视频点播

技术全景文档

VOD Full-Stack Engineering Guide

编解码 · 封装 · 协议 · ABR · CDN · DRM · QoE

面向工程师的全栈学习指南

目录

2.4 编码的其他技巧（不用细记，了解即可）

2.5 五大主流编码一览

H.264 / AVC（2003）：全球通用的"老黄金标准"

H.265 / HEVC（2013）：压得更狠，但授权让人头大

VP9（2013）：Google 的免费替代

AV1（2018）：免版税下一代

H.266 / VVC（2020）：最新一代，还没普及

对比表

2.6 实际项目怎么选编码?

实际常见组合（Bitrate Ladder + 编码组合）

2.7 动手：用 ffmpeg 压一段视频

① 最简：H.264 默认参数

② 用 CRF 控制质量

③ 为流媒体做"关键帧对齐 + faststart"

④ H.265 压同样一段

⑤ 对比实验：量化压缩效果

2.8 Per-Title、Per-Shot 编码：Netflix 的黑科技

2.9 硬编码 vs 软编码

2.10 VMAF、PSNR、SSIM：衡量"画质"的三把尺子

📌 本章要点回顾

3.1 声音怎么变成数字

3.2 采样率（Sample Rate）

3.3 位深（Bit Depth）

3.4 声道（Channel）

第四部分 · 进阶篇：DRM/播放器/QoE

8.1 为什么视频需要"加密 + 许可"

8.2 区分三个"都叫加密"的东西

8.3 HLS AES-128（轻量加密）

8.4 "真 DRM" 三巨头

8.5 CENC：一次加密、三家通吃

两种 CENC 模式

8.6 DRM 的完整工作流

逐步解释

8.7 Widevine 的 L1 / L2 / L3

FairPlay 没有等级

PlayReady 有 SL150 / SL2000 / SL3000

8.8 HDCP：你的 HDMI 线也被检查了

8.9 License Server 在做什么

8.10 SPEKE：Packager 和 Key Server 的对话协议

8.11 离线播放（Download to Go）

8.12 短视频/短剧的轻量保护策略

常见辅助保护

8.13 选择建议

8.14 动手：用 Shaka Packager 做 DRM 打包

生成 Widevine + FairPlay（CBCS 模式）

📌 本章要点回顾

9.1 播放器到底在做什么

9.2 Web 播放器：`<video>` + MSE + EME

C

D

E

F

G

H

I

J

K

L

M

O

P

Q

R

S

T

V

W

Y

按主题分类

- 🎬 视频本质
- 🔧 编解码
- 🎵 音频
- 📦 容器与封装

- 🌐 流媒体协议
- 📊 ABR
- 🚀 CDN
- 🔐 DRM
- 🎮 播放器
- 📈 QoE
- 🔄 工作流
- ☁️ AWS

返回

第一部分 · 入门导览

> 本章 10 分钟读完。读完后你会知道：视频点播到底是做什么的、这本文档怎么读、你需要看哪几章。

先问几个你可能好奇过的问题

你每天都在用视频服务，但可能从没想过：

- 🤔 我点"播放"后，视频是怎么从服务器到手机屏幕的?
- 🤔 为什么 Netflix 2 小时电影只有 2GB，自己手机拍的 2 小时却有 20GB?
- 🤔 为什么网络一弱，视频就"自动变糊"但还能继续看?
- 🤔 为什么 iPhone 只能播放 HLS 而不是 DASH?
- 🤔 为什么从 Netflix 下载的电影不能复制到别人手机上播?
- 🤔 抖音这种短视频怎么做到"滑动就播放"几乎没有加载?

读完这份文档，你可以给朋友讲清楚上面每一个问题 ✅

什么是视频点播（VOD）

视频点播的英文是 Video on Demand，简称 **VOD**。

它的定义非常简单：

> **用户想什么时候看就什么时候看，想看哪一段就拖到哪一段——视频文件早就已经录好并存好在服务器上了。**

跟点播对应的是 **直播（Live Streaming）**：

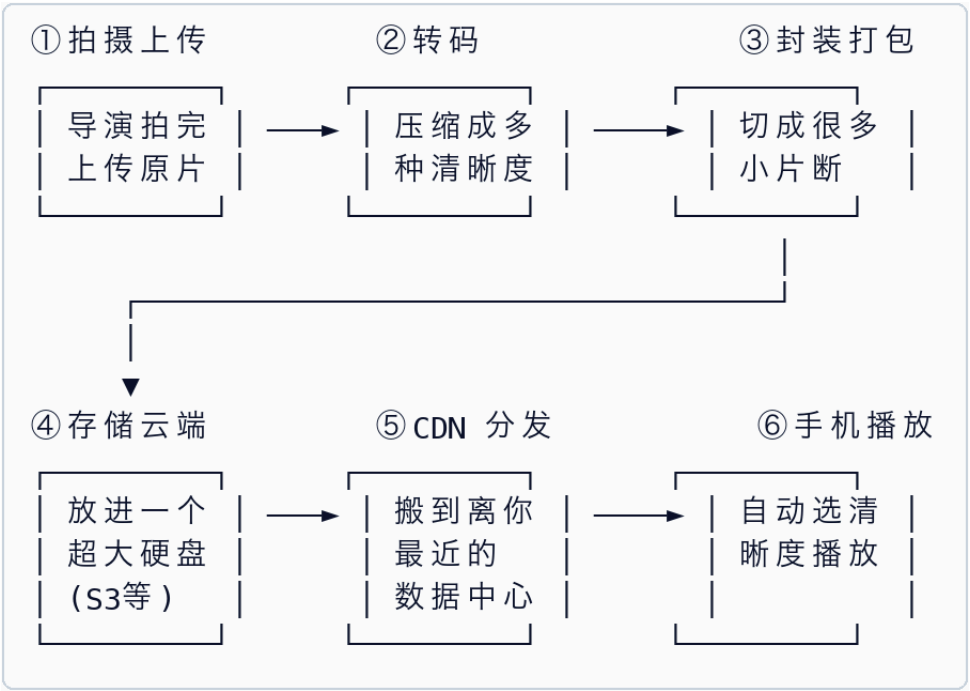
	点播（VOD）	直播（Live）
内容从哪来	**提前录好的文件**	摄像机/OBS 正在拍
能拖进度条吗	可以	不行（或只能回放已录制段）
常见例子	Netflix、YouTube、B站、抖音、在线教育录播	体育赛事直播、电商直播、游戏直播
工程难点	如何用最低成本让最多人流畅看	延迟要低，编码要实时

> 💡 **短视频（TikTok、抖音）属于 VOD 吗？**
>
> **是的**。虽然体验像 "实时"，但其实每一条短视频都是已经上传好的录播文件。只不过它们被打碎成秒级片段、配合算法推荐给你，所以感觉是"源源不断"。

本文档聚焦 **VOD**。但你在这里学到的大部分技术（编码、封装、协议、DRM）在直播中也能用。

VOD 的完整旅程：从"一个视频"到"你看到画面"

一个视频从被拍摄到你在手机上看到，要经历 6 个环节：



每个环节都有一大堆技术术语，但别慌，**它们都对应一个很容易理解的现实问题**：

环节	在解决什么问题	对应本文档章节
①拍摄上传	怎么把大文件快速、稳定送到服务器?	11–端到端工作流
②转码	怎么把 20GB 的原片压缩成 200MB 但还好看?	02–编解码入门、03–音频基础
③封装打包	怎么把视频+音频+字幕装进一个文件？怎么切成小片?	04–文件封装、05–流媒体协议
④存储	如何低成本地存海量视频?	11–端到端工作流、12–AWS 实战参考
⑤CDN 分发	怎么让全球用户都打开很快?	07–CDN分发

环节	在解决什么问题	对应本文档章节
⑥手机播放	怎么自动适应网速？怎么防止盗录？	06–自适应码率ABR、08–DRM版权保护、09–播放器

还有一个贯穿始终的主题——**怎么知道用户看得好不好？**——对应 10–QoE 数据体系。

本系列文档的阅读路径

本系列一共 13 章。我们按"**先讲清楚一个视频是什么，再讲它怎么传到你手机**"的顺序组织：

🟢 第一阶段：基础概念（必读）

> 不懂这四章，后面章节看不懂。

1. **01–视频基础** · 视频到底是什么？像素、分辨率、帧率、色彩、关键帧
2. **02–编解码入门** · 为什么要压缩？H.264、H.265、AV1 到底谁更好？
3. **03–音频基础** · 音频也要压缩，AAC 是谁？声道是什么？
4. **04–文件封装** · `.mp4` 不是一种编码！容器和编码的区别

🟡 第二阶段：核心技术（理解 VOD 怎么跑）

5. **05–流媒体协议** · HLS、DASH、CMAF 是怎么工作的？
6. **06–自适应码率ABR** · 为什么能自动切清晰度？算法如何挑？
7. **07–CDN分发** · 什么是 CDN？为什么全世界打开都很快？

🔵 第三阶段：进阶主题（工程实战关心的）

8. **08–DRM版权保护** · Widevine、FairPlay、PlayReady 与内容加密

9. 09–播放器 · 浏览器、iOS、Android 播放器内部在做什么
10. 10–QoE 数据体系 · 怎么量化用户的观看体验?

🔴 第四阶段：落地与实战

11. 11–端到端工作流 · 从上传到发布的完整生产线
12. 12–AWS实战参考 · 在 AWS 上具体搭一套会长什么样

📖 参考

- 99–术语速查表 · 所有缩略语查询 + 按主题聚类

快速通道：按身份推荐

如果你是后端/全栈工程师，刚接手 VOD 项目：

- 先读 1 → 2 → 4 → 5 → 7 建立全局观
- 再按需阅读 6 / 8 / 11 / 12

如果你是前端/移动端工程师，要做播放器：

- 先读 1 → 2 → 4 → 5 → 9
- 然后读 6（ABR 决定你调哪些参数）、10（你要上报哪些数据）

如果你是产品经理 / 运营 / 数据分析师：

- 读 0（本章）→ 1 → 2（跳过公式）→ 5（跳过代码） → 6 → 10

如果你要做短剧/短视频 APP：

- 读 1 → 2 → 5 → 6 → 7 → 9
- 重点看 ABR 与 CDN 预热，以及 09 章的零 TTFF 预载策略

如果你准备面试：

- 按顺序全读，重点 2、5、6、8、10；术语表过一遍

文档约定

为了提升可读性，每一章会用到下面几种信息块：

> 📌 **关键要点**：你一定要记住的核心结论

> 💡 **生活类比 / 小例子**：用现实世界类比或一个具体的小例子解释

> ⚠️ **常见误区**：容易踩的坑或被误传的说法

> 🔍 **深入阅读**：想进一步研究可以看的论文、标准或博客

> 🛠️ **动手试一试**：你可以立刻在电脑上跑的小实验

一句话理清三组容易混淆的概念

读完后面章节你会经常遇到这三对词，先点明：

1. **编码（Codec）≠ 容器（Container）**
   - 编码是"压缩算法"（H.264、H.265）
   - 容器是"文件格式"（MP4、MKV）
   - `.mp4` 文件里可以装 H.264 也可以装 H.265
2. **协议（Protocol）≠ 封装（Packaging）**
   - HLS、DASH 是"怎么传"的规则（协议）
   - fMP4、TS 是"传输中如何切片打包"的格式（封装）
3. **加密（Encryption）≠ DRM**

- HLS AES-128 是轻量加密，key 泄露就失效
- DRM 是一整套"发 key + 限制设备 + 限制输出"的体系

这三组概念将在后面章节详细讲。

准备好了吗?

从 **01-视频基础** 开始你的旅程 🚀

如果你已经有基础，想速查某个点，直接打开 **99-术语速查表**。

第二部分 · 基础篇：视频/音频/文件

> 本章是全系列的地基。读完你会知道：视频文件里装的究竟是什么、像素/分辨率/帧率/色彩空间这些词是什么意思、为什么"关键帧"对视频压缩这么重要。

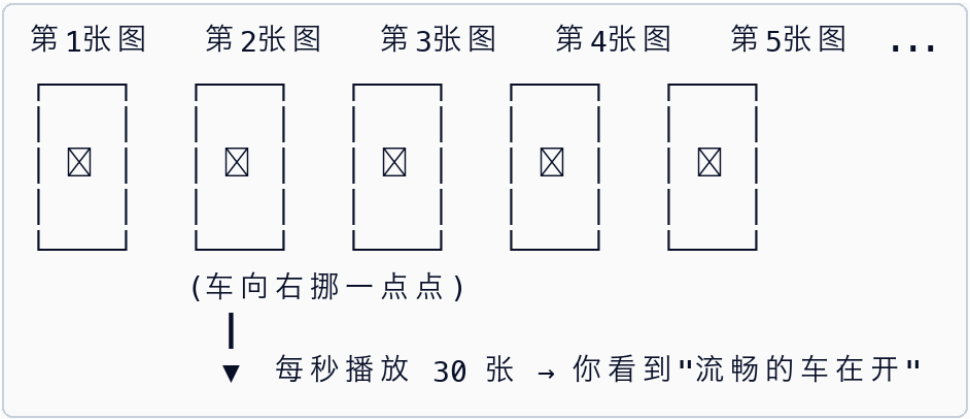
预计阅读时间：20 分钟

1.1 视频其实就是"一叠照片"

这是本章最重要的一句话，也是后面所有知识的出发点：

> 📌 **一段视频 = 一堆连续播放的图片 + 一条音轨**

没错，就这么简单。当你看一段视频，你大脑看到的其实是：

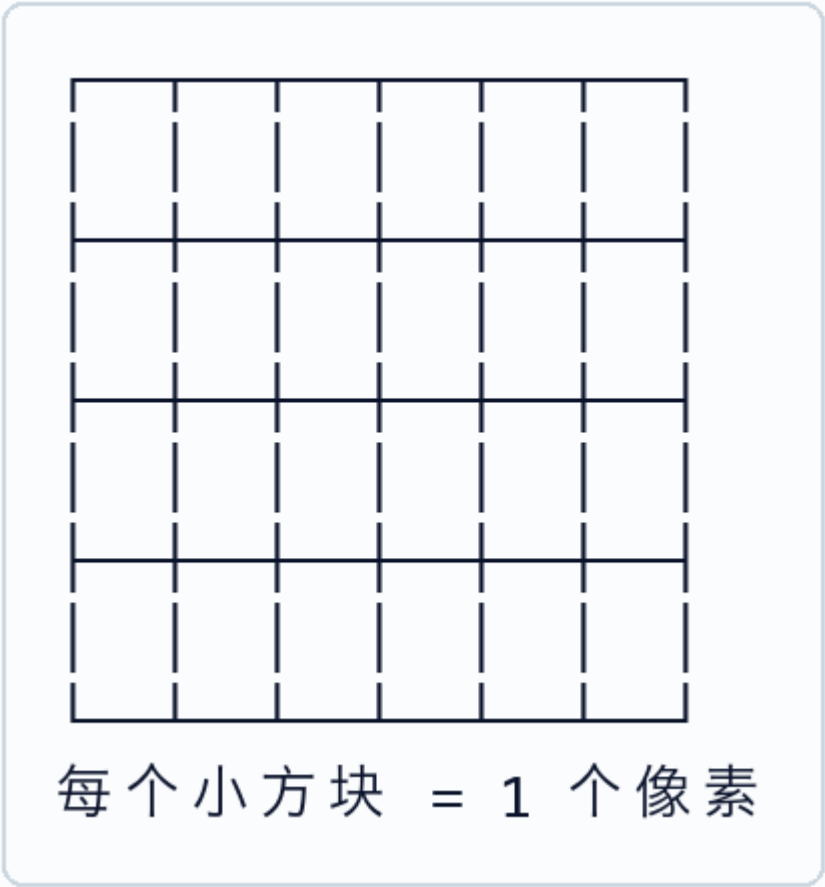


每一张图叫做一**帧（Frame）**。

> 💡 **小贴士**：这是电影发明至今 120 多年没变的原理。1895 年卢米埃尔兄弟的火车进站视频也是这么放的，只不过当年是每秒 16 张胶片。

1.2 一帧画面是什么？—— 像素（Pixel）

把一张照片放大无数倍，你会看到它由密密麻麻的小方块组成：



每个小方块记录一个颜色，这个方块叫做**像素（Pixel）**。

- 一张 1920×1080 的图片，意思是：横 1920 列 × 竖 1080 行，总共 **2,073,600 个像素**（约 200 万）。
- 每个像素存储一个颜色值，需要占几个字节（下一节说）。

分辨率（Resolution）

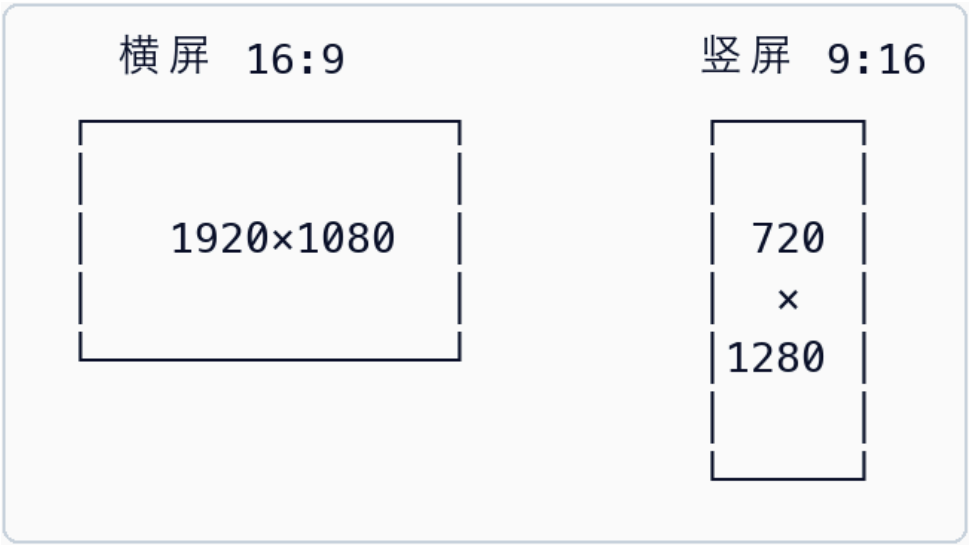
分辨率就是一张画面的像素尺寸。大家常说的"1080p、4K"就是分辨率的别名：

俗称	分辨率（横×竖）	像素总数	文件大小（相对）
240p	426 × 240	~10 万	1x（基准）
360p	640 × 360	~23 万	2.3x
480p（标清）	854 × 480	~41 万	4.1x
720p（HD）	1280 × 720	~92 万	9.2x
1080p（FHD）	1920 × 1080	~200 万	20x
1440p（2K）	2560 × 1440	~370 万	37x
2160p（4K UHD）	3840 × 2160	~830 万	83x
4320p（8K）	7680 × 4320	~3320 万	332x

> ⚠️ **"4K" 其实有两种**：
>
> - **UHD 4K**（消费级，3840×2160）← 你手机/电视看到的都是这个
> - **DCI 4K**（电影院，4096×2160）← 真正的"4K"是电影工业的

竖屏 vs 横屏

手机短视频/短剧是竖屏，比例 **9:16**（例如 720×1280）。横屏电影是 **16:9**（例如 1920×1080）。两者数字互换。



1.3 每个像素怎么存颜色？—— RGB、YUV、位深

方案一：RGB

电脑最直观的方式是记录每个像素的 **红（R）、绿（G）、蓝（B）** 三个值。三原色混合能呈现任何颜色：

- 黑色 = R:0 G:0 B:0
- 白色 = R:255 G:255 B:255
- 红色 = R:255 G:0 B:0

每个颜色分量用 **8 bit（1 字节，0–255）** 表示，所以一个 RGB 像素需要 **3 字节**。

> 🛠️ **算一算**：一张 1080p RGB 图片多大？
>
> 1920 × 1080 × 3 字节 ≈ **6.2 MB / 张**
>
> 如果每秒 30 张，一秒就要 **186 MB**！一部 2 小时电影就是 **1.3 TB**！
>
> 这还没算音频。所以——**视频必须压缩**。

方案二：YUV（视频世界的首选）

视频工业用的是另一种方式：**YUV**（有时写作 YCbCr）。

- **Y**（亮度/Luma）：这个像素有多亮，0（全黑）到 255（全白）
- **U、V**（色度/Chroma）：这个像素偏什么颜色

为什么不用 RGB？关键原因：

> 📌 **人眼对亮度敏感，对颜色不敏感**。

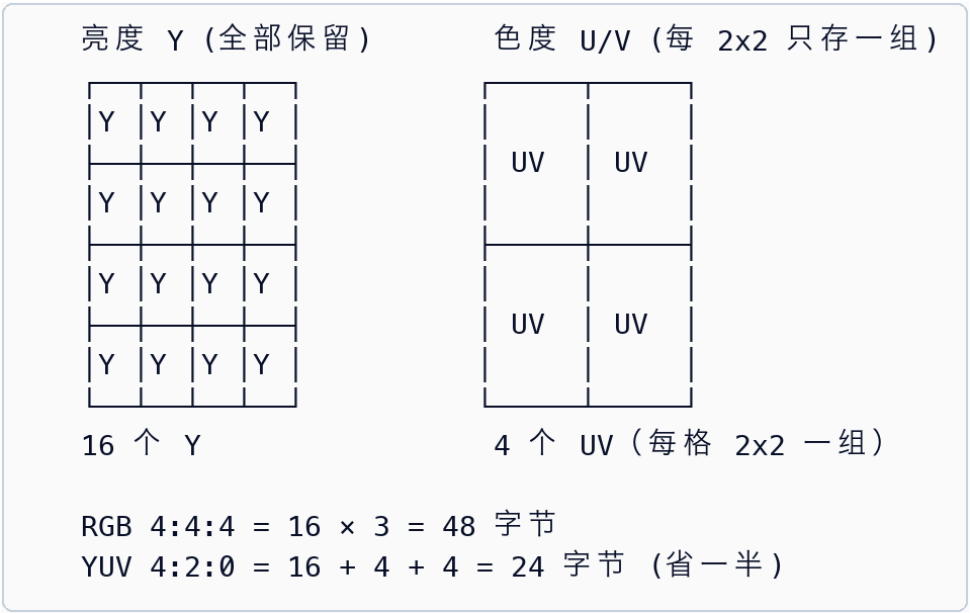
利用这个特点，YUV 可以**少记录一些颜色信息**而视觉上几乎看不出差别。

YUV 子采样（Chroma Subsampling）

最常见的三种采样方式：

方式	说明	数据量	用在哪
4:4:4	每个像素都有完整的 Y/U/V	100%	电影后期、专业制作
4:2:2	两个相邻像素共用一组 U/V	67%	广电、专业录像
4:2:0	四个相邻像素共用一组 U/V	**50%**	**几乎所有消费流媒体**

用下面这张图理解 4:2:0：



> ⚠️ **4:2:0 的代价**：锐利红色文字在纯黑背景上（或反过来）可能略有色渗。但 99% 的自然场景看不出。

位深（Bit Depth）

每个分量用几个 bit 存：

位深	每分量取值范围	一个像素可表达的颜色	用在哪
8-bit	0–255	1677 万 (256^3)	消费流媒体绝大多数
10-bit	0–1023	10.7 亿 (1024^3)	**HDR 必须**；Netflix 4K、蓝光
12-bit	0–4095	687 亿	电影母版、Dolby Vision

> 💡 **8-bit 够用吗?**
>
> 大多数场景够。但在"蓝天从深蓝渐变到浅蓝"这种大面积平滑过渡上，8-bit 经常出现肉眼可见的**色带（banding）**——一条一条的不自然边界。HDR 内容必须 10-bit 才能消除色带。

1.4 帧率（Frame Rate, fps）：每秒放多少张

fps 是 frames per second 的缩写。

- **24 fps**：电影院标准。120 年传统，观感"有电影味儿"。
- **25 fps / 50 fps**：PAL 制式（欧洲、中国电视老标准）。
- **29.97 / 30 fps**：NTSC（北美、日本电视）。手机录像默认多是 30fps。
- **60 fps**：游戏、体育直播、YouTube 高帧率。丝滑顺畅。
- **120 fps / 240 fps**：慢动作、专业拍摄。

> 💡 **为什么电影只要 24 fps 就够?**
>
> 人眼"视觉暂留"大约 1/16 秒。到了 16fps 大脑就认为是连续动作。24fps 是 1920 年代"够流畅 + 最省胶片"的平衡点。
>
> 但 24fps 看快速运动会有"拖影感"。体育、游戏必须 60fps 以上才清晰。

帧率的陷阱

- 29.97fps 不是笔误，是 NTSC 彩色电视为了向下兼容黑白信号特意避让的奇怪数字。
- **高帧率 = 文件更大**。60fps 的文件在同清晰度下大约是 30fps 的 1.7 倍。
- **播放器帧率 ≠ 显示器刷新率**，但显示器刷新率低于视频帧率时会丢帧。

1.5 码率（Bitrate）：每秒塞多少数据

码率是"这段视频每秒要消耗多少 bit"。

单位是 **bps**（bits per second），常见：

- **kbps**（千 bit/秒）：1 Mbps = 1000 kbps
- **Mbps**（兆 bit/秒）：常见单位

一段视频的大小大约等于 **码率 × 时长**：

> 1 Mbps × 60 秒 / 8 (字节转换) ≈ **7.5 MB**

典型码率参考（H.264 编码）

分辨率	推荐码率	1 分钟文件大小
240p	0.3–0.5 Mbps	~3 MB
360p	0.5–0.8 Mbps	~5 MB
480p	0.8–1.2 Mbps	~8 MB
720p	1.5–3 Mbps	~15 MB
1080p	3–6 Mbps	~30 MB
4K	15–30 Mbps	~150 MB

CBR / VBR / CRF

三种"码率控制方式"：

方式	含义	比喻
CBR（恒定码率）	每秒都用恰好 2 Mbps	点餐永远点 2 个菜
VBR（可变码率）	复杂场景多给，简单场景少给	大胃王多点，小胃口少点

方式	含义	比喻
CRF（恒定质量）	质量保持某个目标，码率自适应	不管点啥一定要吃到 8 分饱

> 📌 **VOD 点播优先用 VBR 或 CRF**：文件大小相近时画质更好；直播优先用 CBR：码率稳定便于网络传输。

1.6 关键帧、P 帧、B 帧：视频压缩的核心概念

这是本章最关键的概念。理解它，下一章讲编码就水到渠成。

为什么视频可以压得这么狠?

