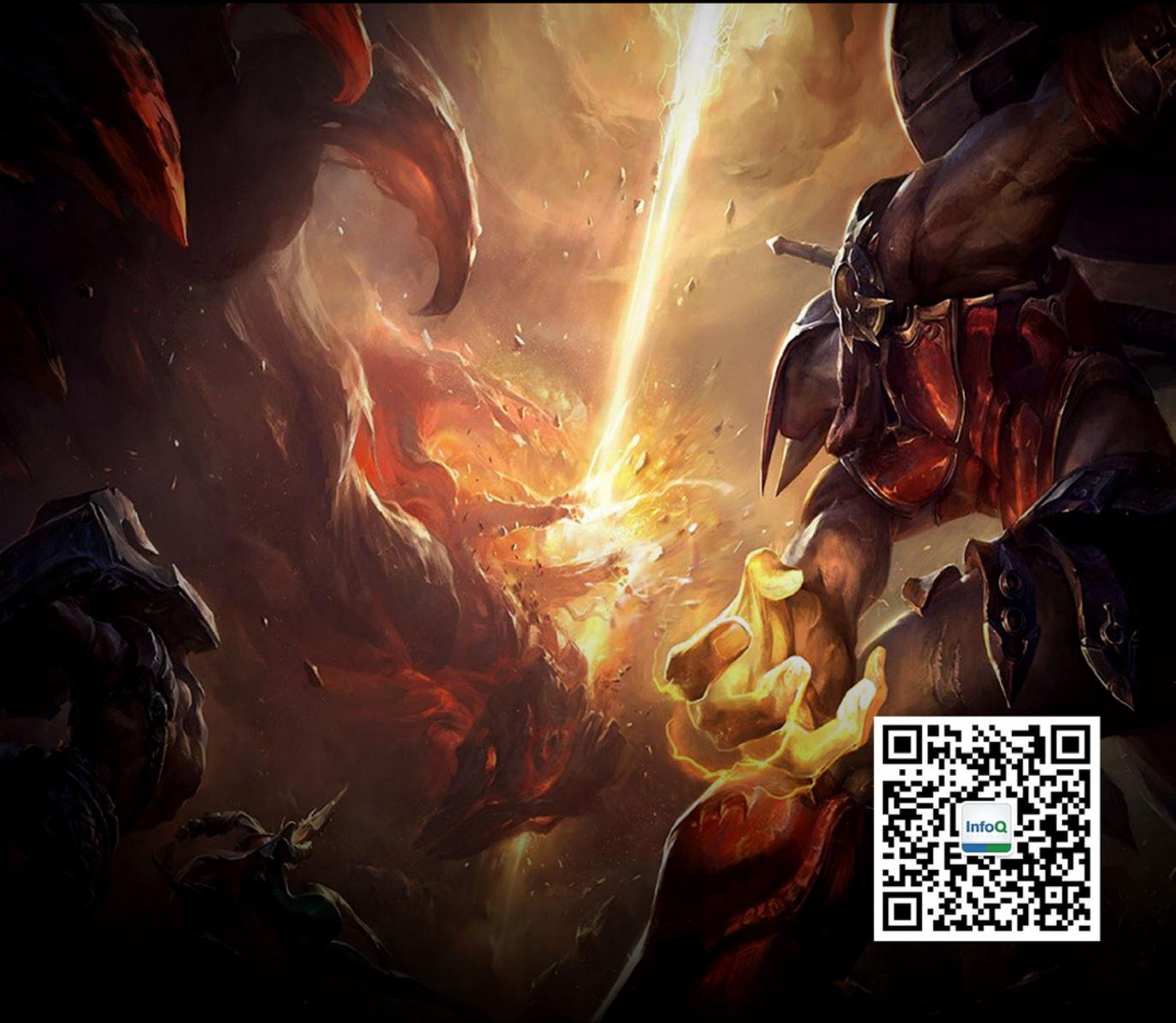




《英雄联盟》 在线服务运维之道

RUNNING ONLINE SERVICES AT RIOT

作者：RIOT / 译者：薛命灯



目录



06 第一章 简介

18 第二章 调度

26 第三章 网络（一）

33 第四章 网络（二）

39 第五章 微服务生态系统

50 第六章 开发者生态系统

聚焦最新技术热点 沉淀最优实践经验

[北京站]2018

北京·国际会议中心

演讲：2018年4月20-22日 培训：2018年4月18-19日

精彩案例 先睹为快

《Netflix的工程文化：是什么在激励着我们？》

Speaker: Katharina Probst

Netflix 工程总监

《Apache Kafka的过去，现在，和未来》

Speaker: Jun Rao

Confluent 联合创始人

《人工智能系统中的安全风险》

Speaker: 李康

360网络安全北美研究院负责人，IoT安全研究院院长

《从C#看开放对编程语言发展的影响》

Speaker: Mads Torgersen

微软 C#编程语言Program Manager

《Lavas：PWA的探索与最佳实践》

Speaker: 彭星

百度 资深前端工程师

《浅谈前端交互的基础设施的建设》

Speaker: 程劭非（寒冬）

淘宝 高级技术专家

《深入Apache Spark流计算引擎：Structured Streaming》

Speaker: 朱诗雄

Databricks软件开发工程师，Apache Spark PMC和Committer

《AI大数据时代电商攻防：AI对抗AI》

Speaker: 苏志刚

京东安全 硅谷研究中心负责人

《QUIC在手机微博中的应用实践》

Speaker: 聂永

新浪微博 技术专家

《阿基米德微服务及治理平台》

Speaker: 张晋军

京东 基础架构部服务治理组负责人，架构师

8折 优惠报名中，立减1360元
团 购 享 受 更 多 优 惠

访问官网获取更多前沿技术趋势

2018.qconbeijing.com

如有任何问题，欢迎咨询

电话：15110019061，微信：qcon-0410



卷首语

作者 薛命灯

美国拳头公司（Riot Games）成立于2006年，2009年发行《英雄联盟》游戏，2015年被腾讯全资收购，成为腾讯旗下的子公司。

几年来，《英雄联盟》获奖无数，来自世界各地的玩家促成了数千万人同时在线游戏的盛况。拳头公司把玩家的游戏体验放在第一位，他们的开发团队和基础设施团队为此做出了不可磨灭的贡献。来自拳头公司基础设施团队的工程师们分享了他们运维在线服务的发展历程，介绍了他们从手动部署到自动运维的演变过程。

拳头公司的在线服务运行环境十分复杂，他们使用了共有云和本地数据中心，这些数据中心遍布世界各地，如何快速做出服务变更，并及时将它们推送给玩家，同时又能保证不对玩家造成不好的影响，这是基础设施团队需要解决的重大难题。

任何一家公司都是从小做起的，刚开始的时候规模小，服务器也少，很多事情可以通过手动进行。但随着规模的增长，服务器越来越多，系统越来越复杂，人工干预容易出错，也跟不上变更的节奏，所以必然会走向自动化。系统也从单体变成了微服务，数据中心从一个变成多个，除了本地数据中心之外，还加入了公有云。开发、部署、测试、监控，这些环节



一个都不能少。网络的部署和管理、应用程序的部署和管理、如何进行快速而安全的变更、如何保证数据的安全、如何进行失效备援以便提高可用性，如何为玩家带来最佳的体验，有太多的问题需要解决。而拳头公司的基础设施团队又是怎么做到这些的呢？

第一章介绍了拳头公司的在线服务发展概况，简述了他们的调度系统、容器化进程、持续集成、网络管理、应用程序动态化。第二章深入介绍了他们的调度系统Admiral。第三章详细介绍了叠加网络、OpenContrail以及与Docker的集成。第四章介绍了他们是如何实现基础设施即代码的，还介绍了他们的负载均衡和失效备援策略。第五章介绍了他们的微服务生态系统，包括高度可移植性、动态配置、服务发现、全局搜索和秘钥获取等方面的内容。第六章介绍了他们的开发者生态系统，包括他们的开发者工具箱，如监控可视化工具、网络管理工具、服务搜索工具、构建跟踪工具等。

正如这些工程师所说的，拳头公司灵活而敏捷的氛围让他们可以为开发团队开发出真正有价值的工具，让他们能够专注在产品的开发上，从而把最好的游戏功能带给玩家。



第一章 简介

我是Jonathan McCaffrey，来自Riot公司的基础设施团队。在讲述我们如何进行后端应用的部署和运维之前，首先要先了解一下我们是如何看待我们的应用开发的。游戏玩家的价值对于Riot来说是至高无上的，我们的开发团队经常与玩家社区互动，为了给玩家们提供最佳的游戏体验，我们必须具备根据玩家反馈快速做出变更的能力。基础设施团队的使命就是为开发人员提供支持，我们提供支持的力度越大，玩家们就能越快体验到最新的功能。

当然，说起来容易做起来难！因为部署的多元化，我们面临着很多挑战。我们有公有云服务器，也有本地数据中心，还要与腾讯和Garena合作。这些环境的复杂性给开发团队增加了不少负担，于是就有了基础设施团队。我们使用基于容器的内部云来简化部署，这个云叫作“rCluster”。在这篇文章里，我将介绍Riot从手动部署到使用rCluster进行部署的演变过程。为了更好地描述rCluster，我将以海克斯工匠系统（Hextech Crafting System）作为例子。

历史背景简述

我在七年前加入Riot，当时我们并没有那么多部署流程和服务器管理流程，我们还只是一个怀揣大梦想的初创公司。我们的预算很有限，做什么事情都要求快。我们为《英雄联盟》构建了一套基础设施，努力满足游戏不断增长的需求，为开发团队提供支持，同时还要支持区域团队拓展新的疆域。我们通过手动配置服务器和应用程序，没有太多指南和战略规划之类的东西。

后来，我们使用[Chef](#)来完成常见的部署和基础设施任务，使用公有云进行大数据计算和提供Web服务。在这一过程中，我们多次重新设计了我们的网络，也更换过供应商，甚至重组了整个团队结构。

我们的数据中心有数千台服务器，每新增一个应用程序就要添加新的服务器。新服务器被接入手动创建的虚拟局域网，并手动配置路由和防火墙规则来提升安全性。虽然这样可以提升安全性，但费时又费力。因为这个痛点的存在，当时的新功能都被设计成小型的Web服务，导致《英雄联盟》生态系统的服务数量大肆膨胀。

除此之外，我们的开发团队对应用程序的测试环节缺乏信心，在进行部署时经常出现一些配置或网络连接问题。因为应用程序与物理基础设施的耦合太过紧密，以至于生产环境与测试环境、Staging环境和公测环境难以保持一致。

我们的应用程序数量一直在增长，我们也在经历着手动配置服务器和网络的艰难过程。与此同时，Docker开始在我们的开发团队中流行起来，用于解决配置和开发环境一致性问题。我们开始意识到，我们可以用Docker做更多的事情，它可以在基础设施方面扮演一个更重要的角色。

2016 全明星赛及其他

基础设施团队的目标是为游戏玩家、开发团队和2016全明星赛提供支持。在2015年底，我们从手动部署转向了自动部署，比如海克斯工匠系统

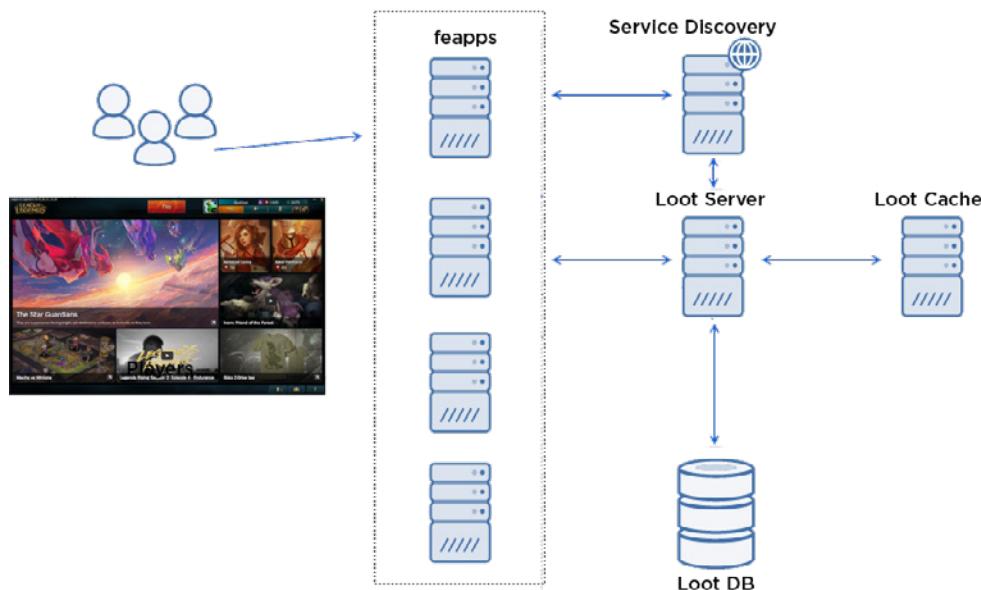
就使用了自动部署。我们开发了rCluster——一个全新的系统，它使用了Docker和微服务风格的SDN（Software Defined Networking）。rCluster解决了不一致性问题和部署流程问题，让产品团队可以集中在他们的产品研发上。

接下来我们将深入探讨rCluster是如何支持[海克斯工匠系统](#)的。

海克斯工匠系统在我们内部被称为“战利品（Loot）”，由三个核心组件组成。

- 战利品服务——一个Java应用程序，通过基于HTTP/JSON的REST API处理战利品请求。
- 战利品缓存——基于Memcached的缓存集群，并使用了一个Go语言开发的小型边车（sidecar）来监控、配置、启动和关闭缓存集群。
- 战利品数据库——一个MySQL集群，包含了一个主数据库和多个从数据库。

在进入工匠系统时，会发生以下一系列事件：



1. 玩家在客户端打开工匠系统的屏幕。

2. 客户端向前端应用程序发起RPC调用，前端应用（也就是“feapp”）就是玩家和后端服务之间的代理。

3. feapp向战利品服务器发起调用请求。

- feapp从“服务发现”（Service Discovery）中找到战利品服务器的IP地址和端口。
- feapp向战利品服务器发起HTTP GET请求。
- 战利品服务器检查战利品缓存里是否保存着玩家的物品。
- 玩家物品不在缓存里，于是战利品服务向数据库发起请求，并将返回的结果放进缓存。
- 战利品服务将结果返回给feapp。

4. feapp将RPC响应消息返回给客户端。

通过与战利品团队的合作，我们将服务器和缓存放进了Docker容器，并使用JSON文件来定义它们的部署配置。

战利品服务器的JSON配置示例：

```
{  
  "name": "euw1.loot.lootserver",  
  "service": {  
    "appname": "loot.lootserver",  
    "location": "lolriot.ams1.euw1_loot"  
  },  
  "containers": [  
    {  
      "image": "compet/lootserver",  
      "version": "0.1.10-20160511-1746",  
      "ports": []  
    }  
  ],  
  "env": [  
    "LOOT_SERVER_OPTIONS=-Dloot.regions=EUW1",  
    "LOG_FORWARDING=true"  
  ],
```

```
    "count": 12,
    "cpu": 4,
    "memory": 6144
}
```

