大家准备面试从以下方面入手

Python：列表，字典，三大器，元类（引入 Django ORM）
MySQL：隔离级别，脏读幻读，优化，索引
Redis：数据类型及应用场景，增删改查方法，击穿，穿透，雪崩及解决方法（布隆过滤器）
Django：流程，中间件
DRF：可以准备下源码，基本没问过
Celery：Broker，Backend，分布式系统下怎么布置
Docker：Dockerfile 中 ADD，COPY，RUN，CMD 等，镜像过大解决方法，docker-compose
Linux 常用指令：查看进程，磁盘，端口等
Git：基本没遇到问的，可以准备一写基本的
算法：快排，重复数据多的优化方法

Python

通过元类简单的实现一个类似于 django 的 orm

既然提到元类，那么什么是元类？
元类也就是创造类的类，在 python 中 type 就是一个元类
要实现 django 中的 orm，首先要了解属性描述符，下面给出两个属性描述符

```
class Field:
    def __init__(self, min_lenth, max_lenth):
        self.min_lenth = min_lenth
        self.max_lenth = max_lenth
        if min_lenth >= max_lenth:
            raise ValueError("最小值应该比最大值小")
class InField(Field):
    """数据描述符"""
    def __init__(self, min_lenth, max_lenth):
        self._value = None
        super(InField, self).__init__(min_lenth, max_lenth)
    def __get__(self, instance, owner):
        return self._value
    def __set__(self, instance, value):
        if isinstance(value, numbers.Integral):
```

```
            if self.min_lenth:
                if value < self.min_lenth:
                    raise ValueError("必须大于等于最小值")
            if self.max_lenth:
                if value > self.max_lenth:
                    raise ValueError("必须小于最大值")
            self._value = value
        else:
            raise ValueError("参数必须输一个 int 类型")
    def __delete__(self, instance):
        pass
class CharField(Field):
    """数据描述符"""
    def __init__(self, min_lenth, max_lenth):
        self._value = None
        super(CharField, self).__init__(min_lenth, max_lenth)
    def __get__(self, instance, owner):
        return self._value
    def __set__(self, instance, value):
        self._value = value
    def __delete__(self, instance):
        pass
```

编写一个元类，这里要讲的就是，既然我们要用元类创建一个类，那么当我们创建这个类的对象的时候用__new__这个魔法函数去控制实例化，在其中注入一些属性

```
class BaseModel(type):
    """因为继承了 type,那么 BaseMode 就是一个元类"""
    def __new__(cls, name, base, attr, **kwargs):
        """__new__魔法方法控制类的实例化"""
        # fields 的作用是把我们属于数据描述符的字段抽取出来，最后操作数据的时候用到
        fields = dict()
        for key, value in attr.items():
            if isinstance(value, Field):
                fields[key] = value
        meta = attr.get("Meta")
        db_table = name.lower()
```

```
        if meta:
            data = meta.__dict__
            if "db_table" in data:
                db_table = data.get("db_table")
            attr["data"] = data
            del attr["Meta"]
        attr["db_table"] = db_table
        attr["fields"] = fields
        # 养成好习惯调用父类的方法返回
        return super().__new__(cls, name, base, attr, **kwargs)
```

创建一个 model 父类

```
# 创建这个的原因呢,是因为我们要实现类似 django 的 orm 功能，因此，这个类中我们需要写一些操作数据库的方法，这样代码看起来回更加整洁，分离性更加好
```

```
class Model(metaclass=BaseModel):
    def __init__(self, *args, **kwargs):
        for key, value in kwargs.items():
            if getattr(self, key, None):
                setattr(self, key, value)
    def save(self):
        fields = []
        values = []
        for key, value in self.fields.items():
            fields.append(key)
            values.append(str(getattr(self, key)))
        sql = "insert in into {db_table}({fields}) values{values}".format(db_table=self.db_table, fields=",".join(fields),
values=",".join(values))
        print(sql)
```

创建 model

```
class User(Model):
    name = CharField(max_lenth=10, min_lenth=2)
    height = InField(max_lenth=300, min_lenth=50)
    class Meta:
```

```
        # 指明数据库中的数据表
        db_table = "用户"
if __name__ == "__main__":
    # user = User()
    # user.name = "witt"
    # user.height = 170
    user = User(name="witt", height=170)
    user.save()
```

这样就基本实现了一个 orm

通过元类简单实现 ORM 中的 insert 功能(django 必备)

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        mappings = dict()
        # 判断是否需要保存
        for k, v in attrs.items():
            # 判断是否是指定的 StringField 或者 IntegerField 的实例对象
            if isinstance(v, tuple):
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v
        # 删除这些已经在字典中存储的属性
        for k in mappings.keys():
            attrs.pop(k)
        # 将之前的 uid/name/email/password 以及对应的对象引用、类名字
        attrs['__mappings__'] = mappings  # 保存属性和列的映射关系
        attrs['__table__'] = name  # 假设表名和类名一致
        return type.__new__(cls, name, bases, attrs)
class User(metaclass=ModelMetaclass):
    uid = ('uid', "int unsigned")
    name = ('username', "varchar(30)")
    email = ('email', "varchar(30)")
    password = ('password', "varchar(30)")
    # 当指定元类之后，以上的类属性将不在类中，而是在__mappings__属性指定的字典中存储
    # 以上 User 类中有
    # __mappings__ = {
```

```
    #     "uid": ('uid', "int unsigned")
    #     "name": ('username', "varchar(30)")
    #     "email": ('email', "varchar(30)")
    #     "password": ('password', "varchar(30)")
    # }
    # __table__ = "User"
    def __init__(self, **kwargs):
        for name, value in kwargs.items():
            setattr(self, name, value)
    def save(self):
        fields = []
        args = []
        for k, v in self.__mappings__.items():
            fields.append(v[0])
            args.append(getattr(self, k, None))
        args_temp = list()
        for temp in args:
            # 判断入如果是数字类型
            if isinstance(temp, int):
                args_temp.append(str(temp))
            elif isinstance(temp, str):
                args_temp.append("""'%s'""" % temp)
        sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fields), ','.join(args_temp))
        print('SQL: %s' % sql)
u = User(uid=12345, name='Michael', email='test@orm.org', password='my-pwd')
# print(u.__dict__)
u.save()
```

执行的效果:

```
Found mapping: uid ==> ('uid', 'int unsigned')
Found mapping: password ==> ('password', 'varchar(30)')
Found mapping: name ==> ('username', 'varchar(30)')
Found mapping: email ==> ('email', 'varchar(30)')
SQL: insert into User (email,uid,password,username) values ('test@orm.org',12345,'my-pwd','Michael')
```

Day 3 - 编写 ORM

在一个 Web App 中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。在 awesome-python3-webapp 中，我们选择 MySQL 作为数据库。

Web App 里面有很多地方都要访问数据库。访问数据库需要创建数据库连接、游标对象，然后执行 SQL 语句，最后处理异常，清理资源。这些访问数据库的代码如果分散到各个函数中，势必无法维护，也不利于代码复用。

所以，我们要首先把常用的 SELECT、INSERT、UPDATE 和 DELETE 操作用函数封装起来。

由于 Web 框架使用了基于 asyncio 的 aiohttp，这是基于协程的异步模型。在协程中，不能调用普通的同步 IO 操作，因为所有用户都是由一个线程服务的，协程的执行速度必须非常快，才能处理大量用户的请求。而耗时的 IO 操作不能在协程中以同步的方式调用，否则，等待一个 IO 操作时，系统无法响应任何其他用户。

这就是异步编程的一个原则：一旦决定使用异步，则系统每一层都必须是异步，“开弓没有回头箭”。

幸运的是 `aiomysql` 为 MySQL 数据库提供了异步 IO 的驱动。

创建连接池

我们需要创建一个全局的连接池，每个 HTTP 请求都可以从连接池中直接获取数据库连接。使用连接池的好处是不必频繁地打开和关闭数据库连接，而是能复用就尽量复用。

连接池由全局变量 `__pool` 存储，缺省情况下将编码设置为 `utf8`，自动提交事务：

```
@asyncio.coroutine
def create_pool(loop, **kw):
    logging.info('create database connection pool...')
    global __pool
    __pool = yield from aiomysql.create_pool(
        host=kw.get('host', 'localhost'),
        port=kw.get('port', 3306),
        user=kw['user'],
        password=kw['password'],
        db=kw['db'],
        charset=kw.get('charset', 'utf8'),
        autocommit=kw.get('autocommit', True),
        maxsize=kw.get('maxsize', 10),
        minsize=kw.get('minsize', 1),
        loop=loop
    )
```

Select

要执行 SELECT 语句，我们用 `select` 函数执行，需要传入 SQL 语句和 SQL 参数：

```
@asyncio.coroutine
def select(sql, args, size=None):
    log(sql, args)
    global __pool
    with (yield from __pool) as conn:
        cur = yield from conn.cursor(aiomysql.DictCursor)
        yield from cur.execute(sql.replace('?', '%s'), args or ())
        if size:
            rs = yield from cur.fetchmany(size)
        else:
            rs = yield from cur.fetchall()
        yield from cur.close()
        logging.info('rows returned: %s' % len(rs))
        return rs
```

SQL 语句的占位符是`?`，而 MySQL 的占位符是`%s`，`select()`函数在内部自动替换。注意要始终坚持使用带参数的 SQL，而不是自己拼接 SQL 字符串，这样可以防止 SQL 注入攻击。

注意到`yield from`将调用一个子协程（也就是在一个协程中调用另一个协程）并直接获得子协程的返回结果。

如果传入`size`参数，就通过`fetchmany()`获取最多指定数量的记录，否则，通过`fetchall()`获取所有记录。

Insert, Update, Delete

要执行 INSERT、UPDATE、DELETE 语句，可以定义一个通用的`execute()`函数，因为这 3 种 SQL 的执行都需要相同的参数，以及返回一个整数表示影响的行数：

```
@asyncio.coroutine
def execute(sql, args):
    log(sql)
    with (yield from __pool) as conn:
        try:
            cur = yield from conn.cursor()
            yield from cur.execute(sql.replace('?', '%s'), args)
            affected = cur.rowcount
            yield from cur.close()
        except BaseException as e:
            raise
```

```
        return affected
```

`execute()`函数和`select()`函数所不同的是，cursor 对象不返回结果集，而是通过`rowcount`返回结果数。

ORM

有了基本的`select()`和`execute()`函数，我们就可以开始编写一个简单的 ORM 了。
设计 ORM 需要从上层调用者角度来设计。
我们先考虑如何定义一个`User`对象，然后把数据库表`users`和它关联起来。

```
from orm import Model, StringField, IntegerField
class User(Model):
    __table__ = 'users'
    id = IntegerField(primary_key=True)
    name = StringField()
```

注意到定义在`User`类中的`__table__`、`id`和`name`是类的属性，不是实例的属性。所以，在类级别上定义的属性用来描述`User`对象和表的映射关系，而实例属性必须通过`__init__()`方法去初始化，所以两者互不干扰：

```
# 创建实例:
user = User(id=123, name='Michael')
# 存入数据库:
user.insert()
# 查询所有 User 对象:
users = User.findAll()
```

定义 Model

首先要定义的是所有 ORM 映射的基类`Model`：

```
class Model(dict, metaclass=ModelMetaclass):
    def __init__(self, **kw):
        super(Model, self).__init__(**kw)
    def __getattr__(self, key):
        try:
```

```
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute '%s'" % key)
    def __setattr__(self, key, value):
        self[key] = value
    def getValue(self, key):
        return getattr(self, key, None)
    def getValueOrDefault(self, key):
        value = getattr(self, key, None)
        if value is None:
            field = self.__mappings__[key]
            if field.default is not None:
                value = field.default() if callable(field.default) else field.default
                logging.debug('using default value for %s: %s' % (key, str(value)))
                setattr(self, key, value)
        return value
```

`Model`从`dict`继承，所以具备所有`dict`的功能，同时又实现了特殊方法`__getattr__()`和`__setattr__()`，因此又可以像引用普通字段那样写：

```
>>> user['id']
123
>>> user.id
123
```

以及`Field`和各种`Field`子类：

```
class Field(object):
    def __init__(self, name, column_type, primary_key, default):
        self.name = name
        self.column_type = column_type
        self.primary_key = primary_key
        self.default = default
    def __str__(self):
        return '<%s, %s:%s>' % (self.__class__.__name__, self.column_type, self.name)
```

映射`varchar`的`StringField`：

```
class StringField(Field):
    def __init__(self, name=None, primary_key=False, default=None, ddl='varchar(100)'):
        super().__init__(name, ddl, primary_key, default)
```

注意到`Model`只是一个基类，如何将具体的子类如`User`的映射信息读取出来呢？答案就是通过 metaclass：`ModelMetaclass`：

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        # 排除 Model 类本身:
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        # 获取 table 名称:
        tableName = attrs.get('__table__', None) or name
        logging.info('found model: %s (table: %s)' % (name, tableName))
        # 获取所有的 Field 和主键名:
        mappings = dict()
        fields = []
        primaryKey = None
        for k, v in attrs.items():
            if isinstance(v, Field):
                logging.info('  found mapping: %s ==> %s' % (k, v))
                mappings[k] = v
                if v.primary_key:
                    # 找到主键:
                    if primaryKey:
                        raise RuntimeError('Duplicate primary key for field: %s' % k)
                    primaryKey = k
                else:
                    fields.append(k)
        if not primaryKey:
            raise RuntimeError('Primary key not found.')
        for k in mappings.keys():
            attrs.pop(k)
        escaped_fields = list(map(lambda f: ``%s`` % f, fields))
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
```

```
        attrs['__table__'] = tableName
        attrs['__primary_key__'] = primaryKey # 主键属性名
        attrs['__fields__'] = fields # 除主键外的属性名
        # 构造默认的 SELECT, INSERT, UPDATE 和 DELETE 语句:
        attrs['__select__'] = 'select `%s`, %s from `%s`' % (primaryKey, ', '.join(escaped_fields), tableName)
        attrs['__insert__'] = 'insert into `%s` (%s, `%s`) values (%s)' % (tableName, ', '.join(escaped_fields), primaryKey,
create_args_string(len(escaped_fields) + 1))
        attrs['__update__'] = 'update `%s` set %s where `%s`=?' % (tableName, ', '.join(map(lambda f: '`%s`=?' %
(mappings.get(f).name or f), fields)), primaryKey)
        attrs['__delete__'] = 'delete from `%s` where `%s`=?' % (tableName, primaryKey)
        return type.__new__(cls, name, bases, attrs)
```

这样，任何继承自 Model 的类（比如 User），会自动通过 ModelMetaclass 扫描映射关系，并存储到自身的类属性如`__table__`、`__mappings__`中。
然后，我们往 Model 类添加 class 方法，就可以让所有子类调用 class 方法：

```
class Model(dict):
    ...
    @classmethod
    @asyncio.coroutine
    def find(cls, pk):
        ' find object by primary key. '
        rs = yield from select('%s where `%s`=?' % (cls.__select__, cls.__primary_key__), [pk], 1)
        if len(rs) == 0:
            return None
        return cls(**rs[0])
```

User 类现在就可以通过类方法实现主键查找：

```
user = yield from User.find('123')
```

往 Model 类添加实例方法，就可以让所有子类调用实例方法：

```
class Model(dict):
    ...
    @asyncio.coroutine
    def save(self):
```

```
        args = list(map(self.getValueOrDefault, self.__fields__))
        args.append(self.getValueOrDefault(self.__primary_key__))
        rows = yield from execute(self.__insert__, args)
        if rows != 1:
            logging.warn('failed to insert record: affected rows: %s' % rows)
```

这样，就可以把一个 User 实例存入数据库：

```
user = User(id=123, name='Michael')
yield from user.save()
```

最后一步是完善 ORM，对于查找，我们可以实现以下方法：

- findAll() - 根据 WHERE 条件查找；
- findNumber() - 根据 WHERE 条件查找，但返回的是整数，适用于 `select count(*)`类型的 SQL。

以及 `update()`和 `remove()`方法。

所有这些方法都必须用`@asyncio.coroutine`装饰，变成一个协程。

调用时需要特别注意：

```
user.save()
```

没有任何效果，因为调用 `save()`仅仅是创建了一个协程，并没有执行它。一定要用：

```
yield from user.save()
```

才真正执行了 INSERT 操作。

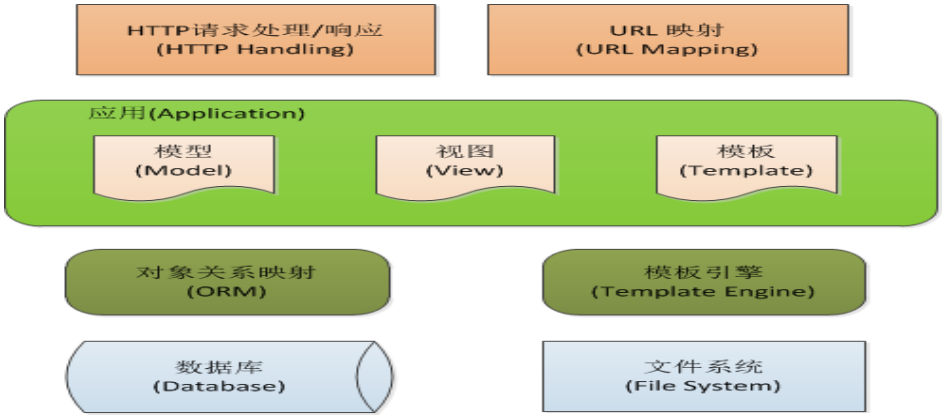
最后看看我们实现的 ORM 模块一共多少行代码？累计不到 300 多行。用 Python 写一个 ORM 是不是很容易呢？

廖雪峰 day-03

django 处理流程分析

一、 处理过程的核心概念

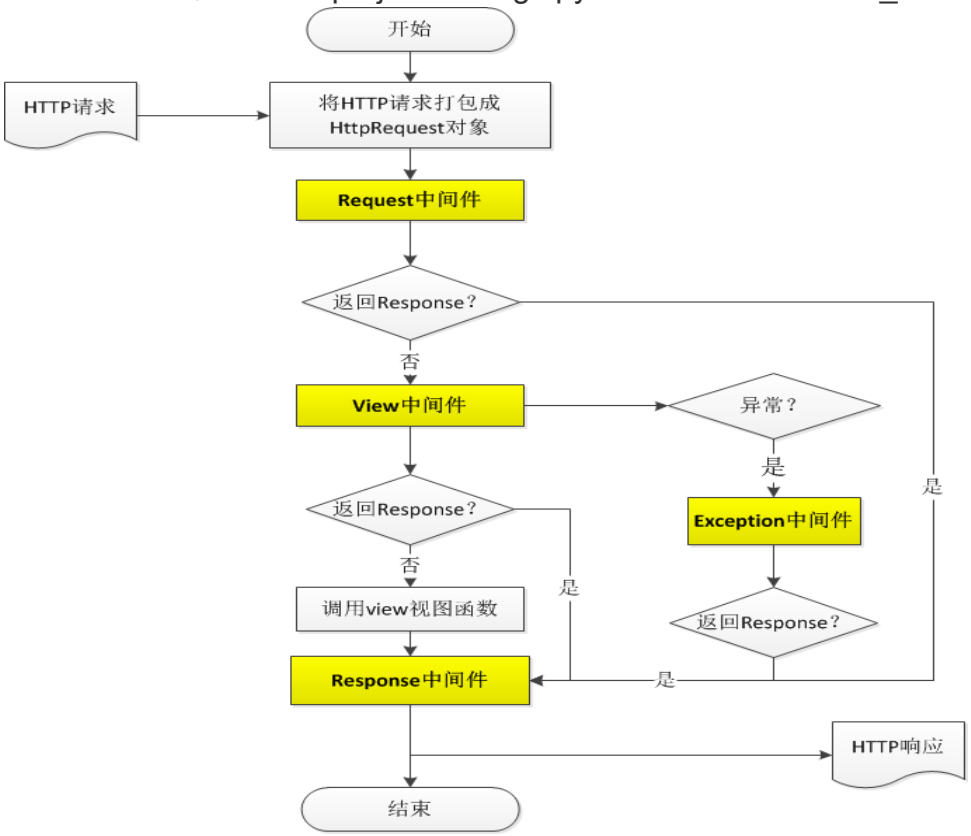
如下图所示 django 的总览图，整体上把握以下 django 的组成：



Django架构总览图

核心在于中间件 middleware，django 所有的请求、返回都由中间件来完成。

中间件，就是处理 HTTP 的 request 和 response 的，类似插件，比如有 Request 中间件、view 中间件、response 中间件、exception 中间件等，Middleware 都需要在 “project/settings.py” 中 MIDDLEWARE_CLASSES 的定义。大致的程序流程图如下所示：



首先，Middleware 都需要在 “project/settings.py” 中 MIDDLEWARE_CLASSES 的定义， 一个 HTTP 请求，将被这里指定的中间件从头到尾处理一遍，暂且称这些需要挨个处理的中间件为处理链，如果链中某个处理器处理后没有返回 response，就把请求传递给下一个处理器；如果链中某个处理器返回了 response，直接跳出处理链由 response 中间件处理后返回给客户端，可以称之为短路处理。

二、 中间件中的方法

Django 处理一个 Request 的过程是首先通过中间件，然后再通过默认的 URL 方式进行的。我们可以在 Middleware 这个地方把所有 Request 拦截住，用我们自己的方式完成处理以后直接返回 Response。因此了解中间件的构成是非常必要的。

Initializer: __init__(self)

出于性能的考虑，每个已启用的中间件在每个服务器进程中只初始化一次。也就是说 __init__() 仅在服务进程启动的时候调用，而在针对单个 request 处理时并不执行。

对一个 middleware 而言，定义 __init__() 方法的通常原因是检查自身的必要性。如果 __init__() 抛出异常 django.core.exceptions.MiddlewareNotUsed ,则 Django 将从 middleware 栈中移出该 middleware。

在中间件中定义 __init__() 方法时，除了标准的 self 参数之外，不应定义任何其它参数。

Request 预处理函数: process_request(self, request)

这个方法的调用时机**在 Django 接收到 request 之后，但仍未解析 URL 以确定应当运行的 view 之前**。Django 向它传入相应的 HttpRequest 对象，以便在方法中修改。

process_request() 应当返回 None 或 HttpResponse 对象.

如果返回 None , Django 将继续处理这个 request，执行后续的中间件， 然后调用相应的 view.

如果返回 HttpResponse 对象，Django 将不再执行任何其它的中间件(无视其种类)以及相应的 view。 Django 将立即返回该 HttpResponse .

View 预处理函数: process_view(self, request, callback, callback_args, callback_kwargs)

这个方法的调用时机在 **Django 执行完 request 预处理函数并确定待执行的 view 之后**，但**在 view 函数实际执行之前**。

request

HttpRequest 对象。

callback

Django 将调用的处理 request 的 python 函数. 这是实际的函数对象本身, 而不是字符串表述的函数名。

args

将传入 view 的**位置参数列表**，但不包括 request 参数(它通常是传入 view 的第一个参数)

kwargs

将传入 view 的关键字参数字典.

如同 process_request() , process_view() 应当返回 None 或 HttpResponse 对象。

如果返回 None , Django 将继续处理这个 request ，执行后续的中间件， 然后调用相应的 view

如果返回 HttpResponse 对象，Django 将不再执行 任何 其它的中间件(不论种类)以及相应的 view。Django 将立即返回

Response 后处理函数: process_response(self, request, response)

这个方法的**调用时机在 Django 执行 view 函数并生成 response 之后**。

该处理器能修改 response 的内容；一个常见的用途是内容压缩，如 gzip 所请求的 HTML 页面。

这个方法的参数相当直观: request 是 request 对象，而 response 则是从 view 中返回的 response 对象。

process_response() 必须 返回 HttpResponse 对象. 这个 response 对象可以是传入函数的那一个原始对象(通常已被修改)，也可以是全新生成的。

Exception 后处理函数: process_exception(self, request, exception)

这个方法**只有在 request 处理过程中出了问题并且 view 函数抛出了一个未捕获的异常时才会被调用**。这个钩子**可以用来发送错误通知，将现场相关信息输出到日志文件, 或者甚至尝试从错误中自动恢复**。

这个函数的参数除了一贯的 request 对象之外，还包括 view 函数抛出的实际的异常对象 exception 。

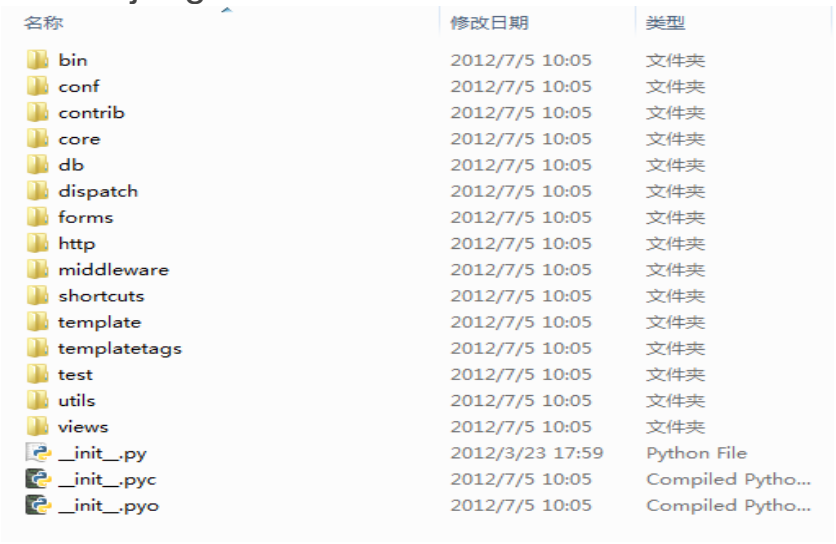
process_exception() 应当返回 None 或 HttpResponse 对象.

如果返回 None , Django 将用框架内置的异常处理机制继续处理相应 request。

如果返回 HttpResponse 对象, Django 将使用该 response 对象，而短路框架内置的异常处理机制

以下几章会详细介绍该机制。

三、 Django 目录结构



名称	修改日期	类型
bin	2012/7/5 10:05	文件夹
conf	2012/7/5 10:05	文件夹
contrib	2012/7/5 10:05	文件夹
core	2012/7/5 10:05	文件夹
db	2012/7/5 10:05	文件夹
dispatch	2012/7/5 10:05	文件夹
forms	2012/7/5 10:05	文件夹
http	2012/7/5 10:05	文件夹
middleware	2012/7/5 10:05	文件夹
shortcuts	2012/7/5 10:05	文件夹
template	2012/7/5 10:05	文件夹
templatetags	2012/7/5 10:05	文件夹
test	2012/7/5 10:05	文件夹
utils	2012/7/5 10:05	文件夹
views	2012/7/5 10:05	文件夹
__init__.py	2012/3/23 17:59	Python File
__init__.pyc	2012/7/5 10:05	Compiled Pytho...
__init__.pyo	2012/7/5 10:05	Compiled Pytho...

conf

主要有两个作用：1) 处理全局配置，比如数据库、加载的应用、 MiddleWare 等 2) 处理 urls 配置，就是 url 与 view 的映射关系。

contrib (贡献)

由 Django 的开发者贡献的功能模块，不过既然都已经随版本发布， 就表示是官方的。

core

Django 的核心处理库，包括 url 分析、处理请求、缓存等，其中处理请求是核心了，比如处理 fastcgi 就是由 wsgi.py 处理。

db

顾名思义，处理与数据库相关的，就是 ORM。

dispatch (分派，派遣)

其实这不是 Django 原创，是 pydispatch 库，主要处 理消费者-工作者模式。

forms && newforms && oldforms

处理 html 的表单，不用多介绍。

middleware

中间件，就是处理 HTTP 的 request 和 response 的，类似插件。比 如默认的 common 中间件的一个功能：当一个页面没有找对对应的 pattern 时， 会自定加上'/'重新处理。比如访问/blog 时，而定义的 pattern 是'^blog/$'， 所以找不到对应的 pattern，会自动再用/blog/查找，当然前提是 APPEND_SLASH=True。

template

Django 的模板

templatetags

处理 Application 的 tag 的 wrapper，就是将 INSTALLED_APPS 中 所有的 templatetags 目录添加到 django.templatetags 目录中，则当使用 {{load blog}}记载 tag 时，就可以使用 import django.templatetags.blog 方式加载了。不过这有一个问题，如果其他 Application 目录中也有 blog.py， 这会加载第一个出现 blog.py 的 tag。其实在 Django 中，有许多需要处理重名 的地方，比如 template，需要格外小心，这个后续在介绍。

utils

公共库，很多公用的类都在放在这里。

views

最基本的 view 方法。

四、 Django 术语

在应用 Django 的时候，我们经常听到一些术语：

Project

指一个完整的 Web 服务，一般由多个模块组成。

Application

可以理解为模块，比如用户管理、博客管理等，包含了数据的组成和数据的显示，Applicaiton 都需要在 "project/settings.py" 中 INSTALLED_APPS 的定义。

Middleware

就是处理 request 和 response 的插件, Middleware 都需要在 "project/settings.py" 中 MIDDLEWARE_CLASSES 的定义。

Loader

模板加载器, 其实就是为了读取 Template 文件的类，默认的有通过文件系统加载和在 "Application/templates" 目录中加载，Loader 都需要在 "project/settings.py" 中 TEMPLATE_LOADERS 的定义。

五、 处理流程

其实和其他 Web 框架一样，HTTP 处理的流程大致相同，

Django 处理一个 Request 的过程是首先通过中间件，然后再通过默认的 URL 方式进行的。我们可以在 Middleware 这个地方把所有 Request 拦截住，用我们自己的方式完成处理以后直接返回 Response。

1. 加载配置

Django 的配置都在 “Project/settings.py” 中定义，可以是 Django 的配置，也 可以是自定义的配置，并且都通过 django.conf.settings 访问，非常方便。

2. 启动

最核心动作的是通过 django.core.management.commands.runfcgi 的 Command 来启动，它运行 django.core.servers.fastcgi 中的 runfastcgi ， runfastcgi 使用了 flup 的 WSGIServer 来启动 fastcgi 。而 WSGIServer 中携带了 django.core.handlers.wsgi 的 WSGIHandler 类的一个实例，通过 WSGIHandler 来处理由 Web 服务器(比如 Apache，Lighttpd 等)传过来的请求，此 时才是真正进入 Django 的世界。

3. 处理 Request

当有 HTTP 请求来时， WSGIHandler 就开始工作了,它从 BaseHandler 继承而来。 WSGIHandler 为每个请求创建一个 WSGIRequest 实例，而 WSGIRequest 是从 http.HttpRequest 继承而来。接下来就开始创建 Response 了.

