LINUX

第13章 项目实践：BT下载软件的开发

13.1 BT 软件简述

可能许多人使用过比特彗星 (BitComet) 、比特精灵 (BitSpirit) 、迅雷下载过自己喜欢的影片、电视剧、网络游戏；还有很多人使用过PPLive、PPStream、沸点、 QQ 直播等免费的网络电视直播软件在线观看自己喜欢的影片。所有这些软件都采用了一种近年来流行起来的协议，BitTorrent 协议，简称 BT 协议。

在互联网中，许多新技术深刻地改变了人们的工作、生活和学习的模式。Tim Berners Lee在1990年设计和发明了 HTTP 协议，从而引发了互联网的变革，使网络冲浪、电子商务成为可能，因此也造就了百度、谷歌等搜索引擎公司以及网易、雅虎、搜狐、新浪、腾讯等门户网站，同时也造就了一个又一个的数字英雄。在 HTTP 协议发明之前，统治互联网的是 SMTP 和 FTP 协议，这两种协议的通信量占据首位； HTTP 协议诞生之后，其通信流量和使用率都占据了第一。2003年，年轻的软件工程师 Bram Cohen 发明了 BitTorrent 协议。在短短的时间内， BT 协议的通信流量占据了互联网总流量的六成以上。 BT 协议成为一种新的变革技术，因此也催生了很多 BT 软件，如BitComet、 BitSpirit 、 Azureus ， PPLive、PPStream。

下面将详细介绍 BT 协议和技术的各个细节，并在此基础上使用 C 语言在 Linux 环境下开发了一个 BT 软件。

13.2 BitTorrent 协议

13.2.1 概要介绍

BitTorrent （简称 BT ）是一个文件分发协议，每个下载者在下载的同时不断向其他下载者上传已下载的数据。而在FTP 、 HTTP 协议中，每个下载者从 FTP 或 HTTP 服务器处下载自己所需要的文件，各个下载者之间没有交互。当非常多的用户同时访问和下载服务器上的文件时，由于 FTP 服务器的处理能力和带宽的限制，下载速度会急剧下降，有的用户根本访问不了服务器。 BT 协议与 FTP 协议不同，它的特点是下载的人越多下载的速度越快，其原因在于每个下载者将已下载的数据提供给其他下载者下载，它充分利用了用户的上载带宽。 BT 协议通过一定的策略保证上传的速度越快，下载的速度也越快。

13.2.2 基于 BT 协议的文件分发系统的构成

基于 BT 协议的文件分发系统由以下几个实体构成。

（1）一个Web服务器。

（2）一个种子文件。

（3）一个 Tracker 服务器。

（4）一个原始文件提供者。

（5）一个网络浏览器。

（6）一个或多个下载者。

Web服务器上保存着种子文件，下载者使用网络浏览器（如IE浏览器）从Web服务器上下载种子文件。种子文件，又称为元原文件或 metafile ，它保存了共享文件的一些信息，如共享文件的文件名、文件大小、 Tracker 服务器的地址。种子文件通常很小，一般大小为1GB的共享文件，其种子文件不足100KB，种子文件以.torrent为后缀。 Tracker 服务器保存着当前下载某共享文件的所有下载者的IP和端口。原始文件提供者提供完整的共享文件供其他下载者下载，它也被称为种子，种子文件就是提供者使用 BT 客户端生成的。每个下载者通过运行 BT 客户端软件下载共享文件。我们把某个下载者本身称为客户端，把其他下载者称为peer。

BT 客户端下载一个共享文件的过程是：客户端首先解析种子文件，获取待下载的共享文件的一些信息，其中包括 Tracker 服务器的地址。然后客户端连接 Tracker 获取当前下载该文件的所有下载者的IP和端口。之后客户端根据IP和端口连接其他下载者，从它们那里下载文件，同时把自己已下载的部分提供给其他下载者下载。

共享文件在逻辑上被划分为大小相同的块，称为 piece ，每个 piece 的大小通常为256KB。对于共享文件，文件的第 1 字节到第 256K （即262144）字节为第一个 piece ，第 256K ＋ 1 字节到第 512K 字节为第二个 piece ，依此类推。种子文件中包含有每个 piece 的hash值。 BT 协议规定使用Sha1算法对每个 piece 生成20字节的hash值，作为每个 piece 的指纹。每当客户端下载完一个 piece 时，即对该 peice 使用Sha1算法计算其hash值，并与种子文件中保存的该 peice 的hash值进行比较，如果一致即表明下载了一个完整而正确的 piece 。一旦某个 piece 被下载，该 piece 即提供给其他 peer 下载。在实际上传和下载中，每个 piece 又被划分为大小相同的 slice，每个slice的大小固定为 16KB （16384 字节）。peer之间每次传输以slice为单位。

从以上描述可以得知，待开发的 BT 软件（即 BT 客户端）主要包含以下几个功能：解析种子文件获取待下载的文件的一些信息，连接 Tracker 获取peer的IP和端口，连接peer进行数据上传和下载、对要发布的提供共享文件制作和生成种子文件。种子文件和 Tracker 的返回信息都以一种简单而高效的编码方式进行编码，称为B编码。客户端与 Tracker 交换信息基于 HTTP 协议，Tracker 本身作为一个Web服务器存在。客户端与其他peer采用面向连接的可靠传输协议TCP进行通信。下面将进一步作详细的介绍。

13.2.3 B编码

种子文件和 Tracker 的返回信息都是经过B编码的。要解析和处理种子文件以及 Tracker 的返回信息，首先要熟悉B编码的规则。B编码中有 4 种类型：字符串、整型、列表、字典。

字符串的编码格式为：<字符串的长度>:<字符串>，其中 <> 括号中的内容为必需。例如，有一个字符串 spam ，则经过B编码后为4:spam。

整型的编码格式为：i<十进制的整型数>e ，即B编码中的整数以 i 作为起始符，以 e 作为终结符， i 为integer的第一个字母， e 为 end 的第一个字母。例如，整数 3 ，经过B编码后为 i3e ，整数－ 3 的B编码为i-3e，整数 0 的B编码为 i0e 。

注意i03e不是合法的B编码，因为03不是十进制整数，而是八进制整数。

列表的编码格式为：l<任何合法的类型>e ，列表以 l 为起始符，以 e 为终结符，中间可以为任何合法的经过B编码的类型， l 为 list 的第一个字母。例如，列表 l4:spam4:eggse 表示两个字符串，一个是 spam ，一个是eggs。

字典的编码格式为：d< 关键字 >< 值 >e ，字典以 d 为起始符，以 e 为终结符，关键字是一个经过 B 编码的字符串，值可以是任何合法的 B 编码类型，在 d 和 e 之间可以出现多个关键字和值对，d 是 dictionary 的第一个字母。例如，d4:spaml3:aaa3:bbbee，它是一个字典，该字典的关键字是 spam ，值是一个列表（以 l 开始，以 e 结束），列表中有两个字符串 aaa 和 bbb 。

又如：d9:publisher3:bob17:publisher-webpage15:www.example.come，它也是一个字典，第一个关键字是 publisher ，对应的值为 bob ，第二个关键字是 publisher-webpage ，对应的值是 www.example.com 。

13.2.4 种子文件的结构

种子文件包含了提供共享的文件的一些信息，它以.torrent为后缀名，种子文件也被称为元信息文件或 metafile ，它是经过 B 编码的。种子文件事实上就是一个 B 编码的字典，它含有以下关键字如表13-1所示。

表 13-1 种子文件的关键字

关 键 字	含 义
info	该关键字对应的值是一个字典，它有两种模式，“singel file”和“multiple file”，文件模式和多文件模式。单文件模式是指待共享的文件只有一个，多文件模式是指提供共享的不止一个文件，而是两个或两个以上。如使用 BT 软件下载一部影片时，影片的上下部可能分别放在不同的文件里
announce	该关键字的值为 Tracker 的 URL
announce－list	可选，它的值存放的是备用 Tracker 的 URL
creation－date	可选，该关键字对应的值存放的是创建种子文件的时间
comment	可选，它的值存放的是种子文件制作者的备注信息，对于下载来说，该关键字基本没有用处，因此不必理会
created by	可选，该关键字对应的值存放的是生成种子文件的 BT 客户端软件的信息，如客户端名 版本号等，一般不必理会

info 是最重要的一个关键字，它的值是一个字典，下面对它再作进一步的介绍。无论是单文件模式还是多文件模式，该字典都包含关键字如表13-2所示。

表13-2 info 包含的关键字

关 键 字	含 义
piece length	每个 piece 的长度，它的值是一个 B 编码的整型，该值通常为 i262144e，即 256K，也有可能为 512K 或 128K
pieces	对应的值为一个字符串，它存放的是各个 piece 的 hash 值，这个字符串的长度一定是 20 的倍数，因为每个 piece 的 hash 值的长度为 20 字节
private	该值如果为 1，则表明客户端必须通过连接 Tracker 来获取其他下载者，即 peer 的 IP 地址和端口号；如果为 0，则表明客户端还可以通过其他方式来获取 peer 的 IP 地址和端口号，如 DHT 方式 ◦DHT 即分布式哈希表（Distribute Hash Tabel），它是一种以分布式的方式来获取 peer 的方法，现在许多 BT 客户端既支持通过连接 Tracker 来获取 peer，也支持通过 DHT 来获取 peer。如果种子文件中没有 private 这个关键字，则表明不限制一定要通过连接 Tracker 来获取 peer

对于单文件模式的种子文件，info 的值还含有的关键字如表13-3所示。

表 13-3 单模式种子文件的关键字

关 键 字	含 义
name	共享文件的文件名，也就是要下载的文件的文件名
length	共享文件的长度，以字节为单位
md5sum	可选，它是共享文件的 md5 值，这个值在 BT 协议中根本没有使用，所以不必理会

对于多文件模式的种子文件， info 的值还含有的关键字如表13-4所示

表 13-4　　多文件模式种子文件的关键字

关 键 字	含　义
name	存放所有共享文件的文件夹名
files	它的值是一个列表，列表中含有多个字典，每个共享文件为一个字典 ·该字典中含有三个关键词

Files的每个共享文件为一个字典字典的关键词如表13-5所示

表13-5　　Files字的关键词

关 键 词	含　义
length	共享文件的长度，以字节为单位
md5sum	可选，同上
path	存放的是共享文件的路径和文件名

建议读者到一些提供 BT 种子文件下载的网站，如 bt.greedland.net 、www.btchina.net，下载几个种子文件并在Windows操作系统下使用记事本打开进行分析，就可以清楚的了解上述概念。

13.2.5 与Tracker交互

完成解析种子文件并从中获取 Tracker 服务器的URL后，即可开始与 Tracker 进行交互。与 Tracker 进行交互主要有两个目的：一是将自己的下载进度告知给 Tracker 以便 Tracker 进行一些相关的统计；二是获取当前下载同一个共享文件的 peer 的IP地址和端口号。

客户端使用 HTTP 协议与 Tracker 进行通信。 Tracker 通过HTTP GET方法获取请求，请求的构成为 Tracker 的URL后面跟一个？ 以及参数和值对，如 http://tk.greedland.net/ announce?param1= value1&param2= value2 。

在客户端发往 Tracker 的GET请求中，通常包含参数如表13-6所示。

表13-6　　GET请求的参数

参　数	含　义
info_hash	与种子文件中 info 关键字对应的值，通过 Sha1 算法计算其 hash 值，该 hash 值就是 info_hash 参数对应的值，该 hash 值的长度固定为 20 字节
peer_id	每个客户端在下载文件前以随机的方式生成的 20 字节的标识符，用于标识自己，它的长度也是固定不变的
port	监听端口号，用于接收其他 peer 的连接请求
uploaded	当前总的上传量，以字节为单位
downloaded	当前总的下载量，以字节为单位
left	还剩余多少字节需要下载，以字节为单位
compact	该参数~~的值一般为 1~~用于指示服务器以何种方式返回 peer，~~·~~该值为 1 时，每个 peer 占 6 六个字节，前 4 四个字节为 peer 的 IP 地址，后两个为 peer 的端口号。
event	它的值为 started、completed、stopped 其中之一。客户端第一次与 Tracker 进行通信时，该值为 started；下载完成时，该值为 completed；客户端即将关闭时，该值为 stopped
ip	可选，将客户端的 IP 地址告知给 Tracker，Tracker 可以通过分析客户端发给 Tracker 的 IP 数据包来获取客户端的 IP 地址，因此该参数是可选的，一般不用指明客户端的 IP
numwant	可选，希望 Tracker 返回多少个 peer 的 IP 地址和端口号。如果该参数缺省，则默认返回 50 个 peer 的 IP 地址和端口号
key	可选，它的值为一个随机数，用于进一步标识客户端。因为已经由 peer_id 来标识客户端，因此该参数一般不使用
trackerid	可选，一般不使用

Tracker 服务器的返回信息是一个经过 B 编码的字典。它含有关键字如表13-7所示。

表13-7 Tracker 服务器返回信息关键字

关 键 字	含 义
failure reason	该关键字对应的值是一个可以读懂的字符串，指明 GET 请求失败的原因，如果返回信息中含有这个关键字，就不会再包含其他任何关键字
warnging message	该关键字对应的值是一个可以读懂的警告字符串
interval	指明客户端在下一次连接 Tracker 前所需等待的时间，以秒为单位
min interval	指明客户端在下一次连接 Tracker 前所需等待的最少时间，以秒为单位
tracker id	指明 Tracker 的 ID
complete	一个整数，指明当前有多少个 peer 已经完成了整个共享文件的下载
incomplete	一个整数，指明当前有多少个 peer 还没有完成共享文件的下载
peers	返回各个 peer 的 IP 和端口号，它的值是一个字符串。首先是第一个 peer 的 IP 地址，然后是其端口号；接着是第二个 peer 的 IP 地址，然后是其端口号；依此类推

以下是一个发往 Tracker 服务器的 HTTP GET 请求的示例：

```
http://tk.greedland.net/announce?info_hash=01234567890123456789&
peer_id=01234567890123456789&port=3210&compact=1&uploaded=0&downloaded=0&left=800
0000&event=started
```

以下是一个 Tracker 服务器回应的示例：

```
d8:completei100e10:incompletei200e8:intervali1800e5:peers300:......e
```

其中，“.....”是一个长度为 300 的字符串，含有50个 peer 的IP地址和端口号。IP地址占 4 字节，端口号占 2 字节，即一个 peer 占 6 字节。

发往 Tracker 服务器的 HTTP GET 请求中，info_hash 和 peer_id 可能含有非数字、非字母的字符，即含有除 0～9、a～z、A～Z 之外的字符，此时要对字符进行编码转换。例如，空格应该转换为%20。否则 Tracker 无法正确处理 GET 请求。

13.2.6 peer 之间的通信协议

peer 之间的通信协议又称为 peer wire protocal，即 peer 连线协议，它是一个基于 TCP 协议的应用层协议。

为了防止有的 peer 只下载不上传，BitTorrent 协议建议，客户端只给那些向它提供最快下载速度的 4 个 peer 上传数据。简单地说就是谁向我提供下载，我也提供数据供它下载；谁不提供数据给我下载，我的数据也不会上传给它。客户端每隔一定时间，比如10秒，重新计算从各个 peer 处下载数据的速度，将下载速度最快的 4 个 peer 解除阻塞，允许这 4 个 peer 从客户端下载数据，同时将其他 peer 阻塞。

一个例外情况是，为了发现下载速度更快的 peer，协议还建议，在任一时刻，客户端保持一个优化非阻塞 peer，即无论该 peer 是否提供数据给客户端下载，客户端都允许该 peer 从客户端这里下载数据。由于客户端向 peer 上传数据，peer 接着也允许客户端从 peer 处下载数据，并且下载速度超过 4 个非阻塞 peer 中的一个。客户端每隔一定的时