战利品缓存的JSON配置示例：

```
{
  "name": "euw1.loot.memcached",
  "service": {
    "appname": "loot.memcached",
    "location": "lolriot.ams1.euw1_loot"
  },
  "containers": [
    {
      "name": "loot.memcached_sidecar",
      "image": "rcluster/memcached-sidecar",
      "version": "0.0.65",
      "ports": [],
      "env": [
        "LOG_FORWARDING=true",
        "RC_GROUP=loot",
        "RC_APP=memcached"
      ]
    },
    {
      "name": "loot.memcached",
      "image": "rcluster/memcached",
      "version": "0.0.65",
      "ports": [],
      "env": ["LOG_FORWARDING=true"]
    }
  ],
  "count": 12,
  "cpu": 1,
  "memory": 2048
}
```

不过要真正部署好它们，我们还需要创建一些集群，它们需要支持南美、北美、欧洲和亚洲的Docker。我们因此需要解决一大堆问题：

- 容器调度
- Docker网络
- 持续交付
- 动态运行应用程序

后续的章节将会详细介绍这些组件，所以在这里先简要提及。

容器调度

我们自己编写了一款叫作Admiral的软件，并把它用在rCluster系统里，进行容器调度。Admiral与一组物理机上的Docker后台进程进行通信，以便了解它们的健康状态。运维人员通过HTTPS发送上述的JSON请求，Admiral则用它们了解相关容器的状态。Admiral会持续地更新集群的状态，并在必要的时候采取相应的行动。Admiral会根据实际情况向Docker后台进程发起请求来启动或停止容器，从而达到预期的状态。

如果容器发生崩溃，Admiral会在另一台主机上启动一个新的容器。这种灵活的机制让服务器的管理变得十分容易，我们可以无缝地“灭掉”它们，做一些维护工作，然后再重启它们。

Admiral与开源工具[Marathon](#)有点像，我们目前也正打算使用Mesos、Marathon和DC/OS的替代方案。如果这样可行，我们将会在后续的文章中分享我们的经验。

Docker 网络

在容器可以运行了之后，我们还要为战利品应用程序和系统的其他部分提供网络连接。我们使用[OpenContrail](#)为每个应用设置私有网络，然后让开发团队使用托管在GitHub上的JSON文件来配置他们自己的网络策略。

战利品服务器网络配置示例：

```
{  
  "inbound": [  
    {  
      "source": "loot.loadbalancer:lolriot.ams1.euw1_loot",  
      "ports": [  
        "main"  
      ]  
    },  
    {  
      "source": "riot.offices:globalriot.earth.alloffices",  
      "ports": [  
        "main",  
        "jmx",  
        "jmx_rmi",  
        "bproxy"  
      ]  
    },  
    {  
      "source": "hmp.metricsd:globalriot.ams1.ams1",  
      "ports": [  
        "main",  
        "logasaurous"  
      ]  
    },  
    {  
      "source": "platform.gsm:lolriot.ams1.euw1",  
      "ports": [  
        "main"  
      ]  
    },  
    {  
      "source": "platform.feapp:lolriot.ams1.euw1",  
      "ports": [  
        "main"  
      ]  
    },  
    {  
      "source": "platform.feapp:lolriot.ams1.euw1",  
      "ports": [  
        "main"  
      ]  
    }  
  ]  
}
```

```
{  
    "source": "platform.beapp:lolriot.ams1.euw1",  
    "ports": [  
        "main"  
    ]  
,  
{  
    "source": "store.purchase:lolriot.ams1.euw1",  
    "ports": [  
        "main"  
    ]  
,  
{  
    "source": "pss.psstool:lolriot.ams1.euw1",  
    "ports": [  
        "main"  
    ]  
,  
{  
    "source": "championmastery.server:lolriot.ams1.euw1",  
    "ports": [  
        "main"  
    ]  
,  
{  
    "source": "rama.server:lolriot.ams1.euw1",  
    "ports": [  
        "main"  
    ]  
}  
,  
]  
,  
"ports": {  
    "bproxy": [  
        "1301"  
    ],  
    "jmx": [  
        "1301"  
    ]  
}
```

```

        "23050"
    ],
    "jmx_rmi": [
        "23051"
    ],
    "logasaurous": [
        "1065"
    ],
    "main": [
        "22050"
    ]
}
}

```

战利品缓存网络配置示例：

```

{
    "inbound": [
        {
            "source": "loot.lootserver:lolriot.ams1.euw1_loot",
            "ports": [
                "memcached"
            ]
        },
        {
            "source": "riot.offices:globalriot.earth.alloffices",
            "ports": [
                "sidecar",
                "memcached",
                "bproxy"
            ]
        },
        {
            "source": "hmp.metricsd:globalriot.ams1.ams1",
            "ports": [
                "sidecar"
            ]
        }
    ]
}

```

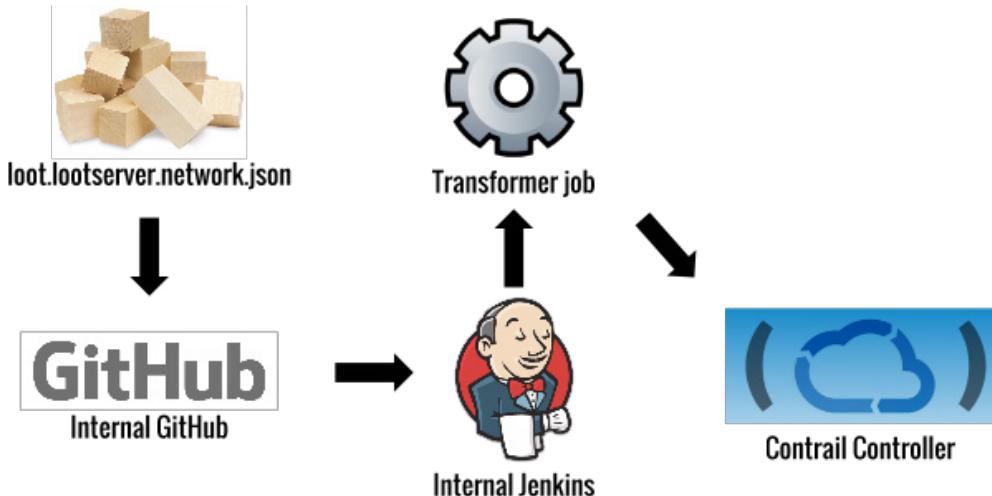
```

},
{
    "source": "riot.las1build:globalriot.las1.
buildfarm",
    "ports": [
        "sidecar"
    ]
}
],
"ports": {
    "sidecar": 8080,
    "memcached": 11211,
    "bproxy": 1301
}
}
}

```

一旦GitHub上的配置文件发生变更，就会启动一个传输作业，调用Contrail的API来创建和更新应用程序的私有网络策略。

Contrail使用了一种叫作叠加网络（Overlay Network）的技术来实现私有网络。在我们的系统里，Contrail在计算机主机之间启用了[GRE](#)通道，并使用网关路由器来管理流经通道的流量。OpenContrail系统的灵感来自



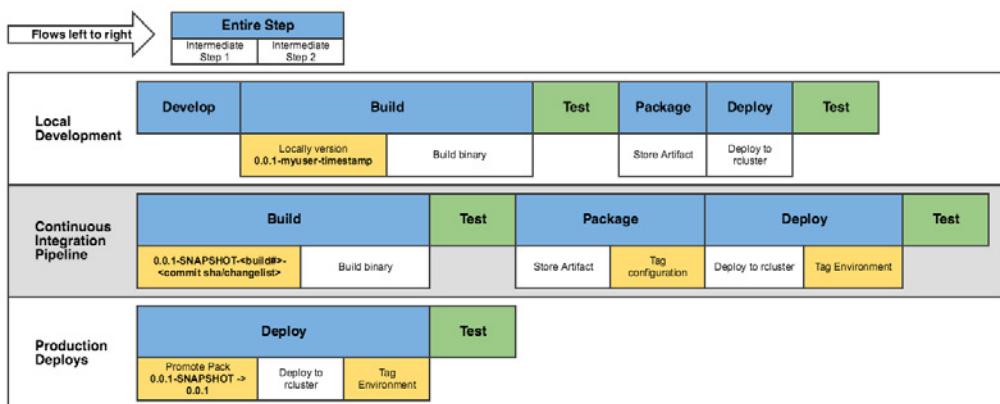
于标准MPLS L3VPN，所以在概念上与之非常相似。

在实现这个系统时，我们必须解决一些关键性挑战：

- 集成Contrail和Docker；
- 允许rCluster之外的网络无缝地访问叠加网络；
- 允许不同集群之间的应用发生交互；
- 在AWS上运行叠加网络；
- 在叠加网络上构建高可用的面向边缘的应用。

持续集成

战利品应用程序的CI流程类似下面这样：



我们的主要目标是这样的：一旦主仓库发生变化，就会创建一个新的应用容器，并将其部署到QA环境中。有了这个流程，我们的团队就可以快速迭代他们的代码，并看到代码实际的运行情况。紧凑的反馈闭环加快了改进用户体验的速度，这也正是Riot的关键目标——以玩家为中心，为玩家提供最佳体验。

动态地运行应用程序

到目前为止，我们谈论了我们是如何构建和部署应用程序的，但如果你曾经在这样的容器环境里工作过的话，你就会知道，我们要面临的挑战远不止之前提到的那些。

在rCluster里，容器有动态的IP地址，而且一直在启动和关闭。这与之

前提到的静态服务器的部署方式是完全不一样的，所以我们需要新的工具和流程。

一些关键问题：

- 如何监控容量和端点一直在变化的应用？
- 如果端点一直在变化，那么应用程序如何才能知道其他应用程序的端点是什么？
- 如果不能通过ssh连接到容器上，而且容器日志在重启之后就会消失，那么该如何进行问题诊断？
- 如果我们是在构建阶段预热（bake）容器，那么如何配置数据库密码，或者如何为其他地区的容器配置不同的参数？

为了解决这些问题，我们开发了一个微服务平台，用于解决服务发现、配置管理和监控问题，我们将在其他详述该平台的细节，以及它为我们解决了哪些问题。



第二章 调度

在第一章中，我们介绍了Riot应用部署的发展简史以及我们曾经面临的挑战。我们特别强调了在发展过程中遇到的困难，为了更好地支持《英雄联盟》，我们加入越来越多的基础设施，而手动分配服务器的方式给我们带来了很大麻烦。Docker的出现改变了服务器的部署方式，我们开发了内部工具Admiral，用于集群的调度和管理。

要知道，这个旅程尚未结束，它还在不断演化，我们后续还准备使用DC/OS。这一章将介绍我们是如何发展到今天这个地步的，以及在这一过程中我们所做的重要决定，希望对读者有所裨益。

什么是调度以及为什么要有调度

随着Docker的问世以及Linux容器化技术的普及，我们意识到对基础设施进行容器化一定能够给我们带来好处。Docker容器镜像提供了一种不可变的可部署文件，一次构建就可以用在开发、测试和生产环境里。除此之外，我们还能确保生产环境的镜像与测试环境的镜像是一样的。

Docker还为我们带来了另一个好处：Docker通过调度器将容器分配给主机，从而解耦了部署单元（容器）和计算单元（主机）。也就是说，服务器和应用程序之间的耦合性没有了，容器可以运行在任意台服务器上。

后端的服务被打包进了Docker镜像里，我们可以在任何时候将它们部署到多台服务器上，因此能够快速做出变更。我们可以开发新的功能，对流量较大的功能进行扩展，快速地进行更新和问题修复。不过，要在生产环境中使用容器进行应用部署，需要解决三个问题：

1. 如何从主机集群中选择适量的服务器来部署容器？
2. 如何远程启动这些容器？
3. 如果容器挂掉了该怎么办？

答案是我们需要一个调度器——一个运行在集群上的服务，执行我们设定好的容器策略。调度器是一个非常重要的组件，它可以确保容器在正确的地方运行，一旦发生崩溃立即将其重启。比如，我们的海克斯工匠系统需要六个容器实例，调度器负责找出拥有足够内存和CPU的主机来运行这些容器，并看管好它们。如果其中的一台服务器发生宕机，调度器需要负责找到另一台替代主机，确保容器可以继续运行。

在考虑使用调度器时，我们要求调度器能够帮助我们快速地创建原型，及早知道容器化的服务是否能够在我们的生产环境中正常运行。另外，我们还要确保现有的开源产品能够适用于我们的环境，而且维护人员愿意使用它们。

为什么要开发自己的调度器

在开发我们自己的调度器Admiral之前，我们对现有的集群管理器和调度器进行了调研。

在调研过程中，我们研究了一些开源项目。

Mesos和Marathon

- 这些框架很成熟，而且支持大规模集群，但它们太过复杂，安装起来也很取巧，所以用户体验并不是很好。

- 那个时候，它们支持的容器很有限，无法跟上Docker的发展步伐，在Docker生态系统里并没有那么耀眼。
- 它们不支持容器群组（pod），但我们需要这样的功能，因为我们要为很多服务绑定边车（sidecar）。

从LMCTFY到Kubernetes

- Kubernetes从LMCTFY发展而来，虽然它看起来雄心勃勃，但我们不确定它未来的演化是否能够满足我们的需求。
- Kubernetes当时还没有一个能够满足我们需求的约束系统。

Fleet

- Fleet当时才刚刚开源，不像现在这么成熟。
- 它似乎更适合用于部署系统服务，而不是用于部署一般的应用服务。

我们也尝试开发了一个命令行小工具，通过REST与Docker API发生交互，并成功地使用该工具进行部署编配。于是我们决定继续使用自己的调度器，我们从调研过的框架里借鉴了一些经验，比如Kubernetes的Pod概念和Marathon的约束系统。我们的目标是搞清楚这些系统的架构和功能，并在未来某个时刻把这些功能汇聚在一起。

Admiral 概览

我们先是开发了一种叫作CUDL（CIUster Description Language，集群描述语言）的元数据语言，它是基于JSON的，用于基础部署。后来，我们开发了Admiral。Admiral将CUDL用在它的REST API当中。CUDL包含了两个主要的组件：

- Cluster——一系列Docker主机。
- Pack——用于启动容器的元数据，类似Kubernetes的Pod和复制控制器。

Cluster和Pack包含了Spec和Live，它们分别代表了容器生命周期的不同阶段。

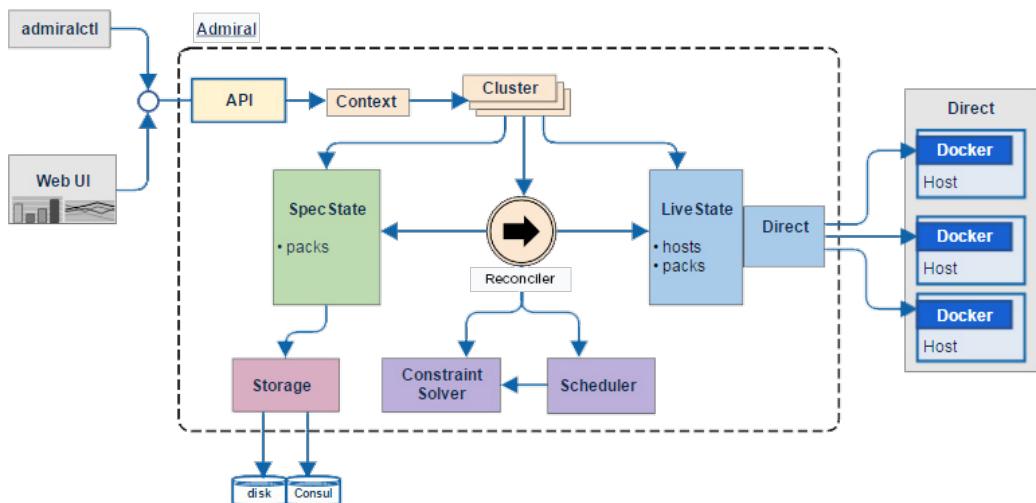
Spec代表了元素的预期状态：

- 从外部源（如源码控制系统）发送到Admiral上。
- 发送到Admiral后就不可变。
- Spec Cluster和Host描述了集群中可用的资源。
- Spec Pack描述了运行一个服务所需的资源、约束和元数据。

