**Python面试题01 基础题**

**Python基础到高阶面试**

**001、**[**为什么学python**](https://blog.csdn.net/Darkman_EX/article/details/81101232)**？**

答题路线：a、python的优点，b、python的应用领域广；

具体：

优点

1、python语法非常优雅，简单易学

2、免费开源

3、跨平台，可以自由移植

4、可扩展，可嵌入性强

5、第三方库丰富

应用领域

1、在系统编程中应用广泛，比如说shell工具。

2、在网络爬虫方面功能非常强大，常用的库如scrapy，request等

3、在web开发中使用也很广泛，如很多大型网站都用python开发的，如ins，youtube等，常用的框架如django，flask等

4、python在系统运维中应用广泛，尤其在linux运维方面，基本上都是自动化运维。

5、在人工智能，云计算，金融等方面也应用非常广泛。

**002、通过什么途径学习python？**

1、通过看官方文档；

2、通过哔哩哔哩上的视频教程；

3、通过百度查资料；

4、通过买python相关方面的书；

**003、公司线上和开发环境使用的什么系统？**

线上用的centos和Ubuntu系统；开发环境用的windows，mac，还有Ubuntu。

**004、python和java、php、c、c#、c++ 等其他语言对比？**

1、Java C# PHP Python (开发效率高)，这些语言本身不需要你去管理内存了。它们都有自己的虚拟机，对于开辟内存、释放内存都由这个虚拟机处理；

2、C 和 Python、Java、C#等C语言： 代码编译得到 机器码 ，机器码在处理器上直接执行，每一条指令控制CPU工作其他语言： 代码编译得到 字节码 ，虚拟机执行字节码并转换成机器码再后在处理器上执行Python 和 C Python这门语言是由C开发而来；

3、对于使用：Python的类库齐全并且使用简洁，如果要实现同样的功能，Python 10行代码可以解决，C可能就需要100行甚至更多；

4、对于速度：Python的运行速度相较与C，绝逼是慢了Python 和 Java、C#等；

5、对于使用：Linux原装Python，其他语言没有；以上几门语言都有非常丰富的类库支持；

6、对于速度：Python在速度上可能稍显逊色所以，Python和其他语言没有什么本质区别，其他区别在于：擅长某领域、人才丰富、先入为主；

**004、简述解释型和编译型编程语言？**

1、解释型语言编写的程序不需要编译，在执行的时候，专门有一个解释器能够将VB语言翻译成机器语言，每个语句都是执行的时候才翻译。这样解释型语言每执行一次就要翻译一次，效率比较低。

2、用编译型语言写的程序执行之前，需要一个专门的编译过程，通过编译系统，把源高级程序编译成为机器语言文件，翻译只做了一次，运行时不需要翻译，所以编译型语言的程序执行效率高，但也不能一概而论，部分解释型语言的解释器通过在运行时动态优化代码，甚至能够使解释型语言的性能超过编译型语言。

**005、python解释器种类以及特点？**

1、CPython：c语言开发的 使用最广的解释器；

2、IPython：基于cpython之上的一个交互式计时器 交互方式增强 功能和cpython一样；

3、PyPy：目标是执行效率 采用JIT技术 对python代码进行动态编译，提高执行效率；

4、JPython：运行在Java上的解释器 直接把python代码编译成Java字节码执行；

5、IronPython：运行在微软 .NET 平台上的解释器，把python编译成. NET 的字节码；

**006、python现在是什么版本了？**

答：3.8

**007、python3.6、3.7和3.8有什么区别？**

很多人在学习了基本的Python语言知识后，就转入应用阶段了，后期很少对语言本身的新变化、新内容进行跟踪学习和知识更新，甚至连已经发布了好几年的Python3.6的新特性都缺乏了解。

本文列举了Python3.6、3.7、3.8三个版本的新特性，学习它们有助于提高对Python的了解，跟上最新的潮流。

一、Python3.6新特性

1、新的格式化字符串方式

新的格式化字符串方式，即在普通字符串前添加 f 或 F 前缀，其效果类似于str.format()。比如

name = "red"

print(f"He said his name is {name}.")

# 'He said his name is red.'

相当于：

print("He said his name is {name}.".format(\*\*locals()))

此外，此特性还支持嵌套字段，比如：

import decimal

width = 10

precision = 4

value = decimal.Decimal("12.34567")

print(f"result: {value:{width}.{precision}}")

#'result: 12.35'

2、变量声明语法

可以像下面一样声明一个变量并指定类型：

from typing import List, Dict

primes: List[int] = []

captain: str # 此时没有初始值

class Starship:

stats: Dict[str, int] = {}

3、数字的下划线写法

允许在数字中使用下划线，以提高多位数字的可读性。

a = 1\_000\_000\_000\_000\_000 # 1000000000000000

b = 0x\_FF\_FF\_FF\_FF # 4294967295

除此之外，字符串格式化也支持\_选项，以打印出更易读的数字字符串：

'{:\_}'.format(1000000) # '1\_000\_000'

'{:\_x}'.format(0xFFFFFFFF) # 'ffff\_ffff'

4、异步生成器

在Python3.5中，引入了新的语法 async 和 await 来实现协同程序。但是有个限制，不能在同一个函数体内同时使用 yield 和 await。Python3.6中，这个限制被放开了，允许定义异步生成器：

async def ticker(delay, to):

"""Yield numbers from 0 to \*to\* every \*delay\* seconds."""

for i in range(to):

yield i

await asyncio.sleep(delay)

5、异步解析器

允许在列表list、集合set 和字典dict 解析器中使用 async 或 await 语法。

result = [i async for i in aiter() if i % 2]

result = [await fun() for fun in funcs if await condition()]

6、新增加模块

标准库（The Standard Library）中增加了一个新的模块：secrets。该模块用来生成一些安全性更高的随机数，用于管理passwords, account authentication, security tokens, 以及related secrets等数据。

7、其他新特性

1、新的 PYTHONMALLOC 环境变量允许开发者设置内存分配器，以及注册debug钩子等。

2、asyncio模块更加稳定、高效，并且不再是临时模块，其中的API也都是稳定版的了。

3、typing模块也有了一定改进，并且不再是临时模块。

4、datetime.strftime 和 date.strftime 开始支持ISO 8601的时间标识符%G, %u, %V。

5、hashlib 和 ssl 模块开始支持OpenSSL1.1.0。

6、hashlib模块开始支持新的hash算法，比如BLAKE2, SHA-3 和 SHAKE。

7、Windows上的 filesystem 和 console 默认编码改为UTF-8。

8、json模块中的 json.load() 和 json.loads() 函数开始支持 binary 类型输入。

更多内容参考官方文档：[What's New In Python 3.6](https://docs.python.org/3.6/whatsnew/3.6.html)

二、Python3.7新特性

Python 3.7于2018年6月27日发布， 包含许多新特性和优化，增添了众多新的类，可用于数据处理、针对脚本编译和垃圾收集的优化以及更快的异步I/O，主要如下：

1、用类处理数据时减少样板代码的数据类。

2、一处可能无法向后兼容的变更涉及处理生成器中的异常。

3、面向解释器的“开发模式”。

4、具有纳秒分辨率的时间对象。

5、环境中默认使用UTF-8编码的UTF-8模式。

6、触发调试器的一个新的内置函数。

1、新增内置函数breakpoint()

使用该内置函数，相当于通过代码的方式设置了断点，会自动进入Pbd调试模式。

如果在环境变量中设置PYTHONBREAKPOINT=0会忽略此函数。并且，pdb 只是众多可用调试器之一，你可以通过设置新的 PYTHONBREAKPOINT 环境变量来配置想要使用的调试器。

下面有一个简单例子，用户需要输入一个数字，判断它是否和目标数字一样：

"""猜数字游戏"""

def guess(target):

user\_guess = input("请输入你猜的数 >>> ")

if user\_guess == target:

return "你猜对了!"

else:

return "猜错了"

if \_\_name\_\_ == '\_\_main\_\_':

a = 100

print(guess(a))

不幸的是，即使猜的数和目标数一样，打印的结果也是‘猜错了’，并且没有任何异常或错误信息。

为了弄清楚发生了什么，我们可以插入一个断点，来调试一下。以往一般通过print大法或者IDE的调试工具，但现在我们可以使用 breakpoint()。

"""猜数字游戏"""

def guess(target):

user\_guess = input("请输入你猜的数 >>> ")

breakpoint() //加入这一行

if user\_guess == target:

return "你猜对了!"

else:

return "猜错了"

if \_\_name\_\_ == '\_\_main\_\_':

a = 100

print(guess(a))

在 pdb 提示符下，我们可以调用 locals() 来查看当前的本地作用域的所有变量。（pdb 有大量的命令，你也可以在其中运行正常的Python 语句）

请输入你猜的数 >>> 100

> d:\work\for\_test\py3\_test\test.py(7)guess()

-> if user\_guess == target:

(Pdb) locals()

{'target': 100, 'user\_guess': '100'}

(Pdb) type(user\_guess)

<class 'str'>

搞明白了，target是一个整数，而user\_guess 是一个字符串，这里发生了类型对比错误。

2、类型和注解

从 Python 3.5 开始，类型注解就越来越受欢迎。对于那些不熟悉类型提示的人来说，这是一种完全可选的注释代码的方式，以指定变量的类型。

什么是注解？它们是关联元数据与变量的语法支持，可以是任意表达式，在运行时被 Python 计算但被忽略。注解可以是任何有效的 Python 表达式。

下面是个对比的例子：

# 不带类型注解

def foo(bar, baz):

# 带类型注解

def foo(bar: 'Describe the bar', baz: print('random')) -> 'return thingy':

上面的做法，其实是Python对自身弱类型语言的强化，希望获得一定的类型可靠和健壮度，向Java等语言靠拢。

在 Python 3.5 中，注解的语法获得标准化，此后，Python 社区广泛使用了注解类型提示。

但是，注解仅仅是一种开发工具，可以使用 PyCharm 等 IDE 或 Mypy 等第三方工具进行检查，并不是语法层面的限制。

我们前面的猜数程序如果添加类型注解，它应该是这样的：

"""猜数字游戏"""

def guess(target:str):

user\_guess:str = input("请输入你猜的数 >>> ")

breakpoint()

if user\_guess == target:

return "你猜对了!"

else:

return "猜错了"

if \_\_name\_\_ == '\_\_main\_\_':

a:int = 100

print(guess(a))

PyCharm会给我们灰色的规范错误提醒，但不会给红色的语法错误提示。

1、用注解作为类型提示时，有两个主要问题：启动性能和前向引用。

2、在定义时计算大量任意表达式相当影响启动性能，而且 typing 模块非常慢

你不能用尚未声明的类型来注解

typing 模块如此缓慢的部分原因是，最初的设计目标是在不修改核心 CPython 解释器的情况下实现 typing 模块。随着类型提示变得越来越流行，这一限制已经被移除，这意味着现在有了对 typing 的核心支持。

而对于向前引用，看下面的例子：

class User:

def \_\_init\_\_(self, name: str, prev\_user: User) -> None:

pass

错误在于 User类型还没有被声明，此时的 prev\_user 不能定义为 User 类型。

为了解决这个问题，Python3.7 将注解的评估进行了推迟。并且，这项改动向后不兼容，需要先导入annotations，只有到Python 4.0后才会成为默认行为。

from \_\_future\_\_ import annotations

class User:

def \_\_init\_\_(self, name: str, prev\_user: User) -> None:

pass

或者如下面的例子：

class C:

def validate\_b(self, obj: B) -> bool:

...

class B:

...

3、新增dataclasses模块

这个特性可能是 Python3.7以后比较常用的，它有什么作用呢？

假如我们需要编写一个下面的类：

from datetime import datetime

import dateutil

class Article(object):

def \_\_init\_\_(self, \_id, author\_id, title, text, tags=None,

created=datetime.now(), edited=datetime.now()):

self.\_id = \_id

self.author\_id = author\_id

self.title = title

self.text = text

self.tags = list() if tags is None else tags

self.created = created

self.edited = edited

if type(self.created) is str:

self.created = dateutil.parser.parse(self.created)

if type(self.edited) is str:

self.edited = dateutil.parser.parse(self.edited)

def \_\_eq\_\_(self, other):

if not isinstance(other, self.\_\_class\_\_):

return NotImplemented

return (self.\_id, self.author\_id) == (other.\_id, other.author\_id)

def \_\_lt\_\_(self, other):

if not isinstance(other, self.\_\_class\_\_):

return NotImplemented

return (self.\_id, self.author\_id) < (other.\_id, other.author\_id)

def \_\_repr\_\_(self):

return '{}(id={}, author\_id={}, title={})'.format(

self.\_\_class\_\_.\_\_name\_\_, self.\_id, self.author\_id, self.title)

大量的初始化属性要定义默认值，可能还需要重写一堆魔法方法，来实现类实例的打印、比较、排序和去重等功能。

如果使用dataclasses进行改造，可以写成这个样子：

from dataclasses import dataclass, field

from typing import List

from datetime import datetime

import dateutil

@dataclass(order=True) //注意这里

class Article(object):

\_id: int

author\_id: int

title: str = field(compare=False)

text: str = field(repr=False, compare=False)

tags: List[str] = field(default=list(), repr=False, compare=False)

created: datetime = field(default=datetime.now(), repr=False, compare=False)

edited: datetime = field(default=datetime.now(), repr=False, compare=False)

def \_\_post\_init\_\_(self):

if type(self.created) is str:

self.created = dateutil.parser.parse(self.created)

if type(self.edited) is str:

self.edited = dateutil.parser.parse(self.edited)

这使得类不仅容易设置，而且当我们创建一个实例并打印出来时，它还可以自动生成优美的字符串。在与其他类实例进行比较时，它也会有适当的行为。这是因为dataclasses除了帮我们自动生成 \_\_init\_\_ 方法外，还生成了一些其他特殊方法，如 repr、eq 和 hash 等。

Dataclasses 使用字段 field来完提供默认值，手动构造一个 field() 函数能够访问其他选项，从而更改默认值。例如，这里将 field 中的 default\_factory 设置为一个 lambda 函数，该函数提示用户输入其名称。

from dataclasses import dataclass, field

class User:

name: str = field(default\_factory=lambda: input("enter name"))

4、生成器异常处理

在Python 3.7中，生成器引发StopIteration异常后，StopIteration异常将被转换成RuntimeError异常，那样它不会悄悄一路影响应用程序的堆栈框架。这意味着如何处理生成器的行为方面不太敏锐的一些程序会在Python 3.7中抛出RuntimeError。在Python 3.6中，这种行为生成一个弃用警告；在Python 3.7中，它将生成一个完整的错误。

一个简易的方法是使用try/except代码段，在StopIteration传播到生成器的外面捕获它。更好的解决方案是重新考虑如何构建生成器――比如说，使用return语句来终止生成器，而不是手动引发StopIteration。

5、开发模式

Python解释器添加了一个新的命令行开关：-X，让开发人员可以为解释器设置许多低级选项。

这种运行时的检查机制通常对性能有重大影响，但在调试过程中对开发人员很有用。

-X 激活的选项包括：

1、asyncio模块的调试模式。这为异步操作提供了更详细的日志记录和异常处理，而异常操作可能很难调试或推理。

2、面向内存分配器的调试钩子。这对于编写CPython扩展件的那些人很有用。它能够实现更明确的运行时检查，了解CPython如何在内部分配内存和释放内存。

3、启用faulthandler模块，那样发生崩溃后，traceback始终转储出去。

6、 高精度时间函数

Python 3.7中一类新的时间函数返回纳秒精度的时间值。尽管Python是一种解释型语言，但是Python的核心开发人员维克多•斯廷纳（Victor Stinner）主张报告纳秒精度的时间。最主要的原因是，在处理转换其他程序（比如数据库）记录的时间值时，可以避免丢失精度。

新的时间函数使用后缀\_ns。比如说，time.process\_time()的纳秒版本是time.process\_time\_ns()。请注意，并非所有的时间函数都有对应的纳秒版本。

7、其他新特性

1、字典现在保持插入顺序。这在 3.6 中是非正式的，但现在成为了官方语言规范。在大多数情况下，普通的 dict 能够替换 collections.OrderedDict。

2、.pyc 文件具有确定性，支持可重复构建 —— 也就是说，总是为相同的输入文件生成相同的 byte-for-byte 输出。

3、新增contextvars模块，针对异步任务提供上下文变量。

4、\_\_main\_\_中的代码会显示弃用警告（DeprecationWarning）。

5、新增UTF-8模式。在Linux/Unix系统，将忽略系统的locale，使用UTF-8作为默认编码。在非Linux/Unix系统，需要使用-X utf8选项启用UTF-8模式。

6、允许模块定义\_\_getattr\_\_、\_\_dir\_\_函数，为弃用警告、延迟import子模块等提供便利。

7、新的线程本地存储C语言API。

8、更新Unicode数据到11.0。

三、Python3.8新特性

Python3.8版本于2019年10月14日发布，以下是 Python 3.8 相比 3.7 的新增特性。

1、海象赋值表达式

新的语法 :=，将值赋给一个更大的表达式中的变量。它被亲切地称为 “海象运算符”(walrus operator)，因为它长得像海象的眼睛和象牙。

“海象运算符” 在某些时候可以让你的代码更整洁，比如：

在下面的示例中，赋值表达式可以避免调用 len () 两次:

if (n := len(a)) > 10:

print(f"List is too long ({n} elements, expected <= 10)")

类似的好处还可体现在正则表达式匹配中需要使用两次匹配对象的情况中，一次检测用于匹配是否发生，另一次用于提取子分组:

discount = 0.0

if (mo := re.search(r'(\d+)% discount', advertisement)):

discount = float(mo.group(1)) / 100.0

此运算符也可用于配合 while 循环计算一个值，来检测循环是否终止，而同一个值又在循环体中再次被使用的情况:

# Loop over fixed length blocks

while (block := f.read(256)) != '':

process(block)

或者出现于列表推导式中，在筛选条件中计算一个值，而同一个值又在表达式中需要被使用:

[clean\_name.title() for name in names

if (clean\_name := normalize('NFC', name)) in allowed\_names]

请尽量将海象运算符的使用限制在清晰的场合中，以降低复杂性并提升可读性。

2、仅限位置形参

新增一个函数形参语法 / 用来指明某些函数形参必须使用仅限位置而非关键字参数的形式。

这种标记语法与通过 help () 所显示的使用 Larry Hastings 的 Argument Clinic 工具标记的 C 函数相同。

在下面的例子中，形参 a 和 b 为仅限位置形参，c 或 d 可以是位置形参或关键字形参，而 e 或 f 要求为关键字形参:

def f(a, b, /, c, d, \*, e, f):

print(a, b, c, d, e, f)

以下是合法的调用:

f(10, 20, 30, d=40, e=50, f=60)

但是，以下均为不合法的调用:

f(10, b=20, c=30, d=40, e=50, f=60) # b 不可以是一个关键字参数

f(10, 20, 30, 40, 50, f=60) # e 必须是一个关键字参数

这种标记形式的一个用例是它允许纯 Python 函数完整模拟现有的用 C 代码编写的函数的行为。例如，内置的 pow () 函数不接受关键字参数:

def pow(x, y, z=None, /):

"Emulate the built in pow() function"

r = x \*\* y

return r if z is None else r%z

另一个用例是在不需要形参名称时排除关键字参数。例如，内置的 len () 函数的签名为 len (obj, /)。这可以排除如下这种笨拙的调用形式:

len(obj='hello') # The "obj" keyword argument impairs readability

另一个益处是将形参标记为仅限位置形参将允许在未来修改形参名而不会破坏客户的代码。例如，在 statistics 模块中，形参名 dist 在未来可能被修改。这使得以下函数描述成为可能:

def quantiles(dist, /, \*, n=4, method='exclusive')

...

由于在 / 左侧的形参不会被公开为可用关键字，其他形参名仍可在 \*\*kwargs 中使用:

>>> def f(a, b, /, \*\*kwargs):

... print(a, b, kwargs)

...

>>> f(10, 20, a=1, b=2, c=3) # a and b are used in two ways

10 20 {'a': 1, 'b': 2, 'c': 3}

这极大地简化了需要接受任意关键字参数的函数和方法的实现。例如，下面是 collections 模块中的代码摘录：

class Counter(dict):

def \_\_init\_\_(self, iterable=None, /, \*\*kwds):

# Note "iterable" is a possible keyword argument

3、f 字符串支持 =

增加 = 说明符用于 f-string。形式为 f'{expr=}' 的 f 字符串将扩展表示为表达式文本，加一个等于号，再加表达式的求值结果。例如：

>>> user = 'eric\_idle'

>>> member\_since = date(1975, 7, 31)

>>> f'{user=} {member\_since=}'

"user='eric\_idle' member\_since=datetime.date(1975, 7, 31)"

f 字符串格式说明符允许更细致地控制所要显示的表达式结果:

>>> delta = date.today() - member\_since

>>> f'{user=!s} {delta.days=:,d}'

'user=eric\_idle delta.days=16,075'

= 说明符将输出整个表达式，以便详细演示计算过程:

>>> print(f'{theta=} {cos(radians(theta))=:.3f}')

theta=30 cos(radians(theta))=0.866

4、 typing模块的改进

Python是动态类型语言，但可以通过typing模块添加类型提示，以便第三方工具验证Python代码。Python 3.8给typing添加了一些新元素，因此它能够支持更健壮的检查：

1、final修饰器和Final类型标注表明，被修饰或被标注的对象在任何时候都不应该被重写、继承，也不能被重新赋值。

2、Literal类型将表达式限定为特定的值或值的列表（不一定是同一个类型的值）。

3、TypedDict可以用来创建字典，其特定键的值被限制在一个或多个类型上。注意这些限制仅用于编译时确定值的合法性，而不能在运行时进行限制。

5、多进程共享内存

multiprocessing模块新增SharedMemory类，可以在不同的Python进城之间创建共享的内存区域。

在旧版本的Python中，进程间共享数据只能通过写入文件、通过网络套接字发送，或采用Python的pickle模块进行序列化等方式。共享内存提供了进程间传递数据的更快的方式，从而使得Python的多处理器和多内核编程更有效率。

共享内存片段可以作为单纯的字节区域来分配，也可以作为不可修改的类似于列表的对象来分配，其中能保存数字类型、字符串、字节对象、None对象等一小部分Python对象。

6、 新版本的pickle协议

Python的pickle模块提供了一种序列化和反序列化Python数据结构或实例的方法，可以将字典原样保存下来供以后读取。不同版本的Python支持的pickle协议不同，而3.8版本的支持范围更广、更强大、更有效的序列化。

Python 3.8引入的第5版pickle协议可以用一种新方法pickle对象，它能支持Python的缓冲区协议，如bytes、memoryviews或Numpy array等。新的pickle避免了许多在pickle这些对象时的内存复制操作。

NumPy、Apache Arrow等外部库在各自的Python绑定中支持新的pickle协议。新的pickle也可以作为Python 3.6和3.7的插件使用，可以从PyPI上安装。

7、性能改进

1、许多内置方法和函数的速度都提高了20%~50%，因为之前许多函数都需要进行不必要的参数转换。

2、一个新的opcode缓存可以提高解释器中特定指令的速度。但是，目前实现了速度改进的只有LOAD\_GLOBAL opcode，其速度提高了40%。以后的版本中也会进行类似的优化。

3、文件复制操作如shutil.copyfile()和shutil.copytree()现在使用平台特定的调用和其他优化措施，来提高操作速度。

4、新创建的列表现在平均比以前小了12%，这要归功于列表构造函数如果能提前知道列表长度的情况下，可以进行优化。

5、Python 3.8中向新型类（如class A(object)）的类变量中的写入操作变得更快。operator.itemgetter()和collections.namedtuple()也得到了速度优化。

**008、python中有哪些可变类型与不可变类型？**

可变类型：会在原来的内存地址上修改元素 比如： 列表，字典；

不可变类型：不会在原来的内存地址上修改元素，而是指向了新的内存引用 比如：整型，字符串，元组；

**010、面向对象的3个特性是什么？**

1、封装：根据职责将属性和方法封装到一个抽象的类中定义类的准则；

2、继承：实现代码的重用，相同的代码不需要重复的编写；

3、多态：不同的子类调用相同的父类，产生不同的结果。

**011、什么是闭包？**

闭包是由两个函数嵌套定义，内部函数里面用到了外部函数里面的变量值，那么这个变量（变量值）加上内部还是里面的代码组成的代码块，组成了一个新的内存空间，我们把这个空间叫做闭包。

闭包 = 函数+环境变量（在函数定义的时候定义，它不是全局变量,这个环境变量一定要被内部函数调用才算是闭包）

**012.匿名函数/函数/闭包/对象在做实参时有什么区别？**

1、匿名函数：能够完成基本的简单功能，传递是这个函数的引用，只有功能

2、普通函数：能够完成比较普通的功能，传递是这个函数的引用，只有功能

3、闭包：能够完成比较复杂的功能，传递是这个闭包中的函数和数据，因此传递是功能+数据

4、对象：能够完成最为复杂的功能，传递是很多数据和很对功能，因此传递是功能+数据

**013、什么是迭代器，为什么要使用它？**

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator

from collections import Iterator  
def Iteror2():  
  '''迭代器原理'''  
  list = [1, 2, 3, 4]  
  it = iter(list) # 创建迭代器  
  while True:  
      try:  
          print(next(it))  
      except StopIteration:  
          sys.exit()  
# 判断是否为迭代器  
isinstance((x for x in range(10)), Iterator)  
#>>> True

  迭代是访问集合元素的一种方式。

1、迭代器保存的是获取数据的方式而不是结果，所以想用的时候就可以生成，节省大量内存空间，它是一个可以记住遍历的位置的对象。

2、迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：iter() 和 next()。 字符串，列表或元组对象都可用于创建迭代器。

**014、什么是生成器，为什么要使用它？**

生成器定义在Python中,一边循环一边计算的机制，使用了 yield 的函数被称为生成器（generator）。

跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个特殊的迭代器。 在调用生成器运行的过程中，每次遇到 yield 时函数会暂停并保存当前所有的运行信息， 返回 yield 的值, 并在下一次执行 next() 方法时从当前位置继续运行。 调用一个生成器函数，返回的是一个迭代器对象。

list5 = [x for x in range(5)]  
print(list5)   #output：[0, 1, 2, 3, 4]

 列表所有数据都在内存中，如果有海量数据的话将会非常耗内存。如仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。如果列表元素按照某种算法推算出来，那我们就可以在循环的过程中不断推算出后续的元素，这样就不必创建完整的list，从而节省大量的内存空间。

简单一句话：我又想要得到庞大的数据，又想让它占用内存空间少，那就用生成器！生成器仅仅保存了一套生成数值的算法，并且没有让这个算法现在就开始执行，而是我什么时候调它，它什么时候开始计算一个新的值，并给你返回。

**015、简述迭代器、生成器的区别？**

能使用for遍历的就叫可迭代对象，能使用next方法的就是迭代器，生成器是特殊的迭代器。生成器能做到迭代器能做的所有事,而且因为自动创建了 iter()和 next()方法,生成器显得特别简洁和高效，使用生成器表达式取代列表解析可以同时节省内存空间。

**016、什么是装饰器，为什么要使用它？**

1、装饰器本质上是一个函数，这个函数的主要作用是包装另一个函数或类包装的- 目的是在不改变原函数名的情况下改变被包装对象的行为。

2、接收一个函数，内部对其包装，然后返回一个新函数，这样子动态的增强函数功能

3、通过高阶函数传递函数参数，新函数添加旧函数的需求，然后执行旧函数。

4、在django中middkeware中间件 其实就是高级的装饰器用法。

**017、.谈谈对不定⻓参数的理解？**

一般分为两种：

一种是args 位置参数 在定义函数时，在形参前面加一个，代表可以接收任意多个实参，用元组类型保存所有数据。一般写成def function(\*args)；

一种是\*\*kwargs 命名参数 代表可以接收任意多个的命名参数，用字典类型保存。

**018、谈谈对缺省参数的理解？**

如果调用函数的时候，传递了对应位置的实参，那就使用这个传递的值，如果没有传递对应的值，那就使用缺省参数的值。

**019、break和continue的区别？**

1、break和continue都是用于while嵌套循环中 ；

2、continue是结束内层的while循环，但并没有终止整个循环；

3、break是结束整个while循环；

**021、\_\_new\_\_() 和 \_\_init\_\_()的区别？**

new作用于init之前。前者可以决定是否调用后者，可以决定调用哪个类的init方法。

**022、range和xrange的区别？**

xrange和range 的用法完全相同，但是xrange返回的是一个生成器。

**023、yield和return的相同点和区别？**

相同点：功能都是返回程序执行结果；

区别：yield返回执行结果并不中断程序执行，return在返回执行结果的同时中断程序执行。

**024、列举5个python常用标准库并说明其作用？**

1、os：提供不少于操作系统相关联的函数；

2、sys：通常用于命令行参数；

3、datetime：日期时间；

4、re：正则匹配；

5、math：数学运算；

**025、谈谈你知道的几种设计模式？**

1、单例模式：保证一个类仅有一个实例，并提供一个访问他的全局访问点，例如框架中的数据库连接

2、装饰器模式：不修改元类代码和继承的情况下动态扩展类的功能，例如框架中的每个controller文件会提供before和after方法。

3、迭代器模式： 提供一个方法顺序访问一个聚合对象中各个元素，在PHP中将继承 Iterator 类

4、命令模式: 将”请求”封闭成对象, 以便使用不同的请求,队列或者日志来参数化其他对象. 命令模式也支持可撤销的操作.

