第四讲:字符串算法

王争

前 Google 工程师



目录

- 1. 字符串匹配算法回顾
- 2. 字典树(Trie)

字符串匹配算法回顾



字符串匹配算法回顾

- BF 算法
- RK 算法
- BM 算法
- KMP 算法
- Trie 树
- AC 自动机

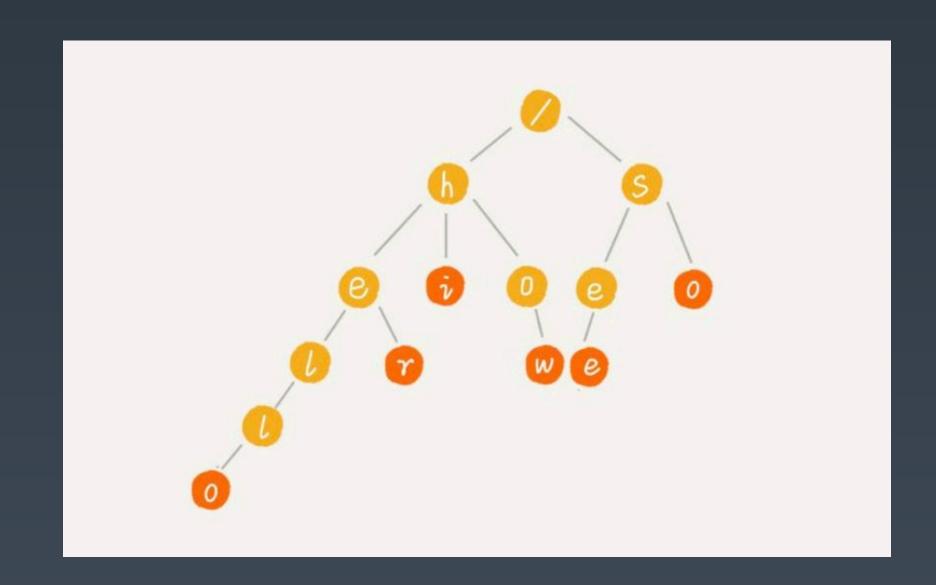
字典树 (Trie)



目录

- 1. Trie 树的介绍
- 2. Trie 树的应用场景举例

Trie 树的介绍

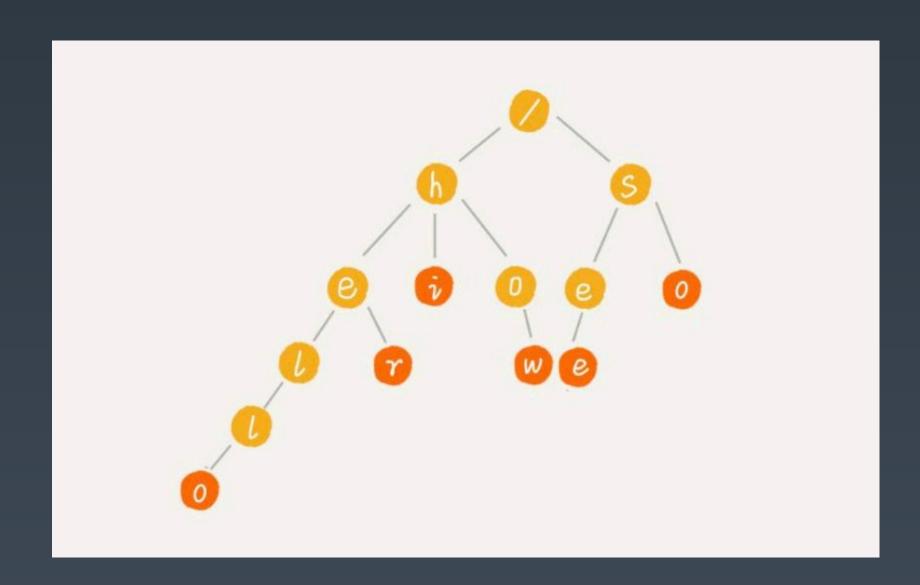


Trie 树的本质,就是利用字符串之间的公共前缀,将重复的前缀合并在一起,构建成树结构。

- 1. Trie 树是多主串匹配算法
- 2. Trie 树是一个多叉树
- 2. 根节点不包含任何信息
- 3. 每个节点表示一个字符串中的一个字符
- 4. 从根节点到红色节点的一条路径表示一个字符串



