**漫谈框架设计**

在应用引擎生态里, 框架是承上启下的:

向上, 约束和规范代码的编写, 组件的划分, 提供开箱即用的功能, 是应用引擎赋能的透出点

向下, 对接应用引擎运行时, 告知系统应用实例的信息, 允许系统对其细粒度的操控配置

因此框架起到一个粘合作用.

在进一步讨论框架的技术细节之前, 先介绍一下设计思路.

* 如何开发
* 应用的运行态

**编写代码**

在应用引擎的生态里如何开发应用, API, 函数…?

我们既要解决现有的问题, 解决当下的槽点, 让用户开心; 还要遵循良好的设计, 保证我们的框架是可持续发展的.

从头设计一个框架往往更能让人获得一个完整体系化的思路, 并且站在正确的设计里去求解我相信会更高效坚实. 从这里入手, 那么如何定义良好正确的设计?

**业务域**

我们面向的业务领域是前端域, 存在的是绝大多数是轻量级的, 模型简单, 二方依赖少的基于 HTTP 协议同步通信的无状态服务端程序. 少数稍微复杂一些, 可能会有一些二方依赖, 如 nsq, kafka, redis, mysql, 通信协议也可能是其它的非 HTTP 的应用层协议, 通信模型可能是异步模型, 甚至是自身有状态的. 后面这些可能更应该划分到后端域, 因为领域模型本身在他们的代码里, 不存在更后端的业务域.

那么针对这种形态的服务, 我们一定希望少一些基础重复代码的编写和拷贝, 对于大多数业务开发, 可能更希望减少底层概念的透出, 减少对业务逻辑的侵入糅合, 减少复杂的架构设计时间. 高效高质量的完成核心业务的开发.

**API**

面向 API 开发是众多开发/架构师在理论和实操后认可的方向, 也是未来长期的一个趋势.

为什么? 以 AWS Serverless, 阿里云函数计算为例.

当我们把应用设计为一组 API 的集合, 并以 API 的维度去独立开发时, 只需要配置触发条件, 编写执行逻辑即可. 这很符合前面的观点: 专注于业务本身.

那为什么 serverless 没有大面积落地像 micro service 一样爆火呢?

我猜测其中一个原因是光凭网络上一套 serverless 的科普和简单的示例是无法完美契合各个企业的场景的. 有了 aws 一样的 serverless, 但大多数的思维还是像以前开发应用一样, serverless 也并没有强制约束如何开发, 所以你依然可以一个函数塞一大坨代码, 各种 require 外部包. 最后写着写着完全变味儿了, 又回到传统应用开发的思路. 这也说明缺少约定, 缺少 best practice, 缺少订制化的赋能支持. 这是要有的.

回到前面, 我们的应用有 HTTP, 也可能有非 HTTP, 可能是同步也可能是异步, 并且随着我们平台的发展, 后面支持函数计算是必然的. 那么我们设计一套能够屏蔽底层差异性的抽象层是不是更好一些, 类比移动端开发界的 react native, 是不是也这思路?

想到这里, 和魔方有了交集. 魔方是专注于 API 开发的. API 的静态形式是:

核心业务代码(函数) + API 的元数据.

我觉得骨子里的良好基因已经有了. 思路稍作调整, 并且允许应用引擎为其赋能, 推广可能会更有底气.

**应用开发**

所以为什么 API/函数的研发模式真正落地度不高, 还有一点原因是超前了, 现在许多场景应用可以很好的 handle, 要改成面向 API/函数模式, 可能要转换下思维, 落地后怎么样后续运维怎么样也是未知的, 并且有些场景可能应用也很适合.

尽管 API/函数开发是诱人的, 但一点点靠近才是理智的. 因此单体应用无论是作为过渡还是特殊场景支持, 其存在的必要性还是有的. 为了更容易更平滑的靠近函数计算模式, 我们可以考虑让框架的核心能同时服务于单体应用和函数/API.

事实是, 应用无非是一组 API, 划分个边界, 我们叫它应用.

在魔方的设计里, 整个工作流都是在围绕单一 API 做开发. 或者说把 API 分的很开.

放在应用开发的场景里, 区分这么细反倒影响使用体验, 使开发效率大打折扣. 因此, 可以考虑提供一个简单并集中的 meta, 描述应用. api 的 meta 甚至可以没有 (当然也可以为了向后兼容而继续生成, 不过最好要自动生成而不是要开发者手写), 只需要编写 api 的 handle, 然后在集中的 meta 里配置他们的路由和中间件. 这样更能突出应用开发的概念, 也和传统web框架更接近.

上面提到了 api meta 不要手写, 其实现在它也不是手写的, 但整个开发是放在 web 端的, 在各层支持都很不完善的情况下, 直接走 web 端开发, 体验能好到哪去? 说好那是骗人的. 所以说产品这里步子迈的相当大了.

