CTorrent 程序源码分析

姚旭晨

目录

图表目录

表格目录

1. 前言

1.1 为什么要写这份文档

BitTorrent 点对点文件传输协议（以下简称 BT 协议）及其客户端应用大行其道的今天，各种各样的客户端不胜枚举（可以参看 http://wiki.theory.org/BitTorrentApplications），而各种各样的 BT 技术论坛讨论的却都是有关客户端软件如何使用的问题，有关底层协议细节和实现方案的讨论少之又少。我碰巧有机会研究过一阵 BT 协议的原理，也看过一部分源代码（CTorrent），虽然现在不再继续 BT 方面的研究了，但有感于当初看代码时遇到的资料的匮乏的窘境，便决心把自己的理解和心得写出来，算是自己的一份总结（这也是我的本科毕业论文），也希望帮助对 BT 协议实现有兴趣的人尽快上手，少走弯路。

有关 BT 协议的论述主要有三篇文章：
1，BT 官方网站上的协议解释：http://www.bittorrent.org/protocol.html。
2，Bittorrent Protocol Specification，http://wiki.theory.org/BitTorrentSpecification。
3，Incentives Build Robustness in BitTorrent，http://www.bittorrent.com/bittorrentecon.pdf。

这三篇文章从不同方面给出了 BT 协议从算法到实现的一个较为简略的描述。为了更深入地理解 BT 协议，自己动手写一个 BT 客户端或阅读一个 BT 客户端的源代码的工作是必不可少的。

1.2 客户端的选择

Bram Cohen 是 BT 协议的创建者。根据这份协议，他写了 BT 的第一个客户端，也就是 BitTorrent 公司的产品：BitTorrent。可以说，BitTorrent 的源码和 BT 协议是门当户对，要理解协议，先从 BitTorrent 的源码开始是最好不过的了。

但 Bram Cohen 是用 Python 语言写的 BitTorrent，这给很多不懂 Python 的人（我也在内）带来了很多麻烦：为了看懂一份源码而去新学一份计算机编程语言是不是有些不值得呢？

好在 BT 客户端是如此之多，我们有很大的选择空间。除了 Python，还有 Java（主要是 Azureus，国外非常流行的多平台的客户端）和 C++（其它大部分客户端）写成的程序。

经过多方比较，我选择了 CTorrent 这个客户端。虽然 CTorrent 是用 C++写成的，但仅仅算

是一个轻量级（light-weighted）的 C++软件。它的库函数依赖型很小，只用到了 Open SSL 库用来计算哈希值，所以可以工作在 Linux, FreeBSD 和 MacOS 平台。CTorrent 没有图形界面，工作在命令行模式。

另外，libtorrent（http://www.rasterbar.com/products/libtorrent.html）也是一个值得一看的客户端。Libtorrent 用到了很多 C++的模板库（主要是 boost），客户端的性能非常好，而且还提供库函数给其它程序调用。只是作者的 C++水平实在太低，对这种重量级的软件掌握不了。

1.3 CTorrent 简介

CTorrent 是由 YuHong 写的一个 BT 客户端。它的代码大部分都可以看作是 C 代码，只是用到了 C++的类概念，还有一小部分构造函数，析构函数，函数和操作符重载的代码。不懂 C++的人只需有一些 C++的基本知识就完全能看懂源代码了。CTorrent 的主页是 http://ctorrent.sourceforge.net，它遵循 GPL?。

作者在 CTorrent 主页上称自己为 YuHong，这里有一篇他写完 CTorrent 后发的帖子：http://www.freebsdchina.org/forum/viewtopic.php?p=39082，想必是中国人吧。

用户在使用时发现 CTorrent 有一些 bug，一个比较明显的例子是 CTorrent 下载完成后不会立即把缓存中的数据写入硬盘，这样如果按下 Ctrl-C 结束程序的话会造成数据的不完整。CTorrent 的最新版本是 1.3.4（2004 年 9 月 7 日发布），作者后面就没有再发布新版本，软件的一些问题也没有得到修正。

虽然有一些 bug，但得益于 CTorrent 是开源项目，很快就有人为 CTorrent 写了一些补丁（http://sourceforge.net/tracker/?group_id=91688&atid=598034）。其中一个叫 Dennis Holmes 的人贡献颇多，他为 CTorrent 打了很多 patch，然后重新发布，取名为 Enhanced CTorrent。

Enhanced CTorrent 的主页是 http://www.rahul.net/dholmes/ctorrent。目前已经更新到了 ctorrent-dhn2 版本，这个版本配合 Dennis Holmes 用 Perl 写的一个 CTorrent Control Server，可以实现对 Enhanced CTorrent 运行状况的监控。

这篇 CTorrent 的源码分析是基于 ctorrent-dhn1.2 版本的，原因是由于我查看 Enhanced CTorrent 较早，那时还没有 ctorrent-dhn2 版本，再加上自己偷懒，没有赶在 ctorrent-dhn2 发布之前把文章写完……比较而言，dnh1.2 版本已经是一个相对稳定的版本了，dnh2 的改进主要是在性能方面，而非 bug fix（容我再强词夺理一句，我简略看过 dnh2 版本的代码，在 dnh1.2 的基础上，看懂 dnh2 是没有问题的）。

另外，Dennis Holmes 虽然重新发布了 CTorrent，但他本人对原作者是极为尊敬的。在他的 dnh 版本中，原封不动地保留了原先代码的痕迹，自己的改动也加上了相应的注释。虽然 CTorrent 有一些 bug，但正如 Dennis Holmes 所言：谁又说其它客户端没有 bug 呢？我的这篇源码分析也统一称 CTorrent 和 Enhanced CTorrent 为 CTorrent，只有在需要两个版本比较时才区分开来。

2. 准备工作

2.1 知识储备

要看懂 CTorrent 源码和本篇源码分析，读者需要具备如下知识：
1，前面列举的 BT 协议的大致了解。
2，网络 socket 编程方面的基本知识，主要是 select()函数的使用。
3，至少会 C 语言，了解 C++的基本使用方法（主要是类，构造函数，析构函数和重载）。

2.2 我对本篇源码分析的说明

1， 源代码中如果出现一些乱码（特别是在终端中查看时），设置：
$export LANG=C
即可看到原作者写的中文注释。

2， 源码解说一般采取流程图的形式，有一些函数的具体功能不是很集中，画流程图也表示不出前后联系来，就直接写了步骤分析。有些源码比较晦涩的，会直接分析源代码。

3， 源代码中的全部变量都有分析。大部分函数都有说明，少数特别简单的函数和见名知意的函数没有说明。

4， 源代码中看似简单的表述实际蕴含着及其严格的操作要求（例如宏 P_HANDSHAKE 的意思是可以进行握手通信了，而不是正在进行握手通信或者已经完成握手通信了）。所以必须正确理解源代码各个宏，变量，函数的确切含义，才能真正理解程序的流程和作用。

5， 分析源码的最终目的是彻底理解 BT 协议的实现结构，以及 BT 通信性能卓越的原因。虽然程序中涉及 BT 协议算法的只有几个函数，但这几个函数是在其它大量代码的基础上构建的。一些有关种子文件的制作和解析的代码虽然看似和 BT 通信关系不大，但若前面的基础没有理解正确，会给后面的算法分析带来很大的麻烦。

6， 原作者的 C 语言技巧相当高，enjoy it!

7， 本文中“函数”指的是当前正在分析的函数，而“程序”指的是整个 CTorrent 程序。

8， 本文中“消息”指的是 peer 发来的固定格式的消息，例如 piece 消息，bitfield 消息等。“数据”指的是客户端要下载的东西，例如一个游戏，一段视频等。

9， 英文中种子文件有很多说法，如.torrent file, metainfo file，本文中均用它们的中文名：种子文件。

10，英文中关于 BT 协议的最小数据单元有很多说法，如 slice,block,subpiece，本文中使用 CTorrent 源代码中的说法：slice。

3. 总述

3.1 CTorrent 的命令行参数的意义

-h/-H：显示帮助命令
-x：只解码并显示种子文件信息，不下载。
-c：只检查已下载的数据，不下载。
-v：打开 debug 调试输出。

下载选项：

-e int	下载完毕后的做种时间（单位：小时），默认为 72 小时。
-p port	绑定端口，默认为 2706。
-s save_as	重命名下载的文件，若是下载的是多个文件，则 sava_as 是包含多文件的目录。
-C cache_size	缓存大小，默认为 16MB。
-f	强制做种模式，不进行 SHA1 HASH 检查。
-b bf_filename	piece 位图文件名，详见 BitField::SetReferFile()。
-M max_peers	客户端最多与多少个 peer 通信。
-m min_peers	客户端至少与多少个 peer 通信。
-n file_number	多文件下，选择哪个文件去下载（例如第二个文件 file_number 就为 2）。
-D rate	限制最大下载速率（单位：KB/s）。
-U rate	限制最大上传速率（单位：KB/s）。
-P peer_id	客户端通信的 ID，默认为-CD0102-。

下载数据文件示例：
ctorrent -s new_filename -e 12 -C 32 -p 6881 eg.torrent
制作种子文件示例：
ctorrent -t file_to_make.avi -s a.torrent -u protocol://address/announce

3.2 CTorrent 的状态栏的意义

CTorrent 运行时输出格式如下：
$/ 1/10/40 [3/148/148] 2MB,1MB | 48,20K/s | 80,40K E:0,1
各项意义为：
/：表明客户端正在工作的符号，以”- \ | /”循环。
1：种子数目。

10：客户端正在通信的非种子的 peer 数目。
40：tracker 服务器知道的 peer 数，也是整个 bt 通信群的 peer 数。
3：客户端已经下载的 piece 数目。
148：数据文件全部的 piece 数目。
148：客户端可以得到的 piece 数目，若此数小于全部 piece 数目则不会下载到完整的数据。
2MB：客户端已经下载的数据量。
1MB：客户端正在上传的数据量。
48：客户端的平均下载速率(KB/s)。
20：客户端的平均上传速率(KB/s)。
80：客户端的即时下载速率(KB/s)。
40：客户端的即时上传速率(KB/s)。
0：客户端与 tracker 服务器通信失败的次数。
1：客户端与 tracker 服务器通信成功的次数。

3.3 各个类实现的具体实例

CTorrent 程序使用了 C++面向对象的特性。在程序中有一些类的实例（instance），分别代表了一个 BT 通信群中的各个对象。

3.3.1 BTCONTENT

BTCONTENT 是 btContent 类实现的实例。它在程序中代表种子文件和本地数据文件。

3.3.2 PENDINGQUEUE

PENDINGQUEUE 是 PendingQueue 类实现的实例。它在程序中代表由于与 peer 的暂时通信中断而搁置等待的 slice 链表的队列。

3.3.3 IPQUEUE

IPQUEUE 是 IpList 类实现的实例。它在程序中代表从 tracker 服务器传来的 peer 列表的链表。

3.3.4 Self

Self 是 btBasic 类实现的实例。它在程序中代表客户端自己。

3.3.5 WORLD

WORLD 是 PeerList 类实现的实例。它在程序中代表所有正在与客户端通信的 peer 的链表

3.3.6 Tracker

Tracker 是 btTracker 类实现的实例。它在程序中代表 tracker 服务器。

3.4 BT 协议的特性和 CTorrent 的实现情况

BT 下载之所以性能出众是由 BT 协议所规定的一系列机制所保证的。判断一个 BT 下载软件性能优秀与否则是看这个软件对 BT 协议中下载机制的执行情况。BT 协议主要规定了两大类机制保证其性能（详细信息请参照” Incentives Build Robustness in BitTorrent”）：

3.4.1 Piece 选择机制

3.4.1.1 初始模式（Initial Mode）：Random First Piece。

当客户端刚开始运行时，它一个完整的 piece 也没有，这时需要尽快下载到一个 piece 以便可以提供上传服务。此时的算法为：第一个随机 piece。客户端会随机找到一个 piece，然后下载。

CTorrent 随机选择 piece，而且更进一步采取了一种加速下载的办法：虽然此时客户端没有 piece，但应该有向其它 peer 的申请 slice 的队列了。客户端只要比较这些队列哪个最短，优先下载最短的队列即可最快获得第一个 piece。

函数 PeerList::Who_Can_Duplicate()实现了此算法的代码。

3.4.1.2 一般模式（Normal Mode）：Strict Priority 和 Rarest First。

1，严格优先（Strict Priority）
一旦某个 slice 被申请，则这个 slice 所在的 piece 中的其它 slice 会先于其它 piece 的 slice 被申请。这样做可以尽量使最先申请的 piece 最先被下载完毕。

这条规则看似简单而且公平，但实现起来非常困难：BT 协议规定一个 piece 中的多个 slice 可以向多个 peer 申请，而客户端又同时从多个 peer 处申请了多个 piece 中的 slice，这么多数据传输队列同时进行，要保证严格优先是非常困难的。

CTorrent 在设计时采取了一种比较简单的方法：一旦向某个 peer 申请了某个 slice，则这个 piece 中的所有 slice 均向这个 peer 申请。为了保证尽快将一个 piece 下载完成，CTorrent 会找出当前正在与之通信的那个 peer（正在通信的 peer 通常比较活跃），然后把所有 slice 请求队列中最慢的那个队列找出来，交给这个 peer 去下载。

函数 PeerList::Who_Can_Abandon()实现了此算法的代码。

2，最少优先（Rarest First）

客户端下载时选择所有 peer 拥有最少的那个 piece 优先下载。
函数 BitField::Random()是有关 piece 选择机制的代码，但它只是随机选择了 piece，没有实现最少优先。

3.4.1.3 结束模式(Endgame Mode)

由于每一个 piece 只向一个 peer 申请，当 peer 数大于还没有申请的 piece 数时，客户端便进入了结束模式。此时客户端可以向所有的 peer 发送还没有下载完毕的 slice 的请求，以便尽快下载完毕好做种子。

CTorrent 变相实现了这个算法，它会找出当前正在与之通信的那个 peer（正在通信的 peer 通常比较活跃），然后把所有 slice 请求队列中最长的那个队列找出来，交给这个 peer 去下载。

函数 PeerList::Who_Can_Duplicate()实现了此算法的代码。

函数 PeerList::Who_Can_Duplicate()和 PeerList::Who_Can_Abandon()的调用环境均是函数 btPeer::RequestPiece()，应将这三个函数一起查看才能清楚 piece 选择机制的实现。

3.4.2 Choking 算法

Choking, Unchoking, Optimistic Unchoking 三个算法是保证 BT 下载公平的基石，即“一报还一报”，“人人为我，我为人人”。具体实现请参照 PeerList::UnChokeCheck()。

总的来说，CTorrent 程序简单而巧妙地实现了 BT 协议中的保证下载性能的核心算法。只是最少优先算法没有实现，会给 BT 通信群的稳定性带来一定的影响。不过，这个问题已经在 CTorrent-dnh2 版本中得到了改正和优化。

点对点(Peer to Peer)通信是 BT 协议最大的特色，它充分利用了互联网上各个下载终端的带宽，使得由服务器上传速率有限所带来的瓶颈问题得以解决。但除此以外，BT 协议本身还通过 piece 选择机制优化了下载：由于数据以 slice 的形式分块下载，一般一个 slice 只有 32KB，即使所有的 peer 的上传速率都很慢，但 slice 是如此之小，以至于从一个很慢的 peer 处获得一个 slice 所需时间也极少，BT 客户端只需合理安排对 slice 的请求，在较活跃的 peer 和下载较慢的 slice 中作出相应的调整和搭配，即可获得较高的下载速率。

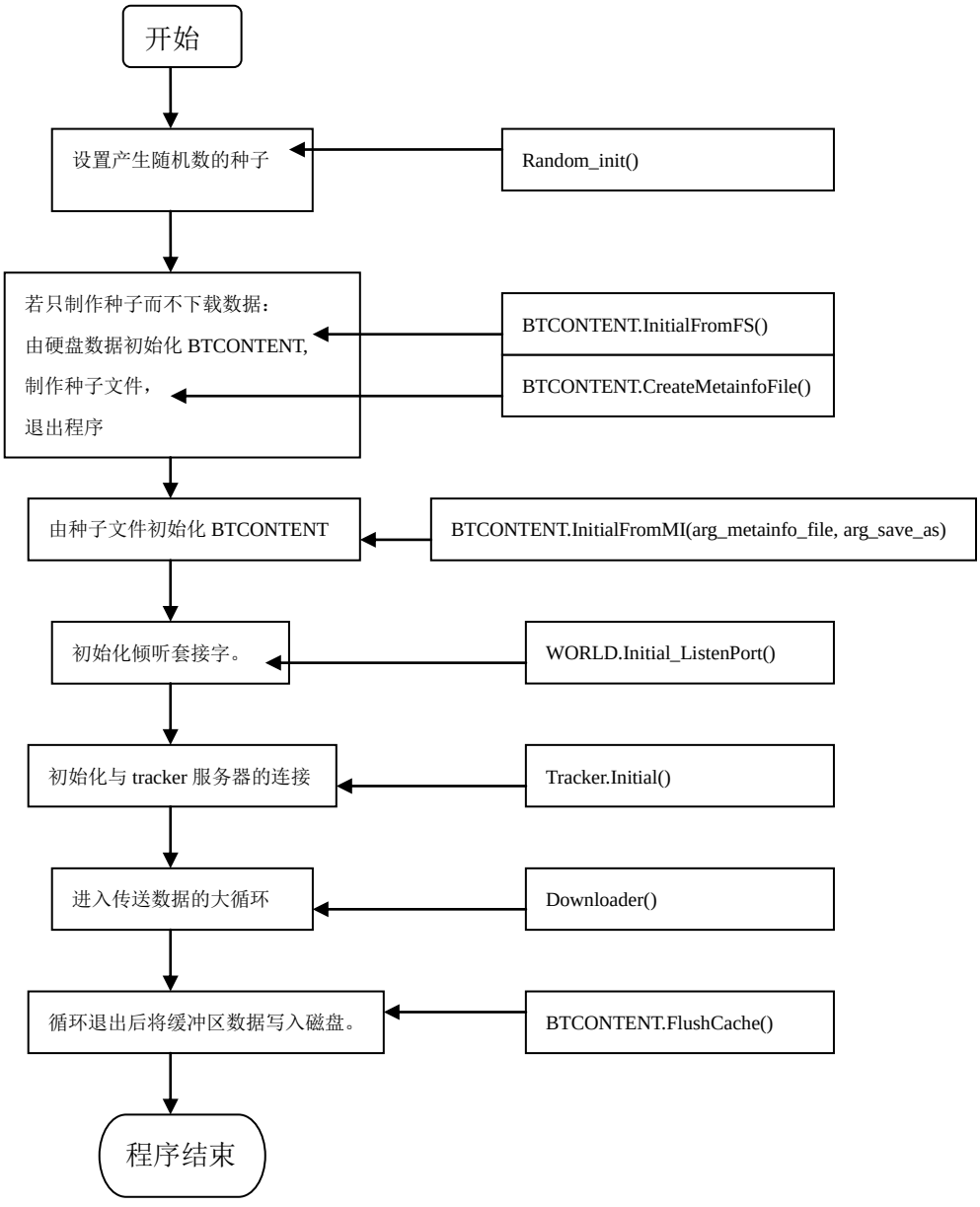
打个比方，火车站检票口处会有多个检票员（peer）在检票。每个人（slice）手里都拿着一张票，排成很多条并排的队伍（slice 队列）等候检票。由于检票员的检票速度有快有慢，控制中心（客户端程序）只需适时作出调整，将较长的队列分配给较快的检票员即可做到全体乘客的快速通过。

总结起来，BT 协议的精髓便是通过化整为零，主动选择来充分利用各个下载终端的带宽，配合以相应的公平机制，保证整个 BT 通信群的高性能和高稳定性。

4. 源代码分析

4.1 ctorrent.cpp

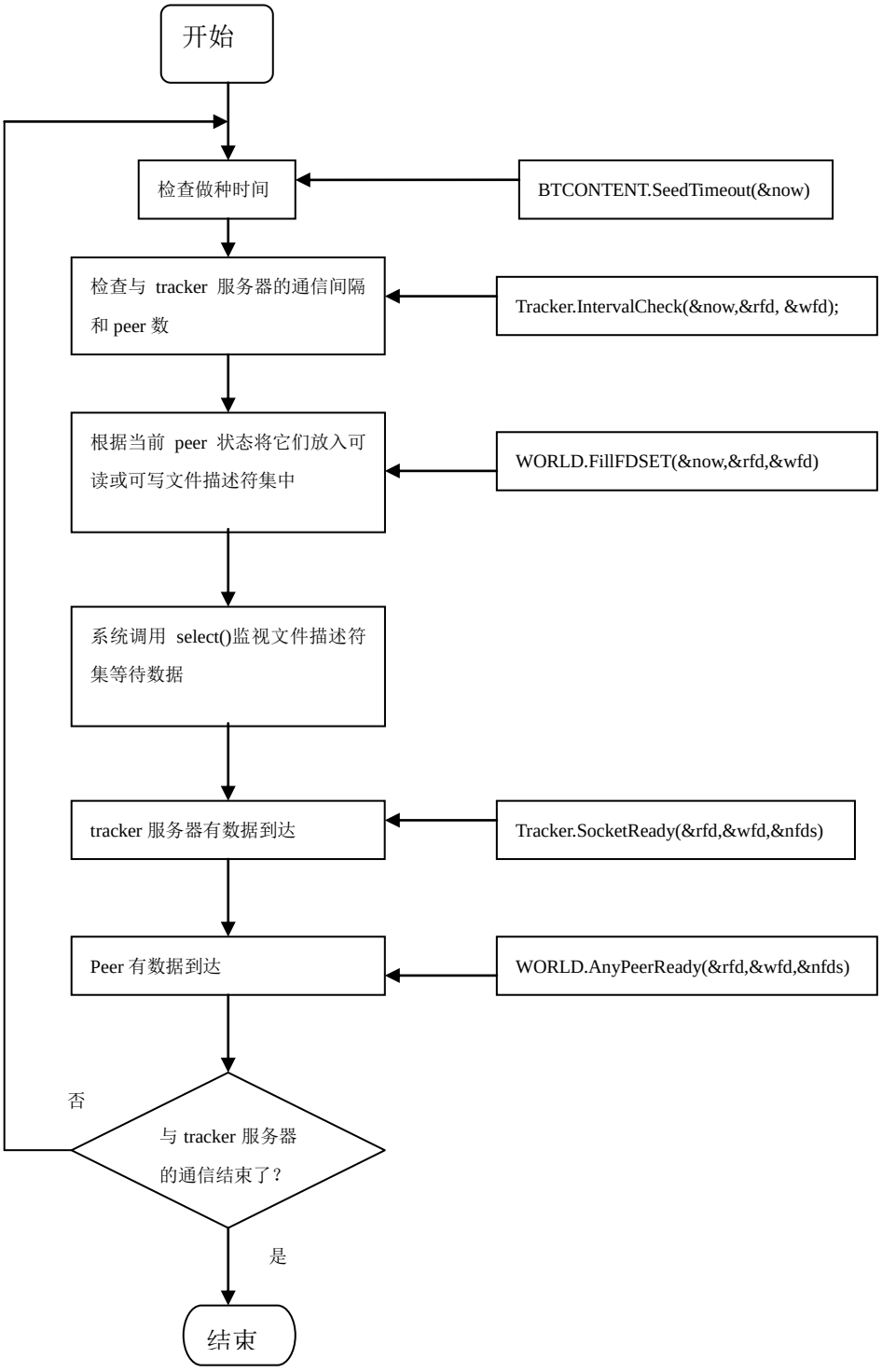
CTorrent 客户端的主程序，主要负责解析用户参数，然后建立磁盘文件并初始化和 tracker 服务器的连接，最后调用 downloader()函数进入一个数据处理的大循环。流程图如下：



