

x264 视频压缩教程完整版

欢迎阅读！本教程难度一般，不具备学术价值，若有什么不会的可以直接加群 [691892901](#) 哦(´ω´)ゞ

部分 1a: 常识啊常识 (´^`)/

封装文件

- **MP4** 是最通用的格式，适合网站，电脑，移动端和网站播放，对字幕的支持性一般(O· · O)/
- **MKV** 可封装几乎所有的已知视频/音频/字幕格式，就是不能直接导入视频编辑软件(•ω•)
- **FLV** 是目前最轻量的，更适合网站，电脑，移动端和网站播放
- **MOV** 由苹果公司研发，是 Adobe Premiere 的最佳支持格式ヾノ。òゝó)/シ
- **M4A** 是 AAC, ALAC 等音频的封装，因为可以多写点元数据 \ (@ゝゝ@) /
- **M3U** 由苹果公司研发在线播放音频的封装，用 http 协议所以延迟低，**M3U8** 指 utf-8 封装

编码-解码叫 encoding, decoding. **编码即压制**，**解码是编码逆运算，即播放**。**硬解**代表将高能低耗用

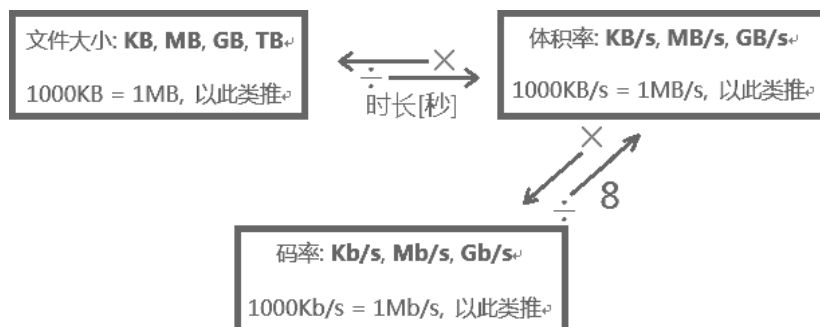
途少的专用电路嵌入 x86/arm 处理器替代**软解**，专治手机看视频烫的优化。方案有 NVDEC, libmxxv,

OpenCL, MMAL, D3D9/11, VDPAU, Media Foundation 等，**硬件编码**方案有 NVENC, 高通

MMF/Venus, 英特尔 QSV, Conexant, Elgato 等，专治手机录像烫手的问题

码率就是文件体积每秒，单位 Kbps 或 Mbps. **10MB**, **1** 分钟的视频，平均是 $(10 \div 1 \times 60) \times 8\text{bit} =$

1333.3kbps. 10MB, 下载 5 分钟，网速即 $10 \div (5 \times 60) = 33.3\text{kBps}$, $\times 8\text{bit} = 266.7\text{kbps}$



色彩是眼睛捕获特定长度的电磁波，再经过脑补而产生的幻觉

滤镜工具有 Premiere, After Effects, Audition, DaVinci Resolve, AviSynth, VapourSynth, ffmpeg, FL Studio, Reaper, Ableton 等等

三角形定律是不可兼得的视频压缩准则，全都要的话只有升级处理器了



向量是物理概念，指瞬间的速度+方向。无论视频怎么动，在每帧里都算直走

宽高比/展弦比 aspect ratio 是视频的宽高标准。 **修改分辨率**需要按比例换算. 换算时将宽/高除以其对应的比例，再乘以临边的比例即可缩放出目标边的比例，小数结果取最接近的偶数即可。如高 720px, 比例是 18:9, 宽就是 $720 \div 9 \times 18 = 1440$ 。图：16:9 下的各种宽和高 (ΦVΦ)

640 : 360		928 : 522		1216 : 684		1504 : 846		1792 : 1008
	800 : 450		1088 : 612		1376 : 774		1664 : 936	
672 : 378		960 : 540		1248 : 702		1536 : 864		1824 : 1026
	832 : 468		1120 : 630		1408 : 792		1696 : 954	
704 : 396		992 : 558		1280 : 720		1568 : 882		1856 : 1044
	864 : 486		1152 : 648		1440 : 810		1728 : 972	
736 : 414		1024 : 576		1312 : 738		1600 : 900		1888 : 1062
	896 : 504		1184 : 666		1472 : 828		1760 : 990	
768 : 432		1056 : 594		1344 : 756		1632 : 918		1920 : 1080

GUIgraphic user interface 图形界面交互(开关旋钮滑块), 上手容易, 但有时缺功能就只能改用 CLI

CLIcommand line interface 命令行交互(输命令敲回车), 上手难但越用越方便, 缺功能就改用 API

APIapp. programming interface 应用编程交互(GUI/CLI 套壳到程序或驱动), 排版瞎眼, 因为设计出来就是给程序用的... 一些视频编码器的修改版就是新加了 CLI 命令来关联到某个 API 的版本

mux 封装, **demux** 解封装分别缩写于 multiplex 混流和 demultiplex 拆流. 压缩包合成多个文件, 频率差距大的波形组成一整波形信号都算混流. 同样硬盘上拷大文件的速度更快. (≡^≡)

录屏是将显卡输出的画面返回给 cpu 压制, 或在显卡, 加速卡上做硬件编码再存到硬盘上的过程. 相比压制, 一般情况下的录屏为了不卡就放弃了画质和文件体积. 目前有显卡录屏, 升级处理器, 上计算卡, 或采集卡导入另一台电脑压制几种办法(ノvノ)

色彩空间 color space 如下表, 或(B, Y 站)的科普视频, 包括

色度采样, 色彩平衡, 调色基础, 调色工具原理, 心理视觉原

理, 多媒体色彩控制

RGB → YCbCr → RGB
录制/未压缩 储存/网络发布 播放

色彩格式	构成色	特点	存在原因	支持范围
RGB	红绿蓝	最常用	使显示器/照相机通用	几乎所有可视媒体
ARGB	α红绿蓝	透明通道	不用抠图了	图片, 部分视频
CMYK	湛洋黄黑	减法色彩	多卖一盒墨	打印纸
YCbCr (YUV, 近似 YPbPr)	白蓝-黄红-绿	压缩	压缩视频图片	所有有损压缩

色度采样 chroma subsampling 写作 A:B:C; 每个长 A 的空间, 首行 B 个色度像素, 第二行 C 个

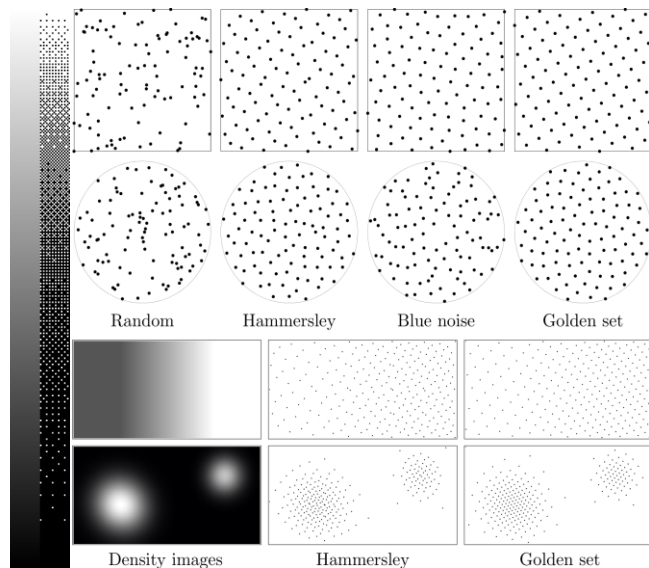
采样	1920x1080 下宏观	色度微观, 宽 A 高 2				特点
4:4:4	亮度, 色度皆 1920x1080	色素	色素	色素	色素	逐行扫描, 直接读写亮色度而不用算, 所以剪视频最快
		色素	色素	色素	色素	
4:2:2	亮度不变, 色度 960x1080	色素 ←		色素 ←		逐行扫描 progressive, 颜色像素靠插值 interpolation 还原. 直接
		色素 ←		色素 ←		
4:2:0	亮度不变, 色度半宽半高 960x540	色素 ← ↑ ↘		色素 ← ↑ ↘		读写亮度所以剪视频比 RGB 快, (反正一般看不出来)
4:1:1	亮度不变, 色度 1/4 宽全高 480x1080	色素 ← ← ← ←				分行扫描 interlaced, 横着扫就不会把另一场的帧参考而浪费算力
		色素 ← ← ← ←				
4:1:1	亮度 3/4 宽全高, 1440x1080 色	色素 5.333x←				了. pixel aspect ratio 指变宽比 (原视频宽: 输出视频宽)
par4:3	度 3/16 宽全高, 360x1080	色素 5.333x←				
4:2:0	亮度 3/4 宽全高, 1440x1080 色	色素 2.667←		色素 2.667←		逐行扫描, 非方形亮度像素, par 通常是 4: 3 要不然太烧脑子啊
par4:3	度 3/8 宽半高, 720x540	↑ 2.848 ↘		↑ 2.848 ↘		

色深 color-depth 代表亮度 Y 的明暗密度. 2¹⁰ 代表 1024, 2⁸ 代表 256 个度, 和子网掩码一样; 度数不够会出现画面渐变色带 banding, 以及音频低音糊掉的问题. 可用频闪或抖动来缓解

抖动滤镜有[随机](#), [蓝噪声](#), [黄金分割](#), [Riemersma](#),

Floyd 等算法, 通过破坏量化使输出结果更接近原始信号, [用分辨率/音频采样率换位深的补偿方法](#), 因此一般在视音频高转低位深时用到, 或像奥博拉丁的回归[用不同抖动算法来区分材质](#)的 ditherpunk 美术. **噪点**源于胶片表面粗糙, 相机/电视强化暗场信号/无信号而产生的白噪, 而降噪手段只有模糊, 或者堆算力来尽量避开

画面细节的模糊. **偏色**是相机内置拜耳滤镜补偿 CMOS 空缺区间, 因算法不同导致的画面问题. 催生了直接导出 **RAW 格式**. 用 PhotoLab, Lightrooms 软件内置拜耳滤镜统一处理的需求



伽马 gamma/希腊符 γ 代表亮度 Y 值或信号强度. **伽马矫正**分为消除录剪播环节的硬件缺陷, 如 CRT 屏要放大电压到 $v^{2.2}$ 才能正确显示灰阶; 以及解决未校正/RAW 画面在演化出黑化非线性光感($\gamma^{1/2.2}$)的人眼看来是漂白变灰"问题"的两用. 由于电压-光感的幂互为倒数, 所以 sRGB 标准中将这个过程简化为"录制时直接白化($\gamma^{2.2}$), 显示器跳过黑化". 因此用任何工具处理媒体时, 要开启"按照伽马系数混合 RGB 颜色, 伽马值=1 (Adobe 软件默认关)"才能正确(且好看)的混合和模糊颜色

