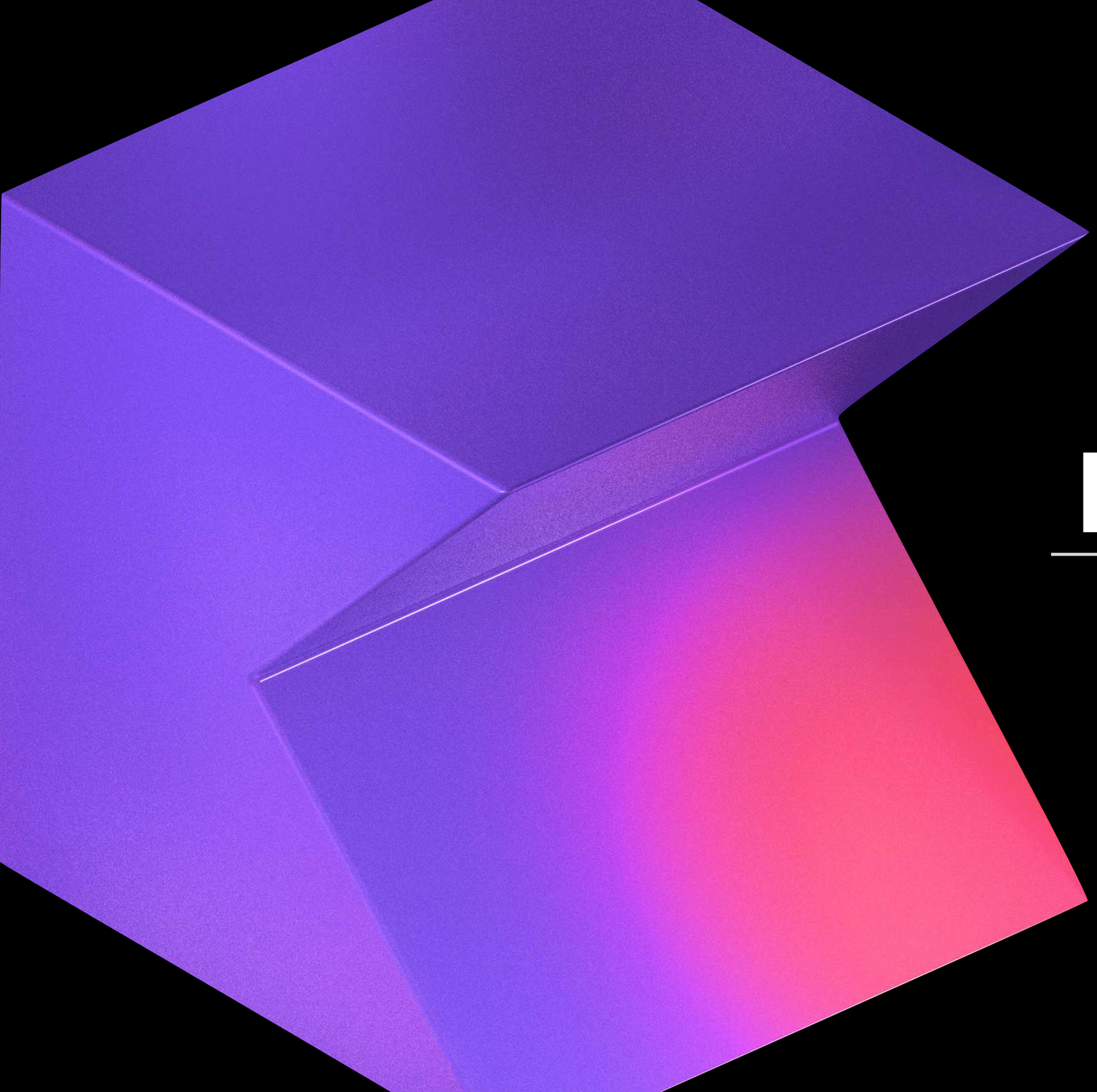


KMM MVI

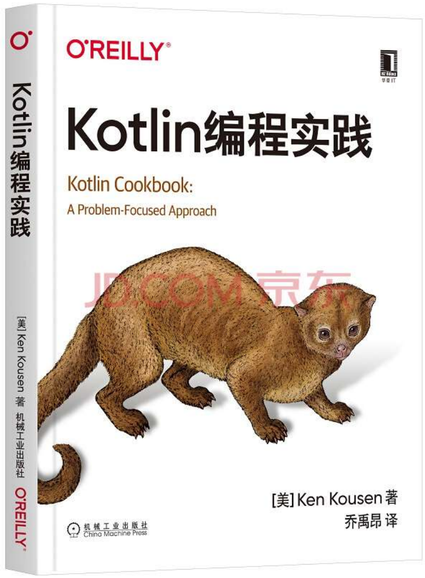
MVIKotlin

乔禹昂

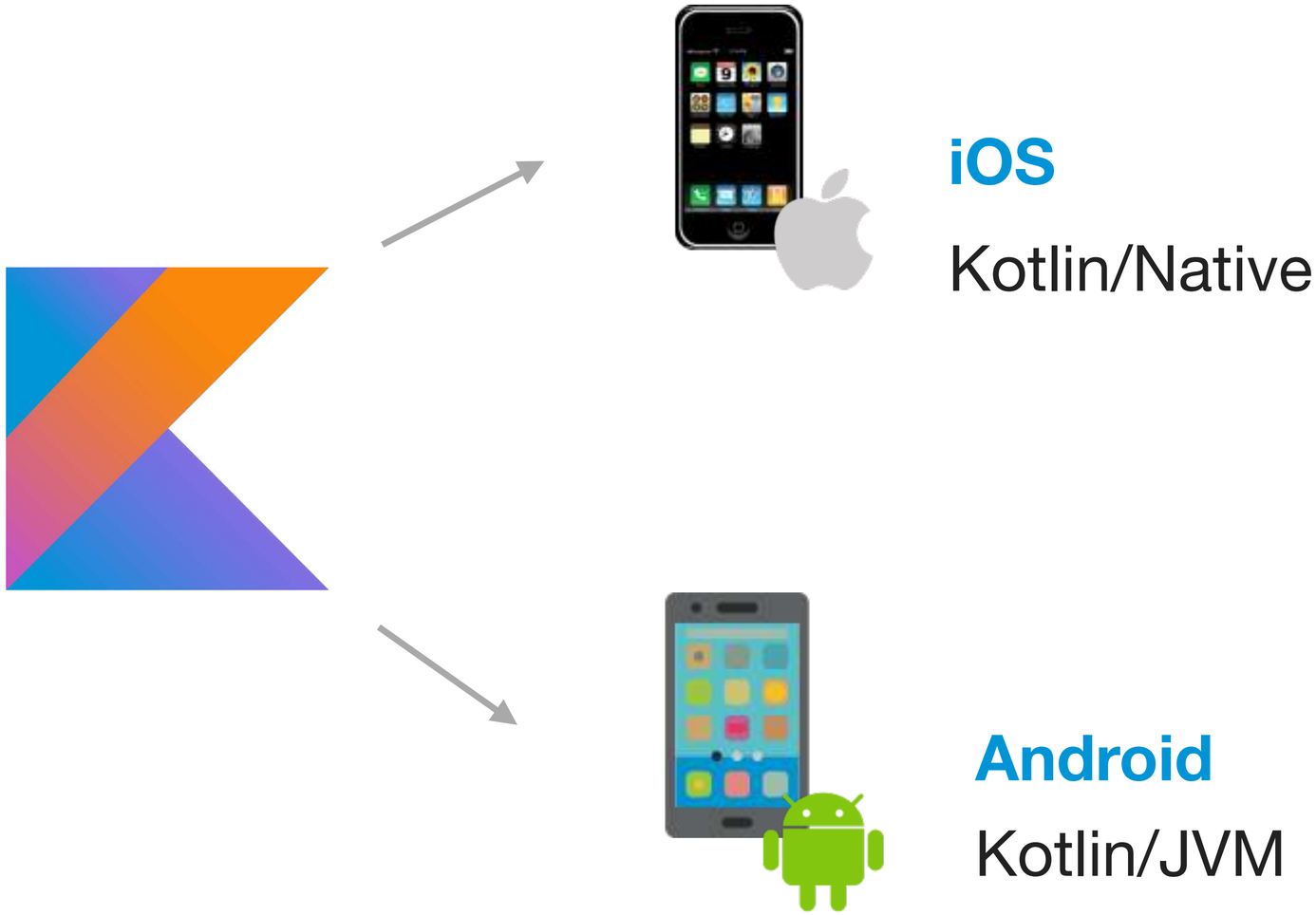
@KUG Shanghai

2021/11/06

- 携程机票资深移动开发工程师
- KUG Shanghai Organizer
- Kotlin Cookbook (O’Reilly, Kotlin 编程实践) 译者
- Kotlin 传教士



Kotlin Multiplatform Mobile



iOS

Kotlin/Native

Android

Kotlin/JVM

哪些公司在用 KMM?


