假设你们班级100个同学每个人的学号是由院系-年级-班级和编号组成，例如学号为01100168表示是1系，10级1班的68号。为了快速查找到68号的成绩信息，可以建立一张表，但是不能用学号作为下标，学号的数值实在太大。因此将学号除以1100100取余，即得到编号作为该表的下标，那么，要查找学号为01100168的成绩的时候，只要直接访问表下标为68的数据即可。这就能够在O(1)时间复杂度内完成成绩查找。实际上这里就用到了散列的思想。

# 概述

## 特点

哈希函数的输入域可以是非常大的范围，但是输出域是固定范围，假设为s，哈希函数的性质：

1. 典型的哈希函数都拥有无限的输入值域；
2. 输入值相同时，返回值一样；
3. 输入值不同时，返回值可能一样，也可能不一样；
4. 不同输入值得到的哈希值，整体均匀的分布在输出域s上（重要！）。

1~3性质是哈希函数的基础，第4点是评价一个哈希函数优劣的关键。例如，对于“aaa1”、“aaa2”、“aaa3”虽然非常相似，但是计算出的哈希值差异巨大。

不同输入值得到的哈希值越均匀分布在s上，该哈希函数越优秀。

图示

中度可信度描述已自动生成

**注：MD5和SHA1算法都是经典的哈希函数算法。**

### 优点

通过关键字实现一对一查找速度非常快。

### 缺点

存在冲突。

## 哈希表 vs 符号表

符号表是一种抽象数据类型，而哈希表是一种实现符号表的具体数据结构。

# 分类

<https://blog.csdn.net/daliaojie/article/details/7166288>

哈希表分类

├── 传统类

│ ├── 链式哈希

│ ├── 线性探测

│ ├── 二次探测

│ └── 双重散列

├── 现代优化类

│ ├── SIMD控制字节类

│ │ ├── Abseil FlatMap

│ │ └── F14 Map

│ ├── 探测距离平衡类

│ │ └── Robin Hood Hashing

│ ├── 多位置替换类

│ │ └── Cuckoo Hashing

│ ├── 邻近跳跃类

│ │ └── Hopscotch Hashing

│ ├── 并发类

│ │ ├── ConcurrentHashMap

│ │ └── DashMap

│ └── 极致压缩类

│ ├── DenseHashMap

│ └── SparseHashMap

## 传统算法

表格

低可信度描述已自动生成

### 静态哈希

特点：拥有固定slot（桶）数

如果数据是**固定**不变的，那么静态哈希较为实用。

由于这些键固定不变，而且可以提前为静态哈希算法所获知，因此目录中的主要页面数量也会保持稳定。

### 动态哈希

动态哈希表通常是在发生冲突后slot数量翻倍增长，而增长后毕竟哈希函数也变了，所以还要把旧slot里的元素重新放置。这种简单的动态哈希(dynamic hash)算法便是SGI版的STL中hash\_map的实现。

如果数据不固定，那么静态哈希的性能就比较低。这种类型的数据适合采用动态哈希技术来处理。

## 现代优化算法



### FlatMap / SwissTable

**原理关键词：**SIMD、控制字节（Control Bytes）、分组探测

**核心机制：**

利用 SIMD 指令进行批量桶探测，提高查找效率。

每个桶前附带一个「控制字节」表示状态（空、删除、匹配），可以连续比较。

使用分组方式（Group probing）批量比较 16 个桶。

**优点：**

极高的缓存命中率和查找速度。

支持高负载因子（90%+）。

**典型实现：**absl::flat\_hash\_map、std::unordered\_map in LLVM。

### F14

**原理关键词：**SIMD探测、预取优化、容器内嵌值

**核心机制：**

类似 FlatMap 使用 SIMD 优化探测操作。

键值对直接嵌入容器中，减少指针解引用。

使用“探测位置缓存”提升删除与查找效率。

**优点：**

快速插入与查找，兼顾低延迟。

异构查找（如字符串 std::string\_view vs std::string）支持好。

**典型实现：**folly::F14FastMap、F14NodeMap

### Robin Hood Hashing

**原理关键词：**探测距离平衡、最小化方差

**核心机制：**

插入时，如果新元素的探测距离更大，就交换位置，逐步“抢占”靠前位置。

保持元素探测距离接近，减少查找不确定性。

**优点：**

平均查找次数非常稳定。

适合高查找频率的系统。

**典型实现：**robin\_hood::unordered\_map

### Cuckoo Hashing

**原理关键词：**多哈希函数、位置驱逐

**核心机制：**

每个键有两个（或多个）合法位置（通过多个哈希函数）。

插入新键时可能驱逐旧键到其另一位置，类似布谷鸟占窝。

**优点：**

查找操作固定在常数次内（最多两次）。

**缺点：**

插入失败概率高，需重哈希或链式备用。

典型应用：内存受限系统、网络包过滤。

### Hopscotch Hashing

**原理关键词：**邻域保持、本地性、位图加速

**核心机制：**

每个桶维护一个「邻近区域（hop range）」的位图，记录距离本桶多远有数据。

插入时尝试保持新元素在原桶附近，重排列局部元素。

**优点：**

极强的局部性，缓存友好。

适合多线程读写。

**典型实现：**hopscotch\_map from Boost or others

### 并发哈希表类

**原理关键词：**分段锁、原子操作、无锁结构

**核心机制：**

将哈希表划分为多个段（segment），各自独立加锁，降低锁粒度。

或者使用 lock-free 数据结构（如 atomic CAS）实现无锁并发。

**优点：**

支持高并发场景下的读写安全。

**典型实现：**

Java：ConcurrentHashMap

Rust：DashMap

C++：libcds, tbb::concurrent\_hash\_map

### DenseHashMap

**原理关键词：**哨兵键值、数组紧凑存储

**核心机制：**

用两个特殊哨兵键（空、删除）标记桶状态。

内部数据极度紧凑，键值对连续排布。

优点：

空间利用率高，适合静态大数据场景。

**缺点：**

删除成本高，探测长度受限。

**典型实现：**google::dense\_hash\_map

### SparseHashMap

**原理关键词：**指针链 + 块管理、低内存占用

**核心机制：**

桶稀疏分布，使用链表或块结构记录键值。

针对稀疏键空间优化。

**优点：**

内存开销极低。

查找性能仍能接受。

**缺点：**

访问速度慢，不适合密集插入。

**典型实现：**google::sparse\_hash\_map

## 选型

图形用户界面, 文本, 应用程序

描述已自动生成

# 原理

理想散列表（哈希表）是一个包含关键字的具有**固定大小**的数组，它能够以**常数时间执行插入，删除和查找操作**。

·这块连续的存储空间称为散列表或哈希表(hash table)

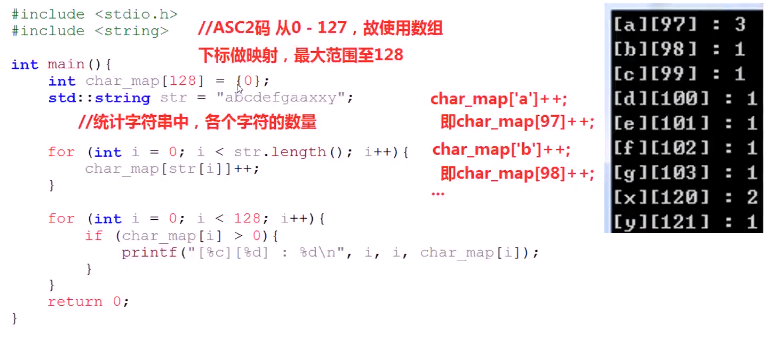
·每个关键字被映射到0到数组大小N-1范围，并且放到合适的位置，这个映射规则就叫散列函数/哈希(hash)函数

·理想情况下，两个不同的关键字映射到不同的单元，然而由于数组单元有限，关键字范围可能远超数组单元，因此就会出现两个关键字散列到同一个值得时候，这就是散列冲突

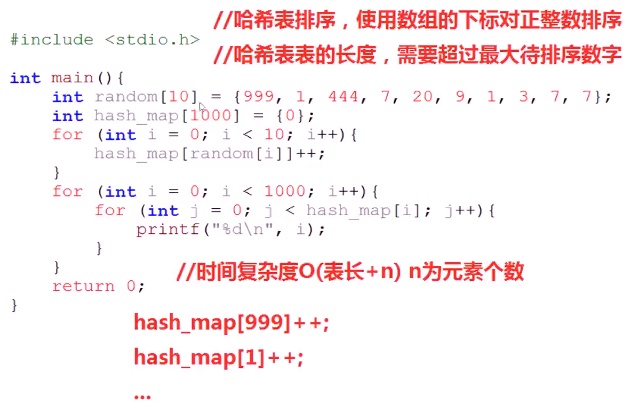
注：当存储或查找记录时，通过散列函数计算出记录的散列地址

那些元素间任何排序信息的操作将不会得到有效的支持，因此，**诸如FindMin、FindMax以及线性时间将排过序的整个表进行打印的操作都是散列所不支持的**。

举例：最简单的哈希-字符哈希



举例：哈希表排序整数



任意元素的映射：

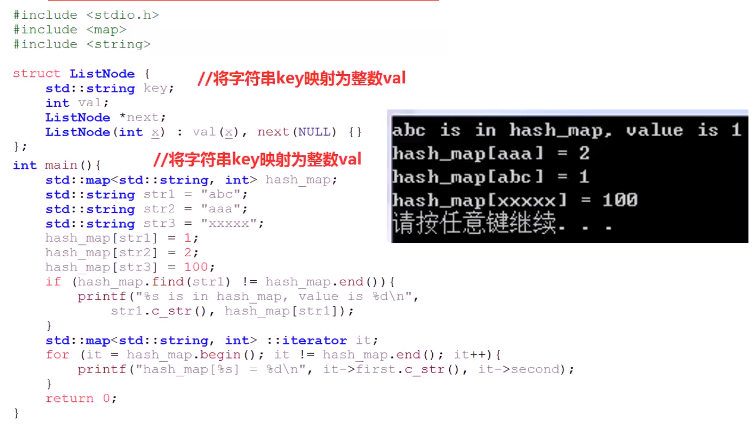
1. 当遇到负数或非常大的整数，如何进行哈希（映射）？如-5、99999999
2. 当遇到字符串，如何进行哈希（映射）？如abcd
3. 当遇其他无法直接映射的数据类型，如浮点数、数组、对象等，如何进行哈希（映射）？如1.23、[1,2,3]

