爱房视频点播系统方案调研报告

吕海涛 *

htlv@aifang.com

2012 年 3 月 20 日

*作者系安居客集团爱房网 PHP 开发工程师

摘 要

笔者深入研究主流视频点播系统，并最终选用苹果的超文本实时流媒体协议作为实验系统进行验证性研究。本文主要阐明了研究的主要工作，包括原始协议的研究与对比、视频切割工具、Flash 播放器、后端分布式存储大文件上传等几个部分。

目录

1 介绍 1

2 协议综述 2

2.1 HTTP 渐进下载 2

2.2 RTSP 协议 3

2.3 HTTP Live Streaming(HLS)协议 6

2.4 总结 9

3 TS 容器介绍 10

4 视频处理 12

4.1 转码 12

4.2 水印 14

4.3 切割 15

5 播放器 17

5.1 JW Player 介绍 17

5.2 JW Player adaptive 插件 17

5.3 JW Player adaptive 缺陷与改进 19

6 分布式网络存储 27

6.1 Ceph 介绍 27

6.2 Ceph 系统配置 27

6.3 Ceph 系统灾备 27

7 文件上传组件 28

7.1 综述 28

7.2 SWFUpload 组件 29

7.3 KUploader 组件 29

7.4 HTML5 断点续传组件 29

8 爱房视频系统展望 30

图 片

2-1 RTSP 协议族 4
2-2 RTSP 工作概览 5
2-3 HTTP Live Streaming 工作概览 6
3-1 TS 封包结构 10
4-1 视频水印示例 14
5-1 JW Player 播放 HLS 19
8-1 爱房视频系统架构 31

表 格

2-1 HTTP Live Streaming 中建议的 MIME 类型配置 8
2-2 流媒体协议比较 9

清 单

2-1 简单 M3U8 文件 7
2-2 M3U8 媒体列表文件 8
4-1 FFmpeg 转换参数 12
4-2 简单打水印参数 15
4-3 FFmpeg 切割脚本 16
5-1 简单 JW Player adaptive 配置 17
5-2 JW Player adaptive 日志类 20
5-3 JW Player adaptive PES 头信息错误 20
5-4 JW Player adaptive TS 构造函数 21
5-5 JW Player adaptive 视频片段处理流程 22
5-6 JW Player adaptive 视频片段解析 22
5-7 JW Player adaptive TS 类改动 23
5-8 JW Player adaptive Loader 类改动 24

如果你真的理解了,请整理一份文档!

§.1 介绍

爱房技术部经理阿胡突然有了一个想法——为爱房网做一个类似优酷、土豆一样的视频分享模块，这样开发商可以上传一些楼盘相关的视频。如此一来，用户可以通过点播视频来更加全面的了解和评估感兴趣的楼盘。相信，如果日后这样一个模块成功上线的话会给爱房网增色不少，而爱房网的找房体验也会大为改善。

阿胡找到我，跟我说了他的需求。而我呢，之前多少了解一些流媒体的知识，觉得没有太大的困难。于是就爽快地揽下了这份工作。谁曾想到，这是我恶梦的开始……

§.2 协议综述

阿胡跟我讲要做流媒体。那什么是流媒体呢？所谓流媒体就基于 IP 协议的视频分发技术。流媒体最显著的特点就是“边下载，边播放”，也就是说，客户端无需下载整个文件，而是一边播放缓冲区内已经下载的视频内容，一边不断地从服务器下载视频。流媒体还有另外一个重要特性，那就是实时性和连续性。一般而言，客户端可以对视频流进行快进、跳拨等操作。因为流媒体的实时性，流媒体又可以应用到电视节目直播，视频会议等现实场景中。

那为什么要选流媒体呢？我想，一个很重要的原因就是阿胡考虑到用户体验和网络带宽成本。如果采用流媒体解决方案，用户点播视频的时候就无需等待整个视频文件都下载完成，一定程度上曾强了用户体验；而用户在跳播的时候也不会下载无关的视频内容，从而节省了带宽。

好了，接下了就介绍一下主流的流媒体技术。

2.1 HTTP 渐进下载

HTTP 渐渐下载，顾名思义，就是一种基于 HTTP 协议的下载技术。严格来讲，这种技术并不能算作流媒体技术。但形式上，也达到了“边下载，边播放”的效果，而且是应用最广泛，也是最成熟的技术。所以我们在这就多费些笔墨。

众所周知，传统的下载播放模式必须等待整个视频文件都下载完毕才能开始播放。基于 HTTP 协议的渐进式下载技术则在此基础上做了小小的改进，即渐进式下载客户端在开始播放前仅仅需要一小段时间下载视频文件最开始的一步数据，在此之后就可以便下边播了。

其中的原理也并不复杂。开始播放之前的下载动作非常重要，因为播放器通过这次下载拿到了视频文件的头信息，里面包含了足够了的信息来初始化解码器。之后的视频内容则可以一边下载，一边解码(播放)了。

在这里特别介绍一下 HTTP 协议里面的 Range [1] 请求指令。通过这个字段，服务器可以只传输一部分资源给客户端。好多下载工具就是通过这个字段实现断点续传功能的。客户端在请求服务器资源的时候加入 Range 字段，例如 Range:bytes=1000-2000，即客户端通知服务器只需传输第 1000 字节到 2000 字节的内容。如果服务器支持

[1]更多细节请参阅 http://tools.ietf.org/html/rfc2616

Range 字段，则会在相应头中加入 Accept-Ranges 字段，字段的内容是 bytes。服务器返回部分内容的时候会加入字段 Content-Range。例如，如果服务器返回 Content-Range:1000-3000/5000，则意味着服务器返回的是长度为 5000 字节内容的第 1000 字节到第 2000 字节的内容。

前面说过，HTTP 渐进下载是当前应用最广泛的视频点播技术。优酷、土豆等网站采用的也是这种技术。我们注意到，优酷等网站在播放视频的时候并非全部缓存视频内容。而是设置了一个时间，比如，只缓冲 5 分钟。经过我们抓包发现，他们利用的正是 Range 字段来实现这样的功能。

在这种模式下，客户端以自己以及 web 服务器和网络所能允许的最大速度尽可能快的从服务器下载资源，这给会给服务器造成很大的压力。大学里的同学应该理解，寝室局域网如果有人看视频，其他人就不用上网了。

在渐进下载模式下，客户端需要在硬盘上存放缓冲数据。播放过程中，用户只能在已经缓存的部分进行进度条搜索和快进等操作，而无法在整个视频文件时间范围内执行这写操作。

这种模式还有另外一个缺点，即视频内容可以轻易被用户获取。用户点播完成后，只需通过检查浏览器缓冲区，就可以拿到下载后的视频文件，这对视频版权的保护来讲，无疑是一个严重的缺陷。

2.2 RTSP 协议

RTSP[2] 全称 Real Time Streaming Protocol，也就是实时流协议。制定 RTSP 协议的主要目的是为娱乐和通信系统提供流媒体控制。它的意义在于使得实时流媒体数据的受控和点播成为可能。

