python基础

1. python中如何交换两个变量的值?
2. print的方法
3. 简述你对input()函数的理解?
4. 快速推导式
5. range和xrange的区别?
6.字符串和列表转换
7.文件读写

python基础2

1.赋值、浅拷贝和深拷贝的区别?
2. **init**和**new**的区别?
3. 说明一下os.path 和sys.path 分别代表什么?
4. Python中有日志吗?怎么使用?
5.Python2与Python3的区别?
6. Python中的作用域?

python高级1

1. 说一下字典和json的区别?
2. 什么是可变、不可变类型?
3. 复杂元素排序
4. 使用list列表实现栈和队列
5. 给定两个list A ,B，请用找出 A ,B中相同的元素，A ,B中不同的元素
6. 列举列表、字符串和字典的常用方法
7. 对以下list，按照字典的值进行排序
8. 二维数组转为一维数组
9. 生成如下一个二位数组

python高级2

1. Python中类方法、类实例方法、静态方法有何区别?
2. Python中如何动态获取和设置对象的属性?
3. Python函数调用的时候参数的传递方式是值传递还是引用传递?
4. 为什么函数名字可以当做参数用?
5. 缺省（默认）参数和不定长参数、关键字参数的理解
6.Python中is和==的区别?
7. Python的魔法方法?
8. 面向对象中怎么实现只读属性?

python高级3

1. 单例模式的应用场景?
2. 什么是闭包?
3. 装饰器的本质-闭包
4.多线程，多进程
5. 线程池、进程池的使用:
6. 进程或者线程的间通信工具--Queue
7. 什么是死锁呢?
8. CPython, Pypy, Jython
9. 同步，异步，阻塞，非阻塞?
10. Python中的进程与线程的使用场景?
11. 线程池的优点
12. 多线程、线程池能够实现并发吗?

python网络编程

1. 简述TCP和UDP的区别以及优缺点?
2.详述三次握手和四次挥手过程?
3.为什么TIME_WAIT状态需要经过2*MSL(最大报文段生存时间)才能返回到CLOSE状态?

4. 说说HTTP和HTTPS区别?
5. Post和Get请求的区别?
6. HTTP协议状态码有什么用，列出你知道的 HTTP 协议的状态码
7. HTTP常见请求头?
8. url的形式?
9. 浏览器请求动态页面过程
10. WSGI的作用
11.如何定义WSGI接口?

Flask

1. 什么是Flask，有什么优点?
2. Flask项目的创建?
3. Flask中的相关配置?
4. 常用的SQLAlchemy查询过滤器?
5. Flask-WTF是什么，有什么特点?
6. 如何防止CSRF攻击?
7. 请求图片验证码的流程?
8. 请求发送短信验证码
9. Flask的请求钩子函数?
10. G变量的生命周期
11. Flask项目中如何实现 session 信息的写入?
12. 蓝图的应用
13、Flask 中正则 URL 的实现?

Django

1. Django 创建项目的命令?
2. Django 创建项目后，项目文件夹下的组成部分（对 mvt 的理解）?
3. 对 MVC,MVT 解读的理解?
4. Django 中 models 利用 ORM 对 Mysql 进行查表的语句（多个语句）?
    F 对象
    Q 对象
    order_by 返回 QuerySet
    聚合函数
    查询支持列表生成式
    通过模型类实现关联查询
    总结:
5. django 中间件的使用?
6. 谈一下你对 uWSGI 和 nginx 的理解?
7. 说说 nginx 和 uWISG 服务器之间如何配合工作的?
8. django 开发中数据库做过什么优化?
9. 验证码过期时间怎么设置?
10. Python 中三大框架各自的应用场景?
11、django 如何提升性能（高并发）?
12、什么是 restful api，谈谈你的理解?
13. 如何设计符合 RESTful 风格的 API
14. 启动 Django 服务的方法?
15. 怎样测试 django 框架中的代码?
16. 有过部署经验？用的什么技术？可以满足多少压力?
17. Django 中哪里用到了线程?哪里用到了协程?哪里用到了进程?
18. django 关闭浏览器，怎样清除 cookies 和 session?
19. 代码优化从哪些方面考虑？有什么想法?
20. Django 中间件是如何使用的?
21. 有用过 Django REST framework 吗?
23. Jieba 分词
24. ngnix 的正向代理与反向代理?

python基础

1. python中如何交换两个变量的值?

```
def swap(a,b):
    temp = a
    a = b
    b = temp
    print(a,b)
```

```
def swap2(a,b):
    a,b = b,a
    print(a,b)

def swap3(a,b):
    """
    ^异或运算 1^1=0 1^0=1 0^0=0
    """
    a = a^b
    b = a^b  # b = (a^b)^b = a
    a = a^b  # a = (a^b)^a = b
    print(a,b)
```

2. print的方法

```
def print(value, ..., end,..., flush)
```

- end 结尾符号,默认"\n"
- flush 是否刷新缓冲区数据到控制台,python的print，默认是将数据写入缓冲区，缓冲区有默认大小，只有数据大小超出缓冲区限制或者遇到换行符才会写入到控制台

```
import time
def main():
    for i in range(10):
        print("*",end="")
        time.sleep(1)

if __name__ == '__main__':
    main()
```

3. 简述你对input()函数的理解?

- 在Python3中，input()获取用户输入，不论用户输入的是什么，获取到的都是字符串类型的。
- 在Python2中有 raw_input()和input(), raw_input()和Python3中的input()作用是一样的，input()输入的是什么数据类型的，获取到的就是什么数据类型的。

4. 快速推导式

```
a = []
for i in range(10):
    a.append(str(i))
print(a)
a = [int(x) for x in a]
print(a)

a=[("张三",18),("赵四",19),("王五",20)]
c = {x[1]:x[0] for x in a}
print(c)
```

5. range和xrange的区别?

- python2中有range和xrange，
  range返回的是一个列表，而xrange的结果是一个生成器，前者是直接开辟一块内存空间来保存列表，后者是边循环边使用，只有使用时才会开辟内存空间，所以当列表很长时，使用xrange性能要比range好
- python3中range返回的其实是一个可迭代对象(类型是range对象),而不是列表类型。

6.字符串和列表转换

```
#字符串转列表
str1="abc"
list1 = list(str1)
print(list1)

str2 = "a-b-c"
list2 = str2.split("-")
print(list2)

# 列表转字符串
list3 = ['a', 'b', 'c']
str3 = "".join(list3)
print(str3)

# 以下代码会打印什么?
list4 = [1, 2, 3]
str4 = "*".join(list4)
print(str4)
```


7.文件读写

由于文件读写时都有可能产生IOError，一旦出错，为了保证无论是否出错都能正确地关闭文件，python中建议使用with open的方法去操作文件

```
with open(file,mode,encoding) as f:
```

注意：encoding默认为运行的操作系统的编码

建议根据文件或数据类型选择合适的mode

- 'b'二进制类型
- 'r'只读
- 'w'可读可写

```
with open("1.txt","r",encoding="utf-8") as f:
    # 读取整个文本
    f.read()
    # 按行读取文本，返回一个生成器对象，每调用一次读取一行
    f.readline()
    # 按行读取文本，返回一个列表，包含了整个文本内容
    f.readlines()
```

python基础2

1.赋值、浅拷贝和深拷贝的区别?

在Python中，对象的赋值就是简单的对象引用,比如

```
b = [1,2,3]
a = b
```

那么a和b的引用地址是一样了，指向了同一块内存地址，a即为b的一个别名

浅拷贝是对象的外层拷贝，即只进行第一层的拷贝

```
a = [[1,2],3,{"key":"value"}]
# 切片操作
b = a[:]
# 列表推导式
b = [x for x in a]
# 工厂函数
b = list(a)
# copy函数
b = copy(a)
```

在浅拷贝的情况下，只是拷贝了最外围的对象本身，内部的元素都只是拷贝了一个引用而已。

```
b[0].apppend(3)
print(b)
print(a)
```

利用copy中的deepcopy方法进行拷贝就叫做深拷贝，外围和内部元素都进行了拷贝对象本身，而不是引用。

注意：对于数字，字符串和其他原子类型对象等，没有被拷贝的说法，即便是用深拷贝，查看id的话也是一样的。比如一个列表变量值包含原子类型对象，即使采用了深拷贝，也和浅拷贝的一样

2. init和new的区别?

1. **init** 通常用于初始化一个新实例，控制这个初始化的过程，比如添加一些属性， 做一些额外的操作，发生在类实例被创建完以后。它是实例级别的方法。
2. **new** 通常用于控制生成一个新实例的过程。它是类级别的方法。

通常，可以利用**new**重写来实现一个单例模式

```
class Singleton(object):
    instance = None
    def __new__(cls):
        # 关键在于这，每一次实例化的时候，我们都只会返回这同一个instance对象
        if not instance:
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance
```

3. 说明一下os.path 和sys.path 分别代表什么?

1. os.path 是一个模块，用来处理目录、路径相关的模块。

```
os.path.join(path1[, path2[, ...]])  #将多个路径组合后返回，第一个绝对路径之前的参数将被忽略。
os.path.abspath(path)  #返回绝对路径
os.path.split(path)    #将path分割成目录和文件名二元组返回
os.path.dirname(path)  #返回path的目录。其实就是os.path.split(path)的第一个元素
```

2. sys.path 是一个列表，返回解释器相关的目录列表、环境变量、注册表等初始化信息

```
import sys
sys.path.insert(0, '引用模块的地址')
# 通常用来实现 动态导入模块
```

4. Python中有日志吗?怎么使用?

Python自带logging模块，调用 logging.basicConfig()方法，配置需要的日志等级和相应的参数， Python 解释器会按照配置的参数生成相应的日志。

```
INFO
DEBUG
WARNING
ERROR
CRITICAL
```

5.Python2与Python3的区别?

1. print
    - py2：print语句，语句就意味着可以直接跟要打印的东西，如果后面接的是一个元组对象，直接打印
    - py3：print函数，函数就以为这必须要加上括号才能调用，如果接元组对象，可以接收多个位置参数，并可以打印

```
# py2
>>> print "hello", "world"
('hello', 'world')
# py3
>>> print("hello", "world")
hello world
```

2. 输入函数
    - py2：raw_input() input()
    - py3：input()
3. 1/2的结果
    - py2：返回0
    - py3：返回0.5，没有了int和long的区别
4. 编码
    - py2：默认编码ascii
    - py3：默认编码utf-8

    > 为了在py2中使用中文，在头部引入coding声明，不推荐使用

5. 字符串
    - py2：unicode类型表示字符串序列，str类型表示字节序列
    - py3:：str类型表示字符串序列，byte类型表示字节序列
6. True和False
    - py2：True 和 False 在 Python2 中是两个全局变量，可以为其赋值或者进行别的操作，初始数值分别为1和0，虽然修改是违背了python设计的原则，但是确实可以更改
    - py3：修正了这个变量，让True或False不可变
7. nonlocal py2：没有办法在嵌套函数中将变量声明为一个非局部变量，只能在函数中声明全局变量

   py3：nonlocal方法实现了，示例如下

```
def func():
    c = 1
    def foo():
        nonlocal c
        c = 12
    foo()
    print(c)
func()   # 12
```

6. Python中的作用域?

Python中，一个变量的作用域总是由在代码中被赋值的地方所决定。当Python遇到一个变量的话 它会按照这的顺序进行搜索： 本地作用域(Local)--->当前作用域被嵌入的本地作用域(Enclosing locals)--->全局/模块作用域(Global)--->内置作用域(Built-in)

python高级1

1. 说一下字典和json的区别?

字典是一种数据结构，json是一种数据的表现形式，字典的key值只要是能hash的就行，json的 必须是字符串。

2. 什么是可变、不可变类型?