解决：利用哈希函数，将关键字值（key）（大整数、字符串、浮点数等）转换为整数再对表长取余，从而关键字值被转换为哈希表的表长范围内的整数。

冲突：



STL map与hash\_map



## 实例演示

通过前面的描述，我们已经了解了一些基本概念，现在来看一个实例。

假设有一个大小为7的表，现在，要将13，18，19，50，20散列到表中。

·选择散列函数，例如使用hash(x)=x%7作为散列函数

·计算数据散列值，并放到合适的位置

计算13 % 7得到6，因此将13放到下标为6的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  |  |  | 13 |

计算18 % 7得到4，因此将18放到下标为4的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  | 18 |  | 13 |

计算19 % 7得到5，因此将19放到下标为5的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  | 18 | 19 | 13 |

计算50 % 7得到1，因此将50放到下标为1的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  |  |  | 13 |

计算20 % 7得到6，因此将20放到下标为6的位置，但是此时6的位置已经被占用了，因此就产生了散列冲突。

将数据散列之后，如何从表中查找呢？例如，查找数值为50的数据位置，只需要计算50 % 7，得到下标1，访问下标1的位置即可。但是如果考虑散列冲突，就没有那么简单了。

通过这个实例，了解了以下几个概念：

·散列函数，散列函数的选择非常重要

·散列冲突，涉及散列表时，因尽量避免散列冲突，对于冲突也要有好的解决方案

·快速从散列表中查找数据

## 哈希算法

一般的说，Hash函数可以简单的划分为如下几类：

### 加法Hash

### 位运算Hash

### 乘法Hash

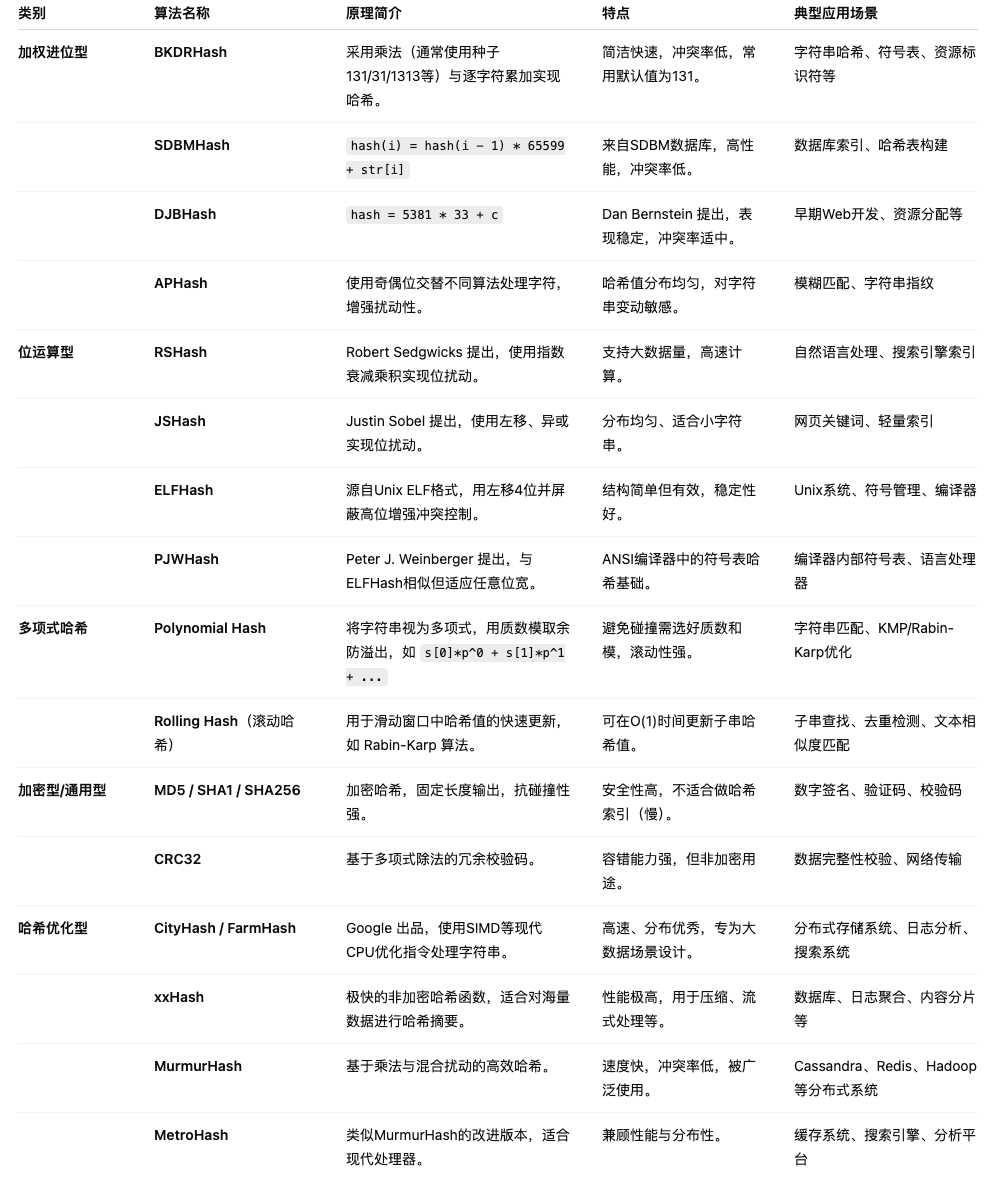
### 除法Hash

### 查表Hash

### 混合Hash

## 字符串哈希算法





### 加权进位型

#### BKDRHash

**参考：**

<https://blog.csdn.net/wanglx_/article/details/40400693>

unsigned int BKDRHash(const char \*str) {

unsigned int seed = 131; // 31, 131, 1313... 常用质数

unsigned int hash = 0;

while (\*str) {

hash = hash \* seed + (\*str++);

}

return (hash & 0x7FFFFFFF);

}

**特点：**

简单，适合小项目；

抗冲突能力弱，不适合高可靠系统。

注：分布式数据库GoldenDB中采用这种哈希算法确定分片信息。

### 位运算型

#### SDBMHash

#### DJBHash

#### APHash

#### RSHash

#### JSHash

#### ElfHash

**思路：**

每次读入一个8位的字符；

先将原h左移4位，再加上读入的字符；

判断最高4位是否为0000（一般在第7次循环时候最高四位出现不全为零的情况）；

若不全为0，则用最高4位与第5-8位进行杂糅；

然后将最高4位改写为0000；

然后进行下一次循环，直至到达字符串的结尾。

**分析：**

一共进行了两种杂糅：

第一种，unsigned char型字符占8位，而h每次左移4位，这样每次读入的新字符的最高4位会与5-8位进行杂糅；

第二种，当循环执行到第7次或第7次之后，最高4位不全为0时，用最高4位和5-8位进行杂糅。

unsigned long ElfHash ( const unsigned char\*name )

{

unsigned long h = 0, g;

while ( \*name )

{

//h左移4位，并在最低8位加上name指向的字符

h = ( h << 4 ) + \*name++;

//如果最高四位不为零，即h存放字符达到7个，第8个字符将无法存放。g存放h最高4位（29-32位）

if ( g = h & 0xF0000000 )

{

//h的5-8位与g的最高四位按位异或

h ^= g >> 24;

}

//清空h的最高四位

h &= ~g;

}

//返回h

return h;

}

#### HashPJW

**思路：**

每次读入一个8位的字符；

先将原h左移4位，再加上读入的字符；

判断最高4位是否为0000（一般在第7次循环时候最高四位出现不全为零的情况）；

若不全为0，则用最高8位与第1-8位进行杂糅；

然后将最高4位改写为0000；

然后进行下一次循环，直至到达字符串的结尾。

**分析：**

一共进行了两种杂糅：

第一种，unsigned char型字符占8位，而h每次左移4位，这样每次读入的新字符的最高4位会与5-8位进行杂糅；

第二种，当循环执行到第7次或第7次之后，最高4位不全为0时，用最高8位和1-8位进行杂糅。

**两算法唯一区别：**

第二种杂糅进行时，ElfHash算法杂糅最高4位和第5-8位；HashPJW算法杂糅最高8位和第1-8位

//int的位数与char位数的乘积

#define BITS\_IN\_int ( sizeof(int) \* CHAR\_BIT )

//BITS\_IN\_int的3/4

#define THREE\_QUARTERS ((int) ((BITS\_IN\_int \* 3) / 4))

//BITS\_IN\_int的1/8

#define ONE\_EIGHTH ((int) (BITS\_IN\_int / 8))

/\*0按位取反位 1111 1111 1111 1111 1111 1111 1111 1111

\*右移ONE\_EIGHTH位 0000 1111 1111 1111 1111 1111 1111 1111

\*再按位取反为 1111 0000 0000 0000 0000 0000 0000 0000

\*/

#define HIGH\_BITS ( ~((unsigned int)(~0) >> ONE\_EIGHTH ))

unsigned int HashPJW ( const char \* datum )

{

unsignedint hash\_value, i;

//循环读入datum

for( hash\_value = 0; \*datum; ++datum )

{

//hash\_value左移4位并在后8位加上datum

hash\_value= ( hash\_value << ONE\_EIGHTH ) + \*datum;

//如果最高4位不全为零，即hash\_value存放字符达到7个，第8个字符将无法存放。

if (( i = hash\_value & HIGH\_BITS ) != 0 )

{

//hash\_value与自己右移24位的结果按位异或并将最高4位全置为0

//结果存入hash\_value

hash\_value=

( hash\_value ^ ( i >>THREE\_QUARTERS )) &

~HIGH\_BITS;

}

}

return( hash\_value );

}

### 多项式哈希

#### Rabin-Karp

用于快速查找字符串中是否包含某一子串。

**原理：**

预先计算每个子串的哈希值，并在滑动窗口中更新。

**基础形式：**

hash = (s[0]\*base^(n-1) + ... + s[n-1]) % mod

**应用：**

适用于文本搜索、重复子串查找。

#### Polynomial Hash

#### Rolling Hash

### 加密型/通用型



#### MD5 / SHA-1 / SHA-256

#### CRC32

#### BLAKE2 / BLAKE3

#### SipHash

**特点：**

由 Google 推出，抗哈希碰撞攻击。

常用于哈希表安全防护（Python、Redis、Rust 标准库）。

### 哈希优化型



#### CityHash / FarmHash

#### MetroHash / HighwayHash

#### xxHash

**特点：**

极致性能（可达 GB/s 级别）

**被广泛用于：**

RocksDB

ClickHouse

Zstandard（压缩算法）

#### MurmurHash

**特点：**

非加密型，速度快、分布好。

无平台依赖，广泛用于数据库和大数据系统。

版本：MurmurHash2、MurmurHash3（推荐）

## 散列函数

构造散列函数的两个基本原则：计算简单+均匀分布

### 直接定址法

我们可以取关键字的某个线性函数值为散列地址，即f(key)=a\*key+b（通过关键字的加减乘除等运算计算地址）。

这种方法简单均匀，但是我们需要事先知道关键字的分布情况，适合查找表比较小且连续的情况，比如按照年月日进行计算。

### 数字分析法

数字分析法通常适合处理关键字位数比较大的情况，例如我们现在要存储某家公司员工登记表，如用手机号作为关键字，那么我们发现抽样后面的四位数字作为散列地址是不错的选择。

