ctrip tech

携程技术中心

携程技术沙龙

Mvvm 前端数据流框架精讲

分享人：黄子毅

讲师介绍

黄子毅

数据流框架 dob
可视化建站 gaea-editor
等开源框架作者

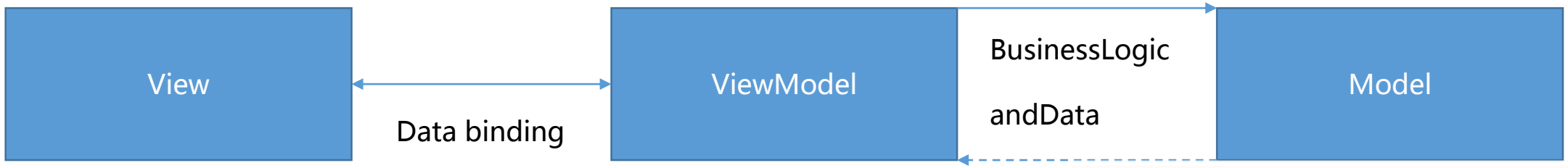
前端精读创刊人

目录

CONTENTS

1 Mvvm 的概念与发展

Mvvm & 单向数据流

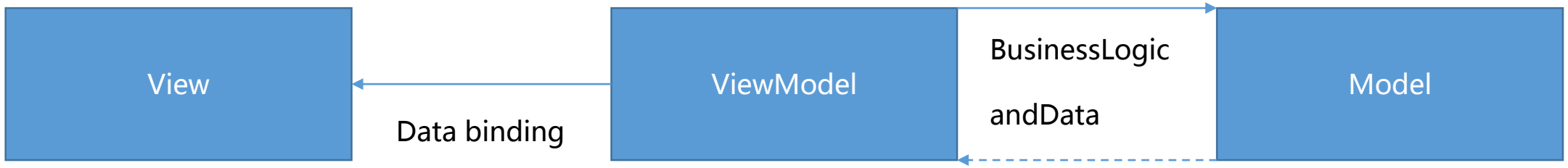



```
<input v-model="message">
```

```
Connect(stores)(View)
```

```
class Store {
    message = "hello world"
}
```

Mvvm & 单向数据流

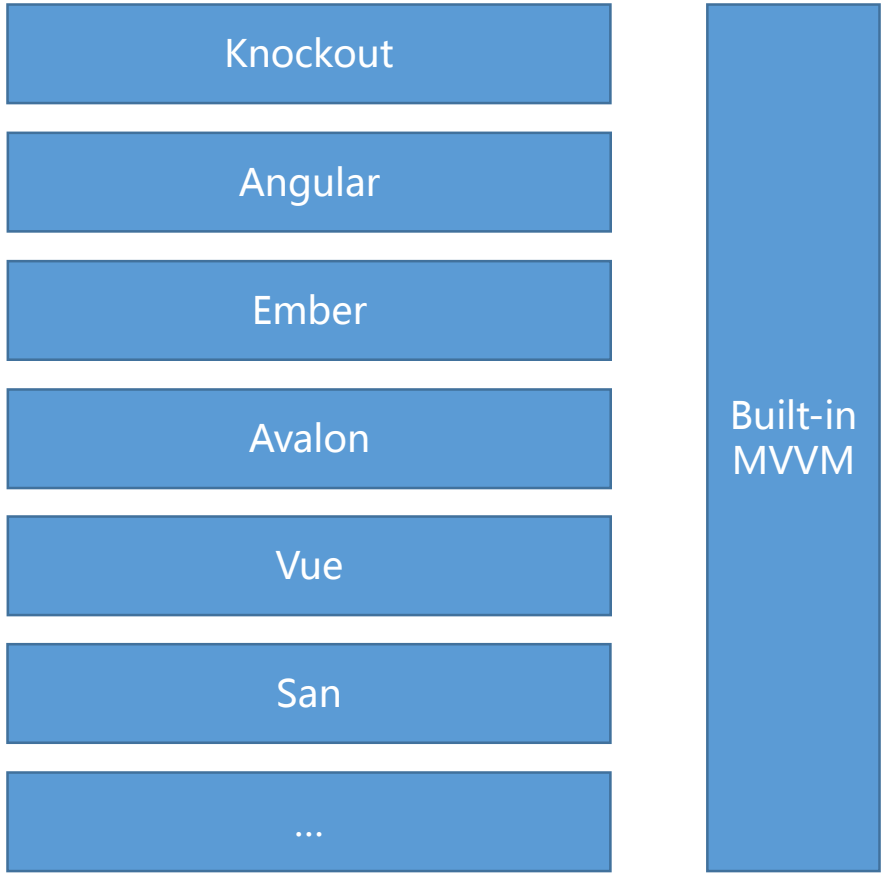



```
export default () => <div />
```

```
Connect(stores)(View)
```

```
class Store {
    articles = []
}
```

