云原生社区 Cloud Native Community | SOFASTACK™

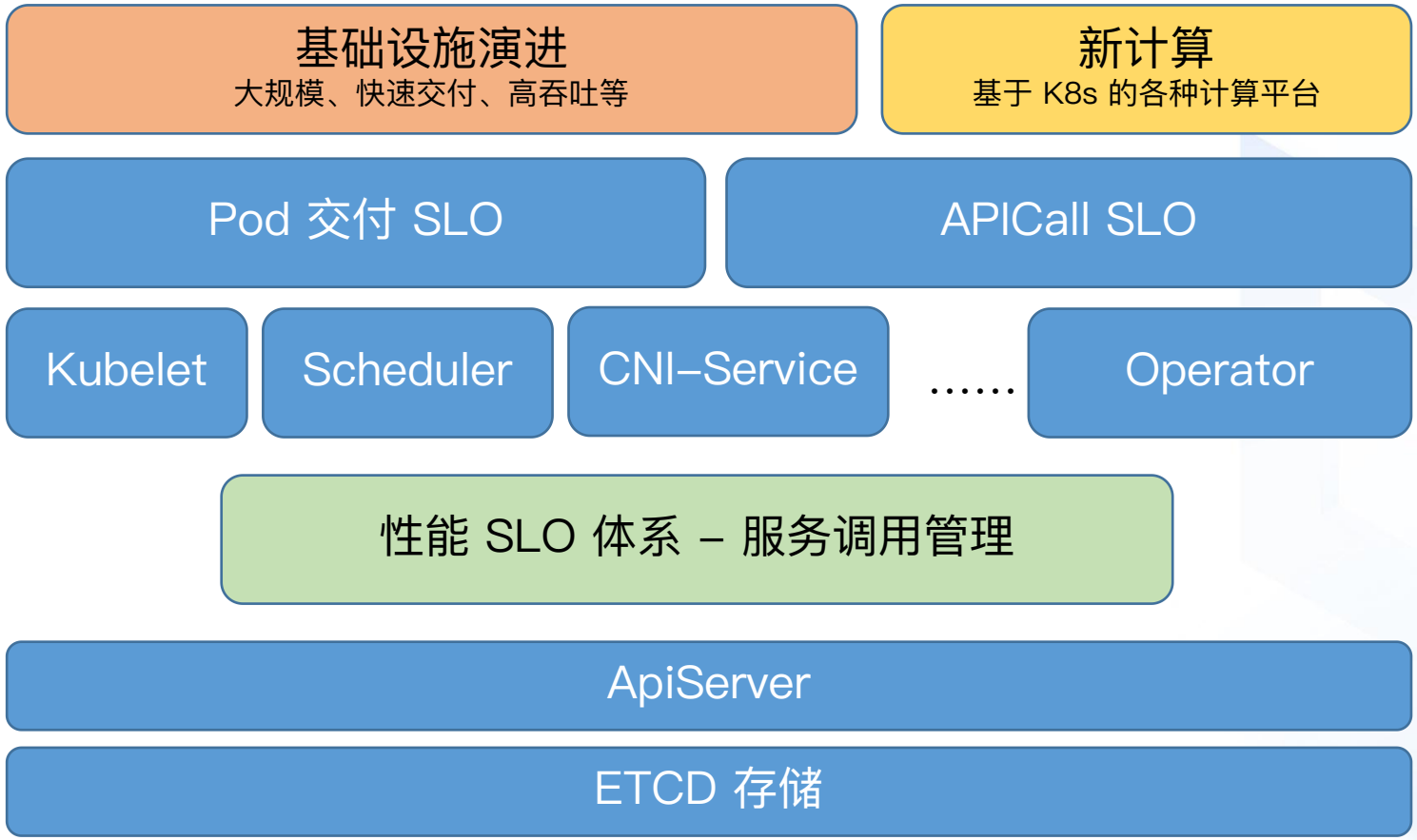
云原生开放日
Cloud Native Open Day
第八期 上海站

攀登规模化的高峰 —
蚂蚁集团大规模 Sigma 集群 ApiServer 优化实践

唐博（花名：博易）
谭崇康（花名：见云）

目录

ApiServer 性能是 K8s 生态体系的基础



K8s 的角色：
云原生资源供给：为业务提供计算资源（pod）
状态数据库：为业务提供状态信息存储

规模
节点数：集群可用资源；
资源数：可容纳业务及配置量。

管控链路 -> 计算链路
对基于 K8s 构建的 Spark、Blink 等新计算平台，APICall Latency 已经成为其计算链路服务 SLO 的关键部分。

规模化带来的性能挑战

社区版 K8s 的规模限制

Quantity	Threshold scope=namespace	Threshold: scope=cluster
#Nodes	n/a	5000
#Namespaces	n/a	10000
#Pods	3000	150000
#Services	5000	10000
#Endpoints per service	250	n/a
#Deployments	2000	TBD

