

DOIS | 2018 · 深圳站
DevOps 落地，从这里开始

DevOps 国际峰会

暨 DevOps 金融峰会

指导单位： 云计算开源产业联盟
Open Source Cloud Alliance for Industry (OSCAR)

主办单位： DevOps时代

 高效运维社区
GreatOPS Community

时间：2018年11月2日-3日

地址：深圳市南山区圣淘沙大酒店（翡翠店）

中小金融企业如何开心玩DevOps

李晓璐 IT架构师

目 录

1

从0构建运维平台 —— 先解决温饱问题



2

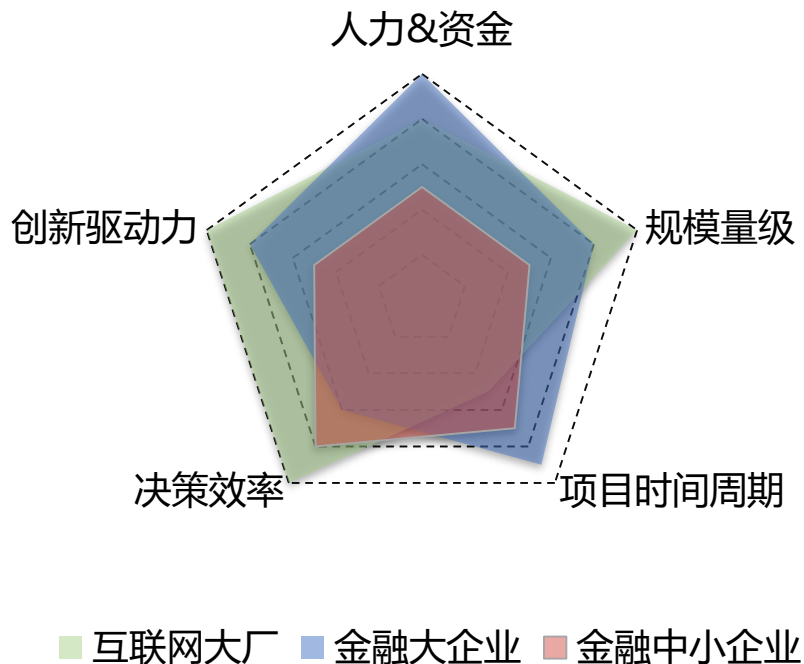
基于微服务的运维开发框架 —— 满足精细化要求



3

DevOps持续集成 —— 走向开发&运维

传统中小金融企业IT建设的特点 —— 适当性原则 DOIS



– 劣势

- x 人员和资金相对不足
- x 创新不是显著的绩效目标

– 优势

- ✓ 规模量级不大
- ✓ 项目时间压力不大
- ✓ 决策效率高

企业特点决定思维方式 —— 工具化建设思路

DOIS



VS



- 工具拿来主义，尽量不造轮子
- **注意耦合性设计，工具之间集成为主，不去刻意修改源码，适当扩展功能**
- 能及时替换不再适合的工具
- 稳定、可靠为重，不刻意追求技术的高大上
- 节约人力资源，聚焦短期目标
- **实现周期短、见效快**
- 要克制满足太多的个性化需求，尽量去适应工具本身

找个好工具

找个好厂商

8/2原则

技术高大上

- 完全按照自主的想法去设计和实现，强调个性化需求，同时控制项目管理
- 使用当下最先进的技术和方法（尝鲜最新技术）
- **寻找技能靠谱且长期能稳定的厂商（或者找到一批开发能力强的工程师）**
- **分阶段，可持续性投入和改进，易用性高，长期容易达成较高的满意度**
- 总体人员和费用成本高
- 技术不能完全自主可控（非互联网大厂）

目 录



1

从0构建运维平台 —— 先解决温饱问题

2

基于微服务的运维开发框架 —— 满足精细化要求

3

DevOps持续集成 —— 走向开发&运维

先解决运维温饱问题，即把监控做好，才能空出时间做进阶的事，比如搞DevOps，当时的情况：

- 仅有彼此独立的几个基础类的监控小工具：机房监控 + 操作系统级监控 等
- 监控覆盖率低，监控工具不稳定、不可靠
- OS和应用监控以ping为主：前端Delphi、后端java
- 券商核心组件的监控主要是看看进程在不在
- 监控参数调整需要硬编码
- 监控策略很简单（有时过于敏感，误报很多）
- 不是一体化的监控平台，数据无法整合与分析



基本就是：凑合着用，监控系统可有可无，各系统以人肉监控为主

一体化监控平台集成了多个监控子系统，全面覆盖了基础设施监控、近2000个OS及其上的中间件、应用、日志、业务等监控

- Zabbix**开放平台监控系统**

- 针对基础设施、网络、Linux&Window系统（不需要抓图形界面）下的系统、数据库、中间件、业务等进行监控

- KCMM**核心业务监控系统**

- 针对需要通过Windows界面元素抓取的系统进行监控，包括BP、XP、各类证券客户端系统等

- TM**交易性能分析系统**

- 通过对核心交换机端口镜像，抓包分析核心业务的性能，包括交易量、响应时间、成功率等

从0构建运维平台

集成模块是一体化监控平台的核心部分，全面汇集了告警数据、性能数据，数据汇集只有秒级延迟

告警集成

- 集成了Zabbix、KCMM、TM监控子系统的告警，并且集成了其它全部已有监控系统的告警，如机房、存储等
- 扩展了Omnibus的事件处理机制，实时把告警从内存数据库转发到Kafka，再建立多个kafka的消费者，如大屏展示、微信通知等

性能集成

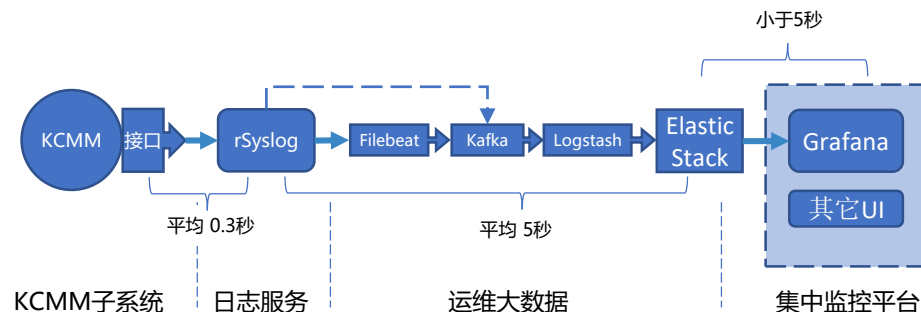
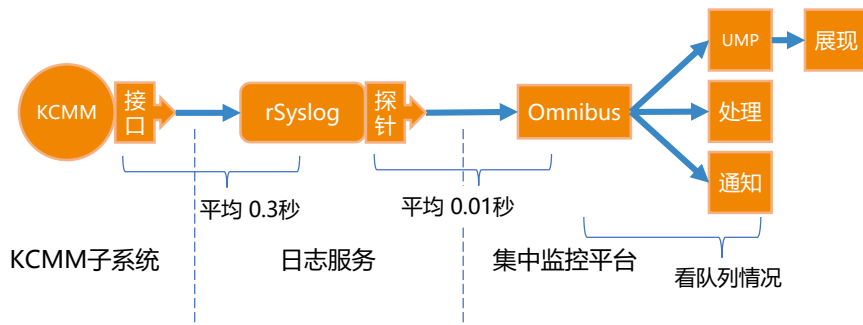
- 集成了来自Zabbix、KCMM的性能数据。短期数据存放到ElasticSearch，长期数据存放到Hadoop平台
- 通过ElasticStack还对业务日志进行了分析，对业务进行监控，例如：外围非法用户访问分析、邮件发送异常分析、核心业务性能分析等