- 可变不可变指的是内存中的值是否可以被改变，
- 不可变类型指的是对象所在内存块里面的值不可以改变，有数值、字符串、元组；
- 可变类型则是可以改变，主要有列表、字典。

3. 复杂元素排序

```
from operator import itemgetter, attrgetter
class Student(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return repr((self.name, self.age))
stu1 = Student("a", 12)
stu2 = Student("b", 5)
stu3 = Student("c", 18)

# 列表元素是实例对象
list1 = [stu1, stu2, stu3]
list1.sort(key=attrgetter("age"))
# list1 = sorted(list1, key=attrgetter("age"))
print(list1)

# 字典排序
dict1 = {
    "item1": 12,
    "item2": 16,
    "item0": 22
}
list1 = sorted(dict1.items(), key=itemgetter(1))
print(list1)

# 列表中是字典元素
list2 = [{"name": "a", "age": 12},
         {"name": "b", "age": 18},
         {"name": "c", "age": 6}]
list2 = sorted(list2, key=itemgetter("age"))
print(list2)
```

4. 使用list列表实现栈和队列

```
class Stack(object):
```

```
    def __init__(self):
        self.data = []

    def push(self,val):
        self.data.append(val)

    def pop(self):
        if len(self.data):
            return self.data.pop()
        else:
            return None

class Queue(object):
    def __init__(self):
        self.data = []

    def is_empty(self):
        return len(self.data) == 0

    def push(self,val):
        self.data.append(val)

    def pop(self):
        if self.is_empty():
            return None
        return self.data.pop(0)
```

5. 给定两个list A ,B，请用找出 A ,B中相同的元素，A ,B中不同的元素

- A、B 中相同元素：print(set(A)&set(B))
- A、B 中不同元素：print(set(A)^set(B))

6. 列举列表、字符串和字典的常用方法

```
list.append()、insert(index)
list.pop(index)
list.index(item)
list.find(item)

str.find("ab") # 找不到返回-1
str.index("ab") # 找不到报错
```

7. 对以下list，按照字典的值进行排序

```
list1 = [{"a": 3},{"b": 2},{"c": 1}]
```

思路：将列表中的字典放入一个大字典中，对大字典的items()进行排序，将结果再转为列表

```
from operator import itemgetter
list1 = [{"a": 12},
         {"b": 18},
         {"c": 6}]

dict1 = {key: value for x in list1 for key, value in x.items()}
list2 = sorted(dict1.items(), key=itemgetter(1))
list3 = [{x[0]: x[1]} for x in list2]
print(list3)
```

8. 二维数组转为一维数组

```
list1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
list2 = [y for x in list1 for y in x]
print(list2)
```

9. 生成如下一个二位数组

`[[1, 2, 3], [2, 4, 6], [3, 6, 9]]`

```
matrix = [[i*j for i in range(1,4)] for j in range(1,4)]
print(matrix)
```

python高级2

1. Python中类方法、类实例方法、静态方法有何区别?

```
class Test(object):
    def __init__(name,age):
        self.name = name
        self.age = age

    @classmethod
    def test1(cls,name): # cls,类对象
        obj = cls(name,18) # 类对象可以创建实例对象

    @staticmethod
    def test2():
        pass

    def test3(self): # self,实例对象
        pass

test = Test("sss",19)
```

- 类方法：是类对象的方法，在定义时需要在上方使用“@classmethod”进行装饰，形参为 cls，表示类对象，类对象和实例对象都可调用；

- 类实例方法：是类实例化对象的方法，只有实例对象可以调用，形参为self，指代对象本身；
- 静态方法：是一个任意函数，在其上方使用@staticmethod”进行装饰，可以用对象直接调用，静态方法实际上跟该类没有太大关系

2. Python中如何动态获取和设置对象的属性?

```
if hasattr(Parent, 'x'):  # hasattr(obj,"") 判断实例对象是否有指定实例属性或者实例方法、类方法、静方法、类属性
    print(getattr(Parent, 'x'))  # getattr(obj,"") 从实例对象中获取指定实例属性或者实例方法、类方法、静方法、类属性
    setattr(Parent, 'x', 3)   # setattr(obj,"","") 为实例对象设置指定实例属性或者实例方法
    print(getattr(Parent, 'x'))
```

3. Python函数调用的时候参数的传递方式是值传递还是引用传递?

- 不可变类型的参数用**值**传递
- 可变类型的参数用**引用**传递

4. 为什么函数名字可以当做参数用?

Python中一切皆对象，函数名是函数在内存中的空间，也是一个对象。

5. 缺省（默认）参数和不定长参数、关键字参数的理解

```
def test(name="xiaoming"):
    print(name)

test()
test("小红")
```

- 缺省参数指在调用函数的时候没有传入参数的情况下，调用参数设定的默认值,如果传入参数，会使用传入的参数值
- `*args` 是不定长参数，他可以表示输入参数是不确定的，可以是任意多个。
- `**kwargs` 是关键字参数，赋值的时候是以键 = 值的方式，

```
def test1(*args,**kwargs): # 函数声明，表示该函数可以接收不定长的参数和关键字参数
    test1(*args,**kwargs) # 拆包
```

6.Python中is和==的区别?

- is 判断的是 a对象是否就是 b对象，是通过 id来判断的。
- ==判断的是 a对象的值是否和b对象的值相等，是通过 value 来判断的。

7. Python的魔法方法?

- **init** 初始化方法，当一个实例被创建的时候初始化的方法
- **new** 才是实例化对象调用的第一个方法
- **call** 允许一个类的实例像函数一样被调用

- **getitem** 定义获取容器中指定元素的行为，相当于 self[key]
- **getattr** 定义当用户试图访问一个不存在属性的时候的行为
- **setattr** 定义当一个属性被设置的时候的行为
- **getattribute** 定义当一个属性被访问的时候的行为
- **del** 定义当一个类的实例对象内存被销毁前的行为

8. 面向对象中怎么实现只读属性?

- 属性私有化
- property

```
class Test(object):
    def __init__(self):
        self.__age = 18

    def get_age(self):
        return self.__age

    @property
    def age(self):
        return self.__age

test = Test()
test.get_age()
```


python高级3

1. 单例模式的应用场景?

单例模式应用的场景一般发现在以下条件下

1. 资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如日志文件，应用配置。
2. 控制资源的情况下，方便资源之间的互相通信。如线程池等。
    - 网站的计数器
    - 应用配置
    - 多线程池
    - 数据库配置，数据库连接池
    - 应用程序的日志应用

2. 什么是闭包?

在函数内部嵌套了一个函数，内函数使用了外函数的局部变量，并且外函数返回了内函数的引用，这样就构成了一个闭包

```
def outfunc(a,b):
    def innerfunc(x):
        return x*a+b
    return innerfunc
```

3. 装饰器的本质-闭包

装饰器的作用：在不违反开放封闭的原则下对现有的函数进行功能扩充，比如插入日志、性能测试、事务处理、缓存、 权限的校验等场景，有了装饰器就可以抽离出大量的与函数功能本身无关的雷同代码。

函数计时的装饰器：

```
import time
def timeit(func):
    def wrapper():
        start = time.time()
        func()
        end =time.time()
        print('used:', end - start)
    return wrapper

 @timeit
 def foo():
    for i in range(10000):
        pass
foo()

# foo = timeit(foo)
```

程序运行到装饰器的时候，会立即对下面的函数进行装饰 带参数的装饰器：

4.多线程，多进程

多线程的使用：

```
def func(x):
    print(x)
t= threading.Thread(target=func,args=(12,))
# 线程启动
t.start()
# 主进程阻塞，等待子进程的退出
t.join()
# 设置线程为主线程的守护线程
t.setDaemon()
```

多进程的使用

```
from multiprocessing import Process
def func(x):
    print(x)
p = Process(target=func,args=(12,))
p.start()# 启动子进程实例(创建子进程)
p.is_alive()# 判断进程子进程是否还在活着
p.join(timeout)# 是否等待子进程执行结束，或者等待多少秒
p.terminate()# 不管任务是否完成，立即终止子进程
p.daemon = True # 设置守护进程
```

5. 线程池、进程池的使用：

```
# 进程池
from multiprocessing import Pool
# 线程池
from multiprocessing.dummy import Pool

pool = Pool(5) # 同时最大运行线程或者进程个数

def _excute(x,y):
        print(x+y)

def _callback(self, temp): # temp必有参数
    print("线程或进程任务完成后的回调")

pool.apply_async(target=_excute, callback=_callback)

poll.close()# 关闭Pool，使其不再接受新的任务；
poll.terminate()# 不管任务是否完成，立即终止；
poll.join()# 主进程阻塞，等待子进程的退出,必须在close或terminate之后使用
```

6. 进程或者线程的间通信工具--Queue

```
# 线程池、多线程 使用的Queue
from queue import Queue
# 多进程使用的queue
from multiprocessing import JoinableQueue as Queue
# 进程池使用的queue
from multiprocessing import Manager
queue = Manager().Queue()

queue = Queue()
queue.qsize()# 返回当前队列包含的消息数量。
queue.empty()# 如果队列为空，返回True,反之False。
queue.full()# 如果队列满了，返回True，反之False。

queue.put(item, block=True, timeout=None)
"""将item消息写入队列，block默认值为True；
```

```
如果block使用默认值，且没有设置timeout（单位秒），消息列队如果已经没有空间可写入，此时程序将被阻塞（停在写入状态），直到从消息列队腾出空间为止，如果设置了timeout，则会等待timeout秒，若还没空间，则抛出"Queue.Full"异常；

如果block值为False，消息列队如果没有空间可写入，则会立刻抛出"Queue.Full"异常；
"""
queue.get(item, block=True, timeout=None)
"""
获取队列中的一条消息，然后将其从队列中移除，block默认值为True。如果block使用默认值，且没有设置timeout（单位秒），消息列队如果为空，此时程序将被阻塞（停在读取状态），直到从消息列队读到消息为止，如果设置了timeout，则会等待timeout秒，若还没读取到任何消息，则抛出"Queue.Empty"异常；
如果block值为False，消息列队如果为空，则会立刻抛出"Queue.Empty"异常；
"""
# get方法并不能让queue的计数-1，必须调用task_done
queue.task_done()

queue.join() # 主程序阻塞，等待队列计数为0时才继续向下执行
```

7. 什么是死锁呢?

若干子线程在系统资源竞争时，都在等待对方对某部分资源解除占用状态，结果是谁也不愿先解锁，互相干等着，程序无法执行下去，这就是死锁。

GIL全局解释器锁(cpython): 限制多线程同时执行，保证同一时刻只有一个线程执行，所以cpython里的多线程其实是伪多线程! 所以Python里常常使用协程技术来代替多线程，协程是一种更轻量级的线程，进程和线程的切换时由系统决定，而协程由我们程序员自己决定，而模块gevent下切换是遇到了耗时操作才会切换。

三者的关系：进程里有线程，线程里有协程

8. CPython, Pypy, Jython

1.CPython

> CPython是用C语言实现Pyhon，是目前应用最广泛的解释器。Python最新的语言特性都是在这个上面先实现，Linux，OS X等自带的也是这个版本，CPython是官方版本加上对于C/Python API的全面支持，基本包含了所有第三方库支持，例如Numpy，Scipy等。但是CPython有几个缺陷，一是全局锁使Python在多线程效能上表现不佳，二是CPython无法支持JIT（即时编译），导致其执行速度不及Java和Javascipt等语言

2.Pypy

> Pypy是用Python自身实现的解释器。针对CPython的缺点进行了各方面的改良，性能得到很大的提升。最重要的一点就是Pypy集成了JIT。但是，Pypy无法支持官方的C/Python API，导致无法使用例如Numpy，Scipy等重要的第三方库。

3.Jython

> Jython是将Python code在JVM上面跑和调用java code的解释器。

9. 同步，异步，阻塞，非阻塞?

- 同步：多个任务之间有先后顺序执行，一个执行完下个才能执行。
- 异步：多个任务之间没有先后顺序，可以同时执行
- 阻塞：卡住调用者，调用者不能继续往下执行
- 非阻塞：调用者不会卡住，可以继续执行，就是说非阻塞的。

同步异步相对于多任务而言，阻塞非阻塞相对于代码执行而言。

10. Python中的进程与线程的使用场景?

- 多进程适合在 CPU 密集型操作(cpu 操作指令比较多，如位数多的浮点运算)。
- 多线程适合在 IO 密集型操作(读写数据操作较多的，比如爬虫、网络编程)。

11. 线程池的优点

1. 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
2. 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
3. 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控

12. 多线程、线程池能够实现并发吗?

由于CPython的GIL(全局解释器锁）的限制，同一时刻只有一个线程被执行，而且不能利用CPU多核的特性 线程执行时，会为当前线程上锁，其他线程等待，遇到以下两种情况会释放锁，去执行其他线程任务 1. 线程遇到阻塞情况 2. 调用了 time.sleep() 并不是真的并发执行，只是速度太快，看上去像并发

python网络编程

1. 简述TCP和UDP的区别以及优缺点?

UDP是面向无连接的通讯协议，UDP数据包括目的端口号和源端口号信息。

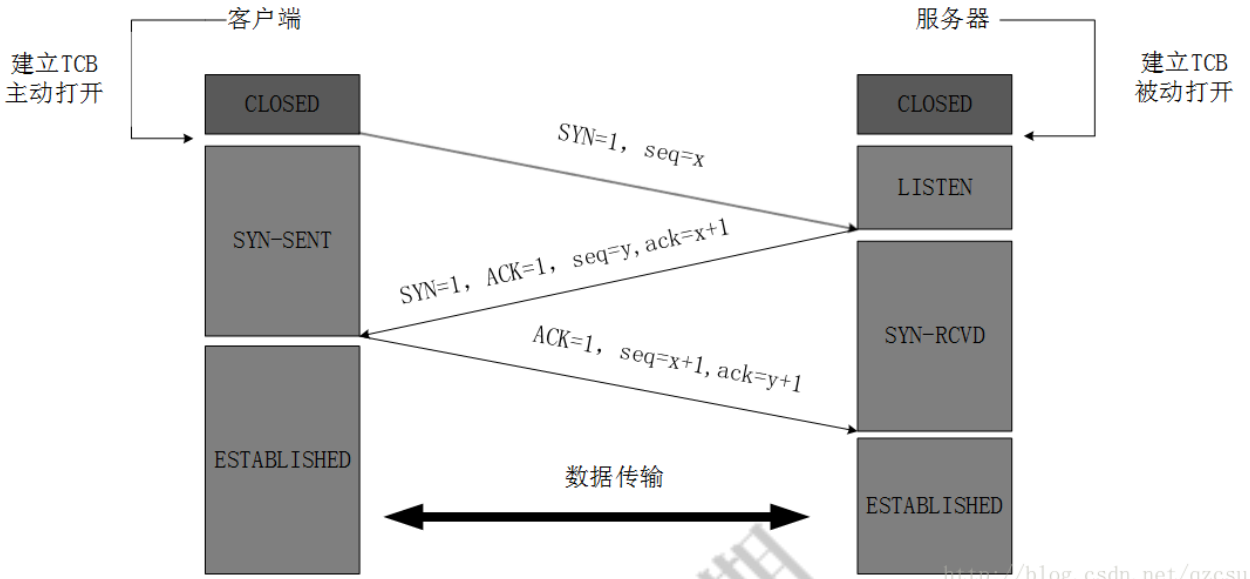
- 优点：UDP速度快、操作简单、要求系统资源较少，由于通讯不需要连接，可以实现广播发送
- 缺点：UDP传送数据前并不与对方建立连接，对接收到的数据也不发送确认信号，发送端不知道数据是否会正确接收，也不重复发送，不可靠。

