**应用引擎平台化底层架构整体设计**

**1. 前置阅读**

* 背景与架构简介: http://gitlab.qima-inc.com/app-engine/docs/issues/56

**2. 术语及缩写约定**

**2.1. 概念**

* 应用引擎域/ae-scope: 一个包含应用引擎服务/产品/工具的完整生态集合, 虚拟概念
* 宿主/host: 应用引擎体系内, 任何操作系统被抽象为一个宿主, 宿主是实际存在的
* 节点/node: 宿主上执行具体工作的模块, 一般是一个进程. 节点分为使能节点和辅助节点
* 使能节点/worker: 执行有代表性的核心任务的节点. 包括网关节点和容器节点
* 辅助节点/agent: 协同使能节点完成基本或专门处理扩展工作的节点. 如日志节点, 下载节点, 监控节点
* 应用/app: 一个虚拟概念, 界定一组同类业务逻辑的集合, 性质等价于 ops 上的应用. 是一个偏静态, 元数据的概念.
* 服务/service: 服务是应用的某个特定版本在运行时的体现, 比如应用 my-test-app@1.0.1 在应用引擎上被托管之后, 便成为一个服务. 服务包含运行时的信息, 以及所有副本组成的实例组.
* 实例/instance: 实例存在于容器节点里, 一个实例表达一个应用/函数的运行时
* 副本/replica: 一个应用在运行时会存在多个实例互备, 这些实例用副本表示
* 实例虚拟机/nano-vm/vm: 实例的底层运行环境, 根据应用语言以及隔离级别的不同而改变, 包括进程虚拟机, 线程虚拟机, 上下文虚拟机, 隔离虚拟机等等
* 隔离级别/iso-level: 隔离级别表示应用间可共享信息的范畴和能力. 一般与所使用的的虚拟机有关.
* 集群/cluster: 节点的逻辑集合, 集群的组建划分是调度中心决定的, 因此是一个虚拟概念
* 区域/region: 宿主的逻辑集合, 区域的组建划分是调度中心决定的, 因此是一个虚拟概念
* 单元/unit: 机房的逻辑集合, 不同单元之间往往在更底层 (公司运维层面) 存在物理或逻辑上的隔离, 为了识别这种差异, 应用引擎引入这一概念. 如主站, 有赞云, 金融云
* 环境/env: 每个 host 的底层标识, 属于公司运维定义使用的机房管理概念
* 机房/idc: 每个 host 的底层标识, 属于物理上的标记.
* 调度中心/控制中心/control-center/dispatch-center: 也叫总控. 管理应用引擎的元数据, 宿主的会话, 并完成关键调度/控制指令的计算编排下发. 同时提供开放调用的 API 给产品层服务使用
* 宿主控制器/housekeeper: 也叫分控. 在宿主上管理一个宿主下的节点, 同时负责和总控保持会话, 上报状态数据, 转发控制信令, 守护节点进程.
* 网关/gateway: 应用引擎数据平面的流量入口, 所有从外部打到应用实例的业务请求都会走网关转发. 网关一般是集群态, 核心由一组网关节点构成
* 容器/container: 托管应用实例所在的角色, 容器一般是集群态, 核心由一组容器节点构成
* 网络分区守卫/network-partition-guard: 从应用层检查网关到容器之间的网络是否可达
* 工作模式/workmode: 总控服务支持多种工作模式, 用于控制系统的某些自动化功能通断. 如开启维护模式后, 总控将拒绝所有宿主的新的注册请求
* 健康度/health-score: 应用可用的实例数/总共分配的实例数, 用于应用服务的可用性评估
* 数据面/数据平面: 从服务网格架构里借鉴来的概念, 业务数据的处理组件构成的子系统集合
* 控制面/控制平面: 同样来自于 service mesh. 发送和处理控制信号的组件构成的子系统集合

**2.2. 术语**

* 可调度: 当一个应用满足被部署/扩容的条件, 则表示它可调度. 可调度并不表示一定会部署/扩容成功.
* 路由自愈: 网关到容器间的网络异动以及容器自身的状态可能会导致路由信息的不准确, ae 自动的检查可达性并将结果反映到路由表上以保证路由的正确性这一过程, 叫做路由自愈
* 网络隔离/不可达: 传输层两个端点之间数据往返超时或者下层不通
* 高可用: 服务满足高可用时, 少量的实例失效不会影响服务的可用性, 或者可用性依然是满足要求的
* 分区/分片: 将一组信息划分为几个子组, 如总控对宿主会话的分片维护, 应用实例在容器集群上的分片托管等
* 负载均衡: 网关层面, 将请求流量依据给定策略均匀的分发到下一服务集群. 容器层面, 应用实例均衡的分布在集群之间, 使各个节点/宿主的负载是趋近一致的. 总控层面, 宿主会话均匀分布在总控成员之间, 各个成员维护的会话数量趋近一致
* 重新平衡: 当出现负载倾斜时, 通过自动或人工手段, 调整分配, 使之重新满足负载均衡, 重新平衡往往需要花费较高的成本, 甚至要以降低服务的可用性为代价
* 优雅启停: 服务就绪后, 流量才导进来; 所有流量处理完毕后, 服务关闭.
* 心跳检查: 间歇性给服务发送一个应用层短报文信号, 一来探测链路是否畅通, 二来刷新对端的超时时间 (如果有)
* 选路: 根据特定策略, 计算流量的目标位置的路径
* 放置: 托管到应用引擎平台上, 启动成一个服务

