携程Redis容器化实践

CIS-Sysdev-李剑

目录

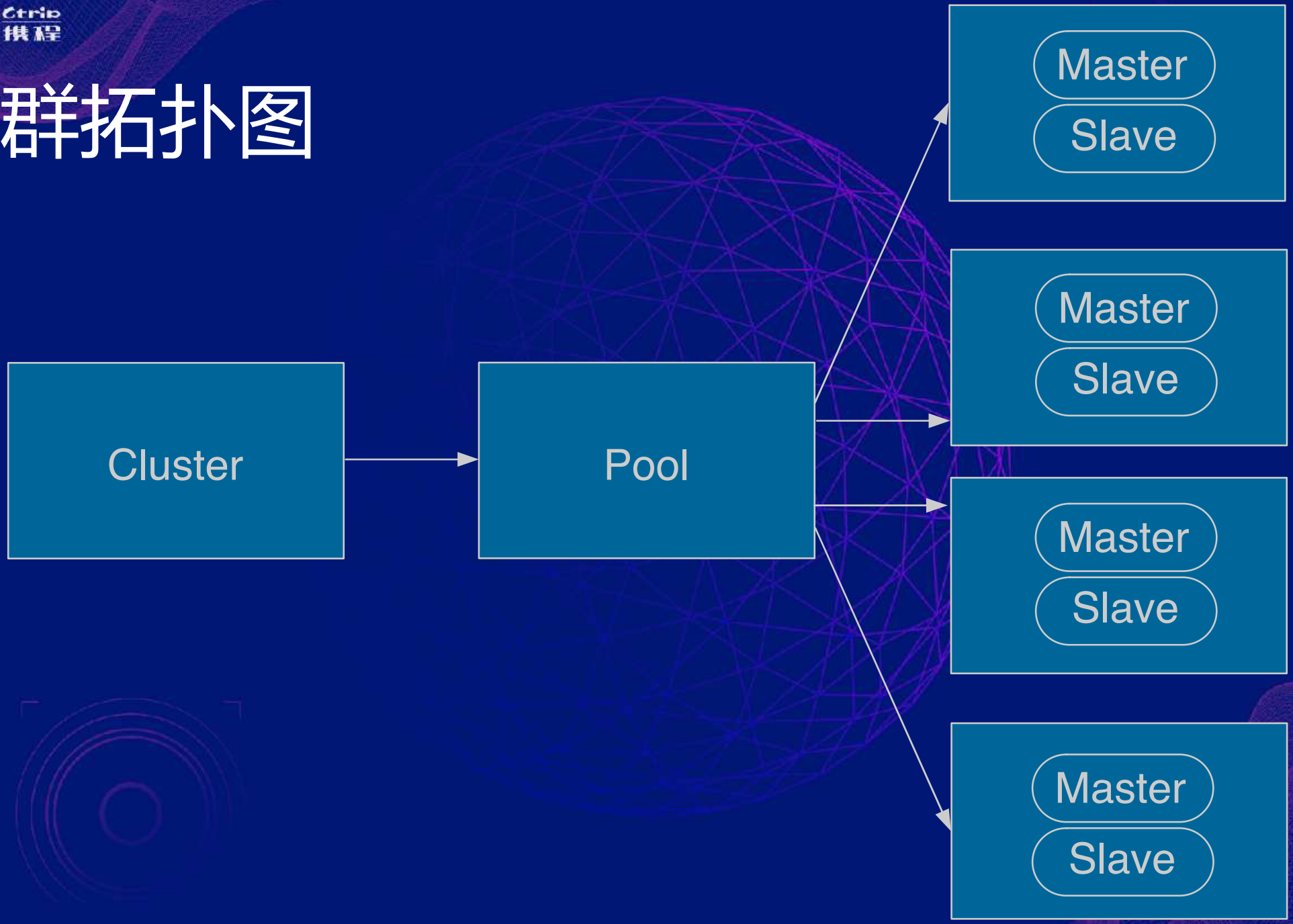
- 背景
- Redis为什么要容器化
- Redis能不能容器化
- 架构和细节
- 一些坑

背景

基本概念

- 集群是访问Redis的基本单位
- 应用通过CRedis提供的客户端通过集群访问到实例
- 多个集群对应一个Pool，一个Pool对应多个Group，每个Group对应一个或多个实例
- Key通过一致性hash散列到每个Group上

集群拓扑图


Cluster
Pool
Master
Slave
Master
Slave
Master
Slave
Master
Slave

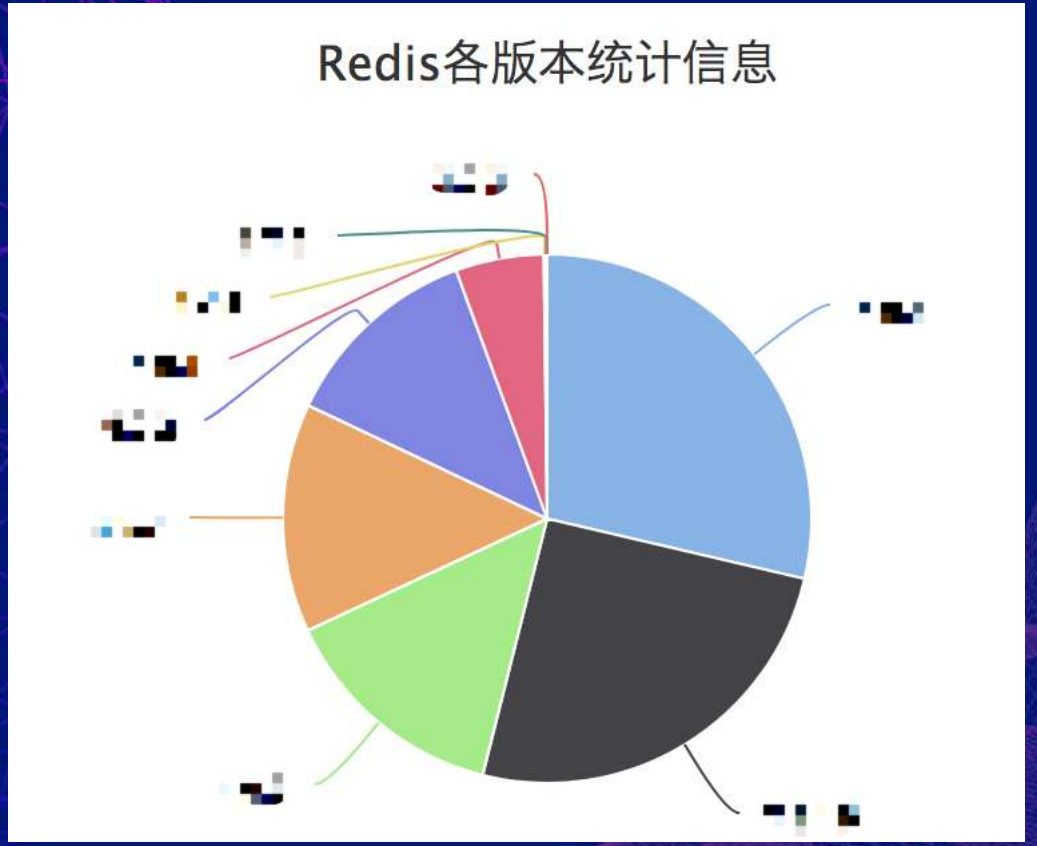
数据

- 总内存200T+，平均每天访问次数超百万亿次
- 宿主机1400+，实例数16000+
- 实例大的高达60G+，小的只有几百M。QPS高的有几万，低的个位数

Redis为什么要容器化

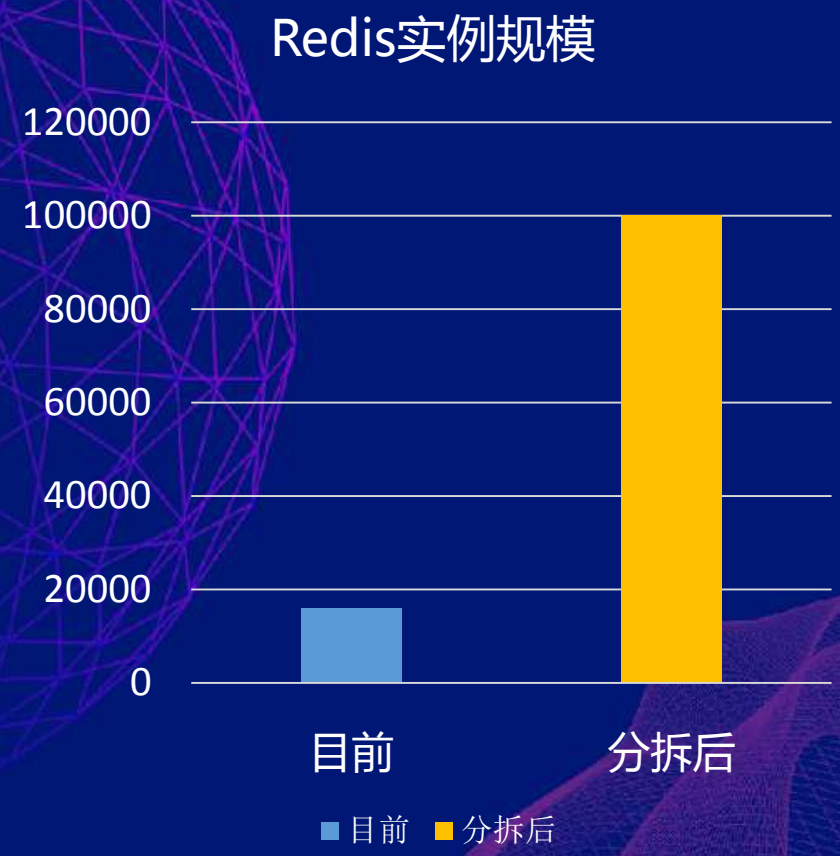
标准化和自动化

- Redis直接部署在物理机上，每个版本之间配置差异巨大，不好维护
- 容器镜像天然支持标准化，与物理机完全无关
- 容器基于K8S自动部署效率，相比人工部署提高了59倍



