

# 亚信DMP平台Redis使用技术交流

亚信大数据事业群 姜明俊



AsiaInfo 亚信



## 1 Feature&Architecture

2 Persistence


3 Replication

4 Transaction

5 Server programme

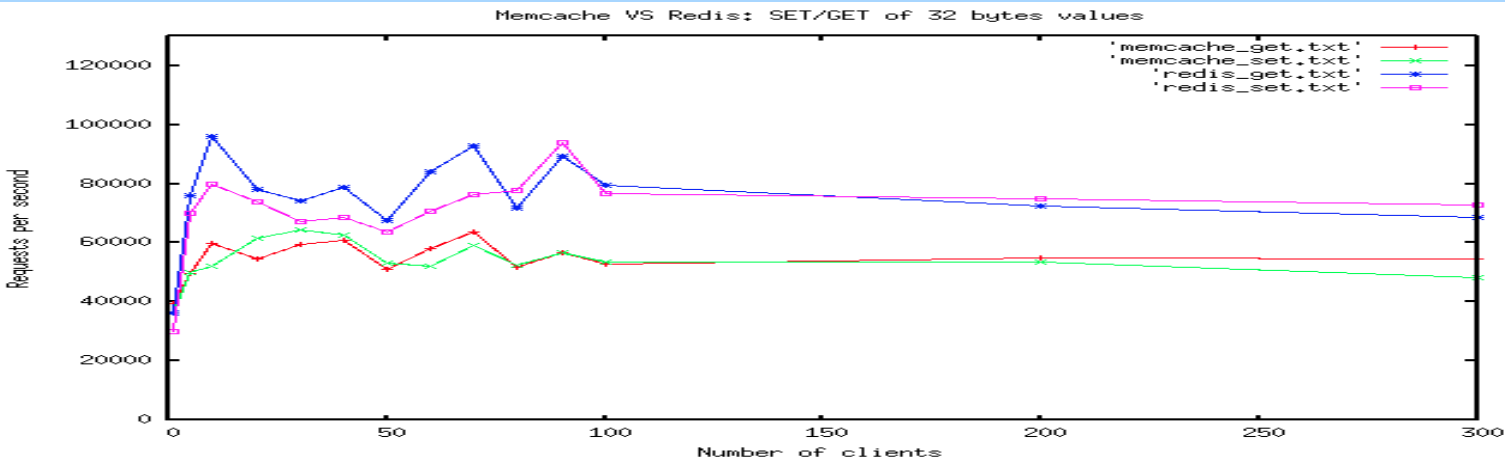
6 DMP platform

## Redis是什么？

- Redis is an open source, advanced **key-value memory store** , like memcachedb. It is often referred to as a **data structure server** since keys can contain strings, hashes, lists, sets and sorted sets.
- single threaded, no lock
- Event demultiplexer, no use libevent
- 2w lines written in c without 3<sup>rd</sup> library
- Faster
-  <http://redis.readthedocs.org/en/latest/>

# Feature&Architecture

	redis	memcached
data structure	string list hash set zset	string
thread model	third(main thread, two auxiliary thread)	many
transaction	yes	no
persistence	yes	no
replication	yes	no
server extension	yes	no

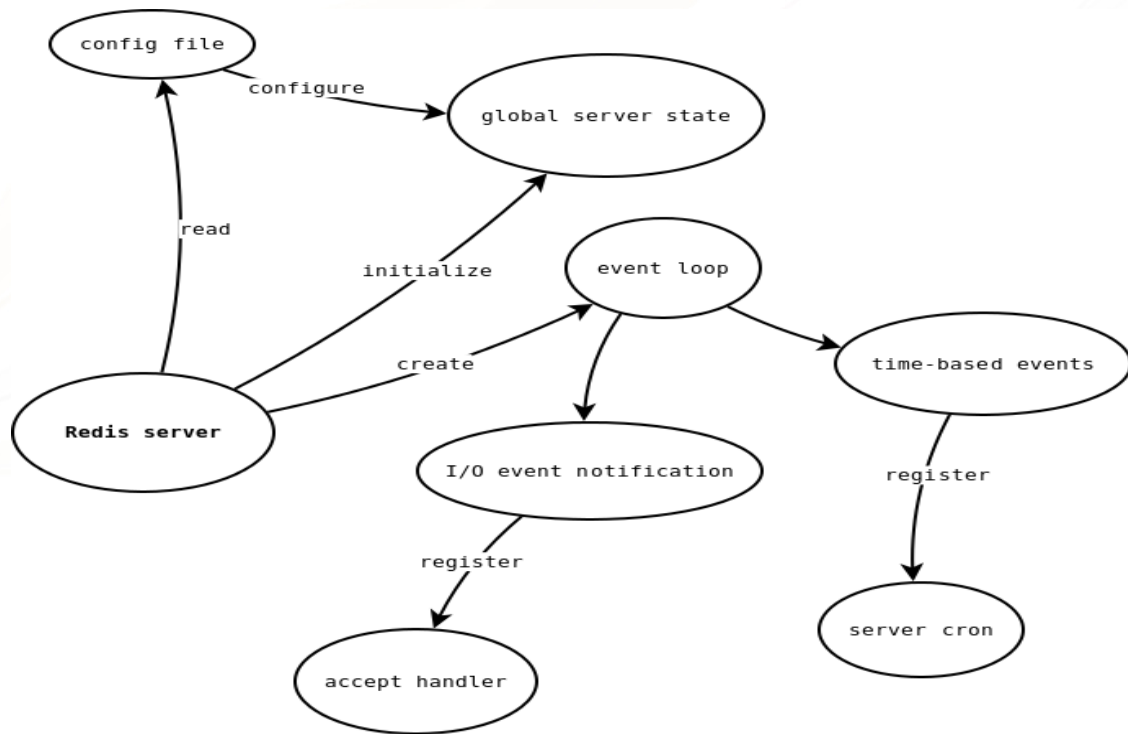


## Redis的常用操作

- get/set/mset/mget
- lpush/lpop, lpush/rpop
- hget/hset/hgetall
- sadd/srem/spop/scard/sort
- del key/move key index
- infor
- save/bgsave/bgrewriteaof/lastsave
- flushall

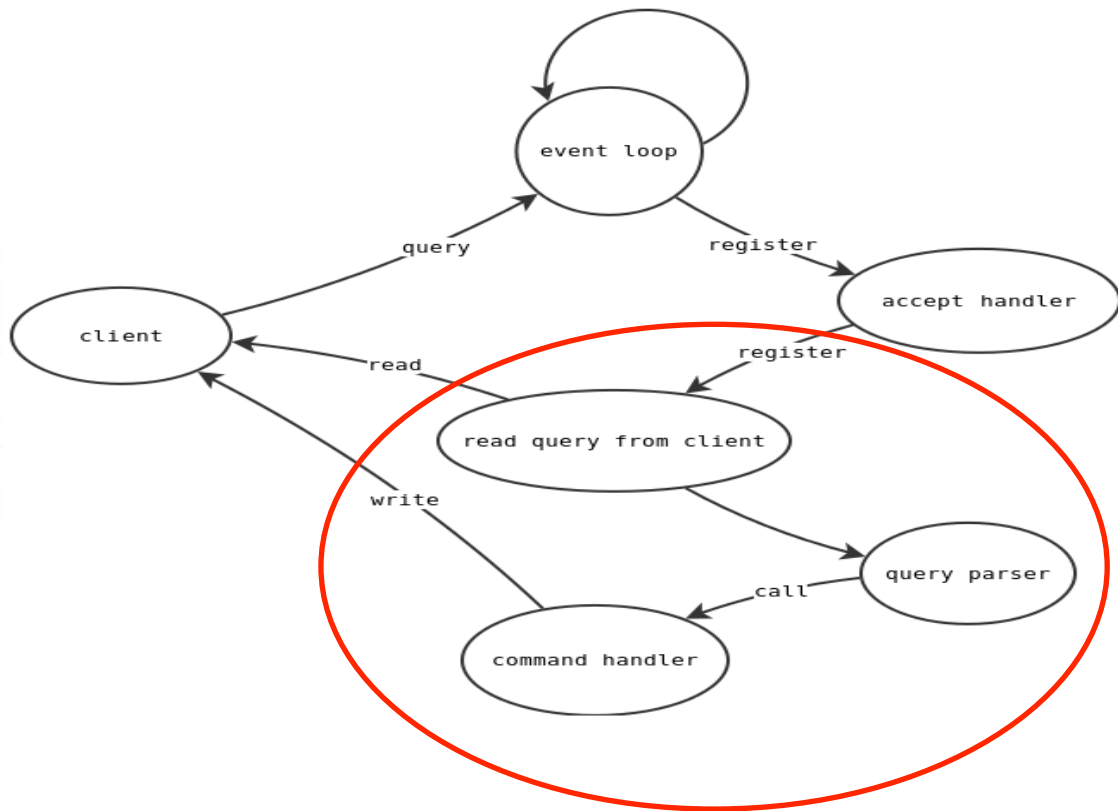
## Redis server的主流程

- ◆ 开始进行全局服务端状态初始
- ◆ 设立命令表，二分查找
- ◆ 导入配置文件redis.conf
- ◆ initServer(),初始化server 结构体,设置信号处理, 创建TCP socket, 打开持久化
- ◆ Event loop轮询监听IO事件和时间事件, 同时为客户端服务, 无阻塞,
- ◆ Server cron 周期性工作, rehash, 持久化, LRU 清理无用连接
- ◆ 载入AOF或dump文件, 重建服务端数据
- ◆ Accept handler 响应客户端请求

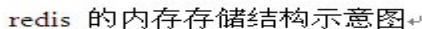


## 处理命令和返回应答的方式

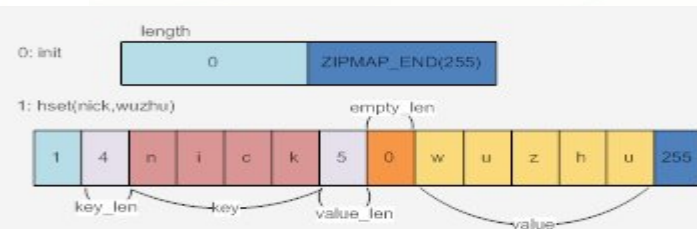
- 接受连接创建客户端对象，并添加到全局客户端链表中，注册一个handler来响应客户端的输入请求
- 读取客户端命令并解析，通过调用call来真正执行，对于写命令，跟踪有多少键已经在某个时期被改变了，或者写入到AOF文件
- 返回客户端，并重置客户端对象