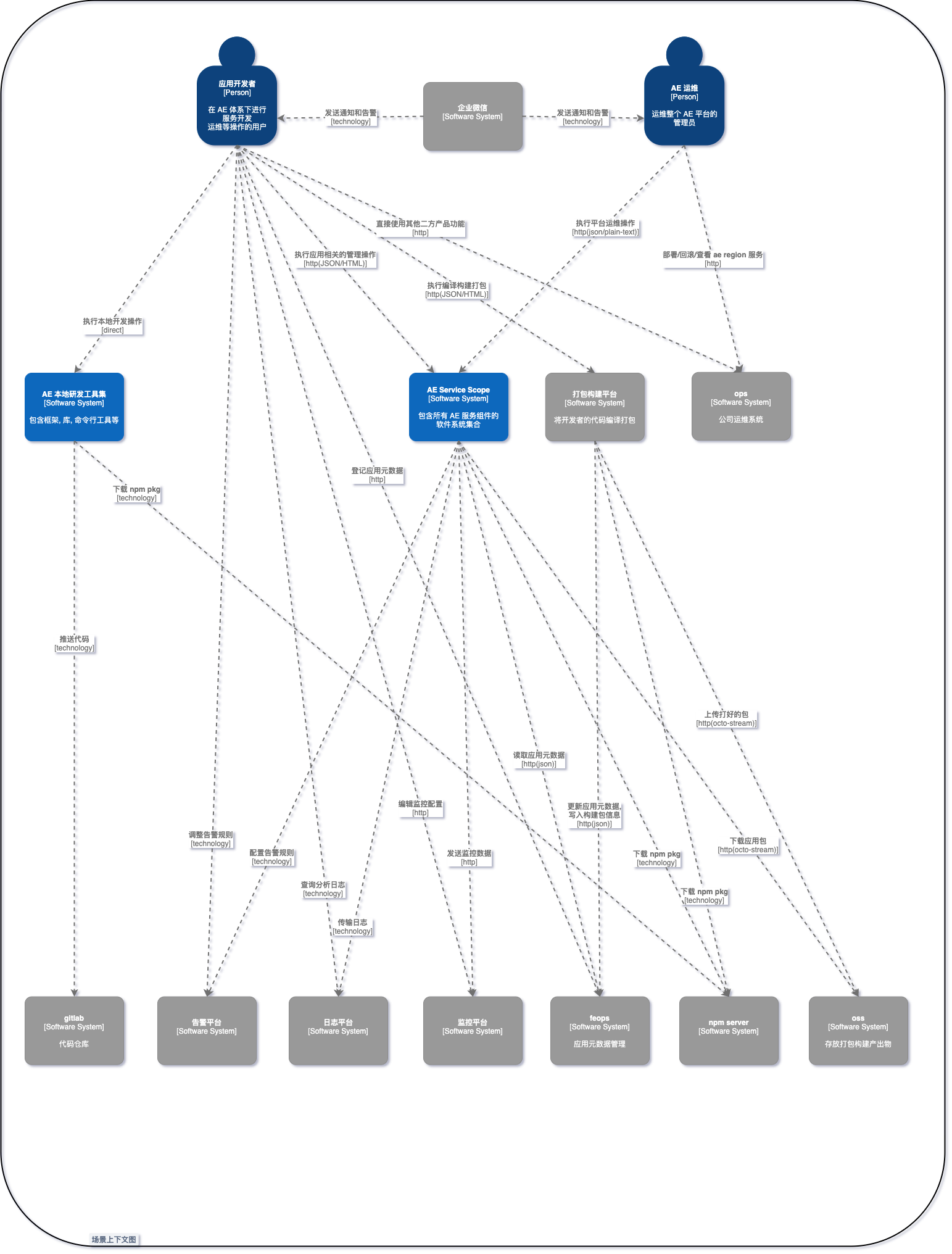
**2.3. 缩写**

* ae/AE: app engine (应用引擎)
* ae-gw: 应用引擎网关
* ae-ct: 应用引擎容器
* ae-cc/ae-dc: 应用引擎调度中心
* ae-hk: 应用引擎宿主控制器
* 总控: 调度中心
* 分控: 宿主控制器