展现集成

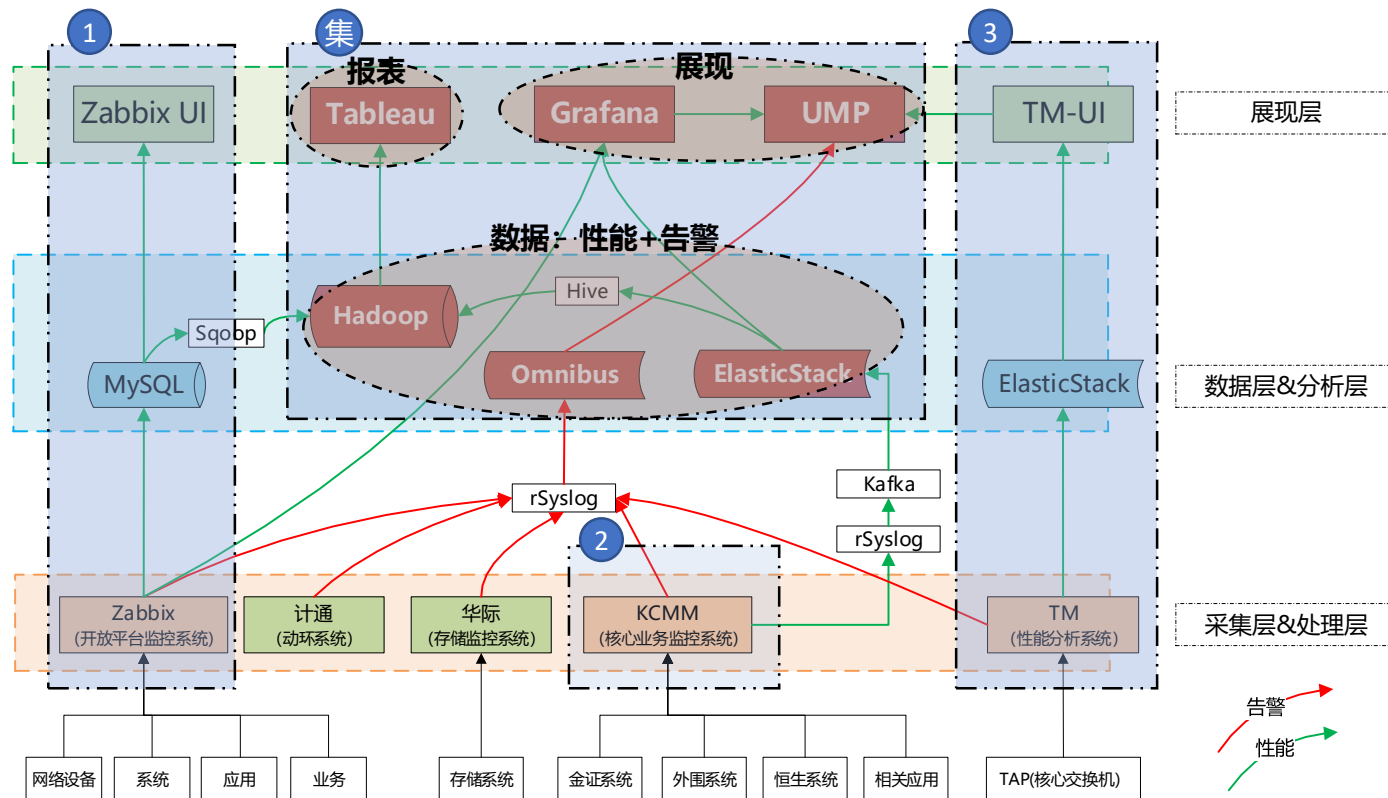
- 采用开源Grafana进行监控信息的展现，统一了展现风格

报表

- 监控历史数据全部汇入Hadoop平台，通过Tableau进行报表分析



监控平台技术架构：3个子系统+1个集中平台



一体化监控平台 —— 监控效果

DOIS

OA应用访问情况



拨测监控



内部管理系统

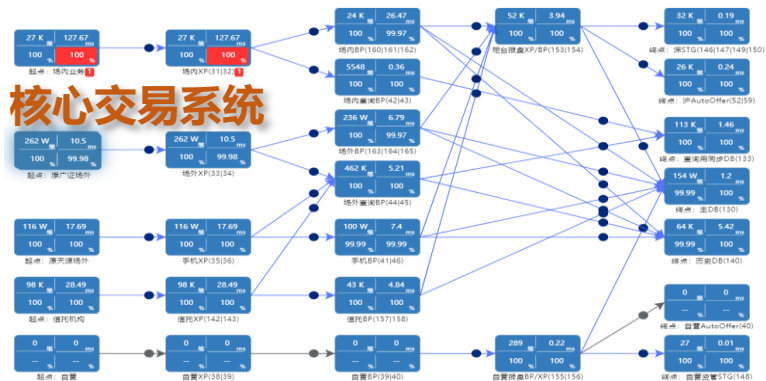


核心交易系统

KCBP进程



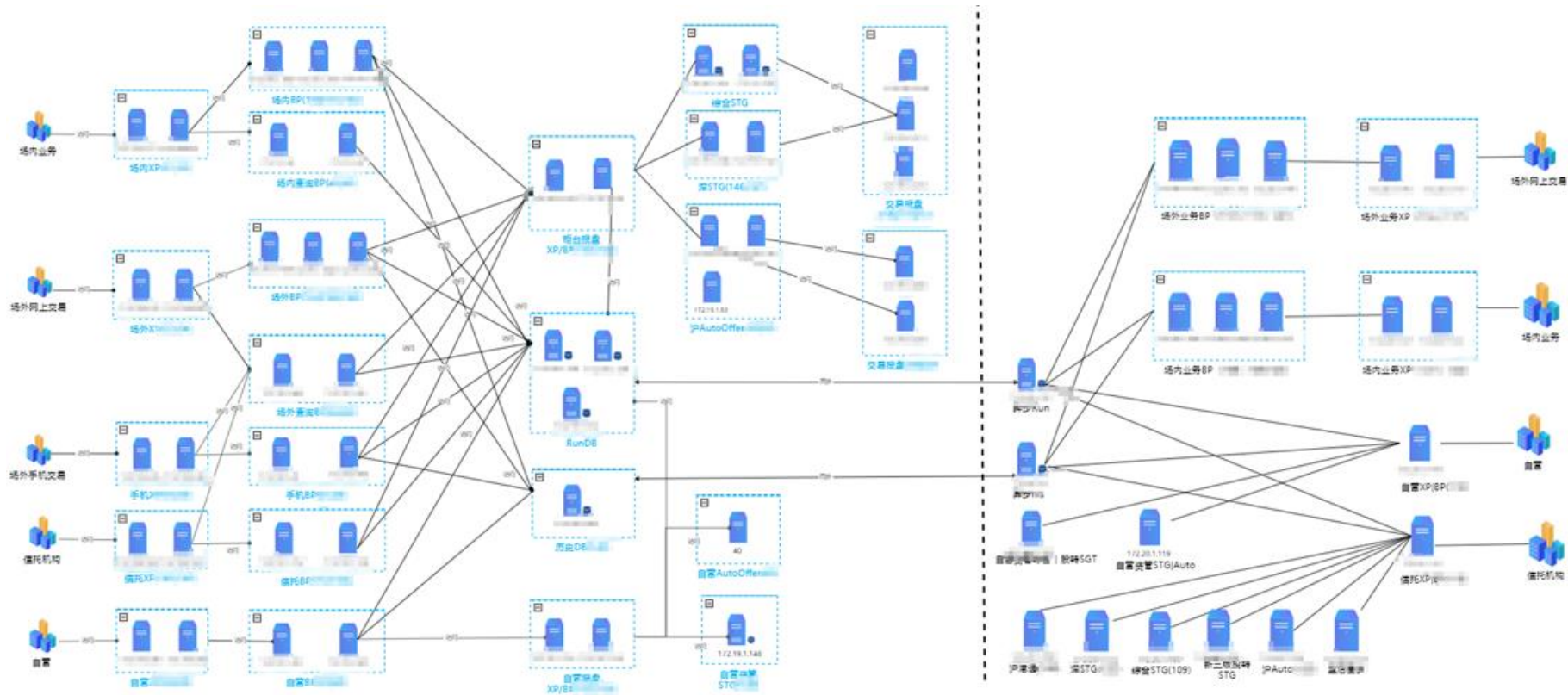
核心交易系统



核心交易系统

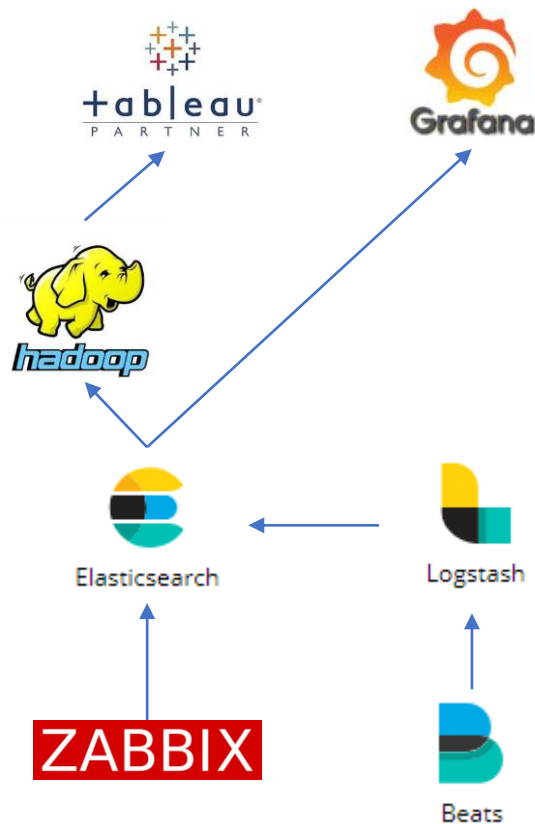
一体化监控平台——监控效果

DOIS



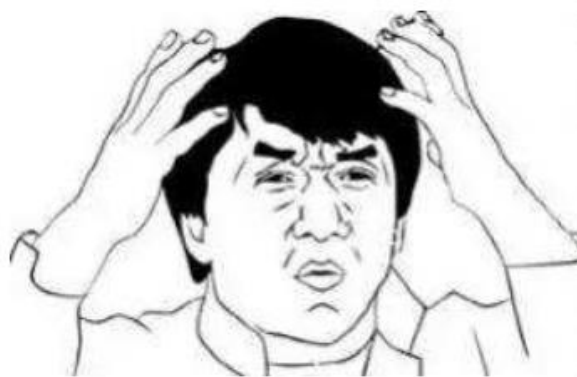
谈谈工具的选择

- Zabbix **or** Prometheus
- ElasticStack **or** InfluxData
- Grafana **or** 自开发监控视图
- Tableau **or** 自开发报表