生态：内置 -> 解耦

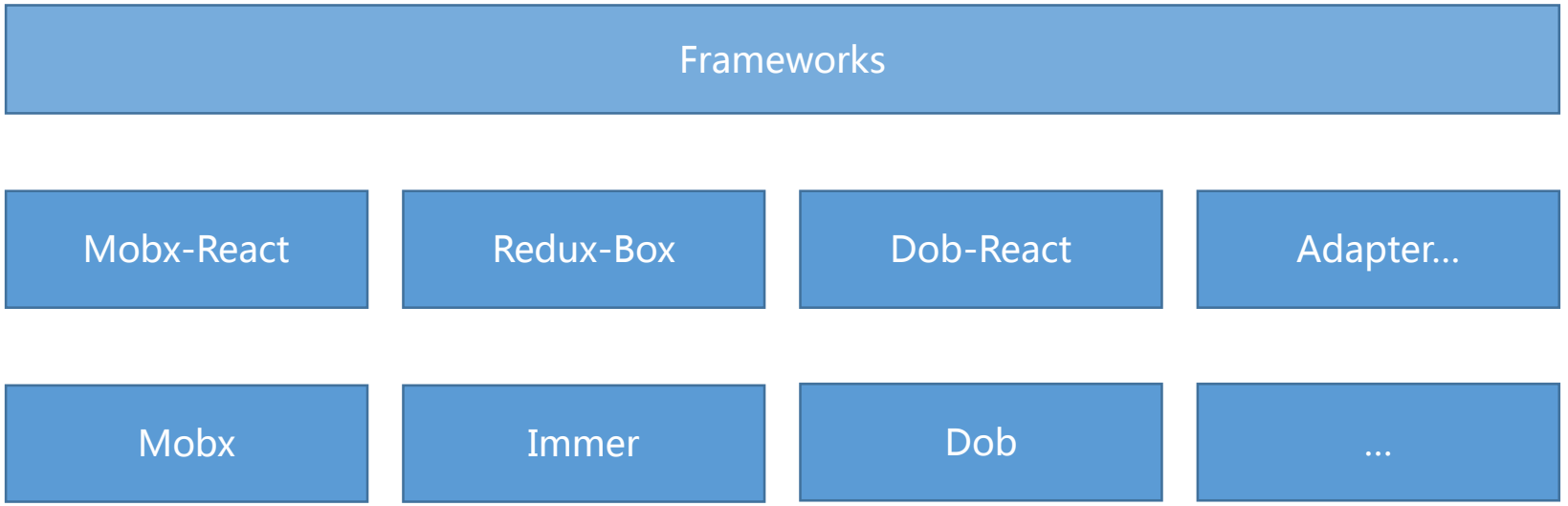


生态：内置 -> 解耦

```html
<div id="app-5">
  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">Reverse Message</button>
</div>
```

```js
var app5 = new Vue({
  el: '#app-5',
  data: {
    message: 'Hello Vue.js!'
  },
  methods: {
    reverseMessage: function () {
      this.message = this.message.split('').reverse().join('')
    }
  }
})
```

生态：内置 -> 解耦

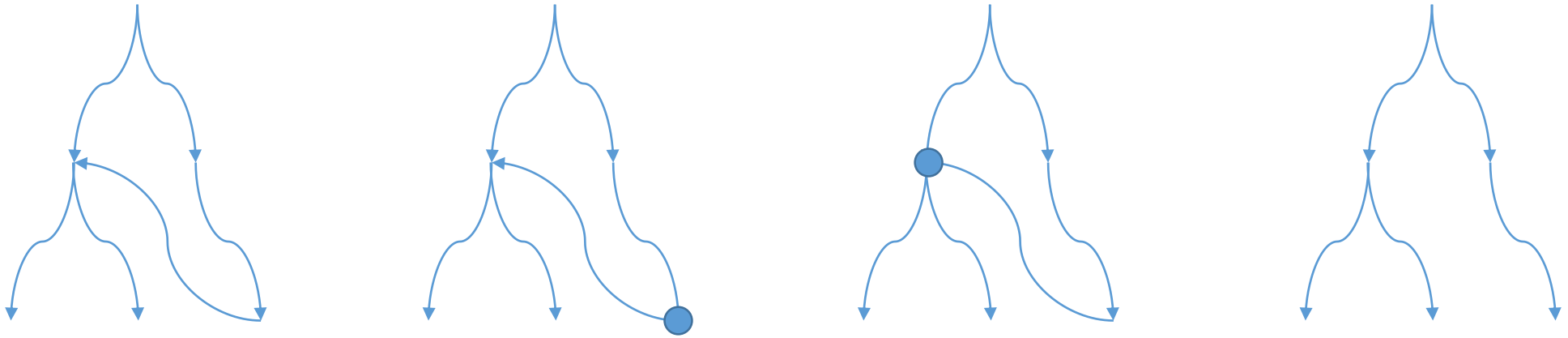

Frameworks
Mobx-React
Redux-Box
Dob-React
Adapter...
Mobx
Immer
Dob
...

生态：内置 -> 解耦

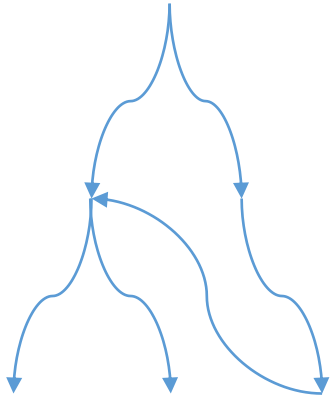
```
import { observable, combineStores, inject } from 'dob'

@observable
class Store {
    name = 'bob'
}
class Action {
    @inject(Store) store

    setName () {
        this.store.name = 'lucy'
    }
}

export stores = combineStores({ Action, Store })
```

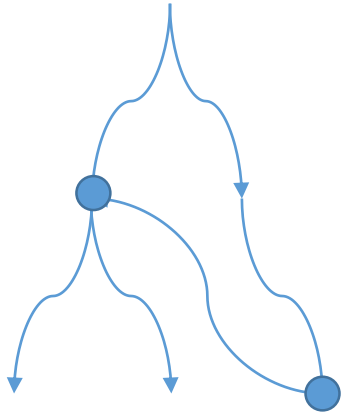
```
import * as React from "react"
import { Connect } from "dob-react"

@Connect(stores)
export default class Page extends React.PureComponent {
    render() {
        return (
            <div
                onClick={this.props.Action.setName}
            >{this.props.Store.name}</div>
        )
    }
}
```

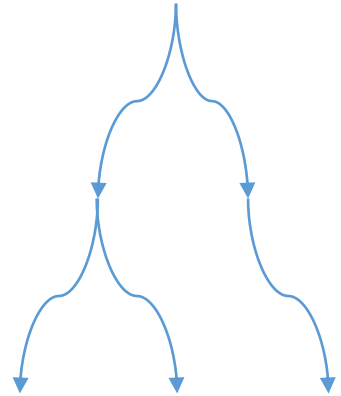
效率：脏检测 -> getter 劫持



效率：脏检测 -> getter 劫持







语法：函数调用 -> 原生

```
<div ng-controller="Counter">
 <span ng-bind="counter"></span>
 <button ng-click="counter=counter+1">increase</button>
</div>
```

```
scope.age = 5
scope.$apply()
```

```
function Counter($scope) {
 $scope.counter = 1;
}
```

语法：函数调用 -> 原生

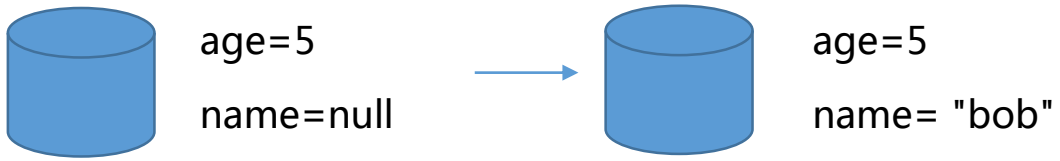
```
class Action {
    doSomeThing() {
        this.store.name = "bob"
    }
}
```

数据变更方式：Mutable -> Immutable

```
const mutations = {
  setAge() {
    this.state.age = 5
  }

  setName() {
    this.state.name = "bob"
  }
}
```

