不洗碗工作室 Lv3
2018年09月05日 阅读 3582

一致性Hash在负载均衡中的应用

作者：不洗碗工作室 - Marklux

出处：Marklux's Pub

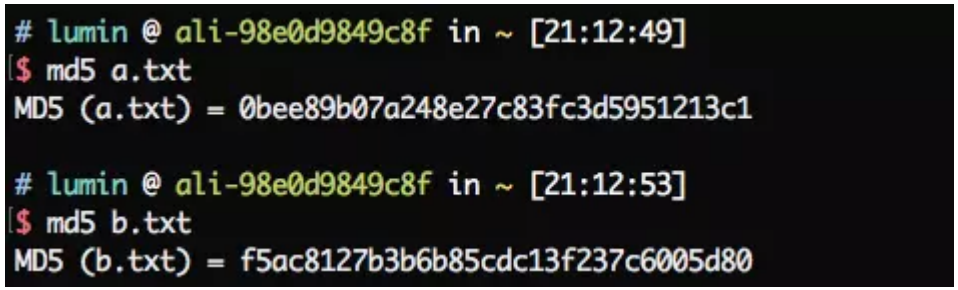
简介

一致性Hash是一种特殊的Hash算法，由于其均衡性、持久性的映射特点，被广泛的应用于负载均衡领域，如nginx和memcached都采用了一致性Hash来作为集群负载均衡的方案。

本文将介绍一致性Hash的基本思路，并讨论其在分布式缓存集群负载均衡中的应用。同时也会进行相应的代码测试来验证其算法特性，并给出和其他负载均衡方案的一些对比。

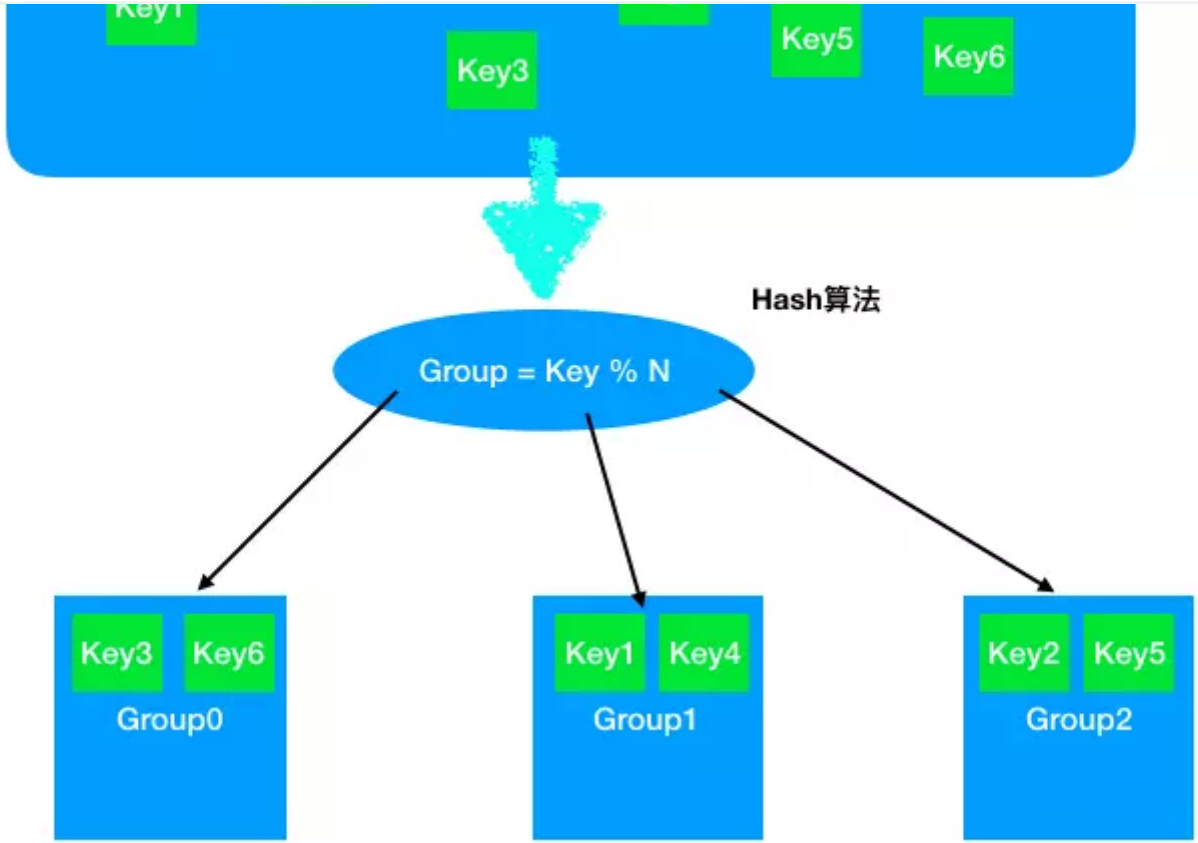
一致性Hash算法简介

比如，对字符串`abc`和`abcd`分别进行md5计算，得到的结果如下：



可以看到，两个在形式上非常相近的数据经过md5散列后，变成了完全随机的字符串。负载均衡正是利用这一特性，对于大量随机的请求或调用，通过一定形式的Hash将他们均匀的**散列**，从而实现压力的平均化。（当然，并不是只要使用了Hash就一定能够获得均匀的散列，后面会分析这一点。）

