第十章 Cuckoo hash

10.1 引例：Memcached 内存缓存

淘宝、京东是深受大家喜爱的购物平台。打开一个淘宝界面，输入某件商品名称，界面便会显示该商品图片、价格、评价等信息。从用户角度看似简单的操作，其实需要淘宝后台复杂的技术支持，这其中最为关键的两大技术就是存储和检索。

在购物网站中，当商品数量庞大时，单一的服务器主机已经无法满足存储要求，此时只能增加服务器数量，将商品分布的存储在多台远程服务器中，并将主机和远程服务器通过高速网络相连。当客户端向服务器主机发送请求时，主机通过网络向远程服务器发送数据库请求，再用返回的结果来处理客户请求。

但频繁的访问数据库操作会导致很大的时间开销，造成请求延迟。为了解决该问题，很多购物网站都引入了内存缓存（memory cache）技术。该技术会在主机内存中开辟一部分空间用于缓存部分商品信息，这些商品往往被高频率访问。这样每次在访问远程数据库前，主机都先访问存储在本地内存中的缓存，如果缓存中没有相应内容，则再访问远程数据库，并将远程数据库返回的内容写入到内存缓存中，以便下次访问。访问本地的内存缓存会远远快于远程服务器，因此这项技术可以有效的提高访问性能。图 10.1.1 展示了某购物平台在使用 memory cache 技术后如何访问后台数据。

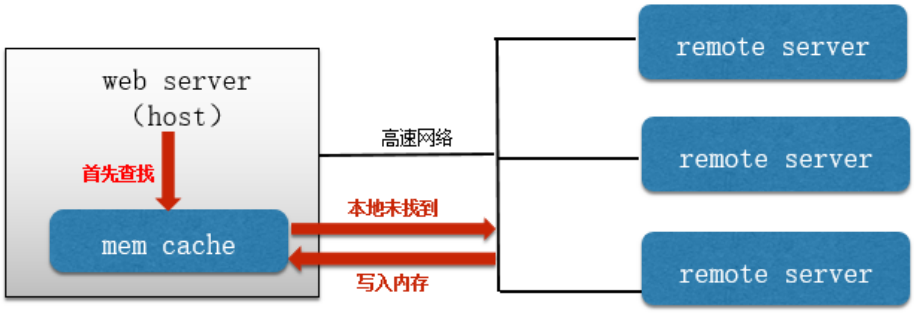

图 10.1.1 访问数据库操作

考虑下面一个现实场景：有一家专门出售服装的网站，该网站之前将商品全部存储在远程数据库中，每次检索某件商品时，都需要访问远程服务器。一件商品有 id，颜色，尺寸，价格这些属性，这些都被存储在如下图所示的数据库表中。

表 10.1.2 数据库中的商品信息

id(主键)	颜色	尺寸	价格

现在为了减少检索开销，该网站决定引入 memory cache 技术，那么如何将数据库中的商品信息缓存至主机内存中呢？首先应该考虑的就是如何表示每一件商品。每一件商品都应该有能唯一标识自己的键，用于查找，这一点数据库里的主键可以满足，所以数据库表里的 id 就是键。其他属性颜色、尺寸、价格可以打包至结构体 struct info 中，作为键对应的值。这样每件商品都可以表示成键值对的形式，键用于查找，值包含了商品的所有信息。

例如当用户询问 id=1 的商品颜色时，主机先在 memory cache 中查找 id=1 的键值对，找到该 id 对应的值后，即 info 结构体，再从该结构体中解析出颜色字段，返回给用户。

那么怎么存储这些键值对呢？我们知道 memory cache 的重要应用就在于它能够快速查找，为了满足该特性，所使用的结构和查找算法一定要快，cuckoo hash 是能满足该要求的一个好方法，因此，我们可以将一件商品以键值对(key, value)的形式存储在内存里的 cuckoo hash 中，key 和数据库中的主键 id 一致，能够唯一表示一件商品，value 集合了颜色、尺寸、样式这些基本属性。

