AREI- immutable

– 不可变应用环境的编排和AIOW运维范式@IaaS

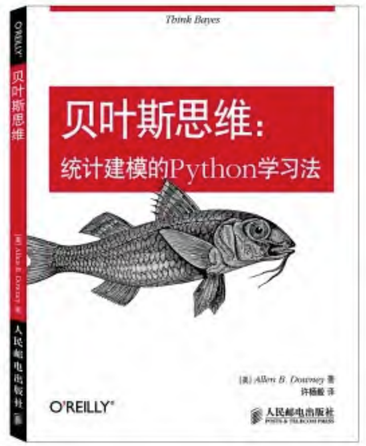
许杨毅 UCloud应用运维部

2016年03月25日

Outline

- IaaS公有云的运维特点-两个 *i*
- ARE是什么
- 实现方法
- 运维的AIOW模型

All about me



- 2000年加入百度，BAT第一代工程师，百度系统部No.2
- 2010年新浪，经历了新浪博客，微博业务的发展过程。
- 2015年加入UCloud，负责应用运维工作，建立应用运维体系
- 数据控，喜欢数学和应用建模，日常研究复杂网络和复杂系统，痴迷网络动力学,爱好阅读，偶尔写书。
- 《贝叶斯思维》《DataJustRight》《DoingMath inPython》中译

IaaS运维的两个 *i* 特点

Infrastructure

- 基础架构（SDN/NFV网络，虚拟机over物理机）
- 多租户环境
- 故障的乘数效应 Fault= f *Sum(*Tenant*)
- 稳定性的指数效应 Stablility = $\prod_{i=1}^{N} S$

Immutable 不可变

- 变更是现网故障的重要原因
- IaaS的稳定来自运行环境的确定性（网络变化，包升级，人为失误）
- “不可变”是一个工程目标，而工程化的基础 → “确定性+稳健性+不可变”

ARE - *A*pplication *R*unning *E*nvironment

基于传统的SCM，我们扩展出了ARE这个概念． *SCM (software configuration mgm)*
并非标新立异，而是为了更好的理解和处理IaaS下的具体问题

背景： UCloud 的每一种IaaS服务，都对应多个ARE，软件版本/产品类型/部署环境/定制化需求 一起构成了不同的应用运行环境。

通常观点：

A. ARE是不可变immutable，除非在以下业务场景下：
   应用环境部署/代码升级/在线变更/回滚/漏洞修复
B. 传统实现方法：
   - 以配置管理的视角看待ARE环境 （OS版本-软件框架-环境变量-配置项-依赖包-依赖文件）
   - 配置结构--- 树状可继承 （最差全矩阵-笛卡尔冂，最优可继承树）
   - 配置内容--- 预先定义，可结构化 （xml,yAML）
   - 配置管理 --- 代码化（Git进行管理）

ARE - *A*pplication *R*unning *E*nvironment

基于传统的SCM，我们扩展出了ARE这个概念． *SCM (software configuration mgm)*
并非标新立异，而是为了更好的理解和处理IaaS下的具体问题

