**应用引擎底层架构详细设计: 核心部分**

**1. 前置阅读**

* 整体设计: <http://gitlab.qima-inc.com/app-engine/docs/issues/66>

**2. 基本类型与数据约定**

**2.1. 节点种类**

|  |  |  |
| --- | --- | --- |
| name | type | service name |
| 网关节点 | gateway | worker/gateway-nodejs |
| 容器节点 | crate | worker/crate-nodejs |
| 日志节点 | logger | agent/logger |
| 下载器节点 | downloader | agent/downloader |
| 网络分区检查节点 | network-partition-guard | agent/network-partition-guard |
| baas 管理节点 | baas | agent/baas |

**2.2. 宿主类型**

|  |  |  |
| --- | --- | --- |
| name | type | desc |
| 容器型 | crate |  |
| 网关型 | gateway |  |
| 混合型 | hybrid | 两种 worker 节点共存 |

**2.3. 实例虚拟机隔离级别 (类型)**

|  |  |
| --- | --- |
| name | isolated level (type) |
| 独占进程 | excluded-process |
| 线程 | thread |
| 共享进程 (上下文模式) | context |
| 共享进程 (虚拟机模式) | isolate |

**2.4. 附加属性和标签 (wip)**

|  |  |  |  |
| --- | --- | --- | --- |
| category | name | value | desc |
| host.crate | node.count.max |  | 可以创建的节点数量硬上限 |
| host.crate | node.hybrid.enabled |  | 是否允许容器型宿主以混合模式创建节点 ( 节点多种 impl) |
| host.crate | schedule.priority |  | 调度优先级, 数值越低越优先 |
| host.crate | traffic.load.balance.weight |  | 负载均衡权重 |
| node.crate | schedule.priority |  | 调度优先级, 数值越低越优先 |
| node.crate | traffic.load.balance.weight |  | 负载均衡权重 |
| node | impl |  | 底层实现 |
| node.crate | vm.hybrid.enabled |  | 是否允许容器节点以混合模式创建实例 (一个容器节点内多种虚拟机共存) |

**2.5. ops 配置**

保持第一版应用引擎的运行, 创建另一套 ops 应用和域名集合运行平台模式.

新建应用:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ops 应用 | desc | idc | replicas | http endpoint |
| ae-dispatch-center | 调度中心 | opsbd, opsbj2 | 8 | api.ae-dc.qima-inc.com, registry.ae-dc.qima-inc.com |
| ae-gateway-default | 默认网关宿主 | bd, bj2 | 8 | N/A |
| ae-gateway-internal | 平台内部使用的网关宿主 | bd, bj2 | 4 | N/A |
| ae-crate-default | 默认容器宿主 | bd, bj2 | 16 | N/A |
| ae-crate-internal | 平台内部使用的容器宿主 | bd, bj2 | 8 | N/A |

**2.5.1. 平台服务启动方式**

默认只启动两类:

* ?ae-dp 总控
* ae-hk 分控

启动方式为通过环境变量 SERVICE\_NAME 区分服务进而初始化.

**2.6. 环境变量的使用 (wip)**

|  |  |  |
| --- | --- | --- |
| env | value | desc |
| SERVICE\_NAME | 节点参见 2.1. 的 service name 部分, 总控为 dispatch-center, 分控为 housekeeper | 服务名 |
| HOST\_TYPE | gateway, crate | 强制标记区分宿主类型, 参见 2.2. |
| HOST\_ALIAS |  | 宿主唯一标识 |

**3. Housekeeper (宿主控制器/分控) 设计**



**3.1. 启动分控**

BootLoader 根据环境变量 SERVICE\_NAME=housekeeper 加载分控模块, 初始化.

**3.1.1. 唯一标识**

分控初始化时按照优先级顺序尝试从环境变量 HOST\_ALIAS > 配置 config.host.alias > hostname > 自动生成 中获取宿主唯一标识.

**3.1.2. 宿主类型**

按照优先级: 环境变量 HOST\_TYPE > 配置 config.host.type 获取宿主类型

**3.2. k8s health check responder 服务**

容器平台会检查每个 docker 实例是否在线, 方法是对于 Node 服务检查 8201 端口是否能够正确响应 HTTP 请求 /\_HB\_?service=online.

这一策略严重影响应用引擎系统的设计规划. 因此分为两部分处理:

**3.2.1. 网关型宿主的 k8s Responder**

正常来说, k8s 应该负责检查应用引擎底层架构是否运转, 即宿主的管理者 housekeeper 是否可以工作.

网关节点属于 housekeeper 产生/管理的对象, 因此网关服务器是否启动, 理论上不应该与 k8s 的健康检查关联, 或者严格意义上讲, 不能以同一个机制穿透检查两个层面的状态.

现在既然 8201 一个端口既检查代表应用引擎的 housekeeper, 又顺带侵入式检查了应用引擎内部的组件节点.

那么在网关型宿主里, hk 需要优先把 k8s Responder 启动起来, 能够响应健康检查, 避免在正常状态下被这个机制误杀.

与此同时, 统一接入层也会把业务流量转发到 8201 端口, 但从应用引擎的视角来看, 这个端口的服务应该是 housekeeper 针对 k8s 健康检查而提供, 并不包含处理数据面的流量, 负责这个事情的是网关节点.

因此, 我们将不得不把 k8s Responder 转移到网关节点里, 统一由网关节点处理.

在这种设计下, 必须及时的启动至少一个网关节点, 同样是为了避免整个分控体系被误杀.

简单讲, 这里 k8s 检查的就是网关节点本身.

在这个前提下, k8s Responder 提供的能力是:

* 负责回复 k8s 的七层检查请求
* 允许动态修改这个数据, 以便特定情况下对检查做出不同响应.

**3.2.2. 容器型宿主的 k8s Responder**

因为容器节点的数据面流量都是应用引擎自己控制的, 因此不像上面那样受到严重制约.

K8s Responder 放在 hk 里, 只是应付 k8s 层面的检查, 别无它用.

**3.3. 连接总控**

连接总控是被动过程, 首先通过 Register 服务发起宿主登记, 总控接下来根据登记信息主动和宿主建立会话.

**3.3.1. 宿主登记**

登记是发起一个 HTTP 请求到总控提供的 API. 宿主登记过程会携带配置信息放到 request body 里, 表明自己的身份和预设参数:

interface BaseHostRegisterData {

alias

env

idc

opsName

hostname

ip

port // 提供控制服务的端口 (rpc)

pid

type: HostType

config: HostConfig

}

Register 服务会按照预设频率发起登记, 直到有一个会话建立后暂停这个过程.

