

摘 要

关键词:

目 录

第一章 绪 论	1
1.1 课程设计的内容	1
1.2 课程设计的工作	1
1.3 报告的结构安排	1
第二章 系统需求分析	2
第三章 技术基础介绍	3
3.1 流媒体播放协议	3
3.1.1 RTSP / RTP / RTCP 协议	3
3.1.1.1 RTP	3
3.1.1.2 RTCP	3
3.1.1.3 RTSP	4
3.1.2 RTMP / RTMPS 协议	4
3.1.3 MMS (Microsoft Media Server) 协议	5
3.1.4 HLS (HTTP live streaming) 协议	5
3.1.5 综述	5
3.2 TS 视频编码	6
3.2.1 ts 层	6
3.2.2 pes 层	7
3.2.3 es 层	8
3.3 M3U8 索引	8
3.3.1 M3U 文件标签及属性	9
3.4 HTTP 协议相关内容	10
3.4.1 HTTP 协议	10
3.4.2 HTTP 报文的格式定义	10
3.4.2.1 请求报文	10
3.4.2.2 响应报文	12
3.5 高性能服务器相关技术	13
3.5.1 Epoll	13
3.5.1.1 Epoll 的优点	14
3.5.1.2 Epoll 的实现原理	14

3.5.1.3 Epoll 的基本使用	16
3.5.2 线程池与 Cache	17
3.5.2.1 使用线程池的目的	17
3.5.2.2 线程池的实现原理	18
3.6 FFmpeg	19
3.7 本章小结	19
第四章 系统概要设计	20
第五章 系统设计与实现	21
5.1 m3u8-segmenter 的设计与实现	21
5.2 HTTP 服务器的基本设计	21
5.3 服务器的高性能优化	22
5.3.1 Epoll 的设计与实现	22
5.3.2 线程池的设计与实现	23
5.3.3 Cache 的设计与实现	25
5.4 本章小结	25
第六章 系统测试	26
致 谢	27
参考文献	28
附录	29

第一章 绪 论

1.1 课程设计的内容

使用面向对象程序设计技术，采用 C++ 语言，设计并实现一个功能较为复杂的流媒体系统。详细内容如下所示：

1. 学习理解视频编解码等相关知识。
2. 设计基于 HLS 协议的流媒体服务器。
3. 设计为 HLS 协议配套的视频切片器及 m3u8 索引生成器。
4. 利用 Web 浏览器作为客户端接收并播放流媒体视频文件。

1.2 课程设计的工作

在本次课程设计中，按照模块化的思路，将 epoll、线程池、http 处理、文件缓存、视频切块及 m3u8 文件生成等模块进行了封装，以便于项目分段完成和调试。并按照软件工程设计要求完善了相关文档。

1.3 报告的结构安排

本文的章节结构安排如下：

- 第一章，绪论。
- 第二章，流媒体系统的需求分析报告。
- 第三章，系统设计中使用的相关技术的基本介绍。
- 第四章，流媒体系统的概要设计与分析。
- 第五章，流媒体系统的设计与实现。
- 第六章，针对流媒体系统的相关测试。

第二章 系统需求分析

第三章 技术基础介绍

当前在 *Internet* 上传送音频和视频等信息主要有两种方式:

- 下载: 完整下载一个视频, 再去播放
- 流式传输: 如优酷、爱奇艺等视频网

在本课程设计中, 使用流式传输的形式完成, 常见的流媒体协议有 RTP、RTCP、RTSP、MMS、HLS 等, 其中 RTP、RTCP、RTSP 一般组合使用, 为了实现更高的服务器性能, 本次课程设计选用了 HLS 协议作为传输协议。本章将给出流媒体系统所用技术的基础简介和协议选择的相关分析。

3.1 流媒体播放协议

3.1.1 RTSP / RTP / RTCP 协议

三个协议需要搭配使用, 才能实现流媒体播放的数据传输、控制以及稳定性保障。三个协议的详细介绍如下:

3.1.1.1 RTP

实时传输协议 (Real-time Transport Protocol 或简写 RTP) 是一个网络传输协议。

RTP 协议详细说明了在互联网上传递音频和视频的标准数据包格式。它一开始被设计为一个多播协议, 但后来被用在很多单播应用中。RTP 协议常用于流媒体系统 (配合 RTSP 协议), 视频会议和一键通 (Push to Talk) 系统 (配合 H.323 或 SIP), 使它成为 IP 电话产业的技术基础。

RTP 为 *Internet* 上端到端的实时传输提供时间信息和流同步, 但并不保证服务质量, 服务质量由 RTCP 来提供。RTP 协议和 RTCP(RTP 控制协议) 一起使用, 而且它是创建在 UDP 协议上的传输层协议。

3.1.1.2 RTCP

实时传输控制协议 (Real-time Transport Control Protocol 或 RTP Control Protocol 或简写 RTCP) 是实时传输协议 (RTP) 的一个姐妹协议。

RTCP 为 RTP 媒体流提供信道外 (out-of-band) 控制。RTCP 本身并不传输数据, 但和 RTP 一起协作将多媒体数据打包和发送。RTCP 定期在多媒体流会话参与者之间传输控制数据。RTCP 的主要功能是为 RTP 所提供的服务质量 (Quality

of Service) 提供反馈。RTCP 收集相关媒体连接的统计信息, 例如: 传输字节数, 传输分组数, 丢失分组数, jitter, 单向和双向网络延迟等等, 网络应用程序即可利用 RTCP 的统计信息来控制传输的品质, 比如当网络带宽高负载时限制信息流量或改用压缩比较小的编解码器。

在传输层上 RTP 使用一个偶数 UDP port; 而 RTCP 则使用 RTP 的下一个 port, 也就是一个奇数 port。

3.1.1.3 RTSP

即时串流协定 (Real Time Streaming Protocol, RTSP) 是用来控制声音或影像的多媒体串流协议, 并允许同时多个串流需求控制

允许同时多个串流需求控制 (Multicast), 除了可以降低服务器端的网络用量, 更进而支持多方视讯会议 (Video Conference)。因为与 HTTP1.1 的运作方式相似, 所以代理服务器 (Proxy) 的缓冲功能 (Cache) 也同样适用于 RTSP, 并因 RTSP 具有重新导向功能, 可视实际负载情况来转换提供服务的服务器, 以避免过大的负载集中于同一服务器而造成延迟。

作为传输层协议, 传输时所用的网络通讯协定并不在其定义的范围内, 服务器端可以自行选择使用 TCP 或 UDP 来传送串流内容, 它的语法和运作跟 HTTP 1.1 类似, 但并不特别强调时间同步, 所以比较能容忍网络延迟。

