摘　要

目前，随着网络的规模不断扩大，对路由协议提出了越来越高的要求，而现今普遍运行的 RIP（Routing Information Protocol）已不能满足这要求。因此，就需要使用其它的路由协议。本课题正是针对这一需求，来实现 OSPF（Open Shortest Path First）协议的。

该论文介绍了现今网络上所使用的多种路由协议，提出了 RIP 协议的不足，并借此阐述了 OSPF 协议的优点。紧接着，以链路状态数据库为核心，层层深入的介绍了 OSPF 协议的原理及工作流程。在实现部分，结合目前教研室已开发了的路由器软件，分析了其体系结构。并以此为基础，研究了 OSPF 路由协议实现的方案，利用 pSOSystem 这一嵌入式实时多任务操作系统，完成并实现了该模块的主体部分以及其中的一些子模块。经调试，该程序已能正常运行。最后，搭建了一个测试平台，模拟实际的网络环境，编制了测试程序对已实现的部分进行了测试，给出了测试的结果。

该工作对路由协议的研究起着非常重要的作用，对程序的进一步优化打下了坚实的基础。

关键字：RIP，OSPF，链路状态数据库，链路状态广告，指派路由器，备份指派路由器

Abstract

With the development of the network scale, it becomes more important to the performance of the routing protocol. However, the RIP protocol, which is prevalently used in routing technology, can not reach this developing requirement. Hence, it needs other routing protocols. This project just aims at this need to implement the OSPF routing protocol.

This dissertation introduces several routing protocols which are used in the Internet, and discusses the deficiency of the RIP and the advantages of the OSPF. Next, it goes gradually deep into the principle and the working flow based on the LSDB. In the implementing chapters, I analyses the structure of the router's software which was developed by our lab team. And based on it, I put forward the project of realizing the OSPF modul, and implement the main module and some of sub-modules based on the pSOSytem. After debugging, the program can operate normally. At last, I make a testing platform which is used to simulate the real network environment, test the program which had been implemented and give the test result.

This project is very useful to the research of routing protocols. Moreover, it construct a solid basis to the further optimization.

Keywords:RIP, OSPF, LSDB, LSA, DR, BDR

独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献已在论文中作了明确的说明并表示谢意。

签名：邓舒　　日期：2001 年 12 月 28 日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复印手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：邓舒　　导师签名：[illegible]

日期：2001 年 12 月 28日

引言

“我们为什么需要网络？”这是一个老师曾经问他的学生的一个问题。有学生说：“因为我们想去更远的地方。”是的，我们能够通过网络了解发生在很远的地方的事情。网络是信息的高速公路，它是靠作用与立交桥一样的路由器将它连接并延伸的。路由器通过查找自己的路由表来获知该将信息往哪一条路上送，由此可以得知，路由器需要掌握网络的路由情况，而路由器又是通过路由协议来得到这一信息的。因此，路由协议对路由器来说是非常重要的。路由协议的好坏会直接影响到路由器的性能。

现在，网络上普遍使用的路由协议是 RIP。它是一个非常简单的路由协议，人们对于它已做了深入的研究，并不断地对它进行改进。从产品的角度来讲，应用它的路由器已经是很成熟的了。但是，由于它自身的一些没有办法改变的致命弱点，限制了它适用的范围，使得它只能适用于规模较小的网络。从市场的角度来讲，随着 Internet 的发展，接入 Internet 的路由器越来越多，路由负载不断增加，网络规模不断扩大，我们需要适用于大网的路由协议，以改善网络的性能。OSPF 就是一个很好的选择。但是，由于它的复杂性，完全掌握该技术的团体和个人还是少数。人们对于它的研究也远不如 RIP 那样深入，该协议也许还有很多需要改进的地方。因此，对于我们来说，我们有必要对该路由协议进行研究，有必要自己去设计一套具有自主知识产权的软件。

第一章　Internet 上路由协议的使用现状

1.1　自治系统的划分

早期的 Internet 是由 Arpanet 及其伙伴网络如 Satnet 发展起来的。起初，它连接的是研究中心的独立计算机，后来，发展到可以向局域网提供访问服务。不过，直到 20 世纪 80 年代早期，它都仍然是一个网络。在这种情况下，所有的路由器都采用同一网关一网关协议（GGP）来共享路由信息，路由表中包含了 Internet 上所有的 IP 网络项和量度值。

随着 Internet 的发展，接入 Internet 的路由器越来越多，路由负载不断增加，路由表的大小也随着接入的网络数量的增加而增加。路由器和链路的数目越多，就越可能出现问题。每次链路通或断，都必须重新计算整个路由表。在这种情况下，重新计算路由表的工作量将大大的增加。为了让整个网络的路由信息一致，还必须在网络上传送变化以后的路由信息，这将引起网络上数据流量的增加。这是管理巨型网络所引起的一个问题：路由负载问题。此外，管理巨型网络还会引起其他的一些问题，如：路由器种类的增加，装配在其上的软件的不相同，将导致各自特定的 GGP 之间无法正常工作，进一步导致无法进行维修和故障隔离；随着路由器数量的增加，当路由算法的新版本出现的时候，还需要在同一时间对所有的路由器升级，这就会给路由算法新版本的推广工作带来很大的困难。

从上面出现的一系列问题，我们可以看出，要想从根本上解决这个问题，就必须改变“单一网络”这种模型。将 Internet 划分成一系列的自治系统，每一个自治系统由同一个机构管理下的一系列路由器和网络组成。一个自治系统内的路由器交换网络拓扑信息，寻找最佳路径。自治系统内的网络拓扑信息不为自治系统外的路由器所知道。自治系统之间通过专门的路由器进行连接，这些路由器之间交换可达性信息，寻找可达路径。这样一来，可大大的减少路由表的条数，减小网络的规模，让网络更加便于管理。

1.2　路由协议的使用

在路由器上使用的路由协议有静态路由协议和动态路由协议之分。静态路由协议不利用网络的信息，只是按照某种固定的规则去选择路由。这样，在网络的拓扑发生变化的时候，它不能及时的调整自己的路由信息，最多只是由操

作人员偶尔对网络的状态的变化作出反应。由于它不能对网络的改变作出反映，故一般用于网络规模不大、拓扑结构固定的网络中。其优点是简单、高效、可靠。与之相反，动态路由协议则能根据网络拓扑的变化（比如某个网络端口不能工作），在一段网络路由信息汇聚的时间后，计算出新的、正确的路由，以适应网络流量和拓扑的变化。当然，动态路由也有不能正常工作的情况，这就需要静态路由作为它的补充。在这里我们讨论的仅是动态路由协议。

在自治系统内的路由器我们称之为内部网关，它们之间通过交换网络拓扑信息，来寻找最佳路径。在此过程中所使用的路由协议，被称之为内部网关协议（IGP）。常见的 IGP 有：RIP、OSPF、IGRP、EIGRP 等。

在自治系统外的路由器我们称之为外部网关，它们之间通过交换可达性信息，来寻找可达路径。连接两个自治系统的外部网关并不需要了解这两个自治系统的具体的网络拓扑，只需要了解通过它可以到达哪些网络。在此过程中所使用的路由协议，被称之为外部网关协议（EGP）。常见的 EGP 有：EGP、BGP、BGP－4 等。

这样的策略也很适合现在的实际情况。不同的 Internet 服务提供商（ISP）根据自己的需要和管理策略，将自己管理的网络划为一个自治系统，在这个自治系统内采用自己的路由策略来管理自己的网络。基于利益的考虑，ISP 不愿意向别人提供自己网络的详细路由信息。同时，基于网络的发展趋势，ISP 之间又必须进行互联。外部网关协议正好满足这一要求，ISP 之间可以通过外部网关协议来进行连接。

这样一来，在市场上就出现了种类和功能繁多的路由器，它们支持各种不同的路由选择算法。有的适用于自治系统之内，有的适用于连接自治系统。

1.3 内部网关协议（IGP）

在适用于自治系统内部的路由器上，即内部网关上，目前多采用的是 RIP 协议。RIP 协议之所以被广泛应用，主要是由于它很简单。RIP 是一种距离向量协议。对于距离向量协议来说，它告知邻居整个网络的拓扑。RIP 通过周期性的将自己的路由表广播出去来实现这一点，这样还可以达到维护路由器之间的相邻关系的作用。同时它也会收到别人广播的路由表，它会根据这些路由表的内容来生成自己的路由表。当然，简单也必须要付出一定的代价：

1. 对于庞大而又复杂的网络来说，RIP 可能根本无法胜任。虽然在网络的拓扑结构发生变化后，RIP 会重新计算新的路由，计算到各个网络和路

由器的距离值，在这种情况下如果遇到距离值计数到无穷大等情况，计算就变得非常缓慢，网络的收敛速度将变得相当缓慢。为了加快网络的收敛速度，将 16 设置为极限值，在进行计数时，只要距离值达到了 16 就认为两点之间不可达。然而，这样就将 RIP 限制在小网络上使用了，因为在大网络上两点之间的距离值往往会大于 16。

2. 周期性的广播路由表将消耗大量的网络带宽。这个问题对于大网，尤其是慢速链路和广域网就更加突出了。
3. 在重新计算路由的过程中，路由器处于一种过渡阶段，网络上会出现大量的广播报文，并会引起循环，从而造成网络暂时的拥塞。
4. RIP 在对两点之间的距离进行量度的时候，其标准是路径上所经过的路由器的数目（hop），选择 hop 数最少的那条路径。这样就没有考虑到网络延迟和链路状态等对两点之间传输距离的影响。

针对于 RIP 的不足，我们大都采用 OSPF 协议。OSPF 是一种链路状态协议。对于链路状态协议来说，它向整个网络告知自己的邻居信息。因此，在这个协议中，各个网络节点不必交换通往目的站点的距离，而只需维护一张网络的“拓扑图”，在网络拓扑结构发生变化的时候及时更新这张拓扑图。各个路由器根据这张图分别计算到不同目的地的距离，从而生成各自的路由表。它解决了 RIP 所不能解决的一些问题：

1. 无 hop 数的限制，因此不必被限制在小网中使用。
2. 每个路由器都掌握了网络的拓扑结构，各自按照拓扑结构来进行路由优化算法，形成自己的优化路由。
3. 更新报文的发送是在路由发生了变化的时候，而不是周期性的发送，故减少了网络上的路由信息的流量，有利于带宽的有效利用。
4. 支持多种度量制式，充分考虑网络延迟、链路状态和吞吐量等因素对两点之间传输距离的影响。
5. 支持具有相同量度值的多条链路之间的负载分担。

OSPF 的功能很强大，这些都只是它的功能中的一部分。另一方面，它也很复杂，因此它远不如 RIP 应用广泛。正是由于这个原因，现在完全掌握这个技术的人还是少数。

目前，我们教研室正在进行路由器的研发工作，考虑到网络的发展趋势，以及组网特点的改变和需求，OSPF 应该是路由器所必须支持的基本协议之一。在这种情况下，我们就有必要自组研究，自行开发具有自主知识产权的 OSPF

软件。我所做的工作正是这其中的一部分。

第二章 OSPF 协议解析

在前面一章我们已经简要的介绍了一些 OSPF 协议的特点，现在，我们将就其中的一些主要细节和关键技术进行一个较深入的讨论。这都将围绕着链路状态数据库来展开。

OSPF 用链路状态数据库来表述网络的拓扑结构，用加权有向图描述了路由的评价方式，定义了几种网络类型，采用三种协议来生成和维护计算路由的相关数据。

2.1 链路状态数据库

网络的拓扑结构在 OSPF 中用链路状态数据库来表述，它是 OSPF 协议中关键的一个部分。在一个自治系统中，所有运行 OSPF 的路由器都需要维护一个相同的链路状态数据库。其实，这个数据库就是一张有关这个自治系统拓扑结构的图，同时它还是一张加权的图。图中每一条边都与一个权值相关联，权值表示沿这条边所示的方向传输数据的代价。这个代价值可以是由管理员自己配置的，也可以是从其它的路由协议中获得的。这样一来，所有路由器都了解了整个自治系统的拓扑结构。在此基础之上，每个路由器根据这张加权图利用 Dijkstra 的 SPF 算法来计算到每一个目的地的最短路径，从而生成路由表。正是由于这个原因，使得尽管路由计算是分布式的，但其计算的结果与集中式计算出来的结果一样精确。

一个自治系统是由一系列的网络和路由器所组成的，那么，我们应该怎样来表示这个自治系统呢？在存放这个自治系统拓扑结构的链路状态数据库中，我们应该怎样来表示这些网络和路由器？

我们用有向图来描述一个自治系统。网络和路由器是这个图中的顶点。网络和路由器之间以及路由器和路由器之间的关系则用图中的点和边来表示：如果表示两个路由器的点用一条边连接起来，则表示这两个路由器通过一个点到点的网络相连；如果一个表示路由器的点和一个表示网络的点用一条边连接起来，则表示这个路由器有一个接口与这个网络相连。按照这样的规则，图 2-1 所示的网络，就可以用图 2-2 所示的图来表示。其中 RTA、RTB、RTC 和 RTD 分别表示路由器 A、B、C 和 D，N1～N5 分别表示网络 1～5。每条边旁边的数字表示这条边所对应的权值。

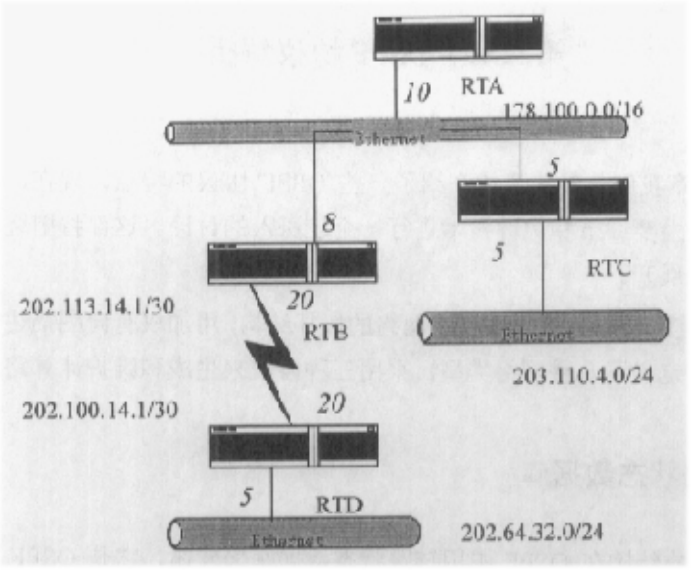


图 2-1　网络实例

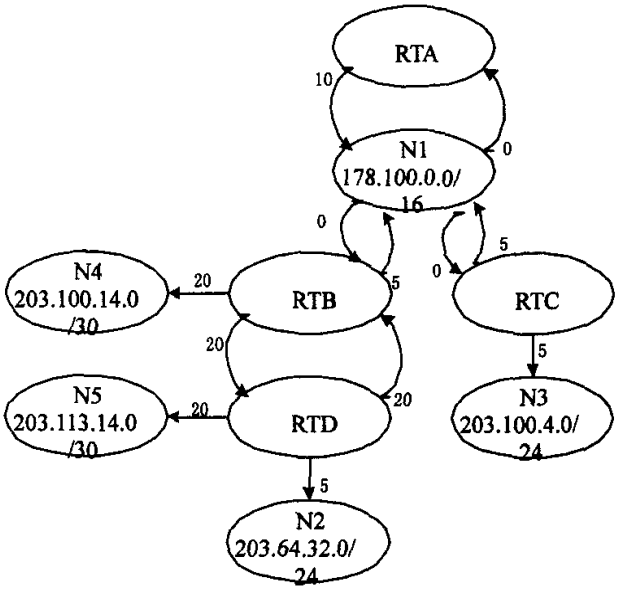


图 2-2

在图 2-2 所示的图中，RTA、RTB 和 RTC 与 N1 之间的连接示双向的，这表示 N1 是一个 transit 网络，它为其它网络间的通信提供通路，而其它的网络与路由器之间的连接都是单向的，对网络而言就是只有入边，而没有出边，这表示这个网络是一个 stub 网络，这种网络只提供网络内的目的之间的通信，不为其他网络间的通信提供通路。这两种网络是 OSPF 中典型的两种网络类型。

前面所讨论的都是怎样用图来表示网络的拓扑结构，那么，在属于该自治系统的所有路由器上又是怎样来存放这张图的呢？表 2-1 描述了链路状态数据库的逻辑表示。

表 2-1 链路状态数据库的逻辑表示

FROM / TO	RTA	RTB	RTC	RTD	N1
RTA					0
RTB				20	0
RTC					0
RTD		20			
N1	Cost=10	5	5		
N2				5	
N3			5		
N4		20			
N5				20	

在有了这个数据库以后，各个路由器就可以根据它来生成最短树，从而计算出自己的路由表。

2.2 生成树

每个 OSPF 路由器利用形成的链路状态数据库，通过 Dijkstra 算法，以自己为根节点得到一个最短树，因此最短树因运行该算法的路由器不同而不同。最短树给出了去任何目的网络和主机的整个路径，但只有下一跳地址在 IP 转发中使用。

现在，我们就以 RTA 为例，在其上运行 Dijkstra 算法，得到最短树，然后再根据这个最短树来计算出 RTA 的路由表。由于这个链路状态数据库只是有关本自治系统内的，所以，得到的路由表也只是本自治系统内的路由表。只是图 2-3 给出了 RTA 计算的最短树。表 2-2 则是 RTA 计算出来的路由表。

表 2-2 RTA 的路由表

目的地址	下一跳网关	费用
N1	*	10
N2	RTB	35
N3	RTC	15
N4	RTB	30
N5	RTB	35

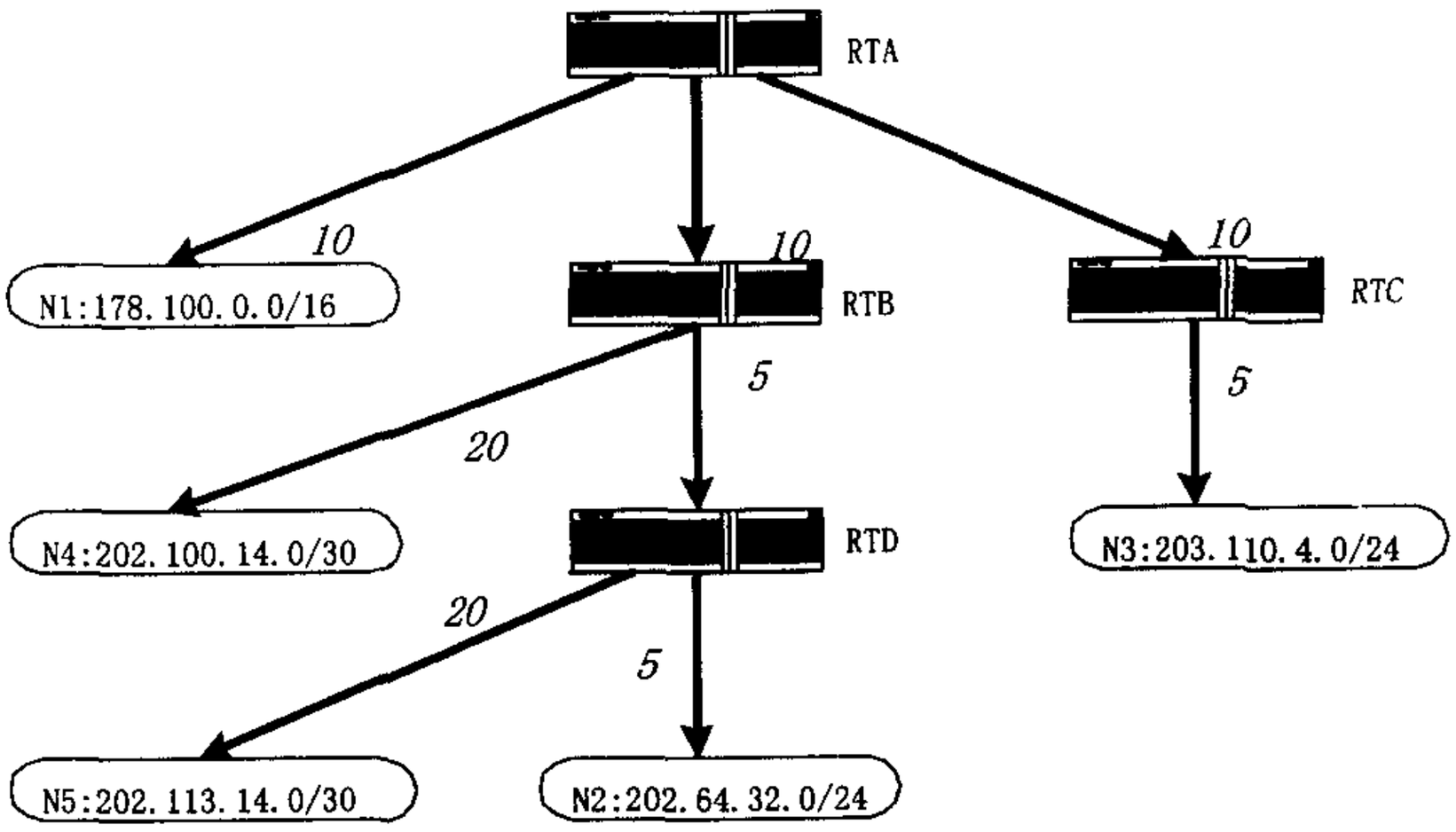


图 2-3　RTA 计算的最短树

从表 2-1 我们可以看出，在这个链路状态数据库中包含了整个自治系统所有的网络和路由器。从表 2-2 可以看出，整个自治系统中所有的网络在路由表中都有对应的表项。这样一来，当自治系统的规模扩大的时候，就会产生一些问题：所包含的网络和路由器的数目就会增多，链路状态数据库也就会变得相当庞大，从而超出一定的限额；利用这个链路状态数据库计算最短树的时间也会因此而超出一定的限额；路由表的大小也会因此而超出一定的限额；当某一条链路的状态发生了变化的时候，OSPF 路由器会向自治系统中所有的路由器通告这个变化，大量的路由更新报文会使得网络带宽的有效利用率变低，我们又一次面对与 RIP 带给我们的同样的问题。为了解决这个问题，OSPF 协议允许把自治系统划分为更小的单位，我们称之为区域（area）。