**3.3.2. 重试策略**

会话通断的检查由两个组件共同作用:

* SessionControl 服务: 使用传输层技术检查 TCP socket 是否断开 (FIN, probe timeout)
* KeepAlive 服务: 使用应用层策略评估整条链路的健康状况, 一般在网络隔离或者高延迟网络环境里起作用, 直接掐断不可用连接.

会话断开后, SessionControl 服务会感知到, 并再次激活 Register 服务, 直到新的会话建立成功为止.

这一行为确保总控和分控尽其所能的自动建立会话.

**3.4. 状态恢复**

状态恢复期间不会启动 GracefulOpenClose 服务, 因此信号不会被特殊处理, 如果遇到 SIGINT/SIGTERM, 那么直接退出进程. 且这个阶段不会启动 Snapshot 服务, 因为这时内存状态尚不稳定, 如果现在做快照, 极有可能导致快照不完整.

只有在状态恢复完毕, 分控或者说宿主才进入就绪 (ready) 的状态, 这一状态表明宿主可以接受总控派遣调度了. 这时, GracefulOpenClose 服务, SnapShot 服务才会启动.

**3.4.1. 处理损坏的快照**

快照处理失败包括:

* 本地文件系统没有快照
* 快照是空的
* 快照无法反序列化
* 快照反序列化后缺少必要的 agent

上述情况, 都需要分控做空白化处理, 并等待总控重新下发配置.

**3.5. 控制信号**

控制信号流转于总控 <- -> 分控, 分控 <- -> 节点.

分控 -> 总控:

* sendConfig (args: config, return: ok | validated config)
* updateBootingProgress (args: status, return: ok)

总控 -> 分控:

* askConfig (return: config)
* runWizard (return: wizard)

**3.6. 节点启动顺序**

因为 agent 节点必须先于 worker 节点启动, 所以宿主初始化必须有个正确的顺序.

**3.6.1. 网关型宿主的节点启动顺序**

1. logger
2. gateway (为了尽快响应 k8s 的启动检查机制)
3. downloader
4. network-partition-guard

**3.6.2. 容器型宿主的节点启动顺序**

1. logger
2. downloader
3. network-partition-guard
4. baas
5. crate

**3.7. 角色初始化**

角色说明了一个 host 是网关型还是容器型. 不同类型宿主初始化过程也是有差异的.

角色的区分, 单从应用引擎视角看是随意的, 也就是一个 host 既可以成为网关又可以成为容器.

但受到底层运维环境的制约, 如何配置角色取决于:

Wip

* 用作网关, host 所属 ops app 必须直接对接统一接入层

**3.7.1. 网关型宿主初始化**

* 读取/合并配置, 取得预置参数
* 检验本地快照可用性
* 如果快照可用, 则根据快照生成启动向导, 排列优先级. (如果有网关节点启动, 那么会在节点里启动 k8s Responder, 响应底层容器检查)
* 发送宿主登记请求
* 总控控制端接入分控控制服务器
* 启动 SessionControl 服务
* 启动 KeepAlive 服务
* 发送 **配置信息**
* 如果总控有响应初始化向导, 则按照向导覆盖初始化节点 (如果有网关节点启动, 那么会在节点里启动 k8s Responder, 响应底层容器检查)
* 等待所有节点启动完毕
* 启动 GracefulOpenClose 服务
* 启动 LivenessReport 服务
* 启动 Snapshot 服务
* 初始化完成, 发送就绪信息给总控, 如果发送失败, 则尝试重新发送直到认定丢失会话

**3.7.2. 容器型宿主初始化**

* 读取/合并配置, 取得预置参数
* 启动 k8s responder 模块, 开始响应底层容器健康检查
* 检验本地快照可用性
* 如果快照可用, 则根据快照生成启动向导, 排列优先级.
* 发送宿主登记请求
* 总控控制端接入分控控制服务器
* 启动 SessionControl 服务
* 启动 KeepAlive 服务
* 发送 **配置信息**
* 如果总控有下发初始化向导, 则继续初始化节点
* 等所有节点启动完毕
* 启动 GracefulOpenClose 服务
* 启动 LivenessReport 服务
* 启动 Snapshot 服务
* 初始化完成, 发送就绪信息给总控, 如果发送失败, 则尝试重新发送直到认定会话丢失

**3.8. 创建节点**

Host 类提供标注的创建节点接口, 传入节点配置数据.

节点配置分为四个象限: 外部的, 内部的, 进程侧的, 控制器侧的

* 外部的表示可以动态配置传入的, 内部的表示一些底层配置好的硬性参数
* 进程侧的表示可以在节点进程内使用的配置, 控制器侧表示在 hk 进程内用于创建节点以及控制节点进程行为的配置.

因此创建节点操作只允许传递外部配置:

interface NodeExternalConfig {

}

最终由系统合并内部配置后成为一个完整的节点配置对象:

interface NodeConfig {

}

创建成功的节点会维护在 hk 内的一个散列表里. 没有默认的删除行为, 因此有必要监视这个资源的增长, 并订制清理策略.

**3.9. 启动节点**

启动节点和创建节点是两个步骤, 便于细粒度操控.

创建后的节点可以根据 alias key 从池里获取, 然后调用相关方法启动.

只有执行了启动, 才会实际创建节点进程.

**3.10. 关闭节点**

标注的关闭节点的方法是给节点发送一个控制信号 stopNode

但因为各种原因可能无法判断节点是否已经关闭了. 因此我们使用 超时 + KILL 的机制做兜底.

这种属于绝大多数场景都凑效, 且尽力而为的策略. 只有一种例外: 节点进程处于 uninterruptable (D) 状态时, 任何指令无法杀死.

**3.10.1. 超时**

节点进程侧有一个配置 stopTimeout 用于节点自己控制其中的逻辑超时.

控制器侧也有超时的概念, 比 stopTimeout 多 1s.

**3.10.2. 闲置资源回收**

关闭节点只是释放了内核维护的 Process, 但并不会将节点控制器从 hk 的节点表中移除.

因此这里存在一种可回收资源: 节点控制器.

可以通过手动或自动的方式清理掉. 参见总控部分.

**3.11. 关闭宿主**

**3.11.1. 优雅关闭**

GracefulOpenClose 服务接受到特定 Signal, 就会开始执行优雅关闭:

* 停止 Register 服务
* 停止 Snapshot 服务
* 给所有节点发送优雅关闭指令, 等待关闭
* 超时时间达到后强制杀死仍然存活的节点
* 关闭 LivenessReport 服务
* 关闭 KeepAlive 服务
* 断开总控会话
* 等待进程退出 + 超时

**3.11.2. 意外宕机**

意外宕机指未成功触发优雅关闭时, 宿主退出. 往往发生于底层系统异常.

对于意外宕机, 可能导致快照残损不可用.