RTSP 是一个流媒体表示协议，主要用来控制具有实时特性的数据，但是它本身不传输任何数据，而必须以来下层的传输协议进行。

RTSP 协议工作在应用层，为视频等实时数据传输领域提供了可扩展的控制框架。RTSP 使用段口号 554，即可以工作在 UDP 上，也可以工作在 TCP 上。

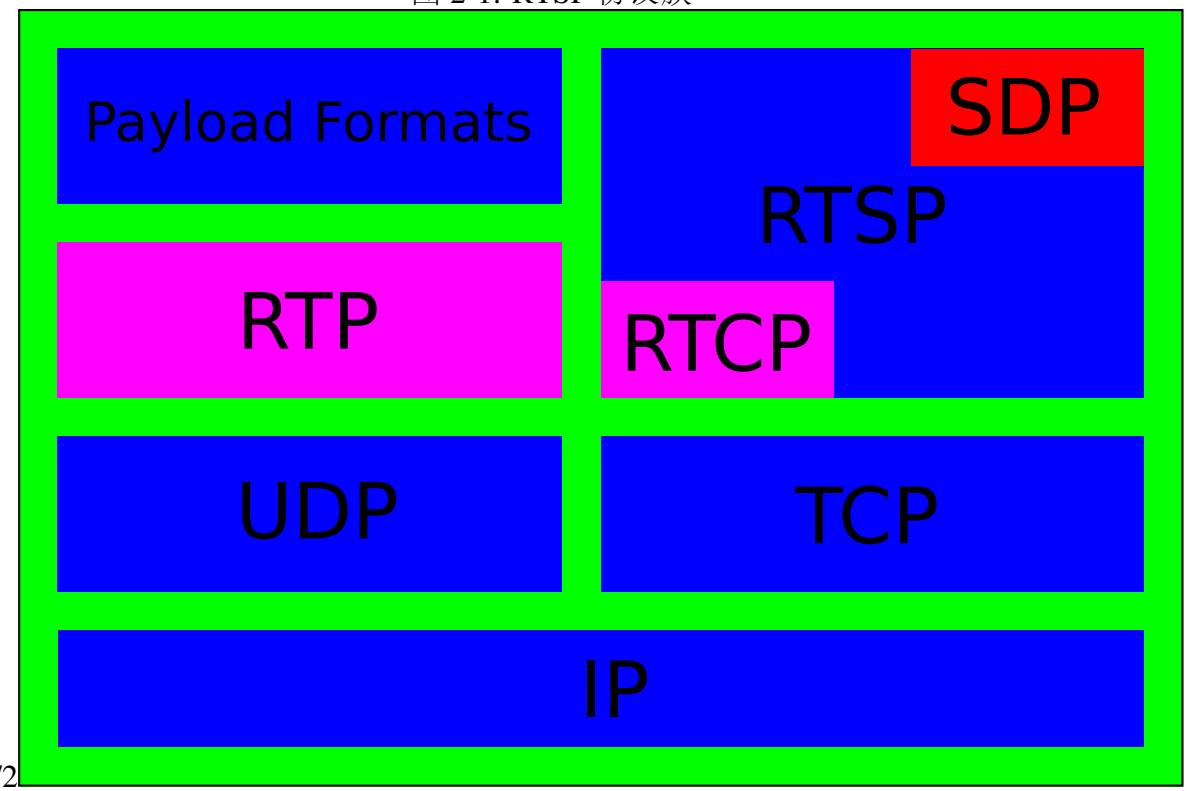
值得注意的是，单纯的 RTSP 协议无法实现流媒体播放服务。RTSP 协议需要其他协议配合才能正常工作。图 2-1展示了相关的协议族。

RTP[3]，全称是 Real-time Transport Protocol，也就是所谓的实时传输协议。RTP 为

[2]大家可以参考http://tools.ietf.org/html/rfc2326

[3]http://en.wikipedia.org/wiki/Real-time_Transport_Protocol

图 2-1: RTSP 协议族



/2/2

IP 网络上的音视频分发工作定义了标准化的封包格式。在诸如电话、视频电话程序、网络电视服务和基于 web 的推送聊天等基于流媒体的通信和娱乐系统中，RTP 有着广泛的应用。

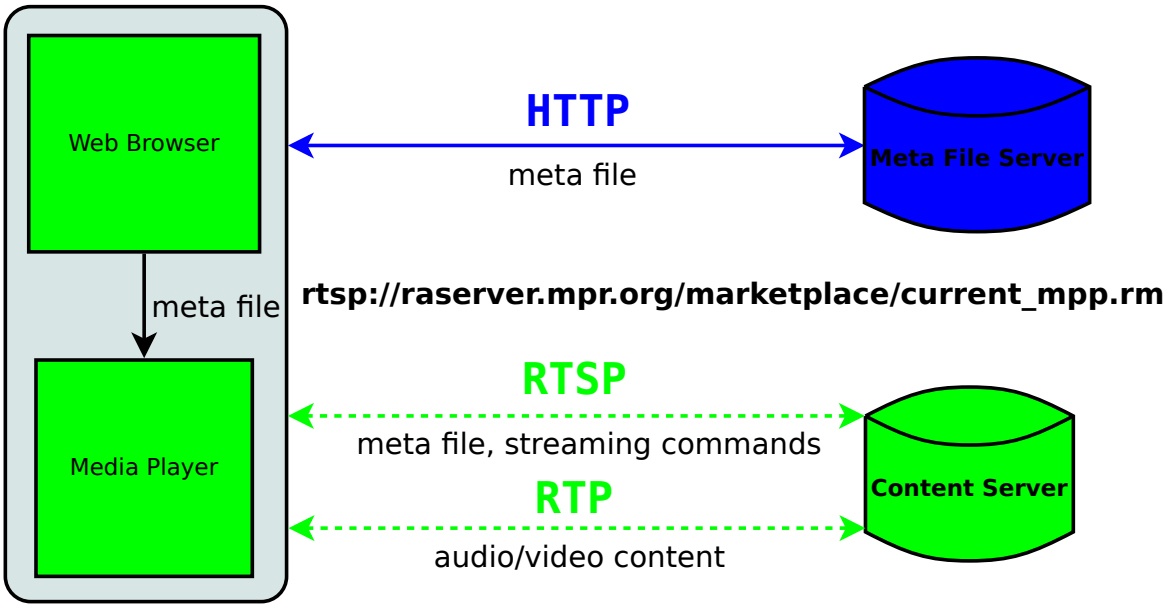
RTP 需要 RTCP(RTP Control Protocol，即 RTP 控制协议)协同工作。RTP 仅仅承担媒体数据传输任务，而 RTCP 负责监控传输过程，同步不同的媒体流，提供 QoS 控制等功能。

介绍完相关的协议，我们来分析 RTSP 的工作过程。图 2-2清楚地说明了 RTSP 流媒体的工作流程。

首先，播放器通过浏览器(或者直接)从服务器获取视频流的元信息。然后，客户端通过 RTSP 协议与服务器通信，开始请求视频流内容。真正的媒体内容由 RTP 协议和 RTCP 协议协同传输给客户端。客户端开始播放媒体流。诸如暂停、快进等所有的控制信息均通过 RTSP 协议交互。

基于 RTSP 协议的流媒体系统需要专门的流媒体服务器参与工作。与渐进下载中媒体数据的被动突发传送不同，在有流媒体服务器参与的媒体分发过程中，媒体数据

图 2-2: RTSP 工作概览



是以与压缩的音视频码率相匹配的速率主动地、智能地发送的。这样就大大提高了网络的利用率。

实现 RTSP 协议的服务端软件和客户端软件有很多，常见的服务端软件有：

- QuickTime 流媒体服务器，苹果公司开发的流媒体服务器
- Darwin 流媒体服务器，QuickTime 流媒体服务器的开源实现
- Helix 流媒体服务器，RealNetworks 公司开发的流媒体服务器
- Live555 流媒体服务器，C++ 开发的开源流媒体服务器

而支持播放 RTSP 流的客户端则更多，如开源的 FFmpeg、MPlayer，苹果公司的 QuickTime，微软的 Windows Media Player 等等。

RTSP 流媒体协议栈相对比较复杂，需要专门的服务器参与，对硬件要求比较高，无法进行 CDN 分布式缓存。基于 RTSP 协议的流媒体系统无论是在前期部署还是在后期维护，都有不小的难度。一般而言，基于 web 的视频点播系统很少采用 RTSP 实现。

2.3 HTTP Live Streaming(HLS)协议

HTTP Live Streaming 最初是苹果公司针对其 iPhone、iPod、iTouch 和 iPad 等移动设备而开发的“流媒体”[4]协议，后来苹果系统中的 QuickTime 播放器也添加了相应的支持[5]。HTTP Live Streaming 允许内容提供者仅仅通过普通的 Web 服务器向支持的客户端发布接近实时的音视频流媒体服务，包括点播和直播。HTTP Live Stream 支持为同一节目提供不同码率的媒体流，客户端可以根据网络带宽的变化在这些不同码率的媒体流之间进行智能切换。此外，HTTP Live Streaming 还支持媒体流加密，可以很好的保护版权。到目前为止，HTTP Live Streaming 已经成为 IETF 的标准草案[6]。

