**Nexus协议，闲鱼一体化开发的幕后玩家**

## **作者：闲鱼技术-海潴**

Serverless是这几年兴起的一个概念，Serverless可以帮助开发者减轻甚至摆脱传统后端应用开发所需要的服务器设备的设置和运维工作，并以服务接口的方式为开发者提供所需要的功能。它希望开发者更加专注于应用逻辑本身，而不是被琐碎的基础设施细节所”绑架“。

而FaaS是Serverless的一种比较好的实践方式。自从亚马逊的AWS在14年推出Lambada之后，FaaS这种后端发开方式迅速被大家接受并应用。它拥有更加轻量、事件驱动的特点。

闲鱼选择使用Flutter + FaaS体系来实现云端一体化的开发模式也正是看中了Flutter和FaaS技术本身都是轻量的、面向应用的技术。与一体化本身希望开发者尽可能关注整体的业务逻辑非常契合。

Flutter + FaaS的云端一体化开发模式已经在闲鱼中被使用了一段时间。同事们之前也有过一些文章来介绍一体化开发在闲鱼演进和落地的过程。在这些文章中，都提到了Logic\_engine、Nexus\_Framework等字眼。它们一直默默得在业务开发同学的身后，支撑着一体化的落地和发展。

今天，我们就来介绍一下这个一体化的幕后推手--- Nexus协议，以及基于它衍生出来的框架和库。

### **Nexus协议的由来**

一开始说要做Flutter + FaaS一体化开发的时候，我们对”一体化“这三个字的认知相对比较模糊，只是知道端侧的同学可以用Dart这门语言来写FaaS函数，这样的语言上的一体化。对于FaaS所能做的事，也仅仅停留在前端实施已久的BFF层面。那个阶段，对于要做些什么，还是比较迷茫的。

阿里的同学经常说：

你不知道能做些什么，是因为想得还不够清楚

本着这样的想法，一体化小组经常聚在一起讨(liao)论(tian)，不管Flutter + FaaS有没有一体化，反正我们小组先”一体化“了再说。

整个一体化的概念在讨论中慢慢变得清晰，首现我们对于一体化进行了定义，它应该是这样的一个形态：

1. 语言一体化
2. 开发模式与架构一体化

最终达到开发Flutter页面和开发FaaS无明显gap，像在开发一整个应用的体验。

#### **语言一体化**

由于Flutter本身是以Dart作为开发语言，那么我们自然也选择它作为FaaS的开发语言。闲鱼在之前已经实践过了Dart Server这种开发方式，在Dart runtime、相关开发工具方面有非常深厚的沉淀。组内的同学将这个runtime经过修改之后移植到了集团的FaaS平台Gaia上。

开发同学不仅可以在端上使用hotreload进行页面快速调试，同样可以使用这项功能在FaaS平台上快速部署与调试，极大得提升了部署和调试的体验。

#### **开发模式与架构一体化**

在语言一体化的基础上，我们同样希望开发者在开发Flutter页面和FaaS函数的时候，有着相同的心智。

在传统的前后端分离开发模式中，端侧的开发与后端开发有着比较明显的不同，端侧通过和后端约定数据结构的方式获取用于页面渲染和处理用户输入的数据。这种模式下，双方仅对数据进行了依赖，各自属于不同的系统。

在一体化的模式下，我们希望开发者能把端侧页面和FaaS函数当成同一个系统来看待。它们应该是一个有机的整体，共同完成一个页面的功能。在职责上，端侧代码主要处理UI的渲染，FaaS函数主要处理逻辑与副作用。

开发者应该可以像在一个系统内一样进行相互的调用，就好像你在本地调用一个对象的函数那样自然。

但显然，端与FaaS现实中还是属于两个系统的，如何能够做到像调用函数一样自然呢？它们之间又以什么样方式进行触发呢？

#### **事件驱动**

在常见的客户端页面开发过程中，端侧逻辑总是围绕着三个操作在进行，不管代码多少，写成什么样，这些逻辑代码最终都会产生：

1. 发起一个网络请求(remote req)
2. 调用一个公共函数(native api)
3. 修改页面数据并渲染(state change)

