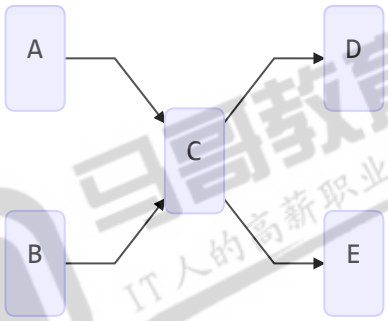
流程系统

目前已经完成了流程定义的实现，下面要使用它来实现任务流程的流转。

目前设计存在如下问题：

- 用户是否需要频繁查取正在运行任务状态？需要的，使用pipeline表以减少对track的查询，提高效率
- pipeline表不能很好的描述多节点，如果描述当前运行的流程有分支，且**正在执行**的超过一个顶点，就不好描述了，因为每一个顶点都有自己的状态，这张表就要描述一对多关系，目前不适合。
- 如果一个流程有多起点，pipeline如何描述？目前不适合
- 如果一个流程任务对应的顶点大于1
    - 如何在pipeline表中表示，目前不适合
    - 运行条件是什么？要求前面所有任务都必须是成功状态



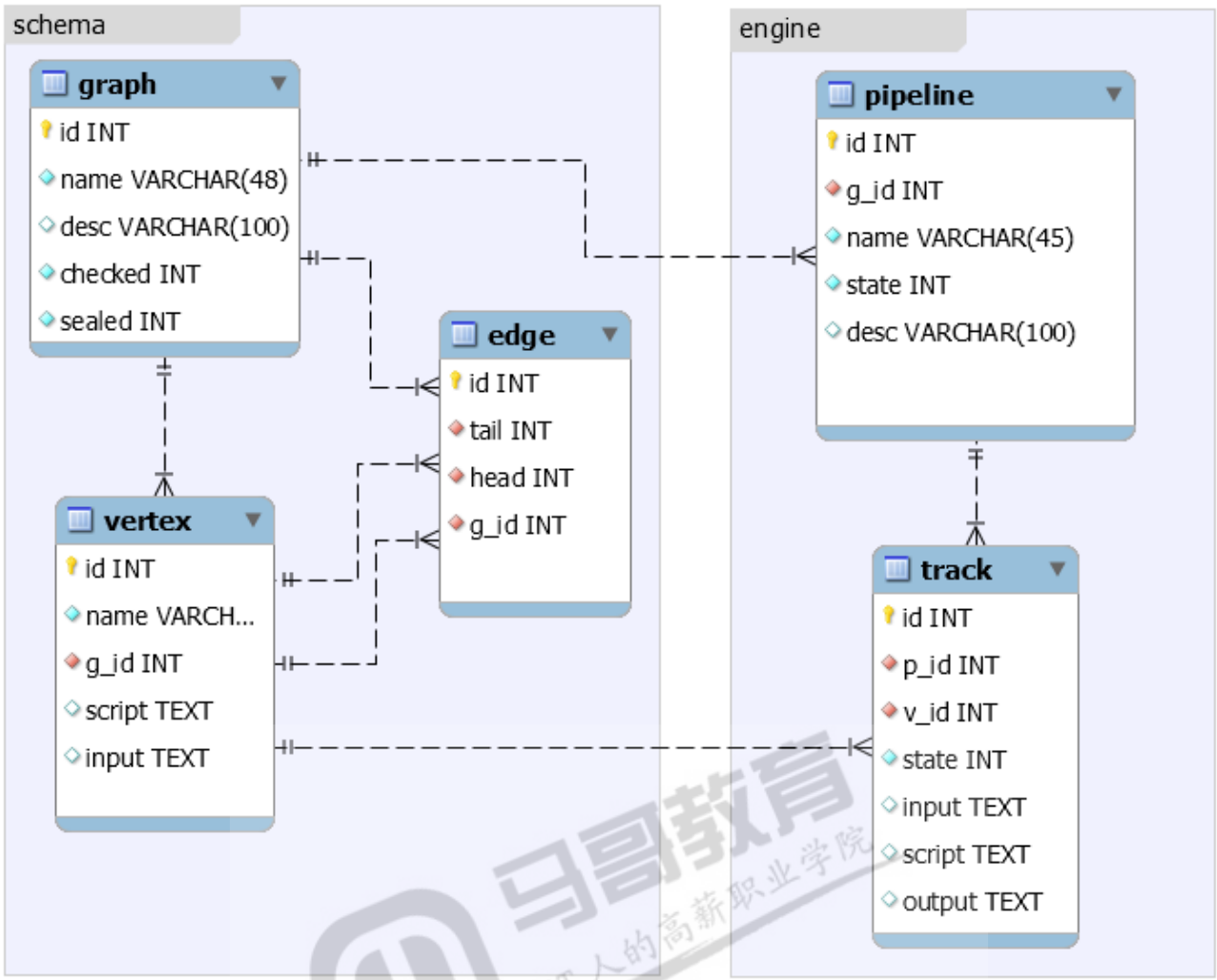
为了解决上面的问题

- 目前pipeline表的设计不能很好地满足业务需求，直接废弃原有设计。将pipeline表改成用来记录任务流信息的，字段有g_id、name、desc、state等。
    - g_id 表示使用哪一个DAG
    - name 任务流名称，例如 WEB服务器检查
    - desc 任务流详细描述
    - state 记录整个任务流的状态，有3种
        - 有节点运行就是STATE_RUNNING
        - 有节点失败就是STATE_FAILED
        - 全部节点都成功就是STATE_FINISH
- 用track表来记录流程信息，一个任务流产生，在它这里记录数据，使用pipeline任务流ID即p_id，并记录状态
- 任务节点执行状态，有5种。没有FINISH，它是描述整个任务流的
    - STATE_WAITING = 0
    - STATE_PENDING = 1
    - STATE_RUNNING = 2

    - STATE_SUCCEED = 3
