Pipeline

> Pipeline，像写流程图一样编程。

> @Author: QiuQi
>
> @Email: quqihyy@163.com
>
> @GIT: https://github.com/quiqi/pipeline-base

Pipeline

0. 前言：

无论什么样的程序，从逻辑上都可以表示成一张流程图，比如一个我们希望在电脑上实现一个叫《镜子》的程序：从电脑摄像头读取一张图片，然后将图片进行一次翻转，最后再显示到窗口中：

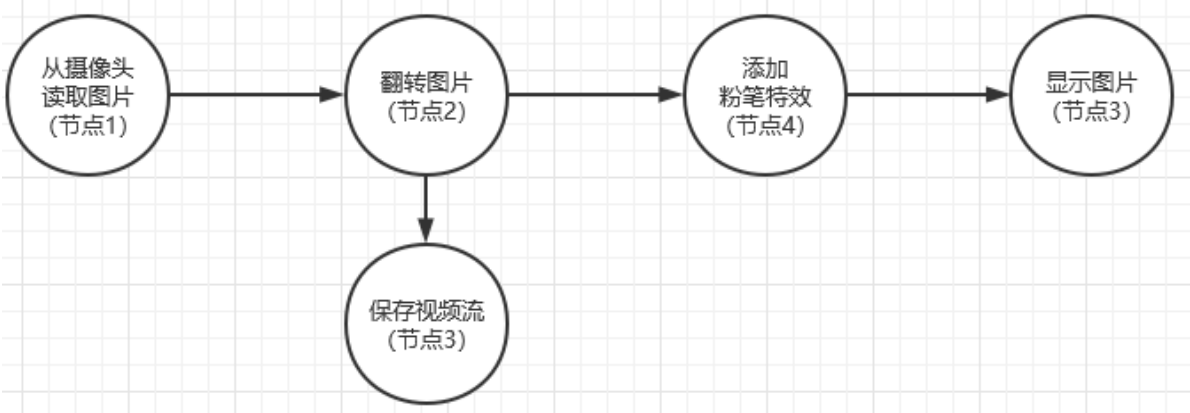
[图0.1]镜子1.0的流程图

这是一个非常简单的程序，如果你的python环境中有opencv，可以用很简单的代码实现上面的程序：

```
# 代码1：用opencv实现《镜子》程序
import cv2       # 导入opencv包


if __name__ == '__main__':
    cap = cv2.VideoCapture(0)    # 获取摄像头，参数0表示本地第一个摄像头

    while True:                  # 不断从摄像头读取数据并显示
        ret, img = cap.read()        # 1. 从摄像头读取一帧图片
        img = cv2.flip(img, 1)       # 2. 将图片翻转
        cv2.imshow('test1', img)     # 3. 将图片显示到 'test1' 窗口
        cv2.waitKey(1)  # 等待一个非常短的时间，让cv2.imshow有时间去绘制图像
```

[代码0.1]《镜子1.0》的脚本实现

这样的程序编写非常简单，但不利于代码的复用和团队的合作，也不符合软件工程上 " 对修改封闭，对拓展开放 " 的基本原则。比如我希望为《镜子》程序添加一个粉笔画特效，以及一个保存视频的功能，在流程图上的修改非常简单：



[图0.2]镜子2.0的流程图

但对于代码实现来说，我们需要在原有代码的基础上进行修改。这显然不符合 "开闭原则"，而且还会产生各种问题，比如随着需求的增加，代码越来越多，对代码的有效组织变得越来越难等等。

这时，如果有一个工具让修改代码的过程变得像修改流程图一样清晰明了，或许能大大方便我们的编程。 Pipeline就是一个这样的后端框架，它支持模块化开发，支持多进程，对团队合作友好，非常适合一些流式处理程序的开发。总之，如果用pipeline来编写《镜子1.0》是这样的：

```
# 代码2：用piple实现镜子,完整代码在 code0.2.py 中
if __name__ == '__main__':
    mirror = NodeSet([      # 实例化一个名为 'mirror' 的任务，该任务有三个节点：
        # Node(节点名称，后继节点，节点功能)
        Node('node1', subsequents=['node2'], worker=ReadCamera()),
        Node('node2', subsequents=['node3'], worker=Flip()),
        Node('node3', worker=ShowImg())
    ])

    while mirror.switch:
        mirror.run(Frame(end='node1'))
```

我们可以很轻松的看出，以上代码是几乎就是 [图0.1]的文字版，即使我不进行任何说明，对照[图0.1] 大家也能很快明白这两者之间的关联，当然，`ReadCamera(), Flip(), ShowImg()` 是需要自己编写的，如果大家熟悉了这种编程方式，并不会觉得这是一个麻烦的事情。 我们将在 “2.1.3” 小节中给出具体的实现。

