分 布 式 系 统

汪 铮

康 奈 尔 大 学

2021 年 3 月 4 日

摘　要

在计算机科学飞速发展的今天，大数据、云计算、人工智能、区块链等技术层出不穷，而对分布式系统的讨论则是掌握以上新技术不可或缺的基础。本书主要研究分布式系统的理论模型，并利用模型讨论系统全序和全局谓词赋值、一致性、状态机复制以及共识等问题，以及关于这些问题的定理和算法。

关键词：**分布式系统、可靠性、状态机复制、共识问题、拜占廷容错**

前　言

分布式系统是一个方兴未艾的领域。对分布式系统理论的研究，包括对领导选举、同步共识、故障侦测器等的研究，从 1970 年代就开始了。研究这些问题，一半是纯理论，一半是由于数据库研究的需要，比如研究原子化确认问题等等。这个阶段最引人注目的是 1986 年 Fisher 等人证明的 FLP 定理，它阐明了在异步不可靠系统中实现共识的不可能性。这个定理在计算机科学中的地位，可以与物理学中的“热力学第二定律”相媲美。

在 2000 年左右，Leslie Lamport 这位分布式系统领域的爱因斯坦写了一篇名为《兼职国会》的论文，借着在希腊小岛 Paxos 上曾经出现过的某政治系统的由头，提出了 Paxos 这个在异步系统中近似实现共识的算法。此后随着计算机性能的进一步提高，互联网的建立，网上购物、社交媒体、大数据、云计算、人工智能、加密货币等应用的兴起，对大量计算机通过网络方式连接提供服务的需求从来没有像当今那样旺盛。在这些新技术、新业态当中，分布式系统的身影无处不在：Paxos 算法、Raft 算法这些共识算法、Gossip 协议、一致哈希算法等分布式系统算法，是云计算基础服务（比如 Zookeeper）的核心。谷歌、脸书、微软、亚马逊、阿里巴巴、腾讯等公司的核心架构，无不是一个保证高可靠、高性能的分布式系统。比特币、以太网等非许可型加密货币以及 Hyperledger Fabric 等许可型加密货币，无不应用了分布式系统理论给出的结论，有的还运用了拜占廷容错理论。可以说，当前是一个分布式系统爆炸式发展的时代。

本书的内容，主要借鉴了康奈尔大学 Lorenzo Alvisi 教授的分布式系统课讲义，介绍了分布式系统理论。本书首先建立了分布式系统的理论模型，然后用理论模型阐述了系统全序、全局谓词赋值、状态机复制、共识等问题，介绍了 FLP 定理、Paxos、PBFT 等算法，并探究了向量钟、故障侦测器等有用的理论模型。希望读者通过学习分布式系统的理论，能对当今世界飞速发展的计算机科学应用有更深入的了解。

目　录

一、始于足下——理论模型

（一）分布式系统及其设计目标

从 1945 年第一台电子计算机 ENIAC 被制造出来，到 1960 年代操作系统原型的出现，再到 1970 年代的“摩尔定律”（指摩尔提出的一块同等大小芯片上的二极管数目每两年翻一番的经验规律），计算机在 20 世纪里体积越来越小，计算速度越来越快，存储空间越来越大，性能也越来越强。计算机硬件的飞速发展，相应地意味着程序员对计算机软件性能的提升动机不大：根据摩尔定律，一个程序员在 1970 年写下的一块代码，不做任何改动，在 1972 年的硬件上运行，性能就能翻一番。既然硬件能保证我代码的性能会不断地提升，我为什么要费尽心机去思考新算法，优化代码本身呢？当时的计算机程序主要是单机程序，对计算机软件优化，包括对并行和并发的需求，并没有那么大。

