流程系统

代码实现

流转

手动流转以后实现。实现自动流转。

如何知道该某个节点该轮到它执行了？
间隔着反复到track表查询什么节点可以执行了吗？
为了减少对数据库的查询，最好的方式应该是由前一个节点成功完成后触发一个查询。
查询完成的节点的下一个节点是否存在、是否具备执行的条件等

- 首先，需要在pipeline中查看当前任务状态是否已经失败，如果失败，则不再继续找下一个节点。否则成功执行，继续下面操作
- 本节点成功执行置为成功，在track表查询一下本任务流除自己之外还有没有其它节点在运行中，遍历所有其它节点
  - 首先判断如果有一个失败，就立即置pipeline的state为STATE_FAILED
  - 如果其它节点都是成功，则置pipeline的state为STATE_FINISH
  - 如果碰到一个STATE_WAITING、STATE_RUNNING，则就搜索下一级节点
- 下一个节点
  - 没有下一级节点，说明该节点是终点。是终点，不代表没有其它终点。本节点没有下一级它就不用管其它节点了，只需要把自己的状态置为成功就行了
  - 如果节点没有执行失败，一定会成功执行，其它节点继续执行，如果最后一个终点执行完，会发现其他节点全是成功状态，所以它将pipeline的state置为STATE_FINISH就可以了

测试数据准备

```
# 由于将script格式更改了，所以重新提供该函数
# 测试数据
def test_create_dag():
    try:
        # 创建DAG
        g = create_graph('test1') # 成功则返回一个Graph对象
        # 增加顶点
        input = {
            "ip":{
                "type":"str",
                "required":True,
                "default":'127.0.0.1'
            }
        }

        script = {
            'script':'echo "test1.A"\nping {ip}',
            'next':'B'
        }
```

```
        # 这里为了让用户方便，next可以接收2种类型，数字表示顶点的id，字符串表示同一个DAG中该名称的节点，不能重复
        a = add_vertex(g, 'A', json.dumps(input), json.dumps(script)) # next顶点验证可以在定义时，也可以在使用时
        b = add_vertex(g, 'B', None, '{"script":"echo B"}')
        c = add_vertex(g, 'C', None, '{"script":"echo C"}')
        d = add_vertex(g, 'D', None, '{"script":"echo D"}')
        # 增加边
        ab = add_edge(g, a, b)
        ac = add_edge(g, a, c)
        cb = add_edge(g, c, b)
        bd = add_edge(g, b, d)

        # 创建环路
        g = create_graph('test2') # 环路
        # 增加顶点
        a = add_vertex(g, 'A', None, '{"script":"echo A"}')
        b = add_vertex(g, 'B', None, '{"script":"echo B"}')
        c = add_vertex(g, 'C', None, '{"script":"echo C"}')
        d = add_vertex(g, 'D', None, '{"script":"echo D"}')
        # 增加边，abc之间的环
        ba = add_edge(g, b, a)
        ac = add_edge(g, a, c)
        cb = add_edge(g, c, b)
        bd = add_edge(g, b, d)

        # 创建DAG
        g = create_graph('test3') # 多个终点
        # 增加顶点
        a = add_vertex(g, 'A', None, '{"script":"echo A"}')
        b = add_vertex(g, 'B', None, '{"script":"echo B"}')
        c = add_vertex(g, 'C', None, '{"script":"echo C"}')
        d = add_vertex(g, 'D', None, '{"script":"echo D"}')
        # 增加边
        ba = add_edge(g, b, a)
        ac = add_edge(g, a, c)
        bc = add_edge(g, b, c)
        bd = add_edge(g, b, d)

        # 创建DAG
        g = create_graph('test4') # 多入口
        # 增加顶点
        a = add_vertex(g, 'A', None, '{"script":"echo A"}')
        b = add_vertex(g, 'B', None, '{"script":"echo B"}')
        c = add_vertex(g, 'C', None, '{"script":"echo C"}')
        d = add_vertex(g, 'D', None, '{"script":"echo D"}')
        # 增加边
        ab = add_edge(g, a, b)
        ac = add_edge(g, a, c)
        cb = add_edge(g, c, b)
        db = add_edge(g, d, b)
    except Exception as e:
        print(e)
```