图表 1 main()函数流程图

4.2 downloader.cpp

此文件只有一个程序：Downloader()，负责客户端与 tracker 服务器的通信，与 peer 的数据交换等，函数具体的实现则是通过调用各司其职的其它函数实现的。



图表 2 Downloader()函数流程图

4.3 bencode.h

此文件提供了一系列种子文件编解码的函数，其中编码函数较为简单，解码函数比较晦涩：

◆ size_t buf_int(const char *b,size_t len,char beginchar,char endchar,size_t *pi)
beginchar 非空时，解析 **i**<*integer encoded in base ten ASCII*>**e** 型表示的整数。例如：
b='i123e', len=5, beginchar='i', endchar='e', *pi=123, return=5。
beginchar 为 0 时，解析<*string length encoded in base ten ASCII*>**:**<*string data*>型表示的字符串。例如，b = ”4:spam”, len = 6, beginchar = 0, endchar = ‘:’, *pi = 4，返回值为 2（’:’和’4’的间隔）。

◆ size_t buf_str(const char *b,size_t len,const char **pstr,size_t* slen)
解析<*string length encoded in base ten ASCII*>**:**<*string data*>型表示的字符串。返回值根据 pstr 和 slen 发生变化。例如：
b="12:announce_url"
若*pstr = "announce_url"，函数将*slen 赋值为 12，返回值无实际意义。
若传递参数*pstr = 0, *slen = 0，则返回值为整个字符串的长度，即 15。

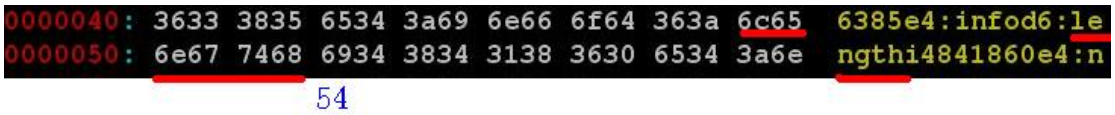
◆ size_t decode_int(const char *b,size_t len)
调用 buf_int()，返回整个以类似 **i**<*integer encoded in base ten ASCII*>**e** 表示的字符个数（’i’和’e’均计算在内）。

◆ size_t decode_str(const char *b,size_t len)
调用 buf_str()，返回整个字符串的长度。

◆ size_t decode_dict(const char *b,size_t len,const char *keylist)
解析种子文件中 dictonary 用的，由于整个种子文件就是一个 dictonary，而这个大的 dictonary 中还有小的 dictonary，所以随着 keylist 的不同，函数会用不同的方法解析 dictonary。

若 kyelist=”announce”，则函数会返回位于”announce”后面的数的位置，如图，为 0x0b。

若 keylist=”info|length”，这表明查询 info 这个 dictonary 中的”length”后面的数的位置。如图，为 0x54。



若 keylist=(char *)0，则返回整个 dictonary 的长度（从’d’到’e’）。

◆ size_t decode_list(const char *b,size_t len,const char *keylist)

返回整个 list 的长度。例如 = “**l***4:spam4:eggs***e”**，则返回 16。

◆ size_t decode_rev(const char *b,size_t len,const char *keylist)
根据*b 的数值分别调用 decode_int(),decode_dict(),decode_list(),decode_str()函数解析。

◆ size_t decode_query(const char *b,size_t len,const char *keylist,const char **ps,size_t *pi,int method)
此函数负责解析是哪个宏函数（meta_str(), meta_int(), meta_pos()）调用了它，并由解析结果去调用相应的函数。

◆ size_t decode_list2path(const char *b, size_t n, char *pathname)
该函数只在多文件情况下被调用，将以 lists 形式表示的多个文件写入 pathname。函数每次只写入一个文件名，需要多次被调用才能将所有文件名解析出来。

◆ size_t bencode_buf(const char *buf,size_t len,FILE *fp)
以如下形式将 str 写入文件 fp：
<string length encoded in base ten ASCII>**:**<string data>

例如，str=”spam”，则函数将”4:spam”（引号不包括）写入文件 fp。

◆ size_t bencode_str(const char *str, FILE *fp)
调用 bencode_buf()向文件中写入 str 的长度和 str。

◆ size_t bencode_int(const int integer, FILE *fp)
以如下形式将整数 integer 写入文件：
i<*integer encoded in base ten ASCII*>**e**

例如，integer=19，则函数将”i19e”（引号不包括）写入文件 fp。

◆ size_t bencode_begin_dict(FILE *fp)
向文件 fp 中写入字符’d’，作为 dictornary 的开始。Dictornary 结束后必须调用 bencode_end_dict_list()写入字符’e’。

◆ size_t bencode_begin_list(FILE *fp)
向文件 fp 中写入字符’l’，作为 list 的开始。list 结束后必须调用 bencode_end_dict_list()写入字符’e’

◆ size_t bencode_end_dict_list(FILE *fp)
向文件 fp 中写入字符’e’，作为 dictionary 或 list 的结束。

◆ size_t bencode_path2list(const char *pathname, FILE *fp)
该函数只在多文件情况下被调用，pathname 里储存了一个文件信息的链表。此函数将所有的文件名以 lists 的形式写入文件 fp。

4.4 bitfield.h

4.4.1 class BitField

BitField 类是一个位图。Piece 以从小到大的索引顺序在位图 unsigned char *b 中占有 1 位（bit）。

4.4.1 变量

- static size_t nbits

piece 总数。

- static size_t nbytes

位图区域 b 的大小。若共有 65 个 piece，则 nbytes = 9。即 b 占 9 个字节，但是只有前 65 位有意义。

- unsigned char *b

位图所在的内存区域，以字符串形式表示。

- size_t nset

已经获得的 piece 数，也就是 b 中被置位的位数。当 nset == nbits 时，文件的所有 piece 全部获得。

4.4.2 函数

◆ void BitField::SetAll()

为 b 开辟内存区域，并使 nset = nbits。

◆ int BitField::SetReferFile(const char *fname)

此函数在用户指定 piece 位图时使用。假定用户在硬盘上有一文件，其内容是要下载的数据文件的 piece 位图。使用”-b”参数将此文件名传给 fname。SetReferFile()便会读取位图文件，并调用 SetReferBuffer()将位图文件内容存储在 BTCONTENT.pBF 中。使用”-b”参数会使用用户指定的位图文件而非程序自己计算位图，一个 bit 之差都会导致下载失败，所以作者提醒用户小心使用。

◆ void BitField::Set(size_t idx)

将 b 中 idx 对应的位置为 1，并更新 nset 的值。

◆ void BitField::UnSet(size_t idx)

将 idx 对应的位置置为 0。注意由于 SetAll 只是开辟了内存区域，并没有将 b 全部置为 1，所以 Unset()函数还要调用_isfull()检查 b 是否已满，若是，则将 b 全部置为 1，然后再将 idx 位置为 0。

◆ int IsSet(size_t idx) const;
查看第 idx 个 piece 在 piece 位图中是否已经存在。

◆ size_t BitField::Random() const
函数随机选择 BitField 位图中存在的一个 piece 的索引 idx，并返回 idx。

程序中只有一处调用此函数：
idx = tmpBitField2.Random();
tmpBitField2 是客户端程序还没有向 peer 请求的所有 piece 的位图。上面调用的目的是随机选择一个索引为 idx 的 piece，然后向 peer 发送 request 信息。

但是函数 Random()设计得并不好，原因是它没有体现出来 BT 协议中的“最少优先”原则。

当很多 peer 在下载同一个数据文件时，总有一些 piece 是大部分 peer 都有的，而另一些 piece 只有少部分 peer 有。客户端程序在选择 piece 下载时应优先选择所有 peer 拥有最少的那个 piece，这样一来可以防止某个拥有唯一 piece 的 peer 突然退出而导致整个 BT 通信群下载的失败，二来可以将 piece 平均分布在各个 peer 中加快下载速率。这样一种选择 piece 下载的机制便叫“最少优先”（rarest first）原则。

很明显，函数只是在所有没有发出请求的 piece 种随机选择了一个 piece 而没有做到最少优先，如果一个 BT 通信群中有很多这样的客户端程序，那么这个 BT 通信群将是比较脆弱的。

◆ void BitField::Comb(const BitField &bf)
函数计算并设置 this.nset 为 this 和 bf 加起来全部的 piece 数（共有的只算 1 个），并更新 piece 位图 this 为两者全部的 piece 位图。若 this 或 bf 已满，则调用 SetAll()设置 this.nset 为 nbits，否则，调用_recalc()计算。

◆ void BitField::Except(const BitField &bf)
如果 piece 位图 b 中有某一个 piece（即相应位被置 1），而 bf.b 中没有，那么 b 的相应位被置 1。除此以外的任何其它情况，b 的相应位都被置 0。

b 有此 piece？	bf.b 有此 piece？	函数调用后位图 b 的相应位：
1	0	1
1	1	0
0	0	0
0	1	0

表格 1 BitField::Except()函数逻辑表

注意此函数中有一个较易混淆的地方，那就是“bf.b 中没有”此 piece 的真正含义。由于很多情况都把 pBFilter 传给 bf，例如：

tmpBitField.Except(*BTCONTENT.pBFilter);

而根据 pBFilter 的表示方式，若 pBFilter 某一位被置 0，则表明“有”此 piece，即需要下载这个 piece。所以，应根据 bf 的具体表示方式来判断函数的作用。

◆ void BitField::Invert()
若 b 为空，函数为 b 重新开辟内存，并使 nset 与 nbit 相等，表示 b 已满。

若 b 已满，函数将 b 全部置为 0，并使 nset 等于 0，表示 b 为空。
若两者皆非，则函数将 b 里的有意义位按位取反，并重新设置 nset。

◆ int BitField::WriteToFile(const char *fname)
程序中数次出现这样的调用：
if(arg_bitfield_file) BTCONTENT.pBF->WriteToFile(arg_bitfield_file);

当用户通过”-b”参数指定硬盘 piece 位图文件名称时，程序会调用 WriteToFile()将 BTCONTENT.pBF（也就是 piece 位图）写入硬盘。

如果用户在指定了”-b”的同时还使用”-c”参数令程序只检查硬盘上的已下载文件而不实际下载文件，程序会在检查完文件后将 piece 位图以文件名 arg_bitfield_file 写入硬盘。

◆ void BitField::SetReferBuffer(char *buf)
拷贝表示 piiece 位图的 buf 到 b 中，并重新计算位图中已拥有的 piece 个数。

◆ void BitField::WriteToBuffer(char *buf)
如果 piece 位图已满，则将 buf 全部置为 1。否则，拷贝位图到 buf 中。

◆ inline void BitField::_setall(unsigned char *buf)
将 buf 中有意义的区域置为 1。所谓有意义是指可以 buf 所代表的位图中可以表示 piece 的位。若 piece 数目不能刚好被 8 整除，buf 中最有会有几位不代表任何 piece，它们一直为 0。

◆ inline void BitField::_set(size_t idx)
函数将 idx 对应的位置为 1。注意此函数并不重新设置 nset，所以只能被已经设置好 nset 的函数（例如 Invert()）调用。

◆ inline void BitField::_recalc()
由位图 b 重新计算 nset 的值

4.5 btcontent.h

4.5.1 BTCACHE 结构体

为提高磁盘性能，类 btContent 维护一个 BTCACHE 链表，缺省为 16MB。客户端程序从磁盘读的数据或想要向磁盘写的数据会先放到 BTCACHE 链表中，直至链表满了才写入磁盘。

- u_int64_t bc_off

此 BTCACHE 链表成员的绝对偏移地址。

- size_t bc_len

此 BTCACHE 链表成员的长度。每个成员的长度不一定相同，取决于客户端想读或写的数据量。

- unsigned char bc_f_flush:1

flush 标志，若为 1 则此 BTCACHE 链表成员中的数据会被写入磁盘。若为 0 则表示此数据是从磁盘读出的。

- unsigned char bc_f_reserved:7

保留项，用途不详。

- time_t bc_last_timestamp

最后一次读或写（由 bc_f_flush 为 1 或 0 决定）的时间。

- char *bc_buf

存储的数据。

- struct _btcache *bc_next

下一个节点。

4.5.2 class btContent

btContent 是有关数据文件和种子文件的类。很多变量以 m 开头，我认为 m 代表了 metainfo，也就是俗称的.torrent“种子”文件。具体来说，它存储了以下一些数据：

4.5.2.1 变量

- char *m_announce

m_announce 存储了 tracker 服务器的 announce URL，例如：”http://192.168.1.111/announce”。

如果用户要制作种子文件，则必须指定 m_announce 的值。如果用户要根据种子文件下载数据，那么 m_announce 便存储了种子文件中的 announce URL。

- unsigned char *m_hash_table，size_t m_hashtable_length

m_hash_table 存储了所有 piece 的 SHA1 hash 值，它的长度是 m_hashtable_length。

- size_t m_piece_length， size_t m_npieces

m_piece_length 是 piece 的长度，制作种子文件时由用户指定，一般为 256KB。m_npieces 则是所有 piece 的总数了。一般最后一个 piece 的长度会小一点。

由于 SHA1 hash 的长度为 20，所以有以下关系：
m_npieces = m_hashtable_length / 20

例如，数据文件为 1000KB，假设 piece 长度定为 256KB，那么 m_npieces = 4，m_hashtable_lenth = 4 × 20 = 80。

- unsigned char m_shake_buffer[68]

m_shake_buffer 存储了 client 和 tracker 或 peer 握手时的数据。详见 BitTorrentSpecification。Ctorrent 的 m_shake_buffer 一般为以下数值：

位数	0	1…….19	20….27	28….47	48……55	56….67
填充	19	BitTorrent protocol	0	计算值	-CD0102-	随机数
解释	握手信息使用的协议，为”BitTorrent protocol”，不算引号，共 19 位		协议保留位，全部填充为 0。	种子文件的 SHA1 HASH 值，共 20 位	最后 20 位为 Peer ID。前面是用户程序的名称(CD)和版本号(0102)，以“-”开头和结尾，后面是随机数，一般为程序启动时产生的随机数。这 20 位唯一地确定了 bt 客户端的名称。	

表格 2 m_shake_buffer[68]位填充情况

CTorrent 本来的 Peer ID 是-CTorrent-，但是因为原本的 CTorrent 程序被很多客户端认为 bug 比较多（“buggy”），它们在 P2P 通信时一旦发现对方客户端是 CTorrent，就干脆采取了放弃的方式，所以 Enhanced CTorrent 将其 peer ID 改为了 CD0102，代表“Ctorrent Dnh1.2”。

- time_t m_create_date, m_seed_timestamp, m_start_timestamp

制作种子文件时，m_create_date 自然是制作时间了。下载数据文件时，m_create_date 是种子文件里“creation date”项的数值，用的是标准 UNIX 计时（1970 年 1 月 1 日 0 点到现在的秒数）。

m_start_timestamp 是客户端程序启动的标准 UNIX 计时。当下载完成后，Ctorrent 会给出下载使用的全部时间。

当下载完成后，客户端便由下载者变为了上传者（这两个词有多种说法：下载者－种子，

downloader-uploader, leecher-seeder），此时 m_seed_timestamp 被设置。程序需要记录这个时间以便在缺省做种时间（72 小时）完毕后退出。

- u_int64_t m_left_bytes

客户端缺少的字节数。程序刚运行时 m_left_bytes 就是数据文件的字节数，然后每获得一个 piece，m_left_bytes 便减去 piece_length，直到下载完毕开始做种，此时为 0。

- btFiles m_btfiles

m_btfiles 储存了种子文件中列出的供用户下载的数据文件的信息。具体请见 btFiles 类。

- BTCACHE *m_cache

详见 BTCACHE 结构体。

- size_t m_cache_size, m_cache_used

m_cache_size 为 BTCACHE 结构体链表中能储存的最大数据量，也就是缓存大小，一般为 16MB。m_cache_used 是已用缓存数。

- BitField *pBF

要下载的数据文件的 piece 位图。每下载成功一个 piece，将相应位（bit）置 1。表明客户端已经拥有此 piece。

- BitField *pBFilter

要下载的文件的过滤器，也是 piece 位图。在多文件情形下，用户可能会只选择自己需要的文件下载。此时程序会调用 btContent::SetFilter()将需要下载的 piece 在位图中所对应的位置 0。程序只下载 pBFilter 中置 0 的 piece。

- char *global_piece_buffer

一个 piece 长度的数据缓冲区，函数调用 btContent::ReadPiece()或 btContent::ReadSlice()从磁盘读取数据时会将数据放入 global_piece_buffer 中。

4.5.1.2 函数

◆ int btContent::InitialFromFS(const char *pathname, char *ann_url, size_t piece_length)

当用户要制作种子文件时，程序调用 InitialFromFS，表示从本地获取数据文件，并通过计算文件大小，SHA1 Hash 值等将 btContent 中的变量初始化。具体来说，此函数的执行步骤为：

1， 设置 m_piece_length, m_announce, m_create_date。

2， 调用函数 BuildFromFS()设置 m_btfiles。

3， 由计算得到的 m_piece_length 为 global_piece_buffer 开辟内存空间。

4， 计算 m_npieces, m_hashtable_length。

5， 调用 GetHashValue()设置 m_hash_table。

制作种子文件时用户很有可能看到这样的输出：
Create hash table(already pieces/total pieces):18/19 Complete.

显示 18/19 后却加了一个 Complete，到底有没有完成呢？

这是 InitialFromsFS()在最后显示上的小 bug：

```
percent = m_npieces / 100;
if( !percent ) percent = 1;

for( n = 0; n < m_npieces; n++){
   if( GetHashValue(n, m_hash_table + n * 20) < 0) return -1;
   if( 0 == n % percent ){
      printf("\rCreate hash table(already pieces/total pieces): %u/%u", n, m_npieces);
      fflush(stdout);
   }
}
printf(" Complete.\n");
```

假设文件被分成 19 个 piece，即 m_npiece 为 19。那么 percent 为 1。由于 piece 的索引从 0 开始，当 n=18 时 Hash 表已经制作完了，所以当 n=19 时 for 循环退出直接显示 complete 了。改正的方法很简单，在

```
printf(" Complete.\n");
```

前加一句：

```
printf("\rCreate hash table(already pieces/total pieces): %u/%u", m_npieces , m_npieces);
```

即可。

◆ int btContent::GetHashValue(size_t idx,unsigned char *md)
此函数将以 idx 为索引的 piece 读入 global_piece_buffer 中（ReadPiece()），计算 SHA1 Hash 值（Sha1()），并将此值储存在 md 中。

◆ ssize_t btContent::ReadPiece(char *buf,size_t idx)
此函数实际是调用了 ReadSlice 将以 idx 为索引的 piece 读入 buf 中。

◆ ssize_t btContent::ReadSlice(char *buf,size_t idx,size_t off,size_t len)
此函数的作用是将第 idx 个（索引从 0 开始）piece 中以 off 为偏移，len 长度的数据读入 buf 中。刚进入函数时，有一判断：

```
if( !m_cache_size ) return m_btfiles.IO(buf, offset, len, 0);
else{…}
```

当用户仅仅只想制作种子文件时，并不需要 m_cache，因此也就没有调用 CacheConfigure()

将 m_cache_size 赋新值。函数直接调用 m_btfiles.IO()，将数据读出。

若 m_cache_size 已被配置（缺省为 16MB），则函数除了将数据读入 buf 中，还会调用 btContent::CacheIO()将数据放入 BTCACHE *m_cache 链表中。

◆ void btContent::CacheConfigure()
配置硬盘缓存 m_cache_size 大小，缺省为 16MB。也可由用户指定，但最大为 128MB。