Live代表了元素的实际状态：

- 反映实际的运行对象。
 - Live Cluster和Host反映了运行中的Docker后台进程。
 - Live Pack反映了运行中的Docker容器群组。
- 通过与Docker后台进程交互来恢复状态。

Admiral使用Go语言开发，在部署到生产环境时被编译并打包到一个Docker容器里。Admiral包含了多个内部子系统，如下图所示。



用户使用基于REST API的命令行工具admiractl与Admiral展开交互。用户可以通过这个工具访问到Admiral的所有功能：POST新的Spec Pack进行调度、DELETE旧的Pack，以及GET当前的状态。

在生产环境，Admiral使用[Consul](#)来保存Spec状态，并定期备份，以备不时之需。如果发生数据丢失，Admiral可以使用从Docker后台进程获得的Live信息来重建部分Spec状态。

调节器（reconciler）是Admiral的核心子系统，是驱动调度工作流的关键组件。调节器定时对实际的Live状态和预期的Spec状态进行比对，如果出现差异，就会调度相应的措施让Live状态恢复正常。

Live状态和它的驱动包为调节器提供支撑，它们缓存Live主机和容器状态，为集群主机上的所有Docker后台进程提供基于REST API的交互。

深度调度

Admiral的调节器直接操作Spec Pack，再将它们转成Live Pack。Spec Pack被提交到Admiral之后，调节器就可以创建容器，并使用后台进程运行容器。调节器就是通过这种机制实现了前面提到的两个高级调度目标。调节器在收到一个Spec Pack时会做以下几件事情：

1. 评估集群资源和Pack的约束，为容器寻找合适的主机。
2. 使用Spec数据在远程主机上启动容器。

我们举一个在Docker主机上启动容器的例子。我将会使用本地的Docker后台进程作为Docker主机，并让它与本地的Admiral实例发生交互。

首先，我们使用`admiral pack create <cluster name> <pack file>`命令启动一个Pack。这个命令会向Admiral提交Spec Pack。

```
Dallen@KALLAHL3:~]
$ admirl -A https://localhost:8080 pack create blog ~/admirl/blog.pack.json
name: dot.blog.scout
description: A pack of the Scout service for usage in our engineering blog post.
constraints: null
owner: ""
accounting: ""
service:
  id: blog
  dev.local.test
  discovery:
    serviceUrls: []
    serviceIds: ""
  options: dot.scout
  version: ""
  constraints: []
  name: ""
  image: data/scout
  version: 1.0.0
  ports:
    - internal: 8080
      external: 8080
      protocol: ""
    mounts: []
    volumesFrom: []
    env: []
    envs: []
    volumes: []
    envs: []
  count: 1
  cpu: 100
  memory: 1G
  constraints: []
  labels: []
  requiresSecret: false

Dallen@KALLAHL3:~]
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
d404cc7f42d6        registry.rcluster.io/data/scout:1.0.0   "/bin/sh -c /app/scout"   2 seconds ago     Up 2 seconds       0.0.0.0:8080->8080/tcp   rc-dot.blog.scout

Dallen@KALLAHL3:~]
```

这个命令会启动一个容器，这个容器使用了Pack文件里指定的参数。

```
{
    "name": "dat.blog_scout",
    "description": "A pack of the Scout service for usage in
our engineering blog post.",
    "service": {
        "location": "dev.local.test",
        "discovery": {},
        "appname": "dat.scout"
    },
    "containers": [
        {
            "image": "datd/scout",
            "version": "1.0.0",
            "ports": [{{
                "internal": 8080,
                "external": 8080
            }}]
        }
    ],
    "count": 1
}
```

接下来，在调用`admiral pack create`之后，我们可以用`show`命令来查看由Admiral创建的Live Pack: `admiral pack show <cluster name> <pack name>`。

最后，我们通过向容器里的服务发起请求来验证我们的Pack是否正确。`admiral pack show`命令为我们返回了一些信息，然后我们使用curl向服务发起请求。

在Admiral里，调节器一直处于运行状态，确保集群的Live状态和预期的Spec状态相匹配。如果某个容器发生崩溃，或者整个服务器因硬件问题导致不可用，我们都能够及时发现。调节器可以确保玩家们不会感觉到被

```

$ admiral -a https://localhost:8080 pack show blog dot.blog.scout
name: dot.blog.scout
status: running
requestedCount: 1
runningCount: 1
availableCount: 0
podset:
- name: dot.blog.scout
  hosts:
    - id: dot.blog.scout-465e981a
      hostName: 192.168.99.100
      ipAddress: 192.168.99.100
      created: 2016-08-11T14:30:35.990151152-07:00
      lastUpdate: 2016-08-11T14:30:35.990151152-07:00
      containerLogs:
        - id: dot.blog.scout-465e981a-scout
          name: scout
          hostName: dot.blog.scout
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: tcp
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_ENVIRONMENT-dev
          name: RC_ENVIRONMENT-dev
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_DATACENTER-local
          name: RC_DATACENTER-local
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_SHARD-test
          name: RC_SHARD-test
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_CONTAINER_ID-dot.blog.scout-465e981a-scout
          name: RC_CONTAINER_ID-dot.blog.scout-465e981a-scout
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_PORTER
          name: RC_PORTER
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_PORT_B_INTERNAL-8080
          name: RC_PORT_B_INTERNAL-8080
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_PORT_B_LOCAL-8080
          name: RC_PORT_B_LOCAL-8080
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_PORT_B_REMOTE-8080
          name: RC_PORT_B_REMOTE-8080
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_CLUSTER-blog
          name: RC_CLUSTER-blog
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
        - id: RC_CLUSTER-blog
          name: RC_CLUSTER-blog
          hostName: dot.blog.scout-465e981a
          tag: 1.0.0
          dockerId: 44094c2f42a63d44905686056e7782f77ffa5f6867cbeb9b4f1513bd45b
          ports: null
          ports:
            - internal: 8080
              external: 8080
              protocol: http
            - internal: 80
              external: 80
              protocol: http
          finishedAt: ""
          networkContainer: false
      status: running
      errorLogs: ""
      ip: 192.168.99.100
      - RC_CLUSTER-blog
    
```

```

Dollar@KALLASM3:~]
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
sf6338e05990        registry.rccluster.io/dot/blog/scout:1.0.0   "/bin/sh -c /app/scout"   About a minute ago   Up About a minute   0.0.0.0:8080->8080/tcp   rc-dot.blog.scout-ds0dc07-scout

Dollar@KALLASM3:~]
$ docker kill $sf
$sf

Dollar@KALLASM3:~]
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
$sf2aff4ff6         registry.rccluster.io/dot/blog/scout:1.0.0   "/bin/sh -c /app/scout"   1 seconds ago     Up 1 seconds       0.0.0.0:8080->8080/tcp   rc-dot.blog.scout-e21ced8b-scout

Dollar@KALLASM3:~]
$ []

```

中断。这就解决了我们之前提到的第三个和第四个问题：如果一个容器突然退出，我们可以快速恢复，将影响降到最低。

下面列出了通过`admiral pack create`命令启动的容器。然后我将容器停掉，几秒钟之后，调节器会启动一个新的容器（拥有不同的ID），因为调节器知道Live状态和预期的Spec状态不一致。

资源和约束

为了更好地分配容器，调度器必须对集群了如指掌。有两个关键组件可用于解决这个问题：

- 资源（Resource）——表示服务器可用的资源，包括内存、CPU、IO和网络。
- 约束（Constraint）——Pack中包含的一系列条件，让调度器知道该将Pack放在哪里。例如，我们有可能将Pack实例放置在以下几个位置：
 - 放在集群的每台主机上。
 - 放在某台叫作“myhost.riotgames.com”的主机上。
 - 放在集群里打了标签的区域。

通过定义主机的资源，我们让调度器具备了足够的灵活性来决定将容器放置在哪台服务器上。通过在Pack中定义约束，我们可以对调度器做出限制，强制将某些模式应用在集群上。

Admiral是Riot部署技术演化当中最为关键的一个组件。借助Docker和调度系统的强大力量，我们可以更快地将后端新功能交付给玩家。



第三章 网络（一）

在这一章，我们将介绍我们的SDN、SDN与Docker的集成以及我们的新基础设施架构。

在第一章中，Jonathan提到了我们在推出新功能时需要面临网络方面的挑战，它们并不像在服务器上安装代码然后按下启动按钮那么简单。

新功能需要网络基础设施的支持：

- 连接性：访问服务的低延迟和高吞吐量；
- 安全性：控制未授权的访问和免受DoS攻击，如果受到攻击，尽量降低受影响的范围；
- 数据包服务：负载均衡、网络地址转换（NAT）、虚拟私有网络（VPN）、连接性和多路广播。

搭建网络服务是网络工程师的拿手好戏，他们登录到网络设备上，输入一些命令就把事情给办了。搭建网络服务要求工程师对网络相关配置有深入的了解，在发生问题时知道该如何应对。因为存在多个数据中心，情

况变得更加复杂，不同数据中心的同一种任务对于网络工程师来说是完全不一样的。

数据中心的网络基础设施变更成为推出新服务的瓶颈所在。所幸的是，在Riot，为玩家推出的服务出现任何问题都会立即引起大家的注意。rCluster平台致力于解决这个瓶颈，接下来我们将深入介绍几个关键组件：叠加网络、OpenContrail以及我们是如何与Docker集成的。

SDN 和叠加网络

不同的人对SDN有不同的看法。有些人认为SDN就是通过软件来定义网络配置，但在Riot，SDN意味着网络特性是可以通过API来编程的。

因为网络是可编程的，我们能够以自动化的方式在我们的网络中进行快速的部署。我们只需要一个命令就可以完成部署，完成一个网络变更从几天变成了几分钟，这样我们就有更多时间去做其他事情。

虽然网络设备已经是可编程的，但这些设备的编程接口一直在发生变化，没有一个单独的标准来约束各个设备厂商。所以，要为各种设备接口编写自动化测试就变成一项艰巨的任务。我们意识到，在硬件层之上构建一个统一的API抽象层对于Riot的网络配置弹性管理来说是至关重要的。于是，我们使用了叠加网络。

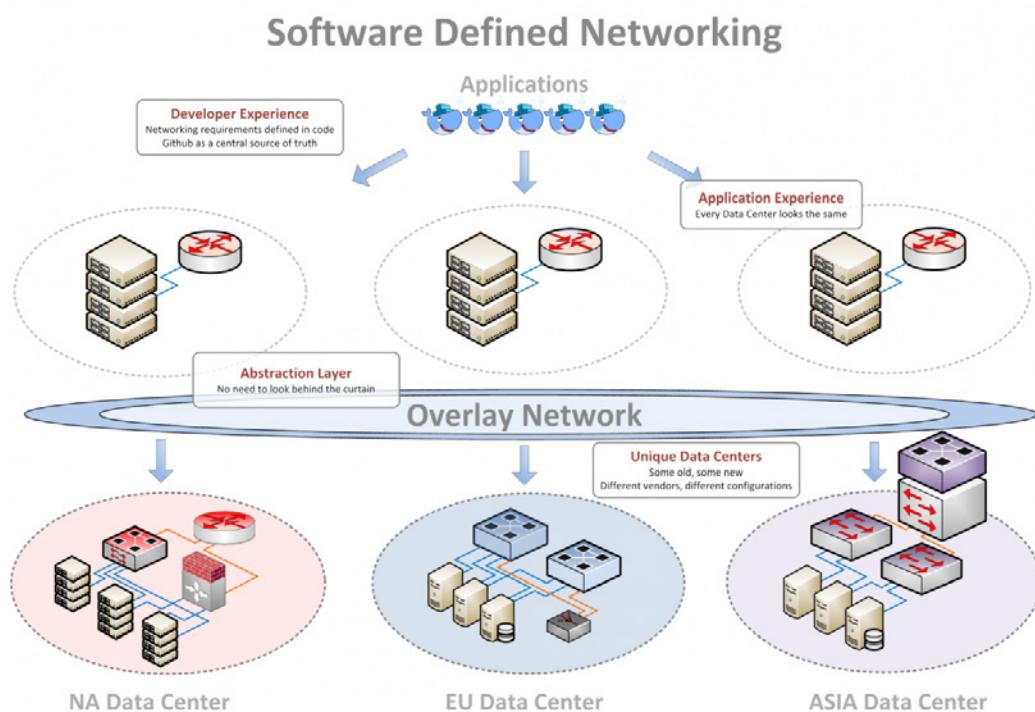
叠加网络处于已有的网络之上，运行在叠加网络中的应用程序并不知道叠加网络的存在，因为它们会认为自己是运行在一个物理网络中。如果你们熟悉虚拟机，就会知道这就有点像“物理机中的虚拟机”。一个物理网络可以托管多个虚拟网络。在一个虚拟机里，应用程序会认为自己是运行在一个物理机上，但实际上它们只拥有整个物理机的一小部分。叠加网络的概念和这个有点类似——一个物理基础设施（也就是底层网络）中存在多个虚拟网络。

我们因此可以隐藏掉物理网络的各种细枝末节，Riot的工程师们本来就不需要了解这些细节。工程师们不会再问类似这样的问题：“它有多少个端口”、“我们的供应商是谁”或者“应该在哪里设置安全策略”。我

们为工程师提供了一组统一的API。Riot的各个数据中心在有了这套统一的API之后，我们就可以在任何时候、任何地方实现自动化。我们也可以使用其他云服务供应商，比如亚马逊、Rackspace、Google Compute等，我们的API仍然可以发挥它的作用。底层的物理网络硬件有可能是思科、Juniper、Arista、戴尔、D-Link、白盒的、黑盒的、一系列承载10G端口的Linux服务器，这些都不是问题。底层网络必须使用特定的方式（比如使用自动化配置模板）进行构建，这样我们就可以解开底层物理网络配置和服务配置之间的耦合。

维护底层网络也有一定的好处。关注点的分离让底层网络更自由了，它只需专注提供高可用的数据包传输，我们在升级物理网络时无需担心会影响到上层的应用程序。这简化了我们的运维工作，我们可以自由地在各个数据中心之间切换服务，而且避免了厂商锁定。