流转代码实现

```
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.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
            else:
                # ++++++++++ 流转代码~~~~~~~~~~~~
                # 所有其他节点
                others = db.session.query(Track).filter((Track.p_id == p_id) & (Track.v_id !=
t_id)).all()
                # 等待，待运行， 运行，成功，失败
                states = {STATE_WAITING:0, STATE_PENDING:0,STATE_RUNNING:0, STATE_SUCCEED:0,
STATE_FAILED:0}
                for other in others:
                    states[other.state] += 1

                print('+' * 30)
                print(states, len(others))
                print('+' * 30)
                if states[STATE_FAILED] > 0:
                    track.pipeline.state = STATE_FAILED
                elif states[STATE_SUCCEED] == len(others): # 除了它之外全是成功说明全部成功
                    track.pipeline.state = STATE_FINISH
                else: # 说明还有没有运行完的，开始找下一级节点们
                    nexts = db.session.query(Edge).filter(Edge.tail == track.v_id).all()
                    if nexts: # 有下一级，将这些节点的state改为STATE_PENDING
                        for n in nexts:
                            print(n.head)
                            t = db.session.query(Track).filter(Track.v_id == n.head).one()
                            t.state = STATE_PENDING
                            db.session.add(t)
                    else: # 没有下一级，是终点
                        # 如果自己是多终点的最后的一个终点，那么其他节点都是成功的
                        # 在上面的判断states[STATE_SUCCEED] == len(others)就成立了
                        pass
                # ++++++++++ 流转代码结束~~~~~~~~~~~~

            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
```

测试代码如下

```
# 测试代码和之前一样，因为流转是内部实现的
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
```

循环测试代码

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

while True:
    ps = show_pipeline(1)  # 返回运行节点列表
    print('-' * 30)
    print(ps)
    print('-' * 30)
    time.sleep(1)
```

```
    print('~~~~~~~~~~~~~~ sleeping ~~~~~~~~~~~~~')
    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
```

至此主流程已经完成，还需扩展功能详细设计和bug调试

完整代码

项目目录结构

```
.
├── app.py
└── pipeline
    ├── __init__.py
    ├── config.py
    ├── model.py
    ├── service.py
    └── executor.py
```

1 config.py

```
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)

DATABASE_DEBUG = True
```

2 model.py

```
from sqlalchemy import Column, Integer, String, Text, ForeignKey, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, relationship
from . import config

STATE_WAITING = 0
STATE_PENDING = 1
STATE_RUNNING = 2
STATE_SUCCEED = 3
STATE_FAILED = 4
STATE_FINISH = 5


Base = declarative_base()

class Graph(Base):
    __tablename__ = 'graph'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(48), nullable=False)
    desc = Column(String(200))
    checked = Column(Integer, nullable=False, default=0)
    sealed = Column(Integer, nullable=False, default=0)

    vertexes = relationship('Vertex')
    edges = relationship('Edge')

    def __repr__(self):
        return "<Graph {} {}>".format(self.id, self.name)

    __str__ = __repr__

class Vertex(Base):
    __tablename__ = 'vertex'

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(48), nullable=False)
    input = Column(Text, nullable=True)
    script = Column(Text, nullable=True)
    g_id = Column(Integer, ForeignKey('graph.id'), nullable=False)

    graph = relationship('Graph')
    # 从1查多
    tails = relationship('Edge', foreign_keys='Edge.tail')
    heads = relationship('Edge', foreign_keys='[Edge.head]')


class Edge(Base):
    __tablename__ = 'edge'
```

```
    id = Column(Integer, primary_key=True, autoincrement=True)
    tail = Column(Integer, ForeignKey('vertex.id'), nullable=False)
    head = Column(Integer, ForeignKey('vertex.id'), nullable=False)
    g_id = Column(Integer, ForeignKey('graph.id'), nullable=False)


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')

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

    __str__ = __repr__


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__

# 单例模式
import functools

def singleton(cls):
    instance = None

    @functools.wraps(cls)
    def getinstance(*args, **kwargs):
```

```
        nonlocal instance
        if not instance:
            instance = cls(*args, **kwargs)
        return instance

    return getinstance

@singleton
class Database:
    def __init__(self, url, **kwargs):
        self._engine = create_engine(url, **kwargs)
        self._session = sessionmaker(bind=self._engine)()

    @property
    def session(self):
        return self._session

    @property
    def engine(self):
        return self._engine

    def create_all(self):
        Base.metadata.create_all(self.engine)

    def drop_all(self):
        Base.metadata.drop_all(self.engine)

db = Database(config.URL, echo=config.DATABASE_DEBUG)
```

3 service.py

```
from .model import db
from .model import Graph, Vertex, Edge
from .model import Pipeline, Track
from .model import STATE_WAITING, STATE_RUNNING, STATE_FAILED, STATE_FINISH, STATE_SUCCEED
from functools import wraps

def transactional(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        try:
            ret = fn(*args, **kwargs)
            db.session.commit()
            return ret
        except Exception as e:
            db.session.rollback()
            raise
    return wrapper