**026、详解Python2和Python3的区别？**

01、python解释器默认编码：python2 解释器默认编码：ascii python3 解释器默认编码：utf-8；

02、输入：python2：name=raw\_input('请输入姓名') python3：name=input('请输入你的姓名')；

03、输出：python2：print "你好" python3：print("你好")；

04、数字表示：

python2 64位机器，范围-2^63~2^63-1 超出上述范围，python自动转化为long(长整型) 注：long(长整型)数字末尾有一个L；

python3 所有整型都是int，没有long(长整型) ；

05、整型除法：python2：只能保留整数位；python3：可以保留所有内容；

06、range / xrange：

python2： xrange：不会在内存中立即创建，而是在循环时，边循环边创建 range：在内存立即把所有的值创建；

python3： 只有range，相当于python2中的xrange range：不会在内存中立即创建，而是在循环时，边循环边创建；

07、包的定义：python2：文件夹中必须有\_ \_ init \_ .py文件 python3：不需要有 \_ init \_ \_.py文件；

08、字典的keys / values / items方法：python2：返回列表，通过索引可以取值；python3：返回迭代器，只能通过循环取值，不能通过索引取值；

09、map / filter：python2：返回列表，直接创建值，可以通过索引取值；python3：返回迭代器，不直接创建值，通过循环，边循环边创建；

10、str(字符串类型)的区别(最大区别，优先写这个)：

python2： str类型，相当于python3中的字节类型，utf-8/gbk等其他编码；unicode类型，相当于python3中的字符串类型，unicode编码；python2中没有字节类型；

python3： str类型，字符串类型，unicode编码；python3中没有unicode类型

11、继承object：

class Foo:  
  pass  
class Foo(object):  
  pass  
# 在python3中这俩的写法是一样，因为所有的类默认都会继承object类，全部都是新式类。  
# 如果在python2中这样定义，则称其为：经典类  
class Foo:  
  pass   
# 如果在python2中这样定义，则称其为：新式类  
class Foo(object):  
  pass  
# 新式类  
# 继承object  
  # 支持super  
  # 多继承 广度优先C3算法  
  # mro方法  
# 经典类  
  # py2中不继承object  
  # 没有super语法  
  # 多继承 深度优先  
  # 没有mro方法

**013、简单列举你所了解的所有Python2和Python3的区别？**

[python2和python3的区别](https://blog.csdn.net/weixin_41819299/article/details/81259721)：

1、python2没有nonlocal关键字，要修改临时变量只能将其改成可变数据类型，如数组。b=[a]；py2中函数用global关键字声明某个变量为全局变量，无法实现在嵌套函数中给一个变量声明为局部变量；py3新增了nonlocal关键字。

2、print()函数代替print语句；print在py2不加括号，print()py3加括号；

3、Python3加入 Unicode 字符串，用以编码存储字符串。py2中unicode类型表示字符串序列，str类型表示字节序列；py3中str类型表示字符串序列，byte类型表示字节序列。比如py3用 utf-8可以用来输入中文；py2默认ASCII编码，py3默认utf-8编码；

4、Python3去掉long类型，新增了bytes；

5、Python 3整数之间做除法可以得到浮点数的结果，不需要进行数据格式转换1/2=0.5；Python 2整数int间除法结果为把运算结果去尾的整数1/2=0，3/2.0=1.5；

6、Python3 中 range()，Python2 中 xrange()；

7、python2中的不等于号可以是!=或者<>，python3只能是!=；

8、python2中raw\_input()用来输入字符串，而python3中使用input()来输入字符串；

**027、列表和元组的区别？**

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）。

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。 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。

这些区别结实率两者在设计哲学上的不同： 列表可被用于保存多个互相独立对象的数据集合 元组用于描述一个不会改的事务的多个属性；

总结：

第一个不同点是元组的声明使用小括号，而列表使用方括号，当声明只有一个元素的元组时，需要在这个元素的后面添加英文逗号；

第二个不同点是元组声明和赋值后，不能像列表一样添加、删除和修改元素，也就是说元组在程序运行过程中不能被修改。

**006、位和字节的关系？**

8位 == 一字节

**007、b、B、kB、MB、GB的关系？**

1、1B=8b；

2、1kB=1024B；

3、1MB=1024kB；

4、1GB=1024MB；

**008、请列出至少5个PEP8规范？**

[PEP8规范](https://www.jianshu.com/p/e132bea1d2c9)：

每个缩进级别使用4个空格；

每行代码的最大长度限制为79个字符；

若是导入多个库函数，应该分开依次导入：道路应按照以下顺序导入 a、标准库导入 b、相关的第三方库导入 c、本地应用程序的库导入；

在表达式中避免无关的空格：在括号或者大括号内，在尾随逗号和后面的右括号之间，在逗号，分号或者冒号前面，函数名的与后面的参数的括号之间。

代码更改时，相应的注释也要随之更改；

命名要规范，通俗易懂；

**009、or 和 and？**

v1=1 or 3

v2=1 and 3

v3=0 and 2 and 1

v4=0 and 2 or 1

v5=0 and 2 or 1 or 4

v6=0 or False and 1

结果： \* v1 = 1 \* v2 = 3 \* v3 = 0 \* v4 = 1 \* v5 = 1 \* v6 = False

基本运算规律

1、在不加括号时候, and优先级大于or；

2、x or y 的值只可能是x或y. x为真就是x, x为假就是y；

3、x and y 的值只可能是x或y. x为真就是y, x为假就是x；

**010、ascii、Unicode、utf-8、gbk的区别？**

**简写版：**

1、ascii 是最早美国用的标准信息交换码，把所有的字母的大小写，各种符号用 二进制来表示，共有256中，加入些拉丁文等字符，1bytes代表一个字符；

2、Unicode是为了统一世界各国语言的不用，统一用2个bytes代表一个字符，可以表达2^16=65556个，称为万国语言，特点：速度快，但浪费空间；

3、utf-8 为了改变Unicode的这种缺点，规定1个英文字符用1个字节表示，1个中文字符用3个字节表示，特点；节省空间，速度慢，用在硬盘数据传输，网络数据传输，相比硬盘和网络速度，体现不出来的；

4、gbk  是中文的字符编码，用2个字节代表一个字符；

**详细版：**

ascii：在计算机内部，所有信息最终都是一个二进制值。每一个二进制位（bit），有0和1两种状态，因此，8个二进制位可以组合出256种状态，这被称为字节（byte)。上个世纪60年代，美国制定了一套字符编码，对英文字符与二进制之间做了联系，这被称为ASCII码，一直沿用至今。ASCII码一共规定了128个字符，比如SPACE是32，A是65，这128个符号只咱用了一个字节的后面七位，最前面的一位统一规定为0。  
unicode：世界上有多种编码方法，同一个二进制数字可以被解释称不同的符号。因此，在打开一个文本文件时候，就必须知道它的编码方式，用错误的编码方式打开，就会出现乱码。Unicode编码，这是一种所有符号的编码。Unicode显然是一个巨大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如U+0041表示英语的大写字母A，U+4e25表示汉字严。  
在Unicode庞大的字符集的优势下，还存在一个问题，比如一个汉字，“严”的Unicode是十六进制4e25，转成二进制足足有15位，也就是，这个符号需要2个字节，表示其他字符还存在3个字节或者更多。计算机怎么区别三个字节表示的是同一个符号而不是分开表示三个呢？如果Unicode统一规定，每个符号用3个字节表示，但是某些字母显然不需要3个，那么就浪费了空间，文本文件大小超出了很多，这显然是不合理的。直到UTF8字符编码出现了。  
utf-8：UTF8的最大特点是，它是一种变长编码，可以使用1-4个字节表示一个符号，根据不同的符号来变化字节长度。  
UTF8编码规则只有两条：  
1）对于单字节的符号，字节的第一位设为0，后面的7位为这个符号的Unicode码。因此，对于英文字母，UTF8编码和ASCII编码是相同的。  
2）对于非单字节（假设字节长度为N）的符号，第一个字节的前N位都设为1，第N+1设为0，后面字节的前两位一律设为10，剩下的没有提及的二进制，全部为这个符号的Unicode码。  
gbk：GBK编码是对GB2312的扩展，完全兼容GB2312。采用双字节编码方案，剔出xx7F码位，共23940个码位，共收录汉字和图形符号21886个，GBK编码方案于1995年12月15日发布。它几乎完美支持汉字，因此经常会遇见GBK与Unicode的转换。  
1，各个编码之间的二进制，是不能互相识别的，会产生乱码。  
2，文件的存储，传输，不能是unicode （只能是utf-8 utf-16 gbk gbk2312 ascii等）

**011、字节码和机器码的区别？**

1、机器码是电脑CPU直接读取运行的机器指令，运行速度最快，但是非常晦涩难懂，也比较难编写，一般从业人员接触不到。

2、字节码是一种中间状态（中间码）的二进制代码（文件）。需要直译器转译后才能成为机器码。

**012、三元运算编写格式？**

表达式1 if 布尔表达式2 else 表达式3； 例如：a=3 if 3 > 4 else 5 。

**014、py2项目如何迁移成py3？**

1、先备份原文件，然后使用python3自带工具2to3.py将py2文件转换位py3文件；

2、手动将不兼容的代码改写成兼容py3的代码；

**015、用一行代码实现数值交换？**

a=1

b=2

答案：a,b=b,a

**016、python3和python2中int和long的区别？**

python2中有long类型，python3中没有long类型，只有int类型。python3中的int类型包括了long类型。

**017、xrange和range的区别？**

xrange和range用法相同，但是xrange是一个生成器，range结果是一个列表。xrange做循环的时候性能比range好。

都在循环时使用，xrange内存性能更好。  
for i in range(0, 20):  
for i in xrange(0, 20):  
What is the difference between range and xrange functions in Python 2.X?  
range creates a list, so if you do range(1, 10000000) it creates a list in memory with 9999999 elements.  
xrange is a sequence object that evaluates lazily.

<http://stackoverflow.com/questions/94935/what-is-the-difference-between-range-and-xrange-functions-in-python-2-x>

**018、如何实现字符串的反转？如：name=felix，反转成name=xilef？**

   name = "felix"  
   # 方法一：  
   name=name[::-1]  
   # 方法二：  
   name2=list(name)  
   name2.reverse()  
   name=''.join(name2)  
   # 方法三：  
   from functools import reduce  
   name=reduce(lambda x, y: y+x, name)

**019、文件操作时，read ，readline ，readlines，xreadlines的区别？**

read 读取整个文件；

readline 读取下一行,使用生成器方法；

readlines 读取整个文件到一个迭代器以供我们遍历，readlines返回的是一个列表；readlines()把文件的全部内容读到内存，并解析成一个list，当文件的体积很大时要占用很多内存;

xreadlines()返回一个iter(file)迭代器，在python2.3之后已经不再推荐使用这种表示方法了，直接使用for循环迭代文件；

**020、请列举布尔值位False的常见值？**

0、''、[]、{}、tuple()、None、set()

**021、列举字符串、列表、元组、字典每个常用的5个方法？**

字符串---[字符串方法总结](https://www.cnblogs.com/chendai21/p/8137285.html)：

1、strip() ->去掉字符串两端的空白符；

2、split() ->对字符串进行分割，默认按照空格分割；

3、join() ->字符串连接；

4、startwith(),endwith() ->判断是否以啥开头或者结尾；

5、replace() -> 字符串替换；

6、find() -> 查找字符串，存在返回第一个索引，不存在返回-1；

列表---[列表方法总结](https://www.cnblogs.com/smelond/p/7857701.html)：

1、count() ->统计在列表中出现的个数；

2、apped() ->在列表末尾添加值；

3、pop() ->删除一个对象，默认最后一个；

4、remove() ->删除指定的第一个匹配项；

5、insert() ->插入对象；

6、index() ->获取索引；

元组：

1、count() ->统计在元组中出现的个数；

2、index() ->获取索引；

字典：

1、keys() ->获取所有的键；

2、pop() ->删除指定的键的键值对；

3、popitem() ->随机删除一个键值对；

4、update() ->更新字典，参数为一个字典，如果键已存在，则更改，不存在则添加；

5、setdefault() ->如果键存在则，返回该键对应的值，如果不存在，设置该键为设置的默认值，然后返回该键对应的值；

6、get() ->返回键对应的值；

7、fromkeys() ->创建字典，第一个参数为可迭代对象，每个值变成字典的键，第二个参数为每个键的默认值；

**022、Python中is运算符和==运算符的区别？**

is不仅是会判断两个对象的值是否相等，还会比较两个对象的id的值是否相同；==比较的是两个对象的值是否相同；

“is”比较的是id，”==”比较的是值；

请参考[《那些年我们踩过的那些坑》](../%E7%95%AA%E5%A4%96%E7%AF%87/%E9%82%A3%E4%BA%9B%E5%B9%B4%E6%88%91%E4%BB%AC%E8%B8%A9%E8%BF%87%E7%9A%84%E9%82%A3%E4%BA%9B%E5%9D%91.md)(番外篇里)。

在讲is和==这两种运算符区别之前，首先要知道Python中对象包含的三个基本要素，分别是：id(身份标识)、[python type()](https://www.iplaypy.com/jichu/type.html)(数据类型)和value(值)。is和==都是对对象进行比较判断作用的，但对对象比较判断的内容并不相同。下面来看看具体区别在哪。

Python中比较两个对象是否相等，一共有两种方法，简单来说，它们的区别如下：

is是比较两个引用是否指向了同一个对象（引用比较）。is也被叫做同一性运算符，这个运算符比较判断的是对象间的唯一身份标识，也就是id是否相同；

==是比较两个对象是否相等。==是[python标准操作符](https://www.iplaypy.com/jichu/symbol.html)中的比较操作符，用来比较判断两个对象的value(值)是否相等；

**总结**

==比较操作符：用来比较两个对象是否相等，value做为判断因素； is同一性运算符：比较判断两个对象是否相同，id做为判断因素。

**实现原理**

is比较的是两者是否是同一个对象，所以比较的是内存地址（id是否相同）。

==是值比较。不可变对象，例如int，str，它会直接进行值比较。对于Python已知的对象，会调用他们的**eq**函数来比较。（其实已知的对象应该也是通过内置的**eq**函数来比较的）。对于自定义的对象，如果实现了**eq**函数，就会使用它比较，如果没有实现，效果和==是一样的。

Python会对比较小的对象缓存，下次用到比较小的对象时，会去缓存区查找，如果找到，不会再开辟新的内存，而是继续把小对象的地址赋给新的值。

**023、1，2，3，4，5能组成多少个互不相同且不重复的三位数？**

排列组合问题： 5\*4\*3=60个

**024、什么是反射，以及应用场景？**

在绝大多数语言当中都有反射机制的存在，可以用字符串的方式去访问对象的属性，调用对象的方法（但是不能去访问方法），python中一切皆对象，都可以使用反射。

[什么是反射的解释](https://www.cnblogs.com/IT-Scavenger/p/9394306.html)：

1、反射就是通过字符串的形式，导入模块；通过字符串的形式，去模块寻找指定函数，并执行。利用字符串的形式去对象（模块）中操作（查找/获取/删除/添加）成员，一种基于字符串的事件驱动！（反射其实就是利用字符串的对对象（模块）中的成员进行查找、获取、删除、添加操作。）；

应用场景：当我们动态的输入一个模块名的时候就可以使用到反射（动态获取对象的属性信息，及动态导入模块）；

1）、反射机制是很多框架的基石，如Django。

2、通过hasattr，getattr，delattr，setattr四个函数来操作；

**025、详述在python中如何拷贝一个对象，简述python的引用、赋值、深浅拷贝？**

引用和copy(),deepcopy()的区别要注意。

1、赋值（=），就是创建了对象的一个新的引用，修改其中任意一个变量都会影响到另一个。

2、浅拷贝：创建一个新的对象，但它包含的是对原始对象中包含项的引用（copy模块的copy()函数）；

3、深拷贝：创建一个新的对象，并且递归的复制它所包含的对象（修改其中一个，另外一个不会改变）（copy模块的deep.deepcopy()函数）；

例子：

import copy

a = [1, 2, 3, 4, ['a', 'b']] #原始对象

b = a #赋值，传对象的引用

c = copy.copy(a) #对象拷贝，浅拷贝

d = copy.deepcopy(a) #对象拷贝，深拷贝

a.append(5) #修改对象a

a[4].append('c') #修改对象a中的['a', 'b']数组对象

print 'a = ', a

print 'b = ', b

print 'c = ', c

print 'd = ', d

输出结果：

a = [1, 2, 3, 4, ['a', 'b', 'c'], 5]

b = [1, 2, 3, 4, ['a', 'b', 'c'], 5]

c = [1, 2, 3, 4, ['a', 'b', 'c']]

d = [1, 2, 3, 4, ['a', 'b']]

首先深拷贝和浅拷贝都是对象的拷贝，都会生成一个看起来相同的对象，他们本质的区别是拷贝出来的对象的地址是否和原对象一样，也就是地址的复制还是值的复制的区别。

copy():浅拷贝指仅仅拷贝数据集合的第一层数据，是对一个对象父级（外层）的拷贝，并不会拷贝子级（内部）；

deepcopy()：深拷贝指拷贝数据集合的所有层，深拷贝对一个对象是所有层次的拷贝（递归），内部和外部都会被拷贝过来。

使用浅拷贝的时候，分为两种情况：

1、如果最外层的数据类型是可变的，比如说列表，字典等，浅拷贝会开启新的地址空间去存放。

2、如果最外层的数据类型是不可变的，比如元组，字符串等，浅拷贝对象的时候，还是引用对象的地址空间。

深拷贝也分两种情况：

1、最外层数据类型可变。这个时候，内部和外部的都会拷贝过来。

2、最外层数据类型不可变，如果里面是可变数据类型，会新开辟地址空间存放。如果内部数据类型不可变，才会如同浅拷贝一样，是对地址的引用。

浅拷贝只是对另外一个变量的内存地址的拷贝，这两个变量指向同一个内存地址的变量值。

浅拷贝的特点：

1、共用一个值；

2、这两个变量的内存地址一样；

3、对其中一个变量的值改变，另外一个变量的值也会改变；

深拷贝是一个变量对另外一个变量的值拷贝；

深拷贝的特点：

1、两个变量的内存地址不同；

2、两个变量各有自己的值，且互不影响；

3、对其任意一个变量的值的改变不会影响另外一个；

4、如果是不可变类型，则深浅拷贝只拷贝引用，如果是可变类型，浅拷贝只拷贝第一层引用，深拷贝无论多少层引用都拷贝；

**025、简述赋值、深浅拷贝？**

面试答：一层两层的说。

直接赋值：其实就是对象的引用（别名）。

浅拷贝(copy)：拷贝父对象，不会拷贝对象的内部的子对象。

深拷贝(deepcopy)： copy 模块的 deepcopy 方法，完全拷贝了父对象及其子对象。

拷贝就是一个变量的值传给另外一个变量。在python 中 id()方法可以查看存放变量的内存地址，这为我们下面理解深浅copy提供了便利。

浅拷贝是指把存放变量的地址值传给被赋值，最后两个变量引用了同一份地址；

深拷贝是指被赋值的变量开辟了另一块地址用来存放要赋值的变量的值（内容）。在python中引用copy模块，copy模块中有deepcopy()方法，调用它完成变量的深copy，观察变量地址。

Python当中对于拷贝，分为两种类型。一种是数字和字符串，另一种就是列表、元组、字典等其他类型了。

**1、赋值：**

结论：对于通过用 = 号赋值，数字和字符串在内存当中用的都是同一块地址。对于赋值，字典、列表、元组等其他类型用的内存地址不会变化。

**2、浅拷贝：**

结论：对于浅拷贝，数字和字符串在内存当中用的也是同一块地址。所以对于浅拷贝，字典、列表、元组等类型，它们只拷贝第一层地址。

**3、深拷贝：**

所以综上所述，对于数字和字符串的赋值、浅拷贝、深拷贝在内存当中用的都是同一块地址。对于深拷贝，字典、列表、元组等类型，它里面嵌套多少层，就会拷贝多少层出来，但是最底层的数字和字符串地址不变。

**026、python是如何进行内存管理的？python的程序会内存泄漏吗？说说有没有什么方面阻止或者检测内存泄漏？**

* python是如何进行内存管理的[参考文章](https://blog.csdn.net/u010967872/article/details/80301633)

**python有自动垃圾回收机制（当对象的引用计数为零时解释器会自动释放内存），出现内存泄露的场景一般是扩展库内存泄露或者循环引用（还有一种是全局容器里的对象没有删除）。**

**Python解释器内核采用内存池方式管理物理内存，当创建新对象时，解释器在预先申请的物理内存块上分配相应的空间给对象使用，这样可以避免频繁的分配和释放物理内存。那么这些内存在什么时候释放呢？这涉及到Python对象的引用计数和垃圾回收。**

**说道这里可能大家会有疑问, python作为动态类型语言同时拥有垃圾回收机怎么会有内存泄露? Python也会内存泄露，Python本身的垃圾回收机制无法回收重写了del的循环引用的对象。例子：在Python程序里，内存泄漏是由于一个长期持有的对象不断的往一个dict或者list对象里添加新的对象, 而又没有即时释放，就会导致这些对象占用的内存越来越多，从而造成内存泄漏。另外，对象的交叉引用也会造成内存无法释放的问题。**

**可能出现内存泄露的情况, 有如下几种:**

**1、对象一直被全局变量所引用, 全局变量生命周期长.**

**2、垃圾回收机制被禁用或者设置成debug状态, 垃圾回收的内存不会被释放。也是非常罕见的内存泄露的方式就是今天遇到的问题, 我周旋这个问题两天才debug出来, 现在分享给大家.客官请您继续往下看；**

**说到查看python内存泄露的工具, 其实有挺多, 现在简短介绍一下：**

**1、gc: python 内置模块, 函数少功能基本, 使用简单, 作为python开发者里边的内容必须过一遍；**

**2、objgraph: 可以绘制对象引用图, 对于对象种类较少, 结构比较简单的程序适用, 我这个一个库套一个库, 内存还用的这么多；**

**3、guppy: 可以对堆里边的对象进行统计, 算是比较实用；**

**4、pympler: 可以统计内存里边各种类型的使用, 获取对象的大小；**

**说说有没有什么方面阻止或者检测内存泄漏？**

**1、程序员管理好每个python对象的引用，尽量在不需要使用对象的时候，断开所有引用；**

**2、尽量少通过循环引用组织数据，可以改用weakref做弱引用或者用id之类的句柄访问对象；**

**3、通过gc模块的接口可以检查出每次垃圾回收有哪些对象不能自动处理，再逐个逐个处理；**

**027、内存里的东西是退出python之后全都消失吗？**

**面试时本人回答：不一定，有些相互引用了的东西不会被回收；**

**百度答案是否定的。1、那些具有对象循环引用或者全局命名空间引用的变量，在 Python 退出是往往不会被释放；2、另外不会释放 C 库保留的部分内容。例如：Python退出，理论上就会释放资源占有的内存，但是Python中也会出现僵尸进程，这样就会出现有些资源无法释放，那样有些内存就无法释放。**

**除了Python，对于任何的程序，在退出时操作系统都会负责回收相关的资源。也就是说，即使解释器没有回收内存，操作系统也会回收。当然例外也是有的，比如你在程序里启动了一个新进程。在Python退出时这个新进程不一定退出，操作系统也就不会回收这个新进程相关的资源。这对于任何语言同样适用的，不过我想这应该也不在讨论范围以内。**

**扩展知识点：**

1、那些具有对象循环引用或者全局命名空间引用的变量，在 Python 退出是往往不会被释放；

2、另外不会释放 C 库保留的部分内容；

先给大家普及一波关于python关于内存分配释放的知识：

关于Python中的内存释放问题：

首先就不得不提到Python解释器在何种情况下会释放变量的内存。Python引用了内存计数这一简单的计数来控制。

下面是引用计数的知识：

增加引用计数 当对象被创建并（将其引用）赋值给变量时，该对象的引用计数被设置为1。

对象的引用计数增加的情况：

1、对象被创建：x = 3.14

2、另外的别名被创建：y = x

3、被作为参数传递给函数（新的本地引用）：foobar(x)

4、成为容器对象的一个元素：myList = [123, x, ‘xyz’]

减少引用计数 对象的引用计数减少的情况：

1、一个本地引用离开了其作用范围。如foobar()函数结束时；

2、对象的别名被显式销毁：del y；

3、对象的一个别名被赋值给其他对象：x = 123；

4、对象被从一个窗口对象中移除：myList.remove(x)；

5、窗口对象本身被销毁：del myList；

6、del语句 Del语句会删除对象的一个引用；

python的程序有两种退出方式：os.\_exit()， sys.exit()。本文介绍这两种方式的区别和选择。

1、os.\_exit()会直接将python程序终止，之后的所有代码都不会继续执行。

2、sys.exit()会引发一个异常：SystemExit，如果这个异常没有被捕获，那么python解释器将会退出。如果有捕获此异常的代码，那么这些代码还是会执行。

**区别：**

**综上，sys.exit()的退出比较优雅，调用后会引发SystemExit异常，可以捕获此异常做清理工作。os.\_exit()直接将python解释器退出，余下的语句不会执行。**

**一般情况下使用sys.exit()即可，一般在fork出来的子进程中使用os.\_exit()**

**026、python的垃圾回收机制？**

Python GC主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率（概述：python采用的是引用计数机制为主，标记-清除和分代回收（收集）两种机制为辅的策略；），此外还有小整数池。

**01、垃圾回收：**当内存中有不再使用的部分时，垃圾收集器就会把他们清理掉。他会去检查那些引用计数为0的对象，然后清除其在内存中的空间。当然除了引用计数为0的会被清除，还有一种情况也会被垃圾收集器清掉，当两个对象相互引用时，他们本身其他引用已经为0了。