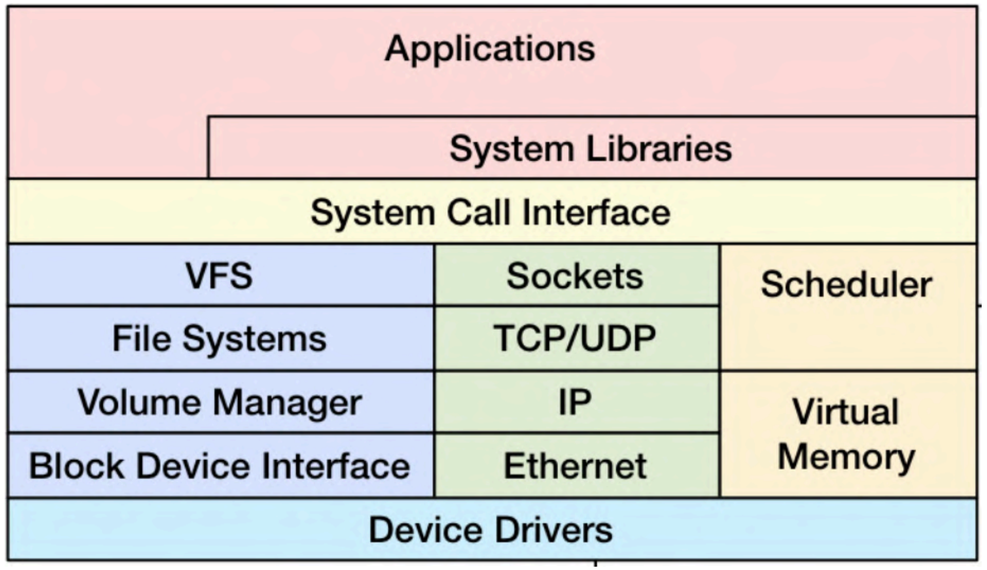
I. 形式上：ARE是一个可分层表示的系统变量的对象栈（OS）

II. 技术上：ARE是一个复合对象的集合

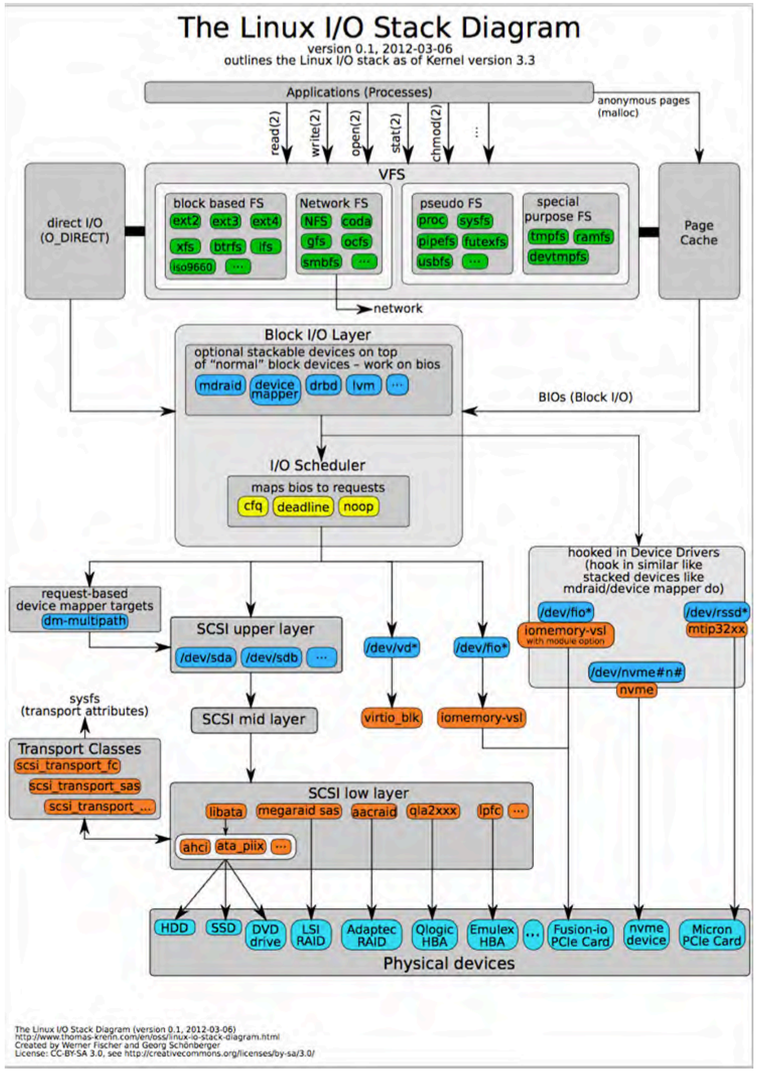
- ARE静态集：{os，os_ver，kernel，gilbc，Python，PM2，JVM，编程框架}
- ARE动态集：{硬件的固件，驱动，内核动态加载的各种***.ko，crontab，daemon，route table，flowTable@SDN，tunnel@SDN，***.so，应用代码 }
- ARE分布式环境：{网络拓扑，服务模块，带宽特征，业务流量特征}

