

# LB 算法学习分享

何为公平?

- 角度
    - Client - LB - RS
  - 差异
    - RS 性能 / 连接状态 / ...
  - 场景
    - 扩缩容 / 跨地域 / ...
- ...  
怎么判断现在谁更有能力或更能及时处理?
- 当前 CPU 负载
  - 已有的连接数量
  - 网络连接空闲状态
  - 「发散」服务器主动接收（或任务窃取）

# 基础公平

雨露均沾

- RR (Round Robin, 轮询)

按照所有 rs 组成的队列，遍历的选择下一个还未调度的 RS，保证了最基础的调度公平但不考虑任何可能的差异

Real Server: A | B | C

i	0	1	2	3	4	5	...
select	A	B	C	A	B	C	...

# 从服务器间存在差异的角度

收集 RS 还有多少任务要处理

or

持续统计所有 RS 的 CPU 占用率

or

由 RS 性能来定义权重 

权重需要动态可调。对业务来说，新上线的 RS 可以自行动态的由低到高的缓慢修改其权重来实现预热等一些需求，或是新增 RS 的权重相应拉高来保证它在开始时有更多的任务去处理来快速分担其他 RS 的压力等

# (I)WRR

- WRR (Weighted Round Robin, 加权轮询)

批量的将与权重数量相当的包直接转发给选择的 RS  
Queue Level

- IWRR (Interleaved WRR, 交错的加权轮询)

轮询选择，权重减一，若其当前权重到 0 就直接跳过  
Packet Level

LVS WRR (反向 IWRR)

```
while (true) {
    i = (i + 1) mod n;      // i: initialized with -1
    if (i == 0) {
        cw = cw - gcd(S); // cw: current weight
        if (cw <= 0) {
            cw = max(S); // max: get the maximum weight of all servers
            if (cw == 0)
                return NULL;
        }
    }
    if (w(Si) >= cw)
        return Si;
}
```

# (I)WRR

- WRR (Weighted Round Robin, 加权轮询)

批量的将与权重数量相当的包直接转发给选择的 RS  
Queue Level

- IWRR (Interleaved WRR, 交错的加权轮询)

轮询选择，权重减一，若其当前权重到 0 就直接跳过  
Packet Level

问题：权重差

1. WRR 的权重配置中如果出现相差较多的权重差，就会导致某一段时间里做出连续相同的调度选择。这对 LB 来说是比较致命的，从 RS 的角度考虑，这一台较高会在短时间内打入大量的新连接 -> 平滑处理，打散高权重的连续调度
2. 一整轮调度中间的权重更新会导致重新开始的调度，从更长的时间线上看会加剧连续调度的问题 -> 权重延迟更新，只更新 RS 权重但不修改当前权重值，会使得新权重在下一轮才会被感知到

## LVS WRR 调度顺序模拟

```
while (true) {
    i = (i + 1) mod n;      // i: initialized with -1
    if (i == 0) {
        cw = cw - gcd(S); // cw: current weight
        if (cw <= 0) {
            cw = max(S); // max: get the maximum weight of all servers
        }
    }
    if (W(Si) >= cw)      // Weight: A(1) | B(5), GCD(1)
        return Si;
}
```

i	0	1	0	1	0	1	0	1	0	1	0	1	...
cw	5	5	4	4	3	3	2	2	1	1	5	5	...
select	B		B		B		B		A	B	B		...

# SWRR

- SWRR (Smooth Weighted Round Robin, 平滑加权轮询)

目的：打散高权重的连续调度

需要：配置权重

维护：当前权重、总权重

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其配置权重

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: **C**

current\_weight:

A

B

C

---

2

2

6

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C

current\_weight:

A

B

C

---

2

2

-4

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C A

current\_weight:

A

B

C

④

4

2

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C A

current\_weight:

A

B

C

---

-6

4

2

---

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C A **C**

current\_weight:

A

B

C

---

-4

6

8

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C A C

current\_weight:

A

B

C

---

-4

6

-2

---

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C A C B

current\_weight:

A

B

C

---

-2

(8)

4

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C A C B

current\_weight:

A

B

C

---

-2

-2

4

---

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C A C B **C**

current\_weight:

A

B

C

---

0

0

10

# SWRR

对于每一次选择：

1. 选择当前权重最高的 RS 调度
2. 选择的 RS 当前权重减去总权重，加速抹平权重间的差值
3. 为了保证当权权重之和仍然等于总权重，为每个 RS 的当前权重增加其定义的权重

RS: A(2) | B(2) | C(6), total\_weight(10), GCD(2)

WRR: C C A B C

SWRR: C A C B C

current\_weight:

A

B

C

---

0

0

0

---

# SWRR

核心逻辑：规律的调整权重但永远保持总权重不变

问题：调度时间开销 和 相同调度选择

1. 每次调度需要遍历所有的 RS 找到当前权重最大的一个，查找调度时间开销为  $O(N)$  或  $O(\log N)$ （如果选择用例如最大堆维护，就需要额外花时间维护堆），更新当前权重  $O(N) \rightarrow$  预先计算调度顺序，调度时可以直接  $O(1)$  选择
2. 集群中每一个单机和单机中每一个 WORKER 都会做出相同的调度选择，使得从整体上来看（或从 RS 的角度看）LB 集群会做出多次连续重复的调度  $\rightarrow$  调度顺序随机初始位置
3. 其他优化方案参见 Nginx 章节

# VNSWRR

- VNSWRR (Virtual Node Smooth Weighted Round Robin, 虚拟节点平滑加权轮询)

预先计算好完整的调度顺序表，并在一开始选择随机的 idx 开始调度

LB(5), SWRR: C A C B C

LB id	random idx	select	select	select	select	select
0	0	C	A	C	B	C
1	1	A	C	B	C	C
2	2	C	B	C	C	A
3	3	B	C	C	A	C
4	4	C	C	A	C	B

# VNSWRR

- VNSWRR (Virtual Node Smooth Weighted Round Robin, 虚拟节点平滑加权轮询)

预先计算好完整的调度顺序表，并在一开始选择随机的 idx 开始调度

LB(5), SWRR: C A C B C

从整体的角度看

SWRR: C C C C C A A A A A C C C C C B B B B B C C C C C

VNSWRR: C A C B C A C B C C C B C C A B C C A C C C A C B

# VNSWRR

- VNSWRR (Virtual Node Smooth Weighted Round Robin, 虚拟节点平滑加权轮询)

预先计算好完整的调度顺序表，并在一开始选择随机的 idx 开始调度

问题：

1. 预先计算的时间开销
2. 每次做权重修改或发生 RS 扩缩容都需要重新计算
3. GCD 优化调度顺序表长度

# VNSWRR

- VNSWRR (Virtual Node Smooth Weighted Round Robin, 虚拟节点平滑加权轮询)

## GCD 优化

调度顺序表的长度取决于总权重，但实际上会以所有 RS 权重的最小公倍数为一轮，所以只需要存放 GCD 后的总权重长度

实际上权重本来就可以除最大公约数进行约分，等效权重

weight	A	B	C
before	2	2	6
after	1	1	3

# VNSWRR - 预计算开销

N: RS number, M: total weight after dividing GCD

伪代码 计算调度顺序表

```
M <- count total weight # O(N)
malloc vrs_table with M
for every loop idx(auto increment) in M: # O(M)
    find RS a with max effective weight # O(N)
    RS a effective weight update # O(1)
    update all RS effective weight # O(N)
    vrs_table[idx] = RS a
```

伪代码 调度

```
random start index i
for every schedule:
    RS a = vrs_table[i] # O(1)
    i = (i + 1) % M
    # make sure RS a alive
    while RS a is not alive:
        RS a = vrs_table[i]
        i = (i + 1) % M
    return RS a
```

# VNSWRR - 预计算开销

N: RS number, M: total weight after dividing GCD

伪代码 计算调度顺序表

```
M <- count total weight # O(N)
malloc vrs_table with M
for every loop idx(auto increment) in M: # O(M)
    find RS a with max effective weight # O(N)
    RS a effective weight update # O(1)
    update all RS effective weight # O(N)
    vrs_table[idx] = RS a
```

伪代码 调度

```
random start index i
for every schedule:
    RS a = vrs_table[i] # O(1)
    i = (i + 1) % M
    # make sure RS a alive
    while RS a is not alive:
        RS a = vrs_table[i]
        i = (i + 1) % M
    return RS a
```

# VNSWRR - 预计算开销

N: RS number, M: total weight after dividing GCD

伪代码 计算调度顺序表

```
M <- count total weight # O(N)
malloc vrs_table with M
for every loop idx(auto increment) in M: # O(M)
    find RS a with max effective weight # O(N)
    RS a effective weight update # O(1)
    update all RS effective weight # O(N)
    vrs_table[idx] = RS a
```

SWRR 计算过程中可以划分 Step 打散对调度顺序表的计算

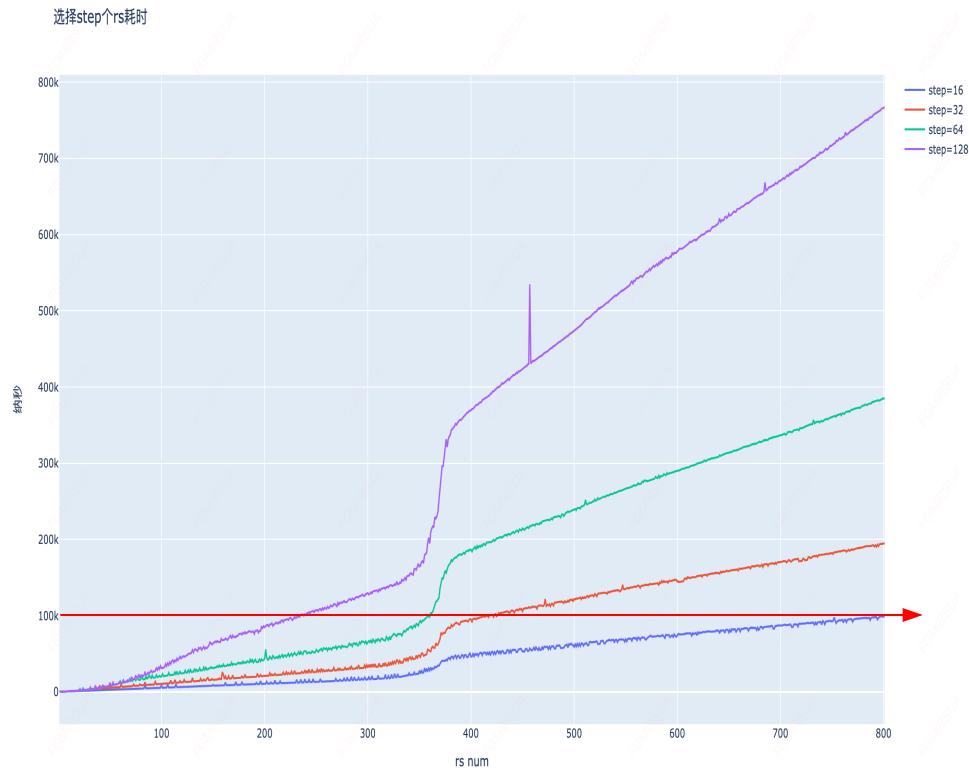
$$O(MN) \rightarrow O(Step \times N) \times \frac{M}{Step}$$

