解密hash算法的工程应用

今日目标：

1：能够说出哈希算法的定义及要求

2：能够说出哈希算法的诸多应用场景

3：完成相关面试实战题目

1、哈希算法

> 注意：讨论哈希算法并不是去研究哈希算法的原理及如何设计一个哈希算法，更多的是说明哈希算法的特点以及应用场景

1.1、定义及要求

哈希算法又称为摘要算法，它可以将任意数据通过一个函数转换成长度固定的数据串，这个映射转换的规则就是哈希算法，而通过原始数据映射之后得到的二进制值串就是哈希值。 可见，摘要算法就是通过**摘要函数f()**对**任意长度的数据data**计算出**固定长度的摘要digest**，目的是为了发现原始数据**是否被人篡改过。**

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个**单向函数**，计算`f(data)`很容易，但通过`digest`反推`data`却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

那有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要呢？完全有可能！因为任何摘要算法都是把无限多的数据集合映射到一个**有限的集合**中。这种情况就是我们说的**碰撞。**

我们要想设计出一个优秀的哈希算法并不是很容易，一个优秀的哈希算法一般要满足如下几点要求：

1. **将任何一条不论长短的信息，计算出唯一的一摘要(哈希值)与它相对应，对输入数据非常敏感，哪怕原始数据只修改了一个 Bit，最后得到的哈希值也大不相同**
2. **摘要的长度必须固定，散列冲突的概率要很小，对于不同的原始数据，哈希值相同的概率非常小**
3. **摘要不可能再被反向破译。也就是说，我们只能把原始的信息转化为摘要，而不可能将摘要反推回去得到原始信息，即哈希算法是单向的**
4. **哈希算法的执行效率要尽量高效，针对较长的文本，也能快速地计算出哈希值**

这些要求都是比较理论的说法，我们那一种企业常用的哈希算法MD5来说明：现使用MD5对三段数据分别进行哈希求值：

> 1.MD5('数据结构和算法') = 31ea1cbbe72095c3ed783574d73d921e
>
> 2.MD5('数据结构和算法很好学')=0fba5153bc8b7bd51b1de100d5b66b0a
>
> 3.MD5('数据结构和算法不好学')=85161186abb0bb20f1ca90edf3843c72

从其结果我们可以看出：MD5的摘要值(哈希值)是固定长度的，是16进制的32位即128 Bit位，无论要哈希的数据有多长，多短，哈希之后的数据长度是固定的，另外哈希值是随机的无规律的，无法根据哈希值反向推算文本信息。

其次2，3表明尽管只有一字之差得到的结果也是千差万别，最后哈希的速度和效率是非常高的，这一点我们可能还体会不到，因为我们哈希的只是很短的一串数据，即便我们哈希的是整个这段文本，用MD5计算哈希值，速度也是非常的快，总之MD5基本满足了我们前面所讲解的这几个要求。

1.2、应用场景

数据加密

用于加密数据的最常用的哈希算法是MD5（MD5 Message-Digest Algorithm，MD5 消息摘要算法）和SHA（Secure HashAlgorithm，安全散列算法）

对用于加密的哈希算法来说，有两点格外重要：

第一点：很难根据哈希值反向推导出原始数据，

第二点：散列冲突的概率要很小。

第一点很好理解，加密的目的就是防止原始数据泄露，所以很难通过哈希值反向推导原始数据，这是一个最基本的要求。对于第二点。实际上，不管是什么哈希算法，我们只能尽量减少碰撞冲突的概率，理论上是没办法做到完全不冲突的。为什么？

基于组合数学中一个非常基础的理论，抽屉原理：桌上有十个苹果，要把这十个苹果放到九个抽屉里，无论怎样放，我们会发现至少会有一个抽屉里面至少放两个苹果。这一现象就是我们所说的“抽屉原理”。

掌握这一原理后我们就知道了哈希算法无法做到零冲突，我们只能说设计优良的哈希算法能够最大程度的避免冲突。

除此之外，没有 绝对安全的加密。越复杂、越难破解的加密算法，需要的计算时间也越长。比如SHA-256 比 SHA-1 要更复杂、更安全，相应的计算时间就会比较长。密码学界也一直致力于找到一种快速并且很难被破解的哈希算法。我们在实际的开发过程中，也需要权衡破解难度和计算时间，来决定究竟使用哪种加密算法。