图 2-3: HTTP Live Streaming 工作概览

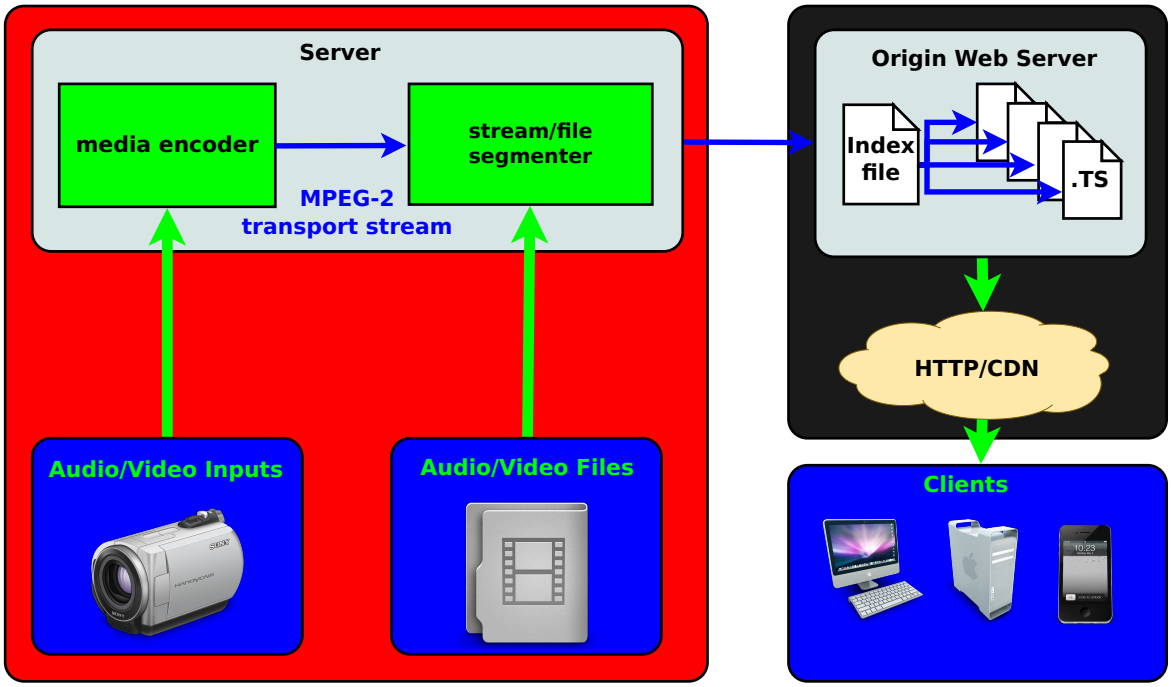


图 2-3是 HTTP Live Streaming 整体架构。HTTP Live Streaming 由三部分组成：服务器组件、分发组件和客户端。

服务器组件 负责将输入的媒体流进行数字编码，以适合发布的格式进行封装。图 2-3中红色大框部分。

[4]此处用使用引号，是因为 HTTP Live Streaming 并非传统意义上的流媒体。
[5]Windows 平台的 QuickTime 貌似不支持呀。
[6]参见 http://tools.ietf.org/html/draft-pantos-http-live-streaming-07

分发组件 由标准的 Web 服务器组成，负责接受客户端的请求，并且将准备好的媒体内容和相关资源发送到客户端。图 2-3中黑色大框部分。

客户端 负责请求合适的媒体内容，下载媒体资源，并将媒体内容重新组装，以连续的媒体流形式展现给用户。图 2-3中蓝色大框部分。

在一个典型的系统中，硬件编码器接受音视频输入，使用 MPEG-4 编码，输出 MPEG-2 TS 流，然后被分段软件划分为一系列简短的时间片段并保存成媒体文件。这写媒体文件片段将被部署到 Web 服务器上。分段软件还将维护所有媒体片段的索引文件。索引文件也会发布到 Web 服务器上，客户端根据索引文件请求对应的媒体文件。

媒体流分段器通常是一个软件，从本地网络读入 TS 流并将它分成一系列相等时间的媒体文件。即使每段都是一个单独的文件，从连续媒体流中生成的视频文件仍然可以被无缝的重构。分段器同时会创建一个包含各媒体文件引用的索引文件。每当分段器完成一个新的媒体文件，索引文件就会被更新。索引文件被用来确认媒体文件的可用性和位置。分段器也可以加密每个媒体段并创建一个密钥文件。

媒体分段被保存为.ts 文件（MPEG-2 媒体流），索引文件被保存为.M3U8 文件，这是保存 MP3 播放列表的.m3u 格式的一种扩展。

清单 2-1: 简单 M3U8 文件

```
1  #EXTM3U
2  #EXT-X-TARGETDURATION:10
3  #EXTINF:10,
4  http://hello.htlv.dev.aifang.com/cyzn/cyzn-1.ts
5  #EXTINF:10,
6  http://hello.htlv.dev.aifang.com/cyzn/cyzn-2.ts
7  #EXTINF:10,
8  http://hello.htlv.dev.aifang.com/cyzn/cyzn-3.ts
9  #EXTINF:10,
10 http://hello.htlv.dev.aifang.com/cyzn/cyzn-4.ts
11 #EXT-X-ENDLIST
```

清单 2-1是一个非常简单 M3U8 文件示例。该示例包含了 3 段未加密的 10 秒媒体文件。行 1表明索引文件的开始。行 2表明索引文件中视频的最长时间为 10 秒。行 3表明接下来的视频片段时长大约为 10 秒。行 4指出了一个视频片段的 url。行 11则是索引文件的结束符。如果要了解索引文件的更多细节，请参阅 IETF 互联网草案[7]。

媒体分段文件通常由媒体流分段器基于编码器的输出产生，由一系列视频编码是

[7] http://tools.ietf.org/html/draft-pantos-http-live-streaming-07

H.264 以及音频编码是 AAC 或者 MP3 的 MPEG-2 TS 流的.ts 文件组成。

分发系统可以是一个 Web 服务器或者 Web 缓存系统，基于 HTTP 协议发布媒体文件和索引文件。不需要定制的服务器模块，只需要对 Web 服务器进行很少的配置。推荐的配置通常只限于指定.M3U8 文件和.ts 文件的 MIME 类型关联，具体见表 2-1。

表 2-1: HTTP Live Streaming 中建议的 MIME 类型配置

文件扩展名	MIME 类型
.m3u8	application/x-mpegURL 或者 vnd.apple.mpegURL
.ts	video/MP2T

因为这些文件会被频繁的更新，每个请求下载的应该是最新版本，所以在服务端不应该开启任何针对 M3U8 文件的缓存设置。

客户端从获取索引文件开始，并基于 URL 来标识一个媒体流。索引文件指定了可用媒体文件的位置，解密的密钥，以及可切换的媒体流。对于选定的媒体流，客户端按顺序下载每个可用的媒体文件。每个文件包含媒体流的一个连续的段。一旦足够的数据下载完毕，客户端开始向用户播放重新组合的媒体流。

对于加密的媒体流，客户端负责获取解密密钥，验证用户，或者按需要显示用户验证和解密媒体文件的用户界面。

以上过程将一直重复直到遇到索引文件中的 #EXT-X-ENDLIST 标签。如果没有遇到 #EXT-X-ENDLIST 标签，则认为该索引文件是正在进行的广播的一部分，客户端会周期性的重新加载新的索引文件。客户端会在更新后的索引文件中查找新的媒体文件以及加密密钥并将这些 URL 加到播放列表中。

值得一提的是，HTL 系统具有根据网络情况动态切换媒体质量的能力。

清单 2-2: M3U8 媒体列表文件

```
1 #EXTM3U
2 #EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=96000
3 http://hello.htlv.dev.aifang.com/ia/96/ia.m3u8
4 #EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=256000
5 http://hello.htlv.dev.aifang.com/ia/256/ia.m3u8
6 #EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=800000
7 http://hello.htlv.dev.aifang.com/ia/800/ia.m3u8
```

清单 2-2是一个简单的媒体列表文件，其中包含了 3 段不同质量的媒体索引文件。行 2的 #EXT-X-STREAM-INF 字段定义了一个媒体索引文件，PROGRAM-ID=1 定义

节目 ID, 相同节目的不同媒体索引文件使用相同的 ID, BANDWIDTH=96000 指定最小带宽需求。行 2需要 96kbps 的带宽。当然, 如果网络状况良好, 客户端会自动切换到更高质量的媒体流, 像行 6(800kbps)。

2.4 总结

我们用一个比较表格来结束本节。三种“流媒体”技术优缺点比较见 表 2-2。

表 2-2: 流媒体协议比较

	渐进下载播放	RTSP/RTP	HTTP Live Streaming
服务器端实现	普通 Web 服务器	流媒体服务器	普通 Web 服务器
客户端实现	容易	较难	容易
系统安装、配置	简单	复杂	简单
支持业务类型	点播	直播、点播	直播、点播
实时性 (起始时延)	>30s	<2s	约 30s(推荐配置下)
客户端缓冲区	硬盘, 文件大小	内存, 小	内存, 较小
VCR 支持	部分支持	支持, 定位精度高	支持, 定位精度略差
网络带宽适应	不支持	部分服务器支持	支持 (灵活流间切换)
服务器故障保护	不支持	不支持	支持
DRM 支持	差	较好	好
防火墙穿透性	好	差	好
网络层多播	不支持	支持	不支持
适用范围	低码率短视频 (如广告、片花等)	大规模、低延迟实时流媒体系统	嵌入式移动流媒体 (实时性要求不是太高)