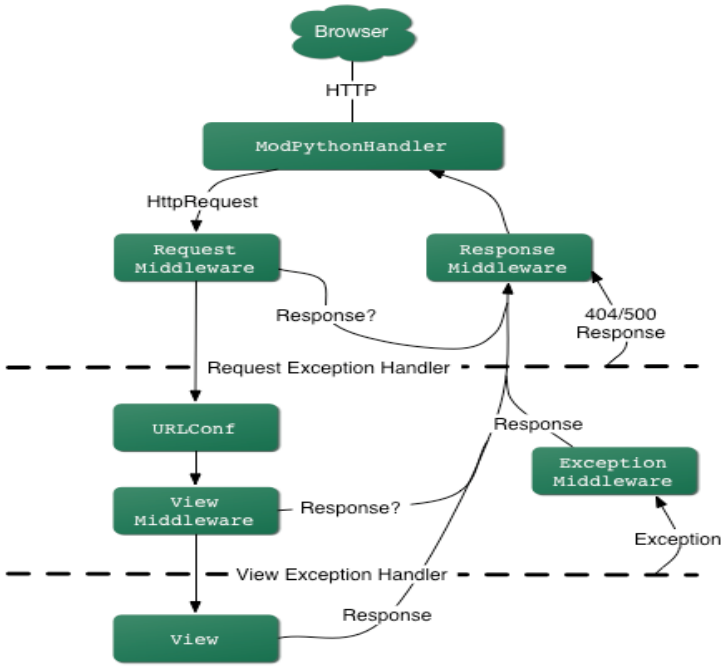
4. 创建 Response

BaseHandler 的 get_response 方法就是根据 request 创建 response ， 而 **具体生成 response 的动作就是执行 urls.py 中对应的 view 函数了**，这也是 Django 可以处理“友好 URL”的关键步骤，每个这样的函数都要返回一个 Response 实例。此时一般的做法是通过 loader 加载 template 并生成页面内 容，其中重要的就是通过 ORM 技术从数据库中取出数据，并渲染到 Template 中，从而生成具体的页面了

5. 处理 Response

Django 返回 Response 给 flup ， flup 就取出 Response 的内容返回给 Web 服务器，由后者返回给浏览器。

总之， Django 在 fastcgi 中主要做了两件事：处理 Request 和创建 Response ， 而它们对应的核心就是“urls 分析”、“模板技术”和“ORM 技术”



如图所示，一个 HTTP 请求，首先被转化成一个 HttpRequest 对象，然后该对象被传递给 Request 中间件处理，如果该中间件返回了 Response，则直接传递给 Response 中间件做收尾处理。否则的话 Request 中间件将访问 URL 配置，确定哪个 view 来处理，在确定了哪个 view 要执行，但是还没有执行该 view 的时候，系统会把 request 传递给 View 中间件处理器进行处理，如果该中间件返回了 Response，那么该 Response 直接被传递给 Response 中间件进行后续处理，否则将执行确定的 View 函数处理并返回 Response，在这个过程中如果引发了异常并抛出，会被 Exception 中间件处理器进行处理。

六、 详细全流程

一个 Request 到达了！

首先发生的是一些和 Django 有关(前期准备）的其他事情，分别是：

1. 如果是 Apache/mod_python 提供服务，request 由 mod_python 创建的 django.core.handlers.modpython.ModPythonHandler 实例传递给 Django。
2. 如果是其他服务器，则必须兼容 WSGI，这样，服务器将创建一个 django.core.handlers.wsgi.WsgiHandler 实例。

这两个类都继承自 django.core.handlers.base.BaseHandler，它包含对任何类 型的 request 来说都需要的公共代码。

有一个处理器(Handler)了

当上面其中一个处理器实例化后，紧接着发生了一系列的事情：

1. 这个处理器（handler）导入你的 Django 配置文件。
2. 这个处理器导入 Django 的自定义异常类。
3. **这个处理器调用它自己的 load_middleware 方法，加载所有列在 MIDDLEWARE_CLASSES 中的 middleware 类并且内省它们**。

最后一条有点复杂，我们仔细瞧瞧。

一 个 middleware 类可以渗入处理过程的四个阶段：request，view，response 和 exception。要做到这一点，只需要定义指定的、恰当的方 法：process_request，process_view， process_response 和 process_exception。middleware 可以定义其中任何一个或所有这些方法，这取 决于你想要它提供什么样的功能。

当处理器内省 middleware 时，它查找上述名字的方法，并建立四个列表作为处理器的实例变量：

1. _request_middleware 是一个保存 process_request 方法的列表（在每一 种情况下，它们是真正的方法，可以直接调用），这些方法来自于任一个定 义了它们的 middleware 类。
1. _view_middleware 是一个保存 process_view 方法的列表，这些方法来自 于任一个定义了它们的 middleware 类。
2. _response_middleware 是一个保存 process_response 方法的列表，这些 方法来自于任一个定义了它们的 middleware 类。
3. _exception_middleware 是一个保存 process_exception 方法的列表，这 些方法来自于任一个定义了它们的 middleware 类。

绿灯：现在开始

现在处理器已经准备好真正开始处理了，因此它给调度程序发送一个信号 request_started（Django 内部的调度程序允许各种不同的组件声明它们正在干 什么，并可以写一些代码监听特定的事件。关于这一点目前还没有官方的文档， 但在 wiki 上有一些注释。）。**接下来它实例化一个 django.http.HttpRequest 的子类**。根据不同的处理器，可能是 django.core.handlers.modpython.ModPythonRequest 的一个实例，也可能是 django.core.handlers.wsgi.WSGIRequest 的一个实例。需要两个不同的类是因 为 mod_python 和 WSGI APIs 以不同的格式传入 request 信息，这个信息需要 解析为 Django 能够处理的一个单独的标准格式。

一旦一个 HttpRequest 或者类似的东西存在了，处理器就调用它自己的 get_response 方法，传入这个 HttpRequest 作为唯一的参数。这里就是几乎所 有真正的活动发生的地方。

Middleware，第一回合

get_response 做的第一件事就是遍历处理器的 _request_middleware 实例变量 并调用其中的每一个方法，传入 HttpRequest 的实例作为参数。

for middleware_method in self._request_middleware:

response = middleware_method(request)

if response:

break

这些方法可以选 择短路剩下的处理并立即让 get_response 返回，通过返回自身的一个值（如果 它们这样做，返回值必须是 django.http.HttpResponse 的一个实例，后面会讨论到）。如果其中之一这样做了，我们会立即回到主处理器代码，get_response 不会等着看其它 middleware 类想要做什么，它直接返回，然后处理器进入 response 阶段。

然而，更一般的情况是，**这里应用的 middleware 方法简单地做一些处理并决定 是否增加，删除或补充 request 的属性。**

关于解析

假设没有一个作用于 request 的 middleware 直接返回 response，处理器下一 步会尝试解析请求的 URL。它在配置文件中寻找一个叫做 ROOT_URLCONF 的配 置，用这个配置加上根 URL /，作为参数来创建 django.core.urlresolvers.RegexURLResolver 的一个实例，然后调用它的 resolve 方法来解析请求的 URL 路径。

URL resolver 遵循一个相当简单的模式。对于在 URL 配置文件中根据 ROOT_URLCONF 的配置产生的每一个在 urlpatterns 列表中的条目，它会检查请 求的 URL 路径是否与这个条目的正则表达式相匹配，如果是的话，有两种选择：

1. 如果这个条目有一个可以调用的 include，resolver 截取匹配的 URL，转 到 include 指定的 URL 配置文件并开始遍历其中 urlpatterns 列表中的 每一个条目。根据你 URL 的深度和模块性，这可能重复好几次。

2. 否则，resolver 返回三个条目：匹配的条目指定的 view function；一个 从 URL 得到的未命名匹配组（被用来作为 view 的位置参数）；一个关键 字参数字典，它由从 URL 得到的任意命名匹配组和从 URLConf 中得到的任 意其它关键字参数组合而成。

注意这一过程会在匹配到第一个指定了 view 的条目时停止，因此最好让你的 URL 配置从复杂的正则过渡到简单的正则，这样能确保 resolver 不会首先匹配 到简单的那一个而返回错误的 view function。

如果没有找到匹配的条目，resolver 会产生 django.core.urlresolvers.Resolver404 异常，它是 django.http.Http404 例 外的子类。后面我们会知道它是如何处理的。

Middleware，第二回合

一旦知道了所需的 view function 和相关的参数，处理器就会查看它的 _view_middleware 列表，并调用其中的方法，传入 HttpRequst，view function，针对这个 view 的位置参数列表和关键字参数字典。

Apply view middleware

for middleware_method in self._view_middleware:

response = middleware_method(request, callback, callback_args, callback_kwargs)

if response:

break

还有，middleware 有可能介入这一阶段并强迫处理器立即返回。

进入 View

如果处理过程这时候还在继续的话，处理器会调用 view function。Django 中的 Views 不很严格因为它只需要满足几个条件：

² 必须可以被调用。

² 必须接受 django.http.HttpRequest 的实例作为第一位值参数。

² 必须能产生一个异常或返回 django.http.HttpResponse 的一个实例。

除了这些，你就可以天马行空了。尽管如此，一般来说，views 会使用 Django 的 database API 来创建，检索，更新和删除数据库的某些东西，还会加载并渲 染一个模板来呈现一些东西给最终用户。

模板

Django 的模板系统有两个部分：一部分是给设计师使用的混入少量其它东西的 HTML，另一部分是给程序员使用纯 Python。

从一个 HTML 作者的角度，Django 的模板系统非常简单，需要知道的仅有三个结构：

² 变量引用。在模板中是这样： {{ foo }}。

² 模板过滤。在上面的例子中使用过滤竖线是这样：{{ foo|bar }}。通常这 用来格式化输出（比如：运行 Textile，格式化日期等等）。

² 模板标签。是这样：{% baz %}。这是模板的“逻辑”实现的地方，你可以 {% if foo %}，{% for bar in foo %}，等等，if 和 for 都是模板标签。

变量引用以一种非常简单的方式工作。如果你只是要打印变量，只要 {{ foo }}，模板系统就会输出它。这里唯一的复杂情况是 {{ foo.bar }}，这时模板系 统按顺序尝试几件事：

1. 首先它尝试一个字典方式的查找，看看 foo[‘bar’] 是否存在。如果存在， 则它的值被输出，这个过程也随之结束。
2. 如果字典查找失败，模板系统尝试属性查找，看看 foo.bar 是否存在。同 时它还检查这个属性是否可以被调用，如果可以，调用之。
3. 如果属性查找失败，模板系统尝试把它作为列表索引进行查找。

如果所有这些都失败了，模板系统输出配置 TEMPLATE_STRING_IF_INVALID 的值，默认是空字符串。

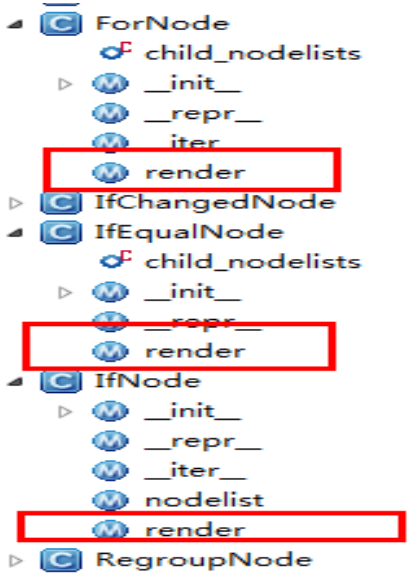
模板过滤就是简单的 Python functions，它接受一个值和一个参数，返回一个新的值。比如，date 过滤用一个 Python datetime 对象作为它的值，一个标准的 strftime 格式化字符串作为它的参数，返回对 datetime 对象应用了格式化字符 串之后的结果。

模板标签用在事情有一点点复杂的地方，它是你了解 Django 的模板系统是如何 真正工作的地方。

Django 模板的结构

在内部，一个 Django 模板体现为一个 “nodes” 集合，它们都是从基本的 django.template.Node 类继承而来。Nodes 可以做各种处理，但有一个共同点： 每一个 Node 必须有一个叫做 render 的方法，它接受的第二个参数（第一个参 数，显然是 Node 实例）是 django.template.Context 的一个实例，这是一个类 似于字典的对象，包含所有模板可以获得的变量。Node 的 render 方法必须返回 一个字符串，但如果 Node 的工作不是输出（比如，它是要通过增加，删除或修 改传入的 Context 实例变量中的变量来修改模板上下文），可以返回空字符串。

Django 包含许多 Node 的子类来提供有用的功能。比如，每个内置的模板标签都 被一个 Node 的子类处理（比如，IfNode 实现了 if 标签，ForNode 实现了 for 标签，等等）。所有内置标签可以在 django.template.defaulttags 找到。



实际上，上面介绍的所有模板结构都是某种形式的 Nodes，纯文本也不异常。变 量查找由 VariableNode 处理，出于自然，过滤也应用在 VariableNode 上，标签是各种类型的 Nodes，纯文本是一个 TextNode。

一般来说，一个 view 渲染一个模板要经过下面的步骤，依次是：

1. 加载需要渲染的模板。这是由 django.template.loader.get_template 完 成的，它能利用这许多方法中的任意一个来定位需要的模板文件。 get_template 函数返回一个 django.template.Template 实例，其中包含 经过解析的模板和用到的方法。

2. 实例化一个 Context 用来渲染模板。如果用的是 Context 的子类 django.template.RequestContext，那么附带的上下文处理函数就会自动添 加在 view 中没有定义的变量。Context 的构建器方法用一个键/值对的字 典（对于模板，它将变为名/值变量）作为它唯一的参数，RequestContext 则用 HttpRequest 的一个实例和一个字典。

3. 调用 Template 实例的 render 方法，Context 对象作为第一个位置参数。

Template 的 render 方法的返回值是一个字符串，它由 Template 中所有 Nodes 的 render 方法返回的值连接而成，调用顺序为它们出现在 Template 中 的顺序。

关于 Response，一点点

一旦一个模板完成渲染，或者产生了其它某些合适的输出，view 就会负责产生一 个 django.http.HttpResponse 实例，它的构建器接受两个可选的参数：

1. 一个作为 response 主体的字符串（它应该是第一位置参数，或者是关键字 参数 content）。大部分时间，这将作为渲染一个模板的输出，但不是必须 这样，在这里你可以传入任何有效的 Python 字符串。

2. 作 为 response 的 Content-Type header 的值（它应该是第二位置参数， 或者是关键字参数 mine_type）。如果没有提供这个参数，Django 将会使 用配置中 DEFAULT_MIME_TYPE 的值和 DEFAULT_CHARSET 的值，如果你没有 在 Django 的全局配置文件中更改它们的话，分别是 “text/html” 和 “utf-8”。

Middleware，第三回合：异常

如果 view 函数，或者其中的什么东西，发生了异常，那么 get_response（我知 道我们已经花了些时间深入 views 和 templates，但是一旦 view 返回或产生异常，我们仍将重拾处理器中间的 get_response 方法）将遍历它的 _exception_middleware 实例变量并调用那里的每个方法，传入 HttpResponse 和这个 exception 作为参数。如果顺利，这些方法中的一个会实例化一个 HttpResponse 并返回它。

仍然没有响应？

这时候有可能还是没有得到一个 HttpResponse，这可能有几个原因：

1. view 可能没有返回值。
2. view 可能产生了异常但没有一个 middleware 能处理它。
3. 一个 middleware 方法试图处理一个异常时自己又产生了一个新的异常。

这时候，get_response 会回到自己的异常处理机制中，它们有几个层次：

1. 如果 exception 是 Http404 并且 DEBUG 设置为 True，get_response 将 执行 view django.views.debug.technical_404_response，传入 HttpRequest 和 exception 作为参数。这个 view 会展示 URL resolver 试图匹配的模式信息。
2. 如果 DEBUG 是 False 并且异常是 Http404，get_response 会调用 URL resolver 的 resolve_404 方法。这个方法查看 URL 配置以判断哪一个 view 被指定用来处理 404 错误。默认是 django.views.defaults.page_not_found，但可以在 URL 配置中给 handler404 变量赋值来更改。
3. 对于任何其它类型的异常，如果 DEBUG 设置为 True，get_response 将执 行 view django.views.debug.technical_500_response，传入 HttpRequest 和 exception 作为参数。这个 view 提供了关于异常的详细 信息，包括 traceback，每一个层次 stack 中的本地变量，HttpRequest 对象的详细描述和所有无效配置的列表。
4. 如果 DEBUG 是 False，get_response 会调用 URL resolver 的 resolve_500 方法，它和 resolve_404 方法非常相似，这时默认的 view 是 django.views.defaults.server_error，但可以在 URL 配置中给 handler500 变量赋值来更改。

此外，对于除了 django.http.Http404 或 Python 内置的 SystemExit 之外的任 何异常，处理器会给调度者发送信号 got_request_exception，在返回之前，构建一个关于异常的描述，把它发送给列在 Django 配置文件的 ADMINS 配置中的 每一个人。

Middleware，最后回合

现在，**无论 get_response 在哪一个层次上发生错误，它都会返回一个 HttpResponse 实例，因此我们回到处理器的主要部分。**一旦它获得一个 HttpResponse 它做的第一件事就是遍历它的 _response_middleware 实例变量并 应用那里的方法，传入 HttpRequest 和 HttpResponse 作为参数。

finally:

```
# Reset URLconf for this thread on the way out for complete
# isolation of request.urlconf
urlresolvers.set_urlconf(None)
try:
# Apply response middleware, regardless of the response
for middleware_method in self._response_middleware:
response = middleware_method(request, response)
```

response = self.apply_response_fixes(request, response)

注意对于任何想改变点什么的 middleware 来说，这是它们的最后机会。

The check is in the mail

是该结束的时候了。一旦 middleware 完成了最后环节，处理器将给调度者发送 信号 request_finished，对与想在当前的 request 中执行的任何东西来说，这 绝对是最后的调用。监听这个信号的处理者会清空并释放任何使用中的资源。比 如，Django 的 request_finished 监听者会关闭所有数据库连接。

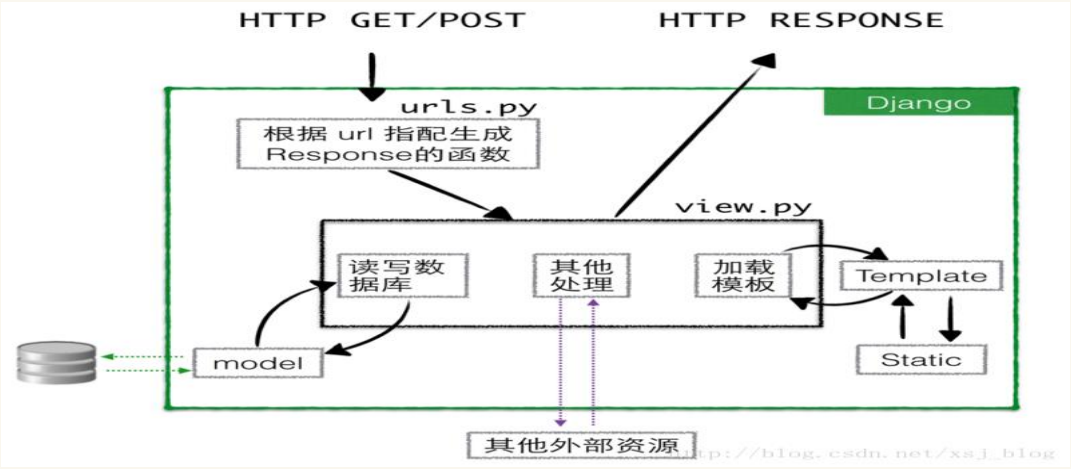
这件事发生以后，处理器会构建一个合适的返回值送返给实例化它的任何东西 （现在，是一个恰当的 mod_python response 或者一个 WSGI 兼容的 response，这取决于处理器）并返回。

呼呼

结束了，从开始到结束，这就是 Django 如何处理一个 request。

一、Django 工作流程

在开始具体的代码之旅前，先来宏观地看下 Django 是如何处理 Http Resquest 的，如下图：



假设你已经在浏览器输入了 http://127.0.0.1:8000/polls/，接下来浏览器会把请求交给 Django 处理。根据上图，我们知道 Django 需要根据 url 来决定交给谁来处理请求，那么 Django 是如何完成这项工作呢？很简单，Django 要求程序员提供 urls.py 文件，并且在该类文件中指定请求链接与处理函数之间的一一对应关系。

在 Django 中的 urls.py 添加以下语句，即可指定请求链接与处理函数之间的一一对应关系。

```
urlpatterns = patterns(
    '',
    url(r'^polls/$', views.index),
)
```

这样当请求链接为 http://127.0.0.1:8000/polls/时，就会用 views.py 中的函数 index()来处理请求。现在 Django 知道由 index 来处理请求了，那么 index 需要做哪些工作呢？

它需要加载返回内容的模板，这里比如说是 index.html。

```
def index(request):
    return render(request,'index.html')
```

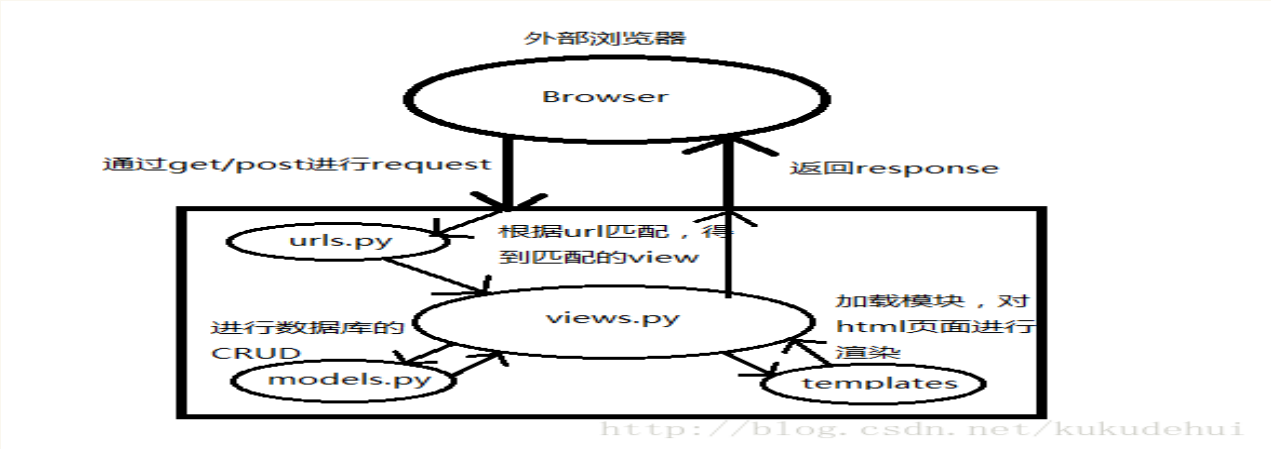
在模板方面，模板文件就是返回页面的一个骨架，我们可以在模板中指定需要的静态文件，也可以在模板中使用一些参数和简单的逻辑语句，这样就可以将其变为用户最终看到的丰满的页面了。

要使用静态文件，比如说 css、javascript 等，只需要用{% load staticfiles %}来声明一下，然后直接引用即可。

在数据库方面，Django 给我们封装了数据库的读写操作，我们不需要用 SQL 语句去查询、更新数据库等，我们要做的是用 python 的方式定义数据库结构(在 model.py 里面定义数据库)，然后用 python 的方式去读写内容。至于连接数据库、关闭数据库这些工作交给 Django 去替你完成吧。

至此，整个框架的简单介绍结束。

二、



这张图片是我对 django 工作流程一个大致的分析。

在你写好一个完整的 django 后，它的工作流程应该是这样的：

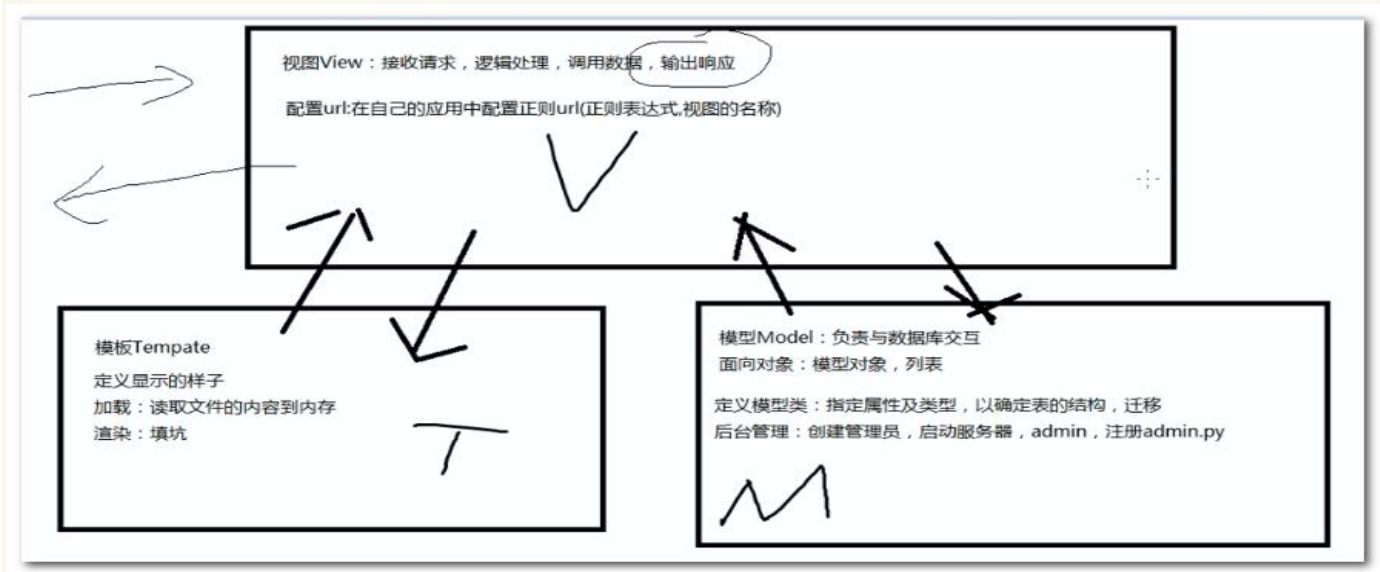
1.用户在客户端浏览器输入 URL 地址，通过 get/post 请求方式，向服务端发起请求。

2.django 服务端接收到客户端请求，通过 urls.py 中地址与处理函数之间的一一对应，找到对应的视图函数。

3.开始执行对应视图函数中的逻辑，通过与 models 交互，进行数据库的 CRUD，django 已经封装好了数据库操作方法，不需要额外的 sql 语句在进行数据库操作。

4.与 templates 交互，将参数返回到前端页面，并通过 templates 进行 html 页面渲染。

这是我学习 django 框架后对它的工作流程的简单了解，如果有不对的地方，欢迎指正。



django 中间件

阅读目录

回到顶部

一、什么是中间件

中间件顾名思义，是**介于 request 与 response 处理之间的一道处理过程**，相对比较轻量级，并且在全局上改变 django 的输入与输出。因为改变的是全局，所以需要谨慎实用，用不好会影响到性能

django 中间价官网定义：

```
Middleware is a framework of hooks into Django’s request/response processing.
It’s a light, low-level “plugin” system for globally altering Django’s input or output.
```

中间件位于 web 服务端与 url 路由层之间

回到顶部

二、中间件有什么用

如果你想修改请求，例如被传送到 view 中的 **HttpRequest** 对象。 或者你想修改 view 返回的 **HttpResponse** 对象，这些都可以通过中间件来实现。

可能你还想在 view 执行之前做一些操作，这种情况就可以用 middleware 来实现。
Django 默认的中间件：（在 django 项目的 settings 模块中，有一个 MIDDLEWARE_CLASSES 变量，其中每一个元素就是一个中间件，如下图）

