Ceph Crush

目录

**** 版本修订记录 ****

版本号	*修订时间*	*修订内容*
v1.0	*2018-08-07*	*初版修订*

一．简介

1.1 介绍

CRUSH(Controlled Replication Under Scalable Hashing)是一种伪随机数据分布算法，Ceph 集群通过 CRUSH 算法来确定数据的存储位置以及如何检索数据,根据每个设备的权重尽可能概率平均地分配数据。客户端的运用自身计算资源，给定一个输入 x 后，CRUSH 使用强大的多重整数 hash 函数根据集群 map,定位规则输出一个确定的有序的存储目标向量 R,客户端便可直接连接 OSD 进行数据的传输,而非通过一个中央服务器查表搜索，使得 Ceph 避免了单点故障，性能瓶颈和伸缩性的物理限制， CRUSH 主要存在一下几个优点:

- 任何组件都可以通过 CRUSH 算法计算出对象的存储位置
- 很少的元数据(cluster map)

CRUSH 需要使用集群的 CRUSH MAP,该 MAP 包括:

- OSD 列表
- BUCKET 列表
- 数据复制规则表

1.2 CRUSH 伪代码

Algorithm 1 CRUSH placement for object x

```
1: procedure TAKE(a)                          ▷ Put item a in working vector i
2:     i ← [a]
3: end procedure

4: procedure SELECT(n,t)                      ▷ Select n items of type t
5:     o ← ∅                                  ▷ Our output, initially empty
6:     for i ∈ i do                           ▷ Loop over input i
7:         f ← 0                              ▷ No failures yet
8:         for r ← 1,n do                     ▷ Loop over n replicas
9:             f_r ← 0                        ▷ No failures on this replica
10:            retry_descent ← false
11:            repeat
12:                b ← bucket(i)              ▷ Start descent at bucket i
13:                retry_bucket ← false
14:                repeat
15:                    if "first n" then      ▷ See Section 3.2.2
16:                        r' ← r + f
17:                    else
18:                        r' ← r + f_r n
19:                    end if
20:                    o ← b.c(r',x)          ▷ See Section 3.4
21:                    if type(o) ≠ t then
22:                        b ← bucket(o)      ▷ Continue descent
23:                        retry_bucket ← true
24:                    else if o ∈ o or failed(o) or overload(o,x)
   then
25:                        f_r ← f_r + 1, f ← f + 1
26:                        if o ∈ o and f_r < 3 then
27:                            retry_bucket ← true        ▷ Retry
   collisions locally (see Section 3.2.1)
28:                        else
29:                            retry_descent ← true       ▷ Otherwise
   retry descent from i
30:                        end if
31:                    end if
32:                until ¬retry_bucket
33:            until ¬retry_descent
34:            o ← [o,o]                      ▷ Add o to output o
35:        end for
36:    end for
37:    i ← o                                  ▷ Copy output back into i
38: end procedure

39: procedure EMIT                            ▷ Append working vector i to result
40:    R ← [R,i]
41: end procedure
```

Figure 1 crush 算法伪代码图

二．操作

2.1 获取 CRUSH MAP

```
获取 CRUSH 图
ceph osd getcrushmap –o crush.dump
```

```
反编译 CRUSH 图
crushtool  -d crush.dump –o crush.txt
```