规模化带来的全方位挑战：

资源层面：随着集群规模增大， 以 IP 为代表的核心资源明显不足，出现 IP 申请超时问题，造成 Pod 交付失败；

节点层面：随着节点规模增大，用于节点鉴权的 node-graph 结构构建出现性能问题，影响 Pod 的创建交付；

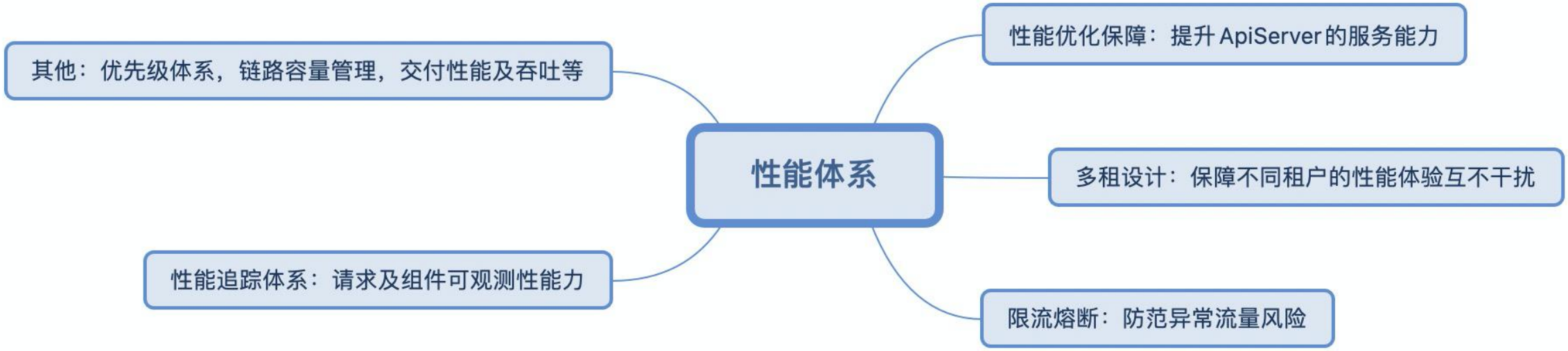
Pod 层面：随着 Pod 规模的增大，Watch Cache 的大小满足不了 Pod 访问的要求，出现了大量的 watch Error 及 relist 情况；

ETCD 层面：随着集群中资源对象数据量的增大，ETCD 存储的数据规模过大，引起 Latency 大幅上涨，同时因请求量过多引起的 Too Many Request 问题也此起彼伏。