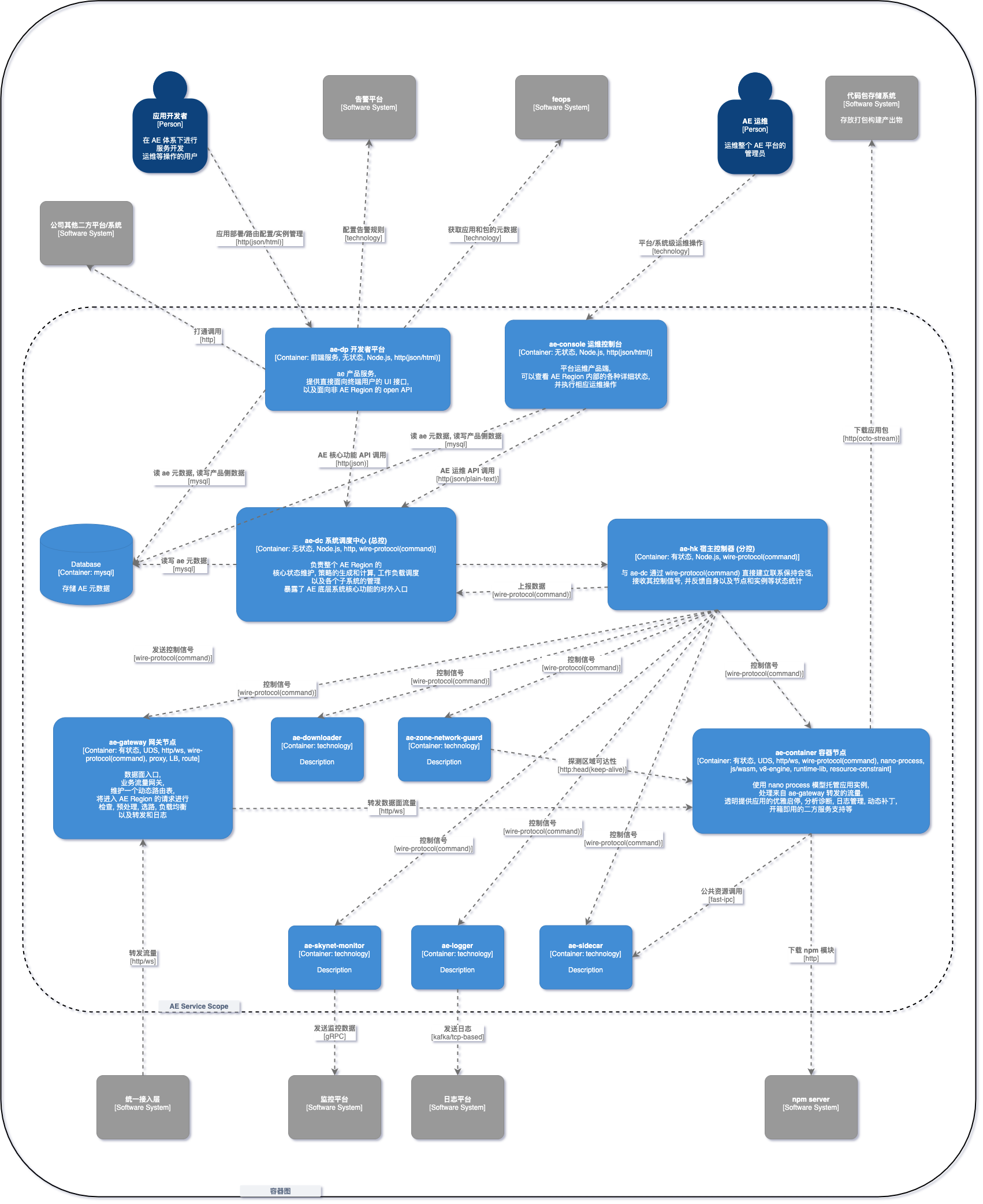
**3. 系统架构, 子系统分工与协作**

**3.1. 架构**

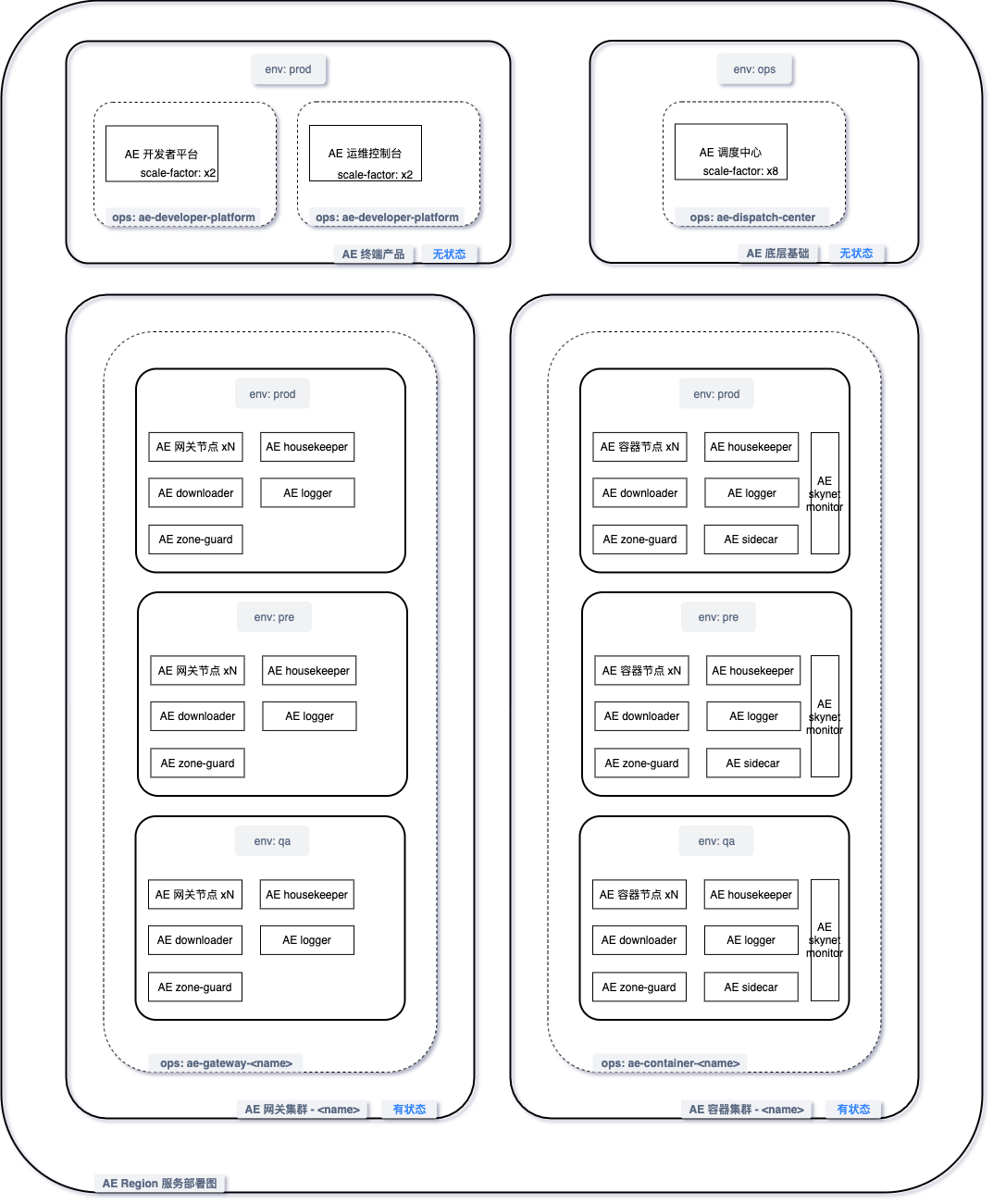
**3.1.1. C4 软件系统全景及系统上下文图**



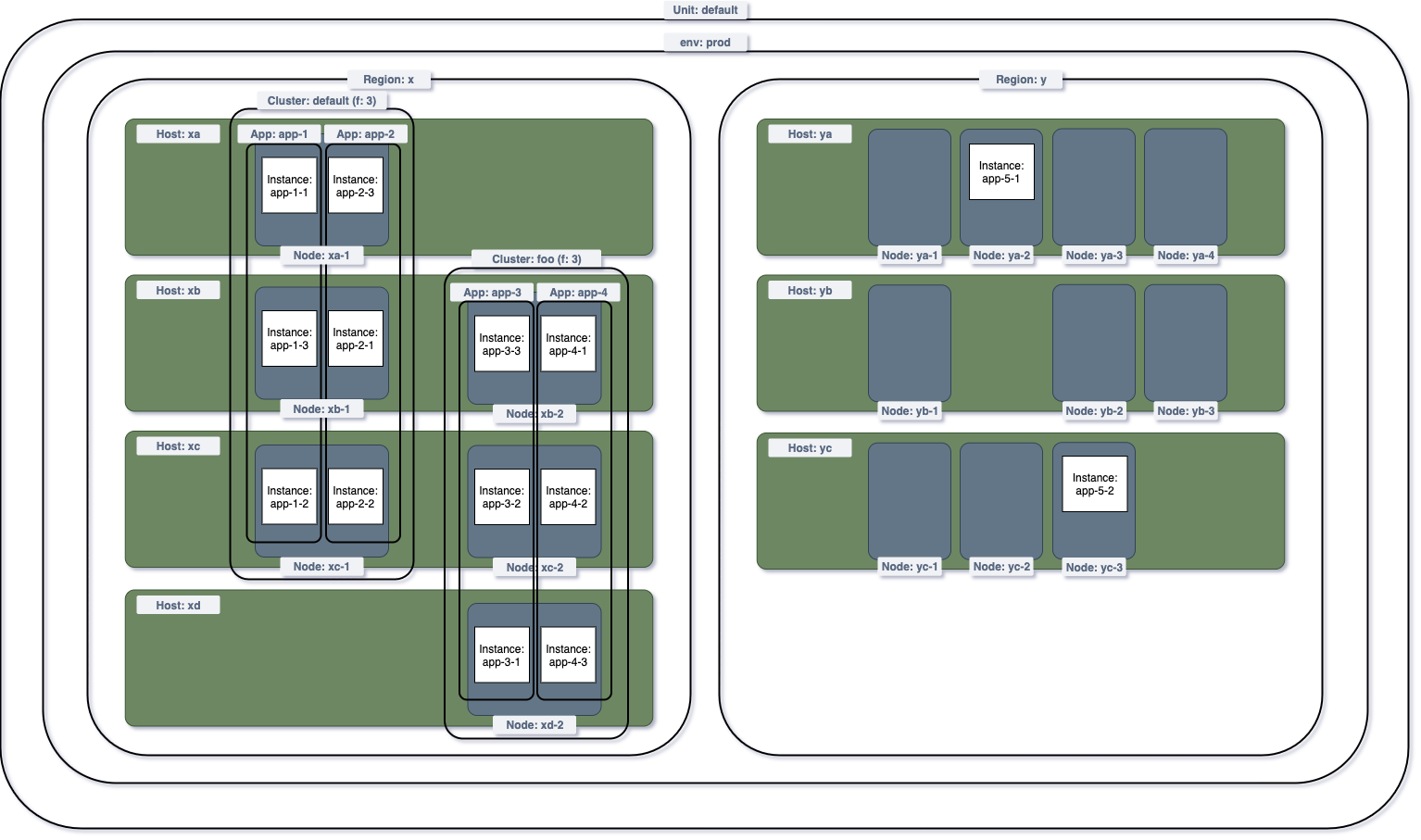
**3.1.2. C4 容器图**



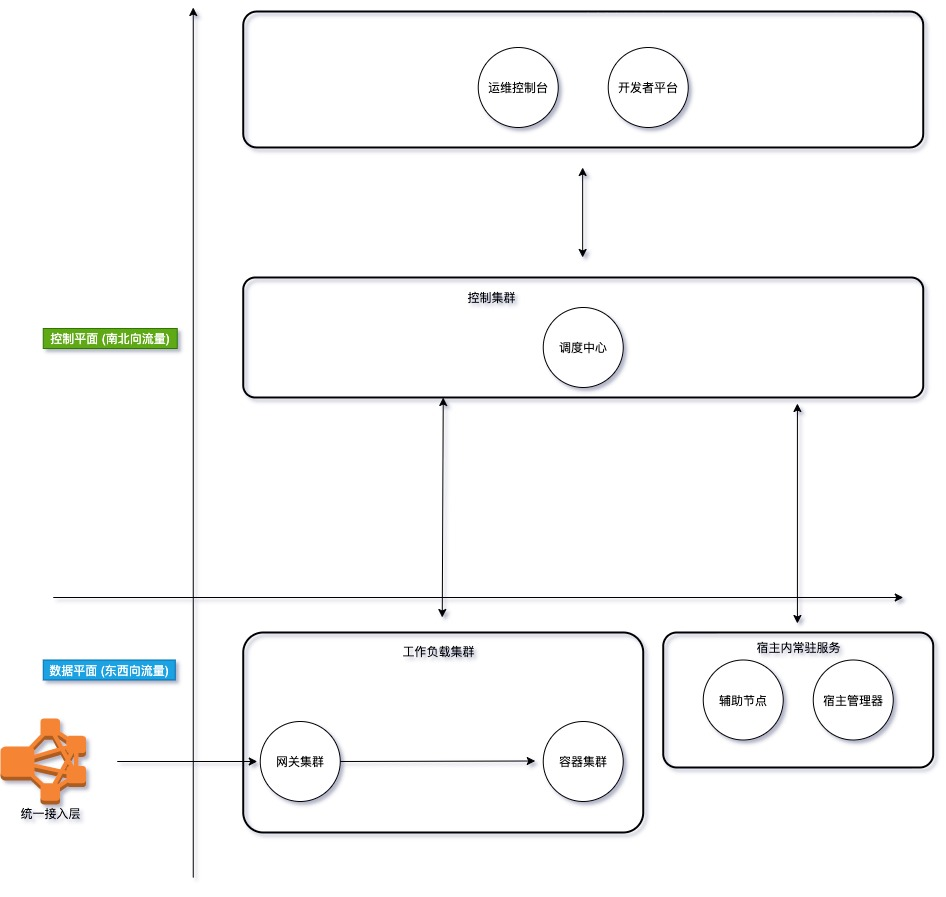
**3.1.3. 平台部署图**



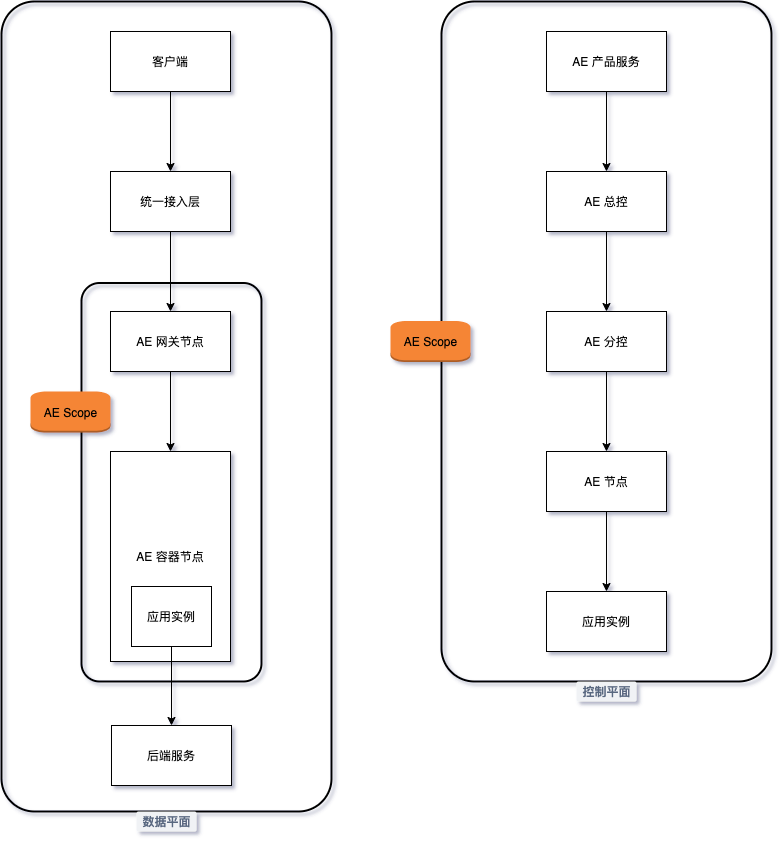
**3.1.4. 应用实例部署图**



**4. 控制面与数据面划分与说明**



**4.1. 数据平面与控制平面数据流图**



**5. 关键子系统设计说明**

**5.1. housekeeper 与节点进程**

宿主控制器 (housekeeper) 是一个独立服务, 除了直接控制宿主下的各个节点之外, 还负责与调度中心通信.

相比调度中心, 它负责的控制任务更具体更底层, 同一宿主下每个节点都是它的子进程, 节点的存亡, 启动配置, 运行参数, 统计信息等等都是由它来掌管. 详细说明参见应用引擎非托管模式的设计:

* 节点状态机: http://gitlab.qima-inc.com/app-engine/docs/issues/16
* housekeeper, 节点的说明: http://gitlab.qima-inc.com/app-engine/docs/issues/7#host
* housekeeper 的定位和设计考虑: http://gitlab.qima-inc.com/app-engine/docs/issues/15

我们会在详细设计里展开叙述这一部分的变更和改动细节.

**5.1.1. 节点**

每个节点也是个独立的服务进程. 节点的分类在开篇术语中已经说明, 应用引擎中的节点有两类: worker, agent.

属于 worker 的节点有:

* gateway node: 执行网关的核心逻辑
* container node: 管理虚拟机, 网络服务器, 并托管应用实例

属于 agent 的节点有:

* logger node: 负责 ae 系统日志, 应用实例日志的采集, 处理, 上报
* downloader node: 负责外部资源的下载