Ctrip
携程


阿里巴巴
Alibaba.com


快手


Tencent 腾讯


美团


NETFLIX


vmware

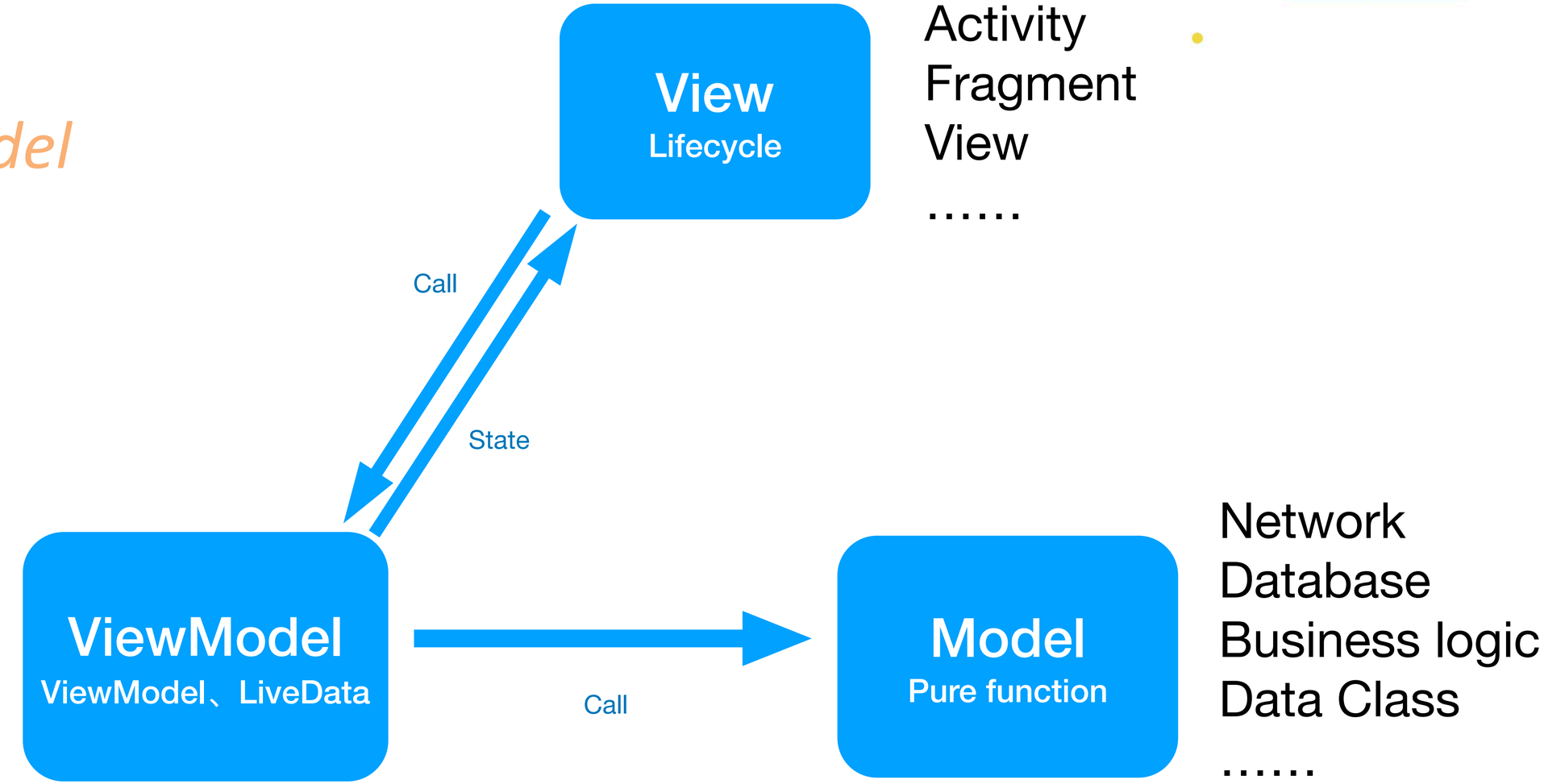

Square

KMM 生态环境

kotlinx.coroutines
kotlinx.serialization
kotlinx.datetime
……

Jetpack Android Architecture Components (MVVM)