2.3　区域的划分

首先我们需要说明的一个问题就是：什么是区域？在 OSPF 协议中允许将连续的网络和主机组合在一起形成一个集合，再加上有接口与这个集合中的任何一个网络相连的路由器就形成了一个区域。这样一来，我们就可以将自治系统划分为若干个区域。每个区域独立的运行链路状态算法，区域内的网络拓扑结构对于区域外的路由器是不可见的，同样区域外的拓扑结构对于区域内的路由器同样是不可见的。处于同一个区域中的路由器共同维护一个相同的链路状态数据库。这就使得位于一个 AS 内的路由器的链路状态数据库也不再完全相同，不同区域的路由器具有不同的链路状态数据库。由于这个数据库中只记录

了该区域中的网络拓扑结构，所以链路状态数据库的大小被缩小了。这种拓扑结构的隔离使得网络的扩散过程止步于区域的边界，从而减少了网络的流量。

这一个一个独立的区域又是怎样联系起来的呢？它们是通过接在不同区域的路由器联系起来的。这样的路由器我们把它称之为区域边界路由器（ABR：Area Border Router）。它们属于多个区域，在其上维护多个链路状态数据库。

为了区分不同的区域，我们对区域进行了编号，不同的区域分以不同的编号。但是，在这里面有一个特殊的区域，它的编号必须是 0。这个区域我们称之为主干区域（Backbone Area），它是由所有的区域边界路由器所组成的。它负责非主干区域间的通信，因此，它必须是连续的（contiguous）,即使不是物理上连续的，也应该通过虚拟链路的建立保证其逻辑上的连续。

这就是在一个自治系统当中我们所采取的一些改进措施，如果一个自治系统要与其它的自治系统进行路由的交换，那又应该怎样办呢？这是通过自治系统边界路由器（AS Border Router）来实现的。通过它将 AS 外的路由信息传进自治系统内，同时也向其它 AS 宣告本 AS 的一些可达信息。

有了这样的划分之后，一个自治系统中的路由器就可以分为以下三种：

1. 区域内部路由器：直连的所有网络都在一个区域的路由器。
2. 区域边界路由器：与多个区域相连的路由器。它将一个区域的路由信息汇总压缩后向主干区域发布，同样也将主干区域的路由信息向其他区域发布。
3. AS 边界路由器：负责与 AS 外的路由器交换路由信息的路由器。

2.4 链路状态广告及其分类

我们将路由器产生的链路状态信息称为链路状态广告（LSA：Link State Advertisement）。链路状态数据库中存放的、路由器之间相互交换的就是一条一条的链路状态广告。根据路由器类型的不同，它所产生的链路状态广告的类型和内容也就不相同。

链路状态广告共分为 5 类，表 2-3 描述各种不同的 LSA 的详细的情况：

表 2-3　OSPF　LSAs

LSA 类型	对该 LSA 的描述
类型 1 Router-LSAs	所有的路由器都要生成该 LSA。它描述了位于同一区域下路由器的接口的状态，只在一个区域内扩散。
类型 2 Network-LSAs	由广播型网络和 NBMA（Non-Broadcast Multi-Access）网络中的指派路由器为该网络生成。它描述了一个网络的情况，记录了位于该网络中的路由器。只在一个区域内扩散。
类型 3 和类型 4 Summary-LSAs	由区域边界路由器（ABR）生成。只在与此 LSA 相应的某个区域中扩散。每一个 summary-LSA 描述了一个区域外部但仍在 AS 内部的路由。类型 3 描述的目的地址为网络，而类型 4 的目的地址为自治系统边界路由器（ASR）。
类型 5 AS-external-LSAs	由自治系统边界路由器生成。每个 AS-external-LSA 描述了一条自治系统以外的路由。在整个自治系统中扩散。

注：指派路由器是在广播型网络和 NBMA（Non-Broadcast Multi-Access）网络上执行特殊功能的路由器。在这些网络中，接在该网络上的所有路由器之间都需要建立邻接关系，以便相互交换链路状态信息。如果接在该网络上的路由器数目很大，则需要建立的邻接关系的数量就很大。指派路由器的引入，网络中的非指派路由器只需要与指派路由器建立邻接关系，非指派路由器之间并不需要建立邻接关系，这样就使得这种类型的网络中所需要建立的邻接关系的数量减少，从而也减小了网络的开销。为了保证该策略的可靠性，我们还引入了备份指派路由器。平时，备份指派路由器与指派路由器和非指派路由器均保持这邻接关系。当指派路由器发生故障的时候，它就接替指派路由器的工作。

在这里面涉及到一个 LSA 的区分问题。我们用与每一个 LSA 相对应的广告路由器标识（Router ID）、链路状态标识（Link State ID）和链路状态类型（Link

State Type）的组合来标识它。不同的 LSA 具有不同的标识。

2.5 链路状态数据库的形成与维护

我们已经讨论了链路状态数据库的内容，现在我们自然该来讨论它是怎样形成的，以及当网络的拓扑结构发生变化的时候，它又是怎样来进行更新，从而维护一张有关网络的最新的拓扑图。

在 OSPF 协议中链路状态数据库的形成与维护是通过三种协议(Hello 协议、交换协议和扩散协议）来完成的。

2.5.1 Hello 协议

从前面的讨论，我们知道运行 OSPF 协议的路由器向全网告知自己的邻居信息，当邻居状态发生变化的时候，它又会向全网通告这个变化。那么，OSPF 路由器怎么知道自己的邻居是谁，怎样检测到自己的邻居已经发生了变化？这一切都是由 Hello 协议来完成的。除此之外，在前面所提到的指派路由器和备份指派路由器的选举，也是由它来完成的。因此，我们可以从以下的几个方面来描述 Hello 协议：

1. 动态发现新邻居

Hello 协议中规定，一个运行 OSPF 协议的路由器从它加入网络起，就需要定期的向网络发送 hello 分组。邻居通过收到这个 hello 分组来发现它的存在。而它则通过收到邻居发送的 hello 分组来发现自己的邻居。

2. 确认邻居间的双向连接关系

邻居之间要进行进一步的操作，必须先建立双向连接。如果 OSPF 路由器检测到邻居发来的 hello 报文的邻居列表中含有自己，说明邻居已经收到了自己发送的 hello 报文，能够在网络上看见自己。此时，邻居间的双向连接关系就建立起来了。

3. 维持与邻居间的邻接关系

一个 OSPF 路由器通过定期（如 10 秒）向网络发送 hello 分组来告知邻居自己的存在，同时它通过收到邻居的 hello 报文，来确保邻居还活着。如果一个 OSPF 路由器在规定的时间内（通常为发送 hello 分组时间间隔的 4 倍，如 40 秒）没有收到邻居发送的 hello 分组，则该路由器可以确认此邻居已经死掉了，从而来发现拓扑结构的变化。

4. 指派路由器的选举

对于广播型网络和 NBMA（Non-Broadcast Multi-Access）网络，我们在所有与该 OSPF 路由器建立了双向连接的邻居之间，通过路由器的优先级、ID（IDentification）的比较，来选出指派路由器。其它的所有路由器需要与它交换彼此的链路状态数据库，从而建立邻接关系。

Hello 分组的发送可以以组播的形式，发给网络上的所有 OSPF 路由器，或者以单播的形式发给自己的邻居。

对于选举了指派路由器的网络，在指派路由器和非指派路由器之间；对于其它的网络，在建立了双向连接的路由器之间，都需要建立邻接关系，这就需要随时保持链路状态数据库的一致。这个一致是通过交换协议和扩散协议来完成的。

2.5.2 交换协议

交换协议仅用于链路状态数据库的初始同步，它规定了刚建立双向连接而又需要建立邻接关系的路由器之间的链路状态数据库怎样进行初始的交换。

在这个交换过程中，路由器双方是非对称的，一个扮演“主”的角色，另外一个扮演“从”的角色。因此，这个过程的第一步就是“主”“从”角色的协商。之后，进行的操作就是“主”“从”之间相互告诉对方自己的链路状态数据库中的内容。在第二步中，“主”路由器主动发起交换过程，告知“从”路由器自己的链路状态数据库中有什么内容，“从”路由器收到后，进行应答，并在应答分组中带上自己的链路状态数据库的内容。此时，双方传送的是“数据库描述分组”，这些分组只标志了各个不同的 LSA，而不是对每条 LSA 的具体内容进行述说。在此过程中双方都会将对方的数据库中的内容与自己的数据库中的内容进行比较，若发现自己的数据库中没有该项记录，则将它放入一个请求链表中，以便稍后向邻居索要该记录。自然，交换过程的第三步就是向对方发自己的请求链表，并在收到对方请求后，将对方请求的 LSA 发给它。

在此过程完成后，双方链路状态数据库的内容都达到了一致，这两个路由器之间的连接关系就建立起来了。

2.5.3 扩散协议

交换协议仅仅保证了在初始时刻，邻接路由器间链路状态数据库的一致，然而，当网络的拓扑结构发生了变化的时候，一个路由器检测到了这样的变化，它就会修改自己的链路状态数据库的内容，为了保证链路状态数据库的一致性，它还必须将这个变化传出去，此时，交换协议就没有办法工作了，这就需要用

其它的协议来完成。这个功能就是由扩散协议完成的。在扩散协议当中通过发送和接收链路状态更新分组来实现。

1. 链路状态更新分组的发送

当一个路由器检测到它的一个邻居的状态发生了变化的时候，会立即更新自己的链路状态数据库中的相应记录，并将更新后的 LSA 装在链路状态更新分组中发给与自己相连的其它节点。在一个链路状态更新分组中可能包含有多个 LSA，它的传送距离只有一跳（hop）。为了保证这个算法的可靠性，发送方在发出更新报文后，必须等待接收方发来的确认。如果，发送方在规定的时间内没有收到确认，它会以一定的时间间隔重新发送更新分组，直到收到对方的确认为止。

2. 链路状态更新分组的接收

当一个路由器收到邻居发来的链路状态更新分组后，它会采取如下的一些措施：

(1) 在数据库中搜索相应的记录；

(2) 如果该记录不存在，就把它加入数据库，并广播该报文；

(3) 如果该记录比自己数据库中的记录新，则替换数据库中的记录，并广播该报文；

(4) 如果数据库中的记录新，就把这个更新的记录告诉发方；

(5) 如果和数据库中的记录一样，则不做任何操作。

在这里面我们涉及到比较相同的 LSA 的新旧问题，此时，我们是通过比较它们的链路状态序列号（LS sequence number）、LS Age 和 LS Checksum 来实现的。

通过这 3 个协议的相互协调工作，路由器上的链路状态数据库得以形成，并随时更新，以保证自己的链路状态数据库所描述的网络拓扑是最新的。每当链路状态数据库发生了变化的时候，我们就必须重新生成最短树，然后根据它重新计算出自己的路由表，以完成 OSPF 路由选择协议的全部功能。

第三章　程序设计总体框架

接下来的工作就是要实现 OSPF 路由协议，完成其中所描述的功能，最终生成路由表。

整个程序的实现是以教研室已开发了的路由器作为平台，在此基础上，加入一个相对独立的模块来实现 OSPF 的功能。路由器中已有的软件为它提供基本的通信服务以及其它的一些信息，而这个 OSPF 模块提供相应的接口供路由器上的其它软件调用它。

下面，我将从整个路由器软件的开发情况、OSPF 路由协议模块的细分、OSPF 模块中的数据结构以及 OSPF 模块怎样与整个路由器软件进行衔接等方面来简要的介绍一下程序设计的总体框架。

3.1　程序的开发环境

既然我们所开发的 OSPF 路由协议软件是以已有的路由器作为平台，那么，我们首先需要了解的就是已有的路由器软件的开发环境、它的结构以及已经实现了的功能。

路由器软件的开发和调试是采用可裁减的 pSOSystem 嵌入式实时多任务操作系统。

3.1.1　pSOSystem 开发环境

pSOSystem 是一个高性能的实时操作系统，它提供了基于开放系统标准的多任务环境。pSOSystem 采用模块化的结构，其系统库提供了以实时多任务内核为核心的若干组件，每一个组件提供一类服务。PSOSytem 软件结构如图 3-1 所示。

在这个结构中：

1. pSOS＋：pSOSytem 实时多任务操作系统的内核，提供任务的管理、调度，任务间的通信、同步，内存的分配、管理等服务；
2. pNA＋：TCP/IP 网络管理器（TCP/IP Network Manager），提供 TCP/IP 网络服务，包括 ARP、IP、ICMP、TCP/UDP 等协议和标准的 socket 接口；
3. pRPC＋：远程进程调用库（Remote Procedure Call Library）；

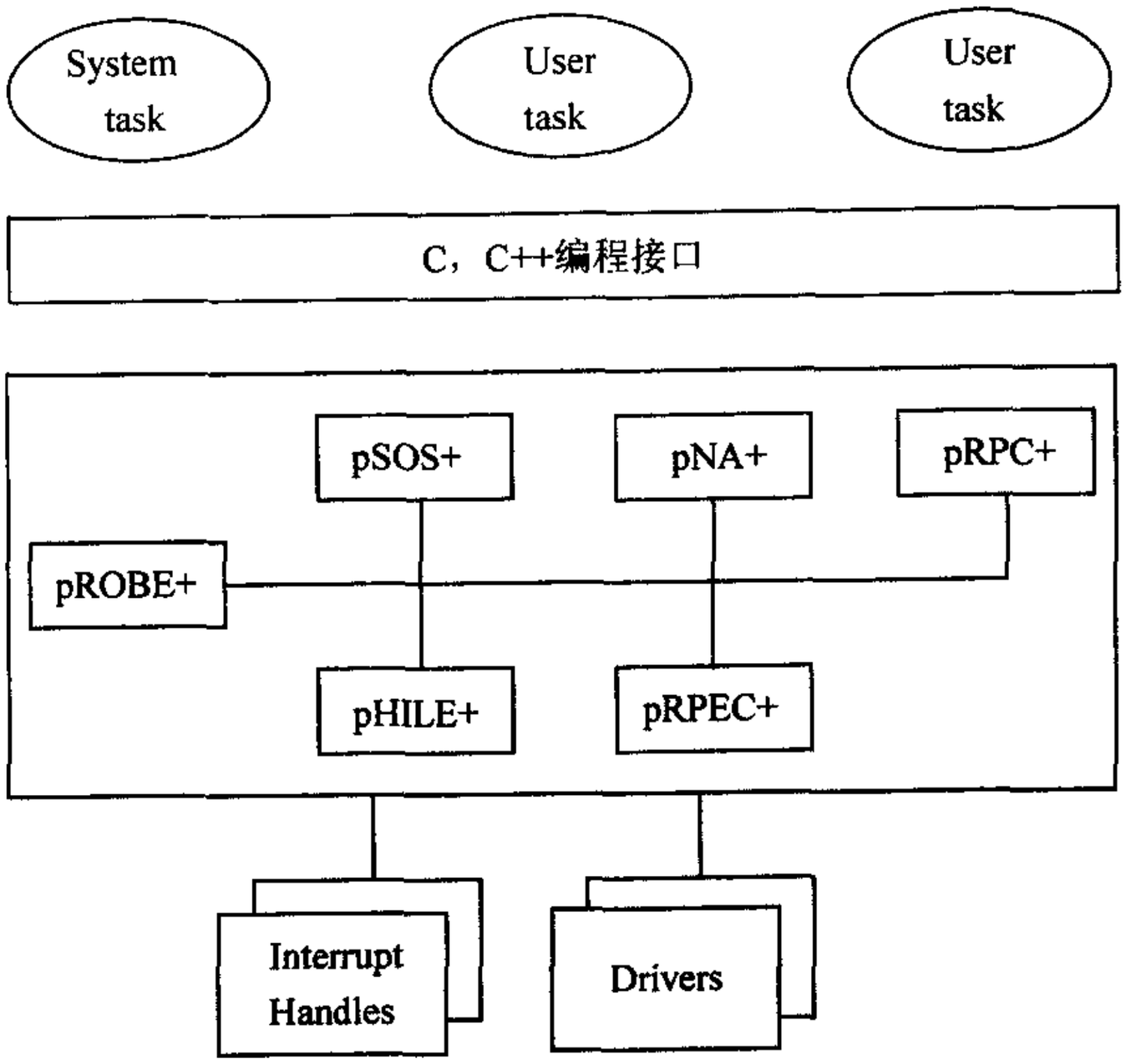


图 3-1　pSOSytem 软件结构

4. pHILE＋：文件系统管理器（File System Manager），提供本地或网络文件的访问；

5. pRPEC＋：ANSI C 标准库；

6. pROBE＋：提供有关调试的服务；

利用 pSOSytem 提供的这一系列服务，用户可以自己选择所需要的组件，将其包含在目标代码中，通过库函数调用的方式来使用这些组件提供的服务。

使用这样的系统，我们在主机上开发应用程序，在目标机上运行可执行的映像文件，这个映像文件包括 pSOSytem 软件和自己的应用程序代码。在调试的时候，我们可以采取两种方式：直接在目标系统上调试或在开发机上进行远程调试。当选择进行远程调试的时候，又可以根据实际情况选择通过 Console 口或利用 pNA＋组件通过 TCP/IP 网络进行调试。

根据所需开发的路由器软件的特点，为了开发一套具有自主知识产权的软件，为了更有利于今后软件的升级和优化，在实际进行路由器软件开发的时候，我们只利用了 pSOSytem 系统的内核（pSOS＋）和 pRPEC＋、pROBE＋、pHILE＋，自己设计了一套 TCP/IP 协议栈的软件。