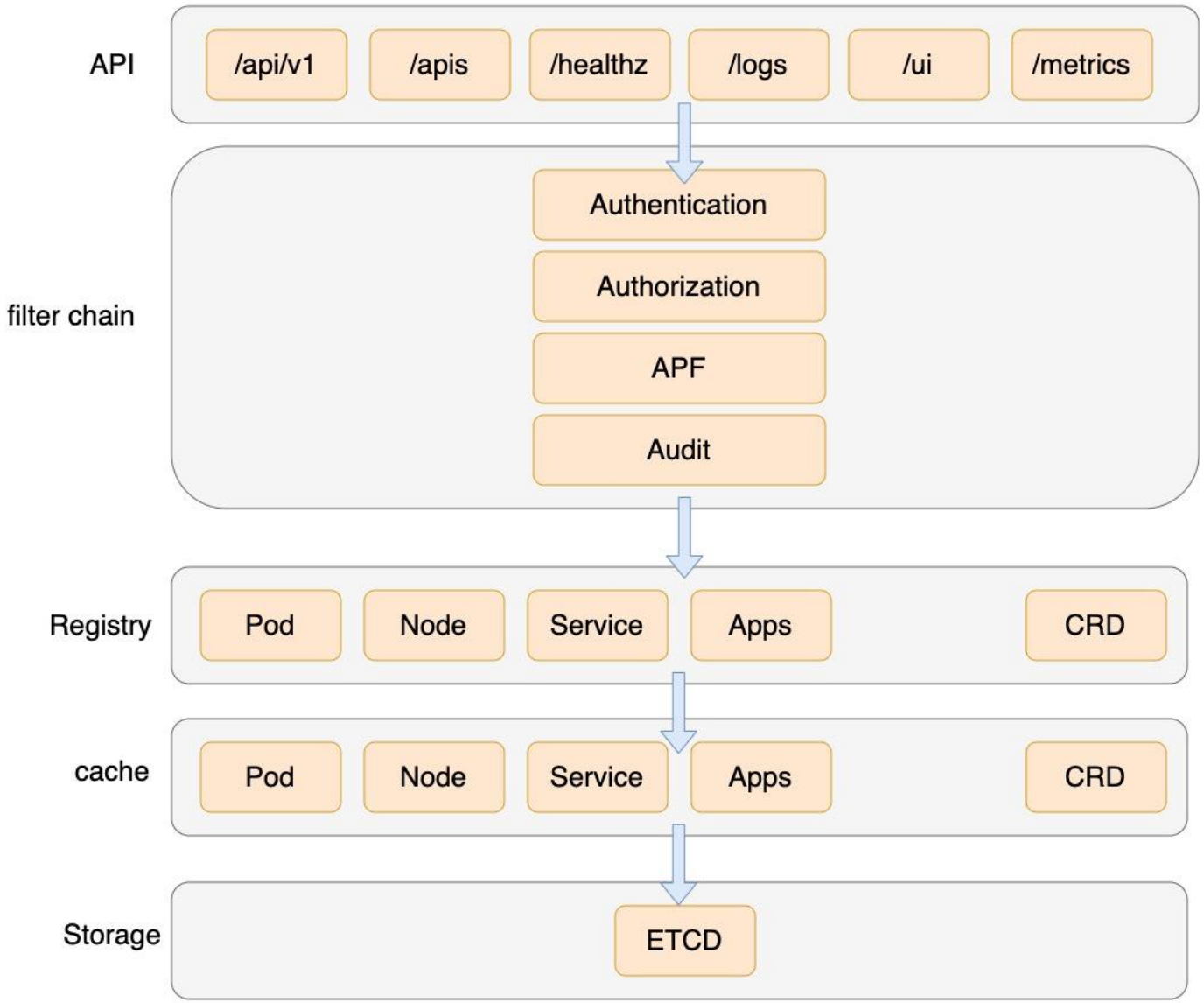
性能衡量标准

Status	SLI	SLO
Official	Latency of mutating API calls for single objects for every (resource, verb) pair, measured as 99th percentile over last 5 minutes	In default Kubernetes installation, for every (resource, verb) pair, excluding virtual and aggregated resources and Custom Resource Definitions, 99th percentile per cluster-day[1] <= 1s fraction of good minutes per day
Official	Latency of non-streaming read-only API calls for every (resource, scope pair, measured as 99th percentile over last 5 minutes	In default Kubernetes installation, for every (resource, scope) pair, excluding virtual and aggregated resources and Custom Resource Definitions, 99th percentile per cluster-day[1] (a) <= 1s if scope=resource (b) <= 5s if scope=namespace (c) <= 30s if scope=cluster

性能体系


性能体系
性能优化保障：提升 ApiServer 的服务能力
多租设计：保障不同租户的性能体验互不干扰
限流熔断：防范异常流量风险
其他：优先级体系，链路容量管理，交付性能及吞吐等
性能追踪体系：请求及组件可观测性能力

Apiserver 简介

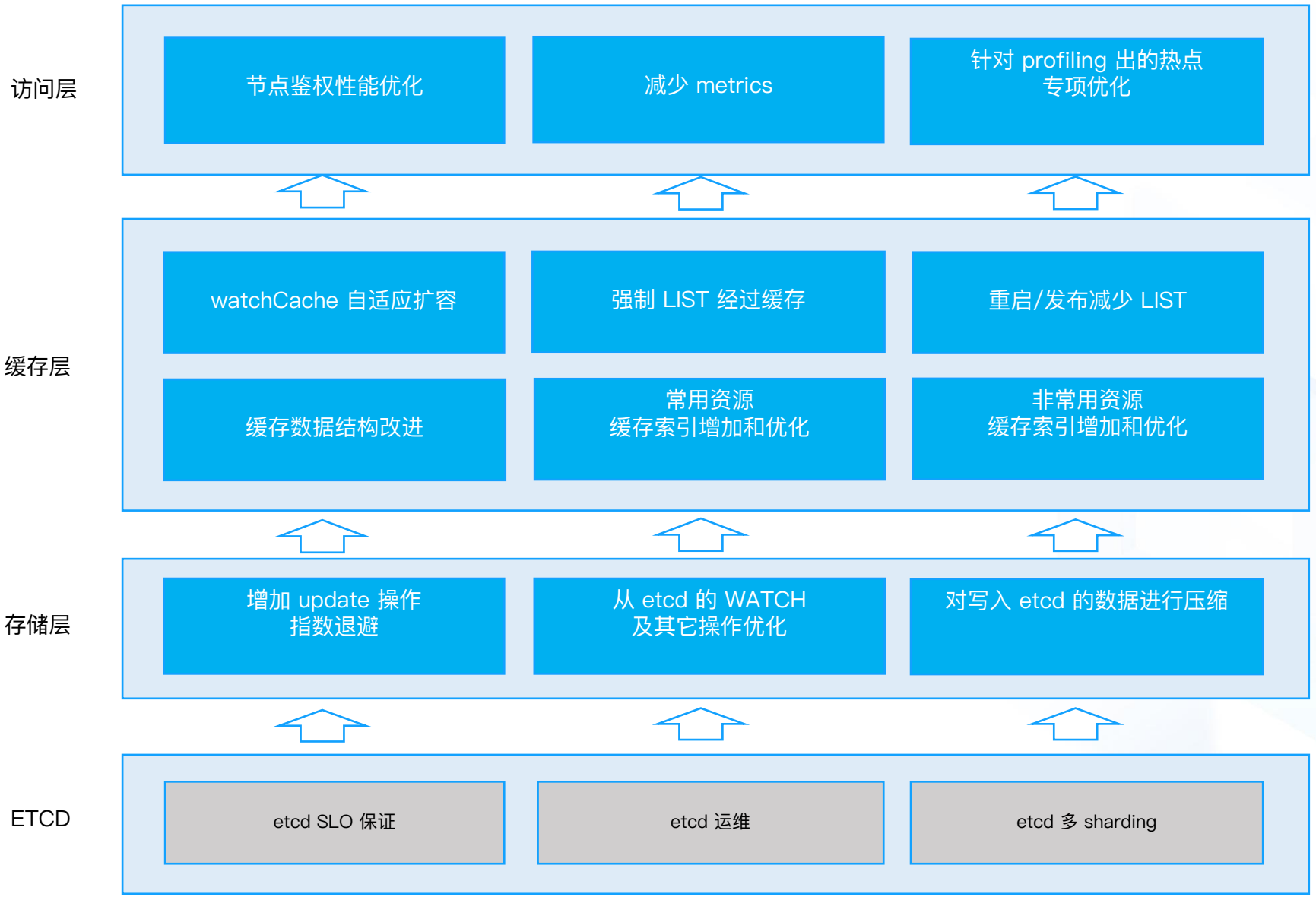


简介

Sigma apiserver 组件是 kubernetes 集群的所有外部请求访问入口，以及 kubernetes 集群内部所有组件的协作枢纽。
apiserver 具备了以下几方面的功能:

1）屏蔽后端数据持久化组件 etcd 的存储细节，并且引入了数据缓存，在此基础上对于数据提供了更多种类的访问机制。
2）通过提供标准 API，使外部访问客户端可以对集群中的资源进行 CRUD 操作。
3）提供了 list-watch 原语，使客户端可以实时获取到资源的状态。

