**日志君导读：**

自动化运维是一个人让人兴奋且易失控的话题！兴奋是因为我想作一次尝试，把它的全貌和细节说清楚；易失控是因为涉及点太多，一则怕遗漏，二则怕顾此失彼。带着这份复杂的心情，我们来一次自动化运维的解析之旅吧。

本文选择公众号「互联网运维杂谈」，作者微信号：waynewang。运维的同学们可以关注哈！

先做个小调查：

说实话，一个运维团队的运维能力如何，其实看一个自动化管理系统便知！

\*\*\*\*\*\*\*\***文章较长，索引目录如下**\*\*\*\*\*\*\*

**一、概述**

**二、运维自动化的三重境界**

**三、运维自动化的多维解读**

\*\*\*\*\*\*第一、基于应用变更场景的维度划分

\*\*\*\*\*\*第二、基于系统层次的维度划分

\*\*\*\*\*\*第三、基于和业务程序耦合紧密程度的维度划分

**四、运维自动化的方法论**

\*\*\*\*\*\*第一、全局驱动

\*\*\*\*\*\*第二、分而治之

\*\*\*\*\*\*第三、自底向上

\*\*\*\*\*\*第四、边界清晰

\*\*\*\*\*\*第五、插件化

**五、运维自动化系统的实现**

\*\*\*\*\*\*第一、DNS管理系统

\*\*\*\*\*\*第二、CMDB管理系统

\*\*\*\*\*\*第三、名字服务中心系统

\*\*\*\*\*\*第四、持续部署管理系统

\*\*\*\*\*\*第五、业务调度管理系统

**六、运维自动系统的API参考实现**

**七、运维自动化依赖的团队模型**

\*\*\*\*\*\*第一、团队的能力模型

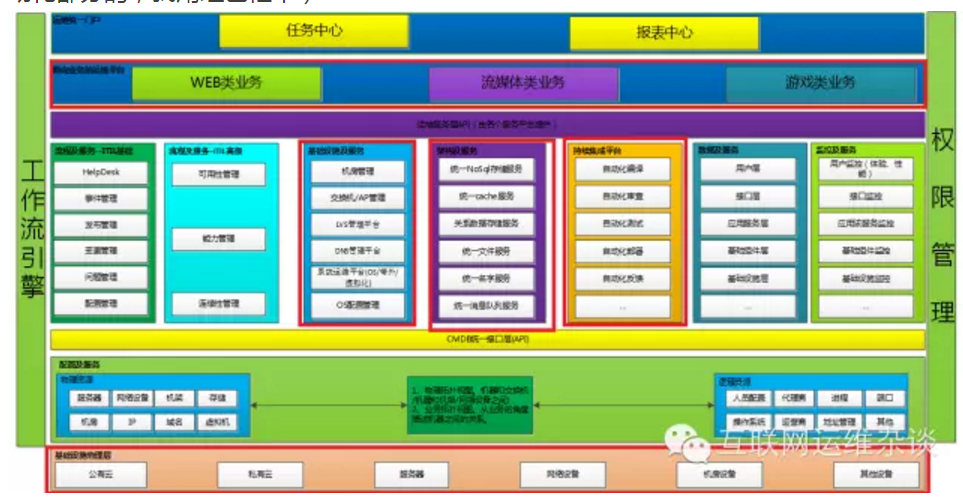
\*\*\*\*\*\*第二、团队的驱动模型

\*\*\*\*\*\*第三、团队的技能模型

\*\*\*\*\*\*第四、参考的运维组织结构

**一、概述**

在前面的文章中，提到【运维的本质---可视化】，在其中着重强调是自动化的可视化和数据化的可视化。在这个文章中，全面解码看看自动化的极致状态为什么是可视化？在前面的另外一篇文章【运维平台全体系介绍】中，也讲到运维平台体系的构成，提出“\*\*及服务”的理念，其中有几部分和自动化密切相关，比如说资源及服务、配置及服务、架构及服务，持续集成服务，最终都服务于面向业务的可视化调度平台目标上去。让我们再回顾一下平台规划体系（涉及自动化部分的，我用红色框中）：



