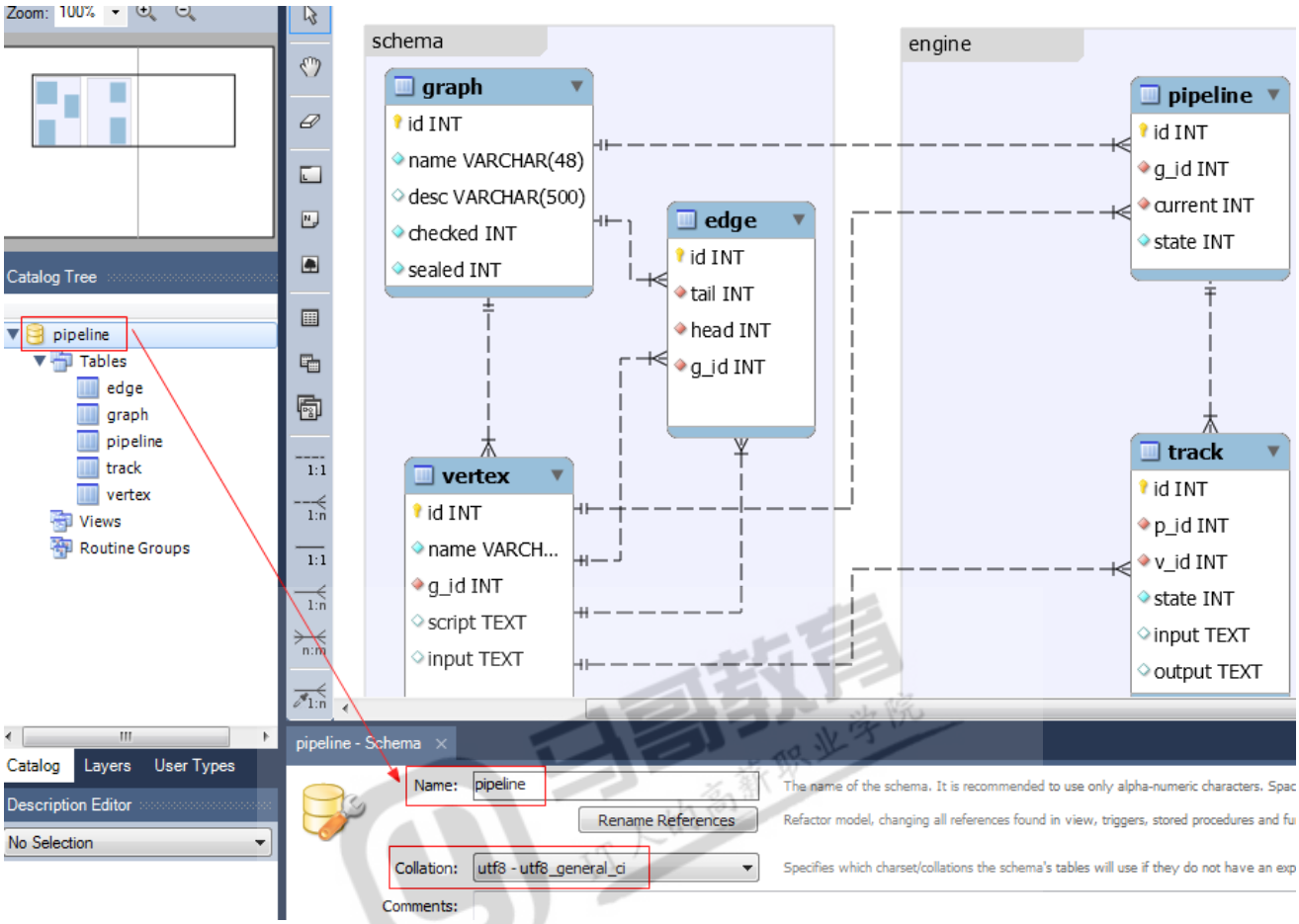
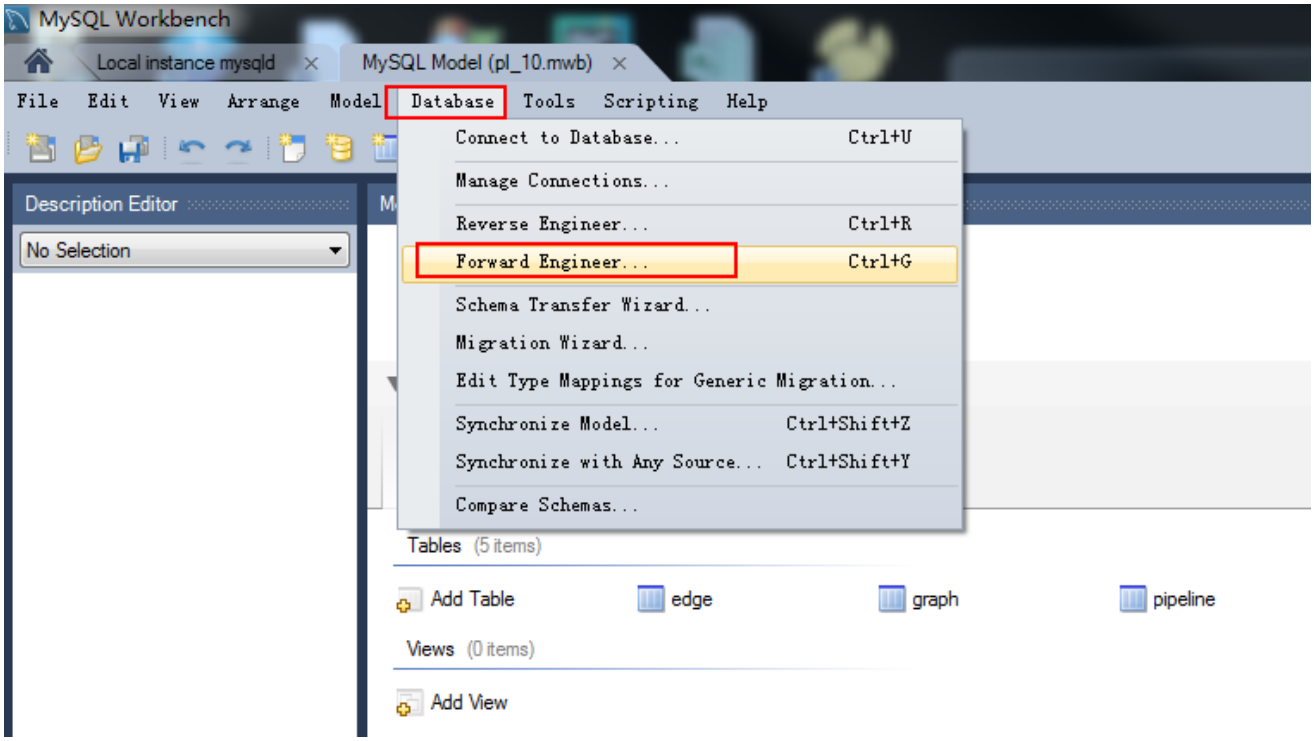
流程系统



