

问题 01:

1. 简述闭包、lamda 表达式以及装饰器之间的异同。

异同

- 闭包:是在函数内部引用了一个变量且能够让里面的嵌套函数使用，外面无法使用该变量。
- lamda: 就是一个匿名函数，在使用的过程当中不需要定义函数名。
- 装饰器: 也是闭包，不过里面的变量需要传递进去，一般时候，这个变量也是可执行的函数。

三者之间，闭包和装饰器十分类似，或者说，装饰器就是通过闭包实现。

闭包更倾向于对变量的局部复用。

装饰器倾向于对代码的多次复用。

lamda 的实现方式和闭包类似，但它大多时候只是一个比较方便的表达式。

2. 如果可以，请分别给出一个场景，常用其中一者而非另外两个。

场景:

- lamda:通常我在使用 map 函数的时候搭配 lamda,因为 lamda 的简便，会让代码十分简洁。
- 闭包: 如果当前代码中有一个变量是经常使用的，且不会变动的，可以考虑封成一个闭包，把变量写死在里面。
- 装饰器: 装饰器的使用场景很广，当该代码段需要在另一段代码前置执行的时候，都可以考虑用装饰器的方式装饰其他函数。

3. 分别用这三者实现单例模式，并说明各个方式的好处。

单例模式

装饰器:

闭包和装饰器实现单例模式比较简单，装饰器内设置 isFlag，并进行判断。当 isFlag 为最初状态则继续，否则返回。

如果用 lamda 实现单例，应该首先解决变量和判断问题。最后根据状态返回空或一个 lamda 生成的表达式。

问题 02:

现有 N 个最大长度为 M 的字符串组成的数组 S，其中可能包含重复相同的字符串。请设计算法，将数组 S 中的字符串进行去重操作，得到不含有相同字符串的字符串集合。并分别分析算法时间复杂度。尽量设计低时间复杂度的算法。请使用一种你熟悉的编程语言写一个完整函数，用代码实现你的算法。

包含重复相同的字符串的两种情况:

1. 对于列表 S 内的字符串，可能存完全相同的字符串,需去重。
2. 对于列表 S 内的字符串内的字符，可能存在相同字符,需去重。

Python 中:

- 情况 1. 直接使用 set 函数去重.通过对比 hash 实现，时间复杂度 O(1)。
- 情况 2. 需要使用双循环. 时间负责度 O(mn)

第一循环获取 N 个字符串。

第二次循环获取 M 个字符。

然后通过获取该字符第一个下标的方式完成去重。

如下:

```
def func(list):  
    for n in len(list)-1:  
        st = list[n]  
        new_st = ''  
        for i in st:  
            index = st.find(i)  
            new_st[index] = st[index]  
        list[n] = new_st
```

问题 03:

贪吃蛇因场景简单，易于上手，是一个非常经典的游戏，简单来说，贪吃蛇通过在方形棋盘格上虚拟出蛇和食物的场景，玩家控制蛇的前进方向，蛇在“吃”到事物后变长，同时刷新新的食物，如此反复，直到蛇头撞到障碍物，结束游戏。假设现在 X 公司需要制作一款网络贪吃蛇游戏，而你是其中的服务端开发工程师，请根据如下要求完成服务端的设计并回答后续问题：

- 1) 游戏终端是浏览器：
- 2) 游戏逻辑运行和数据存储在服务端：
- 3) 以合适的方法用单主机支持 20 个连接(不必在同一场景内)。

- 1: 选择你熟悉的语言、方法、框架实现该服务端，通过一定方法描述你的方案架构、运行流程、必要的数据结构和数据流（方法包括但不限于文字描述、图表、代码及伪代码）。
- 2: 阐述你的设计思路及你觉得程序难点在哪，给出这些部分的伪代码，并解释为什么重要和为什么这么做。
- 3: 以下问题请根据时间自行安排，按照问题 2 的方式，挑选你擅长的进行解答：

- 1) 对于网络较差（包括丢包、带宽、延迟等）的情况下如何修改：
- 2) 对于用户数量和服务主机没有任何假设的情况下如何修改：
- 3) 如果用户在同一场景内进行游戏该如何修改。

1. 采用 Django 框架完成该服务器。

必要的数据结构:

- 用户:

id, 头像(金币等), 战绩.

- 蛇:

用户 id, 长度, 状态(是否死亡)

- 食物:

数量(对应棋盘), 刷新时间.

- 流程:

用户打开浏览器进入该界面, 通过 **session** 获取到用户 **id**. 从而显示对应的头像,战绩,金币等.

用户开始游戏, 创建棋盘, 初始化蛇的状态并随机分布等量食物.

开始比赛.

比赛结束, 保存该战局信息.

2. 对于我来说, 该程序难点在于. 比赛的过程的处理.

用户的蛇的实时位置.

如果逻辑处理放在服务器上的话, 根据每个用户控制的蛇的对应棋盘所在点进行判断的话. 那么首先要解决的就是延时问题.

如果蛇每移动一个坐标都需要上传新的位置到服务器的话, 那么对于服务器来说也是一个不小的负担.

如果一定需要由后端处理该逻辑的话.

我觉得可以由方向来判断蛇的状态.

只获取蛇的朝向.

假设蛇的移动速度是固定的, 每秒 **1** 格.

当游戏开始时候, 初始化蛇. 并获取蛇所在的坐标,和默认方向.

每当蛇进行转弯的时候.

后端计算出来方向改变的时间. 在这段时间内超这个方向移动了多少格. 然后计算与边框的距离.距离小于 **0**, 则蛇状态为 **false**.

每当蛇的方向发生变化时候, 都重新计算一下.

但是这个方法不能实现.. 撞自己逻辑.

所以需要在这个基础上, 添加对单元格的判断.

根据蛇的长度, 以及行走的单元格. 判断哪个格子的状态为 **false**, 当蛇行走走到该单元格, 则蛇状态为 **false**.

- 优化方面:

1. 如果网络条件较差, 一方面可以将一部分逻辑放在前端判断.一方面可以减少用户请求次数.

2. 如果多用户在同场景游戏, 数据库需要引入棋盘, 棋盘大小, 以及 每场游戏的战绩等.

多用户间互动.

除了对棋盘边框进行判断之外.

还要对其他用户的蛇的身体进行判断. (类似于上面 自己蛇身体的单元格 **false**)

分类: [ProblemSet](#)

问题 04:

警察局抓了 **a,b,c,d** 4 名偷窃嫌疑犯, 其中只有一人是小偷。审问中, **a** 说: “我不是小偷。”**b** 说: “**c** 是小偷。”**c** 说: “小偷肯定是 **d**。”**d** 说: “**c** 在冤枉人。”。其中有一个人说的是实话, 一个人说的是假话, 请编程推断谁是小偷: **c** 是小偷。

因为如果 A 说假 zhidao 话，那么 BCD 就是真话，可是 CD 前后矛盾，所以 A 是真话
由此一直推断 如果 D 说假话，那回 ABC 就是真话 因为答 BC 矛盾，所以 D 是真话
如果 C 是假话，那么 ABD 就是真话，与实际相符 ， 所以 C 是小偷。

思路：

四个犯人编号 1， 2， 3， 4，

a 说： thief != 1

b 说： thief = 3

c 说： thief = 4

d 说： thief != 4

四句话，三句话对的，一句话错的。

Python3 实现：

```
for i in range(4):  
    i += 1  
    if 3 == ((i != 1) + (i == 3) + (i == 4) + (i != 4)):  
        str = chr(96 + i) + "是小偷！ "  
        print(str)
```

问题 05：

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级…它也可以跳上 n 级，求该青蛙跳上一个 n 级的台阶总共有多少种跳法？

- 3. 分析：
- 4. A: $f(n) = f(n-1)+f(n-2)+\cdots+f(1)$
- 5. $f(n-1) = f(n-2)+ f(n-3)\cdots+f(1)$
- 6. 两式相减，得到 $f(n) = 2*f(n-1)$.
- 7. 方法（1）：找规律发现 $f(n)=2^{(n-1)}$
- 8. 方法（2）：可以分析，跳到当前台阶总可能数=前面所有台阶可能性+1.即 $f(n)=f(1)+f(2)+\ldots f(n-1)+1$,这个 1 代表直接跳到当前台阶
- 9. 分析：如果只有 1 级台阶，那显然只有一种跳法。如果有 2 级台阶，那就有两种跳法：一种是分两次跳，每次跳 1 级；另一种就是一次跳两级。将 n 级台阶时的跳法看成 n 的函数，记为 。当 $n>2$ 时，第一次跳的时候就有两种选择：一是第一次只跳 1 级，此时跳法数目等于后面剩下的 $n-1$ 级台阶的跳法数目，即 ；二是第一次跳 2 级，此时跳法数目等于后面剩下的 $n-2$ 级台阶的跳法数目，即 。因此，n 级台阶的不同跳法的总数 。因此这实际上就是斐波那契数列问题。
- 10. 代码：
- 11. def jumpFloor(n):
- 12. # 斐波那契初始化
- 13. fn = [1,1]

```
14.
15.     # 计算 f(n) = f(n-1) + f(n-2)
16.     for i in range(n-1):
17.         fn[0], fn[1] = fn[1], fn[0]+fn[1]
18.
19.     # 返回
20.     return fn[-1]
```

问题 06:

列表和元组的区别是什么？

总结

1. 列表是动态数组，它们不可变且可以重设长度（改变其内部元素的个数）。
2. 元组是静态数组，它们不可变，且其内部数据一旦创建便无法改变。
3. 元组缓存于 Python 运行时环境，这意味着我们每次使用元组时无须访问内核去分配内存。

区别：01：都是序列类型；02：列表是可变类型，元组是不可变类型；03：tuple 用于存储异构(heterogeneous)数据，当做没有字段名的记录来用，比如用 tuple 来记录一个人的身高、体重、年龄。而列表一般用于存储同构数据(homogenous)，同构数据就是具有相同意义的数据，比如下面的都是字符串类型。

列表和元组的区别

1. 列表可以看成是动态数组，它们是可变的并且可以重新设定长度
2. 元组可以看成是静态的数组，它们是不可变的，并且长度也是一旦创建就无法改变

从设计上来说：

1. 列表是用来保存多个相互独立对象的数据集合
2. 元组设计的初衷就是为了描述一个不会改变的事物的多个属性

列表常见的操作

列表后面增加一项 append

- 统计某个元素在列表中出现的次数 count
- 列表扩展,将另一个列表追加到原来的列表上 extend
- 获取元素的索引 获取的是第一次出现的索引 index
- 向指定的索引处插入指定元素 insert(index, element)
- 删除最后一个元素 pop() 并且返回的是删除之后的元素
- 删除指定的元素 remove('element') 删除的是第一个位 element 的元素
- 反转整个列表 reverse
- 对列表进行排序,直接在原列表上进行排序,默认按照元素的首字母进行排序 sort, 默认是升序进行排列

元组常见的操作

- 创建空元组
- 创建只有一个元素的元组的时候, 需要在后面加逗号, 不然会被当成其他的数据类型来处理
- 将列表转换为元组 使用 `tuple()`
- 查询
- 删除 元组的元素不支持删除, 但是可以删除整个元组对象
- 统计元组中某个元素出现的个数 `count`
- 查找元素的索引位置 `index`
- 元组的更新

元组是不可变的类型, 虽然在程序的运行中无法对元组的元素进行插入和删除运算. 但是可以利用对一个元组进行重新赋值的方式, 更新原来的元组.

- 元组的合并 元组的合并 是指几个元组相加形成新的元组, 原来的元组并没有改变

结论

元组和列表都是容器对象, 都可以存放不同类型的数据内容. 它们主要有两个不同点

第一: 列表的声明用中括号, 元组的声明用小括号, 并且元组只有一个元素的时候需要在后面加逗号

第二: 列表是可变的, 元组是不可变的. 元组一旦被定义, 里面的元素和个数就不能改变了.

问题 07:

将下面的函数按照执行效率高低排序。它们都接受由 0 至 1 之间的数字构成的列表作为输入。这个列表可以很长，一个输入列表的示例如下：[random.random() for i in range(100000)]。你如何证明自己的答案是正确的。

```
def f1(lin):
```

```
    l1 = sorted(lin)
    l2 = [i for i in l1 if i < 0.5]
    return [i*i for i in l2]
```

```
def f2(lin):
```

```
    l1 = [i for i in lin if i < 0.5]
    l2 = sorted(l1)
    return [i*i for i in l2]
```

```
def f3(lin):
```

```
    l1 = [i*i for i in lin]
    l2 = sorted(l1)
    return [i for i in l2 if i<(0.5*0.5)]
```

按执行效率从高到低排列：**f2**、**f1** 和 **f3**。要证明这个**答案**是对的，你应该知道如何分析自己代码的性能。**Python** 中有一个很好的程序分析包，可以满足这个需求。

```
import cProfile
```

```
lIn = [random.random() for i in range(100000)]
```

```
cProfile.run('f1(1In)')
cProfile.run('f2(1In)')
cProfile.run('f3(1In)')
```

问题 08:

请应用 RBAC（基于角色控制）权限系统设计相关数据库表结构？（列出主要表、字段和类型）

bac 支持两种类，PhpManager（基于文件的） 和 DbManager（基于数据库的）

权限：

就是指用户是否可以执行哪些操作

角色：

就是上面说的一组操作的集合，角色还可以继承

在 Yii2.0 中

- - yii\rbac: Item 为角色或者权限的基类，其中用字段 type 来标识
 - yii\rbac: Role 为代表角色的类
 - yii\rbac: Permission 为代表权限的类
 - yii\rbac: Assignment 为代表用户角色或者权限的类
 - yii\rbac: Rule 为代表角色或权限能否执行的判定规则表

在数据库中存储的 RBAC 层次是低效率的浪费性能的，但要灵活得多

存储角色或权限的表：auth_item

用来存储角色和权限的数据，Role 类和 Permission 类有一个共同的基类 yii\rbac: Item，用\$type 字段来标识是角色还是权限。

角色权限关联表：auth_item_child

用来保存角色和权限的关系

用户角色（权限）表：auth_assignment

用户的权限包含两部分，一部分是所指定的角色代表的权限，一部分就是直接所指定的权限

规则表：auth_rule

如果要在规则表：[auth_rule]中增加一条规则就得要有对应的规则类，并实现方法 **abstract public function execute(\$item, \$params)**具体的逻辑来判定\$item（角色或者权限）是否可执行。

【转】sql 如何设计数据库表实现完整的 RBAC(基于角色权限控制)

RBAC(基于角色的权限控制)是一个老话题了，但是这两天我试图设计一套表结构实现完整的 RBAC 时，发现存在很多困难。

我说的完整的 RBAC，是指支持角色树形结构和角色分组。具体来说，应当包含如下权限控制需求：

1. 父级角色可以访问甚至是修改其子级的数据，包含直接子级直到最终子级。

2. 角色可以访问其所在组的数据。
3. 父级角色可以访问其所有子级(从直接子级到最终子级)所在组的数据。

而具体到我的系统中，还应当有如下需求。

1. 兼容多种数据库产品。只能用简单的表，视图，存储过程和函数等实现。
2. 同时兼容单条数据处理的和批量数据处理的需求。

且不论这些具体需求，RBAC 的基本表应当如下四个：

- **roleList** 表，记录所有的角色和角色组。
 - **roleId**: PK, 角色/组的 ID, 全局唯一，不区分角色和组。
 - **roleName**: 角色/组的名称。
 - **roleType**: R - 角色, G - 组
- **rolePermission** 表，记录每一个角色/组对每一个对象的权限。
 - **permissionID**: PK, 无特定意义。
 - **role**: 角色/组的 ID。
 - **object**: 对象的 ID。
 - **permission**: 权限标识，如读，写，删等。
- **roleRelationship** 表，记录角色/组之间的关系。
 - **relationId**: PK, 无特定意义。
 - **superiorRole**: 父角色/组的 ID。
 - **role**: 子角色，子组，成员角色，成员组的 ID。
 - **relationship**: 关系标识，可在如下设置集中选取一个。
 - **PG** 标识: P - 父子关系, G - 组/成员关系。
 - **PPGG** 标识: 在 PG 集上，再加三种: PP - 间接父级关系, GG - 组内组关系, CG - parentRole 是组, childRole 的子角色或间接子角色是其成员，或其子组(含间接子组)的成员
- **objectList** 表，记录所有的对象。
 - **objectId**: PK, 对象 ID, 全局唯一。
 - **objectName**: 对象名称。
 -

分析上述表结构，不难发现，问题的关键在于从 **rolePermission** 表中读取数据时，如何限定角色/组的范围。

方案一

如果角色和组的总量不大，比如在 100 以内，采用 PPGG 标识关系，读取数据时是最快的。这个时候的 SQL 只需要一个输入参数?roleId:

SELECT object FROM rolePermission p left join roleRelationship r on p.role = r.role WHERE p.role = ?roleId or r.superiorRole = ?roleId. (尚未验证 SQL 的正确性)

但是，这个方案是以极度冗余 **roleRelationship** 表的数据为代价的，比如有 100 个角色，那么 **roleRelationship** 中将会有 $100 * 100 = 10,000$ 条记录。而在每次调整角色和 R 角色组的时候，就要在 **roleRelationship** 中一次增加或删除 100 条记录。这个开销是比较大的。

方案二

只标识 PG，查询时接收的输入参数为一个完整的相关角色列表?roleList。

SELECT object FROM rolePermission WHERE role in (?roleList)

在系统运行时，这个?roleList 通常可以从 **role hierarchy cache** 中取到，比较方便。这个方案的主要问题有二：

- 1) 如果?roleList 过长，使用 in 判断性能会很差。
- 2) 在有些情况下，如报表查询和系统外查询时，取得 **roleList** 不太方便。

方案三

只标识 PG，但使用如下三个数据库函数来判断角色/组之间的关系。

- **boolean isChild(role, parentRole)** - 如 role 为 parentRole 的子，返回 true。
- **boolean isDescendant(role, ancestorRole)** - 如 role 为 ancestorRole 的子或间接子级，返回 true。
- **boolean isMember(role, group)** - 如 role 为 group 的成员或子组的成员，返回 true。
- **boolean descendantIsMember(role, group)** - 如 role 的子或间接子级为 group 的成员，返回 true。
- **boolean isBelong(role, super)** - 如 role 为 super 的子，间接子，成员或间接员，或者 role 的子(含间接子)是 super 的成员或子组成员，返回 true。

在查询时，也只需要接收一个?roleId: **SELECT object FROM rolePermission WHERE isBelong(?roleId, role)**

如何写出高性能的数据库函数是实现这个方法的关键。

上述方法仅是理论分析，我倾向于方案二。

终于想到新的方案了。

方案四，

结合方案一和方案二，在 **roleRelationship** 中，对前两级(也可以是三级或四级)角色，保存其所有的下级角色和组。这样，如果以前两级角色查询数据，就使用方案一，如果以第三级及以下的角色查询数据，就使用方案二。

仍以 100 个角色为例，每个角色要保存三个关系：一级主管角色，二级主管角色，直接主管角色，最多有 300 条数据。

每往角色组中加一个角色，也需要加入三条数据：角色本身，一级主管角色，二级主管角色。

但往角色组中加一个子组，需要加入的数据量就大一些：子组本身，子组所有角色，子组所有角色的一级主管角色和二级主管角色。如在多个子组中发现同一角色，可重复保存，但应在表中附加说明是由哪个子组导入的。这样在删除子组时就可以有选择性的删除。

但重复子组的情况就比较麻烦，还有等考虑。假充有组 g01,g11,g12,g21。g01 包含 g11 和 g12，g11 和 g12 分别包含 g21。从 g01 中删除 g11 时，如何判断 g21 的去留?看来还是应当在维护时判断应不应当删除。

分类: [SQL](#)

标签: [sql](#)

问题 09:

用 Django 开发一个支持评论的博客系统，代码上传到 Github，系统部署到 Heroku。

从开发到部署，使用 django 创建一个简单可用的个人博客

本文参考于：

简书－Django 搭建简易博客教程：<http://www.jianshu.com/p/d15188a74104>

自强学堂－Django 基础教程：<http://www.ziqiangxuetang.com/django/django-tutorial.html>

Django 官方文档中文翻译版：<http://python.usyiyi.cn/django/index.html>

本文主要是一步一步教大家如何使用 Django 构建一个自己的博客，基础的 django 不会讲的太详细，想要详细学习 django 的同学可以参考上面三个教程

本文的所有代码将托管于 github：https://github.com/w392807287/django_blog_tutorial

本文开发环境：

- django 1.8.3
- ubuntu 16.04
- python 2.7.12

迈出第一步

首先，安装 django，这里我们使用的是 1.8.3 版本

1	<code>sudo pip install django==1.8.3</code>
---	---

创建 django 项目并开始一个新的 app，如果是使用 pycharm 的同学可跳过

1	<code>django-admin startproject tutorial</code> <code>cd tutorial</code> <code>python manage.py startapp blog</code>
2	
3	

在 settings.py 中加入 blog，然后

1	<code>python manage.py migrate</code> <code>python manage.py runserver</code>
2	

尝试访问 <http://127.0.0.1:8000/>
访问成功！

为你的博客创建一个模型

迈出第一步之后，我们已经有一个可以访问的项目了，现在我们需要为其创建模型，模型是 Django 提供一个抽象层（Models）以构建和操作你的 web 应用中的数据。
打开 `blog/models.py` 文件

```
1  #coding:utf8
2  from __future__ import unicode_literals
3
4  from django.db import models
5
6  # Create your models here.
7
8  class Article(models.Model):
9      title = models.CharField(u"博客标题",max_length = 100)                #博客标题
10     category = models.CharField(u"博客标签",max_length = 50,blank = True)    #博客标签
11     pub_date = models.DateTimeField(u"发布日期",auto_now_add = True,editable=True)    #博客发布
12     日期
13
14     update_time = models.DateTimeField(u'更新时间',auto_now=True,null=True)
15     content = models.TextField(blank=True, null=True)    # 博客文章正文
16
17     def __unicode__(self):
18         return self.title
19
20     class Meta:
21         #按时间下降排序
22         ordering = ['-pub_date']
23         verbose_name = "文章"
24         verbose_name_plural = "文章"
```

这样我们就创建了第一个属于我们博客的模型——文章。
然后，然后我们就同步数据库咯

1	<code>python manage.py makemigrations</code>
2	<code>python manage.py migrate</code>

登陆管理后台

首先我们先创建一个超级用户，用来登陆后台管理

1	<code>python manage.py createsuperuser</code>
---	---

然后在 `blog/admin.py` 中加入代码：

1	<code>#coding:utf8</code>
2	
3	<code>from django.contrib import admin</code>
4	<code>from blog.models import Article</code>
5	<code># Register your models here.</code>
6	
7	<code>class ArticleAdmin(admin.ModelAdmin):</code>
8	<code> list_display = ('title', 'pub_date')</code>
9	
10	<code>admin.site.register(Article, ArticleAdmin)</code>

现在

1	<code>python manage.py runserver</code>
---	---

就能访问了，管理页面在 <http://127.0.0.1:8000/admin/>

但是现在页面不是那么好看，而且界面是英文的，那么我们需要做一些修改

首先我们先安装一个 **bootstrap** 的插件

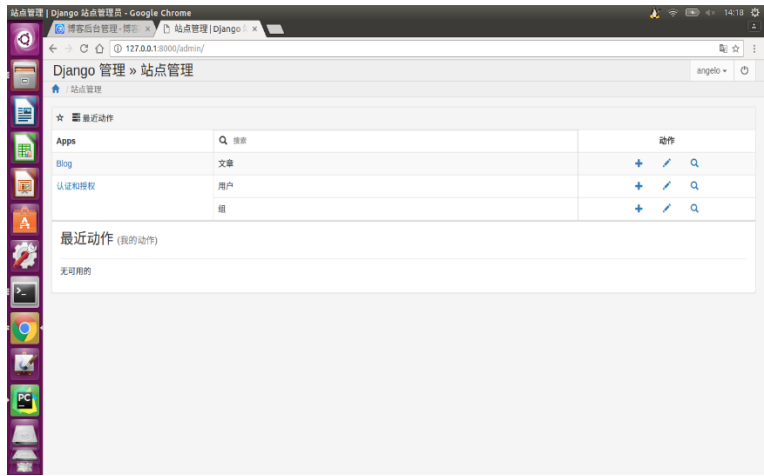
1	(sudo) pip install bootstrap-admin
---	------------------------------------

然后在 `tutorial/settings.py` 中更改一些代码

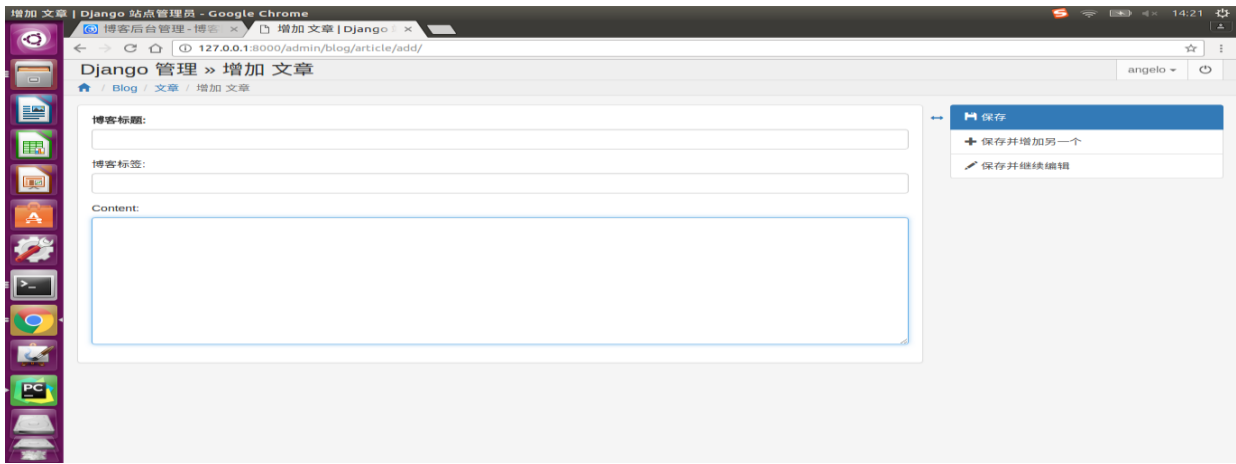
1	INSTALLED_APPS = (
2	'bootstrap_admin',
3	'django.contrib.admin',
4	'django.contrib.auth',
5	'django.contrib.contenttypes',
6	'django.contrib.sessions',
7	'django.contrib.messages',
8	'django.contrib.staticfiles',
9	'blog',
10)

1	LANGUAGE_CODE = 'zh-hans'
2	
3	TIME_ZONE = 'Asia/Shanghai'

红色字样是修改后的，首先插入 **bootstrap** 的管理页面插件，然后语言设置为中文，时区更改为中国。
现在运行就能看到一个正常的界面了



然后可以尝试添加一篇博客



集成 DjangoUeditor 编辑器及静态文件的处理

以上，界面是好看了不少，但是这个文章是不是有点简陋啊，只能写文本性的东西。我们现在来给它加一点点东西。我们下面来集成百度的 Ueditor 到我们的系统：

<https://github.com/twz915/DjangoUeditor3>

上面是 DjangoUeditor 包，可以下载 zip 或者直接 clone，将里面的额 DjangoUeditor 直接放到项目根目录，

1	ls
2	blog db.sqlite3 DjangoUeditor manage.py README.md templates tutorial

然后在 `tutorial/settings.py` 中加入

```
1  INSTALLED_APPS = (  
2      ...  
3  
4      'blog',  
5      'DjangoUeditor',  
6  )
```