2.2 CRUSH MAP 内容

CRUSH MAP 主要包括六个主要部分：

- Tunables:
- 设备：是 CRUSH MAP 树的叶子节点，对应了 OSD 磁盘
- Bucket 类型：定义了在 CRUSH MAP 的层次结构中所用到的 Bucket 类型
- Bucket 实例：该部分声明了 host 的 Bucket 实例，和其他可选的失效域
- Rules：定义了选择在数据存储和检索时选择 Bucket 的规则
- Choose_args:

```
# begin crush map
tunable choose_local_tries 0
tunable choose_local_fallback_tries 0
tunable choose_total_tries 50
tunable chooseleaf_descend_once 1
tunable chooseleaf_vary_r 1
tunable chooseleaf_stable 1
tunable straw_calc_version 1
tunable allowed_bucket_algs 54

# devices # OSD 列表
device 0 osd.0 class ssd # 格式为：device 设备 ID OSDID class 磁盘类型，是 CRUSH MAP 树的叶子节点,用于代表可以存储数据单个 OSD 进程。注意设备 ID 时非负整数，在 CRUSH MAP 中>=0 表示 OSD，<0 代表 Bucket;
device 2 osd.2 class ssd

# types # Bucket 类型定义
type 0 osd # 叶子 Bucket type 0,可以指定任何名字；0:类型 ID，osd:类型名称，可以修改为任意名字；一般编号较高的 Bucket 包含编号较低的 Bucket，并且 type 0 一般表示叶子节点；
type 1 host
type 2 chassis
type 3 rack
type 4 row
type 5 pdu
type 6 pod
type 7 room
type 8 datacenter
type 9 region
type 10 root

# buckets # Bucket 实例，Bucket 类型必须使用，这里的类型为 host
```

```
host mon {
	id -3		# do not change unnecessarily  # Bucket 的 ID 号
	id -4 class ssd		# do not change unnecessarily
	# weight 0.873
	alg straw2 # 表明该 Bucket 在选择低一级 Bucket 时所使用的算法，
	hash 0  # rjenkins1  # 每个 Bucket 使用的 hash 算法，hash 0 表示使用 rjenkins1 算法；
	item osd.2 weight 0.218 # item 声明了该 Bucket 所包含的低一级 Bucket，以及低一级 Bucket 的所有 item 的权重之和；另外，官方推荐使用 1.0 作为存储容量为 1T 的设备的权重，相应的权重为 0.5 代表 500G 的存储设备；
	item osd.3 weight 0.218
	item osd.4 weight 0.218
	item osd.0 weight 0.218
}
host osd {
	id -5		# do not change unnecessarily
	id -6 class ssd		# do not change unnecessarily
	# weight 0.873
	alg straw2
	hash 0  # rjenkins1
	item osd.5 weight 0.218
	item osd.6 weight 0.218
	item osd.7 weight 0.218
	item osd.8 weight 0.218
}
root default { # 默认的根 Bucket
	id -1		# do not change unnecessarily
	id -2 class ssd		# do not change unnecessarily
	# weight 1.746
	alg straw2
	hash 0  # rjenkins1
	item mon weight 0.873
	item osd weight 0.873
}

# rules # 定义了归置，分布或复制策略，可以为不同的 pool 设置不同的规则；
rule replicated_rule { # 默认的复制类型的 pool 的规则，replicated_rule 为规则名称；
	id 0  #规则 ID
	type replicated # pool 的类型
	min_size 1 # 定义 CRUSH 使用该规则时 PG 副本的最小值
	max_size 10 # 定义 CRUSH 使用该规则时 PG 副本的最大值
	step take default #选择 Bucket 名称，并迭代到 CRUSH MAP 树的底部，这里选择了名称为 default 的 root Bucket 为树的根开始；
	step chooseleaf firstn 0 type host # 选择 host 类型的 Bucket,0 表示选择数量为存储池的副本数；0<num<pool-num-replicas,表示就选择 num 个 Bucket,num<0 表示选择 pool-num-relicas-{num}Bucket;
	step emit # 输出选择结果并清空栈，通常应用于规则末尾；
```

```
}
rule erasure-code { # 默认的纠删码类型的 pool 的规则，erasure-code 为规则名称;
        id 1
        type erasure
        min_size 3
        max_size 3
        step set_chooseleaf_tries 5
        step set_choose_tries 100
        step take default
        step chooseleaf indep 0 type host
        step emit
}
```

Bucket 的使用的选择算法有如下几种:

- uniform:当所有的设备拥有相同的权重时可使用该类算法;
- list:该类型的 Bucket 把他们的内容汇聚为链表;
- tree:该类型的 Bucket 是一种二进制搜索树;
- straw:抽签类型的 Bucket,考虑权重;
- straw2:优化的抽签算法,

2.3 选择算法

各类算法的比较:

Action	uniform	list	tree	straw	straw2
算法复杂度	O(1)	O(n)	O(logn)	O(n)	O(n)
增加存储节点 删除存储节点	差 差	最优 差	好 好	最优 最优	最优 最优

- Bucket uniform

该类 Bucket 对应 mapper.c 文件中的函数:

```
static int bucket_perm_choose(const struct crush_bucket *bucket,
                              struct crush_work_bucket *work,
                              int x, int r)
```

该类别算法适用于所有子节点权重相同的情况，而且 bucket 很少增加或删除 item,这时的查询速度是最快的，因为该类 bucket 在选择子节点的时候不考虑权重问题，全部随机选择，所以权重并不会影响选择结果;