§.3 TS 容器介绍

HLS 系统要求媒体内容（音频或者视频）必须以 TS，也就是所谓的 MPEG Ttansport Stream 封装。这是为什么呢？前面已经说过，HLS 系统的核心就是将媒体文件(流)切割成小段文件(推荐 10s 一段)。这样一方面能够实现“流媒体”效果，另一方面可以方便分发媒体内容。但是，这样的做法就自然要求切割出来的媒体片段必须能够独立播放。而 MPEG-2 国际标准中定义的 MPEG TS 容器正好能够满足我们的要求。

那 MPEG TS 又是怎么做到的呢？MPEG TS 是一种非常成熟的媒体封装格式，主要应用于广播电视领域。你可能已经注意到了，无论在什么时候打开电视，我们总是能够看到完整的视频画面。没得说，这是 MPEG TS 的功劳。

MPEG TS 使用固定长度的封包来承载媒体数据。不妨我们就叫这种特殊的封包为 TS 封包。TS 封包最明显的特征就是固定长度。所有的 TS 封包长度都是 188 字节。当然，也有例外。DVB-S 技术对 MPEG TS 做了扩展，其数据封包的长度为 206 字节。这个跟文章的主题无关，不多做介绍。至于为什么是 188 字节，而不是其他数目，这是为了兼容早期的 ATM 网络[8]。MPEG TS 直接封装 PES 媒体数据，然后交付通信链路传输。先给个图吧。不说说 PS 封包的具体格式就无法解释 MPEG TS 流的特点。

图 3-1: TS 封包结构

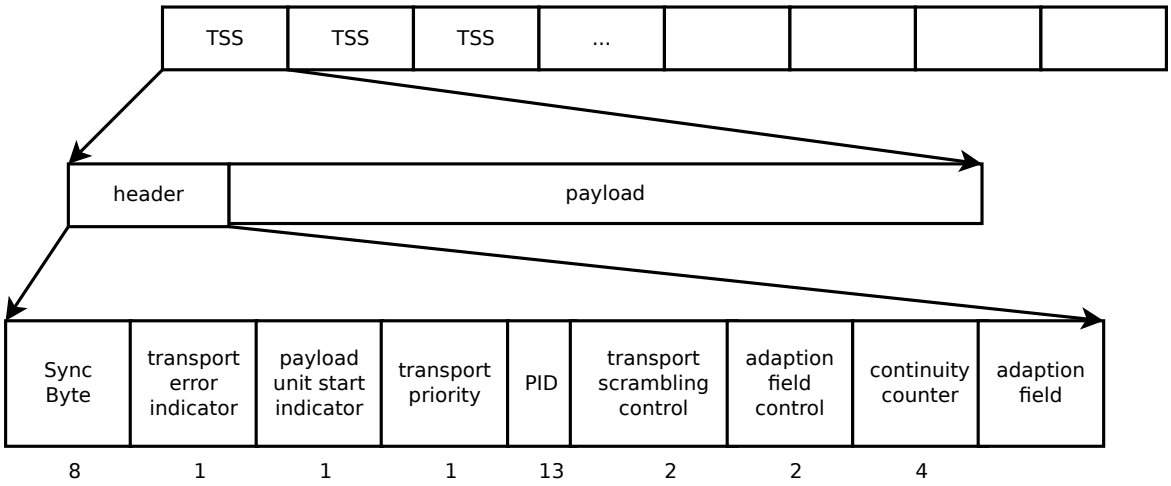


图 3-1是 TS 封包的简化图。一个 TS 封包分为首部和载荷两个部分。TS 封包头

[8]此处不是很确定，需要超找出处！

部又如下字段:

- 封包开始 8 位为 0x47,表示 TS 封包的同步字段。
- 接下来是 3 个标志位,
  1. 第一个是传输错误,解码器会忽略此标志位为 1 的封包。
  2. 第二个是 PES 封包开始标志,此位为 1 时表示 PES 数据包开始;如果为 0,则表示此 TS 包的载荷是上一个数据包的一部分。
  3. 表示传输优先级。
- 接下来是 13 位长的 PID,用来标识数据包的作用。
- 后面的两位用来控制 TS 封包的加密属性。
- 在后面的两位表示附加字段属性。
  1. 01 表示没有附加字段,只有净载荷。
  2. 10 表示只有附加字段。
  3. 11 表示附加字段后面跟着净载荷。
  4. 00 保留以后使用。
- 最后是半字节的 Continuity Counter。

在我们现在的系统中,我们会忽略附加字段和加密相关的字段。如果 TS 封包只含有附加字段,此封包将被丢弃。

MPEG TS 提供了简单的容错机制,即解码器自动忽略有问题的 TS 封包。MPEG TS 具有同时传输多路媒体节目的能力。而有线电视系统也正是使用了 MPEG TS 格式。为了同时传输多路节目,MPEG TS 会定时传输节目服务信息(PSI)。PSI 包括节目关联表(PAT),节目映射表(PMT)。

PID 为 0 的 TS 封包携带 PAT,描述了不同节目所用 PMT 的 PID。更具 PMT 记录的 PID,可以找到特定节目的 PMT,并从中查到不同媒体流(视频、音频、字幕等)所用的 PID;属于同一媒体流的 TS 封包 PID 相同。这样就可以将 TS 封包解码成 PES 封包了。PSI 没隔一定时间会发送一次,保证数据传输失败时可以尽可能快的恢复解码。这也是为什么 MPEG TS 能够很方便地进行切割的原因。

HLS 要求 TS 流中只能包含一路节目,这就大大降低了客户端的复杂度。如果要实现不同节目直播,只能使用多个流文件了。

§.4 视频处理

本节将介绍媒体文件的准备工作,包括转码、水印、和切割三个部分。在这里不得不提 FFmpeg。单凭 FFmpeg 就可实现上述功能。FFmpeg 是一座富矿,水很深。详细研究的话一定会有很大的收获。

4.1 转码

HLS 要求视频编码为 AVC,也就是传说中的 H.264,音频编码可以是 AAC 或者是 MP3。使用 FFmpeg 可以很轻松的完成这一人物。需要注意的是如果你自己编译 FFmpeg,一定要开启 --enable-libx264 选项来支持 H.264 编码。

视频转码主要解决了两个问题:

1. 编码格式。视频用 H.264,音频用 MP3 或者 AAC
2. 文件大小。为保证带宽为 1M 的用户能够流畅观看视频,视频片段(10s)尺寸不应该超过 2M 字节。

音频部分体积不会太大,可以不用过多的关注。只要能够视频部分的尺寸就可以了。为此,我们不得不调整 H.264 编码器的相关参数。

清单 4-1: FFmpeg 转换参数

```
#!/bin/bash
ffmpeg -i input.mp4 \
    -c:a libmp3lame -r:a 48k -b:a 64k \
    -c:v libx264 -b:v 800k -r:v 15 \
    -bsf:v h264_mp4tonnexb \
    -bf 0 \
    -s vga \
    -flags +loop -cmp +chroma \
    -partitions +parti4x4+partp8x8+partb8x8 \
    -subq 5 -trellis 1 -refs 1 -coder 0 \
    -me_range 16 -keyint_min 25 \
    -sc_threshold 40 -i_qfactor 0.71 \
    -bt 200k -maxrate 800k \
    -bufsize 96k -rc_eq 'blurCplx^(1-qComp)' \
    -qcomp 0.6 -qmin 10 -qmax 51 \
    -qdiff 4 -level 30 -g 30 -async 2 \
    -f mpegts output.ts
```

清单 4-1展示了我们所用的 FFmpeg 参数。看着就头大。没有办法,FFmpeg 的相关文档实在太少。清单 4-1是在网络上搜刮而来的各个版本的基础上稍作调整得到