ARE栈 --- 致敬

以栈的形式理解ARE，向启发了我们灵感的专家， Salute !
& CopyRight



Brendan Gregg

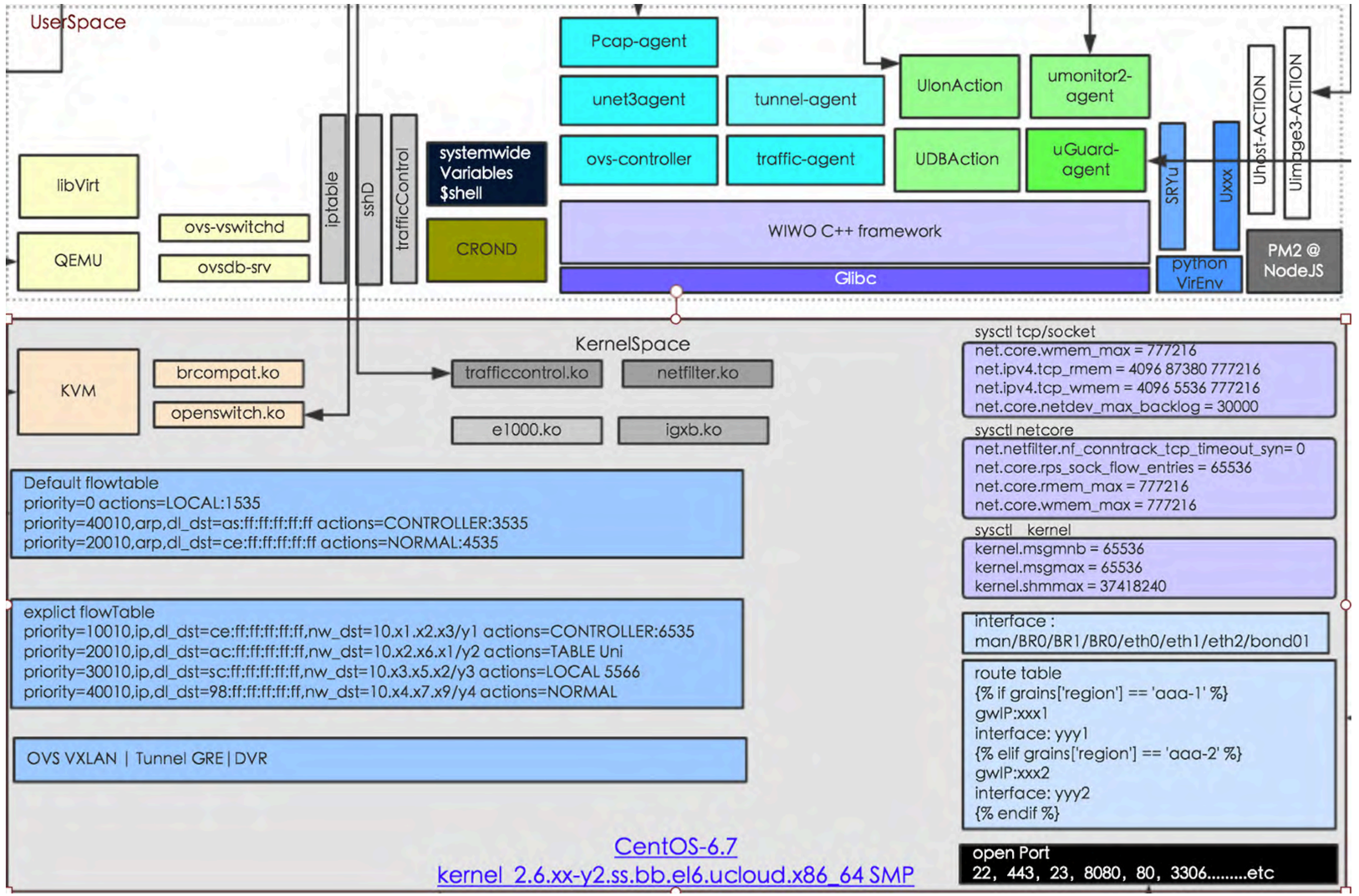


http://www.thomas-krenn.com/en/oss/linux-io-stack-diagram.html
Created by Werner Fischer and Georg Schönberger