3.1.2 RTMP / RTMPS 协议

RTMP(Real Time Messaging Protocol) 实时消息传送协议是 Adobe Systems 公司为 Flash 播放器和服务器之间音频、视频和数据传输开发的开放协议。它有三种变种:

- 工作在 TCP 之上的明文协议, 使用端口 1935。
- RTMPT 封装在 HTTP 请求之中, 可穿越防火墙。
- RTMPS 类似 RTMPT, 但使用的是 HTTPS 连接。

RTMP 协议 (Real Time Messaging Protocol) 是被 Flash 用于对象, 视频, 音频的传输. 这个协议建立在 TCP 协议或者轮询 HTTP 协议之上。

RTMP 协议就像一个用来装数据包的容器, 这些数据既可以是 AMF 格式的数据, 也可以是 FLV 中的视/音频数据. 一个单一的连接可以通过不同的通道传输多路网络流. 这些通道中的包都是按照固定大小的包传输的.

3.1.3 MMS (Microsoft Media Server) 协议

MMS (Microsoft Media Server) 是一种串流媒体传送协议，用来访问并流式接收 Windows Media 服务器中 .asf 文件的一种协议。

MMS 协议用于访问 Windows Media 发布点上的单播内容。MMS 是连接 Windows Media 单播服务的默认方法。若观众在 Windows Media Player 中键入一个 URL 以连接内容，而不是通过超级链接访问内容，则他们必须使用 MMS 协议引用该流。MMS 的预设埠（端口）是 1755。

当使用 MMS 协议连接到发布点时，使用协议翻转以获得最佳连接。“协议翻转”始于试图通过 MMSU 连接客户端。MMSU 是 MMS 协议结合 UDP 数据传送。如果 MMSU 连接不成功，则服务器试图使用 MMST。MMST 是 MMS 协议结合 TCP 数据传送。

3.1.4 HLS (HTTP live streaming) 协议

HTTP Live Streaming (HLS) 是苹果公司 (Apple Inc.) 实现的基于 HTTP 的流媒体传输协议，可实现流媒体的直播和点播，主要应用在 iOS 系统，为 iOS 设备（如 iPhone、iPad）提供音视频直播和点播方案。

HLS 点播，基本上就是常见的分段 HTTP 点播，不同在于，它的分段非常小。相对于常见的流媒体直播协议，例如 RTMP 协议、RTSP 协议、MMS 协议等，HLS 直播最大的不同在于，直播客户端获取到的，并不是一个完整的数据流。HLS 协议在服务器端将直播数据流存储为连续的、很短时长的媒体文件（MPEG-TS 格式），而客户端则不断的下载并播放这些小文件，因为服务器端总是会将最新的直播数据生成新的小文件，这样客户端只要不停的按顺序播放从服务器获取到的文件，就实现了直播。

由此可见，基本上可以认为，HLS 是以点播的技术方式来实现直播。由于数据通过 HTTP 协议传输，所以完全不用考虑防火墙或者代理的问题，而且分段文件的时长很短，客户端可以很快的选择和切换码率，以适应不同带宽条件下的播放。不过 HLS 的这种技术特点，决定了它的延迟一般总是会高于普通的流媒体直播协议。

3.1.5 综述

由于 MMS 协议需要配和 Windows Media 服务器；RTMP 协议需要配合 Adobe Flash 使用；RTP 协议本身实现高并发存在极大难度，因此选择 HLS 协议作为流媒体系统的传输协议，在 HLS 协议的支持下，可以通过 H5 方便的在各种平台进行

视频播放，同时对于火热的移动设备，使用 H5 可以显著降低系统资源占用电量消耗，存在较大优势。

因此，本章接下来将围绕 HTTP、Epoll、线程池、Cache、m3u8 文件、FFmpeg、ts 视频编码等方面进行介绍。

3.2 TS 视频编码

ts 文件为传输流文件，视频编码主要格式 h264/mpeg4，音频为 acc/MP3。ts 文件分为三层（ts 文件的三层结构如图（3-1）所示）：

- ts 层 (Transport Stream): 在 pes 层加入数据流的识别和传输必须的信息。
- pes 层 (Packet Elemental Stream): 在音视频数据上加了时间戳等对数据帧的说明信息。
- es 层 (Elementary Stream): 音视频数据。

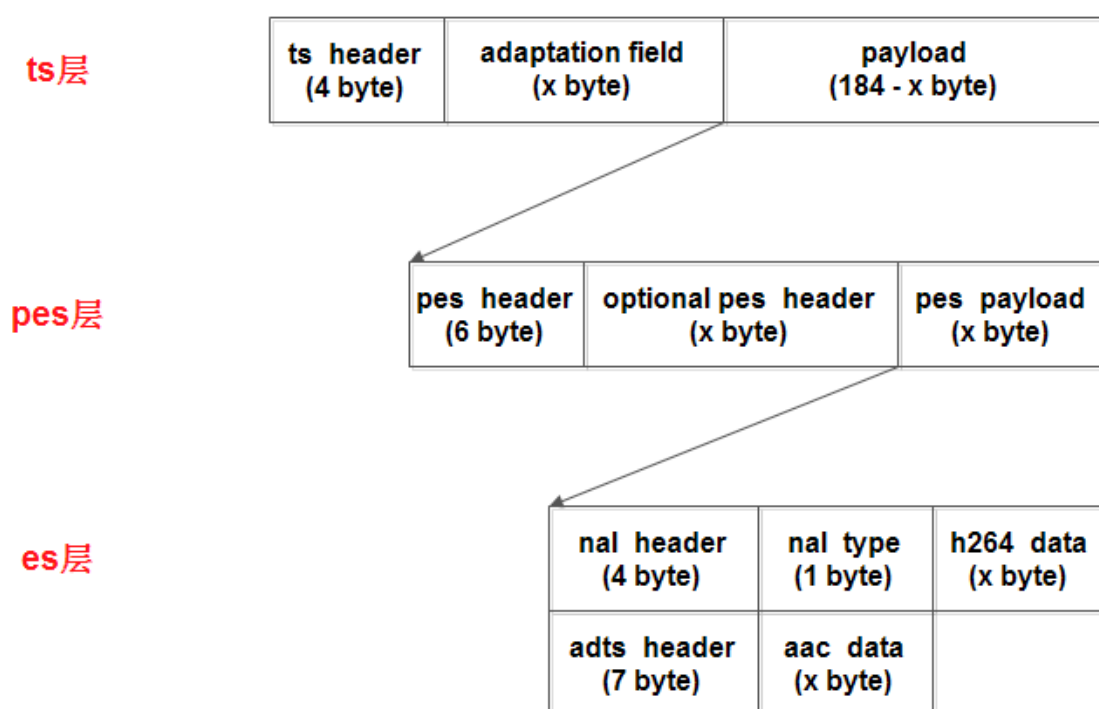


图 3-1 TS 视频文件的三层结构

3.2.1 ts 层

ts 包大小固定为 188 字节，ts 层分为三个部分：ts header、adaptation field、payload。ts header 固定 4 个字节；adaptation field 可能存在也可能不存在，主要作用是给不足 188 字节的数据做填充；payload 是 pes 数据。