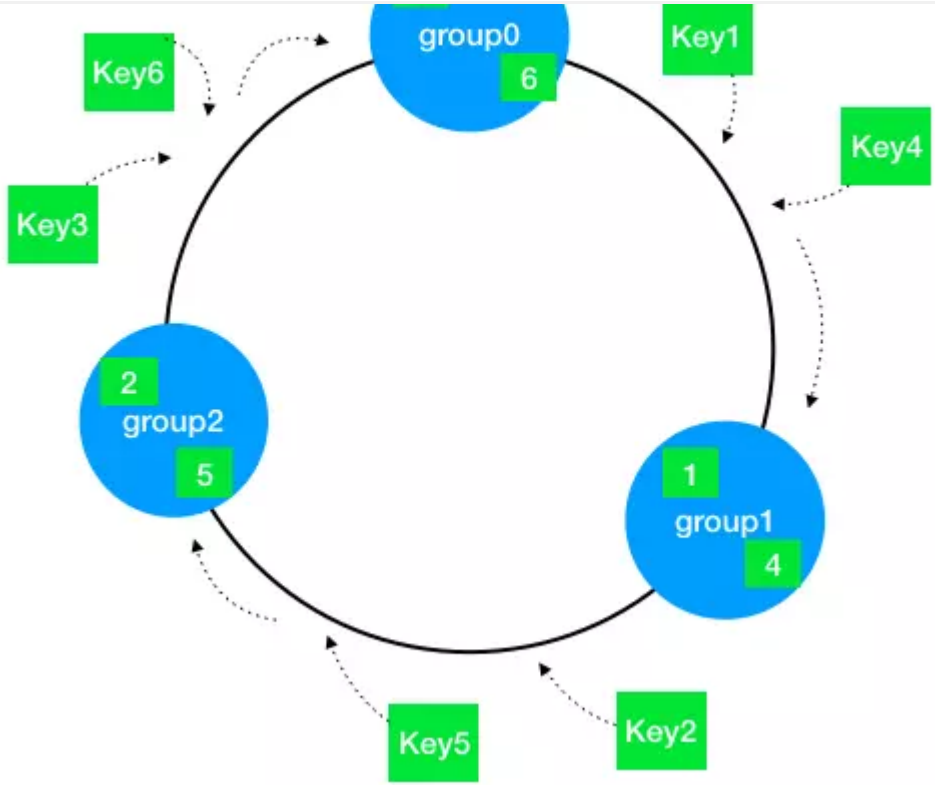
举个例子，如果我们给每个请求生成一个Key，只要使用一个非常简单的Hash算法`Group = Key % N`来实现请求的负载均衡，如下：



（如果将Key作为缓存的Key，对应的Group储存该Key的Value，就可以实现一个分布式的缓存系统，后文的具体例子都将基于这个场景）

不难发现，这样的Hash只要集群的数量N发生变化，之前的所有Hash映射就会全部失效。如果集群中的每个机器提供的服务没有差别，倒不会产生什么影响，但对于分布式缓存这样的系统而言，映射全部失效就意味着之前的缓存全部失效，后果将会是灾难性的。