# VNSWRR - 预计算开销

# VNSWRR - 预计算开销

rs num	step
<= 200	128
<= 300	64
<= 400	32
more	16

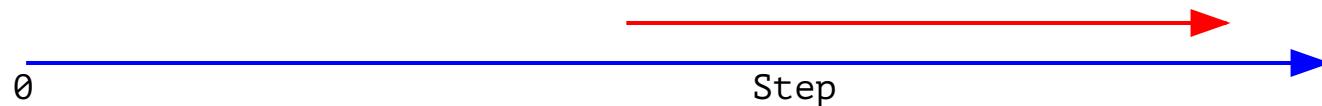
100k 纳秒



# VNSWRR - 划分 Step 后的随机初始 idx

划分 Step 也就意味着，随机初始坐标的范围被限制到  $[0, \text{Step})$  里了

随机打乱同 current weight 时的 RS 选择顺序，扩大随机范围



# VNSWRR

问题：分步计算、权重更新和乱序算法

现在的分布计算会在前几次调度时计算下一个 Step 的调度顺序表

伪代码 调度

```
random start index i in (0, Step)
next init idx j = Step
for every schedule:
    if j < M:
        calculate next Step
        j += Step
    RS a = vrs_table[i] # O(1)
    i = (i + 1) % M
    # make sure RS a alive
    while RS a is not alive:
        RS a = vrs_table[i]
        i = (i + 1) % M
return RS a
```

# VNSWRR

问题：分步计算、权重更新和乱序算法

现在的分布计算会在前几次调度时计算下一个 Step 的调度顺序表

改为 next\_idx 快到 next\_init\_idx 时再计算下一个 Step，打散调度顺序的计算

伪代码 调度

```
random start index i in (0, Step)
next init idx j = Step
for every schedule:
    if j == i:
        calculate next Step
        j += Step
    RS a = vrs_table[i] # 0(1)
    i = (i + 1) % M
    # make sure RS a alive
    while RS a is not alive:
        RS a = vrs_table[i]
        i = (i + 1) % M
return RS a
```

# VNSWRR

问题：分步计算、权重更新和乱序算法

每次有 RS 的权重更新都会导致调度顺序表需要重新计算。如果有频繁的权重更新，会使得调度顺序表的计算开销被放大

- 问题1的解决方案能部分保证不会连续的计算 Step 内调度顺序
- 调度顺序的计算现在放在 CTRL 线程，因读写锁与 WORKER 的调度互斥，改为 RCU 机制使计算时的调度可以继续用旧的调度顺序表来做调度选择？
- 全局完整的调度顺序表计算（CTRL 线程），WORKER 本地只随机 next\_idx in [0, M]？

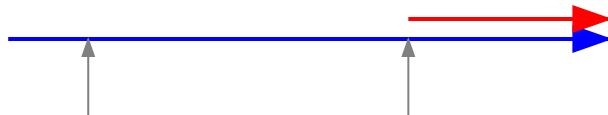
# VNSWRR

问题：分步计算、权重更新和乱序算法

当前的乱序算法是将所有 RS 以随机顺序挨个添加到新的临时链表里， $O(N^2)$

参考洗牌算法 [BV1tNKfz1Eqc](#)

1. 每张牌给随机值，按照随机值排序，时间  $O(N \log N)$ ，空间  $O(N)$  但可以和 Maglev 一致性哈希共用偏好数组
2. 新版 Fisher-Yates 算法：依赖数组  $O(1)$  取值。维护牌堆数组的洗好和未洗两部分，每次将未洗的最后一张牌 `last_idx` 与随机未洗位置 `random_idx` ( $0 \sim \text{last\_idx}$ ) 的牌  $O(1)$  交换，并将最后一张 `last_idx` 当作洗好的部分。最终可以  $O(N)$  时间随机洗牌。但需要额外存储一个 RS 数组用来取 `random_idx`



# 不同的 WRR 算法对比

(I)WRR (Packet number level)	循环调度周期短 实现简单，维护信息少 权重延迟更新	权重差值大连续调度
SWRR (Packet level)	平滑解决连续问题	<b>O(N)</b> 调度 相同序列调度压力
VNSWRR (RS level)	预先计算， <b>O(1)</b> 调度 打散重新计算时的流量 分步更新减缓计算开销	不共享，单独计算 next_idx 随机范围不是完整的调度顺序表长度

# 从服务器连接状态的角度

相比于预定义的权重信息，统计当前的连接状态可以更灵活的处理 RS 的调度选择问题

因为 LB 以集群方式部署，本身从 RS 上收集会更能表示其真实的已有的连接状态，但是需要改造 RS 以例如 Sidecar 模式额外的收集信息且对 RS 造成了额外的开销。对 DGW 更合适的方式是在自己集群内进行全局的统计

- minConn (最小连接数)

维护对所有 RS 的连接数统计，每次选择当前连接数最少的 RS 调度

通过维护连接数最小堆的方式  $O(1)$  的选择调度并  $O(\log N)$  的维护

- 由于 WORKER 之间不共享，需要额外的定时任务来收集所有 WORKER 的连接数信息更新到每一个 WORKER 本地，且面对大量的连接数更新，直接重建堆
- 由于集群内不共享，需要依靠会话同步的方式来获取其他机器的连接数统计

# minConn

问题：

1. 不是从 RS 收集的真实连接数，无法感知其他连接
2. 维护非常麻烦，集群内、WORKER 间 收集，最小堆维护和定时重建
3. 集群内会话同步通常只收集长连接，面对大量持续建立的新短连接没有办法建立全局视野
4. 单纯的 minConn 调度时不会考虑 RS 的权重分配，建立连接时只会将本 WORKER 连接数 + worker\_cnt

# minConn

问题：

1. 不是从 RS 收集的真实连接数，无法感知其他连接
2. 维护非常麻烦，集群内、WORKER 间 收集，最小堆维护和定时重建
3. 集群内会话同步通常只收集长连接，面对大量持续建立的新短连接没有办法建立全局视野
4. 单纯的 minConn 调度时不会考虑 RS 的权重分配，建立连接时只会将本 WORKER 连接数 + worker\_cnt

W-minConn 带权重的最小连接数判断

- 建立连接时连接数 +  $worker\_cnt \times weight$ ，统计时也直接计算  $conn\_cnt \times weight$  来建堆（注意这里"权重"越低，"权重"越高：）
- 除权重后向上取整
- 两两对比时， $conn_A \times weight_B > conn_B \times weight_A$

# 从数据包间存在长短差异的角度

数据包之间会有长短（尽管这在 DGW 认为是一致的），它可能影响的更多是收发包的时间开销

参考上一章的权重分配，这里将数据包的大小作为权重的影响因素。刚发送了长数据包的 RS 不应成为下一次的选择

- DWRR (Deficit Weighted Round Robin, 赤字轮询)

赤字计数器表示此轮可以发送的最大字节数，如果其大于等于数据包大小则此轮可以发送（并将计数器减掉数据包大小），否则放到下一轮考虑

赤字计数器每轮均等增加或按照权重配置增加。对应到 DGW 是为所有 RS 维护赤字计数器，每轮增加权重值，并选择下一个计数器值大于数据包大小的 RS 调度

Packet	w	RS	curr	curr	curr
6	2	A	2	4	4
	4	B	4	8	2

# 从完全随机的角度

通过哈希的方式可以保证相同的明文值每次都会哈希为相同的密文（幂等性），额外的保证

- 相同的 QUIC CID 转发到相同的 RS 进行处理
- 新连接依靠相同的 hash key 打回与上一次连接相同的 RS（天然的会话保持能力）

公平性依靠哈希算法的散列程度

- Hash % Mod

根据某一个特定的值进行哈希（随机打散），再对 RS 数量取模

DGW 本身存储 RS 是以链表和哈希桶的方式，需要再额外以数组形式存储调度顺序表以保证 O(1) 的调度选择

# Hash % Mod

调度公平的影响因素为哈希算法和 RS 数量

1. 首先需要保证哈希明文的唯一和哈希结果尽量分散，尤其当 RS 数量较少的时候，若再遇到哈希碰撞就会导致很明显的调度不均
2. RS 数量变化会导致之前哈希的结果都发生改变

RS number:  $m(4) \rightarrow n(5)$

不变的映射为: 0, 1, 2, 3, 20, 21, 22, 23, ...

LCM(Least Common Multiple), 最小公约数

$$\frac{\min(m, n)}{LCM(m, n)}$$

# Hash % Mod

调度公平的影响因素为哈希算法和 RS 数量

1. 首先需要保证哈希明文的唯一和哈希结果尽量分散，尤其当 RS 数量较少的时候，若再遇到哈希碰撞就会导致很明显的调度不均
2. RS 数量变化会导致之前哈希的结果都发生改变

解决方案：

1. 填充虚拟节点，增加槽位
2. 一致性哈希降低重新映射的数量

K: number of keys, n: number of slots, 槽位改变平均只需要重新映射

$$\frac{K}{n}$$

# Consistent Hash 一致性哈希

## 1. Load Balancing 均衡

每一个 RS 获取任何一个 key 的机会均等

## 2. Minimal Disruption/Remapping 最小化重映射

$$\frac{K}{n}$$

1. 对于新增 RS，已有的 key 映射不变或重映射到新 RS，且总的重映射可控
2. 对于删除 RS，已有映射到非此 RS 的映射不变，映射到此 RS 的分散给其他 RS

哈希范围通常不能和可变值绑定

# Consistent Hash 一致性哈希

- 哈希环（割环法） - 1997
- Jump Hash 跳跃一致性哈希 - 2014
- Multi-Probe 一致性哈希 - 2015
- Maglev 一致性哈希 - 2016
- AnchorHash 一致性哈希 - 2020
- DxHash 一致性哈希 - 2021
- Rendezvous 哈希 - 1997

# Rendezvous 哈希

「核心思想」

如果只是对服务器ID进行哈希，那么当修改服务器的数量时，所有的哈希值都会发生变化。当对目标服务器的选择和服务器的数量没有直接关系时，就可以避免服务器的增删带来的影响