逐行扫描 progressive 按左→右及上→下的顺序播放各帧, 例如宽达 4 像素, 高 1080 像素的逐行扫描视频就和 1920x1080 逐行扫描视频一并叫做 1080p 视频. **分行/隔行扫描** interlaced 用 4:1:1 采样, 分**上场优先**: 先传输上帧/奇数行及**下场优先**: 先传输下帧/偶数行, 用于降低传输模拟电视信号干扰. 同期不受干扰问题的电影标准用逐行扫描. 由于标准同时还决定分为电影 24fps, NTSC 电视 60fps, 和 PAL 电视 50fps 三种帧率, 所以标准间做转换还要将电影源进行 NTSC 3:2 pulldown 或 PAL 2:2 pulldown 转为分行; 反过来要做 inverse telecine (IVTC)才能转回 24fps 的逐行视频, 方法见教程尾

帧数就是图片数量每秒, 电影, 纪录片的帧数通常是 24, 25 帧, 网络视频的帧数应该是 30, 60 帧. 而**小数帧率**如 23.976, 29.97 帧**不是为节省码率**, 而是抗模拟信号干扰用, 网络视频用了会丢帧((; °Δ°)

视频网站定律指对低流量 UP 限码, 高流量 UP 画音质优待的普遍现象, 所以想提升码率要先火起来

架构	优点	缺点
Reduced Instruction Set Computer 精简指令集电脑(RISC 架构, CPU)	堆软件设计, 堆核心	编程难优化, 同频率更慢
Complex Instruction Set Computer 复杂指令集电脑(CISC 架构, CPU)	堆硬件工艺, 堆制程	
Application Specific Integrated Circuit 特定应用集成电路/定制方案(ASIC 架构, TPU)	交换机/矿机/防火墙省电 +快+延迟低+对电源友好	芯片界苹果, 功能+1≈∞ 价格×10
Application Specific Standard Product 特定应用通制方案(SoC, CPU, GPU, NPU)	什么都能塞	ASIC, C/RISC 缺点, 贵, 总有一代产品会炸功耗
Field Programmable Gate Array 当场可编程门阵列(FPGA 架构, CPU)	以上什么都能模拟, 产品 未到驱动先写测试先行	使用门槛高, 同性能下性 价比远低于 ASSP
Advanced RISC Machines (ARM 公司) X86 架构	一司包办现成设计 主 CISC, 副 RISC	只有少量公司能做, 性能 通常不发烧

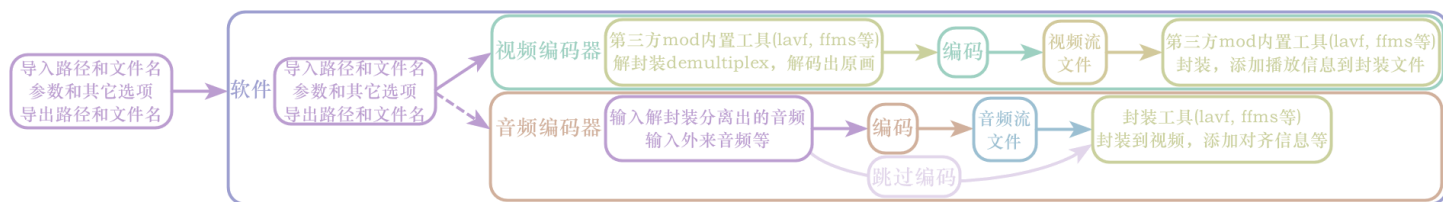
指令集是操作处理器与内存间读写复制粘贴计数寻址来满足软件需求的代码, 随处理器发展变迁

编程语言	优点	缺点/妥协
下层编程语言 Fortan, C 语言, Cyclone, Erlang	编译成机器码后最快	手动内存和指针管理
双层编程语言 C#, C++, Rust-Haskell	+用户输入输出支持	学习成本最高
上层编程语言 tcl/tk, R 语言, PHP, Ruby, Python, JavaScript, Go, F#, Visual Basic 等	自动内存和指针管理	比 C 慢几万倍
折中编程语言 用 Python, Java, .NET 调用 C/Rust 等库	以上所有	你手机的储存空间
非编程语言 SQL, Bash-CMD-PowerShell, GNU grub, Cisco IOS, html, CSS, markdown	反正就是能用	离了软件/系统没用

食用方法, 注意事项

- 1. 使用急用版搭配本篇/精简版配置压制参数, 忘了就打开文档查 ヾ(●ω●)ノ
- 2. 到后面的下载栏 Get 压制软件, 然后研究一下怎么用 \ (@_▽_@) /

压制软件工作流程图解 (不严谨)



命令行参数 command prompt 用法

[引用程序] C:\文件夹\x264.exe

[CLI 参数] --me esa --merange 48 --keyint 100 [...]

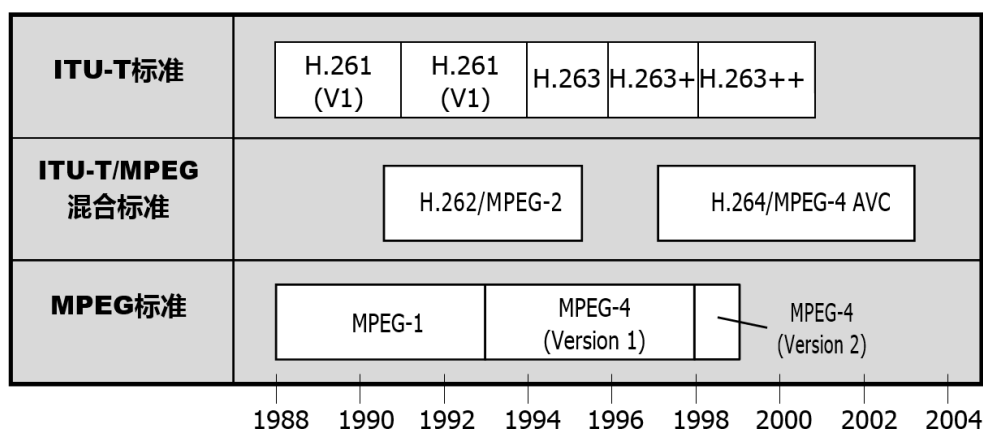
[导出, 空格, 导入] --output C:\文件夹\导出.mp4 C:\文件夹\导入.mp4

[完整 CLI 参数] x264.exe --me esa --merange 24 [...] --output "导出.mp4" "导入.mp4"

命令行参数注意事项: 在 CLI 中, 空格代表命令间的分隔符. 所以带空格的路径要加引号(ΦωΦ)

正片 - x264 的原理及数据类型、(; ∇;)ノ

AVC 标准下, h264/x264 编码器给出的文件由 GOP 组成, 每帧从宏块开始细化到最小 4x4 的块



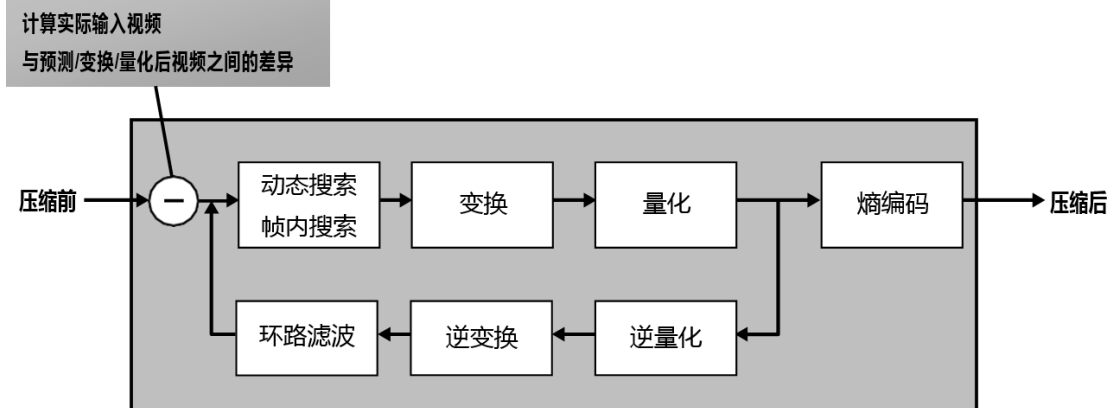
宏块: AVC 视频标准定义为 16x16 大小的初始分块结构, 在此基础上 x264 会根据画面继续分块

GOP: group of pictures, 视频中代表一组由参考帧和非参考帧组成的子视频

关键帧: 由(min-)keyint 参数决定, scenecut 参数判断. 一般作为 gop 开头

- I 帧是图像, 给两侧 B, P, b 帧做参考. 节省设立 IDR 帧的算力

- IDR 帧有创建进度条落点 access point, 以及左右割为独立的子视频, 不再相互参考两个作用



参考帧: 有 IDR, I, P 和 pyramid-B 帧四种. 区分 P, B 帧由 b-adapt 参数决定

- P 帧含 I P 宏块, 上下帧不同但有压缩价值时设立. 往进度条前参考, 叫 prediction frame
- pyramid B 帧含 I P B 宏块, 上下帧几乎一样时设立. 给左右 b 帧参考, 实现了更长的连续 b 帧. 如 IbBbBbBbP

非参考帧: B 帧, 其中的信息完全来自参考帧. 和 pyramid B 帧统称 bi-prediction frame

待定帧: 编码器内部, Lookahead 线程内部未处理且没有被手动指定, 记做 AUTO 的帧

--min-keyint<整数>, 默认 25>判断新发现的转场距离上个 IDR 帧是否小于该值长短. 有两种设定逻辑, 而它们给出的画质都一样:

- 设 5 或更高, 省了设立一些 IDR 帧拖慢速度. 快速编码/直播环境直接设=keyint ^(>_<^)
- 设 1 来增加 IDR 帧, 一帧被判做转场本来就意味着前后溯块的价值不高. 而 P/B 帧内可以放置 I 宏块, x264 会倾向插 P/B 帧. 好处是进度条落点在激烈的动作场面更密集

--keyint<整数>一般设 $9 \times \text{帧率}$ (9 秒一个落点), 拖动进度条越频繁, 就越应该降低 (如 $5 \times \text{帧率}$)

- 短视频, 不拖进度条可以设 keyint -1 稍微降低文件体积, 剪辑素材设 5 秒一个

--ref<整数 1~16, 推荐 $\text{fps} \div 100 + 3.4$, 范围 1~16>溯块参考前后帧数半径, 一图流设 1. 必须要在溯全尽可能多块的情况下降低参考长度, 所以一般设 3 就不用管了

--no-mixed-refs<开关>关闭混合溯块以提速, 增加误参考. 混合代表 16×8 , 8×8 分块的溯帧

初始化与 Lookahead

前瞻与分块等进程最早启动. 建立 GOP 与帧类型(关键帧, 参考帧), 间接影响了 mbtree, VBV 等功能,

以及 ABR, CRF 量化模式的初始数据. x265 教程中展开关于 SPS/PPS 的信息. 大致流程如下:

1. 从--seek 参数指定处开始, 将视频帧导入 Lookahead 进程, 顺序和进度条一样从左往右
2. 后续步骤发现任何 GOP 结尾时, 当前的 h->Lookahead->next->list 帧列表就后移相应的帧数

3. **转场判断①**: 当输入的第 0 帧是 AUTO/I 帧, 且与之前/左侧的非 B 帧差距达到 `--scenecut` 阈值-触发转场时设 IDR 帧
 - 设 IDR (或受 `--open-gop` 限制而设 I 帧) 后, 其前一帧强行设为 P 帧, `min-GOP` 遂变 GOP, Lookahead 列表后移
4. **P/B 帧区分**: 据 `--b-adapt` 指定的算法, `--bframes` 的最大连续 B 帧长度区分出 P/B 帧. 得到如 PBBBPBP 的序列
 - 为保证超长连续 B 帧可行, `--b-pyramid` 参数让偶数的连续 B 帧变成参考帧
5. 单 B 帧/连续 B 帧与其后的 P 帧视为 `min-GOP`
6. mbtree 搜索时-空间范围内的块分布, 找出少见宏块后记下拉高量化值 qp 的偏移程度
7. **转场判断②**: mbtree 跑完后, 计算输入帧与上个关键帧的距离, 超出 `--keyint` 阈值参数则设为 IDR

- 设 IDR (或受 `--open-gop` 限制而设 I 帧) 后, 其前一帧强行设为 P 帧, `min-GOP` 遂变 GOP, Lookahead 列表后移

8. 按 `--rc-lookahead` 参数指定的长度为一组, 通过做差大致给出一组帧的预设量化值 qp 备用
9. VBV 进程检查 `rc-lookahead` 范围内给的 qp 是否能控住码率, 超码则提高这批 qp 备用

`--scenecut`<整数, 不建议修改>Lookahead 中, 两帧差距达到该参数值则触发转场

`--no-mbtree`<开关>关闭少见宏块量化增强偏移. 可能只有 crf 小于 17 才用的到

`--rc-lookahead`<帧数量, 范围 1~250, 推荐 $\text{keyint} \div 2$ >指定 mbtree 的检索帧数, 通常设在帧率的 2.5~3 倍. 高则占用内存增加延迟, 低则降低压缩率和平均画质. mbtree 会
自动选择 `--rc-lookahead` 和 $\max(\text{--keyint}, \max(\text{--vbm-maxrate}, \text{--bitrate}) \div \text{--vbm-buftime} \times \text{fps})$ 中最小的值作为检索帧数

`--lookahead-threads`<整数, Lookahead 线程>开 `opencl` 后可据显卡算力与显存速度, 手动将其提高

`--bframes`<整数, 0~16>连续最多的 B 帧数量, 超出则设立 P 帧, 一般设 14, 手机压片建议设 5 来降低 Lookahead 进程的内存占用

P/B 帧推演-维特比算法

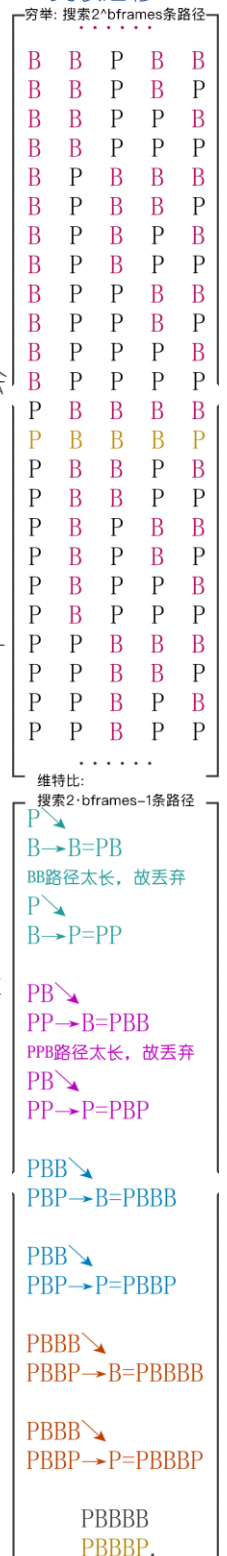
Viterbi-SPF 是多起点-多终点的最短路径 shortest path 1st 算法. 类似的有导航上常见的 Dijkstra-SPF. 推演 B/P 帧相当于解决隐马尔科夫模型问题. 图: `bframes=5`, 每层迭代都保留 B/P 枝梢, 每个枝梢选取一次最短根路径, 最后 P 帧收尾, 得到累计距离最短, 即码率最小的方案

`--b-adapt 2`<所有情况, 整数 0~2, 建议 2>0 代表不设 B 帧

VBV - 基于缓冲条件的量化控制

video buffer verifier 手动指定用户的网络/设备下所允许的缓冲速度 kbps, 以控制

CRF/ABR 模式. 与 CRF 模式一并使用时称为 VBR 双层模式



--vbv-buFSIZE<整数 kbps, 默认关=0, 小于 maxrate>编码器解出原画后, 最多可占的缓存每秒。
 $\text{buFSIZE} \div \text{maxrate}$ =编码/播放时解出每 gop 原画帧数的缓冲用时秒数。相当于要求编完 GOP 平均大小。
编码器用到是因为模式决策要解码出每个压缩步骤中的内容与原画作对比用

--vbv-maxrate<整数 kbps, 默认关=0>峰值红线。用“出缓帧码率-入缓帧码率必须 \leq maxrate”的要求, 让编码器在 GOP 码率超 buFSIZE, 即缓存用完时高压出缓帧的参数。对画质的影响越小越好。当入缓帧较小时, 出缓帧就算超 maxrate 也会因缓存有空而不被压缩。所以有四种状态, 需经验判断

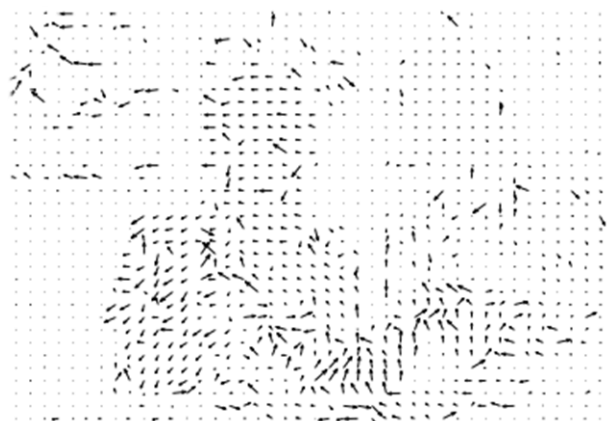
- 大: $\text{GOP 大小} = \text{buFSIZE} = 2 \times \text{maxrate}$, 超限后等缓存满再压, 避开多数涨落, 适合限平均率的串流
- 小: $\text{GOP 大小} = \text{buFSIZE} = 1 \times \text{maxrate}$, 超码率限制后直接压, 避开部分涨落, 适合限峰值的串流
- 超: $\text{GOP 大小} < \text{buFSIZE} = 1 \sim 2 \times \text{maxrate}$, 超码率限制后直接压, 但因视频小/crf 大所以没啥作用
- 欠: $\text{GOP 大小} > \text{buFSIZE} = 1 \sim 2 \times \text{maxrate}$, 超码率限制后直接压, 但因视频大/crf 小所以全都糊掉
- 由于 gop 多样, 4 种状态常会出现在同一视频中。buf/max 实际控制了这些状态的出现概率

--ratetol<浮点, 百分比, 默认 1 (许 1% 错误)>ABR, 2pass-ABR, VBR 的码率超限容错 tolerance

动态搜索

通过算法找到帧间的移动变化信息, 以冗余掉帧间重复块

图: 动态搜索结果



--me<dia, hex, umh, esa>搜索方式, 从左到右依次变得复杂, umh 之前会漏掉信息, 之后收益递减, 所以推荐 umh (° ▽ °) /

--merange<整数>越大越慢的动态搜索范围, 建议 16, 32 或 48。由于是找当前动态向量附近有没有更优值, 所以太大会让编码器在动态信息跑不到的远处找或找错, 造成减速并降低画质

--no-fast-pskip<开关>关闭跳过编码 p 帧的功能。建议在日常和高画质编码中使用

--direct auto<开关, 默认 spatio>指定动态搜索判断方式的参数, 除直播外建议 auto

--non-deterministic<开关, 对称多线程>让动态搜索线程得知旧线程实际搜索过, 而非参数设定区域。理论上能帮助找到正确的动态矢量以增强画质, 增加占用, “deterministic”代表完算性, 即“算完才给出结果的程度”, 反之就是“欠算性”



--no-chroma-me<开关>动态搜索不查色度平面, 以节省一点算力加速压制, 建议直播/录屏用

运动补偿

动态搜索让块间连起来, 运动补偿 motion compensation 用 SAD, SATD 算法找出参考块间子像素最像的源, 将动搜所得的块-帧间插值(运动矢量)细化, 让块间细节精确地连起来. 跳过=大量细节损失 绝对差异求和 Sum of Absolute Difference: 冗余压缩后, 对比压缩前的原始帧逐像素做差, 取绝对值加到一起, 得差异总数为失真程度; 绝对变换差求和 Sum of Absolute Transformed Difference: 改用完成变换计算的帧与源做差, 比冗余压缩往后一步, 距离压缩后的帧更近, 更慢但失真算的更准

$$\text{绝对(变换后)差异和 SA(T)D} = \sum_{x_0 \rightarrow T_x} \sum_{y_0 \rightarrow T_y} |f(x, y) - f'(x, y)|$$

- $\sum_{x=0 \rightarrow T_x}$ 代表块宽度求和范围, $f()$ 和 $f'()$ 分别代表参考块和参考源
- $\sum_{y=0 \rightarrow T_y}$ 代表块高度求和范围, x, y 代表块中的像素坐标, $|$ 求绝对值, 否则求和时正像素值差异会减去负

SAD 和 SATD 在多种编码器, 调制解调器以及滤镜中出现. 因为有信号的地方就有失真, 就要判断失真

--subme<x264 中影响模式决策和率失真优化. 整数范围 0~10, 60fps=9, -=8, +=10>根据片源的帧率判断. 分四个范围. 由于动漫片源制于 24~30fps, 因此可节省一些算力; 但同是动漫源的 60fps 虚拟主播则异. 主流 120Hz 的手机录屏目前最高也不够用. 由于性能损耗大, 所以不建议一直开满

- <1>逐块 1/4 像素 SAD 一次, <2>逐块 1/4 像素 SATD 两次
- <3>逐宏块 1/2 像素 SATD 一次, 再逐块 1/4 像素 SATD 一次
- <4>逐宏块 1/4 像素 SATD 一次, 再逐块 1/4 像素 SATD 一次
- <5>+增加双向参考 b 块
- <6>+率失真优化处理 I, P 帧; <7>+率失真优化处理 I, P, B, b 帧
- <8>+I, P 帧启用 rd-refine; <9>I, P, B, b 帧启用 rd-refine
- <10, 边际效应等于压缩, trellis 2, aq-strength 大于 0>+子像素上跑 me hex 和 SATD 比找参考源
- <11, 边际效应大于压缩>+关闭所有 10 中的跳过加速, 不推荐. 原本是为了 trellis 3 设计的