# 创建DAG
@transactional
def create_graph(name, desc=None):
```

```
    g = Graph()
    g.name = name
    g.desc = desc
    db.session.add(g)
    return g

# 增加顶点
@transactional
def add_vertex(graph:Graph, name:str, input=None, script=None):
    v = Vertex()
    v.name = name
    v.g_id = graph.id
    v.input = input
    v.script = script
    db.session.add(v)
    return v

# 增加边
@transactional
def add_edge(graph:Graph, tail:Vertex, head:Vertex):
    e = Edge()
    e.g_id = graph.id
    e.tail = tail.id
    e.head = head.id
    db.session.add(e)
    return e


# 删除顶点
@transactional
def del_vertex(id):
    query = db.session.query(Vertex).filter(Vertex.id == id)
    v = query.first()
    if v:
        db.session.query(Edge).filter((Edge.tail == v.id) | (Edge.head == v.id)).delete()
        query.delete()
    return v



from collections import defaultdict

def check_graph(graph:Graph) -> bool:
    query = db.session.query(Vertex).filter(Vertex.g_id == graph.id)
    vertexes = {vertex.id for vertex in query}

    query = db.session.query(Edge).filter(Edge.g_id == graph.id)
    edges = defaultdict(list)
    ids = set() # 有入度的顶点
    for edge in query:
        # defaultdict(<class 'list'>, {1: [(1, 2), (1, 3)], 2: [(2, 4)], 3: [(3, 2)]})
        edges[edge.tail].append((edge.tail, edge.head))
        ids.add(edge.head)
```

```
    print('-='*30)
    print(vertexes, edges)

    # ===============测试数据===============
    # {1, 2, 3, 4} defaultdict(<class 'list'>, {1: [(1, 2), (1, 3)], 2: [(2, 4)], 3: [(3, 2)]})
    # vertexes = {1, 2, 3, 4}
    # edges = {1: [(1, 2), (1, 3)], 2: [(2, 4)], 3: [(3, 2)]}
    # ids = set() # 有入度的顶点
    # ====================================

    if len(edges) == 0:
        return False # 一条边都没有，这样的DAG业务上不用
    # 如果edges不为空，一定有ids，也就是有入度的顶点一定会有
    zds = vertexes - ids # zds入度为0的顶点
    # zds为0说明没有找到入度为0的顶点，算法终止
    if len(zds):
        for zd in zds:
            if zd in edges:
                del edges[zd]

        while edges:
            # 将顶点集改为当前入度顶点集ids
            # 能到这一步说明出度为0的已经清除了
            vertexes = ids
            ids = set() # 重新寻找有入度的顶点

            for lst in edges.values():
                for edge in lst:
                    ids.add(edge[1])
            zds = vertexes - ids
            print(vertexes, ids, zds)
            if len(zds) == 0:
                break
            for zd in zds:
                if zd in edges: # 有可能顶点没有出度
                    del edges[zd]
            print(edges)

    # 边集为空，剩下所有顶点都是入度为0的，都可以多次迭代删除掉
    if len(edges) == 0:
        # 检验通过，修改checked字段为1
        try:
            graph = db.session.query(Graph).filter(Graph.id == graph.id).first()
            if graph:
                graph.checked = 1
            db.session.add(graph)
            db.session.commit()
            return True
        except Exception as e:
            db.session.rollback()
            raise e
    return False
```

4 executor.py

```
from .service import Graph, Vertex, Edge, transactional, db
from .service import Pipeline, Track
from .model import STATE_WAITING, STATE_SUCCEED, STATE_RUNNING, STATE_FAILED, STATE_FINISH,
STATE_PENDING

# 开启一个流程，用户指定一个名称、描述
@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
    if g.sealed == 0:
        g.sealed = 1
        db.session.add(g)

    return p

# 查询流程的某种状态节点
@transactional
```

```python
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()

import simplejson

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

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.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
            else:
                # ++++++++++ 流转代码~~~~~~~~~~~~
                # 所有其他节点
                others = db.session.query(Track).filter((Track.p_id == p_id) & (Track.v_id !=
t_id)).all()
                # 等待，待运行， 运行，成功，失败
                states = {STATE_WAITING:0, STATE_PENDING:0,STATE_RUNNING:0, STATE_SUCCEED:0,
STATE_FAILED:0}
                for other in others:
                    states[other.state] += 1

                print('+' * 30)
                print(states, len(others))
                print('+' * 30)
                if states[STATE_FAILED] > 0:
```