所以我说的是这套开发框架主推本地构建, 主推本地开发, 主推本地调试, 能够一键打包上传, 配合本地研发工具约束开发者的操作规范. 我觉得这更接地气, 也更现实, 渐进一点.

面向开发者的框架设计细节, 以及本地研发工具库设计, 暂时没有定论, 后面方案敲定后的详细设计交给 @古几 @大富 同学讲解, 这里就不展开讨论.

**梳理研发侧最佳实践**

最后, 我们再来聊聊如何满足用户的需求. 我们这几个月零零散散收集到的各路对web框架的吐槽和期待提炼如下:

* 滥用的 CSM 模式
* 臃肿的框架
* 冗余的依赖
* 开箱即用的二方能力 (日志, 中间件)
* 易排查问题
* 能实时查看应用状态
* 统一约束和规范

开篇已经做了大胆的假设, 在良好的设计里, 很多问题都能迎刃而解. 那么上述这些问题和需求能否在新的框架里有个归宿?

首先我们要解决的就是 controller 过重的问题, 在旧的体制里, 路由的 handle 是一个 controller 对象方法的序列. 相当于一组koa中间件依次安插到了某一个 controller 类的对应原型方法上. 因为这个模式的存在, 会让你很轻易的把代码写成富 controller 序列. 阅读一些代码样例, 其实我们可以发现, 这些 controller 方法序列, 很多可以抽离为各个独立的 mw. 放在新的框架里, 这些拆分出来的 mw 可以作为某一个/某一组路由的前置中间件, 而对于框架自动扩展了不需要的依赖这种情况, 也可以通过这种方式解决:

|  |
| --- |
| YAML  baas:  - "redis": cluster: "micro"  - "oss": namespace: "test"    route:  "/\*":  - "@youzan/logger"  - "./lib/mw"  - "::redis"  - "::oss"    "/a/\*":  - "@youzan/xyz"  - "@yozan/abc"    "/a/b/c":  - "./lib/a"  - "./lib/x"  - "./PUT:/a/b/c" |

业务核心代码在每个 API path 对应的最后一个 mw 里.

对于那些冗余的依赖, 一个好的现象是有团队在做拆分, 如 iron-base, 不过最好能聊一聊, 避免后面不兼容框架.

开箱即用的能力, 即无需大量配置或者配置技术细节即可使用, 甚至无需显示接触即可自动获得的能力.

比如日志输出后自动规范化/透明轮转/透明上报.

需要时直接从某个已知对象上获取实例化过的 mysql, 无需关心中断重连, 无需关心连接池, 服务端信息等.

所有的调用都自动接入链路追踪. 透明支持 sc 等等.

对于这些能力的差异化, 我觉得也是前面 serverless 没能真正普及的一大原因, 这一块的做法厂商各异, 没有什么最佳实践出来, 而底层环境什么样也不尽相同, 难以说何种做法是最好的. 但在我们的设计里, 可以依据公司现状和底层架构来做针对性设计.

在开发侧, 用户在路由上配置希望使用的开箱即用能力, 如例子里的 "::redis", "::oss", 这里我们用 "::" 两个冒号开头表示对资源依赖的声明, 一旦声明了依赖, 那么将伴随应用的整个生命周期.

在代码里, 可以通过如下方式获取:

|  |
| --- |
| TypeScript  getUser(ctx: Ctx, baas: BaaS) {  return ctx.params.userId  |> await baas.redis.get  |> ctx.success  } |

**应用运行态**

说完了静态的研发侧, 再来聊聊运行态, 即实例.

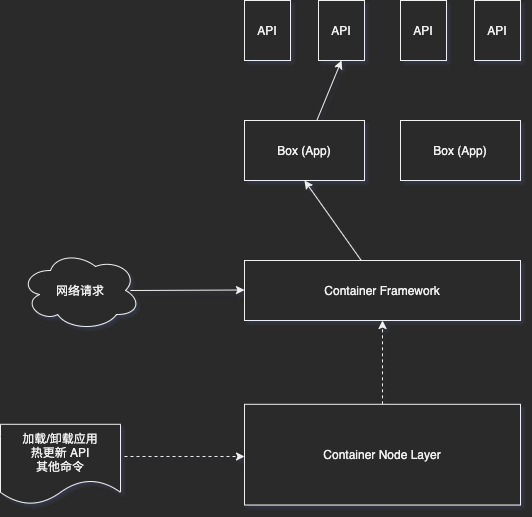
当开发把写好的代码上传打包构建之后, 便可以通过应用引擎部署.

与当下开发者自己申请服务器资源自己部署登录服务器 debug 等等不同, 在应用引擎里, 你是无法声明指定应用部署在哪里的 (但你可以知道, 你也可以配置 HA 副本数). 应用引擎全权为你分配应用实例的住所.

那么如何把我们前面讨论的框架设计, 完美契合到应用引擎运行时, 以实现应用引擎对应用实例的监管和控制呢?

先来讲下应用引擎层面对容器节点的设计.