简而言之，我们认为叠加网络是个好东西。



OpenContrail

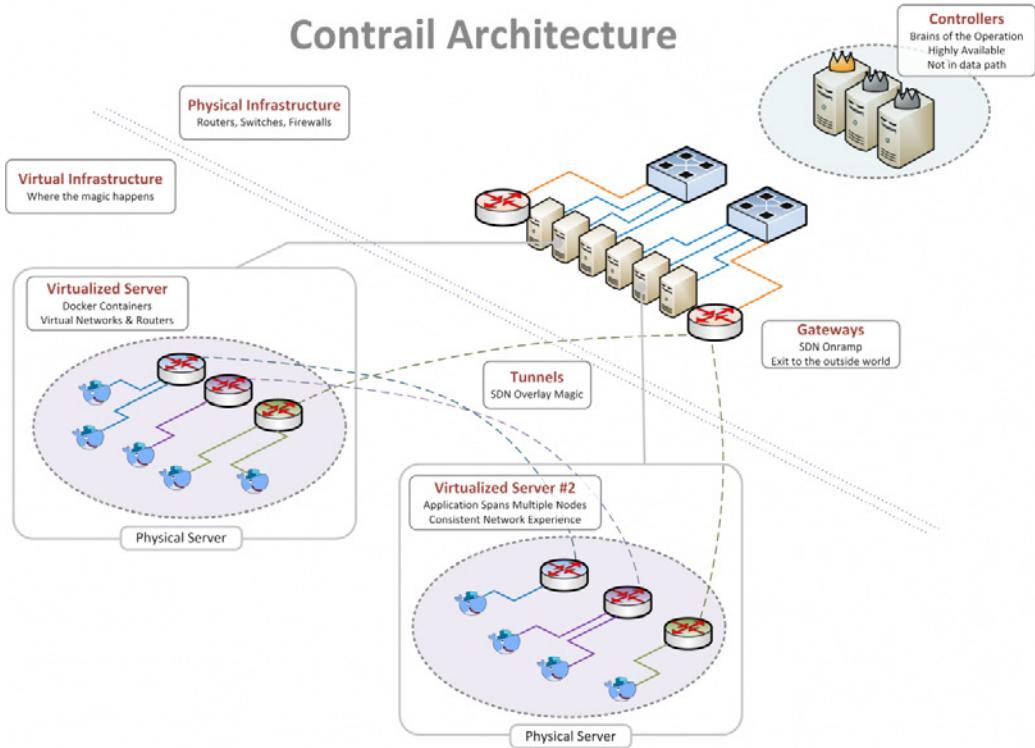
当初在评估SDN时，我们也调研过其他各种产品。它们有些通过集中控制器来配置物理网络，有些则更像是一个抽象层，将API调用转成与特定厂商相关的命令。有些解决方案需要依赖新的硬件，有些则可以运行在现有的基础设施上。有些是由大公司开发的，有些是开源项目，也有一些来自初创公司。简而言之，我们花了大量时间在调研上，要做出选择真的很难。以下是我们的主要需求点：

- 可以支持我们的新旧数据中心、裸机和云。
- 稳定的开源项目，不会突然间停止更新。
- 有专业人士能够提供帮助。

最后我们把目光锁定在了Juniper Networks的OpenContrail上。他们推出OpenContrail的初衷就是要将它作为一个开源项目，并且与厂商无关，适用于任何一种网络。它的核心是BGP和MPLS，这两种为人们所熟知的协议已经被证实可以处理整个互联网的规模。Juniper当然不可能在瞬间就消失，而且在我们设计和安装第一个集群过程中，他们为我们提供了很多帮助。

Contrail由三个主要的组件组成：中心控制器（“大脑”）、vRouter（虚拟路由器）和外部网关。每个组件都是一个高可用的集群，所以个别设备的故障并不会影响到整个系统。调用中心控制器的API会将所有变更散播到vRouter和网关上，然后再被转发到网络中。

叠加网络包含了一系列通道，这些通道从一个vRouter跨越到另一个vRouter。当一个容器要与另一个容器发生交互时，vRouter首会先检查这个容器的位置在哪里，然后在计算节点之间建立通道。接收端的vRouter检查流入的数据包，验证它们的合法性，然后再转发到目的地。如果容器要与非叠加网络中的实体发生交互，我们就会把数据包发到外部网关。外部网关将通道移除，并把数据包发送到外部网络上，保持容器IP原封不动。



集成 Docker

如果我们无法在叠加网络中运行容器，那么之前所提到的一切都只是一种臆想。Contrail是一款虚拟SDN产品，它需要与容器编配器集成在一起，才能达到调度计算实例的目的。Contrail已经通过Neutron API与OpenStack进行了集成，不过我们使用了自己的编配器Admiral，所以我们也需要进行自定义集成。另外，Contrail与OpenStack的集成最初是为虚拟机而设计的，而我们现在需要把它应用在Docker容器上。我们与Juniper合作，开发了一个叫作“Ensign”的服务，它运行在每一台主机上，负责集成Admiral、Docker和Contrail。

为了解释我们如何将Docker与Contrail集成在一起，需要先了解一下Linux网络。Docker使用了Linux内核的网络命名空间来隔离容器，防止它们互相访问。网络命名空间实际上是一个独立的网络技术栈——网络接口、路由表、iptables规则。一个网络命名空间中的这些元素只会被应用于

在该命名空间中启动的进程。这与文件系统的chroot有点类似，只不过它指的是网络。

在开始使用Docker时，我们有四种方式来配置网络命名空间。

- 宿主网络模式：Docker将进程置于宿主网络命名空间中，是完全不隔离的。
- 桥接网络模式：Docker创建了一个Linux桥，将容器网络命名空间连接到宿主网络中，同时管理着从宿主网络到容器的NAT iptable规则。
- From网络模式：Docker使用另一个容器的网络命名空间。
- 无网络模式：Docker创建一个没有接口的网络命名空间，也就是说，进程无法连接到命名空间外部。

无网络模式通常用于第三方网络集成，与我们的场景非常契合。在启动容器之后，第三方实体可以将必要的组件安插到网络命名空间中，将容器连接到网络。不过，这里有一个问题：容器需要事先启动，而且在刚开始的一段时间内是没有网络的。这对于应用程序来说不是什么好事，因为它们都想知道容器在启动时的IP地址是什么。虽然Riot的开发人员可以通过重试的方式解决这个问题，但我们不想给他们添加太多麻烦。另外，很多第三方容器无法处理该问题，我们也无法改变他们。所以，我们需要一个全盘的解决方案。

为了解决这个问题，我们借鉴了Kubernetes的一些想法，引入了"网络"容器的概念，"网络"容器会在主容器之前启动。我们以无网络模式启动该容器，在分配到IP地址之后再启动主应用容器，然后使用From网络模式将主应用容器连接到网络容器的网络命名空间中。这样一来，应用程序在启动时就有了可运行的网络栈。

在物理主机上启动一个新容器时，vRouter会为它分配一个虚拟NIC、一个全局唯一的IP地址以及路由和安全策略。这与默认的Docker网络配置非常不一样，因为同一台主机上的容器共享同一个IP地址，而其它们之间可以自由地发生交互。这有悖于我们的安全策略，因为我们要求应用程序

之间是不能发生交互的。为每个容器分配单独的IP地址简化了我们的网络配置和安全策略，同时避免了容器共享IP地址给我们带来的复杂性。

我们的SDN和基础设施自动化之路并不平坦，而在我们前面仍然有一段很长的路要走。我们已经学会了如何构建自助服务网络、如何调试叠加网络的连接问题、如何应对新故障。我们跨越两代网络架构部署了SDN，并将它与好几个“遗留”数据中心集成在一起。我们花了很多精力进行自动化，学会如何确保系统的可信度。Riot的工程师现在每天基于这个自助工作流来开发、测试和部署他们的服务，它俨然已经成为他们开发工具箱中必不可少的工具。



第四章 网络（二）

在上一章中，我们讨论了rCluster的网络问题，包括叠加网络的概念、如何使用OpenContrail实现叠加网络以及如何与Docker集成。在这一章中，我们将深入探讨其他话题：基础设施即代码、负载均衡和失效备援测试。

基础设施即代码

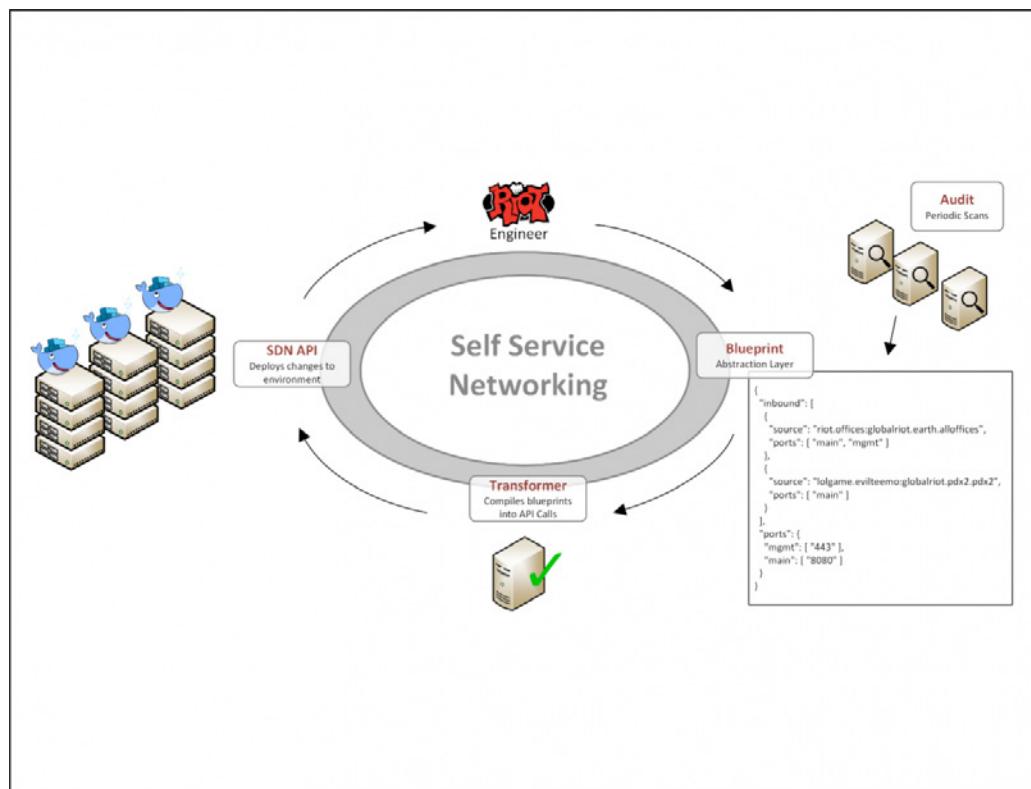
OpenContrail为我们提供了一组用于配置网络的API，帮助我们实现了应用程序网络的自动化。我们使用持续交付的方式来发布应用程序，也就是说，任何一个提交到主干分支的代码都有可能成为一个潜在的发布版本。为了支持持续交付，应用程序必须经过严格的自动化测试，我们还要提供一个自动化的构建和部署管道。部署过程必须是可重复的，在发生问题时可以进行回退。但问题是，应用程序的功能并非只有代码，还包括它们所运行的环境。

为了实现可重复的构建和部署，我们需要给应用程序和它的运行环

境打上版本标签，并对它们进行审计（这样我们就知道谁对它们做了修改）。也就是说，我们不仅要对应用程序的代码进行版本控制，对应用程序运行环境的描述也需要被包含在版本控制当中。

我们构建了一个系统，用于描述应用程序的网络环境，我们使用了一个简单的JSON数据模型来描述网络，并称之为网络蓝图（network blueprint）。然后，我们创建了一个循环作业，从代码版本控制系统上拉取这些蓝图，把它们转换成Contrail API调用。开发人员使用这个数据模型来定义他们的网络需求，比如应用程序之前的交互方式。他们不需要关心IP地址或其他任何本该由网络工程师掌控的细节问题。

开发人员有自己的网络蓝图，要修改它们只需要发起一个拉取请求，修改完毕之后再合并到主干分支上。有了这种自助流程，修改网络配置不再成为瓶颈，因为我们不再依赖那一小撮网络工程师。现在唯一的瓶颈在于开发人员能够在多少时间内修改和提交JSON配置文件。

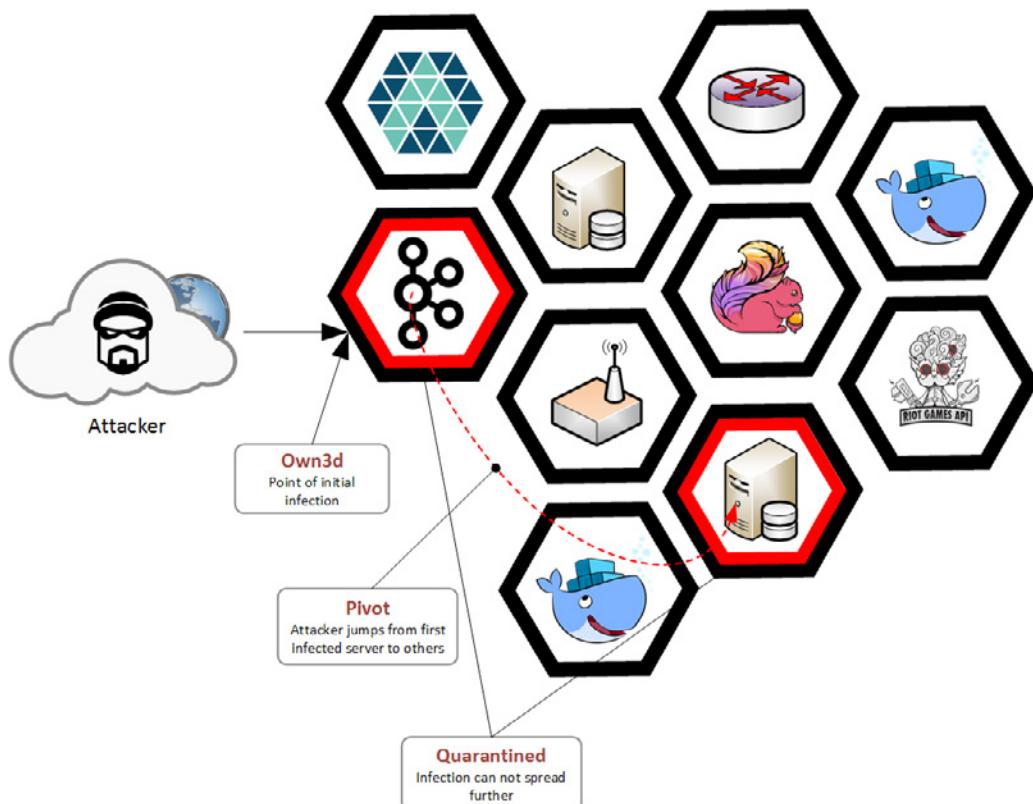


有了这个系统，我们就可以很快地开启必要的网络访问权限。对于

Riot来说，玩家的安全是最重要的，我们要在基础设施中采取安全措施。我们的安全策略有两个原则，即最少权限原则和深层防御原则。

最少权限的意思是说，Riot网络的任何一个访问者都只具备访问满足他们工作所必需资源的权限。访问者有可能是人，也可能是后端的服务。这个原则极大地降低了潜在的入侵可能带来的风险。

深度防御的意思是说，我们同时在基础设施的多个地方部署了安全防控措施。就算攻击者突破了一个安全点，后面还有更多的安全点等着他们。例如，公共Web服务器对支付系统的访问受到了严格限制，而支付系统也有自己的防御手段，比如使用了Layer 7防火墙和入侵检测系统。



Contrail通过vRouter给每一台主机设置了一个安全点。基于基础设施即代码的JSON描述，我们就有了最新的网路安全策略。我们还开发了用于扫描网络规则的工具。速度、强壮的安全实践、审计能力——这些组合在一起形成了一个健壮的安全系统，它不仅不会给开发人员带来麻烦，反

