# CMDB

## 概念

CMDB，configuration Management DB，配置管理数据库。
配置管理不是用来管理配置文件的，而是管理资产的。

狭义的CMDB是偏向纯资产管理的，而宽泛的CMDB集成了很多其他功能，已经发展成一个运维管理信息系统了。
因为，运维系统往往围绕着这些资产管理中心，从这个CMDB中获取公共的数据。

我们今天讲回归CMDB的本源，学习如何设计一个CMDB，核心是DB。

## 设计

本次工具是Mysql提供的Workbench

### 练习

请设计数据库，以实现存储主机信息、交换机信息

### 原始版本

资产管理的初期，有需求需要把凌乱、分散的各种软硬件、配置信息集中管理。
构建MySQL数据库的表，将数据放置其中。
由于管理的资产的多样性，表如果没有很好的设计，就会很难适应需求的需要，经常需要改动，例如增加字段来适应存储更多的属性。
管理服务器、硬件防火墙需要的字段不一样，管理主机信息、数据库信息字段也不一样。而这些信息都需要管理，但又不能每一种设备单独建表。

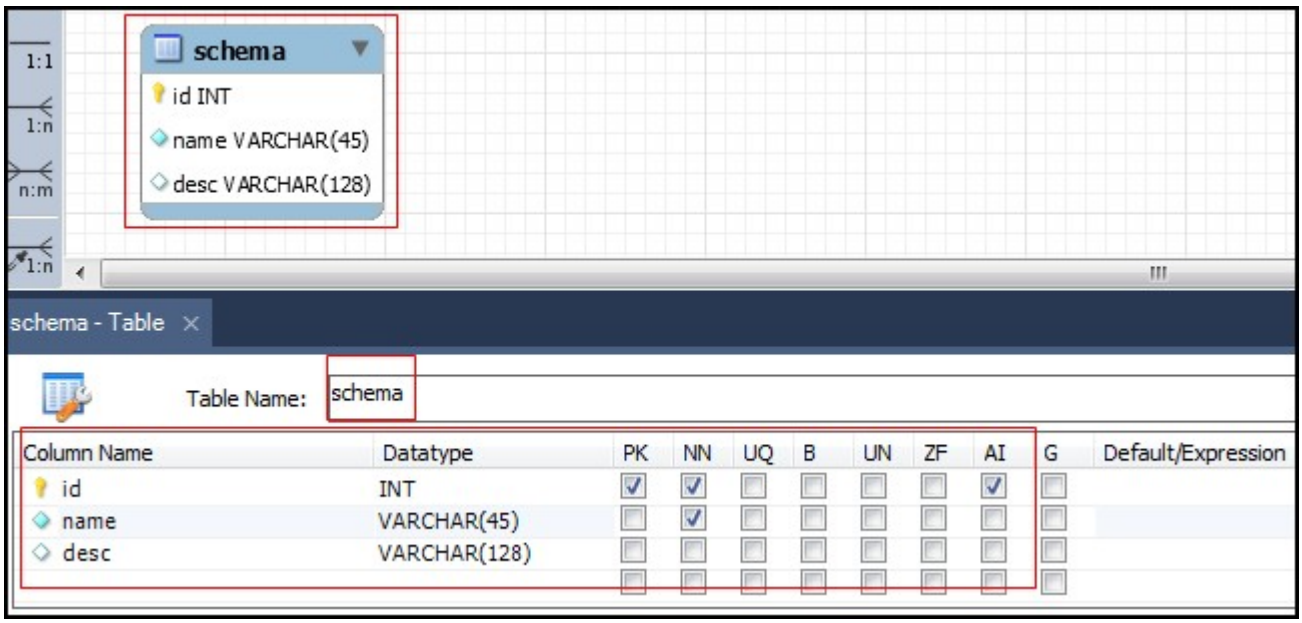
### 进化版本——CMDB实现

依然基于MySQL来使用CMDB。

充分考虑，运维日常管理的信息复杂性，将管理的信息所需要的表、字段、值抽离出去形成不同的表。