想象一段"人在沙发上看电视"的视频：

```
第 1 帧：人坐在沙发上，电视在放动画
第 2 帧：人坐在沙发上，电视在放动画（只是电视里画面变一点点）
第 3 帧：人眨了一下眼，电视在放动画
第 4 帧：人坐在沙发上，电视在放动画
...
```

99% 的像素在相邻帧之间是一模一样的！ 如果每一帧都完整存，就是巨大浪费。

聪明的做法是：

- 偶尔存一张"完整的快照"
- 其余时候只存"这一帧跟上一帧有什么不同"

这就是 **I 帧 / P 帧 / B 帧** 的由来。

三种帧类型

帧类型	全称	内容	文件大小	能不能独立解码
I 帧（关键帧）	Intra-coded	一张完整图片（相当于 JPEG）	**大**	✅ 能
P 帧（前向预测）	Predicted	"和前面某一帧的差异"	**小**	❌ 必须先解码前面的帧
B 帧（双向预测）	Bidirectional	"和前后帧的差异"	**最小**	❌ 必须先解码前后的帧

用图表示：

```
时间轴 →
I - P - P - P - B - P - P - B - I - P - P - P ...
▲                                 ▲
关键帧                            下一个关键帧
(每 N 帧出现一次)
```

> 💡 **类比**：
>
> - **I 帧** = 一张完整的照片
> - **P 帧** = "上张照片基础上，那只狗从左跑到右了"
> - **B 帧** = "结合前一张和后一张，中间那只狗大概在中间位置"

IDR 帧：特殊的 I 帧

IDR 帧（Instantaneous Decoder Refresh）是一个特殊的 I 帧：

> 它后面的所有帧**不允许**参考它之前的任何帧。

IDR 帧是视频的"安全启动点"。拖动进度条跳到视频中段时，播放器会从最近的 IDR 帧开始解码。

GOP（Group of Pictures）：两个 I 帧之间的一组画面

GOP 长度决定视频的"切片粒度"：

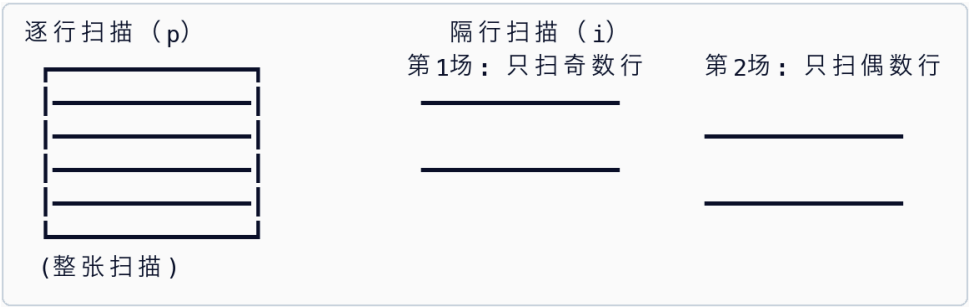
- **短 GOP**（1–2 秒）：切片细，跳转快、启动快；但文件略大（I 帧多）
- **长 GOP**（4–10 秒）：文件小，但拖动定位慢

> 📌 **短剧/短视频常用短 GOP（1–2s）**，因为用户要频繁滑动切剧集。长电影 VOD 可以用长 GOP 省带宽。

1.7 扫描方式：Progressive vs Interlaced

一个历史遗留问题，但面试可能问到。

- **逐行扫描（Progressive，写作 p）**：完整一帧就是完整一张图。`720p`、`1080p` 的 "p"。
- **隔行扫描（Interlaced，写作 i）**：一帧分奇数行和偶数行两次扫描。老电视信号标配。`1080i`。



> ⚠️ **从广电系统拿到 1080i 素材怎么办?**
>
> 必须做 **deinterlacing（去隔行）**，把两场合成一帧，否则在电脑上播放会有"梳状条纹"伪影。ffmpeg 的 `-vf yadif` 是最常用的工具。

现代网络 VOD 几乎全都用 p（逐行）。

1.8 色彩空间与 HDR：BT.709、BT.2020、HDR10

色彩空间（Color Space）

"同一个 R=200, G=50, B=50 的数值"在不同标准下显示出来的颜色是**不一样的**。这就是色彩空间的意义。

标准	用途	色域大小
sRGB	电脑、网页	基准
BT.709	HDTV、1080p 流媒体	约等于 sRGB
BT.2020	HDR、4K/8K	比 BT.709 大约 **72%**
DCI-P3	电影院、苹果生态	介于 BT.709 和 BT.2020 之间

HDR：更亮、更暗、更多颜色

传统 SDR（Standard Dynamic Range，标准动态范围）亮度上限大约 100 尼特。

HDR（High Dynamic Range） 可以做到 1000–4000 尼特峰值亮度，且配合 10-bit 色深 + BT.2020 色域，让画面：

- 夜空中的星星更亮
- 阴影里的细节保留更多
- 颜色饱和不失真

主流 HDR 格式：

格式	由谁	特点
HDR10	蓝光协会	免版税、静态元数据（每部电影一组参数）
HDR10+	三星/亚马逊	HDR10 升级，动态元数据（每场景一组参数）
Dolby Vision	Dolby	12-bit、动态元数据，质量最强；**要交版权费**
HLG（Hybrid Log-Gamma）	BBC/NHK	同时兼容 SDR/HDR 显示器；直播首选

> ⚠️ **注意**：HDR 视频发到 SDR 显示器上不会"自动变好看"。如果没做 **tone mapping（色调映射）**，HDR 视频在 SDR 屏幕上会**发灰**。

1.9 实战：用 ffprobe 查看一个视频文件的参数

> 🛠️ **动手试一试**：安装 ffmpeg 后可以运行 `ffprobe` 看任何视频的详细参数。

```
# macOS / Linux
brew install ffmpeg  # 或 apt install ffmpeg

# 查看视频
ffprobe -v error -show_streams -select_streams v:0 myvideo.mp4
```

典型输出（解读）：

```
codec_name=h264              # 编码格式（H.264） — 详见第 2 章
profile=High                 # 编码档次
width=1920
height=1080                  # ← 分辨率 1080p
r_frame_rate=30000/1001      # ← 帧率 29.97fps
pix_fmt=yuv420p              # ← 像素格式（YUV 4:2:0, 8-bit）
color_space=bt709            # ← 色彩空间（SDR）
bit_rate=4500000             # ← 码率 4.5 Mbps
```

对照本章学的，你现在应该能看懂每一行。

📌 本章要点回顾

1. 视频 = 一连串图片 + 音频；每张图片叫**帧**。
2. 每张图片由**像素**组成；**分辨率**就是像素的尺寸。
3. 视频世界用 **YUV 4:2:0** 存像素（省一半空间，人眼察觉不到）。
4. **位深** 8-bit 够用，但 HDR 需要 10-bit。
5. **帧率** 24fps（电影）/ 30fps（电视）/ 60fps（游戏体育）。
6. **码率** 就是每秒数据量；VBR/CRF 是 VOD 常用方式。
7. **I 帧 / P 帧 / B 帧** 是视频能压缩到原来几十分之一的核心技巧。
8. **GOP** 是两个关键帧之间的画面组；短剧用短 GOP（1–2s）。
9. **HDR** 是 10-bit + 更大色域 + 高亮度，跟 SDR 完全两码事。

> 本章你会理解：视频编码到底是什么、为什么能压缩几十倍、H.264 / H.265 / AV1 / VP9 到底谁更好、怎么用 ffmpeg 压一段视频。

预计阅读时间：25 分钟

2.1 先算个账：不压缩的视频有多大

回忆上一章算过的：

> 一段 1080p、30fps、YUV 4:2:0、8-bit 的**未压缩**视频：
>
> **每秒大小** = 1920 × 1080 × 1.5（YUV 4:2:0 是 RGB 的一半） × 30 ÷ 1024^2 ≈ **89 MB/秒**

所以：

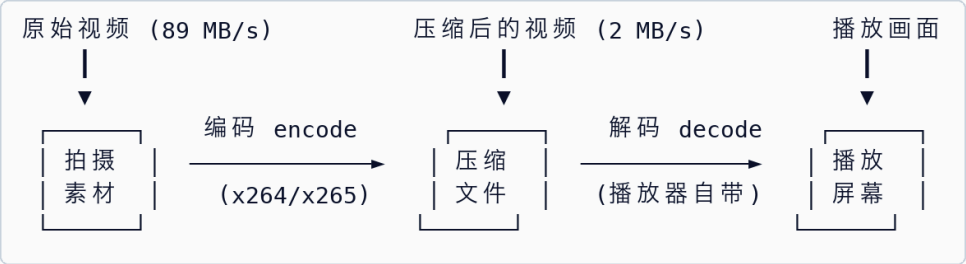
- **1 分钟视频 ≈ 5.3 GB**
- **90 分钟电影 ≈ 480 GB**
- **4K 电影（4 倍像素、30fps） ≈ 1.9 TB**

Netflix 真实的 4K 电影文件一般 5–15 GB。也就是说——

> 📌 **编码把视频压缩到了原始大小的 1% 以下。**

这不是魔法，是几十年的数学成果。我们来看它怎么做到的。

2.2 编码与解码：两个对称的过程



- **编码（Encode）**：把大文件压成小文件 → **慢，要消耗大量 CPU/GPU**
- **解码（Decode）**：把小文件还原成画面 → **快，手机芯片一般都有专门硬件**

编码器（Encoder） 和 **解码器（Decoder）** 合称 Codec（编解码器）。

> 💡 **为什么编码比解码慢那么多？**
>
> 编码要"尝试所有可能性找最佳压缩方案"，解码只要"按压缩方案的说明还原"。
>
> 这就像快递打包一件异形物品——打包的人要精心折叠省空间，开箱的人撕开胶带就行。

2.3 视频压缩的两把斧

压缩视频主要靠两种方式：

斧 1：帧内压缩（Intra-frame）

把**一张图片本身**压缩。原理和 JPEG 基本一样：

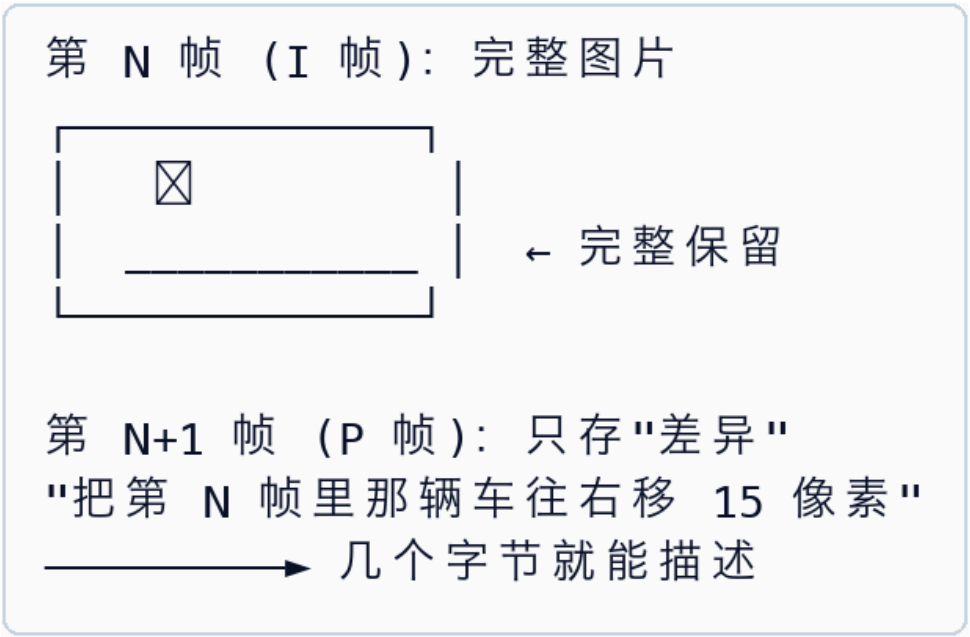
- 人眼对低频信息（大面积色块）敏感、对高频信息（细节噪点）不敏感
- 用 **DCT（离散余弦变换）** 把图片从像素域转到频率域
- 把不重要的高频系数扔掉或用少的 bit 存

> 💡 **类比**：就像你记电话号码"13912345678"时，前面"139"是运营商前缀（重要），后面数字随机（可以精确记忆）。一张图片里大面积的蓝天也类似——主色调很重要，每朵云的边缘细节可以概括。

这种压缩对每一帧独立进行，得到的帧就是 **I 帧**。

斧 2：帧间压缩（Inter-frame）

利用"相邻帧之间几乎一样"这个特点，只存差异。



具体做法叫 **运动估计（Motion Estimation）+ 运动补偿（Motion Compensation）**：

1. 把画面切成小方块（宏块 Macroblock，典型 16x16 或 8x8 像素）
2. 对每一个方块，在前一帧里搜索"最像它的"位置
3. 只记录**运动向量**（向左/右/上/下偏了多少像素）+ **残差**（细微差异）

> 💡 **类比**：摄像师拍摇头镜头，画面整体向右平移 → 编码器发现所有方块都向右移 20 像素，只要记录一次"20 像素"而不是重新存每个方块。

帧间压缩效率远远高于帧内压缩，**视频比 JPEG 图片序列小几十倍就是这个原因**。

2.4 编码的其他技巧（不用细记，了解即可）

现代编码器还会：

技巧	做什么	效果
变换（Transform）	DCT / Integer Transform 把像素变成频率系数	方便后续量化
量化（Quantization）	把频率系数除以一个整数、四舍五入	画质主要损失点、也是压缩主力
熵编码（Entropy Coding）	CABAC / CAVLC：用更少的 bit 存常见符号	无损压缩最后一步
环路滤波（In-loop Filter）	去除块效应（Blocking Artifact）	画面更平滑
SAO / ALF（H.265+）	自适应采样偏移	减少锐利边缘的伪影
Multi-reference	P/B 帧可参考多张历史帧	更准的预测 → 更小的残差

这些你在实际工程中只要通过参数开关就可以，不需要自己实现。

2.5 五大主流编码一览

从老到新：

H.264 / AVC（2003）：全球通用的"老黄金标准"

- **全名**：H.264 / MPEG-4 Part 10 / Advanced Video Coding (AVC)
- **谁定的**：ITU-T + ISO/IEC MPEG 联合
- **兼容性**：几乎**所有**能播视频的设备都支持 ✅✅✅
- **压缩效率**：基准（我们用它作为比较基线）
- **专利**：付费（MPEG LA 专利池），但已成行业默认
- **适合**：兼容性第一、算力受限的场景

> 💡 H.264 已经 20 多年了，依然是 YouTube、Facebook、Zoom 的首选兜底编码。

H.265 / HEVC（2013）：压得更狠，但授权让人头大

- **全名**：H.265 / MPEG-H Part 2 / High Efficiency Video Coding (HEVC)
- **压缩效率**：相同画质下**比 H.264 省约 37%**
  码率（高分辨率场景研究数据）
- **专利**：**混乱且昂贵**——三个专利池（MPEG LA、HEVC Advance、Velos Media）+ 很多未池化专利
- **硬解**：iPhone 7+（2016）、安卓 6+ 主流新机、大部分 4K 电视
- **适合**：苹果生态、4K 流媒体、对存储/带宽成本敏感的场景

> ⚠️ **HEVC 授权乱象**是它在 Web 端推行缓慢的根本原因。Chrome、Firefox 都曾经拒绝支持。

VP9（2013）：Google 的免费替代

- **全名**：VP9，不是字母缩写
- **谁做的**：Google（买了 On2 Technologies 后自研）
- **压缩效率**：接近 H.265

- **专利**：**免版税** ✅
- **主要用途**：YouTube、Google Meet
- **硬解**：安卓、Chromecast、部分智能电视；iOS **不原生**支持

AV1（2018）：免版税下一代

- **全名**：AOMedia Video 1
- **谁做的**：AOMedia 联盟（Google、Netflix、Meta、Amazon、Cisco、微软、Intel、苹果等巨头）
- **压缩效率**：**比 H.264 省约 53%**、比 H.265 再省约 25%（同研究数据集）
- **专利**：**免版税** ✅
- **硬解**：iPhone 15 Pro+（2023）、Pixel 6+、骁龙 8 Gen 2+、Intel Arc、NVIDIA RTX 40 系；2024 后大量普及
- **编码速度**：早期 SVT-AV1 等工程实现大幅改善；现在 CPU 编码成本约是 H.265 的 2–5 倍

> 💡 Netflix、YouTube、TikTok、Meta 都在把 AV1 作为未来主流。

H.266 / VVC（2020）：最新一代，还没普及

- **压缩效率**：比 H.264 省约 **78%**、比 AV1 再省约 25–30%
- **硬解**：2024 起旗舰手机开始带，消费级设备覆盖率仍很低
- **适合**：观望

对比表

	H.264	H.265	VP9	AV1	H.266
年份	2003	2013	2013	2018	2020
压缩效率（vs H.264）	基准	**+37%**	~+30%	**+53%**	**+78%**

	H.264	H.265	VP9	AV1	H.266
编码速度	最快 ⭐⭐⭐⭐⭐	⭐⭐⭐	⭐⭐⭐	⭐⭐	⭐
解码负担	最轻	中	中	较重	重
兼容性	⭐⭐⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐ (Web)	⭐⭐⭐ 且增长中	⭐
专利费	付费	**高且乱**	免	**免**	付费

> ⚠️ **37%、53%、78% 都不是"普适保证"！**
>
> 这些数字来自特定测试集、特定分辨率、特定质量指标（如 BD-rate with VMAF/PSNR）。不同内容类型（动画 vs 实拍 vs 游戏录屏）下差距很大。不要拿来当产品宣传的绝对值。
>
> 数据来源：Topiwala, Kulupana, Krishnan, *Performance Comparison of VVC, AV1, HEVC and AVC for High Resolutions* (ResearchGate 2024)。

2.6 实际项目怎么选编码?

一个实用的决策树：

```
你的用户主要在哪里看视频？
│
├── ① Web 浏览器 + 所有手机 + 老机顶盒
│      → 必须有 H.264 档位（兜底）
│      → 若带宽是主要成本，可追加 H.265（iOS 覆盖）+ AV1（安卓旗舰/新 Chrome）
│
├── ② 只在自家 APP（iOS + Android + 可选 TV）
│      → H.264 + H.265 主力；AV1 按设备能力灰度下发
│
├── ③ Web 为主，且在意全球带宽成本（YouTube/Netflix 场景）
│      → AV1 为主 + H.264 兜底（老设备）
│
└── ④ 4K / HDR 高价值内容
       → H.265 / AV1 + Dolby Vision
```

实际常见组合（Bitrate Ladder + 编码组合）

一份常见的短剧 APP 编码策略：

档位	分辨率	H.264 码率	H.265 码率	AV1 码率（旗舰机）
Low	360p	500 kbps	350 kbps	250 kbps
Mid	540p	900 kbps	650 kbps	480 kbps
Main	720p	1.5 Mbps	1.0 Mbps	750 kbps
High	1080p	3.5 Mbps	2.2 Mbps	1.6 Mbps

2.7 动手：用 ffmpeg 压一段视频

> 🛠️ **动手试一试**：下面每个命令你都可以复制到终端试跑。

准备一个输入文件 `input.mov`。

① 最简：H.264 默认参数

```
ffmpeg -i input.mov -c:v libx264 output.mp4
```

- `-i input.mov`：输入
- `-c:v libx264`：视频编码用 x264（H.264 的开源实现）
- `output.mp4`：输出文件

② 用 CRF 控制质量

```
ffmpeg -i input.mov \
  -c:v libx264 -preset medium -crf 23 \
  -c:a aac -b:a 128k \
  output.mp4
```

参数解释：

参数	含义
`-preset medium`	编码速度档位。可选 ultrafast/superfast/veryfast/faster/fast/**medium**/slow/slower/veryslow。**越慢 = 压得越狠（同质量下文件更小）**
`-crf 23`	**质量目标**，范围 0–51。**数字越小画质越好、文件越大**。日常 18–28 之间，23 是默认。
`-c:a aac -b:a 128k`	音频用 AAC、128 kbps

> 💡 **CRF 速记**：
>
> - CRF 18：视觉无损（普通人看不出差异）
> - CRF 23：高质量（推荐日常使用）
> - CRF 28：能接受（明显可见的压缩痕迹）

③ 为流媒体做"关键帧对齐 + faststart"

做 VOD 发布时一定要加：

```
ffmpeg -i input.mov \
  -c:v libx264 -preset slow -crf 22 \
  -profile:v high -level 4.0 \
  -g 60 -keyint_min 60 -sc_threshold 0 \
  -c:a aac -b:a 128k \
  -movflags +faststart \
  output.mp4
```

新增参数：

参数	为什么
`-g 60 -keyint_min 60`	**每 60 帧一个 I 帧**。30fps 就是 2 秒一个 GOP，方便切片。
`-sc_threshold 0`	禁用场景切换自动插 I 帧；保证所有码率档位 I 帧位置一致。
`-movflags +faststart`	把 MP4 的"目录表"（moov box）放到文件开头，**支持边下边播**。详见第 4 章
`-profile:v high -level 4.0`	兼容性设置；Level 4.0 支持到 1080p30。

④ H.265 压同样一段

```
ffmpeg -i input.mov \
  -c:v libx265 -preset medium -crf 26 \
  -tag:v hvc1 \
  -c:a aac -b:a 128k \
  -movflags +faststart \
  output_hevc.mp4
```

注意：

- H.265 的 CRF 要比 H.264 **加大约 3–5** 才得到相近画质（26 ≈ H.264 的 22）
- `-tag:v hvc1` 让 Apple 设备能正确识别。否则 iOS/Safari 可能拒绝播放。

⑤ 对比实验：量化压缩效果

假设原片 1 分钟 4K mov 大约 2 GB。你压出来：

命令	输出大小	相对
不压缩 YUV	~2 GB（源）	100%
H.264 CRF 23	~30 MB	1.5%
H.264 CRF 18	~80 MB	4%
H.265 CRF 26	~18 MB	0.9%
AV1（SVT-AV1 preset 8）	~12 MB	0.6%

压缩率高达 100–500 倍，但在手机屏幕上肉眼几乎看不出差别。

2.8 Per-Title、Per-Shot 编码：Netflix 的黑科技

默认做法是"所有视频都用同一套码率阶梯"：所有电影都出 360p@500k / 720p@1500k / 1080p@4000k。但：

- 一部**卡通片**（大色块、少细节）在 1080p@1000k 就够好看
- 一部**演唱会**（闪烁灯光、快速运动）1080p@5000k 可能还有压缩痕迹

让每一部电影有自己的最优码率阶梯，就是 **Per-Title Encoding**（Netflix 2015 年提出）：

```
传统:       所有电影都用相同阶梯
        360p@500k / 720p@1500k / 1080p@4000k

Per-Title: 针对每部电影的复杂度测算最优阶梯
        卡通:      360p@300k / 720p@800k / 1080p@1800k    (省钱)
        演唱会:    360p@700k / 720p@2200k / 1080p@5500k   (加钱)
```

Per-Shot / Dynamic Optimizer（Netflix 2018）更狠：

> 把一部电影按"镜头"切开（场景检测），**每个镜头都用自己最优的码率和编码参数**。

Netflix 用 **VMAF**（他们自研的感知质量分数）作为优化目标，声称做到同质量再省 17% 码率。

> 💡 **这需要你做吗?**
>
> 中小平台用云厂商的"智能转码模板"（阿里云 "窄带高清"、腾讯云 "极速高清"、AWS MediaConvert "QVBR"）就能享受大部分收益，不用自研。Netflix 那套是针对数亿订阅用户规模才值得。

2.9 硬编码 vs 软编码

方式	说明	速度	画质	适合场景
软编码（CPU）	libx264/libx265/SVT-AV1	慢	**最好**	VOD 离线转码、需要极致质量
硬编码（GPU/ASIC）	NVIDIA NVENC、Intel QSV、Apple VideoToolbox	**快 5-50 倍**	略差	直播、实时转码、低延迟

📌 **VOD 优先用软编码**：你不用赶时间，多花几小时压一次能永久省带宽。

直播必须硬编码：不能让 1 秒的直播花 10 秒去编码。

2.10 VMAF、PSNR、SSIM：衡量"画质"的三把尺子

怎么知道压缩后的视频"好不好看"？不能全靠人眼。

指标	全称	原理	范围	和人眼相关性
PSNR	Peak Signal-to-Noise Ratio	像素级差异	0–∞ dB（越大越好）	弱
SSIM	Structural Similarity	考虑亮度/对比度/结构	0–1（越大越好）	中
VMAF	Video Multi-Method Assessment Fusion	机器学习融合多特征	0–100（越大越好）	**强**

VMAF 是 Netflix 开源的，工业界公认与人眼感知相关性最高。经验值：

- VMAF ⩾ 93：视觉无损
- VMAF ≈ 80：高质量
- VMAF ≈ 60：可接受
- VMAF ⩽ 40：明显压缩痕迹

📌 本章要点回顾

1. 视频能压缩到原始大小 1% 以下，靠两把斧：**帧内压缩**（压每张图）+ **帧间压缩**（只存帧间差异）。

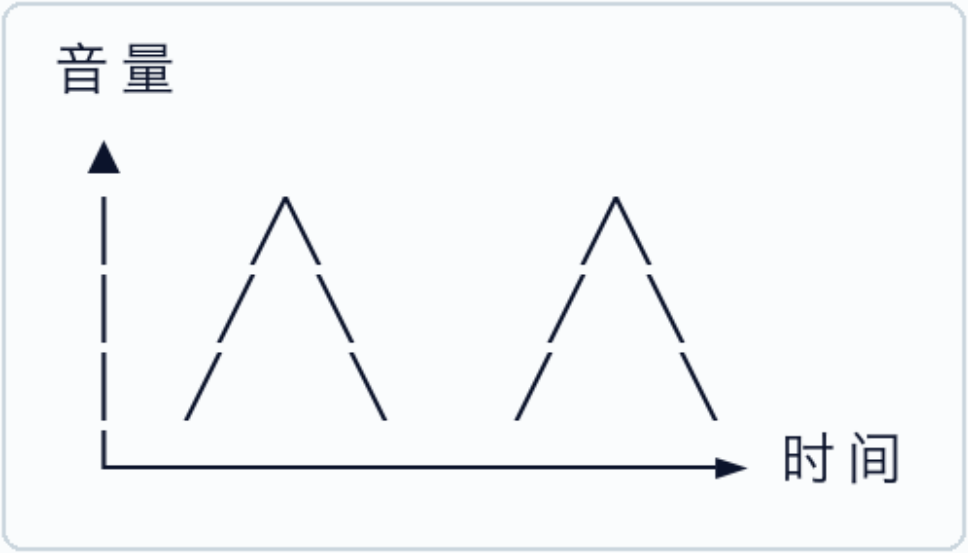
2. 编码慢、解码快；编码器 + 解码器 = **Codec**。
3. **H.264** 是通用兜底，**H.265** 省 37% 带宽但授权贵，**AV1** 免费且压缩更强但编码慢，**VVC** 是未来但还没普及。
4. 别照抄"H.265 比 H.264 省 50%" 这类绝对数据，实际效率随内容变化很大。
5. VOD 转码 4 件必做：**CRF 控质量、GOP 对齐、faststart、关闭 scene-cut**。
6. **Per-title / per-shot** 编码是高阶优化，云厂商的智能转码模板已覆盖大多数收益。
7. **VMAF** 是衡量画质的主流工业指标。