「算法思路」

为每个 key 生成一个随机有序的服务器列表，并选择列表中的第一个作为目标服务器

如果选择的第一台服务器下线时，只需要将 key 转移到列表中的第二台服务器并作为新的第一台服务器即可

1. 对每个服务器计算  $key:rs\_id$  哈希来生成一组整数哈希值
2. 基于该哈希值对服务器进行排序，得到一个随机排列的服务器列表

对于有权重分配的场景，可以基于  $w \div \ln h(x)$  排序， $h(x)$  哈希范围在  $[0, 1]$

# Rendezvous 哈希

优势：

1. 将服务器选择和数量完全解绑，并提供了第二选择服务器，解决了哈希级联故障转移的问题
2. 没有额外的内存存储开销

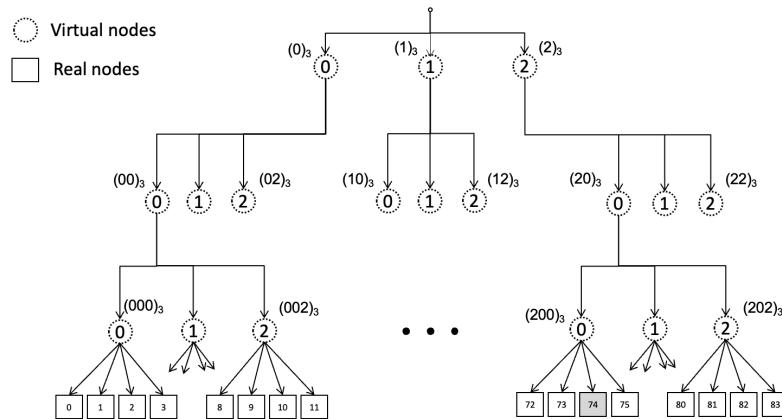
问题：调度的时间开销和扩容影响第一选择

1. 调度时需要对所有 RS 进行哈希计算并排序，时间开销  $O(N \log N)$ 。实际上在取 RS 哈希时应该先检查存活，所以只需要  **$O(N)$**  查找哈希值最大的一个即可
2. 扩容时新服务器可能成为一些已有 key 的第一选择，所以很难维护第一选择的不变。在分布式存储场景下需要重新校验所有 key，但在缓存和负载均衡场景下影响可以接受

# Rendezvous 哈希

「变体」调度时间优化至  $O(\log N)$

把原始节点分成若干个虚拟组，虚拟组一层一层组成一个“骨架”，然后在虚拟组中按照 Rendezvous 哈希计算出最大的节点，从而得到下一层的虚拟组，再在下一层的虚拟组中按同样的方法计算，直到找到最下方的真实节点



# Rendezvous 哈希

「扩展思考」

不要对每一个 RS 都计算哈希，直接对 key 哈希后切分得到有序的服务器列表

切分时为了和 RS 数量解耦，需要按照固定的长度切分，如 8 bits 为一组进行映射（即 RS 最多 256 个），但就会面临大量的 hash\_val 无法映射到节点的问题

->

DxHash 一致性哈希

# DxHash 一致性哈希

NSArray 是长度为大于 RS\_num 的最小  $2^n$  值，  
例如 RS 4 台，NSArray 长度则为 8

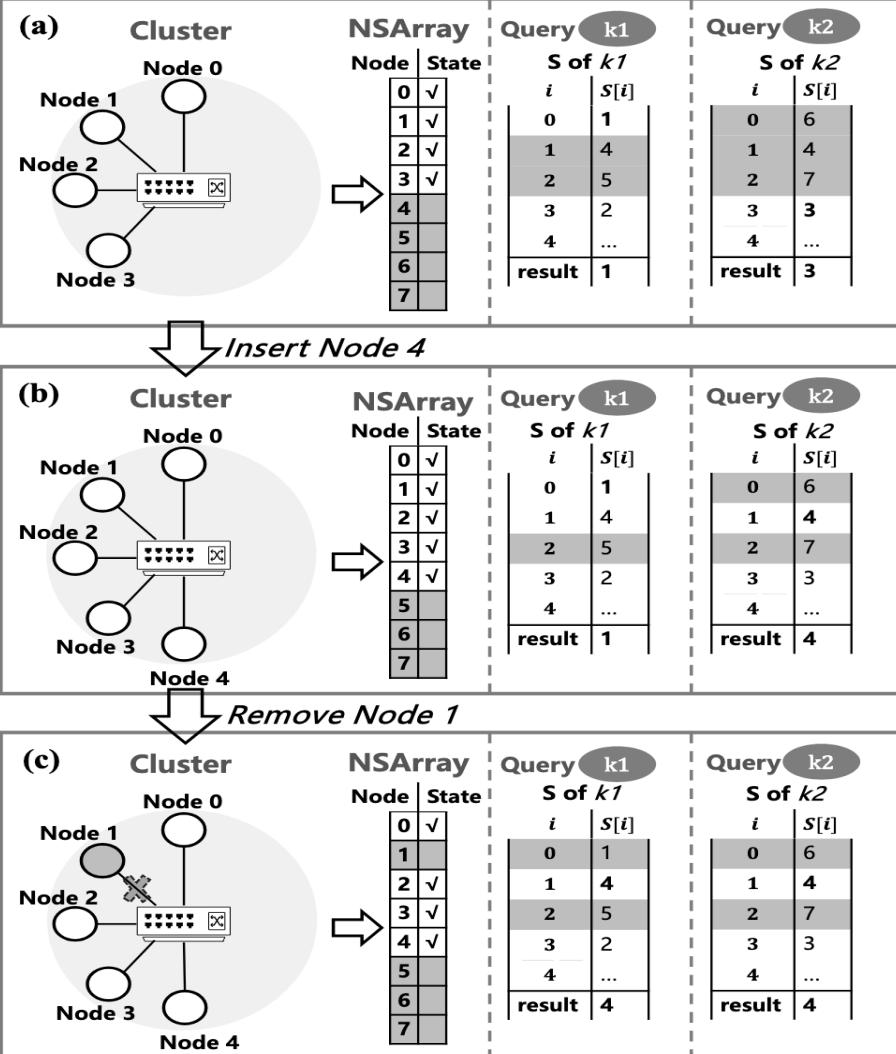
通过伪随机数机制来保证同一个 key 有固定有序  
且无限长度的服务器序列

Minimal Disruption: the changed node is either the  
original or the destination of the remapped keys.

为了防止无限长的服务器序列但一直映射不到活动节点，DxHash 对搜索的次数限制到  $8n$ ， $n$  为 cluster size

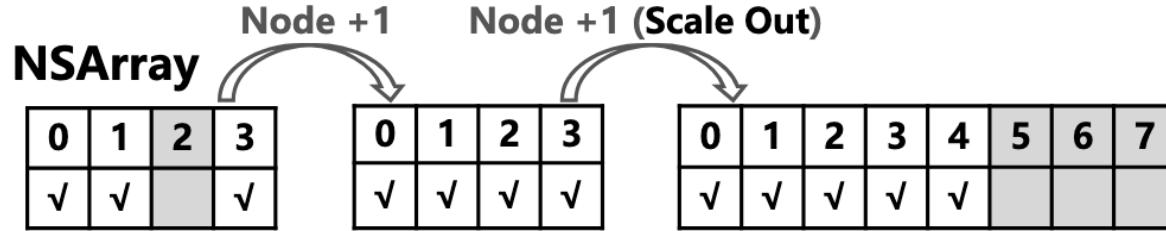
$$P = \left(\frac{n-1}{n}\right)^{8n}$$

当  $n$  足够大， $P$  接近  $\frac{1}{e^8}$ ，约为 0.03%，作者认为概率足够小



# DxHash 一致性哈希

NSArray 两倍扩容 + 节点迁移



可以以切片的形式来保证不会需要节点迁移

「质疑」作者没有讲 NSArray 扩容后出现的大量变化的重映射的问题

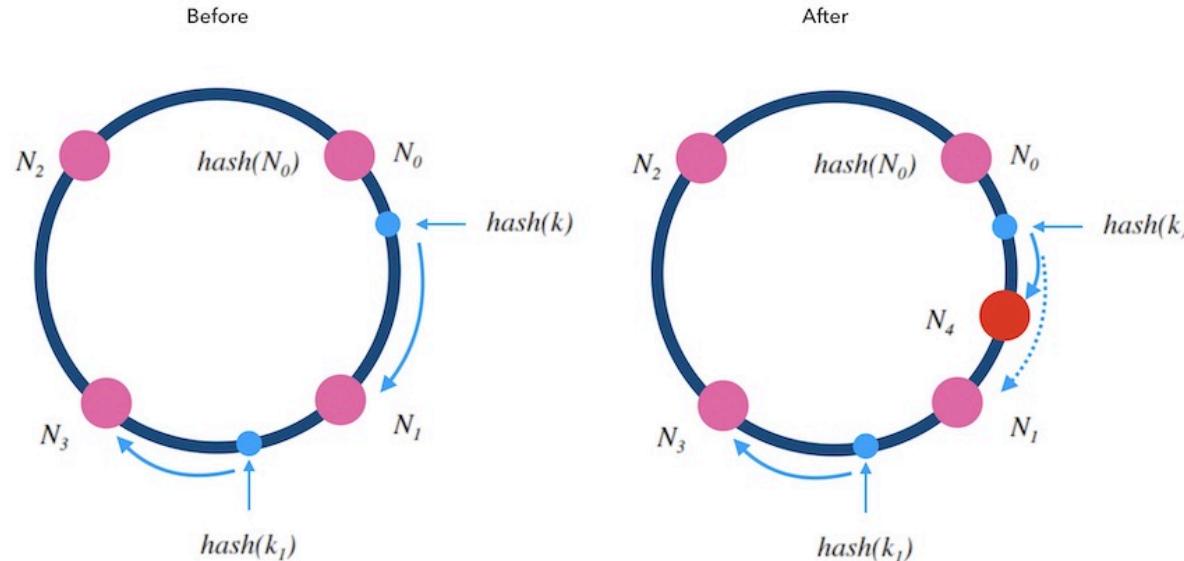
When the cluster reaches its maximum capacity and all items in the NSArray are active, DxHash behaves as a classic hash algorithm that maps objects to nodes with a single calculation.