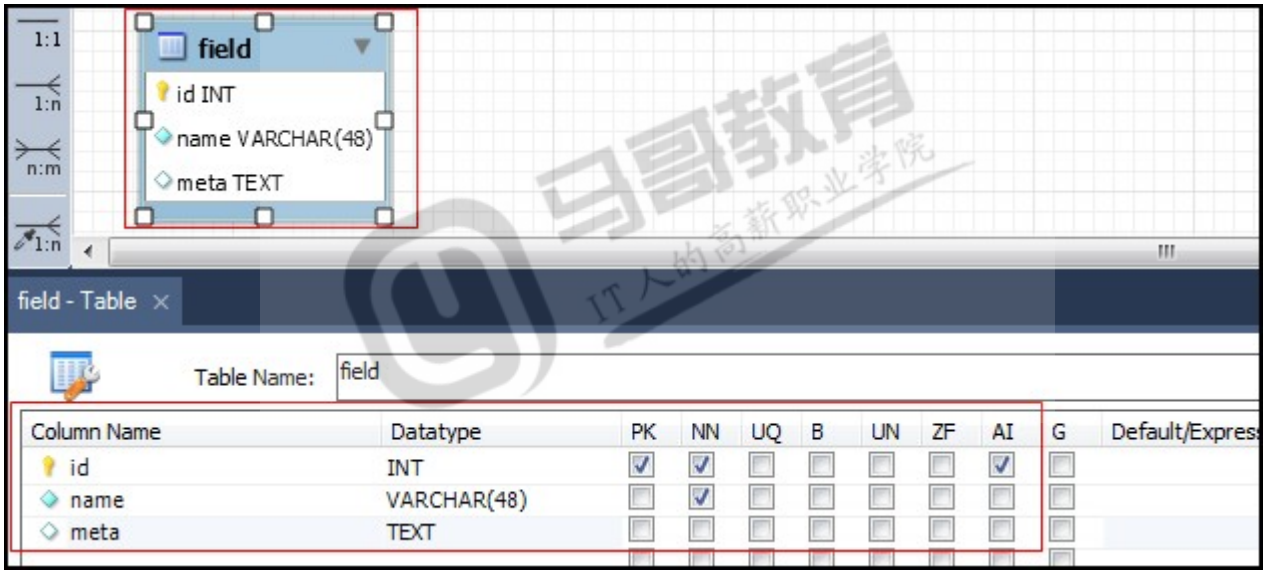
**1 建立数据库叫cmdb**

**2 schema 模型表**



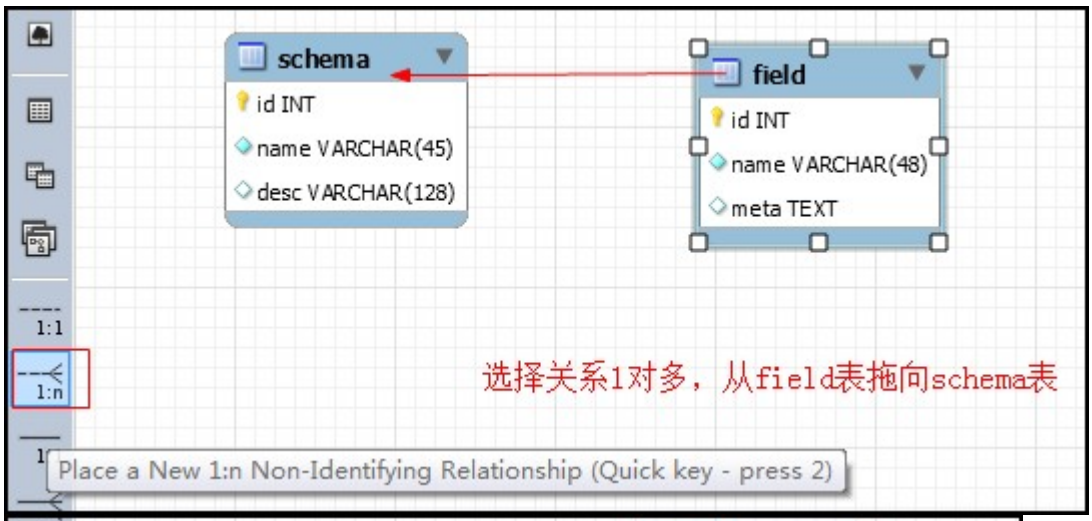

**3 field 字段表**

描述可用字段表，用来给模型表动态增加描述字段




meta，字段的元数据。

建立关系





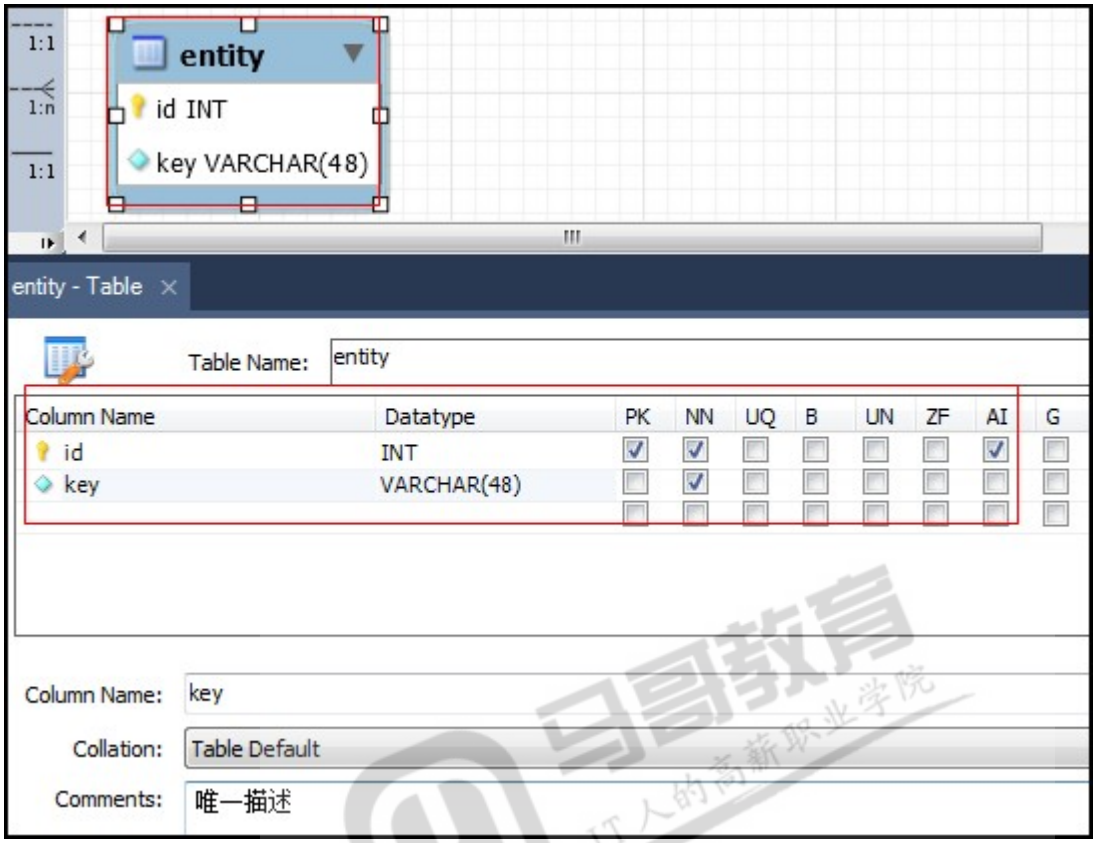
一对多关系，在多端加字段schema_id解决。

描述资产是非常难的事情，不同资产，或不同资产的不同型号，就有着不同的属性和值。很难在一张表中设计固定个数的字段保存数据，所以，这里使用2张表来描述，schema中建立一种资产，就可以为其在field表中建立很多记录来描述属性，以后要加属性只需要在field表中增加一条记录描述。
可以使用 schema + field 构成了一张张虚拟的表定义。

每一个schema的id对应一张表，例如ipaddress表对应的schema_id 为10。
它有2个字段，字段的描述在field表中，（field.id=1,name）和字段（field.id=5,name）。
这2部分构成一张表的定义。