- Omnibus **or** 自开发告警系统



几个问题探讨一下

- 是否值得开发统一界面进行监控策略的定义？
 - KCMM是封闭系统，无法提供开发接口
 - 究竟谁会经常去调整监控策略？
- 对于券商来说，怎样才是好的告警通知？
 - 微信、短信、邮件
 - 通知的时效性非常重要
- 重视推广
 - 改变每个人运维习惯不是那么容易的事情，要持续努力
 - 借鉴互联网公司那种做事风格：主动、把事情做在前面



目 录

1

从0构建运维平台 —— 先解决温饱问题



2

基于微服务的运维开发框架 —— 满足精细化要求

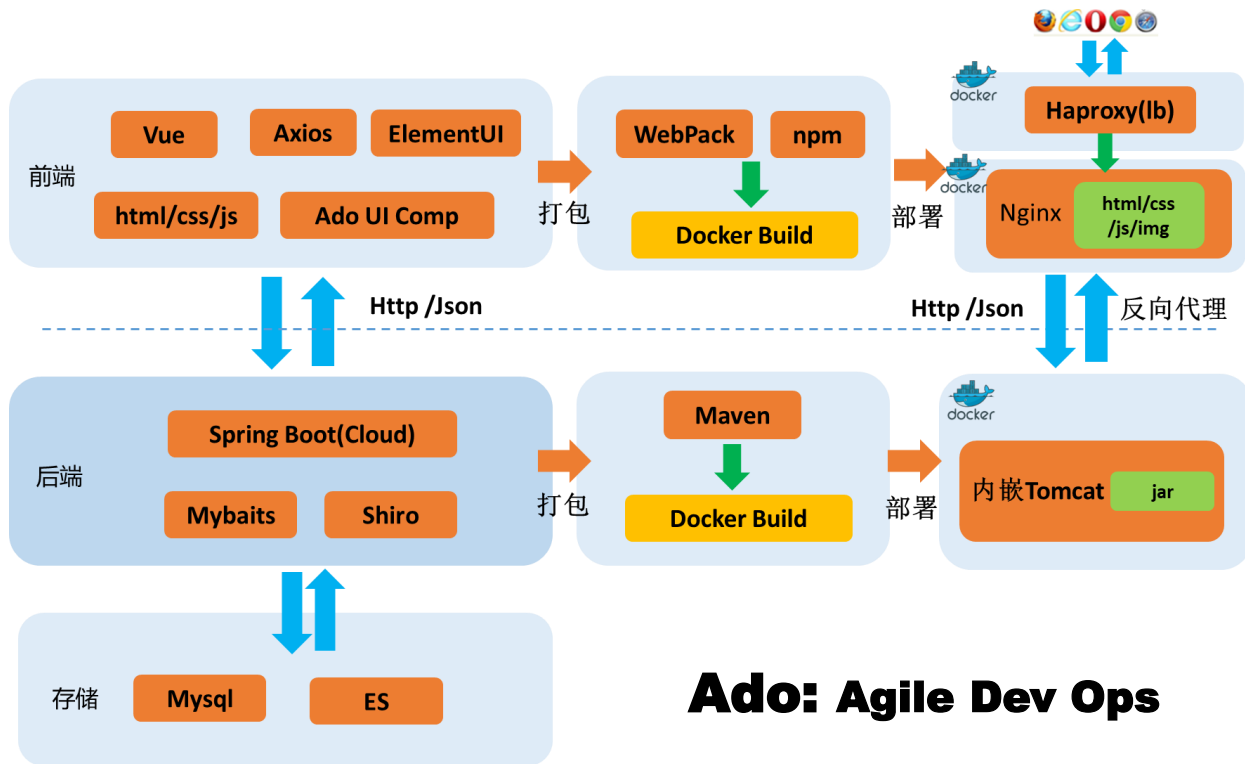
3

DevOps持续集成 —— 走向开发&运维

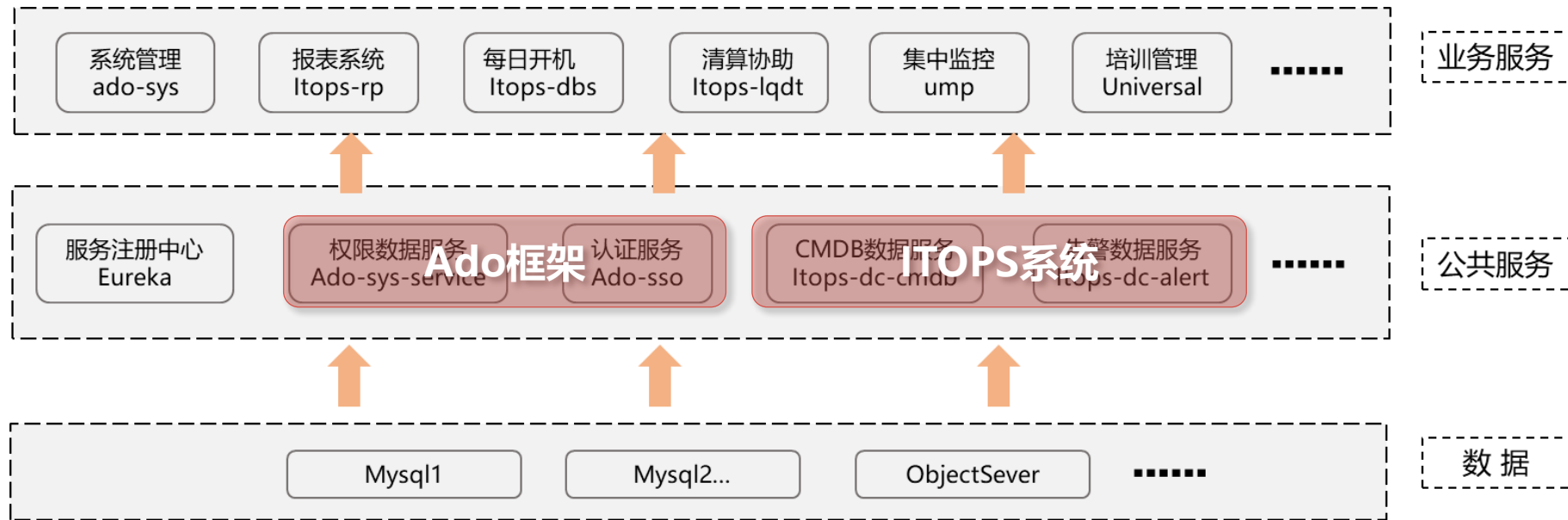
基于微服务的运维开发框架：爱豆-Ado

DOIS

- 运维精细化程度提高，个性化需求增加（有新需求，但基于厂家的平台去修改很困难）
- 从成本考虑，运维平台自主可控是发展趋势
- 技术和技能的内部积累&共享
- 内部业务系统的开发也需要一个良好的框架
- 前后端完全分离
- 采用当下最成熟的开源框架和组件模块，适当进行了扩展

