# CHAPTER 1

「模型」是一种世界观。建模者根据自己的世界观，对事物的性质加以选择和抽象，构成一套属于他的模型。

例如，对于一张图片，至少有两种描述它的方式：

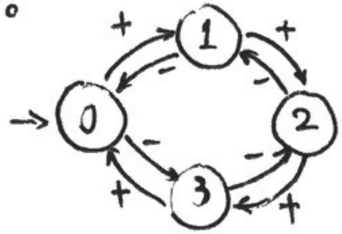
1° 将其视作二维平面上的格点集。每一个格点都很小，小到人眼无法分辨。另外，每个格点都有其对应的颜色。

2° 将其视作许多物体构成的序列。物体依照从远及近的次序堆放在一起。

这两种截然不同的观点造就了不同的图片模型：基于像素的模型和基于对象的模型。在处理实际问题时，二者适用范围有不同，因为其内在的侧重点不同。

本章研究的计算模型是有限自动机模型，它是一种很简单的抽象。它认为：所谓计算，无非是给定一串输入，据此产生「Yes」或「No」的输出。而且，计算是按步就班、只顾眼前的：我目前处在某一状态，在看见下一个输入字符后，我立即根据这一字符做出决断，转移到新的状态。

数字电路中最为基础的时序部件——计数器——就是一种有限自动机。在读到下一指令后，立即作出响应，进入新的状态。




似乎有限自动机这一简单的模型具有很强大的功能（计数就是一例），那么，它是否有局限？它的性质怎么样？这些将是我们考察的话题。

**def** 有限自动机(DFA)（“有限”上方加注：确定的）:

一个五元组 $(Q, \Sigma, \delta, q_0, F)$, 其中

1° $Q$ 是一个有限集合。它指定了该DFA所有可能的状态。

2° $\Sigma$ 是一个有限集合。它指定了该DFA所认识的字符集.

3° $\delta: Q\times\Sigma \to Q$. 它指定了该DFA在各种状态下读到各种字符应作出何种状态转移。

4° $q_0 \in Q$ 是该DFA的起始状态。

5° $F \subseteq Q$ 是该DFA「接纳」的状态集。

**e.g.** 在上一页的计数器DFA中,

$Q=\{0,1,2,3\}$, $\Sigma=\{+,-\}$

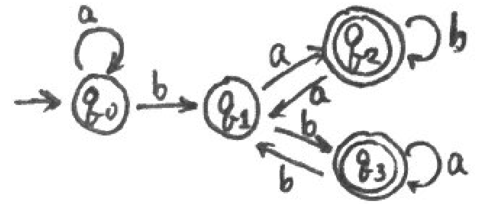
$\delta$:

| 输入＼状态 | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| + | 1 | 2 | 3 | 0 |
| − | 3 | 0 | 1 | 2 |

, $q_0=0$, $F=\emptyset$.

若输入字符串为 $++-+$, 则自动机状态变化为 $0\to1\to2\to1\to2$, 终态未被「接纳」。

**e.g.**




**problem** 删除图中任何一个箭头, 对应的还是一个DFA吗?

$Q=\{q_0,q_1,q_2,q_3\}$, $\Sigma=\{a,b\}$

$\delta$:

| 输入＼状态 | $q_0$ | $q_1$ | $q_2$ | $q_3$ |
|---|---|---|---|---|
| a | $q_0$ | $q_2$ | $q_1$ | $q_3$ |
| b | $q_1$ | $q_3$ | $q_2$ | $q_1$ |

, $F=\{q_2,q_3\}$

若输入字符串为 aabaa, 则状态变化为
$q_0\to q_0\to q_0\to q_1\to q_2\to q_1$, 未被接纳。

若输入字符串为 abb, 则状态变化为
$q_0\to q_0\to q_1\to q_3$, 被接纳。

**remark** 之所以命名为「确定的」有限自动机, 是为了与后面将讨论的「非确定」相区别。目前可无视之。

上面例子中所做的状态转移演示仍然不够数学化。我们尝试将其变得严谨。

**def** DFA的计算流程。

设 $M=(Q,\Sigma,\delta,q_0,F)$ 是一台DFA。对其输入字符串 $w=w_1w_2w_3\dots w_n$ $(w_i\in\Sigma)$。

若序列 $r_0 r_1 \cdots r_n$ 满足

1° $r_0 = q_0$；$\forall i: r_i \in Q$.

2° $\forall i \in [0, n-1]: r_{i+1} = \delta(r_i)$

则称序列 $r_0 r_1 \cdots r_n$ 是 M 对于输入 w 的计算流程，并称 $r_n$ 是终态。

remark 显然，对于给定 DFA 和输入，计算流程存在且唯一。

def 接纳与拒绝：若 M 对于 w 计算的终态 $r_n \in F$，则称 M 接纳 w；否则，称 M 拒绝 w.

def DFA 识别的语言：
对于 DFA M 而言，它识别的语言为
$\mathcal{L}(M) := \{ w \in \Sigma^* \mid M \text{接纳} w \}$

def 正则语言.
设 S 是一个集合。若存在某台 DFA，M，满足 $\mathcal{L}(M) = S$，则称 S 是正则语言。

为方便理解，可认为 DFA 是一种筛选器。把所有可能的输入喂给一台 DFA，其中被其接纳的，即为这台 DFA 所「识别」的语言。（当然，也就必为正则语言）。

反过来，给定一个集合，我们希望弄清它是否为正则语言，那么只要去考察是否有某台 DFA 所「识别」的语言恰好就为该集合。

我们应当能看清，「正则语言」的疆界，标志着 DFA 识别能力的疆界。如果万物皆为正则语言，则万物皆可被 DFA 计算；反之，则 DFA 有其能力瓶颈。

鉴于此，我们很有必要透彻研究正则语言的性质。谨记：以下过程中，我们的目的是探索正则语言的边界（从而得到 DFA 的边界），而不是开发新的计算模型或是研究 DFA 的设计。