然而，从 1990 年代开始，“摩尔定律”逐渐失效了：由于芯片上二极管的数目趋近于物理瓶颈，对单块硬件性能成指数增长的期待愈发成为不切实际的幻想。硬件厂商发现，既然单个计算单元的性能无法获得质的提升，那我就在一块硬件上提供多个计算单元——瞬间，多核 CPU 成为时代的潮流。二核、四核、八核等的 CPU 相继出现。很快，硬件厂家们发现，更多的 CPU 核，并不一定能带来更高的性能——功率成为了限制普通 CPU 核数无线增长的障碍。同时，计算机网络开始快速兴起，直至 21 世纪初成为连接世界的“互联网”。多台拥有 CPU 和存储容量，分布在全球各地的计算机可以通过计算机网络互相连接，提供了单台计算机无可企及的计算潜能，也让诸如 Web、社交媒体、大数据、云计算、人工智能、加密货币等商业应用成为可能。相应地，软件开发者们必须向现实妥协：他们再也无法像过去那样“搭便车”，无须修改代码就能利用单台计算机性能的指数增长提升自己代码的性能。如果想要提升软件的性能，加快软件的运算，利用更大的存储，只能想办法写一个利用多核 CPU，甚至多台计算机的程序。想象一下，用一台家用电脑运行一份单机程序去处理中国春运订火车票规模的请求，可能吗？中国春运订火车票规模的请求需要的瞬时计算量和存储量，是一台家用电脑计算和存储容量的数万倍。

开发者们很快发现，编写一个能利用多核 CPU 的程序很困难。不同的计算内核上同时运行着不同的线程，这些线程可能会访问相同的存储空间，进而产生冲突。他们需要利用锁、条件变量、信号量等基本构件，确保他们的程序能够在运行多个线程的共享存储多核 CPU 上正确执行。

编写一个能利用多台计算机的程序就更困难了。除了考虑多核 CPU 中已经存在的并发问题，程序员们还需要利用计算机网络在多台计算机间通信。可行的方案是，让同一个程序在每一台计算机上都有一个部分，它们通过调用网络协议（protocol）的接口通信，把多台计算机的资源整合成一个整体，实现某个功能。与其把这一套庞大而复杂的计算机代码叫做“程序”，还不如把它叫作“系统”。在这样的系统中，各台计算机通过通信实现整体功能，同时每台计算机上运行的硬件和软件可能有很大区别——

只要这些软件可以通过先前写好的系统协议通信，系统就能实现它的功能。同时，计算机网络无法给出近乎百分之百的通信可靠保证，多台计算机间通信也有很高的时延，这些特点都区别于同一台机器上不同模块之间的通信。一台计算机通过网络调用其他计算机资源的时间，远比调用本地资源的时间要长。1990 年代的情况是，访问本地内存可能只要 100 微秒，而访问其他计算机的内存可能需要数百毫秒。这就意味着程序员不能用写单机程序，甚至是多核 CPU 的方法去写一个在多台计算机上分工协作完成任务的系统，也就意味着把这样的系统看成单机程序是不现实的。这样的一个在多台计算机上通过网络通信共同完成任务的程序就叫做**分布式系统**（distributed system），其中的每台计算机有其独立的计算单元和存储单元，称为**节点**（node）。一般来说，分布式系统与普通程序最重要的区别就在于系统内各计算机间网络通信的时延不可忽略。

一个分布式系统的首要目标就是要充分利用多台计算机的资源。如果一台计算机能够实现程序员想要的功能的话，就没有必要花那么大的代价写一个在多台计算机上运行的系统了。理想的情况是“一份耕耘，一份收获”：假设所有计算机的性能都相同，程序员在分布式系统中投入两倍数目的计算机，就有理由期待分布式系统的性能变成原来的两倍。如果一个分布式系统拥有性能随节点数目线性增长的性质，我们就称它有**延展性**（scalability）。这里的“性能增长”，主要指的是在系统时延（latency）增长不大的条件下，系统的吞吐量（throughput）有提升。这样，我们写一个利用一万台计算机的分布式系统，就有能力把程序的性能提升一万倍，提供用计算机网络在春运期间订火车票的服务了。

然而，延展性并不是分布式系统设计中的唯一目标。计算机科学家和程序员们设计分布式系统，还有很多目标。其中最重要的一个目标是**可靠性**（reliability），即保证分布式系统很少发生故障（failure）。无论是什么样的计算机，都会发生或大或小的故障。硬件方面，计算机的磁盘可能会发生比特翻转，CPU、网卡等可能会达到使用寿命。更换这些硬件的时候，程序无可避免地需要停止运行。在计算机所在地发生自然灾害导致断电的时候，程序也会停止执行。软件方面，任何程序都可能有错误，有的恶意程序还会故意出错，并通过网络传播，威胁计算机程序安全。当今世界对计算机系统的可靠性有非常高的要求。2006 年一项雅虎公司的研究表明，Web 页面的加载时间每上升 100 毫秒，仍然对 Web 页面感兴趣用户的数量就会下降 30%。一个节点宕机（即终止运行，crash）之后，重新启动它可能要花费一分多钟。有哪个 Web 用户会为了你的程序提供的服务，耐心地等待一分多钟呢？停电带来的影响更是不可预测的。如果停电前计算机的数据未能及时存入持久存储，那么即使恢复供电，数据也会永久地丢失。如何确保分布式系统时刻可靠，数据不会丢失，是一个难题。

