又拍云日志服务架构设计实践

张超

@ 云片活动开放日

2018-09-16

我自己

Alex Zhang

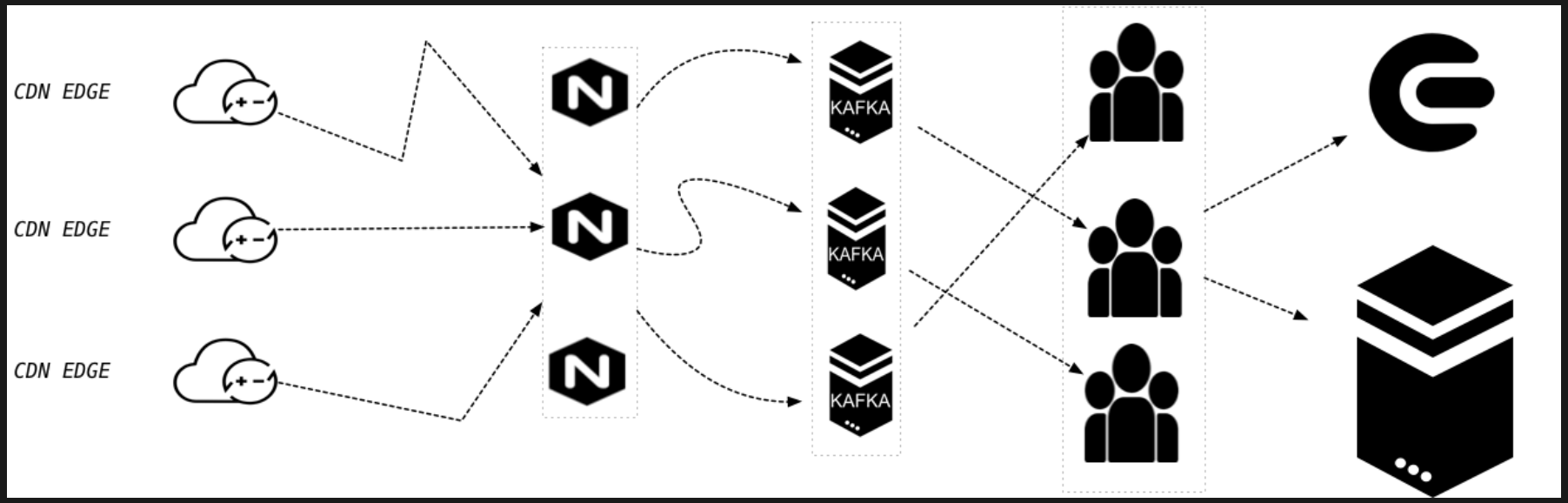
又拍云系统开发工程师，负责又拍云 CDN 反向代理层组件的维护，平时喜欢研究一些开源软件，比如 Nginx、OpenResty 和 Redis，并混迹在这些开源软件的社区中，专注于服务端技术的研究。

- Github: https://github.com/tokers
- Email: zchao1995@gmail.com

为什么要提供日志服务

- 日志归档，根据不同的服务名提供下载服务
- 提供近实时、多维度的日志分析，问题排查，并对接到内部报警系统
- 日志离线分析；复杂的数据模型计算和分析，定期生成报表

日志服务系统概览



- 各个 CDN 边缘节点上传日志数据到日志服务代理层
- 代理层根据日志类型将日志保存到不同的 Kafka topic
- 不同的日志消费者消费不同的 Kafka topic 里的数据
- 根据需求决定最终去向，比如保存到 HDFS 集群或者 ES 集群

潜在的问题

- 由于 CDN 的特点，边缘机房分布在全国各地，通过公网上报日志，难免遇到网络不理想的情况
- 日志消费者消费能力有限时，会导致 Kafka 队列发生堆积

日志上报的方案

针对 Nginx 日志的上报方案：

- 脚本周期性地向 Nginx 发送信号进行日志切割，然后进行上传；
- 或者直接配置 Nginx，使得 access.log 发送到 syslog；

考虑到又拍云 CDN 服务产生的日志数据比较复杂，而 Nginx access.log 描述能力相对薄弱，因此采用了以下的方案：

- CDN 边缘机房引入单独的 log agent 服务，由 log agent 对接数据中心的日志代理层
- log agent 内置 disk queue
- CDN 边缘服务引入 lua-resty-logger-socket，将日志发送到本机的 log agent（127.0.0.1）

lua-resty-logger-socket

https://github.com/cloudflare/lua-resty-logger-socket

该库的核心是一个内存 buffer，buffer 的内容只会在达到一定大小（flush_limit）之后才将数据冲刷出去，如果发送失败，内容不会被丢弃；如果缓冲的数据大小超过一个硬性值（drop_limit），那么当前的日志会被丢弃。
该库基于 ngx_lua 的 Cosocket 技术，Cosocket 通过利用 Lua coroutine 的 yield/resume 特性，完美结合了 Nginx 的事件框架，因此是百分百非阻塞的，这避免了 access.log 写磁盘带来的阻塞问题。和又拍云 CDN 服务的软件架构非常契合。