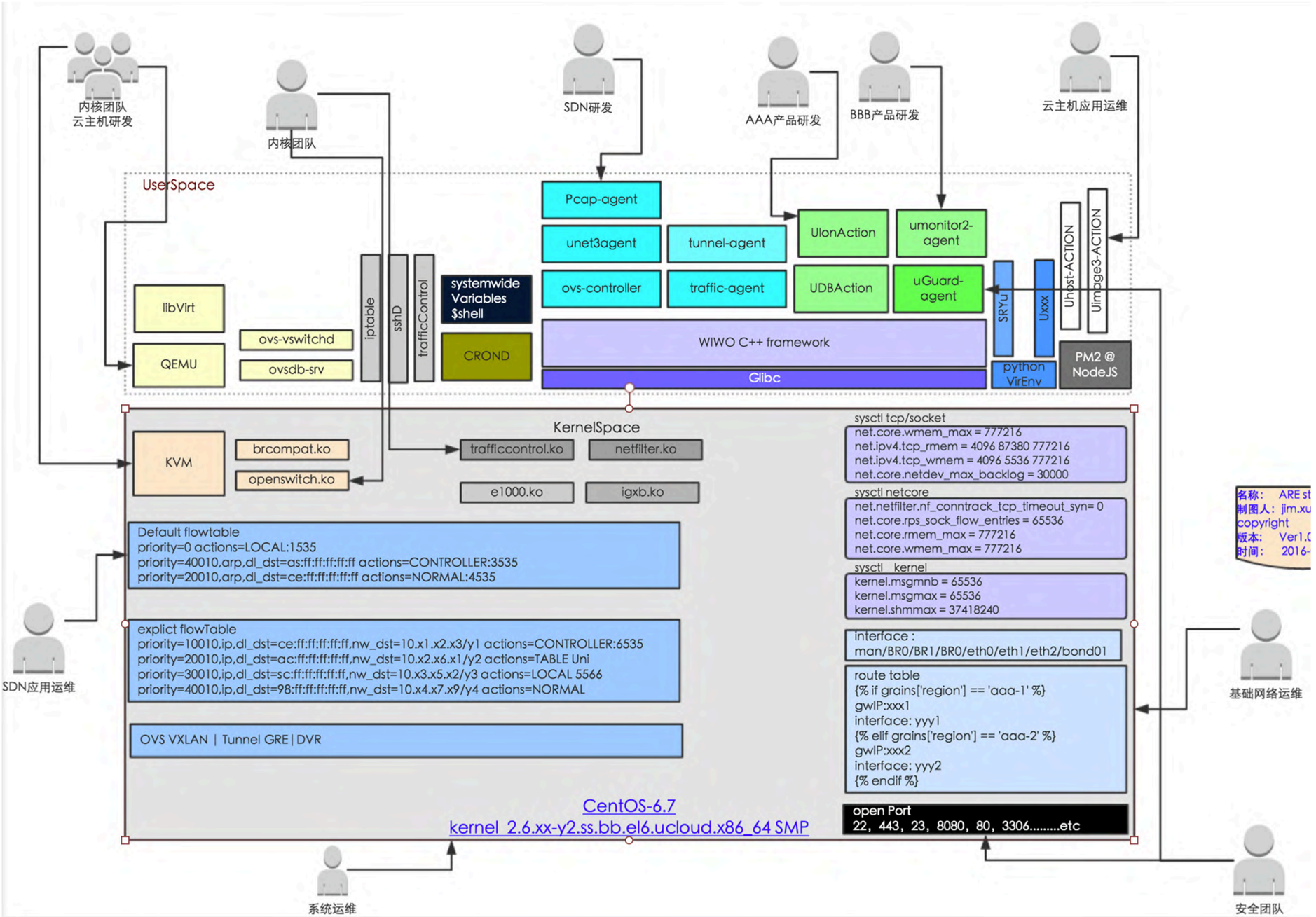
ARE - *A*pplication *R*unning *E*nvironment