权重通过虚拟节点层实现

# CHash - 哈希环（割环法）

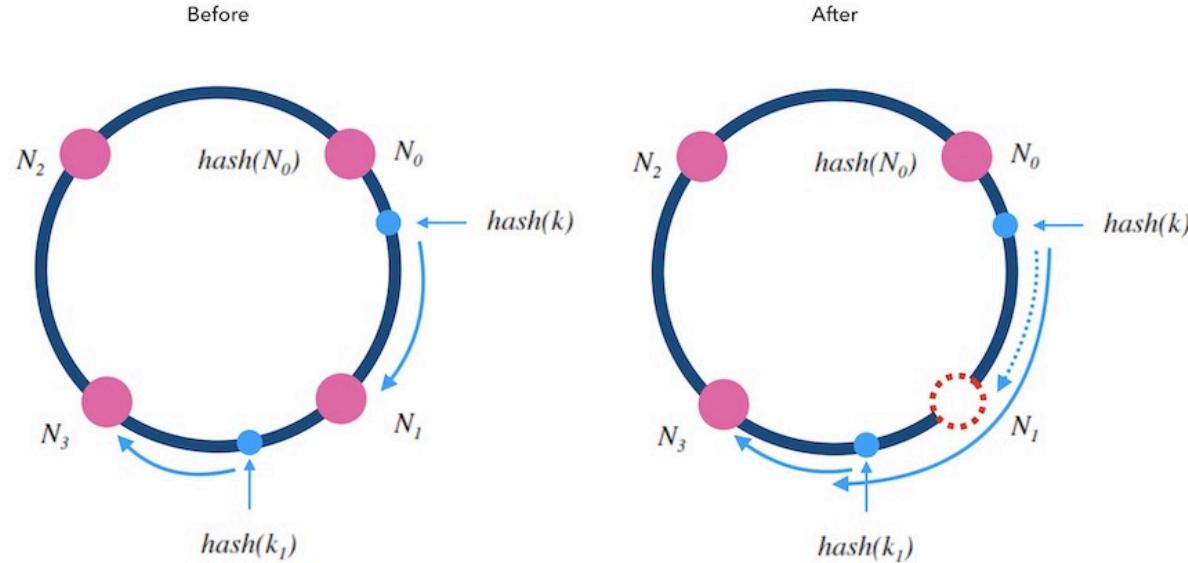
哈希值映射到一个大圆环( $2^{32}$ )空间内的槽位，查找时在圆环中顺时针查找映射过的槽位

当往一个哈希环中新增一个槽位时，只有被新增槽位拦下来的哈希结果的映射结果是变化的



# CHash - 哈希环（割环法）

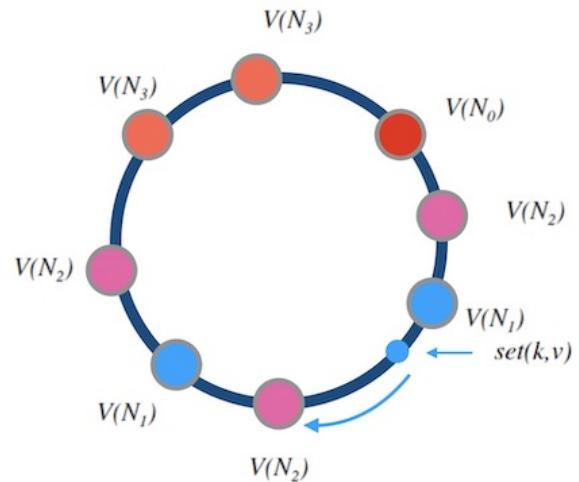
当从一个哈希环中移除一个槽位时，被删除槽位的映射会转交给下一槽位，其他槽位不受影响



# CHash - 哈希环（割环法） - 带权重

在实际应用中，还可以对槽位（节点）添加权重，通过构建很多指向真实节点的虚拟节点，也叫影子节点。通常采用一个节点创建 40 个影子节点，节点越多映射分布越均匀。影子节点之间是平权的，选中影子节点，就代表选中了背后的真实节点。权重越大的节点，影子节点越多，被选中的概率就越大

但是需要注意的是，影子节点增加了内存消耗和查询开销，权重的调整也会带来数据迁移的工作



# CHash - 哈希环（割环法）

问题：调度查找开销和额外空间开销

1. 调度时需要在哈希环中顺序的查找到槽位（或虚拟节点），造成额外的查找开销
2. 需要开辟大量空间用来做哈希环和影子节点
3. 哈希环算法的映射结果不是很均匀，当有 100 个影子节点时，映射结果的分布的标准差约 10%；当有 1000 个影子节点时，降低到约 3.2%

优势：一致性哈希和权重分配

# CHash - 哈希环（割环法）

问题：调度查找开销和额外空间开销

1. 调度时需要在哈希环中顺序的查找到槽位（或虚拟节点），造成额外的查找开销

预先填充整个哈希环， $O(1)$  调度查找

如果是 32bits 的哈希范围，就代表需要开辟  $\text{sizeof}(\text{ptr}) \times 2^{32}$  大小的数组，4 字节指针时总存储约为 17GB。  
但如果 16bits (65536)，就只需要 256KB

所以  $2^{32}$  长度的环是代表哈希范围，为理论的虚拟数组

# CHash - 哈希环（割环法）

「实现方式」链表

建立以影子节点 hash 值排序的虚拟节点链表，并在这基础上以类似跳表的形式加速查询时 hash 定位影子节点

伪代码 链表形式哈希环

```
hash_ring {
    len;          // shadow_node length = sum(node * weight * shadow_cnt)
    shadow_node *list;
}

shadow_node {
    *rs;
    hash;
    *next_shadow_node;
}

jump_list[level][] = {};
```

跳表的层数和删除，逻辑复杂

# CHash - 哈希环（割环法）

「实现方式」数组

改为用数组存储虚拟节点表，同样以影子节点的 hash 值排序。查找时对查询的 key hash 值二分查找，简化逻辑（普遍选择的做法）

查询同样是  $O(\log N)$

2. 需要开辟大量空间用来做哈希环和影子节点
3. 哈希环算法的映射结果不是很均匀，当有 100 个影子节点时，映射结果的分布的标准差约 10%；当有 1000 个影子节点时，降低到约 3.2%

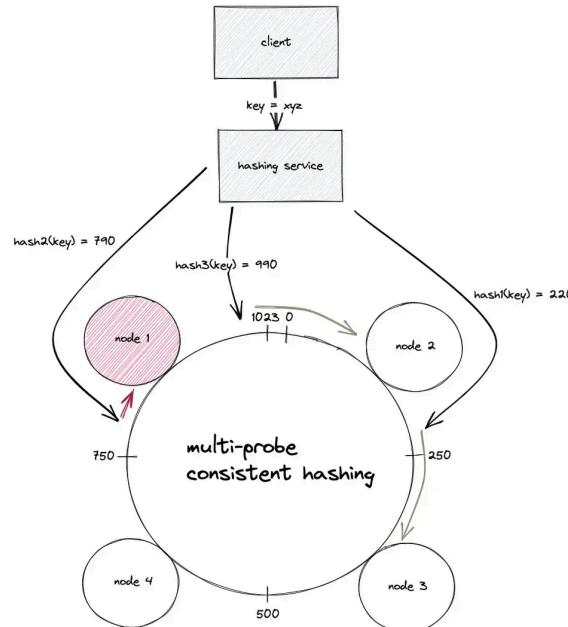
一致性哈希：算法均衡 by Damian Gryski

# CHash - Multi-Probe 一致性哈希 (MPCH)

目标：灵活调节节点大小和降低方差

基本思想是在哈希环的基础上查找时对 key 进行 k 次不同哈希，返回所有哈希查询中距离最近的节点。k 的值由所需的方差决定

对于峰均值比 1.05 (负载最重的节点最多比平均值高 5%)，k 为 21。作为对比，哈希环算法需要  $700 \ln N$  个副本。对于 100 个节点，这相当于超过 1MB 内存



# CHash - Jump Hash 跳跃一致性哈希

```
int32_t JumpConsistentHash(uint64_t key, int32_t num_buckets) {
    int64_t b = -1, j = 0;
    while (j < num_buckets) {
        b = j;
        key = key * 2862933555777941757ULL + 1;
        j = (b + 1) * (double(1LL << 31) / double((key >> 33) + 1));
    }
    return b;
}
```

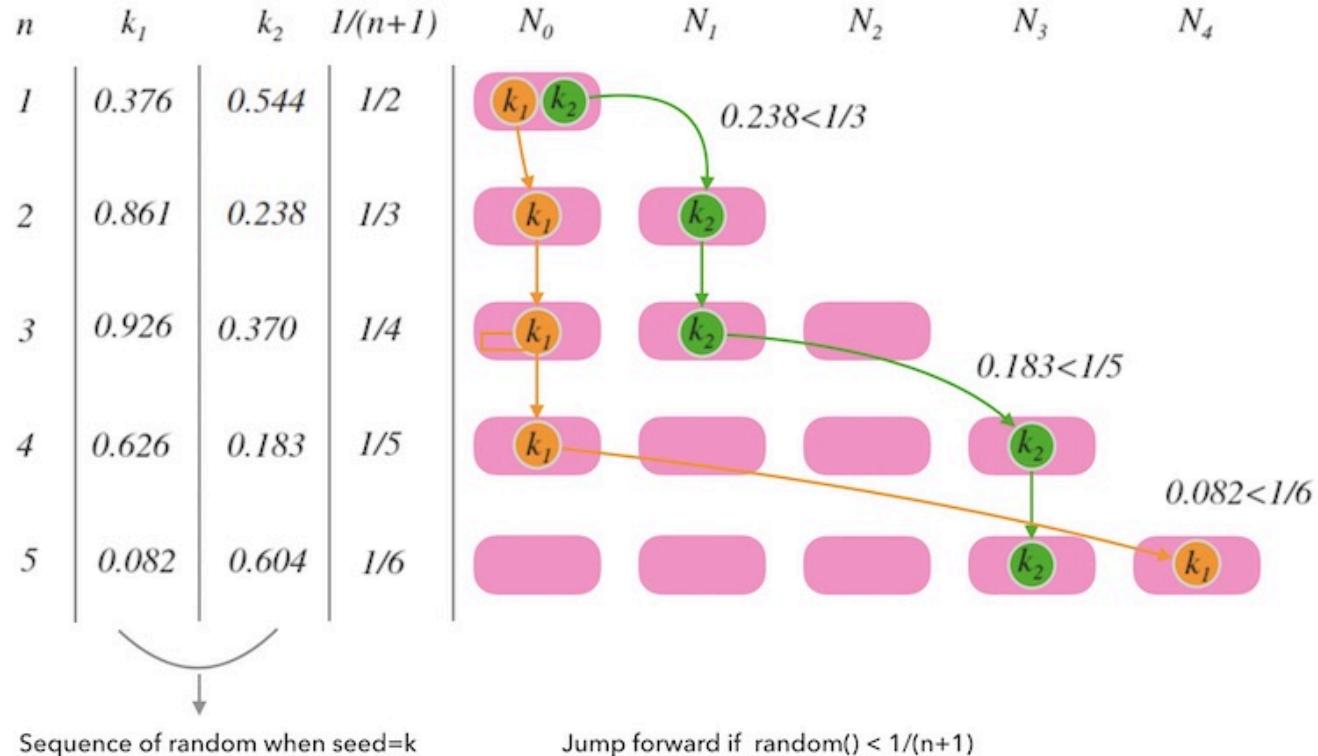
当槽位数量发生变化时，可以直接计算有多少个哈希结果需要重新映射。通过伪随机数来决定一个哈希结果每次要不要跳到新的槽位中去，最终只需要保证新的桶中有

$$key\_cnt \div slot\_cnt = \frac{K}{n}$$

个从前面槽位移动过来的 key 即可保证最小化重新映射

由于是通过伪随机的方式，并将哈希 key 作为伪随机数种子，对于给定的哈希槽位数量，key 的映射结果都是唯一确定的

# CHash - Jump Hash 跳跃一致性哈希



# CHash - Jump Hash 跳跃一致性哈希

「伪随机哈希」

对同一个 key 的跳跃序列是一致的

```
int consistent_hash(int key, int num_buckets) {
    srand(key);
    int b = 0;
    for (int n = 2; n <= num_buckets; ++n) {
        if ((double)rand() / RAND_MAX < 1.0 / n) // 每次都需要判断当前随机结果是否应该跳到最新的 bucket 里
            b = n - 1;
    }
    return b;
}
```