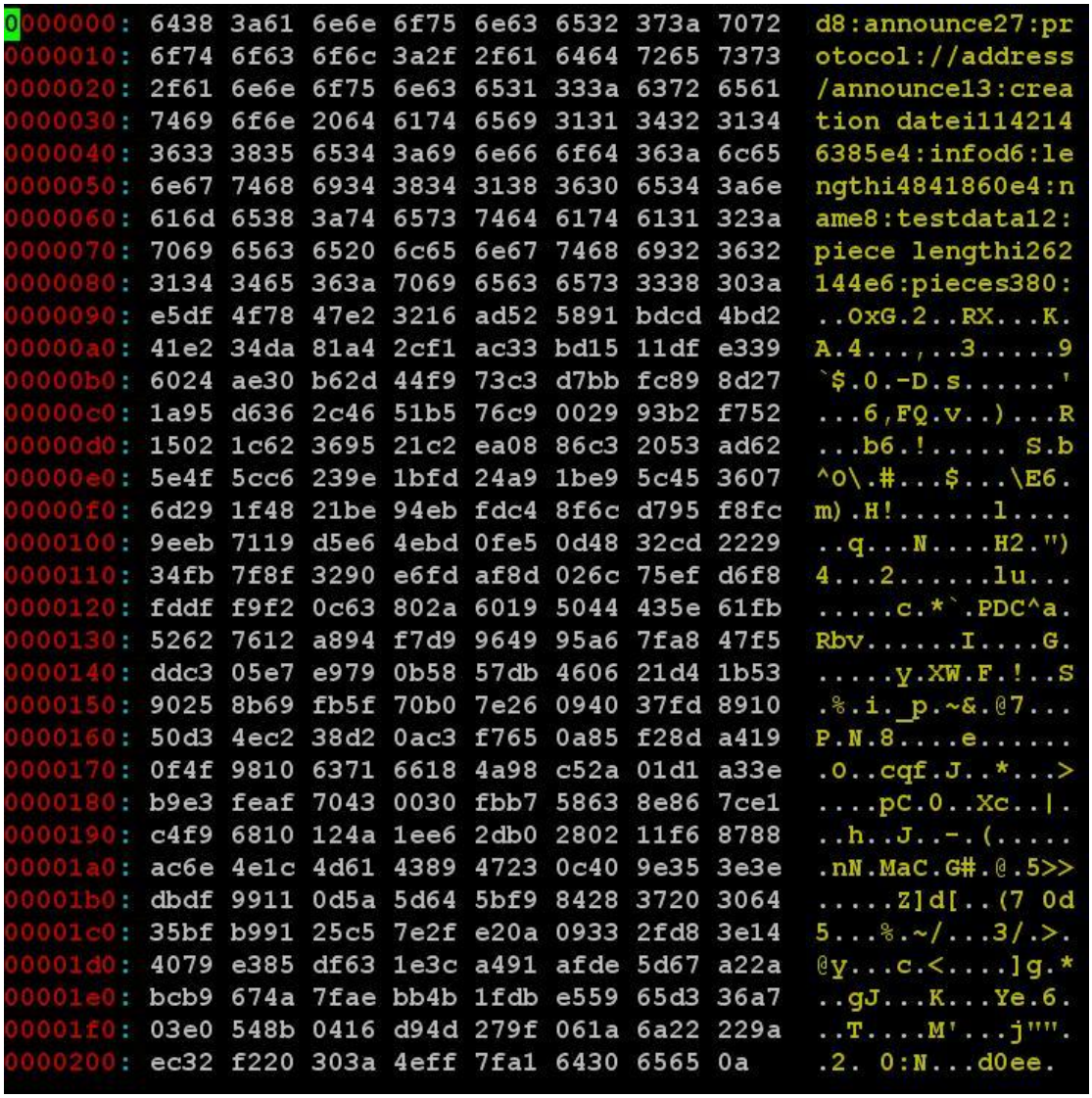
◆ int btContent::CreateMetainfoFile(const char *mifn)
此函数用来制作种子文件，并将文件名保存为 mifn。

有关种子文件的编码规范，请参照 Bit Torrent Specification v1.0。在这里我们举一个例子来说明 CreatMetainfoFile()如何制作种子文件。

取源文件 testdata(4841860B)，假设保存为 a.torrent 种子文件，使用如下命令：
$ ctorrent -t testdata -s a.torrent -u protocol://address/announce

用 vi 打开种子文件，命令：
:%!xxd

将其转换为 16 进制形式（:%!xxd –r 反转换），显示内容（保存后才会有高亮显示）如下：



:%!xxd 的意思是对整个（%）文件执行外部（!）命令（xxd）。注意最后值为 0x0a 的点号'.'，这是 xxd 程序自己加上去的，不是种子文件中有的。

函数运行步骤如下：

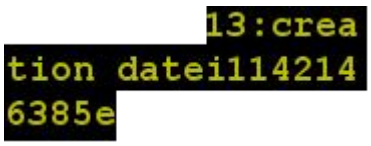
1， 使用 fopen()检测种子文件 a.torrent 是否已经存在，若没有则使用可写方式创建它。

2， 调用函数 size_t bencode_begin_dict(FILE *fp)向文件中插入字符'd'，表示 dictonary。Dictonary 要以'e'结尾，注意上图中倒数第二个字符（位于 020d 位置），也就是最后一个 'e'，便是此结尾。

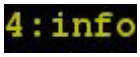
3， 调用函数 bencode_str("announce", fp)向文件中写入入字符串，表示 8 个字符长度的 announce 后面要跟一个 tracher 服务器的 announce 地址。

4， 调用函数 bencode_str(m_announce, fp) 将 m_announce(“protocol://address/announce”)写入文件。

5， 调用函数 bencode_str("creation date", fp)和 bencode_int(m_create_date, fp)向文件中写入长为 13 的字符串”creation date”，然后在将以标志 UNIX 计时表示的文件创建时间 m_create_date 写入。BT 协议规定整数以’i’开头以’e’结束。即：i1142146385e。

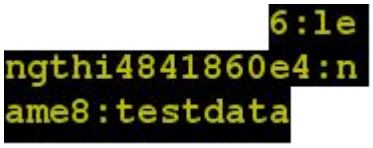


6， 调用函数 bencode_str("info", fp)将”info”字符串写入文件。



7， ‘info’后面的内容也为一个 dictonary，所以仍旧要用 bencode_begin_dict(fp) != 1 写入’d’字符。倒数第三个字符’e’便是此 dictonary 的结束符。

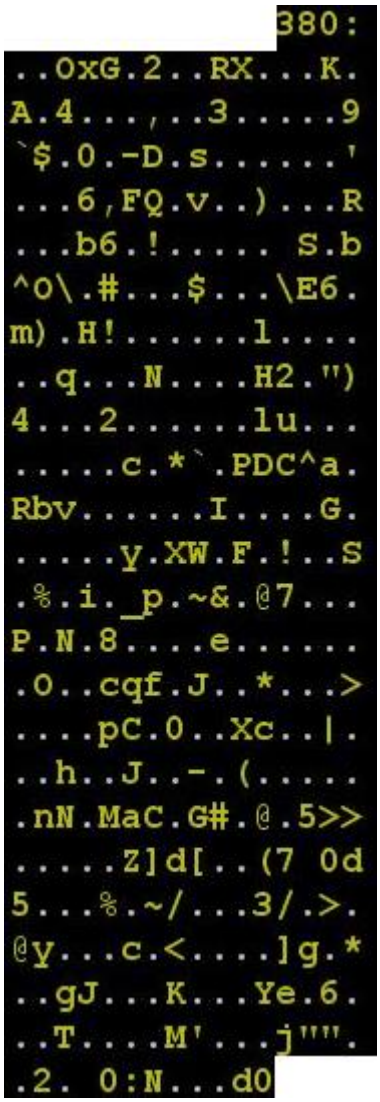
8， 调用函数 m_btfiles.FillMetaInfo()将储存在 m_btfiles 中的数据文件信息写入种子文件。FillMetaInfo()针对源文件是否为多文件有两种做法，我们举的例子是单文件情形，所以此函数写入了”length”和”name”两项。



9， 调用函数 bencode_str("piece length", fp)和 bencode_int(m_piece_length, fp)写入长为 12 的字符串”piece length”和 m_piece_length 的数值：262144，也就是 256K。

10，调用 bencode_str("pieces", fp)将长为 6 的字符串”piece”写入文件。

11，调用 bencode_buf((const char*) m_hash_table, m_hashtable_length, fp)将 m_hash_table 写入文件。注意前 3 个数是 380，表明有 19 个 Hash 值长 20 的 piece。



12，两次调用 bencode_end_dict_list(fp)将 dictonary 的结束符’e’写入文件。

13，退出，种子文件制作完成。

◆ int btContent::InitialFromMI(const char *metainfo_fname,const char *saveas)
此函数读取种子文件包含的信息初始化 btContent 类 BTCONTENT。源代码和注释如下：

```
int btContent::InitialFromMI(const char *metainfo_fname,const char *saveas)
{
#define ERR_RETURN()        {if(b) delete []b; return -1;}
   unsigned char *ptr = m_shake_buffer;
   char *b;
   const char *s;
   size_t flen, q, r;

   //将种子文件信息读入内存区域 b 中。
```

```
  b = _file2mem(metainfo_fname,&flen);
  if ( !b ) return -1;

  // 将 announce URL 的信息拷贝入 s 中，长度为 r。
  if( !meta_str("announce",&s,&r) ) ERR_RETURN();
  if( r > MAXPATHLEN ) ERR_RETURN();
  m_announce = new char [r + 1];
  memcpy(m_announce, s, r);
  m_announce[r] = '\0';

  // infohash
  if( !(r = meta_pos("info")) ) ERR_RETURN();//r 现在处于种子文件中”info”字符串后面一个字节的位置。
  if( !(q = decode_dict(b + r, flen - r, (char *) 0)) ) ERR_RETURN();//解析 info 这个 dictonary。
  Sha1(b + r, q, m_shake_buffer + 28);//将 info 这个 dictonary 的 SHA1 Hash 值（从’d’到’e’）放入 m_shake_buffer 中。
  if( meta_int("creation date",&r)) m_create_date = (time_t) r;

  // 将”pieces”字符串后面的数值（表明了 hashtable 的长度。例如此数值为 390，则有 19 个 piece，hashtable 长 19×20 ＝ 390。）放入 m_hashtable_length 中。
  if( !meta_str("info|pieces",&s,&m_hashtable_length) || m_hashtable_length % 20 != 0) ERR_RETURN();

  m_hash_table = new unsigned char[m_hashtable_length];

#ifndef WINDOWS
  if( !m_hash_table ) ERR_RETURN();
#endif
//将种子文件中的哈希表填充入 m_hash_table，与 tracker 服务器通信时会发送 m_hash_table 的内容以便让 tracker 服务器辨别客户端是否在使用服务器端已有的并且正确的种子文件。
  memcpy(m_hash_table, s, m_hashtable_length);

//将种子文件中规定的 piece 长度赋给 m_piece_length。
  if(!meta_int("info|piece length",&m_piece_length)) ERR_RETURN();
  m_npieces = m_hashtable_length / 20;

  if( m_piece_length > cfg_max_slice_size * (cfg_req_queue_length/2) ){
    fprintf(stderr,"error, piece length too long(big, OK?:-))[%u]. please recompile CTorrent with a larger cfg_req_queue_length or cfg_max_slice_size in <btconfig.h>.\n", m_piece_length);
    ERR_RETURN();
  }

  if( m_piece_length < cfg_req_slice_size )
    cfg_req_slice_size = m_piece_length;
```

```
    else{//确保一个 piece 拥有不超过 64 个 slice。
        for( ;(m_piece_length / cfg_req_slice_size) >= cfg_req_queue_length; ){
            cfg_req_slice_size *= 2;
            if( cfg_req_slice_size > cfg_max_slice_size ) ERR_RETURN();
        }
    }

//调用 BuildFromMI()。
    if( m_btfiles.BuildFromMI(b, flen, saveas) < 0) ERR_RETURN();

    delete []b;
    b = (char *)0;
    PrintOut();
//若只是检查硬盘文件，到此为止，否则便是下载文件。
    if( arg_flg_exam_only ) return 0;

    if( ( r = m_btfiles.CreateFiles() ) < 0) ERR_RETURN();

//为全局 piece 缓冲区开辟内存。
    global_piece_buffer = new char[m_piece_length];
#ifndef WINDOWS
    if( !global_piece_buffer ) ERR_RETURN();
#endif

//piece 位图
    pBF = new BitField(m_npieces);
#ifndef WINDOWS
    if( !pBF ) ERR_RETURN();
#endif

//设置下载文件过滤器
        pBFilter = new BitField(m_npieces);
#ifndef WINDOWS
         if( !pBFilter ) ERR_RETURN();
#endif
    if(arg_file_to_download>0){
        m_btfiles.SetFilter(arg_file_to_download,pBFilter,m_piece_length);
    }


//利用 m_left_bytes 检查 m_piece_length, m_npieces 是否匹配。
    m_left_bytes = m_btfiles.GetTotalLength() / m_piece_length;
    if( m_btfiles.GetTotalLength() % m_piece_length ) m_left_bytes++;
    if( m_left_bytes != m_npieces ) ERR_RETURN();
```

```
//给 m_left_bytes 赋值。
   m_left_bytes = m_btfiles.GetTotalLength();


   if( arg_bitfield_file ){
      if( !arg_flg_check_only ){
         if( pBF->SetReferFile(arg_bitfield_file) >= 0){//将硬盘上的 piece 位图文件读入 pBF 中
            size_t idx;
            r = 0;
            for( idx = 0; idx < m_npieces; idx++ )
               if( pBF->IsSet(idx) ) m_left_bytes -= GetPieceLength(idx);
         }
         else{
            fprintf(stderr,"warn, couldn't set bit field refer file %s.\n",arg_bitfield_file);
         }
      }

      if( r ) CheckExist();//file exists.

   }else if( arg_flg_force_seed_mode ){//强制做种
      pBF->SetAll();
      m_left_bytes = 0;
   }else if( r ){
      CheckExist();
   }

   printf("Already/Total: %u/%u\n",pBF->Count(),m_npieces);

   if( arg_flg_check_only ){//只检查已下载文件时
      if( arg_bitfield_file ) pBF->WriteToFile(arg_bitfield_file);//将 piece 位图 pBF 写入硬盘。
      exit(1);
   }

   CacheConfigure();

/*设置 m_shake_buffer
```

位数	0	1…….19	20….27	28…….47	48……55	56….67
填充	19	BitTorrent protocol	0	20 位 HASH 值	-CD0102-	随机数

```
*/
   *ptr = (unsigned char) 19; ptr++; // protocol string length
   memcpy(ptr,"BitTorrent protocol",19); ptr += 19; //    protocol string
```

```
    memset(ptr,0,8);                    // reserved set zero.

    {                                   // peer id
            char *sptr = arg_user_agent;
            char *dptr = (char *)m_shake_buffer + 48;
            char *eptr = dptr + PEER_ID_LEN;
            while (*sptr) *dptr++ = *sptr++;
            while (dptr < eptr) *dptr++ = (unsigned char)random();
    }
    return 0;
}
```

◆ char* btContent::_file2mem(const char *fname, size_t *psiz)

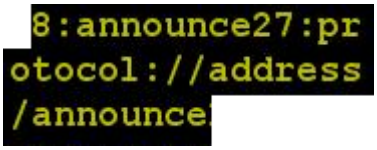
fname 为种子文件的名称。此函数将 psiz 设为种子文件的长度（单位：B），并将种子文件读入到内存中，再返回这段内存的指针。

◆ 宏函数：meta_str(), meta_int(), meta_pos()

这三个函数是函数 decode_query()的变体，提供种子文件的解码。概括一点说，就是找到种子文件中的一些“关键字”（announce, info, creation date 等），再将描述关键字的值提取出来。从源代码举几个例子：

(1) meta_str("announce",&s,&r)（const char * s, size_t r）

将种子文件的 announce URL 放入内存 s，announce URL 的值赋给 r。对于下图，s = “protocol://address/announce”，r = 27。

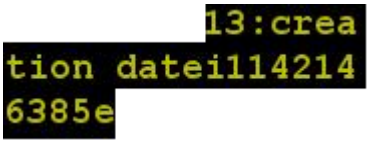


(2) r = meta_pos("info")

返回种子文件中位于”infor”字符串后面一位的字符的位置。对于下图，r=0x4B。

(3) meta_int("creation date",&r)

提取种子文件中”creation date”数值到 r。对于下图，r=1142146385。



◆ int btContent::CheckExist()

若程序启动时发现硬盘上存在数据文件，程序便会调用 CheckExist()按 piece 检查此数据文件的 Hash 值是否与种子文件中的 Hash 值相等。若相等，便将 piece 位图对应位置为 1。

◆ static void Sha1(char *ptr,size_t len,unsigned char *dm)
此函数用于计算 ptr 中的 SHA1 Hash 值，并将计算结果（20 字节）放在 dm 中。

◆ size_t btContent::GetPieceLength(size_t idx)
计算并返回第 idx 个 piece 的长度。

◆ ssize_t btContent::WriteSlice(char *buf,size_t idx,size_t off,size_t len)
函数调用 CacheIO()将 buf 中的数据写入缓冲区，若缓冲区的 bc_f_flush 为 1 则写入硬盘。

◆ void btContent::CacheClean()
此函数寻找 BTCHACHE m_cache 链表中 bc_f_flush 标志位为 0 的 bc_last_timestamp 最小的那个成员，将其删除，并更新 m_cache_used 长度。当 m_cache_used 已经接近 m_cache_size，再加入新的数据就不够用时，此函数被调用。

函数中用了四个比较晦涩的指针：
BTCACHE *p, *pp, *prm, *prmp；

猜测它们表示的意思为：*p: poniter; *pp: poniter previous; *prm: poniter read mark; *prm: pointer read mark previous;

*prm 永远指在 bc_f_flush 为 0 的成员上，但*prmp 不一定。

◆ ssize_t btContent::CacheIO(char *buf, u_int64_t off, size_t len, int method)
函数根据 off 在 BTCACHE m_cache 链表中插入一个新的节点。若 method 为 0 则将硬盘上的数据读入 buf 中，再将 buf 中的的数据读入这个新的节点，若 method 为 1 则将 buf 中的数据拷贝入这个节点。

◆ int btContent::APieceComplete(size_t idx)
函数判断第 idx 个 piece 是否被正确下载。函数首先会比较 piece 的 SHA1 HASH 值，若相同，则说明下载的数据正确，然后更新 piece 位图，最后减少 m_left_bytes。

◆ int btContent::SeedTimeout(const time_t *pnow)
根据 m_seed_timestamp 判断是否需要做种，若客户端已下完数据，且 m_seed_timestamp 为 0 则进入做种状态，此时 m_seed_timestamp 被赋值，返回 0。若已做种时间超过了默认做种时间（72 小时），则返回 1。

4.6 btfiles.h

4.6.1 Struct BTFILE

结构 BTFILE 存储文件的路径，长度，分块等信息。如果要下载的是多个文件，那么 BTFILE 便是一个文件链表，此时包含多个文件的整个链表便被当成一个大的文件看待。

- char *bf_filename

文件的绝对路径

- size_t bf_length

文件长度。单位：B。

- FILE *bf_fp;

文件的 FILE 指针。

- time_t bf_last_timestamp

最后一次进行 io 操作的时间戳。

- size_t bf_completed

已经下载的数据长度。单位：B。

- size_t bf_npieces

文件的 piece 数目。

- unsigned char bf_flag_opened:1

文件打开标志位。若此文件已被_btf_open()打开，则置为 1。

- unsigned char bf_flag_need:1

程序中只给出了定义，没有使用。

- unsigned char bf_reserved:6

保留项，没有使用。

- struct _btfile *bf_next

链表中下一个文件的指针。单文件情况下为空。

4.6.2 Class btFiles

类 btFiles 包含了需要下载的数据文件的信息。这些文件可能是一个，也可能是多个。

4.6.2.1 变量

- BTFILE *m_btfhead

详见 Struct BTFILE。

- char *m_directory

在多文件情形下，制作种子文件时，为包含要制作种子的文件的目录；下载数据文件时，为

种子文件中包含所有文件的那个目录。若只有一个文件，则为空。

- u_int64_t m_total_files_length

所有文件长度。单位：B。

- size_t m_total_opened

已打开的所有文件数目。

- u_int8_t m_flag_automanage:1

m_flag_automanage 是自动管理文件打开数目的标志。虽然初始值为 1，但在构造函数中又将此项赋为 0。所以有关自动管理打开文件的功能并不可用（具体实现函数请见_btf_open()）。

- u_int8_t m_flag_reserved

保留项，程序中没有使用。

4.6.2.2 函数

◆ BTFILE* btFiles::_new_bfnode()

函数创建并返回一个新的 BTFILE 节点。

◆ int btFiles::_btf_open(BTFILE *pbf)

此函数解析 pbf 中的文件名 pbf->bf_filename，将此文件打开。并设置 pbf->bf_flag_opened 为 1， 将 m_total_opened 加 1。

注意函数一开始有一个判断语句：

```
if(m_flag_automanage && (m_total_opened >= MAX_OPEN_FILES)){…}
```

此语句的作用是确保打开的文件总数小于 MAX_OPEN_FILES(20)。它如果 m_total_opened 大于 MAX_OPEN_FILES，那么便比较每个已打开的文件的时间戳，找出最先打开的那个，然后用 fclose()将其关闭。或许这便是 automanage 的含义吧。
不过由于程序中 m_flag_automanage 一直为 0，所以上面的 if 语句也就不起作用了。

◆ int btFiles::_btf_ftruncate(int fd,size_t length)

使文件 fd 的大小正好为 length 个字节。此函数与 c 库函数 ftruncate()相同点是，若文件原长度大于 length，两个函数均将其截断；不同点是，若文件原长度小于 length，ftruncate()将少的部分添 0 补齐，_btf_ftruncate()直接返回-1。

◆ int btFiles::_btf_creat_by_path(const char *pathname, size_t file_length)

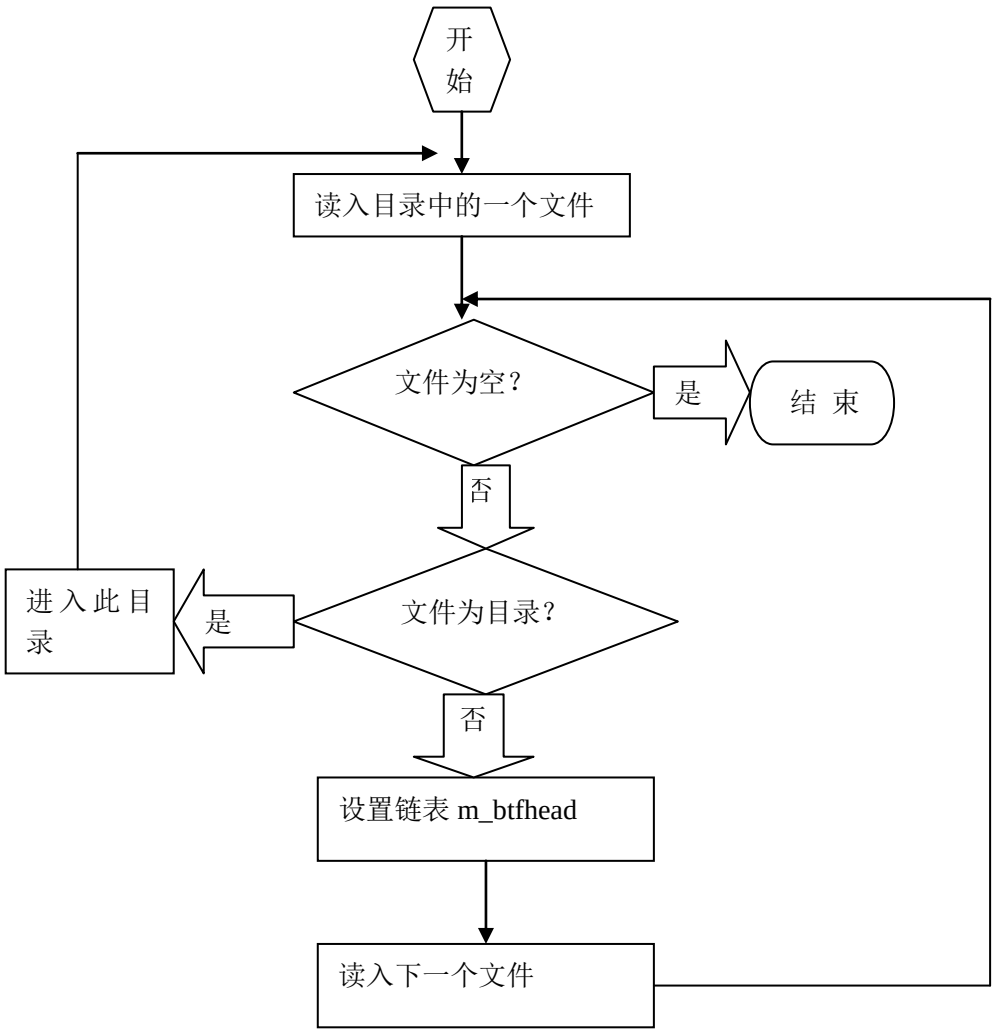
以 pathname 在硬盘上建立文件。由于 pathname 可能包含目录项（例如”dir_a\dir_b\file_c”，BT 协议使用了 Winows 的目录分隔符’\’），函数设置标志位 last 以区分应建立目录（last = 0）还是文件(last = 1)。