一致性Hash通过构建环状的Hash空间代替线性Hash空间的方法解决了这个问题，如下图：

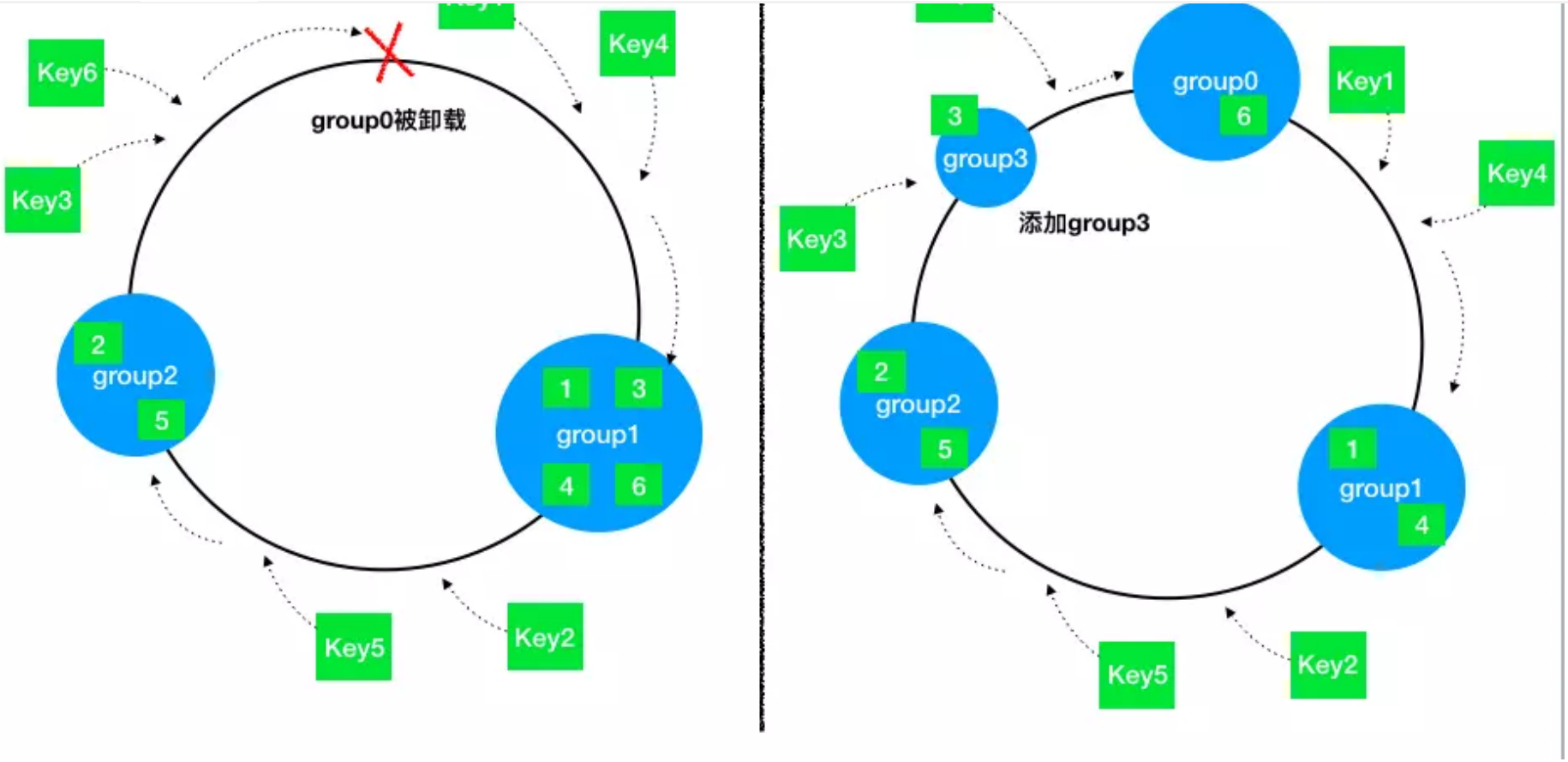


整个Hash空间被构建成一个首尾相接的环，使用一致性Hash时需要进行两次映射。

第一次，给每个节点（集群）计算Hash，然后记录它们的Hash值，这就是它们在环上的位置。

第二次，给每个Key计算Hash，然后沿着顺时针的方向找到环上的第一个节点，就是该Key储存对应的集群。

分析一下节点增加和删除时对负载均衡的影响，如下图：



可以看到，当节点被删除时，其余节点在环上的映射不会发生改变，只是原来打在对应节点上的Key现在会转移到顺时针方向的下一个节点上去。增加一个节点也是同样的，最终都只有少部分的Key发生了失效。不过发生节点变动后，整体系统的压力已经不是均衡的了，下文中提到的方法将会解决这个问题。

问题与优化

数据倾斜

如果节点的数量很少，而hash环空间很大（一般是 0 ~ 2^32），直接进行一致性hash上去，大部分情况下节点在环上的位置会很不均匀，挤在某个很小的区域。最终对分布式缓存造成的影响就是，集群的每个实例上储存的缓存数据量不一致，会发生严重的数据倾斜。

缓存雪崩

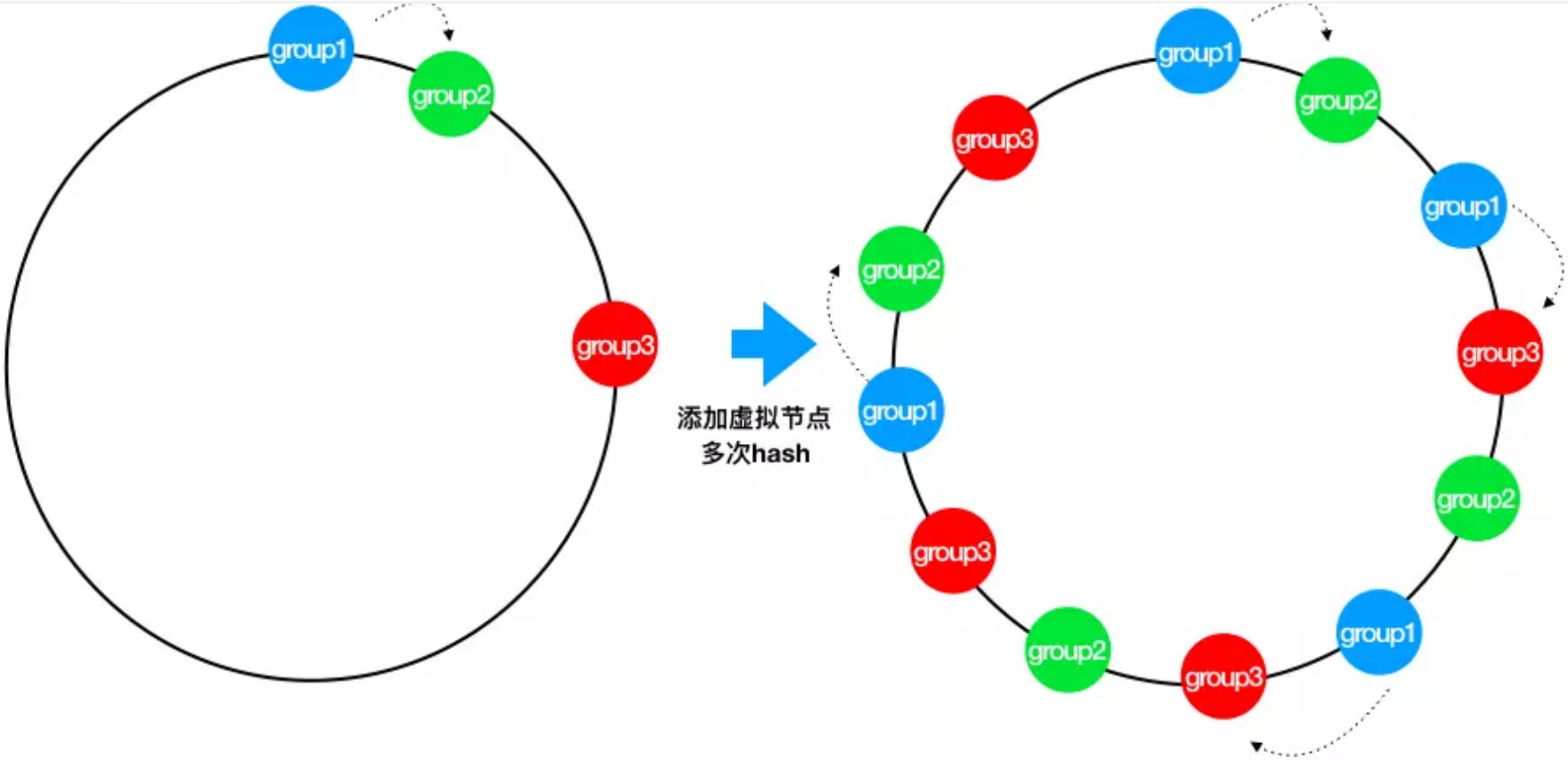
如果每个节点在环上只有一个节点，那么可以想象，当某一集群从环中消失时，它原本所负责的任务将全部交由顺时针方向的下一个集群处理。例如，当group0退出时，它原本所负责的缓存将全部交给group1处理。这就意味着group1的访问压力会瞬间增大。设想一下，如果group1因为压力过大而崩溃，那么更大的压力又会向group2压过去，最终服务压力就像滚雪球一样越滚越大，最终导致雪崩。