## Redis在value上的不同设计之处



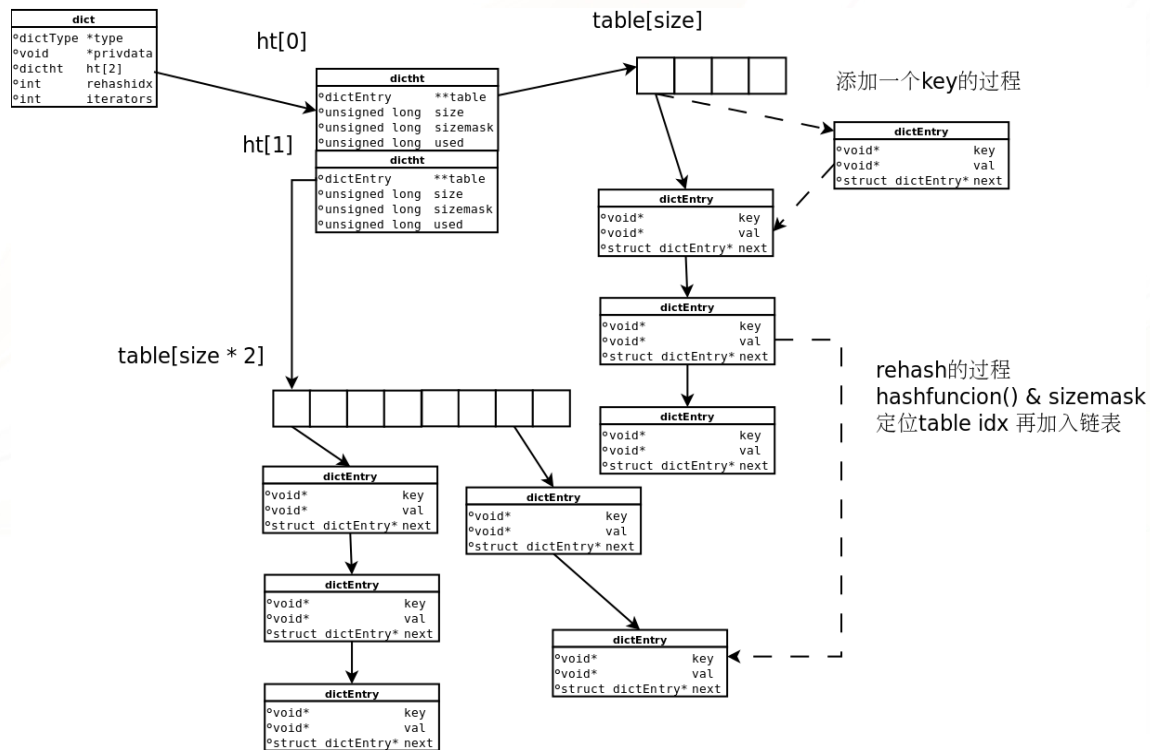
- ◆ Redis主体结构就是实现一个hash table
- ◆ Redis 是支持多数据库(表)的,用RedisDb表示一个数据库(表), 默认情况下, 有16个不同的数据库, 这个可以被用来作为redis server 的命名空间, 不同的库,key可以重复
- ◆ struct dictht 由一个 struct dictEntry 指针数组组成, 链表法解决冲突
- ◆ dict 由 struct dictht 的 哈希表构成, 定义成长度为2, 防止一次性 rehash 期间 redis 服务能力大幅下降
- ◆ Key的类型为sds
- ◆ Value的类型为redisobject
- ◆ HASH存储为例: redisObject的type成员值是 REDIS\_HASH 类型, 当该hash 的 entry 小于配置值: hash-max-zipmap-entries 或者value字符串的长度小于 hash-max-zipmap-value, 编码成 REDIS\_ENCODING\_ZIPMAP 类型存储, 以节约内存. 否则采用 Dict 来存储.





## Rehash

- ◆ 每个db对应两条hash table,大多数情况下只用第一条hash table,第二条在增量hash时会使用,增量hash采用阶段性完成,单次拷贝不能超过1ms,以免影响前台应用过多响应时间。
- ◆ bucket初始大小为4,以2的倍数进行动态扩展。
- ◆ 作增量rehash的过程中,新的值将会写到第二条hash table里。



## 适用场景

- 轻量级的高性能消息队列服务,生产者消费者
- 跨机器的共享内存
- Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，并且它没有原生的可扩展机制，不具有scale（可扩展）能力，要依赖客户端来实现分布式读写，分布式集群版本还没有正式release，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。



**1 Feature&Architecture**

**2 Persistence**

**3 Replication**

**4 Transaction**

**5 Server programme**

**6 DMP platform**

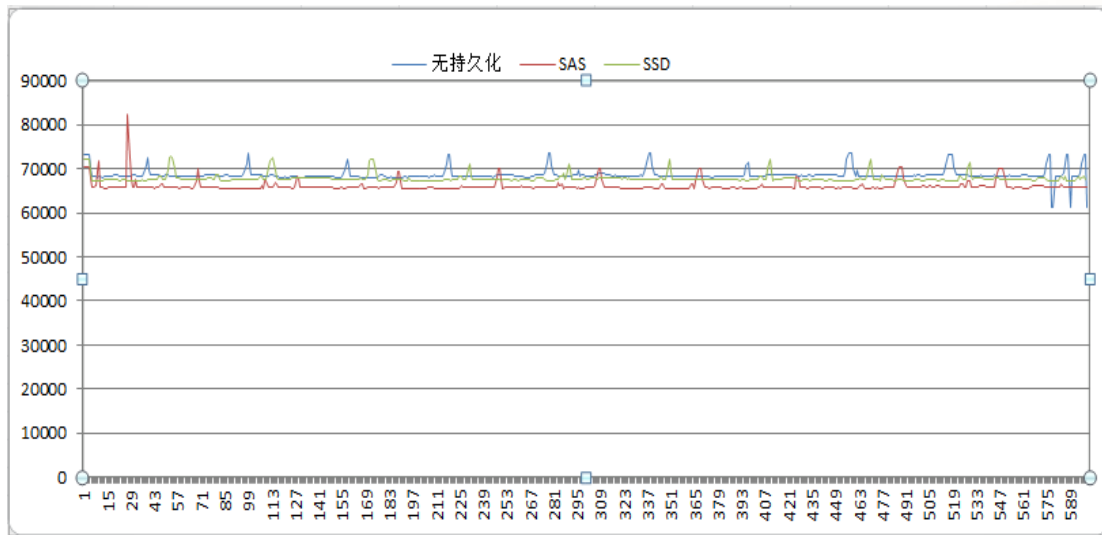
## RDB持久化

- Fork一个进程，利用copy on write原理，遍历所有db的hash table，进行整库的dump
- Save命令,shutdown命令，slave启动都会触发
- 利用LZF进行压缩
- 持久化触发条件：
  - #save 900 1
  - #save 300 10
  - #save 60 10000

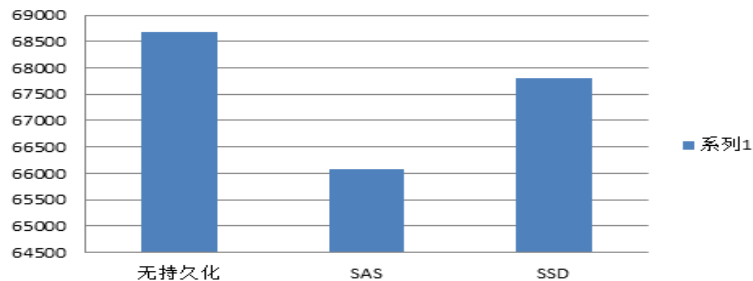
## RDB持久化

### 读写混合场景性能对比

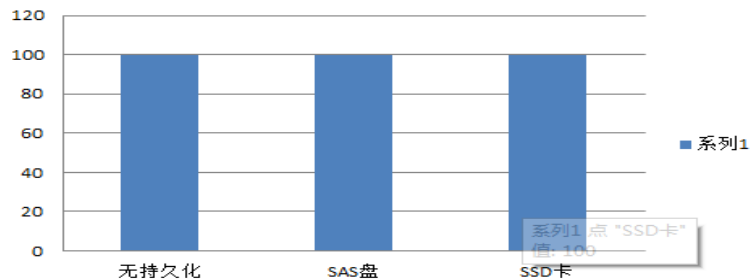
此模式下，模拟业务场景，采用java程序在后台往redis不停的写入数据，然后测试不同的持久化策略下，get的性能数据。



### Benchmark测试Get数据



### 10毫秒以内响应百分比



## AOF持久化

- AOF:把写操作指令连续的写到一个文件里面
- 当redis server异常crash掉的时候，重启时将会进行如下的操作：
  - 假如只配置了aof，起动时加载aof文件
  - 假如同时配置了rdb,aof,起动时只加载aof文件
  - 假如只配置了rdb,起动时将加载dump文件

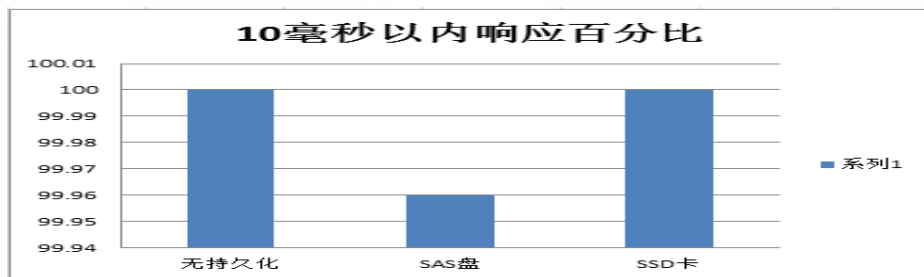
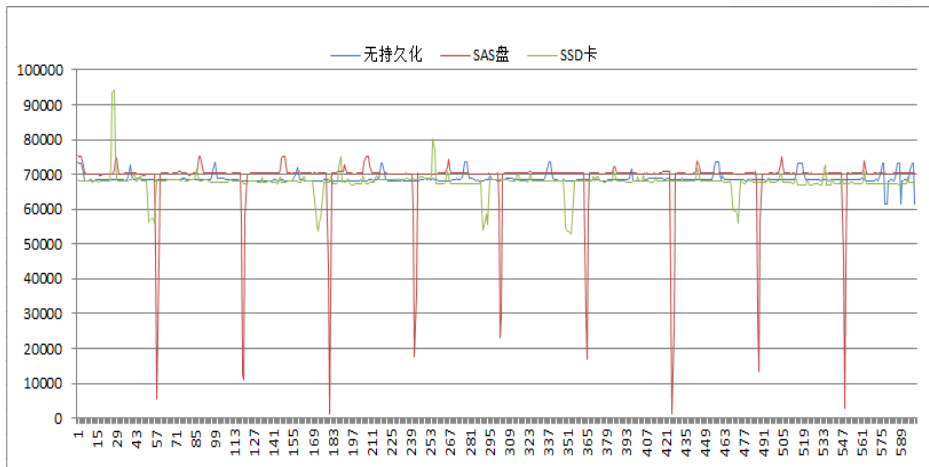
# Persistence