> 本章你会理解：声音怎么变成数字、采样率/位深是什么、声道/立体声/5.1 的含义、AAC 为什么是流媒体首选。

预计阅读时间：12 分钟

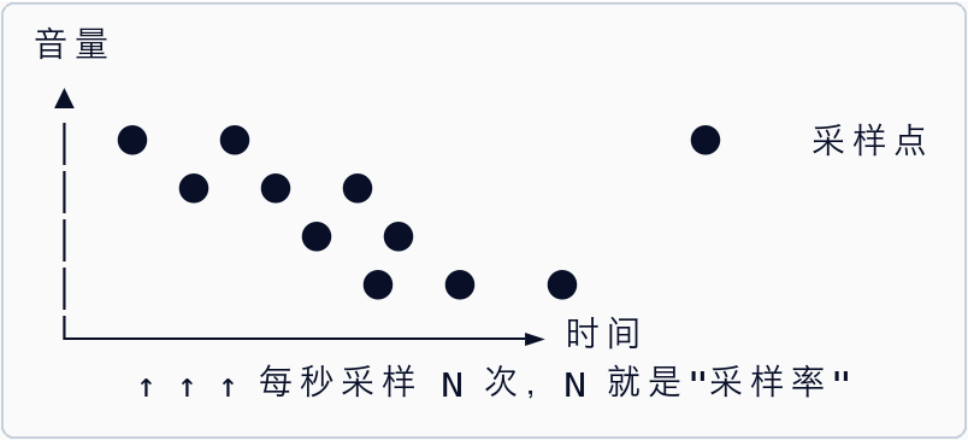
3.1 声音怎么变成数字

声音本质是空气的振动，是一条连续变化的波形：



电脑只能存数字，不能存连续波形。所以要做两件事：

1. **采样（Sampling）**：每隔一小段时间，测一次波形的高度
2. **量化（Quantization）**：把测到的高度数字化



3.2 采样率（Sample Rate）

单位：Hz（赫兹，每秒多少次）

常见采样率：

采样率	场景
8 kHz	电话语音
16 kHz	语音识别、VoIP（Zoom/Teams）
22.05 kHz	老游戏、AM 收音机
44.1 kHz	**CD 唱片、音乐首选**
48 kHz	**视频领域默认**（电影、流媒体、广播）
96 kHz	高保真录音
192 kHz	专业录音室

> 💡 **奈奎斯特定理**：要还原频率 F 的信号，采样率至少要 2F。人耳能听到的频率上限大约 20 kHz，所以 44.1/48 kHz 刚好够（还留有一点余量）。

> 📌 **VOD 音频统一用 48 kHz**。如果你有 44.1 kHz 的源，转码时用 `-ar 48000` 重采样。

3.3 位深（Bit Depth）

每个采样点用几个 bit 存它的"高度"：

位深	能表达的响度级数	场景
8 bit	256 级	老游戏、电话
16 bit	**65,536 级**	**CD、消费流媒体**

位深	能表达的响度级数	场景
24 bit	约 1700 万级	专业录音
32 bit 浮点	天文数字	音频制作内部格式

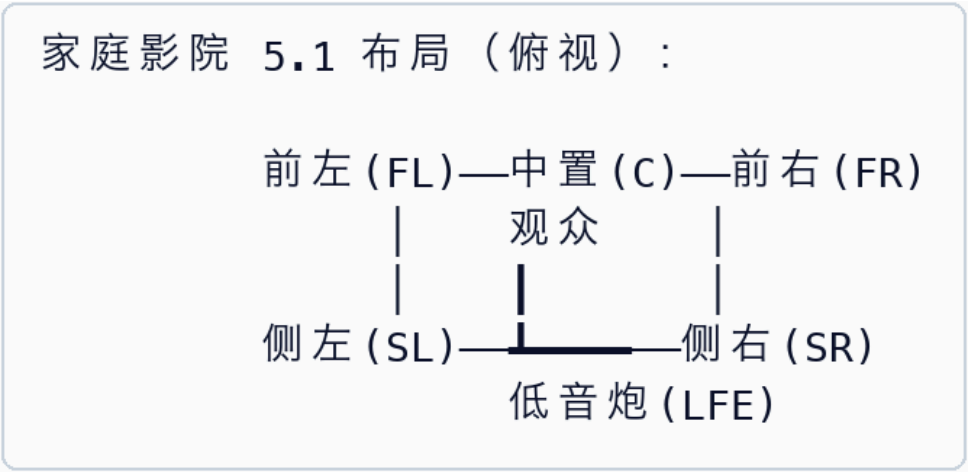
大多数视频里音频都是 16-bit 48 kHz。

3.4 声道（Channel）

声道 = 有几条独立的声音轨。

声道	英文	配置	用途
1.0	Mono	单声道	电话、老电视
2.0	Stereo	左 + 右	**音乐、大部分视频**
5.1	5.1 surround	前左+中+前右+后左+后右+**低音**（.1 指低频炮）	电影院、家庭影院
7.1	7.1 surround	5.1 + 两个侧向	顶级家庭影院
7.1.4	Atmos 等	7.1 + **4 个天空声道**	杜比全景声

布局



3.5 音频码率：多少 kbps 够用?

音频码率也是每秒 bit 数。视频码率是几 Mbps，音频码率是几十到几百 kbps，**只占视频码率的 5–10%**。

码率	听感	场景
32 kbps	能听清语音、音乐破破的	极低带宽
64 kbps	语音清晰、音乐勉强	低码率场景
96 kbps	音乐尚可	广播、YouTube 默认
128 kbps	**音乐好听**	**流媒体默认**
192 kbps	高保真	高质量音乐
256 kbps	发烧级	Apple Music
320 kbps	MP3 最大	音乐爱好者
无损 FLAC	透明	发烧 HiFi

> 📌 **VOD 视频配音频**：立体声 128 kbps AAC 是绝大多数场景的正确答案。

3.6 主流音频编码

AAC（Advanced Audio Coding）：流媒体首选

- **由谁**：MPEG（同 H.264 的组织）
- **年份**：1997
- **兼容性**：**所有**视频平台、浏览器、手机都支持 ✅✅✅
- **变体**：
  - **AAC-LC**（Low Complexity）：最常用 ✅ HLS/DASH 默认
  - **HE-AAC**（High Efficiency）：低码率下更好（64 kbps 以下）
  - **HE-AAC v2**：HE-AAC + 参数化立体声，48 kbps 下依然不错

> 📌 **绝大多数 VOD 项目**，音频用 AAC-LC、48 kHz、立体声、128 kbps，就完事了。

MP3：退出历史舞台

- 经典但效率低于 AAC
- 2017 年原始专利到期
- 新项目没理由再用 MP3

Opus：Web 新贵

- **开源、免版税**
- 从 6 kbps（语音）到 510 kbps（音乐）都表现优秀
- **WebRTC 默认**、Discord 使用

- 但 HLS/DASH 兼容性**不如 AAC**，iOS/Safari 支持有限

Dolby 系列：电影院味儿

编码	用途
AC-3（Dolby Digital）	5.1 声道、蓝光、老 HDTV
E-AC-3 / DD+（Dolby Digital Plus）	5.1 / 7.1，流媒体电影
Dolby Atmos（基于 E-AC-3 + JOC 或 AC-4）	**全景声**，顶级平台

> 💡 **Dolby Atmos 在 Netflix、Disney+、Apple TV+ 是高价值订阅的标志。**

FLAC / ALAC：无损

无损压缩，只能减小 50%–70%，但**完全还原**原始 PCM。用于：

- Apple Music 无损档
- 音乐发烧友
- 视频领域基本不用（码率太大）

3.7 多语言音轨：一个视频带多种语言

一个视频文件里可以装**多条音轨**：

```
MP4 file
├── video track   (H.264)
├── audio track 1 (AAC, English)
├── audio track 2 (AAC, Chinese)
├── audio track 3 (AAC, Japanese)
└── subtitle track (WebVTT)
```

流媒体协议（HLS/DASH）支持**独立分发音轨**，播放器可以只下载用户选中的语言。

对应的 HLS manifest 配置大致如下：

```
#EXT-X-MEDIA:TYPE=AUDIO,GROUP-ID="audio",LANGUAGE="en",NAME="English",DEFAULT=YES,URI="audio/en/index.m3u8"
#EXT-X-MEDIA:TYPE=AUDIO,GROUP-ID="audio",LANGUAGE="zh",NAME="中文",URI="audio/zh/index.m3u8"
```

详见 05-流媒体协议。

3.8 响度标准化（Loudness Normalization）

你一定遇到过：切到广告音量突然变大。这是因为不同内容的"响度"差别很大。

响度标准化是按感知响度（不是峰值音量）统一各内容的响度水平。

常用标准

标准	目标响度	用途
EBU R128	–23 LUFS	欧洲广电
ATSC A/85	–24 LUFS	北美广电
Apple Music / Spotify	–14 LUFS	流媒体音乐
YouTube	–14 LUFS	默认
短视频/移动端	多数 –16 ~ –14 LUFS	手机小喇叭上下限

LUFS（Loudness Units Full Scale）是国际标准的感知响度单位。

> 🛠️ **ffmpeg 做响度标准化**：
>
> ```
> # 把音频规范到 -14 LUFS
> ffmpeg -i input.mp4 -af loudnorm=I=-14:TP=-1.5:LRA=11 -c:v copy
>         output.mp4
> ```

3.9 动手：查看和转码音频

> 🛠️ **动手试一试。**

查看一个视频里有几条音轨

```
ffprobe -v error -show_streams -select_streams a input.mp4
```

典型输出：

```
codec_name=aac
sample_rate=48000
channels=2
channel_layout=stereo
bit_rate=128000
```

把多种音频统一转成 AAC 48 kHz 128 kbps 立体声

```
ffmpeg -i input.mov \
  -c:a aac -b:a 128k -ar 48000 -ac 2 \
  -c:v copy \
  output.mp4
```

- `-c:a aac`：音频编码 AAC
- `-b:a 128k`：码率 128 kbps
- `-ar 48000`：采样率 48 kHz
- `-ac 2`：声道数 2（立体声）
- `-c:v copy`：视频不动、直接复制（节省时间）

📌 本章要点回顾

1. 声音数字化需要**采样率**（时间轴密度）和**位深**（幅度精度）。
2. VOD 默认采样率 **48 kHz**、位深 **16-bit**。
3. 消费流媒体默认声道**立体声（2.0）**，电影级用 **5.1 / Atmos**。
4. **AAC-LC 128 kbps** 是 VOD 项目的默认音频设置。
5. 一个视频文件可以带多条音轨（多语言）。
6. 响度标准化（EBU R128 / −14 LUFS）能避免"切广告就变吵"。

> 本章你会理解：文件后缀和编码格式的区别、MP4/MKV/TS/WebM 是什么、为什么流媒体都用 fMP4、CMAF 又是干嘛的。

预计阅读时间：18 分钟

4.1 一个最容易搞混的概念：容器 ≠ 编码

先看一个每个新人都会犯的错：

> ❌ "视频编码是 MP4。"
>
> ✅ "视频容器是 MP4，编码是 H.264（或 H.265、AV1 等）。"

MP4 是一个**容器**（container），是一个盒子。盒子里可以装任何支持的编码流。

> 💡 **类比**：MP4 就像"快递盒"；H.264 是盒子里装的"商品"。同样一个快递盒可以装手机（H.264）、装衣服（H.265）、装水果（AV1）。

4.2 为什么需要容器?

直接存编码后的二进制流不行吗?

不行。编码流里没有这些信息:

- 视频和音频怎么**同步播放**?
- 字幕怎么对齐?

- 用户拖进度条到 1 分钟，从哪个字节开始读？
- 视频有几条音轨？
- 视频是什么编码？参数？

容器格式负责**组织**这一切，让播放器能**按时间轴**读取。

4.3 主流容器格式对比

容器	后缀	发明者	主要装什么	适合
MP4 / fMP4	`.mp4` `.m4s` `.m4a`	MPEG	H.264/H.265/AV1 + AAC	**最通用**、流媒体默认
MOV	`.mov`	Apple	几乎任意	macOS 制作、素材中转
MKV	`.mkv`	CoreCodec	几乎任意	高清下载、开源社区
WebM	`.webm`	Google	VP9/AV1 + Opus	Web 端（非 iOS）
MPEG-TS	`.ts`	MPEG	H.264/HEVC + AAC/AC3	广电、老版 HLS
FLV	`.flv`	Adobe	H.264 + AAC	已退役，RTMP 残留
AVI	`.avi`	Microsoft	–	**已过时**，不要用
3GP	`.3gp`	3GPP	H.263 + AMR	2G 时代手机，已过时

> 📌 **VOD 项目 99% 的情况只需要两种容器**：
>
> - MP4 / fMP4（流媒体发布）
> - MOV（素材中转、母版）

4.4 MP4 / ISO BMFF 的内部结构

MP4 的正式标准叫 **ISO Base Media File Format（ISO BMFF）**，标准号 ISO/IEC 14496-12。

内部由一个个叫 **Box**（盒子，也叫 atom）的单元组成，Box 可以嵌套：

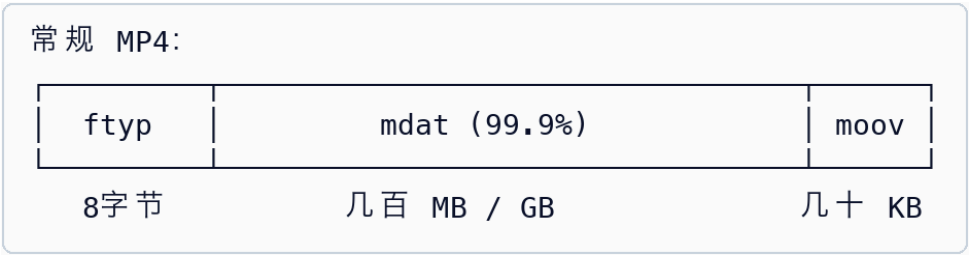
```
File start
│
├── [ftyp]  File Type Box  ← 告诉播放器"这是 MP4"
│
├── [moov]  Movie Box      ← 元数据"目录表":
│    ├── [mvhd]  Movie Header (总时长、时间尺度)
│    ├── [trak]  Track Box (每条音频/视频轨一个)
│    │    ├── [tkhd] Track Header
│    │    └── [mdia] Media
│    │         ├── [mdhd] Media Header
│    │         ├── [hdlr] Handler (vide/soun/subt)
│    │         └── [minf]
│    │              └── [stbl] Sample Table  ← 每一帧在文件哪个字节
│    └── [trak] ... (更多轨)
│
└── [mdat]  Media Data Box  ← 真正的音视频二进制数据
     └── (大块二进制)
```

关键概念：

- `moov`：**元数据 / 目录**。告诉播放器"第 1 帧在字节 12345，第 2 帧在字节 13800..."。没有它不知道怎么播。
- `mdat`：真正的**数据**。纯粹的 H.264 + AAC 二进制。

4.5 关键陷阱：moov 的位置决定能不能"边下边播"

传统 MP4 生成时 **moov 会放在文件末尾**（因为编码完成后才知道完整的时间戳信息）。

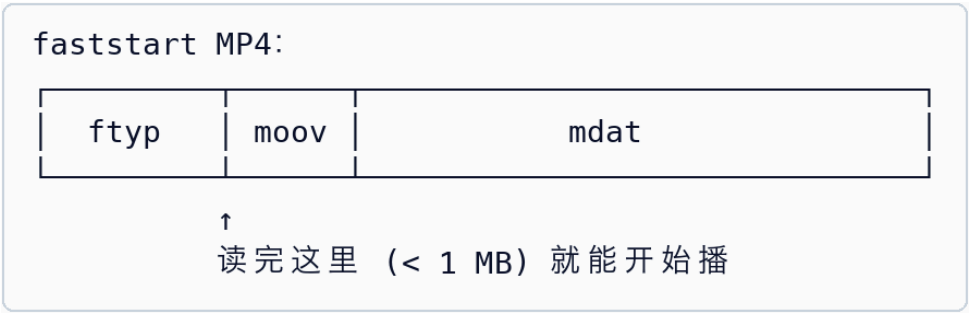


问题来了：Web 播放器要播必须先读 moov，但 moov 在文件末尾 → 要把**整个文件下载完才能开始播**！

这就是你看到"网页视频要等进度条走到 100% 才能播"的原因。

解决：`moov` 前置（faststart）

重新打包，把 moov 挪到 `ftyp` 之后：



用 ffmpeg 加 `-movflags +faststart` 就行：

```
ffmpeg -i input.mov -c copy -movflags +faststart output.mp4
```

> 📌 **VOD 发布前必做 faststart**。没做的话用户点播后会很久没反应。

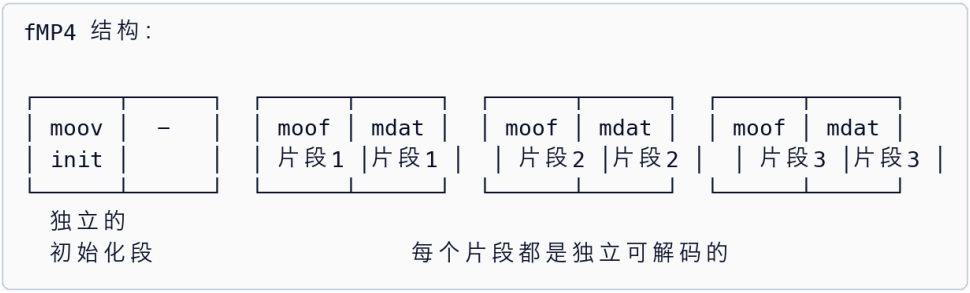
4.6 fMP4（Fragmented MP4）：流媒体的正确选择

传统 MP4 有个大缺点：**`moov` 描述的是整个文件的时间轴**。如果内容很长（几小时的电影），`moov` 会非常大（MB 级），启动慢、修改成本高。

fMP4（Fragmented MP4） 的做法：

> 把文件拆成很多小片段（fragment），**每个片段都自带自己的小 moov**（叫 `moof`）。播放器可以独立解析每个片段。

fMP4 结构：



fMP4 有两个核心好处：

1. **可独立切片**：每个 `moof+mdat` 对可以单独存为一个文件（`.m4s`）。这正是流媒体分发需要的。
2. **启动只需很小的 init segment**（几十 KB），而不是整个 `moov`。

> 📌 **现代 HLS、DASH、CMAF 全部基于 fMP4**。TS（老派 HLS）正在被淘汰。

4.7 MPEG-TS：老 HLS 的默认载体

MPEG-TS（`.ts` 文件）是为**广播电视**设计的格式：

- 所有数据切成**固定 188 字节**的小包
- 每个包自带同步字（0x47 sync byte）和时间戳
- 设计目标：卫星信号偶尔丢包也能恢复

```
TS 结构：

┌─────┬─────┬─────┬─────┬─────┬─────┬─────┐ ...
│188B │188B │188B │188B │188B │188B │188B │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┘

 pkt1  pkt2  pkt3  ...
```

HLS 2009 年诞生时 iOS 强制使用 `.ts`。但 TS 有几个问题：

- **开销大**：每 188 字节就要一个 4 字节头（开销约 2%）
- **PAT/PMT 表** 每隔一段时间就要重复
- **和 MP4 生态割裂**：要多存一套 TS 版本

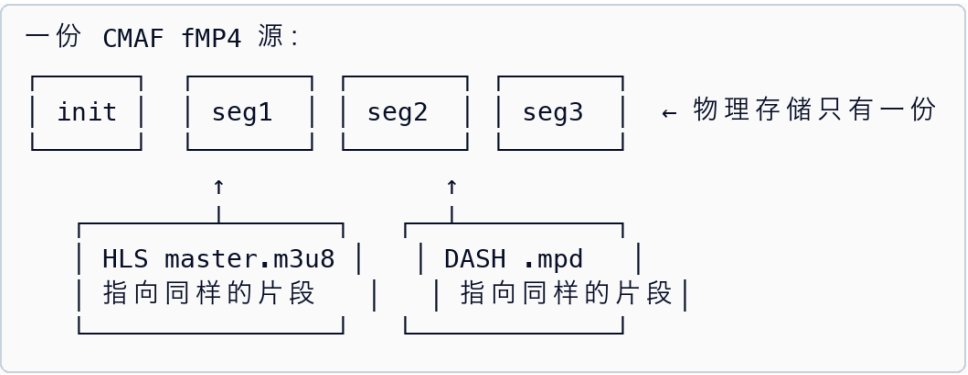
iOS 10（2016）起支持 fMP4，**新项目应统一用 fMP4**，可以被 HLS/DASH 共用。

4.8 CMAF：一份文件通吃 HLS 和 DASH

问题：历史上 Apple 主推 HLS（TS），Google/工业界主推 DASH（fMP4），同一个视频要存两份。

CMAF（Common Media Application Format） 2018 年标准化，核心约定：

> **HLS 和 DASH 共用一份 fMP4 文件。Manifest 不同，segment 完全相同。**



收益：

- 存储减半
- CDN 缓存命中率翻倍
- 转码只跑一次

代价：

- 所有平台要兼容 CMAF（现代平台都支持）
- DRM 需要用 **CBCS 模式**（兼容 FairPlay）

> 📌 **2020 年后新项目请直接用 CMAF**。不要再做"HLS TS + DASH fMP4 双发"。

4.9 段（Segment）的长度选多少?

VOD 切片长度是一个工程权衡：

切片长度	优点	缺点	适合
1–2 秒	起播快、ABR 反应快、低延迟直播	文件多、HTTP 请求多	**短剧、短视频、低延迟直播**

切片长度	优点	缺点	适合
2–4 秒	平衡	–	**HLS/DASH 默认推荐**（4 秒）
6–10 秒	HTTP 请求少、CDN 友好	起播慢、拖动定位粗	长电影、传统广电

> 📌 **短剧/短视频选 2 秒**（因为用户频繁滑动切剧集）；长视频 VOD 选 4–6 秒。

切片长度必须是 GOP 的整数倍，见 01 章关键帧部分。

4.10 字幕装进容器的几种方式

字幕有三种组织方式：

① Sidecar（外挂字幕）

字幕是独立文件，不在视频容器里：

```
video.mp4
video.en.vtt    ← 英文字幕
video.zh.vtt    ← 中文字幕
```

HLS/DASH manifest 引用这些字幕文件。

优点：容易多语言切换、改字幕不用重做视频。 **缺点**：多一些 HTTP 请求。

② 内嵌（Embedded）

字幕作为一条 track 放在 MP4 里：

```
video.mp4
├── video track
├── audio track
└── subtitle track (TTML / WebVTT)
```

③ 烧录（Burned-in / Hardcoded）

把字幕直接绘制到视频画面上。像素级合并，不可关闭。

> 💡 **适合哪种?**
>
> - **VOD 多语言**：用 Sidecar（外挂 WebVTT/IMSC1）
> - **短视频**：有时烧录（字幕是创意的一部分）
> - **广电**：内嵌 EIA-608/708

4.11 常见字幕格式

格式	特点	用在哪
SRT	最简单，文本+时间戳	通用
WebVTT	SRT 增强（样式、定位）	HTML5 / HLS 标准
TTML / IMSC1	XML，支持复杂排版	DASH、广电
ASS / SSA	强大样式，动画特效	番剧爱好者
EIA-608 / 708	广电字幕编码	传统电视、直播 CC

一个 WebVTT 例子：

```
WEBVTT

00:00:01.000 --> 00:00:03.500
Hello, and welcome to our show.

00:00:03.500 --> 00:00:06.000
Today we talk about video streaming.
```

4.12 动手：用 ffmpeg 做封装相关操作

> 🛠️ **动手试一试**

① 查看一个 MP4 里有什么

```
ffprobe -v error -show_streams input.mp4
```

会列出所有 video / audio / subtitle 流。

② 把 `.mov` 无损转成 fMP4 供流媒体使用

```
ffmpeg -i input.mov \
  -c copy \
  -movflags +faststart+frag_keyframe+empty_moov+default_base_moof \
  output_fragmented.mp4
```

解释：

- `-c copy`：不重新编码，直接复制
- `+faststart`：moov 前置
- `+frag_keyframe`：在每个关键帧切片

- `+empty_moov +default_base_moof`：让 moov 为空（只放 init），数据进 moof/mdat

③ 把长视频切成 HLS 切片（fMP4 格式）

```
ffmpeg -i input.mp4 \
  -c:v libx264 -preset slow -crf 22 -g 60 -keyint_min 60 -sc_thresho
       ld 0 \
  -c:a aac -b:a 128k \
  -f hls \
  -hls_time 4 \
  -hls_segment_type fmp4 \
  -hls_playlist_type vod \
  -hls_list_size 0 \
  -hls_segment_filename "seg_%04d.m4s" \
  output.m3u8
```

输出会是：

```
output.m3u8      ← HLS manifest
init.mp4         ← CMAF init segment
seg_0000.m4s
seg_0001.m4s
seg_0002.m4s
...
```

这就是一个**最小可用的 HLS 流媒体包**。把它放到一个 Web 服务器（甚至 `python3 -m http.server`），用 Safari 或 hls.js 就能播。

📌 本章要点回顾

1. **容器 ≠ 编码**。MP4 是容器，H.264 是编码。
2. MP4 内部是由 Box 组成的结构；`moov` 是目录，`mdat` 是数据。

3. **VOD 发布必做** `-movflags +faststart`，让 moov 前置支持边下边播。
4. **fMP4** 把文件切成独立片段，是现代流媒体的基础。
5. **CMAF** 让 HLS 和 DASH 共用一份 fMP4 文件，存储减半。
6. TS 容器是老派 HLS 用的，新项目请用 fMP4/CMAF。
7. 切片长度：短剧短视频 2 秒、长 VOD 4–6 秒。
8. 字幕优先用 Sidecar WebVTT。

第三部分 · 核心篇：协议/ABR/CDN

> 本章你会理解：为什么不能直接下载 MP4、HLS 和 DASH 是怎么工作的、m3u8 / mpd 文件里到底写了什么、LL-HLS 为什么存在。

预计阅读时间：25 分钟

5.1 为什么不能直接下载 MP4 就完事?