的。下面说说我能看懂的参数的含义：

- 行 2 中的 -i 参数制定了输入文件的路径。这个貌似很显然。
- 行 3设置输出文件的音频参数。在这里，
  1. -c:a libmp3lame 指定音频编码器为 MP3。
  2. -r:a 48k 指定了音频采样率为每秒 4 万 8 千次。
  3. -b:a 64k 则设置了音频的波特率为 64kbps。
- 行 4不用说，设置了输出文件的视频参数。
  1. -c:v libx264 指定视频编码器为 AVC(也就是传说中的 H.264)。
  2. -r:v 15 设置视频文件每秒有 15 帧。
  3. -b:v 800k 则设置了视频的波特率为 800kbps。
- 行 5是一个比较关键的参数。之所以加这个参数，是因为 H.264 有两种格式的比特流——一种叫做 length prefixed 模式，一种叫做 start code 模式。反正我是不知到这两种模式的区别，有兴趣的查标准文档吧。但是，有一点可以肯定，普通 MP4 封装的 H.264 比特流是 length prefixed 模式的，而 MPEG TS 封装的 H.264 比特流是 start code 模式的。所以我们需要开启 h264_mp4tonnexb 此参数。
- 行 6又是一个很关键的参数。为什么说它关键，是因为她解决了困扰我们好久的视频马赛克问题。我们采用的 flash 播放器用很老的 flv 格式封装媒体文件，但是 flv 无法很好的处理 B 帧，所以导致马赛克的产生。设置 -bf 0 参数就是告诉 FFmpeg 不要生成 B 帧[9]。
- 行 7设置输出视频的分辨率。可以使用代号，像 vga 表示 640x480 分辨率，wxga 表示 1366x768 分辨率。当然，你也可以直接指定具体的分辨率。值得注意的是，FFmpeg 默认会保持图像的纵横比例，所以光设置 -s 参数有可能得不到你想要的分辨率。如果需要调整纵横比例，请参阅 aspect 参数。
- 行 8到 行 16则设置了 H.264 编码的相关参数。说实话，好多参数我也不知道是什么意思。如果真相弄明白，恐怕只有查阅 H.264 标准文档了。这一堆参数基本保证了产生的视频文件体积较小，清晰度较高。
- 行 17设置输出文件相关参数。其中 -f mpegts 告诉 FFmpeg 输出一个 MPEG TS 格式的文件。FFmpeg 会根据文件扩展名猜测所用的容器格式，所以一般无需设

[9]B 帧在 H.264 系统中极为重要，是保证视频质量，减小文件尺寸的关键。去掉 B 帧会影响压缩效果。我们现在这样做只是权宜之计。解决问题的根本之法在于重写 flash 播放器逻辑，采用支持 B 帧的 f4v 或者 mp4 封装媒体数据。

置 -f 参数。

以上命令可以将常见 MP4 文件转换成我们需要的 MPEG TS 文件。经过测试，可用性比较高。为了降低系统的复杂度，我们暂时只支持 MP4、MPG、MPEG、M4V 等后缀的视频格式。其他格式的视频支持将陆续添加。

4.2 水印

网络上传播的视频难免涉及版权问题。最简单明了的方法就是打水印。HLS 本身支持媒体流加密，奈何还没有研究清楚。目前就先采用“经典”做法。打水印工作依然由 FFmpeg 完成(膜拜呀)。不管三七二十一，先上王道，一睹为快。

图 4-1: 视频水印示例

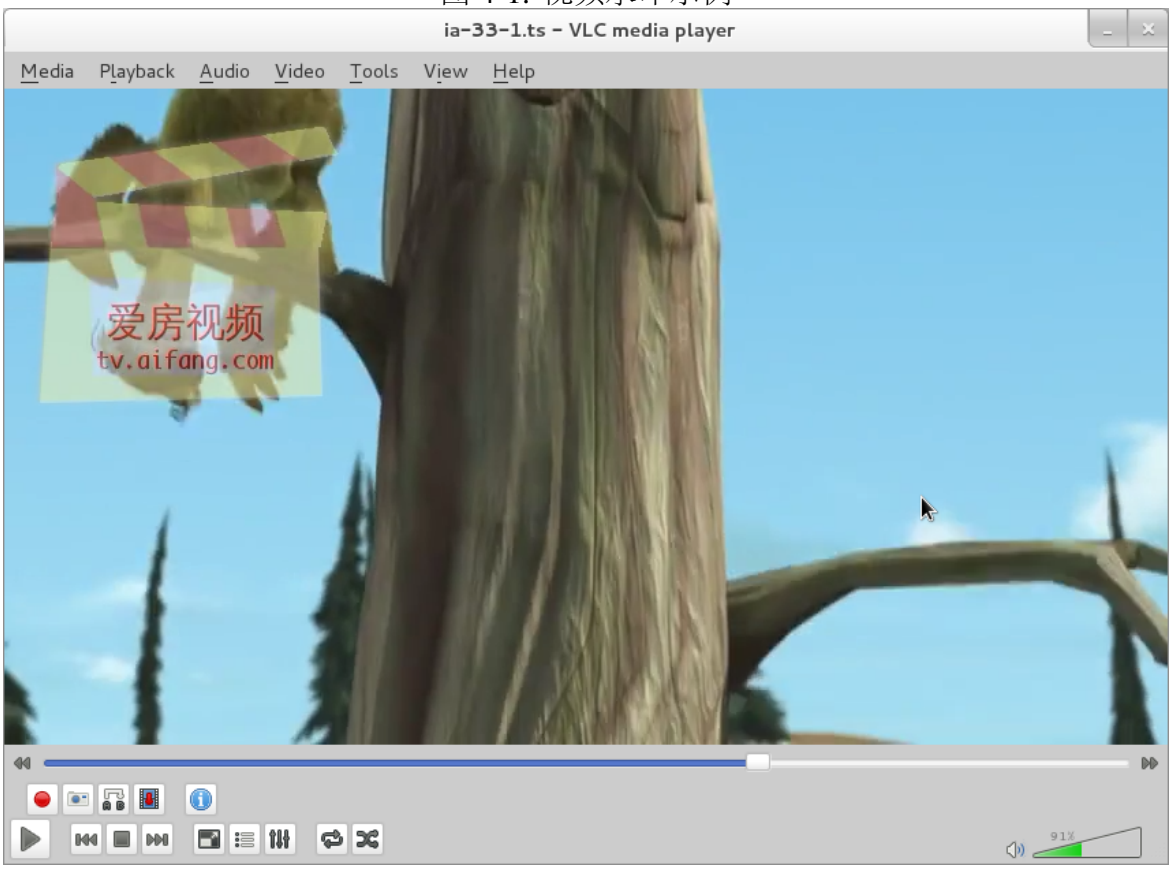


图 4-1是打好水印的视频截图。大家可以看到，画面的左上角有半透明的爱房视频标志。那到底怎么做呢？嘿嘿，咱么言归正传。

FFmpeg 使用 libavfilter 库来完成添加水印的相关工作，而老版本中的 -hook 已经废弃。要给视频打水印的话只要设置 -vf 的参数值就可以了。

清单 4-2: 简单打水印参数

```
1 #!/bin/bash
2
3 ffmpeg -i 801/ia-33.ts \
4     -vf "movie=aifang_video.png␣[wm];␣\␣
5         [in][wm] overlay=5:5 [out]" \
6     -c:v libx264 -c:a aac
7     -strict -2 ia-33-1.ts
```

清单 4-2展示了我们所用的 FFmpeg 配置。行 4的 movie 参数指定水印所用的图片。此处我们使用文件名为 aifang_video.png 的图片作为水印。后面紧跟的 [wm] 可以理解为水印的代号。

特别需要注意的是行 6。FFmpeg 在打水印的时候默认会将视频编码转换成 mpeg2video, 将音频编码转换成 mp2。所以我们需要手动指定视频和音频编码器。行 5的 [in] 和 [out] 代表输入媒体流和输出媒体流。行 5的 ouverlay=5:5 指定了水印的位置。5:5 表示水印离画面左边和顶部各 5 个像素。

FFmpeg 还提供了几个变量供用户使用, 他们是

- main_w 视频画面宽度
- main_h 视频画面高度
- overlay_w 水印宽度
- overlay_h 水印高度

这样, 我们就可以方便的在视频画面上放置水印:

- overlay=10:10 左上角
- overlay=main_w-overlay_w-10:10 右上角
- overlay=10:main_h-overlay_h-10 左下角
- overlay=main_w-overlay_w-10:main_h-overlay_h-10 右下角

我知道的就这么多。其他的请自行谷歌吧[10]。

4.3 切割

苹果公司为 HLS 技术提供了相应的切割工具, 其中 mediafilesegmenter 用来切割视频文件, 而 mediastreamvalidator 用来切割视频流媒体。然而不幸的是, 这两个工具只对苹果的注册开发者开放使用, 而且只能运行在苹果的系统上。

[10] 这里提供一个给力的网页, 供大家参考。
http://www.idude.net/index.php/how-to-watermark-a-video-using-ffmpeg/

网上有牛人根据 HLS 标准草案实现了一个开源的切割工具，名为 segmenter [11]。这个程序可以很好地切割.ts 文件。但是，这个小程序是基于 libav 的早期版本的，里面用到的好多宏定义和方法都已经废弃了。如果自己编译的话会遇到不少问题。不怕折腾的朋友自己去研究吧。不管你用不用，反正我是不用它了。

后来，在使用 FFmpeg 的过程中偶然发现，FFmpeg 也可以切割.tx 文件，而且可以做到无损切割(怎么又是 FFmpeg!!!)。这个实在是太强大了。果断抛弃 segmenter!