**二、运维自动化的三重境界**

宋代禅宗大师青原行思（六祖门下首座）提出参禅的三重境界：

参禅之初，看山是山，看水是水；

禅有悟时，看山不是山，看水不是水；

禅中彻悟，看山仍然山，看水仍然是水。

这三种境界其实和我们眼中运维自动化的三重境界类似。

**自动化第一重界：看山是山，看水是水**。开始接触运维自动化的时候，我们看到了很多工具认为就代表着自动化，比如说早期把expect+ssh封装之后，就以为可以实现批量运维。看到有人说puppet可以做配置管理，这个时候也就认为puppet可以做配置管理，甚至是发布管理。这个时期的典型问题，就是以偏概全，对于某个开源自动化工具来说，还没法去界定它的使用场景和范围，直接影响系统的建设效益。这个时候已经开始知道我们看到的山不是真正的山，是迷雾环绕的深山。

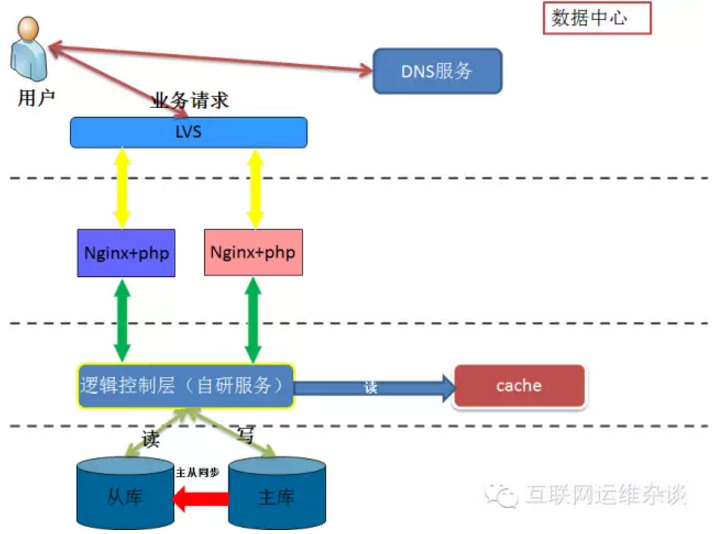
**自动化第二重界：看山不是山，看水不是是水**。此时我们知道expect+ssh不够，随着业务规模的变化，我们需要一个更完整的概念来做发布系统，真正的发布系统要做版本管理、环境管理、配置管理、还要做他们的生命周期管理等等；puppet真正要做自动化，其实还依赖OS和应用层很多标准化。对于其他资源对象的管理来说，生命周期的概念都在穿行其中，比如说DNS、LVS、接口、配置、应用包等等。为了有效标识资源的生命周期状态，需要用大量的数据来实时反馈。这是运维自动化的更具体了，把一个个的山貌看清楚了。

**自动化第三重界：看山还是山，看水还是水**。这是一种自动化本质上的追究，站在山顶之巅，俯览众山，会发出原来如此的感叹：所有自动化的本质都是为了可视化，让所有的人看到一致的服务，确保结果一致；从底层来说，你会说所有自动化的本质都是指令+文件分发的组合；你会进一步抽象系统的能力，提供即插即用的机制；结合服务化的需求，进一步云化所有的运维系统，确保内外一致性的使用。**这是化境**！

**三、运维自动化的多维解读**

**第一、基于应用变更场景的维度划分**

我们增加探讨过所有的运维价值导向最终都是面向业务、面向用户，所以自然而然就需要从业务的维度进行划分。而运维是有很多种场景的，但从业务的角度来说，核心的几种业务场景就那么几种，如：业务上线、业务下线、业务扩容、业务缩容、应用升级等五种。我用一种场景为例带大家把整个流程穿越起来看看，让大家和我一起识别流程的节点到底对接了哪些系统？那么针对其他的业务场景，你也可以用同类的方法去分析。首先预设架构如下：



**1、业务上线。**表示一个完整的应用上线，从无到有部署整个业务上线。具体的流程如下：

  
仔细看其中的流程，我们会发现涉及到多个系统，每个系统完成职能都有不同，这个地方只是大概的描述了一下。但这个流程一清晰的梳理出来，就知道我们真正要实现一个应用全部上线有多么复杂了。但看完这个图又觉得简单了，因为其实从业务上线的流程来看，我们只需要一个上层的流程调度引擎调加对应的执行器，执行器通过API和底层各个系统对接。所以说之前在框架图中，为什么要求各个专业系统一定要向上提供API，并且要求这个API的风格是一致的。