首先，我们看看正则语言类在一些常用运算下是否封闭。

def 一些关于正则语言类的运算

设 $A, B$ 均为正则语言，定义

1° $A\cup B := \{w \mid w\in A \text{ or } w\in B\}$

2° $A\cap B := \{w \mid w\in A \text{ and } w\in B\}$

3° $A\circ B := \{xy \mid x\in A,\ y\in B\}$

4° $A^* := \{x_1x_2\cdots x_k \mid k\in\mathbb{N}_0,\ \forall i\in[1,k]: x_i\in A\}$

eg. $A=\{000, 010, 110\}$

$B=\{0abc, abb0, a, 000\}$

则 $A\cup B=\{000, 010, 110, 0abc, abb0, a\}$

$A\cap B=\{000\}$

$A\circ B=\{000\,0abc, 000abb0, 000a, 000000,$
$0100abc, 010abb0, 010a, 010000,$
$1100abc, 110abb0, 110a, 110000\}$

$A^*=\{\varepsilon, 000, 010, 110, 000000, 000010,$
$000110, 010000, 010010, 010110,$
$110000, 110010, 110110, \cdots\}$

**Theorem 1** 正则语言类对 $\cup$ 运算封闭。

即，$\forall$ 正则语言 $A_1, A_2$，我们有 $A_1\cup A_2$ 仍是正则语言。

proof

因为 $A_1$、$A_2$ 均为正则语言，故存在确定的有限自动机 $M_1$ 与 $M_2$ 分别识别 $A_1$ 与 $A_2$。我们希望构造一台新的自动机 $M$，确保 $\mathcal{L}(M) = A_1\cup A_2$。

思路十分简单。如果我们能同时运行 $M_1$ 与 $M_2$，取出它们的结果若任何一方为「接纳」则令 $M$「接纳」，反之亦然。

既然如此，$M$ 就需要保存 $M_1$ 和 $M_2$ 的运行状态——这依靠一个二维向量即可做到。

下面是严格的写法。

设 $M_1=(Q_1, \Sigma_1, \delta_1, q_1, F_1)$

$M_2=(Q_2, \Sigma_2, \delta_2, q_2, F_2)$

$M=(Q, \Sigma, \delta, q, F)$

令 $Q := Q_1\times Q_2$（构造二维向量，用一个状态保存两个子状态）

$\Sigma := \Sigma_1\cup\Sigma_2$

$\delta: Q\times\Sigma\to Q$（即 $(Q_1\times Q_2)\times\Sigma\to(Q_1\times Q_2)$）

$\delta((s_1, s_2), a) := (\delta_1(s_1, a), \delta_2(s_2, a))$

（两台子机器独立运转）

$q := (q_1, q_2)$

$F := \{(s_1, s_2) \mid s_1 \in F_1 \text{ or } s_2 \in F_2\}$

$= (Q_1 \times F_2) \cup (Q_2 \times F_1)$ ■

remark 上述证明有一点瑕疵，但只须多加讨论即可解决。问题在于$a \notin \Sigma_1$或$a \notin \Sigma_2$时，$\delta$函数并非良定义。请修改证明以消除该瑕疵。

~~problem 证明下面的定理。~~

Theorem2 正则语言类对∩运算封闭。

proof 留作练习。■

那么，同样的方法是否能用于证明对∘运算的封闭性呢？没有那么容易。问题的关键在于：我们不能预先知晓应在何处「切断」字符串。比方说，$A=\{ab, cd, abcd\}$，$B=\{a, ab, cd\}$，那么，输入$w=abcdab$时，机器怎能预知是按照ab/……方式切断，还是按照abcd/…方式切断呢？

有人会反驳道：在证明$A \cup B$正则时，不也有两种选择吗？一是$M_1$接纳，二是$M_2$接纳。则开始并不确定哪一方会接纳，故要把$M_1$与$M_2$均模拟出来。那么，现在为何不能把所有可能性模拟出来呢？

答案是：一般而言，可能性不只两种，而或许有成千上万种；更致命的是，可能性的数目在未知A与B时是不能确定的——这就导致我们不可以构造k维向量（k是定值）来模拟所有可能性。

不过，「模拟可能性」这一点提得并非没有价值。即将介绍的「非确定有限状态自动机」本质上就是在模拟所有可能性，只不过不采取纯粹枚举的形式，而采取了较为精巧的办法。我们介绍它，是为了引入「分叉机制」。我们还将证明它与DFA的等价性，进而方便地构造出「正则语言类对∘运算封闭」的证明。

**def** 非确定的有限状态自动机(NFA):

一个五元组 $(Q, \Sigma, \delta, q_0, F)$，其中

1° $Q$ 是一个有限集合。它指定了该NFA所有可能的状态。

2° $\Sigma$ 是一个有限集合。它指定了该NFA所认识的字符集。

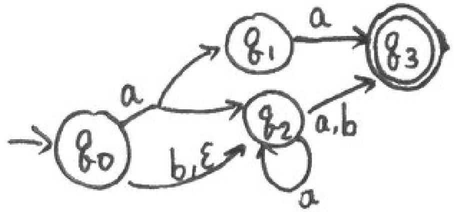
3° $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. 它指定了该NFA在各种状态下读到各种字符应作出何种转移。注意转移的结果不是单个状态，而是一组状态（当然，可能为 $\emptyset$ 或仅包含单个状态）.

4° $q_0 \in Q$ 是该NFA的起始状态。

5° $F \subseteq Q$ 是该NFA「接纳」的状态集。

对比DFA的定义，我们发现NFA的区别在于转移函数由「一转一」改为「一转多」。正是这一区别，给了DFA「分支」或者说「尝试不同选择」的可能。这就好比一个Linux程序，在需要的时候执行fork()函数，生成并行的进程，执行不同的任务。

e.g.




