一致性哈希 Consistant Hash

gaccob

2013 年 06 月 26 日

在动态变化的cache环境中，hash算法应该满足4个适应算法:

- balance，尽量做到负载均衡.
- monotonicity，当cache节点变化时，保护已分配的内容不重新mapping，当然，只能是尽量，因为没有办法完全做到，不然这个cache节点就没有存在的意义了.
- spread，当cache节点不是完全可见时，会导致相同的内容mapping到不同的节点中去，这就是spread，要尽量避免这种情况.
- load，是指每个cache节点的负载要尽量低.

大规模的分布式cache系统中, key-value如何做hash?

- 最常规的莫过于hash计算得到一个整数再取模.
- 在cache节点变化的时刻，会导致大量的cache不命中，需要重新建立mapping关系，这显然不是个好主意.

一致性哈希: Consistant Hash

1. Consistant Hash带来了什么? 在移除，添加一个cache时，它能够尽可能小的改变已存在key映射关系，尽可能的满足monotonicity的要求.
2. Consistant Hash的原理: 选择具体的cache节点不再只依赖于key的hash计算，cache节点本身也参与了hash计算. 参考文章: Consistant Hash and Random Trees.

3. 计算步骤:

   - 将整个哈希值空间组织成一个虚拟的圆环, 如假设某哈希函数H的值空间为$[0, 2^{32}-1]$, 即哈希值是一个32位无符号整形, 整个空间按顺时针方向组织, 0和$[2^{32}-1]$在零点中方向重合.
   - 对各个cache节点进行hash, key可以参考ip + 节点名, 并落在圆环上.
   - 对于数据key做相同的hash计算, 并确定在圆环上的位置, 从此位置顺时针“行走”, 遇到的第一个cache节点就是mapping到的节点.

4. 容错性与扩展性分析:

   - 某个cache节点宕机时, 只有该节点到它之面的第一个节点中间的数据受影响, 需要做remapping.
   - 当增加cache节点时, 也只有该节点和它之前的第一个节点中间的数据受影响, 需要做remapping.

5. 简单的C代码参考(因为是链表组织的, 所以效率一般, 只做简单参考):

```
typedef struct conhash_t
{
    struct list_head node_list;
    hash_func key_hash;
    hash_func node_hash;
} conhash_t;

#define CONHASH_NODE_NAME_SIZE 128

struct conhash_node_t
{
    struct list_head link;
    void* data;
    uint32_t hash_value;
};

void* conhash_node(conhash_t* ch, void* key)
{
    uint32_t val;
    struct conhash_node_t* n;
    struct list_head* l;
    if (!ch || !key) return NULL;
    if (list_empty(&ch->node_list)) return NULL;

    val = ch->key_hash(key);
```

```
26      list_for_each_entry(n, struct conhash_node_t, &ch->
            node_list, link) {
27          if (n->hash_value >= val) return n->data;
28      }
29      l = ch->node_list.next;
30      n = list_entry(l, struct conhash_node_t, link);
31      return n->data;
32  }
```