云•计算

拨开Paas云架构迷雾，释放内在的力量

主讲人介绍

高磊,辽宁省大连人，早年毕业于东北大学，先后在埃森哲、微软、LINE、阿里巴巴等混江湖，略有小成，专注高可用分布式系统以及云计算领域的研究和实践。喜爱古典文化，颇有哲人之风，爱喝茶而不喜欢扯淡。

Agenda


01
Paas成长史和您说一说
02
Paas要解决什么问题?
如何实现和实施？我们
需要反思什么？
03
世界变化太快，Paas
怎么适应变化哪？
Paas的未来在哪里？

云•计算编年史

1956

萌发

Christopher Strachey发表虚拟化论文，而虚拟化就是今日云计算的基石，被视为创纪元的开始。

1984

发酵

Sun公司提出“网络即计算机”，用于描述分布式计算技术带来的新变革。同年一部叫《终结者》的电影火热上映.

I998

婴儿期

VMware公司实现了第一代虚拟化技术，为”基础设施即服务”铺平了道路。

2004

更大的可能性

Google发布MapReduce论文，Hadoop、HDFS、HBase实现并开源。同时另一个方面，AI的创新开始逐渐觉醒！

2007

里程碑

美国大学开始开设云计算课程,预示着人类彻底进入云计算时代,同年Saleforce提出”平台即服务”(Paas)的概念,并推出第一个实际的Paas平台。

2009

中国年

阿里巴巴进入云计算领域，中国移动云计算平台计划启动。

2010

分水岭

微软Azure云平台正式发布，并宣布90%的员工将从事云计算相关工作，预示传统IT与新时代交替的分水岭。同年由Rackspace和NASA发起了名叫OpenStack的开源项目，成为“基础设施即服务”的标准，标志着IAAS平台的成熟。

2013

提升

请记住这个日子，dotCloud公司宣布开源Docker,同时间，Paas从概念走向成熟，各大云计算公司的传统Paas平台落地，同时新型的PAAS借由Docker开始萌发。

2015

爆发

“容器即服务”平台成为现实，逐渐成熟并成为主流，Openstack宣布全面拥抱Docker,当时容器云已经可以支撑超大型电商的核心业务,基于大数据能力的AlphaGo战胜人类棋王。大概这一年《终结者》电影显示了T800的内核版本是Linux 4.1.15。

2016

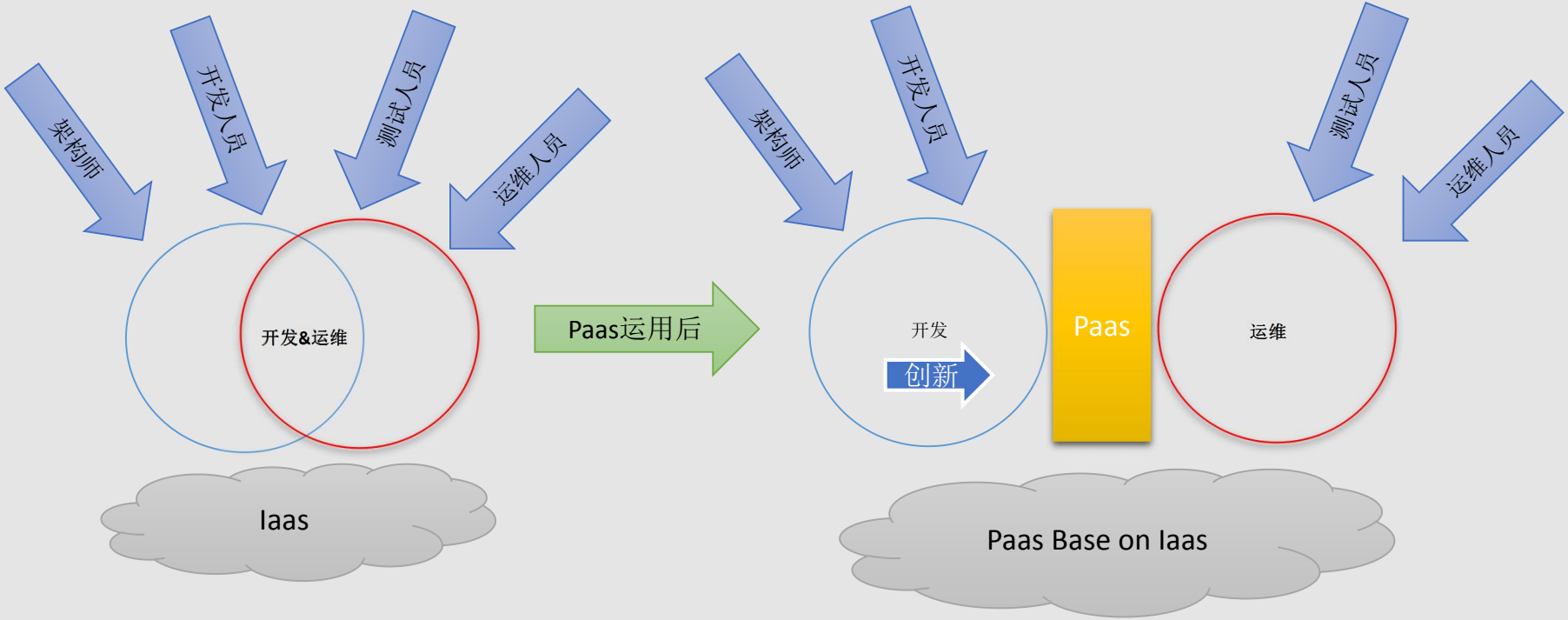
现在与未来

基础设施正稳步向“网络即计算机”和自主AI方向迈进，云计算企业进入了战国时代,几乎重现了1975年PC诞生时的情景，然后今后必然在标准上走向统一，”天网”时代正在走来。


风起了，云来了

为何偏爱了Paas？

Iaas只是解决了物理机部署和治理更加灵活以及保全能力，但是亲的VM上依然需要自己管理OS、版本、配置和所有与开发与部署的事情，并没有解决开发和运维的鸿沟问题，不过在实际当中，
Iaas解决不了的问题更加多，说白了Iaas提供的是一个物理层的虚拟层。所以Iaas之后出现了Paas。



仅仅如此吗？不！绝不仅仅如此！

Paas是下一代的云！除了将开发从运维中脱离出来，将运维外包给平台，它更多的是对开发以及运行时的全面支持，可以看成是开发人员的云、运维人员的云、NoOps的云、产品运行的云、支持创新的云！你见过这么多面孔的云吗？它是个全生命周期的平台，比Iaas更关注应用，而Iaas关心的是物理资源。

解决互联网系统的什么问题？

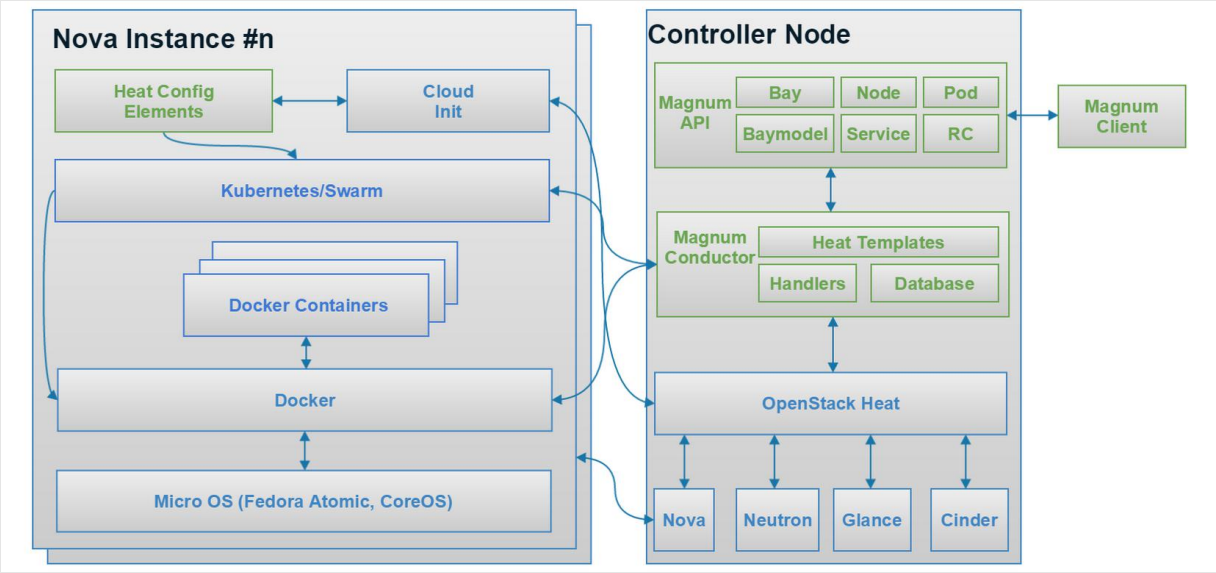
1.提供一致性环境保障。
2.提供应用多租户隔离以及资源的多租户隔离。
3.提供一致的中间件集合并降低移植复杂度。
4.提供开发人员全生命周期的工具：开发、调试以及部署自动化。
5.提供运维人员统一的、聚焦应用的运维工具。
6.提供服务发现、可弹性伸缩、状态管理、资源分配、动态调度等能力。
7.提供非侵入性的运行时支持。
8.提供透明化的监控、自动恢复与容灾能力。
9.逆转以往过分关注平台而非关注人的问题，为人的创新提供更多的支持！
10.提供平台拓展机制，让其更加适应多变的业务场景。
11.提供可定制的数据迁移工具等等。
12.从应用角度来管理Iaas平台。