当多个用户同时发起查找请求时，保证主机的响应时间不受影响最为关键，这就要求主机程序具有并行查找 cuckoo hash 的能力；一般情况下，在用户发起请求时，大多数时候都不会发生对 hash table 的插入、修改，但不排除个别时间存在这样的情况，因此在 cuckoo hash 中，如何在查找、插入同时发生时，既确保数据的一致性又能确保效率又是一大挑战；更极端情况就是多个插入操作同时发生时，又该如何处理？下面的介绍将会重点解决这些问题。

数据库主键

在数据库中，主键由一个或多个属性组成，用来唯一标识数据库表中的一条记录。可以把数据库里的表想象成 excel 表格，一般的 excel 表格都是由头部字段和记录组成。例如统计某个班级的学生信息时，学号、姓名、年龄、家庭住址就构成表格中的字段，班级中每个学生的信息都是表格中的一条记录。字段在数据库表中被称为属性，属性学号可以唯一标识一名学生，因此就可以作为数据库表里的主键。

响应时间

响应时间是指从用户提交请求到系统给出应答的时间。例如淘宝网站中，响应时间就指的是从用户提交某件商品名称开始到界面返回商品信息结束之间的时间。

10.2 Cuckoo hash 基本思想

常规的一些 hash 方法：线性 hash 和链式 hash，这些 hash 方法确实能够提高查找速度，但很难 memory cache 的要求。例如在线性 hash 中，某个键的位置是不确定的，这些位置构成一个探测序列，每次查找一个键时，都需要顺序遍历探测序列，才能确定是否存在该键；链式 hash 将 hash 值相同的键存放在同一个链表中，查找某个键时也需要遍历一次链表。显然常规的 hash 方法都不能满足 memory cache 对查找速度的要求。Cuckoo hash 的重要特点就是能够保证很高的查找效率，因此很适用于 memory cache 系统。

10.2.1 基本组成

Cuckoo hash 的基本组成是 2 个 hash 函数和一个 hash table，并且两个 hash 函数会确保将某个键映射至 table 中的不同位置，也就是说对于任意键 k，h1(k)≠h2(k)。一个键仅可能出现在 table 中的 h1(k)位置或 h2(k)位置，这两个位置中的唯一一个。

10.2.2 基本操作

查找(get)操作：在 cuckoo hash 中，因为一个键仅可能出现在 table 中的 h1(k)位置或者 h2(k)位置，所以查找时仅需要探测这两个位置。

插入(put)操作：和其它 hash 方法一样，cuckoo hash 避免不了插入时的冲突。对于某个键 k，如果 h1(k)位置发生冲突，则查看 h2(k)位置，为空则将 k 插入至 h2(k)位置，但如果 h2(k)非空呢？

也就是说，当一个键在 hash table 中的两个位置都被其它键占据，该如何解决冲突呢？就像 cuckoo 这个名字暗示的一样，解决这类冲突的方法就是踢出。例如下图 hash table 中已存在 A、B、D 键，新来的键 C 满足 h1(C)=h1(A)且 h2(C)=h1(B)，首先 C 会踢出 h1(C)位置的 A，A 被踢出；A 查看另一位置 h2(A)是否为空，发现该位置已经存在键 D，键 A 踢出键 D；键 D 发现 h2(D)位置为空，将自己插入到该位置，踢出过程结束。

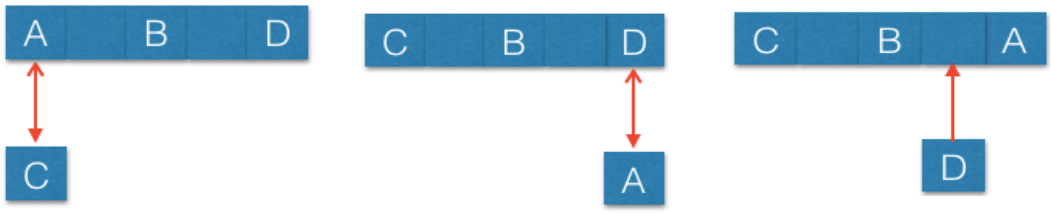

图 10.2.1 插入 C 后的踢出过程

但也有可能出现这样一种异常情况：最后一个被踢出的元素永远无法找到一个空位置，这样整个踢出过程便无法终止。无法终止的踢出过程都会形成一个环，但需要注意的是并不是所有产生环的踢出过程都不能终止，在 10.5.4 会详细说明环的情况。