总控可以感知到分控掉线, 这种情况下, 总控会标记一整套资源树不可用.

**3.12. 配置宿主**

目前的设计里, 宿主配置是静态的, 暂时不支持动态配置, 无法通过总控完成.

基于容器部署, 有很大概率会导致本地数据丢失, 且本期暂时没有接入配置中心, 因为暂不支持.

**3.13. 快照**

interface HouseKeeperSnapShot {

nodes: {

type

alias

config

}[]

}

**3.14. 一致性检查与维护**

一致性需要分控和总控协同实现.

这里讲的数据的一致性体现在总控元数据的资源森林和实际部署的分控资源树组成的森林是否一致, 即元数据的一致性.

资源树:

host -> node (-> instance)

对于每层, 一致性意味内部状态是否一致. 比如网关的路由, 容器的实例.

总控定期按批次检查各个分控资源树, 经过几轮数据交换即可实现最终一致.

原则是以元数据库为准, 对实际拓扑做增减.

一致性检查和维护会以独立的详细设计呈现.

**3.15. liveness 数据上报**

Liveness 数据包含了一个 host 资源树的概况, 用于订正数据面的流量转发规则, 以及调度算法的工作.

interface LivenessInfo {

}

**4. 通用节点层**



**4.1. 意义**

为以后各种 worker/agent 的不同实现和扩展种类打下基础.

**4.2. 标识信息**

通过三元组 (type, alias, pid) 可以在一个 host 内唯一索引一个节点进程.

**4.3. 对接分控**

任何一个节点都要连接分控控制服务, 以接受控制传递信息.

这里使用标准的命令 RPC 框架实现.

**4.4. 可观测服务**

可观测服务能够暴露一些数据, 用于巡检, 监控, 告警等用途.

参见末尾的 Observation 服务设计.

**4.5. 自定义节点的基类**

在 JS 实现里, 提供一个节点基类, 自定义的节点继承, 并实现必要以及可选的方法, 使用基类创建的对象.

**4.6. 优雅启停机制**

**4.6.1. 启动握手阶段**

* 节点进程启动
* 通过 RPC 框架以 IPC 模式连接到分控
* 发送 nodeReg 信号
* 启动 Snapshot 服务
* 启动 Observation 服务
* 执行自定义的初始化逻辑
* 发送 nodeRdy 信号

**4.6.2. 优雅停止**

* 执行自定义的关闭逻辑
* 关闭 Snapshot 服务
* 关闭 Observation 服务
* 断开会话

**4.7. 可获取的通用节点数据**

* Platform#env
* Platform#hostname
* Platform#ip
* Platform#pid
* Platform#args
* Platform#execArgs (Node.js 实现专属)
* Platform#envars
* Platform#idc
* Platform#cpuCore
* Platform#totalNativeMemory
* Platform#totalVmMemory
* limit
* Metrics#uptime()
* Metrics#activeHandles() (Node.js 实现专属)
* Metrics#activeRequest() (Node.js 实现专属)
* Metrics#gc() (Node.js/Deno 实现专属)
* Metrics#eventloop() (Node.js/Deno 实现专属)
* Resource#totalCpuTime()
* Resource#lastCpuTime()
* Resource#nativeHeapUsage()
* Resource#vmHeapUsage()
* Resource#fds()
* Resource#threads()
* io
* Profile#kernelStack()
* Profile#diagnositicReport() (Node.js 实现专属)
* Profile#vmHeapSnapshot()
* cpu profile, heap profile

**5. Gateway 节点设计**



**5.1. 初始化**

* 挂载路由表 (如果有)
* 启动 k8s Responder
* 启动全局 http agent
* 启动 http 服务器

**5.2. 可处理命令**

* loadRouteTable: 加载整张路由表
* loadRoutePatch: 加载路由补丁 (局部变更)

**5.3. pipeline**

每个 http 请求都经过相同的 pipeline 处理:

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

**5.3.1. 解析请求头**

这一步从 HTTP 头部取出链路追踪的 traceId, 设置 X-forward 为下游地址, 如果没有 traceId, 则生成一个.

将 traceid, x-forward 以及原始的 header 传递给下一步.

错误返回 502

**5.3.2. 选路**

选路模块根据 HTTP 协议的 host, path 部分, 选出指向的策略组 (属于某个特定应用),

错误返回 502

选路未匹配返回 404

**5.3.2. 策略执行**

选路得到的结果是一个策略组. 策略组里每个策略指向一组服务. 策略模块

根据服务组的标签选择匹配流量策略的服务, 得到对应实例组, 交给下一步.

错误返回 502

策略未匹配返回 404

**5.3.3. 负载均衡**

负载均衡模块按照给定的算法选择出一个实例三元组 (ip, port, instance-alias), 交个下一步

错误返回 502

**5.3.4. 请求预处理**

这是第一个可供动态配置的点位. 在这里可以局部 (针对一组路由) 或者整张路由表配置插件, 扩展网关的能力.

只允许操作请求头

错误返回 502

**5.3.5. 转发请求**

根据前几步处理后的请求头, 追加额外的 traceid, x-forward, x-instance-alias,

将请求转发到负载均衡阶段确定的 ip:port

逻辑错误返回 502

网络异常返回 503

**5.3.6. 取得响应**

提取响应头, 并把整个 response 对象传递个下一步

逻辑错误 502

网络异常 503

**5.3.7. 请求后处理**

第二个动态配置点位, 同上一个. 只允许操作响应头.

逻辑错误 502

**5.3.8. 回复响应**

将响应返回给客户端.

如果在 pipeline 的 io 过程中, 客户端终止了请求, 则终止 io.

**5.4. 选路模块**

**5.4.1. 路由表结构设计**

路由表通过 radix 树表达, 这里面每个节点是一个 path span. 每条 path 对应一个 endpoint 节点, 如果请求路径走到了这个 endpoint, 就表示有机会命中这个 endpoint 代表的路由.

每个节点还标注了被哪些域名共享, 以此检查是否匹配域名.

interface RadixTree<T> {

root: TreeNode<T>

depth: number

domainNameIndex: Hash

}

type TreeNode<T> = InnerNode | EndpointNode<T>

interface TreeSpanNode {

span: string

path: string

domainNameIndex: Hash

isEndpoint: boolean

isLeaf: boolean

}

interface InnerNode extends TreeSpanNode {

isEndpoint: false

isLeaf: false

}

interface EndpointNode<T> extends TreeSpanNode {

isEndpoint: true

data: T

}

**5.4.2. 加载路由表**

总控计算出原始的路由表, 经编码传输给 downloader 并据此生成一个运行时路由表结构 GatewayRouteTable. 然后借助 v8 的 serialization 直接序列化路由表到本地文件存储.