**02、引用计数：**每当新的引用指向该对象时，引用计数加1，当对该对象的引用失效时，引用计数减1，当对象的引用计数为0时，对象被回收。缺点是，需要额外的空间来维护引用计数，并且无法解决对象的循环引用）。Python内部使用引用计数，来保持追踪内存中的对象，Python内部记录了对象有多少个引用，就是引用计数，当对象被创建时就创建了一个引用计数，当对象不再需要的时候，这个对象的引用计数为0时，他被垃圾回收；

PyObject是每个对象必有的内容，其中ob\_refcnt就是做为引用计数。当一个对象有新的引用时，它的ob\_refcnt就会增加，当引用它的对象被删除，它的ob\_refcnt就会减少.引用计数为0时，该对象生命就结束了。

优点:

1、简单；

2、实时性；

缺点:

1、维护引用计数消耗资源；

2、循环引用；

**03、标记清除：** 标记-清除机制（活动对象会被打上标记，会把那些没有被打上标记的非活动对象进行回收）；解决循环引用的问题。先按需分配，等到没有空闲内存的时候，从寄存器和程序栈上的引用出发，遍历所有对象和引用把所有能访问的打标记，最后将没有标记的对象释放掉；

基本思路是先按需分配，等到没有空闲内存的时候从寄存器和程序栈上的引用出发，遍历以对象为节点、以引用为边构成的图，把所有可以访问到的对象打上标记，然后清扫一遍内存空间，把所有没标记的对象释放。

**04、分代回收技术**（以时间换空间的回收方式）：分代回收的整体思想是：提高效率，提高垃圾回收的效率，按照存活时间，将系统中的所有内存块根据其存活时间划分为不同的集合，每个集合就成为一个“代”，垃圾收集频率随着“代”的存活时间的增大而减小，存活时间通常利用经过几次垃圾回收来度量。

Python默认定义分代对象集合，引用数越大，对象的存活时间越长；

Python默认定义了三代对象集合，索引数越大，对象的存活时间越长。

举例：  
当某些内存块M经过了3次垃圾收集的清洗之后还存活时，我们就将内存块M划到一个集合A中去，而新分配的内存都划分到集合B中去。当垃圾收集开始工作时，大多数情况都只对集合B进行垃圾回收，而对集合A进行垃圾回收要隔相当长一段时间后才进行，这就使得垃圾收集机制需要处理的内存少了，效率自然就提高了。在这个过程中，集合B中的某些内存块由于存活时间长而会被转移到集合A中，当然，集合A中实际上也存在一些垃圾，这些垃圾的回收会因为这种分代的机制而被延迟。

讲一下垃圾回收机制 答：引用计数，相互引用，分代回收。简单来说python的内存管理机制有三种

1）引用计数

2）垃圾回收。①  引用计数，②  标记清除，③  分代回收。

3）内存池

Python 使用标记清除（mark-sweep）算法和分代收集（generational），来启用针对循环引用的自动垃圾回收。  
先来看标记清除算法。我们先用图论来理解不可达的概念。对于一个有向图，如果从一个节点出发进行遍历，并标记其经过的所有节点；那么，在遍历结束后，所有没有被标记的节点，我们就称之为不可达节点。显而易见，这些节点的存在是没有任何意义的，自然的，我们就需要对它们进行垃圾回收。  
当然，每次都遍历全图，对于 Python 而言是一种巨大的性能浪费。所以，在 Python 的垃圾回收实现中，标记清除算法使用双向链表维护了一个[数据结构](http://c.biancheng.net/data_structure/)，并且只考虑容器类的对象（只有容器类对象才有可能产生循环引用）。  
而分代收集算法，则是将 Python 中的所有对象分为三代。刚刚创立的对象是第 0 代；经过一次垃圾回收后，依然存在的对象，便会依次从上一代挪到下一代。而每一代启动自动垃圾回收的阈值，则是可以单独指定的。当垃圾回收器中新增对象减去删除对象达到相应的阈值时，就会对这一代对象启动垃圾回收。  
事实上，分代收集基于的思想是，新生的对象更有可能被垃圾回收，而存活更久的对象也有更高的概率继续存活。因此，通过这种做法，可以节约不少计算量，从而提高 Python 的性能。

内存池机制

Python提供了对内存的垃圾收集机制，但是它将不用的内存放到内存池而不是返回给操作系统。

Python中所有小于256个字节的对象都使用pymalloc实现的分配器，而大的对象则使用系统的 malloc。另外Python对象，如整数，浮点数和List，都有其独立的私有内存池，对象间不共享他们的内存池。也就是说如果你分配又释放了大量的整数，用于缓存这些整数的内存就不能再分配给浮点数。

一 、什么是内存管理和垃圾回收

Python GC主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率。

现在的高级语言如java，c#等，都采用了垃圾收集机制，而不再是c，c++里用户自己管理维护内存的方式。自己管理内存极其自由，可以任意申请内存，但如同一把双刃剑，为大量内存泄露，悬空指针等bug埋下隐患。

对于一个字符串、列表、类甚至数值都是对象，且定位简单易用的语言，自然不会让用户去处理如何分配回收内存的问题。

python里也同java一样采用了垃圾收集机制，不过不一样的是:

最关键的一句话：

python采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的策略

四、垃圾回收与性能调优

1.手动垃圾回收

2.调高垃圾回收阈值  
3.避免循环引用（手动解循环引用和使用弱引用）

**027、python的可变类型和不可变类型的区别？**

这里的可变不可变是指内存中的那块内容是否可以被改变。 可变类型有：列表，字典；不可变类型有：数字，字符串，元组；

**028、阐述列表、集合、字典的底层实现原理？**

**列表实现可以是数组和链表。** 顺序表是怎么回事？顺序表一般是数组。

列表是一个线性的集合，它允许用户在任何位置插入、删除、访问和替换元素。

列表实现是基于数组或基于链表结构的。当使用列表迭代器的时候，双链表结构比单链表结构更快。

有序的列表是元素总是按照升序或者降序排列的元素。 实现细节 python中的列表的英文名是list，因此很容易和其它语言(C++, Java等)标准库中常见的链表混淆。

事实上CPython的列表根本不是列表（可能换成英文理解起来容易些：python中的list不是list）。在CPython中，列表被实现为长度可变的数组。 从细节上看，Python中的列表是由对其它对象的引用组成的连续数组。指向这个数组的指针及其长度被保存在一个列表头结构中。这意味着，每次添加或删除一个元素时，由引用组成的数组需要该标大小（重新分配）。幸运的是，Python在创建这些数组时采用了指数分配，所以并不是每次操作都需要改变数组的大小。但是，也因为这个原因添加或取出元素的平摊复杂度较低。 不幸的是，在普通链表上“代价很小”的其它一些操作在Python中计算复杂度相对过高。

利用 list.insert(i,item) 方法在任意位置插入一个元素——复杂度O(N) 利用 list.pop(i) 或 list.remove(value) 删除一个元素——复杂度O(N)；

列表的算法效率可以采用时间复杂度来衡量：

index() O(1) 、append O(1) 、 pop() O(1) 、pop(i) O(n) 、insert(i,item) O(n) 、del operator O(n) 、iteration O(n) 、contains(in) O(n) 、get slice[x:y] O(k) 、del slice O(n) 、set slice O(n+k) 、reverse O(n) 、concatenate O(k) 、sort O(nlogn) 、multiply O(nk)；

O括号里面的值越大代表效率越低

list和tuple在c实现上是很相似的，对于元素数量大的时候，都是一个数组指针，指针指向相应的对象，找不到tuple比list快的理由。  
但对于小对象来说，tuple会有一个对象池，所以小的、重复的使用tuple还有益处的。  
为什么要有tuple，还有很多的合理性。  
实际情况中的确也有不少大小固定的列表结构，例如二维地理坐标等；  
另外tuple也给元素天然地赋予了只读属性。  
认为tuple比list快的人大概是把python的tuple和list类比成C++中的数组和列表了。

**在Python中，字典是通过散列表或说哈希表实现的。**

字典也被称为关联数组，还称为哈希数组等。也就是说，字典也是一个数组，但数组的索引是键经过哈希函数处理后得到的散列值。哈希函数的目的是使键均匀地分布在数组中，并且可以在内存中以O(1)的时间复杂度进行寻址，从而实现快速查找和修改。哈希表中哈希函数的设计困难在于将数据均匀分布在哈希表中，从而尽量减少哈希碰撞和冲突。由于不同的键可能具有相同的哈希值，即可能出现冲突，高级的哈希函数能够使冲突数目最小化。Python中并不包含这样高级的哈希函数，几个重要（用于处理字符串和整数）的哈希函数是常见的几个类型。

通常情况下建立哈希表的具体过程如下：

字典（dict）数据添加：把key通过哈希函数转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将value存储在以该数字为下标的数组空间里。

字典（dict）数据查询：再次使用哈希函数将key转换为对应的数组下标，并定位到数组的位置获取value。

哈希函数就是一个映射，因此哈希函数的设定很灵活，只要使得任何关键字由此所得的哈希函数值都落在表长允许的范围之内即可。本质上看哈希函数不可能做成一个一对一的映射关系，其本质是一个多对一的映射，这也就引出了下面一个概念–哈希冲突或者说哈希碰撞。哈希碰撞是不可避免的，但是一个好的哈希函数的设计需要尽量避免哈希碰撞。

介绍python中字典和集中时提到，字典和集合的优势在于增删改查非常高效，而高效的原因就在于，python中字典和集合这两个数据结构是基于散列表来实现的。散列表也叫哈希表，英文是hashtable，是一种非常高效的基础数据结构。

今天我们介绍一下散列表的原理，了解了原理之后，就会很容易理解为什么字典和集合的增删改查如此高效。

什么是散列表 散列表听上去好像很玄乎，但是实际上在我们的生活中却非常的常见。举个很简单的例子，我们读大学的时候，辅导员老师那里的excel表里有每个学生的信息，如果我们想要查找某一个学生的信息，使用什么方法最迅速呢？ 一种很容易想到的方法是打开excel,然后从第一行开始一行一行的看，直接看到我们想要找的那个学生的那一行为止，可想而知，如果有5000个学生，很不巧这名学生刚好在excel表的最后一行，是不是要耗很长时间才能找到。这种方法有点类似于列表，查找的时间复杂度为O(n)。 还有另外的方法，因为每个学生都有学号，并且学号是连续的，如果我们存储的时候，直接按学号从小到大的顺序，从第一行开始存，查找的时候，只要知道学生的身份证号，是不是就能很快速的定位到该名学生在excel表格里的第几行，然后就可以直接拿到学生的信息。使用这种方式，只需要经过一次计算，就能确定学生信息的位置，而这个计算的时间是相对固定的。这就是典型的一种散列表的思想，它查找的时间复杂度是O(1)。这个计算偏移的方法，就相当于散列函数。这里学生的学号就是散列表中的键(key)。 用图表示就像下面这样：

我们只需要设计一个散列函数，然后利用这个散列函数，根据学生的学号就可以计算出每个学生信息在excel表里的存储位置。这样就可以实现快速的增删改查，效率非常的高。

以上就是散列表所包含的非常基础的思想。

当然实际的散列表的实现，还要考虑的因素有很多，比如：hash函数的设计，这个函数要求对同样的输入，输出一定要相同，而对于不同的输入，输出尽量减少冲突 如果出现散列冲突(即两个Key经过散列函数之后输出相同)，使用什么方式解决散列冲突装载因子的设置，因为散列表是可以增加元素的，为了避免空间满时减加元素时元素搬移太耗时，需要为散列表预留一些空间，当表中元素达到某个域值时，对整个散列表进行整体搬移和扩容，这个域值就是装载因子如果想更深入的了解散列表，可以自行查阅一些资料，在这里，我们只需要了解散列表的实现原理，以及为什么散列表的增删改查效率如此之高就可以了。

**为什么字典和集合是无序的？**了解了上面所说的散列表的基本原理之后，我们知道，散列表中元素的实际存储位置是由所设计的散列函数对键(key)进行运算后得出的。上面所举的学号的例子比较特殊，使用的散列函数相当于只是对学号数字取了一个偏移来得到学生信息存储位置。而实际上大多数的散列函数在对键进行计算后，得到的存储位置是随机的，并不连续，所以元素的存储位置也就不一定和输入的顺序相同。

**列表可以做为键(key)吗？**讨论这个问题前，我们需要知道字典的键必须要求是可以散列的，而一个可散列的对象必须满足以下要求：

支持hash()函数，并且通过hash()方法所得到的散列值是不变的 支持通过eq()方法检测相等性 若a==b为真，则hash(a) == hask(b)也必须为真 通过下面代码测试一下：

1l = [1,2,3,4] 2print(hash(l)) 运行会报如下错误:

1Traceback (most recent call last): 2 File "<stdin>", line 1, in <module> 3TypeError: unhashable type: 'list' 实际上，list并没有实现hash()这个魔法函数，因为只有不可变对象，才可以进行hash, 像字符串，字典这种都属于不可变对象，而集合，列表，字典都属于可变对象，所以它们无法作为字典的key。

**028、求字典的方法dict.fromkeys（）下面代码的结果？**

v=dict.fromkeys(['k1','k2'],[])  
v['k1'].append(666)  
print(v)  
v['k1']=777  
print(v)

* 结果： {'k1': [666], 'k2': [666]} {'k1': 777, 'k2': [666]}

**029、一行代码实现删除列表中的所有的重复的值？**

lis=[1,1,2,1,22,5]  
lis=list(set(lis))

**030、如何实现"1.2.3"变成['1','2','3']?**

s="1,2,3"  
s=s.split(',')

**031、如何实现['1','2','3']变成[1,2,3]？**

ss=['1', '2', '3']  
ss=[int(i) for i in ss]

**032、比较：a=[1,2,3]和b=[(1),(2),(3)]以及c=[(1,),(2,),(3,)]的区别？**

a和b的结果相同，列表里面的值相同，类型也相同；c中的列表里面的值是元组类型的；

**033、如何用一行代码生成[1,4,9,16,25,36,49,64,81,100]?**

lis=[i\*\*2 for i in range(1,11)]

**034、常用字符串格式化有哪几种？**

字符串格式化:%和.format。

<http://stackoverflow.com/questions/5082452/python-string-formatting-vs-format>

1. 使用百分号

print('hello %s and %s'%('friend','another friend'))

1. 使用format

print('hello {first} and {second}'.format(first='friend',second='another friend'))

.format在许多方面看起来更便利。对于%最烦人的是它无法同时传递一个变量和元组.你可能会想下面的代码不会有什么问题:

"hi there %s" % name；

但是,如果name恰好是(1,2,3),它将会抛出一个TypeError异常.为了保证它总是正确的,你必须这样做:

"hi there %s" % (name,) # 提供一个单元素的数组而不是一个参数

但是有点丑..format就没有这些问题.你给的第二个问题也是这样,.format好看多了.

**你为什么不用它?**

不知道它(在读这个之前)；

为了和Python2.5兼容(譬如logging库建议使用%([issue #4](https://github.com/taizilongxu/interview_python/issues/4)))；

**035、什么是断言(assert)?应用场景？**

assert是用来检查一个条件，如果它为真，就不做任何事。如果它为假，则会抛出AssertError并且包含错误信息。

应用场景：

1、防御型编程；

2、运行时检查程序逻辑；

3、检查约定；

4、程序常量；

5、检查文档；

**036、有两个字符串列表a和b，每个字符串是由逗号隔开的一些字符？**

a=[  
   'a,1',  
   'b,3,22',  
   'c,3,4',  
]  
b=[  
   'a,2',  
   'b,1',  
   'd,5',  
]  
# 按照a，b中每个字符串的第一个值，合并成c如下：  
c=[  
   'a,1,2',  
   'b,3,22,1',  
   'c,3,4',  
   'd,5'  
]

# 解法：  
​  
a=[  
   'a,1',  
   'b,3,22',  
   'c,3,4',  
]  
b=[  
   'a,2',  
   'b,1',  
   'd,5',  
]  
a\_dic={}  
for s in a:  
   k,v = s.split(',',1)  
   a\_dic[k]=v  
b\_dic={}  
for s in b:  
   k,v = s.split(',',1)  
   b\_dic[k]=v  
c\_dic=a\_dic  
for k,v in b\_dic.items():  
   if k in c\_dic:  
       c\_dic[k]=','.join([c\_dic[k],v])  
   else:  
       c\_dic[k]=v  
c=[','.join([k,c\_dic[k]]) for k in c\_dic]  
print(c)

**037、有一个多层嵌套的列表A=[1,2,3,[4,1,['j1',1,[1,2,3,'aa']]]],请写一段代码将A中的元素全部打印出来？**

A=[1,2,3,[4,1,['j1',1,[1,2,3,'aa']]]]  
def my\_print(lis):  
   for i in lis:  
       if type(i)==list:  
           my\_print(i)  
       else:  
           print(i)  
my\_print(A)

**038、a=range(10),则a[::-3]的值是？**

[9,6,3,0] 或者 range(9,-1,-3)

**039、将下面列表中的元素根据位数合并成字典？**

lst = [1,2,4,8,16,32,64,128,256,512,1024,32769,65536,4294967296]  
# 结果  
{1: [1, 2, 4, 8], 2: [16, 32, 64], 3: [128, 256, 512], 4: [1024], 5: [32769, 65536], 10: [4294967296]}  
lst = [1,2,4,8,16,32,64,128,256,512,1024,32769,65536,4294967296]  
dic={}  
for i in lst:  
   len\_i=len(str(i))  
   dic.setdefault(len\_i,[]).append(i)  
print(dic)

**040、用尽量简洁的方法将二维数组合并成一维数组？**

lst = [[1,2,3],[4,5,6],[7,8,9]]  
ll=[]  
for l in lst:  
   # ll+=l  
   ll.extend(l)  
print(ll)

**041、将列表按照下列规则排序？**

1、正数在前，负数在后；

2、正数从小到大；

3、负数从大到小；

例子：

排序前：[7,-8,5,4,0,-2,-5]；

排序后：[0, 4, 5, 7, -2, -5, -8]；

代码实现：

lis = [7,-8,5,4,0,-2,-5]  
lis=sorted(lis,key=lambda x:(x<0,abs(x))) # 这里排序条件返回元组，先比较第一个，再第二个值  
print(lis)

**042、解决哈希冲突的算法有哪几种？分别有什么特点？**

[哈希冲突参考](https://blog.csdn.net/seulzz/article/details/77163878)：

1、开放定址法；

2、再哈希法；

3、链地址法；

4、建立公共溢出区；

**043、简述python字符串的驻留机制？**

[python字符串驻留机制参考文档](https://www.cnblogs.com/asmer-stone/p/4802800.html)

1、相同对象的引用都指向内存中的同一个位置，这个也叫python的字符串驻留机制；

2、python的引用计数机制，并不是对所有的数字，字符串，他只对”[0-9][a-z][A-Z] 和"\_"(下划线)  ”有效“，当字符串中由其他字符比如“！ @ # ￥ % -”时字符驻留机制是不起作用的。

**044、以下代码索引超出列表范围输出什么？**

lis=['a','b','c','d','e']  
print(lis[10:])

* 答案：[]

**045、python哪些类型的数据才能作为字典的key？**

可哈希的类型；

**046、有如下深浅拷贝代码，写出它的执行结果？**

import copy  
a=[1,2,3,[4,5],6]  
b=a  
c=copy.copy(a)  
d=copy.deepcopy(a)  
b.append(10)  
c[3].append(11)  
d[3].append(12)

求a，b，c，d

答案：

a：[1, 2, 3, [4, 5, 11], 6, 10] b：[1, 2, 3, [4, 5, 11], 6, 10] c：[1, 2, 3, [4, 5, 11], 6] d：[1, 2, 3, [4, 5, 12], 6]

**046、对字典d={'a':30,'g':17,'b':25,'c':18,'d':50,'e':36,'f':57,'h':25}按照value字段进行排序？**

d={'a':30,'g':17,'b':25,'c':18,'d':50,'e':36,'f':57,'h':25}

dd=sorted(d.items(),key=lambda x:x[1])

print(dd)

**047、找出两个列表中相同的元素和不同的元素？**

list1=[1,2,3,5,8,7,11,10]

list2=[5,15,25,10]

sim=[i for i in list1 if i in list2]

diff=[i for i in list1+list2 if i not in sim]

print(sim)

print(diff)

**048、二叉树是非线性结构，栈和队列以及线性表都是线性结构，对吗？**

对的

**049、从0-99这100个数中随机取出10个，要求不能重复？**

import random

lis=random.sample(range(0,100),10)

print(lis)

**050、有一个列表lis=['This','is','a','Man','B','!']，对它进行大小写无关的排序？**

lis=['This','is','a','Man','B','!']

lis=sorted(lis,key=str.lower)

print(lis)

**051、描述下字典的items()方法和iteritems()方法有啥不同？**

字典的items方法作用：是可以将字典中的所有项，以列表方式返回。因为字典是无序的，所以用items方法返回字典的所有项，也是没有顺序的。

字典的iteritems方法作用：与items方法相比作用大致相同，只是它的返回值不是列表，而是一个迭代器；

在Python3.x 里面没有iteritems，iteritems()方法已经废除了。在3.x里用 items()替换iteritems() ，可以用于 for 来循环遍历；

在python2中items和iteritems大致相同，items()返回[(key,value)]的列表（对象），iteritems()返回迭代器（对象），iteritems()循环时不可以增删dict的内容。items方法将所有的字典以列表方式返回，其中项在返回时没有特殊的顺序；iteritems方法有相似的作用，但是返回一个迭代器对象；

**052、请列举你所知道的python代码检测工具以及他们之间的区别？**

pylint --- 源代码分析器，可以分析python代码中的错误;

pyflakes --- 检查源文件错误的简单程序，不会检查代码风格;

pep8 --- 检查代码规范的工具;

**053、介绍一下try except的用法和作用？**

主要用来处理异常；

完整用法如下：

try:  
     Normal execution block  
except A:  
     Exception A handle  
except B:  
     Exception B handle  
except:  
     Other exception handle  
else:  
     if no exception,get here  
finally:  
     print("finally")

**054、全局异常捕获？**

面试答：。。。。。

Python全局未知异常捕获；

log日志监测程序运行：

1、Linux重定向将python运行报错记录到log日志中；

2、或者用python内置的logging模块。

但是在用logging模块时，try...except...可以捕获到可预见/考虑到的异常，用logging记录到log日志文件中，但是在对于未知异常或导致程序退出时的情形无法记录到log日志中，用Linux的重定向可以捕捉到。

现在到问题时我不想用Linux系统提供的重定向方法，我需要捕获python运行中未知的异常并记录到log日志中。

推荐使用sys.excepthook。详细参考这篇文章，简洁明了。

也可以用try...except...包裹全部代码，但我觉得这样不够优雅。

当要捕获所有异常的时候，推荐使用：

#推荐写法 可以捕获除与程序退出sys.exit()相关之外的所有异常。

try:

:

except Exception as e:

# error occurred, log 'e', etc.

#不推荐使用以下这种

会捕获所有异常，包括键盘中断和程序退出请求（用sys.exit()就无法退出程序了，因为异常被捕获了），因此慎用。

try:

:

except:

# error occurred, log 'e', etc.

老版本的Python，except语句写作"except Exception, e"，

Python 2.6后应写作"except Exception as e"。

**054、阅读以下列表list.remove()方法代码，写输出结果？**

lis = [2,4,5,6,7]  
for i in lis:  
   if i % 2==0:  
       lis.remove(i)  
print(lis)

* 结果：[4, 5, 7]

**055、对列表[3,1,-4,-2]按照绝对值排序？**

lis=[3,1,-4,-2]

lis=sorted(lis,key=lambda x:abs(x))

print(lis)

**056、获取python解释器版本的方法？**

终端执行:python -V

**057、现有mydict和变量onekey，请写出从mydict中取出onekey的值的方法？**

方法一：mydict[onekey] 这种方法，如果mydict中键不存在的时候程序会报错；

方法二：mydict.get(onekey) 这种方法，如果存在，返回值，不存在返回None；

方法三：mydict.setdefault(onekey,[]) 这种方法：存在的话返回值，不存在的时候创建一个值，值得内容为第二个参数+；

**058、列表中保留顺序和不保留顺序去重？**

* 不保留顺序

lis=[3, 1, 4, 2, 3]

print(list(set(lis)))

* 保留顺序

lis=[3, 1, 4, 2, 3]

T=[]

[T.append(i) for i in lis if i not in T])

print(T)

# 或者

T=sorted(set(lis), key=lis.index)

print(T)

**059、在什么情况下y!=x-(x-y)会成立？**

x，y是两个不相等的非空集合;

**060、实现99乘法表（使用两种方法）？**

print('\n'.join(['\t'.join(['{}\*{}={}'.format(x,y,x\*y) for x in range(1,y+1)]) for y in range(1,10)]))

for i in range(1,10):

for j in range(1,i+1):

print('%s\*%s=%s'%(i,j,i\*j),end='\t')

else:

print()

**061、判断dict中有没有某个key？**

key in dict.keys() 判断;

**062、a = dict(zip(('a','b','c','d','e'),(1,2,3,4,5))) 请问a是什么？**

{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

**064、python中进制转换？**

**进制转换以十进制为媒介 十六进制前面加上0x，八进制加上0o，二进制前面加上0b**

|  | **二进制** | **八进制** | **十进制** | **十六进制** |
| --- | --- | --- | --- | --- |
| 二进制 |  | bin(int(x, 8)） | bin(int(x, 10)) | bin(int(x, 16)) |
| 八进制 | oct(int(x, 2)) |  | oct(int(x, 10)) | oct(int(x, 16)) |
| 十进制 | int(x, 2) | int(x, 8) |  | int(x, 16) |
| 十六进制 | hex(int(x, 2)) | hex(int(x, 8)) | hex(int(x, 10)) |  |

**065、将列表alist=[{'name':'a','age':25},{'name':'b','age':30},{'name':'c','age':20}]，按照age的值从大到小排列？**

alist=[{'name':'a','age':25},{'name':'b','age':30},{'name':'c','age':20}]

blist=sorted(alist,key=lambda x:x['age'],reverse=True)

print(blist)

**065、请按list1中元素的age由大到小排序？**

list1 = [{'name':'a'，'age':20}，{'name':'b'，'age':30}，{'name':'c'，'age':25}]

sorted(list1，key=lambda x:x['age']，reverse=True)

**066、关于Python程序的运行方面，有什么手段能提升性能？**

1、使用多进程，充分利用机器的多核性能；

2、对于性能影响较大的部分代码，可以使用C或C++编写；

3、对于IO阻塞造成的性能影响，可以使用IO多路复用来解决；

4、尽量使用Python的内建函数5、尽量使用局部变量；

**068、一个大小为100G的文件etl\_log.txt，要读取文件的内容，写出具体过程代码？**

with open("etl\_log.txt",'r',encoding='utf8') as f:

for line in f:

print(line,end='')

**069、python代码如何获取命令行参数？**

[获取命令行参数的方法参考](https://www.cnblogs.com/ouyangpeng/p/8537616.html)

1.使用sys模块 \* 通过sys.argv来获取；

2.使用getopt模块；

**070、写代码：如何由tuple1=('a','b','c','d','e')，和tuple2=(1,2,3,4,5)得到res={'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}？**

tuple1=('a','b','c','d','e')

tuple2=(1,2,3,4,5)

res=dict(zip(tuple1,tuple2))

print(res)

**071、1<(2==2)和1<2==2的结果分别是什么？**

第一个为False，第二个为True，暂时按照第一个类型进行相应的转换；

**072、如何打乱一个排好序的列表？**

* 使用random.shuffle()

import random  
alist=[1,2,3,4,5,6]  
random.shuffle(alist)  
print(alist)

