03-如何实现一个性能优异的Hash表？

你好，我是蒋德钧。今天，我们来聊聊Redis中的Hash。

我们知道，Hash表是一种非常关键的数据结构，在计算机系统中发挥着重要作用。比如在Memcached中，Hash表被用来索引数据；在数据库系统中，Hash表被用来辅助SQL查询。而对于Redis键值数据库来说，Hash表既是键值对中的一种值类型，同时，Redis也使用一个全局Hash表来保存所有的键值对，从而既满足应用存取Hash结构数据需求，又能提供快速查询功能。

那么，Hash表应用如此广泛的一个重要原因，就是从理论上来说，它能以O(1)的复杂度快速查询数据。Hash表通过Hash函数的计算，就能定位数据在表中的位置，紧接着可以对数据进行操作，这就使得数据操作非常快速。

Hash表这个结构也并不难理解，但是在实际应用Hash表时，当数据量不断增加，它的性能就经常会受到**哈希冲突**和**rehash开销**的影响。而这两个问题的核心，其实都来自于Hash表要保存的数据量，超过了当前Hash表能容纳的数据量。

那么要如何应对这两个问题呢？事实上，这也是在大厂面试中，面试官经常会考核的问题。所以你现在可以先想想，如果你在面试中遇到了这两个问题，你会怎么回答呢？

OK，思考先到这里，现在我来告诉你Redis是怎么很好地解决这两个问题的。

Redis为我们提供了一个经典的Hash表实现方案。针对哈希冲突，Redis采用了**链式哈希**，在不扩容哈希表的前提下，将具有相同哈希值的数据链接起来，以便这些数据在表中仍然可以被查询到；对于rehash开销，Redis实现了**渐进式rehash设计**，进而缓解了rehash操作带来的额外开销对系统的性能影响。

所以这节课，我就带你来学习Redis中针对Hash表的设计思路和实现方法，帮助你掌握应对哈希冲突和优化rehash操作性能的能力，并以此支撑你在实际使用Hash表保存大量数据的场景中，可以实现高性能的Hash表。

好了，接下来，我们就先来聊聊链式哈希的设计与实现。

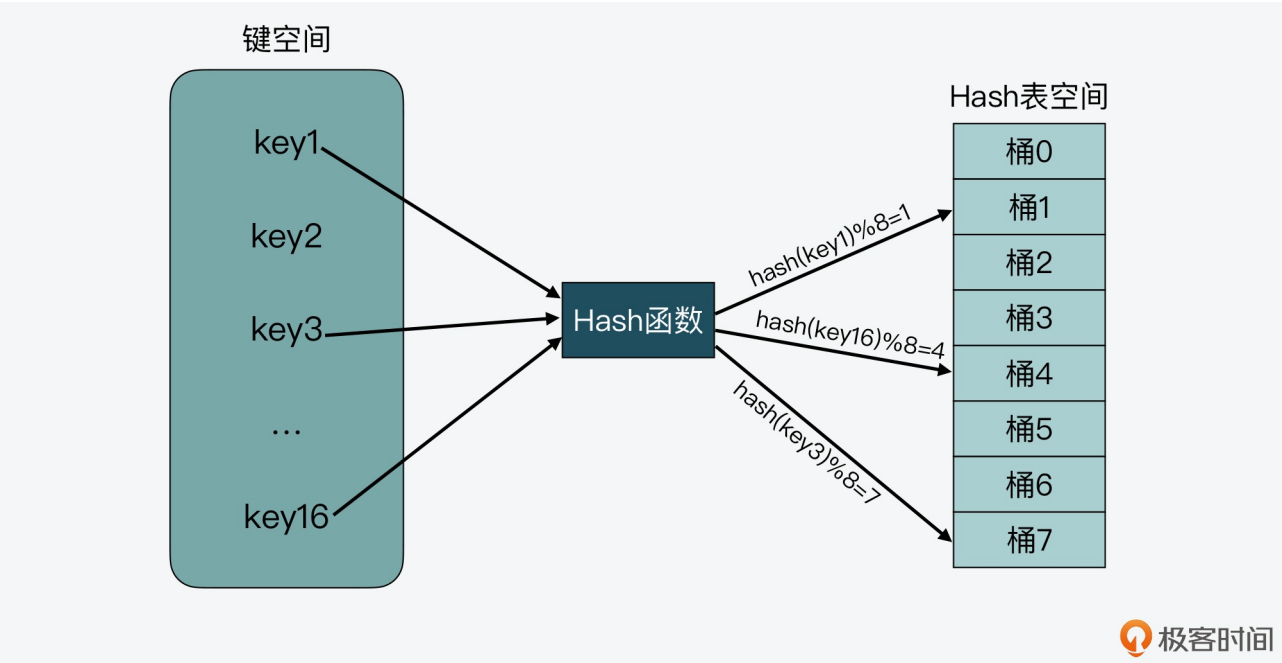
Redis如何实现链式哈希？

不过，在开始学习链式哈希的设计实现之前，我们还需要明白Redis中Hash表的结构设计是啥样的，以及为何会在数据量增加时产生哈希冲突，这样也更容易帮助我们理解链式哈希应对哈希冲突的解决思路。

什么是哈希冲突？

实际上，一个最简单的Hash表就是一个数组，数组里的每个元素是一个哈希桶（也叫做Bucket），第一个数组元素被编为哈希桶0，以此类推。当一个键值对的键经过Hash函数计算后，再对数组元素个数取模，就能得到该键值对对应的数组元素位置，也就是第几个哈希桶。