这就像一个大的工厂，给你提供全套基础设施和工具！大大填补了Iaas的不足。

与分布式系统的关系？

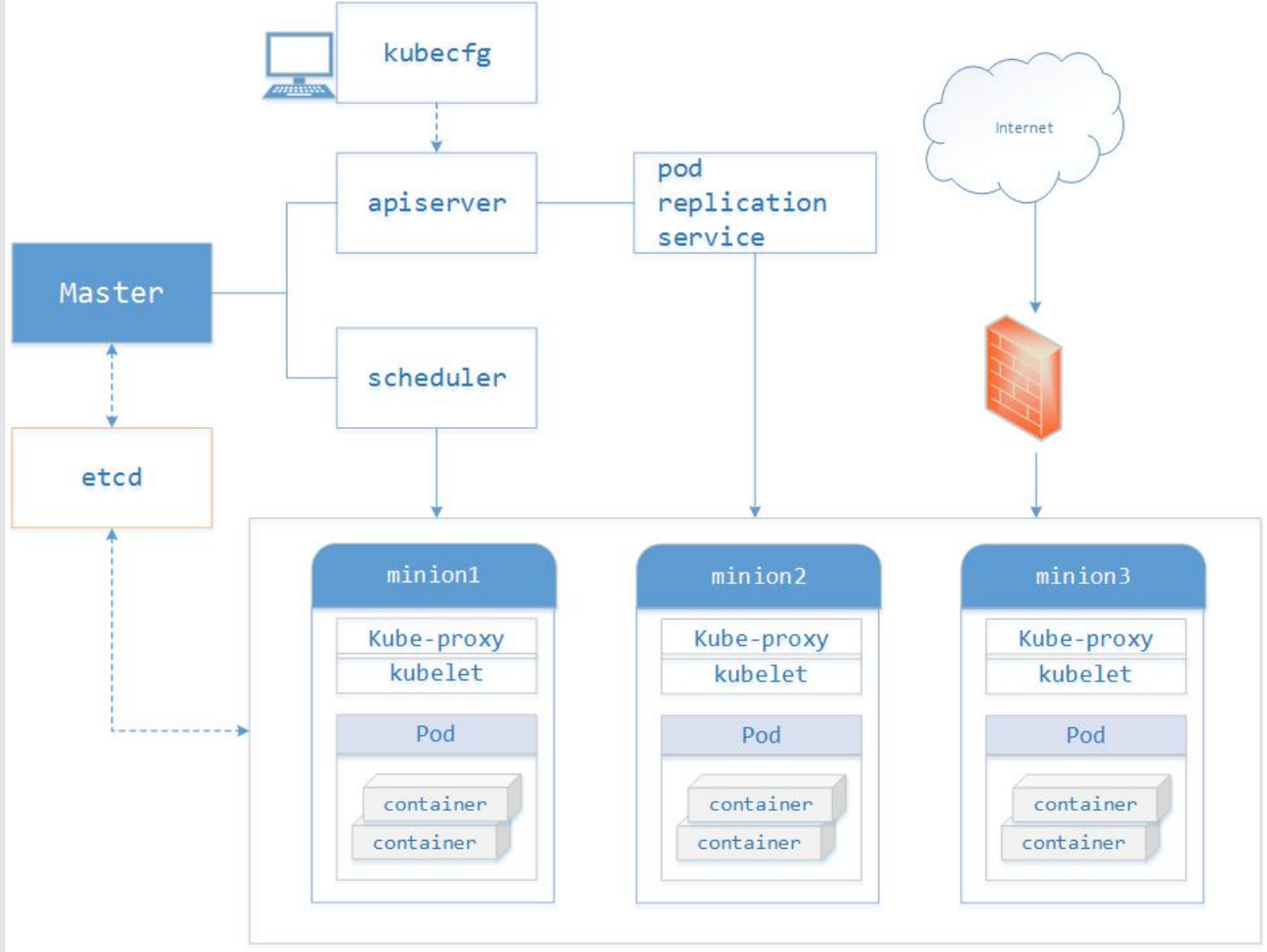
1.技术实现上与传统分布式系统没有什么区别，区别就在于更加聚合、更加在乎治理策略、更加在乎提供平台化服务。

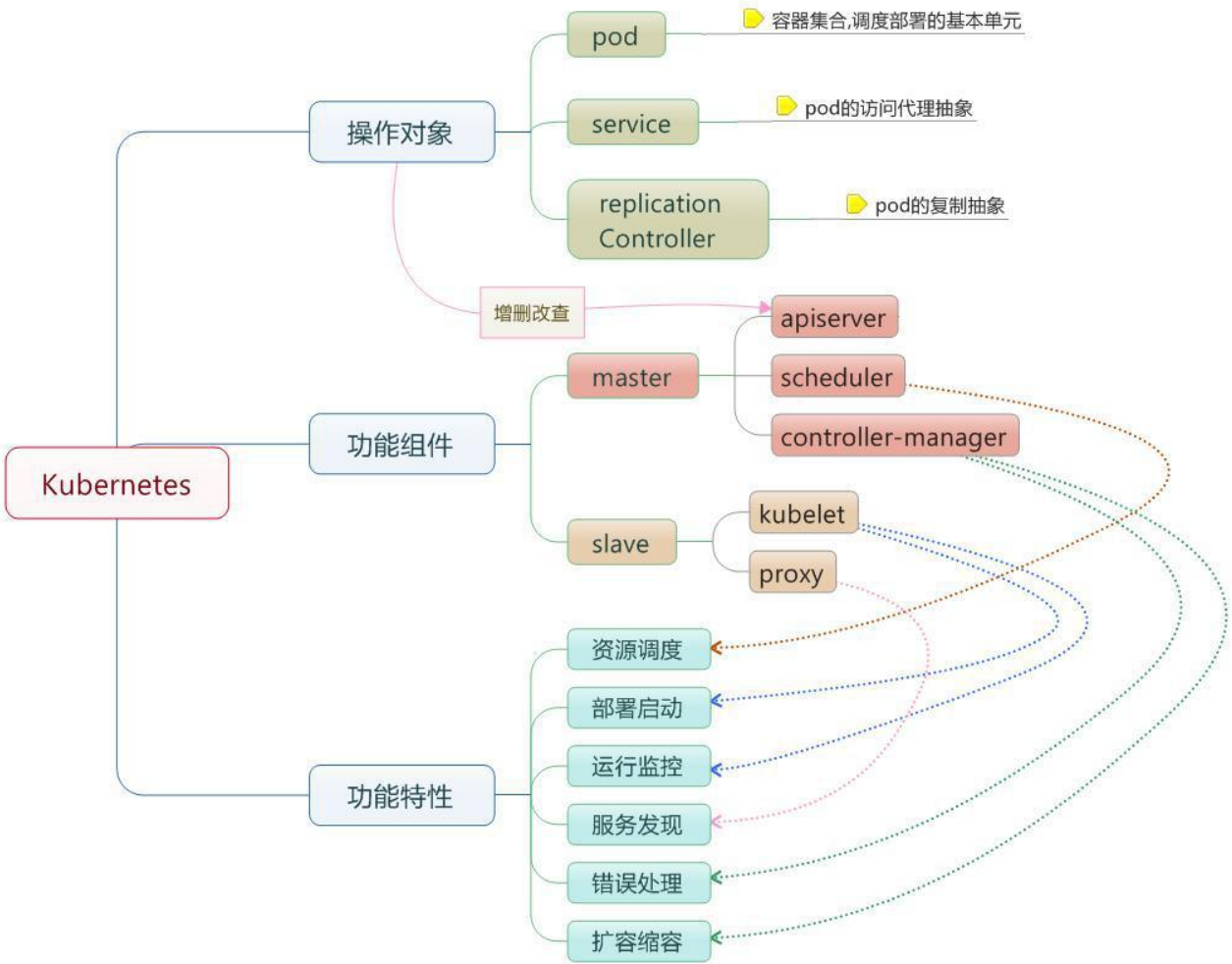
2.辟个谣：目前出现了Docker，但是并没有改变Paas依然是分布式系统的本质，如果您去看K8s的源码，或者Openstack（IAAS）的源代码，其实实现依然依靠分布式的实现方式。Docker只是为应用本身提供了运行时容器。从平台看过来和Tomcat等没有什么区别，只是在技术实现上有区别。---但是不可否认的是Docker的出现，使得内部环境一致、低延时启动和销毁、简化Paas平台自身实现等等方面有不少贡献，但是本质上并没有改变Paas的架构体系，只是影响了它向更加先进的道路迈进.



