实验三：数据库系统开发

一、实验目的及要求

(一) 实验目的

在熟练掌握MySQL基本命令、SQL语言以及用C语言编写MySQL操作程序的基础上，学习简单数据库系统的设计方法，包括数据库概要设计、逻辑设计。

(二) 实验要求

1. 该系统的E-R图至少包括8个实体和7个联系（必须有一对一联系、一对多联系、多对多联系）。
2. 在设计的关系中需要体现关系完整性约束：主键约束、外键约束，空值约束。
3. 对几个常用的查询创建视图、并且在数据库中为常用的属性（非主键）建立索引。
4. 该系统功能必须包括：插入、删除、连接查询、嵌套查询、分组查询。其中插入，删除操作需体现关系表的完整性约束，例如插入空值、重复值时需给予提示或警告等。
5. 加分项：界面友好、包含事务管理、触发器等功能。

二、实验环境

- 操作系统：windows 10
- 数据库：MySql 5.5.29
- 高级语言：python 3.5
- 网站设计：django-1.8

三、实验内容

这一部分的目录如下：

(一) 用户需求

这是一个关于音乐的数据库[1]。有以下需求

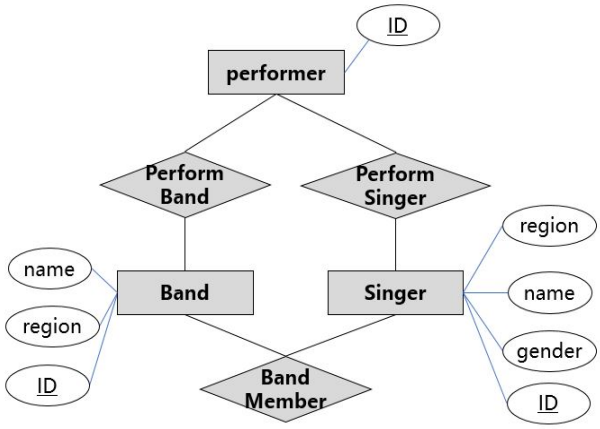
> (1) 歌曲的属性有：名称、发布日期、时长、风格、演唱者、作词人、作曲人 、所属的专辑。
> (2) 演唱者可以是歌手，也可以是组合。组合中的成员由歌手组成。歌手的属性有：名字、性别、地区和所属的唱片公司。组合的属性有：名字、地区和所属的唱片公司。
> (3) 专辑的属性有：名字、发布日期和主打歌。
> (4) 公司的属性有：名字、建立时间和地区。
> (5) 作词人和作曲人的属性有：姓名、性别。

(二) 概念设计

1.演唱者

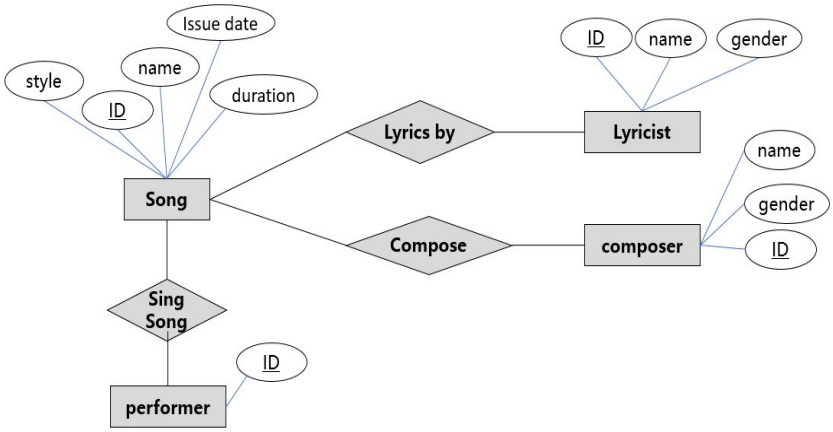
上述数据库系统设计的主要困难在于，如何去处理*演唱者*。

为了解决这个问题，在已有的两个实体——歌手(Singer)和组合(Band)的基础上，增加了演唱者(Performer)这个实体。然后通过两两之间建立联系集，维持它们关系。如下图



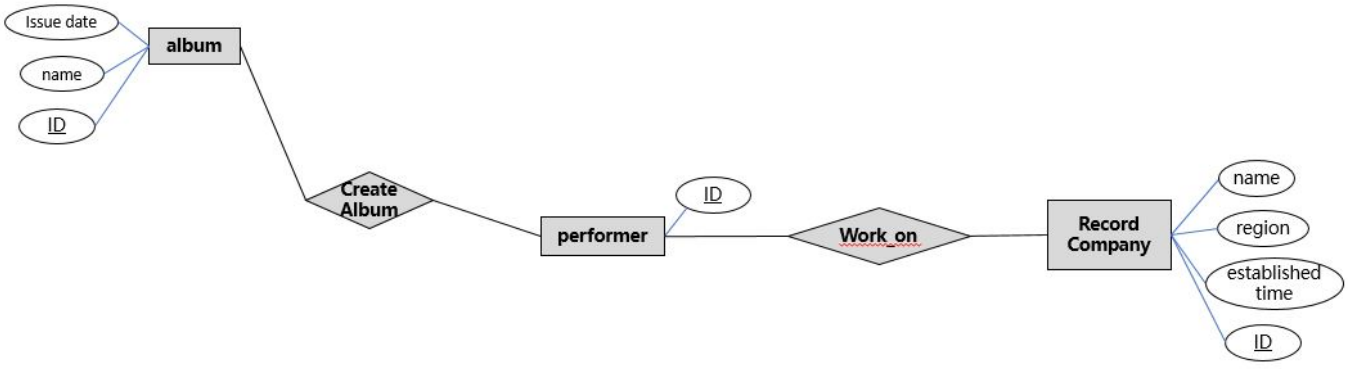
2.歌曲与演唱者、作曲人及作词人

如下图，歌曲(Song)与演唱者(Performer)、作曲人(Composer)及作词人(Lyricist)分别建立联系集：SingSong、LyricsBy、Compose。