引入虚拟节点

解决上述两个问题最好的办法就是扩展整个环上的节点数量，因此我们引入了虚拟节点的概念。一个实际节点将会映射多个虚拟节点，这样Hash环上的空间分割就会变得均匀。

同时，引入虚拟节点还会使得节点在Hash环上的顺序随机化，这意味着当一个真实节点失效退出后，它原来所承载的压力将会均匀地分散到其他节点上去。

如下图：



代码测试

现在我们尝试编写一些测试代码，来看看一致性hash的实际效果是否符合我们预期。

首先我们需要一个能够对输入进行均匀散列的Hash算法，可供选择的有很多，memcached官方使用了基于md5的`KETAMA`算法，但这里处于计算效率的考虑，使用了`FNV1_32_HASH`算法，如下：

```
     * 计算Hash值, 使用FNV1_32_HASH算法
     * @param str
     * @return
     */
    public static int getHash(String str) {
        final int p = 16777619;
        int hash = (int)2166136261L;
        for (int i = 0; i < str.length(); i++) {
            hash =( hash ^ str.charAt(i) ) * p;
        }
        hash += hash << 13;
        hash ^= hash >> 7;
        hash += hash << 3;
        hash ^= hash >> 17;
        hash += hash << 5;

        if (hash < 0) {
            hash = Math.abs(hash);
        }
        return hash;
    }
}
```

实际使用时可以根据需求调整。

接着需要使用一种数据结构来保存hash环，可以采用的方案有很多种，最简单的是采用数组或链表。但这样查找的时候需要进行排序，如果节点数量多，速度就可能变得很慢。

先编写一个最简单的，无虚拟节点的Hash环测试：

java 复制代码

```java
public class ConsistentHashingWithoutVirtualNode {

    /**
     * 集群地址列表
     */
    private static String[] groups = {
        "192.168.0.0:111", "192.168.0.1:111", "192.168.0.2:111",
        "192.168.0.3:111", "192.168.0.4:111"
    };

    /**
     * 用于保存Hash环上的节点
     */
    private static SortedMap<Integer, String> sortedMap = new TreeMap<>();

    /**
     * 初始化，将所有的服务器加入Hash环中
     */
    static {
        // 使用红黑树实现，插入效率比较差，但是查找效率极高
        for (String group : groups) {
            int hash = HashUtil.getHash(group);
            System.out.println("[" + group + "] launched @ " + hash);
            sortedMap.put(hash, group);
        }
    }
```

```
     * @param widgetKey
     * @return
     */
    private static String getServer(String widgetKey) {
        int hash = HashUtil.getHash(widgetKey);
        // 只取出所有大于该hash值的部分而不必遍历整个Tree
        SortedMap<Integer, String> subMap = sortedMap.tailMap(hash);
        if (subMap == null || subMap.isEmpty()) {
            // hash值在最尾部，应该映射到第一个group上
            return sortedMap.get(sortedMap.firstKey());
        }
        return subMap.get(subMap.firstKey());
    }

    public static void main(String[] args) {
        // 生成随机数进行测试
        Map<String, Integer> resMap = new HashMap<>();

        for (int i = 0; i < 100000; i++) {
            Integer widgetId = (int)(Math.random() * 10000);
            String server = getServer(widgetId.toString());
            if (resMap.containsKey(server)) {
                resMap.put(server, resMap.get(server) + 1);
            } else {
                resMap.put(server, 1);
            }
        }

        resMap.forEach(
            (k, v) -> {
                System.out.println("group " + k + ": " + v + "(" + v/1000.0D +"%)");
```

```
}
```

生成10000个随机数字进行测试，最终得到的压力分布情况如下：

复制代码

```
[192.168.0.1:111] launched @ 8518713
[192.168.0.2:111] launched @ 1361847097
[192.168.0.3:111] launched @ 1171828661
[192.168.0.4:111] launched @ 1764547046
group 192.168.0.2:111: 8572(8.572%)
group 192.168.0.1:111: 18693(18.693%)
group 192.168.0.4:111: 17764(17.764%)
group 192.168.0.3:111: 27870(27.87%)
group 192.168.0.0:111: 27101(27.101%)
```