Lifecycle
ViewModel
LiveData



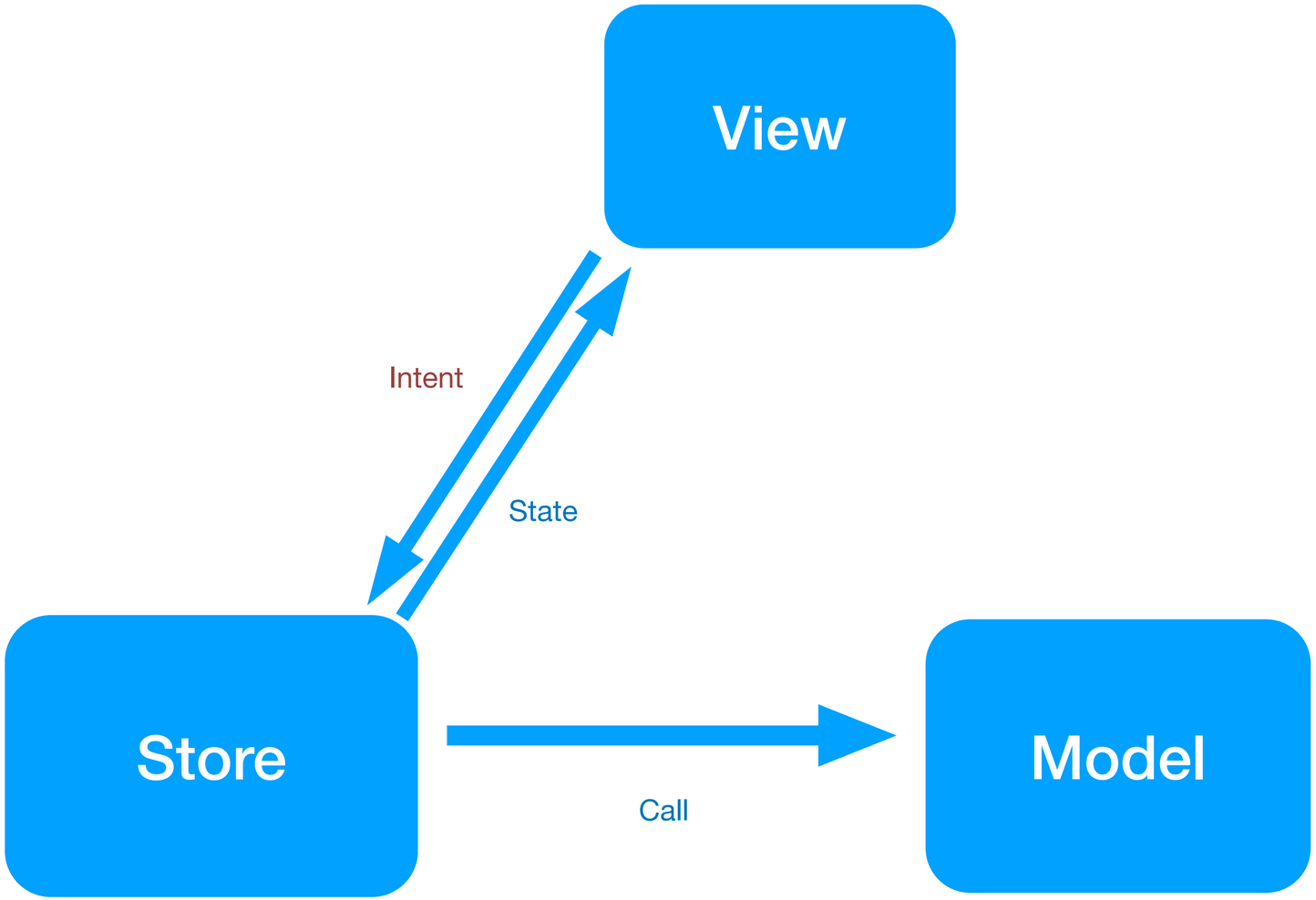
Activity
Fragment
View
……

Data status