### 平方取中法

平方取中法是将关键字平方之后取中间若干位数字作为散列地址。

比较适合于不知道关键字的分布且位数不是很大的情况。

### 折叠法

折叠法是将关键字从左到右分割成位数相等的几部分，然后将这几部分叠加求和，并按散列表表长后几位作为散列地址。

适合于不知道关键字分布且位数较多的情况。

### 除留余数法

除留余数法为最常用的构造散列函数方法，对于散列表长为m的散列函数计算公式为：

f(key) = key mod p(p<=m)

注：p的选择是关键！

事实上，这个方法不仅可以对关键字直接取模，也可以通过折叠、平方取中后再取模。

### 随机数法

选择一个随机数，取关键字的随机函数值为它的散列地址。即：

f(key) = random(key)

这里的random是随机函数，当关键字的长度不等时，采用这个方法构造散列函数是比较合适的。

### 散列函数选择

在实际应用中，我们应该视不同的情况采用不同的散列函数，主要考虑：

1. 计算散列地址所需的时间；
2. 关键字的长度；
3. 散列表的大小；
4. 关键字的分布情况；
5. 记录查找的频率。

## 冲突解决

解决散列冲突通常有以下几种方法：

拉链法

开放定址法

再散列

…

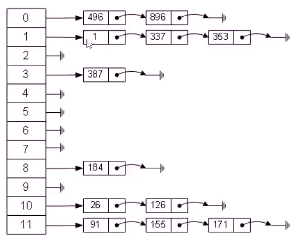
### 分离链接法/拉链法

分离链接法/链地址法的做法是将同一个值的关键字（即哈希结果相同）保存在同一个单链表中。这种方法的特点是需要另外分配新的单元来存储散列到同一个位置的数据。例如，对于前面：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  | 18 | 19 | 13 |

如果再要插入元素20，则在下标为6的位置存储表头，而表的内容是13和20。

若选定的哈希表长度为m，则可将哈希表定义为一个长度为m的指针数组t[0…m-1]，指针数组中的每个指针指向哈希函数结果相同的单链表。



插入值：将元素value插入哈希表，若元素value的哈希函数值为hash\_key，将value对应的节点的头插法的方式插入到以t[hash\_key]为头指针的单链表中（采用头插法插入每次都是插入到头指针的next，这样不需要遍历链表，也不需要设置尾指针NULL，插入复杂度为O(1)）。

查找值：若元素value的哈希函数值为hash\_key，遍历以t[hash\_key]为头指针的单链表，查找链表各个节点的值域是否为value、

查找的时候，除了根据计算出来的散列值找到对应位置外，还需要在链表上进行搜索。而在单链表上的查找速度是很慢的。另外散列函数如果设计得好，冲突的概率其实也会很小。

**代码：**



注：insert参数说明：hash\_table[]：接收插入结点的链表数组（即上图左边的链表），node：待插入的节点，table\_len：链表数组的长度

**测试：**



### 开放定址法

分离链接散列算法的缺点是需要指针，由于给新单元分配地址需要时间，因此这就导致算法的速度多少有些减慢，同时算法实际上还需要对另一种数据结构的实现。除了使用链表解决冲突外，开放定址散列法是另外一种用链表解决冲突的方法。在开放定址散列算法系统中，如果有冲突发生，那么就要尝试选择另外的单元，直到找出空的单元为止。因为所有的数据都要置入表内，所以开放定址散列法所需要的表要比分离链接散列算法用表大。一般来说，对于开放定址散列算法来说，装填因子应该低于λ=0.5。

开放定址法的思想是，如果冲突发生，就选择另外一个可用/空的位置。只要散列表足够大，空的/可用的散列地址总能找到，并将记录存入。

开放定址法中也有常见的几种策略。

#### 线性探测法

公式：

fi(key) = (s(key)+di) MOD m (di=1,2,…,m-1)

还是以前面的为例：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  | 18 | 19 | 13 |

如果此时再要插入20，则20 % 7 = 6，但是6的位置已有元素，因此探测下一个位置（6+1）%7，在这里就是下标为0的位置。因此20的存储位置如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 20 | 50 |  |  | 18 | 19 | 13 |

但这种方式的一个问题是，可能造成一次聚集，因为一旦冲突发生，为了处理冲突就会占用下一个位置，而如果冲突较多时，就会出现数据都聚集在一块区域。这样就会导致任何关键字都需要多次尝试才可能解决冲突。

#### 平方探测法

顾名思义，如果说前面的探测函数是f(i)= i % 7，那么平方探测法就是f(i)= (i^2 )% 7。

可以修改di的取值方式，例如使用平方运算来尽量解决堆积问题：

fi(key) = (f(key)+di) MOD m (di=12,-12,22,-22,…q2,-q2,q<=m/2)

但是平方探测法同样会产生二次聚集问题。

#### 双散列

为了避免聚集，在探测时选择跳跃式的探测，即再使用一个散列函数，用来计算探测的位置。

在冲突时，对于位移量di采用随机函数计算得到，我们称之为随机探测法，公式：

fi(key) = (f(key)+di) MOD m (di是由一个随机函数获得的数列)

假设前面的散列函数为hash1(X)，用于探测的散列函数为hash2(X)，那么一种流行的选择是f(i) = i \* hash2(X)，即第一次冲突时探测hash1(X)+hash2(X)的位置，第二次探测hash1(X)+2hash2(X)的位置。

可以看到，无论是哪种开放定址法，它都要求表足够大。

### 再散列函数法

散列表可以认为是具有固定大小的数组，那么如果插入新的数据时散列表已满，或者散列表所剩容量不多该怎么办？这个时候就需要再散列，常见做法是，建立一个是原来两倍大小的散列表，将原来表中的关键字重新散列到新表中。

公式：

fi(key) = RHi(key) (1,2,3,…k)

注：RH为各种散列函数。

### 公共溢出区法

### 对比

实际应用中真正构造哈希表的还是拉链法，其余方法在冲突和平均复杂度方面不具备优势。

## 数据库哈希算法

在数据库领域中，哈希算法被广泛应用于多个核心组件，提升查询、插入、分区等操作的性能。以下是数据库中常用的哈希算法及其应用场景与原理介绍：

数据库中的哈希算法应用分类：

### 哈希索引（Hash Index）

原理：将键通过哈希函数映射到固定大小的哈希桶中。

用途：加速键值查询，常用于等值查询（`WHERE id = 123`）。

代表系统：

MySQL InnoDB：内部使用自适应哈希索引（Adaptive Hash Index）。

PostgreSQL：支持 hash index（但性能与 B-Tree 相比有限）。

常用哈希算法：

MurmurHash、CityHash、SipHash（防哈希冲突攻击）

### 分区哈希（Hash Partitioning）

原理：将表的数据根据键的哈希值分布到不同的物理/逻辑分区。

用途：大表的水平分片（sharding）、并行查询。

示例：

HASH(id) % 4 将表拆分为4个分区。

应用系统：

Oracle、MySQL、TiDB、Greenplum、ClickHouse 等。

常用哈希函数：

CRC32、MurmurHash、MD5（可控冲突概率）

### Join 算法中的哈希：Hash Join

原理：将一个表的 join key 哈希为哈希表，另一表依此探测匹配行。

类型：

Build-Probe Hash Join

Grace Hash Join（外部哈希 join，支持磁盘溢出）

使用场景：适用于中等大小、等值连接的表。

优化点：

多阶段哈希（多轮分桶）

Bloom Filter 提前过滤

### 哈希聚合（Hash Aggregation）

原理：对聚合键哈希分组并进行计数/汇总操作。

应用：执行 `GROUP BY` 等聚合查询时。

优势：

比排序聚合快（不要求键有序）。

数据库实现：

PostgreSQL、SparkSQL、DuckDB 等广泛使用。

### 布隆过滤器（Bloom Filter）中的哈希

原理：多个哈希函数用于生成位数组，用于快速判断“可能存在/一定不存在”。

应用：

Join 操作前的预过滤（如 Hive、Spark）。

LSM-tree 索引前的快速查询排除（如 RocksDB）。

常用哈希函数：

MurmurHash、xxHash

### 分布式系统中的哈希一致性（Consistent Hashing）

原理：将节点和数据映射到环上，保证最小迁移代价。

应用：

分布式缓存：Redis Cluster、Memcached

分布式数据库：Cassandra、TiKV

常用哈希函数：

MD5、SHA1、Jump Hash、Ketama

### 总结

手机屏幕截图

描述已自动生成

示例：TiDB 中的哈希应用

图形用户界面, 文本, 应用程序

描述已自动生成

## 分布式存储哈希算法

在分布式存储系统中，哈希算法是实现数据分布、负载均衡、快速索引和高可用性的关键工具。下面是哈希算法在分布式存储领域的主要应用及常用算法分类和原理总结：

分布式存储中的哈希算法应用场景：

### 数据分布（Sharding）

用途\*将数据平均分布到多个节点或分片中，避免热点、提高读写吞吐。

常用算法：

取模哈希（e.g., `hash(key) % N`）

一致性哈希（Consistent Hashing）

Rendezvous Hashing（Highest Random Weight）

### 一致性哈希（Consistent Hashing）

原理：将节点和数据都哈希映射到“哈希环”上，数据顺时针存储到下一个节点。

优点：

动态增删节点时仅影响少量数据（约 1/N）。

改进策略：

虚拟节点（Virtual Nodes）提高数据分布均衡性

Ketama、Jump Consistent Hash（更平衡）

应用系统：

Cassandra、TiKV、Ceph、etcd、Redis Cluster、Amazon DynamoDB

### 哈希分区（Hash Partitioning）

原理：通过哈希函数计算键值，并分配到不同分区。

使用场景：分布式文件系统（如 HDFS）、对象存储（如 MinIO）。

典型系统：

HDFS block placement

MongoDB / Couchbase 分区存储

### 哈希索引

原理：将键映射为指针或物理块地址，加速查找。

场景：

分布式对象存储（例如 S3 的底层 key-index）

RocksDB 的哈希索引 SST 结构

常见哈希算法：

MurmurHash、xxHash（速度快）、SipHash（抗冲突）

### 纠删码（Erasure Coding）与哈希

用途：数据冗余切分后，哈希决定每块数据分布位置。

典型系统：

Ceph、QFS、OpenStack Swift

优势：节省存储空间、提高容错性。

**扩展：**

纠删码（Erasure Coding，简称 EC）是一种冗余数据编码技术，广泛应用于 分布式存储系统，其主要作用是提升数据的可靠性和存储效率，在发生硬件故障（如磁盘/节点损坏）时，保证数据不丢失的同时减少冗余开销。