最朴素的做法：把 `video.mp4` 放到 HTTP 服务器，用户浏览器请求 → 下载 → 播放。

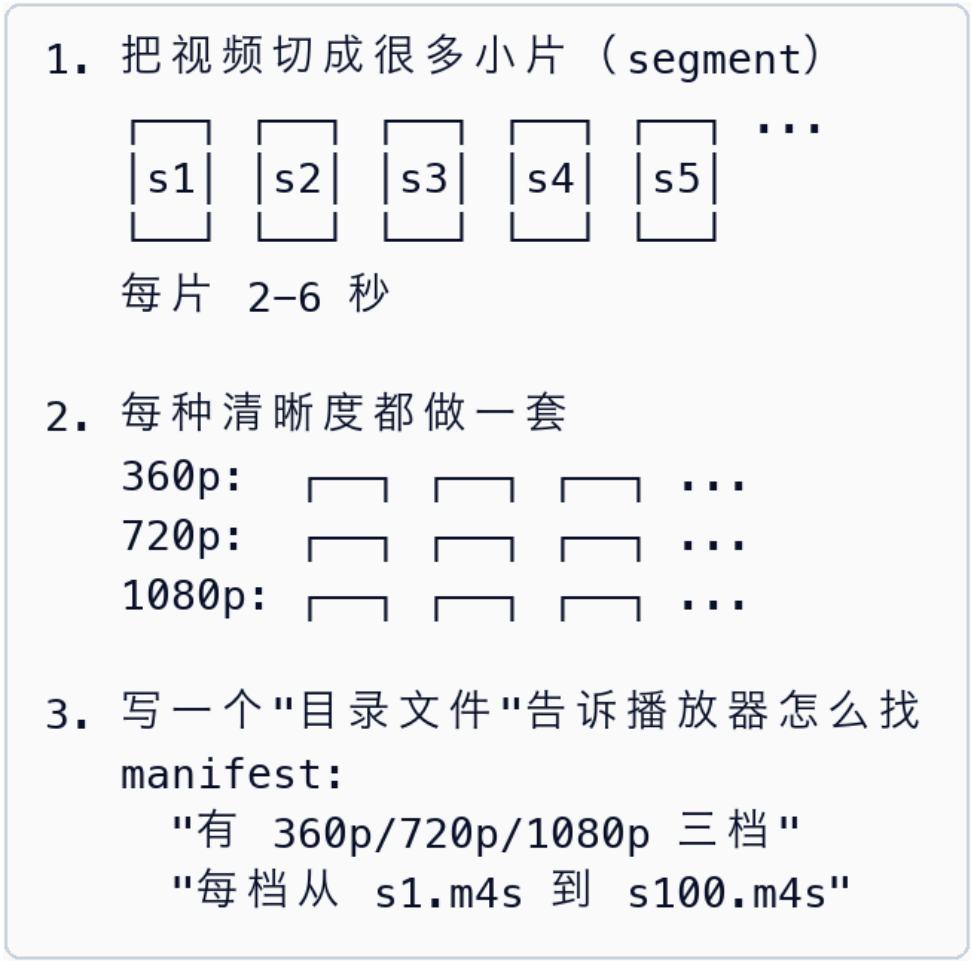
这叫 **渐进式下载（Progressive Download）**。它有几个致命缺点：

1. **没做 faststart 时，要下完才能播**（见 04 章）。
2. **网速变差时还是要硬下**，不能切低清晰度 → 卡顿。
3. **拖动进度条**靠 HTTP Range 头，但一个大文件请求范围 CDN 缓存不友好。
4. **不能根据设备能力给不同版本**：iPhone 收到 1080p 只能硬扛。

所以需要更聪明的做法——**把视频切成小片、配合一个"目录文件"指挥播放器按需下载**。这就是**流媒体协议（Streaming Protocol）**。

5.2 流媒体协议的核心思想

现代流媒体三件套：



播放器看到 manifest 后：

- **第 1 秒**：选一个保守档位开始下
- **第 2 秒**：看实际网速，决定下一片用哪档
- **不停决策**：网好升档、网差降档、卡顿就进一步降档

这个**"看网况换档"**的决策叫 **自适应码率（ABR）**，第 6 章 专门讲。本章先讲协议本身。

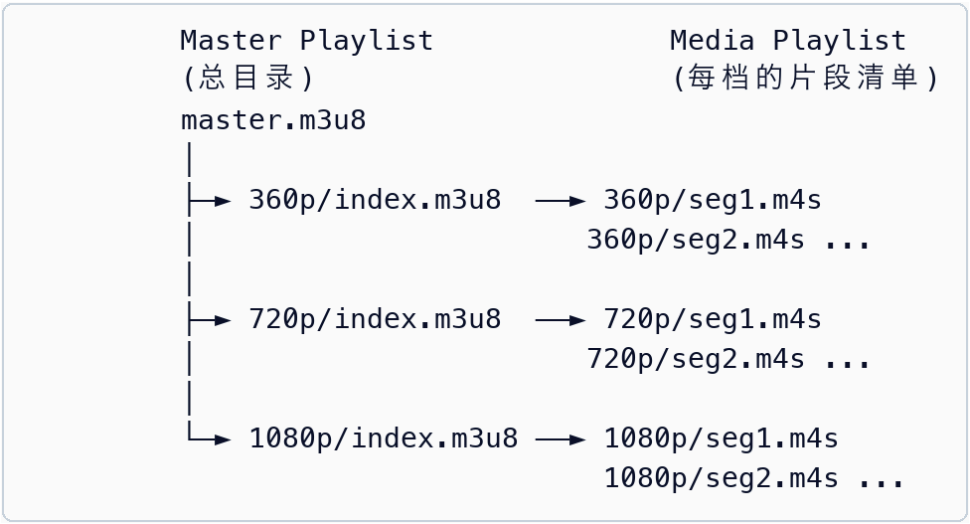
5.3 HLS（HTTP Live Streaming）

发明者：Apple，2009 年随 iOS 3.0 一起推出。

标准：IETF RFC 8216（已更新到 `draft-pantos-hls-rfc8216bis`）。

核心：两级 M3U8 播放列表

M3U8 本质是一个 UTF-8 纯文本文件（扩展的 M3U 格式）。HLS 用两级：



Master Playlist 例子

```
#EXTM3U
#EXT-X-VERSION:7

#EXT-X-STREAM-INF:BANDWIDTH=800000,RESOLUTION=640x360,CODECS="avc1.6
40016,mp4a.40.2"
360p/index.m3u8

#EXT-X-STREAM-INF:BANDWIDTH=2500000,RESOLUTION=1280x720,CODECS="avc
1.64001f,mp4a.40.2"
720p/index.m3u8

#EXT-X-STREAM-INF:BANDWIDTH=5000000,RESOLUTION=1920x1080,CODECS="avc
1.640028,mp4a.40.2"
1080p/index.m3u8
```

每行含义：

- `#EXTM3U`：文件头，必须是第一行
- `#EXT-X-VERSION:7`：HLS 版本（fMP4 需要 ⩾ 7）
- `#EXT-X-STREAM-INF:...`：声明一个清晰度
  - `BANDWIDTH=2500000`：这一档最大码率 2500 kbps（**必填**）
  - `RESOLUTION=1280x720`：分辨率
  - `CODECS="avc1..,mp4a.."`：编码字符串（RFC 6381），告诉浏览器编码
- 下一行是对应的 Media Playlist URL

Media Playlist 例子（fMP4 + VOD）

```
#EXTM3U
#EXT-X-VERSION:7
#EXT-X-TARGETDURATION:4
#EXT-X-PLAYLIST-TYPE:VOD
#EXT-X-MAP:URI="init.mp4"

#EXTINF:4.000,
seg_00001.m4s
#EXTINF:4.000,
seg_00002.m4s
#EXTINF:4.000,
seg_00003.m4s
...
#EXTINF:3.120,
seg_00150.m4s

#EXT-X-ENDLIST
```

每行含义：

- `#EXT-X-TARGETDURATION:4`：声明最大切片时长 4 秒
- `#EXT-X-PLAYLIST-TYPE:VOD`：VOD 模式（另一种是 EVENT 或 LIVE）
- `#EXT-X-MAP:URI="init.mp4"`：fMP4 的初始化段位置
- `#EXTINF:4.000,`：下一行切片时长 4 秒
- `seg_00001.m4s`：切片文件路径
- `#EXT-X-ENDLIST`：列表结束（VOD 必须有；直播没有）

多音轨、字幕怎么声明

```
#EXTM3U
#EXT-X-VERSION:7

# 音频组
#EXT-X-MEDIA:TYPE=AUDIO,GROUP-ID="audio",LANGUAGE="en",NAME="Englis
h",DEFAULT=YES,URI="audio_en/index.m3u8"
#EXT-X-MEDIA:TYPE=AUDIO,GROUP-ID="audio",LANGUAGE="zh",NAME="中文",U
RI="audio_zh/index.m3u8"

# 字幕组
#EXT-X-MEDIA:TYPE=SUBTITLES,GROUP-ID="subs",LANGUAGE="en",NAME="Engl
ish",URI="subs_en/index.m3u8"
#EXT-X-MEDIA:TYPE=SUBTITLES,GROUP-ID="subs",LANGUAGE="zh",NAME="中
文",URI="subs_zh/index.m3u8"

# 视频流（绑定音频组和字幕组）
#EXT-X-STREAM-INF:BANDWIDTH=2500000,RESOLUTION=1280x720,CODECS="avc
1.64001f,mp4a.40.2",AUDIO="audio",SUBTITLES="subs"
720p/index.m3u8
```

> 📌 **HLS 的亮点**：
>
> - **所有 iOS / Safari 原生支持**，不用装插件
> - 用 HTTPS 传输，穿透防火墙能力强
> - 简单文本格式，人类可读
> - **全球市场占有率最高**

5.4 DASH（Dynamic Adaptive Streaming over HTTP）

发明者：MPEG（ISO/IEC 23009-1），2012 年标准化。目的是做一个**开放标准**对抗 Apple 私有的 HLS。

核心：manifest 是 XML 文件 `.mpd`（**M**edia **P**resentation **D**escription）。

MPD 简化例子

```
<?xml version="1.0" encoding="UTF-8"?>
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011"
     type="static"
     mediaPresentationDuration="PT10M30S"
     minBufferTime="PT2S"
     profiles="urn:mpeg:dash:profile:isoff-on-demand:2011">

  <Period>
    <!-- 视频 -->
    <AdaptationSet mimeType="video/mp4" codecs="avc1.64001f">
      <Representation id="360p" bandwidth="800000" width="640" heigh
        t="360">
        <BaseURL>360p/</BaseURL>
        <SegmentTemplate initialization="init.mp4"
                         media="seg_$Number$.m4s"
                         timescale="1000"
                         duration="4000"
                         startNumber="1"/>
      </Representation>
      <Representation id="720p" bandwidth="2500000" width="1280" hei
        ght="720">
        <BaseURL>720p/</BaseURL>
        <SegmentTemplate initialization="init.mp4"
                         media="seg_$Number$.m4s"
                         timescale="1000"
                         duration="4000"
                         startNumber="1"/>
      </Representation>
    </AdaptationSet>

    <!-- 音频 -->
    <AdaptationSet mimeType="audio/mp4" codecs="mp4a.40.2" lang="e
        n">
      <Representation id="audio_en" bandwidth="128000">
        <BaseURL>audio_en/</BaseURL>
        <SegmentTemplate initialization="init.mp4"
                         media="seg_$Number$.m4s"
```

```
                        duration="4000"/>
      </Representation>
    </AdaptationSet>
  </Period>
</MPD>
```

要点：

- **Period**：整个节目。电影通常一个 Period；带广告插入的会分多个 Period。
- **AdaptationSet**：一类媒体（视频 / 音频 / 字幕 / 某种语言）。
- **Representation**：同一类媒体的一个具体版本（360p、720p、1080p 分别一个）。
- **SegmentTemplate**：用模板描述切片文件名，避免列出每个切片（比 HLS 省空间）。

DASH vs HLS 关键区别

	HLS	DASH
Manifest 格式	M3U8（文本）	MPD（XML）
切片容器	TS / fMP4（现代）	fMP4 / WebM
iOS/Safari	✅ 原生	❌ 需要 MSE
Android	✅	✅
Web（Chrome/Firefox）	✅ （通过 hls.js）	✅ （通过 dash.js / Shaka）
智能电视	大多支持	大多支持
开放标准	IETF 标准化中	ISO/IEC 标准
特色	Apple 生态原生	更灵活（codec-agnostic）

5.5 CMAF + 双 Manifest：工业界最佳实践

前面在 04 章 讲过 CMAF：**一份 fMP4 文件被 HLS 和 DASH 共用**。

典型目录结构：

```
/vod/episode-01/
  init.mp4                ← CMAF init segment (fMP4 init)
  seg_00001.m4s           ← 真正的视频数据
  seg_00002.m4s
  seg_00003.m4s
  ...

  hls/
    master.m3u8           ← HLS manifest (引用上面的切片)
    audio/index.m3u8
    360p/index.m3u8
    720p/index.m3u8

  dash/
    manifest.mpd          ← DASH MPD (引用同样的切片)
```

播放过程：

- iPhone 用户请求 → 返回 `master.m3u8` → 播放器下载 `seg_*.m4s`
- Android 用户请求 → 返回 `manifest.mpd` → 播放器下载**同样的** `seg_*.m4s`

CDN 缓存就全命中了。

5.6 延迟问题：为什么直播要 30 秒?

传统 HLS/DASH **起步延迟动辄 20–30 秒**。算一下就知道：

```
编码器：   [pack 6s 切片]──────►
HLS 规范：客户端要看到 3 个切片才开始播 = 3 × 6 = 18s
再加上：   播放缓冲 + 网络传输 = 25-30s
```

对直播、互动、电商直播这个延迟太高了。

LL-HLS：Apple 的 2019 年方案

LL-HLS（Low-Latency HLS） 在 2019 WWDC 发布。目标端到端 **< 2 秒**。

核心手段：

1. **Partial Segment**：把 6 秒切片再切成 200-500ms 的小段（partial），让播放器能比整片完成更早拿到数据。

```
#EXT-X-PART:DURATION=0.250,URI="seg_42.0.m4s",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.250,URI="seg_42.1.m4s"
#EXT-X-PART:DURATION=0.250,URI="seg_42.2.m4s"
#EXTINF:4.000,
seg_42.m4s
```

2. **Blocking Playlist Reload**：播放器用 `?_HLS_msn=X&_HLS_part=Y` 请求，**服务器阻塞**直到有更新才响应（类似长轮询）。
3. **Preload Hint**：告诉播放器"下一个即将出现的是什么"：

```
#EXT-X-PRELOAD-HINT:TYPE=PART,URI="seg_43.0.m4s"
```

4. **HTTP/2 Push**（可选）：服务端直接推送关联的 part 文件（多数 CDN 已弃用这一点）。

LL-DASH / CMAF-LL：对等方案

DASH 侧通过 **HTTP Chunked Transfer Encoding**：编码器每产出一个 CMAF chunk（~300ms）立刻通过分块传输发出，不等整片完成。

协议 vs 延迟对比

协议	典型延迟	复杂度
传统 HLS（TS, 6s × 3）	20–30 秒	低
现代 HLS（fMP4, 4s × 3）	8–15 秒	低
LL-HLS / CMAF-LL	**2–5 秒**	中
WebRTC	**<500ms**	高

5.7 WebRTC：超低延迟的"另一条路"

WebRTC **不是 HLS/DASH 的升级版**，而是完全不同的技术：

	HLS/DASH	WebRTC
传输层	HTTP/HTTPS	UDP + DTLS + SRTP
编码	任意	VP8/VP9/H.264/AV1
CDN 友好	✅ HTTP 缓存	❌ 点对点或专用 SFU
延迟	2–30 秒	< 500ms
规模	千万并发轻松	受 SFU 能力限制
典型场景	电影、点播、一对多直播	视频会议、互动连麦、云游戏

> 📌 **VOD 不用 WebRTC**。本系列后面章节聚焦 HLS/DASH/CMAF。

5.8 协议选择实用指南

```
你做的是什么？
│
├── ① 纯 VOD（点播）
│      → HLS + DASH 双 manifest + CMAF fMP4
│
├── ② 普通直播（<10s 延迟即可）
│      → HLS + DASH + CMAF fMP4
│
├── ③ 低延迟直播（体育、电商，<3s）
│      → LL-HLS + LL-DASH + CMAF-LL
│
├── ④ 超低延迟（互动、连麦，<500ms）
│      → WebRTC 或 RTMP-over-QUIC
│
└── ⑤ 广电 IPTV（运营商盒子）
       → MPEG-TS over UDP/HTTP（不是本书重点）
```

5.9 动手：用 ffmpeg 生成一份 HLS+DASH 双发流

🛠️ **动手试一试**

HLS（fMP4）

```
ffmpeg -i input.mp4 \
  -c:v libx264 -preset slow -crf 22 -g 96 -keyint_min 96 -sc_thresho
      ld 0 \
  -c:a aac -b:a 128k \
  -f hls \
  -hls_time 4 \
  -hls_segment_type fmp4 \
  -hls_playlist_type vod \
  -hls_list_size 0 \
  -hls_segment_filename "hls/seg_%04d.m4s" \
  hls/index.m3u8
```

多档码率 HLS（直接一次出三档）

```
ffmpeg -i input.mp4 \
  -filter_complex "[0:v]split=3[v1][v2][v3]; \
    [v1]scale=640:360[v1out]; \
    [v2]scale=1280:720[v2out]; \
    [v3]scale=1920:1080[v3out]" \
  -map "[v1out]" -c:v:0 libx264 -b:v:0 800k -maxrate:v:0 850k -bufsi
      ze:v:0 1600k \
  -map "[v2out]" -c:v:1 libx264 -b:v:1 2500k -maxrate:v:1 2650k -buf
      size:v:1 5000k \
  -map "[v3out]" -c:v:2 libx264 -b:v:2 5000k -maxrate:v:2 5300k -buf
      size:v:2 10000k \
  -map 0:a -c:a aac -b:a 128k \
  -g 96 -keyint_min 96 -sc_threshold 0 \
  -f hls \
  -hls_time 4 \
  -hls_segment_type fmp4 \
  -hls_playlist_type vod \
  -hls_list_size 0 \
  -master_pl_name master.m3u8 \
  -var_stream_map "v:0,a:0 v:1,a:0 v:2,a:0" \
  "hls/v%v/index.m3u8"
```

输出：

```
hls/
  master.m3u8
  v0/index.m3u8  (360p)
  v1/index.m3u8  (720p)
  v2/index.m3u8  (1080p)
```

用 Shaka Packager 做 CMAF + HLS + DASH（生产级推荐）

```
# 先用 ffmpeg 转码成三档 mp4：input_360p.mp4 / 720p / 1080p
# 再用 shaka-packager 打包
packager \
  in=input_360p.mp4,stream=video,init_segment=cmaf/v0/init.mp4,segme
        nt_template=cmaf/v0/seg_\$Number\$.m4s \
  in=input_720p.mp4,stream=video,init_segment=cmaf/v1/init.mp4,segme
        nt_template=cmaf/v1/seg_\$Number\$.m4s \
  in=input_1080p.mp4,stream=video,init_segment=cmaf/v2/init.mp4,segm
        ent_template=cmaf/v2/seg_\$Number\$.m4s \
  in=input_720p.mp4,stream=audio,init_segment=cmaf/a0/init.mp4,segme
        nt_template=cmaf/a0/seg_\$Number\$.m4s \
  --segment_duration 4 \
  --hls_master_playlist_output=cmaf/master.m3u8 \
  --mpd_output=cmaf/manifest.mpd
```

输出：

```
cmaf/
  master.m3u8       ← HLS
  manifest.mpd      ← DASH
  v0/init.mp4 + v0/seg_*.m4s    (360p)
  v1/init.mp4 + v1/seg_*.m4s    (720p)
  v2/init.mp4 + v2/seg_*.m4s    (1080p)
  a0/init.mp4 + a0/seg_*.m4s    (audio)
```

一份 segment，HLS + DASH 共用 ✅ 。

📌 本章要点回顾

1. 流媒体 = 切片 + manifest + 客户端按需拉取。
2. **HLS**（Apple）用 M3U8 文本 manifest；**DASH**（MPEG）用 MPD XML。
3. iOS/Safari 原生只支持 HLS；其他平台都支持 DASH。
4. **CMAF 让 HLS 和 DASH 共用一份 fMP4**，是工业界最佳实践。
5. 传统 HLS 延迟 20–30 秒；**LL-HLS / CMAF-LL 可做到 2–5 秒**。
6. **WebRTC 是另一条路**（<500ms），但不是 VOD 用的。
7. 生产环境用 Shaka Packager 或云服务（MediaPackage）做打包，不要手搓。

> 本章你会理解：ABR 是什么、播放器用哪些策略切换清晰度、BOLA/MPC/Pensieve 算法的核心思想、工程上该怎么选。

预计阅读时间：20 分钟

6.1 生活场景：为什么它很重要

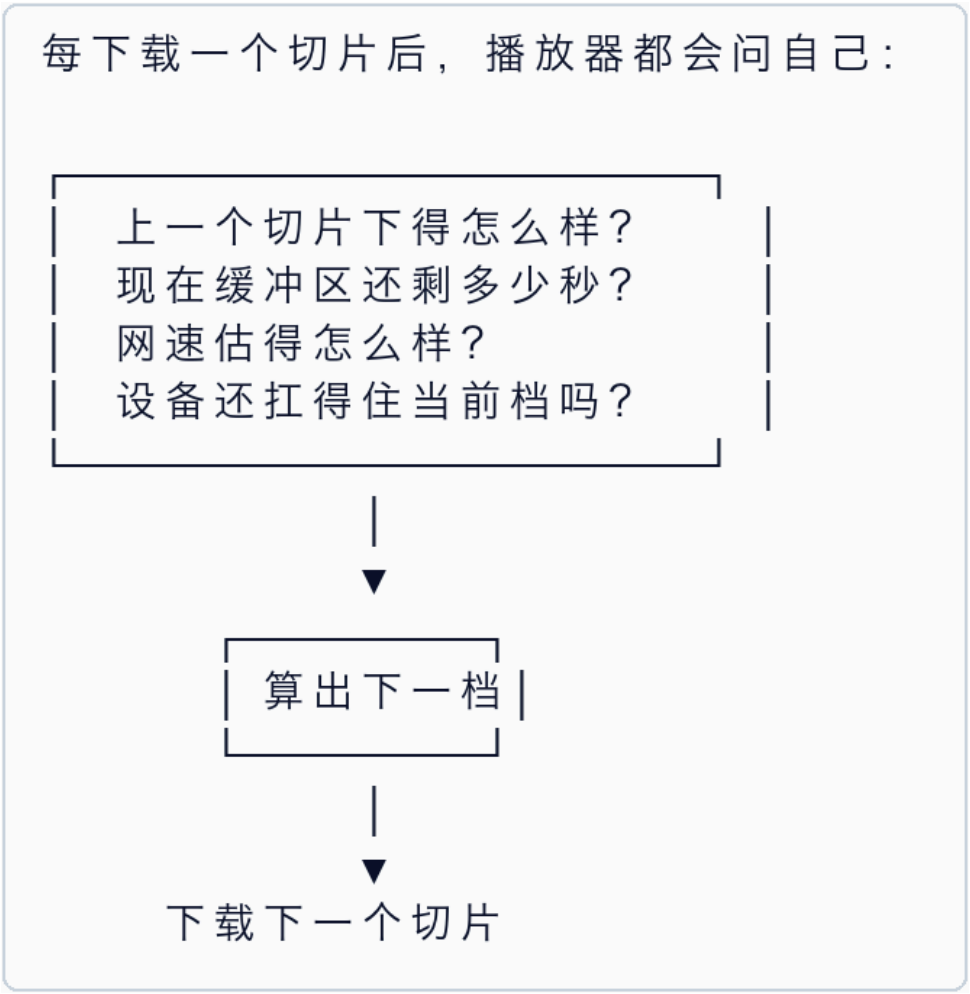
假设你在地铁上看剧，网络在 4G / 弱 4G / 5G 之间频繁切换。

- 如果 APP 锁定 1080p 不变 → 网差时会**卡到转圈圈**。
- 如果 APP 锁定 360p 不变 → 网好时画面**糊得心疼**。

理想状态：**APP 能自己根据网速和缓冲情况切换清晰度，让你不间断看完。**

这就是 **ABR（Adaptive Bitrate）自适应码率** 要做的事。

6.2 ABR 的工作循环



这个循环每 1–4 秒跑一次（每下完一片触发）。

关键信号：

- **最近吞吐量**（Throughput）：最近几片的实际下载速度
- **缓冲占用**（Buffer Level）：播放器已缓存但还没播的秒数
- **当前码率**（Current Bitrate）

- **可选码率集合**（Manifest 列出的 bitrate ladder）

6.3 策略一：基于吞吐（Throughput-Based）

最直觉的做法：

```
估计网速 = 最近 N 个切片的平均下载速度
下一档码率 ≈ 估计网速 × 0.8   （留 20% 余量）
```

示例：

- 最近下载速度 5 Mbps
- × 0.8 = 4 Mbps
- 在 360p(0.5)/720p(2.5)/1080p(5)/4K(15) 中选 ⩽ 4 Mbps 的最高档 → **720p**

优点：简单直观。 **缺点**：

- HTTP / TCP 层测速**噪声很大**（慢启动、队头阻塞、TLS 握手…）
- 遇到"突发慢"和"突发快"会频繁切换，用户看到画面**一会儿清一会儿糊**，体验差

代表实现：早期 hls.js 的 `bandwidthFraction` 策略。

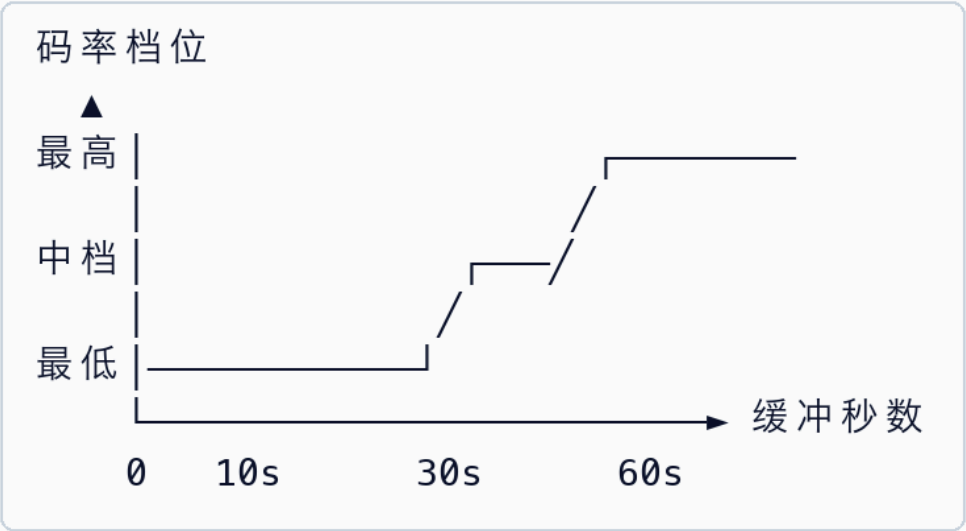
6.4 策略二：基于缓冲（Buffer-Based，BBA）

另一种思路：不管网速，只看**缓冲区还有多少秒**。

核心规则：

```
缓冲 < 10s  → 下最低档（稳住不卡）
缓冲 10–30s → 下中档
缓冲 > 30s  → 下最高档（反正你够看，多享受）
```

画成曲线：



代表方案：**BBA（Buffer-Based Adaptation）**，Huang 等人在 Stanford / Netflix 2014 年 SIGCOMM 论文 *A Buffer-Based Approach to Rate Adaptation* 提出。

优点：

- 对网速噪声免疫（不看网速）
- 稳定，不频繁切换

缺点：

- **启动阶段缓冲不足时很激进地降档**
- 没充分利用带宽（网好时可能选档偏低）

6.5 策略三：BOLA（基于 Lyapunov 优化）

BOLA = **B**uffer **O**ccupancy based **L**yapunov **A**lgorithm

- 作者：Spiteri、Urgaonkar、Sitaraman
- 2016 年 INFOCOM 论文 *BOLA: Near-Optimal Bitrate Adaptation for Online Videos*
- **dash.js 默认算法之一**

核心思想

BOLA 把 ABR 建模为**多目标优化问题**：

$$\max \quad \text{平均画质} \quad - \quad V \cdot (\text{卡顿惩罚} + \text{切换惩罚})$$

`V` 是权衡系数：

- V 大 → 偏保守，优先防卡
- V 小 → 偏激进，优先画质

然后用 **Lyapunov 优化理论** 证明：**只看当前缓冲就能做出近最优决策**。

简化公式

伪代码：

```python
def bola_select_bitrate(buffer_level, bitrates, V):
    best_bitrate = None
    best_score = -inf

    for r in bitrates:
        # 效用函数 = log(码率)  (边际递减)
        utility = log(r)
        # 奖励 = 当前缓冲 × 效用
        score = buffer_level * utility - V * r
        if score > best_score:
            best_score = score
            best_bitrate = r

    return best_bitrate
```

好处

- 理论可证接近最优
- 仅看缓冲，无需精确测网速
- 已在 dash.js 生产环境跑了 10+ 年

实际细节

dash.js 把 BOLA 和 ThroughputRule **动态组合**：

- 缓冲很低（启动时）：用 ThroughputRule
- 缓冲充足：用 BOLA

6.6 策略四：MPC（模型预测控制）

MPC（Model Predictive Control）

- 作者：Yin, Jindal, Sekar, Sinopoli

- 2015 年 SIGCOMM 论文 *A Control–Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP*

思想

与其"只看当下决定"，不如**预测未来几秒的网速，联合优化**未来 K 个切片的选择：

```
现在在第 n 个切片。
假设未来 5 个切片（到第 n+5）网速会是 T1, T2, T3, T4, T5。
试所有码率组合，选一个让：
  （总画质 − 卡顿惩罚 − 切换惩罚）最大
```

预测网速

常用做法：

- **Harmonic Mean**（调和平均）：`1 / mean(1/throughput_i)` —— 比算术平均对慢样本更敏感
- **EWMA**（指数加权移动平均）

优缺点

- 比 BOLA 更智能（看未来）
- 计算量稍大
- **网速预测不准时容易翻车**

6.7 策略五：Pensieve（AI 学习型 ABR）

Pensieve

- 作者：Mao, Netravali, Alizadeh (MIT)
- 2017 年 SIGCOMM *Neural Adaptive Video Streaming with Pensieve*

思想

用强化学习（A3C 算法）训练神经网络做 ABR 决策。

输入特征：

- 过去 K 秒吞吐量
- 当前缓冲
- 上一个码率
- 剩余切片数量
- Manifest 里各档码率

输出：下一个切片的码率选择

训练：在海量真实/模拟网络轨迹上跑仿真，神经网络自己学出最优策略。

效果

在特定测试集上超越 BOLA 和 MPC。但——

局限性：

- **泛化性**：训练时没见过的网络模式下表现不一定好
- **可解释性差**：出问题不好调
- **需要持续 online fine-tune**

工业界在用 Pensieve 吗?