TCP是面向连接的通讯协议，通过三次握手建立连接，通讯完成时四次挥手

- 优点：TCP在数据传递时，有确认、窗口、重传、阻塞等控制机制，能保证数据正确性，较为可靠。
- 缺点：TCP相对于UDP速度慢一点，要求系统资源较多。

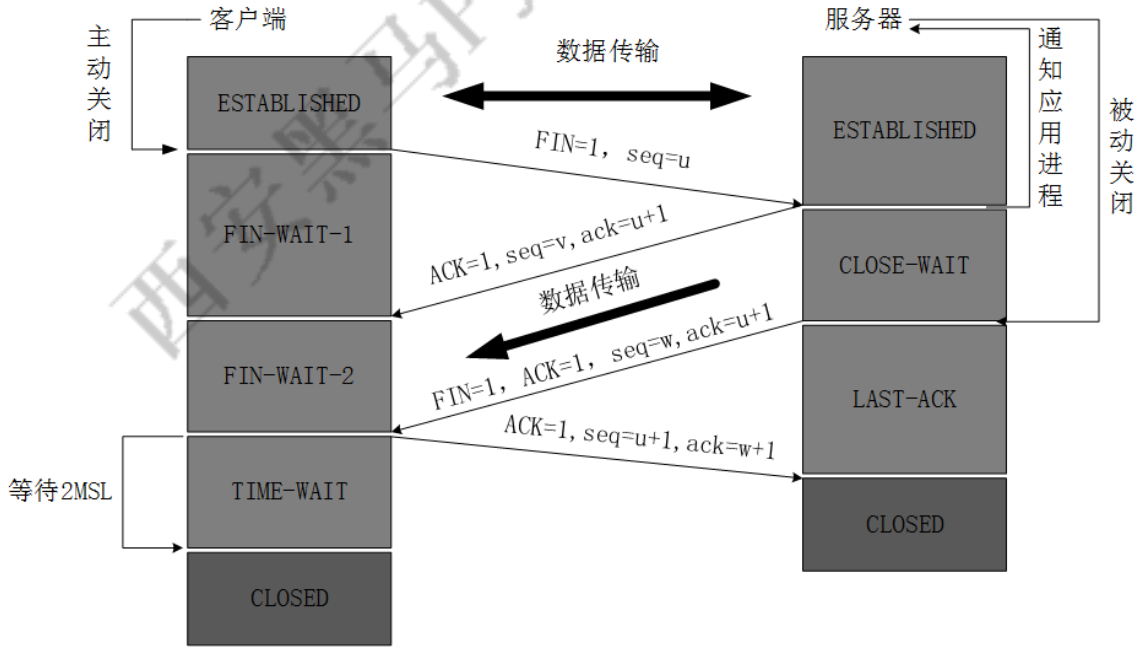
2.详述三次握手和四次挥手过程?

三次握手：



1. 建立连接时，客户端发送SYN包到服务器
2. 服务器收到请求后，会向客户端发送一个SYN包，即SYN+ACK包，此时服务器进入SYN_RECV状态。
3. 客户端收到服务器的SYN+ACK包，向服务器发送一个序列号(seq=x+1)，确认号为ack(客户端)=y+1，此包发送完毕，客户端和服务器进入ESTAB_LISHED(TCP连接成功)状态，完成三次握手

四次挥手：



1. 首先，客户端发送一个FIN，用来关闭客户端到服务器的数据传送，然后等待服务器的确认
2. 服务器收到这个FIN，它发送一个ACK，确认ack为收到的序号加一。
3. 关闭服务器到客户端的连接，发送一个FIN给客户端
4. 客户端收到FIN后，并发回一个ACK报文确认，并将确认序号seq设置为收到序号加一。客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2*MSL（最长报文段寿命）的时间后，才

进入CLOSED状态。
5. 服务器只要收到了客户端发出的确认，当前连接立即进入CLOSED状态。

3.为什么TIME_WAIT状态需要经过2*MSL(最大报文段生存时间)才能返回到CLOSE状态?

虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME_WAIT状态就是用来重发可能丢失的ACK报文。

4. 说说HTTP和HTTPS区别?

1. https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
2. http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
3. http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443
4. http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比http协议安全

5. Post和Get请求的区别?

1. 最直观的就是语义上的区别，get用于获取数据，post用于提交数据。
2. get参数有长度限制（受限于url长度，具体的数值取决于浏览器和服务器的限制），而post无限制
3. GET请求，请求的数据会附加在URL之后，以?分割URL和传输数据，多个参数用&连接，而POST请求会把请求的数据放置在HTTP 请求体中。
4. 更多资料，请查看post请求和get请求的区别

6. HTTP协议状态码有什么用，列出你知道的 HTTP 协议的状态码

通过状态码告诉客户端服务器的执行状态，以判断下一步该执行什么操作。 常见的状态机器码有：

1. 100-199：表示服务器成功接收部分请求，要求客户端继续提交其余请求才能完成整个处理过程。
2. 200-299：表示服务器成功接收请求并已完成处理过程，常用200（OK请求成功）。
3. 300-399：为完成请求，客户需要进一步细化请求，如302（所有请求页面已经临时转移到新的url），304、307（使用缓存资源）。
4. 400-499：客户端请求有错误，如常用404（服务器无法找到被请求页面），403（服务器拒绝访问，权限不够）。
5. 500-599：服务器端出现错误，常用500（请求未完成，服务器遇到不可预知的情况）

7. HTTP常见请求头?

1. Host(主机和端口号)
2. Connection(链接类型)
3. Upgrade-Insecure-Requests(升级为HTTPS请求)
4. User-Agent(浏览器名称)
5. Accept(传输文件类型)
6. Referer(页面跳转处)
7. Accept-Encoding（文件编解码格式）
8. Cookie （Cookie）
9. x-requested-with:XMLHttpRequest (是Ajax异步请求)

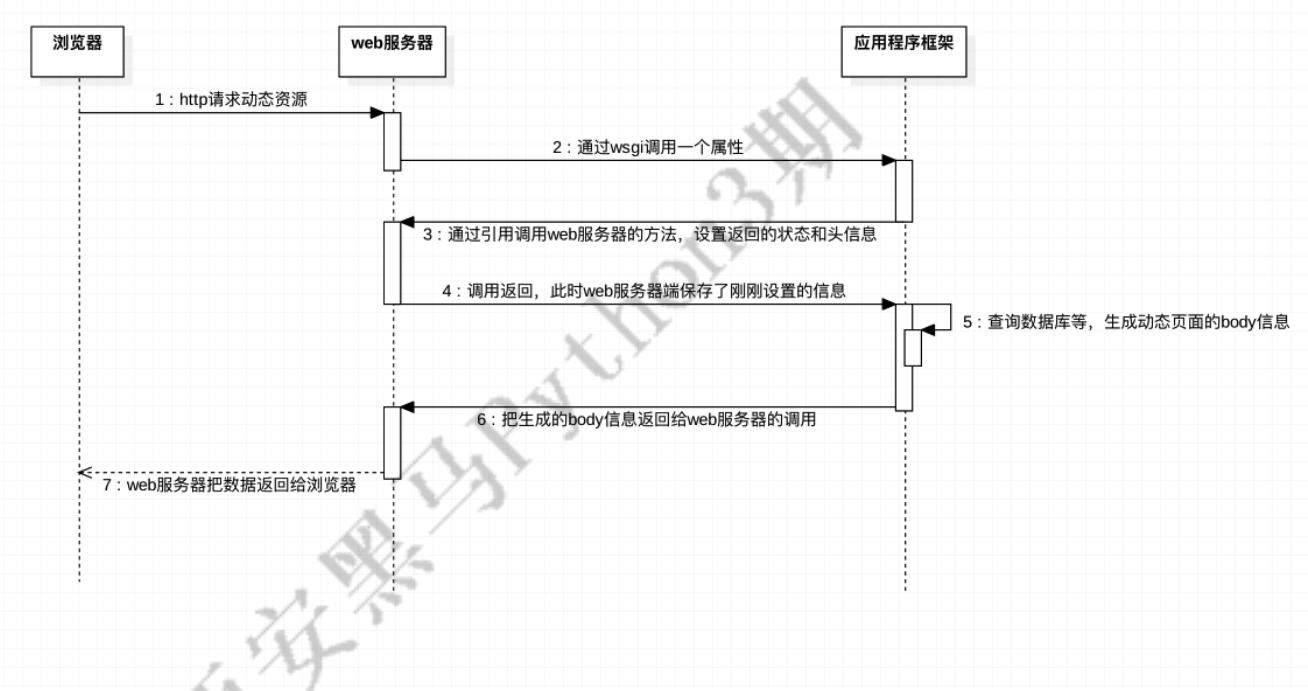
8. url的形式?

形式： scheme://host[:port#]/path/.../[?query-string][#anchor]

- scheme：协议(例如：http， https， ftp)
- host：服务器的IP 地址或者域名
- port：服务器的端口（如果是走协议默认端口，80 or 443）
- path：访问资源的路径
- query-string：参数，发送给http服务器的数据
- anchor：锚（跳转到网页的指定锚点位置）

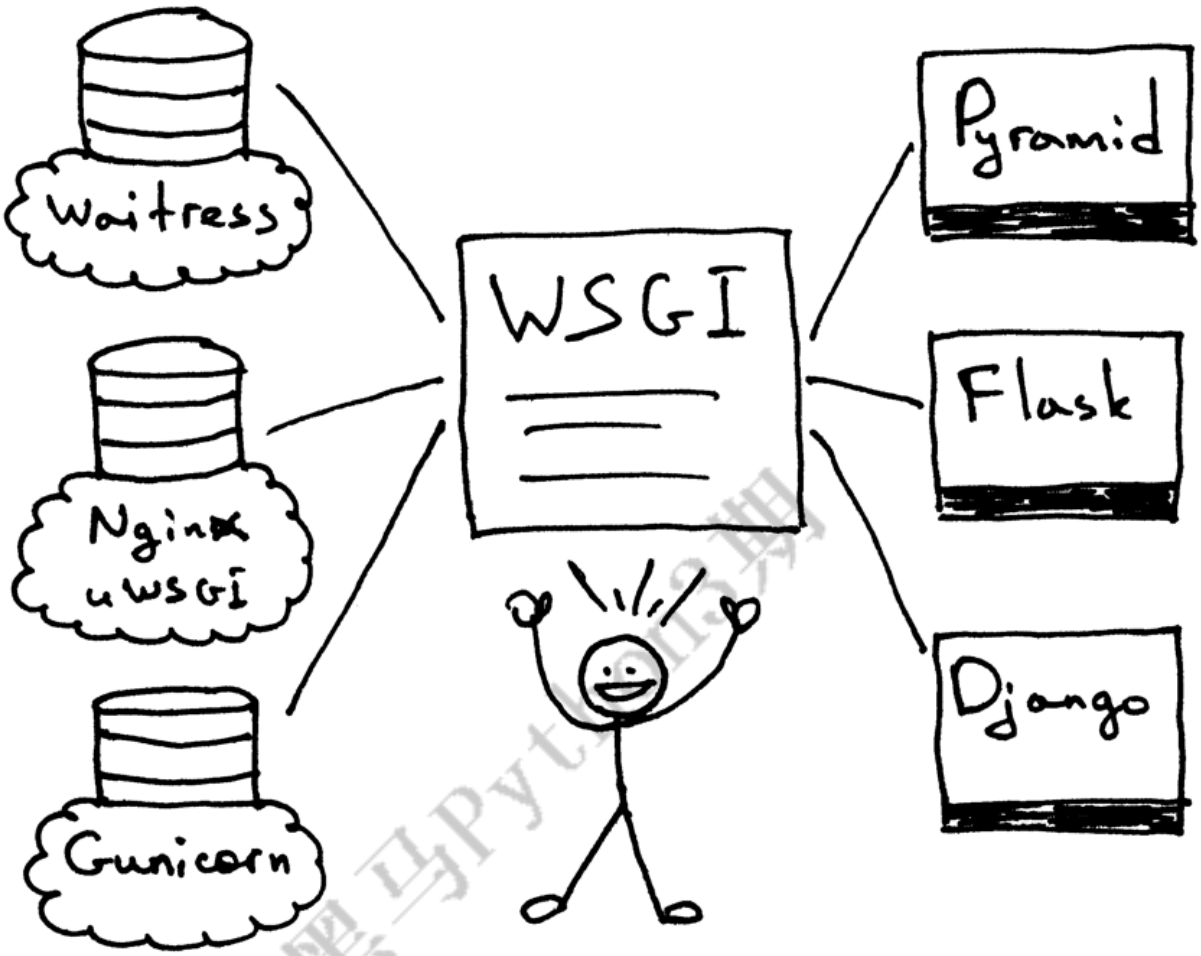
http://localhost:4000/file/part01/1.2.html

9. 浏览器请求动态页面过程



10. WSGI的作用

web框架和服务器之间的通信协议



WSGI允许开发者将选择web框架和web服务器分开。可以混合匹配web服务器和web框架，选择一个适合的配对。比如,可以在Gunicorn 或者 Nginx/uWSGI 或者 Waitress上运行 Django, Flask。

11.如何定义WSGI接口?