也许你会认为，当代计算机硬件的可靠性已经够高的了。就磁盘来说，三五年才需要更换一个是很正常的事情。那我们每隔三五年，把所有的磁盘更换一次，不需要更改分布式系统的软件代码，似乎就能解决磁盘可靠性的问题。然而，磁盘出现故障是一个概率性事件，时刻可能发生。有的磁盘可能安装后五年内不出现故障，而有的磁盘可能两个月就出现故障。在由数千个甚至数万个节点组成的分布式系统里，有可能每个小时就会有一块磁盘发生比特翻转，就会有一个节点宕机。更不用说断电、软件错误等其他故障了。总之，分布式系统的设计，需要把形形色色的故障看成常态。

一个具有可靠性的分布式系统，可以掩盖自身的故障。一个有 3 个节点的分布式系统，如果 3 个节点都能接受用户请求，提供相同的服务，那么其中一个节点的宕机对于用户来说并不会有什么影响。如果一份数据在 3 个节点上都有复制，那么一个节点上数据的丢失也并不会导致数据在分布式系统内的丢失。一个节点出现故障，有其他的节点顶着。甚至自然灾害都不能威胁分布式系统的可靠性——如果一个分布式系统在欧洲、亚洲和北美各有一个提供相同服务的节点，那么北美发生大停电也不会影响欧洲和亚洲的节点提供服务，这种保证可靠性的策略称为**地理复制**（geo-replication）。我们称一个非常可靠，很少下线的服务具有很高的**可用性**（availability）。总之，一个经过精心设计的分布式系统，可以给用户营造一个“永远上线”的假象，甚至可以保证 99.99% 的可用性（指的是 99.99% 的时间内都上线提供保证质量的服务）。

话说回来，保证分布式系统内的所有节点在某些节点宕机或者加入前后提供相同的服务，并没有那么容易。为了保证很高的可用性，当某个节点加入或退出分布式系统时，分布式系统要一直上线，并且让用户不察觉其内部节点的加入和退出，这也就意味着分布式系统在节点动态变化的情况下要满足**一致性**（consistency）——分布式系统的各个节点要提供“相同”的服务。至于这里的“相同”，可能有很多不同的含义。也许你会对分布式系统提出很高的要求：在任何物理时刻，任何分布式系统节点的状态都完全相同，即使网络有时延。这种一致性称为**强一致性**（strong consistency）。我们很快就会发现，强一致性有时候不切实际。

分布式系统中，除了节点故障，有时网络也会故障：某些节点之间无法通信。如果特别巧，网络的故障可能恰好导致分布式系统的节点被分裂到两个（或者多个）不同的分区（partition）。每个分区的节点只能与同一分区内的节点通信，而无法与另一个分区的节点通信。一个可以掩盖分区故障的分布式系统称为分区相容（partition tolerance），也就是在网络分区故障或者愈合前后，分布式系统节点仍在正确工作，执行协议。怎么能让一个分布式系统分区相容呢？如果你考虑到同一个用户可能会把请求发到不同的分区，你就会发现一个分区相容的分布式系统必须作出以下的一个选择：你可以“硬抗”，选择在网络分区期间仍然提供服务。但如果一个用户在两个不同的分区作出了不同的请求，那你就没有任何办法确保不同分区间的节点强一致。如果你一定要确保系统内不同分区的强一致，你只能选择不要“打肿脸充胖子”，在分区期间不提供服务，在分区愈合，分区之间网络通信后再提供服务，然而这样就必须让服务下线，损害了系统的可用性。一个分布式系统是不可能兼有强一致性（C）、可用性（A）和分区相容性（P）的，这个事实被称为 **CAP 定理**。

