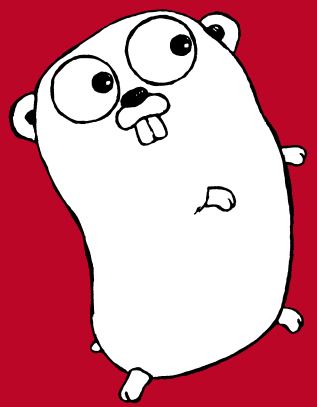
Golang与高性能DSP竞价系统



By @QLeelulu

舜飞科技

专业DSP解决方案供应商

什么是RTB与DSP

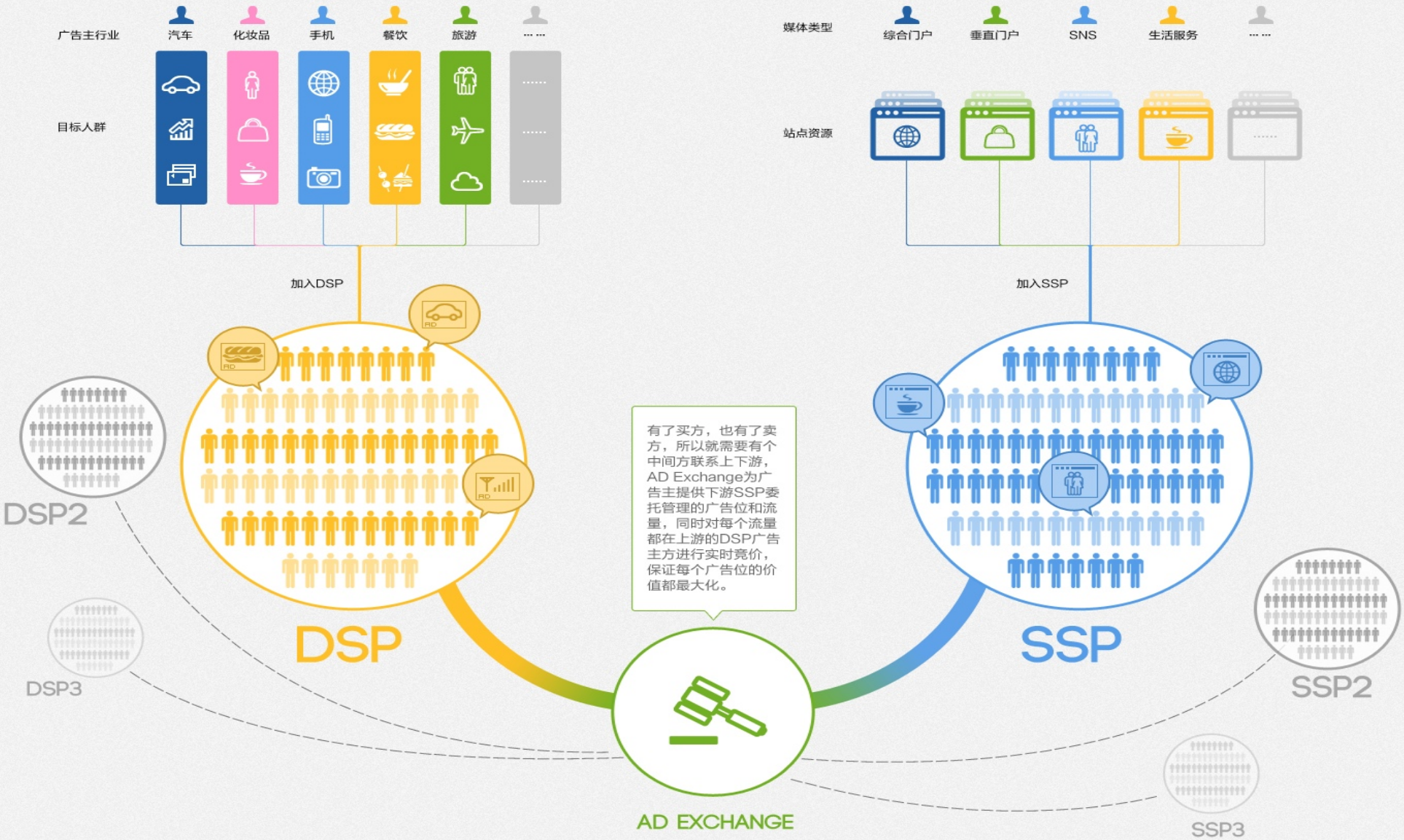
- **RTB**: Real-time Bidding，实时竞价，允许广告买家根据活动目标、目标人群以及费用门槛等因素对每一个广告及每次广告展示的费用进行竞价。
- **DSP**: Demand Side Platform，需求方平台，允许广告客户和广告机构更方便地访问，以及更有效地购买广告库存，因为该平台汇集了各种广告交易平台的库存。

DSP 需求方平台
Demand Side Platform

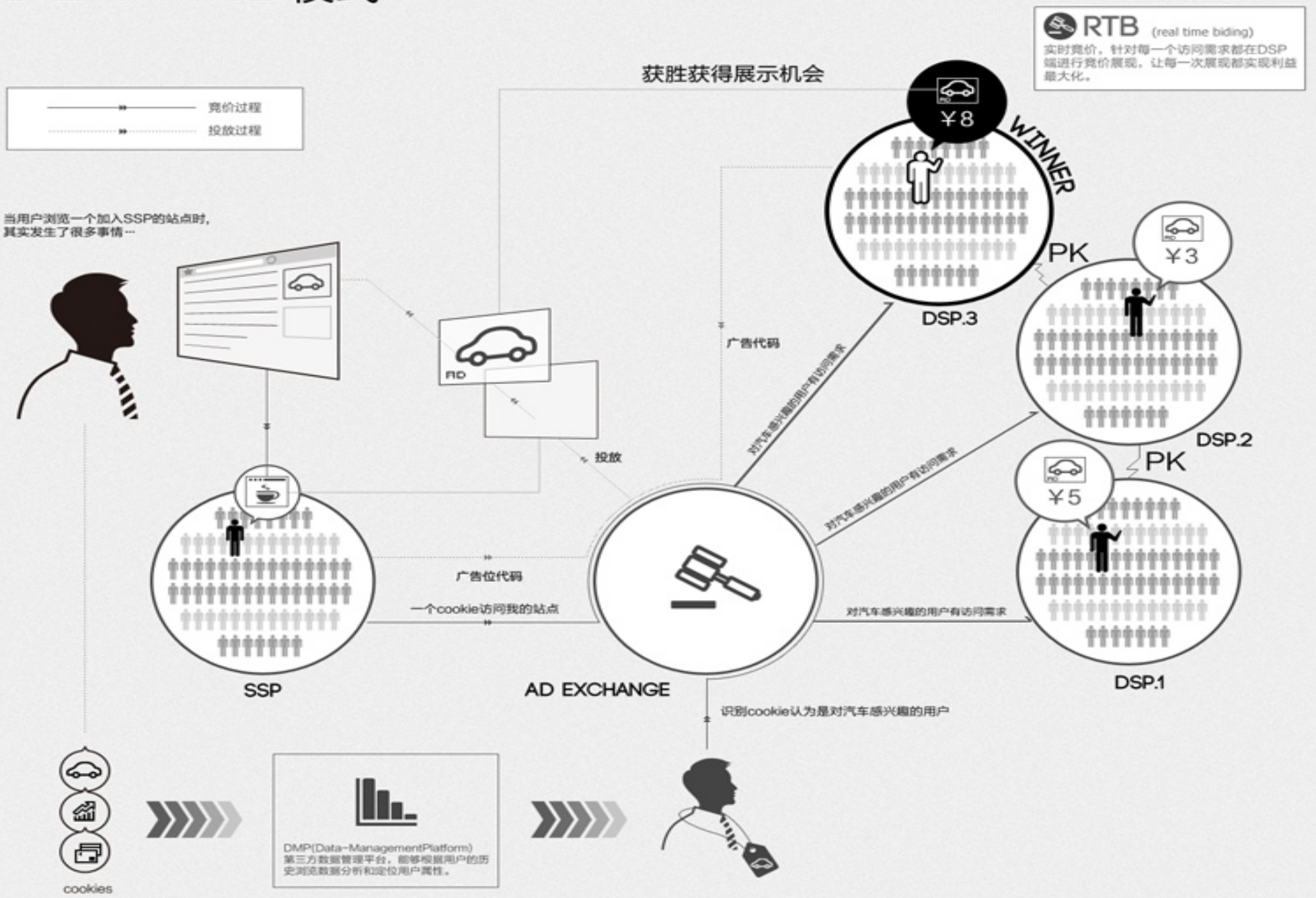
互联网里有成千上万的广告主，他们急需推广自己的产品，寻找优质的媒介和精准的目标用户，优化广告投放策略，提高投入产出比，这种情况下，就诞生了为他们提供服务的专业化平台，DSP。简单的讲，DSP就是广告主服务平台，广告主可以在平台上设置广告的目标受众、投放地域、广告出价等等。