    - STATE_FAILED = 4

- 如何解决频繁查询全部节点信息的状态？创建任务流时，**从vertex表中复制到track表**，所有顶点状态为STATE_WAITING。起点要被设置为STATE_PENDING
- 如何解决反复查询当前正在执行的任务？在state字段上建立**索引**，提高查询效率。
- 什么是正在执行的任务节点？STATE_WAITING 表示等待执行，STATE_PENDING 表示入调度器准备执行，STATE_RUNNING 表示该此节点正在执行。
- 如何描述一个任务流执行完成了？
    - 有一个任务顶点执行失败，则整个任务执行失败，所有节点不再继续执行。pipeline中表示为STATE_FAILED
    - 所有节点都成功STATE_SUCCEED ，则将pipeline中的状态置为STATE_FINISH
- 如果一个流程有多起点，在所有节点信息复制到track表中的时候就将这些节点的状态置为STATE_PENDING
- 如果一个流程节点的入度大于1，需要它的前驱节点都要是成功状态STATE_SUCCEED，它才能被置为STATE_PENDING
- 如果流程节点执行完成，就是成功、失败这两种状态之一，这些状态都要写入track表，如果track表中该p_id的所有节点都成功，pipeline中该任务状态STATE_FINISH
- 如果一个流程有多终点，同上，所有节点都必须成功， 否则就是失败
- 如果一个DAG定义中有孤立的顶点，如A -> B　C，C是一个孤立的顶点，如何解决？
    - 可以认为不合法，使用 `所有顶点集 - 所有边关联的顶点集 = 孤立顶点集`
    - 可以认为合法，就是一个入度为0的顶点，可以执行。可以认为是多起点，同时又是多终点
    - 本项目认为**合法**
- 节点流转，要求其所有前驱节点必须是成功状态STATE_SUCCEED
- track表增加记录用户操作的脚本script字段，减少使用input替换的时间