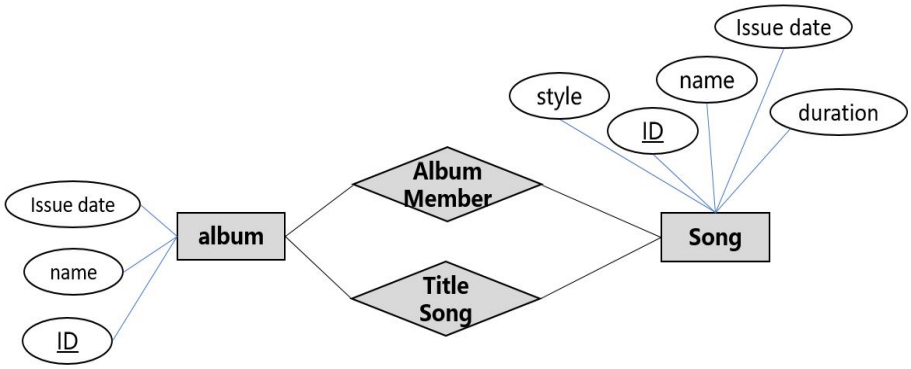
3.演唱者与唱片公司及专辑

如下图，演唱者(Performer)与唱片公司(RecordCompany)和专辑(Album)分别建立联系集：CreateAlbum、WorkOn。

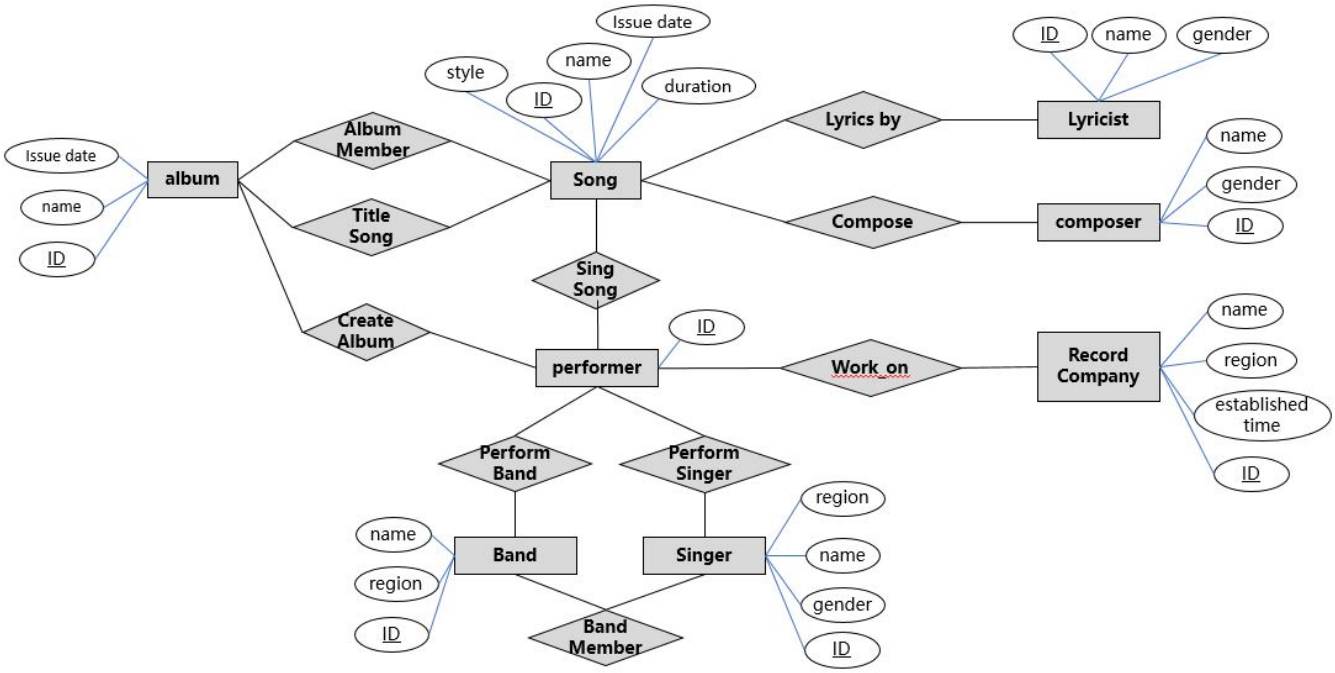


4.歌曲与专辑

如下图，歌曲与专辑建立两个联系集，分别为：AlbumMember、TitleSong，以表示“歌曲属于专辑”和“一个歌曲是专辑的主打歌”这两个联系。



5.总体ER图



6.功能需求规格说明

查询数据

> (1) 根据名称查询歌曲、演唱者、专辑、唱片公司；
> (2) 查看歌曲的演唱者、作词人和作曲人；
> (3) 查看演唱者所属的唱片公司、演唱的所有歌曲和发布的专辑；
> (4) 查看专辑中的所有的音乐和主打歌；
> (5) 查看一个唱片公司中所有的组合和歌手；
> (6) 根据风格查询歌曲。

