

---

# 目錄

關於《認識音檔》	1.1
PCM 格式	1.2
MP3 與 ID3 格式	1.3
AAC 與 MP4 格式	1.4
FLAC 格式	1.5
HLS 與 FairPlay DRM	1.6
MPEG-DASH 與 Widevine DRM	1.7
常用相關工具	1.8
附註	1.9
MP3 Parser 範例	1.9.1
AAC-ADTS Parser 範例	1.9.2
ID3 Parser 範例	1.9.3
FLAC Stream Info Encoder 範例	1.9.4
文件版本	1.9.5

# 關於《認識音檔》

2022 and onwards © Weizhong Yang a.k.a zonble

《認識音檔》是一本我在前一份工作離職前（2020年五月）寫的一本小冊子，就如同標題一般，內容就是講解我在工作上曾經處理過、我所了解的音檔格式。一開始就寫了兩個版本，一個是包含前公司業務內容的版本，另外就是您現在所看到，把公司相關業務內容拿掉，適合公開發佈的版本。原本寫這本手冊，也不是誰特地指派的工作，只是覺得總要有人整理一些 knowhow 而已。

這個版本當中的內容，其實只要花點時間，都可以透過搜尋引擎，在網路上找到相關的公開資料，只是您可能不確定應該要使用什麼關鍵字搜尋，而這本手冊中，算是我個人整理過的系統化知識，希望可以對需要了解音檔格式是什麼的朋友有些幫助。

音檔—或是說，各種電腦檔案格式，並不是各種資訊教育會特別講解的一塊，對絕大多數的電腦用戶來說，只要能夠把聲音播出來就夠了；你問任何一個電腦用戶「什麼是 MP3」？大概所有人都可以講出「MP3 是一種檔案」，但 MP3 是怎樣的檔案？MP3 與 AAC 有什麼不同，能夠回答的就很少，因為沒有特別了解的必要。

甚至在業界，其實也只有在少數公司任職的工程師需要了解，大概也只有遇到產製音檔，或是播放音檔這些需求，需要製作播放軟體時才有特地了解的必要。所以，即使什麼資料在網路上都很容易找到，但是想要了解這些網路上的文件到底在講什麼，其實也要花上一段時間。起碼我自認花了不少時間。

即使是工程師都要花上不少時間，在數位音樂、或是其他與電腦音檔相關的產業，還有其他不同功能的角色，其實也需要了解音檔，如果你是個想要進入數位音樂產業的 PM，你可能上班的第一天，就可能聽到「客戶反應播放音樂會爆音」、「我們計畫推出高音質方案」等，而這些都是需要對音檔有一定了解才能完成的任務。

我在前一份工作中，主要工作內容之一，就是製作 client 端的播放軟體。所以，這份手冊會更偏向 client 端的角度，怎樣解析（parse）檔案格式，而且也比較偏重 container 的部份，畢竟跟 codec 有關的部分，也就只能夠呼叫 library

—一種音檔格式分成 container 與 codec，我們晚點會說明。至於轉檔工具的使用，怎樣在 server 上、CDN 上佈署檔案，就不是我所熟悉的範圍，所以不會在這份手冊中。

預期在讀完這份文件之後，你可以：

- 以後看到有人提到 48000Hz 24bit 的高音質檔案，可以馬上叫出「這是 DVD 音質」
- 知道音檔分成 codec 以及 container，而且知道這些名詞的意義
- 知道 FairPlay 與 Widevine 這些商用 DRM 所保護的是什麼格式

當中有什麼錯漏之處，也祈請大家指正。大家可以直 fork 這本手冊然後發 pull request。然後，根據本人過去經驗，應該沒有什麼時間經營討論區或群組，所以 GitHub 上就直接關閉討論版了，相信會用到這本手冊的應該都是同行，應該可以見諒。

## 授權

這本手冊使用 MIT License 釋出。

## 連結

- 這本手冊位在 [https://zonble.github.io/understanding\\_audio\\_files/](https://zonble.github.io/understanding_audio_files/)
- GitHub 專案位置在 [https://github.com/zonble/understanding\\_audio\\_files](https://github.com/zonble/understanding_audio_files)
- PDF 版本
- EPUB 版本

## 聯絡方式

- [g.dev/zonble](#)
- Twitter [@zonble](#)
- GitHub [@zonble](#)

## 感謝

在發布之後，感謝以下朋友的指正

- [iXerol](#)
- [kojirou1994](#)

## 我工作上曾經遇過的音檔

我在工作上曾經遇過

- PCM
- MP3
- AAC (AAC-MP4 與 AAC-ADTS)
- FLAC
- HLS (HTTP Live-Streaming)
- MPEG-DASH

在接下來的章節中，會逐一說明。

文件更新時間：2022/02/06 00:44 CST

# PCM 格式

## 數位與類比

聲音是人耳可以感受到的、空氣中與水中的震動，是一種類比（Analog）訊號，音檔則是用數位（Digital）的方式將這些震動記錄下來。

人體可以從大自然中感受到的訊號，都是類比訊號—所謂類比訊號，是在訊號中存在著相對的強弱的位比，而數位訊號則是由位元組成，是絕對的 0 與 1，用 0 與 1 所組成的數字，盡可能地描述與還原曾經發生在大自然中的類比訊號。

以顏色來說，大自然中並不會有絕對的白、也不會有絕對的黑，不會有完全無光或是光線數量的極限狀態，以聲音來說，也不會有完全的無聲，或是聲音的極限狀態。

但因為電腦的容量與算力的關係，在電腦上，我們必須訂出一個數值的上限與下限。在大自然當中有無數種色彩，但是我們在電腦上，以 RGB 來說，就是  $256 * 256 * 256$  所組成的、將近一六七七萬種顏色。

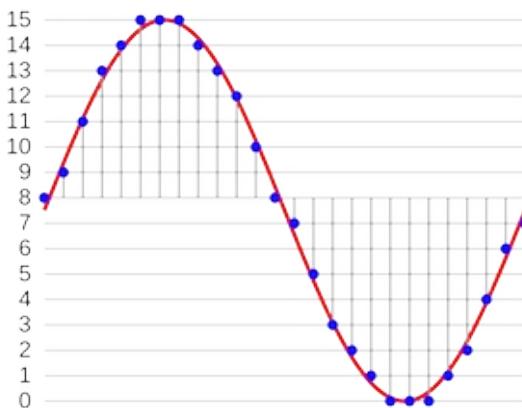
當我們想要把大自然當中的類比訊號轉換成數位訊號的時候，我們就只能夠記錄實際發生在大自然當中的部份訊號，而我們記錄的是人體的感官所能夠感受到的範圍。

類比信號才是人類最後可以感受到的信號，即使我們轉換了數位音檔，還是必須透過耳機或是喇叭等硬體播放出來。我們需要數位音檔的原因是一類比訊號往往無法傳遞到遠方，人聲往往傳遞不過幾十到幾百公尺，但我們希望可以將聲音傳送到更遠的地方；聲音會經過一段時間之後消失，但我們希望可以在事件發生之後，仍然可以重現當時的訊號。但，記錄的過程，也同時是失真的過程——一定會有些片段或事物，在轉換、傳遞的過程當中遺失。

## 採樣

用來偵測空氣中的震動的裝置，我們都知道，那就是麥克風。那麼，在電腦上，會被以什麼方式記錄？

聲音是波，在數學上，如果我們想要描述一個波，會使用二元方程式表達，以 X 軸代表時間，以 Y 軸代表聲波的震動，理論上我們可以將時間分割成無限小，而現實中，物理上時間還是會有一個最小的單位，而我們在電腦上，也沒辦法將時間分割到非常小的單位，所以數位音檔勢必要將這個連續的波型，根據一個合理的時間刻度，在這個時間刻度上，標上一連串、一個又一個的數字。



將連續的波型變成一連串的數字，這個過程叫做**採樣**（Sampling，也稱作取樣，可以參見 [Wikipedia 上的說明](#)），每一個被採集出來的數字，在中文也叫做採樣（Sample）—英文中很多單字的動詞與名詞是兩個字，但是在中文往往會變成同一個詞，在台灣，也比較常直接講英文。

一個波型，有振幅與頻率這兩種性質，在採樣的過程中，我們只擷取一定範圍的振幅與頻率之間的聲音。採樣的過程，其實也就是一種捨棄的過程：

- 一個聲波的上下起伏，代表的是聲音的音量大小，波型的密集程度，則代表聲音的音高，如果波型越密集就代表這個聲音愈高，反之則愈低。音量的單位是**分貝（dB）**，人耳可以聽到的音量在於 0 到 100 分貝之間，超過 100 分貝，就會超過人耳可以忍受的程度，超過 100 分貝的聲音，在採樣過程中，就會被當成 100 分貝處理。
- 人耳大概可以聽到頻率為 22050 Hz（意思是在 1/22050 秒中產生的波型），而如果將頻率乘上兩倍，變成 44100，根據**採樣定理**，人耳就聽不

出差異。不過，近幾年的高音質音檔、以及所謂的 DVD 音質，也用到了像是 48000、96000 這樣的頻率。

我們所謂的**採樣率**（Sample Rate，也稱採樣比、取樣率...等）：就是在一秒鐘當中，我們採集了多少個數字，所謂 44100 的採樣率，就是一秒鐘採集 44100 個數字。

我們往往會採集左右兩個**聲道**（Channel），以創造立體聲的效果。所以在一組立體聲的採樣中，會有兩個數字，兩個聲道的數字會用 LRLR...這樣的左右並排方式排列，這種一左一右排列採樣的方式，我們叫做 Interleave。當我們在計算檔案大小的時候，還需要從 Sample Rate 再乘上 2，把左右聲道兩個 Sample 合併在一起，就是一個 frame。

在某些格式中，可以允許更多的聲道，像 FLAC 就支援到八聲道，而像所謂 5.1 聲道、7.1 聲道...都不只 2 聲道。

電腦有各種表達數字的方式，而由於聲波會上下震盪，所以我們需要用帶正負號的數字來表達往上與往下的起伏，常用到的數字格式包括 16 位元整數（signed int 16）、32 位元整數（signed int 32）、32 位元浮點數（float 32）...等。對許多自己透過數學產生電腦波型的工程師來說，往往會使用 32 位元浮點數（採樣數字介於 1.0 到 -1.0），撰寫各種聲音處理的演算法，在 CD 上是使用 16 位元整數（採樣數字介於正負 2 的 15 次方），用兩個 byte 儲存一個採樣。

至於目前許多高音質格式用的是 24 bit，代表用 3 個 bytes 儲存一個數字，不過，一般寫程式通常不會用到這種數字，所以實際使用上，會把 24 bit 整數再轉換成 32 整數或是浮點數。

## PCM 格式—所謂的原始音檔

這種以一連串數字描述聲音的格式，我們往往叫做原始音檔，而術語叫做**PCM 格式**（Pulse-code modulation，中文叫做「脈衝編碼調變」，但是很少人可以記住這個中文名稱，通常就慣稱 PCM）。像微軟 Windows 上的**WAV 檔案**（用「錄音機」軟體錄製出來的聲音檔案）、蘋果平台上的**AIFF 檔案**，都屬於這種格式，只是通常都還會在最前方加個檔頭。而從上面的描

述，我們也可以知道，雖然同樣叫做 PCM 格式，但是可能會有不同的 Sample Rate、或是用不同的數字格式描述波型，而 WAV、AIFF 檔案的檔頭，用是用來告訴播放軟體應該怎麼解析檔頭之後的資料。

現在用戶通常很少會直接播放 PCM 檔案，在網路上流通的音檔，也很少會是 PCM 格式，但無論是哪種格式，在 client 端最後都得要轉換成 PCM 格式，才能夠再交給 client 端的硬體播放：有可能是透過平台本身已經包好的播放器元件，像 iOS/macOS 平台上還有高階的播放器 AVPlayer，也可能是要使用更低階的 Audio API，自己想辦法呼叫 codec，將各種格式轉換成 PCM 格式。

而如果想要用一些聲音剪輯或編輯軟體，像 Audacity，處理一段錄好的聲音，也需要轉換成 PCM。比方說，我們想要編輯一個 MP3 檔案，Audacity 會先轉成 PCM 格式讓我們編輯，在存檔的時候，再把 PCM 轉回 MP3。另外，在使用 Audacity 的時候，可以注意到有一項 "Import->Raw Data" 的功能，就是用來匯入沒有檔頭的 PCM 資料，而由於沒有檔頭，就必須手動填入聲道數量、sample rate 等...。

## 音訊處理

一般在講數位音訊處理的書籍或文件，接下來往往會講怎樣製作各種聲音效果，像是怎樣對聲音訊號做傅立葉轉換（Fourier Transform），以及怎樣用程式產生波型，像是 sine wave、sawtooth wave...進而打造屬於自己的數位樂器，等等。

在這邊，我們主要講解各種格式的檔案，所以並不討論這部份（其實我自己對這方面也不怎麼了解），不過，我們需要知道：如果要產生改變聲音本身的各種效果，像是 EQ 等化器、迴音、殘響或立體聲效果等，都是透過改變波型達成的，而如果我們想要在 UI 上顯示頻譜圖，也是先讀取 PCM 訊號，然後繪製出轉換後的結果。

也就是說，如果我們選擇了某種音檔格式，這種格式只能夠用特定的播放器播放，讓我們無法直接碰觸到 PCM 格式的資料，我們就無法達到上述的這些效果。比方說，我們打算使用一些經過商用 DRM 保護的格式，像是經過蘋果 FairPlay 保護的 HLS、或是 Google 的 Widevine 保護的 Dash 串流，這些串流

格式就只能夠用特定播放器播放，如果還想要加上 DTS 立體聲效果，就是完全衝突的。或許某些播放元件提供給我們一些選項，但只要不能夠碰到 PCM 格式的資料，我們能夠對聲音效果的客製都是有限的。

## CD 音質、Hi-Res Audio

我們所謂的 **CD 音質**，就是用 **44100 的採樣率、有左右聲道的 16 位元整數 (Sint16)**，所以我們可以預估，一秒鐘的資料量就是  $44100 * 2 * 2 = 176400$ ，一分鐘就需要 10mb 左右，一張 CD 有 640mb 的容量，所以就可以算出一張 CD 可以容納大約  $640 * 1024 * 1024 / (44100 * 2 * 2 * 60)$ ，大約 63.4)。這也是長久以來數位音樂的標準。

我們常常說晚近一些訂閱制音樂服務是「數位音樂」，不過，CD 就已經是數位音樂了，CD 也創造了一個輝煌的唱片銷售時代。

此外還有以下常見的音質標準

- 電話音質：11,025 Hz, 8bits, 單聲道
- 收音機音質：22,050 Hz, 8bits, 單聲道
- DVD 音質：96,000 Hz, 24bits 雙聲道

至於所謂的 **Hi Res Audio**，其實定義很混亂，基本上意義就是 CD 音質以上的檔案，通常是指 96k 或 192k Hz 採樣比的音檔。

更高的採樣比意味著音質的提升，不過，但對於很多用戶來說，如果只是使用一般的耳機設備，其實聽不出更高採樣比有什麼差別，而且很多現在流行的藍芽無線耳機（如蘋果的 Air Pod），在透過藍芽傳遞音訊時，為了讓藍芽傳輸順暢，還會做過轉檔、重新轉成特定採樣比的資料，把高音質的檔案轉成普通的音質播放。

文件更新時間：2022/02/06 00:44 CST

## MP3 格式

Internet 在上個世紀 90 年代開始普及。一首錄製在 CD 上的三分鐘長度的歌曲，大概就要 30mb，以那個時代的頻寬來說，傳遞這樣的檔案是非常吃力的事情。

但是，在 MP3 格式出現之後，一首 30mb 大小的歌曲變成了 3mb 大小，變成可以方便地在網路上傳佈，於是，無論是用戶在非法 P2P 軟體上大量散佈、傳遞 MP3，或是後來出現透過網路購買音檔的音樂商店，甚至更後來出現了月費制音樂服務、甚至是 Freemium 商業模式，MP3 都整個改變了音樂產業以及整個世界。

我們在這邊略過 MP3 的歷史，MP3 如何成為規格，以及音樂產業發生的變化，只討論 MP3 是怎樣的格式，以及工程師可以如何處理 MP3 檔案。

MP3 是一種壓縮格式，於是讓檔案變小，更精確來說，是一種破壞式壓縮檔案，在壓縮過程中，會捨棄掉一些發明這個格式時、覺得可以捨棄掉的部份，如果我們把 CD 上的歌曲轉成 MP3，雖然還是有辦法可以再轉換回 CD 格式的 PCM 檔案，但這樣的檔案已經跟原本的檔案不同了。

## Bit Rate

**Bit rate** 在中文很多時候叫做比特率，描述的是壓縮音檔被壓縮到什麼程度，單位是 bps (bits per seconds，一秒的音訊需要多少 bit)。

我們常見所謂 128k、192k、320k 這些單位，意思是，一秒鐘的壓縮音檔，會需要用到多少 bit 的資料，以 128k 來說，就是一秒鐘有 128 k 的 bit，如果我們想要換算成 byte，就是  $128 * 1024 / 8$  (一個 byte 有 8 個 bit)，我們可以得到 16,384 bytes，因此我們可以算出，如果是一首三分鐘的歌曲，在固定碼率的狀況，檔案大小就在 3mb 左右 ( $128 * 1024 / 8 * 3 * 60 = 2949120$ )。

在壓縮過程中，還有所謂的**變動碼率** ( VBR，Variable Bit Rate) 與**固定碼率** ( CBR，Constant Bit Rate)。如果某個 packet 當中出現的數字都非常近似，像是這個 packet 中的聲音都是靜音的，那麼，我們就有機會可以把這個

packet 壓縮得更小，於是，我們就可以把整個 MP3 檔案壓縮得更小，而每個 packet 的大小則不一，這就是變動碼率。如果我們關閉這種行為，讓每個 packet 的大小固定，就是固定碼率。