Paas架构模型


PaaS平台层
统一运维服务
云平台运维
动态伸缩
平台容错
应用运维
自动部署
应用管理
流量控制
动态伸缩
应用容错
中间件运维
集成管理
动态伸缩
安全服务
单点登录
安全访问框架
日志服务
应用组件服务
用户管理
工作流
认证服务
组织机构
积分管理
审批查询
地图服务
短信服务
.....
.....
.....
.....
服务管理平台
服务注册
服务集成
服务发布
服务管理
数据分析服务
报表引擎
海量数据分析引擎
合规与风险控制引擎
数据交换服务
物联网数据交换
信息共享
ESB业务协同
数据存储服务
数据访问框架
非结构化数据
结构化数据
虚拟资源动态申请模块
IaaS接口服务
虚拟资源动态释放模块


kubecfg
Internet
apiserver
pod
replication
service
Master
scheduler
etcd
minion1
minion2
minion3
Kube-proxy
kubelet
Pod
container
container

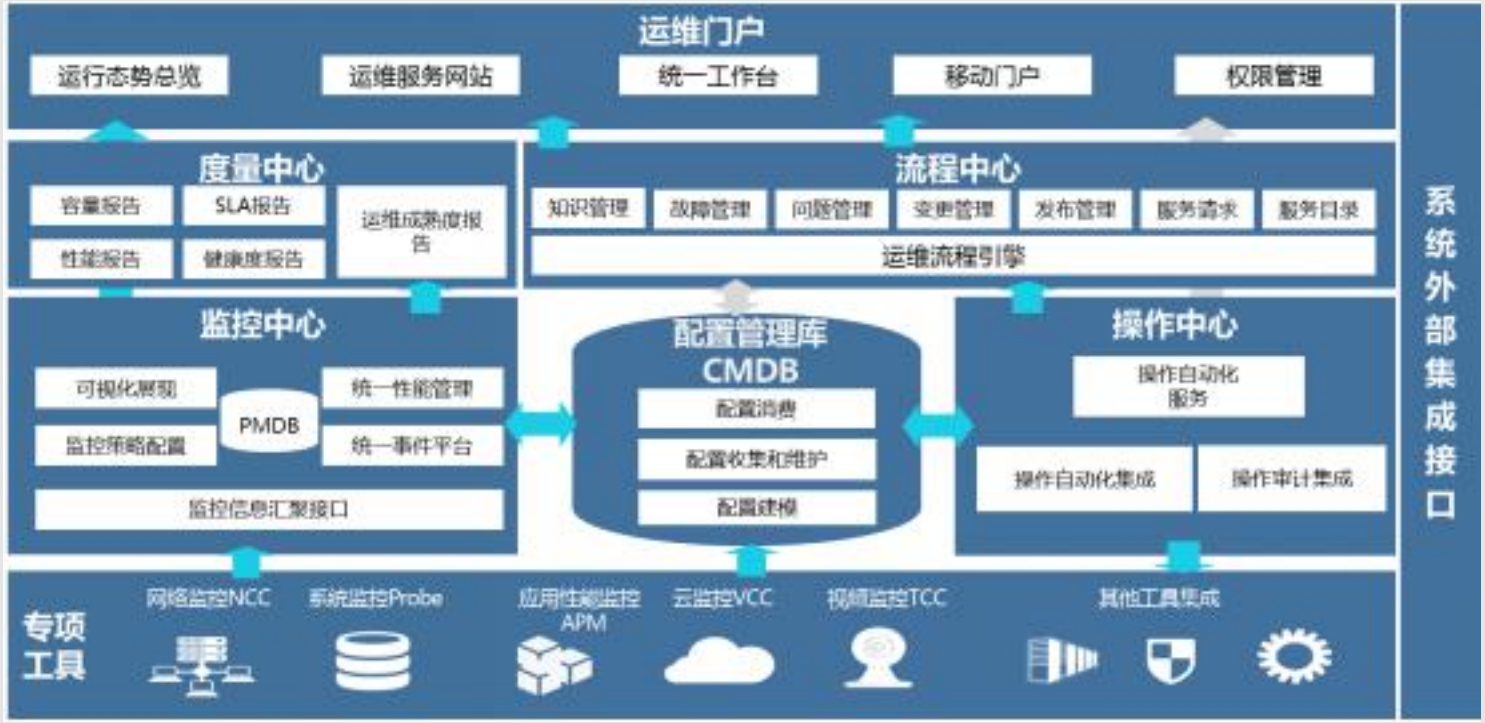

Kubernetes
操作对象
pod
容器集合,调度部署的基本单元
service
pod的访问代理抽象
replication Controller
pod的复制抽象
增删改查
apiserver
功能组件
master
scheduler
controller-manager
slave
kubelet
proxy
功能特性
资源调度
部署启动
运行监控
服务发现
错误处理
扩容缩容

Paas又分成侵入性和非侵入性的。
1.侵入性的Paas，比如GAE，TAE等，用户的程序需要调用Paas API完成工作。这种方式由于没有统一标准导致应用移植性非常差。但是应用对平台的可控性比较强。
2.非侵入性的Paas，由于不依赖Paas API，应用的移植性非常强，或者极少的修改就可以进行移植，对混合云这种场景来说，提供了一种很好的基础，目前来说，这种Paas已经成为主流。

我们接下来，我抽取一些Paas中的组件来展示一下Paas的一些实现逻辑，由于Paas平台非常庞大，涉及很多技术领域的、不同角色的组件的实现，还和整个全局的编排有关，所以不可能一一尽数说明，只列出几个有代表性的。

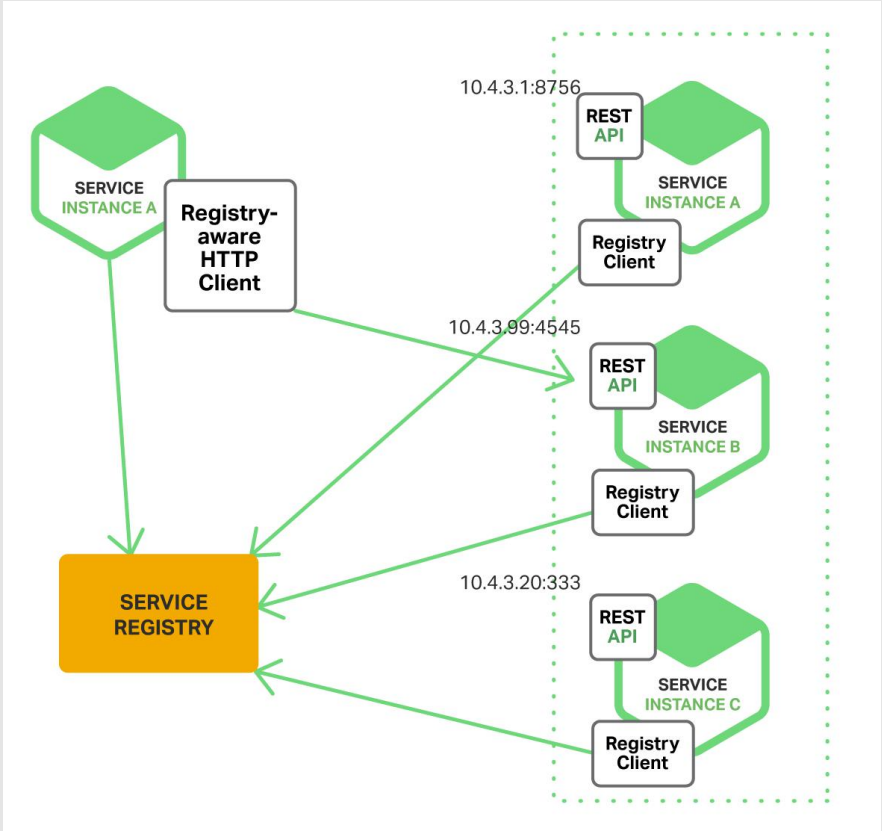
核心实现-配置管理优先

将组成Paas和业务系统的组件的关键能力以及接口元数据描述化并维护其关联关系，是Paas平台治理以及自动化的关键，配置管理比任何时候都要重要！



核心实现-服务发现和注册

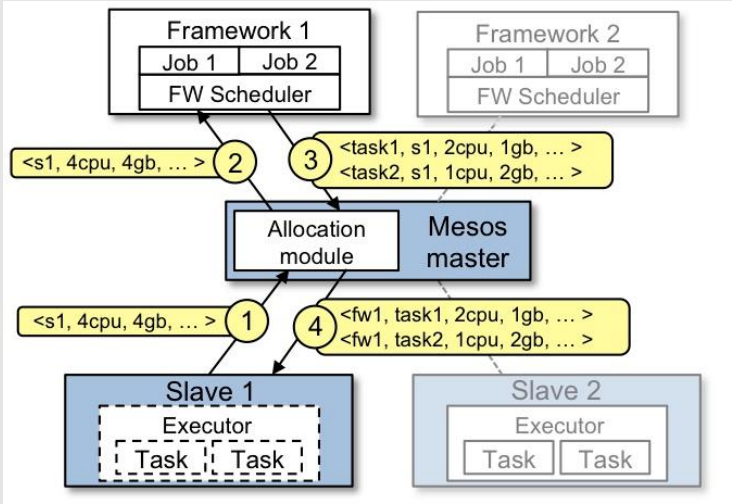
1.服务治理、弹性扩缩容、容灾等的基础
2.App地址透明化的基础。
3.资源分配和调度管理的基础。
4.发现和注册的实现基础是分布式协调软件，比如Zookeeper、Etcd、Consul等
5.以上也可以算是配置管理特殊的一部分！