参见 downloader 节点部分.

loadRouteTable 命令指挥路由模块从本地文件加载路由表对象, 加载完成即可直接使用.

**5.4.3. 变更路由**

每次全量的路由表刷新成本过高. 当只有几个应用的路由发生变更时, 局部修改更合适.

loadRoutePatch 命令用于加载路由 diff.

interface RouteDiff {

add: RoutePatch

del: RoutePatch

}

interface RoutePatch {

[domainname]: path[]

}

Add 部分的路由, 依次增加到现有路由表里.

Del 部分的路由, 依次从路由表里移除.

**5.4.4. 选路算法**

* 拆分请求 path 为一个列表, 每个元素是一个 request span
* 维护一个指针, 初始为空
* 从根节点开始, 逐层 (广度优先) 检索 span 是否与 request span 匹配, 域名是否匹配 request domainname, 最后检查 isEndpoint 是否为真, 如果为真, 则指针指向该节点. 如果前两个检查未通过, 则结束匹配. 如果均匹配成功, 则继续向下一层子节点递归这个过程, 直到到达最后一层, 或者某一层匹配失败.
* 指针指向路由目标 target, 如果指针为空, 表示未命中.

**5.5. 策略模块**

**5.5.1. 策略表**

策略表是一个以策略类型为 key 的 hash.

Hash {

gray: List

bluegreen: List

sc: RadixTree

standard: List

}

Gray 和 bluegreen 对应的 value 是一个列表, 每个元素都是一个二元组 (config, service)

按照新旧排序.

Sc 对应一颗 radix 树, 每个 endpoint 对应一个 service

standard 对应一组 service, 按照新旧顺序排列.

**5.5.2. 策略执行**

* 按照 gray, bluegreen, sc, standard 的优先级, 检查策略组是否存在, 如果存在, 则进行策略匹配.
* gray, bluegreen 按照顺序依次检查每个策略项
* sc 最长前缀匹配
* 基准环境做兜底

**5.5.3. 策略算法**

无论是总控元数据还是策略表中, 维护的都是策略的分类和参数, 策略执行实际执行的是一段逻辑, 这里称之为策略算法. 策略算法也是模块化设计的, 每种策略对应一个具体的算法实现.

**5.6. 负载均衡模块**

负载均衡的状态无需持久化.

默认算法为 RR.

负载均衡算法能够动态配置.

**5.7. 流量处理插件**

目前流量处理插件只支持处理头部

请求预处理插件:

interface RequestPreProcessHandle {

(header: RequestHeader, ctx: PipelineCtx) => Transformed | Returned

}

响应预处理插件:

interface ResponsePreProcessHandle {

(header: ResponseHeader, ctx: PipelineCtx) => Transformed

}

**5.8. 监控点位设计**

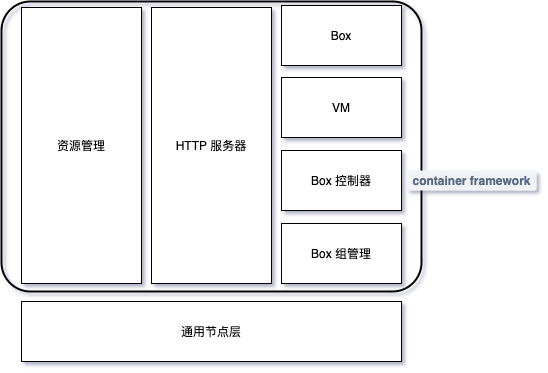
监控点位用于检查 pipeline 各阶段的耗时, 以便精准定位瓶颈, 做出合适的优化.

* 解析请求头: 解析耗时
* 选路: 选路耗时
* 策略执行: 策略执行耗时
* 负载均衡: 负载均衡算法耗时
* 请求预处理 (插件挂载点): 请求预处理耗时
* 转发请求 & 接收响应: 网络周转耗时
* 请求后处理 (插件挂载点): 响应预处理耗时
* 回复响应: 写出耗时
* 整个 pipeline 耗时: 从接收请求, 到写出响应.

**5.9. 快照**

网关节点快照就是路由表+策略表.

**6. Crate 节点设计**



**6.1. crateFramework**

crate 节点核心是管理 crate Framework.

里面维护了诸如资源管理, 服务器, box 控制器等实现实例托管的必要组件.

**6.3. box controller**

Box 仅仅是一个面向应用代码的规范. 落实到托管的实现, 可能是裸跑在 crate 节点里的, 也可能是封装在虚拟机提供的隔离机制里, 而这些虚拟机甚至运行在不同线程里 (包括内核级线程和用户级线程).

为了屏蔽这些托管技术的差异, box controller 被引入进来. 给宿主侧提供了统一的视图和接口, 这样一来, 宿主侧可以视其为一个黑盒, 无需关心实现细节.

因此从宿主视角, 需要管理的就是一个个 box controller, 就像 housekeeper 管理 node controller 一样.

在宿主侧, 这个 box controller 被 box group manager 管理, 维护了 instance alias 到 box controller 对象的映射.

interface BoxController {

vm: BoxVM

alias: string

appInfo: AppInfo

code: Buffer

codeType: "js-source-text" | "code-cache" | "snapshot"

boot()

stop()

kill()

dispose()

callSync()

callAsync()

getStatus()

getVMType()

getStats()

getMetaData()

bindResource()

ctl()

}

interface BoxGroupManager {

boxTable: Hash<string, BoxController>

createBoxController()

getBoxController()

deleteBoxController()

count()

}

**6.4. 实例加载**

loadInstance 命令触发实例加载. 可加载的实例代码有三种形态:

* js 源代码
* v8 startup Snapshot
* v8 code cache

对应三种 loader.

loadInstance 指示了隔离级别, 预设参数, 本地代码路径, 以及代码类型.

创建新的 box controller, 创建虚拟机, 调用 loader 初始化虚拟机.

等待启动完成.

**6.5. 实例卸载**

unloadInstance 命令触发实例销毁.

调用 box controller stop 方法, 请求优雅关闭 box.

完成后, 调用 dispose 销毁虚拟机, 并将 box controller 从 box group manager 里移除

**6.6. 实例健康检查**

对于节点 (宿主侧) 来说, 实例健康的条件是: box controller 存在, vm 是 running 状态, box 是 running 状态.

这个策略后续可能会有所调整.

**6.7. 流量转发链路**

* 检查请求头中 x-ae-instance-alias
* 找不到对应 box, 响应 404 (注意区分网关 404)
* 找到 box, 转发流量给 box controller 处理, 处理失败, 响应 500
* 经过 vm 层, 调用到内部 box 上的选路方法, 走应用内路由
* 响应数据经 vm, 给到 box controller
* 响应下游调用方

**6.8. baas Resource 模块**

供实例代码使用的二方服务客户端, 其生命周期统一由 Baas Resource 管理.