SSP 供应方平台
Supply Side Platform

互联网里也有成千上万拥有丰富媒体资源和用户流量的网站，他们急需把庞大的流量变现来发展壮大，同时还希望每一个流量都能够达到最大的收益。简单的讲，SSP就是一个媒体服务平台，站长们可以在ssp上管理自己的广告位，控制广告的展现，设置补余等等。



AD EXCHANGE 模式



DSP竞价系统的挑战

- 高并发量请求处理（峰值QPS 20万）
- 每天上百亿竞价请求
- 每个竞价请求要在100毫秒内响应（包含网络延迟）
- 复杂的出价算法与逻辑

100毫秒内要做些什么

- 竞价请求解析（JSON 或 Google Protobuf）
- 根据广告位属性过滤活动
- 根据客户端信息过滤活动（浏览器、操作系统类型等）
- 根据地区过滤活动
- 查询Cookie Mapping得到访客在DSP系统的唯一ID
- 根据用户看过广告的频次过滤活动
- 根据访客的人群属性过滤活动
- 根据活动的出价选择胜出的活动
- 其他更细致的过滤条件

为什么选择Golang

第一次签入

- 2012-11-29
- 在 Go 1.1 发布之后

初始化

QLeelulu authored 12 months ago

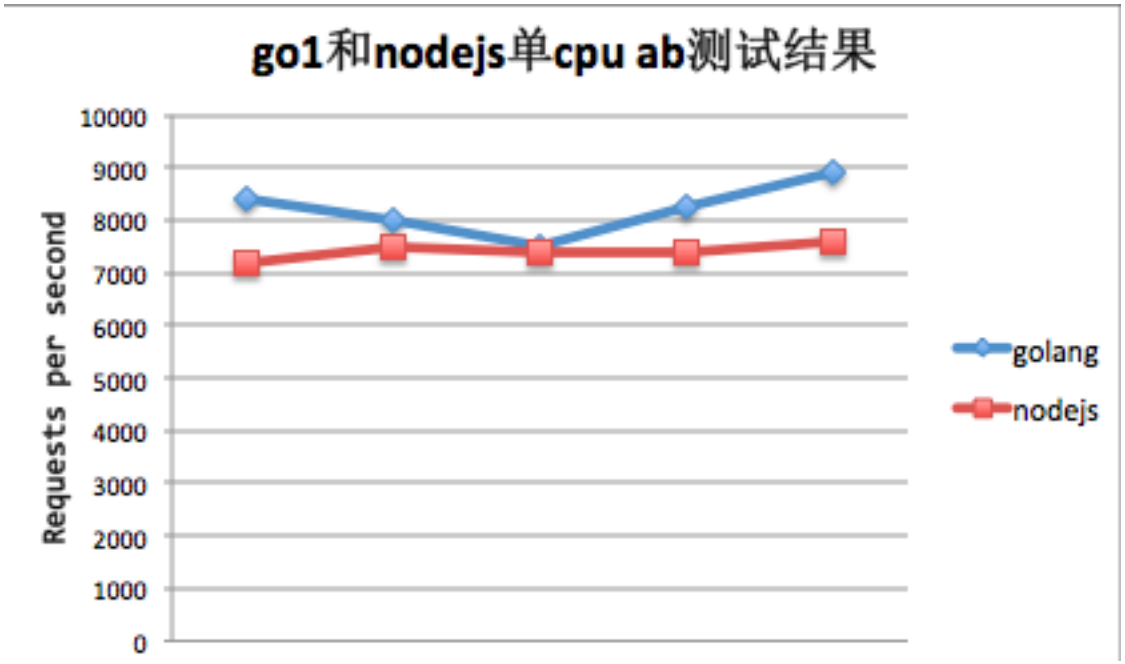
Nov 29, 2012 12:59pm

Showing 24 changed files

+ README.md
+ bin/server.go
+ config/base.json
+ config/config.go
+ config/config_test.go
+ install/README.md
+ install/install-go-deps.sh
+ library/db/db.go
+ library/db/db_test.go
+ library/db/info_db.go
+ library/db/info_db_test.go
+ library/log/log.go

为什么选择Golang

- http包的HelloWorld性能测试



Via: http://www.cnblogs.com/QLeelulu/archive/2012/08/12/2635261.html

为什么选择Golang

- 高性能、天生并发支持
- 性能敏感的模块可以直接使用C编写（当时是这么认为的）
- 编译为本地机器码，部署方便
- 快速上手，学习成本低
- 标准库基本够用
- 带GC（当时不了解GC的性能问题）
- 自带单元测试、性能测试、性能分析工具
- 开发效率不低

备选

- C++
- NodeJS
- Golang ☑

竞价接口

HTTP竞价接口

- 直接用golang的http包
- 只使用gorilla/mux做简单的请求路由
- 封装简单的HTTPBaseHandler

简单的HTTPBaseHandler

```
func (b BaseHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    stv := time.Now()
    defer func() {
        MonitorCount++

        err_ := recover()
        if err_ == nil {
            return
        }
        var stack string
        if DEBUG {
            var buf bytes.Buffer
            buf.Write(debug.Stack())
            stack = buf.String()
            fmt.Fprintf(w, "%v\r\n%s", err_, stack)
        } else {
            fmt.Fprintf(w, "Internal Server Error", err_)
        }
        log.Errorln(fmt.Sprintf("%v", err_), "\n", stack)
        return
    }()

    b.Handle(w, r)
}
 }
```