O(N) 时间查询

# CHash - Jump Hash 跳跃一致性哈希

## 「伪随机哈希优化」

在上面的实现里，需要判断  $N$  次伪随机值来确定是否要跳跃到当前的最大桶上，计算哈希所需要花费的时间时间  $O(N)$ 。但是注意到元素只有很小的概率会移动，它只会在桶增加时移动，且每次移动都必然移动到最新的桶里，即如果一个元素移动到  $b$  号桶（从 0 开始计号）里，必然是因为桶增加到  $b+1$  个导致。所以我们只需要求出下一次移动的目标  $j$ ，即可跳过  $b+2 \dots j$  次伪随机的步骤

下一次移动到  $j$  意味着  $b+2 \dots j$  都伪随机到了不移动。另我们知道当桶增加到  $N$  个时元素的移动概率为  $1/N$ ，不移动的概率为  $(N-1)/N$ 。所以元素移动到  $j$  的概率  $P_j$  为

$$P_j = (b+1)(b+2)(b+3)\dots(j-1)/(b+2)(b+3)(b+4)\dots j = (b+1)/j$$
$$j = (b+1)/P_j$$

那么我们可以改变思路，将  $P_j$  作为伪随机的值  $r$ ，就可以直接通过伪随机值来获取  $j$  了

$$j = \text{floor}((b+1)/r)$$

# CHash - Jump Hash 跳跃一致性哈希

「伪随机哈希优化」

```
int consistent_hash(int key, int num_buckets) {
    srand(key);
    int b = 1, j = 0;
    while (j < num_buckets) {
        b = j;
        r = (double)rand() / RAND_MAX;
        j = floor((b + 1) / r);
    }
    return b;
}
```

O(logN) 时间查询

# CHash - Jump Hash 跳跃一致性哈希

优势：在执行速度、内存消耗、映射均匀性上都比哈希环算法更好，时间可以由  $O(N)$  优化至  $O(\log N)$

问题：

1. 无法自定义槽位标号，必须从 0 开始，意味着需要 **rs\_buf** 数组
2. 只能在尾部增删节点，导致删除中间节点  $i$  需要先把后面所有的  $[i :]$  都删掉，再把  $[i + 1 :]$  的添加回来  
(DGW 不需要维护已建立过的连接 key)

「实际的实现方式」

如果不需要做 100% 保证（不需要维护 key 的移动），只需要使用 `JumpConsistentHash` 选择出调度的 RS 即可

$$RS\_idx \leftarrow JumpConsistentHash \leftarrow key(srcIP/QUIC\ cid), RS\_num$$

# CHash - Maglev 一致性哈希

建立一个固定长度  $M$  的槽位查找表，对输入 key 哈希取余就可以映射到一个槽位

$$RS\_idx = \text{Hash}(key) \% M$$

计算查找表需要为每个槽位生成一个偏好序列 Permutation，按照偏好序列中数字的顺序，每个槽位轮流填充查找表。如果填充的目标位置已被占用，则顺延该序列的下一个目标位置填充

Permutations of B0, B1, B2

	B0	B1	B2
0	3	0	3
1	0	2	4
2	4	4	5
3	1	6	6
4	5	1	0
5	2	3	1
6	6	5	2

Lookup table

0	B1
1	B0
2	B1
3	B0
4	B2
5	B2
6	B0

# CHash - Maglev 一致性哈希

由于存储了偏好序列表，槽位的变动对查找表的影响就是可控的了

Table 1: A sample consistent hash lookup table.

	B0	B1	B2
0	3	0	3
1	0	2	4
2	4	4	5
3	1	6	6
4	5	1	0
5	2	3	1
6	6	5	2

Permutation tables for the backends.

	Before	After
0	B1	B0
1	B0	B0
2	B1	B0
3	B0	B0
4	B2	B2
5	B2	B2
6	B0	B2

Lookup table before and after B1 is removed.

生成偏好序列只需要保证其随机和均匀。查找表建立时间  $O(M \log M)$ , 最坏  $O(M^2)$

## Pseudocode 1 Populate Maglev hashing lookup table.

```
1: function POPULATE
2:   for each  $i < N$  do  $next[i] \leftarrow 0$  end for
3:   for each  $j < M$  do  $entry[j] \leftarrow -1$  end for
4:    $n \leftarrow 0$ 
5:   while true do
6:     for each  $i < N$  do
7:        $c \leftarrow permutation[i][next[i]]$ 
8:       while  $entry[c] \geq 0$  do
9:          $next[i] \leftarrow next[i] + 1$ 
10:         $c \leftarrow permutation[i][next[i]]$ 
11:      end while
12:       $entry[c] \leftarrow i$ 
13:       $next[i] \leftarrow next[i] + 1$ 
14:       $n \leftarrow n + 1$ 
15:    if  $n = M$  then return end if
16:   end for
17: end while
18: end function
```

# CHash - Maglev 一致性哈希

查找表的长度  $M$  应是一个质数，这样可以减少哈希碰撞和聚集，让分布更均匀

如果想要实现带权重的 Maglev 哈希，可以通过改变槽位间填表的相对频率来实现加权

「随机生成偏好序列 permutation」

Google 方法是，取两个无关的哈希函数  $h1$   $h2$ ，给槽位  $b$  生成时，先用哈希计算 offset 和 skip

$$\begin{aligned}offset &= h1(b)\%M \\skip &= h2(b)\%(M - 1) + 1\end{aligned}$$

然后对每个  $j$ ，计算

$$permutation[j] = (offset + j * skip)\%M$$

这里通过类似二次哈希的方法，使用两个独立无关的哈希函数来减少映射结果的碰撞次数，提高随机性。但是这要求  $M$  必须是质数，才能保证与  $skip$  互质，最终遍历完整个  $M$

# CHash - Maglev 一致性哈希

Google 论文中说核心关注的两个问题是：

1. load balancing: each backend will receive an almost equal number of connections.
2. minimal disruption: when the set of backends changes, a connection will likely be sent to the same backend as it was before

其中第一个是最为关键的，同时 Maglev 每个 VIP 绑定到后端的几百个服务器，每个都需要很大的 lookup table。并且，尽管想要最小化一致性哈希在扩缩容场景下的变化，但因为有 connections' affinity 亲和性 (conntrack)（且连接复用的 reset 也是被允许的），少量的槽位增删干扰也是可以接受的

优势：O(1) 的调度查找、均匀且一致性的哈希映射和可以增加权重的影响

问题：

1. 需要额外存储每个槽位的偏好序列和槽位查找表 (rs\_buf)
2. 虽然避免了全局重新映射，但是没有做到最小化的重新映射
3. Google 的测试里，65537 大小的查找表生成时间为 1.8ms，655373 大小的查找表生成时间为 22.9ms

# CHash - Maglev 一致性哈希

A	B	C	Entry	→	B	C	Entry
0	0	1	A		0	1	B
1	2		B		1	2	C
	2		C		2		B

# CHash - Maglev 一致性哈希

A	B	C	Entry	→	B	C	Entry
0	0	1	A		0	1	B
3	1	2	B		1	2	C
	2		C		2		B
3	4	A			3	4	B
4	5	B			4	5	C
5		C			5		B

$$skip = h2(b)\%(M - 1) + 1$$

$$permutation[j] = (offset + j * skip)\%M$$

# CHash - Maglev 一致性哈希

A	B	C	Entry	→	B	C	Entry
0	0	1	A		0	1	B
3	1	2	B		1	2	C
2	3	C			2	3	B
3	4	A			3	4	C
4	5	B			4	5	B
5		C			5		C

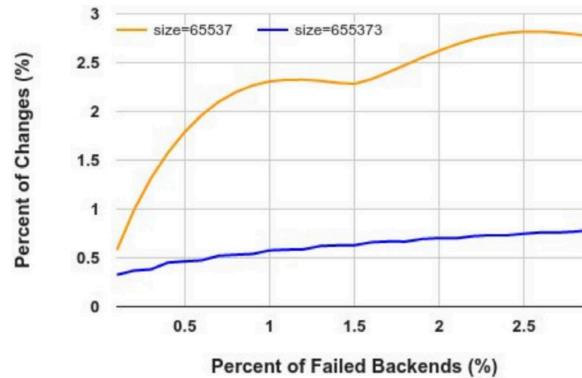
再考慮 weight 的填表频率呢?

# CHash - Maglev 一致性哈希

「槽位增删分析」

Experiments in Google Paper Section 5.3

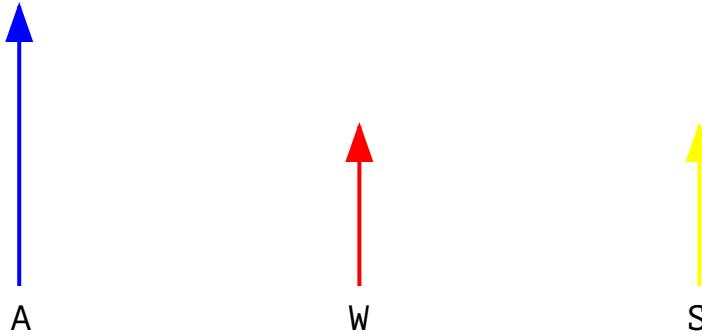
实验设置：1000 台后端服务器，对每个 k-failure 重新生成查找表并检查入口变化，重复 200 次取平均值



按照实验结果， $M = 65537$ ,  $k = 5$  时，只有约 1180 个入口会变化 (约 1.8%)

# CHash - AnchorHash 一致性哈希

池化 + 标记的思想，通过复用来减少重映射



A: fixed size bucket list

W: working list (mapping to resource list (S) 1-1)

$$h = \begin{cases} h_1(k) \equiv \text{hash}(k) \pmod{|\mathcal{A}|} & \text{if } h_1(k) < |\mathcal{W}|, \\ h_2(k) \equiv \text{hash}'(k) \pmod{|\mathcal{W}|} & \text{otherwise,} \end{cases}$$

# CHash - AnchorHash 一致性哈希

想解决槽位增删时的高迁移成本和平衡性下降问题

预先定义固定大小  $a$ （预期可能达到的最大节点规模）的虚拟节点集合为锚点集，工作节点是锚点集的子集

当对 key 分配时，

1. 先用  $H_1(key)$  映射到锚点集的一个桶  $b$
2. 如果桶  $b$  是工作节点，则直接分配
3. 否则启动回填过程
  1. 用另一个哈希  $H_2(key)$  计算一个起始点
  2. 在锚点集按顺序查找下一个工作节点的桶  $b'$

设计了一个 next 数组用来表示当前桶不可用时应该从哪找下一个候选桶，在节点增删时维护 next 指针

# CHash - AnchorHash 一致性哈希 - Removal

	0	1	2	3	4	5	6
$\mathcal{W}$	✓	✓	✓	✓	✓	✓	✓
$\mathcal{W}_b$							

(a) Initial working set  $\mathcal{W} = \{0, 1, 2, 3, 4, 5, 6\}$ .