**4 entity 实体表**

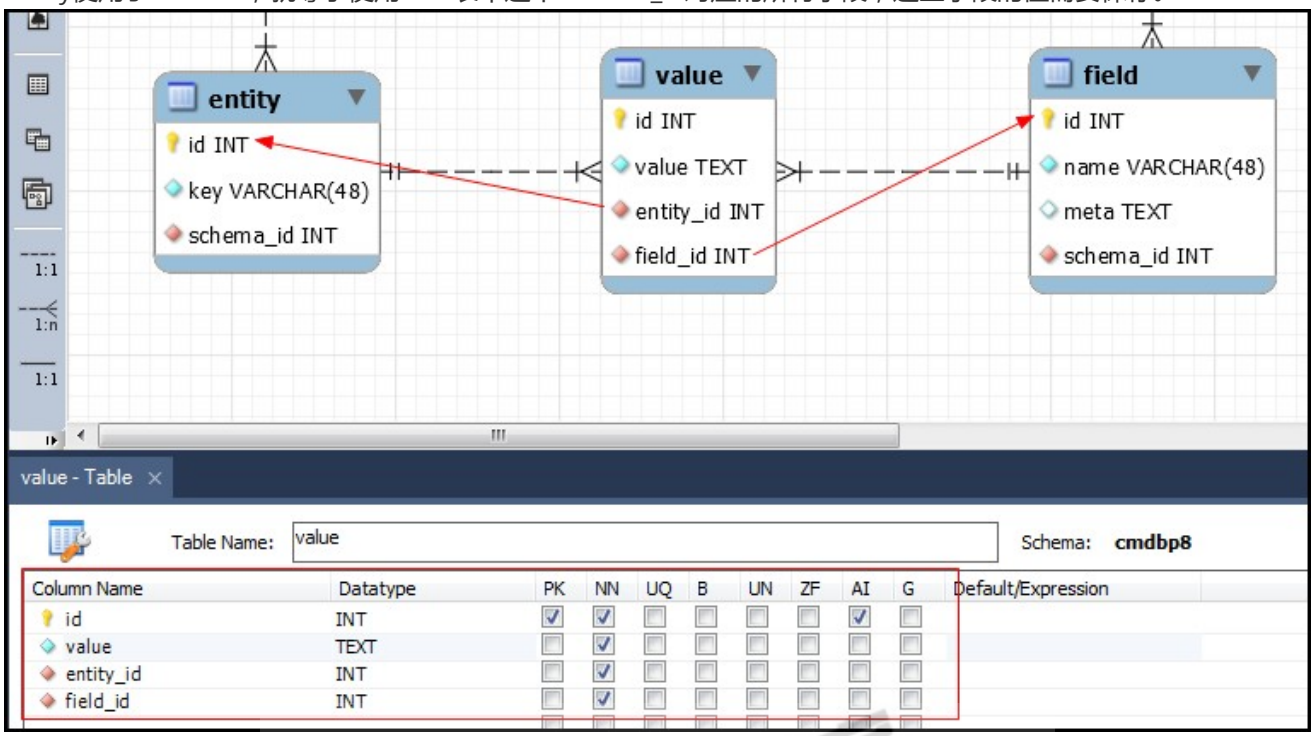
实体数据使用哪一个schema




key使用uuid来描述一个唯一值

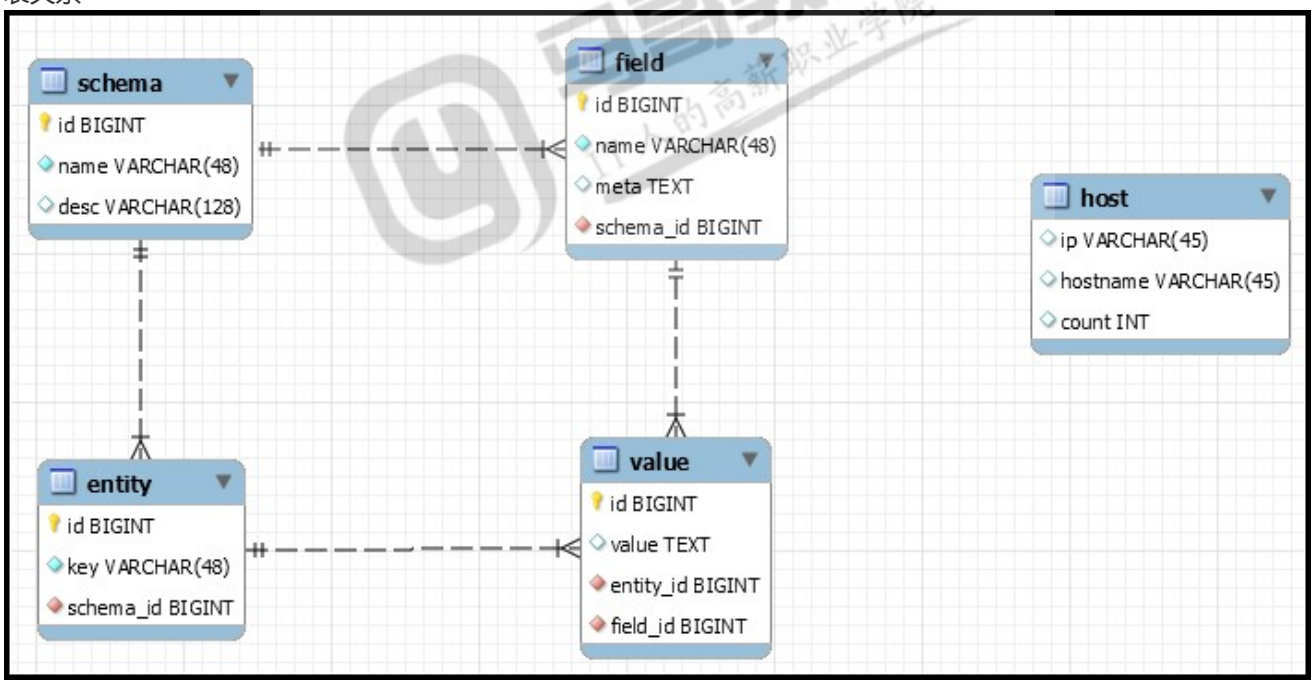
**5 value 记录表**

entity使用了schema，就等于使用field表中这个schema_id对应的所有字段，这些字段的值需要保存。




明确的一个数据，通过entity_id就等于知道了使用哪一张虚拟表，就可以在value表中为虚拟表的字段填入值了。

表关系




注意：右边的host表是假想的虚拟表，仅作参考

全部建表语句

```
SET FOREIGN_KEY_CHECKS=0;

-- ----------------------------
-- Table structure for entity
-- ----------------------------
DROP TABLE IF EXISTS `entity`;
```

```
CREATE TABLE `entity` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `key` varchar(48) NOT NULL COMMENT '唯一描述',
  `schema_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_entity_schema1_idx` (`schema_id`),
  CONSTRAINT `fk_entity_schema1` FOREIGN KEY (`schema_id`) REFERENCES `schema` (`id`) ON DELETE
NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------
-- Table structure for field
-- ----------------------------
DROP TABLE IF EXISTS `field`;
CREATE TABLE `field` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(48) NOT NULL,
  `meta` text,
  `schema_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_field_schema_idx` (`schema_id`),
  CONSTRAINT `fk_field_schema` FOREIGN KEY (`schema_id`) REFERENCES `schema` (`id`) ON DELETE NO
ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------
-- Table structure for schema
-- ----------------------------
DROP TABLE IF EXISTS `schema`;
CREATE TABLE `schema` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `desc` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------