模型设计修改如下



model.py修改

```
class Pipeline(Base):
    __tablename__ = 'pipeline'

    id = Column(Integer, primary_key=True, autoincrement=True)
    g_id = Column(Integer, ForeignKey('graph.id'), nullable=False)
    # current = Column(Integer, ForeignKey('vertex.id'), nullable=False)
    name = Column(String(48), nullable=True) # +名称
    state = Column(Integer, nullable=False, default=STATE_WAITING)
    desc = Column(String(100))

    #vertex = relationship('Vertex')
    # 从pipeline去查所有节点信息
    tracks = relationship('Track', foreign_keys='Track.p_id')


class Track(Base):
    __tablename__ = 'track'

    id = Column(Integer, primary_key=True, autoincrement=True)
    p_id = Column(Integer, ForeignKey('pipeline.id'), nullable=False)
    v_id = Column(Integer, ForeignKey('vertex.id'), nullable=False)
```

```
    state = Column(Integer, index=True, nullable=False, default=STATE_WAITING) # +索引
    input = Column(Text, nullable=True)
    script = Column(Text, nullable=True)
    output = Column(Text, nullable=True)

    vertex = relationship('Vertex')
    pipeline = relationship('Pipeline')

    def __repr__(self):
        return "<{} {} {} {}>".format(self.__class__.__name__, self.id, self.p_id, self.v_id)

    __str__ = __repr__
```

代码实现

执行引擎

开启一个流程

开启一个流程的时候，需要在界面中选取一个checke为1的即验证过的DAG。为流程起名、填写描述，提交。
创建一个流程后，得到流程ID即p_id，将流程所有顶点加入到track表。
读取所有边，找出入度为0的顶点，这些顶点在track表中的状态置为RUNNING，其它非起点节点置为WAITING。

如何用SQL找到入度为0的顶点？子查询实现

```
SELECT id FROM vertex WHERE vertex.g_id = 1 AND id NOT IN (SELECT head FROM edge WHERE edge.g_id
= 1)
```

如何sqlalchemy实现这个子查询呢？

```
# 查询这个graph的所有顶点全部
vertexes = db.session.query(Vertex.id).filter(Vertex.g_id == graph.id)
if not vertexes:
    return

# 查出所有起点，入度为0，子查询实现
query = vertexes.filter(Vertex.id.notin_(db.session.query(Edge.head).filter(Edge.g_id ==
graph.id)))
zds = {x[0] for x in query} # query每一个元素是一个元组
print(zds, '~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
```

在pipeline包中新建一个executor.py

```
# executor.py
from .service import Graph, Vertex, Edge, transactional, db
from .service import Pipeline, Track, STATE_WAITING, STATE_SUCCEED, STATE_RUNNING

# 开启一个流程，用户指定一个名称、描述
```

```python
@transactional
def start(graph:Graph, name:str, desc=None):
    # 判断流程是否存在，且checked为1即检验过的
    g = db.session.query(Graph).filter(Graph.id == graph.id).filter(Graph.checked == 1).first()
    if not g:
        return

    # 写入pipeline表
    p = Pipeline()
    p.name = name
    p.desc = desc
    p.g_id = g.id
    p.state = STATE_RUNNING # 开启一个流程运行
    db.session.add(p)

    # 查询这个graph的所有顶点全部
    vertexes = db.session.query(Vertex.id).filter(Vertex.g_id == graph.id)
    if not vertexes:
        return

    # 查出所有起点，入度为0，子查询实现
    query = vertexes.filter(Vertex.id.notin_(db.session.query(Edge.head).filter(Edge.g_id ==
graph.id)))
    zds = {x[0] for x in query} # query每一个元素是一个元组
    print(zds, '~~~~~~~~~~~~~~~~~~~~~~~~~~')

    for v in vertexes:
        # 写入track表
        t = Track()
        t.pipeline = p
        t.v_id = v.id
        t.state = STATE_WAITING if v.id not in zds else STATE_PENDING
        db.session.add(t)
        print(v, '-------', t.state, v.id)

    # 标记有人使用过了，sealed封闭
    if g.sealed == 0:
        g.sealed = 1
        db.session.add(g)

    return p
```

```
# test.py中
from pipeline.service import Graph, Vertex, db
from pipeline.executor import start
import simplejson

# 测试start
def test_start():
    g = Graph()
    g.id = 1

    p = start(g, '流程1')

test_start()
```