如下图所示，key1经过哈希计算和哈希值取模后，就对应哈希桶1，类似的，key3和key16分别对应哈希桶7和桶4。

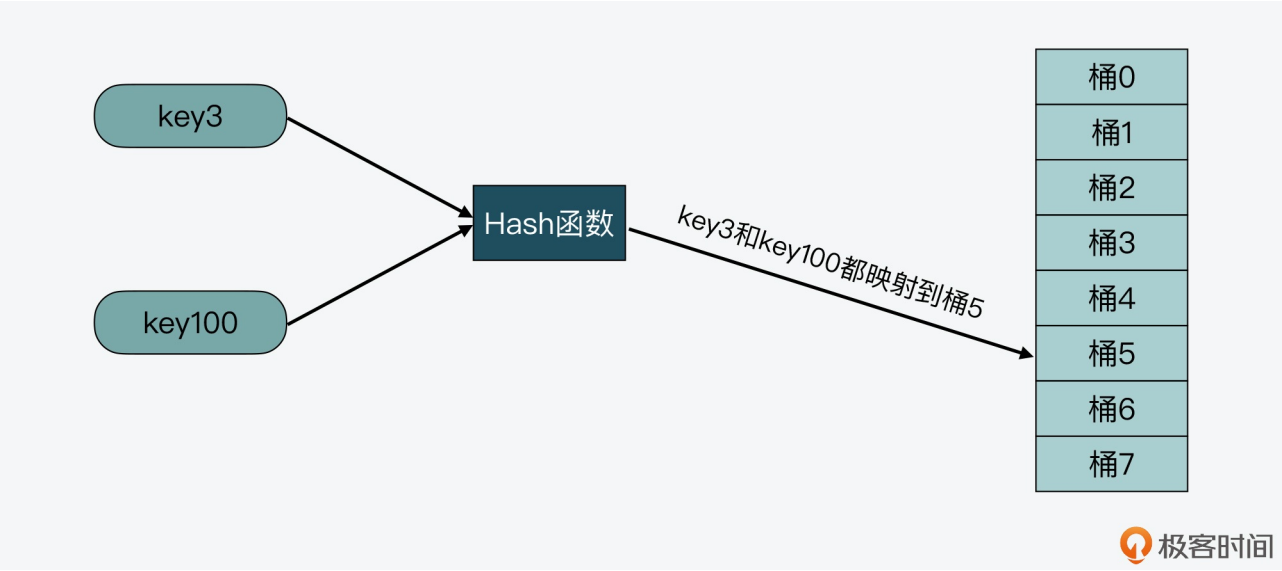


从图上我们还可以看到，需要写入Hash表的键空间一共有16个键，而Hash表的空间大小只有8个元素，这样就会导致有些键会对应到相同的哈希桶中。

我们在实际应用Hash表时，其实一般很难预估要保存的数据量，如果我们一开始就创建一个非常大的哈希表，当数据量较小时，就会造成空间浪费。所以，我们通常会给哈希表设定一个初始大小，而当数据量增加时，键空间的大小就会大于Hash表空间大小了。

也正是由于键空间会大于Hash表空间，这就导致在用Hash函数把键映射到Hash表空间时，不可避免地会出现不同的键被映射到数组的同一个位置上。而如果同一个位置只能保存一个键值对，就会导致Hash表保存的数据非常有限，这就是我们常说的**哈希冲突**。

比如下图中，key3和key100都被映射到了Hash表的桶5中，这样，当桶5只能保存一个key时，key3和key100就会有一个key无法保存到哈希表中了。



那么我们该如何解决哈希冲突呢？可以考虑使用以下两种解决方案：

- 第一种方案，就是我接下来要给你介绍的**链式哈希**。这里你需要先知道，链式哈希的链不能太长，否则会降低Hash表性能。

- 第二种方案，就是当链式哈希的链长达到一定长度时，我们可以使用**rehash**。不过，执行rehash本身开销比较大，所以就需要采用我稍后会给你介绍的渐进式rehash设计。

这里，我们先来了解链式哈希的设计和实现。

链式哈希如何设计与实现？

所谓的链式哈希，就是**用一个链表把映射到Hash表同一桶中的键给连接起来**。下面我们就来看看Redis是如何实现链式哈希的，以及为何链式哈希能够帮助解决哈希冲突。

首先，我们需要了解Redis源码中对Hash表的实现。Redis中和Hash表实现相关的文件主要是**dict.h**和**dict.c**。其中，dict.h文件定义了Hash表的结构、哈希项，以及Hash表的各种操作函数，而dict.c文件包含了Hash表各种操作的具体实现代码。

在dict.h文件中，Hash表被定义为一个二维数组（dictEntry **table），这个数组的每个元素是一个指向哈希项（dictEntry）的指针。下面的代码展示的就是在dict.h文件中对Hash表的定义，你可以看下：

```
typedef struct dictht {
    dictEntry **table; //二维数组
    unsigned long size; //Hash表大小
    unsigned long sizemask;
    unsigned long used;
} dictht;
```

那么为了实现链式哈希， Redis在每个dictEntry的结构设计中，**除了包含指向键和值的指针，还包含了指向下一个哈希项的指针**。如下面的代码所示，dictEntry结构体中包含了指向另一个dictEntry结构的**指针*next**，这就是用来实现链式哈希的：