间，如 30 秒，重新选择优化非阻塞 peer 。

当客户端与peer建立TCP连接后，客户端必须维持的几个状态变量如表13-8所示。

表 13-8 客户端必须维持的状态变量

状态变量	含 义
am_chocking	该值若为 1，表明客户端将远程 peer 阻塞。此时如果 peer 发送数据请求给客户端，客户端将不会理会。也就是说，一旦将 peer 阻塞，peer 就无法从客户端下载到数据；该值若为 0，则刚好相反，即表明 peer 未被阻塞，允许 peer 从客户端下载数据
am_interested	该值若为 1，表明客户端对远程的 peer 感兴趣。当 peer 拥有某个 piece，而客户端没有，则客户端对 peer 感兴趣。该值若为 0，则刚好相反，即表明客户端对 peer 不感兴趣，peer 拥有的所有 piece，客户端都拥有
peer_chocking	该值若为 1，表明 peer 将客户端阻塞。此时，客户端无法从 peer 处下载到数据。该值若为 0，表明客户端可以向 peer 发送数据请求，客户端将进行响应
peer_interested	该值若为 1，表明 peer 对客户端感兴趣。也即客户端拥有某个 piece，而 peer 没有。该值若为 0，表明 peer 对客户端不感兴趣

当客户端与peer建立TCP连接后，客户端将这几个变量的值设置为。

am_chocking = 1 。

am_interested = 0 。

peer_chocking = 1 。

peer_interested = 0 。

当客户端对peer感兴趣且peer未将客户端阻塞时，客户端可以从peer处下载数据。当peer对客户端感兴趣，且客户端未将peer阻塞时，客户端向peer上传数据。

除非另有说明，所有的整数型在本协议中被编码为 4 字节值（高位在前低位在后），包括在握手之后所有信息的长度前缀。

客户端与一个peer建立TCP连接后，首先向peer发送握手消息，peer收到握手消息后回应一个握手消息。

握手消息是一个长度固定为68字节的消息。消息的格式如下：

```
<pstrlen><pstr><reserved><info_hash><peer_id>
```

消息格式中一些参数的含义如表13-9所示。

表 13-9 握手消息

参 数	含 义
pstrlen	pstr 的长度，该值固定为 19
pstr	BitTorrent 协议的关键字，即“BitTorrent protocol”
reserved	占 8 字节，用于扩展 BT 协议，一般这 8 字节都设置为 0。有些 BT 软件对 BT 协议进行了某些扩展，因此可能看到有些 peer 发来的握手消息这 8 个字节不全为 0，不过不必理会，这不会影响正常的通信
info_hash	与发往 Tracker 的 GET 请求中的 info_hash 为同一个值，长度固定为 20 字节

peer_id与发往 Tracker 的GET请求中的peer_id为同一个值，长度固定为20字节。一般从peer_id可以识别出 BT 软件的类型，例如，某peer发来的握手消息中peer_id的前 8 个字节为“-AZ2060-”，则可以断定对方使用的是 Azureus；若为“-BCxxxx-”，x 为数字，则对方使用的是BitComet。

对于除握手消息之外的其他所有消息，其一般的格式为：

```
<length prefix><message ID><payload>
```

length prefix （长度前缀）占 4 个字节，指明 message ID 和payload的长度和。 message ID （消息编号）占一字节，是一个 10进制的整数，指明消息的编号。 payload（负载），长度未定，是消息的内容。

- keep_alive 消息：<len=0000>

keep_alive 消息的长度固定，为 4 字节，它没有消息编号和负载。如果一段时间内客户端与 peer 没有交换任何消息，则与这个 peer 的连接将被关闭。 keep_alive 消息用于维持这个连接，通常如果 2 分钟内没有向 peer 发送任何消息，则发送一个 keep_alive 消息。

- choke 消息：<len=0001><id=0>

choke 消息的长度固定，为 5 字节，消息长度占 4 个字节，消息编号占 1 个字节，没有负载。该消息的功能是，发出该消息的 peer 将接收该消息的 peer 阻塞，暂时不允许其下载自己的数据。

- unchoke 消息：<len=0001><id=1>

unchoke 消息的长度固定，为 5 字节，消息长度占 4 个字节，消息编号占 1 个字节，没有负载。客户端每隔一定的时间，通常为 10秒，计算一次各个 peer 的下载速度，如果某 peer 被解除阻塞，则发送 unchoke 消息。如果某个 peer 原先是解除阻塞的，而此次被阻塞，则发送 choke 消息。

- interested消息：<len=0001><id=2>

interested消息的长度固定，为 5 字节，消息长度占 4 个字节，消息编号占 1 个字节，没有负载。当客户端收到某 peer 的have消息时，如果发现 peer 拥有了客户端没有的 piece ，则发送 interested消息告知该 peer ，客户端对它感兴趣。

- not interested 消息：<len=0001><id=3>

not interested 消息的长度固定，为 5 字节，消息长度占 4 个字节，消息编号占 1 个字节，没有负载。当客户端下载了某个 piece ，如果发现客户端拥有了这个 piece 后，某个 peer 拥有的所有 piece ，客户端都拥有，则发送 not interested 消息给该 peer 。

- have消息：<len=0005><id=4><piece index>

have消息的长度固定，为 9 字节，消息长度占 4 个字节，消息编号占 1 个字节，负载为 4 个字节。负载为一个整数，指明下标为 index 的 piece ， peer 已经拥有。每当客户端下载了一个 piece ，即将该 piece 的下标作为 have消息的负载构造 have消息，并把该消息发送给所有与客户端建立连接的 peer 。

- bitfield消息： <len=0001+X><id=5><bitfield>

bitfield消息的长度不固定，其中 X 是 bitfield(即位图) 的长度。当客户端与 peer 交换握手消息之后，就交换位图。位图中，每个 piece 占一位，若该位的值为 1 ，则表明已经拥有该 piece ；为 0 则表明该 piece 尚未下载。具体而言，假定某共享文件共拥有 801 个 piece ，则位图为 101 个字节，位图的第一个字节的最高位指明第一个 piece 是否拥有，位图的第一个字节的第二高位指明第二个 piece 是否拥有，依此类推。对于第 801 个 piece ，需要单独一个字节，该字节的最高位指明第 801 个 piece 是否已被下载，其余的 7 位放弃不予使用。

- request消息： <len=0013><id=6><index><begin><length>

request消息的长度固定，为17个字节， index 是 piece 的索引， begin 是 piece 内的偏移，

length 是请求peer发送的数据的长度。当客户端收到某个peer发来的 unchoke 消息后，即构造request消息，向该peer发送数据请求。前面提到， peer之间交换数据是以slice（长度为 16KB 的块）为单位的，因此request消息中 length 的值一般为16K。对于一个256KB的 piece ，客户端分16次下载，每次下载一个16K的slice。

piece 消息： <len=0009+X><id=7><index><begin><block>

piece消息是另外一个长度不固定的消息，长度前缀中的 9 是id 、 index 、 begin 的长度总和， index 和 begin 固定为 4 字节， X为 block 的长度，一般为 16K。因此对于piece消息，长度前缀加上id通常为 00 00 40 09 07 。当客户端收到某个 peer的request消息后，如果判定当前未将该 peer阻塞，且peer请求的slice，客户端已经下载，则发送piece消息将文件数据上传给该peer。

- cancel 消息： <len=0013><id<=8><index><begin><length>

cancel 消息的长度固定，为17个字节， len 、 index 、 begin 、 length 都占 4 字节。它与request消息对应，作用刚好相反，用于取消对某个slice的数据请求。如果客户端发现，某个 piece 中的slice，客户端已经下载，而客户端又向其他peer发送了对该slice的请求，则向该peer发送 cancel 消息，以取消对该slice的请求。事实上，如果算法设计合理，基本不用发送 cancel 消息，只在某些特殊的情况下才需要发送 cancel 消息。

- port 消息： <len=0003><id=9><listen-port>

port 消息的长度固定，为 7 字节，其中 listen-port 占两个字节。该消息只在支持 DHT的客户端中才会使用，用于指明DHT监听的端口号，一般不必理会，收到该消息时，直接丢弃即可。

13.2.7 关键算法和策略

1．流水线作业

BT 协议作为一种构建在TCP协议上的应用层协议，可以通过流水线作业来提高数据传输的效率。具体而言，当客户端向peer发送数据请求时（即发送request消息），一次请求多个slice（即在一个数据包中发送多个request消息请求多个slice）。假如客户端一次只发送一个slice请求，则peer给客户端发送完一个slice的数据后就进入等待，等待客户端发送新的数据请求。如果一次发送多个slice请求，则peer发送完一个slice后接着发送下一个slice，从而避免了等待，提高了数据传输的效率。事实上， HTTP 协议的 1.1 版本就广泛地使用了流水线作业的思想，大大地提高了浏览器和Web服务器之间的传输效率。

2．片断选择算法

一个良好的片断选择策略对于提高下载速度至关重要，对于提高整个文件共享系统的性能也有重要影响。

片断选择的第一个策略是，一旦向某个peer发送对某个 piece 中的slice的请求后，则该 piece 中的其他slice也从该peer处下载，这样可以尽快地下载到一个完整的 piece 。因为某个peer拥有某个 piece 中的一个slice，则它必定拥有该 piece 的其他slice，并且如果peer愿意发送一个slice给客户端，它也应该愿意发送 piece 中的其他slice给客户端。该策略也被称为严格的优先级。

片断选择的第二个策略是，最少优先。即某个 piece 在所有peer中的拥有率最低，则优先下载该 piece 。这么做的优点是，第一，可以防止拥有这个 piece 的peer突然离开，导致某个 piece 的缺失，从而当前任何一个参与下载的peer都不能下载到一份完整的文件；第二，如果下

载了某些拥有率较低的 piece ，则其他很多peer会向客户端请求数据，而要想从客户端下载到数据，那些peer就要提供数据给客户端下载，这样对于提高客户端的下载速度也是有帮助的。对于这个共享系统而言，优先下载拥有率较低的 piece 可以使得整个系统提高每个 piece 的拥有度，整个系统会趋向于最优。如果所有peer优先下载拥有率较高的 piece ，会使某些 piece 的拥有率进一步降低，而拥有这些低拥有率 piece 的peer一旦离开共享系统，则整个文件会越来越不完整，最后导致许多peer不能下载到一个完整的文件拷贝。

片断选择的第三个策略是，随机选择第一个要下载的 piece 。开始下载时，不能采用最少优先策略。原因在于，采用最少优先策略，如果某个 piece 的拥有率很低，那么下载到这个 piece 就相对较难。如果随机选择一个 piece ，那么更容易下载到该 piece ，一旦客户端下载到一个完整的 piece ，就可以提供给其他 peer 下载，而由于客户端向其他 peer 上传数据，会导致其他 peer 对客户端解除阻塞，从而有利于在起始阶段获得较高的下载速度。当然在下载到一些 piece 后，客户端应该采用最少优先策略来下载数据，这虽然会导致客户端的下载速度在短期内有所下降，但随后下载速度会有较大提高。

片断选择的第四个策略是，最后阶段模式。有时，从一个传输速度很慢的peer处下载一个 piece 会花费很长时间，在下载的过程中这不是什么大问题。但在下载接近完成时，如果发生这种情况，会导致客户端迟迟不能下载完成。为了解决这个问题，在最后阶段，客户端向所有peer发送对这个 piece 的某些slice的请求，一旦收到某个peer发来的slice，则向其他peer发送 cancel 消息，只从当前这个peer处下载。

3．阻塞算法

BT 并不集中分配资源，每个peer有责任尽可能地提高自己的下载速度。peer从它可以连接的peer下载文件，并根据对方提供的下载速率给予同等的上传回报，对于合作者，提供上传服务，对于不合作的，就阻塞对方。阻塞是一种临时拒绝上传的策略，虽然上传停止了，但是下载仍然继续。在解除阻塞时，连接并不需要重新建立。因为阻塞过程中只是拒绝传输 piece 消息，其他消息，如have消息，interested消息仍可以传输。阻塞算法虽然不是 BT 协议一部分，但是它对提高性能是必要的。

每个客户端一直与固定数量的peer保持疏通（通常是 4 个），那么以什么方式来决定是否保持与某个peer疏通呢？通常的做法是，严格地根据当前的下载速度来决定哪些peer应该保持疏通。但计算当前下载速度是个大难题。当前的实现通常是计算最近10秒从每个peer处下载数据的速度。以10秒为间隔重新选择保持疏通（即解除阻塞）的peer，是为了避免频繁地阻塞和解阻塞，造成资源的浪费。实践表明，10秒足以使下载速度达到最大。

如果只是简单地为提供最高下载速率的 4 个peer提供上载服务，那么就没有办法发现那些空闲的连接是否有更好的下载速度。为了解决这个问题，在任何时候，每个 peer 都拥有一个称为“ optimistic unchoking （优化非阻塞）” peer，这个连接总是保持疏通状态，而不管它的下载速率是多少。每隔30秒，重新选择一个 peer 作为优化非阻塞 peer 。30秒足以让该 peer 的上载能力达到最大。

一旦某个 peer 完成了下载，它不能再通过下载速率（因为下载速率已经为 0 了）来决定为哪些peer提供上载了。目前采用的解决办法是，优先选择那些从它这里得到更好下载速率的 peer，保持与它们疏通。这样做的理由是尽可能的利用上载带宽。一旦某个 peer 完成了下载，那么它也就成为了种子。种子拥有一份完整的文件拷贝，并提供给其他 peer 下载。为了整个系统的性能，

每个peer在完成下载后应该作为种子存在一段时间，作为对整个系统的回报。原始种子，即最初提供文件进行共享、并制作生成了种子文件，并把种子文件发布到 Web服务器的种子，它至少应该存在到系统中生成另外一个种子时才能离开，否则当前参与下载的所有 peer都不能获得一份完整的文件拷贝。

本节对 BT 协议的解释和分析参考了“ BT 协议规范”(Bittorrent Protocol Specification)和 BT 协议设计者所著的“ BT 性能卓越的原因”(Incentives Build Robustness in BitTorrent)一文。

13.3 系统结构设计

整个系统的模块结构如图13-1所示。

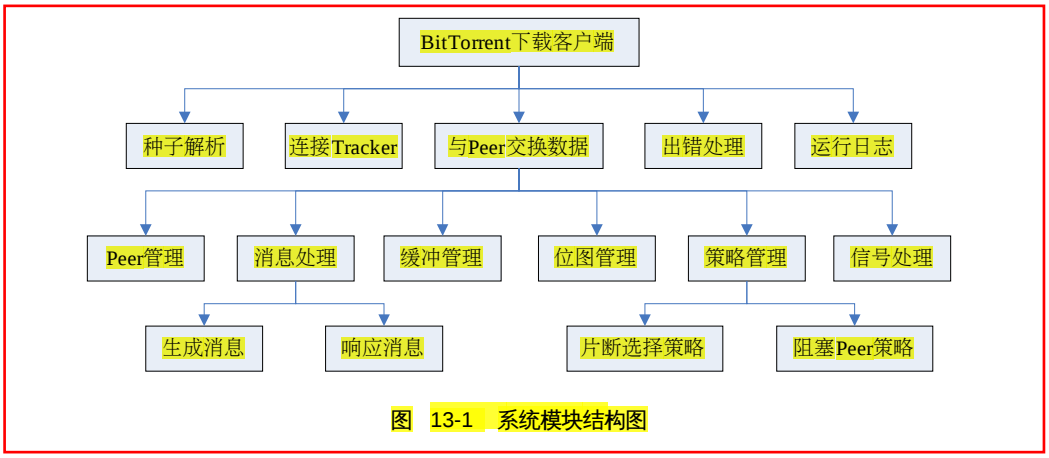

图 13-1 系统模块结构图

整个系统各个模块的功能如下。

（1）种子解析：负责解析种子文件，从中获取 Tracker 服务器的地址，待下载文件的文件名和长度， piece 长度，各个 piece 的hash值。

（2）连接Tracker：根据 HTTP 协议构造获取 Peer 地址的请求，与 Tracker建立连接，解析Tracker的回应消息，从而获取各个peer的IP地址和端口号。

（3）与peer交换数据：根据peer的IP地址和端口号连接peer 、从peer处下载数据并将已下载的数据上传给peer 。

（4）出错处理：定义整个系统可能出现的错误类型，并对错误进行处理。

（5）运行日志：记录程序运行的日志，并保存到文件中以备查看和分析。

模块“与peer交换数据”是本系统的核心和主要构成部分，它又可以划分成如下几个子模块。

（1） peer管理：系统为每一个已建立 TCP连接的peer构造一个peer结构体。该结构体的主要成员有： peer的IP地址和端口号、与该 peer进行通信的套接字、该 peer的id 、当前所处的状态、发送缓冲区、接收缓冲区、数据请求队列、数据被请求队列、从该 peer处已下载的数据量和向该peer上传的数据量、下载速度和上传速度。本模块负责管理peer链表，添加和删除peer结点。