$Q = \{q_0, q_1, q_2, q_3\}$

$\Sigma = \{a, b\}$

$\delta$:

| 输入 \ 状态 | $q_0$ | $q_1$ | $q_2$ | $q_3$ |
|---|---|---|---|---|
| a | $\{q_1, q_2\}$ | $\{q_3\}$ | $\{q_2, q_3\}$ | $\emptyset$ |
| b | $\{q_2\}$ | $\emptyset$ | $\{q_3\}$ | $\emptyset$ |
| $\varepsilon$ | $\{q_2\}$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

$F = \{q_3\}$

**def** NFA的计算流程

设 $N = (Q, \Sigma, \delta, q_0, F)$ 是一台NFA。对其输入字符串 $w = w_1 w_2 \cdots w_n$ ($w_i \in \Sigma_\varepsilon$)。若序列 $r_0 r_1 \ldots r_n$ 满足

即 $\Sigma \cup \{\varepsilon\}$

1° $r_0 = q_0$. $\forall i: r_i \in Q$.

2° $\forall i \in [0, n-1]: r_{i+1} \in \delta(r_i)$

则称序列 $r_0 r_1 \ldots r_n$ 是 $N$ 对于输入 $w$ 的一条计算流程，并称 $r_n$ 是它的终态。

**remark** 显然，一般而言，计算流程不只一条，但必为有限多条。比如对于上面的例子而言，输入 aa，计算流程

**def** 非确定的有限状态自动机(NFA):

一个五元组 $(Q, \Sigma, \delta, q_0, F)$,其中

1° $Q$ 是一个有限集合。它指定了该NFA所有可能的状态。

2° $\Sigma$ 是一个有限集合。它指定了该NFA所认识的字符集。

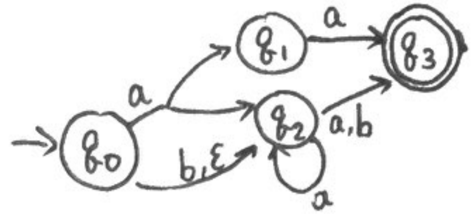
3° $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. 它指定了该NFA在各种状态下读到各种字符应作出何种转移。注意转移的结果不是单个状态,而是一组状态(当然,可能为 $\emptyset$ 或仅包含单个状态).

4° $q_0 \in Q$ 是该NFA的起始状态。

5° $F \subseteq Q$ 是该NFA「接纳」的状态集。

对比DFA的定义,我们发现NFA的区别在于转移函数由「一转一」改为「一转多」。正是这一区别,给了DFA「分支」或者说「尝试不同选择」的可能。这就好比一个Linux程序,在需要的时候执行fork()函数,生成并行的进程,执行不同的任务。

e.g.




$Q = \{q_0, q_1, q_2, q_3\}$

$\Sigma = \{a, b\}$

$\delta$:

| 输入 \ 状态 | $q_0$ | $q_1$ | $q_2$ | $q_3$ |
|---|---|---|---|---|
| a | $\{q_1, q_2\}$ | $\{q_3\}$ | $\{q_2, q_3\}$ | $\emptyset$ |
| b | $\{q_2\}$ | $\emptyset$ | $\{q_3\}$ | $\emptyset$ |
| $\varepsilon$ | $\{q_2\}$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

$F = \{q_3\}$

**def** NFA的计算流程

设 $N = (Q, \Sigma, \delta, q_0, F)$ 是一台NFA。对其输入字符串 $w = w_1 w_2 \cdots w_n$ ($w_i \in \Sigma_\varepsilon$)。若序列 $r_0 r_1 \dots r_n$ 满足 即 $\Sigma \cup \{\varepsilon\}$

1° $r_0 = q_0$. $\forall i: r_i \in Q$.

2° $\forall i \in [0, n-1]: r_{i+1} \in \delta(r_i)$

则称序列 $r_0 r_1 \dots r_n$ 是 $N$ 对于输入 $w$ 的一条计算流程,并称 $r_n$ 是它的终态。

**remark** 显然,一般而言,计算流程不只一条,但必为有限多条。比如对于上面的例子而言,输入 aa,计算流程

可以是 $q_0 \to q_1 \to q_3$，$q_0 \to q_2 \to q_3$ 或 $q_0 \to q_2 \to q_2 \to q_3$.（注意「$\varepsilon$」这一空字符可插在输入的任意位置，且可插入任意多次，故状态转移次数可能多于2）.

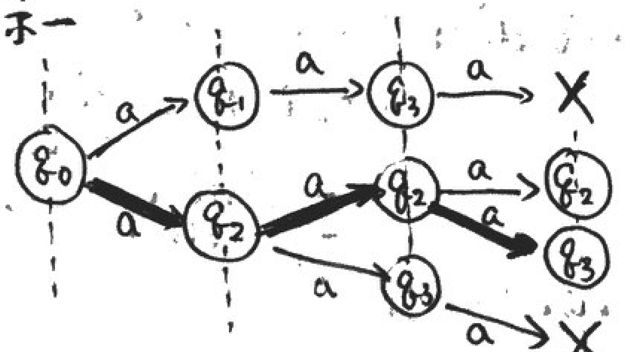
**def** NFA的接纳与拒绝：对输入 $w$，若存在某一条计算流程的终态 $r_n \in F$，则 $w$ 被 N 接纳；否则 $w$ 被 N 拒绝。

换一种定义方式，也可以说成是：

接纳 $\Longleftrightarrow F \cap \{r_n \mid r_1 \dots r_n$ 是对于输入 $w$ 的一条计算流程$\} \neq \emptyset$.