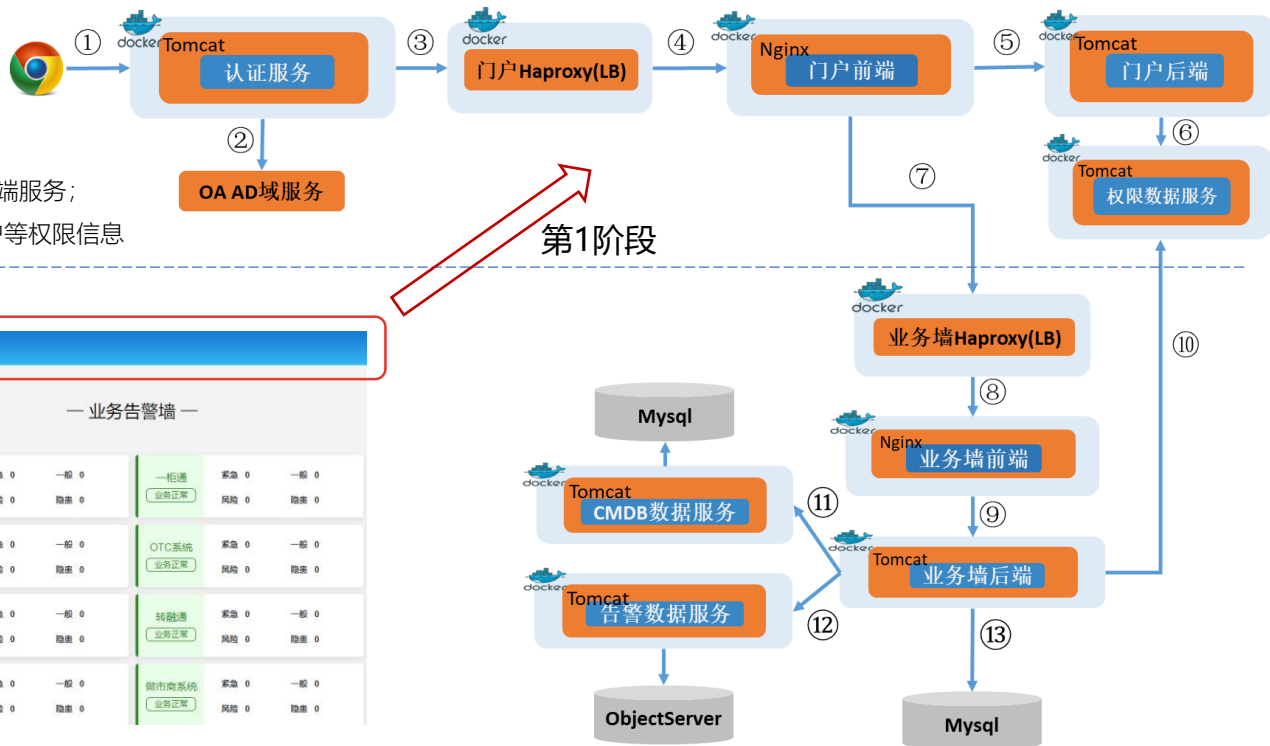


服务组成和拆分的示例

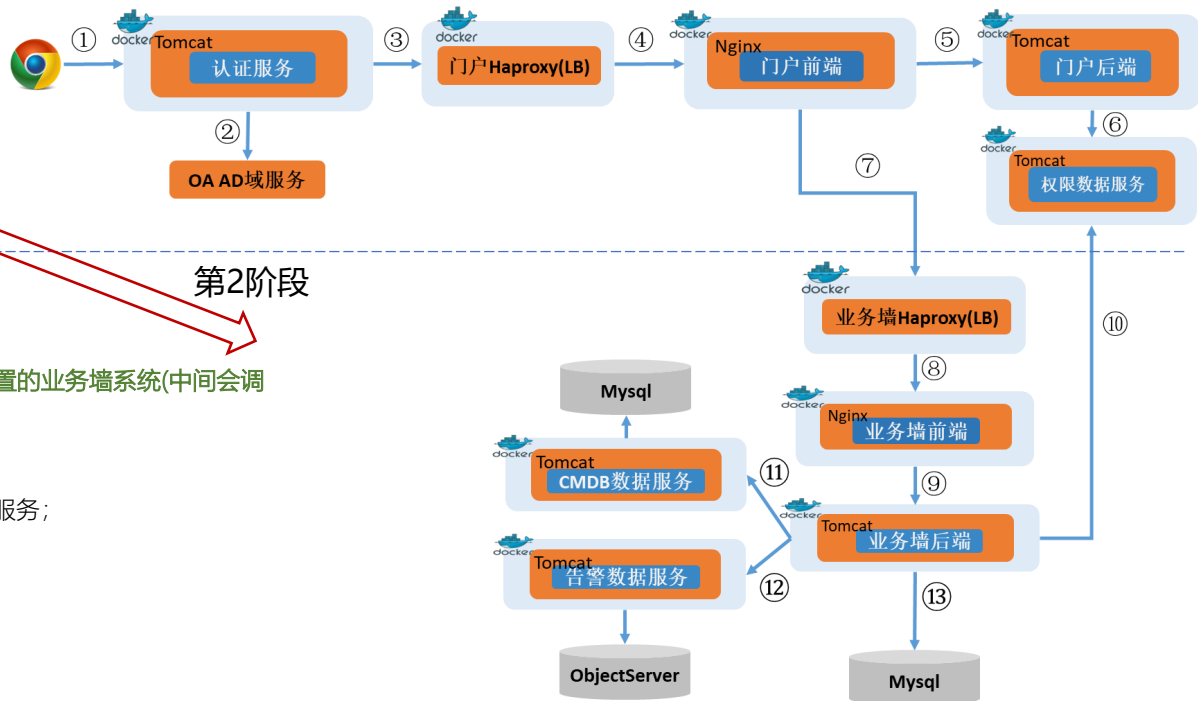


服务间调用的示例：告警业务墙

- ① 发起访问;
- ② 认证服务调用AD域服务验证OA账号密码;
- ③ 验证成功后访问门户的接入层(Haproxy);
- ④ 门户接入层转发访问请求到门户前端;
- ⑤ 门户前端服务通过Nginx反向代理转发到门户后端;
- ⑥ 登录后调用权限数据服务加载菜单、权限、用户等权限信息

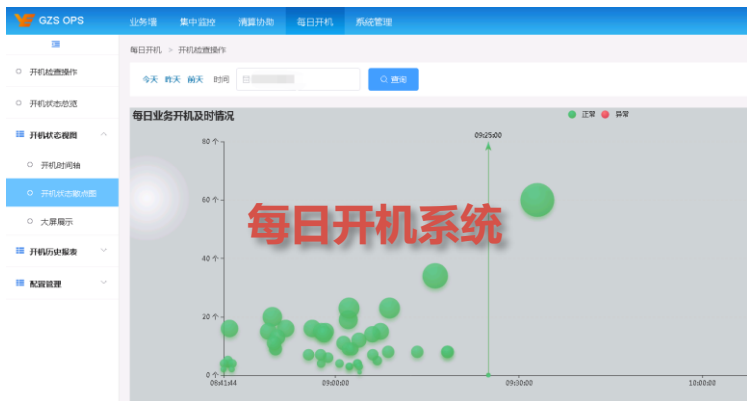


服务间调用的示例：告警业务墙



- 第2阶段
- ⑦ 门户前端加载到菜单后，默认加载处于第一个菜单位置的业务墙系统(中间会调用认证服务进行单点登录);
 - ⑧ 业务墙接入层转发访问请求到业务墙前端服务;
 - ⑨ 业务墙前端服务通过反向代理转发请求到业务墙后端服务;
 - ⑩ 成功访问业务墙后端，加载菜单、权限、用户等信息;
 - ⑪ 调用cmdb数据服务加载业务信息;
 - ⑫ 调用告警数据服务加载告警信息;
 - ⑬ 后端应用服务操作访问数据库;

运维开发框架 —— 业务应用举例



关于开发框架的一点建议

DOIS

- 一开始不要追求开发框架的完美
 - 适合的才是正确的（暂时不要做那种所见即所得的开发平台）
 - 快速迭代（Ado迭代了3次）
 - 在重要功能好用的情况下，不断增加新功能或者服务（持续搭积木）
- 尽快基于开发框架做出几个样板系统
 - 验证开发框架，发现问题，然后优化
 - 让大家看到框架背后的业务价值，进而愿意去使用它
- 积累公共服务或模块
 - 开发过程中，注意提取可能会被共用的部分
 - 公共服务尽可能简单（微服务）
- 重视培训
 - 进行了2次，每次3个下午的开发培训，培养框架使用的火种
 - 过程中听取大家的建议