清单 4-3: FFmpeg 切割脚本

```
1  #!/bin/bash
2
3  BASE_URL=http://hello.htlv.dev.aifang.com/ztw
4  DURATION=10
5
6  ffmpeg -i ztw.ts \
7    -c copy \
8    -map 0 \
9    -f segment \
10   -segment_time 10 ztw-%d.ts
11 echo '#EXTM3U'
12 echo '#EXT-X-TARGETDURATION:'.$DURATION
13 FILES=$(ls -1|grep "-"|sort -t"-" -k 2 -n)
14 for i in $FILES;do
15   real_duration=$(ffmpeg -i $i 2>&1|\
16   grep Duration|awk '{print $2}'|awk -F\, '{print $1
       }'|\
17   awk -F\: '{print $1*3600+$2*60+$3}')
18   echo '#EXTINF:'$real_duration,
19   echo $BASE_URL/$i
20 done
21 echo '#EXT-X-ENDLIST'
```

清单 4-3是一个利用 FFmpeg 切割 MPEG TS 文件(其实可以切割任意文件!)的示例脚本。行 9是这个脚本的亮点。-f segment 参数告诉 FFmpeg 将输出文件进行切割，而行 10则进一步指明每隔 10 秒钟切割一次。读者可能注意到 ztw-%d.ts。%d 表示输出片段按整数编号。有人喜欢用%03d 来加入前导零，这样做的主要考虑是为了方便排序。其实这是没有必要的。行 13所示的排序可以解决无前导零情况下的文件名排序问题(10 不会跑到 2 前面去)。

另外，因为切割不涉及到转码，所以整个过程非常快。FFmpeg 不能自动生成索引文件，所以需要后面的几行的代码。都很简单，不做赘述。

[11]源代码地址 http://svn.assembla.com/svn/legend/segmenter/segmenter.c

§.5 播放器

终于写到播放器了,伤不起呀。

HLS 是苹果推出的技术,在苹果的移动设备上得到原生的支持。苹果 PC 平台的 QuickTime 同样支持 HLS 回放。可是,其他平台呢?Wiondws、Android、Linux 呢?

很显然,我们需要一个通用的客户端来回放我们的媒体流。达成这一目标的最好的方法是使用 Javacript 和 HTML5,但鉴于 HTML5 种种不给力之处,HTML5 还无法胜任这一艰巨的任务。自然,我们想到了 Flash。

虽然 Flash 就为业界所诟病,但其灵活的二进制数据处理能力、出色的视频播放能力让人难以抗拒。我们可以看看有没有开源项目能让 Flash 播放 HLS 媒体流。最坏的情况也只是用 ActionScript 实现一个 HLS 客户端。所幸的是世界上已经有了 JW Player。

5.1 JW Player 介绍

JW Player 号称网络上最流行的开源视频/音频播放器。JW Player 支持 Flash 能够播放的以及 HTML5 支持的所有类型的媒体文件,包括 FLV、H.264、MP4、VP8、WebM、MP3 等等。JW Player 还支持多种流媒体格式,包括 RMTP、HTTP、HLS(亮点在这里)。著名视频网站 YouTube 用的就是 JW Player 播放器。

JW Player 拥有众多的插件来扩展其自身的功能,其中对我们来说最重要的摸过于 adaptive 插件。正是 adaptive 插件为 JW Player 带来了 HLS 回放功能。

5.2 JW Player adaptive 插件

我们先看看 JW Player adaptive 插件是怎样配置的。

清单 5-1: 简单 JW Player adaptive 配置

```
1 <html>
2 <head>
3     <meta http-equiv="content-type" content="text/html
        ;charset=UTF-8" />
4     <script type="text/javascript" src="assets/
        jwplayer.min.js"></script>
5     <title>爱房网视频点播示例</title>
6     <style>
7         body { padding: 50px;
```

```
 8              font: 13px/20px Arial;
 9              background: #EEE;
10          }
11          form { margin-top: 20px; }
12          #player { -webkit-box-shadow: 0 0 5px #999;
13              background: #000;
14          }
15          ul { margin-top: 40px;
16              padding: 0 0 0 20px;
17              list-style-type: square;
18          }
19          h1 { color: #ED1C24;
20              margin:auto;
21              width:640;
22              text-align:center;
23          }
24      </style>
25  </head>
26  <body style="margin:auto;␣width:640;">
27      <h1>爱房网视频点播示例</h1>
28      <video id="player" width="640" height="360" src="
            ia/ia.m3u8">
29          You need Flash to play these tests
30      </video>
31
32      <script type="text/javascript">
33          if (!/(iPhone|iPad|iPod)/i.test(navigator.
                userAgent)) {
34              jwplayer("player").setup({
35                  file: 'ia/ia.m3u8',
36                  flashplayer: 'assets/player.swf',
37                  provider:'assets/
                        adaptiveProvider_debug.swf',
38                  height: 360,
39                  width: 640
40              });
41          }
42      </script>
43  </body>
44  </html>
```

行 4引入所需要的 js 库，jwplayer.min.js。为了兼容移动 iOS 设备，我们需要设置一个 video 标签，如行 28所示。video 的 src 属性指明索引文件的 url，iOS 设备即可原生播放 HLS 流媒体。

对于其他平台，则需要 Flash 支持了。jwplayer.min.js 体统的 API 类似 jQuery。将视频容器的 id 传个 jplayer，然后调用 setup 方法设置播放器参数，见行 34。setup 方法

需要设置三个参数。行 35指明了索引文件 url。行 36设置播放器 url，行 37则指定了 adaptive 插件的 url。如此以来，JW Player 就可以正常工作了。国际惯例，上王道。

图 5-1: JW Player 播放 HLS

5.3 JW Player adaptive 缺陷与改进

adaptive 插件可以很方便地播放 HLS 流，这自然是好事。但事情永远没有这么简单。我们在使用 adaptive 插件过程中遇到了一个很大的问题——视频片段在播放的时候会出现卡顿的现象，而且具有明显的周期性特征。到最后才发现，卡顿现象是由两个原因导致的：

1. Flash 调试日志
2. 单次处理的数据过多

下面详细分解。

adaptive 有一个简单的日志类，源代码见清单 5-2。大家注意行 12是调用了浏览器的 console.log 方法。实践表明，这个操作非常耗时。

清单 5-2: JW Player adaptive 日志类

```
1  package com.longtailvideo.adaptive.utils {
2      import flash.external.ExternalInterface;
3      /**
4       * Class that sends log messages
5       * to browser console.
6       */
7      public class Log {
8          public static var LOGGING:Boolean = true;
9          /** Log a message to the console. **/
10         public static function txt(message:*):void {
11             if(LOGGING && ExternalInterface.available)
                   {
12                 ExternalInterface.call('console.log',
                       String(message));
13             }
14         };
15     }
16 }
```

adaptive 插件在解析 TS 封包的时候遇到错误的 PES 头信息就会记录错误日志，代码片段见清单 5-3，其中行 7负责记录日志。

清单 5-3: JW Player adaptive PES 头信息错误

```
1 pes.writeBytes(_data,_data.position,todo);
2 if(stt) {
3     _audioPES.push(new PES(pes,true));
4 } else if (_audioPES.length) {
5     _audioPES[_audioPES.length-1].append(pes);
6 } else {
7     Log.txt("Discarding␣TS␣audio␣packet␣without␣PES␣
          header.␣");
8 }
```

很巧的是我们之前采用的开源切割工具 segmnter 有 bug，切割出来的视频片段或多或少会有数据损坏，直接表现为每个视频片段开始的时候会有大量(几十到上百不等)PES 头信息有问题的 TS 封包。adaptive 默认开启日志，导致日志类被不断调用，拖

累了整个系统的性能。

开始的时候没有将问题定位到日志系统，所能想到的方法就是改 segmenter 代码。这个难度太大，libav 库的文档实在是太匮乏，根本无从下手。最终的解决方法很简单，就是关闭系统日志……

关闭系统日志以后，卡顿现象得到显著改善，小小的欣慰了一把。可是～可是卡顿现象依然存才。怎么办呢？

根据之前的经验，我们有理由相信是 adaptive 某个部分的代码性能不给力导致卡顿现象的产生。所以我们就在几个可能出现问题的地方加了监控，终于发现清单 5-4里面的构造函数有问题，非常耗时。这段代码乍一看好像没什么问题，但请留意一下 行 5的*无限* 循环。

清单 5-4: JW Player adaptive TS 构造函数

```
1  /** Transmux the M2TS file into an FLV file. **/
2  public function TS(data:ByteArray) {
3      // Extract the elementary streams.
4      while(data.bytesAvailable) {
5          _readPacket(data);
6      }
7      if (_videoPES.length == 0 || _audioPES.length == 0
           ) {
8          throw new Error("No␣AAC␣audio␣or␣AVC␣video␣
               stream␣found.");
9      }
10     // Extract the ADTS or MPEG audio frames.
11     if(_aacId > 0) {
12         _readADTS();
13     } else {
14         _readMPEG();
15     }
16     // Extract the NALU video frames.
17     _readNALU();
18 }
```