3.1.2 路由器软件结构

整个路由器软件的体系结构如图 3-2 所示。

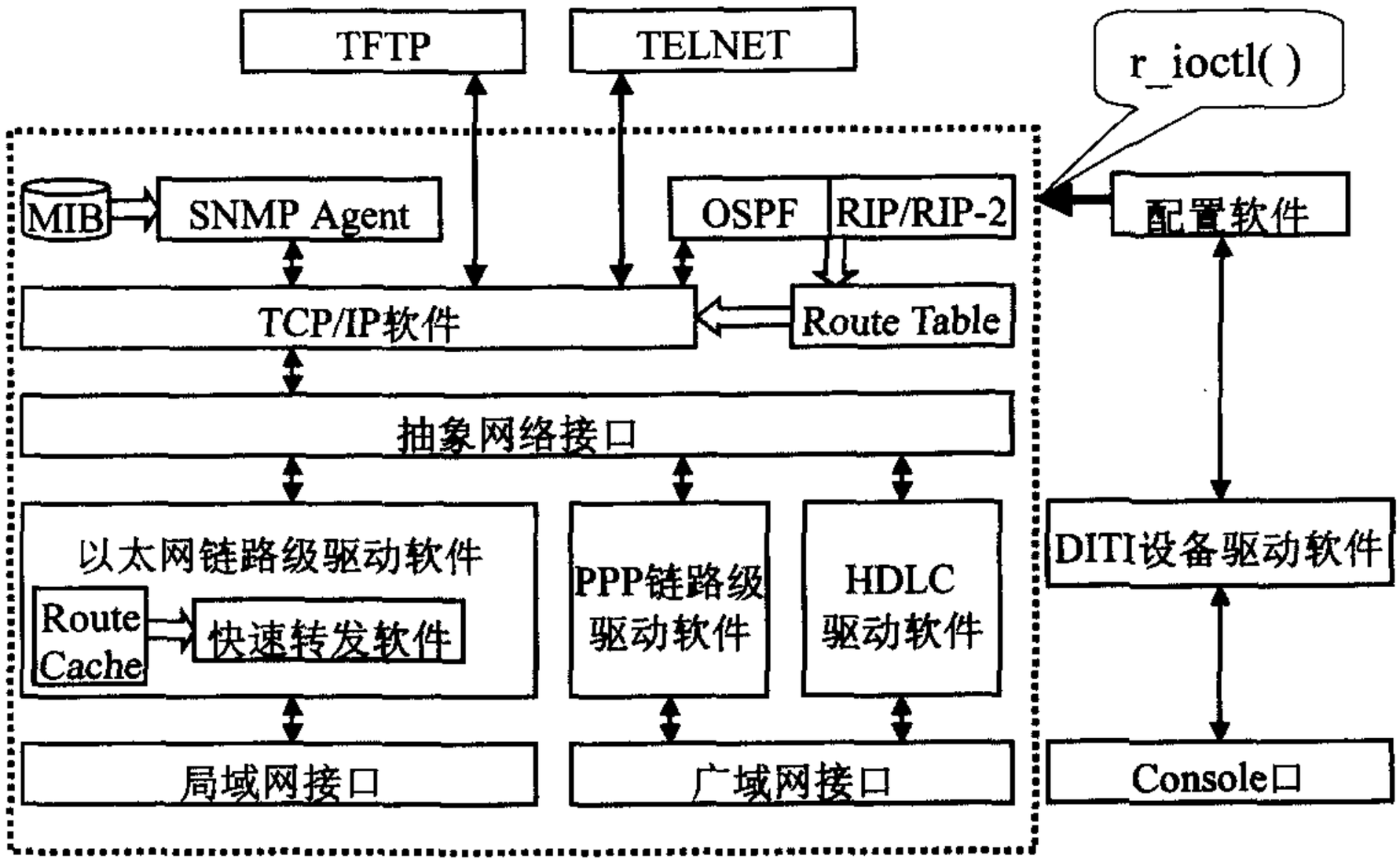


图 3-2 路由器软件结构

1. 在这个结构中，划虚线的部分为路由器软件的核心部分。其中：

(1) 局域网接口主要支持 Ethernet 接口，指 Ethernet 的接口驱动程序；

(2) 广域网接口是指广域网的接口驱动程序；

它们直接与物理网卡打交道。在其上覆盖了相应的链路曾驱动软件。

(3) 抽象网络接口是 IP 层与下面物理网络接口的抽象。该接口屏蔽了具体的物理网络实现细节，使上层（IP）所看到的各种网络都具有完全相同的接口。广域网、局域网在这里的接口是完全相同的。这样，当下层软件发生变化或增加新的下层软件的时候，上层可以不做修改。

(4) TCP/IP 软件包括 TCP/IP 协议族软件和 socket 接口，以此来代替 pSOSytem 中提供的 pNA＋通信模块。

(5) 位于 TCP/IP 软件之上的是一个 SNMP（简单网络管理）的代理，管理员可以通过它来读取路由器设备的一些参数，对设备的状态和参数进行设置。

(6) OSPF/RIP 模块利用 TCP/IP 软件提供的通信服务，与网络中其它的路由

器进行通信，通过计算得到一张路由表，TCP/IP 软件则通过查找这张路由表来决定一个 IP 数据报的转发路径。

2. 位于虚线外的模块则是路由器所提供的一些服务和为了让这个路由器正常运行而进行配置的模块。

3.1.3 现有路由协议软件

路由协议的实现，构成了路由器软件上的一个重要的模块——路由模块。从路由器的软件结构可以看出，路由模块是路由器软件中重要的一个部分，它通过 TCP/IP 软件获得信息，又利用获得的信息生成路由表，从而来影响 TCP/IP 软件的工作。路由模块的结构如图 3-3 所示。

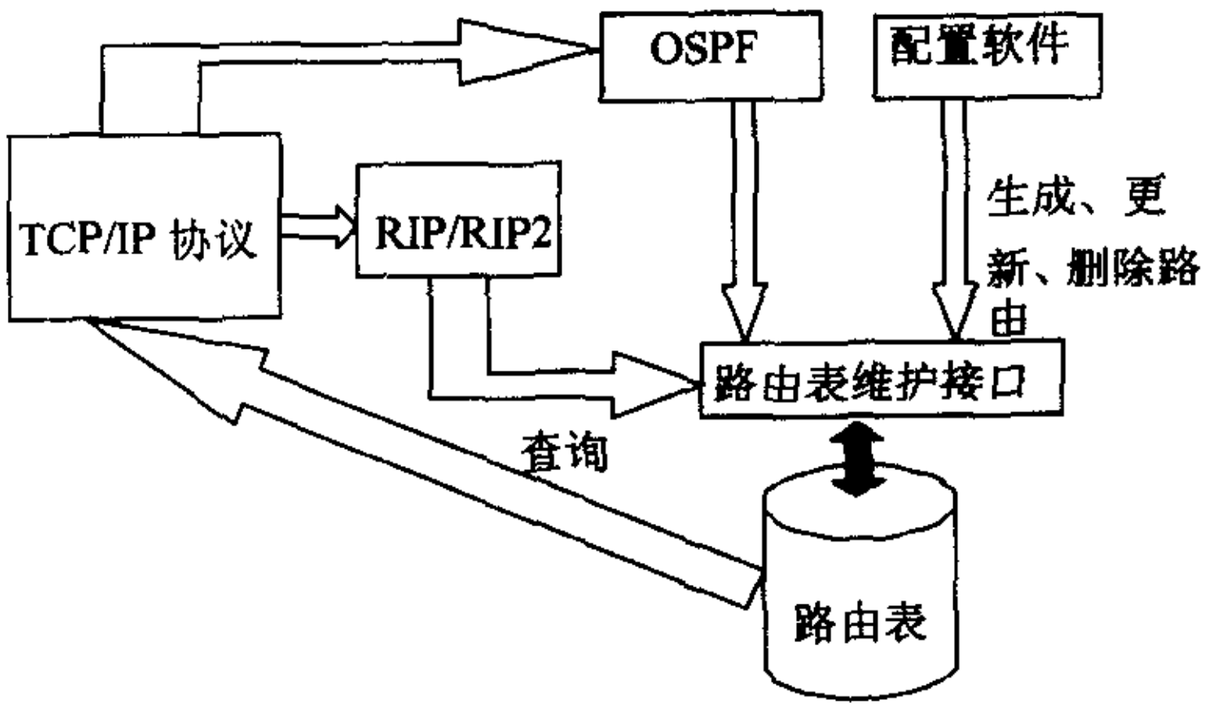


图3-3 路由模块结构

在路由模块中我们已经实现了静态路由和 RIP/RIP2 路由协议。它们已能相互配合完成一定的功能。

1. 静态路由是由网络管理员在路由选择前就建立的，除非网络管理员干预，否则静态路由不会发生变化。其实现是由管理员通过配置模块设置到路由转发表中去的，同样管理员还可以将不正确的静态路由删除。
2. 对于 RIP 协议的实现，我们是按照 RFC1058（RIP）和 RFC1723（RIP2）的规定来进行的。实现了 RIP 和 RIP2 的的兼容。

目前，我们所需要做的工作就是实现 OSPF 这种路由协议软件，以加强路由器的路由选择功能，适应网络的迅速发展。

3.2 OSPF 路由协议模块的划分

OSPF 协议软件的开发，是按照 RFC2328（OSPF Version2）的规定来进行

的。

根据前面的分析，我们可以将整个 OSPF 模块划分为以下几个部分：

1. 通信子块：通过它与 TCP/IP 软件打交道，进行数据的收和发，并完成相应的数据处理。利用 TCP/IP 软件提供的服务，为其它子块提供统一的通信接口，屏蔽掉通信的具体细节。根据 OSPF 协议所描述的功能，我们可以将通信子块进行进一步的细分：

   (1) Hello 协议子块：完成 Hello 协议所描述的功能。

   a. 接收 Hello 分组，检查其中的内容，以发现新邻居的存在，判断邻居之间应建立的状态关系以及已建立的状态是否应发生改变，从而向邻居有限状态机发送相应的信号。检查分组中相应的内容，发送信号给接口有限状态机；

   b. 根据不同的网络接口类型，以不同的形式发送 Hello 分组，告知邻居自己的存在；

   c. 收集邻居路由器的优先级、路由器 ID 等信息，以便在广播型网络和 NBMA 网络上选举指派路由器和备份指派路由器。

   (2) 交换协议子块：完成交换协议所描述的功能。在应建立连接关系的路由器之间，启动并完成链路状态数据库的初始同步。

   (3) 扩散协议子块：完成扩散协议所描述的功能。

   a. 接收更新分组，并将接收到的更新分组中的 LSA 放入链路状态数据库中；

   b. 在链路状态发生变化的时候及时更新链路状态数据库的内容，产生并发送更新分组，向自己的邻居通报这一变化；

2. 指派路由器选举子块：在广播型网络和 NBMA 网络上根据从 Hello 分组中收集到的信息，完成指派路由器和备份指派路由器的选举。

3. 邻居有限状态机：标明了邻居可能的各种状态，接收其它模块发送来的信号，实现各种状态之间的相互转换。

4. 接口有限状态机：标明了接口可能的各种状态，接收其它模块发送来的信号，实现各种状态之间的相互转换。

5. 链路状态数据库：实现链路状态数据库的生成和维护。在数据库的内容发生变化的时候，调用生成树算法，重新计算最短树，从而生成新的路由表。

6. 生成树算法：根据链路状态数据库的内容，按照 Dijkstra 算法，生成一个以自己为根的最短树，并根据这棵树计算出路由表。
7. 配置子块：用户可以通过它来控制 OSPF 模块。通过对 OSPF 模块中的某些数据结构的配置，以保证该模块的正常运行。同时，用户还可以通过它来了解 OSPF 模块运行的状态。该子块是 OSPF 模块中的人机接口。

按照上述的分析，我们可以用图 3-4 来表示个子块之间的相互关系。

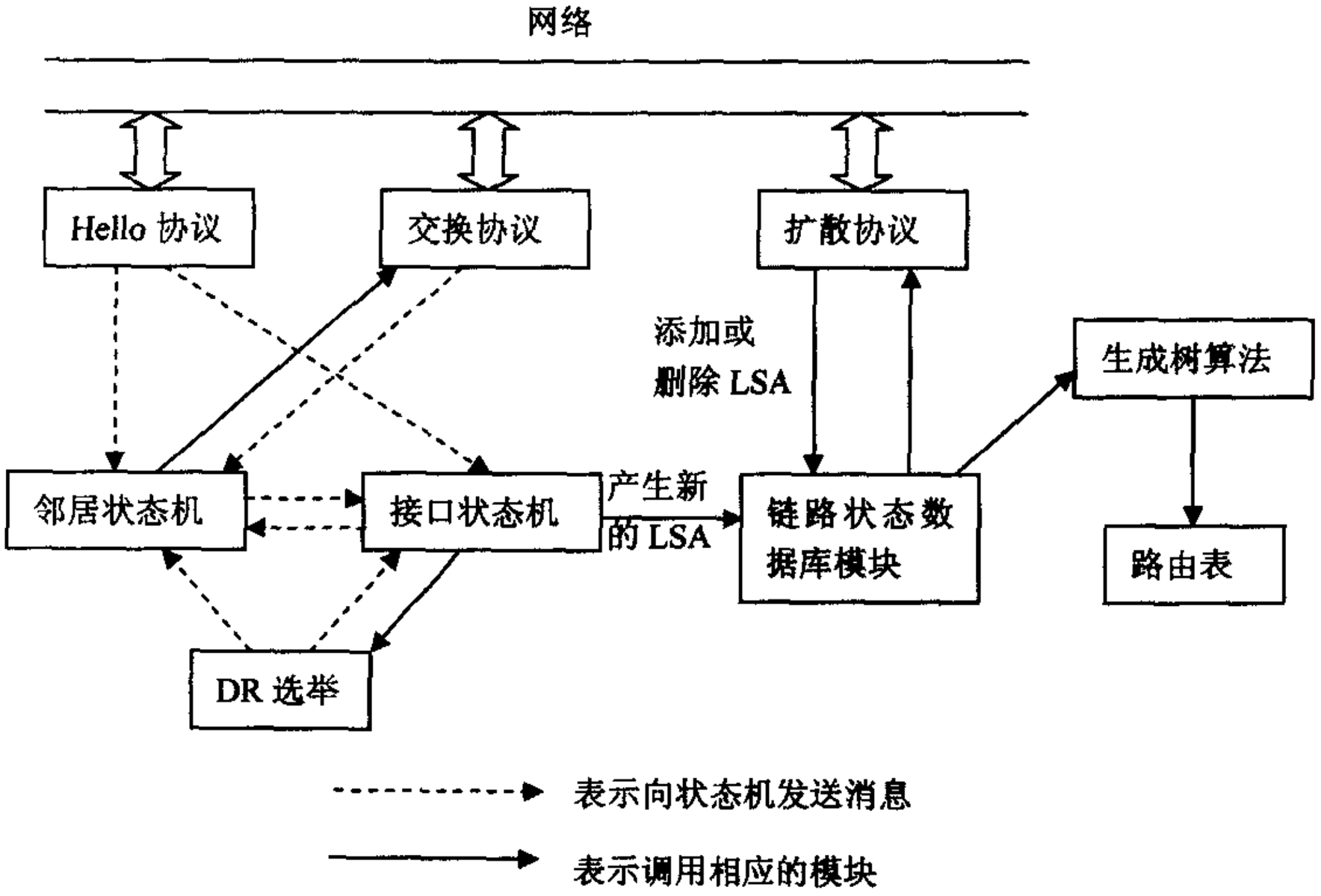


图 3-4 OSPF 模块的结构

注意，在图 3-4 中，我们并没有标出配置子块与其它各子块的相对位置以及关系，这是因为其它子块的运行是建立在一系列已有的数据结构的基础之上，而这些数据结构的建立以及其值的改变，均是由配置子块来完成的。

3.3 主要数据结构

数据结构是 OSPF 协议运行的重要基础，因此，怎样将各种不同的信息进行区分和组合，怎样合理的安排各种数据结构之间的关系就成了我们的一个重大问题，这是在进行程序设计时首先需要考虑的问题。

首先，我们需要邻居数据结构，来存放邻居的某些信息。

其次，每个路由器上有多个接口，每个接口接在一个网络上，那么这个接

口就可能与其它路由器的某个接口成为邻居，当然这个邻居可能不止一个。这就说明邻居关系是依存于路由器的某个接口的。

第三，一系列的网络、主机以及有接口与这些网络相连的路由器形成了一个区域，因此，对于路由器而言，划归它属于某个区域时，应该时将它的某个接口划归该区域。一个区域可能包含同一路由器的多个接口或不同路由器的不同接口。

最后，我们需要一个总体的结构将这些区域联系起来。

根据上面的分析，我们就形成了一个如图 3-5 所示的具有层次关系的数据结构：

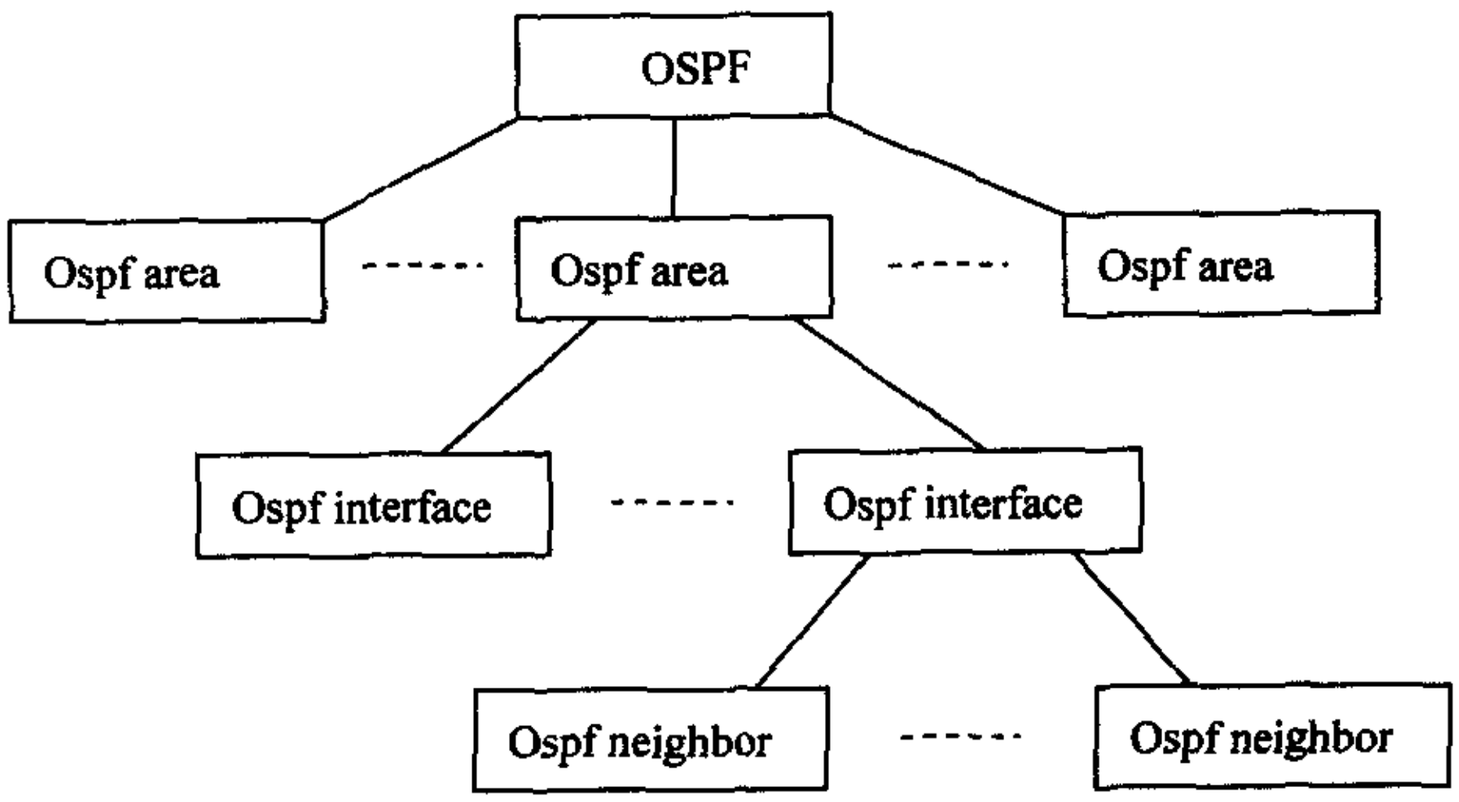


图 3-5　OSPF 数据结构示意图

因此，在全局数据结构里面，就有 4 个主要的数据结构：

1. struct ospf
2. struct ospf_area
3. struct ospf_interface
4. struct ospf_neighbor

它们之间的具体关系如图 3-6 所示：

struct ospf 是一个协议结构，在该结构中包含了运行 OSPF 所必需的一些整体信息。如：路由器 ID、区域链表以及在该路由器上运行 OSPF 协议所生成的路由表等。一个 OSPF 模块只有一个 struct ospf 结构。