## Packet、以及 MP3 檔案的長度

在將 PCM 編碼成 MP3 格式的時候，我們並不是直接壓縮整個檔案，而是先將連續的 Binary Data 切成許多小塊的資料，把一小塊、一小塊的資料壓縮起來之後，再把壓縮後的資料連接起來。用來對這樣小塊資料做壓縮/解壓縮的程式，叫做 **codec**，每一個小塊的單位，叫做 packet（通常在台灣稱呼 packet 為封包）。或這麼說：codec 就是負責壓縮/解壓縮 packet 的程式。

在 MP3 格式中，每個 packet 當中，通常會有 1152 個採樣（其實也可能會有 384 或 576 這樣的採樣數字，但實在很少用到），如此一來，就往往會遇到必須要在最後一個 packet 中，透過填入 0 補足 packet 長度的情形，另外，許多的 encoder 也會在檔案的前後補上一些靜音的採樣（像 LAME 這個 encoder 就是如此，參見 LAME 的[說明](#)）。因此，我們要有一個重要的觀念：**當我們在轉換音檔格式的時候，音檔的長度是會改變的**。

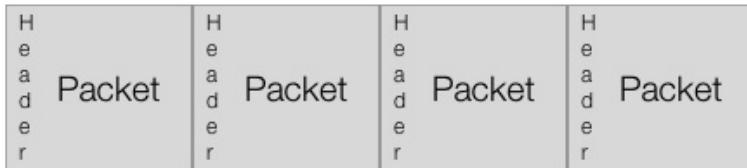
以 CD 音質來說，由於一秒鐘有 44100 個採樣，也就代表，這個檔案中的最短時間單位是  $1/44100$  秒，而 MP3 格式的最短時間單位則是  $1152/44100$  秒；假使我們現在有一個剛好一秒鐘的 PCM 檔案，想要轉換成 MP3 格式， $44100/1152$  是 38.28125，如果我們想要完整包含這一秒的資料，MP3 格式就得用到 39 個 packet，但是第 39 個 packet 得在後面補上零，而轉出來的 MP3 也不會是一秒鐘，而是 1.0187755102 秒。

Client 端的 player 在播放 MP3 時，不用把這個檔案全部抓下來，只要累積一定長度的資料，認為這個長度足夠順暢播放，就會開始播放。計算的方式就是看累積了多少 packet：從上面那個例子，就可以知道，如果 Player 找到了 39 個 packet，就可以算出，這個檔案有大概一秒左右的資料；如果 client 端覺得，有了五秒鐘左右的資料，就可以播放，那就可以開始將 MP3 轉換成 PCM 播放。如果網路有點卡頓，造成某段資料播放完畢，但是等不到接下來的資料，這時候 player 就會進入所謂 **stall** 的狀態：player 會先暫停，等到繼續累積到夠多的 packet，才接續播放。

而想要知道一首 MP3 歌曲有多長，就是把這個檔案讀過一次，找出這個檔案有多少 packet，然後用 packet 的數量算出時間。

所以，原本是同一首歌曲，經過轉檔之後，在不同格式下的長度是不同的，所以，如果你在經營音樂服務，想要統計用戶對一首歌曲到底播放了多少時間，也必須把這些差異考慮進去，即使用戶播放的是同一首歌曲，但因為選擇了不同的格式，最後回報的播放長度，也還是會有毫秒等級的差別。比方說，如果你同時提供一首歌曲的 MP3 與 AAC 版本，在 AAC 格式中，一個 packet 的大小是 1024，所以，MP3 與 AAC 格式的長度就是不同的。

以圖形表示 MP3 格式，大概會像這樣：



## MP3 Frame Header

一個 MP3 檔案中，所包含的是連續的 Binary Data，不過，在我們編製 MP3 檔案的時候，會在每個 packet 前方加上一小段 header，由於這種 header 有明顯的特徵，Player 只要找到 header，就可以知道哪裡是 packet 的開頭與結尾，把 packet 資料抽出來，交給 codec 解壓縮，轉成 PCM 格式，之後就可以交給作業系統的音訊 API 播放。

每個 header 的長度為 4 個 bytes，最大的特色是前 11 個 bit 都是 1，這一段叫做 **Syncword**。找到 syncword 之後，就可以繼續把後面的 bit 都讀完，就可以得到 Mpeg Audio 版本（在 MP3 之前還有 MP1、MP2 兩種格式，但是並不流行，但是 MP3 Header 還是把這個版本列入規格中）、壓縮後的 bitrate... 等等資訊。

Within an MPEG audio file, there is no main header, as an MPEG audio file is just built up from a succession of smaller parts called frames. Each frame is a datablock with its own header and audio information.

In the case of Layer I or Layer II, frames are totally independent from each other, so you can cut any part of an MPEG audio file and play it correctly. The player will then play the music starting from the first full valid frame it will find. However, in the case of Layer III, frames are not always independent. Due to the possible use of the "byte reservoir", which is a kind of internal buffer, frames are often dependent of each other. In the worst case, 9 input frames may be needed before being able to decode one frame.

If you need to retrieve information about an MPEG audio file, you might simply locate the first frame, and retrieve information from its header. Information within other frames should be consistent with the first one, except for the bitrate, as you might be retrieving information from a variable bitrate (VBR) file. In a VBR file, the bitrate can be changed in each frame. It can be used, as an example, to keep a constant sound quality during the whole file, by using more bits when the music is more complex and thus requires more bits to be encoded with a similar quality.

The frame header itself is 32 bits (4 bytes) length. The first twelve bits (or first eleven bits in the case of the MPEG 2.5 extension) of a frame header are always set to 1 and are called "frame sync". Frames may also feature an optional CRC checksum. It is 16 bits long and, if it exists, immediately follows the frame header. After the CRC comes the audio data. By re-calculating the CRC and comparing its value to the sorted one, you can check if the frame has been altered during transmission of the bitstream.

Here are the details of what is within a frame header

AAAAAAAAA AAABBCDD EEEEFFFGH IIJJJKLMM

Sign	Length (bits)	Position (bits)	Description
A	11	(31-21)	Frame sync (all bits must be set)
B	2	(20,19)	MPEG Audio version ID 00 - MPEG Version 2.5 (later extension of MPEG 2.0) 01 - reserved 10 - MPEG Version 2 (ISO/IEC 13818-3) 11 - MPEG Version 1 (ISO/IEC 11172-3)
C	2	(18,17)	Note: MPEG Version 2.5 was added lately to the header of lower sampling frequencies, if your decoder does synchronization instead of 11 bits.
D	1	(16)	Layer description 00 - reserved 01 - Layer III 10 - Layer II 11 - Layer I
E	4	(15,12)	Bitrate index Bit   V1 V1.1 V1.2 V1.3 V2 L1 V2 L2 & L3 ----- 0000 free free free free free 0001 32 32 32 32 6 0010 64 48 40 48 16 0011 96 56 48 56 24 0100 128 64 56 64 32 0101 160 80 64 80 40

```

void LayerIII_Decoder::initTables(int32 ch, int32
    gr_info_s gr_info = k1s1s0ch(ch).gr(p1));
}

extension used for very low bitrate files, allowing the use
of, it is recommended for you to use 12 bits for

    for (int i = 0; i < 12; i++) {
        if (gr_info->block_type == 21) {
            gr_info->bitrate[i] = 16;
        } else {
            gr_info->bitrate[i] = 64;
        }
    }

    for (int i = 0; i < 12; i++) {
        if (gr_info->block_type == 21) {
            gr_info->bitrate[i] = 17 * 64;
        } else {
            gr_info->bitrate[i] = 16 * 64;
        }
    }
}

```

如果我們打開 MP3 header 的規格書，會看到一串「AAAAAAA  
AAABBCCD EEEEFFGH IIJKLMM」像是天書的文字，我們不要被嚇到，這串文字代表的是這 4 個 bytes、32 個 bit 中、每個 bit 的用途。像是一開頭的 11 的 A，就是我們上面講到的 syncword。

用來將 header、packet 從連續的 Binary Data 找出來的程式，叫做 **Parser**。對於有處理過 Binary Data 的工程師來說，都有辦法可以自己撰寫出 MP3 Parser，至於 MP3 codec 則很長時間受到權利保護，一般工程師也沒什麼機會直接接觸 codec。而在像是 macOS 或 iOS 等蘋果的平台上，蘋果只提供我們 decode MP3 的 codec，而沒有 encode MP3 的 codec。

ID3

MP3 格式本身只有定義聲音資料的部份，但很多時候，我們需要一些關於音檔的額外描述資料，像是這種歌的歌名，歌手是誰、出自哪張專輯、屬於哪種樂風...也就是英文所稱的 metadata，並沒有包含在 MP3 格式的規範裡。我們所使用的音檔播放軟體，像是 iTunes（在 macOS 10.15 之後改名 Music）、VLC 等等，之所以可以從檔案中知道這些資訊，則是透過 ID3 格式。



ID3 metadata 也是一連串的 binary data，目前比較通行的是第二版的 ID3v2，會出現在整個 MP3 檔案最前面的位置。ID3 資料的，前三個 bytes 就是 "ID3" 這三個字元的 ASCII code。在讀取 MP3 檔案的時候，只要一開始就讀到這三個字，就可以判斷這個檔案有 ID3 資訊；接著，我們就來處理正段資料的前 10 個 bytes，在這 10 個 bytes 中，除了 "ID3" 這三個字之外，還包含了這是那個版本的 ID3，以及整段 ID3 資料區段的長度有多少等資訊...我們就一直讀到這個長度範圍。

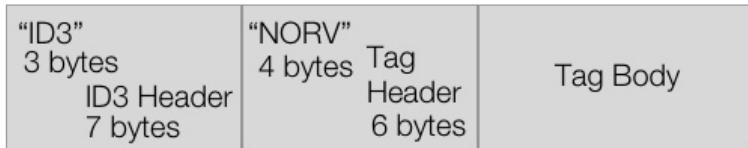
有一點需要注意，ID3 格式中，是用 28 位元不帶正負號整數 (UInt28)，描述 ID3 資料的長度。28 位元整數這個規格有點奇妙，總長度其實也是 4 個 bytes，但是每個 byte 的第一個 bit 要捨棄掉，然後，用這四個 bytes 的剩下 7 個 bit，組合出一個整數。

從第 11 個 byte 開始，我們就可以讀出一個個的 ID3 frame，每個 ID3 frame 也是由 header 與 body 組成，在 header 中可以讀出這個 frame 代表什麼、以及 body 的長度，body 則是實際的內容，像是歌名、歌手...等：

- 每次先讀 10 個 bytes
  - 前 4 個 bytes 是 Tag ID，像 TALB 就是專輯名稱、TCOM 是作曲者...
  - 5-8 這 4 個 bytes 是 frame body 的長度，也是 28 位元無正負號整數
  - 2 個 bytes 的 padding
- 然後根據 header 中的 frame body 長度，往下讀取
  - frame body 的第一個 byte，代表的是這個資訊使用的字元編碼方式
    - 0 : ASCII
    - 1 : UTF-16-LE
    - 2 : UTF-18-BE

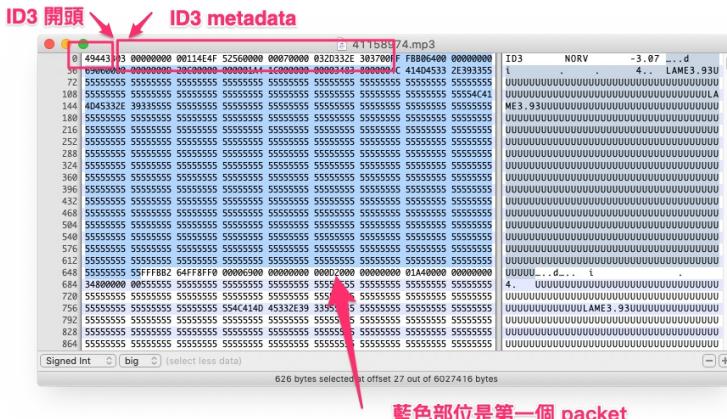
## ■ 3 : UTF-8

每段資料的長度如下圖：



我們也可以參考 ID3 網站上的規格。

我們可以用 Hex Editor 打開一個 MP3 檔案，就可以發現我們所講到的重點：



- 我們可以看到，這個 MP3 檔案的開頭是 "ID3"，代表包含了 ID3 資訊
- 由於 syncword 是連續 11 個 1，所以我們去找 "FFFF"，就可以找到 packet 的開頭
- 這裡的第一個 packet，其實是 LAME 加上去的靜音 packet

## MP3 格式的優缺點

每個 packet 前面都有一個檔頭，同時是 MP3 格式的優點與缺點。好處是，如果播放軟體透過網路，接收到了一段連續的 MP3 資料，只要播放器從資料中發現一個 syncword，就可以找到 packet 並且開始播放，而如果因為網路傳輸

問題，造成某一資料遺失，那麼，一樣只要繼續往下讀取，找到 syncword，就可以恢復播放。

MP3 格式的這種特性，在 90 年代造就了像是 shoutcast 這樣的網路廣播電台。Shoutcast 技術讓 client 與 server 之間建立 socket 連線，server 不斷向 client 端傳送連續的 MP3 資料，client 端一開始連上時，並不認得最前面的一些 bytes，但只要讀到第一個 syncword，就可以讓用戶開始享受網路廣播。

但，如果我們想要知道一個 MP3 音檔的實際長度，以及像是播放位置一分零三秒會對應到檔案的哪個位置，MP3 就不是很有效率的格式。我們想要知道總長度，就得把整個檔案讀取一次，才有辦法知道總共有多少 packet，我們也得讀取了一分零三秒的資料，才有辦法要求播放軟體 seek 到一分零三秒播放。

比方說，我們有一個放在 web server 上的 MP3 音檔，我們想要在播放這個檔案的時候，直接從一分零三秒開始播放，更有效率的方式是，如果我們可以一開始就知道一分零三秒是整個檔案的第幾個 byte，我們就直接帶了 Range HTTP Header 的 HTTP 連線，直接從那個位置開始抓檔，Server 就會回應 HTTP status code 為 206 的回應，只給我們那一段資料，省去載入前一分多鐘的時間。

那麼，我們有什麼辦法可以知道一分零三秒到底在檔案的哪裡？如果是固定碼率的檔案，我們大概還有辦法推算，但如果是變動碼率呢？

於是，我們發現，在 MP3 之後的許多檔案格式，是讓 codec 與 container（中文通常叫「容器」，但習慣上還是直接講英文）分家的。codec 負責的是 packet 的壓縮/解壓縮，而 container 則負責安排 packet 與 packet 的 header 的擺放方式，像 AAC 音檔就有多種不同的 container 格式。在蘋果的平台上，通常把 codec 格式叫做 audio format、把 container 叫做 file format。

## 改變 Bit Rate 會改變 Sample Rate 嗎？

之前有個同事有點把 Bit Rate 與 Sample Rate 搞混了，所以問我，MP3 明明就會把檔案壓小，為什麼壓縮之後，Sample Rate 並沒有改變？我就舉了一個圖檔的例子。

如果你有一張 1920 x 1080 解析度的圖檔，在儲存成 JPEG 圖形的時候，各種修圖軟體或是轉檔軟體，會詢問你要怎樣的比例，如果你想要把圖檔壓縮得越小，那麼壓縮之後的圖片就會越失真，但不管你怎麼壓縮，你重新打開這一張圖檔，解析度仍然是 1920 x 1080。音檔也是同樣的道理，對於 MP3 這些格式來說，不管你把音檔壓縮得多小，最後 codec 把壓縮音檔還原成原始 PCM 格式的時候，還是一樣的 Sample Rate。

而改變 Sample Rate 這件事情，也比較像是你把一張 1920 x 1080 的圖片轉成 1280 x 720 的圖檔。

文件更新時間：2022/02/06 00:44 CST

# AAC 與 MP4 格式

AAC 全名 Advanced Audio Encoding，是一種在上個世紀九零年代底訂出的規格。AAC 是一種 codec，但是可以被包在不同種類的 container 中，所以，我們通常不會單獨稱呼有某種檔案是 AAC 檔案，而是將 AAC 與 container 格式合稱，像是 AAC-MP4、AAC-ADTS...等。

## AAC-ADTS

AAC-ADTS 格式與 MP3 格式接近，一樣是一段 header 之後接著一個 packet，所以，我們也可以看到使用 AAC 格式的網路廣播電台。每個 packet 使用 AAC codec 壓縮，每個 packet 中包含 1024 個 frame，因此我們可以知道，使用 AAC-ADTS 產生出來的音檔的長度，與 MP3 是不同的。

AAC-ADTS 也有自己的 header 格式，長度在 7 到 9 個 bytes 之間。一般來說，附檔名是 .aac 的檔案，便是 AAC-ADTS 檔案。

The screenshot shows the Wikipedia page for "ADTS". The page title is "ADTS". Below the title, there is a brief description: "Audio Data Transport Stream (ADTS) is a format, used by MPEG TS or Shoutcast to stream audio, usually AAC." There is also a note about the header consisting of 7 or 9 bytes (without or with CRC).

**Structure**

```
AAAAAAA AAAABCCD EEEEEFGH HHIIJKLMM MMMMMMMMM MMMOOOOOO OOOOOOPP (QQQQQQQQ QQQQQQQQ)
```

Header consists of 7 or 9 bytes (without or with CRC).