整体优化



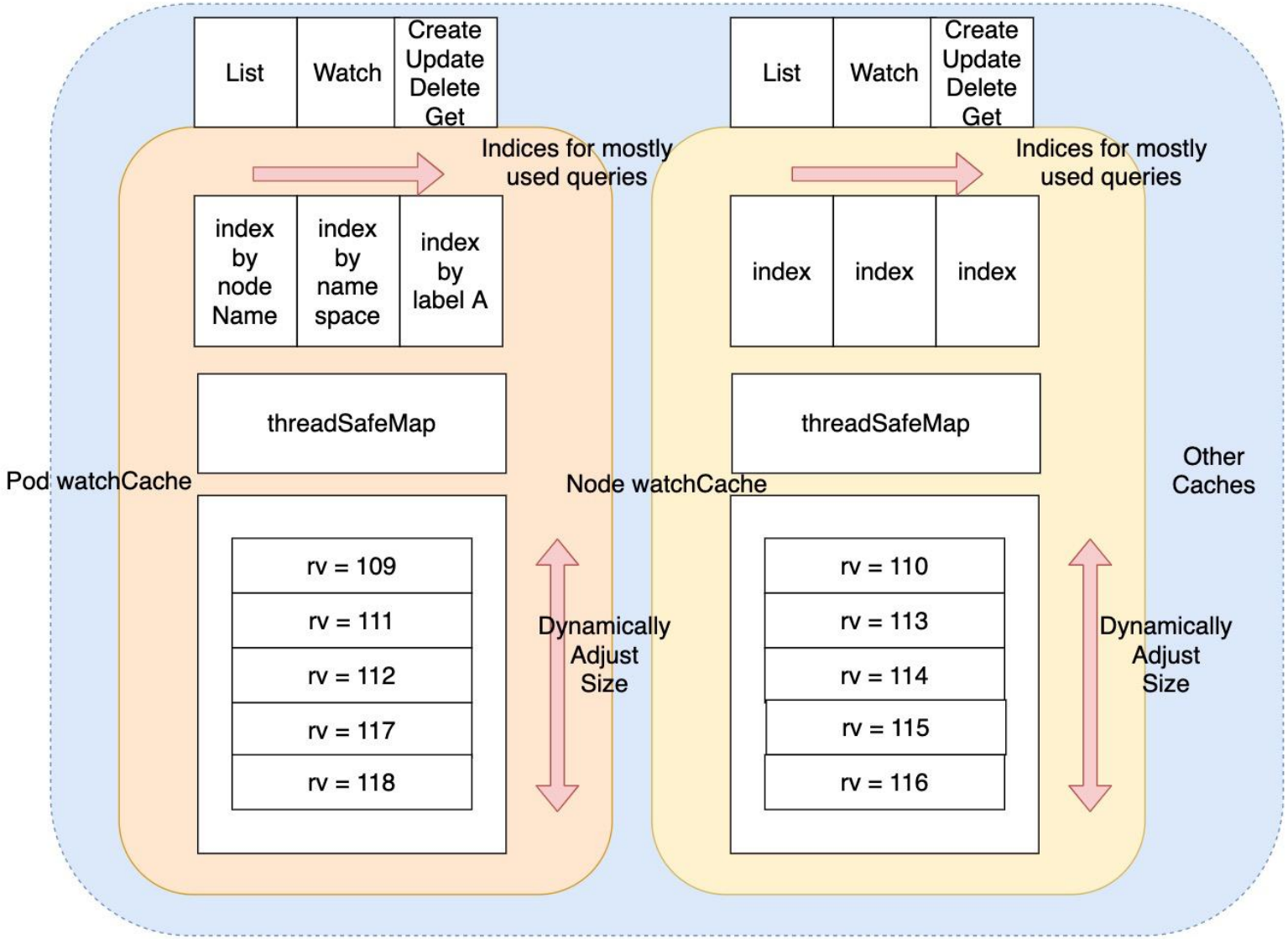
方法论

为了应对如此多的变量对大规模集群带来的复杂影响，我们采用了探索问题本质以不变应万变的方法。
为了可以全面而且系统化地对 apiserver 进行优化，我们由下到上把 apiserver 整体分为三个层面，分别为存储层（storage）、缓存层（cache）、访问层（registry/handler）。

底层的 etcd 是 kubernetes 元数据的存储服务，是 apiserver 的基石。

存储层提供 apiserver 对 etcd 访问功能，包括 apiserver 对 etcd 的 list watch，以及 CRUD 操作。
中间的缓存层相当于是对 etcd 进行了一层封装。1）提供了一层对来自客户端且消耗资源较大的 list-watch 请求的数据缓存，以此提升了 apiserver 的服务承载能力。2）同时，缓存层也提供了按条件搜索的能力。
上面的访问层提供了处理 CRUD 请求的一些特殊逻辑，同时对客户端提供各种资源操作服务。