如图所示, 忽略了细节, 容器节点的技术栈就是:



最底层的通用节点层, 负责标准的 IPC 机制, 命令 RPC 机制, 并负责初始化/触发上层模块

**容器框架**

第二层是容器框架层, 是容器节点有别于其他节点的一层, 提供这种类型节点的核心逻辑, 在这里就是应用实例的管理

更具体地, 容器框架由这几部分组成:



容器框架最核心的是 Server, 因此它负责承接所有打到这个节点的业务流量, 并将流量按照规则转发给 Box Manager 中维护的某一个 Box.

Server 可以有很多种, 最常见的是 HTTP, 也可以是 Message/Event Server, 或者其他扩展的 Server. 具体是什么要根据这个节点的初始化配置而定. 可以通过专属命令动态管理 Server

Box Manager 是另一个核心组件, 负责维护当前节点托管的应用实例. 可以通过专属命令动态管理 Box

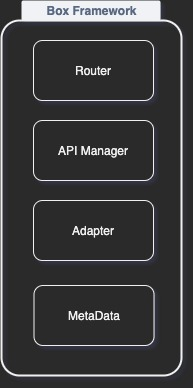
最后 OutofBoxExtension Manager 是开箱即用扩展管理器, 负责初始化baas对象并透出到应用上下文中 (比如 redis), baas 对象的生命周期管理, ACL 等也是位于这里. 可以通过专属命令动态管理可用的 baas 扩展.

**Box**

Box 是一个逻辑上的区域的概念, 将几个逻辑分组 (API) 圈在一起, 表达为一个应用.

Box 包含应用的元信息, 比如名字, 版本, 来源, 静态配置等等

Box 也是一个物理概念, 表示一个隔离的空间 (vm/sandbox). 因此不同 Box 之间默认不会共享内存. (实际取决于隔离级别)



位于容器框架之上的 Box, 其中最核心的逻辑便是路由器. 应用可能有很多路由, 对应一组 API, 而 Router 就是要维护这份路由表.

不同 Server 的流量以及选路机制可能不尽相同, 因此 Adaper 的存在就是为了提供适配不同 Server 的选路机制.

经过选路, 从 API Manager 里挑选命中的 API 开始执行业务逻辑

Box 按照 Adapter 的不同, 有多重类型:

* HTTP Box
* RPC Box
* Event/Message Box
* …

**API**

位于终端的就是 API, 上下文最狭窄的一层.

API 包含实例化的代码, API 的元数据:



根据应用配置情况, 也会包含一些前置的插件, 在请求真正进入 Handle 之前执行他们.

**完整的流程**

好了, 我们回过头来梳理一下应用从诞生到接客的整个流程.

**初始化**

1. 节点接收到加载命令, 调用容器框架
2. 容器框架通知 BoxManager
3. BoxManager 从文件系统给定路径加载应用代码
4. 加载完成, 根据应用配置和内置配置创建新的 Box 对象, 传递依赖资源引用, 完成初始化
5. BoxManager 通知容器框架加载完毕

**节点内的请求: 路由-处理-响应**

1. 应用引擎把流量调配到某个容器集群下的某个容器节点
2. 容器框架服务器接收流量, 根据流量标识 (如 http header: x-box-id) 从 BoxManager 选取命中的 Box
3. 如果有命中, 则 Box 调用 Adapter 解析流量, 并调用 Router 执行选路
4. 如果选路命中, 则调用 API 的 prelude plugins, 在最后调用 handle fn
5. 取得 handle fn 的返回值, 作为响应数据回传

**更多考量**

到此为止, 框架部分的大体设计思路就讲完了. 但仍然有一些点需要进一步考虑, 这里列举部分:

* 前置中间件的结构
* 开发侧框架的实际结构
* 本地调试设计
* Adapter 和 Router 设计
* 非 json api 类型 (如 web page) 支持
* baas 资源管理
* api handle fn 的限制域设定 (如文件系统的访问, 创建进程, 加载模块等)

以 ae 的视角, 框架定位是: 应用引擎与业务代码互通的 **协议**

以 app 的视角, 框架定位是: 既是承载业务逻辑, 允许无痛使用底层能力的 **容器**, 也是放在一个体系内, 以某种思维模型实现最佳实践的 **协议**

why & pros:

* 持续业务赋能的基础: 优雅启停, 开箱即用, 动态加载, 屏蔽底层
* 研发模式升级的基础: 是针对前端域业务模型的本质而定向优化订制的框架. 兼容函数研发模式, 与应用研发模式接近无成本切换, 符合可持续发展目标.
* 规范化: 框架本身的意义也包含对使用者的统一约束, 建议他在这个体系里应该怎么做, 怎么做是最佳实践. 借此机会规范一下诸如日志使用等长期存在的问题.

cons:

* 框架切换成本: 业务同学对新老框架的理解差异, 熟悉度不同, 会导致迁移质量变差, 耗时增加; qa 同学需要重新回归用例.