简单监控接口: GO

```go
//监控handlers，在初始化时启动，提供url接口查询qps信息
func init() {
    if Monitor.Open {
        upTimeStr = time.Now().Format("2006-01-02 15:04:05")
        //只取日期
        verDate := strings.Split(config.VERSION_DATE, " ")[0]
        versionInfo = verDate + "~" + config.VERSION_GIT

        //定期刷新监控的最新信息
        go func() {
            for {
                time.Sleep(qpsInterval * time.Second)
                currentQps = float64((MonitorCount - perTotalCount)) / float64(qpsInterval)
                perTotalCount = MonitorCount
            }
        }()

        http.HandleFunc("/sf-monitor", func(w http.ResponseWriter, r *http.Request) {
            w.Write([]byte(fmt.Sprintf(`{"mem":"%s","ver":"%s","up_time":"%s","total_count":%d,
        })
    }
}
```

简单监控接口

qps情况

名称	版本	起始时间	内存	请求总数	10秒qps	文本信息
whisky-jump	总计		0	20916355	4.1 QPS	
-dsp_whisky_	2015-03-16~git-2ddd374	2015-03-16 18:58:53	89.6MB	5731144	1	[]
-dsp_whisky_	2015-03-16~git-2ddd374	2015-03-16 18:58:32	94.5MB	5715058	1.3	[]
-dsp_whisky_	2015-03-16~git-2ddd374	2015-03-16 18:58:52	85.4MB	5683466	1.6	[]
cookiemap	总计		0	231402143	4409.79999999999	
-dsp_cookiemap_	2015-04-21~git-a4d5c1b	2015-04-21 15:49:36	79.5MB	57780321	1053.3	[]
-dsp_cookiemap_	2015-04-21~git-a4d5c1b	2015-04-21 15:49:11	82.0MB	57968298	1126.9	[]
-dsp_cookiemap_	2015-04-21~git-a4d5c1b	2015-04-21 15:49:36	83.1MB	57649576	1133.2	[]
-dsp_cookiemap_	2015-04-21~git-a4d5c1b	2015-04-21 15:49:39	89.4MB	58003948	1096.4	[]

预警接口: Interface

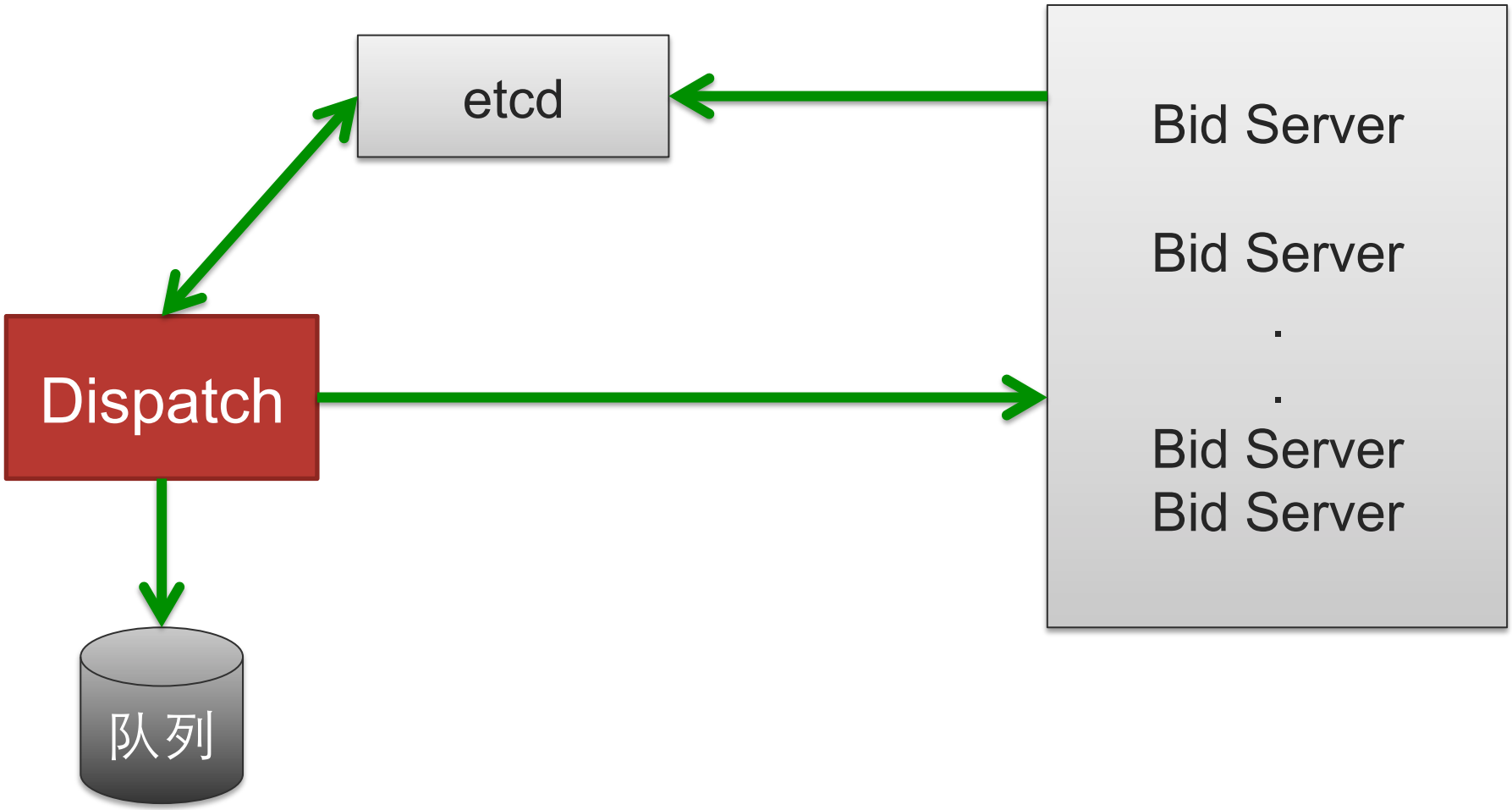
```
//实现该接口表示模块可监控的
type Monitored interface {
    //用于获取状态,当被调用时,返回模块的状态信息
    MonitorStatus() ModuleStatus
}
```

