VCB-Studio教程06: VapourSynth基础与入门

0. 前言

BDRip压制的核心是批处理，会使用avs/vs对片源做预处理，是一个动漫ripper的入门。历史上，AviSynth是最早成体系的，为动漫Rip设计的非编。而VapourSynth在2014年初基本完善可用，现在逐步替代AviSynth。

想学习的人都会碰到这个问题：如果我只学一个，我该学哪个？如果我两个都学，我需要先学哪个？解答这个问题就需要先对比一下avs和vs的优劣：

avs落后，vs先进；

avs教程多，vs教程少；

avs支持广，vs支持窄；

avs不规范，vs规范；

avs入门简单，深入难，vs入门难，深入简单

……

VCB-Studio系列教程选择先介绍VS，原因在于，VS是现在vcb-s的主要产能，而且vs的规范性使得介绍基础知识的时候，更容易让初学者理解和掌握。

VapourSynth主页：<http://www.vapoursynth.com/>

官方使用文档：<http://www.vapoursynth.com/doc/>

在线词典：<http://dict.cn/>

1. 简单的vs脚本

在之前的教程中，我们有给过例子：

import vapoursynth as vs

import sys

import havsfunc as haf

import mvsfunc as mvf

core = vs.get\_core(threads=8)

core.max\_cache\_size = 2000

source = "00001.m2ts"

ripped = "Symphogear Vol1-1.mkv"

src16 = core.lsmas.LWLibavSource(source,format="yuv420p16")

rip16 = core.lsmas.LWLibavSource(ripped,format="yuv420p16")

res = core.std.Interleave([src16,rip16])

res = mvf.ToRGB(res,full=False,depth=8)

res.set\_output()

作为Python的一个扩展，vs脚本本质上是Python的脚本。在最开始我们需要载入(import)各种库，除了必须的VapourSynth核心，还有mvf(maven's VapourSynth functions) 和haf(holy's VapourSynth functions)

core = vs.get\_core(threads=8)

core.max\_cache\_size = 2000

这两句是载入vs运行环境，并且指定最大使用线程数和内存（MB）

接下来的部分，vs主要依赖赋值语句完成。一个赋值语句的格式为：

变量 = 表达式

比如source, ripped, src16, res等就是变量。Python的变量不需要声明，自动会判断是视频系列(clip)，整数(int)，还是字符串(string)等类型。

表达式，则有多种形式：

. 直接赋值，比如source = "00001.m2ts", debug = True, res = dbed 这种直接用值来给定的，值可以是具体的数值，值可以是具体的数值，也可以是其他的变量（比如res = dbed, dbed就是另一个变量）；

. 简单运算，比如 strength = 80/100， output\_depth = debug?8:10 这样的运算；

这里详细讲一下表达式 A?B:C的计算。A叫做判断式， 必须是一个布尔类型的表达式（只有True/False, 或者1/0），B和C则是可能返回的值。

x = A?B:C 等效于 if (A) x=B; else x=C; 如果A成立，则x赋值为B，否则x赋值为C

比如说：

False?0:1 返回的是1，因为判断式不成立，所以返回两个值中的后者

debug?8:10 如果debug是True/1， 则返回8，否则返回10

x<10?10:x<100?100:200 是一个嵌套性的语句；拆开来看：

if (x<10) return 10;

 else if (x<100) return 100

 else return 200

当x小于10的时候，返回10；当x在10-99的时候，返回100，否则，返回200

. 函数赋值，res = core.std.Interleave([src16,rip16])， 这句就是调用core.std.Interleave() 这个函数，输入src16和rip16（严格来说，是它们用[]运算符，运算而出的结果，那就是它们的顺序组合），作为输入变量，来计算一个新的值。

最后，vs的输出，通过set\_output()来完成。res.set\_output()就是输出res这个值。

2. VS函数的调用

可以想象，vs脚本的本体是由大量的函数调用实现的。函数的调用方式一般为：

domain1.domain2…FunctionName(parameter1, parameter2,……)

domain是函数所在的库，比如Core.std.Interleave就是一层Core，二层std，下面才是函数名称Interleave。或者mvf.Depth(), 只有一层mvf.

parameters是函数输入的变量，数值不定。函数的输入，一般doc中有非常明确的规定。比如说根据LWLibavSource的doc:

LWLibavSource(string source, int stream\_index = -1, int threads = 0, int cache = 1, int seek\_mode = 0, int seek\_threshold = 10, int dr = 0, int fpsnum = 0, int fpsden = 1, int variable = 0, string format = "", int repeat = 0, int dominance = 1, string decoder = "")

LWLibavSource一共可以接受 source, stream\_index, …… decoder等14个输入；

这14个输入有着自己的类型要求，比如source要求是string, stream\_index要求是int，等等；

这14个输入并非在调用的时候都需要有赋值。除了source之外所有变量都有设定默认值/缺省值/default value，因此如果你不设定，这些输入自动设定为默认值。

在手动输入参数的时候，有两种方式：

1. 赋值性传递/关键字传递(keyword argument)，表现为A=B的形式，比如format="yuv420p16"。这样的赋值，系统会先去找函数输入中有无一个叫做A的参数，如果有，把B的值给A；

2. 直接传递/位置性传递(positional argument)，表现为直接放一个C。比如：

filename = "00001.m2ts"

src16 = core.lsmas.LWLibavSource(filename,format="yuv420p16")

这里函数内第一个输入的是filename，它没有以赋值性的语句输入，那么系统判定为直接传递，传递的内容是source这个表达式，表达式求值得出，filename是一个变量，值为"00001.m2ts"。所以系统会把"00001.m2ts" 传递给函数第一顺位的输入，也就是source。

以上的脚本还等同于：

src16 = core.lsmas.LWLibavSource("00001.m2ts",format="yuv420p16")。这种是直接把值作为表达式，而不是再用变量传递；

filename = "00001.m2ts"

src16 = core.lsmas.LWLibavSource(source=filename,format="yuv420p16")

或者src16 = core.lsmas.LWLibavSource(source="00001.m2ts",format="yuv420p16")

这种就是用赋值性传递，来干相同的事情。

source = "00001.m2ts"

src16 = core.lsmas.LWLibavSource(source=source,format="yuv420p16")

或者src16 = core.lsmas.LWLibavSource(source,format="yuv420p16")

这种写法也是合法的。注意这里source=source的意义：前一个source，系统会在函数输入中寻找对应，后一个source，系统会在当前脚本中做表达式求值。同理，直接写一个source，系统会先计算source作为一个表达式的值，再去以直接传递的方式去传递给函数。

3. 函数中参数传递的机制

vs关于函数变量传递，遵循着这样的机制：

1. 所有直接传递，必须在变量性传递之前。比如LWLibavSource(source,format="yuv420p16")是可以的，而LWLibavSource(format="yuv420p16",source)在syntax上出错。

2. 传递的过程中，先把所有变量性传递的参数给传递好，剩下没有被传递的参数，一个个按顺序，把直接传递的值给赋值过去。比如说core.std.MaskedMerge这个函数：

std.MaskedMerge(clip clipa, clip clipb, clip mask[, int[] planes, bint first\_plane=0])

从doc看，这个函数输入5个input: clipa, clipb, mask, planes, first\_plane. 其中前三个没有默认值，因此必须在调用的时候输入；后两个用[]裹起来，意思是可以不输入。first\_planes有默认值0， planes因为是需要一个整数数组，长度未知因此没有默认值（读了doc就知道它在调用时候，知道了数组实际长度后，默认的赋值。）

假设我们已经算好了edge, nonedge, mask这三个clip：

core.std.MaskedMerge(nonedge, [0,1,2], False, mask=mask, clipb=edge)

系统会先把mask代表的值，传递给函数中mask这个input，然后把edge代表的值，传递给clipb这个input；

剩下clipa, planes, first\_plane这三个没有输入的input，系统把nonedge传递给clipa, [0,1,2]传递给planes, False传递给first\_clip。 所以它等效为：

core.std.MaskedMerge(clipa=nonedge, clipb=edge, mask=mask, planes=[0,1,2], first\_plane=False)

或者 core.std.MaskedMerge(nonedge, edge, mask, [0,1,2], False)