修改Schema即数据库的名称为pipeline，然后使用模型生成数据库的表。



将模型生成数据库中的表。

代码实现

项目构建

新建一个项目，构建一个包pipeline。
包下有：
config.py 公共配置
model.py ORM映射

配置文件

config.py

```
USERNAME = 'wayne'
PASSWD = 'wayne'
DBIP = '192.168.142.140'
DBPORT = 3306
DBNAME = 'pipeline'

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

DATABASE_DEBUG = True
```

单例模式

一个类只能实例化一次，只能拥有一个实例

```python
# 实现 1

class A:
    def __new__(cls, *args, **kwargs):
        print('~~~~~~~~~~~~')
        print(cls)
        print(args)
        print(kwargs)
        if not hasattr(cls, '_instance'):
            setattr(cls, '_instance', super().__new__(cls))
            setattr(cls, '_count', 0)
        return cls._instance

    def __init__(self, url, debug):
        print('============')
        if self._count == 0:
            self.url = url
            self.debug = debug
            self.__class__._count = 1
        else:
            raise Exception('Just One Instance')

    def __repr__(self):
        return "<B {} {}>".format(self.url, self.debug)


b = A(1, debug=2)
print(b.__dict__)
import time
time.sleep(2)
b1 = A(10, 20)
print(b1.__dict__)
```

装饰器实现

```python
# 单例装饰器
import functools

def singleton(cls):
    instance = None

    @functools.wraps(cls)
    def getinstance(*args, **kwargs):
        nonlocal instance
        if not instance:
            print(args)
            print(kwargs)
            instance = cls(*args, **kwargs)
        return instance
    return getinstance

@singleton
```

```
class B:
    '''class B'''
    def __init__(self, url, debug):
        self.url = url
        self.debug = debug

b = B(1,2)
print(id(b), b.__dict__, b.__doc__)

b1 = B(10, 20)
print(id(b1), b1.__dict__)
```

