第2章　消息传递系统中的基本算法

本章我们介绍第一个分布式计算模型，用于无故障的消息传递系统。我们考虑两种主要的时序模型：同步的和异步的。除了系统的形式化描述之外，我们还定义了主要的复杂度度量——消息数与时间——同时介绍将在描述算法时使用的伪代码惯例。

随后，我们介绍消息传递系统的几种简单算法，这里的消息传递系统具有任意的拓扑结构，可以是同步的，也可以是异步的。这些算法传播信息、收集信息、构造网络生成树。其主要目的是为了便于建立形式化和复杂度度量，引入算法的正确性证明。其中一些算法稍后将被用做本书中其他更复杂算法的构件。

2.1　消息传递系统的形式化模型

本节首先介绍无故障的同步和异步消息传递系统的形式化模型，然后定义基本的复杂度度量，最后介绍用于描述消息传递算法的伪代码惯例。

2.1.1　系统

在一个消息传递系统中，各处理器通过在通信信道上发送消息来进行通信，每个信道在两个特定的处理器之间提供一个双向连接。由各信道确定的连接模式描述了系统的拓扑结构。拓扑结构可以用一个无向图来表示，其中节点代表处理器，当且仅当两个处理器之间有信道时，对应的节点之间有边。我们将专门处理各种连接拓扑。信道的集合常被称为网络。具有特定拓扑结构的消息传递系统的算法，由系统中各处理器上的局部程序所组成。处理器的局部程序为处理器提供执行局部计算的能力，提供向给定拓扑中的每个邻居发送消息，以及从它们那里接收消息的能力。

形式化描述如下，一个系统或算法由 n 个处理器 $p_0,\cdots,p_{n-1}$ 组成，i 是处理器 p_i 的索引。每个处理器 p_i 的模型可以用具有状态集 Q_i 的一个状态机（可能是无限的状态）来描述。处理器用拓扑图中的一个特定节点来标识。拓扑图中与 p_i 关联的边可用整数 $1\sim r$ 标记，其中 r 是 p_i 的度（见图2.1中的例子）。处理器 p_i 的每个状态包含 $2r$ 个特别成员：$outbuf_i[\ell]$ 和 $inbuf_i[\ell]$，$\ell=1,2,\cdots,r$。这些特别成员是消息的集合：$outbuf_i[\ell]$ 中保存的是已由 p_i 通过第 ℓ 条关联信道发送给邻居，但还没有提交给邻居的消息；而 $inbuf_i[\ell]$ 中保存的是已从第 ℓ 条关联信道提交给 p_i，但 p_i 还没有在内部计算步骤中进行处理的消息。各状态集 Q_i 包含不同的初始状态子集；在初始状态，每个 $inbuf_i[\ell]$ 必定是空的，但 $outbuf_i[\ell]$ 不必为空。

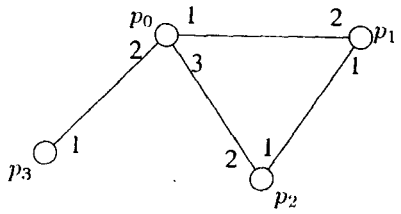

图2.1　一个简单的拓扑图

除 $outbuf_i[\ell]$ 成员外，处理器 p_i 的状态组成它的可存取状态。处理器 p_i 的转移函数将 p_i 的可存取状态作为输入值，产生 p_i 的可存取状态（其中每个 $inbuf_i[\ell]$ 为空）作为输出值。对 1 到 r 之间的每个 ℓ，转移函数还最多产生一条发送给 p_i 的第 ℓ 条关联信道另一端邻居的消息作为输出。这样，先前 p_i 已发送但尚在等待提交的消息，就不会影响 p_i 的当前步骤；每个步骤处理所有等待提交给 p_i 的消息，导致状态的改变，且给每个邻居最多发送一条消息。

配置（configuration）是一个向量 $C=(q_0,\cdots,q_{n-1})$，其中 q_i 是 p_i 的一种状态。配置中 outbuf 变量的状态，表示在通信信道上传送的消息。一个初始配置是一个向量 $(q_0,\cdots,q_{n-1})$，其中 q_i 是 p_i 的一种初始状态；简而言之，就是各处理器都处于初始状态。

系统中发生的事情用事件来模拟。对于消息传递系统，我们考虑两种事件：一种是计算事件（computation event），表示为 $comp(i)$，代表处理器 p_i 的计算步骤，该步骤将 p_i 的转移函数应用到它当前的可存取状态；另一种是提交事件（delivery event），表示为 $del(i,j,m)$，代表消息 m 从处理器 p_i 提交到 p_j。

系统随时间发生的行为用执行来模拟，执行是随事件而改变的配置序列。根据所模拟系统的具体类型，这一序列必须满足多种条件。我们把这些条件分为安全性和活跃性两类。安全性条件（safety condition）是序列的每一有限前缀所必须满足的条件。例如，“处理器 p_i 的每个步骤立即跟随处理器 p_0 的一个步骤”。通俗而言，安全性条件陈述的是还没有发生不好的事情。比如，刚刚给出的例子可重新叙述为：要求 p_1 的步骤不得立即跟随除 p_0 之外任何处理器的步骤。活跃性条件（liveness condition）是必须满足特定次数（可能是无限次）的条件。例如，条件“最终 p_1 结束”要求 p_1 的结束发生一次；条件“p_1 进行无限步数”要求无限地产生“p_1 只进行一个步骤”的条件。通俗而言，活跃性条件陈述的是最终发生某些好的事情。对于一个特定的系统类型，任何一个满足全部所要求的安全性条件的序列，称为一个执行（execution）。如果某一执行也满足全部所要求的活跃性条件，则称该执行是合法的（admissible）。

我们现在针对异步和同步两种类型的消息传递系统，定义执行和合法执行所要求的条件。

2.1.1.1　异步系统

如果一个系统提交消息的时间上界不固定，或某个处理器两个连续步骤间的时间间隔不固定，那么就称该系统是异步的。异步系统的一个例子是 Internet，它的消息（如电子邮件）虽然常常只要数秒的时间就能到达，但也可能要花几天时间才能到达。关于消息延迟和处理器步数，通常存在上界，但有时这些上界非常大，极少情况下能够达到，而且可能随时间而改变。因此，一般不依赖于这些界限设计算法，即通常是设计不依赖于任何特定定时参数的算法，也就是异步算法。

异步消息传递系统的一个执行段（execution segment）α（有限或无限）是一个如下形式的序列：

$$C_0,\phi_1,C_1,\phi_2,C_2,\phi_3,\cdots$$

其中每个 C_k 是一个配置，每个 ϕ_k 是一个事件。如果 α 是有限的，那么它一定以某个配置结束。此外，必须满足下列条件：

- 如果 $\phi_k=del(i,j,m)$，那么 m 必定是 C_{k-1} 中 $outbuf_i[\ell]$ 的一个元素，其中 ℓ 是信

道$\{p_i, p_j\}$中 p_i 的标号。从 C_{k-1}到 C_k 的唯一改变是:在 C_k 中,m 已从 $outbuf_i[\ell]$ 中移走并加入到 $inbuf_j[h]$中,其中 h 是信道$\{p_i, p_j\}$中 p_j 的标号。简而言之,只要消息传送了就提交完成了,其唯一的变化是从发送者的输出缓冲区移到接收者的输入缓冲区(在图 2.1 的例子中,从 p_3 到 p_0 的消息放在 $outbuf_3[1]$中,然后提交到 $inbuf_0[2]$)。

- 如果$\phi_k = comp(i)$,那么从 C_{k-1}到 C_k 的唯一改变是:p_i 依照其转移函数来改变状态,转移函数对 C_{k-1}中 p_i 的可存取状态进行操作,并将所确定的消息集合加入到 C_k 的 outbuf_i 变量中,这些消息在这一事件上称为被发送的。简而言之,p_i 依照其转移函数(本地程序)来改变状态及发送消息,转移函数基于当前状态,包括所有待提交的消息(但不是待输出的消息)。回顾一下,处理器的转移函数要保证 inbuf 变量清空。

一个执行(execution)是一个执行段 $C_0, \phi_1, C_1, \phi_2, C_2, \phi_3, \cdots$,其中 C_0 是初始配置。

对于每个执行(或每个执行段),我们关联一个调度(schedule)或一个调度段,它是执行中的事件序列,即$\phi_1, \phi_2, \phi_3, \cdots$。并非每个事件序列对于所有初始配置都是一个调度;比如,$del(1,2,m)$对于 outbuf 为空的初始配置就不是一个调度,因为不存在 p_1 的前一步骤来发送 m。注意,如果局部程序是确定性的,那么执行(或执行段)就由初始(或起始)配置 C_0 和调度(或调度段)σ 唯一确定,表示为 $exec(C_0, \sigma)$。

在异步模型中,如果每个处理器都有无限的计算事件数,且每条发送的消息最终都被提交,那么该执行是合法的。要求有无限的计算事件数,是模拟处理器不发生故障的事实。这不表示处理器的局部程序必须包含一个无限循环;一个算法终止的非形式化描述,是在处理器已完成任务的某一特定点之后,转移函数已不再改变处理器的状态。换句话说,处理器在该点之后执行"哑步骤"。如果一个调度是合法的执行的调度,那么该调度就是合法的。

2.1.1.2 同步系统

在同步模型中,各处理器执行的步伐一致:执行按轮(round)划分,在每一轮中各处理器可以发送消息给每个邻居,消息被提交;每个处理器基于刚刚接收到的消息进行计算。这种模型虽然在实际的分布式系统中难以获得,但它非常便于设计算法,因为算法无须对付很多的不确定性。我们今后将会看到,一旦为这种理想的时序模型设计了一个算法,那它就可以在其他更现实的时序模型中自动模拟运行。

同步情况下,执行的形式化定义,是对异步情况下的定义增加约束。配置和事件的交替序列可以划分为分离的各轮。每一轮的组成如下:outbuf 变量中每个消息的提交事件,直到所有的 outbuf 变量为空,接下来是每个处理器的计算事件。这样,一轮包括提交所有待处理的消息,然后使每个处理器执行内部计算步骤来处理所有被提交的消息。

对于同步模型,如果一个执行是无限的,那它就是合法的。考虑轮的结构,这意味着每个处理器执行无限的计算步数,且每条发送的消息最终都被提交。如同异步情况一样,假定合法的执行是无限的。这是为了技术讨论的便利,算法终止可按在异步情况中一样进行处理。

注意,在无故障的同步系统中,一旦算法固定,那么唯一可以改变的并与执行相关的方面是初始配置。在异步系统中,同一算法可有许多不同的执行,即便是具有相同的初始配置和无故障,因为处理器各步骤的交错和消息的延迟不是固定的。

2.1.2 复杂度度量

我们将关注分布式算法所需要的两种复杂度度量:消息数和时间量。首先,我们将集中

研究最坏情况下的性能;稍后在书中,我们有时也会关心预期情况下的性能。

为了定义这些度量,需要算法终止的概念。我们假设每个处理器的状态集包括一个终止状态子集,并且每个处理器的转移函数将终止状态仅映射成终止状态。当所有处理器处于终止状态且没有消息在传送中时,我们就称系统(算法)已终止。注意,合法的执行仍然是无限的,但是一旦某个处理器进入一个终止状态,它就停留在该状态下执行"哑"步骤。

一个同步或异步的消息传递系统,其算法的消息复杂度(message complexity),是在算法所有合法的执行中最大的发送消息总数。

同步系统中度量时间的简单方法,是计算终止前的轮数。因此,同步消息传递系统中算法的时间复杂度(time complexity),是在算法所有合法的执行中最大的终止前轮数。

在异步系统中度量时间不太直接。通常的一种方法,也是我们将使用的,是假定任一执行中的最大消息延迟是一个时间单位,然后计算终止前的运行时间。为使这一方法更精确,我们必须在执行中引入时间的概念。

计时(timed)执行是指执行中每个事件关联一个非负的实数,即事件发生的时间。时间必须从 0 开始,必须是非递减的,对每个独立的处理器[①]必须是严格递增的,并且如果执行是无限的,则递增没有限制。这样,执行中的事件可以根据发生的时间进行排序,不在同一处理器上的多个事件可以同时发生,在任一有限时间前只能发生有限数目的事件。

我们定义消息的延迟(delay),是指在发送消息的计算事件和处理消息的计算事件之间流逝的时间。换句话说,它包括消息在发送者的 outbuf 中等待的时间量,加上消息在接收者的 inbuf 中等待的时间量。

异步算法的时间复杂度,是所有计时的、合法的执行中,当各消息延迟最多为 1 时,在终止前的最大时间。这一度量仍然允许事件的任意交替,因为没有对事件发生的时间强加下界。这可以看做是考虑了算法的任意执行,且进行了标准化,使最长的消息延迟变成一个单位的时间。

2.1.3　伪代码惯例

在前面介绍的形式化模型中,用状态转移对算法进行描述。然而,我们很少这么做,因为状态转移更难以让人理解,尤其是控制流的编码在许多情况下要采用相当不自然的方式。

我们将用两种不同的详细程度来描述算法。简单的算法将用文字来描述。更深入的算法将用伪代码来表示。下面描述将在同步和异步消息传递算法中使用的伪代码惯例。

异步算法对每个处理器采用中断驱动的方式进行描述。在形式化模型中,每个计算事件即刻处理等待在处理器的 inbuf 变量中的所有消息。为清楚起见,我们通常分别描述每个消息的效果。这等效于处理器按任意顺序一个接一个地处理等待的消息;在这一过程中,如果产生超过一条以上的消息发向同一接收者,则这些消息可以捆绑起来成为一个大消息。即使没有接收消息,处理器也可能采取某个动作。这里不列出那些导致没有消息发送及没有状态改变的事件。

计算事件中的局部计算,将按与顺序算法典型伪代码一致的方式进行描述。我们使用保留字"terminate"来指出处理器进入终止状态。

---

① $comp(i)$在 p_i 上发生,$del(i,j,m)$在 p_i 和 p_j 上发生。

异步算法也可以在同步系统中运行,因为同步系统是异步系统的一个特例。然而,我们将经常考虑为同步系统而专门设计的算法。这些同步算法对每个处理器按一轮接一轮的方式进行描述。对每一轮,我们将确定处理器发送什么消息,以及对刚接收的消息采取什么动作(注意在第一轮中发送的消息是最初在 outbuf 变量中的那些消息)。在一轮中完成的局部计算,将用与顺序算法典型伪代码一致的方式进行描述。终止是当不再有轮被确定时隐含指出的。

在伪代码中,处理器 p_i 的局部状态变量将不用 i 来作下标;在讨论和证明中,当有必要避免二义性时会加下标。

注解将以"//"开始。

在下一节,我们将给出一些用文字、伪代码来描述算法的例子,和用状态转移是一样的。

2.2 生成树上的广播和敛播

我们现在介绍几个例子,帮助读者来更好地理解分布式算法的模型、伪代码、正确性论据及复杂度度量。这些算法解决信息的收集与分散、计算基础通信网络的生成树等基本任务。它们将作为重要构件用到许多其他算法中。

