开发

环境准备

继续使用前面项目的构建，使用虚拟环境。

安装依赖

```
$ pip install PyMySQL  SQLAlchemy webob
```

Model层

cmdb包下建立models.py

每个表增加deleted字段，所有数据都是逻辑删除，因为有可能使用，不要真删除。

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

Base = declarative_base()

# 逻辑表
class Schema(Base):
    __tablename__ = "schema"

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(48), nullable=False, unique=True)
    desc = Column(String(128), nullable=True)
    deleted = Column(Boolean, nullable=False, default=False)

    fields = relationship('Field')

class Field(Base):
    __tablename__ = "field"
    __table_args__ = (UniqueConstraint('schema_id', 'name'),)

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(48), nullable=False)
    schema_id = Column(Integer, ForeignKey('schema.id'), nullable=False)
    meta = Column(Text, nullable=False)
    ref_id = Column(Integer, ForeignKey('field.id'), nullable=True)
    deleted = Column(Boolean, nullable=False, default=False)

    schema = relationship('Schema')
    ref = relationship('Field', uselist=False) # 1对1，被引用的id
```

```python
# 逻辑表的记录表
class Entity(Base):
    __tablename__ = "entity"

    id = Column(BigInteger, primary_key=True, autoincrement=True)
    key = Column(String(64), nullable=False, unique=True)
    schema_id = Column(Integer, ForeignKey('schema.id'), nullable=False)
    deleted = Column(Boolean, nullable=False, default=False)

    schema = relationship('Schema')

class Value(Base):
    __tablename__ = "value"
    __table_args__ = (UniqueConstraint('entity_id', 'field_id', name='uq_entity_field'),)

    id = Column(BigInteger, primary_key=True, autoincrement=True)
    value = Column(Text, nullable=False)
    field_id = Column(Integer, ForeignKey('field.id'), nullable=False)
    entity_id = Column(BigInteger, ForeignKey('entity.id'), nullable=False)
    deleted = Column(Boolean, nullable=False, default=False)

    entity = relationship('Entity')
    field = relationship('Field')


# 引擎
engine = create_engine(config.URL, echo=config.DATABASE_DEBUG)

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

Session = sessionmaker(bind=engine)
session = Session()
```

使用models.py来drop所有表并生成所有表，然后执行下面的实验语句

```sql
-- 表1 虚拟表host，有2个字段hostname、ip
INSERT INTO `schema` (name) VALUES('host');
INSERT INTO `field` (name, schema_id) VALUES ('hostname', 1);
INSERT INTO `field` (name, schema_id) VALUES ('ip', 1);

-- 表2 虚拟表ippool，有1个字段ip
INSERT INTO `schema` (name) VALUES('ippool');
INSERT INTO `field` (name, schema_id) VALUES ('ip', 2);

-- host表添加记录
INSERT INTO entity (`key`, schema_id) VALUES ('5846d1499dd544198475a9d517766494', 1);
INSERT INTO `value`(entity_id, field_id, `value`) VALUES(1, 1, 'webserver');
```

```
INSERT INTO `value`(entity_id, field_id, `value`) VALUES(1, 2, '192.168.1.10');

INSERT INTO entity (`key`, schema_id) values ('0f51405a04344f0e9f11109895ab2f19', 1);
INSERT INTO `value`(entity_id, field_id, `value`) VALUES(2, 1, 'DBserver');
INSERT INTO `value`(entity_id, field_id, `value`) VALUES(2, 2, '192.168.1.20');

INSERT INTO entity (`key`, schema_id) VALUES ('587723df88a54b2e9f449888d75f50de', 1);
INSERT INTO `value` (entity_id, field_id, `value`) VALUES(3, 1, 'DNS Server');
INSERT INTO `value` (entity_id, field_id, `value`) VALUES(3, 2, '172.16.100.1');

