字符串哈希

Hash 的思想

Hash 的核心思想在于，将输入映射到一个值域较小、可以方便比较的范围。

> 这里的“值域较小”在不同情况下意义不同。
>
> 在哈希表中，值域需要小到能够接受线性的空间与时间复杂度。
>
> 在字符串哈希中，值域需要小到能够快速比较（$10^9、10^{18}$ 都是可以快速比较的）。
>
> 同时，为了降低哈希冲突率，值域也不能太小。

我们定义一个把字符串映射到整数的函数 f，这个 f 称为是 Hash 函数。

我们希望这个函数 f 可以方便地帮我们判断两个字符串是否相等。

具体来说，我们希望在 Hash 函数值不一样的时候，两个字符串一定不一样。

另外，反过来不需要成立。我们把这种条件称为是单侧错误。

我们需要关注的是什么？

时间复杂度和 Hash 的准确率。

通常我们采用的是多项式 Hash 的方法，即 $f(s)=\sum s[i]\times b^i\pmod{M}$。

这里面的 b 和 M 需要选取得足够合适才行，以使得 Hash 函数的值分布尽量均匀。

如果 b 和 M 互质，在输入随机的情况下，这个 Hash 函数在 $[0,M)$ 上每个值概率相等，此时单次比较的错误率为 $\frac 1 M$。所以，哈希的模数一般会选用大质数。

Hash 的实现

参考代码：（效率低下的版本，实际使用时一般不会这么写）

```
using std::string;

const int M = 1e9 + 7;
const int B = 233;

typedef long long ll;

int get_hash(const string& s) {
  int res = 0;
  for (int i = 0; i < s.size(); ++i) {
    res = (ll)(res * B + s[i]) % M;
  }
```

```
  return res;
}

bool cmp(const string& s, const string& t) {
  return get_hash(s) == get_hash(t);
}
```

Hash 的分析与改进

错误率

若进行 n 次比较，每次错误率 $\frac{1}{M}$那么总错误率是 $1-(1-\frac1M)^n$在随机数据下，若 $M=10^9+7,n=10^6$错误率约为 $\frac1{1000}$并不是能够完全忽略不计的。

所以，进行字符串哈希时，经常会对两个大质数分别取模，这样的话哈希函数的值域就能扩大到两者之积，错误率就非常小了。

多次询问子串哈希

单次计算一个字符串的哈希值复杂度是 $O(n)$其中 n 为串长，与暴力匹配没有区别，如果需要多次询问一个字符串的子串的哈希值，每次重新计算效率非常低下。

一般采取的方法是对整个字符串先预处理出每个前缀的哈希值，将哈希值看成一个 b 进制的数对 M 取模的结果，这样的话每次就能快速求出子串的哈希了：

令 $f_i(s)$ 表示 $f(s[1..i])$那么 $f(s[l..r])=\frac{f_r(s)-f_{l-1}(s)}{b^{l-1}}$其中 b^{l-1} 也可以预处理出来，用乘法逆元或者是在比较哈希值时等式两边同时乘上 b 的若干次方化为整式均可。

这样的话，就可以在 $O(n)$ 的预处理后每次 $O(1)$ 地计算子串的哈希值了。

Hash 的应用

字符串匹配

求出模式串的哈希值后，求出文本串每个长度为模式串长度的子串的哈希值，分别与模式串的哈希值比较即可。

最长回文子串

二分答案，判断是否可行时枚举回文中心（对称轴），哈希判断两侧是否相等。需要分别预处理正着和倒着的哈希值。时间复杂度 $O(n\log n)$

这个问题可以使用 manacher 算法在 $O(n)$ 的时间内解决。

参考链接

Oi Wiki