Model层

创建ORM，封装数据操作类

```
$ pip install sqlalchemy pymysql
```

model.py

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


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

Base = declarative_base()

# schema定义
# 图
class Graph(Base):
    __tablename__ = "graph"

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(48), nullable=False, unique=True)
    desc = Column(String(500), nullable=True)

    # 经常从图查看所有顶点、边的信息
    vertexes = relationship('Vertex')
    edges = relationship('Edge')

# 顶点表
class Vertex(Base):
    __tablename__ = "vertex"
```

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

    graph = relationship('Graph')
    # 从顶点查它的边，这里必须使用foreign_keys，其值必须使用引号
    tails = relationship('Edge', foreign_keys='[Edge.tail]')
    heads = relationship('Edge', foreign_keys='Edge.head')


# 边表
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)

# Engine
# pipeline表
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)
    state = Column(Integer, nullable=False, default=STATE_WAITING)

    vertex = relationship('Vertex')

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, nullable=False, default=STATE_WAITING)
    input = Column(Text, nullable=True)
    output = Column(Text, nullable=True) # 任务输出

    vertex = relationship('Vertex')
    pipeline = relationship('Pipeline') # 后面使用方便


# 封装数据库的引擎、会话到类中
# 单例模式
import functools

def singleton(cls):
```

```
    instance = None

    @functools.wraps(cls)
    def getinstance(*args, **kwargs):
        nonlocal instance
        if not instance:
            print(args)
            print(kwargs)
            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

    # 创建表
    def create_all(self):
        Base.metadata.create_all(self._engine)
    # 删除表
    def drop_all(self):
        Base.metadata.drop_all(self._engine)

# 模块加载一次，db也是单例的
db = Database(config.URL, echo=config.DATABASE_DEBUG)
```

service层

需求
1、定义流程DAG，即Schema定义。
2、执行某一个DAG的流程。

问题
DAG是否允许修改？
可以这样考虑，如果DAG定义好还未使用，可以修改，一旦被使用过，不许修改。
所谓使用过，就是pipeline表中使用到了graph的主键id，或者在graph表中增加一个字段表示是否被使用过。

DAG定义

service.py

```
from .model import db
```

```python
from .model import Graph, Vertex, Edge
from .model import Pipeline, Track

# 创建DAG
def create_graph(name, desc=None):
    g = Graph()
    g.name = name
    g.desc = desc

    db.session.add(g)
    try:
        db.session.commit()
        return g
    except:
        db.session.rollback()

# 为DAG增加顶点
def add_vertex(graph:Graph, name:str, input=None, script=None):
    v = Vertex()
    v.g_id = graph.id
    v.name = name
    v.input = input
    v.script = script

    db.session.add(v)
    try:
        db.session.commit()
        return v
    except:
        db.session.rollback()

# 为DAG增加边
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)
    try:
        db.session.commit()
        return e
    except:
        db.session.rollback()

# 删除顶点
# 删除顶点就要删除所有顶点关联的边
def del_vertex(id):
    query = db.session.query(Vertex).filter(Vertex.id == id)
    v = query.first()
    if v: # 找到顶点后，删除关联的边，然后删除顶点
        try:
            db.session.query(Edge).filter((Edge.tail == v.id) | (Edge.head == v.id)).delete()
```

```
            query.delete()
            db.session.commit()
        except:
            db.session.rollback()
    return v

## 其它增删改方法，都差不多，不再赘述
```

通过上面的代码，可以发现事务的处理代码都差不多，提出来使用**装饰器**。

```
from .model import db
from .model import Graph, Vertex, Edge
from .model import Pipeline, Track

from functools import wraps

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

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

# 为DAG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
# 删除顶点就要删除所有顶点关联的边
@transactional
def del_vertex(id):
    query = db.session.query(Vertex).filter(Vertex.id == id)
    v = query.first()
    if v: # 找到顶点后，删除关联的边，然后删除顶点
        db.session.query(Edge).filter((Edge.tail == v.id) | (Edge.head == v.id)).delete()
        query.delete()
    return v

## 其它增删改方法，都差不多，不再赘述
```