（2）消息处理： peer与peer之间以发送和接收消息的方式进行通信。本模块负责根据当前的状态生成并发送消息，接收并处理消息。 BitTorrent 协议共定义了12种消息，其中对下载和上

传数据最重要的是request消息和 piece 消息。request消息向peer发送数据请求，指明请求的是哪个 piece 的哪个slice。Peer接收到request消息后根据当前的状态，决定是否发送数据给对方。如果允许发送，则构造 piece 消息，数据被封装在该消息中。每当下载完一个正确的 piece 时，就向所有peer发送have消息通告已获得该 piece ，其他peer如果没有该 piece 就可以向peer发送数据请求，每次请求都是以slice为单位。

（3）缓冲管理：如果下载完一个 piece 就立即写入硬盘，这样会导致频繁读写硬盘，既影响速度（读写磁盘要花费较多的时间），又不利于保护硬盘（频繁读写磁盘会使硬盘寿命缩短）。为了解决这个问题，几乎所有的 BT 软件都在程序中增加了一个缓冲管理模块。将下载到的数据先缓存起来，等到下载到一定量的数据后再集中写入硬盘。peer请求一个slice的数据时，先将该slice所在的整个 piece 读入到缓冲区中，下次Peer再请求该 piece 的其他slice时，只常缓冲区中获取，避免了频繁读写硬盘。本模块负责维护一个 16MB 的缓冲区（大小可调），将下载到的数据保存在缓冲区中，并在适当时刻写入硬盘的文件中。

（4）位图管理： BT 协议采用位图指明当前哪些 piece 已经下载，哪些 piece 还没有下载。每个 piece 占一位，值为 0 表示该 piece 还未下载到，为 1 则表明已经下载到该 piece 。本模块负责管理位图，客户端与peer建立了连接并进行握手之后，即发送位图给peer告知已下载到哪些 piece ，同时也接收对方的位图并将其保存在Peer结构体中。每下载到一个 piece 就更新自己的位图，并发送have消息给所有已建立连接的peer。每当接收到peer发来的have消息就更新该peer的位图。

（5）策略管理： BT 协议的设计者为了保证整体性能而制定了许多策略，这些策略虽然没有写入 BT 协议，但已经成为事实上的标准， BT 软件开发者一般都使用这些策略来保证程序的性能。本部分负责策略的管理，主要是计算各个peer的下载和上传速度，根据下载速度选择非阻塞peer，采用随机算法选择优化非阻塞peer，以及实现片断选择策略。

（6）信号处理：在运行过程中，程序可能会接收到一些信号，如 SIGINT 、 SIGTERM ，这些信号的默认动作是立即终止程序。这并不是所期望的。在程序终止前需要作一些处理，如释放动态申请的内存、关闭文件描述符、关闭套接字。

13.4 各个模块的设计和实现

13.4.1 种子解析模块的设计和实现

解析种子文件主要在 parse_metafile.h 和 parse_metafile.c 中完成。 parse_metafile.h 文件的代码内容为：

```
parse_metafile.h
#ifndef PARSE_METAFILE
#define PARSE_METAFILE

// 保存从种子文件中获取的 tracker 的 URL
typedef struct _Announce_list {
```

```
        char    announce[128];
        struct  _Announce_list  *next;
    } Announce_list;

    //  保存各个待下载文件的路径和长度
    typedef struct _Files {
        char    path[256];
        long    length;
        struct  _Files  *next;
    } Files;

    int read_metafile(char *metafile_name);                  //  读取种子文件
    int find_keyword(char *keyword,long *position);      //  在种子文件中查找某个关键词
    int read_announce_list();                                //  获取各个  tracker 服务器的地址
    int add_an_announce(char* url);                          //  向 tracker 列表添加一个 URL

    int get_piece_length();       //  获取每个  piece 的长度 ，一般为  256KB
    int get_pieces();                 //  读取各个  piece 的哈希值

    int is_multi_files();         //  判断下载的是单个文件还是多个文件
    int get_file_name();              //  获取文件名，对于多文件，获取的是目录名
    int get_file_length();        //  获取待下载文件的总长度
    int get_files_length_path();      //  获取文件的路径和长度，对多文件种子有效

    int get_info_hash();              //  由 info关键词对应的值计算  info_hash
    int get_peer_id();                //  生成 peer_id，每个 peer 都有一个  20 字节的 peer_id

    void release_memory_in_parse_metafile();//  释放 parse_metafile.c 中动态分配的内存
    int  parse_metafile(char *metafile);    //  调用本文件中定义的函数 ，完成解析种子文件

    #endif
```

以下是 parse_metafile.c 文件的头部，主要是包含了一些头文件和定义一些全局变量，各个函数的定义将在后面列出。

```
    parse_metafile.c
    #include <stdio.h>
    #include <ctype.h>
```

```
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "parse_metafile.h"
#include "sha1.h"

char    *metafile_content = NULL;     //  保存种子文件的内容
long    filesize;                     //  种子文件的长度

int      piece_length    = 0;     //  每个  piece 的长度  , 通常为  256KB 即 262144字节
char    *pieces = NULL;       //  保存每个  pieces的哈希值   , 每个哈希值为 20字节
int      pieces_length = 0;       //  缓冲区  pieces的长度

int          multi_file = 0;           //  指明是单文件还是多文件
char         *file_name = NULL;        //  对于单文件 , 存放文件名 ; 对于多文件 , 存放目录名
long long file_length = 0;             //  存放待下载文件的总长度
Files        *files_head = NULL;       //  只对多文件种子有效   , 存放各个文件的路径和长度

unsigned    char info_hash[20];             //  保存  info_hash 的值  , 连接  tracker
和 peer时使用
unsigned    char peer_id[20];               //  保存  peer_id 的值  , 连接 peer时使用

Announce_list *announce_list_head = NULL;   //  用于保存所有 tracker 服务器的   URL
```

下面对解析种子文件中用到函数功能解释如下。

- int read_metafile(char *metafile_name)

功能：解析种子文件

参数：metafile_name是种子文件名。

返回：处理成功返回 0 ， 否则返回－ 1

附注：将种子文件的内容读入到全局变量metafile_content所指向的缓冲区中以方便处理。该函数的实现代码为：

```
int read_metafile(char *metafile_name)
{
    long  i;

    //  以二进制、只读的方式打开文件
    FILE *fp = fopen(metafile_name,"rb");
    if(fp == NULL) {
```

```
            printf("%s:%d can not open file\n",__FILE__,__LINE__);
            return -1;
        }

        // 获取种子文件的长度，filesize为全局变量，在 parse_metafile.c 头部定义
        fseek(fp,0,SEEK_END);
        filesize = ftell(fp);
        if(filesize == -1) {
            printf("%s:%d fseek failed\n",__FILE__,__LINE__);
            return -1;
        }
        metafile_content = (char *)malloc(filesize+1);
        if(metafile_content == NULL) {
            printf("%s:%d malloc failed\n",__FILE__,__LINE__);
            return -1;
        }

        // 读取种子文件的内容到 metafile_content 缓冲区中
        fseek(fp,0,SEEK_SET);
        for(i = 0; i < filesize; i++)
            metafile_content[i] = fgetc(fp);
        metafile_content[i] = '\0';

        fclose(fp);

    #ifdef DEBUG
        printf("metafile size is: %ld\n",filesize);
    #endif
        return 0;
    }
```

函数代码说明。

（1）编译器预定义的宏 __FILE__ 和 __LINE__ 在程序中可以直接使用。__FILE__ 代表该宏所在的源文件的文件名，在源文件 parse_metafile.c 中该宏的值等于“ parse_metafile.c”，宏 __LINE__ 的值为 __LINE__ 所在行的行号。

（2）种子文件必须以二进制的方式打开，否则如果以字符方式打开可能无法读取整个文件的内容。无法读取的原因在于 piece 的hash值中可能含有字符0x00，若文件以字符形式打开，遇到该字符，库函数就认为文件已经结束。

（3）增加“#ifdef DEBUG #endif”，主要是为了方便调试。如果在 parse_metafile.c 文件的头部增加宏定义语句“ #define DEBUG”，则程序运行时将执行“ #ifdef DEBUG” 和“#endif”之

间的语句。在软件开发阶段，可以使用“ #define DEBUG”来打印和查看某些关键的值，开发完毕，去掉该宏，则打印语句不会执行。

- int find_keyword(char *keyword,long *position)

功能：从种子文件中查找某个关键字

参数： keyword 为要查找的关键字， position 用于返回关键字第一个字符所在的下标。

返回：成功执行并找到关键字返回 1，未找到返回 0，执行失败返回－ 1 。函数实现代码如下所示：

```
int find_keyword(char *keyword,long *position)
{
    long i;

    *position = -1;
    if(keyword == NULL)  return 0;

    for(i = 0; i < filesize-strlen(keyword); i++) {
        if( memcmp(&metafile_content[i], keyword, strlen(keyword)) == 0 ) {
            *position = i;
            return 1;
        }
    }

    return 0;
}
```

函数代码说明。

该函数在种子文件解析模块的源文件 parse_metafile.c 中被频繁使用，用于查找某些关键字。例如，关键字“ 8:announce”和“ 13:announce-list”之后都是 Tracker 服务器的地址，找到该关键字后，便可以获取 Tracker 的地址。

- read_announce_list()

功能：获取 Tracker 地址，并将获取的地址保存到全局变量announce_list_head指向的链表中。该函数实现代码如下：

```
int read_announce_list()
{
    Announce_list  *node = NULL;
    Announce_list  *p    = NULL;
    int            len   = 0;
    long           i;

    if( find_keyword("13:announce-list",&i) == 0 ) {
        if( find_keyword("8:announce",&i) == 1 ) {
```

```
                i = i + strlen("8:announce");
                while( isdigit(metafile_content[i]) ) {
                    len = len * 10 + (metafile_content[i] - '0');
                    i++;
                }
                i++;  //  跳过  ':'

                node = (Announce_list *)malloc(sizeof(Announce_list));
                strncpy(node->announce,&metafile_content[i],len);
                node->announce[len] = '\0';
                node->next = NULL;
                announce_list_head = node;
            }
        }
        else {  //  如果有  13:announce-list 关键词就不用处理  8:announce 关键词
            i = i + strlen("13:announce-list");
            i++;         //  跳过  'l'
            while(metafile_content[i] != 'e') {
                i++;     //  跳过  'l'
                while( isdigit(metafile_content[i]) ) {
                    len = len * 10 + (metafile_content[i] - '0');
                    i++;
                }
                if( metafile_content[i] == ':' )  i++;
                else  return -1;

                //  只处理以  http开头的  tracker 地址 , 不处理以  udp 开头的地址
                if( memcmp(&metafile_content[i],"http",4) == 0 ) {
                    node = (Announce_list *)malloc(sizeof(Announce_list));
                    strncpy(node->announce,&metafile_content[i],len);
                    node->announce[len] = '\0';
                    node->next = NULL;

                    if(announce_list_head == NULL)
                        announce_list_head = node;
                    else {
                        p = announce_list_head;
                        while( p->next != NULL) p = p->next; //  使 p 指针指向最后个结
点
```

```
                        p->next = node; // node 成为 tracker 列表的最后一个结点
                    }
                }

                i = i + len;
                len = 0;
                i++;    // 跳过 'e'
                if(i >= filesize)  return -1;
            } // while 循环结束
        }

    #ifdef DEBUG
        p = announce_list_head;
        while(p != NULL) {
            printf("%s\n",p->announce);
            p = p->next;
        }
    #endif
        return 0;
    }
```

程序说明。

（1）下面是某种子文件开头的一部分，请对照它来理解read_announce_list函数

```
d8:announce32:http://tk.greedland.net/announce13:announce-
listll32:http://tk.greedland.net/announceel33:http://tk2.greedland.net/announceee...
```

第一个字符‘d’是B编码中字典的起始符，接着是关键字“ 8:announce”，该关键字是一个长度为 8 的字符串，其对应的值为长度为32的字符串“32:http://tk.greedland.net/announce”，它是一个 Tracker 服务器的URL，接着是关键字“ 13:announce-list”，该关键字对应的值是一个列表，因为关键字“ 13:announce-list”之后的第一个字符为列表的起始字符‘ l’，该列表中含有两个元素，这两个元素的类型也都是列表。

如果有关键字“ 13:announce-list”就不用处理关键字“ 8:announce”的原因在于，前者对应的值中必定包含后者对应的值。

（2）“#ifdef DEBUG”和“#endif”之间的语句用于打印各个 Tracker 的URL。

- int add_an_announce(char *url)

功能：连接某些 Tracker 时会返回一个重定向的URL，需要连接该URL才能获取peer。函数实现代码如下：

```
int add_an_announce(char *url)
{
Announce_list *p = announce_list_head, *q;
```

```
        //  若参数指定的 URL 在 Tracker 列表中已存在 , 则无需添加
        while(p != NULL) {
             if(strcmp(p->announce,url) == 0)  break;
            p = p->next;
        }
        if(p != NULL)  return 0;

        q = (Announce_list *)malloc(sizeof(Announce_list));
        strcpy(q->announce,url);
         q->next = NULL;

         p = announce_list_head;
        if(p == NULL)  { announce_list_head = q; return 1; }
         while(p->next != NULL)  p = p->next;
        p->next = q;
         return 1;
    }
```

- int is_multi_files()

功能：判断是下载多个文件还是单文件，若含有关键字“5:files”则说明下载的是多个文件。函数实现的代码如下：

```
    int is_multi_files()
    {
        long i;

        if( find_keyword("5:files",&i) == 1 ) {
             multi_file = 1;
             return 1;
        }

    #ifdef DEBUG
        printf("is_multi_files:%d\n",multi_file);
    #endif

         return 0;
    }
```

- int get_piece_length()

功能：获取 piece 的长度。函数实现的代码如下：

```
    int get_piece_length()
    {
```

```
    long i;

    if( find_keyword("12:piece length",&i) == 1 ) {
        i = i + strlen("12:piece length");  // 跳过 "12:piece length"
        i++;  //  跳过 'i'
        while(metafile_content[i] != 'e') {
            piece_length = piece_length * 10 + (metafile_content[i] - '0');
            i++;
        }
    } else {
        return -1;
    }

#ifdef DEBUG
    printf("piece length:%d\n",piece_length);
#endif
    return 0;
}
```

程序说明。

以下是某种子文件的一部分：12:piece lengthi262144e6:pieces16900:...