插入数据

> (1) 插入歌曲、歌手或组合、专辑、唱片公司、作词人、作曲人；
> (2) 插入歌曲时，选择演唱者，选择专辑，选择作词人和作曲人；
> (3) 插入歌手或组合时，选择公司；
> (4) 插入专辑时，选择演唱者。

删除数据

> (1) 删除歌曲、歌手或组合、专辑、唱片公司、作词人、作曲人。

更改数据

> (1) 更改以下实体集的属性：歌曲、歌手或组合、专辑、唱片公司、作词人、作曲人；

(2) 更改歌曲的演唱者、作词人和作曲人及所属专辑，或改为空；
(3) 更改歌手或组合所属的公司，或改为空；
(4) 更改专辑的演唱者，或改为空。

(三) 逻辑设计

1.模式的合并

上面ER图中，共包括8个实体和10个联系。

(1) 一对一联系有：TitleSong, PerformBand, PerformSinger。
(2) 一对多联系有：AlbumMember, CreateAlbum, WorkOn, LyricsBy, Compose。
(3) 多对多联系有：BandMember，SingSong。

需要说明的是

(1) 一个歌手可能存在多个组合中。这可能是因为一些歌手所在组合解散之后，他们加入了另一个组合，或者是有一些组合为非商业性的，允许其成员加入另一个组合。因此，BandMember为多对多联系。
(2) 一首歌曲，可能由两人合唱，但是这两人由不构成一个组合。因此，SingSong为多对多联系。

根据下面的合并规则

(1) 一对多的联系，联系集可以合并到“一”对应的那个实体集的关系模式中；
(2) 一对一的联系，联系集可以合并到的任何一个实体集的关系模式中。

因此

(1) 将TitleSong合并到Album中；
(2) 将PerformBand合并到Band中；
(3) 将PerformSinger合并到Singer中；
(4) 将AlbumMember合并到Song中；
(5) 将CreateAlbum合并到Album中；
(6) 将WorkOn合并到Performer中；
(7) 将LyricsBy合并到Song；
(8) 将Compose合并到Song。

2.将ER图转化为关系模式

实体集

- Song(**id**, **name**, style, issue_date, duration, *lyricist_id*, *composer_id*, *album_id*)
- Album(**id**, **name**, issue_date, *title_song_id*, *performer_id*)
- Performer(**id**, *company_id*)
- Band(**id**, **name**, region, *performer_id*)
- Singer(**id**, **name**, gender, region, *performer_id*)
- Company(**id**, **name**, region, established_time)
- Lyricist(**id**, **name**, gender)
- Composer(**id**,**name**, gender)

联系集

- SingSong(**song_id**, **performer_id**)
- BandMember(**band_id**, **singer_id**)

注：上述关系模式中，带下划线的属性为主键，加粗的为非空属性，斜体的为外键。

3.视图

> **视图关系**可以定义为包含查询结果的关系。视图是有用的，它可以隐藏不需要的属性，可以把信息从多个关系收集到一个单一的视图中。

下面定义了一些视图，在“具体实现”这一节中，我们将对它们进行具体的实现

> - **PerformerName**：查询演唱者的ID和名字（歌手或者组合的名字）。
> - **SongOfPerformer**：查询演唱者演唱的所有歌曲。
> - **AlbumOfPerformer**：查询演唱者创作的所有专辑。
> - **SingerInfor**：查询歌手的详细信息，包括所属的唱片公司。
> - **BandInfor**：查询组合的详细信息，包括所属的唱片公司。
> - **SingerOfBand**：查询组合中所有的歌手。
> - **SongInfor**：查询歌曲的详细信息，包括作词人和作曲人的名称以及所属专辑的名称。
> - **SongOfAlbum**：查询专辑中所有的歌曲。
> - **AlbumInfor**：查询专辑的详细信息，包括创作专辑的表演者和主打歌。
> - **PerformerOfCompany**：查询公司和旗下的所有表演者。

4.索引

创建索引的目的

> **索引**(index)是一种数据结构，它允许数据库系统高级地找到关系中那些在索引属性上给出定值的元祖，而不用扫描关系中的所有元祖。

因此，在下面的关系模式的属性上建立索引

> - Song.name
> - Album.name
> - Singer.name
> - Band.name

(四) 触发器及事务管理

1.触发器

根据以下定义

> **触发器**(trigger)是一条语句，当对数据库作修改时，它被自动执行。要设置触发器机制，必须满足两个要求：
>
> - 指明什么条件下执行触发器。
> - 指明触发器执行时的动作。
>
> 触发器有很多适合的用途，但是在如下场合下，不应该使用触发器。
>
> - 使用触发器是替代级联特性来实现外键约束的on delete cascade特性。因为这样不仅需要完成大量的工作量，而且会使数据库中的约束集合难以理解。
> - 使用触发器来维护视图。
> - 使用触发器来复制和备份数据库。
>
> 此外，使用触发器时应特别小心，因为运行期间一个触发器可以引发另一个触发器。在最坏的情况下，一个触发器错误会导致一个无限的触发链。