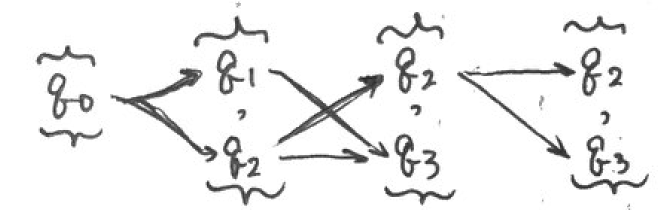
**e.g.** 对于上一页的NFA，我们用两种图示来演示其运作机制。设 $w = aaa$.

图示一




在众多选择中，仅有加粗的一条到达了「接纳」状态。但仅有一条也足够了，该NFA将接纳 aaa.

图示二




这一幅图示相对前一幅更为简洁，因为它把重复的去掉了。之所以能这么做，完全是由于NFA的转移机制 —— 它不关心之前的路径如何，只关心当下的状态为何。既然，无论是由 $q_1$ 转移到 $q_3$，还是由 $q_2$ 转移到 $q_3$，结果都是「转移到了 $q_3$」，那么又何必多费功夫保留两个分支呢？

这样的对比和观察非常重要，因为它告诉我们：尽管分支的可能性很多，分支数目爆炸性增长，但是我们可以剪掉许多冗余枝干而丝毫不影响所能到达的终态。于是，

表面上的一团乱麻不似想象中那么糟糕，我们甚至可以找到途径把NFA转化为DFA。

**Theorem3** 对任何NFA $N$，存在某个DFA $M$ 满足 $\mathcal{L}(N)=\mathcal{L}(M)$.

**proof** 基本想法：用 $M$ 来模拟 $N$。$M$ 的当前状态保存了 $N$ 当前处在的状态集合。因为 $N$ 当前至多不过是处在所有 $|Q|$ 种状态，故其当前状态集的取法是有限的。

设 $N=(Q,\Sigma,\delta,q_0,F)$

$M=(Q',\Sigma',\delta',q_0',F')$

令 $Q':=2^Q$（即N所有状态组合之集合）

$\Sigma':=\Sigma$

又定义 $E(r):=\{s\in Q\mid$ 由状态r出发，经若干字符ε可至状态s$\}$

$(\forall r\in Q)$

对 $\forall R\in Q',\ a\in\Sigma'$

$$\delta'(R,a):=\bigcup_{r\in R}\ \bigcup_{s\in\delta(r)}E(s)$$

（对照上一页图示二。$\bigcup_{r\in R}$ 相当于依次取出大括号内的每一项，$\bigcup_{s\in\delta(r,a)}E(s)$ 相当于求出项r所对应的转移目标及其邻域）

显然 $\delta':Q'\times\Sigma'\to Q'$.

$q_0':=E(q_0)$

$F':=\{R\in Q'\mid R\cap F\neq\emptyset\}$ ■

**remark.** 这一定理的证明思路直接基于上一页图示二。它用M的单个状态表达「N当前所处的所有状态」是最为精妙之处。

**corollary4** A是正则语言当且仅当存在某台NFA $N$ 满足 $\mathcal{L}(N)=A$。也就是说，NFA所能识别的语言和DFA相比，不多也不少。

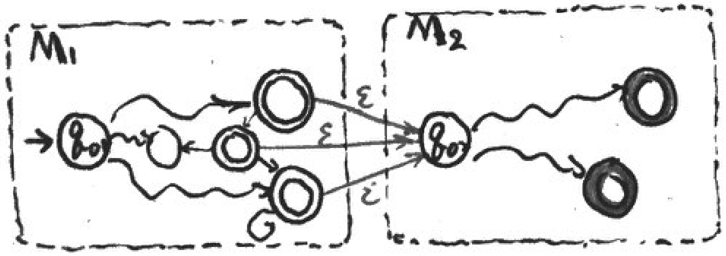
**proof** 留作习题。■

结论很明显：即使我们在DFA模型基础上引入了一定的非确定性，形成NFA模型，但在关于「什么是可被识别的语言」这一问题上，它们并无差别。是故，若我们证明某语言A可被某台NFA识别，那么A也就可被某台DFA识别，A也就是正则语言。(Corollary 4的转述)

基于此，我们来证明正则语言类对。和*运算的封闭性。

**Theorem5** 正则语言类对。运算封闭。

proof 对于任意正则语言 $A_1, A_2$，根据定义，存在两台DFA分别识别之。设 $M_1=(Q_1,\Sigma_1,\delta_1,q_{01},F_1)$ 与 $M_2=(Q_2,\Sigma_2,\delta_2,q_{02},F_2)$ 分别识别 $A_1$ 与 $A_2$。我们构造如下的一台NFA $N$，使 $\mathcal{L}(N)=A_1\circ A_2$.




N

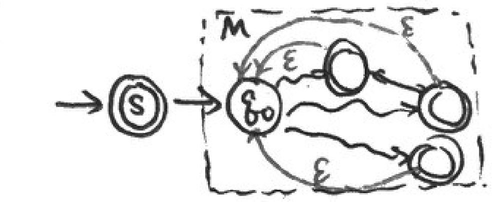
$$N=(Q_1\cup Q_2,\ \Sigma=\Sigma_1\cup\Sigma_2,\ \delta,\ q_{01},\ F_2)$$

$$\delta(q,a):=\begin{cases}\{\delta_1(q,a)\} & q\in Q_1-F_1 \text{ or } (q\in F_1 \text{ and } a\neq\varepsilon)\\ \{q_{02}\} & q\in F_1 \text{ and } a=\varepsilon\\ \{\delta_2(q,a)\} & q\in Q_2\end{cases}$$

由图示可见，这一台NFA给「切断」字符串提供了多种可能，而且我们根本不必计较可能性究竟有多少。显然 $\mathcal{L}(N)=A_1\circ A_2$. ■

**Theorem6** 正则语言类对*运算封闭。

proof




构造类似上图。细节留作练习。■

总而言之，我们已通过或简单或复杂的构造，证明了正则语言类对 $\cup,\cap,\circ,*$ 运算均封闭。这是一条很强的结论。至此，我们已对于DFA/NFA的能力边界有了初步认知——它是从正面来考虑的，即知道如何生成正则语

言。然而，在反方面，我们还缺乏认知。我们尚未清楚是否存在非正则语言。自然，这成为我们下一步的课题。

试考虑以下例子：

e.g. $A=\{s\in\{0,1\}^* \mid s$ 包含相同数目的0和1$\}$

$B=\{s\in\{0,1\}^* \mid s$ 包含相同数目的"01"子串和"10"子串$\}$

比如 $000111\in A$，$00101\notin A$，

$00010\in B$，$1010\notin B$.