cuckoo 意为布谷鸟，布谷鸟会偷偷的在其它鸟的巢穴中产蛋，当布谷鸟幼崽孵化出来后，这些幼崽便会将其它幼鸟踢出巢穴，以获得更大的生存空间。

10.2.3 键值分离存储

以上介绍的都是如何存储键，那么如何存储值呢，以及如何建立起键和值的关联呢？一种简单的方法就是将键和值存储到一起，将<key, value>对打包至结构体中一起存到 hash table 里。当查找某个键时，其实就是查找包含该键的键值对结构体。但这种方法要求 value 的大小固定，不太灵活，并且会耗费一部分 hash table 的存储空间，此外在进行踢出操作时，每次移动的都是 struct 结构体，会十分耗时。

另外一种方法就是实现键值分离存储，将值存储在另外的内存空间中，hash table 中每一项存储的是<key, address>，address 记录了值所在的地址。在 32(或 64)位机器中， address 占 4（或 8）个字节的大小，可以指向不同大小的 value 空间，这样做即灵活又能节省 hash table 的存储空间。

10.3 数据结构

下面代码定义了一个大小为 50 的 cuckoo hash table，两个 hash 函数分别为 hash1 和 hash2，get 就是查找操作，put 为插入操作。

```
1   #include <stdio.h>
2   #include <assert.h>
3   #include <string.h>
4   #include <thread>
5   #include <mutex>
6   #define SIZE (50)
7   namespace cuckoo{
8   typedef int KeyType;
9   class Cuckoo{
10  protected:
11      std::mutex mtx;
12      KeyType T[SIZE];
13      // hash key by hash func 1
14      int hash1(const KeyType &key);
15      // hash key by hash func 2
16      int hash2(const KeyType &key);
17      // find key by hash func 1 in T, exist return key otherwise 0
18      KeyType get1(const KeyType &key);
19      // find key by hash func 2 in T, exist return key otherwise 0
20      KeyType get2(const KeyType &key);
21      void bt_evict(const KeyType &key, int which, int pre_pos);
22  public:
23      Cuckoo();
24      //~Cuckoo();
25      KeyType get(const KeyType &key);
26      void put(const KeyType &key);
27  };
28  };
```

10.4 get 实现

10.4.1 基本实现

查找操作即 get，查找方法是：先在 hash table 中的 h1(k)位置查找，如果有返回键；否则在 h2(k)位置查找，如果有返回键，没有返回空。下面是它的具体实现：

```
1     #include "cuckoo.h"
2     namespace cuckoo{
3     Cuckoo::Cuckoo(){
4         memset(T, 0, sizeof(KeyType) * SIZE);
5     }
6     //~Cuckoo();
7     int Cuckoo::hash1(const KeyType &key){
8         assert(SIZE != 0);
9         int half_siz = SIZE / 2;
10        return key%half_siz;
11    }
12    int Cuckoo::hash2(const KeyType &key){
13        assert(SIZE != 0);
14        int half_siz = SIZE / 2;
15        return key/half_siz%half_siz + half_siz;
16    }
17    // find key by hash func 1 in T, exist return key otherwise 0
18    KeyType Cuckoo::get1(const KeyType &key){
19        return (T[hash1(key)] == key)?key:0;
20    }
21    // find key by hash func 2 in T, exist return key otherwise 0
22    KeyType Cuckoo::get2(const KeyType &key){
23        return (T[hash2(key)] == key)?key:0;
24    }
25    KeyType Cuckoo::get(const KeyType &key){
26        // 0 is reserved for null, invalid input
27        if(key == 0){
28            printf("invalid key\n");
29            return 0;
30        }
31        KeyType result = get1(key);
32        if(result == 0){
33            result = get2(key);
34        }
35        return result;
36    }
37    };
```