* 纠删码的作用

1、提升容错性

纠删码将原始数据划分为多个数据块，并通过算法生成冗余块。

即使部分块丢失，也可以利用剩余块重建原始数据。

例如：(k=4, m=2) 表示将数据分成 4 块，生成 2 个冗余块，任意 4 个块即可恢复原数据。

2、降低冗余开销（比RAID节省空间）

与传统的副本冗余（如三副本）相比，纠删码更节省空间。

示例：

三副本：冗余率 = 200%

(6,3) 纠删码：冗余率 = 50%

节省了 2/3 的冗余存储，适用于大规模数据中心和云存储。

3、数据恢复能力强

即使多个硬盘或节点同时宕机，只要剩下 k 块，数据就能被恢复。

支持节点失效、网络异常、数据损坏等多种情况的恢复。

4、适用于分布式和云环境

被广泛应用于对象存储、分布式文件系统、冷存储系统中：

Ceph

OpenStack Swift

Facebook F4（冷数据）

Hadoop HDFS（Erasure Coding 支持）

* 与三副本对比

表格

描述已自动生成

* 工作原理简述

1、编码阶段：

将数据分为 k 个数据块，通过编码函数生成 m 个冗余块。

总共存储 n = k + m` 块。

2、解码阶段（恢复数据）：

读取任意 `k` 个块（数据块或冗余块），通过解码算法重建原始数据。

3、常用算法：

Reed-Solomon（最经典）

LDPC（低密度奇偶校验）

Raptor codes（高性能流式）

Local Reconstruction Code（微软 Azure Storage）

举例：Ceph 中的纠删码

ceph osd erasure-code-profile set my-ec-profile \

k=4 m=2 plugin=jerasure technique=reed\_sol\_van

这表示：

采用 Reed-Solomon 算法

数据 4 块，冗余 2 块

最多能容忍 2 块丢失的情况

### 布隆过滤器（Bloom Filter）+ 哈希

用途：快速判断某块数据是否可能存在，减少 I/O。

哈希算法要求：多个独立哈希函数，低碰撞率。

使用场景：

LevelDB、RocksDB、Bigtable、Cassandra 索引加速

常用哈希算法：MurmurHash、FNV、xxHash

### 总结

常用哈希算法及特性表：

表格

描述已自动生成

应用系统举例：

图形用户界面, 应用程序

中度可信度描述已自动生成

延伸优化方向：

图形用户界面, 文本, 应用程序

中度可信度描述已自动生成

# 接口实现

注：分离链接法（separatechaining）是目前工程实践中最好的一种方式。

## 定义

#define HASHSIZE 12

#define NULLKEY -32768

typedef struct

{

int \*elem; //数据元素的基址,动态分配数组(数据域)

int count; //当前数据元素个数

}HashTable;

## 初始化

int InitHashTable(HashTable \*H)

{

H->count = HASHSIZE;

H->elem = (int \*)malloc(HASHSIZE \* sizeof()int);

//也可以直接使用固定大小的数组，但是哈希表大小可能需要动态调整，使用指针比较合适

if(!H->elem)

{

return -1;

}

for(i=0;i<HASHSIZE;i++)

{

H->elem[i] = NULLKEY;

}

return 0;

}

//使用除留余数法

int Hash(int key)

{

return key % HASHSIZE;

}

## 插入

//插入关键字到散列表

void InsertHash(HashTable \*H, int key)

{

int addr;

addr = Hash(key);

while(H->elem[addr]!=NULLKEY) //如果不为空,则冲突出现(冲突处理)

{

addr = (addr+1)%HASHSIZE; //开放定址法线性探测(可采用多种方法)

}

H->elem[addr] = key;

}

## 查找

//散列表查找关键字

int SearchHash(HashTable H, int key, int \*addr)

{

\*addr = Hash(key);

while(H.elem[\*addr]!=key)

{

\*addr = (\*addr+1)%HASHSIZE;

if(H.elem[\*addr] == NULLKEY || \*addr == Hash(key))

{

return -1;

}

}

}

# 工业级实现

## 设计

工业级哈希表不仅要支持高效查找/插入/删除，还要：

支持并发读写（多线程环境）

高效处理冲突（冲突解决策略）

支持动态扩容且尽量减少性能抖动

内存占用合理，缓存友好

支持泛型和自定义哈希函数

### 设计要点



### 并发设计技术



## 原理算法原理分类



### 链地址法

#### C++标准库std::unordered\_map

基于链地址法（链表存储冲突）

非线程安全

动态扩容机制合理

适合单线程或加外部锁使用

### 开放寻址法

#### Robin Hood Hashing

#### Cuckoo Hashing

#### Hopscotch Hashing

### 稀疏与密集哈希

#### Google SparseHash

#### DenseHash

SparseHash: 使用开放寻址 + 稀疏数组，内存利用高

DenseHash: 开放寻址，空间利用率极高，但需要额外的标记

高效，但不支持线程安全，需要外部同步

### Robin Hood Hashing变体

#### Google Abseil flat\_hash\_map/FlatMap

#### node\_hash\_map

#### SwissTable

使用开放寻址 + Robin Hood Hashing 算法

更低的缓存未命中，查找更快

单线程版本，不自带并发支持

## 线程支持与并发模型分类



### 单线程

#### std::unordered\_map

#### Google SparseHash/DenseHash

#### Abseil flat\_hash\_map/node\_hash\_map

#### SwissTable

#### F14

### 并发读写

#### Facebook Folly AtomicHashMap

#### ConcurrentHashMap

AtomicHashMap：无锁读写，使用原子操作，支持并发访问

ConcurrentHashMap：分段锁实现，写时加锁，读时无锁

高性能且工业级实用，支持多线程

#### Intel TBB concurrent\_hash\_map

基于分段锁设计，每个桶有自己的锁

支持多线程并发读写

API 接近 STL，适合并行程序使用

#### JCTools & Java ConcurrentHashMap

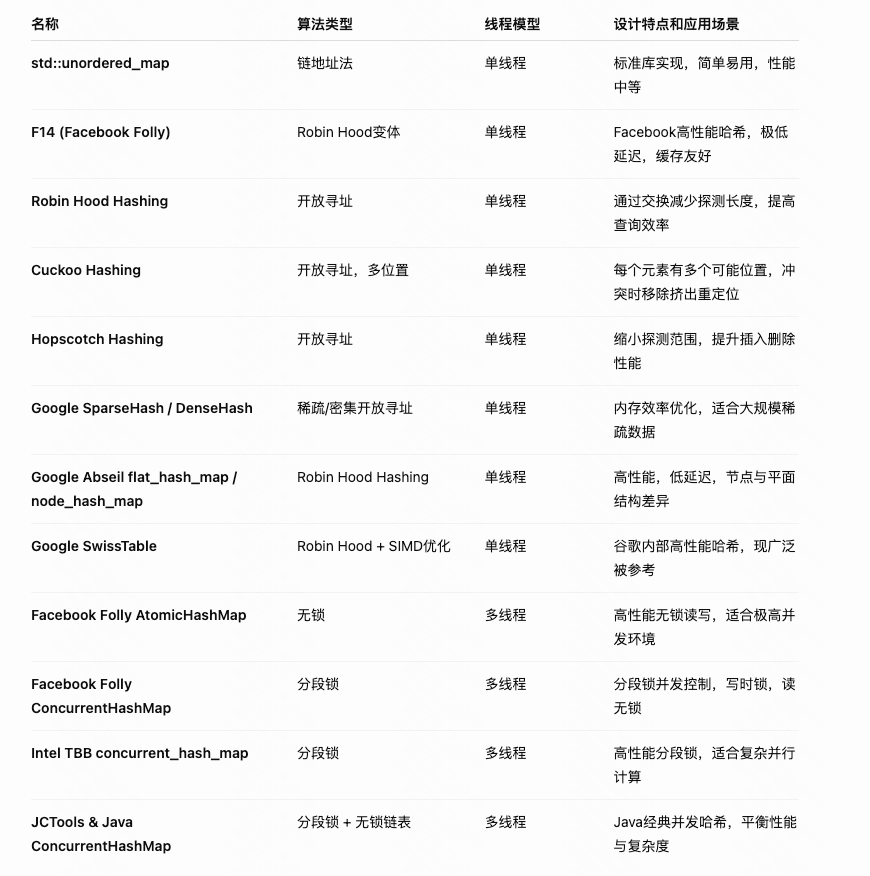
Java ConcurrentHashMap是经典分段锁+无锁链表/树结合实现

分段锁粒度降低锁竞争

支持高并发访问且扩容性能优秀

## 对比





# 应用

散列表应用很广泛。例如做**文件校验或数字签名**。当然还有快速查询功能的实现。例如，redis中的字典结构就使用了散列表，使用MurmurHash算法来计算字符串的hash值，并采用拉链法处理冲突，，当散列表的装载因子（关键字个数与散列表大小的比）接近某个大小时，进行再散列。

下面主要介绍哈希表在数据库中的应用：

## 哈希查找

哈希查找，由于哈希算法能够很好的把数据做比较均匀的分布，所以哈希查找的速度要比B+ Tree查找的速度要快。哈希查找的时间复杂度是O(1)，而B+ Tree的时间复杂度是O(h)。当然，B+ Tree的优势是在范围查找，不然也不会成为关系型数据库的基本算法了。

SQL Server在做联接(JOIN)的时候，如果两个表都不是很小，而且没有在关联列上排序，就很可能使用哈希联接。哈希联接使用的就是哈希查找，它的性能并不差，但是要注意的一点是，哈希联接需要先构造一个哈希表，而哈希表需要消耗不小的内存空间，如果数据库服务器的内存不足的话，SQL Server就只好使用“优雅的哈希联接”(Grace HASH JOIN)或者递归哈希联接(Recursive Hash Join)，这样性能就会受到影响了。在设计数据库的时候，我们应该注意建立/更新适当的索引和统计信息(STATISTICS)，以便SQL Server可以准确的估计联接的输入大小，以便选择正确的算法。

在使用哈希联接的时候，SQL Server的查询优化器会选择算法，通常我们并不需要做任何的指令，编写任何代码。但是在其它一些需要用到哈希算法的时候，就需要认真的选择算法，甚至可能需要自己写哈希算法的代码。