**073、如何查找一个字符串中特定的字符？find和index的差异？**

使用find和index方法查找：

1、find()方法：查找子字符串，若找到返回从0开始的下标值，若找不到返回-1；

2、index()方法：python 的index方法是在字符串里查找子串第一次出现的位置，类似字符串的find方法，不过比find方法更好的是，如果查找不到子串，会抛出异常，而不是返回-1；

**074、把a='aaabbcccdddde'这种形式的字符串，压缩成a3b2c3d4e1这种形式？**

a='aaabbcccdddde'  
aa=''  
for i in sorted(list(set(a)),key=a.index):  
   aa=aa+i+str(a.count(i))  
print(aa)

**075、一个数如果恰好等于它的因子之和，这个数就称为‘完数’，比如6=1+2+3，编程找出1000以内的所有的完数？**

wanshu=[]  
for i in range(1,1001):  
   s=0  
   for j in range(1,i//2+1):  
       if i % j ==0:  
           s+=j  
   else:  
       if i==s:  
           wanshu.append(i)  
print(wanshu)

**076、输入一个字符串，输出该字符串的字符的所有组合。如输入'abc',输出a,b,c,ab,ac,bc,abc？**

def getC(s):  
   if not s:  
       return  
   len\_s=len(s)  
   ss=[]  
   for i in range(len\_s):  
       combination(s,0,i,ss)  
aaa=[]  
def combination(s,index,num,ss):  
   global aaa  
   if num==-1:  
       return  
   if index==len(s):  
       return  
   ss.append(s[index])  
   aaa.append(''.join(ss))  
   combination(s,index+1,num-1,ss)  
   ss.pop()  
   combination(s,index+1,num,ss)  
     
getC('123')  
print(aaa)  
print(sorted(set(aaa),key=lambda x:len(str(x))))

**077、给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。给定的字符串只含有小写英文字母，并且长度不超过10000。例如：'ababab',返回True，'ababa'，返回False？**

def solution(s):  
   ll=len(s)  
   for i in range(1,ll//2+1):  
       if ll % i == 0:  
           j=0  
           while s[:i]==s[j:j+i] and j<ll:  
               j=j+i  
           if j==ll:  
               return True  
   return False  
     
print(solution('abababa'))

**078、python递归的最大层数？**

1000

**079、filter、map、reduce的作用？**

通俗的说..都是用在一堆数据(比如一个列表)上..

filter是筛选出其中满足某个条件的那些数据..字面意思是过滤..比如挑出列表中所有奇数..

map 作用是生成一个新数组，遍历原数组，将每个元素拿出来做一些变换然后放入到新的数组中。map（） 作用：便于对数组中的每个元素进行操作。

reduce是用某种方法依次把所有数据丢进去最后得到一个结果..字面意思是化简..比如计算一个列表所有数的和的过程,就是维持一个部分和然后依次把每个数加进去..。reduce() 作用：对数组中的元素进行汇总。

1. filter() 相当于过滤器的作用:

filter(func, iterable) : 通过判断函数fun，筛选符合条件的元素。

filter函数（筛选），filter(func,iterable)用于过滤，就是把可迭代对象中的每个元素，使用func进行校验，把不满足条件的过滤掉。原则上func只返回True或者flase即可，python中非0 非空 也可表示true也可。

例01：filter(lambda x: x>3, [1,2,3,4,5,6])

>>> <filter object at 0x0000000003813828>

例02：s=[1,2,3,5,6,8,9,10,25,12,30]  
# 筛选出3的倍数  
# 第一个参数为一个返回True或者False的函数，第二个参数为可迭代对象  
# 该函数把可迭代对象依次传入第一个函数，如果为True，则筛选  
d=filter(lambda x:True if x % 3 == 0 else False,s)  
print(list(d))

1. map()函数：

map(func, \*iterable) : 将func用于每个iterable对象。

map函数（原有数据上处理，加工），map(func,iterable)函数接收两个参数：一个是函数，一个是iterable（ 列表，字符串 等 ）作用：map将传入的函数依次作用于序列的每个元素，返回一个新的序列。

例01：map(lambda a,b: a+b, [1,2,3,4], [5,6,7])

>>> [6,8,10]

# 第一个参数为函数，依次将后面的参数传给第一个函数，并执行函数；  
# 如果有多个参数则，依次将后面的对应传给参数；  
例02：s=map(lambda x,y:x+y,range(10),range(10))  
print(list(s))  
例03：ss=map(lambda x:x\*x,range(10))  
print(list(ss))

1. reduce()函数：函数会对参数序列中元素进行累积；

reduce函数（数据合并处理，得到一个处理过的结果），这个函数的运行是一个累积的过程。

reduce(func,iterable) 就是先在可迭代的对象中取前两个元素，作为参数由func进行计算，得到一个结果；然后再在可迭代对象中取第三个元素和刚才的计算结果共同作为参数传递给func 进行计算，得到一个新结果；在取第四个元素和 这个新结果再进行func运算...直到取完所有元素为止。

例01：reduce(lambda x, y: x+y, [1,2,3,4,5]) # 使用 lambda 匿名函数

>>> 15

通俗的说..都是用在一堆数据(比如一个列表)上..

例02：

from functools import reduce  
# 开始的时候将可迭代对象的第一个数和第二个数当成x和y；  
# 然后将第一次函数的执行结果当成x，然后再传入一个数当成y；  
# 再执行函数；  
s=reduce(lambda x,y:x+y,range(101))  
print(s) # 相当于0+1+2+……+99+100

**080、什么是闭包？闭包函数？**

闭包(closure)是函数式编程的重要的语法结构。闭包也是一种组织代码的结构，它同样提高了代码的可重复使用性。

**闭包：**

闭包是语法闭包的简称，是引用自由变量的函数。这个被引用的自由变量将与这个函数一同存在，即使已经离开创造它的环境也不例外。所以，闭包是由函数和与其相关的引用环境组合而成的实体。闭包是函数式编程的重要语法结构。Python也支持这一特性。

当一个内嵌函数引用其外部作用域的变量,我们就会得到一个闭包（将这个内部函数以及用到的一些变量称之为闭包）（在函数内部中可以再（嵌套）定义另一个函数时，如果内部的这个函数引用了外部的函数的变量，则可能产生闭包（也就是闭包函数））。总结一下，创建一个闭包必须满足以下几点:

1、必须有一个内嵌函数；

2、内嵌函数必须引用外部函数中的变量(嵌套函数需要引用上一级函数的变量)；

3、外部函数的返回值必须是内嵌函数；

感觉闭包还是有难度的,几句话是说不明白的,还是查查相关资料.

重点是函数运行后并不会被撤销,就像某些函数运行完后,instance实例并不被销毁,而是继续留在内存空间里.这个功能类似类里的类变量,只不过迁移到了函数上.

闭包就像个空心球一样,你知道外面和里面,但你不知道中间是什么样.

* 闭包可以用来在一个函数与一组“私有”变量之间创建关联关系。在给定函数被多次调用的过程中，这些私有变量能够保持其持久性。

# 内部函数使用了外部函数的变量，就相当于闭包；

闭包函数（closure function）指的是定义在一个函数内部的函数，被外层函数包裹着，其特点是可以访问到外层函数中的名字，如inner函数就是一个闭包函数。

def outer():

num = 1

def inner():

print(num) # 内层函数中不存在num 但可以访问到外层的num

return inner # 基于函数对象的概念我们可以将内层函数返回到外界使用，从而打破函数调用的层级限制，但无论在何处调用，作用域的嵌套关系都是以定义阶段为准的，所以外界得到的不仅仅是一个函数对象（inner），在该函数外还包裹了一层作用域，这使得该函数无论在何处调用，都是访问自己外层包裹的作用域中的名字num；

func = outer() # func == inner func指向的是inner的内存地址，但是func本身确实一个全局变量，可以在任意位置调用func，但无论在何处调用func，都需要按照定义阶段作用域的嵌套关系去查找名字；

num=1000

func() #输出结果：1

**080、装饰器是怎么回事，作用，讲一讲？**

答：

一、两大原则：

python装饰器简单来说可以在不改变某个函数内部实现和原来调用方式的前提下对该函数增加一些附件的功能，提供了对该函数功能的扩展。

二、装饰器

装饰器本质上就是函数，功能是为其他函数添加附加功能。

通俗的来讲，装饰器就是等于高阶函数+函数嵌套+闭包的综合实现。

装饰器有两大原则：第一，不修改被修饰函数的源代码；第二，不修改被修饰函数的调用方法。

3.装饰器调用顺序

使用多个装饰器修饰同个函数时，装饰器的调用顺序与语法糖@的声明顺序相反。

**081、装饰器怎么写？**

装饰器是在不修改原来函数的情况下，对函数增新的功能的一种方法；  
装饰器内可带参数；  
一个函数可以有多个装饰器；

装饰器本身是函数，是为其他函数添加功能的函数，装饰器的好处就是在不用更改原函数的代码前提下给函数增加新的功能。

装饰器原则：  
    （1）不能修改被装饰函数的源代码；  
    （2）不能修改被装饰函数的调用方式

抓住装饰器的两大原则来学习装饰器。装饰器的预备知识：装饰器 = 高阶函数 + 嵌套函数 + 闭包

实现方法

1.原始方法

import time

def deco(func):

def wrapper(\*args, \*\*kwargs):

startTime = time.time()

func(\*args, \*\*kwargs)

endTime = time.time()

msecs = (endTime - startTime)\*1000

print("time is %d ms" %msecs)

return wrapper

@deco

def func(a,b):

print("hello，here is a func for add :")

time.sleep(1)

print("result is %d" %(a+b))

2.类方法

class logging(object):

def \_\_init\_\_(self, func):

self.func = func

def \_\_call\_\_(self, \*args, \*\*kwargs):

print "[DEBUG]: enter function {func}()".format(

func=self.func.\_\_name\_\_)

return self.func(\*args, \*\*kwargs)

@logging

def say(something):

print "say {}!".format(something)

上一篇文章介绍了 装饰器的概念。现在讲一下在程序中怎么来写装饰器。上代码：

1 def X(fun):

2 def Y(b):

3 print(b)

4 fun()

5 return Y

6

7 def test():

8 print('OK')

9

10 test = X(test)

11 test(1)

前05行是一个闭包，因为内层函数的参数是外层函数的变量，而外层函数返回了内存函数的引用。

第10行，在调用函数X时，将函数test的引用（注意不是test(),没有小括号）作为参数传入，此时X(test）返回的是 Y函数的引用。所以第十行的结果就是test指向了Y函数的引用。所以，第11行的test()就是调用的Y函数的引用，同时Y函数中的fun()指向了第7行的test()函数。

第二种写法：

1 def X(fun):

2 def Y(b):

3 print(b)

4 fun()

5 return Y

6

7 @X #相当于 test = X(test)

8 def test():

9 print('OK')

10

11 test(1)

@在python中叫做语法糖，而@X 与test = X(test)等价，所以上面两种方法是等价的，只是第一种方法比较直观的展示了装饰器，而第二种写法比较美观简洁。

**081、高阶函数？**

什么是高阶函数？

（1）函数本身可以赋值给变量，赋值后变量为函数；  
（2）允许将函数本身作为参数传入另一个函数；  
（3）允许返回一个函数；  
以内置函数 abs 求绝对值为例：

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> abs(-10)  10  >>> a = abs  >>> a(-20)  20 |

内置函数 abs 赋值给变量 a ，在通过变量 a 执行，现在变量 a  和函数 abs 具有同样的功能；  
结论：函数本身也可以赋值给变量，即：变量可以指向函数。当变量指向函数时， 变量要可以想函数一样调用

传入函数  
变量即可以指向函数，函数的参数也能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称为 高阶函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def foo():          print('from foo')    def test(func):          return func    # foo 函数作为参数传入另一个函数  t = test(foo)  print(t)    # 运行结果：  # <function foo at 0x0000026A2A4E9048> |

在上面的例子中，test 就是一个高阶函数，接收一个函数作为参数，并返回一个函数，最后的打印是函数的内存地址。

**083、尝试通过高阶函数来实现装饰器的功能？**  
例1 尝试通过装饰器的方式计算 foo 函数的运行时间：

|  |  |
| --- | --- |
|  | import time    def timmer(func):          start\_time = time.time()          func()          print('foo函数运行时间：', time.time()-start\_time)      def foo():          time.sleep(2)          print('hello, foo.')    timmer(foo)    # 运行结果：  # hello, foo.  # foo函数运行时间： 2.0004444122314453 |

装饰器是为其他函数添加附加功能的函数。上面的例子中，timmer 函数确实为 foo 添加了附加功能，计算出了 foo 函数运行的时间。  
我们在通过装饰器的两大原则来比较：  
    （1）不修改被装饰函数的源代码，上面的例子中 foo 函数的源代码没有被修改 -- 满足  
    （2）不修改被装饰函数的调用方式； 上面 foo 函数调用方式应该是 foo()  而上面为了实现附加功能，调用方式修改为 timmer(foo)，调用方式发生了修改 -- 不满足  
所以说，高阶函数不能满足装饰器的两大原则。

**084、嵌套函数？**

 在一个函数中，定义另一个函数：  
例：这就是一个嵌套函数

|  |  |
| --- | --- |
|  | def foo():          def test():                  print('test.') |

 说到嵌套函数就一定会提到作用域；  
（1）作用域  
一个标识符的可见范围，这就是标识符的作用域。一般常说的是变量的作用域  
全局作用域（global）：在整个程序运行环境中都可见；  
局部作用域：在函数、类等内部可见；局部变量使用范围不能超过其所在的局部作用域

|  |  |
| --- | --- |
|  | def foo():          name = 'hkey'          def test():                  print(locals())                  print(name)          print(locals())          test()  foo()    # 执行结果：  # {'test': <function foo.<locals>.test at 0x000002494DA55950>, 'name': 'hkey'}  # {'name': 'hkey'}  # hkey |

上面的实例运行步骤如下：  
（1）首先打印 foo 函数中 print(locals()) --> {'name': 'hkey', 'test': <function foo.<locals>.test at 0x000001C6F4BF5950>}  
打印的是 foo 函数中的局部变量，有变量 name 以及 test 函数的内存地址  
（2）执行 foo 函数中的 test() 函数  
（3）执行嵌套函数 test 中的 print(locals()) --> {'name': 'hkey'} 发现在嵌套函数 test 中能够获取上一级函数定义的变量  
（4）最后在嵌套函数 test 中，打印 name 变量 --> 'hkey'

结论：在嵌套函数中，如果没有定义需要调用局部变量的值，则会去上一层中的局部变量找是否存在，如果上一层不存在，则会去全局变量中找。  
就像找一种微量元素，首先在地球上找，如果地球没有则会去太阳系找，如果太阳系没有则会去银河系找。这里面 银河系包括太阳系，太阳系又包括地球

**085、闭包的作用？**

|  |  |
| --- | --- |
|  | >>> def foo(n):  ...     def test():  ...         return n + 1  ...     return test  ...  >>> f = foo(10)  >>> f  <function foo.<locals>.test at 0x000001D5958EF0D0>  >>> f()  11  >>> f = foo(20)  >>> f()  21 |

 在这段程序中，函数 test 是函数 foo 的内嵌函数，并且 test 是 foo 函数的返回值。  
foo 函数只是返回了内嵌函数 test 的地址，在单独执行 test 函数时将会由于在其作用域中找不到 n 变量而出错。而在函数式语言中，当内嵌函数体内引用到体外的变量时，将会把定义时涉及的引用环境和函数体打包成一个整体返回， 这个实例就是闭包的作用。

**086、编写一个装饰器？**

 通过上面 高阶函数、嵌套函数、闭包的简介及举例，尝试写一个装饰器  
例：计算 foo 函数运行的时间

|  |  |
| --- | --- |
|  | import time    def timmer(func):          def wrapper():                  start\_time = time.time()                  func()                  print('程序运行时间：', time.time()-start\_time)          return wrapper    def foo():          time.sleep(2)          print('foo run finish.')    foo = timmer(foo)  foo()    # 运行结果：  # foo run finish.  # 程序运行时间： 2.0003890991210938 |

（1）首先，定义 timmer函数，定义func为参数，返回值是内嵌函数 wrapper， 被装饰函数为 foo；  
（2）在定义 foo = timmer(foo) 时，实际上是 foo = wrapper 函数内存地址，这里使用高阶函数和闭包的概念；  
（3）foo()  等于在执行 wrapper() 在 wrapper 函数中执行了timmer函数中的参数 func ， 而这里参数func 就是 foo；

foo = timmer() 使用语法糖的方式：  
无参数的装饰器：

|  |  |
| --- | --- |
|  | import time  def timmer(func):          def wrapper():                  start\_time = time.time()                  func()                  print('程序运行时间：', time.time()-start\_time)          return wrapper    @timmer  def foo():          time.sleep(2)          print('foo run finish.')    foo()    # 运行结果：  # foo run finish.  # 程序运行时间： 2.0003890991210938 |

 这样一个无参数的装饰器就完成了， 因为被装饰的函数无需传入任何参数；

 有参数的装饰器：

|  |  |
| --- | --- |
|  | import time  def timmer(func):          def wrapper(\*args, \*\*kwargs):                  start\_time = time.time()                  func(\*args, \*\*kwargs)                  print('app runtime:', time.time()-start\_time)          return wrapper    @timmer  def foo(name):          time.sleep(2)          print('hello,', name)    foo('hkey')    # 运行结果：  # hello, hkey  # app runtime: 2.0005948543548584 |

在有参数的装饰器中 \*args, \*\*kwargs 表示接收了任意类型的参数，关于 \*args, \*\*kwargs 含义请参考：[函数的参数](https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/001431752945034eb82ac80a3e64b9bb4929b16eeed1eb9000)

 有返回值的装饰器：

|  |  |
| --- | --- |
|  | import time  def timmer(func):          def wrapper(\*args, \*\*kwargs):                  start\_time = time.time()                  res = func(\*args, \*\*kwargs)                  print('app runtime:', time.time()-start\_time)                  return res          return wrapper    @timmer  def foo(name):          time.sleep(2)          print('hello,', name)          return '返回foo'    f = foo('hkey')  print(f)    # 运行结果：  # hello, hkey  # app runtime: 2.0003833770751953  # 返回foo |

**087、几种不同的装饰器及执行流程？**

被装饰函数的视角：

　　通过被装饰函数 的特征分为以下几种类型：

　　　　（1）被装饰的函数不带参数和返回值；

　　　　（2）被装饰的函数带参数但没有返回值；

　　　　（3）被装饰的函数带参数和返回值；

|  |  |  |
| --- | --- | --- |
|  |  | def log(f):      def wrapper(\*args, \*\*kwargs):          print('func 函数运行前')          ret = f(\*args, \*\*kwargs)          print('func 函数运行后')          return ret      return wrapper      @log  def func(x, y):      print('hello', x, y)      return 'func: 返回值'      print(func('xiaofei', 'hkey'))      # 执行结果：  # func 函数运行前  # hello xiaofei hkey  # func 函数运行后  # func: 返回值 |

以上 3 种情况被装饰函数的类型，都可以通过上面的装饰器 log 来装饰。具体执行流程如下图：

[](https://img2018.cnblogs.com/blog/828019/201809/828019-20180922144407705-344839000.png)

装饰器视角：  
　　根据装饰器的使用分为以下；

　　　　（1）装饰器带有参数；

　　　　（2）多个装饰器装饰一个函数；

（1）装饰器带有参数

|  |  |
| --- | --- |
|  | def log(text):      def decorator(func):          def wrapper(\*args, \*\*kwargs):              print('函数之前执行')              print('装饰器说明：', text)              ret = func(\*args, \*\*kwargs)              print('函数之后执行')              return ret          return wrapper      return decorator      @log('我是带参数的装饰器')  def func(x, y):      print('hello', x, y)      return 'func: 返回值'      print(func('xiaofei', 'hkey'))      # 执行结果：  # 函数之前执行  # 装饰器说明： 我是带参数的装饰器  # hello xiaofei hkey  # 函数之后执行  # func: 返回值 |

带参数的装饰器执行流程如下：

[](https://img2018.cnblogs.com/blog/828019/201809/828019-20180922150012943-43418192.png)

 （2）多个装饰器同时装饰同一个函数

|  |  |
| --- | --- |
|  | def log1(func): # log(wrapper2)      def wrapper1(\*args, \*\*kwargs):          print('before: log1')          ret = func(\*args, \*\*kwargs)          print('after: log1')          return ret      return wrapper1    def log2(func): # log2(func)      def wrapper2(\*args, \*\*kwargs):          print('before: log2')          ret = func(\*args, \*\*kwargs)          print('after: log2')          return ret      return wrapper2    @log1   # --> func = log1(func) --> log1(wrapper2) 首先执行  @log2   # --> func = log2(func) --> wrapper2  def func():      print('hello')    func()      # 执行结果：  # before: log1  # before: log2  # hello  # after: log2  # after: log1 |

 多个装饰器同时装饰同一个函数流程如下图：

[](https://img2018.cnblogs.com/blog/828019/201809/828019-20180922151153113-1390839370.png)

**088、 装饰器的几个实例？**

 （1）使用装饰器实现登录验证的功能。当用户调用 home 函数的时候必须经过登录验证：

|  |  |
| --- | --- |
|  | def auth(func):          def wrapper(\*args, \*\*kwargs):                  username = input('用户名:').strip()                  passwd = input('密码:').strip()                  if username == 'admin' and passwd == '123':                          res = func(\*args, \*\*kwargs)                          return res                  else:                          print('用户名密码错误！')            return wrapper    @auth  def home():          print('welcome home.')    home()    # 执行结果：  # （1）用户名密码正确  # 用户名:admin  # 密码:123  # welcome home.  #  # （2）用户名密码错误  # 用户名:admin  # 密码:111 |

（2）使用装饰器实现登录功能，当调用 home 函数时，验证方式为 filedb ，当登录购物车时，验证方式为 ldap：

|  |  |
| --- | --- |
|  | def auth(auth\_type):          def decorator(func):                  def wrapper(\*args, \*\*kwargs):                          username = input('用户名：').strip()                          passwd = input('密码：').strip()                          if auth\_type == 'filedb':                                  print('filedb 验证中...')                                  if username == 'filedb' and passwd == '123':                                          res = func()                                          return res                          elif auth\_type == 'ldap':                                  print('ldap 验证中...')                                  if username == 'ldap' and passwd == '123':                                          res = func()                                          return res                    return wrapper            return decorator    @auth('filedb')  def home():          print('welcome home.')  @auth('ldap')  def shopping\_cars():          print('个人购物车.')  home()  shopping\_cars()    # 运行结果：  # 用户名：filedb  # 密码：123  # filedb 验证中...  # welcome home.  # 用户名：ldap  # 密码：123  # ldap 验证中... |

（3）写了一个装饰器，在很多函数中都使用了， 如何简单的全部关闭，如何设计装饰器：

|  |  |
| --- | --- |
|  | Flag = False  # Flag = True的时候使用装饰器中附加功能，否则不使用装饰器中的附加功能      def log(flag):      def decorator(func):          def wrapper(\*args, \*\*kwargs):              if flag:  # 通过 flag 参数来控制是否要使用装饰器中的内容。                  print('----')                  ret = func(\*args, \*\*kwargs)                  print('#####')                  return ret              else:                  ret = func(\*args, \*\*kwargs)                  return ret            return wrapper        return decorator      @log(Flag)  def func1():      print('func: func1.')      @log(Flag)  def func2():      print('func: func2.')      @log(Flag)  def func3():      print('func: func3.')      func1()  func2()  func3() |

**081、简述生成器，迭代器，装饰器以及应用场景？函数装饰器有什么作用？**

**生成器和迭代器的区别？**这个是stackoverflow里python排名第一的问题,值得一看:

<http://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do-in-python>

这是中文版: <http://taizilongxu.gitbooks.io/stackoverflow-about-python/content/1/README.html>

这里有个关于**生成器的创建问题**面试官有考：  
问： 将列表生成式中[]改成() 之后数据结构是否改变？  
答案：是，从列表变为生成器

>>> L = [x\*x for x in range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x\*x for x in range(10))

>>> g

<generator object <genexpr> at 0x0000028F8B774200>

通过列表生成式，可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含百万元素的列表，不仅是占用很大的内存空间，如：我们只需要访问前面的几个元素，后面大部分元素所占的空间都是浪费的。因此，没有必要创建完整的列表（节省大量内存空间）。在Python中，我们可以采用生成器：边循环，边计算的机制—>generator。

**迭代器是一个更抽象的概念**，任何对象，如果它的类有next方法和iter方法返回自己本身，对于string,list,dict,tuple等这类容器对象，使用for循环遍历是很方便的，在后台for语句对容器对象调用iter()函数，iter()是python的内置函数，iter()会返回一个定义了next()方法的迭代器对象，它在容器中逐个访问容器内元素，next()也是python的内置函数，在没有后续元素时，next()会抛出一个StopIteration异常。

**生成器（Generator）是创建迭代器的简单而强大的工具**。它们写起来就像是正规的函数，只是在需要返回数据的时候使用yield语句。每次next()被调用时，生成器会返回它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）。

**区别：** 生成器能做到迭代器能做的所有事，而且因为自动创建iter()和next()方法，生成器显得特别简洁，而且生成器也是高效的，使用生成器表达式取代列表解析可以同时节省内存。除了创建和保存程序状态的自动方法，当发生器终结时，还会自动抛出StopIteration异常。

1、迭代器对象实现了iter()方法；迭代器是指遵循迭代器协议的对象，这类对象在被for循环时，每次迭代生成下一个项，不用一开始就生成整个；

2、迭代器实现了iter()和next()方法，迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束；

3、生成器是迭代器的一种，一个函数调用时返回一个迭代器，这个函数就叫生成器。通常带有yield；

4、装饰器是一个以函数作为参数，并返回一个替换函数的可执行函数，是闭包的一种应用。通常用来给一个函数添加功能；

装饰器:装饰器本质上还是一个函数,比如你想要执行某个函数前记录日志，又不想改动这个函数，就可以通过装饰器来实现,装饰器也是可以带参数的。

例：请实现一个装饰器,限制该函数被调用的频率,如10秒一次

# 答案  
import time  
def time\_pay(func):  
  def inner(\*args, \*\*kwargs):  
      for line in range(10):  
          print(line + 1)  
          time.sleep(1)  
      res = func(\*args, \*\*kwargs)  
      return res  
  return inner  
@time\_pay  
def func1():  
  print('from func1...')  
func1()

迭代器:可迭代对象，简单的来理解就是可以使用 for 来循环遍历的对象。比如常见的 list、set和dict。

生成器:多用于列表生成式， yield 函数记录此时的函数调用位置，下一次从这个位置开始。

**函数装饰器有什么作用？**

装饰器本质上是一个python函数，它可以在让其他函数在不需要做任何代码的变动的前提下增加额外的功能；对函数执行过程，做一些扩展，甚至可以更改本身函数的执行。

装饰器的返回值也是一个函数的对象，它经常用于有切面需求的场景。比如：插入日志，性能测试，事务处理，缓存。权限的校验等场景，有了装饰器就可以抽离出大量的与函数功能本身无关的雷同代码并发并继续使用。

Python装饰器（decorator）在实现的时候，有一些细节需要被注意。例如，被装饰后的函数其实已经是另外一个函数了（函数名等函数属性会发生改变）。这样有时候会对程序造成一些不便，例如笔者想对flask框架中的一些函数添加自定义的decorator，添加后由于函数名和函数的doc发生了改变，对测试结果有一些影响。所以，Python的functools包中提供了一个叫wraps的decorator来消除这样的副作用。写一个decorator的时候，最好在实现之前加上functools的wrap，它能保留原有函数的名称和docstring。