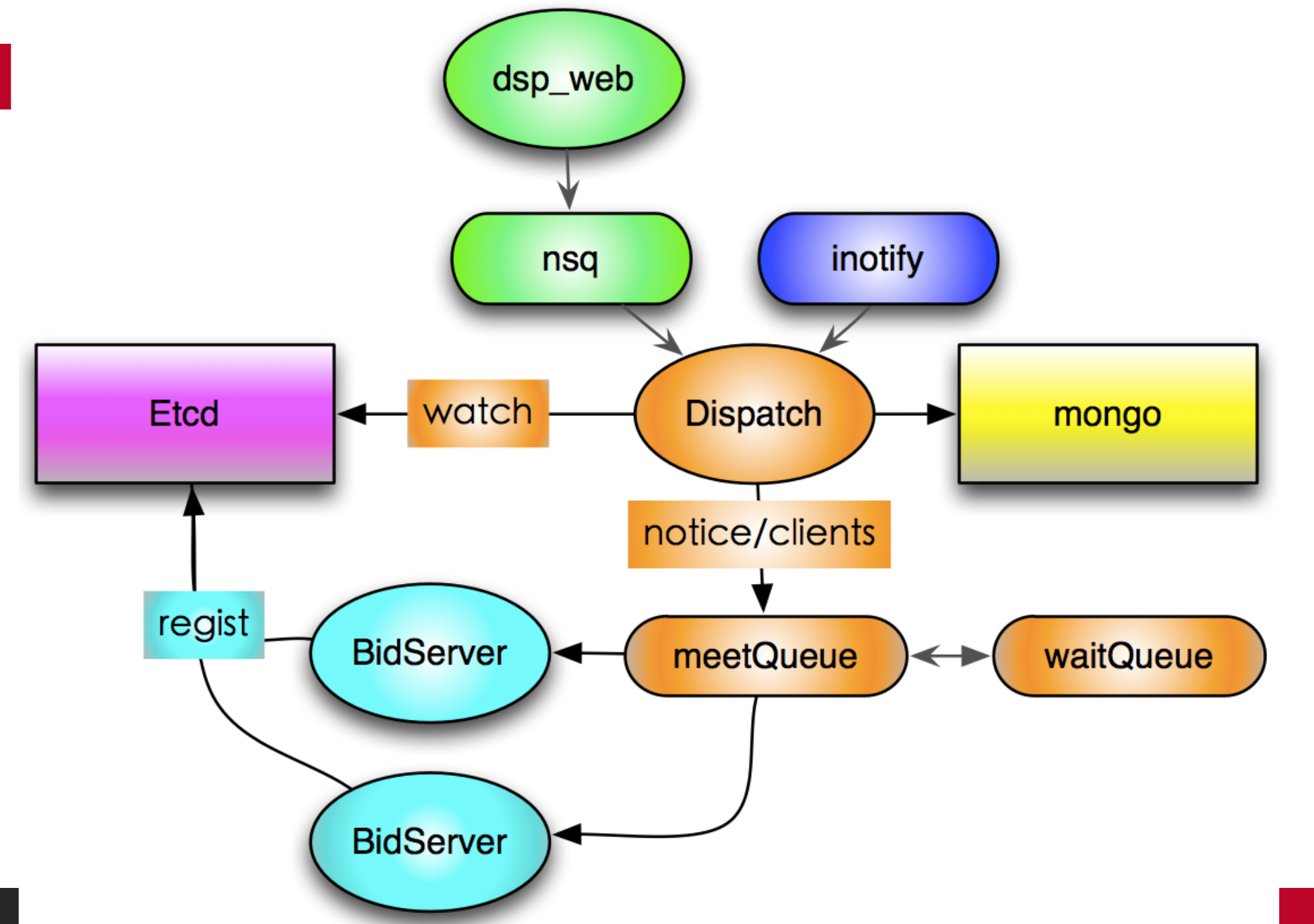
```
//向监控程序注册模块,使得应用的该模块可以被监控
//@m:被监控的模块
func RegisterMonitor(m Monitored) error {
    registerLock.Lock()
    defer registerLock.Unlock()
    registerModules = append(registerModules, m)
    return nil
}
```

预警接口: Redis

```go
//检测redis的状态
type MonitoredRedis struct {
    Name   string
    Client *redis.Client
}

func (m MonitoredRedis) MonitorStatus() ModuleStatus {
    return MonitorRedis(m.Name, m.Client)
}

func MonitorRedis(name string, client *redis.Client) ModuleStatus {
    err := MustPing(client, 5)
    if err != nil {
        return pingFail(name)
    }
    return fine(name)
}

// 注册Redis预警监控
monitor.RegisterMonitor(MonitoredRedis{})
```

Dispatch: 数据、状态同步




dsp_web
nsq
inotify
Etcd
watch
Dispatch
mongo
notice/clients
regist
BidServer
meetQueue
waitQueue
BidServer

Redis集群

- 重度使用Redis
- 存放的数据包括
  - CookieMapping
  - 曝光频次
  - DMP人群数据
  - 实时费用消耗
  - 其他

Redis集群

- Server端：等待Redis官方（当时还没有的）
- Proxy中间代理：twemproxy，维护方便，有一定的性能消耗
- Client端：配置、维护麻烦，几乎无性能损耗

Redis集群

- 最终在Client端实现
- 一致性hash: github.com/stathat/consistent
- 预先开启足够多的Redis实例，预防增加节点带来的数据迁移麻烦

```go
type HashDB struct {
    pool *consistent.Consistent
    rdbs map[string]*db.MyRedis
}

func NewHashDB(copyNum int, dbs map[string]*db.MyRedis) (hdb *HashDB) {
    hdb = &HashDB{
        pool: consistent.New(),
        rdbs: dbs,
    }
    var servers []string
    for server, _ := range dbs {
        servers = append(servers, server)
    }
    hdb.pool.NumberOfReplicas = copyNum
    hdb.pool.Set(servers)
    return
}

func (this *HashDB) GetRdb(key string) (rdb *db.MyRedis, err error) {
    serverName, err := this.pool.Get(key)
    rdb = this.rdbs[serverName]
    return
}
```

Redis集群

- 500个Redis实例
- 占用600G内存
- 峰值QPS在50万

名称	内存占用	连接数	每秒Qps	保存时间
总状态	634.09 G	449554	490882 QPS	总计476个redis,1个存在问题
[redacted]-6114-[redacted]	119.29 M	145	5355 QPS	总计1个redis,0个存在问题
192.168.3.34:6114	119.29M	145	5355	2015-01-31 11:23:36
cost	790.53 M	10937	27821 QPS	总计12个redis,0个存在问题
192.168.3.103:6380	66.57M	957	2406	2015-04-22 17:08:02
192.168.3.103:6381	67.11M	969	3535	2015-04-22 17:08:02
192.168.3.103:6382	67.97M	987	2908	2015-04-22 17:08:02
192.168.3.104:6380	62.28M	746	2685	2015-04-22 17:08:02
192.168.3.104:6381	63.70M	816	2754	2015-04-22 17:08:02
192.168.3.104:6382	61.98M	735	1656	2015-04-22 17:08:02