ts header sync_byte 8b 同步字节，固定为 0x47 transport_error_indicator 1b 传输错误指示符，表明在 ts 头的 adapt 域后由一个无用字节，通常都为 0，这个字节算在 adapt 域长度内 payload_unit_start_indicator 1b 负载单元起始标示符，一个完整的数据包开始时标记为 1 transport_priority 1b 传输优先级，0 为低优先级，1 为高优先级，通常取 0 pid 13b pid 值 transport_scrambling_control 2b 传输加扰控制，00 表示未加密 adaptation_field_control 2b 是否包含自适应区，‘00’保留；‘01’为无自适应域，仅含有效负载；‘10’为仅含自适应域，无有效负载；‘11’为同时带有自适应域和有效负载。continuity_counter 4b 递增计数器，从 0-f，起始值不一定取 0，但必须是连续的 ts 层的内容是通过 PID 值来标识的，主要内容包括：PAT 表、PMT 表、音频流、视频流。解析 ts 流要先找到 PAT 表，只要找到 PAT 就可以找到 PMT，然后就可以找到音视频流了。PAT 表的 PID 值固定为 0。PAT 表和 PMT 表需要定期插入 ts 流，因为用户随时可能加入 ts 流，这个间隔比较小，通常每隔几个视频帧就要加入 PAT 和 PMT。PAT 和 PMT 表是必须的，还可以加入其它表如 SDT（业务描述表）等，不过 hls 流只要有 PAT 和 PMT 就可以播放了。PAT 表：他主要的作用就是指明了 PMT 表的 PID 值。PMT 表：他主要的作用就是指明了音视频流的 PID 值。音频流/视频流：承载音视频内容。

3.2.2 pes 层

pes 层是在每一个视频/音频帧上加入了时间戳等信息，pes 包内容很多，我们只留下最常用的。

pes start code 3B 开始码，固定为 0x000001 stream id 1B 音频取值 (0xc0-0xdf)，通常为 0xc0 视频取值 (0xe0-0xef)，通常为 0xe0 pes packet length 2B 后面 pes 数据的长度，0 表示长度不限制，只有视频数据长度会超过 0xffff flag 1B 通常取值 0x80，表示数据不加密、无优先级、备份的数据 flag 1B 取值 0x80 表示只含有 pts，取值 0xc0 表示含有 pts 和 dts pes data length 1B 后面数据的长度，取值 5 或 10 pts 5B 33bit 值 dts 5B 33bit 值 pts 是显示时间戳、dts 是解码时间戳，视频数据两种时间戳都需要，音频数据的 pts 和 dts 相同，所以只需要 pts。有 pts 和 dts 两种时间戳是 B 帧引起的，I 帧和 P 帧的 pts 等于 dts。如果一个视频没有 B 帧，则 pts 永远和 dts 相同。从文件中顺序读取视频帧，取出的帧顺序和 dts 顺序相同。dts 算法比较简单，初始值 + 增量即可，pts 计算比较复杂，需要在 dts 的基础上加偏移量。音频的 pes 中只有 pts（同 dts），视频的 I、P 帧两种时间戳都要有，视频 B 帧只要 pts（同 dts）。打包 pts 和 dts 就需要知道视频帧类型，但是通过容器格式我们是无法判断帧类型的，必须解析 h.264 内容才可以获取帧类型。举例

说明: I P B B B P 读取顺序: 1 2 3 4 5 6 dts 顺序: 1 2 3 4 5 6 pts 顺序: 1 5 3 2 4
 6 点播视频 dts 算法: $dts = \text{初始值} + 90000 / \text{video_frame_rate}$, 初始值可以随便指定, 但是最好不要取 0, video_frame_rate 就是帧率, 比如 23、30。pts 和 dts 是以 timescale 为单位的, $1s = 90000 \text{ time scale}$, 一帧就应该是 $90000/\text{video_frame_rate}$ 个 timescale。用一帧的 timescale 除以采样频率就可以转换为一帧的播放时长点播音频 dts 算法: $dts = \text{初始值} + (90000 * \text{audio_samples_per_frame}) / \text{audio_sample_rate}$, audio_samples_per_frame 这个值与编解码相关, aac 取值 1024, mp3 取值 1158, audio_sample_rate 是采样率, 比如 24000、41000。AAC 一帧解码出来是每声道 1024 个 sample, 也就是说一帧的时长为 $1024/\text{sample_rate}$ 秒。所以每一帧时间戳依次 0, $1024/\text{sample_rate}$, ..., $1024*n/\text{sample_rate}$ 秒。直播视频的 dts 和 pts 应该直接用直播数据流中的时间, 不应该按公式计算。

3.2.3 es 层

es 层指的就是音视频数据, 我们只介绍 h.264 视频和 aac 音频。h.264 视频: 打包 h.264 数据我们必须给视频数据加上一个 nalu (Network Abstraction Layer unit), nalu 包括 nalu header 和 nalu type, nalu header 固定为 0x00000001 (帧开始) 或 0x000001 (帧中)。h.264 的数据是由 slice 组成的, slice 的内容包括: 视频、sps、pps 等。nalu type 决定了后面的 h.264 数据内容。

F 1b forbidden_zero_bit, h.264 规定必须取 0 NRI 2b nal_ref_idc, 取值 0 3, 指示这个 nalu 的重要性, I 帧、sps、pps 通常取 3, P 帧通常取 2, B 帧通常取 0 Type 5b 参考下表 nal_unit_type 说明 0 未使用 1 非 IDR 图像片, IDR 指关键帧 2 片分区 A 3 片分区 B 4 片分区 C 5 IDR 图像片, 即关键帧 6 补充增强信息单元 (SEI) 7 SPS 序列参数集 8 PPS 图像参数集 9 分解符 10 序列结束 11 码流结束 12 填充 13 23 保留 24 31 未使用红色字体显示的内容是最常用的, 打包 es 层数据时 pes 头和 es 数据之间要加入一个 type=9 的 nalu, 关键帧 slice 前必须要加入 type=7 和 type=8 的 nalu, 而且是紧邻。