◆ int btFiles::_btf_destroy()

函数被 btFiles 类的析构函数调用，函数会清理数据区防止内存泄漏。

◆ int btFiles::_btf_recurses_directory(const char *cur_path, BTFILE* lastnode)

此函数使用了递归调用方法，将目录中的所有文件读出，若是普通文件就设置 struct BTFILE 链表，若目录里包含子目录就调用自己，再将子目录中文件读出。流程如下：



图表 3 btFiles::_btf_recurses_directory()函数流程图

◆ int btFiles::CreateFiles()

此函数根据 BTFILE *m_btfhead 中的文件路径名检查磁盘上的文件。

若文件存在，则返回 1。程序会调用 CheckExist()检查存在的文件的 Hash 值以确定是否为所要下载的数据文件。

若文件不存在，则在循环 for(; pbt; pbt = pbt->bf_next){…}里调用_btf_creat_by_path()建立文件，成功返回 0，失败返回-1。

◆ int btFiles::BuildFromFS(const char *pathname)

当用户要制作种子文件时，将数据文件以“-t”参数传给 arg_metainfo_file，若有多个数据文件，则需要将这些文件放在一个目录里，然后在“-t”参数后接目录名。例如：

ctorrent -t file_to_make -s a.torrent -u protocol://address/announce

ctorrent -t dir_that_conatains_files -s b.torrent -u protocol://address/announce

此函数设置 m_btfhead，pathname 为文件名（单文件时）或目录名（多文件时）。

单文件时较为简单，将 struct BTFILE 赋值。多文件时，由于 pathname 为目录名，所以需要调用_btf_recurses_directory()将所有文件遍历，并设置 BTFILE 链表。

◆ int btFiles::BuildFromMI(const char *metabuf, const size_t metabuf_len, const char *saveas)

此函数将 metabuf 中的种子文件信息赋给 m_btfhead。

如果 metabuf 中有多个文件（存在”files”项），则将 saveas 赋给 m_directory，然后设置 BTFILE *m_btfhead 为表示多个文件的链表。

如果只有一个文件（存在”name”项），则将 saveas 赋给 m_btfhead->bf_filename。若 saveas 为空，那么 m_btfhead->bf_filename 便是种子文件中”name”项的内容。

◆ ssize_t btFiles::IO(char *buf, u_int64_t off, size_t len, const int iotype)

当 iotype 为 0 时，此函数将以 off 为绝对偏移的 len 长的数据读入 buf 中。

当 iotype 为非 0 时，此函数将 buf 中的数据写入本地文件中以 off 为偏移，长为 len 的区域。

注意函数中变量 off 和 pos 的含义：由于在多文件情况下程序将所有文件读入 m_btfhead 链表，并把此链表看作一个大的文件，所以 off 便是此文件的绝对偏移。

但是函数 IO()操作的对象是硬盘上的多个文件，所以函数一开始便需要根据 off 和 len 计算此偏移地址具体在哪个文件的相对偏移上，此相对偏移便是 pos。

◆ size_t btFiles::FillMetaInfo(FILE* fp)

将文件的名称，长度，路径等信息写入种子文件。单文件和多文件的格式不同，以标志位 m_directory 区分。

◆ void btFiles::SetFilter(int nfile, BitField *pFilter, size_t pieceLength)

nfile 是用户选择下载的那个文件的序号（序号从 1 开始）。假设共有四个文件，用户想下载第二个，则：

$ctorrent –n 2 a.torrent

nfile 等于 2。

此函数根据 nfile 和 pieceLength 计算出第 nfile 个文件的所有 piece 在 pFilter 位图中的位号，并将相应位号对应的位置 0。其它位为 1。

如果 nfile 大于文件个数（上例中 nfile > 4），则程序假设所有文件均被下载。

这段程序代码写得质量不是很高，主要是在调用 SetAll()函数时，只是用 new 开辟了一段内存赋给 pFilter，并没有将 pFilter 全部置为 1。以致必须在后面再次调用 Unset()函数将全部位置为 1。最终使得两个函数都有点名不副实。

◆ size_t btFiles::getFilePieces(unsigned char nfile)
多文件情况下函数返回第 nfile 个文件（序号从 1 开始）的 piece 数目。

4.7 btrequest.h

4.7.1 class RequestQueue

每一个 peer 都有两个 RequestQueue 类：RequestQueue request_q 和 RequestQueue reponse_q。前者代表客户端想从 peer 下载的 slice 队列；后者代表 peer 需要客户端上传给它的 slice 队列。

4.7.1.1 变量

➢ PSLICE rq_head
BT 协议中对数据的下载是以 slice 为单位的。rq_head 代表 slice 队列的头，其结构为：

```
typedef struct _slice{
    size_t index;
    size_t offset;
    size_t length;
    struct _slice *next;
}SLICE,*PSLICE;
```

index 为 slice 所在的 piece 索引，offset 为 slice 在 piece 中的偏移，length 为 slice 的长度。

4.7.1.2 函数

◆ void RequestQueue::Empty()
调用_empty_slice_list()删除 PSLICE 链表。

◆ void RequestQueue::SetHead(PSLICE ps)
此函数调用_empty_slice_list()将 slice 队列置空，然后将 ps 赋给 slice 队列的头：rq_head。

◆ int RequestQueue::IsValidRequest(size_t idx,size_t off,size_t len)

当 peer 发来 request 信息向客户端索要某个 slice 时，客户端会先调用此函数检查索要的 slice 是否有效。注意函数源码中有一段判断 len 是否为合法值的：

len <= 2 * cfg_max_slice_size

一般来说 len 必须小于 cfg_max_slice_size，但这儿为什么要乘以 2 呢？原因在于 BT 协议规定 slice 的最大长度为 131072（128KB），但大多数 BT 客户端最大只接受 64KB 的 slice，为了保证自己的 slice 能被别的客户端接收，CTorrent 设置 cfg_max_slice_size 的大小为 65536（64KB），同时为了防止有些客户端发来 128KB 的 slice 请求，Ctorrent 使用了 2 倍的 cfg_max_slice_size 判断来保持兼容性。

◆ int RequestQueue::CopyShuffle(RequestQueue *prq)

把 prg 中的每个 slice 拷贝给以 this.rq_head 开头的 PSLICE 链表。由源程序：

```
  for (ps = prq->GetHead(); ps; ps = ps->next) {
    if (random()&01) {
      if (Add(ps->index, ps->offset, ps->length) < 0) return -1;
    }
    else if (Insert(ps->index, ps->offset, ps->length) < 0) return -1;
  }
```

看出对于每一个 slice，有一半的几率将其放入链表尾（Add()），一半的几率将其放入链表头（Insert()）。

◆ size_t RequestQueue::Qsize()

函数计算并返回 PSLICE rq_head 链表中的成员个数。

◆ int RequestQueue::Insert(size_t idx,size_t off,size_t len)

将第 idx 个 slice 放到 PSLICE 队列（以 rq_head 为头）的最前面。

◆ int RequestQueue::Add(size_t idx,size_t off,size_t len)

将第 idx 个 slice 放到 PSLICE 队列（以 rq_head 为头）的最后。

◆ int RequestQueue::Remove(size_t idx,size_t off,size_t len)

将在第 idx 个 piece 中的偏移地址为 off 长度为 len 的 slice 从 request_q 中移除。

◆ int RequestQueue::Pop(size_t *pidx,size_t *poff,size_t *plen)

从 PSLICE 队列中 pop 出一个 slice，将此 slice 的 index, offset, length 赋给*pidx, *poff, *plen。

◆ int RequestQueue::Peek(size_t *pidx,size_t *poff,size_t *plen) const

若 PSLICE 队列的头 rq_head 存在，则将它的信息（index, offset, length）放入*pidx, *poff, *plen 中。

◆ int RequestQueue::CreateWithIdx(size_t idx)

此函数建立客户端对 peer 的索取队列 RequestQueue。它调用 RequestQueue::Add()将第 idx 个 piece 中的所有 slice 放入 PSLICE rq_head 链表中。

◆ size_t RequestQueue::NSlices(size_t idx) const
计算并返回第 idx 个 piece 中的 slice 个数。

◆ size_t RequestQueue::Slice_Length(size_t idx,size_t sidx) const
计算并返回的 idx 个 piece 中第 sidx 个 slice 的长度。

4.7.2 class PendingQueue

类 PendingQueue 的实例为 PENDINGQUEUE。客户端维护一条 PENDINGQUEUE，每当客户端被 peer 暂时 choke 时，均将准备向 peer 索取的 slice 放入 PENDINGQUEUE，以便 unchoke 后再次利用。

4.7.2.1 变量

➢ PSLICE pending_array[PENDING_QUEUE_SIZE]
存放 PSLICE 队列头 slice 的数组。

➢ size_t pq_count
数组中成员个数。

4.7.2.2 函数

◆ void PendingQueue::Empty()
调用_empty_slice_list()清除 pending_array[]。

◆ int PendingQueue::Pending(RequestQueue *prq)
当 peer 发来 choke 信息时，需要将正在向 peer 请求的 slice 放入 PENDINGQUEUE 中以备 peer unchoke 客户端时再取出。此函数将 prg 放入 PendingQueue 中的 pending_array[]中。实际上只是将 prg 的 rq_head（头 slice，即 prq->GetHead()）放入 pending_array[]中。由于 rq_head 是 PSLICE 链表的头，将 rq_head 放入 pending_array[]中可以保存整个链表。

◆ int PendingQueue::ReAssign(RequestQueue *prq, BitField &bf)
此函数检查 peer 是否有在 PENDINGQUEUE 的 pending_array[]中存放的 piece。注意函数调用的背景：当客户端对 peer 的索取队列 RequestQueue 为空时，函数被调用。此时函数遍历 pending_array[]，若有一个 piece 也在 peer 的 piece 位图 bf 中被置为 1，则调用 RequestQueue::SetHead()将此 piece 设为 peer 的 RequestQueue 的头 rq_head。

◆ int PendingQueue::Exist(size_t idx)

查看第 idx 个 slice 是否在 pending_array[]中。若存在则返回 1，否则返回 0。

◆ int PendingQueue::Delete(size_t idx)
若第 idx 个 piece 存在于 pending_array[i]中，则调用_empty_slice_list()将 pending_array[i]所代表的 PSLICE 链表清除。

◆ int PendingQueue::DeleteSlice(size_t idx, size_t off, size_t len)
将在第 idx 个 piece 中的偏移地址为 off 长度为 len 的 slice 从 PENDINGQUEUE 中移除。

4.7.2.3 全局函数

◆ static void _empty_slice_list(PSLICE *ps_head)
清除以 ps_head 为头的 PSLICE 链表中的所有成员。

4.8 btstream.h

4.8.1 class btStream

类 btStream 起进行 socket 通信和数据发送的作用。之所以叫 btStream，想必是因为 BT 协议是基于 TCP(SOCK_STREAM)而非 UDP(SOCK_DGRAM)吧。

4.8.1.1 变量

➢ SOCKET sock
每一个 peer 都有一条 btStream，客户端与 peer 通信的 socket 便存放在 btStream 中。

➢ BufIo in_buffer
➢ BufIo out_buffer
每一条 btStream 维护两段缓冲：输入缓冲区和输出缓冲区。Peer 给客户端发来的握手信息，数据等都暂存在输入缓冲区；客户端给 peer 发送的握手信息，数据等都暂存在输出缓冲区

4.8.1.2 函数

◆ void btStream::Close()
关闭 sock，并设置 sock 为 INVALID_SOCKET，最后将输入和输出缓冲区关闭（BufIo::Close()）。

◆ ssize_t btStream::PickMessage()
函数调用 BufIo::PickUp()重新整理接收缓冲区。

◆ ssize_t btStream::Feed()
函数调用 BufIo::FeedIn()将 btStream::sock 上的数据接收。

◆ int btStream::HaveMessage()
检查 in_buffer 中是否有消息到来。若有则返回 1，若无则返回 0。

注意函数中有一段检查 in_buffer 中消息是否为合法长度的语句：
if((cfg_max_slice_size + H_PIECE_LEN + 4) < r) return -1;

其中 cfg_max_slice_size = 65536, H_PIECE_LEN = 9。

这句话是针对消息最长时的特殊情况设置的：
piece:<len = 0009 + X><id = 7><index><begin><block>

上面的消息为 piece 消息，即 peer 传给客户端的一个 block（即程序中的 slice）数据。其中 index 是以 0 开始的 piece 索引，begin 是以 0 开始的 block 在第 index 个 piece 中的索引，block 为数据，X 为 block 的长度。各项的长度为：<block>的最大长度为 cfg_max_slice_size； <id>为 1，<index>和<begin>为 4，这三项加起来为 H_PIECE_LEN；<len>为 4；所以最大长度为：cfg_max_slice_size + H_PIECE_LEN + 4。若 r 大于这个长度，则表明出错了。

◆ ssize_t btStream::Send_Keepalive()
BT 协议规定 keepalive 信息每隔 2 分钟发送一次。Keepalive 信息是没有内容的空报文。

◆ ssize_t btStream::Send_State(unsigned char state)
向 peer 发送客户端对它的操作信息（M_CHOKE，M_UNCHOKE，M_INTERESTED，M_NOT_INTERESTED）。这四种信息的格式如下：
<四位的长度><索引号>

四种信息的长度均为 1，索引从 0 至 3。
M_CHOKE： ”00010”
M_UNCHOKE： ”00011”
M_INTERESTED： ”00012”
M_NOT_INTERESTED： ”00013”

◆ ssize_t btStream::Send_Have(size_t idx)
向 peer 发送某个 piece 的 have 消息，信息格式如下：
<len=0005><id=4><piece index>

当客户端成功拥有某个 piece 后会调用此函数向 peer 发送 have 消息。

◆ ssize_t btStream::Send_Piece(size_t idx,size_t off,char *piece_buf,size_t len)
向 peer 发送某个 slice 的 piece 消息，信息格式如下：
<len=0009+X><id=7><index><begin><block>

Index(idx)表示 piece 的索引号，begin(off)表示 slice 在 piece 中的开始位置（偏移量），block 表示 slice 含有的数据，X(len)表示 slice 的长度。

◆ ssize_t btStream::Send_Bitfield(char *bit_buf,size_t len)
向 peer 发送客户端拥有的数据位图，位图信息的格式如下：
<len=0001+X><id=5><bitfield>

X 是位图的长度。
位图信息应该在与 peer 握手成功后立即发送。如果客户端的位图为空，则不必发送。

◆ ssize_t btStream::Send_Request(size_t idx, size_t off,size_t len)
向 peer 发送某个 slice 的 request 消息，信息格式如下：
<len=0013><id=6><index><begin><length>

Index(idx)表示 piece 的索引号，begin(off)表示 slice 在 piece 中的开始位置（偏移量），length(len)表示 slice 的长度。

◆ ssize_t btStream::Send_Cancel(size_t idx,size_t off,size_t len)
向 peer 发送某个 slice 的 cancel 消息，信息格式如下：
<len=0013><id=8><index><begin><length>

Index(idx)表示 piece 的索引号，begin(off)表示 slice 在 piece 中的开始位置（偏移量），length(len)表示 slice 的长度。

◆ ssize_t btStream::Send_Buffer(char *buf, size_t len)
调用 BufIo::PutFlush()将长为 len 的 buf 发送给 peer。

◆ ssize_t btStream::Flush()
调用 BufIo::FlushOut()将输出缓冲区里的数据发送。

4.9 bufio.h

4.9.1 class BufIo

类 BufIo 用于存储缓冲区的数据，并提供了一系列对这些数据进行操作的函数。

4.9.1.1 变量

- char *b

每一个 peer 的发送或接收缓冲区。b 指向缓冲区的开头。

- size_t p

缓冲区目前还有 p 字节。

- size_t n

缓冲区 b 的大小，默认为 32×1024(B)，也就是 32K。

- int f_socket_remote_closed

客户端接收来自远方（tracker 服务器或 peer）的数据，当接收完毕后，远方关闭连接，此时客户端设置 f_socket_remote_closed 为 1。

4.9.1.2 函数

◆ ssize_t BufIo::_realloc_buffer()

将当前缓冲区容量增加 INCREAST_SIZ（32768，32KB）。即 n += INCREAST_SIZ。但是 n 不能超过 MAX_BUF_SIZ（135168，132KB）

若需要发送的数据太长，程序将调用此程序扩大缓冲区。

◆ ssize_t BufIo::_SEND(SOCKET sk, char *buf, size_t len)

函数调用系统调用 send()将缓冲区中的数据发送出去。返回仍存在于缓冲区的字节数。

◆ ssize_t BufIo::_RECV(SOCKET sk, char *buf,size_t len)

此函数将在 sk 上接收的数据放入 buf 中。其中，buf 可以容纳 len 长的数据。

返回值：接收数据失败时返回-1。若是因为阻塞而失败，则返回 0。若数据被全部接收，则返回接收的数据的长度，并设置 f_socket_remote_closed 为 1，表明数据接收完毕，对方关闭连接。

◆ void BufIo::Close()

将缓冲区 delete 掉，防止内存泄漏。

◆ ssize_t BufIo::PickUp(size_t len)

整理 BufIo::b 中的数据并重置 BufIo::p 的位置。一般来说，当客户端将 len 个数据接收到以后，需要将 p 往前移动 len 个字节长度以便在 b 中留出空间接收新的数据，此时程序便会调用 PickUp()。

◆ ssize_t BufIo::FeedIn(SOCKET sk)

函数调用_RECV()接收 sk 上的数据，将其放入以 b+p 开始的缓冲区内。

返回值：若接收失败，则返回-1；若远方连接被关闭，则返回-2；否则返回接收到的数据长度。一般来说，函数会将 sk 上的数据全部接收，此时连接关闭，函数返回-2。

◆ ssize_t BufIo::Put(SOCKET sk, char *buf,size_t len)

若缓冲区的空闲区不足以容纳长为 len 的 buf，则调用 BufIo::FlushOut()将现有缓冲区的数据发送（只调用了一次，未必全部发送完）。若仍不能容纳 buf，则调用 BufIo::_realloc_buffer()扩大缓冲区容量，然后将 buf 所指的数据放入缓冲区内。成功则返回 0，失败返回一个负数。

◆ ssize_t BufIo::PutFlush(SOCKET sk, char *buf,size_t len)

若缓冲区的空闲区不足以容纳长为 len 的 buf，则调用 BufIo::FlushOut()将现有缓冲区的数据发送（只调用了一次，未必全部发送完）。若仍不能容纳 buf，则调用 BufIo::_realloc_buffer()扩大缓冲区容量，然后调用 BufIo::FlushOut()将数据发送。最后返回仍存在于缓冲区的字节数。

PutFlush()比 Put()多做的一点是 PutFlush()将 buf 所指的数据发送出去了，而 Put()仅仅将它们放入缓冲区内。

◆ ssize_t BufIo::FlushOut(SOCKET sk)

函数调用 BufIo::_SEND()将缓冲区中的数据发送出去。返回仍存在于缓冲区的字节数。

乍一看 PutFlush()和 FlushOut()两个函数名字差不多，作用也差不多。但深究起来，PutFlush()有一个在缓冲区中为要发送的数据找个位置的过程，即重在”Put”；而 FlushOut()则是专心地将缓冲区数据发送出去，即重在”Out”。

4.10 connect_nonb.h

◆ int connect_nonb(SOCKET sk,struct sockaddr* psa)

通过 sk 向 psa 连接。返回值很重要：-1 代表连接失败；-2 代表连接正在进行;0 代表连接成功。注意，源程序中的注释：// >0 连接已成功是不对的。

4.11 httpencode.h

◆ char* Http_url_encode(char *s,char *b,size_t n)

将长度为 n 的字符串 b 以 RFC1738 标准编码，返回已编码的字符串并将其放入 s 中。

RFC1738 是一种将除符号 0-9, a-z, A-Z, $-_.+!*'(),外的所有数字编码成一种 16 进制表示的可阅读的格式"%nn"。例如 ANSI 编码的十六进制数字 a 代表换行’\n’，若使用 printf 打印则为一空行，经过 RFC1738 编码后便为”%0a”。而字符’a’在 ANSI 码表中对应的数字为 0x61，经过 RFC1738 编码后便为”%61”。

由于编码后单个字符变为以’%’开头的 3 个字符表示，再加上 s 最后要有一个字符串结束标志’\0’，所以 s 的长度是 b 的长度的 3 倍加 1。例如 b 长 20，则 s 长 61。

◆ int Http_url_analyse(char *url,char *host,int *port,char *path)
此函数用于解析 tracker 服务器的地址信息。

假设传入 url = “http://192.168.1.111:6969/announce", 则函数会设置 host = “192.168.1.111”, port = “6969”, path = “/announce”。

◆ size_t Http_split(char *b,size_t n,char **pd,size_t *dlen)
此函数分离 HTTP 报文的首部和主体。调用函数时将长为 n 的报文放在 b 中，则函数会将报文主体放在 pd 中，长度放在*dlen 中，并返回头部长度。

程序中函数 Http_split()只被函数 btTracker::CheckReponse()调用过一次。CheckReponse()将 tracker 服务器的应答信息传给 Http_split()，例如：
b 的内容为：
HTTP/1.1 200 OK
Connection: Close
Content-Length: 151
Content-Type: text/plain

d8:intervali1800e5:peersld2:ip9:127.0.0.17:peer id20:AzureusqwertyuiopasdUDP04:porti25292e ed2:ip9:127.0.0.17:peer id20:BSs:H1‰+ltUDP04:porti25292eeee