WSGI接口定义非常简单，它只要求Web开发者实现一个函数，就可以响应HTTP请求:

```
# 这是web服务器
def start_response(status, headers):
    # 保存 响应头信息
    self.status = status
    self.headers = headers

# 当接收到浏览器的一个请求并判断是动态资源的请求时；
body = framework.application(env,start_response)

response = self.headers + body
```

```
# 这是框架
def application(environ, start_response):
    """
     application是在框架中定义，在服务器中被调用
     start_response 是服务器中的一个设置响应头信息的函数
    """
    start_response('200 OK', [('Content-Type', 'text/html')])
    # 查询数据库等操作
    return 'Hello World!'
```

上面的`application()`函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数：

- `environ`：一个包含所有HTTP请求信息的dict对象；
- `start_response`：一个发送HTTP响应的函数。

`application()`函数必须由WSGI服务器来调用

Flask

1. 什么是Flask，有什么优点?

概念解释

- Flask是一个Web框架，就是提供一个工具，库和技术来允许你构建一个Web应用程序。这个Web应用程序可以是一些Web页面，博客，wiki，基于Web的日里应用或商业网站。

优点 Flask属于微框架（micro-framework)这一类别，微架构通常是很小的不依赖外部库的框架。

- 框架很轻量
- 更新时依赖小
- 专注于安全方面的bug

Flask的依赖

- Werkzeug 一个WSGI工具包
- jinja2 模板引擎

2. Flask项目的创建?

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(host='127.0.0.1',port=5000)
```

`__name__`的作用：通过传入这个名字确定程序的根目录，以便获得静态文件和模板文件的目录

3. Flask中的相关配置?

```
class Config(object):
    """项目配置"""
    DEBUG = True

    """mysql数据库配置"""
    SQLALCHEMY_DATABASE_URI = "mysql://root:root@127.0.0.1/infomation01"
    # 动态追踪修改设置，如未设置只会提示警告
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    # 设置每次请求结束后会自动执行一次commit操作
    SQLALCHEMY_COMMIT_ON_TEARDOWN = True

    """redis 数据库配置"""
    REDIS_HOST = "127.0.0.1"
    REDIS_PORT = 6379

    """session设置"""
    # session加密key
    SECRET_KEY = "itcast"
    # session保存到redis中
    SESSION_TYPE = "redis"
    # 指定redis
    SESSION_REDIS = StrictRedis(REDIS_HOST, REDIS_PORT)
    # 设置签名
    SESSION_USE_SIGNER = True
    # 设置非永久性
    SESSION_PERMANENT = False
    # 设置存储有效期
    PERMANENT_SESSION_LIFETIME = 60 * 60 * 24

    """设置日志等级"""
    LOG_LEVEL = logging.DEBUG
```


4. 常用的SQLAlchemy查询过滤器?

```
paginate = News.query.filter(News.category_id == cid)
            .order_by(News.create_time.desc())
            .paginate(page, per_page, False)
news_list = paginate.items
total_page = paginate.pages
current_page = paginate.page
```

- filter() 指定条件的查询
- all()　查询所有
- order_by() 按照指定字段排序
- group_by() 按照指定条件进行分组
- paginate() 分页查询

5. Flask-WTF是什么，有什么特点?

Flask-wtf是一个用于表单处理,校验并提供csrf验证的功能的扩展库 Flask-wtf 能把正表单免受CSRF<跨站请求伪造>的攻击

6. 如何防止CSRF攻击?

在flask项目中，可以在配置app时，开启CSRF保护：

```
CSRFPoject (app)
```

开启后，会对POST、DELETE、PUT等请求进行验证，需要满足以下条件才能放行

- cookies中存在csrf_token字段
- 请求头或者请求体中包含csrf_token字段，并且和cookies的一致

但是，CSRFPoject（app）只进行验证，设置csrf_token字段只需要我们手动实现， 我们根据验证条件，需要完成以下两步：

1. 向cookies中设置csrf_token（第一次请求肯定是get请求），统一在每次请求执行完成之后将csrf_toke写入cookie

```
@app.after_request
def set_token(response):
    # 调用函数生成 csrf_token
    csrf_token = generate_csrf()
    # 通过 cookie 将值传给前端
    response.set_cookie("csrf_token", csrf_token)
    return response
```

2. 在ajax的header中通过js代码获取浏览器cookies中的csrf_token

```
$.ajax({
    url: "/passport/login",
    method: "post",
    // 设置headers，带上 csrf_token
    headers: {
        "X-CSRFToken": getCookie(csrf_token)
    },
```

7. 请求图片验证码的流程?

1. 浏览器向后台发起获取验证码的请求，参数：UUID
2. 后台解析请求，获取UUID，生成图片验证码文字、二进制图片信息，UUID：图片验证码文字 -> redis, 返回二进制图片信息，必须设置响应头content-type:image/jpg
3. 对于imge标签，只要设置了src属性，自动向src的地址发起请求，拿到结果如果是图片的话后直接显示

```
@passport_blue.route("/image_code")
def get_image_code():
    """
    返回图片验证码
```

```
    :return:
    """
    #  1. 获取imagecode_id
    imageCodeId = request.args.get("imageCodeId", None)
    # 2.判断参数情况
    if not imageCodeId:
        return abort(403)
    #  3. 生成图片验证码
    name, text, image = captcha.generate_captcha()
    # 4. 保存验证码文字到redis
    try:
        redis_store.set("ImageCode_" + imageCodeId, text,
constants.IMAGE_CODE_REDIS_EXPIRES)
    except Exception as e:
        current_app.logger.error(e)
        abort(500)
    # 5. 返回图片验证码
    response = make_response(image)
    response.headers['Content-Type'] = "image/jpg"
    return response
```

8. 请求发送短信验证码

1. 发起短信验证码请求，参数：UUID，手机号，用户输入的图片验证码信息，ajax ， post
2. 后台解析请求，获取参数、判断参数、根据UUID从reis中取出正确的图片验证码文字进行比对
3. 如果一致，向云通信发起发送短信的请求，等待第三方返回响应，再将响应发回浏览器

```
@passport_blue.route("/sms_code", methods=["POST"])
def send_sms_code():
    """
    发送短信验证码
    :return:
    """
    # 1. 获取参数,
    params_dict = request.json
    mobile = params_dict['mobile']
    image_code = params_dict['image_code']
    image_code_id = params_dict['image_code_id']

    # 2. 检验参数
    if not all([mobile, image_code, image_code_id]):
        return jsonify(errno=RET.PARAMERR, errmsg="参数不正确")

    # 3. 从redis中获取正确的图片文字信息
    try:
        real_image_code = redis_store.get("ImageCode_" + image_code_id)
    except Exception as e:
        current_app.logger.error(e)
        return jsonify(errno=RET.DBERR, errmsg="数据系统故障")

    if not real_image_code:
```

```
        return jsonify(errno=RET.NODATA, errmsg="图片验证码已经过期")

    # 4. 对比验证码
    if real_image_code.upper() != image_code.upper():
        return jsonify(errno=RET.DATAERR, errmsg="输入的验证码错误")

    # 5. 生成短信验证码
    sms_code_str = "%06d" % random.randint(0, 999999)

    # 6. 请求第三方，发送验证码
    result = CCP().send_template_sms(mobile, [sms_code_str,
constants.SMS_CODE_REDIS_EXPIRES / 60], 1)
    if result != 0:
        return jsonify(errno=RET.THIRDERR, errmsg="发送验证码失败")

    # 7. 保存短信验证码到redis
    try:
        redis_store.set("SMS_" + mobile, sms_code_str, constants.SMS_CODE_REDIS_EXPIRES)
    except Exception as e:
        current_app.logger.error(e)
        return jsonify(errno=RET.DBERR, errmsg="数据系统故障")

    return jsonify(errno=RET.OK, errmsg="验证码已发送")
```

9. Flask的请求钩子函数?

1. before_first_request 第一次请求之前
2. before_request 每次请求之前
3. after_request 如果没有未处理的异常抛出，在每次请求后运行
4. teardown_reqeust 无论有没有异常抛出，每次请求结束之后都会运行

10. G变量的生命周期

每个请求的g都是独立的，并且在整个请求内都是可访问修改的。

详细请查看深入理解flask的上下文

11. Flask项目中如何实现 session 信息的写入?

Flask中有三个 session：

1. 数据库中的 session，例如:`db.session.add()`
2. 在 flask_session 扩展中的 session，使用：`from flask_session import Session`，使用第三方扩展的 session 可以把信息存储在服务器中，客户端浏览器中只存储 sessionid。
3. flask 自带的 session，是一个请求上下文， 使用：`from flask import session`。自带的 session 把信息base64加密后都存储在客户端的浏览器 cookie中

12. 蓝图的应用

应用场景： 项目模块化开发，每一个模块对应一个蓝图来管理视图

1. 创建 一个蓝图对象

```
blue = Blueprint("blue", __name__, url_prefix="/blue")
```

2. 在这个蓝图对象上进行操作，例如注册路由、指定静态文件夹、注册模板过滤器...

```
@blue.route('/')
def blue_index():
    return 'Welcome to my blueprint'
```

3. 在应用对象上注册这个蓝图对象

```
from xx import blue  # 注册前导入，防止出现循环导包问题
app.register_blueprint(blue)
```

13、Flask 中正则 URL 的实现?

@app.route('')中 URL 显式支持 string、int、float、path uuid any 6 种类型，隐式支持正则。

具体实现步骤为：

- 导入转换器基类：在 Flask 中，所有的路由的匹配规则都是使用转换器对象进行记录
- 自定义转换器：自定义类继承于转换器基类
- 添加转换器到默认的转换器字典中
- 使用自定义转换器实现自定义匹配规则

1. 导入转换器基类

```
from werkzeug.routing import BaseConverter
```

2. 写正则类（自定义转换器），继承 BaseConverter，将匹配到的值设置为 regex 的值。

```
# 自定义正则转换器
class RegexConverter(BaseConverter):
    def __init__(self, url_map, *args):
        super(RegexConverter, self).__init__(url_map)
        # 将接受的第1个参数当作匹配规则进行保存
        self.regex = args[0]
```

3. 添加转换器到默认的转换器字典中，并指定转换器使用时名字为: re

```
app = Flask(__name__)
# 将自定义转换器添加到转换器字典中，并指定转换器使用时名字为: re
app.url_map.converters['re'] = RegexConverter
```

4. 在url中使用正则

```python
@app.route('/user/<re("[0-9]{3}"):user_id>')
def user_info(user_id):
    return "user_id 为 %s" % user_id
```

Django

1. Django 创建项目的命令?

- django-admin startproject 项目名称
- python manage.py startapp 应用 app 名　如：`python manage.py startapp app01`

2. Django 创建项目后，项目文件夹下的组成部分（对 mvt 的理解）？

项目文件下的组成部分：

- `manage.py` 与项目进行交互的命令行工具集的接口 项目管理器 执行python manage.py来查看所有命令
  与项目同名的目录，项目的一个容器 ,包含项目的配置文件，目录名称不建议修改
- `urls.py`：url配置文件，Django项目中所有地址（页面）都需要我们自己去配置URL
- `settings.py`：项目的总配置文件，里面包含了数据库、web应用、时间等各种配置
- `init.py`：python中声明模块的文件，内容默认为空
- `wsgi.py`：python 服务器网关接口，python应用与web服务器之间的接口，在项目开发中一般不做修改

3. 对 MVC,MVT 解读的理解?

MVC简介：

- M：Model，模型，和数据库进行交互
- V：View，视图，负责产生 Html 页面
- C：Controller，控制器，接收请求，进行处理，与 M 和 V 进行交互，返回应答。

整体流程：

1. 用户点击注按钮，将要注册的信息发送给网站服务器。
2. Controller 控制器接收到用户的注册信息，Controller 会告诉 Model 层将用户的注册信息保 存到数据库
3. Model 层将用户的注册信息保存到数据库
4. 数据保存之后将保存的结果返回给 Model 模型，
5. Model 层将保存的结果返回给 Controller 控制器。
6. Controller 控制器收到保存的结果之后，或告诉 View 视图，view 视图产生一个 html 页面。
7. View 将产生的 Html 页面的内容给了 Controller 控制器。
8. Controller 将 Html 页面的内容返回给浏览器。
9. 浏览器接受到服务器 Controller 返回的 Html 页面进行解析展示

MVT简介：

- M：Model，模型，和 MVC 中的 M 功能相同，和数据库进行交互。
- V：view，视图，和 MVC 中的 C 功能相同，是核心，负责接收请求、获取数据、返回结果
- T：Template，模板，和 MVC 中的 V 功能相同，产生 Html 页面

整体流程:

1. 用户点击注册按钮，将要注册的内容发送给网站的服务器。
2. View 视图，接收到用户发来的注册数据，View 告诉 Model 将用户的注册信息保存进数据库。
3. Model 层将用户的注册信息保存到数据库中。
4. 数据库将保存的结果返回给 Model
5. Model 将保存的结果给 View 视图。
6. View 视图告诉 Template 模板去产生一个 Html 页面。
7. Template 生成 html 内容返回给 View 视图。
8. View 将 html 页面内容返回给浏览器。
9. 浏览器拿到 view 返回的 html 页面内容进行解析，展示。

4. Django 中 models 利用 ORM 对 Mysql 进行查表的语句（多个语句）？

字段查询:

- all():返回模型类对应表格中的所有数据。
- get():返回表格中满足条件的一条数据，如果查到多条数据，则抛异常：MultipleObjectsReturned， 查询不到数据，则抛异常：DoesNotExist。
- filter():参数写查询条件，返回满足条件 QuerySet 集合数据。

条件格式:

- **模型类属性名__条件名=值**

> 注意：此处是模型类属性名，不是表中的字段名

关于 filter 具体案例如下:

1. 判等 exact：