可以看到压力还是比较不平均的，所以我们继续，引入虚拟节点：

java 复制代码

```java
public class ConsistentHashingWithVirtualNode {
    /**
     * 集群地址列表
     */
    private static String[] groups = {
        "192.168.0.0:111", "192.168.0.1:111", "192.168.0.2:111",
        "192.168.0.3:111", "192.168.0.4:111"
    };

    /**
```

```
/**
 * 虚拟节点映射关系
 */
private static SortedMap<Integer, String> virtualNodes = new TreeMap<>();

private static final int VIRTUAL_NODE_NUM = 1000;

static {
    // 先添加真实节点列表
    realGroups.addAll(Arrays.asList(groups));

    // 将虚拟节点映射到Hash环上
    for (String realGroup: realGroups) {
        for (int i = 0; i < VIRTUAL_NODE_NUM; i++) {
            String virtualNodeName = getVirtualNodeName(realGroup, i);
            int hash = HashUtil.getHash(virtualNodeName);
            System.out.println("[" + virtualNodeName + "] launched @ " + hash);
            virtualNodes.put(hash, virtualNodeName);
        }
    }
}

private static String getVirtualNodeName(String realName, int num) {
    return realName + "&&VN" + String.valueOf(num);
}

private static String getRealNodeName(String virtualName) {
    return virtualName.split("&&")[0];
}
```

```
        SortedMap<Integer, String> subMap = virtualNodes.tailMap(hash);
        String virtualNodeName;
        if (subMap == null || subMap.isEmpty()) {
            // hash值在最尾部，应该映射到第一个group上
            virtualNodeName = virtualNodes.get(virtualNodes.firstKey());
        }else {
            virtualNodeName = subMap.get(subMap.firstKey());
        }
        return getRealNodeName(virtualNodeName);
    }

    public static void main(String[] args) {
        // 生成随机数进行测试
        Map<String, Integer> resMap = new HashMap<>();

        for (int i = 0; i < 100000; i++) {
            Integer widgetId = i;
            String group = getServer(widgetId.toString());
            if (resMap.containsKey(group)) {
                resMap.put(group, resMap.get(group) + 1);
            } else {
                resMap.put(group, 1);
            }
        }

        resMap.forEach(
            (k, v) -> {
                System.out.println("group " + k + ": " + v + "(" + v/100000.0D +"%)");
            }
        );
```

这里真实节点和虚拟节点的映射采用了字符串拼接的方式，这种方式虽然简单但很有效，memcached官方也是这么实现的。将虚拟节点的数量设置为1000，重新测试压力分布情况，结果如下：

复制代码

```
group 192.168.0.2:111: 18354(18.354%)
group 192.168.0.1:111: 20062(20.062%)
group 192.168.0.4:111: 20749(20.749%)
group 192.168.0.3:111: 20116(20.116%)
group 192.168.0.0:111: 20719(20.719%)
```

可以看到基本已经达到平均分布了，接着继续测试删除和增加节点给整个服务带来的影响，相关测试代码如下：

java 复制代码

```
private static void refreshHashCircle() {
    // 当集群变动时，刷新hash环，其余的集群在hash环上的位置不会发生变动
        virtualNodes.clear();
    for (String realGroup: realGroups) {
        for (int i = 0; i < VIRTUAL_NODE_NUM; i++) {
                String virtualNodeName = getVirtualNodeName(realGroup, i);
            int hash = HashUtil.getHash(virtualNodeName);
            System.out.println("[" + virtualNodeName + "] launched @ " + hash);
            virtualNodes.put(hash, virtualNodeName);
        }
    }
}
private static void addGroup(String identifier) {
        realGroups.add(identifier);
    refreshHashCircle();
}
```

```
    for (String group:realGroups) {
        if (group.equals(identifier)) {
                realGroups.remove(i);
        }
        i++;
    }
    refreshHashCircle();
}
```

测试删除一个集群前后的压力分布如下：

复制代码

```
running the normal test.
group 192.168.0.2:111: 19144(19.144%)
group 192.168.0.1:111: 20244(20.244%)
group 192.168.0.4:111: 20923(20.923%)
group 192.168.0.3:111: 19811(19.811%)
group 192.168.0.0:111: 19878(19.878%)
removed a group, run test again.
group 192.168.0.2:111: 23409(23.409%)
group 192.168.0.1:111: 25628(25.628%)
group 192.168.0.4:111: 25583(25.583%)
group 192.168.0.0:111: 25380(25.38%)
```

同时计算一下消失的集群上的Key最终如何转移到其他集群上：

复制代码

```
[192.168.0.1:111-192.168.0.4:111] :5255
[192.168.0.1:111-192.168.0.3:111] :5090
```

可见，删除集群后，该集群上的压力均匀地分散给了其他集群，最终整个集群仍处于负载均衡状态，符合我们的预期，最后看一下添加集群的情况。

压力分布：

复制代码

```
running the normal test.
group 192.168.0.2:111: 18890(18.89%)
group 192.168.0.1:111: 20293(20.293%)
group 192.168.0.4:111: 21000(21.0%)
group 192.168.0.3:111: 19816(19.816%)
group 192.168.0.0:111: 20001(20.001%)
add a group, run test again.
group 192.168.0.2:111: 15524(15.524%)
group 192.168.0.7:111: 16928(16.928%)
group 192.168.0.1:111: 16888(16.888%)
group 192.168.0.4:111: 16965(16.965%)
group 192.168.0.3:111: 16768(16.768%)
group 192.168.0.0:111: 16927(16.927%)
```

压力转移：

复制代码

```
[192.168.0.0:111-192.168.0.7:111] :3102
[192.168.0.4:111-192.168.0.7:111] :4060
[192.168.0.2:111-192.168.0.7:111] :3313
[192.168.0.1:111-192.168.0.7:111] :3292
[192.168.0.3:111-192.168.0.7:111] :3261
```