从中可以看到，关键字“ 12:piece length”后面跟一个 B 编码的整型数（以 i 作为起始字符，以 e 作为终结字符）。262144（ 256K ），说明每个 piece 的长度都是 256KB（最后一个 piece 除外）。接着是关键字“ 6:pieces”，它对应的值是一个 B 编码的字符串，存放各个 piece 的 hash 值， 16900 是字符串的长度，该字符串长度除以 20即为 piece 数，因为每个 piece 的 hash 值为固定的20字节。

- get_pieces()

功能：获取每个 piece 的 hash 值，并保存到 pieces 所指向的缓冲区中。函数实现的代码如下：

```
int get_pieces()
{
    long i;

    if( find_keyword("6:pieces", &i) == 1 ) {
        i = i + 8;                // 跳过 "6:pieces"
        while(metafile_content[i] != ':') {
            pieces_length = pieces_length * 10 + (metafile_content[i] - '0');
            i++;
        }
        i++;                      // 跳过 ':'
        pieces = (char *)malloc(pieces_length+1);
```

```
            memcpy(pieces,&metafile_content[i],pieces_length);
            pieces[pieces_length] = '\0';
        } else {
            return -1;
        }
    #ifdef DEBUG
        printf("get_pieces ok\n");
    #endif
        return 0;
    }
```

- get_file_name()

功能：获取待下载的文件的文件名，如果下载的是多个文件，则获取的是目录名。函数实现的代码如下：

```
    int get_file_name()
    {
        long  i;
        int   count = 0;

        if( find_keyword("4:name", &i) == 1 ) {
            i = i + 6;  //  跳过 "4:name"
            while(metafile_content[i] != ':') {
                count = count * 10 + (metafile_content[i] - '0');
                i++;
            }
            i++;        //  跳过 ':'
            file_name = (char *)malloc(count+1);
            memcpy(file_name,&metafile_content[i],count);
            file_name[count] = '\0';
        } else {
            return -1;
        }

    #ifdef DEBUG
        printf("file_name:%s\n",file_name);
    #endif
        return 0;
    }
```

程序说明。

以下是一个完整的较为简单的种子文件：

```
      d8:announce32:http://tk.greedland.net/announce13:announce-
listll32:http://tk.greedland.net/annou
      nceel33:http://tk2.greedland.net/announceee13:creation
datei1187968874e4:infod6:lengthi119861
      306e4:name31:[ymer][naruto][246][jp_cn].rmvb10:name.utf-831:[ymer][naruto][246]
[jp_cn].rmv
       b12:piece lengthi262144e6:pieces9160:...ee
```

关键字“13:creation date”之前的部分已经在介绍read_announce_list函数时分析过了，此处不再赘述。关键字“13:creation date”及其对应的值“ i1187968874e”，它指明了创建种子文件的时间。我们注意到时间是一个整数，它是自1970年 1 月 1 日到种子文件创建时所经过的秒数，Linux 中有专门的库函数处理这种表示类型的时间。

关键字“4:info”对应的值是一个字典，因为该关键字之后的第一个字符是 B 编码中字典的起始符‘d’，与该起始符对应的终止符是文件末尾的倒数第二个‘ e’。计算info_hash时，就是以关键字“4:info”对应的值作为输入，计算其 hash 值，将得到的值作为 info_hash 。文件末尾最后一个字符‘e’与文件开头的’d’对应，因此整个种子文件就是一个B编码的字典。

关键字“6:length”对应的值是待下载文件的长度，以字节为单位，可以大致地确定待下载文件的长度为 119MB 。

关键字“ 4:name” 对应的值为待下载的文件的文件名，在这个种子文件中没有关键字“5:files”说明待下载的是单文件。

关键字“ 10:name.utf-8” 对应的值也是待下载文件的文件名，只不过以 UTF-8 的形式表示，UTF-8的形式可以表示宽字符，即中文、日文、朝鲜文等字符。

- int get_file_length()

功能：获取待下载文件的长度。函数实现的代码如下：

```
int get_file_length()
{
    long i;

    if(is_multi_files() == 1)  {
        if(files_head == NULL)  get_files_length_path();
        Files *p = files_head;
        while(p != NULL) { file_length += p->length; p = p->next; }
    } else {
        if( find_keyword("6:length",&i) == 1 ) {
            i = i + 8;  //  跳过 "6:length"
            i++;        //  跳过 'i'
            while(metafile_content[i] != 'e') {
                file_length = file_length * 10 + (metafile_content[i] - '0');
                i++;
            }
```

```
        }
    }

#ifdef DEBUG
    printf("file_length:%lld\n",file_length);
#endif
    return 0;
}
```

- get_files_length_path()

功能：对于多文件，获取各个文件的路径以及长度。函数实现的代码如下：

```
int get_files_length_path()
{
    long   i;
    int    length;
    int    count;
    Files  *node  = NULL;
    Files  *p     = NULL;

    if(is_multi_files() != 1) {
        return 0;
    }

    for(i = 0; i < filesize-8; i++) {
        if( memcmp(&metafile_content[i],"6:length",8) == 0 )
        {
            i = i + 8;  //  跳过 "6:length"
            i++;        //  跳过 'i'
            length = 0;
            while(metafile_content[i] != 'e') {
                length = length * 10 + (metafile_content[i] - '0');
                i++;
            }
            node = (Files *)malloc(sizeof(Files));
            node->length = length;
            node->next = NULL;
            if(files_head == NULL)
                files_head = node;
            else {
                p = files_head;
```

```
                    while(p->next != NULL) p = p->next;
                     p->next = node;
                }
            }
            if( memcmp(&metafile_content[i],"4:path",6) == 0 )
            {
                 i = i + 6;  //  跳过 "4:path"
                 i++;        //  跳过 'l'
                count = 0;
                while(metafile_content[i] != ':') {
                     count = count * 10 + (metafile_content[i] - '0');
                    i++;
                }
                 i++;        //  跳过 ':'
                 p = files_head;
                while(p->next != NULL) p = p->next;
                 memcpy(p->path,&metafile_content[i],count);
                 *(p->path + count) = '\0';
            }
        }

        return 0;
    }
```

程序说明。

图13-2是一个多文件种子的一部分，可以参照图13-2理解 get_files_length_path 函数。

多文件种子的关键字“5:files”对应的值是比较复杂的。关键字“5:files”说明这是一个多文件种子，它对应的值是一个列表，列表的每个元素是字典，每个字典代表一个待下载文件。

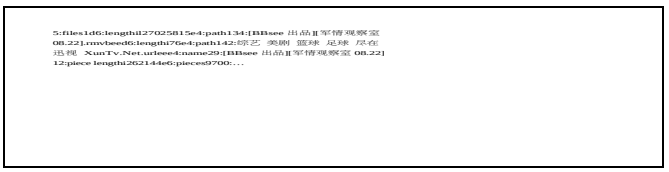

图 13-2 多文件种子示例

“5:filesl”及字符‘d’之后，有一个关键字“6:length”及其值“i127025815e”，然后是关键字“4:path”，其值为一个列表“l34:[BBsee 出品][军情观察室08.22].rmvbe”，“rmvbee”中最后一个‘e’与字符‘d’对应。

然后“d6:lengthi76e4:pathl42:综艺 美剧 篮球 足球 尽在迅视 XunTv.Net.urlee”又是一个字典。“urleee”中最后一个‘e’与“5:files”后的‘l’构成一个列表。“4:name”所跟的是目录名，然后是“12:piece length”关键字，“6:pieces”关键字。

从中可以总结出：有一个目录名“[BBsee出品][军情观察室 08.22]”，其中存放了两个文件“[BBsee 出品][军情观察室 08.22].rmvb”和“综艺 美剧 篮球 足球 尽在迅视 XunTv.Net.url”，长度分别为 127025815 字节和76字节。

int get_info_hash ()

功能：计算info_hash的值。函数实现代码如下：

```
int get_info_hash()
{
    int   push_pop = 0;
    long  i, begin, end;

    if(metafile_content == NULL)  return -1;
    // begin的值表示的是关键字  "4:info"对应值的起始下标
    if( find_keyword("4:info",&i) == 1 )  begin = i+6;
  else  return -1;

    i = i + 6;           //  跳过 "4:info"
    for(; i < filesize; )
        if(metafile_content[i] == 'd') {
            push_pop++;
            i++;
        } else if(metafile_content[i] == 'l') {
            push_pop++;
            i++;
        } else if(metafile_content[i] == 'i') {
            i++;     //  跳过 i
            if(i == filesize)  return -1;
            while(metafile_content[i] != 'e') {
                if((i+1) == filesize)  return -1;
                else i++;
            }
            i++;     //  跳过 e
  } else if((metafile_content[i] >= '0') && (metafile_content[i] <= '9')) {
            int number = 0;
            while((metafile_content[i] >= '0') && (metafile_content[i] <= '9'))
{
                number = number * 10 + metafile_content[i] - '0';
                i++;
            }
            i++;     //  跳过‘！’ :
```

```
                i = i + number;
    } else if(metafile_content[i] == 'e') {
                push_pop--;
                if(push_pop == 0) { end = i; break; }
                else  i++;
            } else {
                return -1;
            }
        if(i == filesize)  return -1;

         SHA1_CTX context;
         SHA1Init(&context);
        SHA1Update(&context, &metafile_content[begin], end-begin+1);
        SHA1Final(info_hash, &context);

    #ifdef DEBUG
         printf("info_hash:");
        for(i = 0; i < 20; i++)
            printf("%.2x ",info_hash[i]);
         printf("\n");
    #endif
         return 0;
    }
```

程序说明。

（1）在种子文件解析模块，由种子文件的内容计算info_hash的值是比较复杂的。前面已经提到，由关键字“4:info”对应的值来计算info_hash，该关键字对应的值是一个B编码的字典，问题的关键在于找到与“4:info”之后的‘d’对应的‘e’。

get_info_hash 函数中找到所需要的‘ e’的思路是：在“ 4:info”之后，每当遇到字典的起始符‘d’，则将 push_pop 的值加 1(push_pop 初始值为 0)，遇到列表的起始符‘ l’也作相同处理；遇到整数的起始符‘ i’则一直扫描直到找到与之对应的终结符‘ e’：遇到一个 0 ～ 9 的数字说明接下来是一个字符串，跳过该字符串继续扫描；遇到‘ e’则将 push_pop 值减 1，如果减 1 后，push_pop 值为 0，说明已经找到了与‘ d’匹配的‘e’。其思路类似于使用数据结构中的“栈”进行括号匹配操作。

（2）以“SHA1”开头的变量和函数用于计算一段文本的hash值，这些变量和函数的定义在 sha1.h 和 sha1.c 文件中， hash 值的计算原理不必深究。计算 hash 值的这段代码的功能是以 metafile_content[begin] ～metafile_content[end]这 end-begin+1 个字符作为输入，计算其hash值，并把结果保存到info_hash所指向的数组中。

- int get_peer_id()

功能：生成一个惟一的peer id。函数实现代码如下：

```
int get_peer_id()
{
    //  设置产生随机数的种子
    srand(time(NULL));
    //  使用 rand函数生成一个随机数  ，并使用该随机数来构造peer_id
// peer_id前 8 位固定为  -TT1000-
    sprintf(peer_id,"-TT1000-%12d",rand());

#ifdef DEBUG
    printf("peer_id:%s\n",peer_id);
#endif
    return 0;
}
```

- void release_memory_in_parse_metafile()

功能：释放动态申请的内存。函数实现代码如下：

```
void release_memory_in_parse_metafile()
{
    Announce_list *p;
    Files         *q;

    if(metafile_content != NULL)      free(metafile_content);
    if(file_name != NULL)             free(file_name);
    if(pieces != NULL)                free(pieces);

    while(announce_list_head != NULL) {
        p = announce_list_head;
        announce_list_head = announce_list_head->next;
        free(p);
    }

    while(files_head != NULL) {
        q = files_head;
        files_head = files_head->next;
        free(q);
    }
}
```

- int parse_metafile(char *metafile)

功能：调用 parse_metafile.c 中定义的函数，完成解析种子文件。该函数由 main.c 调用。

返回：解析成功返回 0，否则返回-1。函数实现代码如下：

```
int parse_metafile(char *metafile)
{
    int ret;

    //  读取种子文件
    ret = read_metafile(metafile);
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    //  从种子文件中获取  tracker 服务器的地址
    ret = read_announce_list();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    //  判断是否为多文件
    ret = is_multi_files();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    //  获取每个  piece 的长度  , 一般为  256KB
    ret = get_piece_length();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    //  读取各个  piece 的哈希值
    ret = get_pieces();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    //  获取要下载的文件名，对于多文件的种子，获取的是目录名
    ret = get_file_name();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    //  对于多文件的种子，获取各个待下载的文件路径和文件长度
    ret = get_files_length_path();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    //  获取待下载的文件的总长度
    ret = get_file_length();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

    //  获得  info_hash , 生成  peer_id
    ret = get_info_hash();
    if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }
```

```
        ret = get_peer_id();
        if(ret < 0) { printf("%s:%d wrong",__FILE__,__LINE__); return -1; }

        return 0;
    }
```

13.4.2 位图管理模块的设计和实现

对位图的操作主要在bitfield.h和bitfield.c中，负责创建位图，设置和获取位图某一位的值，保存位图等。

```
    bitfield.h
    #ifndef BITFIELD_H
    #define BITFIELD_H

    typedef struct _Bitmap {
        unsigned char  *bitfield;         //  保存位图
        int           bitfield_length;    //  位图所占的总字节数
        int           valid_length;       //  位图有效的总位数  ，每一位代表一个  piece
    } Bitmap;

    int  create_bitfield();                                  //  创建位图  ，分配内存并进行初
始化
    int  get_bit_value(Bitmap *bitmap,int index);    //  获取某一位的值
    int  set_bit_value(Bitmap *bitmap,int index, unsigned char value);
    //  设置某一位的值
    int  all_zero(Bitmap *bitmap);                           //  全部清零
    int  all_set(Bitmap *bitmap);                            //  全部设置为 1
    void release_memory_in_bitfield();                       //  释放  bitfield.c 中动态分配的
内存
    int  print_bitfield(Bitmap *bitmap);                     //  打印位图值，用于调试

    int  restore_bitmap();    //  将位图存储到文件中
                              //  在下次下载时 ，先读取该文件获取已经下载的进度
    int  is_interested(Bitmap *dst,Bitmap *src);     //  拥有位图   src 的 peer是否对拥有
                                                     // dst位图的  peer感兴趣
    int  get_download_piece_num();                           //  获取当前已下载到的总 piece 数

    #endif
```

程序说明。

（1）结构体Bitmap中，bitfield_length为指针bitfield所指向的内存的长度（以字节为单位），而valid_length为位图的有效位数。例如，某位图占 100 字节，而有效位数位 795，则位图最后一个字节的最后 5 位 (100 × 8−795) 是无效的。

（2）函数 is_interested 用于判断两个 peer 是否感兴趣，如果 peer1 拥有某个 piece，而 peer2 没有，则 peer2 对 peer1 感兴趣，希望从 peer1 处下载它没有的piece。

（3）函数 get_download_piece_num 用于获得已下载的 piece 数，其方法是统计结构体Bitmap的bitfield成员所指向的内存中值为 1 的位数。

文件 bitfield.c 的头部包含的文件如下：

```
bitfield.c 文件头部包括的内容
#include <stdio.h>
 #include <unistd.h>
 #include <string.h>
 #include <malloc.h>
#include <fcntl.h>
#include <sys/types.h>
 #include <sys/stat.h>
#include "parse_metafile.h"
 #include "bitfield.h"

extern int       pieces_length;
extern char       *file_name;

Bitmap       *bitmap = NULL;           //  指向位图
int              download_piece_num = 0;     //  当前已下载的 piece 数
```

程序说明。

（1）语句“ extern int pieces_length;”声明了一个变量，这个变量是在 parse_metafile.c 中定义的全局变量。如果要在其他源文件中使用某个源文件中定义的变量，需要在使用该变量的源文件的头部以 extern 关键字声明。注意声明和定义的区别，声明仅仅是告知编译器有某个变量，而对于定义，编译器要分配内存空间来存储该变量的值。

（2）全局变量bitmap指向自己的位图，可以从位图中获知下载的进度。peer的位图存放在Peer结构体中。

- int create_bitfield()

功能：创建待下载文件的位图~~该函数较为简单，不另加注释~~

```
 int create_bitfield()
{
     bitmap = (Bitmap *)malloc(sizeof(Bitmap));
     if(bitmap == NULL) {
         printf("allocate memory for bitmap fiailed\n");
```

```
            return -1;
        }

        // pieces_length除以 20即为总的  piece 数
        bitmap->valid_length = pieces_length / 20;
        bitmap->bitfield_length = pieces_length / 20 / 8;
        if( (pieces_length/20) % 8 != 0 )  bitmap->bitfield_length++;

        bitmap->bitfield = (unsigned char *)malloc(bitmap->bitfield_length);
        if(bitmap->bitfield == NULL)  {
            printf("allocate memory for bitmap->bitfield fiailed\n");
            if(bitmap != NULL)  free(bitmap);
            return -1;
        }

        char bitmapfile[64];
        sprintf(bitmapfile,"%dbitmap",pieces_length);
        int  i;
        FILE *fp = fopen(bitmapfile,"rb");
        if(fp == NULL) {  //  若打开文件失败 ，说明开始的是一个全新的下载
            memset(bitmap->bitfield, 0, bitmap->bitfield_length);
    } else {
            fseek(fp,0,SEEK_SET);
            for(i = 0; i < bitmap->bitfield_length; i++)
                (bitmap->bitfield)[i] = fgetc(fp);
            fclose(fp);
            //  给 download_piece_num 赋新的初值
            download_piece_num = get_download_piece_num();
        }

        return 0;
    }
```

- int get_bit_value(Bitmap *bitmap,int index)

功能：获取位图中某一位的值

```
    int get_bit_value(Bitmap *bitmap,int index)
    {
        int                ret;
        int                byte_index;
        unsigned char      byte_value;
```

```
    unsigned char      inner_byte_index;

    if(bitmap==NULL || index >= bitmap->valid_length)  return -1;
    byte_index = index / 8;
    byte_value = bitmap->bitfield[byte_index];
    inner_byte_index = index % 8;

    byte_value = byte_value >> (7 - inner_byte_index);
    if(byte_value % 2 == 0) ret = 0;
    else ret = 1;

    return ret;
}
```