* network-partition-guard node: 负责网络可达性检测, 并做反馈
* skynet-monitor node: 配合公司监控平台实现透明赋能
* sidecar node: 二方服务资源管理
* 其他

**5.2. RPC/IPC**

控制面的一切信号走命令通道传送. 总控和分控间的网络通信, 分控和节点间的进程间通信, 对于对等端来讲都是无感知的, 因为他们都是基于同一的轻量级 RPC 框架实现的互访.

这种轻量的 RPC 基于 wire-protocol 封装, 应用层使用 "命令" 体系实现调用.

具体参见应用引擎非托管模式下早起的设计蓝本:

* <http://gitlab.qima-inc.com/app-engine/docs/issues/10>
* <http://gitlab.qima-inc.com/app-engine/docs/issues/9#%E9%80%9A%E4%BF%A1%E8%AE%BE%E8%AE%A1>

以及具体的实现 (对设计方案稍有改动):

* 序列化/反序列化: http://gitlab.qima-inc.com/app-engine/wire-pack
* <http://gitlab.qima-inc.com/app-engine/wire-proto>

该 RPC 框架并没有包含如 dubbo 一样丰富的功能, 但允许开发者按需定制这些能力.

比如应用引擎的跨网络命令体系基于 wire-protocol 扩展实现了心跳保活能力.

被命令化之后的设计如下:

<http://gitlab.qima-inc.com/app-engine/engine-service/blob/master/common/command/README.md>

**5.3. 网关**

具备网关标记的宿主, 会启动网关节点, 并成为 ae 的网关型宿主.

网关的元数据会持久化到本地存储, 便于重启时快速恢复.

一个网关系统里会启动如下节点:

* gateway
* logger
* downloader
* network-partition-guard

系统启动初始化, 识别自己是网关型之后, 将创建上述节点, 并回写元数据到本地存储, 便于快速恢复, 系统会根据 CPU 数量来创建网关节点, 其余 agent 节点均是单个.

downloader 将总控的路由信息转换数据结构之后写到本地缓存, gateway 直接加载这份数据为路由表结构.