缓存层



watchCache size 自适应

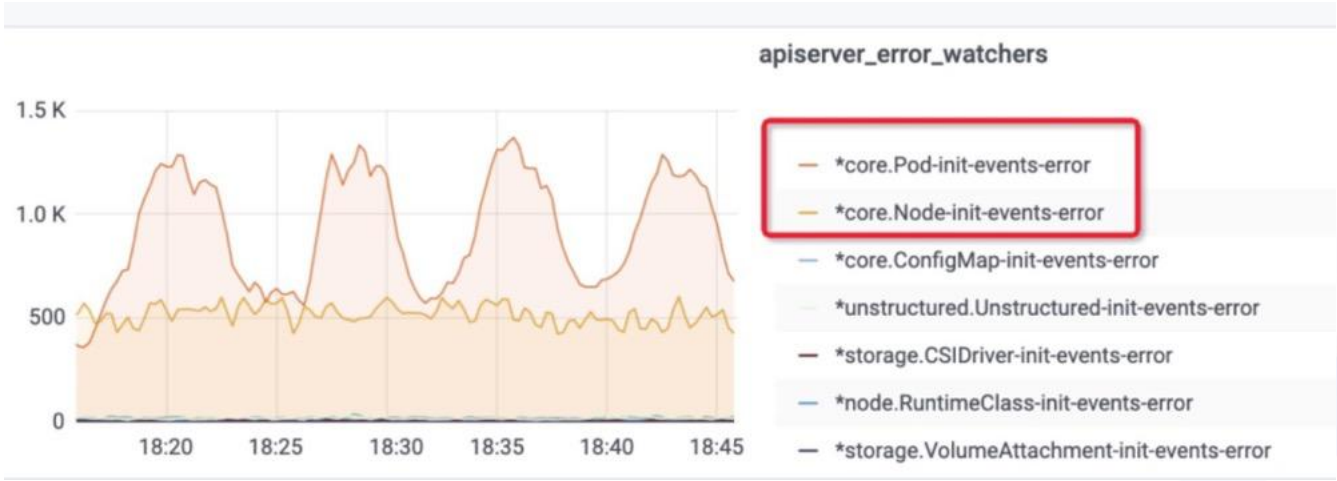
在资源变化率（churn rate）比较大的集群中，apiserver 的 watchCache 大小对 apiserver 的整体稳定性和客户端访问量多少起着很大的作用。太小的 watchCache 会使得客户端的 watch 操作因为在 watchCache 里面查找不到相对应的 resource version 的内容而触发 too old resource version 错误

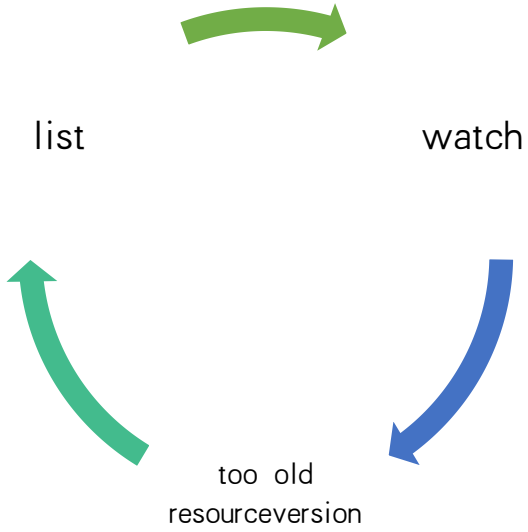
增加 watchCache index

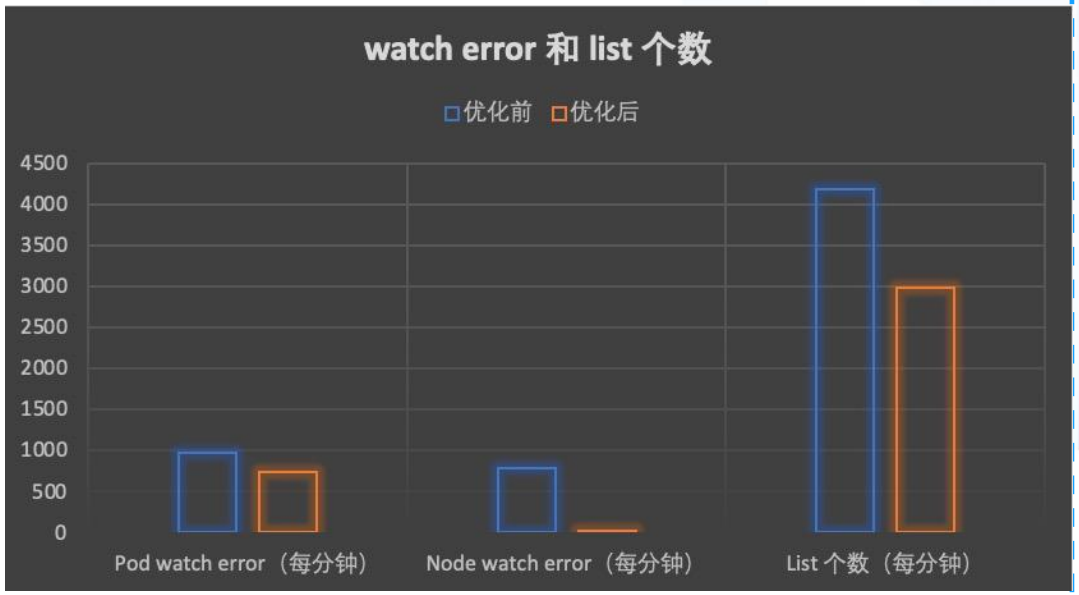
业务方有大量的 list by label 操作

日志分析

缓存层







watchCache size 自适应

在资源变化率（churn rate）比较大的集群中，apiserver 的 watchCache 大小对 apiserver 的整体稳定性和客户端访问量多少起着很大的作用。

太小的 watchCache 会使得客户端的 watch 操作因为在 watchCache 里面查找不到相对应的 resource vesion 的内容而触发 too old resource version 错误，从而触发客户端进行重新 list 操作。而这些重新 list 操作又会进一步对于 apiserver 的性能产生负面的反馈，对整体集群造成影响。极端情况下会触发 list –> watch –> too old resource version –> list 的恶性循环。