唯一ID

比如：如果要在海量的图库中，搜索一张图片是否存在，我们不能单纯地用图片的图片名称来比对，因为有可能存在名称相同但图片内容不同，或者名称不同图片内容相同的情况。那该如何搜索呢？

我们可以给每一个图片取一个唯一标识，或者说信息摘要。比如，我们可以取图片的名称，存储路径，图片的元信息(所有者等信息)或者说使用图片的二进制码信息，然后通过哈希算法（比如 MD5），得到一个哈希字符串，用它作为图片的唯一标识。通过这个唯一标识来判定图片是否在图库中，这样就可以减少很多工作量。

如果还想继续提高效率，我们可以把每个图片的唯一标识，和相应的图片文件在图库中的路径信息，都存储在散列表中。当要查看某个图片是不是在图库中的时候，我们先通过哈希算法对这个图片取唯一标识，然后在散列表中查找是否存在这个唯一标识。如果不存在，那就说明这个图片不在图库中；如果存在，我们再通过散列表中存储的文件路径，获取到这个已经存在的图片，跟现在要插入的图片做全量的比对，看是否完全一样。如果一样，就说明已经存在；如果不一样，说明两张图片尽管唯一标识相同，但是并不是相同的图片

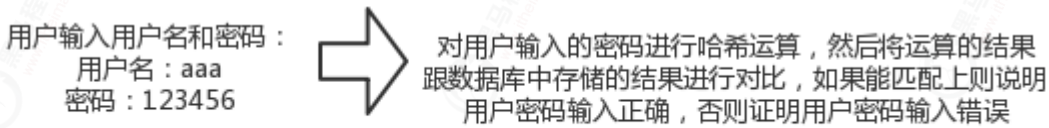
密码校验

现在来看一个很常见的业务场景，用户登陆；我们需要保存密码（比如网站用户名和密码），你要考虑如何保护这些密码数据，像下面那样直接将密码写入数据库中是极不安全的，因为任何可以打开数据库的人，都将可以直接看到这些密码。

id	username	PASSWORD
4	aaa	123456

解决的办法是将密码加密后再存储进数据库，比较常用的加密方法是使用哈希函数比如MD5，用户登录网站的时候，我们可以检验用户输入密码的哈希值是否与数据库中的哈希值相同。

id	username	PASSWORD
4	aaa	e10adc3949ba59abbe56e057f20f883e



由于哈希函数是不可逆的，即使有人打开了数据库，也无法看到用户的密码是多少。

那么存储经过哈希函数加密后的密码是否就是安全的了呢？我们先来看一下几种常见的破解密码的方法:字典破解（Dictionary Attack)，暴力破解（Brute Force Attack），这对于这种单向的哈希算法要想破解说白了就是猜密码。字典破解和暴力破解都是效率比较低的破解方式。如果你知道了数据库中密码的哈希值，你就可以采用一种更高效的破解方式，查表法（Lookup Tables）。还有一些方法，比如逆向查表法（Reverse Lookup Tables）、彩虹表（Rainbow Tables）等，都和查表法大同小异。现在我们来看一下查表法的原理。

查表法不像字典破解和暴力破解那样猜密码，它首先将一些比较常用的密码的哈希值算好，然后建立一张表，当然密码越多，这张表就越大。当你知道某个密码的哈希值时，你只需要在你建立好的表中查找该哈希值，如果找到了，你就知道对应的密码了。

那我应该如何应对这种密码破解呢？我们就需要为密码进行加盐处理

密码加盐（Salt）

盐（Salt）是什么？就是一个随机生成的字符串。我们将盐与原始密码连接（concat）在一起（放在前面或后面都可以），然后将concat后的字符串加密。采用这种方式加密密码，查表法就不灵了（因为盐是随机生成的）

id	username	PASSWORD	salt
4	aaa	e10adc3949ba59abbe56e057f20f883e	8qkk9JAWaA

对于盐的生成我们可以使用apache随机字符串工具类，对于哈希算法我们可以采用spring框架中对MD5算法的封装，下面我们提供部分代码

```
public static void main(String[] args) {
        // 生成盐 指定盐的长度
        String salt = RandomStringUtils.randomAlphanumeric(8);
        //密码加密加盐
        String digestAsHex = DigestUtils.md5DigestAsHex(("123456" + salt).getBytes());

        System.out.println("salt:"+salt+"---password:"+digestAsHex);
    }
```