input验证

开启一个流程后，顶点可能设置了input，这时候就要有一个界面，让用户填写参数。这是一个交互过程，也可以实现为自动填写参数。
获取参数后需要验证，验证失败要抛出异常，验证成功就用来替换执行脚本，生成可以运行的脚本。
然后将参数、脚本存入数据库的track表。
在track表添加script字段，存储执行的脚本

```
input = {
    "ip":{
        "type":"str",
        "required":True,
        "default":'127.0.0.1'
    }
}
required 是否必须，True则用户必须输入值，default缺失值忽略
default 缺省值，如果非必须值，用户没有填写了，使用缺省值

script = {
    'script':'echo "test1.A"\nping {ip}',
    'next': ['B']
}
{ip} 占位符，用户提供参数后，使用名称ip进行替换
```

特别注意，如果使用ping命令测试，windows默认ping4下，Linux下会一直ping下去，所以如果使用Linux测试项目脚本，一定要注意使用 `ping {ip} -c 4` ，此命令发送4个包就会停止命令

以下是Linux ping命令使用，注意命令的返回值。0为正确执行。-w 指定秒数必须完成ping命令，如果没有完成都算失败。

```
$ ping www.baidu.com -w 8 -c 2; echo $?
PING www.a.shifen.com (61.135.169.121) 56(84) bytes of data.
64 bytes from 61.135.169.121: icmp_seq=1 ttl=128 time=4.07 ms
64 bytes from 61.135.169.121: icmp_seq=2 ttl=128 time=5.97 ms

--- www.a.shifen.com ping statistics ---
```

```
2 packets transmitted, 2 received, 0% packet loss, time 1009ms
rtt min/avg/max/mdev = 4.078/5.028/5.978/0.950 ms
0

$ ping 172.16.100.100 -w 8 -c 2; echo $?
PING 172.16.100.100 (172.16.100.100) 56(84) bytes of data.

--- 172.16.100.100 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 8001ms

1

$ ping www.baidu.com -w 2 -c 8; echo $?
PING www.a.shifen.com (61.135.169.121) 56(84) bytes of data.
64 bytes from 61.135.169.121: icmp_seq=1 ttl=128 time=6.78 ms
64 bytes from 61.135.169.121: icmp_seq=2 ttl=128 time=5.22 ms

--- www.a.shifen.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 5.228/6.004/6.780/0.776 ms
1
```

返回指定流程的信息
比如说在浏览器中，获取当前流程的信息

```
# executor.py中
# 查询流程的某种状态节点
@transactional
def show_pipeline(id, state=STATE_PENDING):
    """显示指定的流程的信息"""
    p = db.session.query(Pipeline.id, Pipeline.name, Pipeline.state,
        Track.id, Track.v_id, Track.state, Vertex.input, Vertex.script).\
        join(Track, (Track.p_id == id) & (Pipeline.id == Track.p_id)).\
        join(Vertex, Track.v_id == Vertex.id)\
        .filter(Pipeline.state != STATE_FAILED)\
        .filter(Track.state == state)
        # Pipeline.state != STATE_FAILED 必须是没有失败的
    return p.all()
```

下面要模拟在浏览器中，看到了显示当前浏览器的信息，如果需要提供参数，就显示交互界面，让用户输入。
然后，数据提交，验证后，替换script脚本中的占位符，生成可以执行的脚本

```
# test.py中
# 这部分代码模拟用户提供参数，形成一个字典
from pipeline.executor import show_pipeline
import simplejson

ps = show_pipeline(1)  # 返回运行节点列表
print('-' * 30)
```

```
print(ps)
print('-' * 30)

for p_id, p_name, p_state, t_id, v_id, t_state, inp, script in ps:
    print(p_id, p_name, p_state, t_id, v_id, t_state, inp, script)

    d = {} # 如果参数是必须，则交互，让用户提交
    if inp:
        inp = simplejson.loads(inp)
        for k in inp.keys():
            if inp[k].get('required1', False):
                d[k] = input('{}= '.format(k))
        print(d)
```