每个 Resource 是动态扩展的, 能够通过命令增加减少 Resource.

interface BaaSResourceManager {

map: Hash<id, BaaSResource>

createResource()

getResource()

deleteResource()

}

**6.8.1. resource id**

$resourceType:$resourceSpecificedId

Id 根据 Resource 的配置生成, 不同 Resource 有自己的 id 生成策略.

比如 mysql: 服务器地址(ip:port), kvds: 集群代理地址

Box 中 Baas 对象在申请 resource 时, 底层都是通过 id 来查找的, 只有不存在 id 时, 才会创建一个新的 BaaSResource

**6.8.2. BaaS**

BaaS 是 Box 内二方服务客户端接口.

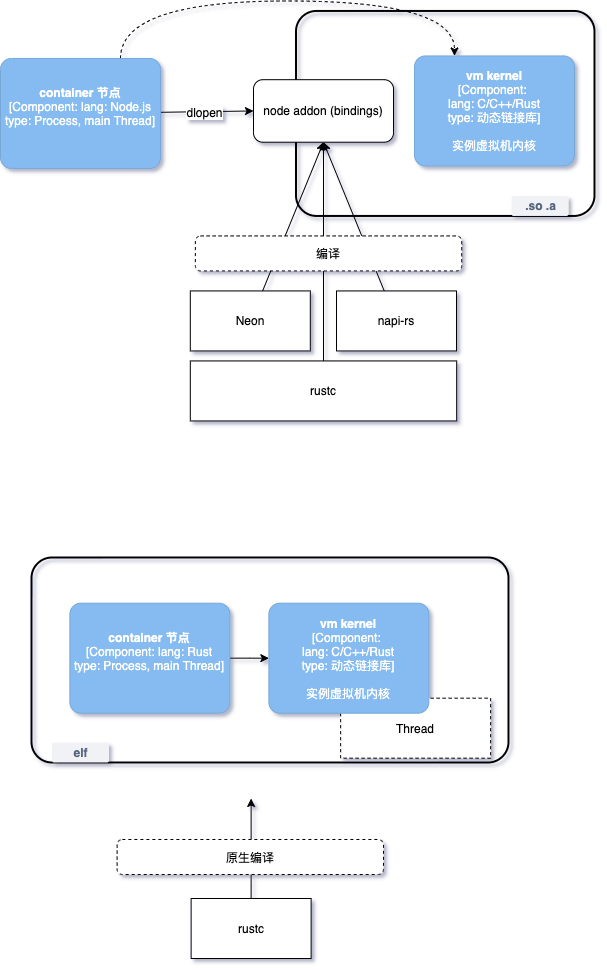
Box 用户对 BaaS 具体实现无感知. BaaS 可能来自底层 shared resource, 也可能是在 Box 内私有管理的 (当底层 resource 不提供时).

**6.9. 虚拟机设计**

目前时间原因, 只支持 level = 0 的隔离级别, 即 context 隔离, 在这一阶段, 仅仅隔离 box global.

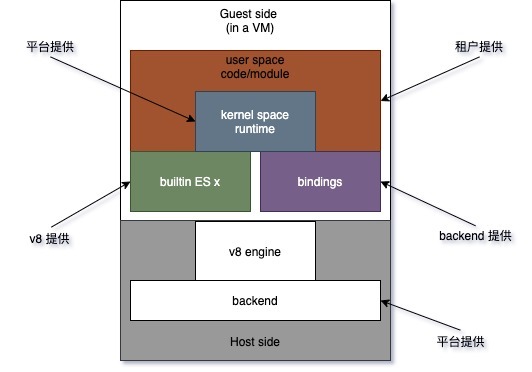
但 Box 内加载的模块都是在 main context 内加载编译的.

自建虚拟机原理:



* 目前 neon 在 n-api 覆盖上进度缓慢, 特别是没有支持 context aware module 的编译, 因此它编译过的 addon 是无法支持多线程的.
* napi-rs 声称覆盖了绝大多数 napi 版本, 实际测试表明, 编译的模块是支持 context aware 的.

**虚拟机架构**



如果提供不同的 JS runtime 支持, 需要平台支持

* 可定制的 kernel space runtime lib 的用户空间接口 (支持 Node, Deno 不同版本运行时)
* 可定制的 v8 引擎 (支持不同的 EcmaScript 标准)
* 可定制的 backend (支持不同的特权调用能力, Node/Deno 都需要)

在租户侧模拟真实的服务端 JS 运行时, 这三个属性应该是绑定的, 不能随意组合.

变化较多的应该尽可能集中在 kernel space runtime lib, 便于高效开发.

**内核空间运行时设计**

由于本期不上自建虚拟机, 因此, 运行时环境保持和 Node 14 一致.

**隔离机制**

process + v8 context

基于 context 的隔离思路如下:

host

---

inner vm (prepare)

---

box vm

const ctx = Object.create(null)

function compile(moduleCode) { return vm.runInContext(moduleCode, ctx) }

function compileScript(moduleScript) { return moduleScript.runInContext(ctx) }

function resolve(id) { }

function createWrappedCode(code) {

return `function (module, export, \_\_dirname, \_\_filename) { ;${code}; }`

}

ctx.require = function (id) {

if (passBy(id)) {

// unsafe zone

return require(id)

} else {

// 1. resolve to code

// or script

const moduleCode = createWrappedCode("xxx")

// 2. compile moduleCode or moduleScript

const moduleFn = compile(moduleCode)

// create new module object inside same vm context

const moduleArg1 = compile("{ exports: {} }")

const exportsArg2 = moduleArg1.exports

const \_\_dirnameArg3 = "xx"

const \_\_filenameArg4 = "xx"

// 3. run module fn

moduleFn(moduleArg1, exportArg2, \_\_dirnameArg3, \_\_filenameArg4)

// 4. return module object

return moduleArg1

}

}

// 设置其他 global 对象

ctx.setTimeout = function setTimeout(...args) {

return setTimeout(...args)

}

// 加载虚拟机代码, 传入 重写的 require 上下文

const vmCode = fs.readFileSync("./js/vm-in-vm.js", "utf8")

const vmFn = vm.runInNewContext(createWrappedCode(vmCode), ctx)

const vmFnArgs1 = compile("{ exports: {} }")

// 开始初始化虚拟机

const bootGuestSystem = vmFn(vmFnArgs1, vmFnArgs1.exports, "/", "vm entrance")

// 加载租户代码

bootGuestSystem("./xxx")

可隔离的对象:

* JS 全局对象和方法访问

**资源配置**

无

**特权访问控制**

收拢管控模块与全局方法的访问:

* overwrite: require, module, fs, net, http, process, console, dns, os, path, timers, tls/ssl, tty, udp
* limited: fs, dns, net, http, process, os, tls/ssl, tty, udp, require, module
* disabled: child process, cluster, v8, vm, wasi, worker\_thread, async\_hooks, domain, http2, inspector, esm, performance\_hooks, policies, report, trace\_events

沙箱化文件系统目录结构访问:

* \_\_dirname: /
* \_\_filename:
* process.cwd(): /
* path 结果映射到沙箱内
* require 只限于特定目录
* fs 只限于访问特定目录

**快照**

crate 节点的快照是实例的元数据, 外加代码.