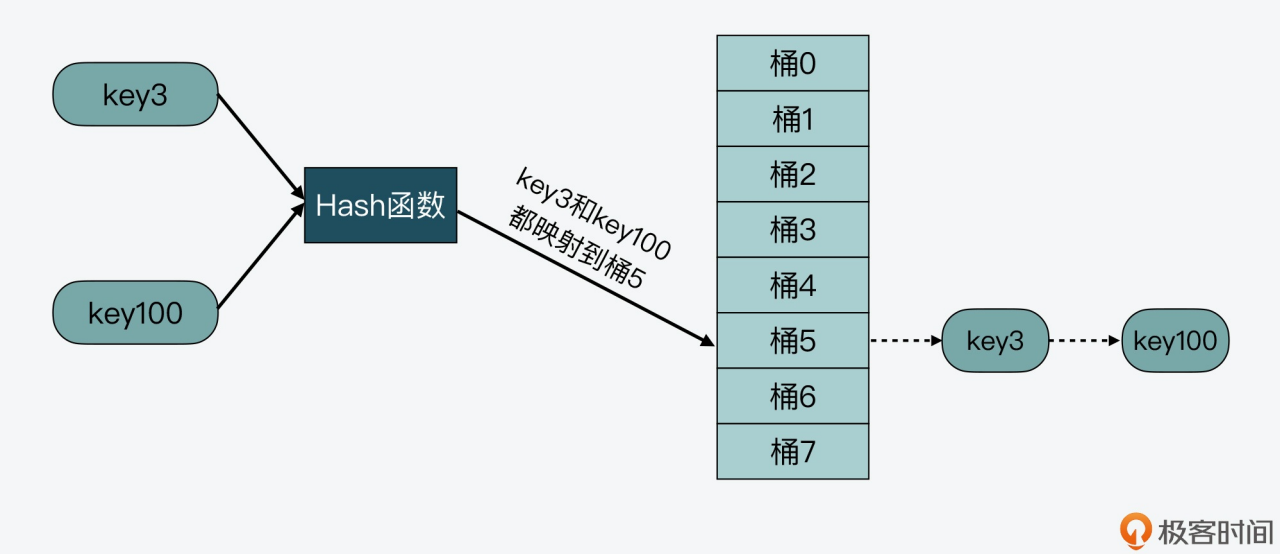
```
typedef struct dictEntry {
    void *key;
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    struct dictEntry *next;
} dictEntry;
```

除了用于实现链式哈希的指针外，这里还有一个值得注意的地方，就是在dictEntry结构体中，键值对的值是由一个**联合体v**定义的。这个联合体v中包含了指向实际值的指针*val，还包含了无符号的64位整数、有符号的64位整数，以及double类的值。

我之所以要提醒你注意这里，其实是为了说明，**这种实现方法是一种节省内存的开发小技巧**，非常值得学习。因为当值为整数或双精度浮点数时，由于其本身就是64位，就可以不用指针指向了，而是可以直接存在键值对的结构体中，这样就避免了再用一个指针，从而节省了内存空间。

好了，那么到这里，你应该就了解了Redis中链式哈希的实现，不过现在你可能还是不太明白，为什么这种链式哈希可以帮助解决哈希冲突呢？

别着急，我就拿刚才的例子来说明一下，key3和key100都被映射到了Hash表的桶5中。而当使用了链式哈希，桶5就不会只保存key3或key100，而是会用一个链表把key3和key100连接起来，如下图所示。当有更多的key被映射到桶5时，这些key都可以用链表串接起来，以应对哈希冲突。



这样，当我们要查询key100时，可以先通过哈希函数计算，得到key100的哈希值被映射到了桶5中。然后，我们再逐一比较桶5中串接的key，直到查找到key100。如此一来，我们就能在链式哈希中找到所查的哈希项了。

不过，链式哈希也存在局限性，那就是随着链表长度的增加，Hash表在一个位置上查询哈希项的耗时就会增加，从而增加了Hash表的整体查询时间，这样也会导致Hash表的性能下降。