这是为了让 `django` 能识别这个模块
在 `tutorial/url.py` 中加入

```
1  from django.conf.urls import include, url  
2  from django.contrib import admin  
3  from DjangoUeditor import urls as djud_urls  
4  from django.conf import settings  
5  
6  urlpatterns = [  
7      url(r'^admin/', include(admin.site.urls)),  
8      url(r'^ueditor/', include(djud_urls)),  
9  ]  
10  
11  if settings.DEBUG:  
12      from django.conf.urls.static import static  
13      urlpatterns += static(settings.MEDIA_URL, document_root = settings.MEDIA_ROOT)
```

这是为了让 `django` 能访问编辑器模块
然后在 `tutorial/settings.py` 中加入

```
1  STATIC_URL = '/static/'  
2  STATIC_ROOT = os.path.join(BASE_DIR, 'static')  
3  
4  #公共的 static 文件  
5  STATICFILES_DIRS = (  
6      os.path.join(BASE_DIR, "common_static"),
```

```
7         os.path.join(BASE_DIR, "media"),
8     )
9
10    #upload floder
11    MEDIA_URL  = '/media/'
12    MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
13
14    STATICFILES_FINDERS = ("django.contrib.staticfiles.finders.FileSystemFinder",
15                           "django.contrib.staticfiles.finders.AppDirectoriesFinder",)
```

这是静态文件的配置，很多初学 **django** 的同学会在这里碰到坑。下面一个一个解释一下

- | | |
|---|--|
| 1 | STATIC_URL 这个是放置静态文件的地方，django 会默认在这个文件夹中寻找需要的静态文件，很多教程教大家上来就把静态文件塞这里面其实这并不是一个好的处理方法，因为在发布前需要统一收集静态文件的时候会从各个文件夹中收集静态文件放入这个文件夹中，期间有可能会覆盖掉原来的文件。 |
|---|--|
- | | |
|---|------------------------------|
| 1 | STATIC_ROOT 这个就是静态文件相对于系统的目录 |
|---|------------------------------|
- | | |
|---|----------------------------|
| 1 | MEDIA_URL 一般会将上传的文件放入这个文件夹 |
|---|----------------------------|
- | | |
|---|--------------------------|
| 1 | MEDIA_ROOT 同 STATIC_ROOT |
|---|--------------------------|
- | | |
|---|---|
| 1 | STATICFILES_DIRS 这一个元组，里面放置开发时静态文件自动搜寻的目录，我们在开发是先建一个 common_static 即公用的静态文件夹，在里面放我们自己的静态文件，等最后使用静态文件收集命令一并处理。 |
|---|---|

然后我们更改 **blog/models.py** 中的模型

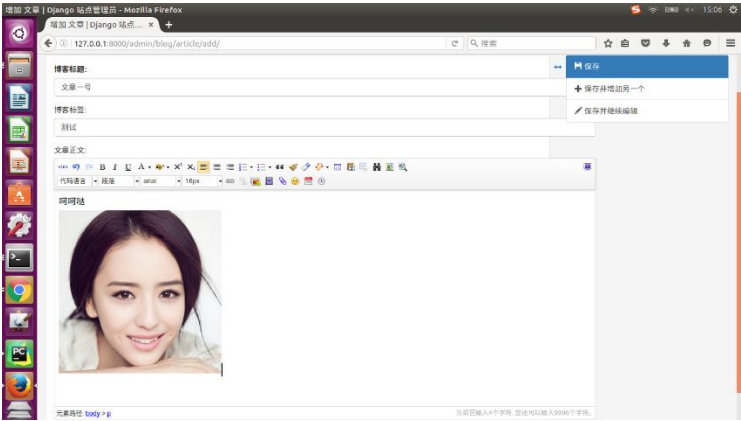
```
1 from DjangoUeditor.models import UEditorField
2
3
```


4	content = UEditorField(u"文章正文"
5	,height=300,width=1000,default=u'',blank=True,imagePath="uploads/blog/images/", toolbars='besttome',filePath='uploads/blog/files/')

跟新数据库

1	python manage.py makemigrations
2	python manage.py migrate

然后运行一下



好了，现在我们的编辑器就是一个能写文字能传图还能自动保存的“厉害编辑器”了，科科。
至此，自己用的算是告一段落，那么，作为一个博客，时需要给别人看的啊，所以下面写一些给别人看的东西。

views 和 urls

我们在 `blog/views.py` 中加入：

1	def Test(request):
2	return HttpResponse("just a test")

在 `blog` 中添加 `urls.py`:

```
1 #coding:utf8
2 from django.conf.urls import url
3 from . import views
4
5 urlpatterns = [
6     url(r'^test/', views.Test, name="blog_test"),
7 ]
```

在 `tutorial` 的 `urls` 中引入 `blog` 的 `urls`:

```
1 import blog.urls as blog_url
2
3 urlpatterns = [
4     url(r'^blog/', include(blog_url)),
5     url(r'^admin/', include(admin.site.urls)),
6     url(r'^ueditor/', include(djud_urls)),
7 ]
```

运行后可以正常访问, 返回 **just a test**。

我们尝试这通过视图 `views` 访问数据库, 更改 `/blog/views.py`:

```
1 from blog.models import Article
2
3 def Test(request):
4     post = Article.objects.all()
5     return HttpResponse(post[0].content)
```

如果你刚在后台添加了文章那么就能看到它了!

使用 **Template** 来展示

在项目目录下有一个名叫 **templates** 的文件夹，如果没有你就建一个咯又花不了多大劲儿。

新建一个 **html** 文件 **templates/blog/test.html**:

1	<!DOCTYPE html>
2	<html>
3	<head>
4	<title>Just test template</title>
5	<style>
6	body {
7	background-color: red;
8	}
9	em {
10	color: LightSeaGreen;
11	}
12	</style>
13	</head>
14	<body>
15	<h1>Hello World!</h1>
16	{{ current_time }}
17	</body>
18	</html>

修改 **views.py** :

1	def Test(request) :
2	return render(request, 'blog/test.html', {'current_time': datetime.now()})

现在运行访问 **http://127.0.0.1:8000/blog/test** 就能看到 **hello world** 和当前时间.

现在我们新建 3 个 **html** 文件

base.html:

1	<!doctype html>
---	-----------------

```
2 <html lang="zh-hans">
3 <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <meta name="description" content="A layout example that shows off a blog page with a list of posts.">
7     <title>李琼羽的博客</title>
8     <link rel="stylesheet" href="http://yui.yahooapis.com/pure/0.5.0/pure-min.css">
9     <link rel="stylesheet" href="http://yui.yahooapis.com/pure/0.5.0/grids-responsive-min.css">
10    <link rel="stylesheet" href="http://picturebag.qiniudn.com/blog.css">
11 </head>
12 <body>
13 <div id="layout" class="pure-g">
14     <div class="sidebar pure-u-1 pure-u-md-1-4">
15         <div class="header">
16             <h1 class="brand-title"><a href="{% url 'blog_home' %}">Angelo Li Blog</a></h1>
17             <h2 class="brand-tagline">李琼羽 - Angelo Li</h2>
18             <nav class="nav">
19                 <ul class="nav-list">
20                     <li class="nav-item">
21                         <a class="button-success pure-button" href="/">主页</a>
22                     </li>
23                     <li class="nav-item">
24                         <a class="button-success pure-button" href="/">归档</a>
25                     </li>
26                     <li class="nav-item">
27                         <a class="pure-button" href="/">Github</a>
28                     </li>
29                     <li class="nav-item">
30                         <a class="button-error pure-button" href="/">微博</a>
31                     </li>
32                     <li class="nav-item">
33                         <a class="button-success pure-button" href="/">专题</a>
34                     </li>
35                     <li class="nav-item">
36                         <a class="button-success pure-button" href="/">About Me</a>
```



```

12 href="#">{{ post.category }}</a>
13                                     </p>
14                                     </header>
15
16                                     <div class="post-description">
17                                         <p>
18                                             {{ post.content|safe }}
19                                         </p>
20                                     </div>
21                                     </section>
</div><!-- /.blog-post -->
{% endblock %}

```

其中, `home.html` 和 `post.html` 继承于 `base.html`
更改 `blog/views.py`

```

1  #coding:utf8
2
3  from django.shortcuts import render
4  from django.http import HttpResponse
5  from blog.models import Article
6  from datetime import datetime
7  from django.http import Http404
8  # Create your views here.
9
10 def home(request):
11     post_list = Article.objects.all()    # 获取全部的 Article 对象
12     return render(request, 'blog/home.html', {'post_list': post_list})
13
14 def Test(request):
15     return render(request, 'blog/test.html', {'current_time': datetime.now()})
16
17 def Detail(request, id):
18     try:

```

```

19         post = Article.objects.get(id=str(id))
20     except Article.DoesNotExist:
21         raise Http404
22     return render(request, 'blog/post.html', {'post': post})

```

blog/urls.py

```

1  #coding:utf8
2  from django.conf.urls import url
3  from . import views
4
5  urlpatterns = [
6      url(r'^post/(?P<id>\d+)/$', views.Detail, name="blog_detail"),
7      url(r'^home/', views.home, name="blog_home"),
8      url(r'^test/', views.Test, name="blog_test"),
9  ]

```

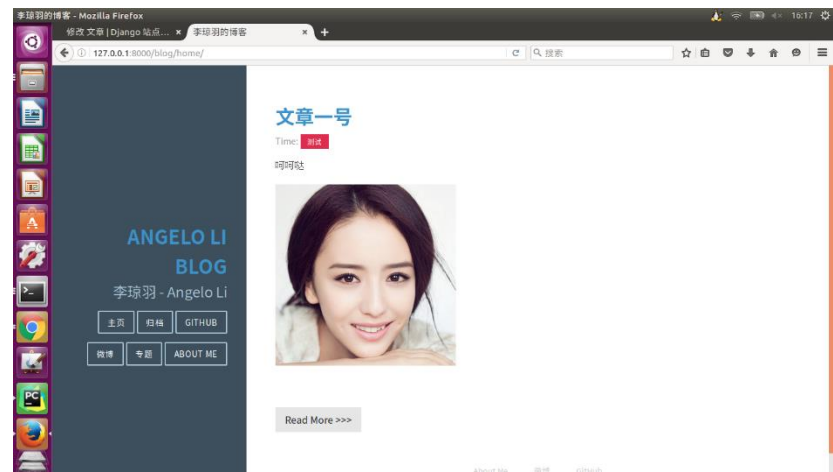
tutorial/urls.py

```

1  from django.conf.urls import include, url
2  from django.contrib import admin
3  import blog.urls as blog_url
4  from DjangoUeditor import urls as djud_urls
5  from django.conf import settings
6
7  urlpatterns = [
8      url(r'^blog/', include(blog_url)),
9      url(r'^admin/', include(admin.site.urls)),
10     url(r'^ueditor/', include(djud_urls)),
11 ]
12
13 if settings.DEBUG:
14     from django.conf.urls.static import static
15     urlpatterns += static(settings.MEDIA_URL, document_root = settings.MEDIA_ROOT)

```


运行访问 <http://127.0.0.1:8000/blog/home/>



以上，开发过程已经完成，当然你可以自己加入需要的东西，比如评论什么的。

使用 uWSGI+nginx 部署 Django 项目

详细步骤见 <http://www.cnblogs.com/Liqiongyu/articles/5893780.html>

这里只讲主要的步骤

首先安装：

1	<code>sudo apt-get install nginx</code>
2	<code>sudo pip install uwsgi</code>

更改 nginx 配置文件

1	<code>sudo vim /etc/nginx/sites-available/default</code>
---	--

如下

1	<code>upstream django {</code>
2	<code>server unix:///home/ubuntu/blogsite/mysite.sock; # for a file socket</code>
3	<code>}</code>
4	

```
5  server {
6      listen 80 default_server;
7      listen [::]:80 default_server ipv6only=on;
8
9      root /usr/share/nginx/html;
10     index index.html index.htm;
11
12     # Make site accessible from http://localhost/
13     server_name localhost;
14
15     charset utf-8;
16
17     fastcgi_connect_timeout 300;
18     fastcgi_send_timeout 300;
19     fastcgi_read_timeout 300;
20
21     client_max_body_size 75M;
22
23     location /media {
24         alias /home/ubuntu/blogsite/media;
25     }
26
27     location /static {
28         alias /home/ubuntu/blogsite/static;
29     }
30
31     location / {
32         # First attempt to serve request as file, then
33         # as directory, then fall back to displaying a 404.
34         #try_files $uri $uri/ =404;
35         # Uncomment to enable naxsi on this location
36         # include /etc/nginx/naxsi.rules
37         uwsgi_pass          django;
38         include              /etc/nginx/uwsgi_params;
39     }
```

40	}
----	---

1	上面的/home/ubuntu/blogsite 是我的项目目录，全部改成你自己的就 O K 然后执行命令：
1	python manage.py collectstatic

这是用来收集静态文件的，上面有提到

1	 然后在项目根目录添加 mysite_uwsgi.ini 文件：
1	[uwsgi]
2	
3	# Django-related settings
4	# the base directory (full path)
5	chdir = /home/ubuntu/blogsite
6	# Django's wsgi file
7	module = blogsite.wsgi
8	# the virtualenv (full path)
9	# home = /path/to/virtualenv
10	
11	# process-related settings
12	# master
13	master = true
14	# maximum number of worker processes
15	processes = 2
16	# the socket (use the full path to be safe
17	socket = /home/ubuntu/blogsite/mysite.sock
18	# ... with appropriate permissions - may be needed
19	chmod-socket = 666
20	# clear environment on exit
21	vacuum = true

然后修改 tutorial/settings.py :

1	DEBUG = False
2	
3	ALLOWED_HOSTS = ['*']

解除 debug 模式，allowed_hosts 中添加你的域名，这里为方便填*
然后重启 nginx,运行 uwsgi

1	sudo service nginx restart
2	uwsgi --ini mysite_uwsgi.ini

访问 <http://youdomain.com/blog/home>
代码更改了 url 后可以直接访问 youdomain.com
以上，网站就能正常访问。有问题欢迎微信留言
代码地址：https://github.com/w392807287/django_blog_tutorial
欢迎多来访问博客：<http://liqiongyu.com/blog>
微信公众号：

问题 10：
学会 Vue 基础，能自己使用 ES6 语法重新实现官方的 TodoMVC 示例，并将其数据存储到后端数据库，并部署上线。略

问题 10：
通过 Dockerfile 构建一个镜像，能够提供一个 Django 服务，然后通过 webhook 实现自动发布，自动部署。略

问题 11：
给定一个未排序的整数数组，找出最长连续序列的长度，要求算法的时间复杂度为 O(n).

示例：
输入：[9, 14, 7, 3, -1, 0, 5, 8, -1,6]
输出：5

解释：最长连续序列是[5, 6, 7, 8, 9]。它的长度为 5。
题意倒是很容易懂。关键是要求时间复杂度为 o(n)，那就不能用排序来做了。这就比较麻烦。
我们的方法其实也挺简单的，就是借助 HashSet<Integer>保存下来。

然后遍历节点，每当遍历到一个新节点的时候，就把跟它连着的所有节点都到 HashSet 中查一遍。

算法思路
此题难度定位为 hard，实则没这么困难，难就难在题目要求时间复杂度必须为 O(n),也就是最多遍历一遍。既然有时间复杂度的要求，说明需要额外的空间，通过牺牲空间来换取时间。此处数组不好删除，所以，引入一个集合 HashSet，存放数组元素，然后遍历每个元素，先前和向后查找连续的元素，并删除（防止重复计算），具体步骤如下：

将数组元素加入集合 set 中;

依次遍历数组的每个元素, 然后判断该元素是否存在 set 集合中, 存在就删除;

从当前元素开始向前查找, 每次减 1, 如果当前元素减一后在 set 中存在, 就 remove 该元素, 当次长度自增 1, 直到不存在为止;
向后同理;

更新 res 的值, 并返回。

程序:

class Solution:

```
def longestConsecutive(self, nums: List[int]) -> int:
```

```
    nums.sort()
```

```
    length = len(nums)
```

```
    if length <= 0:
```

```
        return 0
```

```
    if length == 1:
```

```
        return 1
```

```
    max_count = 1
```

```
    counter = 1
```

```
    for index in range(1,length):
```

```
        if nums[index - 1] == nums[index]:
```

```
            continue
```

```
        if nums[index - 1] < nums[index] and nums[index] - nums[index - 1] == 1:
```

```
            counter += 1
```

```
            max_count = max(max_count, counter)
```

```
        else:
```

```
            counter = 1
```

```
            continue
```

```
    return max_count
```

问题 12:

怎样才能测试到改动一个模块所引起相关联模块?

问题 13:

Selenium 怎样去选择一个下拉框中的 value=xx 的 option?

select 菜单

select 也是比较常见的, selenium 封装了以下方法

创建 select

```
WebElement selector = driver.findElement(By.id("Selector"));
```

```
Select select = new Select(selector);
```

选择 select 的 option 有以下三种方法

- selectByIndex(int index) 通过 index
- selectByVisibleText(String text) 通过匹配到的可见字符
- selectByValue(String value) 通过匹配到标签里的 value

问题 14:

webdriver 的协议是什么?

当您打开 Selenium 的官方页面时, 您首先阅读的内容是“Selenium 自动化浏览器”在“什么是 Selenium?”中。部分。“Selenium 的哪一部分适合我?” 下面提供了 Selenium WebDriver 和 Selenium IDE 之间的选择。据此, 我推断 Selenium 是一个工具集合, 该集合包括 IDE, WebDriver API (语言绑定), Grid, Selenium Standalone Server, 浏览器驱动程序。必须下载适当的项目来构建项目。

什么是 WebDriver?

WebDriver 是一个 API。它用不止一种语言编写, 它们被称为语言绑定。API 具有控制浏览器的功能。您可以使用这些函数编写一个脚本, 以您想要的方式(测试用例)控制浏览器。

1、webdriver client 的原理是什么?

当测试脚本启动 firefox 的时候, selenium-webdriver 会首先在新线程中启动 firefox 浏览器。如果测试脚本指定了 firefox 的 profile, 那么就以该 profile 启动, 否则的话就新启 1 个 profile, 并启动 firefox;

firefox 一般是以-no-remote 的方法启动, 启动后 selenium-webdriver 会将 firefox 绑定到特定的端口, 绑定完成后该 firefox 实例便作为 webdriver 的 remote server 存在;

客户端(也就是测试脚本)创建 1 个 session, 在该 session 中通过 http 请求向 remote server 发送 restful 的请求, remote server 解析请求, 完成相应操作并返回 response

客户端接受 response, 并分析其返回值以决定是转到第 3 步还是结束脚本;

webdriver 是按照 server – client 的经典设计模式设计的

2、webdriver 的协议是什么?

The WebDriver Wire Protocol

3、启动浏览器的时候用到的是哪个 webdriver 协议?

http

在我们 new 一个 webdriver 的过程中, Selenium 首先会确认浏览器的 native component 是否存在可用而且版本匹配。接着就在目标浏览器里启动一整套 web service(实际上就是浏览器厂商提供的 driver, 比如 IEDriver, ChromeDriver, 他们都实现了 WebDriver's wire protocol), 这套 web service 使用了 Selenium 自己设计定义的协议, 名字叫做 The WebDriver Wire Protocol。这套协议非常之强大, 几乎可以操作浏览器做任何事情, 包括打开、关闭、最大化、最小化、元素定位、元素点击、上传文件等等。

WebDriver Wire 是通用的, 也就是说不管是 FirefoxDriver 还是 ChromeDriver, 启动之后都会在某一个端口启动基于这套协议的 Web Service。例如 FirefoxDriver 初始化成功之后, 默认会从 http://localhost:7055 开始, 而 ChromeDriver 则大概是 https://localhost:46350 之类的。接下来, 我们调用 WebDriver 的任何 API, 都需要借助一个 CommandExecutor 发送一个命令, 实际上是一个 HTTP request 给监听端口上的 webservice。在我们的 HTTP request 的 body 中, 会以 webdriver wire 协议规定的 JSON 格式的字符串来告诉 Selenium 我们希望浏览器接下来做什么事情。

可以更通俗的理解: 由于客户端脚本(java, python, ruby)不能直接和浏览器通信, 这时候可以把 webservice 当做一个翻译器, 它可以把客户端代码翻译成浏览器可以识别的代码(比如 js), 客户端(也就是测试脚本)创建一个 session, 在该 session 中通过 http 请求向 webservice 发送 restful 的请求, webservice 翻译成浏览器懂得脚本传给浏览器, 浏览器把执行的结果返回给 webservice, webservice 把返回的结果做了一些封装(一般都是 json 格式), 然后返回给 client, 根据返回值就能判断对浏览器的操作是不是执行成功。

摘自官网对于 chrome driver 的描述:

The ChromeDriver consists of three separate pieces. There is the browser itself("chrome"), the language bindings provided by the Selenium project("the driver") and an executable downloaded from the Chromium project which acts as a bridge between "chrome" and the "driver". This executable is called "chromedriver", but we'll try and refer to it as the "server" in this page to reduce confusion.

大概意思就是我们下载的 chrome 可执行文件(.exe) 是为作为浏览器与 client(language binding)桥梁的作用，也更印证了对于 webservice(driver)的理解。

[WebDriver 工作原理](#)

WebDriver 是 W3C 的一个标准，由 Selenium 主持。

具体的协议标准可以从 http://code.google.com/p/selenium/wiki/JsonWireProtocol#Command_Reference 查看。

从这个协议中我们可以看到，WebDriver 之所以能够实现与浏览器进行交互，是因为浏览器实现了这些协议。这个协议是使用 JSON 通过 HTTP 进行传输。

它的实现使用了经典的 Client-Server 模式。客户端发送一个 request，服务器端返回一个 response。

我们明确几个概念。

Client

编辑代码，发送代码。注意：代码内部已经包装了 webdriver 协议。

Server

运行浏览器的机器。浏览器则通过驱动程序来实现了 WebDriver 的通讯协议，如：Chrome 和 IE 则是通过 ChromeDriver 和 [InternetExplorerDriver](#) 实现的。

Session

服务器端需要维护浏览器的 Session，客户端首次发送请求的字符串是 '/session/\$sessionId/url'。服务器端将根据 url 打开对应的 url 地址，同时将 \$sessionId 解析成真实的值。然后返回给客户端。以后客户端再向浏览器发送请求时，将会携带 session 值一起发送。

问题 15:

点击链接以后，selenium 是否会自动等待该页面加载完毕？

不会的。所以有的时候，当 selenium 并未加载完一个页面时再请求页面资源，则会误报不存在此元素。所以首先我们应该考虑判断，selenium 是否加载完此页面。其次再通过函数查找该元素。

问题 16:

selenium 中用什么函数判断元素是否存在？

判断元素是否存在和是否出现不同，判断是否存在意味着如果这个元素压根就不存在，就会抛出 NoSuchElementException

这样就可以使用 try catch，如果 catch 到 NoSuchElementException 就返回 false。通常在项目中会把这个功能封装在 isElementPresent 方法中。

[小新人~](#)

[面试——selenium 常见面试题及答案（java）](#)

自动化测试面试——selenium 基础篇

目的：考察求职者对自动化测试岗位的 selenium 工具的熟悉程度

1.怎么判断元素是否存在？

判断元素是否存在和是否出现不同，判断是否存在意味着如果这个元素压根就不存在，就会抛出 NoSuchElementException

这样就可以使用 try catch，如果 catch 到 NoSuchElementException 就返回 false。通常在项目中会把这个功能封装在 isElementPresent 方法中。

2.如何判断元素是否出现？

判断元素是否出现，存在两种情况，一种是该元素压根就没有，自然不会出现；另外一种是有这样的元素，但是是 hidden 状态

可以通过先判断是否存在，如果不存在返回 false；如果存在再去判断是否 displayed。

3.selenium 中 hidden 或者是 display = none 的元素是否可以定位到？

不能，想点击的话，可以用 js 去掉 display=none 的属性。

4. selenium 中如何保证操作元素的成功率？也就是说如何保证我点击的元素一定是可以点击的？

- 1.通过封装 **find** 方法实现 **waitForEmelentPresent**，这样在对元素进行操作之前保证元素被找到，进而提高成功率
- 2.在对元素操作之前，比如 **click**，如果该元素未 **display**（非 **hidden**），就需要先滚动到该元素，然后进行 **click** 操作;为啥使用滚动？ 因为如果页面没有完全显示，**element** 如果是在下拉之后才能显示出来，只能先滚动到该元素才能进行 **click**，否则是不能 **click** 操作

1	JavaScriptExecutor js=(JavaScriptExecutor)driver;
2	// roll down and keep the element to the center of browser
3	js.executeScript("arguments[0].scrollIntoViewIfNeeded(true);", download);

- 3.不同方式进行定位，与 **expectedConditions** 判断方法封装，循环判断页面元素出现后再操作；
- 4.开发人员规范开发习惯，如给页面元素加上唯一的 **name,id** 等。

5. 如何去定位页面上动态加载的元素？

 触发动态事件，然后 **findElemnt**
 如果是动态菜单，需要一级一级 **find**（JS 实现）

6.如何去定位属性动态变化的元素？

 属性动态变化是指该 **element** 没有固定的属性值，所以只能通过相对位置定位
 比如通过 **xpath** 的轴， **parent / following—sibling / precent—sibling** 等
 另外也可以尝试 **findbyelements** 遍历

7.点击链接以后，**selenium** 是否会自动等待该页面加载完毕？

不会的。所以有的时候，当 **selenium** 并未加载完一个页面时再请求页面资源，则会误报不存在此元素。所以首先我们应该考虑判断，**selenium** 是否加载完此页面。其次再通过函数查找该元素。

8.自动化测试的时候是否需要连接数据库做数据校验？

一般来说 1、 **UI** 自动化不需要（很少需要）； 2、接口测试会需要：从数据库层面来进行数据校验可以更方便验证系统的数据处理方面是否正确；

9.有几种元素常用定位方式，分别是？你最偏爱哪一种，为什么？

8种：**id、name、class name、tag name、link text、partial link text、xpath、css selector** 偏爱哪一种？答：

我最常用的是 **xpath**（或 **CssSelector**）因为很多情况下，**html** 标签的属性不够规范，无法通过单一的属性定位，这个时候就只能使用 **xpath** 可以去重实现定位唯一 **element**
事实上定位最快的是 **Id**，因为 **id** 是唯一的，然而大多数开发并没有设置 **id**。

10.怎么提高 **selenium** 脚本的自动化执行效率？

- 1.优化测试用例，尽可不使用 **sleep**，减少使用 **ImplicitlyWait**
- 2.多使用 **selenium** 的 **WebDriverWait / FluentWait**，这样可以优化等待时间
- 3.减少不必要的操作步骤，如经过三四步才能打开我们要测试的页面的话，我们就可以直接通过网址来打开，减少不必要的操作。
- 4.中断页面加载，如果页面加载的内容过多，我们可以查看一下加载慢的原因，如果加载的内容不影响我们测试，就设置超时时间，中断页面加载。
- 5.使用性能好的电脑
- 11.用例在运行过程中经常会出现不稳定的情况，也就是这次可以通过，下次无法通过了，如何提高用例的稳定性？

- 1，查找元素前先做判断：**ExpectedConditions** 里面的各种方法；
- 2，显式等待：多使用 **WebDriverWait**，加上显式等待时间，等要操作的元素出现之后再执行下面的操作；
- 3、多用 **try catch** 捕获异常；
- 4，多线程的时候，减少测试用例耦合度，因为多线程的执行顺序是不受控制的；
- 5，尽量使用测试专用环境，避免其他类型的测试同时进行，对数据造成干扰。

12.你的自动化用例的执行策略是什么？

1.自动化测试用例是用来监控的。集成到 jenkins，创建定时任务定时执行；

2.有些用例在产品上线前必须回归。jenkins 上将任务绑定到开发的 build 任务上，触发执行；

3.有些用例不需要经常执行。jenkins 创建一个任务，需要执行的时候人工构建即可。

13.什么是持续集成？

频繁的将代码集成到主干，持续性的进行项目的构架，以便能够快速发现错误，防止分支大幅度偏离主干

14.webdriver client 的原理是什么？

在 selenium 启动以后，driver 充当了服务器的角色，跟 client 和浏览器通信，client 根据 webdriver 协议发送请求给 driver。driver 解析请求，并在浏览器上执行相应的操作，并把执行结果返回给 client.