**085、下列数据结构中,哪一种是不可迭代的？**

A. dict

B. object

C. set

D. str

# 答案：B

**082、使用生成器编写一个函数实现生成指定个数的斐波那契数列？**

def fib2(imax):

t,a,b=0,0,1

while t<imax:

yield b

a,b=b,a+b

t+=1

for i in fib2(10):

print(i)

**083、一行代码通过filter和lambda函数输出alist=[1,22,2,33,23,32]中索引为奇数的值？**

alist=[1,22,2,33,23,32]

ss=[x[1] for x in filter(lambda x:x[0]%2==1,enumerate(alist))]

print(ss)

**084、编写一个函数实现十进制转62进制，分别用0-9A-Za-z,表示62位字母？**

import string

print(string.ascii\_lowercase) # 小写字母

print(string.ascii\_uppercase) # 大写字母

print(string.digits) # 0-9

s=string.digits+string.ascii\_uppercase+string.ascii\_lowercase

def \_10\_to\_62(num):

ss=''

while True:

ss=s[num%62]+ss

if num//62==0:

break

num=num//62

return ss

print(\_10\_to\_62(65))

**085、实现一个装饰器，限制该函数被调用的频率，如10秒一次？**

简写版：

import time

def time\_pay(func):

def inner(\*args,\*\*kwargs):

for line in range(10)：

print(line + 1)

time.sleep(1)

res = func(\*args,\*\*kwargs)

return res

return inner

详细版：

import time

from functools import wraps

def dec(func):

key=func.\_\_name\_\_

cache={key:None}

@wraps(func)

def inner(\*args,\*\*kwargs):

result=None

if cache.get(key) is None:

cache[key]=time.time()

result=func(\*args,\*\*kwargs)

print('执行函数中')

else:

now=time.time()

if now-cache[key]>10:

cache[key]=now

result=func(\*args,\*\*kwargs)

print('执行函数中')

else:

print('函数执行受限')

return result

return inner

@dec

def add(x,y):

print(x+y)

add(1,2)

add(1,3)

time.sleep(10)

add(3,4)

**086、实现一个装饰器，通过一次调用，使函数重复执行5次？**

from functools import wraps

def dec(func):

@wraps(func)

def inner(\*args,\*\*kwargs):

result=[func(\*args,\*\*kwargs) for i in range(5)]

return result

return inner

@dec

def add(x,y):

return x+y

print(add(1,2))

**087、生成器与函数的区别，并实现和简单使用生成器(generator)？**

* 生成器和函数的主要区别在于函数 return a value，生成器 yield a value同时标记或记忆point of the yield 以便于在下次调用时从标记点恢复执行。
* yield 使函数转换成生成器，而生成器反过来又返回迭代器。

# 简单实现生成器

def dec():

n=0

for i in range(10):

yield n

n+=i

for i in dec():

print(i)

**088、列表推导式[i for i in range(10)]和生成式表达式(i for i in range(10))的区别？**

1、列表推导式的结果是一个列表。

2、生成器表达式的结果是一个生成器，它和列表推导式类似，它一次处理一个对象，而不是一口气处理和构造整个数据结构，可以节约内存。

**089、python如何定义函数时如何书写可变参数和关键字参数？**

def func(a,\*args,b=1,\*\*kwargs):

pass

**090、python中enumerate的意思是什么？**

枚举的意思，同时得到可迭代对象，如列表和元组的索引和值，以元组形式返回。

**092、是否使用过functools中的函数？他的作用是什么？**

1、functools.wraps()：在装饰器中用过，如果不使用wraps，则原始函数的name和doc的值就会丢失；

2、functools.reduce()：第一个参数是一个函数，第二个参数是一个可迭代对象，

代码如下：

# 下面代码相当于从1加到9  
from functools import reduce  
a=reduce(lambda x,y:x+y,range(10))  
print(a)

**093、如何判断一个值是方法还是函数？**

1、使用type()来判断，如果是method为方法，如果是function则是函数。

2、与类和实例无绑定关系的function都属于函数（function）；

3、与类和实例有绑定关系的function都属于方法；

**094、请编写一个函数将ip地址转换成一个整数。如10.3.9.12转换成00001010 00000011 00001001 00001100，然后转换成整数？**

def ip2int(ip):  
   nums=ip.split('.')  
   # zfill()函数是补0  
   to\_bin=[bin(int(i))[2:].zfill(8) for i in nums]  
   return int(''.join(to\_bin),2)  
i=ip2int('127.0.0.1')  
print(i)

**095、lambda函数表达式的格式，有什么好处以及应用场景？**

lambda 函数其实就是一个匿名函数,为什么叫lambda?因为和后面的函数式编程有关.推荐: [知乎](http://www.zhihu.com/question/20125256)

lambda 函数是一个可以接收任意多个参数(包括可选参数)并且返回单个表达式值的函数；  
1、lambda 函数比较轻便，即用即仍，很适合需要完成一项功能，但是此功能只在此一处使用，连名字都很随意的情况下；  
2、匿名函数，一般用来给 filter， map 这样的函数式编程服务;  
3、作为回调函数，传递给某些应用，比如消息处理

格式：lambda 参数列表 : 返回表达式；

应用场景：常见的在filter，reduce以及map中使用。

**096、pass的使用？**

通常用来标记一个还未写的代码的位置，pass不做任何事情，一般用来做占位语句，保持程序结构的完整性

**097、\*arg和\*\*kwargs的作用？**

用来接收不确定个数的参数，\*args通常用来接收不确定个数的非关键字参数，而\*\*kwargs通常用来接收不确定个数的关键字参数

**098、如何在函数中设置一个全局变量？**

在函数中使用global关键字定义变量

**099、求以下代码结果？**

def num():  
   return [lambda x:i\*x for i in range(4)]  
print([m(2) for m in num()])

* 答案：[6,6,6,6]

**100、yield from 和 yield 的区别？**

[简述yield和yield from](https://blog.csdn.net/lamusique/article/details/85845225)

# 下面a()和b()是等价的  
def a():  
   yield from [1,2,3,4,5]  
def b():  
   for i in [1,2,3,4,5]:  
       yield i  
for i in a():  
   print(i)  
for i in b():  
   print(i)

* yield将一个函数变成一个生成器；
* yield 返回一个值；
* yield from后面接可迭代对象，一个一个返回值。

**101、求以下代码的输出结果？**

collapse=True

processFunc=collapse and (lambda s:' '.join(s.split())) or (lambda s:s)

print(processFunc('i\tam\ntest\tproject!'))

collapse=False

processFunc=collapse and (lambda s:' '.join(s.split())) or (lambda s:s)

print(processFunc('i\tam\ntest\tproject!'))

* **答案：i am test project! i am test project!**

**102、编写一个函数，找出数组中没有重复的值的和？**

def func(lis):

lis1=[]

del\_lis=[]

for i in lis:

if i not in lis1:

if i not in del\_lis:

lis1.append(i)

else:

del\_lis.append(i)

lis1.remove(i)

return sum(lis1)

def func2(lis):

return sum([i for i in set(lis) if lis.count(i)==1])

print(func2([3,4,1,2,5,6,6,5,4,3,3]))

**103、下面代码的执行结果是？**

a=1

def bar():

a+=3

bar()

print(a)

* 答案：运行出错

**104、写一个函数，计算出以下字母所代表的数字，每个字母值不一样？**

for A in range(1,10):

for B in range(10):

if A==B:

continue

for C in range(1,10):

if C in [A,B]:

continue

for D in range(10):

if D in [A,B,C]:

continue

for E in range(1,10):

if E in [A,B,C,D]:

continue

for F in range(10):

if F in [A,B,C,D,E]:

continue

for G in range(1,10):

if G in [A,B,C,D,E,F]:

continue

for H in range(10):

if H in [A,B,C,D,E,F,G]:

continue

for P in range(1,10):

if P in [A,B,C,D,E,F,G,H]:

continue

if (A\*10+B)-(C\*10+D)==(E\*10+F) and (E\*10+F)+(G\*10+H)==(P\*100+P\*10+P):

print(A,B,C,D,E,F,G,H,P)

**105、写出如下代码的输出结果？**

def decorator\_a(func):

print('Get in decorator\_a')

def inner\_a(\*args, \*\*kwargs):

print('Get in inner\_a')

return func(\*args, \*\*kwargs)

return inner\_a

def decorator\_b(func):

print('Get in decorator\_b')

def inner\_b(\*args, \*\*kwargs):

print('Get in inner\_b')

return func(\*args, \*\*kwargs)

return inner\_b

@decorator\_b #f=decorator\_b(f)

@decorator\_a #f=decorator\_a(f)

def f(x):

print('Get in f')

return x \* 2

f(1)

* 答案

**Get in decorator\_a Get in decorator\_b Get in inner\_b Get in inner\_a Get in f**

* 解释

**当我们对f传入参数1进行调用时，inner\_b被调用了，他会先打印Get in inner\_b,然后在inner\_b内部调用了inner\_a,所以会再打印Get in inner\_a,然后再inner\_a内部调用原来的f,并且将结果作为最终的返回总结：装饰器函数在被装饰函数定义好后立即执行从下往上执行函数调用时从上到下执行；**

**106、写出以下代码的输出结果？**

def test():

try:

raise ValueError('something wrong')

except ValueError as e:

print('error occured')

return

finally:

print('ok')

test()

* 结果(finally无论怎样都会执行)；

error occured ok

**107、求出以下代码的输出结果？**

mydict={'a':1,'b':2}

def func(d):

d['a']=0

return d

func(mydict)

mydict['c']=2

print(mydict)

* 结果：{'a': 0, 'b': 2, 'c': 2}

**108、写个函数接收一个文件夹名称作为参数，显示文件夹中文件的路径，以及其中包含的文件夹中文件的如今？**

# 方法一  
import os  
def Test1(rootDir):  
   list\_dirs = os.walk(rootDir)  
   for root, dirs, files in list\_dirs:  
       for d in dirs:  
           print(os.path.join(root, d))  
       for f in files:  
           print(os.path.join(root, f))  
Test1(r'C:\Users\felix\Desktop\aaa')  
print('###################')  
# 方法二  
import os  
def Test2(rootDir):  
   paths=os.listdir(rootDir)  
   for lis in paths:  
       path=os.path.join(rootDir,lis)  
       print(path)  
       if os.path.isdir(path):  
            Test2(path)  
Test2(r'C:\Users\felix\Desktop\aaa')

**109、re的match和search的区别？**

match()函数是在string的开始位置匹配，如果不匹配，则返回None；

search()会扫描整个string查找匹配；也就是说match()只有在0位置匹配成功的话才有返回；

**110、什么是正则的贪婪匹配？贪婪模式和非贪婪模式的区别？**

贪婪匹配:正则表达式一般趋向于最大长度匹配，也就是所谓的贪婪匹配。非贪婪匹配：就是匹配到结果就好，就少的匹配字符。

区别：默认是贪婪模式；在量词后面直接加上一个问号？就是非贪婪模式。

**111、如何使用python删除一个文件或者文件夹？**

import os

import shutil

os.remove(path) # 删除文件

os.removedirs(path) # 删除空文件夹

shutil.rmtree(path) # 删除文件夹，可以为空也可以不为空

**112、logging模块的作用以及应用场景？**

logging模块定义的函数和类为应用程序和库的开发实现了一个灵活的事件日志系统。

记录日志；

**113、json序列化时可以处理的数据类型有哪些？如何定制支持datetime类型？序列化时，遇到中文转成unicode，如何保持中文形式？**

1. 可以处理的数据类型是 string、int、list、tuple、dict、bool、null
2. 通过自定义时间序列化转换器

import json

from json import JSONEncoder

from datetime import datetime

class ComplexEncoder(JSONEncoder):

def default(self, obj):

if isinstance(obj, datetime):

return obj.strftime(‘%Y-%m-%d %H:%M:%S‘)

else:

return super(ComplexEncoder,self).default(obj)

d = { ‘name‘:‘alex‘,‘data‘:datetime.now()}

print(json.dumps(d,cls=ComplexEncoder))

# {"name": "alex", "data": "2018-05-18 19:52:05"}

1. 使用ensure\_ascii=False参数

**114、写出邮箱的正则表达式？**

import re

pp=re.compile('[a-zA-Z0-9\_-]+@[0-9A-Za-z]+(\.[0-9a-zA-Z]+)+')

if pp.match('1403179190@qq.com'):

print('ok')

**115、写python爬虫分别用到了哪些模块？分别有什么用？**

模块：

1、request，发起请求；

2、pyquery，解析html数据；

3、beautifulsoup，解析html数据；

4、aiohttp，异步发送请求；

框架

1、pyspider，web界面的爬虫框架；

2、scrapy，爬虫框架；

3、selenium，模拟浏览器的爬虫框架；

sys.path.append('xxx')的作用：添加搜索路径；

**116、输入某年某月某日，判断这是这一年的第几天？**

date=input('请输入某年某月某日，格式：xxxx.xx.xx')

def get\_day(date):

days1=[31,28,31,30,31,30,31,31,30,31,30,31]

days2=[31,29,31,30,31,30,31,31,30,31,30,31]

year,month,day = [int(i) for i in date.split('.')]

if year % 400 ==0 or (year % 4==0 and year % 100!=0):

days=days2

else:

days=days1

return sum(days[:month-1])+day

print(get\_day(date))

**117、简述面向对象的三大特性？**

继承，封装和多态；

1、封装：封装:封装指的是把一堆数据属性与方法数据放在一个容器中，这个容器就是对象。让对象可以通过 "." 来调用对象中的数据属性与方法属性。 封装就是把类中的属性和方法定义为私有的，方法就是在属性名或方法名前加双下划线，而一旦这样定义了属性或方法名后，python会自动将其转换为类名\_\_属性名（方法名）的格式，在类的内部调用还是用双下划线加属性名或方法名，在类的外部调用就要用类名\_\_属性名（方法名）。父类的私有属性和方法，子类无法对其进行修改。

2、继承：继承就是继承的类直接拥有被继承类的属性而不需要在自己的类体中重新再写一遍，其中被继承的类叫做父类、基类，继承的类叫做派生类、子类。继承指的是子类可以继承父类的数据属性与方法属性，并可以对其进行修改或使用。

3、多态：多态就是不同的对象可以调用相同的方法然后得到不同的结果，有点类似接口类的感觉，在python中处处体现着多态，比如不管你是列表还是字符串还是数字都可以使用+和\*。在python中的多态指的是让多种类若具备类似的数据属性与方法属性，都统一好命名规范，这样可以提高开发者的代码统一性，使得调用者更方便去理解。

**118、什么是鸭子模型？**

鸭子类型（英语：duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。

在python中不会强制性要求所有人的代码都统一规范，不统一也不会报错，若使用抽象类就会使python代码强制统一规范，这样不符合python动态语言的特性。所以让大家都自觉统一好规范，若大家的对象方法都类似的话就一种规范，只要长得像鸭子，就称之为鸭子类型。

**119、super的作用？**

当子类中的方法与父类中的方法重名时，子类中的方法会覆盖父类中的方法，那么，如果我们想实现同时调用父类和子类中的同名方法，就需要使用到super()这个函数，用法为super().函数名().

使用super()可以在子类中调用父类的方法或属性, 可能你会说, 子类本来就可以调用父类中所有非私有的属性或方法,而我现在说的是, 当子类中实现了某个方法, 父类中也有这个方法, 当你调用这个方法时, 既想执行子类的又想执行父类的, 在这种情况下就可以使用super()；

**120、mro是什么？**

mro全称Method Resolution Order，指的是方法解析顺序。对于支持继承的编程语言来说，其方法（属性）可能定义在当前类，也可能来自于基类，所以在方法调用时就需要对当前类和基类进行搜索以确定方法所在的位置。而搜索的顺序就是所谓的「方法解析顺序」（Method Resolution Order，或MRO）。

**121、什么是c3算法？**

c3算法是python新式类中用来产生mro顺序的一套算法。即多继承的查找规则。

C3算法最早被提出是用于Lisp的，应用在Python中是为了解决原来基于深度优先搜索算法不满足本地优先级，和单调性的问题。  
本地优先级：指声明时父类的顺序，比如C(A,B)，如果访问C类对象属性时，应该根据声明顺序，优先查找A类，然后再查找B类。  
单调性：如果在C的解析顺序中，A排在B的前面，那么在C的所有子类里，也必须满足这个顺序。

**122、列举面向对象中带双下划线的特殊方法？**

\_\_new\_\_：可以调用其它类的构造方法或者直接返回别的对象来作为本类的实例。

\_\_init\_\_： 负责类的实例化 ；

\_\_call\_\_：对象后边加括号，触发执行 ；

\_\_str\_\_：print打印一个对象时。

\_\_doc\_\_：类的注释，该属性是无法继承的。

\_\_getattr\_\_：在使用点调用属性（方式、属性）且属性不存在的时候触发 ；

\_\_setattr\_\_：添加/修改属性会触发它的执行 ；

\_\_delattr\_\_：删除属性的时候会触发 ；

\_\_delete\_\_：采用del删除属性时，触发；

\_\_getattribute\_\_: 不管是否存在,我都会执行；

**123、Python中双下划线和单下划线的区别？**

"单下划线" 开始的成员变量叫做保护变量，意思是只有类对象和子类对象自己能访问到这些变量；

"双下划线" 开始的是私有成员，意思是只有类对象自己能访问，连子类对象也不能访问到这个数据。

>>> class MyClass():

... def \_\_init\_\_(self):

... self.\_\_superprivate = "Hello"

... self.\_semiprivate = ", world!"

...

>>> mc = MyClass()

>>> print mc.\_\_superprivate

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: myClass instance has no attribute '\_\_superprivate'

>>> print mc.\_semiprivate

, world!

>>> print mc.\_\_dict\_\_

{'\_MyClass\_\_superprivate': 'Hello', '\_semiprivate': ', world!'}

\_\_foo\_\_:一种约定,Python内部的名字,用来区别其他用户自定义的命名,以防冲突，就是例如\_\_init\_\_(),\_\_del\_\_(),\_\_call\_\_()这些特殊方法

\_foo:一种约定,用来指定变量私有.程序员用来指定私有变量的一种方式.不能用from module import \* 导入，其他方面和公有一样访问；

\_\_foo:这个有真正的意义:解析器用\_classname\_\_foo来代替这个名字,以区别和其他类相同的命名,它无法直接像公有成员一样随便访问,通过对象名.\_类名\_\_xxx这样的方式可以访问.

详情见：<http://stackoverflow.com/questions/1301346/the-meaning-of-a-single-and-a-double-underscore-before-an-object-name-in-python>

或者: <http://www.zhihu.com/question/19754941>

**124、实例变量和类变量的区别？**

类变量是所有对象共有，其中一个对象将它值改变，其他对象得到的就是改变后的结果；变量是该类所有实例共享的值（值包括属性和方法）（也就是说，它们不是单独分配给每个实例的）。而实例变量则属对象私有，某一个对象将其值改变，不影响其他对象；实例变量是例化之后，对于每个实例都独有的变量（数据）。

**125、实例方法、静态方法和类方法的区别？**

1、实例方法：第一个参数必须是实例对象，通常为self。实例方法只能由实例对象调用。

2、类方法：使用装饰器@classmethod装饰的方法为类方法。第一个参数为当前类的对象，通常为cls。实例对象和类对象都可以调用类方法。类方法只能访问类变量，不能访问实例变量。当我们直接访问初始化后的实例变量时，会提示没有这个属性(AttributeError: type object ‘ClassName‘ has no attribute ‘arg‘)。

3、静态方法：使用装饰器@staticmethod，将普通方法装饰为一个静态方法。没有self和cls参数。方法体中不能使用类或者实例的任何属性和方法，不能访问类变量和实例变量。实例对象和类对象都可以调用。

Python的类就是个语法糖。一个函数写在类里面和写在类外面没有区别，唯一的区别就是参数，所谓实例方法就是第一个参数是self，所谓类方法就是第一个参数是class，而静态方法不需要额外的参数，所以必须区分。

**126、isinstance和type的作用？**

* 两者都用来判断对象的类型；
* 对于一个类的之类对象的类型判断，type就不行了，而isinstance可以。

type和isinstance都可以判断变量是否属于某个内建类型。type只接收一个参数，不但可以判断变量是否属于某个类型，而且可以得到参数变量未知的所属的类型；而isinstance只能判断是否属于某个已知类型，不能直接得到变量未知的所属的类型；

class A(object):  
  pass  
class B(A):  
  pass  
     
ba=B()  
ab=A()  
print(type(ba)==A) # False  
print(type(ab)==A) # True  
print(isinstance(ab,A)) # True  
print(isinstance(ba,A)) # True

**127、有用过with statement(语句)吗?它的好处是什么?**

with语句会在嵌套的代码执行之后，自动关闭文件。这种做法的还有另一个优势就是，无论嵌套的代码是以何种方式结束的，它都关闭文件。如果在嵌套的代码中发生异常，它能够在外部exception handler catch异常前关闭文件。如果嵌套代码有return/continue/break语句，它同样能够关闭文件。

**127、简述with方法打开处理文件帮我我们做了什么？**

打开文件在进行读写的时候可能会出现一些异常状况，如果按照常规的f.open写法，我们需要try,except,finally，做异常判断，并且文件最终不管遇到什么情况，都要执行finally f.close()关闭文件，with方法帮我们实现了finally中f.close（当然还有其他自定义功能，有兴趣可以研究with方法源码）。

**128、with管理上下文的源码怎么写，类似的就行？**

Python with上下文管理及自定义上下文管理：

一、上下文管理器：

上下文管理器是对Context Manager的翻译 ,上下文是 context 直译的叫法,在程序中用来表示代码执行过程中所处的前后环境.

在文件操作时，需要打开，关闭文件，而在文件在进行读写操作时，就是处在文件操作的上下文中，也就是文件操作环境中.

语法: with

with 语句是 Pyhton 提供的一种简化语法,从Python 2.5 开始引入的一种与异常处理相关的功能,适用于对资源进行访问的场合,确保不管使用过程中是否发生异常都会执行必要的“扫尾”操作,释放资源.

比如文件使用后自动关闭、数据库连接的自动关闭等.

二、with上下文管理器示例：

with open('test.txt', 'a') as f:

f.write('上下文管理\n')

运行以上代码,会在代码同级目录下创建一个叫test.txt的文本文件,并在文件中追加内容"上下文管理",并换行.

在执行 with 语句时，首先执行 with 后面的 open 代码(如果已经有test.txt文件则打开,如果没有test.txt文件则创建),然后通过as将代码的结果保存到 f 中(相当于给test.txt起个别名f,f只是一个变量名,您可以自定义)

在with下面是对test.txt执行的操作,如示例中的操作是写入内容.

在操作后，不需要对文件test.txt进行关闭操作f.close()，with上下文管理器会在文件使用完后帮我们关闭test.txt文件.

这么做即可以简化代码,又可以避免因粗心忘记执行关闭操作而出现异常,因为在实际开发中,打开一个文件后,进行的操作可能非常复杂,这种情况是很可能忘记做"扫尾"操作的.

三、with上下文管理的原理：

在使用with上下文管理器时,并不是不需要关闭文件,而是文件的关闭操作在 with 的上下文管理器中已经实现了.当文件操作执行完成后,with语句会自动调用上下文管理器里的关闭语句来关闭文件资源.

我们的应用场景并不是一成必变,with是怎么实现帮我们做"扫尾"操作的呢?

with语句在执行时,调用上下文管理器中的 \_\_enter\_\_ 和 \_\_exit\_\_ 两个方法,这两个方法就是上下文管理器中实现的方法.

\_\_enter\_\_ 方法会在执行with后面的语句时执行,一般用来处理操作前的内容.比如打开文件,创建对象,初始化等.

\_\_exit\_\_ 方法会在with内的代码执行完毕后执行,一般用来处理一些善后收尾工作,比如文件的关闭,数据库的关闭等.

四、自定义上下文管理器

根据上下文管理的原理,上下文管理器的原理是实现了\_\_enter\_\_和\_\_exit\_\_这两个方法,所以我们可以根据此原理来自定义自己的上下文管理器.

在自定义上下文管理器时，只要在类中实现 \_\_enter\_\_ 和 \_\_exit\_\_ 两个方法即可.

class OpenFile(object):

"""自定义上下文管理类"""

def \_\_init\_\_(self, file, mode):

self.\_file = file

self.\_mode = mode

def \_\_enter\_\_(self):

print('\_\_enter\_\_ 打开文件')

self.\_handle = open(self.\_file, self.\_mode)

return self.\_handle

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

print('\_\_exit\_\_ 关闭文件')

self.\_handle.close()

with OpenFile('test01.txt', 'w') as f:

f.write('自定义上下文管理\n')

上面的代码中,我们不是使用Python实现好的open来打开文件,而是通过自定义的OpenFile类来完成文件的操作.

代码运行过程和结果与用open类似,会先执行OpenFile后的代码(自定义类OpenFile中的代码),如果存在文件test01.txt,打开文件并在其中写入"自定义上下文管理",如果不存在文件test01.txt,则创建文件并写入"自定义上下文管理".

说明我们成功实现了自定义上下文管理.

五、上下文管理中的自定义异常处理：

我们看到,\_\_exit\_\_ 方法中有三个参数exc\_type ,exc\_val 和exc\_tb ,这三个参数是用来接收异常信息的，如果代码在运行时发生异常，异常信息会被保存到这三个参数中.

class OpenFile(object):

"""自定义上下文管理类"""

def \_\_init\_\_(self, file, mode):

self.\_file = file

self.\_mode = mode

def \_\_enter\_\_(self):

self.\_handle = open(self.\_file, self.\_mode)

return self.\_handle

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

print('Type: ', exc\_type)

print('Value:', exc\_val)

print('TreacBack:', exc\_tb)

self.\_handle.close()

with OpenFile('test01.txt', 'r') as f:

f.write('自定义上下文管理\n')

上面的代码中,传入的'w'换成了'r',会出现异常,因为以读的模式打开文件,不能进行写操作.运行结果为:

Traceback (most recent call last):

Type: <class 'io.UnsupportedOperation'>

File "python\_demo/with\_demo.py", line 45, in <module>

Value: not writable

f.write('自定义上下文管理\n')

TreacBack: <traceback object at 0x0000024EA5E11508>

io.UnsupportedOperation: not writable

可以看到我们打印的三个参数exc\_type ,exc\_val 和exc\_tb的值.分别存储了异常的以下信息:

exc\_type : 异常类型

exc\_val : 异常值

exc\_tb : 异常回溯追踪

当with中执行的语句发生异常时,异常信息会被发送到 \_\_exit\_\_ 方法的参数中,这时可以根据情况选择如何处理异常.

因为在 \_\_exit\_\_函数执行异常处理时,会根据函数的返回值决定是否将系统抛出的异常继续向外抛出.如果返回值为 False 就会向外抛出,用户就会看到.如果返回值为 True 不会向外抛出,而是显示我们自定义的信息.

我们可以根据这个原理自定义如何处理异常,只要将返回值设置为True就行了.

class OpenFile(object):

"""自定义上下文管理类"""

def \_\_init\_\_(self, file, mode):

self.\_file = file

self.\_mode = mode

def \_\_enter\_\_(self):

self.\_handle = open(self.\_file, self.\_mode)

return self.\_handle

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

self.\_handle.close()

# 通过exc\_type参数接收到的值，来判断程序执行是否出现异常

# 如果是None,说明没有异常

if exc\_type == "None":

print('正常执行')

else:

# 否则出现异常，可以选择怎么处理异常

print(exc\_type, exc\_val)

# 返回值决定了捕获的异常是否继续向外抛出

# 如果是False那么就会继续向外抛出，程序会看到系统提示的异常信息

# 如果是True不会向外抛出,程序看不到系统提示信息，只能看到else中的输出

return True

with OpenFile('test01.txt', 'r') as f:

f.write('自定义上下文管理\n')

通过设置返回值为True,上下文管理器不会向外抛出异常,此时我们在根据exc\_type(异常类型)的值是否为"None"来自定义输出的异常信息,就可以实现上下文管理中的自定义异常了.