加权预测 weighted prediction

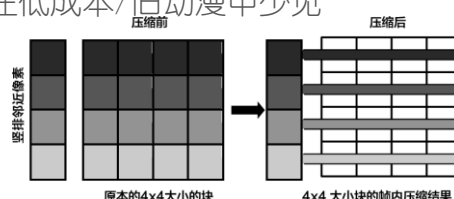


解决淡入淡出过程中部分 pu 误参考, 亮度变化不同步所制造的画面问题. 分为 P, B 条带用的显加权,

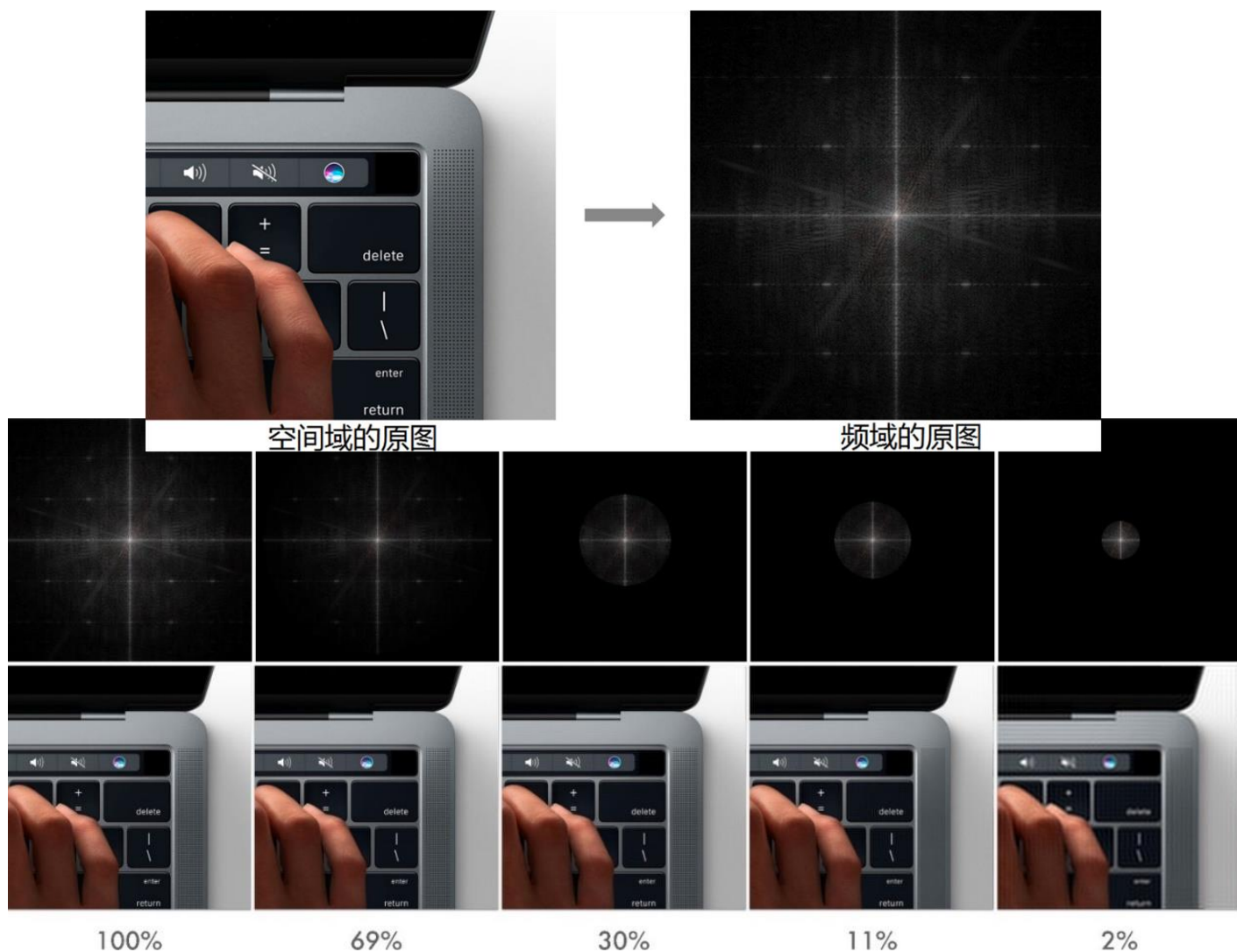
编码器直接从原画和编码过的参考帧做差; B 条带用的隐加权, 用参考帧的距离做加权平均插值

--weightb<开关, 默认关/只加 P 条带>启用 B 条带的显隐加权预测. 条带所在 SPS 中可见 P, B 加权开关状态, 及显加权模式下解码器须知的权重. 光线变化和淡入淡出在低成本/旧动漫中少见

帧内编码: x265 教程中展开

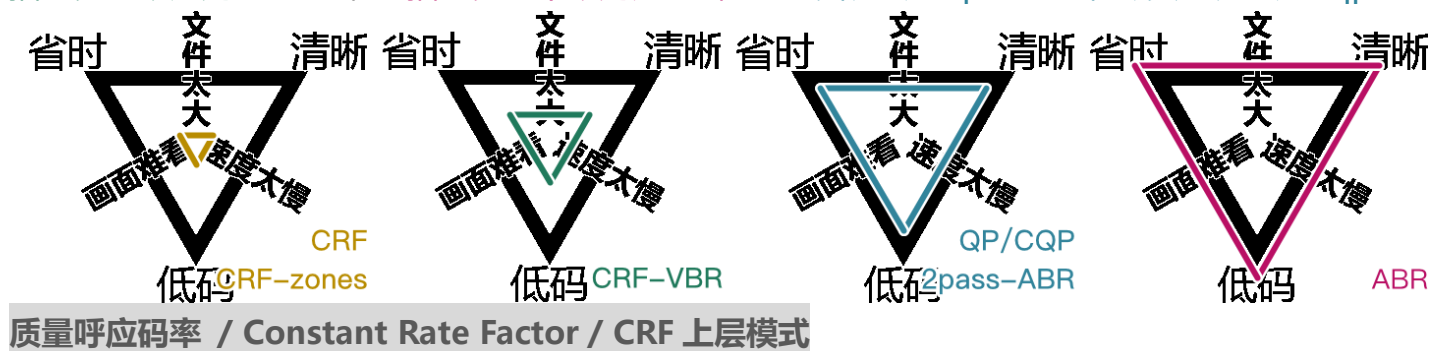


变换-量化: 将频率从低到高的信息列出来, 用低通滤镜砍掉高频信息. x265 教程中展开



码率控制模式

据工况要求与妥协的变化选一种上层模式，搭配 0~全部的下层模式，或特殊模式。一般情况默认 crf，直播时算力够用则 crf~vbr，直播时算力不够则用 abr，码率硬限下用 2pass-abr，做实验测试用 cqp



--crf<浮点范围 0~69，默认 23>据“cplxBlur，mbtree，B 帧偏移”等内部参数实现每帧分配各自 qp 的固定目标质量模式，统称 crf。素材级画质设在 16~18，收藏~高压画质设在 19~20.5，YouTube 是 23。由于动画和录像的内容差距，动画比录像要给低点

平均码率 / Average Bitrate / ABR 上层模式

--**bitrate**<整数 kbps, 指定则关 crf>若视频易压缩且码率给高, 就会得到码率更低的片子; 反过来码率给低了会强行提高量化, 强制码率达标. 一般推流用的“码率选项”即 ABR, 快但妥协了压缩与画质

恒量化值 / Constant Quantizer Parameter / CQP 上层模式

--**qp**<整数, 范围 0~69, 指定则关 crf, 非实验不建议>恒量化. 每 ±6 可以将输出的文件大小减倍/翻倍, 同速度下不如 ABR, 同码率下不如 CRF

常用下层模式

--**qcomp**<浮点范围 0.5~1, 一般建议默认>cplxBlur 迭代值每次能迭代范围的曲线缩放. 越小则复杂度迭代越符合实际状况, crf, mbtree, bframes 越有用, 搭配高 crf 能使直播环境可防止码率突增. 越大则 crf, mbtree, bframes 越没用, 越接近 cqp. 曲线缩放原理见 [desmos 互动示例](#)

--**qpmin**<整数, 范围 0~51>最小量化值, 仅在高压环境建议设 14~16("▽"); --**qpmax**<同上>在要用到颜色键, 颜色替换等需要清晰物件边缘的滤镜时, 可以设 --qpmax 26 防止录屏时物件的边缘被压缩的太厉害, 但其它情况永远不如 no-mbtree (*~▽~)

--**chroma-qp-offset**<整数, 默认 0>h.264 规定 CbCr 的码率之和应=Y 平面, 所以 x264 会拉高 CbCr 的量化. 用 psy-rd 后, x264 会自动给 qp-2 至-4. 不用 psy-rd 时, 4:2:0 的视频可手动设-2至-4

--**ipratio**<浮点, 默认 1.4>P 帧相比 IDR/i 帧; --**pbratio**<浮点, 默认 1.3>B/b 帧相比 P 帧的偏移. 指定 IDR/I 帧 qp17, P 帧 qp20, B/b 帧 qp22 就填写"--qp 17 --ipratio 1.1765 --pbratio 1.1"

不常用: zones 下层模式 - crf-zones 及 abr-zones 两种搭配

--**zones**<开始帧, 结束帧, 参数 A=?, 参数 B=?...>手动在视频中划区, 采用不同上层模式来实现如提高压制速度, 节省平均码率, 提高特定画面码率等用途(一般用来“处理”片尾滚动字幕). zones 内的 me, merange 强度/大小不能超 zones 外. 可用参数有 b=, q=, crf=, ref=, scenecut=, deblock=, psy-rd=, deadzone-intra=, deadzone-inter=, direct=, me=, merange=, subme=, trellis=

10. 参数 b=调整码率比率, 可以限制 zones 内的场景使用当前 0~99999%的码率, 100%相当于不变
11. 参数 q=即 QP 值, 可以用来锁死 zones 内场景使用无损压缩(任何 rate factor)以做到素材用编码

多个划区可以用'/' 隔开: --zones

0, 449, crf=32, me=dia, bframes=10/450, 779, b=0.6, crf=8, trellis=1

不常用: 2pass-CRF-ABR 特殊模式

首遍用 crf 模式分析整个视频总结可压缩信息, 二遍根据 abr 模式的码率限制统一分配量化值. 除非有码率硬限, 否则建议用 crf 模式. 目前所有视频网站一律二压, 因此 2pass-abr 模式没有上传的用

```
--pass 1 --crf 20 --stats "D:\夹\qp.stats" [参数] --output NUL "输入.mp4"
--pass 2 --bitrate x --stats "D:\文件夹\qp.stats" [参数] --output "输出.mp4" "输入.mp4"
```