核心实现-资源分配和调度

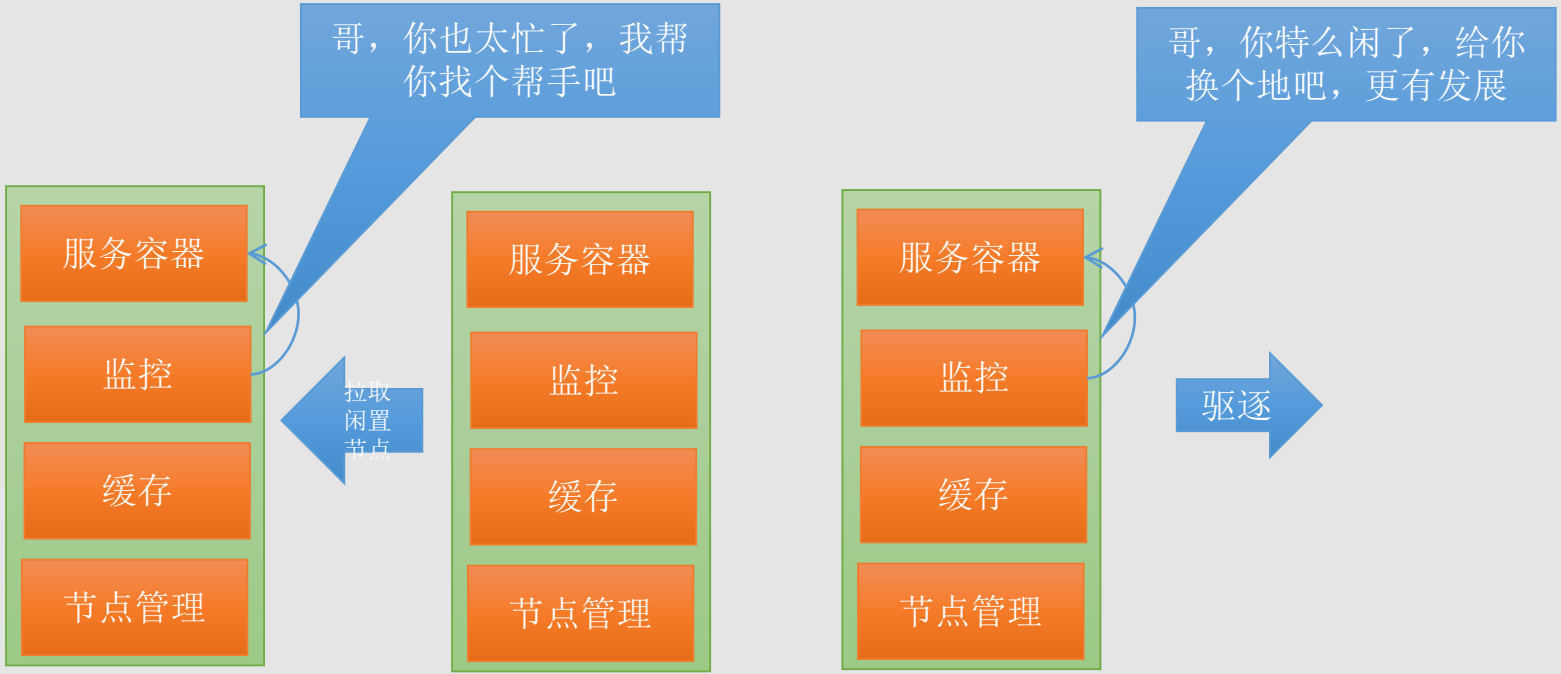
Paas的资源关心的CPU、内存、磁盘和网络，如果Paas下面有Iaas，那么关心的是VM的CPU、内存、磁盘和网络。资源分配是应用程序运行的基础，有了应用程序的运行，对请求的调度才有意义。但是调度并不等于资源分配，资源分配也无法实施调度的责任。所以一般资源管理器和调度器一定是分离设计和实现的。

1.资源分配：把资源规格化，比如1CPU 512内存 200G磁盘 100M带宽作为一个集装箱能力单位，以此为单位进行分配，一个App需要3个单位的资源，另外一个需要4个等。基于监控和Iaas API，就能够弹性管理资源分配。
2.调度：部署时的调度决定服务被部署到什么地方，对用户是透明的。运行时调度决定用户负荷落到哪个一个实例。而这些也需要基于配置管理和监控指标的反馈机制。目前来说有三种：单体调度、双层调度、共享状态调度。



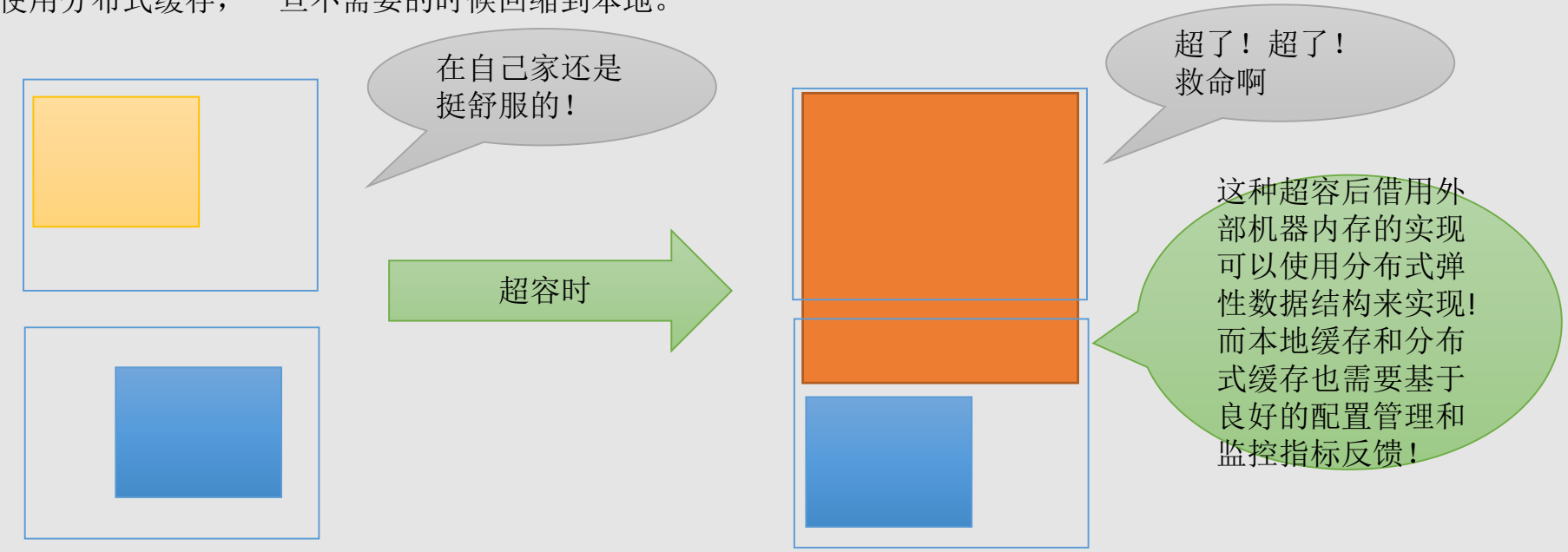
核心实现-向外的和向内的弹性

1.一般集群内保留有冗余的服务器，可以是冷备，也可以是热备，采用底层PXE唤醒或者调度实现容量不足的时候将业务请求分摊到新节点，或者峰值过后收缩资源的占用。这是外部弹性的体征。
2.对内的弹性是指，我们可以在固定的服务器实例数量情况下做到根据热点实际情况的合理分配。如何实现？



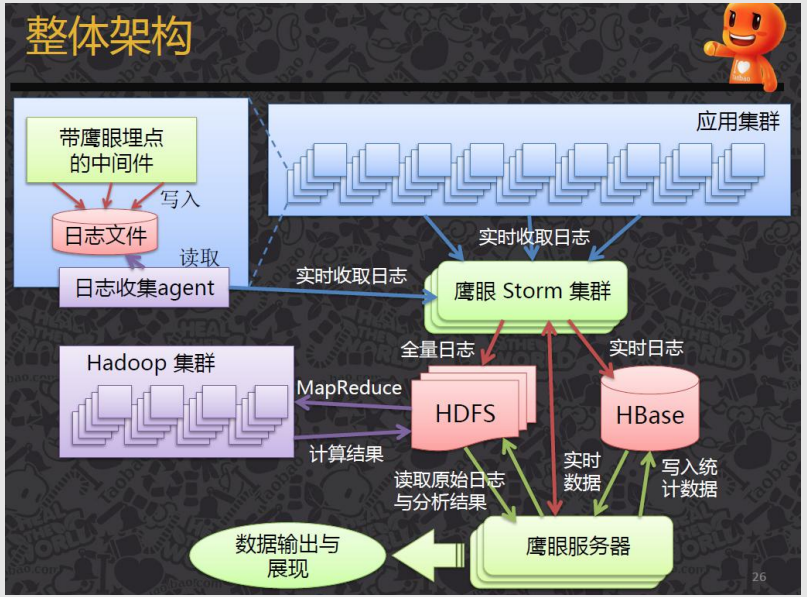
核心实现-缓存本地化与分布式化的折中

很多业务或者类似Paas这样的分布式系统都采用分布式缓存，但是实际优化的方案是：1）频繁变化的数据尽量采用内地缓存 2）次频繁的可以缓存在本地的分布式系统上 3）长时间不变的可以缓存在CDN中。
为了经济性，在每个服务器上规定一个理论容量的边界，在边界内依然采用本地内存，如果超出边界，那么超出部分使用分布式缓存，一旦不需要的时候回缩到本地。