15.webdriver 的协议是什么？The Wire Protocol

16.启动浏览器的时候用到的是哪个 webdriver 协议？http

17. 什么 PO 模式，什么是 page factory？

PO 模式是 page object model 的缩写，顾名思义，是一种设计模式，把每个页面当成一个页面对象，页面层写定位元素方法和页面操作方法,实现脚本的 page 和真实的网站页面 Map 起来，一一对应起来。这样能测试框架更容易维护。 比如一个登陆页面，使用 PO 模式后，会创建一个 LoginPage 的 class，该 class 会定义用户名输入框，密码输入框，登陆按钮的 webElement；用例层从页面层调用操作方法，写成用例，这种模式可以做到定位元素与脚本分离。所以这样的设计理念就是 PO 模式。 而 PageFactory 隶属 PO 模式，是用来初始化每个 PO 模式实现的 Page Class，初始化对象库。

18.怎样去选择一个下拉框中的 value=xx 的 option？

1.select 类里面提供的方法：selectByValue (“xxx”)

2.xpath 的语法也可以定位到

19.如何在定位元素后高亮元素？重置元素属性，给定位的元素加背景、边框

20. 如何设计高质量自动化脚本

1. 使用四层结构实现业务逻辑、脚本、数据分离。

2. 使用 PO 设计模式，将一个页面用到的元素和操作步骤封装在一个页面类中。如果一个元素定位发生了改变，我们只用修改这个页面的元素属性。

3. 对于页面类的方法，我们尽量从客户的正向逻辑去分析，方法中是一个独立场景，例如：登录到退出，而且不要想着把所有的步骤都封装在一个方法中。

4. 测试用例设计中，减少测试用例之间的耦合度。

21.get 和 post 的区别？

1、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。

POST 请求：POST 请求会把请求的数据放置在 HTTP 请求包的包体中。

2、传输数据的大小

使用 GET 请求时，传输数据会受到 URL 长度的限制。

对于 POST，理论上是不会受限制的

3、安全性。POST 的安全性比 GET 的高

先到此为止，后续接着总结.....

小新人~

[测试基础](#)

[测试方法](#)

按测试设计

黑盒

定义

已知产品所具有的功能，通过测试来检查每个功能是否正确，而完全不考虑内部的结构、特性

也称为：功能测试 / 数据驱动测试

软件验证 Verification 确定软件实现了这个功能

★方法

等价划分法

边界值法

因果图

正交分析法

错误推测法

白盒

方法

知道产品内部的工作过程，可通过测试来检测产品内部工作是否按照规格说明书正常进行

按照程序内部的结构测试程序，检测程序中每条同路是否能按照预定要求正确工作。

也称为：结构测试 / 逻辑驱动测试

代码检查法

静态结构分析法

静态质量度量法

逻辑覆盖 (6种方法)

发现错误的能力 弱——>强

1. 语句覆盖 每条语句至少执行一次

2. 判定覆盖 / 分支覆盖 每一个判定 (分支) 至少获得一次真值、假值

3. 条件覆盖 一个判定语句是由多个条件组合而成的复合判定

每个判定的每个条件取到各种可能的值

4. 判定条件覆盖

5. 条件组合覆盖 使得每个判定中条件的各种可能组合都至少出现一次

6. 路径覆盖 每一条路径执行一次

覆盖标准

基本路径测试

循环覆盖

确认 (Validation)

确保功能正确的实现

灰盒

多用于：集成测试阶段

不仅关注输入 / 输出的正确性、也关注程序内部的情况

常常是通过一些表征性现象、事件、标志来判断内部的运行状态

单元测试 对最小可测单元进行检查验证 例如：类 / 函数

功能测试 / 接口测试 根据输入、输出验证模块功能

集成测试 验证几个相互有依赖关系的模块组合起来的功能

场景测试 验证几个模块是否能完成一个用户的场景

测试分类

按测试目的

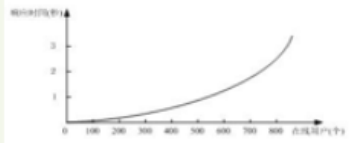
1. 功能测试

- 场景测试 - 验证几个模块是否能完成一个用户的场景
- 系统测试 - 对整个系统的测试
- Alpha测试 - 软件测试人员
在真实用户环境中对软件进行全面测试
- Beta测试 (公测) - 真实用户
在真实的用户环境中的测试

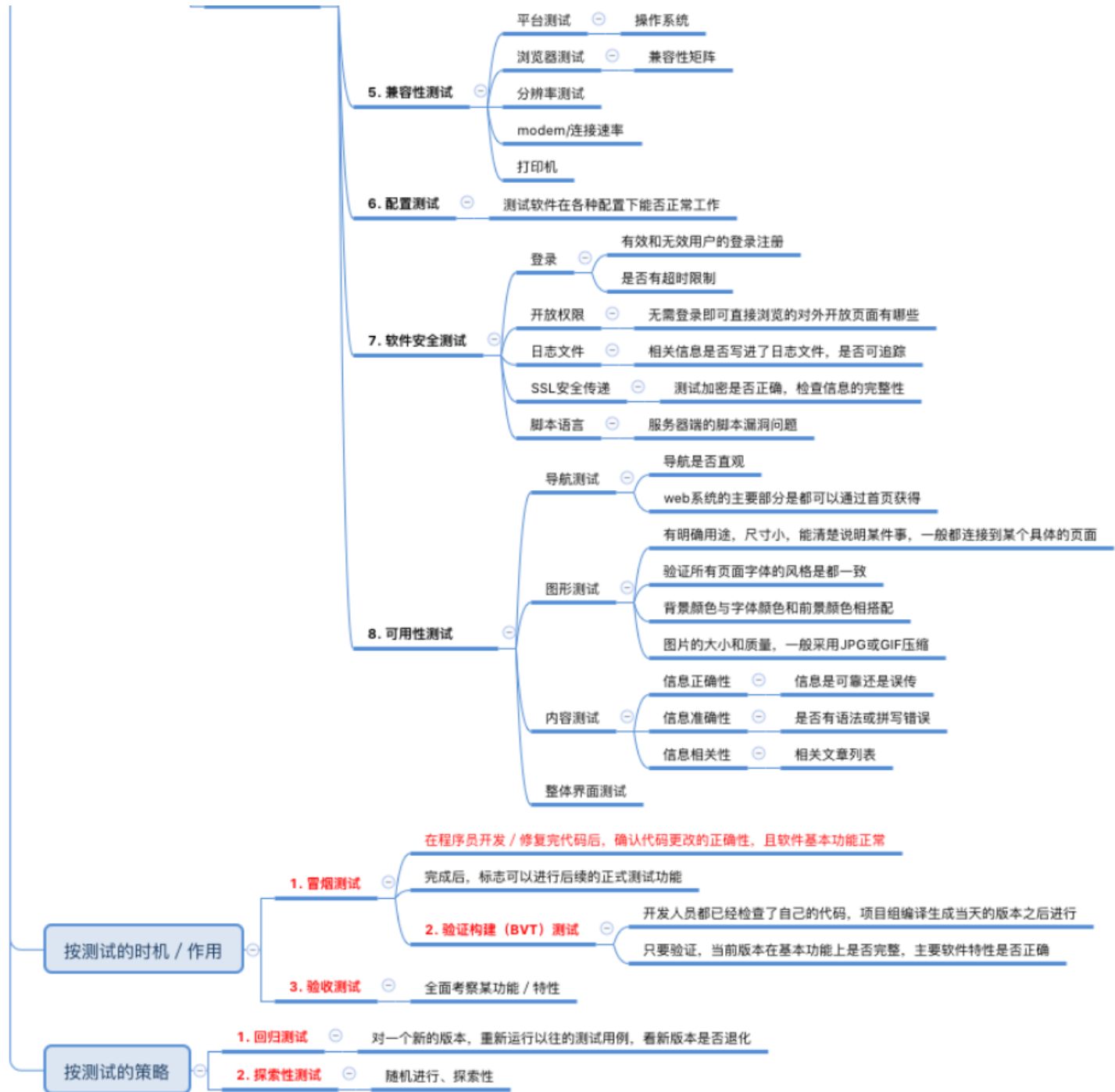
2. 非功能测试

- 1. 性能测试 - 主要评价系统 / 组件的性能是否和具体的性能需求一致

页面加载时间	愿意等待的用户比例
10秒	84%
15秒	51%
20秒	25%
30秒	5%

例如：对访问速度的性能需求和对内存使用情况的需求
- 2. 负载测试 - 通过增加负载来评估组件或系统的性能


例如：通过增加并发用户量 / 事务数量来测量组件 / 系统的性能
- VS 性能测试 - 进行负载测试时，系统负载是逐渐增减，观察在各种负载情况下系统的性能
- 3. 压力测试 - 评估系统处于、超过预期负载时的处理能力 - 不崩溃
例如：
 - 在内存耗尽时，系统运行情况，也不能崩溃
 - 原最大支持并发量为1000，现测试2000-3000，系统也应该不崩溃
- 4. 强度测试 - 检查程序对异常情况的抵抗能力 - 鲁棒性
是检查系统在极限状态下运行的时候性能下降的幅度是否在允许的范围内。
例如：运行可能导致虚拟机或者磁盘崩溃的测试用例
- 疲劳强度测试 - 测试系统在长时间运行后的性能测试表现
例如：7*24小时的压力测试
- 5. 兼容性测试 - 平台测试 - 操作系统
浏览器测试 - 兼容性矩阵
分辨率测试
modem/连接速率



参考：<https://www.cnblogs.com/lesleysbw/p/6400574.html>

重点掌握黑盒（等价类划分、边界值法）、白盒（覆盖方法）。

1、简述什么是静态测试、动态测试、黑盒测试、白盒测试、 α 测试 β 测试

- 静态测试是不运行程序本身而寻找程序代码中可能存在的错误或评估程序代码的过程。
- 动态测试是实际运行被测程序，输入相应的测试实例，检查运行结果与预期结果的差异，判定执行结果是否符合要求，从而检验程序的正确性、可靠性和有效性，并分析系统运行效率和健壮性等性能。
- 黑盒测试一般用来确认软件功能的正确性和可操作性,目的是检测软件的各个功能是否能得以实现,把被测测试的程序当作一个黑盒,不考虑其内部结构,在知道该程序的输入和输出之间的关系或程序功能的情况下,依靠软件规格说明书来确定测试用例和推断测试结果的正确性。
- 白盒测试根据软件内部的逻辑结构分析来进行测试,是基于代码的测试，测试人员通过阅读程序代码或者通过使用开发工具中的单步调试来判断软件的质量，一般黑盒测试由项目经理在程序员开发中来实现。
- α 测试是由一个用户在开发环境下进行的测试，也可以是公司内部的用户在模拟实际操作环境下进行的受控测试，Alpha 测试不能由程序员或测试员完成。
- β 测试是软件的多个用户在一个或多个用户的实际使用环境下进行的测试。开发者通常不在测试现场，Beta 测试不能由程序员或测试员完成。

2、黑盒测试的测试用例常见设计方法都有哪些？请分别以具体的例子来说明这些方法在测试用例设计工作中的应用。

1）等价类划分： 等价类是指某个输入域的子集合.在该子集中,各个输入数据对于揭露程序中的错误都是等效的.并合理地假定:测试某等价类的代表值就等于对这一类其它值的测试.因此,可以把全部输入数据合理划分为若干等价类,在每一个等价类中取一个数据作为测试的输入条件,就可以用少量代表性的测试数据,取得较好的测试结果.等价类划分可有两种不同的情况:有效等价类和无效等价类。

2）边界值分析法：是对等价类划分方法的补充。测试工作经验告诉我,大量的错误是发生在输入或输出范围的边界上,而不是发生在输入输出范围的内部.因此针对各种边界情况设计测试用例,可以查出更多的错误。

使用边界值分析方法设计测试用例,首先应确定边界情况.通常输入和输出等价类的边界,就是应着重测试的边界情况.应当选取正好等于,刚刚大于或刚刚小于边界的值作为测试数据,而不是选取等价类中的典型值或任意值作为测试数据。

3）错误猜测法：基于经验和直觉推测程序中所有可能存在的各种错误，从而有针对性的设计测试用例的方法。

错误推测方法的基本思想：列举出程序中所有可能有的错误和容易发生错误的特殊情况,根据他们选择测试用例。例如，在单元测试时曾列出的许多在模块中常见的错误。以前产品测试中曾经发现的错误等，这些就是经验的总结。还有，输入数据和输出数据为 0 的情况。输入表格为空格或输入表格只有一行。这些都是容易发生错误的情况。可选择这些情况下的例子作为测试用例。

4）因果图方法：前面介绍的等价类划分方法和边界值分析方法,都是着重考虑输入条件,但未考虑输入条件之间的联系,相互组合等。考虑输入条件之间的相互组合,可能会产生一些新的情况。但要检查输入条件的组合不是一件容易的事情,即使把所有输入条件划分成等价类,他们之间的组合情况也相当多。因此必须考虑采用一种适合于描述对于多种条件的组合,相应产生多个动作的形式来考虑设计测试用例。这就需要利用因果图（逻辑模型）。因果图方法最终生成的就是判定表。它适合于检查程序输入条件的各种组合情况。

5）正交表分析法：可能因为大量的参数的组合而引起测试用例数量上的激增，同时，这些测试用例并没有明显的优先级上的差距，而测试人员又无法完成这么多数量的测试，就可以通过正交表来进行缩减一些用例，从而达到尽量少的用例覆盖尽量大的范围的可能性。

6）场景分析方法：指根据用户场景来模拟用户的操作步骤，这个比较类似因果图，但是可能执行的深度和可行性更好。

7）状态图法：通过输入条件和系统需求说明得到被测系统的所有状态，通过输入条件和状态得出输出条件；通过输入条件、输出条件和状态得出被测系统的测试用例。

8）大纲法：大纲法是一种着眼于需求的方法，为了列出各种测试条件，就将需求转换为大纲的形式。大纲表示为树状结构，在根和每个叶子结点之间存在唯一的路径。大纲中的每条路径定义了一个特定的输入条件集合，用于定义测试用例。树中叶子的数目或大纲中的路径给出了测试所有功能所需测试用例的大致数量。

3、软件测试分为几个阶段 各阶段的测试策略和要求是什么？

和开发过程相对应，测试过程会依次经历单元测试、集成测试、系统测试、验收测试四个主要阶段：

- 单元测试：单元测试是针对软件设计的最小单位——程序模块甚至代码段进行正确性检验的测试工作，通常由开发人员进行。
- 集成测试：集成测试是将模块按照设计要求组装起来进行测试，主要目的是发现与接口有关的问题。由于在产品提交到测试部门前，产品开发小组都要进行联合调试，因此在大部分企业中集成测试是由开发人员来完成的。

- **系统测试：**系统测试是在集成测试通过后进行的，目的是充分运行系统，验证各子系统是否都能正常工作并完成设计的要求。它主要由测试部门进行，是测试部门最大最重要的一个测试，对产品的质量有重大的影响。
- **验收测试：**验收测试以需求阶段的《需求规格说明书》为验收标准，测试时要求模拟实际用户的运行环境。对于实际项目可以和客户共同进行，对于产品来说就是最后一次的系统测试。测试内容为对功能模块的全面测试，尤其要进行文档测试。

单元测试测试策略：

自顶向下的单元测试策略：比孤立单元测试的成本高很多，不是单元测试的一个好的选择。

自底向上的单元测试策略：比较合理的单元测试策略，但测试周期较长。

孤立单元测试策略：最好的单元测试策略。

集成测试的测试策略：

大爆炸集成：适应于一个维护型项目或被测试系统较小

自顶向下集成：适应于产品控制结构比较清晰和稳定；高层接口变化较小；底层接口未定义或经常可能被修改；产口控制组件具有较大的技术风险，需要尽早被验证；希望尽早能看到产品的系统功能行为。

自底向上集成：适应于底层接口比较稳定；高层接口变化比较频繁；底层组件较早被完成。

基于进度的集成

优点：具有较高的并行度；能够有效缩短项目的开发进度。

缺点：桩和驱动工作量较大；有些接口测试不充分；有些测试重复和浪费。

系统测试的测试策略：

数据和数据库完整性测试；功能测试；用户界面测试；性能评测；负载测试；强度测试；容量测试；安全性和访问控制测试；故障转移和恢复测试；配置测试；安装测试；加密测试；可用性测试；版本验证测试；文档测试

问题 17：

有哪些指标是可以用来评估 QA 测试质量的？

在软件开发中，软件质量是衡量软件是否符合需求、标准的重要体现。除了[代码质量](#)外，影响软件整体质量的因素还有很多。因此，要确保软件的整体质量，就需要在各个环节严格控制。

本文列出了衡量软件质量的 5 个最常用的指标。

1. SLOC（Source Lines of Code，源代码行）

计算代码行数可能是最简单的衡量指标，主要体现了软件的规模，并为项目增长和规划提供了相关数据。例如，如果每月统计一次代码的行数，就可以绘制一个项目发展概览图。当然，由于存在项目重构或是设计阶段等因素，这种方式并不太可靠，但是可以为项目的发展提供一个视角。

可以只统计逻辑代码行（Source Logical Line of Code，SLLOC），这样可以获得稍准确的信息。逻辑代码行不包含空行、单个括号行和注释行。可以使用 [Metrics](#) 工具来统计。

代码行数不应该用来评估开发者的效率，否则，可能会产生重复、不可维护的或不专业的代码。

2. 每个代码段/模块/时间段中的 bug 数

要想实现更好的测试以及更高的可维护性，bug 跟踪是必不可少的。每个代码段、模块或时间段（天、周、月等）内的 bug 可以很容易通过工具统计出来（如 [Mantis](#)）。这样，可以及早发现并及时修复。

Bug 数可以作为评估开发者效率的指标之一，但必须注意，如果过分强调这种评估方法，软件开发者和测试者可能会成为敌人。在生产企业中，要保证员工彼此之间的凝聚力。

为了更好的实现评估，可以根据重要性和解决成本将 bug 划分为低、中、高三个级别。

3. 代码覆盖率

在单元测试阶段，代码覆盖率常常被拿来作为衡量测试好坏的指标，也用来考核测试任务完成情况。可以使用的工具也有很多，如 [Cobertura](#) 等。

代码覆盖率并不能代表单元测试的整体质量，但可以提供一些测试覆盖率相关的信息，可以和其他一些测试指标一起来使用。

此外，在查看代码覆盖率时，还需注意单元测试代码、集成测试场景和结果等。

4. 设计/开发约束

软件开发中有很多设计约束和原则，其中包括：

- 类/方法的长度
- 一个类中方法/属性的个数
- 方法/构造函数参数的个数
- 代码文件中魔术数字、字符串的使用（魔术数字指直接写在代码中的具体数值，其他人难以理解数字的意义）
- 注释行比例等

代码的可维护性和可读性是很重要的，开发团队可以选择以上这些原则中的一个或全部，并通过一些自动化工具（如 [maven pmd 插件](#)）来遵循这些原则，这将大大提高软件产品的质量。

5. 圈复杂度（Cyclomatic Complexity）

圈复杂度是用来衡量一个模块判定结构的复杂程度，已经成为评估软件质量的一个重要标准，能帮助开发者识别难于测试和维护的模块，在成本、进度和性能之间寻求平衡。圈复杂度可以使用 [pmd](#) 工具来自动化计算。

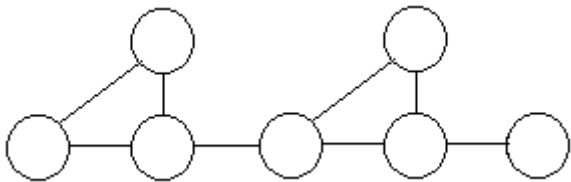
圈复杂度数量上表现为独立路径的条数，即合理的预防错误所需测试的最少路径条数，圈复杂度大说明程序代码可能质量低且难于测试和维护。

计算公式为： $V(G) = E - N + 2P$

E：边，代表节点间的程序流；

N：节点，程序中代码的最小单元

P：出口节点



上图中共 8 条边，7 个节点，因此圈复杂度为 $8 - 7 + 2 \times 1 = 3$ 。这意味着，理论上需要编写 3 个测试用例来覆盖所有的判定条件。

其实，圈复杂度的计算还有更直观的方法，因为圈复杂度所反映的是“判定条件”的数量，所以圈复杂度实际上就是等于判定节点的数量再加上 1，也即控制流图的区域数，对应的计算公式为： $V(G) = \text{区域数} = \text{判定节点数} + 1$ 。

在项目开发中，可以根据项目类型，来定义上限数（6、8 或 10 等）。

以上是最常用的 5 种软件质量度量指标，当然，还可以结合其他的指标，对项目有一个更清晰的认识。

英文原文：[5 Common Automated Software Quality Metrics](#)

分类：[项目管理](#)

问题 18：

ABC 系统是一个无缝集成的系统，包括课程，进度监控工具、协作工具、报告、con 开发工具和管理工具。用户可以访问每个用户类型的各种产品/模块，并授予一个 loc 环境的权限，应用程序必须满足大学的要求，以保护数据。所有组件必须可使用 si 访问，并且必须与 windows 或者移动(IOS,Ar)兼容。

从以上介绍，请回答一下问题：

1) 这个系统有哪几大块功能吗？

- 2) 用户的权限控制是什么？
3) 这个大学对宿主环境和应用程序的要求是什么？

问题 19:

使用 Flask 蓝图编写一个接口，要有权限判断。

近打算和一位算法大神做一个自然语言处理的 AI 项目

由于算法最好是使用 Python 语言来实现，故业务逻辑不打算使用稳如老狗的 JavaEE，采用轻量级 Python Web 框架 Flask（Django 含太多无关模块，略显笨重）

Flask 项目也是有一定的架构，主要指蓝图

网上很多文章把简单的蓝图复杂化，

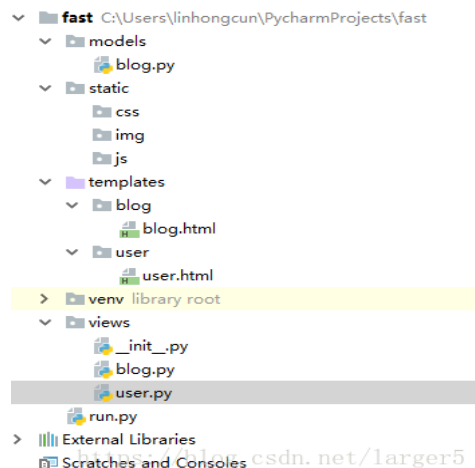
或是缺少代码文件项目截图，或是代码不全，或是代码没有文件名，或是没留源码，或是源码链接失效，

在此澄清小一下思路

二、代码

具体代码发布到 github 上：https://github.com/larger5/python_Flask_lantu.git

架构采用了 Flask 官方文档 蓝图 的 demo 的风格，应该会恰当些



1.入口文件 run.py

```
from flask import Flask, render_template
from views.blog import blog
from views.user import user
app = Flask(__name__)
app.register_blueprint(blog, url_prefix='/blog')
app.register_blueprint(user, url_prefix='/user')
@app.route('/')
def hello_world():
```



```
        return 'Hello World!'
if __name__ == '__main__':
    app.run()
```

2. 蓝图一 views/blog.py

```
from flask import Blueprint, render_template
blog = Blueprint('blog', __name__, template_folder='../templates/blog')
@blog.route('/index/')
def index():
    return render_template('blog.html')
@blog.route('/welcome/')
def welcome():
    return "welcome to blog"
```

3. 蓝图二 views/user.py

```
from flask import Blueprint, render_template
user = Blueprint('user', __name__, template_folder='../templates/user')
@user.route('/index/')
def index():
    return render_template('user.html')
@user.route('/welcome/')
def welcome():
    return "welcome to user"
```

4. blog 模版 blog.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>blog</title>
</head>
<body>
<h1>blogs</h1>
</body>
</html>
```

5. user 模版 user.html

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>user</title>
</head>
<body>
<h1>user</h1>
</body>
</html>
```

采用了 2 个蓝图，方便看出规律

其他文件皆为空文件，就不一一列举了

三、测试



四、其他

代码比较简单，不做过多描述，主要是架构要弄清

问题 20:

用 SQLAlchemy 改写下面 SQL

```
sql = '''SELECT a.id FROM a
        INNER JOIN b ON a.id = b.a_id
        WHERE (SELECT COUNT(1) AS num FROM d
               WHERE d.id = a.id AND d.user_id=:user_id) = 0
        AND a.year>2010 limit 0,:group_size ;'''
```

```
parms = {
    'user_id':user_id,
    'knowPoint_id':knowPoint_id,
    'group_size':group_size*2
}
```

```
list = dbconn.get_sql_session_all_data(sql,parms)
```

问题 21:

简述熟练掌握的机器学习算法。

简要介绍一下机器学习中的经典代表方法。重点是这些方法内涵的思想。

01 回归算法

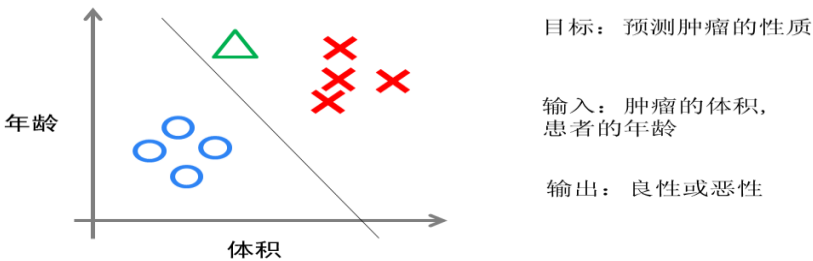
在大部分机器学习课程中，回归算法都是介绍的第一个算法。原因有两个：一回归算法比较简单，介绍它可以让人平滑地从统计学迁移到机器学习中。二回归算法是后面若干强大算法的基石，如果不理解回归算法，无法学习那些强大的算法。回归算法有两个重要的子类：即线性回归和逻辑回归。

线性回归就是我们前面说过的房价求解问题。如何拟合出一条直线最佳匹配我所有的数据？一般使用“最小二乘法”来求解。“最小二乘法”的思想是这样的，假设我们拟合出的直线代表数据的真实值，而观测到的数据代表拥有误差的值。为了尽可能减小误差的影响，需要求解一条直线使所有误差的平方和最小。最小二乘法将最优问题转化为求函数极值问题。函数极值在数学上我们一般会采用求导数为 0 的方法。但这种做法并不适合计算机，可能求解不出来，也可能计算量太大。