## AOF持久化

该模式下，redis对于数据安全性提供最大程度的保障，但对应也会损失较多的性能。在普通的SAS盘情况下，set操作每秒只能处理1187条记录，10毫秒以下响应百分比为0，最低延迟都在18毫秒以上。这种性能损失和高延迟的情况在SSD卡上有明显改善，set操作每秒处理40118条记录，10毫秒以内响应百分比为99.79%。

### 读写混合场景性能对比

此模式下，模拟业务场景，采用java程序在后台往redis不停的写入数据，然后测试不同的持久化策略下，get的性能数据。



## 耗时测试

RDB是Redis默认的持久化方式，采用快照的方式将内存数据dump到存储设备。在大数据量的情况下，这种方式的dump文件要比AOF的日志文件节省空间，对应的加载耗时也会比AOF持久化模式更有优势。但是在我们的测试过程中，由于只测试了10G的情况，在低数据量的场景下，AOF日志有重构机制，会比RDB模式占优势。

列1	10G数据初始化耗时（秒）	10G数据关闭耗时（秒）
无持久化	1000	0
RDB（SAS）	48	95
RDB（SSD）	60	129
AOF（SAS）	31	0
AOF（SSD）	27	0



# Persistence

## 持久化的缺陷：

1. Redis使用buffer IO的方式，而不是Direct IO，一方面写性能低下，而且在内存紧张的情况下，还会导致redis不稳定（高延迟）甚至奔溃。
2. 如果采用rdb方式做持久化，会丢失数据。因为rdb的save机制，是采取在多少时间内如果发生多少命令的操作，才执行持久化
3. 如果使用aof（append only file）的持久化方式，会导致redis读写变慢  
具体的原因是跟aof提供的3个保存模式相关

对于三种AOF 保存模式，它们对服务器主进程的阻塞情况如下：

1. 不保存（AOF\_FSYNC\_NO）：写入和保存都由主进程执行，两个操作都会阻塞主进程。
2. 每一秒钟保存一次（AOF\_FSYNC\_EVERYSEC）：写入操作由主进程执行，阻塞主进程。保存操作由子线程执行，不直接阻塞主进程，但保存操作完成的快慢会影响写入操作的阻塞时长。
3. 每执行一个命令保存一次（AOF\_FSYNC\_ALWAYS）：和模式1 一样。

因为阻塞操作会让Redis 主进程无法持续处理请求，所以一般说来，阻塞操作执行得越少、完成得越快，Redis 的性能就越好。

能不持久化尽量不要持久化，通过主备高可用方式来解决



**1 Feature&Architecture**

**2 Persistence**

**3 Replication**

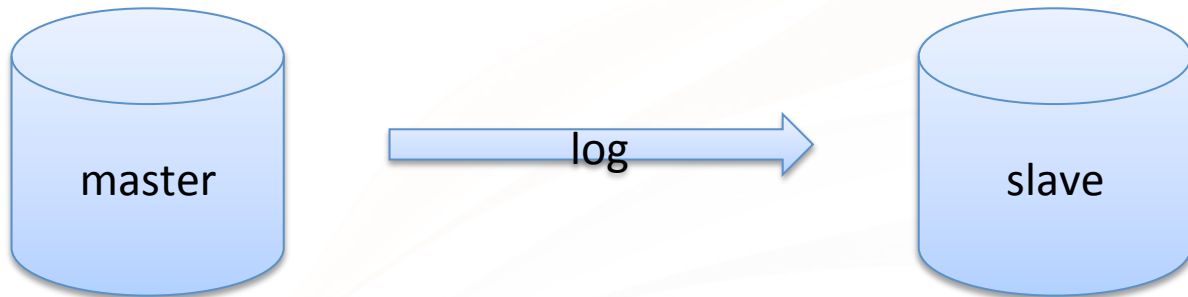
**4 Transaction**

**5 Server programme**

**6 DMP platform**

## Replication

相当于mysql statement模式的SQL复制



优势:

Slave也可以进行读写操作

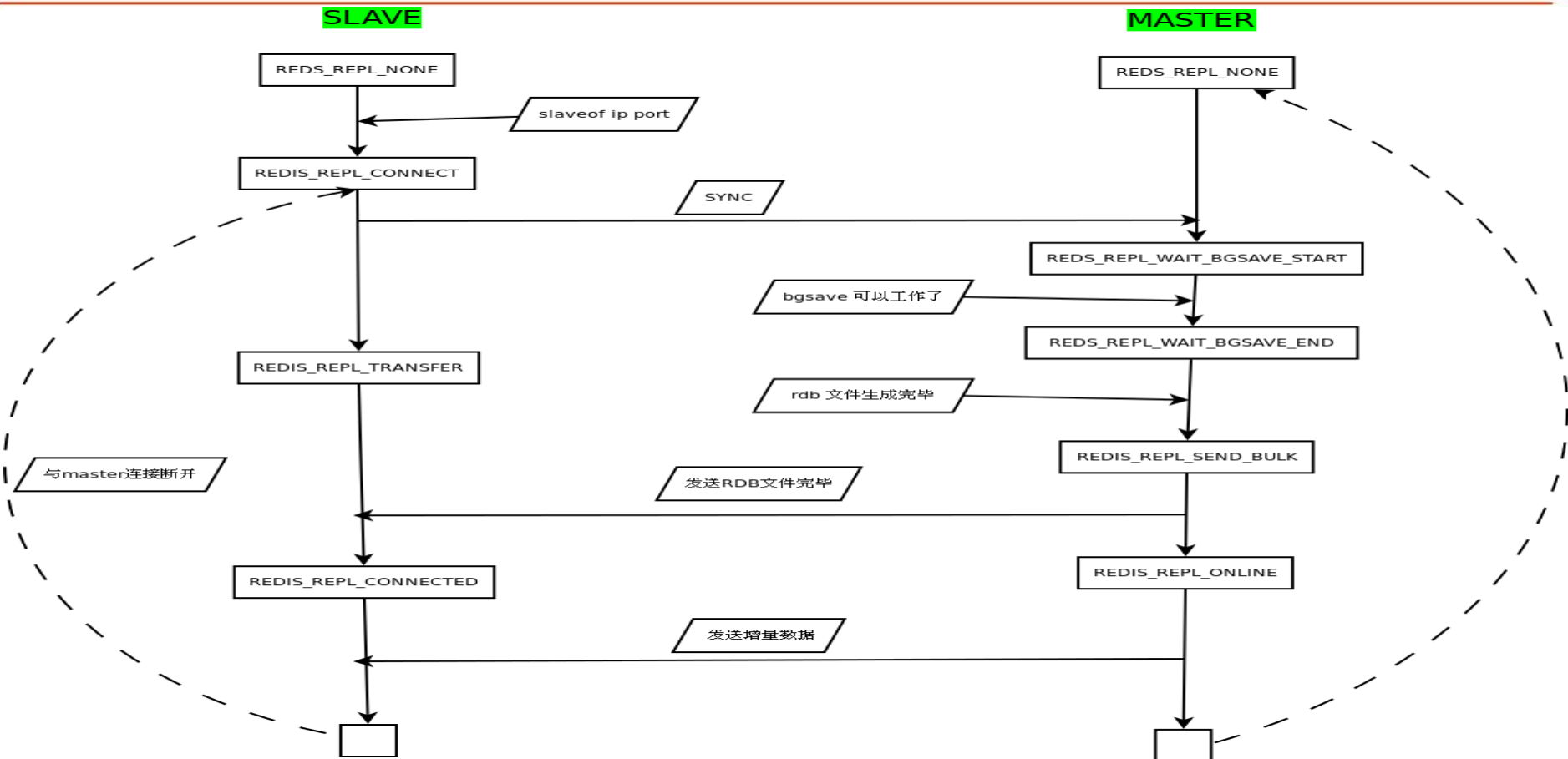
可以级联复制

高并发下主备同步延迟可忽略

不足:

Redis的复制是基于内存快照文件的，没有类似mysql的主从复制的复制位的概念，master会每次dump出全内存的快照文件发送给slave，slave收到快照文件后，开始重建内存数据。如果快照文件很大，那整个过程是非常慢的。如果网络再不稳定，那更是雪上加霜。