在业务系统中，可能生成的业务键（Business key）会比较长，例如某电商网站的订单号会类似这样：202210782169AC7G。这样的业务键如果用作主键的话，会占据16字节，显得有点浪费。如果只是单一系统使用，可以考虑用自动增量的数字作为主键。但是如果在多个系统中使用，比如在一个数据仓库系统中，使用自动增量作为代理键(Surrogate key)，就必须在处理事实表的ETL过程中，用新的自动增量替换掉订单表和详单表中的订单号，这会带来两个大表的JOIN，是一个相当耗时的操作。在这种情况下，我们就可以考虑使用哈希算法来生成代理键，只需要在订单表和详单表都使用同样的哈希算法，就可以保证得到的代理键是可以正确联接的。

## Hashmap

哈希表结构（链表散列：数组+链表）实现，结合数组和链表的优点。当链表长度超过8时，链表转换为红黑树。

HashMap是基于哈希表实现的，每一个元素是一个key-value对，其内部通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

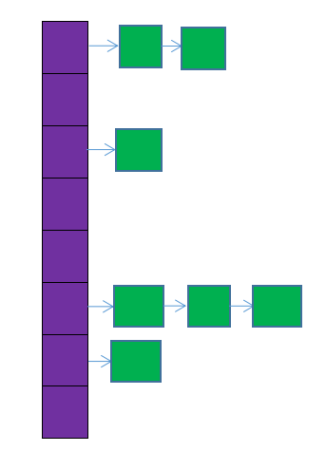
HashMap是非线程安全的，只是用于单线程环境下，多线程环境下可以采用concurrent并发包下的concurrentHashMap。

HashMap 实现了Serializable接口，因此它支持序列化，实现了Cloneable接口，能被克隆。

Hashtable也是JDK1.0引入的类，是线程安全的，能用于多线程环境中。

Hashtable同样实现了Serializable接口，它支持序列化，实现了Cloneable接口，能被克隆。

HashMap的存储结构，如下图所示：



图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

从HashMap的底层结构中我们可以看到，HashMap采用是数组+链表/红黑树的组合来作为底层结构，也就是开放地址法+链地址法的方式来实现HashMap。

**总结：**

针对Hashtable，我们同样给出几点比较重要的总结，但要结合与HashMap的比较来总结。

1、二者的存储结构和解决冲突的方法都是相同的。

2、HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

3、Hashtable中key和value都不允许为null，而HashMap中key和value都允许为null（key只能有一个为null，而value则可以有多个为null）。但是如果在Hashtable中有类似put(null,null)的操作，编译同样可以通过，因为key和value都是Object类型，但运行时会抛出NullPointerException异常，这是JDK的规范规定的。

很明显，如果value为null，会直接抛出NullPointerException异常，但源码中并没有对key是否为null判断，有点小不解！不过NullPointerException属于RuntimeException异常，是可以由JVM自动抛出的，也许对key的值在JVM中有所限制吧。

4、Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

5、Hashtable计算hash值，直接用key的hashCode()，而HashMap重新计算了key的hash值，Hashtable在求hash值对应的位置索引时，用取模运算，而HashMap在求位置索引时，则用与运算，且这里一般先用hash&0x7FFFFFFF后，再对length取模，&0x7FFFFFFF的目的是为了将负的hash值转化为正值，因为hash值有可能为负数，而&0x7FFFFFFF后，只有符号外改变，而后面的位都不变。

**为什么HashMap加载因子一定是0.75？而不是0.8，0.6？**

从上文我们知道，HashMap的底层其实也是哈希表（散列表），而解决冲突的方式是链地址法。HashMap的初始容量大小默认是16，为了减少冲突发生的概率，当HashMap的数组长度到达一个临界值的时候，就会触发扩容，把所有元素rehash之后再放在扩容后的容器中，这是一个相当耗时的操作。

而这个临界值就是由加载因子和当前容器的容量大小来确定的：

临界值 = DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR

即默认情况下是16x0.75=12时，就会触发扩容操作。

那么为什么选择了0.75作为HashMap的加载因子呢？笔者不才，通过看源码解释和大佬的文章，才知道这个跟一个统计学里很重要的原理——泊松分布有关。

泊松分布是统计学和概率学常见的离散概率分布，适用于描述单位时间内随机事件发生的次数的概率分布。

等号的左边，P表示概率，N表示某种函数关系，t表示时间，n表示数量。等号的右边，λ表示事件的频率。

在理想情况下，使用随机哈希码，在扩容阈值（加载因子）为0.75的情况下，节点出现在频率在Hash桶（表）中遵循参数平均为0.5的泊松分布。忽略方差，即X =λt，P(λt = k)，其中λt = 0.5的情况，按公式：

计算结果如上述的列表所示，当一个bin中的链表长度达到8个元素的时候，概率为0.00000006，几乎是一个不可能事件。

所以我们可以知道，其实常数0.5是作为参数代入泊松分布来计算的，而加载因子0.75是作为一个条件，当HashMap长度为length/size ≥ 0.75时就扩容，在这个条件下，冲突后的拉链长度和概率结果为：

0: 0.60653066

1: 0.30326533

2: 0.07581633

3: 0.01263606

4: 0.00157952

5: 0.00015795

6: 0.00001316

7: 0.00000094

8: 0.00000006

**那么为什么不可以是0.8或者0.6呢？**

HashMap中除了哈希算法之外，有两个参数影响了性能：初始容量和加载因子。初始容量是哈希表在创建时的容量，加载因子是哈希表在其容量自动扩容之前可以达到多满的一种度量。

在维基百科来描述加载因子：

对于开放定址法，加载因子是特别重要因素，应严格限制在0.7-0.8以下。超过0.8，查表时的CPU缓存不命中（cache missing）按照指数曲线上升。因此，一些采用开放定址法的hash库，如Java的系统库限制了加载因子为0.75，超过此值将resize散列表。

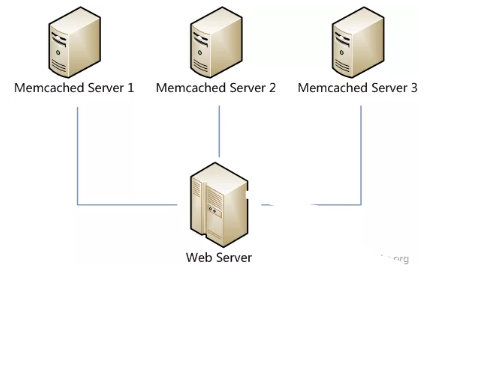
在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少扩容rehash操作次数，所以，一般在使用HashMap时建议根据预估值设置初始容量，以便减少扩容操作。

选择0.75作为默认的加载因子，完全是时间和空间成本上寻求的一种折衷选择。

## 一致性哈希

### 分布式缓存

假设我们有一个网站，最近发现随着流量增加，服务器压力越来越大，之前直接读写数据库的方式不太给力了，于是我们想引入Memcached作为缓存机制。现在我们一共有三台机器可以作为Memcached服务器，如下图所示：

很显然，最简单的策略是将每一次Memcached请求随机发送到一台Memcached服务器，但是这种策略可能会带来两个如下问题：

（1）一是同一份数据可能被存在不同的机器上而造成数据冗余；

（2）二是有可能某数据已经被缓存但是访问却没有命中，因为无法保证对相同key的所有访问都被发送到相同的服务器。因此，随机策略无论是时间效率还是空间效率都非常不好。

要解决上述问题只需做到如下一点：保证对相同key的访问会被发送到相同的服务器。很多方法可以实现这一点，最常用的方法是计算哈希。

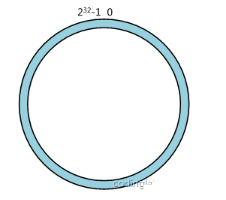
假设对于上图，对于每次访问，可以按如下算法计算其哈希值：h = Hash(key) % 3 其中Hash是一个从字符串到正整数的哈希映射函数。这样，如果我们将Memcached Server分别编号为0、1、2，那么就可以根据上式和key计算出服务器编号h，然后去访问。这个方法虽然解决了上面提到的两个问题，但是存在一些其它的问题。如果将上述方法抽象，可以认为通过：h = Hash(key) % N。这个算式计算每个key的请求应该被发送到哪台服务器，其中N为服务器的台数，并且服务器按照0–(N-1)编号。这个算法的问题在于容错性和扩展性不好。所谓容错性是指当系统中某一个或几个服务器变得不可用时，整个系统是否可以正确高效运行；而扩展性是指当加入新的服务器后，整个系统是否可以正确高效运行。

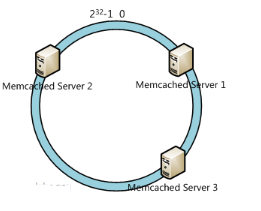
现假设有一台服务器宕机了，那么为了填补空缺，要将宕机的服务器从编号列表中移除，后面的服务器按顺序前移一位并将其编号值减一，此时每个key就要按h = Hash(key) % (N-1)重新计算。

同样，如果新增了一台服务器，虽然原有服务器编号不用改变，但是要按h = Hash(key) % (N+1)重新计算哈希值。因此系统中一旦有服务器变更，大量的key会被重定位到不同的服务器从而造成大量的缓存不命中。而这种情况在分布式系统中是非常糟糕的。

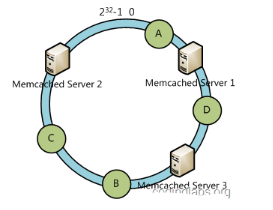
一个设计良好的分布式哈希方案应该具有良好的单调性，即服务节点的增减不会造成大量哈希重定位。一致性哈希算法就是这样一种哈希方案。

### 一致性哈希算法

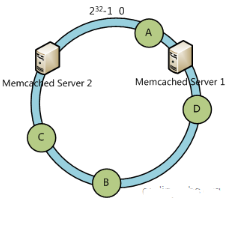
一致性哈希算法（Consistent Hashing）最早在论文《Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web》中被提出。简单来说，一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0-232-1（即哈希值是一个32位无符号整形），整个哈希空间环如下：

整个空间按顺时针方向组织。0和232-1在零点中方向重合。下一步将各个服务器使用H进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中三台服务器使用ip地址哈希后在环空间的位置如下：

接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数H计算出哈希值h，通根据h确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。 例如我们有A、B、C、D四个数据对象，经过哈希计算后，在环空间上的位置如下：

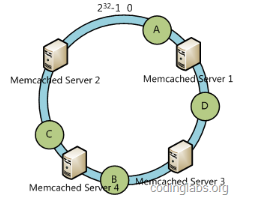
根据一致性哈希算法，数据A会被定为到Server 1上，D被定为到Server 3上，而B、C分别被定为到Server 2上。

### 容错性和扩展性分析

下面分析一致性哈希算法的容错性和可扩展性。现假设Server 3宕机了：

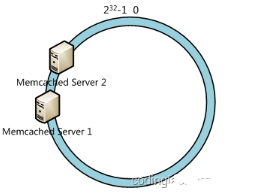
可以看到此时A、C、B不会受到影响，只有D节点被重定位到Server 2。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