**127、使用with语句的好处是什么？**

1、使用with后不管with中的代码出现什么错误，都会进行对当前对象进行清理工作。例如file的file.close()方法，无论with中出现任何错误，都会执行file.close（）方法；

2、只有支持上下文管理器的对象才能使用with，即在对象内实现了两个方法：enter()和exit()；

**128、with 类方法 参考下面代码片段？**

class Context:

pass

with Context as ctx:

ctx.do\_something()

# 请在 Context 类下添加代码完成该类的实现

# 答案:

class Context(object):

def \_\_enter\_\_(self):

pass

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

if all([exc\_type, exc\_val, exc\_tb]):

print 'handler except'

print 'exception {}'.format(exc\_val)

return True

def main():

with tornado.stack\_context.StackContext(Contextor):

async\_task()

**128、写一个的支持with语句的类？**

class W(object):

def \_\_init\_\_(self):

pass

def \_\_enter\_\_(self):

print('进入with语句')

return self

def \_\_exit\_\_(self,\*args,\*\*kwargs):

print('退出with语句')

with W() as w:

print('之前')

print(w)

print('之后')

**129、实现一个单例模式。(尽可能多的方法)？**

**这个绝对常考啊.绝对要记住1~2个方法,当时面试官是让手写的.**

单例模式是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例类的特殊类。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数的控制并节约系统资源。如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。

\_\_new\_\_()在\_\_init\_\_()之前被调用，用于生成实例对象。利用这个方法和类的属性的特点可以实现设计模式的单例模式。

单例模式是指创建唯一对象，单例模式设计的类只能实例；

**1、使用\_\_new\_\_方法：使用基类\_\_new\_\_是真正创建实例对象的方法，所以重写基类的\_\_new\_\_方法，以此保证创建对象的时候只生成一个实例：**

例01：

class Singleton(object):

def \_\_new\_\_(cls, \*args, \*\*kw):

if not hasattr(cls, '\_instance'):

orig = super(Singleton, cls)

cls.\_instance = orig.\_\_new\_\_(cls, \*args, \*\*kw)

return cls.\_instance

class **Foo**(Singleton):

pass

foo1 = Foo()

foo2 = Foo()

print foo1 is foo2 #True

**2、元类，元类是用于创建类对象的类，类对象创建实例对象时一定要调用\_\_call\_\_方法，因此在调用\_\_call\_\_时候保证始终只创建一个实例即可，type是python的元类：**

**class** **Singleton**(type):

**def** **\_\_call\_\_**(cls,\*args,\*\*kwargs):

**if** **not** hasattr(cls,'\_instance'):

cls.\_instance = super(Singleton,cls).\_\_call\_\_(\*args,\*\*kwargs)

**return** cls.\_instance

**class** **Foo**(object):

\_\_metaclass\_\_ = Singleton

foo1 = Foo()

foo2 = Foo()

**print** foo1 **is** foo2 *#True*

**3、共享属性：**

创建实例时把所有实例的\_\_dict\_\_指向同一个字典,这样它们具有相同的属性和方法.

class Borg(object):

\_state = {}

def \_\_new\_\_(cls, \*args, \*\*kw):

ob = super(Borg, cls).\_\_new\_\_(cls, \*args, \*\*kw)

ob.\_\_dict\_\_ = cls.\_state

return ob

class MyClass2(Borg):

a = 1

**4、装饰器版本：**

def singleton(cls):

instances = {}

def getinstance(\*args, \*\*kw):

if cls not in instances:

instances[cls] = cls(\*args, \*\*kw)

return instances[cls]

return getinstance

@singleton

class MyClass(object):

pass

foo1 = Foo()

foo2 = Foo()

print foo1 is foo2 ##True

**4 import方法**

作为python的模块是天然的单例模式

# mysingleton.py

class My\_Singleton(object):

def foo(self):

pass

my\_singleton = My\_Singleton()

# to use

from mysingleton import my\_singleton

my\_singleton.foo()

[**单例模式伯乐在线详细解释**](http://python.jobbole.com/87294/)

# 例01：使用\_\_new\_\_()

import threading

class Singleton(object):

\_instance\_lock = threading.Lock()

def \_\_init\_\_(self):

pass

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if not hasattr(Singleton, "\_instance"):

with Singleton.\_instance\_lock:

if not hasattr(Singleton, "\_instance"):

Singleton.\_instance = object.\_\_new\_\_(cls)

return Singleton.\_instance

obj1 = Singleton()

obj2 = Singleton()

print(obj1 is obj2)

# 例02：使用元类来创建

import threading

class SingletonType(type):

\_instance\_lock = threading.Lock()

def \_\_call\_\_(cls, \*args, \*\*kwargs):

if not hasattr(cls, "\_instance"):

with SingletonType.\_instance\_lock:

if not hasattr(cls, "\_instance"):

cls.\_instance = super().\_\_call\_\_(\*args, \*\*kwargs)

return cls.\_instance

class Singleton(metaclass=SingletonType):

def \_\_init\_\_(self):

pass

obj1 = Singleton()

obj2 = Singleton()

print(obj1 is obj2)

**129、单例模式的优点和应用？**

单例模式的优点：

1、由于单例模式要求在全局内只有一个实例，因而可以节省比较多的内存空间；

2、全局只有一个接入点，可以更好地进行数据同步控制，避免多重占用；

3、单例可长驻内存，减少系统开销。

单例模式的应用举例：

1、生成全局惟一的序列号；

2、访问全局复用的惟一资源，如磁盘、总线等；

3、单个对象占用的资源过多，如数据库等；

4、系统全局统一管理，如Windows下的Task Manager；

5、网站计数器。

6、数据库配置，数据库连接池

7、应用程序的日志应用

**129、单例模式的缺点？**

1、单例模式的扩展是比较困难的；

2、赋于了单例以太多的职责，某种程度上违反单一职责原则（六大原则后面会讲到）;

3、单例模式是并发协作软件模块中需要最先完成的，因而其不利于测试；

4、单例模式在某种情况下会导致“资源瓶颈”。

**129、单例模式的应用场景有那些？**

单例模式应用的场景一般发现在以下条件下：

资源共享的情况下，避免由于资源操作时导致的性能或损耗等，如日志文件，应用配置。

控制资源的情况下，方便资源之间的互相通信。如线程池等，1,网站的计数器 2,应用配置 3.多线程池 4数据库配置 数据库连接池 5.应用程序的日志应用...

**130、如何判断一个对象是否可调用？哪些对象是可调用对象？如何定义一个类，使其对象本身就是可调用对象？**

使用callable函数判断。

可调用对象有7类：

1、用户自定义函数；

2、内置函数；

3、内置方法；

4、方法(定义在类中的函数)；

5、类；

6、类实例(如果类中定义了call方法，那么这个类的实例就是可调用对象)；

7、生成器函数 ；

在类中定义call方法，实例对象加()是即调用call的方法；

**134、类的加载和实例化过程？**

1、在堆内存中生成class对象, 把静态变量和静态方法加载到方法区, 这个堆内存中的class对象是方法区数据的入口；

2、静态变量默认初始化；

3、静态变量显式初始化；

4、执行静态代码块；

5、成员变量默认初始化, 显示初始化；

6、执行构造函数；

**135、类的加载顺序(类中有继承有构造有静态)?**

1、 类对象；

2、 实例对象；

3、 self变量名称问题；

4、 类属性、实例变量、局部变量；

5、 类方法；

6、 实例方法；

7、 类方法与实例方法相互调用；

8、 静态方法；

9、 继承时三类方法的影响；

**139、\*args and \*\*kwargs？**

用\*args和\*\*kwargs只是为了方便并没有强制使用它们.

当你不确定你的函数里将要传递多少参数时你可以用\*args.例如,它可以传递任意数量的参数:

>>> def print\_everything(\*args):

for count, thing in enumerate(args):

... print '{0}. {1}'.format(count, thing)

...

>>> print\_everything('apple', 'banana', 'cabbage')

0. apple

1. banana

2. cabbage

相似的,\*\*kwargs允许你使用没有事先定义的参数名:

>>> def table\_things(\*\*kwargs):

... for name, value in kwargs.items():

... print '{0} = {1}'.format(name, value)

...

>>> table\_things(apple = 'fruit', cabbage = 'vegetable')

cabbage = vegetable

apple = fruit

你也可以混着用.命名参数首先获得参数值然后所有的其他参数都传递给\*args和\*\*kwargs.命名参数在列表的最前端.例如:

def table\_things(titlestring, \*\*kwargs)

\*args和\*\*kwargs可以同时在函数的定义中,但是\*args必须在\*\*kwargs前面.

当调用函数时你也可以用\*和\*\*语法.例如:

>>> def print\_three\_things(a, b, c):

... print 'a = {0}, b = {1}, c = {2}'.format(a,b,c)

...

>>> mylist = ['aardvark', 'baboon', 'cat']

>>> print\_three\_things(\*mylist)

a = aardvark, b = baboon, c = cat

就像你看到的一样,它可以传递列表(或者元组)的每一项并把它们解包.注意必须与它们在函数里的参数相吻合.当然,你也可以在函数定义或者函数调用时用\*.

<http://stackoverflow.com/questions/3394835/args-and-kwargs>

**140、面向切面编程AOP和装饰器（装饰器问题前面已写，这里简写）？**

这个AOP一听起来有点懵,同学面阿里的时候就被问懵了...；

装饰器是一个很著名的设计模式，经常被用于有切面需求的场景，较为经典的有插入日志、性能测试、事务处理等。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量函数中与函数功能本身无关的雷同代码并继续重用。概括的讲，**装饰器的作用就是为已经存在的对象添加额外的功能。**

这个问题比较大,推荐: <http://stackoverflow.com/questions/739654/how-can-i-make-a-chain-of-function-decorators-in-python>

中文: <http://taizilongxu.gitbooks.io/stackoverflow-about-python/content/3/README.html>

**140、最近写bug时遇到一个关于函数执行超时的问题，就是某一步执行时间过长，或者程序直接卡死了，找到了三种解决方法，现在叙述下，帮助需要的人？**

（1）使用eventlet。（2）使用signal设置装饰器。（3）使用threading（使用threading的timer定时器）。

**141、鸭子类型？**

“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

我们并不关心对象是什么类型，到底是不是鸭子，只关心行为。

比如在python中，有很多file-like的东西，比如StringIO,GzipFile,socket。它们有很多相同的方法，我们把它们当作文件使用。

又比如list.extend()方法中,我们并不关心它的参数是不是list,只要它是可迭代的,所以它的参数可以是list/tuple/dict/字符串/生成器等.

鸭子类型在动态语言中经常使用，非常灵活，使得python不想java那样专门去弄一大堆的设计模式。

**142、Python中重载？**

引自知乎:<http://www.zhihu.com/question/20053359>

函数重载主要是为了解决两个问题。

1、可变参数类型。2、可变参数个数。

另外，一个基本的设计原则是，仅仅当两个函数除了参数类型和参数个数不同以外，其功能是完全相同的，此时才使用函数重载，如果两个函数的功能其实不同，那么不应当使用重载，而应当使用一个名字不同的函数。

**好吧，那么对于情况1 ，函数功能相同，但是参数类型不同，python 如何处理？**答案是根本不需要处理，因为 python 可以接受任何类型的参数，如果函数的功能相同，那么不同的参数类型在 python 中很可能是相同的代码，没有必要做成两个不同函数。

**那么对于情况2 ，函数功能相同，但参数个数不同，python 如何处理？**大家知道，答案就是缺省参数。对那些缺少的参数设定为缺省参数即可解决问题。因为你假设函数功能相同，那么那些缺少的参数终归是需要用的。

好了，鉴于情况1跟情况2都有了解决方案，python 自然就不需要函数重载了。

**143、新式类和旧式类？**

这个面试官问了,我说了老半天,不知道他问的真正意图是什么.

[stackoverflow](http://stackoverflow.com/questions/54867/what-is-the-difference-between-old-style-and-new-style-classes-in-python)

这篇文章很好的介绍了新式类的特性: <http://www.cnblogs.com/btchenguang/archive/2012/09/17/2689146.html>

Python中类分两种：旧式类和新式类：

➤新式类都从object继承，经典类不需要。

➤新式类的MRO(method resolution order 基类搜索顺序)算法采用C3算法广度优先搜索，而旧式类的MRO算法是采用深度优先搜索；

➤新式类相同父类只执行一次构造函数，经典类重复执行多次。

新式类很早在2.2就出现了,所以旧式类完全是兼容的问题,Python3里的类全部都是新式类.这里有一个MRO问题可以了解下(新式类是广度优先,旧式类是深度优先),<Python核心编程>里讲的也很多.

一个旧式类的深度优先的例子

class A():

def foo1(self):

print "A"

class B(A):

def foo2(self):

pass

class C(A):

def foo1(self):

print "C"

class D(B, C):

pass

d = D()

d.foo1()

# A

**按照经典类的查找顺序从左到右深度优先的规则，在访问d.foo1()的时候,D这个类是没有的..那么往上查找,先找到B,里面没有,深度优先,访问A,找到了foo1(),所以这时候调用的是A的foo1()，从而导致C重写的foo1()被绕过**

**144、\_\_new\_\_和\_\_init\_\_的区别？**

这个\_\_new\_\_确实很少见到,先做了解吧.

\_\_new\_\_是一个静态方法,而\_\_init\_\_是一个实例方法.

\_\_new\_\_方法会返回一个创建的实例,而\_\_init\_\_什么都不返回.

只有在\_\_new\_\_返回一个cls的实例时后面的\_\_init\_\_才能被调用.

当创建一个新实例时调用\_\_new\_\_,初始化一个实例时用\_\_init\_\_.

[stackoverflow](http://stackoverflow.com/questions/674304/pythons-use-of-new-and-init)

ps: \_\_metaclass\_\_是创建类时起作用.所以我们可以分别使用\_\_metaclass\_\_,\_\_new\_\_和\_\_init\_\_来分别在类创建,实例创建和实例初始化的时候做一些小手脚.

\_init\_\_是初始化方法，创建对象后，就立刻被默认调用了，可接收参数。

1、\_\_new\_\_至少要有一个参数cls，代表当前类，此参数在实例化时由Python解释器自动识别；

2、\_\_new\_\_必须要有返回值，返回实例化出来的实例，这点在自己实现\_\_new\_\_时要特别注意，可以return父类（通过super(当前类名, cls)）\_\_new\_\_出来的实例，或者直接是object的\_\_new\_\_出来的实例；

3、\_\_init\_\_有一个参数self，就是这个\_\_new\_\_返回的实例，\_\_init\_\_在\_\_new\_\_的基础上可以完成一些其它初始化的动作，\_\_init\_\_不需要返回值；

4、如果\_\_new\_\_创建的是当前类的实例，会自动调用\_\_init\_\_函数，通过return语句里面调用的\_\_new\_\_函数的第一个参数是cls来保证是当前类实例，如果是其他类的类名，；那么实际创建返回的就是其他类的实例，其实就不会调用当前类的\_\_init\_\_函数，也不会调用其他类的\_\_init\_\_函数。

**146、Python中的作用域？**

Python 中，一个变量的作用域总是由在代码中被赋值的地方所决定的。

当 Python 遇到一个变量的话他会按照这样的顺序进行搜索：本地作用域（Local），当前作用域被嵌入的本地作用域（Enclosing locals），全局/模块作用域（Global），内置作用域（Built-in）；

**151、Python函数式编程？**

这个需要适当的了解一下吧,毕竟函数式编程在Python中也做了引用.推荐: [酷壳](http://coolshell.cn/articles/10822.html)

python中函数式编程支持:

filter 函数的功能相当于过滤器。调用一个布尔函数bool\_func来迭代遍历每个seq中的元素；返回一个使bool\_seq返回值为true的元素的序列。

>>>a = [1,2,3,4,5,6,7]

>>>b = filter(lambda x: x > 5, a)

>>>print b

>>>[6,7]

map函数是对一个序列的每个项依次执行函数，下面是对一个序列每个项都乘以2：

>>> a = map(lambda x:x\*2,[1,2,3])

>>> list(a)

[2, 4, 6]

reduce函数是对一个序列的每个项迭代调用函数，下面是求3的阶乘：

>>> reduce(lambda x,y:x\*y,range(1,4))

6

**152、DNS域名解析过程？**

01、浏览器检查缓存中有没有这个域名对应的解析后的IP地址，如果缓存中有，解析过程结束。缓存大小、时间都有限制，时间由TTL属性决定；

02、如果浏览器缓存中么有，浏览器会查找操作系统缓存中有无这个域名DNS解析后的结果。操作系统也有一个域名解析的过程，windows通过C:\Windows\System32\drivers\etc\hosts，浏览器会优先使用这个解析结果（Win7已将hosts设置为只读），linux系统中/etc/named.conf。目前为止都是在本机完成，如果未完成，才会真正请求域名服务器解析域名。

03、“网络配置”中都会有“DNX服务器地址”，操作系统会把域名发送给这个LDNS，本地区的域名服务器，通常都会提供一个本地互联网接入的DNS解析服务。就在你所在城市的某个角落，通过ipconfig可以看到。

04、如果LDNS仍然没有命中，则向RootServer域名服务器请求解析。

05、根域名服务器向本地域名服务器返回一个所查询域的主域名服务器（gTLD Server）。国际顶级域名服务器（.com、.cn、.org等），全球13台。

06、本地域名服务器（Local DNS Server）再向上一步返回的gTLD发送请求。

07、gTLD返回域名对应NameServer域名服务器地址，通常由你购买域名的服务商提供。

08、NameServer服务器查询域名与IP映射关系表，返回目标IP记录和TTL值给DNS Server域名服务器。

09、Local DNS Server根据TTL缓存该IP解析。

10、缓存结果返回给用户，用户根据TTL缓存到本地操作系统中，域名解析过程结束。

**153、JavaScript(或者jQuery)如何选择一个id为main的容器？**

jquery：$('#id') JavaScript：document.getElementById("id"))

**154、css如何隐藏一个元素？**

display：none

**155、前后端分离的基本原理？**

前后端分离并非仅仅只是一种开发模式，而是一种架构模式（前后端分离架构）。前端项目与后端项目是两个项目，放在两个不同的服务器，需要独立部署，两个不同的工程，两个不同的代码库，不同的开发人员。前后端工程师需要约定交互接口，实现并行开发，开发结束后需要进行独立部署，前端通过Ajax来调用HTTP请求调用后端的restful api。前端只需要关注页面的样式与动态数据的解析&渲染，而后端专注于具体业务逻辑。

**156、如何保证api调用时数据的安全性？**

1、通信使用https；

2、请求签名，防止参数被篡改；

3、身份确认机制，每次请求都要验证是否合法；

4、app中使用ssl pinning防止抓包操作；

5、对所有的请求和响应都进行加解密操作；

**157、如何实现响应式布局？**

1、流体布局：其实就是度量单位的改变。在响应式设计的布局中，不在把像素(px)作为唯一的单位，而是采用%或者是混合%、px为单位，设计出自己想要的布局方式。

2、媒体查询：媒体查询可以在你根据特定的环境下查询到各种属性---------比如设备类型，分辨率、屏幕物理尺寸以及色彩等。通过使用媒体查询，可以获得设备的一些特性，以及响应式的布局方案。

3、弹性图片：其实在做响应式布局时，大多用到的是弹性盒子进行布局，那么在设置图片的地方也应该具有一些变化以适应布局的变化。出了图片外，像图标啦！视频啦也应做一些调整用以适应布局的变化。

**158、曾经使用过哪些前端框架？**

react，vue，bootstrap，elementUI，Echarts

**159、什么是ajax请求？手写一个ajax请求？**

* ajax（异步JavaScript和XML）是指一种创建交付式网页应用的网页开发技术。可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

// 不使用第三方  
var xhr = new XMLHttpRequest();  
xhr.open("GET", url, false);  
xhr.onreadtstatechange = function () {  
   if (xhr.readystate == 4) {  
       //响应内容解析完成，可以在客户端调用了  
       if (xhr.status == 200) {  
           //客户端的请求成功了  
           alert(xhr.responseText);  
      }  
  }  
}  
xhr.send(null);  
// 使用ajax  
$.ajax({  
           type:"GET",  
           url:"",  
           dataType:"json",  
           success:function(data){  
          },  
           error:function(jqXHR){  
          }  
      });

**160、什么是轮询和长轮询？**

* 轮询是在特定的的时间间隔（如每1秒），由浏览器对服务器发出HTTP request，然后由服务器返回最新的数据给客户端的浏览器。这种传统的HTTP request 的模式带来很明显的缺点 – 浏览器需要不断的向服务器发出请求，然而HTTP request 的header是非常长的，里面包含的有用数据可能只是一个很小的值，这样会占用很多的带宽。

var xhr = new XMLHttpRequest();           
   setInterval(function(){  
       xhr.open('GET','/user');   
       xhr.onreadystatechange = function(){ };   
       xhr.send();  
  },1000)

* 长轮询是ajax实现:在发送ajax后,服务器端会阻塞请求直到有数据传递或超时才返回。 客户端JavaScript响应处理函数会在处理完服务器返回的信息后，再次发出请求，重新建立连接。

function ajax(){  
       var xhr = new XMLHttpRequest();  
       xhr.open('GET','/user');  
       xhr.onreadystatechange = function(){  
             ajax();  
      };  
       xhr.send();  
  }

**167、vuex的作用？**

1、组件之间的数据通信；

2、使用单向数据流的方式进行数据的去中心化管理；

**168、vue中的路由拦截器的作用？**

1、当某些页面需要访问权限时，可以使用路由拦截器对用户权限进行判断；

**169、axios的作用？**

axios是基于promise的用于浏览器和nodejs的HTTP客户端，本身有以下特征：

1、从浏览器中创建XMLHttpRequest；

2、从nodejs发出http请求；

3、支持promiseAPI；

4、拦截 请求和响应；

5、转换请求和响应数据；

6、取消请求；

7、自动转换JSON数据；

8、客户端支持防止CSRF/XSRF攻击；

**170、简述jsonp及其原理？**

1、JSONP是JSON with Padding的略称。它是一个非官方的协议，它允许在服务器端集成Script tags返回至客户端，通过javascript callback的形式实现跨域访问（这仅仅是JSONP简单的实现形式）

2、原理：<script>标签的src属性并不被同源策略所约束，所以可以获取任何服务器上脚本并执行。

<script type="text/javascript" src="http://localhost:20002/test.js"></script>

**171、简述http协议的认识以及常用请求头？**

HTTP(超文本传输协议)是一个应用层协议，由请求和相应构成，是一个标准的客户端服务器模型。HTTP通常承载于TCP协议之上，有时也承载于TLS或SSL协议层之上，这个时候，就成了常说的HTTPS。默认HTTP的端口号为80，HTTPS的端口号为443。

**谈谈你对http协议的认识：**浏览器本质，socket客户端遵循Http协议

　　HTTP协议本质：通过\r\n分割的规范，请求响应之后断开链接 ==> 短连接、无状态

具体：

　　Http协议是建立在tcp/ip之上的，是一种规范，它规范定了发送的数据的数据格式，然而这个数据格式是通过\r\n进行分割的，请求头与请求体也是通过2个\r\n分割的，响应的时候，响应头与响应体也是通过\r\n分割，并且还规定已请求已响应就会断开链接，即-->短连接、无状态

常用请求头：

01、Host (主机和端口号)；

02、Accept(传输文件类型)：向服务器申明客户端（浏览器）可以接受的媒体类型（MIME）的资源；

03、Accept-encoding（文件编解码格式）：向服务器申明客户端（浏览器）接收的编码方法，通常为压缩方法；

04、Accept-Language：向服务器申明客户端（浏览器）接收的语言；

05、Cookie：告诉服务器关于 Session 的信息，存储让服务器辨识用户身份的信息；

06、Connection (链接类型)；

07、Referer (页面跳转处)：告诉服务器该页面从哪个页面链接的；

08、User-agent(浏览器名称)：向服务器发送浏览器的版本、系统、应用程序的信息；

09、Upgrade-Insecure-Requests (升级为HTTPS 请求)；

10、x-requested-with :XMLHttpRequest (是Ajax 异步请求)；

**173、简单阐述HTTPS的工作原理？**

1、向CA机构申请数字证书然后配置在server端。

2、client端向server端发送HTTPS请求。

3、server端返回数字证书，加密算法和公钥。

4、client端验证接受到的数字证书是否和本地受信任的证书的信息一致。

5、证书验证有效之后，client端生成一个随机数用于对称加密的密钥。

6、client端用证书里面的公钥对自己生成的密钥进行加密。

7、client端向server端发送该密钥的密文。

8、server端通过私钥来解密该密文，拿到对称加密的密钥。

9、之后client和server就可以通过该密钥来进行加密通信。

**173、简述websocket协议及实现原理？**

WebSocket用于在Web浏览器和服务器之间进行任意的双向数据传输的一种技术。WebSocket协议基于TCP协议实现，包含初始的握手过程，以及后续的多次数据帧双向传输过程。其目的是在WebSocket应用和WebSocket服务器进行频繁双向通信时，可以使服务器避免打开多个HTTP连接进行工作来节约资源，提高了工作效率和资源利用率。

websocket是给浏览器新建的一套（类似于http）协议，协议规定：浏览器和服务器连接之后不断开，以达到服务端向客户端主动推送消息。

本质：创建一个连接后不断开的socket

当连接成功之后：

客户端（浏览器）会自动向服务端发送消息，包含：Sec-WebSocket-Key: iyRe1KMHi4S4QXzcoboMmw==

服务端接收之后，会对于该数据进行加密：base64(sha1(swk + magic\_string))

构造响应头：

HTTP/1.1 101 Switching Protocols\r\n

Upgrade:websocket\r\n

Connection: Upgrade\r\n

Sec-WebSocket-Accept: 加密后的值\r\n

WebSocket-Location: ws://127.0.0.1:8002\r\n\r\n

发给客户端（浏览器）

建立：双工通道，接下来就可以进行收发数据

发送数据是加密，解密，根据payload\_len的值进行处理

payload\_len <= 125

payload\_len == 126

payload\_len == 127

获取内容：

mask\_key

数据

根据mask\_key和数据进行位运算，就可以把值解析出来。

**174、什么是magic string？**

客户端向服务端发送消息时，会有一个'sec-websocket-key'和'magic string'的随机字符串(魔法字符串)，服务端接收到消息后会把他们连接成一个新的key串，进行编码、加密，确保信息的安全性。

**174、django中如何实现websocket？**

通过使用channels模块来实现

**175、python web开发中跨域问题的解决思路？**

1、使用django-cors-headers模块，给跨域增加忽略

2、使用jsonp

3、修改对应的api实现函数views.py，允许其他域通过ajax请求数据。

**176、简述http缓存机制？**

对于强制缓存，服务器通知浏览器一个缓存时间，在缓存时间内，下次请求，直接用缓存，不在时间内，执行比较缓存策略。

对于比较缓存，将缓存信息中的Etag和Last-Modified通过请求发送给服务器，由服务器校验，返回304状态码时，浏览器直接使用缓存。

**177、什么是wsgi？**

WSGI是Python在处理HTTP请求时，规定的一种处理方式。如一个HTTP Request过来了，那么就有一个相应的处理函数来进行处理和返回结果。WSGI就是规定这个处理函数的参数长啥样的，它的返回结果是长啥样的？至于该处理函数的名子和处理逻辑是啥样的，那无所谓。简单而言，WSGI就是规定了处理函数的输入和输出格式。

### 178、谈谈你对restful规范的认识？

推荐: <http://www.ruanyifeng.com/blog/2011/09/restful.html>

RESTful架构，就是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。

一、起源(略)；

二、名称：

Fielding将他对互联网软件的架构原则，定名为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"。

如果一个架构符合REST原则，就称它为RESTful架构。

要理解RESTful架构，最好的方法就是去理解Representational State Transfer这个词组到底是什么意思，它的每一个词代表了什么涵义。如果你把这个名称搞懂了，也就不难体会REST是一种什么样的设计。

首先restful是一种软件架构风格或者说是一种设计风格，并不是标准，它只是提供了一组设计原则和约束条件，主要用于客户端和服务端交互类的软件。 就像设计模式一样，并不是一定要遵循这些原则，而是基于这个风格设计的软件可以更简洁，更有层次，我们可以根据开发的实际情况，做相应的改变。

**它里面提到了一些规范，例如：**

01.restful 提倡面向资源编程,在url接口中尽量要使用名词，不要使用动词；

02、在url接口中推荐使用Https协议，让网络接口更加安全 ；# <https://www.bootcss.com/v1/mycss>？page=3 #（Https是Http的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL， # 因此加密的详细内容就需要SSL（安全套接层协议））；

03、在url中可以体现版本号 # <https://v1.bootcss.com/mycss> #不同的版本可以有不同的接口，使其更加简洁，清晰；

04、url中可以体现是否是API接口 # <https://www.bootcss.com/api/mycss> ；

05、url中可以添加条件去筛选匹配 # <https://www.bootcss.com/v1/mycss>？page=3 ；

06、可以根据Http不同的method，进行不同的资源操作 #（5种方法：GET / POST / PUT / DELETE / PATCH）；

07、响应式应该设置状态码；

08、有返回值，而且格式为统一的json格式；

09、返回错误信息 #返回值携带错误信息；

10、返回结果中要提供帮助链接，即API最好做到Hypermedia；

#如果遇到需要跳转的情况，携带调转接口的URL ：ret = { code: 1000, data:{ id:1, name:'小强', depart\_id:<http://www.luffycity.com/api/v1/depart/8/> } }；

三、资源（Resources）：

REST的名称"表现层状态转化"中，省略了主语。"表现层"其实指的是"资源"（Resources）的"表现层"。

所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。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。

所谓"上网"，就是与互联网上一系列的"资源"互动，调用它的URI。

四、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）。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。

五、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。

六、综述：

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）客户端和服务器之间，传递这种资源的某种表现层；

　　（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

七、误区：

RESTful架构有一些典型的设计误区。

最常见的一种设计错误，就是URI包含动词。因为"资源"表示一种实体，所以应该是名词，URI不应该有动词，动词应该放在HTTP协议中。

举例来说，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

另一个设计误区，就是在URI中加入版本号：

　　http://www.example.com/app/1.0/foo

　　http://www.example.com/app/1.1/foo

　　http://www.example.com/app/2.0/foo

因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个URI。版本号可以在HTTP请求头信息的Accept字段中进行区分（参见Versioning REST Services）：

Accept: vnd.example-com.foo+json; version=1.0

　　Accept: vnd.example-com.foo+json; version=1.1

　　Accept: vnd.example-com.foo+json; version=2.0

### 179、RESTful架构((SOAP,RPC)？

**SOAP（原为Simple Object Access Protocol的首字母缩写，即简单对象访问协议）是交换数据的一种协议规范**，使用在计算机网络Web服务（web service）中，交换带结构信息。

SOAP为了简化网页服务器（Web Server）从XML数据库中提取数据时，节省去格式化页面时间，以及不同应用程序之间按照HTTP通信协议，遵从XML格式执行资料互换，使其抽象于语言实现、平台和硬件。

**RPC（Remote Procedure Call Protocol）——远程过程调用协议**，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。

RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

总结:服务提供的两大流派.传统意义以方法调用为导向通称RPC。为了企业SOA,若干厂商联合推出webservice,制定了wsdl接口定义,传输soap.当互联网时代,臃肿SOA被简化为http+xml/json.但是简化出现各种混乱。以资源为导向,任何操作无非是对资源的增删改查，于是统一的REST出现了.