这样的三个效果。

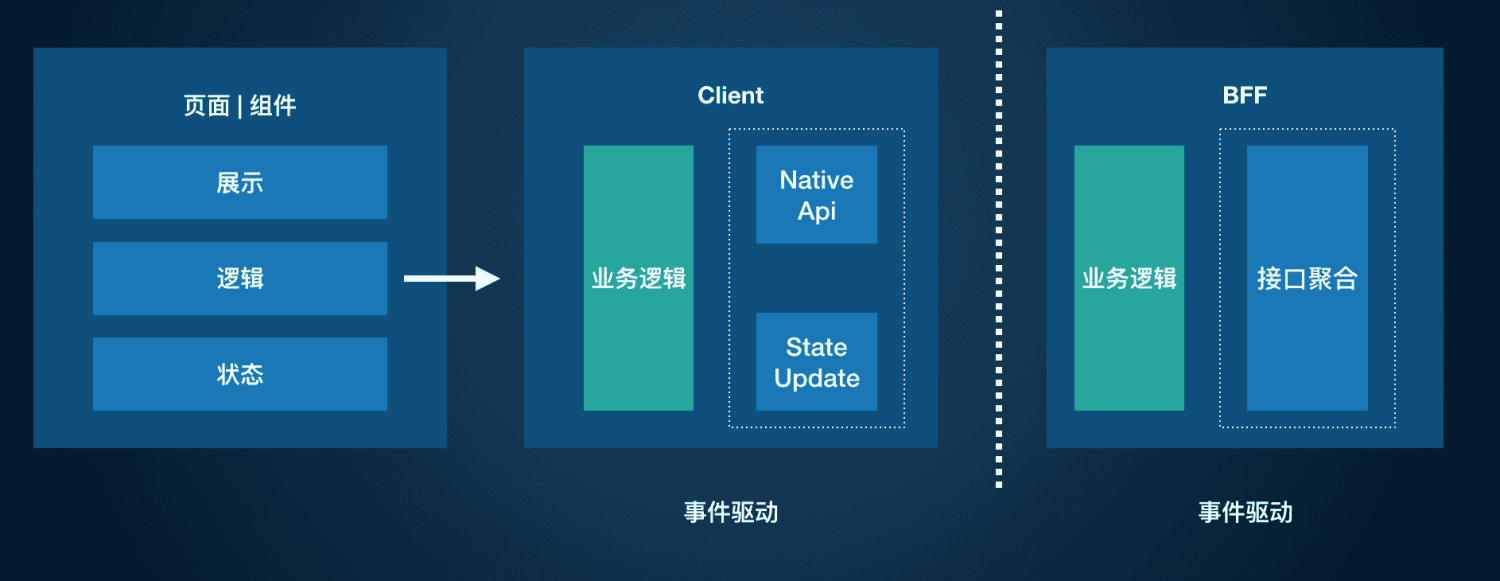
比如页面的初始化过程，就是典型的 remote req=>state change=>render的过程。

当然这是精简之后的流程，由于一个http请求回来后的数据并不能直接作用于页面state，通常还需要先对数据进行一下处理。

这些动作都会由一个明显的事件来触发，通常来说是用户进行的交互事件，不论是请求、页面渲染，或者弹出一个Dialog，进行一次页面间的跳转，它们都不会自发得进行(否则看上去有些诡异)。