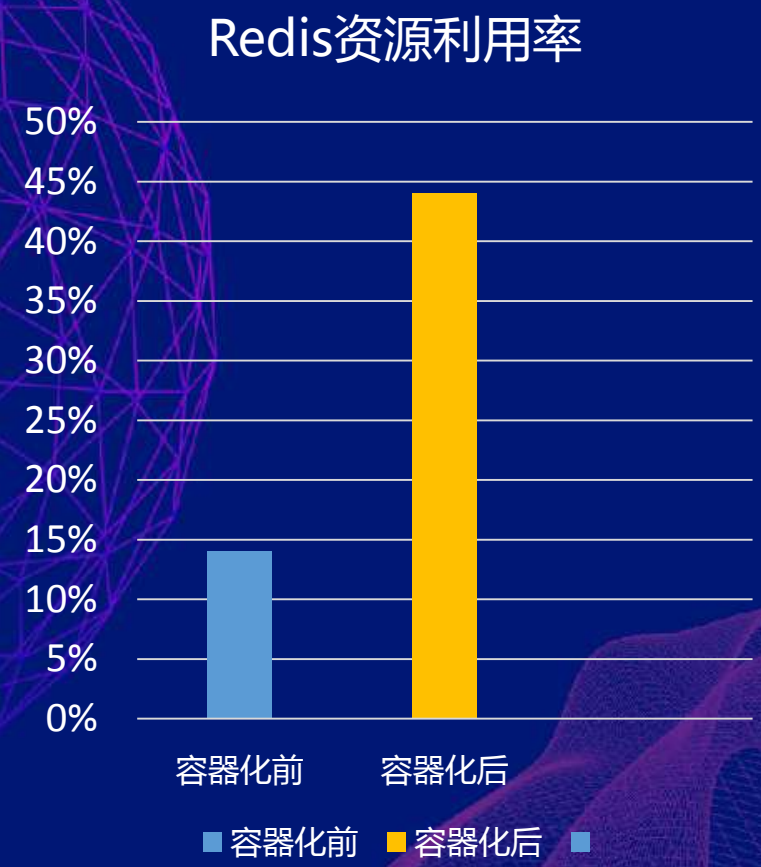
规模化

- 有别于社区方案，携程技术方案演进需要对大的实例分拆
- 实例分拆造成实例数急剧膨胀，靠人力难以运维
- 容器化能对分拆后的实例很好的管理和运维



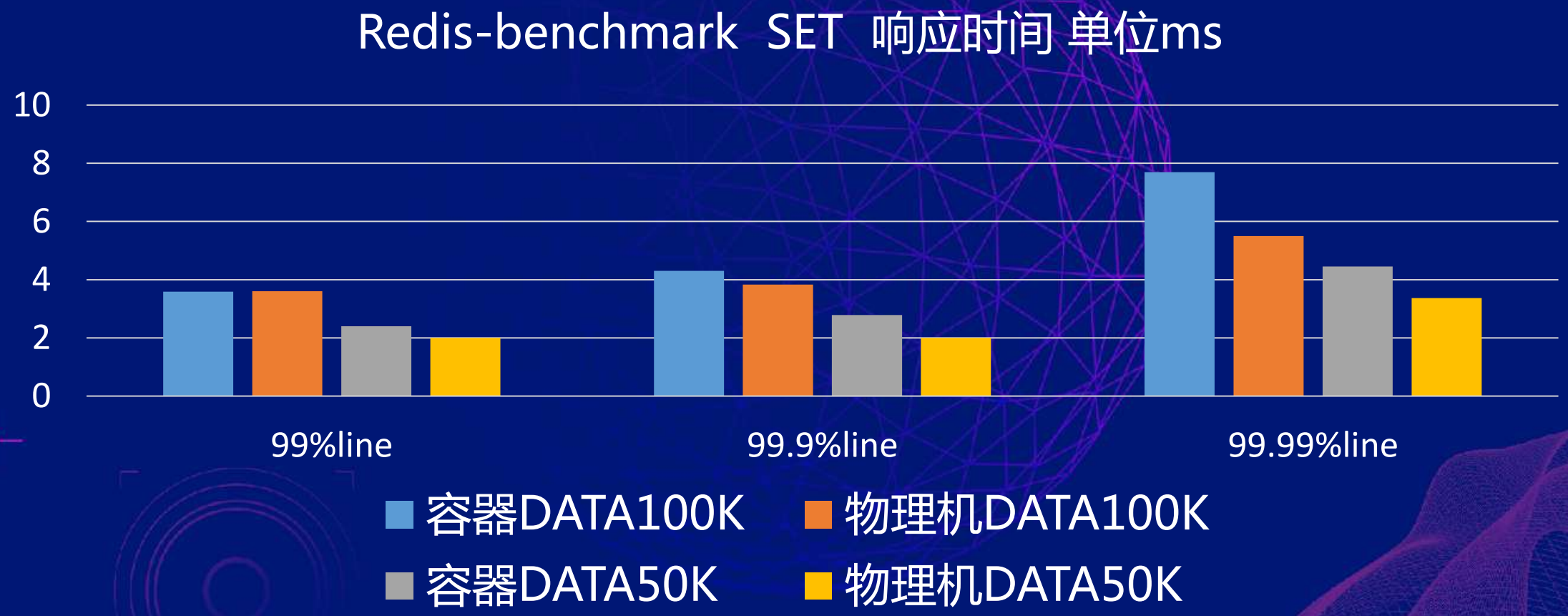
提高资源利用率

- 借助于容器化和上层的编排系统，我们很轻易的就可以做到资源利用率的提升

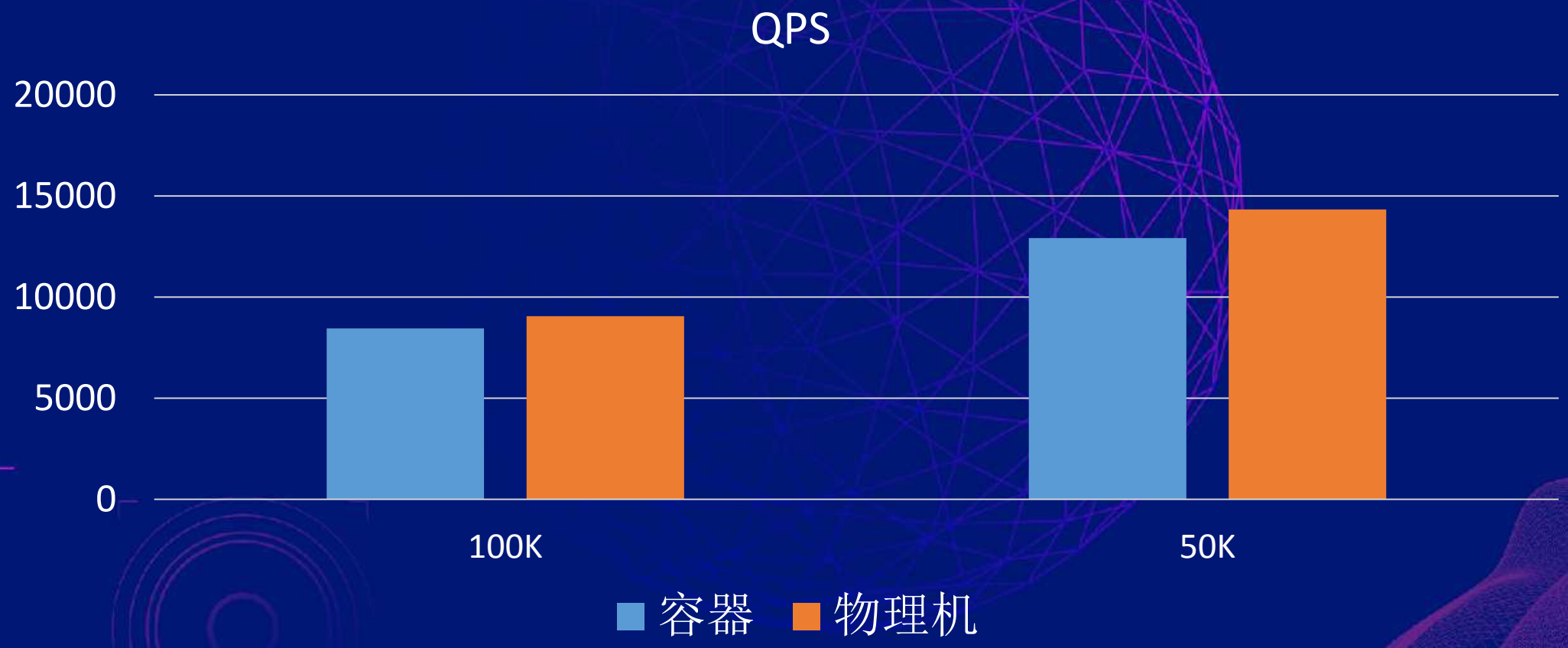


Redis能不能容器化

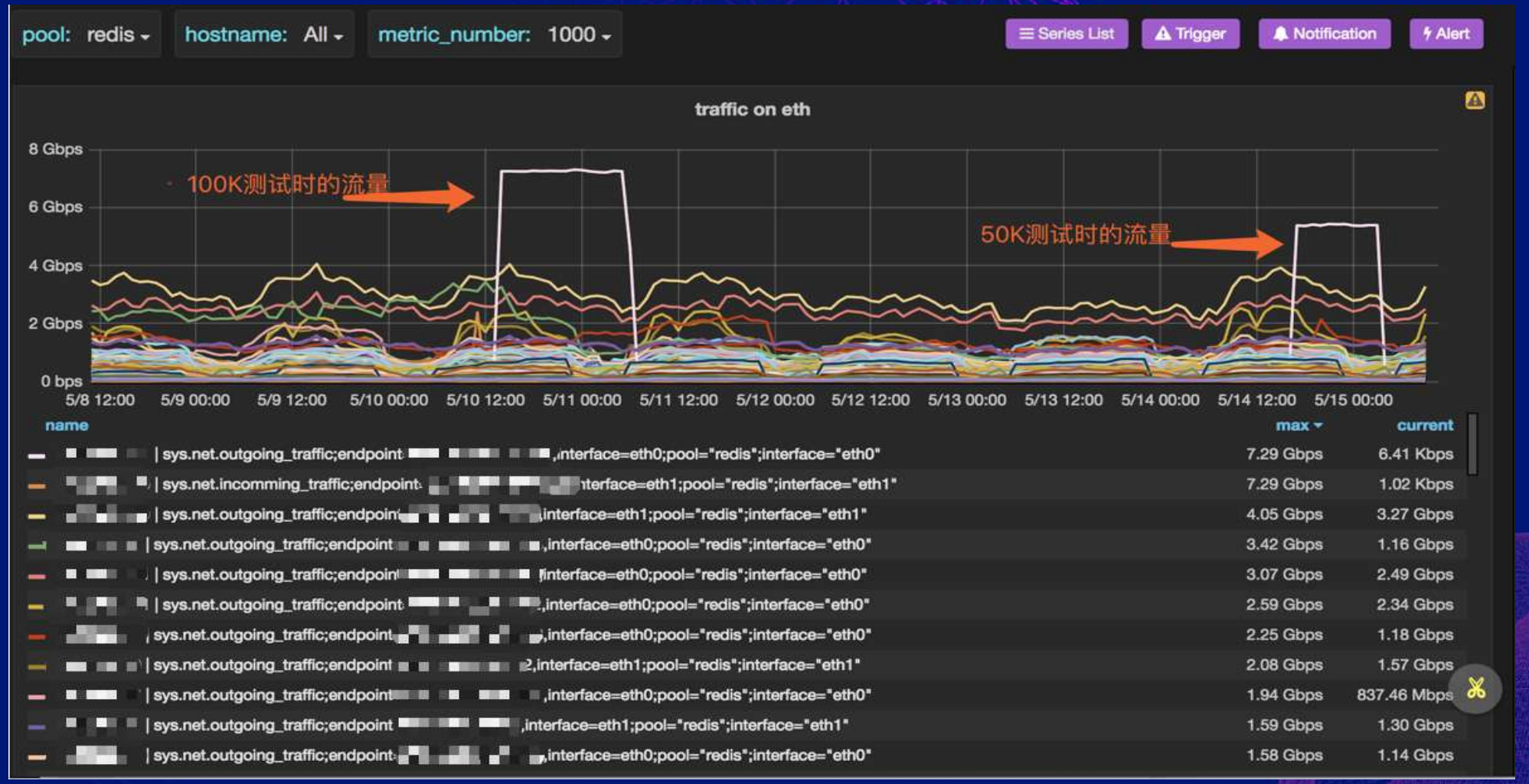
性能测试-响应时间


Redis-benchmark SET 响应时间 单位ms
10
8
6
4
2
0
99%line
99.9%line
99.99%line
容器DATA100K
物理机DATA100K
容器DATA50K
物理机DATA50K

性能测试-QPS



性能测试-监控流量


pool: redis
hostname: All
metric_number: 1000
Series List
Trigger
Notification
Alert
traffic on eth
8 Gbps
6 Gbps
4 Gbps
2 Gbps
0 bps
100K测试时的流量
50K测试时的流量
5/8 12:00
5/9 00:00
5/9 12:00
5/10 00:00
5/10 12:00
5/11 00:00
5/11 12:00
5/12 00:00
5/12 12:00
5/13 00:00
5/13 12:00
5/14 00:00
5/14 12:00
5/15 00:00
name
max
current
| sys.net.outgoing_traffic;endpoint ,interface=eth0;pool="redis";interface="eth0" 7.29 Gbps 6.41 Kbps
| sys.net.incomming_traffic;endpoint terface=eth1;pool="redis";interface="eth1" 7.29 Gbps 1.02 Kbps
| sys.net.outgoing_traffic;endpoin interface=eth1;pool="redis";interface="eth1" 4.05 Gbps 3.27 Gbps
| sys.net.outgoing_traffic;endpoint ,interface=eth0;pool="redis";interface="eth0" 3.42 Gbps 1.16 Gbps
| sys.net.outgoing_traffic;endpoin interface=eth0;pool="redis";interface="eth0" 3.07 Gbps 2.49 Gbps