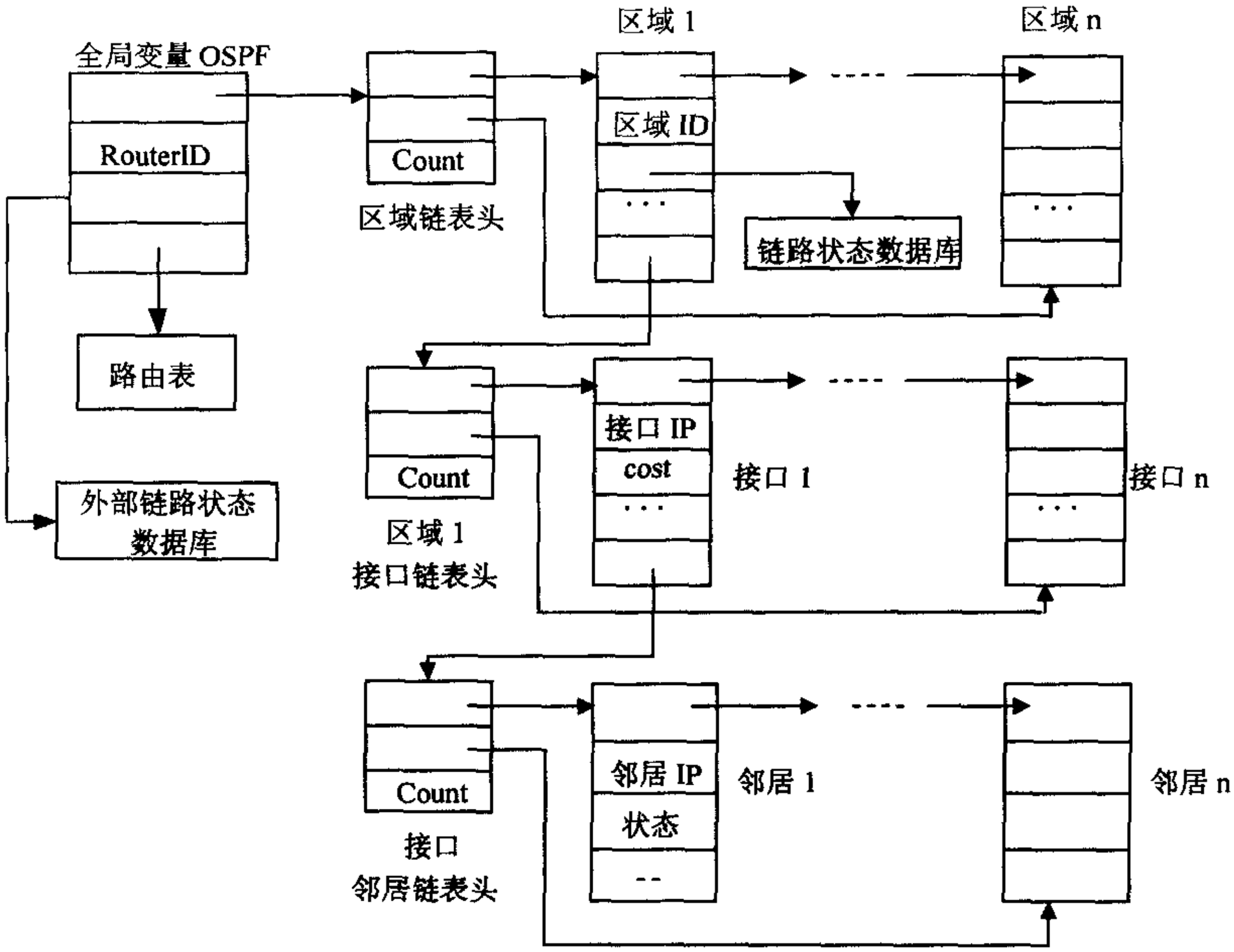


图 3-6　OSPF 中主要数据结构关系

```
struct ospf
{
    uint32 router_id;   /*   路由器 ID   */

    UCHAR type;       /*   路由器类型   */
#define OSPF_NONE 0
#define OSPF_ABR 1
#define OSPF_ASBR 2

    struct ospf_area *backbone;   /*   指向主干区域的指针   */
    list areas;          /*   区域链表   */
    list vlinks;         /*   虚链路链表      */

    struct ospf_interface *iflist[MAX_IF_NUM * 2];   /*接口链表数组  */

     ……
```

```
    /*   external LSDB   */
    struct ospf_lsdb *external_lsdb;   /*   外部链路数据库   */
    struct route_table *table;   /*   路由表   */
};
```

struct ospf_area 是区域结构，它描述了一个 ospf 区域的一些共同特征。在一个 OSPF 路由器上可以有多个 ospf_area 数据结构，它们是由 struct ospf 组织起来。

```
struct ospf_area
{
    uint32 area_id;          /*   区域 ID   */
    struct ospf *top;        /*   指向该区域所属的协议数据结构   */

    list ifaces;             /*   属于该区域的接口链表   */
    list address_range;      /*   该区域的地址范围   */

    UCHAR external_routing;  /*   该区域的外部路由能力   */

    uint32 default_cost;     /*   stub 区域的 cost   */

    UCHAR auth_type;         /*   该区域所采用的验证类型   */
union
{
    ......
}u;                          /*   验证数据   */

 /*   区域内的链路状态数据库   */
struct ospf_lsdb *router_lsdb;
struct ospf_lsdb *network_lsdb;
struct ospf_lsdb *summary_lsdb;

/*   进行 SPF 计算的参数   */
uint32 spf_hlodtime;
uint32 spf_delay;
    struct spf *tree;
     ......
};
```

struct ospf_interface 是接口结构，描述了路由器的某一个接口的信息。一个

接口对应一个该结构。

```
struct ospf_interface
{
    struct ospf *ospf;
    struct ospf_area *area;
    struct ospf_vl_data *vl_data;   /*   虚链路接口数据   */
    list neighbors;                 /*   邻居链表   */
    uint32 ip;
    uint32 netmask;
    uint32 ifnum;    /*   接口号 */
    uint32 mtu;      /*   该接口的 MTU */
    int fd ;     /*     传输数据时所用的 socket   */
    UCHAR priority;    /*   接口的优先级   */
    int status;  /*   接口状态   */
    UCHAR type;        /*   接口类型   */
    uint32 transmit_delay;  /*   传输延迟   */
    uint32 output_cost;  /*   传输一个分组的度量   */
    uint32 retransmit_interval;  /*   重传时间间隔   */
    uint32 hello_interval;  /*   发送 Hello 分组的时间间隔   */
    uint32 dead_interval;  /*   检测邻居死亡的时间间隔   */
    ……
    UCHAR auth_type;  /*   0 - no authentication,1 - simple authentication
*/
        ……
        uint32 d_router;   /*   该接口所接网络上的 DR   */
        uint32 bd_router;   /*   该接口所接网络上的 BDR   */
         /*   自己产生的 LSA   */
        struct ospf_lsa *network_lsa_self; /* network-LSA. */
        struct ospf_lsa *summary_lsa_self;      /* summary-LSA. */
         ……
};
```

struct ospf_neighbor 是邻居结构，描述了该路由器的某个邻居的信息。

```
struct ospf_neighbor
{
  struct ospf_interface *iface;   /*   该邻居所属的接口   */
  UCHAR status;              /*    邻居状态   */
  UCHAR dd_flags;            /*   在 DD 中，该邻居是 Master/Slave     */
  uint32 dd_seqnum;          /*    DD 序列号   */
  /*   有关邻居具体信息   */
  uint32 address;
```

```
uint32 netmask;
uint32 router_id;
UCHAR options;
UCHAR priority;
uint32 d_router;
uint32 bd_router;

struct ospf_packet *last_send;     /*   上一次发送的 DD 报文   */
  /*   在 DD 交换中所需要用到的数据   */
  struct ospf_lsdb ls_retransmit;
  struct ospf_lsdb ls_request;
  struct ospf_lsdb db_summary;
  struct ospf_lsa *request_last;
        ......
};
```

对于这些数据结构的详细说明见附录。

3.4 OSPF 主体程序设计

在设计了程序的数据结构以后，我们开始考虑程序的主体设计。

考虑到 OSPF 对实时性的要求不是很强，我们在程序中采用消息队列的方式，用两个任务（task）来完成它的功能。一个任务用于接收和处理消息（OspfTask），另一个任务用于从网络上接收 OSPF 数据（OspfReader）。

在程序初始化的时候，我们完成如下的操作：

1. 设置一个消息队列；
2. 设置 OspfTask，并启动之；
3. 设置一个 Timer，并启动之。该 Timer 的调用是由系统来完成。

对于 OspfTask 这个任务，其运作如图 3-7 所示。

对于每一个消息，我们用一个长度为 4 的 ULONG 数组来表示。这 4 个 ULONG 字段标明了消息的类型，以及执行相应的处理所需要的参数。在这里，共有 3 类消息：

1. 定时器消息：系统每当定时到的时候（每 1000 毫秒），就会往消息队列中放一个定时器消息。在定时器消息的处理函数中，检查 timer 队列中有没有定时到的 timer，若有就执行相应的处理函数或产生相应的事件。
2. 收到 OSPF 报文：OspfReader 在检测到有 OSPF 报文到来后，会往消息

队列中放一个消息，以通知 OspfTask，由它调用相应的函数去接收并处理新报文。

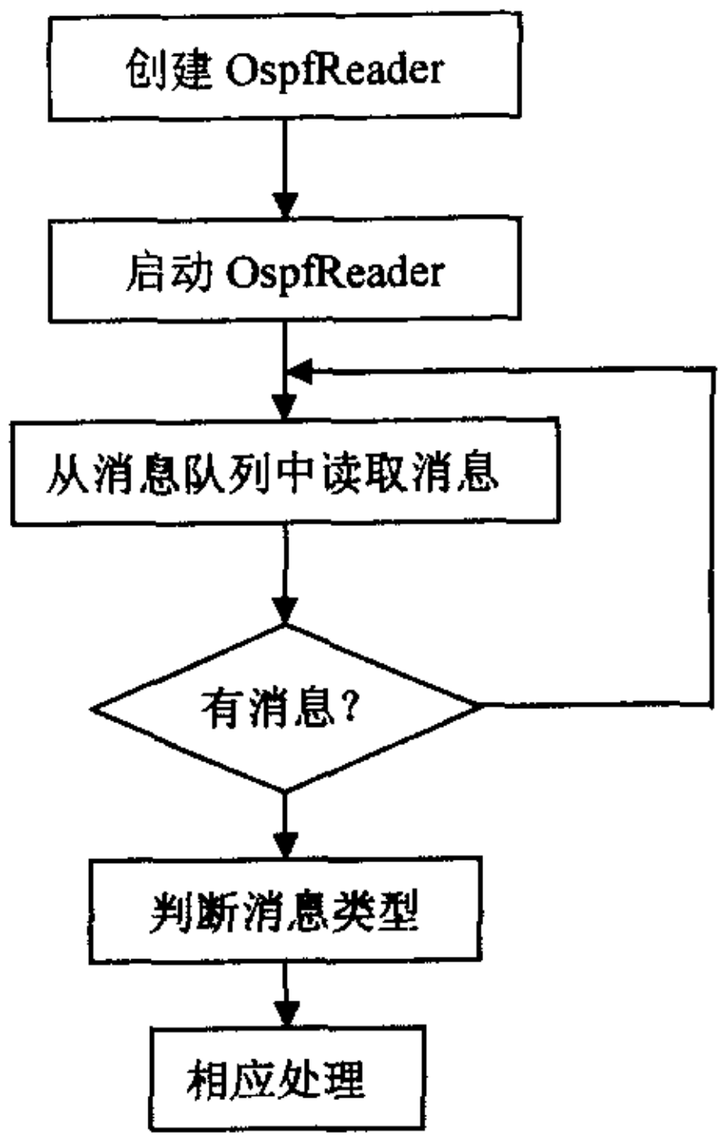


图 3-7　OspfTask 的处理流程

3. 收到 OSPF 命令：系统在收到一个 OSPF 配置命令时，只是往消息队列中放一个消息，告诉 OspfTask 有配置命令来了，而不对它进行具体的操作，其执行在消息处理函数中实现。

3.5　软件接口关系

OSPF 模块是一个较独立的模块，它利用其它模块提供的一些接口，同时也提供一些接口供其它模块来调用它，从而与整个软件相融合。

1. OSPF 在发送和接收报文的时候，需要用到其它模块提供的接口，来使用网络的通信服务。它所使用的是 raw_socket，我们用如下的接口函数来建立这样的 socket：

```
s = socket(AF_INET, SOCK_RAW, 89);
```

这里的 89 是 OSPF 协议的编号，它表明这个 socket 收和发的数据都是 OSPF 数据。

2. 接口状态机提供接口以接收从其它模块传送来的信息，如从底层协议送来地 InterfaceUp 或 LoopInd 的指示，。

3. 使用组播

OSPF 分组的发送是以组播的形式来实现的，因为这样做可以实现更高的效率。要使用组播就需要使用其它模块提供的接口函数。

在这里，首先我们需要把一个接口加入到组播地址 224.0.0.5（所有 OSPF 路由器）中；其次，如果该路由器为指派路由器或备份指派路由器还要把它加入到组播地址 224.0.0.6（所有指派路由器）中。其实现的过程是这样的：

```
#define   ALL_OSPF_ROUTER   0xe0000005   （224.0.0.5）
#define   ALL_DR_ROUTER   0xe0000006   （224.0.0.6）
```

如将一个接口号为 if_num 的接口加入到组播地址 224.0.0.5 中：

```
MMAPREQ   mma;
mma.prAddr = ALL_OSPF_ROUTER;
r_ioctl(if_num,SIOCADDMCAST,&mma);
```

4. 配置接口

在这一步完成之后，该接口就可以利用这个接口进行数据的收发了。

它是 OSPF 模块提供给外界的接口，对于它的介绍将放在下面一章。

第四章　配置命令解析及实现

4.1　配置命令的功能和作用

配置命令是 OSPF 模块与管理人员之间的接口。为什么需要这个接口？如果没有这个接口，又会出现什么样的情况呢？

我们已经谈到，OSPF 模块是一个相对独立的模块，那么由谁来调动它的运行呢？当然，我们可以在路由器一开始运行的时候就启动它的运行，这样就不需要人为的控制了，但是这样做显然少了灵活性。因此，我们还是需要配置命令，让它来控制 OSPF 的运行。

OSPF 模块运行起来以后，会进行区域的划分，将不同的接口划归不同的区域，那么怎么决定一个接口究竟该属于哪个区域呢？对于区域的划分，OSPF 模块光靠自己的信息是没有办法的，这就需要管理人员进行操作。在区域划分好之后，位于同一个区域中的接口应该共同遵循一些规则，如：应具有相同的验证数据、应对区域的类型（是否为 stub 区域）达成一致等。这样的一些数据都是 OSPF 模块自己没有办法得到的，因此，也必须要有管理人员的参与。

对于一个接口而言，它传送一个分组所需花费的代价、发送 Hello 分组的时间间隔、重发 LSA 的时间间隔以及检测邻居是否还存活的时间间隔等，虽然也可以在启动 OSPF 协议运行的时候载入，但是这样做同样少了灵活性。如果能够让管理人员参与配置和管理这些参数，他们就可以根据网络的实际状态或得到的某些信息来进行动态的调整，或者可以通过这样的手段来测试这些参数对网络性能的影响，从而得到最佳数值。这样，我们就可以让网络工作在最佳状态下，让它发挥其潜能。

通过前面的叙述，可以得知，为了让 OSPF 模块正常的工作，管理人员必须参与对其中的参数进行配置的工作。那么，管理人员怎样知道配置的结果是什么呢？除此之外，为了管理整个网络，管理人员必须随时了解网络的连接情况，于是他们必须知道一个路由器有哪些邻居，而在 OSPF 模块运行的过程中，邻居的发现是动态的，管理人员是通过什么样的手段来得知这些邻居的情况的呢？同样，当他需要了解链路状态数据库以及其它一些内容的时候，他又是通过什么样的一些方式呢？所有的这些，都可以由配置命令来完成，管理人员按一定的格式输入命令，告诉路由器自己想要知道的内容，路由器在执行了这些配置命令后，会将结果显示在屏幕上，以满足管理人员的请求。

4.2 配置命令分类

通过了解各种不同的配置命令的功能和作用，我们可以将配置命令分为三类：

1. OSPF 配置命令：通过它可以控制 OSPF 协议的运行与否，可以对区域进行划分，以及对区域中的参数进行设置。这些命令这要是对 struct ospf_area 结构中的参数进行配置。其中包括：
   (1) start：在路由器上启动 OSPF 协议的运行。
   (2) stop：在路由器上停止 OSPF 协议的运行。
   (3) area：设置区域中的一些参数。在该命令下还包括很多子命令：
      a. authentication：在区域内起用身份验证；
      b. cost：若该区域为 stub 区域，则设置该区域的汇总路由得默认成本；
      c. nssa：将一个区域设置成 NSSA 区域。这种类型的区域与 stub 区域类似，唯一不同的就是在 NSSA 区域中，类型为 3 的汇总链路状态广告会被导入，而 stub 区域则不会；
      d. range：如果该路由器为 ABR，则该命令的使用会在区域边界统一和汇总路由，以便向外发布；
      e. stub：将一个区域设置成 stub 区域；
      f. virtual_link：建立一条虚链路，以保证主干区域的全连通性。
2. OSPF 接口命令：通过它可以控制 OSPF 协议是否运行在某个接口上，可以对 *struct ospf_interfce* 结构中的参数进行设置。其中包括：
   (1) authentication：配置适用于接口的纯文本身份验证的 password。
   (2) cost：配置在一个接口上发送一个分组的 cost。
   (3) dead_interval：配置邻居之间检测存活的最大时间限制，如果在这段时间内该接口都没有收到来自邻居的 Hello 报文，则表示该邻居已经 Down 掉。这个值若设置的过大，则会影响协议的灵敏度；若过小，则会使网络的拓扑发生频繁的不正确的变化，从而浪费网络带宽；
   (4) hello_interval：配置该接口发送 Hello 报文的时间间隔；
   (5) retransmit_interval：配置在该接口上重发丢失的链路状态宣告的时

间间隔；

(6) transmit_delay：配置链路状态发送延迟，即在一个接口上发送链路状态更新报文所用的时间。

(7) priority：设置该接口的优先级，以便在广播型网络和 NBMA 网络上选举指派路由器。

3. OSPF 监控命令：这些命令的设置是为了方便管理人员了解网络动态变化的情况。包括如下的内容：

(1) show ospf：显示 OSPF 协议整体运行的情况；

(2) show ospf interface：显示运行 OSPF 协议的某一接口的信息；

(3) show ospf neighbor：显示某一接口的邻居的信息；

(4) show ospf database：显示链路状态数据库的信息；

(5) show ospf border：显示 OSPF ABR 的信息。

4.3 配置命令设计

我们将配置命令设计为一个层次结构，如图 4-1 所示。

在图 4-1 中列出了所有有关 OSPF 配置的命令，而对于其它的命令则选取了一部分。

由于 OSPF 是一种路由协议，故我们将它和 RIP 一起挂在对路由器进行配置的子命令下。这也说明了 OSPF 配置命令是在 router 这一级子目录下。

在对接口进行 OSPF 配置的时候，需先进入对接口进行配置的一级子目录。由于 OSPF 协议属于 IP 协议中的一种路由选择协议，故，它又应该挂在 ip 这一级目录下。