下面考虑另外一种情况，如果我们在系统中增加一台服务器Memcached Server 4：

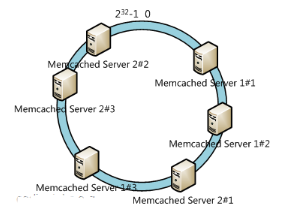
此时A、D、C不受影响，只有B需要重定位到新的Server 4。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

### 虚拟节点

一致性哈希算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题。 例如我们的系统中有两台服务器，其环分布如下：

此时必然造成大量数据集中到Server 1上，而只有极少量会定位到Server 2上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。

具体做法可以在服务器ip或主机名的后面增加编号来实现。例如上面的情况，我们决定为每台服务器计算三个虚拟节点，于是可以分别计算“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”、“Memcached Server 2#1”、“Memcached Server 2#2”、“Memcached Server 2#3”的哈希值，于是形成六个虚拟节点：

同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”三个虚拟节点的数据均定位到Server 1上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

### 总结

目前一致性哈希基本成为了分布式系统组件的标准配置，例如Memcached的各种客户端都提供内置的一致性哈希支持。更深入的内容可以参看论文《Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web》。

## 适用场景

不适合使用哈希表解决的问题：

1. 要求数据之间顺序不能变动的问题
2. 涉及多维数据的问题
3. 需要执行前缀搜索才能解决的问题，特别是键本身很长，而且各键长度又不一致的问题
4. 包含动态数据的问题
5. 无法通过独特的键来确定数据身份的问题

# 总结

一个设计良好的散列表能够几乎在O(1)时间复杂度内完成插入，删除和查找，但前提是散列函数设计得足够优雅，以及有着合适散列冲突解决方案。常见冲突解决方案有：

拉链法

开放地址检测法

其中拉链法在实际中是很常见的一种解决方案。

**参考：**

《数据结构与算法分析》

《redis设计与实现》

<https://en.wikipedia.org/wiki/Hash_table>

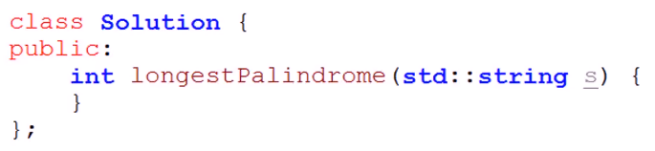
# 相关案例

哈希表适合查找某字符个数，比如重复字符，只出现N次的元素，数组交集，K值。

## 最长回文串

**题目：**已知一个只包括大小写字符的字符串，求用该字符串中的字符可以生成的最长回文字符串长度。

例如，s=“abccccddaa”，可生成的最长回文字符串长度为9，如dccaaaccd、adccdccda、acdcacdca等，都是正确的。



注：Leetcode 409

**分析：**

例如在s=“abccccddaa”中，有3个a，1个b，4个c，2个d。使用字符串s中的字符，任意组合，生成新的字符串，若生成的字符串为回文字符串，需要除了中心字符（奇数个），其余字符只要头部出现，尾部就要对应出现。

例如：a…a、ccd…dcc、cc…d…cc

在字符串中，字符数量为偶数和奇数的字符，应该分别如何处理？

1. 在字符串中，字符数量为偶数的字符：

全部使用，头部放一个字符，尾部就对应放一个。

如，4个c和2个d可全部用上：…ccd…dcc…、…cdc…cdc…、…dcc…ccd…等。

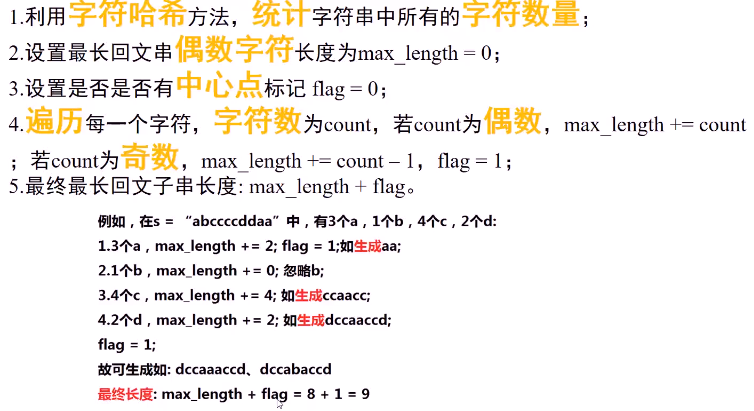
1. 子字符串中，字符数量为奇数的字符：

丢掉一个字符，剩下的字符数为偶数个，按照字符数量为偶数的字符处理。

如，3个a中，有2个a可以用上：…a…a…

1. 若有剩余的字符：

如：1个a、1个b，随便选择1个字符当做中心字符：…a…、…b…



算法思路：

1. 利用字符哈希方法，统计字符串中所有的字符数量；
2. 设置最长回文串偶数字符长度为max\_length=0；
3. 设置是否有中心点标记flag=0；
4. 遍历每一个字符，字符数为count：

若count为偶数，max\_length+=count；

若count为奇数，max\_length+=count-1，flag=1

1. 最终最长回文子串长度：max\_length+flag

例如：在s=“abccccddaa”中，有3个a，1个b，4个c，2个d

1、3个a，max\_length+=2，flag=1，如生成aa；

2、1个b，max\_length+=0，忽略b；

3、4个c，max\_length+=4，如生成ccaacc；

4、2个d，max\_length+=2，如生成dccaaccd；

flag=1；

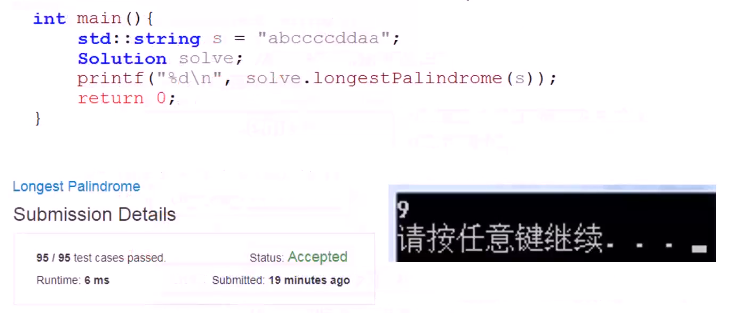
故可生成如：dccaaaccd，dccadaccd

最终长度：max\_length+flag=8+1=9

**代码：**



**测试：**



## 词语模式

**题目：**已知字符串pattern与字符串str，确认str是否与pattern匹配。Str与pattern匹配代表字符串str中的单词与pattern中的字符一一对应。（其中pattern中只包括小写字符，str中的单词只包含小写字符，使用空格分隔）

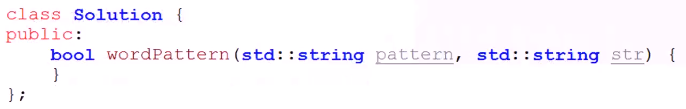
例如：

pattern=“abba”，str=“dog cat cat dog”匹配

pattern=“abba”，str=“dog cat cat fish”不匹配

pattern=“aaaa”，str=“dog cat cat dog”不匹配

pattern=“abba”，str=“dog dog dog dog”不匹配



注：Leetcode 290

**分析：**

匹配：字符串str中的单词与pattern中的字符一一对应。

如：

1. 假设pattern=“abba”，str=“dog cat cat”；则\*一定代表dog；
2. 假设pattern=“abb?”，str=“dog cat cat dog”，则？一定代表a；
3. 假设pattern=“abb?”，str=“dog cat cat dog”，则？一定不是a或b；
4. 假设pattern=“abba”，str=“\*”，则\*代表了4个单词；
5. 假设pattern=“\*”，str=“dog cat cat dog”，则\*代表了4个单词。

分析结论：

1. 当拆解出一个单词时，若该单词已出现，则当前单词对应的pattern字符必为该单词曾经对应的pattern字符；
2. 当拆解出一个单词时，若该单词未曾出现，则当前单词对应的pattern字符也必须未曾出现；
3. 单词的个数与pattern字符串中的字符数量相同。

算法思路：

pattern=“abb?”，str=“dog cat cat \*”；

dog->a；cat->b

1. 设置单词（字符串）到pattern字符的映射（哈希），使用数组used[128]记录pattern字符是否使用；
2. 遍历str，按照空格拆分单词，同时对应的向前移动指向pattern字符的指针，每拆分出一个单词，判断：

2.1 如果该单词从未出现在哈希表中：

如果当前的pattern字符已被使用，则返回false；

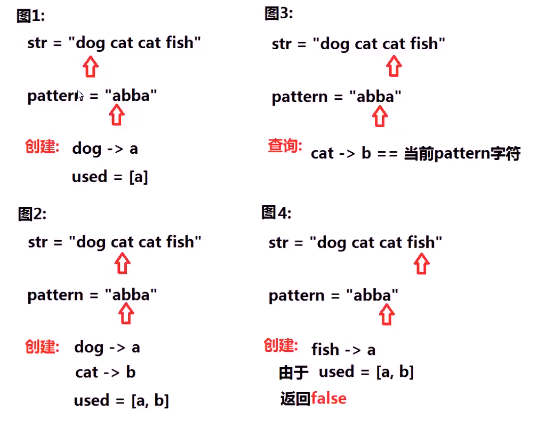
将单词与当前指向的pattern字符做映射；

标记当前指向的pattern字符已使用。

2.2 否则：

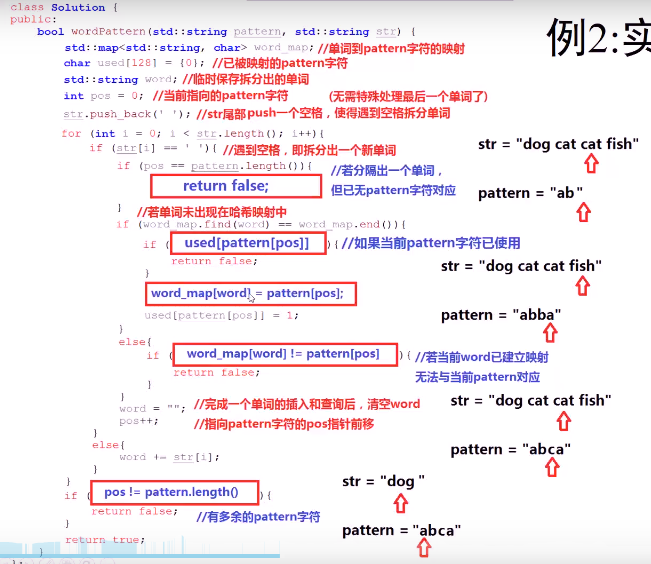
如果当前单词在哈希表中的映射字符不是当前指向的pattern字符，则返回false。