| sys.net.outgoing_traffic;endpoint ,interface=eth0;pool="redis";interface="eth0" 2.59 Gbps 2.34 Gbps
sys.net.outgoing_traffic;endpoint ,interface=eth0;pool="redis";interface="eth0" 2.25 Gbps 1.18 Gbps
| sys.net.outgoing_traffic;endpoint 2,interface=eth1;pool="redis";interface="eth1" 2.08 Gbps 1.57 Gbps
| sys.net.outgoing_traffic;endpoint ,interface=eth0;pool="redis";interface="eth0" 1.94 Gbps 837.46 Mbps
| sys.net.outgoing_traffic;endpoint ,interface=eth1;pool="redis";interface="eth1" 1.59 Gbps 1.30 Gbps
sys.net.outgoing_traffic;endpoint ,interface=eth0;pool="redis";interface="eth0" 1.58 Gbps 1.14 Gbps

小结

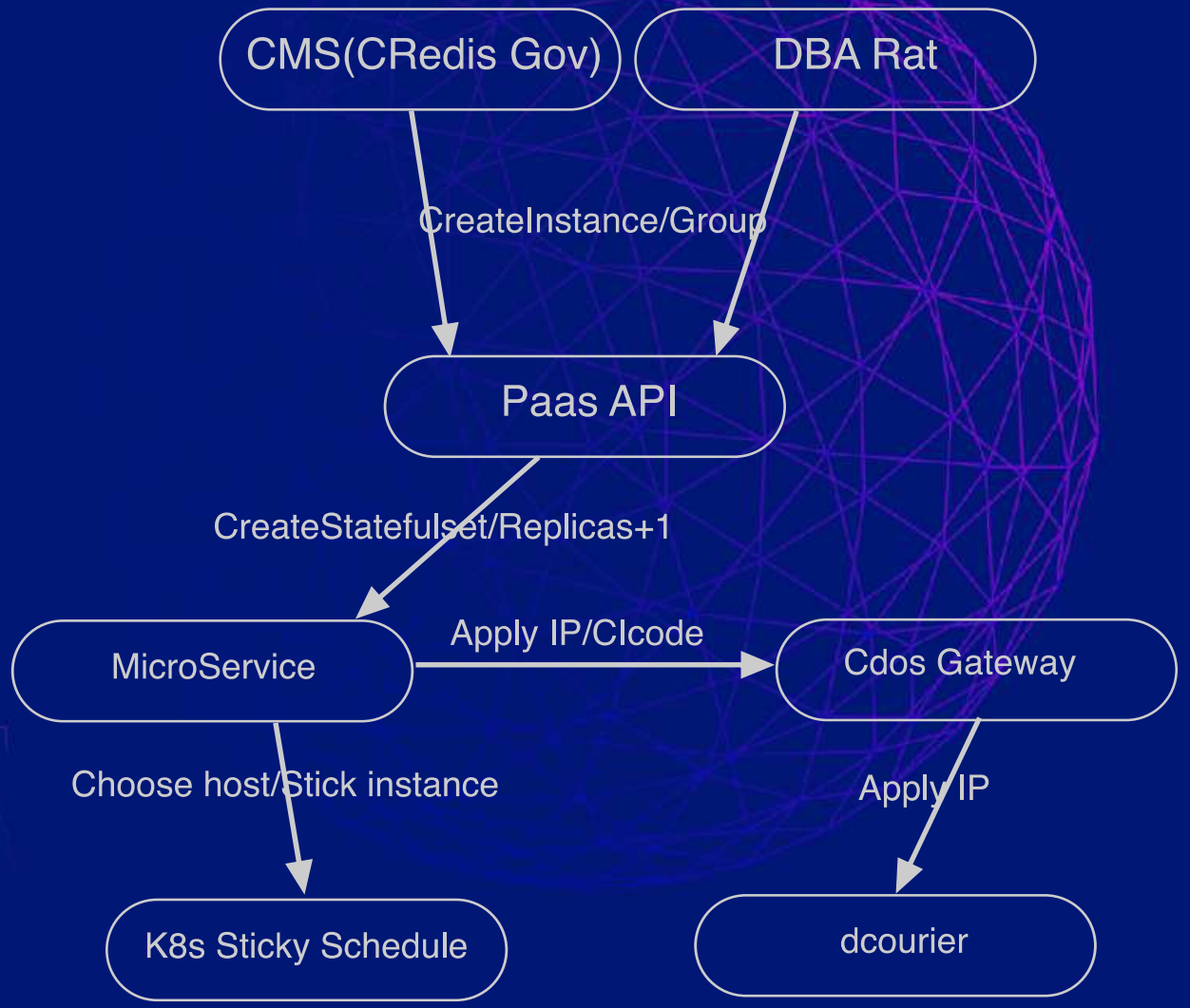
- 容器与物理机性能有细微差别,大约5-10%
- 携程的使用场景Redis完全可以容器化

架构

总体架构

- 运维和治理工具:CRedis Gov, Rat
- Paas提供统一接口对外接口
- Redis微服务提供Redis实例的CRD功能以及自定义的Redis调度策略
- 基础设施包括OVS/Neutron/Telegraf/Quota/StickyScheduler提供网络存储监控磁盘配额等方面能力

流程


CMS(CRedis Gov)
DBA Rat
CreateInstance/Group
Paas API
CreateStatefulset/Replicas+1
MicroService
Apply IP/CIcode
Cdos Gateway
Choose host/Stick instance
Apply IP
K8s Sticky Schedule
dcourier

细节

问题

- 客户端直连，IP固定和宿主机固定，Master/Slave不能在一台宿主机上
- 部署基于物理机，端口来区分，监控也通过端口来区分
- 重启实例Redis.conf文件配置不丢失
- Master挂了不希望被K8S立刻拉起来，而得让Sentinel感知
- 几乎没有任何内存控制