适用于子节点变化概率小的情况，在 size 发生变化时，perm 数组会完全重新排列，也就意味着保存在子节点的所有数据都要发生重排，造成数据迁移;

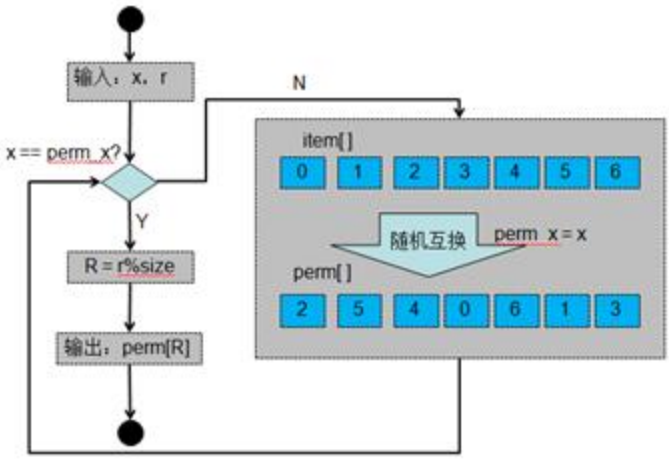

Figure 2 Crush Uniform 算法

先来说明一下 uniform 的要素。bucket 的所有子节点都保存在 item[]数组之中。perm_x 是记录这次随机排布时 x 的值，perm[]是在 perm_x 时候对 item 随机排列后的结果。r 则是选择第几个副本。

定位子节点过程。这时我们重新来看 uniform 定位子节点的过程。根据输入的 x 值判断是否为 perm_x，如果不是，则需要重新排列 perm[]数组，并且记录 perm_x=x。如果 x==perm_x 时，这时算 R = r%size，算后得到 R，最后返回 perm[R]

- Bucket list

该类 Bucket 对应 mapper.c 文件中的函数:
*static int bucket_list_choose(const struct crush_bucket_list *bucket,*
int x, int r)
该类算法适用于集群扩展类型，当增加 item 时，会产生最优的数据移动。缺点是只能顺序查找 item,时间复杂度为 O(n);

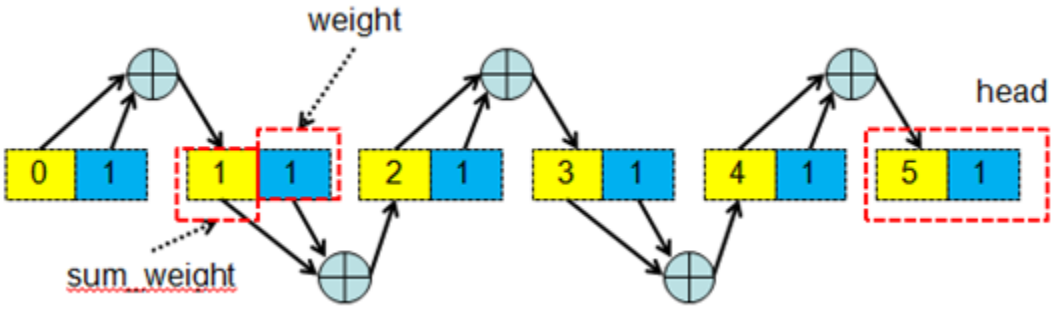

Figure 3 Crush List 数据结构

list bucket 定位数据在子节点的方法。从 head 开始，会逐个的查找子节点是否保存数据。如何判断当前子节点是否保存了数据呢？首先取了一个节点之后，根据 x，r 和 item 的 id 进行

crush_hash 得到一个 w 值。这个值与 sum_weight 之积，最后这个 w 再向右移 16 位，最后判断这个值与 weight 的大小，如果小于 weight 时，则选择当前的这个 item，否则进行查找下一个 item。