log agent

CDN 边缘日志 agent 服务，内部称之为“logger”，接收 CDN 边缘服务发送过来的日志，然后转发。

logger 会周期性地将日志转发出去，因此网络质量理想的情况下，日志只会被短暂的缓冲在内存中。

当网络异常时，内存里的日志会越来越多，当超过某个阈值的时候，日志会自动加入到 disk queue 中，这是一个基于文件系统的先进先出队列，当需要转发日志的时候，会优先处理在磁盘中排队的日志数据。

通过这套机制保证日志即使在网络情况不理想的时候也不会被丢弃。

如此，CDN 边缘机房日志上报可靠性得到了保障。

“消费”跟不上“生产”

消费者不给力，直接会导致日志数据无法及时传递到下一层，破坏了系统近实时的传输转发能力。

如何增强消费能力?

消费者在设计上需要支持水平可扩容，才能从容应对日志量突增的情况，避免服务可用性降低。

消费者容器化

如果新增一种消费者程序或者扩容时，还需要：

- 提前申请物理机
- 人工推送服务上线
- 部署对应监控报警程序
- ...

操作周期过长，紧急时很难及时完成，可能还会带来不小的损失。

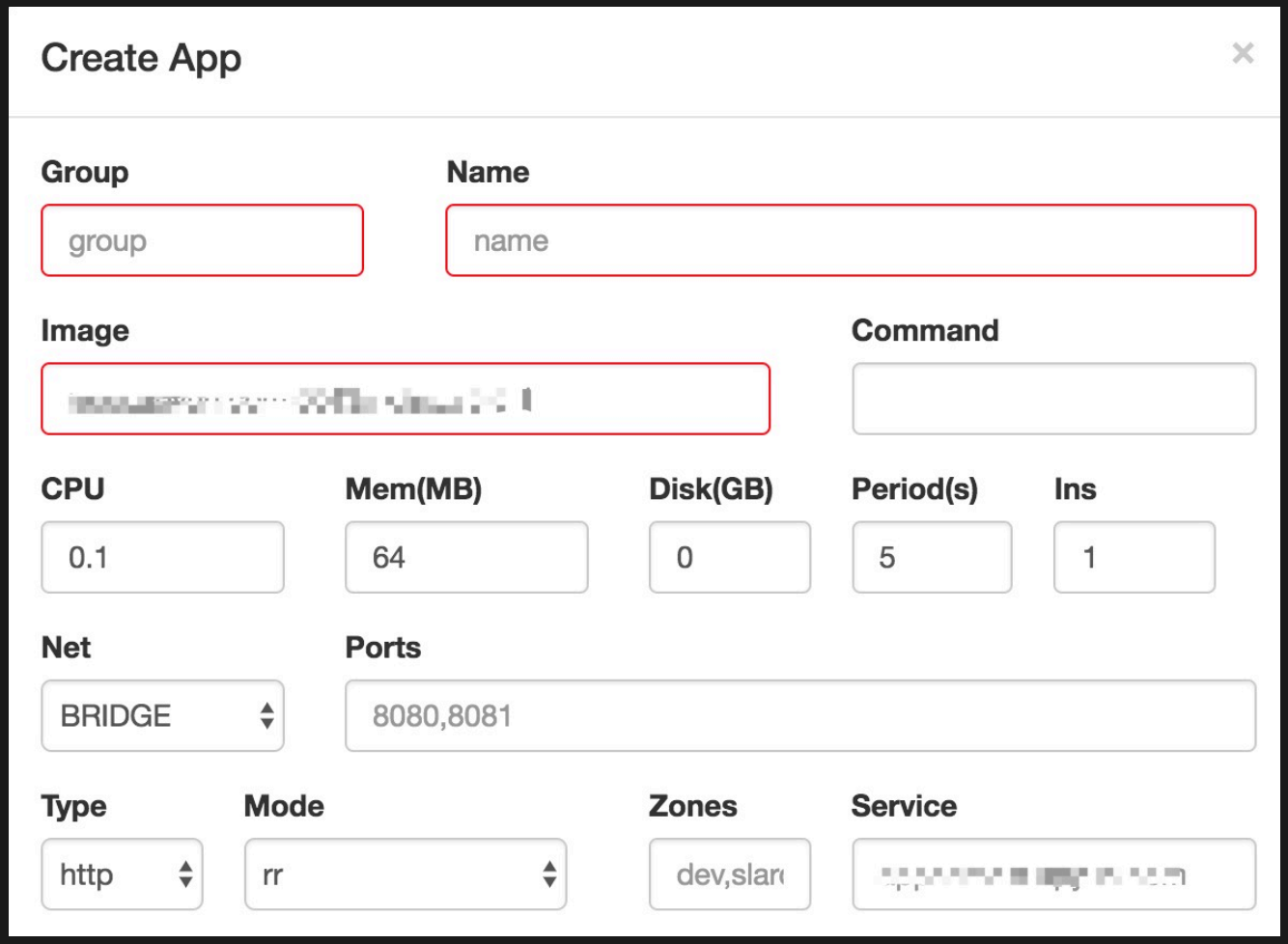
https://www.upyun.com/products/dockercloud

