****

**Ceph Crush**

Contents

[**一．简介** 3](#_Toc521872726)

[1.1 介绍 3](#_Toc521872727)

[1.2 CRUSH伪代码 3](#_Toc521872728)

[**二．操作** 4](#_Toc521872729)

[2.1 获取CRUSH MAP 4](#_Toc521872730)

[2.2 CRUSH MAP内容 5](#_Toc521872731)

[2.3 选择算法 7](#_Toc521872732)

[2.4 注入CRUSH MAP 11](#_Toc521872733)

[2.5 命令行 11](#_Toc521872734)

[2.6 源码解读 12](#_Toc521872735)

[**三．自定义CRUSH MAP** 13](#_Toc521872736)

[3.1 device声明格式 13](#_Toc521872737)

[3.2 type声明格式 13](#_Toc521872738)

[3.3 Bucket声明格式 13](#_Toc521872739)

[3.4 Rule声明格式 13](#_Toc521872740)

[3.5 案列 13](#_Toc521872741)

[**四．附录** 13](#_Toc521872742)

[4.1 bucket\_perm\_choose源码 14](#_Toc521872743)

[4.2 bucket\_tree\_choose源码 15](#_Toc521872744)

[4.3 bucket\_straw\_choose源码 16](#_Toc521872745)

[4.4 bucket\_straw2\_choose源码 16](#_Toc521872746)

[**五.参考资料** 17](#_Toc521872747)

***\* 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* |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

***\* Release Copyleft©free \****

**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

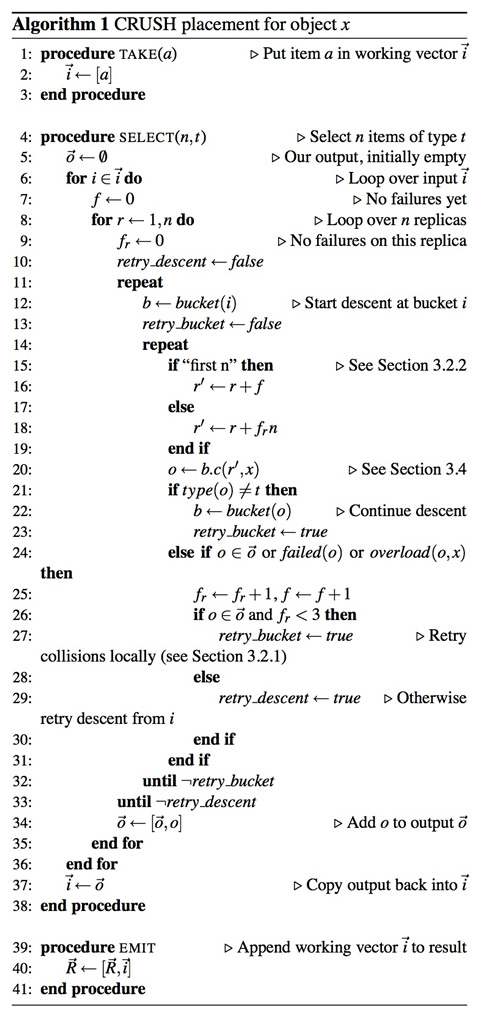


Figure 1 crush算法伪代码图

**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； |

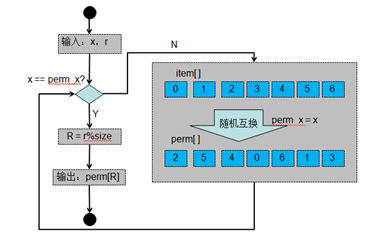


Figure 2 Crush Uniform算法

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); |

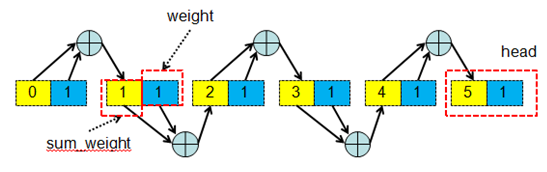


Figure 3 Crush List数据结构

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。

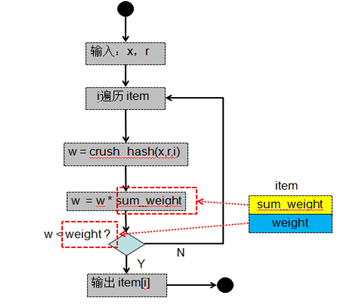


Figure 4 Crush List算法流程

* 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。 |

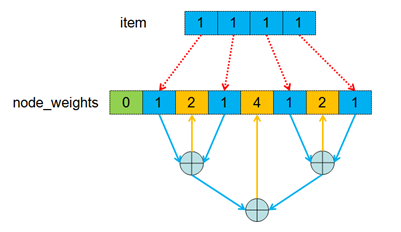


Figure 5 Crush Tree数据结构图

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。

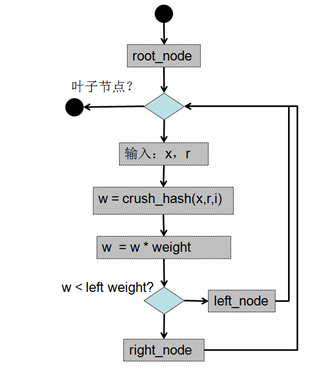


Figure 6 Crush Tree算法流程图

* 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。 |

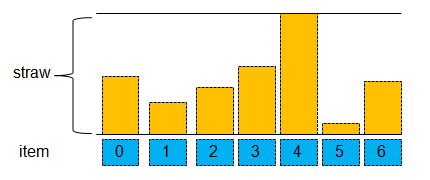


Figure 7 Crush Straw数据结构图

* 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](https://ceph.com/planet/ceph-%E7%9A%84crush%E7%AE%97%E6%B3%95-straw/)

【2】[Ceph源码解析：CRUSH算法](https://www.cnblogs.com/chenxianpao/p/5568207.html)

【3】[Ceph的CRUSH算法原理](https://www.oschina.net/translate/crush-controlled-scalable-decentralized-placement-of-replicated-data?lang=chs&page=1)

【4】[ceph的数据存储之路(3) ----- pg选择osd的过程(crush 算法)](https://my.oschina.net/u/2460844/blog/531722)