计算机科学界专门有一个学科叫“数值计算”，专门用来提升计算机进行各类计算时的准确性和效率问题。例如，著名的“梯度下降”以及“牛顿法”就是数值计算中的经典算法，也非常适合来处理求解函数极值的问题。梯度下降法是解决回归模型中最简单且有效的方法之一。从严格意义上来说，由于后文中的神经网络和推荐算法中都有线性回归的因子，因此梯度下降法在后面的算法实现中也有应用。

逻辑回归是一种与线性回归非常类似的算法，但是，从本质上讲，线型回归处理的问题类型与逻辑回归不一致。线性回归处理的是数值问题，也就是最后预测出的结果是数字，例如房价。而逻辑回归属于分类算法，也就是说，逻辑回归预测结果是离散的分类，例如判断这封邮件是否是垃圾邮件，以及用户是否会点击此广告等等。

实现方面的话，逻辑回归只是对对线性回归的计算结果加上了一个 Sigmoid 函数，将数值结果转化为了 0 到 1 之间的概率(Sigmoid 函数的图像一般来说并不直观，你只需要理解对数值越大，函数越逼近 1，数值越小，函数越逼近 0)，接着我们根据这个概率可以做预测，例如概率大于 0.5，则这封邮件就是垃圾邮件，或者肿瘤是否是恶性的等等。从直观上来说，逻辑回归是画出了一条分类线，见下图。



逻辑回归的直观解释

假设我们有一组肿瘤患者的数据，这些患者的肿瘤中有些是良性的(图中的蓝色点)，有些是恶性的(图中的红色点)。这里肿瘤的红蓝色可以被称作数据的“标签”。同时每个数据包括两个“特征”：患者的年龄与肿瘤的大小。我们将这两个特征与标签映射到这个二维空间上，形成了我上图的数据。

当我有一个绿色的点时，我该判断这个肿瘤是恶性的还是良性的呢？根据红蓝点我们训练出了一个逻辑回归模型，也就是图中的分类线。这时，根据绿点出现在分类线的左侧，因此我们判断它的标签应该是红色，也就是说属于恶性肿瘤。逻辑回归算法划出的分类线基本都是线性的(也有划出非线性分类线的逻辑回归，不过那样的模型在处理数据量较大的时候效率会很低)，这意味着当两类之间的界线不是线性时，逻辑回归的表达能力就不足。下面的两个算法是机器学习界最强大且重要的算法，都可以拟合出非线性的分类线。

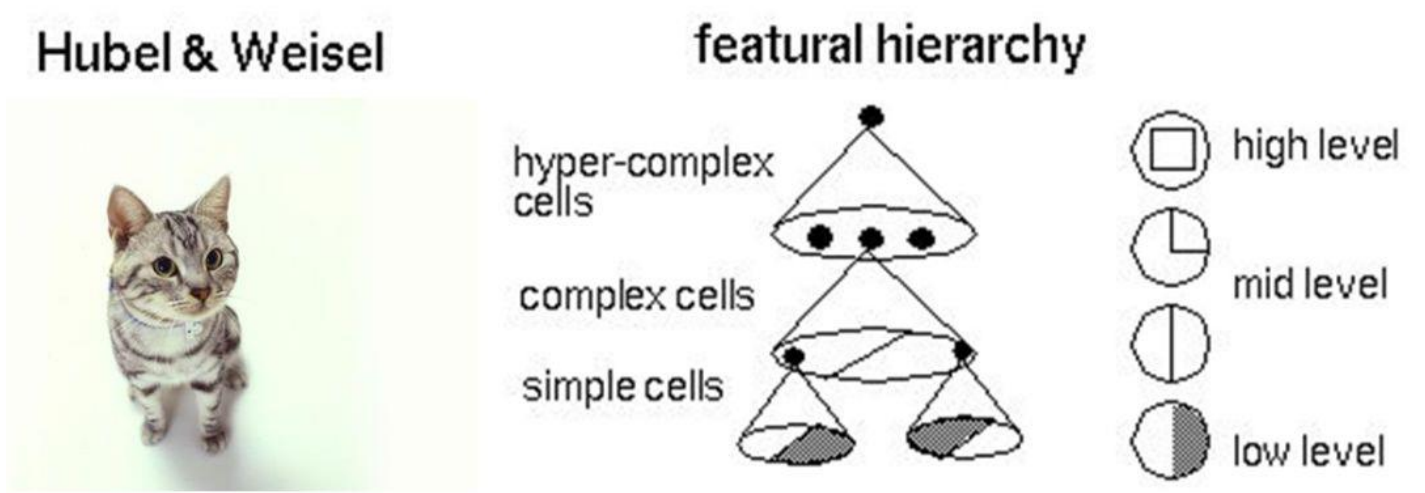
02 神经网络

神经网络(也称之为人工神经网络，ANN)算法是 80 年代机器学习界非常流行的算法，不过在 90 年代中途衰落。现在，携着“深度学习”之势，神经网络重装归来，重新成为最强大的机器学习算法之一。

神经网络的诞生起源于对大脑工作机理的研究。早期生物界学者们使用神经网络来模拟大脑。机器学习的学者们使用神经网络进行机器学习的实验，发现在视觉与语音的识别上效果都相当好。在 BP 算法(加速神经网络训练过程的数值算法)

诞生以后，神经网络的发展进入了一个热潮。BP 算法的发明人之一是前面介绍的机器学习大牛 Geoffrey Hinton(图 1 中的中间者)。

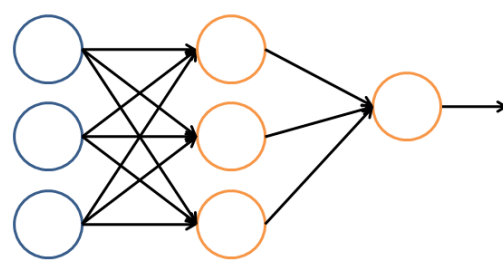
具体说来，神经网络的学习机理是什么？简单来说，就是分解与整合。在著名的 Hubel-Wiesel 试验中，学者们研究猫的视觉分析机理是这样的。



Hubel-Wiesel 试验与大脑视觉机理

比方说，一个正方形，分解为四个折线进入视觉处理的下一层中。四个神经元分别处理一个折线。每个折线再继续被分解为两条直线，每条直线再被分解为黑白两个面。于是，一个复杂的图像变成了大量的细节进入神经元，神经元处理以后再进行整合，最后得出了看到的是正方形的结论。这就是大脑视觉识别的机理，也是神经网络工作的机理。

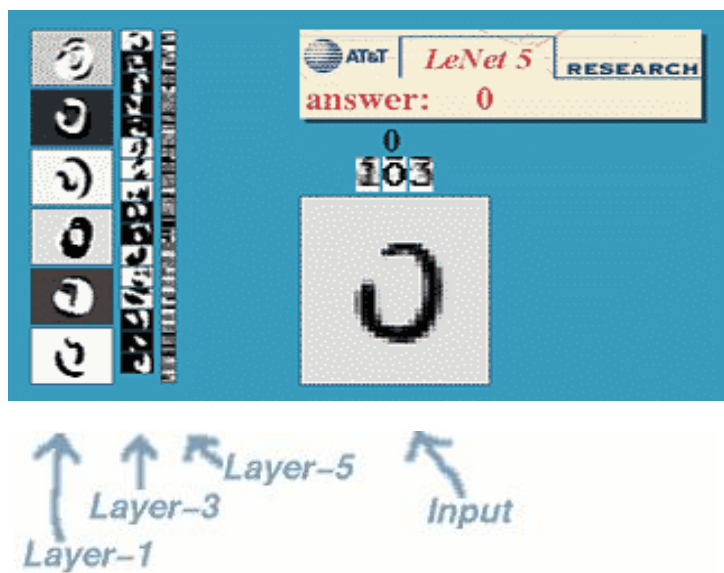
让我们看一个简单的神经网络的逻辑架构。在这个网络中，分成**输入层**，**隐藏层**，和**输出层**。输入层负责接收信号，隐藏层负责对数据的分解与处理，最后的结果被整合到输出层。每层中的一个圆代表一个处理单元，可以认为是模拟了一个神经元，若干个处理单元组成了一个层，若干个层再组成了一个网络，也就是"神经网络"。



神经网络的逻辑架构

在神经网络中，每个处理单元事实上就是一个逻辑回归模型，逻辑回归模型接收上层的输入，把模型的预测结果作为输出传输到下一个层次。通过这样的过程，**神经网络可以完成非常复杂的非线性分类**。

下图会演示神经网络在图像识别领域的一个著名应用，这个程序叫做 LeNet，是一个基于多个隐层构建的神经网络。通过 LeNet 可以识别多种手写数字，并且达到很高的识别精度与拥有较好的鲁棒性。



LeNet 的效果展示

右下方的方形中显示的是输入计算机的图像，方形上方的红色字样“answer”后面显示的是计算机的输出。左边的三条竖直的图像列显示的是神经网络中三个隐藏层的输出，可以看出，随着层次的不深入，越深的层次处理的细节越低，例如层 3 基本处理的都已经是线的细节了。LeNet 的发明人就是前文介绍过的机器学习的大牛 Yann LeCun(图 1 右者)。

进入 90 年代，神经网络的发展进入了一个瓶颈期。其主要原因是尽管有 BP 算法的加速，神经网络的训练过程仍然很困难。因此 90 年代后期支持向量机(SVM)算法取代了神经网络的地位。

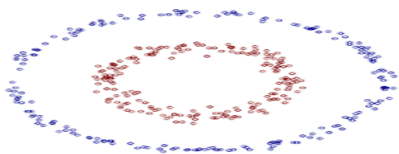
03 SVM (支持向量机)

支持向量机算法是诞生于统计学习界，同时在机器学习界大放光彩的经典算法。

支持向量机算法从某种意义上来说是逻辑回归算法的强化：通过给予逻辑回归算法更严格的优化条件，支持向量机算法可以获得比逻辑回归更好的分类界线。但是如果没有某类函数技术，则支持向量机算法最多算是一种更好的线性分类技术。

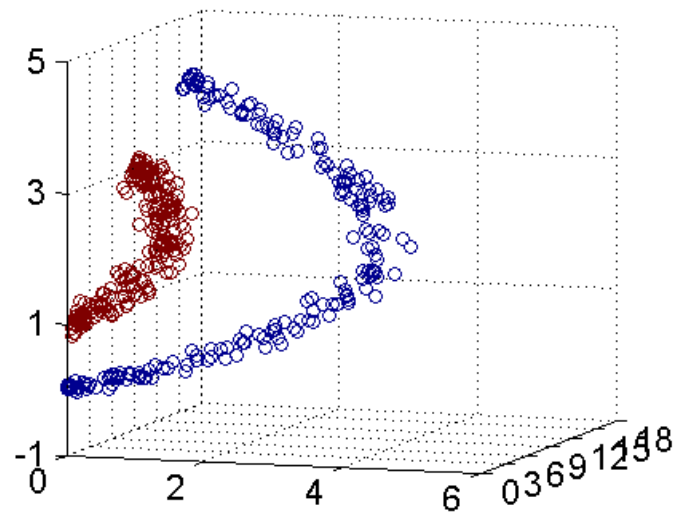
但是，通过跟高斯“核”的结合，支持向量机可以表达出非常复杂的分类界线，从而达成很好的的分类效果。“核”事实上就是一种特殊的函数，最典型的特征就是可以将低维的空间映射到高维的空间。

例如下图所示：



支持向量机图例

我们如何在二维平面划分出一个圆形的分类界线？在二维平面可能会很困难，但是通过“核”可以将二维空间映射到三维空间，然后使用一个线性平面就可以达成类似效果。也就是说，二维平面划分出的非线性分类界线可以等价于三维平面的线性分类界线。于是，我们可以通过在三维空间中进行简单的线性划分就可以达到在二维平面中的非线性划分效果。



三维空间的切割

支持向量机是一种数学成分很浓的机器学习算法（相对的，神经网络则有生物科学成分）。在算法的核心步骤中，有一步证明，即将数据从低维映射到高维不会带来最后计算复杂性的提升。于是，通过支持向量机算法，既可以保持计算效率，又可以获得非常好的分类效果。因此支持向量机在 90 年代后期一直占据着机器学习中最核心的地位，基本取代了神经网络算法。直到现在神经网络借着深度学习重新兴起，两者之间才又发生了微妙的平衡转变。

04 聚类算法

前面的算法中的一个显著特征就是我的训练数据中包含了标签，训练出的模型可以对其他未知数据预测标签。在下面的算法中，训练数据都是不含标签的，而算法的目的则是通过训练，推测出这些数据的标签。这类算法有一个统称，即无监督算法(前面有标签的数据的算法则是有监督算法)。无监督算法中最典型的代表就是聚类算法。

让我们还是拿一个二维的数据来说，某一个数据包含两个特征。我希望通过聚类算法，给他们中不同的种类打上标签，我该怎么做呢？简单来说，聚类算法就是计算种群中的距离，根据距离的远近将数据划分为多个族群。

聚类算法中最典型的代表就是 K-Means 算法。

05 降维算法

降维算法也是一种无监督学习算法，其主要特征是将数据从高维降低到低维层次。在这里，维度其实表示的是数据的特征量的大小，例如，房价包含房子的长、宽、面积与房间数量四个特征，也就是维度为 4 维的数据。可以看出来，长与宽事实上与面积表示的信息重叠了，例如面积=长 × 宽。通过降维算法我们就可以去除冗余信息，将特征减少为面积与房间数量两个特征，即从 4 维的数据压缩到 2 维。于是我们将数据从高维降低到低维，不仅利于表示，同时在计算上也能带来加速。

刚才说的降维过程中减少的维度属于肉眼可视的层次，同时压缩也不会带来信息的损失(因为信息冗余了)。如果肉眼不可视，或者没有冗余的特征，降维算法也能工作，不过这样会带来一些信息的损失。但是，降维算法可以从数学上证明，从高维压缩到的低维中最大程度地保留了数据的信息。因此，使用降维算法仍然有很多的好处。

降维算法的主要作用是压缩数据与提升机器学习其他算法的效率。通过降维算法，可以将具有几千个特征的数据压缩至若干个特征。另外，降维算法的另一个好处是数据的可视化，例如将 5 维的数据压缩至 2 维，然后可以用二维平面来可视。降维算法的主要代表是 PCA 算法(即主成分分析算法)。

06 推荐算法

推荐算法是目前业界非常火的一种算法，在电商界，如亚马逊，天猫，京东等得到了广泛的运用。推荐算法的主要特征就是可以自动向用户推荐他们最感兴趣的东西，从而增加购买率，提升效益。推荐算法有两个主要的类别：

一类是基于物品内容的推荐，是将与用户购买的内容近似的物品推荐给用户，这样的前提是每个物品都得有若干个标签，因此才可以找出与用户购买物品类似的物品，这样推荐的好处是关联程度较大，但是由于每个物品都需要贴标签，因此工作量较大。

另一类是基于用户相似度的推荐，则是将与目标用户兴趣相同的其他用户购买的东西推荐给目标用户，例如小 A 历史上买了物品 B 和 C，经过算法分析，发现另一个与小 A 近似的用户小 D 购买了物品 E，于是将物品 E 推荐给小 A。

两类推荐都有各自的优缺点，在一般的电商应用中，一般是两类混合使用。推荐算法中最有名的算法就是协同过滤算法。

07 其他

除了以上算法之外，机器学习界还有其他的如高斯判别，朴素贝叶斯，决策树等等算法。但是上面列的六个算法是使用最多，影响最广，种类最全的典型。机器学习界的一个特色就是算法众多，发展百花齐放。下面做一个总结，按照训练的数据有无标签，可以将上面算法分为监督学习算法和无监督学习算法，但推荐算法较为特殊，既不属于监督学习，也不属于非监督学习，是单独的一类。

监督学习算法：线性回归，逻辑回归，神经网络，SVM

无监督学习算法：聚类算法，降维算法

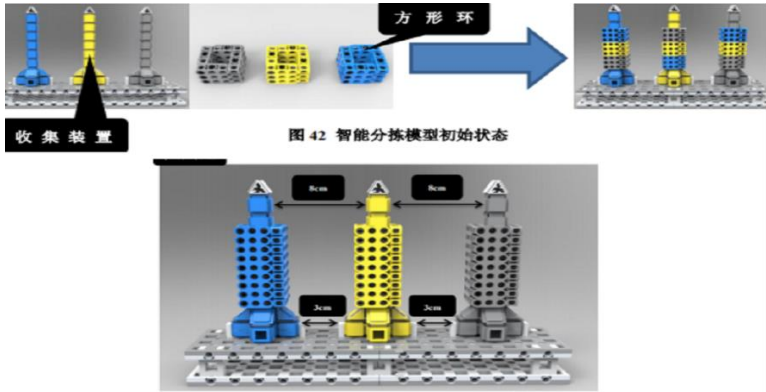
特殊算法：

推荐算法

除了这些算法以外，有一些算法的名字在机器学习领域中也经常出现。但他们本身并不算是一个机器学习算法，而是为了解决某个子问题而诞生的。你可以理解他们为以上算法的子算法，用于大幅度提高训练过程。其中的代表有：梯度下降法，主要运用在线型回归，逻辑回归，神经网络，推荐算法中；牛顿法，主要运用在线型回归中；BP 算法，主要运用在神经网络中；SMO 算法，主要运用在 SVM 中。

问题 22：

下图模型中，灰、黄、蓝三种颜色的收集装置各 1 个。灰、黄、蓝三种颜色的方形环各 3 个。方形环随机摆放在收集装置上。要求将方形环进行移动，通过最少次移动，使方形环颜色与收集装置颜色一致。每次可移动一个方形环。每个收集装置上面方形环的数量不超过 5 个。（竞赛题略）



问题 23：

说明：技术不限于 Django 或 Django REST Framework

1.使用 drfAPI 实现基于 JWT 登录的接口返回 TOKEN,并将 TOKEN 登录的有效期设置为 60 分钟。

创建超用户 admin;获取一次 admin 的 TOKEN

2.接口实现

用户模型 emp

结构:

UUID,主键

name 姓名, char 列, 不可为空

empno 号, BIGINT 列, 唯一, 索引, 最大长度 50, 不可为空, 工号存入时数值不可大于 50000

Group 组织用多对多关联到组织表

组织模型 org

ID id, 主键

name 名称, char 列, 唯一, 索引, 长度 50

type 枚举类型, 可选项:开发、 运维、 产品、 测试

根据上述模型, 实现返回如下数据格式, 并实现根据工号搜索、过滤、分页

{empno:'工号', name:'用户名'(Groupname: '组织名称') }

将接口注册到 api 首页

使用 POSTMAN,通过 admin 用户账号/密码请求这个 API,

使用 POSTMAN,通过 admin 用户的 TOKEN 请求这个 API

3.设计一个中间件或装饰, 记录 emp 接口访问日志{请求时间、请求方法、请求接口路径、请求 user、请求 IP、 请求参数}

4.设计一 个异步接口, 实现将上面接口添加用户功能, 传入的数据使用 Celery 异步的方式进行处理。使用 Redis 作为任务队列

环境说明:本机所有密码均为 123

数据库使用本地 sqllite/mysql

Redis 本机 REDIS

Python3 虚拟环境 workon aops 或者 自建

问题 24:

服务订购到期日

在真实的业务场景中, 客户可以在任意一天订购我们的服务, 订购周期可以是一个月、一个季度、半年或一年, 在订购日后的这么多月后的凌晨 0 点订单到期。

以下举例说明订购的过期计算方式:

在 2016-11-10 订购一个月, 订购将在 2016-12-10 过期

在 2016-12-10 订购一个月, 订购将在 2017-01-10 过期

在 2017-01-30 订购一个月, 订购将在 2017-02-28 过期

在 2017-05-31 订购三个月, 订购将在 2017-08-31 过期

问题 1 (无需编程)

假定我们每天可以收到 4 个订单, 其中一个是一个月周期的, 一个是一个季度周期的, 一个是半年周期的, 一个是一年周期的。这样的订单从很久以前, 比如五年前就开始了。

请通过分析回答: 在哪一天到期的订单会最多, 有多少个? 具体是哪几个订单?

并对分析过程进行解释。

问题 2

请实现一个方法，给定订购的年月日（year, month, day），已知订购时长是一个月，返回订单的到期日。year, month, day 均为整数，保证是正确的日期。

在考虑我们的喜好（Python == Javascript > Java > C/C++）的基础上，选择你最擅长的语言，给出「可在生产环境中放心使用」级别的代码。

Python 请实现以下方法：

```
def getExpirationDate(year, month, day):
```

```
    # TODO
```

```
    return [year, month, day]
```

Javascript 请实现以下方法：

```
function getExpirationDate(year, month, day) {
```

```
// TODO
```

```
    return [year, month, day];
```

```
}
```

其余语言请参考上述语言入参和返回值进行实现。

主体代码避免使用日期相关的系统库/第三方库。

Python 应避免使用 datetime 及同类库。

Javascript 应避免使用 Date 对象。

测试代码部分无限制。需要包括至少 3 个测试数据。

简要解释解决思考过程和代码实现思路。

结论：闰年后一年的 2 月 28 日。

如 2017 年 2 月 28 号，共有 13 个订单到期； 分别为：2017 年 1 月 28 号、1 月 29 号、1 月 30 号、1 月 31 号；2016 年 2 月 28 号、2 月 29 号；8 月 28 号、8 月 29 号、8 月 30 号、8 月 31 号；11 月 28 号、11 月 29 号、11 月 30 号

问题 2： # 闰年判断函数 isLeapTear def isLeapYear(year): if (year%4 == 0) & (year%100 != 0): return 1 elif year%400 == 0: return 1 else: return 0 # 计算订单到期日的函数 def service_EndDate(y,m,d): assert y>0 and m>0 and m<=12 and d>0 and d<=31, '订单日期不正确' month_days = (0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31) leap_month_days = (0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31) if m == 12: endYear = y + 1 endMonth = 1 endDay = d else: if isLeapYear(y): #闰年到期日计算 assert d <= leap_month_days[m], '订单日期不正确' endYear = y endMonth = m + 1 if leap_month_days[m+1] >= d: endDay = d else: endDay = leap_month_days[m+1] else:# 非闰年到期日计算 assert d <= month_days[m], '订单日期不正确' endYear = y endMonth = m + 1 if month_days[m + 1] >= d: endDay = d else: endDay = month_days[m + 1] print(' 订购日： {}-{}-{}, 订单到期日： {}-{}-{}'.format(y,m,d,endYear,endMonth,endDay)) # 测试数据 def main(): service_EndDate(2016,1,30) service_EndDate(2018,1,29) service_EndDate(2000,1,31) service_EndDate(1996,7,31) service_EndDate(2008,12,31) if __name__ == '__main__': main()

中软国际面试内容

问题 01：

迭代器，list 是迭代器吗（list 是迭代器吗？）

迭代 生成

for 循环遍历的原理

for 循环遍历的原理就是迭代，in 后面必须是可迭代对象

为什么要有迭代器

对于序列类型：字符串、列表、元组，我们可以使用索引的方式迭代取出其包含的元素。但对于字典、集合、文件等类型是没有索引的，若还想取出其内部包含的元素，则必须找出一种不依赖于索引的迭代方式，这就是迭代器

1.可迭代对象

有 `__iter__` 方法的对象，都是可迭代对象，有以下 6 种

可迭代的对象：Python 内置 `str`、`list`、`tuple`、`dict`、`set`、`file` 都是可迭代对象

```
"zx".__iter__()
["zx"].__iter__()
{"zx":"wl"}.__iter__()
("zx",).__iter__()
{"z","x"}.__iter__()
with open("prize.txt","r") as file:
    file.__iter__()
```

2.迭代器对象

1.可迭代的对象执行 `__iter__` 方法得到的返回值是迭代器对象。2.迭代器对象指的是即内置有 `__iter__` 又内置有 `__next__` 方法的对象

```
list=[1,2,3,4,5,6]
zx=list.__iter__()
while True:
    try:
        print(zx.__next__())
    except:
        break
```

文件类型是迭代器对象

```
open('a.txt').__iter__()
open('a.txt').__next__()
```

迭代器对象一定是可迭代对象，而可迭代对象不一定是迭代器对象

生成器

生成器就是一种迭代器。迭代器有一个特点就是可以被 `next()` 函数调用并不断返回下一个值的对象，并且只能迭代它们一次。原因很简单,因为它们不是全部存在内存里,它们只在要调用的时候在内存里生成。

生成器就是迭代器（函数返回的结果就是生成器,而且它同时有 `__iter__` 和 `__next__`）

```
def zx():
    yield 1
    yield 2

x=zx()
x.__next__()
x.__iter__()
```

实验室

每次 `__next__` 之后，会执行到相应的 `yield` 不会执行下面的内容。

为啥？这个实验执行了呢，因为用了 `for`，他不知道最后执行到哪里，他就会一直 `__next__`，直到最后抛出异常，然而 `for in` 里面自带捕获异常的内容，所有没有打印异常信息。

```
def func():
    yield [1,1,23] # yield 会使函数 func() 变成生成器对象，因此他就具有__iter__方法
    print(789) # yield 会停止函数，当运行下一次 next 才会继续运行下面的代码
    yield 101112 # 一个 yield 对应一个 next
    print(131415)

g = func()
for i in g:
    print(i)
```

[1, 1, 23]

789

101112

131415

关于迭代器和和生成器的区别

生成器是一种特殊的迭代器，生成器实现了 `迭代器协议--iter--,--next--`

生成器是可以改变迭代的值的，然而迭代器随意改值会有问题

关于迭代器和 list 的区别

list 直接把所有数据加载到内存

而迭代器是一个一个取值，在需要下一个值的时候才回去计算取出这个值到内存（可以把迭代器想象成一个生成器的代码，一次 `next`，运行下面的代码，然后返回值）

问题 02:

装饰器，传参

python 装饰器和函数传参

装饰器<#>

装饰器是一个返回函数的高阶函数。

- 装饰器常见用法:
打印日志

```
def logger(func):  
    def wrapper(*args, **kw):  
        print 'do {}'.format(func.__name__)  
        func(*args, **kw)  
        print 'finish'  
    return wrapper  
  
@logger  
def add(x, y):  
    print '{} + {} = {}'.format(x, y, x+y)  
  
add(3, 5)
```

在函数执行前，打印一行日志 do...；函数执行结束，打印一行日志 finish。执行结果如下：

```
do add  
3 + 5 = 8  
finish
```

计算时间

```
import time  
def timer(func):  
    def wrapper(*args, **kw):  
        t1 = time.time()  
        func(*args, **kw)  
        t2 = time.time()  
        cost_time = t2 - t1  
        print 'cost time: {} s'.format(cost_time)  
    return wrapper  
  
@timer  
def cost_time(sleep_time):
```

```
        time.sleep(sleep_time)

cost_time(10)
```

- 带参数的函数装饰器

```
def say_hello(country):
    def wrapper(func):
        def decorate(*args,**kw):
            if country == 'en':
                print 'hello'

            elif country == 'usa':
                print 'hi'

            else:

                return

            func(*args,**kw)

        return decorate

    return wrapper

@say_hello("usa")
def usa():
    print 'i am from usa'

@say_hello("en")
def en():
    print 'i am from england'

usa()
print '-----',
en()
```

装饰器本身是一个函数，使用两层嵌套传参，执行结果如下：

```
hi
i am from usa
-----

hello
i am from england
```