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    # 'app01.mymid.MyMid1',
    # 'app01.mymid.MyMid2',
]
```

请求进来是自上而下，通过反射找到类，用 for 循环来执行，可以自定义中间件，但是也要写在 MIDDLEWARE 中，可以在 app01 下创建一个 mymid.py 文件来写我们自定义的中间件
每一个中间件都有具体的功能

回到顶部

三、自定义中间件

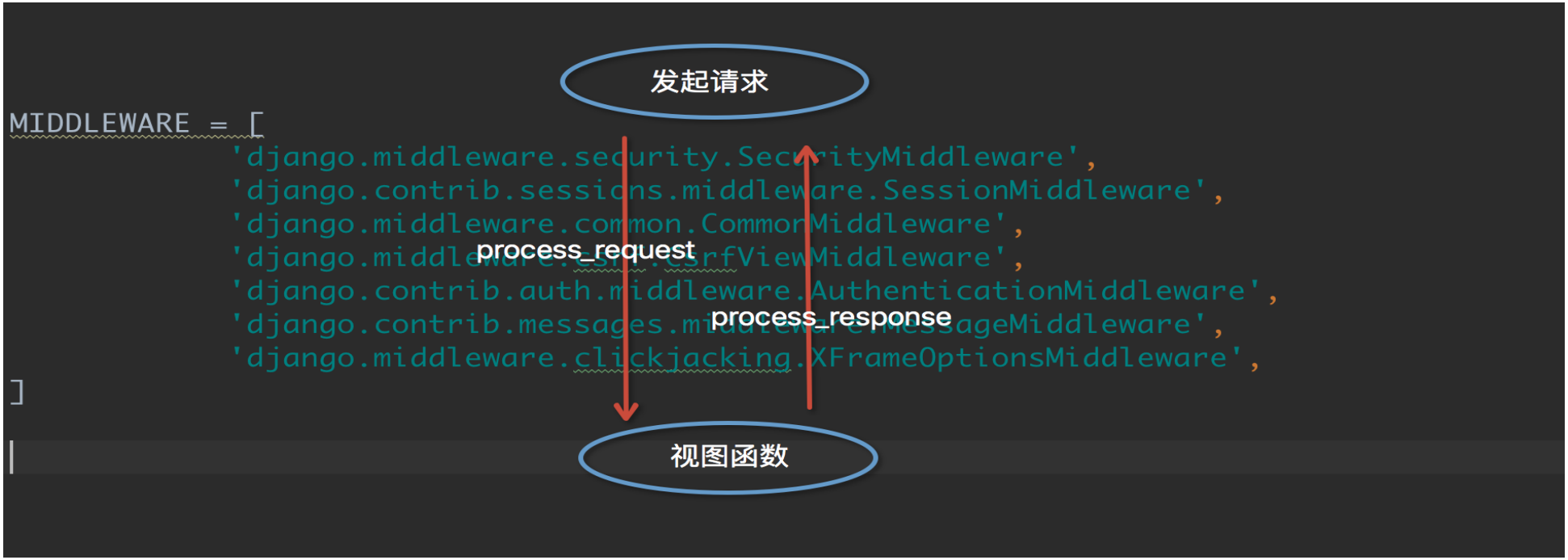
中间件可以定义五个方法，分别是：（主要的是 process_request 和 process_response）

```
1、process_request(self,request)
2、process_view(self, request, callback, callback_args, callback_kwargs)
3、process_template_response(self,request,response)
4、process_exception(self, request, exception)
5、process_response(self, request, response)
```

以上方法的返回值可以是 None 或一个 HttpResponse 对象，如果是 None，则继续按照 django 定义的规则向后继续执行，如果是 HttpResponse 对象，则直接将该对象返回给用户。
详细用法见下方：

1、process_request 和 process_response

当用户发起请求的时候会依次经过所有的的中间件，这个时候的请求时 process_request,最后到达 views 的函数中，views 函数处理后，在依次穿过中间件，这个时候是 process_response,最后返回给请求者。



上述截图中的中间件都是 django 中的，我们也可以自己定义一个中间件，我们可以自己写一个类，**但是必须继承 MiddlewareMixin**

第一步：导入

```
from django.utils.deprecation import MiddlewareMixin
```

第二步：自定义中间件

```
from django.utils.deprecation import MiddlewareMixin#
from django.shortcuts import HttpResponse
#
class Md1(MiddlewareMixin):
#
    def process_request(self,request):
        print("Md1 请求")
 #
    def process_response(self,request,response):
        print("Md1 返回")
        return response
#
class Md2(MiddlewareMixin):
```

```
#
    def process_request(self,request):
        print("Md2 请求")
        #return HttpResponse("Md2 中断")
    def process_response(self,request,response):#
        print("Md2 返回")
        return response
```

第三步：在 view 中定义一个视图函数（index）

```
def index(request):

    print("view 函数...")
    return HttpResponse("OK")
```

第四步：在 settings.py 的 MIDDLEWARE 里注册自己定义的中间件

运行结果：

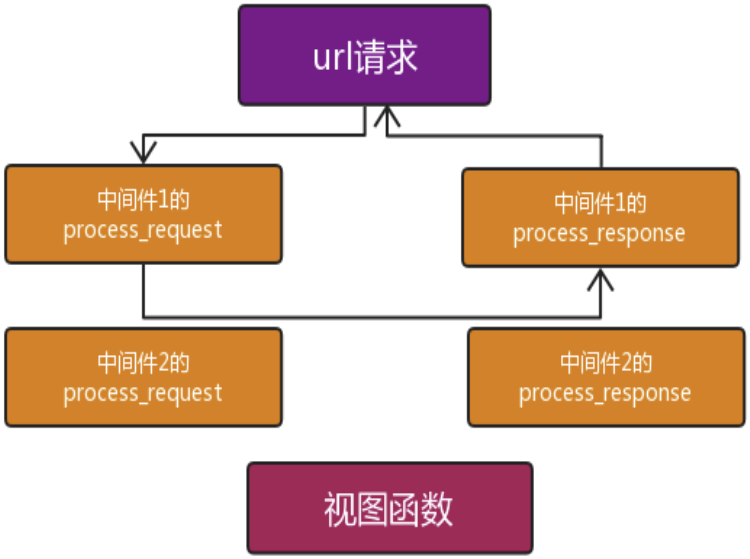
```
Md1 请求#
Md2 请求#
view 函数...#
Md2 返回#
Md1 返回#
```

注意：如果当请求到达请求 2 的时候直接不符合条件返回，即 return HttpResponse("Md2 中断")，程序将把请求直接发给中间件 2 返回，然后依次返回到请求者，结果如下：

返回 Md2 中断的页面，后台打印如下：

```
Md1 请求#
Md2 请求#
Md2 返回#
Md1 返回#
```

流程图如下：



由此总结一下：

1. 中间件的 process_request 方法是在执行视图函数之前执行的。
2. 当配置多个中间件时，会按照 MIDDLEWARE 中的注册顺序，也就是列表的索引值，从前到后依次执行的。
3. 不同中间件之间传递的 request 都是同一个对象

   多个中间件中的 process_response 方法是按照 MIDDLEWARE 中的注册顺序**倒序**执行的，也就是说第一个中间件的 process_request 方法首先执行，而它的 process_response 方法最后执行，最后一个中间件的 process_request 方法最后一个执行，它的 process_response 方法是最先执行。

2、process_view

process_view(self, request, view_func, view_args, view_kwargs)

该方法有四个参数

request 是 HttpRequest 对象。
view_func 是 Django 即将使用的视图函数。 （它是实际的函数对象，而不是函数的名称作为字符串。）
view_args 是将传递给视图的位置参数的列表（无名分组分过来的值）.
view_kwargs 是将传递给视图的关键字参数的字典（有名分组分过来的值）。 view_args 和 view_kwargs 都不包含第一个视图参数（request）。
Django 会在调用视图函数之前调用 process_view 方法。
它应该返回 None 或一个 HttpResponse 对象。 如果返回 None，Django 将继续处理这个请求，执行任何其他中间件的 process_view 方法，然后在执行相应的视图。 如果它返回一个 HttpResponse 对象，Django 不会调用适当的视图函数。 它将执行中间件的 process_response 方法并将应用到该 HttpResponse 并返回结果。

```
process_view(self, request, callback, callback_args, callback_kwargs)
```

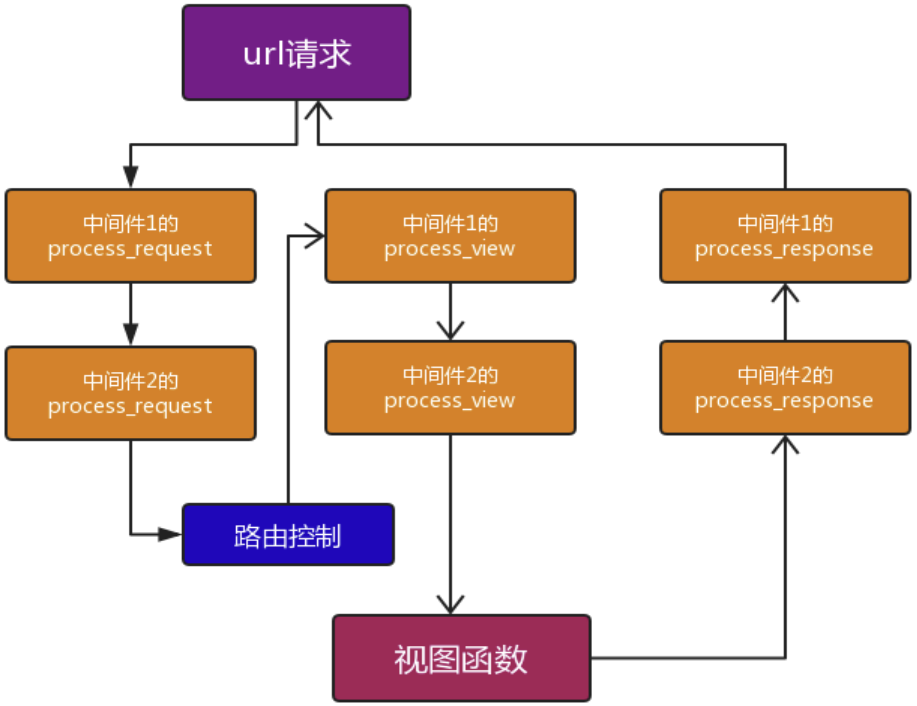
```
from django.utils.deprecation import MiddlewareMixin
from django.shortcuts import HttpResponse
class Md1(MiddlewareMixin):
    def process_request(self,request):
        print("Md1 请求")
        #return HttpResponse("Md1 中断")
    def process_response(self,request,response):
        print("Md1 返回")
        return response
    def process_view(self, request, callback, callback_args, callback_kwargs):
        print("Md1view")
class Md2(MiddlewareMixin):
    def process_request(self,request):
        print("Md2 请求")
        return HttpResponse("Md2 中断")
    def process_response(self,request,response):
        print("Md2 返回")
        return response
    def process_view(self, request, callback, callback_args, callback_kwargs):
        print("Md2view")
```

运行结果：

```
Md1 请求#
Md2 请求#
Md1view#
```

```
Md2view#
view 函数...#
Md2 返回#
Md1 返回#
```

下图进行分析上面的过程：



当最后一个中间的 process_request 到达路由关系映射之后，返回到中间件 1 的 process_view，然后依次往下，到达 views 函数，最后通过 process_response 依次返回到达用户。

process_view 可以用来调用视图函数：

```
class Md1(MiddlewareMixin):
    def process_request(self,request):
        print("Md1 请求")
        #return HttpResponse("Md1 中断")
    def process_response(self,request,response):
```

```
        print("Md1 返回")
        return response
    def process_view(self, request, callback, callback_args, callback_kwargs):
        # return HttpResponse("hello")
        response=callback(request,*callback_args,**callback_kwargs)
        return response
```

运行结果：

```
Md1 请求#
Md2 请求#
view 函数...#
Md2 返回#
Md1 返回#
```

注意：process_view 如果有返回值，会越过其他的 process_view 以及视图函数，但是所有的 process_response 都还会执行。

3、process_exception

process_exception(self, request, exception)

该方法两个参数：

一个 HttpRequest 对象

一个 exception 是视图函数异常产生的 Exception 对象。

这个方法只有在视图函数中出现异常了才执行，它返回的值可以是一个 None 也可以是一个 HttpResponse 对象。如果是 HttpResponse 对象，Django 将调用模板和中间件中的 process_response 方法，并返回给浏览器，否则将默认处理异常。如果返回一个 None，则交给下一个中间件的 process_exception 方法来处理异常。它的执行顺序也是按照中间件注册顺序的倒序执行。

```
process_exception(self, request, exception)
```

实例如下：

```
class Md1(MiddlewareMixin):
    def process_request(self,request):
        print("Md1 请求")
        #return HttpResponse("Md1 中断")
    def process_response(self,request,response):
        print("Md1 返回")
        return response
    def process_view(self, request, callback, callback_args, callback_kwargs):
        # return HttpResponse("hello")
        # response=callback(request,*callback_args,**callback_kwargs)
        # return response
```

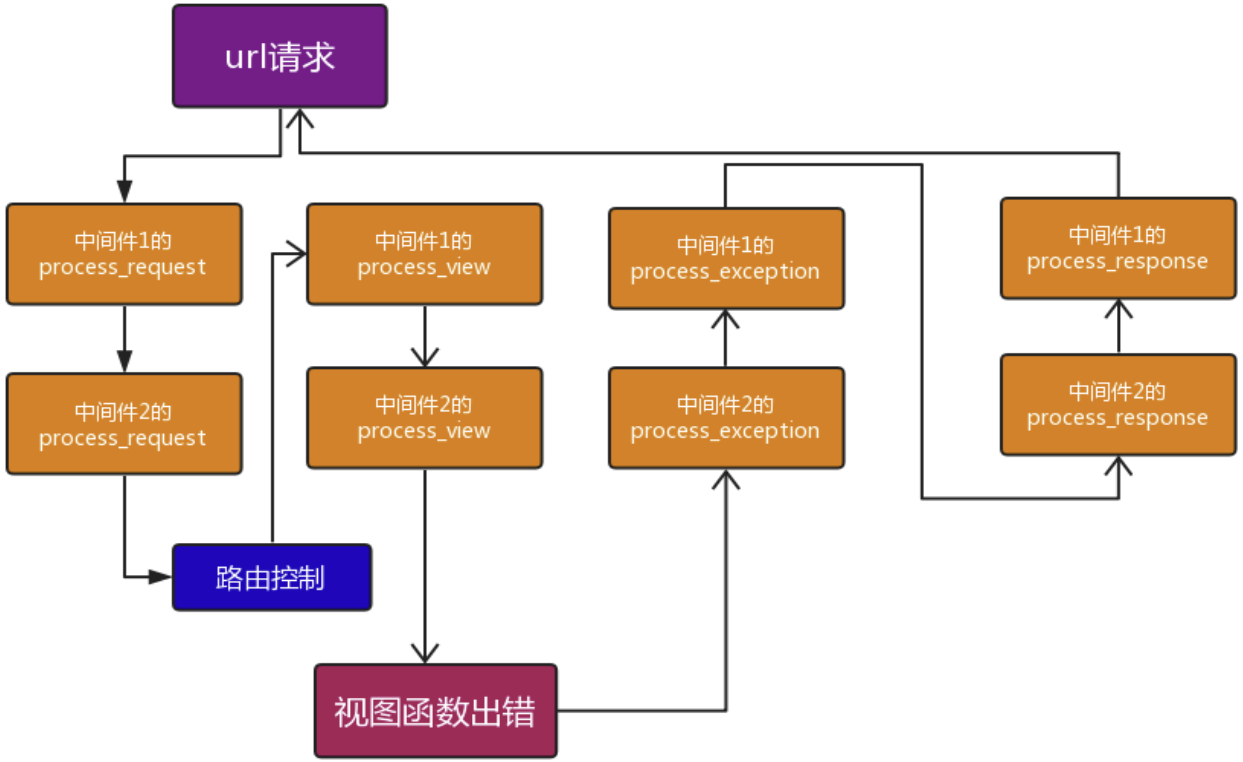
```
        print("md1 process_view...")
    def process_exception(self,request,exception):
        print("md1 process_exception...")
class Md2(MiddlewareMixin):
    def process_request(self,request):
        print("Md2 请求")
        # return HttpResponse("Md2 中断")
    def process_response(self,request,response):
        print("Md2 返回")
        return response
    def process_view(self, request, callback, callback_args, callback_kwargs):
        print("md2 process_view...")
    def process_exception(self,request,exception):#只有报错了才会执行 exception
        print("md1 process_exception...")
```

运行结果：

```
Md1 请求
Md2 请求
md1 process_view...
md2 process_view...
view 函数...
Md2 返回
Md1 返回
```

流程图如下：

当 views 出现错误时：



将 md2 的 process_exception 修改如下：

```
  def process_exception(self,request,exception):
        print("md2 process_exception...")
        return HttpResponse("error")
```

运行结果：

```
Md1 请求#
Md2 请求#
md1 process_view...#
md2 process_view...#
view 函数...#
md2 process_exception...#
Md2 返回#
Md1 返回#
```

4、process_template_response(self,request,response)

该方法对视图函数返回值有要求，必须是一个含有 render 方法类的对象，才会执行此方法

```
class Test:
    def __init__(self,status,msg):
        self.status=status#
        self.msg=msg#
    def render(self):#
        import json#
        dic={'status':self.status,'msg':self.msg}#
        return HttpResponse(json.dumps(dic))
def index(response):#
    return Test(True,'测试')
```

回到顶部

四 中间件应用场景

1、做 IP 访问频率限制

某些 IP 访问服务器的频率过高，进行拦截，比如限制每分钟不能超过 20 次。

2、URL 访问过滤

如果用户访问的是 login 视图（放过）

如果访问其他视图，需要检测是不是有 session 认证，已经有了放行，没有返回 login，这样就省得在多个视图函数上写装饰器了！

作为延伸扩展内容，有余力的同学可以尝试着读一下以下两个自带的中间件：

```
'django.contrib.sessions.middleware.SessionMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware',
```

回到顶部

五 CSRF_TOKEN 跨站请求伪造

csrf 介绍：

1、什么是 csrf

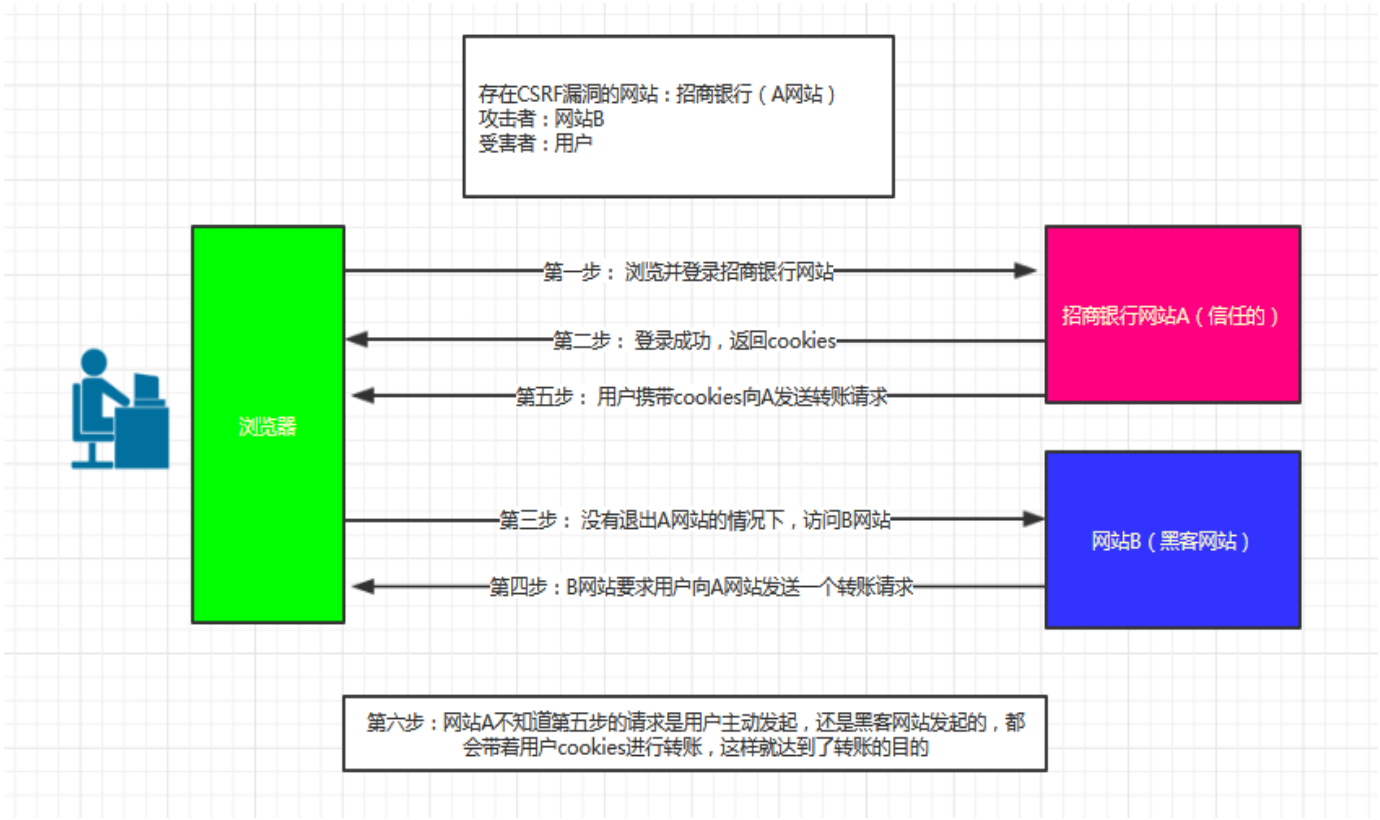
CSRF（Cross-site request forgery）跨站请求伪造，也被称为"One Click Attack"或者 Session Riding，通常缩写为 CSRF 或者 XSRF，是一种对网站的恶意利用。尽管听起来像跨站脚本（XSS），但它与 XSS 非常不同，XSS 利用站点内的信任用户，而 CSRF 则通过伪装来自受信任用户的请求来利用受信任的网站。与 XSS 攻击相比，CSRF 攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比 XSS 更具危险性

可以这样来理解：

攻击者盗用了你的身份，以你的名义发送恶意请求，对服务器来说这个请求是完全合法的，但是却完成了攻击者所期望的一个操作，比如以你的名义发送邮件、发消息，盗取你的账号，添加系统管理员，甚至于购买商品、虚拟货币转账等。 如下：其中 Web A 为存在 CSRF 漏洞的网站，Web B 为攻击者构建的恶意网站，User C 为 Web A 网站的合法用户

2、csrf 攻击原理

如下图：



从上图可以看出，要完成一次 CSRF 攻击，受害者必须依次完成两个步骤：

1.登录受信任网站 A，并在本地生成 Cookie。

2.在不登出 A 的情况下，访问危险网站 B。

看到这里，你也许会说："如果我不满足以上两个条件中的一个，我就不会受到 CSRF 的攻击"。是的，确实如此，但你不能保证以下情况不会发生：

1.你不能保证你登录了一个网站后，不再打开一个 tab 页面并访问另外的网站。

2.你不能保证你关闭浏览器了后，你本地的 Cookie 立刻过期，你上次的会话已经结束。（事实上，关闭浏览器不能结束一个会话，但大多数人都会错误的认为关闭浏览器就等于退出登录/结束会话了.....）

3.上图中所谓的攻击网站，可能是一个存在其他漏洞的可信任的经常被人访问的网站。

3、csrf 攻击防范

目前防御 CSRF 攻击主要有三种策略：验证 HTTP Referer 字段；在请求地址中添加 token 并验证；在 HTTP 头中自定义属性并验证

（1）验证 HTTP Referer 字段

根据 HTTP 协议，在 HTTP 头中有一个字段叫 Referer，它记录了该 HTTP 请求的来源地址。在通常情况下，访问一个安全受限页面的请求来自于同一个网

站，比如需要访问 http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory，用户必须先登陆 bank.example，然后通过点击页面上的按钮来触发转账事件。这时，该转帐请求的 Referer 值就会是转账按钮所在的页面的 URL，通常是以 bank.example 域名开头的地址。而如果黑客要对银行网站实施 CSRF 攻击，他只能在他自己的网站构造请求，当用户通过黑客的网站发送请求到银行时，该请求的 Referer 是指向黑客自己的网站。因此，要防御 CSRF 攻击，银行网站只需要对于每一个转账请求验证其 Referer 值，如果是以 bank.example 开头的域名，则说明该请求是来自银行网站自己的请求，是合法的。如果 Referer 是其他网站的话，则有可能是黑客的 CSRF 攻击，拒绝该请求。

这种方法的显而易见的好处就是简单易行，网站的普通开发人员不需要操心 CSRF 的漏洞，只需要在最后给所有安全敏感的请求统一增加一个拦截器来检查 Referer 的值就可以。特别是对于当前现有的系统，不需要改变当前系统的任何已有代码和逻辑，没有风险，非常便捷。

然而，这种方法并非万无一失。Referer 的值是由浏览器提供的，虽然 HTTP 协议上有明确的要求，但是每个浏览器对于 Referer 的具体实现可能有差别，并不能保证浏览器自身没有安全漏洞。使用验证 Referer 值的方法，就是把安全性都依赖于第三方（即浏览器）来保障，从理论上来讲，这样并不安全。事实上，对于某些浏览器，比如 IE6 或 FF2，目前已经有一些方法可以篡改 Referer 值。如果 bank.example 网站支持 IE6 浏览器，黑客完全可以把用户浏览器的 Referer 值设为以 bank.example 域名开头的地址，这样就可以通过验证，从而进行 CSRF 攻击。

即便是使用最新的浏览器，黑客无法篡改 Referer 值，这种方法仍然有问题。因为 Referer 值会记录下用户的访问来源，有些用户认为这样会侵犯到他们自己的隐私权，特别是有些组织担心 Referer 值会把组织内网中的某些信息泄露到外网中。因此，用户自己可以设置浏览器使其在发送请求时不再提供 Referer。当他们正常访问银行网站时，网站会因为请求没有 Referer 值而认为是 CSRF 攻击，拒绝合法用户的访问。

（2）在请求地址中添加 token 并验证

CSRF 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于 cookie 中，因此黑客可以在不知道这些验证信息的情况下直接利用用户自己的 cookie 来通过安全验证。要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 cookie 之中。可以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有 token 或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。

这种方法要比检查 Referer 要安全一些，token 可以在用户登陆后产生并放于 session 之中，然后在每次请求时把 token 从 session 中拿出，与请求中的 token 进行比对，但这种方法的难点在于如何把 token 以参数的形式加入请求。对于 GET 请求，token 将附在请求地址之后，这样 URL 就变成 http://url?csrftoken=tokenvalue。 而对于 POST 请求来说，要在 form 的最后加上 <input type=”hidden” name=”csrftoken” value=”tokenvalue”/>，这样就把 token 以参数的形式加入请求了。但是，在一个网站中，可以接受请求的地方非常多，要对于每一个请求都加上 token 是很麻烦的，并且很容易漏掉，通常使用的方法就是在每次页面加载时，使用 javascript 遍历整个 dom 树，对于 dom 中所有的 a 和 form 标签后加入 token。这样可以解决大部分的请求，但是对于在页面加载之后动态生成的 html 代码，这种方法就没有作用，还需要程序员在编码时手动添加 token。

该方法还有一个缺点是难以保证 token 本身的安全。特别是在一些论坛之类支持用户自己发表内容的网站，黑客可以在上面发布自己个人网站的地址。由于系统也会在这个地址后面加上 token，黑客可以在自己的网站上得到这个 token，并马上就可以发动 CSRF 攻击。为了避免这一点，系统可以在添加 token 的时候增加一个判断，如果这个链接是链到自己本站的，就在后面添加 token，如果是通向外网则不加。不过，即使这个 csrftoken 不以参数的形式附加在请求之中，黑客的网站也同样可以通过 Referer 来得到这个 token 值以发动 CSRF 攻击。这也是一些用户喜欢手动关闭浏览器 Referer 功能的原因。

（3）在 HTTP 头中自定义属性并验证

这种方法也是使用 token 并进行验证，和上一种方法不同的是，这里并不是把 token 以参数的形式置于 HTTP 请求之中，而是把它放到 HTTP 头中自定义的属性里。通过 XMLHttpRequest 这个类，可以一次性给所有该类请求加上 csrftoken 这个 HTTP 头属性，并把 token 值放入其中。这样解决了上种方法在请求中加入 token 的不便，同时，通过 XMLHttpRequest 请求的地址不会被记录到浏览器的地址栏，也不用担心 token 会透过 Referer 泄露到其他网站中去。

在 form 表单中应用：

```
<form action="" method="post">
    {% csrf_token %}
    <p>用户名：<input type="text" name="name"></p>
    <p>密码：<input type="text" name="password"></p>
    <p><input type="submit"></p>
</form>
```

在 Ajax 中应用：

放在 data 里：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script src="/static/jquery-3.3.1.js"></script>
    <title>Title</title>
</head>
<body>
<form action="" method="post">
    {% csrf_token %}
    <p>用户名：<input type="text" name="name"></p>
    <p>密码：<input type="text" name="password" id="pwd"></p>
    <p><input type="submit"></p>
</form>
<button class="btn">点我</button>
</body>
<script>
    $(".btn").click(function () {
        $.ajax({
            url: '',
            type: 'post',
            data: {
                'name': $('[name="name"]').val(),
                'password': $("#pwd").val(),
                'csrfmiddlewaretoken': $('[name="csrfmiddlewaretoken"]').val()
```

```
            },
            success: function (data) {
                console.log(data)
            }

        })
    })
</script>
</html>
```

