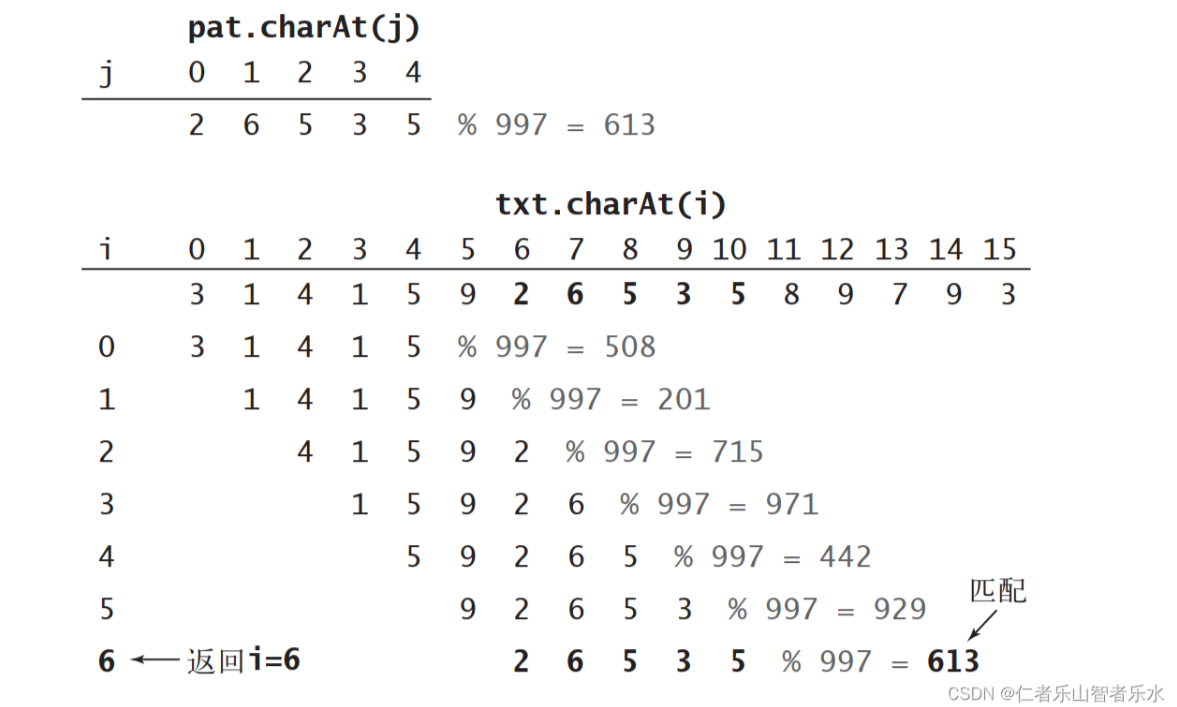
<https://blog.csdn.net/qq_39559641/article/details/122284388>

M.O.Rabin 和 R.A.Karp 发明了一种完全不同的基于散列的字符串查找算法。我们需要计算模式字符串的散列函数，然后用相同的散列函数计算文本中所有可能的 M 个字符的子字符串散列值并寻找匹配。如果找到了一个散列值和模式字符串相同的子字符串，那么再继续验证两者是否匹配。这个过程等价于将模式保存在一张散列表中，然后在文本的所有子字符串中进行查找。但不需要为散列表预留任何空间，因为它只会含有一个元素。根据这段描述直接实现的算法将会比暴力子字符串查找算法慢很多（因为计算散列值将会涉及字符串中的每个字符，成本比直接比较这些字符要高得多）。Rabin 和 Karp 发明了一种能够在常数时间内算出 M 个字符的子字符串散列值的方法（需要预处理），这样就得到了在实际应用中的运行时间为线性级别的字符串查找算法。

**基本思想**

长度为 M 的字符串对应着一个 R 进制的 M 位数。为了用一张大小为 Q 的散列表来保存这种类型的键，需要一个能够将 R 进制的 M 位数转化为一个 0 到 Q-1 之间的 int 值散列函数。除留余数法是一个很好的选择：将该数除以 Q 并取余。在实际应用中会使用一个随机的素数Q，在不溢出的情况下选择一个尽可能大的值。（因为我们并不会真的需要一张散列表。）理解这个方法最简单的办法就是取一个较小的 Q 和 R=10 的情况，如下所示。要在文本3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3中找到模式2 6 5 3 5，首先要选择散列表的大小 Q（在这个例子中是 997），则散列值为26535 % 997 = 613，然后计算文本中所有长度为 5 个数字的子字符串的散列值并寻找匹配。在这个例子中，在找到 613 的匹配之前，得到的散列值分别为 508、 201、 715、 971、 442 和 929，如下图所示：