1. 若单词个数与pattern字符个数不匹配，则返回false。

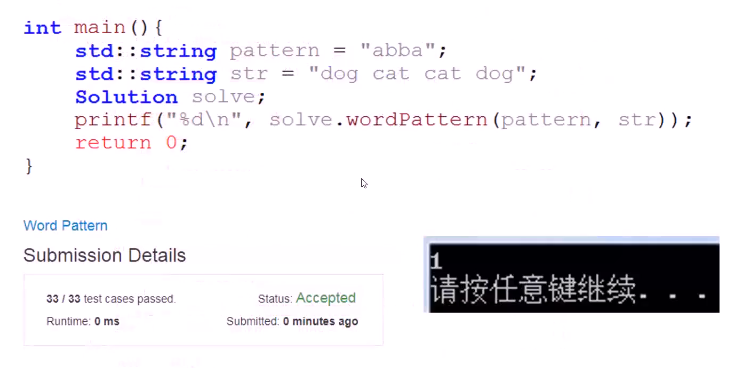




**代码：**



**测试：**

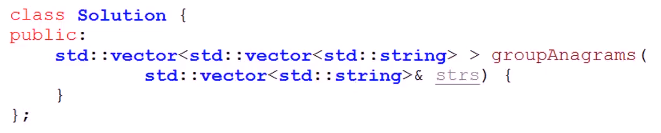


## 同字符词语分组

**题目：**已知一组字符串，将所有anagram（由颠倒字母顺序而构成的字）放到一起输出。

例如：[“eat”，“tea”，“tan”，“ate”，“nat”，“bat”]

返回：[[“ate”，“eat”，“tea”]，[“nat”，“tan”]，[“bat”]]



注：Leetcode 49

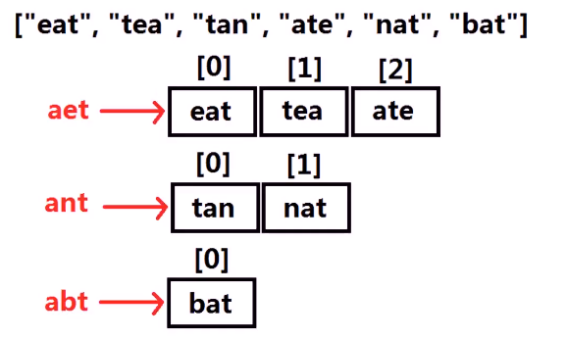
**分析：**

anagram分组：若某两个字符串，出现的各个字符数相同，则它们应该为同一分组。

如何建立哈希表，怎么设计哈希表的key和value，就可将各个字符数相同的字符串映射到一起？

方法一：

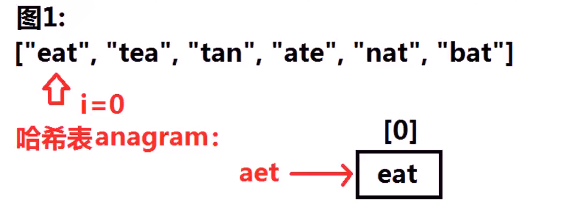
哈希表以内部进行排序的各个单词为key，以字符串向量（vector<string>）为value，存储各个字符数量相同的字符串（anagram）。

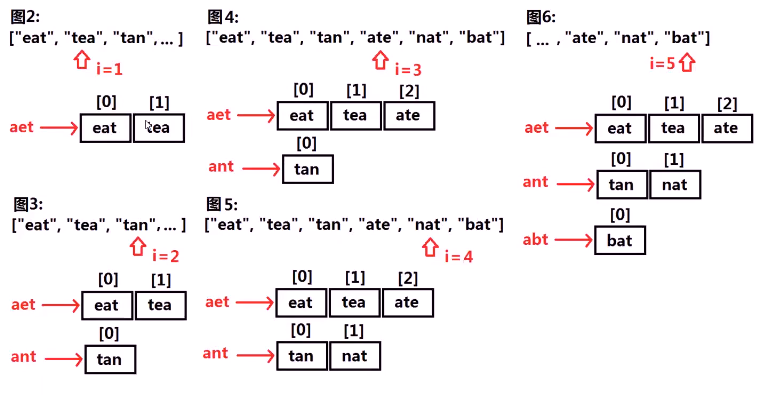


设置字符串到字符串向量的哈希表anagram，遍历字符串向量strs中的单词strs[i]：

1. 设置临时变量str=strs[i]，对str进行排序；
2. 若str未出现在anagram中，设置str到一个空字符串向量的映射；
3. 将strs[i]添加至字符串向量anagram[str]中；
4. 遍历哈希表anagram，将全部key对应的value push到最终结果中。

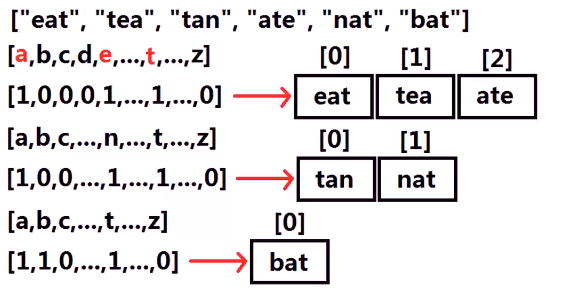
具体过程：





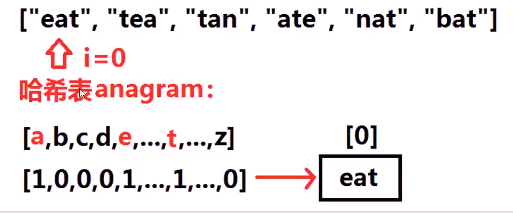
方法二：

哈希表以26个字母的字符数量（一个长度为26的vector，统计单词中各个字符的数量）为key，以字符串向量（vector<string>）为value，存储各个字符数量相同的字符串（anagram）。



设置vector到字符串向量的哈希表anagram，遍历字符串向量strs中的单词strs[i]：

1. 统计strs[i]中的各个字符数量，存储至vec；
2. 若vec未出现在anagram中，设置vec到一个空字符串向量的映射；
3. 将strs[i]添加至字符串向量anagram[vec]中；
4. 遍历哈希表anagram，将全部key对应的value push到最终结果中。

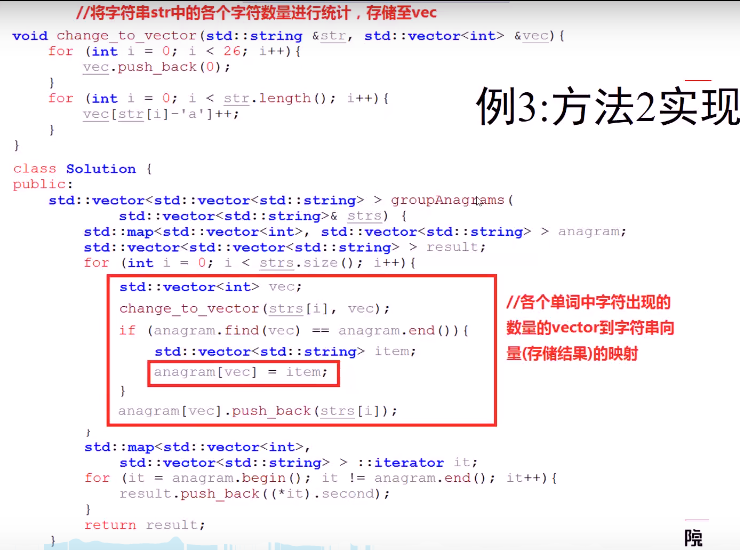


**代码：**

**方法一：**

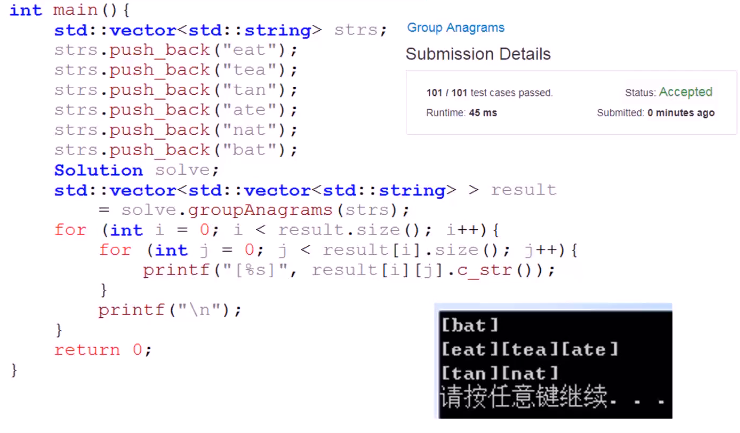


方法二：



注：两种方法性能差不多。

**测试：**

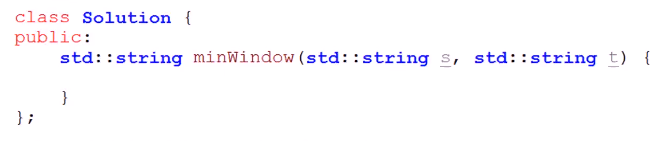


## 最小包含字符窗口

**题目：**给定一个字符串S（源字符串）和一个字符串T（目标字符串），使用O(n)的时间复杂度在S中找到一个最小的窗口大小，使得该窗口包含T中的所有字符。

例如：s=“ADOBECODEBANC”；T=“ABC”

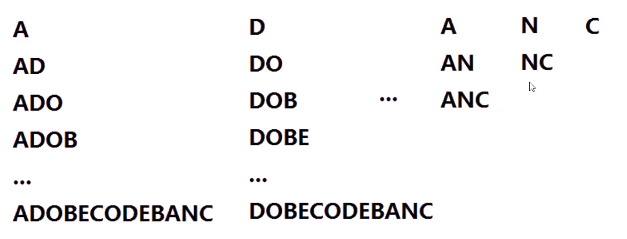
包含T的子区间中，有“ADOBEC”，“CODEBA”，“BANC”等等，最小窗口包含区间是“BANE”。