2.2.1 广播

我们从(单条消息)广播问题的一个简单算法开始,假定网络的生成树已经给出。一个特定的处理器 p_r,具有某一信息,即消息〈M〉,它希望发送给所有的其他处理器。该消息的复制沿着以 p_r 为根的树发送,并跨越网络中的所有处理器。以 p_r 为根的生成树采用分布的方式来维护,即每个处理器有一条特定的信道连到它在树中的父节点,同时有一组信道连到它在树中的各子节点。

这里给出该算法的文字描述。图 2.2 演示了算法的一个异步执行示例,其中实线描绘了生成树中的信道,虚线描绘了不在生成树中的信道,而带阴影的节点表示已接收到〈M〉的处理器。根 p_r 向所有连到其子节点的信道发送消息〈M〉(见图 2.2(a))。当某个处理器从信道上接收到来自其父节点的消息〈M〉时,将向连到其各子节点的信道发送〈M〉(见图 2.2(b))。

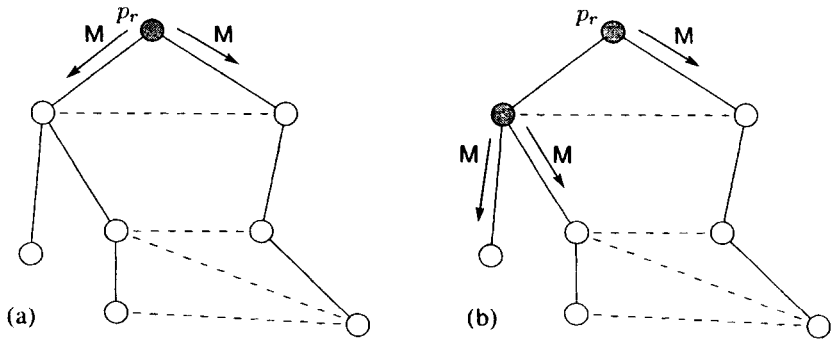

图 2.2 广播算法执行中的两个步骤

这个算法的伪代码见算法 1;对于没有接收消息和没有状态改变的计算步骤,没有伪代码。

最后，我们在状态转移这一级来描述算法。每个处理器 p_i 的状态包含：

- 变量 $parent_i$，保存一个处理器索引或为空；
- 变量 $children_i$，保存一组处理器索引；
- 布尔量 $terminated_i$，指示 p_i 是否处在终止状态。

变量 *parent* 和 *children* 的初始值使它们形成一棵以 p_r 为根的生成树。所有的 *terminated* 变量的初始值为假。初始时，$outbuf_r[j]$保存 $children_r$ 中对应每个 j 的〈M〉[①]；所有的其他 *outbuf* 变量为空。$comp(i)$的结果如下：如果对某个 k，〈M〉在 $inbuf_i[k]$中，那么对 $children_i$ 中的每个 j，〈M〉被放到 $outbuf_i[j]$中，p_i 通过设置 $terminated_i$ 为真而进入终止状态。如果 $i = r$ 且 $terminated_r$ 为假，那么 $terminated_r$ 被设置为 *true*。否则什么也不做。

注意，不管系统是同步的，还是异步的，这个算法都是正确的。此外，正如我们下面讨论的一样，该算法的消息与时间复杂度在两种模型中是相同的。

该算法的消息复杂度是多少？显然，消息〈M〉在属于生成树（从父节点到子节点）的每个信道上只发送一次，在同步和异步情况下都一样。也就是说，算法中发送的消息总数，正好是以 p_r 为根的生成树的边数。回顾一下，n 个节点的生成树正好有 $n-1$ 条边；因此，在算法中正好发送了 $n-1$ 条消息。

现在我们分析算法的时间复杂度。当通信是同步的且时间按轮度量时，分析起来比较容易。

下面的引理表明，在第 t 轮结束时，消息〈M〉到达了生成树中距离 p_r 为 t（或小于 t）的所有处理器。这是一个简单的断言，其证明很简单，我们在此详细介绍，以帮助读者熟悉分布式算法的模型和证明。稍后在书中我们将这类简单的证明留给读者。

算法 1　生成树广播算法

Initially ⟨M⟩ is in transit from p_r to all its children in the spanning tree.

Code for p_r:
1: upon receiving no message:　　// first computation event by p_r
2: 　terminate

Code for p_i, $0 \le i \le n-1, i \ne r$:
3: upon receiving ⟨M⟩ from parent:
4: 　send ⟨M⟩ to all children
5: 　terminate

引理 2.1　对同步模型中广播算法的每一次合法执行，生成树中距离 p_r 为 t 的每个处理器在第 t 轮接收到消息〈M〉。

证明　通过对处理器与 p_r 的距离 t 进行归纳来证明。

基础步骤：$t = 1$，从算法的描述可知，p_r 的每个子节点在第 1 轮从 p_r 接收到〈M〉。

现在假定生成树中距离 p_r 为 $t-1 \geqslant 1$ 的每个处理器，在第 $t-1$ 轮接收到消息〈M〉。

① 这里变量 *inbuf* 和 *outbuf* 的索引采用邻居索引，而不用信道标号。

我们必须证明在生成树中距离 p_r 为 t 的每个处理器 p_i 在第 t 轮接收〈M〉。设 p_j 是生成树中 p_i 的父节点。因为 p_j 距离 p_r 为 $t-1$,所以根据归纳假设,p_j 在第 $t-1$ 轮接收消息〈M〉。根据算法的描述,p_j 在下一轮发送消息〈M〉给 p_i。 □

根据引理 2.1,算法的时间复杂度是 d,其中 d 是生成树的深度。回顾一下,当生成树是一条链时,d 最多为 $n-1$。

因此我们有:

定理 2.2 *当一棵有根的生成树,其深度 d 已知时,存在一个同步的广播算法,其消息复杂度为 $n-1$,时间复杂度为 d。*

当通信是异步时,可以应用类似的分析。再次说明,其关键是证明到时间 t 时,消息〈M〉到达生成树中距离 p_r 是 t(或小于 t)的所有处理器。这意味着,当通信是异步时,算法的时间复杂度也是 d。下面将更细致地分析这一情形。

定理 2.3 *对异步系统中广播算法的每一次合法执行,生成树中距离 p_r 为 t 的每个处理器在时间 t 接收到消息〈M〉。*

证明 通过对处理器与 p_r 的距离 t 进行归纳来证明。

基础步骤:$t=1$,从算法的描述可知,〈M〉最初是传送给距离 p_r 为 1 的各处理器 p_i。根据异步模型的时间复杂度的定义,p_i 在时间 1 时接收〈M〉。

我们必须证明,在生成树中距离 p_r 为 t 的每个处理器 p_i 在时间 t 接收〈M〉。设 p_j 是生成树中 p_i 的父节点。因为 p_j 距离 p_r 为 $t-1$,故根据归纳假设,p_j 在时间 $t-1$ 接收〈M〉。根据算法的描述,p_j 在接收〈M〉时,即在时间 $t-1$,同时发送〈M〉给 p_i。根据异步模型的时间复杂度的定义,p_i 在时间 t 接收〈M〉。 □

因此我们有:

定理 2.4 *当一棵有根的生成树,其深度 d 已知时,存在一个异步的广播算法,其消息复杂度为 $n-1$,时间复杂度为 d。*

2.2.2 敛播

广播问题需要单向通信,从根 p_r 到树中的所有节点。现在考虑互补问题,称为敛播(convergecast),是将信息从树中各节点收集至根节点。为简单起见,我们考虑问题的一个特定变种,其中每个处理器 p_i 从数值 x_i 开始,我们希望将这些值中的最大值转发给根 p_r。(练习 2.3 涉及一个通用的敛播算法,收集网络中的所有信息。)

像广播问题中的一样,我们再次假设生成树是以分布方式来维护的。广播算法是从根开始的,而敛播算法是从叶节点开始的。注意,生成树的叶节点可以很容易地辨别,因为它没有子节点。

从概念上看,算法是递归的,且需要每个处理器计算以其为根的子树中的最大值。从叶节点开始,每个处理器 p_i 计算以其为根的子树中的最大值,表示为 v_i,并将 v_i 发送给其父节点。父节点从它的所有子节点中收集这些值,计算其子树中的最大值,并发送最大值给它的父节点。

算法更详细的描述如下所述。如果节点 p_i 是叶节点,那么它将其值 x_i 发送给它的父节

点而启动算法(见图 2.3(a))。一个非叶节点 p_j,有 k 个子节点,等待从其子节点 $p_{i_1},\cdots,p_{i_k}$ 接受包含 $v_{i_1},\cdots,v_{i_k}$ 的消息。然后计算 $v_j=\max\{x_j,v_{i_1},\cdots,v_{i_k}\}$,并将 v_j 发送给它的父节点(见到图 2.3(b))。

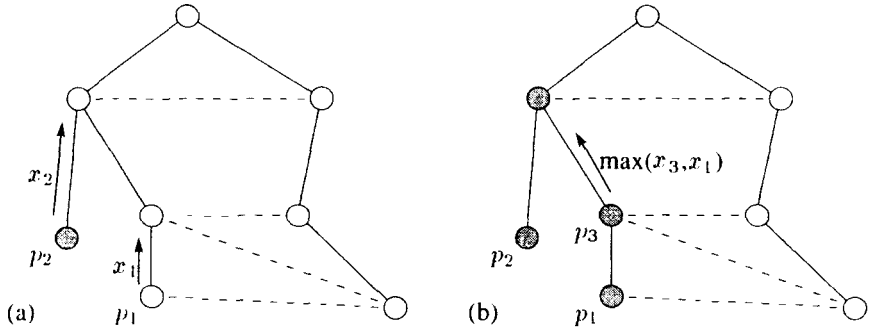

图 2.3　敛播算法执行中的两个步骤

敛播算法的消息和时间复杂度的分析非常类似于广播算法(练习 2.2 指出了如何分析敛播算法的时间复杂度)。

定理 2.5　一棵有根的生成树,当其深度 d 已知时,存在一个异步的敛播算法,其消息复杂度为 $n-1$,时间复杂度为 d。

有时将广播算法和敛播算法结合起来很有用。举例来说,根节点为获取某信息,通过广播发布一个请求,然后各响应再通过敛播汇集到根节点。

2.3　洪泛算法及构造生成树

2.2 节中介绍的广播和敛播算法,都是假定存在通信网络的、以特定处理器为根的生成树。现在我们来考虑更为复杂的广播问题,该广播在没有预先存在生成树的前提下,从某个特定的处理器 p_r 开始。首先考虑异步系统。

这种算法称为洪泛(flooding)算法,从 p_r 开始,发送消息〈M〉给它的所有邻居,即向它的所有通信信道发送。当处理器 p_i 从某个邻接的处理器 p_j 首次接收到〈M〉时,它将〈M〉发送给除 p_j 外的所有邻居(见图 2.4)。

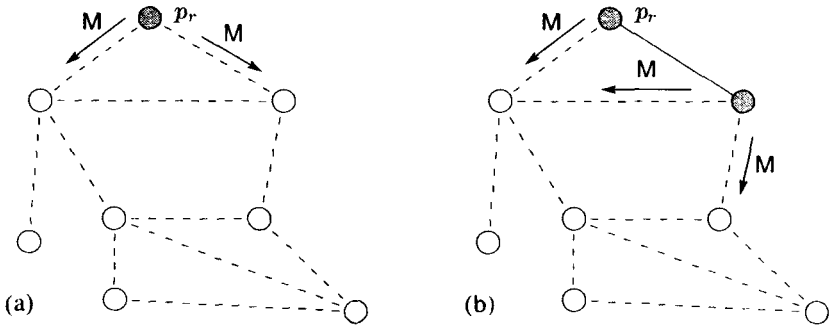

图 2.4　洪泛算法执行的两个步骤;实线指出执行中某点上生成树中的信道

显然,一个处理器在任一通信信道上发送〈M〉不会超过一次。这样在每个通信信道上,〈M〉最多发送两次(由使用此信道的每个处理器发送一次);注意,在某些执行中,消息〈M〉除首次接收外,在所有通信信道上发送两次(见练习 2.6)。因此,可能发送的消息数为 $2m-(n-1)$,其中 m 是系统中的通信信道数,最大可以达到 $\frac{n(n-1)}{2}$。

我们下面讨论洪泛算法的时间复杂度。

实际上,洪泛算法引起一棵生成树,根在 p_r、处理器 p_i 的父节点是 p_i 首次接收〈M〉的发送者。p_i 有可能并发地从多个处理器接收到〈M〉,因为从处理器上最后一个 *comp* 事件后,*comp* 事件将处理所有已提交的消息;在这种情况下,p_i 的父节点可从中任选一个。

可以修改洪泛算法,使之明确构造生成树。首先,p_r 发送〈M〉给它的所有邻居。如前所述,处理器 p_i 可能从多个处理器首次接收到〈M〉。当这种情况发生时,p_i 从发送〈M〉给它的邻接处理器中挑选一个,比如 p_j,指示它作为父节点,并发送一条〈parent〉消息给它。对所有的其他处理器,以及对以后发送〈M〉给它的任何其他处理器,p_i 发送一条〈already〉消息,指出 p_i 已经在树中。在 p_i 发送〈M〉给所有其他的邻居(p_i 前面未从它们那里接收过〈M〉)后,就等待从它们那里来的响应,或是〈parent〉消息,或是〈already〉消息。那些用〈parent〉消息响应的就指示为 p_i 的子节点。一旦所有接收到 p_i 的〈M〉消息的接收者都已用〈parent〉或〈already〉进行响应,则 p_i 终止(见图 2.5)。

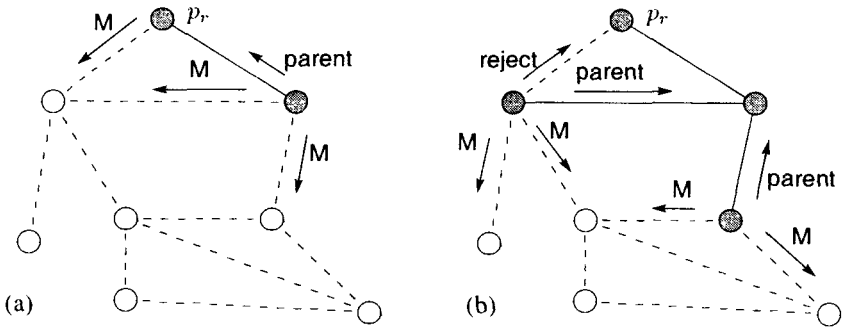

图 2.5 生成树构造中的两个步骤

修改后的洪泛算法的伪代码见算法 2。

引理 2.6 对异步模型中的每一次合法执行,算法 2 构造一个根在 p_r 的网络生成树。

证明 通过检查代码,揭示出关于算法的两个重要事实。首先,一旦某个处理器设置了它的 parent 变量,就不会再改变(而且它只有一个父节点)。其次,某个处理器的子节点集不会减少。这样,最后由 parent 和 children 变量形成的图结构是静态的,而且在不同节点上的 parent 变量和 children 变量是一致的。也就是说,如果 p_j 是 p_i 的子节点,那么 p_i 是 p_j 的父节点。我们可以证明结果图(称之为 G)是一棵根在 p_r 的有向生成树。