核心实现-流式日志

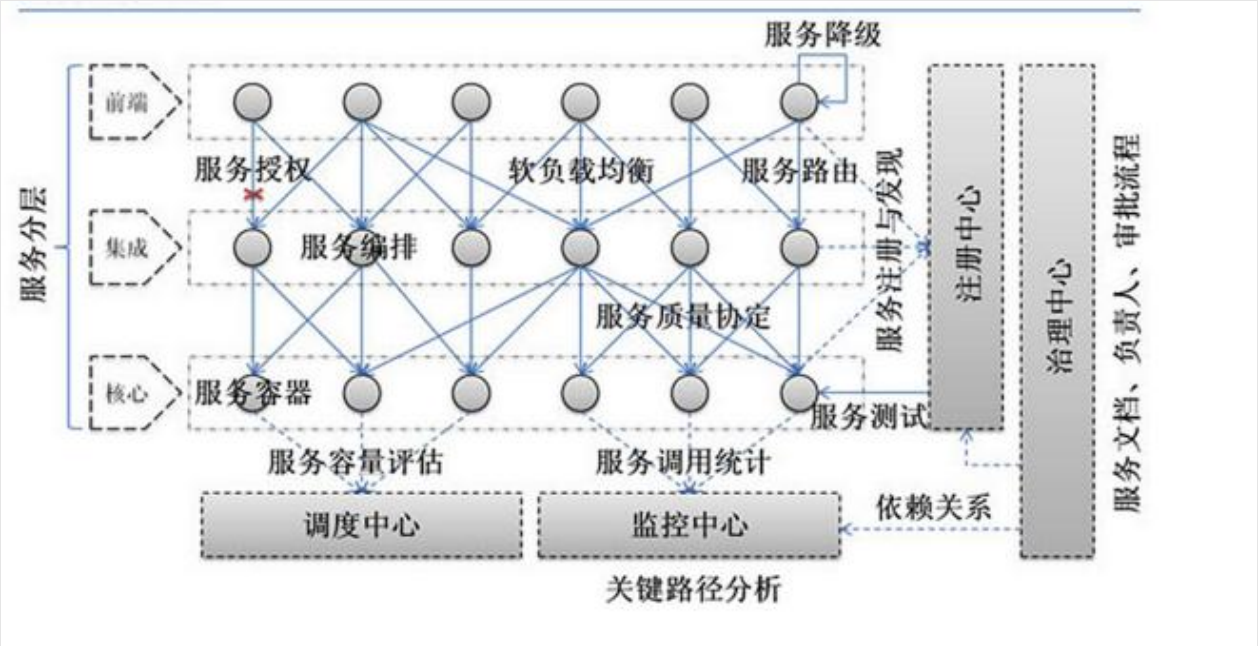
之所以日志成了流，就是因为服务化、微服务化、容器化之后服务本身生生灭灭，如果本地保存日志，该如何迁移哪？所以干脆不存了！采集点以及收集系统本身要良好运用，也要基于配置管理!



这里用了淘宝发布的图

核心实现-编译时依赖和运行时依赖

1.编译时的依赖是指对二方或者三方包的依赖，如果版本或者配置不对引起的问题。所以问题解决的关键还是在配置管理，配置管理的中心在于配置项的关联，我们可以使用Maven、Gradle等来声明依赖关系，但是这些声明又保存在配置管理系统并与App版本关联，在使用时下发到开发环境、测试环境以及正式环境。
2.运行时依赖，依靠调用链跟踪、配置管理下发配置数据来实现，根据环境指向依赖的Mock服务器、测试服务或者正式服务器。



核心实现–多租户资源隔离

云的特点是资源池化，甚至为每个用户开辟他自己独有的应用资源空间，而与其他应用隔离起来。我们称之为“多租户机制”,比如k8s的namespace就是实现了这种隔离机制.

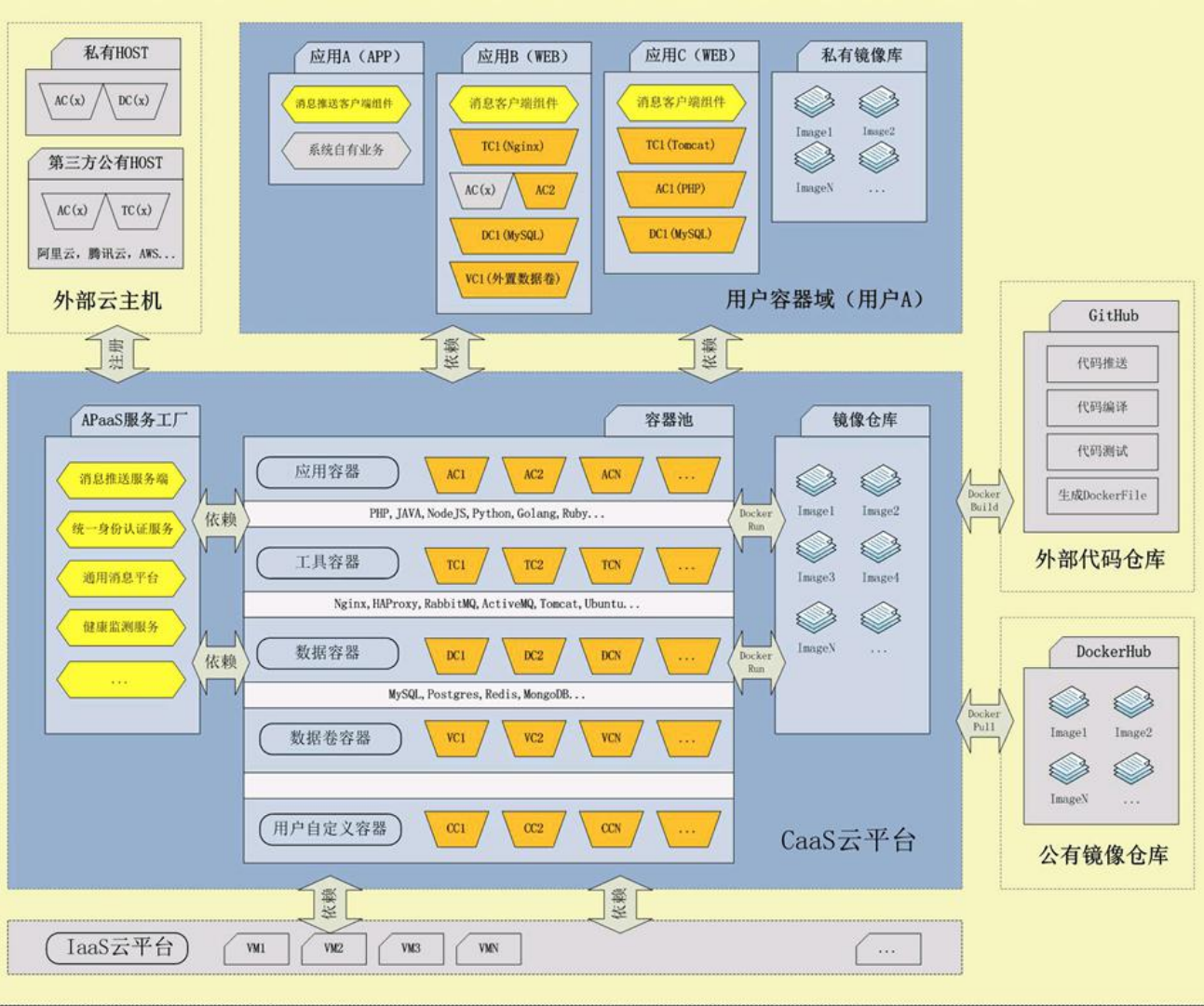
如何实现哪？

1）对于Iaas，VM是资源隔离的非常好的手段，但是比较重。

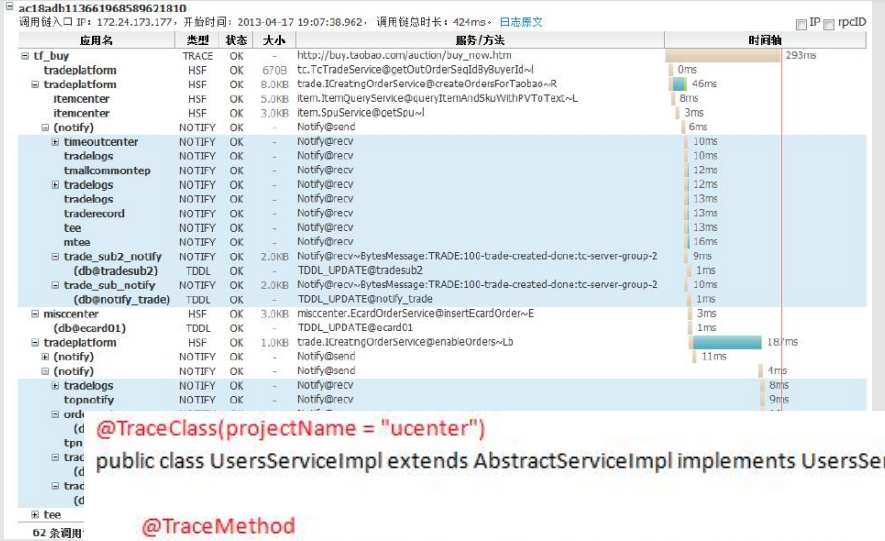
2）LXC实现了OS级别的资源隔离，并演化成类似Docker这种隔离手段。但是依然是本地隔离手段。

3）需要结合1-2的本地级隔离手段，在Paas上层的资源分配机制上进行处理，资源分配时将各个资源汇聚成一个隔离的组并挂接或者注册到一个App的“名下”，这些都需要配置和元数据管理的支持。以后再分配资源时，其他App将不会获得当前App的资源，并且当前资源如果崩溃，也不会影响到其他应用。