以栈的形式理解应用的运行环境

- OS（kernel + 发行版 + packages）
- 内核运行态参数（sysctl –a ）
- 按需加载的众多内核模块***.ko
- 解析型语言环境（pm2&python）
- 软件运行框架（WIWO framework）
- CronD
- Running Deamon
- 路由表
- Open TCP/UDP Port
- OVS ko/userspace utils
- SDN flowtable
- SDN tunnel list
- Systemwide Variables
- Process Variables



ARE - Application Running Environment



不同团队,不同关注点

- 系统运维
- 基础网络运维
- 内核团队
- 安全团队
- 应用运维A组
- 产品研发团队
- 应用运维B组

形成的实际局面就是每个团队都从ARE的不同切面进入

ARE - *A*pplication *R*unning *E*nvironment

产品研发团队 - *“请保证产品研发的进度”*

- ***快速迭代*** --- 交付产品特性，代码发布/回滚 --- 因此ARE的变化是常态
- ***管理灵活*** --- 产品特性需要灰度实现，因此ARE要能被方便的修改，在多维组合上，具备细粒度控制能力 (即灰度能力，这导致ARE是一个配置项多维矩阵)

  例如（region + IDC + 产品 + customerID + Time ）

挑战 1-- 协作团队的目标天然不一致

运维团队 - *“请follow运维的规范”*

- ***配置统一*** --- 配置是统一并且是一致的，最好全局范围？最好是树状可继承结构?
- ***不可变immutable*** --- 最好上线即不可变更，变更是主要的现网故障原因，血泪教训
- ***业务稳定*** --- 运维对应用的稳定性负责，天经地义

*immutable*只是运维的一厢情愿？！

Infrastructure immutable 源自Google SRE

- 前提条件：顶尖工程师，强大的运维团队，稳健的业务架构，优秀的软件耦合设计
- Docker包装好了软件运行依赖的一切环境，容器化银弹来了
- 良好的软件设计和开发质量，完善的测试工作流保障，稳定可靠的运行环境，成熟的CI/CD过程依然是软件应用稳定的保证。

ARE - *A*pplication *R*unning *E*nvironment

其他挑战

- ARE在形式上是分层栈式的 --- Hierarchy Stack
- 参与团队的多样化 & 仅对局部负责，导致了无法找到权威的仲裁者，如何定义一个具体ARE对象，因此传统的软件配置管理（SCM）方法受阻。
- 从配置管理的视角，假设最理想的情况下，ARE是一颗可继承的树状结构，所有的配置项都有明确的依赖关系和邻居关系，但是。。。。实际上?

  eg : 有一个ARE实例 = {os ,os_ver, kernel_ver,open, framework, application.so}，其组合的空间笛卡尔积= M * N * O * P，是一个多维矩阵。 经典兼容性测试的难题，变量越多，空间越大。

ARE的笛卡尔积空间 = $\begin{pmatrix} OS\ type \\ CentOS \\ RHEL \\ Ubuntu \end{pmatrix} \begin{pmatrix} release\ ver \\ 6.3 \\ 6.5 \\ 7.0 \\ 4.2 \\ 5.0 \end{pmatrix} \begin{pmatrix} kernel\ ver \\ 2.6.32.x \\ 4.1.20.x \\ 4.4.6.x \end{pmatrix} \begin{pmatrix} libVirt \\ 0.9.12 \\ 1.2.1 \\ 1.2.9 \\ 1.1.2 \\ 0.10.2 \end{pmatrix} \begin{pmatrix} Qemu/KVM \\ 1.1.2 \\ 1.7.0 \\ 2.1.0 \\ 4.2 \\ 5.0 \end{pmatrix} \begin{pmatrix} OpenSSL \\ \\ 1.0.1m \\ 1.0.0r \\ 0.9.8zf \\ 1.0.2a \\ \\ \end{pmatrix}$ =6000

