



**QCon** 全球软件开发大会  
INTERNATIONAL SOFTWARE  
DEVELOPMENT CONFERENCE

BEIJING 2018

# 《51信用卡在微服务架构下的监控平台架构实践》

演讲者 / 杨帆

# 关于我

- 杨帆
- 51信用卡架构师，主要负责监控系统的设计
- 也参与过私有云，数据库等运维平台的开发
- 喜欢折腾，是个猫奴

# 目录

公司介绍

1



2

微服务的监控

当互金遇上微服务

3



4

51信用卡的应对之道

智能化实践

5



6

未来展望



# 1) 公司介绍



# 中国领先科技金融独角兽

51信用卡，中国领先金融科技独角兽，业务涵盖个人信用管理服务、信用卡科技服务、线上信贷撮合及投资服务三大业务板块，旗下有“51信用卡管家”、“51人品”、“51人品贷”、“给你花”等核心APP，拥有超过1亿激活用户。







## 2) 微服务的监控

# 传统的监控分层



# 传统的解决方案 - Zabbix



## 优点 - 成熟可靠

老牌监控软件，社区成熟，插件众多。



## 优点 - 系统指标全面

各种监控需求都有对应的解决方案，指标很全，还能使用自定义脚本。



## 缺点 - 使用成本很高

加一个监控成本很高，很多运维也只有基本的了解。



## 缺点 - 应用监控接入困难

对于微服务等复杂的架构，接入的方案并不方便。



当传统架构转向微服务后，视角发生改变

# 以服务为维度的监控



# 社区的解决方案





### 3) 当互金遇上微服务



# 微服务监控有什么特点

**服务功能单一**  
每个服务提供的功能单一，所以造成服务的数量非常多。