为什么从根可以到达每个节点?用矛盾法假设 G 中某一节点不能从 p_r 到达。由于网络是连接的,所以存在两个处理器 p_i 和 p_j,它们之间有一信道,在 G 中可从 p_r 到达 p_j,但不能到达 p_i。练习 2.4 要求验证:在 G 中某处理器可从 p_r 到达的条件为当且仅当它设置过 parent 变量。这样 p_i 的 parent 变量在整个执行中保持为 *nil*,而 p_j 在某个点上设置了它的 parent 变

量。因而在算法第 9 行，p_j 发送〈M〉给 p_i。由于执行是合法的，消息最终由 p_i 接收，所以导致 p_i 设置其 parent 变量。从而产生矛盾。

算法 2　修改后的洪泛算法构造生成树：处理器 p_i 的代码，$0 \leqslant i \leqslant n-1$。

```
Initially parent = ⊥, children = ∅, and other = ∅.

1:  upon receiving no message:
2:      if p_i = p_r and parent = ⊥ then          // root has not yet sent ⟨M⟩
3:          send ⟨M⟩ to all neighbors
4:          parent := p_i

5:  upon receiving ⟨M⟩ from neighbor p_j:
6:      if parent = ⊥ then                        // p_i has not received ⟨M⟩ before
7:          parent := p_j
8:          send ⟨parent⟩ to p_j
9:          send ⟨M⟩ to all neighbors except p_j
10:     else send ⟨already⟩ to p_j

11: upon receiving ⟨parent⟩ from neighbor p_j:
12:     add p_j to children
13:     if children ∪ other contains all neighbors except parent then
14:         terminate

15: upon receiving ⟨already⟩ from neighbor p_j:
16:     add p_j to other
17:     if children ∪ other contains all neighbors except parent then
18:         terminate
```

为什么没有环呢？用矛盾法假设存在一个环，比如 $p_{i_1}, p_{i_2}, \cdots, p_{i_k}, p_{i_1}$。注意，如果 p_i 是 p_j 的子节点，那么 p_i 在 p_j 之后首次接收到〈M〉。由于每个处理器都是环中下一个处理器的父节点，这就表示 p_{i_1} 在(它自己)之前首次接收到〈M〉，从而矛盾。 □

显然，修改后的洪泛算法构造生成树，其时间复杂度只是增加了一个常数因子。

在异步通信模型中，很容易看到在时间 t，消息〈M〉到达距离 p_r 为 t(或小于 t)的所有处理器。因此：

定理 2.7　*给定一个特定节点，存在可找到有 m 条边且直径为 D 的网络生成树的异步算法，其消息复杂度为 $O(m)$、时间复杂度为 $O(D)$。*

修改后的洪泛算法无须改变就能在同步情况下工作，它的分析类似于异步情况。然而，在同步情况下，与异步不同的是，所构造的生成树应保证是广度 - 优先搜索(breadth-first search，BFS)树：

引理 2.8　*对同步模型中的每一次合法执行，算法 2 构造一棵根在 p_r 的 BFS 树。*

证明　使用关于 t 的归档法来证明，从第 t 轮开始时，(1)根据当前 parent 变量所构造的图是一棵 BFS 树，它包括距离 p_r 最多为 $t-1$ 的所有节点，而且(2)消息〈M〉只从距离 p_r 正好是 $t-1$ 的节点传送。

基础步骤:$t=1$,即初始时,所有的 parent 变量是 nil,而且消息〈M〉只从 p_r 输出。

假设对于第 $t-1\geqslant 1$ 轮,断言是真的。在第 $t-1$ 轮中,来自距离为 $t-2$ 的节点的消息〈M〉被接收。接收〈M〉的任一节点与 p_r 的距离是 $t-1$ 或更小。parent 变量不为 nil 的接收者节点,即距离 p_r 是 $t-2$ 或更小的节点,不会改变它的 parent 变量或发送出去一条〈M〉消息。距离 p_r 为 $t-1$ 的每个节点,在第 $t-1$ 轮接收到一条〈M〉消息,因为它的 parent 变量为 nil,就设置它为适当的父节点,并发送出去一条〈M〉消息。不在距离 p_r 为 $t-1$ 的节点不会接收到〈M〉消息,从而不会发送任何消息。 □

定理 2.9 *给定一个特定节点,存在可找到有 m 条边且直径为 D 的 BFS 树的同步算法,其消息复杂度为 $O(m)$、时间复杂度为 $O(D)$。*

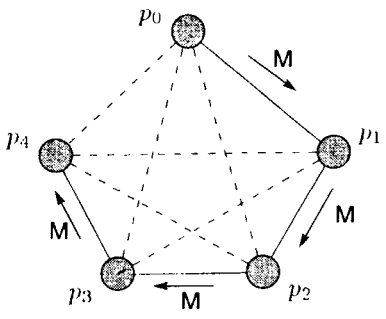

图 2.6 一棵非 BFS 树

在异步系统中,修改后的洪泛算法可能不会构造 BFS 树。考虑具有 5 个节点(p_0 到 p_4)的完全连接的网络,其中 p_0 是根(见图 2.6)。假定消息〈M〉按照 $p_0\rightarrow p_1$,$p_1\rightarrow p_2$,$p_2\rightarrow p_3$,以及 $p_3\rightarrow p_4$ 的顺序快速传播,而其他〈M〉消息很缓慢。则结果产生的生成树是从 p_0 到 p_4 的链,而不是一棵 BFS 树。此外,该生成树虽然直径只有 1,但深度为 4。注意,算法的运行时间是与直径而不是与节点数成正比的。练习 2.5 要求针对有 n 个节点的图来推广这一观测结果。

修改后的洪泛算法可以与前面描述的敛播算法相结合,来请求并收集信息。结合的算法可在同步或异步系统中工作。但是,结合算法的时间复杂度在两种模型中是不同的;因为我们不必在异步模型中获取 BFS 树,敛播就可能被应用到深度为 $n-1$ 的树中。然而在同步情况中,敛播总是被应用到深度最多为网络直径的树中。

2.4 构造指定根的深度 - 优先搜索生成树

另一基本算法是以某一特定节点为根,构造通信网络的深度 - 优先搜索(depth-first search,DFS)树。构造 DFS 树是通过每次增加一个节点完成的,比用算法 2 构造生成树更渐进些,算法 2 是试图将树中同一级的所有节点同时增加到图中。

深度 - 优先搜索的伪代码见算法 3。

因为此算法的执行中没有并发性,所以算法 3 的正确性本质上是根据串行 DFS 算法的正确性得出的。引理 2.10 的详细证明留作练习。

引理 2.10 *对异步模型中的每一次合法执行,算法 3 构造网络的一棵根为 p_r 的 DFS 树。*

为计算该算法的消息复杂度,注意每个处理器在它的每条邻接边上最多发送一次〈M〉;同时,每个处理器最多生成一条消息(〈already〉或〈parent〉),以回应从每条邻接边上接收〈M〉。因此,算法 3 最多发送 $4m$ 条消息。证明算法的时间复杂度是 $O(m)$,留给读者作为练习。我们总结如下:

定理 2.11　给定一个特定节点，存在可找到有 m 条边和 n 个节点的网络深度－优先搜索生成树的异步算法，其消息复杂度为 $O(m)$、时间复杂度为 $O(m)$。

算法 3　指定根的深度－优先搜索生成树算法：处理器 p_i 的代码，$0 \leqslant i \leqslant n-1$

Initially *parent* $= \perp$, *children* $= \emptyset$, *unexplored* = all neighbors of p_i

```
1:  upon receiving no message:
2:      if p_i = p_r and parent = ⊥ then                    // root wakes up
3:      parent := p_i
4:      explore()

5:  upon receiving ⟨M⟩ from p_j:
6:      if parent = ⊥ then                    // p_i has not received ⟨M⟩ before
8:          parent := p_j
9:          remove p_j from unexplored
10:         explore()
11:     else
12:         send ⟨already⟩ to p_j                           // already in tree
13:         remove p_j from unexplored
14: upon receiving ⟨already⟩ from p_j:
15:     explore()

16: upon receiving ⟨parent⟩ from p_j:
17:     add p_j to children
18:     explore()

19: procedure explore():
20:     if unexplored ≠ ∅ then
21:         let p_k be a processor in unexplored
22:         remove p_k from unexplored
23:         send ⟨M⟩ to p_k
24:     else
25:         if parent ≠ p_i then send ⟨parent⟩ to parent
26:         terminate                  // DFS subtree rooted at p_i has been built
```

2.5　构造不指定根的深度－优先搜索生成树

算法 2 和算法 3 构建通信网络的生成树，具有合理的消息与时间复杂度。然而，它们两者都需要存在一个特定节点，从该节点开始构造生成树。在本节，我们讨论当不存在特定节点时如何构建一棵生成树。假设各节点都有唯一的自然数标识符；如同将在第 3.2 节中看到的一样，这一假设是必需的。

为构建一棵生成树，每个自动醒来(wake up)的处理器试图构建以自己为根的 DFS 树，使用的是算法 3 的独立复制。如果两棵 DFS 树试图连接到同一节点(不必在同时)，则该节点将加入到根有较高标识符的 DFS 树中。

伪代码见算法 4。为实现上述思想，每个节点将到目前为止所看到的最大标识符保存到变量 leader 中，该变量初始化为小于所有标识符的值。

当一个节点自动醒来时,设置 leader 为自己的标识符,并发送一条携带其标识符的 DFS 消息。当某个节点接收到一条带有标识符 y 的 DFS 消息时,它将比较 y 和 leader。如果 $y > leader$,那么这可能是具有最大标识符的处理器的 DFS 消息;在这种情况下,节点将 leader 修改为 y,设置它的 parent 变量为发送该消息的节点,并且继续发送带标识符 y 的 DFS 消息。如果 y = leader,那么该节点就已经属于这棵生成树。如果 $y <$ leader,那么该 DFS 消息所属的节点,其标识符小于到目前为止所见到的最大识别符;在这种情况下,不发送消息,停止用标识符 y 来进行 DFS 树的构造。最后,携带标识符 leader(或更大的标识符)的 DFS 消息,将到达标识符为 y 的节点,并将它连接到树中。

算法 4 生成树的构造:处理器 p_i 的代码,$0 \leqslant i \leqslant n-1$

```
Initially parent = ⊥, leader = −1, children = ∅, unexplored = all neighbors of p_i

1:  upon receiving no message:
2:      if parent = ⊥ then                                  // wake up spontaneously
3:          leader := id
4:          parent := p_i
5:          explore()

6:  upon receiving ⟨leader,new-id⟩ from p_j:
7:      if leader < new-id then                             // switch to new tree
8:          leader := new-id
9:          parent := p_j
10:         children := ∅
11:         unexplored := all neighbors of p_i except p_j
12:         explore()
13:     else if leader = new-id then
14:         send ⟨already,leader⟩ to p_j                    // already in same tree
                // otherwise, leader > new-id and the DFS for new-id is stalled

15: upon receiving ⟨already,new-id⟩ from p_j:
16:     if new-id = leader then explore()

17: upon receiving ⟨parent,new-id⟩ from p_j:
18:     if new-id = leader then                             // otherwise ignore message
19:         add p_j to children
20:         explore()

21: procedure explore():
22:     if unexplored ≠ ∅ then
23:         let p_k be a processor in unexplored
24:         remove p_k from unexplored
25:         send ⟨leader,leader⟩ to p_k
26:     else
27:         if parent ≠ p_i then send ⟨parent,leader⟩ to parent
28:         else terminate as root of spanning tree
```

只有所构造的生成树的根可以明确终止;其他节点不会终止而是保持等待消息。可以修改算法使根用算法 1 来发送终止消息。

证明该算法的正确性，比本章前面的算法要更复杂；在此仅仅粗略描述论据。考虑自动醒来的各节点，且设 p_m 是其中标识符最大的节点；m 是 p_m 的标识符。

首先观察到 leader id = m 的〈leader〉消息从不停止，因为根据 m 的定义不会有更大的 leader id。

其次，leader id = m 的〈already〉消息从不停止，因为它们有错误的 leader id。为什么这么说呢？假如 p_i 从 leader id = m 的 p_j 接收到一条〈already〉消息，p_j 发送该消息给 p_i 的原因是它从 leader id = m 的 p_i 接收到一条〈leader〉消息。一旦 p_i 将其 leader id 设置为 m，它不会再重新设置，因为 m 是系统中最大的 leader id。因此，当 p_i 从 p_j 处接收到 leader id = m 的〈already〉消息时，p_i 仍然保持它的 leader id 为 m，接收该消息，并执行 explore()。

第三，leader id = m 的〈parent〉消息从不停止，因为它们有错误的 leader id。其论据与〈already〉消息的相同。

最后，leader id = m 的消息从不停止，因为接收者已经终止。用矛盾法假设某个 p_i 在接收到具有 leader id = m 的消息前已经终止，那么 p_i 认为它是 leader，但它的 id，比如 i，小于 m。对于 leader id = i 的算法 3 复制，应该已到达图中每个节点，包括 p_m。但是 p_m 可能没有响应 leader 消息，因而这份算法 3 复制可能没有完成，产生矛盾。

因此，对于 leader id = m，算法 3 的复制已完成，而且算法 3 的正确性表明算法 4 是正确的。

算法的一个简单分析使用以下事实：在最坏情况下，每个处理器都试图构造一棵 DFS 树。因此，算法 4 的消息复杂度，最多是算法 3 的消息复杂度的 n 倍，即 $O(n \cdot m)$；时间复杂度类似于算法 3 的时间复杂度，即 $O(m)$。

定理 2.12　*算法 4 可找到有 m 条边和 n 个节点的生成树，其消息复杂度为 $O(n \cdot m)$，时间复杂度为 $O(m)$。*

练习

2.1　用状态转移方式，对一个简单算法进行编码。

2.2　当通信是同步和异步时，分析第 2.2 节中敛播算法的时间复杂度。

提示：对于同步情况，证明在第 $t+1$ 轮中，高度为 t 的某个处理器发送一条消息给它的父节点。对于异步情况，证明到时间 t，高度为 t 的某个处理器发送一条消息给它的父节点。

2.3　推广第 2.2 节的敛播算法，使它收集所有信息。也就是说，当算法终止时，根应当有所有处理器的输入值。分析位复杂度，即在通信信道上发送的位数总和。

2.4　**证明**：在引理 2.6 的证明中使用的断言，即某处理器在 G 中可从 p_r 到达的条件是当且仅当它设置过自己的 parent 变量。

2.5　描述修改后的洪泛算法（算法 2）在 n 个节点的异步系统中执行时，不能构造一棵 BFS。

2.6　描述算法 2 在某一异步系统中的执行，该系统中消息在通信信道上发送两次，这些信道在生成树中未连接到父节点及其子节点。

2.7 在同步和异步模型中,对修改后的洪泛算法(算法 2)的时间复杂度进行精确分析。

2.8 在同步情况下,解释该如何从修改后的洪泛算法中删除〈already〉消息,且仍使算法正确。结果算法的消息复杂度如何?