数据面, 网关的核心结构是一个 pipeline:

|  |
| --- |
| Haskell  接收请求 -> 解析请求头 -> 选路 -> 请求预处理 (插件挂载点) -> 负载均衡 -> 转发请求 -> 请求后处理 (插件挂载点) -> 回复响应 |

经过网关选路和负载均衡后, 会选出一个应用实例的编号, 请求头会额外携带这个的信息以及其他上下文信息, 用于标识目标应用实例以及协助诊断分析

**5.4. 容器**

具备容器标记的宿主, 会启动容器节点, 并称为 ae 的容器型宿主.

容器的元数据同样会被持久化到本地, 便于快速启动.

容器系统启动如下节点:

* container
* logger
* downloader
* network-partition-guard
* skynet-monitor
* sidecar

系统会根据预置配置和分配策略来预创建合适数量的容器节点, 其他 agent 数量均为 1.

并且, 系统会根据当前状况和预估的未来情况来动态创建新的容器节点.

因为容器节点会启动不同端口的服务, 所以也不太适合 ops 粗粒度的统一 8201 端口检查, 并且对应用启动成功与否的检查最好能放开超时时间. 这里建议调大/关闭 docker 容器的启动检查, 由 ae 自行管理.

容器节点的数据面核心组件如下:

|  |
| --- |
| Rust  网络服务器 (http/ws) -> (潜在的 vm) -> box -> plugins |

对于上述所有的组件, 在控制面都可以进行操控观察.

创建实例前, downloader 会把实例的 pkg 下载到本地, 转换成 container 可以处理的格式. container 收到创建实例指令后, 从本地路径加载 pkg, 生成一个新的 box 对象和一个 vm.

接收到数据面请求后, 系统会提取请求头上用于标定应用实例编号的信息, 拿到对应的 box, 执行应用内部的路由和具体业务代码执行.

**5.4.1. baas 资源共享**

公司的二方能力以及其他建的能力会在应用引擎里初始化/实例化. 同一个 container 内所有 box 内共享这些服务, 避免应用里多次或者错误的初始化资源.

举例来说, 对于打开同一个 db 的 mysql, 同一个集群的 kvds, 已经建立的连接会在多个 box 间共享.

container 创建的资源会传递给 vm 以及 box 的相关组件, box 内部会将其封装为合适的形式交给业务代码使用.

综上, 宿主是系统的调度控制单元, 需要容器平台暴露 API, 以便对宿主实现独立的启停控制操作.

**5.5. 调度中心**

应用引擎的主脑, 记录各种组件的元数据和状态, 以及大量的拓扑关系.

但从设计上讲, 自身故障不会直接影响数据面的服务.

调度中心的架构设计参见此前的设计蓝本, 期间发生过多次重构和优化, 去除了 nsq 和 kvds 依赖, 实现了分布式路由表, 但模型并未发生实质性改变. 早期的方案以及中间的调整如下:

* <http://gitlab.qima-inc.com/app-engine/docs/issues/9#%E6%80%BB%E6%8E%A7-%E6%8E%A7%E5%88%B6%E4%B8%AD%E5%BF%83%E8%AE%BE%E8%AE%A1>
* <http://gitlab.qima-inc.com/app-engine/docs/issues/31>
* <http://gitlab.qima-inc.com/app-engine/engine-service/issues/42>
* <http://gitlab.qima-inc.com/app-engine/engine-service/issues/41>
* <http://gitlab.qima-inc.com/app-engine/engine-service/issues/37>

在平台化里, 调度中心承载所有对外的 API, 以及几乎所有 ae 底层的业务逻辑. 因此将有大量的数据模型和逻辑的增改. 但本次平台化相比当前的架构, 仅仅在功能上有所调整和增加. 因此整体设计里不做过多赘述. 至于其 API 和业务逻辑, 我们将在接下来的核心功能设计一节合并介绍.

**6. 系统核心功能设计**

下面根据产品层设计的使用场景, 来拆解总控的核心功能

> 这里提到的并非全部产品功能, 仅仅是强依赖底层支持的部分, 为了清楚讲述底层设计因此把这些功能单独提出来.

**6.1. 初始化宿主**

一切都是基于宿主, 因此我们把这个功能放在最开始讲.

第一次登记, 宿主的信息通过 housekeeper (分控) 和调度中心 (总控) 建立会话后存储到调度中心的数据库里.

调度中心会根据宿主的种类/属性/标签/预置配置等信息为其订制专属的初始化流程. 就像宿主告诉调度中心我需要变成什么样子, 然后调度中心来指导协助宿主完成目标.

当宿主完成初始化, 调度中心将修改数据库里的记录状态, 避免重建会话时重复的初始化.

同样, 宿主也可以主动告诉调度中心是否需要被初始化.

为了留出充裕的可扩展性, 我们在设计上允许调度中心实施完全自定义的初始化 (而非绝对依据宿主的诉求)

**6.2. 单元 (Unit)**

单元是可选的, 默认就一个(主站)单元. 单元管理的是宿主 + 二方服务 + 其他资源等一整条链路.

公司现在有 主站, 金融云, 有赞云, 我们把对应到三个单元.

单元用于逻辑隔离不同的业务域, 物理隔离不同的业务域能使用的资源.

**6.2. 环境 (Env)**

环境是必选的, 无论几个单元. 环境是从底层运维视角对物理资源依据不同的网络策略而划分的机房的集合. 因此一个单元下可以存在多个环境. 一个机房只能隶属于一个单元下的一个环境, 不可跨环境跨单元存在.

**6.2. 区域 (Region)**

为精细化资源的管理, 我们引入区域的概念.

一个区域包含一组宿主, 并且每个宿主只能属于一个区域.

区域和单元的关系是: 一个单元可以包含多个完整的区域, 但一个区域不可被划分给多个单元.

默认只有一个区域. 区域的作用主要针对应用运行环境, 显式的从资源池里划分出可调度的边界: 属于不同区域的应用只能在对应区域的宿主里调度.

出于方便管理的目的, 我们把调度中心了解的资源池分为多个区域, 每个区域包含一组容器宿主. 一旦区域里某个宿主处于危险边缘 (比如负载偏高/饱和, 系统故障), 我们就可以暂时将它移除区域, 不再接受调度 (但不影响他上面的服务). 新的实例就不会再调度到这个宿主上面, 避免了更大的故障. 当问题解决后, 可以重新加入进来参与调度安排. 简言之, 区域为原始容器池划分出了调度算法的硬性生效边界.

**6.3. 集群 (Cluster)**

类似单元和区域, 集群默认也一个. 和单元,区域均不同的是, 集群管理的是宿主下的节点.

如果两个节点分别归属于两个不同区域的宿主, 那么他们必然不能组成为一个集群. 这就是集群和区域以及单元的关系.

所以, 有了单元和领域为什么要有集群?

集群在 ae 系统里是纯粹的逻辑层面的概念, 我们依赖它去给一张巨大的路由表做分片, 给可调度容器一个更安全更细粒度的控制阀.

**6.3.1. 网关集群**

网关节点内部持有系统可以路由的全部目标, 为了降低 GC 影响, 堆内存的消耗, 以及分散不同业务的影响, 可独立配置资源, 我们允许创建多个集群, 不同集群维护完整路由表的子集, 不仅方便管理, 还降低风险. 和领域不同的是, 如果以领域能力来规划, 那么一个宿主上的所有网关节点必须同构, 而不同的集群可以借用同一个宿主上的不同节点, 比领域能更细致的做调配.