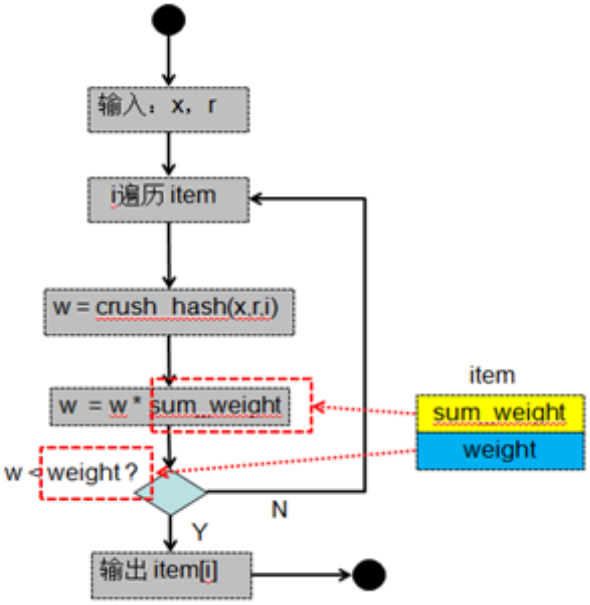

Figure 4 Crush List 算法流程

- Bucket tree

该类 Bucket 对应 mapper.c 文件中的函数：
*static int bucket_tree_choose(const struct crush_bucket_tree *bucket,*
int x, int r)
树状 Bucket 是一种加权二叉排序树，数据项位于树的叶子节点，每个递归节点有左右子树的总权重，并根据一种固定的算法进行标记。

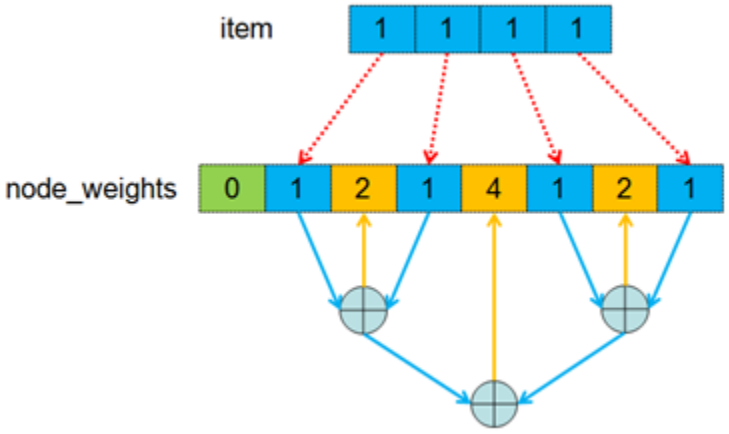

Figure 5 Crush Tree 数据结构图

tree bucket 会借助一个叫做 node_weight[ ]的数组来进行帮助搜索定位 item。首先是 node_weight[ ]的形成，nodeweight[ ]中不仅包含了 item，而且增加了很多中间节点，item 都作为叶子节点。父节点的重量等于左右子节点的重量之和，递归到根节点如下图。

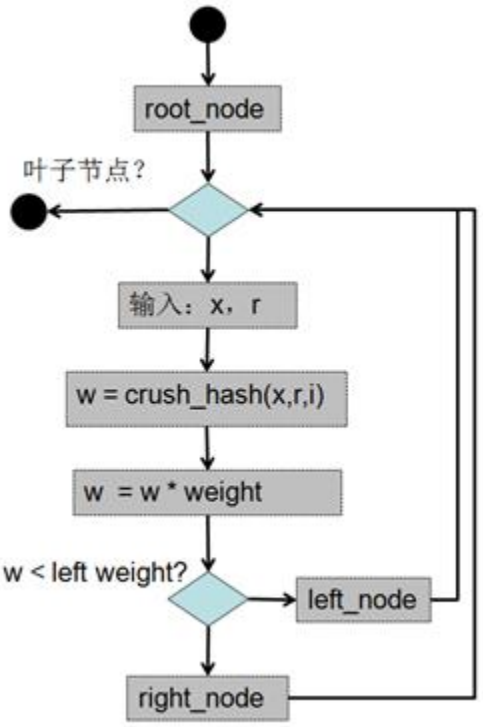

Figure 6 Crush Tree 算法流程图

- Bucket straw

该类 Bucket 对应 mapper.c 文件中的函数：
*static int bucket_straw_choose(const struct crush_bucket_straw *bucket,*
int x, int r)
straw 类型的 Bucket 允许所有 item 以抽签的形式来公平竞争，定位副本时，最长长度的 Bucket 会被选中。