2.9 广播和敛播算法依赖于系统中的节点数吗?

2.10 修改算法 3,使它能够正确处理特定节点无邻居的情况。

2.11 修改算法 3,使所有的节点终止。

2.12 证明算法 3,可以构造根在 p_r 的网络 DFS 树。

2.13 证明算法 3 的时间复杂度是 $O(m)$。

2.14 修改算法 3,使它能够构造节点的 DFS 编号,该编号可以指出消息〈M〉到达节点的顺序。

2.15 修改算法 3,使它成为一个新算法,用来构造具有时间复杂度 $O(n)$的 DFS 树。
提示:当某个节点首次接收到消息〈M〉时,它通知所有邻居,但只将消息传递给其中之一。

2.16 证明定理 2.12。

2.17 证明在算法 4 中,如果 leader 变量不包括在〈parent〉消息中,且第 18 行的测试没有执行,那么算法是不正确的。

本章注释

本章的第一部分介绍分布式消息传递系统的形式化模型;这一个模型最接近由 Attiya, Dwork, Lynch 和 Stockmeyer[27]所使用的模型,而在关于分布式算法的文献中,有许多文章使用了类似的模型。

将分布式算法中每个处理器模拟成状态机的思想,至少可追溯到 Lynch 和 Fischer[176]。最早明确将分布式系统中的执行,表示为状态转移序列的两篇文章是 Owicki 和 Lamport[204],以及 Lynch 和 Fischer[176]。同样的思想也在 Owicki 和 Gries[203]的论文中体现,但不够明显。

一些研究人员(如 Fischer, Lynch 和 Paterson[110])使用术语"合法的",以区分出那些满足附加约束的执行。也就是说,术语"执行"是指满足某些相对基本句法约束的序列,而"合法的"指出要满足附加的特性。但是具体细节在不同文章中都不一样。Emerson 写的一章[103]中包含了对安全性和活跃性的详细讨论,提供了很多参考文献。

异步的时间复杂度度量,最早是由 Peterson 和 Fischer[214]针对共享存储器的情况提出来的,并且很自然地被扩展到了消息传递的情况中,如 Awerbuch 的文章[36]。

本章中介绍的形式化模型,在本书的第一部分使用,没有涉及合成及与用户的交互;本书第二部分将讨论这些问题。

本章的第二部分介绍了消息传递系统的一些算法,展示了如何使用形式化和前面介绍的复杂度度量。这些算法解决广播、敛播、DFS、BFS 及领导者选举问题;Gafni[115]将这些问题包含在一组有用的"构件"中,用来构造消息传递系统的算法。

本章中介绍的算法是众人的智慧。第 2.2 节的广播和敛播算法是由 Segall[240]描述的,他还描述了以某一特定节点为根找到 BFS 树的异步算法。BFS 树对以最少时间在网络中广播信息是很有用的,它使信息在处理器间沿最短路径进行传送。从指定根构造 DFS 树的算法

(算法 3)最早出现在 Cheung[78]中。一个具有线性时间复杂度的构造 DFS 树的算法(练习 2.15)是由 Awerbuch[37]提出的。

算法 4 用来构造一棵生成树,其工作原理是消去低标识符处理器的 DFS 树。这一算法是众人的智慧,我们的介绍是受到 Gallager 算法[117]的启发。Gallager 算法的消息复杂度是 $O(m + n \log n)$;它对树的展开进行了仔细平衡,该树由各个不同的处理器构造,它确保只消去小的树,并且不丢失太多的操作。

练习 2.17 是由 Uri Abraham 提出的。

算法 4 和 Gallager 算法产生网络的生成树,没有考虑在不同通信链上发送消息的代价。如果根据其代价,在代表通信链的边上赋以权值,那么网络的最小加权生成树(minimum-weight spanning tree)可将广播和敛播的通信代价减到最少。

构造最小加权生成树的算法是由 Gallager, Humblet 和 Spira[118]给出的;该算法的消息复杂度是 $O(m + n \log n)$。下一章包含的下界指出选举领导者需要的消息数为 $\Omega(n \log n)$。找到一棵最小加权生成树的算法,其时间复杂度是 $O(n \log n)$;Awerbuch[39]介绍了一个寻找最小加权生成树的算法,其时间复杂度是 $O(n)$、消息复杂度是 $O(n \log n)$。

因为在任一生成树算法中,只有一个节点终止为根,所以可以称之为领导者;领导者非常有用,第 3 章专门讨论环拓扑中通过尽可能少的消息来选举领导者的问题。

第 3 章　环中领导者选举算法

本章考虑拓扑结构是环的消息传递系统。环是一种简便的消息传递系统结构，有相应的实际通信系统，如令牌环。我们研究领导者选举(leader election)问题，即在一组处理器中选择其中一个成为领导者。领导者的存在能够简化处理器间的协调，并有助于达到故障容错和节省资源——回顾一下，第 2 章中因特殊处理器 p_r 的存在，而使广播问题得到简单解决。此外，领导者选举问题，代表了一类打破对称的问题。举例来说，当发生死锁时，因为处理器之间彼此循环等待，通过选举其中一个处理器成为领导者，并将它从循环中移去，就能打破死锁。

3.1　领导者选举问题

领导者选举问题有多个变种，下面我们定义最一般的形式。通俗而言，就是每个处理器最终要决定自己是领导者或不是领导者，约束条件是只能有一个处理器经过决定成为领导者。按照我们的形式化模型，一个算法如果满足以下条件，就认为它解决了领导者选举问题。

- 终止状态分为被选举状态和未被选举状态。一旦某个处理器进入被选举(或未被选举)状态，它的转移函数只能将它转移到另一个(或同一个)被选举(或未被选举)状态。
- 在每次合法执行中，只有一个处理器(领导者)进入被选举状态，而所有其他处理器进入未被选举状态。

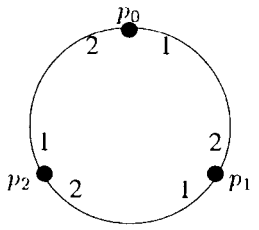

图 3.1　一个简单可定向的环

我们将注意力集中在系统拓扑结构是环的情形。特别地，假设拓扑结构图中的边是在 p_i 和 p_{i+1} 之间(对所有 i，$0 \leqslant i < n$，且在加 1 时取模 n)。此外，假设处理器有一致的左、右边概念，形成一个可定向的环。根据要求，这一假设的形式化模型如下：对每个 i，$0 \leqslant i < n$，从 p_i 到 p_{i+1} 的通道标记为 1，也就是往左或顺时针方向；而 p_i 到 p_{i-1} 的通道标记为 2，也就是往右或逆时针方向(同样，加和减均取模 n)。图 3.1 是一个环的简单例子，包含三个节点(关于可定向的更多介绍见本章注释)。

3.2　匿名环

环系统中领导者选举算法是“匿名的(anonymous)”，是指各处理器没有唯一标识符可供算法使用。更形式化的描述为：每个处理器在系统中具有相同的状态机。在描述匿名算法时，消息的接受者仅能根据通道标记来指定，如左邻和右邻。

对于算法，处理器数 n 是一个有用的信息。如果 n 对于算法是未知的，也就是说，对 n 没有预先进行硬编码，则称算法是“一致性的(uniform)”，因为算法对每个 n 值来说，看起来是相同的。形式化的描述如下：在一个匿名的、一致性的算法中，不论环的规模，所有处理器只

有一个状态机；在一个匿名的、非一致性的算法中，对每个 n 值（即环的规模）都有单个状态机，但对不同的规模有不同的状态机，也就是说，n 可以在代码中显式表达。

我们将证明环系统不可能存在匿名的领导者选举算法。

考虑到一般性和简单性，我们证明非一致性算法和同步环的上述结论。从同步环的不可能性，可立即推断出异步环的相同结论（见练习 3.1）。类似地，从非一致性算法的不可能性，即算法中处理器数 n 为已知，可推断出算法中 n 为未知时的不可能性（见练习 3.2）。

回顾一下，在同步系统中，算法按轮推进，在每轮中所有待处理的消息被提交，然后每个处理器执行一个计算步骤。处理器的初始状态在 outbuf 变量中，它包含了在第一轮中要提交给左邻和右邻的消息。

支持上述不可能性结论的思想是：在一个匿名环中，各处理器之间的对称性总能保持；也就是说，如果没有某种初始的不对称性，比如由唯一标识符引起的不对称性，对称性将不可能被打破。显然，在匿名环算法中所有处理器从相同状态开始启动。由于各处理器是相同的，并执行相同程序（即它们具有相同的状态机），所以在每一轮中它们都发送相同的消息。因此，它们在每一轮中都接收相同的消息，完全相同地改变状态。这样的结果是：如果一个处理器被选举，那么所有处理器都被选举。所以，不可能存在一个算法，在这种环中选举出单个的领导者。

为使这种直觉形式化，考虑一个环 R，其规模为 $n > 1$。采用反证法，假设存在一种匿名算法 A，可从这种环中选举出一个领导者。因为环是同步的，且只有一种初始配置，故在 R 上 A 只有唯一合法执行。

引理 3.1　*在 R 上 A 的合法执行中，对每一轮 k，所有处理器在 k 结束时的状态是相同的。*

证明　证明采用对 k 的归纳法。基本步骤：$k = 0$（在第一轮之前）是显然的，因为各处理器都从相同的初始状态开始。

在归纳步骤中，假设 $k-1$ 时引理成立。因为各处理器在 $k-1$ 轮时处于相同状态，它们全都往右发送相同的消息 m_r，往左发送相同的消息 m_ℓ。在第 k 轮时，各处理器从右接收消息 m_ℓ，从左接收消息 m_r。这样，所有的处理器在第 k 轮时都接收完全相同的消息；因为它们执行相同的程序，所以在第 k 轮结束时都处于相同的状态。　□

上述引理意味着：如果在某一轮结束时，某一处理器宣布自己成为领导者，进入被选举状态，那么所有其他处理器也同样会如此。这与 A 是一种领导者选举算法的假设相矛盾，从而证明了以下定理：

定理 3.2　*在同步环中不存在非一致性的匿名领导者选举算法。*

3.3　异步环

本节介绍异步环中领导者选举问题的消息复杂度上界和下界。因为定理 3.2 只是表明在环中不存在匿名的领导者选举算法，所以在本章的其他部分，我们假设各处理器具有唯一的标识符。

假设环中每个处理器都有一个唯一的标识符，每个自然数都可作为一个标识符。当每个处理器 p_i 都关联一个状态机（局部程序）时，就有一个可识别的状态成员 id_i，初始化为标识符的值。

我们从最小的标识符开始,按顺时针次序列出各处理器的标识符,以此来确定一个环。这样,给每个处理器 $p_i(0 \leqslant i < n)$分配一个标识符 id_i。注意,当两种标识符分配,其中一种是另一种的循环移位时,按这种定义就是同一个环,因为各处理器本身的索引号(例如,处理器 p_{97} 中的 97)是不可获得的。

有了唯一标识符后,对一致性算法和非一致性算法的表述稍微有些不同。

一个(非匿名)算法是一致性的,是指每个标识符都对应有一个状态机;不管环的规模如何,只要处理器分配了对应其标识符的唯一状态机,算法就是正确的。也就是说,对具有给定标识符的处理器,只有一个局部程序,而无论该处理器是处于哪种规模的环中。

一个(非匿名)算法是非一致性的,是指对应每一个 n 和每一个标识符,都有一个状态机。对每一个 n,给定规模为 n 的任何一个环,当算法中每个处理器具有对应其标识符和环规模 n 的状态机时,算法必须是正确的。

我们从一个非常简单的异步环中领导者选举算法开始,它需要的消息数为 $O(n^2)$。然后在此算法上,导出一个更有效率的算法,只需要消息数为 $O(n \log n)$。我们证明该算法具有最优的消息复杂度,它在选举一个领导者所需要的消息数上,具有一个下界 $\Omega(n \log n)$。

3.3.1 $O(n^2)$算法

在本算法中,各处理器发送带有自己标识符的消息给它的左邻,然后等待来自右邻的消息。当它接收到这种消息时,检查消息中的标识符,如果该标识符大于它自己的标识符,那么就转发该消息给左邻;否则它就"吞没(swallows)"该消息,不再转发。如果某处理器接收到带有自己标识符的消息,就通过向左邻发送一条终止消息来宣布自己是领导者,并以领导者的身份结束处理。当一个处理器接收到终止消息时,它就向左邻转发,并以非领导者结束处理。注意,算法并不依赖于环的规模,即它是一致性的。

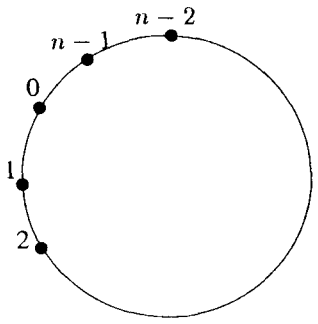

图 3.2 消息数为 $\Theta(n^2)$的环

我们注意到,在任一合法执行中,只有带有最大标识符的处理器消息不会被"吞没"。因此,只有具备最大标识符的处理器,能够接收到带有它自己标识符的消息,从而宣布自己为领导者。其他所有处理器将接收到终止消息,而不会被选举为领导者。这说明了算法的正确性。

显然,在任一合法执行中,算法发送的消息数不会超过 $O(n^2)$。此外,存在以下合法执行,其中算法发送的消息数为 $\Theta(n^2)$:考虑处理器标识符为 $0,\cdots,n-1$ 的环,它们如图 3.2 那样排序。在这种结构中,带标识符 i 的处理器消息正好被发送 $i+1$ 次。因此,加上 n 次终止消息,消息总数为 $n + \sum_{i=0}^{n-1}(i+1) = \Theta(n^2)$。

3.3.2 $O(n \log n)$算法

基于与上节算法相同的思想,我们提出了一个更有效的算法。同样,各处理器在环中发送自己的标识符,算法确保只有带有最大标识符的处理器消息,能够遍历整个环并返回。但算法应用了一个更聪明的方法来转发标识符,从而将最坏情况下的消息数从 $O(n^2)$减少到 $O(n \log n)$。

为了描述该算法，我们首先定义环中处理器 p_i 的 k-邻域（k-neighborhood），即环中距离 p_i 最多是 k（从左或从右）的处理器集合。注意，一个处理器的 k-邻域正好包含 $2k+1$ 个处理器。

算法按阶段（phase）来执行，为方便起见，开始阶段的编号设为 0。在第 k 阶段，处理器试图成为该阶段的获胜者；为了成为获胜者，它必须在 2^k-邻域中有最大的 id（标识符）。只有在第 k 阶段成为获胜者的处理器，才能继续进入第 $k+1$ 阶段的竞争。这样，越高的阶段就有越少的处理器，最后只有一个处理器成为获胜者，并被选举为整个环的领导者。