	0	1	2	3	4	5	6
$\mathcal{W}$	✓	✓	✓	✓	✓	✓	✗
$\mathcal{W}_b$							$\{0, 1, 2, 3, 4, 5\}$

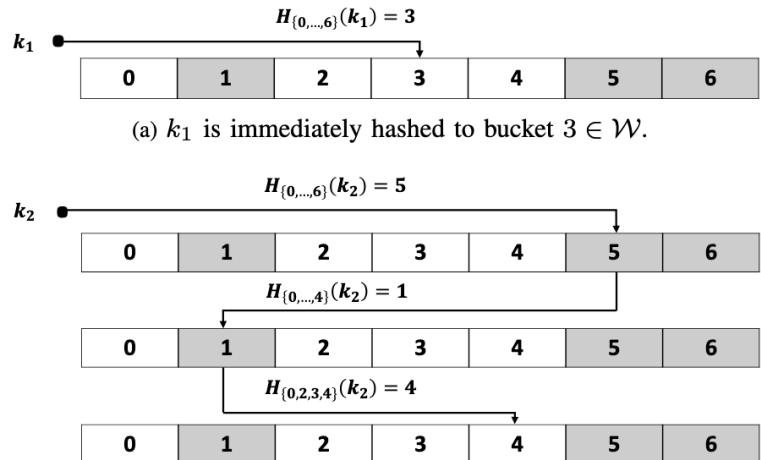
(b) Removing bucket 6 with  $\mathcal{W}_6 = \{0, 1, 2, 3, 4, 5\}$ .

	0	1	2	3	4	5	6
$\mathcal{W}$	✓	✓	✓	✓	✓	✗	✗
$\mathcal{W}_b$							$\{0, 1, 2, 3, 4\}$ $\{0, 1, 2, 3, 4, 5\}$

(c) Removing bucket 5 with  $\mathcal{W}_5 = \{0, 1, 2, 3, 4\}$ .

	0	1	2	3	4	5	6
$\mathcal{W}$	✓	✗	✓	✓	✓	✗	✗
$\mathcal{W}_b$		$\{0, 2, 3, 4\}$				$\{0, 1, 2, 3, 4\}$	$\{0, 1, 2, 3, 4, 5\}$

(d) Removing bucket 1 with  $\mathcal{W}_1 = \{0, 2, 3, 4\}$ .



(b)  $k_2$  is initially hashed to bucket 5. Then, since  $5 \notin \mathcal{W}$ , following Fig. 2(d),  $k_2$  is rehashed into the set  $\{0, \dots, 4\}$ . Assume that it is rehashed to 1  $\in \mathcal{W}_5$ . Since  $1 \notin \mathcal{W}$  as well,  $k_2$  is rehashed again to 4  $\in \mathcal{W}_1$ . Since  $4 \in \mathcal{W}$ , the process terminates.

Fig. 3: Example of possible key lookups in the state presented in Fig. 2(d).

# CHash - AnchorHash 一致性哈希

---

**Algorithm 1 — AnchorHash**


---

```

1: function INITANCHOR( $\mathcal{A}, \mathcal{W}$ )
2:    $\mathcal{R} \leftarrow \emptyset$ 
3:   for  $b \in \mathcal{A} \setminus \mathcal{W}$  do
4:      $\mathcal{R}.push(b)$ 
5:      $\mathcal{W}_b \leftarrow \mathcal{A} \setminus \mathcal{R}$ 
6:
7: function GETBUCKET( $k$ )
8:    $b \leftarrow H_{\mathcal{A}}(k)$ 
9:   while  $b \notin \mathcal{W}$  do
10:     $b \leftarrow H_{\mathcal{W}_b}(k)$ 
11:   return  $b$ 
12:
13: function ADDBUCKET( )
14:    $b \leftarrow \mathcal{R}.pop()$ 
15:   delete  $\mathcal{W}_b$ 
16:    $\mathcal{W} \leftarrow \mathcal{W} \cup \{b\}$ 
17:   return  $b$ 
18:
19: function REMOVEBUCKET( $b$ )
20:    $\mathcal{W} \leftarrow \mathcal{W} \setminus \{b\}$ 
21:    $\mathcal{W}_b \leftarrow \mathcal{W}$ 
22:    $\mathcal{R}.push(b)$ 

```

---



---

**Algorithm 2 — AnchorHash Wrapper**


---

```

1: function INITWRAPPER( $\mathcal{A}, \mathcal{S}$ )
2:    $M \leftarrow \emptyset, \mathcal{W} \leftarrow \emptyset$ 
3:   for  $i \in (0, 1, \dots, |\mathcal{S}| - 1)$  do
4:      $M \leftarrow M \cup \{(\mathcal{A}[i], \mathcal{S}[i])\}$ 
5:      $\mathcal{W} \leftarrow \mathcal{W} \cup \{\mathcal{A}[i]\}$ 
6:   INITANCHOR( $\mathcal{A}, \mathcal{W}$ )
7:
8: function GETRESOURCE( $k$ )           ▷ Compute resource for key k
9:    $b \leftarrow GETBUCKET(k)$ 
10:   $\xi \leftarrow M(b)$ 
11:  return  $\xi$ 
12:
13: function ADDRESOURCE( $\xi$ )
14:    $b \leftarrow ADDBUCKET()$ 
15:    $M \leftarrow M \cup \{(b, \xi)\}$ 
16:
17: function REMOVERESOURCE( $\xi$ )
18:    $b \leftarrow M^{-1}(\xi)$ 
19:    $M \leftarrow M \setminus \{(b, \xi)\}$ 
20:   REMOVEBUCKET( $b$ )

```

---

M: Mapper from anchor A to rs S, R: Removed label stack

$$1 + \ln\left(\frac{a}{w}\right) \text{ on } GetBucket$$

# CHash - AnchorHash 一致性哈希

优势：

1. 删除节点时，只影响直接映射到该节点桶以及通过回填路径依赖该桶的 key
2. 存储固定大小的数组记录，不需要虚拟节点
3. 查找通常只需要一次初始哈希和平均不到一次回填跳转
4. 不会因为其他节点的增删而影响当前节点的映射 key
5. 负载平衡性高较均匀，键分配到工作节点的方差较低

问题：

1. 额外的内存开销来存储三个  $O(a)$  数组 workers, removed, next
2. 新增节点只接收映射到它自身桶的新键和未来因其他节点离开而回填的键，不会立即分担现有节点的负载
3. 需要维护 next 指针的逻辑较为复杂

# CHash 算法对比

	Memory usage	Lookup time	Init time	Resize time
Ring	$\Theta(VW)$	$O(\log_2(VW))$	$O(VW\log_2(VW))$	$O(V\log_2(VW))$
Rendezvous	$\Theta(W)$	$\Theta(W)$	$\Theta(W)$	$\Theta(1)$
Jump	$\Theta(1)$	$O(\ln(W))$	$\Theta(1)$	$\Theta(1)$
Multi-probe	$\Theta(W)$	$O(P\log_2(W))$	$O(W\log_2(W))$	$O(\log_2(W))$
Maglev	$\Theta(MW)$	$\Theta(1)$	$\Theta(MW)$	$\Theta(MW)$
Anchor	$\Theta(A)$	$O(\ln(\frac{A}{W})^2)$	$\Theta(A)$	$\Theta(1)$
Dx	$\Theta(A)$	$O(\frac{A}{W})$	$\Theta(A)$	$\Theta(1)$

- $V$ : Ring 中每个物理节点对应的虚拟节点个数
- $W$ : 集群中的 working 节点数目
- $P$ : Multi-probe 中的探针(哈希)个数
- $M$ : Maglev 中每个节点在查询表中的位置数
- $A$ : 集群中的所有节点数(包括 working 和非 working 的节点)

# CHash 算法对比

Shards	Ketama	CHash	MultiProbe	Jump	Rendezvous	Maglev
8	279	115	121	55.2	34.6	30.8
16	282	122	131	55.6	45.1	32.2
32	300	131	132	55.9	67.3	32.4
64	323	149	160	58.8	113	32.4
128	361	171	279	72.6	210	34.1
256	419	208	336	84.2	402	38.7
512	467	241	374	91.9	787	41.8
1024	487	283	381	99	1546	
2048	538	326	421	102	2991	
4096	606	372	443	107	5980	
8192	1060	504	481	112	11953	

不同节点数量时单次查询的时间（纳秒）

# CHash 算法对比 - 分布均匀性测试

测试条件：100个节点，100,000个键

算法	标准差	评价
跳跃一致性哈希	25.34	最优
直接哈希取模	29.18	优秀
Rendezvous哈希	32.13	良好
Maglev哈希	35.74	良好
Multi-Probe CH( $k=21$ )	144.83	一般
Multi-Probe CH( $k=5$ )	362.18	较差
DxHash	543.58	较差
哈希环	83.59	一般
AnchorHash	8954.69	很差

分布均匀性测试衡量的是当大量键被分配到固定数量的节点时，每个节点被分配到的键数量是否接近平均值。我们使用标准差来衡量分布均匀性：

- 标准差越小：表示各节点被分配到的键数量越接近平均值，分布越均匀
- 标准差越大：表示某些节点被分配到的键数量远高于或低于平均值，分布不均匀

# CHash 算法对比 - 查询性能测试

测试条件：1000个节点，100,000个键

算法	查询时间	评价
Maglev哈希	12.42ms	最优
直接哈希取模	10.60ms	优秀
跳跃一致性哈希	17.39ms	优秀
哈希环	23.27ms	良好
AnchorHash	45.39ms	良好
DxHash	103.29ms	一般
Rendezvous哈希	11.51s	很差
Multi-Probe CH(k=5)	1m1.77s	极差
Multi-Probe CH(k=21)	4m22.96s	极差

性能测试衡量的是算法处理单个键查询的速度

时间为 100,000 个键查询总时间

# CHash 算法对比 - 添加节点时的重映射测试

测试条件：1000个初始节点，新增10个节点，100,000个键

算法	重映射键数	重映射比例	评价
AnchorHash	466	0.47%	最优
跳跃一致性哈希	969	0.97%	优秀
Rendezvous哈希	977	0.98%	优秀
Multi-Probe CH(k=21)	989	0.99%	优秀
Multi-Probe CH(k=5)	1003	1.00%	优秀
哈希环	1078	1.08%	良好
DxHash	1194	1.19%	良好
Maglev哈希	3418	3.42%	一般
直接哈希取模	98971	98.97%	很差