试问A、B是否为正则语言？

我们先进行一些分析。如果让我们设计一台能识别A的DFA，自然的想法是做成「计数器」，每读到一个0则加一，每读到一个1则减一，终态若回到原点则接收。但这样的设计有一致命问题：我们不能预知输入串有多长。假如输入$0^{50}1^{50}$，则计数器需要50个状态，而输入$0^{1000}1^{1000}$，则计数器需要1000个状态。由于DFA的状态数有限且不能根据输入而动态调节，故总存在某输入使我们的DFA「内存不足」。（说远一点，若将A中的元素长度限制在常数$l$以内，则显然能依上述方法构造出正确的DFA）。

既然「计数器」思路不可行，那么是否有其它思路，把状态数压缩在有限个数以内呢？不妨抽象地论证一番：假设 $s_1=0^i1^i\in A$，$s_2=0^j1^j\in A$ $(i\neq j)$。那么当DFA读到$s_2$的第$j$个0时，与它读到$s_1$的第$i$个0时，必然处在不同的状态，否则会产生$s_1/s_2$中某个接纳不了的情形。不难推知，对$\forall n\in\mathbb{N}$，DFA的状态数$|Q|\geq n$，故状态数无限，这是无法做到的。

再来看B。观察得知：出现一次"01"以后，若想再度出现"01"，则必须先由1跳回0，即出现一次"10"。是故"01"与"10"必定交错出现。这一性质极大地方便了我们的

设计——两个状态足矣。不难构造出一台识别B的DFA。

A与B看似如此相似，但一个不是正则语言而另一个则相反，究其原因，关键是语言结构所含信息之多少。若信息能用有限状态来表达，则是正则的。

然而，上述讨论终究有些粗糙，不够精确。面对一门语言，我们是否有更系统的方法来论证其正则性？

**Lemma 7** (*Pumping Lemma*)...

若A是正则语言，则$\exists p>0$，$\forall s\in A$且$|s|\geq p$，存在一种分割把s切成三段$s=xyz$并满足：

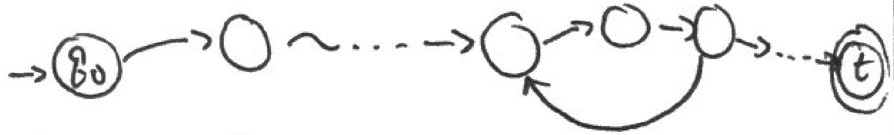
1° $|y|>0$

2° $|xy|\leq p$

3° $xy^iz\in A$，$\forall i\in\mathbb{N}$。

换句话说，对于足够长的字符串$s\in A$，总有办法把s中某个子串y取出来，重复若干次，再接回x—z里去，得到的串仍属于A.

看似复杂的定理，实质上无非是鸽笼原理。当一个串足够长，它被输入DFA之后必然会经历重复的状态，即走了一个环。把这个




环重复任意次，都不会改变被DFA接纳的实质。

proof 设DFA M能识别A。又记M的状态集为Q。

取$p=|Q|$。对$\forall s=s_1s_2\cdots s_l\in A$ $(l\geq p)$，我们知道它能被M接纳，设M经历的状态为

$$q_0\xrightarrow{s_1}q_1\xrightarrow{s_2}q_2\xrightarrow{s_3}\cdots\xrightarrow{s_l}q_l$$

$(q_l\in F)$

因为$l \geq p$，故这条计算过程所经历的状态数严格大于$p=|Q|$，亦即必存在$q_i=q_j$ $(i<j)$。不妨设$q_i, q_j$是第一对重复的状态。

$$q_0 \to q_1 \to \cdots \to \underbrace{q_i \xrightarrow{s_{i+1}} \cdots \xrightarrow{s_j} q_j}_{} \to \cdots \to q_l.$$

那么显然$j \leq p$。我们来研究$q_i \rightsquigarrow q_j$ ($q_j = q_i$)这一段过程．易知，当M处在状态$q_i$下时，连续输入$s_{i+1}, s_{i+2}, \cdots, s_j$会使M重又回到状态$q_i$，是故这一段字符重复若干次~~无论~~无非是使N原地踏步。

那么，令 $x = ~~q_0 q_1~~ s_1 s_2 \cdots s_i$

$y = s_{i+1} s_{i+2} \cdots s_j$

$z = s_{j+1} s_{j+2} \cdots s_l$

则必然有

1° $|y| \geq 1 > 0$

2° $|xy| = j \leq p$

3° $xy^kz$沿过程$q_0 \to q_1 \to \overbrace{q_i \to \cdots \to q_j}^{\text{若干次}} \to \cdots \to q_l$ 被M接纳 $\Rightarrow$ $xy^kz \in A$，$\forall k \in \mathbb{N}_0$ ■

依据该引理，我们能便捷地验证前面例子中A的非正则性。

e.g. $A = \{s \in \{0,1\}^* \mid s$中含相同数目的0和1$\}$不是~~非~~正则语言。若不然，则$\exists p>0$，使$s=0^p1^p \in A$能被「延长」．