目 录

1

从0构建运维平台 —— 先解决温饱问题

2

基于微服务的运维开发框架 —— 满足精细化要求



3

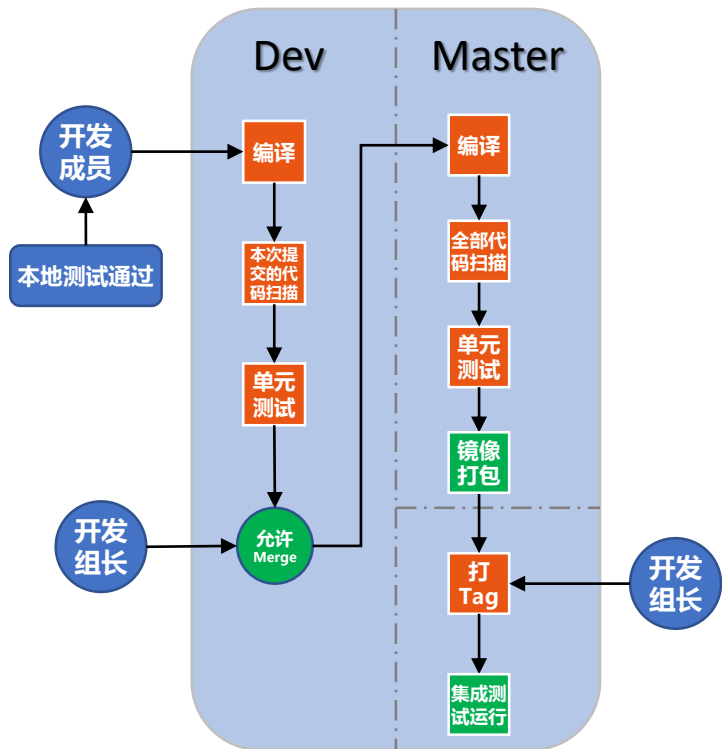
DevOps持续集成 —— 走向开发&运维

DevOps持续集成&持续交付

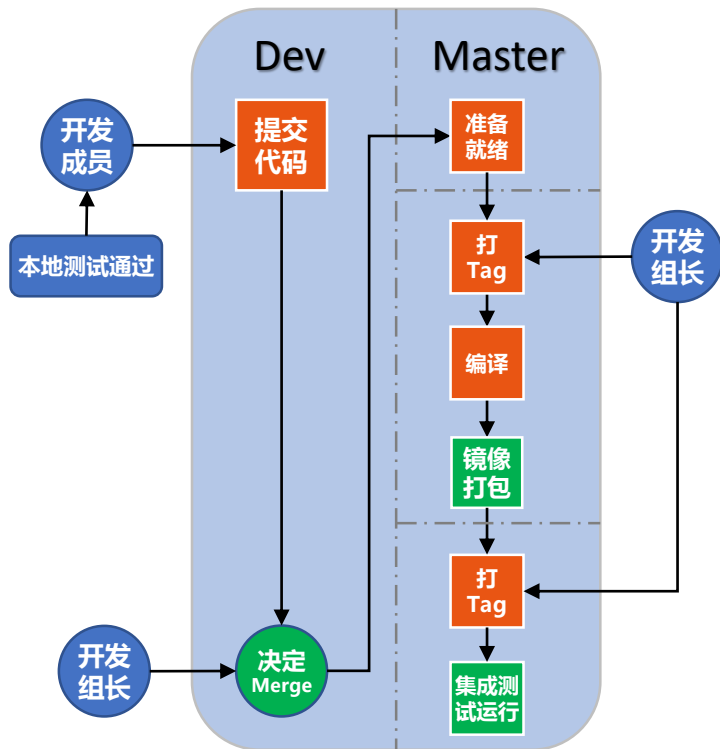


目前DevOps实现的应用：每日开关机业务、清算辅助系统、大屏展现、监控平台封装、内部培训辅助程序、CMDB后台服务、单点登录后台服务等

集成&交付过程的选择 —— 流程求完整，还是求适用？

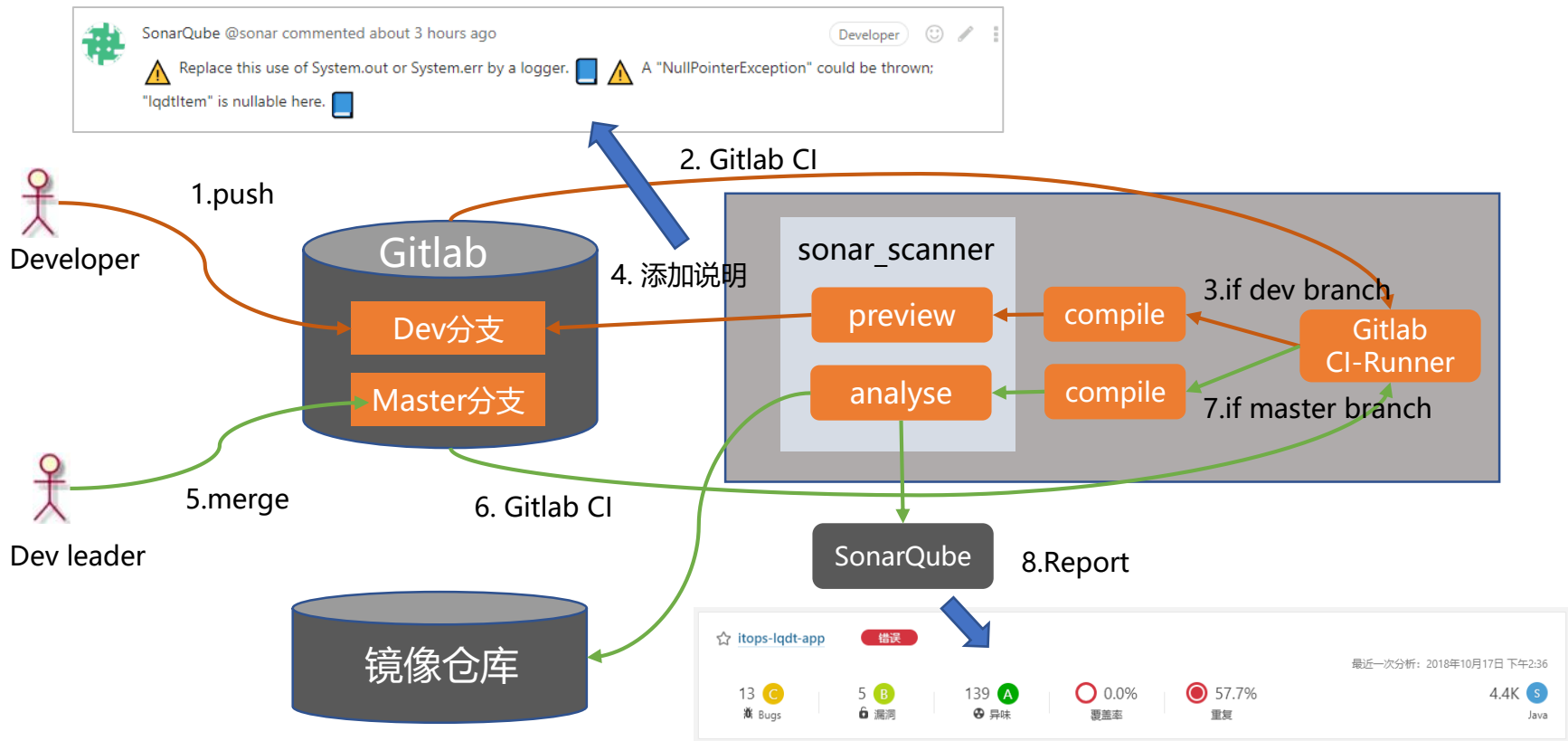


过程1

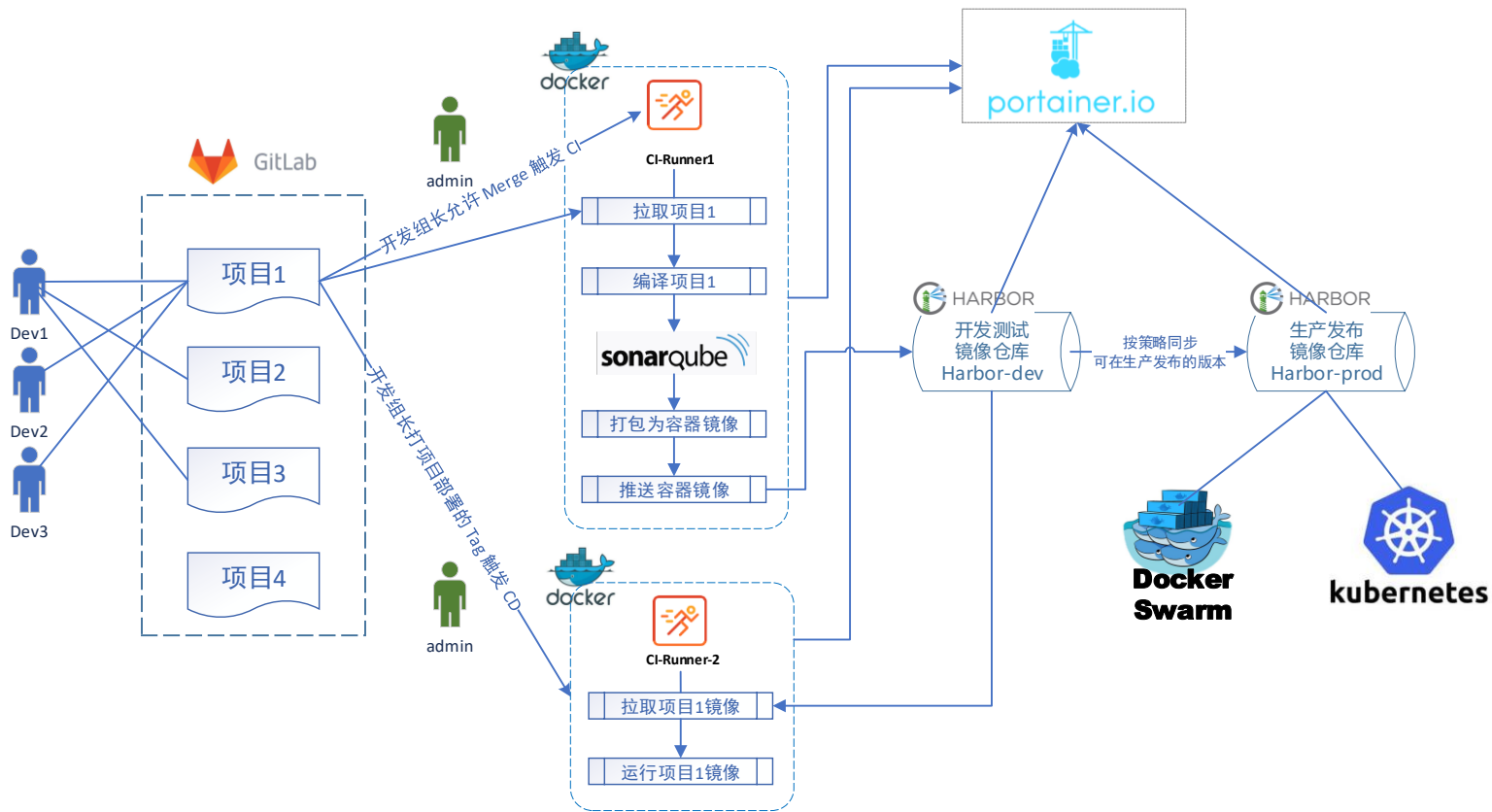


过程2

集成和交付过程的选择 —— 选择适用，同时考虑扩展



DevOps组件和架构



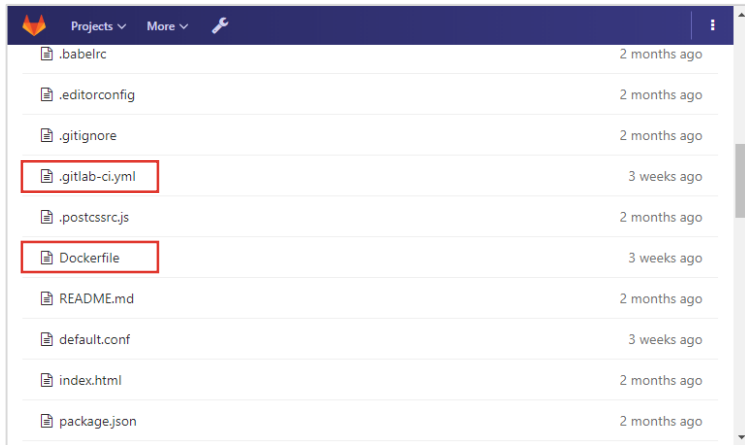
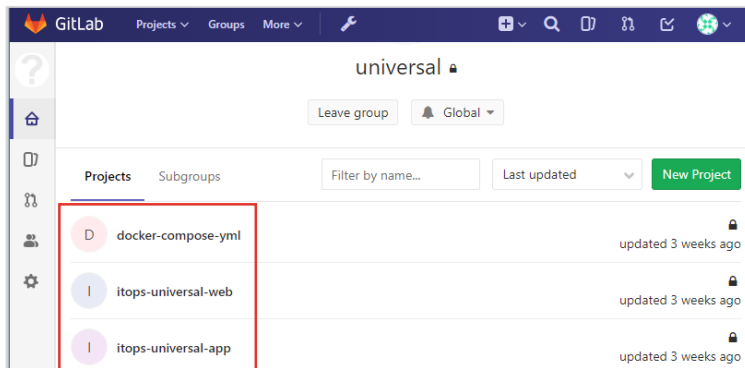
过程举例说明

Step1-建立Gitlab项目群组

- Web前端建立**xxx-web**项目，用于前端开发
- 后端建立**xxx-app**项目，用于后端开发
- 再建立**xxx-compose.yml**项目，用于发布到开发测试环境

项目群组的每个子项目都有两个重要的文件

- **.gitlab-ci.yml**: 通过tag进行构建
- **Dockerfile**: 打包前端、后端程序为容器镜像
- 这两个文件都由**项目负责人**来维护



过程举例说明

Dockerfile:

- 从开发测试镜像库 (Harbor) 拉取基础镜像
- 修改相关程序配置和镜像配置

注: 类似于Springboot的针对开发/生产不同环境的配置区别, 可以在compose文件的环境变量中定义, 以便在通过Swarm等工具进行发布时指定:

environment:

- JAVA_OPTS=-Dspring.profiles.active=test

```
Dockerfile 336 Bytes
1 FROM dk-repo-dev.gzs.com/common/java:8
2 ENV JAVA_OPTS=-Dspring.profiles.active=test
3 COPY ./target/itops-lqdt-app-0.0.2-SNAPSHOT.jar /myapp/
4 WORKDIR /myapp