采用这种层次型的配置方式，更有利于用户理解各种命令的作用，也极大的方便了他们的使用。

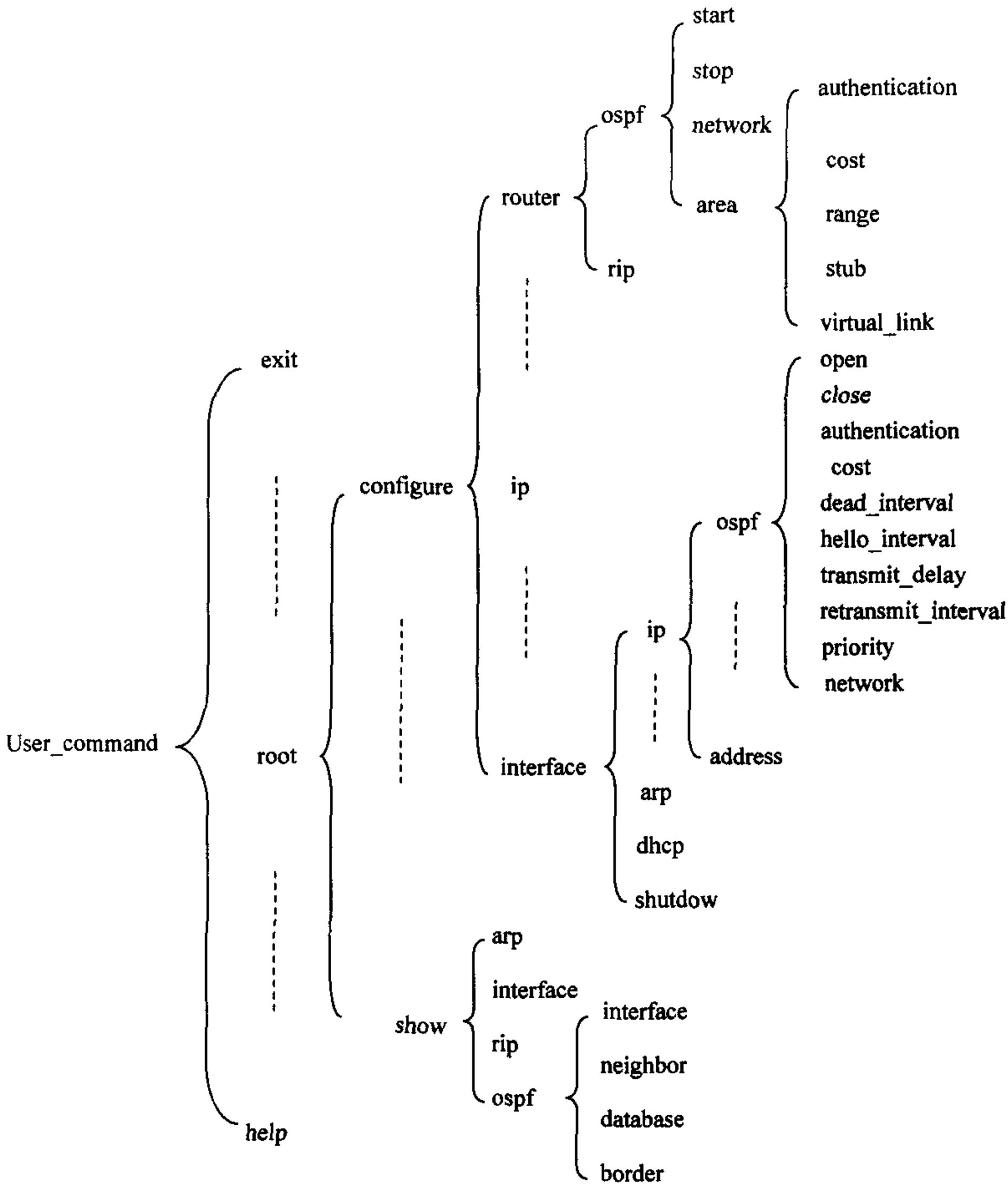


图 4-1　配置命令层次关系

4.4　配置命令的实现

目前，本系统的配置可以采用两种方式：

1. 用户可以通过 Telnet 的方式登上目标机，将配置信息传送给目标机；
2. 用户可以通过串口将配置信息传送给目标机。

配置信息的通信则依靠 pSOS 提供的 DITI 设备接口软件，通过它将用户送

来的配置信息交给系统的命令解释程序，然后调用相应的命令处理函数，以完成配置命令的功能。

命令解释程序收到用户传来的配置信息后，它怎么知道该怎样去查找与之相对应的命令处理函数？我们可以将命令处理函数与命令的名称对应起来，这样命令解释程序就可以根据它们之间的对应关系来查找命令处理函数。那么，我们应该怎样将它们对应起来呢？对应起来后，我们又应该怎样去管理这些命令的名称呢？

首先，我们用一个数据结构 struct cmds 将命令的名称与命令处理函数以及其它的一些相关信息对应起来。

```
struct cmdstr
{
    char *name;                    /*  命令的名称  */
    int (*func)(int argc,char *argv[],void *p);  /*  命令处理函数  */
    void (*helpfunc)(int argc,char *argv[]);  /*  帮助函数  */
    short  minchar;  /*  在输入该命令名称时，所需要输入的最少字母数
*/
    short argcmax;  /*  该命令所带参数的最大个数  */
    short argcmin;  /*  该命令所带参数的最少个数  */
    short *showmsg;  /*  当用户需要帮助，输入“？”时，给予的提示  */
};
```

将同属于一个命令下的各个子命令组织起来，形成一个 struct cmds 结构的命令数组。命令解释程序在识别命令的时候，只需要根据用户传来的命令行参数的第一个参数，查找相应的命令数组，找到后将剩余的参数交给下刚找到那个子命令，或直接调用命令处理函数，或继续查找直到找到其相对应的命令处理函数，从而执行之。例如，我们在组织 OSPF 配置命令的时候就采用了如下的数据结构：

```
struct cmds Cnfg_Ospf_Cmds[] = {
/*  name        func()                    help                    char    argc
showmsg  */
"start",      do_ospf_start,      help_cr,                3,      2,1,
        "start   Start OSPF Task.",
"stop",       do_ospf_stop,       help_cr,                3,      2,1,
        "stop    Stop OSPF Task",
"area",       do_ospf_area,       help_ospf_area,         1,      6,1,
```

```
        "area    Config OSPF Area",
"network",   do_ospf_network,   help_ospf_network,    2,        6,6,
        "network     Config OSPF Network",
"quit",        do_quit,              help_cr,              1,        2,1,
        "quit     Exit to root commands",
"no",          no_ospf,              help_no_ospf,         2,        2,0,
        "no     Negate the command",
0,             0,                    0,                    0,        0,0,
        0
};
```

在程序主体设计中我们谈到，配置命令的实现是在命令解析函数找到相对应的命令处理函数后往 OSPF 消息队列中放一个消息，等到主体程序在读到这个消息时才真正的执行命令处理函数。对于监控命令，我们则采取收到命令后不往消息队列中放消息，而是直接调用主体程序中提供的命令处理函数的方式，以此来实现命令的直接处理并让它发挥作用。

第五章　接口有限状态机及其实现

配置命令的执行往往会造成接口状态的改变，从而引起接口有限状态机的执行，因此，配置命令子块和接口有限状态机子块有着密切的联系。

5.1　接口的状态

一个接口一共有 8 种状态：

1. ISM_Down

接口的初始状态，表明该接口不可用。

2. ISM_Loopback

测试网络用的。在该状态下，接口不能获得正常的数据流量，它可以通过发 ICMP 报文或一些测试比特来获得有关该接口的信息。

3. ISM_Waiting

网络接口可用后，接口状态由 Down 进入 Waiting。在此状态下此接口不能参加指派路由器和备份指派路由器的选举，它检查所收到的 Hello 分组中的 DR 或 BDR，以决定自己的 DR 或 BDR。只有经过规定的时间后，或者路由器在网络上发现了指派路由器，才能跳出该状态，参加 DR 和 BDR 的选举。

4. ISM_Point-to-Point

该状态表示网络是一个物理点到点网络或者虚链路。在这个状态下，路由器必须和邻居建立邻接关系。

5. ISM_DR other

该状态表明该接口接在一个广播型的网络或 NBMA 网络上，并且在该网络上，本路由器没有被选为 DR 或 BDR。此时，若网络上存在 DR 或 BDR，该接口就会试图与之建立邻接关系。

6. ISM_Backup

在一个广播型的网络或 NBMA 网络上，路由器被选为备份指派路由器。此时，它应与网络中所有的路由器建立邻接关系。

7. ISM_DR

在一个广播型的网络或 NBMA 网络上，路由器被选为指派路由器。此时，它应与网络中所有的路由器建立邻接关系。

8. ISM_DependUpon

表示下一个状态不能确定。如：当邻居状态发生了变化的时候，需要进行 DR 和 BDR 的重新选举，此时的接口就处于一个不明了的状态，只有等到计算结果出来以后，才能确定接口的状态。

5.2 引起接口状态发生变化的事件

接口状态发生变迁可能由 7 种事件触发，为了整个事件集的完备性，我们可以向接口状态有限机发送如下的 8 种消息：

1. NoEvent

这是一个无效事件。

2. InterfaceUp

该事件的产生是由于底层协议检测到该接口可用，这将使接口从 ISM_Down 状态中跳出；同时，这也有可能是 SPF 计算的结果，它算出该接口应该作为虚链路的一个端点，故也会产生该事件，使该接口有效。

3. WaitTimer

该事件是由 Wait Timer 触发,表示接口的等待时间结束，可以参加指派路由器的选举。

4. BackupSeen

该事件的产生是由于路由器检测到网络中存在备份指派路由器，或只有指派路由器而没有备份指派路由器，它标志着 Waiting 状态的结束。我们可以通过两种方式发现这些情况：从某个邻居处收到 hello 分组，声明自己为备份指派路由器；或者从某个邻居处收到 hello 分组，声明自己为指派路由器，并称该网络中没有备份指派路由器。但是，无论是哪一种方式，路由器的 ID 必须出现在邻居 hello 分组中，这就说明该路由器已与邻居建立了双向连接。

5. NeighborChange

此事件的产生说明网络上与该路由器建立了双向连接的邻居有所变化， 需要重新选举指派路由器和备份指派路由器。该变化可能是由以下的几种事件引起，它们都是通过检查 Hello 报文得到的：

(1) 与一个邻居建立了双向连接；

(2) 与一个邻居失去了双向连接；

(3) 某个双向连接的邻居新宣称自己为指派路由器或备份指派路由器；

(4) 某个双向连接的邻居不再宣称自己为指派路由器或备份指派路由器；

(5) 某个双向连接的邻居的优先级发生了变化.

6. LoopInd

该事件是由网管或底层协议发出的，指示该接口已经“Loop back to itself”。

7. UnLoopInd

该事件同样是由网管或底层协议发出的，指示该接口不再“Loop back to itself”。

8. InterfaceDown

该事件是由底层协议发送的，指示该接口不再是处于活跃状态，这会使接口的状态恢复到 InterfaceDown。

5.3 接口状态转移图

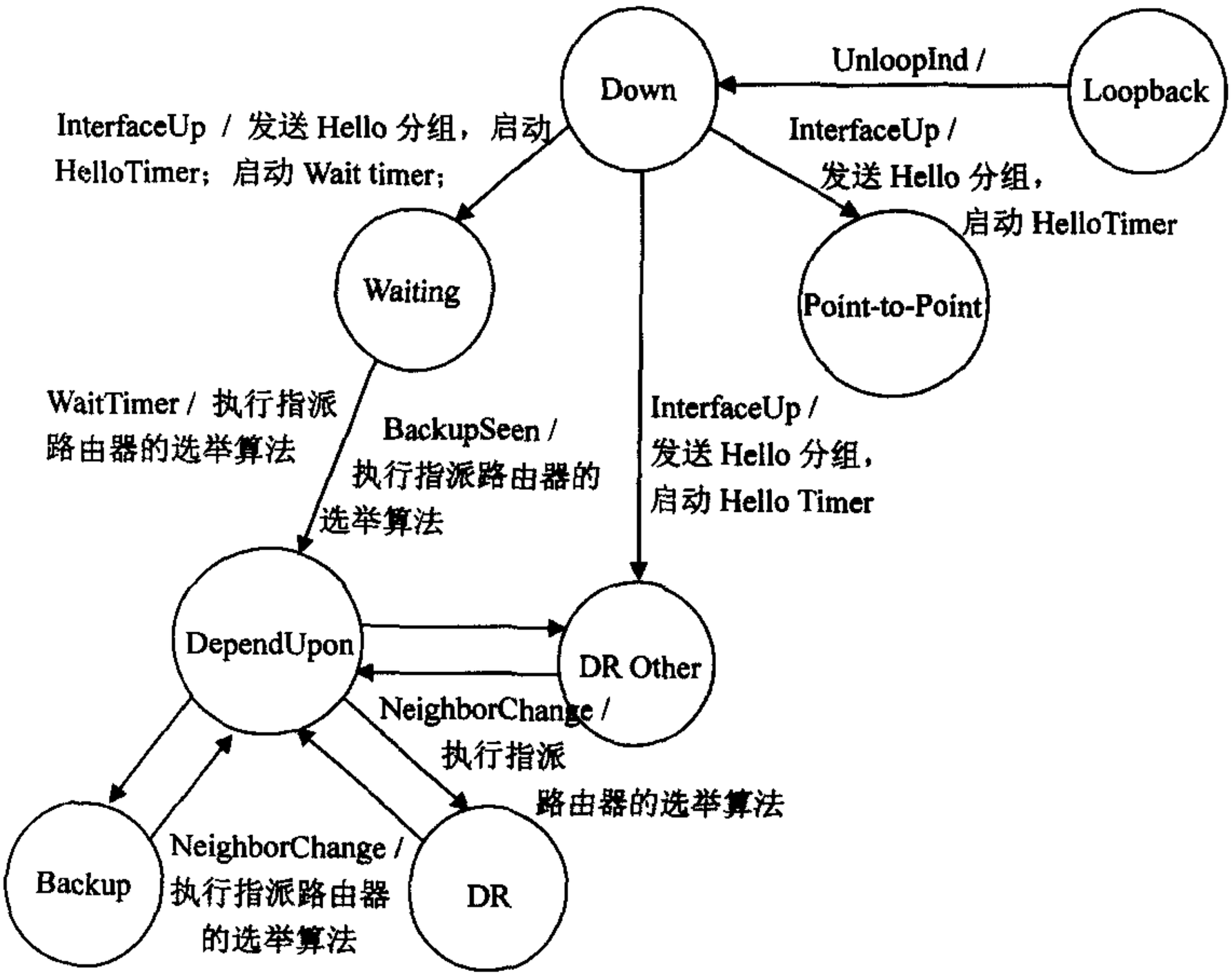


图 5-1 接口有限状态机

由于各种事件的触发，使得接口的状态在上述 8 种状态中变迁，那么，这些状态之间是怎样变迁的？在发生状态变迁的时候又会有什么样的一些动作呢？我们可以用接口状态转移图来描述它。如图 5.1 所示。

在图 5-1 中并没有显示完所有的状态转移变化，如：不论接口现今处于什么样的状态，只要它收到 InterfaceDown 事件，它会将所有接口变量设为初始值，清除掉设置在该接口上的 Hello Timer 或 WaitTimer 等定时器，同时发送 KillNbr 事件给所有与该接口相连的邻居，接着跳转到 ISM_DOWN 状态。

5.4 SDL 状态处理描述

通过上面的分析，我们可以以任何一个状态作为初态，用 SDL 图来描述这个接口有限状态机。在这里，我们以初始状态 Down 为例来说明这个问题。如图 5-2 所示。

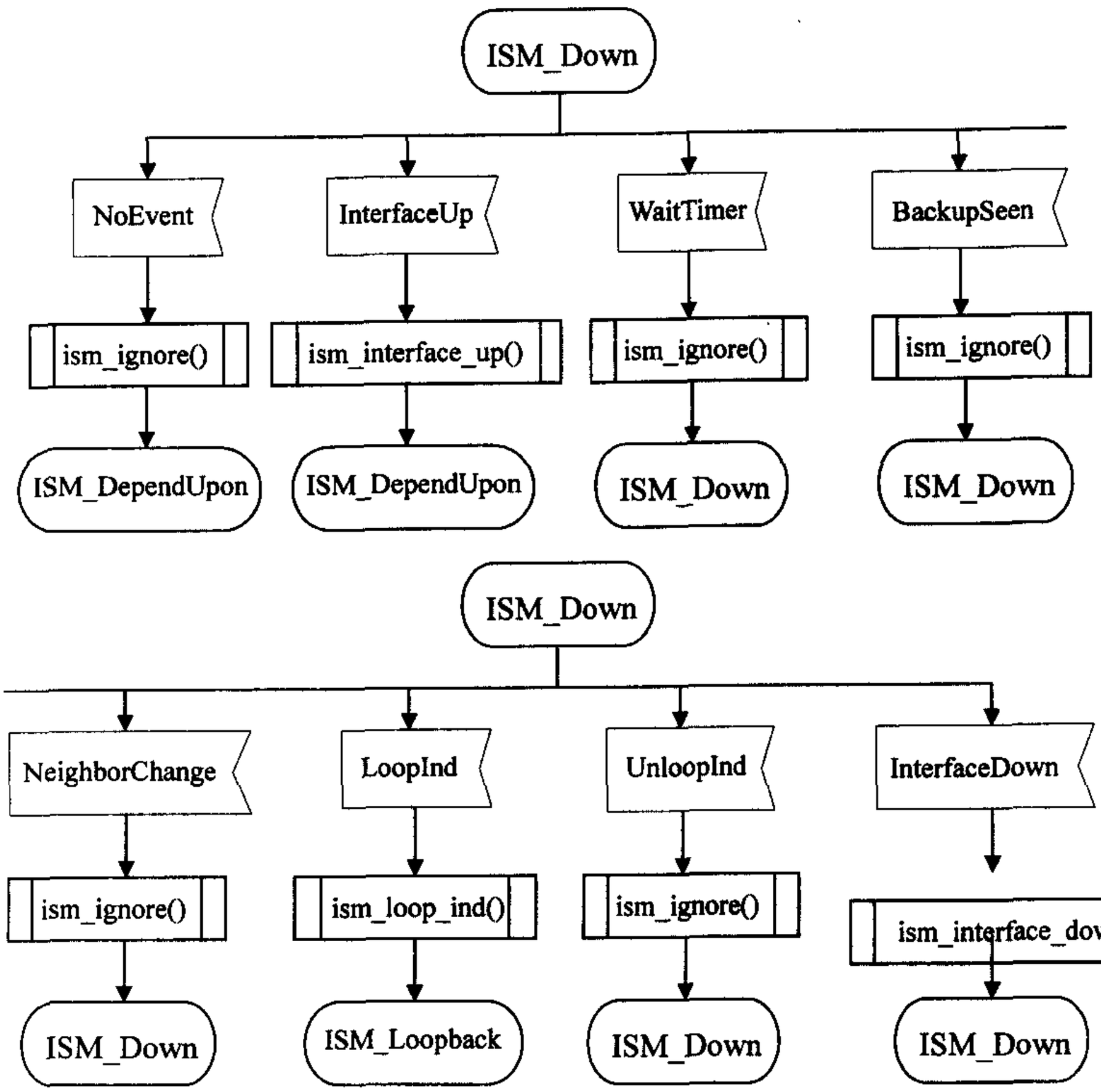


图 5-2 以 ISM_Down 为初态的接口状态机 SDL 图

在图 5-2 中，所有的函数都没有标明参数，其参数均为：（接口指针，发生的接口事件）。其它的 SDL 图也可以用相同的方法画出来。

从图中我们可以看出，状态之间的迁移及应完成的动作都可以通过函数的调用来实现，并且这些函数都具有相同的参数，因此，我们可以将这些状态和函数用统一的形式管理起来。

5.5 状态处理数据结构

对于在这里，我们利用了一个二维数组来管理接口有限状态机的一系列信息。数组的行表示会引起接口状态发生变化的事件，数组的列表示接口的初始状态，数组中的每个元素则表示接口在这样的初始状态下收到这样的事件会引发的操作，以及完成这些操作后接口的新状态。因此，数组中的元素为一个含有 2 个分量的结构，一个分量为处理函数，另一个分量为下一个状态。由于接口的状态有 8 种，引起接口状态发生改变的事件也有 8 种，故接口有限状态机实际上就是一张行为 8，列为 8 的表。以下是该表的一部分，对该表的完整说明见附录。

```
#define OSPF_ISM_STATUS_MAX    8
#define OSPF_ISM_EVENT_MAX     8
    /* Interface State Machine */
struct {
   int (*func) ();
   int next_state;
} ISM [OSPF_ISM_STATUS_MAX][OSPF_ISM_EVENT_MAX] =
{
   ......
   {
      /* Down:*/
      { ism_ignore,          ISM_DependUpon }, /* NoEvent          */
      { ism_interface_up,    ISM_DependUpon },  /* InterfaceUp      */
      { ism_ignore,          ISM_Down },       /* WaitTimer        */
      { ism_ignore,          ISM_Down },       /* BackupSeen       */
      { ism_ignore,          ISM_Down },       /* NeighborChange */
      { ism_loop_ind,        ISM_Loopback },   /* LoopInd          */
      { ism_ignore,          ISM_Down },       /* UnloopInd        */
      { ism_interface_down,  ISM_Down },       /* InterfaceDown    */
  },
.....
  {
```

```
    /* DR: */
    { ism_ignore,              ISM_DependUpon }, /* NoEvent          */
    { ism_ignore,              ISM_DR },         /* InterfaceUp      */
    { ism_ignore,              ISM_DR },         /* WaitTimer        */
    { ism_ignore,              ISM_DR },         /* BackupSeen       */
    { ism_neighbor_change, ISM_DependUpon },   /* NeighborChange */
    { ism_loop_ind,            ISM_Loopback },   /* LoopInd          */
    { ism_ignore,              ISM_DR },         /* UnloopInd        */
    { ism_interface_down,   ISM_Down },         /* InterfaceDown    */
},
};
```