最复杂的业务上线已经梳理完成之后，其实业务下线就很简单，它是上线过程的逆过程，上线负责装，下线负责拆。

业务上线之后，随着用户活跃度的上升，此时业务的容量会有出现不足的情况，此时就需要进行业务扩容。扩容就很简单，当哪类节点不足的时候，对他进行扩容。至于扩容要做哪些变更，其实都是业务上线的子流程。比如说web层容量不够，那就无非申请机器，安装组件、下发应用包，自动化测试。这个时候需要注意：在业务上线的过程中，我们把很多的配置信息都下放到CMDB中了，但我们选择扩容的时候，就从CMDB把信息读取出来，指导变更。

**应用升级**，目前持续集成讲的自动化都是在这块。简单来讲，就是升级程序包、升级配置、执行额外指令等等，逃脱不了这几种模式。如果你说的这么简单，是不是我把ssh封装一个UI出来，就可以了。当然不是，这个时候需要你带着运维的理解，需要在底层做一些标准化的工作，否则你提供的是一个工具，完全没有运维的思路，比如说程序运行属主、运行路径、监控的策略等等。另外建设应用发布平台的目的，就是要让测试、Production环境的运维变更可控。

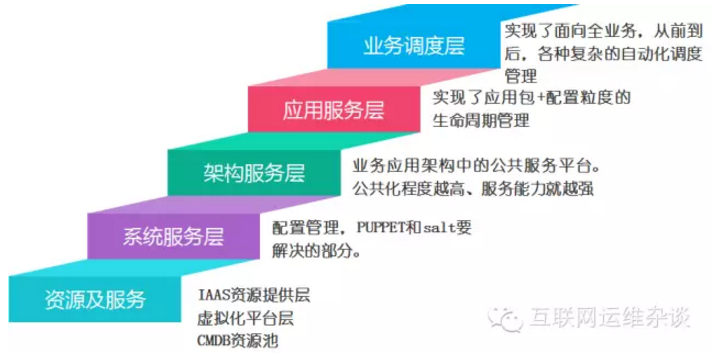
是不是以上几个运维场景的自动化要一次全部做完呢？不是，是有先后和主次之分。对于以上的运维场景，我在当前我负责的游戏运维中下做过统计，数据如下：

**A、持续部署的数量**，**一个月2000次左右**

**B、其他场景的分布情况.**一个月上线一次、下线两次、扩容一次左右。

有了这个数据，我们建设一个自动化系统的时候，就能识别先做什么后做什么。当然不同的企业有不同的实际，还是要找到核心痛点。不是一上来就建设完整的业务变更系统，见效不快，且容易让项目收益不大，而遇到很大的阻力。

**第二、基于系统层次的维度划分**



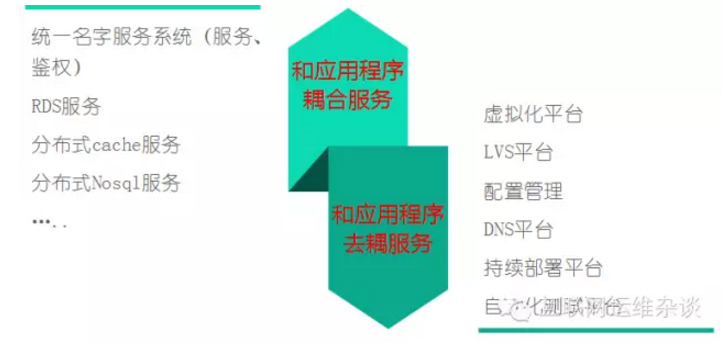
给出这样的分层体系图，其实是为了让大家更好的识别系统的职责和范围，下层干上层的活，上层干下层的活都是越界，越界带来的是耦合。举个例子，系统服务层puppet（或者chef）配置管理，在网上看到的很多资料都说可以来做发布，就是说可以做应用服务层的事情。后来我看过几家公司，用puppet来做应用服务层的发布，最后都走不下去，应用包的需求变化太大，靠puppet编写factor的模式来适应所有的场景，基本上是不可能，所以说它适合的是系统配置管理。以上说的就是一种越界！