在pipeline框架下，如果希望实现程序《镜子2.0》，我们只需要实现对应两个节点的功能，并修改代码为 [代码0.2]，就能实现功能的添加：

```
# 该部分的全部源码在 code0.2.py 中
if __name__ == '__main__':
    mirror = NodeSet([
        Node('node1', subsequents=['node2'], worker=ReadCamera()),
        Node('node2', subsequents=['node4', 'node5'], worker=Flip()),
        Node('node4', subsequents=['node3'], worker=ChalkEffects()), # 实现粉笔特效
        Node('node5', worker=Save()),   # Save()是 pipeline框架中自带的，无需自己实现
        Node('node3', worker=ShowImg())
    ])

    while mirror.switch:
        mirror.run(Frame(end='node1'))
```

[代码0.2] 《镜子2.0》的pipeline实现

就像修改流程图那样简单。

也许到此你还是不明白pipeline到底是什么，没有关系，在接下来的部分，我们将从安装开始，一步一步向大家介绍这个框架是如何使用的。

在正文的第一部分，我们将介绍pipeline的安装方式；在第二部分，我们将以三个简单的小任务让大家快速上手 pipeline；文章的第三部分是pipeline的说明文档，将依次介绍pipeline的基本逻辑、pipeline中类和函数的使用方式、以及在pipeline建议的编程规范等等；文章的最后一段将为大家介绍几个开发案例，以便更清晰的说明pipeline的使用方法。

[图0.3] 《镜子2.0》的输出

1. 安装

1.1 运行环境：

1. 操作系统：Ubuntu \ Windows
2. 编程语言：Python3

1.2 安装步骤：

1. 在GitHub上下载该项目：https://github.com/quiqi/pipeline-base.git
2. 解压后进入dist文件夹中，安装最新版的 *.whl包：

```
pip install pipeline-1.1-py3-none-any.whl
```

3. 安装完毕后就可以用 `import` 导入pipeline包了，如果能运行下面的代码，就说明安装成功了

```
>>> import pipeline.core as core
>>> a = core.Node('node1', ['node2'])
>>> b = core.Node('node2')
>>> c = core.NodeSet([a,b])
```

[图1.1] 导入pipeline包并测试是否安装成功

2. 快速上手

在Pipeline的开发中，一条非常重要的理念就是：**用简单的代码做简单的事。**你不需要学会Pipeline中的全部内容，只需要知道你需要用到的部分，就可以可以使用Pipeline实现你想要的东西。该部分为大家准备了三个循序渐进的小任务，完成这些任务，您就基本掌握Pipeline了。

2.1 将流程图抽象为代码

2.1.1 节点抽象为代码

对于一个从来没有尝试过编程的人来说，他仍然可以通过流程图的方式表达自己需要的程序。这就是程序的逻辑部分，理论上而言，只要有这个部分，一个程序的输入、输出、功能等都已经被表述清晰了，其他具体的实现都是细节。

[图0.1]镜子1.0的流程图

以[图0.1]为例，每个节点可以抽象为三个信息：**（节点名称，节点后继，节点功能）**。比如第一个节点可以被抽象为：

节点名称	节点后继	节点功能
节点1	节点2	从摄像头读取图片

如你说想到的，在pipeline中，一个节点正是被抽象为如下三个参数，同样以第一个节点为例，它在代码中会被抽象为一个节点对象：

```
import pipeline.core as core

# 节点1：节点名称：'node1'，节点后继：['node2']，节点功能：ReadCamera()
node1 = core.Node('node1', ['node2'], worker=ReadCamera())
```

[代码2.1]：构建一个节点对象

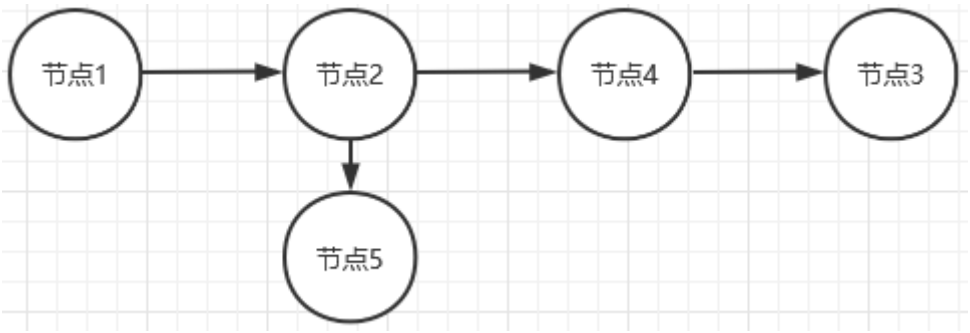
[代码2.1.1]中出现的 `Node(name, subsequents, worker)` 是节点类，用于构建节点对象，是pipeline中最核心的类之一，它被放在 `pipeline.core` 包中。代码中的 `ReadCamera` 是“从摄像头读取图片”的具体实现类，是 `Worker` 类的子类，用于实现具体的功能。