5.6 接口有限状态机主体软件框架

接口有限状态机的主体软件实际上就是对上述的表进行查询的操作。它把发生事件的接口指针和发生的事件作为输入，根据接口指针找到相应的接口数据结构，明确接口目前所处的状态，再将发生的事件和这一状态做为行和列的值对二维数组 ISM 进行查询。那么由谁来完成这个查询的操作呢？在这里我们通过一个函数 ospf_ism_event()来完成这一功能。

```
int ospf_ism_event(struct ospf_interface *iface,int event);
```

在函数体中，通过查询找到相应的下一状态和具体的处理函数，然后调用该处理函数，并据此改变接口的状态。

接口状态机的运行是靠其它模块发来的消息触发的，因此接口有限状态机必须提供出接口以供其它的子块调用。在这里，函数 ospf_ism_event()也是接口状态机子块为其它模块提供的接口。

第六章　Hello 报文及其相关处理

6.1　Hello 报文的格式

Hello 报文有着如表 6-1 所示的协议数据格式。该表格的每一行均为 4 个字节。

表 6-1　Hello 报文格式

OSPF 分组报头，类型 ＝ 1（指明该分组为 Hello 分组）		
网络掩码		
Hello 间隔（Hello_interval）	选项	优先权
死亡间隔（Dead_interval）		
指派路由器（DR）		
备份指派路由器（BDR）		
邻机		
……		
邻机		

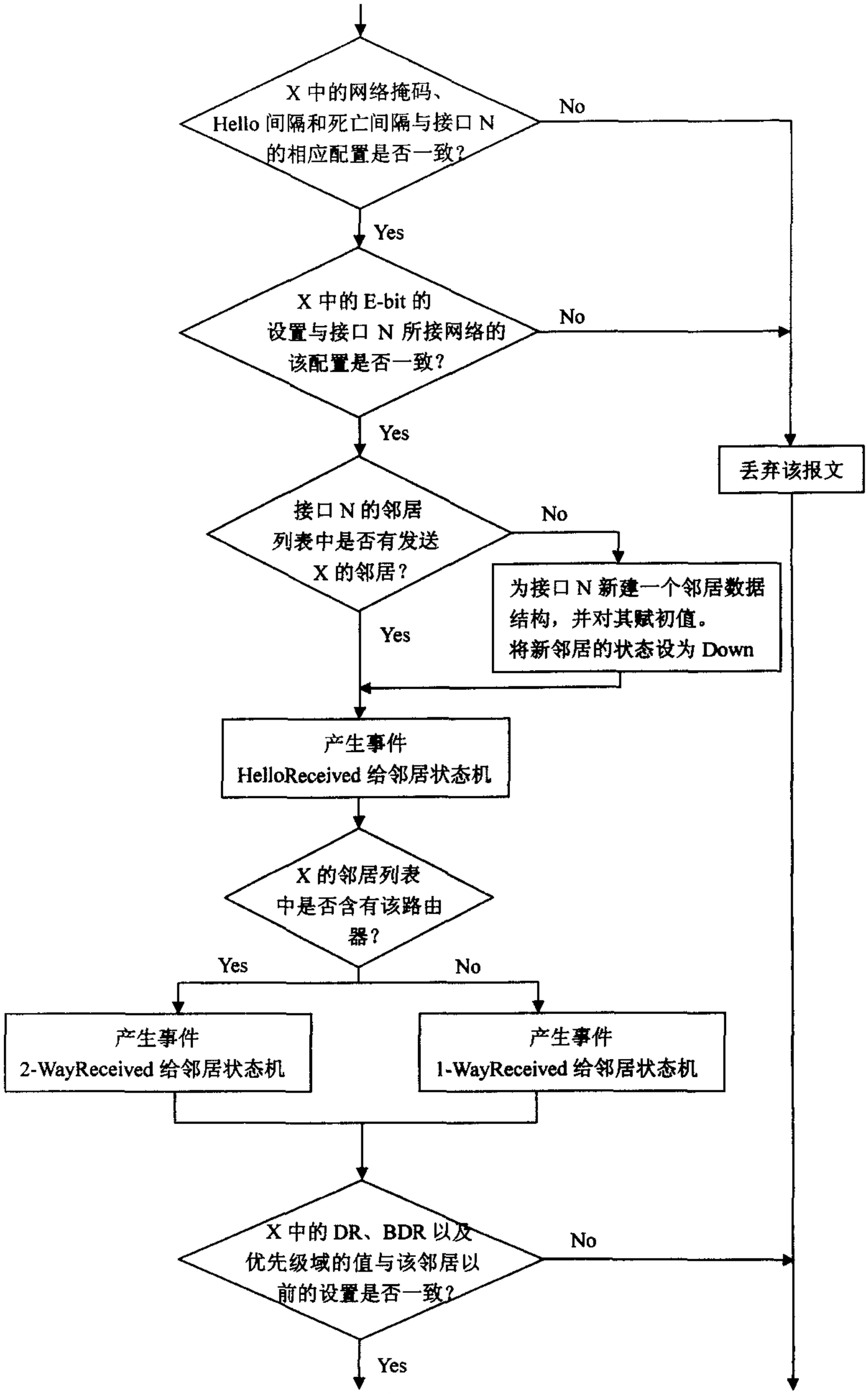
由于 Hello 报文的发送是由路由器接在特定网络上的接口完成的，故 Hello 报文中每一个域的值都是在该接口设定的相应的值。其中，选项字段为可选的路由器功能：是否具有 TOS 能力（T bit），是否具有接收和发送外部路由的能力（E bit）。若该路由器具有，则将相应的位置 1。在该报文的最后，有一个邻机列表，它记录了在这个接口上发现的所有邻居，其中的每一值都为邻居路由器的 Router ID。

6.2　Hello 报文的处理

对 Hello 报文的处理包括两个部分：对收到的邻居发来的 Hello 报文进行处理；定时向网络发送 Hello 报文。

6.2.1　Hello 报文的接收处理

假定接口 N 接收到了 Hello 报文 X，那么，对于该报文的处理应该依照如图 6-1 所示的顺序。


X 中的网络掩码、
Hello 间隔和死亡间隔与接口 N
的相应配置是否一致？
No
Yes
X 中的 E-bit 的
设置与接口 N 所接网络的
该配置是否一致？
No
Yes
丢弃该报文
接口 N 的邻居
列表中是否有发送
X 的邻居？
No
Yes
为接口 N 新建一个邻居数据
结构，并对其赋初值。
将新邻居的状态设为 Down
产生事件
HelloReceived 给邻居状态机
X 的邻居列表
中是否含有该路由
器？
Yes
No
产生事件
2-WayReceived 给邻居状态机
产生事件
1-WayReceived 给邻居状态机
X 中的 DR、BDR 以及
优先级域的值与该邻居以
前的设置是否一致？
No
Yes

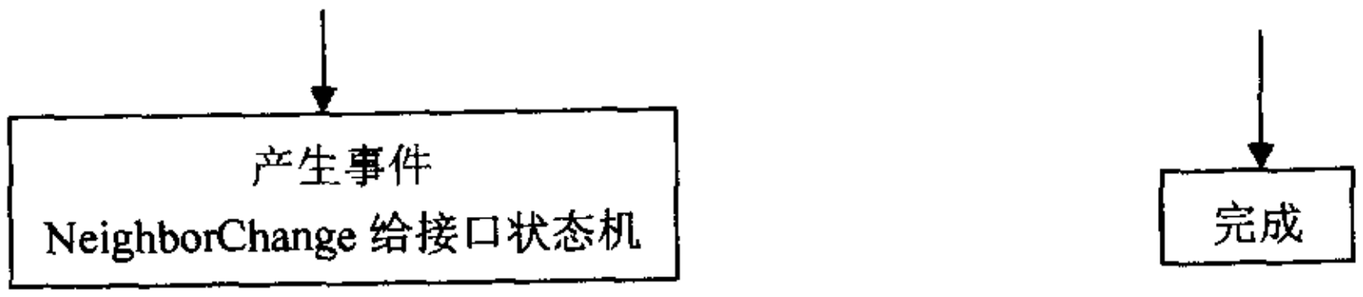


图 6-1　接收 Hello 报文后的处理流程

6.2.2　Hello 报文的发送

Hello 报文的发送是一个周期性的动作，为了实现它，我们在每次发送它后，都会启动一个Hello Timer，将它的值设为接口上的Hello间隔，然后把这个Timer放入 OSPF 模块的 Timer 队列。每当定时到的时候，所执行的操作仍然是发送 Hello 报文，启动 Hello Timer。这就保证了 Hello 报文周期性的发送。

至于 Hello 报文的具体发送方式，依据接口所接的网络类型的不同，其实现方式也就不同。

1. 广播型网络和物理上的点到点网络：每隔 Hello 间隔，以组播的形式发送给所有的 OSPF 路由器（其组播地址为：224.0.0.5）。
2. 虚链路：每隔 Hello 间隔，以单播的形式发送给虚链路的另一端。
3. 点到多点网络：每隔 Hello 间隔，以单播的形式发送给每一个邻居。
4. NBMA 网络：在这种类型的网络上，Hello 报文是发送给一个邻居集合，这个集合该怎样来确定呢？
   (1) 若该路由器可能成为指派路由器（该路由器的优先级大于 0），则周期性的发送 Hello 分组给与自己一样可能成为 DR 的路由器；
   (2) 若该路由器不可能成为 DR（该路由器的优先级小于 0），此时，如果网络上存在DR或BDR，该路由器就应该向它周期性的发送Hello分组；如果从一个可能成为 DR 的路由器处收到了 Hello 分组，则应该发送 Hello 分组以作为相应。
   (3) 若该路由器本身就是 DR 或 BDR，则周期性的发送 Hello 分组给该网络上所有的路由器；

在所讨论的这些情况中，发送 Hello 分组的周期应取决于邻居的状态。若邻居的状态为 Down，则该周期的大小应为探询间隔（Poll_Interval），否则，该周期的大小应为 Hello 间隔。

至于单播和组播的实现，已经在 OSPF 软件接口中描述过了，它们是通过调用其它模块提供的接口函数来完成的。

6.3 指派路由器的选举

在广播型网络和 NBMA 网络上，我们通过收到从邻居处发来的路由器 ID 和路由器的优先级等信息，就可以完成 DR 和 BDR 的选举。而该模块的调用则交给了接口状态机。

那么，指派路由器究竟应该怎样来选举呢？它的选举算法是这样的：在广播型网络或 NBMA 网络上选择优先级最高的路由器，让它作为指派路由器。如果最高优先级的路由器有多个，则选 Router ID 最大的那个路由器。

按照这样的算法，我们可以用如下的叙述来具体描绘指派路由器的选举过程。

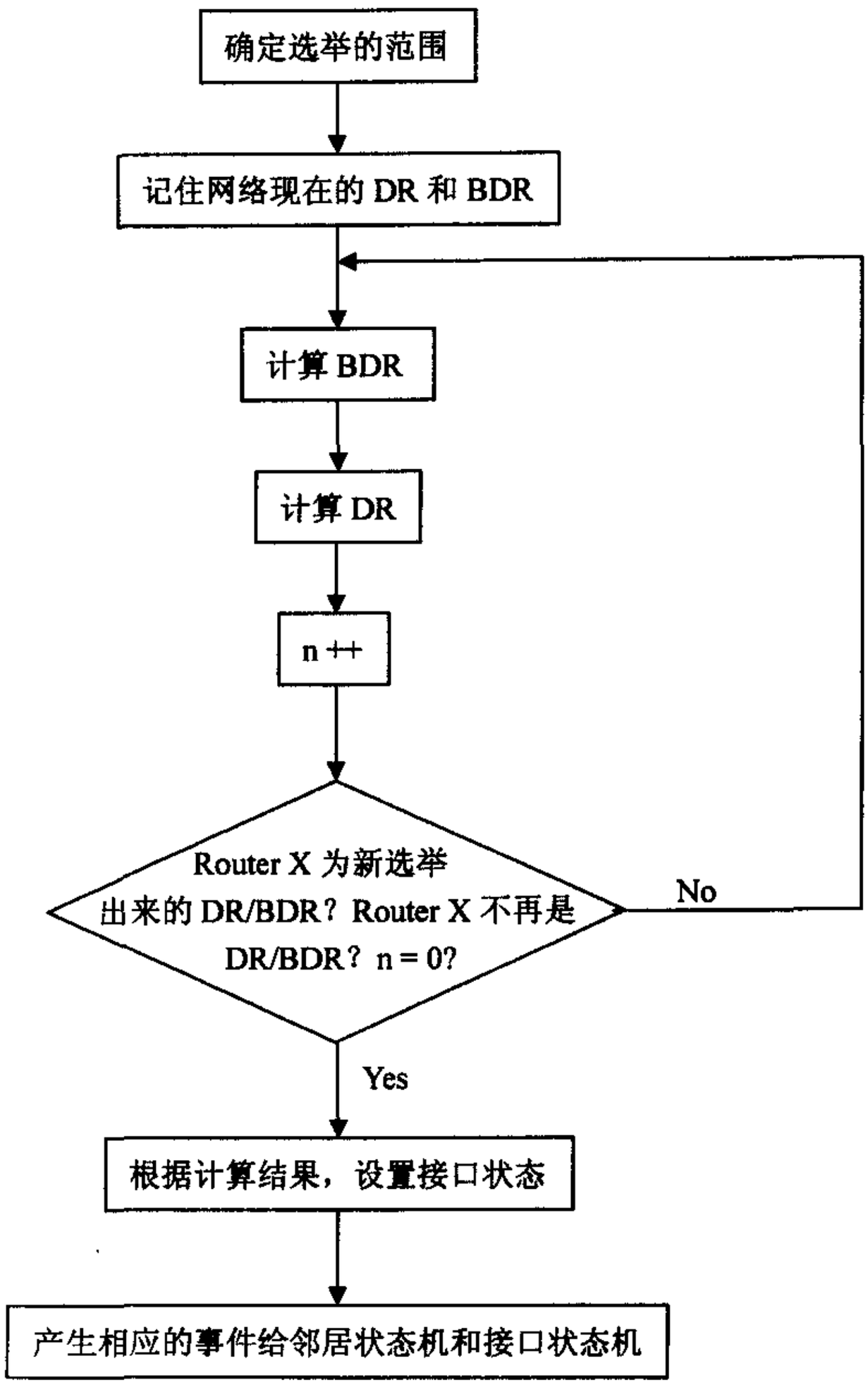


图 6-2 指派路由器选举算法的流程

假设运行指派路由器选举算法的路由器为 Router X，设定一个初值为 0 的计数值 n，那么这个算法的主要流程如图 6-2 所示。

在这个选举过程中，之所以要设定一个计数值 n，是为了保证计算出来的 DR 和 BDR 不同时为 Router X。

在这个流程中，第一步：确定选举的范围，实际上就是在该接口的邻居列表中找出邻居状态大于或等于 2-Way（与该路由器建立了双向连接），且优先级大于 0 的路由器集合，设定该集合为 R。

在计算 BDR 的时候：

1. 找出 R 中没有申明自己为 DR 的路由器，设为集合 R1；
2. 若 R1 中有路由器申明自己为 BDR，则选择 Router Priority 最大的路由器，最为 BDR。如果优先级一样，则选择 Router ID 最大的路由器；
3. 若没有 R1 中没有路由器申明自己为 BDR，则在 R1 中选择 Router Priority 最大的路由器作为 BDR。如果优先级一样，则选择 Router ID 最大的路由器。

至于 DR 的计算：

1. 在 R 中找出申明自己为 DR 的路由器，选择 Router Priority 最大的路由器作为 DR；如果优先级一样，则选择 Router ID 最大的路由器；
2. 若 R 中没有路由器申明自己为 DR，则将刚选举出来的 BDR 作为 DR。如果优先级一样，则选择 Router ID 最大的路由器。

根据计算的结果，我们可以判定接口应处于什么样的一个状态，从而可以进一步进行下面的操作。

第七章　测试

7.1　测试内容

对整个程序来说，需要测试的是各个相对独立的子块能否正常的运作，子块之间能否协调的工作，整个 OSPF 模块能否与路由器软件的其它模块配合起来工作。根据这样的要求，我们需测试如下的内容：

1. 配置子块的结构、配置子块能否完成其功能；
2. Hello 报文的发送和接收；
3. Hello 报文的处理；
4. 接口状态机的运行过程及结果；
5. 指派路由器选举算法是否有效。
6. 配置子块、Hello 协议、邻居状态机子块以及指派路由器选举算法子块之间信息的正确传递和相互正确的调用。

7.2　测试环境

在分析了测试的特点、要求以及目的的基础之上，根据程序运行的要求特设定了以下的环境：

将两台主机同时用两种方式互联起来，一台作为目标机，另外一台作为调试机。这两种方式分别为：通过网络，用 TCP/IP 协议互联；通过串口互联。在目标机上运行路由器软件，重新编制一个测试软件，让它运行在调试机上。这样一来，我们既可以模拟实际的网络环境，进行协议数据的收和发，同时还可以通过串口对路由器进行配置，从而有利于测试结果的观察。