详细描述如下，在阶段 0，各处理器试图变成阶段 0 的获胜者，并发送一个包含其标识符的探查消息〈probe〉给它的 1-邻域，也就是它的左右两个邻居。对于接收到该探查消息的邻居，如果其标识符大于探查消息中的标识符，它就吞没该消息；否则，它就回送一个回答消息〈reply〉。如果某个处理器接收到它两个邻居的回答消息，那么该处理器就成为阶段 0 的获胜者并继续进入阶段 1。

一般而言，在阶段 k，处理器 p_i 是阶段 $k-1$ 的获胜者，它发送带有其标识符的〈probe〉消息给它的 2^k-邻域（每个方向一条），这一消息将逐一通过 2^k 个处理器。当处理器发现探查消息中的标识符小于它自己的标识符时，就吞没该消息。如果探查消息到达邻域中最后一个处理器而没有被吞没，那么最后这个处理器回送一条〈reply〉消息给 p_i。当 p_i 从两个方向接收到回答消息时，它就成为阶段 k 的获胜者并继续进入到阶段 $k+1$。当一个处理器接收到它自己的〈probe〉消息时，就终止算法且成为领导者，并向环中发送终止消息。

注意，为了实现该算法，2^k-邻域中最后一个处理器必须返回一个〈reply〉消息，而不是转发〈probe〉消息。这样，每条〈probe〉消息中就有三个域：标识符、阶段号和跳步（hop）计数器。跳步计数器初始化为 0，且在处理器每次转发该消息时加 1。如果某处理器接收到阶段 k 消息，且跳步计数器为 2^k，那么它就是 2^k-邻域中的最后一个处理器。

伪代码见算法 5。对某处理器而言，阶段 k 对应的时期，是从第 4 行或第 15 行用第三个参数 k 发送〈probe〉消息，到用第三个参数 $k+1$ 发送〈probe〉消息。代码中省略了在环中发送终止消息的细节，只有领导者执行终止操作。

算法的正确性采用与前述简单算法的相同方式来说明，因为它们采用了相同的吞没规则。很显然，具有最大标识符的处理器探查消息不会被吞没；因此，该处理器将终止算法而成为领导者。另一方面，也没有其他〈probe〉消息能够遍历整个环而不被吞没。所以，具有最大标识符的处理器是算法选举出的唯一领导者。

为分析算法合法执行中最坏情况下的发送消息数，首先注意到在阶段 k 时探查距离是 2^k，因此一个特定的竞争处理器在阶段 k 发送的消息数为 $4\cdot 2^k$。那么在最坏的情况下，有多少处理器参与竞争呢？对于 $k=0$，数量是 n，因为所有处理器都启动算法。对于 $k\geqslant 1$，参与阶段 k 竞争的处理器是阶段 $k-1$ 的获胜者。下面的引理给出了各阶段中获胜者数量的上界。

引理 3.3　对于 $k\geqslant 1$，成为阶段 k 获胜者的处理器数至多为 $\frac{n}{2^{k-1}+1}$。

证明　如果处理器 p_i 是阶段 k 的获胜者，那么在 p_i 的 2^k-邻域中，各处理器的 id（标识符）一定小于 p_i 的 id。阶段 k 两个最靠近的获胜者 p_i 和 p_j，p_i 的 2^k-邻域左面正好是 p_j 的 2^k-邻域右面。也就是说，在 p_i 和 p_j 之间有 2^k 个处理器。当环中保持这种密集性时，阶段 k 的获胜者数达到最大，这种情况下的获胜者数为 $\frac{n}{2^{k-1}+1}$。　□

算法 5 异步领导者选举:处理器的 p_i 代码,$0 \leq i < n$

```
Initially, asleep = true

1:  upon receiving no message:
2:      if asleep then
3:          asleep := false
4:          send ⟨probe,id,0,1⟩ to left and right

5:  upon receiving ⟨probe,j,k,d⟩ from left (resp., right):
6:      if j = id then terminate as the leader
7:      if j > id and d < 2^k then                          // forward the message
8:          send ⟨probe,j,k,d + 1⟩ to right (resp., left)   // increment hop counter
9:      if j > id and d ≥ 2^k then                          // reply to the message
10:         send ⟨reply,j,k⟩ to left (resp., right)
                                          // if j < id, message is swallowed

11: upon receiving ⟨reply,j,k⟩ from left (resp., right):
12:     if j ≠ id then send ⟨reply,j,k⟩ to right (resp., left)    // forward the reply
13:     else                                                     // reply is for own probe
14:         if already received ⟨reply,j,k⟩ from right (resp., left) then
15:             send ⟨probe,id,k + 1,1⟩                          // phase k winner
```

根据以上引理,当阶段数至少为 $\log(n-1)$时,只有一个获胜者。在下一阶段,该获胜者选举自己成为领导者。那么包括 $4n$ 阶段 0 条消息和 n 条终止消息,消息的总数至多为:

$$5n + \sum_{k=1}^{\lceil \log(n-1) \rceil + 1} 4 \cdot 2^k \cdot \frac{n}{2^{k-1}+1} < 8n(\log n + 2) + 5n$$

总结出以下定理:

定理 3.4 *存在一种异步领导者选举算法,其消息复杂度为 $O(n \log n)$。*

注意,与第 3.3.1 节的简单算法相反,本算法使用环中的双向通信。本算法的消息复杂度中常数因子 8 不是最优的,本章注释中介绍了达到更小常数因子的文章。

3.3.3 下界 $\Omega(n \log n)$

本节要证明第 3.3.2 节的领导者选举算法是接近最优的,也就是说,要证明在一个异步环中任何选举领导者的算法,至少发送的消息数为 $\Omega(n \log n)$。我们证明的这一下界适用于一致性算法,即不知道环的规模的算法。

我们针对一种特殊类型的领导者选举问题来证明这一下界,这种类型选举的领导者一定是环中有最大标识符的处理器;另外,所有的处理器必须知道所选举的领导者的标识符。也就是说,各处理器在终止前,将所选举的领导者的标识符写入一个特殊变量中。对于更一般化定义的领导者选举问题的下界的证明,可按归纳法进行,留作练习 3.5。

假设给定一个一致性算法 A 来解决上述类型的领导者选举问题。我们将证明存在 A 的某一合法执行,其中发送消息数为 $\Omega(n \log n)$。从直观来看,针对规模为 $n/2$ 的环,建立算法

的一种“耗费型(wasteful)”执行方式,在执行中发送大多数消息。然后,再将两个规模为 $n/2$ 的环“粘贴在一起(paste together)”,形成规模为 n 的环。通过这一方式,我们可将较小环的“耗费型”执行组合在一起,并强制接收 $\Theta(n)$ 条附加消息。

尽管前面的讨论是把执行“粘贴在一起”,但实际操作中采用的是调度,其原因是执行包含了配置,它限制了环中的处理器数。我们将把同样的事件序列应用于具有不同处理器数的各种环。在提出下界证明的细节之前,首先定义可以“粘贴在一起”的调度。

定义 3.1　对某个特定环,如果存在一条边 e,在 A 的调度中 σ 没有任何消息从任一方向经过 e,那么就称 σ 是开放的,e 是 σ 的一条开放边。

注意,开放的调度不必是合法的;特别地,它可以是有限的,且处理器可能尚未终止。

从直观来看,由于各处理器不知道环的规模,所以我们可以把两个小环的开放调度“粘贴在一起”,形成一个更大环的开放调度。注意,这种扩展是依赖于以下事实,即算法是一致性的,且对各种环的规模以相同的方式执行。

我们现在给出详细证明。为清楚起见,在下面的证明中假设 n 是 2 的整数次幂(练习 3.6 要求对 n 的其他值证明下界)。

定理 3.5　对于每个 n 及 n 个标识符的集合,存在一个使用这些标识符的环,它有 A 的一个开放调度,在该调度中至少接收 $M(n)$ 条消息。其中 $M(2)=1$,且对于 $n>2$,$M(n)=2M\left(\frac{n}{2}\right)+\frac{1}{2}\left(\frac{n}{2}-1\right)$。

因为 $M(n)=\Theta(n \log n)$,这一定理表明了所期望的下界。定理的证明采用归纳法。引理 3.6 是基本步骤($n=2^1$),引理 3.7 是归纳步骤($n=2^i, i>1$)。在基本步骤中,环包含两个处理器,我们假定存在两条不同的链路连接处理器。

引理 3.6　对包含两个标识符的每一个集合,存在一个使用该两个标识符的环 R,它有 A 的一个开放调度,在该调度中至少接收一条消息。

证明　假设 R 包含处理器 p_0 和 p_1,且 p_0 的标识符(设为 x)大于 p_1 的标识符(设为 y)(见图 3.3)。

设 α 是环上 A 的一次合法执行,因为 A 是正确的,所以在 α 中 p_1 最终必定写入 p_0 的标识符 x。注意,在 α 中至少接收一条消息;否则,如果 p_1 不从 p_0 获得一条消息,它就不会知道 p_0 的标识符是 x。设 σ 是 α 具有最短前缀的调度,它包含接收一条消息的首个事件。注意到,除接收首条消息的边以外,另一条边是开放的。因在 σ 中只接收一条消息且有一条边是开放的,故 σ 显然是一种开放的调度,满足引理的要求。

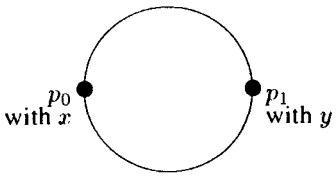

图 3.3　引理 3.6 图示

下一引理将介绍粘贴过程的归纳步骤。如前所述,一般的方法是基于两个较小环上的开放调度,其中已接收了大多数消息,将它们在开放边处粘贴在一起,形成一个更大环上的开放调度,其中除上述消息外,还有额外接收的消息。从直观上看,两个开放调度粘贴在一起,仍将保持同样的行为(将在下面做形式化证明)。然而,关键的步骤是强制接收附加的消息。在两个较小环粘贴起来后,在未包含最终领导者的那一半中,各处理器必须以某种方式学习到最终领导者的 id,而这只能通过消息交换来实现。我们要接通在连接的开放边上阻塞的消

息,并继续进行调度,证明必须接收许多消息。我们的主要问题是如何在更大环上产生一个开放调度,使得引理可以递归使用。其困难是:如果预先选择连接两个部分的两条边中的哪条接通,那么算法可以选择在另一条边上等待信息。为避免这一问题,我们首先创建一个"测试"调度,用来了解接通时,两条边中哪条接收到更多的消息数,然后我们回到原来的粘贴过程并只接通上述这条边。

引理 3.7 选择 $n>2$。假设对包含 $n/2$ 个标识符的每一个集合,存在一个使用这些标识符的环,它有 A 的一个开放调度,调度中至少接收 $M\left(\frac{n}{2}\right)$ 条消息。那么对包含 n 个标识符的每一集合,存在一个使用这些标识符的环,它有 A 的一个开放调度,调度中接收的消息数至少为 $2M\left(\frac{n}{2}\right)+\left(\frac{1}{2}\right)\left(\frac{n}{2}-1\right)$。

证明 设 S 是 n 个标识符的集合。将 S 划分为两个集合 S_1 和 S_2,每个规模为 $n/2$。根据假设,在 S_1 中存在一个使用标识符的环 R_1,有 A 的一个开放调度 σ_1,在该调度中至少接收 $M\left(\frac{n}{2}\right)$ 条消息。类似地,在 S_2 中存在一个使用标识符的环 R_2,有 A 的一个开放调度 σ_2,在调度中至少接收 $M\left(\frac{n}{2}\right)$ 条消息。设 e_1 和 e_2 分别为 σ_1 和 σ_2 的开放边。e_1 连接的处理器用 p_1 和 q_1 表示,e_2 连接的处理器用 p_2 和 q_2 表示。通过删除边 e_1 和 e_2,用边 e_p 连接 p_1、p_2,边 e_q 连接 q_1、q_2,而将 R_1 和 R_2 粘贴在一起;将结果环表示为 R(如图 3.4 和 3.5 所示)。

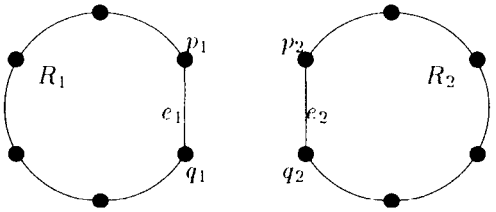

图 3.4 R_1 和 R_2

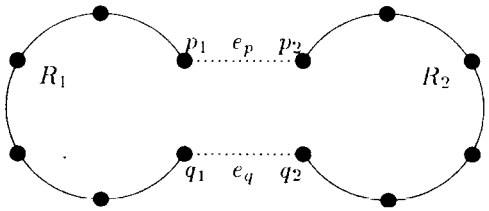

图 3.5 将 R_1 和 R_2 粘贴成 R

下面展示如何在 R 上构造 A 的一个开放调度 σ,其中接收到 $2M\left(\frac{n}{2}\right)+\frac{1}{2}\left(\frac{n}{2}-1\right)$ 条消息。其思想是首先让每个较小的环分别执行其"耗费型"开放调度。

现在解释为什么 σ_1 接上 σ_2,就构成了环 R 上 A 的一个调度。考虑事件序列 σ_1 的发生开始于环 R 的初始配置中,由于 R_1 中的处理器在这些事件中不能识别 R_1 是独立环,还是 R 的子环,所以它们按 R_1 是独立的来执行 σ_1。考虑在环 R 中随后发生的事件序列 σ_2,同样由于没有消息在连接 R_1 和 R_2 的边上提交,所以 R_2 中的处理器在这些事件中无法识别 R_2 是独立环,还是 R 的子环。注意,这里的关键是依赖于一致性的假设。

因此,$\sigma_1\sigma_2$ 是 R 的一个调度,其中至少接收 $2M\left(\frac{n}{2}\right)$ 条消息。

现在描述如何通过接通和强制算法(不是两者同时)接收 $\frac{1}{2}\left(\frac{n}{2}-1\right)$ 条附加消息。

考虑形式为 $\sigma_1\sigma_2\sigma_3$ 的有限调度,其中 e_p 和 e_q 都保持开放。如果存在一种调度,在 σ_3 中至少接收 $\frac{1}{2}\left(\frac{n}{2}-1\right)$ 条消息,那么就证明了引理。

假设不存在上述调度,那么就存在某种调度 $\sigma_1\sigma_2\sigma_3$,在相应的执行中导致一种"静默的(quiescent)"配置。一个处理器状态称为"静默的",是指不存在计算事件序列在这种状态下发

送消息的情况。也就是说，处理器在接收消息前不会发送另一消息。某一配置称为“静默的”（关于 e_p 和 e_q），是指在开放边 e_p 和 e_q 之外没有消息通过，且各处理器处于“静默的”状态。

为不失一般性，现假设 R 中标识符最大的处理器是在子环 R_1 中。由于没有消息从 R_1 提交到 R_2，R_2 中的处理器不知道领导者的标识符，因此 R_2 中没有处理器可以在 $\sigma_1\sigma_2\sigma_3$ 结束时终止（如同引理 3.6 中的证明）。 □