**第三、基于和业务程序耦合紧密程度的维度划分**

这点非常重要，这个划分其实是确定系统建设的Owner，避免让运维团队承担过多的系统建设职能，而让运维能力提升缓慢。那怎么来判断和业务程序的耦合紧密程度？我的准则就非常简单，程序直接调用的就是紧耦合，类似api/SDK类的后端服务，比如研发建设；程序不直接使用的就是松耦合的就是运维来建设。

那有一种情况，我们很多应用程序中,DNS和LVS服务也在程序调用链中存在，怎么办？在我的方案中，绝对不允许内部服务走DNS和LVS。我们都知道DNS和LVS的服务对于服务异常的处理（DNS无状态、LVS是七层能力弱），远远达不到线上服务的要求，所以要坚决拒绝。如果他们真的要使用，第一告诉他们业务风险；第二，故障产生的时候，需要让研发参与处理。另外这也是系统的边界没划清楚，是让运维组件去承担业务上应该具备的容灾容错功能，会给后面的运维系统建设增加很多不必要的功能。

我这边的分类就很简单，如下：



**四、运维自动化的方法论**

**第一、全局驱动**

无论是从全部的自动化管理平台规划，还是某个平台的规划，都希望大家都能找到一个全局的立足点。比如说我们当时成立持续部署服务平台的时候，大家把全局目标一说，开发、测试、运维很快就达成共识了。目前这个平台建设完成之后，运维已经彻底的退出发布变更流程之中，真正实现了让运维变成的审核者。

**第二、分而治之**

在上面的几个维度看到了很多系统，我们发现每个系统都要建设的话，其实周期和难度都很大。所以需要分而治之，特别是线上架构组件的管理系统，更需要随着组件的交付一起交付管理能力。之前我也表达过类似的观点，所有只交付组件，不交付管理能力的研发都是耍流氓。因为从运维的角度来说，越来越多这样低价值的交付产品，会导致运维不堪重负。而让运维从头去构建这个管理，则需要花费运维很多的时间去了解，让系统建设周期拉长。举个例子，比如说某个分布式cache服务，做的不好的，通过读取日志然后对其监控，做的好的，给你开启一个管理端口，从端口中读取状态信息。这就大大降低的系统的复杂度（不用日志采集和处理组件了）。

分而治之，其实就是让不同的团队做不同的事情，不要全部压给运维；其次不同的时期建设不同的系统，不要在同一时刻做很多系统，避免战线过长。当然如果有很多运维研发人员来说，另当别论哈。

**第三、自底向上**

自底向上，其实是让大家找到一个更清晰而具体的系统建设目标来展开工作。从系统分解上，来规避大家被一个庞大而模糊的目标带入歧途。如果一上来，我们就说要做一个全自动的运维管理系统，很容易让运维研发团队迷失方向。所以这个地方可以把全局和最终目标设定在哪儿（全自动化），然后从底下逐步构建地基，做框架，最后再盖一个完整的房子。

**第四、边界清晰**

边界有两个维度，一个是管理边界；一个是职能边界。第一个边界是从Owner者的角度出发的，谁产生服务，谁就是owner，管理统一都是运维。比如研发提供一个统一分布式消息队列服务，Owner是研发，他应该对可运维性负第一责任，不要让运维去承担这个服务的webAdmin管理系统建设任务。其次是职能边界，深层次的理解是组件的功能范围。有时候对运维架构师的考验就在这儿，比如说让LVS去承担业务异常的容灾和容错切换，不合适；让DNS跨过LVS层，负责对后端服务异常的自动容错处理，也不合适。没有把职能界定清楚，会导致系统做很多无用功。