Letter	Length (bits)	Description
A	12	syncword 0FFF, all bits must be 1
B	1	MPEG Version: 0 for MPEG-4, 1 for MPEG-2
C	2	Layer: always 0
D	1	protection absent, Warning: set to 1 if there is no CRC and 0 if there is CRC
E	2	profile, the MPEG+ Audio Object Type minus 1
F	4	MPEG-4 Sampling Frequency Index (15 is forbidden)
G	1	private bit, guaranteed never to be used by MPEG, set to 0 when encoding, ignore when decoding
H	3	MPEG-4 Channel Configuration (in the case of 0, the channel configuration is sent via an inband PCE)
I	1	originality, set to 0 when encoding, ignore when decoding
J	1	home, set to 0 when encoding, ignore when decoding
K	1	copyrighted id bit, the next bit of a centrally registered copyright identifier, set to 0 when encoding, ignore when decoding
L	1	copyright id start, signals that this frame's copyright id bit is the first bit of the copyright id, set to 0 when encoding, ignore when decoding
M	13	frame length, this value must include 7 or 9 bytes of header length: FrameLength = (ProtectionAbsent == 1 ? 7 : 9) + size(AACFrame)
O	11	Buffer fullness
P	2	Number of AAC frames (RDBs) in ADTS frame minus 1, for maximum compatibility always use 1 AAC frame per ADTS frame
Q	16	CRC if protection absent is 0

打開 ADTS 的規格，我們又看到「AAAAAAA AAAABCCD EEEEEFGH HHIIJKLMM MMMMMMMMM MMMOOOOOO OOOOOOPP (QQQQQQQQ QQQQQQQQ)」這種文字，我們既然處理過 MP3，我們也知道怎麼處理這種

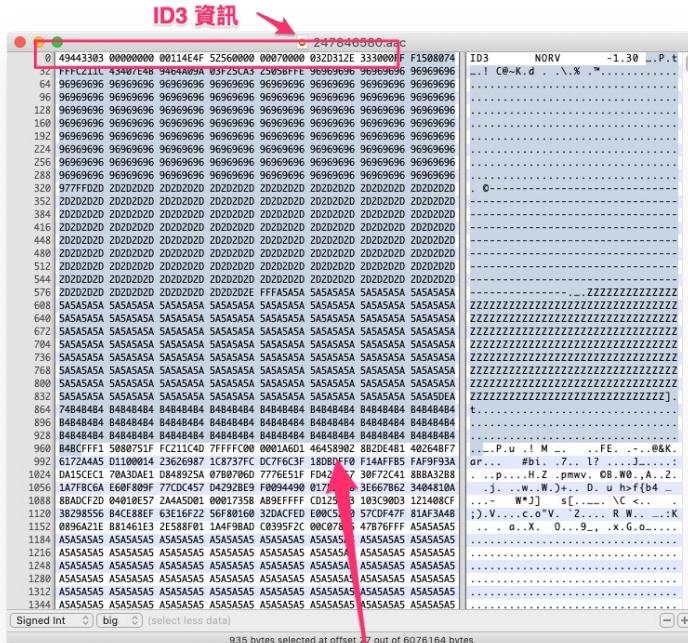
header °

前 12 個 bit 是 syncword，所以，只要連續讀到 12 個 1，就可以判斷是 header 的開始。從第 30 個 bit 開始，也就是 M 這段，裡頭是整個 packet 的長度—請注意，這個長度也包含 header 的部份—只要繼續往下讀，就可以讀出 packet。

AAC-ADTS 格式也可能包含 ID3 檔頭。在 iOS/macOS 平台上，需要注意：

Core Audio API 所提供的 parser 無法正確解析檔案前方有麟掉的資料的 AAC-ADTS 資料，我們需要自己寫一個 parser，手動把 Core Audio 無法解析的部份濾掉—也就是說，我們要自己想辦法找到第一個 syncword。

我們也可以打開 Hex Editor 看看。一個包含 ID3 資訊的 AAC-ADTS 檔案，可能會是像這樣：



AAC-MP4

AAC-MP4 格式的檔案，通常副檔名是 `.mp4` 或 `.m4a`。而當我們在討論 HLS 以及 MPEG Dash 時，也需要知道：這些格式也建立在 MP4 格式上，像 HLS 裡頭的每個 TS 往往就是 MP4 檔案，而 MPEG Dash 更是相當倚賴一種叫做 fMP4 的格式。

MP4 格式淵源自蘋果的 QuickTime 格式，所以 MP4 可說與 QuickTime 互通。跟我們在前面講過的格式比較，MP4 container 有很多不同：

- MP4 不只是一種音樂格式，也是一種影片格式—某方面來說，我們可以把 MP4 container，想像成是一種把所有畫面都拿掉的影片
- MP4 是一種樹狀/巢狀的結構
- MP4 格式本身就有 metadata 的區段
- MP4 包含分開來的時間/資料如何對應的區段

我們來看一下每個 MP4 檔案的結構。

## Atoms

MP4 是以 **atom** 所構成。每個 atom，都可以想成是一個樹狀結構的節點，在樹狀結構的根部有一些基本 atom，在這些基本 atom 底下，每個子 atom（即 sub atom）是被一個上層節點所包圍。每個 atom 都有一段 header，header 當中包含這個 atom 的各種資訊，包括這個 atom 的 header 與 body 的長度、種類、ID、有多少子 atom...等（參見蘋果文件 [QT Atoms and Atom Containers](#)，這份文件也說明了 atom 的規格，在這邊不贅述），播放器在知道某個 atom 的意義之後，就可以繼續從這個 atom 的 Body 區段，繼續尋找這個 atom 的子 atom。

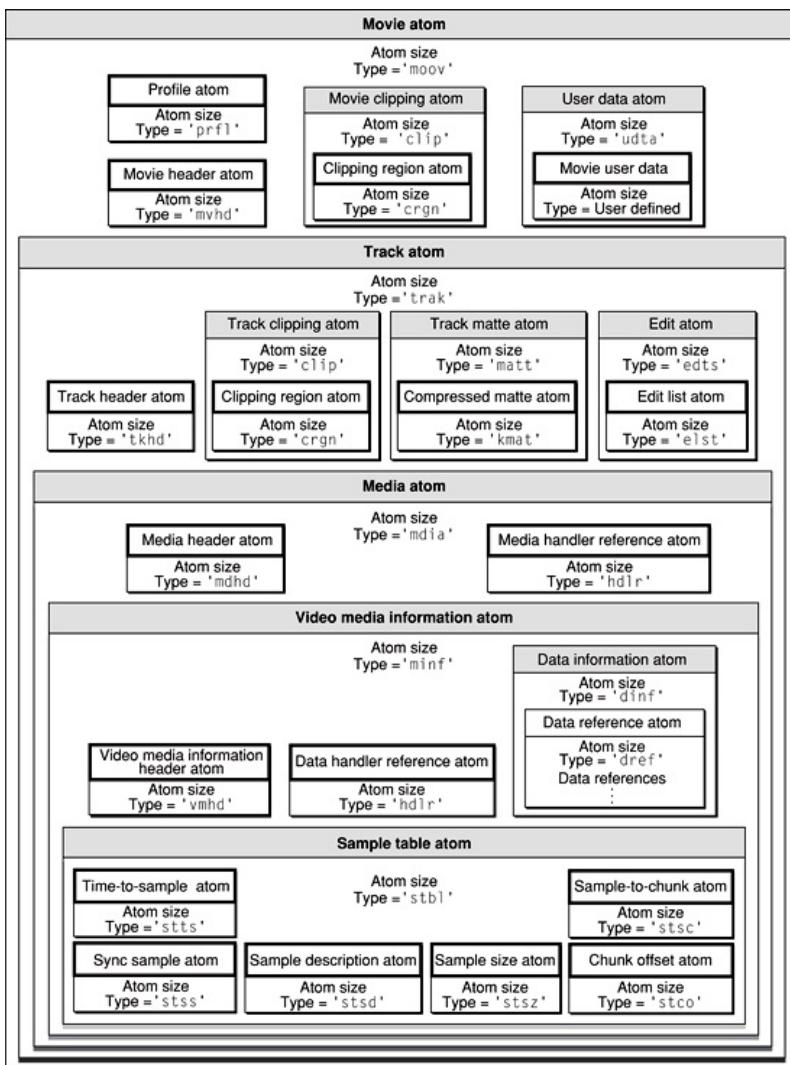
在 MP4 規格中，最上層的 atom 包括是 `ftyp`、`moov`、`mdat` 等最重要的區段，我們可以想像成在連續的 Binary Data 中，先被分成了這幾段，然後在下方還有其他 atom。假如我們拿 MP4 Parse 工具裡頭解析一個 MP4 檔案，就應該可以看到這些 atom：

0	18	<b>ftyp</b>
18	8	<b>free</b>
20	836ba8	<b>mdat</b>
<b>836bc8</b>	<b>30fed</b>	<b>moov</b>

- **ftyp**：用來標示這個 MP4 檔案的類型，像是這個檔案是部影片，或是只有音樂的音檔等
- **moov**：這個檔案相關的 metadata，像是歌手、專輯、歌曲名稱，或是封面圖等等。MP4 可以標示的 metadata 相對來說比較複雜，比方說，一部影片可能有多個章節，那麼，**moov** 中就會包含各個章節的相關資料
- **mdat**：音檔資料實際被存放的位置
- 其他：在 MP4 的規格中，也定義了像是 **free**、**skip**、**wide** 這些類型的 atom，代表的是可以跳過的空資料

## MP4 格式如何達成快速 Random Seek

前面提到，在處理 MP3 或是 AAC-ADTS 格式的時候，必須要把整個檔案從頭到尾讀一遍，才知道有多少 packet，以及 packet 的所在位置，不利於快速計算歌曲的總長度，以及快速 seek 到某個位置播放。我們可以看一下 **moov** 以下有哪些 atom，方便解決 MP3 格式以及 ADTS 格式的問題：



以上圖片來自蘋果開發者網站 [QuickTime File Format Specification: Movie Atoms](#)。

我們可以注意到，在 `moov` 底下，包含像是 `stsz`、`stco` 等 atom，`stsz` 這個 atom 中包含的就是 packet 的數量，`stco` 則是每個 packet 在 `mdat` atom 中的 offset 位置。

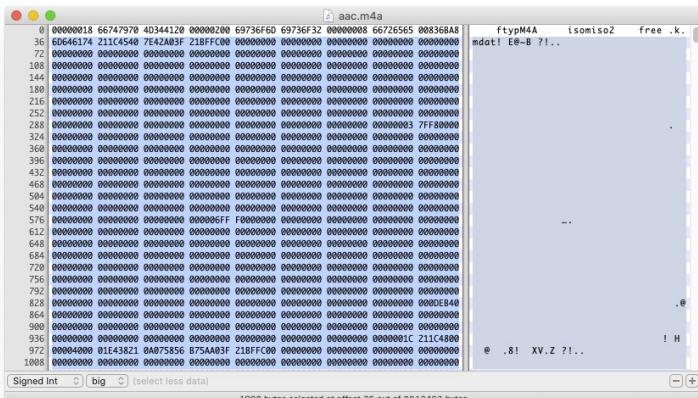
在平台上所提供的全功能播放元件，像是 iOS/macOS 的 AVPlayer，在播放一個位在線上的 MP4 檔案的時候，會發出多條連線，嘗試在檔案的最前方或是最後方找到 `moov` 區段，然後從 `moov` 區段中找到 `stsz`、`stco` 等區段，有了 `stsz`，就可以畫出進度條，提供用戶做 random seek 的 UI，當用戶 seek 到某個地方的時候，就根據 `stco`，發送帶有 Range Header 的 HTTP 連線，從哪個地方抓取，如果沒有 `stsz` 以及 `stco`，才去把 `mdat` 讀過一遍，找出 packet 位置。如果是一份被燒錄在 DVD 或是藍光碟片的大檔，也一樣會嘗試先從檔案的最前方或是最後方，找到 `moov` 區段。

由於播放軟體在載入 MP4 檔案的時候，不是直接載入可以播放的資料，而是先去尋找 `moov` 區段，所以，在播放 MP4 的時候，其實可以感受到，會有一段前置處理的時間。像 Chromecast 上的 audio player，採取的就是這樣的播放行為，每次對 Chromecast 呼叫 random seek，往往就會觸發一個帶 Range header 的連線，而如果已經過了一分鐘才做 random seek，就會因為音檔已經不存在而無法播放，所以，我們在產品上，就做過在 Chromecast 上無法 random seek 的奇妙設計。

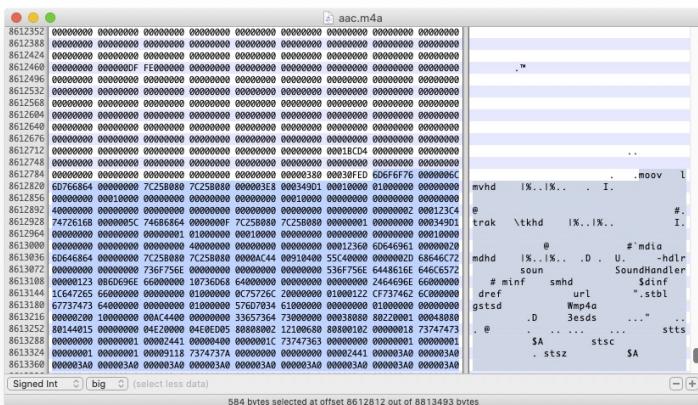
另外要注意：如果我們使用 iOS/macOS 上的 CoreAudio API 的 Audio Parser，像是 `AudioFileStreamID`，就只能夠解析 `moov` 放在 `mdat` 前方的 MP4 檔案，不然就會跳出 `kAudioFileStreamError_NotOptimized` 錯誤，代碼的意思是「這個檔案沒有做過最佳化」，意思其實不清不楚，我們也是花了點時間才了解蘋果的意思。很多時候，我們在 iOS/macOS 上，可以用 QuickTime 或是 AVPlayer 播放的檔案，用 CoreAudio API 却不見得有辦法播出，也就是說，蘋果在自己產品上使用的 Parser，跟開放給外部開發者使用的 API，其實是不同的。

## AAC-MP4 檔案

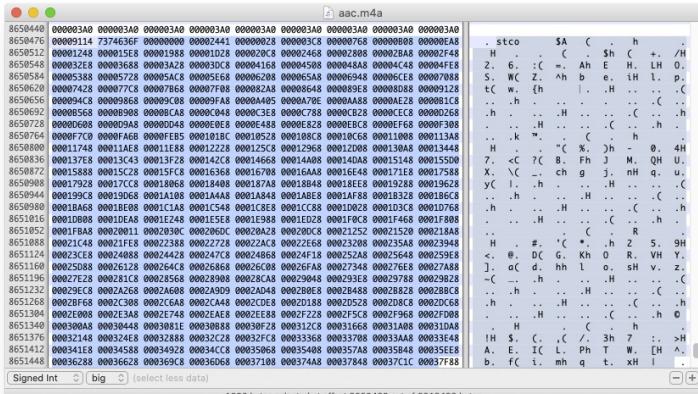
我們也可以拿 Hex Editor，打開一個 AAC-MP4 檔案看看。可以看到在最前方是 `ftyp` 欄位，標示這個檔案是 M4A，然後從第 36 個 byte 開始，就是 `mdat` 區段。



這個檔案的 `moov` 區段位在後方：



在 `moov` 當中包含 `stco` 這段用來方便做 random seek 的 atom：



另外，如果想要解析 MP4 檔案當中的 Atom，在網路上已經有不少現成的工具，光是用 Google 搜尋「MP4 Atom Parser」，就可以找到不少。我們簡單列出幾個：

- mp4parser.com
  - onlinemp4parser.com
  - mp4explorer，用 C# 寫成的 Windows 程式

文件更新時間：2022/02/06 00:44 CST

## FLAC 格式

FLAC 全名為 Free Loseless Audio Codec，是由 [Xiph.org](#) 從 2001 年發展迄今的開放無損音訊格式。相對於 MP3 這些破壞性壓縮格式，所謂的無損（loseless），就是在壓縮的過程中並沒有破壞，我們將 PCM 格式轉換成 FLAC 格式之後，仍然可以將 FLAC 轉換回原本的 PCM 格式。許多平台上都內建了 FLAC codec，此外，我們也可以在網路上，找到 FLAC 的 [程式碼](#)，整合到我們的產品中。

FLAC 除了是一種 codec 之外，本身有一套自己的 container 格式，使用這種格式的 FLAC，叫做 plain FLAC，通常附檔名是 .flac 的檔案，就是這種。另外，也很常見到將 FLAC 資料包裝到 OGG container 中，這樣的檔案叫做 Ogg FLAC，[Xiph.org](#) 也提供解析、播放 Ogg FLAC 的工具。

在 iOS/macOS 上，iOS 11 與 macOS 10.13 開始直接內建 FLAC codec，如果是更早的作業系統，就得要自己把 libFlac 打包到應用程式中。

## Plain FLAC Container

The screenshot shows the official FLAC website. At the top is the logo "flac" with a colorful bar graphic above it, followed by the text "free lossless audio codec". Below the logo is a green horizontal bar containing links: "home" (highlighted), "faq", "news", "download", "documentation", "comparison", "changelog", "links", and "developers". Underneath this bar, the word "format" is bolded in a dark grey header. A paragraph of text follows, stating: "This is a detailed description of the FLAC format. There is also a companion document that describes [FLAC-to-Ogg mapping](#). For a user-oriented overview, see [About the FLAC Format](#)." Below this is a "Table of Contents" section with a list of topics, including "ACKNOWLEDGMENTS", "SCOPE", "ARCHITECTURE", "DEFINITIONS", "BLOCKING", "INTERCHANNEL DECORRELATION", "PREDICTION", "RESIDUAL CODING", "FORMAT", "FLAC SUBSET", "SPECIFICATION", "STREAM", "METADATA\_BLOCK", "METADATA\_BLOCK\_HEADER", "METADATA\_BLOCK\_DATA", "METADATA\_BLOCK\_STREAMINFO", "METADATA\_BLOCK\_PADDING", "METADATA\_BLOCK\_APPLICATION", "METADATA\_BLOCK\_SEEKTABLE", "SEEKPOINT", "METADATA\_BLOCK\_VORBIS\_COMMENT", "METADATA\_BLOCK\_CUESHEET", "CUESHEET\_TRACK", "CUESHEET\_TRACK\_INDEX", "METADATA\_BLOCK\_PICTURE", "FRAME", "FRAME\_HEADER", and "FRAME\_FOOTER".