进化的顺序: RPC -> SOAP -> RESTful；

**远程过程调用 (RPC) 是一种协议，程序可使用这种协议向网络中的另一台计算机上的程序请求服务 ；**

1.RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器；

2.首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息；

3.在服务器端，进程保持睡眠状态直到调用信息到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息；

4.最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行；

**180、什么是rpc与HTTP及应用场景?**

RPC主要用于公司内部的服务调用，性能消耗低，传输效率高，服务治理方便。HTTP主要用于对外的异构环境，浏览器接口调用，APP接口调用，第三方接口调用等...

**179、接口的幂等性是什么意思？**

1、是系统的接口对外一种承诺(而不是实现)；

2、承诺只要调用接口成功，外部多次调用对系统的影响都是一致的，不会对资源重复操作；

**180、为什么要使用API？**

系统间为了调用数据；

数据传输格式：json和xml；

1、跟前端进行交互，确定前端要什么；

2、把需求写成文档保存；

**182、简述MVC和MTV？**

MVC软件系统分为三个基本部分：模型(Model)、视图(View)和控制器(Controller)：

1、Model：负责业务对象与数据库的映射(ORM)

2、View：负责与用户的交互

3、Control：接受用户的输入调用模型和视图完成用户的请求

Django框架的MTV设计模式借鉴了MVC框架的思想,三部分为：Model、Template和View：

1、Model(模型)：负责业务对象与数据库的对象(ORM)；

2、Template(模版)：负责如何把页面展示给用户；

3、View(视图)：负责业务逻辑，并在适当的时候调用Model和Template；

此外,Django还有一个urls分发器,它将一个个URL的页面请求分发给不同的view处理,view再调用相应的Model和Template；

**183、现有字典d = {'a': 24, 'g': 52, 'i': 12, 'k': 33}，如何按字典中的值对字典进行排序得到排序后的字典。**

d = {item[0]: item[1] for item in sorted(d.items(), key=lambda x:x[1])}  
或者  
d = {key:value for key, value in (sorted(d.items(), key=lambda x: x[1]))}  
sorted(d.items(),key = lambda x:x[1])  
使用内置函数sorted   
1.利用zip将字典数据转换成为元组 ：print sorted(zip(dict2.itervalues(),dict2.iterkeys()),reverse=True)；  
2.传递sorted函数的key参数 ：sorted(dict2.iteritems(),key=lambda x:x[1],reverse=True) ；  
3.可以重新获得已经排序的字典 ：dict(sorted(dict2.iteritems(),key=lambda x:x[1],reverse=True))；

**184、实现将字符串k1:v1|k2:v2|k3:v3处理成字典{'k1': 'v1', 'k2': 'v2', 'k3': 'v3'}？**

{key: value for key, value in (  
   item.split(':') for item in 'k1:v1|k2:v2|k3:v3'.split('|')  
)}

**185、写出生成从m到n公差为k的等差数列的生成器？**

(value for value in range(m, n + 1, k))

或

def generate(m, n, k):  
   for value in range(m, n + 1, k):  
       yield value

或

def generate(m, n, k):  
   yield from range(m, n + 1, k)

**186、请写出你能想到的反转一个字符串的方式？**

''.join(reversed('hello'))

或

'hello'[::-1]

或

def reverse(content):

return ''.join(content[i] for i in range(len(content) - 1, -1, -1))

reverse('hello')

或

def reverse(content):

return reverse(content[1:]) + content[0] if len(content) > 1 else content

reverse('hello')

**187、不使用任何内置函数，将字符串'123'转换成整数123？**

nums = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}

total = 0

for ch in '123':

total \*= 10

total += nums[ch]

print(total)

**188、写一个返回bool值的函数，判断给定的非负整数是不是回文数？**

用Python代码来判断是否回文：

def is\_palindrom(s):

if len(s) < 2:

return True

if s[0] == s[-1]:

return is\_palindrom(s[1:-1])

else:

return False

name1 = 'ABBA'

name2 = '1234'

print(is\_palindrom(name1))

print(is\_palindrom(name2))

输出：

True

False

def is\_palindrome(num):

temp = num

total = 0

while temp > 0:

total = total \* 10 + temp % 10

temp //= 10

return total == num

解析1：

num = 12345

temp = num

total: 0 5 54 543 5432 54321

temp: 12345 1234 123 12 1 0

return total == num

解析2：

num = 123321

temp = num

total: 0 1 12 123 1233 12332 123321

temp: 12321 12332 1233 123 12 1 0

**189、用一行代码实现求任意非负整数的阶乘？**

from functools import reduce

(lambda num: reduce(int.\_\_mul\_\_, range(2, num + 1), 1))(5)

reduce(lambda x, y: x\*y, range(2, num + 1), 1)

**190、写一个函数返回传入的整数列表中第二大的元素？**

def second(ln):

max = 0

s = {}

for i in range(len(ln)):

flag = 0

for j in range(len(ln)):

if ln[i] >= ln[j] and i != j:

flag = flag + 1

s[i] = flag

if flag > max:

max = flag

print(s)

for i in s:

if s[i] == max - 1:

break

print(ln[i])

second([1,2,7,4,5,6,8,5,3,3,9,9,10,11,5,3,20,19])

def max2(x):

m1, m2 = (x[0], x[1]) if x[0] > x[1] else (x[1], x[0])

for index in range(2, len(x)):

if x[index] > m1:

m1, m2 = x[index], m1

elif x[index] > m2:

m2 = x[index]

return m1, m2

print(max2([2, 5, 6, 8, 21, 33]))

**191、删除列表中的重复元素并保留原有的顺序？**

方法1：先用set()结合去重，然后转换成列表

l1 = ['b','c','d','c','a','a'] l2 = list(set(l1)) print(l2)

[‘d’, ‘b’, ‘a’, ‘c’]

方法2：循环遍历，再创建一个新的列表，并向里面添加元素。

l1 = ['b','c','d','c','a','a'] l2 = [] for i in l1: if not i in l2: l2.append(i) print(l2)

方法3：

l1 = ['b','c','d','c','a','a'] l2 = sorted(set(l1),key=l1.index) print(l2)

list1 = [2, 3, 2, 12, 5, 25, 3]

new\_list1 = []

mid\_list1 = [new\_list1.append(item) for item in list1 if item not in new\_list1]

list1 = [2, 3, 2, 12, 5, 25, 3]

new\_lsit1 = list(set(list1))

new\_list1.sort(key=list1.index)

**192、找出两个列表中的相同元素和不同元素？**

# 接口返回值

list1 = ['张三', '李四', '王五', '老二']

# 数据库返回值

list2 = ['张三', '李四', '老二', '王七']

a = [x for x in list1 if x in list2] #两个列表表都存在

b = [y for y in (list1 + list2) if y not in a] #两个列表中的不同元素

print('a的值为:',a)

print('b的值为:',b)

c = [x for x in list1 if x not in list2] #在list1列表中而不在list2列表中

d = [y for y in list2 if y not in list1] #在list2列表中而不在list1列表中

print('c的值为:',c)

print('d的值为:',d)

输出为

a的值为: ['张三', '李四', '老二']

b的值为: ['王五', '王七']

c的值为: ['王五']

d的值为: ['王七']

用列表推导式来写

list1 = [1,3,65,2,7]

list2 = [3,2,5,4]

c = [x for x in list1 if x in list2]

d = [y for y in (list1+list2) if y not in c]

print(c)

print(d)

用普通的for循环来写

list1 = [1,3,65,2,7]

list2 = [3,2,5,4]

list3 = []

list4 = []

for i in list1:

for j in list2:

if i == j:

list3.append(i)

for b in (list1 + list2):

if b not in list3:

list4.append(b)

print(list3)

print(list4)

list1 = [1, 2, 33, 4, 21, 6]

list2 = [2, 3, 44, 5, 6, 7]

相同的元素：list(set(list1) & set(list2))

不同的元素: list(set(list1) ^ set(list2))

**193、列表中的某个元素出现次数占列表元素总数的半数以上，找出这个元素？**

list1 = [1, 1, 2, 5, 6, 5, 1, 1, 1, 44, 1]

result = [item for item in set(list1) if list1.count(item) > len(list1) // 2]

**194、实现对有序列表进行二分查找的算法？**

def binary\_search(olist, num):

low = 0

heigh = len(olist) - 1

while low <= heigh:

mid = int((low + heigh) / 2)

guess\_num = olist[mid]

if guess == num:

return guess

if guess < olist[mid]:

low = mid + 1

else:

heigh = mid - 1

return None

def main():

olist = [1, 3, 5 ,7, 12, 23, 34, 56, 67, 78, 89, 90, 100]

print(binary\_search(olist, 78))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**195、输入年月日，输出这一天是这一年的第几天？**

def is\_leap\_year(year):

return year % 4 == 0 and year % 100 != 0 or year % 400 == 0

def which\_day(year, month, day):

days\_of\_month = [[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],

[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]][is\_leap\_year(year)]

total = 0

for index in range(month - 1):

total += days\_of\_month[index]

total += day

return f'这一天是这一年的第{total}天'

def main():

print(which\_day(1980, 11, 28))

print(which\_day(1981, 12, 31))

print(which\_day(2018, 1, 1))

print(which\_day(2019, 12, 24))

if \_\_name\_\_ == '\_\_main\_\_':

main()

**196、统计一个字符串中各个字符出现的次数？**

str1 = 'hello, world'

count\_str = {item:str1.count(item) for item in str1}

**197、下面的代码会输出什么？**

class A:

def \_\_init\_\_(self, value):

self.\_\_value = value

@property

def value(self):

return self.\_\_value

a = A(1)

a.\_\_value = 2

print(a.\_\_value) # 2

print(a.value) # 1

**198、实现一个记录函数执行时间的装饰器？**

import time

def runtime(func):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

fn = func(\*args, \*\*kwargs)

execute\_time = time.time() - start\_time

return fn

return wrapper

**199、写一个遍历指定目录下指定后缀名的文件的函数？**

import os

def find\_file\_ext(filepath, ext):

file\_list = os.listdir(filepath)

result = []

for item in file\_list:

file\_ext = os.path.splitext(item)[1]

if file\_ext == ext:

result.append(item)

return result

**200、有如下所示的字典，请将其转换为CSV格式？**

转换前：

dict\_corp = {

'cn': {'id': 1, 'name': '土豆', 'desc': '土豆', 'price': {'gold': 20, 'kcoin': 20}},

'en': {'id': 1, 'name': 'potato', 'desc': 'potato', 'price': {'gold': 20, 'kcoin': 20}},

'kr': {'id': 1, 'name': '감자', 'desc':'감자', 'price': {'gold': 20, 'kcoin': 20}},

'jp': {'id': 1, 'name': 'ジャガイモ', 'desc': 'ジャガイモ', 'price': {'gold': 20, 'kcoin': 20}},

}

转换后：

,id,name,desc,gold,kcoin

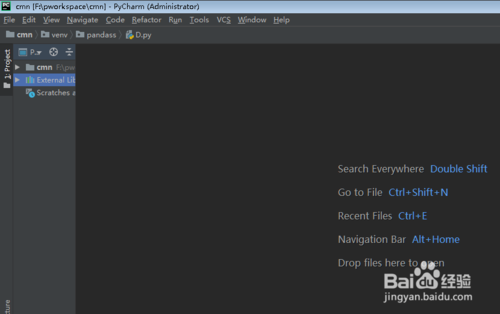
cn,1,土豆,土豆,20,20

en,1,potato,potato,20,20

kr,1,감자,감자,20,20

jp,1,ジャガイモ,ジャガイモ,20,20

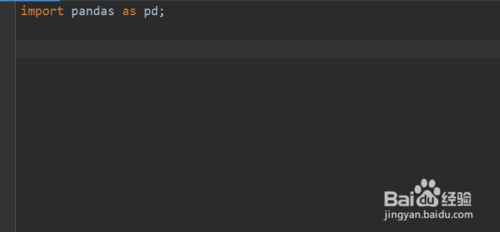
* 1. 打开pycharm开发工具，新建python项目，检查pandas是否安装

[](http://jingyan.baidu.com/album/fc07f989cd89b552fee51967.html?picindex=2)

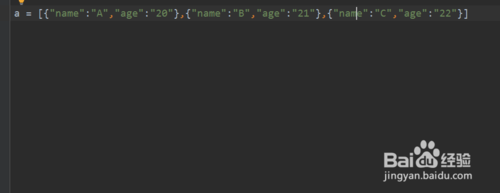
* 1. 在项目指定的包下鼠标右键，新建python文件，输入文件名称，选择Python file

[](http://jingyan.baidu.com/album/fc07f989cd89b552fee51967.html?picindex=3)

* 1. 创建完毕后，导入pandas库，使用import命令

[](http://jingyan.baidu.com/album/fc07f989cd89b552fee51967.html?picindex=4)

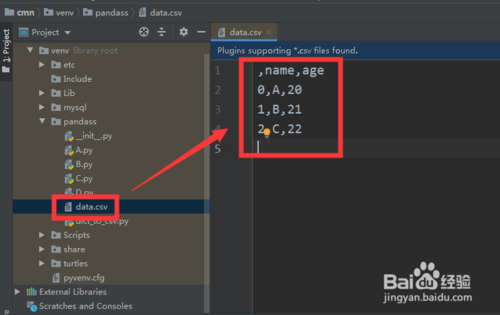
* 1. 定义一个列表变量a，元素是由字典构成的，由多个字典组成

[](http://jingyan.baidu.com/album/fc07f989cd89b552fee51967.html?picindex=5)

* 1. 调用pandas库中的DataFrame，将a转换成矩阵；然后调用to\_csv方法，导入数据到csv文件

[](http://jingyan.baidu.com/album/fc07f989cd89b552fee51967.html?picindex=6)

* 1. 保存代码并直接运行，查看在Python文件同目录下是否生成一个csv文件

[](http://jingyan.baidu.com/album/fc07f989cd89b552fee51967.html?picindex=7)

END

**总结**

* 1. 1、打开pycharm工具

2、新建Python项目

3、创建Python文件

4、导入pandas库

5、定义字典列表

6、调用方法转换

**201、有如下所示的日志文件，请用Python程序或Linux命令打印出独立IP并统计数量？**

221.228.143.52 - - [23/May/2019:08:57:42 +0800] ""GET /about.html HTTP/1.1"" 206 719996

218.79.251.215 - - [23/May/2019:08:57:44 +0800] ""GET /index.html HTTP/1.1"" 206 2350253

220.178.150.3 - - [23/May/2019:08:57:45 +0800] ""GET /index.html HTTP/1.1"" 200 2350253

218.79.251.215 - - [23/May/2019:08:57:52 +0800] ""GET /index.html HTTP/1.1"" 200 2350253

219.140.190.130 - - [23/May/2019:08:57:59 +0800] ""GET /index.html HTTP/1.1"" 200 2350253

221.228.143.52 - - [23/May/2019:08:58:08 +0800] ""GET /about.html HTTP/1.1"" 206 719996

221.228.143.52 - - [23/May/2019:08:58:08 +0800] ""GET /news.html HTTP/1.1"" 206 713242

221.228.143.52 - - [23/May/2019:08:58:09 +0800] ""GET /products.html HTTP/1.1"" 206 1200250

解题思路：

1，读取log内容，过滤出ip地址，保存到一个列表a中

2，将列表a去重获得列表b

3，以b列表长度为最大循环次数做循环获得列表a1，将a1的值添加到b1，然后重置a1

4，将b1进行反向排序，打印出来

5，让使用者选择输出访问最多的前几个ip，也可以给函数传递参数，给一个默认参数打印前几个

import re

import sys

def getIP():

log = raw\_input('请输入要查询的日志文件名，包括扩展名：')

re1 = r'^\d+\.\d+\.\d+\.\d+'

reg = re.compile(re1)

fp = open(log)

f\_content = fp.readlines()

fp.close()

a = []

for lines in f\_content:

ip = re.findall(reg,lines)

a.append(ip[0])

b = list(set(a))

b1 = []

for i in range(len(b)):

a1 = []

a1.append(a.count(b[i]))

a1.append(b[i])

b1.append(a1)

b1.sort(reverse=True)

k = 1

num = raw\_input('请输入一个数字要查询前几个ip:')

num = int(num)

for j in b1:

print j[0],j[1]

if k == num:

break

k += 1

if \_\_name\_\_ == '\_\_main\_\_':

getIP()

取消了re正则匹配的麻烦：

# !/usr/bin/env python

# coding:utf-8

def getIP():

log = raw\_input('请输入要查询的日志文件名，包括扩展名：')

fp = open(log)

f\_lines = fp.readlines()

fp.close()

ips = []

for line in f\_lines:

ip = line.split()[0]

ips.append(ip)

b = list(set(ips))

b1 = []

for i in range(len(b)):

a1 = []

a1.append(ips.count(b[i]))

a1.append(b[i])

b1.append(a1)

b1.sort(reverse=True)

k = 1

num = raw\_input('请输入一个数字要查询前几个ip:')

num = int(num)

for j in b1:

print j[0],j[1]

if k == num:

break

k += 1

if \_\_name\_\_ == '\_\_main\_\_':

getIP()

shell 只需一条命令：

awk '{print $1}' www.log |sort|uniq -c|sort -nr|head -n5

**202、请写出从HTML页面源代码中获取a标签href属性的正则表达式？**

pattern = re.complie('<a href="(.\*?)"')

result = re.findall(pattern, html)

python正则表达式之获取a标签href属性

url = re.compile(r'<h2.\*?><a\b[^>]+\bhref=\\"([^"]\*)"[^>]\*>.+?</a></h2>').findall(li)

<h2.?> #表示在h2标签内

<a\b #匹配a标签的开始

[^>]+ #匹配a标签中href之前的内容

\bhref="([^"])" #匹配href的值，并将匹配内容捕获到分组1当中

[^>]\*> #匹配a标签中href之后的内容

.+? #匹配a标签内的内容

#匹配a标签的结束

（）表示需要取的内容

我们要根据自己的文本做适当的改变，匹配都得对应上，再能取到我们需要的内容

**203、正则表达式对象的search和match方法有什么区别？**

search()函数会在**整个字符串内查找**模式匹配,只到找到第一个匹配然后返回一个包含匹配信息的对象,该对象可以通过调用group()方法得到匹配的字符串,如果字符串没有匹配，则返回None。

import re

print(re.search("tion", "function"))

# 打印结果 <\_sre.SRE\_Match object; span=(4, 8), match='tion'>

print(re.search("tion", "function").span())

# 打印结果 (4, 8)

print(re.search("tion1", "function"))

# 打印结果 None

match()函数**只检测字符串开头位置**是否匹配，匹配成功才会返回结果，否则返回None

import re

print(re.match("func", "function"))

# 打印结果 <\_sre.SRE\_Match object; span=(0, 4), match='func'>

print(re.match("func", "function").span())

# 打印结果 (0, 4)

print(re.match("func1", "function"))

# 打印结果 None

**001、Python的函数参数传递：**

看两个例子:

a = 1

def fun(a):

a = 2

fun(a)

print a # 1

a = []

def fun(a):

a.append(1)

fun(a)

print a # [1]

所有的变量都可以理解是内存中一个对象的“引用”，或者，也可以看似c中void\*的感觉。

通过id来看引用a的内存地址可以比较理解：

a = 1

def fun(a):

print "func\_in",id(a) # func\_in 41322472

a = 2

print "re-point",id(a), id(2) # re-point 41322448 41322448

print "func\_out",id(a), id(1) # func\_out 41322472 41322472

fun(a)

print a # 1

注：具体的值在不同电脑上运行时可能不同。

可以看到，在执行完a = 2之后，a引用中保存的值，即内存地址发生变化，由原来1对象的所在的地址变成了2这个实体对象的内存地址。

而第2个例子a引用保存的内存值就不会发生变化：

a = []

def fun(a):

print "func\_in",id(a) # func\_in 53629256

a.append(1)

print "func\_out",id(a) # func\_out 53629256

fun(a)

print a # [1]

这里记住的是类型是属于对象的，而不是变量。而对象有两种,“可更改”（mutable）与“不可更改”（immutable）对象。在python中，strings, tuples, 和numbers是不可更改的对象，而 list, dict, set 等则是可以修改的对象。(这就是这个问题的重点)

当一个引用传递给函数的时候,函数自动复制一份引用,这个函数里的引用和外边的引用没有半毛关系了.所以第一个例子里函数把引用指向了一个不可变对象,当函数返回的时候,外面的引用没半毛感觉.而第二个例子就不一样了,函数内的引用指向的是可变对象,对它的操作就和定位了指针地址一样,在内存里进行修改.