然后填充脚本script

```
# service.py中
import simplejson

# 类型转换用
TYPES = {
    'str': str,
    'string': str,
    'int': int,
    'integer': int
}

@transactional
def finish_params(t_id, d:dict, inp):
    """完成所有参数值"""
    params = {}  # 最终的参数
    if inp:
        print(inp)
        print(d)
        for k,v in inp.items():
            print(k,v)
            val = d.get(k)
            if isinstance(val, TYPES.get(v['type'], str)):
                params[k] = val
            elif v.get('default'): # 类型不对，但是有缺省值
                params[k] = v.get('default')
            else:
                raise TypeError('参数类型错误')

        # 将input存入数据库
        track = db.session.query(Track).filter(Track.id == t_id).first()
        if track:
            track.input = simplejson.dumps(params)  # 转成字符串
            db.session.add(track)

    return params
```

```
@transactional
def finish_script(t_id, script:str, params:dict):
    '''使用参数替换脚本'''
    newline = ''
    if script:
        if isinstance(script, str):
            script = simplejson.loads(script).get('script')

        import re
        regex = re.compile(r'{([^{}]+)}')

        start = 0

        for matcher in regex.finditer(script):
            newline += script[start:matcher.start()]
            print(matcher, matcher.group(1))
            key = matcher.group(1)
            tmp = params.get(key, '')
            newline += str(tmp)
            start = matcher.end()
        else:
            newline += script[start:]

        # 把生成的script存入库
        track = db.session.query(Track).filter(Track.id == t_id).first()
        if track:
            track.script = newline  # 转成字符串
            db.session.add(track)

    return newline
```

测试代码

```
# test.py中
from pipeline.executor import show_pipeline
from pipeline.executor import finish_params, finish_script
import simplejson

ps = show_pipeline(1)  # 返回运行节点列表
print('-' * 30)
print(ps)
print('-' * 30)

for p_id, p_name, p_state, t_id, v_id, t_state, inp, script in ps:
    print(p_id, p_name, p_state, t_id, v_id, t_state, inp, script)

    d = {} # 如果参数是必须，则交互，让用户提交
    if inp:
        inp = simplejson.loads(inp)
        for k in inp.keys():
            if inp[k].get('required', False):
```

```
                d[k] = input('{}= '.format(k))
        print(d)

    params = finish_params(t_id, d, inp)
    print(params)  # 准备好参数
    print(script, '+++++++++')
    script = finish_script(t_id, script, params)
    print(script)  # 拿到替换好的脚本，准备执行

# 数据库Track表如下
id  p_id  v_id  state   input                 script                          output
1    1     1      1     {"ip": "127.0.0.1"}   echo "test1.A" ping 127.0.0.1
```

执行

执行脚本，而脚本执行的是命令，而命令就是写好的程序，这些程序执行就是一个个进程。

python有很多运行进程的方式，不过都过时了。
建议使用标准库subprocess模块，启动一个子进程。

> class subprocess.Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=())

shell为True，则使用shell来执行args，建议args是一个字符串。

```
from subprocess import Popen, PIPE

p = Popen('echo hello', shell=True, stdout=PIPE)
code = p.wait() # 阻塞等，code为0是正确执行
text = p.stdout.read() # bytes
print(code, text)
```

脚本执行的输出可能非常大，使用PIPE不太合适，使用临时文件模块

```
from subprocess import Popen, PIPE
from tempfile import TemporaryFile

with TemporaryFile('w+') as f:
    p = Popen('echo magedu', shell=True, stdout=f)
    code = p.wait()  # 阻塞等，code为0是正确执行
    f.seek(0)  # 回到开头
    text = f.read()
    print(code, text)
```

由于wait会阻塞，所以使用多线程，使用subprocess的Popen开启子进程执行。但是开启线程后返回的结果就不能直接拿到了。使用concurrent.futures来异步并发执行，并获取返回的结果。