方案-K8S原生策略

- 基于K8S的Statefulset
- nodeAffinity保证调度到指定宿主机上
- podAntiAffinity保证同一个Statefulset的pod不调度到同一台宿主机上
- Toleations保证可以调度到taint的宿主机上，而该宿主机不会被其他资源类型调度到，如mysql,app等

方案-Sticky Scheduler

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  annotations:
    redis-instance0: host0
    redis-instance1: host1
  creationTimestamp: 2018-06-14T11:38:17Z
  generation: 1
  labels:
    app: redis-instance
  name: redis-instance
  namespace: test-redis
  resourceVersion: "78632983"
  selfLink: /apis/apps/v1/namespaces/test-redis/statefulsets/redis-instance
  uid: 6e678f82-6fc7-11e8-aca4-567909392929
```

方案-quotas

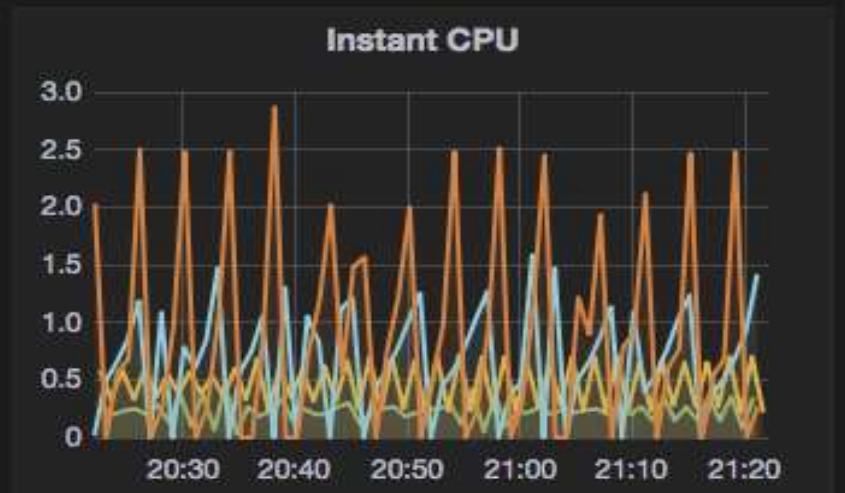
```
volumes:
- flexVolume:
    driver: cloud.ctrip.com/chostpath
    options:
      quota-hard: 4096Mi
      quota-soft: 4096Mi
  name: data
- flexVolume:
    driver: cloud.ctrip.com/cemptydir
    options:
      quota-hard: 1024Mi
      quota-soft: 1024Mi
  name: log
```

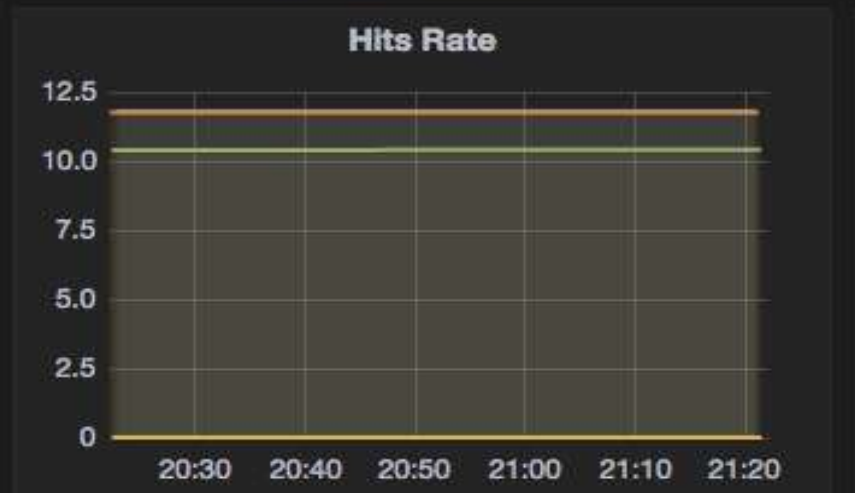
- 自研chostpath和cemptydir基于xfs的quota限额