我们将一个流程图上的信息分为两部分：由节点名称和节点功能构成的**结构部分**，以及像 `ReadCamera` 这样的**功能部分**。在接下来的两个小节中，我们将依次介绍如何搭建一个流程图的“结构部分”和“功能部分”

> 在 3.1 小节中，这两个部分将分别对应框架中的“节点层”和“应用层”。

2.1.2 实现流程图的结构部分

如果将一个流程图的功能部分去掉，一个流程图就被简化为了一个有向图，比如将流程图 [图0.2] 的功能部分去掉，就得到了一个下面这样的有向图：



[图2.1] 从 [图0.2]抽象得到的有向图

按照上一小节中说到的，每个节点都可以被抽象为一行代码，在这个小节，我们将介绍一种可以将这些节点组合在一起的容器： `NodeSet`，通过它，可以将多个节点组成一张图：

```
import pipeline.core as core

# 初始化每一个节点
node1 = core.Node('node1', ['node2'])
node2 = core.Node('node2', ['node4', 'node5'])
node3 = core.Node('node3')
node4 = core.Node('node4', ['node3'])
node5 = core.Node('node5')

# 将以上节点放入NodeSet中构建流程图
chart = core.NodeSet([node1, node2, node3, node4, node5])

# 生成帧对象，end='node1' 表示从名为'node1'的节点进入。
frame = core.Frame(end='node1')
fs = chart.run(frame)   # 运行一次该流程图
for f in fs:
    print(f.visited)
```

[代码2.2]：构建流程图的结构部分

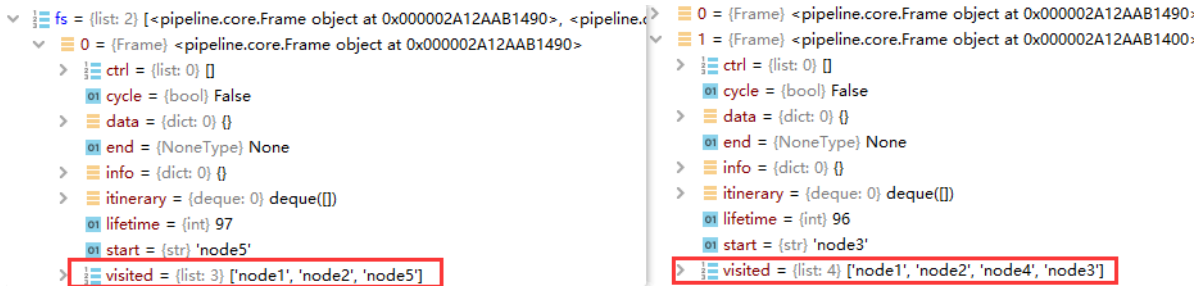
[代码2.2]中的 `chart` 对象就代表由 `node1~node5` 组成的流程图。同时从[代码2.2]中我们可以看到，一个节点的后继节点可以有多个，当后继节点参数缺省时，表示没有后继节点。

[代码2.2]中的最后一行，每调用一次 `chart.run(frame)` 函数，就表示该流程图被运行一次，其输入参数为一个数据帧。数据帧类 `Frame` 也是pipeline的一个核心类，是流处理架构中数据流动的载体。为了更加清晰的描述接下来的内容，我需要花费一些篇幅初略的介绍一下 `Frame` 在框架中的作用。

假设有5个小朋友在上课的时候想交流晚上放学后去哪玩，但是应为怕被老师发现不认真，不敢大声说话，于是通过递纸条的方式交流。如果将节点比作这五个小朋友，那么 `Frame` 就是纸条，它负责在不同节点之间传递信息。但与传纸条这个例子不一样的是：

- 节点中的帧 `Frame` 只会从一个节点传到其后继节点；
- 当一个节点有多个后继时，默认会将帧深拷贝成n份，发给每一个后继（n为后继的个数）。非默认情况暂时不提。

我们可以尝试用调试工具查看 `chart.run(frame)` 返回对象 `fs` 的具体内容，我们会发现，`fs` 是一个 `Frame` 对象的列表，列表中一共有两个帧，每个帧都会有一个 `visited` 变量，记录了这个帧访问过的节点：



[图2.2] fs两个帧中的 visited 变量

从上图可以看出，被传入 `chart.run(frame)` 的帧在节点2 `node2` 被复制为两个，一个访问完 `node5` 后被抛出，一个依次访问 `node4, node3` 后被抛出。