3.3 M3U8 索引

1. m3u8 是一种可扩展的播放列表文件格式。它是一个包含 UTF-8 编码文字的 m3u 播放列表。m3u 是包含媒体文件 URL 的一个事实上的播放列表标准。这种格式被用来作为 HTTP Live 媒体流索引文件的格式。
2. m3u8 是一种视频列表格式, 里面有真正的视频的链接, 另外在 m3u8 里面还可以再嵌套一层 m3u8。

3. m3u8 是视频列表，视频编码可以是 H.264 等。
4. m3u8 并非苹果独占，m3u8 这种列表其实编码格式是公开的。

3.3.1 M3U 文件标签及属性

M3U 文件标签及属性说明：M3U 文件中可以包含多个 tag，每个 tag 的功能和属性如下：

- **#EXTM3U**

每个 M3U 文件第一行必须是这个 tag，请标示作用。

- **#EXT-X-MEDIA-SEQUENCE:140651513**

每一个 media URI 在 PlayList 中只有唯一的序号，相邻之间序号 +1，一个 media URI 并不是必须要包含的，如果没有，默认为 0

- **#EXTINF:duration**

指定每个媒体段 (ts) 的持续时间 (秒)，仅对其后面的 URI 有效，title 是下载资源的 url

- **#EXT-X-TARGETDURATION**

指定最大的媒体段时间长 (秒)。所以 #EXTINF 中指定的时间长度必须小于或是等于这个最大值。这个 tag 在整个 PlayList 文件中只能出现一次 (在嵌套的情况下，一般有真正 ts url 的 m3u8 才会出现该 tag)

- **#EXT-X-PROGRAM-DATE-TIME**

将一个绝对时间或是日期和一个媒体段中的第一个 sample 相关联，只对下一个 media URI 有效，格式如 #EXT-X-PROGRAM-DATE-TIME:

- **#EXT-X-ALLOW-CACHE**

是否允许做 cache，这个可以在 PlayList 文件中任意地方出现，并且最多出现一次，作用效果是所有的媒体段。

- **#EXT-X-PLAYLIST-TYPE**

提供关于 PlayList 的可变性的信息，这个对整个 PlayList 文件有效，是可选的，格式如下：#EXT-X-PLAYLIST-TYPE:: 如果是 VOD，则服务器不能改变 PlayList 文件；如果是 EVENT，则服务器不能改变或是删除 PlayList 文件中的任何部分，但是可以向该文件中增加新的一行内容。

- **#EXT-X-ENDLIST**

表示 PlayList 的末尾，它可以在 PlayList 中任意位置出现，但是只能出现一个。

3.4 HTTP 协议相关内容

3.4.1 HTTP 协议

HTTP 是 Hyper Text Transfer Protocol（超文本传输协议）的缩写。它的发展是万维网协会（World Wide Web Consortium）和 Internet 工作小组 IETF（Internet Engineering Task Force）合作的结果，（他们）最终发布了一系列的 RFC，RFC 1945 定义了 HTTP/1.0 版本。其中最著名的就是 RFC 2616。RFC 2616 定义了今天普遍使用的一个版本——HTTP 1.1。

它可以使浏览器更加高效，使网络传输减少。它不仅保证计算机正确快速地传输超文本文档，还确定传输文档中的哪一部分，以及哪部分内容首先显示（如文本先于图形）等。

3.4.2 HTTP 报文的格式定义

HTTP 客户程序（例如浏览器），向服务器发送请求的时候必须指明请求类型（一般是 GET 或者 POST）。如有必要，客户程序还可以选择发送其他的请求头。大多数请求头并不是必需的，但 Content-Length 除外。对于 POST 请求来说 Content-Length 必须出现。

3.4.2.1 请求报文

HTTP 请求报文由请求行、请求头部、空行和请求包体 4 个部分组成，如图 (3-2) 所示：



图 3-2 HTTP 请求报文格式

下面对请求报文格式进行简单的分析：

请求行

第一行为请求行，由方法字段、URL 字段和 HTTP 协议版本字段 3 个部分组成，他们之间使用空格隔开。常用的 HTTP 请求方法有 GET、POST 等。

- GET: 当客户端要从服务器中读取某个资源时, 使用 GET 方法。GET 方法要求服务器将 URL 定位的资源放在响应报文的数据部分, 回送给客户端, 即向服务器请求某个资源。使用 GET 方法时, 请求参数和对应的值附加在 URL 后面, 利用一个问号 (“?”) 代表 URL 的结尾与请求参数的开始, 传递参数长度受限制。
- POST: 当客户端给服务器提供信息较多时可以使用 POST 方法, POST 方法向服务器提交数据, 比如完成表单数据的提交, 将数据提交给服务器处理。GET 一般用于获取/查询资源信息, POST 会附带用户数据, 一般用于更新资源信息。POST 方法将请求参数封装在 HTTP 请求数据中, 以名称/值的形式出现, 可以传输大量数据。

请求头部

请求头部由关键字/值对组成, 每行一对, 关键字和值用英文冒号“:”分隔。请求头部通知服务器有关于客户端请求的信息, 典型的请求头有:

- User-Agent: 产生请求的浏览器类型。
- Accept: 客户端可识别的响应内容类型列表; 星号“*”用于按范围将类型分组, 用“/*/*”指示可接受全部类型, 用“type/*”指示可接受 type 类型的所有子类型。
- Accept-Language: 客户端可接受的自然语言。
- Accept-Encoding: 客户端可接受的编码压缩格式。
- Accept-Charset: 可接受的应答的字符集。
- Host: 请求的主机名, 允许多个域名同处一个 IP 地址, 即虚拟主机。
- connection: 连接方式 (close 或 keepalive)。
- Cookie: 存储于客户端扩展字段, 向同一域名的服务端发送属于该域的 cookie。

空行

最后一个请求头之后是一个空行, 发送回车符和换行符, 通知服务器以下不再有请求头。

请求包体

请求包体不在 GET 方法中使用, 而是在 POST 方法中使用。POST 方法适用于需要客户填写表单的场合。与请求包体相关的最常使用的是包体类型 Content-Type 和包体长度 Content-Length。