太大的 watchCache 又会对于 apiserver 的内存使用造成压力。

因此，动态地调整 apiserver watchCache 的大小，并且选择一个合适的 watchCache size 的上限对于大规模集群来说是非常重要的。
我们对于 watchCache size 进行了动态的调整，根据同一种资源（pod/node/configmap）的变化率(create/delete/update 操作的频次)来动态调整 watchCache 的大小；并且根据集群资源的变化频率以及 list 操作的耗时计算了 watchCache size 大小的上限。

接入层

优化点

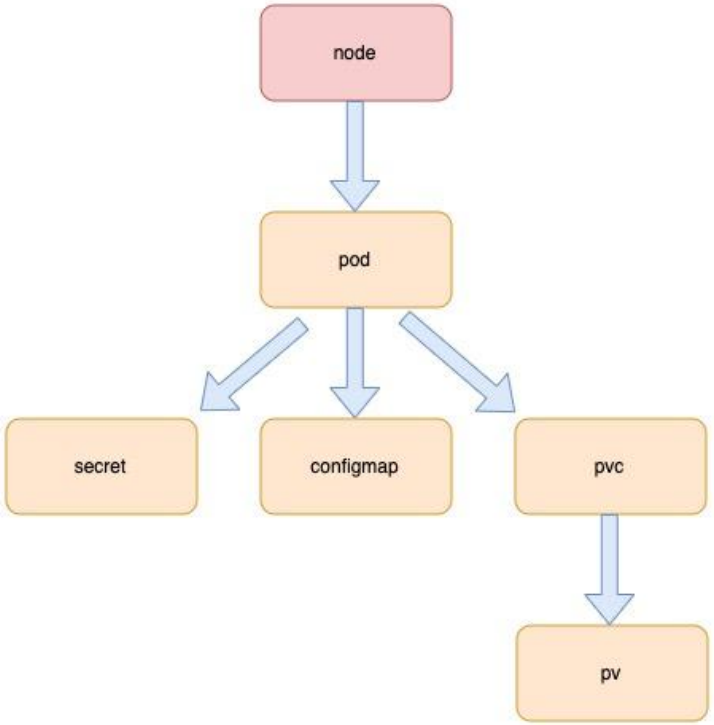
在用户 list event 时可以看到 events.GetAttrs/ToSelectableFields 会占用很多的 cpu，我们修改了 ToSelectableFields， 单体函数的 cpu util 提升 30%，这样在 list event 时候 cpu util 会有所提升

另外，通过 profiling 可以发现，当 metrics 量很大的时候会占用很多 cpu，在削减了 apiserver metrics 的量之后，大幅度降低了 cpu util。

接入层

Sigma apiserver 对于鉴权模型采用的是 Node，RBAC，Webhook，对于节点鉴权，apiserver 会在内存当中构建一个相对来说很大的图结构，用来对 kubelet 对 apiserver 的访问进行鉴权。

当集群出现大量的资源（pod/secret/configmap/pv/pvc）创建或者变更时，这个图结构会进行更新；当 apiserver 进行重启之后，图结构会进行重建。在大规模集群中，我们发现在 apiserver 重启过程中，kubelet 会因为 apiserver 的 node authorizer graph 还在构建当中而导致部分 kubelet 请求会因为权限问题而受阻。定位到是 node authorizer 的问题后，我们也发现了社区的修复方案，并 cherry-pick 回来进行了性能上的修复提升。



- etcd 对于每个存储的资源都会有 1.5MB 大小的限制，并在请求大小超出之后返回 etcdserver: request is too large；为了防止 apiserver 将大于限制的资源写入 etcd，apiserver 通过 limitedReadBody 函数对于大于资源限制的请求进行了限制。我们对 limitedReadBody 函数进行了改进，从 http header 获取 Content-Length 字段来判断 http request body 是否超过了 etcd 的单个资源（pod，node 等）的 1.5MB 的存储上限。

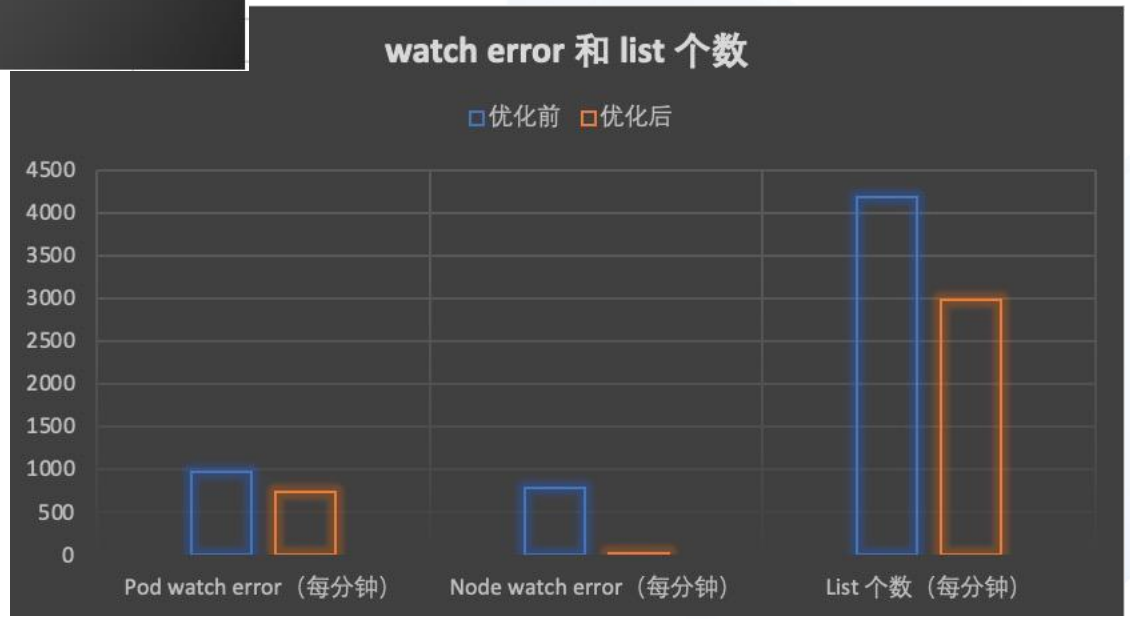
- 当然也不是所有方案都会有所提升。例如，我们进行了一些其它编码方案测试，把 encoding/json 替换成为了 jsoniter。相比之下，apiserver 的 cpu util 虽有降低但是 memory 使用有很大的增高，因此会继续使用默认的 encoding/json。