重映射测试衡量的是当系统中增加新节点时，有多少比例的键需要重新分配到不同的节点：

- 重映射比例越低：表示系统扩展时对现有服务的影响越小
- 重映射比例越高：表示系统扩展时对现有服务的影响越大

# CHash 算法对比 - 基准测试结果

测试条件：1000个节点

算法	每次操作时间	每次操作内存分配	评价
Maglev哈希	128.2 ns/op	15 B/op (1 allocs/op)	最优
直接哈希取模	112.8 ns/op	0 B/op (0 allocs/op)	优秀
跳跃一致性哈希	200.1 ns/op	0 B/op (0 allocs/op)	优秀
哈希环	223.1 ns/op	0 B/op (0 allocs/op)	良好
AnchorHash	416.0 ns/op	15 B/op (1 allocs/op)	良好
DxHash	959.2 ns/op	191 B/op (9 allocs/op)	一般
Rendezvous哈希	113918 ns/op	0 B/op (0 allocs/op)	很差
Multi-Probe CH(k=5)	624716 ns/op	76059 B/op (5005 allocs/op)	极差
Multi-Probe CH(k=21)	2571089 ns/op	319368 B/op (21021 allocs/op)	极差

# CHash 算法对比 - 性能综合对比

算法	查询性能	分布均匀性	重映射比例	内存占用
直接哈希取模	最好	一般	最高	最低
跳跃一致性哈希	好	最好	最低	最低
Maglev哈希	最好	好	一般	高
Rendezvous哈希	最差	好	低	最低
哈希环	一般	差	一般	低
AnchorHash	一般	最差	最低	中等
DxHash	一般	差	一般	中等
Multi-Probe CH( $k=5$ )	差	较差	一般	高
Multi-Probe CH( $k=21$ )	最差	一般	一般	最高

1. 查询性能优先：选择直接哈希取模或Maglev哈希
2. 最优分布均匀性：选择跳跃一致性哈希
3. 最小化重映射：选择AnchorHash或跳跃一致性哈希
4. 平衡各方面需求：选择哈希环算法，并适当调整虚拟节点数
5. 对重映射极其敏感：选择AnchorHash

# 厂商产品实现中的 LB 算法

- DGW
  - WRR
  - minConn
  - srcIP hash
  - QUIC cid hash
- Huawei
  - WRR
  - W-minConn
  - CHash (srcIP, QUIC ID)
- 美团 MGW
  - 哈希环 srcIP
- 爱奇艺 DPVS
- Nginx (L7)
- Bilibili (L7)
- Cloudflare Unimog
- GitHub GLB
- Google Maglev
- Aliyun
  - RR
  - WRR
  - W-minConn
  - CHash (srcIP, 4-tuple, QUIC ID)

支持全调度算法的会话保持

只有 CHash 算法本身提供了会话保持能力

虽然 NLB 和 CLB 支持 QUIC ID 哈希，但仅支持 Q10、Q29 版本

# DPVS

- fo(weighted fail over) `dpvs/src/ipvs/ip_vs_fo.c`
  - 返回的是 !overload && avail && highest weight 的 RS
- rr `dpvs/src/ipvs/ip_vs_rr.c`
- wrr `dpvs/src/ipvs/ip_vs_wrr.c`
- wlc `dpvs/src/ipvs/ip_vs_wlc.c`
  - 计算所有 RS 的负载情况 (dest overhead) / dest->weight
  - `dest overhead = dest->actconns << 8 + dest->inactconns`
- conhash(带权重哈希环) `dpvs/src/ipvs/ip_vs_conhash.c`
  - $\text{weight} / \text{weight\_gcd} * \text{REPLICAS}(160)$
- mh(Maglev) `dpvs/src/ipvs/ip_vs_mh.c`
  - `dp_vs_mh_populate` 填充用不变的 offset | skip | turns (weight / gcd), 变的 perm(next offset)
  - `dp_vs_mh_get_fallback` if selected server is unavailable, loop around the table starting from idx to find a new dest

# Nginx

- WRR 默认和兜底，其他调度算法一直尝试失败会改为默认算法
  - nginx/src/http/ngx\_http\_upstream\_round\_robin.c : ngx\_http\_upstream\_get\_peer
- least-connect (W-minConn)
  - nginx/src/http/modules/ngx\_http\_upstream\_least\_conn\_module.c :  
    ngx\_http\_upstream\_get\_least\_conn\_peer
- W-ip-hash (srcIP)
  - nginx/src/http/modules/ngx\_http\_upstream\_ip\_hash\_module.c : ngx\_http\_upstream\_get\_ip\_hash\_peer
- W-chash 哈希环
  - nginx/src/http/modules/ngx\_http\_upstream\_hash\_module.c : ngx\_http\_upstream\_update\_chash
- random
  - nginx/src/http/modules/ngx\_http\_upstream\_random\_module.c : ngx\_http\_upstream\_peek\_random\_peer
- Fair Queueing 公平队列：根据后端节点服务器的响应时间来分配请求

{ 注意，文档中说只有 ip-hash 有会话保持 }

# Nginx

## 「W-hash」

- 哈希桶实现。创建了 `weight * 160` 个影子节点，根据哈希值排序

## 「ip-hash」

- 对 IP 地址哈希后取模 RS 总权重，然后依次减掉每一个 RS 的权重看落在哪个 RS 里，从而将权重作为影响因素放入直接哈希

## 「least-connect」

- 顺序遍历  $O(N)$  查找连接数最小的 RS
- 出现相同的最小连接数时，回退到 SWRR 进行选择
- 比较连接数时，因为是遍历的两两比较，可以通过 `peer->conns * best->weight < best->conns * peer->weight` 来考虑权重

# Nginx - SWRR

```
nginx/src/http/ngx_http_upstream_round_robin.c : ngx_http_upstream_get_peer
```

```
for (peer = rrp->peers->peer, i = 0; peer; peer = peer->next, i++) {
    if (peer->down) { continue; }

    if (peer->maxfails && peer->fails >= peer->maxfails && now - peer->checked <= peer->fail_timeout) { continue; }

    if (peer->maxconns && peer->conns >= peer->maxconns) { continue; }

    peer->current_weight += peer->effective_weight;
    total += peer->effective_weight;

    if (peer->effective_weight < peer->weight) {
        peer->effective_weight++;
    }

    if (best == NULL || peer->current_weight > best->current_weight) {
        best = peer;
        p = i;
    }
}
best->current_weight -= total;
return best;
```

# Nginx - SWRR

```
if (peer->effective_weight < peer->weight) {  
    peer->effective_weight++;  
}
```

- weight: 配置文件中的权重
- effective\_weight: 动态的有效权重, 初始化为 weight。当和 RS 通信发生错误时减小, 后续逐步恢复

「扩展思考」

在系统初始时, 通过将所有 effective\_weight 初始化为 1 来放大随机效果

选 best 时可以添加随机数判断 `peer->current_weight == best->current_weight` 时是否要替换 `best = peer`

但是参考 `ngx_http_upstream_init_round_robin` 初始化中的赋值, 都会有 `peer.weight = peer.effective_weight = server.weight`

# SWRR - Start from 1 & Random Choice

weight start from 1 & random choice

```
peer.weight = server.weight;
peer.effective_weight = 1;
random(tid) // different in WORKERs
for all RS {
    peer->current_weight += peer->effective_weight;
    total += peer->effective_weight;

    if (peer.effective_weight < peer.weight) {
        peer.effective_weight++;
    }

    if (best == NULL
        || peer->current_weight > best->current_weight
        || (peer->current_weight == best->current_weight && random.next() > THRESHOLD))
    {
        best = peer;
    }
}
best->current_weight -= total;
return best;
```

# SWRR - Start from 1 & Random Choice

Server	eff	cur	eff	cur	cur	eff	cur	cur	eff	cur	cur
A(2)	1	1	2	-2	0	2	0	2	2	2	4
B(3)	1	1	2	1	3	3	-3	0	3	0	3
C(4)	1	1	2	1	3	3	3	6	4	-2	2
total		3			6			8			9

Select: A B C, before current weight 4 3 2

# SWRR - Start from 1 & Random Choice

Server

A(2)	4	-5	-3	-3	-1	-1	1	1	3	3	5	-4	-2	-2	0	0	2	2	4
B(3)	3	3	6	-3	0	0	3	3	6	-3	0	0	3	3	6	-3	0	0	3
C(4)	2	2	6	6	10	1	5	-4	0	0	4	4	8	-1	3	3	7	-2	2

Select: A B C C B A C B C

A(2), B(3), C(4)

# Nginx

为什么 Nginx 实现的所有的调度算法都没有想要空间换时间，全部都是  $O(N)$  的查找？

## 1. 动态权重和状态：

- 核心原因-权重可变性：在权重频繁调整（即使是偶尔）或服务器数量（N）很大的场景下，这个重新计算的成本会变得非常高，并且可能发生在处理请求的关键路径上，造成延迟抖动
- 状态感知算法：像 `least_conn`（最小连接数）和 `least_time`（最短响应时间）这类算法，其选择标准完全依赖于后端服务器的实时状态，无法预先计算出一个固定的调度序列

## 2. N 通常较小：

- 现实集群规模：在绝大多数实际部署中，一个 `upstream` 块包含的后端服务器数量 N 一般在几十台到一两百台的量级。 $O(1)$  带来节省有限，甚至可能因为哈希计算、哈希冲突处理等原因，其常数因子比简单的顺序遍历更大。顺序遍历对 CPU 缓存非常友好（线性访问内存）

# Nginx

为什么 Nginx 实现的所有的调度算法都没有想要空间换时间，全部都是  $O(N)$  的查找？

## 3. 内存效率：

- 预计算序列的内存开销：预生成完整的轮询序列（尤其是考虑权重时）需要额外的  $O(S)$  内存空间，其中  $S$  是所有服务器权重之和
- Nginx 的内存优化倾向：Nginx 以高性能和低内存消耗著称

## 4. 算法简单性与鲁棒性：

- 实现简单： $O(N)$  的遍历算法实现起来相对简单、清晰，代码易于理解和维护
- 鲁棒性好：简单的顺序遍历对后端服务器的动态变化（增、删、权重改、状态变）响应直接且自然。预计算序列在动态变化时需要复杂的更新或失效机制，容易引入边界条件错误

# Bilibili

LB 1.0

- WRR

问题:

1. 无法快速摘除有问题的节点
2. 无法均衡后端负载
3. 无法降低总体延迟

LB 2.0

- 动态感知的 WRR

方式:

1. 利用每次 RPC 请求返回的 Response 夹带 CPU 使用率
2. 每隔一段时间整体调整一次节点的权重分数

$$\text{peer.score} = \text{success\_rate} \div (\text{lantency} \times \text{cpuUsage})$$