5 RUN /bin/cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime && echo 'Asia/Shanghai' >/etc/
6 ENTRYPOINT ["sh","-c","java $JAVA_OPTS -jar itops-lqdt-app-0.0.2-SNAPSHOT.jar"]
```

.gitlab-ci.yml:

- TAG可作为发布版本的标识: `$CI_COMMIT_TAG`
- 镜像推送到开发测试镜像库 (Harbor)

```
.gitlab-ci.yml 889 Bytes
1 stages:
2   - build_push_img
3   - code_analyze
4   - code_preview
5
6 before_script:
7   - docker login -u "$HARBOR_DEV_USER" -p "$HARBOR_DEV_PASSWORD" "$HARBOR_DEV_SVR"
8
9 sonar_analyze:
10  stage: code_analyze
11  script:
12    - mvn clean package
13    - sonar-scanner -X
14  tags:
15    - dev-32
16  only:
17    - master
18
19 sonar_preview:
20  stage: code_preview
21  script:
22    - mvn clean package
23    - sonar-scanner -Dsonar.analysis.mode=preview -Dsonar.gitlab.commit_sha=$CI_BUILD_REF -Dsonar.gitlab
24  tags:
25    - dev-32
26  except:
27    - master
28
29 build_push_img:
30  stage: build_push_img
31  script:
32    - mvn clean package
33    - docker build --pull -t "$HARBOR_DEV_SVR/$HARBOR_DEV_PROJ/itops-lqdt-app:$CI_COMMIT_TAG" .
34    - docker push "$HARBOR_DEV_SVR/$HARBOR_DEV_PROJ/itops-lqdt-app:$CI_COMMIT_TAG"
```