-- ip表添加记录
INSERT INTO entity (`key`, schema_id) VALUES('3dea5d2e39eb47b5a5b95cee6fc64f8d', 2);
INSERT INTO entity (`key`, schema_id) VALUES('6bbd0d91e6cf44cba7e71207ddaa06d6', 2);
INSERT INTO entity (`key`, schema_id) VALUES('fc377c758e5a463cb246ff693ab11434', 2);
INSERT INTO `value` (entity_id, field_id, `value`) VALUES(4, 3, '192.168.1.10');
INSERT INTO `value` (entity_id, field_id, `value`) VALUES(5, 3, '192.168.1.20');
INSERT INTO `value` (entity_id, field_id, `value`) VALUES(6, 3, '192.168.1.30');

-- 查询
SELECT
`schema`.id AS sid,
`schema`.`name` AS sname,
entity.id AS eid,
entity.`key`,
field.id AS fid,
field.`name` AS fname,
`value`.id,
`value`.`value`
FROM
`value`
INNER JOIN entity ON `value`.entity_id = entity.id AND entity.deleted = FALSE
INNER JOIN `schema` ON entity.schema_id = `schema`.id AND `schema`.deleted = FALSE
INNER JOIN field ON `value`.field_id = field.id AND field.deleted = FALSE
WHERE `value`.deleted = FALSE
```

问题
schema中设置了name为unique，但是如果删除一个逻辑表后，加入一个同名的逻辑表名，就会报错。
field表中使用schema_id和name构成unique，也一样有这种问题。
问题的关键在于怕名称冲突，而名称又是给人看的，所以加上deleted字段构成unique。
这样看似解决了删除一个名称，再次输入同名的问题，但是解决不了这个名称再次删除后的unique冲突。
所以依靠物理表的unique，不好解决这个问题，还需要自己编码最终解决这个问题。

meta处理

field表中meta字段，需要对json数据进行包装。

这里略作一些改动
1、把option放到type中，因为它和type类型有关。
2、引用仿照MySQL，指定表名和字段名。因为字段有可能删除后重新添加同名的字段，id就变了

```
{
  "type": {
```

```
    "name": "cmdb.types.IP",
    "option": {
      "prefix": "192.168"
    }
  },
  "value": "192.168.0.1,192.168.0.2",
  "nullable": true,
  "unique": false,
  "default": "",
  "multi": true,
  "reference": {
    "schema": "ippool",
    "field": "ip",
    "on_delete": "cascade|set_null|disable",
    "on_update": "cascade|disable"
  }
}
```

type简化写法

```
{
  "type": "cmdb.types.IP",
  "unique": true
}
```

在models.py中增加meta解析类，并未Field提供一个属性

```
import json
from .types import get_instance

class Reference:
    def __init__(self, ref:dict):
        self.schema = ref['schema'] # 引用的schema
        self.field = ref['field'] # 引用的field
        self.on_delete = ref.get('on_delete', 'disable')  # cascade,set_null,disable
        self.on_update = ref.get('on_update', 'disable')  # cascade,disable

class FieldMeta:
    def __init__(self, metastr:str):
        meta = json.loads(metastr)

        if isinstance(meta, str):
            self.instance = get_instance(meta['type'])
        else:
            option = meta['type'].get('option')
            if option:
                self.instance = get_instance(meta['type']['name'], **option)
            else:
                self.instance = get_instance(meta['type']['name'])
        self.unique = meta.get('unique', False)
        self.nullable = meta.get('nullable', True)
```

```
        self.default = meta.get('default')
        self.multi = meta.get('multi', False)
        # 引用是一个json对象
        ref = meta.get('reference')
        if ref:
            self.reference = Reference(ref)
        else:
            self.reference = None

class Field(Base):
    __tablename__ = "field"
    __table_args__ = (UniqueConstraint('schema_id', 'name'),)

    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(48), nullable=False)
    schema_id = Column(Integer, ForeignKey('schema.id'), nullable=False)
    meta = Column(Text, nullable=False)
    ref_id = Column(Integer, ForeignKey('field.id'), nullable=True)
    deleted = Column(Boolean, nullable=False, default=False)

    schema = relationship('Schema')
    ref = relationship('Field', uselist=False) # 1对1，被引用的id

    @property # 增加一个属性将meta解析成对象，注意不要使用metadata这个名字
    def meta_data(self):
        return FieldMeta(self.meta)
```

Service层

日志

在app.py中加入日志配置