值得一提的是，`NodeSet` 类是 `Node` 类的子类，也就是说，`NodeSet` 也可以被当成一个 `Node` 对象被放入一个更大的 `Nond` 中实现套娃。

```
import pipeline.core as core
# 第一个NodeSet
chart1 = core.NodeSet([
    core.Node('head1', ['node1']),      # head1 一般起到跳板的作用
    core.Node('node1', ['node2']),
    core.Node('node2', ['node3', 'node5']),
    # core.Node('node2', ['node3', 'head2/node5'])
    core.Node('node3'),
    core.Node('node4')
])
# 第二个NodeSet
chart2 = core.NodeSet([
    core.Node('head2', ['node1']),
    core.Node('node1', ['node2']),
    core.Node('node2', ['node3', 'node4']),
    core.Node('node3'),
    core.Node('node5')
])
# 将两个NodeSet组成一个更大的NodeSet：chart
chart = core.NodeSet([chart1, chart2])
# 运行一次chart
fs = chart.run(core.Frame(end='head1'))
for f in fs:
    print(f.visited)
```

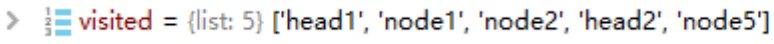
[代码2.3] NodeSet的套娃

[代码2.3] 将两个 `NodeSet` 套入了一个更到的流程图 `chart` 中，我们看到这两个不同的 `NodeSet` 有许多同名的节点，但丝毫不会影响程序的进行，因为当每一个节点发送帧数据时，会优先在最小的 `NodeSet` 结构中寻找，如果找不到才会去更上一级的范围中去寻找。

除此之外，相比于 [代码2.2]，每个小的 `NodeSet` 都有一个以 `head*` 命名的节点，这个节点一般不会完成具体的功能，而是作为数据进入该有向图的跳板，对于该有向图之外的节点而言，只有 `head*` 节点是可见的。

比如 `chart1` 和 `node2` 节点的的第一个后继 `node3` 可以在 `chart1` 内部找到，于是会优先发往内部的 `node3` 节点。可其第二个后继 `node5` 在 `chart1` 中无法找到，但也绝不会找到 `chart2` 中的 `node5` 节点。如果希望将帧发往 `chart2` 中的 `node5`，需要将代码改写成 `core.Node('node2', ['node3', 'head2/node5'])`。

修改后运行，`fs` 中会出现两个帧对象，其中一个的 `visited` 为：

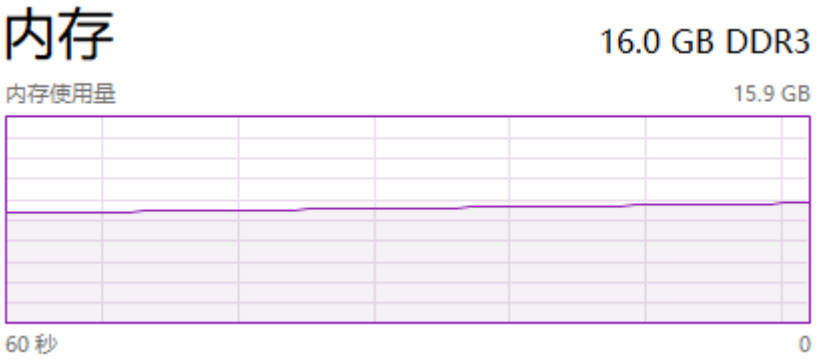


[图2.3] 跨过两个NodeSet的数据帧的访问记录

在大多数情况下，流程图中是不能有环的，因为这样会让数据帧在有向图中不断的兜圈子，如果这个循环中还有一个分支，那么每运行一次，流程图中的包的数量就会翻一倍，电脑内存会被快速消耗。

```
import pipeline.core as core

# 初始化每一个节点
node1 = core.Node('node1', ['node2'])
node2 = core.Node('node2', ['node3', 'node4'])
node3 = core.Node('node3', ['node1'])# 将数据发回node1形成死循环
node4 = core.Node('node4', ['node1'])# 将数据发回node1形成死循环
# node4 = core.Node('node4')

# 将以上节点放入NodeSet中构建流程图
chart = core.NodeSet([node1, node2, node3, node4])

frame = core.Frame(end='node1')      # 生成帧对象
fs = chart.run(frame)                # 运行一次该流程图
```

[代码2.4]一份有死循环的代码



[图2.4]运行[代码2.4]内存用量稳步上升

如果让 `node4` 不在将数据发回 `node1`，虽然仍让存在 `node1->node2->node3->node1` 的循环，但程序其实是可以结束的，因为默认一个帧数据只能被传递100次，超过100次则会被自动丢弃。