**应用直接调用复杂**  
每个服务都会调用若干个应用，造成调用关系复杂。



**指标数量多**  
服务数量多造成指标数量多。



**告警数量多**  
每个服务都需要保证监控及告警，造成告警数量多。



# 互金 + 微服务会碰撞出哪些问题？

# 互金下的微服务监控



## 对故障容忍程度低

业务和资金息息相关，互金领域非常重视质量



## 追求全面的监控

对系统的各个角落都需要监控到，资金无小事



## 追求快速的告警

需要第一时间告警，控制资损需要和时间赛跑



## 在复杂的微服务下需要快速诊断

监控需要不但需要告诉你有问题，还需要告诉你哪里有问题



# 51 微服务监控的初期

## Prometheus



拉模式

Http 接口



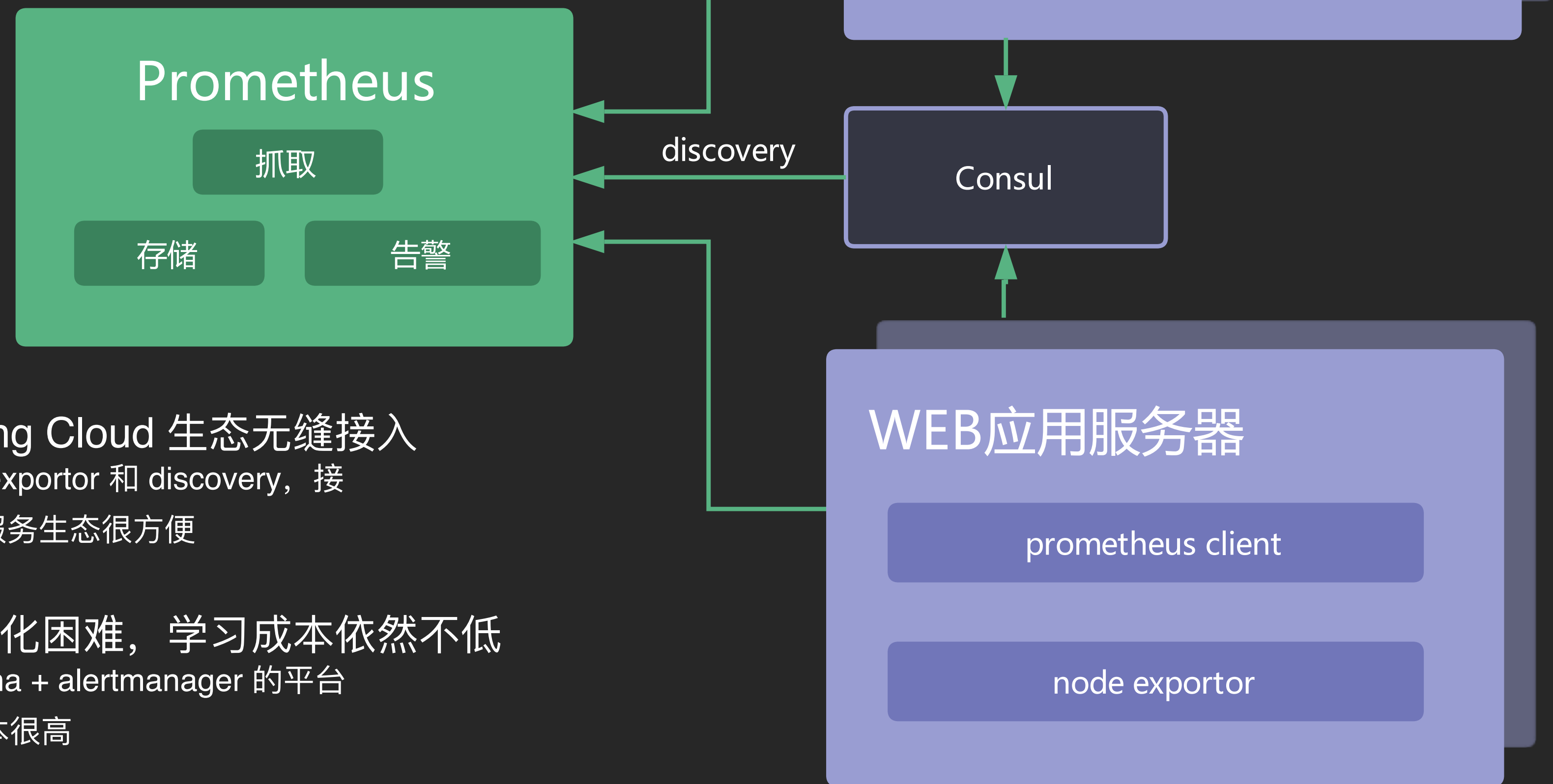
自带告警

Grafana 视图






# Prometheus 下的监控



 采集粒度灵活  
提供 label 的概念，聚合灵活

 Spring Cloud 生态无缝接入  
提供 exporter 和 discovery，接入微服务生态很方便

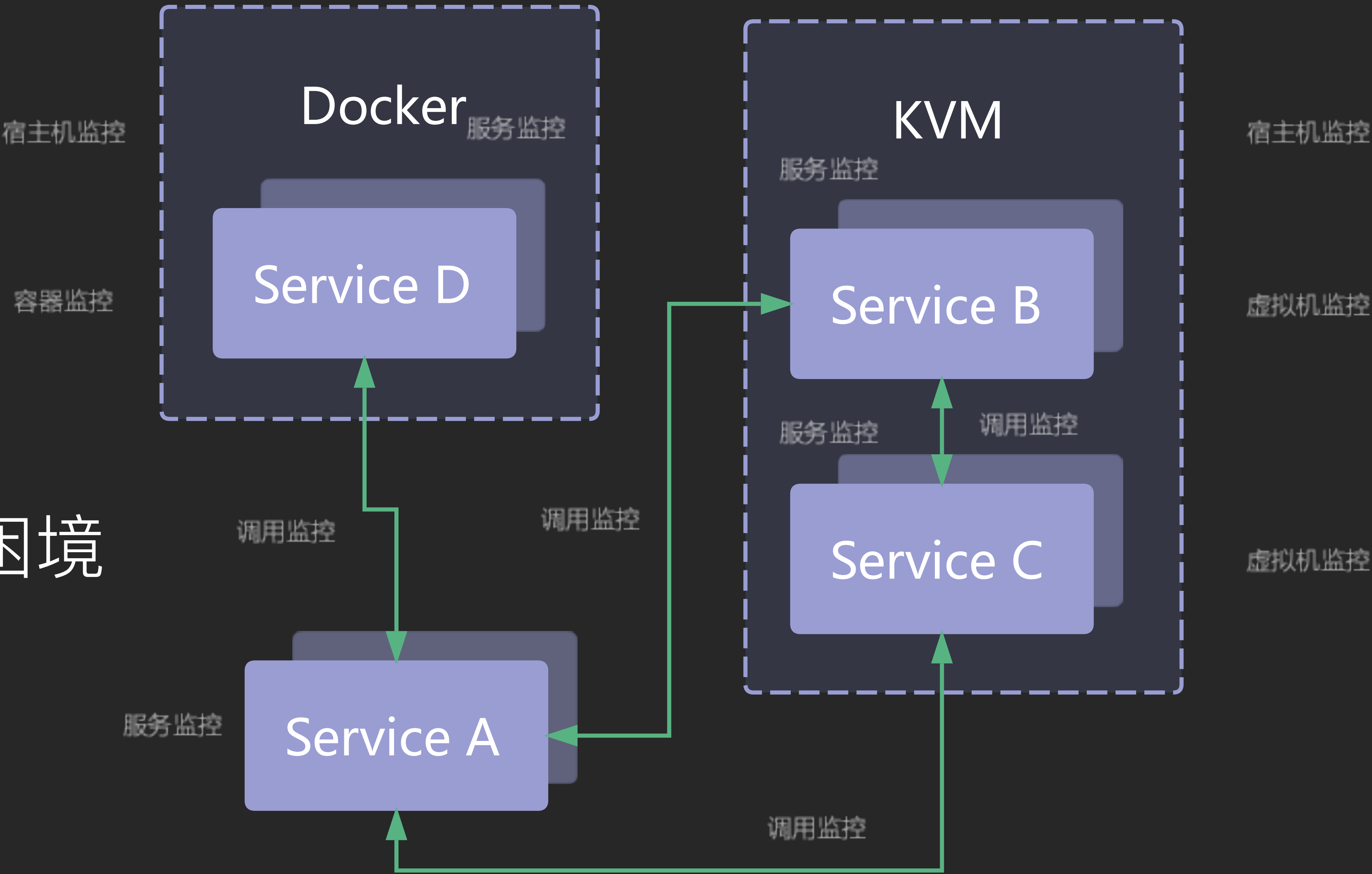
 没有分布式方案  
只能单机运行，聚合、存储和告警都只能单机运行

 平台化困难，学习成本依然不低  
grafana + alertmanager 的平台化成本很高

# 随着服务增长，开始不断踩坑

- 分布式成本很高，40 cores + 256G + ssd 的单机很快碰到性能瓶颈
- 告警诊断的诉求越来越多，但基于单机并不好做
- PromQL 学习成本不低，对新手并不友好
- 拉模式在兼容不同数据源上面开始变得越来越困难