为了方便对 get-bit-value 函数的理解，可以假设某位图为 2 字节（其值为10110011 01010100），有效位数为14位，也就是待下载文件共有14个 piece 。位图第一个字节指明 index 为 0 ～ 7 的 piece 是否已下载，第二个字节指明 index 为 8 ～13的 piece 是否已下载。现在要判断 index 为 8 的 piece 是否已经下载，也就是要获取位图第二个字节最高位的值。

- int set_bit_value(Bitmap *bitmap,int index,unsigned char value)

功能：设置位图中某一位的值

```
int set_bit_value(Bitmap *bitmap,int index,unsigned char v)
{
    int     byte_index;
    unsigned charinner_byte_index;

    if(bitmapv==NULL || index >= bitmap->valid_length)  return -1;
    if((v != 0) && (v != 1))   return -1;
    byte_index = index / 8;
    inner_byte_index = index % 8;
    v = v << (7 - inner_byte_index);
    bitmap->bitfield[byte_index] = bitmap->bitfield[byte_index] | v;

    return 0;
}
```

- int all_zero(Bitmap *bitmap)

功能：将位图所有位清 0

```
int all_zero(Bitmap *bitmap)
{
    if(bitmap->bitfield == NULL)  return -1;
    memset(bitmap->bitfield,0,bitmap->bitfield_length);
```

```
    return 0;
}
```

- int all_set(Bitmap *bitmap)

功能：将位图所有位放置 1

```
int all_set(Bitmap *bitmap)
{
    if(bitmap->bitfield == NULL)  return -1;
    memset(bitmap->bitfield,0xff,bitmap->bitfield_length);
    return 0;
}
```

- void release_memory_in_bitfield()

功能：释放本模块所申请的动态内存

```
void release_memory_in_bitfield()
{
    if(bitmap->bitfield != NULL) free(bitmap->bitfield);
    if(bitmap != NULL)  free(bitmap);
}
```

- int print_bitfield(Bitmap *bitmap)

功能：打印位图，用于调试程序

```
int print_bitfield(Bitmap *bitmap)
{
    int i;

    for(i = 0; i < bitmap->bitfield_length; i++) {
        printf("%.2X ",bitmap->bitfield[i]); //  以16进制的方式打印每个位图中的字节
        if( (i+1) % 16 == 0)  printf("\n");  //  每行打印  16个字节
    }
    printf("\n");

    return 0;
}
```

- int restore_bitmap()

功能：保存位图，用于断点续传

```
int restore_bitmap()
{
    int  fd;
    char bitmapfile[64];

    if( (bitmap == NULL) || (file_name == NULL) )  return -1;
```

```
        sprintf(bitmapfile,"%dbitmap",pieces_length);
        fd = open(bitmapfile,O_RDWR|O_CREAT|O_TRUNC,0666);
        if(fd < 0)  return -1;
        write(fd,bitmap->bitfield,bitmap->bitfield_length);
        close(fd);

        return 0;
    }
```

- int is_interested(Bitmap *dst,Bitmap *src)

功能：判断具有 src 位图的 peer 对具有 dst 位图的 peer 是否感兴趣

```
    int is_interested(Bitmap *dst,Bitmap *src)
    {
        unsigned char const_char[8] = { 0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
        unsigned char c1, c2;
        int             i, j;

        if( dst==NULL || src==NULL )  return -1;
        if( dst->bitfield==NULL || src->bitfield==NULL )  return -1;
          if( dst->bitfield_length!=src->bitfield_length || dst->valid_length!=src->
valid_length )
            return -1;
        // 如果 dst 中某位为 1 而 src 对应为 0，则说明 src 对 dst 感兴趣
        for(i = 0; i < dst->bitfield_length-1; i++) {
            for(j = 0; j < 8; j++) {  // 比较某个字节的所有位
                c1 = (dst->bitfield)[i] & const_char[j];  // 获取每一位的值
                c2 = (src->bitfield)[i] & const_char[j];
                if(c1>0 && c2==0)  return 1;
            }
        }

        j = dst->valid_length % 8;
        c1 = dst->bitfield[dst->bitfield_length-1];
        c2 = src->bitfield[src->bitfield_length-1];
        for(i = 0; i < j; i++) {  // 比较位图的最后一个字节
            if( (c1&const_char[i])>0 && (c2&const_char[i])==0 )
                return 1;
        }
        return 0;
    }
```

以上函数的功能正确性测试代码如下：

```
//  测试时可以交换  map1.bitfield 和  map2.bitfield 的值或赋于其他值
Bitmap map1, map2;
unsigned char bf1[2] = { 0xa0, 0xa0 };  //  位图每一位的值为   10100000 10100000
unsigned char bf2[2] = { 0xe0, 0xe0 };  //  位图每一位的值为   11100000 11100000

map1.bitfield        = bf1;
map1.bitfield_length  = 2;
map1.valid_length    = 11;
map2.bitfield        = bf2;
map2.bitfield_length  = 2;
map2.valid_length    = 11;

int ret = is_interested(&map1,&map2);
 printf("%d\n",ret);
```

在编写模块时，测试其中的每一个函数是很有必要的，否则无法知道模块中每一个函数是否达到预期的功能。限于篇幅，不能列出每个模块的测试代码。由于每个模块的相对独立性，读者不妨编写一些测试代码来测试某些模块的代码。

- int get_download_piece_num()

功能：获取当前已下载到的总 piece 数。函数实现代码如下：

```
int get_download_piece_num()
{
     unsigned char const_char[8] = { 0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
     int           i, j;

     if(bitmap==NULL || bitmap->bitfield==NULL)  return 0;
     download_piece_num = 0;

     for(i = 0; i < bitmap->bitfield_length-1; i++) {
         for(j = 0; j < 8; j++) {
             if( ((bitmap->bitfield)[i] & const_char[j]) != 0)
                  download_piece_num++;
         }
     }

     unsigned char c = (bitmap->bitfield)[i];  // c 存放位图最后一个字节的值
     j = bitmap->valid_length % 8;         // j 是位图最后一个字节的有效位数
     for(i = 0; i < j; i++) {
         if( (c & const_char[i]) !=0 ) download_piece_num++;
```

```
        }
        return download_piece_num;
    }
```

13.4.3 出错处理模块的设计和实现

该模块由 bterror.h 和 bterror.c 文件构成，主要定义了一些错误类型，以及发生导致程序终止的致命性错误时程序的响应。

```
bterror.h
#ifndef BTERROR_H
#define BTERROR_H

#define FILE_FD_ERR                         -1      //  无效的文件描述符
#define FILE_READ_ERR                       -2      //  读文件失败
#define FILE_WRITE_ERR                      -3      //  写文件失败
#define INVALID_METAFILE_ERR        -4      //  无效的种子文件
#define INVALID_SOCKET_ERR          -5      //  无效的套接字
#define INVALID_TRACKER_URL_ERR     -6      //  无效的  Tracker URL
#define INVALID_TRACKER_REPLY_ERR -7    //  无效的  Tracker 回应
#define INVALID_HASH_ERR            -8      //  无效的  hash值
#define INVALID_MESSAGE_ERR         -9      //  无效的消息
#define INVALID_PARAMETER_ERR       -10     //  无效的函数参数
#define FAILED_ALLOCATE_MEM_ERR     -11     //  申请动态内存失败
#define NO_BUFFER_ERR                       -12     //  没有足够的缓冲区
#define READ_SOCKET_ERR             -13     //  读套接字失败
#define WRITE_SOCKET_ERR            -14     //  写套接字失败
#define RECEIVE_EXIT_SIGNAL_ERR     -15     //  接收到退出程序的信号

//  用于提示致命性的错误，程序将终止
void btexit(int errno,char *file,int line);

#endif
```

以下是 bterror.c 文件：

```
bterror.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "bterror.h"

void btexit(i nt errno,char *file,int line)
```

```
{
     printf("exit at %s : %d with error number : %d\n",file, line, errno);
    exit(errno);
}
```

13.4.4 运行日志模块的设计和实现

本模块负责记录程序运行的日志，以备查询和分析程序行为，由log.h和log.c两个文件构成。

```
log.h
#ifndef  LOG_H
#define  LOG_H
 #include <stdarg.h>

 //  用于记录程序的行为
void logcmd(char *fmt,...);

 //  打开日志文件
int init_logfile(char *filename);

 //  将程序运行日志记录到文件
int logfile(char *file,int line,char *msg);

#endif
```

以下是log.c文件：

```
bterror.c
#include <stdio.h>
 #include <unistd.h>
 #include <stdlib.h>
 #include <string.h>
 #include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include "log.h"
//  日志文件的描述符
int logfile_fd = -1;
//  在命令行上打印一条日志
void logcmd(char *fmt,...)
{
     va_list ap;

     va_start(ap,fmt);
    vprintf(fmt,ap);
     va_end(ap);
}
```

```
//  打开记录日志的文件
int init_logfile(char *filename)
{
    logfile_fd = open(filename,O_RDWR|O_CREAT|O_APPEND,0666);
    if(logfile_fd < 0) {
        printf("open logfile failed\n");
        return -1;
    }
    return 0;
}
//  将一条日志写入日志文件
int logfile(char *file,int line,char *msg)
{
    char buff[256];

    if(logfile_fd < 0)  return -1;
    snprintf(buff,256,"%s:%d %s\n",file,line,msg);
    write(logfile_fd,buff,strlen(buff));
    return 0;
}
```

程序说明。

函数logcmd是一个变长参数的函数，也就是函数的参数个数是可变的，类似于 printf 函数。语句“logcmd(“%s:%d error\n”,__FILE__, __LINE__);”的功能与“printf(“%s:%d error\n”,__FILE__, __LINE__);”功能相同。

13.4.5 信号处理模块的设计和实现

在运行过程中，程序可能会接收到一些信号，如 SIGINT 、SIGTERM，这些信号的默认动作是立即终止程序。在信号处理模块，定义处理这些信号的函数。当信号产生时，系统自动调用相应的信号处理函数以便执行一些善后操作，如释放动态申请的内存、关闭文件描述符、关闭套接字。本模块由 signal_hander.h 和 signal_hander.c 两个文件构成。

```
signal_hander.h
#ifndef SIGNAL_HANDER_H
#define SIGNAL_HANDER_H

//  做一些清理工作 ，如释放动态分配的内存
void do_clear_work();
//  处理一些信号
void process_signal(int signo);
//  设置信号处理函数
```

```
int set_signal_hander();


#endif
```

以下是 signal_hander.c 文件：

```
signal_hander.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include "parse_metafile.h"
#include "bitfield.h"
#include "peer.h"
#include "data.h"
#include "tracker.h"
#include "torrent.h"
#include "signal_hander.h"

extern int  download_piece_num;
extern int  *fds;
extern int  fds_len;
extern Peer *peer_head;

// 程序将退出时，执行一些清理工作
void do_clear_work()
{
    // 关闭所有 peer的 socket
    Peer *p = peer_head;
    while(p != NULL) {
        if(p->state != CLOSING)  close(p->socket);
        p = p->next;
    }
    // 保存位图
    if(download_piece_num > 0) {
        restore_bitmap();
    }
    // 关闭文件描述符
    int i;
    for(i = 0; i < fds_len; i++) {
        close(fds[i]);
```

```
    }
    //  释放动态分配的内存
    release_memory_in_parse_metafile();
    release_memory_in_bitfield();
    release_memory_in_btcache();
    release_memory_in_peer();
    release_memory_in_torrent();

    exit(0);
}

void process_signal(int signo)
{
    printf("Please wait for clear operations\n");
    do_clear_work();
}

// 设置信号处理函数
int set_signal_hander()
{
    if(signal(SIGPIPE,SIG_IGN) == SIG_ERR) {
        perror("can not catch signal:sigpipe\n");
        return -1;
    }

    if(signal(SIGINT,process_signal) == SIG_ERR) {
        perror("can not catch signal:sigint\n");
        return -1;
    }

    if(signal(SIGTERM,process_signal) == SIG_ERR) {
        perror("can not catch signal:sigterm\n");
        return -1;
    }

    return 0;
}
```

13.4.6 Peer 管理模块的设计和实现

系统为每一个与之建立 TCP 连接的 Peer 构造一个 Peer 结构体。Peer 管理模块负责管理由各个 Peer 结点构成的 Peer 链表，主要工作是创建结点，添加结点到 Peer 链表，从 Peer 链表中删除结点等。

```
peer.h
#ifndef PEER_H
#define PEER_H
 #include <string.h>
 #include <time.h>
 #include "bitfield.h"

#define  INITIAL              -1       //  表明处于初始化状态
#define  HALFSHAKED      0    //  表明处于半握手状态
#define  HANDSHAKED      1    //  表明处于全握手状态
#define  SENDBITFIELD         2    //  表明处于已发送位图状态
#define  RECVBITFIELD         3    //  表明处于已接收位图状态
#define  DATA                 4    //  表明处于与peer交换数据的状态
#define  CLOSING              5    //  表明处于即将与 peer断开的状态

 //  发送和接收缓冲区的大小,16K可以存放一个 slice,2K用来存放其他消息
 #define  MSG_SIZE    （ 2*1024+16*1024）

typedef struct _Request_piece {
    int     index;                    //  请求的  piece 的索引
    int     begin;                    //  请求的  piece 的偏移
    int     length;                   //  请求的长度，一般为  16KB
     struct _Request_piece *next;
 } Request_piece;  //   定义数据请求队列的结点

 typedef struct  _Peer {
    int            socket;                  //  通过该  socket与 peer进行通信
    char           ip[16];                  // peer 的 ip地址
    unsigned short   port;                       // peer 的端口号
    char           id[21];                  // peer 的 id

    int            state;                       //  当前所处的状态
    int            am_choking;              //  是否将  peer阻塞
    int            am_interested;               //  是否对  peer感兴趣
```

```
    int             peer_choking;                   //  是否被  peer阻塞
    int             peer_interested;            //  是否被  peer感兴趣

    Bitmap          bitmap;                     //  存放 peer的位图

    char           *in_buff;                    //  存放从  peer处获取的消息
    int             buff_len;                   //  缓存区   in_buff 的长度
    char           *out_msg;                    //  存放将发送给 peer的消息
    int             msg_len;                    //  缓冲区   out_msg 的长度
    char           *out_msg_copy;               // out_msg的副本  , 发送时使用该缓冲区
    int             msg_copy_len;               //  缓冲区  out_msg_copy的长度
    int             msg_copy_index;             //  下一次要发送的数据的偏移量

    Request_piece  *Request_piece_head;             //  向 peer请求数据的队列
    Request_piece  *Requested_piece_head;           //  被 peer请求数据的队列

    unsigned int    down_total;                     //  从该 peer下载的总字节数
    unsigned int    up_total;                       //  向该 peer上传的总字节数

    time_t          start_timestamp;                //  最近一次接收到 peer消息的时间
    time_t          recet_timestamp;                //  最近一次发送消息给   peer的时间
    time_t          last_down_timestamp;            //  最近下载数据的开始时间
    time_t          last_up_timestamp;              //  最近上传数据的开始时间
    long long       down_count;         //  本计时周期从 peer下载的数据的字节数
    long long       up_count;                   //  本计时周期向 peer上传的数据的字节数
    float           down_rate;                  //  本计时周期从 peer处下载数据的速度
    float           up_rate;                        //  本计时周期向 peer处上传数据的速度

     struct _Peer   *next;
} Peer;

int   initialize_peer(Peer *peer);          //  对 peer各个成员进行初始化
Peer* add_peer_node();                          //  添加一个   peer结点
int   del_peer_node(Peer *peer);                //  删除一个   peer结点
void  free_peer_node(Peer *node);               //  释放一个   peer结点的内存
int   cancel_request_list(Peer *node);          //  撤消当前请求队列
int   cancel_requested_list(Peer *node);        //  撤消当前被请求队列
void  release_memory_in_peer();                 //  释放 peer.c中的动态分配的内存
void  print_peers_data();                       //  打印 peer链表中某些成员的值   , 用
```

```
于调试


    #endif
```