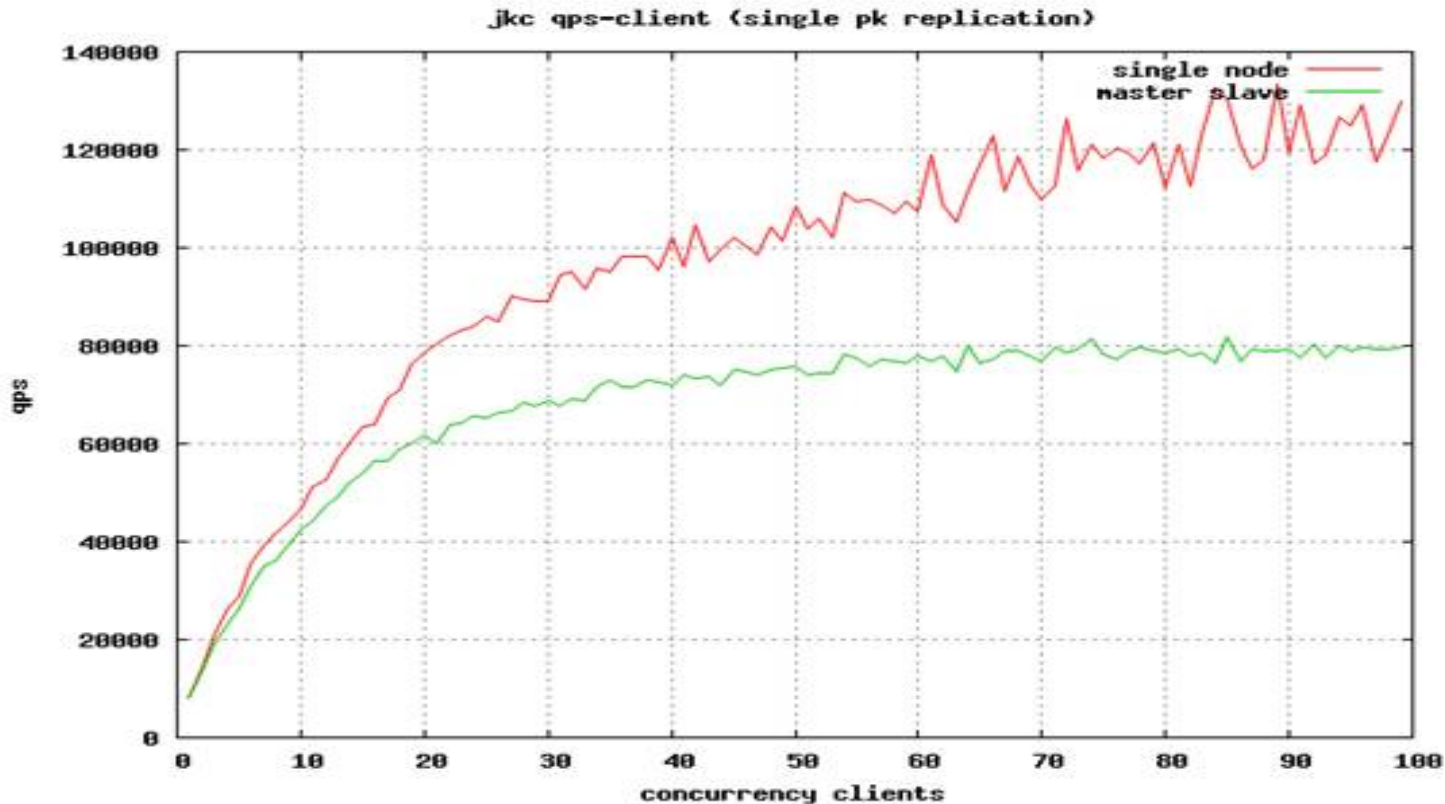
# Replication



# Replication

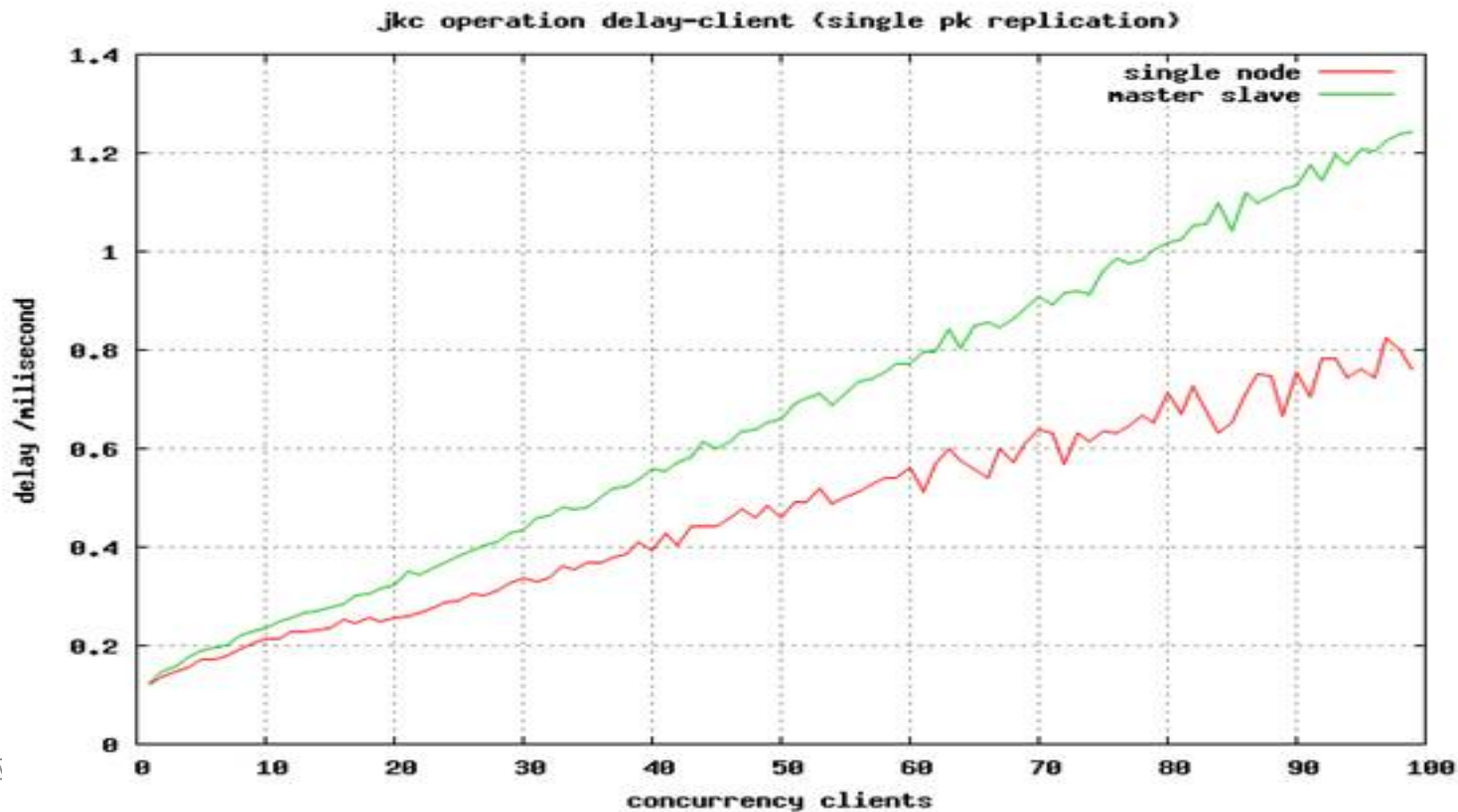
## 复制对TPS的影响(-30%)

我自己写  
了一个jkc  
命令



# Replication

## 复制时的jkc指令响应时间





- 1 Feature&Architecture
- 2 Persistence
- 3 Replication
- 4 Transaction
- 5 Server programme
- 6 DMP platform

# Transaction

**WATCH** 监视一个(或多个) key , 如果在事务执行之前这个(或这些) key 被其他命令所改动, 那么事务将被打断;

**UNWATCH** 取消 WATCH 命令对所有 key 的监视;

**MULTI** 标记一个事务块的开始, 指事务块内的多条命令会按照先后顺序被放进一个队列当中, 最后由EXEC命令原子性(atomic)地执行;

**DISCARD** 取消事务, 放弃执行事务块内的所有命令;

**EXEC** 执行所有事务块内的命令;

MULTI命令使用示例:

```
1 redis> MULTI
2 OK
3 redis> INCR user_id
4 QUEUED
5 redis> INCR user_id
6 QUEUED
7 redis> INCR user_id
8 QUEUED
9 redis> PING
10 QUEUED
11 redis> EXEC
12 1) (integer) 1
13 2) (integer) 2
14 3) (integer) 3
15 4) PONG
```

- ◆ 批量操作在发送 EXEC 命令前被放入队列缓存
- ◆ 收到 EXEC 命令后进入事务执行, 事务中任意命令执行失败, 其余的命令依然被执行
- ◆ 在事务执行过程, 其他客户端提交的命令请求不会插入到事务执行命令序列中



# Transaction

## ◆ 原子性(Atomicity)

原子意味着操作的不可再分，要么执行要么不执行。Redis 本身提供的所有 API 都是原子操作，那么 Redis 事务其实是要保证批量操作的原子性。Redis 实现批量操作的原理是在一个事务上下文中（通过 MULTI命令开启），所有提交的操作请求都先被放入队列中缓存，在 EXEC 命令提交时一次性批量执行。这样保证了批量操作的一次性执行过程，但 Redis 在事务执行过程的错误情况做出了权衡取舍，那就是放弃了回滚。

Redis 官方文档对此给出的解释是：

- 1.Redis 操作失败的原因只可能是语法错误或者错误的数据库类型操作，这些都是在开发层面能发现的问题不会进入到生产环境，因此不需要回滚。
- 2.Redis 内部设计推崇简单和高性能，因此不需要回滚能力。

## ◆ 一致性(Consistency)

一致性意味着事务结束后系统的数据依然保证一致。在事务开始之前，数据保持有效的状态，事务结束后也如此。显然在前面讨论原子性时，Redis 舍弃了回滚的设计，基本上也就舍弃对数据一致性的有效保证。不过对于一个高效的 key-value store 或 data structure server，数据操作一致性很多时候更多应该依赖应用层面，**事实也是我们使用 Redis 时很多时候都是分片和集群的，数据一致性无法依靠任何事务机制。**

## ◆ 隔离性(Isolation)

隔离性保证了在事务完成之前，该事务外部不能看到事务里的数据改变。也不能访问一些中间状态，因为假如事务终止的话，这些状态将永远不会发生。Redis 采用单线程设计，在一个事务完成之前，其他客户端提交的各种操作都无法执行因此自然没法看见事务执行的中间状态，隔离性得到保证。

## ◆ 持久性(Durability)

Redis 一般情况下都只进行内存计算和操作，持久性无法保证。但 Redis 也提供了2种数据持久化模式，SNAPSHOT 和 AOF，SNAPSHOT的持久化操作与命令操作是不同步的，无法保证事务的持久性。而AOF模式意味着每条命令的执行都需要进行系统调用操作磁盘写入文件，可以保证持久性，但会大大降低 Redis 的访问性能。

