dict.c中的dictScan函数，用来遍历字典，迭代其中的每个元素。该函数使用的算法非常精妙！！！所以必须记录一下。

遍历一个稳定的字典，当然不是什么难事，但Redis中的字典因为有rehash的过程，使字典可能扩展，也可能缩小。这就带来了问题，如果在两次遍历中间，字典的结构发生了变化（扩展或缩小），字典中的元素所在的位置相应的会发生变化，那如何保证字典中原有的元素都可以被遍历？又如何能尽可能少的重复迭代呢？

这就是该算法的精妙所在，使用该算法，可以做到下面两点：

a：开始遍历那一刻的所有元素，只要不被删除，肯定能被遍历到，不管字典扩展还是缩小；

b：该算法可能会返回重复元素，但是已经把返回重复元素的可能性降到了最低;

**一：游标cursor的演变**

该算法使用了游标cursor来遍历字典，它表示本次要访问的bucket的索引。bucket中保存了一个链表，因此每次迭代都会把该bucket的链表中的所有元素都遍历一遍。

第一次迭代时，cursor置为0，dictScan函数的返回值作为下一个cursor再次调用dictScan，最终，dictScan函数返回0表示迭代结束。

首先看一下cursor的演变过程，也是该算法的核心所在。这里cursor的演变是采用了reverse binary iteration方法，也就是每次是向cursor的最高位加1，并向低位方向进位。

下面具体解释，首先，根据dictScan写一个简单的测试函数，用来看cursor的演变过程：

void test\_dictScan\_cursor**(**int tablesize**)**

**{**

unsigned long v**;**

unsigned long m0**;**

v **=** 0**;**

m0 **=** tablesize**-**1**;**

printbits**(**v**,** **(**int**)**log2**(**tablesize**));**

printf**(**" --> "**);**

**do**

**{**

v **|=** **~**m0**;**

v **=** rev**(**v**);**

v**++;**

v **=** rev**(**v**);**

printbits**(**v**,** **(**int**)**log2**(**tablesize**));**

printf**(**" --> "**);**

**}while** **(**v **!=** 0**);**

printf**(**"\b\b\b\b\b \n"**);**

**}**

参数tablesize表示哈希表的大小，printbits用来打印v的低n二进制位，n等于log2(tablesize)，以tablesize为8和16分别运行该函数，结果如下：

000 **-->** 100 **-->** 010 **-->** 110 **-->** 001 **-->** 101 **-->** 011 **-->** 111 **-->** 000

0000 **-->** 1000 **-->** 0100 **-->** 1100 **-->** 0010 **-->** 1010 **-->** 0110 **-->** 1110 **-->** 0001 **-->** 1001 **-->** 0101 **-->** 1101 **-->** 0011 **-->** 1011 **-->** 0111 **-->** 1111 **-->** 0000

这就是所谓的reverse binary iteration方法，也就是每次是向v的最高位加1，并向低位方向进位。比如1101的下一个数是0011，因为1101的前三个数为110，最高位加1，并且向低位进位就是001，所以最终得到0011。

在Redis中，字典的哈希表长度始终为2的n次方。因此m0始终是一个低n位全为1，其余为全为0的数。整个计算过程，都是在v的低n位数中进行的，比如长度为16的哈希表，则n=4，因此v是从0到15这几个数之间的转换。下面解释一下计算过程：

第一步：v |= ~m0; //用于保留v的低n位数，其余位全置为1：

1

. . .

1

1

. . .

第二步：v = versebits2(v); //将v的二进制位进行翻转，所以，v的低n位数成了高n位数，并且进行了翻转：

. . .

1

1

. . .

. . .

1

第三步：v++;

. . .

0

. . .

. . .

0

0

**进位方向**

最后一步：v = versebits2(v); //再次翻转

. . .