过程举例说明



Step2 - 开发人员各自提交代码到Dev分支

注: Dev分支的自动编译、代码检查等环节略过, 同下面的步骤

Step3- Gitlab项目负责人

- 第2步完成后才允许Merge, 项目管理员(开发组长)确认后执行
- CI-Runner是事先定义好的开发测试环境, 可以选择不同的机器作为Runner

Step4- CI Runner

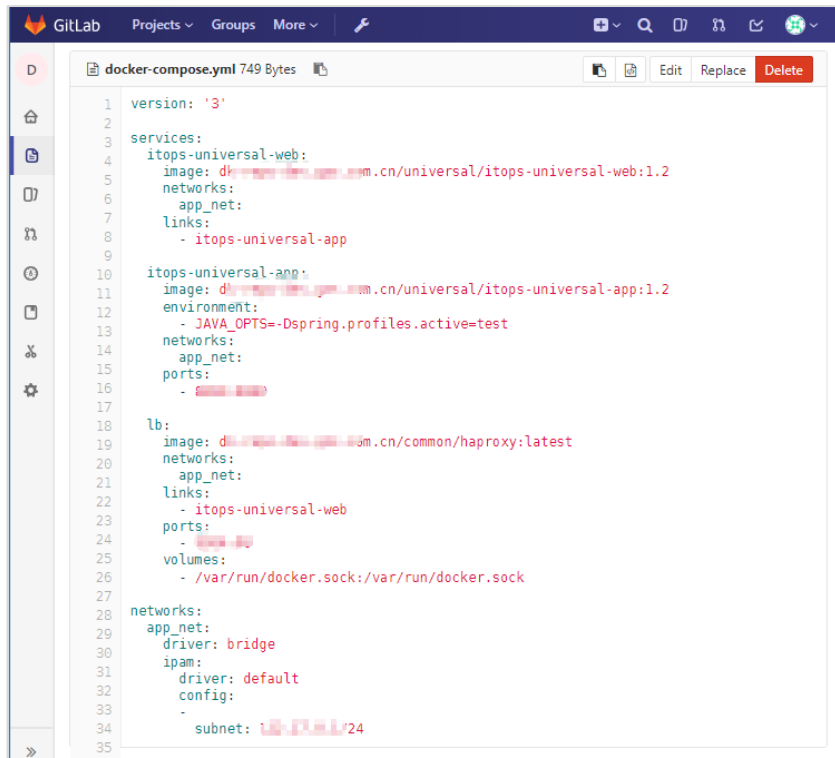
- 运行在开发测试机器上的Runner根据pipeline从Gitlab上拉取对应的项目, 然后编译、语法检查、打包, 并push到开发测试的Harbor镜像库

```
1 stages:
2   - build_push_img
3
4 before_script:
5   - docker login -u "$HARBOR_DEV_USER" -p "$HARBOR_DEV_PASSWORD" "$HARBOR_DEV_SVR"
6
7 build_push_img:
8   stage: build_push_img
9   script:
10    - cnpm install
11    - cnpm run build
12    - docker build --pull -t "$HARBOR_DEV_SVR/$HARBOR_DEV_PROJ/itops-universal-web:$CI_COMMIT_TAG" .
13    - docker push "$HARBOR_DEV_SVR/$HARBOR_DEV_PROJ/itops-universal-web:$CI_COMMIT_TAG"
14
15 tags:
16   - dev-32
17
18 only:
19   - tags
```

过程举例说明

Step5- Gitlab项目负责人

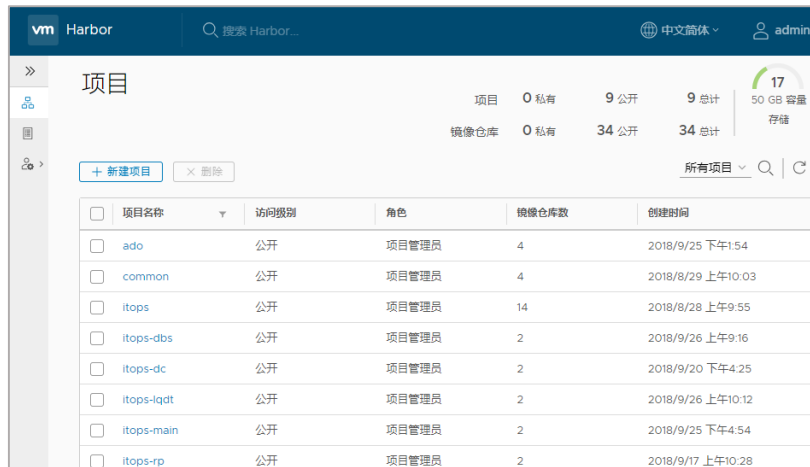
- 当前后端开发都完成了开发测试打包为镜像，则对xxx-compose-yml项目发起pipeline，在修改compose文件的docker image对应版本之后，通过打tag触发部署流程，部署到测试环境进行**集成测试**



```
1 version: '3'
2
3 services:
4   itops-universal-web:
5     image: d[redacted].cn/universal/itops-universal-web:1.2
6     networks:
7       app_net:
8         links:
9           - itops-universal-app
10
11   itops-universal-app:
12     image: d[redacted].cn/universal/itops-universal-app:1.2
13     environment:
14       - JAVA_OPTS=-Dspring.profiles.active=test
15     networks:
16       app_net:
17     ports:
18       - [redacted]
19
20   lb:
21     image: d[redacted].cn/common/haproxy:latest
22     networks:
23       app_net:
24     links:
25       - itops-universal-web
26     ports:
27       - [redacted]
28     volumes:
29       - /var/run/docker.sock:/var/run/docker.sock
30
31 networks:
32   app_net:
33     driver: bridge
34     ipam:
35       driver: default
36       config:
37         - subnet: 1[redacted].24
```

xxx-compose-yml项目下的compose部署文件

使用Harbor做容器镜像管理



<input type="checkbox"/>	项目名称	访问级别	角色	镜像仓库数	创建时间
<input type="checkbox"/>	ado	公开	项目管理员	4	2018/9/25 下午1:54
<input type="checkbox"/>	common	公开	项目管理员	4	2018/8/29 上午10:03
<input type="checkbox"/>	itops	公开	项目管理员	14	2018/8/28 上午9:55
<input type="checkbox"/>	itops-dbs	公开	项目管理员	2	2018/9/26 上午9:16
<input type="checkbox"/>	itops-dc	公开	项目管理员	2	2018/9/20 下午4:25
<input type="checkbox"/>	itops-lqdt	公开	项目管理员	2	2018/9/26 上午10:12
<input type="checkbox"/>	itops-main	公开	项目管理员	2	2018/9/25 下午4:54
<input type="checkbox"/>	itops-rp	公开	项目管理员	2	2018/9/17 上午10:28

Gitlab按照项目推送镜像到
开发测试镜像库



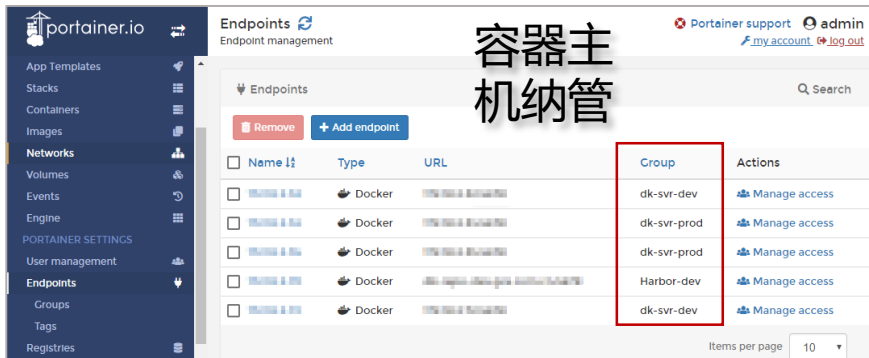
名称	项目	描述	目标名	触发模式
<input type="radio"/> dev2prod	universal	从70开发环境, 复制到71生产环境	172.18.2.71-prod	Scheduled
<input type="radio"/> ado2prod	ado	-	172.18.2.71-prod	Manual
<input type="radio"/> itops-dbs2prod	itops-dbs	itops-dbs 复制到 生成库	172.18.2.71-prod	Manual
<input type="radio"/> itops-dc2prod	itops-dc	itops-dc复制到生成库	172.18.2.71-prod	Manual
<input type="radio"/> itops-lqdt2prod	itops-lqdt	itops-lqdt到生产库	172.18.2.71-prod	Manual

从开发测试镜像库通过手工或者定时
同步相应版本的镜像到生产环境镜像库

通过Portainer进行容器的管理

DOIS

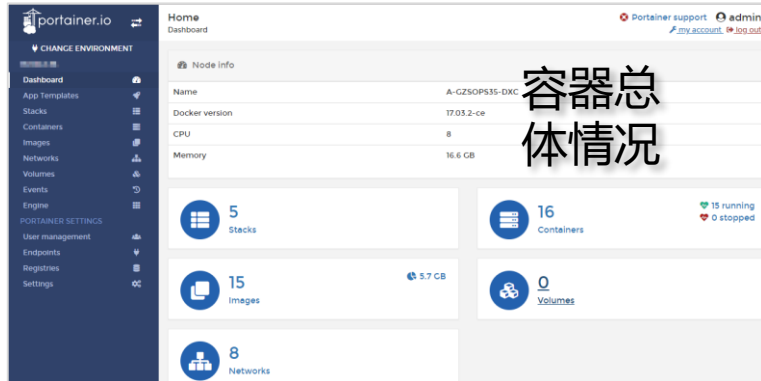
容器主机纳管



The screenshot shows the 'Endpoints' management page in Portainer. A table lists several endpoints, all of which are Docker engines. The 'Group' column is highlighted with a red box, showing groups like 'dk-svr-dev', 'dk-svr-prod', and 'Harbor-dev'. The interface includes a sidebar with navigation options and a top bar with user information.