而让他们能够更快更方便地完成他们的工作。

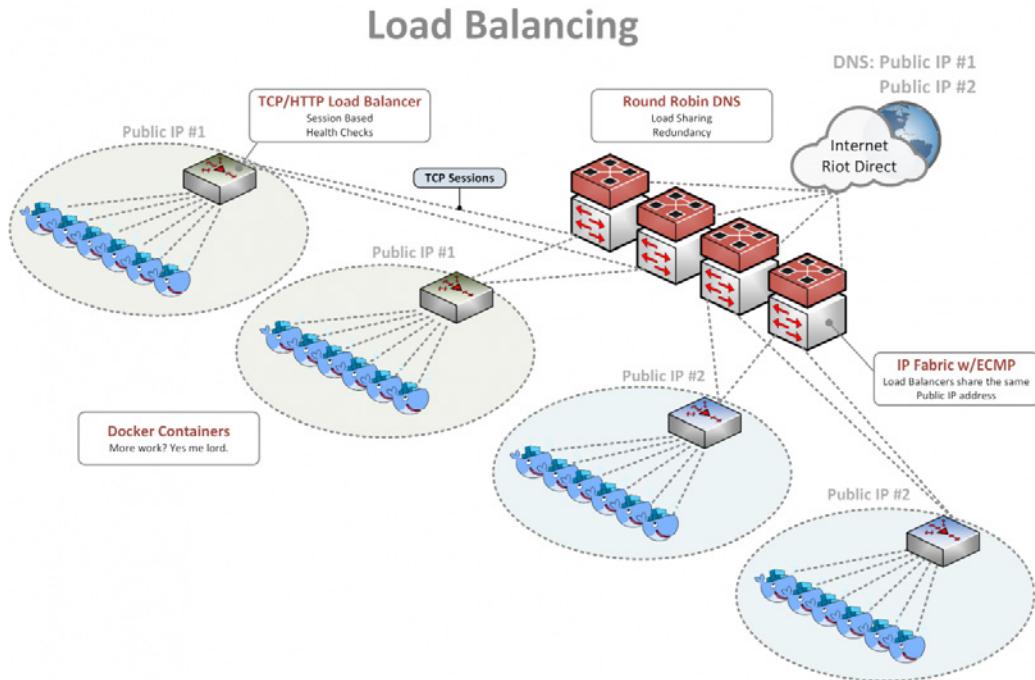
负载均衡

为了满足应用程序不断增长的需求，我们基于DNS、Equal Cost Multi-Pathing（ECMP）和传统的基于TCP的负载均衡器（HAProxy或NGINX）构建了一套功能丰富的高可用负载均衡解决方案。

我们通过DNS在全球的多个IP地址上分发流量。玩家可以使用类似“riotplzmoarkatskins.riotgames.com”这样的域名，我们的服务器会返回多个IP地址。有一半玩家可能会收到一个地址列表，列表的第一个地址是服务器A，另一半玩家则收到另一个地址列表，其中第一个地址是服务器B。如果服务器A或服务器B发生宕机，客户端会尝试连接到另一台服务器上，因此玩家不会感觉到出现了服务中断。

我们的内部网络中有多台服务器被配置成可以对服务器A的IP地址做出应答。在网络看来，这些服务器都是可用的目标服务器。在收到一个新的玩家连接后，我们使用散列函数来计算该请求应该被转发给哪一台服务器。散列函数使用了IP地址和TCP端口。为了将服务器发生宕机的影响降到最低，我们使用了一致性散列，这样可以确保只有使用了那台宕机服务器的玩家会受到影响。在大多数时候，客户端会自动重新发起一个新的连接，玩家甚至都感觉不到服务中断。在收到新的连接后，因为宕机的服务器已经被移除，我们不会再尝试把请求转发给这台服务器。我们的大部分系统会自动启动一个新的实例，然后把它加到一致性散列环中。负载均衡的最后一层是使用传统的基于TCP或HTTP的负载均衡器，如HAProxy或NGINX。

来自ECMP层的请求会达到负载均衡器实例上。这些实例监测后端的Web服务器，确保它们都是活跃的，并能够在一定时间内返回响应，时刻准备接收新的请求。如果这些条件都满足，服务器就会收到请求，并将响应结果逐层返回给玩家。我们在这一层进行蓝绿部署和智能健康监测，比如检查“/index.html是否返回200响应码”。除此之外，我们还可以进行



canary部署，先让10台服务器中的一台部署最新的应用，剩下9台仍然运行旧版应用。我们对新应用进行严密监控，如果一切顺利，那么就再拿出两台服务器部署新版应用，并以此类推。如果中间出现了问题，我们就回退到上一个稳定版本，并修复问题。这样，我们就可以持续地改进我们的服务，同时又能将风险降到最低。

有了这些层（DNS、ECMP、TCP或Layer 7负载均衡），我们为开发者和玩家提供了一个功能丰富、稳定且可伸缩的解决方案。

失效备援测试

高可用系统最重要的能力是在系统发生故障时能够进行自动恢复。在我们刚开始构建数据中心时，我们通过让工程师拔掉线缆或随机重启服务器的方式来模拟故障。但在数据中心建好并投入使用后，这样做其实很难。我们被要求在做好某些事情之后就永远不要再去动它们。我们确实基于这个流程发现了一些问题，也避免了一些中断，但它仍然有需要改进的地方。

于是我们按照数据中心的比例搭建了一个staging环境，该环境的规模要足够精确，避免造成不必要的浪费。我们使用了两个机架，每个机架上有5台服务器。生产环境的机器比staging要多得多，不过staging环境足够我们进行测试了。

在将变更发布到生产环境之前，我们先在staging环境对它们进行测试。我们对每一个变更进行快速的基础测试，以便发现潜在的bug。这样我们就不用再担心我们的自动化系统会变脆弱，也不用担心太多的变更会让我们不堪重负。我们要在进入生产环境之前，把bug扼杀在staging环境里。

除了基础测试，我们也进行更为复杂的破坏性测试，我们停掉一些重要的组件，强制让系统在受损的状态下运行。在只剩下3到4个子系统可以运行的时候，我们就知道我们可以承受多大程度的损失，并在不影响玩家的情况下修复系统。

这一整套测试比基础测试更加耗时、更加具有破坏性，也更加复杂，所以我们要减少运行整套测试的频次。我们让链接失效、重启机器、重启机架、禁用SDN控制器，只要我们能想到的，我们就去做。然后我们统计系统多久可以恢复，并确保一切运行顺畅。如果出现了偏差，我们就查看从上一次运行到现在究竟做出了哪些代码变更，在这些变更进入生产环境之前把它们都找出来。如果我们在生产环境中遇到了之前没有测试到的问题，我们就把它加到测试用例中，确保同样的问题不会再次发生。我们的目标是尽早发现问题，问题越早被发现，就可以越快修复它们。为此，我们不仅能够走得更快，而且可以走得更自信。

在这一章中，我们介绍了我们是如何实现基础设施即代码、负载均衡和失效备援测试的。“唯一不变的就是变化”这句话很好地概括了我们的处境。基础设施是有生命的，它一直在成长和演化。在它成长的时候，我们要为它提供资源，在它生病的时候我们照顾好它，我们要在全球范围内以尽可能快的速度做好这些。我们要认真面对这样的现实，我们的工具、流程需要跟上不断变化的环境。



第五章 微服务生态系统

这一章将深入介绍微服务以及在Riot容器平台上运行的应用需要具备的五个关键因素，它们分别是：

1. 高度可移植
2. 动态配置
3. 可被发现
4. 可被感知
5. 获取秘钥

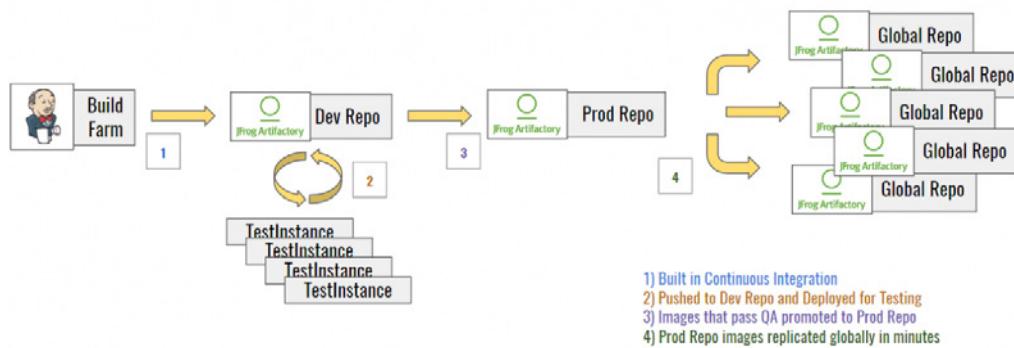
要做到这些，需要额外的服务和工具的支持。有些工具是为“开发人员”准备的，有些则是给“运维人员”使用的。在Riot，开发人员和运维人员都不是某种固定的工作头衔，工程师需要在这两种角色之间来回切换。一个工程师可能今天在开发一个服务，明天就要把它部署出去。我将深入讲解这五个关键因素以及在这一过程中用到的辅助工具，并简要介绍我们的实现方式。

高度可移植

Riot需要在全球范内部署应用程序，我们的服务被部署到世界各地的几十个数据中心，而每个数据中心又可以分为多个区域。我们的目标是“一次构建，多处部署”，所以我们的微服务必须是高度可移植的。

为了可移植性，我们要将服务容器化。我们已经花了很长篇幅讨论了容器和它们的各种应用场景，也讨论了Docker容器技术，但只是将应用放进容器并不能解决所有问题，我们还要将这些打包好的容器发送到全球各地的数据中心。

我们借助强大的JFrog Artifactory建立了多地域复制的Docker注册中心，下图展示了构建一个容器镜像的生命周期：

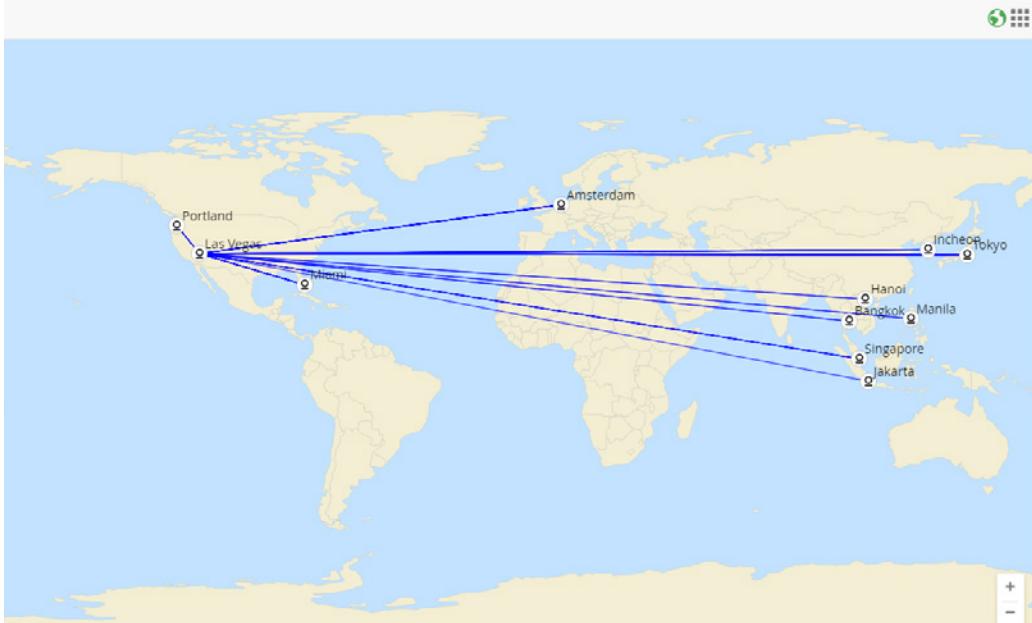


Riot每年要进行125万次构建，其中有一部分就是用来构建微服务的Docker镜像。Docker镜像按照我们的持续交付流程开始构建，并被部署在内部的Docker注册中心上。等到它们准备好进入生产环境，就会被打上“promoted”标签，然后被放进复制仓库中，复制仓库负责将Docker镜像发送给我们的数据中心。

因为这些Docker镜像是基于可重用的层而构建的，所以可以在几分钟内被复制到世界各地。它们的体积很小，因为只有发生变化的部分需要重新构建。

JFrog Mission Control is happily serving **27** instances

You are running Mission Control version 1.6



动态配置

我们目前运行了超过1万个容器，每一个微服务都是由若干个容器组成的。这些容器就像刚出生的婴儿一样，咪蒙着双眼，沐浴在生产环境的灿烂光辉中。它们对生产环境一无所知，所以我们要告知它们的处境，并快速做好配置。在传统的部署系统里，应用程序包含了一些配置信息，然后使用一些工具（如Chef或Puppet）来维护和管理配置。但为了实现可移植性，我们的应用程序必须支持在运行时进行部署和维护。

于是配置即服务登场了。我们希望能够继续使用我们的作用域方案，我们调研了很多开源解决方案，到最后发现自己开发配置服务才能实现最大程度的灵活性。

解决命名问题相对比较简单。在应用程序启动的时候，它们知道自己是谁，也知道自己在哪里，因为我们的调度器已经通过简单的环境变量注入告诉它们了。

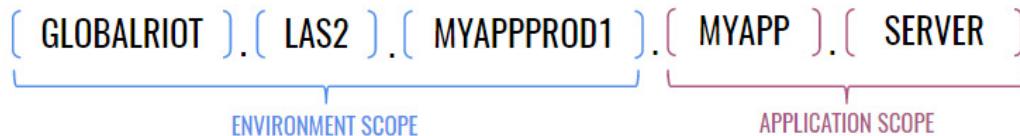
我们的作用域方案被分为两个部分：

[ENVIRONMENT SCOPE]. [APPLICATION SCOPE]

环境作用域被分为三个部分，应用程序作用域则被分为两个部分：



我使用一个叫作“ MyApp ”的小工具作为例子。 MyApp 可以被部署到我们的第二个拉斯维加斯往事数据中心里，它只包含了一个服务器组件，看起来是这样的：



那个叫作“ myappprod1 ”的环境组件非常重要。我可以在同一个数据中心里再部署一个 QA 版本（ myappqa1 ）或者开发版本（ myappdev1 ），或者同时运行两个不同的生产环境版本。我们的作用域方案让我们可以在同一个集群里创建不同的环境。

我们将配置数据发送给配置服务，这样就可以对外提供配置查询。举个例子，如果我想要发送所有部署在“ globalriot.las2.myappprod1 ”上的应用程序的数据，我会这样发送这些数据：

```
[ GLOBALRIOT ]. [ LAS2 ]. [ MYAPPPROD1 ]. [ * ]. [ * ]
```

在“ MyApp ”启动并知道自己是谁之后，它会进行前三个作用域的匹配，根据通配符获取配置数据。如果我想要发送某个特定实例的配置数据，我会这样做：

```
[ GLOBALRIOT ]. [ LAS2 ]. [ MYAPPPROD1 ]. [ MYAPP ]. [ SERVER ]
```

在这个作用域中的所有实例会获取到这些数据。数据本身只不过是一些键值对形式的属性。例如：

```
http.ProxyType=http
http.ListenPort=80
http.DomainNames=myapp.somedomain.io
```

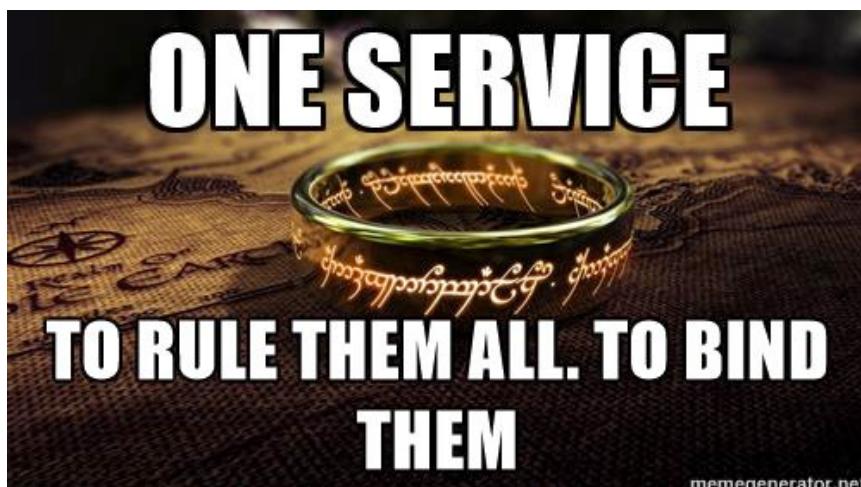
配置数据可以被实时更新，比如速率限定属性：ratelimit.txsec=1000