**6.3.2. 容器集群**

容器节点组成的集群, 也类似一个分布式状态存储. 每次放置实例, 该如何选择边界? 是要从领域里的整个资源池里捞取合适的节点? 还是所有宿主都部署一遍?

同样是出于方便管理的目的, 我们在每个领域内从逻辑上划分为多个小组, 每一组包含具有相同属性或标签的 container 节点. 比如要放置的实例参数匹配了某一个集群, 那么资源的选择只会参考这个集群, 而不会影响其他的节点. 因此, 我们甚至可以按照资源的优劣来调配应用可以享受的待遇. 和领域类似, 一旦集群里某个节点处于危险边, 我们就可以暂时将它移除集群, 不再接受调度 (但不影响他上面的服务). 新的实例就不会再放置到这个节点上面, 避免了更大的故障. 当问题解决后, 可以重新加入进来参与调度安排.

**6.4. 划分依据**

尽管系统提供了逻辑上各个层级细粒度的资源管理, 但是受底层运维策略制约, 实际做资源分配时需要遵守一定规则:

* 建议一个 ops ae container 应用里的所有 host 组成一个唯一独立的 region, 但不强制
* 一个 ops 单元下的所有 ae 应用组成一个独立的 unit
* 一个 ops ae gateway 应用下所有个 host 只能组成一个唯一独立的 region, 并且其下只能存在一个唯一独立的 cluster (统一接入层路由规则限制)

**6.4. 应用部署**

作为托管平台, 应用部署算是最基本的能力. 为了减少歧义, 提升我们设计的灵活性同时保持简洁, 这里我们将我们熟知的部署动作拆分为: 启动实例 + 绑定路由 两步.

这将大大提升系统组件的灵活性以及可复用程度, 后期的函数计算能力以及多种多样的流量策略都依赖路由的调整, 在此将其明确分离是个好事.

以往我们都是先申请路由, 然后再启动服务, 隐藏的一个问题就是没办法做优雅启动, 即在服务准备好之后才接受请求, 需要网关配合, 这往往需要经过几轮的健康检查才能做调整.

**6.4.1. 可调度性计算**

我们先来看启动实例, 那么第一个问题是实例在哪里?

从独立运行模式到托管模式的一个最大的差异是, 实例不再是部署在自己申请的主机里, 而是通过应用引擎动态分配的, 这个过程对使用者来说是一个黑盒, 开发者无感知. 为使这个过程是合理可靠的, 我们需要有一个算法来约束引导.

然而现实是异步不可知的, 难以保证哪个算法自始至终都是最合适的, 也不能说哪个算法完全没有缺陷. 因此, 调度算法也将独立一个模块, 可独立配置.

假设算法经过一系列分析和计算, 给出了可用于放置实例的宿主/节点, 接下来, 能否放置成功呢? 还是上面的问题, 整个系统是分布式的, 除非我们有一个全局事件日志保存每个事件发生时的全局快照, 否则当前做出的决断在下一时刻很可能就是不可接受的.

所以即便可调度, 也未必能放置成功. 能否放置成功取决于在执行调度的瞬间, 目标是否有能力满足调度参数, 比如这一刻 cpu 使用率是否饱和? 内存是否留有充足的余量? 虚拟机个数是否达到软限制等等.

现在可以总结下, 启动一个应用需要走哪些环节:

1. 评估当前系统是否满足调度的参数约束
2. 系统给出合适的宿主和节点组合
3. 给这些合适的宿主和节点发送创建实例信号

任何一步失败, 应用的启动就算作失败.

详细的调度算法, 我们将在详细设计里讨论.

**6.4.2. 创建实例**

选好宿主和节点组合, 即将开始创建实例.

在 ae 系统里面, 基本上所有的操作都是尽可能最小化, 可组合的. 启动应用也是一样的.

系统不会直接让 container 节点下载 app pkg, 然后解压, 这个工作交给 agent 来做, 因为 agent 的作用之一就是尽可能转移 worker 的负载和其他非主线逻辑.

downloader 就是这样一个 agent. 作为一个紧密配合系统工作的节点, 除了具备通用的下载器功能之外, 还会顺带将特定性质的数据处理成 worker 节点能够直接 (几乎无额外开销) 使用的数据布局.

所以 downloader 先把 pkg 各个宿主下载一份. 然后系统通知 container 加载这个 pkg

**6.4.3. 优雅启动与路由绑定**

container 节点负责启动应用的网络服务器, 因此每次加载卸载实例, 都没有操作服务器相关的操作, 速度会更快.

当实例启动完毕, 进入就绪态, 应用框架上的钩子会触发 container 节点的就绪通知, 反馈给分控, 分控告知总控实例已经就绪了.

如果已经关联了路由表, 那么这个实例也将加入路由表的 target 里.

更稳妥的做法是等待所有实例就绪后再绑定路由.

**6.4.4. 优雅停止与实例多版本共存**

一般部署一个新版本, 往往会销毁前一个旧版本. 那么如何保证老版本的流量都已经妥善处理?

答案就是直接调用应用框架的关停方法, 还记得我们提到应用不需要接触服务器吗?

框架和 vm 负责应用的优雅关闭, vm 确保异步的资源已经被回收, 然后只要从 box manager 里移除对应的 box (以及它的虚拟机)

完整流程如下:

* 放置新版本实例成功
* 调整路由表指向
* 优雅关闭旧版本实例

实际上, 我们可以更精细的控制合适销毁旧版实例, 比如在快速回滚的场景下, 只有开发者确定新版已经稳定, 至少短期内不会回滚了, 才真正执行销毁操作.

刚刚提到的能力属于实例的多版本共存, 后面还有更多有用的场景依赖这个设定, 我们继续往后看.

**6.4.5. sc 支持**

支持 sc 主要是维护 sc 和服务之间的关系以及请求透传 sc 标识. 但现在 ops 对 sc 的设计是以宿主维度的, 也就是没办法直接在一个宿主内提供多个 sc 标签. 因此应用引擎里要做一些配合.

**6.4.5.1. sc 标签生成**

仍然是通过 ops 创建项目来触发新的 sc 标签生成.

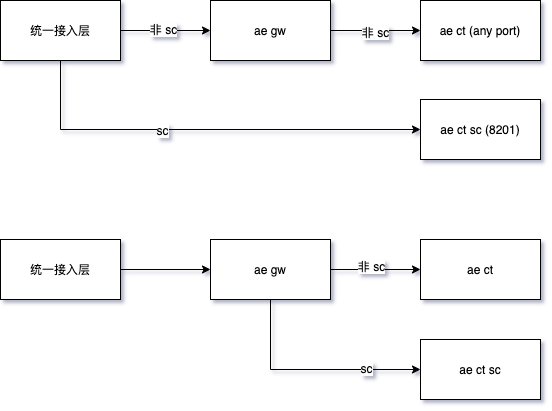
**6.4.5.1. sc 标签存储**

上一步 sc 生成之后, 在 ae 系统里开始 sc 模式部署时, 将 ops 生成的 sc 传递给 ae. (ops 项目管理可以切换前端服务的部署 api 为 ae 部署 api, 这样就可以无缝对接了)

这样在 ae 进行应用放置时, sc 和服务的对应信息就可以存储在调度中心数据库里, 供生成路由表和创建实例使用了.