数据变更方式：Mutable -> Immutable

```
const mutations = {
  SET_NAME: (state, action) => (state.name = 5),
  SET_NAME: (state, action) => (state.name ="bob")
}
```



2 从 TFRP 到 mvvm

TFRP 驱动 mvvm - autorun

```
const obj = dynamicObject({
    value: 1
})

autoRun(() => {
    console.log(obj.value)
})
// 1

obj.value = 2
// 2
```

TFRP 驱动 mvvm - reaction

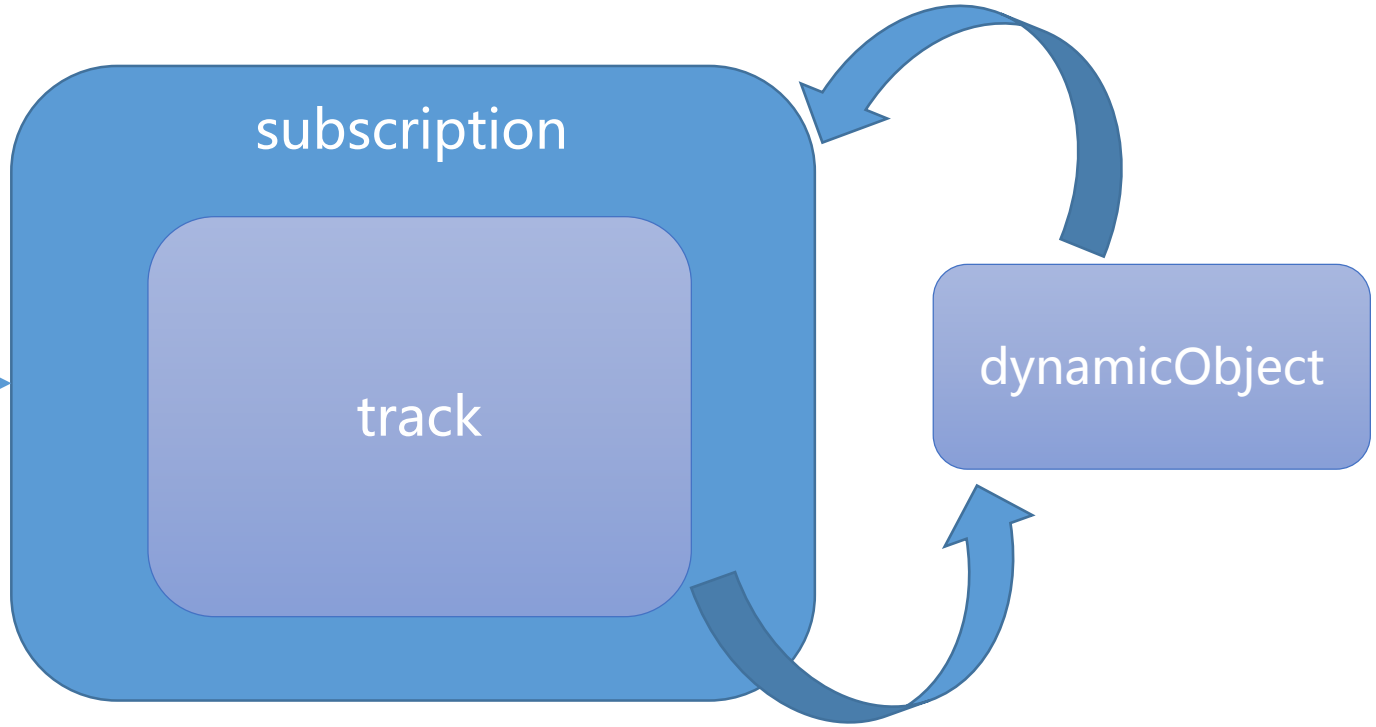
```
const obj = dynamicObject({
    value: 1
})

const reaction = new Reaction(() => {
  console.log("obj.value changed")
})

reaction.track(() => {
  const someThing = obj.value
})


obj.value = 2
// obj.value changed
```

TFRP 驱动 mvvm – reaction -> autorun

```
function observe(callback: Func) {
  const reaction = new Reaction(() => {
    reaction.track(callback)
  })

  reaction.run()

  return {
    unobserve: () => {
      reaction.dispose()
    }
  }
}
```

TFRP 驱动 mvvm – reaction -> autorun

```
function observe(callback: Func) {
  const reaction = new Reaction(() => {
    reaction.track(callback)
  })

  reaction.run()

  return {
    unobserve: () => {
      reaction.dispose()
    }
  }
}
```



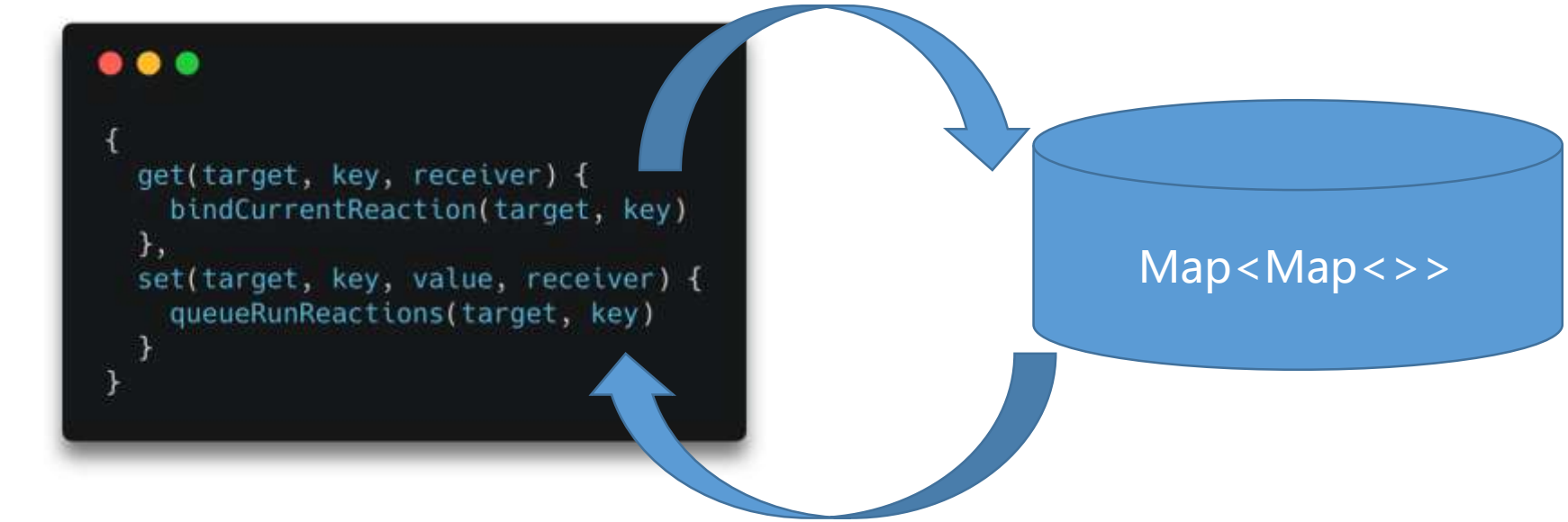
依赖追踪

```
autoRun(() => {
    obj.value
})
```

```
new Proxy(obj, {
  get() {
    ..
  },
  set() {
    ..
  }
})
```

```
Object.defineProperty(obj, "value", {
    get() {
      ..
    },
    set(value) {
      ..
    }
})
```