我可以发送一个新的ratelimit.txsec值，应用程序会实时地检查配置文件，并动态进行调整。我们因此具备了实时修改服务行为的能力。曾几何时，修改《英雄联盟》的排行榜数据需要进行完整的重新部署，而现在，我们可以将配置数据推送到我们的配置服务，我们的服务器会拿到这些新的配置信息，并自动进行调整，完全不会对玩家造成影响。

可被发现

我们的配置服务也是一个微服务，那么其他应用程序如何知道到哪里找到这个服务呢？一个微服务想要与另一个微服务发生通信，它要怎样才能找到这个微服务？这是一个鸡生蛋蛋生鸡的问题。

我们的微服务不需要域名，实际上，它们在启动的时候被分配了随机的IP地址。我们使用发现服务来解决这个问题，或者说，我们使用了“一个服务来统管它们”。我们为发现服务提供了一个域名，其他服务可以通过这个域名找到发现服务。



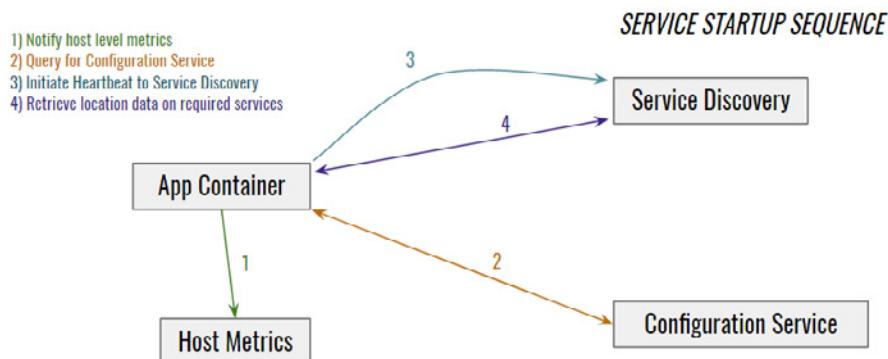
memegenerator.net

刚开始，我们受到了Netflix Eureka的启发。实际上，我们的第一批部署就使用了Eureka实例。Eureka非常好用，但随着时间推移，我们认为我们需要一些能够更懂我们运行环境的东西。

应用程序在启动之后，通过发现服务找到配置服务的地址。应用程序需要事先做好配置，把自己注册到发现服务上。这样，其他服务就可以找到它们，并知道它们提供了哪些服务。下面的例子展示了我们的一个度量指标服务，它负责收集我们的一个QA环境的度量指标：

Discoverous v1.0.46	
Application: hmp.metricssd	
Location: lolqa.las1.riot3	
Version	Contracts
2.0.144	riot.control:1.0.0 riot.query:1.0.0 riot.swagger:1.2.0
Served Locations	Instances
	240:1102 (UP) 246:1102 (UP) 234:1102 (UP) 159:1102 (UP) 237:1102 (UP) 244:1102 (UP) 224:1102 (UP) 233:1102 (UP)

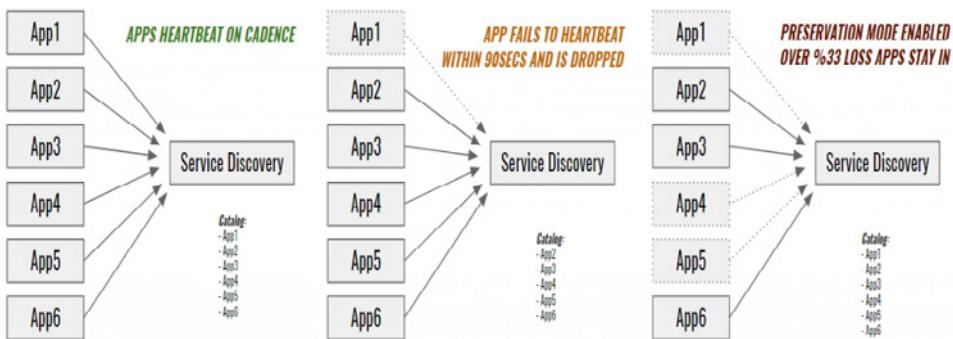
应用程序在找到发现服务后开始定位配置服务，然后发送度量指标。当然，它也可以从发现服务上找到其他服务，然后与它们发生交互。



这些看起来似乎很简单，但也有一些复杂的场景需要引起我们的注意。例如，如果一个服务发生宕机，我们要把它从发现服务中移除，否则

就会返回错误的地址；如果一个服务改变了地址，我们要在发现服务中及时更新它，否则流量会被路由到错误的节点上。

我们使用了简单的心跳机制。如果一个服务在指定的时间内无法返回响应，它就被认为已经宕机，并从发现服务中移除。不过，在生产环境中，事情可能会变得更加复杂。如果在数据中心层面发生了非常严重的故障，我们需要采取恰当的措施。比如，如果发现服务遇到大量的注册服务停止发送心跳，那么可以肯定的是，数据中心出现了大面积的网络通信故障。这个时候，发现服务开始进入“保留”模式。在该模式下，发现服务会保留注册信息，并立即通知运维人员赶紧采取措施修复问题，在问题得到修复之后，继续提供服务。



可被感知

Riot的所有微服务都会向某个端点发送心跳检测信息，类似“健康”、“降级”、“失效”等。我们可以通过简单的REST调用来查询发现服务，从而了解所有服务的健康状态。

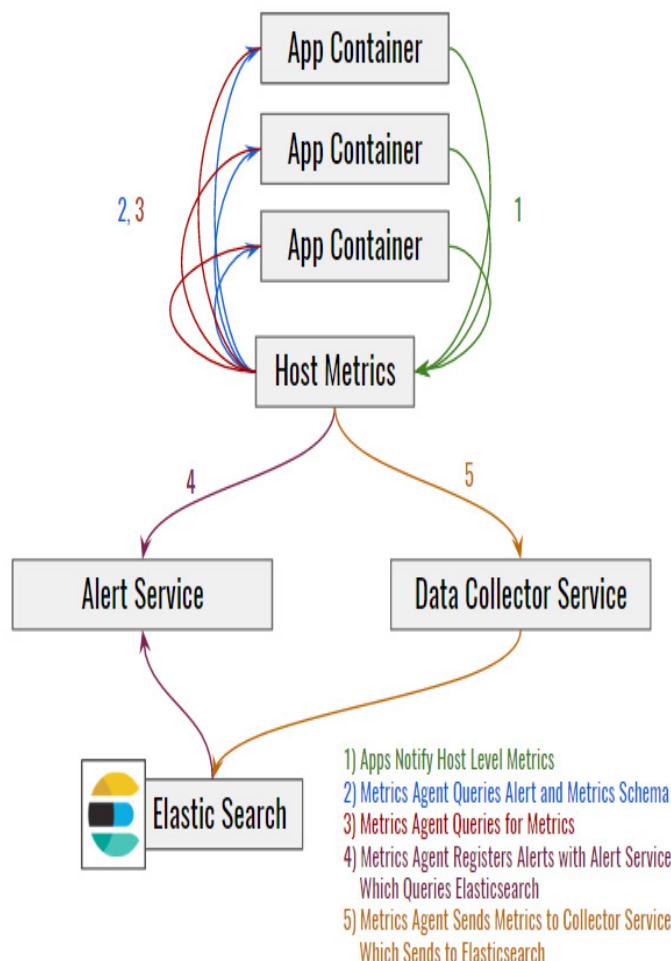
但这样还远远不够。如果一个服务注册失败会怎样？或者如果一个服务因为发生故障而取消注册了会怎样？如果一个服务没有注册到发现服务上，那么我们怎么知道它的状态？

于是，我们的告警和度量指标系统开始派上用场。

度量指标系统向应用程序查询度量指标，并把这些信息发送到度量指标管道中，然后被推送给数据中心的数据收集器。这些数据被保存到

Elasticsearch里，然后watcher会基于这些数据发出告警。

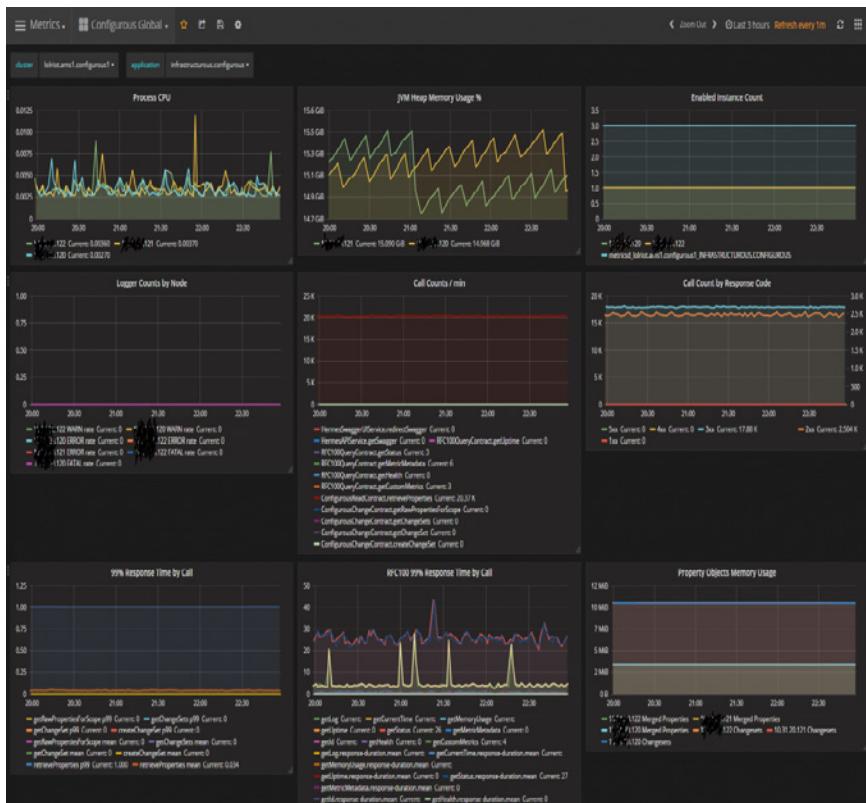
应用程序提供了告警端点。告警服务会注册这些端点，并通过监控服务来监控度量指标的状态变更。如果一个应用程序的状态从“健康”变成“降级”，并且它已经注册过告警，那么告警服务就会向注册过的联系人端点（通过邮件、手机等方式）发送通知。



那么度量指标系统是怎么找到数据收集器的？答案是通过服务发现！开发人员可以通过在配置服务上设置度量指标类型、发送时间间隔或告警信息来实时地修改度量指标和告警内容。如果觉得某种告警的数量太多，那么就修改一下配置，让应用程序取消掉这个告警。

随后，度量指标被聚合到数据仓库中。我们将数据移动到实时的数据

管道中，该管道主要使用了Elasticsearch，并由Riot的数据产品和解决方案团队进行运维。数据在进入管道之后，我们就可以创建仪表盘。因为应用程序在度量指标中包含了它们的作用域和指标数据，我们可以查到某个具体应用程序的度量指标。



上图是配置服务的度量指标截图。从图中可以看到它的CPU负载（最右边），以及它每分钟接收2万个左右的请求。这个实例来自我们的阿姆斯特丹数据中心——它的“集群”是“lolriot.ams1.configurous1”（部署作用域），应用程序是“infrastructure.configurous”（应用程序作用域）。

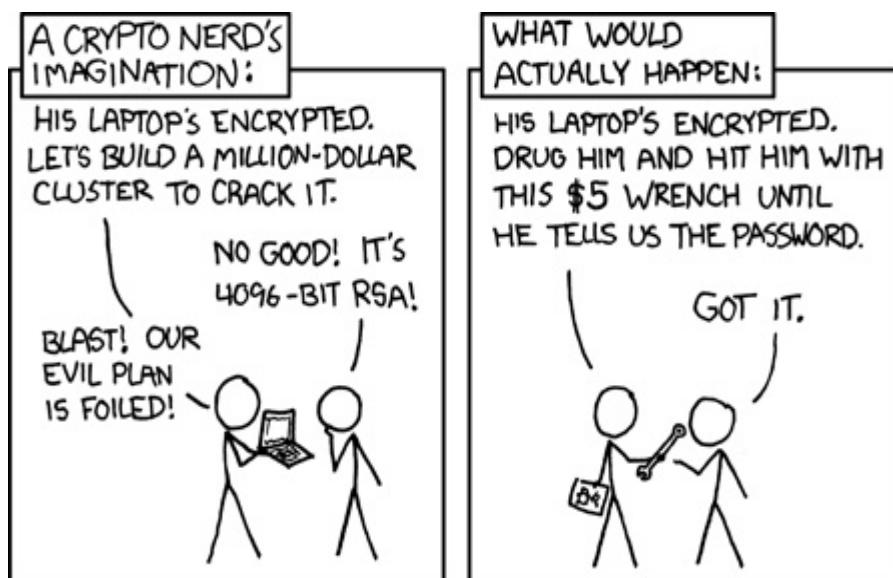
如果有必要，我们可以为这些度量指标创建告警。比如，我们可以开启“Enabled Instance Count”告警，如果可用实例少于“3”，就发送告警联系某人。

开发人员在前期把他们的应用程序注册到发现服务和配置服务上，后续就能自动获得这些报表信息。

获取秘钥

之前一直没有提到安全问题，实际上，通信安全是任何高度可移植、可动态配置的微服务系统需要关注的问题。HTTPS流量或API认证令牌的SSL证书需要被保护起来，我们把它们放在配置服务里，方便其他服务访问，但我们绝对不能使用明文保存。那么我们是怎么做的呢？

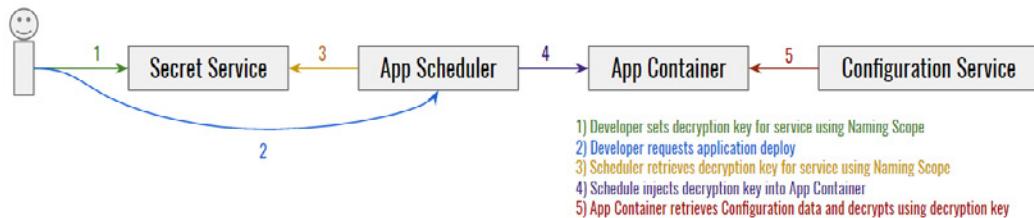
如果我们在配置服务中加密和保存数据会怎样？如果是这样的话，应用程序在获取证书后需要对其进行解密，而且要确保获取证书的应用程序有解密秘钥。这就是我们整个运维蓝图的最后一道关口：秘钥管理。



为了实现秘钥管理，我们对HashiCorps的Vault服务进行了封装。Vault提供的功能比我们需要的要多很多，我们只需要保存解密秘钥，应用程序在拿到秘钥后可以解密数据。我们封装的服务提供了REST端点，应用程序可以通过这些端点获取秘钥。

从理论上看，这个很简单，开发人员将解密秘钥放到秘钥服务器上，并提供应用程序的命名空间。在应用容器启动的时候，我们的容器调度器Admiral将这些秘钥注入到应用容器里（根据应用程序提供的命名空间来查找特定的容器）。应用容器拿到解密秘钥后，就可以用它解密从配置服