定义如下触发器

> - InsertPerformer
>   - 触发条件：在插入一个歌手或者一个组合之后。

> - 动作：在Perfromer表中插入一条数据，然后将相应的Performer.id作为这个歌手或者组合属性performer_id的值。

2.事务管理

事务的定义如下[2]

> 事务(transaction)是访问并可能更新各种数据项的一个程序执行单元。
>
> 事务具有ACID特性：原子性、一致性、隔离性、持久性。
>
> - 原子性保证事务的所有映像在数据库中要么全部反映出来，要么根本不反映；一个故障不能让数据库处于事务部分执行后的状态。
> - 一致性保证若数据库一开始是一致的，则事务（单独）执行后数据库仍处于一致状态。
> - 隔离性保证并发执行的事务是相互隔离，使得每个事务感觉不到系统中其他的事务的并发执行。
> - 持久性保证一旦一个事务提交后，它对数据库的改变不会丢失，既是系统可能出现故障。

事务管理的目的就是，维护事物的ACID特性，分为并发控制和恢复系统。虽然这些显得十分复杂，但在界面设计那一节中，我们将看到如何用简单的django语句实现简单的事务管理。

(五) 具体实现(MySql)

1.建表

以下为创建MySQL数据表的SQL通用语法：

```
CREATE TABLE table_name (column_name column_type);
```

在上一节中，一共定义了10张数据表。在附录2中给出了创建它们的sql语句。

2.添加视图

以下为创建视图的SQL通用语法：

```
CREATE VIEW view_name AS + [查询语句];
```

在上一节中，一共定义了10个视图。在附录3中给出了创建它们的sql语句。

3.添加触发器

以下为创建触发器的通用语法：

```
CREATE TRIGGER trigger_name
AFTER/BEFORE
INSERT/UPDATE/DELETE ON table_name
FOR EACH ROW
BEGIN
    sql语句
END;
```

因此，使用如下sql语句创建两个触发器

```
create trigger InsertPerformer1
after
insert on Singer
for each row
begin
    insert into Performer values();
    set new.performer_id = (select max(id) from  Performer);
end;

create trigger InsertPerformer2
```

```
before
insert on Band
for each row
begin
    insert into Performer values();
    set new.performer_id = (select max(id) from  Performer);
end;
```

4.添加索引

创建索引的语法相对来说比较简单，如下

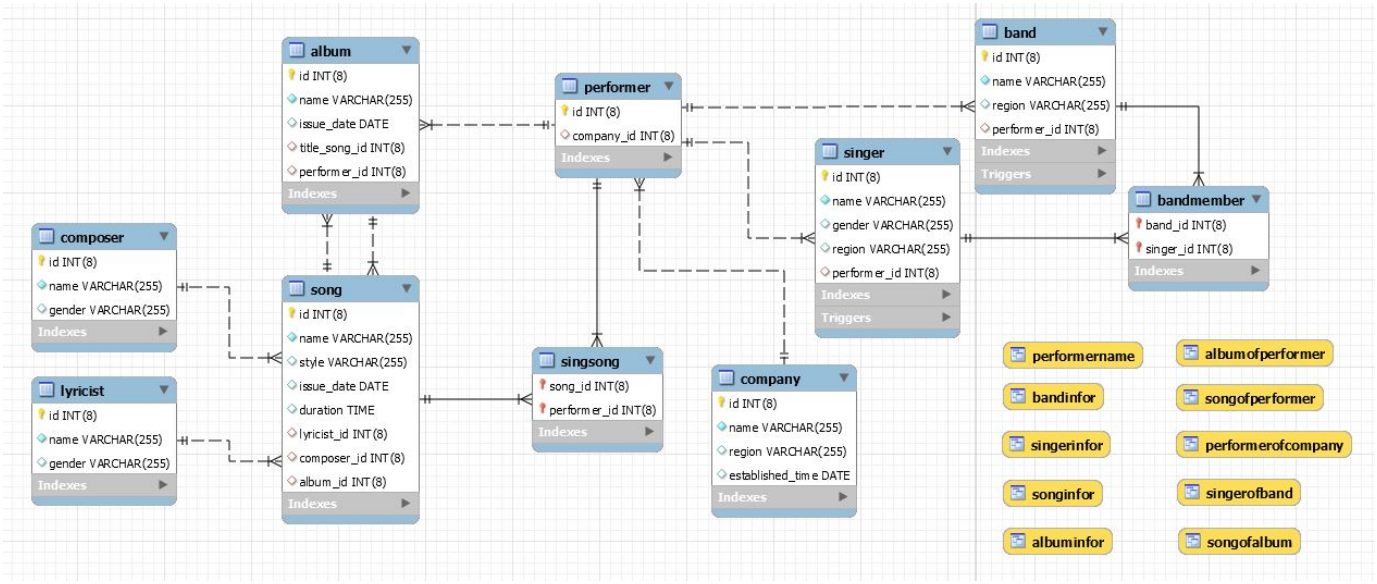
```
CREATE INDEX index_name on table_name(column_name);
```

因此，使用如下语句创建4个索引

```
create index Song_name_index on Song(name);
create index Album_name_index on Album(name);
create index Singer_name_index on Singer(name);
create index Band_name_index on Band(name);
```

5.导出EER图

下面的的EER图(Enhanced Entity-Relationship Model)，是使用MySQL Workbench导出的



(六) 界面设计(django)