那么，有没有什么其他的方法可以减少对Hash表性能的影响呢？当然是有的，这就是接下来我要给你介绍的rehash的设计与实现了。

Redis如何实现rehash？

rehash操作，其实就是指扩大Hash表空间。而Redis实现rehash的基本思路是这样的：

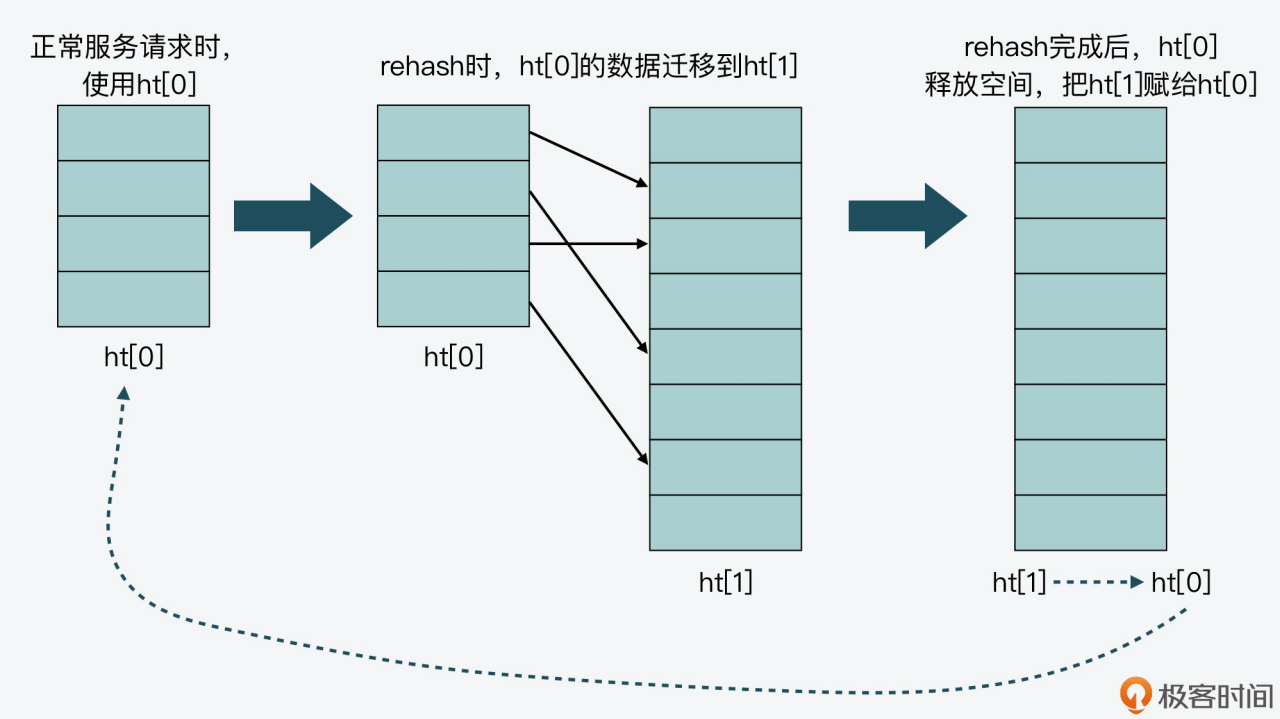
- 首先，Redis准备了两个哈希表，用于rehash时交替保存数据。

我在前面给你介绍过，Redis在dict.h文件中使用dictht结构体定义了Hash表。不过，在实际使用Hash表时，Redis又在dict.h文件中，定义了一个dict结构体。这个结构体中有一个数组（ht[2]），包含了两个Hash表ht[0]和ht[1]。dict结构体的代码定义如下所示：

```
typedef struct dict {
    …
    dictht ht[2]; //两个Hash表，交替使用，用于rehash操作
    long rehashidx; //Hash表是否在进行rehash的标识，-1表示没有进行rehash
    …
} dict;
```

- 其次，在正常服务请求阶段，所有的键值对写入哈希表ht[0]。
- 接着，当进行rehash时，键值对被迁移到哈希表ht[1]中。
- 最后，当迁移完成后，ht[0]的空间会被释放，并把ht[1]的地址赋值给ht[0]，ht[1]的表大小设置为0。这样一来，又回到了正常服务请求的阶段，ht[0]接收和服务请求，ht[1]作为下一次rehash时的迁移表。

这里我画了一张图，以便于你理解ht[0]和ht[1]交替使用的过程。



好，那么在了解了Redis交替使用两个Hash表实现rehash的基本思路后，我们还需要明确的是：在实现rehash时，都需要解决哪些问题？我认为主要有以下三点：

- 什么时候触发rehash?
- rehash扩容扩多大?
- rehash如何执行?

所以下面，我就带你来逐一学习Redis对这三个问题的代码实现，通过代码实现，你就能明晰Redis针对这三个问题的设计思想了。

什么时候触发rehash?

首先要知道，Redis用来判断是否触发rehash的函数是 **_dictExpandIfNeeded**。所以接下来我们就先看看，_dictExpandIfNeeded函数中进行扩容的触发条件；然后，我们再来了解下_dictExpandIfNeeded又是在哪些函数中被调用的。

实际上，_dictExpandIfNeeded函数中定义了三个扩容条件。

- 条件一：ht[0]的大小为0。
- 条件二：ht[0]承载的元素个数已经超过了ht[0]的大小，同时Hash表可以进行扩容。
- 条件三：ht[0]承载的元素个数，是ht[0]的大小的dict_force_resize_ratio倍，其中，

dict_force_resize_ratio的默认值是5。

下面的代码就展示了_dictExpandIfNeeded函数对这三个条件的定义，你可以看下。

```
//如果Hash表为空，将Hash表扩为初始大小
if (d->ht[0].size == 0)
   return dictExpand(d, DICT_HT_INITIAL_SIZE);

//如果Hash表承载的元素个数超过其当前大小，并且可以进行扩容，或者Hash表承载的元素个数已是当前大小的5倍
if (d->ht[0].used >= d->ht[0].size &&(dict_can_resize ||
             d->ht[0].used/d->ht[0].size > dict_force_resize_ratio))
{
    return dictExpand(d, d->ht[0].used*2);
}
```

那么，对于条件一来说，此时Hash表是空的，所以Redis就需要将Hash表空间设置为初始大小，而这是初始化的工作，并不属于rehash操作。

而条件二和三就对应了rehash的场景。因为在这两个条件中，都比较了Hash表当前承载的元素个数（d->ht[0].used）和Hash表当前设定的大小（d->ht[0].size），这两个值的比值一般称为**负载因子（load factor）**。也就是说，Redis判断是否进行rehash的条件，就是看load factor是否大于等于1和是否大于5。

实际上，当load factor大于5时，就表明Hash表已经过载比较严重了，需要立刻进行库扩容。而当load factor大于等于1时，Redis还会再判断dict_can_resize这个变量值，查看当前是否可以进行扩容。

你可能要问了，这里的dict_can_resize变量值是啥呀？其实，这个变量值是在dictEnableResize和dictDisableResize两个函数中设置的，它们的作用分别是启用和禁止哈希表执行rehash功能，如下所示：

```
void dictEnableResize(void) {
    dict_can_resize = 1;
}

void dictDisableResize(void) {
    dict_can_resize = 0;
}
```

然后，这两个函数又被封装在了updateDictResizePolicy函数中。

updateDictResizePolicy函数是用来启用或禁用rehash扩容功能的，这个函数调用dictEnableResize函数启用扩容功能的条件是：**当前没有RDB子进程，并且也没有AOF子进程。**这就对应了Redis没有执行RDB快照和没有进行AOF重写的场景。你可以参考下面给出的代码：

```
void updateDictResizePolicy(void) {
    if (server.rdb_child_pid == -1 && server.aof_child_pid == -1)
        dictEnableResize();
    else
```