**第五、插件化**

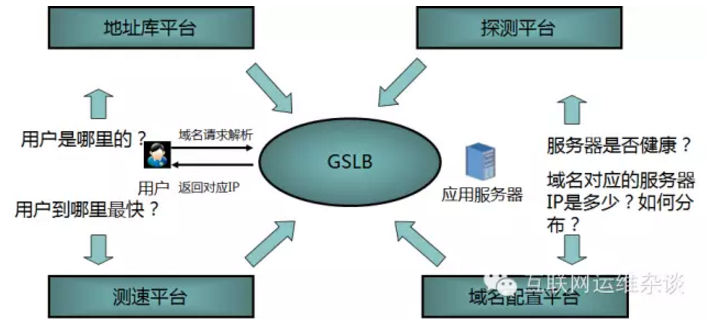
插件化的思维无处不在，但我们面对纷繁复杂的管理对象时，我们进行抽象，提供管理模式，具体的实现交给用户，这个我们日常所见的运维系统中经常可以简单。比如说nagios就是一种插件化的采集思路。对于配置管理来说，puppet也是采用这个思路。到我们最上层的调度管理系统来说，可以让运维自己去编写自己的执行器，特别是和业务紧密相关的，但最终运维整形控制权还是交给平台。而我的经验是，在【应用服务层】和【架构服务层】，不要引入插件化的管理方案，过多的插件化部署，会让我们生产环境的管理最终混乱不堪，最终失控。所以提供类SSH界面的运维发布和部署平台，是没有任何运维价值的。

**五、运维自动化系统的实现**

挑战一个自动化的极致场景（可视化），是运维人对极致的追求。接下来，我会拿几个典型的运维自动化系统供大家参考。

**第一、DNS管理系统**

**简介：**DNS系统传统web形态下的一个重要入口，用户服务的访问严格依赖这个服务入口。现在外面一般都叫GSLB（全局服务负载均衡调度），目前是CDN服务里面的重要服务节点。实现的目标都是要解决运维从哪里来，到那里去最快，当目标机房故障的时候，如何把服务调度走。概念图如下：



在移动app的今天，DNS协议的缺点已经逐渐暴露出来了，DNS解析时间长，另外还经常被劫持的。因为有端的控制，现在逐渐开始走httpDNS的服务，通过http服务的方式获取域名对应的IP地址，此时由DNS平台直接提供http服务对外。在有端APP的情况下，还可以识别非自己权威DNS域名是否存在被劫持的情况下，这个可以借助端的数据挖掘技术。此时系统需要保持和业务的与时俱进。

这个地方还有一个问题要注意的，内部DNS是不是可以统一管理？理论是可以的，把一个一个机房当成一个个的view，不过我不建议两个场景耦合在一起，虽然能够实现。

**系统Demo：**

**第二、CMDB管理系统**

简介：在之前的【运维平台之CMDB系统建设】，我有全面的体系化介绍，不再细述。

系统Demo：

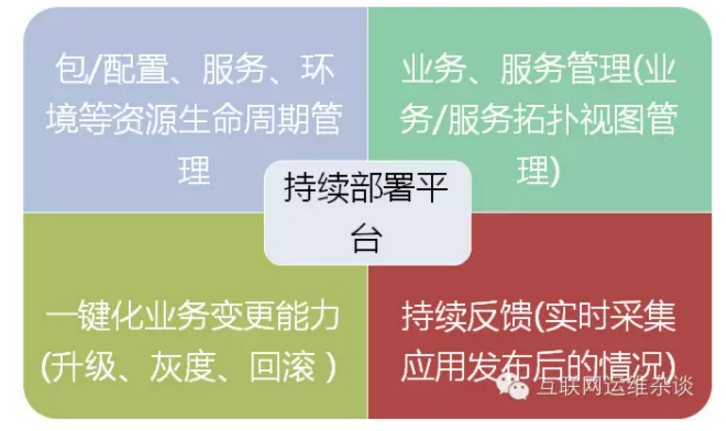
**第三、名字服务中心系统**

简介：概念最初来自于zookeeper，我们结合我们的实际，实现了名字服务中心。把程序接口之间的调用抽象一个一个的服务之间的调用，在服务中心来实现调度的统一注册、鉴权、acl、容灾容错控制。说这是线上服务最核心的系统，一点不为过，并且是收益最大的系统，直接替换掉DNS、LVS。我后面挑一个专门的章节来介绍这个系统，以便给你们的线上服务架构提供参考。

系统Demo：

**第四、持续部署管理系统**

简介：持续部署，是我们应用升级的核心系统，每个月承担着大量的变更。在系统规划之初，我们就给他设定了清晰的业务管理目标：做持续集成的一部分，实现四个下图的四个维度管理目标；也设定了具体业务运维目标：结果所有的包、配置升级，且让业务运维彻底的退出业务变更流程。如下：