Publisher
……

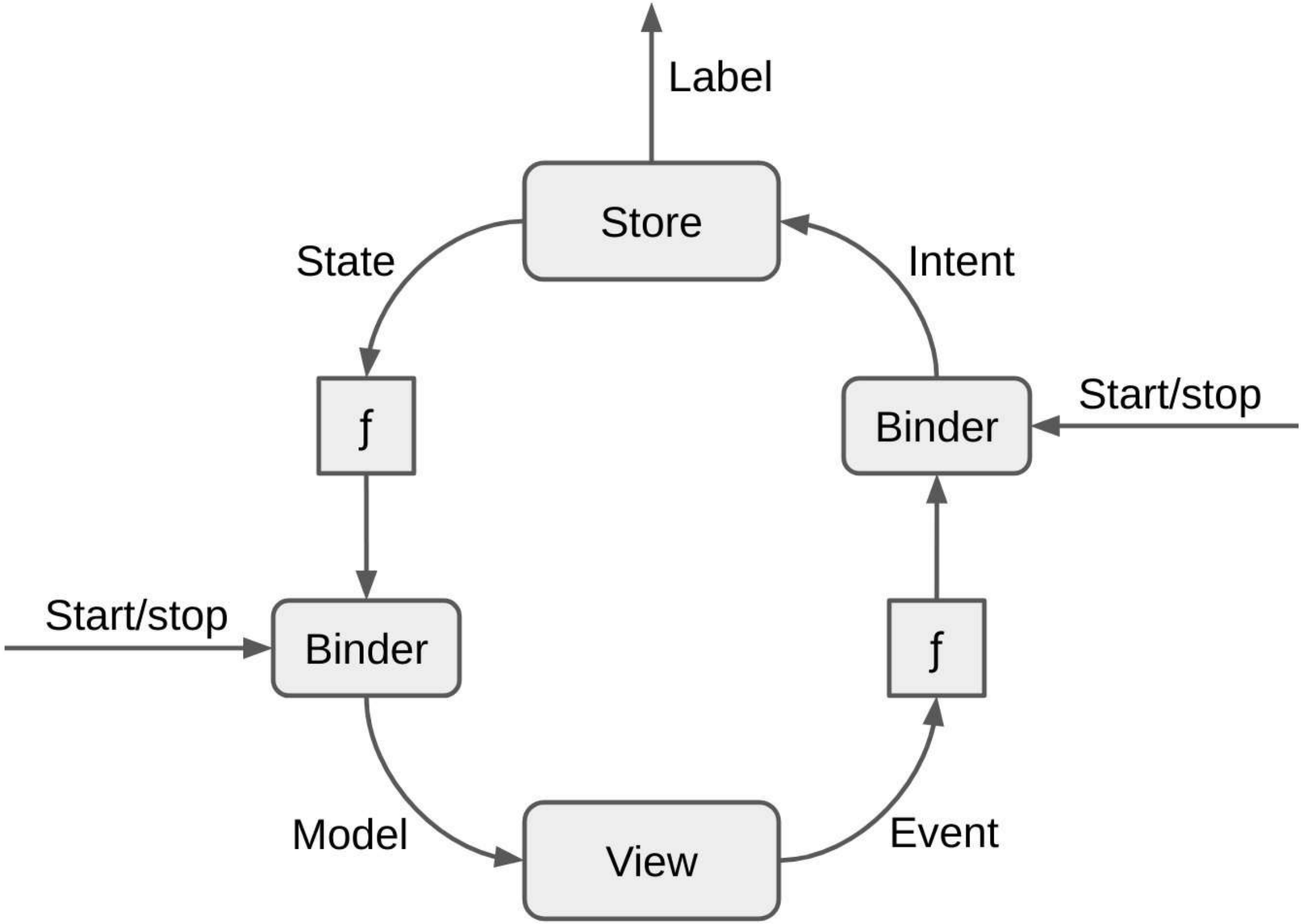
Network
Database
Business logic
Data Class
……