而一个端上的事件，也可能会传导到FaaS上，来驱动FaaS上的逻辑函数对这个事件进行处理。当我们把端和FaaS看成一个整体的时候，这个事件就是在一个系统中流转。

于是我们总结出了第一张图：



#### **逻辑归一与互相调用**

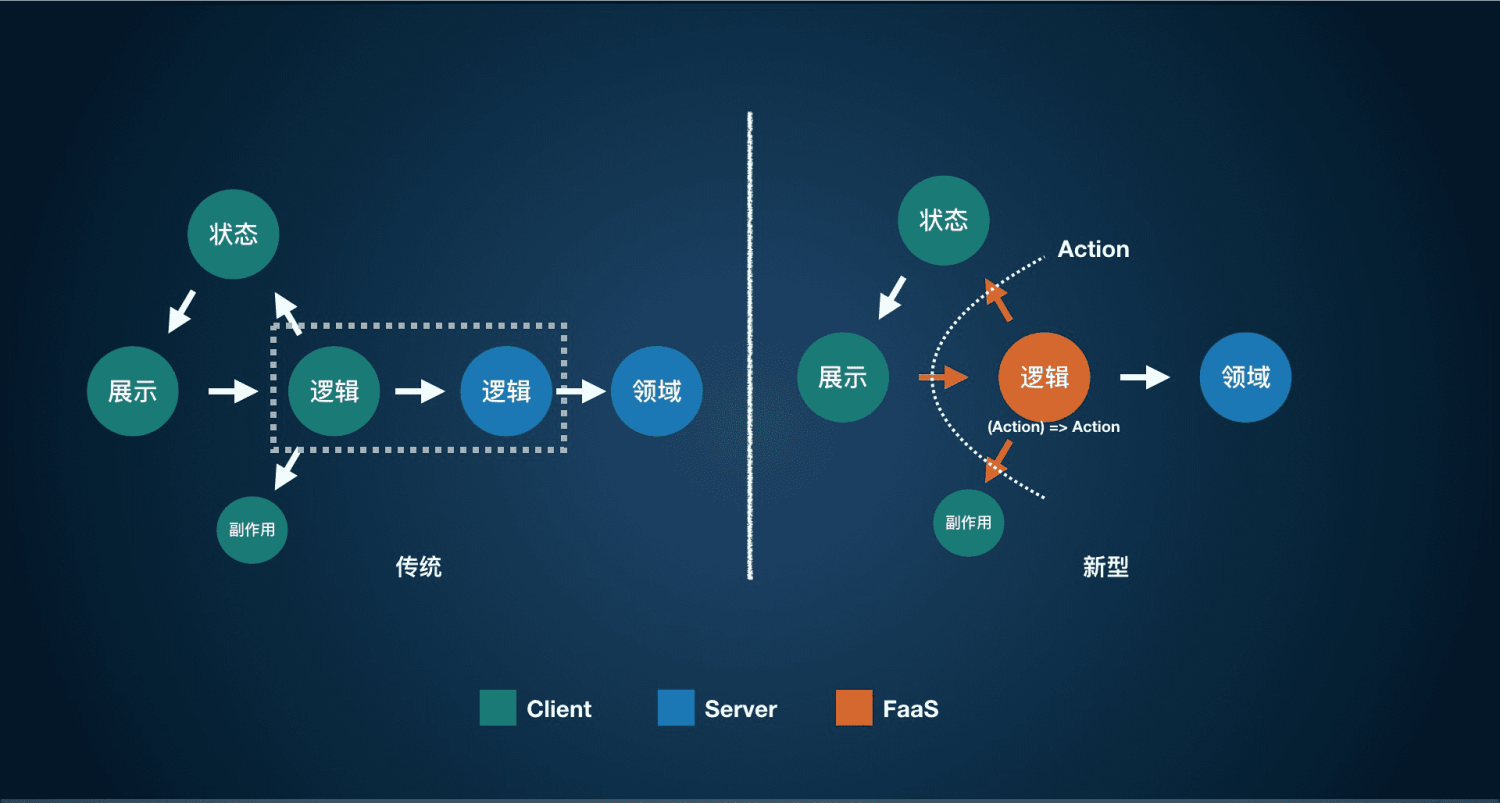
在传统的开发模型下，页面逻辑、状态、展示三者之间的流转是在端侧进行的，后端负责了一部分的逻辑处理(通常这部分逻辑是需要对于各种领域接口进行调用)。

而还有一部分领域数据到UI state的一些转换逻辑，则是端、后端都会做一部分。这两部分逻辑分散在两端，通过某种弱的协议进行连接。

引入了FaaS之后，自然可以把逻辑放到FaaS上实现，那么请求回来的数据理论上可以直接作用于页面渲染。

如果我们再进一步，不如直接让FaaS来指挥端上的UI怎么做好了。就好像FaaS是一个导演，而端侧UI是一个提线木偶，FaaS怎么说，UI怎么变。这样把业务相关的逻辑都搬上FaaS去，端侧专注于如何将state渲染到UI上，两个部分组合成为一个页面整体，岂不是更加一体化？