根據 FLAC 規格，一個 FLAC 檔案是由以下成分所組成：

- FLAC 檔案的前四個 bytes，是 "fLaC" 四個字元。
- 在 "fLaC" 四個字元之後，會有多個 metadata block 組成，這些 metadata block 可以任意改變順序，但是需要標示，哪一個 metadata block 是最後一個 block。這些 block 的種類包括
  - Stream info：包括像是 sample rate 等用來讓 player 解析後面 data frame block 的資訊，一定要有這一段。代號 0。
  - Cue sheet：演出表，包括歌名、專輯，以及有哪些歌手在當中演出。非必要。代號 5。
  - Vorbis comment：一些註解欄位。非必要。代號 4。
  - Application：標示這個 FLAC 檔案是由哪個應用程式製作的。非必要。代號 2。
  - Picture：封面圖片。非必要。代號 6。
  - Seektable：就是在 frame block 區段的每個 block 的 offset，方便 client 找到 offset 時，可以快速 seek 到指定區段，像是我們在介紹 MP4 格式時講到的 `stsz`、`stco` 的用途。非必要。代號 3。

- Padding：一些用來補足長度用的空資料。非必要。代號 1。
- Frame Block：基本上，雖然說用詞不同，但也就是我們在前面所說的 packet。Frame block 是由 frame header、footer 與 subframe 組成，在 frame header 的開頭一樣有 syncword，syncword 總共 14 的 bit，為 "111111111111110"。每個 FLAC 檔案中的 frame block 的大小可能是不同的，要根據 Stream info 當中的內容，才有辦法知道這個檔案中每個 frame block 的大小為何。

所以一個 FLAC 檔案中的資料大概如下圖：



## 實作 FLAC 檔案的分段載入

因為 FLAC 格式一樣是由連續的 packet 組成，所以我們還是可以從一個 FLAC 檔案的中間開始抓取、播放，我們也可以看到有 FLAC 格式的高音質網路廣播電台。不過，要能夠播放局部的 FLAC 資料，我們還是要補上最前方的 "fLaC" 四個字元，以及最少一個 metadata block：stream info。

想要提供給 player 一段 stream info，我們大概有幾種方式：1. 發送多個連線，先抓取 FLAC 檔案最前方一定數量的資料，然後丟給 player 解析、播放，不過，由於 metadata block 並不保證順序，所以我們其實無法正確預期 stream info 一定位在我們所抓取的範圍內。；2. 自己產生 stream info 區段的資料，不過，我們也不見得可以保證，這個 FLAC 檔案的格式，一定可以符合我們的預期。

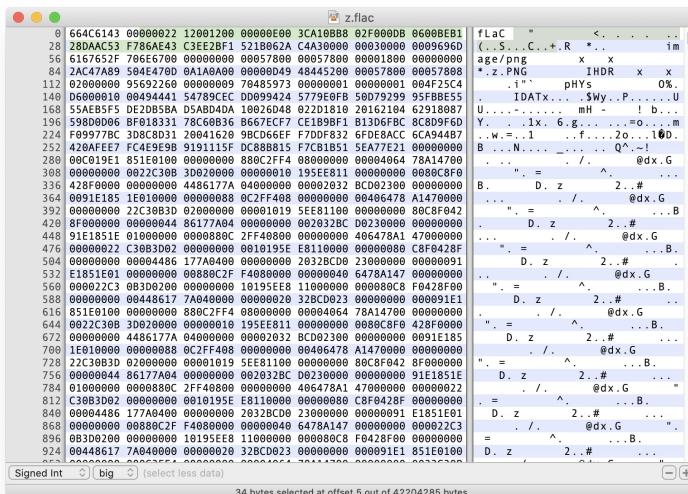
如果我們想要自己產生 stream info，就得來了解這段資料的格式：stream info 的長度為 34 個 byte，裡頭的數字使用 Big endian 整數。包括：

- The minimum block size (in samples)，16 bits，必填
- The maximum block size (in samples)，16 bits，必填
- The minimum frame size (in bytes)，24 bits，可填零
- The maximum frame size (in bytes)，24 bits，可填零
- Sample Rate，20 bits，必填

- 聲道數 -1，FLAC 最多可以支援 8 個聲道，3 bits，必填
- 每個聲道所用 bits 數 -1，可為 4-32，5 bits，必填
- 整個檔案的 Sample 數量，36 bits，可填零
- MD5 checksum，128 bits，可填零

在 34 個 bytes 的前面，我們還要用一個 byte 表示這是一個 metadata block，其中第一個 bit 表示這是不是最後一個 metadata block，既然我們只想要一個 metadata block，那就填 1，剩下七個 bits 則用來表示 metadata block 的種類，stream info 的代號為 0，所以我們就填入 "10000000"。加上 "fLaC" 四個 bytes，我們總共要產生 39 個 bytes 的資料。

我們可以實際打開一個 FLAC 檔案看看：



我們可以看到，檔案開始的前 4 個 bytes，就是 "fLaC" 四個字，第五個 byte 為 0：第一個 bit 是 0，代表後面還有其他的 metadata block，另外 7 個 bits 也是 0，代表這段是 stream info，然後就是 34 個 bytes 的 stream info 的內容（圖片中被選取起來的地方）一不過，我們就得要讀出每個 bits，才能了解當中的意義了。至於 stream info 之後，是第二個 metadata block，我們可以判斷出，這邊放了一張封面圖片。

## FLAC 網路廣播電台

前面提到，網路上有使用 MP3 格式的網路廣播電台，而用戶可以隨時抓到一段資料就可以開始播放，是因為找到 syncword 之後，就可以解析出 packet，而 AAC 格式的網路廣播電台，則是使用 AAC ADTS container 格式。那，FLAC 格式呢？

FLAC 格式的高音質網路廣播電台，其實並沒有用到我們前面提到的這些自己產生 FLAC 檔頭的相關工作。我們可以看到的 FLAC 網路廣播，大概都被包裝在 OGG container 裡頭，而這種網路廣播的 server 軟體，像是 [icecast](#)，在 client 端連上時，就會回應 OGG 的檔頭，讓 client 端知道如何解析接下來拿到的資料。

文件更新時間：2022/02/06 00:44 CST

# HLS 與 FairPlay DRM

HLS 全名是 **HTTP Live Streaming**，是蘋果在 2009 年時所推出的網路直播技術—跟前面提到的 MP4 格式一樣，HLS 也不是只用在跟聲音有關的應用，我們不太應該把 HLS 當成是一種音檔，只是 HLS 的應用範圍也包括聲音的串流。

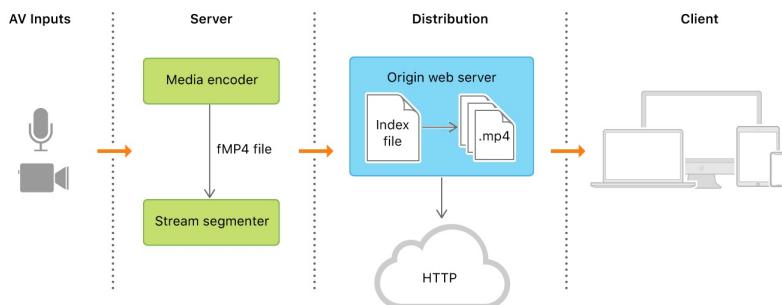
HLS 在 21 世紀第一個十年快要結束的時候推出，最大的意義就是，HLS 技術一口氣擴充了一場網路影片直播的同時人數上限。HLS 以及後來所推出的各種新格式，像是 MPEG-DASH 等，讓我們現在所習慣的萬人同時觀看直播，成為可能。

在 HLS 推出之前，在二十一世紀的第一個十年，主流的影片直播技術大概是 Windows Media Server、Adobe Media Server 等，這些技術的最大共通性，是讓播放影片直播的 client 與 server 之間建立 socket 連線，中間使用像是 MMS、RTMP、RTSP …等協定溝通，server 不斷把最新的影音資料推送到各個 client 端，而單一 server 能夠接受的網路連線是有限的，所以，一場網路影片直播，最多大概上百人同時觀看，就已經到達極限。

HLS 改變 client/server 之間建立 socket 連線的架構，從 HLS 的名稱就可以看出，利用 HTTP 技術解決人數上限問題。在直播訊源與眾多觀看直播的 client 之間，多了幾個步驟：首先直播訊源會將資料編碼成 fMP4（我們在下一章討論）格式之後，送到一台主機上，這台主機會把每隔一段時間（比方說，每隔 5 秒、10 秒…）的影片，轉成一個小檔案—這種將連續的資料變成一個個資料切片的程式，我們叫做 segmenter，每個被切出來的小檔案叫做 ts（MPEG Transport Stream，也是一種封裝格式，因為我們不太會去解析這套格式，就不解釋了），這些被切出來的小檔案會被佈署到 CDN 上，另外產生一個文字檔，附檔名是 .m3u8，是這些小檔案的 playlist，指示 client 如何播放這些 ts，而當有新的 ts 被產生、佈署，這個 .m3u8 檔案也會隨之更新。

在 HLS 規格中，除了 TS 之外，**HLS 的切片也支援使用 fMP4 封裝**。但使用 fMP4 檔案封裝的 HLS 流，在多數平台都不被支援，蘋果平台也僅在 macOS 10.12, iOS 10, tvOS 10 之後的版本才支援播放。本章節後續均基於 TS 切片描述，fMP4 檔將在 MPEG-DASH 章節介紹。

以下圖片來自蘋果的 [HLS 技術官方頁面](#)：



Client 端在播放的時候，首先抓取 `.m3u8` 檔案，檢查當中有哪些 ts 檔案，在直播的狀況下，可能在 `.m3u8` 檔案中，就留了最新的三個 ts 檔，client 端就從第一個 ts 檔案開始輪流抓取、播放，在快要把這三個 ts 檔都快要播完的時候，就再去檢查一次 `.m3u8` 檔案，查看是不是還有更新的 ts 檔案，如果有，就繼續抓取這些新的 ts 檔。這個步驟就不斷的輪迴，直到發現沒有更新的 ts 檔案，代表直播已經結束。

由於檔案是佈署在 CDN 上，用戶也是從距離他最接近的 CDN 抓取檔案，而 Web server 所能夠負載的連線，遠大於單一讓眾多 client 建立 socket 連線的 server。隨著 21 世紀第二個十年開始，幾大 CDN 廠商的業務規模擴張，CDN 技術愈來愈普及，像 [Akamai](#) 這家 CDN 服務，大概在 HLS 問世時推出服務，隨著一起壯大，HLS 以及後來的新技術，也整個代替了之前的各種影片直播技術，HLS 也在 2017 年成為 [RFC 8216 規格](#)。目前業界所使用的直播技術，大概都是 HLS 以及之後的其他新技術。

HLS 雖然最早用在直播，但之後也陸續應用在影片以及音檔的用途上。假如一個 `.m3u8` 檔案中的 ts 不會隨時間更新，而是靜態的，我們不是將直播訊號切成小檔，而是將一部電影或是影片的大檔切成小檔，那麼，這個 `.m3u8` 串流就可以是一部電影或是電視劇。而如果把靜態的 `.m3u8` 中的 ts，從影片再換成音檔，那也就是一個被切成眾多小檔案的歌曲串流了。

## 直播的延遲

雖然與我們跟音檔的相關討論比較沒有關係，不過我們可以來聊一下，在直播過程中會發生的延遲（latency）。細數一下從訊源到最後的收視端，我們可以發現，每一個步驟，都需要耗費時間處理，導致觀看者看到某個事件時，已經與事件實際發生的時間之間，有一段時間差：

- 負責錄影/錄音的設備，需要拍攝一定數量的影格/聲音（其實英文都是 frame），才有辦法累積成 packet，透過 RTMP 發送到負責切檔的主機
- RTMP 傳送資料到主機上的傳輸時間
- 主機需要累積一定數量的 packet，才能夠產生 ts。由於 HLS 支援多種不同品質的影像/聲音的 track，所以也往往會一次產生多個不同品質的 ts
- 將 ts 以及新的 .m3u8 檔案佈署到 CDN 上的時間
- 播放端 client 抓取 .m3u8 的時間
- 播放端 client 抓取 ts 檔案，到有足夠數量的 packet 可以播放的時間，這段時間也視用戶的頻寬品質而有所不同
- 因為 HTTP proxy 的 cache，用戶也可能抓到比較舊的檔案

所以，從事件發生，到用戶實際觀看到，往往會有大概 15 秒到半分鐘左右的延遲。蘋果在 2019 年 WWDC 時推出新的 [Low Latency HLS 規格](#)，這個新規格用了一些技巧，想辦法減少直播過程當中的延遲，大概是讓用戶盡可能去抓取最新的 ts，然後在 .m3u8 檔案中加上時間相關資訊，在抓到 ts 之後直接跳到最新的指定秒數，並且盡量不要被 proxy cache 所影響，可以參考 [WWDC 2019 年的說明](#)。但這樣的 HLS server 的實作在現在（2020 年初）還不普及，播放端也需要最新的作業系統與播放軟體支援，要等到普及，相信還要花上一段時間。

## .m3u8 檔案

我們可以打開 Akamai 的一個範例 [m3u8 檔案](#)，看看裡頭的內容：

```
#EXTM3U
#EXT-X-VERSION:5

#EXT-X-MEDIA:TYPE=AUDIO,GROUP-ID="audio",NAME="English stereo",LANGUAGE="en",
AUTOSELECT=YES,URI="f08e80da-bf1d-4e3d-8899-f0f6155f6efa_audio_1_stereo_12800
0.m3u8"

#EXT-X-STREAM-INF:Bandwidth=628000,Codecs="avc1.42c00d,mp4a.40.2",Resolution=
320x180,Audio="audio"
f08e80da-bf1d-4e3d-8899-f0f6155f6efa_video_180_250000.m3u8
#EXT-X-STREAM-INF:Bandwidth=928000,Codecs="avc1.42c00d,mp4a.40.2",Resolution=
480x270,Audio="audio"
```

```
f08e80da-bf1d-4e3d-8899-f0f6155f6efa_video_270_400000.m3u8
#EXT-X-STREAM-INF: BANDWIDTH=1728000, CODECS="avc1.42c00d, mp4a.40.2", RESOLUTION
=640x360, AUDIO="audio"
f08e80da-bf1d-4e3d-8899-f0f6155f6efa_video_360_800000.m3u8
#EXT-X-STREAM-INF: BANDWIDTH=2528000, CODECS="avc1.42c00d, mp4a.40.2", RESOLUTION
=960x540, AUDIO="audio"
f08e80da-bf1d-4e3d-8899-f0f6155f6efa_video_540_1200000.m3u8
#EXT-X-STREAM-INF: BANDWIDTH=4928000, CODECS="avc1.42c00d, mp4a.40.2", RESOLUTION
=1280x720, AUDIO="audio"
f08e80da-bf1d-4e3d-8899-f0f6155f6efa_video_720_2400000.m3u8
#EXT-X-STREAM-INF: BANDWIDTH=9728000, CODECS="avc1.42c00d, mp4a.40.2", RESOLUTION
=1920x1080, AUDIO="audio"
f08e80da-bf1d-4e3d-8899-f0f6155f6efa_video_1080_4800000.m3u8
```

這是一部影片的 `.m3u8` 檔案。每個 `.m3u8` 檔案以 `#EXTM3U` 開頭，然後接著一些基本的 metadata，像，這部影片裡頭用的語文是英文。

在這裡，我們可以看到，這份 HLS 串流支援多種不同的品質，影像解析度最低到 320x180、最高到 1080p (1920x1080)。client 端的播放器可以根據現在的頻寬品質，從多個不同品質的串流中，選擇目前最適合、能夠順暢播播放的最高品質，而當頻寬品質改變時，也會切換到另一個最適合的串流。這種可以動態使用最適合品質的串流播放方式，叫做 [Adaptive Streaming](#))。雖然有多種不同的影片格式，在聲音方面，則共用同一個 128k 的聲音 "f08e80da-bf1d-4e3d-8899-f0f6155f6efa\_audio\_1\_stereo\_128000.m3u8"。

然後我們可以看一下 [f08e80da-bf1d-4e3d-8899-f0f6155f6efa\\_video\\_180\\_250000.m3u8](#) 這個影片相關的 `.m3u8` 檔案當中的內容：

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-MEDIA-SEQUENCE:0
#EXT-X-TARGETDURATION:4