放在 cookie 里：
获取 cookie：document.cookie
是一个字符串，可以自己用 js 切割，也可以用 jquery 的插件
获取 cookie：$.cookie('csrftoken')
设置 cookie：$.cookie('key','value')

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script src="/static/jquery-3.3.1.js"></script>
    <script src="/static/jquery.cookie.js"></script>
    <title>Title</title>
</head>
<body>
<form action="" method="post">
    {% csrf_token %}
    <p>用户名：<input type="text" name="name"></p>
    <p>密码：<input type="text" name="password" id="pwd"></p>
    <p><input type="submit"></p>
</form>
<button class="btn">点我</button>
</body>
<script>
    $(".btn").click(function () {
        var token=$.cookie('csrftoken')
        //var token='{{ csrf_token }}'
```

```
        $.ajax({
            url: '',
            headers:{'X-CSRFToken':token},
            type: 'post',
            data: {
                'name': $('[name="name"]').val(),
                'password': $("#pwd").val(),
            },
            success: function (data) {
                console.log(data)
            }

        })
    })
</script>
</html>
```

其它操作

全站禁用：注释掉中间件 'django.middleware.csrf.CsrfViewMiddleware',

局部禁用：用装饰器（在 FBV 中使用）

```
from django.views.decorators.csrf import csrf_exempt,csrf_protect
# 不再检测，局部禁用（前提是全站使用）
# @csrf_exempt
# 检测，局部使用（前提是全站禁用）
# @csrf_protect
def csrf_token(request):#
    if request.method=='POST':
        print(request.POST)

        return HttpResponse('ok')
    return render(request,'csrf_token.html')#
```

在 CBV 中使用：

```
# CBV 中使用
from django.views import View
from django.views.decorators.csrf import csrf_exempt,csrf_protect
from django.utils.decorators import method_decorator
```

```
# CBV 的 csrf 装饰器，只能加载类上（django 的 bug）
# 给 get 方法使用 csrf_token 检测
@method_decorator(csrf_exempt,name='get'), #不能直接放在函数上，可以放在分发函数 dispatch 上不需要指定名字，是什么请求就会分发到指定的函数上
# 给 post 加
@method_decorator(csrf_exempt,name='post')#
# 给所有方法加
@method_decorator(csrf_exempt,name='get')#
class Foo(View):
    def get(self,request):#
        pass
    def post(self,request):#
        pass
```

Django Drf 到底是什么东西

在序列化与反序列化时，虽然操作的数据不尽相同，但是执行的过程却是相似的，也就是说这部分代码是可以复用简化编写的。

在开发 REST API 的视图中，虽然每个视图具体操作的数据不同，但增、删、改、查的实现流程基本套路化，所以这部分代码也是可以复用简化编写的：

增：校验请求数据 -> 执行反序列化过程 -> 保存数据库 -> 将保存的对象序列化并返回

删：判断要删除的数据是否存在 -> 执行数据库删除

改：判断要修改的数据是否存在 -> 校验请求的数据 -> 执行反序列化过程 -> 保存数据库 -> 将保存的对象序列化并返回

查：查询数据库 -> 将数据序列化并返回

Django REST framework 可以帮助我们简化上述两部分的代码编写，大大提高 REST API 的开发速度。

认识 Django REST framework

Django REST framework 框架是一个用于构建 Web API 的强大而又灵活的工具。

通常简称为 DRF 框架 或 REST framework。

DRF 框架是建立在 Django 框架基础之上，由 Tom Christie 大牛二次开发的开源项目。

特点

提供了定义序列化器 Serializer 的方法，可以快速根据 Django ORM 或者其它库自动序列化/反序列化；

提供了丰富的类视图、Mixin 扩展类，简化视图的编写；

丰富的定制层级：函数视图、类视图、视图集合到自动生成 API，满足各种需要；

多种身份认证和权限认证方式的支持；

内置了限流系统；

直观的 API web 界面；
可扩展性，插件丰富

Django&DRF 重点内容大盘点

本文只是将学习过程中需要深刻记忆，在工作中常用的一些命令或者知识点进行一个罗列并阐释，不会全面的将所有内容进行讲解。大家可以在了解了 Django 框架和 DRF 框架之后再来看这边文章。否则会有点不知所云。

1.Django

1.1 创建 Django 项目

这一命令必须熟记于心：

```
django-admin startproject 项目名
```

1.2 创建子应用

1.在工作中我们要开发很多项目，肯定需要很多模块，创建子应用肯定也需要掌握：

```
python manage.py startapp 子应用名
```

注意：此命令需要在项目的目录下进行输入。

2.创建完子应用中之后，千万不要忘记去 INSTALLED_APPS 中进行注册，这个参数在 setting 文件中。

1.3 一个程序注意的点

1.3.1 视图函数的定义

1）定义视图函数之后，要有一个 request 形参接收请求对象。

2）返回的时候用到了 HttpResponse 这一命令返回响应对象

1.3.2url 地址的配置

1.在子应用中的 urls.py 文件中设置当前子应用中 url 地址和视图对应关系

```
urlpatterns = [
    url(r'^url 正则表达式$',views.视图函数名)
]
```

2.在项目总的 urls.py 文件中包含子应用中的 urls.py 文件

```
urlpatterns = [
    url(r'^',include('users.urls'))
]
```

1.4url 配置

在子应用中进行 url 地址的配置时，建议严格匹配开头和结尾，避免在地址匹配时候出错。

1.5 项目配置项

1.BASE_DIR 指的是 Django 项目根目录

2.语言和时区本地化：

```
LANGUAGE_CODE = 'zh-Hans' # 中文语言
TIME_ZONE = 'Asia/Shanghai' # 中国时间
```

1.6 客户端向服务器传递参数途径

1.6.1 通过 URL 地址传递参数

在我们的 url 地址中的参数，我们如果想要获取可以在子应用中的 urls 文件中进行设置

/weather/城市/年份

```
url(r'weather/(?P<city>\w+)/(?P<year>\d{4})/$'views.weather)
```

1.6.2 通过查询字符串传递参数

```
# /qs/?a=1&b=2&a=3
def qs(request):
    a = request.GET.get('a')
    b = request.GET.get('b')
    alist = request.GET.getlist('a')
    print(a)  # 3
    print(b)  # 2
    print(alist)  # ['1', '3']
    return HttpResponse('OK')
```

重要：查询字符串不区分请求方式，即假使客户端进行 POST 方式的请求，依然可以通过 request.GET 获取请求中的查询字符串数据。

1.6.3 通过请求体传递数据

1)post 表单提交的数据

/form/

```
def form_data(request):
    name = request.POST.get('name')
    age = request.POST.get('age')
    return HttpResponse('OK')
```

2)json 数据

/json/

```
def json_data(request):
    req_data = request.body
    json_str = req_data.decode()
    req_dict = json.loads(json_str)
    name = req_dict.get('name')
    age = req_dict.get('age')
    return HttpResponse('OK')
```

1.6.4 通过请求头传递数据（了解即可）

1.6.5request 对象的属性

request 请求对象的属性	说明
GET	查询字符串参数
POST	请求体重的表单数据
body	请求体中原始的 bytes 数据
method	请求方式
path	请求的 url 路径
META	请求头
COOKIES	客户端发送的 cookie 信息
FILES	客户端上传的文件

1.7 相应对象构造

1.7.1 响应时返回 json 数据

```
def get_json(request):
    res_dict = {
        'name':'xiaoyan',
        'age':'18'
    }
    return JsonResponse(res_dict)
```

1.7.2 响应时进行页面重定向

```
def redirect_test(request):
    # 第一个参数是 namespace，第二个参数是 name
    req_url = reverse('users:index')
    return redirect(req_url)
```

1.8 状态保持之 session

1）将 session 信息保存到 redis 中

```
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
        }
    }
}
```

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
SESSION_CACHE_ALIAS = "default"
```
2)设置 session
```
request.session['<key>'] = '<value>'
```
3)获取 session
```
request.session.get('<key>')
```
1.9 类视图
1.9.1 类视图的使用
1）定义类视图
/register/
```
class RegisterView(View):
    def get(self,request):
        return HttpResponse('返回注册页面')
    def post(self,request):
        return HttpResponse('进行注册处理...')
    def put(self,request):
        return HttpResponse('put 方法被调用')
```
2）进行 url 配置
```
url(r'^register/$',views.RegisterView.as_view())
```
1.9.2 类视图添加装饰器
使用 Django 框架提供 method_decorator 将针对函数视图装饰器添加到类视图的方法上面
```
# 为全部请求方法添加装饰器
@method_decorator(my_decorator, name='dispatch')
class DemoView(View):
    def get(self, request):
        print('get 方法')
        return HttpResponse('ok')
    def post(self, request):
        print('post 方法')
        return HttpResponse('ok')

# 为特定请求方法添加装饰器
@method_decorator(my_decorator, name='get')
class DemoView(View):
```

```
    def get(self, request):
        print('get 方法')
        return HttpResponse('ok')

    def post(self, request):
        print('post 方法')
        return HttpResponse('ok')
```

1.10 中间件

1.10.1 定义中间件

```
def simple_middleware(get_response):
    # 此处编写的代码仅在 Django 第一次配置和初始化的时候执行一次。

    def middleware(request):
        # 此处编写的代码会在每个请求处理视图前被调用。

        response = get_response(request)

        # 此处编写的代码会在每个请求处理视图之后被调用。

        return response

    return middleware
```

1.10.2 在 MIDDLEWARE 中注册中间

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    # 'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'users.middleware.my_middleware',  # 添加中间件
]
```

1.10.3 注意：中间件是全局的

1.11 使用模板的详细步骤

1.加载模板：指定使用模板文件，获取模板对象

```
from django.template import loader
temp = loader.get_template('模板文件名')
```

2.模板渲染：给模板文件传递变量，将模板文件中的变量进行替换，获取替换之后的 html 内容

```
res_html = temp.render(字典)
```

3.创建响应对象

```
return HttpResponse(res_html)
```

1.12 数据库

1.12.1 数据库链接配置

1.在 settings.py 进行配置

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'HOST': '127.0.0.1',  # 数据库主机
        'PORT': 3306,  # 数据库端口
        'USER': 'root',  # 数据库用户名
        'PASSWORD': 'mysql',  # 数据库用户密码
        'NAME': 'django_demo'  # 数据库名字
    }
}
```

2.首次启动时需要在项目同名的目录 **init**.py 添加

```
pip install pymysql
import pymysql
pymysql.install_as_MySQLdb
```

1.12.2 定义模型类

```
class 模型类名(models.Model):
    # 字段名 = models.字段类型(选项参数)
    # ...

    class Meta:
        db_table = '<表名>'
```

定义外键属性

```
  hbook = models.ForeignKey(BookInfo, on_delete=models.CASCADE, verbose_name='图书')  # 外键
```

cascade 是级联，删除主表数据时连同外键表中数据一起删除

1.12.3 迁移生成表

1）生成迁移文件

```
python manage.py makemigrations
```

2）同步到数据库中

```
python manage.py migrate
```

1.12.4 通过模型类和对象进行数据库操作（增删改查）

新增：

1. 创建模型类对象-->对象.save()

注意：添加 HeroInfo 时，可以给 hbook 赋值，也可以直接表中 hbook_id 赋值

HeroInfo 是定义的英雄模型类，与图书 BookInfo 对应。

2. 模型类.objects.create(...)

修改：

1. 查询对象->修改对象属性->对象.save()
2. 模型类.objects.filter(...).update(...)

删除：

1. 查询对象->对象.delete()
2. 模型类.objects.filter(...).delete()

查询：

基本查询

模型类.objects.查询函数

条件查询

对应 get, filter, exclude 参数中可以写查询条件

格式: 属性名__条件名=值

注意：可以写多个查询条件，默认是且的关系

F 对象

用于查询时字段之间的比较

```
from django.db.models import F
```

Q 对象

用于查询时条件之间的逻辑关系

```
from django.db.models import Q
```

&(与) |(或) ~(非)

聚合

聚合类: from django.db.models import Count, Sum, Avg, Max, Min

aggregate

排序

排序默认是升序，降序在排序字段前加-

order_by

关联查询

1. 查询和指定对象关联的数据

由 1 查多

一对象.多类名小写_set.all()

例：book.heroinfo_set.all()

由多查 1

多对象.外键属性

例：hero.hbook

2. 通过模型类进行关联查询

查图书(一)

一类.objects.get|filter(多类名__字段__条件=值)

例：books = BookInfo.objects.filter(heroinfo__hcomment__contains='八')

查英雄(多)

多类.objects.filter(外键属性__字段__条件=值)

例：heros = HeroInfo.objects.filter(hbook__bread__gt=30)

1.13admin 站点

上传图片

Django 自带文件存储系统，可以直接通过 Admin 站点进行图片的上传，默认上传的文件保存在服务器本地。

使用

1）在配置文件中设置配置项 MEDIA_ROOT='上传文件的保存目录'

2）定义模型类时，图片字段的类型使用 ImageField

3）迁移生成表并在 admin.py 注册模型类，直接登录 Admin 站点就可以进行图片上传

2.DRF 框架

2.1 目的

利用 DRF 框架快速的实现 RestAPI 接口的设计

2.2RestfulAPI 接口设计风格

关键点

1）url 地址尽量使用名词，不要使用动词

2）请求 url 地址采用不同的请求方式执行不同的操作

GET(获取)

POST(新增)
PUT(修改)
DELETE(删除)
3）过滤参数可以放在查询字符串中
4）响应数据返回&响应状态码

状态码	说明
200	获取或修改成功
201	新增成功
204	删除成功
404	资源不存在
400	客户请求有误
500	服务器错误

5）响应数据的格式：json

2.3django 自定义 RestAPI 接口

RestAPI 接口核心工作
1.将数据库数据序列化为前端所需要的格式，并返回
2.将前端发送的数据反序列化为模型类对象，并保存到数据库中

2.4DRF 框架

2.4.1 作用：大大提高 RestAPI 接口开发效率

2.5 序列化器 Serializer

2.5.1 功能

进行数据的序列化和反序列化

2.5.2 序列化器的定义

继承自 serializers.Serializer

```
from rest_framework import serializers
# serializers.Serializer:DRF 框架中所有序列化器的父类，定义序列化器类时，可以直接继承此类
# serializers.ModelSerializer:Serializer 类的子类，在父类的基础山，添加一些功能
class 序列化器类名(serializers.Serializer):
    # 字段名 = serializers.字段名(选项参数)
```

序列化器对象创建：
序列化器类(instance=<实例对象>,data=<数据>,**kwargs)

2.5.3 序列化功能

说白了就是将实例对象转换为字典数据
1）序列化单个对象
```
book = BookInfo.objects.get(id=1)
serializer = BookInfoSerializer(book)
res = json.dumps(serializer.data,ensure_ascii=False,indent=4)
```
2）序列化多个对象，其实就是添加了一个 many 参数
```
books = BookInfo.objects.all()
serializer = BookInfoSerializer(books,many=True)
res = json.dumps(serializer.data,ensure_ascii=False,indent=4)
```
3）关联对象的嵌套序列化
1.将关联对象序列化为关联对象的主键
```
hbook = serializers.PrimaryKeyRelatedField(label='图书', read_only=True)
```
2.采用指定的序列化器将关联对象进行序列化
```
hbook = BookInfoSerializer(label='图书')
```
3.将关联对象序列化为关联对象模型类_*str*_方法的返回值
```
hbook = serializers.StringRelatedField(label='图书')
```
注意：和对象关联的对象如果有多个，在序列化器中定义嵌套序列化字段时，需要添加 many=True。

2.5.4 反序列化功能

反序列化-数据校验：
```
data = {'btitle':'python','bpub_date':'1980-1-1'}
serializer = BookInfoSerializer(data)
serializer.is_valid()
serializer.errors
serialiser.validated_data
```
当系统提供的校验不能满足我们的需求的时候，我们可以补充额外的验证：
1）指定特定字段的 validators 参数进行补充验证
```
btitle = serializers.CharField(label='标题',max_length=20,validators=[about_django])
```
注意：此处的 about_django 为我们自定义的校验函数
2）在序列化器类中定义特定方法 validate_<fieldname>针对特定字段进行补充验证
```
def validate_btitle(self,value):
    if 'django' not in value.lower():
        raise serializers.validationError('图书不是关于 Django 的')
    return value
```
3）定义 validate 方法进行补充验证（结合多个字段内容验证）
```
def validate(self,attrs):
```

```
    """
    此处的attrs是一个字典，创建序列化器对象时，传入data数据
    """
    bread = attrs['bread']
    bcomment = attrs['bcomment']
    if bread<=bcomment:
        raise serializers.ValidationError('图书阅读量必须大于评论量')
    return attrs
```

2.5.5 反序列化-数据保存（新增&更新）

校验通过之后，可以调用serializer.save()进行数据保存

1)数据新增

```
def create(self,validated_data):
    """
    validated_data:字典，校验之后的数据
    """
    book = BookInfo.objects.create(**validated_data)
    return book
-----------------------------------
data = {'btitle':'python','bpub_data':'1802-1-1','bread':30,'bcomment:20'}
serializer = BookInfoSerializer(data=data)
serializer.is_valid()
serializer.save()
serializer.data
```

2)数据更新

```
def update(self,instance,validated_data):
    """
    instance:创建序列化器对象时传入实例对象
    validated_data:是一个字典，校验之后的数据
    """
    btitle = validated_data.get('btitle')
    bpub_data_date = validated_data.get('bpub_date')

    instance.btitle = btitle
    instance.bpub_date = bpub_date
    instance.save()
```

```
    return instance
```

2.5.6 使用序列化器改写 RestAPI 接口

我们举两个代表性的例子即可：

```
# /books/
class BookListView(View):
    def get(self, request):
        """
        获取所有的图书的信息：
        1. 查询所有的图书的数据
        2. 返回所有图书的 json 的数据
        """
        # 1. 查询所有的图书的数据
        books = BookInfo.objects.all() # QuerySet
        # 2. 返回所有图书的 json 的数据，状态码: 200
        books_li = []
        for book in books:
            # 将 book 对象转换成 dict
            book_dict = {
                'id': book.id,
                'btitle': book.btitle,
                'bpub_date': book.bpub_date,
                'bread': book.bread,
                'bcomment': book.bcomment,
                'image': book.image.url if book.image else ''
            }
            books_li.append(book_dict)
        return JsonResponse(books_li, safe=False)
    def post(self, request):
        """
        新增一本图书的信息：
        1. 获取参数 btitle 和 bpub_date 并进行校验
        2. 创建图书信息并添加进数据表中
        3. 返回新增的图书的 json 数据，状态码: 201
        """
```

```
        # 需求: 前端需要传递新增图书的信息(btitle, bpub_date)，通过 json 传递
        # 1. 获取参数 btitle 和 bpub_date 并进行校验
        # 获取 json 的原始数据
        req_data = request.body # bytes
        # 将 bytes 转换为 str
        json_str = req_data.decode()
        # 将 json 字符串转换 dict
        req_dict = json.loads(json_str)
        # 获取 btitle 和 bpub_date
        btitle = req_dict.get('btitle')
        bpub_date = req_dict.get('bpub_date')
        # TODO: 省略参数校验过程...
        # 2. 创建图书信息并添加进数据表中
        book = BookInfo.objects.create(
            btitle=btitle,
            bpub_date=bpub_date
        )
        # 3. 返回新增的图书的 json 数据，状态码: 201
        # 将 book 对象转换成 dict
        book_dict = {
            'id': book.id,
            'btitle': book.btitle,
            'bpub_date': book.bpub_date,
            'bread': book.bread,
            'bcomment': book.bcomment,
            'image': book.image.url if book.image else ''
        }
        return JsonResponse(book_dict, status=201)
```

修改之后：

```
class BookListView(View):
    def get(self, request):
        books = BookInfo.objects.all() # QuerySet
        serializer = BookInfoSerializer(books, many=True)
        return JsonResponse(serializer.data, safe=False)
    def post(self, request):
```

```
req_data = request.body # bytes
json_str = req_data.decode()
req_dict = json.loads(json_str)
serializer = BookInfoSerializer(data=req_dict)
serializer.is_valid(raise_exception=True)
# 反序列化-数据保存(新增) -> create
serializer.save()
return JsonResponse(serializer.data, status=201)
```

Python 并行分布式框架 Celery

生产者消费者模式

在实际的软件开发过程中，经常会碰到如下场景：某个模块负责产生数据，这些数据由另一个模块来负责处理（此处的模块是广义的，可以是类、函数、线程、进程等）。产生数据的模块，就形象地称为生产者；而处理数据的模块，就称为消费者。

单单抽象出生产者和消费者，还够不上是生产者消费者模式。该模式还需要有一个缓冲区处于生产者和消费者之间，作为一个中介。生产者把数据放入缓冲区，而消费者从缓冲区取出数据，如下图所示：

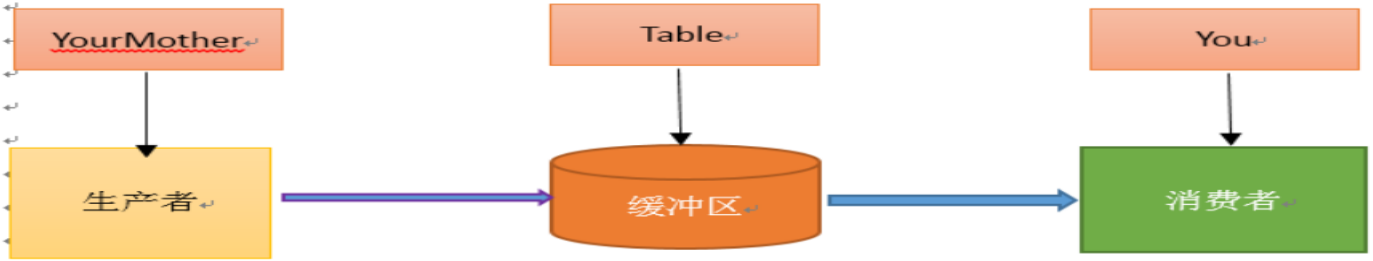
生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过消息队列（缓冲区）来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给消息队列，消费者不找生产者要数据，而是直接从消息队列里取，消息队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个消息队列就是用来给生产者和消费者解耦的。---------->这里又有一个问题，什么叫做解耦？

解耦：假设生产者和消费者分别是两个类。如果让生产者直接调用消费者的某个方法，那么生产者对于消费者就会产生依赖（也就是耦合）。将来如果消费者的代码发生变化，可能会影响到生产者。而如果两者都依赖于某个缓冲区，两者之间不直接依赖，耦合也就相应降低了。生产者直接调用消费者的某个方法，还有另一个弊端。由于函数调用是同步的（或者叫阻塞的），在消费者的方法没有返回之前，生产者只好一直等在那边。万一消费者处理数据很慢，生产者就会白白糟蹋大好时光。缓冲区还有另一个好处。如果制造数据的速度时快时慢，缓冲区的好处就体现出来了。当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。等生产者的制造速度慢下来，消费者再慢慢处理掉。

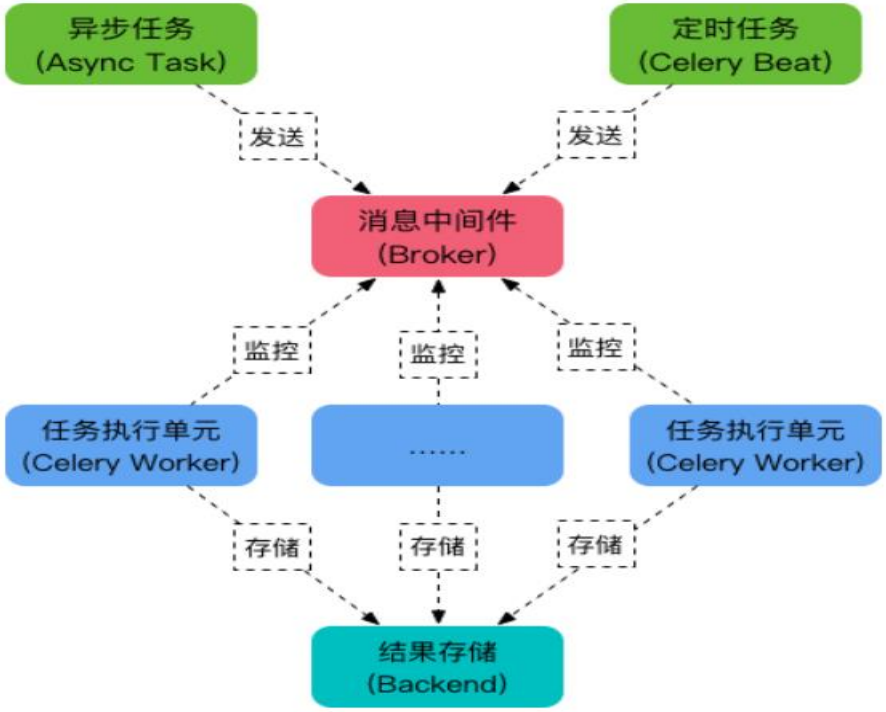
因为太抽象，看过网上的说明之后，通过我的理解，我举了个例子：吃包子。

假如你非常喜欢吃包子（吃起来根本停不下来），今天，你妈妈（生产者）在蒸包子，厨房有张桌子（缓冲区），你妈妈将蒸熟的包子盛在盘子（消息）里，然后放到桌子上，你正在看巴西奥运会，看到蒸熟的包子放在厨房桌子上的盘子里，你就把盘子取走，一边吃包子一边看奥运。在这个过程中，你和你妈妈使用同一个桌子放置盘子

和取走盘子，这里桌子就是一个共享对象。生产者添加食物，消费者取走食物。桌子的好处是，你妈妈不用直接把盘子给你，只是负责把包子装在盘子里放到桌子上，如果桌子满了，就不再放了，等待。而且生产者还有其他事情要做，消费者吃包子比较慢，生产者不能一直等消费者吃完包子把盘子放回去再去生产，因为吃包子的人有很多，如果这期间你好朋友来了，和你一起吃包子，生产者不用关注是哪个消费者去桌子上拿盘子，而消费者只去关注桌子上有没有放盘子，如果有，就端过来吃盘子中的包子，没有的话就等待。对应关系如下图：



考察了一下，原来当初设计这个模式，主要就是用来处理并发问题的，而 Celery 就是一个用 python 写的并行分布式框架。
然后我接着去学习 Celery
Celery 是一个强大的 分布式任务队列 的 异步处理框架，它可以让任务的执行完全脱离主程序，甚至可以被分配到其他主机上运行。我们通常使用它来实现异步任务（async task）和定时任务（crontab）。我们需要一个消息队列来下发我们的任务。首先要有一个消息中间件，此处选择 rabbitmq (也可选择 redis 或 Amazon Simple Queue Service(SQS)消息队列服务)。推荐 选择 rabbitmq 。使用 RabbitMQ 是官方特别推荐的方式，因此我也使用它作为我们的 broker。它的架构组成如下图：



通过 celery worker 启动的是启动消费者

通过 celery beat 是启动任务生产者

python 中使用 delay 生产任务

Celery 的定义

Celery（芹菜）是一个简单、灵活且可靠的，处理大量消息的分布式系统，并且提供维护这样一个系统的必需工具。
我比较喜欢的一点是：Celery 支持使用任务队列的方式在分布的机器、进程、线程上执行任务调度。然后我接着去理解什么是任务队列。
任务队列
任务队列是一种在线程或机器间分发任务的机制。
消息队列
消息队列的输入是工作的一个单元，称为任务，独立的职程（Worker）进程持续监视队列中是否有需要处理的新任务。
Celery 用消息通信，通常使用中间人（Broker）在客户端和职程间斡旋。这个过程从客户端向队列添加消息开始，之后中间人把消息派送给职程，职程对消息进行处理。
如下图所示：

Celery 系统可包含多个职程和中间人，以此获得高可用性和横向扩展能力。
Celery****的架构
Celery 的架构由三部分组成，消息中间件（message broker），任务执行单元（worker）和任务执行结果存储（task result store）组成。
消息中间件
Celery 本身不提供消息服务，但是可以方便的和第三方提供的消息中间件集成，包括，RabbitMQ,Redis,MongoDB 等，这里我先去了解 RabbitMQ,Redis。
linux 安装 redis 参考：https://blog.csdn.net/luanpeng825485697/article/details/81205424
docker 安装 redis 参考：https://blog.csdn.net/luanpeng825485697/article/details/81209596
docker 安装 rabbitmq 参考：https://blog.csdn.net/luanpeng825485697/article/details/82078416
任务执行单元

Worker 是 Celery 提供的任务执行的单元，worker 并发的运行在分布式的系统节点中
任务结果存储
Task result store 用来存储 Worker 执行的任务的结果，Celery 支持以不同方式存储任务的结果，包括 Redis，MongoDB，Django ORM，AMQP 等，这里我先不去看它是如何存储的，就先选用 Redis 来存储任务执行结果。
然后我接着去安装 Celery，在安装 Celery 之前，我已经在自己虚拟机上安装好了 Python，版本是 3.6，
安装 celery，版本为 4.2.1
sudo apt install python-celery-common
因为涉及到消息中间件，所以我先去选择一个在我工作中要用到的消息中间件（在 Celery 帮助文档中称呼为中间人<broker>），为了更好的去理解文档中的例子，我安装了两个中间件，一个是 RabbitMQ,一个 redis。
在这里我就先根据 Celery 的帮助文档安装和设置 RabbitMQ。要使用 Celery，我们需要创建一个 RabbitMQ 用户、一个虚拟主机，并且允许这个用户访问这个虚拟主机。下面是我在个人 pc 机 Ubuntu16.04 上的设置：
$ sudo rabbitmqctl add_user forward password
#创建了一个 RabbitMQ 用户,用户名为 forward，密码是 password
$ sudo rabbitmqctl add_vhost ubuntu
#创建了一个虚拟主机，主机名为 ubuntu
$ sudo rabbitmqctl set_permissions -p ubuntu forward ".*" ".*" ".*"
#允许用户 forward 访问虚拟主机 ubuntu，因为 RabbitMQ 通过主机名来与节点通信
$ sudo rabbitmq-server
之后我启用 RabbitMQ 服务器，结果如下，成功运行：