_readPacket 方法比较复杂，使用纯 ActionScript 实现了 TS 封包的解析。具体代码不贴了，但里面几乎就是一个字节一个字节地来解析 TS 封包，耗时耗力。行 5居然来了一个无限循环，如果 TS 封包很多的话性能肯定有问题。我们所用的视频片段大约有几万个 TS 封包，一口气循环完成的话大约需要 300ms 左右的时间。这么长的时间直接导致播放卡住，但因为持续时间不长，宏观就表现为卡顿现象断断续续。

好了，找到问题就好办了。我们的解决思路很简单，就是把这个强大的”一口气“循环改成“小碎步”循环——即将大任务拆分成几个小任务[12]。我们可以通过 Flash 的事件机制和计时器机制来实现“伪多线程”，最终达到拆分任务的目标。但是，在拆分之前我们要做的第一件事情是梳理清楚 adaptive 相关方法的调用顺序。

清单 5-5: JW Player adaptive 视频片段处理流程

```
1  /** Fragment load completed. **/
2  private function _completeHandler(event:Event):void {
3      // Calculate bandwidth
4      var delay:Number = (new Date().valueOf() -
           _started) / 1000;
5      _bandwidth = Math.round(_loader.bytesLoaded * 8 /
           delay);
6      // Extract tags.
7      var tags:Vector.<Tag> = new Vector.<Tag>();
8      try {
9          tags = _parseTS(tags);
10         _switched = false;
11         _callback(tags);
12         _adaptive.dispatchEvent(new AdaptiveEvent(
               AdaptiveEvent.FRAGMENT, getMetrics()));
13     } catch (error:Error) {
14         _adaptive.dispatchEvent(new AdaptiveEvent(
               AdaptiveEvent.ERROR, error.toString()));
15     }
16 };
```

清单 5-5里的方法在视频片段下载完成后会被自动调用。里面代码显示了视频片段的处理流程：

1. 行 9真正解析视频片段。此方法需要异步返回，这是解决问题的关键。不然等话整个系统都会卡在这里。
2. 行 11是后期处理程序，需要在行 9异步调用成功后触发。
3. 行 12则是向整个系统报告视频片段解析完毕。也需要在 行 9异步调用成功后执行。

另外，行 9和 行 11的方法需要传入参数，这明显不行。我们将 tags 转换成成员变量，各个方法做相应的调整即可避免传参。

好了，接下来我们看看 _parseTS 的拆分如何实现。

[12]Flash 运行环境本身用多线程实现的，但不提供多线程相关的 API。

清单 5-6: JW Player adaptive 视频片段解析

```
1 private function _parseTS(tags:Vector.<Tag>):Vector.<
     Tag> {
2     var ts:TS = new TS(_loader.data);
3     ......
4
5     return tags;
6 };
```

清单 5-6里的方法比较长, 此处省略了大多数无关紧要的代码。核心代码就两句, 行 2实例化一个 TS 对象, 用来解析 TS 封包[13]; 行 5则是返回处理以后的结果。因为要实现异步调用, 我们首先将 行 2异步化, 调用成功后用事件对象返回 tags 数据。

离成功越来越近了。让我们再仔细看看清单 5-4 那个强大的“一口气”循环。我们可以将行 5放到一个独立的方法, 设置定时器, 定时执行。此方法每次只处理有限个(比如 2000)TS 封包。通过这样老土的方法就可以释放 CPU 供前端显示使用。而后面的相关操作则放到另外一个独立的方法内。等所有 TS 封包都解析完成后触发相应事件执行。

好了, 思路就梳理到这里。实现起来就比较简单了。直接看注释吧。清单 5-7是 TS 类改进的代码。

清单 5-7: JW Player adaptive TS 类改动

```
1  public function TS(data:ByteArray, loader:Loader) {
2      _data = data;
3      _loader = loader;
4      // 注册TS_PKG事件回调函数
5      this.addEventListener(AdaptiveEvent.TS_PKG,
           _readAllPackageCallback);
6      // Extract the elementary streams.
7      // 设置定时器, 每毫秒执行一次
8      _intv = setInterval(_readPacketByTime, 1);
9  };
10 /**
11  * 间歇解析TS封包, 500个每毫秒
12  * 防止卡住播放界面
13  */
14 private function _readPacketByTime():void {
15     var i:int = 500;
16     while (_data.bytesAvailable && --i) {// 每次处理
           500个封包
17         _readPacket();
```

[13]还记得前面说的“一口气”循环吗?

```
18      }
19      if (!_data.bytesAvailable) { //
            所有封包处理完成触发TS_PKG事件
20          this.dispatchEvent(new AdaptiveEvent(
                AdaptiveEvent.TS_PKG));
21          clearInterval(_intv);
22      }
23  }
24  /**
25   * 捕获TS_PKG事件, 执行后续处理
26   */
27  private function _readAllPackageCallback(event:
        AdaptiveEvent):void {
28      Log.txt('_readAllPackageCallback');
29      if (_videoPES.length == 0 || _audioPES.length == 0
            ) {
30          throw new Error("No␣AAC␣audio␣or␣AVC␣video␣
                stream␣found.");
31      }
32      // Extract the ADTS or MPEG audio frames.
33      if(_aacId > 0) {
34          _readADTS();
35      } else {
36          _readMPEG();
37      }
38      // Extract the NALU video frames.
39      _readNALU();
40      // 通知上层TS类实例化完成
41      _loader.dispatchEvent(new AdaptiveEvent(
            AdaptiveEvent.TS_TS, this));
42  }
```

清单 5-8是改进后的 Loader 类。

清单 5-8: JW Player adaptive Loader 类改动

```
1  /** Create the loader. **/
2  public function Loader(adaptive:Adaptive):void {
3      _adaptive = adaptive;
4      _adaptive.addEventListener(AdaptiveEvent.MANIFEST,
           _levelsHandler);
5      _loader = new URLLoader();
6      _loader.dataFormat = URLLoaderDataFormat.BINARY;
7      _loader.addEventListener(IOErrorEvent.IO_ERROR,
           _errorHandler);
8      // 下载完成后触发_completeHandler方法
9      _loader.addEventListener(Event.COMPLETE,
           _completeHandler);
10     // _parseTS完成后触发_parseTSCompleteHandler方法
```

```
11      this.addEventListener(AdaptiveEvent.LOADER_PARSETS
            , _parseTSCompleteHandler);
12      // TS类实例化完成后触发_parseTS方法
13      this.addEventListener(AdaptiveEvent.TS_TS,
            _parseTS);
14  };
15  /**
16   * 处理流程开始
17   */
18  private function _completeHandler(event:Event):void {
19      // Calculate bandwidth
20      var delay:Number = (new Date().valueOf() -
            _started) / 1000;
21      _bandwidth = Math.round(_loader.bytesLoaded * 8 /
            delay);
22      // Extract tags.
23      try {
24          // 异步实例化TS对象，立即返回，进入事件循环
25          var ts:TS = new TS(_loader.data, this);
26      } catch (error:Error) {
27          _adaptive.dispatchEvent(new AdaptiveEvent(
                AdaptiveEvent.ERROR, error.toString()));
28      }
29  };
30  /**
31   * 将TS封包转换成flv
32   */
33  private function _parseTSCompleteHandler(event:
        AdaptiveEvent):void {
34      Log.txt('_parseTSCompleteHandler');
35      var tags:Vector.<Tag> = event.tags;
36      _switched = false;
37      _callback(tags);
38      // 通知上层程序转换完毕
39      _adaptive.dispatchEvent(new AdaptiveEvent(
            AdaptiveEvent.FRAGMENT, getMetrics()));
40  }
41  /**
42   * 具体将TS转换成flv
43   */
44  private function _parseTS(event:AdaptiveEvent):void {
45      Log.txt('_parseTS');
46      var tags:Vector.<Tag> = new Vector.<Tag>();
47      // 注意，此处ts变量又事件传入
48      var ts:TS = event.ts;
49      // Save codecprivate when not available.
50      if(!_levels[_level].avcc && !_levels[_level].adif)
            {
51          _levels[_level].avcc = ts.getAVCC();
```

```
52          _levels[_level].adif = ts.getADIF();
53          // _adaptive.dispatchEvent(new AdaptiveEvent(
                AdaptiveEvent.MANIFEST,_levels));
54      }
55      // Push codecprivate tags only when switching.
56      if(_switched) {
57          var avccTag:Tag = new Tag(Tag.AVC_HEADER,ts.
                videoTags[0].stamp,true,0,_level,_fragment)
                ;
58          avccTag.push(_levels[_level].avcc,0,_levels[
                _level].avcc.length);
59          tags.push(avccTag);
60          if(ts.audioTags[0].type == Tag.AAC_RAW) {
61              var adifTag:Tag = new Tag(Tag.AAC_HEADER,
                    ts.audioTags[0].stamp,true,0,_level,
                    _fragment);
62              adifTag.push(_levels[_level].adif,0,2)
63              tags.push(adifTag);
64          }
65      }
66      // Push regular tags into buffer.
67      for(var i:Number=0; i < ts.videoTags.length; i++)
            {
68          ts.videoTags[i].level = _level;
69          ts.videoTags[i].fragment = _fragment;
70          tags.push(ts.videoTags[i]);
71      }
72      for(var j:Number=0; j < ts.audioTags.length; j++)
            {
73          ts.audioTags[j].level = _level;
74          ts.audioTags[j].fragment = _fragment;
75          tags.push(ts.audioTags[j]);
76      }
77      Log.txt('Dispatch␣Event␣AdaptiveEvent.
            LOADER_PARSETS');
78      this.dispatchEvent(new AdaptiveEvent(AdaptiveEvent
            .LOADER_PARSETS, tags));
79  };
```