记$s=xyz$，$|xy| \leq p$且$|y|>0$，故y只能~~...~~包含0，即$y=0^q$ $(0<q \leq |xy|)$．无论如何，$xy^5z$之中0与1的个数都无法相等，即$xy^5z \notin A$，与Lemma 7矛盾． ■

remark 仔细想来，上述论证说的是：设有某台DFA能识别A，则它必能识别$s=0^p1^p$，然而在前p个字符时（由于状态数的不足而）已形成循环，导致诸如$0^{p+i}1^p$之类的字符串也被自DFA接纳。从哲学层面上来说，Pumping Lemma揭示了「存储受限导致溢出，而溢出导致结果不可靠」这一道理。反

过来想，若存储能力相当强（虽则不及无限），那么使之溢出的输入便需相当长，故在更多的情况下我们能得到可靠的输出。扩大存储能力的理由之一便在于此。

> **problem** Pumping Lemma 是否可逆？若不可逆，那么是否有关于正则性更精准的刻画？

若我们已知 A 不是正则语言，而希望证明某语言 A' 也不是正则语言，还可以通过以下途径：

1° 假设 A' 是正则语言

2° 试图将 A 表达成 A' 与某一正则语言 A'' 的交/并/连接/复杂表达式。

3° 那么 A' 与 A'' 的运算结果 (A) 也必为正则语言 ⇒ 矛盾！

截至目前，我们可以说：我们已从正反两方面摸索了 DFA/NFA/正则语言的边界，也理解了什么语言是无法被该模型计算的。在下一章，我们将介绍更强大的模型，用以扩展 DFA/NFA 的能力范围。结束本章以前，我们介绍「正则表达式」的概念，并证明它与 DFA/NFA 之等价性。其目的可概括为：

1° 提供一套符号系统，方便我们描述正则语言。这在计算机科学的诸多领域很实用。

2° 给我们一种符号化 DFA/NFA 的可能，既直观地体现某台 DFA/NFA 的功能，又不必绘制状态转移图。

所谓「表达式」，即由一些元素和运算符组成的式子，它**唯一**地对应于某个结果。例如我们熟知的

$$5 + 2\times 3\times 6 + 2$$

和

$$22\times 3 - 23$$

都是代数表达式。两式的结果恰好都是 43，但却是不同的表达式。就**形式而言，它们不是同一个东西**；就

其对应的结果（含义）而言，它们则是等价的。

类似地，我们所要说的「正则表达式」，也是这么一些符号与运算的组合。每个正则表达式都对应了一个结果。

**def** 正则表达式。设 $\Sigma$ 是字符集。$R$ 是一个正则表达式当且仅当 $R$ 写作下面各情况之一：

1°. $a$ $(a \in \Sigma_\varepsilon)$

2°. $\emptyset$

3°. $R_1 \cup R_2$ （$R_1$、$R_2$ 均为正则表达式）

4°. $R_1 \circ R_2$ （$R_1$、$R_2$ 均为正则表达式）

5°. $R_1^*$ （$R_1$ 是正则表达式）

而 $R$ 所对应的结果（记为 $\mathcal{L}(R)$）按如下原则确定：对情况1°，$\mathcal{L}(R)=\{a\}$；对情况2°，$\mathcal{L}(R)=\emptyset$；对情况3°，$\mathcal{L}(R)=\mathcal{L}(R_1)\cup\mathcal{L}(R_2)$；对情况4°，$\mathcal{L}(R)=\mathcal{L}(R_1)\circ\mathcal{L}(R_2)$；对情况5°，$\mathcal{L}(R)=[\mathcal{L}(R_1)]^*$.

**remark** 不难看出，$\mathcal{L}(R)$ 是良定义的。另外，在不造成理解偏差的前提下，表达式中的“$\circ$”符号可省略。

e.g. 以下各式均为正则表达式（$\Sigma=\{a,b\}$）.

$abaa$

$a^*ba^*$

$b(a\cup b)^*(bab)^*\cup\varepsilon$

$a\Sigma b$ （注意：我们用“$\Sigma$”代指“$(a\cup b)$”）

就和“$5+2\times3\times6+2 = 22\times3-23$”一样，我们也可定义正则表达式的相等”。

**def** 正则表达式相等：设 $R_1$ 与 $R_2$ 为正则表达式，若 $\mathcal{L}(R_1)=\mathcal{L}(R_2)$，则称 $R_1$ 与 $R_2$ 相等，记为 $R_1=R_2$.

对于“相等”的理解应按前述：表达式相等是结果意义上的相等。

同一门语言（如 $\{aab, abb\}$）有多种正则表达式写法（如 $aab\cup abb$, $a\Sigma b$）。形式

不同，实质相同。这就好比一个人可以设计好几种DFA来识别同一种语言。

也许你已猜到：正则表达式的表达范围就是正则语言，即其与DFA(NFA)的能力相同。下面我们分两步证明。

**Lemma 8** 对任意正则表达式 $R$，总存在某台NFA $N$：$\mathcal{L}(R)=\mathcal{L}(N)$

proof. 构造是显然的。留作习题。■

**Lemma 9** 对任意DFA $M$，总存在正则表达式 $R$：$\mathcal{L}(M)=\mathcal{L}(R)$.

欲证此引理，自然希望把 $M$ 的图结构写成正则表达式。基本思路是把「链」对应为一系列「∘」的结合，把「分支」对应为「∪」，而把「环」对应为「*」。可以考虑先把 $q_0\to F$ 的所有简单路径找出来，再处理分支与环的问题。这样的证明是可行的，但过于繁琐，留作思考题。

另一种想法是「逐步地把状态吃掉，每次一个」。欲达成此种目标，必须将状态转移方式推广，允许转移条件写成正则表达式。教材《计算理论导引》中即采用这种证明思路。

最后一种思路格外简洁巧妙，它如图论中的Floyd-Warshall算法，逐步构建出所需表达式。下面我们介绍这一证明。

proof. ~~令 R(i,j,k) ::= 从状态~~ $(q_0=q_1)$.