之后我安装 Redis,它的安装比较简单,如下：
$ sudo pip install redis
然后进行简单的配置，只需要设置 Redis 数据库的位置:
BROKER_URL = 'redis://localhost:6379/0'
URL 的格式为：
redis://:password@hostname**:port/db_number**
URL Scheme 后的所有字段都是可选的，并且默认为 localhost 的 6379 端口，使用数据库 0。我的配置是：
redis://:password@ubuntu**:6379/5**

之后安装 Celery，我是用标准的 Python 工具 pip 安装的，如下：

```
$ sudo pip install celery
```

celery 的命令

```
Global Options:
   -A APP, --app APP
   -b BROKER, --broker BROKER
   --result-backend RESULT_BACKEND
   --loader LOADER
   --config CONFIG
   --workdir WORKDIR
   --no-color, -C
   --quiet, -q

---- -- - - ---- Commands- ---------- --- ----------

+ Main:
|       celery worker
|       celery events
|       celery beat
|       celery shell
|       celery multi
|       celery amqp
```

开始使用 Celery

使用 celery 包含三个方面：1. 定义任务函数。2. 运行 celery 服务。3. 客户应用程序的调用。

注意：发送者和消费者职能处理启动前就定义好的任务，不能代码中现场定义。也就是说必须在定义任务后才能启动消费者。

定义

为了测试 Celery 能否工作，我运行了一个最简单的任务，编写 app1.py：

```
from celery import Celery
# broker 设置中间件，backend 设置后端存储
app = Celery('app_name',broker='redis://127.0.0.1:6379/5',backend='redis://127.0.0.1:6379/6')
@app.task(name='task_name')           #  被标记的都属于这个 app 的任务
def add(x,y):
      return x+y
```

编辑保存退出后，我在当前目录下运行如下命令(记得要先开启 redis)：

启动一个 worker

$ celery worker -A app1 --loglevel=info

其中 celery 会根据--app 后面的参数作为命令启动目录的相对目录，去找个这个相对目录文件中的 app 变量，并把所有用@app.task 修饰的函数作为任务来记录元数据。所以不同目录的启动命令是不一样的。

#查询文档，了解到该命令中-A 参数表示的是 Celery APP 的名称，这个实例中指的就是 app1.py（和文件名一致），后面的 app1 就是 APP 的名称，worker 是一个执行任务角色，后面的 loglevel=info 记录日志类型默认是 info,这个命令启动了一个 worker,用来执行程序中 add 这个加法任务（task）。

然后看到界面显示结果如下：

```
-------------- celery@luanpeng-XPS15R v4.2.1 (windowlicker)
---- **** -----
--- * ***  * -- Linux-4.15.0-33-generic-x86_64-with-Ubuntu-16.04-xenial 2018-09-24 19:19:11
-- * - **** ---
- ** ---------- [config]
- ** ---------- .> app:         tasks:0x7f34ba22dcc0
- ** ---------- .> transport:   redis://127.0.0.1:6379/5
- ** ---------- .> results:     disabled://
- *** --- * --- .> concurrency: 8 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=celery(direct) key=celery


[tasks]
  . tasks.add

[2018-09-24 19:19:11,186: INFO/MainProcess] Connected to redis://127.0.0.1:6379/5
[2018-09-24 19:19:11,193: INFO/MainProcess] mingle: searching for neighbors
[2018-09-24 19:19:12,212: INFO/MainProcess] mingle: all alone
[2018-09-24 19:19:12,224: INFO/MainProcess] celery@luanpeng-XPS15R ready.
```

我们可以看到 Celery 正常工作在名称 luanpeng-XPS15R 的虚拟主机上，版本为 v4.2.1，在下面的[config]中我们可以看到当前 APP 的名称 tasks，运输工具 transport 就是我们在程序中设置的中间人 redis://127.0.0.1:6379/5，result 我们没有设置，暂时显示为 disabled,然后我们也可以看到 worker 缺省使用 perfork 来执行并发，当前并发数显示为 1，然后可以看到下面的[queues]就是我们说的队列，当前默认的队列是 celery,然后我们看到下面的[tasks]中有一个任务 task_name.

生产者

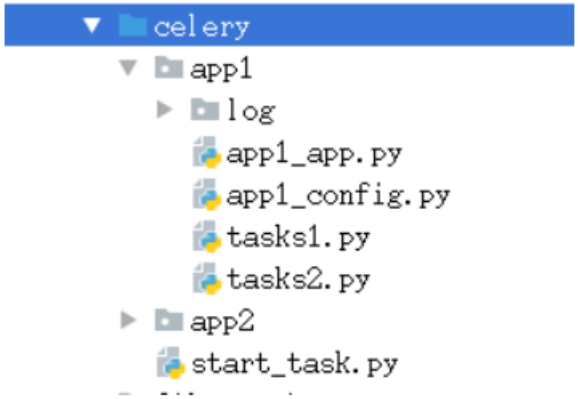
生产者发送消息启动。按照生产者所在目录引入 app 中的 task 函数（也会把 task 的名称获取到）。调用 delay 函数（apply_async()方法的升级版）时会发起这样一个 name 的 task 就可以了。
例如在上层目录发起任务

```
from app1.app1 import add
result = add.delay(1,2)     # apply_async()方法的升级版
print(result)
```

启动后，消费者开始处理消息

```
[2018-09-24 20:07:11,496: INFO/MainProcess] Received task: app1.tasks.add[8207c280-0864-4b1e-8792-155de5417406]
[2018-09-24 20:07:11,501: INFO/ForkPoolWorker-4] Task app1.tasks.add[8207c280-0864-4b1e-8792-155de5417406] succeeded in 0.003930353002942866s: 12
```

第一行说明 worker 收到了一个任务：app1.tasks.add,这里我们和之前发送任务返回的 AsyncResult 对比我们发现，每个 task 都有一个唯一的 ID，第二行说明了这个任务执行 succeed,执行结果为 12。
查看资料说调用任务后会返回一个 AsyncResult 实例，可用于检查任务的状态，等待任务完成或获取返回值（如果任务失败，则为异常和回溯）。但这个功能默认是不开启的，需要设置一个 Celery 的结果后端（backend），也就是 tasks.py 设置的使用 redis 进行结果存储。
通过这个例子后我对 Celery 有了初步的了解，然后我在这个例子的基础上去进一步的学习。
因为 Celery 是用 Python 编写的，所以为了让代码结构化一些，就像一个应用



app1/app1_app.py

```
from celery import Celery
import os,io
#  在 app1 目录同级目录执行 celery -A app1.app1_app worker -l info
app = Celery(main='app1.app1_app',include=['app1.tasks1','app1.tasks2'])  #  创建 app，并引入任务定义。main、include 参数的值为模块名，所以都是指定命令的相对目录
app.config_from_object('app1.app1_config')    #  通过配置文件进行配置，而着这里是相对目录
```

```
# broker 设置中间件，backend 设置后端存储
# app = Celery('app1.app1_app',broker='redis://127.0.0.1:6379/5',backend='redis://127.0.0.1:6379/6',include=['app1.tasks1','app1.task2'])
if __name__ == "__main__":
    log_path = os.getcwd()+'/log/celery.log'
    if(not os.path.exists(log_path)):
        f = open(log_path, 'w')
        f.close()
    # 在 app1 目录同级目录执行 celery -A app1.app1_app worker -l info
    app = Celery(main='app1_app',include=['tasks1', 'tasks2'])  # 创建 app，并引入任务定义。main、include 参数的值为模块名，所以都是指定命令的相对目录
    app.config_from_object('app1_config')  # 通过配置文件 c 进行配置，而着这里是相对目录
    # 使用下面的命令也可以启动 celery，不过要该模块的名称，是的相对目录正确
    app.start(argv=['celery', 'worker', '-l', 'info', '-f', 'log/celery.log', "-c", "40"])
```

一定要注意模块的相对目录，和你想要执行命令的目录

#首先创建了一个 celery 实例 app,实例化的过程中，制定了任务名(也就是包名.模块名)，Celery 的第一个参数是当前模块的名称，我们可以调用 config_from_object()来让 Celery 实例加载配置模块，我的例子中的配置文件起名为 app1_config.py,配置文件如下：

```
BROKER_URL = 'redis://127.0.0.1:6379/5'    # 配置 broker  中间件
CELERY_RESULT_BACKEND = 'redis://127.0.0.1:6379/6'    # 配置 backend 结果存储
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
```

在配置文件中我们可以对任务的执行等进行管理，比如说我们可能有很多的任务,但是我希望有些优先级比较高的任务先被执行,而不希望先进先出的等待。那么需要引入一个队列的问题. 也就是说在我的 broker 的消息存储里面有一些队列，他们并行运行，但是 worker 只从对应 的队列里面取任务。在这里我们希望 tasks.py 中的某些任务先被执行。task 中我设置了两个任务：

所以我通过 from celery import group 引入 group,用来创建并行执行的一组任务。然后这块现需要理解的就是这个@app.task,@符号在 python 中用作函数修饰符，到这块我又回头去看 python 的装饰器(在代码运行期间动态增加功能的方式)到底是如何实现的，在这里的作用就是通过 task()装饰器在可调用的对象（app）上创建一个任务。

tasks1.py

```
from app1.app1_app import app

@app.task
def deal1(text):
    print(text)
    return text+"======="
```

tasks2.py

```
from app1.app1_app import app
```

```
@app.task
def deal2(text):
    print(text)
    return text+"++++++++++"
```

队列

了解完装饰器后，我回过头去整理配置的问题，前面提到任务的优先级问题，在这个例子中如果我们想让 deal1 这个任务优先于 deal2 任务被执行，我们可以将两个任务放到不同的队列中，由我们决定先执行哪个任务，我们可以在配置文件 app1_config.py 中增加这样配置：

URL 的格式为：

redis://:password@hostname:port/db_number

```
from kombu import Exchange,Queue
BROKER_URL = 'redis://127.0.0.1:6379/5'     # 配置 broker   中间件
CELERY_RESULT_BACKEND = 'redis://127.0.0.1:6379/6'     # 配置 backend 结果存储
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
# (当使用 Redis 作为 broker 时，Exchange 的名字必须和 Queue 的名字一样)
CELERY_QUEUES = (
                    Queue("default", Exchange("default"), routing_key = "default"),
                    Queue("for_task1", Exchange("for_task1"), routing_key="task_a"),
                    Queue("for_task2", Exchange("for_task2"), routing_key="task_b")
)
# 定义任务的走向，不同的任务发送   进入不同的队列，并为不同的任务设定不同的 routing_key
# 若没有指定这个任务 route 到那个 Queue 中去执行，此时执行此任务的时候，会 route 到 Celery 默认的名字叫做 celery 的队列中去。
CELERY_ROUTES = {
    'app1.tasks1.deal1': {"queue": "for_task1", "routing_key": "task_a"},
    'app1.tasks2.deal2':{"queue":"for_task2","routing_key":"task_b"}
}
```

先了解了几个常用的参数的含义：

Exchange:交换机，决定了消息路由规则；

Queue:消息队列；

Channel:进行消息读写的通道；

Bind:绑定了 Queue 和 Exchange，意即为符合什么样路由规则的消息，将会放置入哪一个消息队列；

我将 deal1 这个函数任务放在了一个叫做 for_task1 的队列里面，将 deal2 这个函数任务放在了一个叫做 for_task2 的队列里面，然后我在当前应用目录下执行命令：

celery -A app1.app1_app worker -l info -Q for_task1

这个 worker 就只负责处理 for_task1 这个队列的任务，这是通过在启动 worker 是使用-Q Queue_Name 参数指定的。
我们定义任务调用文件 start_task.py

```
from __future__ import print_function
from app1.app1_app import app
if __name__=="__main__":
    for i in range(10):
        text = 'text'+str(i)
        app.send_task('app1.tasks1.deal1',args=[text])    # 任务的名称必须和 Celery 注册的任务名称相同
        app.send_task('app1.tasks2.deal2',args=[text])   # 任务的名称必须和 Celery 注册的任务名称相同
        print('push over %d'%i)
```

执行上述代码文件

```
python start_task.py
```

任务已经被执行，我在 worker 控制台查看结果（只有 app1.appa_app.deal1 任务被这个 worker 执行了）：

```
[2018-09-24 22:26:38,928: INFO/ForkPoolWorker-8] Task app1.tasks1.deal1[b3007993-9bfb-4161-b5b2-4f0f022f2f8b] succeeded in 0.0008255800021288451s:
'text4======='
[2018-09-24 22:26:38,928: INFO/ForkPoolWorker-6] Task app1.tasks1.deal1[df24b991-88fc-4253-86bf-540754c62da9] succeeded in 0.004320767002354842s:
'text3======='
[2018-09-24 22:26:38,929: INFO/MainProcess] Received task: app1.tasks1.deal1[dbdf9ac0-ea27-4455-90d2-e4fe8f3e895e]
[2018-09-24 22:26:38,930: WARNING/ForkPoolWorker-4] text5
[2018-09-24 22:26:38,931: INFO/ForkPoolWorker-4] Task app1.tasks1.deal1[dbdf9ac0-ea27-4455-90d2-e4fe8f3e895e] succeeded in 0.0006721289973938838s:
'text5======='
```

可以看到 worker 收到任务，并且执行了任务。

Scheduler (定时任务，周期性任务)

在这里我们还是在交互模式下手动去执行，我们想要 crontab 的定时生成和执行，我们可以用 celery 的 beat 去周期的生成任务和执行任务，在这个例子中我希望每 10 秒钟产生一个任务，然后去执行这个任务，我可以这样配置（在 app1_config.py 文件中添加如下内容）：

```
# 设计周期任务
CELERY_TIMEZONE = 'Asia/Shanghai'
from celery.schedules import crontab     # 设置定时任务
from datetime import timedelta
# 每隔 30 秒执行 app1.tasks1.deal 函数
CELERYBEAT_SCHEDULE = {
    'deal-every-30-seconds': {
        'task': 'app1.tasks1.deal1',
        'schedule': timedelta(seconds=30),
        'args': ['hello']
```

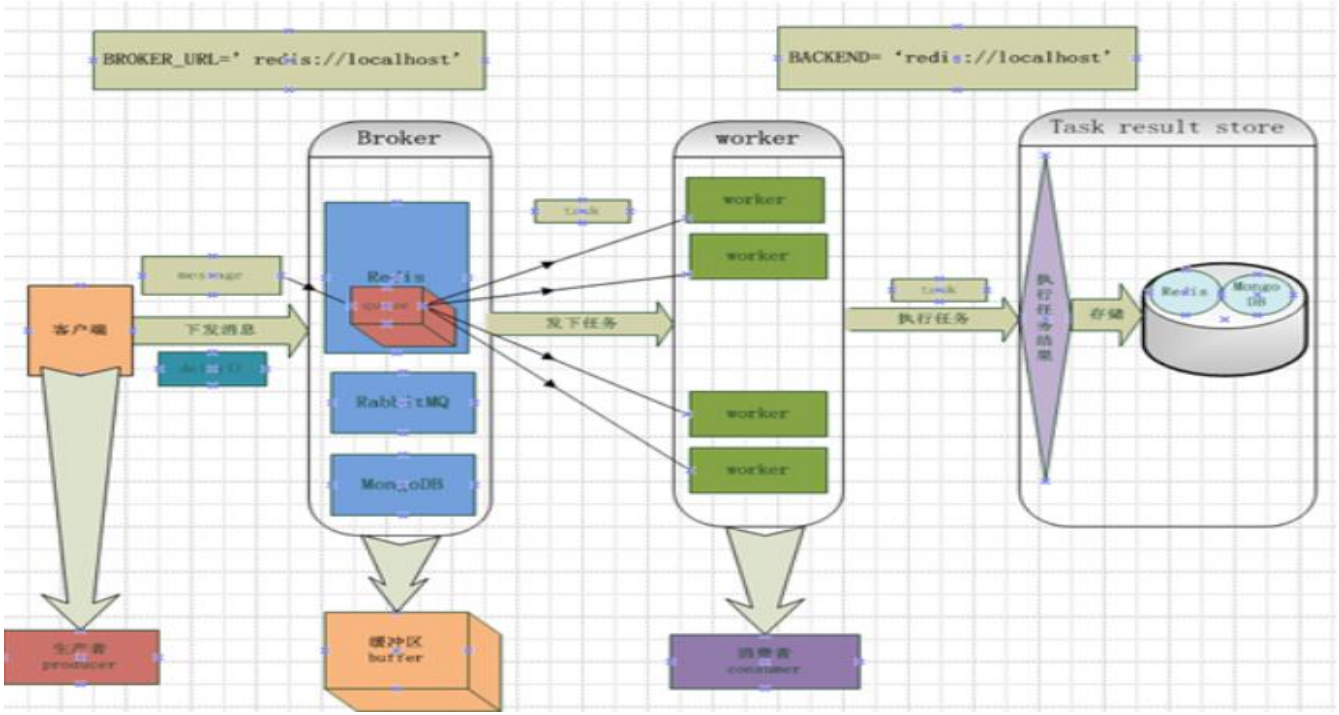
```
    },
    'deal-every-10-seconds': {
        'task': 'app1.tasks2.deal2',
        'schedule': timedelta(seconds=10),
        'args': ['hello']
    },
     # Executes every Monday morning at 7:30 A.M
    'deal-every-monday-morning': {
        'task': 'app1.tasks2.deal2',
        'schedule': crontab(hour=7, minute=30, day_of_week=1),
        'args': ['hello']
    },
}
```

使用了 scheduler，要制定时区：CELERY_TIMEZONE = ‘Asia/Shanghai’，启动 celery 加上-B 的参数。

celery -A app1.app1_app worker -l info -B

前两个任务为周期任务，第三个任务为定时任务，指定时间点开始执行分发任务，让 worker 取走执行，可以这样配置：

看完这些基础的东西，我回过头对 celery 在回顾了一下，用图把它的框架大致画出来，如下图：



celery 任务调度的配置项
https://docs.celeryproject.org/en/latest/userguide/configuration.html
celery 任务监控
celery flower

Name	Status	Monitoring	Remote Control
RabbitMQ	Stable	Yes	Yes
Redis	Stable	Yes	Yes
Amazon SQS	Stable	No	No
Zookeeper	Experimental	No	No

celery 控制命令
 # 查询命令
celery [Global Options] inspect [command] [Remote Control Options]

Global Options:
-A APP, --app APP
-b BROKER, --broker BROKER
--result-backend RESULT_BACKEND
--loader LOADER
--config CONFIG
--workdir WORKDIR
--no-color, -C
--quiet, -q

[Commands]
| active
| List of tasks currently being executed.
| active_queues
| List the task queues a worker is currently consuming from.
| clock

```
|           Get current logical clock value.
|       conf [include_defaults=False]
|           List configuration.
|       memdump [n_samples=10]
|           Dump statistics of previous memsample requests.
|       memsample
|           Sample current RSS memory usage.
|       objgraph [object_type=Request] [num=200 [max_depth=10]]
|           Create graph of uncollected objects (memory-leak debugging).
|       ping
|           Ping worker(s).
|       query_task [id1 [id2 [... [idN]]]]
|           Query for task information by id.
|       registered [attr1 [attr2 [... [attrN]]]]
|           List of registered tasks.
|       report
|           Information about Celery installation for bug reports.
|       reserved
|           List of currently reserved tasks, not including scheduled/active.
|       revoked
|           List of revoked task-ids.
|       scheduled
|           List of currently scheduled ETA/countdown tasks.
|       stats
|           Request worker statistics/information.


Remote Control Options:
  --timeout TIMEOUT, -t TIMEOUT
                        Timeout in seconds (float) waiting for reply
  --destination DESTINATION, -d DESTINATION
                        Comma separated list of destination node names.      指定 worker
  --json, -j            Use json as output format.
```

控制命令

```
Global Options:
  -A APP, --app APP
  -b BROKER, --broker BROKER
  --result-backend RESULT_BACKEND
  --loader LOADER
  --config CONFIG
  --workdir WORKDIR
  --no-color, -C
  --quiet, -q


  [Commands]
|     add_consumer <queue> [exchange [type [routing_key]]]
|         Tell worker(s) to consume from task queue by name.
|     autoscale [max [min]]
|         Modify autoscale settings.
|     cancel_consumer <queue>
|         Tell worker(s) to stop consuming from task queue by name.
|     disable_events
|         Tell worker(s) to stop sending task-related events.
|     election
|         Hold election.
|     enable_events
|         Tell worker(s) to send task-related events.
|     heartbeat
|         Tell worker(s) to send event heartbeat immediately.
|     pool_grow [N=1]
|         Grow pool by n processes/threads.
|     pool_restart
|         Restart execution pool.
|     pool_shrink [N=1]
|         Shrink pool by n processes/threads.
|     rate_limit <task_name> <rate_limit (e.g., 5/s | 5/m | 5/h)>
```

| Tell worker(s) to modify the rate limit for a task by type.
| revoke [id1 [id2 [... [idN]]]]
| Revoke task by task id (or list of ids).
| shutdown
| Shutdown worker(s).
| terminate <signal> [id1 [id2 [... [idN]]]]
| Terminate task by task id (or list of ids).
| time_limit <task_name> <soft_secs> [hard_secs]
| Tell worker(s) to modify the time limit for task by type.

例如 先停止一个 worker 消费任务便于手动重启升级

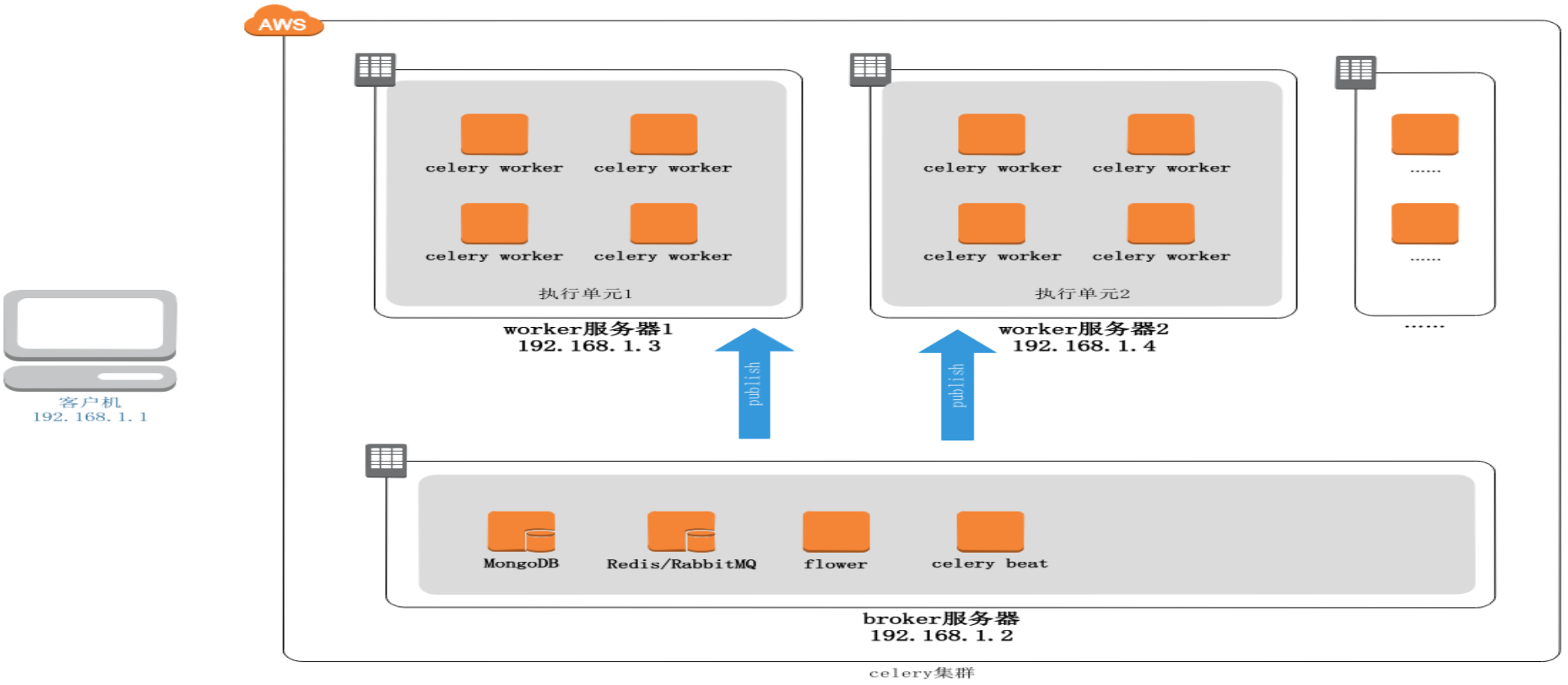
celery control -b redis://:yourpassword@redis-host:6379/1 cancel_consumer default -d celery@airflow-worker-1

笔记：集群部署 celery 分布式任务队列

估计来看此文的已经对 celery 有一些了解了，基本概念不再赘述。
网上找来找去全都是单机版的基础部署教程，也没有深入讲解分布式的部署过程。没办法只好靠着我渣渣英语强行研究了一波官方文档。
只有三台阿里云的服务器，不过做集群演示是够了。服务器配置比较低，所以选用 redis 作为消息中间件。

准备工作

实验环境：centos 6.5，python 3.4
三台服务器都装上 python 3.4 和 celery 4.1
我 pip 使用的阿里云的源，默认 celery 版本是 4.1.0，直接使用 sudo pip install celery 即可安装
其中一台做主服务器装上 redis 和 mongodb，和 celery 的 flower 插件
redis 做 broker
mongodb 用来做 backend，存储任务执行结果
flower 用来监测 celery 集群的状态
安装教程请自行搜索



注：

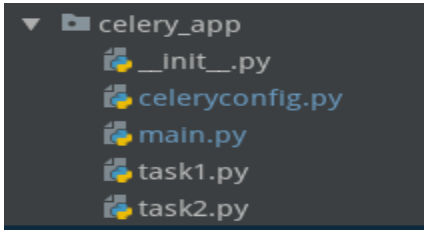
celery 命令的详细参数请看官方文档：http://docs.celeryproject.org/en/master/reference/celery.bin.worker.html

从 celery 4.0 开始配置文件有所变化，具体请看官方文档：http://docs.celeryproject.org/en/master/userguide/configuration.html#example-configuration-file

queue 配置：http://blog.csdn.net/tmpbook/article/details/52245716

编写代码

项目结构：



init.py

```
from celery import Celery

app = Celery("demo")
app.config_from_object("celery_app.celeryconfig")
```

celeryconfig.py

```
from celery.schedules import crontab
from datetime import timedelta
from kombu import Queue
from kombu import Exchange

result_serializer = 'json'

broker_url = "redis://192.168.1.2"
result_backend = "mongodb://192.168.1.2/celery"
timezone = "Asia/Shanghai"
imports = (
    'celery_app.task1',
    'celery_app.task2'
)

beat_schedule = {
    'add-every-20-seconds': {
        'task': 'celery_app.task1.multiply',
        'schedule': timedelta(seconds=20),
        'args': (5, 7)
    },
    'add-every-10-seconds': {
        'task': 'celery_app.task2.add',
         'schedule': crontab(hour=9, minute=10)
        'schedule': timedelta(seconds=10),
        'args': (23, 54)
    }
}
```

```
task_queues = (
    Queue('default', exchange=Exchange('default'), routing_key='default'),
    Queue('priority_high', exchange=Exchange('priority_high'), routing_key='priority_high'),
    Queue('priority_low', exchange=Exchange('priority_low'), routing_key='priority_low'),
)

task_routes = {
    'celery_app.task1.multiply': {'queue': 'priority_high', 'routing_key': 'priority_high'},
    'celery_app.task2.add': {'queue': 'priority_low', 'routing_key': 'priority_low'},
}

# 每分钟最大速率
# task_annotations = {
#     'task2.multiply': {'rate_limit': '10/m'}
# }
```

task1.py

```
import time
from celery_app import app

@app.task
def multiply(x, y):
    print("multiply")
    time.sleep(4)
    return x * y
```

task2.py

```
import time
from celery_app import app


@app.task
def add(x, y):
    print(add)
```

```
    time.sleep(2)
    return x + y
```