--**stats**<文件名>默认在 x264 所在目录下导出/入的 qp 值逐帧分配文件, 一般不用设置

--**output NUL**<不输出视频文件>; --**pass 1**<导出 qp.stats>; --**pass 2**<导入 qp.stats>

不常用: FTQP 手动模式

--qpfile<文件名>手动指定特定帧为 IDR, i, P, B, b 帧, 及 closed-gop 下 K 帧的 frame type qp 下层模式. 文件内含"号位 帧类型 QP 值(换行)". 指定 qp 值为-1 时使用上层的 crf, abr, cqp 模式

自适应量化

CRF/ABR 设定每帧量化/qp 后, 方差自适应量化 variance adaptive quantizer 再根据其信息复杂度来分成高低频两类, 从而将 qp 细化到块上. AQ 与 VAQ 存在混淆. **方差** variance 代表差-方-和-均的计算顺序. 将数据样本逐个与整体平均值做差, 通过累计偏差本身来判断数据的平衡性, 乘方是为方便正

负数一起求和计算(不过 σ 也有平方): $\sigma^2 = \frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N}$

VAQ 中, 样本就是宏块里的像素, 要比的平均值就是宏块范围的平均像素值. 求和 Σ 因为差异平方和 SSE, 以及差异和 SAD 算过了所以不写, 样本量 N 就是宏块范围的像素数量. 通过计算像素值的方差, 就能看出哪些宏块含高频信号多, 就能实现 vaq 了

$$Variance = \frac{SSE_{pixels\ within\ MB} - (SAD_{Macroblock})^2}{256}$$

高压缩下, aq 强度不足则纹理边缘的码率不足; 过高则平面/暗处的码率不足, 造成涂抹失真; 无损/快速编码时, aq 强度低则好, 总的说强度要随 CRF/ABR 而动. 由于 aq 不计算帧间关系, 所以 aq 给出的结果往后还要用 mbtree-lookahead, 率失真优化量化 rdoq 来宏观地重分配

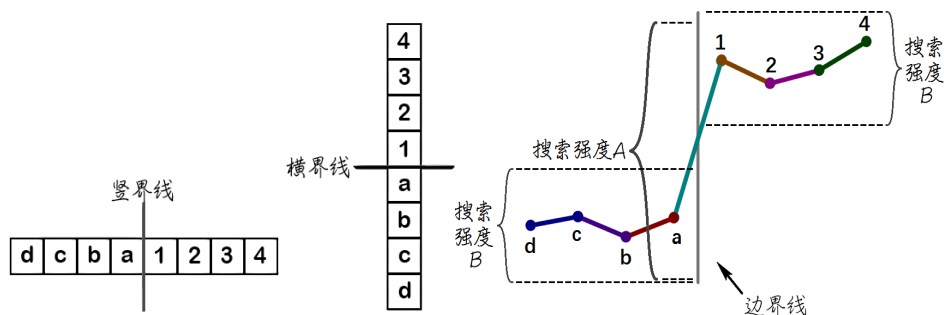
--aq-mode<整数 0~3>据原画和 crf/abr 设定, 以及码率不足时(crf<18/低码 abr)如何分配 qp

- <1>标准自适应量化(急用, 简单平面)
- <2>+启用 aq-variance, 自动调整 aq-strength 强度(录像-电影以及 crf<17 推荐)
- <3>+码率不够用时倾向保暗场(接受更明显的涂抹失真, 慎用)
- <4>+码率不够用时更加倾向保纹理(接受平面上的涂抹失真, 实验性, 慎用)

--aq-strength<浮点>自适应量化强度. 搭配 aq-mode, 如动漫 1:0.8, 2:0.9, 3:0.7 用. 录像上可加 0.1~0.2, 画面混乱/观众难以注意平面时可再增加. 注意低成本动漫的平面居多, 因此码率不足时反而要妥协纹理

环路滤波

修复高量化时宏块间出现明显横纵割痕瑕疵的平滑滤镜. 编码器内做去块能用到压缩待遇信息而减少误判. 码率跟不上就一定会出现块失真, 所以除直播关掉以加速外, 任何时候都应该用; 但去块手段目前仍是平滑滤镜, 因此要降低强度才适用于高码视频, 动漫, 素材录屏等锐利画面. **边界强度** boundary strength(去块力度判断): 取最小 8x8 块间的界线举例. (不是 4x4)



- 平滑 4: a 与 1 皆为帧内块, 且边界位于 CTU/宏块间, 最高值
- 平滑 3: a 或 1 皆为帧内块, 但边界不在 CTU/宏块间
- 平滑 2: a 与 1 皆非帧内块, 含一参考源/已编码系数
- 平滑 1: a 与 1 皆非帧内块, 皆无参考源/已编码系数, 溯异帧或动态向量相异
- 平滑 0: a 与 1 皆非帧内块, 皆无参考源/已编码系数, 溯同帧或动态向量相同, 滤镜关

--deblock<平滑强度:搜索精度, 默认 1:0, 推荐 0:0, -1:-1, -2:-1>两值于原有强度上增减

- 平滑<≥1>时用以压缩, <0~1>时略微降低锐度, 适合串流
- 平滑<-2~-1>适合锐利视频源, 4k 电影, 游戏录屏. 提高码率且会出现块失真
- 平滑<-3~-2>适合高码, 高锐动画源和高画质的桌面录屏. 高码率, 增块失真, 但高码动漫观感还是比 1 好
- 搜索<大于 2>易误判, <小于-1>会遗漏, 建议保持<0~-1>, 除非 qp>26 时设<1>

模式决策

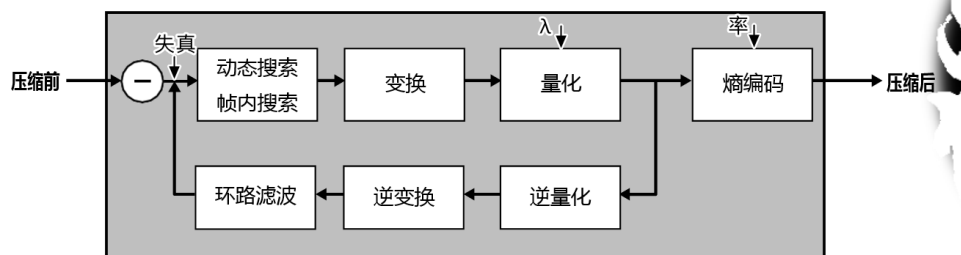
mode decision 整合搜到的信息, 在各种选项中给宏块定制如何分块, 参考, 跳过的优化. 若没有 psy-rd, psy-rdoq(x265)和 trellis 优化, 模式决策就一定用码率最小, 复杂动态下全糊的方案集. mb 树也是 md 的一部分(" ∩ ∇ ∩)八(" ∩ ∇ ∩ ")

--deadzone-inter<整数 0~32, 默认 21, trellis=2 时无效, 小于 2 自动开启>简单省算力的帧间量化, 细节面积小于死区就糊掉, 大就保留. 一般建议 8, 高画质建议 6 √ (≧ ∇ ≦*)o

--deadzone-intra<整数, 范围 0~32, 默认 11>这个顾及帧内. 一般建议 5, 高画质建议 4

率失真优化 RDO 控制

心理视觉 psychological visual 是对人眼感知清晰度的研究



率失真优化 rate distortion optimization 据多码率下测得的失真程度

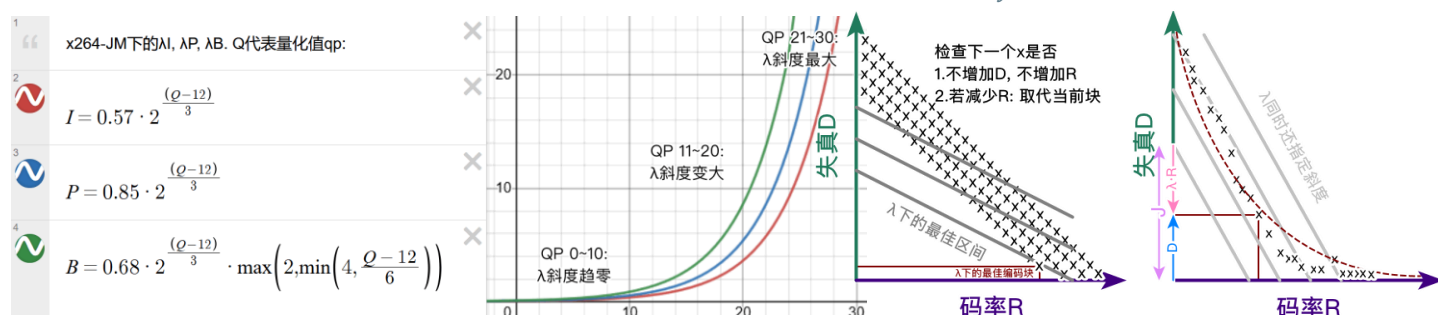
(编码前

后像素值差异)点, 挑出低于 RD 曲线的值. 其中拉格朗日代价函数(开销=失真+ λ ·码率; $J = D + \lambda \cdot R$)实现

模式决策. 失真 D 用差异平方和 SSE 和总差异 SAD 判断. SSE 多取一步平方, 使较大的差异呈指数增

长, 进而分到更多码率实现补偿. 取平方的负值转正效果是良性副作用图: 在真人录像上常见, 但不支持动漫的高频细节

x264 差异平方和 Sum of Squared Error = $\sum_{x_0 \rightarrow T_x} \sum_{y_0 \rightarrow T_y} |f(x, y) - f'(x, y)|^2$



拉格朗日值 λ 从 qp 值得出, 即 crf, abr 指定的率失真斜率区间. qp 越大斜度越小. $\lambda=0$ 则无斜度, 即代

价=失真, 给多少码画面都一样(允许最大压缩). λ 趋 0 则代价趋失真, 即压缩一截下去不会影响多少画

质, 稍微给点码率意思意思; λ 远大于 0 则代价>失真, 提升画质的收益>压缩率降低的收益(保画质)

x264 还有直接根据噪声容忍度判断失真的算法. 极为先进, libx264, x265 都不支持=_=

高频加权总差方 NSSE = $\sum_{x_0 \rightarrow T_x} \sum_{y_0 \rightarrow T_y} |[N(x, y) - N'(x, y)] \cdot fgo| + |f(x, y) - f'(x, y)|^2$

--trellis<整数, 范围 0~2, 推荐 2>一种率失真优化量化 rdoq 算法. <1>调整 md 处理完的块, 快速压制用, <2>+帧内帧间参考和分块

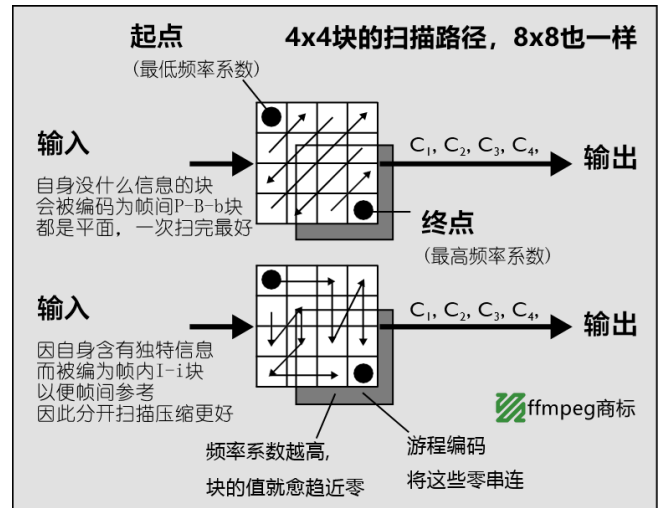
--fgo<整数默认关, 推荐 15 左右>改用 NSSE, 提高画质, libx264 不支持的 Film Grain Opt.