程序说明。

（1）Peer结构体是整个程序最重要的数据结构，也是最复杂的数据结构。peer.h 中首先定义了 7 种状态。各个状态的转换图如图13-3所示。

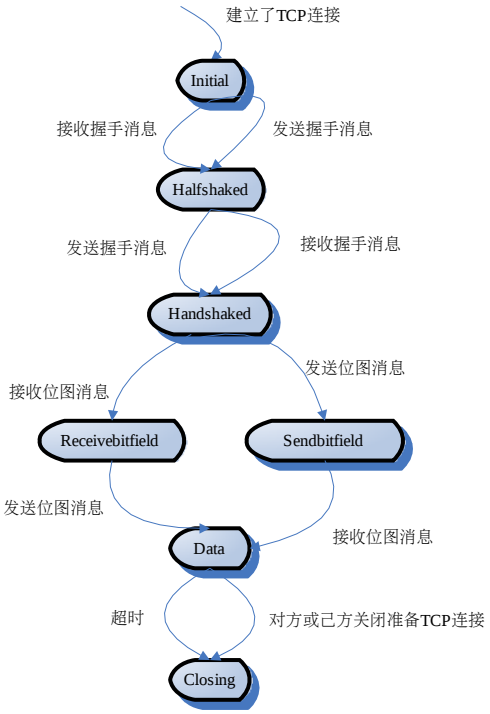

图 13-3 状态转换图

Halfshaked （半握手状态）是指已发送握手消息但未收到对方的握手消息，或者已经接收到对方的握手消息，但己方未发送握手消息。处于Data状态时双方可以交换数据，此时 Peer 结构体中的 am_choking 、 am_interested 、 peer_choking 、 peer_interested4 个成员变量有效。

如图 13-4 所示，当 am_interested = 1， peer_choking = 0 时，也就是客户端对 peer 感兴趣，而且 peer 没有将客户端阻塞，此时可以发送数据请求，即发送 request 消息请求 peer 发送数据，peer 接收到请求后发送 piece 消息，数据就被封装在 piece 消息中。

如图 13-5 所示，当 peer_interested = 1， am_choking = 0 时也就是 peer 对客户端感兴趣，而且我们没有将该 peer 阻塞，此时如果 peer 发送 request 消息请求数据，则应该构造并发送 piece 消息，其中数据被封装在 piece 消息中。

图13-5中“发送 unchoke 消息”的时机是执行选择非阻塞 peer 算法时，选中该 peer 作为非阻塞 peer 或者选中该 peer 作为优化非阻塞 peer 。“发送 choke 消息”的时机类似。

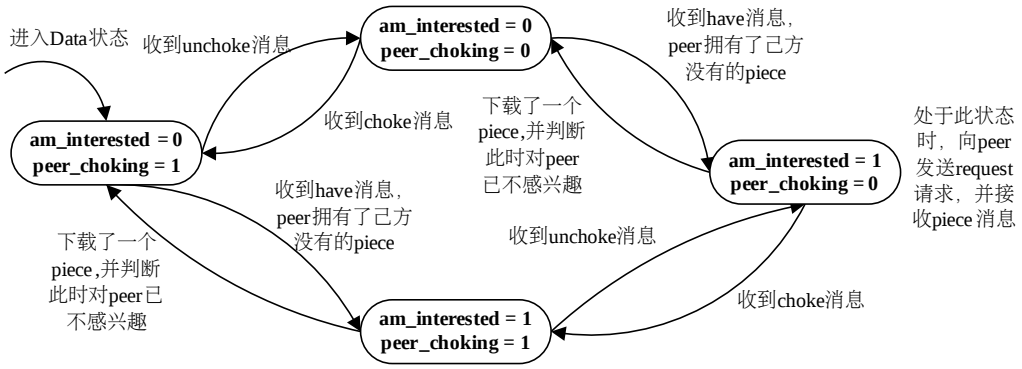

图 13-4 从下载数据的角度看到的处于 Data 状态时的内部状态

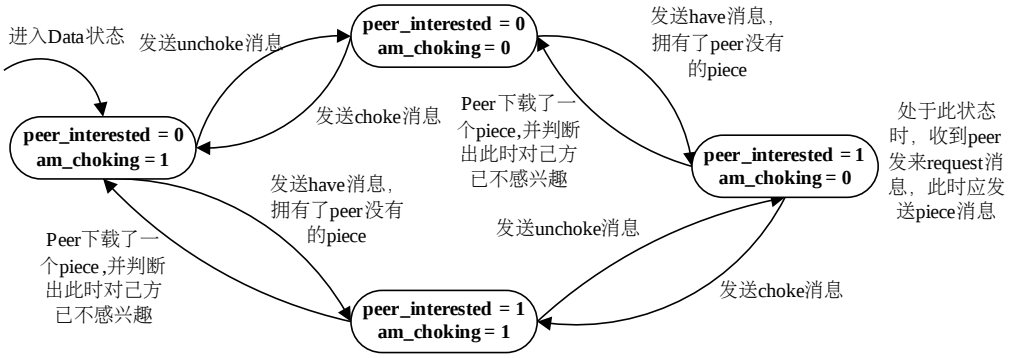

图 13-5 从上传数据的角度看到的处于 Data 状态时的内部状态

“发送 have 消息，拥有了 peer 没有的 piece” 的含义是己方刚刚下载到一个 piece ，此时通过发送 have 消息告知所有 peer 客户端已拥有了某个 piece ，如果 peer 没有这个 piece 且原先 peer 对本客户端不感兴趣，则发送 have 消息后，该 peer 就对该客户端感兴趣了。

（2）Peer 结构体中定义两个发送缓冲区 out_msg 和 out_msg_copy，将刚刚生成的消息暂存在 out_msg 中，调用套接字函数 send 向 peer 发送消息时使用 out_msg_copy 缓冲区。out_msg_copy 缓冲区的大小是 18KB ，而 send 函数一次最多发送 1500字节，因此要使用 msg_copy_index 记录下次应该发送的数据的起始下标。事实上， send 函数一次也可以发送超过 1500字节的数据，不过若以这种方式发送会导致发送数据混乱，具体原因后面将会解释。其变量的含义请参考消息处理模块。

peer.c 文件的头部包含的代码如下：

```
peer.c
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include "peer.h"
#include "message.h"
#include "bitfield.h"

extern Bitmap *bitmap;

// 指向当前与之进行通信的 peer链表
Peer *peer_head = NULL;
```

peer.c 中各个函数的定义如下。

initialize_peer(Peer *peer)

功能：初始化 Peer 结构体

```
int  initialize_peer(Peer *peer)
{
```

```
if(peer == NULL)   return -1;
peer->socket = -1;
memset(peer->ip,0,16);
peer->port = 0;
memset(peer->id,0,21);
peer->state = INITIAL;

peer->in_buff = NULL;
peer->out_msg = NULL;
peer->out_msg_copy = NULL;

peer->in_buff = (char *)malloc(MSG_SIZE);
if(peer->in_buff == NULL)  goto OUT;
memset(peer->in_buff,0,MSG_SIZE);
peer->buff_len = 0;

peer->out_msg = (char *)malloc(MSG_SIZE);
if(peer->out_msg == NULL)  goto OUT;
memset(peer->out_msg,0,MSG_SIZE);
peer->msg_len  = 0;

peer->out_msg_copy = (char *)malloc(MSG_SIZE);
if(peer->out_msg_copy == NULL)  goto OUT;
memset(peer->out_msg_copy,0,MSG_SIZE);
peer->msg_copy_len   = 0;
peer->msg_copy_index = 0;

peer->am_choking = 1;
peer->am_interested = 0;
peer->peer_choking = 1;
peer->peer_interested = 0;

peer->bitmap.bitfield = NULL;
peer->bitmap.bitfield_length = 0;
peer->bitmap.valid_length = 0;

peer->Request_piece_head = NULL;
peer->Requested_piece_head = NULL;
```

```
        peer->down_total = 0;
        peer->up_total = 0;

        peer->start_timestamp = 0;
        peer->recet_timestamp = 0;

        peer->last_down_timestamp = 0;
        peer->last_up_timestamp = 0;
        peer->down_count = 0;
        peer->up_count = 0;
        peer->down_rate = 0.0;
        peer->up_rate = 0.0;

        peer->next = (Peer *)0;
        return 0;
    OUT:
        if(peer->in_buff != NULL)              free(peer->in_buff);
        if(peer->out_msg != NULL)              free(peer->out_msg);
        if(peer->out_msg_copy != NULL)         free(peer->out_msg_copy);
        return -1;
    }
```

- Peer* add_peer_node()

功能：向peer链表添加一个结点

```
Peer* add_peer_node()
{
    int  ret;
    Peer *node, *p;

    //  分配内存空间
    node = (Peer *)malloc(sizeof(Peer));
    if(node == NULL)  {
        printf("%s:%d error\n",__FILE__,__LINE__);
        return NULL;
    }
    //  进行初始化
    ret = initialize_peer(node);
    if(ret < 0) {
        printf("%s:%d error\n",__FILE__,__LINE__);
        free(node);
```

```
            return NULL;
        }
        //  将 node加入到  peer链表中
        if(peer_head == NULL)  { peer_head = node; }  // node 为 peer链表的第一个结点
        else {
            p = peer_head;   //  使 p 指针指向  peer链表的最后一个结点
            while(p->next != NULL)  p = p->next;
            p->next = node;
        }
        return node;
    }
```

- int del_peer_node(Peer *peer)

功能：从 peer 链表中删除一个结点

```
    int del_peer_node(Peer *peer)
    {
        Peer *p = peer_head, *q;
        if(peer == NULL)  return -1;

        while(p != NULL) {
            if( p == peer ) {
                if(p == peer_head)  peer_head = p->next;
                else  q->next = p->next;
                free_peer_node(p);
                return 0;
            } else {
                q = p;
                p = p->next;
            }
        }

        return -1;
    }
```

- int cancel_request_list(Peer *node)

功能：//撤销当前请求队列

```
    int cancel_request_list(Peer *node)
    {
    Request_piece  *p = node->Request_piece_head;

        while(p != NULL) {
```

```
            node->Request_piece_head = node->Request_piece_head->next;
            free(p);
            p = node->Request_piece_head;
        }

        return 0;
    }
```

- int cancel_requested_list(Peer *node)

功能：//撤销当前被请求队列

```
    int cancel_requested_list(Peer *node)
    {
     Request_piece  * p = node->Requested_piece_head;

        while(p != NULL) {
            node->Requested_piece_head = node->Requested_piece_head->next;
            free(p);
            p = node->Requested_piece_head;
        }

        return 0;
    }
```

- void free_peer_node(Peer *node)

功能：释放一个peer结点的内存

```
    void  free_peer_node(Peer *node)
    {
        if(node == NULL)  return;
        if(node->bitmap.bitfield != NULL) free(node->bitmap.bitfield);
        if(node->in_buff != NULL)      free(node->in_buff);
         if(node->out_msg != NULL) free(node->out_msg);
        if(node->out_msg_copy != NULL) free(node->out_msg_copy);
        //  撤销请求队列和被请求队列
        cancel_request_list(node);
        cancel_requested_list(node);
        //  释放完  peer成员的内存后 ，再释放  peer所占的内存
        free(node);
    }
```

- void release_memory_in_peer()

功能：释放peer管理模块中动态申请的内存

```
    void  release_memory_in_peer()
```

```
{
Peer *p;

    if(peer_head == NULL)  return;
    p = peer_head;
    while(p != NULL) {
        peer_head = peer_head->next;
        free_peer_node(p);
        p = peer_head;
    }
}
```

- void print_peers_data()

功能：打印 peer 结点的一些信息，用于调试程序

```
void print_peers_data()
{
    Peer *p = peer_head;
    int  index = 0;

    while(p != NULL) {
            printf("peer:  %d   IP  %s   down_rate:  %.2f  \n",  index,  p->ip,  p-
>down_rate);
        index++;
        p = p->next;
    }
}
```

13.4.7 消息处理模块的设计和实现

消息处理模块负责根据当前的状态生成并发送消息，接收以及处理消息。消息处理模块由messag.h 和message.c两个文件构成，理解和分析本模块的代码时请参考图 13-3 、图 13-4 和图 13-5 。

```
message.h
#ifndef MESSAGE_H
#define MESSAGE_H
#include "peer.h"

int int_to_char(int i, unsigned char c[4]);      //  将整型变量 i 的 4 个字节存放到数组 c 中
int char_to_int(unsigned char c[4]);                 //  将数组  c 中的  4 个字节转换为一个整型数

//  以下函数创建各个类型的消息，创建消息的函数请参考  BT协议加以理解
int create_handshake_msg(char *info_hash,char *peer_id,Peer *peer);
int create_keep_alive_msg(Peer *peer);
int create_chock_interested_msg(int type,Peer *peer);
int create_have_msg(int index,Peer *peer);
int create_bitfield_msg(char *bitfield,int bitfield_len,Peer *peer);
int create_request_msg(int index,int begin,int length,Peer *peer);
int create_piece_msg(int index,int begin,char *block,int b_len,Peer *peer);
int create_cancel_msg(int index,int begin,int length,Peer *peer);
int create_port_msg(int port,Peer *peer);

//  判断接收缓冲区中是否存放了一条完整的消息
int is_complete_message(unsigned char *buff,unsigned int len,int *ok_len);
//  处理收到的消息 ，接收缓冲区中存放着一条完整的消息
int parse_response(Peer *peer);
//  处理收到的消息 ，接收缓冲区中除了存放着一条完整的消息外 ，还有其他不完整的消息
int parse_response_uncomplete_msg(Peer *p,int ok_len);
//  根据当前的状态创建响应消息
int create_response_message(Peer *peer);
//  为发送 have消息作准备 ,have 消息较为特殊 ，它要发送给所有 peer
int prepare_send_have_msg();
//  即将与  peer断开时  ，丢弃套接字发送缓冲区中的所有未发送的消息
void discard_send_buffer(Peer *peer);
```

```
#endif
```

message.c 文件的头部包括的代码如下：

```
message.h
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <sys/socket.h>
#include "parse_metafile.h"
#include "bitfield.h"
#include "peer.h"
#include "policy.h"
#include "data.h"
#include "message.h"

#define HANDSHAKE        -2   //  握手消息
#define KEEP_ALIVE       -1   // keep_alive 消息
#define CHOKE            0    // choke消息
#define UNCHOKE          1    // unchoke消息
#define INTERESTED       2    // interested 消息
#define UNINTERESTED     3    // uninterested 消息
#define HAVE             4    // have 消息
#define BITFIELD         5    // bitfield 消息
#define REQUEST          6    // request消息
#define PIECE            7    // piece消息
#define CANCEL           8    // cancel 消息
#define PORT             9    // port 消息

//  如果 45秒未给某  peer发送消息，则发送  keep_alive消息
#define KEEP_ALIVE_TIME 45

extern Bitmap    *bitmap;             //  在 bitmap.c中定义，指向己方的位图
extern char       info_hash[20];      //  在 parse_metafile.c 中定义，存放 info_hash
extern char       peer_id[20];        //  在 parse_metafile.c 中定义，存放 peer_id
extern int        have_piece_index[64];      //  在 data.c中定义，存放下载到的 piece 的
index
```

```
extern Peer      *peer_head;          //  在peer.c中定义，指向peer链表
```

message.c 中各个函数的定义如下：

- int int_to_char(int i, unsigned char c[4])

功能：获取 i 的各个字节，并保存到字符数组 c 中

```
int int_to_char(int i, unsigned char c[4])
{
    c[3] = i%256;
    c[2] = (i-c[3])/256%256;
    c[1] = (i-c[3]-c[2]*256)/256/256%256;
    c[0] = (i-c[3]-c[2]*256-c[1]*256*256)/256/256/256%256;

    return 0;
}
```

程序说明。

假设 i = 123456789，若以16进制表示该数为 0x75BCD15，则 c[0] = 07， c[1] = 5B， c[2] = CD， c[3] = 15 。函数 char_to_int 的功能与本函数刚好相反。

- int char_to_int(unsigned char c[4])

功能：将字符数组中的字符转换为一个整型

```
int char_to_int(unsigned char c[4])
{
    int i;

    i = c[0]*256*256*256 + c[1]*256*256 + c[2]*256 + c[3];

    return i;
}
```

- int create_handshake_msg(char *info_hash,char *peer_id,Peer *peer)

功能：创建握手消息

参数：info_hash在 parse_metafile.c 中由种子文件计算而得；peer_id也在 parse_metafile.c 中生成；peer 为要发送握手消息给某一个 peer 的指针变量。