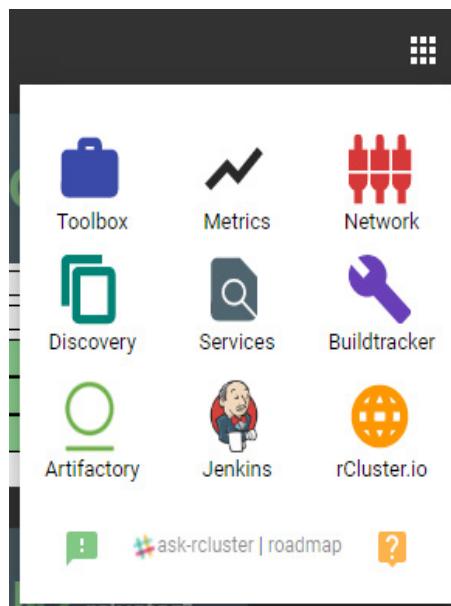
务获取的配置数据。配置数据的提供者在将数据推送给配置服务之前需要使用加密密钥来加密数据。



有了这些流程，我们的服务实现了高度可移植、可动态配置、自我感知、可被发现，并能够安全地处理数据。

开发者生态系统

我们已经介绍了所有在Riot生产环境中运行的服务，不过我们的生态系统里还有其他很多东西。为了使用好这些系统，我们创建了很多Web和命令行工具。如果说之前所讨论的只是我们的生产生态系统，那么接下来还需要介绍我们的开发者系统。在下一章中，我将继续介绍我们的开发者生态系统。这里先透露一张截图，它是我们的一个Web应用控件，我们通过它访问生态系统的工具和数据。





第六章 开发者生态系统

在上一章结尾部分，我向大家展示了用来访问我们微服务生态系统的Web工具。我们之前已经介绍过其中的一些应用：用于保存容器镜像的JFrog Artifactory、发现服务、度量指标系统以及用于自动化和持续交付的Jenkins。除了这些，还有其他很多工具。

因为时间的关系，我没办法逐一介绍它们，不过我会集中介绍几个在管理生态系统关键部分给我们带来巨大帮助的工具。我们借助这些工具来：

1. 可视化和检视全球的容器集群（Toolbox）
2. 简化软件网络规则的处理（network.rcluster）
3. 在全球范围内查找服务（服务发现）
4. 对构建和部署过程进行跟踪（Build Tracker）

可视化集群

下面是Toolbox的截图，也就是我们的容器可视化工具。在之前的章

节中，我们介绍了Admiral调度器，这张图片所显示的就是从这个调度器的REST API返回的数据可视化结果。我们可以从图中看到全球集群的状态（可以看到总共有16个集群），并根据部署区域分开显示。Riot的集群分布世界各地，包括台北、雅加达、迈阿密、阿姆斯特丹、韩国和日本。

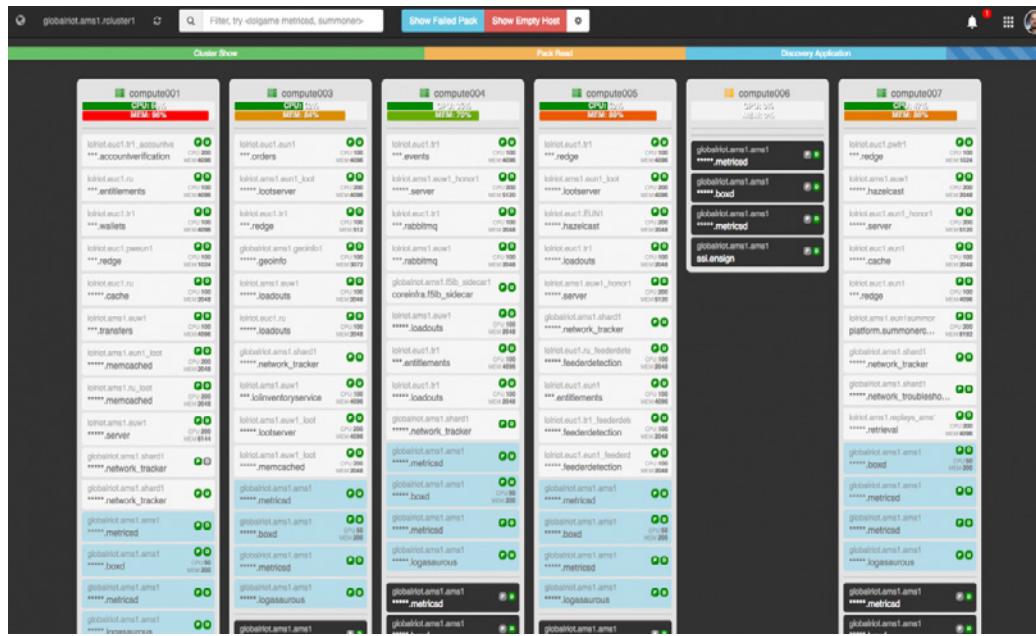


我们运行了2400多个应用程序实例，并将这些实例叫作Pack。这些实例需要5000多个Docker容器来运行。在过去一两年中，这些Pack一直处于运行状态。不过，上图只显示了运行在容器当中的服务，Riot还有其他很多服务并没有在张图上显示出来。

Toolbox不仅提供了一个全局视图，我们还可以点进去查看各个数据中心的详细信息。

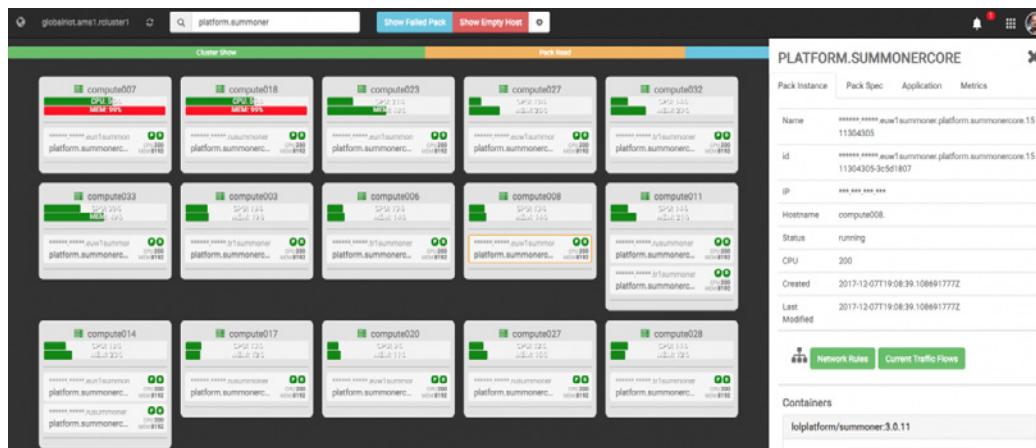
一张截图无法显示所有内容，不过从这张阿姆斯特丹系统的简单视图中，我们可以看到运行中的应用程序的数量。我们可以看到底层的叠加网络服务，还能看到节点应用程序、Pack的状态，以及哪些应用程序注册到了发现服务上。开发人员和运维人员可以基于这些视图更好地了解服务的运行情况。

如果要查看服务的细节，可以继续点进去。下面以我负责的 Summoner（召唤者）服务为例。该服务负责处理来自Riot Chat和



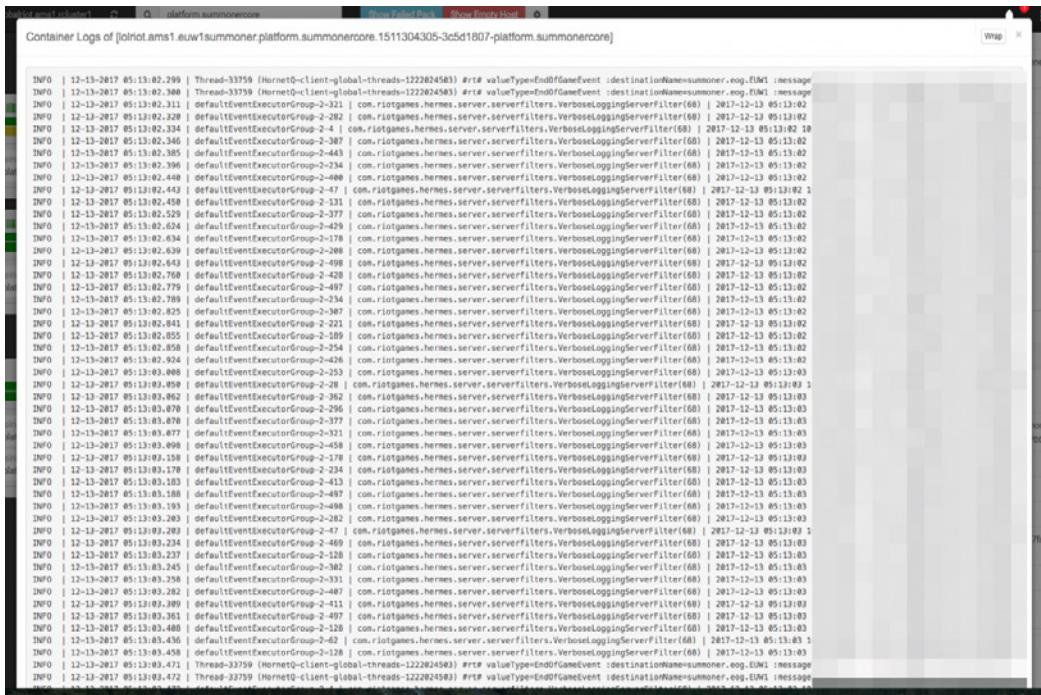
Developer API Portal的流量。

我们基于命名空间和作用域系统来管理应用程序。如下图所示，Toolbox根据应用程序命名空间和作用域进行过滤，在这张图中，我们查看的应用程序叫作“platform.summonercore”。我们可以看到应用程序实例是如何分布的，包括它在AMS1中使用了多种部署作用域进行部署。比如，“lolriot.ams1.rusummoner”和“lotriot.ams1.tr1summoner”分别用来支持俄罗斯和土耳其。

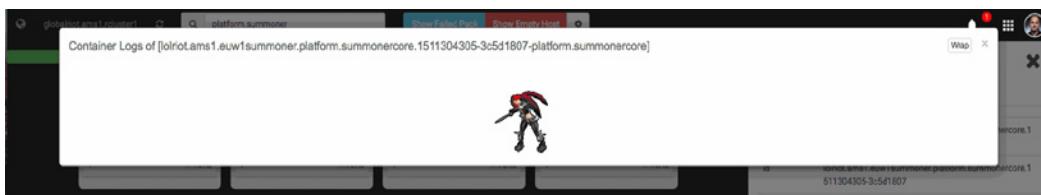


右边的侧边栏包含了额外的信息，如Pack中包含的容器数量、IP地

址、基本状态、日期信息，等等。我们还能在上面查看容器日志。



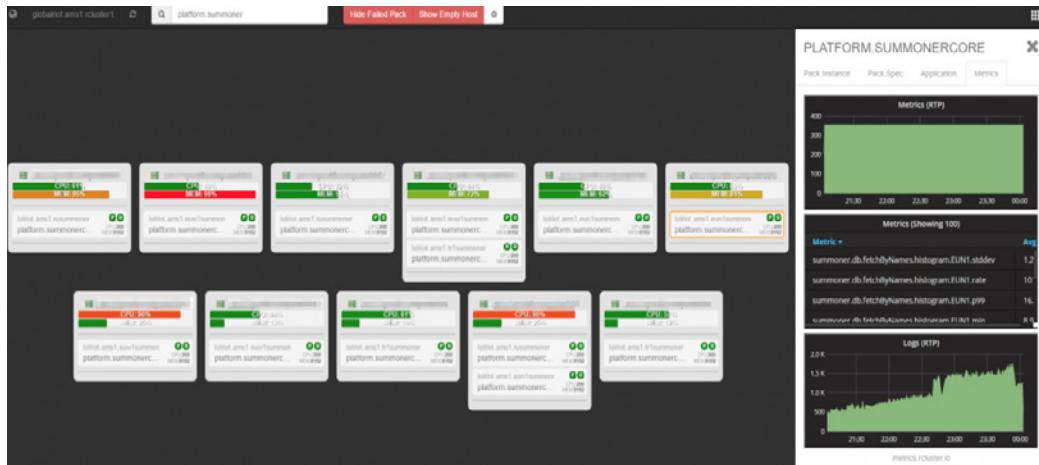
这张图展示的是我最喜欢的一个功能。在加载日志过程中，我们可以看到一张GIF图片——跳舞的Katarina（游戏中的一个角色）。没错，她会出现在我们内部各种工具的加载屏幕上。



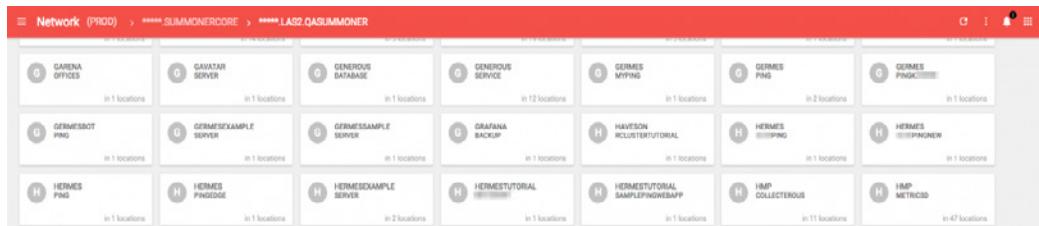
Toolbox中的度量指标系统为我们提供了一站式的核心服务信息查看功能，如查看服务状态和服务位置。在发生问题时，我们可以立即进行问题诊断。我们还能进行截屏，如下图所示。

管理复杂的网络规则

在之前的章节中，我们介绍了我们是如何通过Contrail和JSON配置文件来管理网络的。JSON很不错，但如果文件太长，阅读起来十分不方便。为了方便工程师查看JSON文件，我们开发了一个可视化工具，并把



它叫作“network.rcluster”。

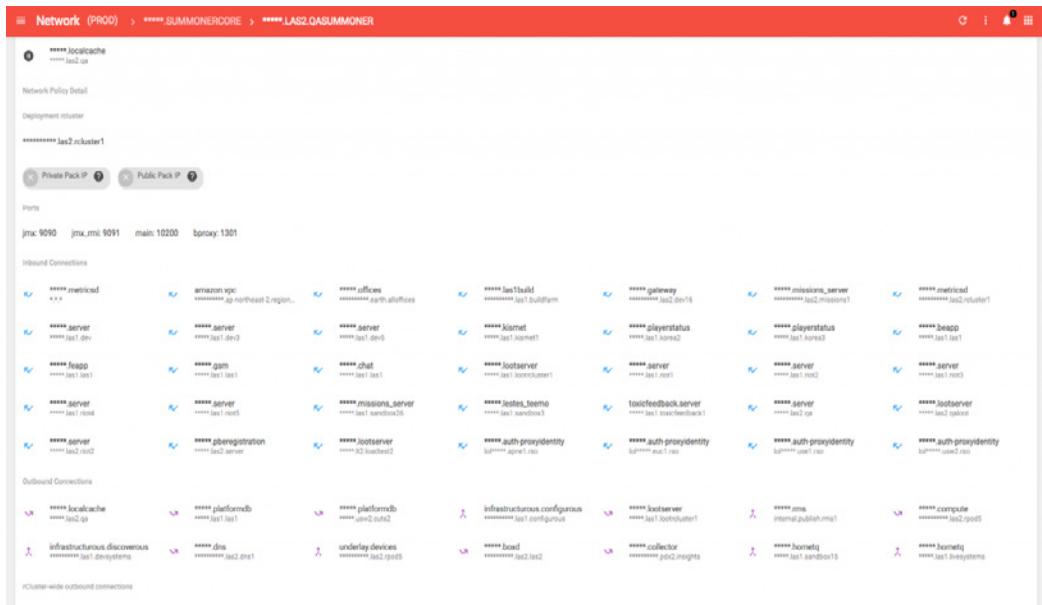


在登录系统之后，可以看到一行一行的部件，它们表示不同集群的网路规则。每一个部件都是通过一个JSON文件来生成的。现在让我们仔细看一下之前提到的Summonercore。

The screenshot shows the 'platform.summonercore' configuration interface. It includes sections for 'Approval Policy' (with a table for 'Target Locations (pattern) of platforms.summonercore' and 'Approved Services (pattern) for incoming Connections'), and a 'Locations' section where users can select locations from a dropdown menu. The locations listed are: lolarena.blk2.summoner, lolarena.jct2.summoner, lolarena.mrn2.summoner, lolarena.sq1.summoner, lolarena.sgn3.summoner, lolarena.tpd1.summoner, lolqa.lnt1.summoner, lolqa.lsc2.summoner, lolqa.lsc2.qasummoner, lolqa.lt2.locales2summoner, lolqa.pdr2.summoner, lolqa.unv2.summoner, and loliot.ams1.summoner.