一个分布式系统必须放弃 C、A、P 中的一个，这个结论似乎令人沮丧。许多现实中的分布式系统，比如亚马逊、谷歌提供的主要服务等等，都在 CAP 定理的“不可能三角”中选择放弃强一致性。这个选择可能有些违反常理：如果一个分布式系统都保证不了强一致性，这个分布式系统都不正确，怎么能提供优质的服务？然而有时候可用性和分区相容性显得更为重要。不要忘了之前提到的雅虎研究：用户可不愿意耐心地等待分区愈合。与其在网络分区时“罢工”，不如在网络分区时提供不能保证强一致性的服务，即暂时允许用户使用服务，比如购买商品等等，如果分区愈合之后发现强一致性被违反了（比如另一个分区内也有一个用户购买了这个商品，可库存只有一

件），再通过其他方法解决（比如通过客户服务，告知用户商品购买失败，可以给予一定补偿等等）。鉴于通常两个用户在网络分区情况下同时购买同一件商品的概率很小，放弃强一致性在这种情况下是最有利于服务提供者的选择。

（二）分布式系统的理论模型

CAP 定理说明，要写出一个满足以上列出的多个设计目标的分布式系统并不容易，有时还是不可能的。那么，到底满足什么样性质的分布式系统是可能的呢？分布式系统能怎样满足以上列出的可靠性、一致性、延展性等性质呢？

在物理学中，为了考察物体的运动，我们忽略了物体的形状、大小、化学性质等特点，将物体抽象为“质点”——一个有质量的点——进而考虑质点的运动规律。这是一个经典的从具体中抽象，建立理论模型，尝试找出客观规律的例子。那么，在对于分布式系统的研究中，我们能否从千变万化的各种分布式系统，抽象出所有分布式系统都拥有的特点，进而建立分布式系统的理论模型，探索分布式系统的客观规律呢？

一个分布式系统最核心的要素就在于有一定计算和存储功能的节点以及连接它们的网络。于是我们就可以把任何一个分布式系统抽象为由节点和网络组成的一个图。图的节点就是分布式系统的节点，图的边就是分布式系统的网络通路。那节点有什么样的能力呢？我们可以利用计算理论对计算机的抽象，把节点想象成图灵机——具有无限内存的一条纸带，和一个由状态机组成的带头。不同于对单个计算机的理论抽象，分布式系统中的图灵机可以通过网络从其他节点获取输入输出，也就是说，如果节点 A 与节点 B 之间有网络链路，则节点 A 可以向节点 B 发送消息，节点 B 可以接收节点 A 的消息。这个模型不考虑每个节点内部的存储能力和计算能力，而是落脚于“节点”和“网络”这两个分布式系统中最核心的要素上，应该是可以接受的。

为了让以上的模型更符合分布式系统的实际，我们还要回答两个问题。

如何在模型中体现“网络通信时延不可忽略”？为了考虑“时延”这个区分分布式系统与单机系统的重要因素，我们需要考虑系统的时间线。我们可以把节点想象成许多距离甚远的星球，星球之间以电磁波通信，而电磁波的速度是光速。我们当然可以在这个星球宇宙中考虑一个“全局时间”，也即一个物理上被所有节点分享的、绝对的时间度量，但狭义相对论告诉我们，这种模型过于绝对，与实验观察到的光速不变原理相矛盾。狭义相对论给出的答案是放弃绝对时间的存在，允许每个星球拥有自己的时钟参考系，有自己的时间度量。类似地，在分布式系统中，每个节点都可能有自己的时间度量。每个节点可能对自己当前所在的时刻有不同的看法（各个节点的时钟可能有或大或小的不同步）。每个节点图灵机执行一条指令的速度也不应该是确定的——有些节点执行得快，有些节点执行得慢。节点与节点之间发送和接收消息的网络也有很高的不确定性——有时发送的消息很快就能被接收，有时则要花很久，以至于接收方开始怀疑发送方是否出现了故障。我们的模型对时间的度量应该考虑这些情况。

我们定义，在分布式系统中每个节点（以下又称进程，process）p_i 上，在每个时刻（根据它自己的时钟）可以发生以下两类**事件**（event），记为 e_i：