如果你想演示出效果，须得导入maven坐标

```
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.5.RELEASE</version>
    </dependency>
<dependency>
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <version>2.6</version>
    </dependency>
```

这样我们使用盐来提高了密码的安全性，但是这样产生的一个问题就是，我们需要一个额外的数据库字段来存储盐，如果盐被人知晓仍然是可以被破解的，接下来我们介绍一个更加安全高级的做法。

BCrypt

在spring-security中提供了一个类BCryptPasswordEncoder，实现了PasswordEncoder接口，其内部使用了BCrypt类中的相关方法，PasswordEncoder接口声明如下

```
public interface PasswordEncoder {

    /**
     * 对密码进行加密
     */
    String encode(CharSequence rawPassword);

    /**
     * 对原始密码和加密后的密码进行匹配
     */
    boolean matches(CharSequence rawPassword, String encodedPassword);

}
```

在BCryptPasswordEncoder中采用SHA-256 +随机盐对密码进行加密，其中SHA系列就是一种哈希算法，这个类中对接口的两个方法进行了实现

（1）加密(encode)：使用SHA-256+随机盐把用户输入的密码进行hash处理，得到密码的hash值，然后将其存入数据库中，注意BCryptPasswordEncoder的好处就是我们向数据库中进行存储的时候无需开辟额外的存储字段存储盐，只需存储对密码加密的结果即可，而且我们也不知道盐是多少，这个盐的生成是在BCryptPasswordEncoder的encode方法内部去生成的，并且它会将这个盐散落在最终加密好的结果中，到时候我们进行验证的时候它会自主的从加密串中提取盐，然后进行验证。

（2）密码匹配(matches)：用户登录时，密码匹配阶段并没有进行密码解密（因为密码经过Hash处理，是不可逆的），而是使用相同的算法把用户输入的密码进行hash处理，得到密码的hash值跟之前存储在数据库中已经加密的值按照规则去匹配。

对于BCryptPasswordEncoder的具体使用我们写了一个简单测试代码如下：

```
public static void main(String[] args) {
    // 创建BCryptPasswordEncoder对象
        BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder();
    // 对密码进行加密
        String pass = bCryptPasswordEncoder.encode("admin");
    // 对密码进行验证
        boolean matches = bCryptPasswordEncoder.matches("admin", pass);
    // 输出加密结果及验证结果
        System.out.println(pass+"-----"+matches);

    }
```

散列函数

前面的章节中讲到，散列函数是设计一个散列表的关键。它直接决定了散列冲突的概率和散列表的性能。不过，相对哈希算法的其他应用，散列函数对于散列算法冲突的要求要低很多。即便出现个别散列冲突，只要不是过于严重，我们都可以通过开放寻址法或者链表法解决。

不仅如此，散列函数对于散列算法计算得到的值，是否能反向解密也并不关心。散列函数中用到的散列算法，更加关注散列后的值是否能平均分布，也就是，一组数据是否能均匀地散列在各个槽中。

除此之外，散列函数执行的快慢，也会影响散列表的性能，所以，散列函数用的散列算法一般都比较简单，比较追求效率。

负载均衡

对于负载均衡的算法有很多，比如轮询、随机、加权轮询等。那如何才能实现一个会话粘滞（session sticky）的负载均衡算法呢?

也就是说，在同一个客户端上，一次会话中的所有请求都路由到同一个服务器上。最直接的方法就是，维护一张映射关系表，这张表的内容是客户端 IP 地址或者会话 ID 与服务器编号的映射关系。客户端发出的每次请求，都要先在映射表中查找应该路由到的服务器编号，然后再请求编号对应的服务器。这种方法简单直观，但也有几个弊端：

1：如果客户端很多，映射表可能会很大，比较浪费内存空间；

2：客户端下线、上线，服务器扩容、缩容都会导致映射失效，这样维护映射表的成本就会很大

如果借助哈希算法，这些问题都可以非常完美地解决。我们可以通过哈希算法，对客户端 IP 地址或者会话 ID 计算哈希值，将取得的哈希值与服务器列表的大小进行取模运算，最终得到的值就是应该被路由到的服务器编号。这样，我们就可以把同一个 IP 过来的所有请求，都路由到同一个后端服务器上。