需要说明的是，这一节的大部分内容已经超出了实验的要求，但是为了以后参考的方便，在这里总结以下django搭建网站的主要步骤。因此，这部分更像一个关于django教程。本次实验中所有的代码都可以在https://github.com/1140310118/music_pool处获取。以下大部分的内容，参考自http://www.kancloud.cn/wizardforcel/django-chinese-docs-18/。

需要再次强调的是，本次实验使用的django版本是1.8。这是因为，不同版本的django之间的存在一定的差异。

这一节的目录如下

> 7. 使用表单
> 8. 简单的事务管理

1. 安装django

在命令行中使用如下命令安装django

```
pip install django==1.8
```

安装结束，在命令行中输入python。然后尝试导入django

```
>>> import django
>>> django.get_version()
'1.8'
```

如果你使用的是Python3，那么安装后可能存在关于数据库的问题。

> django 使用的MySQLdb连接数据库，但是MySQLdb只支持Python2.，还不支持python3。

因此，我们使用PyMySQL代替MySQLdb。具体来说，在下面的文件中

```
[Python的路径]\Lib\site-packages\django\db\backends\mysql\__init__.py
```

添加如下语句

```
import pymysql
pymysql.install_as_MySQLdb()
```

2. 创建项目mysite

在命令行中，使用cd命令进入存储你想存储代码的目录，然后运行以下命令

```
django-admin.py startproject mysite
```

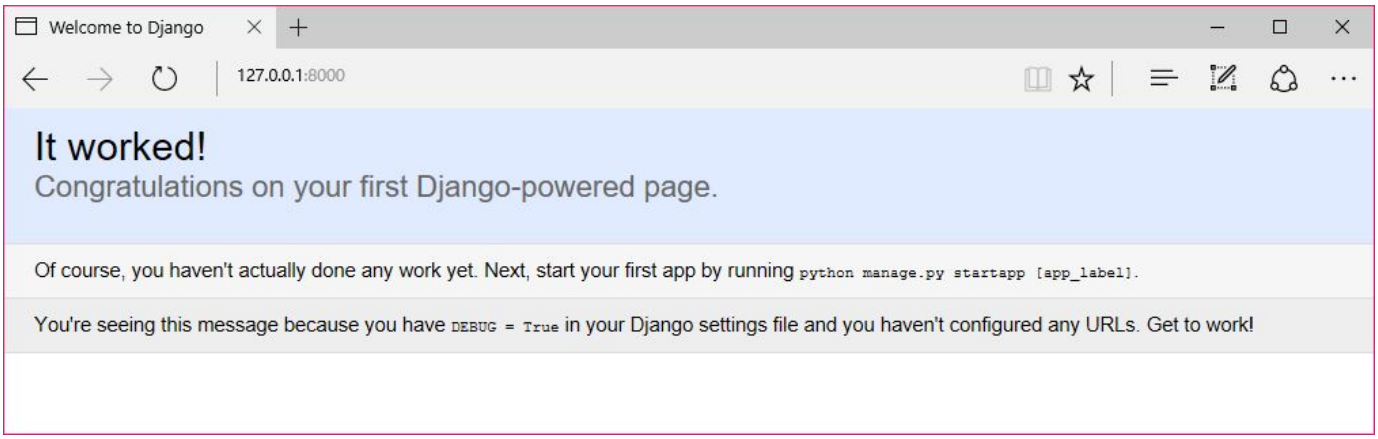
于是，当前目录下建立了一个mysite目录，结构如下

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

从外层的mysite目录切换进去，通过如下命令，我们就能运行服务器

```
python manage.py runserver
```

在http://127.0.0.1:8000/，我们将看到如下页面



3. 编辑setting.py

在mysite/settings.py中保存Django项目的配置。现在通过编辑它，修改项目的配置。

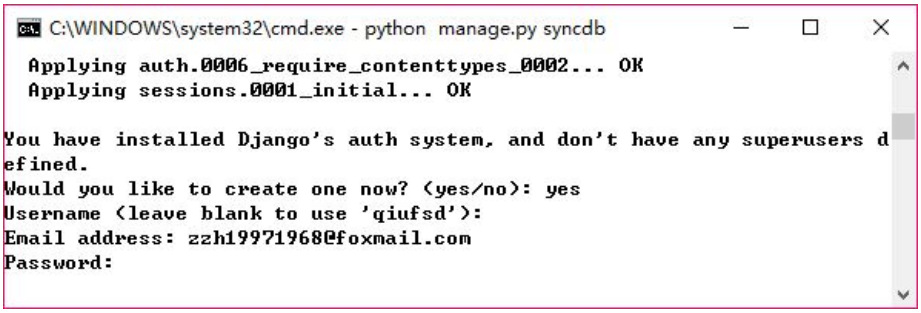
修改数据库设置(DATABASES)为

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'music_pool',
        'USER': 'root',
        'HOST': '',
        'PORT': '3306',
        'PASSWORD':[你的密码],
    }
}
```

修改完成之后，运行如下命令，进行数据库的同步

```
python manage.py syncdb
```

运行此命令后，命令行会提示是否创建管理员账户，选择是，因为这将会非常有用。



设置语言版本

```
LANGUAGE_CODE = 'zh-cn'
```

> 注：在1.9及以后的版本中，需要使用'zh-hans'。

更改时区(TIME_ZONE)