设 $M$ 的状态集为 $Q=\{q_1,q_2,\cdots,q_n\}$.

令 $R(i,j,k)$ ::= 从状态 $q_i$ 出发，中间经过 $\{q_1,q_2,\cdots,q_k\}$ 之中的0个或多个状态，最终到达状态 $q_j$ 的所有可能输入之集合.

注意 $R(i,j,k)$ 有可能为 $\emptyset$.

我们有边界条件

$$R(i,j,0)=\{w\in\Sigma \mid \delta(q_i,w)=q_j\}$$

$\forall i,j$

以及方程

$$R(i,j,k) = \underbrace{R(i,j,k-1)}_{\text{不经}q_k\text{而直达}} \cup \underbrace{R(i,k,k-1)}_{\text{先从}i\text{到}k} \circ \underbrace{[R(k,k,k-1)]^*}_{\text{然后在}k\text{处打转若干次}} \circ \underbrace{R(k,j,k-1)}_{\text{然后从}k\text{到}j}.$$

或 [先从i到k 然后在k处打转若干次 然后从k到j]

中间可能会打转，但所有可能已被囊括在这两个集合中.

经过有限步递推，可算出全部 $R(i,j,k)$。

最终，$\mathcal{L}(M) = \bigcup_{q_t \in F} R(1,t,n)$

现在，我们不妨把表达式展开，直至式中仅包含边界 $R(i,j,0)$. 进一步，把边界拆分成若干单元集合之并。最终，我们得到一个仅含单元集合、∪、∘、* 的集合表达式，其结果 $= \mathcal{L}(M)$。这时，我们把所有单元集合替换为正则单元（比如 $\{a\}$ 换成 a），生成正则表达式R，且 $\mathcal{L}(R)$ 显然就等于上述集合表达式之结果，即 $\mathcal{L}(M)$. ∎

**Theorem 10** 语言A是正则语言当且仅当A可表成某正则表达式。

proof. 由Lemma 8, 9 立得。∎

所以，说来说去，DFA/NFA/正则表达式实际上是一个东西。对一个正则语言，我们既可以用DFA来实现，也可以用NFA，抑或正则表达式来实现。途径之差别不影响效果之殊途同归。

# APPENDIX

# Characterizations of Regular Languages

我们已说明了正则语言就是可被DFA/NFA识别的语言，也就是能被正则表达式生成的语言。这都是从正面给出的刻画。有了它们，我们容易证明一个正则语言的确是正则的，但却不好证明一个非正则语言不是正则的。（试想有人教我们去证明所有DFA都不能识别A，那我们很可能束手无策。）

所幸我们有 *Pumping Lemma*。利用之，我们常可通过反证法说明某语言A非正则。只不过，*Pumping Lemma* 是正则的必要不充分条件；换言之，存在一些非正则的语言同样满足之。例如

$$A := \{0^i1^j2^k \mid i,j,k\in\mathbb{N}_0\text{. 若 } i>0 \text{ 则要求 } j\le k\text{；若 } i=0 \text{ 则 } j,k \text{ 无限制}\}$$

$$= \{1^*2^*\} \cup \{00^*1^j2^k \mid j,k\in\mathbb{N}_0,\ j\le k\}$$

首先，A不是正则语言，因为 $A\cap\{00^*1^*2^*\} = \{00^*1^j2^k \mid j\le k\}$ 不是正则语言。

其次，A满足 *Pumping Lemma*。取常数 $p:=1$，那么，对任何 $s\in A$ 且 $|s|\ge p$，

1° 若 $s=1\cdots12\cdots2$，那么随便把 $s$ 切割成 $s = \underbrace{}_{u}\underbrace{1}_{v}\underbrace{\cdots12\cdots2}_{w}$，便有 $uv^iw\in A$ ($\forall i$)

2° 若 $s=0\cdots0\underbrace{1\cdots1}_{j}\underbrace{2\cdots2}_{\le k}$，则取 $u=\varepsilon$, $v=0$, $w=$ 余下，便仍有 $uv^iw\in A$ ($\forall i$).

可见，我们没法用反证法 + *Pumping Lemma* 证明A的非正则性。这引导我们去寻找一些关于正则语言的充要条件。本附录介绍两个相关成果。

def 「/」运算.

设A是一门语言，$w\in\Sigma^*$是一个字符串。

定义 $A/w := \{w'\in\Sigma^* \mid ww'\in A\}$.

remark. 换种说法，$A/w$ 就是把A中所有以$w$开头的串删掉开头所构成的语言。在某种意义下，可视/为。的逆运算。

def 由A诱导的等价关系 $\equiv_A$.

设A是一门语言。对于任何 $w_1, w_2\in\Sigma^*$，我们定义

$$w_1 \equiv_A w_2 \iff A/w_1 = A/w_2$$

不难验证关系「$\equiv_A$」具有自反性、对称性和传递性，故它是一个等价关系。

对于任何语言A，均可诱导等价关系$\equiv_A$。而$\equiv_A$将$\Sigma^*$划分为若干（可能无穷多）个等价类。比如语言$A=\{0^*1^*\}$便由$\equiv_A$关系将$\Sigma^*=\{0,1\}^*$划分为三个等价类：

| 第一类 | 第二类 | 第三类 |
|---|---|---|
| $\equiv_A 0$ | $\equiv_A 1$ | $\equiv_A 10$ |