# 微服务监控困境



# 数据构成



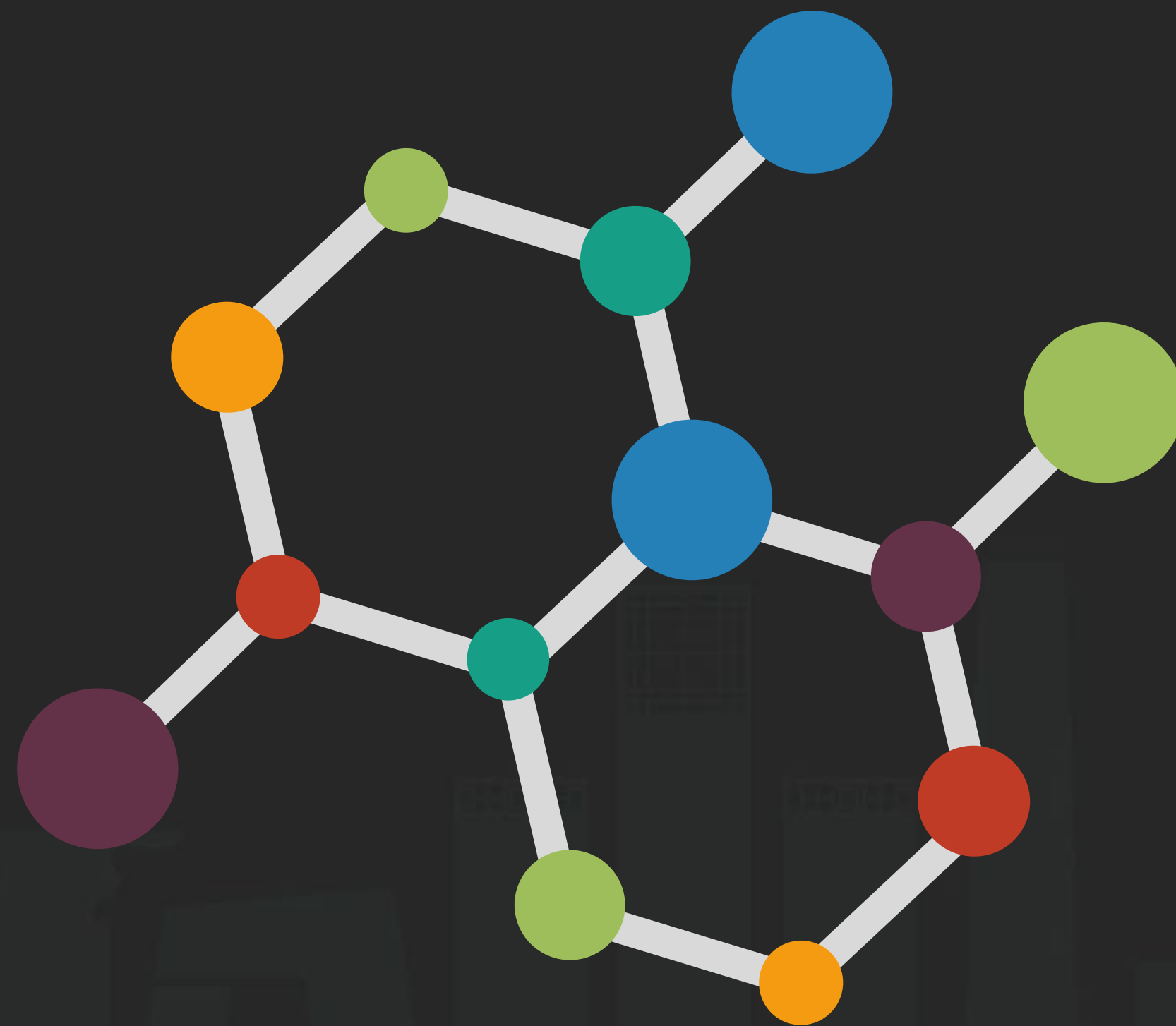
主机性能监控  
宿主机的 CPU、网卡等



应用性能指标  
API、RT、GC 等数据



应用间调用指标  
应用间调用关系，熔断，RT 等  
信息



虚拟机、容器性能监控  
容器的性能指标，也同样包括  
CPU、网卡等



日志指标  
包括系统及应用的错误数，异常  
数等指标



业务指标  
和业务相关指标



# 算笔账， 每增加一个服务

- 一个服务拥有 10 个 API
- 一个服务平均 8 个实例
- 一个服务平均调用 10 个服务
- 每增加一个服务， 增加约 5000 个指标
- 如果再增加维度（例如 dc）， 加快采集频率， 这个数字还得翻几番

# 为什么指标粒度这么细？

- 互金 + 微服务的架构下，我们对监控诊断有更细致的要求
- 我们的目的不止是为了发现问题，而且也要迅速告诉我们问题在哪里
- devops, APM, 监控, 告警这些概念正在不断融合发生化学反应