2.1.3 实现流程图的功能部分

如果说`Node`和`NodeSet`用于实现流程图的结构部分，管理数据帧(`Frame`)的流动，那么`Worker`就是用于实现数据帧`Frame`的加工。在正式开始介绍`Worker`类之前，需要先更详细的了解`Frame`类。

`Frame`类的成员变量并不多，其`__init__`函数源码如下：

```
class Frame:
    """
    帧类，数据流动的基本单位
    """

    def __init__(self, start=None, end=None, lifetime: int = 100, cycle: bool =
False):
        self.data = {}  # 数据字典：组件之间的数据交流载体
        self.info = {}  # 信息列表：为基础组件预留的接口，用户一般不要使用，以免发生错误
        self.ctrl = []  # 控制列表：控制数据，用于系统控制和人机交互
        self.visited = []   # 经过节点：该帧经过的节点的名称
        self.start = start  # 最后发出该帧的节点
        self.end = end  # 该帧目前的目标节点
        self.itinerary = collections.deque()  # 行程计划
        self.lifetime = lifetime    # 生存时间
        self.cycle = cycle # 是否为循环帧，如果该cycle=True,lifetime不起作用
```

[代码2.5-1] Frame类的 __init__ 函数源码

其中最常用的成员变量为`data`，这是一个字典数据，当一个节点需要给其后继节点传递数据时，一般会将数据放在`data`中。

让我们回到`Worker`类，读者可以在 pipeline/core.py 中找到它的源码，在此仅仅粘贴其最需要被用户知道的一部分。

```
class Worker(Model):
    ... 省略部分代码
    def process(self, frame: Frame):
        """
        承担功能实现的具体函数，一般在功能worker中被实现
        :param frame:帧数据
        :return:修改后的数据
        """
        return frame
    ... 省略部分代码
```

[代码2.5-2] Worker类的 process 函数源码

用户可以继承`Worker`类，并重写`process`函数而实现具体的功能，这么说可能有些抽象，让我们看具体的实例。

仍然以前言中的《镜子1.0》为例：

[图0.1]镜子1.0的流程图

我们需要实现三个功能：1. 从摄像头读取图片、2. 翻转图片、3. 显示图片。

我们的基本实现思路是：

1. 在节点1中，从摄像头读取数据放到数据帧的 `data` 字典中；
2. 节点2作为节点1的下游节点，可以获得节点1存放在 `data` 中的图片，并将该图片翻转；
3. 节点3将 `data` 中的图片显示。

具体实现如下：

```
import cv2
from pipeline.core import *
from pipeline.utils import *


class ReadCamera(Worker):
    """
    从摄像头读取图片
    """

    def __init__(self, name: str = 'camera', url: str = 0):
        super().__init__(name)
        self.camera = cv2.VideoCapture(url)

    def process(self, frame: Frame):
        # frame就是流经该组件的帧对象
        ret, img = self.camera.read()
        # 将读取到的图片放入 frame.data 的 'img' 的关键字下
        frame.data['img'] = img
        # 返回修改后的帧
        return frame


class Flip(Worker):
    """
    将图片翻转
    """

    def __init__(self, name: str = 'flip'):
        super().__init__(name)

    def process(self, frame: Frame):
        # 从 frame.data 的 'img' 的关键字下获得上游传来的图片，并翻转后放回
        frame.data['img'] = cv2.flip(frame.data['img'], 1)
        # 返回修改后的帧
        return frame
```

```
class ShowImg(Worker):
    """
    显示图片
    """

    def __init__(self, name: str = 'show_img'):
        super().__init__(name)

    def process(self, frame: Frame):
        # 显示 frame.data 的 'img' 的关键字下的帧
        cv2.imshow(self.name, frame.data['img'])
        cv2.waitKey(1)
        # 返回帧
        return frame


if __name__ == '__main__':
    mirror = NodeSet([
        Node('node1', subsequents=['node2'], worker=ReadCamera()),
        Node('node2', subsequents=['node3'], worker=Flip()),
        Node('node3', worker=ShowImg())
    ])

    while mirror.switch:
        mirror.run(Frame(end='node1'))
```

[代码2.6] 《镜子1.0》 的pipeline实现

从[代码2.6]可以看到，相比于[代码0.1]，在代码编写的简便程度上，pipeline实现可能并不占据优势，但pipeline的优点是模块化和可拓展性，代码的可读性和复用性也更强，毕竟谁会看不懂流程图呢?