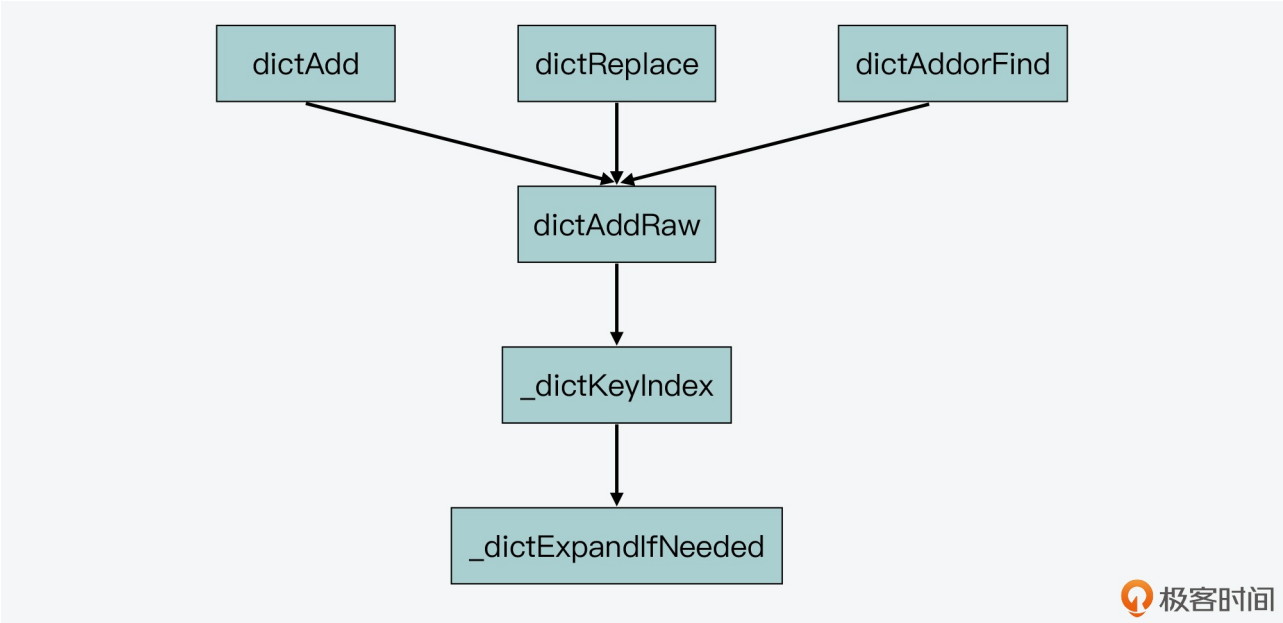
```
        dictDisableResize();
}
```

好，到这里我们就了解了_dictExpandIfNeeded对rehash的判断触发条件，那么现在，我们再来看下Redis会在哪些函数中，调用_dictExpandIfNeeded进行判断。

首先，通过在dict.c文件中查看_dictExpandIfNeeded的被调用关系，我们可以发现，_dictExpandIfNeeded是被_dictKeyIndex函数调用的，而_dictKeyIndex函数又会被dictAddRaw函数调用，然后dictAddRaw会被以下三个函数调用。

- dictAdd：用来往Hash表中添加一个键值对。
- dictRelace：用来往Hash表中添加一个键值对，或者键值对存在时，修改键值对。
- dictAddorFind：直接调用dictAddRaw。

因此，当我们往Redis中写入新的键值对或是修改键值对时，Redis都会判断下是否需要进行rehash。这里你可以参考下面给出的示意图，其中就展示了_dictExpandIfNeeded被调用的关系。



好了，简而言之，Redis中触发rehash操作的关键，就是_dictExpandIfNeeded函数和updateDictResizePolicy函数。**_dictExpandIfNeeded函数**会根据Hash表的负载因子以及能否进行rehash的标识，判断是否进行rehash，而**updateDictResizePolicy函数**会根据RDB和AOF的执行情况，启用或禁用rehash。

接下来，我们继续探讨Redis在实现rehash时，要解决的第二个问题：rehash扩容扩多大？

rehash扩容扩多大？

在Redis中，rehash对Hash表空间的扩容是通过**调用dictExpand函数**来完成的。dictExpand函数的参数有两个，**一个是要扩容的Hash表，另一个是要扩到的容量**，下面的代码就展示了dictExpand函数的原型定义：

```
int dictExpand(dict *d, unsigned long size);
```

那么，对于一个Hash表来说，我们就可以根据前面提到的_dictExpandIfNeeded函数，来判断是否要对其进行扩容。而一旦判断要扩容，Redis在执行rehash操作时，对Hash表扩容的思路也很简单，就是**如果当前表的已用空间大小为size，那么就将表扩容到size*2的大小。**

如下所示，当_dictExpandIfNeeded函数在判断了需要进行rehash后，就调用dictExpand进行扩容。这里你可以看到，rehash的扩容大小是当前ht[0]已使用大小的2倍。

```
dictExpand(d, d->ht[0].used*2);
```

而在dictExpand函数中，具体执行是由_dictNextPower函数完成的，以下代码显示的Hash表扩容的操作，就是从Hash表的初始大小（DICT_HT_INITIAL_SIZE），不停地乘以2，直到达到目标大小。

```
static unsigned long _dictNextPower(unsigned long size)
{
    //哈希表的初始大小
    unsigned long i = DICT_HT_INITIAL_SIZE;
    //如果要扩容的大小已经超过最大值，则返回最大值加1
    if (size >= LONG_MAX) return LONG_MAX + 1LU;
    //扩容大小没有超过最大值
    while(1) {
        //如果扩容大小大于等于最大值，就返回截至当前扩到的大小
        if (i >= size)
            return i;
        //每一步扩容都在现有大小基础上乘以2
        i *= 2;
    }
}
```

好，下面我们再来看看Redis要解决的第三个问题，即rehash要如何执行？而这个问题，本质上就是Redis要如何实现渐进式rehash设计。

渐进式rehash如何实现？

那么这里，我们要先搞清楚一个问题，就是**为什么要实现渐进式rehash？**

其实这是因为，Hash表在执行rehash时，由于Hash表空间扩大，原本映射到某一位置的键可能会被映射到一个新的位置上，因此，很多键就需要从原来的位置拷贝到新的位置。而在键拷贝时，由于Redis主线程无法执行其他请求，所以键拷贝会阻塞主线程，这样就会产生**rehash开销**。