**系统Demo**：

这个界面实现了最复杂的配置管理职能。

**第五、业务调度管理系统**

简介：面向业务的调度管理系统，是一个流程调度引擎+执行器来完成的，目前我们当前正在实现中。其实大家可以看看云编排服务，基本原理类似。

Demo（09年左右实现的一键化变更系统），如下：

还有数据库运维管理平台和分布式cache管理系统......都有相应的实现，这个地方不贴图介绍了。

**六、运维自动系统的API参考实现**

所有底层的系统对外提供服务都是通过API暴漏的，供各个系统使用。接口的使用需要通过授权获得，建议这个授权可以基于系统级别，也可以到接口级别，而不是统一开放的模式。另外接口内需要有相应的一些权限控制，避免底层服务被任意操作。

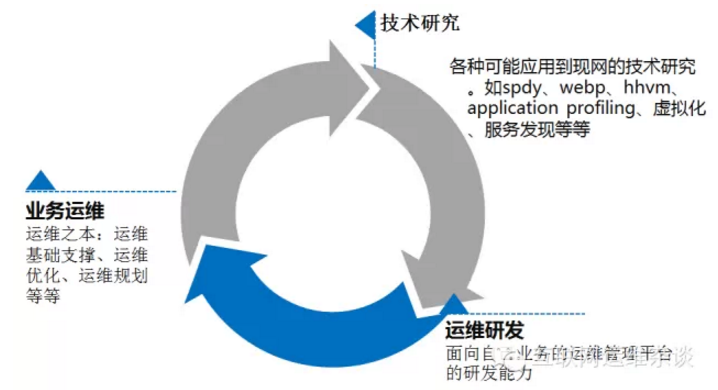
可以仿照AWS的接口实现方式，统一实现API的接口开放访问地址，同时协议统一（http、https），协议可以使用Get的方式进行访问。



**七、运维自动化依赖的团队模型**

我从能力模型、驱动模型和技能模型三个角度来阐述运维团队和个人的能力要求，最后给出一个参考的组织结构。

**第一、团队的能力模型**



在我带过的应用运维团队中，我都会从如上三个方面的能力对组员提出要求。

业务运维。在我们的考核体系中放的比重越来越低，因为这块能力要求越来越低。日常的变更、扩容、故障定位、运维规划对人的能力要求都非常的低，这些工作都能模式化且平台化，可以减少对人的倚重，

运维研发。我希望每个应用运维人都有运维研发的能力，但现实是不可能的。但对于一个应用运维团队和一个运维部门来说，运维研发的配备必不可少。在应用运维团队内部，可以让有研发能力的人迅速承担面向业务运维平台的建设或者参与到部门的运维系统建设中，可以50%时间参与研发。运维研发能力是让团队价值迅速达成的保证，没有研发能力的运维不是一个好运维（包括我自己）。

技术研究。运维是个技术团队，需要通过技术体现价值，当找到好的技术就想着如何应用到业务上，给用户带来价值，比如说用户体验提升，成本减少等等。

这个时候有个问题，应用运维团队里面的人也会运维研发，然后又有专职的运维研发团队？那他们的职责分工如何解决，在工作上是否会存在重复建设？我的回答是这样的：

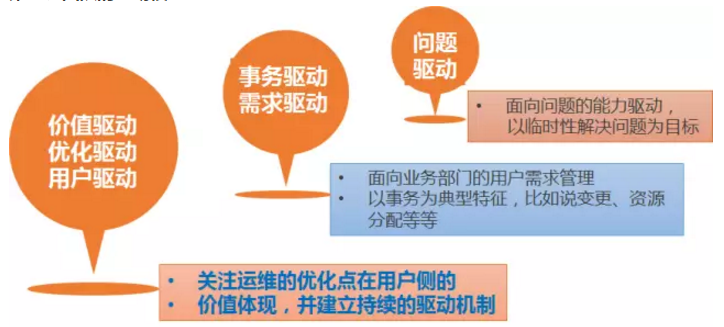
首先，可以把运维研发初期定位在公共服务平台研发上，比如说DNS、LVS、配置管理、监控系统、CMDB、数据分析平台等等。