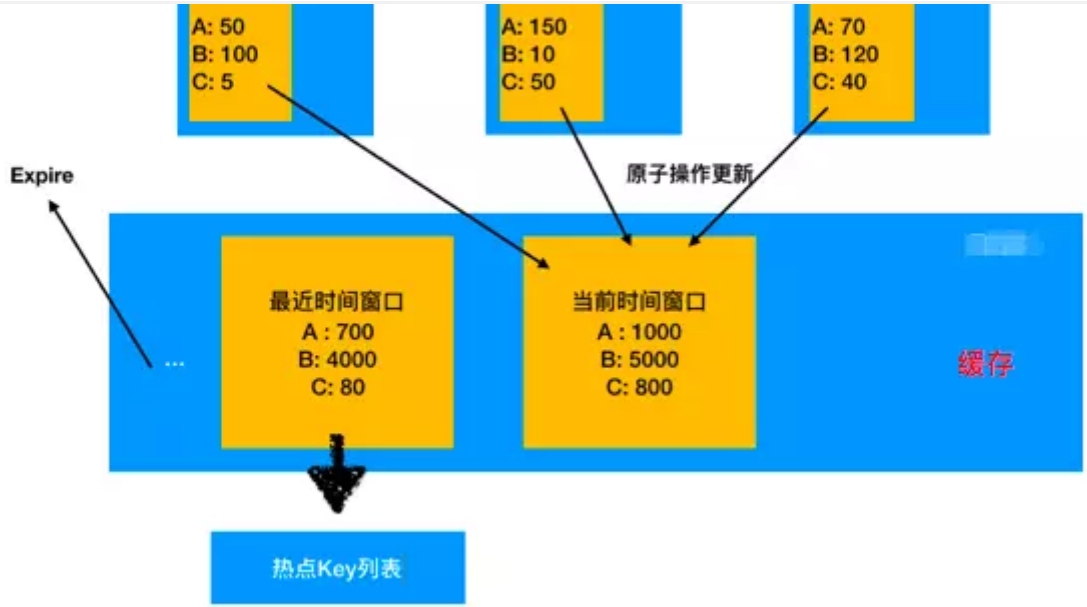
优雅缩扩容

缓存服务器对于性能有着较高的要求，因此我们希望在扩容时新的集群能够较快的填充好数据并工作。但是从一个集群启动，到真正加入并可以提供服务之间还存在着不小的时间延迟，要实现更优雅的扩容，我们可以从两个方面出发：

1. 高频Key预热

   负载均衡器作为路由层，是可以收集并统计每个缓存Key的访问频率的，如果能够维护一份高频访问Key的列表，新的集群在启动时根据这个列表提前拉取对应Key的缓存值进行预热，便可以大大减少因为新增集群而导致的Key失效。

   具体的设计可以通过缓存来实现，如下：

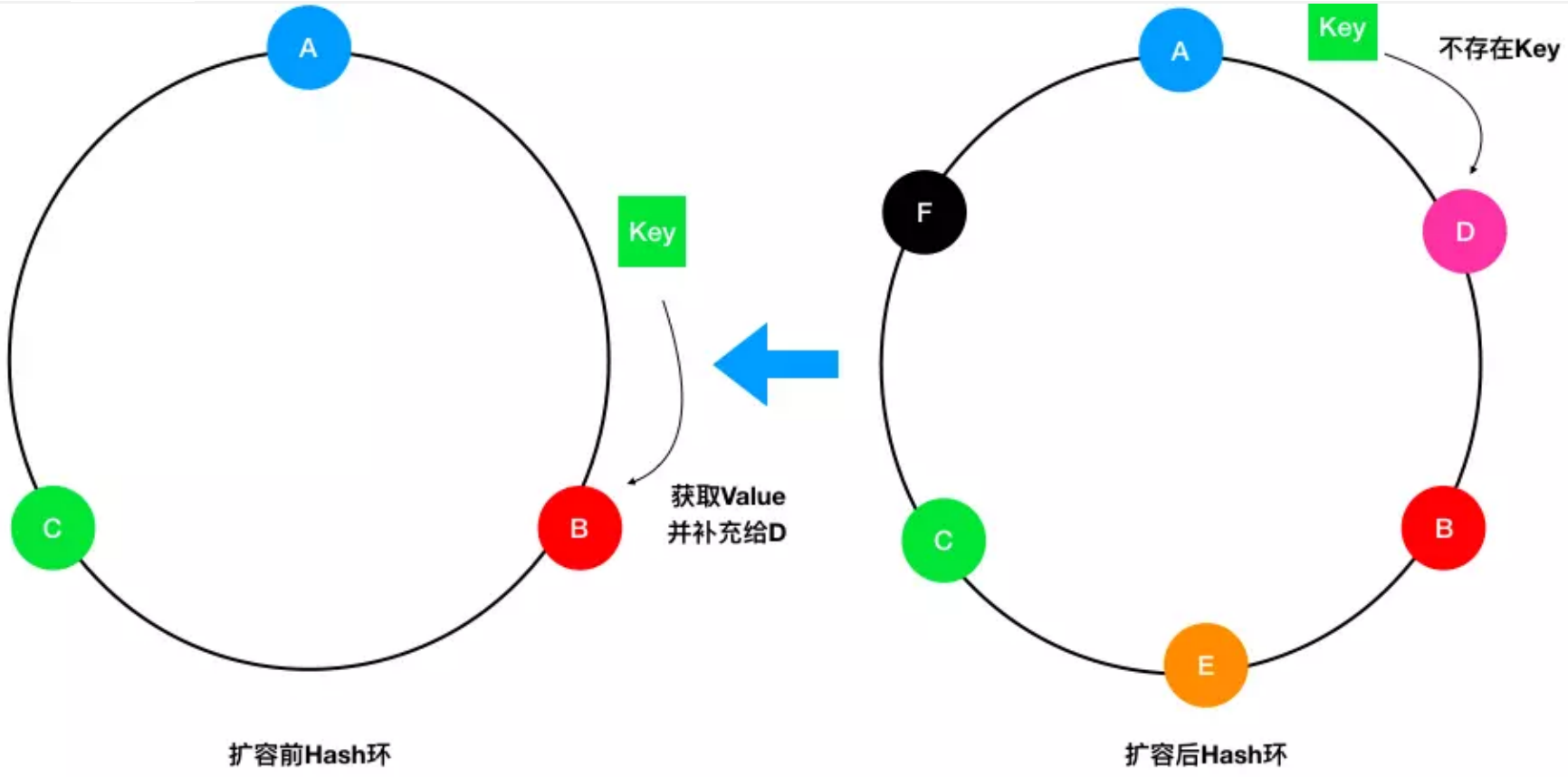


不过这个方案在实际使用时有一个很大的限制，那就是高频Key本身的缓存失效时间可能很短，预热时储存的Value在实际被访问到时可能已经被更新或者失效，处理不当会导致出现脏数据，因此实现难度还是有一些大的。