- **头部公司**（Netflix、YouTube）自研了类似的学习型 ABR
- **中小平台**用 BOLA 或 MPC 已经足够

6.8 实际工业界是怎么做的?

Netflix

- 客户端侧用类似 BBA 的策略
- 服务端给 hint（例如"当前 CDN 负载高，建议你降档"）
- 结合 VMAF 决定 bitrate ladder

YouTube

- 自研，结合"用户观看时长预测"做联合优化
- 复杂的 A/B 测试体系

普通平台

- 直接用开源播放器（hls.js、Shaka Player、ExoPlayer）自带的 ABR
- 参数调优：`maxBufferLength`、`bandwidthFraction`、码率档位设置

6.9 短视频/短剧场景的特殊考虑

和长视频不同，短剧/短视频场景 ABR 需求变了：

特殊点

- **首帧时间（TTFF）最重要** ——用户滑动 300ms 没画面就走了
- **单集短**：完播 90 秒，ABR 可能还没来得及向上切就结束了
- **预载多集**：可能同时下载 N+1、N+2 的切片

常见调整

- **启动档位更低**（保证快速起播）

- **向上切换更慢**（防止刚起播就变清导致下载卡顿）
- **预载时下低档**（省流量）
- **正在播放的下高档**（画质优先）

6.10 ABR 常见问题与工程建议

❓ 为什么会"看着看着突然变糊"?

大概率是：

1. **网速实测下降**（4G 转弱信号、家里路由器切换了信道）
2. **CDN 边缘节点出问题**（换到另一个节点时慢）
3. **设备热控**（手机过热，降频导致解码跟不上）

❓ 为什么会"一直糊、明明网很好"?

- 启动策略太保守
- 测速累计样本还是慢的
- Manifest 里没有高档位（转码时没生成 1080p）

❓ 为什么频繁切换清晰度、看得眼花?

- 码率档位设得太密（每档差距小）
- 算法对吞吐噪声敏感（用 Throughput-based 但没滤波）
- 解决：**码率档位间距拉大到 1.5x 左右、加切换惩罚（switching penalty）、用 BOLA 替代 Throughput-based**

❓ 工程上 ABR 最重要的配置是什么?

不是算法，是 **bitrate ladder**（码率阶梯）！

一个不合理的阶梯，再好的算法也救不了。推荐阶梯设计原则：

- 相邻档位**码率比 1.5x–2x**（差太小切换无意义、差太大切换突兀）
- 最低档足够低（能覆盖 2G/弱 3G 用户）
- 最高档不浪费（1080p 用 8 Mbps 跟 5 Mbps 肉眼差别不大）

> 📌 **Per-title encoding**（02 章）就是要给每部片子定制最优 ladder。

6.11 动手：观察一个 ABR 决策过程

> 🛠️ **动手试一试**：在 Chrome 里用 hls.js 播放时观察 ABR。

1. 打开 `https://hls-js.netlify.app/demo/`
2. 粘贴一个 m3u8 URL（或用默认的 Apple 测试流）
3. 按 F12 打开 DevTools → Network 面板
4. 播放，你能看到每个 `.m4s` / `.ts` 切片的下载时间、大小
5. Network 面板打开 Throttling 选成 "Slow 3G"，观察 hls.js 自动降档

hls.js Stats 面板会显示：

- `bandwidthEstimate`：当前带宽估计
- `level`：当前档位 index
- `nextAutoLevel`：下一个切片会用哪档

📌 本章要点回顾

1. ABR = 自适应码率，每下一片后决定下一片下哪档。
2. 四类经典策略：
    - **Throughput-based**：看网速，简单但噪声敏感

- **Buffer-based (BBA)**：看缓冲，稳但偏保守
- **BOLA**：基于 Lyapunov，理论最优，生产可用
- **MPC**：预测未来，更智能
- **Pensieve**：强化学习，头部公司自研类似方案

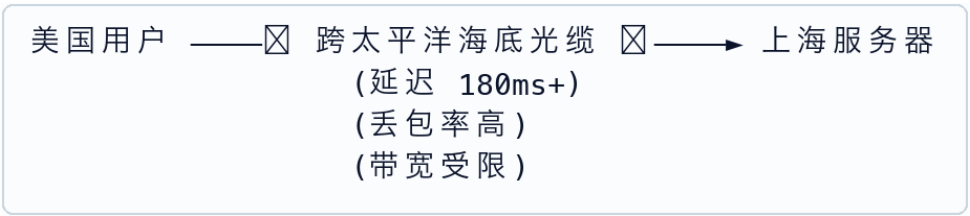
3. **工程上，ABR 算法不是最重要的，bitrate ladder 设置才是。**
4. 短剧/短视频场景 ABR 要配合预载和 TTFF 优化。
5. 档位间距 1.5–2x、足够低的最低档、合理的最高档是黄金三条。

> 本章你会理解：CDN 是什么、为什么视频必须上 CDN、边缘/中转/源站三层结构、签名防盗链、多 CDN 调度、以及一个 VOD 平台的 CDN 账单大概长什么样。

预计阅读时间：20 分钟

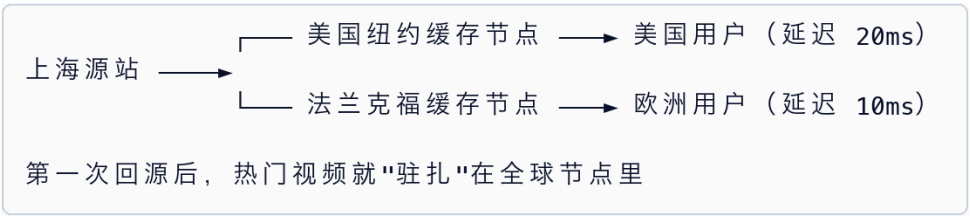
7.1 为什么必须上 CDN

假设你在上海搭了一台服务器放视频，美国用户来看：



每个美国用户每秒从你的上海服务器拉 2 Mbps 视频，10 万美国并发用户 = **200 Gbps 跨国出口**。这在技术和成本上都不可能。

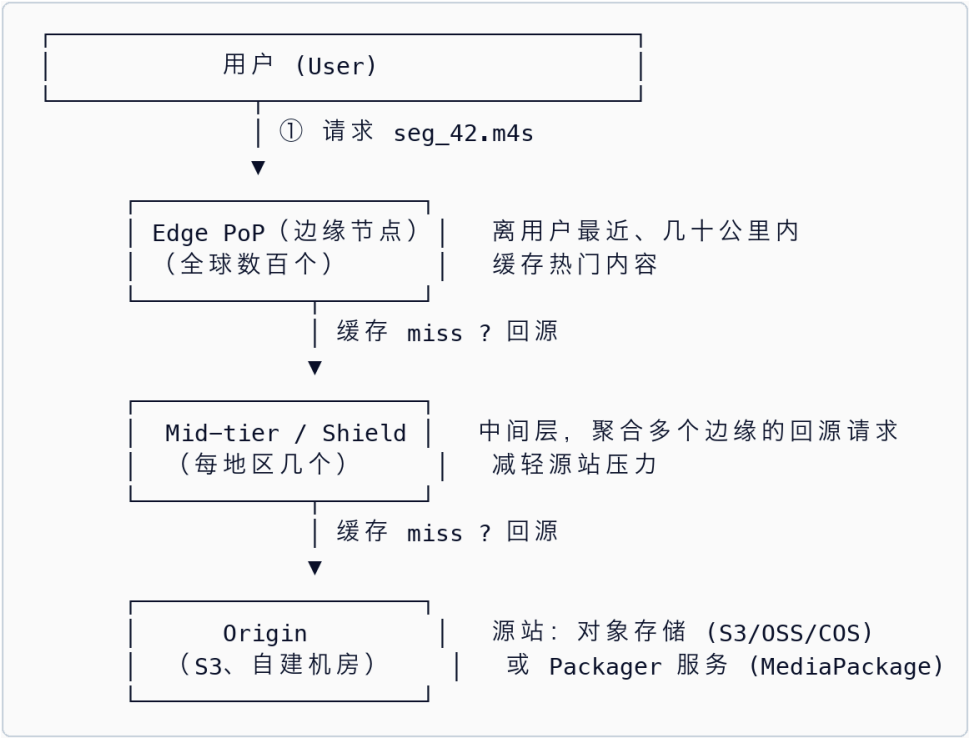
CDN（Content Delivery Network，内容分发网络） 解决方案：



💡 **类比**：CDN 像**全球连锁便利店**。总部在一个地方生产商品（源站），但分销到每个城市的便利店（边缘节点）。用户就近购买，既便宜又快。

7.2 CDN 的三层结构

典型 CDN 架构：



三个概念：

- **Edge**（边缘）：离用户最近的缓存层。用户请求先到这里。
- **Mid-tier / Shield**（中转层）：省钱和稳源。多个边缘 miss 时汇聚到一个 shield 节点，再去源站。
- **Origin**（源站）：真正存所有内容的地方。通常是对象存储（S3、OSS、COS）或一个打包服务。

> 📌 **缓存命中率（Cache Hit Ratio）** 是 CDN 最重要的指标。越高越省钱、越快。目标 **> 95%**。

7.3 回源（Origin Pull）过程

用户请求一个切片的完整流程：

```
1. 用户 → Edge: GET /video/seg_42.m4s
2. Edge 查本地缓存：
   ├── 命中 (HIT) → 直接返回 ☒ 用户拿到切片
   └── 未命中 (MISS)：
       3. Edge → Shield: GET /video/seg_42.m4s
       4. Shield 查缓存：
          ├── 命中 → 回给 Edge → Edge 缓存并回给用户
          └── 未命中：
              5. Shield → Origin: GET /video/seg_42.m4s
              6. Origin 响应 → Shield 缓存 → Edge 缓存 → 用户
```

一个切片的命运：

- 第 1 个用户触发 MISS → 全链路回源 → 慢（多几百 ms）
- 第 2-N 个用户在 Edge 命中 → 快（几十 ms）
- 缓存 TTL 到期后下一个用户再 MISS

这就是为什么"新剧刚上架第一个用户体验最差"，也就是下面要讲的"预热"要解决的问题。

7.4 缓存策略：TTL、Cache-Control

CDN 怎么知道一个文件能缓存多久？看 HTTP 头。

```
HTTP/1.1 200 OK
Content-Type: video/iso.segment
Cache-Control: public, max-age=31536000   ← 1 年
```

VOD 推荐的缓存策略

文件类型	Cache-Control 推荐值	理由
Manifest (.m3u8/.mpd)	`max-age=60` 到 `max-age=300`	可能更新（如加广告、改字幕）
Init segment (.mp4)	`max-age=31536000` (1 年)	不变
Media segment (.m4s/.ts)	`max-age=31536000` (1 年)	不变
字幕 (.vtt)	`max-age=3600` (1 小时)	可能修订
封面/缩略图	`max-age=86400` (1 天)	可能更新

> ⚠️ **常见坑**：如果切片 URL 里带签名参数（`?token=xxx`）没排除在缓存 key 外，**每个用户的 URL 都是独立缓存条目，命中率直接为 0**。详见本章 7.6。

7.5 请求合并（Request Collapsing）

假设 100 个用户同时请求 `seg_42.m4s` 且都 MISS：

没有合并：

```
100 个用户的 MISS 请求 → 100 个回源请求砸到源站 → 源站爆炸
```

有合并（现代 CDN 默认开启）：

```
100 个用户请求 → Edge 认定是同一个对象 → 只发 1 个回源请求 → 响应后返回给 100 个用户
```

这叫 **Request Collapsing / Coalescing**。

大促开播、新剧首播时这是救命功能。

7.6 防盗链：签名 URL / Token

问题：CDN 上的切片 URL 是公开的，有人可能直接嵌入自己网站盗播。

解决方案：**签名 URL / Token**。

原理

视频分发前先签名：

```
CDN 节点校验:
  1. 是否过期?（看 Expires 参数）
  2. 签名是否正确?（HMAC-SHA256 验证）
  3. 参数未被篡改?
  4. 可选：IP 绑定、UA 绑定

任何不符 → 返回 403 Forbidden
```

CloudFront 签名 URL 示例

```
https://d1234.cloudfront.net/video/seg_42.m4s
  ?Expires=1715084800
  &Signature=nitfHRCrtziwO2HwPfWw~yYDhUF5EwRunQA-j19DzZrvDh6hQ73lDx~
-ar3UocvvRQVw__
  &Key-Pair-Id=APKAEIBAERJR2EXAMPLE
```

- `Expires`：过期时间戳（Unix）
- `Signature`：服务端用私钥签的
- `Key-Pair-Id`：对应的公钥 ID

后端签名代码（Python, 用 cryptography 库）：

```python
import time
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import padding
import base64

def sign_url(resource_url, expires_in_sec, private_key_pem, key_pair
        _id):
    expires = int(time.time()) + expires_in_sec
    policy = f'{{"Statement":[{{"Resource":"{resource_url}","Conditi
        on":{{"DateLessThan":{{"AWS:EpochTime":{expires}}}}}}}]}}'

    private_key = serialization.load_pem_private_key(private_key_pe
        m, password=None)
    signature = private_key.sign(policy.encode(), padding.PKCS1v15
        (), hashes.SHA1())
    signature_b64 = base64.b64encode(signature).decode().replace
        ('+','-').replace('=','_').replace('/','~')

    return f"{resource_url}?Expires={expires}&Signature={signature_b
        64}&Key-Pair-Id={key_pair_id}"
```

让签名不影响缓存命中率

签名参数需要在 CDN **排除在 cache key 之外**，或者**由 Edge 计算**：

```
常见做法：
Cache Key = URL path（不包含 query string）
  or
Cache Key = URL path + 一些允许的白名单参数

签名参数（Expires/Signature/Key-Pair-Id）排除。
```

CloudFront、阿里云 CDN、Cloudflare 都有"基于 URL 签名但共享缓存"的配置。

其他防盗链手段

手段	作用
Referer 白名单	只允许从特定域名引用
UA 检查	过滤爬虫和非法 UA
IP 限制	按地域、国家屏蔽
Rate Limit	单 IP 请求频率限制
签名 Cookie	HLS 加载多片段时比签名 URL 更方便

7.7 预热（Pre-warm / Pre-push）

新剧上架时，所有切片都**不在边缘缓存里**。第一波用户全 MISS → 体验差。

预热：CMS 发布后主动让 CDN 去源站拉一份到所有边缘节点。

```
CMS 发布钩子 → 调用 CDN 预热 API
  ├── POST /prewarm { urls: ["…/seg_0001.m4s", ..., "…/seg_0500.m4s"] }
  └── CDN 调度所有边缘节点去回源

几分钟后：全球边缘节点都有了这批切片
第一波用户请求直接 HIT ☒
```

各家 CDN 预热 API：

- AWS CloudFront：Invalidation 是失效，**预热需要自己触发请求**或用 AWS MediaPackage 的 `prefetch`
- 阿里云 CDN：`PushObjectCache`
- 腾讯云 CDN：`PrefetchUrls`
- Cloudflare：企业版 Prefetching

预热不是每个切片都预热，通常预热：

- Init segment
- 前 5–10 个切片（用户看头几秒最关键）
- 所有档位的上述切片

7.8 JIT Packaging vs Pre-packaging

两种打包策略：

Pre-packaging（离线打包）

```
转码完成 → 一次性生成所有 HLS/DASH/CMAF manifest 和 segment → 存 S3 → CDN 回源
```

优点：

- 简单，CDN 完全命中
- 延迟最低

缺点：

- 存储成本高（要存多套格式）
- 变更麻烦（加个字幕要重跑）

JIT Packaging（即时打包）

```
S3 只存 CMAF fMP4 源
用户请求 manifest → Origin 服务实时生成 → 返回 → CDN 缓存
```

优点：

- 存储只存一份
- 灵活（临时加 DRM、换字幕都容易）

缺点：

- 第一次命中稍慢（Origin 要 CPU 计算）
- Origin 服务要扛流量

代表实现：AWS MediaPackage-VOD、Unified Origin、开源 `mp4box`。

> 📌 **推荐**：
>
> - 小库、高频访问内容：Pre-packaging
> - 大库、长尾内容多、频繁变动：JIT Packaging

7.9 多 CDN 策略

为什么一家 CDN 不够：

- **单点故障**：CDN 挂了你全网挂
- **地域覆盖**：没有一家 CDN 在全球每个地方都最好
- **议价能力**：多家竞争你能拿到更好的价格
- **性能对比**：实时选最快的那家

调度方式

方式 1：DNS 调度

方式 2：客户端探测 + 中心调度服务

```
APP 启动时：
  并发测试三家 CDN 的 TTFB 和丢包率
  → 选最快的那家

每 5 分钟重测一次
```

方式 3：专业调度服务

- Cedexis / Conviva Traffic Steering
- NS1、AWS Route 53 Latency-based Routing
- 自研调度中心

故障切换

播放器内置：

```
if (CDN A 连续失败 3 次):
    切换到 CDN B
    上报告警
```

7.10 HTTP 协议：HTTP/2、HTTP/3、QUIC 和视频

HTTP/1.1

- 默认协议
- **队头阻塞（HOL Blocking）**：一个 TCP 连接上一个请求慢，后面全部堵住
- 视频加载慢主要瓶颈之一

HTTP/2

- **多路复用（Multiplexing）**：一个 TCP 连接上并发多个请求
- 头部压缩、Server Push
- **视频场景受益明显**（同时下多个切片）

HTTP/3 / QUIC

- 基于 **UDP**（不是 TCP）
- **消除 TCP 层队头阻塞**（UDP 独立包）
- **0-RTT 建连**（已访问过的站点瞬间连上）
- **弱网、高丢包场景性能大幅提升**

> 📌 **VOD / 直播启用 HTTP/3 后**：移动网络、跨国、高丢包场景的起播时间和卡顿率会显著改善。主流 CDN（Cloudflare、CloudFront、阿里云）均已支持。

7.11 成本估算：一个 VOD 平台的 CDN 账单

常见带宽单价（Tier 3–4 折扣后参考）

区域	$/GB
北美 / 欧洲	0.005—0.010
拉美	0.020—0.050
中东	0.050—0.080
亚太 / 印度	0.015—0.040

快速估算

```
假设：
  DAU = 100 万
  日均观看时长 = 30 分钟
  平均码率 = 1.2 Mbps（720p H.264）

单用户日流量 = 1.2 Mbps × 1800s ÷ 8 ≈ 270 MB
DAU 日总流量 = 100 万 × 270 MB = 270 TB
月总流量 = 270 TB × 30 = 8.1 PB

按 $0.02/GB（加权平均）：
  月 CDN 成本 ≈ 8.1 PB × 1000 × $0.02 = $162,000
```

8 PB 算还好。同 DAU 但观看 60 分钟 + 1080p 1080p 3 Mbps，月成本翻 5 倍到 $81 万。

省钱手段

1. **编码优化**：HEVC 比 H.264 省 37%、AV1 再省 20–30% → **直接减账单**
2. **Per-title encoding**：为每部片子定制 ladder 省 10–20%
3. **下调默认档**：手机默认 720p 比 1080p 省 40%

4. **多 CDN 竞价**
5. **CDN 阶梯折扣**：月流量 > 1 PB 开始有大幅折扣
6. **私有 CDN**：Netflix Open Connect 把服务器塞到 ISP 机房

7.12 动手：测一下一个视频的 CDN 表现

> 🛠️ **动手试一试**

查 CDN 命中情况

```
curl -I 'https://cdn.example.com/video/seg_42.m4s'
```

看响应头：

```
x-cache: HIT                    ← 命中（CloudFront、Cloudflare）
cf-cache-status: HIT            ← Cloudflare 专用
age: 120                        ← 缓存了 120 秒
```

或者 MISS 时：

```
x-cache: MISS
```

查 CDN 节点

```
dig +short video.example.com
curl -I 'https://video.example.com/seg.m4s' | grep -i 'server\|via\|
        x-'
```

会看到类似 `server: cloudfront`、`via: 1.1 abcd.cloudfront.net`。

压测一下边缘命中率

用 `ab` 或 `wrk` 对同一个切片发 1000 请求，看命中率。第一个 MISS，后面应该全部 HIT。

📌 本章要点回顾

1. CDN = 把内容缓存到离用户近的"便利店"。
2. 三层结构：Edge → Shield → Origin。
3. **缓存命中率 > 95%** 是目标，签名参数一定要排除在 cache key 外。
4. **预热新剧** 避免第一波用户全 MISS。
5. **JIT Packaging** 省存储但要 Origin 扛；**Pre-packaging** 简单但费存储。
6. **多 CDN** 提升稳定性、议价能力、地域覆盖。
7. **HTTP/3** 是弱网场景大礼包，尽快切。
8. VOD 带宽成本主导：**编码效率 + 档位设置 + 多 CDN 竞价** 是三大杠杆。

第四部分 · 进阶篇：DRM/播放器/QoE

> 本章你会理解：Widevine / FairPlay / PlayReady 是什么、一次加密三家通吃是怎么做到的、License 流程、L1/L2/L3 级别为什么决定你能不能看 4K、短视频/短剧常用的轻量保护手段。

预计阅读时间：22 分钟

8.1 为什么视频需要"加密 + 许可"

假设你花 $50 买下 Netflix 一个月。Netflix 要保证：

- **只有你**可以看（不能复制给朋友）
- **只能在有限时间内**看（月费到期就不能看）
- **只能在授权设备上**看（不能导出到一个"播放器外壳"疯狂传播）
- **高清 4K 只给安全设备**（防止有人破解后把 4K 视频泄露全网）

这就是 **DRM（Digital Rights Management，数字版权管理）** 要做的事。

DRM 的核心思路：

1. **加密视频内容**（用一个密钥 CEK 加密每个切片）
2. **密钥分发给播放器**要经过严格校验（你是谁？订阅了吗？设备靠谱吗？）
3. **密钥永远不暴露给 APP 层**，只存在硬件安全区内

> 💡 **类比**：视频就像一本加密的书。书本身（加密切片）可以随便复制，但**读书用的钥匙**（CEK）发放时要检查证件，且钥匙只在"密室"（硬件安全区 TEE）里使用，永远不会出门。

8.2 区分三个"都叫加密"的东西

新手最容易混淆：

名称	能叫 DRM 吗	强度
HTTPS / TLS	❌ 只是传输加密	抓包能绕过
HLS AES-128	❌ 轻量加密	key URL 泄漏即失效
HLS SAMPLE-AES + FairPlay	✅ FairPlay DRM	强
CMAF + CENC + Widevine/FairPlay/PlayReady	✅ **真正的 DRM**	强

下面分别讲。

8.3 HLS AES-128（轻量加密）

这是 HLS 规范里自带的简单加密机制：

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:6
#EXT-X-KEY:METHOD=AES-128,URI="https://api.example.com/key?ep=123",I
V=0x1234...