## 4) 51信用卡的应对之道

# 平台化



# 为什么要平台化?



为新加入的开发人员提供统一的体验



监控、告警以及诊断的一站式服务



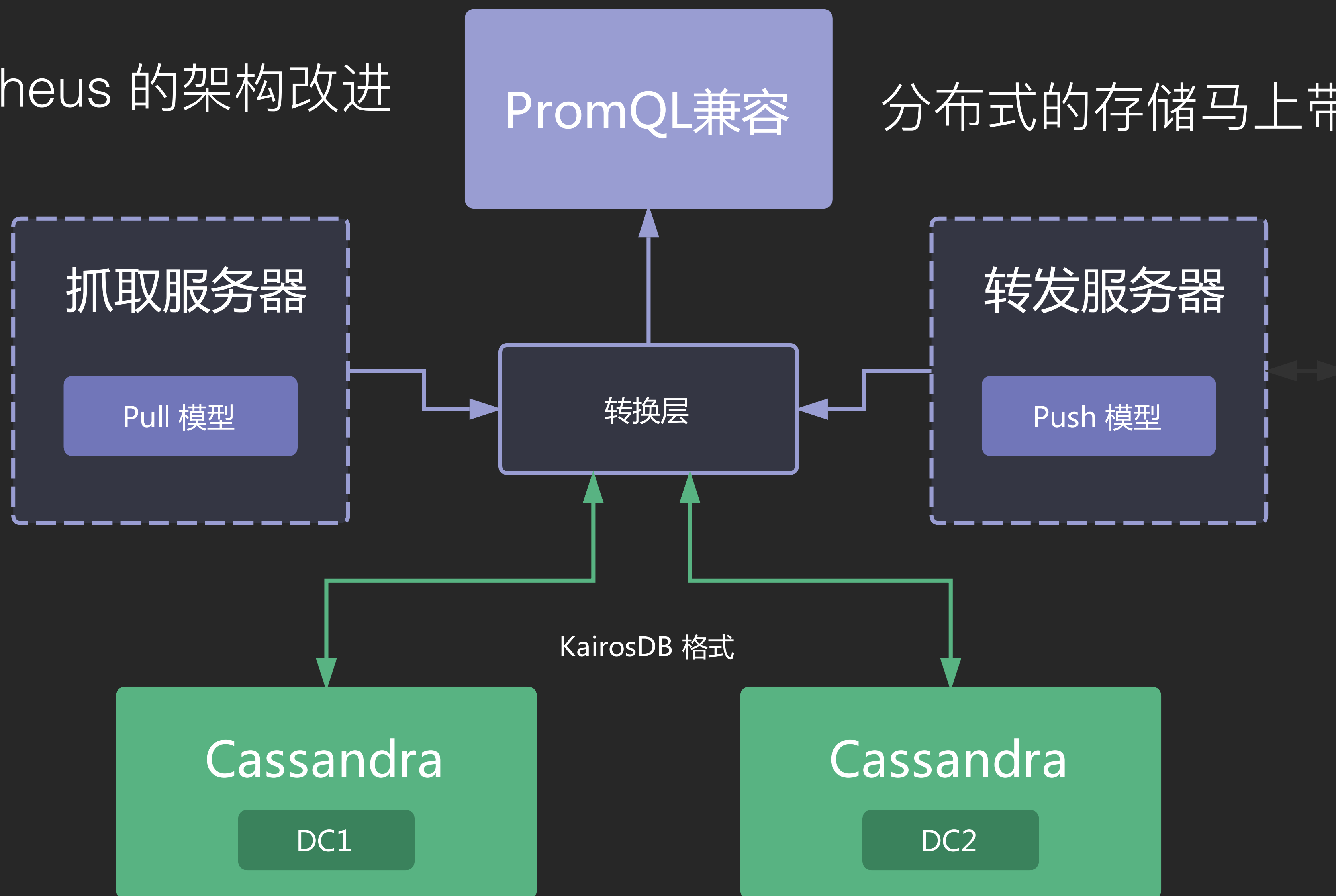
为 AIOps 提供想象力

# 如何构建对上层统一的存储

# 基于 Prometheus 的架构改进

PromQL兼容

分布式的存储马上带来新的问题

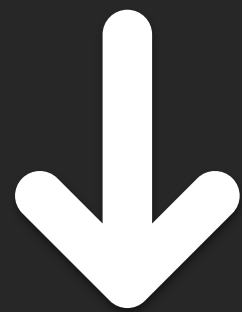


# 告警的性能瓶颈

- Labels 匹配的搜索效率
  - 没有预聚合造成读取的查询瓶颈
  - Metric 过长的查询性能
  - 维度爆炸及维度重复困境
- +无法解决的 LSM 写快读慢