这个测试软件是在 Visual C++的集成环境下完成的，其主要功能为：

1. 接收 Hello 报文，并显示其中的内容；
2. 构造一个 Hello 报文，在网络上进行发送。

7.3　测试方案设计

在测试内容和环境的基础之上，我设计了一个测试的方案，它可以连续的执行，完成所有测试的内容，并显示测试的结果。该方案如图 7-1 和图 7-2 所

示。图 7-1 描述的是在测试机上的测试程序的工作流程，图 7-2 描述的是目标机上的动作流程。

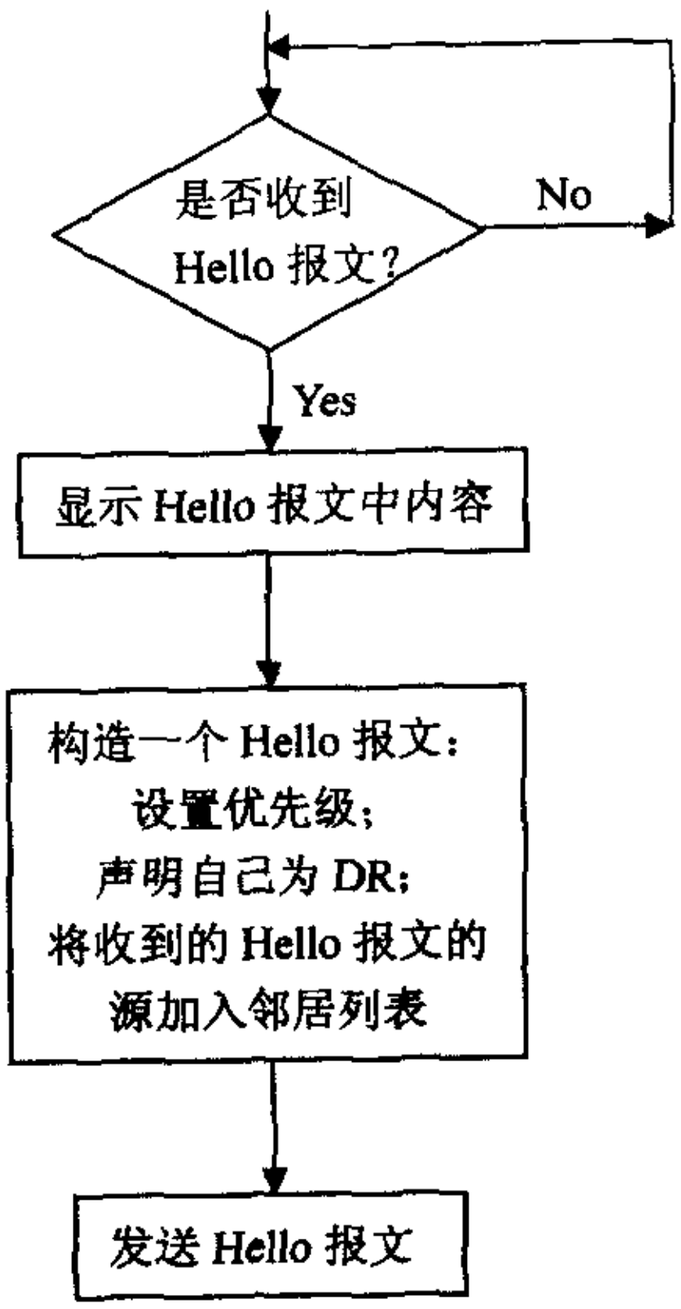


图 7-1　测试程序工作流程

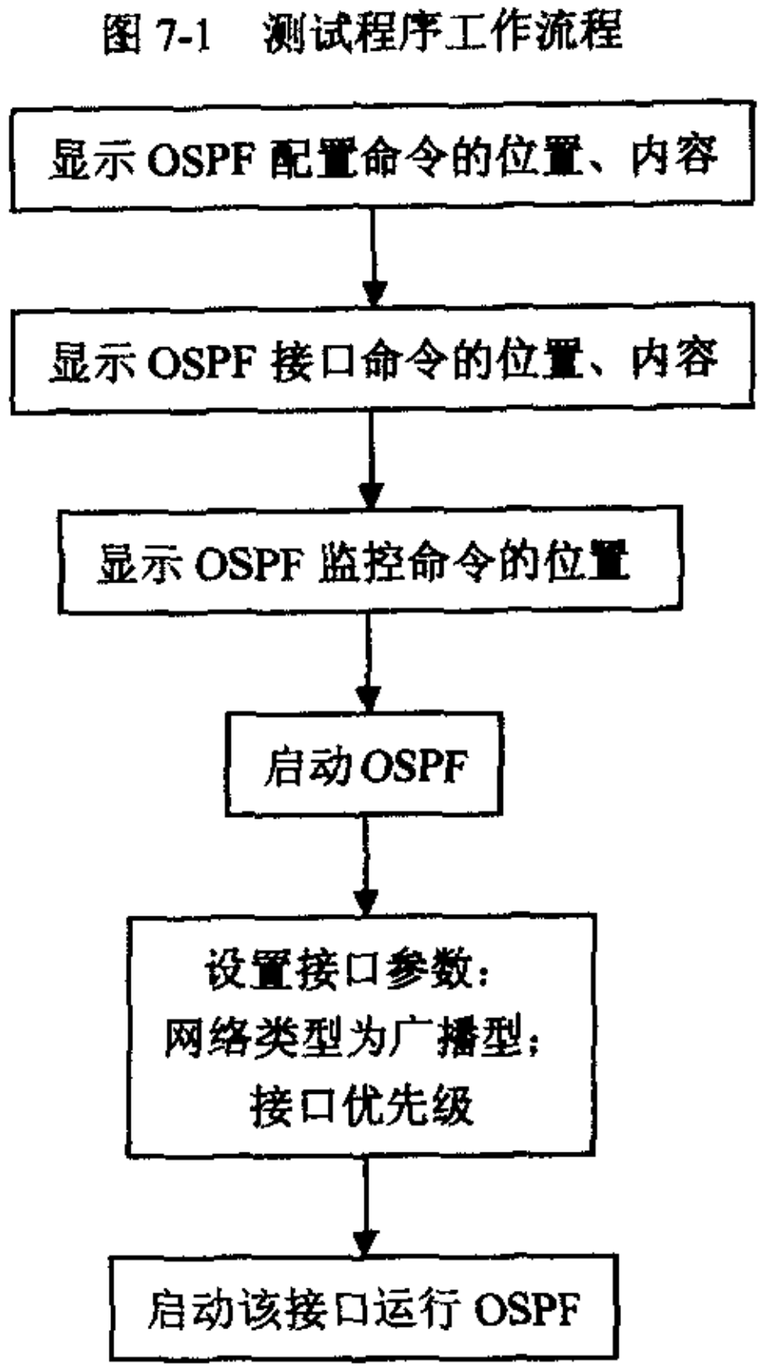


图 7-2　目标机的工作流程

根据对协议的分析，目标机在接口上启动了 OSPF 后，会按照图 7-3 所示的步骤工作。

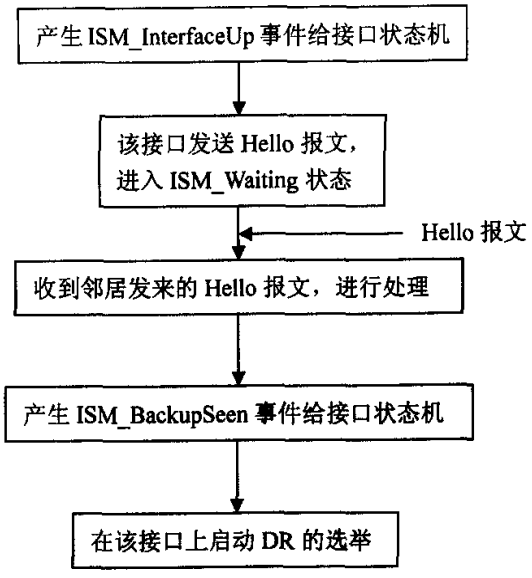


图 7-3　目标机应自动执行的流程

7.4　测试结果

在真正运行后，我们得到了如下的结果：

1. 三种命令的设置和内容。

图 7-4 为配置命令的设置和内容，图 7-5 为接口命令的设置和内容，图 7-6 为监控命令的设置。

2. 具体设置。

为进行测试，我们将接口的类型设为广播型的，以便进行指派路由器的选举，同时，将该接口的优先级为 3。其具体配置过程如图 7-7 所示。

在这样的配置之后，我们利用 show ospf interface 命令可以得到执行上述操作后的结果。如图 7-8 所示。

接着，我们在该接口上用命令 ip ospf open 来启动 OSPF 服务。此时，用 show ospf 命令就可以观察到整个 OSPF 协议运转的整体情况。如图 7-9 所示。

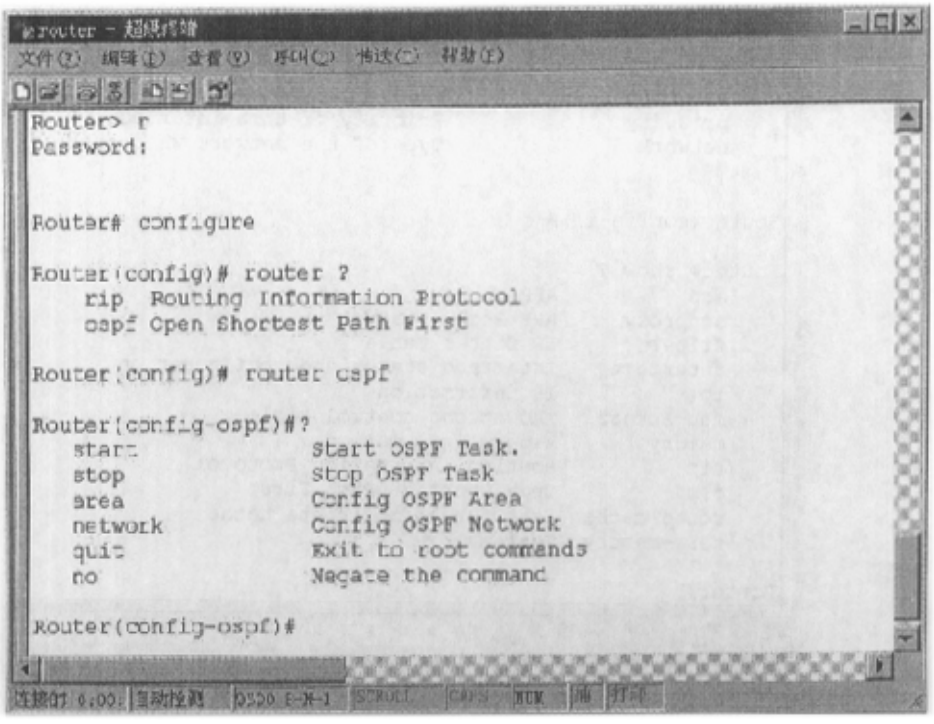


图 7-4 配置命令的设置和内容

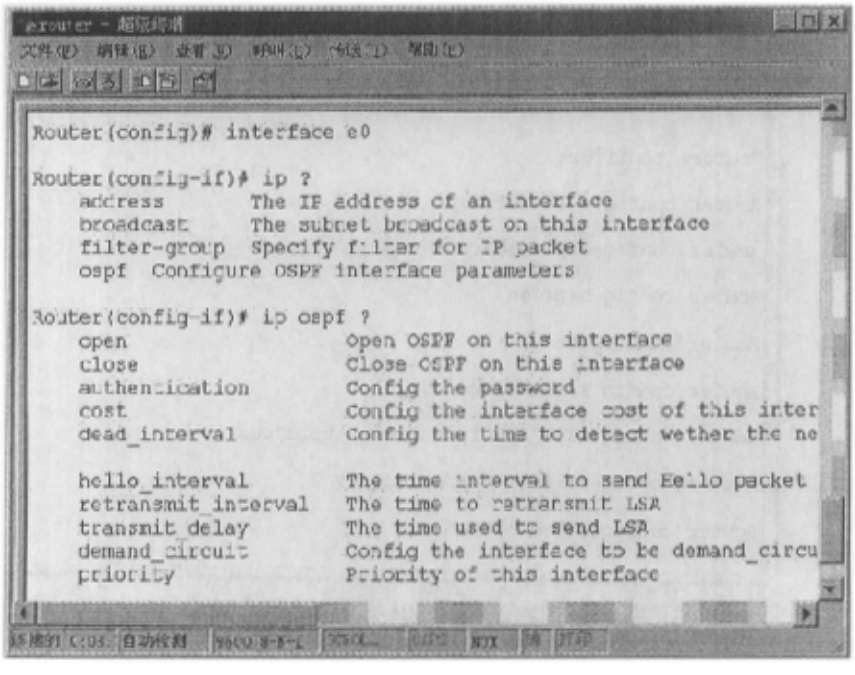


图 7-5 接口命令的设置和内容

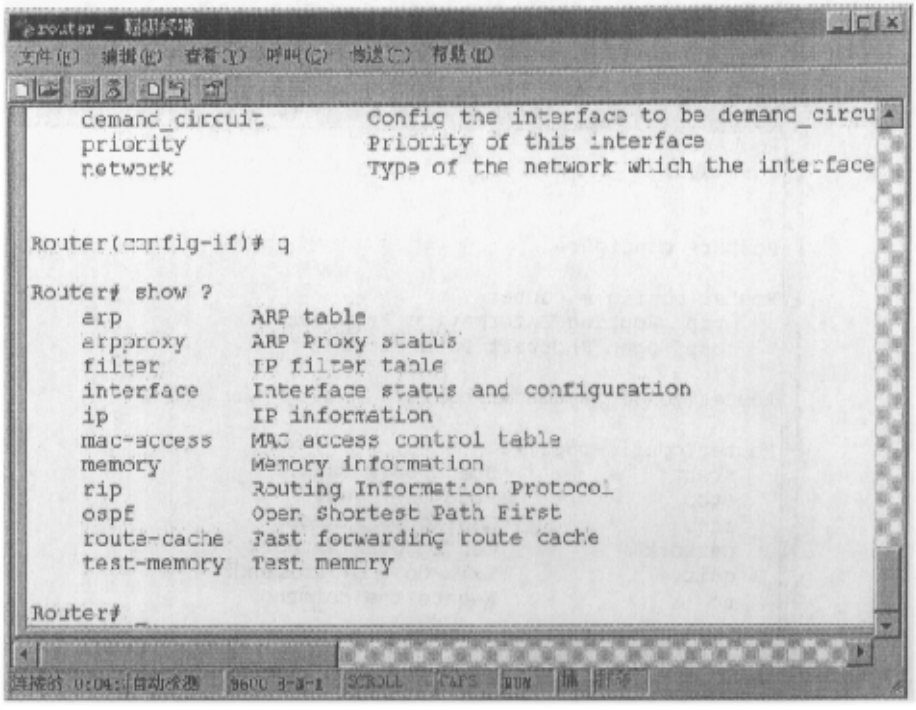


图 7-6　监控命令的设置

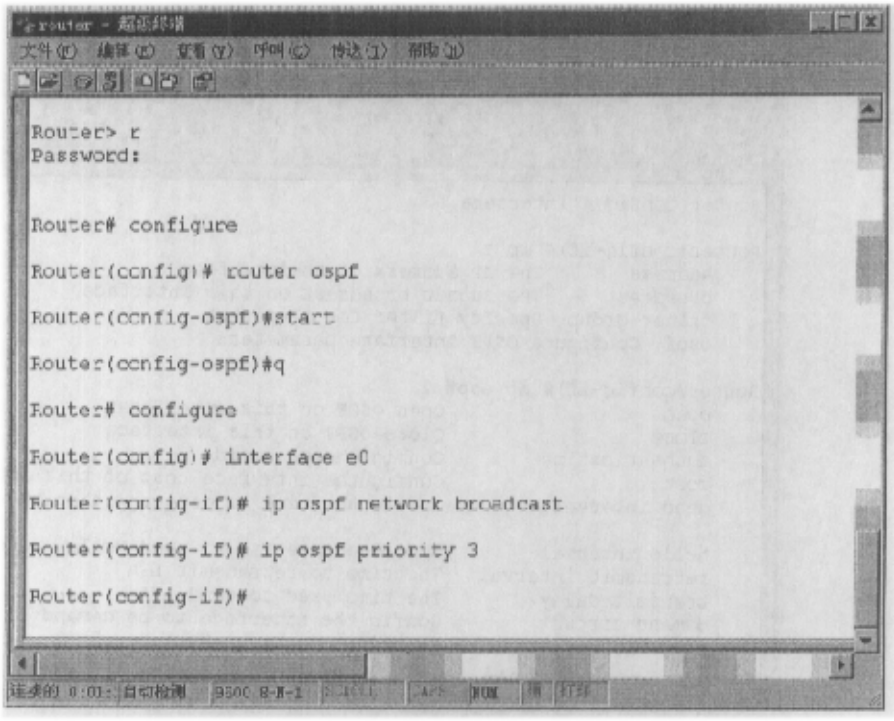


图 7-7　具体的配置过程

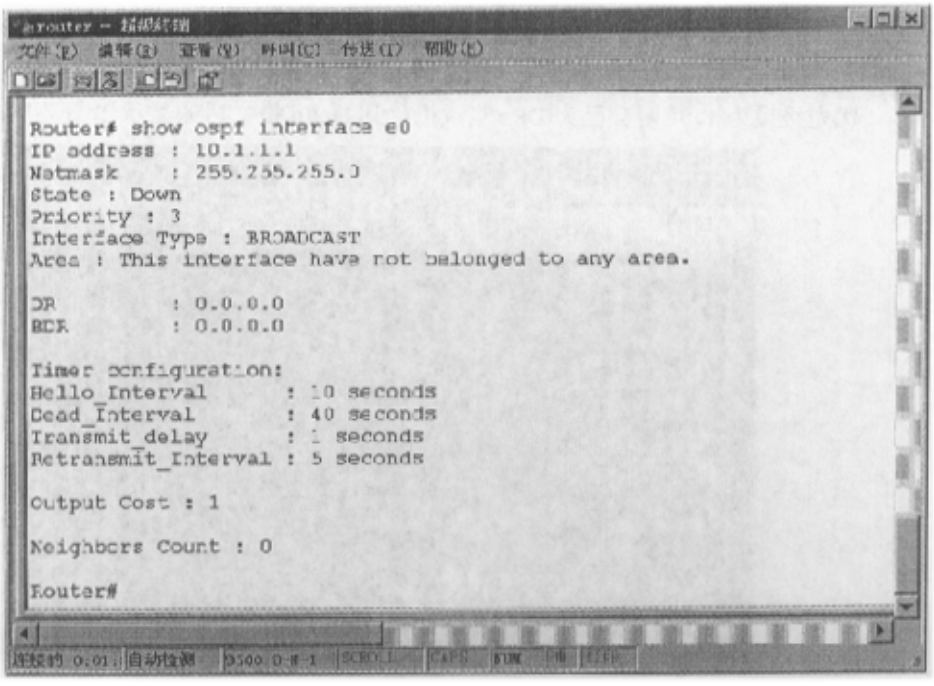


图 7-8　show ospf interface 的结果

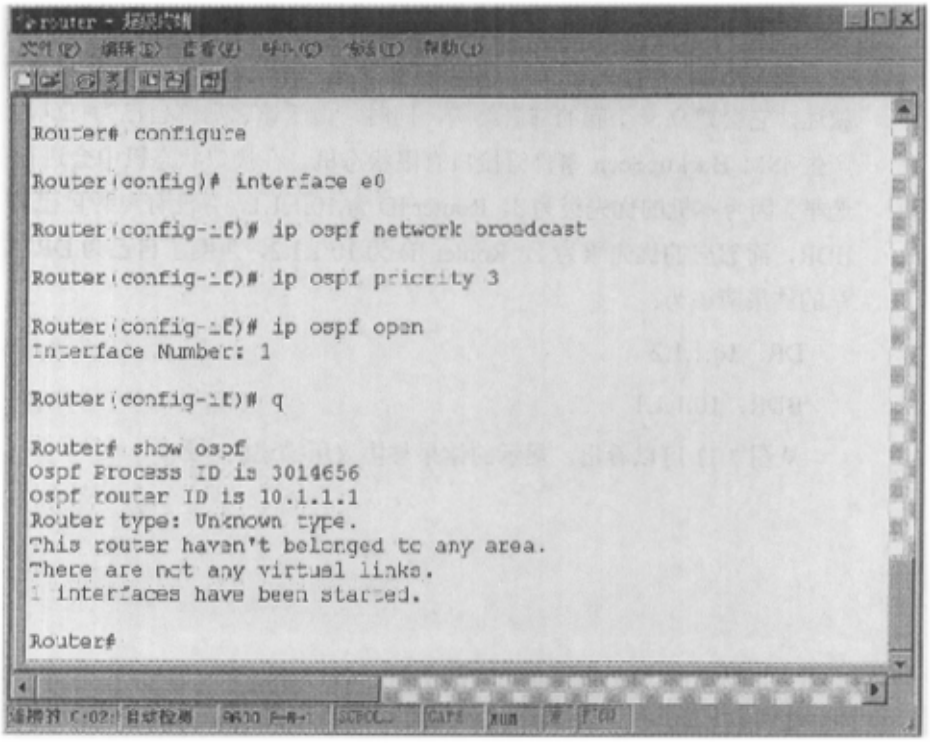


图 7-9　show ospf 的结果

3、目标机执行 ISM_InterfaceUp 事件，向网络发送 Hello 分组。测试机在收到 Hello 报文后，将显示这一报文的内容，并回送一个 Hello 报文，将自己的优先级设为 1，声明自己为 DR。测试机收到 Hello 报文后的显示如图 7-10 所示。