其次，运维研发还需要制定相应的运维研发规范，代码规范、UI规范、测试规范等等，让所有参与运维研发的人统一遵守，包括应用运维研发的组员。

最后，说一下应用运维小组内的研发能力该如何发挥的问题。其实在很多运维团队中，运维都是跟随业务，一则可以让应用运维研发人员开发面向业务的运维系统，他们最懂该业务的需求，实现自己想要的；另外一种更好的操作方式，让应用运维小组内的研发人员50%时间参与到以运维研发牵头成立的虚拟研发小组中。一则可以进一步提高应用运维的研发水平；另可以提高运维研发对业务运维的理解，同时提高带队作战的能力。

关于运维研发和应用运维的比例该设置成多少？2:1吧，这也是初步拍的。大家也可以自检一下，自己的运维团队到底设置了多少运维研发人员？另外运维研发配备是否足够，可以周期性看看运维团队取得的进步，特别是效率、质量维度。

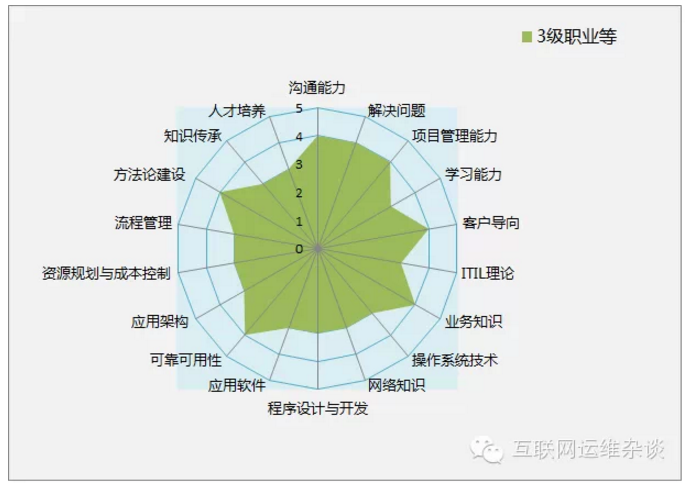
**第二、团队的驱动模型**



团队的驱动力不同，带来结果的就完全不同。为什么很多运维人员都说自己很苦逼，这个地方可以看看到底是什么在引导着你在做运维工作？传统的维护，都是集中在第一和第二阶段，而进入到高阶运维体系之后，我们需要迅速切换到价值驱动、用户驱动的维度上来。有了用户驱动和价值驱动，对运维的效率、质量都有了更高的要求，反向驱动我们必须走自动化和平台这条道路。

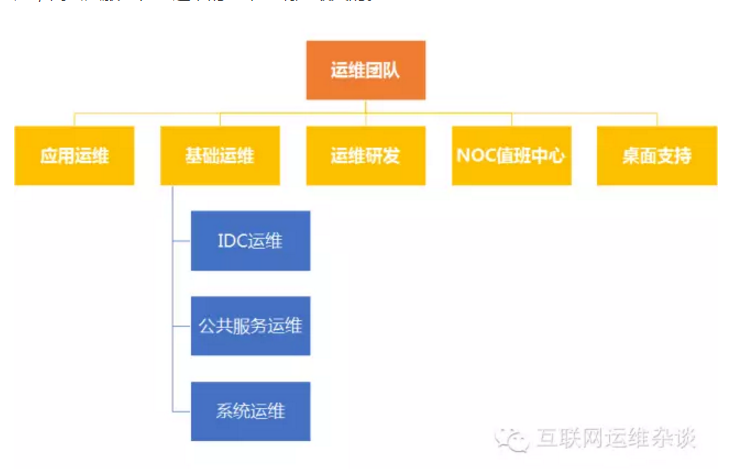
**第三、团队的技能模型**

在BAT很早就实施了职业通道体系，对于运维人员的成长有明确的要求和衡量体系，在此我就不详细介绍了。下图是腾讯的高级应用运维工程师的技能要求雷达图，供大家参考：



**第四、参考的运维团队组织结构**

不一定要按照这个结构明确设置运维小组，但是运维的职能差不多是这样。我还有另外一个建议，最好公共服务研发团队最好和运维团队放在一个组织结构下，会有利于公共化服务的推广，而公共服务化对运维的效率影响是最大的。



至此，自动化平台的深度解码已经完成。从多个层面带大家了解运维自动化，其实还是希望给大家带去一点借鉴意义。大胆的往前走吧，一切都有可能，唯独那些实现不了，都是我们人的问题，别无其他。