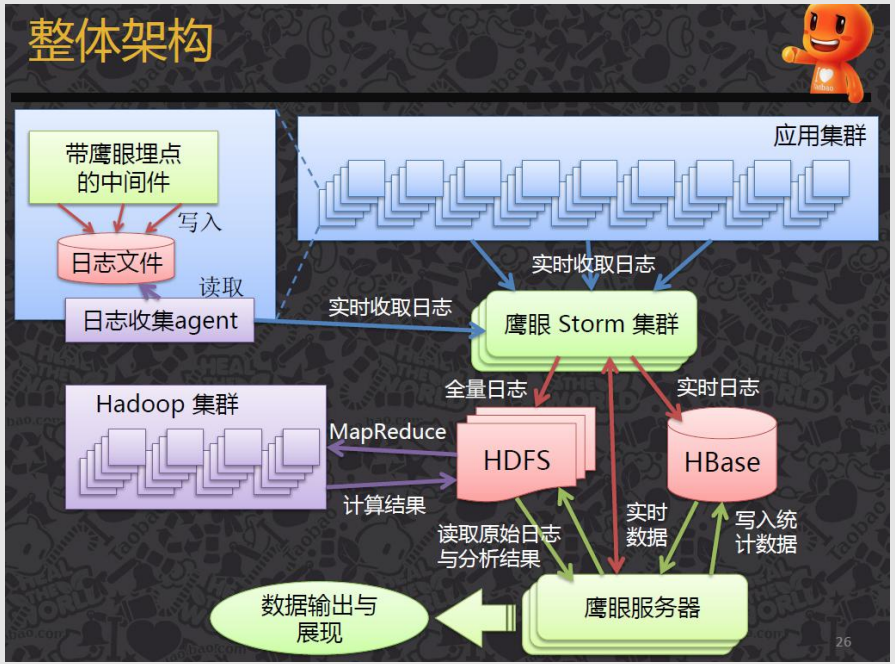
4）网络也可以进行隔离，比如Vxlan，或者划分专有网络：计算网络、存储网络、管理网络等等


私有HOST
AC(x)
DC(x)
第三方公有HOST
AC(x)
TC(x)
阿里云，腾讯云，AWS...
外部云主机
应用A（APP）
消息推送客户端组件
系统自有业务
应用B（WEB）
消息客户端组件
TC1(Nginx)
AC(x)
AC2
DC1(MySQL)
VC1(外置数据卷)
应用C（WEB）
消息客户端组件
TC1(Tomcat)
AC1(PHP)
DC1(MySQL)
私有镜像库
Image1
Image2
ImageN
...
用户容器域（用户A）
注册
依赖
依赖
APaaS服务工厂
消息推送服务端
统一身份认证服务
通用消息平台
健康监测服务
...
依赖
依赖
容器池
应用容器
AC1
AC2
ACN
...
PHP, JAVA, NodeJS, Python, Golang, Ruby...
工具容器
TC1
TC2
TCN
...
Nginx, HAProxy, RabbitMQ, ActiveMQ, Tomcat, Ubuntu...
数据容器
DC1
DC2
DCN
...
MySQL, Postgres, Redis, MongoDB...
数据卷容器
VC1
VC2
VCN
...
用户自定义容器
CC1
CC2
CCN
...
Docker Run
Docker Run
镜像仓库
Image1
Image2
Image3
Image4
ImageN
...
CaaS云平台
Docker Build
Docker Pull
GitHub
代码推送
代码编译
代码测试
生成DockerFile
外部代码仓库
DockerHub
Image1
Image2
ImageN
...
公有镜像仓库
依赖
依赖
IaaS云平台
VM1
VM2
VM3
VMN
...

核心实现-全链路跟踪和分析系统

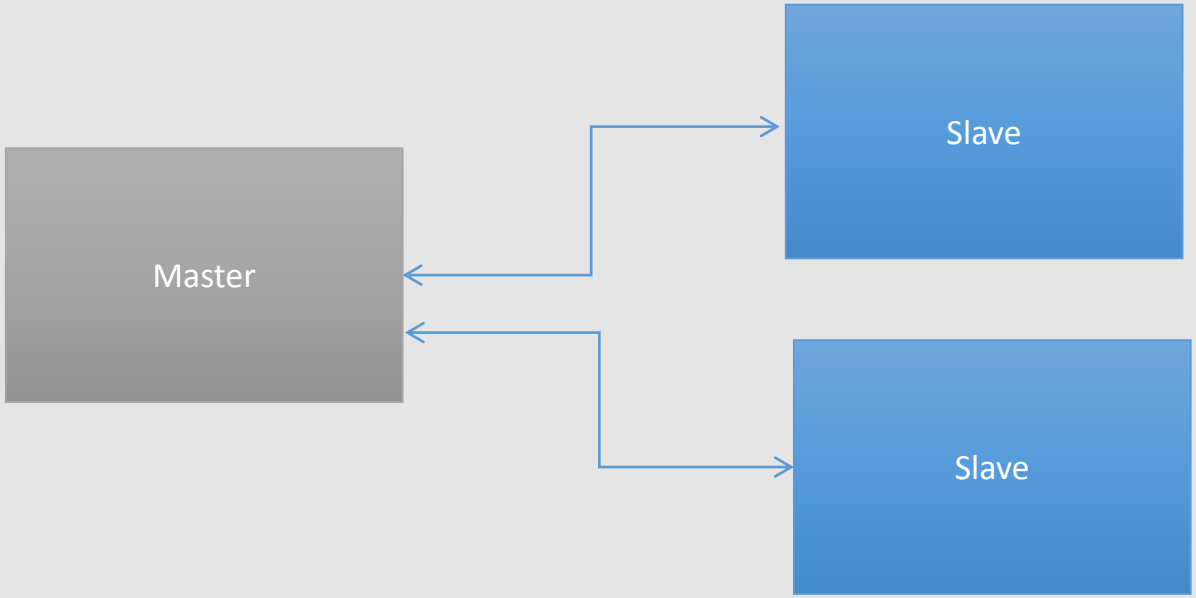



```
@TraceClass(projectName = "ucenter")
public class UsersServiceImpl extends AbstractServiceImpl implements UsersService{

    @TraceMethod
    public UserResult findUserInfoByNick(final String nick,final RpcCallArg rpcCallArg) {
        //如果参数中有 Message 对象，则优先从 message 中取 rootId 及 parentId
        //你的业务逻辑
    }

    @TraceMethod(messageType=MessageTypeEnum.CENTS)
    public UserResult findUserInfoByNick(final String nick) {
        //你的业务逻辑
    }

}
```



核心实现-链路优化

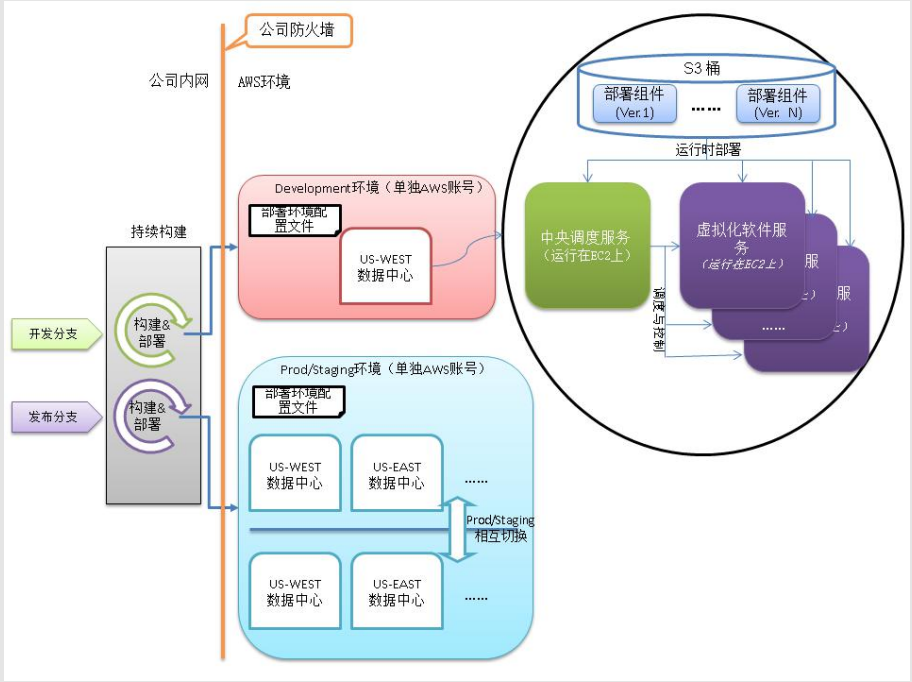
将监控集群状态的心跳与集群命令下发合并，优化通信线路的负载。



核心实现-部署自动化-系统层面

核心实现-部署自动化-应用层面

1.部署时涉及编排的问题，也就是服务应该以何种状态部署哪一个容器或者节点上，以及与其他节点的关系策略：相亲性和反相亲性。
2.部署时还涉及版本管理、配置管理、持续发布和持续集成的问题，用于更好的走完生产的最后一公里。但是在开发应用期间是建议频繁地在测试环境部署验证的，可以尽快发现问题并演练部署策略和程序。



核心实现–不可忽视的本地治理

1.整体由部分组成，部分的效能最大化就是整体效能的最大化的重要因素之一，整体效能还取决于各个组件的整合和协作方式的设计以及实现。首先实现本地合理的治理，是必须实现的事情。
2.RPC的效能、服务器线程以及IO的处理方式、埋点的透明化、本地监控、本地缓存、本地调用调度（节点管理器），本地升级策略等等都是要关注的点。
3.生命周期管理：节点管理器必须实现对服务实例的生命周期管理，并与集群管理器紧密结合,在节点服务失败或者崩溃时可以试图重新恢复和拉起服务进程或者通知上层集群管理器重新调度。