先学习一个例子

```
from concurrent.futures import ThreadPoolExecutor, as_completed
```

```python
import random
import threading
import time


def test_func(s, key):
    print('enter~~{} {}s key={}'.format(threading.current_thread(), s, key))
    threading.Event().wait(s)
    if key == 3:
        raise Exception("{} failed~~~".format(key))
    return 'ok {}'.format(threading.current_thread())

futures = {}

def run(fs):
    print('~~~~~~~~~~~~~~~')
    while True:
        time.sleep(1)
        print('-'*30)
        print(fs)
        # 只要有一个任务没有完成就阻塞，完成一个，执行一次
        # 如果内部有异常result()会将这个异常抛出
        # 有异常也算执行完了complete
        # fs为空也不阻塞
        for future in as_completed(fs):
            id = fs[future]
            try:
                print(id, future.result())
            except Exception as e:
                print(e)
                print(id, 'failed')

threading.Thread(target=run, args=(futures,)).start()

#executor = ThreadPoolExecutor(max_workers=3)
with  ThreadPoolExecutor(max_workers=3) as executor:
    for i in range(7):
        futures[executor.submit(test_func, random.randint(1,8), i)] = i # 生成任务

# 运行结果
~~~~~~~~~~~~~~~
enter~~<Thread(Thread-2, started daemon 6408)> 7s key=0
enter~~<Thread(Thread-3, started daemon 5640)> 3s key=1
enter~~<Thread(Thread-4, started daemon 8352)> 2s key=2
------------------------------
{<Future at 0x2c1a320 state=pending>: 6, <Future at 0x2c0bb38 state=running>: 0, <Future at
0x2c1a160 state=pending>: 3, <Future at 0x2c1a278 state=pending>: 5, <Future at 0x1108a90
state=running>: 1, <Future at 0x2c1a1d0 state=pending>: 4, <Future at 0x2c0bef0 state=running>:
2}
enter~~<Thread(Thread-4, started daemon 8352)> 5s key=3
2 ok <Thread(Thread-4, started daemon 8352)>
enter~~<Thread(Thread-3, started daemon 5640)> 1s key=4
1 ok <Thread(Thread-3, started daemon 5640)>
```

```
enter~~<Thread(Thread-3, started daemon 5640)> 7s key=5
4 ok <Thread(Thread-3, started daemon 5640)>
enter~~<Thread(Thread-2, started daemon 6408)> 4s key=6
0 ok <Thread(Thread-2, started daemon 6408)>
3 failed~~~
3 failed
6 ok <Thread(Thread-2, started daemon 6408)>
5 ok <Thread(Thread-3, started daemon 5640)>
------------------------------
{<Future at 0x2c1a320 state=finished returned str>: 6, <Future at 0x2c0bb38 state=finished
returned str>: 0, <Future at 0x2c1a160 state=finished raised Exception>: 3, <Future at 0x2c1a278
state=finished returned str>: 5, <Future at 0x1108a90 state=finished returned str>: 1, <Future
at 0x2c1a1d0 state=finished returned str>: 4, <Future at 0x2c0bef0 state=finished returned str>:
2}
5 ok <Thread(Thread-3, started daemon 5640)>
1 ok <Thread(Thread-3, started daemon 5640)>
2 ok <Thread(Thread-4, started daemon 8352)>
6 ok <Thread(Thread-2, started daemon 6408)>
0 ok <Thread(Thread-2, started daemon 6408)>
3 failed~~~
3 failed
4 ok <Thread(Thread-3, started daemon 5640)>
```

可以看出当所有任务完成后，as_completed就不会等待了。

执行器类实现

调用执行器的执行execute方法，此方法会自动将任务提交，并异步执行