对此，我们找到的方法→

ARE - *A*pplication *R*unning *E*nvironment

思想：

- 分而治之 Divide and Conquer --- 各自负责治理和定义ARE的细分项
- 民主协商 Democracy --- 出现配置项内容冲突，协同处理
- 面向切面 Aspect Oriented的ARE配置管理 --- 放弃本位思维，体现了多业务方视角和利益 。

收益：

- 不再寻求***预先定义***，不再自上而下的定义一个ARE对象，例如：力图将某个产品全局统一为一个配置版本（OS+Kernel+ovs+uhost）
- ARE对象从全局一致 → ***局部一致***
- 可***自助管理*** 各个部门对负责的ARE子集进行管理：{定义配置，提交配置，检查配置分布，提交ARE的变更申请 }

ARE - *A*pplication *R*unning *E*nvironment

方法：

1. 设计ARE schema：eg :{ ARE集合中的所有对象；组合形式 }
2. 自助定义某个具体的ARE pattern：可以自助设定一个具体的ARE pattern,并且保存，从而成为一个具体的，传统概念上的软件配置对象。

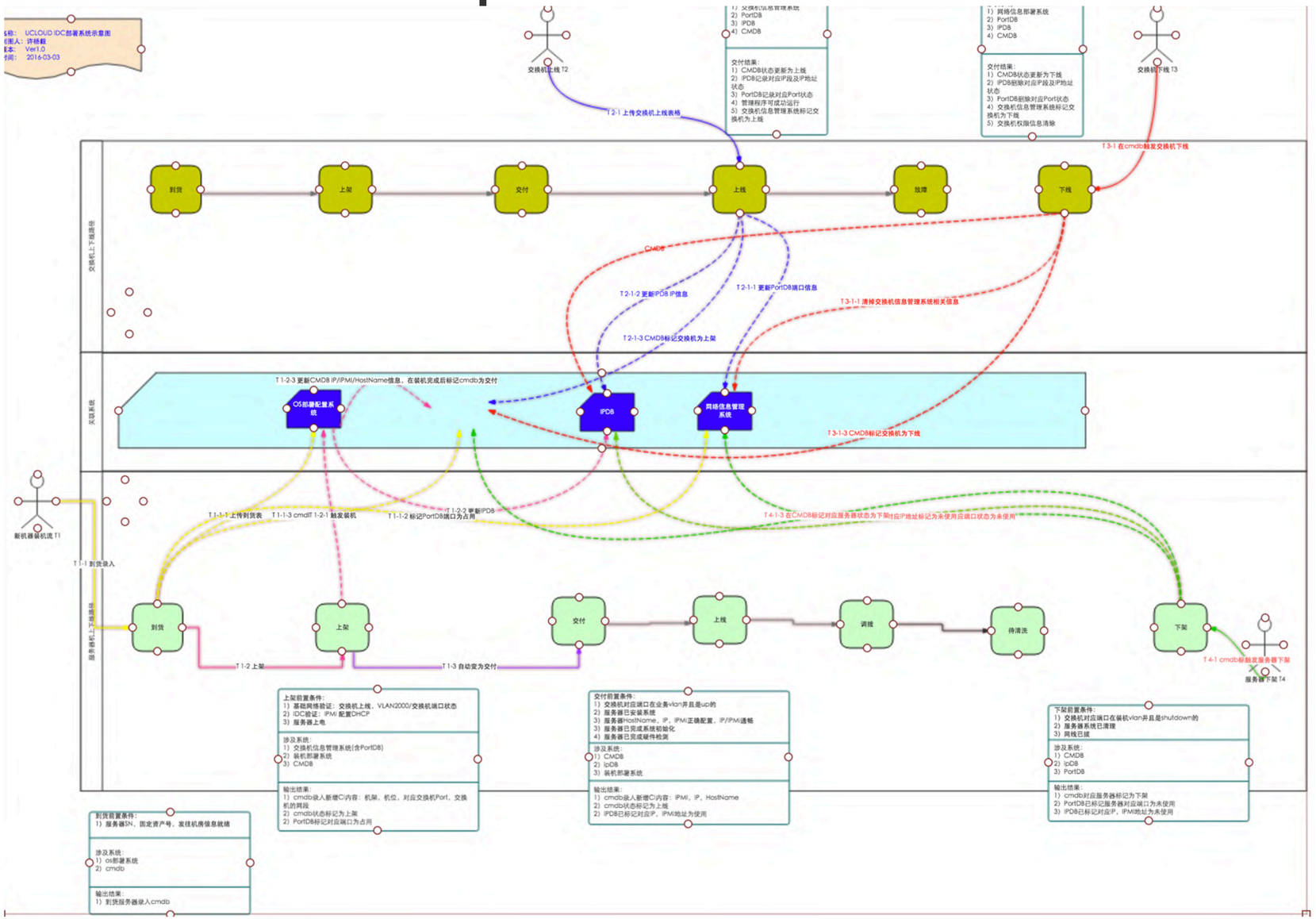
   eg: {os: “centOS”,kernel: “2.6.XX”,openSSL_ver:{1.0.2a, 1.0.1m, 1.0.0r ,0.9.8zf} } #需要进行openSSL修复的CVE-2016-0800 https://www.openssl.org/news/secadv/20160301.txt