--psy-rd<a:b 浮点, 默认 1:0>心理学优化设置. a 保留画面纹理, b 在 a 的基础上保留噪点细节. 影片的复杂度越高, 相应的 ab 值就越高. 压制动漫时建议选择<0.4~.6:0.1~.15>, --no-mbtree 时可尝试将 b 设为 0; 压制真人选择<0.7~1.3:0.12~.2>

--no-psy<开关>若视频量化很低纹理很清楚, 右图毛刺对画质不好就关. 录像中这些毛刺很重要

游程编码-霍夫曼树

从像素上看, 量化后的块属其实就是矩阵数据. 用游程编



码串联成一维上的字串, 再做文本压缩即可节省最高 1/3 的体积. 游程编码是文本压缩技术. 目的是建立 0, 1 代表的二叉树. 传统熵编码是使得于原点最近, 出现最多的字放在树的左边, 同时让枝干尽可能短-即编写~解读步骤越少; 出现少则靠右, 让枝干尽可能长-即编写~解读步骤越多. 这样从树根开始, 通过 0→左, 1→右地走到树梢上, 就实现了霍夫曼编码. x264 使用了更好的可变长度编码 VLC, 以及目前最好的二进制算术编码 CABAC, 统称为熵编码, 考虑到文档受众所以放在了 x265 教程中. 到此, 视频就被压缩的差不多了

最终还要经过解码, 在宏观层面上对比各种压缩模式, 确保收束到质量最好, 同时做到体积最小的率-失真优化计算, 最终将调优过的方案做熵编码, 然后组装为 h264 视频流输出, 视频的编码就完成了

色彩信息

光强/光压的单位是 candela. 1 candela=1 nit

--master-display<G(x,y)B(,)R(,)WP(,)L(最大,最小)>, 与 x265 的格式不一样>写进 SEI 信息里, 告诉解码端色彩空间/色域信息用, 搞得这么麻烦大概是因为业内公司太多. 默认未指定. 绿蓝红 GBR 和白点 WP 指马蹄形色域的三角+白点 4 个位置的值 × 50000. 光强 L 单位是 candela × 10000

SDR 视频的 L 是 1000,1. 压 HDR 视频前一定要看视频信息再设 L, 见下

- DCI-P3 电影业内: G(13250, 34500)B(7500, 3000)R(34000, 16000)WP(15635, 16450)L(?, 1)
- bt709: G(15000, 30000)B(7500, 3000)R(32000, 16500)WP(15635, 16450)L(?, 1)
- bt2020 超清: G(8500, 39850)B(6550, 2300)R(35400, 14600)WP(15635, 16450)L(?, 1)

RGB 原信息 (对照小数格式的视频信息, 然后选择上面对应的参数):

- DCI-P3: G(x0.265, y0.690), B(x0.150, y0.060), R(x0.680, y0.320), WP(x0.3127, y0.329)
- bt709: G(x0.30, y0.60), B(x0.150, y0.060), R(x0.640, y0.330), WP(x0.3127, y0.329)
- bt2020: G(x0.170, y0.797), B(x0.131, y0.046), R(x0.708, y0.292), WP(x0.3127, y0.329)

--c1l<最大内容光强, 最大平均光强>压 HDR 一定照源视频信息设, 找不到不要用, 例子见下

```
Bit depth           : 10 bits
Bits/(Pixel*Frame)  : 0.120
Stream size         : 21.3 GiB (84%)
Default             : Yes
Forced              : No
Color range         : Limited
Color primaries     : BT.2020
Transfer characteristics : PQ
Matrix coefficients : BT.2020 non-constant
Mastering display color primaries: R: x=0.680000 y=0.320000,
G: x=0.265000 y=0.690000, B: x=0.150000 y=0.060000, White point: x=0.
Mastering display luminance: min: 0.0000 cd/m2, max: 1000.0000 cd/m2
Maximum Content Light Level: 1000 cd/m2
Maximum Frame-Average Light Level: 640 cd/m2
White point: x=0.312700 y=0.329000
位深: 10 位
数据密度【码率/(像素*帧率)】: 0.251
流大小: 41.0 GiB (90%)
编码函数库: ATEME Titan File 3.8.3 (4.8.3.0)
Default: 是
Forced: 否
色彩范围: Limited
基色: BT.2020
传输特质: PQ
矩阵系数: BT.2020 non-constant
控制显示基色: Display P3
控制显示亮度: min: 0.0050 cd/m2, max: 4000 cd/m2
最大内容亮度等级: 1655 cd/m2
最大帧平均亮度等级: 117 cd/m2
```

图 1: c1l 1000,640. master-display 由 G(13250...开头, L(10000000,1)结尾

--colorprim<字符>播放用基色, 指定给和播放器默认所不同的源, 查看视频信息可知: bt470m, bt470bg, smpte170m, smpte240m, film, bt2020, smpte428, smpte431, smpte432. 如图为 bt. 2020

图 2: c1l 1655,117/L(40000000,50)/colorprim bt2020/colormatrix bt2020nc/transfer smpte2084

--colormatrix<字符>播放用矩阵格式/系数: fcc, bt470bg, smpte170m, smpte240m, GBR, YCgCo, bt2020nc, bt2020c, smpte2085, chroma-derived-nc, chroma-derived-c, ICtCp, 不支持 bt2020nc

--transfer<字符>传输特质: bt470m, bt470bg, smpte170m, smpte240m, linear, log100, log316, iec61966-2-4, bt1361e, iec61966-2-1, bt2020-10, bt2020-12, smpte2084, smpte428, arib-std-b67, 上图 PQ 即 st. 2084 的标准, 所以参数值为 smpte2084

灰度/色深, x264 压制 log

[info]: indexing input file [0.7%]

打开视频

ffms [info]:

```
Format      : mov,mp4,m4a,3gp,3g2,mj2
Codec       : h264
PixFmt      : yuvj420p
Framerate   : 60/1
Timebase    : 1000/15360000
Duration    : 0:01:30
```

视频流信息

ffms [info]: 1920x1080p 0:1 @ 60/1 fps (vfr)

ffms [info]: color matrix: bt709

x264 [info]: using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX

x264 [info]: OpenCL ... GeForce GTX 970

软硬件信息

x264 [info]: AVC Encoder x264 core 148 ...xiaowan [8-bit@all X86_64]

x264 [info]: profile: High ... bit-depth: 8-bit

x264 [info]: opencl=1 cabac=1 ref=3 deblock=1:0:0 ... aq3=0

x264 [info]: started at Sat Nov 24 02:58:23 2018

[0.0%] 1/5412 frames, 0.238 fps, 17882 kb/s, 36.38 KB, eta 6:19:13, est.size 192.28 MB

压制信息

[99.4%] 5378/5412 frames, 44.60 fps, 271.22 kb/s, 2.90 MB, eta 0:00:00, est.size 2.92 MB

x264 [info]: frame I:12 Avg QP:21.49 size: 82888

输出视频流信息

(此处被大幅省略)

x264 [info]: kb/s:269.65

encoded 5412 frames, 44.83 fps, 269.72 kb/s, 2.90 MB

x264 [info]: ended at

x264 [info]: encoding duration 0:02:01

其他信息

--fullrange<开关, 7mod x264 自动>启用范围更广的显示器 0~255 色彩范围, 而不是默认的旧电视色彩范围 16~235, 由于 16~235 的颜色管理更准确且码率更小, 所以能不用 Fullrange 就不用

其他命令行参数

--seek<整数, 默认 0>从第 x 帧开始压缩, **--frames**<整数, 默认全部>一共压缩 x 帧 **--fps**<整数, 特殊情况>告诉 x264 帧数

线程

--threads<整数, 建议默认 1.5 倍线程>参考帧步骤要等其之前的步骤算完才开始, 所以远超默认的值会因为处理器随机算的特性而降低参考帧的计算时间, **使码率增加, 画质降低, 速度变慢**

另外一种方案是让 threads 低于 cpu 实际线程, 而是尽可能接近 4 以增加参考帧的优先级, 但这就不如 sliced-threads 了. 而且 24 线程~threads=36 下, 画面的损失应该不会超过 0.5%

--sliced-threads<开关, 默认关>x264 默认每帧逐线程, 速度更快但代价是有的线程没法在确定的时间内吐出结果, 参考帧很容易等不到所以忽略掉本可以压缩的内容, 打开后降低处理器占用, 但压缩率可能会提高, 建议 16+线程的处理器, 或高压缩用 \sim ($\sim \omega \sim$)

--slices<整数, 默认自动但不可靠>手动指定“分片逐线程”下可以有多少分片, 建议等于线程数

裁剪, 加边, 缩放/更改分辨率, 删除/保留视频帧, 降噪, 色彩空间转换

--vf crop: 左, 上, 右, 下/resize: 宽, 高, 变宽比, 装盒, 色度采样, 缩放算法/select_every: 步, 帧, 帧...