- 不带参数的类装饰器

基于类装饰器的实现，必须实现__call__和__init__两个内置函数。

__init__：接收被装饰函数

__call__：实现装饰逻辑

```
class logger(object):  
    def __init__(self, func):  
        self.func = func  
  
    def __call__(self, *args, **kwargs):  
        print 'the function {} () is running...'\  
            .format(self.func.__name__)  
        return self.func(*args, **kwargs)  
  
@logger  
def say(something):  
    print 'say {}'.format(something)  
  
say('hello')
```

运行结果如下：

```
the function say() is running...  
say hello!
```

- 带参数的类装饰器

带参数和不带参数的类装饰器有很大的不同。

__init__：不再接收被装饰函数，而是接收传入参数

__call__：接收被装饰函数，实现装饰逻辑

```
class logger(object):  
    def __init__(self, level='INFO'):  
        self.level = level  
  
    def __call__(self, func):  
        def wrapper(*args, **kwargs):  
            print '{level}: the function {func} () is running...'\  
                .format(level=self.level, func=func.__name__)  
            func(*args, **kwargs)  
        return wrapper
```

```
@logger(level='WARNING')
def say(something):
    print 'say {}!'.format(something)

say('hello')
```

运行结果如下:

```
WARNING: the function say () is running...
say hello!
```

函数的参数#

- 位置参数

```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

power(x, n)函数有两个参数: x 和 n, 这两个参数都是位置参数, 调用函数时, 传入的两个值按照位置顺序依次赋值给参数 x 和 n。

- 默认参数

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

power(x, n)函数有两个参数: x 和 n, 如果想在传入 n 值时, 默认计算 x 的平方, 此时可以将 n 设为默认值 2。

- 可变参数(*args)

```
def function(f_arg, *args):
    print f_arg, type(f_arg)
```

```
        print args, type(args)

nums = ['a','b','c']

function(1,2,*nums)
```

定义可变参数时，需要在参数前面加一个*号，可变参数的个数是可变的。在函数内部，参数*args 接收到的是一个 tuple。输出结果如下：

```
1 <type 'int'>
(2, 'a', 'b', 'c') <type 'tuple'>
```

- 关键字参数(**kwargs)

```
def person(name, age, **kwargs):
    print 'name:', name, 'age:', age, 'other:', kwargs, type(kwargs)

person('mark', 30, city='shanghai')
```

**kwargs 允许将不定长度的键值对，作为参数传递给一个函数，关键字参数在函数内部自动组装为一个 dict。输出结果如下：

```
name: mark age: 30 other: {'city': 'shanghai'} <type 'dict'>
```

- 将函数作为参数传递给另一个函数

```
def hi():
    return 'hi friends'

def function(func):
    print 'just test'
    print func()

function(hi)
```

function() 函数将 hi 函数作为参数接收，输出结果如下：

```
just test
hi friends
```

time 模块<#>

- 获取当前时间


```
>>> time.localtime()
time.struct_time(tm_year=2019, tm_mon=8, tm_mday=21, tm_hour=14, tm_min=31, tm_sec=18, tm_wday=2, tm_yday=233, tm_isdst=0)
```

- 获取格式化的时间

```
>>> time.ctime()
'Wed Aug 21 14:51:28 2019'
>>> time.asctime()
'Wed Aug 21 14:51:34 2019'
```

- 格式化日期

```
>>> time.strftime('%Y-%m-%d %H:%M:%S',time.localtime())
'2019-08-21 14:35:02'
>>> time.strftime('%a %b %d %H:%M:%S %Y',time.localtime())
'Wed Aug 21 14:36:09 2019'
```

- 计算运行时间

```
Copy
import time
start = time.time()
time.sleep(2)
end = time.time()
print end-start
```

装饰器--万能传参

装饰器的万能传参 (*args, **kwargs)

案例

一、环境：以上为线上代码，需要添加 1 个统计执行时间的功能。线上代码如下：

```
1 #!/usr/bin/env python
2 # -*- coding:utf8 -*-
3 # Author:Dong Ye
5 import time
8 def test1():
9     time.sleep(3)
```

```
10     print('in the test1')
11 def test2():
12     time.sleep(3)
13     print('in the test2')
16 test1()
17 test2()
```

二、需求：在不修改源代码（test1 & test2）和原代码调用方式的情况下，给 test1 新增这个功能。

三、思路：

- 1、结合装饰器的特点：高阶函数 + 嵌套函数 = 装饰器
- 2、使用嵌套函数把新增功能和源代码结合起来，并返回嵌套函数的内存地址。
- 3、通过高阶函数把内存地址返回，然后重新覆盖掉 test1 的函数名。

代码实现 1

一、操作步骤：

- 1、先定义个高阶函数的 decorator。
- 2、嵌套函数将高阶函数的内存地址返回给 test1。
- 3、在高阶函数中调用 test1 的函数体与新增结果结合。

实例：（以下是装饰器的执行顺序）

```
1 def timer(func):    #2、timer(test1)    func = test1
2     def deco():    #3、在内存里定义了一个变量
3         start_time = time.time()    #7、执行开始时间
4         func()    #8、func() 执行，其实就是执行了原代码 test1
5         stop_time = time.time()    #9、执行结束时间
6         print('the func run time is %s' % (stop_time-start_time))    #10、打印出 test1 的执行时间
7     return deco    #4、 返回 deco 函数的内存地址
9 test1 = timer(test1)    #1、调用 timer 函数，将 test1 的变量传给 func
10    #5、将 deco 的内存地址返回给 test1
11 test1()    #6、执行 test1()，其实是执行了 deco() 函数体
13 #test2 = timer(test2)
14 #test2()
```

代码实现 2

优化第一个装饰器代码（无参数传值）：

- 1、由于第一个装饰器是按照函数调用传值的方式展现的。
- 2、如果装饰函数体过多则会显得装饰器很乱，不易读写。

优化后代码：

```
1 import time    #1 导入 time 模块
```

```

2 def timer(func):    #2 相当于在内存中定义一个变量    #4 @timer 其实是 tist1 = timer(test1)，所以会调用到 timer 函数。
3     def deco():    #5、将 func 定义成一个高阶函数    #8 在调用 test1 时，实际调用的是 deco 函数
4         start_time = time.time()    #9 获取当前值
5         func()    #10 调用源代码函数
6         stop_time = time.time()    #14 返回装饰器，继续下一个功能
7         print('the func run time is %s'%(stop_time-start_time))    #15 打印
8     return deco    #6 返回高阶函数的内存地址
10 @timer    #这个表示 test1 = timer(test1)    #3 指定需要装饰的源代码
11 def test1():    #11 调用源代码
12     time.sleep(3)    #12 调用源代码
13     print('in the test1')    #13 调用源代码
17 test1()    #7 调用 test1()，实际执行的是装饰器中的 deco。

```

代码实现 3

一旦这个函数被装饰器装饰，那么这个函数将会被重新赋值，赋值成装饰器函数的内存函数。

```

1 #定义嵌套函数
2 def timer(func):    #定义 timer 函数为了传递 test 参数 func = test1
3     def deco(*args,**kwargs):    #定义功能函数    #由于在工作中参的参数和参数功能不统一，所以在 deco() 和 func() 函数中使用 2 个万能参数*args 和**kwargs;
4         start_time=time.time()
5         func(*args,**kwargs)    #执行 test 传参的函数 func() = test1()
6         stop_time = time.time()
7         print('the func run time is %s' % (stop_time-start_time))
8     return deco    #返回 deco 函数的内存地址
11 @timer    #test1 = timer(test1) = deco    test1() = deco()
12 def test1():
13     time.sleep(3)
14     print('in the test1')
16 @timer    #test2 = timer(test2) = deco    test2(name,age) = deco(name,age)
17 def test2(name,age):
18     time.sleep(3)
19     print('name: %s    age: %s' % (name, age))
21 test1()    # test1() = deco()
22 test2('dongye',33)    #test2(name,age) = deco(name,age)
26 # 注释:
27 # test2 的赋值是嵌套函数 timer(test2) 返回来的 deco 的内存地址;
28 # 在调用 test2() 的时候，实际上是在调用 deco() 函数

```

```
29 # 需要注意的 2 个地方：
30 #      1、调用 test2() 函数时，传值的 neme 和 age 实际是传值给了 deco 嵌套函数中；
31 #      2、如果 deco() 函数与内部执行的 func() 中，没有指定形参，则会报错；
32 #      3、由于在工作中产的参数和参数功能不统一，所以在 deco() 和 func() 函数中使用 2 个万能参数*args 和**kwargs；
```

问题 03:

多继承（C 继承 A，B，继承顺序），答得一般，没有答深度优先，广度优先以及 mro

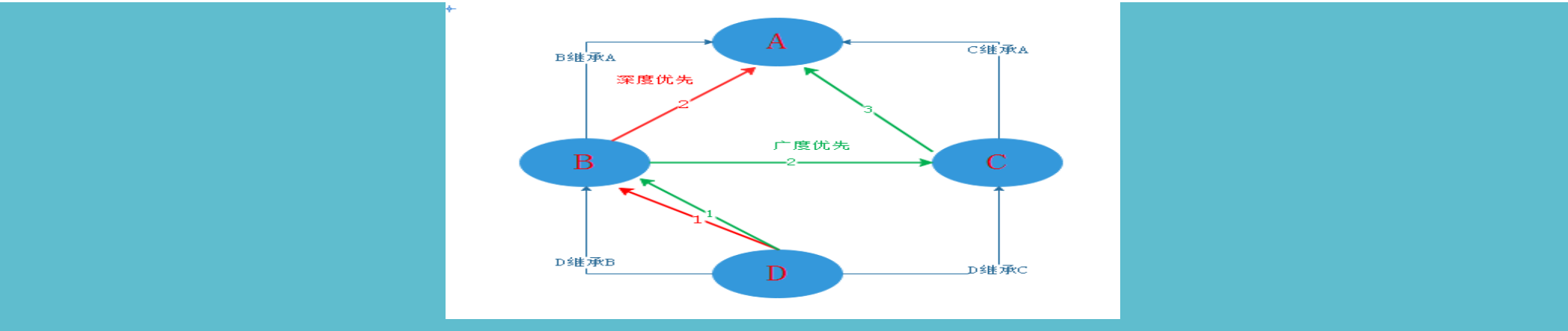
先传进来哪个就找哪个，如果 A 上层还有，继续找上层

经典类深度优先，新式类广度优先

如果关系比较乱，使用 mro()方法查看继承图

深度优先 or 广度优先

深度优先和广度优先是两种不同的算法策略，两者有什么区别呢？我直接上图解释吧。



如图，B 继承 A， C 继承 A， D 继承 B 和 C。

深度优先遍历是从 D 开始往上搜索到 B，若 B 没有数据，则继续往上搜索到 A；

广度优先遍历是从 D 开始往上搜索到 B，若 B 没有数据，则搜索和 B 同级的 C 里的数据，若同级的 C 里还是没有数据，再继续往上搜索到 A 。

Tips: py2 经典类是按深度优先来继承的，新式类是按广度优先来继承的。py3 经典类和新式类都是统一按广度优先来继承的。

python 之 多继承的顺序

python 支持多继承，但对与经典类和新式类来说，多继承查找的顺序是不一样的。

经典类:

```
class P1:
    def foo(self):
        print 'p1-foo'

class P2:
    def foo(self):
        print 'p2-foo'

    def bar(self):
        print 'p2-bar'

class C1 (P1,P2):
    pass

class C2 (P1,P2):
    def bar(self):
        print 'C2-bar'

class D(C1,C2):
    pass
```

新式类

```
class P1(object):
    def foo(self):
        print 'p1-foo'

class P2(object):
    def foo(self):
        print 'p2-foo'

    def bar(self):
        print 'p2-bar'

class C1 (P1,P2):
    pass

class C2 (P1,P2):
    def bar(self):
        print 'C2-bar'

class D(C1,C2):
    pass
```

1. 经典类

```
d = D()
d.foo() # 输出 p1-foo
d.bar() # 输出 p2-bar
```

实例 d 调用 foo() 时, 搜索顺序是 D => C1 => P1

实例 d 调用 bar() 时, 搜索顺序是 D => C1 => P1 => P2

换句话说, 经典类的搜索方式是按照“从左至右, 深度优先”的方式去查找属性。d 先查找自身是否有 foo 方法, 没有则查找最近的父类 C1 里是否有该方法, 如果没有则继续向上查找, 直到在 P1 中找到该方法, 查找结束。

2. 新式类

```
d=D()
d.foo() # 输出 p1-foo
d.bar() # 输出 c2-bar
```

实例 d 调用 foo() 时, 搜索顺序是 D => C1 => C2 => P1

实例 d 调用 bar() 时, 搜索顺序是 D => C1 => C2

可以看出, 新式类的搜索方式是采用“广度优先”的方式去查找属性。

问题 04:

哪些不会被继承, 析构方法 (没答上)

魔法方法, 如: `__del__()`

1、类方法是从属于类对象的方法, 通过装饰器 `@classmethod` 来调用. 使用规范如下:

a、@classmethod 必须位于方法上面一行

b、第一个 cls 必须有，是类对象本身

c、类方法和静态方法不能访问实例属性和实例方法。

d、子类继承父类的方法时，传入的 cls 是子类对象，而非父类对象。

2、静态方法是允许定义与“类对象”无关的方法，称为静态方法，通过装饰器@staticmethod 来调用，使用规则如下：

a、@staticmethod 必须位于方法最上面

b、类方法和静态方法不能访问实例属性和实例方法。

3、析构方法的理解：__del__方法称为析构方法，用于实现对象被销毁时所需要的操作。比如：释放对象占用的资源。

Pythod 实现自动的垃圾回收机制，当对象没有引用时，自动调用__del__析构方法。

class Del:

```
    def __del__(self):
```

```
        print("对象{0}已被删除".format(self))
```

```
s1=Del()
```

```
s2=Del()
```

```
del s1
```

```
print("over")
```

```
#返回值
```

```
对象<__main__.Del object at 0x000001B58435FAC8>已被删除
```

```
over
```

```
对象<__main__.Del object at 0x000001B58435FB38>已被删除
```

```
#del s1 调用对象<__main__.Del object at 0x000001B58435FAC8>已被删除
```

```
#over 打印
```

```
#程序结束，s2 自动销毁 #对象<__main__.Del object at 0x000001B58435FB38>已被删除
```

构造函数

用于初始化类的内容部状态，Python 提供的构造函数式 __init__(), 也就是当该类被实例化的时候就会执行该函数，__init__()方法是可选的，如果不提供，Python 会给出默认的__init__方法。

析构函数

“__del__”就是一个析构函数了，当使用 del 删除对象时，会调用他本身的析构函数，另外当对象在某个作用域中调用完毕，在跳出其作用域的同时析构函数也会被调用一次，这样可以用来释放内存空间。

__del__()也是可选的，如果不提供，则 Python 会在后台提供默认析构函数

如果要显式的调用析构函数，可以使用 del 关键字： del obj

垃圾回收机制

1	s = '123'
2	print('del...running')
3	del s

当我们用 del 删除一个对象时，其实并没有直接清除该对象的内存空间。Python 采用 ‘引用计数’ 的算法方式来处理回收，即：当某个对象在其作用域内不再被其他对象引用的时候，Python 就自动清除对象。

而析构函数 __del__()在引用的时候就会自动清除被删除对象的内存空间。

析构放法：当对象在内存中被释放时，自动触发执行。
注：此方法一般无需定义，因为 python 是一门高级语言，
程序员在使用时无需关心内存的分配和释放，
析构函数的调用是由解释器在进行垃圾回收时是自动触发执行的

```
class Foo:
    def __init__(self,name):
        self.name=name
    def __del__(self):
        print('正在执行')

f1=Foo('pl')
del f1 #删除对象 所以可以触发
# del f1.name 这是属性的删除 不会触发
print("____>") # 进行垃圾回收 自动触发 del 函数
#del 整个实例删除是才会触发
```

python 析构方法__del__

析构方法
此方法一般无须定义，因为 Python 是一门高级语言，程序员在使用时无需关心内存的分配和释放，因为此工作都是交给 Python 解释器来执行，所以，析构函数的调用是由解释器在进行垃圾回收时自动触发执行的。
示例代码

“模拟 open 的函数，可以使用析构方法，释放内存，关闭打开的文件”

```
class Open:
    def __init__(self,filepath,mode="r",encode="utf-8"):
        self.f = open(filepath,mode=mode,encoding=encode)
    def write(self):
        pass
    def __getattr__(self, item):
        return getattr(self.f,item)
```

```
def __del__(self):
    print("---->del")
    self.f.close()
f = Open("a.txt", "w")
del f #关闭文件，释放内存
```

问题 05:

map()与 reduce() (reduce()没答上)

map() 会根据提供的函数对指定序列做映射。

map(function, iterable, ...)

第一个参数 **function** 以参数序列中的每一个元素调用 **function** 函数，返回包含每次 **function** 函数返回值的新列表。

Python 3.x 返回迭代器。

print(map()) 返回迭代器地址

一般和 list 一起用 才能输出

reduce 函数对参数迭代器中的元素进行类累积，先对集合中的第 1、2 个元素进行操作，得到的结果再与第三个数据用 **function** 函数运算，最后得到一个结果。

格式为: **reduce(func,iter,init)**

func 为函数，**iter** 为序列，**init** 为固定初始值，无初始值时从序列的第一个参数开始。

reduce(function, iterable[, initializer])

```
>>>def add(x,y):# 两数相加
```

```
... return x + y
```

```
...
```

```
>>> reduce(add, [1,2,3,4,5]) # 计算列表和: 1+2+3+4+5
```

```
15
```

```
>>> reduce(lambda x,y: x+y, [1,2,3,4,5]) # 使用 lambda 匿名函数
```

```
15
```

问题 06:

多线程的使用场景，GIL 原理（答得含糊，没有分析）

使用 **threading.Thread()** 方法

继承 **threading.Thread** 类

由于 GIL 的原因， Python 解释器只允许同一时间执行一个线程。多线程不能用，使用多进程

I/O 密集型使用多线程

CPU 密集型使用多进程

问题 07：
TCP 三次握手，四次挥手

问题 08：
CSRF 预防

项目进行安全测试时需要预防 CSRF 攻击，记录一下学习的过程。

一.CSRF 攻击是什么？

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。举个例子，你登录了信任网站 www.aaa.com（A 网站），A 网站服务器里的 session 存放了你的登录状态，并调用接口 http://api.aaa.com/getUserInfo 以获取你的个人信息，你在未登出时，又点击了恶意网站 www.bbb.com（B 网站），此时如果 B 网站上有类似,<script src="http://api.aaa.com/getUserInfo"></script>的代码，当 A 网站没有做 CSRF 预防，A 网站的服务器端就收到了 http://api.aaa.com/getUserInfo 的请求，并且因为 cookie 的存在，判断你处于登录的状态，返回了数据，至此，B 网站就窃取了你的个人信息。

二.如何预防？

CSRF 攻击是源于 Web 的会话身份验证机制，虽然可以保证一个请求是来自于某个用户的浏览器，但却无法保证该请求是用户批准发送的，这个情况是有可能发生的，预防一般是服务端来做。

- 1.检测请求的 referer，这种方式可以预防相当一部分的攻击，但是不全，因为 referer 可以被伪造。
- 2.添加 csrf_token，项目为后端渲染成 html 时可以生成一个 csrftoken 到表单中，请求时携带这个 token，前后端分离时的具体方法如下：
 - 1.用户登录时，服务端可以初始化一个不可预测的，随机 csrftoken 放到服务端 session 中，同时又放到前端 cookie 中；
 - 2.前端通过 js 抓取 cookie 里面的 csrftoken 放到请求的 head 中去；
 - 3.服务端收到请求后，会验证前端传递来的 csrftoken 与之前存在 session 中的 csrftoken 是否一致，一致则说明没别劫持；
 - 4.原因是，黑客可以构造请求数据，并且携带 cookie 去请求，但是却无法获取 cookie 的值，而且请求中的 csrftoken 是变化的，黑客就无法通过服务端的验证，请求就会失败；

问题 09：
MySQL 隔离级别，哪些产生脏读，哪些产生幻读

- 1.脏读：脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。
- 2.不可重复读：是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。例如，一个编辑人员两次读取同一文档，但在两次读取之间，作者重写了该文档。当编辑人员第二次读取文档时，文档已更改。原始读取不可重复。如果只有在作者全部完成编写后编辑人员才可以读取文档，则可以避免该问题。
- 3.幻读：是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。例如，一个编辑人员更改作者提交的文档，但当生产部门将其更改内容合并到该文档的主副本时，发现作者已将未编辑的新材料添加到该文档中。如果在编辑人员和生产部门完成对原始文档的处理之前，任何人都不能将新材料添加到文档中，则可以避免该问题。

补充：基于元数据的 Spring 声明性事务：

Isolation 属性一共支持五种事务设置，具体介绍如下：

- | DEFAULT 使用数据库设置的隔离级别（默认），由 DBA 默认的设置来决定隔离级别。
- | READ_UNCOMMITTED 会出现脏读、不可重复读、幻读（隔离级别最低，并发性能高）
- | READ_COMMITTED 会出现不可重复读、幻读问题（锁定正在读取的行）
- | REPEATABLE_READ 会出幻读（锁定所读取的所有行）
- | SERIALIZABLE 保证所有的情况不会发生（锁表）

不可重复读的重点是修改：

同样的条件，你读取过的数据，再次读取出来发现值不一样了

幻读的重点在于新增或者删除

同样的条件，第 1 次和第 2 次读出来的记录数不一样

隔离级别	
隔离级别	作用
Serializable(串行化)	避免脏读、不可重复读、幻读
Repeatable(可重复读)	避免脏读、不可重复读
Read committed(读已提交)	避免脏读
Read uncommitted(读未提交)	none

- mysql支持上面4种隔离级别，默认为可重复读

问题 10：

MySQL 什么情况下创建索引

较频繁地作为查询条件的字段
出现 where 的字段

- 1、什么事索引（本质：数据结构）
索引是帮助 MySQL 高效获取数据的数据结构。
- 2、优势：
 - 1、提高数据检索的效率，降低数据库 IO 成本
 - 2、通过索引对数据进行排序，降低数据排序的成本，降低了 CPU 的消耗
- 3、劣势：

降低更新表的速度，如对表进行 **update** 、**delete**、**insert** 等操作时，MySQL 不急要保存数据，还要保存一下索引文件每次添加了索引列的字段，都会调整因为更新带来的键值变化后的索引信息。

4、适合创建索引条件

- 1、主键自动建立唯一索引：表的主关键字自动建立唯一索引
- 2、频繁作为查询条件的字段应该建立索引
- 3、查询中与其他表关联的字段，外键关系建立索引
- 4、单键/组合索引的选择问题，组合索引性价比更高
- 5、查询中排序的字段，排序字段若通过索引去访问将大大提高排序效率
- 6、查询中统计或者分组字段

5、不适合创建索引条件

- 1、表记录少的：唯一性太差的字段不适合建立索引

什么是唯一性太差的字段。如状态字段、类型字段。那些只存储固定几个值的字段，例如用户登录状态、消息的 **status** 等。这个涉及到了索引扫描的特性。例如：通过索引查找键值为 A 和 B 的某些数据，通过 A 找到某条相符合的数据，这条数据在 X 页上面，然后继续扫描，又发现符合 A 的数据出现在了 Y 页上面，那么存储引擎就会丢弃 X 页面的数据，然后存储 Y 页面上的数据，一直到查找完所有对应 A 的数据，然后查找 B 字段，发现 X 页面上面又有对应 B 字段的数据，那么他就会再次扫描 X 页面，等于 X 页面就会被扫描 2 次甚至多次。以此类推，所以同一个数据页可能会被多次重复的读取，丢弃，在读取，这无疑给存储引擎极大地增加了 IO 的负担。

- 2、经常增删改的表或者字段：更新太频繁地字段不适合创建索引
- 3、**where** 条件里用不到的字段不创建索引
- 4、过滤性不好的不适合建索引

一，什么情况下使用索引

1. 表的主关键字

自动建立唯一索引

2. 表的字段唯一约束

ORACLE 利用索引来保证数据的完整性

3. 直接条件查询的字段

在 SQL 中用于条件约束的字段

如 **zl_yhjbqk**（用户基本情况）中的 **qc_bh**（区册编号）

```
select * from zl_yhjbqk where qc_bh='7001'
```

4. 查询中与其它表关联的字段

字段常常建立了外键关系

如 **zl_ydcf**（用电成份）中的 **jlbd_bh**（计量点表编号）

```
select * from zl_ydcf a,zl_yhdb b where a.jlbd_bh=b.jlbd_bh and b.jlbd_bh='540100214511'
```

5. 查询中排序的字段

排序的字段如果通过索引去访问那将大大提高排序速度