```python
# executor.py中
from subprocess import Popen, PIPE
from tempfile import TemporaryFile
from concurrent.futures import ThreadPoolExecutor, as_completed
import threading
import uuid
from queue import Queue

class Executor:
    def __init__(self, workers=5):
        self.__pool = ThreadPoolExecutor(max_workers=workers)
        self.__event = threading.Event()
        self.__tasks = {}
        self.__queue = Queue()
        threading.Thread(target=self._run).start()
        threading.Thread(target=self._save_track).start()

    def _execute(self, script:str):
        with TemporaryFile('w+') as f:
            output = []
            code = 0
            for line in script.splitlines():
                p = Popen(line, shell=True, stdout=f)
                code = p.wait()  # 阻塞等，code为0是正确执行
```

```
                f.seek(0)  # 回到开头
                text = f.read()
                output.append(text)
                code += code
            return code, '\n'.join(output)

    def execute(self, p_id, t_id, script:str):
        """异步执行方法，提交数据就行了，运行后，会提供运行结果，或返回失败"""
        key = uuid.uuid4().hex # uuid没有用上，只是说以后不重复key或id可以用uuid
        try:
            self.__tasks[self.__pool.submit(self._execute, script)] = (key, p_id, t_id) # futere
对象
            # 修改状态为准备执行RUNNING
            track = db.session.query(Track).filter(Track.id == t_id).one()
            track.state = STATE_RUNNING
            db.session.add(track)
            db.session.commit()
        except Exception as e:
            db.session.rollback()
            print(e)

    def _run(self): # 线程等待任务
        while not self.__event.wait(1):
            for future in as_completed(self.__tasks):
                key, p_id, t_id = self.__tasks[future]
                try:
                    code, text = future.result()
                    del self.__tasks[future]
                    self.__queue.put((p_id, t_id, code, text))
                except Exception as e:
                    print(key, e)
                    del self.__tasks[future] # 失败任务以后处理 TODO

    def _save_track(self):
        while not self.__event.is_set():
            p_id, t_id, code, text = self.__queue.get() # 阻塞取

            track = db.session.query(Track).filter(Track.v_id == t_id).first()
            track.state = STATE_SUCCEED if code==0 else STATE_FAILED# 修改状态
            track.output = text

            if code != 0: # 失败，必须立即将任务流状态设置为失败
                track.pipeline.state = STATE_FAILED
            db.session.add(track)

            try:
                db.session.commit()
            except Exception as e:
                print(e)
                db.session.rollback()


EXECUTOR = Executor() # 全局任务执行器对象
```

注意，有可能出现下面的错误

`'latin-1' codec can't encode characters in position 55-56: ordinal not in range(256)`

运行的没有问题，字符串也没有错误，甚至数据库客户端执行都没有问题，但是自己写的程序显示latin-1长度超了。原因在于数据库连接没有设定字符集。

```
# config.py中对数据库连接指定字符集
DATABASE_DEBUG = True
USERNAME = 'wayne'
PASSWD = 'wayne'
DBIP = '192.168.142.140'
DBPORT = 3306
DBNAME = 'pipeline'
PARAMS = "charset=utf8mb4"

URL = 'mysql+pymysql://{}:{}@{}:{}/{}?{}'.format(USERNAME, PASSWD, DBIP, DBPORT, DBNAME, PARAMS)
```

测试代码

```
from pipeline.executor import show_pipeline
from pipeline.executor import finish_params, finish_script
import simplejson
from pipeline.executor import EXECUTOR

ps = show_pipeline(1)  # 返回运行节点列表
print('-' * 30)
print(ps)
print('-' * 30)

for p_id, p_name, p_state, t_id, v_id, t_state, inp, script in ps:
    print(p_id, p_name, p_state, t_id, v_id, t_state, inp, script)

    d = {} # 如果参数是必须，则交互，让用户提交
    if inp:
        inp = simplejson.loads(inp)
        for k in inp.keys():
            if inp[k].get('required1', False):
                d[k] = input('{}= '.format(k))
        print(d)

    params = finish_params(t_id, d, inp)
    print(params)  # 准备好参数
    print(script, '+++++++++')
    script = finish_script(t_id, script, params)
    print(script)  # 拿到替换好的脚本，准备执行

    EXECUTOR.execute(p_id, t_id, script)  # 异步执行
```

测试通过