其它优化

除此之外为了保障 apiserver 的高可用，蚂蚁 kubernetes apiserver 进行了分层分级别的限流，采用了 sentinel-go 加 APF 的限流方案。其中 sentinel-go 来限制总量，进行了 ua 维度，verb 维度等多维度混合限流，防止服务被打垮，APF 来保障不同业务方之间的流量可以公平接入。然而，sentinel-go 中自带了周期性内存采集功能，我们将其去掉之后带来了一定的 cpu 利用率的提升。

另外，我们也在和客户端一起优化 apiserver 的访问行为。截止目前，Sigma 和业务方一起对 blink operator（flink on k8s） / tekton pipeline / spark operator 等服务进行了 apiserver 使用方式方法上的代码优化。

优化效果

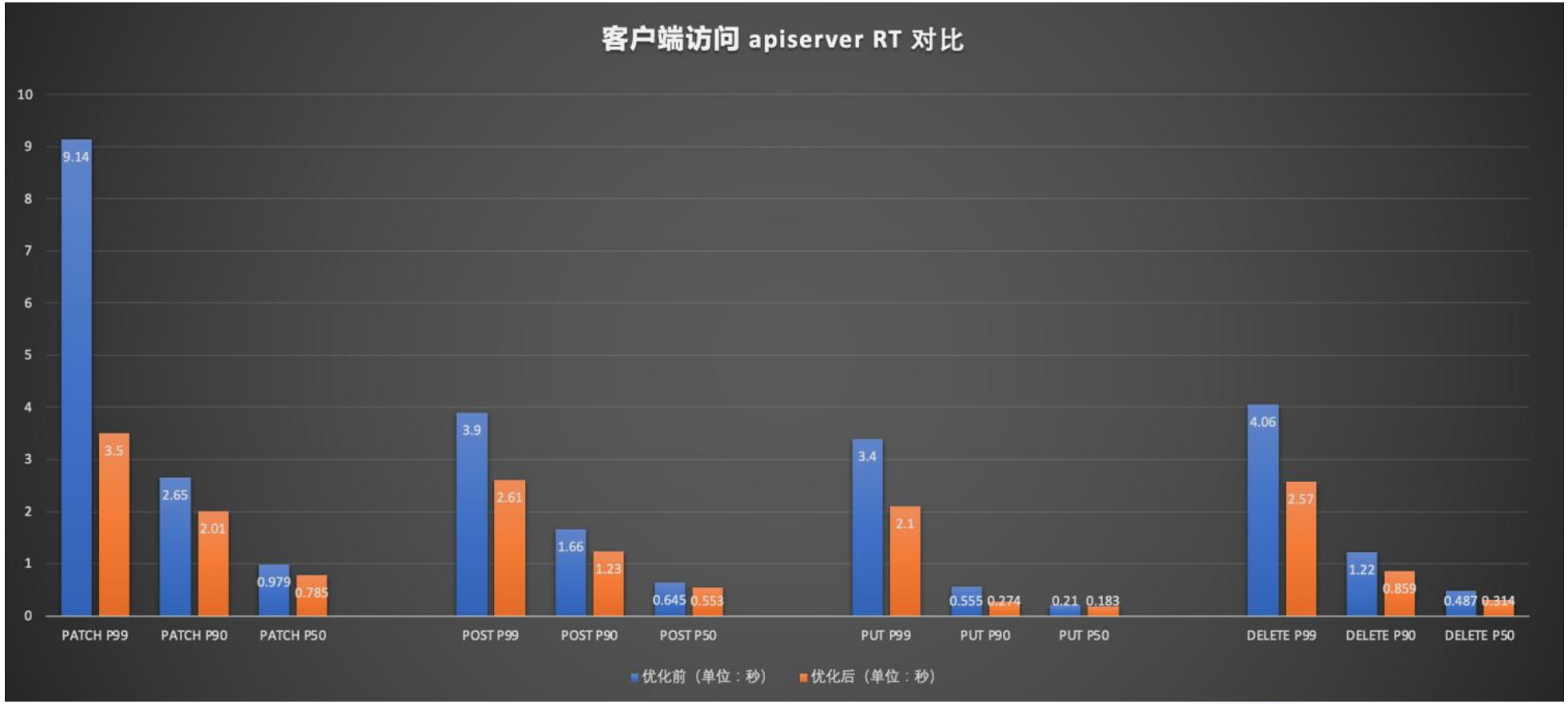


效果

下图分别为云化之后集群分钟级别流量的对比

各类写请求都有明显的上升。其中 create 和 delete 请求比较明显，业务方面的迁移逐步推进，整体集群运行平稳

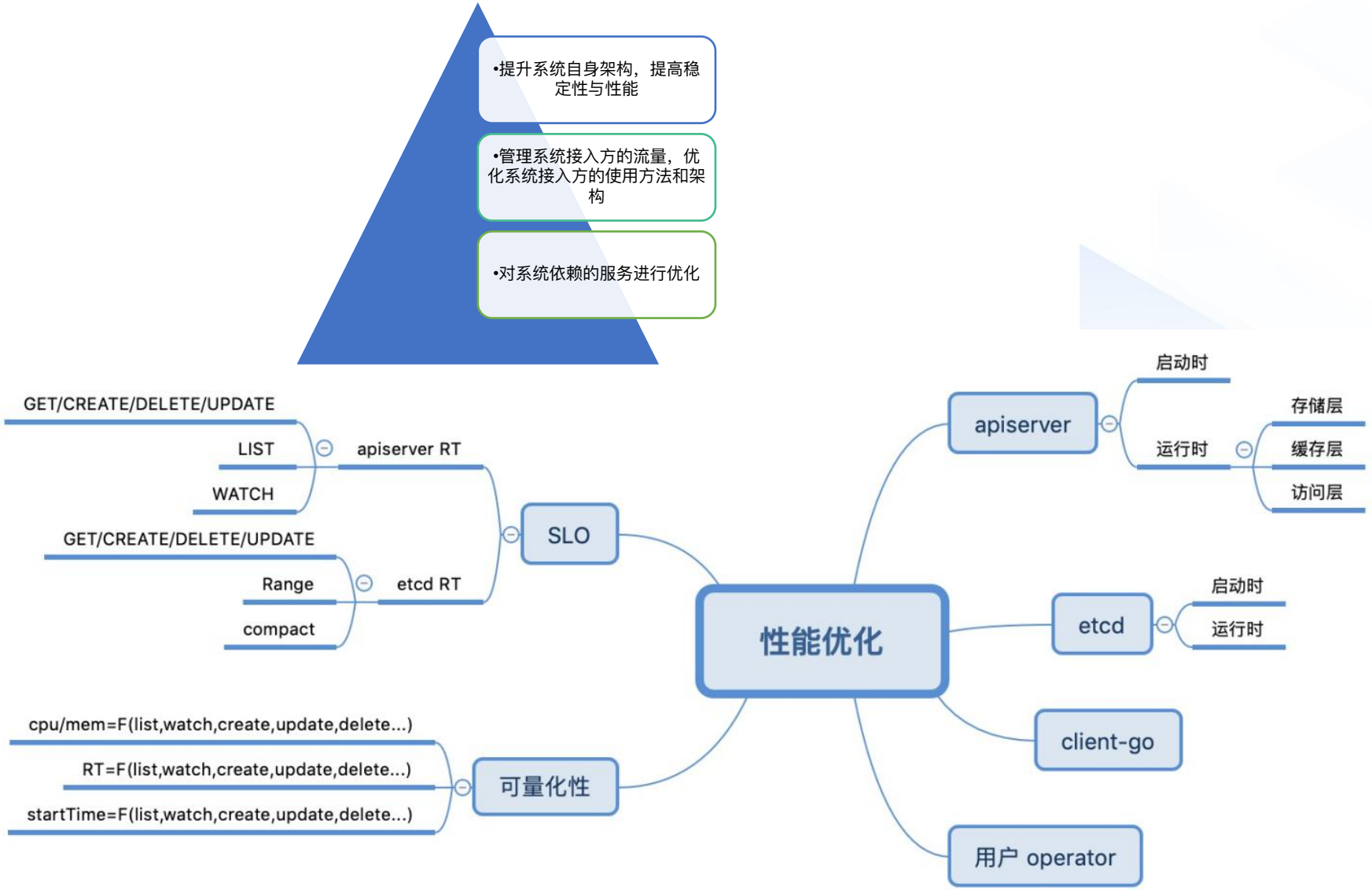
优化效果



效果