Redis跨机房同步

- 直接配置主从同步
- 机房间网络差，非常差（北京<->香港，北京<->北美）
- 会触发Redis全量同步（超过repl-backlog-size时）

需要替代方案

Redis跨机房同步

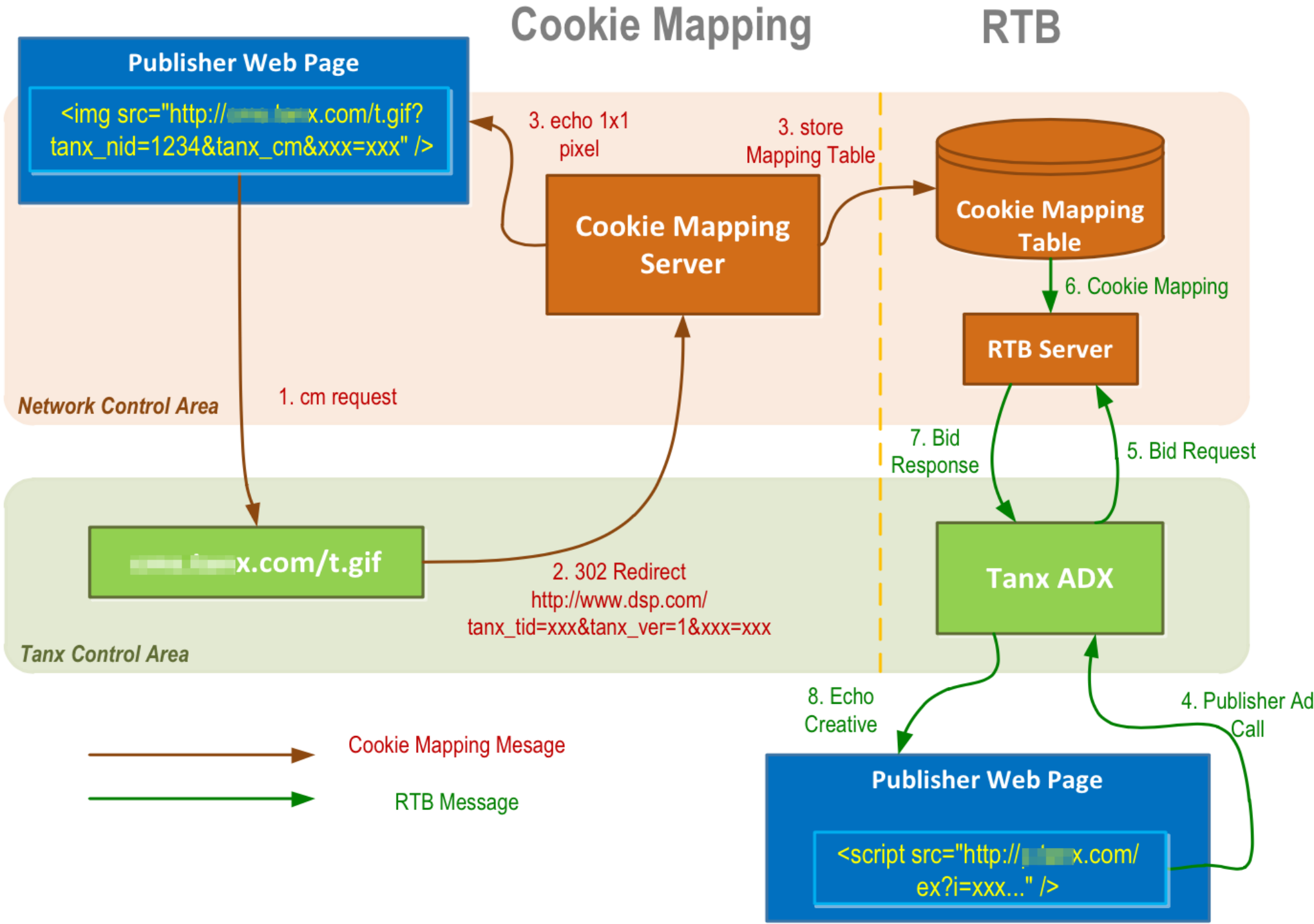
- 取消Redis的主从同步
- 写主Redis时，同时写一份到NSQ，异步写入其他机房
- 使用SoftLayer的香港云主机作为中转（why?）

Redis运维

- 内存占用过大时，可以切分为多个实例，减少单个实例的内存占用，减少BgSave和重启时Load数据的时间
- 一致性要求不是非常高的业务，可以把自动的BgSave关闭，在凌晨或者空闲时候手动调用BgSave

CookieMapping

- 广告投放时是用Cookie来定向用户的
- CM保存了DSP与ADX（如百度、淘宝）间CookieId的映射关系
- 在RTB竞价的时候，DSP可以根据ADX-CookieId从映射表中获取到DSP-CookieId


Cookie Mapping
RTB
Publisher Web Page
<img src="http://[redacted]x.com/t.gif?
tanx_nid=1234&tanx_cm&xxx=xxx" />
3. echo 1x1
pixel
3. store
Mapping Table
Cookie Mapping
Server
Cookie Mapping
Table
6. Cookie Mapping
RTB Server
Network Control Area
1. cm request
7. Bid
Response
5. Bid Request
[redacted]x.com/t.gif
2. 302 Redirect
http://www.dsp.com/
tanx_tid=xxx&tanx_ver=1&xxx=xxx
Tanx ADX
Tanx Control Area
8. Echo
Creative
4. Publisher Ad
Call
Cookie Mapping Mesage
RTB Message
Publisher Web Page
<script src="http://[redacted]x.com/
ex?i=xxx..." />

CookieMapping存储

- 数据量大，单个渠道全量映射达到20亿以上Id
- 渠道多达几十个
- 每个竞价请求查一次
- 响应控制在1ms以内

CookieMapping 第一版

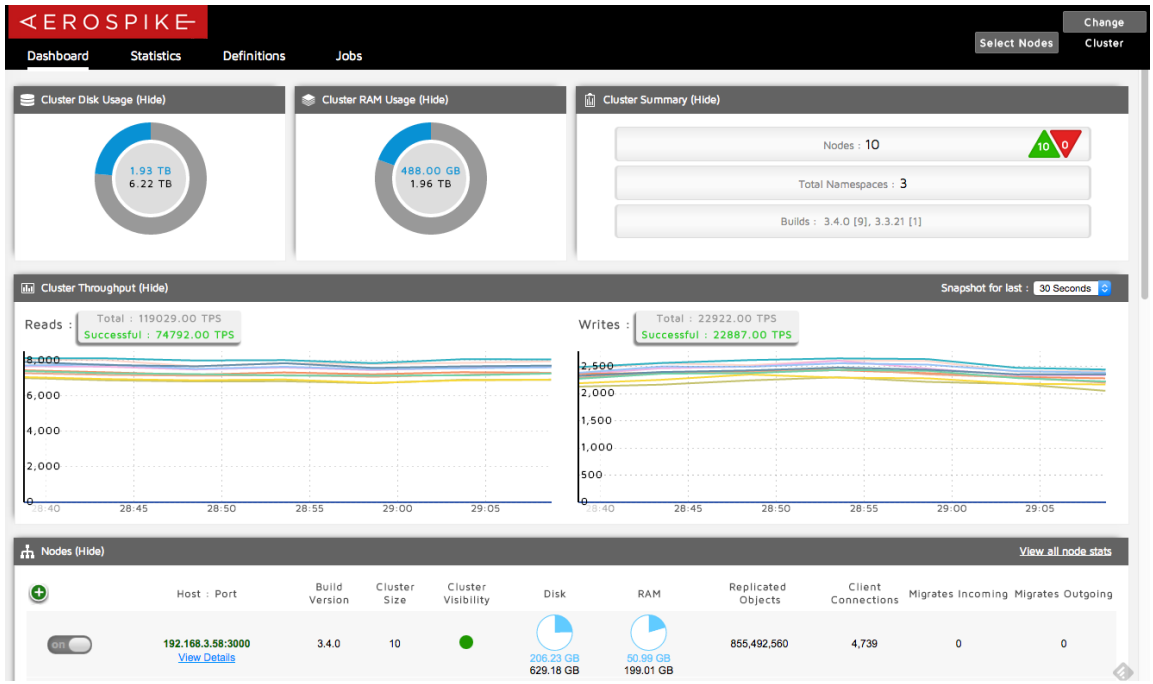
- 数据存在Redis中
- 占用内存大，达到2T内存
- 内存成本高
- Redis没有集群，维护成本高（嗯，当时是还没的）