依赖追踪

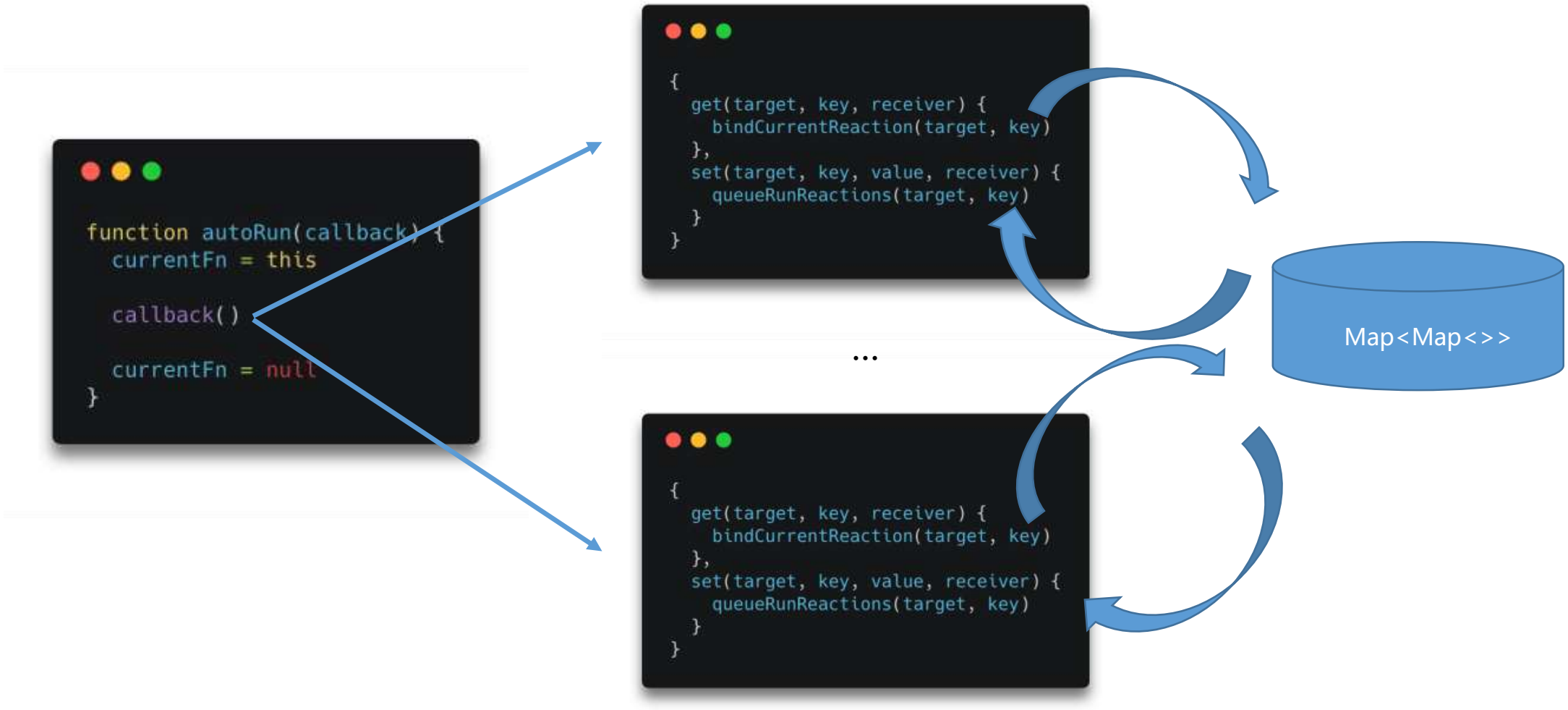

{
get(target, key, receiver) {
bindCurrentReaction(target, key)
},
set(target, key, value, receiver) {
queueRunReactions(target, key)
}
}
Map<Map<>>>