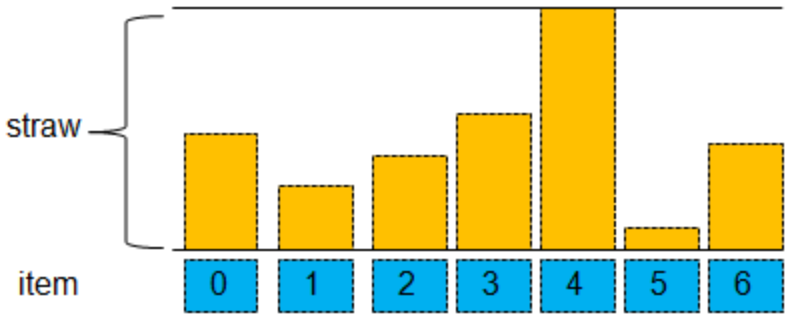

Figure 7 Crush Straw 数据结构图

- Bucket straw2

该类 Bucket 对应 mapper.c 文件中的函数:

```
static int bucket_straw2_choose(const struct crush_bucket_straw2 *bucket,
                                int x, int r, const struct crush_choose_arg *arg,
                                int position)
```

2.4 注入 CRUSH MAP

```
编译 CRUSH 图
crushtool –c crush.txt –o crush.dump
注入 CRUSH 图
ceph osd setcrushmap –i crush.dump
```

2.5 命令行

```
[cpu@mon ~]$ sudo ceph daemon mon.mon config show | grep osd_crush_update
   "osd_crush_update_on_start": "true", # osd 启动时 crush map 自动检测管理

ceph osd crush rule ls #查看定义的规则集
ceph osd crush rule dump #输出规则集内容
ceph osd crush tree # 输出 CRUSH MAP 的树状图
ceph osd crush rule ls #获取 CRUSH MAP 中的规则列表
ceph osd crush rule dump # 输出 CRUSH MAP 中的规则
ceph osd crush set-device-class <class> <osd-name> # 设置设备的类别，其中 class 有 hdd,ssd,nvme 等类型;
ceph osd crush rm-device-class <osd-name> # 移除设备上的类别，设备的类别只有移除后才能增加;
```

```
ceph osd crush rule create-replicated <rule-name> <root> <failure-domain> <class> #新增一个 replicated 规则
ceph osd crush rule create-erasure {name} {profile-name} # 新增一个 erasure 规则
ceph osd crush rule rm {rule-name} # 删除一个规则
ceph osd pool set <pool-name> crush_rule <rule-name> # 设置该 pool 在选择数据归置时所使用的规则
ceph osd crush tree --show-shadow # 查看设备的 shadow 类
ceph osd crush set {name} {weight} root={root} [{bucket-type}={bucket-type}…] # 增加 OSD
ceph osd crush reweight {name} {weight} # 修改设备的权重值
ceph osd crush remove {name} #删除 CRUSH MAP 中的 OSD
ceph osd crush add-bucket {bucket-name} {bucket-type} #增加一个 Bucket
ceph osd crush  move {bucket-name} {bucket-type}={bucket-name},[…] # 将 Bucket 移动到其他位置
ceph osd crush remove {bucket-name} # 删除一个 Bucket
```

2.6 源码解读

```
# 基础数据结构
struct crush_bucket {
        __s32 id;       /*!< bucket identifier, < 0 and unique within a crush_map */ Bucketd 的 ID
        __u16 type;     /*!< > 0 bucket type, defined by the caller */ Bucket 的类型对应的整形数字
        __u8 alg;       /*!< the item selection ::crush_algorithm */ Bucket 选择算法类型
        /*! @cond INTERNAL */
        __u8 hash;      /* which hash function to use, CRUSH_HASH_* */
        /*! @endcond */
        __u32 weight;   /*!< 16.16 fixed point cumulated children weight */
        __u32 size;     /*!< size of the __items__ array */
        __s32 *items;   /*!< array of children: < 0 are buckets, >= 0 items */
};
# bucket uniform
__u32 item_weight;

# bucket list
__u32 *item_weights;  /*!< 16.16 fixed point weight for each item */
__u32 *sum_weights;   /*!< 16.16 fixed point sum of the weights */

# bucket tree
__u8 num_nodes;
__u32 *node_weights;

# bucket straw
__u32 *item_weights;   /* 16-bit fixed point */
__u32 *straws;        /* 16-bit fixed point */

# bucket straw2
__u32 *item_weights;   /*!< 16.16 fixed point weight for each item */
```