测试数据

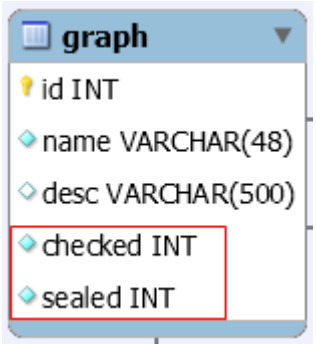
编写test.py，测试函数

```
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":'192.168.0.100'
            }
        }

        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, 'echo B')
        c = add_vertex(g, 'C', None, 'echo C')
```

```
        d = add_vertex(g, 'D', None, '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, 'echo A')
        b = add_vertex(g, 'B', None, 'echo B')
        c = add_vertex(g, 'C', None, 'echo C')
        d = add_vertex(g, 'D', None, '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, 'echo A')
        b = add_vertex(g, 'B', None, 'echo B')
        c = add_vertex(g, 'C', None, 'echo C')
        d = add_vertex(g, 'D', None, '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, 'echo A')
        b = add_vertex(g, 'B', None, 'echo B')
        c = add_vertex(g, 'C', None, 'echo C')
        d = add_vertex(g, 'D', None, '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)

test_create_dag()
```

DAG验证

当增加一个DAG定义后，或修改了DAG定义，就需要对DAG进行验证，判断是否是一个DAG图。如何知道一个写入数据库的DAG是有效的呢？在graph表增加一个字段checked，为1就是检测通过，以后可以创建一个流程执行，为0检测不通过。
注意，如果有一个流程使用了这个DAG，它就不允许被修改和删除。为了实现这个功能，且不要每一次都查询一下这个DAG被使用，可以在graph表提供一个**字段sealed**，一旦设置就不能修改和删除，表示有人用了。
在DAG定义后、修改后，就立即进行DAG检验，这样使用的时候就不用每次都检验。



图graph

字段名	类型	说明
id	int	主键
name	varchar	非空、唯一，图的名称
desc	varchar	可为空，描述
checked	int	不可为空，默认0。0表示为通过验证不能使用，1表示可以创建执行流程
sealed	int	不可为空，默认0。0表示未使用，1表示已经有执行流程使用了，被封闭不可修改

```
# 图
class Graph(Base):
    __tablename__ = "graph"

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(48), nullable=False, unique=True)
    desc = Column(String(500), nullable=True)
    checked = Column(Integer, nullable=False, default=0)
    sealed = Column(Integer, nullable=False, default=0)

    # 经常从图查看所有顶点、边的信息
    vertexes = relationship('Vertex')
    edges = relationship('Edge')
```

查找所有入度为0的顶点

```
-- 找出graph id为1的所有顶点和边
select * from vertex v INNER JOIN edge e on v.g_id = e.g_id AND v.g_id = 1

-- 找出graph id为1的顶点和边，且弧尾是顶点的，因为左联，有head为null
select * from vertex v LEFT JOIN edge e on v.g_id = e.g_id AND e.head = v.id WHERE v.g_id = 1

-- 增加一个条件edge head为null就可以提取出指定graph中入度为0的顶点
SELECT v.*
FROM vertex v LEFT JOIN edge e
ON v.g_id = e.g_id AND e.head = v.id
WHERE v.g_id = 1 AND e.head IS NULL
```

采用左联找edge里面找null的方式，找入度为0的顶点。
但是这种找法不适合验证DAG，因为第一批入度0的顶点找到后，还需要再次查询，找第二批顶点。
能否换个思路呢？
把所有的顶点、边都先查一遍，然后在客户端数据结构中想办法处理，而不是多次来查询数据库。

