**YAE BaaS 能力技术架构概览**

**0. 前置阅读**

* [iron-base\_wsc-pc-base 下沉 YAE 产品设计方案 - 飞书云文档 (feishu.cn)](https://qima.feishu.cn/docs/doccni6QvZL0cImJmkfx5EMkzlc)

1. **背景**
2. 基础框架/库的升级难以推动, 进展缓慢, 影响 Bug/漏洞修复效率, 无法实现版本治理; 目前以应用为维度升级, 成本过高, 周期过长.
3. 相同的依赖, 构建期仍然会独立安装, 在同一台主机上仍然会以副本的形式存在多份, 影响构建产物的体积, 系统资源开销 (CPU / 硬盘), 拖长变更时间, 影响效率和体验.
4. **目标及价值**

* 缩短构建/分发/加载(启动)时间
* 依赖版本选择可控
* 可以低成本快速替换依赖版本
* ~~提供统一的基础能力访问接口~~

1. **系统架构层次及原则**

BaaS 管理是一种面向运行时服务切面的变更管理体系, 和 App 管理不同, 它操纵的是 BaaS 组件的生命周期, 因此也与 App 管理面交叉.

**基础层**

除了提供完整应用级别的实例声明周期管理, 增加提供实例运行时 (指 JavaScript Runtime Library) 管理和缓存管理的能力.

|  |
| --- |
| **架构考量点**  考虑未来如何接入基础监控  流量平滑性保障  CPU/内存/IO 使用率和频次评估  运行时管理和实例管理的竞态, 协同  运行时管理成本, 包括初始化, 更改, 还原, 加载  运行时管理的安全性, 内存泄漏, 不兼容变更 有办法检测并处理; 具备细粒度的可控制性  缓存管理的正确性保障  缓存共享及独立控制  缓存管理的空间开销可预期  缓存管理具备细粒度的可控制性  pnpm 参与缓存管理的可行性  BaaS 在 CT host 上时非持久存储的 (CT host 一旦重启, BaaS 暂储就失效了) |