What is MVI?

MVIKOTLIN

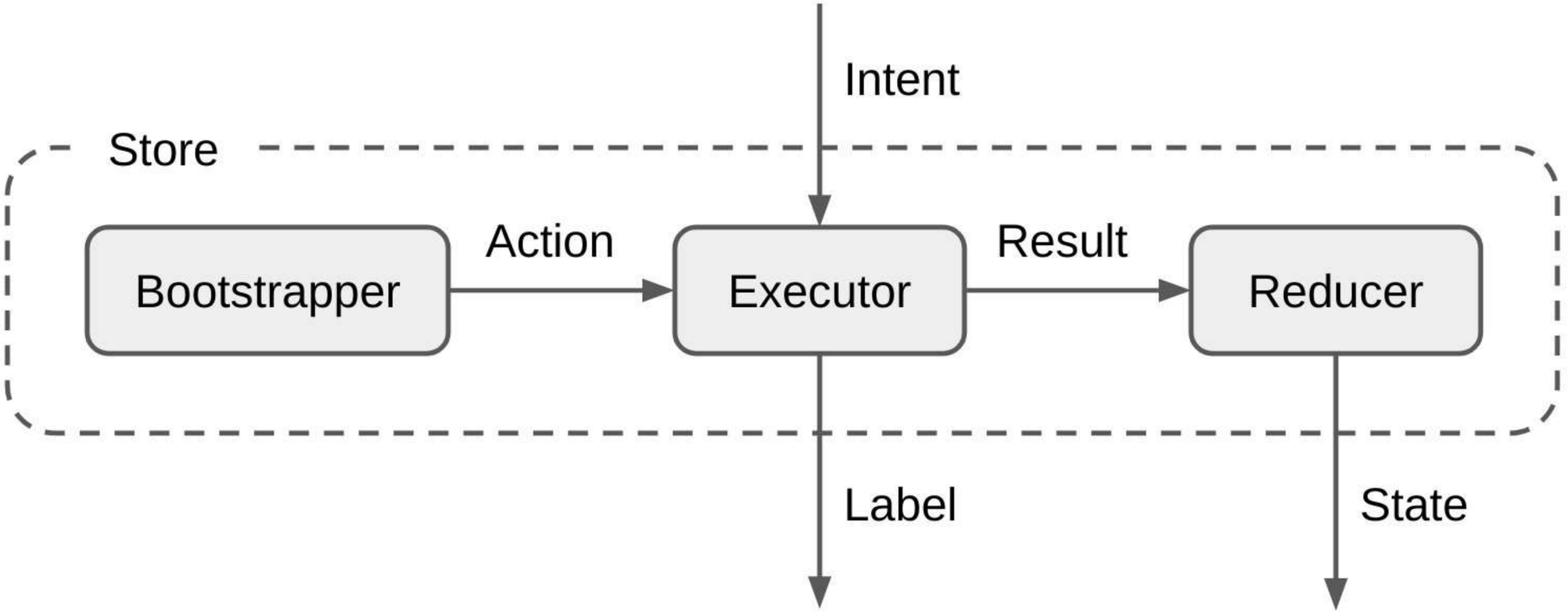


MVIKotlin



图片来源：https://arkivanov.github.io/MVIKotlin/

MVIKotlin Store



图片来源：https://arkivanov.github.io/MVIKotlin/

MVIKotlin Store


```
internal interface CalculatorStore : Store<Intent, State, Nothing> {
    sealed class Intent {
        object Increment : Intent()
        object Decrement : Intent()
        data class Sum(val n: Int): Intent()
    }
    data class State(
        val value: Long = 0L
    )
}
```