```
                    track.pipeline.state = STATE_FAILED
                elif states[STATE_SUCCEED] == len(others): # 除了它之外全是成功说明全部成功
                    track.pipeline.state = STATE_FINISH
                else: # 说明还有没有运行完的，开始找下一级节点们
                    nexts = db.session.query(Edge).filter(Edge.tail == track.v_id).all()
                    if nexts: # 有下一级，将这些节点的state改为STATE_PENDING
                        for n in nexts:
                            print(n.head)
                            t = db.session.query(Track).filter(Track.v_id == n.head).one()
                            t.state = STATE_PENDING
                            db.session.add(t)
                    else: # 没有下一级，是终点
                        # 如果自己是多终点的最后的一个终点，那么其他节点都是成功的
                        # 在上面的判断states[STATE_SUCCEED] == len(others)就成立了
                        pass
                # ++++++++++ 流转代码结束~~~~~~~~~~~~

            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
```

5 app.py

```
# 目前放着测试代码和运行代码
import json
from pipeline.service import Graph, Vertex, db
from pipeline.service import create_graph, add_vertex, add_edge


# 测试数据
def test_create_dag():
    try:
        # 创建DAG
        g = create_graph('test1') # 成功则返回一个Graph对象
        # 增加顶点
        input = {
            "ip":{
                "type":"str",
                "required":True,
                "default":'127.0.0.1'
            }
        }

        script = {
            'script':'echo "test1.A"\nping {ip}',
            'next':'B'
```

```
        }
        # 这里为了让用户方便，next可以接收2种类型，数字表示顶点的id，字符串表示同一个DAG中该名称的节点，不能重复
        a = add_vertex(g, 'A', json.dumps(input), json.dumps(script)) # next顶点验证可以在定义时，也可以在使用时
        b = add_vertex(g, 'B', None, '{"script":"echo B"}')
        c = add_vertex(g, 'C', None, '{"script":"echo C"}')
        d = add_vertex(g, 'D', None, '{"script":"echo D"}')
        # 增加边
        ab = add_edge(g, a, b)
        ac = add_edge(g, a, c)
        cb = add_edge(g, c, b)
        bd = add_edge(g, b, d)

        # 创建环路
        g = create_graph('test2') # 环路
        # 增加顶点
        a = add_vertex(g, 'A', None, '{"script":"echo A"}')
        b = add_vertex(g, 'B', None, '{"script":"echo B"}')
        c = add_vertex(g, 'C', None, '{"script":"echo C"}')
        d = add_vertex(g, 'D', None, '{"script":"echo D"}')
        # 增加边，abc之间的环
        ba = add_edge(g, b, a)
        ac = add_edge(g, a, c)
        cb = add_edge(g, c, b)
        bd = add_edge(g, b, d)

        # 创建DAG
        g = create_graph('test3') # 多个终点
        # 增加顶点
        a = add_vertex(g, 'A', None, '{"script":"echo A"}')
        b = add_vertex(g, 'B', None, '{"script":"echo B"}')
        c = add_vertex(g, 'C', None, '{"script":"echo C"}')
        d = add_vertex(g, 'D', None, '{"script":"echo D"}')
        # 增加边
        ba = add_edge(g, b, a)
        ac = add_edge(g, a, c)
        bc = add_edge(g, b, c)
        bd = add_edge(g, b, d)

        # 创建DAG
        g = create_graph('test4') # 多入口
        # 增加顶点
        a = add_vertex(g, 'A', None, '{"script":"echo A"}')
        b = add_vertex(g, 'B', None, '{"script":"echo B"}')
        c = add_vertex(g, 'C', None, '{"script":"echo C"}')
        d = add_vertex(g, 'D', None, '{"script":"echo D"}')
        # 增加边
        ab = add_edge(g, a, b)
        ac = add_edge(g, a, c)
        cb = add_edge(g, c, b)
        db = add_edge(g, d, b)
    except Exception as e:
```

```
        print(e)


from pipeline.service import Graph, db
from pipeline.service import check_graph

def test_check_all_graph():
    query = db.session.query(Graph).filter(Graph.checked == 0).all()
    for g in query:
        if check_graph(g):
            g.checked = 1
            db.session.add(g)
    try:
        db.session.commit()
        print('done')
    except Exception as e:
        print(e)
        db.session.rollback()


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')


db.drop_all()
db.create_all()
test_create_dag()
test_check_all_graph()
test_start() # p_id = 1
#
import time
print(end='\n'*10)
time.sleep(1)

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

while True:
    ps = show_pipeline(1)  # 返回运行节点列表
    print('-' * 30)
    print(ps)
    print('-' * 30)
    time.sleep(1)
```

```
    print('~~~~~~~~~~~~~~ sleeping ~~~~~~~~~~~~~')
    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
```