**平台层**

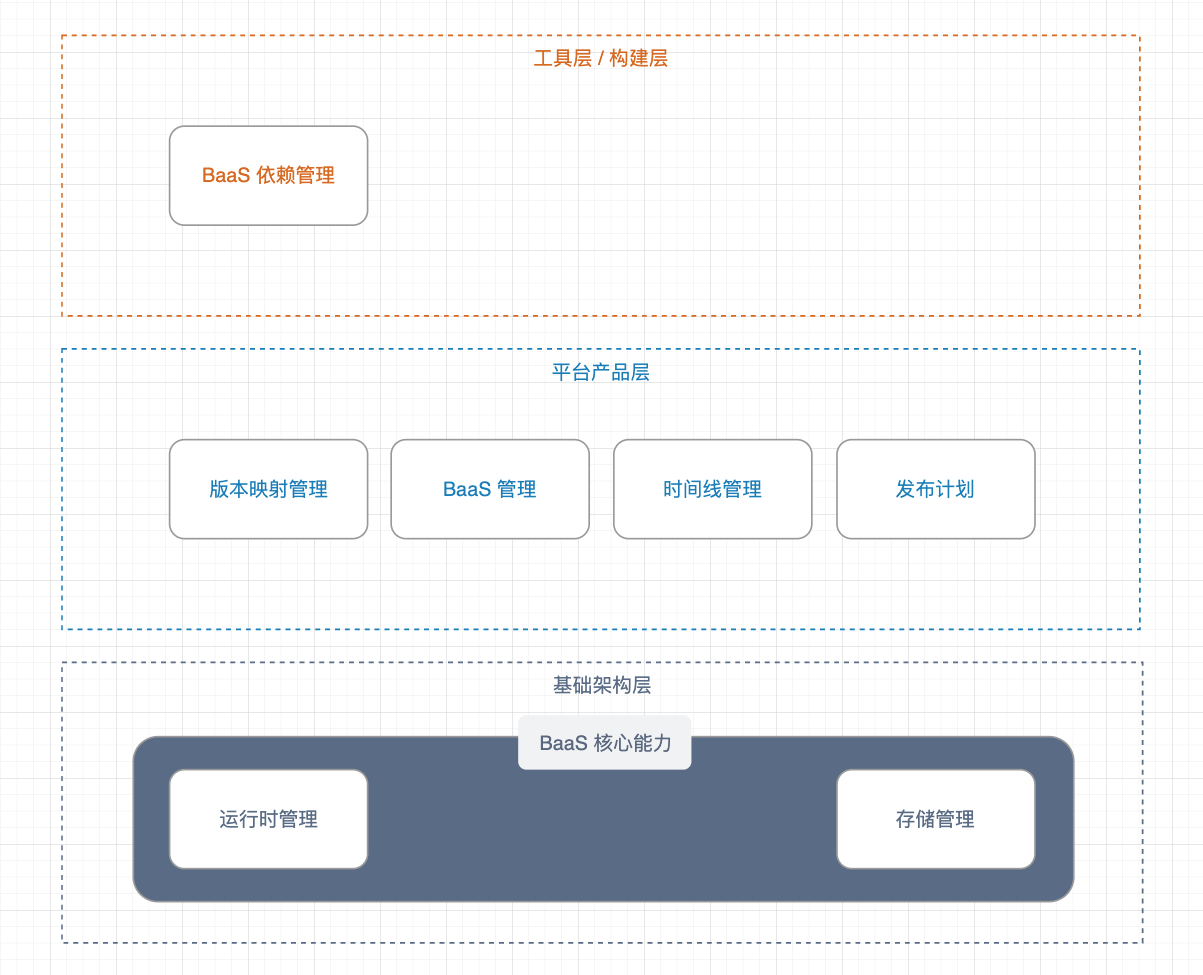
增加负责 BaaS 管理业务模型

负责版本中心的版本管理

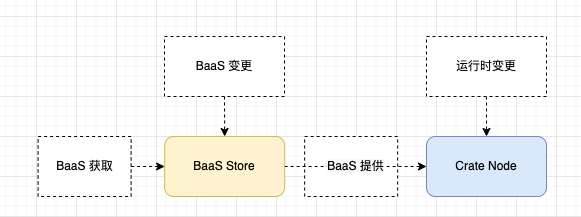
|  |
| --- |
| **架构考量点**  具备全局性开关和控制逻辑, 可以快速安全管理基础侧  处理和现有应用&服务管理之间的交叉点, 确保正确的调用基础侧能力  运行时缓存 & 版本管理 版本一致性幂等保证 |

**研发工具层**

能够理解 package.json 新的 DSL (baas 声明)



1. **核心能力建模**



基础架构侧的核心逻辑, 围绕该图展开.

BaaS 变更, 其过程分为:

* BaaS Store 就绪
* Crate Node Runtime Lib 替换

两个过程独立展开, 互不了解, 通过外部 barrier 机制协作, 竞态的管理也放在外层.

**BaaS 变更**

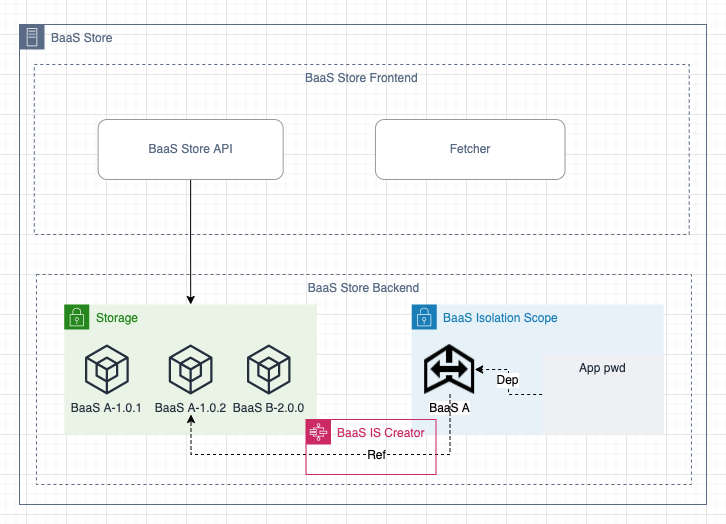
BaaS 的变更对应两个时机:

* 预取 (Crate Host 预热, 准备基础的 BaaS 依赖, 完成 BaaS Store 初始化)
* 更新 (通过 BaaS Store API 触发的事件, 局部增加/删除 BaaS 模组)

**aBaaS Store**

BaaS 资源在一个 CT Host 里是全局共享的, 抽象为 BaaS Store.

BaaS Store 分为 FrontEnd 和 Backend



所有对 BaaS pkg 的访问均通过 BaaS Store FrontEnd 提供的一组 API 完成:

* GetLocation: 获取一个 BaaS 的位置
* Put: 从一个本地位置更新一个 BaaS 的 pkg
* Del

这些 API 作为 BaaS 资源操纵 "原语", 可以在基础上扩展更复杂的 API.

Fetcher 负责从外部获取 BaaS pkg, 存放到 BaaS Store TemporaryStash (暂存区).

BaaS Store 存储了一个 CT Host 上所有业务所需的 BaaS pkgs, 通过 BaaS Store BackEnd 维护.

BaaS Storage 是所有 BaaS pkg 的物理存储区, 不同 baas 及其不同版本.

业务服务实例对 BaaS pkg 的依赖, 通过 BaaS Scope 解耦.

BaaS Isolation Scope (IS) 提供了一个相对隔离的环境, 隔离的主要是 node require 模块解析机制.

应用的工作目录 (代码目录) 位于 IS 内部, IS 内的 BaaS pkg 均以映像 (Mirror) 的方式存在, 引用到 BaaS Storage 的一个 BaaS pkg.

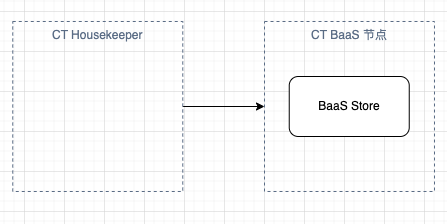
这一过程是由 BaaS IS Creator 完成, 它负责管理 BaaS IS, 包括如何建立 BaaS pkg 映像 (Mirror), IS 内部结构等.

**BaaS Store 节点**

BaaS Store 由 BaaS 节点负责管理.

Note: BaaS Store FrontEnd 其中的 Fetch 能力, 可以放到 Downloader 节点管理, 不做强制.

BaaS 节点是一种 Agent 型节点, 每个 CT Host 里会单独启动一个, 管理全局的 BaaS pkg.



**Runtime 变更**

BaaS Store 变更完毕, IS 就绪后, 开始变更 Runtime, 无需关心资源就绪问题.

在 Runtime 变更上, 最关键 QoS 是: 保证业务流量平滑, 容量足够支撑工作负载. 这也是我们在做设计实现时要时刻叮嘱自己的.

有两种变更 Runtime 的手段:

1. app 重发
2. app 的 replica set 局部重载

**节点级变更**

重新发布 app 是一种前期成本低廉的解法, 通过完全复用现有的技术, 实现运行时的变更.

这种做法开发成本低廉, 运行时隔离性好, 变更安全.

但存在的短板也显而易见:

* 链路长
* ~~重新发布导致 app 依赖的 node\_modules 版本可能发生变更 (破坏了我们只变更 BaaS 的初衷)~~
* 短时间内需要更多, 或者双倍的资源
* 频繁的进程启停, 对系统稳定性带来一定的挑战, 增加了不确定性

**实例级变更**

相比于前一种技术, 这种显得性价比更高一些.