**7. Downloader 节点设计 (另起文档)**

应用代码包

压缩的路由表

依赖

下载后处理

配置

命令

空闲资源回收

**8. NetworkPartitionGuard 节点设计**

如何判断一个服务是否可用? 我们可以分成三部分来检查:

* 网关路由表是否包含这个服务的指向
* 网关到服务所在的主机之间网络是否可达
* 服务自身是否存活可响应

该节点的工作在第二层, 协助网关检查目标实例所在的 crate 主机是否可达.

方式很简单, 从应用层 (HTTP) 发起一个 HEAD 请求到目标 crate 主机的 NetworkPartitionGuard 节点.

**8.1. 判定规则**

M 次检查失败 (传输层及更下层错误, 超时) 后, 认为网络不通. 一旦再次检查通过, 则标记连通.

**8.2. 获取检查目标**

检查结果应用于路由表计算源头, 因此也是总控负责触发检查, 提供检查目标.

Target [x,y,z,...]

**9. Logger 节点设计 (另起文档)**

监视日志存放目录

采集模块

转换模块

上报模块

清理模块

快照内容

配置

命令

空闲资源回收

**10. BaaS 节点设计 (低优先级, 暂不考虑)**

初始化 client

exposed 接口

审计控制

资源副本

连接池

client 优雅开闭

超时

配置

快照

命令

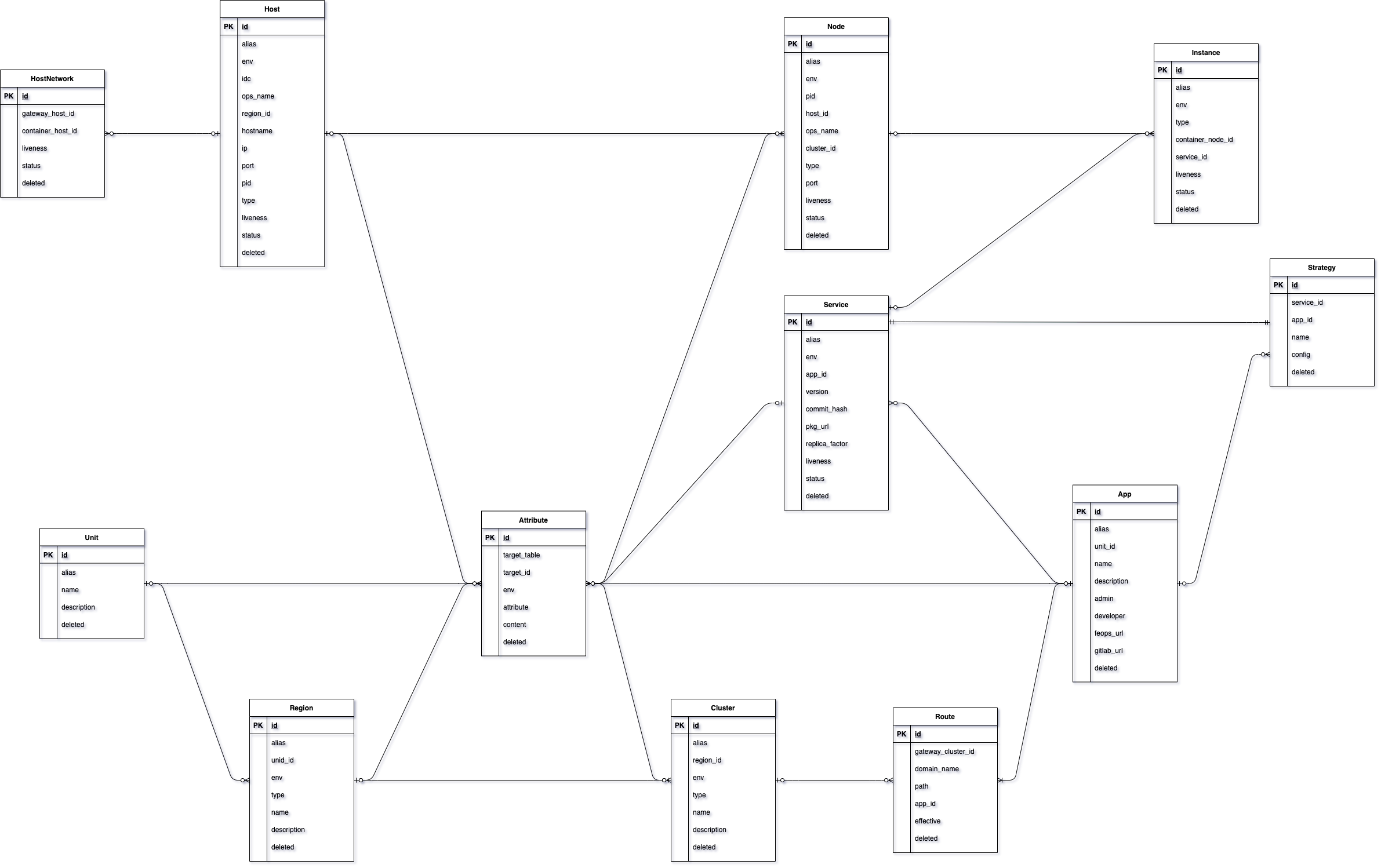
空闲资源回收

**11. DispatchCenter (调度中心/总控) 设计**

**11.1. 模块分层**



**11.2. 元数据数据库表设计**



note: 考虑到结构的略微差异和数据量, 将 attribute 表按照各个主表进行拆分. app 表增加 env

**11.2. 受理宿主登记请求**

入参: host register data

* 尝试建立会话
* 握手
* 协商心跳周期
* 等待分控发送命令检查快照
* 如果快照有问题, 则生成初始化向导下发
* dc 集群广播宿主路由变更
* 写入登记信息到数据库

出参: 受理节点地址 + 结果

**11.3. 处理宿主注销动作**

入参: host alias

* 标记数据库里宿主资源树 liveness=1

**11.4. 创建服务(实例组)**

入参: app alias, meta data, vm\_requirements

* 可调度性计算, 产出分配方案
* 编排命令配置, 下发给对应分控, 等待完成
* 更新数据库

**11.5. 关闭服务**

入参: service alias

* 找出对应主机分控, 下发命令
* 更新数据库

**11.7. 创建路由项**

入参: domain name, path, [gateway cluster id, app\_id, effective]