10.4.2 并行 get

当多个 get 请求同时发生时，因为该操作不会修改任何数据，所以不需要额外的同步机制，上述代码不变。下面代码中，开辟了 10 个线程，并行的查找一组键。

```
1     #include <vector>
```

```
#include "cuckoo.cpp"
using namespace cuckoo;
static const int TOTAL = 10;
int main(int argc, char* argv[]){
    Cuckoo test;
    // single-thread to put [1, TOTAL]
    for(int i = 1; i <= TOTAL; ++i){
        test.put(i);
    }
    // create multiple threads to get in parallel
    std::vector<std::thread> threads;
    threads.clear();
    for(int i = 1; i <= TOTAL; ++i){
        threads.emplace_back([&](int thread_id){
                printf("thread: %d get %d\n", thread_id, test.get(thread_id));
        }, i);
    }
    for(int i = 0; i < TOTAL; ++i){
        threads[i].join();
    }
    return 0;
}
```

10.5 put 实现

10.5.1 基本实现

如何实现踢出过程是 put 操作的重点，下面代码的黑色部分就是踢出的过程：

```
#include "cuckoo.h"
namespace cuckoo{
template <typename T>
inline void swap(T* a, T* b){
    assert(a != NULL && b != NULL);
    T tmp = *a;
    *a = *b;
    *b = tmp;
}
void Cuckoo::put(const KeyType &key){
    if(key == 0){
        printf("invalid key\n");
        return;
    }
    if(get(key) != 0){
        printf("duplicate key, put fail\n");
        return;
    }
    // basic way
    if(T[hash1(key)] == 0){
        T[hash1(key)] = key;
    }else if(T[hash2(key)] == 0){
```

```
            T[hash2(key)] = key;
        }else{ // two place for one certain key has been occupied, need evict others
            // basic way
            KeyType evicted = key;
            // determine which pos hash1 or hash2 to put key
            // 0 is hash1, 1 is hash2
            int which = 0;
            // first evict key in hash1
            int idx = hash1(evicted);
            // != 0 means place has been occupied
            // if there is a cycle, maybe cannot terminate
            int pre_pos = -1;
            while(T[idx] != 0){
                printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
                swap(&evicted, &T[idx]);
                pre_pos = idx;
                which = 1 - which;
                idx = (which == 0)?hash1(evicted):hash2(evicted);
            }
            printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
            T[idx] = evicted;
        }
    }
    };
```

在黑色部分代码中，evicted 保存当前被踢出的 key，idx 保存将要存放 evicted 的新位置，当新位置 idx 不为空时，踢出过程需要继续进行，swap 操作交换了 evicted 和 idx 位置的键，evicted 被插入到 idx 位置上，之前 idx 位置的键成为下一个被踢出的元素，while 循环会一直执行直到为被踢出的键找到一个空位置。

基于回溯的实现

在很多购物网站中，对键值对的查询远远多于对键值对的修改，因此，Put 操作要远远少于 Get 操作。在该场景下，可以使用串行的 Put 操作(单次只发送一个 Put 请求)，从而避免并行 Put 时多个线程之间的同步代价。但此时还有一个问题就是如果沿用之前的踢出方法，Put 和 Get 仍然需要同步。假设在插入键 A 之后，产生的踢出序列为 A->B->C->D->nil，nil 代表一个空位置，按照之前的操作方法：先将 A 放到 B 所在的位置，B 成为一下个被踢出的元素，在将 B 插入到 C 所在位置之前，两个 hash table 中都不存在键 B，如果此时正好产生了一个 Get(B)的请求且没有同步机制，得到的结果就是不存在该键，这显然是不对的，这就是 Put 和 Get 需要同步的理由。

我们都知道同步是需要时间代价的，那么有没有一种方法，在串行 Put 和多个并行 Get 同时发生时，不需要同步也不会影响 Get 的返回结果？换种说法就是，有没有一种踢出方法，可以保证被踢出元素一直存在在 hash table 中？答案是可以的，回溯法就是其中一种。

提到回溯大家可能第一个想到的就是递归。递归是基于回溯思想的一种算法结构，例如下面这个求解阶乘的简单示例：

```
    int fac(int n) {
        if(n==1)
            return n;
        else
            return n * fac(n-1);
    }
```