函数解析后会将*pd 赋为：
d8:intervali1800e5:peersld2:ip9:127.0.0.17:peer id20:AzureusqwertyuiopasdUDP04:porti25292e ed2:ip9:127.0.0.17:peer id20:BSs:H1‰+ltUDP04:porti25292eeee

即 HTTP 报文主体。

dlen 为 151，即*pd 的长度。

返回值为 78，报文头部的长度。

◆ int Http_reponse_code(char *b,size_t n)
解析 HTTP 报文的状态码。常见状态码如下：

HTTP/1.1 200 OK	正常
HTTP/1.1 301 Moved Permanent	网站永久重定向
HTTP/1.1 302 Moved Temporarily	网站暂时重定向
HTTP/1.1 403 Forbidden	“您无权查看该网页 您可能没有权限用您提供的凭据查看此目录或网页。”
HTTP/1.1 404 Not Found	“浏览器提示页面不存在或者已经删除”

函数返回状态码，例如上例中的 200，301，302 等。

◆ int Http_get_header(char *b,int n,char *header,char *v)
函数分析 http 报文的头，找出 header 中表示的数据，存储在 v 中。

当 tracker 服务器被重定向到别的地址时，此函数会被调用：
Http_get_header(m_reponse_buffer.BasePointer(), hlen, "Location", redirect)

函数会分析 http 报文，找出”Location: http://redirect_usl”一项，然后把”http://redirect_usl”放入 redirect 中。

4.12 iplist.h

4.12.1 struct _iplist

结构_iplist 被实现为以下形式：
```
typedef struct _iplist{
   struct sockaddr_in address;
   struct _iplist *next;
}IPLIST;
```

注意 IPLIST 和下面的 IpList 的大小写之分，前者是一个结构，后者是一个类。且前者是后者的一个成员。

4.12.2 class IpList

类 Iplist 被实现为 IPQUEUE。当客户端得到 tracker 服务器的 peer 信息时，便调用 IpList::Add() 函数将有关 peer 的 IP 地址加入到 IPLIST 链表中。

4.12.2.1 变量

➢ IPLIST *ipl_head
IPLIST 链表的头。

➢ size_t count
IpList 类中现有 IPLIST 结构的个数。

4.12.2..2 函数

◆ void IpList::_Emtpy()
清空 IPLIST 链表，并将 count 置 0。

◆ int IpList::Add(const struct sockaddr_in *psin)
此函数检查 psin 是否已经在 IPLIST 链表中，若是，则返回-1。否则，便将 psin 加入到 IPLIST 链表头的位置（即 push），并将 count 加 1。

◆ int IpList::Pop(struct sockaddr_in *psin)
将 IPLIST 链表头弹出，有关 IP 信息放入 psin 中，并将 count 减 1。

4.13 peer.h

peer.h 提供了一个表示 peer 状态的结构体 BTSTATUS，两个类：btBasic 与 btPeer，和两个全局函数：set_nl()与 get_nl()。

4.13.1 宏

```
#define P_CONNECTING (unsigned char) 0          // connecting
#define P_HANDSHAKE   (unsigned char) 1          // handshaking
#define P_SUCCESS (unsigned char) 2              // successful
#define P_FAILED (unsigned char) 3               // failed
```

✧ P_CONNECTING
程序调用 connect()与 peer 进行通信，若 connect()返回 EINPROGRESS，则表明通信正在进行中。此时设置 peer 状态为 P_CONNECTING。若以后检测到 peer 为此状态，则必须在 select()时将与 peer 通信的 socket 设入可写文件描述符集中以便完成通信。

✧ P_HANDSHAKE
若程序调用 connect()成功，则设置 peer 状态为 P_HANDSHAKE。若以后检测到 peer 为此状态，则应该调用 btPeer::Send_ShakeInfo()发送握手信息。

✧ P_SUCCESS
当客户端接受了 peer 的握手信息并发送自己的 piece 位图信息成功时，设置 peer 状态为 P_SUCCESS。

✧ P_FAILED
当客户端与 peer 之间的连接关闭时，设置 peer 状态为 P_FAILED。

4.13.2 struct _btstatus

```
typedef struct _btstatus{
   unsigned char remote_choked:1;
   unsigned char remote_interested:1;
   unsigned char local_choked:1;
   unsigned char local_interested:1;

   unsigned char reserved:4;                    /* unused */
}BTSTATUS;
```

- remote_choked

此项置 1 表明 peer 将客户端 choke 了。此时 peer 不会给客户端上传数据，但是可能会从客户端下载数据。

- remote_interested

此项置 1 表明 peer 对客户端有兴趣。此时 peer 准备向客户端索要数据。

- local_choked

此项置 1 表明客户端将 peerchoke 了。此时客户端不会给 peer 上传数据，但是可能会从 peer 下载数据。

- local_interested

此项置 1 表明客户端对 peer 有兴趣。此时客户端准备向 peer 索要数据。

当客户端对 peer 有兴趣而且 peer 没有 choke 客户端时（local_interested = 1, remote_choked = 0），slice 被客户端从 peer 处下载。当 peer 对客户端有兴趣而且客户端没有 choke peer 时（remote_interested = 1, local_choked = 0），slice 被客户端上传给 peer。

4.13.3 class btBasic

由 peer.h 的最后：extern btBasic Self，可以看出类 btBasic 是用来描述客户端自己的。其变量较为简单。

4.13.3.1 变量

- Rate rate_dl

描述客户端或 peer 下载速率的类。

对于客户端，类 rate_dl 描述了当前的即时下载速率。对于 peer，则是 peer 从客户端下载的

速率。

- Rate rate_ul

描述客户端或 peer 上传速率的类。

对于客户端，类 rate_dl 描述了当前的即时上传速率。对于 peer，则是 peer 向客户端上传的速率。

- struct sockaddr_in m_sin

存储自身地址，端口信息的 INET 协议簇地址结构

4.13.3.2 函数

◆ int btBasic::IpEquiv(struct sockaddr_in addr)l

此函数的实现方法很简单：调用 memcmp()进行比较。但其作用却非常重要：tracker 服务器发给客户端的 peer 列表中肯定包含客户端自己的信息。程序需要调用这个函数来查看它获得的 peer 是不是自己。

4.13.4 class btPeer:public btBasic

btPeer 类用来描述 peer 信息，包括当前状态，时间戳等。

4.13.4.1 变量

- time_t m_last_timestamp, m_unchoke_timestamp

m_last_timestamp 是客户端接收到 peer 的消息时的时间。m_unchoke_timestamp 是客户端接收到 peer 的 choke 或 unchoke 消息时的时间。这两个变量在程序中只是记录一下时间，不参与计算，所以意义不大。

- unsigned char m_f_keepalive:1

是否需要发送 keepalive 消息的标志位。当 m_f_keepalive 为 0 时，程序会发送 keepalive 消息。

- unsigned char m_status:4

peer 当前的状态（P_CONNETING, P_HANDSHAKE, P_SUCCESS, P_FAILED）。

- unsigned char m_reserved:3

程序保留项。

- BTSTATUS m_state

Peer 的状态信息，详见 BTSTATUS 结构体。

- size_t m_cached_idx

若 peer 发来一条 HAVE 信息说明某个这个 peer 已经拥有某个 piece，则将 m_cached_idx 设为这个 piece 的索引。以后如果需要从这个 peer 索取 piece 的话，便先查看 m_cached_idx 可不可以获得，可以获得的话就省去了寻找可从 peer 下载的 piece 的麻烦。

- size_t m_err_count

每当客户端与 peer 的通信出现错误（各种各样的错误都算在内，如接收错误，发送错误等）时，m_err_count 便会加 1，当 m_err_count 超过 64 时，程序便会关闭与 peer 的连接（注意这只是充分条件，不是必要条件）。

- int m_standby

peer 是否与客户端有交互。若 peer 对客户端有兴趣或 peer 发来了 HAVE 消息表明 peer 拥有某个 piece，则 m_standby 被置为 0；当在 endgame 模式下客户端不需要向 peer 索取数据时 m_standby 会被置为 1。客户端程序与 peer 进行通信前会检查这个标志，若为 1 则不会与 peer 通信。

- BitField bitfield

Peer 的 piece 位图。Peer 发来位图消息时会被全部更新。每当客户端成功获得 peer 的一个 piece 后 bitfield 会被局部更新。

- btStream stream

详见 btStream 类。

- RequestQueue request_q

客户端准备从 peer 下载的 slice 队列。当队列由满变空时表明一个 piece 已被下载。

- RequestQueue reponse_q

客户端准备向 peer 上传的 slice 队列。当队列由满变空时表明一个 piece 已被上传。

4.13.4.2 函数

◆ int btPeer::PieceDeliver(size_t mlen)

当客户端收到 peer 的数据时会调用此函数。此函数虽然名叫“PieceDeliver”，实际上是处理了 slice 大小的数据，只不过接收了”Piece”消息罢了。

```
{
  size_t idx,off,len;
  char *msgbuf = stream.in_buffer.BasePointer();

  idx = get_nl(msgbuf + 5);//idx 为 piece 的索引。
```

```
  off = get_nl(msgbuf + 9);//off 为 slice 在 piece 中的偏移。
  len = mlen - 9;//len 为 slice 的长度。

  if( request_q.Remove(idx,off,len) < 0 ){//已经获得 slice，所以将 slice 从索取队列中移除。
     m_err_count++;
     if(arg_verbose) fprintf(stderr, "err: %p (%d) Bad remove\n",
        this, m_err_count);
     return 0;
  }

  if(BTCONTENT.WriteSlice((char*)(msgbuf + 13),idx,off,len) < 0){
     return 0;//将 slice 数据写入缓冲区内。
  }

  Self.StartDLTimer();
  Self.DataRecved(len);//更新接收到的数据量，为计算速率做准备。
  DataRecved(len);

  // Check for & cancel requests for this slice from other peers in initial
  // and endgame modes.
  if( BTCONTENT.pBF->Count() < 2 ||
        WORLD.Pieces_I_Can_Get() - BTCONTENT.pBF->Count() < WORLD.TotalPeers() ){
     WORLD.CancelSlice(idx, off, len);//向其它 peer 发送此 slice 的 cancel 消息。
     PENDINGQUEUE.DeleteSlice(idx, off, len);//将此 slice 从 PENDINGQUEUE 中移除。
  }
  /* if piece download complete. */
  return request_q.IsEmpty() ? ReportComplete(idx) : 0;//若 slice 所在的 pice 下载完毕，则将所有 peer 发送 complete 消息。
}
```

◆ int ReportComplete(size_t idx)

当第 idx 个 piece 被下载完毕后，此函数被调用。接下来函数判断 piece 是否真被下载完毕，若是则调用 PeerList::Tell_World_I_Have()发 HAVE 消息给所有 peer，然后将此 piece 从 PENDINGQUEUE 中去除，最后调用 PeerList::CheckInterest()给 peer 发送客户端对其有兴趣与否的消息；若 piece 没有下载完毕，函数被错误地调用，则将 peer 的 m_err_count 加 1。函数最后还要调用 btPeer::RequestCheck()查看是否可以从 peer 处下载数据。

◆ int btPeer::RequestCheck()

函数检查是否需要向 peer 发送 request 信息，若需要，则将 request_q 赋值。源码分析如下：

```
int btPeer::RequestCheck()
{
  if( BandWidthLimitDown() ) return 0;//当前下载速率太大，需要降低下载速率，故返回，不发送 request 请求。
```

```
  if( BTCONTENT.pBF->IsFull() ){//客户端是种子
    if( bitfield.IsFull() ){ return -1; }//peer 也是种子
    return SetLocal(M_NOT_INTERESTED);//则两者互不相干。
  }

  if( Need_Remote_Data() ){//客户端需要 peer 的数据。
    if(!m_state.local_interested && SetLocal(M_INTERESTED) < 0) return -1;//设置 peer 的状态为 M_INTERESTED，表明客户端准备从 peer 下载数据了。
    if(request_q.IsEmpty() && !m_state.remote_choked){//若索取队列 request_q 为空且客户端没有被 peer choke
      if( RequestPiece() < 0 ) return -1;//则从 peer 处索取数据。
    }
  } else
    if(m_state.local_interested && SetLocal(M_NOT_INTERESTED) < 0) return -1;//客户端不需要 peer 的数据，则设置 peer 的状态为 M_NOT_INTERESTED。

  if(!request_q.IsEmpty()) StartDLTimer();//为计算下载速率做准备。
  else StopDLTimer();
  return 0;
}
```

◆ int btPeer::SendRequest()

调用 btStream::Send_Request()向 peer 发送 request 信息。

◆ int btPeer::CancelRequest(PSLICE ps)

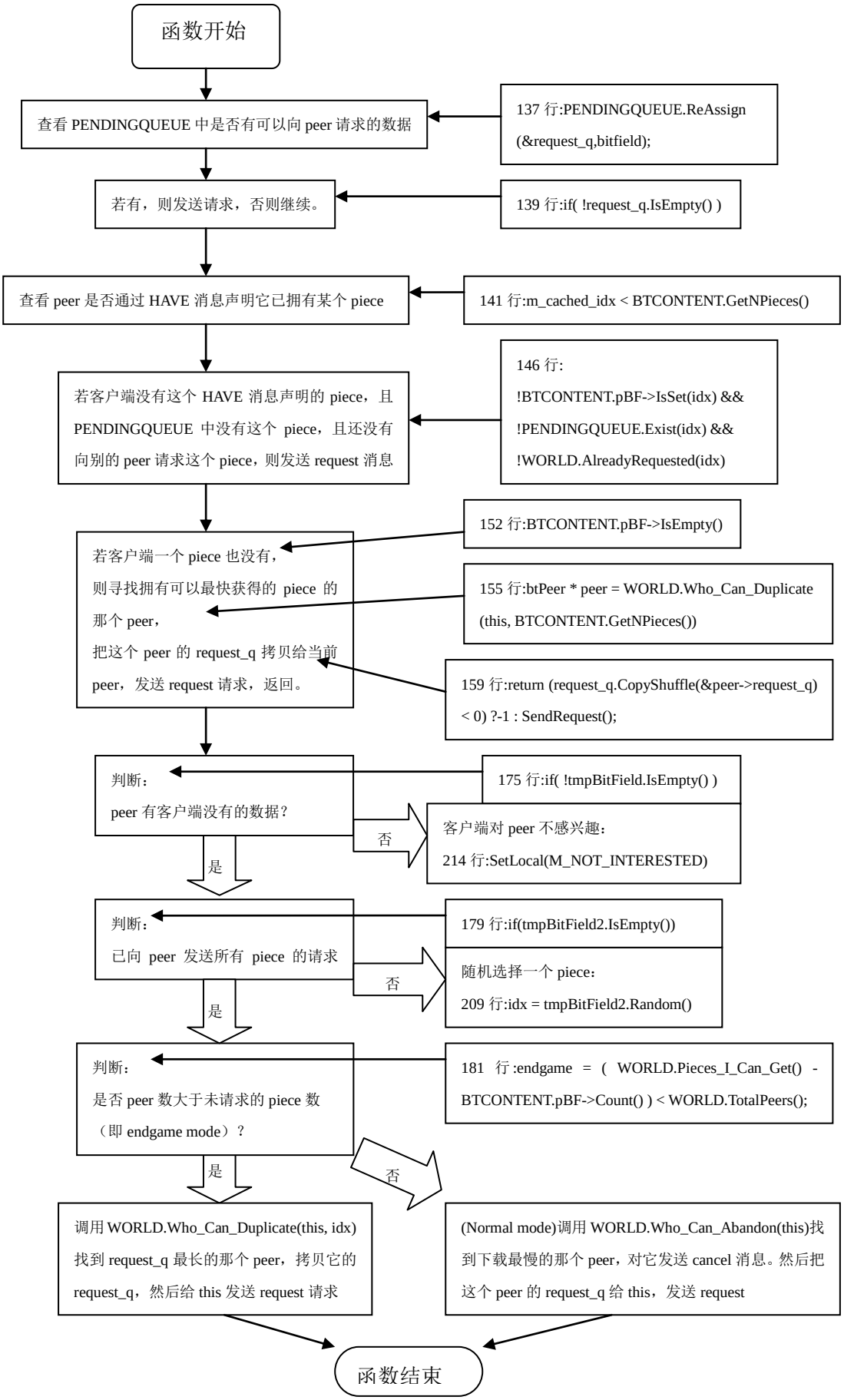
向 peer 发送以 ps 开头的 PSLICE 链表中的所有 slice 的 cancel 消息。当客户端被 peer choke 时或客户端不需要从 peer 处获得 PSLICE 链表中的 slice 时需要调用此函数。

◆ int btPeer::ReponseSlice()

当客户端需要给 peer 上传数据时需要调用此函数。函数首先查看输出缓冲区时候有足够空间存放发送数据，若有则调用 RequestQueue::Pop()从回应队列中弹出一个 slice，然后调用 btContent::ReadSlice() 将这个 slice 读入到 global_piece_buffer 中，最后调用 btStream::Send_Piece()将这个 slice 发送。

◆ int btPeer::RequestPiece()

函数主要检查需要向 peer 请求哪一个 piece。Piece 的选择机制有很多算法，函数进行了从易到难的选择，流程图如下：


函数开始
查看 PENDINGQUEUE 中是否有可以向 peer 请求的数据
137 行:PENDINGQUEUE.ReAssign (&request_q,bitfield);
若有，则发送请求，否则继续。
139 行:if(!request_q.IsEmpty())
查看 peer 是否通过 HAVE 消息声明它已拥有某个 piece
141 行:m_cached_idx < BTCONTENT.GetNPieces()
若客户端没有这个 HAVE 消息声明的 piece，且 PENDINGQUEUE 中没有这个 piece，且还没有向别的 peer 请求这个 piece，则发送 request 消息
146 行: !BTCONTENT.pBF->IsSet(idx) && !PENDINGQUEUE.Exist(idx) && !WORLD.AlreadyRequested(idx)
若客户端一个 piece 也没有，则寻找拥有可以最快获得的 piece 的那个 peer，把这个 peer 的 request_q 拷贝给当前 peer，发送 request 请求，返回。
152 行:BTCONTENT.pBF->IsEmpty()
155 行:btPeer * peer = WORLD.Who_Can_Duplicate (this, BTCONTENT.GetNPieces())
159 行:return (request_q.CopyShuffle(&peer->request_q) < 0) ?-1 : SendRequest();
判断：peer 有客户端没有的数据？
175 行:if(!tmpBitField.IsEmpty())
否
客户端对 peer 不感兴趣：214 行:SetLocal(M_NOT_INTERESTED)
是
判断：已向 peer 发送所有 piece 的请求
179 行:if(tmpBitField2.IsEmpty())
否
随机选择一个 piece：209 行:idx = tmpBitField2.Random()
是
判断：是否 peer 数大于未请求的 piece 数（即 endgame mode）？
181 行 :endgame = (WORLD.Pieces_I_Can_Get() - BTCONTENT.pBF->Count()) < WORLD.TotalPeers();
是
否
调用 WORLD.Who_Can_Duplicate(this, idx) 找到 request_q 最长的那个 peer，拷贝它的 request_q，然后给 this 发送 request 请求
(Normal mode)调用 WORLD.Who_Can_Abandon(this)找到下载最慢的那个 peer，对它发送 cancel 消息。然后把这个 peer 的 request_q 给 this，发送 request
函数结束

图表 4 btPeer::RequestPiece()函数流程图

- int btPeer::MsgDeliver()

此函数根据 peer 发来的储存在 in_buffer 中的信息类型作出相应的处理。具体来说，信息有如下几种类型（详细格式请参看 BitTorrent Specification）：

- M_CHOKE

Peer 将客户端 choke 了。

此时客户端需要将 peer 的 remote_choked 置为 1。若客户端向 peer 请求的 slice 队列不为空的话，需要调用 PENDINGQUEUE.Pending()将 slice 队列放入 PENDINGQUEUE 中，最后调用 CancelRequest()向 peer 发送 cancel 消息。

- M_UNCHOKE

Peer 将客户端 unchoke 了。

此时客户端设置 remote_choked 为 0。若客户端向 peer 请求的 slice 队列不为空，则调用 SendRequest()发送 request 消息，否则调用 RequestCheck()将 request_q 赋值。

- M_INTERESTED

peer 对客户端有兴趣。

此时 peer 准备向客户端索取数据了。

- M_NOT_INTERESTED

peer 对客户端没有兴趣。

此时 peer 不会要求客户端给它上传数据。客户端会将针对 peer 的上传计时器停掉，并清空 peer 的回应数据队列 reponse_q。

- M_HAVE

Peer 声明它已获得某个 piece。

此时客户端会更新 peer 的 piece 位图，然后查看自己是否具有这个 piece，若自己没有的话则将 m_cached_idx 设为这个 piece，这样下次向这个 peer 索要数据时直接索要 m_cached_idx 就可以了，省去了再次计算应索要哪个 piece 的麻烦。同时客户端还设置 m_standby 为 0，表明这个 peer 与自己有交互。

- M_REQUEST

Peer 向客户端请求某个或某几个 slice。

客户端首先会调用 reponse_q.IsValidRequest()检查此 slice 是否有效，若有效则调用 reponse_q.Add()将此 slice 加入到客户端的回应队列中。

- M_PIECE

函数调用 PieceDeliver()处理接收到的 slice 数据（消息的名字叫 piece 消息，其实接收到的是 slice 数据）。

- M_BITFIELD

Peer 向客户端发来 bitfield 信息说明自己的 piece 位图。

函数调用 BitField::SetReferBuffer()将 peer 的 piece 位图拷贝到本地。

- M_CANCEL

Peer 向客户端发来 cancel 信息表明自己不再需要某个 slice 了。

客户端调用 reponse_q.Remove(idx,off,len)将准备响应 peer 的 slice 从 reponse_q 中移除。

◆ int btPeer::CouldReponseSlice()

若 peer 没有被客户端 choke，且输出缓冲区内有空间存放客户端给 peer 的上传数据，则函数返回 1 表明可以给 peer 回应数据。

◆ int BandWidthLimit()

程序只给出了声明，没有给出定义。

◆ int BandWidthLimitUp();

判断客户端当前上传速率是否大于限制上传速率。

◆ int BandWidthLimitDown();

判断客户端当前下载速率是否小于下载限制速率。

◆ int btPeer::RecvModule()

