan Abramov

组件对状态的管理

```
constructor(props) {
  super(props)
  this.state = { b: 1 }
}

render(props) {
  const a = this.state.b + props.c;
  return (<div>{a}</div>)
}
```

如何结合内外状态?

```
const mapStateToProps = (state: { a }) => {
  return { a }
}

const localState = { b: 1 }
const mapLocalStateToProps = localState => localState

const ComponentA = (props) => {
  const { a, b } = props
  const c = a + b
  return (<div>{ c }</div>)
}

return connect(mapStateToProps, mapLocalStateToProps)(ComponentA)
```

MVI架构

MVI

```
App := View(Model(Intent({ DOM, Http, WebSocket })))
```

- Intent，负责从外部的输入中，提取出所需信息
- Model，负责从Intent生成视图展示所需的数据
- View，负责根据视图数据渲染视图

MVI的理念

- 一切都是事件源
- 使用Reactive的理念构建程序的骨架
- 使用sink来定义应用的逻辑
- 使用driver来隔离有副作用的行为（网络请求、DOM渲染）

MVI的开发过程

- 从driver中获取action
- 把action映射成数据流
- 处理数据流，并且渲染成界面
- 从界面的事件中，派发action去进行后续事项的处理

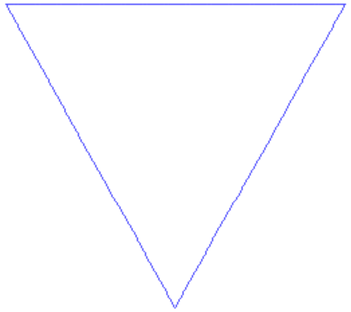
全局状态与本地状态

MVI的优势在于提供了一种清晰的处理过程，把组件之外的全局状态和组件内部状态融合在一起：

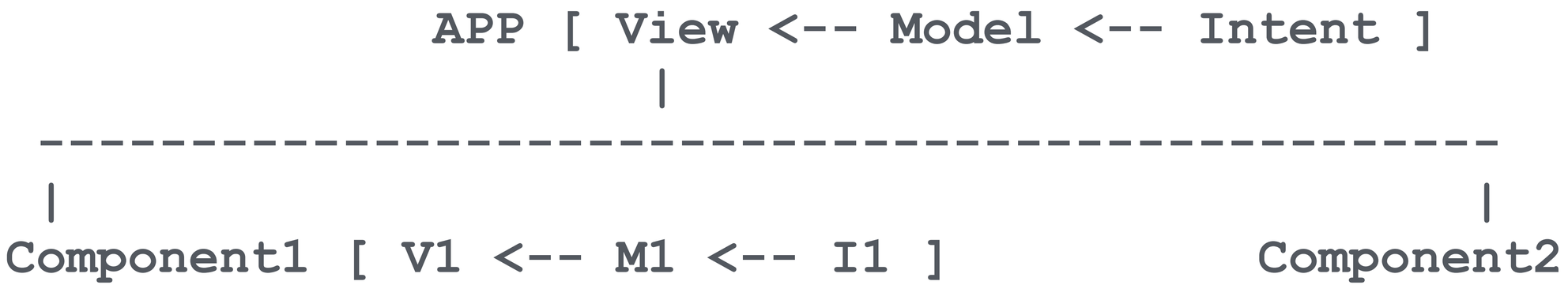
```
state$ = xs.combine(
  props,
  localState$
)
```

这样，组件的展示部分就可以纯化，并且，组件内部逻辑与展示部分的距离很近。

分形



分形之后的MVI



状态的变更过程

对数据管道的reduce操作

```
const updateActions$ = changes$
   .map(todo => ({type: 'updateTodo',
payload: todo}))

const todo$ = updateActions$
   .fold((state, action) => {
     const { payload } = action
     return {...state, ...payload}
   }, initState)
```

Redux的单Store

全局有一个大state对象用于描述状态

- 某个reducer修改了state的某个分支
- combineReducer把修改结果合并到state上
- react-redux拿到新的state去更新视图

reducer在修改数据的时候，是精确知道修改了state上面什么数据的，但被combineReducer合并结束之后，再也不知道了，只知道整个state变了。

合并之后统一reduce

```
// 我们可以先把这些action全部merge之后再fold，跟Redux的理念类似
const actions = xs.merge(
  addActions$,
  updateActions$,
  deleteActions$
)

const localState$ = actions.fold((state, action) => {
  switch(action.type) {
    case 'addTodo':
      return addTodo(state, action)
    case 'updateTodo':
      return updateTodo(state, action)
    case 'deleteTodo':
      return deleteTodo(state, action)
  }
}, initState)
```