依赖收集

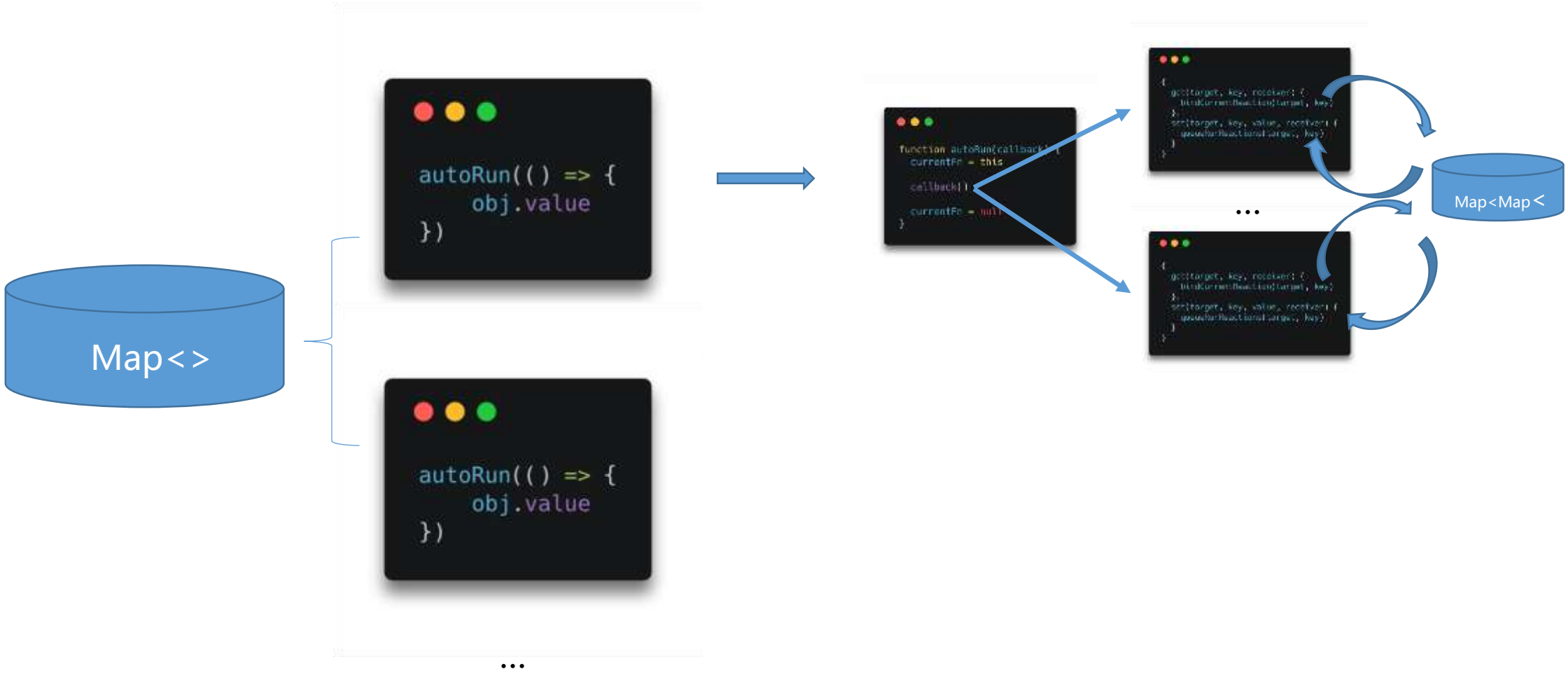
```
autoRun(() => {
    obj.value
})
```

```
function autoRun(callback) {
  currentFn = this

  callback()

  currentFn = null
}
```

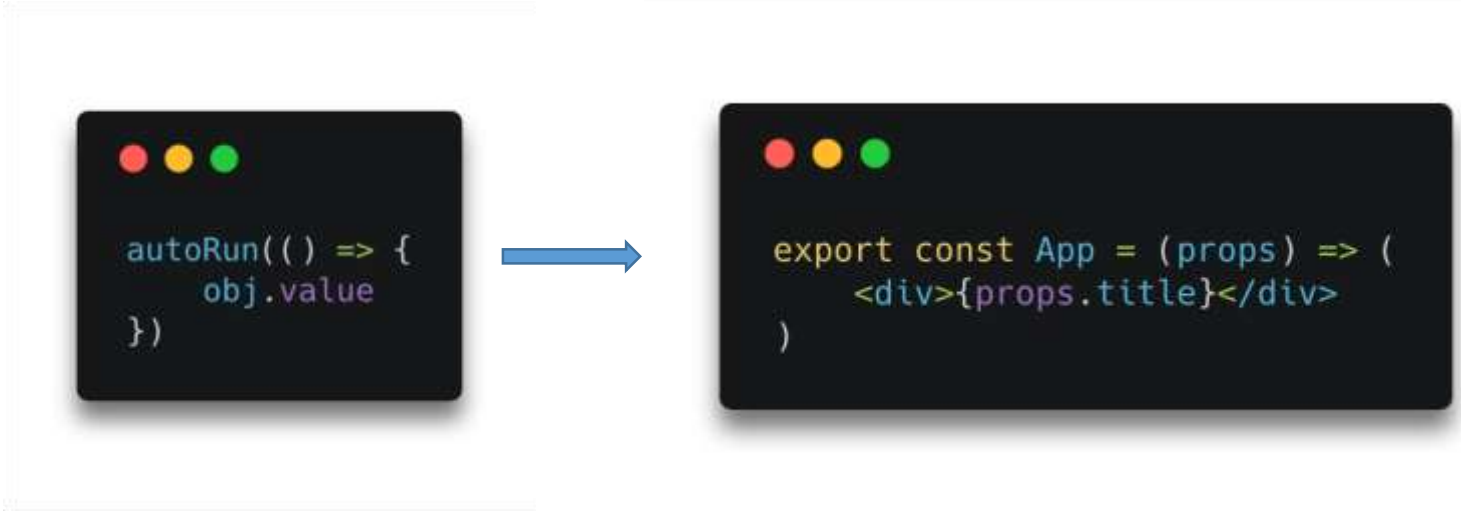
依赖收集


function autoRun(callback) {
currentFn = this
callback()
currentFn = null
}
{
get(target, key, receiver) {
bindCurrentReaction(target, key)
},
set(target, key, value, receiver) {
queueRunReactions(target, key)
}
}
...
{
get(target, key, receiver) {
bindCurrentReaction(target, key)
},
set(target, key, value, receiver) {
queueRunReactions(target, key)
}
}
Map<Map<>>