CookieMapping: Aerospike

- 性能不比Redis差
- SSD优化
- 完备的分布式集群
- 二级索引
- 开源，企业版支持跨机房的集群
- 99%的请求1ms响应
- 支持的数据结构类型偏简单

CookieMapping: Aerospike

- 采用SSD来存储（Intel S3500，SATA口）
- 数据在SSD中，索引在内存中（1G内存索引16M记录）
- 10个节点，replication-factor: 1，写一份到Ardb做备份
- 官方提供Go的Client
- 线上半年无故障



CookieMapping: Aerospike

====writes_reply====

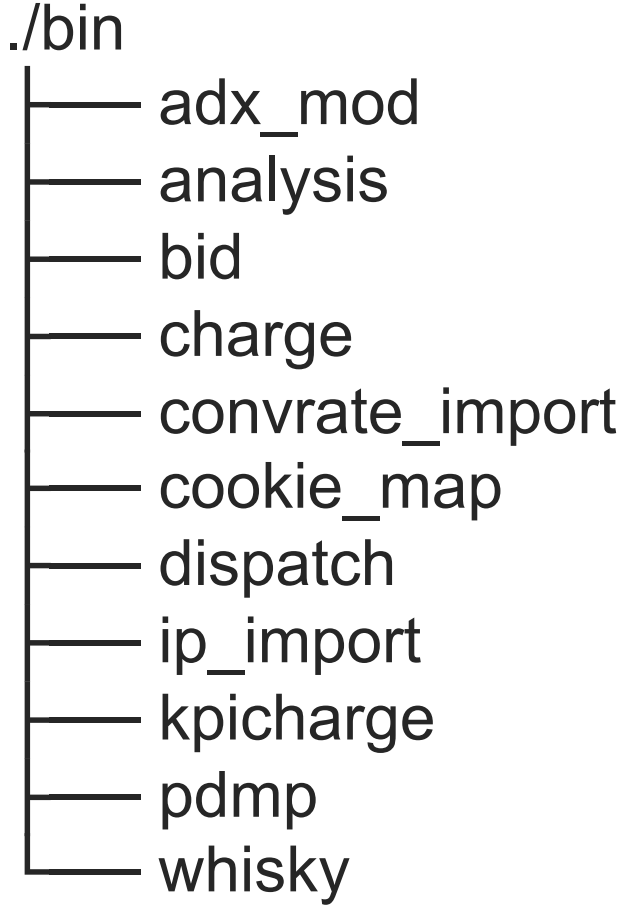
	timespan	ops/sec	>1ms	>8ms	>64ms
192.168.3.52:3000	10:25:12-GMT->10:25:22	2378.6	0.06	0.00	0.00
192.168.3.53:3000	10:25:04-GMT->10:25:14	2273.1	0.03	0.00	0.00
192.168.3.54:3000	10:25:13-GMT->10:25:23	2292.5	0.04	0.00	0.00
192.168.3.55:3000	10:25:09-GMT->10:25:19	2481.1	0.03	0.00	0.00
192.168.3.56:3000	10:25:12-GMT->10:25:22	2359.8	0.11	0.00	0.00

====reads====

	timespan	ops/sec	>1ms	>8ms	>64ms
192.168.3.52:3000	10:25:12-GMT->10:25:22	11974.2	0.18	0.00	0.00
192.168.3.53:3000	10:25:04-GMT->10:25:14	11741.3	0.06	0.00	0.00
192.168.3.54:3000	10:25:13-GMT->10:25:23	10975.6	0.12	0.00	0.00
192.168.3.55:3000	10:25:09-GMT->10:25:19	12223.2	0.09	0.00	0.00
192.168.3.60:3000	10:25:13-GMT->10:25:23	11266.2	0.10	0.00	0.00

把服务划分为独立的进程

- 独立部署
- 方便更新与重启



部署：进程管理

- Go不支持以后台进程的方式启动
- 使用Supervise来管理进程
- 配合parallel-ssh做集群管理

部署：web服务器

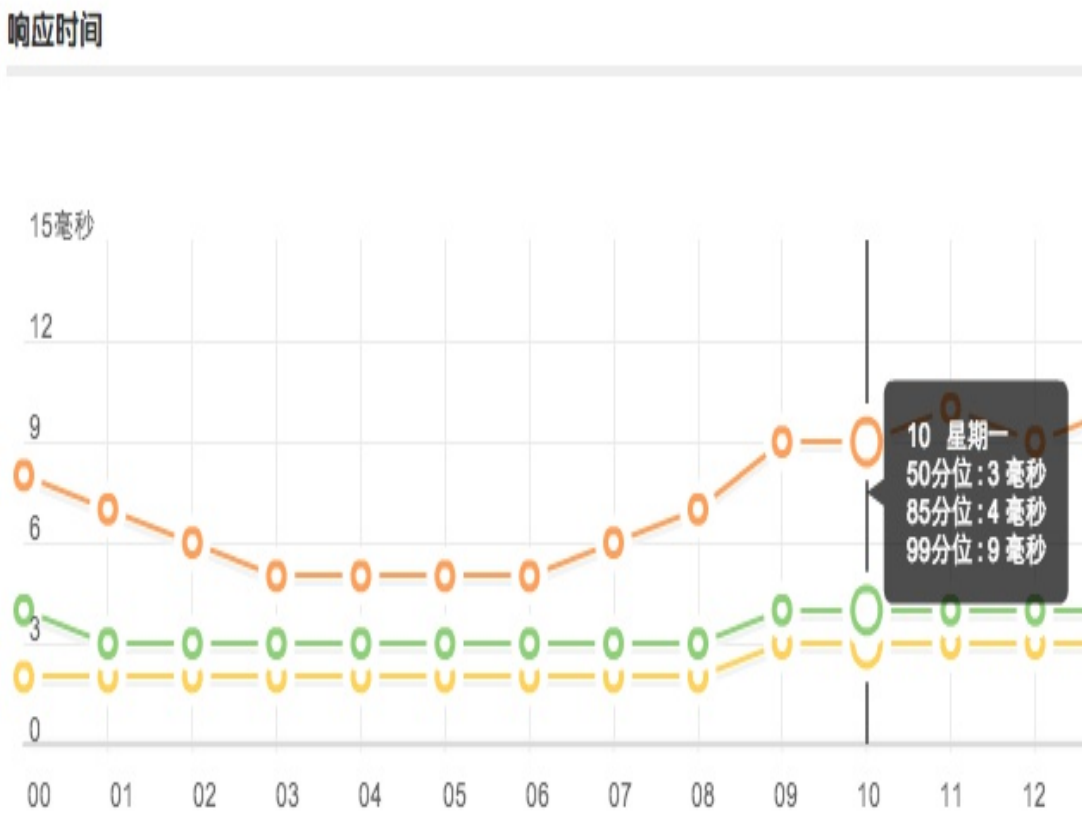
- Nginx做前端
- 开多个Go进程
- Nginx的upstream做负载均衡
- Nginx的upstream记得开keepalive