# Labels 匹配的搜索效率

```
sum(increase(service_api_count{port=~"8.*"})) by instance
```



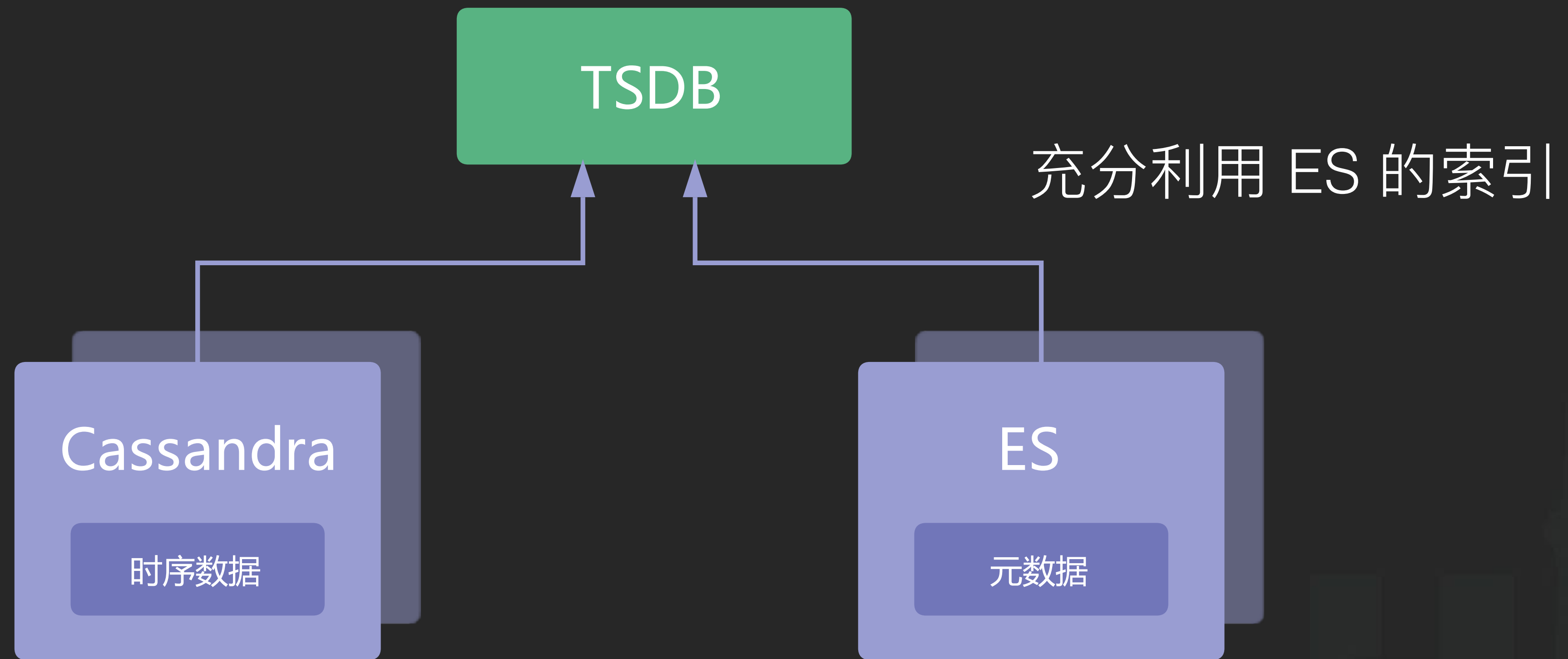
Inverted Index

```
service_api_count{instance="192.168.1.1", port="80"}  
service_api_count{instance="192.168.1.2", port="81"}  