-- Table structure for value
-- ----------------------------
DROP TABLE IF EXISTS `value`;
CREATE TABLE `value` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `value` text NOT NULL,
  `entity_id` int(11) NOT NULL,
  `field_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `index4` (`entity_id`,`field_id`),
  KEY `fk_value_entity1_idx` (`entity_id`),
  KEY `fk_value_field1_idx` (`field_id`),
  CONSTRAINT `fk_value_entity1` FOREIGN KEY (`entity_id`) REFERENCES `entity` (`id`) ON DELETE
NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT `fk_value_field1` FOREIGN KEY (`field_id`) REFERENCES `field` (`id`) ON DELETE NO
ACTION ON UPDATE NO ACTION
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

**举例**

定义一张主机表来管理主机。

```
insert into `schema`(name) values('host');
-- 假设插入成功，id为1
select * from `schema`;
insert into `field` (name, schema_id) values ('hostname', 1);
insert into `field` (name, schema_id) values ('ip', 1);
select * from `field`;

select * from `schema`,`field` where `field`.schema_id = `schema`.id and `schema`.id = 1;
```

上面的语句就可以成功创建一张虚拟的host表，有2个字段hostname、ip。

假设现在需要记录一条主机信息，hostname为webserver，ip为192.168.1.10。

```
insert into entity (`key`, schema_id) values ('5846d1499dd544198475a9d517766494', 1);
select * from entity;
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);
select * from entity;
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');
```

好，这样就把数据存进去了。

查询看看