```
select * from zl_yhjbqk order by qc_bh（建立 qc_bh 索引）
```

select * from zl_yhjbqk where qc_bh=‘7001’ order by cb_sx （建立 qc_bh+cb_sx 索引，注：只是一个索引，其中包括 qc_bh 和 cb_sx 字段）

6. 查询中统计或分组统计的字段

select max(hbs_bh) from zl_yhjbqk

select qc_bh,count(*) from zl_yhjbqk group by qc_bh

二，什么情况下应不建或少建索引

1. 表记录太少

如果一个表只有 5 条记录，采用索引去访问记录的话，那首先需访问索引表，再通过索引表访问数据表，一般索引表与数据表不在同一个数据块，这种情况下 ORACLE 至少要往返读取数据块两次。而不用索引的情况下 ORACLE 会将所有的数据一次读出，处理速度显然会比用索引快。

如表 zl_sybm（使用部门）一般只有几条记录，除了主关键字外对任何一个字段建索引都不会产生性能优化，实际上如果对这个表进行了统计分析后 ORACLE 也不会用你建的索引，而是自动执行全表访问。如：

select * from zl_sybm where sydw_bh=‘5401’（对 sydw_bh 建立索引不会产生性能优化）

2. 经常插入、删除、修改的表

对一些经常处理的业务表应在查询允许的情况下尽量减少索引，如 zl_yhbm，gc_dfss，gc_dfys，gc_fpdy 等业务表。

3. 数据重复且分布平均的表字段

假如一个表有 10 万行记录，有一个字段 A 只有 T 和 F 两种值，且每个值的分布概率大约为 50%，那么对这种表 A 字段建索引一般不会提高数据库的查询速度。

4. 经常和主字段一块查询但主字段索引值比较多的表字段

如 gc_dfss（电费实收）表经常按收费序号、户标识编号、抄表日期、电费发生年月、操作 标志来具体查询某一笔收款的情况，如果将所有的字段都建在一个索引里那将会增加数据的修改、插入、删除时间，从实际上分析一笔收款如果按收费序号索引就已 经将记录减少到只有几条，如果再按后面的几个字段索引查询将对性能不产生太大的影响。

不论你在什么时候开始，重要的是开始之后就不要停止。

问题 11：

Redis，缓存穿透与缓存雪崩，解决办法（解决办法答的不够多）

缓存穿透：缓存与数据库中都没有，恶意访问，频繁的查数据库，压力大

解决办法：查询一次，如果没有，缓存中增加空值

采用布隆过滤器（将所有可能存在的数据哈希到一个足够大的 bitmap 中）

缓存雪崩：同一时间，大量键过期，同时访问数据库，压力大

解决办法：设置过期时间为随机

缓存击穿：前台高并发访问一个已经在缓存中恰好过期的数据，全部打到后台数据库

解决办法：互斥锁，Redis 的 SETNX 功能，数据没有，返回一个设定值，再去查数据库

热点数据设置永不过期

缓存击穿

缓存在同一时间内大量键过期（失效），接着来的一大波请求瞬间都落在了数据库中导致连接异常。

解决方案：

方案 1、也是像解决缓存穿透一样加锁排队，实现同上；

方案 2、建立备份缓存，缓存 A 和缓存 B，A 设置超时时间，B 不设值超时时间，先从 A 读缓存，A 没有读 B，并且更新 A 缓存和 B 缓存；

方案 3、设置缓存超时时间的时候加上一个随机的时间长度，比如这个缓存 key 的超时时间是固定的 5 分钟加上随机的 2 分钟，酱紫可从一定程度上避免雪崩问题；

缓存穿透：第一次看到这个名字，会觉得是一个很高深的名词。但是和其他许多概念一样，它只是描述了一个很容易理解的现象：请求了不存在的数据。造成大量的请求没有命中缓存场景之一：数据库使用了 id 为正整数作为键，但是黑客使用负整数向服务器发起请求，这时所有的请求都没有在缓存中命中，从而导致大量请求数据库，如果超过了数据库的承载能力，会导致数据库服务器宕机。

解决缓存穿透的方案主要有两种：

- 1，当查询不存在时，也将结果保存在缓存中。但是这可能会存在一种问题：大量没有查询结果的请求保存在缓存中，这时我们就可以将这些请求的 key 设置得更短一些。
- 2，提前过滤掉不合法的请求，Redis 实现了布隆过滤器，我们可以使用它来达到这个目的。布隆过滤器很好理解，可以参考[布隆过滤器\(Bloom Filter\)的原理和实现](#)。

缓存穿透是指查询一个一定不存在的数据，由于缓存不命中，接着查询数据库也无法查询出结果，因此也不会写入到缓存中，这将会导致每个查询都会去请求数据库，造成缓存穿透；

解决方案

布隆过滤

对所有可能查询的参数以 hash 形式存储，在控制层先进行校验，不符合则丢弃，从而避免了对底层存储系统的查询压力；

缓存空对象

当存储层不命中后，即使返回的空对象也将其缓存起来，同时会设置一个过期时间，之后再访问这个数据将会从缓存中获取，保护了后端数据源；

但是这种方法会存在两个问题：

如果空值能够被缓存起来，这就意味着缓存需要更多的空间存储更多的键，因为这当中可能会有很多的空值的键；

即使对空值设置了过期时间，还是会存在缓存层和存储层的数据会有一段时间窗口的不一致，这对于需要保持一致性的业务会有影响。

缓存雪崩

缓存雪崩是指，由于缓存层承载着大量请求，有效的保护了存储层，但是如果缓存层由于某些原因整体不能提供服务，于是所有的请求都会达到存储层，存储层的调用量会暴增，造成存储层也会挂掉的情况。

解决方案

保证缓存层服务高可用性

即使个别节点、个别机器、甚至是机房宕掉，依然可以提供服务，比如 Redis Sentinel 和 Redis Cluster 都实现了高可用。

依赖隔离组件为后端限流并降级

在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待。

数据预热

可以通过缓存 reload 机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存不同的 key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

缓存并发

缓存并发是指，高并发场景下同时大量查询过期的 key 值、最后查询数据库将缓存结果回写到缓存、造成数据库压力过大

分布式锁

在缓存更新或者过期的情况下，先获取锁，在进行更新或者从数据库中获取数据后，再释放锁，需要一定的时间等待，就可以从缓存中继续获取数据。

缓存雪崩问题：缓存雪崩是指缓存大量失效，导致大量的请求都直接向数据库获取数据，造成数据库的压力。缓存大量失效的原因可能是缓存服务器宕机，或者大量 Redis 的键设置的过期时间相同。

解决缓存雪崩我们也有两种解决方案：

- 1，在设置 Redis 键的过期时间时，加上一个随机数，这样可以避免。
- 2，部署分布式的 Redis 服务，当一个 Redis 服务器挂掉了之后，进行故障转移。

问题 12:

Redis 的内存清理机制

Redis 内存回收机制

为什么需要内存回收？

原因有如下两点：

- 在 Redis 中，Set 指令可以指定 Key 的过期时间，当过期时间到达以后，Key 就失效了。
- Redis 是基于内存操作的，所有的数据都是保存在内存中，一台机器的内存是有限且很宝贵的。

基于以上两点，为了保证 Redis 能继续提供可靠的服务，Redis 需要一种机制清理掉不常用的、无效的、多余的数据，失效后的数据需要及时清理，这就需要内存回收了。

Redis 的内存回收机制

Redis 的内存回收主要分为过期删除策略和内存淘汰策略两部分。

过期删除策略

删除达到过期时间的 Key。

- ①定时删除
- ②惰性删除
- ③定期删除：过期删除策略原理

为了大家听起来不会觉得疑惑，在正式介绍过期删除策略原理之前，先给大家介绍一点可能会用到的相关 Redis 基础知识。

- ①RedisDB 结构体定义
- ②expires 属性
- ③Redis 清理过期 Key 的时机
- ④过期策略的实现
- ⑤删除 Key

小结：总的来说，Redis 的过期删除策略是在启动时注册了 serverCron 函数，每一个时间时钟周期，都会抽取 expires 字典中的部分 Key 进行清理，从而实现定期删除。

另外，Redis 会在访问 Key 时判断 Key 是否过期，如果过期了，就删除，以及每一次 Redis 访问事件到来时，beforeSleep 都会调用 activeExpireCycle 函数，在 1ms 时间内主动清理部分 Key，这是惰性删除的实现。

内存淘汰策略

Redis 的内存淘汰策略，是指内存达到 maxmemory 极限时，使用某种算法来决定清理掉哪些数据，以保证新数据的存入。

Redis 的内存淘汰机制如下：

- noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。
- allkeys-lru：当内存不足以容纳新写入数据时，在键空间（server.db[i].dict）中，移除最近最少使用的 Key（这个是最常用的）。
- allkeys-random：当内存不足以容纳新写入数据时，在键空间（server.db[i].dict）中，随机移除某个 Key。

- **volatile-lru**: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间 (`server.db[i].expires`) 中, 移除最近最少使用的 **Key**。
- **volatile-random**: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间 (`server.db[i].expires`) 中, 随机移除某个 **Key**。
- **volatile-ttl**: 当内存不足以容纳新写入数据时, 在设置了过期时间的键空间 (`server.db[i].expires`) 中, 有更早过期时间的 **Key** 优先移除。

在配置文件中, 通过 `maxmemory-policy` 可以配置要使用哪一个淘汰机制。

总结

Redis 对于内存的回收有两种方式, 一种是过期 **Key** 的回收, 另一种是超过 **Redis** 的最大内存后的内存释放。

对于第一种情况, **Redis** 会在:

- 每一次访问的时候判断 **Key** 的过期时间是否到达, 如果到达, 就删除 **Key**。
- **Redis** 启动时会创建一个定时事件, 会定期清理部分过期的 **Key**, 默认是每秒执行十次检查, 每次过期 **Key** 清理的时间不超过 **CPU** 时间的 25%。
即若 `hz=1`, 则一次清理时间最大为 `250ms`, 若 `hz=10`, 则一次清理时间最大为 `25ms`。

对于第二种情况, **Redis** 会在每次处理 **Redis** 命令的时候判断当前 **Redis** 是否达到了内存的最大限制, 如果达到限制, 则使用对应的算法去处理需要删除的 **Key**。

问题 13:

Restful 接口, 如果是做计算, 群增等设计, 返回的状态码

一、重要概念: **REST**,即 **Representational State Transfer** 的缩写。我对这个词组的翻译是"表现层状态转化"。

Resource (资源) : 对象的单个实例。 例如, 一只动物。它可以是一段文本、一张图片、一首歌曲、一种服务, 总之就是一个具体的实在。

你可以用一个 **URI** (统一资源定位符) 指向它, 每种资源对应一个特定的 **URI**。要获取这个资源, 访问它的 **URI** 就可以, 因此 **URI** 就成了每一个资源的地址或独一无二的识别符。

集合: 对象的集合。 例如, 动物。

第三方: 使用我们接口的开发者

表现层 (**Representation**)

"资源"是一种信息实体, 它可以有多种外在表现形式。我们把"资源"具体呈现出来的形式, 叫做它的"表现层" (**Representation**)。

状态转化 (**State Transfer**)

访问一个网站, 就代表了客户端和服务器的一个互动过程。在这个过程中, 势必涉及到数据和状态的变化。互联网通信协议 **HTTP** 协议, 是一个无状态协议。这意味着, 所有的状态都保存在服务器端。因此, 如果客户端想要操作服务器, 必须通过某种手段, 让服务器端发生"状态转化" (**State Transfer**)。而这种转化是建立在表现层之上的, 所以就是"表现层状态转化"。

客户端用到的手段, 只能是 **HTTP** 协议。具体来说, 就是 **HTTP** 协议里面, 四个表示操作方式的动词: **GET**、**POST**、**PUT**、**DELETE**。

它们分别对应四种基本操作: **GET** 用来获取资源, **POST** 用来新建资源 (也可以用于更新资源), **PUT** 用来更新资源, **DELETE** 用来删除资源。

比如, 文本可以用 `txt` 格式表现, 也可以用 **HTML** 格式、**XML** 格式、**JSON** 格式表现, 甚至可以采用二进制格式; 图片可以用 `JPG` 格式表现, 也可以用 `PNG` 格式表现。

URI 只代表资源的实体, 不代表它的形式。严格地说, 有些网址最后的".html"后缀名是不必要的, 因为这个后缀名表示格式, 属于"表现层"范畴,

而 **URI** 应该只代表"资源"的位置。它的具体表现形式, 应该在 **HTTP** 请求的头信息中用 **Accept** 和 **Content-Type** 字段指定, 这两个字段才是对"表现层"的描述。

综合上面的解释, 我们总结一下什么是 **RESTful** 架构:

- (1) 每一个 **URI** 代表一种资源;
- (2) 客户端和服务端之间, 传递这种资源的某种表现层;
- (3) 客户端通过四个 **HTTP** 动词, 对服务器端资源进行操作, 实现"表现层状态转化"。

二、REST 接口规范

1、动作

- GET （SELECT）：从服务器检索特定资源，或资源列表。
- POST （CREATE）：在服务器上创建一个新的资源。
- PUT （UPDATE）：更新服务器上的资源，提供整个资源。
- PATCH （UPDATE）：更新服务器上的资源，仅提供更改的属性。
- DELETE （DELETE）：从服务器删除资源。

首先是四个半种动作：

post、delete、put/patch、get

因为 put/patch 只能算作一类，所以将 patch 归为半个。

另外还有有两个较少知名的 HTTP 动词：

HEAD - 检索有关资源的元数据，例如数据的哈希或上次更新时间。

OPTIONS - 检索关于客户端被允许对资源做什么的信息。

2、路径（接口命名）

路径又称"终点"（endpoint），表示 API 的具体网址。

在 RESTful 架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的"集合"（collection），所以 API 中的名词也应该使用复数。

举例来说，有一个 API 提供动物园（zoo）的信息，还包括各种动物和雇员的信息，则它的路径应该设计成下面这样。

接口尽量使用名词，禁止使用动词，下面是一些例子。

GET	/zoos：列出所有动物园
POST	/zoos：新建一个动物园
GET	/zoos/ID：获取某个指定动物园的信息
PUT	/zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）
PATCH	/zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）
DELETE	/zoos/ID：删除某个动物园
GET	/zoos/ID/animals：列出某个指定动物园的所有动物
DELETE	/zoos/ID/animals/ID：删除某个指定动物园的指定动物

反例：

```
/getAllCars
/createNewCar
/deleteAllRedCars
```


再比如，某个 URI 是/posts/show/1，其中 show 是动词，这个 URI 就设计错了，正确的写法应该是/posts/1，然后用 GET 方法表示 show。
如果某些动作是 HTTP 动词表示不了的，你就应该把动作做成一种资源。比如网上汇款，从账户 1 向账户 2 汇款 500 元，错误的 URI 是：

```
POST /accounts/1/transfer/500/to/2
```

正确的写法是把动词 transfer 改成名词 transaction，资源不能是动词，但是可以是一种服务：

```
POST /transaction HTTP/1.1
Host: 127.0.0.1
from=1&to=2&amount=500.00
```

理清资源的层次结构，比如业务针对的范围是学校，那么学校会是一级资源(/school)，老师(/school/teachers)，学生(/school/students)就是二级资源。

3、版本（Versioning）

应该将 API 的版本号放入 URL。如：

```
https://api.example.com/v1/
```

另一种做法是，将版本号放在 HTTP 头信息中，但不如放入 URL 方便和直观。Github 采用这种做法。

4、过滤信息（Filtering）

如果记录数量很多，服务器不可能都将它们返回给用户。API 应该提供参数，过滤返回结果。
下面是一些常见的参数。

```
?limit=10: 指定返回记录的数量
?offset=10: 指定返回记录的开始位置。
?page_number=2&page_size=100: 指定第几页，以及每页的记录数。
?sortby=name&order=asc: 指定返回结果按照哪个属性排序，以及排序顺序。
?animal_type_id=1: 指定筛选条件
参数的设计允许存在冗余，即允许 API 路径和 URL 参数偶尔有重复。比如，
GET /zoo/ID/animals 与 GET /animals?zoo_id=ID 的含义是相同的。
```

5、状态码（Status Codes）

状态码范围

```
1xx 信息，请求收到，继续处理。范围保留用于底层 HTTP 的东西，你很可能永远也用不到。
2xx 成功，行为被成功地接受、理解和采纳
3xx 重定向，为了完成请求，必须进一步执行的动作
4xx 客户端错误，请求包含语法错误或者请求无法实现。范围保留用于响应客户端做出的错误，例如。他们提供不良数据或要求不存在的东西。这些请求应该是幂等的，而不是更改服务器的状态。
5xx 范围的状态码是保留给服务器端错误用的。这些错误常常是从底层的函数抛出来的，甚至
```

开发人员也通常没法处理，发送这类状态码的目的以确保客户端获得某种响应。
当收到 5xx 响应时，客户端不可能知道服务器的状态，所以这类状态码是要尽可能的避免。

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的 HTTP 动词）。

- 200 OK - [GET]: 服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
- 201 CREATED - [POST/PUT/PATCH]: 用户新建或修改数据成功。
- 202 Accepted - [*]: 表示一个请求已经进入后台排队（异步任务）
- 204 NO CONTENT - [DELETE]: 用户删除数据成功。
- 400 INVALID REQUEST - [POST/PUT/PATCH]: 用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
- 401 Unauthorized - [*]: 表示用户没有权限（令牌、用户名、密码错误）。
- 403 Forbidden - [*] 表示用户得到授权（与 401 错误相对），但是访问是被禁止的。
- 404 NOT FOUND - [*]: 用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
- 406 Not Acceptable - [GET]: 用户请求的格式不可得（比如用户请求 JSON 格式，但是只有 XML 格式）。
- 410 Gone -[GET]: 用户请求的资源被永久删除，且不会再得到的。
- 422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
- 500 INTERNAL SERVER ERROR - [*]: 服务器发生错误，用户将无法判断发出的请求是否成功。
- 502 网关错误
- 503 Service Unavailable
- 504 网关超时

错误处理（Error handling）

如果状态码是 4xx，就应该向用户返回出错信息。一般来说，返回的信息中将 **error** 作为键名，出错信息作为键值即可。

```
{
  error: "Invalid API key"
}
```

返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

- GET /collection: 返回资源对象的列表（数组）
- GET /collection/resource: 返回单个资源对象
- POST /collection: 返回新生成的资源对象
- PUT /collection/resource: 返回完整的资源对象
- PATCH /collection/resource: 返回完整的资源对象
- DELETE /collection/resource: 返回一个空文档

Hypermedia API

RESTful API 最好做到 Hypermedia，即返回结果中提供链接，连向其他 API 方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向 `api.example.com` 的根目录发出请求，会得到这样一个文档。

```
{
  "link": {
    "rel": "collection https://www.example.com/zoos",
    "href": "https://api.example.com/zoos",
    "title": "List of zoos",
    "type": "application/vnd.yourformat+json"
  }
}
```

上面代码表示，文档中有一个 `link` 属性，用户读取这个属性就知道下一步该调用什么 API 了。`rel` 表示这个 API 与当前网址的关系（`collection` 关系，并给出该 `collection` 的网址），`href` 表示 API 的路径，`title` 表示 API 的标题，`type` 表示返回类型。

Hypermedia API 的设计被称为 HATEOAS。Github 的 API 就是这种设计，访问 api.github.com 会得到一个所有可用 API 的网址列表。

```
{
  "current_user_url": "https://api.github.com/user",
  "authorizations_url": "https://api.github.com/authorizations",
  // ...
}
```

从上面可以看到，如果想获取当前用户的信息，应该去访问 api.github.com/user，然后就得到了下面结果。

```
{
  "message": "Requires authentication",
  "documentation_url": "https://developer.github.com/v3"
}
```

上面代码表示，服务器给出了提示信息，以及文档的网址。

问题 14：

Django 的路由层，视图层，模板层，模型层的工作流程

Django 作为 web 框架，想要了解其流程，就必须了解一下 web 服务器与 web 框架之间的关系

请求与响应过程

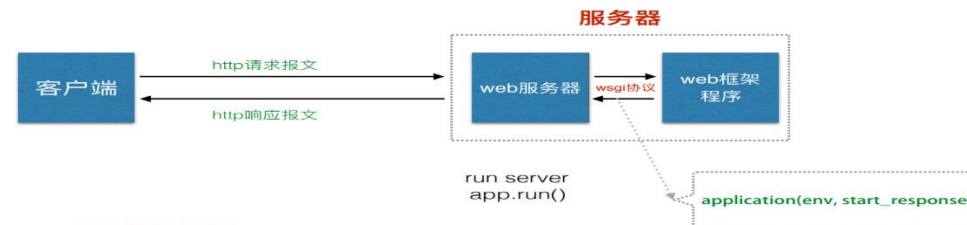
由客户端发起请求，服务器进行响应，请求与响应过程遵循 HTTP 协议



HTTP是客户端浏览器或其他程序与Web服务器之间的应用层通信协议。
HTTP是一个客户端和服务端请求和应答的标准。

web 服务器与 web 框架

当客户端发起请求时，web 服务器负责解析请求报文，调用 web 框架，然后再由 web 框架根据 url 找到对应处理函数，进行业务处理之后，由 web 服务器组织响应报文，返回内容给客户端



web服务器的作用:

1. 解析请求报文，调用框架程序处理请求。
2. 组织响应报文，返回内容给客户端。

web框架程序的作用:

1. 路由分发（根据url找到对应的处理函数）
2. 调用处理函数进行业务的处理。

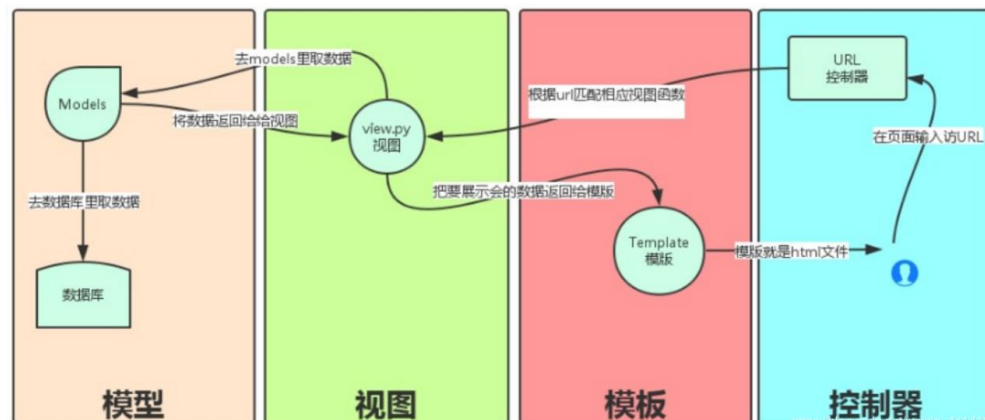
Django 流程介绍

Django 使用了 MVT 模式

M 全拼为 Model，负责和数据库交互，进行数据处理。

V 全拼为 View，接收请求，进行业务处理，返回应答。

T 全拼为 Template，负责封装构造要返回的 html。



用户通过浏览器请求页面

拿到用户的 url 会通过 urls.py 文件进行匹配，找到相应的 View（视图）

调用 View 中的函数

View 中的方法可以通过 Models 访问数据库数据，并将数据返回给 View

如果需要 views 可以使用 Context，context 被传递给 Template（模板）来生成 html 页面

返回响应对象到浏览器，给用户呈现效果

MVC 模式与 MVT 模式

MVC 模式

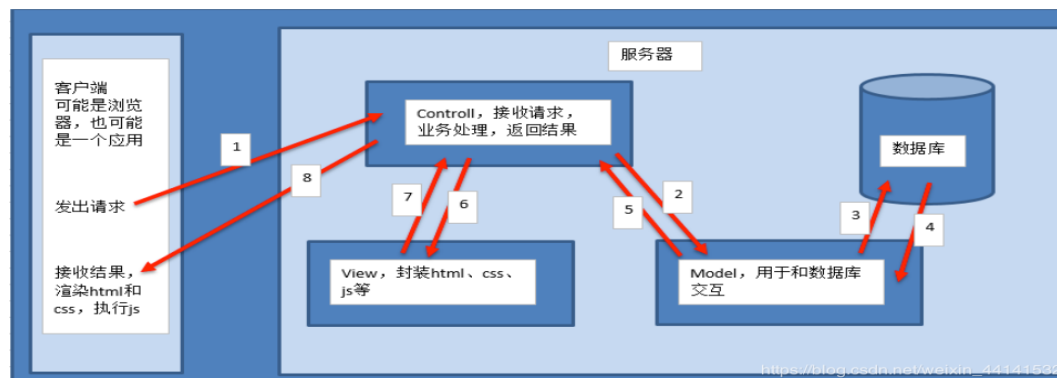
MVC 是一种程序设计模式，核心思想是分工、解耦，让不同的代码块之间降低耦合，增强代码的可扩展性和可移植性，实现向后兼容。

MVC 模式说明

M 全拼为 Model，主要封装对数据库层的访问，对数据库中的数据进行增、删、改、查操作。

V 全拼为 View，用于封装结果，生成页面展示的 html 内容。

C 全拼为 Controller，用于接收请求，处理业务逻辑，与 Model 和 View 交互，返回结果。



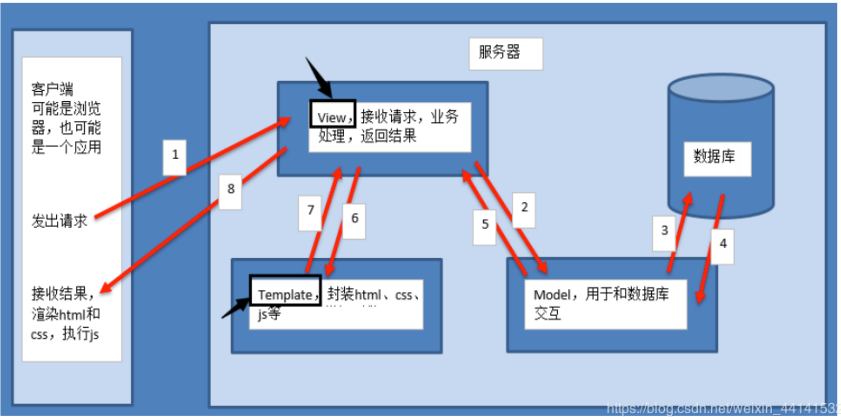
MVT 模式

MVT 模式中的'M', 与 MVC 中的 M 功能相同, 负责和数据库交互, 进行数据处理。

MVT 模式中的'V', 与 MVC 中的 C 功能相同, 接收请求, 进行业务处理, 返回应答。

MVT 模式中的'T', 与 MVC 中的 V 功能相同, 负责封装构造要返回的 html。

MVT 与 MVC 模式具体差异不是很大, 它们的思路是一样的



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 替你完成吧

问题 15:

Flask 的请求流程, 请求上下文 (栈实现)

上下文: 有两种

`RequestContext` 请求上下文: 内部有 `request` 和 `session` 对象

`AppContext` 程序上下文: 内部有 `g` 和 `current_app` 对象

第一步: 创建上下文

Flask 根据 `WSGI Server` 封装的请求等的信息(`environ`)新建 `RequestContext` 对象 和 `AppContext` 对象

第二步: 入栈

将 `RequestContext` 对象 `push` 进 `_request_ctx_stack` 里面。在这次请求期间, 访问 `request` 对象, `session` 对象将指向这个栈的栈顶元素

第三步: 请求分发

`response = self.full_dispatch_request()`

Flask 将调用 `full_dispatch_request` 函数进行请求的分发, 之所以不用给参数, 是因为我们可以通过 `request` 对象获得这次请求的信息。`full_dispatch_request` 将根据请求的 `url` 找到对应的蓝本里面的视图函数, 并生成一个 `response` 对象。注意的是, 在请求之外的时间, 访问 `request` 对象是无效的, 因为 `request` 对象依赖请求期间的 `_request_ctx_stack` 栈。

第四步:上下文对象出栈

这次 HTTP 的响应已经生成了，就不需要两个上下文对象了。分别将两个上下文对象出栈，为下一次的 HTTP 请求做出准备。

第五步：响应 WSGI

调用 Response 对象，向 WSGI Server 返回其结果作为 HTTP 正文。Response 对象是一个可调用对象，当调用发生时，将首先执行 WSGI 服务器传入的 start_response()函数 发送状态码和 HTTP 报文头。

问题 15：

Django ORM 跨表一次性查询

select_related 内部自动连表，连表的时候比较消耗资源，但走数据库的次数少

prefetch_related 内部不做连表，多次查询的时候比较消耗资源，但刚给用户的感觉和连表操作一样

问题 16：

Celery Broker 使用的什么，结果要保存吗，保存到哪里

Celery 介绍

Celery 是一个功能完备即插即用的任务队列。它使得我们不需要考虑复杂的问题，使用非常简单。celery 看起来似乎很庞大，我们先对其进行简单的了解，然后再去学习其他一些高级特性。 celery 适用异步处理问题，当发送邮件、或者文件上传，图像处理等等一些比较耗时的操作，我们可将其异步执行，这样用户不需要等待很久，提高用户体验。 celery 的特点是：

- 简单，易于使用和维护，有丰富的文档。
- 高效，单个 celery 进程每分钟可以处理数百万个任务。
- 灵活，celery 中几乎每个部分都可以自定义扩展。
- celery 非常易于集成到一些 web 开发框架中。

Celery 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(message broker)

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 等等

任务执行单元(worker)

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。

任务结果储存(task result store)

如果我们想跟踪任务的状态，Celery 需要将结果保存到某个地方。有几种保存的方案可选:SQLAlchemy、Django ORM、Memcached、 Redis、RPC (RabbitMQ/AMQP)。

使用场景

异步执行：解决耗时任务

延迟执行：解决延迟任务

定时执行：解决周期(周期)任务

Celery 的安装配置