如果传递的过程中，某一个input被输入了两次（这种情况只可能是重复用赋值性传递来输入，想想为什么？），那么vs会报错；

如果传递完毕后，必须输入的变量（doc中没有默认值，也没有用[]框起来）并未完全赋值，那么vs也会报错；

传递过程中，如果输入值的类型跟变量定义类型不匹配，比如你把一个字符串给了整数类型的变量，vs也会报错。

从代码可读性和减少出错的角度说，应该永远鼓励赋值性传递。

VS函数传递，可以允许嵌套以及串联。比如说：

src16 = core.lsmas.LWLibavSource(source="00001.m2ts")

res = core.rgvs.RemoveGrain(src16, 20)

用嵌套的写法：

res = core.rgvs.RemoveGrain(core.lsmas.LWLibavSource(source="00001.m2ts"), 20)

就是直接将函数作为一个表达式

用串联的写法(必须是vs规范的函数，比如都是core下面的)：

res = core.lsmas.LWLibavSource(source="00001.m2ts").rgvs.RemoveGrain(20)

串联的时候，后续的core可以省略。效果是将前面生成的clip，作为下一个函数，第一个直接传递的值。

4. 一些简单的视频编辑

在本章中，我们讲述一些vs中常见的用法，方便大家学习和上手

4.1 裁剪和缩放

裁剪靠的是std.CropRel, 缩放靠的是resize.Spline36

doc分别为：

<http://www.vapoursynth.com/doc/functions/crop.html>

<http://www.vapoursynth.com/doc/functions/resize.html>

假设我们读入一个原生4:3，通过加黑边做成1920x1080的视频，我们先把它切割成1440x1080（就是左右各240个像素），然后缩放成720p:

src = …

cropped = core.std.CropRel(clip=src, left=240, right=240)

res = core.Resize.Spline36(clip=cropped, width=960, height=720)

res.set\_output()

看doc就知道，恰好所有的输入都是滤镜要求的前三顺位，所以上述代码可以简化为（用串联写法）：

src = …

core.std.CropRel(src, 240, 240).Resize.Spline36(960, 720).set\_output()

4.2 分割与合并

分割靠的是std.Trim (<http://www.vapoursynth.com/doc/functions/trim.html>)

合并靠的是std.Slice(<http://www.vapoursynth.com/doc/functions/splice.html>)

**以下是整个vcb-s教程体系中，我们对帧数标号的规定：**

在绝大多数场合下（除了mkvtoolnix），视频的帧数是从0开始标号的。简单说，如果一个视频有1000帧，那么所有帧的标号为：

0, 1, 2…999

mkvtoolnix是从1开始标号的: 1, 2, 3…1000。然而，除非指定了是mkvtoolnix，任何讨论都假设帧数从0开始标号。

无论从0还是1开始标号，总帧数=末号-首号+1

如果我们说从a帧到b帧，我们默认是包括首尾的。比如20-100帧，就是20,21,…99,100帧，一共是100-20+1=81帧。

回到Trim的用法：

std.Trim(clip clip[, int first=0, int last, int length])

clip是必须输入的，first指定从哪一帧开始切割（默认是0），然后last和length两个指定一个。（doc中告诉你如果两个都指定了会报错。）如果不用赋值传递,比如std.Trim(clip, 20,100)，那么输入的100会被判为last，因为last的序位在前

确定了first和last，Trim会切出clip从first到last的所有帧，注意是包括首尾的，总帧数为last-first+1;

如果是指定length，Trim会切出clip从first开始，一共length帧，这时候等效于指定last为length+first-1。

vs中有继承自Python的语法糖（Syntactic Sugar）帮助你简单的写Trim:

video = clip[20:101]

相当于video = core.std.Trim(clip,20,101-1)

注意Trim里面写法是从x到y，语法糖写法是[x:y+1]，然而这两个效果都是切出从x到y这y-x+1帧。

因为在avs里面，切割也是用trim这个函数名称，写法和规则也是从x到y，所以实际操作时候，**分割视频建议不要采用语法糖写法**，而是坚持用传统的用法。不然容易造成队友的混淆（因为量产中需要队友自己改Trim参数）

合并的Splice比较简单：

longvideo = core.std.Splice([video1, video2])

就是把video1和video2按照顺序前后合并。这么写要求video1和video2的尺寸和像素类型(比如同为YUV420P8)必须一致，帧率等其他性质可以不一致。

如果你要强行把两个尺寸或者像素类型不同的视频合并，vs也能办到：

longvideo = core.std.Splice([video1, video2], mismatch=1)