#EXTINF:6.000,
seg_00001.ts
#EXTINF:6.000,
seg_00002.ts
...
```

工作流程：

1. 切片用 AES-128-CBC 全量加密
2. Key URL 里下发 16 字节 AES 密钥
3. 播放器解密后播放

优点：

- 简单，任何播放器基本都支持
- 不需要接 DRM License Server

缺点：

- **Key 以明文形式在 JS 层可见**（Web 端）
- 抓包就能拿到 key
- **不能阻止录屏、不能阻止用户用抓包工具下载**

适合场景：

- 防"非付费用户直接盗链"
- 配合 Signed URL + Token 使用，增加门槛

不适合：

- Netflix 这种有真金白银版权的大厂
- 对防盗录有需求的高价值内容

8.4 "真 DRM" 三巨头

DRM	厂商	覆盖平台
Widevine	Google	Android、ChromeOS、Chrome、Firefox、Edge、大部分智能电视
FairPlay Streaming (FPS)	Apple	iOS、iPadOS、macOS (Safari)、tvOS

DRM	厂商	覆盖平台
PlayReady	Microsoft	Windows、Xbox、Edge、部分智能电视

问题来了：同一部电影要给 iPhone 和 Android 同时看，需要两套加密、两套 key，还要在两个 License Server 上配?

答案是：**不用**。有个东西叫 **CENC** 解决了这个问题。

8.5 CENC：一次加密、三家通吃

CENC（Common **En**cryption），ISO/IEC 23001-7 国际标准。

核心贡献：**定义统一的加密格式，让 Widevine、FairPlay、PlayReady 的播放器都能解同一份加密文件**。

两种 CENC 模式

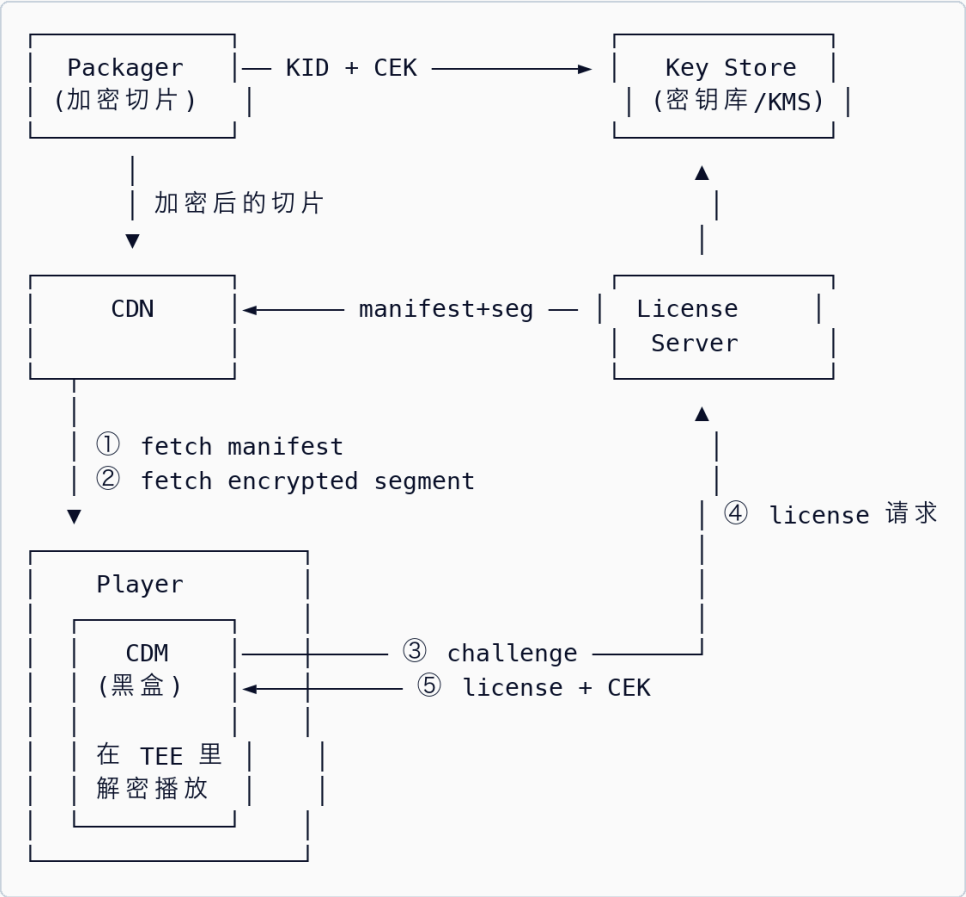
模式	算法	特点	DRM 支持
`cenc`	AES-128 **CTR**	精确加密	Widevine、PlayReady（经典）
`cbcs`	AES-128 **CBC** + Pattern	带 pattern 加密、硬件解码友好	**FairPlay（必须）**、Widevine（现代）、PlayReady（现代）

结论：用 `cbcs` 模式一次加密，Widevine + FairPlay + PlayReady 三家通吃 ✅ 。

这就是现代 CMAF + CBCS 的黄金组合。

8.6 DRM 的完整工作流

一次带 DRM 的播放过程：



关键概念：

- **KID**（Key ID）：每个切片对应一个 key id
- **CEK**（Content Encryption Key）：加密切片的 16 字节 AES 密钥
- **CDM**（Content Decryption Module）：播放器里的 DRM 组件，操作在硬件安全区（TEE）中

- **License**：服务器下发给 CDM 的东西，里面包含加密后的 CEK + 使用规则

逐步解释

1. **打包阶段**：Packager 从 Key Store 要 KID 和 CEK，加密每个切片，把 KID 和 PSSH（Protection System Specific Header）写进 manifest。
2. **播放阶段**：
   - Player 加载 manifest → 发现有 DRM → 初始化 CDM
   - CDM 生成一个 **challenge**（请求消息，包含设备指纹、公钥等）
   - Player 把 challenge 发给 License Server（业务接口）
   - License Server 校验："这个用户订了吗？这个设备安全级别够吗？"
   - 合格 → 返回 license（里面的 CEK 用 CDM 的公钥加密）
   - CDM 在 TEE 里解密 license → 用 CEK 解密切片 → 直接给 GPU 渲染

CEK 全程不进内存、不进 JavaScript。这是 DRM 的根本安全保证。

8.7 Widevine 的 L1 / L2 / L3

Widevine 把设备分三个**安全级别**：

等级	视频解密位置	解码位置	安全性	最高画质
L1	TEE（硬件安全区）	TEE	最高 ⭐⭐⭐	4K / HDR
L2	TEE	软件	中 ⭐⭐	1080p 上限
L3	软件（app 层）	软件	最低 ⭐	480p / 720p 上限

> 📌 **为什么 4K 只给 L1?**
>
> Netflix 4K 流有 $1M+ 的版权价值。如果 L3 设备能解 4K，攻击者用虚拟机 + 逆向就能把 4K 源保存下来泄露到盗版网站。
>
> Netflix、Disney+ 在 License Server 校验 `securityLevel >= L1` 才下发 4K 的 key。

如何检查设备 Widevine 级别?

- Android：用 `drm_info` APP 查看
- Chrome：`chrome://settings/content/protectedContent` → 看证书

很多主流 Android 旗舰（三星 S 系、小米 Pro 版、Pixel）是 L1；中低端很多是 L3。

FairPlay 没有等级

Apple 不像 Widevine 分级。所有苹果设备统一用 Secure Enclave 保护，实质等同于 L1。

PlayReady 有 SL150 / SL2000 / SL3000

类似 Widevine 的分级：

- **SL150**：软件级别，对应 L3
- **SL2000**：主流硬件级别，对应 L2
- **SL3000**：最高硬件级别，对应 L1

8.8 HDCP：你的 HDMI 线也被检查了

HDCP（High-bandwidth Digital Content Protection）：HDMI/DisplayPort 链路上的保护协议。

当你用 iPad 外接 HDMI 到电视放 Netflix，这条链路要协商 HDCP。如果：

- 你的 HDMI 线太老（只支持 HDCP 1.4）→ Netflix 可能只给 1080p
- 线和显示器都支持 HDCP 2.2 → 可以给 4K

License Server 常强制 `requireHdcp: "2.2"`，否则拒绝下发 key。

> ⚠️ **一个常见坑**：测试环境用 HDMI 采集卡录视频发现"黑屏"——因为采集卡不支持 HDCP 合规协议，link 被切断。**这是 DRM 在起作用**。

8.9 License Server 在做什么

License Server 是业务自己搭（或买托管）。收到 challenge 后：

```
def handle_license_request(challenge, user_jwt):
    # 1. 验证 JWT（用户身份、订阅、剧集权限）
    user = verify_jwt(user_jwt)
    if not user.subscribed:
        return 403

    # 2. 识别 challenge 是哪种 DRM（Widevine/FairPlay/PlayReady）
    drm_type = detect_drm_type(challenge)

    # 3. 校验设备安全级别
    security_level = extract_security_level(challenge)
    if video_is_4k and security_level < L1:
        return 403  # 4K 不给不安全设备

    # 4. 找到 CEK（按 KID）
    kid = extract_kid(challenge)
    cek = key_store.get(kid)

    # 5. 调用 DRM 厂商的 SDK 生成 license
    license_blob = drm_sdk[drm_type].generate_license(
        challenge=challenge,
        cek=cek,
        policy={
            "expires_in": 86400,        # 1 天有效
            "hdcp_required": "2.2",     # HDMI 强制 HDCP 2.2
            "allow_offline": False,     # 不允许下载
            "output_protection": True,
        }
    )

    return license_blob
```

常见托管 SaaS：

- EZDRM
- PallyCon
- Irdeto

- BuyDRM KeyOS
- Axinom
- VdoCipher

价格：按 license 请求计费，每 1000 次约 $0.1–1。

8.10 SPEKE：Packager 和 Key Server 的对话协议

SPEKE（Secure Packager and Encoder Key Exchange）是 AWS 推出的标准接口，让 Packager（如 MediaPackage、Shaka Packager）用统一的方式跟不同 DRM 厂商的 Key Server 对话。

接入 SPEKE 后：

- MediaPackage 配一个 URL → 指向 EZDRM/PallyCon 的 SPEKE endpoint
- 打包时 Packager 自动向 Key Server 请求 KID + CEK
- Key Server 返回，Packager 自动生成带 DRM 的 manifest

业界接入 DRM 的事实标准。

8.11 离线播放（Download to Go）

"下载到本地离线看"也要通过 DRM：

- 客户端下载加密切片到本地存储
- License Server 下发一个 **persistent license**（带"离线有效 48 小时"这种条款）
- CDM 把 license 存在硬件安全区
- 离线播放时用存着的 license 解密

Widevine 的 offline license、FairPlay 的 persistent license 都支持。

> ⚠️ **"下载到手机 SD 卡就能分享吗？"** — 不能。密文切片没有 license 无法播放，license 又绑定设备。

8.12 短视频/短剧的轻量保护策略

短视频/短剧平台的内容：

- 单集价值不高（几毛到几块）
- 用户量大、License 调用开销巨大
- 多是"先看到再说"（不看就流失）

用完整 DRM 反而不划算。常见做法：

```
分三级：

L0（免费试看段）：    明文 HLS + Signed URL
                      （前几集）

L1（中等保护）：      HLS AES-128 + 动态 IV + Signed URL
                      + 客户端 SDK 做 key 派生 + 反抓包
                      （绝大多数付费解锁剧集）

L2（高）：            完整 Multi-DRM（CBCS + Widevine + FairPlay）
                      （顶级独家爆款）
```

常见辅助保护

1. **Signed URL + Token** 绑定用户 / 订单
2. **Key Rotation**：每 N 秒换一次 key，提高破解成本
3. **客户端 SDK 内嵌 key 派生逻辑**：攻击者要逆向 SDK 才能提取
4. **反录屏**：

- iOS：`UIScreen.isCaptured` 检测，AirPlay 强制降清晰度
- Android：`FLAG_SECURE` 禁止录屏截屏

5. **反调试 / 反越狱**：检测 Frida、root、LLDB
6. **动态水印**：左下角小字叠 user_id + 时间戳，泄漏溯源
7. **Private Encryption**：云厂商（BytePlus、阿里云、腾讯云）的私有加密方案，CDN 边缘解密

8.13 选择建议

```
你的内容是什么？
│
├── 好莱坞大片 / 独家精品长视频
│     → 完整 Multi-DRM (CBCS + Widevine L1 + FairPlay + PlayReady SL3000)
│     → License Server 强制 HDCP 2.2、securityLevel=L1（4K）
│
├── 普通 VOD 视频（B 级电影、纪录片、教学）
│     → Widevine L3 + FairPlay + PlayReady（硬件等级可放宽）
│
├── 付费短剧 / 中低价值付费视频
│     → HLS AES-128 + Signed URL + 反录屏
│     → 客户端 SDK 做 key 派生
│
├── 用户付费订阅下的自制内容
│     → 按版权方要求灵活（多半 DRM + 水印）
│
└── 免费视频 / UGC
      → 不加密，用 Signed URL 防盗链即可
```

8.14 动手：用 Shaka Packager 做 DRM 打包

🛠️ **动手试一试**。Shaka Packager 是 Google 开源的 DRM 打包工具。

生成 Widevine + FairPlay（CBCS 模式）

```
packager \
  in=video.mp4,stream=video,init_segment=v/init.mp4,segment_template
        =v/seg_$Number$.m4s \
  in=audio.mp4,stream=audio,init_segment=a/init.mp4,segment_template
        =a/seg_$Number$.m4s \
  --protection_scheme cbcs \
  --enable_raw_key_encryption \
  --keys label=VIDEO:key_id=31323334353637383930313233343536:key=323
        33435363738393031323334353637,label=AUDIO:key_id=3132333435
        36373839303132333435 36:key=32333435363738393031323334353637
        \
  --protection_systems Widevine,FairPlay \
  --hls_master_playlist_output master.m3u8 \
  --mpd_output manifest.mpd
```

生产环境不会硬写 key，会接 **SPEKE endpoint**：

```
packager \
  ... \
  --key_server_url https://speke.ezdrm.com/v2 \
  --content_id 'my-video-123' \
  --protection_systems Widevine,FairPlay,PlayReady \
  ...
```

📌 本章要点回顾

1. DRM = 内容加密 + 严格 key 分发 + 硬件级解密保护。
2. **HLS AES-128 不是真 DRM**，只是轻量加密。
3. **三大 DRM**：Widevine（Google）、FairPlay（Apple）、PlayReady（微软）。
4. **CENC（CBCS 模式）** 让一份 CMAF 文件三家 DRM 通吃。

5. **Widevine L1 = 4K 通行证**；L3 最多给到 720p。
6. **HDCP 2.2** 是 4K 外接显示器的前置条件。
7. 短剧/短视频一般用 **轻量加密 + 反录屏**，而不是真 DRM。
8. 生产环境用 **SPEKE** 接 EZDRM / PallyCon 等托管服务。

> 本章你会理解：播放器点击"播放"到画面出现都做了什么、Web/iOS/Android 播放器各有什么限制、首帧优化的招数、自研和开源怎么选。

预计阅读时间：20 分钟

9.1 播放器到底在做什么

你点击"播放"后，播放器内部要完成至少 10 件事：

```
①  解析 manifest (m3u8/mpd)
②  决定从哪个档位开始
③  下载 init segment + 第一个 media segment
④  解析 fMP4 box 结构
⑤  分离 video ES + audio ES
⑥  送入解码器（硬解或软解）
⑦  解码出 YUV 图像 + PCM 音频
⑧  音视频同步（lipsync）
⑨  YUV → RGB 色彩转换
⑩  送到显示器渲染
```

同时还要：

- 持续下载后面的切片
- 运行 ABR 算法
- 上报 QoE 数据
- 响应用户操作（暂停、seek、调清晰度）

- 处理 DRM challenge

一个现代播放器动辄十几万行代码，不是随便写一个 `<video>` 标签那么简单。

9.2 Web 播放器：`<video>` + MSE + EME

原生 `<video>` 够吗

```
<video src="video.mp4" controls></video>
```

这能播一个 MP4，但**不能播 HLS/DASH/CMAF**：

- `<video>` 只能收一个连续的 MP4 文件
- HLS/DASH 是"一堆切片"，需要 JS 把切片塞进 `<video>` 里

Safari 是个例外——它**原生支持 HLS**（Apple 自家的东西），直接 `<video src="index.m3u8">` 能播。

MSE（Media Source Extensions）

MSE 是 W3C 的 API，允许 JS 动态往 video 里塞字节。

```
const video = document.querySelector('video');
const mediaSource = new MediaSource();
video.src = URL.createObjectURL(mediaSource);

mediaSource.addEventListener('sourceopen', () => {
    const sourceBuffer = mediaSource.addSourceBuffer('video/mp4; cod
        ecs="avc1.64001f,mp4a.40.2"');

    fetch('seg_01.m4s')
        .then(r => r.arrayBuffer())
        .then(buffer => sourceBuffer.appendBuffer(buffer));
});
```

有了 MSE，JS 就能：

- 解析 m3u8/mpd 自己实现
- 按需下载切片
- 塞进 SourceBuffer
- 浏览器负责渲染

hls.js 和 **Shaka Player** 就是基于 MSE 建立的。

EME（Encrypted Media Extensions）

EME 是 W3C 的 DRM API，让 JS 对接浏览器内置的 CDM（Widevine in Chrome、PlayReady in Edge、FairPlay in Safari）。

```
video.addEventListener('encrypted', async (event) => {
    const keySystem = 'com.widevine.alpha';  // or 'com.apple.fps.1_
        0' etc.
    const mediaKeys = await navigator.requestMediaKeySystemAccess(ke
        ySystem, config)
                                     .then(a => a.createMediaKeys());
    video.setMediaKeys(mediaKeys);

    const session = mediaKeys.createSession();
    session.addEventListener('message', async (event) => {
        // event.message 是 CDM 生成的 challenge
        const license = await fetch('/license', { method: 'POST', bo
        dy: event.message })
                              .then(r => r.arrayBuffer());
        session.update(license);
    });
    session.generateRequest('cenc', event.initData);
});
```

主流 Web 播放器开源库

库	主打	维护者	规模
hls.js	HLS	Dailymotion / video-dev	最成熟 HLS JS
Shaka Player	DASH + HLS	Google	功能最全
dash.js	DASH	DASH-IF	DASH 参考实现
Video.js	UI 框架	Brightcove	支持多种后端

推荐：HLS-only 选 **hls.js**；DASH 或混合选 **Shaka Player**。

9.3 iOS 播放器：AVPlayer 一家独大

AVPlayer 是什么

- iOS 系统原生播放器
- **唯一官方方式**播放 FairPlay DRM 内容
- 苹果的 HLS 起源和最强支持

AVPlayer 的限制

限制 1：协议只能用 HLS（不原生支持 DASH）。

限制 2：ABR 是黑盒。只有少量 API 可调：

```
// 限制最大码率
playerItem.preferredPeakBitRate = 2_000_000  // 最多 2 Mbps

// 前向缓冲秒数
playerItem.preferredForwardBufferDuration = 10  // 10 秒
```

无法自定义"选哪档切片"的逻辑。

限制 3：切片下载队列不可见。想预载、想自定义 cache 策略要绕开。

绕开限制：AVAssetResourceLoaderDelegate

要自定义行为需要用 **AVAssetResourceLoaderDelegate** —— 劫持 manifest 和 segment 请求，返回自己准备的字节：

```
class CustomLoader: NSObject, AVAssetResourceLoaderDelegate {
    func resourceLoader(_ resourceLoader: AVAssetResourceLoader,
                        shouldWaitForLoadingOfRequestedResource loadi
        ngRequest: AVAssetResourceLoadingRequest) -> Bool {
        let url = loadingRequest.request.url!

        // 自己下载或从本地预载池拿
        fetchFromOurCache(url) { data in
            loadingRequest.dataRequest?.respond(with: data)
            loadingRequest.finishLoading()
        }
        return true
    }
}
```

工程复杂，但头部 APP（TikTok、短剧 APP）常这么做以实现零 TTFF。

AVQueuePlayer：预载多条队列

```
let queue = AVQueuePlayer()
queue.insert(AVPlayerItem(url: episode1URL), after: nil)
queue.insert(AVPlayerItem(url: episode2URL), after: nil)  // 预载
queue.insert(AVPlayerItem(url: episode3URL), after: nil)  // 预载
```

`AVQueuePlayer` 会自动预载队列里的下一个 item（部分预载，具体由系统决定）。

9.4 Android 播放器：ExoPlayer / Media3

ExoPlayer

Google 官方的开源播放器，已在 Android 生态取代老的 `MediaPlayer`：

- 支持 HLS、DASH、SmoothStreaming、Progressive

- 原生 Widevine DRM（通过 `MediaDrm` API）
- **完全开源可自定义**

Media3

AndroidX Media3 是 ExoPlayer 的"下一代封装"（2022+）：

```
implementation "androidx.media3:media3-exoplayer:1.3.0"
implementation "androidx.media3:media3-exoplayer-hls:1.3.0"
implementation "androidx.media3:media3-exoplayer-dash:1.3.0"
```

```
val player = ExoPlayer.Builder(context)
    .setTrackSelector(DefaultTrackSelector(context).apply {
        setParameters(buildUponParameters().setMaxVideoBitrate(2_000
        _000))
    })
    .setLoadControl(DefaultLoadControl.Builder()
        .setBufferDurationsMs(15_000, 30_000, 1_500, 2_500)
        .build())
    .build()

player.setMediaItem(MediaItem.fromUri("https://.../master.m3u8"))
player.prepare()
player.play()
```

可自定义的东西比 iOS 多得多

- `TrackSelector`：码率选择、分辨率限制
- `LoadControl`：缓冲参数
- `MediaSourceFactory`：自定义下载、CDN 调度
- `RenderersFactory`：自定义渲染（后处理滤镜等）

📌 **跨平台**：想在 iOS 和 Android 一套代码？常见方案 React Native Video（底层 iOS=AVPlayer、Android=ExoPlayer）、Flutter video_player、或自己用 FFmpeg + MediaCodec/VideoToolbox 实现。

9.5 首帧优化：怎么让点击到出画面最快

首帧时间 **TTFF（Time To First Frame）** 是 VOD/短视频最敏感的指标。影响因素：

```
① DNS 解析          ~20–100ms
② TCP 握手          ~30–100ms
③ TLS 握手          ~50–200ms
④ 拉 manifest       ~30–200ms
⑤ 拉 init segment   ~50–100ms
⑥ 拉第 1 个切片     ~100–500ms
⑦ 解码 + 渲染       ~50–200ms
```

累加起来动辄 1–2 秒。怎么压到 300ms 以下?

优化手段清单

手段	节省时间
DNS Prefetch（APP 启动就解析域名）	20–100ms
HTTP/3 + 0-RTT（之前连过直接建连）	50–200ms
Preconnect（提前建 TLS）	50–200ms
Manifest Prefetch（下一集 manifest 提前拉）	200ms
短 GOP（1–2s）+ 短 segment（2s）	1–2 秒（启动不用等 6s）
启动档位用低码率	切片小下载快
Init Segment 本地缓存	50–100ms

手段	节省时间
预载下一集首 3 片	对切集几乎零延迟
硬件解码	解码 ~0ms 开销

短剧 APP 的"零 TTFF"方案

核心思想：

```
当前播放第 N 集：

┌──────────────────────────────────────────────┐
│  N-1 集（保留缓冲 5s）  [万一用户回滑]          │
│  N   集（完整加载）                            │
│  N+1 集（预载 init + 首 3 切片 ≈ 6-12s）       │
│  N+2 集（预取 init + 首 1 切片）               │
└──────────────────────────────────────────────┘
```

9.6 缓冲策略：不让用户看到转圈圈

Buffer（缓冲区） 是已下载但还没播的秒数。经典设计：

```
播放头 ──►  [已播放 → 播放点]  [已下载但未播 → 缓冲]  [还没下]
                              ◄──── Buffer Level ────►
```

三个阈值：

- **Min Buffer**（起播需要的最小缓冲，如 3s）
- **Target Buffer**（理想缓冲，如 30s）
- **Max Buffer**（缓冲上限，如 60s，防止预下太多浪费）

业务场景差异

场景	Min	Target	Max
长电影	3s	30s	120s
短剧	1s	10s	20s
低延迟直播	0.5s	2s	6s

缓冲耗尽（Rebuffer）应对

缓冲归零 → 必须停下等 → 用户看到转圈圈。策略：

- 缓冲 < 安全线 → ABR 下一个切片**强制选最低档**（保命）
- 切换 CDN 节点重试
- 上报事件触发告警

9.7 音视频同步（Lip Sync）

视频帧和音频帧是分别解码的，怎么保持同步？

依据 PTS（Presentation Timestamp）：每帧都有一个"该什么时候显示"的时间戳。