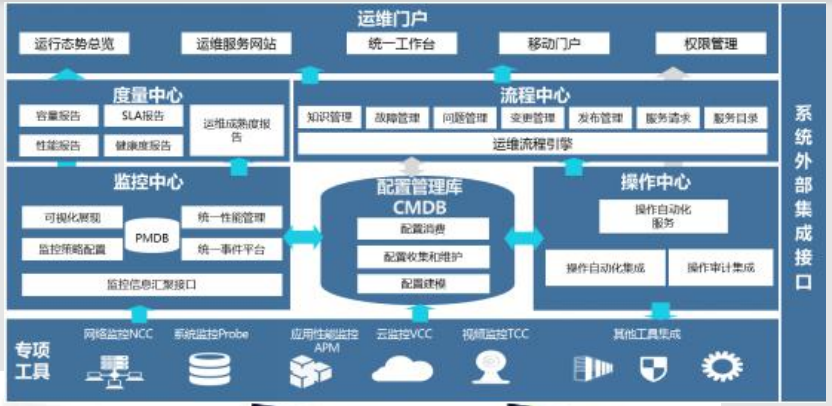
核心实现-监控闭环

Iaas&容器&基础中间件&资源分配&调度API

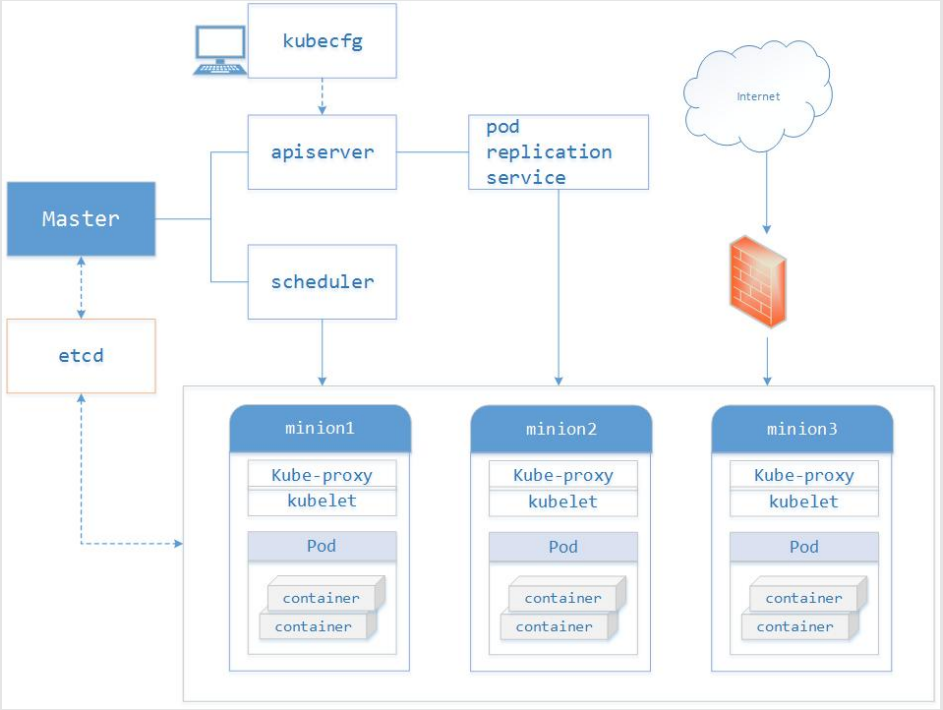
服务治理、部署管理、配置管理 API 等等

监控与运维工具，监控指标回馈给运维工具，运维工具调用内层API。

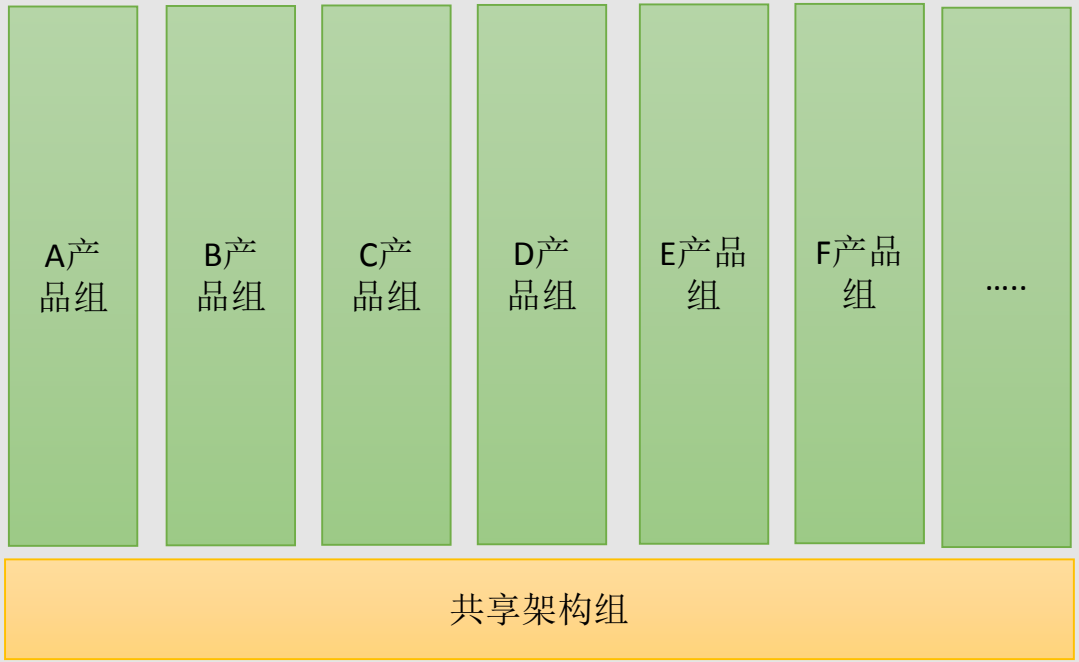
可以这么讲Paas从垂直方向治理着IAAS!



……
核心依然是分布式系统的设计，关键在于怎么
组合成一个有机的整体！云系统的内部实现是相当复杂的，所以这里只是讲了一些例子，实际情况非常复杂。
目前也缺乏标准，所以更多是厂家自己的思路。下面是K8s的一个例子。



如何实施？



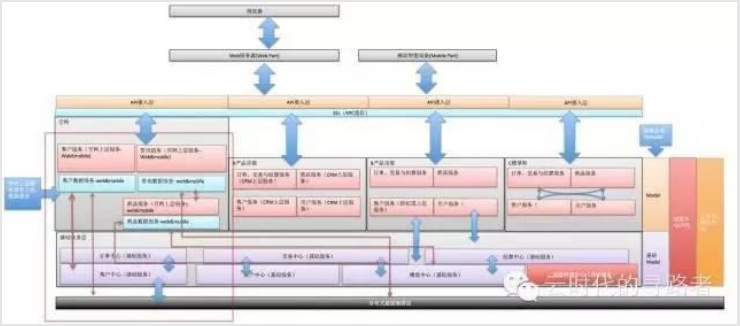
移植遗留系统-大话题

1.以上的内容，说明了Paas的三个核心内容：开发设施自动化、运维自动化和运行时环境自动化。
2.但是这样的情况直接导致了遗留系统迁移的困难，因为老时代的系统往往很“重”，一开始就被锁定在厚重的铠甲中，另外一个更加让人为难的就是释放铠甲必须修改应用代码，这就涉及业务、功能重构！这也是很多企业实施Paas很艰难的原因！
3.具体的一些情况：
A:系统服务之间使用独立密码库授权通信，难以使得自动化手段开通策略，实现自动扩容和伸缩。
B:老应用代码依赖于应用服务器的Api，比如很老的系统使用EJB架构。使得适应新的Paas变得非常困难！
C:上Paas的动因也在于老系统面临的环境从内部走向外部了，但是之前架构师设计系统时是按由里往外的思路进行的，比如重点放在交易模块上（因为此时线下活动占主流），因为业务的发展，对系统的要求越来越高，要适应互联网环境的要求，比如互联网上查询的量要远远大于交易的量，正好和原来的思路相反，是从外向里的，这造成应用架构的极大不同，那么就要求调整到分而治之的架构，那么必然要求要进行调整和重构。
D:.....