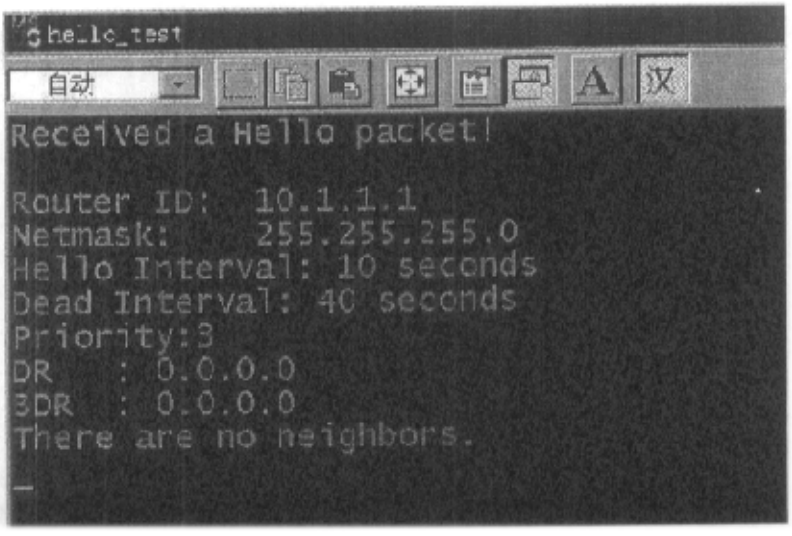


图 7-10　测试机收到的 Hello 报文的内容

4、目标机收到测试机发来的 Hello 报文后的反应如图 7-11 所示。

目标机在收到测试机发来的 Hello 报文后，对该报文进行处理。根据协议的叙述，它会建立一个新的邻居结构，同时，由于该邻居称自己为 DR，因此会产生 ISM_BackupSeen 事件给接口有限状态机。在接口状态机中会进行 DR 的选举。因为本机的优先级为 3，Router ID 为 10.1.1.1，并没有声明自己为 DR 或 BDR，而邻居的优先级为 1，Router ID 为 10.1.1.2，声明了自己为 DR，故，选举的结果应该为：

DR：10.1.1.2

BDR：10.1.1.1

从图 7-11 可以看出，显示的结果与协议所描述的一致。

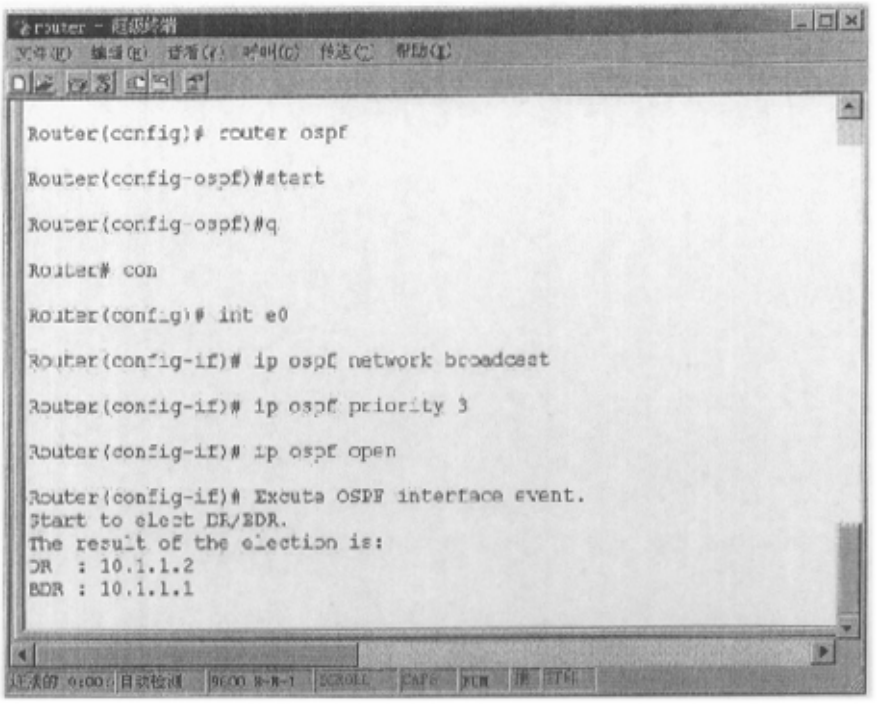


图 7-11　目标机收到 Hello 报文后的执行结果

7.5　测试结论

在经过如上所设置的测试过程的测试以后，发现该程序：

1. 配置命令结构设置正确；
2. 配置命令已能正确运行并能正确的改变程序运行中参数的值；
3. 接口状态机已能正确接收消息，并运转；
4. 指派路由器选举子块已能正确的选出指派路由器和备份指派路由器；
5. 已能正确发送和接收 Hello 报文，并且能对接收到的 Hello 报文进行处理；
6. 各模块之间的工作已能协调的运行。

与测试的目的相符合。

由于程序的其它部分由别的同学来完成，因此，该部分程序还没有与其它厂家的路由器进行联调。但是，我通过截包软件 NetXray 截获了 Cisco 路由器所发出的 Hello 报文和我所编制的软件所发出的 Hello 报文，经过比较发现这两个报文是一致的。这就说明了我们的 Hello 的报文是能够被 Cisco 路由器所识别

的，进而说明它们之间是可以互通的。

结论

本论文简要的介绍了OSPF路由协议，基于已有的路由器平台，研究了OSPF协议的实现方法，并做了如下的实现，同时完成了与主程序的接口。

1. OSPF 模块主体程序的设计；
2. OSPF 配置命令、接口命令和监控命令的结构设计和功能实现；
3. Hello 协议子块的实现；
4. 接口有限状态机的实现；
5. 指派路由器选举算法的实现；

最后，经过模拟网络环境的测试，发现该部分程序已能正常工作。

当然，也有不足的地方：

1. 对协议的分析工作做的还不够；
2. 程序的某些地方还值得去优化；
3. 可以用路由器来搭建真正的网络环境，来对协议的实现进行测试等等。
4. 与其它厂商的路由器还没有进行联调。

参考文献

(1) 谭浩强. C 程序设计. 北京：清华大学出版社. 1997. 10~254

(2) Christian Huitem 著，陶文星译. 因特网路由技术. 北京：清华大学出版社，1998. 75～104，122～124

(3) Douglas E. Comer 著，林遥、蒋慧、杜蔚轩等译，谢希仁校. 用 TCP/IP 进行网际互连，第 1 卷：原理、协议和体系结构（第 3 版）. 北京：电子工业出版社，1999 年. 199~215

(4) Douglas E. Comer 著，张娟、王海译，谢希仁校. 用 TCP/IP 进行网际互连，第 2 卷：设计、实现和内部构成（第 2 版）. 北京：电子工业出版社，1998 年. 325～377

(5) William R. Parkhurst 著，潇湘工作室译. Cisco 路由器 OSPF 设计与实现. 北京：机械工业出版社，1999 年

(6) Terry Slattery，Bill Burton 著，达达翻译组译.Cisco 网络高级 IP 路由技术（第 2 版）.北京：机械工业出版社.2001.145~186

(7) J. Moy. OSPF Version2. RFC2328. 1998. 1~244

(8) Jim Geier. Overview of Common Routing Protocols.Wireless-Nets Ltd.,1998

(9) OSPF Commands. http: //www.cisco.com，2000 年

(10)Cisco – OSPF Design. http: //www.cisco.com，2000 年

致谢

一年多以来，我顺利的完成了毕业设计的任务，为自己的硕士研究生的生活划上了一个圆满的句号。这一切都是和 108 教研室的老师们和同学们的帮助分不开的。

首先，我要感谢的是我的导师毛玉明教授。在进入研究生学习以前，我所学到的知识都是书本上的，对于很多东西的认识很抽象，没有一个具体的概念。是毛老师给了我机会，教我用实践的态度来对待所学的书本上的知识和所遇到的问题，教给我学习和思考问题的方法。老师渊博的知识和深厚的专业素养帮助我拓宽了研究问题的思路，提高了我分析问题和解决问题的能力，让我成为了一个具有一定能力的教学和科研人员。

其次，我要感谢的是 108 教研室的其他老师和同学，他们不厌其烦的为我解答问题，帮我解决我在学习和工作中所遇到的问题。

最后，要感谢家人对我的支持。正是亲人对我精神上和物质上的大力帮助支持，我才得以顺利完成学业。

在 108 教研室的这一段学习和工作的经历将是我人生美好回忆的不可缺少的一部分。

附录

论文中所介绍的该程序中主要的数据结构 struct ospf，struct ospf_area，struct ospf_interface，struct ospf_neighbor 的完整描述如下所示：

```
struct ospf
{
    uint32 router_id;
    UCHAR type;       /* router type    */
#define OSPF_NONE 0
#define OSPF_ABR 1
#define OSPF_ASBR 2

    struct ospf_area *backbone;

    list areas;          /*    area list */
    list vlinks;    /*    virtual link list    */

    struct ospf_interface *iflist[MAX_IF_NUM * 2];    /*    interface list */

UCHAR active_iface;      /*    This parameter show how many interfaces are
                              configured actine with OSPF .    */

    /*    external LSDB    */
    struct ospf_lsdb *external_lsdb;

    struct route_table *table;
};


struct ospf_area
{
    uint32 area_id;          /*    area id of this area     */
    struct ospf *top;  /*    the ospf structure which this area belong to    */

        list ifaces;    /*    interfaces list which belong to this area*/
        list address_range;    /*    the address range of this area   */

        UCHAR active_ifaces;    /*how many interface are active in this area,i.e.the
inteerface status is not Down.*/
```

```
    UCHAR external_routing;   /*External routing capability*/
#define OSPF_AREA_DEFAULT      0
#define OSPF_AREA_STUB 1
#define OSPF_AREA_NSSA 2

    UCHAR transit;     /*transit capability*/
#define OSPF_TRANSIT_FAULT     0
#define OSPF_TRANSIT_TRUE 1

int no_summary;     /*Don't inject summaries into stub*/

uint32 default_cost;   /*Stub default cost*/

UCHAR auth_type;
union
{
        /* Simple Authentication. */
    UCHAR auth_data [OSPF_AUTH_SIMPLE_SIZE];
        /* Cryptographic Authentication. */
    struct
    {
        list auth_crypt;
        uint32 crypt_seqnum;
    } crypt;
}u;

/*    LSDB   including router、network、summary LSA */
struct ospf_lsdb *router_lsdb;
struct ospf_lsdb *network_lsdb;
struct ospf_lsdb *summary_lsdb;

/*   spf parameters   */
uint32 spf_hlodtime;
uint32 spf_delay;
struct spf *tree;
    /*Router numbers in this area*/
    int abr_count;
    int asbr_count;
    int total_count;
};
```

```
struct ospf_interface
{
    struct ospf *ospf;
    struct ospf_area *area;
    struct ospf_vl_data *vl_data;
    list neighbors;

    uint32 ip;
    uint32 netmask;
    uint32 ifnum;      /* interface number    */
    uint32 mtu;  /*    MTU of this interface*/

     int fd ; /*    input socket fd    */
    UCHAR flag;    /*OSPF in this interface is on or off*/
#define OSPF_IF_OFF 0
#define OSPF_IF_ON 1

    UCHAR priority;     /*    the priority of this interface    */

    int status;    /*interface status*/

    UCHAR passive_interface;
#define OSPF_IF_PASSIVE 0
#define OSPF_IF_ACTIVE 1

    UCHAR type;          /*    interface type of this interface */
#define OSPF_IF_NONE 0
#define OSPF_IF_PTOP 1
#define OSPF_IF_PTOMP 2
#define OSPF_IF_NBMA 3
#define OSPF_IF_BROADCAST 4
#define OSPF_IF_VIRTUALLINK 5

    /*time*/
    uint32 transmit_delay;
    uint32 output_cost;
    uint32 retransmit_interval;
    uint32 hello_interval;
    uint32 dead_interval;
    uint32 delay_ack;

/*    timer list    */ /*    there is not any initial operation    */
struct thread *t_hello;
```

```
struct thread *t_poll;
struct thread *t_wait;
struct thread *t_ls_ack;
struct thread *t_ls_ack_direct;
struct thread *t_ls_upd_event;
struct thread *t_network_lsa_self;       /* self-originated network-LSA
                                                reflesh thread. */

/*Authentication data*/
UCHAR auth_type;  /*   0 - no authentication,1 - simple authentication
*/
        UCHAR auth_simple[OSPF_AUTH_SIMPLE_SIZE];
        list auth_crypt;
        uint32 crypt_seqnum;

        uint32 d_router;
        uint32 bd_router;

         /* self-originated LSAs. */
         struct ospf_lsa *network_lsa_self;     /* network-LSA. */
         struct ospf_lsa *summary_lsa_self;     /* summary-LSA. */

    };

struct ospf_neighbor
{
   struct ospf_interface *iface;

   UCHAR status;              /* NSM status. */
   UCHAR dd_flags;            /* master or slave */
#define MASTER    1
#define SLAVE  0
   uint32 dd_seqnum;                /* DD Sequence Number. */

   /* Neighbor Interface Address. */
   uint32 src;
   uint32 netmask;

   uint32 router_id;       /* Router ID. */
   UCHAR options;             /* Options. */
   UCHAR priority;                  /* Router Priority. */
   uint32 d_router;        /* Designated Router. */
   uint32 bd_router;       /* Backup Designated Router. */
```

```
        struct ospf_packet *last_send;     /* Last sent Database Description packet.
*/

        /* Last received Databse Description packet. */
        struct
        {
          UCHAR options;
          UCHAR flags;
          uint32 dd_seqnum;
        } last_recv;

         /*    LSA retransmit list,LSA request list,database summary list    */
        struct ospf_lsdb ls_retransmit;
        struct ospf_lsdb ls_request;
        struct ospf_lsdb db_summary;
        struct ospf_lsa *request_last;

        /*    timer list    */
        struct thread *t_hello_reply;
        struct thread *t_inactivity;
        struct thread *t_db_desc;
        struct thread *t_ls_req;
          struct thread *t_ls_upd;

        };

        struct ospf_vl_data
        {
          ULONG       vl_peer;       /* Router-ID of the peer for VLs. */
          ULONG       vl_area_id;         /* Transit area for this VL. */
          int format;                              /* area ID format */
          struct ospf_interface *vl_oi;          /* Interface data structure for the VL. */
          struct ospf_interface *out_oi;      /* The interface to go out. */
          ULONG       peer_addr;          /* Address used to reach the peer. */
          UCHAR flags;
        };
```

接口有限状态机状态处理数据结构：

```
/* Interface State Machine */
struct {
```

```
  int (*func) ();
  int next_state;
} ISM [OSPF_ISM_STATUS_MAX][OSPF_ISM_EVENT_MAX] =
{
  {
    /* DependUpon: dummy state. */
    { ism_ignore,          ISM_DependUpon },   /* NoEvent          */
    { ism_ignore,          ISM_DependUpon },   /* InterfaceUp      */
    { ism_ignore,          ISM_DependUpon },   /* WaitTimer        */
    { ism_ignore,          ISM_DependUpon },   /* BackupSeen       */
    { ism_ignore,          ISM_DependUpon },   /* NeighborChange */
    { ism_ignore,          ISM_DependUpon },   /* LoopInd          */
    { ism_ignore,          ISM_DependUpon },   /* UnloopInd        */
    { ism_ignore,          ISM_DependUpon },   /* InterfaceDown    /
  },
  {
    /* Down:*/
    { ism_ignore,          ISM_DependUpon },   /* NoEvent          */
    { ism_interface_up,    ISM_DependUpon },   /* InterfaceUp      */
    { ism_ignore,          ISM_Down },         /* WaitTimer        */
    { ism_ignore,          ISM_Down },         /* BackupSeen       */
    { ism_ignore,          ISM_Down },         /* NeighborChange */
    { ism_loop_ind,        ISM_Loopback },     /* LoopInd          */
    { ism_ignore,          ISM_Down },         /* UnloopInd        */
    { ism_interface_down,  ISM_Down },         /* InterfaceDown    */
  },
  {
    /* Loopback: */
    { ism_ignore,          ISM_DependUpon },   /* NoEvent          */
    { ism_ignore,          ISM_Loopback },     /* InterfaceUp      */
    { ism_ignore,          ISM_Loopback },     /* WaitTimer        */
    { ism_ignore,          ISM_Loopback },     /* BackupSeen       */
    { ism_ignore,          ISM_Loopback },     /* NeighborChange */
    { ism_ignore,          ISM_Loopback },     /* LoopInd          */
    { ism_ignore,          ISM_Down },         /* UnloopInd        */
    { ism_interface_down,  ISM_Down },         /* InterfaceDown    */
  },
  {
    /* Waiting: */
    { ism_ignore,          ISM_DependUpon },   /* NoEvent          */
    { ism_ignore,          ISM_Waiting },      /* InterfaceUp      */
    { ism_wait_timer,      ISM_DependUpon },   /* WaitTimer        */
    { ism_backup_seen,     ISM_DependUpon },   /* BackupSeen       */
```

```
    { ism_ignore,              ISM_Waiting },      /* NeighborChange */
    { ism_loop_ind,            ISM_Loopback },     /* LoopInd        */
    { ism_ignore,              ISM_Waiting },      /* UnloopInd      */
    { ism_interface_down,   ISM_Down },            /* InterfaceDown  */
  },
  {
    /* Point-to-Point: */
    { ism_ignore,              ISM_DependUpon },   /* NoEvent        */
    { ism_ignore,              ISM_PointToPoint }, /* InterfaceUp    */
    { ism_ignore,              ISM_PointToPoint }, /* WaitTimer      */
    { ism_ignore,              ISM_PointToPoint }, /* BackupSeen     */
    { ism_ignore,              ISM_PointToPoint }, /* NeighborChange */
    { ism_loop_ind,            ISM_Loopback },     /* LoopInd        */
    { ism_ignore,              ISM_PointToPoint }, /* UnloopInd      */
    { ism_interface_down,      ISM_Down },         /* InterfaceDown  */
  },
  {
    /* DROther: */
    { ism_ignore,              ISM_DependUpon },   /* NoEvent        */
    { ism_ignore,              ISM_DROther },      /* InterfaceUp    */
    { ism_ignore,              ISM_DROther },      /* WaitTimer      */
    { ism_ignore,              ISM_DROther },      /* BackupSeen     */
    { ism_neighbor_change, ISM_DependUpon },       /* NeighborChange */
    { ism_loop_ind,            ISM_Loopback },     /* LoopInd        */
    { ism_ignore,              ISM_DROther },      /* UnloopInd      */
    { ism_interface_down,   ISM_Down },            /* InterfaceDown  */
  },
  {
    /* Backup: */
    { ism_ignore,              ISM_DependUpon },   /* NoEvent        */
    { ism_ignore,              ISM_Backup },       /* InterfaceUp    */
    { ism_ignore,              ISM_Backup },       /* WaitTimer      */
    { ism_ignore,              ISM_Backup },       /* BackupSeen     */
    { ism_neighbor_change, ISM_DependUpon },       /* NeighborChange */
    { ism_loop_ind,            ISM_Loopback },     /* LoopInd        */
    { ism_ignore,              ISM_Backup },       /* UnloopInd      */
    { ism_interface_down,   ISM_Down },            /* InterfaceDown  */
  },
  {
    /* DR: */
    { ism_ignore,              ISM_DependUpon },   /* NoEvent        */
    { ism_ignore,              ISM_DR },           /* InterfaceUp    */
    { ism_ignore,              ISM_DR },           /* WaitTimer      */
```

```
        { ism_ignore,              ISM_DR },          /* BackupSeen      */
        { ism_neighbor_change, ISM_DependUpon },      /* NeighborChange */
        { ism_loop_ind,            ISM_Loopback },    /* LoopInd         */
        { ism_ignore,              ISM_DR },          /* UnloopInd       */
        { ism_interface_down,  ISM_Down },            /* InterfaceDown   */
    },
};
```

个人简历、研究成果及获奖情况

个人简历

邓舒，女，1977 年 11 月 10 日出生；

1995 年～1999 年，就读于电子科技大学通信与信息工程学院。1999 年 7 月获工学学士学位；

1999 年至今，就读于电子科技大学通信与信息工程学院，攻读通信与信息系统硕士学位。

研究成果

在读研期间，参加了本教研室路由器项目的开发工作，已完成其中 DHCP 模块的设计与调试；目前正在完成其中 OSPF 模块的设计工作。

获奖情况

1999～2000 年，被评为电子科技大学优秀学生干部；

2000～2001 年，获电子科技大学一等奖学金，三星专项奖学金；同时被评为电子科技大学优秀学生干部。