* 数据库新增一条路由记录

**11.6. 配置路由项**

入参: 路由 id, 网关集群 id, app id

* 找到路由项, 更新 网关集群和应用的 id, effective = 0
* 路由下发

**11.7. 配置策略**

入参: service\_id, 名称, 配置

* 更新策略表
* 策略下发

**11.8. 解绑路由**

入参: 路由 id

* 更新路由项 effective=1

**11.9. 销毁服务**

入参: service\_id

* 删除 service 关联项
* 如果关联过策略, 触发策略下发
* 找到分控, 下发关闭指令

**11.10. 创建实例**

入参: service id, factor

* 可调度性计算, 得出分配方案
* 编排命令, 下发给分控
* 更新数据库
* 检查 service 是否关联了策略, 如果已关联, 下发策略

**11.11. 关闭实例**

入参: instance id

* 删除 instance
* 如果 service 关联了策略, 触发策略下发
* 下发关闭指令

**11.12. 调度**

一次调度计算的产出结果是 不可调度 或者 分配方案.

我们已经在整体设计中花了不些许篇幅讨论 **可调度性**, 这里展开说明一下.

应用实例有自己的最小资源消耗, 我们要满足这个需求, 因此, 可调度的最基本需求是: 能找到足够的资源分给这个应用.

而后, 有的应用需要降低安全性来获得更高的性能, 有些应用为了确保不被不安全的代码损坏, 需要沙箱环境运行. 应用之间还存在不同事业群/业务域的概念, 因此他们希望各自使用彼此不想干的资源池. 还有更多各种各样的规则.

另一方面, 应用引擎知道的 “物理” 资源是连接到它的主机群. 为了便于实现可控的细粒度的调度, 我们将这些物理资源按照不同维度做了逻辑上的分割, 即: 单元, 环境, 区域, 集群.

接下来以放置一个服务为例阐述调度算法核心, 讲一下这些概念是如何影响调度决策的.

输入: 环境, 应用id, 预期的副本数

**11.12.1. 收敛资源池**

* 根据应用id, 找到 归属单元, 应用的预设属性 (平均内存消耗, 需要的隔离级别, 期望使用的区域和集群等需求)
* 匹配单元. 查找单元的预设属性, 检查是否满足应用的需求.
* 根据单元和预期的区域, 查找是否存在可使用的区域, 并做区域匹配, 参见上一步.
* 根据区域和预设的集群, 查找是否有可用的集群, 做集群匹配, 参见上一步.
* 最终确定的资源池可能是一个区域或者集群, 取决于应用的预设参数.

**11.12.2. 模拟演算**

* (node 归属的) host 数量是否 >= 预期的副本数
* 单位归属机房是否 >= 2 (取决于应用是否有预设属性)
* 将单位按照可用空闲资源大小, cpu 空闲度, 逻辑上可放置实例数, 负载, 延迟等因素排序
* 依次筛选全部匹配应用预置参数的单位
* 如果能够满足放置预期的副本数, 则表示可调度.

**11.12.3. 尝试调度**

* 根据上一步的分配方案, 编排为命令序列, 下发给分控
* 具体到分控检查实际的空闲资源以及阈值是否满足调度需求
* 放置实例

**11.13. 资源重新分配**

需要重新调度的场景主要发生在:

* 调度算法失精, 累计的误差出现显著的负载倾斜
* 底层容器重调度, 导致部分 host 本地状态丢失

第一种情况, 问题在于整个容器型资源池分配出现不均衡, 需要做 **全局** 的重排, 本质上是资源规划和实例的迁移.

**11.13.1. 资源规划**

**11.13.2. 实例迁移**

**11.13.3. 状态监测 & 复制 host 资源树**

第二种情况会比较频繁, 随时都有可能发生. 解决方案也同样分成两部分: 确定主机异常关闭, 扫描资源树, 选择一个空白的相同型号宿主, 复制资源树.

主机异常只能以会话终止形式体现. 但自身异常导致的底层系统重启, 并不会改变 host alias, 本地状态依然存在. 只有k8s 重新调度时, 才会在原宿主关闭同时有一个新的 host 接入总控.

总控一旦确定主机 **丢失状态**, 就会尝试寻找一个 **符合条件** 的空白 host 来复原资源树. 如果找不到这样的 host, 则放弃.

如何确定丢失状态? 只要会话中断的主机不是授权下线, 并且下线后在 M 时间内没有复活, 那么认为这个 host 已经丢失状态了.

如何定义符合条件? 需要在相同的区域里寻找, 能够满足资源足够容纳原有资源树的 host, 同样各种预设属于也需要匹配.

**11.14. 数据平面流量拓扑**

数据平面的 HTTP 请求选路被设计成两部分:

* 根据域名 (domain-name) 和路径 (path) 定位应用. 因为一个域名路径组合只能唯一确定一个应用 (命名空间的概念).
* 根据策略在确定的应用 (命名空间) 里选择服务.

这里面有个先决条件:

* 应用绑定至少一条路由项, 且路由项 effective=0
* 应用下至少有一个服务
* 服务必须绑定了策略

因此数据面流量拓扑结构为: 路由 -> 应用 -> 策略表 -> 服务 -> 实例

**11.14.1. 生成原始路由信息**

总控按照如下流程计算出某个特定网关集群的原始路由表:

* 从 route 表查询给定 gateway\_cluster\_id 的路由信息: domain-name, path, app\_id
* 对于每条路由的 app\_id, 从 service 表筛选出对应的服务
* 对于每个服务的 id, 从 instance 表筛选出匹配的实例, 并转换至每个实例所在的节点ip 地址和端口, 以及实例的 alias
* 对于每个服务的 id, 从 strategy 表筛选出对应的规则 (name, config), 与上一步的数据绑定

至此, 一份原始路由表构建完成. 表达的信息如下:

# 原始的单条路由项, 以行的形式描述为:

domain-name, path, service-alias (instance-alias,ip,port)[], strategy-name, strategy-config

**11.14.2. 数据压缩传输**

原始路由信息可能会存在大量的重复信息, 如果未经处理直接下发, 可能会造成不必要的 IO 负担.

总控对获取后的数据做初步的处理, 为了简单阐明原理, 我们举个例子:

对于原始路由项中的首部, 会存在大量的冗余, 因为存在很多应用共享同一个域名. 我们采用扁平化策略, 提取所有 domain-name 做去重, 作为单独的一块数据, 原有路由项中只存储这个数据块中的 domain-name 索引.