```
SELECT
    entity.id as entity_id, entity.`key`, entity.schema_id,
    `schema`.`name`,
    field.id, field.`name` AS fname,
    `value`.`value`
FROM
    entity
INNER JOIN `value` ON `value`.entity_id = entity.id
INNER JOIN `schema` ON entity.schema_id = `schema`.id
INNER JOIN field ON `value`.field_id = field.id
```

信息 结果1 概况 状态

| entity_id | key | schema_id | name | id | fname | value |
|---|---|---|---|---|---|---|
| 1 | 5846d1499dd544198475a9d517766494 | 1 | host | 1 | hostname | webserver |
| 1 | 5846d1499dd544198475a9d517766494 | 1 | host | 2 | ip | 192.168.1.10 |
| 2 | 0f51405a04344f0e9f11109895ab2f19 | 1 | host | 1 | hostname | DBserver |
| 2 | 0f51405a04344f0e9f11109895ab2f19 | 1 | host | 2 | ip | 192.168.1.20 |

**好处：**
不管需要多少配置项，需要就在schema表中添加。
不管理配置项有多少属性字段，需要就在field表中增加即可。

**坏处：**
表结构复杂了，关系复杂了，原来一张张表变成了多张表的关系。复杂的同时，带来了灵活。
ORM不认识这种表，需要自己封装实现。

**问题**
有这样一个需求，host中业务需要所有主机ip不可重复记录，也就是一台主机占用这个IP地址并记录了，不可以重复记录，也不可以其他主机使用这个IP了。如何实现？
为value表的value字段加上unique唯一约束行吗？
看似可以，但是这个value字段还要记录其它虚拟表的值，可能会写不进来数据。例如，记录数据库服务器地址，可能使用了这个IP，就会写入失败，因为value字段唯一键约束。

如何使用约束？
数据库的约束无法使用了，只能在应用层代码上解决了。
这个时候，刚才在field表中预留的meta字段就派上用场了。

# 约束设计

## 思路1

meta字段中存储一个校验字段的方式
1、数据类型
例如int、str，这些都可以被程序后续用作类型转换。
但是如果是IP地址，怎么转换？或者更加复杂类型，如何实现
2、如何校验
可以使用正则表达式。
正则表达式较难掌握，实现规则多样，复杂难以测试，且不是面向对象的实现方式。

## 思路2

meta是text类型，就是字符串。是否可以使用Json来描述呢？
将Json中的字符串变成Python代码运行，使用反射动态加载运行。
这样实现需要约定好调用的接口。

# 开发

构建项目，名称为cmdb。
使用虚拟环境，Python版本3.5

本项目主要代码都放在cmdb包下。

# 约束实现

cmdb.types包里面定义类BaseType和其子类Int

```
class BaseType:
    '''cmdb 类型基类'''
    def stringify(self, value):
        '''基类方法，未实现'''
        raise NotImplementedError()

    def destringify(self, value):
        '''基类方法，未实现'''
        raise NotImplementedError()


class Int(BaseType):
    def stringify(self, value):
        return str(int(value))

    def destringify(self, value):
        return value
```

项目根目录下，建立app.py

```
import json

# 模拟字段保存的json字符串
jsonstr = """
{
    "type":"cmdb.types.Int",
    "value":300
}
"""

obj = json.loads(jsonstr)
print(obj)

# 结果为 {'type': 'cmdb.types.Int', 'value': 300}
```

`'cmdb.types.Int'`这个字符串中包含我们要的类型信息。如何取？
1、建立字典映射
2、使用反射
采用第二种反射方式完成。

```
import json

# 模拟字段保存的json字符串
jsonstr = """
{
    "type":"cmdb.types.Int",
    "value":300
}
```

```
"""

obj = json.loads(jsonstr)
print(obj)

# 结果为 {'type': 'cmdb.types.Int', 'value': 300}

import importlib
from cmdb.types import BaseType

def get_instance(type:str):
    m, c = type.rsplit('.', maxsplit=1)
    print(m,c)
    mod = importlib.import_module(m)
    cls = getattr(mod, c)
    obj = cls()
    if isinstance(obj, BaseType):
        return obj
    raise TypeError('Wrong Type {}. Not subclass of BaseType.'.format(type))

Int = get_instance(obj.get('type'))
print(Int.stringify(obj.get('value')))
```

get_instance可以放到cmdb.types模块中去。

## IP地址的约束

Python 3.3 提供了ipaddress库。

ipaddress.ip_address(address)
address可以是int或者str，返回一个IPAddress对象，使用str返回一个IP地址。

连通get_instance的移动，编写cmdb.types.IP类。

```
import importlib
import ipaddress


def get_instance(type: str):
    m, c = type.rsplit('.', maxsplit=1)
    print(m, c)
    mod = importlib.import_module(m)
    cls = getattr(mod, c)
    obj = cls()
    if isinstance(obj, BaseType):
        return obj
    raise TypeError('Wrong Type {}. Not subclass of BaseType.'.format(type))


class BaseType:
    '''cmdb 类型基类'''

    def stringify(self, value):
```

```
        '''基类方法，未实现'''
        raise NotImplementedError()

    def destringify(self, value):
        '''基类方法，未实现'''
        raise NotImplementedError()


class Int(BaseType):
    def stringify(self, value):
        """转换，错误的数据不要给默认值，就抛异常让外部捕获"""
        return str(int(value))

    def destringify(self, value):
        return value


class IP(BaseType):
    '''实现IP数据校验和转换'''

    def stringify(self, value):
        """转换，错误的数据不要给默认值，就抛异常让外部捕获"""
        return str(ipaddress.ip_address(value))

    def destringify(self, value):
        return value
```

app.py中使用如下

```
import json

# 模拟字段保存的json字符串
jsonstr = """
{
    "type":"cmdb.types.Int",
    "value":300
}
"""

obj = json.loads(jsonstr)
print(obj)

# 结果为 {'type': 'cmdb.types.Int', 'value': 300}

from cmdb.types import get_instance

Int = get_instance(obj.get('type'))
print(Int.stringify(obj.get('value')))

ipdict = {
    "type":"cmdb.types.IP",
    "value":"192.168.142.135"
```

```
}

IP = get_instance(ipdict.get('type'))
print(IP.stringify(ipdict.get('value')))
```

使用反射实现动态加载类型的方式，非常的灵活，可以扩展更多的类型，并把数据验证、转换交给类型自己完成。这是一种插件化编程思想的具体实现。

## 增加限制

例如，Int类型能否提供一个验证机制，给出函数是一种办法。
json中存函数可以做到，但Python不太好解析。可以考虑其他语言动态生成函数并验证，但是和python通信是个问题。问题复杂了。
那么，Int类型无非就是一个范围，提供最小值就是要求大于它，提供最大值，就是一定要小于它。