Redis 提供的事务机制与传统数据库事务（ACID）有些不同，Redis不支持事务回滚，放弃了原子性，和一致性的保证，原子性和一致性需要应用进行保证



**1 Feature&Architecture**

**2 Persistence**

**3 Replication**

**4 Transaction**

**5 Server programme**

**6 DMP platform**

# Server programme

## ■ Lua

- 客户端使用Lua脚本，直接在服务器端原子的执行多个Redis命令
- 降低网络开销，提高并发和降低响应延时，著名场景：抢红包
- 利于服务扩展
- Redis使用脚本字典来保存所有被EVAL命令执行过的脚本，这些脚本可用于实现SCRIPT EXISTS命令
- SCRIPT FLUSH命令会清空服务器字典中保存的脚本，并重置Lua环境
- 服务器在执行脚本之前，会为lua环境设置一个超时处理的钩子，当出现超时运行时，可通过执行SCRIPT KILL命令来让钩子停止正在运行的脚本

```
[root@localhost src]# ./redis-cli
127.0.0.1:6379> set 12345 23
OK
127.0.0.1:6379> set 23 "abc|123|ddd"
OK
127.0.0.1:6379> eval "${cat test.lua}" 0 12345
(error) ERR Error compiling script (new function): user_script:1: unexpected symbol near '$'
127.0.0.1:6379> eval "${cat test.lua}" 0 12345
(error) ERR Error compiling script (new function): user_script:1: unexpected symbol near '$'
127.0.0.1:6379> quit
[root@localhost src]# ./redis-cli eval "${cat test.lua}" 0 12345
"abc|123|ddd"
```

```
1 link_id = redis.call("get", ARGV[1])
1 label = redis.call("get", link_id)
return label
```

| "test.lua" 6L, 100C



**1 Feature&Architecture**

**2 Persistence**

**3 Replication**

**4 Transaction**

**5 Server programme**

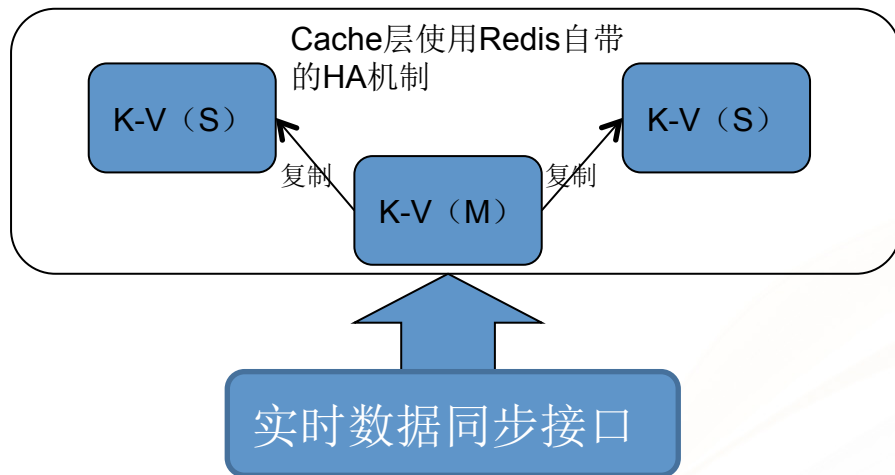
**6 DMP platform**

- ◆ 架构演进
- ◆ 存储优化
- ◆ 访问优化
- ◆ 分布式代理集群优化



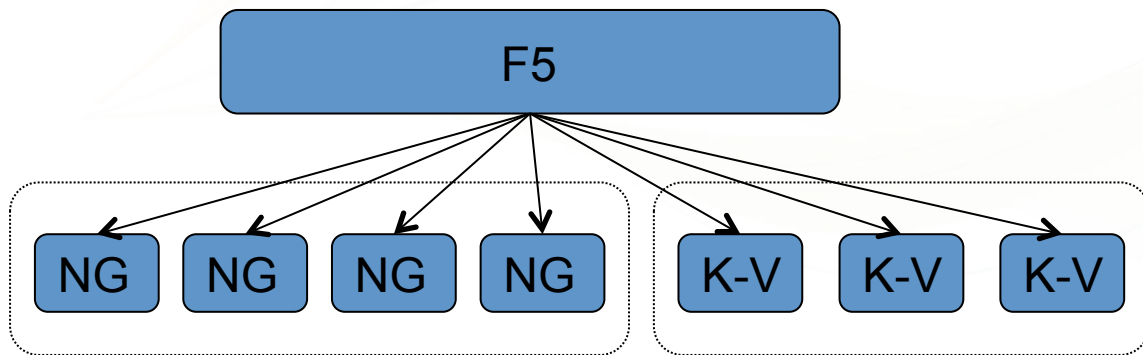
10w request/秒  
每个响应延时  
<10ms

# DMP Architecture V0



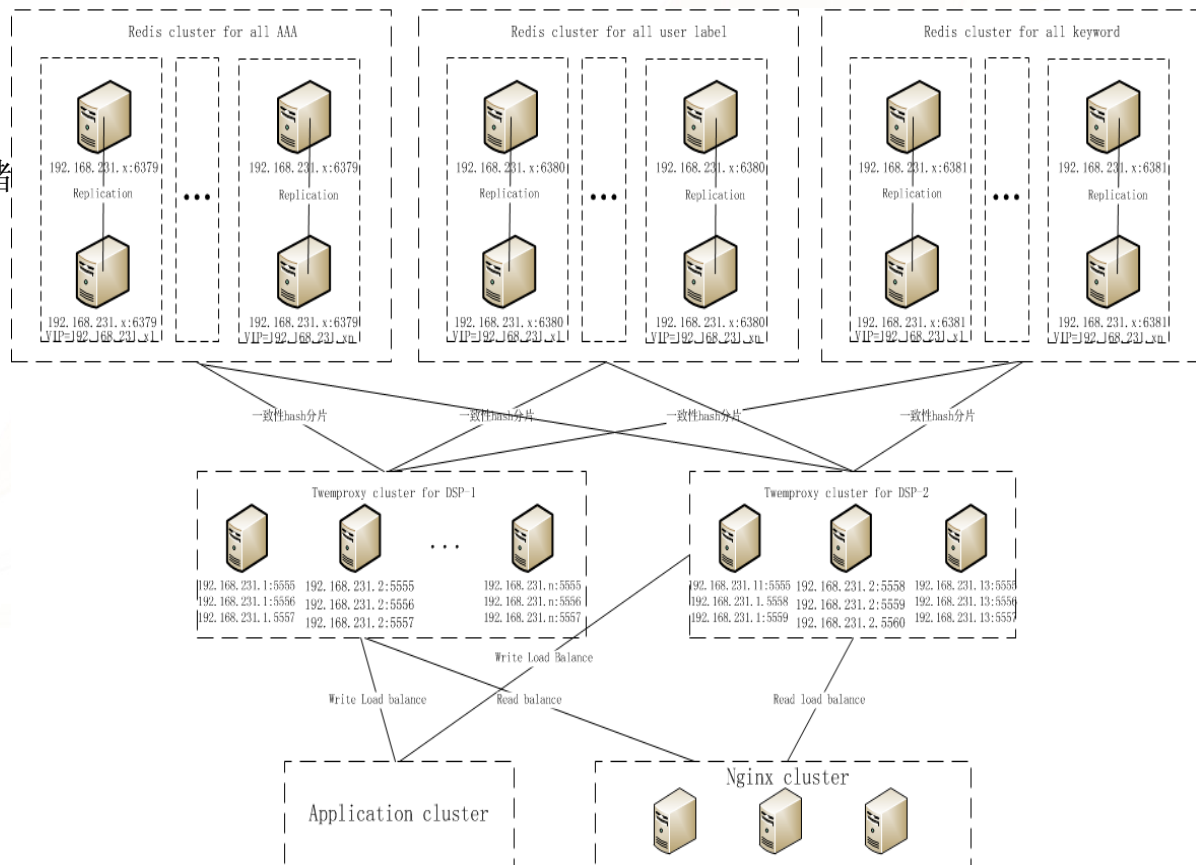
## 容量计算：

- 按类型ID方式存储1000w数据占用3G内存，3500w用户需要10G，再分3个接口(关键字，用户标签，域名)对外提供服务，每个接口需要对应的一张K-V映射表，这样总的容量估算应该是30G。目前单台机器是64G内存。可以3台K-V中都存一样的数据，提高读并发操作和高可以性
- 每个接口对应一个Redis实例

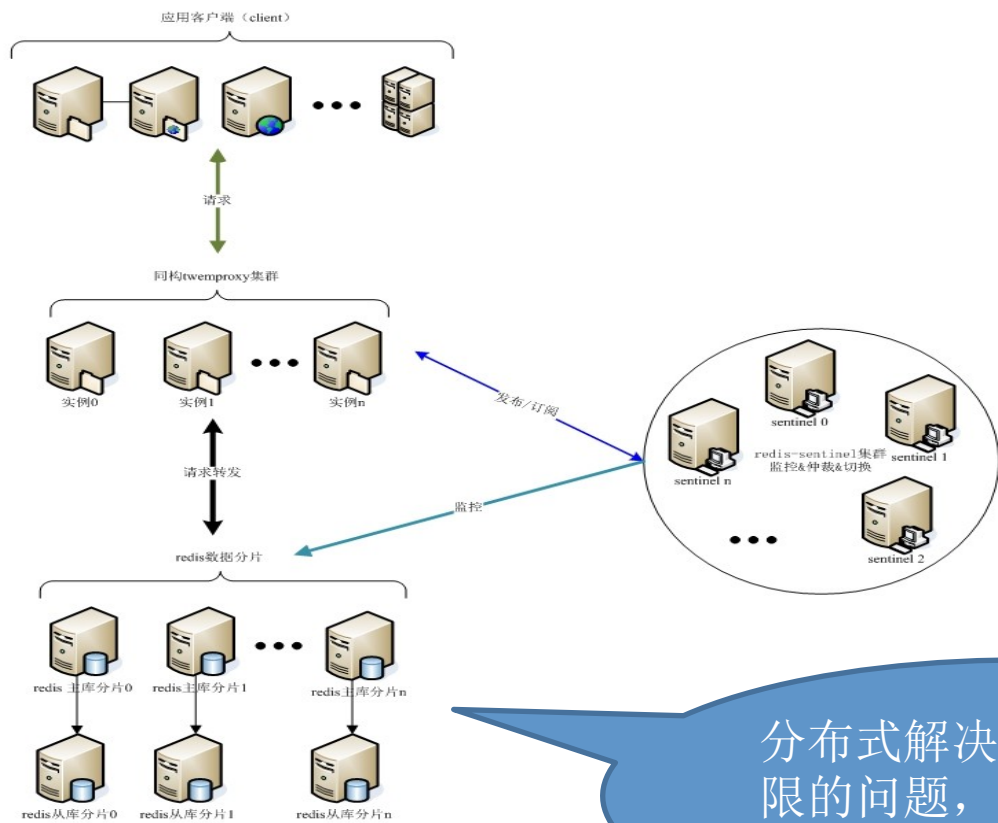


# DMP Architecture V1

实际的平台部署图  
统一的Redis集群，不同的实例不同的存储  
统一的代理，不同的服务



# DMP Architecture V1



- ◆ Nginx 提供http访问和对 twemproxy反向代理和负载均衡
- ◆ twemproxy: redis 的代理系统, 可以选择多种数据分片算法
- ◆ redis: 集群的 redis 存储节点
- ◆ sentinel: redis 官方的集群高可用组件, 可以监控 redis 主节点故障, 并进行主备切换