在pipeline中，我们将一个实现具体功能的 `Worker` 子类称为组件。

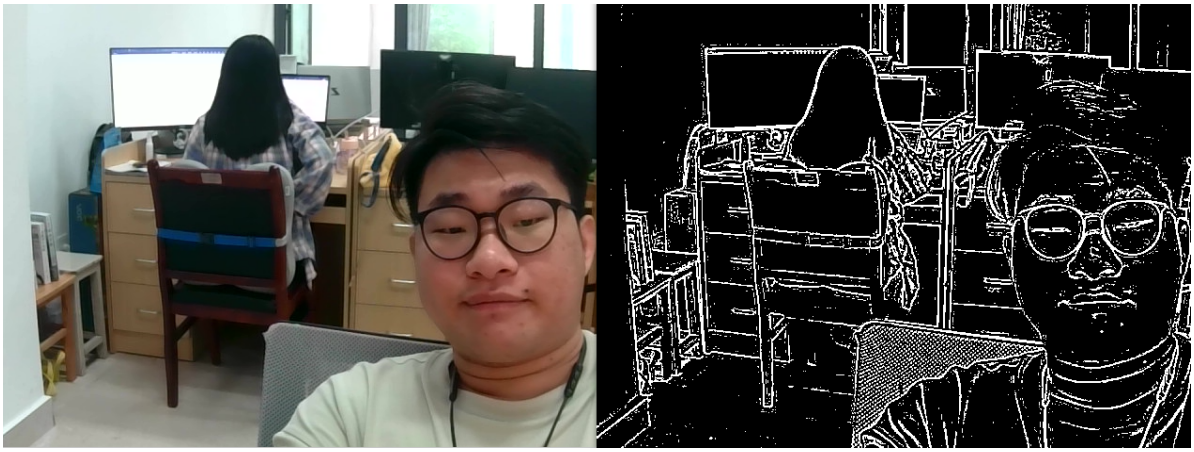
如果我们需要新增加一个粉笔画效果，只需要编写一个新的`Worker`子类，并插入流程图中合适的位置即可。

```
class ChalkEffects(Worker):
    '''
    粉笔画效果
    '''
    def __init__(self, name: str = 'chalk_effects'):
        super().__init__(name)

    def process(self, frame: Frame):
        # 将图片灰度化
        img = cv2.cvtColor(frame.data['img'], cv2.COLOR_BGR2GRAY)
        # 自适应二值化
        img = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
                                    cv2.THRESH_BINARY, 5, 3)
        # 对二值化图像取反最终得到粉笔画效果
        frame.data['img'] = cv2.bitwise_not(img)
        # 返回修改后的帧
        return frame
```

[代码2.6+] 实现粉笔画效果的Worker子类

```
if __name__ == '__main__':
    mirror = NodeSet([
        Node('node1', subsequents=['node2'], worker=ReadCamera()),
        Node('node2', subsequents=['node4'], worker=Flip()),# 修改node2的后继
        Node('node4', subsequents=['node3'], worker=ChalkEffects())
        Node('node3', worker=ShowImg())
    ])

    while mirror.switch:
        mirror.run(Frame(end='node1'))
```

[代码2.7] 在流程图中插入新的节点 node4

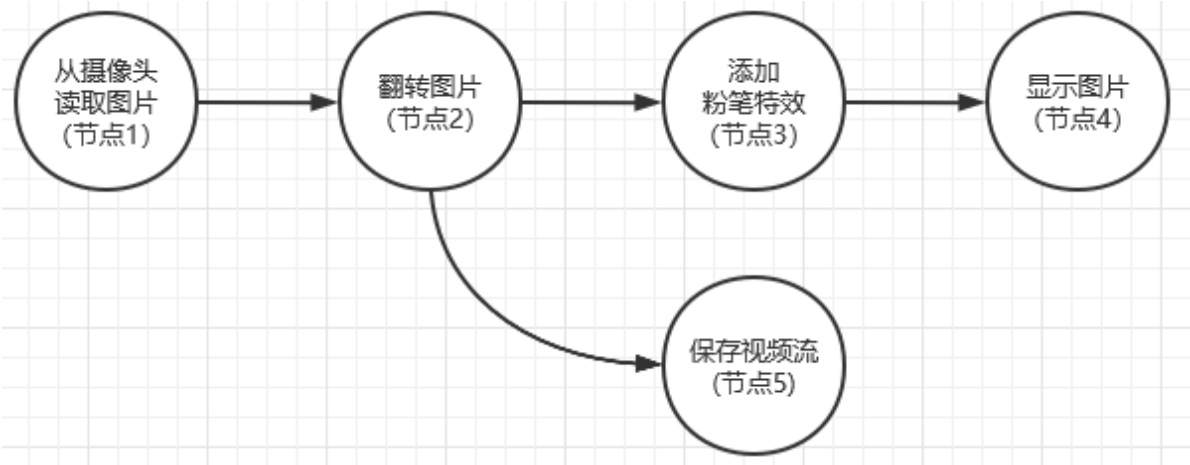


[图2.5] 添加node4前后效果对比

2.2 数据流的保存和重现

在开发一些对以硬件设备要求较高的项目时，常常存在各种各样的困扰，比如程序中有一个组件A需要用到高性能GPU，但高性能GPU被装载在办公室的台式机上，而项目中正在开发的组件B虽然不需要直接用到GPU，但仍然需要依赖来自GPU的数据。而您希望在家里用笔记本开发组件B，那么pipeline 中的**数据流保存和重现技术**或许能帮到您。