3.4.2.2 响应报文

HTTP 响应报文由状态行、响应头部、空行和响应包体 4 个部分组成，如图 (3-3) 所示：

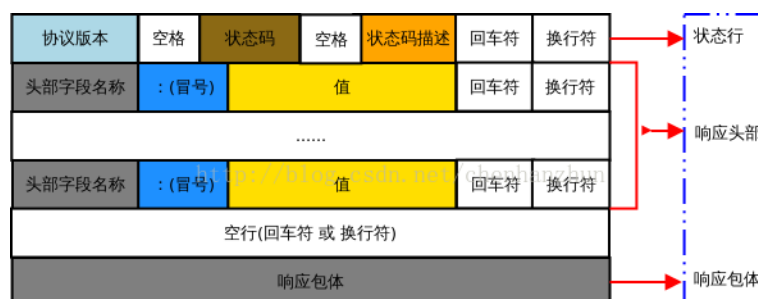


图 3-3 HTTP 响应报文格式

下面对响应报文格式进行简单的分析：

状态行

状态行由 HTTP 协议版本字段、状态码和状态码的描述文本 3 个部分组成，他们之间使用空格隔开。

状态码由三位数字组成，第一位数字表示响应的类型，常用的状态码有五大类如下所示：

- 1xx：表示服务器已接收了客户端请求，客户端可继续发送请求。
- 2xx：表示服务器已成功接收到请求并进行处理。
- 3xx：表示服务器要求客户端重定向。
- 4xx：表示客户端的请求有非法内容。
- 5xx：表示服务器未能正常处理客户端的请求而出现意外错误。

状态码描述文本有如下取值：

- 200 OK：表示客户端请求成功。
- 400 Bad Request：表示客户端请求有语法错误，不能被服务器所理解。
- 401 Unauthorized：表示请求未经授权，该状态代码必须与 WWW-Authenticate 报头域一起使用。
- 403 Forbidden：表示服务器收到请求，但是拒绝提供服务，通常会在响应正文中给出不提供服务的原因。
- 404 Not Found：请求的资源不存在，例如，输入了错误的 URL。
- 500 Internal Server Error：表示服务器发生不可预期的错误，导致无法完成客户端的请求。

- 503 Service Unavailable: 表示服务器当前不能够处理客户端的请求, 在一段时间之后, 服务器可能会恢复正常。

响应头部

响应头部可包括以下信息:

- Location: Location 响应报头域用于重定向接受者到一个新的位置。例如: 客户端所请求的页面已不存在原先的位置, 为了让客户端重定向到这个页面新的位置, 服务器端可以发回 Location 响应报头后使用重定向语句, 让客户端去访问新的域名所对应的服务器上的资源。
- Server: Server 响应报头域包含了服务器用来处理请求的软件信息及其版本。它和 User-Agent 请求报头域是相对应的, 前者发送服务器端软件的信息, 后者发送客户端软件 (浏览器) 和操作系统的信息。
- Vary: 指示不可缓存的请求头列表。
- Connection: 连接方式。
 1. 对于请求来说: close(告诉 web 服务器或者代理服务器, 在完成本次请求的响应后, 断开连接, 不等待本次连接的后续请求); keepalive(告诉 WEB 服务器或者代理服务器, 在完成本次请求的响应后, 保持连接, 等待本次连接的后续请求)。
 2. 对于响应来说: close(连接已经关闭); keepalive(连接保持着, 在等待本次连接的后续请求); Keep-Alive: 如果浏览器请求保持连接, 则该头部表明希望 web 服务器保持连接多长时间 (秒); 例如: Keep-Alive: 300。

空行

最后一个响应头部之后是一个空行, 发送回车符和换行符, 通知服务器以下不再有响应头部。

响应包体

服务器返回给客户端的文本信息。

3.5 高性能服务器相关技术

3.5.1 Epoll

epoll 是 Linux 内核为处理大批量文件描述符而作了改进的 poll, 是 Linux 下多路复用 IO 接口 select/poll 的增强版本。

3.5.1.1 Epoll 的优点

epoll 可以显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率。同时，在获取事件时它无须遍历整个被侦听的描述符集，只需要遍历那些被内核 IO 事件异步唤醒而加入 Ready 队列的描述符集合即可。

epoll 除了提供 select/poll 中 IO 事件的水平触发 (Level Triggered) 外，还提供了边缘触发 (Edge Triggered)，使得用户空间程序有可能缓存 IO 状态，减少 epoll_wait/epoll_pwait 的调用，提高应用程序效率。

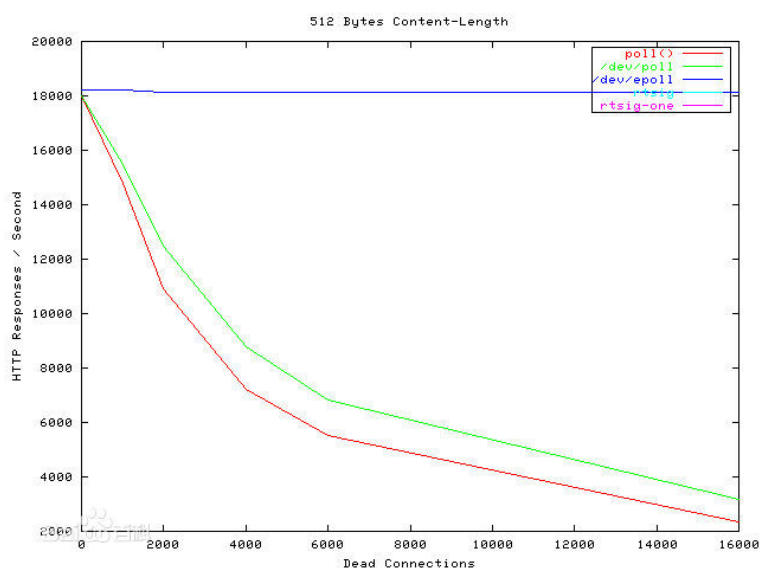


图 3-4 Epoll, Poll, Select 对比

相对于 poll 和 select，其主要优势有以下三点：

1. 支持一个进程打开大数目的 socket 描述符
2. IO 效率不随 FD 数目增加而线性下降
3. 使用 mmap 加速内核与用户空间的消息传递

3.5.1.2 Epoll 的实现原理

当服务器存在大量连接，但同时仅有少量连接处于活状态时，非常适合 epoll 的使用。在使用 epoll 实现高并发时，epoll 通过在 Linux 内核中申请一个简易的文件系统 (B+ 树，如图 (3-5) 所示)。将 select/poll 的调用分成了 3 个部分：

1. 调用 epoll_create() 建立一个 epoll 对象 (在 epoll 文件系统中为这个句柄对象分配资源)
2. 调用 epoll_ctl 向 epoll 对象中添加所有连接的 Socket
3. 调用 epoll_wait 收集发生的事件的连接