初一看好像没有什么，它只不过是一个部署作用域清单而已。我们可以看到Summoner有很多部署作用域网络规则。因为在有《英雄联盟》的地方，都会有Summoner，所以这些网络规则是必需的。

如果选择其中一条规则，我们可以看到Summoner的访问权限设置。



这里有非常多的路由规则，我们可以查看端口信息和流进流出的网络连接。从这张图中，我们可以看到Summoner可以与“rtp.collector”和“infrastructurous.discoverous”发生交互，后者是我们的发现服务。这张截图是从QA环境中获取的，所以还能从中看到一些测试应用程序。

全局搜索

运行如此多的服务，如何跟踪这些服务就会成为一个问题。我们可以使用Toolbox来查看每个集群，但它只显示了运行中的Pack和容器。Riot还有很多遗留的系统部署在物理机上，我们希望能够把它们也管理起来。

于是，我们开发了查询服务，或者叫做信息聚合器。我们把这个工具叫作“services.rcluster”，它为我们提供了各种基于上下文的搜索服务。下图展示了我们使用这个工具在全球范围内查找Summoner服务。

这里需要澄清的是，查询服务与服务发现是不一样的。它提供的是

Host	Location	Status	Service	Version	ServiceIP	HostIP	Chef/Merlin	Roles	Status	OS
lolqa.last1.summonersb15		✓ UP	platform.summonercore	3.0.8-SNAPSHOT				Roles ...	Enabled	
lolqa.last1.summonersb15		✓ UP	platform.summonercore	3.0.8-SNAPSHOT				Roles ...	Enabled	
lolqa.last1.summonersb15		✓ UP	platform.summonercore	3.0.8-SNAPSHOT				Roles ...	Enabled	
lolrot.usw2.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolrot.usw2.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolrot.usw2.esports		✓ UP	platform.summonercore	1.0.48				Roles ...	Enabled	
lolrot.usw2.esports		✓ UP	platform.summonercore	1.0.48				Roles ...	Enabled	
lolrot.usw2.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolrot.usw2.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolrot.euc1.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolrot.euc1.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolrot.euc1.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolrot.euc1.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolrot.euc1.esports		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolgarena.tpe2		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolgarena.tpe2		✓ UP	platform.summonercore	1.2.3				Roles ...	Enabled	
lolgarena.tpe2.summoner		✓ UP	platform.summonercore	1.1.2				Roles ...	Enabled	

基于上下文的搜索，用于查找未注册的服务。例如，它会扫描Admiral调度器，返回匹配的结果。如果你只记得“platform.summonercore”中的“summoner”，那么就可以用这个字符串来查找这个服务。

在这张图上可以看到“Location”列，它显示了命名空间作用域。服务名称显示的是应用程序作用域。

构建跟踪器

我们已经介绍了我们是如何管理生产环境服务的。不过，这些服务在进入生产环境之前还有很长的路要走。我们每年要进行一百万次构建，如果没有一个跟踪系统，事情会变得一团糟。

Buildtracker是另一个基于API和Web驱动的工具，开发团队可以通过自动或手动的方式提交或查询数据。他们通过这个工具来跟踪他们的代码和构建过程。

我可以为这个写一本书，这也是第一次公开介绍这个工具，但其实我们已经用这个工具3、4年时间了——早在使用微服务之前就开始使用它。

因为要构建的服务太多了，我们不希望开发团队通过无数的构建管道

Login Successful!
Welcome back to Build Tracker, Maxfield Stewart!

Build Tracker

I can help you do that better!

It's 7pm. Do you know where your code is? Do you know where it has been?
Now, with Build Tracker, you are able to keep tabs on your changes as they travel through the Riot pipeline to final deployment on Live. Build Tracker collates existing metadata such as Perforce or Git changelist, version number and creation date, then combines it with categorized and labeled tags to create an informative and thorough depiction of the build status.

Builds track rudimentary metadata such as version, changelist, etc.
Tags explain the significance of a build through documenting the what, where, and why.

[View on GitHub](#) [Submit Feedback](#) [Changelog](#)

日志来跟踪服务的构建过程。Buildtracker提供了一组干净的API，可与持续集成系统（或任何自动化部署系统）集成起来，进行构建、打标签、查询。

如果某个团队决定要开发一个微服务，就可以先生成一个微服务构建管道。他们也可以自己搭建构建管道，然后使用这组API进行跟踪。他们可以使用API来查询构建结果，如下图所示：

Build Type	Branch	Configuration	Changelist	Build Date	Version	Tags	Actions
configurable	master	rcluster-shippable	799abfbac1...	2017-04-28 00:28:55	1.0.144	recluster-shippable Maven-Release	
configurable	master	merlin-shippable	799abfbac1...	2017-04-28 00:27:45	1.0.144-201704280023	merlin-shippable Maven-Release Sandbox10 Smoke-Sandbox10-Passed Functional-Pass Load-Test-Pass DevSystems Smoke-DevSystems-Pass PERF1 Smoke-PERF1-Pass	
configurable	master	rcluster-shippable	2aa26f4451...	2017-04-13 04:13:57	1.0.143	recluster-shippable Maven Release	
configurable	master	merlin-shippable	2aa26f4451...	2017-04-13 04:12:59	1.0.143-201704130407	merlin-shippable Maven Release Sandbox10 Smoke-Sandbox10-Passed Functional-Pass Load-Test-Pass PERF1 Smoke-DevSystems-Pass Smoke-PERF1-Pass	
configurable	ltbreaks	rcluster-ltbreaks	e42052a774...	2017-04-12 08:51:38	1.0.141-20170412-0850	recluster-ltbreaks	
configurable	ltbreaks	merlin-ltbreaks	e42052a774...	2017-04-12 08:50:46	1.0.141-201704120848	merlin-ltbreaks	
configurable	ltbreaks	rcluster-ltbreaks	ea26e76dee...	2017-04-12 08:35:53	1.0.141-20170412-0834	recluster-ltbreaks	

这张图显示的是我们的配置服务。我们添加了很多种过滤器，如变更

列表、在构建时使用的版本和其他各种标签。标签可用于跟踪多种行为，如应用文件的部署环境（红色部分）和QA传入的事件（灰色部分）。开发团队可以使用Buildtracker标签来标记各个构建是否已经通过测试，然后过滤出已经通过测试的构建。团队因此可以创建出可信任的持续交付管道，确保只部署已经经过严格测试的项目。

即使团队不会完全采用这个流程，他们仍然可以查看构建历史的详细信息。

The screenshot shows the Build Tracker application interface. At the top, there's a navigation bar with links for 'Search', 'Release Tools', 'Help', 'Build Types', and 'Admin'. On the right, it says 'Logged in as: Maxfield Stewart' with a notification icon showing a red '1'.

The main content area displays a build entry for 'configurous 1.0.144-201704280023'. It has three tabs: 'Build' (selected), 'Source Control', and 'Jenkins'.

Build section details:

- Build Date: April 28th, 2017 12:27 AM
- Build Configuration: merlin-shippable
- File Type: zip
- Path: [/Builds/configurous/master/merlin-shippable/799abfbac1cdcaa1a01758864cbf3812a9dad9e48/configurous-1.0.144-201704280023.zip](#)
- Notes: Enter notes here... (with an 'Apply' button)

Tag Timeline and **Build Status Timeline** sections are also visible.

Source Control section details:

- Changelist: 799abfbac1cdcaa1a01758864cbf3812a9dad9e48
- Changelist Date: April 28th, 2017 12:27 AM
- Source Control: Github
- Branch: master

Jenkins section details:

- Jenkins Job URL: [/job/configurous-build-pipeline/](#)
- Jenkins Build URL: [/job/configurous-build-pipeline/175/](#)
- Jenkins Configuration: Show Full Jenkins Configuration...

Includes section is present but empty.

上图中包含了部署文件的路径、构建作业的链接和各个事件的时间线。我们可以从Buildtracker的Release Management视图中看到个多元数据信息。

这张图片展示了我们的一个发布团队在管理《英雄联盟》版本发布时的内容。其中包含了客户端、服务器、音效包和服务信息。从中还能看到很多标签，如补丁标签、环境标签、QA流程标签等。

在开发大量的服务和应用时，有这样一个聚合器将会为你带来很大

的帮助。

The screenshot shows the Build Tracker interface with the title "Latest Builds for 7.22_Package_1". The interface includes a search bar, release tools, help, build types, and admin options. It displays a table of builds categorized by environment (Release Pool, Current Pool, Current RC, 7.20, 7.21, 7.22, 7.23, 7.24) and build type (ejabberd, game-audio, game-client-code, game-client-content, game-server). Each row provides details like branch, configuration, changelist, build date, version, and tags. A tooltip over the "Tags" column for the 7.22 build for game-client-content shows a list of tags including "game-client-content", "7.22", "7.22_Package_1", and "7.22_Package_2".

	Build Type	Branch	Configuration	Changelist	Build Date	Version	Tags
Release Pool							
Current Pool							
Release Tags	<input type="checkbox"/>	ejabberd	master	shipable	25e953f39b...	2017-10-26 17:28:50	7.22.5 shippable Propoff Testenv Devs DevOff Build jenkins, slave001, 2017-10-26 Devs DevOff Host Host Host Headless Host Host src src src src src src src script script script script script script script unit unit unit unit unit unit unit integration integration integration integration integration integration integration Host Kernel L7I Chat General Tencent-Auto-Pipe Devs Headless Propoff DevOff 7.22_Package_1 7.22_Package_2
Current RC							
7.20							
7.21							
7.22	<input checked="" type="checkbox"/>	game-audio	Releases/7.22	now	2079916	2017-11-02 10:44:30	7.22.207.9916.0- 20171102-173298 Build Jenkins, Slave001, 2017-11-02-10:44:30 Devs DevOff Host Host Host Headless Host Host src src src src src src src script script script script script script script unit unit unit unit unit unit unit integration integration integration integration integration integration integration LOM MLO Propoff Propoff Propoff 7.22_Package_1 plus complete plus
7.23							
7.24							
Latest Builds for Tags							
Riot1	<input checked="" type="checkbox"/>	game-client-code	Releases/7.22	public-win32	2079916	2017-11-02 10:38:48	7.22.207.9916.0- 20171102-173298 Build Jenkins, Slave001, 2017-11-02-10:38:48 Devs DevOff Host Host Host Headless Host Host src src src src src src src script script script script script script script unit unit unit unit unit unit unit integration integration integration integration integration integration integration LOM MLO Propoff Propoff Propoff 7.22_Package_1 plus complete plus
Preprod1							
PBE							
Staging							
Live	<input checked="" type="checkbox"/>	game-client-code	Releases/7.22	public-mac	2079916	2017-11-02 10:33:30	7.22.207.9916.0- 20171102-173298 Build Jenkins, Slave001, 2017-11-02-10:33:30 Devs DevOff Host Host Host Headless Host Host src src src src src src src script script script script script script script unit unit unit unit unit unit unit integration integration integration integration integration integration integration LOM MLO Propoff Propoff Propoff 7.22_Package_1 plus complete plus
KR Live							
	<input checked="" type="checkbox"/>	game-client-content	Releases/7.22	now	2079916	2017-11-02 10:40:44	7.22.207.9916.0- 20171102-173298 Build Jenkins, Slave001, 2017-11-02-10:40:44 Devs DevOff Host Host Host Headless Host Host src src src src src src src script script script script script script script unit unit unit unit unit unit unit integration integration integration integration integration integration integration LOM MLO Propoff Propoff Propoff 7.22_Package_1 plus complete plus
	<input type="checkbox"/>	game-server	Releases/7.22	public-now	2079916	2017-11-02 10:37:48	7.22.207.9916.0- 20171102-173703 Build Jenkins, Slave001, 2017-11-02-10:37:48 Devs DevOff Host Host Host Headless Host Host src src src src src src src script script script script script script script unit unit unit unit unit unit unit integration integration integration integration integration integration integration LOM MLO Propoff Propoff Propoff Game-Auto-Pipe Devs Headless Propoff DevOff 7.22_Package_1 plus complete plus

在这一章中介绍的大部分工具都是自动化运行的。有一些是可选择的，团队可以根据需要决定要不要使用它们。我们的策略是，如果一个工具很有用，那么团队就会用它，而不是自己再去开发。这种灵活、敏捷的氛围让我们可以集中精力为团队创建最有价值的工具，真正给他们带来帮助。而借助这些工具，产品团队就可以快速地将新的功能交付给玩家。

版权声明

InfoQ 中文站出品

《英雄联盟》在线服务运维之道

©2018 北京极客邦科技有限公司

本文最初发布于 Riot 博客，经授权由 InfoQ 中文站翻译并分享，未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：北京极客邦科技有限公司

北京市朝阳区来广营容和路叶青大厦北园 5 层

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com.cn。

网 址：www.infoq.com.cn

ArchSummit

全球架构师峰会

2018 · 深圳站

从2012年开始算起，InfoQ已经举办了9场

ArchSummit全球架构师峰会，有来自Microsoft、Google、Facebook、Twitter、LinkedIn、阿里巴巴、腾讯、百度等技术专家分享过他们的实践经验，至今累计已经为中国技术人奉上了近千场精彩演讲。

2017.07.07-08 深圳站

how to use sagas to maintain data consistency in a microservice architecture

--Chris Richardson, *POJOs in Action* 作者，知名微服务专家

2017.12.08-11 北京站

《创新是人类的自信》

--王坚博士，阿里巴巴集团技术委员会主席

2018.7.06-09 深圳站

限时**7折报名中**，名额有限，快快抢购。

7折报名中
名额有限，快快抢购

华南地区架构领域最有影响力的会议，届时有哪些专题和演讲，敬请扫描右方二维码浏览官网。





极客时间

重拾极客精神·提升技术认知

「专栏订阅」

每天 10 分钟，邀请顶级技术专家，探究技术本质，解读科技动态。

「极客新闻」

每天早上 8 点，朝闻技术天下事。

「热点专题」

最前沿的专题，最独特的视角，最风趣的解读。

「二叉树视频」

一档属于技术人的直播和短视频节目，记录与时代并行的技术人。



关注我，获取更多干货