3. 利用ARE pattern的查询结果，获得targeting，一组符合pattern的宿主机列表，集合X
4. 通过UCloud统一作业系统，对targeting集合X进行操作，修改ARE。

ARE编排 - Landscape

ARE 成为IaaS环境下编排的重要活动之一，以ARE为操作对象

- 业务环境部署：initial release
- 业务变更：ARE的局部更新，ARE数据结构的修改
- 代码上线/回滚：ARE对象的新版本释放，旧版本恢复
- 漏洞修复： OpenSSL，Qemu/libVirt热补丁



AIOW - *A*ction *In* & *Out* + *W*etail

接下来，我们谈谈云计算运维工作的模型和操作规范，因为：

- IaaS 运维的难度更高
- 失误/故障影响范围更大
- 事务数量也更多

云计算公司就是个整个互联网行业的运维团队，毫无疑问！

怎么解决这些问题？

杜绝流程缺失，最小化人为失误可能，云计算的6sigma 能不能实现？

AIOW模型 – 通过模型来抽象操作

A - Action
I - Input 输入量/前置条件
O - Output输出量/验证条件
W - who/where/what/how细节

C.2 部分信息汇总如下：

ITEM 项目	INPUT 前置条件	Action 模型				Output 交付条件、验证内容
		Who	Where	What	How	
nodeJS运行环境	1. registry.ucloud.cn 2. cnpmjs-ucloud.qiniudn.com 可正常解析	林生	目标管理MGRVM	安装 nodeJS 运行环境 Nodejs/pm2/node-tester	1.NPM="/usr/bin/npm --registry=http://registry.ucloud.cn --coche=${HOME}/.npm/.coche/lnpm --userconfig=${HOME}/.lnpmrc" 2.yum install --nogpgcheck -y nodejs npm telnet lsof 3.$NPM install pm2@0.12.10 -g 4.$NPM install node-fromework@0.0.4 5.$NPM install node-tester	人工交叉验证 验证程序
				7) 编写 start_server.sh		

uAppo运维规范 – 通过口诀来实现规范

P原则 --- 优先

- 故障处理优先级： 通知相关方--->恢复业务 --->保存现场 ---> 定位rootcause
- 运维日常操作优先级： 既定流程优先 ---> 请示工作导师 ---> 请示上级