数据分片

举例说明如下：

需求：现有2T的日志文件，里面记录的是网站每天用户的搜索关键字，我们需要统计出每个关键词被搜索的次数，如何实现？

这里有两个重难点：1：日志数据太大，一台机器的内存不够加载，2：如果只用一台机器来处理的话时间会非常的长

解决方案：对数据进行分片，然后用多台机器并行处理，提高处理速度，比如我们用n台机器并行处理，我们从搜索记录的日志文件中，依次读出每个搜索关键词，并且通过哈希函数计算哈希值，然后再跟 n 取模，最终得到的值，就是应该被分配到的机器编号。这样，哈希值相同的搜索关键词就被分配到了同一个机器上。也就是说，同一个搜索关键词会被分配到同一个机器上。每个机器会分别计算关键词出现的次数，最后合并起来就是最终的结果。实际上，这里的处理过程也是 MapReduce 的基本设计思想。

针对这种海量数据的处理问题，我们都可以采用多机分布式处理。借助这种分片的思路，可以突破单机内存、CPU 等资源的限制。

2、面试实战

Two Sum 系列问题在 LeetCode 上有好几道，我们挑出有代表性的几道，看一下这种问题怎么解决。

2.1、1. 两数之和

腾讯，字节，阿里最近面试题，1. 两数之和

注意：数组nums中的数是无序的，但是我们又不能对其排序，因为最终要返回的是其下标，如果重新排序后下标肯定变了。

1，暴力解决，穷举

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        //暴力解决
        for (int i=0;i<nums.length;i++) {
            for (int j=i+1;j<nums.length;j++) {
                if (nums[i] + nums[j] == target) {
                    return new int[]{i,j};
                }
            }
        }
        return new int[]{};
    }
}
```

时间复杂度O(n^2)，空间复杂度是O(1)

2，通过哈希表构造缓存，降低时间复杂度

```java
class Solution {
    public int[] twoSum(int[] nums, int target) {
        //特殊判断
        if (nums==null || nums.length <=1) {
            return new int[]{};
        }
        //构造哈希
        Map<Integer,Integer> hash = new HashMap();
        for (int i=0;i<nums.length;i++) {
            if (hash.containsKey(target-nums[i])) {
                return new int[]{hash.get(target-nums[i]),i};
            }else {
                hash.put(nums[i],i);
            }
        }
        return new int[]{};
    }
}
```

时间复杂度O(N)，空间复杂度O(N)

2.2、167. 两数之和 II

字节，谷歌最近面试题，167. 两数之和 II - 输入有序数组

输入数组元素有序，可以使用双指针夹逼思想

```java
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        //数组元素已经有序了,可以使用双指针夹逼,因此定义双指针
        int i=0;
        int k=numbers.length -1;

        while (i<k) {
            int sum = numbers[i] + numbers[k];
            if (sum == target) {
                return new int[]{i+1,k+1};
            }else if (sum < target) {
                i++;
            }else {
                k--;
            }
        }
        return new int[]{};
    }
}
```

2.3、15. 三数之和

字节，华为，腾讯最近面试题，15. 三数之和

两套思路：

思路1：使用hash，分解为n-2个两数之和

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        //特殊判断
        if (nums == null || nums.length <3) {
            return Collections.EMPTY_LIST;
        }
        Set<List> set = new HashSet();
        for (int i=0;i<nums.length-2;i++) {
            int target = -nums[i];
            Map<Integer,Integer> map = new HashMap();
            for (int j = i+1;j<nums.length;j++) {
                if (map.containsKey(target-nums[j])) {
                    List<Integer> list = new ArrayList();
                    list.add(nums[i]);
                    list.add(target-nums[j]);
                    list.add(nums[j]);
                    list.sort(Comparator.naturalOrder());
                    set.add(list);
                }else {
                    map.put(nums[j],j);
                }
            }
        }
        return new ArrayList(set);
    }
}
```

> 复杂度高：O(n^2)

思路2：对数组排序，然后使用双指针夹逼（左右指针）

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        //特殊判断
        if (nums == null || nums.length < 3) {
            return new ArrayList();
        }
        //对数组排序
        Arrays.sort(nums);
        List<List<Integer>> result = new ArrayList();
        for (int i=0;i<nums.length-2;i++) {
```