service_api_count{instance="192.168.1.2", port="71"}  
service_api_count{instance="192.168.1.2", port="72"}
```





# 重新构建分布式时序库



# 预聚合数据

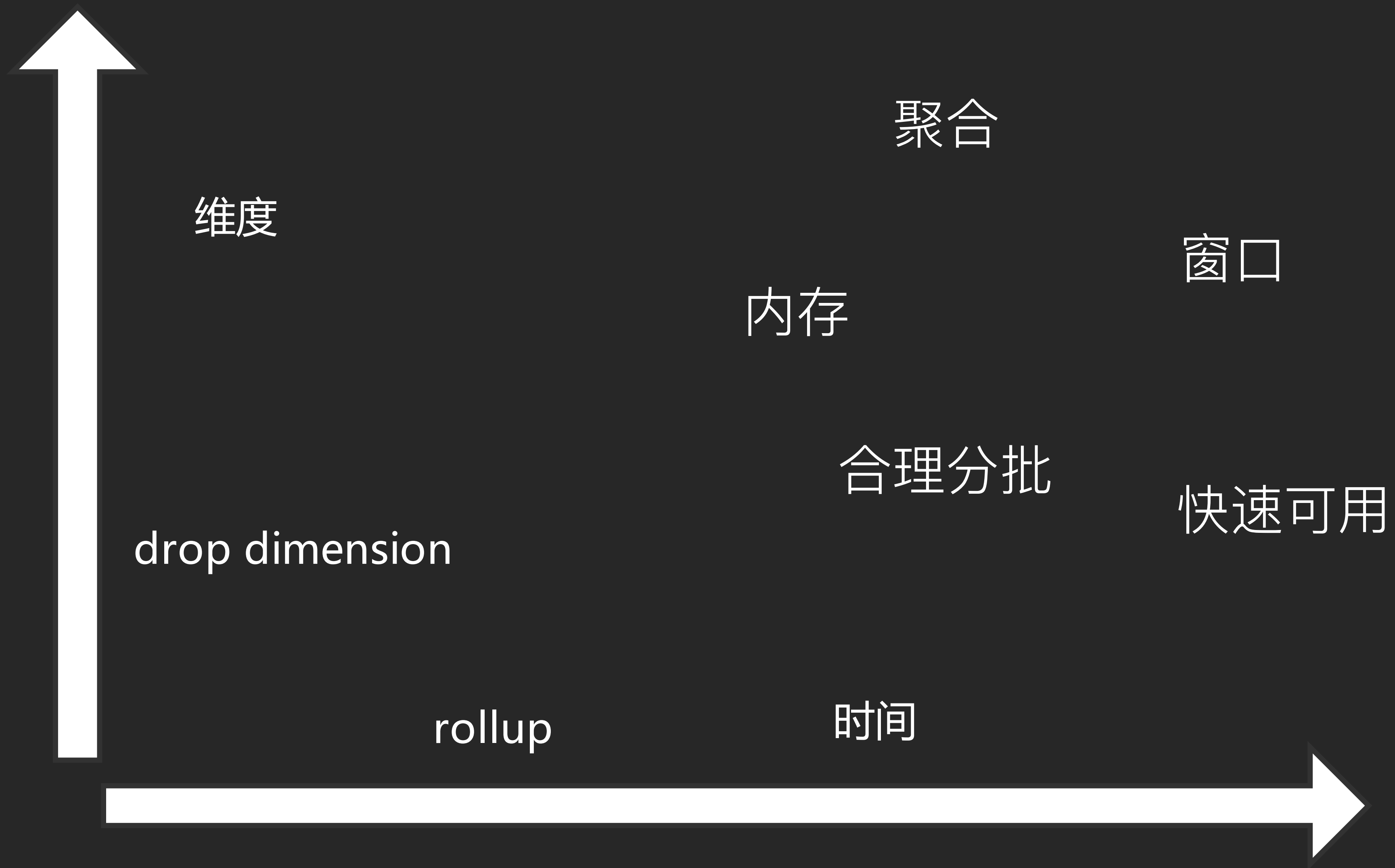
```
sum(increase(service_api_count)) by instance
```