仍然以《镜子2.0》的代码为例，假设我们需要新开发一个油画组件：将图像变为油画风格，但开发油画组件风格的团队成员A电脑上没有摄像头，那么有摄像头的成员B就可以在他的电脑上得到一些数据供成员A使用，成员B的代码对应的流程图如下：



[图2.6]保存翻转后的数据流

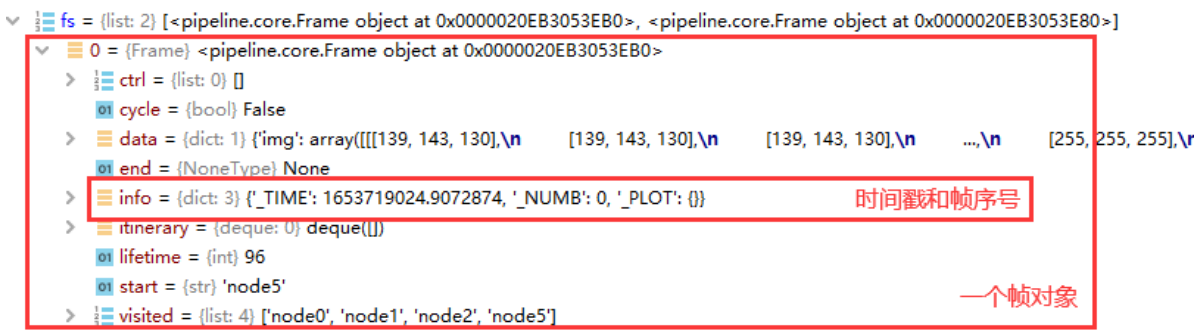
以上流程图会将节点2之后的所有数据帧保存到磁盘，对应代码如下：

```
if __name__ == '__main__':
    # pipeline.utils 是pipeline的官方工具包
    import pipeline.utils as utils
    mirror = NodeSet([
        # 使用官方工具包中的 Source 组件作为起始组件
        Node('node0', subsequents=['node1'], worker=utils.Source())
        Node('node1', subsequents=['node2'], worker=ReadCamera()),
        Node('node2', subsequents=['node3', 'node5'], worker=Flip()),
        Node('node3', subsequents=['node4'], worker=ChalkEffects()),
        Node('node4', worker=ShowImg()),
        # 使用官方工具包中的 Save 组件保存数据流
        Node('node5', worker=utils.Save(save_path='./output/')),
    ])

    while mirror.switch:
        fs = mirror.run(Frame(end='node0'))
        print(fs[0].info)
```

[代码2.8] 保存数据流

数据流的保存组件 `Save` 已经在官方工具包 `pipeline.utils` 中实现好了，其默认保存路径为 `./output/`，但可以通过参数 `save_path` 进行修改。

有必要提及的是，**在此必须添加一个 `Source` 组件作为源头组件**，也就是[代码2.8]中的 `node0` 节点对应的组件， `Source` 的功能是为 `Frame` 打上序号和时间戳，这样才方便后期将保存到磁盘的数据流读回程序。



[图2.7] Source 的作用是将时间戳(_TIME)和帧序号(_NUMB)写入帧的成员变量info中

运行[代码2.8]一段时间后，将会发现项目根目录下出现了一个`output`文件夹，成员B可以将该文件夹复制给成员A，A可以使用官方工具包 `pipeline.utils` 中的 `Load` 组件读取，代码如下：

```
if __name__ == '__main__':
    mirror = NodeSet([
        # 使用官方工具包中的 Load 组件读取'./output/'中的数据
        Node('node1', subsequents=['node2'],
             worker=utils.Load(load_path='./output/', reappear=True)),
        Node('node2', worker=ShowImg()),
    ])

    while mirror.switch:
        mirror.run(Frame(end='node1'))
```

[代码2.9] 从磁盘读取数据流

由于`./output/`保存的是读取摄像头图片并翻转之后的数据流，所以[代码2.9] 等价于流程图[图2.8]：

[图2.8] 与[代码2.9]等价的流程图

2.3 多进程启动器：MulIgnition

python中实现多进程并不是一件太方便的事情，但在pipeline中你可以非常轻松的做到这一点，我们只需要导入 `pipeline.mul` 包，调用里面的 `MulIgnition`，就可以让单进程直接变成多进程。