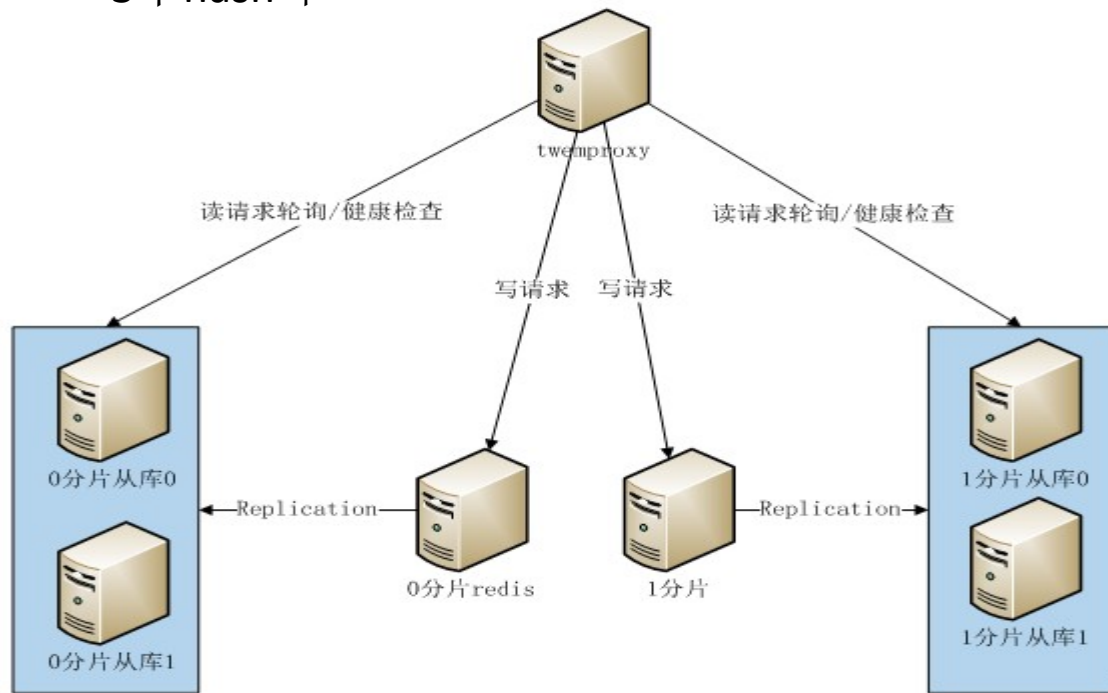
分布式解决了平台内存受限的问题, 如何还能保持高并发和低延时?





# DMP Architecture V1

◆ 优化手段：一主两从，除了提供高可用，还需要发挥读写分离的服务能力，组成3个hash环



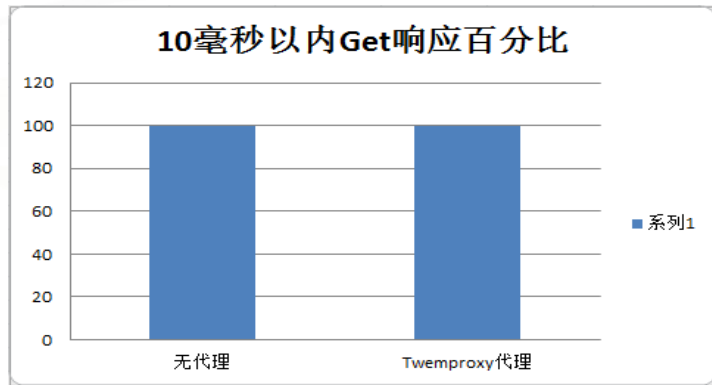
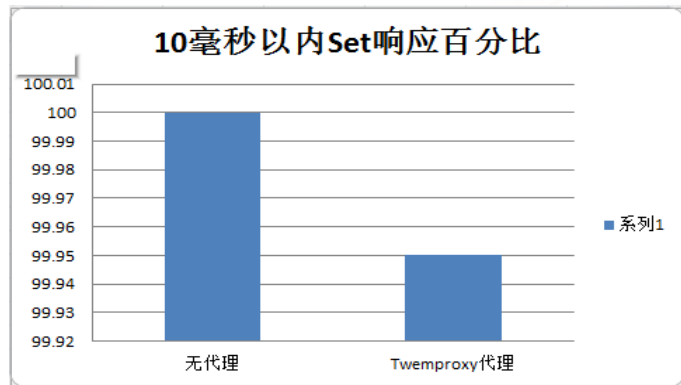
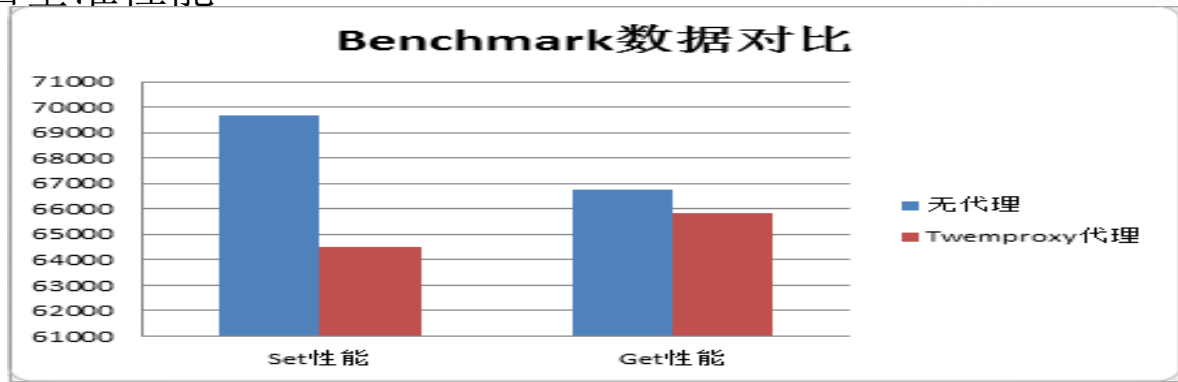
```
write:
listen: 172.16.13.19:26380
hash: fnv1a_64
distribution: ketama
auto_eject_hosts: true
redis: true
server_retry_timeout: 2000
server_failure_limit: 1
servers:
- 172.16.13.17:6380:1 server1 #master of 6380
- 172.16.13.18:6381:1 server2 #master of 6381
```

```
read:
listen: 172.16.13.19:26381
hash: fnv1a_64
distribution: ketama
auto_eject_hosts: true
redis: true
server_retry_timeout: 2000
server_failure_limit: 1
servers:
- 172.16.13.18:6380:1 server1 #slave of 6380
- 172.16.13.17:6381:1 server2 #slave of 6381
```



# DMP Architecture V1

## ◆ 读写分离后基准性能



# DMP Architecture V1

## ◆ v1 架构和v0架构的性能对比

4Nginx + 3Twemproxy									
并发数	100	200	300	400	500	600	700	800	900
每秒处理请求数	55440	65860	64162	78478	72216	79867	87994	82214	79043
平均每次请求延迟 (毫秒)	1.804	3.037	4.676	5.097	6.924	7.512	7.955	9.731	11.386
4Nginx + 3Redis									
并发数	100	200	300	400	500	600	700	800	900
每秒处理请求数	52158	56778	54353	66423	63352	68793	80687	80002	80264
平均每次请求延迟 (毫秒)	1.917	3.522	5.519	6.022	7.892	8.722	8.675	10	11.213

# DMP Architecture V1

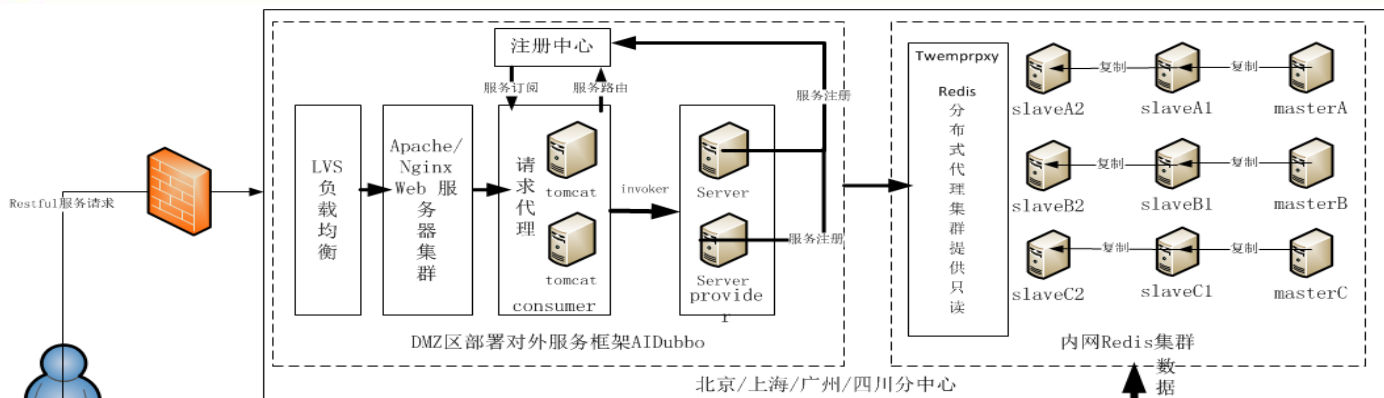
## ◆ v1 架构和v0架构的性能对比

	100	200	300	400	500	600	700	800	900
无代理	52619	55009	57155	61026	68324	71334	76135	78858	78141
2分片1复制	53878	64856	79959	87433	87245	87592	78798	84639	83101
2分片2复制	51467	63248	75729	83443	88998	86304	86676	83279	84823
3分片1复制	55874	71783	78419	76428	89071	89112	86265	83228	84084
3分片2复制	52382	62464	75407	81822	95647	86613	87165	83580	86698