（1）p_i 执行图灵机的一步操作；

（2）p_i 发送（send）或者接收（receive）一条消息 m，记为 $send(m)$ 和 $receive(m)$。

下面考察任意两个事件 e_i 和 e_j 之间的关系。有可能 e_i 和 e_j 都发生在同一个进程 p_i 上，也有可能它们发生在不同的进程 p_i 和 p_j 上。我们知道，每个进程都有它们自己的时钟，因此也就能给发生在自己进程的事件排序。我们假设一个进程上不可能在同一时刻发生两个事件。这是一个很合理的假设，因为如果上面的情况可能发生，我们不如把这一个进程拆分为两个进程。根据这个假设，如果 e_i 和 e_j 都发生在同一个进程 p_i 上，那么就必然有一个先于另一个发生。如果 e_i 和 e_j 分别发生在两个进程 p_i 和 p_j 上呢？这就不好说了。有可能这两个事件压根就没有什么关系——p_i 和 p_j 老死不相往来，永远也不和对方发送和接收消息。如果这个分布式系统内有一个全局时钟，那我们可能还可以判断 e_i 和 e_j 的先后。但既然我们要给分布式系统的全局时钟定义以更大的弹性，判断 e_i 和 e_j 的先后在这种情况下就没有什么意义。然而，在有些时候，是可以判断 e_i 和 e_j 的先后的：如果 $e_i = send(m)$，$e_j = receive(m)$，那么 e_i 肯定先于 e_j。于是，我们就可以为分布式系统中的事件定义一个满足以下性质的偏序“$\rightarrow$”，代表某一事件先于另一事件发生：

（1）对于发生在同一个进程 p_i 的两个事件 e_i 和 e_j，如果根据进程 p_i 的时钟，e_i 先于 e_j，那么 $e_i \rightarrow e_j$。

（2）对于发生在进程 p_i 的事件 e_i 和发生在进程 p_j 的事件 e_j，如果 $e_i = send(m)$，$e_j = receive(m)$，那么 $e_i \rightarrow e_j$。

（3）如果 $e_i \rightarrow e_j$，$e_j \rightarrow e_k$，那么 $e_i \rightarrow e_k$。

第（3）条又称传递性，保证了以上定义的“$\rightarrow$”是一个偏序。事实上，“$\rightarrow$”刻画了分布式系统中最基本的因果关系（causal relationship）：如果 $e_i \rightarrow e_j$，那么 e_i 一定是因，e_j 一定是果。倘若我们的分布式系统有个全局时钟 T，可以对系统内任何一个事件 e_i 排序（事实上也就定义了一个函数 T 满足 $T(e_i)$ 落在一个全序集，比如 $\mathbf{R}$），那么这个全局时钟 T 一定满足，对任意两个满足 $e_i \rightarrow e_j$ 的事件 e_i, e_j，$T(e_i) < T(e_j)$。

当然，在 p_i 和 p_j 老死不相往来的情况下，我们就没有办法让 $e_i \rightarrow e_j$ 或者 $e_j \rightarrow e_i$。如果两个事件 e_i, e_j 满足 $e_i \not\rightarrow e_j$ 以及 $e_j \not\rightarrow e_i$，则称 e_i 与 e_j 同时发生（concurrent），记为 $e_i || e_j$。站在“$\rightarrow$”的角度，区分 e_i 与 e_j 的先后没有意义；两个事件之间没有因果关系。

在分布式系统的研究中，我们还经常考虑同步和异步的概念。如果我们假设一个系统**同步**（synchronous），那么就意味着在以上关于分布式系统时间的基本假设之外，我们对系统模型作出如下三条假设：

（1）系统有全局时钟；

（2）如果两个进程间有网络通路，那么这两个进程间发送和接收消息的时间差有上界；

（3）各进程间执行某个图灵机指令的时间有上界。

相应地，**异步**（asynchronous）的分布式系统就没有以上三条性质。这意味着异步系统可能没有全局时钟，即使有全局时钟也不能保证进程间发送和接收消息的时间差有上界，也不能保证各个进程的相对指令执行速度有上界。我们之后会发现，有些分布式系统理论问题，在同步假设下能够解决，在异步假设下就不可能解决。

在实践中，由于网络速度高度不确定，以及网速比本地钟周期长得多，经常需要采用异步模型下的算法来保证分布式系统的可靠性。在有比异步假设更强的时钟假设时（比如能够给出网络发送和接收消息的上界等时），可以采用同步模型下的算法。

如何在模型中描述故障，从而分析分布式系统的可靠性？节点可以发生各种各样的硬件软件故障，但在分布式系统的理论模型中，我们要对故障的细节大幅简化，只考虑故障带来的后果。在分布式系统中节点与节点间通过以上定义的两类事件执行的算法一般称为协议。一个容错的协议应该在某些节点出现故障时仍然能正确执行，满足协议的要求。我们在分布式系统理论中一般考虑以下的节点故障模型：