4 次 kv 读取

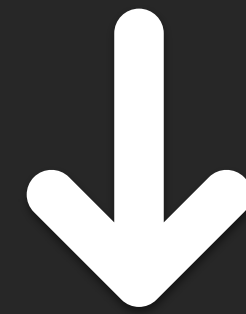
```
service_api_count{instance="192.168.1.1", port="80"}  
service_api_count{instance="192.168.1.2", port="81"}  
service_api_count{instance="192.168.1.2", port="71"}  
service_api_count{instance="192.168.1.2", port="72"}
```

为什么不直接存一份数据？



# Metric 的长度

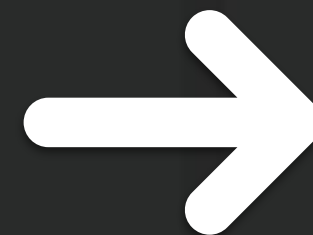
```
service_api_count{instance="192.168.1.1", port="80"}=1.0
```



Cassandra 存储

```
byte(service_api_count+t1+v1+t2+v2),timestamp1,1.0  
byte(service_api_count+t1+v1+t2+v2),timestamp2,2.0
```

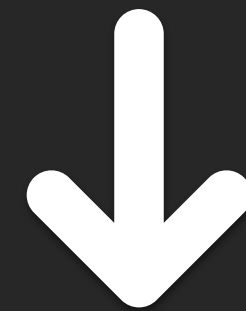
Key 的重复困境



Bitmap 索引

# 维度重复

```
service_api_count{instance="192.168.1.1", port="80"}=1.0  
service_api_time{instance="192.168.1.1", port="80"}=1.0
```



Cassandra 存储

```
byte(service_api_count+t1+v1+t2+v2),timestamp1,1.0  
byte(service_api_time+t1+v1+t2+v2),timestamp1,1.0
```

value 能不能是个复杂类型，比如 Map?

# 解决方案

Bitmap

预聚合

复合  
类型

倒排  
索引

社区已有方案

冷热  
数据

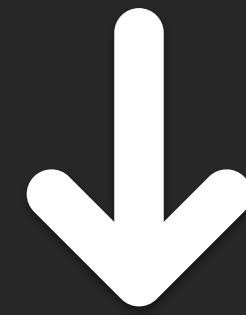


Druid



# 转换

```
service_api_count{instance="192.168.1.1", port="80"}=1.0  
service_api_time{instance="192.168.1.1", port="80"}=1.0
```