```
# 例：查询编号为1的图书。
BookInfo.objects.filter(id=1)
# 注意：如果tilter()没有参数即全部查到
BookInfo.objects.filter(id__exact=1)
# 以上两者意思一样
```

2. 模糊查询 like

```
# 例：查询书名包含'传'的图书。contains
BookInfo.objects.filter(btitle__contains = "传")
# 例：查询书名以'部'结尾的图书 endswith 开头:startswith
BookInfo.objects.filter(btitle__endswith="天")
BookInfo.objects.filter(btitle__startswith="天")
```

3. 空查询 where 字段名 isnull

```
# 例：查询书名不为空的图书。isnull
BookInfo.objects.filter(btitle__isnull=False)
```

4. 范围查询 where id in (1，3，5)

```
# 例：查询id编号为1或3或5的图书。
b = BookInfo.objects.filter(id__in=[1,2,3])
```

5. 比较查询 gt lt(less than) gte(equal) lte

```
# 例：查询编号大于3的图书。gt、gte、lt、lte：大于、大于等于、小于、小于等于
BookInfo.objects.filter(id__gt = 2)
```

6. 日期查询

```
# year、month、day、week_day、hour、minute、second：对日期时间类型的属性进行运算。
#例：查询1980年发表的图书。
BookInfo.objects.filter(bpub_date__year=1980)
BookInfo.objects.filter(bpub_date__month = 7)
# 例：查询1980年1月1日后发表的图。
b = BookInfo.objects.filter(bpub_date__gt = date(1980,1,1))
```

7. exclude:返回不满足条件的数据。 相当于sql语句中where部分的not关键字。----->返回一个QuerySet对象

```
例：查询id不为3的图书信息。
BookInfo.objects.exclude(id = 3)
```

F 对象

> 作用：用于类属性之间的比较条件

- 导入：

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

- 查询：

```
# 查询图书阅读量大于评论量图书信息。
BookInfo.objects.filter(bread__gt = F('bcomment'))
# 查询图书阅读量大于2倍评论量图书信息
BookInfo.objects.filter(bread__gt = F("bcomment")*2)
```

Q 对象

> 作用：用于查询时的逻辑条件。not and or，可以对Q对象进行&|~操作

- 导入：

`from django.db.models  import Q`

- 查询：

```
#例：查询所有图书的信息，按照id从小到大进行排序。
BookInfo.objects.order_by("id")
BookInfo.objects.all().order_by("id")
# 以上两种都可以--默认为升序
# 例：查询所有图书的信息，按照id从大到小进行排序。
BookInfo.objects.all().order_by("-id")
# 降序
# 例：把id大于3的图书信息按阅读量从大到小排序显示;
BookInfo.objects.filter(id__gt = 3).order_by("-bread")
```

order_by 返回 QuerySet

> 作用：进行查询结果进行排序。-> 返回一个QuerySet对象

```
#例：查询所有图书的信息，按照id从小到大进行排序。
BookInfo.objects.order_by("id")
BookInfo.objects.all().order_by("id")
# 以上两种都可以--默认为升序
# 例：查询所有图书的信息，按照id从大到小进行排序。
BookInfo.objects.all().order_by("-id")
# 降序
# 例：把id大于3的图书信息按阅读量从大到小排序显示;
BookInfo.objects.filter(id__gt = 3).order_by("-bread")
```

聚合函数

> 作用：对查询结果进行聚合操作。----->返回一个字典

- 导入：

```
from django.db.models import sum, count, max, min, avg
```

- aggregate：调用这个函数来使用聚合。

```
# 例：查询所有图书的数目。
BookInfo.objects.aggregate(Count("id"))
>>>>{'id__count': 4}
# 注意：返回值为一个字典 key值为查询时的条件字段名_聚合操作===>字段名_聚合操作
#例：查询所有图书阅读量的总和。
BookInfo.objects.aggregate(Sum("bread"))
{'bread__sum': 126}
```

- count 函数

> 作用：统计满足条件数据的数目 ->返回一个值

```
# 例：统计id大于3的所有图书的数目。
BookInfo.objects.filter(id__gt = 3).count()
```

- exists

  > 作用：判断查询集中是否有数据，如果有则返回True，没有则返回False。

  ```
  # 例：判断查询集b当中是否有数据
  b = BookInfo.objects.all()
  b.exists()
  ```

查询支持列表生成式

```
# 例：查询所有书的id
[book.id for book in BookInfo.objects.all()]
```

关联查询：

```
a b c
join b on a.id = b.aid join c on
```

关联查询（一对多）

在一对多关系中，一对应的类我们把它叫做一类，多对应的那个类我们把它叫做多类，我们把多类中定义的建立关联的类属性叫做关联属性。

通过对象实现关联查询：

```
# 例：查询图书id为1的所有英雄的信息。
b = BookInfo.objects.get(id=1)
b.heroinfo_set.all()
# 例：查询id为1的英雄所属图书信息。
h = HeroInfo.objects.get(id=1)
# 获取图书时有三种方法
h.hbook-----获取对应书的对象
h.hbook.id-----获取对应书的id
h.hbook_id-----获取对应书的id
```

格式：

1. 由一类的对象查询多类的时候：

   `一类的对象.多类名小写_set.all() #查询所用数据`

2. 由多类的对象查询一类的时候：

   `多类的对象.关联属性 #查询多类的对象对应的一类的对象`

3. 由多类的对象查询一类对象的id时候：

   `多类的对象. 关联属性_id`

> 总结：通过对象实现查询的时候先获取条件的对象，然后再查询最终查询的内容

通过模型类实现关联查询

```
# 例：查询图书，要求图书中英雄的描述包含'八'。
BookInfo.objects.filter(heroinfo__hcomment__contains= " 八")
# 例：查询图书，要求图书中的英雄的id大于3.
BookInfo.objects.filter(heroinfo__id__gt = 3)
# 例：查询书名为“天龙八部”的所有英雄。
HeroInfo.objects.filter(hbook__btitle="天龙八部")
```

格式：

1. 通过多类的条件查询一类的数据：

   `一类名.objects.filter(多类名小写__多类属性名__条件名)`

2. 通过一类的条件查询多类的数据：

   `多类名.objects.filter(关联属性__一类属性名__条件名)`

总结：

当一类为条件的时候，多类当中有关联属性，直接拿关联属性即可找到一类对应属性的条件

1. 显示什么就先写什么
2. 判断条件为一类还是多类，如果条件为多类--通过类名找，如果条件为一类--通过关联属性查
3. 显示什么就先写什么，Django框架会遍历先写的模型类里的内容如果条件为一类的此时先写的模型类当中有关联属性，此时就可以根据关联属性查找。

5. django 中间件的使用?

Django 在中间件中预置了六个方法，这六个方法的区别在于不同的阶段执行，对输入或输出进行干 预，方法如下：

- 初始化：无需任何参数，服务器响应第一个请求的时候调用一次，用于确定是否启用当前中间件

```
def __init__():
    pass
```

- 处理请求前：在每个请求上调用，返回 None 或 HttpResponse 对象。

```
def process_request(request):
    pass
```

- 处理视图前：在每个请求上调用，返回 None 或 HttpResponse 对象。

```
def process_view(request, view_func, view_args, view_kwargs):
    pass
```

- 处理模板响应前：在每个请求上调用，返回实现了 render 方法的响应对象。

```
def process_template_response(request, response):
    pass
```

- 处理响应后：所有响应返回浏览器之前被调用，在每个请求上调用，返回 HttpResponse 对象

```
def process_response(request, response):
    pass
```

- 异常处理：当视图抛出异常时调用，在每个请求上调用，返回一个 HttpResponse 对象。