```
import logging

#logging.basicConfig(level=logging.INFO, format="%(asctime)s [%(name)s %(funcName)s] %
(message)s")
```

这是全局设置，影响范围很大。sqlalchemy也会受影响。
所以模块单独设置自己的logger，就不在app.py设置了。

cmdb包下建立service模块，在`__init__.py`中编写

```
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO) # 单独设置
logger.propagate = False # 阻止传送给父logger

handler = logging.FileHandler('o:/test.log')
handler.setLevel(logging.INFO)
formatter = logging.Formatter(fmt="%(asctime)s [%(name)s %(funcName)s] %(message)s")
handler.setFormatter(formatter)
logger.addHandler(handler)
```

由于其他模块也使用，所以封装成函数，放到cmdb/utils.py

```
import logging

def getlogger(mod_name:str, filepath:str):
    logger = logging.getLogger(mod_name)
    logger.setLevel(logging.INFO)  # 单独设置
    logger.propagate = False  # 阻止传送给父logger
    handler = logging.FileHandler(filepath)
    handler.setLevel(logging.INFO)
    formatter = logging.Formatter(fmt="%(asctime)s [%(name)s %(funcName)s] %(message)s")
    handler.setFormatter(formatter)
    logger.addHandler(handler)
    return logger
```

模块使用

```
from ..utils import getlogger

logger = getlogger(__name__, 'o:/{}.log'.format(__name__)) # 路径自行更换
```

schema 接口

```
from ..models import session, Schema, Field, Entity, Value
import logging
import math

logger = logging.getLogger(__name__)

# schema接口
# 返回一个schema对象
def get_schema_by_name(name:str, deleted:bool=False):
    query = session.query(Schema).filter(Schema.name == name.strip())
    if not deleted:
        query = query.filter(Schema.deleted == False)

    return query.first()
```

```
# 增加一个schema
def add_schema(name:str, desc:str=None):
    schema = Schema()
    schema.name = name.strip()
    schema.desc = desc
    session.add(schema)
    try:
        session.commit()
        return schema
    except Exception as e:
        session.rollback()
        logger.error('Fail to add a new schema {}. Error: {}'.format(name, e))

# 删除使用id，id唯一，比使用name删除好
def delete_schema(id:int):
    try:
        schema = session.query(Schema).filter((Schema.id == id) & (Schema.deleted == False))
        if schema:
            schema.deleted = True
            session.add(schema)
            try:
                session.commit()
                return schema
            except Exception as e:
                session.rollback()
                raise e
        else:
            raise ValueError('Wrong ID {}'.format(id))
    except Exception as e:
        logger.error('Fail to del a schema. id = {}. Error: {}'.format(id, e))

# 列出所有逻辑表
def list_schema(page:int, size:int, deleted:bool=False):
    try:
        query = session.query(Schema)
        if not deleted:
            query = query.filter(Schema.deleted == False)

        page = page if page>0 else 1
        size = size if 0 < size < 101 else 20
        count = query.count()
        pages = math.ceil(count/size)
        result = query.limit(size).offset(size*(page - 1)).all()

        return result,(page,size,count,pages)
    except Exception as e:
        logger.error()
```

list_schema 方法是列表显示所有信息，其它信息显示，实际上也要用这个通用逻辑，所以，抽出一个函数。

```
# 列出所有逻辑表
def list_schema(page:int=1, size:int=20, deleted:bool=False):
```

```
    query = session.query(Schema)
    if not deleted:
        query = query.filter(Schema.deleted == False)

    return paginate(page, size, query)

# 通用分页函数
def paginate(page, size, query):
    try:
        page = page if page > 0 else 1
        size = size if 0 < size < 101 else 20

        count = query.count()
        pages = math.ceil(count / size)
        result = query.limit(size).offset(size * (page - 1)).all()
        return result, (page, size, count, pages)
    except Exception as e:
        logger.error("{}".format(e))
```

field接口

```
# field接口
# 获取字段
def get_field(schema_name, field_name, deleted=False):
    schema = get_schema_by_name(schema_name)
    if not schema:
        raise ValueError('{} is not a Tablename'.format(schema_name))

    query = session.query(Field).filter((Field.schema_id == schema.id) & (Field.name ==
field_name))
    if not deleted:
        query = query.filter(Field.deleted == False)
    return query.first()