```
{
  "type":"cmdb.types.Int",
  "option":{
    "min":10,
    "max":30
  },
  "value":28
}
```

如何把数据送给Int类型呢？
通过构造函数送入。

```
class BaseType:
    '''cmdb 类型基类'''
    def __init__(self, **option):
        self.__dict__['option'] = option

    def __getattr__(self, item):
        # 注意，如果属性不存在，理解为不需要这个约束，返回None
        return self.option.get(item)

    def stringify(self, value):
        '''基类方法，未实现'''
        raise NotImplementedError()

    def destringify(self, value):
        '''基类方法，未实现'''
        raise NotImplementedError()
```

为了可以通过属性的方式方便地访问option，增加了`__getattr__`方法。

用户并不能直接创建BaseType或者其子类的实例，是通过get_instance方法动态创建，所以需要在这个函数中增加option参数。

```
def get_instance(type: str, **option):
    m, c = type.rsplit('.', maxsplit=1)
    print(m, c)
    mod = importlib.import_module(m)
    cls = getattr(mod, c)
    obj = cls(**option) # 从这里导入选项
    if isinstance(obj, BaseType):
        return obj
    raise TypeError('Wrong Type {}. Not subclass of BaseType.'.format(type))
```

在Int中实现，增加最大值、最小值的验证，放在stringify函数中。

```
class Int(BaseType):
    def stringify(self, value):
        """转换，错误的数据不要给默认值，就抛异常让外部捕获"""
        val = int(value)
        min = self.min
        if min and val < min:
            raise ValueError('too small')
        max = self.max
        if max and val > max:
            raise ValueError('too big')
        return str(val)

    def destringify(self, value):
        return value
```

Int 使用方式

```
import json

# 模拟字段保存的json字符串
jsonstr = """
{
    "type":"cmdb.types.Int",
    "value":800,
    "option":{
        "max":100,
        "min":1
    }
}
"""

obj = json.loads(jsonstr)
print(obj)

from cmdb.types import get_instance

Int = get_instance(obj.get('type'), **obj.get('option'))
print(Int.stringify(obj.get('value')))
```

练习
实现IP的限制。例如，要求IPv4地址必须以192开头prefix。

```
class IP(BaseType):
    '''实现IP数据校验和转换'''

    def stringify(self, value):
        """转换，错误的数据不要给默认值，就抛异常让外部捕获"""
        prefix = self.prefix
        if prefix and not str(value).startswith(prefix):
            raise ValueError('Must start with {}'.format(prefix))
        return str(ipaddress.ip_address(value))

    def destringify(self, value):
        return value

# 使用
meta = {
    "type":"cmdb.types.IP",
    "value":"192.168.0.1",
    "option":{
        "prefix":"192.1"
    }
}

IP = get_instance(meta.get('type'), **meta.get('option'))
print(IP.stringify('192.168.0.1'))
```

## 拆分get_instance

get_instance 这个函数分为2部分：
1、加载初始化类
2、创建初始化
按照上面的功能拆分。

```
def get_class(type:str):
    m, c = type.rsplit('.', maxsplit=1)
    print(m, c)
    mod = importlib.import_module(m)
    cls = getattr(mod, c)

    if issubclass(cls, BaseType):
        return cls
    raise TypeError('Wrong Type {}. Not subclass of BaseType.'.format(type))

def get_instance(type: str, **option):
    obj = get_class(type)(**option) # 从这里导入选项
    return obj
```

## 缓存

为了使用一个实例，每一次都需要获取类创建实例。能够缓存？
减少每次都需要重新创建实例的过程，采用懒加载思想，第一次用才创建。

`classes_cache = {} # 类缓存`
缓存什么？key是什么？
动态加载模块、类的函数get_class，加入类缓存是为了减少类加载过程吗，避免重复加载吗？
cmdb.types.IP作为key，value是类对象。

```
classes_cache = {} # 类缓存

def get_class(type:str):
    # 使用缓存
    cls = classes_cache.get(type)
    if cls:
        return cls

    m, c = type.rsplit('.', maxsplit=1)
    print(m, c)
    mod = importlib.import_module(m)
    cls = getattr(mod, c)

    if issubclass(cls, BaseType):
        classes_cache[type] = cls
        return cls
    raise TypeError('Wrong Type {}. Not subclass of BaseType.'.format(type))
```

其实这个类缓存是为了加快获取类对象的速度，原来是导入模块后搜索类对象，变成了直接到classes_cache字典中使用key搜索。

`instances_cache = {} # 实例缓存`
key是什么？cmdb.types.IP 行吗？
假设有如下的meta描述

```
{
    "type":"cmdb.types.IP",
    "value":"192.168.0.1",
    "option":{
        "prefix":"192.168"
    }
}

{
    "type":"cmdb.types.IP",
    "value":"172.16.10.1",
    "option":{
        "prefix":"176.16"
    }
}
```

它们是否使用缓存的同一个对象？
差异就在stringify方法中，option中prefix不一样。所以，相同option的实例创建一个就够了，但是option是一个字典，如何解决？