而为了降低rehash开销，Redis就提出了渐进式rehash的方法。

简单来说，渐进式rehash的意思就是Redis并不会一次性把当前Hash表中的所有键，都拷贝到新位置，而是会分批拷贝，每次的键拷贝只拷贝Hash表中一个bucket中的哈希项。这样一来，每次键拷贝的时长有限，对主线程的影响也就有限了。

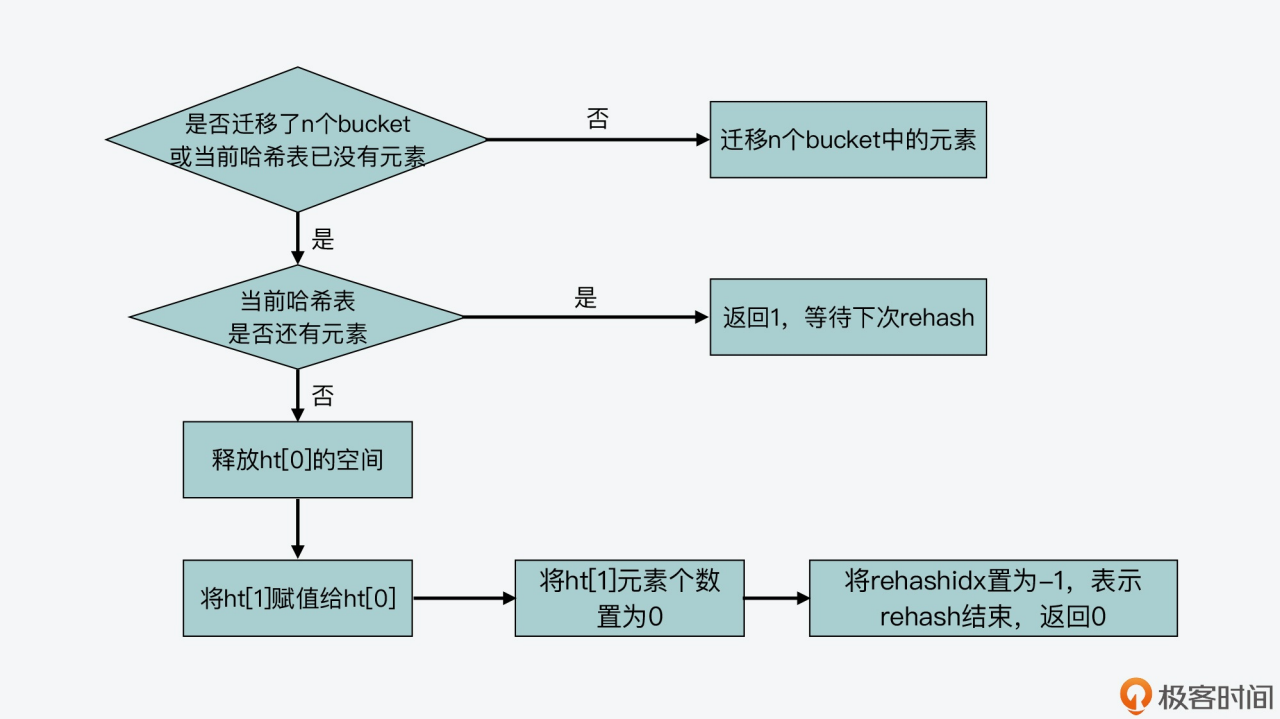
那么，渐进式rehash在代码层面是如何实现的呢？这里有两个关键函数：dictRehash和_dictRehashStep。

我们先来看**dictRehash函数**，这个函数实际执行键拷贝，它的输入参数有两个，分别是全局哈希表（即前面提到的dict结构体，包含了ht[0]和ht[1]）和需要进行键拷贝的桶数量（bucket数量）。

dictRehash函数的整体逻辑包括两部分：

- 首先，该函数会执行一个循环，根据要进行键拷贝的bucket数量n，依次完成这些bucket内部所有键的迁移。当然，如果ht[0]哈希表中的数据已经都迁移完成了，键拷贝的循环也会停止执行。
- 其次，在完成了n个bucket拷贝后，dictRehash函数的第二部分逻辑，就是判断ht[0]表中数据是否都已迁移完。如果都迁移完了，那么ht[0]的空间会被释放。因为Redis在处理请求时，代码逻辑中都是使用ht[0]，所以当rehash执行完成后，虽然数据都在ht[1]中了，但Redis仍然会把ht[1]赋值给ht[0]，以便其他部分的代码逻辑正常使用。
- 而在ht[1]赋值给ht[0]后，它的大小就会被重置为0，等待下一次rehash。与此同时，全局哈希表中的rehashidx变量会被标为-1，表示rehash结束了（这里的rehashidx变量用来表示rehash的进度，稍后我会给你具体解释）。

我画了下面这张图，展示了dictRehash的主要执行流程，你可以看下。



同时，你也可以通过下面代码，来了解dictRehash函数的主要执行逻辑。

```
int dictRehash(dict *d, int n) {
    int empty_visits = n*10;
    ...
```

```
    //主循环，根据要拷贝的bucket数量n，循环n次后停止或ht[0]中的数据迁移完停止
    while(n-- && d->ht[0].used != 0) {
       ...
    }
    //判断ht[0]的数据是否迁移完成
    if (d->ht[0].used == 0) {
        //ht[0]迁移完后，释放ht[0]内存空间
        zfree(d->ht[0].table);
        //让ht[0]指向ht[1]，以便接受正常的请求
        d->ht[0] = d->ht[1];
        //重置ht[1]的大小为0
        _dictReset(&d->ht[1]);
        //设置全局哈希表的rehashidx标识为-1，表示rehash结束
        d->rehashidx = -1;
        //返回0，表示ht[0]中所有元素都迁移完
        return 0;
    }
    //返回1，表示ht[0]中仍然有元素没有迁移完
    return 1;
}
```