三．自定义 CRUSH MAP

3.1 device 声明格式

```
# devices
device {num} {osd.name} [class {class}]
```

3.2 type 声明格式

```
#types
type {num} {bucket-name}
```

3.3 Bucket 声明格式

```
[bucket-type] [bucket-name] {
        id [a unique negative numeric ID]
        weight [the relative capacity/capability of the item(s)]
        alg [the bucket type: uniform | list | tree | straw  | straw2]
        hash [the hash type: 0 by default]
        item [item-name] weight [weight]
    }
```

3.4 Rule 声明格式

```
rule <rulename> {
        ruleset <ruleset>
        type [ replicated | erasure ]
        min_size <min-size>
        max_size <max-size>
        step take <bucket-name>
        step [choose|chooseleaf] [firstn| indep] <N> <bucket-type>
        step emit
}
```

3.5 案列

四．附录

4.1 bucket_perm_choose 源码

```
static int bucket_perm_choose(const struct crush_bucket *bucket,
                              struct crush_work_bucket *work,
                              int x, int r)
{
        unsigned int pr = r % bucket->size;
        unsigned int i, s;

        /* start a new permutation if @x has changed */
        if (work->perm_x != (__u32)x || work->perm_n == 0) {
                dprintk("bucket %d new x=%d\n", bucket->id, x);
                work->perm_x = x;

                /* optimize common r=0 case */
                if (pr == 0) {
                        s = crush_hash32_3(bucket->hash, x, bucket->id, 0) %
                                bucket->size;
                        work->perm[0] = s;
                        work->perm_n = 0xffff;   /* magic value, see below */
                        goto out;
                }

                for (i = 0; i < bucket->size; i++)
                        work->perm[i] = i;
                work->perm_n = 0;
        } else if (work->perm_n == 0xffff) {
                /* clean up after the r=0 case above */
                for (i = 1; i < bucket->size; i++)
                        work->perm[i] = i;
                work->perm[work->perm[0]] = 0;
                work->perm_n = 1;
        }

        /* calculate permutation up to pr */
        for (i = 0; i < work->perm_n; i++)
                dprintk(" perm_choose have %d: %d\n", i, work->perm[i]);
        while (work->perm_n <= pr) {
                unsigned int p = work->perm_n;
                /* no point in swapping the final entry */
                if (p < bucket->size - 1) {
                        i = crush_hash32_3(bucket->hash, x, bucket->id, p) %
                                (bucket->size - p);
                        if (i) {
                                unsigned int t = work->perm[p + i];
                                work->perm[p + i] = work->perm[p];
                                work->perm[p] = t;
                        }
```

```
                        dprintk(" perm_choose swap %d with %d\n", p, p+i);
                }
                work->perm_n++;
        }
        for (i = 0; i < bucket->size; i++)
                dprintk(" perm_choose  %d: %d\n", i, work->perm[i]);

        s = work->perm[pr];
out:
        dprintk(" perm_choose %d sz=%d x=%d r=%d (%d) s=%d\n", bucket->id,
                bucket->size, x, r, pr, s);
        return bucket->items[s];
}
```

4.2 bucket_tree_choose 源码

```
static int bucket_tree_choose(const struct crush_bucket_tree *bucket,
                              int x, int r)
{
        int n;
        __u32 w;
        __u64 t;

        /* start at root */
        n = bucket->num_nodes >> 1;

        while (!terminal(n)) {
                int l;
                /* pick point in [0, w) */
                w = bucket->node_weights[n];
                t = (__u64)crush_hash32_4(bucket->h.hash, x, n, r,
                                          bucket->h.id) * (__u64)w;
                t = t >> 32;

                /* descend to the left or right? */
                l = left(n);
                if (t < bucket->node_weights[l])
                        n = l;
                else
                        n = right(n);
        }

        return bucket->h.items[n >> 1];
}
```

4.3 bucket_straw_choose 源码

```
static int bucket_straw_choose(const struct crush_bucket_straw *bucket,
                               int x, int r)
{
        __u32 i;
        int high = 0;
        __u64 high_draw = 0;
        __u64 draw;

        for (i = 0; i < bucket->h.size; i++) {
                draw = crush_hash32_3(bucket->h.hash, x, bucket->h.items[i], r);
                draw &= 0xffff;
                draw *= bucket->straws[i];
                if (i == 0 || draw > high_draw) {
                        high = i;
                        high_draw = draw;
                }
        }
        return bucket->h.items[high];
}
```