通过清单 5-7和 清单 5-8所示的改进,视频播放卡顿的现象完全没有了,播放效果令人满意。

§.6 分布式网络存储

6.1 Ceph 介绍

Ceph 是加州大学 Santa Cruz 分校的 Sage Weil (DreamHost 的联合创始人) 专为博士论文设计的新一代自由软件分布式文件系统。自 2007 年毕业之后, Sage 开始全职投入到 Ceph 开发之中, 使其能适用于生产环境。Ceph 的主要目标是设计成基于 POSIX 的没有单点故障的分布式文件系统, 使数据能容错和无缝的复制。2010 年 3 月, Linus Torvalds 将 Ceph client 合并到内核 2.6.34 中。

Ceph 中有很多在分布式系统领域非常新颖的技术点, 对解决分布式文件系统中一些常见的问题的研究非常有指导意义。所以值得研究。

CEPH 中使用了一个比较先进的算法 crush 算法, 据翻译出来, 为分布式基于对象的存储系统设计了一个可升级的伪随机的数据分布函数, 它能够有效地管理数据对象和存储设备, 而不需要通过一个中心目录。由于大系统都是动态的, CRUSH 被设计成为一个当把不需要的数据迁移最小化时, 能方便的增加或移除存储设备。这个算法提供了一个大范围的不同种类的数据复制和可靠性机制, 以及根据用户自定义的策略来分配数据, 这种策略迫使数据复制从故障领域分离出来。

另外 CEPH 使用的文件系统为 btrfs, 这个文件系统具有很多先进的特性, 为下一代 Linux 使用的文件系统。BTRFS 最终可能会给 ZFS 等带来更多威胁, 它具有在线碎片整理功能 (只有固态盘有这项功能) 、Copy-On-Write 技术、数据压缩、镜像、数据条带和快照等等。另外, BTRFS 在数据存储方面比 ext 更完善。它包括一些逻辑卷管理和 RAID 硬件功能, 可以对内部元数据和用户数据进行检验和, 同时内嵌了快照功能。ext4 也可以实现以上一些功能, 但是需要与文件系统和逻辑卷管理器进行通信。

6.2 Ceph 系统配置

6.3 Ceph 系统灾备

§.7 文件上传组件

7.1 综述

周一到现在(周三上午),我着重对大文件上传技术作了调研。先说现状。作为国内在线视频行业的佼佼者,优酷视频支持通过网页上传最大 1G 的视频文件,而且优酷视频为用户上传大尺寸视频提供了有限的断点续传功能。之所以说有限,是因为断点续传功能只有在 FireFox、Chrome 等浏览器下有效,并不支持 IE 系列浏览器。

大文件传输无非要解决两个问题:

1. 上传信息显示,包括上传速度、上传进度
2. 断点续传

而实现大文件上传的技术也是五花八门,包括但不限于 HTML5、ActiveX、Flash、SilverLight、JavaFX 等。但无论哪种方案,都无法很好的兼容多种平台。像 ActiveX 控件的确很强大,但只能工作在 IE 浏览器下。HTML5 的 File API 貌似也行,但是直到 IE9 都没有得到良好的支持。Flash 相对来讲是兼容性最好的技术,但是提供的接口着实有限,无法很好的胜任断点续传任务。

我们要达成的目标:

1. 尽可能多的兼容各种平台
2. 尽可能多的实现断点续传

就目前情况而言,我们需要综合使用 HTML5 和 Flash 两种技术来实现这两个目标。

先说 Flash。ActionScript 有一个类叫 FileReference,提供了有限的文件操作功能。FileReference 可以通过 upload 方法将指定文件上传到服务器。但是,整个上传过程我们无法控制,断点续传也就无从谈起。FileReference 还提供了一个所谓的 load 方法,同样非常不给力。load 方法可以将文件内容以二进制的形式载入内存,然后我们就可以做进一步处理。所有基于 Flash 的断点上传方案都使用了这个 load 方法,然后将内容分段切割,分段上传。此方案貌似完美,但是有一点需要指名——load 方法一次性加载所有内容。你懂得,使用此方案无法上传太大的文件。小心你的内存。我自己做过测试,虽然 Adobe 官方推荐的最大文件尺寸为 100M,但是通过 load 方法加载 1G 大小的文件也不会有太大的问题。不过,实在是太占内存了!!

再说 HTML5。HTML5 提供了一组非标准化的文件 API,而其中的 slice 最惹人注目。slice 方法为浏览器带来了动态加载指定范围文件内容的能力。也就是说,通过

slice 方法，你可以读取文件内容的某个部分，然后进行处理，或者用 AJAX 技术将其发送到服务端。slice 应该不存在占用内存过多的问题[14]。

那我设想的方案是这样子的：

- 定制 SWFupload 项目（Flash 方案），提供最基础的大文件上传功能，兼容所有平台，没有断点续传功能。
- 使用 Flash 实现有限的断点续传功能（文件体积不能过大，几百兆还是可以的），提示用户使用。IE 用户如果想使用断点续传功能的话只能使用此方案。
- 使用 HTML5 文件 API 实现断点续传功能，FireFox、Chrome 等浏览器均可使用。暂时无视 IE10 以前的版本。

上传组件还是比较复杂的。断点续传功能还需要相应的后端脚本配合。

7.2 SWFUpload 组件

7.3 KUploader 组件

7.4 HTML5 断点续传组件

[14]这个没有测试，前端的脚本技术实在太弱。

§.8 爱房视频系统展望

之前几节介绍了视频系统的准备工作,接下来的任务则是尝试着对未来爱房视频系统的可能配置做出尝试性的规划。

视频核心采用 HTTP Live Streaming 技术架构。原始视频文件被转码成 MPEG TS 格式,并切割存储。播放器采用 JW Player 的 adaptive 插件,同时使用 HTML5 的 video 标签来兼容苹果的移动设备。在测试期间,存储可以采用简单的磁盘阵列,后期随着业务量的增加可以考虑使用 Ceph 分布式存储系统。文件上传采用基于 Flash 的 SWFUpload 组件。考虑到以后扩展性,我们可以采用 HTML5 接口实现大文件上传组件来支持现代浏览器。

除了上述部分,整个系统还有一个核心部分,那就是视频处理服务器。我们需要配置一台或者多台高性能的服务器来处理视频文件。视频文件的转码、切割和添加水印等操作均可以通过 FFmpeg 实现。目前的状况是手工切割,这自然无法满足生产环境的需求。我们要做的是实现一个视频处理服务,根据相关要求,自动对用户上传的视频文件进行处理。

所有的技术问题都解决后,我们还必须处理视频系统同爱房主站的衔接问题。视频系统将作为独立部分为爱房网提供服务(类似图片系统)。用户上传视频以后,系统将返回视频的唯一标识。爱房主站可以将这一标识同用户信息相互关联,并在相关页面展示。

图 8-1显示了整个系统结构。

我们接下来要做的具体工作是:

- 视频自动处理服务
- 大文件上传组件
- 视频跟踪组件(视频索引文件 url)
- 定制播放器,体现爱房特色

还有很重要的一部分则是版权保护。业界通常的做法是添加版权水印,但这并非最合适的解决方案。一个最主要的原因就是迁移成本——你无法随意地更换水印。如果能够充分利用 HLS 的版权保护技术,我们就可以避免添加水印。相反,我们可以在客户端动态生成水印,想换就换,岂不会哉。

图 8-1: 爱房视频系统架构