#EXTINF:4.0
../video/180_250000/hls/segment_0.ts
#EXTINF:4.0
../video/180_250000/hls/segment_1.ts
#EXTINF:4.0
../video/180_250000/hls/segment_2.ts
#EXTINF:4.0
../video/180_250000/hls/segment_3.ts
#EXTINF:4.0
../video/180_250000/hls/segment_4.ts
#EXTINF:4.0
../video/180_250000/hls/segment_5.ts
....
```

當中就是一連串的 ts 檔案，在 `#EXTINF` 中，則描述了每個 ts 檔案的長度是 4.0 秒。

蘋果在 2016 年的時候宣布，除了可以將檔案切成 ts 之外，也可以支援 fMP4 格式的 byte range，如此一來，會讓 HLS 與後面要講到的 MPEG Dash 格式進一步相容。不過，我們也是留在下一章討論。

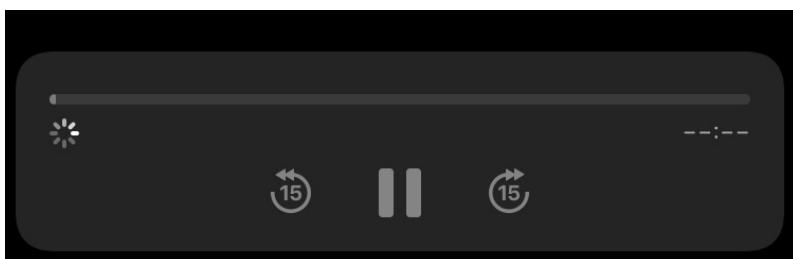
## 播放 HLS

HLS 格式是開放的，所以，除了蘋果的方案以外，第三方也可以自己實作 HLS 的 server、segmenter 與 client 端。

我們可以看到，目前有許多的影音托管服務，像是 [Wowza](#) 等，可以幫想要播放內容的使用者產生 HLS 串流，也可以同時產生 MPEG Dash 等其他格式，如果上傳的是聲音訊號，也可以產生 shoutcast、icecast 形式的網路聲音廣播。

在播放軟體方面，蘋果官方的播放元件，自然是 AVPlayer。我們也可以看到一些第三方的實作，像是 [JWPlayer](#)、[Kaltura Player](#) 等等，都可以播放 HLS。但是能不能夠播放 FairPlay 保護的 HLS，就要看各家 Player 的支援程度。

由於 HLS 同時可以用在直播，或是播放線上的影片或是音樂，所以就會有兩種播放 HLS 的模式：在播放線上的影片或音樂的狀態下，我們可以任意跳到影片或音樂的任一位置，而播放軟體在跳到指定位置的時候，也只需要從 `.m3u8` 檔案中，找到符合時間位址的 ts 載入；至於在直播的模式下，由於我們只有少部分的 ts 可以使用，所以，播放器介面就往往呈現成「往前十秒」或「往後十秒」播放。



而在支援的音訊格式方面，TS container 並沒有限制當中的資料格式。所以，如果將 HLS 用在音訊的串流上，在 iOS/macOS 上，只要是 Core Audio 所支援的格式，包括 MP3、AAC、FLAC...，都可以使用 HLS 發佈。

## FairPlay Streaming

[FairPlay Streaming](#) 簡稱 FPS，是蘋果用來保護 HLS 串流的商用 DRM 機制。我們要解釋一下 FairPlay 與 FairPlay Streaming 這兩個名詞的差異，FairPlay 是一套蘋果用在各種產品上的 DRM 技術，像是 iTunes Store 上面購買的音樂、App Store 上購買的 app，蘋果都使用 FairPlay 保護，不過，只有用來保護 HLS 的 FPS，才從 2015 年起開放給外部使用。所以，當我們在蘋果外部講到 FairPlay，講的往往是 FPS。而即使蘋果開放外部使用，我們還是要額外跟蘋果申請以及簽約。

在一個受到 FPS 保護的 HLS 串流中，會把每個 ts 檔案加上 AES 加密，client 端要能夠播放，就必須要取得這把 key，FPS 的重點就是如何保證這把 key 在傳遞的過程中不會被外部破解。因為是被蘋果保護起來，所以我們也搞不清楚實際上蘋果做了什麼，根據蘋果的文件，我們只要照以下這麼做就對了：

找到 **Asset ID**，把 Asset ID 跟 **Certificate** 結合起來產生 **SPC**，傳到 server 上，如果可以從 server 成功拿到 **CKC**，把 CKC 塞進播放元件，就可以播了。

…這一段話裡頭充滿了許多奇妙的關鍵字，像是 SPC、CKC…一定要解釋一下才有辦法理解。我們在下面會解釋一下在 iOS/macOS app 上的流程。當中不會講解太多實際在平台上會呼叫哪些 API，但是至少得解釋像 CKC、SPC 這些關鍵字，才有辦法了解 FPS。

## 處理 FairPlay Streaming 的流程

### 先從 .m3u8 中解析出 asset ID

一個被 FairPlay 保護起來的 .m3u8 檔案，裡頭會有像這樣的一段：

```
#EXTM3U
...
#EXT-X-KEY:METHOD=SAMPLE-AES,URI="skd://f3c5e0361e6654b28f8049c778b23946+5c3d
```

```
e6de8c2bb1295712e8da387add37", KEYFORMATVERSIONS="1", KEYFORMAT="com.apple.streamingkeydelivery"
```

"f3c5e0361e6654b28f8049c778b23946+5c3de6de8c2bb1295712e8da387add37" 這一段，就是一個 asset ID，是 FairPlay 機制當中用來表示一個特定媒體，像是特定影片、歌曲的方式。在播放一個 HLS 串流，播放器就會嘗試去尋找是否有這段資訊，如果沒有，就代表沒有加上保護，播放元件就會直接播放這個 HLS 串流。

在 iOS/macOS app 中，我們會先把 `.m3u8` 串流放在一個 AVAsset 中，在嘗試載入這個 AVAsset 的時候，如果遇到被保護、需要 Key 的 HLS，就會觸發對應的 delegate method。在 Safari 瀏覽器中，我們可以對 video tag 的 "webkitneedkey" 事件加上 listener，然後，這個 video tag 播放的影片需要有一把 Key 時，就會觸發後續的行為。

## 產生、上傳 SPC，取得 CKC

SPC 全名叫做 Server Playback Context，簡單來說，就是用兩種來源的資料，透過蘋果的一套被保護起來的演算法，產生出來的授權需求資料，我們往 server 上傳這份資料之後，就可以拿到最後可以用來播放的 key—不過，這把 Key 也是被保護起來的，這個把 Key 包起來的資料，叫做 CKC，Content Key Context。

我們在通過蘋果的審核，可以開始使用 FairPlay 之後，我們可以在蘋果的開發者後台上建立一份 FPS 使用的 Certificate，因為我們可以隨時更新這份 certificate，所以不太應該直接放在 client 端，而是應該放在一個 Web API 上，讓 client 端在有需要的時候抓取。

在 iOS/macOS 平台上，我們接下來要呼叫 `AVAssetResourceLoadingRequest` 的 `streamingContentKeyRequestData(forApp:contentIdentifier:options:)`，這邊 `forApp:` 要傳入的，就是上面提到的 certificate，`contentIdentifier` 則是 asset ID；這個 API 回傳的資料，就是 SPC。至於這個 method 後面做了什麼事情，沒有人知道。

在 Safari 瀏覽器中，在 `webkitneedkey` 被呼叫的時候，則要結合 asset ID 與 certificate，呼叫 video tag 的 `webkitSetMediaKeys`，以及 `WebKitMediaKeys` 的 `createSession`。呼叫方式，可以參考 FPS SDK（可以從蘋果官網下載）中的範例程式：

```

    }

    function onneedkey(event)
    {
        var video = event.target;
        var initData = event.initData;
        var contentId = extractContentId(initData);
        initData = concatInitDataAndCertificate(initData, contentId, certificate);

        if (!video.webkitKeys)
        {
            selectKeySystem();
            video.webkitSetMediaKeys(new WebKitMediaKeys(keySystem));
        }

        if (!video.webkitKeys)
            throw "Could not create MediaKeys";

        var keySession = video.webkitKeys.createSession("video/mp4", initData);
        if (!keySession)
            throw "Could not create key session";

        keySession.contentId = contentId;
        waitForEvent('webkitkeymessage', licenseRequestReady, keySession);
        waitForEvent('webkitkeyadded', onkeyadded, keySession);
        waitForEvent('webkitkeyerror', onkeyerror, keySession);
    }

    /*
     * This function assumes the Key Server Module understands the following POST format --
     * spc=<base64 encoded data>&assetId=<data>
     * ADAPT: Partners must tailor to their own protocol.
     */
    function licenseRequestReady(event)
    {
        var session = event.target;
        var message = event.message;           ← 結合 Asset ID 與 certificate
        var request = new XMLHttpRequest();
        var sessionId = event.sessionId;
        request.responseType = 'text';
        request.session = session;
        request.addEventListener('load', licenseRequestLoaded, false);
        request.addEventListener('error', licenseRequestFailed, false);
        var params = 'spc=' + base64EncodeUint8Array(message) + '&assetId=' + encodeURIComponent(session.contentId);
        request.open('POST', serverProcessSPCPath, true);
        request.setRequestHeader('Content-type', "application/x-www-form-urlencoded");
        request.send(params);
    }
}

```

我們的 Server 需要準備另外一支 Web API，這支 API 後面，要橋接蘋果在 FairPlay SDK 裡頭的 Key Server Module，拿蘋果的 library 來處理 client 端上傳的 SPC，回傳的結果，就是 CKC，把 CKC 塞到 Player 上（在 iOS/macOS app 中，呼叫 [AVAssetResourceLoadingDataRequest 的 respond\(with:\)](#)）；至於在 Safari 瀏覽器中，則是呼叫 [WebKitMediaKeySession 的 update](#)。

換句話說，一個完整的 FairPlay server 應該要有兩個 API end point：1. 下載 certificate、2. 驗證 SPC 並回傳 CKC。

## FairPlay Streaming 近年的演變

蘋果在 2015 年釋出公開版本的 FPS 時，只提供在播放的過程中抓取 CKC 的流程，之後，蘋果也陸續擴充 FPS，在每年的 WWDC 公布更多 FPS 的新功能。這些新功能都在 iOS/macOS 的 app 端，在 Web 上，

### WWDC 2016 - Offline HLS

蘋果在 2016 年推出 Offline HLS，可以讓第三方廠商在自己的 app 中，將 HLS 串流下載到本地端離線播放，而 HLS 的 Key 也分成了兩種，一種是播放用的（Streaming Key）、另外一種是離線儲存用的（Persistent Key）。與 Offline HLS 的相關資訊參見 [What's New in HTTP Live Streaming](#)。

## WWDC 2017 - Content Key Session

蘋果在 2017 年推出 Content Key Session，把抓取 CKC 的流程，與播放流程區分開來。

這個設計在於解決幾個問題：

- 如果某個直播活動會在某個時間開播，當直播開始之後，就會有大批的用戶湧入，當所有用戶都同時跟 server 要求 CKC 的時候，就會產生很大的瞬間流量。所以，我們就希望可以在活動開播之前，用戶就完成 DRM 的驗證流程，拿到 CKC，而不要到了開播之後才做。
- 在 2016 年推出 Offline HLS 之後，可以設定一份 Offline HLS 在固定時間過期，但是在過期之後，原本的機制，只能夠在用戶重新播放的時候更新離線憑證，不能事先讓用戶知道已經過期，使用體驗並不友善。

Content Key Session 獨立於播放元件之外，可以讓第三方廠商，可以在適當的時機，就抓取、更新 CKC；此外蘋果增加了 HEVC 影像格式支援、IMSC1 字幕、新的下載管理員元件…等。參見 [Advances in HTTP Live Streaming](#)。

跟 Content Key Session 有關的部分，也可以參見 WWDC 2018 的 [AVContentKeySession Best Practices](#)

## WWDC 2019 - Low Latency HLS

如前所述，蘋果推出 Low Latency HLS 規格，嘗試降低 HLS 的延遲時間。

## 使用 HLS/FairPlay Streaming 的優缺點

HLS/FairPlay Streaming 是在 iOS/macOS/tvOS 等蘋果平台，以及 Safari 瀏覽器中播放直播串流—尤其是受保護的串流—的首選，我們直接享受平台官方的套件以及支援服務。

但，用來播放 HLS 的元件，其實是設計給影片，而不是設計給音樂服務用的—如果只是 HLS，我們還有一些其他選擇，但如果要播放 FairPlay DRM 保護的內容，就只有 AVPlayer 等官方的元件—影片播放軟體並不會考慮一些音樂播放的特殊情境，像是一些特殊的迴音、等化器效果，或是會讓兩首歌曲重疊的淡入淡出效果—沒有人會把兩部電影放在一起混音播放，但是在音樂上卻是非常常見的情境。

拿 HLS 實作音樂播放，就得同我們要捨棄這些音訊效果，如果我們是打造一套全新的服務，可能還可以從一開始就不打算支援；但如果一套產品上線已久，這些效果，是已經存在的產品功能，那是否應該改用 AVPlayer 以及 HLS 這一整套技術，就得要做全盤的評估。

另外，就是 Offline HLS 的離線播放授權憑證更新機制。每部 HLS 下載之後，都有各自的 CKC，在更新憑證的時候，也要一部一部影片各自更新。由於這些機制主要為了影片而設計，線上影視服務的用戶，大概只會下載一定數量的影片，而且往往看過一次就刪除，用戶往往會重複播放相同的音樂，但很少重複觀看相同的電影。如果拿 Offline HLS 做音樂服務的離線下載功能，我們也可以預期，如果用戶下載了大量歌曲，使用體驗可能並不理想。

而這整套技術，都與蘋果的生態系綁得非常緊密，如果要在其他平台上使用商用 DRM 保護內容，往往還是得挑選其他的方案。

文件更新時間：2022/02/06 00:44 CST

# MPEG-DASH 與 Widevine DRM

HLS 與蘋果生態系綁得非常緊密，在蘋果的生態系之外，就經常可以看到使用 MPEG-DASH 做為線上串流的解決方案。

DASH 是 Dynamic Adaptive Streaming over HTTP，大概可以翻譯成「基於 HTTP 上的動態適應串流」（通常不太會有人這樣稱呼），跟 HLS 一樣，可以用在直播/影片/音訊等各種媒體應用上。DASH 是一套 MPEG 公開標準，並且與許多的軟硬體廠商組成一個聯盟—[DASH IF](#) (DASH Industry Forum)，成員包括 Google、微軟、SONY、Intel、Akamai...等（參見 DASH IF 的[頁面](#)）。

DASH 也是目前 Google 主要推廣的串流技術，我們可以看到 Google 為 Dash 格式做了許多播放軟體以及相關工具，而 [Widevine](#) 便是 Google 用來保護 Dash 格式包裝的內容的 DRM 機制，用在 Chrome 瀏覽器、Android 行動裝置與 Android TV 上。

從名稱上就可以看出，Dash 與 HLS 相近，也是一套建立在 HTTP 的基礎上，利用 HTTP server 的能力承受大量連線與流量。所以，在 Dash 與 HLS 之間，雖然是不同的格式，但是有一些觀念可以互通：例如，HLS 的 playlist 檔案是 [.m3u8](#) 檔案，在 Dash 中可以對應到 **MPD**，在 HLS 中，實際的影音資料是放在 TS 檔案中，在 Dash 中則使用 **fMP4** (Fragmented MP4) 格式。

## MPD

播放 HLS 的時候，播放器要先載入 [.m3u8](#) 檔案，在播放 Dash 的時候，則要讓播放器載入 MPD，全名是 Media Presentation Description。

MPD 的格式與 [.m3u8](#) 有些不同，[.m3u8](#) 檔案是一行一行寫下來的文字格式，MPD 則是 XML 格式，當中記錄了這個串流（影片或聲音）的基本資料，以及實際上要載入的媒體資料位置。

在 DASH-IF 網站上，有一些測試用的 MPD，可以讓我們了解 Dash 規格。我們來打開其中一個看看，[48K AAC LC Stereo Beeps](#) 是一個只會連續發出一個小時嘩嘩聲的純 audio 串流，當中內容是這樣的：

```

<MPD xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="urn:mpeg:da
sh:schema:mpd:2011" xsi:schemaLocation="urn:mpeg:dash:schema:mpd:2011 DASH-MP
D.xsd" profiles="urn:mpeg:dash:profile:isoff-live:2011,http://dashif.org/guid
elines/dash-if-simple" maxSegmentDuration="PT2S" minBufferTime="PT2S" type="s
tatic" mediaPresentationDuration="PT1H">
    <ProgramInformation>
        <Title>Media Presentation Description for audio only from DASH-IF live
simulator</Title>
    </ProgramInformation>
    <Period id="precambrian" start="PT0S">
        <AdaptationSet contentType="audio" mimeType="audio/mp4" lang="eng" segm
entAlignment="true" startWithSAP="1">
            <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main"/>
            <SegmentTemplate startNumber="1" initialization="$RepresentationID$/init.mp4"
duration="2" media="$RepresentationID$/Number$.m4s"/>
            <Representation id="A48" codecs="mp4a.40.2" bandwidth="48000" audioS
amplingRate="48000">
                <AudioChannelConfiguration schemeIdUri="urn:mpeg:dash:23003:3:aud
io_channel_configuration:2011" value="2"/>
            </Representation>
        </AdaptationSet>
    </Period>
</MPD>

```

在檔案的開頭，我們先看到一堆 XML 宣告。這個 XML 的 root node 是 MPD，然後 XML 的 schema 位在哪個位置...。接下來 ProgramInformation 是關於內容的描述，我們知道這個影片的標題是「Media Presentation Description for audio only from DASH-IF live simulator」。

然後是 Period 與 AdaptationSet。Period 表示的是這個節目有哪些時間區段，比方說，如果這個 MPD 表示的是一場演唱會，那麼，我們就可能會用上多個 Period，表示演唱會當中每一首歌的時間區間。在這段時間內，可以有多種不同品質（音質或畫質）的媒體可以選用—比方說，如果是影片的話，就可能有 360p、480p...等等不同的解析度—播放器會根據目前的網路狀況選擇最適合的。

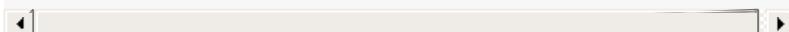
AdaptationSet 就代表一組可用的畫質/音質，相關的檔案位置與 codec 資訊...等。在這個檔案中，由於只有一種音質的資料，所以也就只有一組 AdaptationSet，在有多種品質可以挑選的 MPD 中，就會看到多組 AdaptationSet。

從 SegmentTemplate 中，我們可以看到音檔的 URL 的組合規則：