返回：创建消息成功返回 0，创建失败返回-1。函数实现代码如下：

```
int create_handshake_msg(char *info_hash,char *peer_id,Peer *peer)
{
    int           i;
    unsigned char  keyword[20] = "BitTorrent protocol", c = 0x00;
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int           len = MSG_SIZE - peer->msg_len;

    if(len < 68)  return -1;  // 握手消息的长度固定为68字节
```

```
        buffer[0] = 19;
        for(i = 0; i < 19; i++)       buffer[i+1]  = keyword[i];
        for(i = 0; i < 8;  i++)       buffer[i+20] = c;
        for(i = 0; i < 20; i++)       buffer[i+28] = info_hash[i];
        for(i = 0; i < 20; i++)       buffer[i+48] = peer_id[i];

        peer->msg_len += 68;
        return 0;
    }
```

程序说明。

将生成的握手消息存放在 peer 结点的发送缓冲区中（即 msg_out），其中 msg_out[0] ～ msg_out[msg_len-1] 已存放了其他消息。函数中变量 len 指明缓冲区还有多少空闲。初始情况下，msg_len 的值应为 0 。

- int create_keep_alive_msg(Peer *peer)

功能：创建 keep_alive 消息

```
int create_keep_alive_msg(Peer *peer)
{
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int            len = MSG_SIZE - peer->msg_len;

    if(len < 4)  return -1;  // keep_alive 消息的长度固定为  4
    memset(buffer,0,4);
    peer->msg_len += 4;
    return 0;
}
```

- int create_chock_interested_msg(int type,Peer *peer)

功能：创建 chock 消息

```
int create_chock_interested_msg(int type,Peer *peer)
{
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int            len = MSG_SIZE - peer->msg_len;

    // choke、 unchoke 、interested、uninterested消息的长度固定为  5
    if(len < 5)  return -1;
    memset(buffer,0,5);
    buffer[3] = 1;
    buffer[4] = type;
```

```
        peer->msg_len += 5;
        return 0;
    }
```

程序说明。

该函数可创建 4 种消息，即 choke 、 unchoke 、interested、uninterested消息。 choke 消息的 type 值为 0， unchoke 消息为 1， interested消息为 2， uninterested消息为 3 。

- int create_have_msg(int index,Peer *peer)

功能：创建have消息

```
int create_have_msg(int index,Peer *peer)
{
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int            len = MSG_SIZE - peer->msg_len;
    unsigned char  c[4];

    if(len < 9)  return -1;      // have 消息的长度固定为  9
    memset(buffer,0,9);
    buffer[3] = 5;
    buffer[4] = 4;
    int_to_char(index,c);       // index 为 piece 的下标
    buffer[5] = c[0];
    buffer[6] = c[1];
    buffer[7] = c[2];
    buffer[8] = c[3];

    peer->msg_len += 9;
    return 0;
}
```

- int create_bitfield_msg(char *bitfield,int bitfield_len,Peer *peer)

功能：创建bitfield消息

```
int create_bitfield_msg(char *bitfield,int bitfield_len,Peer *peer)
{
    int            i;
    unsigned char  c[4];
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
    int            len = MSG_SIZE - peer->msg_len;

    if( len < bitfield_len+5 )  {  // bitfield 消息的长度为 bitfield_len+5
        printf("%s:%d buffer too small\n",__FILE__,__LINE__);
        return -1;
```

```
        }
        int_to_char(bitfield_len+1,c); //  位图消息的负载长度为位图长度加 1
        for(i = 0; i < 4; i++)  buffer[i] = c[i];
        buffer[4] = 5;
        for(i = 0; i < bitfield_len; i++) buffer[i+5] = bitfield[i];

        peer->msg_len += bitfield_len+5;
        return 0;
    }
```

- int create_request_msg(int index,int begin,int length,Peer *peer)

功能：创建数据请求消息。

参数： index 为请求的 piece 的下标； begin 为 piece 内的偏移量； length 为请求数据的长度。函数实现的代码如下：

```
    int create_request_msg(int index,int begin,int length,Peer *peer)
    {
        int            i;
        unsigned char  c[4];
        unsigned char  *buffer = peer->out_msg + peer->msg_len;
        int            len = MSG_SIZE - peer->msg_len;

        if(len < 17)  return -1;  // request 消息的长度固定为 17
        memset(buffer,0,17);
        buffer[3] = 13;
        buffer[4] = 6;
        int_to_char(index,c);
        for(i = 0; i < 4; i++)  buffer[i+5]  = c[i];
        int_to_char(begin,c);
        for(i = 0; i < 4; i++)  buffer[i+9]  = c[i];
        int_to_char(length,c);
        for(i = 0; i < 4; i++)  buffer[i+13] = c[i];

        peer->msg_len += 17;
        return 0;
    }
```

- int create_piece_msg(int index,int begin,char *block,int b_len,Peer *peer)

功能：创建 piece 消息。

参数： block 指向待发送的数据； b_len 为 block 所指向的数据的长度。函数实现的代码如下：

```
    int create_piece_msg(int index,int begin,char *block,int b_len,Peer *peer)
```

```
{
     int            i;
    unsigned char  c[4];
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
     int            len = MSG_SIZE - peer->msg_len;

    if( len < b_len+13 ) {  // piece 消息的长度为 b_len+13
        printf("%s:%d buffer too small\n",__FILE__,__LINE__);
        return -1;
    }

     int_to_char(b_len+9,c);
    for(i = 0; i < 4; i++)      buffer[i] = c[i];
    buffer[4] = 7;
     int_to_char(index,c);
    for(i = 0; i < 4; i++)      buffer[i+5] = c[i];
     int_to_char(begin,c);
    for(i = 0; i < 4; i++)      buffer[i+9] = c[i];
    for(i = 0; i < b_len; i++)  buffer[i+13] = block[i];

    peer->msg_len += b_len+13;
    return 0;
}
```

- int create_cancel_msg(int index,int begin,int length,Peer *peer)

功能：创建 cancel 消息

```
int create_cancel_msg(int index,int begin,int length,Peer *peer)
{
     int            i;
    unsigned char  c[4];
    unsigned char  *buffer = peer->out_msg + peer->msg_len;
     int            len = MSG_SIZE - peer->msg_len;

    if(len < 17)  return -1;  // cancel消息的长度固定为 17
    memset(buffer,0,17);
     buffer[3] = 13;
    buffer[4] = 8;
     int_to_char(index,c);
     for(i = 0; i < 4; i++)  buffer[i+5]  = c[i];
     int_to_char(begin,c);
```

```
        for(i = 0; i < 4; i++)  buffer[i+9]  = c[i];
        int_to_char(length,c);
        for(i = 0; i < 4; i++)  buffer[i+13] = c[i];

        peer->msg_len += 17;
        return 0;
    }
```

- int create_port_msg(int port,Peer *peer)

功能：创建 port 消息

附注：实际上程序从未发送过本消息，因为根据 BT 协议，该消息是为那些以 DHT 的方式获取 peer 地址的应用程序所准备的。函数实现的代码如下：

```
    int create_port_msg(int port,Peer *peer)
    {
        unsigned char   c[4];
        unsigned char  *buffer = peer->out_msg + peer->msg_len;
        int            len = MSG_SIZE - peer->msg_len;

        if( len < 7)  return 0;  // port 消息的长度固定为  7
        memset(buffer,0,7);
        buffer[3] = 3;
        buffer[4] = 9;
        int_to_char(port,c);
        buffer[5] = c[2];
        buffer[6] = c[3];

        peer->msg_len += 7;
        return 0;
    }
```

- int is_complete_message(unsigned char *buff,unsigned int len,int *ok_len)

功能：判断缓冲区中是否存放了一条完整的消息。

参数： buff 指向存放消息的缓冲区； len 为缓冲区 buff 的长度； ok_len 用于返回缓冲区中完整消息的长度，即 buff[0]～buff[ok_len-1] 存放着一条或多条完整的消息。函数实现的完整代码如下：

```
    int is_complete_message(unsigned char *buff,unsigned int len,int *ok_len)
    {
        unsigned int    i;
        char          btkeyword[20];
        unsigned char  keep_alive[4]      = { 0x0, 0x0, 0x0, 0x0 };
        unsigned char  chocke[5]         = { 0x0, 0x0, 0x0, 0x1, 0x0};
```

```
unsigned char  unchocke[5]       = { 0x0, 0x0, 0x0, 0x1, 0x1};
 unsigned char  interested[5]          = { 0x0, 0x0, 0x0, 0x1, 0x2};
 unsigned char  uninterested[5]      = { 0x0, 0x0, 0x0, 0x1, 0x3};
 unsigned char  have[5]            = { 0x0, 0x0, 0x0, 0x5, 0x4};
 unsigned char  request[5]         = { 0x0, 0x0, 0x0, 0xd, 0x6};
 unsigned char  cancel[5]          = { 0x0, 0x0, 0x0, 0xd, 0x8};
 unsigned char  port[5]            = { 0x0, 0x0, 0x0, 0x3, 0x9};

if(buff==NULL || len<=0 || ok_len==NULL)  return -1;
*ok_len = 0;
btkeyword[0] = 19;
memcpy(&btkeyword[1],"BitTorrent protocol",19);  // BitTorrent协议关键字

 unsigned char  c[4];
 unsigned int   length;
 for(i = 0; i < len; ) {
     //  握手、 chocke、have等消息的长度是固定的
     if( i+68<=len && memcmp(&buff[i],btkeyword,20)==0 )         i += 68;
      else if( i+4 <=len && memcmp(&buff[i],keep_alive,4)==0 )      i += 4;
      else if( i+5 <=len && memcmp(&buff[i],chocke,5)==0 )          i += 5;
      else if( i+5 <=len && memcmp(&buff[i],unchocke,5)==0 )        i += 5;
      else if( i+5 <=len && memcmp(&buff[i],interested,5)==0 )        i += 5;
      else if( i+5 <=len && memcmp(&buff[i],uninterested,5)==0 )      i += 5;
      else if( i+9 <=len && memcmp(&buff[i],have,5)==0 )             i += 9;
     else if( i+17<=len && memcmp(&buff[i],request,5)==0 )         i += 17;
     else if( i+17<=len && memcmp(&buff[i],cancel,5)==0 )          i += 17;
     else if( i+7 <=len && memcmp(&buff[i],port,5)==0 )              i += 7;
    else if( i+5 <=len && buff[i+4]==5 )  {  // bitfield消息的长度是变化的
         c[0] = buff[i];   c[1] = buff[i+1];
         c[2] = buff[i+2]; c[3] = buff[i+3];
         length = char_to_int(c);
          //  消息长度占 4 字节 ，消息本身占length个字节
         if( i+4+length <= len )  i += 4+length;
          else { *ok_len = i; return -1; }
     }
     else if( i+5 <=len && buff[i+4]==7 )  {  // piece 消息的长度也是变化的
         c[0] = buff[i];   c[1] = buff[i+1];
         c[2] = buff[i+2]; c[3] = buff[i+3];
          length = char_to_int(c);
```

```
                // 消息长度占 4 字节 ，消息本身占length个字节
                if( i+4+length <= len )  i += 4+length;
                else { *ok_len = i; return -1; }
            }
            else {
            // 处理未知类型的消息
                if(i+4 <= len) {
                    c[0] = buff[i];   c[1] = buff[i+1];
                    c[2] = buff[i+2]; c[3] = buff[i+3];
                    length = char_to_int(c);
                    // 消息长度占 4 字节 ，消息本身占length个字节
                    if(i+4+length <= len)  { i += 4+length; continue; }
                    else { *ok_len = i; return -1; }
                }
            // 如果不是未知消息类型，则认为目前接收的数据还不是一个完整的消息
                *ok_len = i;
                return -1;
            }
        } // for语句结束

        *ok_len = i;
        return 1;
    }
```

- int process_handshake_msg(Peer *peer,unsigned char *buff,int len)

功能：处理接收到的一条握手消息。

参数：从 peer 接收到这条握手消息； buff 指向握手消息； len 为 buff 的长度。函数实现的代码如下：

```
    int process_handshake_msg(Peer *peer,unsigned char *buff,int len)
    {
        if(peer==NULL || buff==NULL) return -1;
        if(memcmp(info_hash,buff+28,20) != 0) {  //  若 info_hash 不一致则关闭连接
            peer->state = CLOSING;
            // 丢弃发送缓冲区中的数据
            discard_send_buffer(peer);
            clear_btcache_before_peer_close(peer);
            close(peer->socket);
            return -1;
        }
        // 保存该 peer的 peer_id
```

```
        memcpy(peer->id,buff+48,20);
        (peer->id)[20] = '\0';
        //  若当前处于 Initial 状态，则发送握手消息给 peer
        if(peer->state == INITIAL) {
            create_handshake_msg(info_hash,peer_id,peer);
           peer->state = HANDSHAKED;
        }
        //  若握手消息已发送，则状态转换为已握手状态
        if(peer->state == HALFSHAKED)  peer->state = HANDSHAKED;
        //  记录最近收到该 peer消息的时间
    //  若一定时间内 ( 如两分钟  ) 未收到来自该 peer的任何消息，则关闭连接
        peer->start_timestamp = time(NULL);
        return 0;
    }
```

- int process_keep_alive_msg(Peer *peer,unsigned char *buff,int len)

功能：处理刚刚接收到的来自peer的keepv_alive消息。函数实现的代码如下：

```
    int process_keep_alive_msg(Peer *peer,unsigned char *buff,int len)
    {
        if(peer==NULL || buff==NULL)  return -1;
        //  记录最近收到该 peer消息的时间
    //  若一定时间内 ( 如 2min) 未收到来自该 peer的任何消息，则关闭连接
        peer->start_timestamp = time(NULL);
        return 0;
    }
```

- int process_choke_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 choke 消息。函数实现的代码如下：

```
    int process_choke_msg(Peer *peer,unsigned char *buff,int len)
    {
        if(peer==NULL || buff==NULL)  return -1;
    //  若原先处于 unchoke 状态，收到该消息后更新 peer中某些变量的值
        if( peer->state!=CLOSING && peer->peer_choking==0 ) {
            peer->peer_choking = 1;
           peer->last_down_timestamp = 0;      //  将最近接收到来自该  peer数据的时间清零
            peer->down_count = 0;      //  将最近从该peer处下载的字节数清零
            peer->down_rate = 0;       //  将最近从该peer下载数据的速度清零
        }

        peer->start_timestamp = time(NULL);
        return 0;
```

```
    }
```

- int process_unchoke_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到 unchoke 消息。函数实现的代码如下：

```
    int process_unchoke_msg(Peer *peer,unsigned char *buff,int len)
    {
        if(peer==NULL || buff==NULL)  return -1;
        //  若原来处于 choke 状态且与该peer的连接未被关闭
        if( peer->state!=CLOSING && peer->peer_choking==1 ) {
            peer->peer_choking = 0;
            //  若对该 peer感兴趣，则构造 request 消息请求 peer发送数据
            if(peer->am_interested == 1)  create_req_slice_msg(peer);
            else {
                peer->am_interested = is_interested(&(peer->bitmap), bitmap);
                if(peer->am_interested == 1)  create_req_slice_msg(peer);
                else  printf("Received unchoke but Not interested to IP:%s \n",peer-
>ip);
            }
            //  更新一些成员的值
            peer->last_down_timestamp = 0;
            peer->down_count = 0;
            peer->down_rate = 0;
        }

        peer->start_timestamp = time(NULL);
        return 0;
    }
```

- int process_interested_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的interested消息。函数实现的代码如下：

```
    int process_interested_msg(Peer *peer,unsigned char *buff,int len)
    {
        if(peer==NULL || buff==NULL)  return -1;
        if( peer->state!=CLOSING && peer->state==DATA ) {
            peer->peer_interested = is_interested(bitmap, &(peer->bitmap));
            if(peer->peer_interested == 0)  return -1;
            if(peer->am_choking == 0) create_chock_interested_msg(1,peer);
        }

        peer->start_timestamp = time(NULL);
        return 0;
```

```
}
```

- int process_uninterested_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的uninterested消息。函数实现的代码如下：

```
int process_uninterested_msg(Peer *peer,unsigned char *buff,int len)
{
     if(peer==NULL || buff==NULL)  return -1;
    if( peer->state!=CLOSING && peer->state==DATA ) {
        peer->peer_interested = 0;
        cancel_requested_list(peer);
    }

    peer->start_timestamp = time(NULL);
     return 0;
}
```