（1）宕机（crash）。指在同步模型中，某节点不再工作。由于我们假设同步模型，节点之间发送和接受消息时间的上界和节点间相对运行速度的上界可以让其他正确节点探查到故障节点的宕机：如果根据协议，应该在此时刻收到来自节点 A 的消息，但自己的时钟已经超过了同步模型假设的时间上界，则我就可以判断 A 已经宕机。

（2）停止（fail-stop）。指在异步模型中，某节点不再工作。在异步模型中，节点之间发送和接受消息的时间等没有上界，所以如果其他正确节点不采用分布式系统以外的手段，仅凭分布式系统所提供的网络，就无法判断一个没有向自己发送消息的节点是已经停止，还是运行速度比较慢。

（3）忽略（omission）。指按照协议，某节点应该发送或者接收消息时并没有执行这些事件。未能发送消息称为发送忽略（sender omission），未能接收消息称为接收忽略（receiver omission）。如果在某个模型中两者皆有，我们就称这个模型有通用忽略（general omission）。宕机和停止故障可以看成特殊的通用忽略，因为宕机和停止的节点在发生故障后可以看成一直未能发送和接收消息。如果网络出现故障，无法保证发送和接收消息的可靠性，那么我们一般也可以把这种模型看成节点的通用忽略。

以上的（1）、（2）、（3）三种故障称为良性故障，因为这三种故障纯粹是节点“应有所为而不为”。它们都没有以下讲到的这两种故障厉害，因为以下的两种故障不仅包括以上三种故障，也包括“不应有所为而为”的恶意行为。

（4）带有消息认证的任意故障（arbitrary failure with message authentication）。指某个节点不再执行协议，开始执行任意的指令，包括随意向其他节点发送消息，随意接收其他节点的消息等。只不过该故障节点在发送的消息中无法伪造自己的身份。当它发送伪造消息时，其他正确节点仍能判断是该节点发送了正确节点接收到的消息。

（5）任意故障（arbitrary failure），又称拜占廷故障（Byzantine failure）。指某个节点不再执行协议，开始执行任意的指令，而其他节点甚至无法判断其接收到的消息是来自该节点，还是该节点伪造的其他身份。当然，在这种最低限度的故障假设下，正确节点可以假设网络不会出现任意故障，也就是网络不会平白无故、无中生有地“推送”出来一条消息。很明显，故障（5）包含故障（4）。至于为什么这种故障叫做拜占廷故障，我们在讲到拜占廷容错时会详细介绍。

在分布式系统理论模型中，经常对可能发生故障的节点数目给出上界 f。这个假设看起来有些奇怪：在实践中，确实可能有多于 f 个节点出现故障。作出这个假设的原因，一是因为实践中同时有很多节点（甚至大多数节点）出现故障的可能性比较小，二是这样有利于我们从理论中得到更好的结论。

（三）计算机程序的性质

无论分布式系统的结构多么复杂，满足的要求多么矛盾，分布式系统终究不过是一个计算机程序，或者说是一个计算机系统。为了方便讨论分布式系统的性质，我们在这里先讨论一下一般计算机程序的性质。

任何一个计算机程序都有它自己的性质。“无穷循环”这个程序是不会终止的；而如果计算机能调用的资源无限，计算某个自然数阶乘的程序是会终止的，哪怕这个自然数非常大。判断一个程序在某一输入下能否终止的问题就是大名鼎鼎的停机问题，而停机问题早已被证明是无法用计算机解决的。终止与否就是程序运行的一个重要性质。对于一个并发程序来说，是否会出现死锁也是程序运行的一个重要性质。能否保证可靠地传输数据，是计算机网络运行的一个重要性质。前面我们谈到的分布式系统的可靠性、延展性、一致性等等，经过严格定义之后，都能成为分布式系统运行时的性质。

一般地，一个在程序某次运行过程中赋值的命题（proposition，在本书中又称谓词，predicate）称为程序的**性质**（property）。比如说，“在该程序该次运行中，所有接收的消息之前都被发送过”、“该程序的该次运行会终止”、“窗口不会崩溃”等，都是关于其描述的程序的性质。关于这个定义，需要注意的是，不是所有关于程序的命题都是程序的性质。比如“在该程序所有的运行中平均每次会发送 50 条消息”就不是该程序的性质，因为这个命题无法在程序某一次特定的运行中赋值。性质这个概念只对程序的某次运行有意义。