相比之下，使用容器云平台的优势就很明显了：

- 资源池化
- 故障转移，弹性伸缩
- 运维成本低
- 自带监控报警

消费者容器化

图: 在又拍云容器云平台上创建一个服务



消费者容器化

图: 一个日志消费者服务的部分运行监控

消费者设计

日志种类繁多，需求也不尽相同，需要一个通用的消费者框架，避免重复劳动，满足不同的需求——Morgans 项目。

为什么不使用开源的解决方案?

- Logstash 使用了 JRuby 作为插件编写语言，而 JRuby 对资源的消耗比较重
- Logstash 本身存在不少缺陷，容易踩坑
- 某些业务场景十分复杂，例如需要适配客户的 FTP 服务器；或者某些客户的服务器需要先进行鉴权；又或者日志需要进行分类和打包
- Mozilla Heka 项目多年未更新，许多问题也得不到修复
- 自行设计能够更加贴近业务需求，更加轻量，即使出错也能够快速定位解决

消费者内部结构

一切皆模块——

- 从 Kafka 或其他日志源消费原始格式的日志——Input module
- 根据需求转换成目标类型的日志——Filter module
- 根据需求转发或者暂存到目标服务——Output module
- 消费者框架代码——Core module

模块设计

Morgans 的模块设计借鉴了 Nginx，每个模块拥有自己的配置结构，可以自行注册配置指令，注册各类功能的钩子函数。这些钩子函数在不同的阶段发挥各自的作用。

```
type MorgansModule struct {
    Name              string
    Commands          []MorgansCommand
    Type              int
    Index             int
    PostParse         func(*MorgansConfig) error
    PreParse          func(*MorgansConfig) error
    InitProcess       func() error
    MainConf          interface{}
}
```

比如：

- 处理到某条配置文件指令时，某个模块的某个配置指令钩子会被调用，从而正确解析到该配置
- 配置文件解析完毕后，模块的 PostParse 钩子会被调用
- 准备开始工作时，模块的 InitProcess 钩子会被调用

模块扩展和集成

同样 Morgans 还支持添加自定义的模块。

```
# ./auto/configure \
  --with-input_tcp_module \
  --add-module=/path/to/mymodule

# cat /path/to/mymodule/config
MODULE_NAME="my filter module"
MODULE_TYPE=FILTER
MODULE_INIT=MorgansFilterInitMyModule
MODULE_FIND_CONFIG=MorgansFilterMyFind
MODULE_FILE="morgans_filter_my_module.go"
```

通过只编译需要的模块，还可以减小编译得到的二进制文件大小。

灵活的配置文件设计

- 指令式的配置方式，简洁而不失描述能力
- 支持从外部的 key-value 存储中加载配置
- 结合消费者框架内置的变量系统，实现了变量插值功能

示例

```
input::kafka mykafka {
    topic mytopic;
    delimiter "\n";
    broker 192.168.1.10:9092;
    broker 192.168.1.11:9092;
    broker 192.168.1.12:9092;
    broker 192.168.1.13:9092;
    broker 192.168.1.14:9092;
    group mygroup;
    offset earliest;
}
filter::common myfilter;
output::hdfs myhdfs {
    replication 3;
    block_size 1m;
    user root;
    namenode 192.168.1.11:8020;
    sync_size 40m;
    sync 1m;
    filepath /hdfs/output/$hostname/log.$mylog_timestamp;
}
```

变量插值

Morgans 内置了一套简易的运行时变量系统，和 Nginx 的类似，这套变量系统关联到每个日志对象，每个模块可以自行注册变量，以及变量被使用时需要调用的钩子，钩子则可以关联到一个日志对象的上下文。
使用分成两步，第一步会处理源字符串当中每个变量和常量并解析得到 ComplexValue 对象，第二步是发生在需要使用时，调用这项变量和常量的 handler 从而拼接得到经过解释后的完整字符串。

```
c := corgi.New()
c.RegisterNewVariable(&corgi.Variable{
    Name: "hostname",
    Get: HostnameHandler,
})

cv := c.Parse("Hostname is ${hostname}")
data := c.Code(cv)
```

https://github.com/tokers/corgi

运行现状

- 接管了所有日志定制和内部监控数据的处理
- 消耗的 CPU、内存均比较少，稳定且基本无害

Thanks!