程序中主管客户端接收 peer 数据的函数。函数首先调用 btStream::HaveMessage()查看是否有消息到达，若有则调用 btPeer::MsgDeliver()处理到达的消息，然后调用 btStream::PickMessage()重新整理接收缓冲区，再次调用 btStream::HaveMessage()查看消息，如此循环直至接收缓冲区内不再有消息。

◆ int btPeer::SendModule()

客户端给 peer 上传数据时调用的发送模块，结构比 RecvModule()简单许多。函数首先判断是否可以 peer 上传数据，若可以则调用 btPeer::ReponseSlice()回复数据，最后还不失时机地调用 btPeer::RequestCheck()从 peer 下载数据。

◆ int btPeer::SetLocal(unsigned char s)

此函数将根据 s 的值将 peer 的 BTSTATUS 设置好，然后调用 btStream::Send_State()向 peer 发送对应的消息（M_CHOKE，M_UNCHOKE，M_INTERESTED，M_NOT_INTERESTED）。

◆ int btPeer::CancelSliceRequest(size_t idx, size_t off, size_t len)

函数查看第 idx 个 piece 中偏移为 off，长度为 len 的 slice 是否在 peer 的 request_q 中，如果

在，则调用 request_q.Remove()将其移除，并向 peer 发送 cancel 信息。

◆ int btPeer::NeedWrite()

判断是否需要把 peer 的 socket 设置在可写文件描述符集中。函数的判断条件比较多，所以分析源码：

```
{
  int yn = 0;
  if( stream.out_buffer.Count() || //发送缓冲区不为空，此时需要将数据发送。
       (!reponse_q.IsEmpty() && CouldReponseSlice() &&！ BandWidthLimitUp())
//客户端回应队列不为空，并且可以回应数据，并且上传速率没有超限。
 ||
       ( !m_state.remote_choked && request_q.IsEmpty()//客户端没有 chokepeer 并且 peer 的请求队列已空。
                && m_state.local_interested// peer 对客户端有兴趣
                && !BandWidthLimitDown() && !m_standby ) //客户端下载速率超过限制速率，并且 beer 与客户端有交互
||
       P_CONNECTING == m_status ) //peer 正与客户端通信
     yn = 1;
  return yn;
}
```

◆ int btPeer::NeedRead()

若客户端当前的下载速率比较小，可以申请更多的数据以提高下载速率，并且向某个 peer 的请求队列已经空了，则函数返回 1。

◆ void btPeer::CloseConnection()

将 peer 的状态设为 P_FAILED，然后关闭与 peer 通信的 stream 类（btStream::Close()）。

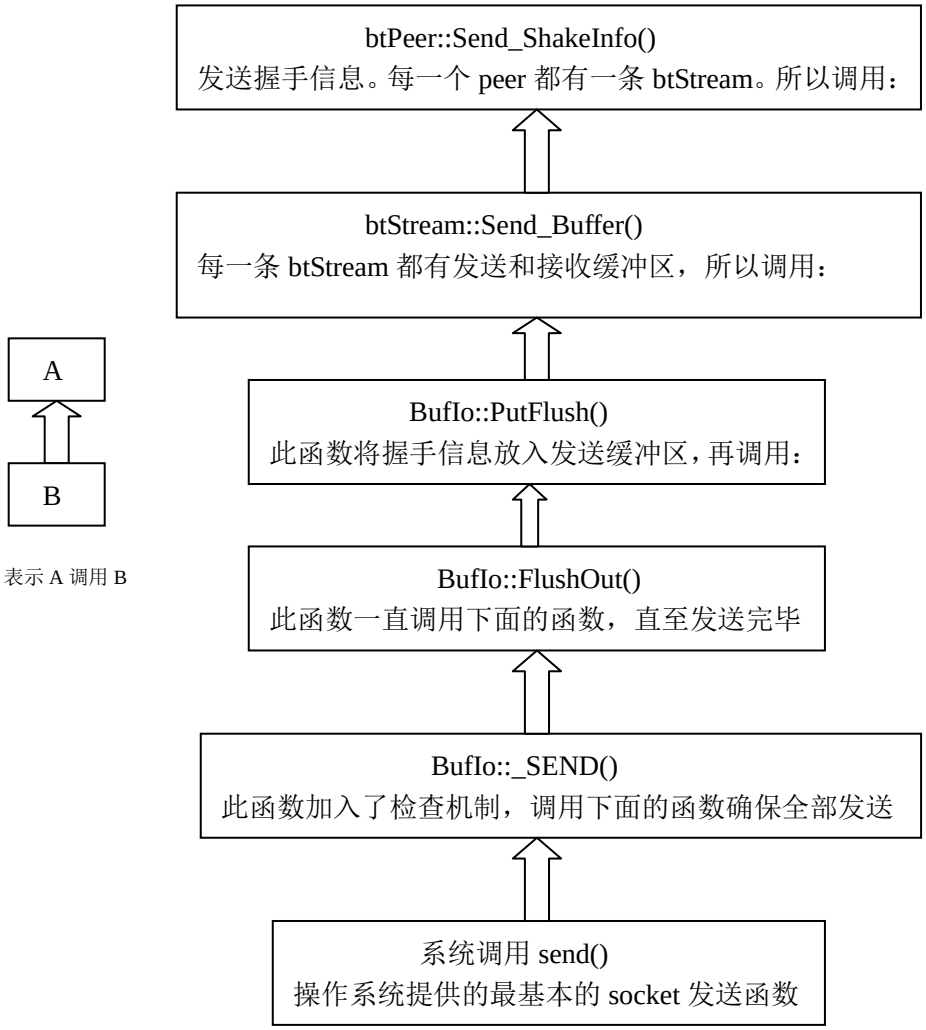
◆ int btPeer::AreYouOK()

发送 keepalive 信息给 peer（Send_Keepalive()），通过返回值判断 peer 的状态。若返回值小于 0，则表明与 peer 的连接关闭了（例如远方的 peer 下载完毕后“下线”了）。

◆ int btPeer::Send_ShakeInfo()

调用 btStream:Send_Buffer()向 peer 发送握手信息。当客户端先 listen()再 accept()一个 peer 的连接后，需要调用此函数以帮助双方建立连接。

此函数嵌套了很深的函数调用。大体结构如下：



图表 5 btPeer::Send_ShakeInfo()函数调用关系图

◆ int btPeer::HandShake()

此函数接收 peer 发送来的 68 字节长的握手信息，然后发送 piece 位图消息给 peer。
函数写得比较多，但如果把详细输出开关关闭（即 if(arg_verbose)不起作用），则逻辑就简单明了了。

由于接收的是 68 字节的握手信息，所以函数一开始便判断：
if(r < 68){…; return 0;}
这种情况很少发生。

随后便是处理握手信息的 20 到 27 位（默认是保留位，全部为 0）。若 peer 的这 8 位不为 0 的话，函数便主动将它们置为 0。

然后比较握手信息的前 47 位，这里面包含了种子文件的 SHA1 HASH 值，如果不相同的话说明客户端和 peer 用的不是同一个种子文件，函数会返回-1。

最后一切无误，函数调用 btStream::Send_Bitfield()将客户端的 piece 位图发送给 peer，然后调用 stream.in_buffer.PickUp(68)重新整理与 peer 通信的输入缓冲区，并设置 peer 的状态为 P_SUCCESS，表示与 peer 成功握手。

◆ int btPeer::Need_Remote_Data()

此函数判断客户端是否可以从 peer 处获得数据：若客户端是种子，则不必获取数据，返回 0；若 peer 是种子，则可以获得数据，返回 1；若两者均不是种子，则调用 Except()将双方的 piece 位图相比较，若 peer 有客户端没有的数据，则可以从 peer 下载。

◆ int btPeer::Need_Local_Data()

若 peer 对客户端有兴趣（interested），并且 peer 不是种子，则 peer 需要下载数据。但 peer 想要的数据客户端未必有，所以进一步判断：如果客户端是种子，那么可以从客户端下载，否则，调用 Except()函数将 peer 和客户端的 piece 位图 bitfield 相比较，若客户端有 peer 没有的数据，则可以从客户端下载。

4.13.4.3 全局函数

◆ void set_nl(char *sto, size_t from)

32 位 CPU 上编译器认为 size_t 长度为 4，char 型长度为 1。此函数将一个 4 字节长的 from 所代表的数字分成 4 个 1 字节长的数放入 sto 中，大数在前，小数在后。例如：
from = 0x1234，则 sto[0] = 1，sto[1] = 2，sto[2] = 3，sto[3] = 4。注意 sto != “1234”，

◆ size_t get_nl(char *sfrom)

此函数是 set_nl()的反操作，即把 sfrom 中的四个 1 字节数组合起来返回 1 个 4 字节的数。

4.14 peerlist.h

CTorrent 程序将所有 peer 信息储存在一个 PEERNODE 链表中。

4.14.1 struct _peernode

➢ btPeer *peer

储存在类 btPeer 中的具体信息。

➢ size_t click

click 是一个判断 peer 活跃程度的标志。每当 peer 的状态为 P_SUCCESS 或可读或可写时便将 click 加 1。这样 click 越大，peer 越活跃，那么在经过 Sort()排序后 peer 在 PEERNODE 列表中的位置便越靠近头部 m_head，然后 UnChokeCheck()检查对 peer 就越有利。

- struct _peernode *next

下一个 peer。

4.14.2 class PeerList

类 PeerList 包含了客户端保存的所有 peer 的信息。这些 peer 以 PEERNODE 结构体链表的形式储存。PeerList 类的变量成员并不是太多，但请注意它包含了一个 PEERNODE 结构体链表，PEERNODE 结构体内有一个 btPeer 类成员，而此成员又由 btBasic 继承而来，其复杂性足以提供描述每一个 peer 的变量和对这些变量进行操作的函数。

4.14.2.1 变量

- SOCKET m_listen_sock

客户端用于绑定和倾听的 socket 描述符。当有其它 peer 发起连接时，会在 m_listen_sock 上侦听到事件，随后调用 Accept()接受。

- PEERNODE *m_head

Peer 链表的头。

- size_t m_peers_count

peer 数目。

注意 PeerList::m_peers_count 和 Tracker::m_peers_count 的不同之处：前者是经过函数 NewPeer()判断后加入到 PEERNODE 链表中的 peer 数目；后者是 tracker 服务器可以提供的 peer 数目，也就是客户端最多可以得到的 peer 数。两者显然是不等的，一个最明显的例子就是前者不包括客户端自己而后者包括。

- size_t m_seeds_count

种子数目。与 tracker 服务器成功通信后会被告知种子数目，当客户端获知 peer 已拥有全部数据开始做种时会更新此项。

- time_t m_unchoke_check_timestamp

执行最后一次 unchoke 检查的时间。Unchoke check 信息每隔 10 秒发送一次。注意此处“执行”的意思仅仅是检查现在的时间（*pnow）和 m_unchoke_check_timestamp 相差是否超过 10 秒。真正“发送”unchoke check 信息会稍后进行。

- time_t m_keepalive_check_timestamp

执行最后一次 keepalive 检查的时间。Keepalive 信息每个 2 分钟发送一次。此处“执行”的意思与上面相同。

- time_t m_last_progress_timestamp

程序中没有用到。

- time_t m_opt_timestamp

发送最后一次 optimistic unchoke check 信息的时间。optimistic unchoke check 每隔 30 秒进行一次。

实际上 m_opt_timestamp 在函数 FillFDSET()中还被用作了一次临时变量：

```
if( f_unchoke_check ) {
   memset(UNCHOKER, 0, (MAX_UNCHOKE + 1) * sizeof(btPeer*));
   if (OPT_INTERVAL <= *pnow - m_opt_timestamp) m_opt_timestamp = 0;
}
```

上述代码的意思是若距上次 optimistic unchoke 检查的时间超过了 OPT_INTEVAL，则设置 m_opt_timestamp 为 0，代表需要进行 optimistic unchoke 检查（在函数 UnChokeCheck()中进行）。此处 m_opt_timestamp 所代表的含义比较迷惑人，请一定注意。

当 optimistic unchoke 检查完毕要发送 optimistic unchoke 信息时，m_opt_timestamp 被设成了当时的时间（即发送的时间，而非执行检查的时间）。

- unsigned char m_live_idx:2

打印表明程序正在工作标志（'-''/''\'）的数组索引。

- unsigned char m_reserved:6

程序保留项。

- Rate m_pre_dlrate, m_pre_ulrate

上一次计算的下载速率和上传速率。计算即时速率时使用。

4.14.2.2 函数

- int PeerList::Accepter()

当 m_listen_port 上有连接请求时，调用此函数接受请求，并调用 PeerList::NewPeer()将新建立的连接加入到 PeerList 中。

- void PeerList::Sort()

根据 PeerList::click 大小降序排列 PeerList 类中的 PEERNODE 链表。

- void PeerList::UnChokeCheck(btPeer* peer, btPeer *peer_array[])

此函数实现了 BT 协议中有关 choke, unchoke, optimistic unchoke 的机制。

为更好地理解函数，先大致说一说 BT 协议的 choke，unchoke check 和 optimistic unchoke check

是怎么工作的。详细信息请参照 BitTorrent Specification 和 Incentives Build Robustness in BitTorrent。

首先要明白 BT 协议的**原则**：一报还一报（从英文 tit-for-tat 翻译过来的）。也就是汉语里常说的“礼尚往来”。下面所有的机制和算法都可以用这个原理解释：

- Choke（每隔 10 秒进行一次）

Choke 的意思是阻塞，但这个阻塞是单向的，即不上传数据给 peer，但仍然可以从 peer 处下载数据。之所以会 choke，原因有二：

第一，本着“保护自己”的思想，防止有些“自私”的 peer 只下载不上传或上传速率很小。这种 peer 会经常陷入被 choke 的尴尬境地。

第二，本着“惠及他人”的思想，客户端 choke 上传速率慢的 peer，用节省下来的带宽多给上传速率快的 peer 用。

BT 协议是一种重在给予的协议，所以上面两条中第一条并不是“严格”执行的，这也就是为什么有些人虽然一点都不上传，但却仍能下载数据的原因。当然，optimistic unchoke 机制也起到了很重要的作用。

- Unchoke（每隔 30 秒进行一次）

BT 协议要求客户端始终 unchoke 4 个上传最快的 peer 以保证客户端自己有较好的下载速率。这里讲的“上传最快”指的是上传给客户端的那部分数据的速率，而不是一个 peer 的整体上传速率。这四个上传最快的 peer 叫 downloaders。之所以把上传最快的 peer 叫做 downloaders，是因为按照一报还一报的原则，上传最快的 peer 对从对方那里下载是最感兴趣的。

Choke 和 unchoke 机制似乎可以保证大家都“有来有往”，“付出最多回报也最多”，但这样却会导致一个经济学里的难题：囚徒困境。

囚徒困境讲了这样一个故事：甲和乙一起犯了罪被抓。他们分别被告知：如果都不招，则判 1 年；如果一个招，则招的人释放，另一个判 10 年；如果都招了，则各判 5 年。于是甲就想：如果乙不招，那么我招，可以不判刑；如果乙招了，那么我招，可以少判 5 年；不管怎样，招供都是最佳选择。而乙恰恰也是这么想的。最后两人都被判 5 年。

显然两人都作出了最明智的选择，最后却得到了一个不是众乐乐（应是各判 1 年）的局面。囚徒困境告诉我们：最符合个体理性的选择，却是集体非理性的。应用到 BT 协议，也是如此：两个 peer 互通有无，不亦乐乎，但最后很有可能会导致全盘皆输。所以，引入了第三种机制 optimistic unchoke。

- Optimistic unchoke（每隔 30 秒进行一次）

Optimistic unchoke 机制不管 peer 的上传速率是不是比 4 个 downloaders 快，均给它们比 downloaders 多 3 倍的机会使 peer 成为 downloaders，然后给 peer 发送 unchoke 信息。而落选的那个 downloaders 只能被 choke 了。这样做有两个好处：

第一，可以发现上传速率比 4 个 downloaders 还快的 peer，把此 peer 加入到 downloaders 中可以改善下载速率。

第二，给上传速率不是那么快的 peer 机会让它们也可以下载数据。这样便打破了囚徒困境，使整个协议可以有效实施。

现在让我们来看看函数的流程：

函数写得比较大，也比较复杂，必须结合上下文来理解。

程序中只有一处调用 UnChokeCheck()函数，在 FillFDSET()中：

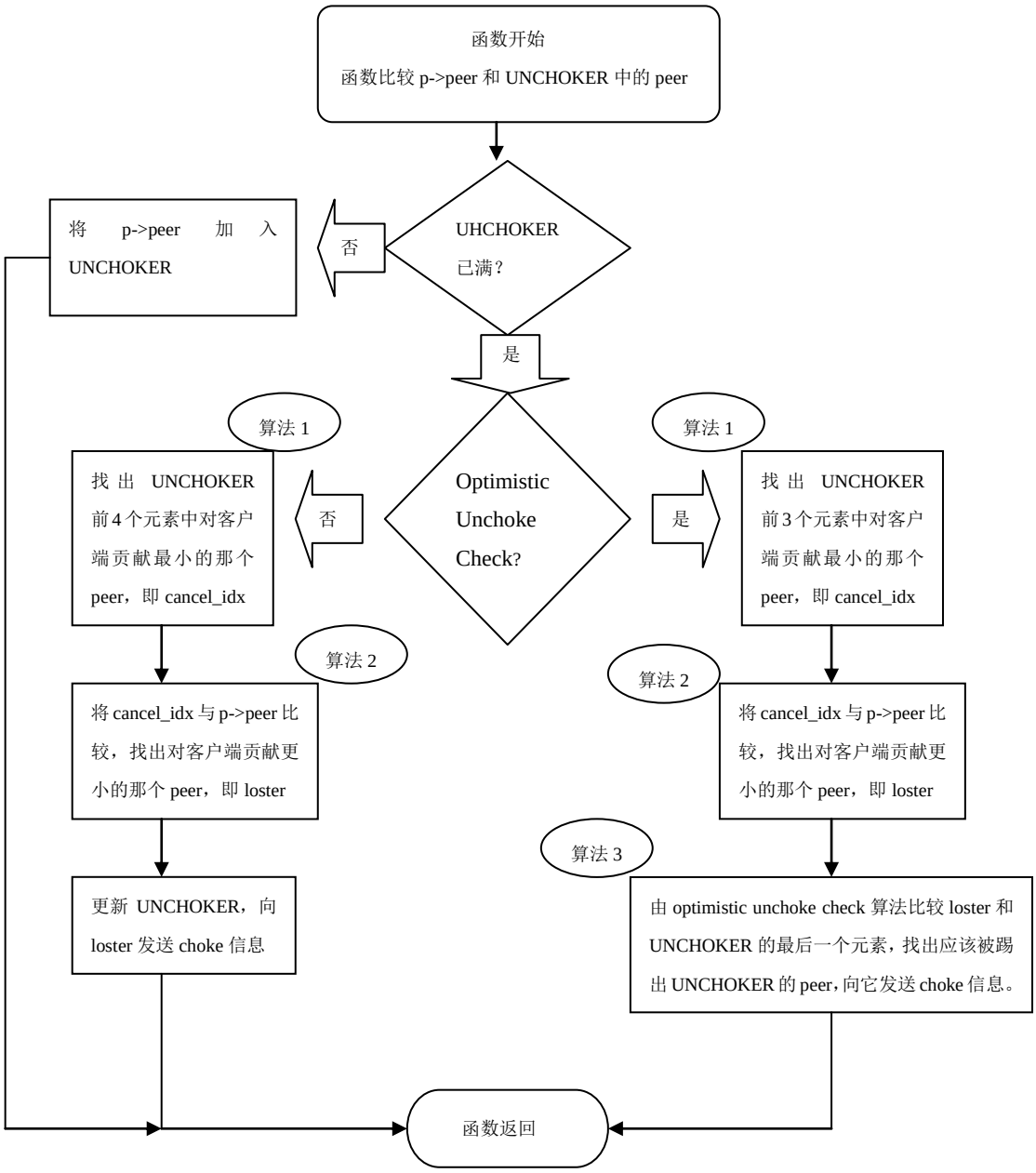
```
for(p = m_head; p;)
{
…
    if( p->peer->Is_Remote_Interested() && p->peer->Need_Local_Data() )
            UnChokeCheck(p->peer, UNCHOKER);
    else if(p->peer->SetLocal(M_CHOKE) < 0)
    {
            p->peer->CloseConnection();
    }
…
p = p->next;
}
```

这段代码揭示了 choke check 检查的一部分算法：如果 peer 对客户端有兴趣并且客户端有 peer 需要的数据，那么就让 peer 进入 unchoke check 检查；否则，发送 choke 消息给 peer。Choke check 检查的另一部分算法机制在 UnChokeCheck()函数体里有表述，我们下面再谈。注意传给函数的参数：

```
        UnChokeCheck(p->peer, UNCHOKER);
```

UNCHOKER 是容量为 4(MAX_UNCHOKER+1)的 btPeer 指针数组，实际上里面放的就是上面说的 4 个 downloaders。UNCHOKER 的最后一个元素：UNCHOKER[MAX_UNCHOKER]是留给 optimistic unchoke check 用的。

函数整体的流程图如下：



图表 6 PeerList::UnChokeCheck()函数流程图

首先说说函数的例外情况：如果 UNCHOKER 还没有装满，那么就把 p->peer 放入 UNCHOKER 中，然后返回。

然后是是否进行 optimistic unchoke check 的判断标准：

如果 m_opt_timestamp 不为为 0，那么就不需要进行 optimistic unchoke check（no_opt = 1，no_opt 代表 no optimistic unchoke check），函数中代码如下：