```
def process_exception(request,exception):
    pass
```

6. 谈一下你对 uWSGI 和 nginx 的理解?

- WSGI

  它不是服务器、python模块、框架、API或者任何软件，只是一种描述web服务器 如何与web应用程序（如用Django、Flask框架写的程序）通信的规范、协议。

- uWSGI

  uWSGI是一个全功能的HTTP服务器，实现了WSGI协议、uwsgi协议、http协议等。它要做的就是把HTTP协议转化成语言支持的网络协议。比如把HTTP协议转化成WSGI协议，让Python可以直接使用。

- uwsgi

  与WSGI一样，是uWSGI服务器的独占通信协议，用于定义传输信息的类型。每一个uwsgi packet前4byte为传输信息类型的描述，与WSGI协议是两种东西

- Nginx

  Nginx是一个Web服务器其中的HTTP服务器功能和uWSGI功能很类似，但是Nginx还可以用作更多用途，比如最常用的反向代理功能。

  nginx的作用：

  1. 反向代理，可以拦截一些web攻击，保护后端的web服务器
  2. 负载均衡，根据轮询算法，分配请求到多节点web服务器
  3. 缓存静态资源，加快访问速度，释放web服务器的内存占用，专项专用

- uWSGI的适用：

  1. 单节点服务器的简易部署
  2. 轻量级，好部署

7. 说说 nginx 和 uWISG 服务器之间如何配合工作的?

1. 浏览器发起 http 请求到 nginx 服务器2
2. Nginx 根据接收到请求包，进行 url 分析，判断访问的 资源类型，
   - 如果是静态资源，直接读取静态资源返回给浏览器，
   - 如果请求的是动态资源就转交给 uwsgi 服务器，
3. uwsgi 服务器根据自身的 uwsgi 和 WSGI 协议，找到对应的 Django 框架，Django 框架下的 应用进行逻辑处理后，将返回值发送到 uwsgi 服务器
4. 然后 uwsgi 服务器再返回给 nginx
5. 最后 nginx 将返回值返回给浏览器进行渲染显示给用户。

8. django 开发中数据库做过什么优化?

1. 使用标准的数据库优化技术：

   比如给字段加索引，通过使用 `django.db.models.Field.db_index` 来给一个Django模型类的字段加索引，设置这个属性字段的 `Field.db_index=True` 。

   注：django对model中的fk和unique = True的字段将自动创建索引。

2. 理解Django中QuerySet的工作机制对数据库访问优化至关重要：

   QuerySet是懒加载的，它只有在需要的时候才会被执行，并且会将执行的结果保存在内存中。

3. 理解Django中QuerySet的缓存机制：

   QuerySet对调用方法是不执行缓存的。比如下面的两端代码，其中一个会被缓存，另一个不会：

4. 使用模板语言中的with标签：

   在视图模板中，针对QuerySet对象使用with标签，可以让数据被缓存起来使用。

5. 使用iterator()方法：

   对于缓存的QuerySet使用iterator()方法。

6. 将查询计算操作放在数据库中完成，不要在Python代码中完成。
   1. 使用filter,exclude完成查询过滤；
   2. F()查询表达式；
   3. 使用聚合函数来完成数据库聚合操作。
7. 使用QuerySet.extra()明确的指出要查询的字段。
8. 对于复杂的数据库查询操作，使用原生SQL实现。
9. 尽量一次查询出所有需要的信息。
10. 只查询需要的数据：
    1. 某些情况下，只使用 QuerySet.values()和 values_list()方法，查询出符合条件的结果集而不是完整的对象结果集；
    2. 某些情况下，只使用 QuerySet.defer() 和 only()过滤数据。
11. 如果只是查询集合的数量，使用QuerySet.count()函数，而不是len(QuerySet)；
12. 如果想知道某个记录是否包含在某个结果集中，使用 QuerySet.exists()函数；
13. 避免过多的使用 count() 和 exists() 函数；
14. 对于批量更新和删除操作使用 QuerySet.update() 和 QuerySet.delete()；

9. 验证码过期时间怎么设置?

将验证码保存到数据库或 session，设置过期时间为 1 分钟，然后页面设置一个倒计时(一般是前端 js 实现 这个计时)的展示，一分钟过后再次点击获取新的信息。

10. Python 中三大框架各自的应用场景?

1. Django

> Django是一个开放源代码的Web应用框架，由Python写成。采用了MVT的框架模式，即模型M，模板T和视图V。主要是用来搞快速开发的，他的亮点就是快速开发，节约成本

2. Flask

> 轻量级，主要是用来写接口的一个框架，实现前后端分离，提升开发效率，Flask 本身相当于一 个内核，其他几乎所有的功能都要用到扩展（邮件扩展 Flask-Mail，用户认证 Flask-Login），都需要 用第三方的扩展来实现。比如可以用 Flask-extension 加入 ORM、窗体验证工具，文件上传、身份验 证等。Flask 没有默认使用的数据库，你可以选择 MySQL，也可以用 NoSQL。

3. Tornado

> Tornado是一种 Web 服务器软件的开源版本。Tornado 和现在的主流 Web 服务器框架（包括大多数 Python 的框架）有着明显的区别：它是非阻塞式服务器，而且速度相当快。得利于其 非阻塞的方式和对epoll的运用，Tornado 每秒可以处理数以千计的连接，因此 Tornado 是实时 Web 服务的一个 理想框架。

11、django 如何提升性能（高并发）？

> 对一个后端开发程序员来说，提升性能指标主要有两个一个是并发数，另一个是响应时间网站性能 的优化一般包括 web 前端性能优化，应用服务器性能优化，存储服务器优化。

对前端的优化主要有：

1. 减少 http 请求，减少数据库的访问量，比如使用雪碧图。
2. 使用浏览器缓存，将一些常用的 css，js，logo 图标，这些静态资源缓存到本地浏览器，通过设 置 http 头中的 cache-control 和 expires 的属性，可设定浏览器缓存，缓存时间可以自定义。
3. 对 html，css，javascript 文件进行压缩，减少网络的通信量。

对我个人而言，我做的优化主要是以下三个方面：

1. 合理的使用缓存技术，对一些常用到的动态数据，比如首页做一个缓存，或者某些常用的数据做 个缓存，设置一定得过期时间，这样减少了对数据库的压力，提升网站性能。
2. 使用 celery 消息队列，将耗时的操作扔到队列里，让 worker 去监听队列里的任务，实现异步操 作，比如发邮件，发短信。
3. 就是代码上的一些优化，补充：nginx 部署项目也是项目优化，可以配置合适的配置参数，提升 效率，增加并发量。
4. 如果太多考虑安全因素，服务器磁盘用固态硬盘读写，远远大于机械硬盘，这个技术现在没有普 及，主要是固态硬盘技术上还不是完全成熟， 相信以后会大量普及。
5. 另外还可以搭建服务器集群，将并发访问请求，分散到多台服务器上处理。
6. 最后就是运维工作人员的一些性能优化技术了

12、什么是 restful api，谈谈你的理解?

REST:Representational State Transfer 的缩写，翻译：“具象状态传输”。一般解释为“表现层 状态转换”。

REST 是设计风格而不是标准。是指客户端和服务器的交互形式。我们需要关注的重点是如何设计 REST 风格的网络接口。

REST 的特点：

1. 具象的。一般指表现层，要表现的对象就是资源。比如，客户端访问服务器，获取的数据就是资 源。比如文字、图片、音视频等。
2. 表现：资源的表现形式。txt 格式、html 格式、json 格式、jpg 格式等。浏览器通过 URL 确定资 源的位置，但是需要在 HTTP 请求头中，用 Accept 和 Content-Type 字段指定，这两个字段是对资源 表现的描述。
3. 状态转换：客户端和服务器交互的过程。在这个过程中，一定会有数据和状态的转化，这种转化 叫做状态转换。其中，GET 表示获取资源，POST 表示新建资源，PUT 表示更新资源，DELETE 表示删 除资源。HTTP 协议中最常用的就是这四种操作方式。

RESTful 架构：

1. 每个 URL 代表一种资源；
2. 客户端和服务器之间，传递这种资源的某种表现层；
3. 客户端通过四个 http 动词，对服务器资源进行操作，实现表现层状态转换。

13. 如何设计符合 RESTful 风格的 API

1. 域名：

   将 api 部署在专用域名下：`http://api.example.com`

   或者将 api 放在主域名下：`http://www.example.com/api/`

2. 版本：

   将 API 的版本号放在 url 中。

   `http://www.example.com/app/1.0/info`

   `http://www.example.com/app/1.2/info`

3. 路径：

   路径表示 API 的具体网址。每个网址代表一种资源。 资源作为网址，网址中不能有动词只能有 名词，一般名词要与数据库的表名对应。而且名词要使用复数。

   错误示例：

   `http://www.example.com/getGoods`

`http://www.example.com/listOrders`

正确示例：

获取单个商品 `http://www.example.com/app/goods/1`

获取所有商品 `http://www.example.com/app/goods`

4. 使用标准的 HTTP 方法：

   对于资源的具体操作类型，由 HTTP 动词表示。 常用的 HTTP 动词有四个。

   - GET SELECT： 从服务器获取资源。
   - POST CREATE： 在服务器新建资源。
   - PUT UPDATE： 在服务器更新资源。
   - DELETE DELETE： 从服务器删除资源。

   示例：

   - 获取指定商品的信息 `GET http://www.example.com/goods/ID`
   - 新建商品的信息 `POST http://www.example.com/goods`
   - 更新指定商品的信息 `PUT http://www.example.com/goods/ID`
   - 删除指定商品的信息 `DELETE http://www.example.com/goods/ID`

5. 过滤信息：

   如果资源数据较多，服务器不能将所有数据一次全部返回给客户端。API 应该提供参数，过滤返 回结果。 实例：

   - 指定返回数据的数量 `http://www.example.com/goods?limit=10`

- 指定返回数据的开始位置 `http://www.example.com/goods?offset=10`
- 指定第几页，以及每页数据的数量 `http://www.example.com/goods?page=2&per_page=20`

6. 状态码：

   服务器向用户返回的状态码和提示信息，常用的有：

   - 200 OK： 服务器成功返回用户请求的数据
   - 201 CREATED： 用户新建或修改数据成功。
   - 202 Accepted：表示请求已进入后台排队。
   - 400 INVALID REQUEST： 用户发出的请求有错误。
   - 401 Unauthorized： 用户没有权限。
   - 403 Forbidden： 访问被禁止。
   - 404 NOT FOUND： 请求针对的是不存在的记录。
   - 406 Not Acceptable： 用户请求的的格式不正确。
   - 500 INTERNAL SERVER ERROR： 服务器发生错误。

7. 错误信息：

   一般来说，服务器返回的错误信息，以键值对的形式返回 `{ error: 'Invalid API KEY' }`

8. 响应结果： data

   针对不同结果，服务器向客户端返回的结果应符合以下规范

   - 返回商品列表
     `GET http://www.example.com/goods`
   - 返回单个商品
     `GET http://www.example.com/goods/cup`
   - 返回新生成的商品
     `POST http://www.example.com/goods`
   - 返回一个空文档
     `DELETE http://www.example.com/goods`

9. 使用链接关联相关的资源：

   在返回响应结果时提供链接其他 API 的方法，使客户端很方便的获取相关联的信息。

10. 其他：

    服务器返回的数据格式，应该尽量使用 JSON，避免使用 XML。

14. 启动 Django 服务的方法?

runserver 方法是调试 Django 时经常用到的运行方式，它使用 Django 自带的 WSGI Server 运 行，主要在测试和开发中使用，并且 runserver 开启的方式也是单进程 。

15. 怎样测试 django 框架中的代码?

Django的单元测试使用python的unittest模块，这个模块使用基于类的方法来定义测试。类名为 `django.test.TestCase`,继承于python的 `unittest.TestCase` 。

执行目录下所有的测试(所有的test*.py文件)：运行测试的时候，测试程序会在所有以test开头的文件中查找所有的test cases(inittest.TestCase的子类),自动建立测试集然后运行测试。

```
# 执行所有的测试用例
python manage.py test
# 执行animals项目下tests包里的测试：
```

```
python manage.py testanimals.tests
# 执行animals项目里的test测试：
python manage.py testanimals
# 单独执行某个test case：
python manage.py testanimals.tests.AnimalTestCase
# 单独执行某个测试方法：
python manage.py testanimals.tests.AnimalTestCase.test_animals_can_speak
为测试文件提供路径：
python manage.py testanimals/
# 通配测试文件名：
python manage.py test-pattern=”tests_*.py”
# 启用warnings提醒：
python -Wall manage.py test
```

16. 有过部署经验？用的什么技术？可以满足多少压力？

1. 有部署经验，在阿里云服务器上部署的
2. 技术有：nginx + uwsgi 的方式来部署 Django 项目
3. 无标准答案（例：压力测试一两千）

17. Django 中哪里用到了线程?哪里用到了协程?哪里用到了进程?

Django 中耗时的任务用一个进程或者线程来执行，比如发邮件，使用 celery。

部署 django 项目的时候，配置文件中设置了进程和协程的相关配置。

18. django 关闭浏览器，怎样清除 cookies 和 session?

设置 Cookie

```
def cookie_set(request):
     response = HttpResponse("<h1>设置 Cookie，请查看响应报文头</h1>")
     response.set_cookie('h1', 'hello django')
     return response
```

读取 Cookie

```
def cookie_get(request):
    response = HttpResponse("读取 Cookie，数据如下：<br>")
    if request.COOKIES.has_key('h1'):
    response.write('<h1>' + request.COOKIES['h1'] + '</h1>')
    return response
```

以键值对的格式写会话。 `request.session['键']=值`

根据键读取值。 `request.session.get('键',默认值)`

清除所有会话，在存储中删除值部分。 `request.session.clear()`

清除会话数据，在存储中删除会话的整条数据。 `request.session.flush()`

删除会话中的指定键及值，在存储中只删除某个键及对应的值。 `del request.session['键']`

设置会话的超时时间，如果没有指定过期时间则两个星期后过期。 如果 value 是一个整数，会话将在 value 秒没有活动后过期。 如果 value 为 0，那么用户会话的 Cookie 将在用户的浏览器关闭时过期。 如果 value 为 None，那么会话在两周后过期。 `request.session.set_expiry(value)`

Session 依赖于 Cookie，如果浏览器不能保存 cookie 那么 session 就失效了。因为它需要浏览 器的 cookie 值去 session 里做对比。session 就是用来在服务器端保存用户的会话状态。

cookie 可以有过期时间，这样浏览器就知道什么时候可以删除 cookie 了。 如果 cookie 没有设 置过期时间，当用户关闭浏览器的时候，cookie 就自动过期了。可以改变 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 的设置来控制 session 框架的这一行为。

SESSION_COOKIE_AGE：设置 cookie 在浏览器中存活的时间。

缺省情况下， `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `False` ，这样，会话 cookie 可以在用户浏览器中保 持有效达 `SESSION_COOKIE_AGE` 秒（缺省设置是两周，即 1，209，600 秒）如果你不想用户每次 打开浏览器都必须重新登陆的话，用这个参数来帮你。如果 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `True` ，当浏览器关闭时，Django 会使 cookie 失效。

19. 代码优化从哪些方面考虑？有什么想法？

1. 优化算法时间复杂度

   算法的时间复杂度对程序的执行效率影响最大，在 Python 中可以通过选择合适的数据结构来优化 时间复杂度，如 list 和 set 查找某一个元素的时间复杂度分别是 O(n)和 O(1)。不同的场景有不同的 优化方式，总得来说，一般有分治，分支界限，贪心，动态规划等思想 。

2. 减少冗余数据

   如用上三角或下三角的方式去保存一个大的对称矩阵。在0元素占大多数的矩阵里使用稀疏矩阵表示。

3. 合理使用copy与deepcopy

   对于dict和list等数据结构的对象，直接赋值使用的是引用的方式。而有些情况下需要复制整个对象，这时可以使用copy包里的copy和deepcopy，这两个函数的不同之处在于后者是递归复制的。效率也不一样：（以下程序在ipython中运行）

4. 使用dict或set查找元素

   python dict和set都是使用hash表来实现(类似c++11标准库中unordered_map)，查找元素的时间复杂度是O(1)

5. 合理使用生成器（generator）和yield

   使用 `()` 得到的是一个generator对象，所需要的内存空间与列表的大小无关，所以效率会高一些。在具体应用上，比如set(i for i in range(100000))会比set([i for i in range(100000)])快。

   对于内存不是非常大的list，可以直接返回一个list，但是可读性 `yield` 更佳(人个喜好)。

6. 优化循环

   循环之外能做的事不要放在循环内

7. 优化包含多个判断表达式的顺序

   对于and，应该把满足条件少的放在前面，对于or，把满足条件多的放在前面

8. 使用join合并迭代器中的字符串

   `join` 对于累加的方式，有大约5倍的提升。

9. 不借助中间变量交换两个变量的值

使用`a,b=b,a`而不是`c=a;a=b;b=c;`来交换a,b的值，可以快1倍以上。

10. 使用`if is`

    使用`if is True`比`if == True`将近快一倍。

11. 并行编程

    因为GIL的存在，Python很难充分利用多核CPU的优势。但是，可以通过内置的模块multiprocessing实现下面几种并行模式：

    - 多进程：对于CPU密集型的程序，可以使用multiprocessing的Process,Pool等封装好的类，通过多进程的方式实现并行计算。但是因为进程中的通信成本比较大，对于进程之间需要大量数据交互的程序效率未必有大的提高。
    - 多线程：对于IO密集型的程序，multiprocessing.dummy模块使用multiprocessing的接口封装threading，使得多线程编程也变得非常轻松(比如可以使用Pool的map接口，简洁高效)。
    - 分布式：multiprocessing中的Managers类提供了可以在不同进程之共享数据的方式，可以在此基础上开发出分布式的程序。

不同的业务场景可以选择其中的一种或几种的组合实现程序性能的优化。

12. 终级大杀器：PyPy

    PyPy是用RPython(CPython的子集)实现的Python，根据官网的基准测试数据，它比CPython实现的Python要快6倍以上。快的原因是使用了Just-in-Time(JIT)编译器，即动态编译器，与静态编译器(如gcc,javac等)不同，它是利用程序运行的过程的数据进行优化。由于历史原因，目前pypy中还保留着GIL，不过正在进行的STM项目试图将PyPy变成没有GIL的Python。

    如果python程序中含有C扩展(非cffi的方式)，JIT的优化效果会大打折扣，甚至比CPython慢（比Numpy）。所以在PyPy中最好用纯Python或使用cffi扩展。

20. Django 中间件是如何使用的?

在http请求 到达视图函数之前 和视图函数return之后，django会根据自己的规则在合适的时机执行中间件中相应的方法。

中间件的执行流程

1. 执行完所有的request方法 到达视图函数。
2. 执行中间件的其他方法
3. 经过所有response方法 返回客户端。

注意：如果在其中1个中间件里 request方法里 return了值，就会执行当前中间的response方法，返回给用户 然后报错。。不会再执行下一个中间件。

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

- process_request(self,request)
- process_view(self, request, view_func, view_args, view_kwargs)
- process_template_response(self,request,response)
- process_exception(self, request, exception)
- process_response(self, request, response)

中间件不用继承自任何类（可以继承 object），下面一个中间件大概的样子：