4.4 bucket_straw2_choose 源码

```
static int bucket_straw2_choose(const struct crush_bucket_straw2 *bucket,
                                int x, int r, const struct crush_choose_arg *arg,
                  int position)
{
        unsigned int i, high = 0;
        __s64 draw, high_draw = 0;
    __u32 *weights = get_choose_arg_weights(bucket, arg, position);
    __s32 *ids = get_choose_arg_ids(bucket, arg);
        for (i = 0; i < bucket->h.size; i++) {
          dprintk("weight 0x%x item %d\n", weights[i], ids[i]);
                if (weights[i]) {
                        draw = generate_exponential_distribution(bucket->h.hash, x, ids[i], r,
weights[i]);
                } else {
                        draw = S64_MIN;
                }

                if (i == 0 || draw > high_draw) {
                        high = i;
                        high_draw = draw;
```

```
                }
        }

        return bucket->h.items[high];
}
```

五．源码分析

5.1 crush_do_rule

```
int crush_do_rule(const struct crush_map *map,
                  int ruleno, int x, int *result, int result_max,
                  const __u32 *weight, int weight_max,
                  void *cwin, const struct crush_choose_arg *choose_args)
```

- crush_map: crush 图
- ruleno: 规则 rule 的 ID 号
- x: hash 值输入
- result: 返回值的指针
- result_max: 返回值数据的最大大小
- weight: 叶子节点的权重向量
- weight_max: 权重向量的最大值
- cwin: 指向至少 map->working_size 字节内容的指针或 NULL
- choose_args:

(1) 判断如果 ruleno>max_rules，则报错(”bad ruleno $d”, map->max_rules)
(2) 通过 ruleno 获取规则(rule = map->rules[ruleno];)
(3) for 便利规则中的 step,每次便利获取一个 step(const struct crush_rule_step *curstep = &rule->steps[step];)
(4) switch 判断该 step 的操作类型
(5) a. CRUSH_RULE_TAKE:
判断 take 的参数是否符合要求，即当选择的设备时，其 id 大于 0 并且小于设备的最大数量，如果选择的时 bucket 时，其 id 要小于 0、小于最大的 bucket 数量并且相应的 bucket 要存在,将选择到的 bucket 保存在 w[0],wsize=1；否则报错：“bad take value $d “ curstep->arg1 (buckets[-1-curstep->arg1])

b. CRUSH_RULE_SET_CHOOSE_TRIES:
如果其参数值大于 0，则覆盖可调参数 tunable choose_total_tries 的值；

c. CRUSH_RULE_SET_CHOOSELEAF_TRIES:
如果其参数大于 0，则覆盖可调参数 tunable choose_leaf_tries 的值；

d. CRUSH_RULE_SET_CHOOSE_LOCAL_TRIES:
如果其参数大于 0，则覆盖可调参数 tunable choose_local_retries 的值；

e. CRUSH_RULE_SET_CHOOSE_LOCAL_FALLBACK_TRIES:
如果该参数大于 0，则覆盖可调参数 tunable choose_local_fallback_retries 的值；

f. CRUSH_RULE_SET_CHOOSELEAF_VARY_R
如果该参数大于 0，则覆盖可调参数 tunable chooseleaf_vary_r 的值；

g. CRUSH_RULE_SET_CHOOSELEAF_STABLE

```
                如果该参数大于 0，则覆盖可调参数 tunable chooseleaf_stable 的值；
        h. CRUSH_RULE_CHOOSELEAF_FIRSTN
          CRUSH_RULE_CHOOSE_FIRSTN
                设置变量 firstn = 1;并执行一下 i 部分的内容；
        i. CRUSH_RULE_CHOOSELEAF_INDEP
          CRUSH_RULE_CHOOSE_INDEP
                .如果 wsize==0，则跳出；
                .当类型为 CHOOSELEAF 时，实则变量 recurse_to_leaf 为 1；
                .初始化设置输出的大小 osize=0
                .对 w 数组进行 for 循环：
                        .设置需要返回的 item 数量，(numrep = curstep->arg1;)当其参数<=0 时，设
设置为(numrep += result_max;)
                        .获取 bucket 的 ID(bno = -1 - w[i];)；
                        .如果 ID<0 或大于最大的 bucket(map->max_buckets),则报错 continue("bad
w[i] %d", w[i])
                        .如果 firstn = 1:
                                .choose_leaf_tries>0 时，将该值赋值给 recurse_tries；或 map->
chooseleaf_descend_once>0 时，recurse_tries=1，或 choose_tries 复制给
recurse_tries,choose_tries 初始值为 map->choose_total_tries + 1
                                . 设置输出大小 osize += crush_choose_firstn()
                        .firsten!=0
                                . 设置 out_size 为 numrep 和 result_max-osize 的小的值；
                                . 调用 crush_choose_indep()函数；
                .如果 recurse_to_leaf = 1，也就是该 bucket 的 item 为叶子节点：
                        .则表示已经获取了 OSD 列表，将数据进行复制到输出列表；
                将算法结果的数据赋值给 w.并设置 wsize=osize(输出数组的大小)
        j. CRUSH_RULE_EMIT
                .该步骤代表输出结果集，将计算的结果复制给 result 参数,并设置
wsize=0,(result[result_len] = w[i])
        k. default
                .报错：unknown op %d at step %d
```