MVIKotlin Store Bootstrapper


```
private sealed class Action {
    class SetValue(val value: Long): Action()
}

private class BootstrapperImpl : CoroutineBootstrapper<Action>() {
    override fun invoke() {
        scope.launch { this: CoroutineScope
            val sum = withContext(Dispatchers.Default) { (1L..1000000.toLong()).sum() }
            dispatch(Action.SetValue(sum))
        }
    }
}
```

MVIKotlin Store Executor


```
private class ExecutorImpl : CoroutineExecutor<Intent, Action, State, Result, Nothing>() {
    override fun executeAction(action: Action, getState: () → State) =
        when (action) {
            is Action.SetValue → dispatch(Result.Value(action.value))
        }

    override fun executeIntent(intent: Intent, getState: () → State) =
        when (intent) {
            is Intent.Increment → dispatch(Result.Value( value: getState().value + 1))
            is Intent.Decrement → dispatch(Result.Value( value: getState().value - 1))
            is Intent.Sum → sum(intent.n)
        }

    private fun sum(n: Int) {
        scope.launch { this: CoroutineScope
            val sum = withContext(Dispatchers.Default) { (1L..n.toLong()).sum() }
            dispatch(Result.Value(sum))
        }
    }
}
```

MVIKotlin Store Reducer


```
private sealed class Result {
    class Value(val value: Long) : Result()
}

private object ReducerImpl : Reducer<State, Result> {
    override fun State.reduce(result: Result): State =
        when (result) {
            is Result.Value → copy(value = result.value)
        }
}
```

MVIKotlin StoreFactory


```
internal class CalculatorStoreFactory(private val storeFactory: StoreFactory) {

    fun create(): CalculatorStore =
        object : CalculatorStore, Store<Intent, State, Nothing> by storeFactory.create(
            name = "CounterStore",
            initialState = State(),
            bootstrapper = BootstrapperImpl(),
            executorFactory = ::ExecutorImpl,
            reducer = ReducerImpl
        ) {}
```

MVIKotlin View


```
interface CalculatorView : MviView<Model, Event> {

    data class Model(
        val value: String
    )

    sealed class Event {
        object IncrementClicked: Event()
        object DecrementClicked: Event()
    }
}
```

MVIKotlin Binder

```
open class CalculatorController(
    private val onRender: ((Model) -> Unit)? = null
) : BaseMviView<Model, Event>(), CalculatorView {
    private val store = CalculatorStoreFactory(DefaultStoreFactory()).create()
    private var binder: Binder? = null

    @OptIn(ExperimentalCoroutinesApi::class)
    fun onCreate() {
        binder = bind { this: BindingsBuilder
            val view = this@CalculatorController
            store.states.map { Model(it.value.toString()) } bindTo view
            view.events.map { it: Event
                when (it) {
                    Event.IncrementClicked -> CalculatorStore.Intent.Increment ^map
                    Event.DecrementClicked -> CalculatorStore.Intent.Decrement ^map
                }
            } bindTo store
        }
    }
    fun onStart() = binder?.start() ?: Unit
    fun onStop() = binder?.stop() ?: Unit
    fun onDestroy() {
        binder = null
        store.dispose()
    }
    override fun render(model: Model) {
        super.render(model)
        onRender?.invoke(model)
    }
}
```