**进位方向**

因此，最终得到的新v，就是向最高位加1，且向低位方向进位。

**二：为什么要这样**

这样设计的原因就在于，字典中的哈希表有可能扩展，也有可能缩小。在字典不稳定的情况下，既要遍历到所有没被删除的元素，又要尽可能较少的重复遍历。

下面详细解释一下这样设计的好处，以及为什么不是按照正常的0,1,2,...这样的顺序迭代？

计算一个哈希表节点索引的方法是hashkey&mask，其中，mask的值永远是哈希表大小减1。哈希表长度为8，则mask为111，因此，节点的索引值就取决于hashkey的低三位，假设是abc。如果哈希表长度为16，则mask为1111，同样的节点计算得到的哈希值不变，而索引值是?abc，其中?既可能是0，也可能是1，也就是说，该节点在长度为16的哈希表中，索引是0abc或者1abc。以此类推，如果哈希表长度为32，则该节点的索引是00abc，01abc，10abc或者11abc中的一个。

重新看一下该算法中，哈希表长度分别为8和16时，cursor变化过程：

000 **-->** 100 **-->** 010 **-->** 110 **-->** 001 **-->** 101 **-->** 011 **-->** 111 **-->** 000

0000 **-->** 1000 **-->** 0100 **-->** 1100 **-->** 0010 **-->** 1010 **-->** 0110 **-->** 1110 **-->** 0001 **-->** 1001 **-->** 0101 **-->** 1101 **-->** 0011 **-->** 1011 **-->** 0111 **-->** 1111 **-->** 0000

哈希表长度为8时，第i个cursor（0 <= i <= 7），扩展到长度为16的哈希表中，对应的cursor是2i和2i+1，它们是相邻的，这点很重要。

首先是字典扩展的情况，假设当前字典哈希表长度为8，在迭代完索引为010的bucket之后，下一个cursor为110。假设在下一次迭代前，字典哈希表长度扩展成了16，110这个cursor，在长度为16的情况下，就成了0110，因此开始迭代索引为0110的bucket中的节点。

在长度为8时，已经迭代过的cursor分别是：000，100，010。哈希表长度扩展到16后，在这些索引的bucket中的节点，分布到新的bucket中，新bucket的索引将会是：0000，1000，0100，1100，0010，1010。而这些，正好是将要迭代的0110之前的索引，从0110开始，按照长度为16的哈希表cursor变化过程迭代下去，这样既不会漏掉节点，也不会迭代重复的节点。

再看一下字典哈希表缩小的情况，也就是由16缩小为8。在长度为16时，迭代完0100的cursor之后，下一个cursor为1100，假设此时哈希表长度缩小为8。1100这个cursor，在长度为8的情况下，就成了100。因此开始迭代索引为100的bucket中的节点。

在长度为16时，已经迭代过的cursor是：0000，1000，0100，哈希表长度缩小后，这些索引的bucket中的节点，分布到新的bucket中，新bucket的索引将会是：000和100。现在要从索引为100的bucket开始迭代，这样不会漏掉节点，但是之前长度为16时，索引为0100中的节点会被重复迭代，然而，也就仅0100这一个bucket中的节点会重复而已。

原哈希表长度为x，缩小后长度为y，则最多会有x/y – 1个原bucket的节点会被重复迭代。比如由16缩小为8，则最多就有1个bucket节点会重复迭代，要是由32缩小为8，则最多会有3个。

当然也有可能不产生重复迭代，还是从16缩小为8的情况，如果已经迭代完1100，下一个cursor为0010，此时长度缩小为8，cursor就成了010。

长度为16时，已经迭代过的cursor为0000，1000，0100，1100，长度缩小后，这些cursor对应到新的索引是000和100，正好是010之前的索引，从010开始，按照长度为8的cursor走下去，不会漏掉节点，也不会重复迭代节点。

所以说这种算法，保证了：能迭代完所有节点而不会漏掉；又能尽可能较少的重复遍历。

如果按照正常的顺序迭代，下面分别是长度为8和16对应的cursor变化过程：

000 **-->** 001 **-->** 010 **-->** 011 **-->** 100 **-->** 101 **-->** 110 **-->** 111 **-->** 000

0000 **-->** 0001 **-->** 0010 **-->** 0011 **-->** 0100 **-->** 0101 **-->** 0110 **-->** 0111 **-->** 1000 **-->** 1001 **-->** 1010 **-->** 1011 **-->** 1100 **-->** 1101 **-->** 1110 **-->** 1111 **-->** 0000

字典扩展的情况，当前字典哈希表长度为8，假设在迭代完cursor为010的bucket之后，下一个cursor为011。迭代011之前，字典长度扩展成了16，011这个cursor，在长度为16的情况下，就成了0011，因此开始迭代索引为0011的bucket中的节点。

在长度为8时，已经迭代过的cursor是：000，001，010。哈希表长度扩展到16后，这些索引的bucket中的节点，会分布到新的bucket中，新bucket的索引将会是：0000，1000，0001，1001，0010和1010。现在要开始迭代的cursor为0011，而1000，1001，1010这些bucket中的节点在后续还是会遍历到，这就产生了重复遍历。