我们可以讨论计算机程序某次运行的哪些性质呢？可以先来考虑下面的这个例子：A 地要新建一座化工厂。这座化工厂将会为 A 地创造大量的就业岗位和税收，带动上下游产业的发展，增长 A 地经济。然而，这些诱人的好处都必须建立在某些前提上：工厂必须保证安全，不污染环境，工人不加班等等。A 地仅仅要实现“保证安全”、“不污染环境”、“工人不加班”这些目标是非常容易的，容易到有些滑稽——只要 A 地不建化工厂就可以。什么都不干，当然能“保证安全”或者“不污染环境”了。工人没有工作，自然不会加班。同样地，如果 A 地仅仅要创造大量的就业岗位，仅仅要增加税收，也是很容易的——那样的话 A 地政府就免去了监管工厂生产安全、污染排放和保护工人权利的职责，多省事啊！遗憾的是，在现实生活中，我们没有办法“一边倒”：只实现某一方面的目标是不可接受的。明智的 A 地一定会想两全其美：既要创造就业岗位，又要保护工人权利，既要提供税收，又要保证安全，保护环境等等，而同时实现这两方面的目标，远远比只实现某一方面的目标困难。在建造化工厂的过程中，投资者与环保主义者、当地居民等的博弈，就体现了这个困难的过程。

说来可能会让你诧异，在计算机程序的编写过程中，像上面兴建化工厂一样的“博弈”案例司空见惯。为了讨论清楚这样的“博弈”是怎么回事，我们考虑程序的安全性质和活健性质这两个概念。描述在程序运行的过程中不会有坏事发生的性质称为程序的**安全性质**（safety property），而描述在程序运行的过程中最终会有好事发生的性质称为程序的**活健性质**（liveness property）。“不会同时有 k 个进程处于程序的临界区段”、“消息是按照发送的顺序被接收的”、“窗口不会崩溃”等性质都是安全性质，而

“希望进入临界区段的进程最终都能进入”、“某个消息会被接收”、“窗口最终能够启动”等性质都是活健性质。

如果把“化工厂的运行”看成一个正在执行的程序，那么“保证安全”、“工人不加班”等性质就是这个“程序”的安全性质，而“创造就业岗位”、“提供税收”就是这个“程序”的活健性质。通过这个例子，我们可以发现，在生活中安全性质和活健性质无处不在，只是我们把这些性质在计算机系统中的体现抽象了出来，单独研究而已。

如果一个安全性质在程序运行的全过程中满足，那么在程序运行的前缀中，这个安全性质也一定能满足。比如说，如果在程序运行的 10 秒内窗口都没有崩溃，那么在程序运行的前 2 秒内窗口肯定没有崩溃。因此我们说，安全性质是“前缀封闭”的。相应地，一个活健性质在程序运行的前缀中不满足，不代表它最后就不会满足。例如，某程序运行 10 秒，前 8 秒都没有收到消息，不代表最后 2 秒内就不会收到消息。

现在用符号语言表达上面的概念。记 t_0 为我们开始关心程序某性质的时刻。程序的安全性质 P 满足 $\forall t \geq t_0, P(t) = \text{False}$。程序的活健性质 P 满足 $\forall t_1 \geq t_0, \exists t \geq t_1, t < \infty$ 满足 $P(t) = \text{True}$。

跟以上提到的化工厂例子类似，编写一个只满足安全性质的程序是非常容易的——做“甩手老板”，不干活就可以。如果我们不允许任何进程进入程序的临界区段，那么自然就不会同时有 k 个进程进入程序的临界区段。然而，没有人会想编写一个只满足安全性质的退化程序，因为这还不如不写。类似地，编写一个只满足活健性质的程序也相对比较容易。把所有希望进入临界区段的进程都立即放进去，自然可以让所有希望进入临界区段的进程最终都能进入。但是，跟化工厂的例子一样，编写一个既满足安全性质，也满足活健性质的程序就要困难得多。操作系统课程的并发部分一般都会花很大篇幅讨论临界区段问题，其实质就是写一个能同时满足“不会同时有 k 个进程处于程序的临界区段”这个安全性质和“希望进入临界区段的进程最终都能进入”这个活健性质的程序。