1. 当 n!=1 时，程序会不断的向下调用，形成一个没有分叉的递归调用树（这里的没有分叉指的是每个父节点都有且仅有一个子节点）。当 n==1 时，程序从调用树的叶子节点返回计算结果，并且每一层都会向调用层返回自己这一层的计算结果，到达根节点时便会得到最终结果。

对于上面上个踢出序列 A->B->C->D->nil，如果从上向下(从 A 到 D)踢出各个元素，便可能产生某个元素不在 hash table 中的问题，那如果反过来呢？先踢出 D，依次向上直到 A，我们便可以发现：在保证将某个键插入到指定位置的操作是原子的前提下，就可以确保这些元素始终在 hash table 里。Cuckoo 数据结构中定义的 bt_evict 就是此操作，具体实现可以使用递归法：

```
1     #include "cuckoo.h"
2     namespace cuckoo{
3     void Cuckoo::bt_evict(const KeyType &key, int which, int pre_pos){
4         int idx = (which == 0)?hash1(key):hash2(key);
5         // basic case: find a empty pos for the last evicted element
6         if(T[idx] == 0){
7             printf("evicted key %d from %d to %d\n", key, pre_pos, idx);
8             T[idx] = key;
9             return;
10        }
11        printf("evicted key %d from %d to %d\n", key, pre_pos, idx);
12        KeyType cur = T[idx];
13        // first evict latter elements
14        bt_evict(cur, 1 - which, idx);
15        T[idx] = key;
16    }
17
18    void Cuckoo::put(const KeyType &key){
19        if(key == 0){
20            printf("invalid key\n");
21            return;
22        }
23        if(get(key) != 0){
24            printf("duplicate key, put fail\n");
25            return;
26        }
27        // basic way
28        if(T[hash1(key)] == 0){
29            T[hash1(key)] = key;
30        }else if(T[hash2(key)] == 0){
31            T[hash2(key)] = key;
32        }else{ // two place for one certain key has been occupied, need evict others
33            // backtrace way
34            bt_evict(key, 0, -1);
35        }
36    }
37    };
```

并行 put

Put 操作用来向 hash table 中插入键值对。与并行 get 不同，当多个 put 请求同时发生时，因为发生了对数据的修改，就需要同步机制来确保数据的一致性。

Cuckoo 数据结构体中定义了一个 mutex 对象 mtx，负责同步多个 put 请求。如下面代码所示，std::unique_lock<std::mutex>用来管理该 mutex 对象：一旦创建 unique_lock 实例，该实例便自动获取

mtx 对象，进入锁住状态，直到该实例被析构时，才自动释放 mtx 锁。因此，通过 std::unique_lock<std::mutex> lck 的保护，可以确保黑色加粗部分有且仅有一个线程执行，从而保证了数据一致性。

```
#include "cuckoo.h"
namespace cuckoo{
void Cuckoo::put(const KeyType &key){
    if(key == 0){
        printf("invalid key\n");
        return;
    }
    if(get(key) != 0){
        printf("duplicate key, put fail\n");
        return;
    }
    // basic way
    if(T[hash1(key)] == 0){
        T[hash1(key)] = key;
    }else if(T[hash2(key)] == 0){
        T[hash2(key)] = key;
    }else{ // two place for one certain key has been occupied, need evict others
        // lock way
        // need lock for write-operations
        std::unique_lock<std::mutex> lck(mtx);

        KeyType evicted = key;
        // determine which pos hash1 or hash2 to put key
        // 0 is hash1, 1 is hash2
        int which = 0;
        // first evict key in hash1
        int idx = hash1(evicted);
        // != 0 means place has been occupied
        // if there is a cycle, maybe cannot terminate
        int pre_pos = -1;
        while(T[idx] != 0){
            printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
            swap(&evicted, &T[idx]);
            pre_pos = idx;
            which = 1 - which;
            idx = (which == 0)?hash1(evicted):hash2(evicted);
        }
        printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
        T[idx] = evicted;
      }
}
};
```