依赖收集



框架层结合



框架层结合

```
autoRun(() => {
    obj.value
})
```

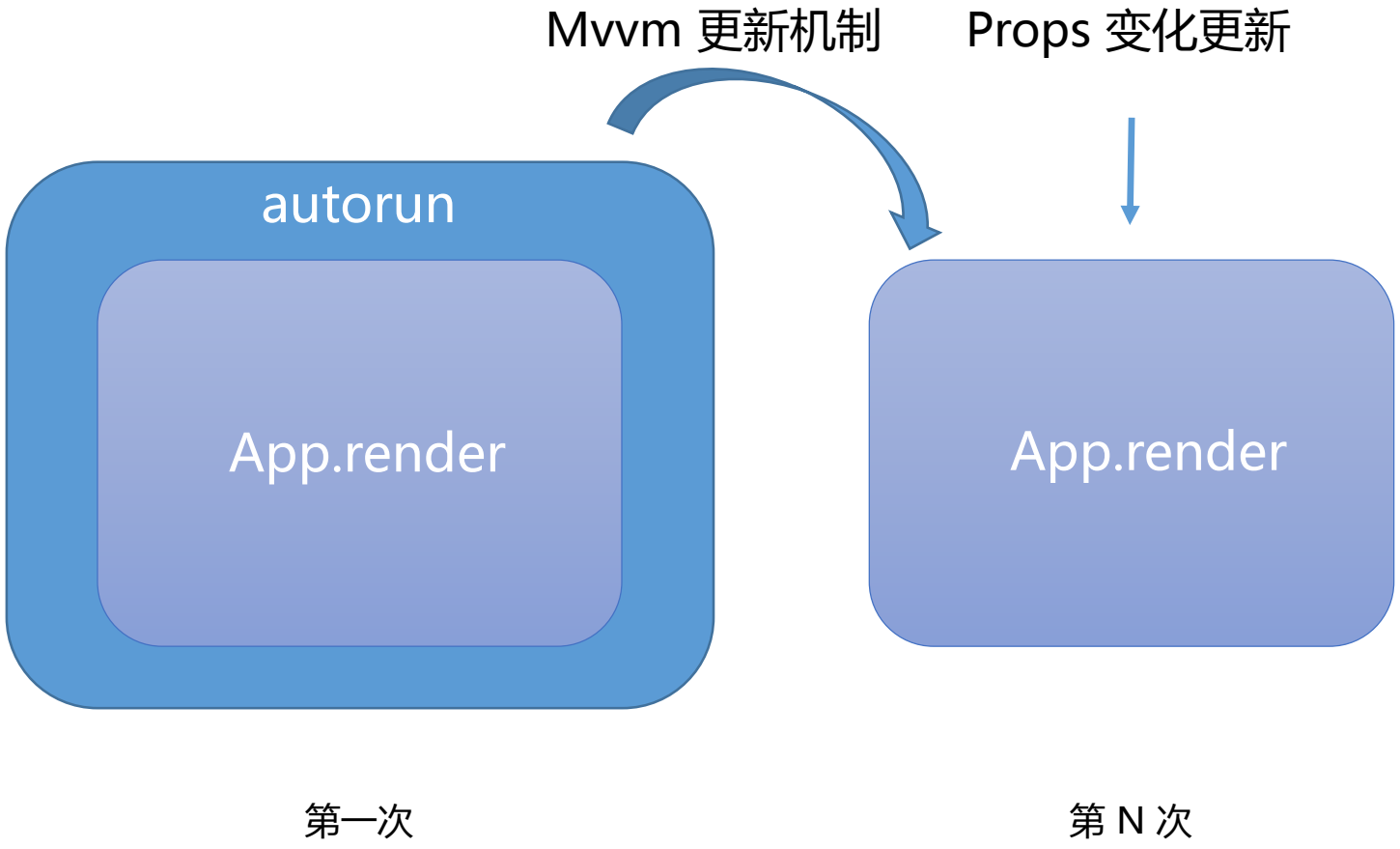
初始状态执行一次

```
export const App = (props) => (
    <div>{props.title}</div>
)
```

didMount

框架层结合

```
export default Connect()(App)
```



3 Mvvm 的缺点与解法？

监听范围局限

```
const vm = new Vue({
  data: {
    a:1
  }
})

vm.b = 2 // Not work
```

监听范围局限

```
Object.defineProperty(obj, "someKey", fn)
```



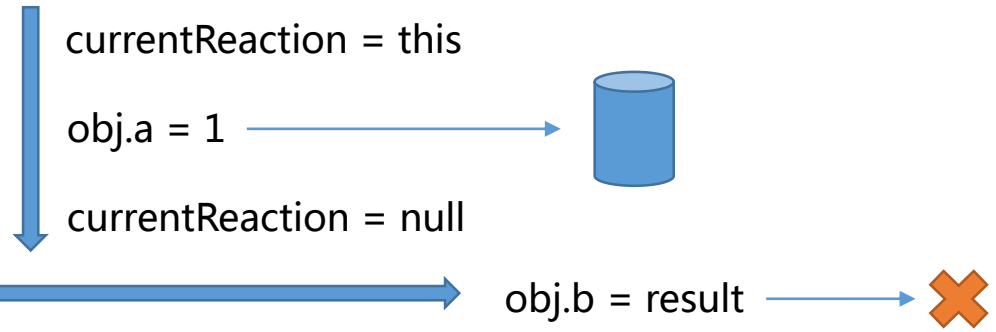
```
const dynamicObject = new Proxy(obj, {
  get(target, key) {}
})
```

异步问题

```
autorun(() => {
  obj.a = 1
  fetchUser().then(result => {
    obj.b = result // Not work
  })
})
```

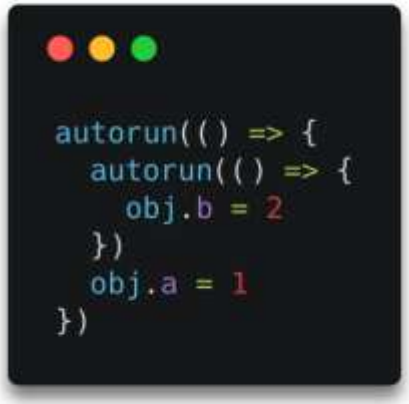
异步问题

```
autorun(() => {
  obj.a = 1
  fetchUser().then(result => {
    obj.b = result // Not work
  })
})
```