上传项目

写个同步文件 sync.sh，方便将项目文件分别上传到三台服务器

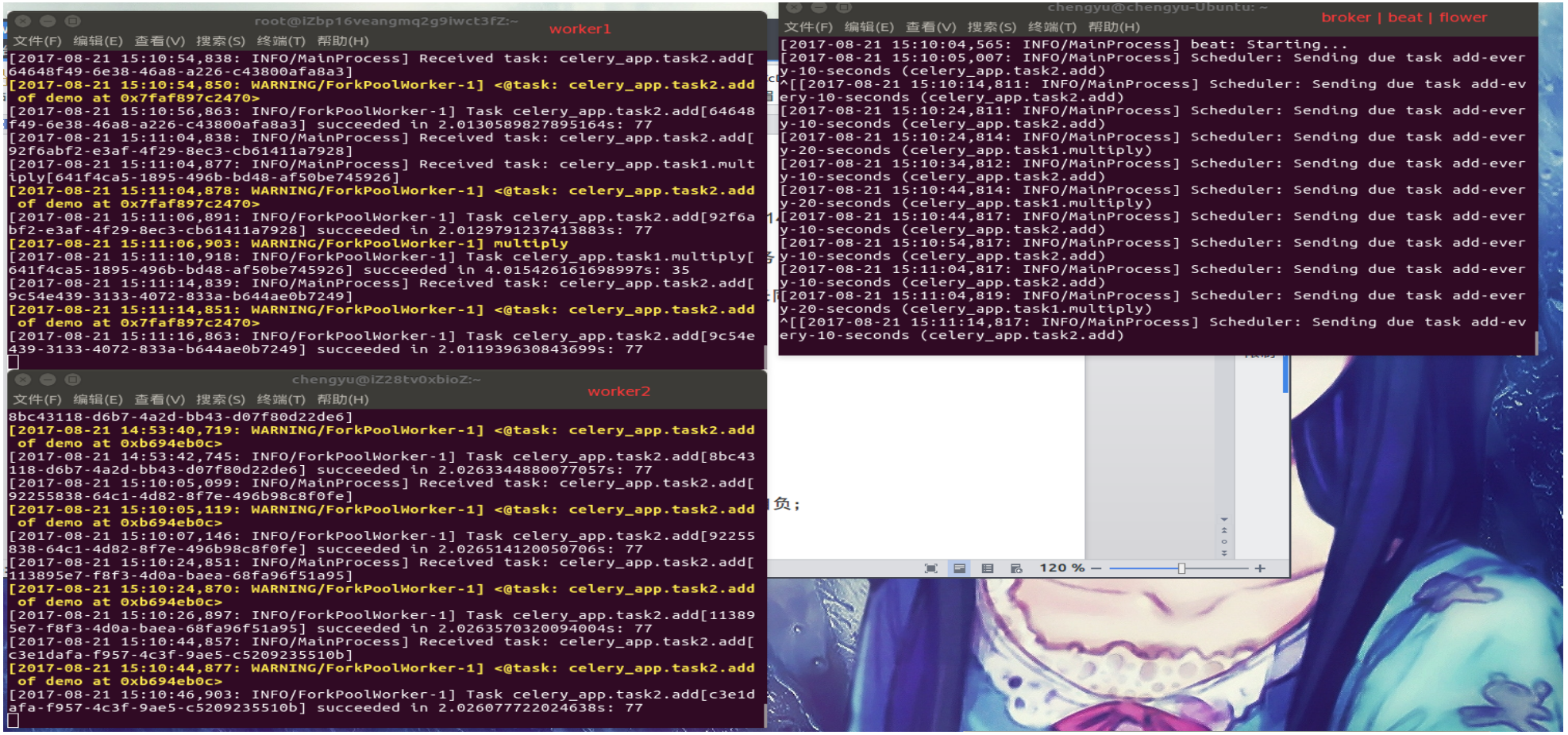
```
scp -r celery_app root@192.168.1.2:~
scp -r celery_app root@192.168.1.3:~
scp -r celery_app root@192.168.1.4:~
#开始上传
chmod +x sync.sh
./sync.sh
```

其实不是每台服务器都需要所有的文件，只是为了方便就全部上传上去了

启动

```
#worker 1
celery -A celery_app -l info -n worker1
#worker 2
celery -A celery_app -l info -n worker2
#broker
/var/mongodb/bin/mongod -f /var/mongodb/conf/mongod.conf
nohup redis-server &amp;
nohup celery -A celery_app flower -l info &amp;
celery -A celery_app beat -l info
```

全部启动完成之后可以看到 celery 已经开始自动分配定时任务了



提交自定义任务

test.py

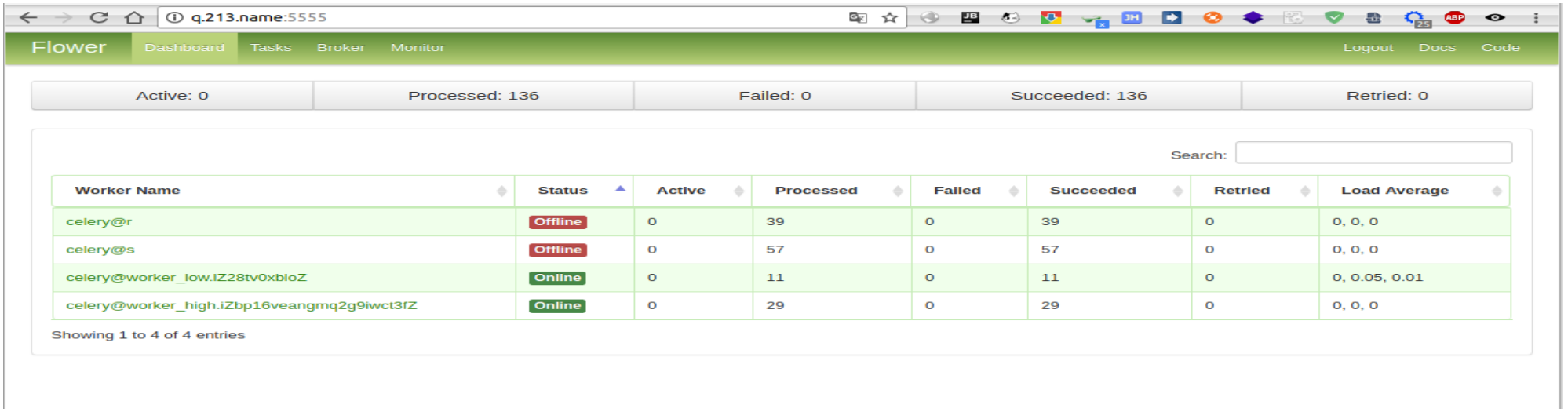
```
from celery_app import task1
from celery_app import task2
re = task1.multiply.delay(2, 8)
re2 = task2.add.delay(5, 6)
print("ok")
print(re.get())
```

```
print(re2.get())
```

输出：

```
/usr/bin/python3.4 /media/chengyu/Project/Github/CMSpider/test.py
ok
16
11
Process finished with exit code 0
```

进入 flower 监控服务器



可能遇到的错误

pymongo.errors.NotMasterError: not master

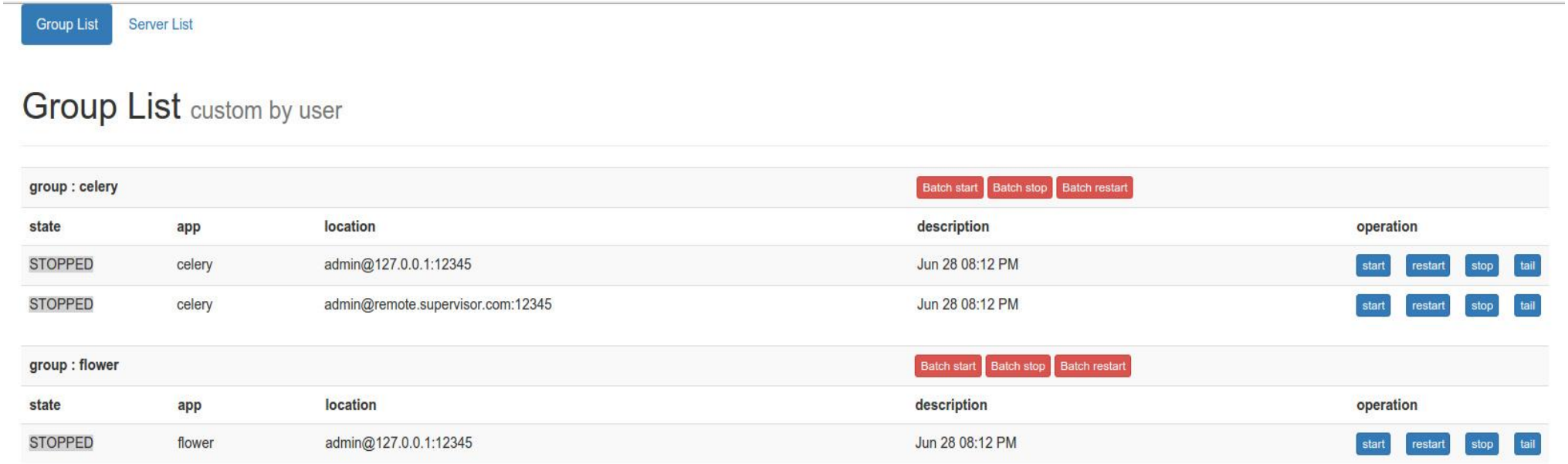
关闭 mongodb 的数据集

其他

当服务器数量较多的时候，管理起来会很不方便，可以使用 python 的 supervisor 来管理后台进程，遗憾的是它并不支持 python3，不过也可以装在 python2 的环境

虽然用了 supervisor 可以很方便的管理 python 程序，但是还是得一个个登陆不同的服务器的去管理，咋办捏？

我在 github 上找到一个工具 supervisor-easy，可以批量管理 supervisor，如图:



地址：https://github.com/trytofix/supervisor-easy

dockerfile 中的命令：run, cmd, entrypoint， copy 和 add

总结一下，

run 可以有多个，**cmd** 和 **entrypoint** 只能有一个（常用来跑 **app**）

cmd 可以被 **docker** 指令 **overwrite** ，**entrypoint** 不可以

此命令会在容器启动且 **docker run** 没有指定其他命令时运行。

如果 **docker run** 指定了其他命令，**CMD** 指定的默认命令将被忽略。

如果 **Dockerfile** 中有多个 **CMD** 指令，只有最后一个 **CMD** 有效。

ENTRYPOINT 的 **Exec** 格式用于设置要执行的命令及其参数，同时可通过 **CMD** 提供额外的参数。

ENTRYPOINT 中的参数始终会被使用，而 **CMD** 的额外参数可以在容器启动时动态替换掉。

比如下面的 **Dockerfile** 片段：

```
ENTRYPOINT ["/bin/echo", "Hello"]
CMD ["world"]
```

当容器通过 **docker run -it [image]** 启动时，输出为：

```
Hello world
```

而如果通过 **docker run -it [image] CloudMan** 启动，则输出为：

Hello CloudMan

docker 2、优化 docker 镜像大小方法

1 查看镜像 id

即执行如下命令:

```
[root@node-1 ~]# docker images|grep xxx
docker.io/xxx                 1.0                 5d1b456495fc        20 hours ago        1.64 GB
```

注意:

请将 xxx 替换为你需要查找的镜像名称

2 查看该镜像的分层情况

即执行如下命令:

```
docker history <image_id>
```

解释:

docker history 是查看镜像的创建历史，最下面是分层的最底层。

```
[root@node-1 ~]# docker history 5d1b456495fc
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
5d1b456495fc        20 hours ago        /bin/sh -c #(nop)  USER yyy                     0 B
<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL maintainer=xxx...      0 B
<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL last-commit-id=08...   0 B
<missing>           20 hours ago        /bin/sh -c chmod 750 /etc/sudoers.d       &&...   535 B
<missing>           20 hours ago        /bin/sh -c pip install tenacity===3.1.0    ...    186 MB
<missing>           20 hours ago        /bin/sh -c #(nop) COPY file:f483b3f3d6368f...     117 B
<missing>           20 hours ago        /bin/sh -c #(nop) COPY file:a82f5a713f9ec5...     418 B
<missing>           20 hours ago        /bin/sh -c mkdir /gnocchi-base-source && c...     370 MB
<missing>           20 hours ago        /bin/sh -c mkdir -p /var/www/cgi-bin/gnocc...     21.2 kB
<missing>           20 hours ago        /bin/sh -c yum -y install httpd mod_ssl mo...     27 MB
<missing>           20 hours ago        /bin/sh -c usermod --append --home /var/li...     23.4 kB
<missing>           20 hours ago        /bin/sh -c #(nop)  LABEL maintainer=EasySt...     0 B
<missing>           6 months ago        /bin/sh -c pip --no-cache-dir install --up...     94.6 MB
<missing>           6 months ago        /bin/sh -c #(nop) COPY file:2e2bd3f2cc2c54...     1.04 kB
<missing>           6 months ago        /bin/sh -c pip --no-cache-dir install --up...     738 kB
<missing>           6 months ago        /bin/sh -c yum -y install gcc gcc-c++ libf...     631 MB
<missing>           6 months ago        /bin/sh -c yum -y install git iproute open...     75.8 MB
<missing>           6 months ago        /bin/sh -c #(nop)  LABEL maintainer=...     0 B
<missing>           6 months ago        /bin/sh -c touch /usr/local/bin/kolla_exte...     15.8 kB
<missing>           6 months ago        /bin/sh -c rm /etc/localtime       && ln -s ...     0 B
```

<missing> 6 months ago /bin/sh -c #(nop) COPY file:b89240fb6cba42... 97 B
<missing> 6 months ago /bin/sh -c mkdir -p /root/.pip/ 0 B
<missing> 6 months ago /bin/sh -c curl http://..../dum... 10.2 MB
<missing> 6 months ago /bin/sh -c #(nop) COPY file:f67b9142d3f871... 154 B
<missing> 6 months ago /bin/sh -c #(nop) COPY file:1c53e6b1b4655f... 831 B
<missing> 6 months ago /bin/sh -c #(nop) COPY file:945dd7de8a0bc7... 503 B
<missing> 6 months ago /bin/sh -c #(nop) COPY file:6557b170983c4c... 14.4 kB
<missing> 6 months ago /bin/sh -c yum -y install curl iproute isc... 35.5 MB
<missing> 6 months ago /bin/sh -c #(nop) COPY file:3a59b0f663154a... 130 B
<missing> 6 months ago /bin/sh -c #(nop) COPY file:30bbaf3e30e53d... 119 B
<missing> 6 months ago /bin/sh -c #(nop) COPY file:7a5b14cdd97eb5... 459 B
<missing> 6 months ago /bin/sh -c CURRENT_DISTRO_RELEASE=$(awk '{... 3.97 kB
<missing> 6 months ago /bin/sh -c #(nop) ENV PS1=$(tput bold)$(t... 0 B
<missing> 6 months ago /bin/sh -c cat /tmp/kolla_bashrc >> /etc/s... 705 B
<missing> 6 months ago /bin/sh -c #(nop) COPY file:1020b4b8ead353... 149 B
<missing> 6 months ago /bin/sh -c #(nop) ENV KOLLA_BASE_DISTRO=e... 0 B
<missing> 6 months ago /bin/sh -c #(nop) LABEL kolla_version=5.0.1 0 B
<missing> 6 months ago /bin/sh -c groupadd --force --gid 42491 al... 18.7 MB
<missing> 6 months ago /bin/sh -c #(nop) LABEL maintainer=EasySt... 0 B
<missing> 9 months ago /bin/sh -c #(nop) ENV LC_ALL=en_US.utf8 0 B
<missing> 9 months ago /bin/sh -c #(nop) ADD file:4dadc4b90f7e5a4... 195 MB

分析:

下面的部分是 gnocchi-base 打镜像的各个步骤对应的镜像层级

IMAGE CREATED CREATED BY SIZE COMMENT
5d1b456495fc 20 hours ago /bin/sh -c #(nop) USER gnocchi 0 B
<missing> 20 hours ago /bin/sh -c #(nop) LABEL maintainer=EasySt... 0 B
<missing> 20 hours ago /bin/sh -c #(nop) LABEL last-commit-id=08... 0 B
<missing> 20 hours ago /bin/sh -c chmod 750 /etc/sudoers.d &&... 535 B
<missing> 20 hours ago /bin/sh -c pip install tenacity===3.1.0 ... 186 MB
<missing> 20 hours ago /bin/sh -c #(nop) COPY file:f483b3f3d6368f... 117 B
<missing> 20 hours ago /bin/sh -c #(nop) COPY file:a82f5a713f9ec5... 418 B
<missing> 20 hours ago /bin/sh -c mkdir /gnocchi-base-source && c... 370 MB
<missing> 20 hours ago /bin/sh -c mkdir -p /var/www/cgi-bin/gnocc... 21.2 kB
<missing> 20 hours ago /bin/sh -c yum -y install httpd mod_ssl mo... 27 MB
<missing> 20 hours ago /bin/sh -c usermod --append --home /var/li... 23.4 kB

<missing> 20 hours ago /bin/sh -c #(nop) LABEL maintainer=... 0 B

下面分析占用空间占用较大的的层级

可以看到:

<missing> 20 hours ago /bin/sh -c pip install tenacity===3.1.0 ... 186 MB

这里是 gnocchi 中需要下载的第三方库对应的镜像层级，对应代码如下:

```
RUN pip install tenacity===3.1.0 \
    && pip install oslo.middleware===3.30.1 \
RUN chmod 750 /etc/sudoers.d \
    && chmod 640 /etc/sudoers.d/kolla_gnocchi_sudoers \
    && touch /usr/local/bin/kolla_gnocchi_extend_start \
    && chmod 755 /usr/local/bin/kolla_extend_start /usr/local/bin/kolla_gnocchi_extend_start
```

将多个连续的 RUN 命令放在一起，即变成如下形式:

```
RUN pip install tenacity===3.1.0 \
    && pip install oslo.middleware===3.30.1 \
    && chmod 750 /etc/sudoers.d \
    && chmod 640 /etc/sudoers.d/kolla_gnocchi_sudoers \
    && touch /usr/local/bin/kolla_gnocchi_extend_start \
    && chmod 755 /usr/local/bin/kolla_extend_start /usr/local/bin/kolla_gnocchi_extend_start
```

3 docker 镜像大小优化方法

优化思路:

1) 将连续的 RUN 命令合并在一起。

2) 设置某个组件的基础镜像，这个基础镜像中只存放这个组件中不变的代码，将这个组件中会变化的部分放入该组件的各个服务中。

以 gnocchi 为例，设置 gnocchi-base 作为 gnocchi 这个组件的基础镜像，里面放入不会改变的部分，

将会修改的内容放入 gnocchi 的各个服务中，例如: gnocchi-metricd, gnocchi-statsd, gnocchi-api

3) 删除镜像中不用的安装库，删除 yum 的缓存。

即在执行执行 yum 安装命令后，就执行 yum clean all 清理缓存和 rm -rf /var/lib/yum/yumdb

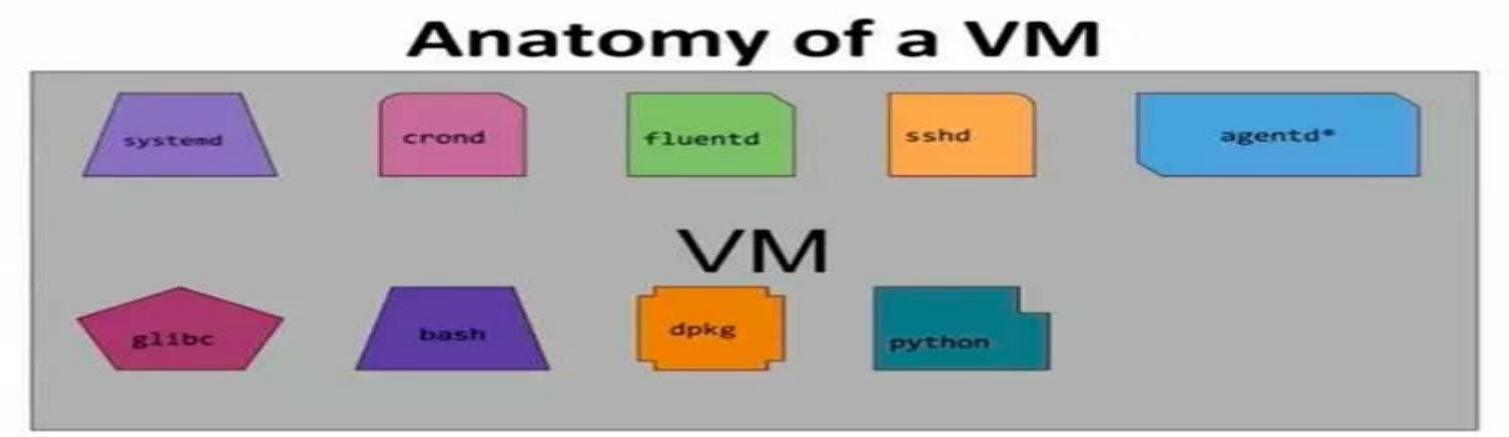
例子如下:

```
RUN yum -y install epel-release \
    && yum -y install rsyslog
    && yum clean all
    && rm -rf /var/lib/yum/yumdb
```

谷歌技术人员解决 Docker 镜像体积太大问题的方法

这篇文章主要介绍了谷歌技术人员解决 Docker 镜像体积太大问题的方法，涉及虚拟机，谷歌 docker 镜像构建实践及构建工具 bazel 的介绍等相关内容，具有一定参考价值，需要的朋友可以了解下。

虚拟机的问题



最初，大家都使用虚拟机作为软件的运行环境，对外提供服务。为了在虚拟机上运行你的 Service，你不得不运行一大堆程序：

系统进程
定时任务
SSH
安装 Agent
安装 Bash
安装一大堆 libs

其实，你仅仅只是想让你的 Service 运行起来，但你不得不维护一个 40GB 的虚拟机。
然后你开始试用 Docker

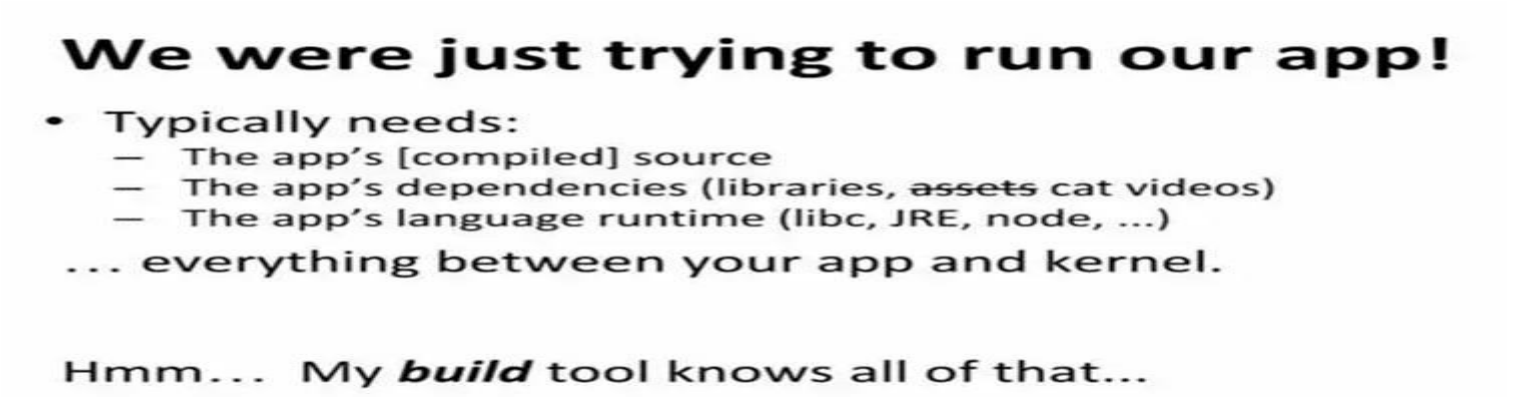
开始试用 Docker，你毫不犹豫选择了和之前虚拟机一样的镜像：Ubuntu 1404，将之前的虚机的内容复制到了 Docker 镜像，安装了一堆软件，最后发现你的 Docker 镜像有 8GB。

谷歌的 Docker 镜像构建实践

找到最小的基础镜像

Alpine Linux 是基于 musl 和 BusyBox 的操作系统，目的是为了为用户提供更高效的资源使用效率。它的特性是体积小，最小的 Alpine Linux 体积可以只有 5MB。谷歌某些团队使用 Alpine Linux 作为 Docker Build 的基础镜像。

目的：仅仅为了运行 Service



谷歌认为，为了运行一个 Service，并不需要将那些无关联的包、程序打包到容器里，换句话说，Docker 镜像里只留下需要用到的，其他的都删除，从而得到一个最小的镜像。这需要考虑以下几点：

1、程序编译后的二进制文件（从 Artifactory 获取）

2、程序的所有依赖（从 Artifactory 获取）

3、程序语言的运行时（libc，JRE，node，...）

4、任何程序和 Kernel 之间的中间件

其实这一切的信息，构建工具都已经知道。

谷歌的构建工具 Bazel

介绍 Bazel 之前，先介绍下谷歌的开发模式，对于服务器端代码库，谷歌的开发流程如下：

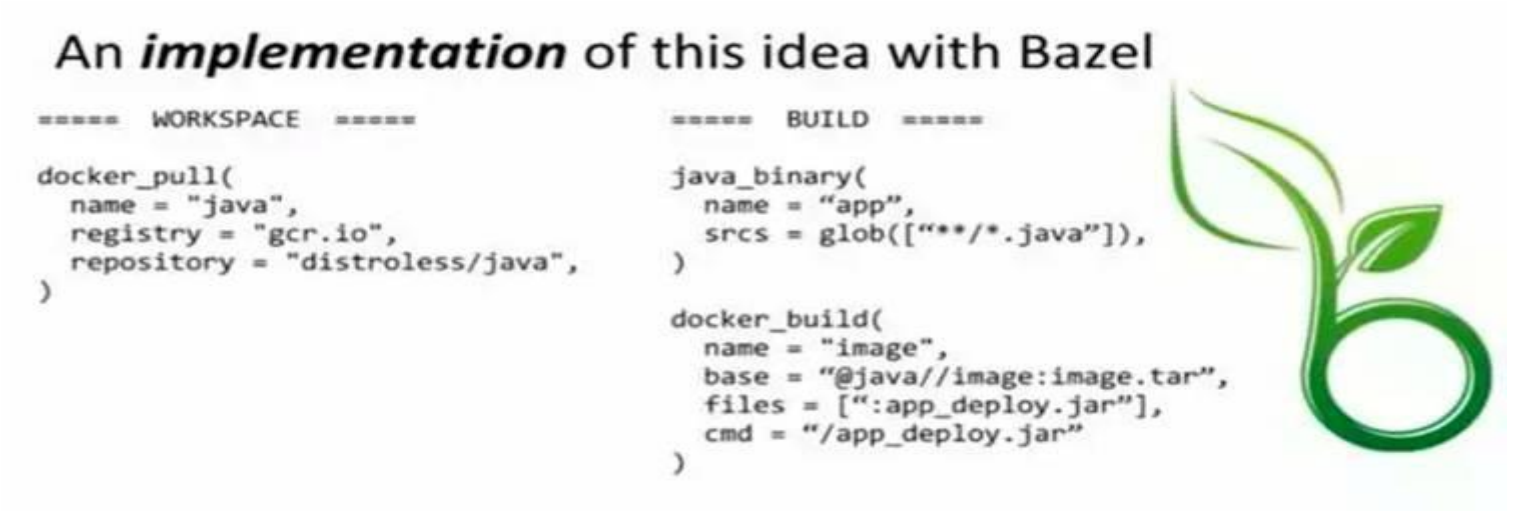
1、所有的服务器端代码库都在一个巨大的版本控制系统里

2、每个人都用 Bazel 构建软件

3、不同的组负责源码树的不同部分，所有的组件都是作为 BUILD 目标来用

4、分支主要是用来管理发布，所以每个人都在最新版本上开发软件

Bazel（https://bazel.build/）是 Google 内部用来构建自己的服务器端软件的工具。目前变成谷歌公司贡献的一个开源项目，目的是帮助开发者将软件的构建和测试变得更快、更可靠。



从上图可以看到，Bazel 有 WORDSPACE 的概念，WORDSPACE 文件用来准备 Docker 镜像构建所依赖的所有材料和来源。BUILD 文件用来告诉 Bazel 这个镜像应该使用什么命令进行构建，以及如何构建、如何测试。
使用 Bazel 的声明式语言：WORKSPACE 和 BUILD，开发者可以用文件描述整个构建和部署的环境。谷歌使用 Bazel 进行 Docker 的构建已经很多年，它为谷歌带来以下收益：
1、支持跨平台构建，分布式缓存，优化依赖解析，并行构建，增量构建。
2、支持多语言（Java, C++, Android, iOS, Go 等等）。
3、跨平台。
4、水平扩展和自定义扩展。
Bazel 是以下理念的奠基石：由于 Bazel 需要所有的依赖都被完整地指定，我们可以预测改动影响了哪些程序和测试，并在提交前执行他们。

谷歌提供的 Distroless 镜像构建文件

Distroless (https://github.com/GoogleCloudPlatform/distroless)是谷歌内部使用的镜像构建文件，包括 Java、Node、Python 等镜像构建文件，Distroless 仅仅只包含运行服务所需要的最小镜像，不包含包管理工具、shell 命令行等其他功能。
为什么你需要这些镜像？这些镜像是谷歌和其他大公司的最佳实践的产物，经过了漏洞扫描，镜像会持续更新，保持安全性。
如何使用？Distroless 提供的构建镜像的 BUILD 文件，通过 Bazel 可以直接进行构建。
谷歌为大家介绍了内部如何解决镜像过大的问题，以及进行大规模并发构建、测试所用到的构建工具 Bazel，并且开源了 Docker 镜像构建文件 Distroless 项目。如果你认为你的镜像也存在体积太大的问题，可以参考谷歌的实践，体验他们的工具。