Platform View (Android)


```
class MVIFragment : Fragment() {
    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View =
        FrameLayout(requireContext()).apply { this: FrameLayout
            val textView = TextView(requireContext())
            val incrementButton = Button(requireContext())
            val decrementButton = Button(requireContext())
            val calculatorController = CalculatorController { it: CalculatorView.Model
                textView.text = it.value
            }
            incrementButton.setOnClickListener { // Click to send increment event
                calculatorController.dispatch(CalculatorView.Event.IncrementClicked)
            }
            decrementButton.setOnClickListener { // Click to send decrement event
                calculatorController.dispatch(CalculatorView.Event.DecrementClicked)
            }
        }
}
```

Platform View (iOS)


```
class CalculatorViewBinder: CalculatorController, ObservableObject {
    @Published var model: CalculatorViewModel?

    override func render(model: CalculatorViewModel) {
        self.model = model
    }
}

struct CalculatorView: View {

    @ObservedObject var binder = CalculatorViewBinder()

    var body: some View {
        VStack {
            Text(binder.model?.value ?? "")
            Button(action: { self.binder.dispatch(event: CalculatorViewEvent.IncrementClicked()) }) {
                Text("Increment")
            }
            Button(action: { self.binder.dispatch(event: CalculatorViewEvent.DecrementClicked()) }) {
                Text("Decrement")
            }
        }
    }
}
```

MVIKotlin 的缺点

- 用法比较复杂
- 与生命周期的集成不够完善
- 需要定义的 class 过多
- 样板代码较多
- Store 颗粒度太小

是否还有别的选择?

- workflow-kotlin (Square)
- moko-mvvm
- Writing by yourself

欢迎联系我

KUG Shanghai：掘金（Kotlin 上海用户组）

微信公众号：Kotlin 维修车间

Email：qiaoyuang2012@gmail.com

微信

额外内容

- 鉴于直播受到时长限制，所以有部分观众问题没有时间解答，我将在额外内容中对通过石墨文档、Slido，以及 B 站直播弹幕中的问题进行解答。

问答 1：

- 问：在一般安卓内有需要使用MVI吗 还是这是KMM的情境使用的?

- 答：MVI 是个架构模式，通常我们会在业务代码足够复杂的时候才需要使用架构模式。MVI 出现在 Android 平台比 KMM 要早，当时主要是为了在 MVVM 的基础上解除 View 对 ViewModel 的依赖，使解耦合更加彻底。在 Android 目前最流行的还是使用 Jetpack AAC 实现的 MVVM 模式，如果 MVVM 对你的业务代码来说足够使用，倒也无需切换到 MVI 模式，如果你想在纯 Android 环境使用 MVI，可以使用开源社区的 MVICore 这个库，或者您也可以将 Jetpack AAC 的使用方式稍加变换来实现 MVI 模式，毕竟模式只是理论，具体实现还是要看具体的框架或库。

问答 2：

问：對於 KMM 來說，使用 MVIKotlin 這種函示庫的最大動機是什麼呢？如果沒有使用 MVIKotlin，做甚麼事情會很麻煩呢？在選擇其他類似函示庫時，你覺得你希望這樣的函示庫最少需要支援那些功能呢？

答：主要是我们公司团队的现有业务代码比较复杂，经过数年的历史堆积已经阅读困难，去年 Android 代码在我们重构后已经全部改写为基于 Jetpack AAC 的 MVVM 模式。所以目前要将代码迁移至 KMM 后，KMM 代码也需要遵循某些高效的架构模式才可以。MVIKotlin 虽然并不好用，但也是目前 KMM 不够发达的生态环境下极少的选择之一。如果有别的类似的架构组件库，我希望它有如下几个特点：1.使用便捷，2.对 Coroutines 友好，3.便于集成生命周期（最好能实现 KMM 版的 Jetpack Lifecycle）。

问答 3：

- 问：Action 和 Intent 不是一样吗?

- 答：没错，从功能上来说一样，不仅 Action 和 Intent，Event 和 Intent，Model 和 State 其实都没什么本质区别，在编写实际代码的时候我们完全可以用同一个类型来表示 Action、Intent、Event 这三者，以及用同一个类型来表示 Model 与 State 这两者。这不仅能让我们编写的代码更简洁，还能省略掉许多重复的 class 定义，一举两得。