循环路径

下面这段代码向 table 中依次插入键 1 至 28，代码如下：

```
#include <vector>
```

```
2       #include "cuckoo.cpp"
3       using namespace cuckoo;
4       static const int TOTAL = 28;
5       int main(int argc, char* argv[]){
6           Cuckoo test;
7           // single-thread to put [1, TOTAL]
8           for(int i = 1; i <= TOTAL; ++i){
9               test.put(i);
10          }
11          return 0;
12      }
```

在插入键 28，程序陷入死循环，程序输出了踢出操作中键的转移过程，通过分析这些转移，我们可以发现产生了无法终止的循环路径。

程序输出如下：

```
1 evicted key 28 from -1 to 3
2 evicted key 3 from 3 to 25
3 evicted key 2 from 25 to 2
4 evicted key 27 from 2 to 26
5 evicted key 26 from 26 to 1
6 evicted key 1 from 1 to 25
7 evicted key 3 from 25 to 3
8 evicted key 28 from 3 to 26
9 evicted key 27 from 26 to 2
10 evicted key 2 from 2 to 25
11 evicted key 1 from 25 to 1
12 evicted key 26 from 1 to 26
13 evicted key 28 from 26 to 3
14 evicted key 3 from 3 to 25
15 evicted key 2 from 25 to 2
```

图 10.5.1 程序输出

我们可以借用图来表示这些转移过程。首先将 hash table 中所有位置抽象成图里的顶点，并按照位置的先后顺序给顶点编号。如果一个被踢出的键从一个位置移除，存储至另一个位置，那么图中这两个位置对应的顶点之间就有一条有向边，从原始位置指向新位置。最后图表示如下：

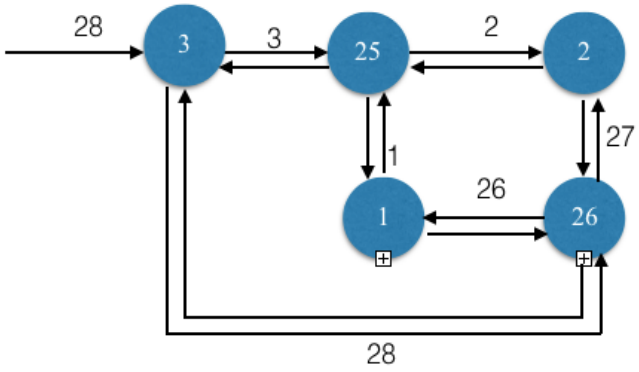

图 10.5.2 插入 28 后的踢出过程图

例如刚插入键 28 时，键 3 被踢出，从位置 3 转移到位置 25，于是图中有一条从顶点 3 指向顶点 25 的边，边的值为 3，代表被踢出的键为 3。在为了更加清楚的看出循环路径，我们对上图进行进一步简化。

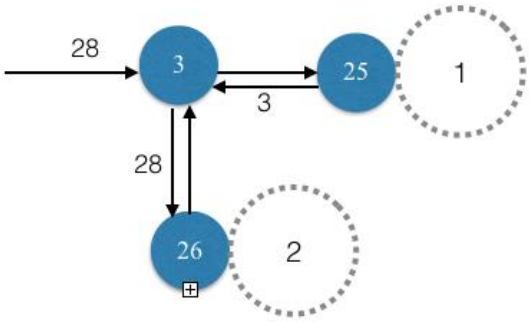

图 10.5.3 简化图

虚线环 1 代表图中 3 到 6 行的转移过程，虚线环 2 代表图中 9 到 12 行的转移过程，可以发现键 28 在两个可选位置 3 和 26 上都各自形成了一个环，也就是说无论 28 暂时放在哪个位置，稍后都会被其他键踢出，并在这两个位置之间不断往复，无法终止。

上面这个例子描述了无法终止的环，让我们再考虑另外一种可以终止的环。下面这个例子中，在插入 A 后，踢出了位置 0 的 B，B 踢出了 C，C 又反过来踢出了刚插入位置 0 的 A，使得图上形成了环，A 只能被存储到 h2(A)上，踢出了 D，D 发现自己的可选位置 h2(D)为空，就直接存储在该位置 上，整个过程得以终止。