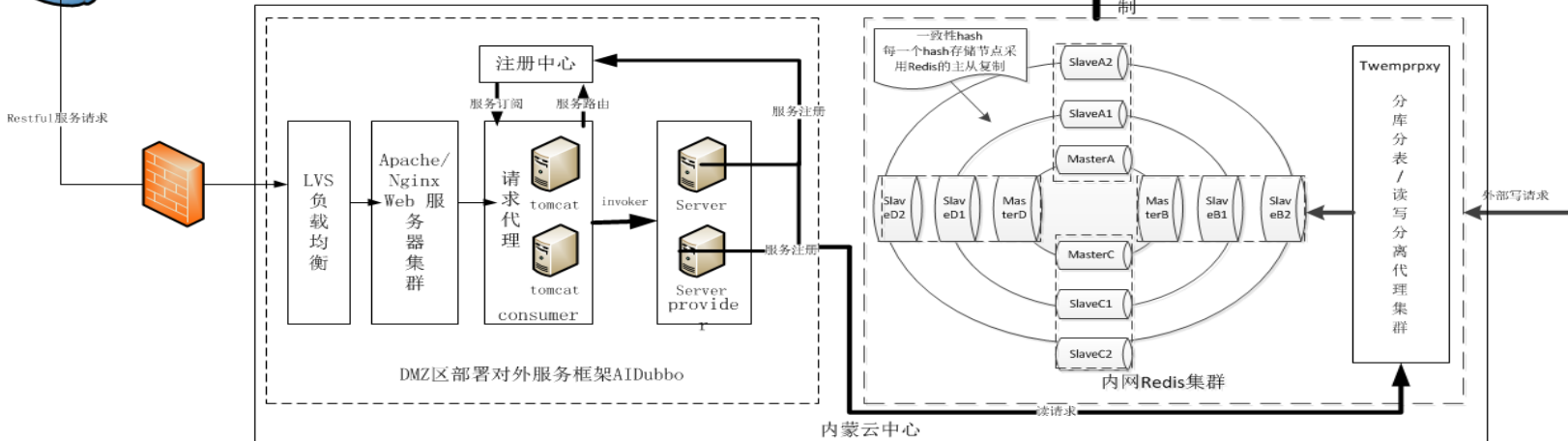
相同分片场景下，增加复制降低性能

相同数据副本场景下，增加数据分片对性能有提升

# DMP Architecture V2



- ◆ 统一的服务框架+Redis服务集群
- ◆ Redis端Lua实现业务逻辑，降低交互次数，提高并发和响应时间
- ◆ 存储结构Hash



# Performance

		1	10	25	50	100	200	300	400	500	600	700	800	900	1000	2000	3000
基准结果	A10+4NG+3proxy+6Redis	0.424	0.414	0.524	0.595	1.092	2.184	3.338	4.858	5.486	6.709	7.784	9.197	10.337	11.298	23.449	40.391
基准结果	A10+4NG+3proxy+3Redis	0.347	0.442	0.719	0.927	1.541	2.904	4.31	5.749	7.486	9.944	10.09	11.32	12.873	14.476	32.418	47.104

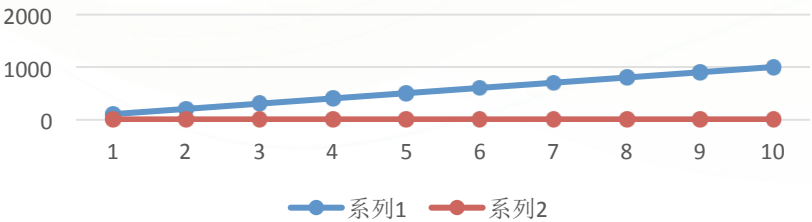
理论预期	A10+4NG+3proxy+6Redis	2358.5	24155	47710	84034	91575	91575	89874	82338	91141	89432	89928	86985	87066	88511	85291	74274
理论预期	A10+4NG+3proxy+3Redis	2881.8	22624	34771	53937	64893	68871	69606	69577	66791	60338	69348	70653	69914	69080	61694	63689

实测结果	A10+4NG+3proxy+6Redis	2358	24157	47682	84096	91545	91580	89885	82341	91135	89428	89929	86984	87069	88515	85289	74274
实测结果	A10+4NG+3proxy+3Redis	2881	22614	34759	53960	64910	68879	69604	69571	66791	60336	69350	70655	69916	69079	61694	63689

方差	A10+4NG+3proxy+6Redis	0	-3	27	-63	30	-5	-11	-3	6	4	-1	0	-4	-4	2	-1
方差	A10+4NG+3proxy+3Redis	0	10	11	-23	-18	-9	1	6	0	1	-2	-3	-3	0	0	-1

- 结果数据第一列为测试分析的原始响应时间，在结果数据100-1000，每次递增100个并发的延迟分析，响应时间是线性递减的
- 上方表格第二行数据为推测数据，计算公式为  $1000\text{ms}/\text{单位响应时间} \times \text{并发数}$ ，通过将理论值与实测结果对减的方差来看，测

响应延迟与并发比



## 存储设计

- Design a login user system
  - Heap table
    - | userid | login_times | last_login_time |
|--------|-------------|-----------------|
| 1      | 5           | 2011-1-1        |
| 2      | 1           | 2011-1-2        |
| 3      | 2           | 2011-1-3        |
    - Last login man? Create index on last\_login\_time
    - Max login man? Create index on login\_times
    - Index, explain plan, compute statistics is suck!!!

## 存储设计

- data
  - Set userid:1:login\_times 5
  - Set userid:2:login\_times 1
  - Set userid:3:login\_times 2
  - Set userid:1:last\_login 2011-1-1
  - Set userid:2:last\_login 2011-1-2
  - Set userid:3:last\_login 2011-1-3
- Last login
  - lpush user\_last\_login 1
  - lpush user\_last\_login 2
  - lpush user\_last\_login 3
  - ltrim user\_last\_login 0 1
- Max login man
  - zadd user:login\_times 5 1
  - zadd user:login\_times 1 2
  - zadd user:login\_times 2 3
  - zcard user:login\_times
  - zrangebyscore user:login\_times 3 5 withscores

**ZADD key score member [[score member] [score member] ...]**

将一个或多个 member 元素及其 score 值加入到有序集 key 当中

**ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]**

返回有序集 key 中, 所有 score 值介于 min 和 max 之间(包括等于 min 或 max )

的成员。有序集成员按 score 值递增(从小到大)次序排列。



## 存储设计

a) 用户活跃基站

输入：用户手机号码

输出：( 列表类型, list )

工作日TOP1基站, 工作日TOP2基站, 工作日TOP3基站

休息时TOP1基站, 休息时TOP2基站, 休息时TOP3基站

```
JSON{  
    '12345' : array{      'name' : 'Post A',      'val2' : 'blah blah',      'val3' : 'blah blah blah',  },  
    '54321' : array{      'name' : 'Post B',      'val2' : 'blah blah',      'val3' : 'blah blah blah',  },  
    '99887' : array{      'name' : 'Post C',      'val2' : 'blah blah',      'val3' : 'blah blah blah',  }  
};
```

b) 基站信息：

输入：基站ID

输出：基站名称, 室分类型, 经度, 维度

## 存储设计

### 结构扁平化

Redis多种数据类型只支持单层的存储，不支持多层组合嵌套存储，所以面对对层存储，需要把结构压平 比如多层的 key-val 结构，可以把 key 压平，继续存在 redis hash 里 object-id: [key: val, key1.keyl2: val, ...] 根据 redis 的文档，这种结构在 key 的数量很大（超过 255 个以上，有相关配置）时效率开始下降。

```
hset 133934**** [ "基站名称": "xxxx", "室分类型": "xxx", "经度" : "xxx" , "维度" : "xxx" ]  
hget 133934**** "基站名称"
```

## 存储设计

Lua脚本+json数据格式或messagepack格式

1	if redis.call("EXISTS", KEYS[1]) == 1 then
2	local payload = redis.call("GET", KEYS[1])
3	return cjson.decode(payload)[ARGV[1]]
4	else
5	return nil
6	end

1	redis-cli set 手机号 '{"基站名称": "fruit", "室分类型": "xxx", "经度": "xxx", "维度": "xxx"}'
2	=> OK
3	
4	redis-cli eval "\$cat json-get.lua" 1 (key的个数) 手机号 (key) 基站名称 (参数)
5	=> "fruit"

## 存储设计

### Hash Map

```
redis 127.0.0.1:6379> HMSET TEST_12345 name "Post A" val2 "Blah Blah" val3  
"Blah Blah Blah"  
OK  
redis 127.0.0.1:6379> HMSET TEST_54321 name "Post B" val2 "Blah Blah" val3  
"Blah Blah Blah"  
OK  
redis 127.0.0.1:6379> HMSET TEST_998877 name "Post C" val2 "Blah Blah" val3  
"Blah Blah Blah"  
OK
```

设置指定的值

```
redis 127.0.0.1:6379> HMSET TEST_12345 name "aaaaaa"  
OK  
redis 127.0.0.1:6379> HGETALL TEST_12345  
1) "name"  
2) "aaaaaa"  
3) "val2"  
4) "Blah Blah"  
5) "val3"  
6) "Blah Blah Blah"
```

使用 HMSET 设置单独的 fields 字段值，不影响其他字段值

获取指定的值

```
redis 127.0.0.1:6379> HGETALL TEST_12345  
1) "name"  
2) "Post A"  
3) "val2"  
4) "Blah Blah"  
5) "val3"  
6) "Blah Blah Blah"  
redis 127.0.0.1:6379> HGET TEST_12345 name  
1) "Post A"
```

## 数据结构的管理成本

- ◆ String类型的内存大小 = 键值个数 \* (dictEntry大小(24 bytes) + redisObject大小 (16 bytes) + 包含key的sds大小 + 包含value的sds大小) + bucket个数 \* 4
- ◆ zipmap类型的内存大小 = hashkey个数 \* (dictEntry大小(24 bytes) + redisObject大小(16 bytes) + 包含key的sds大小 + subkey的总大小) + bucket个数 \* 4