第一版正式上线

- 三个人（Golang部分）
- 三个多月

线上运行情况

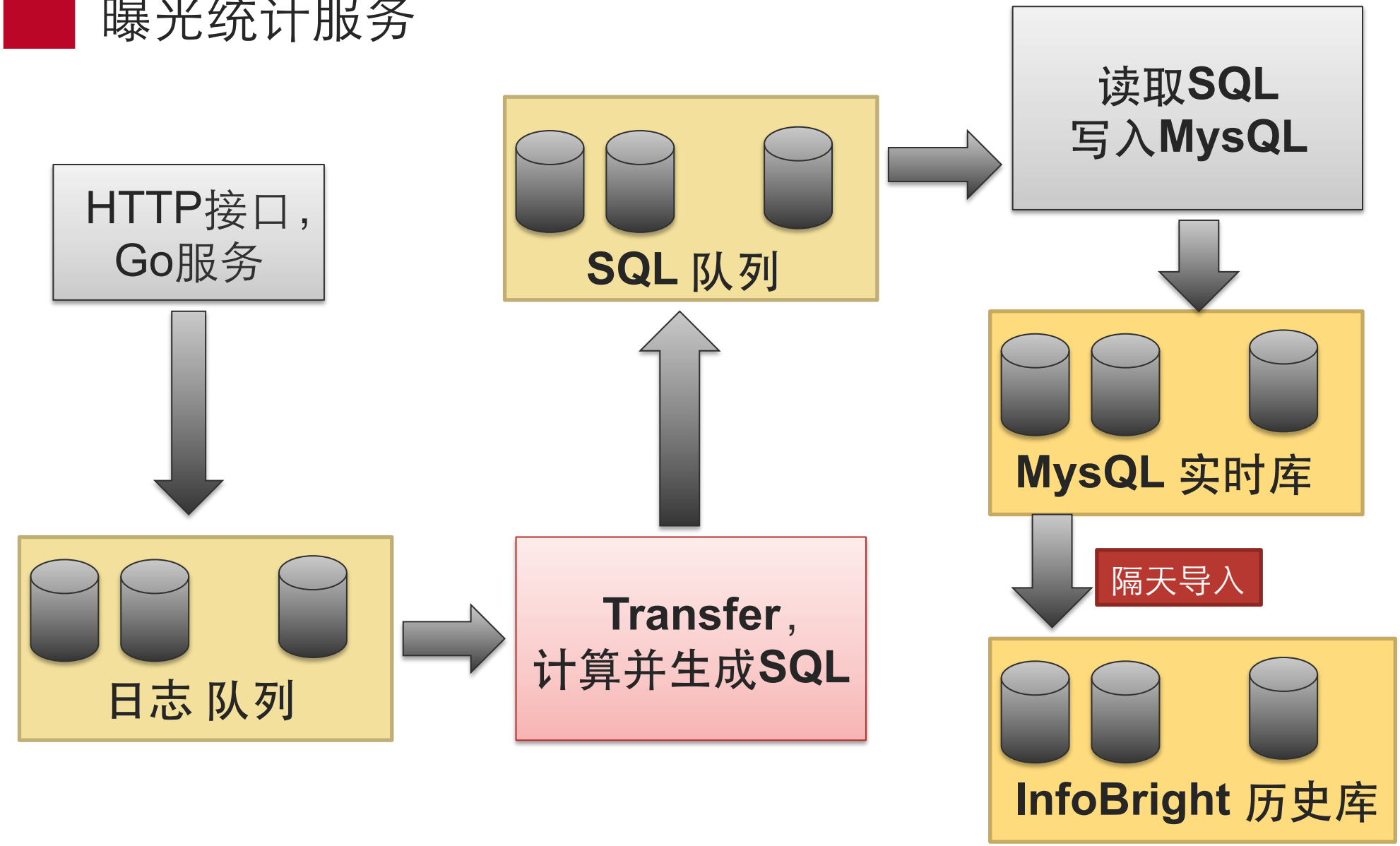
- 30台竞价服务器，CPU为8核16线程
- 每天100多亿竞价请求
- 峰值20万QPS
- 98%响应在10ms以内
- 稳定



曝光统计服务

- 每天N亿曝光
- 30个维度，40个指标
- 表的数据量和维度的离散程度相关
  - 如广告位有5000个，全国500个城市，时间粒度到小时级别，则地区汇总表一个推广活动一天最多就有 5000*500*24 = 6千万 记录
- 实时统计
  - 实时计算、入库
  - 实时查询，秒级响应

曝光统计服务



Golang重写原来PHP的Transfer

- 应届毕业生
- 一个人
- 三个星期
- 接近一万行代码
- 性能提升7倍
- 部署、维护更方便

曝光统计服务

- 写入做一些合并，减少写入的SQL数量
- MySQL库只保留最近7天的数据
- MySQL使用MyISAM引擎
- MySQL做分库、分表后还可以应付
- InfoBright是列存储
- InfoBright压缩率奇高
- InfoBright使用的是社区版