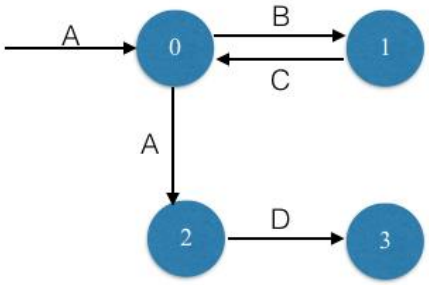

图 10.5.4 可以终止的环

通过以上两个例子的对比可以发现：只有当一个键的两个可选位置都各自形成一个环结构时，才会导致整个过程无法终止，例 2 虽然产生了环，但当 A 被踢出到 h2(A)位置上时，并不会形成环，也就是说可以为最后一个被踢出的键找到一个为空的位置。

在具体实现中，检测循环路径的方法也比较简单，可以预先设定一个阈值(threshold)，当循环次数或者递归调用次数超过阈值时，就可以认为产生了循环路径。一旦发生循环路径之后，常规方法就是进行 rehash 操作。

10.6 性能分析

为了证明 cuckoo hash 确实能取得优越的查找性能，本章对 cuckoo hash 和链式 hash 进行比较。

链式 hash 是一种实现简单、性能较好的 hash 策略。它将 hash 值相同的键插入到同一

个 bucket 里并用链表实现，之后记录下链表的地址，这样在下次查找该 hash 值的某个键时，便可以通过该地址，迅速找到键所在的链表。但是由于无法确定该键在链表中的确切位置，所以需要遍历操作，也就是从表头到表尾顺序扫描一遍（如果找到该键就提前结束）。可以发现，影响链式 hash 查找速度最主要的因素就是链表的遍历。

在查找某个键 k 时，cuckoo hash 会查看 h1(k)和 h2(k)两个位置的键，并将这两个位置的键与 k 比较，当查找一组键时，这些比较次数的累加就是决定 cuckoo hash 性能的主导因素。同理，链式 hash 也需要将链表中的键与 k 进行比较，查找一组键时，比较总次数越少，链式 hash 查找速度越快。所以为了对比两种 hash 方法的查找性能，我们可以简单的比较它们在查找一组键时的比较次数。为了便于观察，这里采用平均比较次数，例如查找一组键 kset，键总数为|kset|，查找过程中产生的总比较次数为 C，那么平均比较次数的计算公式如下：

$$\text{平均比较次数} = \frac{C}{|kset|}$$

下表对两种 hash 方法的平均比较次数做了对比：

表 10.6.1

查找键总数 / Hash 方法	50	250	375	500
Cuckoo hash	1	1	1.33	1.5
链式 hash	1	3	4.67	5.5

一共进行了四组对比实验，每次查找的键的总数分别为 50、250、375、500。当然查找每组键之前，必须确保它们都已经被插入到两种方法的 hash 表中。Cuckoo hash 的总容量限制为 500 个键，链式 hash 因为采取动态分配 list 的方法，所以容量不受限制（在计算机内存容量以内），buckets 的总数为 50。

通过上表的数据可以发现，随着查找键数量的增多，两种方法的平均比较次数都有增加，但 cuckoo hash 的增加速度要显著慢于链式 hash。最开始时，链式 hash 能取得和 cuckoo hash 一样的查找性能，这是因为链式方法使用了 50 个 buckets，插入 50 个键时冲突较少，每个 bucket 中大约只有一个键，查找这些键时自然也就比较快。但随着键的总数越来越多，键的 hash 值冲突也就越多，每个 bucket 的长度也会有所增加，最终导致查找时比较次数的增多。但 cuckoo hash 不同，因为查找某个键时，它最多只会访问两个位置的键，所以每次的比较次数不会超过 2，平均比较次数当然就在 2 以内。从表中也可以发现，当 cuckoo hash 的负载因子分别为 0.10、0.50、0.75、1.00 时，平均比较次数都维持在 2 以内。