Trie 树的介绍

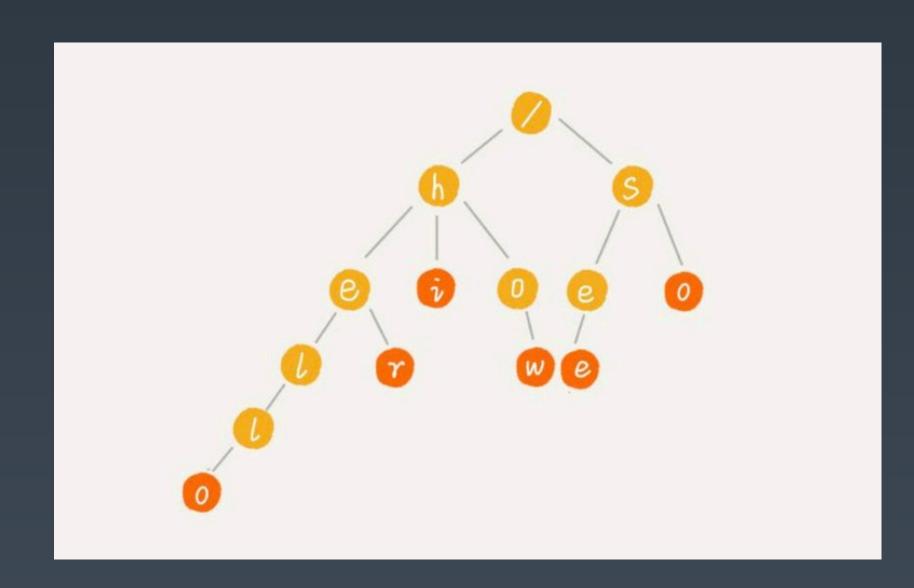


有 n 个字符串,每个字符串平均长度是 k, 要查找的字符串长度 m

- 1. 利用 KMP 算法查找的时间复杂度 O((k+m)*n)
- 2. 利用 Trie 树查找的时间复杂度 O(k)
- 3. 借助散列表查找的时间复杂度 O(k+m)
- 4. 借助红黑树查找的时间复杂度 O((k+m)*logn)



Trie 树的介绍

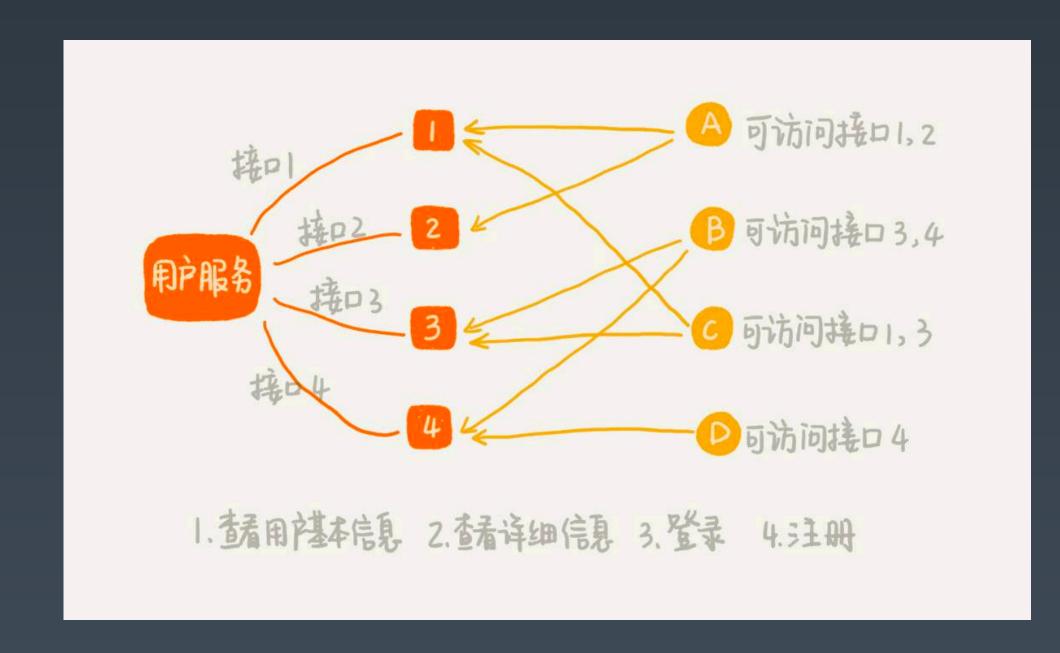


应用场景特点:

- 1. 用于多主串匹配场景、前缀匹配场景
- 2. 字符串集合中的字符串,有大量重复的公共前缀
- 3. 字符串集合趋于静态 (无频繁的增删改)
- 4. 字符集不能太大



Trie 树的应用场景举例:接口鉴权功能



接口鉴权背景介绍:

假设我们有一个微服务叫用户服务。它提供很多用户相关的接口,比如获取用户信息、注册、登录等,给公司内部的其他应用使用。

但是,并不是公司内部所有应用,都可以访问这个用户服务,也并不是每个有访问权限的应用,都可以访问用户服务的所有接口。



Trie 树的应用场景举例:接口鉴权功能

规则(省略接口GET/POST参数)

App_ID_A:
/user/info/base
/user/info/detail

App_ID-B: /user/register /user/login

App_ID_C: /user/login /user/info/base

App_ID_D: /user/register

请求举例:

App_ID_A访问/user/info/base PASS
App_ID_A访问/user/login REJECT
App_ID_C访问/user/info/detail REJECT
App_ID_D访问/user/register PASS

接口鉴权实现思路:

要实现接口鉴权功能,我们需要事先将应用对接口的访问权限规则设置好。

当某个应用访问其中一个接口的时候,我们就可以 拿应用的请求 URL,在规则中进行匹配。

如果匹配成功,就说明允许访问;如果没有可以匹配的规则,那就说明这个应用没有这个接口的访问权限,我们就拒绝服务。

Trie 树的应用场景举例:如何实现精确匹配规则?

只有当请求 URL 跟规则中配置的某个接口精确匹配时,这个请求才会被接受、处理。

规则(省略接口GET/POST参数) App_ID_A: /user/info/base /user/info/detail 请求举例: App_IO.A访问/user/info/base PASS App_ID_B: App_ID_A访问/user/login /user/register App_ID_Citig/user/info/detail REJECT /user/login App_ID_Diti可/user/register App_ID_C: PASS /user/login /user/info/base App_ID_D: /user/register

算法思路:

- 1. 不同的应用对应不同的规则集合。我们可以采用散列表来存储这种对应关系。
- 2. 针对这种匹配模式,我们可以将每个应用对应的权限规则,存储在一个字符串数组中。当用户请求到来时,我们拿用户的请求URL,在这个字符串数组中逐一匹配,匹配的算法就是我们之前学过的字符串匹配算法(比如 KMP、BM、BF等)。



Trie 树的应用场景举例:如何实现精确匹配规则?

只有当请求 URL 跟规则中配置的某个接口精确匹配时,这个请求才会被接受、处理。

App-ID-A: /user/info/base /user/register /user/login App-ID-C: /user/info/base App-ID-D: /user/register

请求举例:

App_ID_A访问/user/info/base PASS
App_ID_A访问/user/login REJECT
App_ID_C访问/user/info/detail REJECT
App_ID_D访问/user/register PASS

优化算法思路:

规则不会经常变动,所以,为了加快匹配速度,我们可以按照字符串的大小给规则排序,把它组织成有序数组。

当要查找某个 URL 能否匹配其中某条规则的时候,我们可以采用二分查找算法,在有序数组中进行匹配。

二分查找算法的时间复杂度是 O(logn)(n 表示规则的个数),这比起时间复杂度是 O(n) 的顺序遍历快了很多。

对于规则中接口长度比较长,并且鉴权功能调用量非常大的情况,这种优化方法带来的性能提升还是非常可观的。



Trie 树的应用场景举例:如何实现前缀匹配规则?

只要某条规则可以匹配请求 URL 的前缀,我们就说这条规则能够跟这个请求 URL 匹配。



算法思路:

- 1. 不同的应用对应不同的规则集合。我们采用散列表来存储这种对应关系。
- 2. 每个用户的规则集合,组织成 Trie 树这种数据结构。

Trie 树的应用场景举例: 如何实现前缀匹配规则?



```
public class Node {
   String pathDir;
   Map<String, Node> edges = new HashMap<>();
```

Trie 树中的每个节点不是存储单个字符,而是存储接口被 "/" 分割之后的子目录(比如 "/user/name" 被分割为 "user" "name" 两个子目录)。因为规则并不会经常变动,所以,在 Trie 树中,我们可以把每个节点的子节点们组织成有序数组这种数据结构。当在匹配的过程中,我们可以利用二分查找算法,决定从一个节点应该跳到哪一个子节点。



#