**6.4.5.1. sc 标签透传**

统一接入层不记录应用引擎的 sc 信息. 这样直接把请求交给 ae gateway 节点, 根据 sc 标签辅助选路实现动态链路能力.



**6.5. 应用回滚**

前面提到了快速回滚, 但局限是已经销毁的实例无法实现这个能力, 需要走标准流程的回滚.

回滚等价于重新放置一组实例, 只不过相比常规的部署, 回滚操作会默认销毁当前实例.

**6.6. 应用下线**

得益于系统的架构, 应用下线几乎和部署的最后一个环节一致.

如果应用是支持多版本部署, 那么下线后将回退到上一个版本, 以此类推.

如果不支持多版本, 那么应用优雅关闭后, 系统会解绑路由表设置.

**6.8. 路由配置**

应用服务是否可用取决于两点: 实例是否就绪, 路由是否配置正确.

下面讲一下路由功能的设计.

路由信息完整的存储在调度中心的数据库里, 哪条路由对应哪一组实例, 哪一个版本的应用.

每个路由项是一个四元组: (domainName, path, exprString, instanceGroup)

前两部分正是 http 里面的域名和路径, 第三部分是我们增加的自定义选路策略, 最后一块是路由目标实例组.

其中表达式部分可以设计为受限的 js 表达式, 或者受限的 DSL.

借助路由表, 我们可以实现丰富多变的流量管理策略, 比如我们熟知的灰度和蓝绿. 其核心无非是检查流量的特征, 然后根据既定的规则做分发.

**6.8.1. 灰度发布**

(对齐公司现有的灰度策略)

TODO

**6.8.2. 蓝绿发布**

(对齐公司现有的蓝绿策略)

TODO

**6.8.3. 其他流量分发策略**

开启多版本共存后, 我们即可透过表达式订制各种各样的策略. 但相应地, 网关负载和延迟会相应上升.

**6.9. 分析诊断**

提供对应用运行时的深入分析和诊断能力, 以及度量统计数据的观察.

* 在线调试: 借助 v8 inspector, 连接到运行中的 vm 进行调试
* 分析: js 对象在 v8 堆内的使用情况, gc 开销, 堆快照, core dump 分析等
* 在线诊断: 提供从 os 内核到平台进程到实例虚拟机不同级别的指标和实时数据提取

为实现上述能力, container 节点将从 /proc, v8::getHeapSpaceStatistics, inspector 等模块中获取必要数据, 具体支持的指标, 收集方法等参见详细设计.

**6.11. 关闭/重启/升级宿主**

当物理机出现不可逆转的损耗, 或者宿主需要迁移到其他地方, 平台升级, 都牵扯到服务的关闭和重启. 这是一个理论上低频的操作, 并且一旦出现会在一定程度上降低整个平台上大批量应用的可用性, 甚至导致另外的宿主过载. 这里只讨论如何正确的关闭和重启宿主系统.

**6.11.1. 本地快照**

前面提到过, 为了快速的恢复, 宿主都是有快照的, 系统会周期性把当前内存生成快照写入本地文件, 快照里只包含引导性配置和元数据, 不会包含内部的状态, 这样可以从任何空的宿主完成恢复. 快照包含:

* gateway: 路由表
* container: 实例表
* logger: 读取偏移量, 上报进度等
* housekeeper: 节点表
* sidecar: baas 资源表

**6.11.3. 系统优雅启停**

**6.11.3.1. 优雅关闭宿主**

housekeeper 终止快照服务, 并立即显式触发一次快照回写, 然后发给节点发送优雅关闭信号, 等待所有节点关闭或超时杀死他们, housekeeper 进程退出, 宿主关闭.

**6.11.3.2. 优雅关闭节点**

节点通用框架里提供 gracefulShutdown 的钩子, 当节点感知到要关闭时, 优先尝试调用该钩子, 节点的实现者负责实现特定节点的优雅下线逻辑. 触发钩子后, 通用层首先终止快照服务, 显式触发一次快照回写, 然后调用节点自定义的优雅下线逻辑, 结束后退出进程.

不同节点的优雅下线将在详细设计里叙述.

**6.11.2. 状态恢复**

启动一个 housekeeper 服务后便进入自检阶段, 如果发现了本地快照, 那么会从快照中恢复之前的状态. 如果快照已损坏无法使用, 那么跳过状态恢复, 进入未初始化状态, 等待调度中心重新配置.

**6.11.4. 重新分配**

重新分配是指宿主服务的快照无法使用时, 进入空白状态, 调度中心感知到整个系统缺少的状态, 通过计算重新把缺失的部分调度到合适的宿主上. 细节参见详细设计.

**7. 系统稳定性, 高可用, 容错设计**

**7.1. 平滑切流**

不同组件的平滑切流方式如下:

* 网关: 配合统一接入层实现健康检查机制, 在主动关闭网关服务前, 设置健康检查为 offline, 观察到网关不再转发流量过来后, 再关闭服务. 对应用来说, 也是优先调整路由表, 然后再销毁不需要的实例.
* 容器: 走标准的优雅下线

**7.2. 风险隔离**

可用的隔离机制如下:

* 集群: 隔离异常的节点
* 实例虚拟机: 隔离异常的实例

**7.2. 多点互备**

* 系统级: 每个宿主都尽可能分布到不同物理机 (公司运维平台保证)
* 平台级: 多宿主服务组建使能集群, 调度中心也是由多点构成保证高可用
* 应用级: 多实例分布在不同的宿主上, 调度策略尽可能保证实例组分布到不同的物理机房中

**7.3. 心跳与可达性检查**

* 网关与容器: 网关到容器之间的可达性影响路由表的正确性, 因此网关会周期 ping 容器, 如果发现网路不通, 会临时封闭这条路, 直到再次检查畅通.
* 会话: 所有基于网络的会话都会配置保活机制, 而且是双方互查的. 如在一个会话里, 分控周期 ping 总控, 总控也会周期检查距离分控最后一次 ping 过了多久是否超时, 确保会话的活跃与可用性, 也能避免资源泄漏.

**7.4. 重试**

* 分控与总控: 分控到总控的会话时控制的核心, 为了保证最大程度的可控, 分控会在会话断开后主动间歇尝试连接总控直到再次建立会话.

**7.5. 降低 k8s 容器重新调度对 AE 系统造成的影响**

目前公司内对业务型应用的策略是无通知杀死容器, 透明启动新的容器.

这个策略对 AE 的影响还是很大的.

在 ops 平台杀死一个容器, 并重启到一个新的容器上时, 原宿主拥有的 分控, logger, downloader 等一系列节点需要重启初始化, 网关节点的路由表需要重新下发, 容器节点的实例信息需要重新下发, 应用包需要重新拉取部署.

如果ops容器平台能确保通知到总控, 那么总控需要根据新的信息及时规划整个系统的资源分配, 如果总控没能及时取得这个底层重调度信息, 那么 ae 层状态无法及时恢复, 轻则负载压力上升, 重则导致服务能力下降甚至出现不可用.

建议的做法是:

1. 支持有状态, 即便 k8s 重新调度也能保证是相同的网络标识(hostname), 本地也能拿到之前的数据
2. ops 必须确保通过重试机制保证 ae 一定拿到了重新调度的通知 (如返回 ok), 否则不做调度.