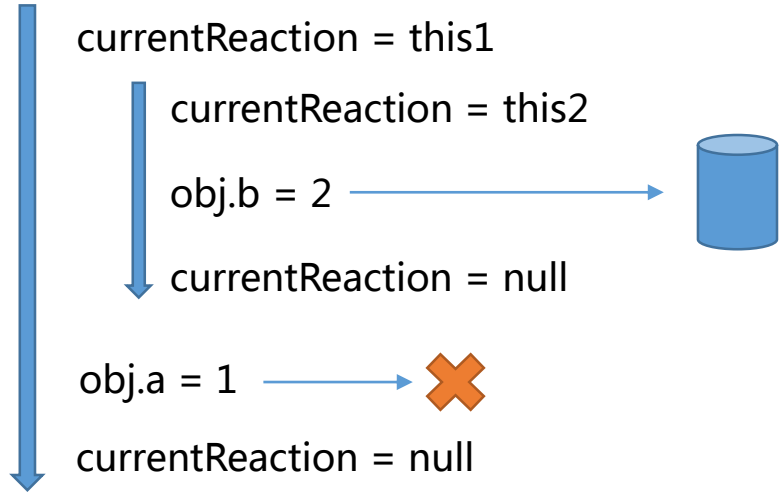
Render 同步

嵌套问题



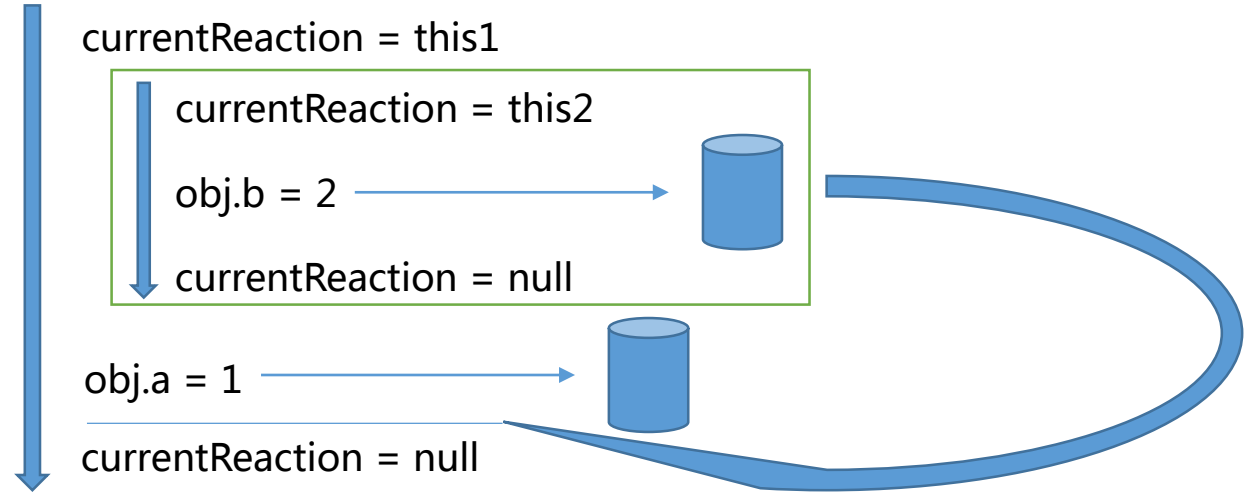
嵌套问题

```
autorun(() => {
  autorun(() => {
    obj.b = 2
  })
  obj.a = 1
})
```

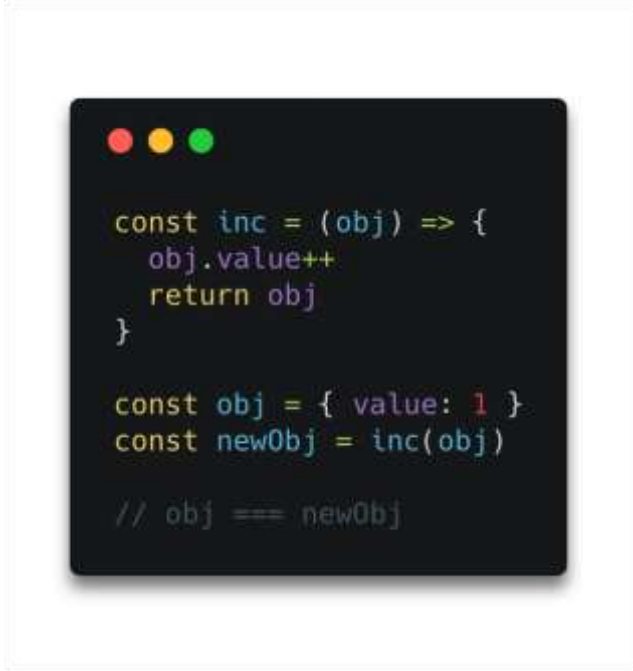


嵌套问题

```
autorun(() => {
  autorun(() => {
    obj.b = 2
  })
  obj.a = 1
})
```



无数据快照



```
const inc = (obj) => {
  obj.value++
  return obj
}

const obj = { value: 1 }
const newObj = inc(obj)

// obj === newObj
```

无数据快照

```
const inc = produce(draft => {
    draft.value++
})

const obj = { value: 1 }
const newObj = inc(obj)

// obj !== newObj
```

无数据快照


new Proxy(obj)
const newObj = inc(obj)
const inc = produce(draft => {
draft.value++
})
Immutable setter
newObject

无副作用隔离

```
app.model({
  namespace: 'products',
  state: ...,
  effects: {
    ['products/query']: async () {
      const result = await getProducts()
      await put({
        type: 'products/query/success',
        payload: result,
      })
    },
  },
  reducers: {
    ['products/query'](state) {...},
    ['products/query/success'](state, { payload }) {...},
  },
})
```

dva

无副作用隔离

```
class Products {
  @inject(Store) store

  async query() {
    const result = await getProducts()
    this.store.products = result
  }
}
```

未隔离

```
class Products {
  @inject(Store) store

  async query() {
    const result = await getProducts()
    this.productReducer(result)
  }

  productReducer(products) {
    this.store.products = products
  }
}
```

隔离

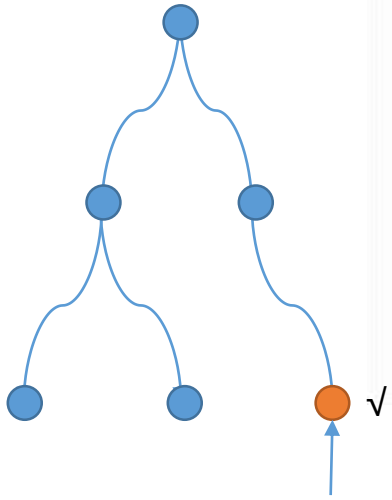
无副作用隔离

```
app.model({
  namespace: 'products',
  state: ...,
  effects: {
    ['products/query']: async () {
      const result = await getProducts()
      await put({
        type: 'products/query/success',
        payload: result,
      })
    },
  },
  reducers: {
    ['products/query'](state) {...},
    ['products/query/success'](state, { payload }) {...},
  },
})
```

```
app.model({
  namespace: 'products',
  state: ...,
  effects: {
    query: async () {
      const result = await getProducts()
      this.querySuccess(result) // 无法访问 state
    },
  },
  reducers: {
    query() { ... }
    querySuccess(products) {
      // 不支持异步
      this.state.products = products
    }
  },
})
```

Mutable 和 Immutable 在 Connect 的区别

```
reducers: {
  querySuccess(products) {
    this.state.products = products
  }
}
```

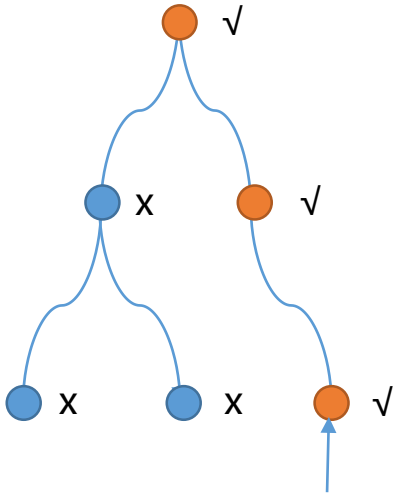



```
@Connect
class App {...}
```

Mutable

```
@Connect(state => ({
  products: state.products
}))
class App {...}
```