B原则 --- 备份一切

- 操作前备份，现场保留备份

C原则 --- 控制原则

- 运维操作范围可控
- 运维操作结果可控
- 运维操作内容可控

T原则 --- 时间是任何操作要素

- 任何运维事务，都有时间属性
- 如果有标准SLA,按照标准，交付事务结果
- 如果非标准SLA，给出申请方预计完成时间，超时告知不可完成

O原则 --- 对象原则

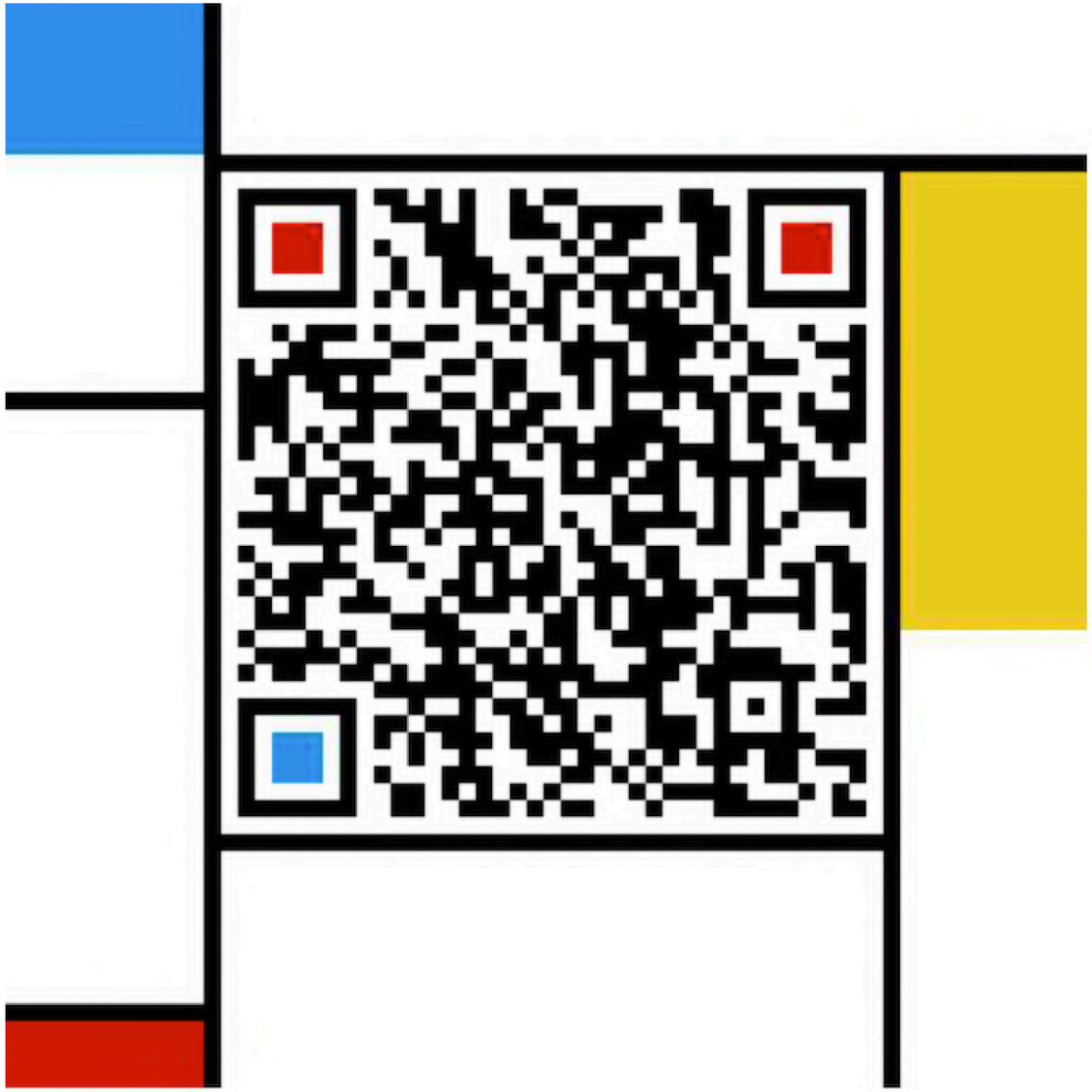
- 命名一切对象： 运维工具、操作命令、被操作的文件/目录 、操作的数据库、数据表、都必须明确命名，并且引用。
- 禁止使用口语： *“你的，我的，他的，这个，那个”*
- 命名标准： uAppo-xxx

谢谢大家!

我们正在组建强大的OpsDev团队
我们在实现云计算运维的工程控制方法

Soft define engineering process @IaaS
软件定义工程化

请用简历砸晕我 jim.xu@ucloud.cn



鸣谢:
我的老友Airwing 张翅@Povital
Shanks.zhao /sid.cao @uAppo