我们断言，在扩展 $\sigma_1\sigma_2\sigma_3$ 的每一合法调度中，子环 R_2 中各处理器必须在终止前至少接收一条附加消息。这是因为 R_2 中的各处理器，只能通过来自 R_1 的消息学习到领导者的标识符。由于 $\sigma_1\sigma_2\sigma_3$ 中没有消息在 R_1 和 R_2 之间提交，这样各处理器就必须在终止前接收另外的消息。这一论据是基于所有处理器必须学习领导者 id 的假设。

以上论据清楚地表明，在 R 上必须接收附加的 $\Omega\left(\frac{n}{2}\right)$ 条消息。但我们不能就此完成证明，因为以上断言假定 e_p 和 e_q 都是接通的（因调度必须是合法的），这样调度就不是开放的。我们无法预言当 e_p 独自接通时就会接收到所有附加消息，因为算法也可能决定去等待 e_q 上的消息。但我们可以证明只接通 e_p 或 e_q 中的一条边就足够了，仍可强制算法接收到 $\Omega\left(\frac{n}{2}\right)$ 条消息，这在以下的断言中证明。

断言 3.8　*存在一个有限的调度段 σ_4，其中接收到 $\frac{1}{2}\left(\frac{n}{2}-1\right)$ 条消息，因而 $\sigma_1\sigma_2\sigma_3\sigma_4$ 成为一个开放调度，其中 e_p 或 e_q 之一是开放的。*

证明　设 σ_4'' 使 $\sigma_1\sigma_2\sigma_3\sigma_4''$ 成为一个合法调度，这样所有消息在 e_p 或 e_q 上提交且所有处理器终止。如前所述，因为 R_2 中的每一个处理器在终止前必须接收一条消息，所以在 A 终止前 σ_4'' 中至少接收 $n/2$ 条消息。设 σ_4' 是 σ_4'' 的最短前缀，其中接收 $n/2-1$ 条消息。考虑 R 上在 σ_4' 中接收消息的所有处理器，由于 σ_4' 开始于一种“静默的”配置，其中消息只在 e_p 和 e_q 上通过，这些处理器形成 P 和 Q 两个连贯的处理器集合。P 中因 e_p 的接通而包含被唤醒（awakened）的处理器，从而至少包含 p_1 和 p_2 中的一个。同样，Q 中因 e_q 的接通而包含被唤醒的处理器，从而至少包含 q_1 和 q_2 中的一个（见图 3.6）。

因为最多有 $n/2-1$ 个处理器包含在这些集合中，且集合是连贯的，所以可知这两个集合是不相交的。进一步讲，在每一集合中处理器接收的消息数至少为 $\frac{1}{2}\left(\frac{n}{2}-1\right)$。为不失一般性，假设这一集合是 P，即包含 p_1 或 p_2。设 σ_4 是 σ_4' 的子序且只包含 P 中处理器上的事件。因 σ_4' 中不存在 P 中处理器和 Q 中处理器间的通信，故 $\sigma_1\sigma_2\sigma_3\sigma_4$ 就是一个调度。根据假设，在 σ_4 中至少接收到 $\frac{1}{2}\left(\frac{n}{2}-1\right)$ 条消息。另外根据构造，在 e_q 上没有消息提交，因此 $\sigma_1\sigma_2\sigma_3\sigma_4$ 就是所期望的开放调度。 □

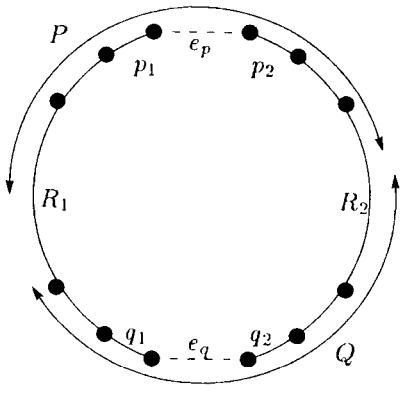

图 3.6　断言 3.8 图示

总结如下，我们从 R_1 和 R_2 上的两个独立调度开始，其中接收了 $2M\left(\frac{n}{2}\right)$ 条消息。然后，

我们强制环进入一种“静默的”配置。最后,强制从“静默的”配置出发,接收$\frac{1}{2}\left(\frac{n}{2}-1\right)$条附加消息,同时保持$e_p$或$e_q$之一开放。这样我们就构造出一个开放调度。其中接收的消息数至少为$2M\left(\frac{n}{2}\right)+\frac{1}{2}\left(\frac{n}{2}-1\right)$。□

3.4 同步环

现在转向同步环中选举领导者的问题,同样将提出消息数的上界和下界。对于上界,我们提出消息数为$O(n)$的两种领导者选举算法。显然,这些算法的消息复杂度是最优的,但是运行时间没有受环规模的任一(单独)函数所限制,且算法是以非常规的方式使用处理器的标识符。对于下界,我们将证明任何只用标识符比较的算法,或受时间限制(即在若干依赖于环规模的时间轮后终止)的算法,需要的消息数至少是$\Omega(n \log n)$。

3.4.1 上界 $O(n)$

第3.3.3节提供的异步环中领导者选举下界$\Omega(n \log n)$的证明,主要是依赖于将消息延迟任意长的周期。我们自然想知道在同步模型中是否可以达到更好的结果,这里消息延迟是固定的。我们将看到,在同步模型中信息的获取,不仅可以通过接收消息,而且可以通过在特定轮中不接收消息而达到。

本节提出同步环中选举领导者的两种算法,两种算法所需要的消息数都是$O(n)$。算法针对单向环,通信按顺时针方向进行。当然,同样的算法也可用于双向环。第一个算法是非一致性的,且要求环中的所有处理器在同一轮启动,正如同步模型中定义的那样。第二个算法是一致性的,各处理器可在不同轮启动,即算法采用的模型要稍弱于标准同步模型。

3.4.1.1 非一致性算法

非一致性算法选举标识符最小的处理器成为领导者。算法按阶段进行,每一阶段包括n轮。在阶段$i(i \geqslant 0)$,如果存在一个标识符为i的处理器,那么它就被选举为领导者,且算法终止。因此,选举的是标识符最小的处理器。

详细描述如下:阶段i包括第$n \cdot i+1, n \cdot i+2, \cdots, n \cdot i+n$轮。在阶段$i$开始时,如果某处理器的标识符为$i$,且它尚未终止,那么该处理器往环中发送一条消息并终止成为领导者。如果处理器的标识符不是i,且在阶段i接收到一条消息,那么它转发该消息并终止成为非领导者。

因为标识符各不相同,显然只有标识符最小的唯一处理器将终止成为领导者。另外,算法中正好发送了n条消息;这些消息是在发现获胜者的阶段中发送的。但是,轮数则是依赖于环中的最小标识符。更准确地说,如果m是最小标识符,则算法经历的轮数为$n \cdot (m+1)$。

注意,以上算法基于前面提到的要求:已知n且同步启动。下一算法则克服了这些限制。

3.4.1.2 一致性算法

下面的领导者选举算法不要求知道环的规模。另外,算法运行模式要稍弱于标准同步模型,其中各处理器不必同时启动算法。更准确地说,一个处理器要么在任一轮上自动醒来,要么在接收到另一处理器的消息时醒来(见练习3.7)。

该一致性算法采用了两个新的想法。首先,在不同处理器上发起的消息以不同速率转

发。准确地说，在标识符 i 的处理器上发起的消息，在接收它的各处理器上延迟 2^i-1 轮，再按顺时针转发给下一处理器。其次，为了克服非同步的启动，加入一个预醒阶段。在此阶段，自动醒来的各处理器往环中发送一条“wake-up”消息，该消息无延迟地转发。在启动算法前接收到 wake-up 消息的处理器，不参与算法，而只作为一个中继，转发或吞没消息。在预醒阶段之后，在参与的处理器集合中选举领导者。

由处理器发送的 wake-up 消息包含处理器的标识符。这一消息在每一轮中以正常速率经过一条边，排除那些在接收到该消息时尚未醒来的处理器。当从标识符为 i 的处理器来的消息到达一个参与的处理器时，该消息开始以 2^i 的延迟速率传送；为实现这种降速，接收到该消息的每个处理器在转发前使之延迟 2^i-1 轮。注意，一旦消息到达某个醒来的处理器，它随后将到达的所有处理器都是醒来的。一条消息在被一个参与的处理器接收前，处于第一阶段(first phase)；在到达一个参与的处理器后，就处于第二阶段(second phase)，并以 2^i 的速率转发。

在整个算法中，各处理器转发消息。但像我们前面介绍的领导者选举算法那样，处理器有时不转发而是吞没消息。在本算法中，吞没消息采用以下规则：

1. 参与的处理器吞没消息，其条件是消息中标识符大于该处理器到目前为止见过的最小标识符，包括它自己的标识符；
2. 中继的处理器吞没消息，其条件是消息中标识符大于该处理器到目前为止见过的最小标识符，不包括它自己的标识符。

伪代码见算法 6。

我们下面证明，在第一个处理器醒来后的 n 轮，只剩下第二阶段的消息，并且领导者将在参与的处理器中选举出来。吞没规则确保只有一个标识符最小的参与处理器，接收到它自己的消息并终止成为领导者。这在引理 3.9 中证明。

对每个 i，$0\leqslant i<n$，设 id_i 是处理器 p_i 的标识符，$\langle id_i\rangle$ 是 p_i 发起的消息。

引理 3.9　*在参与的处理器中，只有标识符最小的处理器可接收到它自己的消息。*

证明　设 p_i 是参与的处理器中标识符最小的处理器(注意，至少有一个处理器会参与到算法中)。显然，无论是参与的或不参与的，没有处理器能够吞没 $\langle id_i\rangle$。

进一步说，由于 $\langle id_i\rangle$ 在各处理器上最多延迟 2^{id_i} 轮，所以 p_i 最终会接收到它自己的消息。

采用反证法，假设某一其他处理器 p_j，$j\neq i$，也接收到自己的消息 $\langle id_j\rangle$。这样，$\langle id_j\rangle$ 就必定经过了环中所有的处理器，包括 p_i。但 $id_i<id_j$，因 p_i 是参与的处理器，它不会转发 $\langle id_j\rangle$，故矛盾。　□

以上引理表明，仅有一个处理器可接收到它自己的消息。因而该处理器将是唯一声明自己是领导者的，于是证明了算法的正确性。我们现在分析在算法的合法执行中所发送的消息数。

为计算在算法的合法执行中所发送的消息数，我们将其划分为三类：

1. 第一阶段消息；
2. 在最终领导者消息进入第二阶段前发送的第二阶段消息；
3. 在最终领导者消息进入第二阶段后发送的第二阶段消息。

算法 6 同步领导者选举:处理器 p_i 的代码,$0 \leqslant i < n$。

```
Initially waiting is empty and status is asleep

1:  let R be the set of messages received in this computation event
2:  S := ∅                                          // the messages to be sent

3:  if status = asleep then
4:      if R is empty then                          // woke up spontaneously
5:          status := participating
6:          min := id
7:          add ⟨id,1⟩ to S                         // first phase message
8:      else
9:          status := relay
10:         min := ∞

9:  for each ⟨m,h⟩ in R do
10:     if m < min then
11:         become not elected
12:         min := m
13:         if (status = relay) and (h = 1) then            // m stays first phase
14:             add ⟨m,h⟩ to S
15:         else                                    // m is/becomes second phase
16:             add ⟨m,2⟩ to waiting tagged with current round number
17:     elseif m = id then become elected
                                    // if m > min then message is swallowed

18: for each ⟨m,2⟩ in waiting do
19:     if ⟨m,2⟩ was received 2^m − 1 rounds ago then
20:         remove ⟨m⟩ from waiting and add to S

21: send S to left
```

引理 3.10 第一类消息总数最多为 n。

证明 通过证明各处理器最多转发一条第一阶段的消息,从而证明该引理。

采用反证法,假设某一处理器 p_i 在第一阶段转发两条消息:来自 p_j 的$\langle id_j \rangle$和来自 p_k 的$\langle id_k \rangle$。为不失一般性,假设按顺时针方向,p_j 比 p_k 离 p_i 更近。这样,$\langle id_k \rangle$在到达 p_i 前必须经过 p_j。如果$\langle id_k \rangle$在 p_j 醒来并发送$\langle id_j \rangle$之后到达 p_j,那么$\langle id_k \rangle$就将以 2^{id_k} 的速率继续成为第二阶段的消息;否则,p_j 就不会参与且$\langle id_j \rangle$不会发送。因此,要么$\langle id_k \rangle$到达 p_i 成为第二阶段的消息,要么$\langle id_j \rangle$不会发送,从而矛盾。 □

设 r 是某处理器启动算法执行的第一轮,p_i 是这种处理器之一。为了限制第二类中的消息数,我们首先证明在第一个处理器启动算法执行 n 轮之后,所有消息处于第二阶段。

引理 3.11 若 p_j 与 p_i 的距离(顺时针方向)为 k,则 p_j 接收到第一阶段消息不迟于 $r+k$ 轮。

证明 采用对 k 的归纳法来证明。基本步骤 $k=1$ 是显然的。因为 p_i 的邻居在 $r+1$ 轮接收

p_i 的消息。对于递归步骤，假定与 p_i 距离(顺时针方向)为 $k-1$ 的处理器，在不迟于 $r+k-1$ 轮接收到第一阶段消息。如果该处理器在接收到第一阶段消息时已经醒来，那它就已经发送一条第一阶段消息给其邻居 p_j；否则，它在 $r+k$ 轮转发第一阶段消息给 p_j。 □

引理 3.12　*第二类中的消息总数最多为 n。*

证明　引理 3.10 证明了在每条边上最多发送一条第一阶段的消息。由于在第 $r+n$ 轮，每条边上发送了一条第一阶段的消息，从而在第 $r+n$ 轮后已没有了第一阶段的消息。根据引理 3.11，最终领导者的消息最多在算法发送第一条消息后的第 n 轮进入第二阶段。这样，第二类中的消息只在第一个处理器醒来后的 n 轮中发送。

第二阶段消息$\langle i\rangle$在转发前延迟 2^i-1 轮，这样$\langle i\rangle$在该类中最多发送$\frac{n}{2^i}$次。因为包含较小标识符的消息转发更频繁些，所以当所有处理器都参与，且各标识符尽可能小时(即 0，1，…，$n-1$)，将获得最大消息数。注意，最终领导者的第二阶段消息(在此为 0)不在此类中计算，因此在第二类中的消息数最多为$\sum_{i=1}^{n-1}\frac{n}{2^i}\leqslant n$。 □

设 p_i 是标识符最小的处理器；各处理器在转发$\langle id_i\rangle$之后不再转发消息。一旦$\langle id_i\rangle$返回到 p_i，则环中所有处理器就都已转发过它，于是就有了下面的引理：