```
TIME_ZONE = 'Asia/Shanghai'
```

4. 创建应用music_pool

使用如下命令，创建应用

```
python manage.py startapp music_pool
```

这将创建一个music_pool目录，如下

```
music_pool/
    __init__.py
    models.py
    tests.py
    views.py
```

为了激活这个应用，我们编辑setting.py文件，在INSTALLED_APPS设置中加入'music_pool'，结果如下

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'music_pool',
)
```

5. 将数据库导出到 models.py 中

在music_pool/models.py中，我们可以创建按照如下的方式，创建模型

```
from django.db import models

class Company(models.Model):
    name = models.CharField(max_length=255)
    region = models.CharField(max_length=255)
    established_time = models.DateTimeField('date established')

class Performer(models.Model):
    company = models.ForeignKey(Company)

#...
```

然后在命令行中使用如下命令，在数据库中创建相应的表

```
python manage.py syncdb
```

但是，由于我们已经在数据库中创建了表，因此采取导入MySQL数据库的办法。在命令行中，使用如下语句

```
python manage.py inspectdb >music_pool/models.py
```

这样，MySQL数据库中的表就导出到models.py中。比如在models.py建立了一个Album对象与数据库中的Album表相对应

```
class Album(models.Model):
    name = models.CharField(max_length=255)
    issue_date = models.DateField(blank=True, null=True)
    title_song = models.ForeignKey('Song', blank=True, null=True)
    performer = models.ForeignKey('Performer', blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'album'
```

`manage.py = False` 表明告诉Django不要管理此表的创建、修改和删除。为了预防一些潜在的错误，建议注释掉这一行。这里需要注意的是

> 当Django根据models.py中的模型创建数据库中的表时，会在未指明主键的情况下为每个模型分配

> 一个id作为主键；每个模型中的外键属性，在表中的名字会被附上"_id"。
>
> 因此，相应地，从数据库中导数的models中的对象
> 1. 不会有id属性；
> 2. 外键属性中的"_id"也会被去掉。

因此，导出之后会存在一些错误。如在Bandmember中

```
unique_together = (('band_id', 'singer_id'),)
```

我们改成

```
unique_together = (('band', 'singer'),)
```

这里的unique_together是为了解决django不支持多字段主键的问题。

此外，在启动服务器还出现如下错误

```
 HINT: Rename field 'Song.album', or add/change a related_name argument to the definition for field 'Album.tit
```

因此，类Album进行相应的修改

```
title_song = models.ForeignKey('Song', blank=True, null=True, related_name='title_song')
```

修改完毕之后，再次运行 `python manage.py syncdb` 以同步据库。

6. 启用网站管理

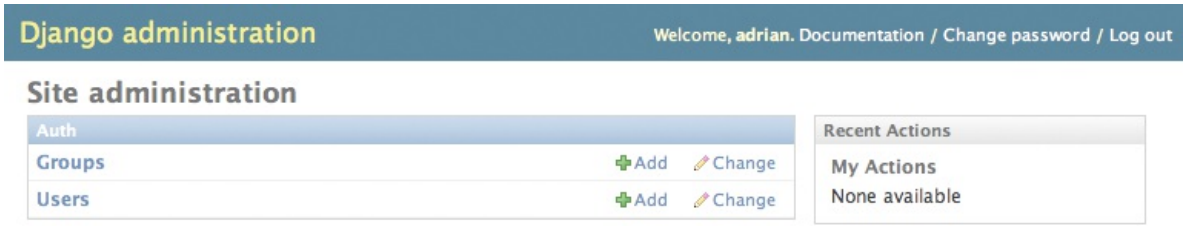
> Django 是在新闻编辑室环境下编写的，“内容发表者”和“公共”网站之间有 非常明显的界线。网站管理员使用这个系统来添加新闻、事件、体育成绩等等， 而这些内容会在公共网站上显示出来。Django 解决了为网站管理员创建统一 的管理界面用以编辑内容的问题。
>
> 管理界面不是让网站访问者使用的。它是为网站管理员准备的。