问题:

1. 信息滞后和分布式带来的羊群效应 (VNSWRR 想解决的问题)

# Bilibili

## LB 3.0

- 带时间衰减的 Exponentially Weighted Moving Average 带系数的滑动平均值，实时更新延迟、成功率等信息，尽可能获取最新的信息
- 引入 best of two random choices 算法，加入随机性，在信息延迟较高的场景有效果
- 引入 inflight 作为参考，平衡坏节点流量，inflight 越高被调度到的机会越少

计算权重分数  $success \times metaWeight \div (cpu \times math.Sqrt(lag) \times (inflight + 1))$

success: 客户端成功率

metaWeight: 在服务发现中设置的权重

cpu: 服务端最近一段时间内的 cpu 使用率

lag: 请求延迟

inflight: 当前正在处理的请求数

# Bilibili

$$success \times metaWeight \div (cpu \times math.Sqrt(lag) \times (inflight + 1))$$

compute lag

```
// 获取&设置上次测算的时间点
stamp := atomic.SwapInt64(&pc.stamp, now)
// 获得时间间隔
td := now - stamp
// 获得时间衰减系数
w := math.Exp(float64(-td) / float64(tau))
// 获得延迟
lag := now - start
oldLag := atomic.LoadUint64(&pc.lag)
// 计算出平均延迟
lag = int64(float64(oldLag) * w + float64(lag) * (1.0 - w))
atomic.StoreUint64(&pc.lag, uint64(lag))
```

# Bilibili

$$success \times metaWeight \div (cpu \times math.Sqrt(lag) \times (inflight + 1))$$

best of two random choices

```
a := rand.Intn(len(p.conns))
b := rand.Intn(len(p.conns) - 1)
if b >= a {
    b = b + 1
}
nodeA := p.conns[a]
nodeB := p.conns[b]
if nodeA.load() * nodeB.health() * nodeB.Weight > nodeB.load() * nodeA.health() * nodeA.Weight {
    pick_node = nodeB
} else {
    pick_node = nodeA
}
```

Weight 是指配置的权重

# Cloudflare Unimog

自研硬件服务器，SDN 架构

可以根据 RS 的负载动态调整连接数量

机房内所有机器都安装了四层 LB，路由器无论把包发给哪台都会转发到正确的 RS 上

优势：

1. LB 不需要做容量规划
2. 最大限度防 DDoS
3. 运维架构简单，所有机器都一样

LB 之间不同步状态，连接保持方案类似 Maglev：所有 LB 自己决定，但保证对于相同的 4-tuple 会转发相同的 RS。所有 LB 都接收由控制平面统一下发的转发表，通过 4-tuple-hash 查表。转发表中每个 RS 可以设置多个 bucket（权重），RS 下线只修改对应 bucket

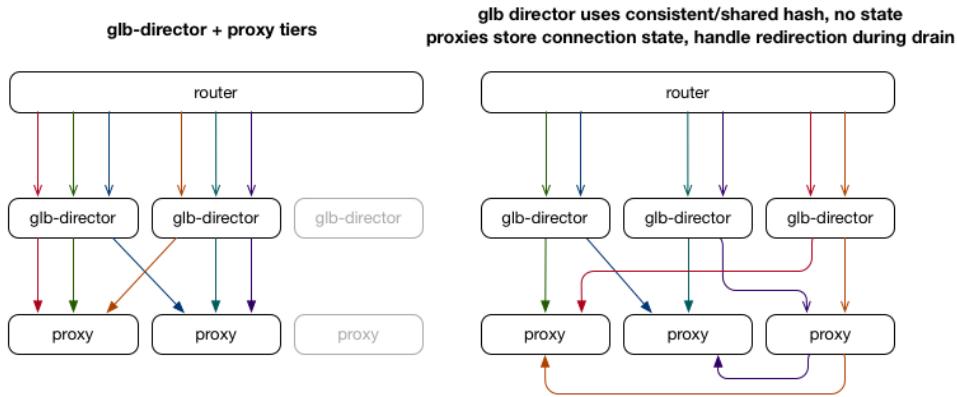
# Cloudflare Unimog

自研硬件服务器，SDN 架构

Before update		After update	
<i>First hop</i>	<i>Second hop</i>	<i>First hop</i>	<i>Second hop</i>
DIP of server A		DIP of server B	DIP of server A
DIP of server B		DIP of server B	
DIP of server C		DIP of server C	
DIP of server A		DIP of server C	DIP of server A
DIP of server B		DIP of server B	
DIP of server C		DIP of server C	
⋮		⋮	

由于没有 ct，它通过备份转发表 (second hop) 方式来继续先前的转发。当一个机器收到包时，先检查当前机器有没有这个 TCP socket，如果没有就转发到 second hop

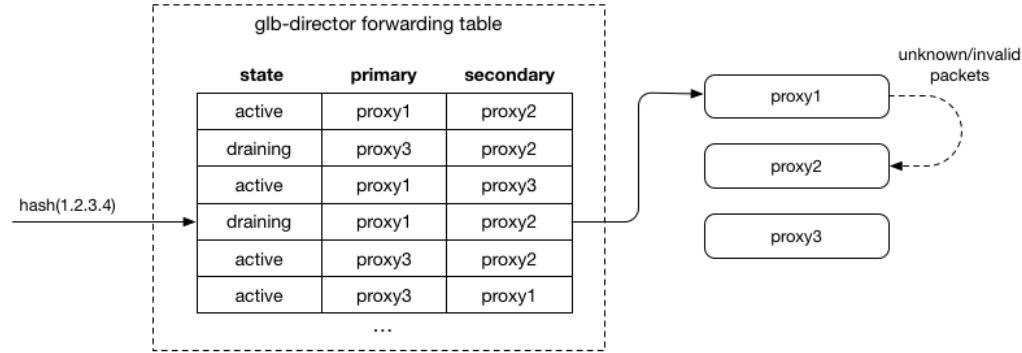
# GitHub GLB



「转发过程」

1. 根据 hash 查找转发表，找到对应的 2 个 RS，一个是主 RS 一个备 RS，然后转发到主 RS
2. 主 RS 收到包之后，检查这个包是不是属于自己机器上的连接，如果是，就交给协议栈处理，如果不是，就转发到备 RS（备 RS 的地址记录在 GLB 发过来的包中）

# GitHub GLB



「转发表」

- 在 RS 修改的时候，只有变化的 RS 在表中会修改。即不能对整个表完全重新 hash
- 表的生成不依赖外部的状态
- 每一行的两个 RS 不应该相同
- 所有 RS 在表中出现的次数应该是大致相同的（负载均衡）

# GitHub GLB

转发表的实现方式类似 Rendezvous hashing

对于每一行，将行号 + RS IP 进行 Hash 得到一个数字，作为“分数”，所有的 RS 在这一行按照分数排序，取前两名，作为主 RS 和备 RS 放到表中。然后按照以下的四个条件来分析：

- 如果添加 RS，那么只有新 RS 排名第一的相关的行需要修改，其他的行不会改变
- 生成这个表只会依赖 RS 的 IP
- 每一行的两个 RS 不可能相同，因为取的前两名
- Hash 算法可以保证每一个 IP 当第一名的概率是几乎一样的

注意：在想要删除 RS 的时候，要交换主 RS 和备 RS 的位置，这样，主 RS 换到备就不会有新连接了，等残留的连接都结束，就可以下线了；在添加 RS 的时候，每次只能添加一个，因为如果一次添加两个，那么这两个 RS 如果出现在同一行的第一名和第二名，之前的 RS 就会没来得及 drain 就没了，那么之前的 RS 的连接都会断掉

# GitHub GLB

转发表的实现方式类似 Rendezvous hashing

对于每一行，将行号 + RS IP 进行 Hash 得到一个数字，作为“分数”，所有的 RS 在这一行按照分数排序，取前两名，作为主 RS 和备 RS 放到表中。然后按照以下的四个条件来分析：

- 如果添加 RS，那么只有新 RS 排名第一的相关的行需要修改，其他的行不会改变
- 生成这个表只会依赖 RS 的 IP
- 每一行的两个 RS 不可能相同，因为取的前两名
- Hash 算法可以保证每一个 IP 当第一名的概率是几乎一样的

优势：

1. 提供第二选择
2. 不需要保存数据
3. GLB 实例可以和 RS 同时做变化

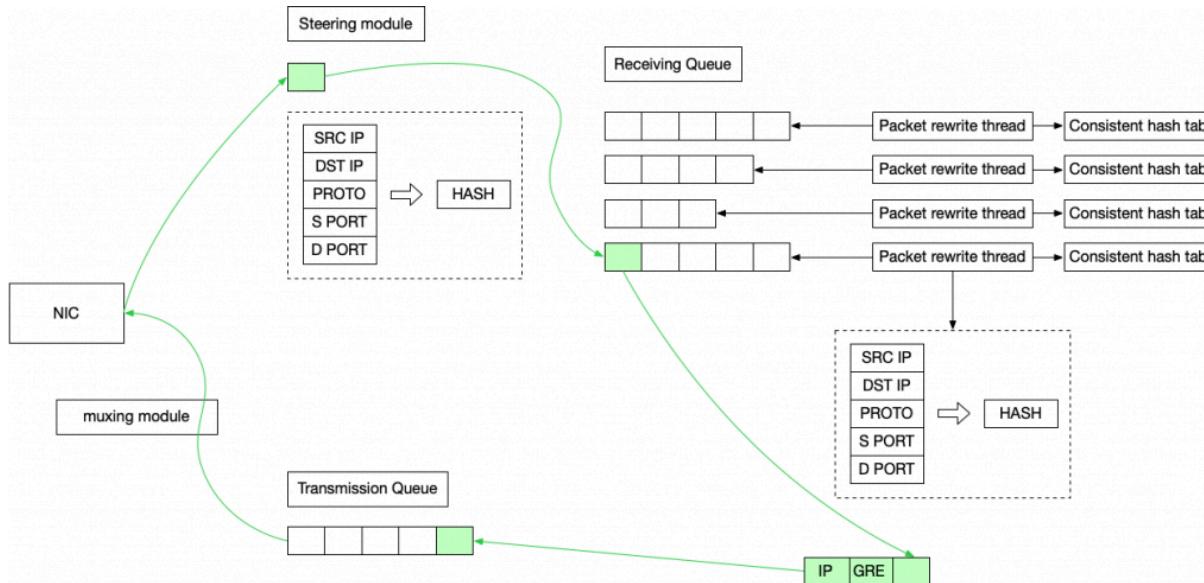
问题：

1. 要动 RS
2. 主备逻辑不是很清晰，RS 变动的影响还是很大

# Google Maglev

DSR 转发模式，Maglev 不会做 NAT

通过 GRE 将二层封装进 IP 包里（需要 MSS 预留空间）



上图是包的转发流程，绿色的是包经过的路径