```

<SegmentTemplate startNumber="1" initialization="$RepresentationID$/init.mp4"
duration="2" media="$RepresentationID$/Number$.m4s"/>

```



這代表，根據 `initialization` 當中的資料，我們應該首先載入 `init.mp4`，然後根據編號（\$Number\$ 是一個變數，是從 1 開始的一連串數字）逐一載入後面的 m4s 檔案。在組成 URL 的時候，首先根據 MPD 所在的位置，找出這些檔案應該所在的根目錄路徑。

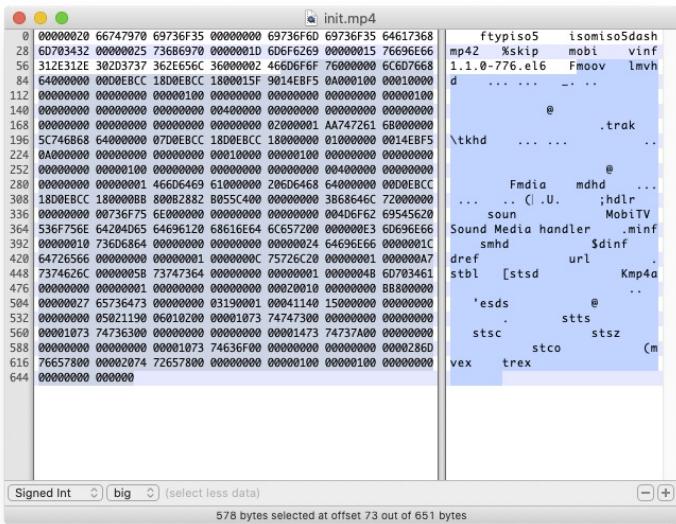
- 這個 MPD 位在 [https://livesim.dashif.org/dash/vod/testpic\\_2s/audio.mpd](https://livesim.dashif.org/dash/vod/testpic_2s/audio.mpd)
- 所以這些檔案應該位在 [https://livesim.dashif.org/dash/vod/testpic\\_2s/](https://livesim.dashif.org/dash/vod/testpic_2s/) 之下
- 我們要填入 `RepresentationID`，從 MPD 中可以看到，這邊的 ID 是 `A48`，所以再把這段加到 URL 上：  
[https://livesim.dashif.org/dash/vod/testpic\\_2s/A48/](https://livesim.dashif.org/dash/vod/testpic_2s/A48/)
- `init.mp4` 就位在 [https://livesim.dashif.org/dash/vod/testpic\\_2s/A48/init.mp4](https://livesim.dashif.org/dash/vod/testpic_2s/A48/init.mp4)
- 後面的檔案規則就是  
[https://livesim.dashif.org/dash/vod/testpic\\_2s/A48/1.mps](https://livesim.dashif.org/dash/vod/testpic_2s/A48/1.mps)，依此類推

## fMP4

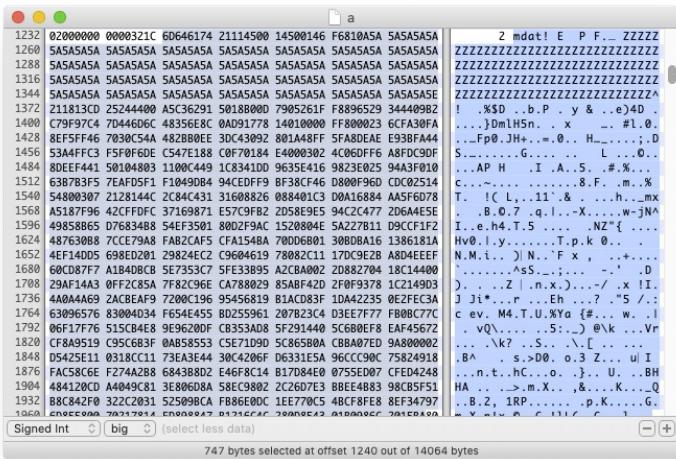
我們在講解 AAC 格式的時候提到 MP4 格式，也講到一個單獨的 MP4 的檔案，裡頭需要 `ftyp`、`moov`、`mdat` 這些基本的 atoms。

在 DASH 中，我們把這些 atom 拆開：`init.mp4` 當中存放 `moov`，\$Number\$.m4s 裡頭則存放 `mdat` 資料，每個 m4s 檔案是把 `mdat` 中的影片資料根據一段定義好的時間長度切出來的小檔，這樣的小檔叫做 `fragment`。在播放的時候，播放器首先抓取 `moov`，然後一次又一次把這段 `moov` 與後面從 M4S 檔案中拿到的 `mdat` 結合起來播放。`init.mp4` 與 m4s 檔案中還有一些其他的 atoms，姑且不表。

在 `init.mp4` 中的 `moov` 資訊：



在 M4S 當中的 mdat 資訊：



我們在前面看到使用 `SegmentTemplate` 的範例，另外一種定義 m4s 檔案位置的作法，則是直接全部列出來。比方說，我們可以看一個來自 Wowza 的範例，這個 MPD 位在 [https://wowzaec2demo.streamlock.net/vod/elephantsdream\\_1100kbps-enc-wv.mp4/manifest\\_mvlist.mpd](https://wowzaec2demo.streamlock.net/vod/elephantsdream_1100kbps-enc-wv.mp4/manifest_mvlist.mpd)，就寫成：

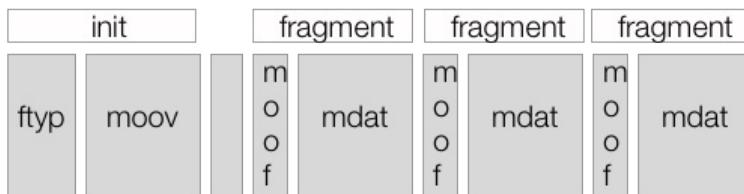
```
<SegmentList presentationTimeOffset="0" timescale="90000" duration="900000" startNumber="1">
  <Initialization sourceURL="chunk_ctvideo_ridp0va0br994339_cinit_w1146251977_mpd.m4s"/>
    <SegmentURL media="chunk_ctvideo_ridp0va0br994339_cn1_w1146251977_mpd.m4s"/>
    <SegmentURL media="chunk_ctvideo_ridp0va0br994339_cn2_w1146251977_mpd.m4s"/>
    <SegmentURL media="chunk_ctvideo_ridp0va0br994339_cn3_w1146251977_mpd.m4s"/>
  ...
</SegmentList>
```

## 連續的 fMP4

另外一種常見的作法，則是使用 fMP4 格式的檔案提供影音資料。

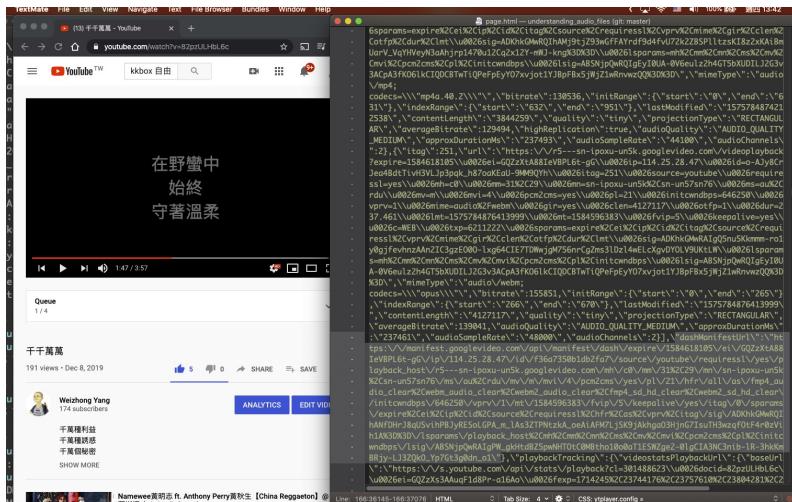
因為我們已經看過前面用 `init.mp4` 加上一堆 M4S 檔案的例子，我們於是比較容易理解 fMP4 格式：fMP4 就是把這些檔案合併成一個大檔案。然後，MPD 當中會標明每個對應到一個 M4S 格式的 byte range，播放器在載入 MPD 之後，先抓取 fMP4 的開頭部分（對應到我們的 `init.mp4`），然後，根據 MPD 中定義的 byte range，透過帶有 Range HTTP header 的連線，只抓取 fMP4 檔案中想要的部份。

這個檔案的資料大概像這樣：



我們可以來看看 YouTube 使用的 DASH 檔案。比方說，我們在 YouTube 上，打開一個我們自己上傳的頁面（在這邊用的是我在 2019 年寫的一首歌 [《千千萬萬》](#)），然後查看頁面的 HTML 原始檔，就可以找到 `dashManifestUrl` 這

段：



因此我們可以知道，這部影片的 MPD 位在 [https://r5---sn-ipoxu-un5k.googlevideo.com/videoplayback/expire/1584618105/...](https://r5---sn-ipoxu-un5k.googlevideo.com/videoplayback/expire/1584618105/)。我們可以打開來看看（不過，這個 URL 有效期，你現在點下去應該沒用）：

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <MPD xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="urn:mpeg:DASH:schema:MPD:2011" xmlns:y="http://youtube.com/yt/2012/10/10"
   xsi:schemaLocation="urn:mpeg:DASH:schema:MPD:2011 DASH-MPD.xsd" minBufferTime="PT1.500S"
   profiles="urn:mpeg:dash:profile:isoff-main:2011" type="static"
   mediaPresentationDuration="PT237.586S"><Period id="0" mimeType="audio/mp4"
   subsegmentAlignment="true"><Role schemeIdUri="urn:mpeg:DASH:role:2011"
   value="main"/><SegmentList startNumber="0" timescale="1000"><SegmentTimeline><S
   d="9985"/><S d="9984"/><S d="9985"/><S d="9984"/><S d="9985"/><S d="9984"/><S
   d="9985"/><S d="9984"/><S d="9985"/><S d="9985"/><S d="9984"/><S d="9985"/><S
   d="9985"/><S d="9984"/><S d="9985"/><S d="9984"/><S d="9985"/><S d="9984"/><S
   d="9985"/><S d="9984"/><S d="7941"/></SegmentTimeline></SegmentList><Representation
   id="139" codecs="mp4a.40.5" audioSamplingRate="22050" startWithSAP="1"
   bandwidth="49978"><AudioChannelConfiguration
   schemeIdUri="urn:mpeg:dash:23003:audio_channel_configuration:2011"
   value="2"/><BaseURL>

```

我們於是可以在：首先，這部影片有一段資料格式是 `audio/mp4` 的 `AdaptationSet`，代表 YouTube 幫我們把影片的圖片跟音軌拆開來了，這邊有一段獨立的音軌，URL 也是位在 [https://r5---sn-ipoux-un5k.googlevideo.com/videoplayback/expire/1584618105/...](https://r5---sn-ipoux-un5k.googlevideo.com/videoplayback/expire/1584618105/) 下方：

然後是這兩段：

```

<SegmentList startNumber="0" timescale="1000"><SegmentTimeline><S
   d="9985"/><S d="9984"/><S d="9985"/><S d="9984"/><S d="9985"/><S
   d="9985"/><S d="9984"/><S d="9985"/><S d="9984"/><S d="9985"/><S
   d="9984"/><S d="9985"/><S d="9984"/><S d="9985"/><S d="9984"/><S
   d="9985"/><S d="9984"/><S d="7941"/></SegmentTimeline></SegmentList>

```

```

<SegmentList><Initialization sourceURL="range/0-640"/><SegmentURL media="range/961-62378"/>
   <SegmentURL media="range/62379-123214"/><SegmentURL media="range/123215-184180"/>
   <SegmentURL media="range/184181-245118"/><SegmentURL media=

```

```
"range/245119-305980"/><SegmentURL media="range/305981-366726"/><SegmentURL media="range/366727-427627"/><SegmentURL media="range/427628-488601"/><SegmentURL media="range/488602-549420"/><SegmentURL media="range/549421-610280"/><SegmentURL media="range/610281-671202"/><SegmentURL media="range/671203-732053"/><SegmentURL media="range/732054-792887"/><SegmentURL media="range/792888-853681"/><SegmentURL media="range/853682-914615"/><SegmentURL media="range/914616-975451"/><SegmentURL media="range/975452-1036483"/><SegmentURL media="range/1036484-1097267"/><SegmentURL media="range/1097268-1158216"/><SegmentURL media="range/1158217-1219017"/><SegmentURL media="range/1219018-1279927"/><SegmentURL media="range/1279928-1340738"/><SegmentURL media="range/1340739-1401680"/><SegmentURL media="range/1401681-1449774"/></SegmentList>
```

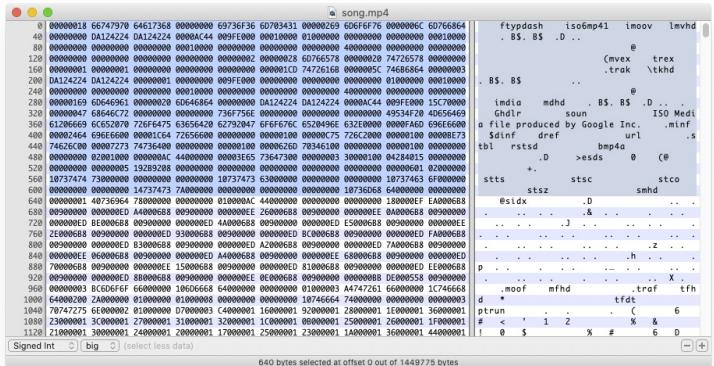
在 `SegmentTimeline` 當中，列出了這個音軌當中的時間軸，整個音軌被切成 24 的片段，每個 `s` 代表一個片段，裡頭的 `d` 則是這個片段的 `duration`，時間單位則根據 `SegmentTimeline` 當中的 `timescale`。以第一個 `s` 為例，代表的是這首歌的第一個時間區間，`d` 為 9985，`timescale` 為 1000，所以我們可以算出是 9.985 秒，整部影片大概是以十秒鐘為一個區間切成小片，最後一個 `s` 將近八秒，所以全部大概有 238 秒（3 分 58 秒）。

每個 `s` 會對應到 `SegmentList` 裡頭的 `SegmentURL`。`SegmentList` 第一段 `Initialization` 就代表我們之前講到的 `init.mp4` 的角色，裡頭包含 `ftyp` 與 `moov` 等 atoms，這段在第 0 到第 640 個 byte 的位置，後面每個 `SegmentURL`，也都代表這個區段在檔案中的哪個 byte 到哪個 byte 的區間。

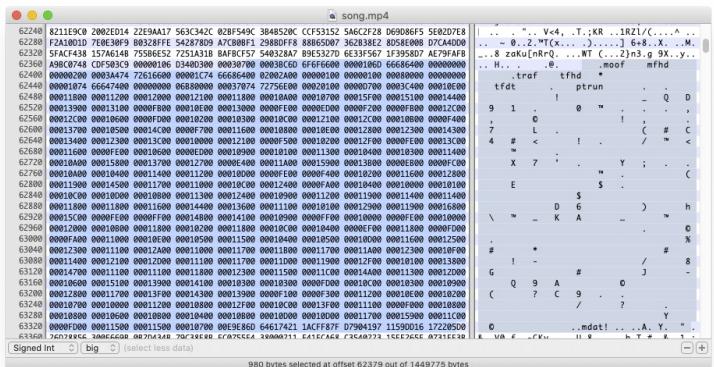
所以，假如我們在播放這個檔案的時候，一開始就想從第 15 秒開始播放，我們就可以算出，第 15 秒位在第二個 `s` 定義的區段，這時候就只要發出帶 Range header 的 HTTP 連線，先抓取 0-640 的 `ftyp` 與 `moov` 等資訊，再直接去找第二個 `s` 的範圍 62379-123214，在載入這段檔案後，因為第二個 `s` 是從第 10 秒開始，只要再 seek 五秒鐘，就可以成功達成「從第 15 秒開始播放」。

我們可以打開這個檔案看看：

從 0-640，可以看到 `ftyp` 與 `moov`。



從 62379 的一段 mdat 資料：



## 播放 MPEG-DASH 串流

播放 MPEG-DASH 串流的流程大概是：

- 抓取 MPD，並且解析裡頭的 XML
- 從 MPD 中判斷有哪些 Period 與 AdaptationSet
- 從 AdaptationSet 中挑選最適合目前頻寬的影片/音軌的組合

- 從 `AdaptationSet` 中組合出實際的 URL 播放

DASH IF 本身提供一套 JavaScript 撰寫的播放軟體 `dash.js`，Google 這幾年在 Android 平台上致力推廣的 [ExoPlayer](#) 也支援 DASH 格式，另外也有一套在網頁上播放 DASH 的 [Shaka Player](#)，Google 在 Widevine SDK 的文件中，就建議使用這兩個 Player 元件。此外，還有各種支援 MPEG DASH 的實作，在這邊不一一列舉。

在 iOS app 中播放 DASH 串流的方式有點迂迴。在 iOS 上我們看到的不是可以直接播放 DASH 的播放元件，而是把 DASH 轉成 HLS，然後用 AVPlayer 播放。

Google 有一套叫做 [Universal DASH Transmuxer](#)（簡稱 UDT）的軟體，是一套 Open Source 的函式庫，在 GitHub 上可以取得程式碼，裡頭就是將 DASH 轉換成 HLS 的相關實作，至於實際怎麼用，還是得看 Google 的 Widevine iOS SDK，但這部份沒有 Open Source。

基本原理是：

- 先在 app 中，開啟一個 local 的 HTTP server
- 這個 local HTTP server 在背後抓取一個指定的 MPD
- 解析 MPD，把裡頭的 `AdaptationSet` 抽取出來，轉換成 `.m3u8` 格式的 playlist，抓取 fMP4 的初始區段
- 如果是受保護的 DASH 串流，這時候會去抓取 license
- 當有 player 指定要播放某一段的 TS 時，local http server 在收到要求之後，再根據之前取得的 MPD，把對應的 M4S 檔案抓下來，並且轉換成 TS 格式回應
- 如此不斷輪迴

## Widevine

Widevine 保護內容的方式，是對 `mdat` atom 中的資料加密。一個受到 Widevine 保護的檔案，外觀上還是可以看出是一個 MP4 檔案，還是可以解析出 `ftyp`、`moov`、`mdat` 這些 atom，播放器需要先解析出 `mdat` 區段的資料，再透過一份 license 以及 Google 的演算法，把裡頭的資料解密回來。