比如以[代码2.3]为例，只需要简单的修改，就能将单进程变为多进程：

```
import pipeline.mul as mul
import pipeline.core as core


# 请一定要添加 if __name__ == '__main__':
if __name__ == '__main__':
    chart1 = core.NodeSet([
        core.Node('head1', ['node1']),
        core.Node('node1', ['node2']),
        core.Node('node2', ['node3', 'head2/node5']),
        core.Node('node3'),
        core.Node('node4')
    ], source='head1')  # 指定数据入口为 'head1'

    chart2 = core.NodeSet([
        core.Node('head2', ['node1']),
        core.Node('node1', ['node2']),
        core.Node('node2', ['node3', 'node4']),
        core.Node('node3'),
        core.Node('node5')
```

```
    ])

    chart = mul.MulIgnition([chart1, chart2])
    chart.run()
```

[代码2.10] 将 [代码2.3]修改为多进程

其中`mul.MulIgnition`默认以一个列表为初始参数，列表中的元素均为`Node`对象，`mul.MulIgnition`会将每一个元素都实例化为一个进程，比如[代码2.10]的`chart1`和`chart2`分别被实例化为一个进程并行运算。

除此之外，每个进程都可以通过`source`参数指定一个入口节点，比如`chart1`就指定自己的数据入口为`head1`。而`chart2`没有指定自己的入口节点，那么多进程起始节点不会直接为其发送数据，其数据只能由其他进程指向它的节点发来（比如`chart1`的`node2`节点）。

为了更好的说明多进程的运行逻辑，[代码2.10] 的单进程版本等效于 [代码2.11]:

```
import pipeline.core as core


if __name__ == '__main__':
    # 在多进程中，程序会为流程图自动添加一个公共起点，但在单进程中需要手动编写
    start_node = core.Node('start_node', ['head1'])
    chart1 = core.NodeSet([
        core.Node('head1', ['node1']),
        core.Node('node1', ['node2']),
        core.Node('node2', ['node3', 'head2/node5']),
        core.Node('node3'),
        core.Node('node4')
    ])#单进程中不需要指定入口节点，因为多进程中的节点入口在单进程中变成了start_node的后继

    chart2 = core.NodeSet([
        core.Node('head2', ['node1']),
        core.Node('node1', ['node2']),
        core.Node('node2', ['node3', 'node4']),
        core.Node('node3'),
        core.Node('node5')
    ])

    chart = core.NodeSet([start_node, chart1, chart2])
    # 多进程中会自动实现类似于循环的结构，但单进程中需要手动编写
    while chart.switch:
        chart.run(core.Frame('start_node'))
```

[代码2.11] 与[代码2.10]运行逻辑完全等效的单进程版本

2.4 *WorkerSet：线性流程图下的简便写法

在[代码2.7]中，为了给程序添加`node4`节点，除了需要将`node4`节点添加进去之外，还需要修改`node2`的后继，但无论是 [代码2.6] 还是 [代码2.7]， 他们对应的流程图都是线性的，那么是否可以直接用组件在列表中的先后顺序表示组件的运行顺序呢?

秉承 **用简单的代码做简单的事** 的理念，pipeline 提供了一个针对线性流程图的简便实现方案：`WorkerSet`。

`WorkerSet` 是 `pipeline.core` 下的一个类，是 `Worker` 类的子类，也就是说，它也可以挂在 `Node` 对象上。它的初始化参数有两个：`name: 该WorkerSet的名字`，`workers: 一个Woker列表`。

使用WorkerSet后，[代码2.7]可以写成：

```
if __name__ == '__main__':
    mirror = WorkerSet('mirror',[
        ReadCamera(),
        Flip(),
        ChalkEffects(),
        ShowImg()
    ])

    while mirror.switch:
        # 默认从列表的第一个组件开始运行，不需要指定起始节点
        fs = mirror.run(Frame())
```

[代码2.12] [代码2.17]使用WorkerSet的编写方案

由于默认从 `workers` 的第一个组件开始所以不需要在 `Frame()` 中用 `end` 参数指定起始节点。其实也没法指定节点，因为在 `WorkerSet` 中根本没有 `Node` 或者 `NodeSet`。

使用 `WorkerSet` 的好处是方便，但坏处是降低了程序的可拓展性，因为它只能实现线性的流程图。

3. 说明文档（未完成）

3.1 框架逻辑

3.2 源码介绍

3.3 编程规范

4. 开发案例（未完成）

4.1 川剧变脸

4.2 训练faceNet