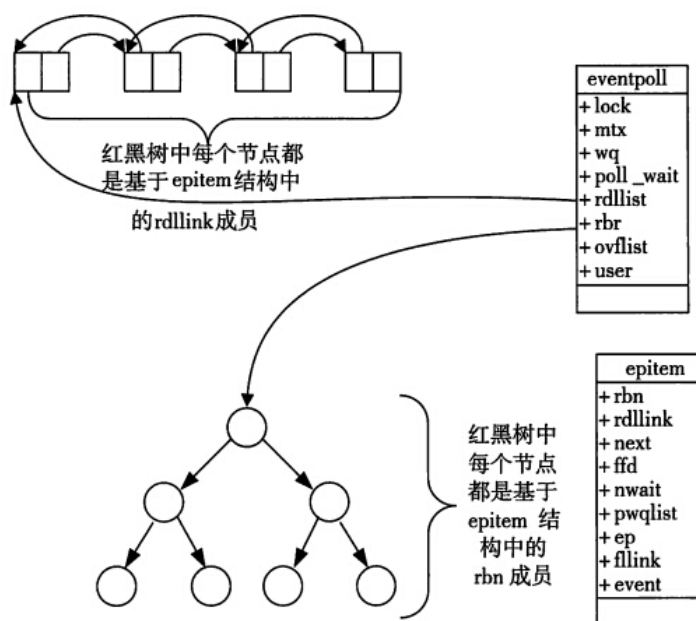


图 3-5 Epoll 数据结构示意图

使用 epoll 时，只需要在进程启动时建立一个 epoll 对象，之后只在需要时向这个 epoll 对象中添加或者删除连接。

当某一进程调用 epoll_create 方法时，Linux 内核会创建一个 eventpoll 结构体，这个结构体中有两个成员与 epoll 的使用方式密切相关。eventpoll 结构体如下所示：

```
struct eventpoll{
    ....
    /*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件*/
    struct rb_root rbr;
    /*双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ....
};
```

每一个 epoll 对象都有一个独立的 eventpoll 结构体，用于存放通过 epoll_ctl 方法向 epoll 对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来 (红黑树的插入时间效率是 $\lg n$ ，其中 n 为树的高度)。

而所有添加到 epoll 中的事件都会与设备 (网卡) 驱动程序建立回调关系，也就是说，当相应的事件发生时调用这个回调方法。这个回调方法在内核中叫 ep_poll_callback，它会将发生的事件添加到 rdlist 双链表中。

在 epoll 中，对于每一个事件，都会建立一个 epitem 结构体，如下所示：

```
struct epitem{
```

```

struct rb_node rbn; //红黑树节点
struct list_head rdllink; //双向链表节点
struct epoll_filefd ffd; //事件句柄信息
struct eventpoll *ep; //指向其所属的eventpoll对象
struct epoll_event event; //期待发生的事件类型
}

```

当调用 `epoll_wait` 检查是否有事件发生时，只需要检查 `eventpoll` 对象中的 `rdlist` 双链表中是否有 `epitem` 元素即可。如果 `rdlist` 不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。

3.5.1.3 Epoll 的基本使用

`epoll` 由 `epoll_create`, `epoll_ctl`, `epoll_wait` 三个基本接口组成，具体含义如下。

epoll_create

在新版本的 Linux 内核中，`epoll_create` 函数的参数基本失去作用，一般调用 `epoll_create(0)` 即可。该函数会返回一个 `epoll` 专用的文件描述符，用于操作 `epoll` 的行为^[1]。

epoll_ctl

创建完 `epoll` 之后，使用 `epoll_ctl` 来处理 `epoll` 中的事件，该函数原型如下所示：

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

其中第 1 个参数就是 `epoll` 描述符，第二个参数 `op` 表示要执行的动作，可能有 3 个宏。

1. `EPOLL_CTL_ADD`: 注册新的 `fd` 到 `epfd` 中。
2. `EPOLL_CTL_MOD`: 修改已经注册的 `fd` 中的监听事件。
3. `EPOLL_CTL_DEL`: 从 `epfd` 中删除一个 `fd`。

第 3 个参数是要监听的文件描述符，第 4 个参数是要监听的事件，事件的结构体定义如下所示。

```

struct epoll_event{
    __uint32_t events;
    epoll_data_t data;
}

```

`events` 表示事件类型，而 `data` 则是用户自定义的数据。`epoll` 支持的事件类型如下。

1. EPOLLIN: 表示对应的文件描述符可读（包括对端 Socket 正常关闭）。
2. EPOLLOUT: 表示对应的文件描述符可写。
3. EPOLLPRI: 表示对应的文件描述符有紧急的数据可读（这里主要是有带外数据到来）。
4. EPOLLERR: 表示对应的文件描述符发生错误。
5. EPOLLHUP: 表示对应的文件描述符被挂断。
6. EPOLLET: 将 epoll 设置为边缘触发 (Edge Triggered) 模式，这是相对于水平触发 (Level Triggered) 来说的
7. EPOLLONESHOT: 只监听一次事件，当监听完这次事件之后，如果还需要继续监听该 Socket，需要再次将这个 Socket 加入到 epoll 队列中。

epoll_wait

该函数用于等待事件发生，其原型如下所示。

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

该函数的第 1 个参数是描述符，第 2 个参数是一个事件组，需要用户分配内存，第 3 个参数是等待的事件最大数，最后一个参数是超时事件。

3.5.2 线程池与 Cache

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件），则线程池将插入另一个辅助线程来使所有处理器保持繁忙。超过任务数最大值的线程可以排队，但他们要等到其他线程完成后才启动。

同时应用 Cache 将访问过的文件暂存于内存中，使得重复访问相同文件时可以避免对磁盘的 I/O 操作，极大的缩短处理时间，提高服务器的并发性能。

3.5.2.1 使用线程池的目的

目前的大多数网络服务器，包括 Web 服务器、Email 服务器以及数据库服务器等都具有一个共同点，就是单位时间内必须处理数目巨大的连接请求，但处理时间却相对较短。

传统多线程方案中我们采用的服务器模型则是一旦接受到请求之后，即创建一个新的线程，由该线程执行任务。任务执行完毕后，线程退出，这就是“即时创建，即时销毁”的策略。尽管与创建进程相比，创建线程的时间已经大大的缩短，