这是由于 $\forall w\in\Sigma^*$，无非以下三种情形：

1° $w$形如 $0^i1^j$ $(i\geq 1)$，那么不难证明 $A/w = A = A/0$，故 $w\equiv_A 0$，属第一类。

2° $w$形如 $1^i$，那么 $A/w = A/1$，故 $w\equiv_A 1$，属第二类。

3° $w$不形如 $0^i1^j$。那么 $A/w=\emptyset=A/10$，故 $w\equiv_A 10$，属第三类。

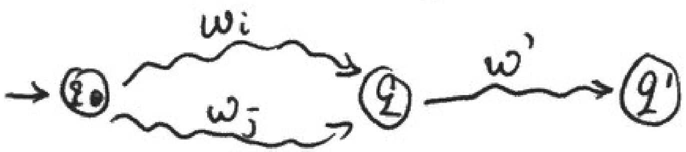
如此定义出的等价关系有何内涵呢？其实，若 $w_1\equiv_A w_2$，则意味着，以$w_1$开头的字符串 $S_1=w_1w_1'$ 与以$w_2$开头的字符串 $S_2=w_2w_2'$，在尾部是别无二致的。换成自动机的语言，便是：识别$w_1'$与$w_2'$可以共用同一套流程。我们下面证明的定理之核心即在于此。

# Myhill-Nerode Theorem

A是正则语言 $\iff$ $\equiv_A$ 将$\Sigma^*$划分为有限多个等价类。

proof. 1°(⇒) 若A是正则语言，则存在DFA $D=(Q,\Sigma,\delta,q_0,F)$。我们用反证法说明$\equiv_A$划分的等价类个数$\le|Q|$。若不然，则设其中$|Q|+1$个等价类为$[w_1],[w_2],\dots,[w_{|Q|+1}]$，这里的$w_1,\dots,w_{|Q|+1}$是每个类的代表元素。显然它们两两不等。

由于D的状态仅有$|Q|$个，所以，必定有2个字符串$w_i$与$w_j$在输入进D以后会「相撞」，来到状态$q$。这么一来，$\forall w'\in\Sigma^*$，$w_iw'$与$w_jw'$总是使得D进入相同的状态，故$A/w_i=A/w_j \Rightarrow [w_i]=[w_j]$，矛盾。




2°(⇐) 若$\equiv_A$将$\Sigma^*$划分为有限多个等价类，记之为$[w_1],[w_2],\dots,[w_k]$。

我们直接据此构造DFA $D$: $\mathcal{L}(D)=A$.

$D=(Q,\Sigma,\delta,q_0,F)$.

$Q:=\{1,2,3,\dots,k\}$

$q_0:=\varepsilon$所在类的下标

$F:=\{i\mid \exists s\in[w_i]: s\in A\}$

$\delta(i,a):=(w_ia)$所在类的下标.

可用归纳法证明：对任何$w\in\Sigma^*$，将其输入D后，终结状态必为$w$所在类的下标。又因为同一类的字符串要么均在A中，要么均不在A中（可用反证法证明），且F已正确地指派了回答，故$\mathcal{L}(D)=A$，从而A是正则的。■

hint. 归纳中用到如下的简单性质：

$$(A/w)/y=A/(wy).$$

但该定理用起来稍有些臃肿。有没有一个长得像Pumping Lemma的、「直观的」充要条件呢？答案是肯定的。

## Jaffe's Pumping Lemma

$A$ 是正则语言 当且仅当：

$\exists p>0\ \forall s\in\Sigma^p$，$\exists$一种 $s$ 的分割 $s=uvw\ (v\neq\varepsilon)$ 满足 $\forall s'\in\Sigma^*\ \forall i\in\mathbb{N}_0$

$$ss'\in A \leftrightarrow uv^iws'\in A.$$

proof. 1°($\Rightarrow$) 同 Pumping Lemma，略.

2°($\Leftarrow$) 我们来证明满足条件的 $A$ 所诱导的 $\equiv_A$ 关系至多将 $\Sigma^*$ 划分为 $1+|\Sigma|+|\Sigma|^2+\cdots+|\Sigma|^{p-1}$ 个等价类，从而由 Myhill-Nerode Theorem 立即可得 $A$ 是正则语言.

设长度小于 $p$ 的所有字符串为

$$s_1, s_2, \ldots, s_m \quad (m=1+|\Sigma|+|\Sigma|^2+\cdots+|\Sigma|^{p-1})$$

它们所属的等价类分别为

$$[s_1], [s_2], \ldots, [s_m] \quad (\text{可能不互异})$$

现在，考察任意字符串 $xt$: $|x|\geq p$。总能把 $xt$ 写成 $xt=\underbrace{uvw}_{长为p}yt$，且使得 $\forall z\in\Sigma^*\ \forall i\in\mathbb{N}_0$ 有 $uvwz\in A \leftrightarrow uv^iwz\in A$.

作为特殊情形，令 $z:=yt$ 且 $i:=0$，则

$$\underbrace{uvwyt}_{=xt}\in A \leftrightarrow \underbrace{uwyt}_{=x't}\in A$$

类似地，对 $x'$ 重复，便有

$$x't\in A \leftrightarrow x''t\in A \quad (|x''|<|x'|)$$

等等。有限步内(其实是归纳法)必有

$$xt\in A \leftrightarrow \hat{x}t\in A \quad (|\hat{x}|<p)$$

下面，我们指出 $A/x=A/\hat{x}$。这是因为

$$\begin{aligned} A/x &= \{t\in\Sigma^* \mid xt\in A\} \\ &= \{t\in\Sigma^* \mid \hat{x}t\in A\} \\ &= A/\hat{x} \end{aligned}$$

故 $x\equiv_A\hat{x}$ 对任何 $x\in\Sigma^*$ $(|x|\geq p)$ 均成立，其中 $\hat{x}$ 是某个长度小于 $p$ 的字符串。换言之，$[s_1], [s_2], \ldots, [s_m]$ 事实上覆盖了全空间，证毕。■