Immutable



4 Mvvm store 组织形式

对象形式，代表框架 – mobx

```
const store = observable({ products: [] })

const action = {
  getProducts: async () => {
    const result = await fetchProducts()
    store.products = result
  }
}
```

- 数据定义最简洁
- Typescript 类型支持

- 无强制副作用隔离

Class + 注入，代表框架 – dob

```
class Store {
  products = []
}

class Action {
  @inject(Store) store

  async getProducts() {
    const result = await fetchProducts()
    this.store.products = products
  }
}
```

- 灵活注入的能力
- Typescript 类型支持

- 无强制副作用隔离

数据结构化，代表框架 – mobx-state-tree

```
const Product = types.model("Product", {
  title: types.string
})

const Products = types.model("Products", {
  products: types.array(Product)
}).views(self => {
  return {
    async getProducts() {
      const result = await getProducts()
      self.products = result
    }
  }
})
```

- Store 之间产生关联结构
- Typescript 类型支持

- 需要记住一套约定
- 无强制副作用隔离

约定与集成，代表框架 – 类 dva

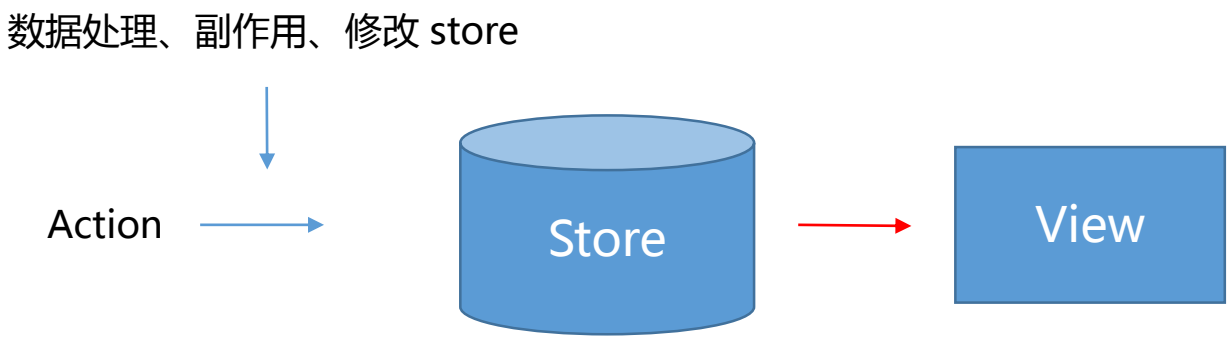
```
app.model({
  state: {
    products: []
  },
  actions: {
    getProducts: async() {
      const result = await fetchProducts()
      this.reducers.setProducts(result)
    }
  },
  reducers: {
    setProducts(result) {
      this.state.products = result
    }
  },
})
```

- 强制副作用隔离
- 数据结构扁平
- 高层抽象，便于底层实现的迭代
- Typescript 类型支持

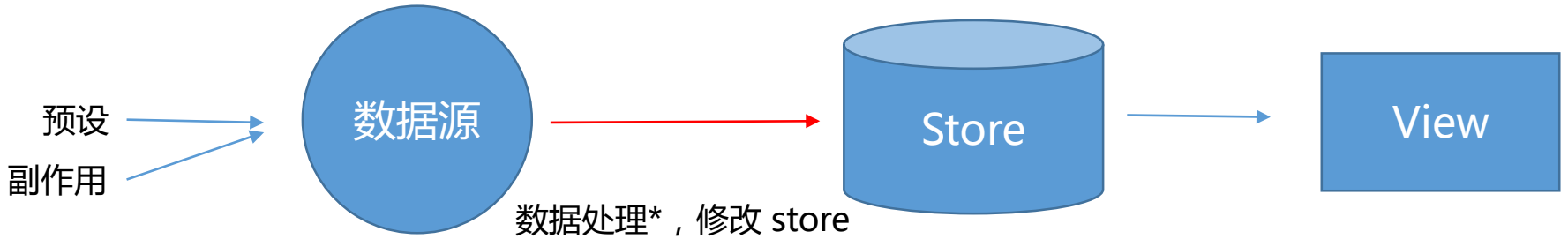
- 需要记住一套约定

5 Mvvm vs Reactive programming

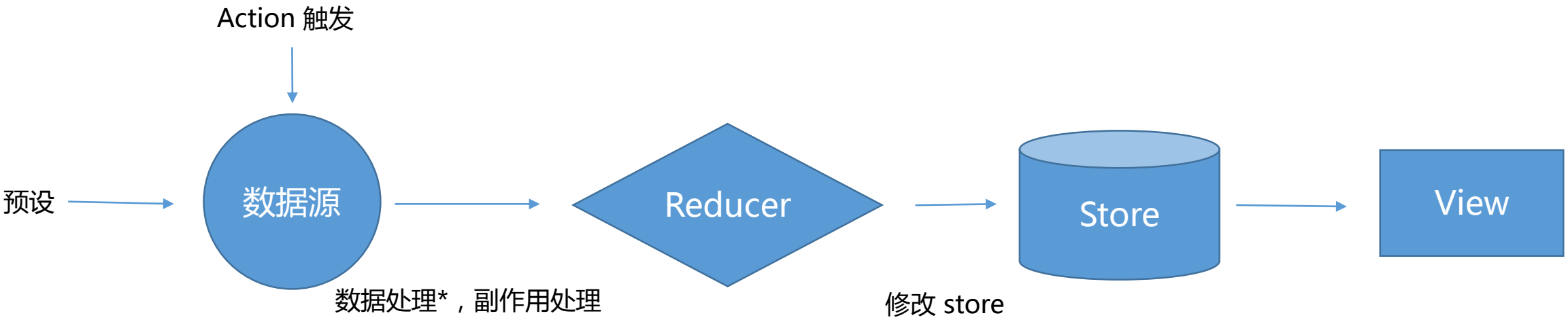
Observable(mvvm) & Observable(reactive)



Observable(mvvm) & Observable(reactive)



Mvvm 结合 Reactive



Mvvm 结合 Reactive

```
app.model({
  state: {
    products: []
  },
  actions: {
    getProducts: stream$ => stream$
      .mergeMap(value => fetchProducts(value))
      .mapTo(data => ({
        type: "setProducts",
        payload: data.products
      }))
  },
  reducers: {
    setProducts(data) {
      this.state.products = data.payload
    }
  },
})
```


黄子毅
北京 海淀
扫一扫上面的二维码图案，加我微信


ctrip tech
携程技术中心
THANK YOU!
Q&A
本PPT来自携程技术沙龙
更多信息，欢迎搜索关注“携程技术中心”微信公号~