```
            if (nums[i] > 0) {
                break;
            }else if (i> 0 && nums[i] == nums[i-1]) {
                continue;
            }
            //定义双指针
            int j=i+1;
            int k = nums.length-1;
            while (j<k) {

                if (nums[i] + nums[j] + nums[k] ==0) {
                    while (j<k && nums[j] == nums[j+1]) {
                        j++;
                    }
                    while (j<k && nums[k] == nums[k-1]) {
                        k--;
                    }

                    List<Integer> list = new ArrayList();
                    list.add(nums[i]);
                    list.add(nums[j]);
                    list.add(nums[k]);
                    result.add(list);
                    j++;
                    k--;
                }else if (nums[i] + nums[j] + nums[k] < 0) {
                    j++;
                }else {
                    k--;
                }
            }
        }

        return result;
    }
}
```

总结：

对于 TwoSum 系列问题，一个难点就是给的数组**无序**。对于一个无序的数组，我们似乎什么技巧也没有，只能暴力穷举所有可能。

一般情况下，我们会首先把数组排序再考虑双指针（左右指针）技巧。TwoSum 启发我们，HashMap 或者 HashSet 也可以帮助我们处理无序数组相关的简单问题。

有时候双指针不仅是应用在一个数组（或链表）上，还可以应用到多个数组（或链表）上，如下：

2.4、4. 寻找两个正序数组的中位数

字节，携程最近面试题，4. 寻找两个正序数组的中位数

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        //构造新的排序数组
        int[] data = new int[nums1.length+nums2.length];
        int k=0;//新数组下标,填充值
        //定义两个指针分别对应两个数组的下标
        int m=0;
        int n=0;
        while (m < nums1.length && n < nums2.length) {
            data[k++] = nums1[m] > nums2[n] ? nums2[n++]:nums1[m++];
        }
        while (m < nums1.length) {
            data[k++] = nums1[m++];
        }
        while (n < nums2.length) {
            data[k++] = nums2[n++];
        }
        int len = data.length;
        if (len % 2 ==0) {
            return (data[ (len / 2) -1 ] + data[ len / 2 ])/2.0;
        }else {
            return data[len / 2];
        }
    }
}
```

2.5、21. 合并两个有序链表

快手，百度最近面试题，21. 合并两个有序链表

快慢指针+哨兵

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        //特殊判断
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        ListNode l3 = new ListNode(-1);
        ListNode result = l3;
        while (l1!= null && l2 !=null) {
            if (l1.val > l2.val) {
```

```
                l3.next = l2;
                l2 = l2.next;
            }else {
                l3.next = l1;
                l1 = l1.next;
            }
            l3 = l3.next;
        }
        if (l1 != null) {
            l3.next = l1;
        }
        if (l2 != null) {
            l3.next = l2;
        }
        return result.next;
    }
}
```

2.6、23. 合并K个升序链表

vivo，阿里，字节最近面试题，23. 合并K个升序链表

1：最简单的做法，逐个依次合并（两两合并）

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        //特殊判断
        if (lists == null ) {
            return null;
        }
        ListNode result = null;

        for (int i=0;i<lists.length;i++) {
            result = mergeTwoLists(result,lists[i]);
        }
        return result;

    }


    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        //特殊判断
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        ListNode l3 = new ListNode(-1);
```

```
        ListNode result = l3;
        while (l1!= null && l2 !=null) {
            if (l1.val > l2.val) {
                l3.next = l2;
                l2 = l2.next;
            }else {
                l3.next = l1;
                l1 = l1.next;
            }
            l3 = l3.next;
        }
        if (l1 != null) {
            l3.next = l1;
        }
        if (l2 != null) {
            l3.next = l2;
        }
        return result.next;
    }
}
```

> 时间复杂度是O(k^2 * n)：k是数组的长度，n是链表最长长度

2：**使用优先级队列**

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        //特殊判断
        if (lists == null || lists.length < 1) {
            return null;
        }
        //构造优先级队列
        PriorityQueue<ListNode> queue = new PriorityQueue<>(new Comparator<ListNode>() {
            @Override
            public int compare(ListNode o1, ListNode o2) {
                if (o1.val < o2.val) {
                    return -1;
                }
                if (o1.val > o2.val) {
                    return 1;
                }
                return 0;
            }
        });