一個受到 Widevine 保護串流中，會在 MPD 當中，每一段 `AdaptationSet` 中，包含一段 **PSSH** ( Protection System Specific Header ) 資訊。我們可以看一個來自 Wowza 的 MPD，就可以看到這段：

```
<AdaptationSet id="0" group="1" mimeType="video/mp4" width="512" height="288"
par="16:9" frameRate="30" segmentAlignment="true" startWithSAP="1" subsegment
Alignment="true" subsegmentStartsWithSAP="1">
  <ContentProtection schemeIdUri="urn:mpeg:dash:mp4protection:2011" value="cenc"
  cenc:default_KID="02948959-9D75-5DE2-BBF0-FDCA3FA5EAB7"/>
    <ContentProtection schemeIdUri="urn:uuid:edef8ba9-79d6-4ace-a3c8-27dcfd51d
21ed" value="Widevine">
      <cenc:pssh>AAAAW3Bzc2gAAAAA7e+LqXnWSs6jyCfc1R0h7QAAADsIARIQApS5wZ11xe
k78P3KP6XqtxoNd21kZXzpbmVfdGVzdCIQZmtqM2xqYVNkZmFsa3IzaioCU0QyAA=</cenc:pssh>
    </ContentProtection>
  <SegmentTemplate ....> ... </SegmentTemplate>
  <Representation id="p0va0br676240" codecs="avc1.42c015" sar="1:1" bandwid
th="676240" />
</AdaptationSet>
```

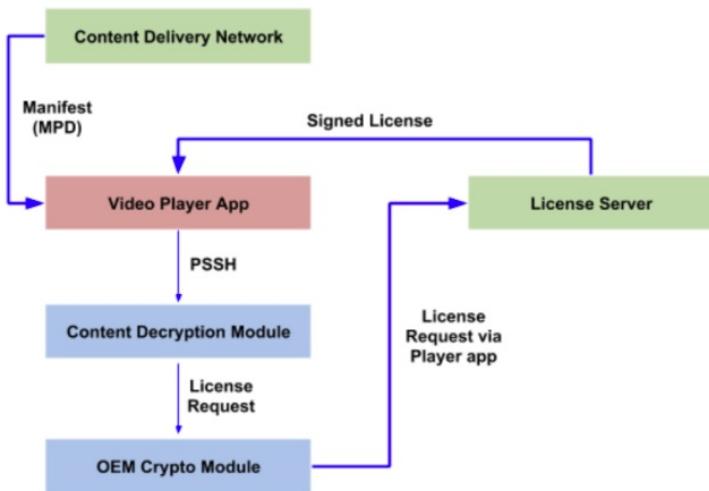
在 `<cenc:pssh>` 欄位中，就是我們所要的 PSSH 資訊，PSSH 接近我們在 HLS 裡頭用到的 Asset ID—在 HLS 裡頭，我們用 Asset ID 與蘋果簽發的 certificate 產生 SPC，在 Widevine 裡頭，我們也用 PSSH 與 **CDM** 產生出來的資訊，產生授權需求，上傳到 server 上產生、取得解密用的 license。

CDM 全名 Content Decryption Module，在行動平台上，CDM 是一套用 C++ 寫成的 library，在每種平台上 CDM 的專屬軟體叫做 CDM host，Google 只有給第三方廠商編譯好的 binary 以及 certificate。

我們首先用一些基本的裝置資訊，對 CDM host 進行初始話，然後，在抓到 PSSH 之後，我們把 PSSH 送到 CDM host 上，CDM host 再 callback 回來，回傳 CDM 組合好的憑證需求的資料，CDM 會把這段資料，送到我們的一段代理人程式—術語是發送 (send) 一段訊息 (message)—我們就可以把這段資料傳到 Google 的 License Proxy 主機上（像

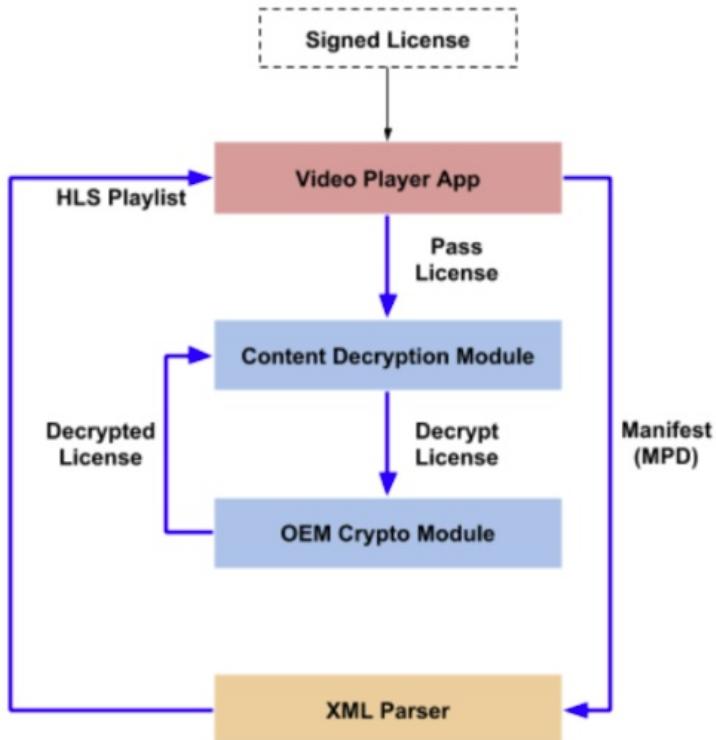
<https://license.widevine.com/getlicense/<provider>> 這樣的網址，參見 [Google 文件](#)），取得 license。

Google 提供的流程圖如下，再知道一些專有名詞的意義後，會比較好懂。這一段是傳送 PSSH 取得 license 的流程



在 Android 上，ExoPlayer 支援 Widevine DRM，詳情可以參考 ExoPlayer 的文件 [Digital rights management](#)，主要是透過 Android 本身的 [MediaDrm API](#) 管理 DRM。

在 iOS 上，拿到 license 之後，就可以為播放器準備 HLS 格式的串流：



Web 上，這整套取得 license 的流程，被包在 EME 裡頭，Google 有一篇文章專門介紹 EME：[What is EME?](#)。至於播放軟體方面，Google 也一樣推薦用 Shaka Player 播放 Winevine 保護的 Dash 串流。

## 離線下載

Widevine 也支援離線播放。基本上，要下載一個 DASH 格式的串流，就是先打開 MPD、解析裡頭的內容，然後根據 SegmentTemplate 或 SegmentList 找到對應的檔案，把所有檔案以及 MPD 全部下載回來，然後在本地端維持相同的目錄結構即可。

在播放離線檔案之前，一樣要一份 license，取得 license 的流程與線上播放的流程一模一樣，不過，需要額外跟告訴 CDM 我們要的是離線用的 license。取得 license 之後要自己在本機存放起來，並且注意這份 license 的期限是否過期—CDM 中有相關的 function，可以檢查 license 的到期時間，不過用戶似乎可以把系統時間改到過去，這點我們還沒有研究得很深。

## 在 iOS 上播放 Widevine 保護的 DASH 串流

前面提到，在 iOS 上播放 Dash 時，會在中間透過 UDT 轉換成 HLS 播放，而如果用 Widevine 的 iOS SDK 播放，轉換出的 HLS，會另外重新加密。iOS 版本的 Widevine DRM 裡頭也有一份 UDT，但這份 UDT 跟開放原始碼版本的 UDT 不同，中間加上了重新加密這段，而且 Google 也只有提供我們 binary 版本。

文件更新時間：2022/02/06 00:44 CST

# 常用相關工具

跟音檔相關的工具非常多，我們先介紹一些常用的工具

## file 指令

如果你拿到一個檔案，卻無法確定這個檔案是什麼類型，甚至沒有副檔名可以判斷，就可以先用 `file` 這個命令列指令確認看看。

`file` 這個指令的用途，就是幫助你判斷檔案類型，在 macOS 以及很多 UNIX-like 的系統上是內建的指令，另外也有專供 Windows 平台使用的版本可以 [下載](#)。`file` 可以幫你檢查出非常多種種類的檔案類型，並不局限於音檔，各種圖片、文件格式，都可以檢查出來。

在下面的範例中，我們有一個檔案叫做「song」，但是沒有副檔名，`file` 很輕易的幫我們判斷出，類型是 ISO Media, Apple iTunes ALAC/AAC-LC (.M4A) Audio —這一個 M4A 檔案，不過他覺得有可能是 AAC codec，也可能是蘋果自己的無損格式 ALAC。

```
➜ ~ file book.pdf
book.pdf: PDF document, version 1.4
➜ ~ file song
song: ISO Media, Apple iTunes ALAC/AAC-LC (.M4A) Audio
➜ ~ █
```

## ffmpeg

`ffmpeg` 大概是在所有平台上都相當通行的工具。這組工具在 Linux 上可以用 `apt-get` 等套件管理工具安裝，在 macOS 則可以用 `homebrew` 安裝，至於 Windows 版本，則可以去官網上[下載](#)。

整套 `ffmpeg` 中，有三套工具：

- `ffmpeg`：負責檔案格式的轉換
- `ffplay`：命令列播放工具
- `ffprobe`：檔案格式解析工具

`ffprobe` 可以幫我們讀出歌曲的 metadata，以及像是 sample rate、bit rate 等基本資訊，如下圖：

```
libpostproc 55. 5.100 / 55. 5.100
[mp4,m4a,3gp,3g2,mj2 @ 0x7f8899805400] Unknown cover type: 0x0.
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'aac.m4a':
  Metadata:
    major_brand     : M4A
    minor_version   : 512
    compatible_brands: isomiso2
    creation_time   : 1970-01-01T00:00:00.000000Z
    encoder         : Lavf52.111.0
    track           : 1/16
    artist          : Keane (基音樂團)
    album           : Strangeland (夢奇地)
    date            : 2012-05-07
    title            : You Are Young
Duration: 00:03:35.51, start: 0.000000, bitrate: 327 kb/s
  Stream #0:0(und): Audio: aac (LC) (mp4a / 0x6134706D), 44100 Hz
, stereo, fltp, 319 kb/s (default)
  Metadata:
    creation_time   : 1970-01-01T00:00:00.000000Z
    handler_name     : SoundHandler
```

我們看到，`ffprobe` 幫我們解析出

## afconvert、afplay、與 afinfo

`afconvert`、`afplay` 與 `afinfo` 是個 macOS 下的命令列指令，這三個指令，其實可以與 `ffmpeg` 提供的三個工具之間對應起來：

功能	<code>ffmpeg</code> 指令	macOS 指令
轉檔	<code>ffmpeg</code>	<code>afconvert</code>
播放	<code>ffplay</code>	<code>afplay</code>
解析	<code>ffprobe</code>	<code>afinfo</code>

跟 ffmpeg 比較，這些系統工具最主要的特色是使用 Core Audio API，所以，如果你有一個用 Core Audio 開發的播放器，想要確認是否可以順利播放，或是想知道用 Core Audio API 可以解析出哪資訊，就可以善用這些指令。相對地，如果 Core Audio 並不支援這種檔案格式（像我們在講 HLS 提到的 ts、還有前章講到的 DASH...），那麼就派不上用場了。

`afplay` 的用途是直接在命令列下播放檔案，相較於有 GUI 的播放軟體，這個指令可以直接用參數，指定要播放多久以及播放速率，比方說，下了 `afplay -t 5 -r 1.5 song.mp4` 指令，就可以用 1.5 倍速率，只播放這個檔案五秒鐘。

`afinfo` 則可以查看檔案的相關資訊。以一個 aac 檔案來說，我們可能會看到這樣的資訊：

```
→ ~ afinfo song.mp4
File:          song.mp4
File type ID: m4af
Num Tracks:   1
-----
Data format:   2 ch, 44100 Hz, 'aac' (0x00000000) 0 bits/channel, 0 bytes/packet, 1024 frames/packet, 0 bytes/frame
               no channel layout.
estimated duration: 215.456508 sec
audio bytes: 8612768
audio packets: 9281
bit rate: 319725 bits per second
packet size upper bound: 1405
maximum packet size: 1405
audio data file offset: 40
not optimized
audio 9501632 valid frames + 2112 priming + 0 remainder = 9503744
format list:
[ 0] format:   2 ch, 44100 Hz, 'aac' (0x00000000) 0 bits/channel, 0 bytes/packet, 1024 frames/packet, 0 bytes/frame
Channel layout: Stereo (L R)
-----
→ ~ |
```

從這裡我們就可以看到，`afinfo` 幫我們找出 Sample Rate 是 44100，有兩個聲道、每個 packet 有 1024 個 frame 等資訊，每個 packet 有多少 bytes 為 0，代表 bit rate 會變動，但從 bit rate 算出來是 319725，那差不多是個 320k 的音檔。`afinfo` 幫我們找到 9281 個 packet，算一下  $9281 * 1024 / 44100$ ，大概 215.504399，不過 `afinfo` 算出來是 215.456508，似乎有點誤差，主要原因可能是扣去了前方 2112 個填零的 frame (priming frame)。

看到「not optimized」則要注意，這種檔案的 `moov` atom 是放在 `mdat` 後方，那麼，iOS 上的 Core Audio 的 parser，就可能無法解析這種檔案，通常我們用 `ffmpeg` 轉出的檔案 MP4 或 ALAC 檔案便會如此，這時候就可以試試看使用 `afconvert` 轉檔。請參考 [AAC 與 MP4 格式](#) 這一章中的說明。

## mdls 與 mdfind

這也是 macOS 上的工具，而且嚴格來說，這是 macOS 系統功能 Spotlight 搜尋工具的相關命令列工具。`mdls` 的用途是從檔案中抽出資訊，然後讓 Spotlight 建立檢索（index），所以 `mdls` 所呈現出來的就是根據 Spotlight 欄位的資料。

用 `mdls` 呈現出來的資料如下：

```
zonble@zonbook: ~/Music/iTunes/Music/Music/交工樂隊/菊花夜行軍
brew (curl)          36% 261   ..源/菊花夜行軍 (zsh)      62  +
↳ 菊花夜行軍 mdls 01\ 縣道 184.mp3
kMDItemAlbum           = "菊花夜行軍"
kMDItemAlternateNames = (
    "01 \U7e23\U9053184.mp3"
)
kMDItemAudioBitRate    = 128000
kMDItemAudioChannelCount = 2
kMDItemAudioEncodingApplication = "iTunes v6.0.4"
kMDItemAudioSampleRate = 44100
kMDItemAudioTrackNumber = 1
kMDItemAuthors         = (
    "\U4ea4\U5de5\U6a02\U968a"
)
kMDItemContentCreationDate = 2016-07-24 04:17:28 +0000
kMDItemContentCreationDate_Ranking = 2016-07-24 00:00:00 +0000
kMDItemContentModificationDate = 2020-03-24 15:22:33 +0000
kMDItemContentModificationDate_Ranking = 2020-03-24 00:00:00 +0000
kMDItemContentType       = "public.mp3"
kMDItemContentTypeTree   = (
    "public.mp3",
    "public.audio",
    "public.audiovisual-content",
    "public.data",
    "public.item",
    "public.content"
)
kMDItemDateAdded        = 2019-11-18 09:27:13 +0000
kMDItemDateAdded_Ranking = 2019-11-18 00:00:00 +0000
kMDItemDisplayName       = "縣道 184"
kMDItemDocumentIdentifier = 0
```

然後，因為我們知道有哪些欄位可用，所以有時候我們可以透過 `mdfind` 做一些奇特的搜尋。比方說，我們突然想要找出電腦裡頭所有 bit rate 都是 128k 的音檔，就可以用以下指令：

```
mdfind "kMDItemTotalBitRate == 128000"
```

結果如下：

```
brew (curl)          mdfind (curl)          mdfind (mdfind)          Terminal
▶ 菊花夜行軍 mdfind "kMDItemTotalBitRate == 128000" | more
/Users/zonble/Music/iTunes/iTunes Music/Music/The Smiths/Louder Than
    Bombs/08 Girl Afraid.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/交工樂隊/菊花夜行軍/01
    縣道184.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/The Album Leaf/In a Sa
    fe Place/08 Streamside.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/The Album Leaf/In a Sa
    fe Place/06 Over the Pond.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/The Album Leaf/In a Sa
    fe Place/05 The Outer Banks.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/The Album Leaf/In a Sa
    fe Place/04 Twenty Two Fourteen.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/The Album Leaf/In a Sa
    fe Place/03 On Your Way.mp3
/Users/zonble/Desktop/orz.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/Grateful Dead/The Best
    Of Skeletons From The Closet/09 Turn on Your Love Light.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/Grateful Dead/The Best
    Of Skeletons From The Closet/06 Uncle John's Band.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/Grateful Dead/The Best
    Of Skeletons From The Closet/04 Sugar Magnolia.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/Grateful Dead/The Best
    Of Skeletons From The Closet/02 Truckin'.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/Compilations/0 Brother
    , Where Art Thou/_06 Hard Time Killing Floor Blues.mp3
/Users/zonble/Work/pyKKBOX/Tests/TestData/test.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/The Libertines/The Lib
    ertines/05 Music When The Lights Go Out.mp3
/Users/zonble/Music/iTunes/iTunes Music/Music/Firehose/Live Totem P
```

這邊也列出一些可以試試看的搜尋條件：

- 找出電腦上所有的 FLAC 檔案：`mdfind "kMDItemContentTypeTree=org.xiph.flac"`
- 找出電腦上所有的 MP4 檔案：`mdfind "kMDItemContentTypeTree=public.mpeg-4-audio"`
- 找出電腦上所有的搖滾樂風的檔案：`mdfind "kMDItemMusicalGenre=Rock"`
- 搜尋長度大於五分鐘的檔案：`mdfind "kMDItemDurationSeconds>300"`

文件更新時間：2022/02/06 00:44 CST



## 附註

- [MP3 Parser 範例](#)
- [AAC-ADTS Parser 範例](#)
- [ID3 Parser 範例](#)
- [FLAC Stream Info Encoder 範例](#)
- [文件版本](#)