先单独reduce再合并

```
// 如果数据没有相关性，可以分别reduce之后再合并
const a$ = actionsA$.fold(reducerA, initA)
const b$ = actionsB$.fold(reducerB, initB)
const c$ = actionsC$.fold(reducerC, initC)

const state$ = xs.combine(a$, b$, c$)
   .map(([a, b, c]) => ({a, b, c}))

// 如果足够简单，可以省去action
const state$ = xs.fromEvent($('.btn'), click)
   .map(e => e.data)
```

按照模型分类的数据流

```
// 按照业务模型组合的reducer
const projectModel$ = xs
  .combine(addProjectReducer$, editProjectReducer$)
const articleModel$ = xs
  .combine(addArticleReducer$, editActicleReducer$)

// 修改整个state的reducer
const state$ = xs
  .combine(projectModel$, articleModel$)
```

完整的变更过程

状态的分组与管理

面向领域模型的分组

以下是DVA的一个Model的定义（VueX也类似）

```
export const project = {
  namespace: 'project',
  state: {},
  reducers: {},
  effects: {},
  subscriptions: {}
}
```

MobX的Store

```
class TodoStore {
  authorStore

  @observable todos = []
  @observable isLoading = true

  constructor(authorStore) {
    this.authorStore = authorStore
    this.loadTodos()
  }

  loadTodos() {}
  updateTodoFromServer(json) {}
  createTodo() {}
  removeTodo(todo) {}
}
```

State里面如何存储数据

- 按照原始数据存储
  - 视图使用的时候常常需要作转换
- 按照视图的形态存储
  - 存储的时候可能有冗余
  - 容易出现不一致
- 如果兼而有之
  - 同步是个大问题

Normalizr对数据的处理

原始数据：

```
[{
  id: 1,
  title: 'Some Article',
  author: {
    id: 1,
    name: 'Dan'
  }
}, {
  id: 2,
  title: 'Other Article',
  author: {
    id: 1,
    name: 'Dan'
  }
}]
```

处理之后：

```
{
  result: [1, 2],
  entities: {
    articles: {
      1: {
        id: 1,
        title: 'Some Article',
        author: 1
      },
      2: {
        id: 2,
        title: 'Other Article',
        author: 1
      }
    },
    users: {
      1: {
        id: 1,
        name: 'Dan'
      }
    }
  }
}
```

数据加工的可选位置

数据的流动过程

- 网络传输
- 范式化（normalizr）
- 存入state
- 流入视图组件，触发视图刷新
- 组合、格式化，变成视图数据
- 渲染

可复用的数据和计算过程

```
const list$ = xs.from(arr)
const tree$ = list$.map(listToTree)

list$.subscribe(/* 以列表形式展示 */)
tree$.subscribe(/* 以树的形状展示 */)
```

可以同时具有不同形态的数据管道，只需控制好它们的转换关系

前端的ORM

```
class Todo extends Model {}
Todo.modelName = 'Todo'
Todo.fields = {
  user: fk('User', 'todos'),
  tags: many('Tag', 'todos'),
}

class Tag extends Model {}
Tag.modelName = 'Tag';
Tag.backend = {
  idAttribute: 'name'
}

class User extends Model {}
User.modelName = 'User'
```

分离对State的读、写

对State的读写操作实际上是分离的：

- 通过action负责写入
- 通过模型数据管道组合订阅来读取

在此基础上，可以继续深入：

- 写入操作的原子化
- 读取操作的响应化
- 与持久存储的结合
- 需要控制好时序关系

小结

- 不需要显式定义整个应用的state结构
- 全局状态和本地状态可以良好地统一起来
- 可以存在非显式的action，并且action可以不集中解析，而是分散执行
- 可以存在非显式的reducer，它附着在数据管道的运算中
- 异步操作先映射为数据，然后通过单向联动关系组合计算出视图状态
- 数据的写入和读取过程分离

Reactive库的优势

- 对事件的高度抽象
- 同步和异步的统一化处理
- 数据变更的持续订阅（订阅模式）
- 精确更新
- 数据的连续变更（管道拼接）
- 数据变更的的组合运算（管道组合）
- 懒执行（无订阅者，则不执行）
- 缓存的中间结果
- 可重放的历史记录
- ……等等

本文地址：

https://zhuanlan.zhihu.com/p/26426054

关注QCon微信公众号，
获得更多干货！

Thanks!

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

主办方 Geekbang. 极客邦科技 InfoQ