Name	Type	URL	Group	Actions
dk-svr-dev	Docker	https://192.168.1.10:2376	dk-svr-dev	Manage access
dk-svr-prod	Docker	https://192.168.1.11:2376	dk-svr-prod	Manage access
dk-svr-prod	Docker	https://192.168.1.12:2376	dk-svr-prod	Manage access
Harbor-dev	Docker	https://192.168.1.13:2376	Harbor-dev	Manage access
dk-svr-dev	Docker	https://192.168.1.14:2376	dk-svr-dev	Manage access

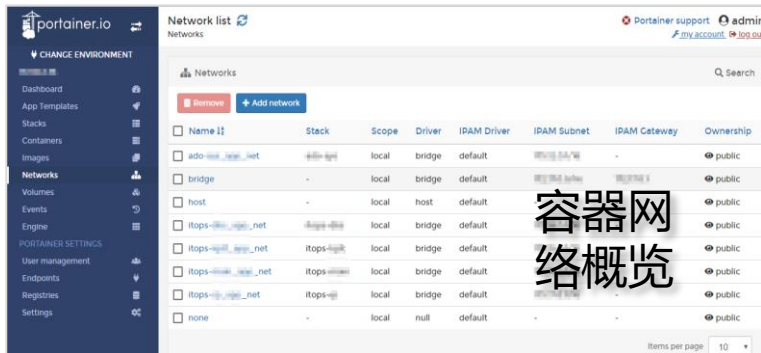
容器总体情况



The screenshot shows the 'Home' dashboard in Portainer. It provides an overview of the system's status, including the number of stacks (5), containers (16), images (15), and networks (8). The dashboard also displays the Docker version (17.03.2-ce) and the total memory usage (5.7 GB).

Category	Count	Status
Stacks	5	
Containers	16	15 running, 1 stopped
Images	15	
Networks	8	

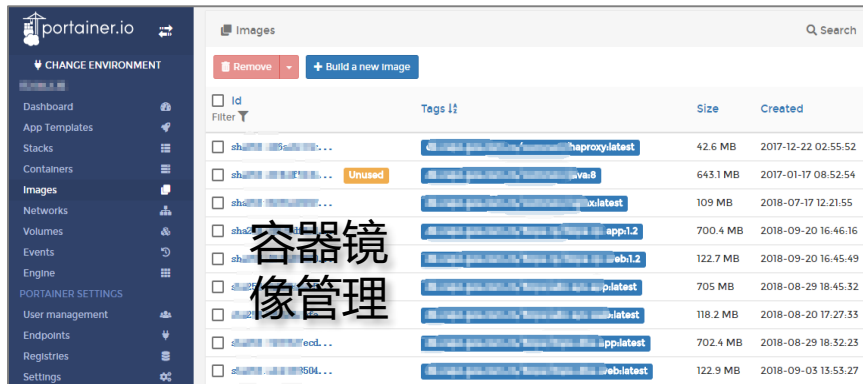
容器网络概览



The screenshot shows the 'Network list' page in Portainer. It displays a table of all networks configured in the system. The table includes columns for Name, Stack, Scope, Driver, IPAM Driver, IPAM Subnet, IPAM Gateway, and Ownership.

Name	Stack	Scope	Driver	IPAM Driver	IPAM Subnet	IPAM Gateway	Ownership
ado-ssl_net	ado-ssl	local	bridge	default	192.168.1.10/24	192.168.1.1	public
bridge	-	local	bridge	default	192.168.1.10/24	192.168.1.1	public
host	-	local	host	default	-	-	public
itops-ssl_net	itops-ssl	local	bridge	default	192.168.1.10/24	192.168.1.1	public
itops-ssl_net	itops-ssl	local	bridge	default	192.168.1.10/24	192.168.1.1	public
itops-ssl_net	itops-ssl	local	bridge	default	192.168.1.10/24	192.168.1.1	public
itops-ssl_net	itops-ssl	local	bridge	default	192.168.1.10/24	192.168.1.1	public
itops-ssl_net	itops-ssl	local	bridge	default	192.168.1.10/24	192.168.1.1	public
none	-	local	null	default	-	-	public

容器镜像管理



The screenshot shows the 'Images' management page in Portainer. It displays a table of all images stored in the system. The table includes columns for Id, Filter, Tags, Size, and Created. The 'Unused' status is highlighted for one of the images.

Id	Filter	Tags	Size	Created
ch...	Unused	...	42.6 MB	2017-12-22 02:55:52
ch...	Unused	...	643.1 MB	2017-01-17 08:52:54
ch...	Unused	...	109 MB	2018-07-17 12:21:55
ch...	Unused	...	700.4 MB	2018-09-20 16:46:16
ch...	Unused	...	122.7 MB	2018-09-20 16:45:49
ch...	Unused	...	705 MB	2018-08-29 18:45:32
ch...	Unused	...	118.2 MB	2018-08-20 17:27:33
ch...	Unused	...	702.4 MB	2018-08-29 18:32:23
ch...	Unused	...	122.9 MB	2018-09-03 13:53:27

通过Portainer进行容器的管理

The screenshot shows the Portainer web interface for managing containers. The top bar includes 'Container list', 'Containers', 'Portainer support', and a user profile 'admin' with links for 'my account' and 'log out'. Below the top bar is a search bar and a 'Settings' icon. A row of action buttons (Start, Stop, Kill, Restart, Pause, Resume, Remove, Add container) is visible. The main content is a table of containers with the following columns: Name, State, Quick actions, Stack, Image, IP Address, Published Ports, and Ownership. The table lists several containers, including 'compent_image' (created) and various 'itops' containers (running). The bottom of the interface shows a pagination bar with 'Items per page' set to 10 and page numbers 1 and 2.

Name	State	Quick actions	Stack	Image	IP Address	Published Ports	Ownership
compent_image	created	[Icons]	-	dk-repo.g...cn/c.../nginx	-	-	public
itops-..._lb_1	running	[Icons]	itops-rp	dk-repo.g...cn/c.../haproxy	[IP]	[Ports]	public
itops-..._itops-rp-web_1	running	[Icons]	itops-rp	dk-repo.g...cn/itops-rp-web_1.2	[IP]	-	public
itops-..._itops-rp-app_1	running	[Icons]	itops-rp	dk-repo.g...cn/itops-rp-app_1.2	[IP]	-	public
itops-..._itops-rp-db_1	running	[Icons]	itops-dbs	dk-repo.g...cn/c.../haproxy	[IP]	[Ports]	public
itops-..._itops-rp-db-web_1	running	[Icons]	itops-dbs	dk-repo.g...cn/itops-rp-db-web	[IP]	-	public
itops-..._itops-rp-db-app_1	running	[Icons]	itops-dbs	dk-repo.g...cn/itops-rp-db-app	[IP]	-	public
itops-main-..._lb_1	running	[Icons]	itops-main	dk-repo.g...cn/c.../haproxy	[IP]	[Ports]	public
itops-main-..._itops-main-web_1	running	[Icons]	itops-main	dk-repo.g...cn/itops-main-web	[IP]	-	public
itops-main-..._itops-main-app_1	running	[Icons]	itops-main	dk-repo.g...cn/itops-main-app	[IP]	-	public

Docker单节点以及Swarm的管理Portainer基本够用

关于DevOps的一些体会



- 传统中小金融企业DevOps的驱动力
 - 体量小、工程师文化不足等特点，造成这类企业很难采用自上而下通过文化推动的方式落地DevOps，反而是通过构建相关工具，让大家感受到个中价值才是DevOps的起点
- 适当裁剪DevOps的过程
 - 不是每个企业或者业务系统都适用于完美的DevOps过程，从安全性、适当性考虑，也许部分过程采用才是出路
 - 总之，目标是提高效率，满足业务需求，目的达成就可以了，例如：ops开发的工具系统，dev能用，且这个过程中除非出现故障或者特殊情况，否则dev无需ops干预就能发布应用，那么这个DevOps的一点效果基本就达到了

关于DevOps的一些体会



- 为什么不选择K8S?
 - K8S相关的可选模块太多，需要大量人力和时间的投入
 - 目前我们实际管理的容器环境规模并不大
 - 等待更合适的成熟产品，例如准备试用Rancher2.1
- Dev和Ops的分工边界
 - dev: (代码 + Dockerfile) => image
 - ops: image + compose文件
- 省事vs省时间
 - Gitlab CI Runner使用docker模式比较省事，可以避免各种不同运行时的环境混乱，但执行时间可能会更长

感谢团队!





Thanks

DevOps 时代社区 荣誉出品

想第一时间看到高效运维社区的
最新动态吗?