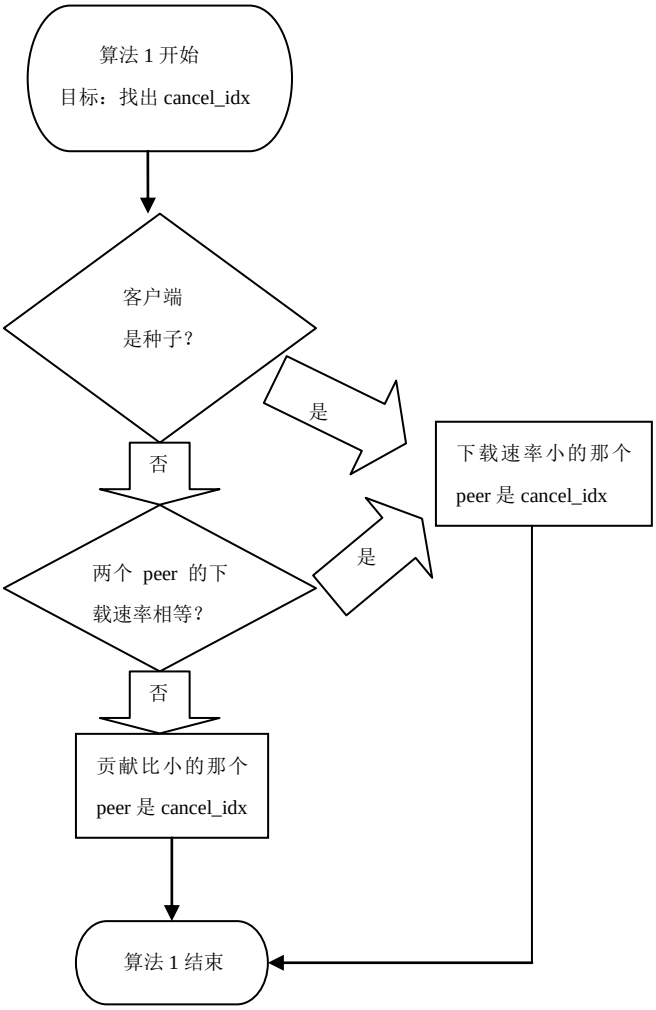
```
if (m_opt_timestamp) no_opt = 1;
```

上面的代码便是 m_opt_timestamp 的“例外”用法，详细说明请见 time_t m_opt_timestamp。

最后便是流程图中三个算法的描述了：

算法 1：找出 UNCHOKER 前 3 或 4 个元素中对客户端贡献最小的那个 peer，即 cancel_idx。关于“贡献更小”的标准，我们以流程图表示，在这之前，为了表述方便，先定义一个名词：贡献比。

一个 peer 的总上传量与总下载量的比称为这个 peer 的贡献比。
贡献比越低，说明这个 peer 上传少，下载多，则对客户端的“贡献更小”。



图表 7 算法 1 流程图

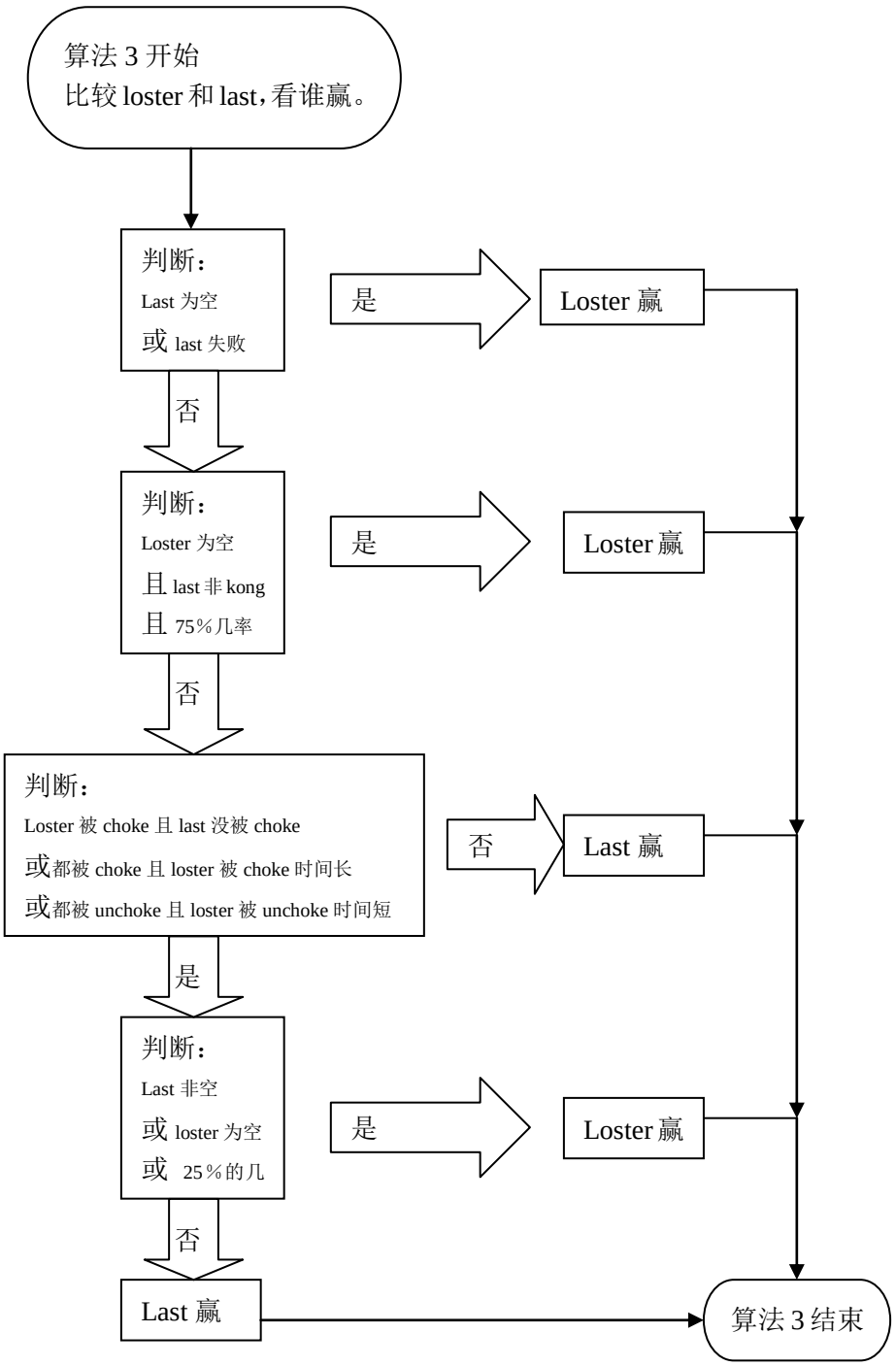
算法 2：将 cancel_idx 与 p->peer 比较，找出对客户端贡献更小的那个 peer，即 loster

算法 2 和算法 1 实际上是一样的，只不过是比较的对象不同而已。

算法 3：由 optimistic unchoke check 算法比较 loster 和 UNCHOKER 的最后一个元素（即 UNCHOKER[MAX_UNCHOKER]），找出应该被踢出 UNCHOKER 的 peer，向它发送 choke

信息。

简单起见，我们以 last 代表 UNCHOKER[MAX_UNCHOKER]。算法 3 即是 last 与 loster 的比较，看结果是谁赢（输的被 choke，赢的进入 UNCHOKER）。



图表 8 算法 3 流程图

流程图中“75％几率”的解释为：

表达式 random()&3 为 1 的几率为 75％。

由于 random()产生一个 0 到 RAND_MAX（2147483647）的数，无论这个数是多少，它与二进制数 00000011 相与后为 1 的几率为 75％（即 01，10，11，00 四个数中有三个数与 11 相与后为 1）。

“25％几率”自然就是取反了：!(random()&3)。

“75％几率”符合 Optimistic unchoke 机制，不管 peer 的上传速率是不是比 4 个 downloaders 快，均给它们比 downloaders 多 3 倍的机会使 peer 成为 downloaders”的说法。

函数找出“输”的 peer 后便向其发送 choke 信息暂时阻塞向 peer 的上传通信。

一般来说，对 peer 进行 choke 和 unchoke 不仅是通过“奖惩分明”的方法“鼓励”peer 多上传以保证 BT 协议优秀的下载性能。BT 客户端还必须要做到 unchoke 某个 peer 后必须还要 choke 另一个 peer 以保证网络通信负载均衡。否则，一个庞大的 BT 通信群体 choke 和 unchoke 不当所带来的网络振荡（“fibrillation”）会让 TCP 协议在底层所保证的拥塞控制荡然无存。

◆ int PeerList::Initial_ListenPort()

此函数初始化倾听套接字。如果 cfg_max_listen_port 端口（2706）被其它程序占用导致使用 bind()无法绑定，则将端口号逐一递减再次绑定，直至成功或抵达 cfg_min_listen_port 端口（2106）为止。

绑定成功后程序调用 listen(m_listen_sock,5)确保倾听队列里可以容纳 5 个完全建立的连接。随后调用 setfd_nonblock(m_listen_sock)设置倾听套接字为非阻塞型。

◆ int PeerList::NewPeer(struct sockaddr_in addr, SOCKET sk)

此函数的主要作用是将地址为 addr 的新 peer 加入到以 m_head 为开头的 PEERNODE 链表中，粗略一点说，就是加入到 PeerList 里。

首先，函数根据 addr 检查此 peer 是不是客户端自己（因为从 tracker 服务器传来的 peer 列表里肯定包含客户端自己），若是则返回-3。

然后，根据 sk 分两种情况处理：

若 sk 是 INVALID_SOCKET，则表明这个 addr 所代表的 peer 是刚从 IP 队列 IPQUEUE 里 Pop()出来的，这时函数以非阻塞方式连接（调用 connect_nonb()）这个 peer，若成功了，便新建一个 peer，然后把这个新 peer 的地址和 socket 设好。

若 sk 不是 INVALID_SOCKET，则在调用函数 NewPeer()之前，肯定调用了系统调用 accept()。accept()返回了一个不是 INVALID_SOCKET 的 sk，接着调用 NewPeer()，新建一个 peer，然后把这个新 peer 的地址和 socket 设好。

最后，向 peer 发送握手信息，把这个新建好的 peer 设为 m_head，加入到 PEERNODE 链表中。

◆ void PeerList::CloseAllConnectionToSeed()

函数调用 btPeer::CloseConnection()关闭和所有 seed 的连接。

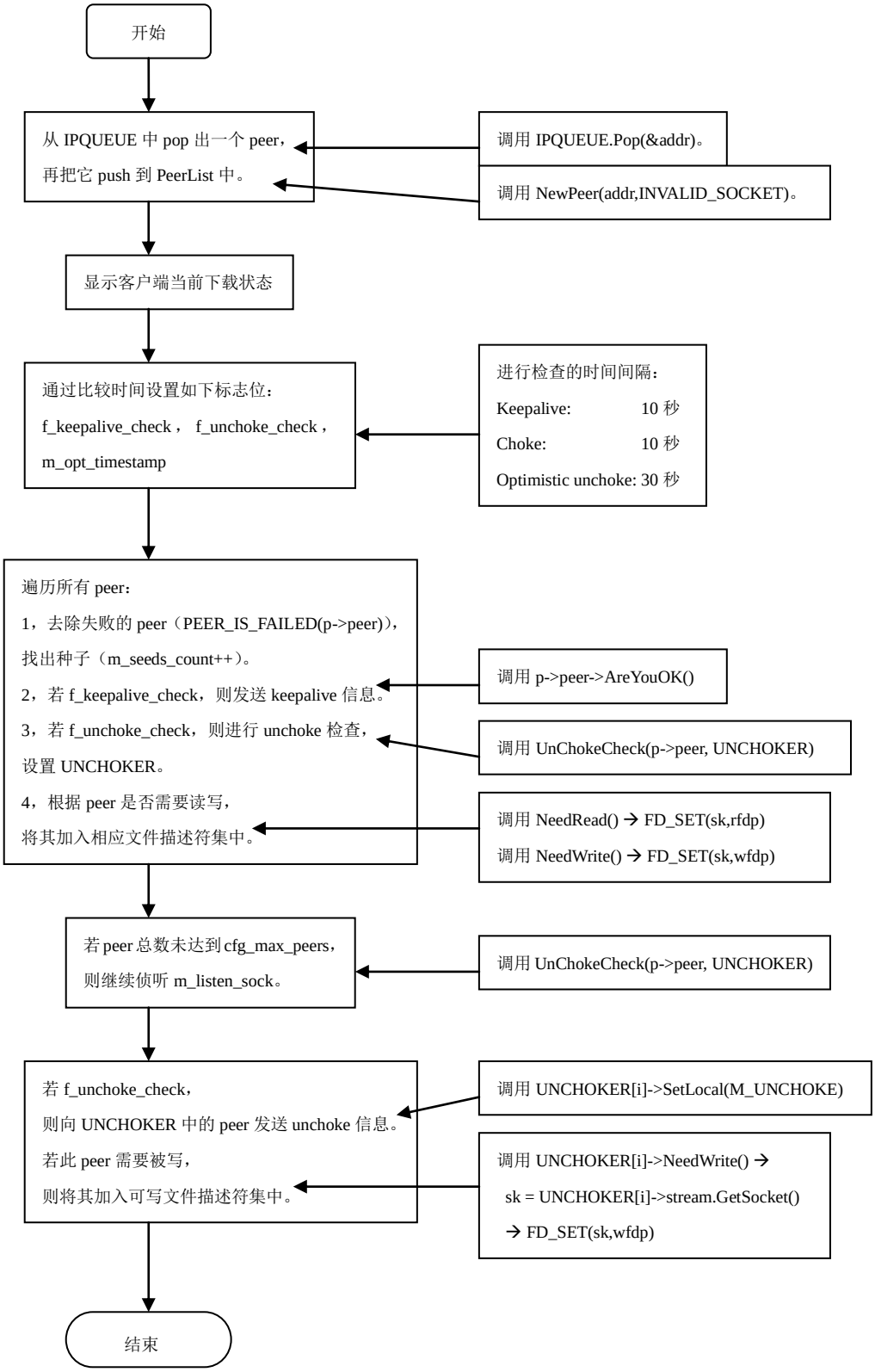
◆ void PeerList::CloseAll()

函数删除 PEERNODE 链表中的所有 peer。

◆ int PeerList::FillFDSET(const time_t *pnow,fd_set *rfdp,fd_set *wfdp)

函数 FillFDSET()主要是遍历在 PeerList 中每一个 peer，向它们发送 keepalive 信息，进行 unchoke 和 optimistic unchoke 检查，并根据 peer 的状态判断是否需要向它们上传或给它们下载数据。可以说，BT 客户端性能好坏与此函数有很大关系。

由于函数较为复杂，在这里只画出它的流程图，具体实现请参照相关调用函数。

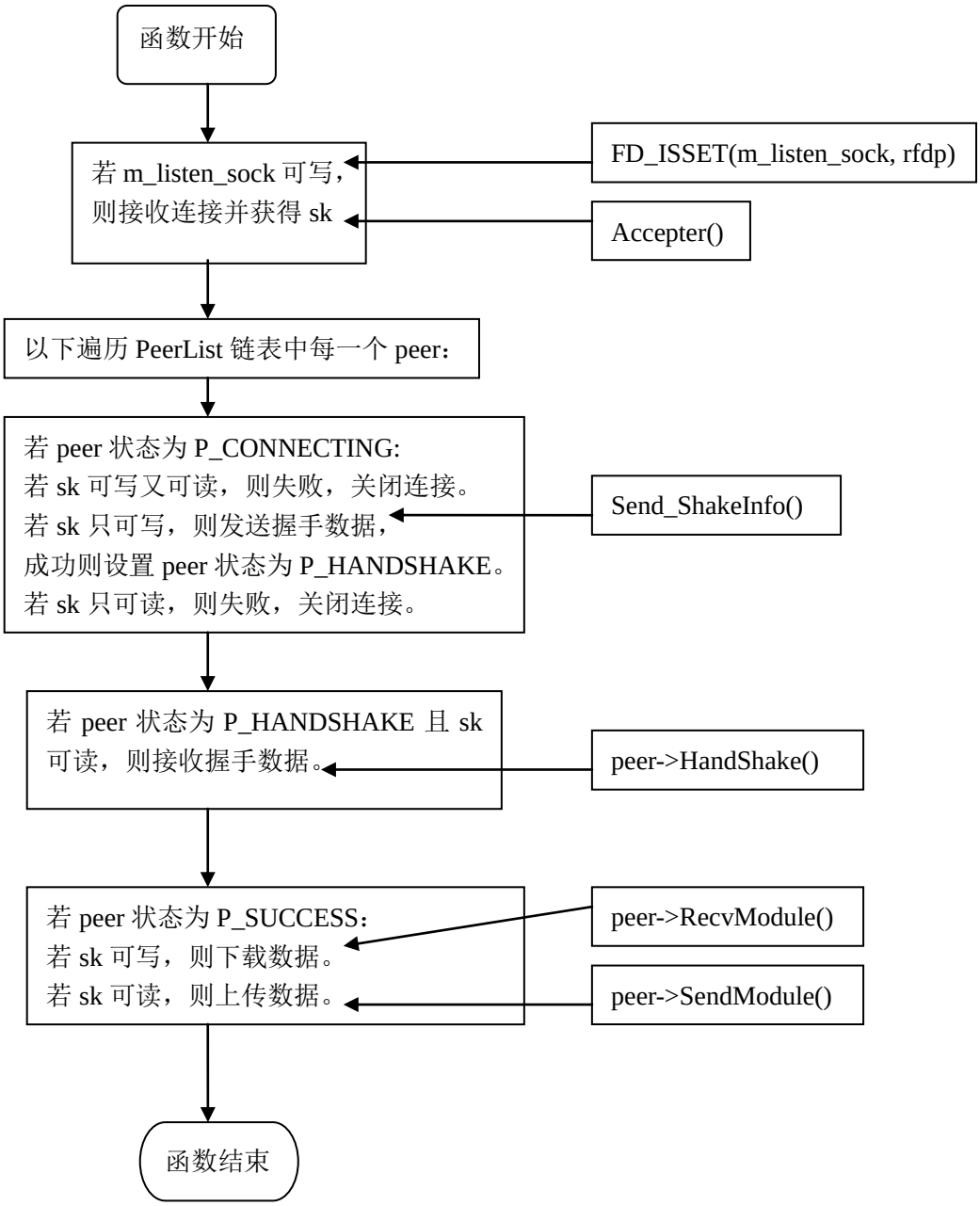


图表 9 PeerList::FillFDSET()函数流程图

◆ void PeerList::AnyPeerReady(fd_set *rfdp, fd_set *wfdp, int *nready)

当 select()函数返回值表明有数据到达后，程序会调用此函数对 peer 进行一系列操作，包括发送握手信息，接收握手信息，给 peer 上传数据，从 peer 下载数据等。

函数流程图如下，sk 代表与 peer 通信的 socket，“可写”代表 sk 在可写文件描述符集中，“可读”代表 sk 在可读文件描述符集中：



图表 10 PeerList::AnyPeerReady()函数流程图

◆ void PeerList::Tell_World_I_Have(size_t idx)

当客户端成功获得一个 piece 后，会调用此函数向每个在 PeerList 中的 peer 发送 HAVE 消息声明自己已经拥有了这个 piece。若客户端是种子，则还会将向 peer 的请求队列 request_q 清空，并设置 peer 的状态为 M_NOT_INTERESTED，表明客户端对 peer 的数据无兴趣。

此函数中客户端向所有的 peer 均发送了 have 消息。实际中，在发送之前，客户端可以先判断一下 peer 是否拥有此 piece，若 peer 已经拥有，则不必发送 have 消息，这样可以减少网络流量。毕竟，每获得一个 piece 便向所有 peer 发送 have 消息对客户端和网络流来说都是不小的负担。

◆ btPeer* PeerList::Who_Can_Abandon(btPeer *proposer)

当程序工作在正常状态（既不处于 initial piece mode 也不处于 endgame mode），且需要向某一个 pere（即 proposer）发送 request 信息，而这个 peer 的请求队列 request_q 又为空时，调用 Who_Can_Abandon()找出下载最慢的那个 peer。程序会在函数体外给这个 peer 发送 cancel 信息，并把这个 peer 的 request_q 拷贝给 proposer。

◆ btPeer* PeerList::Who_Can_Duplicate(btPeer *proposer, size_t idx)

调用此函数的前提条件是客户端向 proposer 的索取队列 request_q 为空。

当客户端一个 piece 也没有时，需要尽快获取一个 piece 以便有数据提供给别的 peer 下载。这种状态称为 initial-piece mode。此时与其新建一个 request_q，不如检查 PeerList 链表中的所有 peer 的 request_q，选取长度最短的那个 request_q，将其拷贝（在函数体外调用 RequestQueue::CopyShuffle()）给当前正在通信的 peer（即 proposer）的 request_q，以便尽快下载数据。

BT 协议允许从不同的 peer 处获得同一个 piece 中的不同的 slice，但是为了程序设计上的方便，CTorrent 采取从一个 peer 获取整个 piece 的方法。所以长度最短的那个 request_q 表明这个 piece 中已经有最多的 slice 被下载下来了，此时复制这个索取队列可以尽快下载完剩下的 slice 以便获得一整个 piece。

当客户端已经下载了很多 piece，还需要得到的 piece 数小于 peer 数时（此时已无法向每一个 peer 请求一个不同的 piece 了，因为 peer 太多而需要的 piece 太少），程序进入 endgame mode。此时可以向许多 peer 请求一个 slice，这样可以加速完成下载从而开始做种。因此函数遍历所有 peer，找出长度最大的那个 request_q，将其拷贝（在函数体外调用 RequestQueue::CopyShuffle()）给当前正在通信的 peer（即 proposer）的 request_q，以便尽快下载数据。

长度最大的那个 request_q 表明 request_q 正在请求的这个 piece 的下载工作“落后”了，此时复制这个索取队列可以从其它 peer 获得帮助尽快下载完这个 piece。

◆ void PeerList::CancelSlice(size_t idx, size_t off, size_t len)

函数遍历 PeerList 链表对每个 peer 调用 request_q.GetRequestIdx()查看第 idx 个 piece 是否在某个 peer 的 request_q 中，若在，则调用 CancelSliceRequest()查看以 off 和 len 为特征的 slice 是否在 request_q 队列中。

◆ void PeerList::CheckBitField(BitField &bf)

此函数遍历 PeerList 中的所有 peer，如果已向某个 piece 请求了某个 piece，则把这个 piece 在 bf 中相应的位设为 0。

函数返回的结果就是把所有已经向 peer 请求的 piece 从 bf 中去除了，bf 中剩下的全部都是没有被请求的 piece。

◆ int PeerList::AlreadyRequested(size_t idx)

此函数遍历 peerlist 中的所有 peer，查看第 idx 个 peer 是否已经在某个 peer 的 request_q 队列里了。

◆ size_t PeerList::Pieces_I_Can_Get()

函数遍历 PeerList 链表中的所有 peer，找出存在于 peer 中的所有 piece 数，调用 BitField::Comb() 计算并返回这个数值，表示客户端可以得到的 piece 数目。

◆ void PeerList::CheckInterest()