原始数据:

a.com, /abc/aaa, service-001, [...], none, none

a.com, /abc/aaa, service-002, [...], sc, prj1024.debug.a

a.com, /abc/aaa, service-003, [...], gray, percentage:10%

a.com, /abc/bcd, service-004, [...], none, none

a.com, /bcd/aaa, service-005, [...], none, none

压缩 domain-name 后 (可视化表示, 非实际布局):

[domain-name]

[a.com](http://a.com)

[route]

0, /abc/aaa, service-001, [...], none, none

0, /abc/aaa, service-002, [...], dynamic-link, prj1024.debug.a

0, /abc/aaa, service-003, [...], gray, percentage:10%

0, /abc/bcd, service-004, [...], none, none

0, /bcd/aaa, service-005, [...], none, none

诸如此类, 像 service alias 没有什么用处, 可以直接剔除. 大多数情况下 strategy name 是不存在的, 此时 strategy config 同样不存在, 也可以直接忽略掉.

再把 strategy name, strategy config 提取为 block. 路由替换为索引.

注意每一个路由都包含一个实例列表, 我们可以用另一个 block 表示所有的实例列表, 当然, 也可以应用上述优化策略, 注意这里 instance 无法做压缩, 我们保留到原始表中:

[ip]

10.0.0.1

172.16.0.4

10.0.0.2

10.0.0.3

10.2.17.8

[port]

15728

65532

3306

4298

9570

[instance]

instance-001, 0, 0

instance-002, 0, 3

instance-003, 4, 1

...

现在看一下经过压缩后的 (可视化) 格式:

[domain-name]

[a.com](http://a.com)

[ip]

10.0.0.1

172.16.0.4

10.0.0.2

10.0.0.3

10.2.17.8

[port]

15728

65532

3306

4298

9570

[instance]

instance-001, 0, 0

instance-002, 0, 3

instance-003, 4, 1

[strategy name]

gray

dynamic-link

[strategy config]

percentage:10%

prj1024.debug.a

[route]

domain-name:0, /abc/aaa, instance:[0, 2, 4]

0, /abc/aaa, [1,2,3], 1, 1

0, /abc/aaa, [2,5], 0, 0

0, /abc/bcd, [1,6]

0, /bcd/aaa, [3,1,5]

最后还剩下路径部分没有压缩.

为了构建 path 检索表,

我们先把所有的 path 合并成一颗前缀树,

然后从根开始, 广度优先遍历.

对于每个节点, 检查当前 span 是否存在于 path block 中, 如果存在, 则使用 path block 索引替换, 再填写当前编码后的路径到 path block, 如果前缀已经存在于 path block, 则新的路径使用引用来替代, 如 /a/b/c/d, 前缀 /a/b/c 在 path block 的索引为 5, 则 /a/b/c/d 在 path block 中表示为 $5/d, 以此类推.

最终我们得到的 (可视化后的) 数据结构布局如下:

[domain-name]

[a.com](http://a.com)

[ip]

10.0.0.1

172.16.0.4

10.0.0.2

10.0.0.3

10.2.17.8

[port]

15728

65532

3306

4298

9570

[instance]

instance-001, 0, 0

instance-002, 0, 3

instance-003, 4, 1

[strategy name]

gray

dynamic-link

[strategy config]

percentage:10%

prj1024.debug.a

[path]

0: abc

1: $0

2: bcd

3: $1

4: aaa

5: $0/$4

6: $0/$2

7: $2/$4

[route]

domain-name:0, /abc/aaa, instance:[0, 2, 4]

0, 1, [1,2,3], 1, 1

0, 1, [2,5], 0, 0

0, 6, [1,6]

0, 7, [3,1,5]

这种压缩策略生效的条件时原始路由信息存在重复, 如果没有任何重复段落, 或者整体数据量过小, 则会导致体积膨胀. 数据重复区域长度越长, 重复度越高, 压缩比越高.

**11.5. 命令**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| command | target | args | ok relply | err reply |
|  |  |  |  |  |

**11.6. Open HTTP API**

总控 API 分为三部分维护:

* 开放 API (用于外部服务/产品的调用)
* 内部 API (底层架构间的特殊消息的 HTTP 形式通信)
* 运维 API (系统监控运维使用, 暴露更细操控粒度, 更多可用概念)

内部 API 和运维 API 这里暂时没有稳定下来, 因此暂不展示.

开放使用的 HTTP API 遵循固定的响应格式标准:

interface HTTPOkResponse<T> {

code: 0

data: T

}

interface HTTPErrResponse<ErrMsg extends string> {

code: 1

data: ErrMsg

}

type HTTPResp = HTTPOkResponse | HTTPErrResponse

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| desc | method | path | query | request | ok data | err data |
| 创建路由 | POST | /route |  | clusterId,  domainName,  path,    (effective, appId) | routeId |  |
| 配置路由 | PUT | /route |  | routeId,    (clusterId, effective, appId) | ack |  |
| 创建策略 | POST | /strategy |  | serviceId,  appId,  name,  config, | strategyId |  |
| 移除策略 | DELETE | /strategy |  | strategyId | ack |  |
| 分配应用运行时实例组 | POST | /service/allocate |  | appId,  env,  version,  commitHash,  pkgURL,  replicaFactor,  (attribute), | ack |  |
| 创建集群 | POST | /cluster |  | regionId,  name,  description,  (attribute) | clusterAlias |  |
|  | PUT | /cluster |  | clusterAlias    (name,  description,  attribute) | ack |  |
| 创建区域 | POST | /region |  | unitId,  env,  type,  name,  description,  (attribute) | regionAlias |  |
|  | PUT | /region |  | regionAlias,    (name,  description,  attribute) | ack |  |
| 创建单元 | POST | /unit |  | name  description  (attribute) | unitAlias |  |
|  | PUT | /unit |  | unitAlias    (name,  description,  attribute) | ack |  |
| 更新宿主资源树的状态 | POST | /task/updateLiveness |  | (unitAlias,  env,  regionAlias,  clusterAlias,  hostAlias,  ) | ack |  |
| 下发路由 | POST | /task/dispatchRoute |  | (unitAlias,  env,  regionAlias,  clusterAlias,  hostAlias,  nodeAlias,  ) | ack |  |
| 检查网关到容器网络可达性 | POST | /task/checkNetworkReachable |  | (unitAlias,  env,  regionAlias,  hostAlias,  ) | ack |  |

**12. 其他**

**资源池伸缩**

目前不考虑缩, 扩展直接操作 ops 扩容.

**定时任务实现周期性控制**

暂时使用 TSP REST 定时任务, 调用总控暴露的 HTTP 接口, 参见 11.6. Open HTTP API