但是如果提交给线程的任务是执行时间较短，而且执行次数极其频繁，那么服务器将处于不停的创建线程，销毁线程的状态。

我们将传统方案中的线程执行过程分为三个过程： T_1 、 T_2 、 T_3 。

- T_1 ：线程创建时间。
- T_2 ：线程执行时间，包括线程的同步等时间。
- T_3 ：线程销毁时间。

那么可以看出，线程本身的开销所占的比例为 $\frac{T_1+T_3}{T_1+T_2+T_3}$ 。如果线程执行的时间很短的话，这比开销可能占到 20%-50% 左右。如果任务执行时间很频繁的话，这笔开销将是不可忽略的。

3.5.2.2 线程池的实现原理

线程池采用预创建的技术，在应用程序启动之后，将立即创建一定数量的线程 (N1)，放入空闲队列中。这些线程都是处于阻塞 (Suspended) 状态，不消耗 CPU，但占用较小的内存空间。当任务到来后，缓冲池选择一个空闲线程，把任务传入此线程中运行。当 N1 个线程都在处理任务后，缓冲池自动创建一定数量的新线程，用于处理更多的任务。在任务执行完毕后线程也不退出，而是继续保持在池中等待下一次的任务。如图 (3-6) 所示。

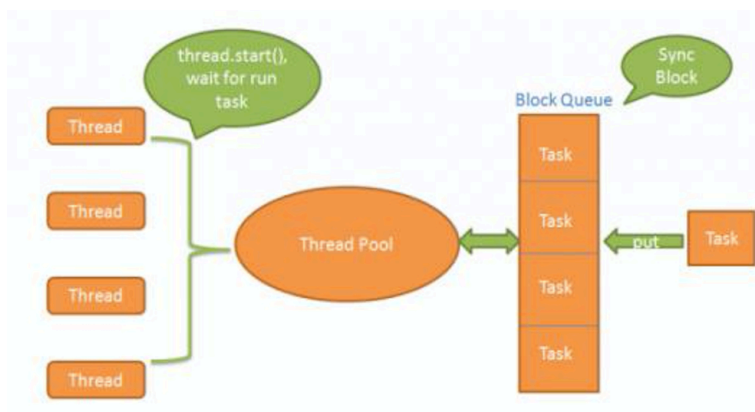


图 3-6 线程池工作方式

同时，线程池能够减少创建的线程个数。通常线程池所允许的并发线程是有上界的，如果同时需要并发的线程数超过上界，那么一部分线程将会等待。

线程池将线程创建和销毁本身所带来的开销分摊到了各个具体的任务上，执行次数越多，每个任务所分担到的线程本身开销则越小。在这种方式下，可以极大的减小常规多线程由于线程创建、销毁带来的大量资源消耗。

3.6 FFmpeg

FFmpeg 是一套可以用来记录、转换数字音频、视频，并能将其转化为流的开源计算机程序。它包括了领先的音/视频编码库 libavcodec 等。其包含的如下主要库将用于实现视频文件的切割分片以及 m3u8 文件的生成。

- *libavformat*: 用于各种音视频封装格式的生成和解析，包括获取解码所需信息以生成解码上下文结构和读取音视频帧等功能；
- *libavcodec*: 用于各种类型声音/图像编解码；
- *libavutil*: 包含一些公共的工具函数；
- *libswscale*: 用于视频场景比例缩放、色彩映射转换；
- *libpostproc*: 用于后期效果处理；
- *ffmpeg*: 该项目提供的一个工具，可用于格式转换、解码或电视卡即时编码等；
- *ffsever*: 一个 HTTP 多媒体即时广播串流服务器；
- *ffplay*: 是一个简单的播放器，使用 ffmpeg 库解析和解码，通过 SDL 显示；

3.7 本章小结

本章论述了流媒体传输协议的选择；HLS 协议的相关内容；HTTP 协议以及 HTTP 请求处理的基本内容；为了实现服务器高并发能力采用的 Epoll、线程池、Cache 等相关技术的基本概念；以及 FFmpeg 开源框架在视频编解码方面的相关内容。为系统的实现做好了知识和技术预备。

第四章 系统概要设计

第五章 系统设计与实现

5.1 m3u8-segmenter 的设计与实现

5.2 HTTP 服务器的基本设计

在本课程设计中，定义如下 5 个宏，分别表示 HTTP 服务处于：操作成功、读、写、关闭连接以及错误状态，以便于对 HTTP 服务的状态进行传递与处理。

```
#define STATUS_SUCCESS 0
#define STATUS_WRITE 1
#define STATUS_READ 2
#define STATUS_CLOSE 3
#define STATUS_ERROR -1
```

并在 HttpServe 类中定义如下 private 变量，以完成对 HTTP 服务的参数设定。

```
static const int READ_BUFFER_SIZE = 1024; // 读缓冲区的大小
static const int WRITE_BUFFER_SIZE = 1024; // 写缓冲区的大小
static const char rootDir_[]; // 网页的根目录
static const char homePage_[]; // 所指代的网页
static Cache& cache_; // 全局cache_
static int epollfd_;
int sockfd_; //该HTTP连接的socket和对方的socket地址
boost::shared_ptr<FileInfo> fileInfo_;
char readBuf_[READ_BUFFER_SIZE]; //读缓冲区
char postDataBuf_[READ_BUFFER_SIZE]; //请求附加内容
size_t nRead_; //标志读缓冲区中已经读入的客户数据的最后一个字节的下一个位置
size_t nChecked_; //当前正在分析的字符在读缓冲区中的位置
bool keepAlive_; //是否保持连接
bool sendFile_; //是否发送文件
char writeBuf_[WRITE_BUFFER_SIZE]; //写缓冲区
size_t nStored_; //写缓冲区中待发送的字节数
size_t written_; //已经写了多少字节
```

下面是 HttpServe 类的定义，该类定义了 HTTP 服务的根目录绝对路径、文件数据读取缓存；HTTP 请求到达之后的请求报文读取、uri 参数的解析、所请求文件的发送、错误信息的发送方法。

```
class HttpServe{
public:
    void init(int fd);
    void process();
    static void setEpollfd(int epollfd) {
```

```

        epollfd_ = epollfd;
    }
private:
    int processRead(); //读操作
    int processWrite(); //写操作
    void reset(); //重置
    bool addResponse(const char* format, ...);
    bool addResponse(char *const); //
    int getLine(char *buf, int maxsize); //在请求信息缓存中读取一行
    void static getFileType(char *fileName, char *fileType); //获取文件类型信息，用于制作响
        应头
    void sendErrorMsg(char *cause, char *errnum, char *shortmsg, char *longmsg); //错误信息
        发送
    void serveStatic(char *filename, size_t fileSize); //静态网页处理
    void serveDynamic(char *text, int len); //动态网页处理
    void readRequestHdrs(); // 读取请求头
    int parseUri(char *uri, char *fileName, char *cgiargs); //处理GET路径
    bool read();
};