使用如下命令，启动服务器

```
python manage.py runserver
```

使用浏览器，访问http://127.0.0.1:8000/admin/。我们将看到如下的登录界面

输入在前面创建的用户名和密码，登录进去。我们将看到



这些是一些可编辑的内容，包括group，users 和 sites。这些都是django默认情况下自带的核心功能。

下面，我们 Song 对象添加到这个页面。首先找到music_pool/admin.py，如果没有，那就创建一个。

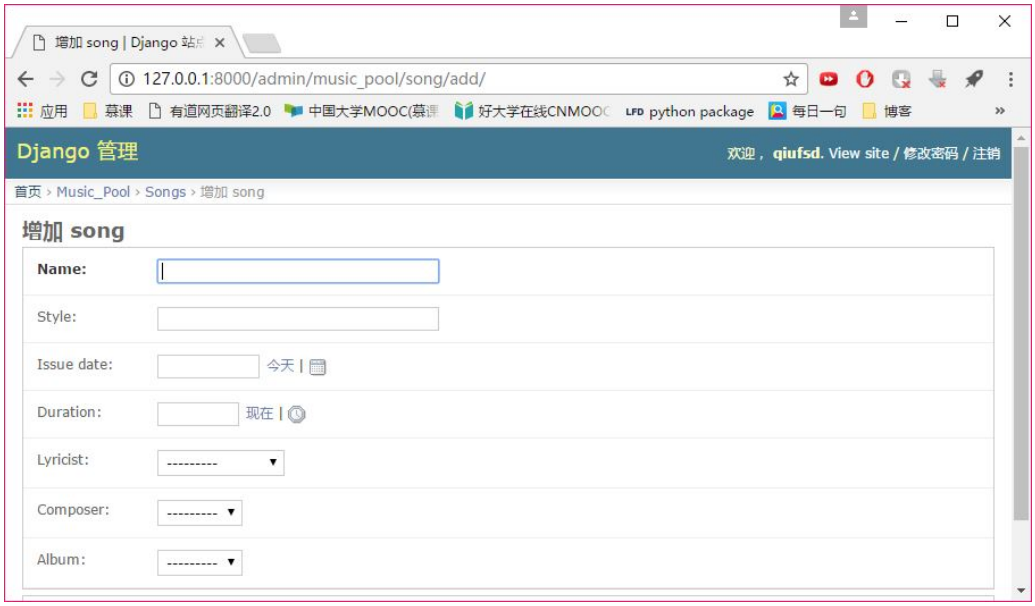
```
music_pool/
    __init__.py
    models.py
    tests.py
    views.py
    admin.py
```

在admin.py文件中添加如下内容：

```
from django.contrib import admin
from music_pool.models import Song
admin.site.register(Song)
```

重启服务器，登录管理员账户，我们将看到如下页面

然后，我们就可以对Song对象进行编辑了！比如添加一条记录。



7. 创建页面

> 在这一步将创建一个页面，用来显示所有的歌曲及其作词人、作曲人和所属专辑。

在music_pool目录下创建一个名为url.py的文件。

```
music_pool/
    __init__.py
```

```
    admin.py
    models.py
    tests.py
    urls.py
    views.py
```

在music_pool/url.py文件中添加以下内容

```
from django.conf.urls import patterns, url
from music_pool import views

urlpatterns = patterns('',
    url(r'^$', views.all_song, name='songs')
)
```

接下来，在mysite/urls.py中添加一个include()方法，于是这个文件内容变成下面的样子

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^music_pool/', include('music_pool.urls')),
    url(r'^admin/', include(admin.site.urls)),
)
```

URL部分的任务完成了，接下来，我们需要编写URL对应的views.all_song这个方法了。

在music_pool/views.py文件中添加如下语句

```
from django.shortcuts import render
from music.models import Song

def all_song(request):
    song_list=Song.get_all()
    context = {'song_list': song_list}
    return render(request, 'music_pool/song_list.html', context)
```

在函数views.all_song，我们首先通过models.Song查询得到了所有的歌曲，然后将参数传递给模板music/song_list.html。

首先，我们来实现Song.get_all()这个函数，编辑music_pool/models.py，在Song这个类下添加如下方法

```
@classmethod
def get_all(self):

    sql=""" select name, style, issue_date, duration, lyricist_name, composer_name, album_name
            from `SongInfor`
        """
    song_list=Song.objects.raw(sql)
    return song_list
```

接下来，我们添加模板music_pool/song_list.html。在music_pool下创建一个templates目录，在这个目录下创建一个music_pool目录，并在其中创建一个song_list.html文件。于是，我们的模板文件的路径应该是mysite/music_pool/templates/music_pool/song_list.html。

在song_list.html添加如下内容

```
{% if song_list %}
    <ul>
    {% for song in song_list %}
        {{song.name}} 
        {{song.style}} 
```

```
        {{song.issue_date}} 
        {{song.duration}} 
        {{song.lyricist_name}} 
        {{song.composer_name}} 
        {{song.album_name}}
    {% endfor %}
    </ul>
{% else %}
    <p>No Song are available.</p>
{% endif %}
```

8. 使用表单

略

9. 简单的事务管理

下面实现了事务的原子性。

```
from django.db import transaction

@transaction.atomic
def view_func(request):
    try:
        with transaction.atomic():
            # ...
    except IntegrityError:
        # ...
```

在比1.8更高的版本中，提供了更加高级的事务管理接口。

四、实验体会

1. 本次实验的主要任务是数据库设计，过分地将精力放在django上，实在不好，不好。

附录1 数据库设计阶段

1. 刻画用户需求
2. 概念设计
   - 构建ER图：实体、实体的属性、实体之间的联系、实体和联系上的约束
   - 明确功能需求：描述在数据上进行的各类操作
3. 逻辑设计
   - 将高层概念模式映射到将使用的数据库系统的实现数据模型上，通常是将ER模型映射到关系模式。
4. 物理设计
   - 指明数据库的物理特征，包括文件组织格式和索引结构的选择。

附录2 MySQL-创建数据表

```
Create table Song(`id` int(8) not null auto_increment,
                    name varchar(255) not null,
                    style varchar(255),
                    issue_date date,
                    duration time,
                    lyricist_id int(8),
                    composer_id int(8),
                    album_id int (8),
                    primary key(`id`));

Create table Album(`id` int(8) not null auto_increment,
                    name varchar(255) not null,
                    issue_date date,
```

```
                    title_song_id int(8),
                    performer_id int(8),
                    primary key(`id`));

Create table Performer(`id` int(8) not null auto_increment,
                    company_id int(8),
                    primary key(`id`));

Create table Band(`id` int(8) not null auto_increment,
                    name varchar(255) not null,
                    region varchar(255),
                    performer_id int(8),
                    primary key(`id`));