函数根据客户端是否需要 peer 的数据来设置 peer 的状态为 M_INTERESTED 或 M_NOT_INTERESTED。

4.15 rate.h

Rate 类提供了两大类变量：数据变量和时间变量，以及一系列根据数据和时间进行速率计算的函数。

4.15.1 变量

➢ time_t m_last_timestamp

调用 StartTimer()开始计时的时间。

➢ time_t m_total_timeused

程序每次调用 StartTimer()和 StopTimer()中间经历的时间的总计时。

➢ u_int64_t m_count_bytes

总共上传或下载的字节数。

➢ u_int64_t m_recent_base

m_recent_base 是函数重置下载速率类 Rate 时 Rate 类拥有的 m_count_bytes 数。通过当前的 m_count_bytes 和以前保存的 m_recent_base 可以调用 Rate::RateMeasure(const Rate &ra_to)计算即时速率。

➢ size_t n_samples

m_timestamp_sample[]中现有的成员个数。

➢ time_t m_timestamp_sample[MAX_SAMPLES]

记录时间戳的数组，和 m_bytes_sample[]对应，用于测量速率。

- u_int64_t m_bytes_sample[MAX_SAMPLES]

记录最近上传或下载数据的字节数的数字，和 m_timestamp_sample[]对应，用于测量速率。

4.15.2 函数

◆ void Rate::StartTimer()

每当有客户端向 peer 上传或从 peer 下载数据时都会调用此函数，为计算总时间和速率做准备。

◆ void Rate::StopTimer()

每当有客户端向 peer 上传或从 peer 下载数据完毕时都会调用此函数，为计算总时间和速率做准备。

◆ void Rate::CountAdd(size_t nbytes)

此函数将 m_count_bytes 更新，并且在 m_timestamp_sample[]和 m_bytes_sample[]中记录下当前的时间和数据量，以便为 RateMeasure()计算速率做准备。

◆ size_t Rate::RateMeasure() const

计算速率。

客户端每次调用 StartTimer()便将当时时间记录在 m_last_timestamp 中。以后每次接收一段数据，便将接收时间和数据量记录到 m_timestamp_sample[]和 m_bytes_sample[]中。计算速率时，只需找出当前时间和 m_last_timestamp 的时间差 timeused，然后找出在这个时间差中下载的所有数据量，两者相除，即得速率。

◆ size_t Rate::RateMeasure(const Rate &ra_to) const

由两个 Rate 类中的数据计算即时下载速率。

4.16 setnonblock.h

◆ int setfd_nonblock(SOCKET socket)

由系统调用 fcntl()设置 socket 通信为非阻塞模式。

4.17 sigint.h

◆ void sig_catch(int sig_no)

sig_catch()的调用环境如下：

signal(SIGINT,sig_catch);

signal(SIGTERM,sig_catch);

当用户按下 Ctrl-C 或 Ctrl-D 中止程序时，系统调用 signal()会捕捉到用户的操作并在程序退出前调用 sig_catch()做一些程序的收尾工作。sig_catch()首先会调用 Tracker.SetStoped()重置客户端与 tracker 服务器的通信，然后调用 sig_catch2()。

◆ static void sig_catch2(int sig_no)

此函数将缓存中的数据写入硬盘，并将 PeerList 列表所占用的内存释放掉，最后向当前进程发送中断或中止信号。

4.18 tracker.h

4.18.1 宏

```
#define T_FREE          0
#define T_CONNECTING    1
#define T_READY         2
#define T_FINISHED      3
```

✧ T_FREE

由于默认与 tracker 服务器通信的时间间隔为 30 分钟，所以服务器大部分时间都处于 T_FREE 状态。此时，如果客户端获得的 peer 数太少，可以提前与 tracher 服务器连接以获取更多的 peer 信息。

✧ T_CONNECTING

当调用 btTracker::Connect()时，服务器状态被设为 T_CONNECTING。此时，需要将 m_sock 放入可写文件描述符集里，以备 Connect()函数写 m_sock 用。

✧ T_READY

当调用 btTracker::SendRequest()向 tracker 服务器发起请求成功时，服务器状态被设为 T_READY。此时，需要将 m_sock 放入可读文件描述符集里，以方便接收服务器的应答信息。

✧ T_FINISHED

当客户端结束做种时，服务器状态被设置为 T_FINISHED。此时程序便可以退出了。

4.18.2 变量

➢ char m_host[MAXHOSTNAMELEN]

tracker 服务器的 ip 地址或域名（例如 192.168.1.111 或 www.***.com）。

- char m_path[MAXPATHLEN]

tracker 服务器 announce 页面的路径（例如”/announce”）。

客户端发给 tracker 服务器的 Get 请求报文信息。

- int m_port

trakcer 服务器提供服务的端口号，一般为 6969。

- struct sockaddr_in m_sin

与 socket 套接字有关的地址结构。

- unsigned char m_status

tracker 服务器当前的状态。

- unsigned char m_f_started, m_f_stoped, m_f_completed

这三个标志位是用来判断向服务器发送何种 event 信息（started, stopped, completed 或空）的。但用法并不像其字面意思所表示的（例如若 m_f_started == 1，就发送 started 信息），具体算法请参照函数 btTracker::SendRequest()。

- unsigned char m_f_pause，m_f_reserved

函数中只定义没有使用。

- time_t m_interval

与 tracker 服务器通信的时间间隔。程序中一般将其初始化为 15（秒）。在实际通信中，tracker 服务器为了减轻负载，一般会通知客户端将此值设为 1800（秒），也就是半个小时。

- time_t m_last_timestamp

最后一次与 tracker 服务器通信的时间。一般在 btTracker::Connect()中将其更新。另外，在 btTracker::Reset()中也会将其更新。

- size_t m_connect_refuse_click

与 tracker 服务器连接失败的计数。每次连接失败后都会重置客户端，连接成功后重新为 0。

- size_t m_ok_click

与 tracker 服务器连接成功的计数。

- size_t m_peers_count

tracker 服务器传来的总共的 peer 数目。注意 tracker::m_peers_count 和 peerlist::m_peers_count 表示的含义不同。前者是整个 bt 通信群中所有的 peer 数，后者是客户端正在通信的 peer 数。

- size_t m_prevpeers

上一次检查 peer 总数时的 peer 数。

- SOCKET m_sock

客户端与服务器进行通信用的 socket。

- BufIo m_reponse_buffer

储存 tracker 服务器回应消息的缓冲区类。

4.18.3 函数

◆ int btTracker:: _IPsin(char *h, int p, struct sockaddr_in *psin)

根据 tracker 服务器的 m_host（h）和 m_port（p）设置 m_sin（psin）的函数。

◆ int btTracker:: _s2sin(char *h,int p,struct sockaddr_in *psin)

根据 tracker 服务器的 m_host（h）和 m_port（p）设置 m_sin（psin）的函数。_s2sin()与_Ipsin()不同的一点是若由 m_host 转换得到的二进制网络地址没有意义，则调用 gethostbyname()再次获得网络地址。

◆ int btTracker::_UpdatePeerList(char *buf,size_t bufsiz)

此函数根据 tracker 服务器发来的 peer 列表更新 m_interval, m_peers_count,等信息，然后调用 IPQUEUE.Add()将 peer 加入 IpList 链表中。

注意函数报文中"complete"，"incomplete"信息更新 m_peers_count，但由于有些 tracker 服务器（取决于服务器程序怎么写）并不发送这些信息，所以 m_peers_count 的数值可能为 0。实际上 CTorrent 程序并不十分关心 tracker::m_peers_count，程序运行时会根据 IPQUEUE 中的 peer 信息更新另一个 peer 计数：PeerList::m_peers_count。

◆ int btTracker::Initial()

函数名字叫 Initial()，好像是对 tracker 服务器进行初始化，但实际上是填充 Get 报文格式和设置本地 IP 地址用的。具体分析如下：

```
int btTracker::Initial()
{
   char ih_buf[20 * 3 + 1],pi_buf[20 * 3 + 1],tmppath[MAXPATHLEN];
//ih 代表 InfoHash，pi 代表 PeerId。两者大小为 61 的原因请见函数 Http_url_encode()。

   if(Http_url_analyse(BTCONTENT.GetAnnounce(),m_host,&m_port,m_path) < 0){
      fprintf(stderr,"error, invalid tracker url format!\n");
      return -1;
   }
//详见函数 Http_url_analyse()分析。
   strcpy(tmppath,m_path);

   if(MAXPATHLEN < snprintf((char*)m_path,MAXPATHLEN,REQ_URL_P1_FMT,
         tmppath,Http_url_encode(ih_buf,(char*)BTCONTENT.GetInfoHash(),20),
```

```
        Http_url_encode(pi_buf,(char*)BTCONTENT.GetPeerId(),20), cfg_listen_port)){
      return -1;
   }
/*
REQ_URL_P1_FMT 是客户端向服务器发送的 Get 请求报文的格式。
m_path 的形式类似于：
GET  /announce?info_hash=%0Bp%A3%40%EB%A9%27%21%F4%19%B7%E5NLu%DAn4XI
&peer_id=%2DCD0102%2D%7C%9E%FD%E1%AF%2C%92X%FE%0C%A5%0E&port=2706
*/
   /* get local ip address */
   // 1st: if behind firewall, this only gets local side
   //如果客户端是在防火墙或局域网内，设置自身 IP（一般为 192.168.*.*网段）。这时，在
接下来的 2 nd 块中的内容一般会不起作用（返回-1）。
   {
   struct sockaddr_in addr;
   socklen_t addrlen = sizeof(struct sockaddr_in);
   if(getsockname(m_sock,(struct sockaddr*)&addr,&addrlen) == 0)
         Self.SetIp(addr);
   }
   // 2nd: better to use addr of our domain
    {
         struct hostent *h;
         char hostname[128];
         char *hostdots[2]={0,0}, *hdptr=hostname;

         if (gethostname(hostname, 128) == -1) return -1;//获得主机名
//       printf("%s\n", hostname);
         while(*hdptr) if(*hdptr++ == '.') {
                  hostdots[0] = hostdots[1];
                  hostdots[1] = hdptr;
         }
         if (hostdots[0] == 0) return -1;//本地主机。
//       printf("%s\n", hostdots[0]);
         if ((h = gethostbyname(hostdots[0])) == NULL) return -1;//由主机名解析 IP。
         //printf("Host domain    : %s\n", h->h_name);
         //printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *)h->h_addr)));
         memcpy(&Self.m_sin.sin_addr,h->h_addr,sizeof(struct in_addr));
   }
   return 0;
}
```

◆ void btTracker::Reset(time_t new_interval)

当客户端与 tracker 服务器连接失败时调用此函数重置客户端的设置。

◆ int btTracker::Connect()

此函数调用 connect_nonb()以无阻塞方式与 tracker 服务器通信。若 connect_nonb()返回-2，则设置 m_status 为 T_CONNECTING，若 connect_nonb()返回成功则调用 btTracker::SendRequest()发送请求信息，否则关闭 m_sock，结束通信返回-1。

◆ int btTracker::SendRequest()

此函数用来向 tracker 服务器发送 GET 请求。GET 请求格式与下面类似：

GET
/announce?info_hash=%0Bp%A3%40%EB%A9%27%21%F4%19%B7%E5NLu%DAn4XI&peer_id=%2DCD0102%2D%F4%17%F4%03%82%26%F8%CCQ%5F%E0%C3&port=2706&uploaded=0&downloaded=0&left=0&compact=1&event=stopped&numwant=100 HTTP/1.0

请求以”GET”字符开始，随后说明 announce 地址在 tracker 服务器上的路径（”/announce”）。然后是种子文件的 Hash 表（”info_hash=…”），然后是客户端 ID（”peer_id=…”），端口号（”port=2706”），客户端已上传的字节（”uploaded=…”），已下载的字节（”downloaded=”），剩余的字节（”left=…”），紧凑模式（”compact=1”），事件类型（”event=…”），客户端想从 tracker 服务器获取的种子数（”numwant=…”），和协议类型（”HTTP/1.0”）。

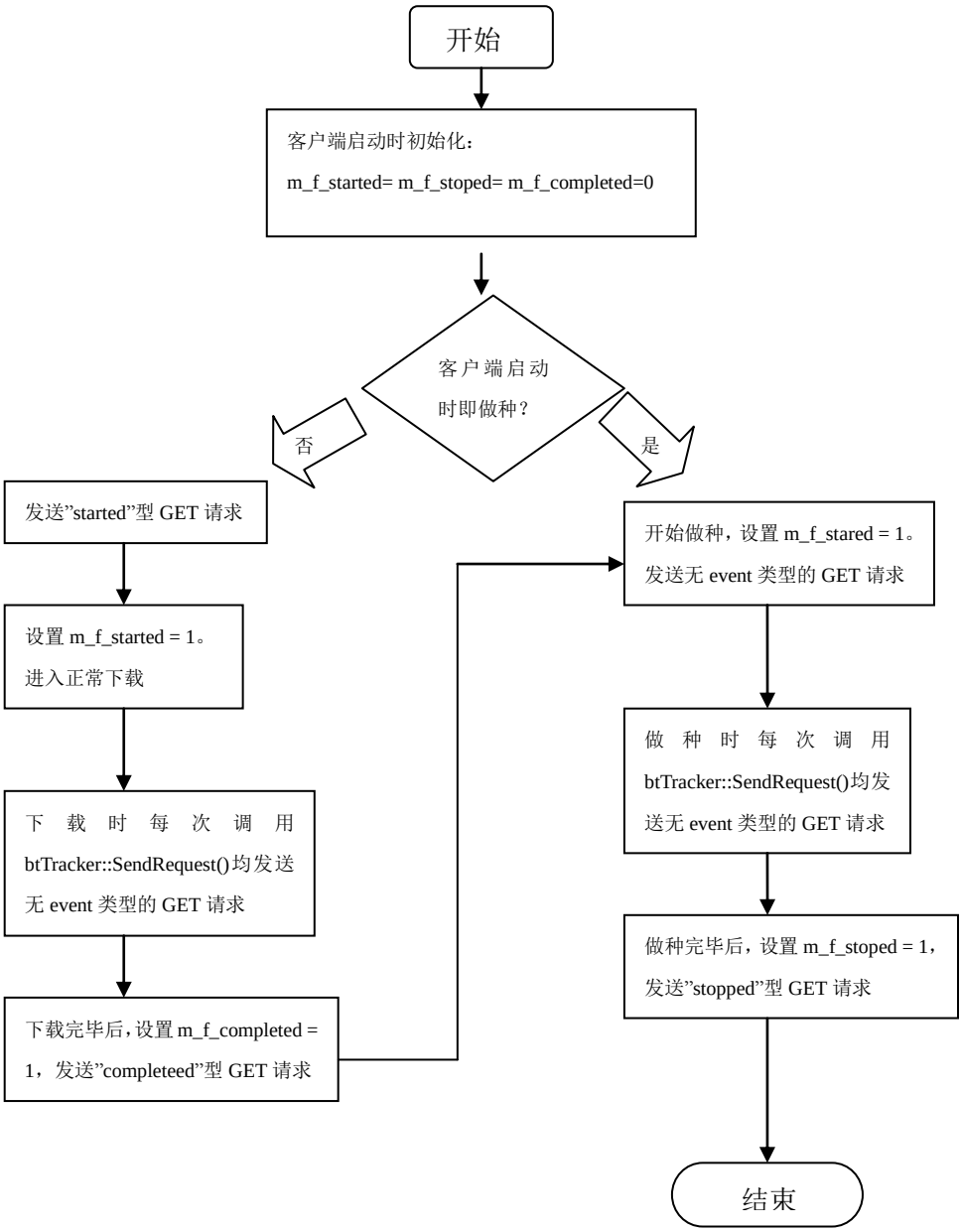
关于四种类型：

Started：客户端第一次向 tracker 发送请求时必须是”started”事件。

Stopped：客户端做种完毕要退出程序时发送”stopped”事件。

Completed：客户端下载完毕发送”completed”事件，但如果客户端在运行前已下载完所有数据，运行时单纯做种，则”completed”事件不必发送。
无：客户端正常下载时发送的 GET 请求中不必包含事件类型，即没有”event=…”字样。

根据标志位 m_f_started，m_f_stoped，m_f_completed 判断发送何种事件类型的算法如下：



图表 11 btTracker::SendRequest()函数流程图

◆ int btTracker::CheckReponse()

此函数主要根据 tracker 服务器发来的 peer 列表信息更新 PeerList 和 IPQUEUE。

函数首先调用 BufIo::FeedIn()将 tracker 服务器发来的信息存储在 m_reponse_buffer 中，然后调用 Http_split()和 Http_reponse_code()对信息进行分析，若 tracker 服务器正常则调用 _UpdatePeerList()更新 peer 链表，若 tracker 服务器被重定向到其它地址则调用 btTracker::Connect()重新连接，否则出错返回。

◆ int btTracker::IntervalCheck(const time_t *pnow, fd_set *rfdp, fd_set *wfdp)

检查与 tracker 服务器通信的时间间隔和客户端得到的 peer 个数，若超过时间间隔（一般为 1800 秒）或客户端没有 peer 可用了，则调用 btTracker::Connect()与 tracker 服务器重新通信，

并根据 tracker 服务器的状态重新设置 m_sock 到可读或可写文件描述符集里。
具体分析如下：

```
int btTracker::IntervalCheck(const time_t *pnow, fd_set *rfdp, fd_set *wfdp)
{
  /* tracker communication */
  if( T_FREE == m_status ){
//   if(*pnow - m_last_timestamp >= m_interval)
     if(*pnow - m_last_timestamp >= m_interval ||   //已经 m_interval 秒没有和 tracker 通信了
          // Connect to tracker early if we run out of peers.
          (!WORLD.TotalPeers() && m_prevpeers &&   //客户端没有 peer 了。
             *pnow - m_last_timestamp >= 15) ){
        m_prevpeers = WORLD.TotalPeers();
  //connect to tracker and send request.
        if(Connect() < 0){ Reset(15); return -1; }       //向 tracker 发起请求。

        if( m_status == T_CONNECTING ){           //若请求正在发送
          FD_SET(m_sock, rfdp);                    //设置 m_soce 到可读 fd_set 中以便接收服务器的响应。
          FD_SET(m_sock, wfdp);                    //设置 m_sock 到可写 fd_set 中以便发送请求。
        }else{
          FD_SET(m_sock,  rfdp);                   //若请求已经发送完毕，则只设置 m_soce 到可读 fd_set 中以便接收服务器的响应
        }
     }
  }else{         //else 的情况很有可能是 m_status == T_CONNECTING 或 m_status == T_READY。
     if( m_status == T_CONNECTING ){
        FD_SET(m_sock, rfdp);
        FD_SET(m_sock, wfdp);
     }else if (INVALID_SOCKET != m_sock){
        FD_SET(m_sock, rfdp);
     }
  }
  return m_sock;
}
```

◆ int btTracker::SocketReady(fd_set *rfdp, fd_set *wfdp, int *nfds)

此函数主要是检查 m_sock 是否在 wfdp 和 wfdp 中。若在 wfdp 中，则表明 m_sock 可写，调用 btTracker::SendRequest()发送请求信息到 tracker 服务器；若在 rfdp 中，则表明 m_sock 上有数据到达，调用 btTracker::CheckReponse()检查 tracker 服务器返回的信息。

5. 后记

5.1 开源和 BitTorrent，不得不说的话

就像 Eric Raymond 在他的《大教堂和市集》中所说的，一个一致而稳定的系统（linux）奇迹般地从那个有着各种不同议程和方法的乱哄哄的市集中产生了。CTorrent 充分体现了这样一条道路：它刚发布时并不稳定，性能也不佳，但因为它是开源的，很多热心人都加入到讨论中来，找 bug，改代码－－现在的 CTorrent 已经今非昔比，它性能出色，资源占用极低，并且代码浅显易懂，易于移植（您可以毫不费力地把它移植到机顶盒，PDA 甚至 Windows 中），是一个非常出色的客户端。CTorrent 的所有发布版本，都遵循 GPL 并有相应的代码发放，您可以自由地阅读，修改，重新发布……只要它们遵守 GPL。大家受惠其中，又回报于它，这样一种其乐融融的开发和维护模式，让我如痴如醉……

但在这美好的憧憬中，已经出现了一些危险的因素：BT 客户端是如此之多，有些客户端自己定义了一些优化性能的机制，如果很多人都使用这样的客户端，会带来非常出色的性能。但由于这些客户端并没有开放源代码（这并没有什么不对，源码开放与否是开发者的自由），也没有说明它们到底是使用了何种机制来提高性能，这就导致了其它客户端与其通信时不能达到最优的效果。一旦各种各样的客户端形成了各自用户群上的优势，基于相同 BT 协议的 BT 客户端软件必然会走上一条相互不能很好兼容的道路。或许，分裂，是一条不该有的归宿。

但我相信，这一切，不是开源之罪。

5.2 BT 的精神：共享，公平和宽容

BT 协议实现了大家共享数据，互通有无，每个人都无私奉献自己的数据的人类理想。在这个实现过程中，BT 协议力保公平：一报还一报，上传越快，下载也就越快。同时，也以宽容之心对待一些由于各种原因不能提供上传的客户端，为它们提供下载。可以说，BT 协议是一个从协议内容本身就体现开源精神的协议，而不是像其它一些协议，只是在协议的实现形式（即软件）上遵循开源，这是 BT 协议的特殊之处。

5.3 本篇文档的版权和莫做害群之马

还是那句老话：作者水平有限，错误在所难免，若您发现了任何问题或有任何建议，欢迎与我联系。除此之外，请所有通过阅读本篇文档而对 BT 协议和 CTorrent 客户端有所熟悉的人注意：不得随便改变客户端程序做损人利己的事情！

我在阅读和分析 CTorrent 源代码的过程中，除了分析程序本身有哪些可以改进的地方，还

想到了一些如何改写程序以显著加快下载速率的方法(但是这些方法是建立在对其它客户端的损害之上的)。一个熟悉 CTorrent 代码的人完全可以写出一个对自己有利而损害他人利益的客户端程序，从而在 BT 下载时获益颇多。我想，这种想法应该永远被禁止付诸实践－－虽然这种规范仅仅是在道义上的，但这对一个真正的程序员来说已经足够了。

5.4 我的敬意

我以最诚挚的敬意向 BT 协议的作者 Bram Cohen，CTorrent 的作者 YuHong，Enhanced CTorrent 的作者 Dennis Holmes，以及所有为 CTorrent 作出贡献的人表示感谢，他们教给了我精神，思想，技巧和方法。同时，没有我的恩师方元老师的大力支持，这篇源码分析只能是空中楼阁－－感谢所有人!

5.5 结语

最后，以 Bram Cohen 的一句话结束”CTorrent 程序源码分析”：

I decided I finally wanted to work on a project that people would actually use, would actually work and would actually be fun.