**计算函数散列**

对于 5 位的数值，只需使用 int 值即可完成所有所需的计算。但如果 M 是 100 或者1000 怎么办？这里使用的是 Horner 方法，代码如下面所示。这段代码计算了用 char 值数组表示的 R 进制的 M 位数的散列函数，所需时间与 M 成正比。（将 M 作为参数传递给该方法，这样就可以将它同时用于模式字符串和正文。）对于这个数中的每一位数字，将散列值乘以 R，加上这个数字，除以 Q 并取其余数。例如，这样计算示例模式字符串散列值的过程如下图解所示。我们也可以用同样的方法计算文本中的子字符串散列值，但这样一来字符串查找算法的成本就将是对文本中的每个字符进行乘法、加法和取余计算的成本之和。在最坏情况下这需要 NM 次操作，相对于暴力子字符串查找算法来说并没有任何改进。

**Horner方法，用于除留余数法计算散列值**

private long stringHash(String str, int m) {

    long hash = 0;

    for(int i = 0; i < m; i++) {

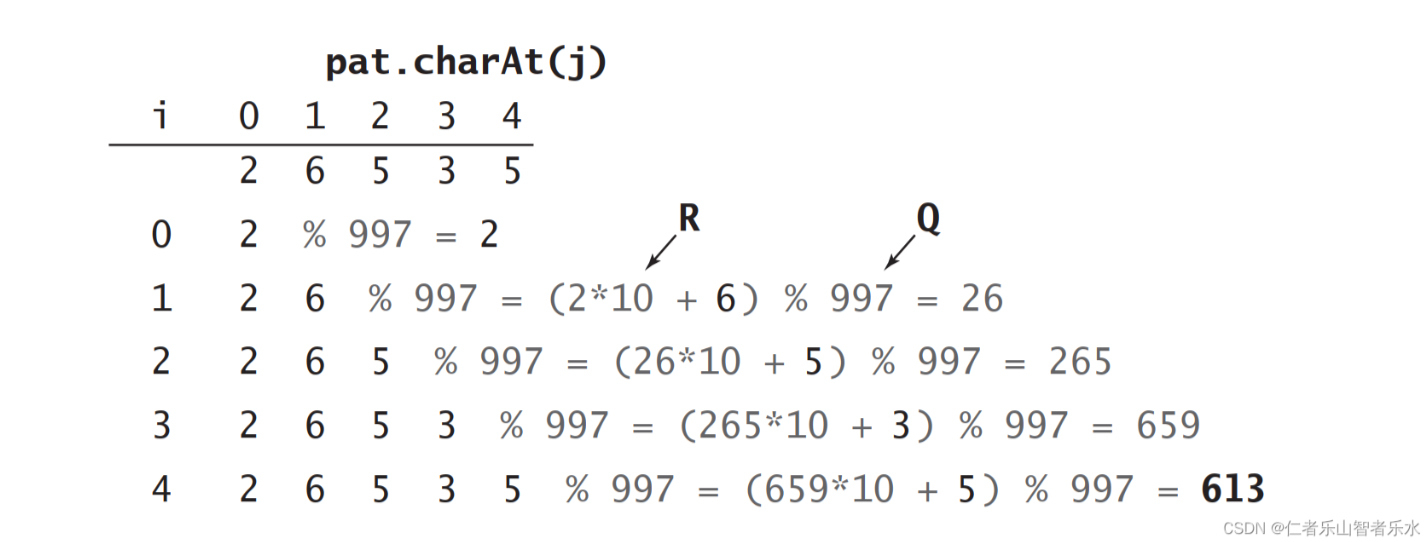
        hash = (hash \* R + str.charAt(i)) % Q;

    }

    return hash;

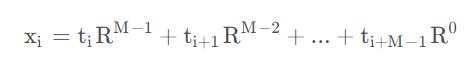
}

**使用 Horner 方法计算模式字符串的散列值图解**



**关键思想**

Rabin-Karp 算法的基础是对于所有位置 i，高效计算文本中 i+1 位置的子字符串散列值。这可以由一个简单的数学公式得到。我们用ti表示txt.charAt(i)，那么文本txt中起始于位置i的含有M个字符的字符串对应的数即为：