```
pip install celery
```

消息中间件：RabbitMQ/Redis

```
app=Celery('任务名', broker='xxx', backend='xxx')
```

celery 框架工作流程

- 创建 celery 框架对象 app，配置 broker 和 backend，得到的 app 就是 worker
- 给 worker 对应的 app 添加可处理的任务函数，用 include 配置给 worker 的 app
- 完成提供的任务的定时配置 app.conf.beat_schedule
- 启动 celery 服务，运行 worker，执行任务
- 启动 beat 服务，运行 beat，添加任务

Celery 执行异步任务

包架构封装

```
project
├── celery_task    # celery 包
│   ├── __init__.py # 包文件
│   ├── celery.py   # celery 连接和配置相关文件，且名字必须交 celery.py
│   └── tasks.py    # 所有任务函数
├── add_task.py    # 添加任务
└── get_result.py  # 获取结果
```

使用

celery.py

- # 1) 创建 app + 任务
- # 2) 启动 celery(app) 服务：
- # 非 windows


```
# 命令: celery worker -A celery_task -l info
# windows:
# pip3 install eventlet
# celery worker -A celery_task -l info -P eventlet
# 3) 添加任务: 手动添加, 要自定义添加任务的脚本, 右键执行脚本
# 4) 获取结果: 手动获取, 要自定义获取任务的脚本, 右键执行脚本
```

Celery 执行延迟任务

Celery 执行定时任务

django 中使用

celery 配置 django 缓存

在项目根目录下先建一个包 celery_task, 包中先建两个文件 celery.py 和 tasks.py

问题 17:

uWSGI 的工作流程, 基于什么协议, 使用的什么配置 (回答的默认) (详细说明略)

nginx 配置文件

```
server {
    listen      80;
    server_name localhost;
    index index.html index.htm;
    client_max_body_size 35m;
    location / {
        include uwsgi_params;
        uwsgi_pass 127.0.0.1:9090;
        uwsgi_param UWSGI_SCRIPT demosite.wsgi;    #指定加载的模块
        uwsgi_param UWSGI_CHDIR /webser/www/demosite; #指定项目目录
    }
}
```

问题 18:

快速排序, 大量重复数据重复影响效率吗, 怎么处理 (会影响效率, 优化没回答上来)

会影响效率

解决办法：分组

三相切割快速排序：分三组

快速排序(Quick Sort)是一种有效的排序算法。虽然算法在最坏的情况下运行时间为 $O(n^2)$ ，但由于平均运行时间为 $O(n \log n)$ ，并且在内存使用、程序实现复杂性上表现优秀，尤其是对快速排序算法进行随机化的可能，使得快速排序在一般情况下是最实用的排序方法之一。

快速排序被认为是当前最优秀的内部排序方法。

冒泡排序的基本概念是：依次比较相邻的两个数，将大数放在前面，小数放在后面。即首先比较第 1 个和第 2 个数，将大数放前 636f70797a686964616f31333335336531，小数放后。然后比较第 2 个数和第 3 个数，将大数放前，小数放后，如此继续，直至比较最后两个数，将大数放前，小数放后，此时第一趟结束，在最后的数必是所有数中的最小数。重复以上过程，仍从第一对数开始比较（因为可能由于第 2 个数和第 3 个数的交换，使得第 1 个数不再大于第 2 个数），将大数放前，小数放后，一直比较到最小数前的一对相邻数，将大数放前，小数放后，第二趟结束，在倒数第二个数中得到一个新的最小数。如此下去，直至最终完成排序。

由于在排序过程中总是大数往前放，小数往后放，相当于气泡往上升，所以称作冒泡排序。

用二重循环实现，外循环变量设为 i ，内循环变量设为 j 。外循环重复 9 次，内循环依次重复 9, 8, ..., 1 次。每次进行比较的两个元素都是与内循环 j 有关的，它们可以分别用 $a[j]$ 和 $a[j+1]$ 标识， i 的值依次为 1,2,...,9，对于每一个 i, j 的值依次为 1,2,...10- i 。

问题 19：

Dockerfile 中 ADD 和 COPY 的区别，是用 ADD 还是 COPY

COPY 是从本地，推荐使用，从本地拷贝，不直接解压

ADD 是增强版 COPY，可以使用 url 拷贝，直接解压

问题 20：

JWT 校验后，是只能一个用户登录还是多用户可以同时在线(回答只能一个，分析了一下不能，追问怎么实现只能一个设备登录，没答上来)

实现一个用户登录：

将 token 存入 redis，每次登录就刷新 redis 里的 token，这样就能保证 token 永远只有一个

jwt 的主要认证方式有以下四种：

基于django-rest-framework的JWT的登陆与认证流程

JWT(Json web token)的介绍 分析

基于django-rest-framework的登陆认证方式常用的大体可分为四种：

1. BasicAuthentication：账号密码登陆验证
2. SessionAuthentication：基于session机制会话验证
3. TokenAuthentication：基于令牌的验证
4. JSONWebTokenAuthentication：基于Json-Web-Token的验证

其中最常用的是JWT(Json-Web-Token),如果对JWT不理解的推荐一篇文章：[前后端分离之JWT用户认证](#)

JWT具有以下优点：

1. 签名的方式验证用户信息，安全性较之一般的认证高
2. 加密后的字符串保存于客户端浏览器中，减少服务器存储压力
3. 签名字符串中存储了用户部分的非私密信息，一定程度上能够减少服务器数据库查询用户信息的开销

缺点：

1. 采用对称加密，一旦被恶意用户获取到加密方法，就可以不断破解入侵获取信息，不过基本加密方法很难被破解
2. 加大了服务器的计算开销，—不过相对于磁盘开销，这都不算啥

总的来说，JWT没多少去缺点，所以很多公司都用这个业务做用户认证，当然，涉及到金钱的，还是选择非对称加密方式比较好。[https://www.cnblogs.com/5987](#)

使用 APIView 时，如果你的登录验证使用的是 ObtainJSONWebToken，那么你的登录验证就要加上：authentication_classes =

[JSONWebTokenAuthentication]

```
class UserLoginView(ObtainJSONWebToken):  
    """用户登录"""  
    def post(self, request, *args, **kwargs):  
        response = super().post(request, *args, **kwargs)
```

```
# 修改密码
class UpdatePasswordView(APIView):
    """修改密码"""
    authentication_classes = [JSONWebTokenAuthentication]

    def put(self, request):
        # 获取用户
        data = request.data
        # 验证数据
        ser = ChangePasswordSerializer(data=data)
        ser.is_valid()
```

https://blog.csdn.net/weixin_43745987

所以最重要的是先要弄清楚这四种方式的区别与使用方式。

jwt 的规范目前只检测 jwt 的发布者,过期时间,签名等信息.大部分现成的库都是按照标准写的.但是标准没有要求 jwt 带入登录时间等信息,因此用户连续登录多次,后台返回的 token 在有效期内都能访问后台 api.也就是用户可以在多个设备同时登录.

有人会采用在登录时将 jwt 用 redis 等储存起来,在 api 的中间件检查时去查数据库中是否储存了这个 token,设置过期时间.如果用户再次登录,那么将该 token 用新生成的 token 替代.

我们也可以不新建一个表,因为我们多数应用的用户表中都会储存用户的最后登录时间,最后登录 ip 等信息,我们可以在登录时将这些信息一边储存到数据库中,一边注入到 jwt 中.然后在用户访问 api 时在中间件中从中间件取出这些信息,和用户表中的信息进行对比,如果不符合,就不允许用户访问 api.

初次尝试 JWT, 发现一个账号生成多个 JWT token 后, 只要在这些 token 还没过期前都是有效的

例如账号 A 在 15:30 分登录生成了 token0, 有效时间为 1 小时, 过了 5 分钟账号 A 再次登录生成了 token1 有效时间还是为 1 小时。这时候 token0 和 token1 都是合法的? 我之前以为 token1 会把 token0 给刷新掉……

那么如何实现一个账号只能同时在一个设备(端)登录呢? 想到了如下几种方式:

- 1.将 token 存入 redis, 每次登录就刷新 redis 里的 token, 这样就能保证 token 永远只有一个(虽然实际上还是有多个, 但是我只认 redis 里的这一个)。但是这样的话我就不需要 JWT 了啊, 直接 redis 实现就好了
- 2.结合 session, 每次登录将 session 写入 token, 然后判断当前 session 和 token 里的是否一致。同上面, 这样是不是有点冗余了, 直接用 session 实现就好了?
- 3.将用户最后一次登录信息写入 token, 判断 token 里的登录信息和数据库里最后一次的登录信息是否一致。这样做的话会频繁查询数据库, 数据库负载会变高

问题 21:

logging 模块, 日志过大怎么办(没答上)

很多程序都有记录日志的需求, 并且日志中包含的信息即有正常的程序访问日志, 还可能有错误、警告等信息输出, python 的 logging 模块提供了标准的日志接口, 你可以通过它存储各种格式的日志, logging 的日志可以分为 debug(), info(), warning(), error() and critical() 5 个级别, 下面我们看一下怎么用。

级别排序:CRITICAL > ERROR > WARNING > INFO > DEBUG

debug: 打印全部的日志,详细的信息,通常只出现在诊断问题上

info: 打印 info,warning,error,critical 级别的日志,确认一切按预期运行

warning: 打印 warning,error,critical 级别的日志,一个迹象表明,一些意想不到的事情发生了,或表明一些问题在不久的将来(例如.磁盘空间低”),这个软件还能按预期工作

error: 打印 error,critical 级别的日志,更严重的问题,软件没能执行一些功能

critical：打印 critical 级别,一个严重的错误,这表明程序本身可能无法继续运行

2、部分名词解释

Logging.Formatter: 这个类配置了日志的格式，在里面自定义设置日期和时间，输出日志的时候将会按照设置的格式显示内容。

Logging.Logger: Logger 是 Logging 模块的主体，进行以下三项工作：

- 1. 为程序提供记录日志的接口
- 2. 判断日志所处级别，并判断是否要过滤
- 3. 根据其日志级别将该条日志分发给不同 handler

常用函数有：

Logger.setLevel() 设置日志级别

Logger.addHandler() 和 Logger.removeHandler() 添加和删除一个 Handler

Logger.addFilter() 添加一个 Filter,过滤作用

Logging.Handler: Handler 基于日志级别对日志进行分发，如设置为 WARNING 级别的 Handler 只会处理 WARNING 及以上级别的日志。

常用函数有：

setLevel() 设置级别

setFormatter() 设置 Formatter

1、日志级别

Python 标准库 logging 用作记录日志，默认分为六种日志级别（括号为级别对应的数值），NOTSET（0）、DEBUG（10）、INFO（20）、WARNING（30）、ERROR（40）、CRITICAL（50）。我们自定义日志级别时注意不要和默认的日志级别数值相同，logging 执行时输出大于等于设置的日志级别的日志信息，如设置日志级别是 INFO，则 INFO、WARNING、ERROR、CRITICAL 级别的日志都会输出。

2、logging 流程

官方的 logging 模块工作流程图如下：

从下图中我们可以看出看到这几种 Python 类型，Logger、LogRecord、Filter、Handler、Formatter。

类型说明：

Logger: 日志，暴露函数给应用程序，基于日志记录器和过滤器级别决定哪些日志有效。

LogRecord：日志记录器，将日志传到相应的处理器处理。

Handler：处理器，将(日志记录器产生的)日志记录发送至合适的目的地。

Filter：过滤器，提供了更好的粒度控制,它可以决定输出哪些日志记录。

Formatter: 格式化器，指明了最终输出中日志记录的布局。

logging 流程图.png

- 1. 判断 Logger 对象对于设置的级别是否可用，如果可用，则往下执行，否则，流程结束。
- 2. 创建 LogRecord 对象，如果注册到 Logger 对象中的 Filter 对象过滤后返回 False，则不记录日志，流程结束，否则，则向下执行。
- 3. LogRecord 对象将 Handler 对象传入当前的 Logger 对象，（图中的子流程）如果 Handler 对象的日志级别大于设置的日志级别，再判断注册到 Handler 对象中的 Filter 对象过滤后是否返回 True 而放行输出日志信息，否则不放行，流程结束。
- 4. 如果传入的 Handler 大于 Logger 中设置的级别，也即 Handler 有效，则往下执行，否则，流程结束。
- 5. 判断这个 Logger 对象是否还有父 Logger 对象，如果没有（代表当前 Logger 对象是最顶层的 Logger 对象 root Logger），流程结束。否则将 Logger 对象设置为它的父 Logger 对象，重复上面的 3、4 两步，输出父类 Logger 对象中的日志输出，直到是 root Logger 为止。

3、日志输出格式

日志的输出格式可以认为设置，默认格式为下图所示（略）。

4、基本使用

logging 使用非常简单，使用 `basicConfig()` 方法就能满足基本的使用需要，如果方法没有传入参数，会根据默认的配置创建 `Logger` 对象，默认的日志级别被设置为 **WARNING**，默认的日志输出格式如上图，该函数可选的参数如下表所示。

参数名称	参数描述
filename	日志输出到文件的文件名
filemode	文件模式，r[+]、w[+]、a[+]
format	日志输出的格式
datefat	日志附带日期时间的格式
style	格式占位符，默认为 "%" 和 "{}"
level	设置日志输出级别
stream	定义输出流，用来初始化 StreamHandler 对象，不能 filename 参数一起使用，否则会ValueError 异常
handles	定义处理器，用来创建 Handler 对象，不能和 filename 、 stream 参数一起使用，否则也会抛出 ValueError 异常

5、自定义 Logger

上面的基本使用可以让我们快速上手 logging 模块，但一般并不能满足实际使用，我们还需要自定义 Logger。

6、Logger 配置

通过上面的例子，我们知道创建一个 Logger 对象所需的配置了，上面直接硬编码在程序中配置对象，配置还可以从字典类型的对象和配置文件获取。打开 `logging.config Python` 文件，可以看到其中的配置解析转换函数。

7、实战中的问题：1、中文乱码，2、临时禁用日志输出，3、日志文件按照时间划分或者按照大小划分：

如果将日志保存在一个文件中，那么时间一长，或者日志一多，单个日志文件就会很大，既不利于备份，也不利于查看。我们会想到能不能按照时间或者大小对日志文件进行划分呢？答案肯定是可以的，并且还很简单，logging 考虑到了我们这个需求。logging.handlers 文件中提供了 **TimedRotatingFileHandler** 和 **RotatingFileHandler** 类分别可以实现按时间和大小划分。打开这个 `handles` 文件，可以看到还有其他功能的 `Handler` 类，它们都继承自基类 **BaseRotatingHandler**。

Python 官网虽然说 logging 库是线程安全的，但在多进程、多线程、多进程多线程环境中仍然还有值得考虑的问题，比如，如何将日志按照进程（或线程）划分为不同的日志文件，也即一个进程（或线程）对应一个文件。由于本文篇幅有限，故不在这里做详细说明，只是起到引发读者思考的目的，这些问题我会在另一篇文章中讨论。

总结：Python logging 库设计的真的非常灵活，如果有特殊的需要还可以在这个基础的 logging 库上进行改进，创建新的 `Handler` 类解决实际开发中的问题。

问题 22：

python 的元类，Django ORM 底层用到元类(没答好)

元类其实就是产生类的类，我们可以通过元类来拦截类的创建过程，这个地方我自己通过元类写过一个简易版本的 ORM

首先 ORM 全称叫对象关系映射，能够让不会数据库操作的程序员通过面向对象的方法简单快捷的操作数据库，ORM 有三层映射关系

* 类映射数据库的表

* 对象映射成数据库的表中的一条条记录

* 对象获取属性映射成数据库的表中的某条记录某个字段对应的值

具体做法就是在类创建过程中通过元类拦截它的创建，在类创建出来之前给类赋上表该有的属性表名，主键字段，其他普通字段

问题 23:

MySQL 事务+悲观锁，直接用 MySQL 不会崩吗

mysql 悲观锁，高并发

1.高并发的时候有 2 种处理

1) 后端进行线程安全处理，synchrnoized,还有其他不同粒度的锁

2) 在数据库设置锁，当你读的时候，不允许其他人修改。可以用 mysql 的悲观锁

2.悲观锁

select * from 表名 for update

for update 很重要，就是如果你查询这个事务没有结束前，别人不能去修改它的内容。

3. 那怎么自己实践呢？

打开 2 个 mysql 命令行

一个 mysql: 然后 use 数据库名，输入 begin;就是开始一个事务，然后 select * from 表名 for update;不要打 commit;就是提交事务。

另一个: 然后 use 数据库名，然后 select 的语句是没问题的，但是当你你要 update 数据的时候是没办法的，它会等待。只有等第一个事务 commit 之后，才能进行修改。

数据库管理系统中并发控制的任务是确保在多个事务同时存取数据库中同一数据不破坏事务的隔离性和统一性以及数据库的统一性

乐观锁和悲观锁式并发控制主要采用的技术手段

悲观锁

在关系数据库管理系统中，悲观并发控制（悲观锁，PCC）是一种并发控制的方法。它可以阻止一个事务以影响其他用户的方式来修改数据。如果一个事务执行的操作的每行数据应用了锁，那只有当这个事务锁

释放，其他事务才能够执行与该锁冲突的操作

悲观并发控制主要应用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本环境

悲观锁，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度（悲观），因此在整个暑假处理过程中，将数据处于锁定状态。悲观锁的实现，一般依靠数据库提

供的锁机制（推荐教程：MySQL 教程）

数据库中，悲观锁的流程如下

- 在对任何记录进行修改之前，先尝试为该记录加上排他锁
- 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或抛出异常
- 如果成功加锁，则就可以对记录做修改，事务完成后就会解锁
- 其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常

MySQL InnoDB 中使用悲观锁

要使用悲观锁，必须关闭 mysql 数据库的自动提交属性，因为 MySQL 默认使用 autocommit 模式，也就是当你执行一个更新操作后，MySQL 会立即将结果进行提交

```
//开始事务

5  begin;/begin work;/start transaction;（三者选一个）
7
9  select status from t_goods where id=1 for update;
10
11 //根据商品信息生成订单
12   insert into t_orders (id,goods_id) values (null,1);
13   //修改商品 status 为 2
14
15   update t_goods set status=2;
16
17   // 提交事务

   commit;/commit work;
```

以上查询语句中，使用了 **select...for update** 方式，通过开启排他锁的方式实现了悲观锁。则相应的记录被锁定，其他事务必须等本次事务提交之后才能够执行。我们使用 **select ... for update** 会把数据给锁定，不过我们需要注意一些锁的级别，MySQL InnoDB 默认行级锁。行级锁都是基于索引的，如果一条 SQL 用不到索引是不会使用行级锁的，会使用表级锁把整张表锁住。

特点

为数据处理的安全提供了保证
效率上，由于处理加锁的机制会让数据库产生额外开销，增加产生死锁机会
在只读型事务中由于不会产生冲突，也没必要使用锁，这样会增加系统负载，降低并行性

乐观锁

乐观并发控制也是一种并发控制的方法。
假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的那部分数据，在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没其他事务修改该数据，如果有则回滚正在提交的事务
乐观锁相对悲观锁而言，是假设数据不会发生冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误信息，让用户决定如何做
乐观锁实现一般使用记录版本号，为数据增加一个版本标识，当更新数据的时候对版本标识进行更新

实现

使用版本号时，可以在数据初始化时指定一个版本号，每次对数据的更新操作都对版本号执行+1 操作。并判断当前版本号是不是该数据的最新版本号

```
1.查询出商品信息
select (status,status,version) from t_goods where id=#{id}

2.根据商品信息生成订单

3.修改商品 status 为 2
update t_goods
```



```
set status=2,version=version+1
where id=#{id} and version=#{version};
```

特点

乐观并发控制相信事务之间的数据竞争概率是较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁

什么是悲观锁

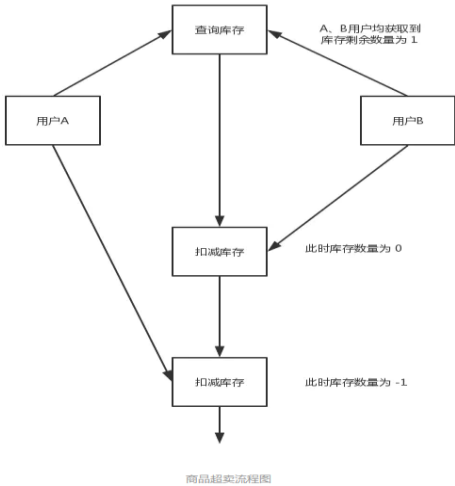
在[关系数据库管理系统](#)里，**悲观并发控制**（又名“**悲观锁**”，Pessimistic Concurrency Control，缩写“PCC”）是一种并发控制的方法。它可以阻止一个事务以影响其他用户的方式来修改数据。如果一个事务执行的操作读某行数据应用了锁，那只有当这个事务把锁释放，其他事务才能够执行与该锁冲突的操作。

悲观并发控制主要用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本的环境中。

简而言之，悲观锁主要用于保护数据的完整性。当多个事务并发执行时，某个事务对数据应用了锁，则其他事务只能等该事务执行完了，才能进行对该数据进行修改操作。

使用场景

在商品购买场景中，当有多个用户对某个库存有限的商品同时进行下单操作。若采用先查询库存，后减库存的方式进行库存数量的变更，将会导致超卖的产生。



商品超卖流程图

若使用悲观锁，当B用户获取到某个商品的库存数据时，用户A则会阻塞，直到B用户完成减库存的整个事务时，A用户才可以获取到商品的库存数据。则可以避免商品被超卖。

如何使用悲观锁

用法：SELECT ... FOR UPDATE；

例如，

```
select * from tbl_user where id=1 for update;
```

获取锁的前提：结果集中的数据没有使用排他锁或共享锁时，才能获取锁，否则将会阻塞。

需要注意的是，FOR UPDATE 生效需要同时满足两个条件时才生效：

- 数据库的引擎为 innodb
- 操作位于事务块中（BEGIN/COMMIT）

体验悲观锁

Step 1 初始化表结构和数据

```
CREATE TABLE `tbl_user` (  
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
  `status` int(11) DEFAULT NULL,  
  `name` varchar(255) COLLATE utf8_bin DEFAULT NULL,  
  PRIMARY KEY (`id`)  
);  
  
INSERT INTO `tbl_user` (`id`, `status`, `name`)  
VALUES  
  (1,1,X'7469616E'),  
  (2,1,X'63697479');
```

Step 2

窗口 1

// 关闭 mysql 数据库的自动提交属性

```
set autocommit=0;
```

// 开启事务

```
BEGIN;
```

```
SELECT * FROM tbl_user where id=1 for update;
```

窗口 2

此时，我们在窗口 2 执行下面这条命令，尝试获取悲观锁：

```
SELECT * FROM tbl_user where id=1 for update;
```

执行完后，窗口 2 并没有像窗口 1 一样，立刻返回结果，而是发生了阻塞。

若超时间未获取锁，将会得到一个锁超时错误提示。如下图所示：

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tbl_user where id=1 for update;
+-----+
| id | status | name |
+-----+
| 1 | 1 | tian |
+-----+
1 row in set (0.00 sec)

mysql> 
```

mysql (mysql)

```
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13
Server version: 5.6.41 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use tiancity
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM tbl_user where id=1 for update;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
mysql> 
```

获取锁超时

行锁与表锁

当执行 `select ... for update` 时，将会把数据锁住，因此，我们需要注意一下锁的级别。MySQL InnoDB 默认为行级锁。当查询语句指定了主键时，MySQL 会执行「行级锁」，否则 MySQL 会执行「表锁」。

常见情况如下：

- 若明确指明主键，且结果集有数据，行锁；
- 若明确指明主键，结果集无数据，则无锁；
- 若无主键，且非主键字段无索引，则表锁；
- 若使用主键但主键不明确，则使用表锁；

`select * from tbl_user where id<>1 for update;`

若需要了解更多情况，可以阅读 [这篇文章](#) 了解更多。

小结： InnoDB 的行锁是通过给索引上的索引项加锁实现的，因此，只有通过索引检索数据，才会采用行锁，否则使用的是表锁。

总结

悲观锁采用的是「先获取锁再访问」的策略，来保障数据的安全。但是加锁策略，依赖数据库实现，会增加数据库的负担，且会增加死锁的发生几率。此外，对于不会发生变化的只读数据，加锁只会增加额外不必要的负担。在实际的实践中，对于并发很高的场景并不会使用悲观锁，因为当一个事务锁住了数据，那么其他事务都会发生阻塞，会导致大量的事务发生积压拖垮整个系统。

问题 24：

什么是高可用

一、什么是高可用

高可用 HA（High Availability）是分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。

假设系统一直能够提供服务，我们说系统的可用性是 100%。

如果系统每运行 100 个时间单位，会有 1 个时间单位无法提供服务，我们说系统的可用性是 99%。

很多公司的高可用目标是 4 个 9，也就是 99.99%，这就意味着，系统的年停机时间为 8.76 个小时。

百度的搜索首页，是业内公认高可用保障非常出色的系统，甚至人们会通过 www.baidu.com 能不能访问来判断“网络的连通性”，百度高可用的服务让人留下啦“网络通畅，百度就能访问”，“百度打不开，应该是网络连不上”的印象，这其实是对百度 HA 最高的褒奖。

二、如何保障系统的高可用

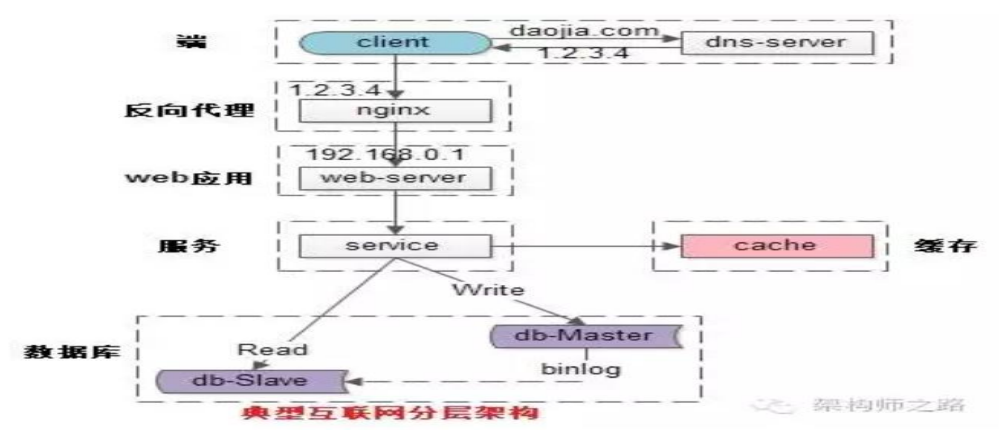
我们都知道，单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量在系统设计的过程中避免单点。方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了服务会受影响；如果有冗余备份，挂了还有其他 backup 能够顶上。

保证系统高可用，架构设计的核心准则是：冗余。

有了冗余之后，还不够，每次出现故障需要人工介入恢复势必会增加系统的不可服务实践。所以，又往往是通过“自动故障转移”来实现系统的高可用。

接下来我们看下典型互联网架构中，如何通过冗余+自动故障转移来保证系统的高可用特性。

三、常见的互联网分层架构



常见互联网分布式架构如上，分为：

- (1) 客户端层：典型调用方是浏览器 browser 或者手机应用 APP
- (2) 反向代理层：系统入口，反向代理
- (3) 站点应用层：实现核心应用逻辑，返回 html 或者 json
- (4) 服务层：如果实现了服务化，就有这一层
- (5) 数据-缓存层：缓存加速访问存储
- (6) 数据-数据库层：数据库固化数据存储