```
视频帧 PTS:  0.000   0.033   0.066   0.100 ...
音频帧 PTS:  0.000   0.021   0.042   0.064 ...
               ▲ 以音频为基准
               ▲ 视频帧播放时间 = 音频当前时间 + 偏差校正
```

大多数播放器用音频作为主时钟（人耳对时间偏差更敏感），视频按音频对齐。

9.8 硬件解码 vs 软件解码

	硬件解码	软件解码
性能	快，能解 4K 60fps	慢，4K 可能吃不消
功耗	低	高
灵活性	受硬件支持限制	任意格式
兼容性问题	某些手机对特殊码流兼容差	稳定

优先用硬解。只有硬解不支持（AV1 在老芯片、不常见编码参数）时再回落到软解。

9.9 自研播放器 vs 开源

方案	适用	成本
直接用开源（hls.js / ExoPlayer / AVPlayer）	99% 的 VOD 平台	低，几人月集成
在开源上做少量定制	有特殊 UI / ABR / 埋点需求	中
深度自研（替换 ExoPlayer 内核、iOS 绕开 AVPlayer）	头部 APP（TikTok、短剧 APP、直播平台）	高，几十人月

> 📌 **不要轻易自研播放器内核**。除非你有明确的性能或体验问题是开源播放器无法满足，且有预算维护。

9.10 播放器必做的埋点（为下一章铺垫）

无论用开源还是自研，下面埋点一定要做：

事件	含义
`video_attempt`	用户触发播放
`video_start`	第一帧渲染
`video_rebuffer_start`	卡顿开始
`video_rebuffer_end`	卡顿结束
`bitrate_change`	切换了码率档位
`video_complete`	播放完成
`video_error`	出错
`video_exit`	用户退出

每事件附带：`video_id`、`user_id`、`cdn`、`network_type`、`device`、`bitrate`、`buffer_level` 等维度。

详见 10–QoE 数据体系。

📌 本章要点回顾

1. Web 播放 HLS/DASH 必须靠 **MSE**；DRM 必须靠 **EME**。
2. iOS 播放 HLS + FairPlay 只能用 **AVPlayer**，ABR 是黑盒。
3. Android **ExoPlayer / Media3** 开源可自定义，灵活度高。
4. **TTFF 优化** 的核心手段：DNS Prefetch + HTTP/3 + 短 GOP + 预载。
5. 缓冲三阈值：Min / Target / Max；按业务场景调整。
6. 音视频同步以音频为主时钟。
7. 尽量用开源播放器，别轻易自研内核。

> 本章你会理解：QoE 和 QoS 的区别、6 个核心 QoE 指标的定义与目标值、数据上报管道怎么搭、怎么从数据里定位问题。

预计阅读时间：18 分钟

10.1 QoE vs QoS：两个经常被混淆的词

缩写	全称	定义	视角
QoS	Quality of Service	网络/服务的客观性能（带宽、丢包、延迟）	运维 / 网络工程师
QoE	Quality of Experience	**用户感知**的体验	产品 / 用户研究

> 💡 QoS 说"我给你 10 Mbps"，QoE 说"用户刷抖音流畅吗"。
>
> 同样的 QoS 下，不同的播放器 / ABR 策略 / 编码参数 → 不同的 QoE。

我们关心的是 QoE。做视频平台不看 QoE 跟运营高速公路不看"堵车时长"一样。

10.2 六个核心 QoE 指标

① Video Startup Time（VST，起播时间）

定义：用户点击播放到第一帧渲染的秒数。

目标：

- 移动端短视频/短剧：**P50 < 300ms、P95 < 800ms**

- 长视频 VOD：**P50 < 1s、P95 < 2s**

最敏感。用户不会忍受 2 秒黑屏。

② Rebuffering Ratio（RBR，卡顿率）

定义：`rebuffer_time / (rebuffer_time + play_time)`。

例：用户看了 60 秒，其中卡顿累计 3 秒 → RBR = 3/(60+3) = 4.8%。

目标：**< 0.5%**（优秀），< 1%（合格）。

影响留存：Conviva 研究表明 RBR 每上升 1%，观看时长下降 2–5%。

③ Video Start Failure（VSF）

定义：用户触发播放但第一帧从未渲染（因为 404、CORS、DRM 错误等）。

目标：**< 1%**。

Conviva 还细分：

- **VSF-T**（Technical）：技术原因失败
- **VSF-B**（Business）：业务原因（用户无权限等），不计入 QoE

④ Exit Before Video Start（EBVS，起播前退出）

定义：用户触发播放但还没看到第一帧就**主动关了**（不是报错，是不想等了）。

目标：**< 3%**。

强相关于 VST。VST 太慢 → EBVS 高 → 留存差。

⑤ Video Playback Failure（VPF）

定义：播放中途报错失败（如解码错误、证书过期、CDN 断流）。

目标：**< 0.5%**。

⑥ Average Bitrate（平均码率）

定义：实际观看时各片段码率的时间加权平均。

用途：衡量用户实际看到的画质档位是否足够。如果 70% 用户平均停在 480p，可能是：

- 网络普遍差
- ABR 算法太保守
- 高档位没转码

辅助指标

指标	说明
Rebuffer Frequency	单位时长内 rebuffer 次数（<0.1 次/分钟）
Bitrate Switching	码率切换次数和幅度（稳定优先）
Video Complete Rate	完播率（业务）
Time to Key Decode	DRM 下发 license 的时间
First Byte Time	第一个字节到达时间

10.3 Conviva 的 SPI：一个综合指标

SPI（Streaming Performance Index）：Conviva 的综合 KPI，表示"**好或很好体验的会话占比**"。

一个 session 算"好"，需要同时满足：

- 没有 VSF-T / VPF-T 错误
- 没有或极低 Rebuffering（CIRR < 某阈值）
- 平均码率达到屏幕尺寸合格线
- Video Start Time 在可接受范围

- 如果用户在退出前等了很久，不算 EBVS

单一指标容易误导（例如 RBR 好但码率极低），SPI 综合反映体验。

10.4 分析维度：多维下钻是灵魂

不能只看"总 RBR 是多少"。要**多维度切片**：

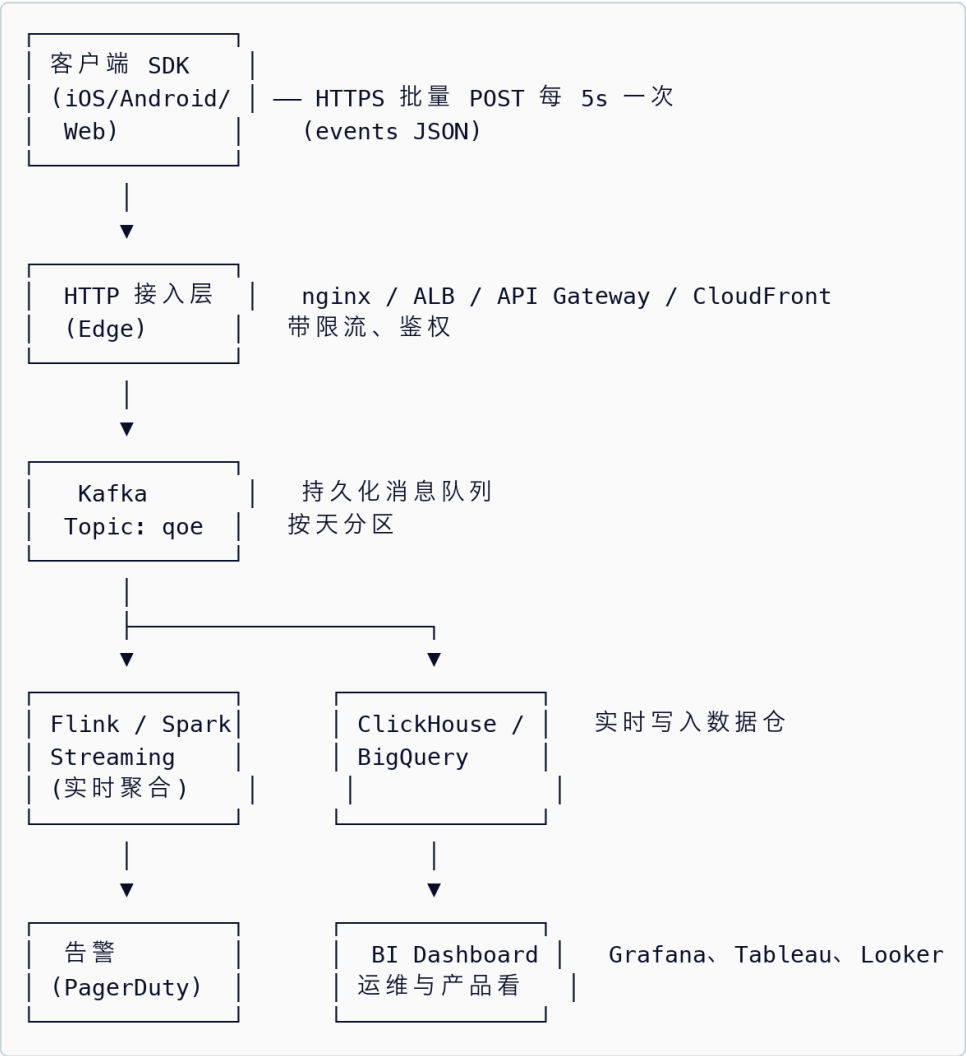
维度	举例
地理	国家 / 省 / 城市 / ISP
设备	OS 版本、机型、芯片、屏幕尺寸
网络	WiFi / 4G / 5G、吞吐区间
CDN	供应商、PoP、上层 Shield
内容	剧集、分辨率档、编码、时长
时间	小时、天、周
用户	新/老、付费/免费、地区

查问题的标准姿势：

```
总 RBR 升到 2% → 不知道原因
按 CDN 下钻 → CDN-A 的 RBR 5%、CDN-B 的 RBR 0.3%
按 CDN-A 地域下钻 → 印度孟买的 RBR 12%
按 CDN-A 孟买时段下钻 → 19:00-22:00 尖峰
结论：CDN-A 在孟买晚高峰出问题 → 切换到 CDN-B
```

10.5 数据上报管道

从客户端到 BI 的典型管道



上报事件的规范

每个事件包含：

```
{
  "event": "video_rebuffer_start",
  "session_id": "uuid-...",
  "user_id": "u-12345",
  "video_id": "ep-789",
  "timestamp": 1715084800123,
  "player_version": "2.3.4",
  "device": {
    "os": "iOS",
    "os_version": "17.4",
    "model": "iPhone 15 Pro"
  },
  "network": {
    "type": "cellular",
    "carrier": "Verizon",
    "effective_type": "4g"
  },
  "cdn": "cloudfront",
  "bitrate": 2500000,
  "buffer_level_sec": 0.8,
  "position_sec": 45.2
}
```

批量 vs 实时

不要每个事件一次 HTTP（10 万 DAU × 100 事件/人 = 一千万请求/天）。

批量策略：客户端累积 10 秒或 50 事件就一次 POST。

10.6 分析套路：几个真实的故障定位案例

案例 1：整体 VST 升高

```
周一早上 9 点，VST P50 从 400ms 升到 1.2s
  |
按 OS 下钻 → Android VST 涨到 2s，iOS 正常
  |
按 APP 版本下钻 → 3.4.5 版本全员 2s，3.4.4 正常
  |
看变更日志 → 3.4.5 引入了新播放器库
  |
紧急热修 / 回滚到 3.4.4
```

案例 2：特定剧集卡顿

```
某新剧第 3 集 RBR 异常高 5%
  |
按 CDN 下钻 → 所有 CDN 都高（不是 CDN 问题）
  |
检查切片 → 发现第 3 集某段切片异常大（20 MB，别的 2 MB）
  |
检查编码日志 → 该集中有 10 秒激烈动作戏，编码器码率瞬间飙高
  |
解决方案：重新转码，设 MaxBitrate 限制峰值
```

案例 3：某地区转化率突降

```
印度新增用户首小时完播率从 30% 跌到 15%
|
按 VST 看 → 印度 VST P50 从 0.8s 升到 3s
|
按 CDN 看 → 印度的 CDN-A 边缘节点延迟升高
|
Ping 测试 → CDN-A 孟买 POP 连续 4 小时延迟 400ms
|
立即切换印度流量到 CDN-B，同时联系 CDN-A 运维
```

10.7 自研 vs 买托管（Mux / Conviva / Datadog RUM）

托管服务

服务	长处
Mux	开发者友好、集成简单、$1.25 / 千次 session
Conviva	企业级、最全面、最贵
Datadog RUM	综合 APM 一体
NPAW（YOUBORA）	欧洲主流

优点：几小时集成好、看板开箱即用、不用维护。 **缺点**：贵、数据在第三方、定制有限。

自研

优点：完全定制、数据可深度关联业务（订单、留存）、规模大时成本优势。
缺点：开发维护成本高、SDK 多端一致性难。

常见组合

- **起步期**：买 Mux，快速获得可用看板
- **规模化**：自研 + 保留 Mux 做 benchmark 对标

10.8 客户端 SDK 的几个关键点

① 不要拖慢播放

QoE SDK 本身的开销不能影响体验：

- 上报用独立低优先级线程
- 网络失败静默重试，不阻塞 UI
- SDK 崩溃不能拉着 APP 一起崩

② 离线补偿

用户断网看完后上线补上报：

- 事件写到本地 sqlite/文件
- 有网后按 FIFO 批量补报

③ 时钟对齐

用户手机时间可能不准：

- 用服务端时间戳（HTTP Date header）作基准
- 事件带相对时间（`delta_ms` 相对于 session 开始）

④ 采样

海量用户时 100% 上报太贵：

- 关键错误事件 100% 上报
- 普通事件按 10%–30% 采样
- 按 user_id hash 保证同用户全采或全不采（便于会话分析）

10.9 看板设计：几个必须有的视图

Dashboard 1：总览大盘

- DAU、播放 session 数
- VST P50/P95
- RBR、VSF、VPF
- SPI（综合分）
- 按国家 Top 10 下钻

Dashboard 2：CDN 健康

- 每个 CDN 的 RBR、VST、错误率
- CDN 比对面板（同时间段对比）
- CDN 边缘节点地图

Dashboard 3：内容质量

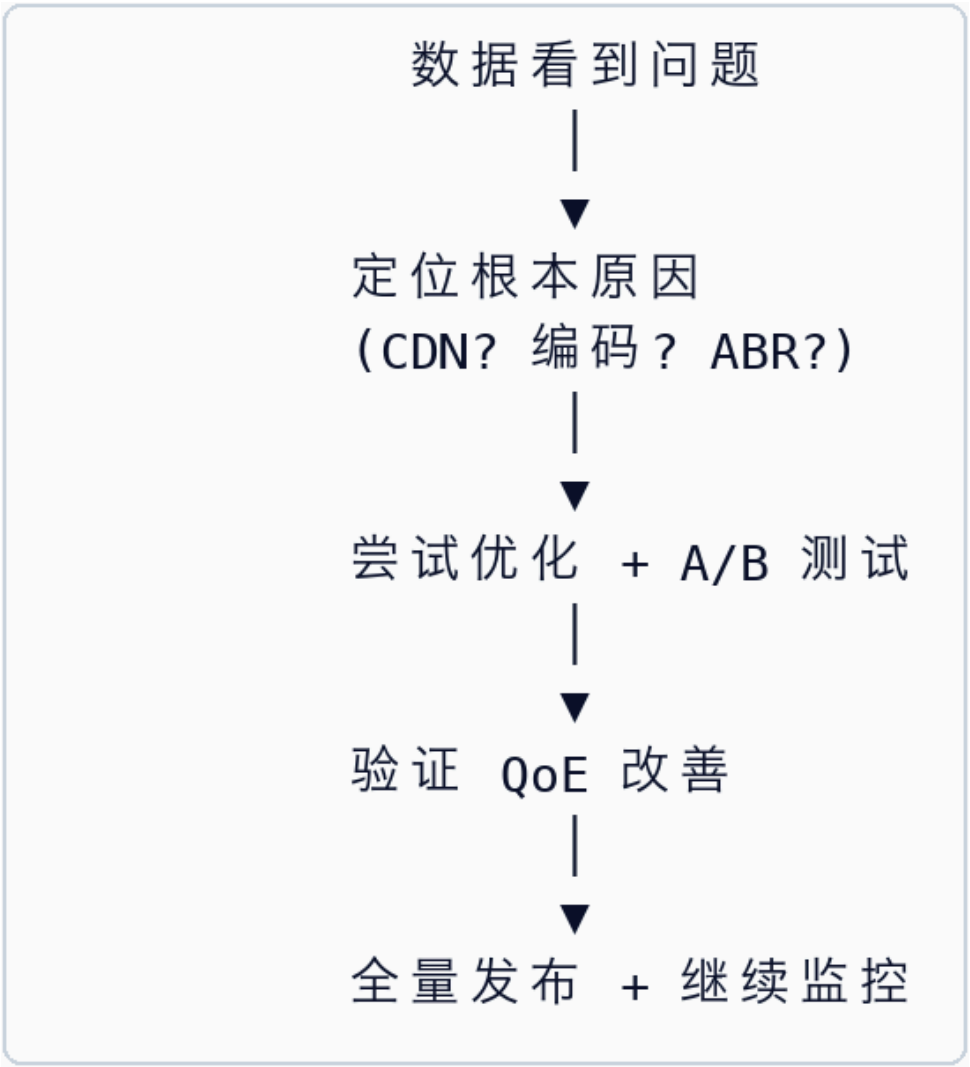
- 新剧上线首 24 小时质量指标
- 按剧集的完播率、RBR
- 异常剧集告警

Dashboard 4：设备与版本

- 按 APP 版本的错误率
- 按 OS 版本的 VST
- 按机型的 RBR

10.10 数据驱动的优化闭环

QoE 数据不是用来"看看就算了"，而是驱动**工程决策**：



每周 review QoE 数据，是所有成熟视频团队的标配。

📌 本章要点回顾

1. QoE 衡量用户感知体验，不是网络指标。

2. 六大核心指标：**VST / RBR / VSF / EBVS / VPF / Avg Bitrate**。
3. Conviva 的 **SPI** 是"好体验会话占比"的综合 KPI。
4. 数据要**按多维度下钻**，单一全局数字无法定位问题。
5. 数据管道标配：**Client SDK → Kafka → Flink/ClickHouse → BI**。
6. 起步期买 Mux / Conviva，规模化再自研。
7. **QoE 数据驱动决策**，每周 review。

第五部分 · 实战篇：工作流/AWS

> 本章你会理解：一个视频从导演把文件交给你到全球用户能看到，完整要经过哪些环节、每步在解决什么问题、用什么工具编排。

预计阅读时间：18 分钟

11.1 VOD 工作流全景

看一眼，感受下复杂度：

每一步都可能出问题。一个成熟平台会在每步有**重试、告警、人工兜底**。

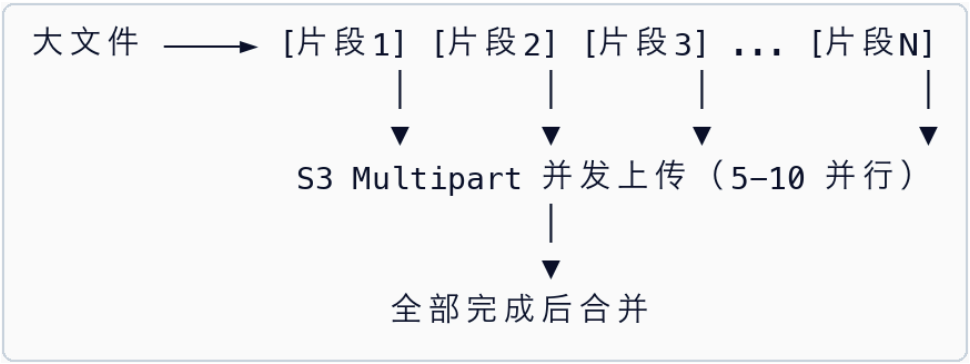
11.2 ① 上传：把大文件稳稳送到云

挑战：

- 母版文件常 5–50 GB（ProRes / DNxHD）
- 网络不稳定（跨国、办公室 WiFi）
- 用户可能在上传中关电脑 / 断网

分片上传（Multipart Upload）

把文件切成 5–100 MB 的片段，并发上传：



好处：

- 并行加速
- 失败只重传单个片段（断点续传）
- S3/OSS/COS 都原生支持

秒传（MD5/CRC 预检查）

上传前客户端算 MD5，发给后端：

```
客户端 → 后端：  "我要上传文件，md5=abc123"
后端  → 客户端：  "这个文件我已经有了，你不用上传"  ← 秒传
                or
                "没有，请开始上传"
```

实战：S3 Multipart 的三步 API

```
import boto3

s3 = boto3.client('s3')

# 1. 发起 multipart upload
resp = s3.create_multipart_upload(Bucket='my-bucket', Key='video.mo
        v')
upload_id = resp['UploadId']

# 2. 上传每个 part
parts = []
for i, chunk in enumerate(chunks, start=1):
    resp = s3.upload_part(
        Bucket='my-bucket', Key='video.mov',
        UploadId=upload_id, PartNumber=i,
        Body=chunk)
    parts.append({'PartNumber': i, 'ETag': resp['ETag']})

# 3. 完成合并
s3.complete_multipart_upload(
    Bucket='my-bucket', Key='video.mov',
    UploadId=upload_id,
    MultipartUpload={'Parts': parts})
```

11.3 ② 入库扫描：内容合规

上传到**"隔离区" bucket**（不是正式存储），过完审核才能进正库。

必做检查

检查	工具
病毒扫描	ClamAV、VirusTotal API

检查	工具
NSFW 检测	AWS Rekognition Content Moderation、阿里云绿网
暴力/血腥/政治敏感	同上
版权指纹	Audible Magic（音乐）、ACRCloud
人脸识别（按需）	AWS Rekognition、自研
字幕合规	关键词过滤、地域适配

人工复审

AI 只能初筛。高价值或不确定的内容进入人工队列：

- 审核员看一遍（抽样或全量）
- 标记合规或不合规
- 记录理由

11.4 ③ 探针校验（Probe）

ffprobe 扫一遍文件，读出元数据：

```
ffprobe -v error -show_format -show_streams -of json input.mov > pro
        be.json
```

获取：

- 分辨率、帧率、编码
- 音轨数、采样率
- 时长
- 是否有变帧率、异常时间戳
- 是否含 HDR 元数据

校验规则：

- 时长 < 30 秒 → 拒绝（不是剧集）
- 分辨率低于 720p → 拒绝（画质不够）
- 帧率异常（不是 23.976/25/29.97/30/50/60）→ 警告
- HDR 但色彩空间不是 BT.2020 → 修正或拒绝

11.5 ④ 转码：核心生产环节

产出物清单

一个源文件转码后通常产出：

```
/vod/ep-001/
  mezz/original.mov              ← 母版备份（冷存）
  v_360p.mp4                     ← H.264 360p
  v_480p.mp4                     ← H.264 480p
  v_720p.mp4                     ← H.264 720p
  v_1080p.mp4                    ← H.264 1080p
  v_720p_hevc.mp4                ← H.265 720p
  v_1080p_hevc.mp4               ← H.265 1080p
  v_720p_av1.mp4                 ← AV1 720p（旗舰机）
  audio_en.mp4                   ← English AAC
  audio_zh.mp4                   ← Chinese AAC
  subs_en.vtt                    ← English subtitle
  subs_zh.vtt                    ← Chinese subtitle
  thumbnails.vtt + sprite.jpg    ← 缩略图预览（进度条悬停缩略图）
```

并行加速

一个剧集可能要转 10+ 个档位。如果串行，等半天。**并行切分有两种**：

① **按档位并行**（简单）：

```
转码集群：
  Worker 1: 源 → 360p H.264
  Worker 2: 源 → 720p H.264
  Worker 3: 源 → 1080p H.264
  Worker 4: 源 → 720p H.265
  ...所有 Worker 同时跑
```

② **按 GOP 切分并行**（复杂但快）：

```
把源文件按 IDR 边界切成 N 段
每段丢给不同 Worker 独立转码
最后拼接 bitstream
```

Netflix / YouTube 在云上动用几百个实例并行转一部电影，一小时完成。

转码服务选择

服务	特点
自建 ffmpeg 集群	控制力最强、成本最低（大规模）、运维麻烦
AWS MediaConvert	按分钟付费、QVBR、HDR/DRM 集成好
阿里云 VOD / 腾讯云 VOD	国内业务接入简单
Bitmovin / Mux	开发者友好、企业级
BytePlus VOD	短剧场景优化好

成本优化

- **Spot / Preemptible 实例** + 断点续转：省 70–80%

- **长尾内容只做 H.264**（节省转码成本 + 存储成本）
- **热门内容追加 H.265/AV1**

11.6 ⑤ 打包与加密

转码后的 MP4 要变成流媒体格式。工具见 05–流媒体协议 和 08–DRM：

```
转码产出的 MP4
        │
        ▼
Shaka Packager / MediaPackage / mp4box
        │
        ▼
产出：
  CMAF fMP4 切片
  HLS master.m3u8 + media.m3u8
  DASH manifest.mpd
  (可选)CENC/CBCS 加密切片 + License 元数据
```

11.7 ⑥–⑦ 发布到存储 + 写元数据

转码/打包产物上传到**正式存储**（常是 S3 的"生产 bucket"）：

```
/vod-production/ep-001/
  init.mp4
  seg_*.m4s
  master.m3u8
  manifest.mpd
  thumbnails/...
```

同时写业务数据库：

```
INSERT INTO episodes (
    episode_id, drama_id, title, duration_sec,
    manifest_url, thumbnail_url,
    status, publish_at, ...
) VALUES (...);
```

写搜索索引（Elasticsearch / Algolia）以便用户搜索。

11.8 ⑧ CDN 预热

见 07–CDN §7。

不要等第一批用户来触发回源。对新剧、爆款剧做主动预热。

11.9 ⑨ 通知与上架

- CMS 里状态从 "processing" → "ready"
- 推送给订阅用户："你追的剧新一集来了"
- 首页推荐位 / 榜单更新
- 广告素材上架（如果是付费剧）

11.10 ⑩ 监控与回归

上线后持续监控：

- 播放成功率
- QoE 指标是否正常
- 反馈到 CMS 的异常（比如某一集特别多卡顿）
- 发现问题 → 重新转码或回滚

11.11 编排工具：把流程串起来

上面 10 步要可靠地一步接一步执行，还要处理**重试、并行、失败分支**。不能用一个大脚本硬编码。

常用编排工具

工具	特点	适合
AWS Step Functions	Serverless、可视化、紧密集成 AWS	AWS 原生，官方 VOD 方案默认
Temporal	分布式工作流、强一致、类型安全	自研控制面 / 跨云
Airflow	批量 ETL、定时任务	数据处理，VOD 里用于离线分析
Argo Workflows	Kubernetes 原生	K8s 团队
自研（Kafka 驱动）	事件驱动，每步 publish 下一事件	追求极致定制

Step Functions 示例（简化）

```
{
  "StartAt": "Probe",
  "States": {
    "Probe": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:...:ProbeVideo",
      "Next": "ParallelTranscode"
    },
    "ParallelTranscode": {
      "Type": "Parallel",
      "Branches": [
        {"StartAt": "Transcode360p", "States": {
          "Transcode360p": {"Type": "Task", "Resource": "arn:aws:med
        iaconvert:...", "End": true}}},
        {"StartAt": "Transcode720p", "States": {
          "Transcode720p": {"Type": "Task", "Resource": "arn:aws:med
        iaconvert:...", "End": true}}},
        {"StartAt": "Transcode1080p", "States": {
          "Transcode1080p": {"Type": "Task", "Resource": "arn:aws:me
        diaconvert:...", "End": true}}}
      ],
      "Next": "Package"
    },
    "Package": {"Type": "Task", "Resource": "...Package...", "Next":
        "Prewarm"},
    "Prewarm": {"Type": "Task", "Resource": "...Prewarm...", "Next":
        "Notify"},
    "Notify": {"Type": "Task", "Resource": "...Notify...", "End": tr
        ue}
  }
}
```

Step Functions 自带重试、超时、分支、可视化。

11.12 灾备与回滚

多地区备份

- 源文件（母版）跨 region 备份（S3 Cross-Region Replication）
- 热门内容在多 region 有副本（就近分发 + 容灾）

回滚场景

- 新剧上线后发现内容问题 → 一键下架 + CDN 失效缓存
- 转码版本有 bug → 切回上个版本的 manifest

数据备份

- 元数据 DB 每日备份
- 用户观看进度（存大量）→ 分层存储：热数据在 Redis，冷数据归档到 S3 Glacier

11.13 一个典型的工作流时间线

一部 60 分钟的电影从上传到上线：