# 逻辑表是否已经使用
def table_used(schema_id, deleted=False):
    query = session.query(Entity).filter(Entity.schema_id == schema_id)
    if not deleted:
        query = query.filter(Entity.deleted == False)
    return query.first() is not None

# 直接添加字段
def _add_field(field:Field):
    session.add(field)
    try:
        session.commit()
        return field
    except Exception as e:
        session.rollback()
        logger.error('Failed to add a field {}. Error: {}'.format(field.name, e))

# 2种情况：1完全新增 2已有表增加字段
```

```
def add_field(schema_name, name, meta):
    schema = get_schema_by_name(schema_name)
    if not schema:
        raise ValueError('{} is not a Tablename'.format(schema_name))

    # 解析meta，from ..models import FieldMeta
    meta_data = FieldMeta(meta)
    field = Field()
    field.name = name.strip()
    field.schema_id = schema.id
    field.meta = meta # 能解析成功说明符合格式要求
    # ref_id 引用
    if meta_data.reference:
        ref = get_field(meta_data.reference.schema, meta_data.reference.field)
        if not ref:
            raise TypeError('Wrong Reference {}.{}'.format(
                meta_data.reference.schema, meta_data.reference.field))
        field.ref_id = ref.id
    # 判断字段是否已经使用
    if not table_used(schema.id): # 未使用的逻辑表，直接加字段
        return _add_field(field)

    # 已使用的逻辑表
    if meta_data.nullable: # 可以为空，直接加字段
        return _add_field(field)

    # 到这里已经有一个隐含条件即不可为空
    if meta_data.unique: # 必须唯一
        # 当前的条件是 对一个正在使用的逻辑表加字段不可以为空又要唯一，做不到
        raise TypeError('This field is required an unique.')

    # 到这里的隐含条件是，不可以为空，但可以不唯一
    if not meta_data.default: # 没有缺省值
        raise TypeError('This field requires a default value.')
    else:
        # 为逻辑表所有记录增加字段，操作entity表
        entities = session.query(Entity).filter((Entity.schema_id == schema.id) &
(Entity.deleted == False)).all()
        for entity in entities: # value表新增记录
            value = Value()
            value.entity_id = entity.id
            value.field = field
            value.value = meta_data.default
            session.add(value)
        return _add_field(field)
```

上面的代码中，最后一个遍历entity表，可以考虑使用生成器

```
def add_field(schema_name, name, meta):
    # ..... 省略

    # 到这里的隐含条件是，不可以为空，但可以不唯一
```

```
    if not meta_data.default: # 没有缺省值
        raise TypeError('This field requires a default value.')
    else:
        # 为逻辑表所有记录增加字段，操作entity表
        for entity in iter_entities(schema.id): # value表新增记录
            value = Value()
            value.entity_id = entity.id
            value.field = field
            value.value = meta_data.default
            session.add(value)
        return _add_field(field)


def iter_entities(schema_id, patch=100):
    page = 1
    while True:
        query = session.query(Entity).filter((Entity.schema_id == schema_id) & (Entity.deleted
== False))
        result = query.limit(patch).offset((page - 1) * patch).all()
        if not result:
            return None
        yield from result
        page += 1
```

说明
可以拿到增加字段已经非常繁琐了，修改字段也是一样。
生产环境中，对已经使用的逻辑表，除非万不得已，否则不要增加和修改字段。

到目前为止，代码已经基本说明如何实现这个cmdb库了。剩下的按照设计完成即可。

总结

本项目通过MySQL数据的设计，实现了一个复杂的cmdb。
目前这个cmdb功能还是比较简陋的，很多功能还未实现。例如如何实现锁机制等。
但是到目前为止，基本功能已经可以实现和完成了，其实生产环境中真正用的功能也就是现在这样了。
及时花了很大的功夫，把更加复杂的功能实现了，也未必有用户使用。

本项目学习目的
1、学习数据库设计
2、进一步巩固SqlAlchemy使用
3、学习Service层的写法
4、巩固日志的使用
5、学习插件化开发思想
6、学习复杂逻辑的设计、开发，锻炼严谨的思维能力，应用到项目开发中