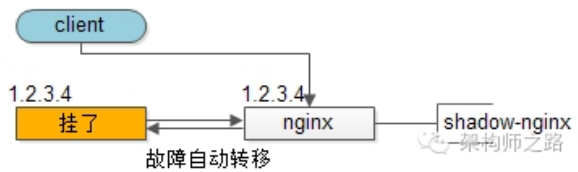
整个系统的高可用，又是通过每一层的冗余+自动故障转移来综合实现的。

四、分层高可用架构实践

【客户端层->反向代理层】的高可用



【客户端层】到【反向代理层】的高可用，是通过反向代理层的冗余来实现的。以 nginx 为例：有两台 nginx，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是 keepalived 存活探测，相同 virtual IP 提供服务。

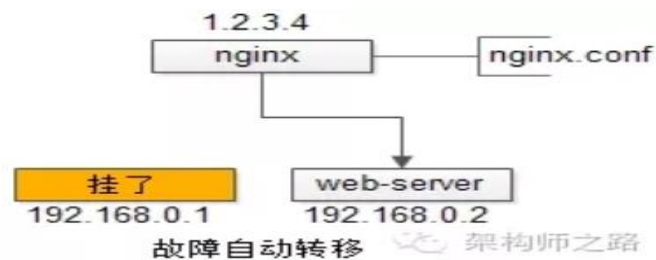


自动故障转移：当 nginx 挂了的时候，keepalived 能够探测到，会自动的进行故障转移，将流量自动迁移到 shadow-nginx，由于使用的是相同的 virtual IP，这个切换过程对调用方是透明的。

【反向代理层->站点层】的高可用

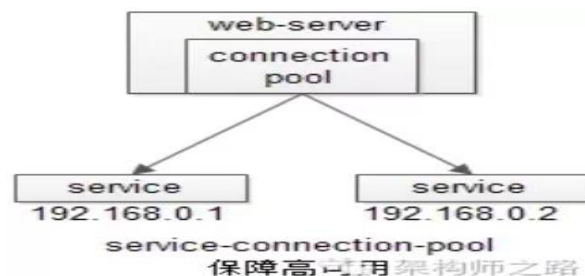


【反向代理层】到【站点层】的高可用，是通过站点层的冗余来实现的。假设反向代理层是 nginx，nginx.conf 里能够配置多个 web 后端，并且 nginx 能够探测到多个后端的存活性。

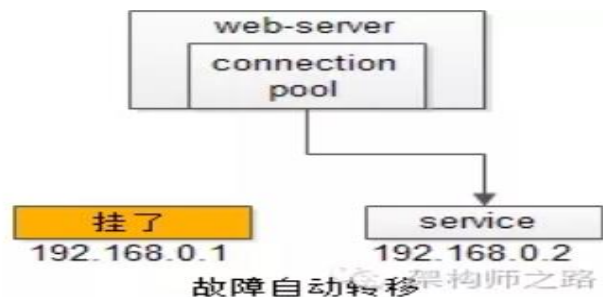


自动故障转移：当 web-server 挂了的时候，nginx 能够探测到，会自动的进行故障转移，将流量自动迁移到其他的 web-server，整个过程由 nginx 自动完成，对调用方是透明的。

【站点层->服务层】的高可用

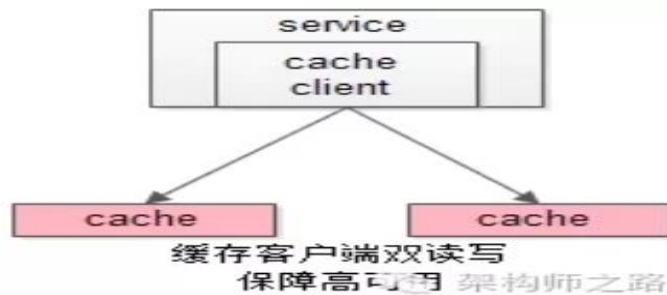


【站点层】到【服务层】的高可用，是通过服务层的冗余来实现的。“服务连接池”会建立与下游服务多个连接，每次请求会“随机”选取连接来访问下游服务。



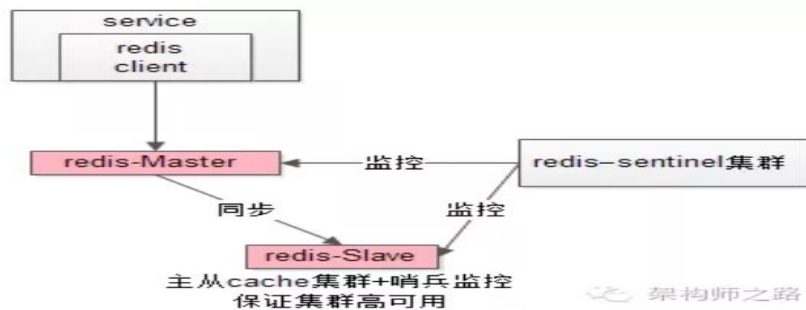
自动故障转移：当 service 挂了的时候，service-connection-pool 能够探测到，会自动的进行故障转移，将流量自动迁移到其他的 service，整个过程由连接池自动完成，对调用方是透明的（所以说 RPC-client 中的服务连接池是很重要的基础组件）。

【服务层>缓存层】的高可用



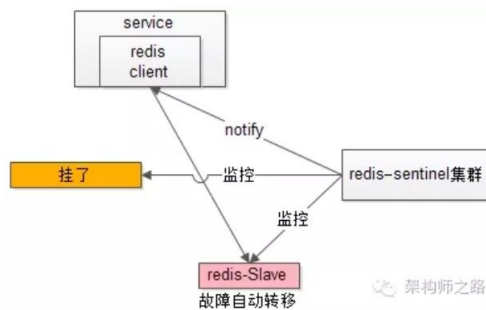
【服务层】到【缓存层】的高可用，是通过缓存数据的冗余来实现的。

缓存层的数据冗余又有几种方式：第一种是利用客户端的封装，service 对 cache 进行双读或者双写。



缓存层也可以通过支持主从同步的缓存集群来解决缓存层的高可用问题。

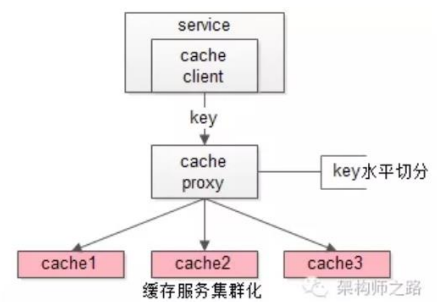
以 redis 为例，redis 天然支持主从同步，redis 官方也有 sentinel 哨兵机制，来做 redis 的存活性检测。



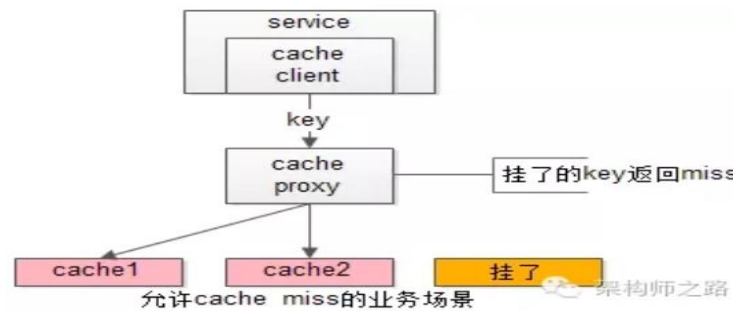
自动故障转移：当 redis 主挂了的时候，sentinel 能够探测到，会通知调用方访问新的 redis，整个过程由 sentinel 和 redis 集群配合完成，对调用方是透明的。

说完缓存的高可用，这里要多说一句，业务对缓存并不一定有“高可用”要求，更多的对缓存的使用场景，是用来“加速数据访问”：把一部分数据放到缓存里，如果缓存挂了或者缓存没有命中，是可以去后端的数据库中再取数据的。

这类允许“cache miss”的业务场景，缓存架构的建议是：



将 kv 缓存封装成服务集群，上游设置一个代理（代理可以用集群冗余的方式保证高可用），代理的后端根据缓存访问的 key 水平切分成若干个实例，每个实例的访问并不做高可用。

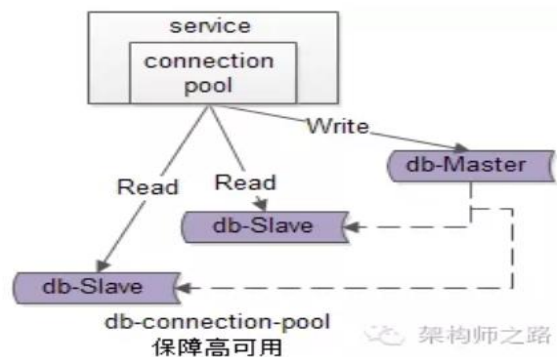


缓存实例挂了屏蔽：当有水平切分的实例挂掉时，代理层直接返回 cache miss，此时缓存挂掉对调用方也是透明的。key 水平切分实例减少，不建议做 re-hash，这样容易引发缓存数据的不一致。

【服务层>数据库层】的高可用

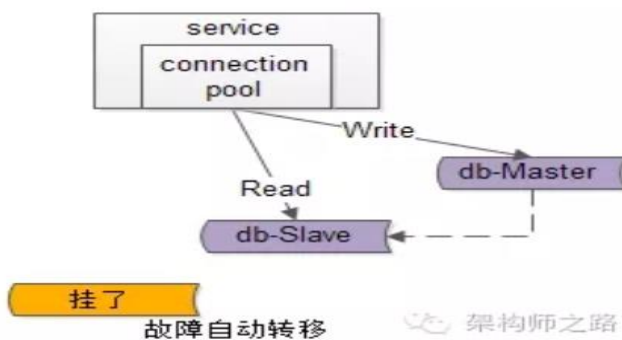
大部分互联网技术，数据库层都用了“主从同步，读写分离”架构，所以数据库层的高可用，又分为“读库高可用”与“写库高可用”两类。

【服务层>数据库层“读”】的高可用



【服务层】到【数据库读】的高可用，是通过读库的冗余来实现的。

既然冗余了读库，一般来说就至少有 2 个从库，“数据库连接池”会建立与读库多个连接，每次请求会路由到这些读库。



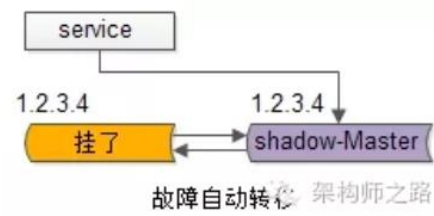
自动故障转移：当读库挂了的时候，db-connection-pool 能够探测到，会自动的进行故障转移，将流量自动迁移到其他的读库，整个过程由连接池自动完成，对调用方是透明的（所以说 DAO 中的数据库连接池是很重要的基础组件）。

【服务层>数据库层“写”】的高可用



【服务层】到【数据库写】的高可用，是通过写库的冗余来实现的。

以 mysql 为例，可以设置两个 mysql 双主同步，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是 keepalived 存活探测，相同 virtual IP 提供服务。



自动故障转移：当写库挂了的时候，keepalived 能够探测到，会自动的进行故障转移，将流量自动迁移到 shadow-db-master，由于使用的是相同的 virtual IP，这个切换过程对调用方是透明的。

五、总结

高可用 HA（High Availability）是分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。

方法论上，高可用是通过冗余+自动故障转移来实现的。

整个互联网分层系统架构的高可用，又是通过每一层的冗余+自动故障转移来综合实现的，具体的：

- （1）【客户端层】到【反向代理层】的高可用，是通过反向代理层的冗余实现的，常见实践是 keepalived + virtual IP 自动故障转移
- （2）【反向代理层】到【站点层】的高可用，是通过站点层的冗余实现的，常见实践是 nginx 与 web-server 之间的存活性探测与自动故障转移
- （3）【站点层】到【服务层】的高可用，是通过服务层的冗余实现的，常见实践是通过 service-connection-pool 来保证自动故障转移
- （4）【服务层】到【缓存层】的高可用，是通过缓存数据的冗余实现的，常见实践是缓存客户端双读双写，或者利用缓存集群的主从数据同步与 sentinel 保活与自动故障转移；更多的业务场景，对缓存没有高可用要求，可以使用缓存服务化来对调用方屏蔽底层复杂性
- （5）【服务层】到【数据库“读”】的高可用，是通过读库的冗余实现的，常见实践是通过 db-connection-pool 来保证自动故障转移
- （6）【服务层】到【数据库“写”】的高可用，是通过写库的冗余实现的，常见实践是 keepalived + virtual IP 自动故障转移

[软件架构篇]浅谈高可用

1、什么是高可用

高可用指系统的可用程度。没有 100%的可用性。打个夸张的比方说，部署在全球的所有机房都同时停电了，那么系统就不能再提供服务。一般我们只需要做到 4 个 9 就已经很不错了，如下图：

可用性	一年中可故障时长	一天中可故障时长
90%	36.5天	144分钟
99%	3.6天	14.4分钟
99.9%	8.8小时	86.4秒
99.99%	52.6分钟	8.6秒
99.999%	5.3分钟	860毫秒
99.9999%	31.5秒	86毫秒

image.png

2、高可用分类

按照业务=逻辑+数据来分，高可用分为计算高可用和存储高可用，逻辑即数据，数据即存储。

2.1 计算高可用

常见的计算高可用架构分为主备、主从、对称集群、非对称集群。

主备：

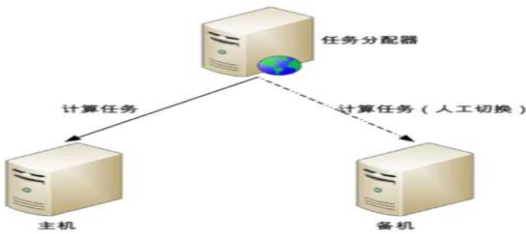


image.png

主从：

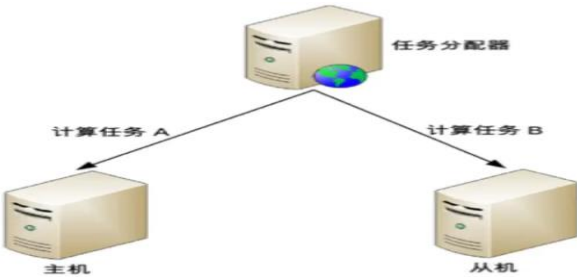


image.png

对称集群：

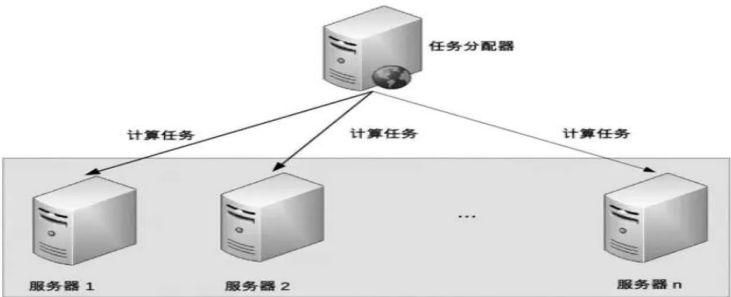


image.png

非对称集群：

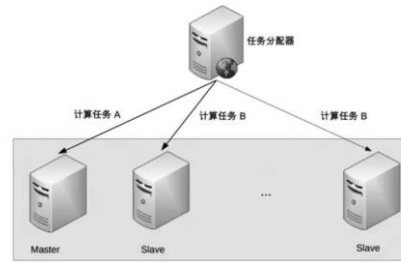


image.png

2.2 存储高可用

常见的存储高可用有主备、主从、主备/主从切换，主主，集群（数据集中集群和数据分散集群），分区（洲际、国家、城市、同城分区）。

主备：

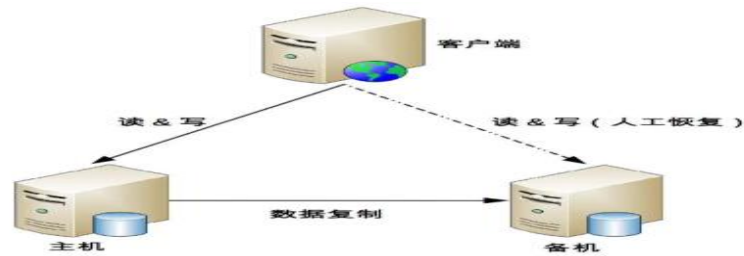


image.png

主从：

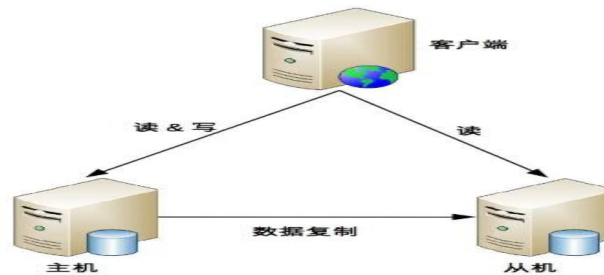


image.png

数据集中式集群：

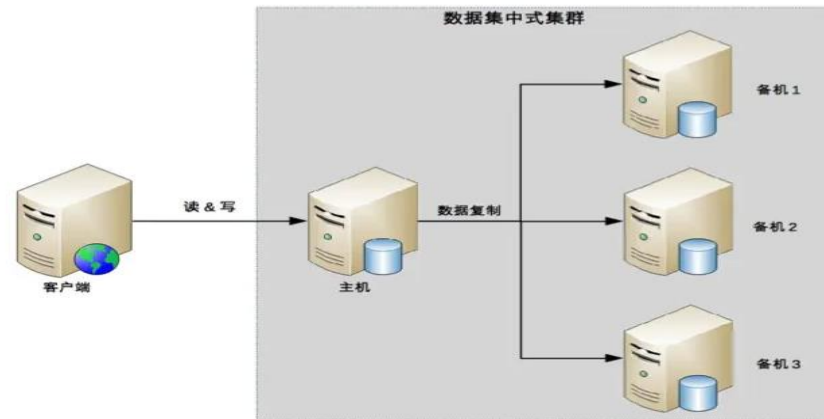


image.png

主主集群:

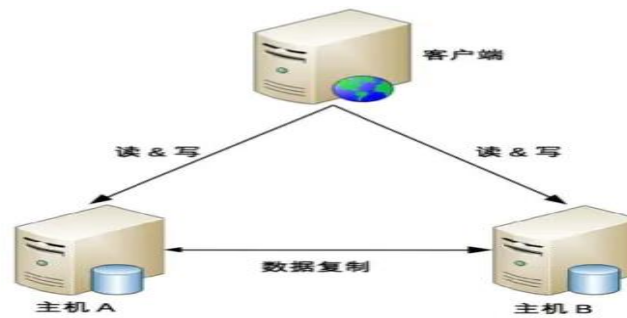
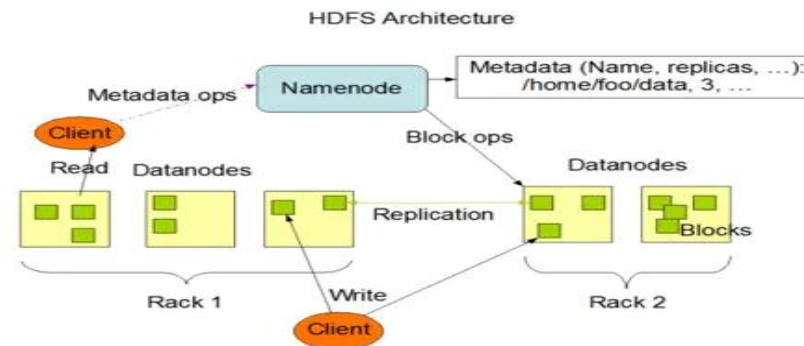


image.png

数据分散集群:

比如 HDFS 的架构。



问题 25:

集群、分布式、微服务的区别

概念:

集群是个物理形态，分布式是个工作方式。

1. 分布式：一个业务分拆多个子业务，部署在不同的服务器上

2. 集群：同一个业务，部署在多个服务器上

分布式是指将不同的业务分布在不同的地方。而集群指的是将几台服务器集中在一起，实现同一业务。

分布式中的每一个节点，都可以做集群。而集群并不一定就是分布式的。

举例：比如如新浪网，访问的人多了，他可以做一个集群，前面放一个响应服务器，后面几台服务器完成同一业务，如果有业务访问的时候，响应服务器看哪台服务器的负载不是很重，就将给哪一台去完成。

而分布式，从窄意上理解，也跟集群差不多，但是它的组织比较松散，不像集群，有一个组织性，一台服务器垮了，其它的服务器可以顶上来。

分布式的每一个节点，都完成不同的业务，一个节点垮了，那这个业务就不可访问了。

简单说，分布式是以缩短单个任务的执行时间来提升效率的，而集群则是通过提高单位时间内执行的任务数来提升效率。

例如：如果一个任务由 10 个子任务组成，每个子任务单独执行需 1 小时，则在一台服务器上执行该任务需 10 小时。

采用分布式方案，提供 10 台服务器，每台服务器只负责处理一个子任务，不考虑子任务间的依赖关系，执行完这个任务只需一个小时。（这种工作模式的一个典型代表就是 Hadoop 的 Map/Reduce 分布式计算模型）

而采用集群方案，同样提供 10 台服务器，每台服务器都能独立处理这个任务。假设有 10 个任务同时到达，10 个服务器将同时工作，1 小时后，10 个任务同时完成，这样，整体来看，还是 1 小时内完成一个任务！

好的设计应该是分布式和集群的结合，先分布式再集群，具体实现就是业务拆分成很多子业务，然后针对每个子业务进行集群部署，这样每个子业务如果出了问题，整个系统完全不会受影响。

另外，还有一个概念和分布式比较相似，那就是微服务。

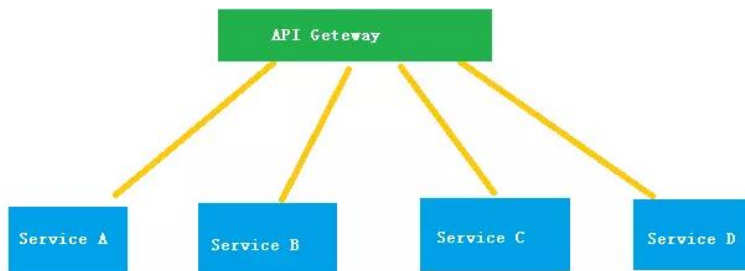
微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力。

区别:

1. 分布式

那么分布式又是啥？

分布式服务顾名思义服务是分散部署在不同的机器上的，一个服务可能负责几个功能，是一种面向 SOA 架构的，服务之间也是通过 rpc 来交互或者是 webservice 来交互的。逻辑架构设计完后就该做物理架构设计，系统应用部署在超过一台服务器或虚拟机上，且各分开部署的部分彼此通过各种通讯协议交互信息，就可算作分布式部署，生产环境下的微服务肯定是分布式部署的，分布式部署的应用不一定是微服务架构的，比如集群部署，它是把相同应用复制到不同服务器上，但是逻辑功能上还是单体应用。



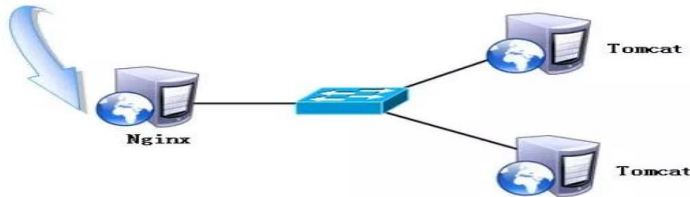
将一个大的系统划分为多个业务模块，业务模块分别部署到不同的机器上，各个业务模块之间通过接口进行数据交互。区别分布式的方式是根据不同机器不同业务。

上面：service A、B、C、D 分别是业务组件，通过 API Gateway 进行业务访问。

注：分布式需要做好事务管理。

分布式事务可参考：[微服务架构的分布式事务解决方案](#)

2. 集群模式



集群模式是不同服务器部署同一套服务对外访问，实现服务的负载均衡。区别集群的方式是根据部署多台服务器业务是否相同。

注：集群模式需要做好 session 共享，确保在不同服务器切换的过程中不会因为没有获取到 session 而中止退出服务。

一般配置 Nginx 的负载均衡实现：静态资源缓存、Session 共享可以附带实现，Nginx 支持 5000 个并发量。

分布式是否属于微服务？

答案是肯定的。[微服务](#)的意思也就是将模块拆分成一个独立的服务单元通过接口来实现数据的交互。

微服务架构

微服务是啥？

这里不引用书本上的复杂概论了，简单来说微服务就是很小的服务，小到一个服务只对应一个单一的功能，只做一件事。这个服务可以单独部署运行，服务之间可以通过 RPC 来相互交互，每个微服务都是由独立的小团队开发，测试，部署，上线，负责它的整个生命周期。

微服务架构又是啥？

在做架构设计的时候，先做逻辑架构，再做物理架构，当你拿到需求后，估算过最大用户量和并发量后，计算单个应用服务器能否满足需求，如果用户量只有几百人的小应用，单体应用就能搞定，即所有应用部署在一个应用服务器里，如果是很大用户量，且某些功能会被频繁访问，或者某些功能计算量很大，建议将应用拆解为多个子系统，各自负责各自功能，这就是微服务架构。

微服务的设计是为了不因为某个模块的升级和 BUG 影响现有的系统业务。微服务与分布式的细微差别是，微服务的应用不一定是分散在多个服务器上，他也可以是同一个服务器。

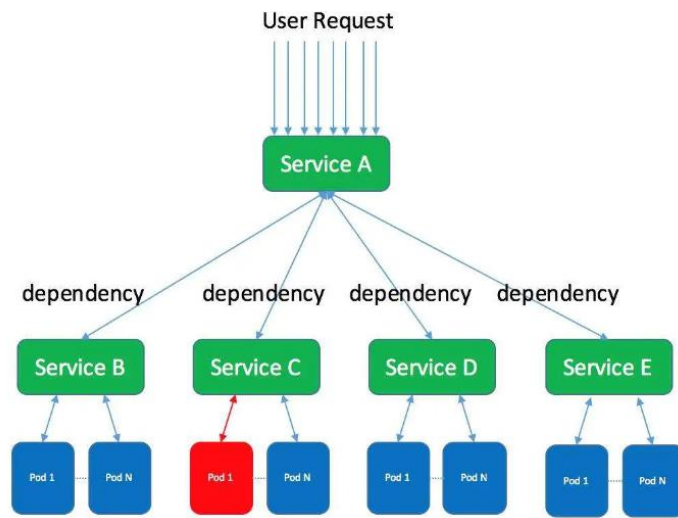
微服务相比分布式服务来说，它的粒度更小，服务之间耦合度更低，由于每个微服务都由独立的小团队负责，因此它敏捷性更高，分布式服务最后都会向微服务架构演化，这是一种趋势，不过服务微服务化后带来的挑战也是显而易见的，例如服务粒度小，数量大，后期运维将会很难。

微服务是架构设计方式，分布式是系统部署方式，两者概念不同

微服务是指很小的服务，可以小到只完成一个功能，这个服务可以单独部署运行，不同服务之间通过 rpc 调用。

分布式是指服务部署在不同的机器上，一个服务可以提供一个或多个功能，服务之间也是通过 rpc 来交互或者是 webservice 来交互的。

两者的关系是，系统应用部署在超过一台服务器或虚拟机上，且各分开部署的部分彼此通过各种通讯协议交互信息，就可算作分布式部署，生产环境下的微服务肯定是分布式部署的，分布式部署的应用不一定是微服务架构的，比如集群部署，它是把相同应用复制到不同服务器上，但是逻辑功能上还是单体应用。



分布式和微服的[架构](#)很相似，只是部署的方式不一样而已。