2. 历史Hash环保留

回顾一致性Hash的扩容，不难发现新增节点后，它所对应的Key在原来的节点还会保留一段时间。因此在扩容的延迟时间段，如果对应的Key缓存在新节点上还没有被加载，可以去原有的节点上尝试读取。

举例，假设我们原有3个集群，现在要扩展到6个集群，这就意味着原有50%的Key都会失效（被转移到新节点上），如果我们维护扩容前和扩容后的两个Hash环，在扩容后的Hash环上找不到Key的储存时，先转向扩容前的Hash环寻找一波，如果能够找到就返回对应的值并将该缓存写入新的节点上，找不到时再透过缓存，如下图：



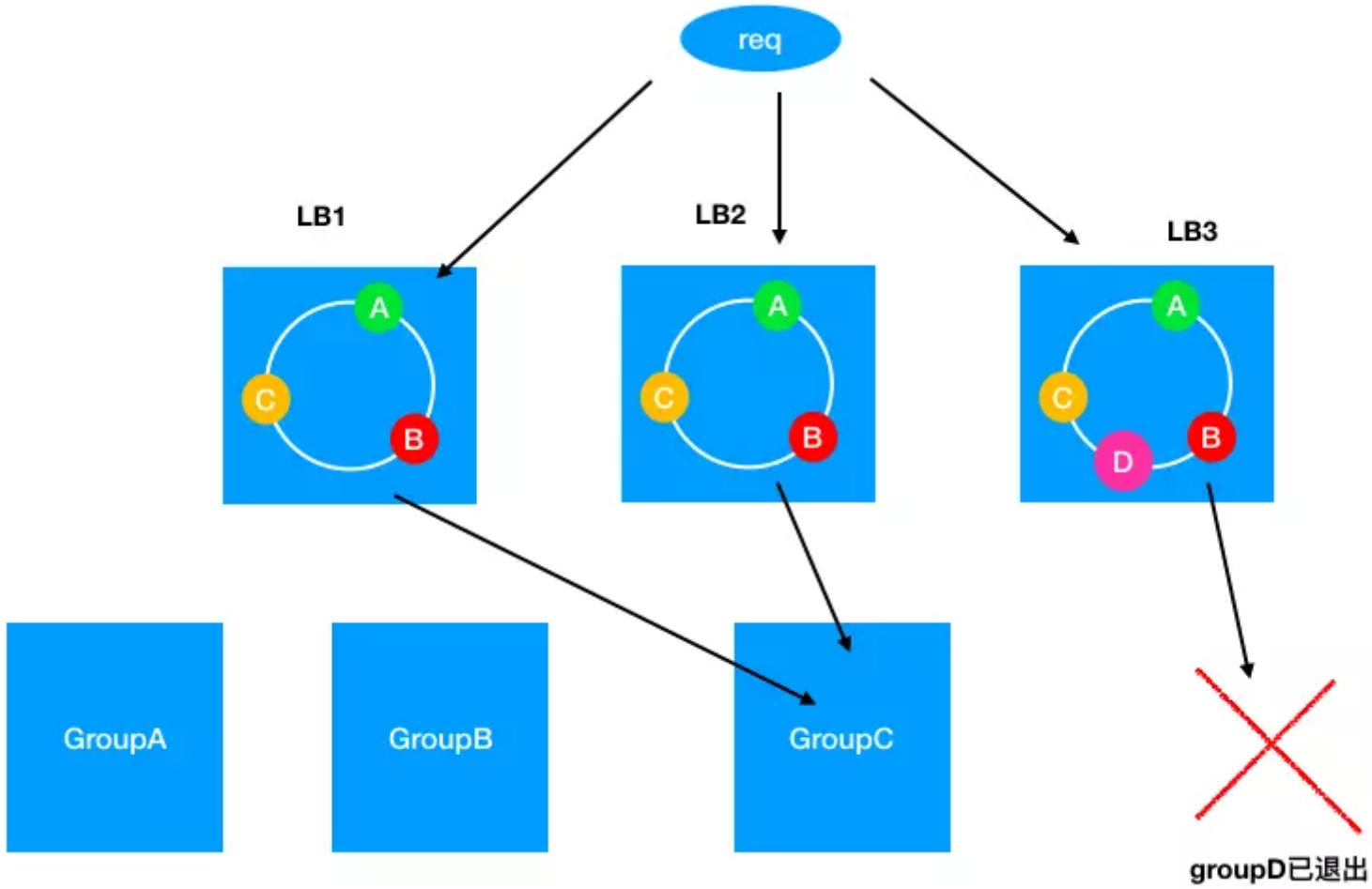
这样做的缺点是增加了缓存读取的时间，但相比于直接击穿缓存而言还是要好很多的。优点则是可以随意扩容多台机器，而不会产生大面积的缓存失效。