        //构造哨兵
        ListNode dummy = new ListNode(-1);
        ListNode result = dummy;
        //将各链表头节点加入队列
        for (ListNode head:lists) {
            if (head!=null) {
```

```
                queue.offer(head);
            }
        }

        while (!queue.isEmpty()) {
            //依次从队列中获取最小值节点queue.poll();
            dummy.next = queue.poll();
            dummy = dummy.next;
            //将各链表下一节点加入队列
            if (dummy.next != null) {
                queue.offer(dummy.next);
            }
        }

        return result.next;
    }
}
```

> 时间复杂度：O(n*log(k))，`n`是所有链表中元素的总和，`k`是链表个数。

2.7、3. 无重复字符的最长子串

[字节，华为，阿里最近面试题，3. 无重复字符的最长子串]

典型的"滑动窗口"

核心思想：维护一个窗口（有点像队列），不断滑动，然后更新答案，下方是**滑动窗口问题**解决思路的**模板**

> 滑动窗口是一种高级的双指针思想

```
int left = 0, right = 0;
// data代表要处理的数据流,window代表我们维护的窗口
while ( right < data.size() ) {`

    while (window needs shrink) { //当窗口需要缩小时
        //将一些数据移出窗口
        window.remove( data[left] );

        //左测缩小窗口
        left++;

        //窗口内数据更新等其他操作
        ......
    }

    //向窗口内添加数据---这个步骤有时候需要在前面
    window.add( data[right] );

    // 右侧增大窗口
```

```
    right++;

    //窗口内数据更新等其他操作
    ......

}
```

> 这个算法技巧的时间**复杂度是 O(N)**，比暴力算法要高效得多，**一般用于处理字符串字串问题**

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        //特殊判断
        if (s == null || s.length() < 1) {
            return 0;
        }

        int left,right;
        left = right = 0;
        //用于判断是否有重复字符
        Set<Character> set = new HashSet();
        //定义最大值
        int max = 0;
        while ( right < s.length()) {
            //当窗口需要缩小时
            while (set.contains(s.charAt(right))) {
                set.remove(s.charAt(left));
                left++;
            }

            set.add(s.charAt(right));
            right++;
            max = Math.max(max,right - left);
        }
        return max;
    }
}
```

2.8、76. 最小覆盖子串

字节，华为，腾讯最近面试题，76. 最小覆盖子串

滑动窗口

```
class Solution {
     public String minWindow(String s, String t) {
        //特殊判断
        if (s == null || t == null || s.length() < t.length()) {
            return "";
```

```
        }
        //构造两个哈希表
        int[] dict = new int[128];//字典:记录t中所有字符及出现的次数
        int[] window = new int[128];//记录滑动窗口中每个字符出现的次数

        //记录s
        for (int i=0;i<t.length();i++) {
            dict[t.charAt(i)]++;
        }

        //定义滑动窗口左右边界指针left, right
        int left,right;
        left = right = 0 ;
        //定义窗口内已有字典中字符的个数
        int count = 0;
        //定义窗口的最小大小
        int min = s.length()+1;//+1是为了后续判断比这个小，因为有种极端情况是最小窗口就是s的长度
        //定义返回的字符串
        String res = "";

        while ( right < s.length()) {
            char cr = s.charAt(right);
            window[cr]++;
            //判断进入窗口的这个字符是否是t中的
            if (dict[cr] > 0 && dict[cr] >= window[cr]) { //dict[cr] >= window[cr]避免了窗口中进入了过多的重复字符导致误判,比如t="ABC" 窗口中进入了"AAA"
                count++;
            }
            right++;

            //当窗口内已完全包含t中所有字符时窗口开始收缩
            while ( count == t.length()) {
                //缩小窗口求最小窗口
                char cl = s.charAt(left);

                //如果要移出窗口的这个字符是字典中,则窗口不能再收缩了,这是底线了
                if (dict[cl] > 0 && dict[cl] >= window[cl]) { //dict[cl] >= window[cl]这个地方避免了移出过多的重复字符导致误判,比如t="ABC" 窗口中有"AABC"
                    count--;
                    if (right-left < min) {
                        min = right - left;
                        res = s.substring(left,right);
                    }
                }

                window[cl]--;
                left++;
            }

        }
        return res;
    }

}
```