Create table Singer(`id` int(8) not null auto_increment,
                    name varchar(255) not null,
                    gender varchar(255),
                    region varchar(255),
                    performer_id int(8),
                    primary key(`id`));

Create table Company(`id` int(8) not null auto_increment,
                    name varchar(255) not null,
                    region varchar(255),
                    established_time date,
                    primary key(`id`));

Create table Lyricist(`id` int(8) not null auto_increment,
                    name varchar(255) not null,
                    gender varchar(255),
                    primary key(`id`));

Create table Composer(`id` int(8) not null auto_increment,
                    name varchar(255) not null,
                    gender varchar(255),
                    primary key(`id`));

Create table SingSong(song_id int(8) not null,
                    performer_id int(8) not null,
                    primary key(song_id,performer_id));

Create table BandMember(band_id int(8) not null,
                    singer_id int(8) not null,
                    primary key(band_id,singer_id));
```

添加外键约束

```
alter table Song
    add constraint lyricsBy foreign key (lyricist_id)
    references Lyricist (`id`)
    ON DELETE SET NULL
    ON UPDATE SET NULL;

alter table Song
    add constraint Compose foreign key (composer_id)
    references Composer (`id`)
    ON DELETE SET NULL
    ON UPDATE SET NULL;

alter table Song
    add constraint album_member foreign key (album_id)
    references Album (`id`)
    ON DELETE SET NULL
    ON UPDATE SET NULL;

alter table Album
    add constraint title_song foreign key (title_song_id)
```

```
    references Song (`id`)
    ON DELETE SET NULL
    ON UPDATE SET NULL;

alter table Album
    add constraint create_album foreign key (performer_id)
    references Performer (`id`)
    ON DELETE SET NULL
    ON UPDATE SET NULL;

alter table Performer
    add constraint work_on foreign key (company_id)
    references Company (`id`)
    ON DELETE SET NULL
    ON UPDATE SET NULL;

alter table Band
    add constraint performer_band foreign key (performer_id)
    references Performer (`id`)
    ON DELETE SET NULL
    ON UPDATE SET NULL;

alter table Singer
    add constraint performer_Singer foreign key (performer_id)
    references Performer (`id`)
    ON DELETE SET NULL
    ON UPDATE SET NULL;

alter table SingSong
    add constraint sing_song foreign key (song_id)
    references Song (`id`)
    ON DELETE CASCADE
    ON UPDATE CASCADE;

alter table SingSong
    add constraint song_performer foreign key (performer_id)
    references Performer (`id`)
    ON DELETE CASCADE
    ON UPDATE CASCADE;

alter table BandMember
    add constraint bandmember_band foreign key (band_id)
    references Band (`id`)
    ON DELETE CASCADE
    ON UPDATE CASCADE;

alter table BandMember
    add constraint bandmember_singer foreign key (singer_id)
    references Singer (`id`)
    ON DELETE CASCADE
    ON UPDATE CASCADE;
```

附录3 MySQL-创建视图

```
create view PerformerName as
    select Performer.id as id, Band.name as band_name, Singer.name as singer_name
    from Performer left outer join Band on (Performer.id=Band.performer_id)
                   left outer join Singer on (Performer.id=Singer.performer_id);

create view SongOfPerformer as
    select SingSong.performer_id, SingSong.song_id, Song.name
    from  SingSong join Song on (SingSong.song_id=Song.id);

create view AlbumOfPerformer as
    select performer_id, id as album_id, name as album_name
    from  Album;
```

```
create view SingerInfor as
    select Singer.id, Singer.name, Singer.gender, Singer.region, Singer.performer_id, Company.name as company_
    from  Singer join Performer on (Singer.performer_id=Performer.id)
                left join Company on (Performer.company_id=Company.id);

create view BandInfor as
    select Band.id, Band.name, Band.region, Band.performer_id, Company.name as company_name, Company.id as com
    from  Band join Performer on (Band.performer_id=Performer.id)
                left join Company on (Performer.company_id=Company.id);

create view SingerOfBand as
    select BandMember.band_id, Singer.performer_id as singer_performer_id, Singer.name as singer_name
    from  BandMember join Band on (Band.id=BandMember.band_id)
                    join Singer on (Singer.id=BandMember.singer_id);

create view SongInfor as
    select Song.id, Song.name, Song.style, Song.issue_date, Song.duration,
           lyricist_id, Lyricist.name as lyricist_name,
           composer_id, Composer.name as composer_name,
           album_id, Album.name as album_name
    from  Song left outer join Album on (Song.album_id=Album.id)
               left outer join Composer on (Song.composer_id=Composer.id)
               left outer join Lyricist on (Song.lyricist_id=Lyricist.id);

create view SongOfAlbum as
    select Song.id as song_id, Song.name as song_name, Song.album_id
    from  Song;

create view AlbumInfor as
    select Album.id, Album.name, Album.issue_date, title_song_id, performer_id, Song.name as title_song_name
    from  Album join Song on (Album.title_song_id=Song.id);

create view PerformerOfCompany as
    select Company.id as company_id, Company.name as company_name, Performer.id as performer_id
    from  Company join Performer on (Company.id=Performer.company_id);
```

参考文献

1. CSDN博客——（七）数据库ER图（二） 2017/5/25
   http://blog.sina.com.cn/s/blog_68163d080100j6m6.html ↩
2. 数据库系统概念 第6版。 ↩