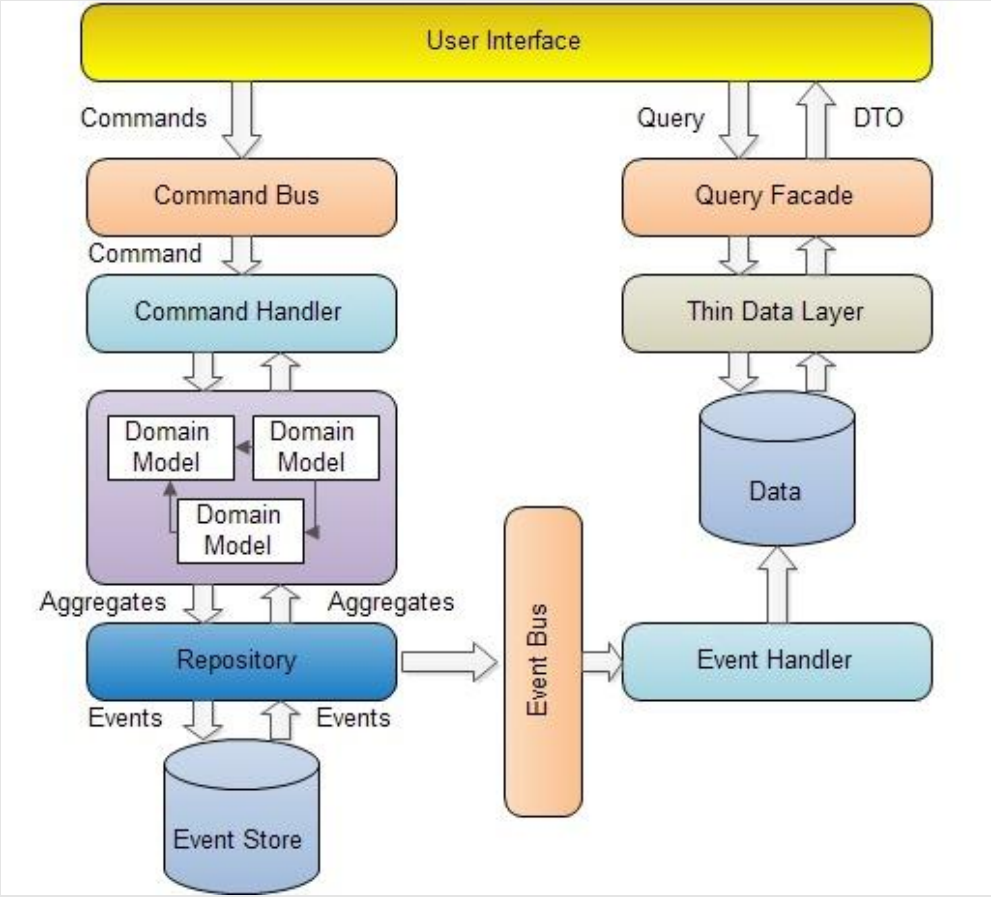
那么就没有很好的办法尽量减少这些麻烦吗？因为麻烦意味着风险和成本

移植遗留系统-解决思路

1.总的方向就是向自动化、自省化、弹性化等努力。
2.我们可以通过回答以下的问题来决定如何调整您的系统：
A:本地应用不要依赖于本地的存储来持久化数据（可以临时性的，但是给系统带来风险），日志变成流汇聚到分布式日志系统中，如果非要使用存储，请使用类似HDFS来保存，您的系统是否依赖于本地存储哪？
B:如果应用上传文件，那么这些文件应该如何存储？请参考问题一。
C:集群中多实例的配置文件是否完全与底层OS解耦，配置文件是否不依赖于主机名和Ip？（可以采用环境变量来解决）
D:应用是否存在需要很长时间处理的任务类型？（尽量异步化的方式解决）
E:集群中实例是否采用了独立安全授权的方式组建？（需要拆除）
F:集群中是否使用了硬件负载均衡设备？（需要撤除）
G:需要把服务无状态化，您的App到底有多少部分需要依赖状态？
H：您如果拆分业务系统，是按业务视角还是用户视角？（建议采用业务视角，因为热点根本无法预测）
I：业务拆分的原则是：识别基础核心业务能力、业务核心能力、上层业务能力等
.....

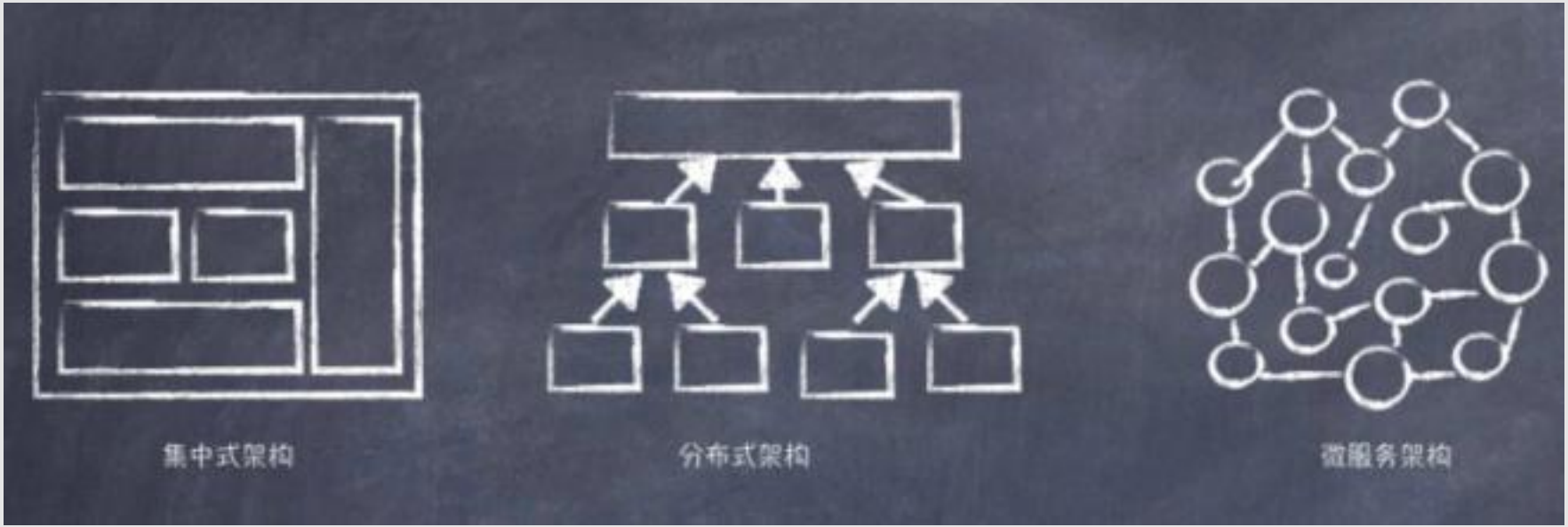


应用架构:CQRS


User Interface
Commands
Query
DTO
Command Bus
Query Facade
Command
Command Handler
Thin Data Layer
Domain Model
Domain Model
Domain Model
Data
Aggregates
Aggregates
Repository
Event Bus
Event Handler
Events
Events
Event Store

微服务是什么？

在分布式、云等基础设施支持下，从SOA演变而来的、具备明确的事务上下边界的、松散耦合的、可以并行开发、简单开发、简单或自动运维的、相互协作形成有机整体并为外部应用提供自省治理的、不间断的、高性能的服务系统。



微服务和云

前面提到互联网的分而治之的策略，大规模的分布式服务化系统在稳健的服务治理、弹性拓展、容灾以及开发周期能力支持下变得可管理、开跟踪、高性能、高生命力。服务化的基础是基础设施的稳定力！只要有一个坚如磐石的基础设施，服务化就是非常可行的决策！

套娃-软件架构之殇


PaaS平台层
统一运维服务
云平台运维
动态伸缩
平台容错
应用运维
自动部署
应用管理
流量控制
动态伸缩
应用容错
中间件运维
集成管理
动态伸缩
安全服务
单点登录
安全访问框架
日志服务
应用组件服务
用户管理
工作流
认证服务
组织机构
积分管理
审批查询
地图服务
短信服务
.....
.....
.....
.....
服务管理平台
服务注册
服务集成
服务发布
服务管理
数据分析服务
报表引擎
海量数据分析引擎
合规与风险控制引擎
数据交换服务
物联网数据交换
信息共享
ESB业务协同
数据存储服务
数据访问框架
非结构化数据
结构化数据
虚拟资源动态申请模块
IaaS接口服务
虚拟资源动态释放模块

套娃-软件架构之殇

1.在介绍核心实现的部分中，提到配置必须先行的道理，就是要保证内环境和外环境的一致性。
2.Docker虽然可以保证内部小环境的一致性，诸如CoreOS之类可以保证OS级别的版本一致性，但是Paas由于是分布式系统，它的各个组成组件也需要一致性保证，否则很难保证服务质量和延续性。
可以采用冗余服务+滚动式升级的办法解决这个难题。

不要把焦点仅仅放在Docker上！

最近一年来，看到Docker几乎弥漫在各种系统中了，但是Docker解决了App内部环境一致性、解决了传统Paas弹性拓展和容灾时效的问题\基准镜像分发等等，但是从架构上看过去，
它本身并不能解决App生产的其他大部分问题，所以当您仅仅关注Docker的话，那就失去了Paas的能力，K8s、Swarm、Mesos等解决了一部分自动化问题，但是还不完整，只是算是Mini Paas。

微服务化与Docker的关系

微服务是活在Tomcat或者Docker里面并没有本质区别，之所以业界喜欢采用Docker作为微服务的设计期和运行时基础，是因为它有着诸多好的有点，比如环境一致性可以简化配置管理、简化Paas生命周期管理的难度等等，但是本质来讲，微服务和Docker并无强关系，只是Docker的诸多方便之处，更加适合而已，所以采用Docker的系统未必是微服务，微服务构建的系统也未必是Docker构成。

Paas的未来

- 经济的发展、业务的发展所导致工作量的增加将促进PaaS得到采用。
- IaaS提供商将往堆栈的上层移动，涵盖PaaS IT，大量的中间件被云化。
- 公共PaaS将赢得中小企业市场，因为它提供了开箱即用的基础设施。
- 公共服务将把大企业市场让给私有PaaS。
- 开源PaaS平台将蓬勃发展，形成助推的作用。
- 开源PaaS平台将通过流行的Linux发行版来提供，进一步简化Paas的管理复杂度。
- 专有的PaaS将开始如同开源产品，也体现出业界标准化的意愿。
- PaaS兼容性?更像是PaaS冲突性，由于各个厂家为了在竞争中取胜，也会更多的在自己的特殊性上下功夫，造成标准的不统一的局面，但是在市场的驱动力下，标准迟早会统一起来。
- Paas变成一种生产互联网产品的工厂，逐渐成为标配，并融合在日常的基础设施中。

End