- 将Redis.conf和data目录挂载出，保证重启容器后配置不丢失，也保证容器重启后可以读rdb数据

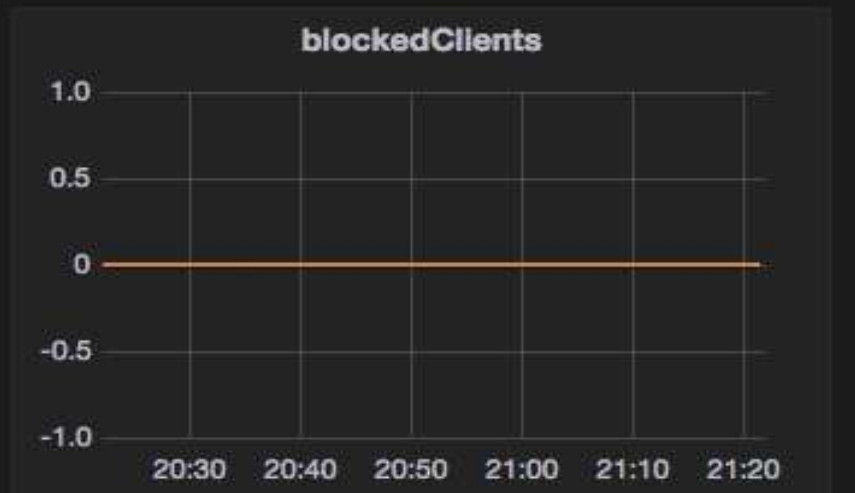
方案-监控

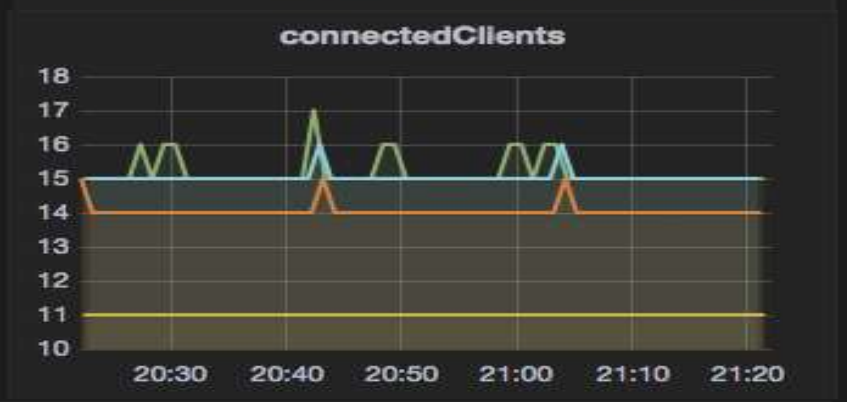
- 每个POD两个容器,一个Redis实例,一个监控程序telegraf
- 监控程序基于物理机监控脚本移植
- 所有telegraf脚本固化在物理机上，一旦修改方便统一推送，对Redis实例无任何影响

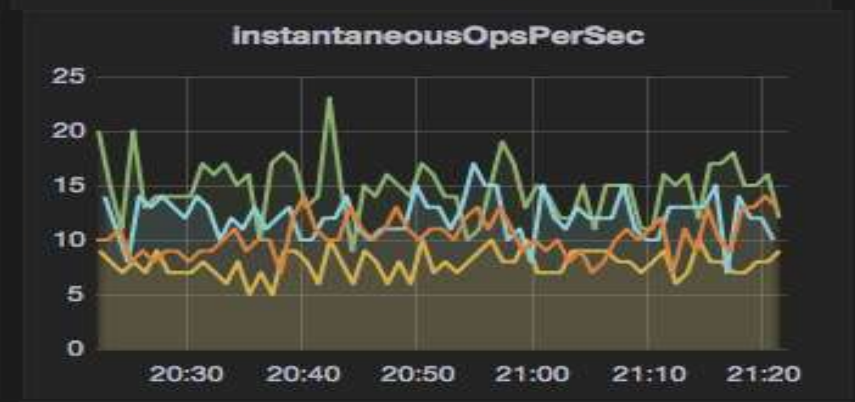
物理机REDIS性能指标

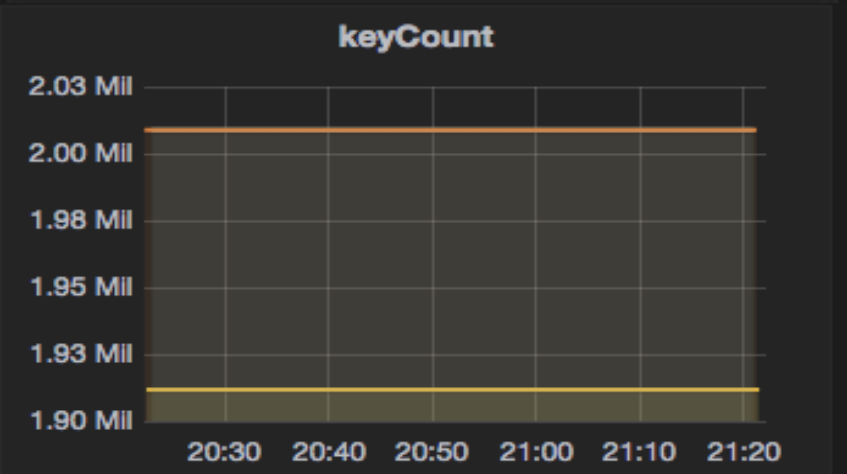


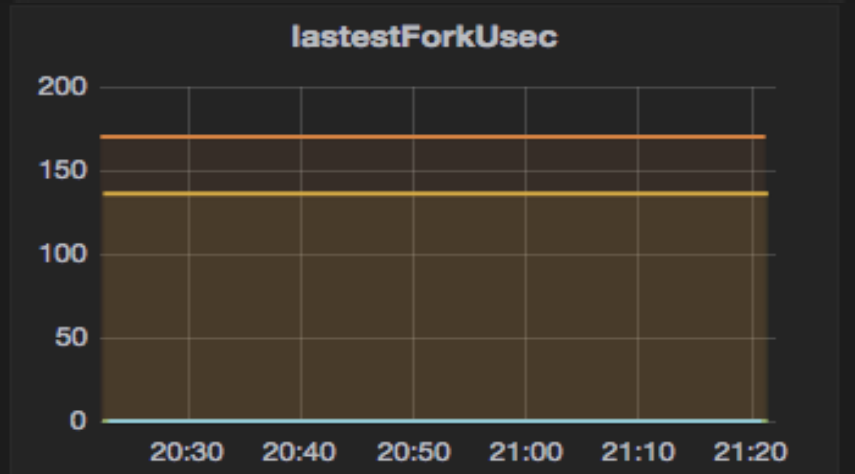


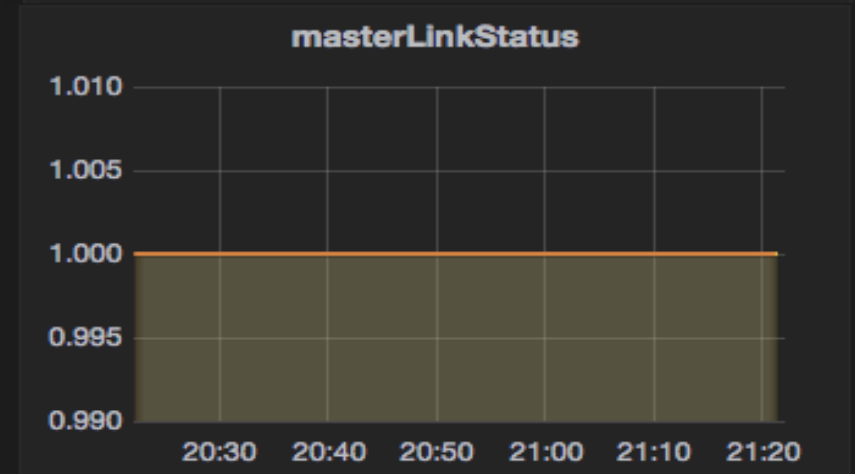








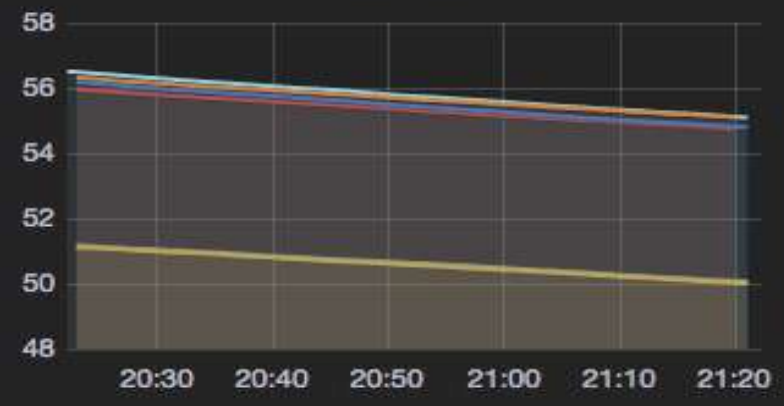




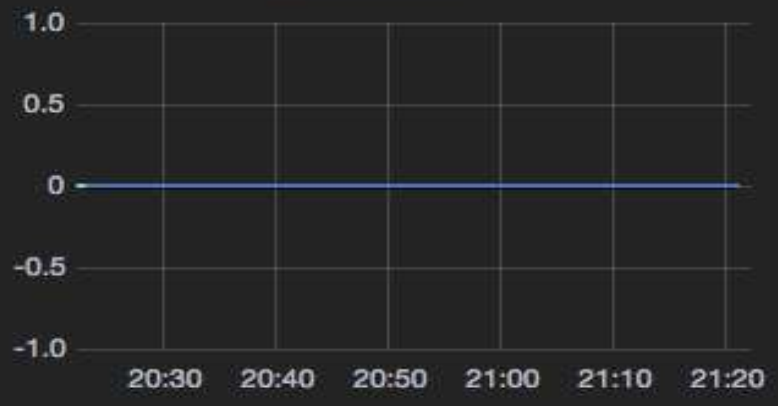
K8S 容器REDIS性能指标

CPU Util %

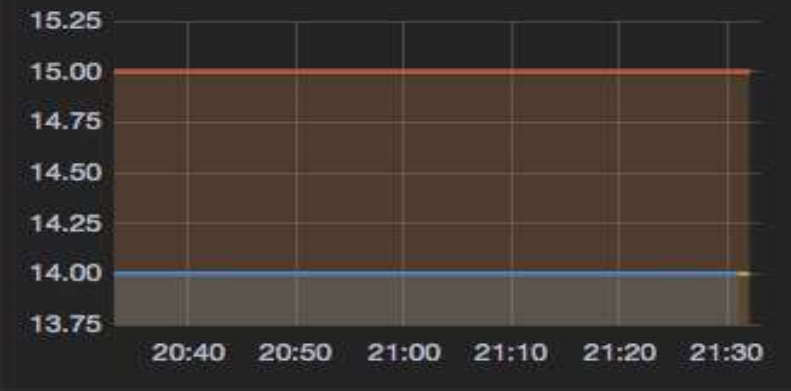
hitsRate



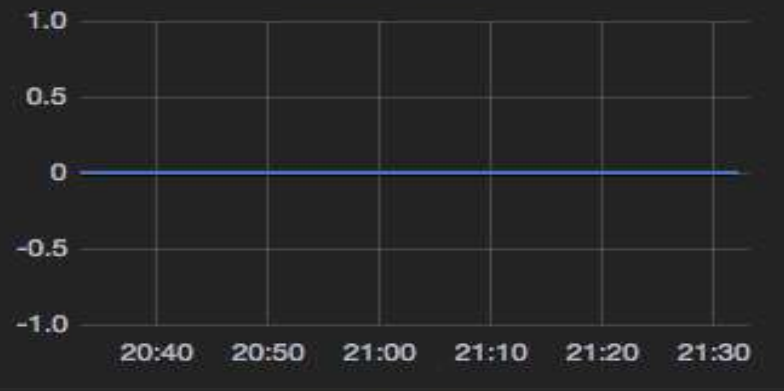
blockedClients



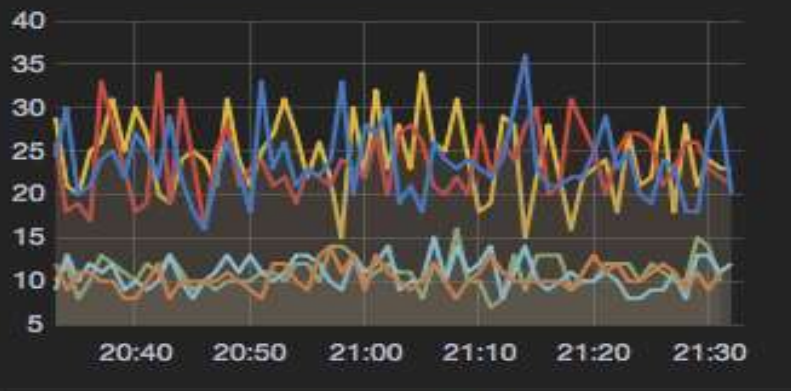
connectedClients



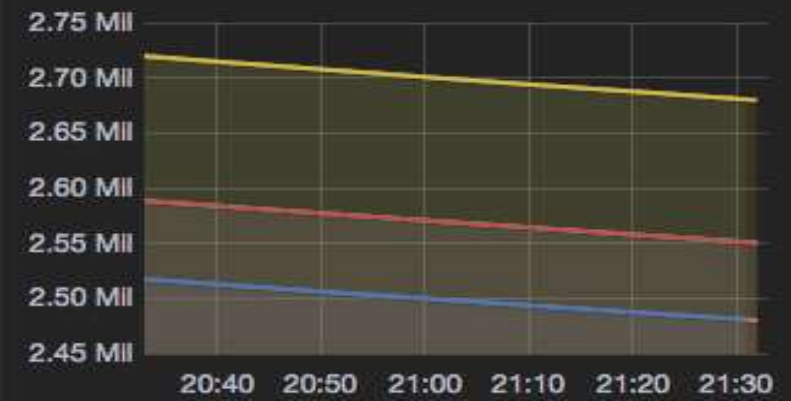
evicted_keys



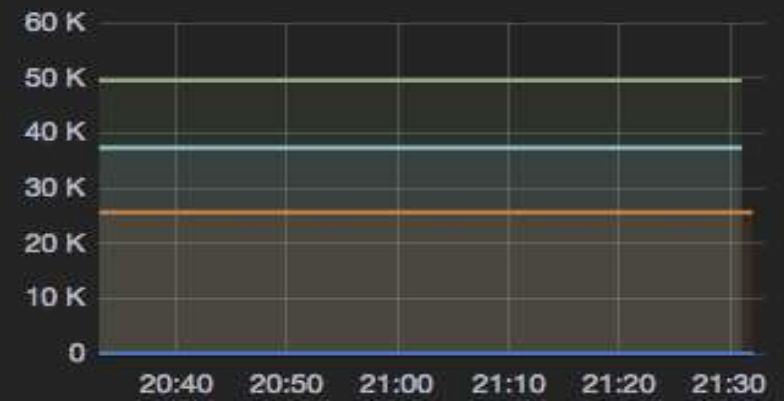
InstantaneousOpsPerSec



keyCount



lastestForkUsec



masterLinkStatus

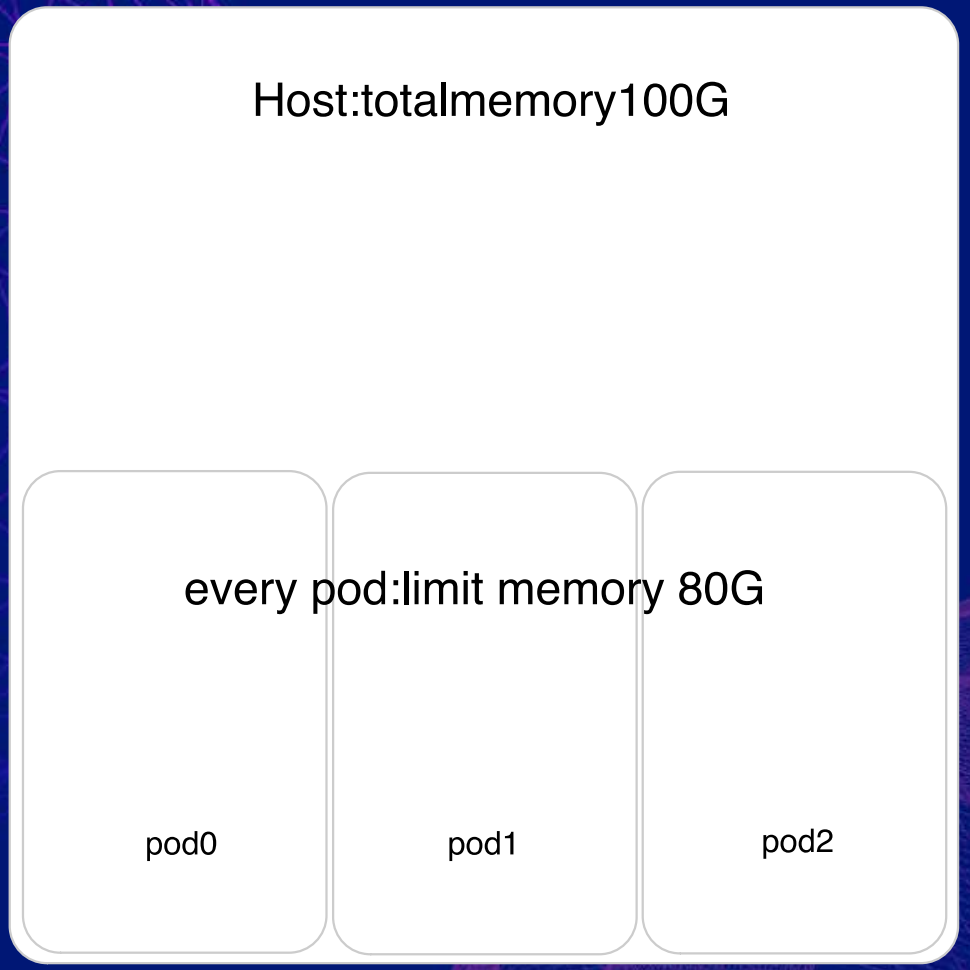
方案-supervisord

- 1号进程supervisord
- Redis挂掉后，supervisord不去主动拉起
- 等哨兵执行master/slave切换后再删pod重建