写请求的 RT 对于集群和业务的稳定性是最关键的指标之一。经优化过后，客户端访问 apiserver 的各类写请求的 P99，P90，P50 RT 均有明显的下降，并且数值更加趋于平稳，表明 apiserver 在向着高效且稳定的方向发展。

未来展望



下一步规划

针对 apiserver 自身，一些可能的优化点包括：优化 apiserver 启动总时间，提升 watchCache 构建速度；threadSafeStore 数据结构优化；对 get 操作采用缓存；对 apiserver 存入 etcd 的数据进行压缩，减小数据大小，借此提升 etcd 性能 等等。

除了优化 apiserver 本身之外，蚂蚁 Sigma 团队也在致力于优化 apiserver 上下游的组件。例如 etcd 多 sharding，异步化等高效方案；以及对于各种大数据实时和离线任务的 operator 的整体链路的优化。

当然 SLO 的牵引必不可少，同时也会在各个指标的量化上进行增强。只有这些协调成为一个有机的整体，才能说我们有可能达到为运行在基础设施上面的业务方提供了优质的服务。

云原生社区 Cloud Native Community | SOFASTACK™

Thank You

微信扫码进群
与五湖四海的开发者们
进行技术交流，探索技术创新

扫码关注公众号，参与活动抽奖
与 2.8W+ 技术精英
交流技术干货 & 开源组件