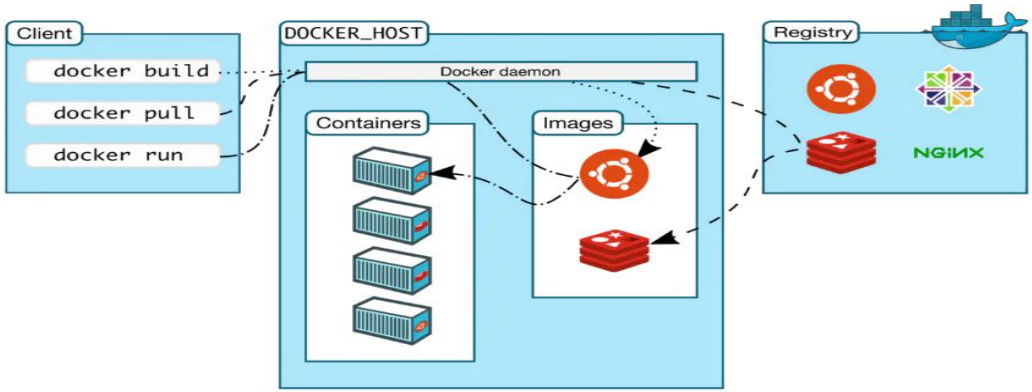
总结

以上就是本文关于谷歌技术人员解决 Docker 镜像体积太大问题的方法的全部内容，希望对大家有所帮助。感兴趣的朋友可以可以继续参阅本站：

Docker 及 Docker-Compose 的使用

菜鸟 Docker
阮一峰的 Docker 教程
Docker 是一个开源的容器引擎，它有助于更快地交付应用。方便快捷已经是 Docker 的最大优势，过去需要用数天乃至数周的任务，在 Docker 容器的处理下，只需要数秒就能完成。

架构



Docker架构图

- Docker daemon（ Docker 守护进程）：Docker daemon 是一个运行在宿主机（ DOCKER-HOST）的后台进程。可通过 Docker 客户端与之通信。

- Client（ Docker 客户端）：Docker 客户端是 Docker 的用户界面，它可以接受用户命令和配置标识，并与 Docker daemon 通信。图中， docker build 等都是 Docker 的相关命令。
- Images（ Docker 镜像）：Docker 镜像是一个只读模板，它包含创建 Docker 容器的说明。它和系统安装光盘有点像，使用系统安装光盘可以安装系统，同理，使用 Docker 镜像可以运行 Docker 镜像中的程序。
- Container（容器）：容器是镜像的可运行实例。镜像和容器的关系有点类似于面向对象中，类和对象的关系。可通过 Docker API 或者 CLI 命令来启停、移动、删除容器。
- Registry：Docker Registry 是一个集中存储与分发镜像的服务。构建完 Docker 镜像后，就可在当前宿主机上运行。但如果想要在其他机器上运行这个镜像，就需要手动复制。此时可借助 Docker Registry 来避免镜像的手动复制。一个 Docker Registry 可包含多个 Docker 仓库，每个仓库可包含多个镜像标签，每个标签对应一个 Docker 镜像。这跟 Maven 的仓库有点类似，如果把 Docker Registry 比作 Maven 仓库的话，那么 Docker 仓库就可理解为某 jar 包的路径，而镜像标签则可理解为 jar 包的版本号。Docker Registry 可分为公有 Docker Registry 和私有 Docker Registry。 最常用的 DockerRegistry 莫过于官网的 Docker Hub， 这也是默认的 Docker Registry。 Docker Hub 上存放着大量优秀的镜像， 我们可使用 Docker 命令下载并使用。

安装

按照菜鸟的步骤，使用 yum 安装即可。

常用命令

镜像相关

- docker search java：在 Docker Hub（或阿里镜像）仓库中搜索关键字（如 java）的镜像
- docker pull java:8：从仓库中下载镜像，若要指定版本，则要在冒号后指定
- docker images：列出已经下载的镜像
- docker rmi java：删除本地镜像
- docker build：构建镜像

容器相关

- docker run -d -p 91:80 nginx ：在后台运行 nginx，若没有镜像则先下载，并将容器的 80 端口映射为宿主机的 91 端口。
  - -d：后台运行
  - -P：随机端口映射
  - -p：指定端口映射
  - -net：网络模式

- docker ps：列出运行中的容器
- docker ps -a ：列出所有的容器
- docker stop 容器 id：停止容器
- docker kill 容器 id：强制停止容器
- docker start 容器 id：启动已停止的容器
- docker inspect 容器 id：查看容器的所有信息
- docker container logs 容器 id：查看容器日志
- docker top 容器 id：查看容器里的进程
- docker exec -it 容器 id /bin/bash：进入容器
- exit：退出容器
- docker rm 容器 id：删除已停止的容器
- docker rm -f 容器 id：删除正在运行的容器

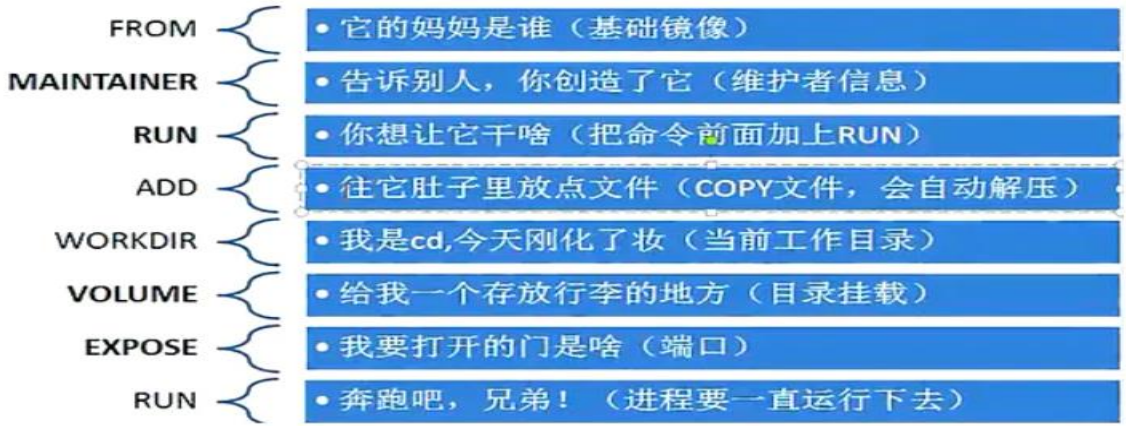
所有命令

- docker
- docker COMMAND --help

构建镜像

1. 确定镜像模板：如 java、nginx
2. 新建 Dockerfile 文件
3. 使用 Dockerfile 的指令完善 Dockerfile 的内容
4. 在 Dockerfile 文件的所在路径执行 `docker build -t imageName:tag .`，-t 指定镜像名称，末尾的点标识 Dockerfile 文件的路径
5. 执行 `docker run -d -p 92:80 imageName:tag` 即可

常用指令如下图，直白用法点我，官方介绍点击我



个性化指令解释

备注：RUN 命令在 image 文件的构建阶段执行，执行结果都会打包进入 image 文件；CMD 命令则是在容器启动后执行。另外，一个 Dockerfile 可以包含多个 RUN 命令，但是只能有一个 CMD 命令。注意，指定了 CMD 命令以后，docker container run 命令就不能附加命令了，否则它会覆盖 CMD 命令。

Docker Compose

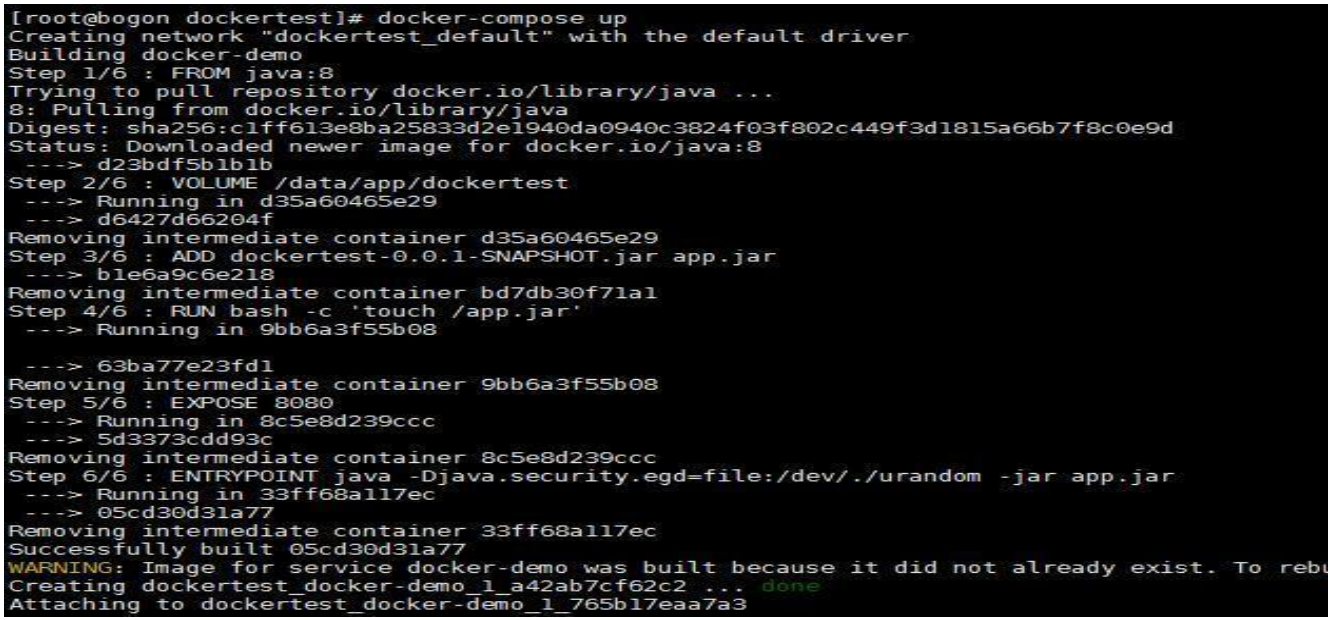
Docker Compose 是 docker 提供的一个命令行工具，用来定义和运行由多个容器组成的应用。使用 compose，我们可以通过 YAML 文件声明式的定义应用程序的各个服务，并由单个命令完成应用的创建和启动。

本文主要说 Docker Compose，但是在这之前，有必要知道一下，Docker，Docker Compose，Docker Swarm，Kubernetes 之间的区别。具体的如图，简单点说，Docker Compose 是单机管理 Docker 的，Kubernetes 是多节点管理 Docker 的。虽然 Docker Swarm 也是多节点管理，但基本已弃用，了解一下就好了。

安装 Docker Compose
接上一篇的例子，安装 Docker Compose，并进行赋权和检验。
sudo curl -L https://github.com/docker/compose/releases/download/1.21.2/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-composesudo chmod +x /usr/local/bin/docker-composedocker-compose -v 最后能看到版本信息，就是安装好了，有人可能会遇到一个错误。[53396] Cannot open self /usr/local/bin/docker-compose or archive /usr/local/bin/docker-compose.pkg。删除掉/usr/local/bin/docker-compose，重新下载一下，可能是网络原因下载有中断情况，而这个文件又仍然会生成。

```
[root@bogon bin]# sudo curl -L https://github.com/docker/compose/releases/download/1.23.0-rc3/docker-compose-`uname -s`-`uname
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   617    0   617    0     0    418      0 --:--:--  0:00:01 --:--:--   418
100 11.1M  100 11.1M    0     0   857k      0  0:00:13  0:00:13 --:--:--  620k
[root@bogon bin]# sudo chmod +x /usr/local/bin/docker-compose
[root@bogon bin]# docker-compose version
docker-compose version 1.23.0-rc3, build ea3d406e
docker-py version: 3.5.0
CPython version: 3.6.6
OpenSSL version: OpenSSL 1.1.0f  25 May 2017
```

准备 Dockerfile 和 docker-compose.yml
在 jar 包所在目录，新建 Dockerfile 文件，touch Dockerfile，然后输入内容 vi Dockerfile：
FROM java:8VOLUME /data/app/dockertestADD dockertest-0.0.1-SNAPSHOT.jar app.jarRUN bash -c 'touch /app.jar'EXPOSE 8080ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","app.jar"] 对于这个配置文件的参数含义，可以百度查一下，这里说一下 volume 就是你 jar 包的文件夹位置，add 后的第一个就是宿主机的待发布应用，后面就是到容器后的（可以回想上一篇最后的启动命令）。
再在这个目录里，新建 docker-compose.yml，参考输入内容：
version: '2' # 表示该 Docker-Compose 文件使用的是 Version 2 fileservices: docker-demo: # 指定服务名称 build: . # 指定 Dockerfile 所在路径 ports: # 指定端口映射 - "8080:8080"启动 docker-compose
配置完后，就可以通过 Docker Compose 启动你配置的镜像了。
先关掉之前使用 docker run 启动的镜像（docker stop $(docker ps -aq)），再启动 Docker Compose（docker-compose up），启动命令后面也有很多参数，比如-d 在后台运行服务容器。这里就简单启动一下看看效果：



看到绿色的 done，就知道已经成功了，再等会儿，就打印出应用控制台的内容了，这时候访问一下 ip:8080（这里配置的应用端口），发现应用可以正常访问到了。

Docker 入门之 docker-compose

一，Docker-compose 简介

1，Docker-compose 简介

Docker-Compose 项目是 Docker 官方的开源项目，负责实现对 Docker 容器集群的快速编排。

Docker-Compose 将所管理的容器分为三层，分别是工程（project），服务（service）以及容器（container）。Docker-Compose 运行目录下的所有文件（docker-compose.yml，extends 文件或环境变量文件等）组成一个工程，若无特殊指定工程名即为当前目录名。一个工程当中可包含多个服务，每个服务中定义了容器运行的镜像，参数，依赖。一个服务当中可包括多个容器实例，Docker-Compose 并没有解决负载均衡的问题，因此需要借助其它工具实现服务发现及负载均衡。

Docker-Compose 的工程配置文件默认为 docker-compose.yml，可通过环境变量 COMPOSE_FILE 或-f 参数自定义配置文件，其定义了多个有依赖关系的服务及每个服务运行的容器。

使用一个 Dockerfile 模板文件，可以让用户很方便的定义一个单独的应用容器。在工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器，甚至还包括负载均衡容器等。

Compose 允许用户通过一个单独的 docker-compose.yml 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。

Docker-Compose 项目由 Python 编写，调用 Docker 服务提供的 API 来对容器进行管理。因此，只要所操作的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理。

2，Docker-compose 的安装

安装环境查看

```
root@test-docker1:/data/php# uname -a
Linux test-docker1 4.15.0-62-generic #69-Ubuntu SMP Wed Sep 4 20:55:53 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
root@test-docker1:/data/php# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.1 LTS
Release:        18.04
Codename:       bionic
```

安装

```
apt  install  python-pip
pip  install  docker-compose
```

PS：centos 使用命令 yum -y install python-pip 安装 pip

查看安装的版本

```
# docker-compose --version
docker-compose version 1.24.0, build 0aa5906
```

3，Docker-compose 卸载

```
pip uninstall docker-compose
```

二，Docker-compose 常用命令

1，Docker-compose 命令格式

```
docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]
```

命令选项如下

```
-f --file  FILE 指定 Compose 模板文件，默认为 docker-compose.yml
-p --project-name NAME 指定项目名称，默认使用当前所在目录为项目名
--verbose    输出更多调试信息
-v，-version 打印版本并退出
--log-level LEVEL 定义日志等级(DEBUG, INFO, WARNING, ERROR, CRITICAL)
```

2，docker-compose up

```
docker-compose up [options] [--scale SERVICE=NUM...] [SERVICE...]
选项包括：
-d 在后台运行服务容器
-no-color 不是有颜色来区分不同的服务的控制输出
-no-deps 不启动服务所链接的容器
--force-recreate 强制重新创建容器，不能与-no-recreate 同时使用
-no-recreate 如果容器已经存在，则不重新创建，不能与 -force-recreate 同时使用
-no-build 不自动构建缺失的服务镜像
-build 在启动容器前构建服务镜像
-abort-on-container-exit  停止所有容器，如果任何一个容器被停止，不能与-d 同时使用
-t, -timeout TIMEOUT 停止容器时候的超时（默认为 10 秒）
-remove-orphans 删除服务中没有在 compose 文件中定义的容器
```

3，docker-compose ps

```
docker-compose    ps  [options] [SERVICE...]
列出项目中所有的容器
```

4，docker-compose stop

```
docker-compose stop [options] [SERVICE...]
选项包括
-t, -timeout TIMEOUT 停止容器时候的超时（默认为 10 秒）
docker-compose stop
停止正在运行的容器，可以通过 docker-compose start 再次启动
```

5，docker-compose -h

```
docker-compose -h
查看帮助
```

6，docker-compose down

```
docker-compose down [options]
停止和删除容器、网络、卷、镜像。
选项包括：
 - rmi  type，删除镜像，类型必须是：all，删除 compose 文件中定义的所有镜像；local，删除镜像名为空的镜像
-v,  - volumes，删除已经在 compose 文件中定义的和匿名的附在容器上的数据卷
 - remove-orphans，删除服务中没有在 compose 中定义的容器
docker-compose down
停用移除所有容器以及网络相关
```

7，docker-compose logs

```
docker-compose logs [options] [SERVICE...]
查看服务容器的输出。默认情况下，docker-compose 将对不同的服务输出使用不同的颜色来区分。可以通过 - no-color 来关闭颜色。
docker-compose logs
查看服务容器的输出
-f 跟踪日志输出
```

8，docker-compose bulid

```
 docker-compose build [options] [--build-arg key=val...] [SERVICE...]
 构建（重新构建）项目中的服务容器。
 选项包括：
  - compress 通过 gzip 压缩构建上下环境
  - force-rm  删除构建过程中的临时容器
  - no-cache 构建镜像过程中不使用缓存
  - pull 始终尝试通过拉取操作来获取更新版本的镜像
 -m,  - memory MEM 为构建的容器设置内存大小
  - build-arg key=val 为服务设置 build-time 变量
```

```
10  服务容器一旦构建后，将会带上一个标记名。可以随时在项目目录下运行 docker-compose build 来重新构建服务
```

9，docker-compose pull

```
1  docker-compose pull [options] [SERVICE...]
2  拉取服务依赖的镜像。
3  选项包括：
4   - ignore-pull-failures，忽略拉取镜像过程中的错误
5   - parallel，多个镜像同时拉取
6   - quiet，拉取镜像过程中不打印进度信息
7  docker-compose pull
8  拉取服务依赖的镜像
```

10，docker-compose restart

```
1  docker-compose restart [options] [SERVICE...]
2  重启项目中的服务。
3  选项包括：
4  -t, - timeout TIMEOUT，指定重启前停止容器的超时（默认为 10 秒）
5  docker-compose restart
6  重启项目中的服务
```

11，docker-compose rm

```
1  docker-compose  rm  [options] [SERVICE...]
2  删除所有（停止状态的）服务容器。
3  选项包括：
4   - f, - force，强制直接删除，包括非停止状态的容器
5  -v，删除容器所挂载的数据卷
6  docker-compose  rm
7  删除所有（停止状态的）服务容器。推荐先执行 docker-compose stop 命令来停止容器。
```

12，docker-compose start

```
1  docker-compose start [SERVICE...]
```

```
2
3
```
docker-compose start
启动已经存在的服务容器。

13，docker-compose run

```
1
2
3
4
```
docker-compose run [options] [-v VOLUME...] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]
在指定服务上执行一个命令。
docker-compose run ubuntu ping www.baidu.com
在指定容器上执行一个 ping 命令。

14，docker-compose scale

```
1
2
```
docker-compose scale web=3 db=2
设置指定服务运行的容器个数。通过 service=num 的参数来设置数量

15，docker-compose pause

```
1
2
```
docker-compose pause [SERVICE...]
暂停一个服务容器

16，docker-compose kill

```
1
2
3
4
```
docker-compose kill [options] [SERVICE...]
通过发送 SIGKILL 信号来强制停止服务容器。
支持通过-s 参数来指定发送的信号，例如通过如下指令发送 SIGINT 信号：
docker-compose kill -s SIGINT

17，docker-compose config

```
1
2
3
4
5
6
```
docker-compose config [options]
验证并查看 compose 文件配置。
选项包括：
 - resolve-image-digests 将镜像标签标记为摘要
-q, - quiet 只验证配置，不输出。 当配置正确时，不输出任何内容，当文件配置错误，输出错误信息

```
-services 打印服务名，一行一个
-volumes 打印数据卷名，一行一个
```

18，docker-compose create

```
docker-compose create [options] [SERVICE...]
为服务创建容器。
选项包括：
-force-recreate：重新创建容器，即使配置和镜像没有改变，不兼容-no-recreate 参数
-no-recreate：如果容器已经存在，不需要重新创建，不兼容-force-recreate 参数
-no-build：不创建镜像，即使缺失
-build：创建容器前     ，生成镜像
```

19，docker-compose exec

```
docker-compose  exec  [options] SERVICE COMMAND [ARGS...]
选项包括：
-d 分离模式，后台运行命令。
-privileged 获取特权。
-user USER 指定运行的用户。
-T 禁用分配 TTY，默认 docker-compose  exec 分配 TTY。
-index=index，当一个服务拥有多个容器时，可通过该参数登陆到该服务下的任何服务，例如：docker-compose  exec  -index=1
web  /bin/bash  ，web 服务中包含多个容器
```

20，docker-compose port

```
docker-compose port [options] SERVICE PRIVATE_PORT
显示某个容器端口所映射的公共端口。
选项包括：
-protocol=proto，指定端口协议，TCP（默认值）或者 UDP
-index=index，如果同意服务存在多个容器，指定命令对象容器的序号（默认为 1）
```

21，docker-compose push

```
docker-compose push [options] [SERVICE...]
推送服务依的镜像。
```

```
选项包括：
 - ignore-push-failures 忽略推送镜像过程中的错误
```

22，docker-compose stop

```
docker-compose stop [options] [SERVICE...]
显示各个容器运行的进程情况。
```

23，docker-compose uppause

```
docker-compose unpause [SERVICE...]
恢复处于暂停状态中的服务。
```

三，Docker-compose 模板文件

1，Docker-compose 模板文件简介

Compose 允许用户通过一个 docker-compose.yml 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。
Compose 模板文件是一个定义服务、网络和卷的 YAML 文件。Compose 模板文件默认路径是当前目录下的 docker-compose.yml，可以使用.yml 或.yaml 作为文件扩展名。
Docker-Compose 标准模板文件应该包含 version、services、networks 三大部分，最关键的是 services 和 networks 两个部分。

举例

```
version: '2'
services:
    web:
        image: dockercloud/hello-world
        ports:
            - 8080
        networks:
            - front-tier
            - back-tier

    redis:
        image: redis
        links:
            - web
```

```
        networks:
            - back-tier

    lb:
        image: dockercloud/haproxy
        ports:
            - 80:80
        links:
            - web
        networks:
            - front-tier
            - back-tier
        volumes:
            -  /var/run/docker.sock:/var/run/docker.sock

networks:
    front-tier:
        driver: bridge
    back-tier:
        driver: bridge
```

Compose 目前有三个版本分别为 Version 1，Version 2，Version 3，Compose 区分 Version 1 和 Version 2（Compose 1.6.0+，Docker Engine 1.10.0+）。Version 2 支持更多的指令。Version 1 将来会被弃用。

2，image

image 是指定服务的镜像名称或镜像 ID。如果镜像在本地不存在，Compose 将会尝试拉取镜像。

```
services:
        web:
                image: hello-world
```

3，build

服务除了可以基于指定的镜像，还可以基于一份 Dockerfile，在使用 up 启动时执行构建任务，构建标签是 build，可以指定 Dockerfile 所在文件夹的路径。Compose 将会利用 Dockerfile 自动构建镜像，然后使用镜像启动服务容器。

```
build:  /path/to/build/dir
```

也可以是相对路径，只要上下文确定就可以读取到 Dockerfile。

```
build: ./dir
```

设定上下文根目录，然后以该目录为准指定 Dockerfile。

```
build:
    context: ../
    dockerfile: path/of/Dockerfile
```

build 都是一个目录，如果要指定 Dockerfile 文件需要在 build 标签的子级标签中使用 dockerfile 标签指定。
如果同时指定 image 和 build 两个标签，那么 Compose 会构建镜像并且把镜像命名为 image 值指定的名字。

4,context

context 选项可以是 Dockerfile 的文件路径，也可以是到链接到 git 仓库的 url，当提供的值是相对路径时，被解析为相对于撰写文件的路径，此目录也是发送到 Docker 守护进程的 context

```
build:
    context: ./dir
```

5,dockerfile

使用 dockerfile 文件来构建，必须指定构建路径

```
build:
    context: .
    dockerfile: Dockerfile-alternate
```

6,commond

使用 command 可以覆盖容器启动后默认执行的命令。

```
command: bundle  exec  thin -p 3000
```

7,container_name

Compose 的容器名称格式是：<项目名称><服务名称><序号>
可以自定义项目名称、服务名称，但如果想完全控制容器的命名，可以使用标签指定：

```
container_name: app
```

8,depends_on

在使用 Compose 时，最大的好处就是少打启动命令，但一般项目容器启动的顺序是有要求的，如果直接从上到下启动容器，必然会因为容器依赖问题而启动失败。例如在没启动数据库容器的时候启动应用容器，应用容器会因为找不到数据库而退出。depends_on 标签用于解决容器的依赖、启动先后的问题

```
version: '2'
services:
    web:
        build: .
        depends_on:
            - db
            - redis
    redis:
        image: redis
    db:
        image: postgres
```

上述 YAML 文件定义的容器会先启动 redis 和 db 两个服务，最后才启动 web 服务。

9,PID

pid: "host"
将 PID 模式设置为主机 PID 模式，跟主机系统共享进程命名空间。容器使用 pid 标签将能够访问和操纵其他容器和宿主机的名称空间。

10,ports

ports 用于映射端口的标签。
使用 HOST:CONTAINER 格式或者只是指定容器的端口，宿主机会随机映射端口。

```
ports:
  - "3000"
  - "8000:8000"
  - "49100:22"
```

```
5       - "127.0.0.1:8001:8001"
```

当使用 HOST:CONTAINER 格式来映射端口时，如果使用的容器端口小于 60 可能会得到错误得结果，因为 YAML 将会解析 xx:yy 这种数字格式为 60 进制。所以建议采用字符串格式。

11,extra_hosts

添加主机名的标签，会在/etc/hosts 文件中添加一些记录。

```
1  extra_hosts:
2    - "somehost:162.242.195.82"
3    - "otherhost:50.31.209.229"
```

启动后查看容器内部 hosts：

```
1  162.242.195.82    somehost
2  50.31.209.229      otherhost
```

12,volumes

挂载一个目录或者一个已存在的数据卷容器，可以直接使用 [HOST:CONTAINER]格式，或者使用[HOST:CONTAINER:ro]格式，后者对于容器来说，数据卷是只读的，可以有效保护宿主机的文件系统。

Compose 的数据卷指定路径可以是相对路径，使用 . 或者 .. 来指定相对目录。

数据卷的格式可以是下面多种形式

```
1   volumes:
2       //  只是指定一个路径，Docker 会自动在创建一个数据卷（这个路径是容器内部的）。
3       - /var/lib/mysql
4       //  使用绝对路径挂载数据卷
5       - /opt/data:/var/lib/mysql
6       //  以 Compose 配置文件为中心的相对路径作为数据卷挂载到容器。
7       - ./cache:/tmp/cache
8       //  使用用户的相对路径（~/ 表示的目录是  /home/<用户目录>/ 或者  /root/）。
9       - ~/configs:/etc/configs/:ro
10      //  已经存在的命名的数据卷。
11      - datavolume:/var/lib/mysql
```

如果不使用宿主机的路径，可以指定一个 volume_driver。

```
volume_driver: mydriver
```

13,volumes_from

从另一个服务或容器挂载其数据卷：

```
volumes_from:
      - service_name
          - container_name
```

14,dns

自定义 DNS 服务器。可以是一个值，也可以是一个列表。

```
dns：8.8.8.8
dns:
        - 8.8.8.8
            - 9.9.9.9
```

15,expose

暴露端口，但不映射到宿主机，只允许能被连接的服务访问。仅可以指定内部端口为参数，如下所示：

```
expose:
        -  "3000"
        -  "8000"
```

16,links

链接到其它服务中的容器。使用服务名称（同时作为别名），或者"服务名称:服务别名"（如 SERVICE:ALIAS），例如：

```
links:
        - db
        - db:database
        - redis
```

17,net

设置网络模式。

```
net: "bridge"
net: "none"
net: "host"
```

四，Docker-compose 模板文件示例

1，Docker-compose 模板文件编写

docker-compose.yml

```
version: '2'
services:
    web1:
        image: nginx
        ports:
            - "6061:80"
        container_name: "web1"
        networks:
            - dev
    web2:
        image: nginx
        ports:
            - "6062:80"
        container_name: "web2"
        networks:
            - dev
            - pro
    web3:
        image: nginx
        ports:
            - "6063:80"
        container_name: "web3"
        networks:
            - pro

```

```
networks:
    dev:
        driver: bridge
    pro:
        driver: bridge
```

docker-compose.yml 文件指定了 3 个 web 服务

2，启动应用

创建一个 webapp 目录，将 docker-compose.yaml 文件拷贝到 webapp 目录下，使用 docker-compose 启动应用。

```
docker-compose up -d
```