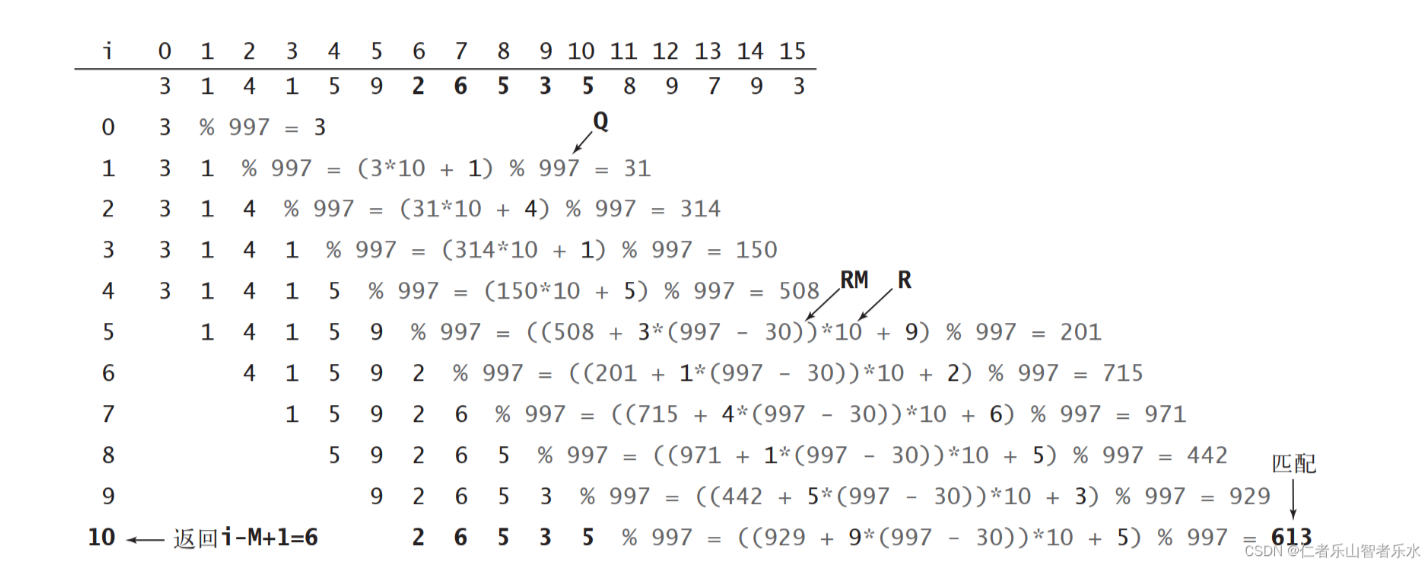


假设已知h(Xi)=Xi mod Q。将模式字符串右移一位即等价于将Xi替换为：

image.png

即将它减去第一个数字的值，乘以 R，再加上最后一个数字的值。现在，关键的一点在于不需要保存这些数的值，而只需要保存它们除以 Q 之后的余数。取余操作的一个基本性质是如果在每次算术操作之后都将结果除以 Q 并取余，这等价于在完成了所有算术操作之后再将最后的结果除以 Q 并取余。曾经在用 Horner 方法实现除留余数法时利用过这个性质。这么做的结果就是无论 M 是 5、100 还是 1000，都可以在常数时间内高效地不断向右一格一格地移动 。

**图解**



**蒙特卡罗法**

在文本 txt 中找到散列值与模式字符串相匹配的一个 M 个字符的子字符串之后，你可能会逐个比较它们的字符以确保得到了一个匹配而非相同的散列值。 我们不会这么做，因为这需要回退文本指针。作为替代，这里将散列表的“规模” Q 设为任意大的一个值，因为我们并不会真构造一张散列表而只是希望用模式字符串验证是否会产生冲突。我们会取一个大于10^20的long型值，使得一个随机键的散列值与模式字符串冲突的概率小于10^20。这是一个极小的值，如果它还不够小，你可以将这种方法运行两遍，这样失败的几率将会小于10^-40。这是蒙特卡罗法的一种著名早期应用，既可以保证运行时间，失败的概率又非常小，检查匹配的其他方法可能很慢，但能够保证正确性，这种算法被称为拉斯维加斯算法。

**实现类**

package com.design.string;

public class RabinKarp {

    private int R = 26; // 进制

    private long Q = 200000111111111L; // 一个很大的素数

    private int M; // 模式串长度

    private long patHash; // 模式串的哈希

    private long RM; // R^M % Q

    public RabinKarp(String pattern) {

        this.M = pattern.length();

        this.RM = 1;

        for(int i = 0; i < this.M; i++) {

            this.RM = this.RM \* this.R % this.Q;

        }

        this.patHash = this.stringHash(pattern, M);

    }

    private long stringHash(String str, int m) {

        long hash = 0;

        for(int i = 0; i < m; i++) {

            hash = (hash \* R + str.charAt(i)) % Q;

        }

        return hash;

    }

    public int search(String text) {

        // 文本比模式串还短

        int n = text.length();

        if(n < M) {

            return -1;

        }

        long tempHash = this.stringHash(text, M);

        // 一开始就匹配

        if(tempHash == this.patHash) {

            return 0;

        }

        for(int i = M; i < n; i++) {

            tempHash = tempHash \* R % Q;

            tempHash = tempHash % Q;

            if(this.patHash == tempHash) {

                return i - M + 1;

            }

        }

        return -1;

    }

    public static void main(String[] args) {

        String pat = "sheet";

        RabinKarp rabinKarp = new RabinKarp(pat);

        System.out.println(rabinKarp.search("ssheet"));

        System.out.println(rabinKarp.search("sheeet"));

    }

}

**测试结果**