```
instances_cache = {} # 实例缓存

def get_instance(type: str, **option):
    key = ",".join("{}={}".format(k,v) for k,v in sorted(option.items()))
    key = "{}|{}".format(type, key)

    instance = instances_cache.get(key)
    if instance:
        return instance

    obj = get_class(type)(**option) # 从这里导入选项
    instances_cache[key] = obj
    return obj
```

## 名称简化

`cmdb.types.IP` 这个名字太长不好记忆，对于用户来说，只需要记住 `IP` 并填写就行了。
这里做个约定，`cmdb.types` 下的类名用户可以直接作为type的名称。
也就是说，使用IP这个短名称，也要对应到IP这个类。这还需要建立字典吗？
不需要。因为模块一旦加载创建后，这些类都会放在模块的全局字典中。
为了方便，把这个短名称即类名注入到classes_cache这个字典中。
长名称也可以同时注入进来。

```
def inject_classes_cache():
    mod = globals().get('__package__')
    for k,v in globals().items():
        if type(v) == type and k != 'BaseType' and issubclass(v, BaseType):
            classes_cache[k] = v
            classes_cache[".".join((mod, k))] = v

inject_classes_cache() # 函数调用放在模块后面
```

```
# 模块加载后，classes_cache中的内容如下
`{'cmdb.types.Int': <class 'cmdb.types.Int'>, 'Int': <class 'cmdb.types.Int'>, 'IP': <class
'cmdb.types.IP'>, 'cmdb.types.IP': <class 'cmdb.types.IP'>}`
```

那么，只要模块加载了，classes_cache就有了长名称、短名称对应的类对象的映射。
如果送入一个类型 `cmdb.types.IP` ，直接查字典就可以了，不需要动态加载模块了。因此，简化get_class函数如下

```
def get_class(type:str):
    # 使用缓存
    cls = classes_cache.get(type)
    if cls:
        return cls
    raise TypeError('Wrong Type {}. Not subclass of BaseType.'.format(type))
```

## 单值约束

目前，可以认为value里面存放的是 **一个** 值。
对这种单一值，可以做一些约束，例如nullable、unique。

nullable选项
值是否可以为空。
nullable选项，如果设置就应用到值的验证上。
如果设置为false，则值不可以为空，如果为空抛异常；如果为true，值可以为空，就直接过。

unique选项
值是否唯一。
unique选项，如果设置就应用到值的验证上。
如果设置为false，则不检查值的唯一性；如果设置为true，则需要检查值的唯一性。
怎么判断唯一？
schema + filed表构成的虚拟表，entity表使用同一个schema_id就是同一张表的数据。
依照host举例，找ip是否重复，schema_id=1，ip对应field_id=2，在value表中管理schema表查找所有entity_id对应的schema_id=1的所有field_id=2的value字段的数据，且value='192.168.0.10'的数据的count是否大于1，大于1说明现有的数据重复了，等于1说明里面有一条了，等于0说明数据库中还没有。
假设准备插入一个IP数据进去，如果count > 0就不可以插入，说明已经有同样的IP被使用过了。

meta的json描述

```
{
    "type":"cmdb.types.IP",
    "value":"192.168.0.1",
    "nullable":false,
    "unique":false,
    "option":{
        "prefix":"192.168"
    }
}
```

## 多值约束

需求
一个主机名对应多个IP地址，如何描述？

meta的json描述

```
{
    "type":"cmdb.types.IP",
    "value":"192.168.0.1,192.168.0.2",
    "nullable":false,
    "unique":false,
    "option":{
        "prefix":"192.168"
    },
    "multi":true
}
```

某一个主机如果允许绑定多个ip地址，有2中存储方法：
1、在value表的value字段上存储'192.168.0.1,192.168.0.2'
好处是实现简单，但是，提取修改不方便。
IP存储顺序发生变化，就成了不同的IP值了，无法判断是否重复，
2、使用多条记录存储
在value表中使用多条存储，但是有唯一键约束`UNIQUE KEY` index4 `(` entity_id `,` field_id `)`。
2个ip存储就会有2个一样的entity_id和field_id，会违反唯一键约束的。需要移除这个唯一键约束。
移除语句

```
ALTER TABLE `value` DROP INDEX `index4`;
```

好处是字段容易控制，而且还可以使用unique约束来约束某一个IP在虚拟表中只能出现一次。

采用第二种方式。注意需要移除唯一键约束。

## 多值约束设计

**multi=false（单值，默认）**
提交一个值，值存在就更新；不存在就新增。
举例：一个主机只能记录一个IP，那么提交上来数据，不存在就增加，存在数据就覆盖。

**multi=true（多值）**
多值情况较为复杂，需要分析。
1、假设该主机没有记录一个IP，现在提交了多条
2、假设该主机已经有IP记录，又有IP提交，又可以分3种情况
2.1、提交的IP都是新的IP
2.2、提交的IP部分是新的IP
2.3、提交的IP全是已经存在的IP
怎么办？
我的思路是，思考提交的IP代表什么？是原有IP的修改，还是用户提交的新IP？
想象用户看到的界面，上面有好多IP，用户通过修改IP列表，然后提交眼前看到的IP列表，也就是说用户提交的才是最新的IP列表，原有的列表它不关心了。
所以，不管数据库中当前IP列表是什么，不关心，最终只保存用户提交的就行了。
当然，这里面也许有的IP当前就在数据库中存着，可以删除，但是删除代价大于更新，所以尽可能的更新。
但是更新，又需要判断哪些IP是否在，哪些IP不在，麻烦。
因为，相对来说，用户修改IP这种事做的少，所以可以有下面的设计。

> 设计如下：
> 将这个主机的所有IP查回来，条目数为c1。新提交的IP数目为c2。
> 如果`c1 == c2`，循环迭代，用新IP替换所有库中IP；
> 如果`c1 < c2`，替换c1个数据，剩下c2-c1个新增。
> 如果`c1 > c2`，替换c2个数据，剩下c1-c2个删除。

---

# 多表关系设计

前面讨论的都是单表之间的关系，如何在这种设计上实现多表关联？
提供reference来表示，它要说明它引用了哪一张虚拟表的哪一个字段，即schema_id等于几同时field_id等于几。

meta的json描述例子