于是我们有了第二张图：



逻辑归一到FaaS之后，FaaS已经可以跳过传统的弱协议，直面端侧页面了。对于后端来说，一个请求可以映射到一个具体的处理函数。我们可以不太严谨得说，一直以来，客户端是有调用后端函数的能力的。那么既然我们现在想让FaaS来指挥端上的UI的变化，势必也要让FaaS具有调用端侧函数的能力。

我们把一次调用抽象为一个Action，每一个Action的背后都有一个特定的函数为它提供真实的逻辑，也就是说，一个特定的Action，可以用来描述一个特定的函数与函数背后的逻辑代码，Action本身就是一个函数签名。

那么端侧需要提供多少函数给FaaS呢？当逻辑归一到FaaS上之后，我们会发现端侧的大部分实现都围绕着两部分进行：

1. UI展现
2. 副作用处理

UI的展现是”纯“的，它基本上都可以由一个页面的state数据来描述，也就是说，大部分情况下，一个state就描述当前UI的状态。那么对于端侧来说，只需要提供一个state到UI的映射函数，理论上就可以让FaaS具有更新端侧UI的能力。也就是说，假设FaaS函数想要更新页面，只需要下发一个state change的Action，带上页面所需要的所有state数据，就可以达到效果。现实场景中，某些页面的state数据可能巨大无比，不好直接传输。我们做了一个JsonPatch库来解决这种场景下的问题，如果FaaS只修改了state中的一部分数据，则可以通过下发patch的方式由端来合成一个新的、完整的state。

对于副作用处理的部分，大部分的副作用都来自于比如Dialog，页面跳转等。这类操作是通用的，有共性的，我们同样使用一类叫做native api的Action来描述，这些Action与它们背后的处理函数，将面向所有使用了Nexus协议的页面提供这样的能力。

这两类函数的抽象，已经可以cover 80%的页面需求了，而剩下的20%复杂交互的页面，我们提供custom类型的Action来让开发者进行自定义。

#### **总结**

通过对于一体化的定义，以及拆分了需要的功能之后，Nexus协议就破土而出了。它是一个

基于Action的，提供Client/FaaS系统间调用的协议

### **Action的调度者-LogicEngine**

我们有了可以用于两个系统间进行相互调用的协议，相当于我们有了一门语言，这门语言只有我们自己认识，所以还需要一个解释器来执行它。

LogiceEngine就是这样的一个执行器。

如果我们给它下一个定义，LogicEngine就是一个：

基于Nexus的Action协议的，提供Client、FaaS之间相互调用能力的库

Engine本身不提供任何具体的逻辑能力，所有的逻辑能力都需要通过函数的形式注册到Engine中，并绑定到一个具体类型的Action上去。

所以Engine的设计相对明确：

1. 对外，它提供函数注册和基于消息(action)执行函数调用功能
2. 对内，进行消息解析、函数匹配和执行上下文管理

开发者通过post函数来发出一个Action，相当于通过Engine调用了一个函数，这个函数可能在本地，也可能在FaaS上。这并不是开发者需要关心的内容。甚至于，这个调用会产生什么影响，也不是当前调用者所需要关心的。

因为调用的发起者实际只是发出了自己的一个意图，比如在实践中，我们会在用户按下"下单"按钮的时候提交一个意图(Action)。

这个意图最终会产生什么样的UI变化，FaaS会通过一个state change或者native api形式的Action直接调用到具体的实现函数去。

而端侧注册在Engine的函数不会很多，前面有提到过，大部分UI编程中的逻辑，都可以被归纳为三类。所以我们大多数时候只需要注册三种固定类型的处理函数就可以了。

### **展望**

有了Nexus协议、三类通用处理函数的抽象和LogicEngine，意图=>作用中间过程就可以变得透明。

但这些远远不够，

后续我们还希望对协议进行升级，从现有的json提升到一个更加类型安全的协议上。

我们也希望有一个IDL工具，可以自动得将面向Action调用转换成面向接口调用，让开发者有更好的调用体验。

我们还希望改变现有单向的请求=>应答模型，让FaaS可以自由得调用端侧函数，再次突破两个系统之间的gap，变得更加一体化。