```python
from django.utils.deprecation import MiddlewareMixin
class MD1(MiddlewareMixin):
    def process_request(self, request):
        print("MD1里面的 process_request")
    def process_response(self, request, response):
        print("MD1里面的 process_response")
        return response
```

21. 有用过 Django REST framework 吗?

Django REST framework 是一个强大而灵活的 Web API 工具。使用 RESTframework 的理由 有：

Web browsable API 对开发者有极大的好处

包括 OAuth1a 和 OAuth2 的认证策略

支持 ORM 和非 ORM 数据资源的序列

全程自定义开发——如果不想使用更加强大的功能，可仅仅使用常规的 function-based views 额外的文档和强大的社区支持

23. Jieba 分词

jieba分词算法使用了基于前缀词典实现高效的词图扫描，生成句子中汉字所有可能生成词情况所构成的有向无环图(DAG), 再采用了动态规划查找最大概率路径，找出基于词频的最大切分组合，对于未登录词，采用了基于汉字成词能力的HMM模型，使用了Viterbi算法。

jieba分词支持三种分词模式：

1. 精确模式, 试图将句子最精确地切开，适合文本分析：
2. 全模式，把句子中所有的可以成词的词语都扫描出来，速度非常快，但是不能解决歧义；
3. 搜索引擎模式，在精确模式的基础上，对长词再词切分，提高召回率，适合用于搜索引擎分词。

24. ngnix 的正向代理与反向代理?

web 开发中，部署方式大致类似。简单来说，使用 Nginx 主要是为了实现分流、转发、负载均衡，以及分担服务器的压力。Nginx 部署简单，内存消耗少，成本低。Nginx 既可以做正向代理，也可以做反向代理。

正向代理：正向代理类似一个跳板机，代理访问外部资源。 作用：

- 访问原来无法访问的资源，如google
- 可以做缓存，加速访问资源
- 对客户端访问授权，上网进行认证
- 代理可以记录用户访问记录（上网行为管理），对外隐藏用户信息

反向代理：实际运行方式是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个服务器。作用：

- 保证内网的安全，可以使用反向代理提供WAF功能，阻止web攻击
- 负载均衡，通过反向代理服务器来优化网站的负载

区别：两者的区别的对象不一样，[正向代理]代理对象是客户端,[反向代理]代理的对象是服务端。

25. 简述 Django 下的（内建的）缓存机制?

django根据设置缓存方式，浏览器骑一次请求时，cache会缓存单个变量或者整个网页到磁盘或者内存，同时设置了response头部，第二次访问的时候，根据缓存时间的判断，将内容将缓存数据返回给客户端。

26. 请简述浏览器是如何获取一枚网页的?

1.在用户输入目的 URL 后，浏览器先向 DNS 服务器发起域名解析请求； 2.在获取了对应的 IP 后向服务器发送请求数据包； 3.服务器接收到请求数据后查询服务器上对应的页面，并将找到的页面代码回复给客户端； 4.客户端接收到页面源代码后，检查页面代码中引用的其他资源，并再次向服务器请求该资源； 5.在资源接收完成后，客户端浏览器按照页面代码将页面渲染输出显示在显示器上；

27、对 cookie 与 session 的了解？他们能单独用吗?

Session 采用的是在服务器端保持状态的方案，而 Cookie 采用的是在客户端保持状态的方案。但是禁用 Cookie 就不能得到 Session。因为 Session 是用 Session ID 来确定当前对话所对应的服务器 Session，而 Session ID 是通过 Cookie 来传递的，禁用 Cookie 相当于失去了 SessionID，也就得不到 Session。

28. Django HTTP 请求的处理流程?

1. 加载settings.py

   在通过django-admin.py创建project的时候，Django会自动生成默认的settings文件和manager.py等文件

2. 创建WSGIServer

   不管是使用runserver还是uWSGI运行Django项目，在启动时都会调用django.core.servers.basehttp中的run()方法，创建一个django.core.servers.basehttp.WSGIServer类的实例，之后调用其serve_forever()方法启动HTTP服务。

3. 处理Request

4. 返回Response

```
1. 用户通过浏览器请求一个页面
2. 请求到达Request Middlewares，中间件对request做一些预处理或者直接response请求
3. URLConf通过urls.py文件和请求的URL找到相应的View
4. View Middlewares被访问，它同样可以对request做一些处理或者直接返回response
5. 调用View中的函数
6. View中的方法可以选择性的通过Models访问底层的数据
7.所有的Model-to-DB的交互都是通过manager完成的
8. 如果需要，Views可以使用一个特殊的Context
9. Context被传给Template用来生成页面
    a. Template使用Filters和Tags去渲染输出
    b. 输出被返回到View
    c. HTTPResponse被发送到Response Middlewares
    d. 任何Response Middlewares都可以丰富response或者返回一个完全不同的response
    e. Response返回到浏览器，呈现给用户
```

29. Django 里 QuerySet 的 get 和 filter 方法的区别?

- 输入参数

get 的参数只能是model中定义的那些字段，只支持严格匹配 filter 的参数可以是字段，也可以是扩展的where查询关键字，如in，like等

- 返回值

  get 返回值是一个定义的model对象

  filter 返回值是一个新的QuerySet对象，然后可以对QuerySet在进行查询返回新的QuerySet对象，支持链式操作 QuerySet一个集合对象，可使用迭代或者遍历，切片等，但是不等于list类型(使用一定要注意)

- 异常

  get 只有一条记录返回的时候才正常,也就说明get的查询字段必须是主键或者唯一约束的字段。当返回多条记录或者是没有找到记录的时候都会抛出异常

filter 有没有匹配的记录都可以

30. django 中当一个用户登录 A 应用服务器（进入登录状态），然后下次请求被 nginx 代理到 B 应用服务器会出现什么影响?

如果用户在 A 应用服务器登陆的 session 数据没有共享到 B 应用服务器，那么之前的登录状态就没 有了。

31. Django 对数据查询结果排序怎么做，降序怎么做，查询大于某个字段怎么做?

- 排序使用 order_by()
- 降序需要在排序字段名前加-
- 查询字段大于某个值：使用 filter (字段名_gt=值)

33. Django 重定向你是如何实现的？用的什么状态码?

使用 HttpResponseRedirect

redirect 和 reverse

状态码：302，301

34. 生成迁移文件和执行迁移文件的命令是什么?

`python manage.py makemigrations`

`python manage.py migrate`

35. 关系型数据库的关系包括哪些类型?

- `ForeignKey`：一对多，将字段定义在多的一端中。
- `ManyToManyField`：多对多：将字段定义在两端中。
- `OneToOneField`：一对一，将字段定义在任意一端中。

36. 查询集返回列表的过滤器有哪些?

- `all()`：返回所有数据。
- `filter()`：返回满足条件的数据。
- `exclude()`：返回满足条件之外的数据。
- `order_by()`：对结果进行排序。

37. 判断查询集正是否有数据?

exists()：判断查询集中否有数据，如果有则返回 True，没有则返回 False。

38. Django 本身提供了 runserver，为什么不能用来部署?

runserver 方法是调试 Django 时经常用到的运行方式，它使用 Django 自带的WSGI Server 运行，主要在测试和开发中使用，并且 runserver 开启的方式也是单进程 。

uWSGI 是一个 Web 服务器，它实现了 WSGI 协议、uwsgi、http 等协议。注意 uwsgi 是一种通信协议，而 uWSGI 是实现 uwsgi 协议和 WSGI 协议的 Web 服务器。uWSGI 具有超快的性能、低内存占用和多 app 管理等优点，并且搭配着 Nginx就是一个生产环境了，能够将用户访问请求与应用 app 隔离开，实现真正的部署 。相比来讲，支持的并发量更高，方便管理多进程，发挥多核的优势，提升性能

39、apache 和 nginx 的区别?

Nginx

1. 轻量级，采用 C 进行编写，同样的 web 服务，会占用更少的内存及资源
2. 抗并发，nginx 以 epoll and kqueue 作为开发模型，处理请求是异步非阻塞的，负载能力比 apache 高很多，而 apache 则是阻塞型的。在高并发下 nginx 能保持低资源低消耗高性能 ，而 apache 在 PHP 处理慢或者前端压力很大的情况下，很容易出现进程数飙升，从而拒绝服务的现象。
3. nginx 处理静态文件好，静态处理性能比 apache 高三倍以上
4. nginx 的设计高度模块化，编写模块相对简单
5. nginx 配置简洁，正则配置让很多事情变得简单，而且改完配置能使用 -t 测试配置有没有问题，apache 配置复杂 ，重启的时候发现配置出错了，会很崩溃
6. nginx 作为负载均衡服务器，支持 7 层负载均衡
7. nginx 本身就是一个反向代理服务器，而且可以作为非常优秀的邮件代理服务器
8. 启动特别容易, 并且几乎可以做到 7*24 不间断运行，即使运行数个月也不需要重新启动，还能够不间断服务的情况下进行软件版本的升级
9. 社区活跃，各种高性能模块出品迅速

Apache

1. apache 的 rewrite 比 nginx 强大，在 rewrite 频繁的情况下，用 apache
2. apache 发展到现在，模块超多，基本想到的都可以找到
3. apache 更为成熟，少 bug ，nginx 的 bug 相对较多
4. apache 超稳定
5. apache 对 PHP 支持比较简单，nginx 需要配合其他后端用
6. apache 在处理动态请求有优势，nginx 在这方面是鸡肋，一般动态请求要 apache 去做，nginx 适合静态和反向。
7. apache 仍然是目前的主流，拥有丰富的特性，成熟的技术和开发社区

40. varchar 与 char 的区别?

char 长度是固定的，不管你存储的数据是多少他都会都固定的长度。而 varchar 则处可变长度但他 要在总长度上加 1 字符，这个用来存储位置。所以在处理速度上 char 要比 varchar 快速很多，但是对 费存储空间，所以对存储不大，但在速度上有要求的可以使用 char 类型，反之可以用 varchar 类型。

41. 查询集两大特性？惰性执行?

1. 惰性执行

   创建查询集不会访问数据库，直到调用数据时，才会访问数据库，调用数据的情况包括迭代、序列化、与if合用

2. 缓存

   使用同一个查询集，第一次使用时会发生数据库的查询，然后Django会把结果缓存下来，再次使用这个查询集时会使用缓存的数据，减少了数据库的查询次数。

42. git 常用命令?

- git clone 克隆指定仓库
- git status 查看当前仓库状态
- git diff 比较版本的区别
- git log 查看 git 操作日志
- git reset 回溯历史版本
- git add 将文件添加到暂存区
- git commit 将文件提交到服务器
- git checkout 切换到指定分支
- git rm 删除指定文件
- git merge 合并分支

43. 电商网站库存问题

一般团购，秒杀，特价之类的活动，这样会使访问量激增，很多人抢购一个商品，作为活动商品， 库存肯定是很有限的。控制库存问题，数据库的事务功能是控制库存超卖的有效方式。

1. 在秒杀的情况下，肯定不能如此频率的去读写数据库，严重影响性能问题，必须使用缓存，将需 要秒杀的商品放入缓存中，并使用锁来处理并发情况，先将商品数量增减（加锁、解析）后在进行其他 方面的处理，处理失败再将数据递增（加锁、解析）,否则表示交易成功。
2. 这个肯定不能直接操作数据库的，会挂的。直接读库写库对数据库压力太大了，要用到缓存。
3. 首先，多用户并发修改同一条记录时，肯定是后提交的用户将覆盖掉前者提交的结果了。这个直 接可以使用加乐观锁的机制去解决高并发的问题。

44. HttpRequest 和 HttpResponse 是什么?干嘛用的?

HttpRequest 是 django 接受用户发送多来的请求报文后，将报文封装到 HttpRequest 对象中去。

HttpResponse 返回的是一个应答的数据报文。render 内部已经封装好了 HttpResponse 类

45. 什么是反向解析?

使用场景：模板中的超链接，视图中的重定向 使用：在定义 url 时为 include 定义 namespace 属性，为 url 定义 name 属性 在模板中使用 url 标签：{% url 'namespace_value:name_value'%} 在视图中使用 reverse 函数：redirect(reverse('namespce_value:name_value')) 根据正则表达式动态生成地址，减轻后期维护成本。 注意反向解析传参数，主要是在我们的反向解析的规则后面天界了两个参数，两个参数之间使用空格隔 开：位置参数

爬虫

1. 爬取数据后使用哪个数据库存储数据的，为什么?

一般是mysql和mongodb

爬取的数据通常都是非结构化数据，这在关系模型的存储和查询上面都有很大的局限性。但也有一个可能性是你感兴趣的网站都是同样类型的网站，你只对网页上的特定内容有兴趣，这样可以把它们组织成结构化数据，从而在这方面MySQL仍然是可以胜任的

200w到2000w的数据量相对来说不是很大，二者都可以。但是基本上数据库达到千万级别都会有查询性能的问题，所以如果数据持续增长的话，可以考虑用mongodb。毕竟mongodb分片集群搭建起来比mysql集群简单多了。而且处理起来更灵活，

之前项目当时日志处理和归档，对每天所生成的访问日志进行冷热统计，生成各种数据报表等等，开始做项目考虑过MYSQL，不过MYSQL单表在超过千万级以上性能表现不佳，所以当时还是选择使用mongodb 其实做的也很简单，无非是用python定时将每日的服务器日志抓取到本地，然后利用pandas库，将数据构造成自己想要的数据结构，需要计算分组聚合的就聚合，最终把每日的数据结果扔到mongodb中。 现在公司mongodb数据大概放了有8KW条左右，进行数据检索效率还是可以的，当然记得加索引。

我这边除了把数据记录到mongodb之外，还用flask写了个restfulAPI，专门给运营系统调用数据统计结果，运营那边也会在MYSQL上创建一张表，将我mongodb统计出的结果再次统计出一个总数据，放到MYSQL里，这样就不用每次从API那边拿数据调用mongodb中进行重复聚合计算了。

这里有一个更专业的回答：点击查看

2. 你用过的爬虫框架或者模块有哪些？谈谈他们的区别或者优缺点?

- requests
  request是一个HTTP 库， 它只是用来，进行请求，对于HTTP 请求，他是一个强大的库，下载，解析全部自己处理，灵活性更高，高并发与分布式部署也非常灵活，对于功能可以更好实现
- Scrapy
  scrapy 是封装起来的框架，他包含了下载器，解析器，日志及异常处理，基于多线程，twisted的方式处理，对于固定单个网站的爬取开发，有优势，但是对于多网站爬取 100个网站，并发及分布式
  处理方面，不够灵活，不便调整与括展。

3.详述 Scrapy的优缺点

- scrapy 是异步的;
- 采取可读性更强的xpath代替正则;
- 强大的统计和log系统;
- 同时在不同的url上爬取;
- 支持shell方式，方便独立调试;
- 写middleware,方便写一些统一的过滤器;
- 通过管道的方式存入数据库;
- 基于twisted框架，运行中的exception是不会干掉reactor，并且异步框架出错后是不会停掉
  其他任务的，数据出错后难以察觉

4. 写爬虫是用多进程好？还是多线程好？ 为什么?

IO密集型代码(文件处理、网络爬虫等)，多线程能够有效提升效率(单线程下有IO操作会进行IO 等待，造成不必要的时间浪费，而开启多线程能在线程A等待时，自动切换到线程B，可以不浪费CPU的资源，从而能提升程序执行效率)。

在实际的数据采集过程中，既考虑网速和响应的问题，也需要考虑 自身机器的硬件情况，来设置多进程或多线程。

5. 常见的反爬虫和应对方法?

1. 通过Headers反爬虫：

   很多网站都会对请求Headers的User-Agent进行检测，还有一部分网站会对Referer进行检测（一些资源网站的防盗链就是检测Referer）。这个容易解决，之前收集了一大堆的User-Agent，如果用的scrapy框架，就在爬虫中间件中对reqeust请求添加一个随机的User-Agent，如果是requests库，在请求方法中传入一个包含User-Agent的headers字典即可

2. 验证码，

   爬虫爬久了通常网站的处理策略就是让你输入验证码验证是否机器人，此时有三种解决方法

   - 第一种把验证码down到本地之后，手动输入验证码验证，此种成本相对较高，而且不能完全做到自动抓取，需要人为干预。
   - 第二种图像识别验证码，自动填写验证，但是现在的情况是大部分验证码噪声较多复杂度大，对于像我这样对图像识别不是很熟悉的人很难识别出正确的验证码。
   - 第三种也是最实用的一种，接入自动打码平台，个人感觉比上两种方法好些。

3. 用户行为检测

   有很多的网站会通过同一个用户单位时间内操作频次来判断是否机器人，比如像新浪微博等网站。这种情况下我们就需要先测试单用户抓取阈值，然后在阈值前切换账号其他用户，如此循环即可。

   当然，新浪微博反爬手段不止是账号，还包括单ip操作频次等。所以可以使用代理池每次请求更换不同的代理ip，也可以考虑维护了一个cooies池，在每次请求时随机选择一个cookies

4. ajax请求参数被js加密

   使用selenium + phantomJS，调用浏览器内核，并利用phantomJS执行js来模拟人为操作以及触发页面中的js脚本。从填写表单到点击按钮再到滚动页面，全部都可以模拟，不考虑具体的请求和响应过程，只是完完整整的把人浏览页面获取数据的过程模拟一遍。

4. 分布式爬虫，

   分布式能在一定程度上起到反爬虫的作用，当然相对于反爬虫分布式最大的作用还是能做到高效大量的抓取

5. 注意配合移动端、web端以及桌面版

   其中web端包括m站即手机站和pc站，往往是pc站的模拟抓取难度大于手机站，所以在m站和pc站的资源相同的情况下优先考虑抓取m站。同时如果无法在web端抓取，不可忽略在app以及桌面版的也可以抓取到目标数据资源。

应对反爬虫的策略，首先要发现网站的反爬虫手段是什么？这个发现的过程就是不断测试的过程，有点类似于A/B测试，弄清楚它的反爬虫机制，就成功了一大半了。

6. 验证码的解决?

图形验证码：干扰、杂色不是特别多的图片可以使用开源库Tesseract进行识别，太过复杂的需要借助第三方打码平台。

点击和拖动滑块验证码可以借助selenium、无图形界面浏览器（chromedirver或者phantomjs）和pillow包来模拟人的点击和滑动操作，pillow可以根据色差识别需要滑动的位置。

selenium, 计算缺口的偏移量，找到特征点，控制鼠标实现,破解“极验”滑动验证码 详细请点击

7. 代理 IP 里的“透明”“匿名”“高匿”分别是指?

- 透明代理虽然可以直接“隐藏”你的IP地址，但是还是可以从HTTP_X_FORWARDED_FOR来查到你是谁。

- 匿名代理比透明代理进步了一点：别人只能知道你用了代理，无法知道你的IP地址。
- 与匿名代理相同，如果使用了混淆代理，别人还是能知道你在用代理，但是会得到一个假的IP地址，伪装的更逼真：
- 高匿代理让别人根本无法发现你是在用代理

8. 编写过哪些爬虫中间件?

user-agent、代理池、cookies池、selenium

9. 爬的那些内容数据量有多大，多久爬一次，爬下来的数据是怎么存储?

京东整站的数据大约在1亿左右，爬下来的数据存入数据库，mysql数据库中如果有重复的url建议去重存入数据库，可以考虑引用外键。评分，评论如果做增量，Redis中url去重，评分和评论建议建立一张新表用id做关联。

多久爬一次这个问题要根据公司的要求去处理，不一定是每天都爬。 Mongo 建立唯一索引键（id）可以做数据去重 前提是数据量不大 2台电脑几百万的数据库需要做分片 （数据库要设计合理）。 例：租房的网站数据量每天大概是几十万条 ， 每周固定爬取。

10. requests返回的content和text的区别?

- response.text

  类型：str

  解码类型： 根据HTTP 头部对响应的编码作出有根据的推测，推测的文本编码

  如何修改编码方式：response.encoding=”gbk

- response.content

  类型：bytes

  解码类型： 没有指定

  如何修改编码方式：response.content.deocde(“utf8”)

11. 描述下scrapy框架运行的机制?

1. 通过spider的start_requests()方法，遍历start_urls构造Request请求，由引擎交给调度器入请求队列，
2. 调度器将请求队列里的请求交给下载器去获取请求对应的响应资源，并将响应交给指定的解析方法做提取处理：
    1. 如果提取出需要的数据，则交给管道文件处理；
    2. 如果提取出Request对象，则将请求交给调度器，放入请求队列中
3. 重复2 过程，直到请求队列里没有请求，程序结束

12. 怎么样让 scrapy 框架发送一个 post 请求?