好，在了解了dictRehash函数的主体逻辑后，我们再看下渐进式rehash是如何按照bucket粒度拷贝数据的，这其实就和全局哈希表dict结构中的rehashidx变量相关了。

rehashidx变量表示的是**当前rehash在对哪个bucket做数据迁移**。比如，当rehashidx等于0时，表示对ht[0]中的第一个bucket进行数据迁移；当rehashidx等于1时，表示对ht[0]中的第二个bucket进行数据迁移，以此类推。

而dictRehash函数的主循环，首先会判断rehashidx指向的bucket是否为空，如果为空，那就将rehashidx的值加1，检查下一个bucket。

那么，有没有可能连续几个bucket都为空呢？其实是有可能的，在这种情况下，渐进式rehash不会一直递增rehashidx进行检查。这是因为一旦执行了rehash，Redis主线程就无法处理其他请求了。

所以，渐进式rehash在执行时设置了一个**变量empty_visits**，用来表示已经检查过的空bucket，当检查了一定数量的空bucket后，这一轮的rehash就停止执行，转而继续处理外来请求，避免了对Redis性能的影响。下面的代码显示了这部分逻辑，你可以看下。

```
while(n-- && d->ht[0].used != 0) {
    //如果当前要迁移的bucket中没有元素
    while(d->ht[0].table[d->rehashidx] == NULL) {
        //
        d->rehashidx++;
        if (--empty_visits == 0) return 1;
    }
    ...
}
```

而如果rehashidx指向的bucket有数据可以迁移，那么Redis就会把这个bucket中的哈希项依次取出来，并根据ht[1]的表空间大小，重新计算哈希项在ht[1]中的bucket位置，然后把这个哈希项赋值到ht[1]对应

bucket中。

这样，每做完一个哈希项的迁移，ht[0]和ht[1]用来表示承载哈希项多少的变量used，就会分别减一和加一。当然，如果当前rehashidx指向的bucket中数据都迁移完了，rehashidx就会递增加1，指向下一个bucket。下面的代码显示了这一迁移过程。

```
while(n-- && d->ht[0].used != 0) {
    ...
    //获得哈希表中哈希项
    de = d->ht[0].table[d->rehashidx];
    //如果rehashidx指向的bucket不为空
    while(de) {
        uint64_t h;
        //获得同一个bucket中下一个哈希项
        nextde = de->next;
        //根据扩容后的哈希表ht[1]大小，计算当前哈希项在扩容后哈希表中的bucket位置
        h = dictHashKey(d, de->key) & d->ht[1].sizemask;
        //将当前哈希项添加到扩容后的哈希表ht[1]中
        de->next = d->ht[1].table[h];
        d->ht[1].table[h] = de;
        //减少当前哈希表的哈希项个数
        d->ht[0].used--;
        //增加扩容后哈希表的哈希项个数
        d->ht[1].used++;
        //指向下一个哈希项
        de = nextde;
    }
    //如果当前bucket中已经没有哈希项了，将该bucket置为NULL
    d->ht[0].table[d->rehashidx] = NULL;
    //将rehash加1，下一次将迁移下一个bucket中的元素
    d->rehashidx++;
}
```

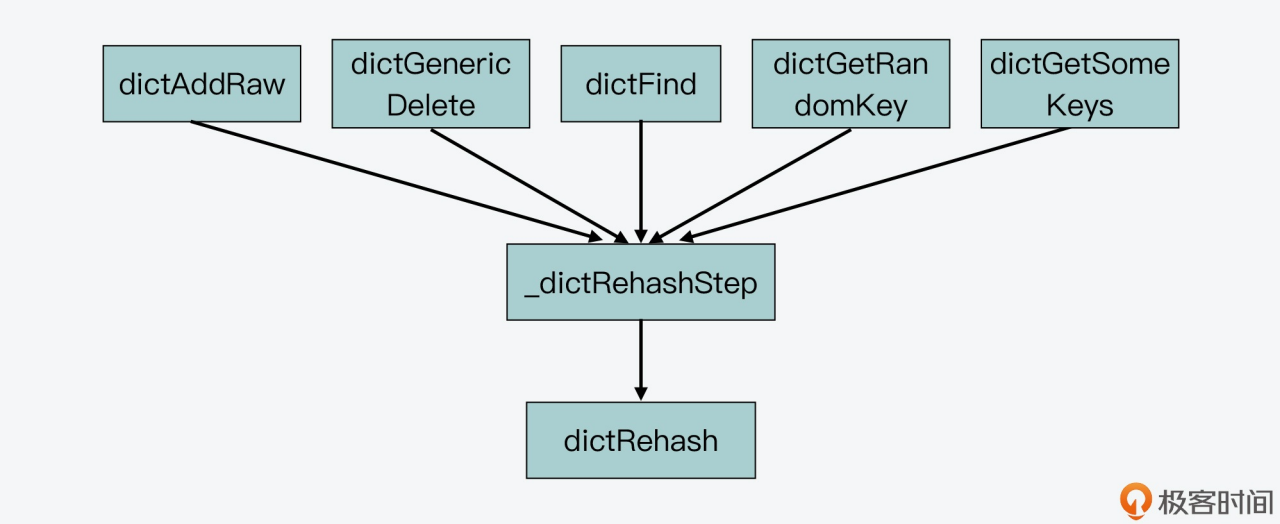
好了，到这里，我们就已经基本了解了dictRehash函数的全部逻辑。

现在我们知道，dictRehash函数本身是按照bucket粒度执行哈希项迁移的，它内部执行的bucket迁移个数，主要由传入的循环次数变量n来决定。但凡Redis要进行rehash操作，最终都会调用dictRehash函数。

接下来，我们来学习和渐进式rehash相关的第二个关键函数 **_dictRehashStep**，这个函数实现了每次只对一个bucket执行rehash。

从Redis的源码中我们可以看到，一共会有5个函数通过调用_dictRehashStep函数，进而调用dictRehash函数，来执行rehash，它们分别是：dictAddRaw，dictGenericDelete，dictFind，dictGetRandomKey，dictGetSomeKeys。