虽然这种情况不会发生漏掉节点的情况，但是肯定会有重复的情况发生，而且长度变化发生的时机越晚，重复遍历的节点越多，比如长度为8时，迭代完110后，下一个cursor为111，长度扩展为16后，这个cursor就成了0111。

长度为8时，已经迭代过的cursor为000，001，010，011，100，101，110，扩展到长度为16的哈希表中，这些bucket中的节点会分布到索引为：0000，1000，0001，1001，0010，1010，0011，1011，0100，1100，0101，1101，0110，1110。现在长度为16，要开始迭代cursor为0111，而1000，1001，1010，1011和1110这些节点后续还会遍历到，重复的节点增多了。

再看一下长度缩小的情况，长度由16缩小为8。在长度为16时，迭代完0100的cursor之后，下一个cursor为0101，此时长度缩小为8。0101这个cursor，在长度为8的情况下，就成了101。

在长度为16时，尚未迭代过的cursor是：0101，0110，0111，1000，1001，1010，1011，1100，1101，1110，1111。这些cursor，在哈希表长度缩小后，分配到新的bucket中，索引将会是：000，001，010，011，100，101，110，111。现在要开始迭代的cursor为101，那101之前的000，001，010，011，100这些cursor就不会迭代了，这样，原来的某些节点就被漏掉了。

另外，还是从16缩小为8的情况，如果已经迭代完1100，下一个cursor为1101，在长度为8的情况下，就成了101。

长度为16时，已经迭代过的cursor为0000，0001，0010，0011，0100，0101，0110，0111，1000，1001，1010，1011，1100。这些cursor，在哈希表长度缩小后，分配到新的bucket中，索引分别是：000，001，010，011，100，101，110，111。长度变为8后，从101开始，很明显，原来已经迭代过的0101，0110，0111就会产生重复迭代。

因此，顺序迭代不是一个满足要求的迭代方法。

上面的算法是由Pieter Noordhuis设计实现的，Redis之父Salvatore Sanfilippo对该算法的评价是” Hard to explain but awesome.”，可见其牛逼！！！Pieter Noordhuis对该算法的解释见：https://github.com/antirez/redis/pull/579#issuecomment-16871583

了解完该算法的核心之后，剩下的就是具体的迭代过程了，dictScan代码如下：

unsigned long dictScan**(**dict **\***d**,**

unsigned long v**,**

dictScanFunction **\***fn**,**

void **\***privdata**)**

**{**

dictht **\***t0**,** **\***t1**;**

const dictEntry **\***de**;**

unsigned long m0**,** m1**;**

**if** **(**dictSize**(**d**)** **==** 0**)** **return** 0**;**

**if** **(!**dictIsRehashing**(**d**))** **{**

t0 **=** **&(**d**->**ht**[**0**]);**

m0 **=** t0**->**sizemask**;**

*/\* Emit entries at cursor \*/*

de **=** t0**->**table**[**v **&** m0**];**

**while** **(**de**)** **{**

fn**(**privdata**,** de**);**

de **=** de**->**next**;**

**}**

**}** **else** **{**

t0 **=** **&**d**->**ht**[**0**];**

t1 **=** **&**d**->**ht**[**1**];**

*/\* Make sure t0 is the smaller and t1 is the bigger table \*/*

**if** **(**t0**->**size **>** t1**->**size**)** **{**

t0 **=** **&**d**->**ht**[**1**];**

t1 **=** **&**d**->**ht**[**0**];**

**}**

m0 **=** t0**->**sizemask**;**

m1 **=** t1**->**sizemask**;**

*/\* Emit entries at cursor \*/*

de **=** t0**->**table**[**v **&** m0**];**

**while** **(**de**)** **{**

fn**(**privdata**,** de**);**

de **=** de**->**next**;**

**}**

*/\* Iterate over indices in larger table that are the expansion*

*\* of the index pointed to by the cursor in the smaller table \*/*

**do** **{**

*/\* Emit entries at cursor \*/*

de **=** t1**->**table**[**v **&** m1**];**

**while** **(**de**)** **{**

fn**(**privdata**,** de**);**

de **=** de**->**next**;**

**}**

*/\* Increment bits not covered by the smaller mask \*/*

v **=** **(((**v **|** m0**)** **+** 1**)** **&** **~**m0**)** **|** **(**v **&** m0**);**