kahn算法实现

算法1

```
def check_graph(graph:Graph) -> bool:
    """验证是否是一个合法的DAG"""
    # 反正要遍历所有顶点和边，不如一次性把说有所有顶点和边都查回来，在内存中反复遍历
    query = db.session.query(Vertex).filter(Vertex.g_id == graph.id)
    vertexes = [vertex.id for vertex in query] # 顶点列表
    query = db.session.query(Edge).filter(Edge.g_id == graph.id)
    edges = [(edge.tail, edge.head) for edge in query]

    # ([1, 2, 3, 4], [(1, 2), (1, 3), (3, 2), (2, 4)])
    # 遍历顶点，去找
    while True:
        vis = [] # 就放一个索引，用列表是为了用的方便
        for i,v in enumerate(vertexes):
            for _,h in edges:
                if h == v: # 当前顶点有入度
                    break
            else: # 没有break，说明遍历一遍边，没有找到该顶点作为弧头，就是入度为0
                ejs = []
                for j,(t,_) in enumerate(edges):
                    if t == v: # 找这个顶点的出度的边
                        ejs.append(j)
                vis.append(i) # 待移除的入度为0的顶点的索引
                for j in reversed(ejs): # 逆向
                    edges.pop(j)
                break # 一旦找到入度为0的顶点，就需要从列表中删除，列表重新遍历
        else: # 遍历一遍剩余顶点，都没有break，说明没有找到入度0的顶点
            return False
        for i in vis:
            vertexes.pop(i)
        print(vertexes, edges)
        if len(vertexes) + 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
```

算法思路：
一次把一个DAG的所有顶点、所有边都拿回来。
遍历顶点，拿出一个顶点，就去边列表中找它是否作为弧头，如果它是弧头，立即判断下一个顶点。如果这个顶点在边列表中都没有找到它作为弧头，就是入度为0的顶点，就可以移除它作为弧尾的边和它本身了。
注意，因为移除会导致列表索引的变化，所以采用了先记录索引，后倒序删除索引的方式。
如果入度为0的顶点和它作为弧尾的有向边都移除，最后剩下一个空图，就说明此图是DAG。空图的判断使用非负整数的相加为0，一定都是0的依据。
如果一轮遍历，没有找到入度为0的顶点，说明它不是DAG。

算法1迭代次数太多了

算法2

```
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
```

算法思路

还是一次把顶点、边都从数据库拿出来，减少和数据库的交互。
顶点id不可能重复，所以采用set。
边从库中拿出的时候，就把弧尾作为字典key便于删除入度为0的顶点的边。

注意一点，只要边字典有值，就说明一定有入度不为0的顶点。
如果用当前的顶点集减去所有入度不为0的顶点集，结果有2种可能：
1、不为空集，说明这是入度为0的顶点集
2、空集，说明有环

判断依据

- 如果边字典为空退出循环，说明已经没有边了，但是顶点集可能还有顶点。

    - 如果顶点集还有顶点，都是入度为0的顶点，都可以移除的
    - 说明就是DAG
- 如果入度为0的顶点没有找到退出
    - 如果边字典不为空，说明有环

两种算法效率测试

```
def check_graph1(graph=None) -> bool:
    """验证是否是一个合法的DAG"""

    # ===============测试数据===============
    # ([1, 2, 3, 4], [(1, 2), (1, 3), (3, 2), (2, 4)])
    vertexes = [1, 2, 3, 4]
    edges = [(1, 2), (1, 3), (3, 2), (2, 4)]
    # ======================================

    # 遍历顶点，去找
    while True:
        vis = []  # 就放一个索引，用列表是为了用的方便
        for i, v in enumerate(vertexes):
            for _, h in edges:
                if h == v:  # 当前顶点有入度
                    break
            else:  # 没有break，说明遍历一遍边，没有找到该顶点作为弧头，就是入度为0
                ejs = []
                for j, (t, _) in enumerate(edges):
                    if t == v:
                        ejs.append(j)
                vis.append(i)
                for j in reversed(ejs):  # 逆向
                    edges.pop(j)
                break  # 一旦找到入度为0的顶点，就需要从列表中删除，列表重新遍历
        else:  # 遍历一遍剩余顶点，都没有break，说明没有找到入度0的顶点
            return False
        for i in vis:
            vertexes.pop(i)
        #print(vertexes, edges)
        if len(vertexes) + len(edges) == 0:
            return True
        return False


from collections import defaultdict


def check_graph2(graph=None) -> bool:
    """验证是否是一个合法的DAG"""

    # ===============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]

    # 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:
        return True
    return False


import datetime

start = datetime.datetime.now()
for _ in range(100000):
    check_graph1()
print((datetime.datetime.now() - start).total_seconds())

start = datetime.datetime.now()
for _ in range(100000):
    check_graph2()
print((datetime.datetime.now() - start).total_seconds())


# 测试结果
0.244
0.134 算法2有明显优势
```

使用算法2

```
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()

test_check_all_graph()
```

验证成功，就会设置验证的所有图定义的checked字段为1。
业务上应该在创建一个新的DAG的时候立即验证，或在修改一个DAG后立即验证。