引理 3.13　*在返回到后无消息转发。*

引理 3.14　*第三类中的消息总数最多为 $2n$。*

证明　设 p_i 是最终领导者，p_j 是另一个参与的处理器。根据引理 3.9，$id_i < id_j$。根据引理 3.13，在 p_i 接收到自己的消息后，环中没有消息。因为$\langle id_i\rangle$在各处理器上最多延迟 2^{id_i} 轮，$\langle id_i\rangle$返回到 p_i 最多需要 $n\cdot 2^{id_i}$ 轮，因而第三类中的消息仅在 $n\cdot 2^{id_i}$ 轮期间发送。在此期间，$\langle id_j\rangle$转发次数最多为

$$\frac{1}{2^{id_j}}\cdot n\cdot 2^{id_i}=n\cdot 2^{id_i-id_j}$$

因此，在第三类中传送的消息总数最多为

$$\sum_{j=0}^{n-1}\frac{n}{2^{id_j-id_i}}$$

根据引理 3.12 证明中的相同论据，这个总数小于或等于下值：

$$\sum_{k=0}^{n-1}\frac{n}{2^k}\leq 2n$$

引理 3.10、3.12 和 3.14 表明：

定理 3.15　*存在一种同步领导者选举算法，其消息复杂度最多为 $4n$。*

现在考虑算法的时间复杂度。根据引理 3.13，计算是在所选举的领导者接收到它自己的消息时结束。这在第一个处理器启动算法执行后的 $O(n2^i)$轮中发生，其中 i 是所选举的领导者的标识符。

3.4.2 受限算法的下界 $\Omega(n \log n)$

在第 3.4.1 节,我们介绍了同步环中选举领导者的两个算法,它们在最坏情况下的消息复杂度为 $O(n)$。这两个算法均有两个不理想的属性:第一,它们以非标准的方式使用标识符(确定消息应延迟多长时间);第二,也是更重要的,在每一合法执行中的轮数依赖于处理器的标识符。不理想的原因是处理器的标识符相对于 n 来说可能很大。

在本节,我们证明上述两个属性对任何基于消息的有效算法是固有的。特别地,如果一个算法只将标识符用于比较,它需要的消息数为 $\Omega(n \log n)$。然后,我们采用递归法,证明如果算法只使用有限的轮数,且独立于标识符,那么它需要的消息数也是 $\Omega(n \log n)$。

同步下界不能从异步下界(定理 3.5)中推导出来,因为第 3.4.1 节提出的算法指出,为保持同步下界需要附加的假设。同步下界对于非一致性算法也成立,而异步下界只对一致性算法成立。有趣的是,从同步中导出的异步的反向推导,却是正确的,且提供了非一致性算法的异步下界,这将在练习 3.11 中研究。

3.4.2.1 基于比较的算法

在本节,我们正式定义"基于比较的算法"的概念。

为了研究下界,我们假定所有处理器在同一轮开始执行。

回顾一下,从最小的标识符开始,按顺时针次序,列出各处理器的标识符,就可确定一个环。注意,在同步模型中,算法的合法执行完全由初始配置来定义,因为没有消息延迟的选择或处理器步调的相对次序。而系统的初始配置,又完全是由环来定义的,即根据以上规则列出处理器的标识符表。当算法选定后,我们将环 R 确定的合法执行表示为 $exec(R)$。

环 R_1 中的 p_i 和环 R_2 中的 p_j 两个处理器,当在各自的环描述中具有相同位置时,就是相匹配的(matching)。注意,相匹配的处理器在各自环中与标识符最小的处理器的距离相等。

从直观上看,如果一个算法在各种环中的行为相同,这些环具有相同次序的标识符模式,那么该算法就是基于比较的算法。其形式化描述如下,两个环($x_0, \cdots, x_{n-1}$ 和 $y_0, \cdots, y_{n-1}$)如果对每个 i 和 j,当且仅当 $y_i < y_j$ 时 $x_i < x_j$,则是次序等价的(order equivalent)。回顾一下,环中处理器 p_i 的 k-邻域,是处理器 $p_{i-k}, \cdots, p_{i-1}, p_i, p_{i+1}, \cdots, p_{i+k}$(所有索引按模 n 计算)共 $2k+1$ 个标识符的序列。可用显而易见的方式将次序等价概念扩展到 k-邻域。

现在定义"行为相同"的含义。从直观上看,我们可以断言两个次序等价的环 R_1 和 R_2 上的合法执行中,发送的消息和所做的判定都是相同的。但一般来说,算法发送的消息包含处理器的标识符;这样 R_1 上发送的消息将不同于 R_2 上发送的消息。然而根据目标,我们将集中于消息的模式,即消息在什么时候、什么地方发送,而不是它们的内容和所做的抉择。特别地,考虑两个执行 α_1 和 α_2 及两个处理器 p_i 和 p_j,当以下条件满足时,我们称 α_1 中 p_i 的行为,与 α_2 中 p_j 的行为,在第 k 轮是相似的(similar):

1. 在 α_1 中的第 k 轮 p_i 终止成为领导者,当且仅当 p_j 在 α_2 中的第 k 轮作为领导者终止。
2. 在 α_1 中的第 k 轮 p_i 向左(右)邻发送消息,当且仅当 p_j 在 α_2 中的第 k 轮向左(右)邻发送消息。

现在可以形式化地定义基于比较的算法:我们称在 α_1 中 p_i 的行为与在 α_2 中 p_j 的行为是相似的,如果它们在所有轮 $k \geqslant 0$ 都是相似的。

定义 3.2　一个算法是基于比较的，是指对于每一对次序等价的环 R_1 和 R_2，在 $exec(R_1)$ 和 $exec(R_2)$ 中的每一对相匹配的处理器都有相似的行为。

3.4.2.2　基于比较的算法下界

设 A 是一种基于比较的领导者选举算法。证明中考虑在次序模式上非常对称的环，即在环中存在许多次序等价的邻域。从直观上看，只要两个处理器有次序等价的邻域，它们在 A 下的行为相等。我们通过在非常对称的环上执行 A 来推导下界，并论证如果一个处理器在某一特定轮中发送一条消息，则具有次序等价邻域的所有处理器也在该轮发送一条消息。

证明中极重要的一点是：将处理器获取到信息的轮，与没有获取到信息的轮区分开来。回顾一下，在同步环中，一个处理器甚至可以在没有接收到消息时也可获取到信息。比如，在第 3.4.1 节的非一致性算法中，在轮 $1 \sim n$ 中实际上没有接收到消息，这表明环中没有标识符为 0 的处理器。在下面的证明中，关键是对以下情况的观察：对处理器 p_i 来说，在某轮 r 中消息的不存在是有用的，其前提是仅当该轮在另一个次序等价的环中接收到了消息。例如，在非一致性算法中，若环中某一处理器的标识符为 0，则在轮 $1, \cdots, n$ 会接收到消息。因此，如某轮在任何次序等价的环中无消息发送，那它就是没有用的。而有用的轮被称为是活跃的(active)，其定义如下：

定义 3.3　在环 R 中，如果某处理器在执行的第 r 轮中发送消息，则称该轮 r 在该执行中是活跃的。当 R 在上下文中已知时，用 r_k 表示第 k 个活跃轮[①]。

根据定义，基于比较的算法在次序等价的环上产生相似的行为。这意味着，对于次序等价的环 R_1 和 R_2，某一轮当且仅当在 $exec(R_2)$ 中活跃时，它在 $exec(R_1)$ 中活跃。

由于消息中的信息在 k 轮中只经过环中的 k 个处理器，因而在第 k 轮后，处理器的状态仅依赖于它的 k-邻域。然而，我们还有一个更强的属性，即在第 k 个活跃轮后，处理器的状态仅依赖于它的 k-邻域。这可根据信息只在活跃轮获取而直观地得出，见引理 3.16 中的形式化证明。注意，该引理不要求各处理器是相匹配的(否则的话，该断言可由定义直接得到)，但要求它们的邻域是相同的。这一引理需要假设两个环是次序等价的，其原因是要保证所考虑的两个执行具有相同的活跃轮集合，这样 r_k 就是良定义的(well-defined)。

引理 3.16　设 R_1 和 R_2 是次序等价的环，且设 R_1 中的 p_i 和 R_2 中的 p_j 是具有相同 k-邻域的两个处理器。那么 p_i 在 $exec(R_1)$ 的第 $1 \sim r_k$ 轮经历的转移序列，与 p_j 在 $exec(R_2)$ 的第 $1 \sim r_k$ 轮经历的转移序列相同。

证明　简单地说，这一证明是要说明在第 k 个活跃轮后，处理器只可能从与它距离最多为 k 的范围内了解其他处理器。

下面将用基于 k 的归纳法进行形式化的证明。对于基本步骤 $k = 0$，注意到具有相等 0-邻域的两个处理器，具有相同的标识符，因而它们处于相同的状态。

对于归纳步骤，假设具有相同 $(k-1)$-邻域的每两个处理器，在第 $(k-1)$ 个活跃轮后处于相同的状态。由于 p_i 和 p_j 具有相同的 k-邻域，它们也有相同的 $(k-1)$-邻域。因此，根据递

① 回顾一下，一旦环确定，全部合法执行就确定，因为系统是同步的。

归假设,p_i 和 p_j 在第$(k-1)$个活跃轮后处于相同状态。进一步讲,它们各自的邻居具有相同的$(k-1)$-邻域。所以,根据递归假设,它们各自的邻居在第$(k-1)$个活跃轮后处于相同状态。

在第$(k-1)$个活跃轮与第 k 个活跃轮之间的轮(如果存在的话),没有处理器接收到任何消息,从而 p_i 和 p_j 相互之间保持相同的状态,它们各自的邻居也一样(注意,p_i 在非活跃轮可能改变其状态,但因 p_j 具有相同的转移函数,故也进行相同的状态转换)。在第 k 个活跃轮,如果 p_i 和 p_j 都未接收到消息,那么它们在轮结束时处于相同状态。如果 p_i 从其右邻接收到一条消息,那么 p_j 也会从其右邻接收到一条同样的消息,因为邻居都处于相同状态;从左邻接收消息也一样。因此,p_i 和 p_j 在第 k 个活跃轮结束时处于相同状态,证明完毕。 □

引理 3.17 将上述断言从具有相同 k-邻域的处理器,扩展到具有次序等价 k-邻域的处理器。它依赖于 A 是"基于比较的"这一事实。进一步讲,它要求环 R 是间隔的(spaced),其直观含义是在 R 的每两个标识符之间有 n(n 是环的规模)个未使用的标识符。在形式上,规模为 n 的环是间隔的,是指对环中任一标识符 x,标识符 $x-1 \sim x-n$ 都不在环中。

引理 3.17 *设 R 是一个间隔环,p_i 和 p_j 是 R 中具有次序等价 k-邻域的两个处理器,那么 p_i 和 p_j 在 $exec(R)$ 的第 $1 \sim r_k$ 轮有相似的行为。*

证明 我们构造另一个环 R',它满足以下条件:

- p_j 的 k-邻域与 R 中 p_i 的 k-邻域相同;
- R' 中的标识符唯一;
- R' 与 R 次序等价,R' 中 p_j 与 R 中 p_j 相匹配。

因 R 是间隔的,故 R' 可构造出(见图 3.7 中实例)

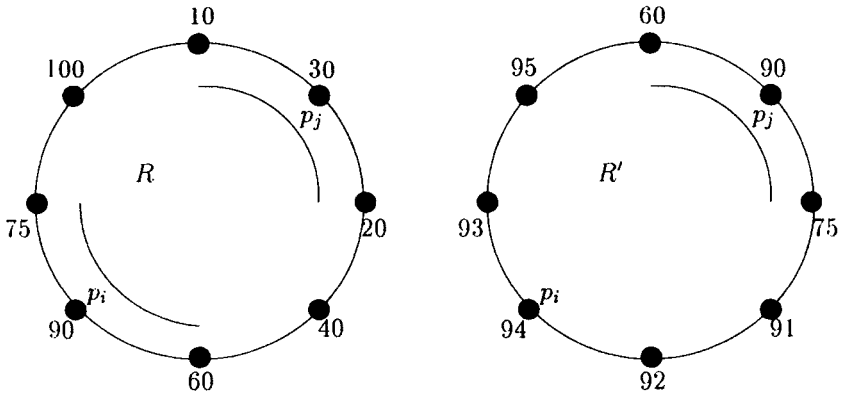

图 3.7 引理 3.17 证明中的实例;$k=1$ 且 $n=8$

根据引理 3.16,p_i 在 $exec(R)$中第 $1 \sim r_k$ 轮经历的转移序列,与 p_j 在 $exec(R')$中第 $1 \sim r_k$ 轮经历的转移序列相同,这样 p_i 在 $exec(R)$中第 $1 \sim r_k$ 轮的行为与 p_j 在 $exec(R')$中第 $1 \sim r_k$ 轮的行为相似。由于算法是基于比较的,且 R' 中 p_j 与 R 中 p_j 是相匹配的,p_j 在 $exec(R')$中第 $1 \sim r_k$ 轮的行为就与 p_j 在 $exec(R)$中第 $1 \sim r_k$ 轮的行为相似。因此,在 $exec(R)$的第 $1 \sim r_k$ 轮,p_i 的行为与 p_j 的行为是相似的。 □

我们现在可以证明主定理：

定理 3.18　对每个 $n \geqslant 8$（n 是 2 的幂），存在一个规模为 n 的环 S_n，对每个同步的、基于比较的领导者选举算法 A，在 S_n 上 A 的合法执行中发送的消息数为 $\Omega(n \log n)$。

证明　确定任一这种算法 A。证明的关键是构造一个非常对称的环 S_n，其中多个处理器有多个次序等价的邻域。S_n 的构造分为两步。

首先，定义 n 个处理器的环 R_n^{rev} 如下：对每个 i，$0 \leqslant i < n$，设 p_i 的标识符为 $\text{rev}(i)$，这里 $\text{rev}(i)$ 为整数，它采用 $\log n$ 位的二进制表示，是 i 的二进制表示的反码（见图 3.8 中的具体实例，其 $n = 8$）。考虑将 R_n^{rev} 划分为长度为 j 的连续片段（j 是 2 的幂），可以证明所有这些片段是次序等价的（见练习 3.9）。

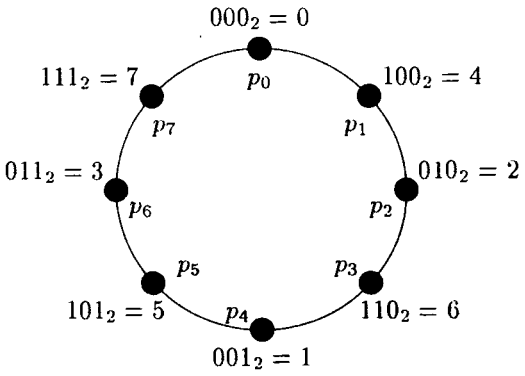

图 3.8　环图 R_8^{rev}

S_n 是 R_n^{rev} 的一个间隔版本，通过将 R_n^{rev} 中的每一个标识符乘以 $n+1$，然后再加 n 而获得。这些变化不改变各片段的次序等价。