文件更新時間：2022/02/06 00:44 CST

# MP3 Parser 範例

以下是使用 Python 2 撰寫的 MP3 以及 ID3 Parser 範例。作者為 zonble。

```
#!/usr/bin/env python
# encoding: utf-8

import os
import sys
import struct

class MP3Parser:
    """
    Parse mp3 file to check if there is invalid frame.
    """

    class _Header:
        """
        Represent the ID3 header in a tag.
        """

        def __init__(self):
            self.majorVersion = 0
            self.revision = 0
            self.flags = 0
            self.size = 0
            self.bUnsynchronized = False
            self.bExperimental = False
            self.bFooter = False

        def __str__(self):
            return str(self.__dict__)

    @classmethod
    def _getSyncSafeInt(self, bytes):
        """
        Get integer from 4 bytes
        """

        assert len(bytes) == 4
        if type(bytes) == type(''):
            bytes = [ord(c) for c in bytes]
        return (bytes[0] << 21) + (bytes[1] << 14) + (bytes[2] << 7) + bytes[3]

    @classmethod
    def isValidFrame(self, inInputFilePath):
        """
        Check if all frames are valid.

        If we encounter frame that is not MPEG version 1, layer 3,
        sample rate 44.1 KHz, we return False.

        Because MP3 File should be frame after frame, if we skip any
        byte between frames, we return False.

        Otherwise, this function will return True.

        :param inInputFilePath: the mp3 file path to check.
        :type inInputFilePath: str
        :returns: whether all frames are valid.
        """

        # Implementation details omitted for brevity.
```

```

:rtype: bool
...
content = self.loadFile(inInputFilePath)
return self.isAllFramesInDataValid(content)

@classmethod
def isAllFramesInDataValid(self, content, offset = 0, shouldCheckID3Tag =
True):
    ...
    Check if all frames are valid in bytes.

:param offset: offset into content.
:type offset: integer
:type shouldCheckID3Tag: whether we should check for id3 tag
    or not.
:type shouldCheckID3Tag: bool

...
if offset > len(content):
    return False
MP3BitrateLookup = [0, 32000, 40000, 48000, 56000, 64000, 80000, 96000,
, 112000, 128000, 160000, 192000, 224000, 256000, 320000, 0]

# The first 512 bytes are Internal Header, so we can skip them.
# Based on mac's code, it seems that we always will get ID3
# tag, so we can also just scan for ID3 tag.
i = offset
foundFirstFrame = False

# skip id3 tag
while i + 10 < len(content) and shouldCheckID3Tag:
    header = content[i:i+10]
    hstuff = struct.unpack("!3BBBBBB", str(header))
    if hstuff[0] == "ID3":
        header = self._Header()
        header.majorVersion = hstuff[1]
        header.revision = hstuff[2]
        header.flags = hstuff[3]
        header.size = self._getSyncSafeInt(hstuff[4:8])
        hasFooter = not not (header.flags & 0x40)
        headerBodyLength = header.size
        headerLength = headerBodyLength + (20 if hasFooter else 10)
        i += headerLength
        break
    else:
        i += 1

while i + 2 < len(content):
    frameSync = (content[i] << 8) | (content[i + 1] & (0x80 | 0x40 |
0x20))
    if frameSync != 0xffe0:
        if foundFirstFrame:
            # After founding first frame, we shouldn't skip
            # any byte, so if we execute to here, there is an
            # error.
            # print 'skipping byte at:' + repr(i)
            pass
        i += 1
        continue

    # frame start
    if not foundFirstFrame:
        foundFirstFrame = True
    # print i
    # AAAAAAAA AAABBCCD EEEEFFGH IIJJJKLM

```

```

audioVersion = (content[i + 1] >> 3) & 0x03;
layer = (content[i + 1] >> 1) & 0x03
hasCRC = not(content[i + 1] & 0x01)
bitrateIndex = content[i + 2] >> 4;
sampleRateIndex = content[i + 2] >> 2 & 0x03;
# print 'audioVersion:%d, layer:%d, sampleRateIndex:%d' % (audioVersion, layer, sampleRateIndex)
if not(audioVersion == 0x03 and
       layer == 0x01 and
       sampleRateIndex == 0x00):
    # we only support MPEG version 1, layer 3, sample rate
    # 44.1 KHz--and we ignore the error altogether
    print "Unsupported MPEG audio version."
    return False

bitrate = MP3bitrateLookup[bitrateIndex];
hasPadding = not(not((content[i + 2] >> 1) & 0x01))
# print 'hasCRC:%d, hasPadding:%d' % (hasCRC, hasPadding)

frameLength = 144 * bitrate / 44100 + \
              (1 if haspadding else 0) + \
              (2 if hasCRC else 0)
i += frameLength
if not foundFirstFrame:
    return False
return True

@classmethod
def loadFile(self, inInputFilePath):
    """
    Load file into bytearray

    :param inInputFilePath: path of the input file.
    :type inInputFilePath: str
    """
    inputFile = open(inInputFilePath, 'rb')
    data = bytearray(inputFile.read())
    inputFile.close()
    return data

def printUsage():
    print '''Usage: python %s [MP3 file]
Example: python %s 1311434.mp3
It will parse all frames in mp3 file and return parse result as bool.''' % (sys.argv[0], sys.argv[0])

    if __name__ == '__main__':
        if len(sys.argv) != 2:
            printUsage()
            exit()
        filename = str(sys.argv[1])
        if not(os.path.exists(filename)):
            print 'File not found: %s' % (filename)
            printUsage()
            exit()
        result = MP3Parser.isAllFramesValid(filename)
        print result

```

文件更新時間：2022/02/06 00:44 CST



# AAC-ADTS Parser 範例

以下是使用 Python 2 撰寫的 ID3 Parser 範例。作者為 Oliver Huang。

```
#!/usr/bin/env python
# encoding: utf-8

import os
import sys
import time

class ADTSParser:
    """
    Parse AAC-ADTS file to find first frame offset and check if there is invalid frame.
    """

    ADTS_MINIMUM_HEADER_SIZE = 7

    class ADTSHeader:
        """
        # AAAAAAAA AAAABCCD EEEEEFGH HHIJKLMM MMMMMMMM MMMO0000 000000PP (QQQQQQQQ)
        # AAAAAAAA AAAA: syncword
        # B: MPEG version
        # CC: Layer
        # FFFF: Sample rate
        # HHH: Channel configuration
        # MM MMMMMMMM MM: Frame length
        # SampleRateTable = (96000, 88200, 64000, 48000, 44100, 32000, 24000, 2050,
        # 16000, 12000, 11025, 8000, 7350)
        """

        def __init__(self, data):
            assert type(data) is bytearray
            self.data = data

        def isValid(self):
            if len(self.data) < ADTSParser.ADTS_MINIMUM_HEADER_SIZE:
                return False
            if self.syncword() != 0xffff:
                return False
            if self.MPEGVersion() != 4:
                return False
            if self.layer() != 0:
                return False
            if self.sampleRate() != 44100:
                return False
            if self.channelConfig() != 2:
                return False
            return True

        def syncword(self):
            return (self.data[0] << 8) | (self.data[1] & 0xf0)

        def MPEGVersion(self):
            return 4 if (self.data[1] & 0x08) == 0 else 2

        def layer(self):
            return (self.data[1] & 0x06) >> 1

        def sampleRate(self):
            index = (self.data[2] & 0x3c) >> 2
```

```

        if index >= len(self.SampleRateTable):
            return 0
        return self.SampleRateTable[index]

    def channelConfig(self):
        return ((self.data[2] & 0x01) << 2) | ((self.data[3] & 0xc0) >> 6
)

    def frameLength(self):
        if not self.isValid():
            return 0
        return ((self.data[3] & 0x03) << 11) | (self.data[4] << 3) | ((self.data[5] & 0xe0) >> 5)

    @classmethod
    def parseAACfile(self, inInputFilePath):
        """
        Check if all frames are valid and get offset of first frame.

        A valid frame will be MPEG-4, 2-channels and sample rate 44.1 KHz.

        This function will find first two valid frames without bytes between
        it.

        If we encounter frame that is not valid after first frame, we return
        False.

        Otherwise, this function will return True.

        :param inInputFilePath: the aac file path to check.
        :type inInputFilePath: str
        :returns: A tuple (first ADTS frame offset, whether all frames after
        offset are valid)
        :rtype: 2-tuple

        """
        inputFile = open(inInputFilePath, 'rb')
        content = bytearray(inputFile.read())
        inputFile.close()

        return self.isAllFramesInDataValid(content)

    @classmethod
    def isAllFramesInDataValid(self, content):
        """
        Check if all frames are valid in bytes.

        :param content: audio content to check.
        :type content: bytearray
        :returns: A tuple (first ADTS frame offset, whether all frames after
        offset are valid)
        :rtype: 2-tuple

        """
        offset = 0
        firstOffset = -1
        length = len(content)
        tStart = time.time()
        tLast = tStart

        while offset + self.ADTS_MINIMUM_HEADER_SIZE < length:
            tCurrent = time.time()
            if tCurrent - tLast > 1:
                print 'Total file length: %d, current offset: %d, time elapse
d: %.3f' % (length, offset, tCurrent - tStart)
                tLast = tCurrent

```

```
        frame = self.ADTSHeader(content[offset:offset + self.ADTS_MINIMUM
_HEADERSIZE])
        if firstOffset == -1:
            if frame.isValid() and frame.frameLength() > 0:
                secondOffset = offset + frame.frameLength()
                if secondOffset + self.ADTS_MINIMUM_HEADER_SIZE > length:
                    return (-1, False)
                second = self.ADTSHeader(content[secondOffset:secondOffset
+ self.ADTS_MINIMUM_HEADER_SIZE])
                if second.isValid() and second.frameLength() > 0:
                    firstOffset = offset
                    offset = secondOffset + second.frameLength()
                    continue
                offset += 1
            elif not frame.isValid():
                return (-1, False)
            else:
                offset += frame.frameLength()

        return (firstOffset, firstOffset >= 0)

def printUsage():
    print '''Usage: python %s [AAC file]
Example: python %s 24023614.aac
It will find first ADTS frame offset and parse all frames in aac file and return parse result as bool.''' % (sys.argv[0], sys.argv[0])

if __name__ == '__main__':
    if len(sys.argv) != 2:
        printUsage()
        exit()
    filename = sys.argv[1]
    if not os.path.exists(filename):
        print 'File not found: %s' % (filename)
        printUsage()
        exit()
    offset, isValid = ADTSParser.parseAACFile(filename)
    print isValid
    if isValid:
        print "First frame offset: %d" % offset
```

文件更新時間：2022/02/06 00:44 CST

# ID3 Parser 範例

以下是使用 Python 2 撰寫的 ID3 Parser 範例。作者為 zonble。

```
#!/usr/bin/env python
# encoding: utf-8

"""
The :mod:`KKID3TagReader` module helps to read ID3 tags from an MP3 audio
file.
"""

def _parseID3TagsFromFileStream(f):
    """
    Parses ID3 tags from a file stream

    :param f: path of the file.
    :type f: file
    :returns: ID3 tags
    :rtype: list

    .. note:: Not tested on IronPython.
    """
    HEADER_LENGTH = 10
    HEADER_BODY_LENGTH_INFO_OFFSET = 6
    HEADER_BODY_LENGTH_INFO_LENGTH = 4
    FRAME_HEADER_LENGTH = 10
    FRAME_ID_LENGTH = 4
    FRAME_BODY_LENGTH_INFO_LENGTH = 4

    def _readUInt28(bytes):
        MASK = lambda bits : ((1 << (bits)) - 1)
        BITSUSED = 7
        val = 0
        for byte in bytes: val = (val << BITSUSED) | (byte & MASK(BITSUSED))
        return min(val, MASK(BITSUSED * 4))

    bytes = bytearray(f.read())
    ID3TagHeader = bytes[0:HEADER_LENGTH]
    if not str(ID3TagHeader).startswith('ID3'): return {}

    tags = {}
    ID3TagBodyLength = _readUInt28(list(ID3TagHeader)[HEADER_BODY_LENGTH_INFO_OFFSET:HEADER_BODY_LENGTH_INFO_OFFSET+HEADER_BODY_LENGTH_INFO_LENGTH])
    readHead = HEADER_LENGTH
    while readHead < ID3TagBodyLength:
        frameHeader = bytes[readHead:readHead + FRAME_HEADER_LENGTH]
        frameID = str(frameHeader[0:FRAME_ID_LENGTH])
        frameBodyLength = _readUInt28(frameHeader[FRAME_ID_LENGTH:FRAME_ID_LENGTH + FRAME_BODY_LENGTH_INFO_LENGTH])

        frameBody = bytes[readHead + FRAME_HEADER_LENGTH : readHead + FRAME_HEADER_LENGTH + frameBodyLength]
        textEncodingType = int(frameBody[0])
        frameContent = frameBody[1:]
        frameContent = [
            lambda x : str(x),
            lambda x : x.decode('utf-16'),
            lambda x : x.decode('utf-16-be'),
```

```
lambda x : x.decode('utf-8')
][textEncodingType](frameContent) if textEncodingType <= 3 \
else str(frameContent)
if str(frameID).startswith('NORV') or str(frameID).startswith('T'):
    tags[unicode(frameID)] = frameContent.strip()
readHead += FRAME_HEADER_LENGTH + frameBodyLength
return tags

def parseID3TagsFromFileStream(f):
try:
    return _parseID3TagsFromFileStream(f)
except Exception, e:
    raise(e)

def parseID3TagsFromfilepath(path):
    ...
    Parses ID3 tags from a specific MP3 audio file.

:param path: path of the file.
:type path: str
:returns: ID3 tags
:rtype: list
...
return parseID3TagsFromFileStream(open(path, 'rb'))
```

文件更新時間：2022/02/06 00:44 CST

# FLAC Stream Info Encoder 範例

以下是使用 Python 2 撰寫的 FLAC Stream Info Encoder 範例。作者為 zonble。

```
#!/usr/bin/env python
# encoding: utf-8

class BitWriter:
    def __init__(self):
        super().__init__()
        self.out_data = bytearray()
        self._out_byte = 0
        self._out_count = 0

    def write_bit(self, bit):
        self._out_byte = (self._out_byte << 1) | (1 if bit == True else 0)
        self._out_count += 1
        if self._out_count == 8:
            self.out_data.append(self._out_byte)
            self._out_byte = 0
            self._out_count = 0

    def flush(self):
        if self._out_count == 0:
            return
        if self._out_count < 8:
            self._out_byte = self._out_byte << (8 - self._out_count)
        self.out_data.append(self._out_byte)
        self._out_byte = 0
        self._out_count = 0

    def write_unsigned_int(self, number, bit_count):
        for i in range(bit_count -1, -1, -1):
            bit = (number & (1<<i)) != 0
            self.write_bit(bit)

class StreamInfo:
    MIN_BLOCK_SIZE_LEN = 16
    MAX_BLOCK_SIZE_LEN = 16
    MIN_FRAME_SIZE_LEN = 24
    MAX_FRAME_SIZE_LEN = 24
    SAMPLE_RATE_LEN = 20
    CHANNELS_LEN = 3
    BITS_PER_SAMPLE_LEN = 5
    TOTAL_SAMPLES_LEN = 36
    MD5SUM_LEN = 128

    def __init__(self, **kwargs):
        super().__init__()
        self.is_last_metadata_block = kwargs.get('is_last_metadata_block', True)
        self.min_block_size = kwargs.get('min_block_size', 4608)
        self.max_block_size = kwargs.get('max_block_size', 4608)
        self.min_frame_size = kwargs.get('min_frame_size', 14)
        self.max_frame_size = kwargs.get('max_frame_size', 15521)
        self.sample_rate = kwargs.get('sample_rate', 48000)
        self.n_channels = kwargs.get('n_channels', 2)
        self.bits_per_channel = kwargs.get('bits_per_channel', 16)
        self.n_samples = kwargs.get('n_samples', 0)
```

```

self.checksum = None

def data(self, withHeader = True):
    writer = BitWriter()
    if withHeader:
        writer.write_unsigned_int(1 if self.is_last_metadata_block else 0
, 1)
        writer.write_unsigned_int(0, 7)
        length = (StreamInfo.MIN_BLOCK_SIZE_LEN +
StreamInfo.MAX_BLOCK_SIZE_LEN +
StreamInfo.MIN_FRAME_SIZE_LEN +
StreamInfo.MAX_FRAME_SIZE_LEN +
StreamInfo.SAMPLE_RATE_LEN +
StreamInfo.CHANNELS_LEN +
StreamInfo.BITS_PER_SAMPLE_LEN +
StreamInfo.TOTAL_SAMPLES_LEN +
StreamInfo.MD5SUM_LEN) / 8
        writer.write_unsigned_int(int(length), 24)
    writer.write_unsigned_int(self.min_block_size, StreamInfo.MIN_BLOCK_S
IZE_LEN)
    writer.write_unsigned_int(self.max_block_size, StreamInfo.MAX_BLOCK_S
IZE_LEN)
    writer.write_unsigned_int(self.min_frame_size, StreamInfo.MIN_FRAME_S
IZE_LEN)
    writer.write_unsigned_int(self.max_frame_size, StreamInfo.MAX_FRAME_S
IZE_LEN)
    writer.write_unsigned_int(self.sample_rate, StreamInfo.SAMPLE_RATE_LE
N)
    writer.write_unsigned_int(self.n_channels -1, StreamInfo.CHANNELS_LEN
)
    writer.write_unsigned_int(self.bits_per_channel -1, StreamInfo.BITS_P
ER_SAMPLE_LEN)
    writer.write_unsigned_int(self.n_samples, StreamInfo.TOTAL_SAMPLES_LE
N)
    data = writer.out_data
    if self.checksum != None and len(self.checksum) == 16:
        data += self.checksum
    else:
        writer.write_unsigned_int(0, StreamInfo.MD5SUM_LEN)
        data = writer.out_data
    return data

```

文件更新時間：2022/02/06 00:44 CST

## 文件版本

- 2022 年二月第一次發布公開版本

文件更新時間：2022/02/06 00:44 CST