5.2 crush_choose_firstn

```
static int crush_choose_firstn(const struct crush_map *map,
                               struct crush_work *work,
                               const struct crush_bucket *bucket,
                               const __u32 *weight, int weight_max,
                               int x, int numrep, int type,
                               int *out, int outpos,
                               int out_size,
                               unsigned int tries,
                               unsigned int recurse_tries,
                               unsigned int local_retries,
```

```
                                unsigned int local_fallback_retries,
                                int recurse_to_leaf,
                                unsigned int vary_r,
                                unsigned int stable,
                                int *out2,
                                int parent_r,
                        const struct crush_choose_arg *choose_args)
```

- map: crush map 的数据；
- work:
- bucket: 选择的 bucket
- weight: 叶子节点的权重数组
- weight_max: 最大权重
- x: crush 算法的输入
- numrep: 该个 bucket 里面需要选出的 item 的数量
- type: 选择的 item 的类型
- out: 指向输出数组的指针
- outpos: 在输出数组中的位置
- out_size: 剩余的该输出的大小
- tries: 尝试的次数，choose_tries
- recurse_tries: recurse_tries
- local_retries: choose_local_retries
- local_fallback_retries: choose_local_fallback_retries
- recurse_to_leaf: 是否需要获取叶子节点
- vary_r: chooseleaf_vary_r，默认为 1
- stable: chooseleaf_stable 默认为 1;
- out2: 指向叶子节点数组的指针
- parent_r: 从父节点传过来的 vary_r;
- choose_args: 其他参数

(1)for 循环 numrep：

a.基本初始化，ftotal=0,skip_rep=0

b. while(retry_descent)循环：

.初始化 in 变量为 bucket;

.while(retry_bucket)循环；

.判断，如果输入的 bucket 的 item 的 size 为 0 则设置 reject=1 并跳转的 reject；

.判断条件 local_fallback_retries > 0 &&flocal >= (in->size>>1) &&flocal > local_fallback_retries 成立，调用函数：item = bucket_perm_choose(, work->work[-1-in->id],x, r);

.否则调用函数: item = crush_bucket_choose(in, work->work[-1-in->id], x, r,
(choose_args ? &choose_args[-1-in->id] : 0),
outpos);

.判断如果选中的 item 大于最大的设备 ID，则报错(bad item %d\n", item),设置 skip_rep = 1;并退出该层循环；

.通过 item 的 type 类型，如果类型不匹配，如果 item >= 0 || (-1-item) >= map->max_buckets，报错 bad item type，并设置 skip-rep = 1,退出该层循环；修改 in 值为 map->buckets[-1-item];设置 retry_bucket 为 1，跳转到下次循环；

.判断如果 item 已经在 out 数组中了，则 collide=1

.如果!collide && recurse_to_leaf，如果 item<0，表示选择到的为 bucket,需要从该 bucket 中选出叶子节点，否则表示选出的就是 device,out2[outpos] = item；

(2)返回 outpos

5.3 bucket_perm_choose

```
# bucket 随机排列组合
static int bucket_perm_choose(const struct crush_bucket *bucket,
                              struct crush_work_bucket *work,
                              int x, int r)
```

六.参考资料

【1】Ceph 的 CRUSH 算法 straw

【2】Ceph 源码解析：CRUSH 算法

【3】Ceph 的 CRUSH 算法原理

【4】ceph 的数据存储之路(3) ----- pg 选择 osd 的过程(crush 算法)