这是一种直接修改 js 对象引用, 以达到变更的技术.

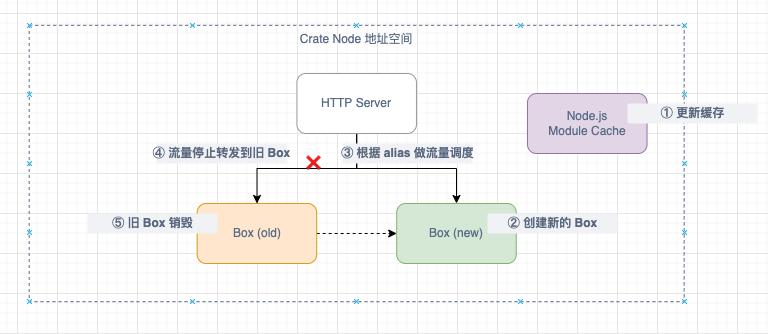
由于只操作 v8 对象, 基本可以保证流量无感知切换. 这也是我们最理想的业务变更方式.

然而目前来讲, 直接使用 Node.js 提供的能力, 劣势显而易见:

* 资源控制 (v8 内外) 不透明, 各种私有 api 调用, 十分黑盒, 不易测试.
* 回滚可能非预期 (甚至引发意外)
* 但我们可以一定范围内保证可控, 只要用户代码遵守我们的规范 (很难):
* 代码在一个 v8 实例里无副作用
* 无 Native 引用
* 可以通过 require cache 操作
* 避免全局变量访问
* 提供初始化和销毁接口
* 允许多分对象实例共同存在
* 无状态实现优先

Note: 对于热更新的使用, 更倾向于新建对象, 而非直接修改对象. 从稳定性考虑, 这样有机会实现可控的灰度和回滚.

这个技术的原理如下:



在替换模式下, 在当下没有办法销毁 v8 实例的情况下, 需要重点关注资源/对象的缓存和引用情况, 避免泄漏!

此外, 由于 require cache 只有一份, 需要处理多个实例共存时, require cache 隔离/维护的问题.

这个方案密切关联 "节点多实例" 技术. 如果考虑安全, 可观测性等方面, 是强依赖 多实例 技术的.

**流量调度**

Runtime 变更结束前(或后), 需要调整流量的走向, 按照规则走到正确的副本集.

根据上一节不同的 runtime 替换方案, 我们这里有不同的流量调度变更策略.

节点级变更技术, 和应用发布一致.

副本级变更技术, 也是沿用三层调度模型, 创建新的 runtime 变更灰度 service, 但需要在不销毁 Crate Node 的同时创建 service 下的实例.

需要注意的是, 这时整体流量仍然是走到同一批 Crate Node, 现有监控无法直接区分. (属于节点多实例场景)

1. **边界 case**

**预载**

* 需要关注一次 CT Host 初始化的耗时, 销毁 CT Host 的影响

**CT Host 就绪条件**

CT Host 以 K8s Pod 形式存在, 其生命周期受 K8S 制约, 因此需要满足一定条件:

K8s 要求 以 Pod (Container) 内 TCP 端口 8021 被监听作为 Pod 就绪条件.

我们的 CT Housekeeper 监听 8201, 负责对接 Ready 这项工作.

但引入 BaaS 运行时切面管理之后, Ready 这个词会包含两个语境: Pod 就绪, 可接受调度.

引入 BaaS 管理之后, 应用依赖的一部分能力转移到 BaaS 层, 因此实例启动的必要条件增加一条:

* BaaS pkg 必须就绪 (否则应用无法正确实例化)

同理, CT Host 接受 yae 调度的条件也应该依赖 BaaS Store 就绪.

**临界条件控制**

BaaS 管理包含 BaaS Store 操作以及 Crate Node 运行时的操作. 我们的设计原则是互不干扰, 互相独立.

来看几个涉及临界的场景:

1. BaaS Store 内部的并发

BaaS Store 提供两种 W 操作: put, del, 以及预载操作.