/crop: 左, 上, 右, 下指定左上右下各裁剪多少, 最终必须得出偶数行才能压制

/resize: 宽, 高,,,更改输出视频的宽高, 建议搭配缩放算法后使用

/resize: ,,变宽比,,,减少宽度上的像素, 剩下的伸成长方形来达到压缩的参数. 任何视频网站都不支持, 但网盘/商用的视频可以用这种压缩方法. 格式为**源宽度: 输出宽度**

宽从 1920 到 1060, 就是 96:53(约分后), 就是 **resize:1060,高,96:53,,,缩放算法**

/resize: ,,,,色度采样,有 i420, i422, i444 和 rgb 四种, 默认 i420. 在缩小视频分辨率, 或者处理无损源视频时可以尝试使用已获得更好的大屏幕体验. **注意, 被压缩掉色彩空间的视频就不能再还原了**

/resize: ,,,,缩放算法

/select_every: 步, 帧 1, 帧 2...通过少输出一些帧以加速压制, 用于快速预览压制结果, 比如:

8 帧一步, 输出其中第 0, 1, 3, 6, 8 号帧: **--vf select_every: 8, 0, 1, 3, 6, 8**

90 帧一步, 输出其中第 0~25 号帧(最大 100 帧/步): **--vf select_every:**

90, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25

(仅 7mod-降噪) **/hqdn3d:** 空域亮度降噪, 空域色度降噪, 时域亮度降噪, 时域色度降噪

默认值是 4,3,6,4.5 若是编码画面模糊的源可以尝试默认值 1 到 1.7 倍. 若一定要用此参数来降低码率, 可以考虑使用视频编辑软件的模糊滤镜

(仅 7mod-加边) /pad: ←, ↑, →, ↓, 改宽(不和加边混用), 改高(不和加边混用)

用例(帧率减半, 降噪): --vf select_every:2, 0/hqdn3d:0, 0, 3, 1.5


分场扫描/隔行扫描

--tff<开关>上场优先. --bff<开关>下场优先. --nal-hrd<开关, 默认关, 开 vbr>开启假想对照解码参数 hypothetical ref. decoder param. 零丢包零延迟环境中判断解码器额外所需信息, 写进每段序列参数集 sps 及辅助优化信息 sei 里, 适合串流

部分 3: 开工 - 软件下载与使用方法(((￣へ￣井))

ShanaEncoder


ffmpeg-CLI 或 GUI 控制少量选项, 高级功能(水印, 高级字幕)用 ffmpeg 参数控制, 上手需要时间. ffmpeg 内嵌编码器, 不能替换文件



Simple x264 Launcher


英文软件, 适合批量压制, 需要自行封装音频

下图是画的汉化! 只能压视频, 但处理封装与压制音频, 查看媒体元数据不如小丸工具箱, 上手速度快.



小丸工具箱

提取码"crhu" 中文软件, 压缩音视频, 渲染字幕等, 操作简单. 导入视频, 点击自定义, 将参数拷入, 选好输出格



式与滤镜就可以压制了, 网上能搜到详细教程. 内嵌 MediaInfo, mp4box, Mkvtoolnix 可查看媒体元数据, 封装/解封装, 最适合新手前两个小时

mpv 播放器 比 Potplayer 好在没有音频滤镜, 不用手动关; 没有颜色偏差, 文件体积小

OBS 直播与录屏 支持 AVS 滤镜, 设置复杂但强大, 去 [x264 教程急用版](#)照着设置就行了

x264 by Patman, LigH√lavf 编解码, 合并 8~10bit

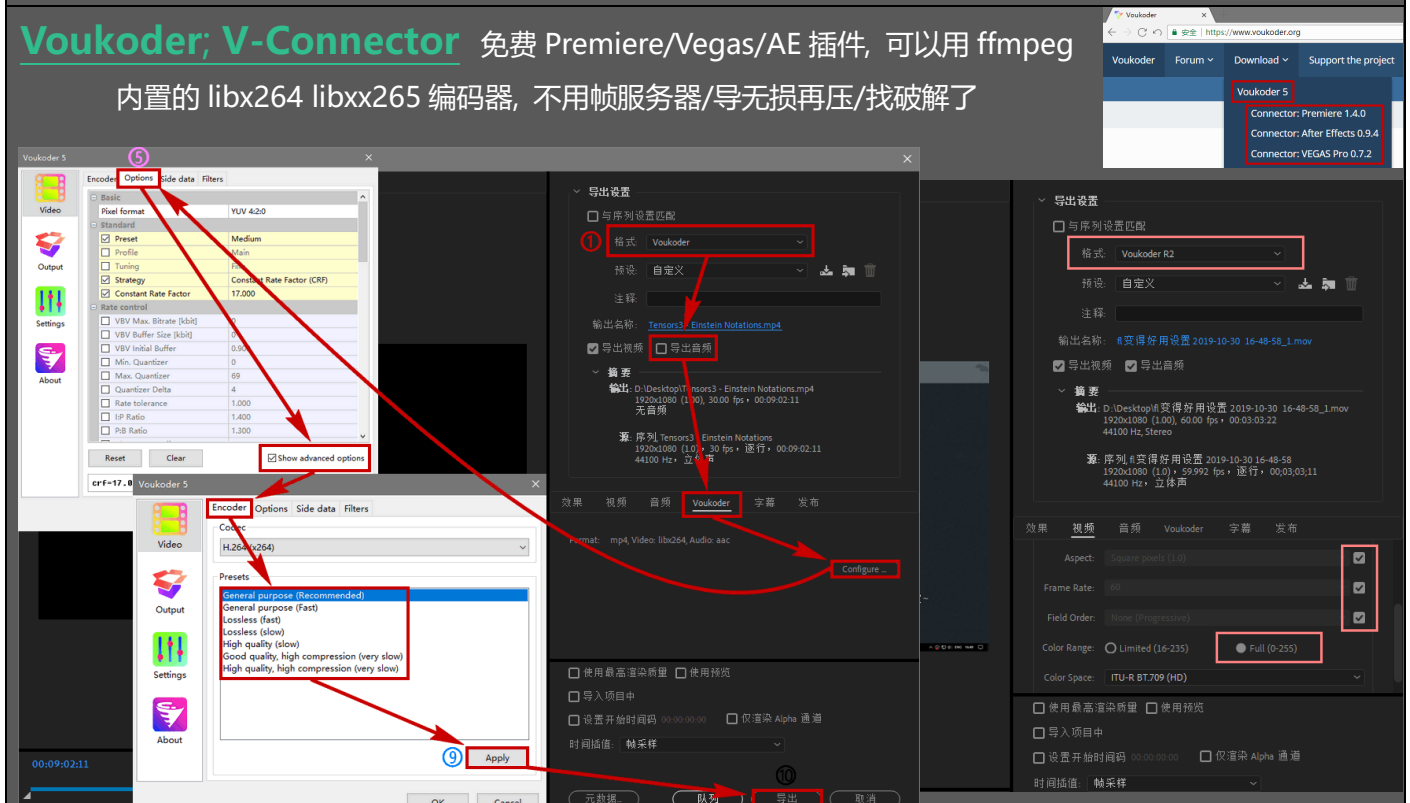
x264 tMod by jspdr √lavf 编解码, 支持 MCF 线程管理库(比 posix 和 win32 性能更好)

x264 7mod 谷歌盘/百度云 √lavf 编解码, √hqdn3d 降噪

ffmpeg(全系统): 备用地址 ottverse.com/ffmpeg-builds

Voukoder, V-Connector 免费 Premiere/Vegas/AE 插件, 可以用 ffmpeg

内置的 libx264 libxx265 编码器, 不用帧服务器/导无损再压/找破解了



去可变帧率

ffmpeg 去 VFR: 防止录像抽干手机电池的技术但差在兼容, 编辑时需转换. 手机先进则建议不用

pipe 的用法: "1. exe [输入] [参数] - | 2. exe [参数] [输出] -". 其中 "- | -"就是传递参数, 第一个 "-"代表输出, 第二个 "-"代表输入. 而 "- | - | - | -"就可以让同一个文件经过多个程序、(= ω · ω · ω))

```
(仅 pipe) ffmpeg -i 输入 -f yuv4mpegpipe - | x264 [参数] --demuxer y4m --output "输出"
(去 vfr)  ffmpeg -i 输入 -y -vf fps=60 -hide_banner -f yuv4mpegpipe - | x264 [参数] --
demuxer y4m --output "输出"
```

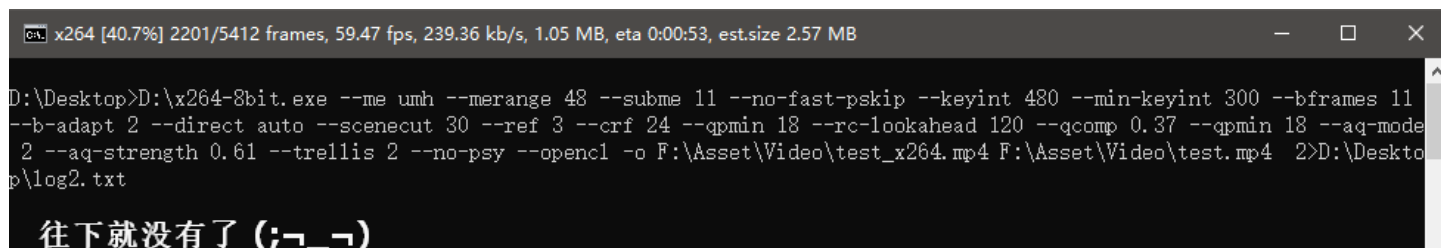
-f yuv4mpegpipe<预设字符>ffmpeg pipe 输出封装格式, 此处设为 yuv for mpeg

-i<字符>ffmpeg 输入参数, **-y**<开关>不询问, 直接覆盖掉同名的文件

-hide_banner<开关>ffmpeg 不显示 banner 信息, 减少 cmd 窗口阅读量

--demuxer y4m<预设字符>x264 pipe 解封装格式, 此处设为 yuv for mpeg

Win-CMD/Linux-bash 输出 log 日志



- Windows CMD: x264.exe[参数] 2>C:\文件夹\日志.txt [参数还可以写在右边]
- Linux Bash(或其它): x264 [参数] 2>&1 | tee /home/文件夹/日志.txt

附录

Potplayer 播放器音量忽大忽小 右键→声音/音讯→声音处理→反勾选标准化/规格化即可, 但建议用 mpv

CMD 窗口操作技巧**%~dp0** "%~"是填充字的命令(不能直接用于 CMD). d/p/0 分别表示 drive 盘/path 路径/当前的第 n 号文件/盘符/路径, 数字范围是 0~9 所以即使输入"%~dp01.mp4" 也会被理解为命令 dp0 和 1.mp4

这个填充展开后可能是"C:\"+ "...\"+1.mp4, 路径取决于当前.bat 所处的位置, 这样只要.bat 和视频在同一目录下就可以省去写路径的功夫了

若懒得改文件名参数, 可以用%~dpn0, 然后直接重命名这个.bat, n 会将输出的视频, 例子: 文件名=S.bat → 命令=--output %~dpn01.mp4 → 结果=1.mp4 转输出"S.mp4" (/·ω·)/^

.bat 文件操作技巧 .bat 中, 命令之后加回车写上 pause 可以不直接关闭 CMD, 可以看到原本一闪而过的报错(╯_╰)

.bat 文件存不了 UTF-8 字符 在另存为窗口底部选择 UTF-8 格式

UTF-8 .bat 文件中文乱码 开头加上 chcp 65001, 打开 cmd--右键标题栏--属性--选择

.bat 文件莫名其妙报错 Windows 记事本会将所有保存的文件开头加上 0xefbbbf, 要留空行避开

CMD 操作技巧: 换色, 试试这些命令: color B0; color E0; color 3F; color 6F; color 8F; color B1; color F1;

color F6; color 6; color 17; color 27; color 30; color 37; color 67

命令行报错直达桌面, 无错则照常运行: [命令行] 2> [桌面]\报错.txt

生成透明 rawvideo: ffmpeg -f lavfi -i "color=c=0x000000@0x00:s=sntsc:r=1:d=1,format=rgba" -c:v copy output.avi