```
T+0min    导演上传 mezz 文件到隔离 bucket
T+5min    上传完成 → 触发 Step Functions
T+7min    NSFW + 病毒扫描完成
T+8min    ffprobe 校验通过
T+10min   启动并行转码（H.264 x 4档 + H.265 x 2档 + AV1 x 1档）
T+70min   全部转码完成（60 分钟电影约需 1x 实时，并行多档）
T+72min   Packager 打包 HLS+DASH+DRM
T+73min   上传到生产 S3
T+74min   CDN 预热完成（北美/欧洲/亚太）
T+75min   写元数据、ES 索引、通知
T+75min   上线 ✅
```

短剧 APP 场景因为单集 60–90 秒，端到端在几分钟内完成。

📌 本章要点回顾

1. VOD 工作流 10 大步，每步都可能出问题，要有重试和告警。
2. **上传**用 Multipart + MD5 秒传；**存储**用隔离 bucket + 扫描。
3. **ffprobe 探针**是转码前必做。
4. **转码**占整个流程最长时间，靠并行 + Spot 实例降本。
5. **Packager + DRM** 是发布前的最后一步。
6. **CDN 预热**对新剧上架至关重要。
7. **Step Functions / Temporal** 是常见编排工具。
8. 做好**灾备、回滚**的预案。

> 本章你会看到：AWS 上搭一个 VOD 平台的完整服务清单、架构图、关键 API 调用、其他云的等价服务、真实的成本量级。

预计阅读时间：20 分钟

12.1 AWS 媒体服务家族

AWS 把 VOD/直播相关服务叫 **AWS Elemental**。核心四件套：

服务	干什么
AWS Elemental MediaConvert	文件转码（离线、VOD）
AWS Elemental MediaLive	实时编码（直播）
AWS Elemental MediaPackage	流媒体打包（HLS/DASH/CMAF）、DRM

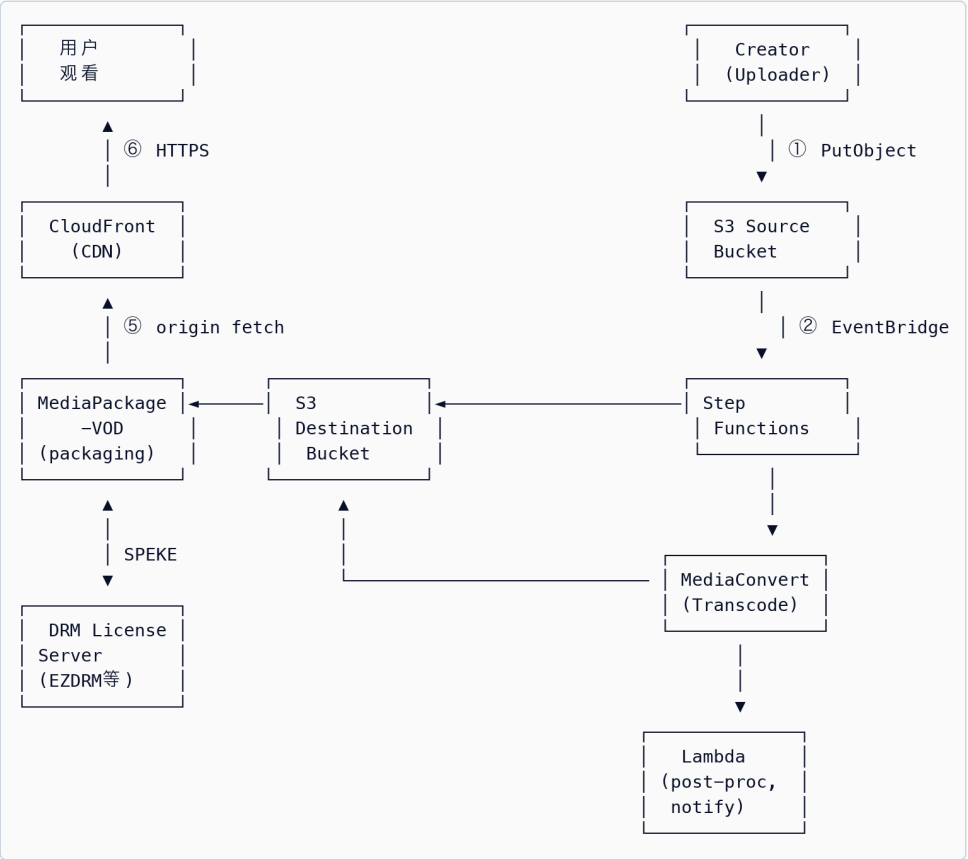
服务	干什么
AWS Elemental MediaStore / MediaTailor	低延迟 origin / 广告插入

配合基础服务：

服务	干什么
Amazon S3	存储
Amazon CloudFront	CDN
AWS Lambda	触发器、轻量处理
AWS Step Functions	工作流编排
Amazon DynamoDB	资产元数据存储
Amazon SNS/SQS	事件通知、异步队列
Amazon CloudWatch	监控告警

12.2 官方参考架构

AWS 官方的 *Video on Demand on AWS* 方案架构：



流程：

1. Creator 上传文件到 S3 Source Bucket
2. S3 Event → Step Functions 被触发
3. Lambda 先 probe + 验证
4. Step Functions 调用 MediaConvert 转码
5. 转码结果写到 S3 Destination Bucket
6. MediaPackage-VOD 打包 HLS/DASH + DRM
7. CloudFront 作为 CDN 分发
8. 完成后 Lambda 通知用户

12.3 一步步走一遍

步骤 1：准备 S3 Bucket

```
# Source bucket（上传原片）
aws s3 mb s3://my-vod-source --region us-east-1

# Destination bucket（转码产物）
aws s3 mb s3://my-vod-dest --region us-east-1

# 启用 EventBridge 通知
aws s3api put-bucket-notification-configuration \
  --bucket my-vod-source \
  --notification-configuration '{
    "EventBridgeConfiguration": {}
  }'
```

步骤 2：创建 IAM Role 给 MediaConvert 用

MediaConvert 需要从 S3 读、往 S3 写：

```
aws iam create-role --role-name MediaConvertRole \
  --assume-role-policy-document file://trust-policy.json

aws iam attach-role-policy --role-name MediaConvertRole \
  --policy-arn arn:aws:iam::aws:policy/AmazonS3FullAccess
```

步骤 3：MediaConvert Job 模板（最小可用）

```
{
  "Role": "arn:aws:iam::123456789012:role/MediaConvertRole",
  "Settings": {
    "Inputs": [{
      "FileInput": "s3://my-vod-source/episode01.mov",
      "AudioSelectors": {"Audio Selector 1": {"DefaultSelection": "D
        EFAULT"}}
    }],
    "OutputGroups": [{
      "Name": "CMAF",
      "OutputGroupSettings": {
        "Type": "CMAF_GROUP_SETTINGS",
        "CmafGroupSettings": {
          "Destination": "s3://my-vod-dest/ep01/",
          "SegmentLength": 4,
          "FragmentLength": 2,
          "WriteHlsManifest": "ENABLED",
          "WriteDashManifest": "ENABLED"
        }
      },
      "Outputs": [
        {
          "NameModifier": "_1080p",
          "VideoDescription": {
            "Width": 1920, "Height": 1080,
            "CodecSettings": {
              "Codec": "H_264",
              "H264Settings": {
                "RateControlMode": "QVBR",
                "QvbrSettings": {"QvbrQualityLevel": 8},
                "MaxBitrate": 5000000,
                "GopSize": 2, "GopSizeUnits": "SECONDS"
              }
            }
          },
          "AudioDescriptions": [{
            "CodecSettings": {"Codec": "AAC", "AacSettings": {
```

```
            "Bitrate": 128000, "SampleRate": 48000, "CodingMode":
        "CODING_MODE_2_0"
            }}
          }]
        },
        { "NameModifier": "_720p",  /* ... 2500 kbps ... */ },
        { "NameModifier": "_540p",  /* ... 1200 kbps ... */ },
        { "NameModifier": "_360p",  /* ... 600 kbps ... */ }
      ]
    }]
  }
}
```

关键参数：

- `QvbrQualityLevel: 8`：QVBR 目标质量（1–10，越高越好）
- `MaxBitrate: 5000000`：峰值限制
- `GopSize: 2 SECONDS`：2 秒一个 GOP
- `SegmentLength: 4`：4 秒切片
- `FragmentLength: 2`：2 秒 fragment（子切片，for CMAF）

提交 Job：

```
aws mediaconvert create-job --cli-input-json file://job.json \
  --endpoint-url https://abc.mediaconvert.us-east-1.amazonaws.com
```

步骤 4：加 DRM (SPEKE)

在 CMAF OutputGroup 里加：

```
"Encryption": {
  "EncryptionMethod": "SAMPLE_AES",
  "SpekeKeyProvider": {
    "ResourceId": "episode01",
    "SystemIds": [
      "edef8ba9-79d6-4ace-a3c8-27dcd51d21ed",    // Widevine
      "94ce86fb-07ff-4f43-adb8-93d2fa968ca2",    // FairPlay
      "9a04f079-9840-4286-ab92-e65be0885f95"     // PlayReady
    ],
    "Url": "https://speke.ezdrm.com/v2",
    "CertificateArn": "arn:aws:acm:..."
  }
}
```

`SPEKE URL` 指向 DRM 厂商（EZDRM/PallyCon等）的 endpoint。

步骤 5：CloudFront 分发

创建一个 distribution 指向 S3 Destination Bucket：

```
aws cloudfront create-distribution --distribution-config file://cf-c
        onfig.json
```

关键配置：

- **Origin**: S3 Destination Bucket
- **Viewer Protocol Policy**: Redirect HTTP to HTTPS
- **Allowed Methods**: GET, HEAD
- **Cache Policy**: `Managed-CachingOptimizedForUncompressedObjects`
- **Origin Access Control**: 启用，防止绕过 CloudFront 直接访问 S3
- **Signed URLs**: 需要时启用 Trusted Key Groups

步骤 6：Signed URL 生成

客户端请求播放时后端生成短时效签名 URL：

```python
from datetime import datetime, timedelta
import boto3

cf_signer = boto3.client('cloudfront-signer', ...)  # 或用 rsa 库自签

def generate_signed_url(video_path, user_id, expires_in=3600):
    url = f"https://d1234.cloudfront.net/{video_path}"
    expires = datetime.utcnow() + timedelta(seconds=expires_in)
    # 用 CloudFront key-pair 签名
    signed = cf_signer.generate_presigned_url(url, date_less_than=ex
        pires)
    return signed
```

步骤 7：Step Functions 编排

状态机串起整个流程（见 11 章 §11.11）。

12.4 Terraform 一键部署片段

用 Terraform 声明基础设施：

```
# Source bucket
resource "aws_s3_bucket" "source" {
  bucket = "my-vod-source"
}

# Destination bucket
resource "aws_s3_bucket" "dest" {
  bucket = "my-vod-dest"
}

# EventBridge rule to trigger Step Functions on S3 upload
resource "aws_cloudwatch_event_rule" "upload" {
  name = "vod-upload-trigger"
  event_pattern = jsonencode({
    source = ["aws.s3"]
    "detail-type" = ["Object Created"]
    detail = {
      bucket = { name = [aws_s3_bucket.source.bucket] }
    }
  })
}

# Step Functions state machine
resource "aws_sfn_state_machine" "vod_pipeline" {
  name     = "vod-pipeline"
  role_arn = aws_iam_role.sfn_role.arn
  definition = file("${path.module}/vod_pipeline.asl.json")
}

resource "aws_cloudwatch_event_target" "sfn" {
  rule = aws_cloudwatch_event_rule.upload.name
  arn  = aws_sfn_state_machine.vod_pipeline.arn
  role_arn = aws_iam_role.event_bridge_role.arn
}

# CloudFront distribution
resource "aws_cloudfront_distribution" "vod" {
  origin {
    domain_name = aws_s3_bucket.dest.bucket_regional_domain_name
```

```
    origin_id   = "s3-vod-dest"
    s3_origin_config {
      origin_access_identity = aws_cloudfront_origin_access_identit
y.vod.cloudfront_access_identity_path
    }
  }

  enabled             = true
  default_root_object = ""

  default_cache_behavior {
    allowed_methods  = ["GET", "HEAD"]
    cached_methods   = ["GET", "HEAD"]
    target_origin_id = "s3-vod-dest"
    viewer_protocol_policy = "redirect-to-https"
    cache_policy_id = data.aws_cloudfront_cache_policy.caching_optim
ized.id
    trusted_key_groups = [aws_cloudfront_key_group.vod.id]
  }

  restrictions {
    geo_restriction {
      restriction_type = "none"
    }
  }

  viewer_certificate {
    cloudfront_default_certificate = true
  }
}
```

12.5 成本估算

MediaConvert 转码成本

按"**输出分钟数 × 编码复杂度**"计费：

档位	单价（美东 us-east-1）
Basic SD (< 30 fps, < 720p, AVC)	$0.0075 / output min
Basic HD (< 30 fps, 720p–1080p, AVC)	$0.015 / output min
Pro HD (60 fps 或 HEVC 或 HDR)	**$0.030** / output min
Pro UHD (4K+)	$0.060 / output min
AV1 加价约 2–3x	

一部 90 分钟电影 × 6 档 (4 H.264 + 2 H.265)：

```
90 min × 4 档 × $0.015 = $5.4 (H.264)
90 min × 2 档 × $0.030 = $5.4 (H.265)
总转码成本 ≈ $10.8 / 电影
```

S3 存储

- 存储 Standard：$0.023 / GB / 月（us-east-1）
- 一部电影所有版本 ~ 3 GB → $0.07 / 月

大库（10 万小时）：约 1 PB → **$23,000 / 月**。 冷内容用 S3 Glacier 可省 80%。

CloudFront 出站带宽（见 07 章 §11）

按区域 0.005–0.08 / GB，阶梯折扣。

MediaPackage JIT packaging

约 $0.065 / GB（源文件流量）。大量访问热门内容划算。

12.6 其他云的等价服务

AWS	阿里云	腾讯云	Google Cloud	Azure
S3	OSS	COS	GCS	Blob Storage
MediaConvert	VOD / IMM	VOD / MPS	Transcoder API	Media Services
MediaPackage	VOD DRM	VOD DRM	Media CDN	Media Services DRM
CloudFront	CDN / DCDN	CDN / ECDN	Cloud CDN / Media CDN	Azure CDN
Step Functions	Serverless Workflow	SCF + WorkflowRun	Workflows	Durable Functions
Lambda	FC / SAE	SCF	Cloud Functions	Azure Functions

选云建议：

- 全球业务、海外为主：**AWS**（覆盖最广）
- 中国大陆为主：**阿里云 / 腾讯云**
- 短剧出海：**AWS + 阿里云国际 / 火山引擎**（多云分发）

12.7 一个典型短剧 APP 的 AWS 账单（推断）

按 DAU 2M、日均 30 分钟观看、720p 主码流推断：

科目	月成本量级
S3 存储 (10 PB)	50*K*–150K
MediaConvert (每天 100 集新上)	1*K*–5K
CloudFront (24 PB/月)	400*K*–800K

科目	月成本量级
MediaPackage JIT	50*K*–150K
Lambda + Step Functions	5*K*–20K
DynamoDB + RDS	10*K*–30K
DRM SaaS	20*K*–80K
合计	**500*K*–1.2M**

12.8 从 0 到生产上线 Roadmap

第 1 周：最小可用 DEMO

开 AWS 账号，启用 S3、MediaConvert、CloudFront

手动跑一个 MediaConvert Job，转出一个 HLS

用 Safari 打开 m3u8 能播 ✅

第 2–4 周：自动化管线

Step Functions 串联：S3 Upload → MediaConvert → S3 → CloudFront

后端生成 Signed URL

iOS/Android APP 集成播放器

第 2–3 个月：生产特性

多档码率 + per-title 优化

DRM 接入（EZDRM / PallyCon）

QoE 数据接入 Mux 或自建

多 CDN 调度

第 4–6 个月：规模化

H.265 / AV1 档位

LL-HLS（如果做直播）

多区域部署 + 跨区复制

成本优化（Spot、Glacier 冷归档）

12.9 常见陷阱

❌ CloudFront 签名 URL 缓存命中率为 0

原因：签名参数（Signature, Expires, Key-Pair-Id）进入了 cache key。

解决：在 Cache Policy 里只把 path 作为 cache key，signed parameters 放 Query String 但**不参与缓存**。

❌ MediaConvert 输出文件异常大

原因：没配 QVBR 或 MaxBitrate，CRF 太低导致峰值爆炸。

解决：`QvbrQualityLevel: 7-8` + `MaxBitrate` 约束。

❌ HLS 在 iOS 上卡在加载

原因：Master Playlist 里缺 `CODECS` 属性。

解决：确保每个 `EXT-X-STREAM-INF` 都带 `CODECS="avc1.xxxxxx,mp4a.40.2"`。

❌ 跨境分发延迟高

原因：CloudFront 回源到 us-east-1 S3 对亚洲用户延迟高。

解决：启用 **CloudFront Origin Shield**，放在离 origin 近的地方；或者 S3 跨区复制，让 CloudFront 就近回源。

📌 本章要点回顾

1. AWS 核心四件套：**MediaConvert + MediaPackage + CloudFront + S3**。
2. **Step Functions** 做编排，Lambda 做轻处理。
3. 用 **SPEKE** 接 DRM 托管服务。
4. 生产环境用 **Terraform 声明式管理**。
5. 成本里 **CDN 带宽是大头**，其次是存储，转码相对便宜。
6. 其他云都有对等服务；选云按市场定位。
7. 从 MVP → 生产一般 2–6 个月。

附录 · 术语速查表

> 所有缩略语、关键术语集中索引。两种查法：按字母序、按主题分类。

按字母序

A

术语	全称	说明	章节
AAC	Advanced Audio Coding	流媒体首选音频编码	03
ABR	Adaptive Bitrate	自适应码率，按网络自动切清晰度	06
AC-3 / E-AC-3	Dolby Digital / Plus	电影 5.1/7.1 音频编码	03
AOM / AOMedia	Alliance for Open Media	AV1 的制定组织	02
Atmos	Dolby Atmos	杜比全景声	03
AV1	AOMedia Video 1	免版税的下一代视频编码	02
AVC	Advanced Video Coding	H.264 的学名	02
AVPlayer	–	iOS 原生播放器	09

B

术语	全称	说明	章节
BBA	Buffer-Based Adaptation	基于缓冲的 ABR 策略	06
BD-rate	Bjøntegaard-Delta rate	视频编码效率对比指标	02
Bitrate	–	码率，每秒 bit 数	01
Bitrate Ladder	–	多档码率阶梯	02
BMFF	Base Media File Format	ISO/IEC 14496-12，MP4 基础	04
BOLA	Buffer Occupancy based Lyapunov Algorithm	基于 Lyapunov 的 ABR 算法	06

C

术语	全称	说明	章节
CABAC	Context-Adaptive Binary Arithmetic Coding	H.264 的熵编码	02
CBCS	–	CENC 加密的一种模式（CBC+Pattern），FairPlay 必须	08
CBR	Constant Bitrate	恒定码率	01
CDN	Content Delivery Network	内容分发网络	07
CDM	Content Decryption Module	浏览器/OS 内的 DRM 解密组件	08

术语	全称	说明	章节
CEK	Content Encryption Key	内容加密密钥（16 字节 AES key）	08
CENC	Common Encryption	ISO/IEC 23001-7，统一加密标准	08
CMAF	Common Media Application Format	ISO/IEC 23000-19，HLS/DASH 共用容器	04, 05
CRF	Constant Rate Factor	x264/x265 恒定质量参数	02

D

术语	全称	说明	章节
DASH	Dynamic Adaptive Streaming over HTTP	MPEG 标准流媒体协议	05
DCT	Discrete Cosine Transform	编码中的频率变换	02
Dolby Vision	–	杜比顶级 HDR	01
DRM	Digital Rights Management	数字版权管理	08

E

术语	全称	说明	章节
EBU R128	–	欧洲广电响度标准（-23 LUFS）	03
EBVS	Exit Before Video Start	起播前退出率	10

术语	全称	说明	章节
EME	Encrypted Media Extensions	W3C DRM API	09
ExoPlayer	–	Android 官方播放器	09

F

术语	全称	说明	章节
FairPlay	FairPlay Streaming	Apple 的 DRM	08
faststart	–	MP4 moov 前置选项	04
ffmpeg	–	开源音视频处理瑞士军刀	01, 02
ffprobe	–	ffmpeg 的元数据探针	01, 11
fMP4	Fragmented MP4	分片 MP4，流媒体标配	04
FLAC	Free Lossless Audio Codec	无损音频	03
fps	frames per second	帧率	01

G

术语	全称	说明	章节
GOP	Group of Pictures	两个 I 帧之间的一组图像	01

H

术语	全称	说明	章节
H.264	–	= AVC，最通用视频编码	02
H.265	–	= HEVC，压缩率更高	02
H.266	–	= VVC，最新一代	02
HDCP	High-bandwidth Digital Content Protection	HDMI 链路保护	08
HDR	High Dynamic Range	高动态范围	01
HDR10	–	免版税 HDR 格式	01
HE-AAC	High Efficiency AAC	低码率优化的 AAC	03
HEVC	High Efficiency Video Coding	= H.265	02
HLG	Hybrid Log-Gamma	广电兼容的 HDR	01
HLS	HTTP Live Streaming	Apple 的流媒体协议	05
HLS AES-128	–	HLS 内置轻量加密（不是 DRM）	08
HTTP/2 / HTTP/3 / QUIC	–	新一代 HTTP 协议	07

I

术语	全称	说明	章节
I 帧	Intra-coded frame	独立解码的关键帧	01
IDR 帧	Instantaneous Decoder Refresh	特殊 I 帧，是切片对齐点	01
IFrame	–	HLS 中的 I-frame only playlist（快进用）	05

J

术语	全称	说明	章节
JIT Packaging	Just-in-Time Packaging	即时打包	07

K

术语	全称	说明	章节
KID	Key ID	切片对应的加密 key ID	08

L

术语	全称	说明	章节
L1 / L2 / L3	–	Widevine 的安全等级	08
LL-HLS	Low-Latency HLS	低延迟 HLS	05
LUFS	Loudness Units Full Scale	感知响度单位	03

M

术语	全称	说明	章节
m3u8	–	HLS manifest 文件扩展名	05
MediaConvert	AWS Elemental MediaConvert	AWS 转码服务	12
MediaPackage	AWS Elemental MediaPackage	AWS 打包+DRM 服务	12
Mezzanine	–	母版/源文件	11
moov	–	MP4 元数据 Box	04
moof	–	fMP4 片段元数据 Box	04
MP3	MPEG-1 Audio Layer III	老音频编码	03
MP4	–	最常见容器格式	04
MPC	Model Predictive Control	ABR 预测控制算法	06
MPD	Media Presentation Description	DASH manifest	05
MPEG	Moving Picture Experts Group	国际多媒体标准组织	–
MPEG-TS	–	广电/老 HLS 用的容器	04
MSE	Media Source Extensions	W3C JS API	09

O

术语	全称	说明	章节
Opus	–	免版税高效音频编码	03
Origin	–	CDN 源站	07

P

术语	全称	说明	章节
P 帧	Predicted frame	前向预测帧	01
B 帧	Bidirectional frame	双向预测帧	01
Per-Title Encoding	–	Netflix 的每部片子定制编码	02
Per-Shot Encoding	–	每个镜头定制编码	02
Pensieve	–	AI 强化学习 ABR	06
PlayReady	–	Microsoft 的 DRM	08
Preconnect	–	提前建立 TCP/TLS 连接	07
Prewarm	Pre-warm	CDN 预热	07
PSNR	Peak Signal-to-Noise Ratio	客观画质指标	02
PSSH	Protection System Specific Header	DRM 加密头信息	08
PTS	Presentation Timestamp	帧显示时间戳	09

Q

术语	全称	说明	章节
QoE	Quality of Experience	用户感知的体验质量	10
QoS	Quality of Service	网络/服务客观性能	10
QVBR	Quality-Defined Variable Bitrate	AWS MediaConvert 的质量模式	12

R

术语	全称	说明	章节
RBR	Rebuffering Ratio	卡顿率	10
Rendition	–	一个 ABR 档位的输出版本	05
RGB	Red-Green-Blue	三原色像素表示	01

S

术语	全称	说明	章节
Segment	–	切片	04
Shaka Packager	–	Google 开源打包工具	05
Shaka Player	–	Google 开源播放器	09
Signed URL	–	带签名的防盗链 URL	07
SPEKE	Secure Packager and Encoder Key Exchange	AWS DRM 对接协议	08

术语	全称	说明	章节
SPI	Streaming Performance Index	Conviva 的综合 QoE 指标	10
SSIM	Structural Similarity	客观画质指标	02
SVT-AV1	–	Intel/Netflix 开源 AV1 编码器	02

T

术语	全称	说明	章节
TS	Transport Stream	MPEG-TS 容器	04
TEE	Trusted Execution Environment	硬件安全区	08
TTFF	Time to First Frame	首帧时间	09, 10

V

术语	全称	说明	章节
VBR	Variable Bitrate	可变码率	01
VMAF	Video Multi-Method Assessment Fusion	Netflix 感知画质指标	02
VOD	Video on Demand	视频点播	00
VP9	–	Google 的视频编码	02
VPF	Video Playback Failure	播放中途失败	10
VSF	Video Start Failure	起播失败	10
VST	Video Startup Time	起播时间	10

术语	全称	说明	章节
VVC	Versatile Video Coding	= H.266	02

W

术语	全称	说明	章节
WebRTC	Web Real-Time Communication	超低延迟音视频	05
WebVTT	Web Video Text Tracks	HTML5 字幕格式	04
Widevine	–	Google 的 DRM	08

Y

术语	全称	说明	章节
YUV / YCbCr	–	视频世界的像素格式	01
YUV 4:2:0 / 4:2:2 / 4:4:4	–	色度子采样方式	01

按主题分类

🎬 视频本质

像素 · 分辨率 · 帧率 · RGB · YUV · 子采样 · 位深 · 色彩空间 · HDR · SDR · I/P/B 帧 · GOP · IDR · 码率 · CBR/VBR/CRF

→ 01–视频基础

🔧 编解码

H.264/AVC · H.265/HEVC · VP9 · AV1 · H.266/VVC · CRF · Preset · BD-rate · Per-title · Per-shot · VMAF · PSNR · SSIM · DCT · 量化 · CABAC

→ 02-编解码入门

🎵 音频

AAC · HE-AAC · MP3 · Opus · AC-3 · E-AC-3 · Atmos · FLAC · 采样率 · 位深 · 声道 · LUFS · EBU R128

→ 03-音频基础

📦 容器与封装

MP4 · fMP4 · MOV · MKV · WebM · MPEG-TS · Box · moov · moof · mdat · faststart · CMAF · WebVTT · IMSC1

→ 04-文件封装

🌐 流媒体协议

HLS · DASH · CMAF · m3u8 · MPD · Segment · Rendition · LL-HLS · CMAF-LL · WebRTC · MPEG-TS

→ 05-流媒体协议

📊 ABR

BBA · BOLA · MPC · Pensieve · Throughput-based · Buffer-based · Bitrate Ladder · 切换惩罚

→ 06-自适应码率ABR

🚀 CDN

Edge · Shield · Origin · 缓存命中率 · Signed URL · Prewarm · JIT / Pre-packaging · 多 CDN · HTTP/3 / QUIC · Request Collapsing

→ 07-CDN分发

🔐 DRM

Widevine · FairPlay · PlayReady · CENC · CBCS · CDM · CEK · KID · PSSH · L1/L2/L3 · HDCP · SPEKE · HLS AES-128

→ 08-DRM版权保护

🎮 播放器

MSE · EME · AVPlayer · ExoPlayer · Media3 · hls.js · Shaka Player · TTFF · Buffer · 硬解 · 软解

→ 09-播放器

📈 QoE

VST · RBR · VSF · EBVS · VPF · SPI · Conviva · Mux · Average Bitrate

→ 10-QoE数据体系

🔄 工作流

Mezzanine · Probe · Transcode · Package · Publish · Prewarm · Step Functions · Temporal · Airflow

→ 11-端到端工作流

☁️ AWS

S3 · MediaConvert · MediaPackage · CloudFront · Lambda · Step Functions · DynamoDB · SPEKE · Terraform

→ 12–AWS实战参考

返回

- **00–入门导览** · 从头开始
- **README** · 项目主页