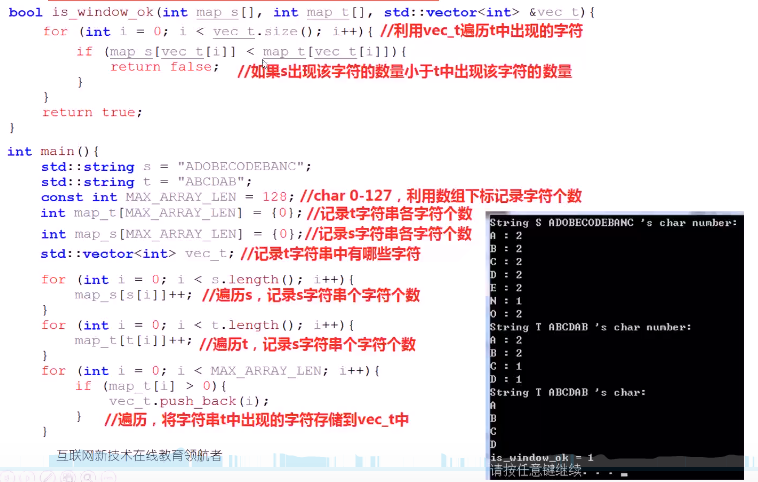
注：Leetcode 76

**分析：**

枚举s=“ADOBECODEBANC”的所有子字符串：



检查以上所有子字符串，是否包含字符串T=“ABC”。



算法思路：

1. 设置两个字符哈希数组，map\_s和map\_t，map\_s代表当前处理的窗口区间中的字符数量，map\_t代表子串T的字符数量；
2. 设置两个指针（记作指针i与指针begin）指向字符串第一个字符；
3. i指针向后逐个扫描字符串中的字符，在这个过程中，循环检查begin指针是否可以向前移动：

3.1 如果当前begin指向的字符T中没出现，直接前移begin；

3.2 如果begin指向的字符T中出现了，但是当前区间窗口中的该字符数量足够，向前移动begin，并更新map\_s；

3.3 否则，不能移动begin，跳出检查。

1. 指针i每向前扫描一个字符，即检查一下是否可以更新最终结果（找到最小的包含T中各个字符的窗口）。

在整个过程中，使用begin与i维护一个窗口，该窗口中的子串满足题目条件（包含T中所有字符），窗口线性向前滑动，整体时间复杂度为O(n)。



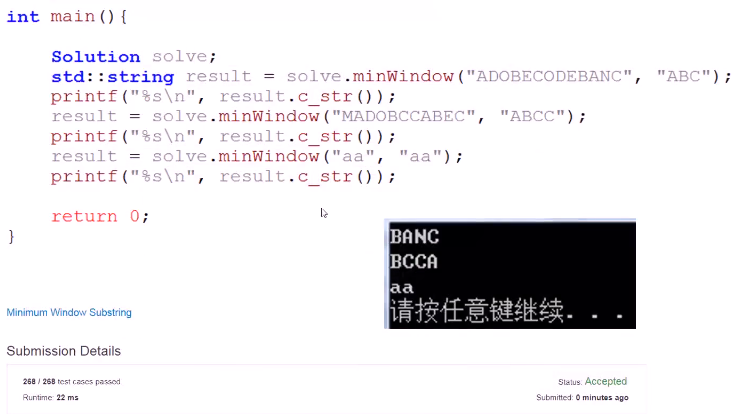




**代码：**



**测试：**



**代码：**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

题目：一个字符串能包含另一个字符串中所有字幕的子串的最小长度

思路：使用两个指针，一个hash表，两个指针为了记录宽带，hash表为了记录是否存在目标子串在两个指针之间出现了多少次，直到所有字符都出现在这段子串内，移动前面的指针，直到某一个子串出现在子串中一次那么这个空间的长度就是当前最短的，这样遍历直到把所有字符串遍历结束

\*/

string MinLength(string& src,string& dest)

{

int i=0,j=0;

int flag =0;

int len = src.size();

int pos =0;

vector<int> hash(26,-1);

for(i=0;i<dest.length();i++)

hash[dest[i]-'A'] =0;

//开始处理

for(i=0;i<src.size();i++)

{

if(hash[src[i]-'A'] >= 0)

{

hash[src[i]-'A']++;

if(hash[src[i]-'A'] ==1)

flag++;

}

if(flag == dest.length())

{

for(;j<i;j++)

{

if(hash[src[j]-'A'] ==1)

break;

else

hash[src[j]-'A']--;

}

if(len>i-j+1)

{

len =i-j+1;

pos = j;

}

hash[src[j]-'A'] =0;

j++;

flag--;

}

}

cout<<string(src,pos,pos+len)<<" pos is "<<pos<<" len is "<<len<<endl;

return string(src,pos,pos+len);

}

int main()

{

string src("ADOBECODEBANCAAABC");

string dest("ABC");

cout<<MinLength(src,dest)<<endl;

return 0;

}

## 重复的DNA序列

## 最长无重复字符子串

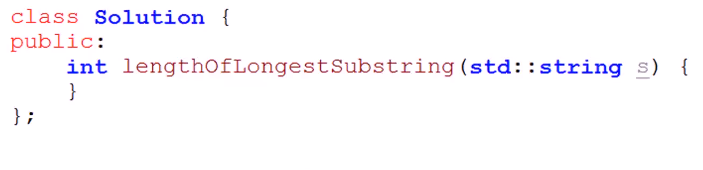
**题目：**给定一个字符串S，在该字符串中找到一个最长的没有重复字符的子串。

例如：

s=“abcabcbb”->“abc”，3

s=“bbbbb”->“b”，1

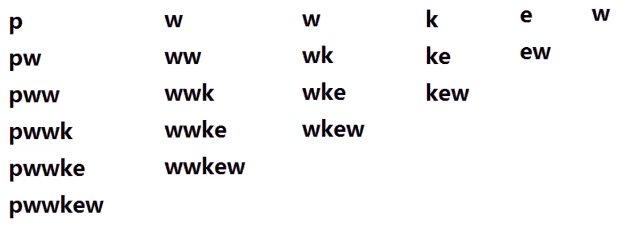
s=“pwwkew”->“wke”，3（注意pwke是子序列而非子串）



注：Leetcode 3

**分析：**

枚举s=“pwwkew”的所有子字符串：

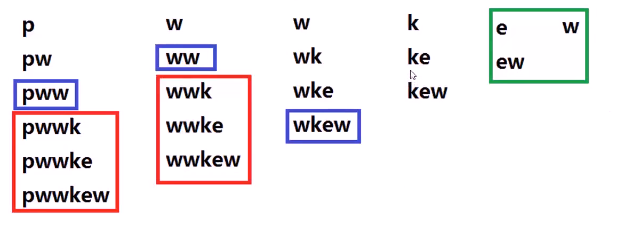


检查以上所有子字符串，是否满足无重复字符的条件，取最长的满足条件的子串作为结果。

该方法的复杂度为：O(n2)

该题目的时间复杂度优化目标是从O(n2)优化至O(n)（中间没有优化至nlogn的条件），所以需要将两重循环的枚举扫描修改为一层扫描。

题目中最关键的条件，无重复字符的子串，观察子串：



红色方框中的枚举均无意义，因为蓝色方框中已出现了重复字符。

绿色方框中的也无意义，因为后续的枚举不会出现满足条件的更优子串。

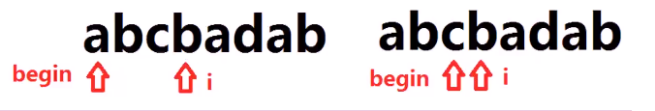
算法思路：

1. 设置一个记录字符数量的字符哈希char\_map；
2. 设置一个记录当前满足条件的最长子串变量word；
3. 设置两个指针（记作指针i与指针begin）指向字符串第一个字符；
4. 设置最长满足条件的子串的长度result；
5. i指针向后逐个扫描字符串中的字符，在这个过程中，使用char\_map记录字符数量；

5.1 如果word中没出现过该字符：对word尾部天剑字符并检查result是否需要更新；

5.2 否则：begin指针向前移动，更新char\_map中的字符数量，直到字符s[i]的数量为1；更新word，将word赋值为begin与i之间的子串。

在整个过程中，使用begin与i维护一个窗口，该窗口中的子串满足题目条件（无重复的字符），窗口线性向前华东，整体时间复杂度为O(n)。



**代码：**



**测试：**



**代码：**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

题目：最长没有重复字符的子串

思路：假设所有的字符都是26个小写英文字符。以此建立hash表，hash表内记录着字符出现的位置，如果发现当前字符在以前遍历过的字符串中已经存在那么从上次出现这个字符的位置的下一个字符重新遍历那么把hash表清空，继续从当前位置开始遍历

\*/

int Longest\_substring(string& str)

{

vector<int> hash(26,-1);

int i;

int cur =0;

int maxlen =0;

int tmp;

for(i=0;i<str.size();i++)

{

if(hash[str[i]-'a'] ==-1)

{

hash[str[i]-'a'] =i;

++cur;

maxlen = maxlen > cur?maxlen:cur;

}

else

{

tmp = hash[str[i]-'a'];//此时记录的位置

i = tmp;

cur =0;

memset(&hash[0],-1,sizeof(int)\*hash.size());

}

}

return maxlen;

}

/\*

同样的思想 简洁的方法

\*/

int lengthOfLongeststring(string s)

{

vector<int> charIndex(256,-1);//这里扩展到了所有的字符

int longest =0;

int m=0;

for(int i=0;i<s.size();i++)

{

m = max(charIndex[s[i]]+1,m);

charIndex[s[i]] =i;

longest = max(longest,i-m+1);

}

return longest;

}

int main()

{

string str("abcabferafjlkcbb");

cout<<lengthOfLongeststring(str)<<endl;

return 0;

}

## 直线上最多点的个数

**题目：**有一个二维坐标点的集合，查找共线最多的点的个数。

**代码：**

#include <iostream>

#include <vector>

using namespace std;

/\*

共线最多的点的个数

思路：可以看所有两点能够构造的直线的斜率，根据斜率来判断共线的点的个数 这样斜率和该斜率上点的个数就构成了一个键值对，可以使用hash表来存储

\*/

int maxPoints(vector<pair<int,int> >& points )

{

if(points.size() == 0)

return 0;

int max =1;

double ratio=0.0;

//开始遍历点，构造键值对

for(int i=0;i<points.size()-1;i++)

{

hash\_map<double,int> map;

int numofSame=0;

int localMax=1;

//从当前点和其之后的点构成的斜率

for(int j=i+1;;j<points.size();j++)

{

if(points[j].first == points[j].first && points[j].second == points.second)

{

numofSame++;

continue;

}

else if(points[j].first == points[i].first)

{

ratio = numeric\_limits<double>::max();

}

else if(points[j].second == points[i].second)

{

ratio =0.0;

}

else

{

ratio = (double)(points[j].second-

points[i].second)/(double)(points[j].first-points[i].first);

}

int num;

if(map.find(ratio) != map.end())

map[ratio]++;

else

map[ratio]=2;

}

//开始查找某一斜率下最多的点的个数

hash\_map<double,int>::iterator it = map.begin();

fo(;it!=map.end();it++)

localMax = max(localMax,it->second);

localMax += numofSame;

max = max(max,localMax);

}

return max;

}