3，服务访问

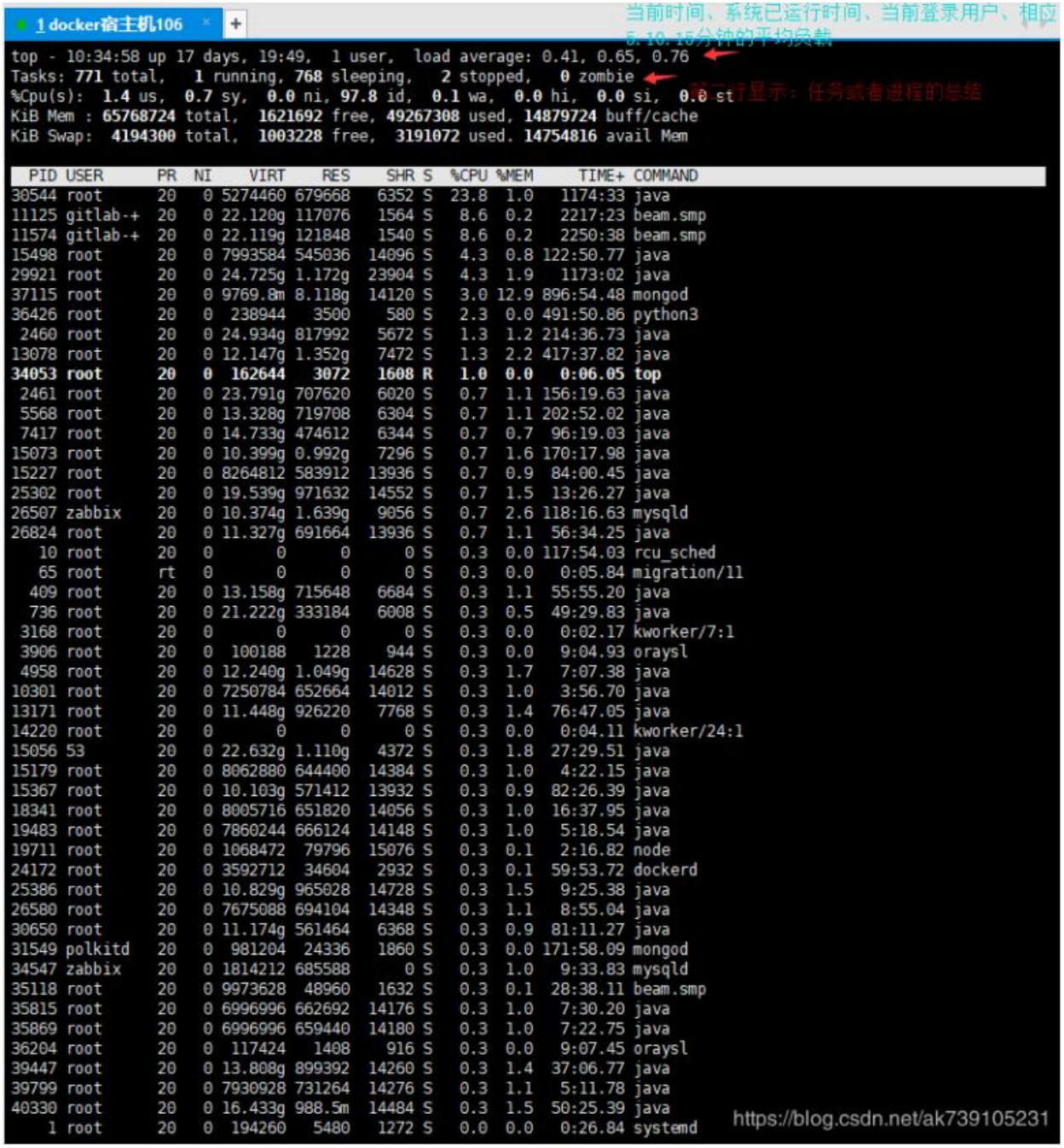
通过浏览器访问 web1，web2，web3

```
http://127.0.0.1:6061
http://127.0.0.1:6062
http://127.0.0.1:6063
```

分类: Docker, Linux, Ubuntu

1.top

top 是 linux 中自带的系统监控命令，实时监控系统各项指标


1 docker宿主机106
当前时间、系统已运行时间、当前登录用户、相应
5、10、15分钟的平均负载
第二行显示：任务或者进程的总结
top - 10:34:58 up 17 days, 19:49, 1 user, load average: 0.41, 0.65, 0.76
Tasks: 771 total, 1 running, 768 sleeping, 2 stopped, 0 zombie
%Cpu(s): 1.4 us, 0.7 sy, 0.0 ni, 97.8 id, 0.1 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 65768724 total, 1621692 free, 49267308 used, 14879724 buff/cache
KiB Swap: 4194300 total, 1003228 free, 3191072 used. 14754816 avail Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
30544 root 20 0 5274460 679668 6352 S 23.8 1.0 1174:33 java
11125 gitlab-+ 20 0 22.120g 117076 1564 S 8.6 0.2 2217:23 beam.smp
11574 gitlab-+ 20 0 22.119g 121848 1540 S 8.6 0.2 2250:38 beam.smp
15498 root 20 0 7993584 545036 14096 S 4.3 0.8 122:50.77 java
29921 root 20 0 24.725g 1.172g 23904 S 4.3 1.9 1173:02 java
37115 root 20 0 9769.8m 8.118g 14120 S 3.0 12.9 896:54.48 mongod
36426 root 20 0 238944 3500 580 S 2.3 0.0 491:50.86 python3
2460 root 20 0 24.934g 817992 5672 S 1.3 1.2 214:36.73 java
13078 root 20 0 12.147g 1.352g 7472 S 1.3 2.2 417:37.82 java
34053 root 20 0 162644 3072 1608 R 1.0 0.0 0:06.05 top
2461 root 20 0 23.791g 707620 6020 S 0.7 1.1 156:19.63 java
5568 root 20 0 13.328g 719708 6304 S 0.7 1.1 202:52.02 java
7417 root 20 0 14.733g 474612 6344 S 0.7 0.7 96:19.03 java
15073 root 20 0 10.399g 0.992g 7296 S 0.7 1.6 170:17.98 java
15227 root 20 0 8264812 583912 13936 S 0.7 0.9 84:00.45 java
25302 root 20 0 19.539g 971632 14552 S 0.7 1.5 13:26.27 java
26507 zabbix 20 0 10.374g 1.639g 9056 S 0.7 2.6 118:16.63 mysqld
26824 root 20 0 11.327g 691664 13936 S 0.7 1.1 56:34.25 java
10 root 20 0 0 0 0 S 0.3 0.0 117:54.03 rcu_sched
65 root rt 0 0 0 0 S 0.3 0.0 0:05.84 migration/11
409 root 20 0 13.158g 715648 6684 S 0.3 1.1 55:55.20 java
736 root 20 0 21.222g 333184 6008 S 0.3 0.5 49:29.83 java
3168 root 20 0 0 0 0 S 0.3 0.0 0:02.17 kworker/7:1
3906 root 20 0 100188 1228 944 S 0.3 0.0 9:04.93 oraysl
4958 root 20 0 12.240g 1.049g 14628 S 0.3 1.7 7:07.38 java
10301 root 20 0 7250784 652664 14012 S 0.3 1.0 3:56.70 java
13171 root 20 0 11.448g 926220 7768 S 0.3 1.4 76:47.05 java
14220 root 20 0 0 0 0 S 0.3 0.0 0:04.11 kworker/24:1
15056 53 20 0 22.632g 1.110g 4372 S 0.3 1.8 27:29.51 java
15179 root 20 0 8062880 644400 14384 S 0.3 1.0 4:22.15 java
15367 root 20 0 10.103g 571412 13932 S 0.3 0.9 82:26.39 java
18341 root 20 0 8005716 651820 14056 S 0.3 1.0 16:37.95 java
19483 root 20 0 7860244 666124 14148 S 0.3 1.0 5:18.54 java
19711 root 20 0 1068472 79796 15076 S 0.3 0.1 2:16.82 node
24172 root 20 0 3592712 34604 2932 S 0.3 0.1 59:53.72 dockerd
25386 root 20 0 10.829g 965028 14728 S 0.3 1.5 9:25.38 java
26580 root 20 0 7675088 694104 14348 S 0.3 1.1 8:55.04 java
30650 root 20 0 11.174g 561464 6368 S 0.3 0.9 81:11.27 java
31549 polkitd 20 0 981204 24336 1860 S 0.3 0.0 171:58.09 mongod
34547 zabbix 20 0 1814212 685588 0 S 0.3 1.0 9:33.83 mysqld
35118 root 20 0 9973628 48960 1632 S 0.3 0.1 28:38.11 beam.smp
35815 root 20 0 6996996 662692 14176 S 0.3 1.0 7:30.20 java
35869 root 20 0 6996996 659440 14180 S 0.3 1.0 7:22.75 java
36204 root 20 0 117424 1408 916 S 0.3 0.0 9:07.45 oraysl
39447 root 20 0 13.808g 899392 14260 S 0.3 1.4 37:06.77 java
39799 root 20 0 7930928 731264 14276 S 0.3 1.1 5:11.78 java
40330 root 20 0 16.433g 988.5m 14484 S 0.3 1.5 50:25.39 java
1 root 20 0 194260 5480 1272 S 0.0 0.0 0:26.84 systemd
https://blog.csdn.net/ak739105231

第一行各字段含义：

这些字段显示：
当前时间
系统已运行的时间
当前登录用户的数量
相应最近 5、10 和 15 分钟内的平均负载。
Ps：1 核 cpu 饱满负载为 1，1 以下均正常不会出现拥堵情况
总核数 = 物理 CPU 个数 X 每颗物理 CPU 的核数
总逻辑 CPU 数 = 物理 CPU 个数 X 每颗物理 CPU 的核数 X 超线程数
查看物理 CPU 个数 cat /proc/cpuinfo| grep "physical id"| sort| uniq| wc -l
查看每个物理 CPU 中 core 的个数(即核数)cat /proc/cpuinfo| grep "cpu cores"| uniq
查看逻辑 CPU 的个数 cat /proc/cpuinfo| grep "processor"| wc -l
1.物理 cpu 数：主板上实际插入的 cpu 数量，可以数不重复的 physical id 有几个（physical id）
2.cpu 核数：单块 CPU 上面能处理数据的芯片组的数量，如双核、四核等 （cpu cores）
3.逻辑 cpu 数：一般情况下，逻辑 cpu=物理 CPU 个数×每颗核数，如果不相等的话，则表示服务器的 CPU 支持超线程技术（HT：简单来说，它可使处理器中的 1 颗内核如 2 颗内核那样在操作系统中发挥作用。这样一来，操作系统可使用的执行资源扩大了一倍，大幅提高了系统的整体性能，此时逻辑 cpu=物理 CPU 个数×每颗核数 x2）
第二行字段含义：
第二行显示的是任务或者进程的总结。进程可以处于不同的状态。这里显示了全部进程的数量。除此之外，还有正在运行、睡眠、停止、僵尸进程的数量（僵尸是一种进程的状态）。
第三行字段含义：
第三行主要显示 cpu 信息：
us, user： 运行(未调整优先级的) 用户进程的 CPU
sy，system: 运行内核进程的 CPU
ni，niced：运行已调整优先级的用户进程的 CPU
Id，空闲 cpu
wa，IO wait: 用于等待 IO 完成的 CPU
hi：处理硬件中断的 CPU
si: 处理软件中断的 CPU
st：这个虚拟机被 hypervisor 偷去的 CPU（译注：如果当前处于一个 hypervisor 下的 vm，实际上 hypervisor 也是要消耗一部分 CPU 处理时间的）。
第四行和第五行字段含义：
第四行是物理内存使用：
物理内存显示如下:全部可用内存、已使用内存、空闲内存、缓冲内存。
第五行是虚拟内存使用(交换空间)：
相似地：交换部分显示的是：全部、已使用、空闲和缓冲交换空间。

第四行中使用中的内存总量（used）指的是现在系统内核控制的内存数，空闲内存总量（free）是内核还未纳入其管控范围的数量。纳入内核管理的内存不见得都在使用中，还包括过去使用过的现在可以被重复利用的内存，内核并不把这些可被重新使用的内存交还到 free 中去，因此在 linux 上 free 内存会越来越少，但不用为此担心。
如果出于习惯去计算可用内存数，这里有个近似的计算公式：第四行的 free + 第四行的 buffers + 第五行的 cached，按这个公式此台服务器的可用内存：
对于内存监控，在 top 里我们要时刻监控第五行 swap 交换分区的 used，如果这个数值在不断的变化，说明内核在不断进行内存和 swap 的数据交换，这是真正的内存不够用了。
第 7 行各字段含义：
PID：进程 ID，进程的唯一标识符
USER：进程所有者的实际用户名。
PR：进程的调度优先级。这个字段的一些值是'rt'。这意味这这些进程运行在实时态。
NI：进程的 nice 值（优先级）。越小的值意味着越高的优先级。负值表示高优先级，正值表示低优先级
VIRT：进程使用的虚拟内存。进程使用的虚拟内存总量，单位 kb。VIRT=SWAP+RES
RES：驻留内存大小。驻留内存是任务使用的非交换物理内存大小。进程使用的、未被换出的物理内存大小，单位 kb。RES=CODE+DATA
SHR：SHR 是进程使用的共享内存。共享内存大小，单位 kb
S：这个是进程的状态。它有以下不同的值:
D - 不可中断的睡眠态。
R – 运行态
S – 睡眠态
T – 被跟踪或已停止
Z – 僵尸态
%CPU：自从上一次更新时到现在任务所使用的 CPU 百分比。
%MEM：进程使用的可用物理内存百分比。
TIME+：任务启动后到现在所使用的全部 CPU 时间，精确到百分之一秒。
COMMAND：运行进程所使用的命令。进程名称（命令名/命令行）
Top 其他快捷命令：
1.按 1 显示所有逻辑 cpu 的情况
2.按 e 显示各进程占用资源单位单位
3.按 E 显示整体占用资源单位
4.进程排序默认按%cpu 进程，shift+”<”或”>”改变排序字段
5.d 指定每两次屏幕信息刷新之间的时间间隔。当然用户可以使用 s 交互命令来改变之。
6.p 通过指定监控进程 ID 来仅仅监控某个进程的状态。
7.i 使 top 不显示任何闲置或者僵死进程。
8.c 显示整个命令行而不只是显示命令名
2.查看进程的几种方式
1.ps -ef 查看当前运行的所有进程
2.ps -aux 查看正在内存中的进程

3.查看系统端口
netstat -lnpt

```
[root@localhost ~]# netstat -inpt
Kernel Interface table
Iface      MTU    RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
br0       1500  4802394      0 333100 0        740933      0      0      0 BMRU
docker0   1500        0      0      0 0             0      0      0      0 BMU
enp4s0f0  1500 76148648      0      1 0      86997319      0      0      0 BMRU
enp4s0f1  1500        0      0      0 0             0      0      0      0 BMU
enp4s0f2  1500        0      0      0 0             0      0      0      0 BMU
enp4s0f3  1500        0      0      0 0             0      0      0      0 BMU
lo       65536  4533915      0      0 0       4533915      0      0      0 LRU
veth1pl3  1500 16905322      0      0 0      16726806      0    364      0 BMRU
veth1pl7  1500   453922      0      0 0       4708248      0    366      0 BMRU
veth1pl1  1500 96436983      0      0 0      132657220      0    366      0 BMRU
veth1pl1  1500 12376703      0      0 0      12474012      0    364      0 BMRU
veth1pl2  1500 146803320      0      0 0      101114772      0    365      0 BMRU
veth1pl2  1500    51891      0      0 0       4298727      0    365      0 BMRU
veth1pl2  1500  3462758      0      0 0       7721355      0    365      0 BMRU
veth1pl3  1500  1707190      0      0 0       6402203      0    364      0 BMRU
veth1pl3  1500  7037202      0      0 0       8633096      0    364      0 BMRU
veth1pl3  1500  1222552      0      0 0       5580056      0    364      0 BMRU
veth1pl3  1500 55822311      0      0 0      88713555      0    364      0 BMRU
veth1pl3  1500 17865658      0      0 0      16232470      0    364      0 BMRU
veth1pl4  1500  5166457      0      0 0       7107706      0    364      0 BMRU
```


4.查看内存
free -mh

```
veth1pl4  1500  5166457      0      0 0       7107706      0    364      0 BMRU
[root@localhost ~]# free -mh
              total        used        free      shared  buff/cache   available
Mem:            62G         47G        1.1G        581M         14G         13G
Swap:          4.0G        3.0G        978M
[root@localhost ~]#
```

5.重定向
1.追加重定向 >>
2.覆盖重定向 >
6.查看磁盘空间

df -Th 统计磁盘空间
du -sh 统计当前目录大小
du -sh * 列出当前目录下所有文件和文件夹大小
7.kill
kill 是 linux 结束进程的命令，发送指定的信号到相应进程。
Kill 使用：kill [option] [pid]
kill [pid] 进程在退出之前可以清理并释放资源
Kill -9 [pid] 强制杀死该进程，可能会出现数据丢失情况

三种快排与四种优化

1、快速排序的基本思想：
快速排序使用分治的思想，通过一趟排序将待排序列分割成两部分，其中一部分记录的关键字均比另一部分记录的关键字小。之后分别对这两部分记录继续进行排序，以达到整个序列有序的目的。
2、快速排序的三个步骤：
(1)选择基准：在待排序列中，按照某种方式挑出一个元素，作为“基准”（pivot）
(2)分割操作：以该基准在序列中的实际位置，把序列分成两个子序列。此时，在基准左边的元素都比该基准小，在基准右边的元素都比基准大
(3)递归地对两个序列进行快速排序，直到序列为空或者只有一个元素。
3、选择基准的方式：
对于分治算法，当每次划分时，算法若都能分成两个等长的子序列时，那么分治算法效率会达到最大。也就是说，基准的选择是很重要的。选择基准的方式决定了两个分割后两个子序列的长度，进而对整个算法的效率产生决定性影响。
最理想的方法是，选择的基准恰好能把待排序序列分成两个等长的子序列
我们介绍三种选择基准的方法：(3 种)
方法(1)：固定位置
思想：取序列的第一个或最后一个元素作为基准
基本的快速排序

```
1 int SelectPivot(int arr[],int low,int high)
2 {
3     return arr[low];//选择选取序列的第一个元素作为基准
4 }
```

注意：基本的快速排序选取第一个或最后一个元素作为基准。但是，这是一直很不好的处理方法。
测试数据：
随机生成 1 百万个数字，进行排序
重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms

测试数据分析：如果输入序列是随机的，处理时间可以接受的。如果数组已经有序时，此时的分割就是一个非常不好的分割。因为每次划分只能使待排序序列减一，此时为最坏情况，快速排序沦为起泡排序，时间复杂度为 Θ(n^2)。而且，输入的数据是有序或部分有序的情况是相当常见的。因此，使用第一个元素作为枢纽元是非常糟糕的，为了避免这个情况，就引入了下面两个获取基准的方法。

方法(2)：随机选取基准

引入的原因：在待排序列是部分有序时，固定选取枢轴使快排效率底下，要缓解这种情况，就引入了随机选取枢轴

思想：取待排序列中任意一个元素作为基准

随机化算法

```
/*随机选择枢轴的位置，区间在 low 和 high 之间*/
int SelectPivotRandom(int arr[],int low,int high)
{
    //产生枢轴的位置
    srand((unsigned)time(NULL));
    int pivotPos = rand()%(high - low) + low;

    //把枢轴位置的元素和 low 位置元素互换，此时可以和普通的快排一样调用划分函数
    swap(arr[pivotPos],arr[low]);
    return arr[low];
}
```

测试数据：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms

测试数据分析：:这是一种相对安全的策略。由于枢轴的位置是随机的，那么产生的分割也不会总是会出现劣质的分割。在整个数组数字全相等时，仍然是最坏情况，时间复杂度是 O(n^2）。实际上，随机化快速排序得到理论最坏情况的可能性仅为 1/(2^n）。所以随机化快速排序可以对于绝大多数输入数据达到 O(nlogn）的期望时间复杂度。一位前辈做出了一个精辟的总结：“随机化快速排序可以满足一个人一辈子的人品需求。”

方法(3)：三数取中（median-of-three）

引入的原因：虽然随机选取枢轴时，减少出现不好分割的几率，但是还是最坏情况下还是 O(n^2），要缓解这种情况，就引入了三数取中选取枢轴

分析：最佳的划分是将待排序的序列分成等长的子序列，最佳的状态我们可以使用序列的中间的值，也就是第 N/2 个数。可是，这很难算出来，并且会明显减慢快速排序的速度。这样的中值的估计可以通过随机选取三个元素并用它们的中值作为枢纽元而得到。事实上，随机性并没有多大的帮助，因此一般的做法是使用左端、右端和中心位置上的三个元素的中值作为枢纽元。显然使用三数中值分割法消除了预排序输入的不好情形，并且减少快排大约 14%的比较次数

举例：待排序序列为：8 1 4 9 6 3 5 2 7 0
左边为：8，右边为 0，中间为 6.
我们这里取三个数排序后，中间那个数作为枢轴，则枢轴为 6
注意：在选取中轴值时，可以从由左中右三个中选取扩大到五个元素中或者更多元素中选取，一般的，会有（2t+1）平均分区法（median-of-(2t+1)，三平均分区法英文为 median-of-three）。
具体思想：对待排序序列中 low、mid、high 三个位置上数据进行排序，取他们中间的那个数据作为枢轴，并用 0 下标元素存储枢轴。
即：采用三数取中，并用 0 下标元素存储枢轴。

```
/*函数作用：取待排序序列中 low、mid、high 三个位置上数据，选取他们中间的那个数据作为枢轴*/
int SelectPivotMedianOfThree(int arr[],int low,int high)
{
    int mid = low + ((high - low) >> 1);//计算数组中间的元素的下标

    //使用三数取中法选择枢轴
    if (arr[mid] > arr[high])//目标: arr[mid] <= arr[high]
    {
        swap(arr[mid],arr[high]);
    }
    if (arr[low] > arr[high])//目标: arr[low] <= arr[high]
    {
        swap(arr[low],arr[high]);
    }
    if (arr[mid] > arr[low]) //目标: arr[low] >= arr[mid]
    {
        swap(arr[mid],arr[low]);
    }
    //此时，arr[mid] <= arr[low] <= arr[high]
    return arr[low];
    //low 的位置上保存这三个位置中间的值
    //分割时可以直接使用 low 位置的元素作为枢轴，而不用改变分割函数了
}
```

测试数据：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms

测试数据分析：使用**三数取中**选择枢轴优势还是很明显的，但是还是处理不了重复数组

4、四种优化方式：

优化 1：当待排序序列的长度分割到一定大小后，使用插入排序

原因：对于很小和部分有序的数组，快排不如插排好。当待排序序列的长度分割到一定大小后，继续分割的效率比插入排序要差，此时可以使用插排而不是快排

截止范围：待排序序列长度 N = 10，虽然在 5~20 之间任一截止范围都有可能产生类似的结果，这种做法也避免了一些有害的退化情形。摘自《数据结构与算法分析》Mark Allen Weiness 著

```
if (high - low + 1 < 10)
{
    InsertSort(arr,low,high);
    return;
}//else 时，正常执行快排
```

测试数据：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	125ms	63ms	250 ms	699516 ms

测试数据分析：针对随机数组，使用三数取中选择枢轴+插排，效率还是可以提高一点，真是针对已排序的数组，是没有任何用处的。因为待排序序列是已经有序的，那么每次划分只能使待排序序列减一。此时，插排是发挥不了作用的。所以这里看不到时间的减少。另外，三数取中选择枢轴+插排还是不能处理重复数组

优化 2：在一次分割结束后，可以把与 Key 相等的元素聚在一起，继续下次分割时，不用再对与 key 相等元素分割

举例：

待排序序列 1 4 6 7 6 6 7 6 8 6

三数取中选取枢轴：下标为 4 的数 6

转换后，待分割序列：6 4 6 7 1 6 7 6 8 6
枢轴 key：6
本次划分后，未对与 key 元素相等处理的结果：1 4 6 6 7 6 7 6 8 6
下次的两个子序列为：1 4 6 和 7 6 7 6 8 6
本次划分后，对与 key 元素相等处理的结果：1 4 6 6 6 6 6 7 8 7
下次的两个子序列为：1 4 和 7 8 7
经过对比，我们可以看出，在一次划分后，把与 key 相等的元素聚在一起，能减少迭代次数，效率会提高不少
具体过程：在处理过程中，会有两个步骤
第一步，在划分过程中，把与 key 相等元素放入数组的两端
第二步，划分结束后，把与 key 相等的元素移到枢轴周围
举例：
待排序序列 1 4 6 7 6 6 7 6 8 6
三数取中选取枢轴：下标为 4 的数 6
转换后，待分割序列：6 4 6 7 1 6 7 6 8 6
枢轴 key：6
第一步，在划分过程中，把与 key 相等元素放入数组的两端
结果为：6 4 1 6(枢轴) 7 8 7 6 6 6
此时，与 6 相等的元素全放入在两端了
第二步，划分结束后，把与 key 相等的元素移到枢轴周围
结果为：1 4 66(枢轴) 6 6 6 7 8 7
此时，与 6 相等的元素全移到枢轴周围了
之后，在 1 4 和 7 8 7 两个子序列进行快排

```
void QSort(int arr[],int low,int high)
{
    int first = low;
    int last = high;

    int left = low;
    int right = high;

    int leftLen = 0;
    int rightLen = 0;
    if (high - low + 1 < 10)
    {
        InsertSort(arr,low,high);
```

```
    return;
}
//一次分割
int key = SelectPivotMedianOfThree(arr,low,high);//使用三数取中法选择枢轴
while(low < high)
{
    while(high > low && arr[high] >= key)
    {
        if (arr[high] == key)//处理相等元素
        {
            swap(arr[right],arr[high]);
            right--;
            rightLen++;
        }
        high--;
    }
    arr[low] = arr[high];
    while(high > low && arr[low] <= key)
    {
        if (arr[low] == key)
        {
            swap(arr[left],arr[low]);
            left++;
            leftLen++;
        }
        low++;
    }
    arr[high] = arr[low];
}
arr[low] = key;
//一次快排结束
//把与枢轴 key 相同的元素移到枢轴最终位置周围
int i = low - 1;
int j = first;
while(j < left && arr[i] != key)
```

```
        {
            swap(arr[i],arr[j]);
            i--;
            j++;
        }
        i = low + 1;
        j = last;
        while(j > right && arr[i] != key)
        {
            swap(arr[i],arr[j]);
            i++;
            j--;
        }
        QSort(arr,first,low - 1 - leftLen);
        QSort(arr,low + 1 + rightLen,last);
}
```

测试数据：

随机生成 1 百万个数字，进行排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	125ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	125ms	63ms	250 ms	699516 ms
三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms

测试数据分析：**三数取中选择枢轴+插排+聚集相等元素**的组合，效果竟然好的出奇。

原因：在数组中，如果有相等的元素，那么就可以减少不少冗余的划分。这点在重复数组中体现特别明显啊。

其实这里，插排的作用还是不怎么大的。

优化 3：优化递归操作

快排函数在函数尾部有两次递归操作，我们可以对其使用尾递归优化

优点：如果待排序的序列划分极端不平衡，递归的深度将趋近于 n，而栈的大小是很有限的，每次递归调用都会耗费一定的栈空间，函数的参数越多，每次递归耗费的空间也越多。优化后，可以缩减堆栈深度，由原来的 O(n)缩减为 O(logn)，将会提高性能。

```
void QSort(int arr[],int low,int high)
{
    int pivotPos = -1;
    if (high - low + 1 < 10)
    {
        InsertSort(arr,low,high);
        return;
    }
    while(low < high)
    {
        pivotPos = Partition(arr,low,high);
        QSort(arr,low,pivot-1);
        low = pivot + 1;
    }
}
```

注意：在第一次递归后，low 就没用了，此时第二次递归可以使用循环代替

测试数据：

三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms
三数取中+插排+聚集相等元素+尾递归	110ms	32ms	32 ms	10 ms

测试数据分析：其实这种优化编译器会自己优化，相比不使用优化的方法，时间几乎没有减少

优化 4：使用并行或多线程处理子序列（略）

所有的数据测试：

随机生成 1 百万个数字，进行升序排序

重复数组：待排序数组全为 10

算法	随机数组	升序数组	降序数组	重复数组
固定枢轴	133ms	745125ms	644360ms	755422ms
随机枢轴	218ms	235ms	187 ms	701813ms
三数取中	141ms	63ms	250 ms	705110ms
三数取中+插排	131ms	63ms	250 ms	699516 ms
三数取中+插排+聚集相等元素	110ms	32ms	31ms	10 ms
三数取中+插排+聚集相等元素+尾递归	110ms	32ms	32 ms	10 ms
STL 中的 Sort 函数	125ms	27ms	31ms	8ms

概括：这里效率最好的快排组合 是：三数取中+插排+聚集相等元素,它和 **STL** 中的 **Sort** 函数效率差不多

注意：由于测试数据不稳定，数据也仅仅反应大概的情况。如果时间上没有成倍的增加或减少，仅仅有小额变化的话，我们可以看成时间差不多。

转载自： **http://blog.csdn.net/hacker00011000/article/details/52176100**