```
PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
  1  0.0  0.0 123588 18592 ?        Ss   Oct16   4:32 /usr/bin/python /usr/bin/supervisord
 18  0.0  0.0 105884  7508 ?        S    Oct16   0:00 /usr/sbin/sshd -D
 19  0.0  0.0  45316  5296 ?        Sl   Oct16  20:41 /opt/app/redis/sbin/redis-server *:6
 23  0.5  0.0  15448  3536 ?        Ss   00:30   0:00 bash
 39  0.0  0.0  51096  3804 ?        R+   00:30   0:00 ps aux
```

方案-内存CPU配额

- CPU超分不限制
- 容器内存超分到80G,limit=80G

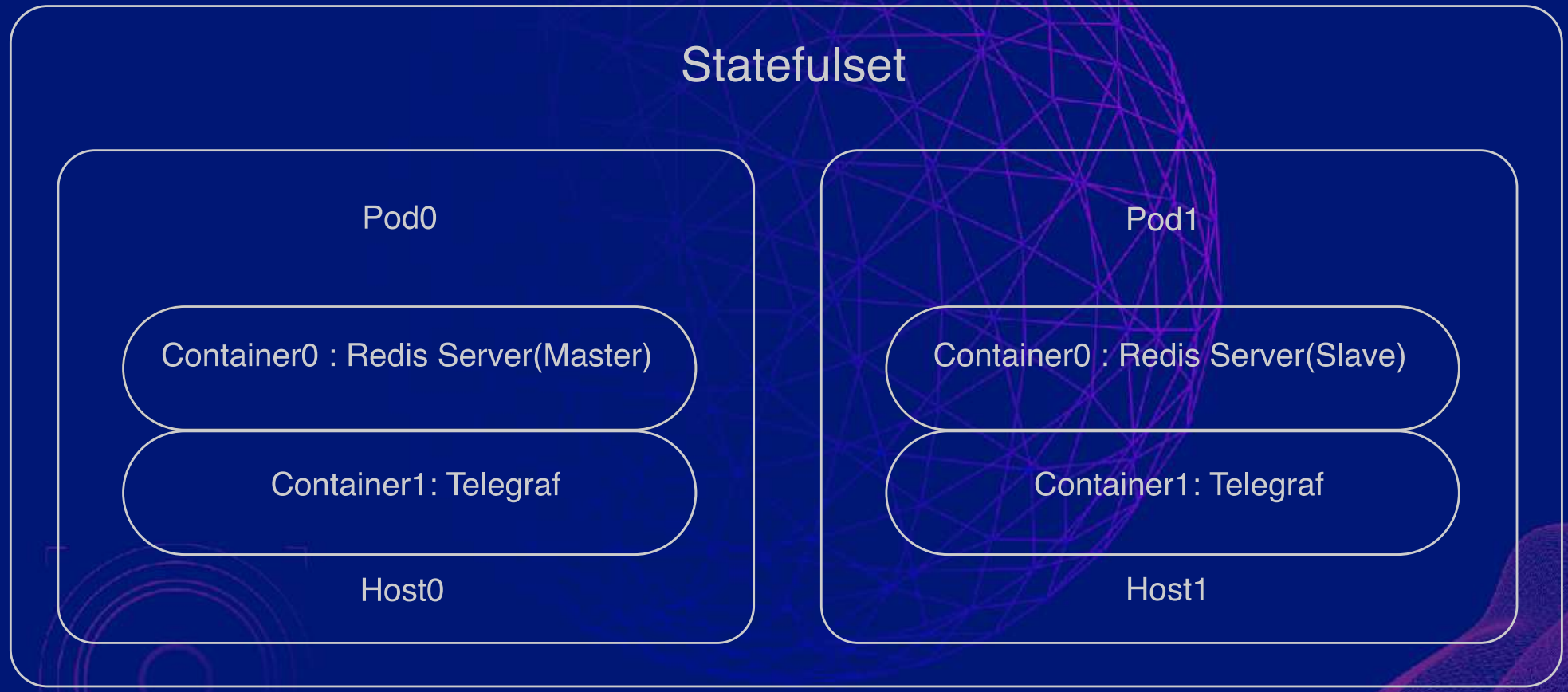


超分大法好,OOM了怎么办？

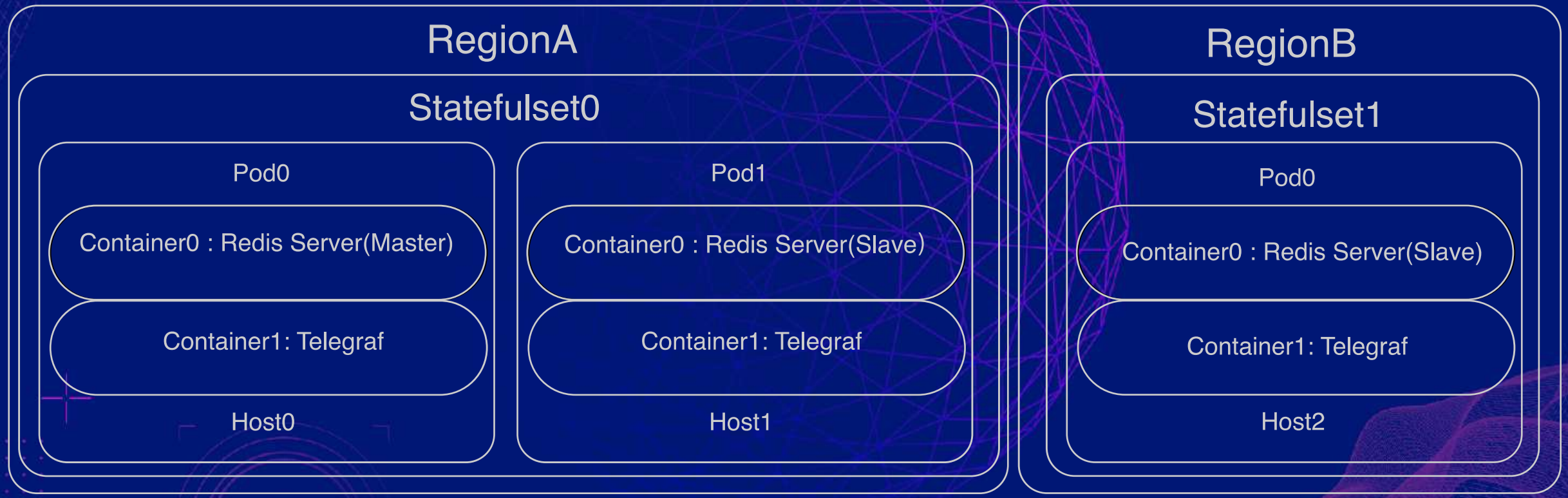
方案-调度策略

- 集群重要性划分
- 基于重要性调度到低中高配机器上
- 基于重要性决定是否调度到多Region上

单Region

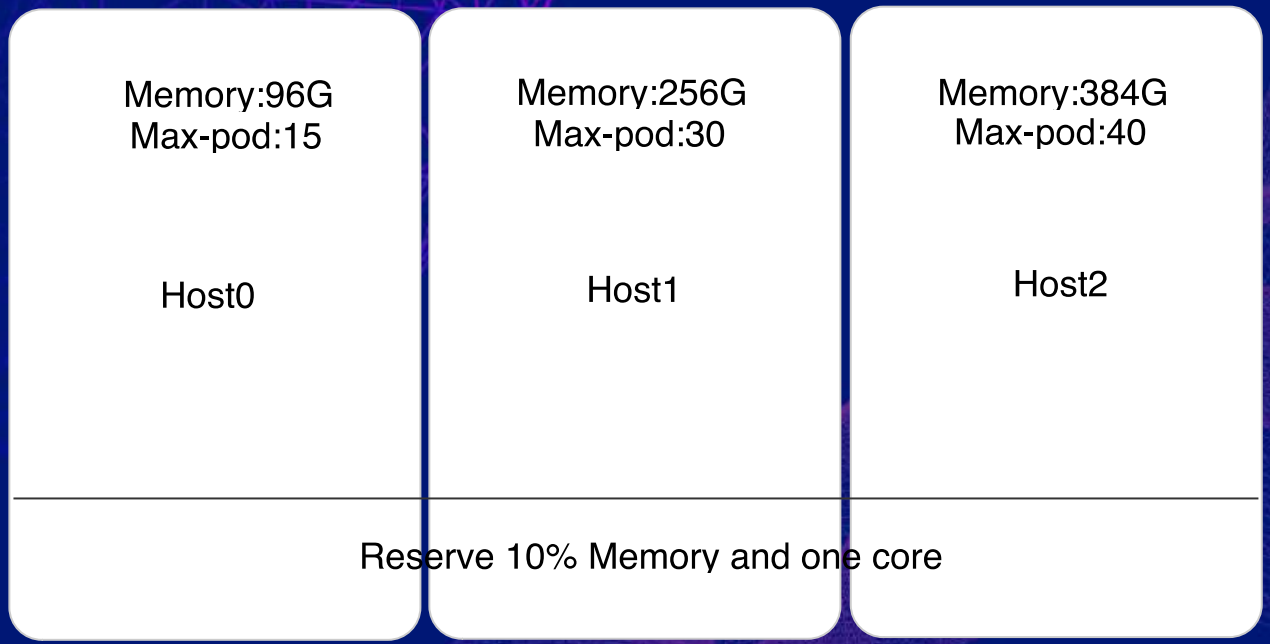

Statefulset
Pod0
Container0 : Redis Server(Master)
Container1: Telegraf
Host0
Pod1
Container0 : Redis Server(Slave)
Container1: Telegraf
Host1

多Region


RegionA
Statefulset0
Pod0
Container0 : Redis Server(Master)
Container1: Telegraf
Host0
Pod1
Container0 : Redis Server(Slave)
Container1: Telegraf
Host1
