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 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。