**8. 最终一致性设计**

系统的元数据/核心数据都是由调度中心数据库维护的, 考虑到整个系统设计的简单, 并且恰好可用. 我们把调度中心和分控所在宿主之间的状态一致性设计为 eventually consistent, 而非 serializable consistent, 非 strong consistent.

这意味着存在某个时刻, 调度中心里的状态和宿主的实际状态不一致. 比如总控分控之间出现网络异常, 数据库操作失败, 分控操作节点失败, 节点逻辑执行异常. 简单讲, 只要走网络有延迟, 那么就会出现短期的不一致, 但为了提升可用性和灵活性, 我们不去做强一致约束.

然而对于数据不一致带来的问题还是要提起重视的, 我们的思路不是牺牲可用换区时刻一致, 而是出现不一致时如何解决使之满足最终一致.

我们采用三种办法: 1. 人工运维辅助; 2. 自动协调; 3. 幂等接口

平台模式的最初版本实现应该不会上自动协调, 这个方案还有很多点要考虑, 我们放到迭代优化里做. 人工辅助是一个简单的手段. 而幂等实现则很大程度上保证状态一致或很容易的趋于一致, 结合人工辅助可以实现最终一致性.

人工运维辅助方法可以参见 10. 小结可控性设计. 幂等接口的设计会在详细设计里说明.

**8. 系统可扩展性设计**

越是庞大的系统越要考虑其各个层面的可扩展性, 应用引擎也不例外.

**8.1. 模块组件级别的可扩展性**

系统各个功能需要有清晰的分层分块, 往后会比较容易增删改以及复用.

举例说明: 调度算法, 节点, 控制命令, 观测组件

**8.1. 总控服务可扩展性**

搭配分控配合 (自动注册), 总控可以无痛 (注册做负载均衡) 实现集群的扩缩, 直接增减主机即可.

**8.2. 宿主服务可扩展性**

宿主虽然有状态, 但有调度中心的辅助 (集群规划, 宿主管理, 优雅启停), 也可以比较轻松的扩缩容.

弹性伸缩需要更底层容器平台暴露创建 docker container 实例的 API, 以便 ae 系统根据负载动态的调整平台自身的资源配额.

**9. 系统可观测性设计**

为方便系统巡检, 应用引擎提供很多种观察系统内部状态的方法和数据分类.

可观测性在无论是在系统设计还是在编码实现期间都是作为首要因素考量的, 并且可观测数据会随着系统的演进而调整变化. 后续会逐步参考 OpenTelemetry 规范走标准化.

另外, 在设计可观测性的同时, 需要结合开销和损耗评估.

状态的观测可以借助 1. 运维接口; 2. 日志; 3. 标准化上报路径 实现.

其中运维接口参见下一节 10. 系统可控性设计.

标准化放在后期迭代里实现.

目前系统提供如下几种可观测数据.

**9.1. Metrics**

提供度量数据, 比如事件循环里活跃句柄统计吗, gc 统计, 启动时间等

**9.2. Platform**

提供系统相关的静态信息, 比如 ip, env, hostname, pid, config, 启动参数等

**9.3. Resource**

提供资源使用情况, 比如内存, cpu, 线程, fd 等

**9.4. Profile**

提供剖析能力, 比如 v8 堆内内存快照, cpu 分析, 堆分析, 诊断报告等

**9.5. Log**

日志提供一个标准化的输出接口, 规范化的输出格式, 允许通过 Log 获取详细的上下文

规范格式和标准在详细设计里给出.

**9.6. Trace**

trace id 整合到各个模块以及日志中, 允许追溯前后上下文

**10. 系统可控制设计**

系统设计上, 对于平台运维来讲是几乎没有黑盒的. 绝大多数状态都可以观测并调控.

**10.1. 总控**

总控通过 http api 支持运维能力, 包括对元数据的管理, 宿主的控制.

**10.2. 分控**

分控通过控制命令实现直接对特定宿主以及节点的运维.

**10.3. 节点**

节点是会话发起者, 可以通过分控实现操控.

**11. 系统可测性设计**

作为设计原则之一, 在编码和测试期间, 系统可测性作为另一个核心要素存在. 大到平台, 服务; 小到进程, 模块. 必须能够被独立检验 (能做白盒测试).

**12. ops 运维体系对接设计**

* 每个区域/集群对应一个 ops 应用. 例, ae 的交易业务网关区域/集群, 对应到 ops 应用 ae-gw-trade
* ops 上每个 ae 应用都使用相同的 codebase
* 通过环境变量 SERVICE\_NAME 区分运行的服务, 同样地, 通过其他环境变量区分服务的预置属性和标签
* 流量规则 (灰度/蓝绿) 和现有规则保持一致

**13. 业务应用高可用设计**

放置应用实例时有一个副本因数, 不同环境都有个默认值, 测试和预发环境是 1, 生产是 2. 实际放置时可以调整为其它数值. 一般大于等于 2 即可满足高可用. 额外的副本可以作为服务能力的补充, 提升应用吞吐. 参见 3.1.4. 应用部署图.

**13. 业务应用的扩展性设计**

系统根据各个宿主的负载情况, 结合实例的请求量, 给相应的实例组临时扩容.

具体见详细设计.

**14. 业务应用的服务治理设计**

**14.1. 监控告警**

**14.1.1. 日志**

业务日志通过 logger 节点传送给天网, 基于日志的告警暂时仍然使用天网提供的告警能力

**14.1.2. 可用性**

一种方式是通过定时任务检查完整访问链路下所有业务应用的可用性, 系统自身不做这种检查机制.

另一种方式需要告警平台能够支持这种 http 的告警能力, 让 ae 直接使用.

每次创建服务(实例组)后增加对应的告警检查.

**14.1.3. 指标**

包括流量, 运行时的一系列指标, 参见可观测性设计.

目前指标无法告警, 需要告警平台支持.

**14.2. 透明化日志管理**

应用的日志直接落到本地文件系统, 日志节点统一管理日志的标准化和天网上报还有日志规模的管理, 最小化数据丢失和写延迟.

**14.3. 开箱即用二方服务**

sidecar 节点中初始化这些能力, 并交付给 container 节点, box 封装这些资源, 提供给应用的 plugin 中直接使用, 无需再做额外的配置/创建/初始化/池化/限流等.

应用引擎会监控这些二方服务的使用, 并保护他们不会被滥用和超载.

**14.4. 透明化优雅启停**

前文已经提到, 从 housekeeper, 到 container 节点, 到 vm 最后到 box 框架, 层层关联 hook 了优雅启停动作. 因此业务代码无需关注. 系统自动保障.

**14.5. 资源访问限制**

* 受限的文件系统目录树
* 受限的环境变量访问
* 可用的全局对象
* 受限的方法调用
* 受限的模块导入
* cpu 时间
* v8 堆内存

目前不会对所有能力做限制, 并且可访问范围取决于使用的 vm, 详细设计中说明.

**15. Appendix**

详细设计分为 1 + N 来做. 一个主干详细设计, 只包含系统的最核心部分, 如总控/分控/节点内部的设计, 核心流程等.

剩余的设计将独立提出详细设计, 并且其中一部分可能会分散到各个快速迭代阶段, 包括但不限于:

* sc 部署调用支持
* 日志格式规范化
* 链路追踪
* 业务应用透明化监控告警
* 标准化错误类
* 多种类虚拟机
* 系统间自主协商数据一致性
* 公司二方产品对接
* 可观测性能力标准化
* APM
* 其他