事实上，Alpern 和 Schneider 曾证明，任何一个性质都可以表示为一个安全性质和一个活健性质的组合。这个定理的证明，不在本书的讨论范围。

我们在之后的分布式系统设计中，需要讨论关于分布式系统的很多复杂性质。许多分布式系统的经典算法，其实质就是安全性质与活健性质的艰难平衡。在本书之后的讨论中，我们有时还会证明，同时满足某些安全性质和活健性质的算法是不存在的。

（四）习题

在本书中，习题与正文一样重要。习题例证了正文中定义的概念，拓展了正文的知识内容，在程序设计中也有很多应用。因此，本书中的习题应该认真完成。

1. 时空图

2. 两将军问题

东汉末年，天下大乱。作为刘备军队仅次于诸葛亮的二号军师和为数不多的将军之一，你肩负着铲除曹操奸党，匡扶汉室的重任。

你和关羽将军得到情报，南下的曹军主力正在两个山头中间的山谷扎寨，准备今晚在此露宿一宿。你的军队准备埋伏在东边的山头，关羽的军队准备埋伏在西边的山头。你知道，如果东山和西山两边的军队**同时**从山头冲下山谷，突然袭击，曹军主力必然会被歼灭。然而，你和关羽都知道，仅凭东山或者西山的军队，数量不够，是无法战胜曹军的。在部署在山谷之前，你和关羽有机会商量袭击策略，不过由于曹军安营扎寨诡密多端，需要在部署后多花时间观察曹军动态。如果你或者关羽在部署后认为曹军过于强大，你们两方都可能为了保存实力而放弃袭击计划，作出撤退决定。这些因素都导致你和关羽不可能在部署在山头前，确定是否袭击曹军，以及何时一起袭击。

你知道，为了成功袭击，必须与关羽将军在部署后通风报信，把袭击的时间约定（比如约定“破晓时分攻击”或者“二更时分攻击”）。为此，你可以派一位小兵，穿过曹军扎寨的山谷，从东山跑到西山，告诉关羽自己的想法。然而，关羽并不一定同意你的想法，因为他可能认为在你选择的时刻，他的部下还没有准备好。因此关羽可能会派另一位小兵穿过山谷，从西山跑到东山，告诉你他的看法。在知道他的想法后，你可能还会再派一位小兵到关羽那边，告诉他你的想法。这样来来回回，可能最终能达成共识。然而，通风报信是危险的：如果穿过山谷的小兵被曹军认出，小兵的脑袋肯定就要掉下来——没有任何对这种通风报信方式可靠性的保证。你与关羽的通信方式，只有小兵一种：在山上放篝火、点灯等通信方式，都会向曹军暴露你们的身份。而且曹军扎寨的山谷是小兵的必经之路，没有绕路通信的选择。在本题中，不考虑小兵叛变投敌做卧底，传播虚假消息的可能性。

证明：你与关羽将军之间，不可能在以上的情形下设计一种非退化的用小兵通信约定袭击时间的协议，使你与关羽要么撤退，要么同时袭击曹军。所谓“非退化”，是指在部署在山头后，你部和关羽部可能袭击，也可能撤退；如果袭击，就可能在当晚的任何时刻袭击。

3. 程序的性质

在“两将军问题”中，你考虑了能否设计一种非退化的用小兵通信与关羽将军约定袭击时间的协议。下面的针对满足“两将军问题”要求的通信协议的命题（1）是安全性质吗？（2）是活健性质吗？（3）是安全性质和活健性质的组合吗？（4）还是根本不是性质？

（1）如果你和关羽之间一位将军决定袭击，那么在 5 分钟之内，你们中的另一位将军也会决定袭击。

（2）你和关羽之间没有人会决定袭击。

（3）你和关羽之间最终会达成共识，决定袭击的准确时间。

（4）多数情况下你和关羽都不会袭击。

（5）关羽最多发送 10 条消息（送出 10 个小兵）。

（6）你最少发送 5 条消息（送出 5 个小兵）。

4. **额头抹泥的小朋友**

5. **领导选举问题**

二、统揽全局——系统全序和全局谓词赋值

三、合众为一——一致性

四、整齐划一——状态机复制

五、民主集中——同步模型下的共识协议

六、无可奈何——FLP 不可能定理

七、柳暗花明——绕过 FLP 定理的方法

八、道高一丈——拜占廷容错共识协议

参考文献

附录

123

后记

123