曝光统计服务： 问题

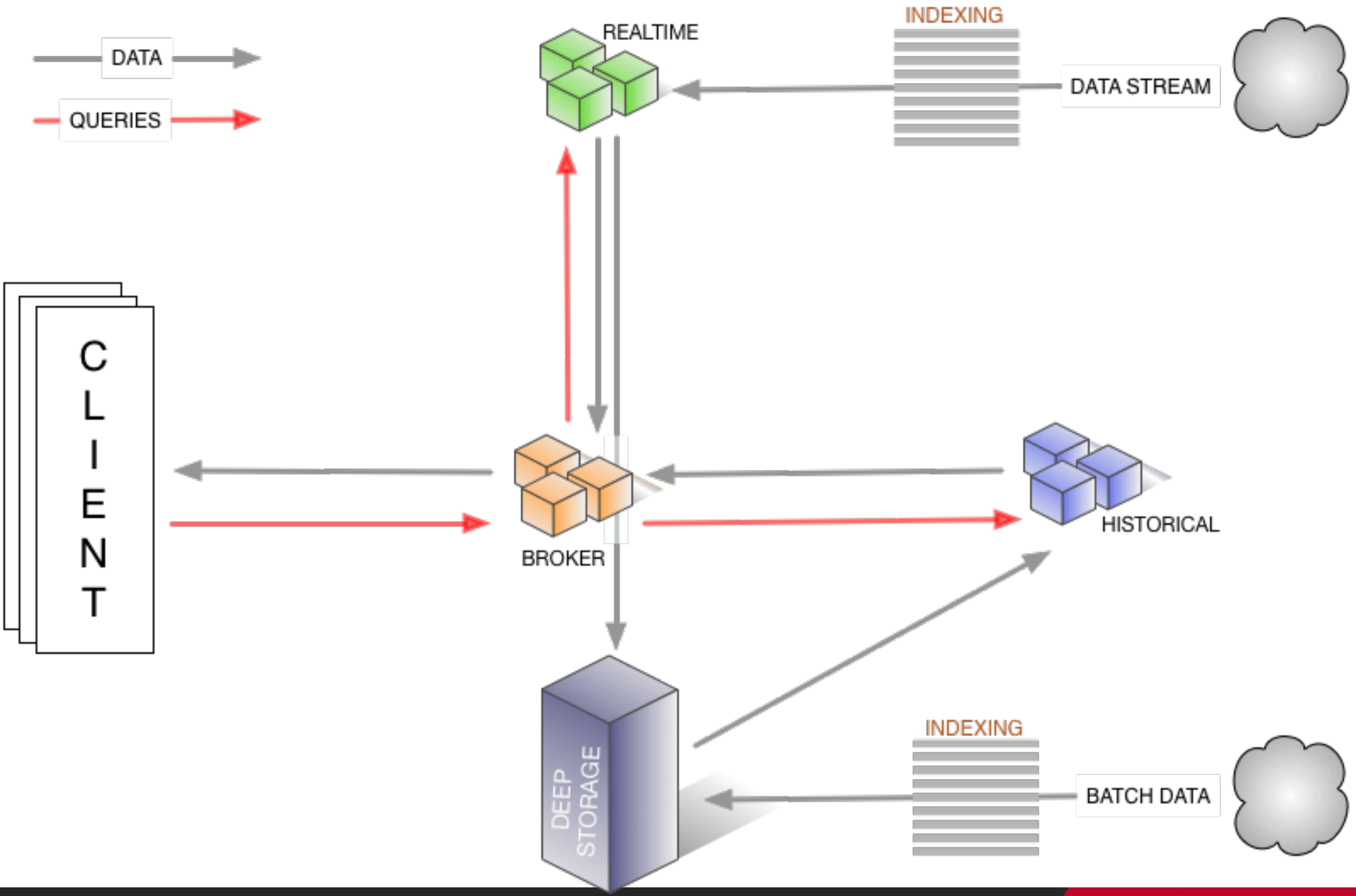
- MySQL 不适合OLAP类应用
- MySQL和InfoBright横向扩展都不方便
- 数据更新麻烦

实时统计存储

- Apache Drill
- HP Vertica
- EMC GreenPlum
- Cassandra
- InfiniDB
- MonetDB
- VoltDB
- InfluxDB
- MemSQL
- opentsdb
- kairosDB
- Kylin
- Druid
- …

曝光统计服务: Druid

- 专为统计分析而生
- 列存储
- shared-nothing的分布式架构，可扩展性、高可用
- 秒级以内对十亿行级别的表进行统计查询
- 对内存要求高，相当于内存数据库
- JAVA系，开源

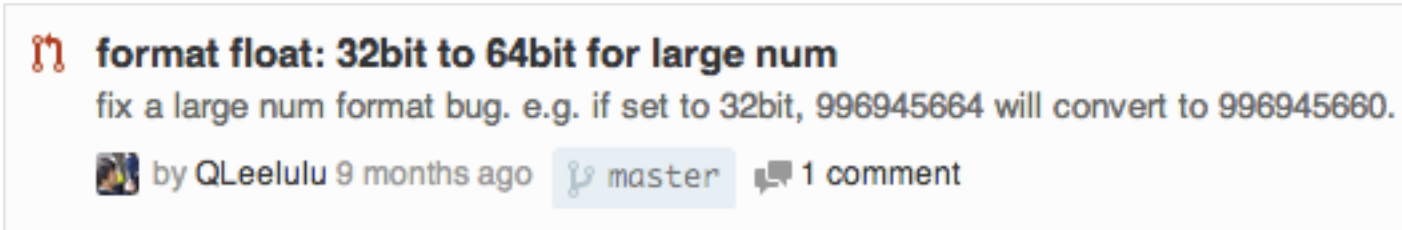


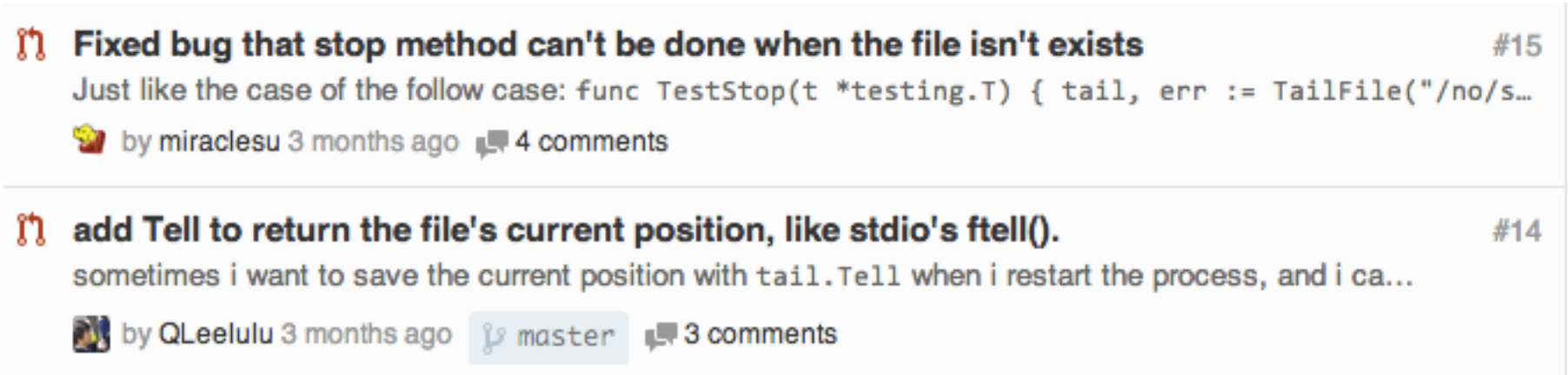
曝光统计服务: Druid

- 没有Go的Client，所以我们写了一个
- github.com/shunfei/godruid

回馈社区

- 第三方包会有一些小坑





Q&A

Thanks!

欢迎加入我们!

舜飞科技

网络运营全流程解决方案供应商