预载过程由于所有资源均是互斥的, 因此存在天然的 W 隔离.

更新操作, 我们视每个 BaaS pkg 为 immutable 对象, 一旦创建, 不可修改. 因此, put 原则上只能增加, 不允许覆盖. 而 del 执行前需要判断是否存在引用. (重点检查)

由于 put 动作仅仅是创建引用, 因此没有中间态存在. 所有 get (r) 操作在并发时不会遇到问题.

1. BaaS Store 和 Crate Node 重载的协调

一般而言, 只有 BaaS IS 就绪, Crate Node 才会发生重载, 这意味着 BaaS 变更和 运行时变更时存在barrier 的.

但 Crate Node 的重启是难以预料的. 比如应用发布, 进程异常重启, 调试重启等. 我们不能寄希望于 Crate Node 永不抖动. 在这个场景下, IS 发挥了作用, 因为 IS 是在应用启动前就绪的, 所以BaaS pkg mirror 已经 固化 在 IS 内了, 无论应用实例怎么重启, 都不会受 BaaS Store 变更的影响 (只要 BaaS Store 操作遵循规则), put 操作只创建, del 操作仅在 app/service dispose 且没有其它引用后.

这里要强调的一点是, 我们对于运行时变更+BaaS Store 变更的临界场景的承诺是: 要么 Crate Node 能读到正确的 (就绪的) BaaS pkg, 要么读不到, 而不会出现一个中间态 (读到了不完整的 BaaS pkg). 总结就是 "避免脏读, 写竞态"

如果暂时无法读到 BaaS pkg, 应用启动会失败, 可以适当的 Try Again.

**平台侧判断版本更新成功的条件**

所有的计划都完成时, 更新 应用 - BaaS 版本依赖关系是绝对安全的做法, 但某些时候可能不是最搞笑的选择 (N 个操作, 成功了 N-1, 失败 1 个, 如果定义为失败, 那么随着 N 的增加, 效率将直线下降).

可以考虑提供强制更新版本关系的能力.

**Crate Node -> instance 比例关系修订**

如果采用了 Runtime Lib 对象级替换方案, 那么数据模型关系会出现略微的差异. 原有部分逻辑存在 CrateNode -> instance 1:1 的假设, 以及根据 instance 操作 crate 状态的行为, 需要 review 并仔细修订, 保护相关的逻辑的正确性.

**副本保持**

对于选用了副本级变更的方案, 需要在副本保持策略上做些微调, 能够让新旧副本仍然绑定在一个节点上.

**对象热更新**

面临的问题:

* 全局对象 (module cache 等全局资源的修改)
* 打开的资源 (如 io 资源, 没有提供释放的接口)
* 长期的引用 (不会被 gc, 如异步回调闭包等, 没有提供终止手段)
* 执行的逻辑 (如循环执行, 缺少终止接口)
* 同个命名空间 (修改互相影响)

1. **基本流程**

**Baas 发布**

BaaS 层面的发布, 是一种对应用实例运行时切面修改, 和 App 发布存在重叠, 为了避免未定义行为和风险出现, 我们引入 "发布计划" 概念, 通过唯一的时间线约束对运行时的变更.

**发布计划 (Online Plan)**

baas 变更前, 需要预先声明一个发布计划. (app 也可以声明发布计划).

Baas 发布计划定义了要变更的 baas, 变更时间区间, 变更策略. 以及参与变更的应用群体.