```
{
    "type":"cmdb.types.IP",
    "value":"192.168.0.1,192.168.0.2",
    "nullable":false,
    "unique":false,
    "option":{
        "prefix":"192.168"
    },
    "multi":true,
    "reference":{
        "schema":1,
        "field":1
    }
}
```



上面的意思是这个字段引用schema_id等于1对应的虚拟表的field_id为1的字段。
例如这个字段是host表的IP字段，它引用了ipaddress表的IP字段。
host表称为Source表，ipaddress表称为Target表。

**类型一致校验**
增加了引用后，需要先查询看看schema_id为1的表且field_id为1的字段的meta字段里面的type定义是否和当前字段定义的一致？如果一致，才能继续。

**约束设计**
Source表
1、新增数据，首先做类型校验，例如是不是int类型，是否在取值范围内，可否为空，是否唯一，这些检验做完，再做外键约束。去查Target表，看看被引用字段中是否存在当前Source表的字段值，如果有，数据可以插入，不存在就抛异常。
2、修改数据，同上，就相当于加入新的值。
3、删除数据，直接删除。
4、查询数据，直接查即可。

Target表
1、查询数据，直接查即可。
2、增加数据，新增的数据还没有被引用，直接插入即可。
3、删除数据，可能已经被引用了，所以需要有删除策略。
3.1、级联删除cascade

先查询Host表中使用这个字段值对应的记录并删除，然后再去删除ipaddress表的记录。
举例：
host.ip和switch.ip都引用了ipaddress.ip。
ipaddress.ip要删除192.168.1.10，那么就要去value表先删除host表、switch表的引用。
3.2、置空set_null
如果Target表的主键字段的值删除，那么引用这个值的外键字段都要置null。
举例：
host.ip和switch.ip都引用了ipaddress.ip。
ipaddress.ip要删除192.168.1.10，那么就要去value表先置空host表、switch表的引用。
3.3、禁用disable
被引用了就不允许删除。
host.ip和switch.ip都引用了ipaddress.ip。
ipaddress.ip要删除192.168.1.10，不允许。
4、修改数据
4.1、级联更新cascade
和删除类似，不过这里是把source表更新成新值。
Target表更新这个新值要通过检验，Source表也要通过自己字段的检验，否则抛异常。
4.2、disable禁用
如果被引用，就更新失败。
注意这里不用设计set_null，因为Target表字段更新值，Source表的字段应该和其一直，而不是设置为null。

Source表meta的json描述

```
{
    "type":"cmdb.types.IP",
    "value":"192.168.0.1,192.168.0.2",
    "nullable":false,
    "unique":false,
    "option":{
        "prefix":"192.168"
    },
    "multi":true,
    "reference":{
        "schema":1,
        "field":1,
        "on_delete":"cascade|set_null|disable",
        "on_update":"cascade|disable"
    }
}
```

以上设计，都可以转换为当前MySQL数据库表的各种SQL语句。

多表关联的外键约束设计，非常复杂，代码实现成本非常高，非常难控制。
所以这种设计要少用外键约束，约束还是要用的，但是建议约束在业务层实现，不要放到数据库中实现。

思考
如果设计了这种外键约束，Target表主键变动了，首先不知道谁引用了，需要自己代码实现，去遍历field表所有记录，从meta字段中解析谁引用了这个字段，非常没有效率。
有没有办法提高效率？
可以不可以不遍历？

不遍历是可以的，有2种方法：
1、在Target表的meta字段中记录，谁引用了它。
2、在field表中，增加一个字段reference，如果是Source表，它引用Target表的字段，则这个reference字段一定有值。
使用SQL语句

```
select * from field where reference = 2;
```

这个2指的是field表的主键值，也就是说2对应一条唯一的记录，这个记录指的是schema_id=1的虚拟表的字段ip的信息。
用这种方法，就可以是Target表快速找到引用自己的Source表的字段。

上面2种方法都采用了冗余设计的方式，但第二种更佳。

---

实验语句

```
INSERT INTO `schema` (name) VALUES('ippool');
SELECT * FROM `schema` ORDER BY id desc;
INSERT INTO field (name, schema_id) VALUES ('ip', 2);
SELECT `schema`.id,`schema`.`name`,field.id AS fid,field.`name` AS fname,field.meta FROM
`schema` INNER JOIN field ON field.schema_id = `schema`.id

-- 表2 ippool
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` (`value`, entity_id, field_id) VALUES('192.168.1.10', 3, 3);
INSERT INTO `value` (`value`, entity_id, field_id) VALUES('192.168.1.20', 4, 3);
INSERT INTO `value` (`value`, entity_id, field_id) VALUES('192.168.1.30', 5, 3);


-- 表1 host
INSERT INTO entity (`key`, schema_id) VALUES ('587723df88a54b2e9f449888d75f50de', 1);
INSERT INTO `value` (`value`, entity_id, field_id) VALUES('DNS Server', 6, 1);
INSERT INTO `value` (`value`, entity_id, field_id) VALUES('172.16.100.1', 6, 2);
```