所以，当查找键的数量越多时，cuckoo hash 的性能优势就越明显。

负载因子

负载因子，英文为 load factor，是用来衡量 hash 表空/满的程度，计算公式为：

$$\text{负载因子} = \frac{\text{键的总数}}{\text{总容量}}$$

10.7 本章小结

本章从 memory cache 技术着手，引入了 cuckoo hash 的基本概念。本章着重介绍了 cuckoo hash 的两个常用操作：查找(get)和插入(put)。cuckoo hash 的查找操作简单，并且对比其他 hash 方法能取得较好的性能。插入时可能产生踢出操作，但只要 hash table 的内存空间足够大，就能减少冲突次数，从而减少踢出次数。此外，本章也对插入过程中可能产生的循环路径问题作了说明，目前该问题没有很好的解决方法，只能进行 rehash 操作。

10.8 练习题

本章的示例代码中并没有存储值，而是简单的用键代替了值。请你设计一个能真正存储键值对的cuckoo hash系统。有如下两种方案：

1，键值对合并存储在hash table中。

2，键值分离存储（详见10.2.3）。

类的声明如下：

```
// 键值对合并存储
class CuckooPair{
protected:
    std::mutex mtx;
    KeyValue T[SIZE];
    // 请设计你自己的第一个hash函数
    int hash1(const KeyType &key);
    // 请设计你自己的第二个hash函数
    int hash2(const KeyType &key);
    // find key-value by hash func 1 in T, exist return key
    // otherwise 0
    KeyValue get1(const KeyType &key);
    // find key-value by hash func 2 in T, exist return key
    // otherwise 0
    KeyValue get2(const KeyType &key);
    void bt_evict(const KeyType &key, int which, int pre_pos);
public:
    Cuckoo();
    //~Cuckoo();
    // 根据键key返回相应的键值对
    KeyValue get(const KeyType &key);
    // 插入键值对kv
    void put(const KeyValue &kv);
};
// 键值分离存储
class CuckooSeparate {
protected:
    std::mutex mtx;
```

```
    KeyAddr T[SIZE];
    // 使用vector存储键值对，键值对在vector中的位置就是它的地址
    std::vector<KeyValue> kv_store;
    // 请设计你自己的第一个hash函数
    int hash1(const KeyType &key);
    // 请设计你自己的第二个hash函数
    int hash2(const KeyType &key);
    // find key-value by hash func 1 in T, exist return key
    // otherwise 0
    KeyValue get1(const KeyType &key);
    // find key-value by hash func 2 in T, exist return key
    // otherwise 0
    KeyValue get2(const KeyType &key);
    void bt_evict(const KeyType &key, int which, int pre_pos);
public:
    Cuckoo();
    //~Cuckoo();
    // 根据键key返回相应的键值对
    KeyValue get(const KeyType &key);
    // 插入键值对kv
    void put(const KeyValue &kv);
};
```

以上两种方法中，KeyValue结构体存储的是键和值，KeyAddr结构体存储的是键和值的地址，如果用vector存储键值对的话，值的地址就可以表示成vector中的位置，请你合理的设计自己的KeyValue和KeyAddr结构体。

Project说明如下：

1， 请你根据以上类的声明，分别实现两种方法，尽量避免踢出操作时产生循环路径。

2， 为了避免循环路径的问题，一种可行的输入数据如下：

hash1	Key%(SIZE/2)
hash2	Key/(SIZE/2)+(SIZE/2)
Key(int type)	[1, SIZE/2], {y|y=x*(SIZE/2)+x} x=1,2…(SIZE/2)

当hash1、hash2分别为表中所示时，插入表中两段区间的key值将不会发生任何死循环，并可以插满整个hash table。你可以通过这种方法生成键，并随机生成每个键对应的值。

3， 比较两种方案的运行时间：请分别在负载因子为0.1、0.5、1.0时，比较两种方法查询一组键值对query_set的时间（请自行确定query_set的大小和内容）。

负载因子	方法1	方法2
0.1		
0.5		
1.0		

10.9 文献阅读

[Litwin, Witold (1980)] 和 [*Cormen, Thomas H.;等（2009）*] 分别对线性 hash、链式 hash 作了详细介绍，是很好的参考资料。Cuckoo hash 的思想最初来源于论文[Pagh, Rasmus; Rodler, Flemming Friche (2001)]，论文中也包含了详细的理论证明。[https://web.stanford.edu/class/cs166/lectures/13/Small13.pdf]对循环路径问题作了很好的说明。