PowerShell 内实现 UNIX pipe: 由于 PowerShell 内部跑完整个 pipe 再导出结果的机制不适用于程序间的 pipe 操作, 因此需要用 cmd /s /c --% 以在 PS 内部调用 CMD, 例:

```
cmd /s /c --% "D:\ffmpeg.exe -loglevel 16 -hwaccel auto -y -hide_banner -i `".\导入.mp4`" -an -f yuv4mpegpipe -strict unofficial -pix_fmt yuv420p - | D:\x265.exe --preset slow --me umh --subme 5 --merange 48 --weightb --aq-mode 4 --bframes 5 --ref 3 --hash 2 --allow-non-conformance --y4m - --output `".\输出.hevc`""
```

处理图像序列

一组连续图像, 格式不限. 自带 lavf 的 x264/5 可根据 Win/Linux 中的序号表达式, 设定输入文件名 + **%0Nd** 序号范围实现文件导入, **0** 代表编号前有空白补位 0; **N** 代表数位. Win/Linux 的区别仅为路径:

- x264.exe [参数] --fps 30 --output /home/tmp/导出视频.mp4 /home/tmp/图片%02d.png

ffmpeg 还可以用 image2 解码器搭配起始序号 **start_number** 和表达式类型 **pattern_type** 两个参数

编解图像序列; 其中 sequence 还是 %0Nd, glob 改用通配符*

- ffmpeg.exe -r 30 -f image2 -start_number 0 -pattern_type sequence -i C:\文件夹\图片%02d.jpg "C:\文件夹\导出视频.mp4"
- ffmpeg.exe -r 30 -f image2 -start_number 0 -pattern_type glob -i C:\文件夹\图片*.bmp "C:\文件夹\导出视频.mp4"

Windows 上对 **%0Nd** 的支持略差, 对图片文件名有要求也不便于跳过, 此时可以用 ffmpeg

concatenation 滤镜的编码方法. 首先创建 txt 列表:

- #Concat 格式: 只能用正斜杠以及单引号, 用#注释, Win/Linux 区别仅路径
- #可用 gif 格式的单图, 可混用图片格式, 分辨率一样就行, pdf 没试过
- file 'F:/Asset/2013/IMG_20130411_182854.jpg'
- file 'F:/Asset/2013/IMG_20130412_125214.png'
- file 'F:/Asset/2013/IMG_20130412_125307.tif'
- file 'F:/Asset/2013/IMG_20130508_155625.bmp'
- file 'F:/Asset/2013/IMG_20130508_155819.dpx'
- file 'F:/Asset/2013/IMG_20130613_133840.jpg'

- file 'F:/Asset/2013/IMG_20130727_191243.ext'
- file 'F:/Asset/2013/IMG_20130727_191255.isa'

帧率未知, 以及不确定能不能跑通时, 就做一个长 10 帧左右的列表副本来生成一段未压缩视频流排查

- ffmpeg.exe -loglevel 16 -hwaccel auto -y -hide_banner -f concat -safe 0 -r 帧率 -i "C:\导入图片列表.txt" -video_size 2560x1440 -an -pix_fmt yuv420p "C:\导出视频帧列表_帧率 x.yuv"

-f concat 以选择滤镜, **-safe 0** 以支持绝对路径(从根目录开始的路径), **-r** 决定帧率, **-video_size** 淘汰

了原本的-s 决定分辨率(解决图片格式支持奇数分辨率, 但视频不支持的冲突), **-an** 关闭音频编码, **-**

pix_fmt 选择和图片格式匹配的色彩空间. 一般情况下可以直接用视频播放器打开, 且应正确显示帧率

上方命令跑通, 或确认了帧率后, 就能使用 ffmpeg-libx264/5, 从急用版教程里找到对应参数来编码了

- ffmpeg.exe -loglevel 16 -hwaccel auto -y -hide_banner -f concat -safe 0 -r 15 -i "C:\导入图片列表.txt" -video_size 2560x1440 -an -pix_fmt yuv420p -x264-params "rc-lookahead=33: me=umh:bframes=12:b-adapt=2: subme=9:merange=48:fast-pskip=0:direct=auto:weightb=1:keyint=230:min-keyint=3:ref=3: crf=19:qpmin=9:chroma-qp-offset=-3:aq-mode=3:aq-strength=1.1:trellis=2:deblock=0,-1:psy-rd=0.5:0.3:nr=4" "C:\导出视频帧列表_x264_帧率 15.mp4"
- ffmpeg.exe -loglevel 16 -hwaccel auto -y -hide_banner -f concat -safe 0 -r 15 -i "C:\导入图片列表.txt" -video_size 2560x1440 -an -pix_fmt yuv420p -x265-params "ctu=64:tu-intra-depth=4:tu-inter-depth=4:limit-tu=1:me=star:subme=3:merange=48:weightb=1:ref=3:max-merge=4:open-gop=0:min-keyint=3:keyint=230:fades=1:bframes=8:b-adapt=2:radl=3:constrained-intra=1:b-intra=1:crf=22.8:qpmin=8:crqpoffs=-3:ipratio=1.2:pbratio=1.5:rdoq-level=2:aq-mode=4:aq-strength=1:qg-size=8:rd=3:limit-refs=0:rskip=0:rc-lookahead=33:rect=1:amp=1:psy-rd=2:qp-adaptation-range=3:deblock=0,-1:limit-sao=1:sao-non-deblock=1:selective-sao=3:hash=2:allow-non-conformance=1" "C:\导出视频帧列表_x265_帧率 15.mp4"

如果要导出图片序列, 则使用 ffmpeg 的 image2 解码器来导入, 指定图片格式导出即可:

- ffmpeg.exe -loglevel 16 -hwaccel auto -y -hide_banner -i 导入视频.avi -r 1 -f image2 -start_number 0 -pattern_type sequence -fps_mode cfr /home/tmp/图片%03d.jpg

IVTC, 分行转逐行

处理分行扫描源时, 注意帧数 fps 的参数实际上代表 $2 \times \text{fps}$ 的场数. 将逐行 fps 转换到分行的变换处理

一般叫 pulldown/up. 往回转就是 Inverse Telecine, IVTC. IVTC 的通用解是找出并删掉重复场, 重组

回完整帧. 进一步修复包括去梳 De-comb 和降噪滤镜来缓解合成出的横线失真. ffmpeg 用例如下:

- -vf "fieldmatch=mode=pc_nub:combmatch=full:blockx=16:blocky=24:combpel=128,yadif" -r 24

A:B Pulldown / Telecine 胶片放映机按照从上往下的顺序曝光胶卷得到视频帧, 所以"下"就是视频的"后", pulldown 就是后延. 逐行帧变成 NTSC/PAL 电视的分行后即上-下场 top-bottom field, 因此每帧占据 2 场(A=2 或 B=2). 此时将 A 或 B 多延后一场(A=3 或 B=3), 把后面的 TF 或 BF 挤兑一个渲染窗口, 从而让 24fps 的逐行电影拉伸到 PAL 25fps/50 场或 30fps/60 场上播放

2:2/2:2:2:2 (Not) pulldown

和游戏机一样没有不存在"一帧占三场"的情况, 根据 PAL 制式要加速匹配电视, 所以视频略快于音频.

注意音频流被加速(重采样)以匹配视频速度的情况. 如果是游戏机原生渲染的帧, 则找到并关闭游戏的

逐行转分行滤镜(顺便节约算力)就是画质最高的"转逐行", 不能重录的话用 yadif 滤镜, 或画质略好的

nnedi 转逐行

- -vf "nnedi=weights=C:\[下载](#)
[\nnedi3_weights.bin](#):field=af:nsize=s48x6:nns=n128:qual=slow:etype=mse:pscrn=new3"
- -vf "yadif=deint=1"

3:2/3:2:3:2 pulldown

第 1 帧拷一上场, 第 3 帧拷一下场. IVTC 过程就是每 5 帧(10 场)删除场 5 和场 10, 然后重组回逐行帧. IVTC 可以用上面的 fieldmatch 滤镜, 也能用 x264/AviSynth 的

Select_every 滤镜手动删除, 然后 yadif/nnedi 转回逐行, 再标注帧率 24 即可

2:3/2:3:2:3 pulldown 相比 3:2 pulldown 延后一个场. 见右图第 2 项

3:3:2:2/2:3:3:2/3:2:2:3/2:2:3:3 相当于 2:3 和 3:2 的四种排列组合, 比较过分

2:2:2:4/2:2:4:2/2:4:2:2/4:2:2:2 这样播放会让延迟更明显, 所以不怎么常见

5:5/5:5:5:5 pulldown 从上面的例子显而易见, 就是每帧五场, 但这样播放会大幅提高

延迟而很少见, 注意音频流被减速(重采样)以匹配视频速度的情况

2:2:2:2:2:2:2:2:2:2:3 (Euro) pulldown

24fps 电影格式每 12 帧拷一上场, 每 24 帧拷一下场, 加快 25/24 倍得到 25fps 或 50

场/秒的 PAL 视频. IVTC 过程就是每 25 帧(50 场)删除场 25 和场 50, 然后重排上下场:

- [ffmpeg 3232 pulldown IVTC] -vf "select='eq(mod(n\, 5))'"
- [x264/AviSynth 3232 pulldown IVTC] --vf Select_every(10, 1, 2, 3, 4, 6, 7, 8, 9)
-
- [ffmpeg Euro pulldown IVTC] -vf "select='eq(mod(n\, 25))'"
- [x264/AviSynth Euro pulldown IVTC]
- --vf Select_every(50, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49)

源视频用的哪个 pulldown?: 用播放器的逐帧查看功能识别. 如果片源里出现了多种

pulldown, 可能是 NTSC/PAL 源在没有恢复到电影 24fps 逐行前就被二次 pulldown

了. 这是片源方的恶性挑衅最好的处理是换源

TF	BF	Φ
A01	A02	1
A01	B04	2
B03	C06	3
C05	C06	4
D07	D08	5
3:2:3:2		
A01	A02	1
B03	B04	2
B03	C06	3
C05	D08	4
D07	D08	5
2:3:2:3		
A01	A02	1
B03	B04	2
B03	C06	3
C05	D08	4
D07	D08	5
2:3:3:2		
A01	A02	1
A01	B04	2
B03	C06	3
C05	D08	4
D07	D08	5
3:2:2:3		
A01	A02	1
B03	B04	2
C05	C06	3
C05	D08	4
D07	D08	5
T2:2:3:3		
AQA01	A02	1
BQA01	B04	2
CB03	B04	3
DC05	C06	4
ED07	D08	5
E03:3:2:2		
G1A01	A02	1
H1B03	B04	2
I1C05	C06	3
J1D07	D08	4
K1D07	D08	5
K22:2:2:4		
L2		
L22:2:4:2		
M2		
M22:4:2:2		
N2		
N24:2:2:2		
O29	P32	16
P31	Q34	17
Q33	R35	18
R35	S38	19
S37	T40	20
T39	U42	21
U41	V44	22
V43	W46	23
W45	X48	24
X47	X48	25