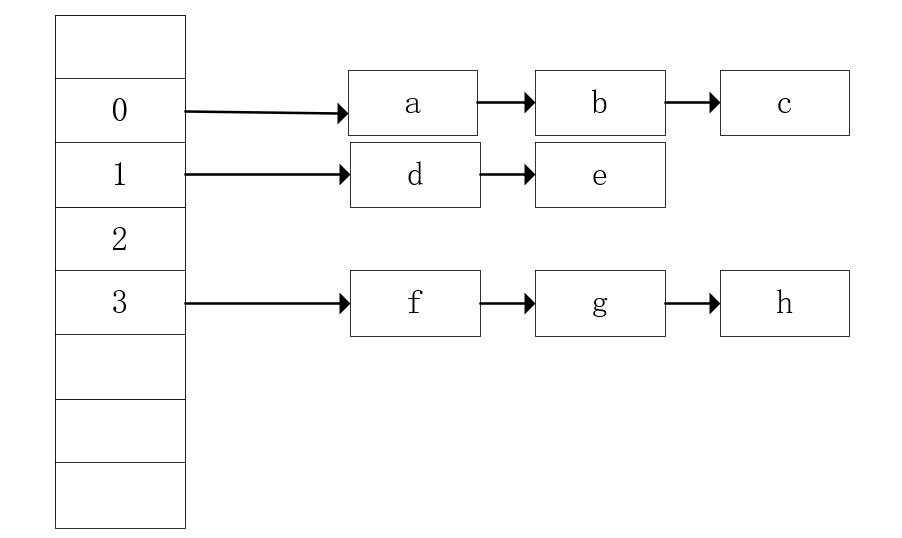
经常会说到HashMap在多线程下是不安全的，那么不安全会引起什么问题呢？

多线程下，对同一个HashMap进行修改操作时，会造成元素丢失或者链表闭环。

1、HashMap的存储结构

首先看下HashMap的存储结构，HashMap的存储结构是Entry数组+链表的结构,如下图



2、先说一下元素丢失是怎么引起的

上图中，两个线程分别插入元素 g和h，经过hash计算，插入位置都是数组索引为3的链表中，g和h分别将到f的next指向自身（JDK8版本），g和h分别拿到尾结点f，g先将f的next指向g，接着h又将f的next做了修改指向h。最终，g节点丢失，f的next指向h。

这是多线程下使用临界资源经常面临的一个问题，很容易理解。

3、那链表闭环又是怎么引起的呢，往链表中添加元素为什么会引起链表闭环呢？

这需要先从HashMap的扩容说起了。

HashMap是为了快速查询而存在的容器，为了能够快速查询，采用了通过空间换时间的方法。HashMap为了方便查询，使用了Hash处理，快速定位元素数组所在下标位置。常规状态下HashMap中数组长度永远大于元素个数的。

那么问题来了，数组长度是不可变的，如果元素数量超过loadFactor\*capacity的数值，那么HashMap就需要重新申请一个新的数组，并把原来存储的元素转移到新的数组中。数组的长度变了，就需要重新计算元素的Hash位置。（不了解这块知识点的可以打开链接，看一下HashMap中Hash计算和Hash碰撞的问题）。

扩容这个过程中发生了什么呢，让我们一块看一下resize这个函数

**void** resize(**int** newCapacity) {

Entry[] oldTable = table;

**int** oldCapacity = oldTable.length;

**if** (oldCapacity == *MAXIMUM\_CAPACITY*) {

threshold = Integer.*MAX\_VALUE*;

**return**;

}

Entry[] newTable = **new** Entry[newCapacity];

transfer(newTable, initHashSeedAsNeeded(newCapacity));

table = newTable;

threshold = (**int**)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);

}

/\*\*

\* Transfers all entries from current table to newTable.

\*/

**void** transfer(Entry[] newTable, **boolean** rehash) {

**int** newCapacity = newTable.length;

**for** (Entry<K,V> e : table) {

**while**(**null** != e) {

Entry<K,V> next = e.next;

**if** (rehash) {

e.hash = **null** == e.key ? 0 : hash(e.key);

}

**int** i = *indexFor*(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

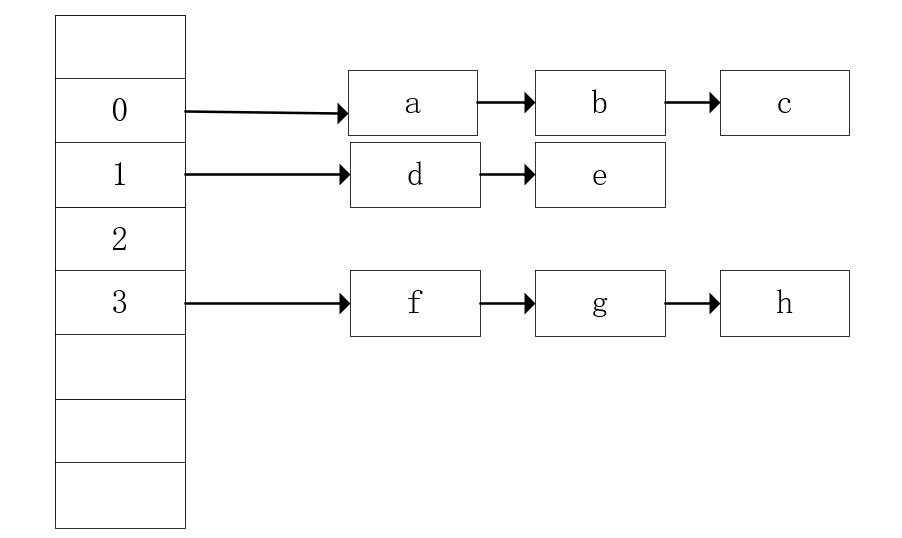
}

}

}

resize这个函数很容易理解，就是申请了一个新的Entry数组newTable，进行空间扩容，然后接下来调用transfer 函数，将原来的数组中的链表节点元素，排列到新的Entry数组中。

在多线程下这个过程为什么会出现链表闭环呢？两个线程一起调用resize,分别对HashMap进行扩容。继续以这个图作为例子



假设HashMap现在容量是32，进行扩容到64，线程A和B都先申请一个新的64长度的数组，然后移动原来的元素，A先移动了f和g放到索引是n的位置，f的next指向g，B将元素f,g也放到索引是n的位置，不过是将g的next指向f。

那么现在出问题了，f的next指向g，g的next指向f，扩容完成。查询h元素，经过Hash计算，找到索引为n的数组元素，遍历链表查找f然后next操作一直在g和f之间循环，进入闭环链表中了，HashMap查询出问题了。

注意：JDK8以后HashMap使用红黑树，已经不会出现闭环的问题了，不过元素丢失还是没法避免。

HashMap会引起这些问题，最主要的原因是对临界资源的处理没有加锁。

4、解决方案

ConcurrentHashMap

JDK7采用分片加锁的方案，支持多线程操作一个Map容器

JDK8采用红黑树+数组+链表的结构，通过synchronized+CAS保证了线程安全，并实现了多线程协助扩容，加快扩容速度