引理 3.19 对 S_n 中存在多少给定规模的次序等价邻域进行定量。这一结果稍后在引理 3.20 中用来证明算法活跃轮数的下界，在引理 3.21 中用来证明每一活跃轮中发送的消息数的下界。通过结合后两种下界而获得所期望的界限 $\Omega(n \log n)$。

引理 3.19　对所有 $k < n/8$，且对 S_n 的所有 k-邻域 N，存在超过 $\frac{n}{2(2k+1)}$ 个 k-邻域是与 N 次序等价的（包括 N 本身）。

证明　N 由含 $2k+1$ 个标识符的序列所组成。设 j 是大于 $2k+1$ 的 2 的幂中最小的。将 S_n 划分为 n/j 个连续片段，这样一个片段就完全包含了 N。根据 S_n 的构造，所有这些片段都是次序等价的。因此至少有 n/j 个邻域是与 N 次序等价的。因 $j < 2(2k+1)$，故与 N 次序等价的邻域数大于 $\frac{n}{2(2k+1)}$。□

引理 3.20　$exec(S_n)$ 中的活跃轮数至少是 $n/8$。

证明　设 T 是活跃轮数。按反证法假设 $T < n/8$。设处理器 p_i 在 $exec(S_n)$ 中被选举为领导者。根据引理 3.19，有超过 $\frac{n}{2(2T+1)}$ 的 T-邻域与 p_i 的 T-邻域是次序等价的。根据对 T 的假设：

$$\frac{n}{2(2T+1)} > \frac{n}{2(2n/8+1)} = \frac{2n}{n+4}$$

因 $n \geqslant 8$，故 $\frac{2n}{n+4} > 1$。这样除 p_i 外，存在某处理器 p_j，它的 T-邻域与 p_i 的 T-邻域是次序等价的。根据引理 3.17，p_j 也被选举，这与 A 的正确性假设矛盾。□

引理 3.21　对每个 k，$1 \leqslant k \leqslant n/8$，在 $exec(S_n)$ 的第 k 个活跃轮中发送的消息数至少是 $\frac{n}{2(2k+1)}$。

证明 考虑第 k 个活跃轮。由于它是活跃的,所以至少有一个处理器(如 p_i)发送一条消息。根据引理 3.19,有超过 $\frac{n}{2(2k+1)}$ 个处理器,它的 k-邻域与 p_i 的 k-邻域是次序等价的。根据引理 3.17,这些处理器中的每一个在第 k 个活跃轮也都发送一条消息。 □

现在可以完成主定理的证明。根据引理 3.20 和引理 3.21,在 $exec(S_n)$ 中发送的消息总数至少为

$$\sum_{k=1}^{n/8} \frac{n}{2(2k+1)} \geqslant \frac{n}{6} \sum_{k=1}^{n/8} \frac{1}{k} > \frac{n}{6} \ln \frac{n}{8}$$

也就是 $\Omega(n \log n)$。 □

注意,为使这一定理成立,对取自自然数的每一标识符集合,算法不需要是基于比较的,而只需对取自 $\{0,1,\cdots,n^2+2n-1\}$ 的标识符集合是基于比较的。其原因是,S_n 中最大的标识符是 $n^2+n-1=(n+1)\cdot \mathrm{rev}(n-1)+n$(回顾一下,$n$ 是 2 的幂,且 $n-1$ 的二进制表示是 1 的序列)。我们要求算法对于 0 和 n^2+2n-1 之间的所有标识符是基于比较的,而不只是对于 S_n 中已有的标识符,因为引理 3.17 的证明使用了一个事实根据,即算法对于范围从小于 S_n 中最小,到大于 S_n 中最大的所有标识符,都是基于比较的。

3.4.2.3 限时算法的下界

下面的定义不允许算法的运行时间依赖于标识符:它要求运行时间对各种环的规模都是受限的,而标识符不受限,因为它们取自自然数集合。

定义 3.4 *同步算法 A 是限时(time-bounded)的,是指对每个 n,在规模为 n 的、标识符取自自然数的各种环上,A 在最坏情况下的运行时间是有限的。*

下面通过对基于比较算法的归纳,证明限时算法的下界 $\Omega(n \log n)$。我们首先说明如何将限时算法映射到基于比较的算法;然后使用基于比较算法的消息数下界,来获得限时算法所发送的消息数下界。因为基于比较算法的下界只针对 n 是 2 的幂进行了描述,这里同样采用这一方法,实际上这一下界对所有的 n 值都是成立的(见本章注释)。

为从限时算法映射到基于比较的算法,我们需要一个定义来描述算法在受限时间内的行为。

定义 3.5 *一个同步算法 A,对环规模为 n、在标识符集合 S 上是基于 t-比较的(t-comparison based),是指对于规模为 n 的任意两个次序等价的环 R_1 和 R_2,每一对相匹配的处理器在 $exec(R_1)$ 和 $exec(R_2)$ 的第 $1 \sim t$ 轮中具有相似的行为。*

从直观上看,在 S 上基于 r-比较的算法,只要标识符从 S 中选取,在前 r 轮中与基于比较的算法是行为相似的。如果算法在 r 轮内终止,那么就与 S 上基于比较的算法在所有轮都相同。

第一步是要说明在输入的子集上,每一个限时算法的行为类似于基于比较的算法,前提是输入集足够大。为此,我们使用 Ramsey 定理的有限版本。通俗而言,该定理表述的是:如果对一个规模很大的元素集合,用 t 种颜色之一给大小为 k 的每个子集涂色,那么就能找到某个大小为 ℓ 的子集,其所有大小为 k 的子集具有相同的颜色。如果把涂色看成是划分等

价类(大小为 k 的两个子集若具有相同颜色,则属于相同的等价类),则该定理说的就是存在一个大小为 ℓ 的集合,其所有大小为 k 的子集处在相同的等价类中。稍后,如果各环中相匹配处理器的行为相似,我们将给各环涂上相同的颜色。

为了完整性,我们重复 Ramsey 定理如下:

Ramsey 定理(有限版本) 对所有整数 k, ℓ 和 t,存在整数 $f(k, \ell, t)$,对每个大小至少为 $f(k, \ell, t)$的集合 S 及每个 S 的 k-子集的 t – 涂色,S 的某个 ℓ-子集的所有 k-子集将有相同的颜色。

在引理 3.22 中,我们使用 Ramsey 定理将任一限时算法映射到一个基于比较的算法。

引理 3.22 *设 A 是运行时间为 $r(n)$的一个同步限时算法,那么对每个 n,存在 n^2+2n 个标识符的集合 C_n,对规模为 n 的环,A 是 C_n 上基于 $r(n)$-比较的。*

证明 固定 n。设 Y 和 Z 表示 N(自然数)的任意两个 n-子集。如果对于每一对次序等价的环——标识符来自 Y 的 R_1 和标识符来自 Z 的 R_2,在 $exec(R_1)$和 $exec(R_2)$的第 $1 \sim t(n)$轮,相匹配的处理器具有相似的行为,那么称 Y 和 Z 是等价子集(equivalent subsets)。这一定义将 N 的 n-子集划分为有限多的等价类,因为术语"相似行为"只是指消息和终止状态的有或无。我们给 N 的 n-子集涂色,这样两个 n-子集具有相同颜色的条件是当且仅当它们处在相同的等价类中。

根据 Ramsey 定理,如果认为 t 是等价类(颜色)的数目,ℓ 是 n^2+2n,k 是 n,那么由于 N 是无限的,所以存在基数为 n^2+2n 的 N 的子集 C_n,使 C_n 的所有 n-子集属于相同的等价类。

我们断言,对于规模为 n 的环,A 是 C_n 上基于 $r(n)$-比较的算法。考虑两个次序等价的环 R_1 和 R_2,规模为 n,具有取自 C_n 的标识符。设 Y 是 R_1 中的标识符集合,Z 是 R_2 中的标识符集合。Y 和 Z 是 C_n 的 n-子集;那么它们属于相同的等价类。这样,相匹配的处理器在 $exec(R_1)$和 $exec(R_2)$的第 $1 \sim r(n)$轮具有相似的行为。所以,对于规模为 n 的环,A 是基于 $r(n)$-比较的算法。 □

定理 3.18 表明,每个基于比较的算法具有最坏消息复杂度 $\Omega(n \log n)$。我们现在不能直接应用这一定理,因为我们仅仅证明了限时算法 A 在特定的 id 集合上是基于比较的,而不是在所有的 id 集合上。然而,我们将用 A 来设计另一个算法 A',具有与 A 相同的消息复杂度,在规模为 n 的环上,它是基于比较的,其 id 取自集合$\{0,1,\cdots,n^2+2n-1\}$。如定理 3.18 证明后所讨论的那样,这足以证明 A'的消息复杂度是 $\Omega(n \log n)$。因而 A 的消息复杂度是$\Omega(n \log n)$。

定理 3.23 *对每一个同步的限时领导者选举算法 A,以及每一个 $n \geqslant 8$(n 是 2 的幂),存在一个规模为 n 的环 R,使 A 在 R 上合法的执行中发送的消息数为 $\Omega(n \log n)$。*

证明 给定算法 A 满足定理的假设,运行时间为 $r(n)$。固定 n;设 C_n 为引理 3.22 允许的标识符集合,$c_0, c_1, \cdots, c_{n^2+2n-1}$是按递增顺序排列的 C_n 中的元素。

在规模为 n、标识符取自集合$\{0,1,\cdots,n^2+2n-1\}$的环上,定义一个基于比较的算法 A',它具有与 A 相同的时间复杂度和消息复杂度。在算法 A'中,标识符为 i 的处理器执行算法 A,如同具有标识符 c_i。因为 A 对于规模为 n 的环,是 C_n 上基于 $r(n)$-比较的,而且因为 A 在 $r(n)$轮内终止,从而 A'对于规模为 n、标识符取自集合$\{0,1,\cdots,n^2+2n-1\}$的环,是基于比较的。

根据定理 3.18,存在一个规模为 n、标识符取自集合$\{0,1,\cdots,n^2+2n-1\}$的环,其中 A' 发送的消息数为 $\Omega(n \log n)$。根据 A' 的构造,在规模为 n、标识符取自 C_n 的环中,存在一个 A 的执行,它发送相同的消息数,从而证明了定理。 □

练习

3.1 证明异步环系统中不存在匿名的领导者选举算法。

3.2 证明同步环系统中不存在匿名的、一致性的领导者选举算法。

3.3 若同步环中仅有一个处理器具有与其他处理器相同的标识符,那么是否存在领导者选举算法?给出一个算法或证明不可能性结论。

3.4 考虑异步环中的下列领导者选举算法:每个处理器发送其标识符给它的右邻;每个处理器仅在消息所包含的标识符大于它自己的标识符时转发该消息(给它的右邻)。
证明本算法发送的平均消息数为 $O(n \log n)$,假设标识符是均匀分布的整数。

3.5 在第 3.3.3 节,我们已经知道在异步环中选举领导者所需要的消息数下界为 $\Omega(n \log n)$。该下界的证明依赖于两个附加属性:(a)标识符最大的处理器被选举;(b)所有处理器必须知道被选举的领导者的标识符。
证明当这两个要求不满足时,该下界仍然成立。

3.6 把定理 3.5 扩展到 n 不是 2 的整数次幂的情况。
提示:考虑最大的 $n' < n$,它是 2 的整数次幂,对 n' 证明该定理。

3.7 修改同步消息传递系统的形式化模型,描述第 3.4.1 节的非同步启动模型,即说明执行和合法执行必须满足的条件。

3.8 证明引理 3.17 的证明中次序等价的环 R' 总是可以构造出的。

3.9 回顾定理 3.18 的证明中的环 R_n^{rev}。对于将 R_n^{rev} 划分成 n/j 个连续片段的每一种划分,j 是 2 的幂,证明所有这些片段是次序等价的。

3.10 考虑一个匿名环,其中各处理器用二进制输入来启动。
1. 证明不存在一致性的同步算法,用来计算各输入位的与(AND)。
提示:利用反证法假设这种算法存在,并考虑算法在全 1 的环上执行;然后将该环嵌入到有单个 0 的更大的环中。
2. 提出一个计算 AND 的异步(非一致性)算法;该算法在最坏情况下发送的消息数为 $O(n^2)$。
3. 证明对于任何计算 AND 的异步算法,消息复杂度的下界是 $\Omega(n^2)$。
4. 提出一个计算 AND 的同步算法;算法在最坏情况下发送的消息数应为 $O(n)$。

3.11 从同步模型的下界,推导出异步通信模型中领导者选举所需要的消息数下界 $\Omega(n \log n)$。在异步模型中,证明不应依赖于算法是基于比较的或限时的。

本章注释

本章对环拓扑结构的消息传递系统中的领导者选举问题进行了深入研究。环网络之所以吸引了如此多的研究,是因为它们的行为易于描述;而且从环网络推导出的下界,可应用于

具有任意拓扑结构的网络算法设计。另外，环对应有令牌环网[18]。

我们首先证明了在异步环中选择领导者的不可能性(定理 3.2)；这一结论是由 Angluin 证明的[17]。在第 14 章中，我们将描述如何使用随机化来克服这一不可能性。

随后，我们提出了异步环中的领导者选举算法。异步环中领导者选举的 $O(n^2)$算法(在第 3.3.1 节中介绍)是基于 LeLann 的算法[165](他是第一个研究领导者选举问题的人)，以及 Chang and Roberts 对该算法的优化[70]。本算法可以看做是算法 4 的一种特例。Chang and Roberts 还证明了对所有可能的输入，该算法的平均消息复杂度是 $O(n \log n)$(练习 3.4)。

异步环中领导者选举的 $O(n \log n)$算法是由 Hirschberg 和 Sinclair 首先提出的[139]；这是在第 3.3.2 节中介绍的算法。随后，环中领导者选举问题在很多论文中被研究，这里不再一一列出。当前已知的最好算法是由 Higham 和 Przytycka 提出来的[137]，其消息复杂度是 $1.271\ n \log n + O(n)$。

Hirschberg 和 Sinclair 算法假设环是双向的；单向情况下的 $O(n \log n)$算法是由 Dolev、Klawe 和 Rodeh[95]，以及 Peterson[212]提出来的。

定向的问题由 Attiya、Snir 和 Warmuth 进行了深入讨论[33]；他们的论文中有练习 3.10 的答案。

本章中，很大部分是在讨论环中选举领导者所需要的消息数下界。异步情况下的下界是由 Burns 提出的[61]，这一下界只适用于一致性的算法。异步环中领导者选举的平均消息复杂度下界 $\Omega(n \log n)$，是由 Pachl、Korach 和 Rotem 提出的[205]。在该下界中，对所有特定规模的环进行了平均，因此，该下界可适用于非一致性算法。

线性算法及同步情况下的下界，在第 3.4 节中做了介绍，它来源于 Frederickson 和 Lynch 的论文[111]；我们对基于比较的算法的形式化处理，与他们的稍有不同。规模为 n 的对称环的构造(其中 n 不是 2 的整数次幂)，是在[33,111]中出现的。练习 3.11 是根据 Eli Gafni 的观察而来的。