其中，dictAddRaw和dictGenericDelete函数，分别对应了往Redis中增加和删除键值对，而后三个函数则对应了在Redis中进行查询操作。下图展示了这些函数间的调用关系：



但你要注意，不管是增删查哪种操作，这5个函数调用的_dictRehashStep函数，给dictRehash传入的循环次数变量n的值都为1，下面的代码就显示了这一传参的情况。

```
static void _dictRehashStep(dict *d) {
//给dictRehash传入的循环次数参数为1，表明每迁移完一个bucket ，就执行正常操作
    if (d->iterators == 0) dictRehash(d,1);
}
```

这样一来，每次迁移完一个bucket，Hash表就会执行正常的增删查请求操作，这就是在代码层面实现渐进式rehash的方法。

小结

实现一个高性能的Hash表不仅是Redis的需求，也是很多计算机系统开发过程中的重要目标。而要想实现一个性能优异的Hash表，就需要重点解决哈希冲突和rehash开销这两个问题。

今天这节课，我带你学习了Redis中Hash表的结构设计、链式哈希方法的实现，以及渐进式rehash方法的设计实现。Redis中Hash表的结构设计很特别，它的每个哈希项都包含了一个指针，用于实现链式哈希。同时，Redis在全局哈希表中还包含了两个Hash表，这种设计思路也是为了在实现rehash时，帮助数据从一个表迁移到另一个表。

此外，Redis实现的渐进式rehash是一个用于Hash表扩容的通用方法，非常值得我们学习。这个设计方法的关键是每次仅迁移有限个数的bucket，避免一次性迁移给所有bucket带来的性能影响。当你掌握了渐进式rehash这个设计思想和实现方法，你就可以把它应用到自己的Hash表实现场景中。

每课一问

Hash函数会影响Hash表的查询效率及哈希冲突情况，那么，你能从Redis的源码中，找到Hash表使用的是哪一种Hash函数吗？

欢迎在留言区分享你的答案，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

精选留言：

- Kaito 2021-07-31 02:35:49

1、Redis 中的 dict 数据结构，采用「链式哈希」的方式存储，当哈希冲突严重时，会开辟一个新的哈希表，翻倍扩容，并采用「渐进式 rehash」的方式迁移数据

2、所谓「渐进式 rehash」是指，把很大块迁移数据的开销，平摊到多次小的操作中，目的是降低主线程的性能影响

3、Redis 中凡是需要 O(1) 时间获取 k-v 数据的场景，都使用了 dict 这个数据结构，也就是说 dict 是 Redis 中重中之重的「底层数据结构」

4、dict 封装好了友好的「增删改查」API，并在适当时机「自动扩容、缩容」，这给上层数据类型（Hash/Set/Sorted Set）、全局哈希表的实现提供了非常大的便利

5、例如，Redis 中每个 DB 存放数据的「全局哈希表、过期key」都用到了 dict：

```
// server.h
typedef struct redisDb {
dict *dict; // 全局哈希表，数据键值对存在这
dict *expires; // 过期 key + 过期时间 存在这
...
}
```

6、「全局哈希表」在触发渐进式 rehash 的情况有 2 个：

- 增删改查哈希表时：每次迁移 1 个哈希桶（文章提到的 dict.c 中的 _dictRehashStep 函数）
- 定时 rehash：如果 dict 一直没有操作，无法渐进式迁移数据，那主线程会默认每间隔 100ms 执行一次迁移操作。这里一次会以 100 个桶为基本单位迁移数据，并限制如果一次操作耗时超时 1ms 就结束本次任务，待下次再次触发迁移（文章没提到这个，详见 dict.c 的 dictRehashMilliseconds 函数）

（注意：定时 rehash 只会迁移全局哈希表中的数据，不会定时迁移 Hash/Set/Sorted Set 下的哈希表的数据，这些哈希表只会在操作数据时做实时的渐进式 rehash）

7、dict 在负载因子超过 1 时（used: bucket size >= 1），会触发 rehash。但如果 Redis 正在 RDB 或 AOF rewrite，为避免父进程大量写时复制，会暂时关闭触发 rehash。但这里有个例外，如果负载因子超过了 5（哈希冲突已非常严重），依旧会强制做 rehash（重点）

8、dict 在 rehash 期间，查询旧哈希表找不到结果，还需要在新哈希表查询一次

课后题：Hash 函数会影响 Hash 表的查询效率及哈希冲突情况，那么，你能从 Redis 的源码中，找到 Hash 表使用的是哪一种 Hash 函数吗？

找到 dict.c 的 dictFind 函数，可以看到查询一个 key 在哈希表的位置时，调用了 dictHashKey 计算 key 的哈希值：

```
dictEntry *dictFind(dict *d, const void *key) {
// 计算 key 的哈希值
h = dictHashKey(d, key);
...
}
```

继续跟代码可以看到 dictHashKey 调用了 struct dict 下 dictType 的 hashFunction 函数：

```
// dict.h
dictHashKey(d, key) (d)->type->hashFunction(key)
```

而这个 hashFunction 是在初始化一个 dict 时，才会指定使用哪个哈希函数的。

当 Redis Server 在启动时会创建「全局哈希表」：

```
// 初始化 db 下的全局哈希表
for (j = 0; j < server.dbnum; j++) {
// dbDictType 中指定了哈希函数
server.db[j].dict = dictCreate(&dbDictType,NULL);
...
}
```

这个 dbDictType struct 指定了具体的哈希函数，跟代码进去能看到，使用了 SipHash 算法，具体实现逻辑在 siphash.c。

（SipHash 哈希算法是在 Redis 4.0 才开始使用的，3.0-4.0 使用的是 MurmurHash2 哈希算法，3.0 之前是 DJBX33A 哈希算法） [2赞]