- int process_have_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的have消息。函数实现的代码如下：

```
int process_have_msg(Peer *peer,unsigned char *buff,int len)
{
     int       rand_num;
    unsigned  char c[4];

     if(peer==NULL || buff==NULL)  return -1;
    srand(time(NULL));
    rand_num = rand() % 3;  // 生成一个  0 ～ 2 的随机数
    if( peer->state!=CLOSING && peer->state==DATA ) {
        c[0] = buff[5]; c[1] = buff[6];
         c[2] = buff[7]; c[3] = buff[8];
         //  更新该 peer的位图
         if(peer->bitmap.bitfield != NULL)
 set_bit_value(&(peer->bitmap),char_to_int(c),1);
         if(peer->am_interested == 0) {
             peer->am_interested = is_interested(&(peer->bitmap), bitmap);
             //  由原来的对peer不感兴趣变为感兴趣时，发送 interested消息
             if(peer->am_interested == 1) create_chock_interested_msg(2,peer);
         } else {  //  收到  3 个 have则发一个  interested消息
            if(rand_num == 0) create_chock_interested_msg(2,peer);
         }
    }
```

```
    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_cancel_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 cancel 消息。函数实现的代码如下：

```
int process_cancel_msg(Peer *peer,unsigned char *buff,int len)
{
    unsigned char  c[4];
    int           index, begin, length;

    if(peer==NULL || buff==NULL)  return -1;
    c[0] = buff[5];  c[1] = buff[6];
    c[2] = buff[7];  c[3] = buff[8];
    index = char_to_int(c);
    c[0] = buff[9];  c[1] = buff[10];
    c[2] = buff[11]; c[3] = buff[12];
    begin = char_to_int(c);
    c[0] = buff[13]; c[1] = buff[14];
    c[2] = buff[15]; c[3] = buff[16];
    length = char_to_int(c);
    //  在被请求队列中删除指定的请求
    Request_piece *p, *q;
    p = q = peer->Requested_piece_head;
    while(p != NULL) {
        if( p->index==index && p->begin==begin && p->length==length ) {
            if(p == peer->Requested_piece_head)
                peer->Requested_piece_head = p->next;
            else
                q->next = p->next;
            free(p);
            break;
        }
        q = p;
        p = p->next;
    }

    peer->start_timestamp = time(NULL);
    return 0;
}
```

- int process_bitfield_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的位图消息。函数实现的代码如下：

```
int process_bitfield_msg(Peer *peer,unsigned char *buff,int len)
{
     unsigned char c[4];

     if(peer==NULL || buff==NULL)  return -1;
    if(peer->state==HANDSHAKED || peer->state==SENDBITFIELD) {
        c[0] = buff[0];   c[1] = buff[1];
         c[2] = buff[2];   c[3] = buff[3];
        //  若原先已收到一个位图消息，则清空原来的位图
        if( peer->bitmap.bitfield != NULL ) {
             free(peer->bitmap.bitfield);
            peer->bitmap.bitfield = NULL;
        }
        peer->bitmap.valid_length = bitmap->valid_length;
        if(bitmap->bitfield_length != char_to_int(c)-1) {  // 若收到的一个错误位图
             peer->state = CLOSING;
             //  丢弃发送缓冲区中的数据
             discard_send_buffer(peer);
             clear_btcache_before_peer_close(peer);
             close(peer->socket);
             return -1;
        }
        //  生成该 peer的位图
        peer->bitmap.bitfield_length = char_to_int(c) - 1;
        peer->bitmap.bitfield = (unsigned char*)malloc(peer->bitmap.bitfield_length);
        memcpy(peer->bitmap.bitfield,&buff[5],peer->bitmap.bitfield_length);

        //  如果原状态为已握手  ，收到位图后应该向 peer发位图
        if(peer->state == HANDSHAKED) {
             create_bitfield_msg(bitmap->bitfield,bitmap->bitfield_length,peer);
             peer->state = DATA;
        }
        //  如果原状态为已发送位图，收到位图后可以进入 DATA状态准备交换数据
        if(peer->state == SENDBITFIELD) {
             peer->state = DATA;
        }
        //  根据位图判断 peer是否对本客户端感兴趣
```

```
            peer->peer_interested = is_interested(bitmap,&(peer->bitmap));
            //  判断对  peer是否感兴趣，若是则发送interested消息
            peer->am_interested = is_interested(&(peer->bitmap), bitmap);
            if(peer->am_interested == 1) create_chock_interested_msg(2,peer);
        }

        peer->start_timestamp = time(NULL);
        return 0;
    }
```

- int process_request_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的request消息。函数实现代码如下：

```
    int process_request_msg(Peer *peer,unsigned char *buff,int len)
    {
        unsigned char   c[4];
        int            index, begin, length;
        Request_piece  *request_piece, *p;

        if(peer==NULL || buff==NULL)  return -1;
        if(peer->am_choking==0 && peer->peer_interested==1) {
            c[0] = buff[5];  c[1] = buff[6];
            c[2] = buff[7];  c[3] = buff[8];
            index = char_to_int(c);
            c[0] = buff[9];  c[1] = buff[10];
            c[2] = buff[11]; c[3] = buff[12];
            begin = char_to_int(c);
            c[0] = buff[13]; c[1] = buff[14];
            c[2] = buff[15]; c[3] = buff[16];
            length = char_to_int(c);

            //  查看该请求是否已存在，若已存在   ，则不进行处理
            p = peer->Requested_piece_head;
            while(p != NULL) {
                if(p->index==index && p->begin==begin && p->length==length) {
                    break;
                }
                p = p->next;
            }
            if(p != NULL)  return 0;
```

```
            //  将请求加入到请求队列中
            request_piece = (Request_piece *)malloc(sizeof(Request_piece));
           if(request_piece == NULL)  {
                printf("%s:%d error",__FILE__,__LINE__);
                return 0;
            }
            request_piece->index  = index;
            request_piece->begin  = begin;
           request_piece->length = length;
            request_piece->next  = NULL;
           if( peer->Requested_piece_head == NULL )
                peer->Requested_piece_head = request_piece;
           else {
                p = peer->Requested_piece_head;
                while(p->next != NULL)  p = p->next;
                p->next = request_piece;
            }
            //  打印提示信息
    printf("***add a request FROM IP:%s index:%-6d begin:%-6x ***\n",
    peer->ip,index,begin);
        }

        peer->start_timestamp = time(NULL);
        return 0;
    }
```

- int process_piece_msg(Peer *peer,unsigned char *buff,int len)

功能：处理收到的 piece 消息。函数实现代码如下：

```
    int process_piece_msg(Peer *peer,unsigned char *buff,int len)
    {
        unsigned char   c[4];
        int            index, begin, length;
        Request_piece  *p;

        if(peer==NULL || buff==NULL)  return -1;
        if(peer->peer_choking==0) {
            c[0] = buff[0];    c[1] = buff[1];
            c[2] = buff[2];    c[3] = buff[3];
            length = char_to_int(c) - 9;
            c[0] = buff[5];    c[1] = buff[6];
```

```
            c[2] = buff[7];    c[3] = buff[8];
            index = char_to_int(c);
           c[0] = buff[9];    c[1] = buff[10];
           c[2] = buff[11];   c[3] = buff[12];
            begin = char_to_int(c);
           //  判断收到的 slice 是否是请求过的
           p = peer->Request_piece_head;
           while(p != NULL) {
                if(p->index==index && p->begin==begin && p->length==length)
                    break;
               p = p->next;
           }
            if(p == NULL) {printf("did not found matched request\n"); return -1;}
           //  开始记时，并累计收到数据的字节数
           if(peer->last_down_timestamp == 0)
               peer->last_down_timestamp = time(NULL);
          peer->down_count += length;
          peer->down_total += length;
           //  将收到的数据写入缓冲区中
           write_slice_to_btcache(index,begin,length,buff+13,length,peer);
           //  生成请求数据的消息，要求继续发送数据
          create_req_slice_msg(peer);
       }

      peer->start_timestamp = time(NULL);
       return 0;
  }
```

- int parse_response(Peer *peer)

功能：处理收到的消息（peer 的接收缓冲区中可能存放着多条消息）。函数实现代码如下：

```
  int parse_response(Peer *peer)
  {
       unsigned char   btkeyword[20];
       unsigned char   keep_alive[4] = { 0x0, 0x0, 0x0, 0x0 };
       int            index;
       unsigned char  *buff = peer->in_buff;  // in_buff 为接收缓冲区
       int            len = peer->buff_len;  // buff_len 为接收缓冲区中有效数据的长度

       if(buff==NULL || peer==NULL)  return -1;
      btkeyword[0] = 19;
```

```
        memcpy(&btkeyword[1],"BitTorrent protocol",19);  // BitTorrent协议关键字

        //  分别处理  12种消息
        for(index = 0; index < len; ) {
            if( (len-index >= 68) && (memcmp(&buff[index],btkeyword,20) == 0) ) {
                process_handshake_msg(peer,buff+index,68);
                index += 68;
            }
            else if( (len-index >= 4) && (memcmp(&buff[index],keep_alive,4) == 0)) {
                process_keep_alive_msg(peer,buff+index,4);
                index += 4;
            }
            else if( (len-index >= 5) && (buff[index+4] == CHOKE) ) {
                process_choke_msg(peer,buff+index,5);
                index += 5;
            }
            else if( (len-index >= 5) && (buff[index+4] == UNCHOKE) ) {
                process_unchoke_msg(peer,buff+index,5);
                index += 5;
            }
            else if( (len-index >= 5) && (buff[index+4] == INTERESTED) ) {
                process_interested_msg(peer,buff+index,5);
                index += 5;
            }
            else if( (len-index >= 5) && (buff[index+4] == UNINTERESTED) ) {
                process_uninterested_msg(peer,buff+index,5);
                index += 5;
            }
            else if( (len-index >= 9) && (buff[index+4] == HAVE) ) {
                process_have_msg(peer,buff+index,9);
                index += 9;
            }
            else if( (len-index >= 5) && (buff[index+4] == BITFIELD) ) {
                                        process_bitfield_msg(peer,buff+index,peer-
>bitmap.bitfield_length+5);
                index += peer->bitmap.bitfield_length + 5;
            }
            else if( (len-index >= 17) && (buff[index+4] == REQUEST) ) {
                process_request_msg(peer,buff+index,17);
```

```
            index += 17;
        }
        else if( (len-index >= 13) && (buff[index+4] == PIECE) ) {
            unsigned char  c[4];
            int            length;

            c[0] = buff[index];    c[1] = buff[index+1];
            c[2] = buff[index+2];  c[3] = buff[index+3];
            length = char_to_int(c) - 9;

            process_piece_msg(peer,buff+index,length+13);
            index += length + 13;  // length+13为 piece 消息的长度
        }
        else if( (len-index >= 17) && (buff[index+4] == CANCEL) ) {
            process_cancel_msg(peer,buff+index,17);
            index += 17;
        }
        else if( (len-index >= 7) && (buff[index+4] == PORT) ) {
            index += 7;
        }
        else {
            //  如果是未知的消息类型，则跳过不予处理
            unsigned char c[4];
            int           length;
            if(index+4 <= len) {
                c[0] = buff[index];   c[1] = buff[index+1];
                c[2] = buff[index+2]; c[3] = buff[index+3];
                length = char_to_int(c);
                if(index+4+length <= len)  { index += 4+length; continue; }
            }
            //  如果是一条错误的消息，清空接收缓冲区
            peer->buff_len = 0;
            return -1;
        }
    } // for语句结束

    //  接收缓冲区中的消息处理完毕后  ，清空接收缓冲区
    peer->buff_len = 0;
    return 0;
```

```
}
```

- int parse_response_uncomplete_msg(Peer *p,int ok_len)

功能：处理收到的消息。

参数：ok_len为接收缓冲区中完整消息的长度。函数实现代码如下：

```
int parse_response_uncomplete_msg(Peer *p,int ok_len)
{
    char  *tmp_buff;
    int    tmp_buff_len;

    //  分配存储空间 ，并保存接收缓冲区中不完整的消息
    tmp_buff_len = p->buff_len - ok_len;
    if(tmp_buff_len <= 0)  return -1;
    tmp_buff = (char *)malloc(tmp_buff_len);
    if(tmp_buff == NULL) {
        printf("%s:%d error\n",__FILE__,__LINE__);
        return -1;
    }
    memcpy(tmp_buff,p->in_buff+ok_len,tmp_buff_len);
    //  处理接收缓冲区中前面完整的消息
    p->buff_len = ok_len;
    parse_response(p);
    //  将不完整的消息拷贝到接收缓冲区的开始处
    memcpy(p->in_buff,tmp_buff,tmp_buff_len);
    p->buff_len = tmp_buff_len;
    if(tmp_buff != NULL)  free(tmp_buff);

    return 0;
}
```

- int prepare_send_have_msg()

功能：为发送have消息作准备

说明：当下载完一个 piece 时，应该向所有的peer发送have消息。函数实现的代码如下：

```
int prepare_send_have_msg()
{
    Peer *p = peer_head;
    int  i;

    if(peer_head == NULL)  return -1;
    if(have_piece_index[0] == -1)  return -1;
```

```
          while(p != NULL) {
              for(i = 0; i < 64; i++) {
                                                    if(have_piece_index[i]   !=   -1)
create_have_msg(have_piece_index[i],p);
                  else  break;
              }
              p = p->next;
          }
          for(i = 0; i < 64; i++) {
              if(have_piece_index[i] == -1)  break;
              else  have_piece_index[i] = -1;
          }

          return 0;
      }
```

- int create_response_message(Peer *peer)

功能：主动创建发送给 peer 的消息，而不是等收到某个消息后再创建响应消息。函数实现的代码如下：

```
      int create_response_message(Peer *peer)
      {
          if(peer==NULL)  return -1;
          if(peer->state == INITIAL) {  // 处于 Intial状态时主动发握手消息
              create_handshake_msg(info_hash,peer_id,peer);
              peer->state = HALFSHAKED;
              return 0;
          }
          if(peer->state == HANDSHAKED) {  //  处于已握手状态，主动发位图消息
              if(bitmap == NULL)  return -1;
              create_bitfield_msg(bitmap->bitfield,bitmap->bitfield_length,peer);
              peer->state = SENDBITFIELD;
              return 0;
          }
      //  如果条件允许（未将该 peer阻塞，且  peer发送过请求），则主动发送 piece 消息
          if( peer->am_choking==0 && peer->Requested_piece_head!=NULL ) {
              Request_piece *req_p = peer->Requested_piece_head;
                      int  ret  =  read_slice_for_send(req_p->index,req_p->begin,req_p-
>length,peer);
              if(ret < 0 ) { printf("read_slice_for_send ERROR\n");}
              else {
```

```
                if(peer->last_up_timestamp == 0)
                    peer->last_up_timestamp = time(NULL);
                peer->up_count += req_p->length;
                peer->up_total += req_p->length;

                peer->Requested_piece_head = req_p->next;
                //  打印提示信息
    // printf("*** sending a slice TO:%s index:%-5d begin:%-5x ***\n",
     //      peer->ip,req_p->index,req_p->begin);
                free(req_p);
                return 0;
            }
        }
        //  如果 3 分钟没有收到任何消息关闭连接
        time_t now = time(NULL);  // 获取当前时间
        long interval1 = now - peer->start_timestamp;
        if( interval1 > 180 ) {
            peer->state = CLOSING;
            //  丢弃发送缓冲区中的数据
            discard_send_buffer(peer);
            //  将从该 peer处下载到的不足一个  piece 的数据删除
            clear_btcache_before_peer_close(peer);
            close(peer->socket);
        }
        //  如果 45秒没有发送和接收到消息 ，则发送一个keep_alive消息
        long interval2 = now - peer->recet_timestamp;
        if( interval1>45 && interval2>45 && peer->msg_len==0)
            create_keep_alive_msg(peer);

        return 0;
    }
```

- void discard_send_buffer(Peer *peer)

功能：即将与peer断开时，丢弃发送缓冲区中的消息。函数实现的代码如下：

```
    void discard_send_buffer(Peer *peer)
    {
        struct linger     lin;
        int               lin_len;

        lin.l_onoff = 1;
```

```
        lin.l_linger = 0;
        lin_len = sizeof(lin);

        //  通过设置套接字选项来丢弃未发送的数据
        if(peer->socket > 0) {
            setsockopt(peer->socket,SOL_SOCKET,SO_LINGER,(char *)&lin,lin_len);
        }
    }
```