```
yield scrapy.FormRequest(
    url = url,
    formdata = {"email" : "xxx",  "password" : "xxxxx"},  callback = self.parse_page )
```

13. 怎么判断网站是否更新?

使用 MD5 数字签名： 第一次请求指定url，把服务器返回的数据body进行MD5 数字签名 S1，下次请求相同ur，同理生成签名 S2，比较 S2 和 S1， 如果相同，则页面没有更新，否则网页就有更新

还有其他办法，请点击查看

14. 爬虫向数据库存数据开始和结束都会发一条消息，是scrapy哪个模块实现?

Scrapy使用信号来通知事情发生，是signals 模块。

15. 爬取下来的数据如何去重

1. 通过MD5生成电子指纹来判断页面是否改变
2. 数据量不大时，可以直接放在内存里面进行去重，python可以使用set进行去重。当去重数据需要持久化时可以使用redis的set数据结构
3. 当数据量再大一点时，可以用不同的加密算法先将长字符串压缩成 16/32/40 个字符，在利用set集合去重，scrapy中对request对象的去重就是利用hashlib.sha1生成request指纹放入set中进行去重的
4. 当数据量达到亿（甚至十亿、百亿）数量级时，内存有限，必须用“位”来去重，才能够满足需求。Bloomfilter（布隆过滤器）就是将去重对象映射到几个内存“位”，通过几个位的 0/1值来判断一个对象是否已经存在
，之前在对新浪微博数据进行爬取的时候，对url的去重就是利用redis的“位”来去重，1G的内存大概实现80亿url的去重

16. scrapy和scrapy-redis有什么区别？为什么选择redis数据库?

- scrapy 是一个Python爬虫框架，爬取效率极高，具有高度定制性，但是不支持分布式。
- 而scrapy-redis一套基于redis数据库、运行在scrapy 框架之上的组件，可以让scrapy 支持分布式策略，Slaver端共享Master 端redis数据库里的item队列、请求队列和请求指纹集合。
- 为什么选择redis数据库，因为redis支持主从同步，而且数据都是缓存在内存中的，所以基于redis的分布式爬虫，对请求和数据的高频读取效率非常高。

17. 你所知道的分布式爬虫方案有哪些?

三种分布式爬虫策略：

1. Slaver 端从 Master 端拿任务（Request/url/ID）进行数据抓取，在抓取数据的同时也生成新任务，并将任务抛给 Master。Master 端只有一个 Redis 数据库，负责对 Slaver 提交的任务进行去重、加入待爬队列。
    - 优点： scrapy-redis 默认使用的就是这种策略，我们实现起来很简单，因为任务调度等工作 scrapy-redis 都已经帮我们做好了，我们只需要继承 RedisSpider、指定 redis_key 就行了。
    - 缺点： scrapy-redis 调度的任务是 Request 对象，里面信息量比较大（不仅包含 url，还有 callback 函数、headers 等信息），导致的结果就是会降低爬虫速度、而且会占用Redis 大量的存储空间。当然我们可以重写方法实现调度 url 或者用户 ID。
2. Master 端跑一个程序去生成任务（Request/url/ID）。Master 端负责的是生产任务，并把任务去重、加入到待爬队列。Slaver 只管从 Master 端拿任务去爬。
    - 优点： 将生成任务和抓取数据分开，分工明确，减少了 Master 和 Slaver 之间的数据交流；Master 端生成任务还有一个好处就是：可以很方便地重写判重策略（当数据量大时优化判重的性能和速度还是很重要的）。
    - 缺点： 像 QQ 或者新浪微博这种网站，发送一个请求，返回的内容里面可能包含几十个待爬的用户 ID，即几十个新爬虫任务。但有些网站一个请求只能得到一两个新任务，并且返回的内容里也包含爬虫要抓

取的目标信息，如果将生成任务和抓取任务分开反而会降低爬虫抓取效率。毕竟带宽也是爬虫的一个瓶颈问题，我们要秉着发送尽量少的请求为原则，同时也是为了减轻网站服务器的压力，要做一只有道德的 Crawler。所以，视情况而定。

3. Master 中只有一个集合，它只有查询的作用。Slaver 在遇到新任务时询问 Master 此任务是否已爬，如果未爬则加入 Slaver 自己的待爬队列中，Master 把此任务记为已爬。它和策略一比较像，但明显比策略一简单。策略一的简单是因为有 scrapy-redis 实现了scheduler 中间件，它并不适用于非 scrapy 框架的爬虫。
    - 优点： 实现简单，非 scrapy 框架的爬虫也适用。Master 端压力比较小，Master 与Slaver 的数据交流也不大。
    - 缺点：“健壮性”不够，需要另外定时保存待爬队列以实现“断点续爬”功能。各 Slaver的待爬任务不通用。

    如果把 Slaver 比作工人，把 Master 比作工头。策略一就是工人遇到新任务都上报给工头，需要 干活的时候就去工头那里领任务；策略二就是工头去找新任务，工人只管从工头那里领任务干活；策略 三就是工人遇到新任务时询问工头此任务是否有人做了，没有的话工人就将此任务加到自己的“行程 表”