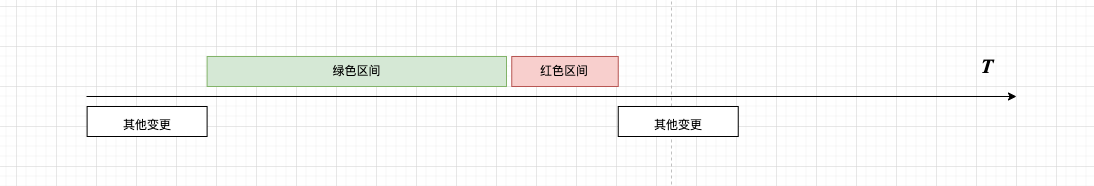
发布计划是一种排他策略的实现, 当一个 baas 发布计划指定了某个应用后, 任何对该应用的变更企图 (应用发布/其他发布计划的 BaaS 变更) 都将被禁止. 反之, 当该应用处于app发布中状态时, 发布计划也是禁止被创建的.

发布计划允许创建者执行撤销. (需要注意撤销造成的连带影响)

**发布区间**

发布计划需要明确声明时间段.

只有在区间内, 才允许执行发布计划里的变更任务.



考虑到发布计划的异步性, 以及延时, 这里引入两个颜色区间:

绿色区间表示可以安全的进行变更的时间范围;

红色区间表示禁止执行增量的变更操作的时间范围; 在这个区间里, 整个发布计划等待最后尚在执行中的任务结束. 确保后续的变更不会出现意外的重叠.

**变更任务**

发布计划是由一批变更任务组成.

每个任务表示要变更的应用.

任务是串行进行的, 因此需要创建者把控好发布区间, 避免长时间阻塞和时间不足.

任务支持重试和取消. 在一个发布计划里, 可以一次性取消所有任务, 表示终止发布计划.

**发布策略**

分为全量和灰度. (注意, 以下操作均发生在安全区间内)

**灰度发布**

当所有任务都准备就绪 (准备好新的实例), 然后才允许推流. 灰度策略复用应用的灰度策略.

灰度策略可以取消. 如果灰度存在时间超过了安全区间, 那么在进入危险区间前, 自动取消.

**回滚**

表示对某个任务执行回滚. 可批量回滚, 期间可跳过回滚失败的任务.

回滚保证幂等的情况下, 允许重试.

**回滚正确性保障**

如果版本中心里存在新的版本, 则跳过回滚.

**本地开发**

* 从版本登记中心获取当前应用的 baas 版本
* 外层安装 baas, 内层安装 deps (双层 node\_modules)

**Baas 服务端预取**

* BaaS 节点收到指令, 拉取 BaaS pkg, 存放到原始存储区
* 获取成功 -> 执行 BaaS Store PUT, 在 BaaS Storage 创建一个到原始存储区的引用
* 根据 pkg 的属性
* L1 的 pkg 必须都预取成功, 整个操作才算完成
* L2 的 pkg 不影响整个预取操作的结果

**CT Host 初始化流程**

* 启动 CT hk
* DC 建联
* Dc 初始化 ct 节点
* Dc 请求 ct 预取
* Ct 预取成功 -> 请求 dc 加入区域/集群
* Ct 预取失败 -> 报错 (可以选择手工加入区域/集群, 但这时可能 BaaS 不完整)

**应用发布 (CT 端)**

* 下载 app pkg, 解压
* 准备 BaaS IS: 基于解压后的应用路径, 根据应用依赖的 BaaS, 创建 IS
* 启动实例

**BaaS Pkg 更新**

* Baas 节点收到更新指令, 检查本地是否存在对应版本的 pkg, 存在则跳过, 返回 not modified
* 拉取 baas pkg, 存放到原始存储区
* 获取成功 -> 执行 BaaS Store PUT, 在 BaaS Storage 创建一个到原始存储区的引用
* 获取失败 -> 操作失败

**BaaS Pkg 移除**

* 检查是否存在引用
* 如果不存在, 删除到 BaaS Storage 到原始存储区的引用及数据

1. **排期规划**

* 全链路开发联调测试, 预计 3 人月
* 细粒度的排期待详细方案产出后敲定

1. **扩展阅读**

* [YAE Application Lifetime 基础建模和优化 - 飞书云文档 (feishu.cn)](https://qima.feishu.cn/docs/doccnNNQgLC0b07vByXcpSENEph#)