谈完了扩容，再谈谈缩容。

1. 熔断机制

2. 多集群LB的更新延迟

这个问题在缩容时比较严重，如果你使用一个集群来作为负载均衡，并使用一个配置服务器比如ConfigServer来推送集群状态以构建Hash环，那么在某个集群退出时这个状态并不一定会被立刻同步到所有的LB上，这就可能会导致一个暂时的调度不一致，如下图：



- 缓慢缩容，等到Hash环完全同步后再操作。可以通过监听退出集群的访问QPS来实现这一点，等到该集群几乎没有QPS时再将其撤下。
- 手动删除，如果Hash环上对应的节点找不到了，就手动将其从Hash环上删除，然后重新进行调度，这个方式有一定的风险，对于网络抖动等异常情况兼容的不是很好。
- 主动拉取和重试，当Hash环上节点失效时，主动从ZK上重新拉取集群状态来构建新Hash环，在一定次数内可以进行多次重试。

对比：HashSlot

了解了一致性Hash算法的特点后，我们也不难发现一些不尽人意的地方：

- 整个分布式缓存需要一个路由服务来做负载均衡，存在单点问题（如果路由服务挂了，整个缓存也就凉了）
- Hash环上的节点非常多或者更新频繁时，查找性能会比较低下

针对这些问题，Redis在实现自己的分布式集群方案时，设计了全新的思路：基于P2P结构的HashSlot算法，下面简单介绍一下：

1. 使用HashSlot

   类似于Hash环，Redis Cluster采用HashSlot来实现Key值的均匀分布和实例的增删管理。

   首先默认分配了16384个Slot（这个大小正好可以使用2kb的空间保存），每个Slot相当于一致性Hash环上的一个节点。接入集群的所有实例将均匀地占有这些Slot，而最终当我们Set一个Key时，使用`CRC16(Key) % 16384`来计算出这个Key属于哪个Slot，并最终映射到对应的实例上去。

   那么当增删实例时，Slot和实例间的对应要如何进行对应的改动呢?

```
节点A    0－5460
节点B    5461－10922
节点C    10923－16383
```

现在假设要增加一个节点D，RedisCluster的做法是将之前每台机器上的一部分Slot移动到D上（注意这个过程也意味着要对节点D写入的KV储存），成功接入后Slot的覆盖情况将变为如下情况：

复制代码

```
节点A    1365-5460
节点B    6827-10922
节点C    12288-16383
节点D    0-1364,5461-6826,10923-12287
```

同理删除一个节点，就是将其原来占有的Slot以及对应的KV储存均匀地归还给其他节点。

2. P2P节点寻找

现在我们考虑如何实现去中心化的访问，也就是说无论访问集群中的哪个节点，你都能够拿到想要的数据。其实这有点类似于路由器的路由表，具体说来就是：

- 每个节点都保存有完整的`HashSlot – 节点`映射表，也就是说，每个节点都知道自己拥有哪些Slot，以及某个确定的Slot究竟对应着哪个节点。
- 无论向哪个节点发出寻找Key的请求，该节点都会通过`CRC(Key) % 16384`计算该Key究竟存在于哪个Slot，并将请求转发至该Slot所在的节点。

总结一下就是两个要点：**映射表**和**内部转发**，这是通过著名的**Gossip协议**来实现的。

对比一下，HashSlot + P2P的方案解决了去中心化的问题，同时也提供了更好的动态扩展性。但相比于一致性Hash而言，其结构更加复杂，实现上也更加困难。

而在之前的分析中我们也能看出，一致性Hash方案整体上还是有着不错的表现的，因此在实际的系统应用中，可以根据开发成本和性能要求合理地选择最适合的方案。总之，两者都非常优秀，至于用哪个、怎么用，就是仁者见仁智者见智的问题了。

- 服务端架构之缓存篇（2）：分布式缓存
- 一致性哈希算法之Ketama算法
- 一致性哈希模型如何优雅扩容
- 对一致性Hash算法，Java代码实现的深入研究

关注下面的标签，发现更多相似文章

Redis 算法 负载均衡 Memcached

不洗碗工作室 Lv3 秦皇岛牛客科技有限公司
发布了 40 篇专栏 · 获得点赞 1,552 · 获得阅读 48,336

关注

评论

输入评论...

掘金值得关注的技术团队 优质技术团队推荐官 @ 掘金

Hi，不洗碗君

我是掘金官方运营的账号：掘金值得关注的技术团队，掘金将上线一个基于关注关系新的版本--「动态」，新版本里关注你的粉丝将看到你

1年前 回复

hhlucky🌞

👍 非常棒

1年前 回复

lp

赞，写的不错

1年前 回复

knock_小新 Lv2

写的不错，nice!

1年前 回复

相关推荐

专栏 · 进击的小喽啰 · 12小时前 · 算法

搞定技术面试: 结合 LeetCode 谈谈哈希表在算法问题上的应用

4

专栏 · LeviDing · 1小时前 · 算法 / LeetCode

[LeetCode] 204. 计数质数：JavaScript 实现埃拉托斯特尼筛法

4

36 2

专栏 · 北海北方 · 1年前 · HTTP

HTTP----HTTP缓存机制

699 45

专栏 · 小兀666 · 2天前 · 算法

从libuv源码中学习红黑树

53 4

荐 · 专栏 · yanglbme · 1天前 · Redis

面试官：redis 的过期策略都有哪些？内存淘汰机制都有哪些？手写一下 LRU 代码实现？

23 11

专栏 · 呼延十 · 1天前 · 算法

如何计算两个字符串之间的文本相似度?

18 1

专栏 · 王磊的博客 · 1天前 · Redis / Java

Redis中的键值过期操作

11

实时数据并发写入 Redis 优化

14 2

专栏 · 小鹿动画学编程 · 3天前 · 算法

动画：二分查找(上) | 面试官问我如何在 1 亿数据中快速查找某一整数?

10 1