# DMP platform optimize

## Redis内存使用对比

采用标签ID用整型存储, 2000w条记录, 每条记录存21个标签ID总内存 **占用2.78G**

2000w条记录, 每条标签长度1K用字符串存储, 总内存 **占用25.43G**

建议使用标签ID的方式来存储

```
127.0.0.1:6379> info
# Server
redis_version:2.8.5
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:93f0290a76c4cdcc4
redis_mode:standalone
os:Linux 2.6.32-220.el6.x86_64 x86_64
arch_bits:64
multiplexing_api:epoll
gcc_version:4.4.6
process_id:4216
run_id:8368cbec2a04e1d3412d2be9e61b879c8944a055
tcp_port:6379
uptime_in_seconds:22214
uptime_in_days:0
hz:10
lru_clock:930216
config_file:/etc/redis_6379.conf

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:2989243632
used_memory_human:2.78G
used_memory_rss:3056955392
used_memory_peak:2989262752
used_memory_peak_human:2.78G
used_memory_lua:33792
mem_fragmentation_ratio:1.02
mem_allocator:jemalloc-3.2.0

# Keyspace
db0:keys=20000000,expires=0,avg_ttl=0
127.0.0.1:6379>
```

```
127.0.0.1:6378> info
# Server
redis_version:2.8.5
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:93f0290a76c4cdcc4
redis_mode:standalone
os:Linux 2.6.32-220.el6.x86_64 x86_64
arch_bits:64
multiplexing_api:epoll
gcc_version:4.4.6
process_id:4221
run_id:f688b3f487867af1c882c93747f9974ea6969f90
tcp_port:6378
uptime_in_seconds:24934
uptime_in_days:0
hz:10
lru_clock:930489
config_file:/etc/redis_6378.conf

# Clients
connected_clients:3
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:27309362120
used_memory_human:25.43G
used_memory_rss:27599908864
used_memory_peak:27309361192
used_memory_peak_human:25.43G
used_memory_lua:33792
mem_fragmentation_ratio:1.01
mem_allocator:jemalloc-3.2.0

# Keyspace
db0:keys=200000001,expires=0,avg_ttl=0
127.0.0.1:6378>
```

## Redis内存使用对比

是否可以继续优化，节约内存？

```
SET label:1155315 939  
GET label:1155315  
> 939
```

其中1155315是标签ID，939是标签值，我们将每一个标签ID为作key，值作为value来存成key-value对。  
将数据按上面的方法存储，**100w条数据会用掉70MB内存，3亿条记录就会用掉21GB的内存**

使用Hash结构后

```
HSET 1155 315 939  
HGET 1155 315  
> 939
```

**结果总内存占用从21个G降低到3个G.**

将数据分段，每一段使用一个Hash结构存储，由于Hash结构会在单个Hash元素在不足一定数量时进行压缩存储，所以可以大量节约内存，相关参数：

#如果Hash中字段的数量小于参数值，Redis将对该Key的Hash Value采用特殊编码。

hash-max-zipmap-entries

#如果Hash中各个字段的最大长度不超过512字节，Redis也将对该Key的Hash Value采用特殊编码方式。

hash-max-zipmap-value

尽量使用HASH

# DMP platform optimize

## DMP 平台对Lua的简单使用

- 通过redis端lua脚本封装代替nginx端lua脚本，将原来的两次网络交互，变成一次交互  
(IP->UID->Label)

```
location /geo_dsp_request {
    internal;
    if ($arg_act ~ 'label') {
        content_by_lua_file /usr/local/nginx/conf/redis_get_label.lua;
    }
    if ($arg_act ~ 'keyword') {
        content_by_lua_file /usr/local/nginx/conf/redis_get_keyword.lua;
    }
    if ($arg_act ~ 'host') {
        content_by_lua_file /usr/local/nginx/conf/redis_get_host.lua;
    }
    error_page 404 = @fetch;
}
```



```
location /redis_get_label {
    #internal;
    set $key $arg_key;
    #redis2_query get $key;
    redis2_query eval "return {redis.call('get',redis.call('get', ARGV[1]))} " 0 $key;
    redis2_pass label_stream;
    error_page 404 = @fetch;
}

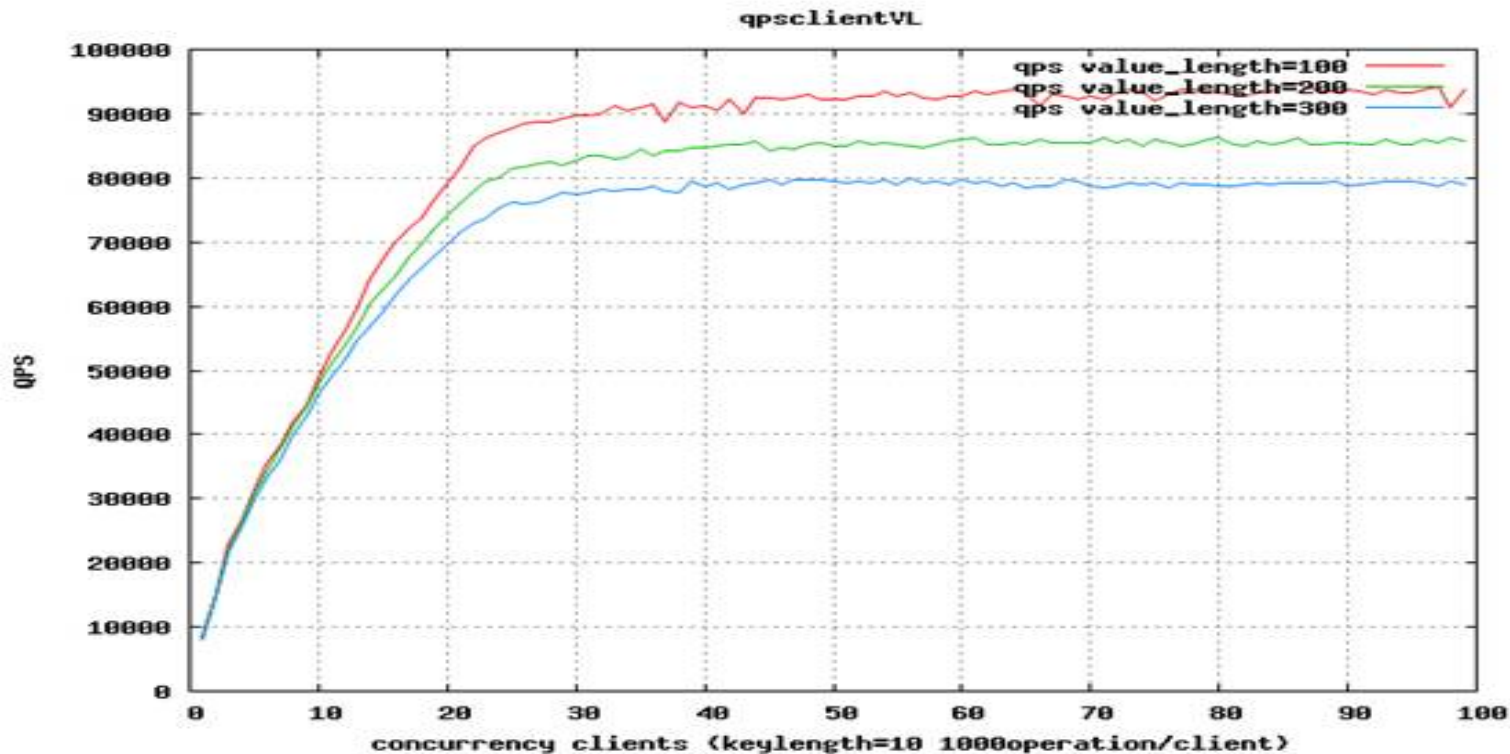
location /redis_get_keyword {
```





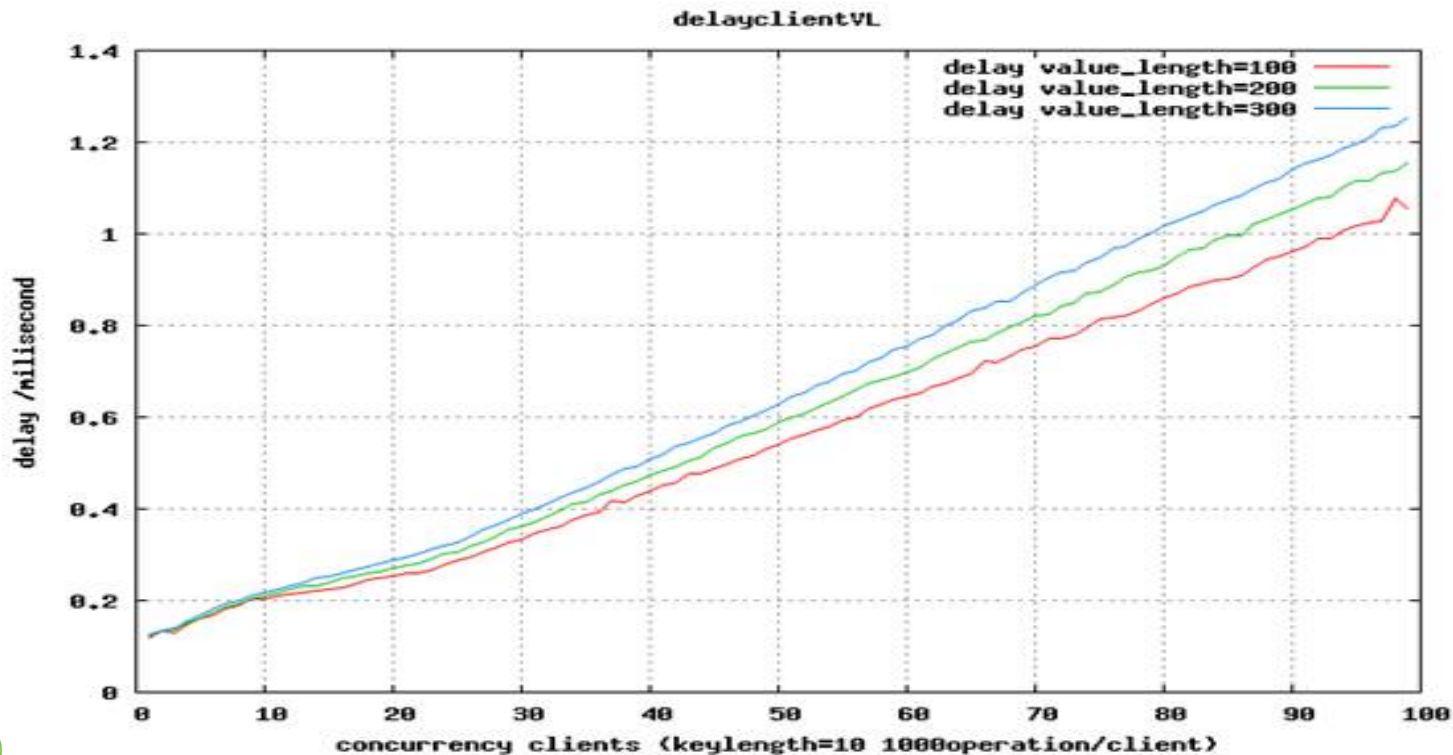
# Performance

不同的value\_length(r:w=4:1)



# Performance

不同的value\_length(r:w=4:1)



# Thanks



AsialInfo 亚信