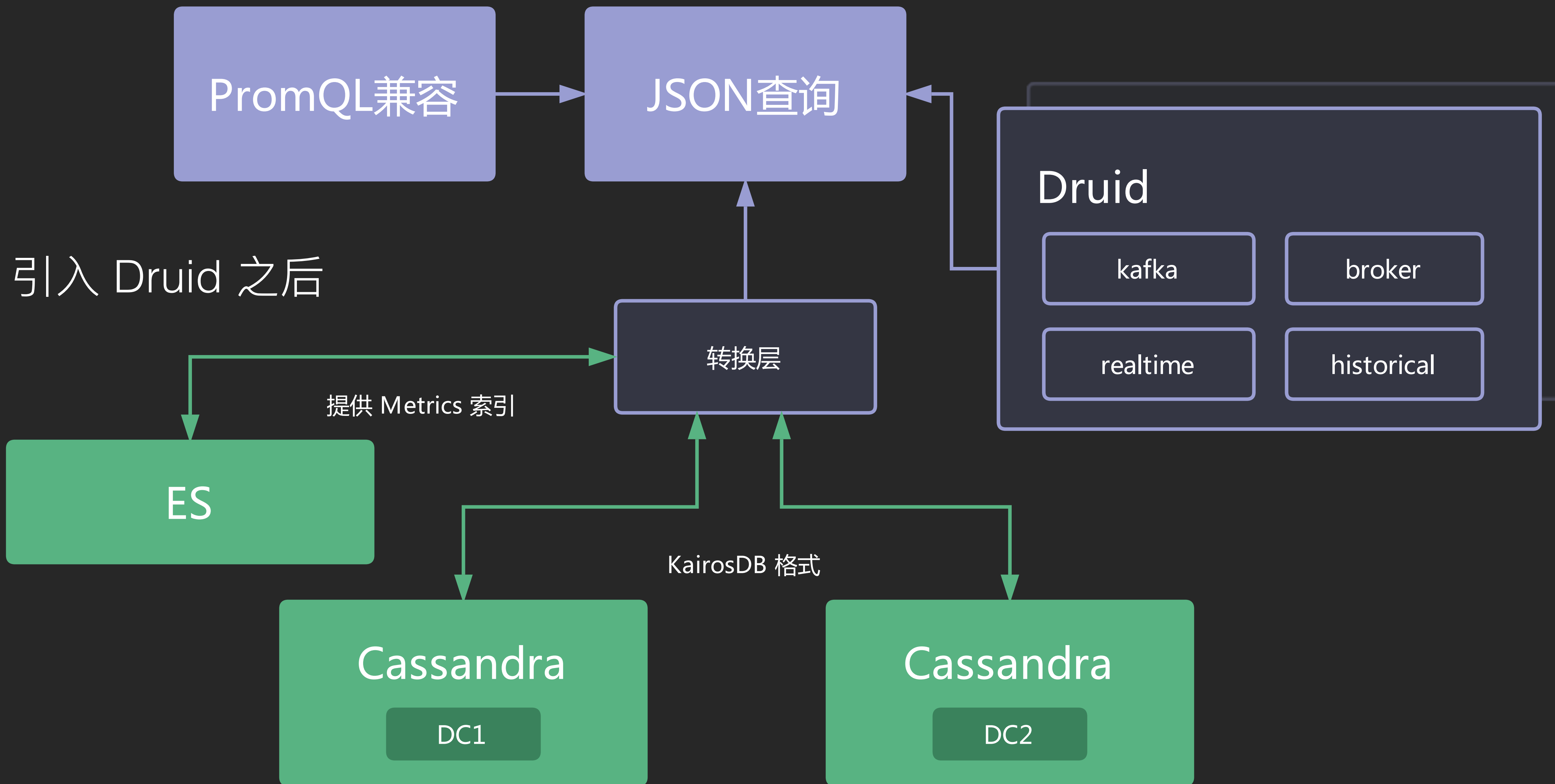
```
{"count": 1.0, "instance": "192.168.1.1", "port": 80, "time": 1.0}
```

doubleSum

维度1

维度2

doubleSum / count





## 5) 智能诊断的一些实践

# 和日志监控的联动

- 取告警的前后时间窗口为范围查询日志
- 以 ERROR, Exception 等关键字做筛选
- 以相似算法做日志排序发现故障原因

# 和链路监控的联动

- 取告警的前后时间窗口为范围查询日志
- 以日志的中的 tracking id 查询链路性能
- 以错误节点做排序发现故障原因



## 6) 未来展望



# 未来