*/\* Continue while bits covered by mask difference is non-zero \*/*

**}** **while** **(**v **&** **(**m0 **^** m1**));**

**}**

v **|=** **~**m0**;**

*/\* Increment the reverse cursor \*/*

v **=** rev**(**v**);**

v**++;**

v **=** rev**(**v**);**

**return** v**;**

**}**

其中的rev函数用来对无符号整数进行二进制位的翻转，具体算法参考《翻转整数的二进制位》，这里不再赘述。

如果字典当前没有rehash，则比较简单，直接根据v找到需要迭代的bucket索引，针对该bucket中链表中的所有节点，调用用户提供的fn函数。

如果字典当前正在rehash，则需要先遍历较小的哈希表，然后是较大的哈希表。

首先使t0指向小表，t1指向大表；m0为小表的mask，m1为大表的mask。

根据v&m0，找到t0中需要迭代的bucket，然后迭代其中的每个节点即可。

接下来的代码稍显复杂，但是，本质上，就是t0中，索引为v&m0的bucket中的所有节点，再其扩展到t1中后，遍历其所有可能的bucket中的节点。语言不好描述，举个例子就明白了：若t0长度为8，则m0为111，v&m0就是保留v的低三位，假设为abc。若t1长度为32，则m1为11111，该过程就是：遍历完t0中索引为abc的bucket之后，接着遍历t1中，索引为00abc、01abc、10abc、11abc的bucket中的节点。

下面是抽取核心代码的逻辑而写的测试代码：

void test\_dictScan\_iter**(**int smalltablesize**,** int largetablesize**)**

**{**

unsigned long v**;**

unsigned long m0**,** m1**;**

v **=** 0**;**

m0 **=** smalltablesize**-**1**;**

m1 **=** largetablesize**-**1**;**

**do**

**{**

printf**(**"\nsmall v is: "**);**

printbits**(**v **&** m0**,** **(**int**)**log2**(**smalltablesize**));**

printf**(**"\n"**);**

**do**

**{**

printf**(**"large v is: "**);**

printbits**(**v **&** m1**,** **(**int**)**log2**(**largetablesize**));**

printf**(**"\n"**);**

v **=** **(((**v **|** m0**)** **+** 1**)** **&** **~**m0**)** **|** **(**v **&** m0**);**

**}while** **(**v **&** **(**m0 **^** m1**));**

v **|=** **~**m0**;**

v **=** rev**(**v**);**

v**++;**

v **=** rev**(**v**);**

**}while** **(**v **!=** 0**);**

**}**

以test\_dictScan\_iter(8, 32);运行代码，结果如下：

small v is**:** 000

large v is**:** 00000

large v is**:** 01000

large v is**:** 10000

large v is**:** 11000

small v is**:** 100

large v is**:** 00100

large v is**:** 01100

large v is**:** 10100

large v is**:** 11100

small v is**:** 010

large v is**:** 00010

large v is**:** 01010

large v is**:** 10010

large v is**:** 11010

small v is**:** 110

large v is**:** 00110

large v is**:** 01110

large v is**:** 10110

large v is**:** 11110

small v is**:** 001

large v is**:** 00001

large v is**:** 01001

large v is**:** 10001

large v is**:** 11001

small v is**:** 101

large v is**:** 00101

large v is**:** 01101

large v is**:** 10101

large v is**:** 11101

small v is**:** 011

large v is**:** 00011

large v is**:** 01011

large v is**:** 10011

large v is**:** 11011

small v is**:** 111

large v is**:** 00111

large v is**:** 01111

large v is**:** 10111

large v is**:** 11111

可见，无论v取何值，只要字典开始扩展了，都会遍历大表中，相应于小表的所有节点。具体的核心逻辑代码如下：

**do**

**{**

de **=** t1**->**table**[**v **&** m1**];**

**...**

v **=** **(((**v **|** m0**)** **+** 1**)** **&** **~**m0**)** **|** **(**v **&** m0**);**

**}while** **(**v **&** **(**m0 **^** m1**));**

首先迭代t1中，索引为v&m1的bucket，接下来的语句：

v = (((v | m0) + 1) & ~m0) | (v & m0); 就是对v的低m1-m0位加1，并保留v的低m0位。循环条件v & (m0 ^ m1)，表示直到v的低m1-m0位到低m1位之间全部为0为止。

（完）