```

在错误信息中，实现了 403、404、501 三个常见错误代码的响应，并且可以动态返回错误出现原因。实现了 html、css、jpg、png、otf、js、m3u8、ts 等文件的发送。

5.3 服务器的高性能优化

5.3.1 Epoll 的设计与实现

针对课程设计的需求，将 epoll 的操作封装为以下 5 个函数。

```

int setnonblocking(int fd);
void removefd(int epollfd, int fd);
void modfd(int epollfd, int fd, int ev);
void addfd(int epollfd, int fd, bool one_shot);
int Epoll_wait(int epfd, struct epoll_event* events, int maxevents, int timeout);
int Epoll_create(int size);

```

其中 Epoll_create 和 Epoll_wait 函数是对原有函数 epoll_create 和 epoll_wait 的简单封装，以便于在发生错误时输出错误信息。setnonblocking 函数用于其他函数调用，将文件描述符设置为非阻塞模式。removefd、modfd、addfd 三个函数的实现如下所示。

```

void addfd(int epollfd, int fd, bool oneShot){
    epoll_event event;

```

```

    event.data.fd = fd;
    event.events = EPOLLIN | EPOLLET; // ET触发
    if (oneShot)
        event.events |= EPOLLONESHOT; // 设置一次性监听
    epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &event);
    setnonblocking(fd);
}

void removefd(int epollfd, int fd){
    epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, 0);
    close(fd);
}

void modfd(int epollfd, int fd, int ev){
    epoll_event event;
    event.data.fd = fd;
    event.events = ev | EPOLLET; // ET触发
    epoll_ctl(epollfd, EPOLL_CTL_MOD, fd, &event);
}

```

addfd 函数用于向 epfd:epollfd 中注册新的 fd，第三个参数 bool 型 oneShot 用于确定是否需要将该 fd 设置为只监听一次事件。removefd 函数封装了 epoll 文件描述符的删除操作，modfd 用于修改已经注册过的 fd 的监听事件。所有要监听的事件均设置为 ET 触发以提高性能。

5.3.2 线程池的设计与实现

在本次课程设计中，通过参考 muduo 网络库，建立了一个如下所示基于生产者消费者模型的线程池类。

```

class ThreadPool{
public:
    typedef boost::function<void()> Task; // 需要执行的任务
    ThreadPool(int threadNum, int maxTaskNum);
    ~ThreadPool(){};
    bool append(Task&&); // 往工作队列中添加任务
    void run();
    static void* startThread(void* arg); // 任务线程
private:
    bool isFull();
    Task take();
    size_t queueSize();
    int threadNum_; // 线程的数目
    int maxQueueSize_;

```

```
std::list<Task> queue_; //任务队列
MutexLock mutex_;
Condition notEmpty_;
Condition notFull_;
};
```

在线程池接口处使用 boost 库提供的 function 函数代替传统的函数指针，以提供更大的灵活性，同时便于与 bind 函数配合使用，以取代虚函数。同时多线程环境下，由于向缓冲区中写入数据和从缓冲区内读取数据必须互斥进行，因此引入互斥锁类（定义于 mutex.hpp）完成线程的加锁、销毁操作。

如下所示，在构造函数中，包含两个条件变量用于线程调度，一个是 notEmpty_，一个是 notFull_，同时接受两个参数，一个是线程的数目，一个是最大的队列的大小。

```
ThreadPool::ThreadPool(int threadNum, int maxQueueSize)
    : threadNum_(threadNum)
    , maxQueueSize_(maxQueueSize)
    , mutex_()
    , notEmpty_(mutex_)
    , notFull_(mutex_)
{
    assert(threadNum >= 1 && maxQueueSize >= 1);
    /* 接下来构建threadNum个线程 */
    pthread_t tid_t;
    for (int i = 0; i < threadNum; i++) {
        Pthread_create(&tid_t, NULL, startThread, this);
    }
}
```

线程池模块最重要的外部接口 append 函数中，通过使用右值引用、move 语义实现任务向线程池的交付。

```
bool ThreadPool::append(Task&& task){
    { // 使用了右值引用
        MutexLockGuard lock(mutex_); // 首先加锁
        while (isFull())
            notFull_.wait(); // 等待queue有空闲位置
        assert(!isFull());
        queue_.push_back(std::move(task));
        //std::cout<<"put task onto queue!"<<std::endl;
    }
    notEmpty_.notify(); // 通知任务队列中有任务可做了
}
```

在调用时只需要实例化 ThreadPool 类，然后调用 append 函数传递任务至线程池即可，具体使用如下所示。由于建立线程数目过多反而会导致性能下降，因此一般将线程池内线程数量定为处理器核心数量，以达到最佳效果。

```
ThreadPool pools(4, 10000);
pools.append(boost::bind(&HttpServe::process, &handle[sockfd]));
```

5.3.3 Cache 的设计与实现

Cache 类本质上就是将需要访问的文件、数据从速度较低的磁盘中预先读取到速度较快的内存中，并且建立文件名到内存地址的映射，以实现查找所需文件时快速从内存中完成，减少 I/O 消耗。

下面给出 Cache 类的定义：

```
class FileInfo{
public:
    void *addr_; // 地址信息
    int size_; // 文件大小
    FileInfo(std::string& fileName, int fileSize); //读取文件并完成映射
    ~FileInfo(); //解除映射关系
};

class Cache{
private:
    std::map<std::string, boost::shared_ptr<FileInfo>> cache_; // 实现文件名称到地址的映射
    static const size_t MAX_CACHE_SIZE = 100; // 最多缓存100个文件
    MutexLock mutex_;
public:
    Cache() : mutex_() {}
    void getFileAddr(std::string fileName, int fileSize, boost::shared_ptr<FileInfo>& ptr)
        ; //优先查找cache中是否存在指定文件，不存在时才进行读取操作（需要线程锁）
};
```

5.4 本章小结

本章首先阐述了 HTTP 协议的相关内容，然后分析了 HTTP 报文的格式，最后给出了 HTTP 请求处理的实现。

第六章 系统测试

致 谢

参考文献

- [1] 卢誉声. 分布式实时处理系统：原理、架构与实现 [M]. 北京: 机械工业出版社, 2016, 103-106
- [2] D. P. B. . M. CESATI. 深入理解 linux 内核 [M]. 中国电力出版社, 2007
- [3] W. S. . S. A. Rago. Unix 环境高级编程 [M]. 人民邮电出版社, 2014
- [4] J. F. Kurose. 计算机网络自顶向下方法 [M]. 机械工业出版社, 2016

附 录