不过实际操作中少有这样的例子就是了，毕竟不同尺寸和类型在一起加工限制很多，一般都需要你先转换统一格式，再合并。

如果要合并多个视频，只要增加数组就好了：

longvideo = core.std.Splice([video1, video2, video3])

**合并的写法更推荐用语法糖（avs里面就是这种写法）：**

longvideo = video1+video2+video3

简单明确易懂。这时候要求video1, video2 和 video3的尺寸和像素类型必须一致，帧率等其他性质可以不一致，相当于默认mismatch=False。

4.3 简单的降噪，去色带和加字幕

降噪用的是std.RemoveGrain(), 去色带用的是f3kdb.Deband(), 加字幕用的是assvapour.AssRender()

到这个点总该会自己去找doc了吧。提示: 先从vs doc主页右下方的search入手，找不到就Google关键字：滤镜 vapoursynth

src = …

nr = core.std.RemoveGrain(src, [11,4])

dbed = core.f3kdb.Deband(nr,12,32,24,24,0,0)

res = core.assvapour.AssRender(dbed, "xxx.ass")

res.set\_output()

尝试自己找到doc，对应着看看，每一个参数都是输入给哪个input，这个input的意义（至少在doc里字面意义）是什么。

5. VS里面对视频性质(clip property)和帧性质(frame property)的读取

vs里面可以直接读取一些关于视频和帧本身的性质，比如说视频的总长度，帧率，一帧的长宽，类型等。这部分在[http://www.vapoursynth.com/doc/pythonreference.html#classes-and-functions](http://www.vapoursynth.com/doc/pythonreference.html%23classes-and-functions)中有详细解释，我们只列举最常用的几个：

clip.num\_frames 返回clip的总帧数。所以要切掉视频的首帧（第0帧），可以这么写：

res = core.std.Trim(clip, 1, clip.num\_frames-1)

clip.width, clip.height 返回clip的宽和高。比如我们想缩放到1/2大小：

res = core.Resize.Spline36(clip, clip.width//2, clip.height//2)

#注意这里//2是做整数除法，出来的类型是int，否则/是浮点数除法，出来类型是float。

#而Resize类型滤镜要求输入int类型的数据。

顺道说一句关于Python的注释，#在Python中是注释掉后面一行字的作用，相当于C/C++中的//

如果想要大面积注释，类似/\*\* \*\*/

可以用''' '''

6. 选择分支和Python中的缩进（indentation）

Python作为一个全能性编程语言，对很多现代编程中的概念都支持，最基础的选择分支和循环等自然不在话下。这里我们举个列子：

src16作为源，res作为处理后准备输出的clip。我们设置一个开关Debug，如果Debug=1/True则将src16和res交织输出，并转换为8bit RGB， 否则将res转为YUV - 10bit准备送给编码器：

Debug = 0

if Debug:

 res = core.std.Interleave([src16,res]) #Interleave是将输入的视频一帧帧间隔显示

 res = mvf.ToRGB(res,full=False,depth=8) #ToRGB的作用是转为RGB，bitdepth已经指定为8

else: res = core.fmtc.bitdepth(res,bits=10) #bitdepth是做精度转换

如果用类似C的伪代码写，大概风格为:

Debug = 0;

if (Debug) {

 res = core.std.Interleave([src16,res]);

 res = mvf.ToRGB(res,full=False,depth=8);

} else

res = core.fmtc.bitdepth(res,bits=10);

可见，Python里面没有类似{}来把一段代码组合起来，Python用的是缩进。不同缩进层次来区分不同组合。比如说：

res = core.std.Interleave([src16,res])

 res = mvf.ToRGB(res,full=False,depth=8)

这两句都是通过一个tab来缩进，所以这两个相当于被大括号给框住，成为一段。如果是：

Debug = 0

if Debug:

 res = core.std.Interleave([src16,res])

res = mvf.ToRGB(res,full=False,depth=8)

执行逻辑就截然不同了；res = mvf.ToRGB(res,full=False,depth=8)这句是跟if并列的，一定会在if语句做完之后被执行。

Python对缩进非常严格，任何不匹配都会报错。注意tab和空格不可等同，哪怕在你看来4个空格等于一个tab。

其他一些VS的高级用法，比如runtime机制，比如自定义函数，我们会在以后的教程中详细说。