RegionB
Statefulset1
Pod0
Container0 : Redis Server(Slave)
Container1: Telegraf
Host2

方案-宿主机预留(调度前)

- 不同配置宿主机限定不同max-pod数量
- 占位pod预留10%资源禁止调度
- 设定pod的request=redis max_memory



方案-调度中和调度后

- 基于宿主机实际容量调度
- 定时轮询所有宿主机查看内存分配是否合理，对于不合理的集群，自动触发迁移

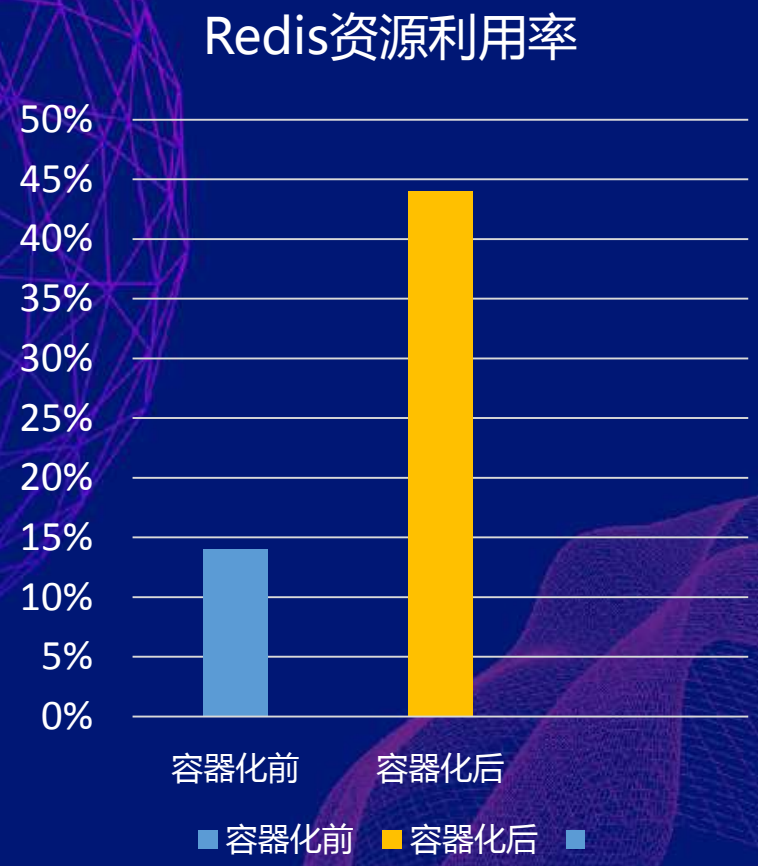
方案-其他策略

- 动态调整HZ（减小used_memory)
- 动态打开关闭自动碎片整理(减小rss_memory)
- 宿主机内存告警 (>80%)

小结

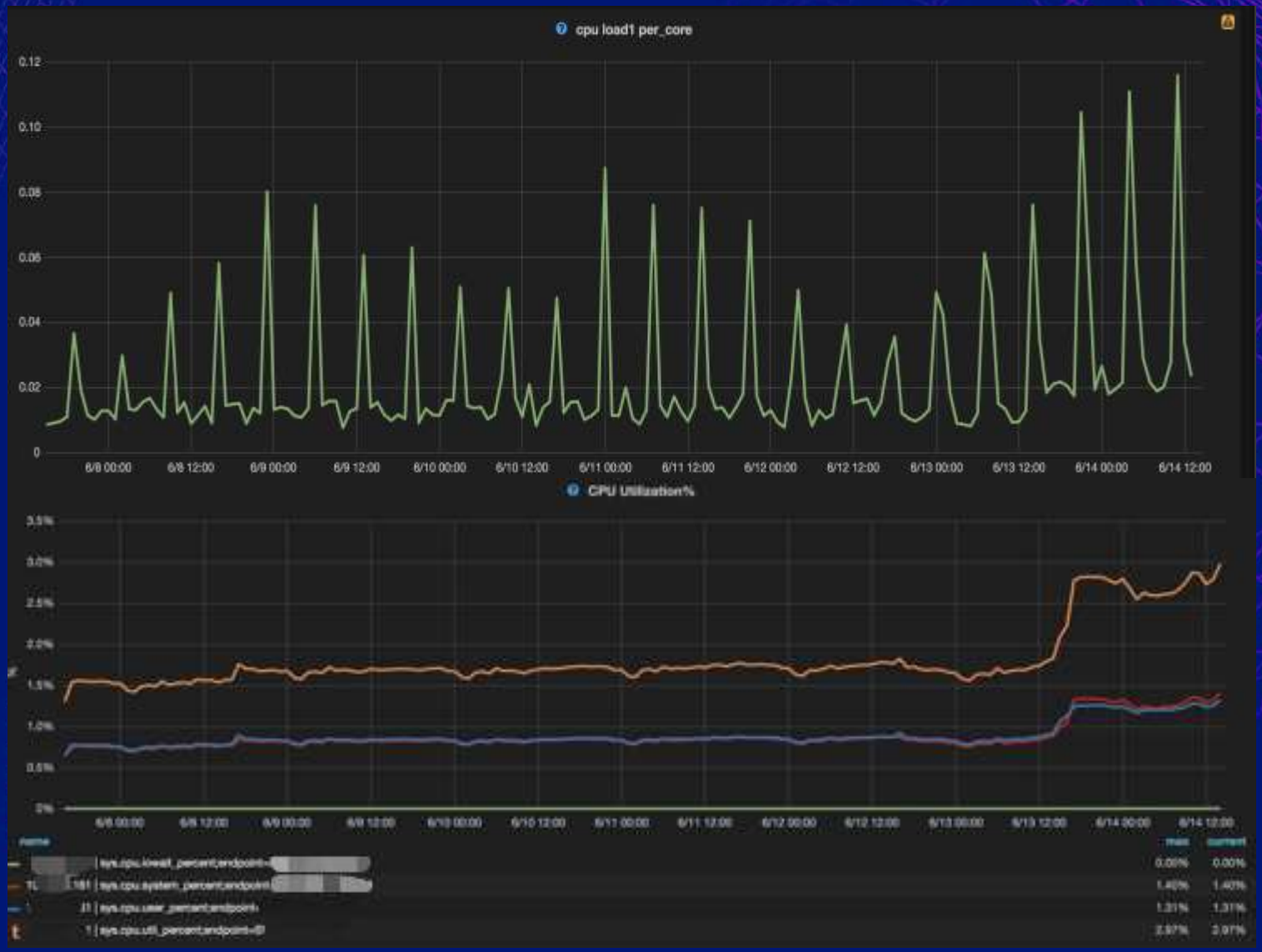
- Redis跑在容器上，多个组件共同协作的才能达成
- 现状决定必须超分
- 超分后如何不OOM是关键
- 思路需要发散，容器层面和Redis层面都

可以相应的对策



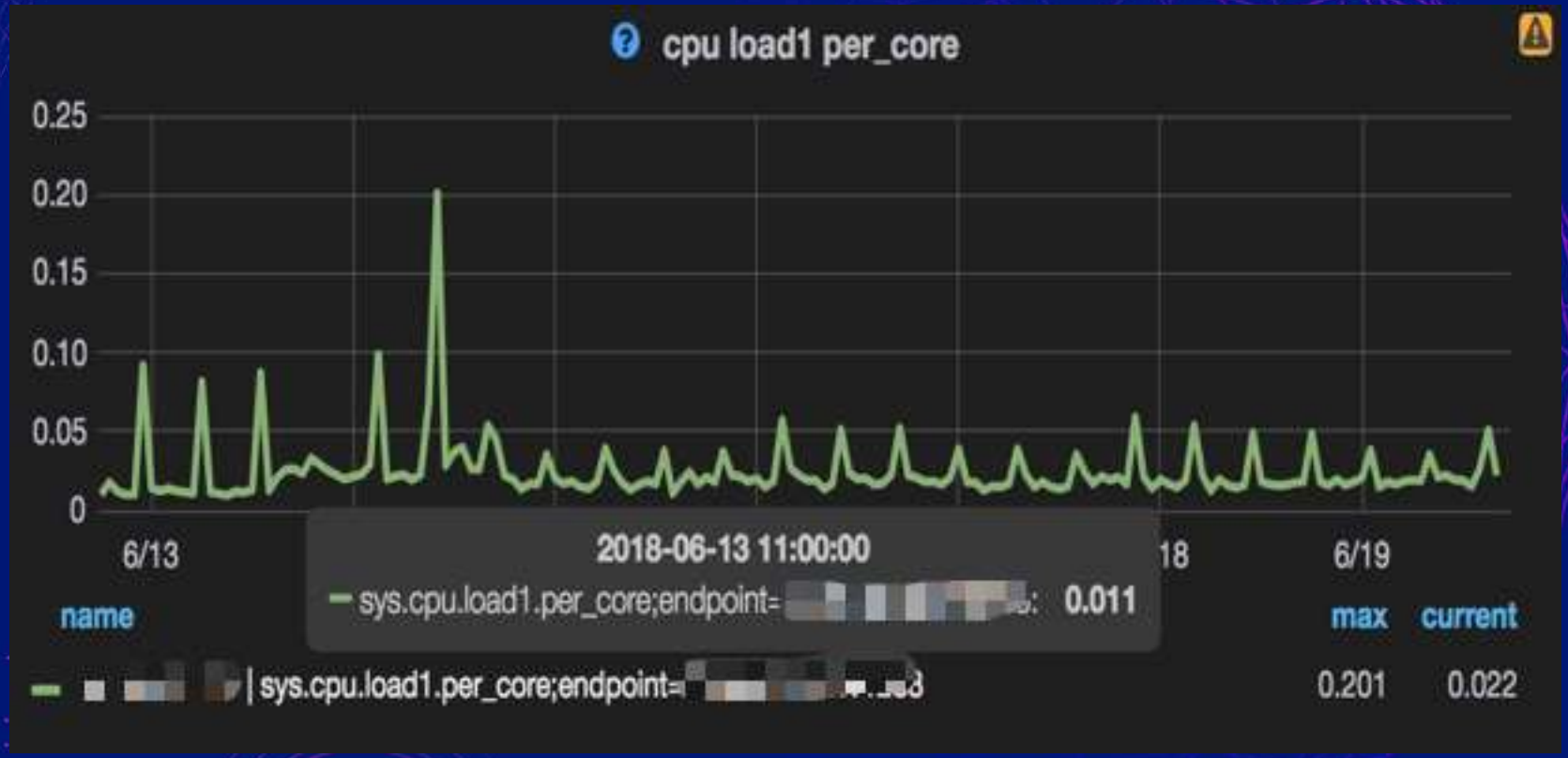
一些坑

System load有规律毛刺



增加POD毛刺上升，而且看上去跟CPU利用率无关

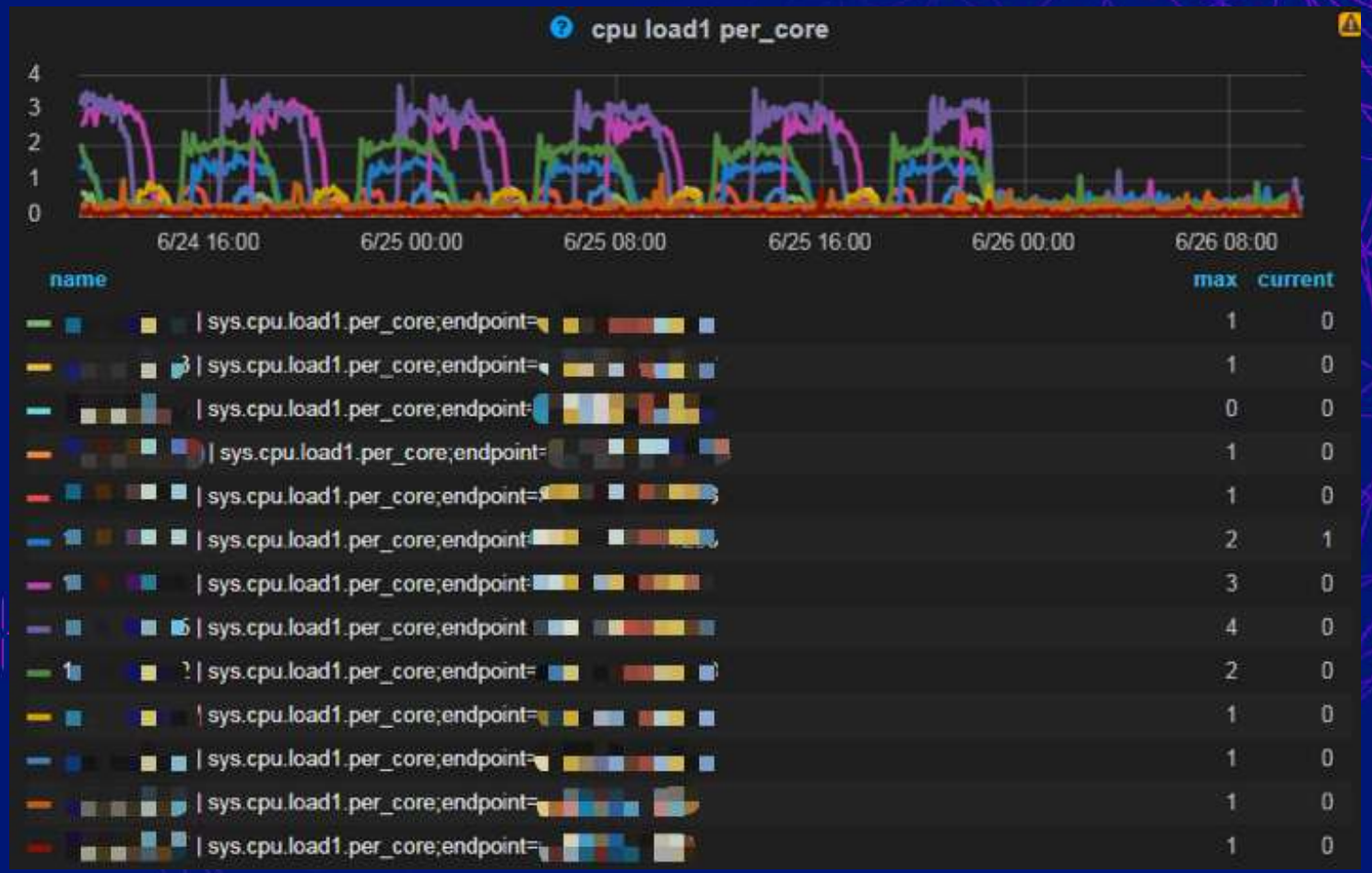
System load有规律毛刺



降低POD数量，毛刺减小，

但并没有消失

System load有规律毛刺



- 修改telegraf中的collection_jitter值，用来设置一个随机的抖动来控制tellgraf采集前的休眠时间，让瞬间不会爆发上百个进程

Slowlog异常


redis慢日志查询
集群名称
开始时间
2018-08-06 00:00:00
结束时间
2018-08-06 23:59:59
查询
Redis慢日志详情
导出excel
设定慢日志阈值
过滤
集群名称
实例ip
实例端口
慢日志
执行时间(ms)
开始时间
1819
2018-08-06 22:42:32
1819
2018-08-06 22:26:49
1810
2018-08-06 21:53:08

Slowlog异常

