pat表示模式串，长度为M，txt表示文本串，长度为N。KMP 算法是在txt中查找子串pat，如果存在，返回这个子串的起始索引，否则返回 -1。

分析： 为了描述状态转移图，我们定义一个二维 dp 数组，它的含义如下：

```
dp[j][c] = next
0 <= j < M，代表当前的状态
0 <= c < 256，代表遇到的字符（ASCII 码）
0 <= next <= M，代表下一个状态

dp[4]['A'] = 3 表示：
当前是状态 4，如果遇到字符 A，
pat 应该转移到状态 3

dp[1]['B'] = 2 表示：
当前是状态 1，如果遇到字符 B，
pat 应该转移到状态 2
```

这个 dp 数组的定义和刚才状态转移的过程，我们可以先写出 KMP 算法的 search 函数代码：

```
int search(String txt) {
    int M = pat.size();
    int N = txt.size();
    // pat 的初始态为 0
    int j = 0;
    for (int i = 0; i < N; i++) {
        // 当前是状态 j，遇到字符 txt[i]，
        // pat 应该转移到哪个状态？
        j = dp[j][txt[i]];
        // 如果达到终止态，返回匹配开头的索引
        if (j == M) return i - M + 1;
    }
    // 没到达终止态，匹配失败
    return -1;
}

void KMP(String pat) {
        int M = pat.size();
        // dp[状态][字符] = 下个状态
        dp = new int[M][256];
        // base case
        dp[0][pat[0]] = 1;
        // 影子状态 X 初始为 0
        int X = 0;
        // 当前状态 j 从 1 开始
        for (int j = 1; j < M; j++) {
            for (int c = 0; c < 256; c++) {
```

```
                if (pat[j] == c)
                    dp[j][c] = j + 1;
                else
                    dp[j][c] = dp[X][c];
            }
            // 更新影子状态
            X = dp[X][pat[j]];
        }
    }
```