```
    start = ustime();//call gettimeofday()
    c->cmd->proc(c);
    duration = ustime()-start;

//ustime function
/* Return the UNIX time in microseconds */
long long ustime(void) {
    struct timeval tv;
    long long ust;

    gettimeofday(&tv, NULL);
    ust = ((long long)tv.tv_sec)*1000000;
    ust += tv.tv_usec;
    return ust;
}
```

- 具体分析可以查看携程技术中心微信公众号：ctriptech

Xfs bugs

```
[root@[redacted] ~]# xfs_db -r /dev/mapper/VolDocker-dockerdata
Metadata corruption detected at xfs_agf block 0x7dc01001/0x200
xfs_db: cannot init perag data (-117). Continuing anyway.
xfs_db> q
[root@[redacted] ~]# xfs_db -r /dev/mapper/VolDocker-dockerbase
Metadata corruption detected at xfs_agf block 0x92b56801/0x200
xfs_db: cannot init perag data (-117). Continuing anyway.
xfs_db>

typedef struct xfs_agfl {
    __be32      agfl_magicnum;
    __be32      agfl_seqno;
    uuid_t      agfl_uuid;
    __be64      agfl_lsn;
    __be32      agfl_crc;
    __be32      agfl_bno[]; /* actually xfs_agfl_size(mp) */
} __attribute__((packed)) xfs_agfl_t;
```

- xfsprogs4.5(mkfs.xfs)使用的是没有**attribute**((packed))，64位上sizeof (xfs_agfl)是40字节，
- 内核态(linux4.5以后)的XFS有**attribute**((packed))，64位机器上sizeof 是36字节，会导致内核在写xfs_agfl时候误判有多一个agfl_bno,写出界导致Metadata corruption

Xfs bugs

- Xfsaild进入D状态导致宿主机大量D状态进程和僵尸进程,最终导致宿主机僵死
- 与khuagepaged有关
- 升级内核到4.14.67后backport4.15-4.19的bugfix，关闭透明大页，压测问题依然存在，但能控制free内存在3G以上不复现，实际生产上使用4.14内核还没再复现过。

本PPT来自2018携程技术峰会

更多技术干货，请关注“携程技术中心”微信公众号