如果还不明白的话,这里有更好的解释: http://stackoverflow.com/questions/986006/how-do-i-pass-a-variable-by-reference

**003、如何实现字符串的反转?如:name="张全蛋"请反转为name= "蛋全张"**

name = "张全蛋"

name[::-1]

python数据类型说一下 答：巴拉巴拉。

ython基本数据类型一般分为：数字、字符串、列表、元组、字典、集合这六种基本数据类型。

数字

|  |  |
| --- | --- |
| 1  2  3  4  5 | 数字类型是不可更改的对象。对变量改变数字值就是生成/创建新的对象。Python支持多种数字类型：  整型(标准整型和长整型(Python2.7及之前的有这种类型))、布尔型、双精度浮点型、十进制浮点型、复数。  标准整型  int，标准整型，在大多数32位机器上标准整型取值范围是-2^31到2^31-1，也就是-2147483648~2147483647，如果在64位机器使用64位  编译器，那么这个系统的标准整型将是64位。 |

布尔型

bool，从Python2.3开始Python中添加了布尔类型。布尔类型有两种True和False。对于没有\_\_nozero\_\_方法的对象默认是True。

对于值为0的数字、空集（空列表、空元组、空字典等）在Python中的布尔类型中都是False。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> bool(1)  True  >>> bool('a')  True  >>> bool(0)  False  >>> bool('')  False |

浮点型

|  |  |
| --- | --- |
| 1  2  3  4  5 | float，每个浮点型占8个字节（64位），完全遵守IEEE754号规范（52M/11E/1S）,其中52个位用于表示底，11个位用于表示指数（可表示  的范围大约是±10\*\*308.25），剩下的一个位表示符号。这看上去相当完美，然而，实际精度依赖于机器架构和创建Python解释器的编译  器。  浮点型值通常都有一个小数点和一个可选的后缀e（大写或小写，表示科学计数法）。在e和指数之间可以用正（+）或负（-）表示指数  的正负（正数的话可以省略符号）。 |

以上是Python核心编程的对浮点型(双精度浮点型)的说明。经过Python实测浮点型默认长度是24字节如果超出这个范围会自动

复数类型

complex，在复数中虚数不能单独存在，它们总是和一个值为0.0的实数部分一起来构成一个复数。

|  |  |
| --- | --- |
| 1  2  3  4 | 复数由实数部分和虚数部分构成。  表示虚数的语法：real+imagj。  实数部分和虚数部分都是浮点型。  虚数部分必须有后缀j或J。 |

字符串

|  |  |
| --- | --- |
| 1  2  3  4 | 字符串(string)是不可变类型，就是说改变一个字符串的元素需要新建一个新的字符串。字符串是由独立的字符组成的，并且这些字符  可以通过切片操作顺序地访问。Python里面通过在引号间包含字符的方式创建字符串，单引号和双引号的作用是相同的。  Python用“原始字符串”操作符来创建直接量字符串，所以再做区分就没什么意义了。其他的语言，比如C语言里面用单引号来标示  字符，双引号标示字符串，而在Python里面没有字符这个类型。这可能是双引号和单引号在Python里面被视作一样的另一个原因。 |

Python实际上有三类字符串。通常意义的字符串（str） 和Unicode字符串（unicode） 实际上都是抽象类basestring的子类。这个basestring是不能实例化的。

列表

|  |  |
| --- | --- |
| 1  2 | 列表(list)像字符串类型一样，列表类型也是序列式的数据类型。字符串只能由字符组成，而且是不可变的（不能单独改变它的某  个值），而列表则是能保留任意数目的Python对象的灵活的容器。 |

列表不仅可以包含Python的标准类型，而且可以用用户定义的对象作为自己的元素。列表可以包含不同类型的对象，而且要比C或者Python自己的数组类型（包含在array扩展包中）都要灵活，因为数组类型所有的元素只能是一种类型。列表可以执行pop,sort、reverse等操作。列表也可以添加或者减少元素，还可以跟其他的列表结合或者把一个列表分成几个。可以对单独一个元素或者多个元素执行insert、update或remove操作。

元组

|  |  |
| --- | --- |
| 1  2  3 | 元组类型在很多操作上都跟列表一样，许多用在列表上的例子在元组上照样能跑，我们有一节内容专门讲解元组类型。它们的主要不同  在于元组是不可变的，或者说是只读的，所以那些用于更新列表的操作，比如用切片操作来更新一部分元素的操作，就不能用于元  组类型。 |

字典

字典是Python语言中唯一的映射类型。映射类型对象里哈希值（键，key） 和指向的对象（值。value） 是一对多的关系。一个字典对象是可变的，它是一个容器类型，能存储任意个数的Python对象，其中也包括其他容器类型。字典类型和序列类型容器类（列表、元组） 的区别是存储和访问数据的方式不同。序列类型只用数字类型的键（从序列的开始起按数值顺序索引） 。映射类型可以用其他对象类型做键，一般最常见的是用字符串做键。和序列类型的键不同，映像类型的键直接或间接地和存储的数据值相关联。但因为在映射类型中，我们不再用“序列化排序”的键，所以映像类型中的数据是无序排列的映射类型不要求用数字值做索引以从一个容器中获取对应的数据项。你可以用键直接“映射”到值，这就是为什么叫映射类型（“mappingtype”） 的原因。映射类型通常被称做哈希表，是因为字典对象就是哈希类型的。字典是Python中最强大的数据类型之一。

字典常见的操作有：字典创建、字典赋值、字典中值的访问、字典更新、字典元素删除等操作。

集合

集合最早出现在Python2.3版本中，通过集合模块来创建，并通过ImmutableSet类和Set类进行访问。

集合有两种不同的类型，可变集合（set）和不可变集合（frozenset）。可变集合，你可以添加和删除元素，对不可变集合则不允许这样做。请注意，可变集合不是可哈希的，因此既不能用做字典的键也不能做其他集合中的元素。不可变集合则正好相反，即，他们有哈希值，能被用做字典的键或是作为集合中的一个成员。集合对象是一组无序排列的可哈希的值。

集合支持用in和not in操作符检查成员，由len()内建函数得到集合的基数（大小） ，用for循环迭代集合的成员。但是因为集合本身是无序的，你不可以为集合创建索引或执行切片（slice） 操作，也没有键可用来获取集合中元素的值。更多学习内容，请点击[Python学习网。](https://www.py.cn/)

见一下列表元祖和字典 答：讲一下可变不可变，有序无序

定义方法的不同：

列表    定义方法：list = ['abc',  123 ]

元组    定义方法：tup = ('abc',  123)

字典    定义方法：dict = {'abc': 123, 'efg': 456}

集合    定义方法：parame = {‘abc’, 123 }   或者    set(value)

列表——list：类型相同的元素，可以改变元素的值，即可进行增删改查操作。用中括号表示[num1,num2,num3]

元组——tuple：和列表功能相似，但不能改变其元素的值，即不可以进行增删改的操作，只能执行查询操作。用小括号表示(num1,num2,num3)

字典——infor：类型不同的元素，由键值对组成，与列表一样可以进行增删改查。用大括号表示{key1:value1,key2:value2,key3：value3}

列表中的元素可以是元组和字典。

另外，在python中，大部分参数是不可以被修改的，如字符串，数字，元组，而列表和字典中的内容是可以被修改的，所以在字典中，列表和字典是不可以被当作key值的。

列表和元组是Python中最常用的两种数据结构，字典是第三种。 相同点：

都是序列

都可以存储任何数据类型

可以通过索引访问

语法差异

使用方括号[]创建列表，而使用括号()创建元组

是否可变

列表是可变的，而元组是不可变的，这标志着两者之间的关键差异。

我们可以修改列表的值，但是不修改元组的值。

由于列表是可变的，我们不能将列表用作字典中的key。 但可以使用元组作为字典key。

重用与拷贝

元组无法复制。 原因是元组是不可变的。 如果运行tuple(tuple\_name)将返回自己。

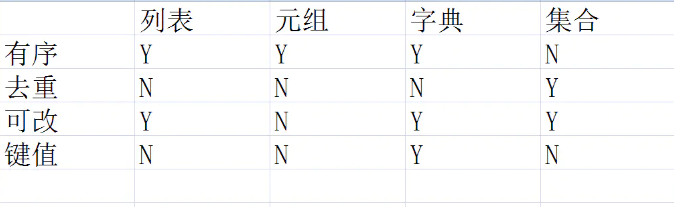
大小差异

Python将低开销的较大的块分配给元组，因为它们是不可变的。 对于列表则分配小内存块。 与列表相比，元组的内存更小。 当你拥有大量元素时，元组比列表快。列表的长度是可变的。

同构与异构

习惯上元组多用于用于存储异构元素，异构元素即不同数据类型的元素，比如(ip,port)。 另一方面，列表用于存储异构元素，这些元素属于相同类型的元素，比如[int1,in2,in3]。

各自特性的异同：



字典内部是怎么存储数据的 答：开辟一块内存空间，申请一个名称空间，指向内存空间。

前几天出去面试的时候，有被问到字典类型的数据在内存中是如何存储的，说实话从事开发工作3年来还真没有研究过这个问题，于是我也是看了几篇文章才学到的，那么我就来讲解一下这个问题

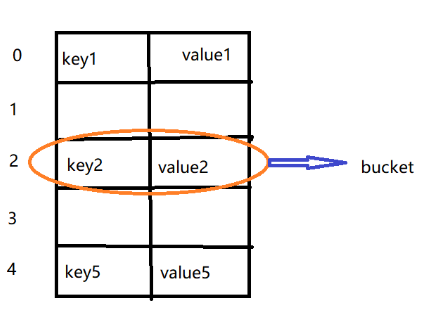
在这之前呢，我们说说什么是hash（摘要百度百科）：

Hash，一般翻译做散列、杂凑，或音译为哈希，是把任意长度的输入（又叫做预映射pre-image）通过散列算法变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

字典的底层原理

字典对象的核心其实是个散列表，而散列表是一个稀疏数组(不是每个位置都有值),每个单元叫做bucket,每个bucket有两部分:一个是键对象的引用,一个是值对象的引用，由于,所有bucket结构和大小一致,我们可以通过偏移量来指定bucket的位置

散列表如图：



将一对键值放入字典的过程：

先定义一个字典，再写入值

dict = {}

dict["age"] = "ming"

在执行第二行时，键name = 值ming放入字典中时，第一步就是计算name的散列值，python中可以用hash函数得到hash值，再用再将得到的值放入bin函数，返回int类型的二进制

print(bin(hash("age")))

# 结果：0b100010001110001000100010110000101101011110100010000101011001111

假设数组长度为10，我们取出计算出的散列值,最右边3位数作为偏移量,即111,十进制是数字7,我们查看偏移量为7对应的bucket的位置是否为空,如果为空,则将键值放进去,如果不为空,依次取右边3位作为偏移量001,十进制是数字1,再查看偏移量1的bucket是否为空,直到单元为空的bucket将键值放进去

以上就是字典的存储原理

字典的查询原理

dict['age'], dict.get('age')

散列值：0b100010001110001000100010110000101101011110100010000101011001111

查询就很简单了，第一步与存储一样，先计算键的散列值，取出后三位111，十进制为7的偏移量，找到对应的bucket的位置，查看是否为空，如果为空就返回None，不为空就获取键并计算键的散列值，计算后将刚计算的散列值与要查询的键的散列值比较，相同就返回对应bucket位置的value，不同就往前再取三位重新计算偏移量，依次取完后还是没有结果就返回None

可以做键的对象条件

键必须可散列

       数字,字符串,元组,都是可散列的

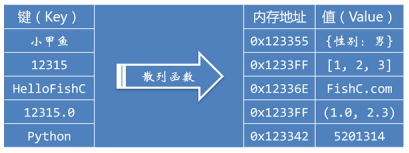
自定义对象做键的条件

       对象内部，有\_\_hash\_\_()和\_\_eq\_\_()方法。

Python 的字典有好多个名称（"映射"、"哈希"、"散列"或者"关系数组"），那你知道为什么字典会被称为 Hash（翻译为"哈希"或"散列"）吗？  
你知道为什么字典对于键（Key）的存储数据要求比较严格，但对于对应的值（Value）的存储却要求很宽松吗？  
读完这篇文章，你将深刻理解这些问题背后的真相！  
首先我们来解释一下什么是 Hash，来自维基百科：

散列函数（或散列算法，又称哈希函数，英语：Hash Function）是一种从任何一种数据中创建小的数字“指纹”的方法。散列函数把消息或数据压缩成摘要，使得数据量变小，将数据的格式固定下来。该函数将数据打乱混合，重新创建一个叫做散列值的指纹。散列值通常用来代表一个短的随机字母和数字组成的字符串。好的散列函数在输入域中很少出现散列冲突。在散列表和数据处理中，不抑制冲突来区别数据，会使得数据库记录更难找到。

世界上没有两片完全相同的树叶，也没有两个相同的指纹，散列函数是用于从数据中创建小的数字指纹的方法。

我们看下图示：  


如图，Python 调用内部的散列函数，将键（Key）作为参数进行转换，得到一个唯一的地址（这也就解释了为什么给相同的键赋值会直接覆盖的原因，因为相同的键转换后的地址是一样滴），然后将值（Value）存放到该地址中。  
对于 Python 来说，键（Key）必须是可哈希的，换句话说就是要可以通过散列函数计算出唯一地址的。那如果拿一个变量当键（Key）可以吗？肯定不行。因为变量随时都可能改变，不符合可哈希原则！  
同样的，列表、字典、集合这些都是可变的，所以都不能做为键（Key）来使用。  
那有朋友可能会问，那元组呢？元组总该是不变的吧？  
其实不然，因为元组里边可以存放列表这类可变因素，所以如果实在想拿元组当字典的键（Key），那必须对元祖做限制：元组中只包括像数字和字符串这样的不可变元素时，才可以作为字典中有效的键（Key）。  
另外还需要注意的一点是，Python 的哈希算法对相同的值计算得到的结果是一样的，也就是说 12315 和 12315.0 的值相同，他们被认为是相同的键（Key）。

多进程和多线程怎么开 答：直接开或者池，扯到gil锁。

**006、现有字典 dict={‘a’:24，‘g’:52，‘i’:12，‘k’:33}请按字典中的 value 值进行排序？**

dic={'a':24,'g':52,'i':12,'k':33}

a = sorted(dic.items(),key= lambda x:x[1])

print(dict(a))

'''

sorted 语法：sorted(iterable, cmp=None, key=None, reverse=False)

参数说明：

iterable -- 可迭代对象。

cmp -- 比较的函数，这个具有两个参数，参数的值都是从可迭代对象中取出，此函数必须遵守的规则为，大于则返回1，小于则返回-1，等于则返回0。

key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。

reverse -- 排序规则，reverse = True 降序 ， reverse = False 升序（默认）。

返回值：返回重新排序的列表。

'''

**008、Python自省？**

这个也是python彪悍的特性.

自省就是面向对象的语言所写的程序在运行时,所能知道对象的类型.简单一句就是运行时能够获得对象的类型.比如type(),dir(),getattr(),hasattr(),isinstance().

a = [1,2,3]

b = {'a':1,'b':2,'c':3}

c = True

print type(a),type(b),type(c) # <type 'list'> <type 'dict'> <type 'bool'>

print isinstance(a,list) # True

**011、下面代码数据的结果是什么？**

list = ['a'， 'b'， 'c'， 'd'， 'e']

print(list[10:])

'''

下面的代码将输出[]，不会产生IndexError错误。就像所期望的那样，尝试用超出成员的个数的index来获取某个列表的成员。例如，尝试获取list[10]和之后的成员，会导致IndexError。'''

**012、写一个列表生成式，产生一个公差为11的等差数列**

a = [i\*11 for i in range(10) ]

print(a)

**013、给定两个列表，怎么找出他们相同的元素和不同的元素?**

1. list1 = [1，2，3]

2. list2 = [3，4，5]

3. set1 = set(list1)

4. set2 = set(list2)

5. print(set1&set2)

6. print(set1^set2)

**014、请写出一段Python代码实现删除一个list里面的重复元素?**

1. l1 = ['b'，'c'，'d'，'b'，'c'，'a'，'a']

2. l2 = list(set(l1))

如果想要保持他们原来的顺序用list类的sort方法：

1. l1 = ['b'，'c'，'d'，'b'，'c'，'a'，'a']

2. l2 = list(set(l1))

3. l2.sort(key=l1.index)#按照索引排个序

4. print(l2)

还可以用遍历

1. l1 = ['b', 'c', 'd', 'b', 'c', 'a', 'a']

2. l2 = []

3. for i in l1:

4. if not i in l2:

5. l2.append(i)

6. print(l2)

**015、有一个list["This","is","a","Boy","!"],所有元素都是字符串,对他进行大小写 无关的排序**

l1 = ['This','is','a','Boy','!']

print(sorted(l1))

**016、列举字符串、列表、元组、字典每个常用的5个方法?**

字符串：replace、strip、split、rsplit、upper、lower、join、reverse。。。

列表：append、pop、insert、remove、sort、count、index

元组：len、index、count、dir

字典：get、keys、items、pop、values、popitems、clear、update。。。

**018、1 < (2 == 2)和1 < 2 == 2的结果分别是什么, 为什么？**

print(1 < (2 == 2)) --> False  
print(1 < 2 == 2) --> True

**023、给出下面代码片段的输出？**

def say\_hi(func):

def wrapper(\*args, \*\*kwargs):

print("HI")

ret = func(\*args, \*\*kwargs)

print("BYE")

return re

return wrapper

def say\_yo(func):

def wrapper(\*args, \*\*kwargs):

print("YO")

return func(\*args, \*\*kwargs)

return wrapper

@say\_hi

@say\_yo

def func():

print("ROCK & ROLL")

func()

# 答案

'''

HI

YO

ROCK & ROLL

BYE

'''

**025、为什么函数名字可以当参数使用？**

python中一切皆对象，函数名在内存中的空间，也是一个对象。

**026、递归函数停止的条件？**

递归函数终止的条件一般定义在递归函数内部，在递归调用前要做一个条件判断，根据判断的结果选择是继续调用自身还是return，终止递归。

终止递归的条件：

1）判断递归的次数是否达到某一限定值

2）判断运算的结果是否达到某个范围，根据设计的目的来选择

**027、Python中如何动态设置和获取对象属性？**

setattr(object, name, value)和getattr(object, name[, default])内置函数，其中object是对象，name是对象的属性名，value是属性值。这两个函数会调用对象的\_\_getattr\_\_和\_\_setattr\_\_魔术方法。

**027、getattr(object, name[,default]) 函数：**

获取对象object的属性或者方法，如果存在则打印出来，如果不存在，打印默认值，默认值可选。  
注意：如果返回的是对象的方法，则打印结果是：方法的内存地址，如果需要运行这个方法，可以在后  
面添加括号()。

1. functiondemo = function\_demo()

2. getattr(functiondemo, 'name') #获取name属性，存在就打印出来--- demo

3. getattr(functiondemo, "run") #获取run方法，存在打印出 方法的内存地址---<bound method function\_demo.run of <\_\_main\_\_.function\_demo object at 0x10244f320>>

4. getattr(functiondemo, "age") #获取不存在的属性，报错如下：

5. Traceback (most recent call last):

6. File "/Users/liuhuiling/Desktop/MT\_code/OpAPIDemo/conf/OPCommUtil.py", line 39, in <module>

7. res = getattr(functiondemo, "age")

8. AttributeError: 'function\_demo' object has no attribute 'age'

9. getattr(functiondemo, "age", 18) #获取不存在的属性，返回一个默认值

**setattr(object,name,values)函数：**

给对象的属性赋值，若属性不存在，先创建再赋值

1．class function\_demo(object):

2． name = 'demo'

3． def run(self):

4． return "hello function"

5．functiondemo = function\_demo()

6．res = hasattr(functiondemo, 'age') # 判断age属性是否存在，False

7．print(res)

8．setattr(functiondemo, 'age', 18 ) #对age属性进行赋值，无返回值

9．res1 = hasattr(functiondemo, 'age') #再次判断属性是否存在，True

**综合使用：**

1．class function\_demo(object):

2． name = 'demo'

3． def run(self):

4． return "hello function"

5．functiondemo = function\_demo()

6．res = hasattr(functiondemo, 'addr') # 先判断是否存在if res:

7． addr = getattr(functiondemo, 'addr')

8． print(addr)else:

9． addr = getattr(functiondemo, 'addr', setattr(functiondemo, 'addr', '北京首都'))

10． #addr = getattr(functiondemo, 'addr', '美国纽约')

11． print(addr)

**031、os.path和sys.path的区别？**

os.path是module，包含了各种处理长文件名(路径名)的函数。

sys.path是由目录名构成的列表，Python 从中查找扩展模块( Python 源模块, 编译模块,或者二进制扩展). 启动 Python 时,这个列表从根据内建规则,PYTHONPATH 环境变量的内容, 以及注册表( Windows 系统)等进行初始化.

**032、有一个jsonline格式的文件爱file.txt 大小约为10K？**

def get\_lines():

l = []

with open('file.txt','rb) as f:

for eachline in f:

l.append(eachline)

return l

if \_\_name\_\_ == '\_\_main\_\_':

for e in get\_lines():

**process(e)** #处理每一行数据

现在要处理一个大小为10G的文件，但是内存只有4G，如果在只修改get\_lines 函数而其他代码保持不变的情况下，应该如何实现？需要考虑的问题都有那些？

**def** **get\_lines**():

l = []

**with** open('file.txt','rb') **as** f:

data = f.readlines(60000)

l.append(data)

**yield** l

要考虑的问题有：内存只有4G无法一次性读入10G文件，需要分批读入分批读入数据要记录每次读入数据的位置。分批每次读取数据的大小，太小会在读取操作花费过多时间。

**033、补充缺失的代码？**

**def** **print\_directory\_contents**(sPath):

"""

这个函数接收文件夹的名称作为输入参数

返回该文件夹中文件的路径

以及其包含文件夹中文件的路径

"""

**import** os

**for** sChild **in** os.listdir(sPath):

sChildPath = os.path.join(sPath,sChild)

**if** os.path.isdir(sChildPath):

print\_directory\_contents(sChildPath)

**else**:

print(sChildPath)

**034、输入日期， 判断这一天是这一年的第几天？**

**import** datetime

def **dayofyear**():

year = input("请输入年份: ")

month = input("请输入月份: ")

day = input("请输入天: ")

date1 = datetime.data(year=**int**(year),month=**int**(month),day=**int**(day))

date2 = datatime.date(year=**int**(year),month=1,day=1)

**return** (date2-date1 +1).days

**035、打乱一个排好序的list对象alist？**

**import** random

alist = [1,2,3,4,5]

random.shuffle(alist)

print(alist)

**036、现有字典 d= {'a':24,'g':52,'i':12,'k':33}请按value值进行排序?**

sorted(d.items(),key=**lambda** x:x[1])

**037、字典推导式？**

d = {key:**value** **for** (key,**value**) **in** iterable}

**038、请反转字符串 "aStr"?**

print("aStr"[::-1])

**039、将字符串 "k:1 |k1:2|k2:3|k3:4"，处理成字典 {k:1,k1:2,...}**

str1 = "k:1|k1:2|k2:3|k3:4"

def **str2dict**(str1):

dict1 = {}

**for** iterms **in** str1.split('|'):

key,**value** = iterms.split(':'):

dict1[key] = **value**

**return** dict1

**040、请按alist中元素的age由小到大排序？**

alist = [{'name':'a','age':20},{'name':'b','age':30},{'name':'c','age':25}]

**def** **sort\_by\_age**(list1):

**return** sorted(alist,key=**lambda** x:x['age'],reverse=**True**)

**041、下面代码的输出结果将是什么？**

**list** = ['a','b','c','d','e']

**print**(**list**[10:])

代码将输出[],不会产生IndexError错误，就像所期望的那样，尝试用超出成员的个数的index来获取某个列表的成员。例如，尝试获取list[10]和之后的成员，会导致IndexError。然而，尝试获取列表的切片，开始的index超过了成员个数不会产生IndexError，而是仅仅返回一个空列表。这成为特别让人恶心的疑难杂症，因为运行的时候没有错误产生，导致Bug很难被追踪到。

**042、写一个列表生成式，产生一个公差为11的等差数列**

print([x\*11 **for** x **in** range(10)])

**043、给定两个列表，怎么找出他们相同的元素和不同的元素？**

list1 = [1,2,3]

list2 = [3,4,5]

set1 = set(list1)

set2 = set(list2)

print(set1 & set2)

print(set1 ^ set2)

**044、请写出一段python代码实现删除list里面的重复元素？**

l1 = ['b','c','d','c','a','a']

l2 = list(set(l1))

print(l2)

用list类的sort方法:

l1 = ['b','c','d','c','a','a']

l2 = list(set(l1))

l2.sort(key=l1.index)

print(l2)

也可以这样写:

l1 = ['b','c','d','c','a','a']

l2 = sorted(set(l1),key=l1.index)

print(l2)

也可以用遍历：

l1 = ['b','c','d','c','a','a']

l2 = []

**for** i **in** l1:

**if** **not** i **in** l2:

l2.append(i)

print(l2)

**045、给定两个list A，B ,请用找出A，B中相同与不同的元素**

A,B 中相同元素： print(set(A)&set(B))

A,B 中不同元素: print(set(A)^set(B))

**046、python新式类和经典类的区别？**

a. 在python里凡是继承了object的类，都是新式类

b. Python3里只有新式类

c. Python2里面继承object的是新式类，没有写父类的是经典类

d. 经典类目前在Python里基本没有应用

**047、python中内置的数据结构有几种？**

a. 整型 int、 长整型 long、浮点型 float、 复数 complex

b. 字符串 str、 列表list、 元祖tuple c. 字典 dict 、 集合 set

**049、反转一个整数，例如-123 --> -321**

**class** **Solution**(object):

**def** **reverse**(self,x):

**if** -10<x<10:

**return** x

str\_x = str(x)

**if** str\_x[0] !="-":

str\_x = str\_x[::-1]

x = int(str\_x)

**else**:

str\_x = str\_x[1:][::-1]

x = int(str\_x)

x = -x

**return** x **if** -2147483648<x<2147483647 **else** 0

**if** \_\_name\_\_ == '\_\_main\_\_':

s = Solution()

reverse\_int = s.reverse(-120)

print(reverse\_int)

**050、设计实现遍历目录与子目录，抓取.pyc文件**

第一种方法：

**import** os

**def** **getFiles**(dir,suffix):

res = []

**for** root,dirs,files **in** os.walk(dir):

**for** filename **in** files:

name,suf = os.path.splitext(filename)

**if** suf == suffix:

res.append(os.path.join(root,filename))

print(res)

getFiles("./",'.pyc')

第二种方法：

**import** os

**def** **pick**(obj):

**try**:

**if** obj.[-4:] == ".pyc":

print(obj)

**except**:

**return** **None**

**def** **scan\_path**(ph):

file\_list = os.listdir(ph)

**for** obj **in** file\_list:

**if** os.path.isfile(obj):

pick(obj)

**elif** os.path.isdir(obj):

scan\_path(obj)

**if** \_\_name\_\_=='\_\_main\_\_':

path = input('输入目录')

scan\_path(path)

**051、一行代码实现1-100之和**

count = sum(**range**(0,101))

print(count)

**052、对设计模式的理解，简述你了解的设计模式？**

设计模式是经过总结，优化的，对我们经常会碰到的一些编程问题的可重用解决方案。一个设计模式并不像一个类或一个库那样能够直接作用于我们的代码，反之，设计模式更为高级，它是一种必须在特定情形下实现的一种方法模板。 常见的是工厂模式和单例模式

**055、对装饰器的理解，并写出一个计时器记录方法执行性能的装饰器？**

装饰器本质上是一个python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。

import time

def timeit(func):

def wrapper():

start = time.clock()

func()

end = time.clock()

print('used:',end-start)

return wrapper

@timeit

def foo():

print('in foo()'foo())

**056、property是什么？**

Python property() 函数：

property() 函数的作用是在新式类中返回属性值。

语法

以下是 property() 方法的语法:

class property([fget[, fset[, fdel[, doc]]]])

参数

fget -- 获取属性值的函数

fset -- 设置属性值的函数

fdel -- 删除属性值函数

doc -- 属性描述信息

返回值

返回新式类属性。

1. 基本认识

property属性可以用来给属性添加约束，比如温度属性，我们不允许低于-273度；成绩属性，我们不允许0分以下等等。而且使用property属性，将来修改约束条件的时候也很方便，可以在代码的调用方式不变的情况下改变结果。

python中使用property属性有两种方法。使用@property装饰器和使用property()函数。

我们通过廖雪峰官方网站的实例来对此加深认识。

2. @property装饰器

@property装饰器就是负责把一个方法变成属性调用的。如下实例就可以通过s.score来获得成绩，并且对score赋值之前做出了数据检查。

3. property()函数

python中关于property()函数的介绍如下，在jupyter notebook中输入property??，即可查看用法。

**057、**[**python 魔法方法补充（\_\_setattr\_\_,\_\_getattr\_\_,\_\_getattribute\_\_）**](https://www.cnblogs.com/big-handsome-guy/p/8618078.html)

\_\_getattr\_\_方法：

拦截点号运算。对象.属性：如果找不到属性，就会用属性名作为字符串，调用该方法；如果继承树可以找到该属性，则不会调用该方法

\_\_setattr\_\_方法：  
会拦截所有属性的的赋值语句。如果定义了这个方法，self.arrt = value 就会变成self.\_\_setattr\_\_("attr", value).

这个需要注意。当在\_\_setattr\_\_方法内对属性进行赋值是，不可使用self.attr = value,因为他会再次调用self.\_\_setattr\_\_("attr", value),则会形成无穷递归循环，最后导致堆栈溢出异常。应该通过对属性字典做索引运算来赋值任何实例属性，也就是使用self.\_\_dict\_\_['name'] = value.

**058、一个字符串，让你倒序输出，例子 my name is >>> is name my？**

#python解决字符串倒序输出

def string\_reverse(m):

num=len(m)

a=[]

for i in range(num):

a.append(m[num-1-i])#从最后一位的元素开始往新list内添加元素

return a

**059、对于一个给定的字符串，逆序输出，这个任务对于python来说是一种很简单的操作，毕竟强大的列表和字符串处理的一些列函数足以应付这些问题 了，今天总结了一下python中对于字符串的逆序输出的几种常用的方法，一共总结出来了六种方法，个人认为比较全面了，如有其他方法欢迎前来补充？**

方法一：

直接使用字符串切片功能逆转字符串

方法二：

将字符串转换为列表使用reverse函数

方法三：

新建一个列表，从后往前添加元素

方法四：

借助于collections模块现成的方法extendleft

方法五：

递归实现

方法六：

借助基本的Swap操作，以中间为基准交换对称位置的字符

    下面是具体的实现：

#!usr/bin/env python

#encoding:utf-8

'''''

\_\_Author\_\_:沂水寒城

功能：输入一个字符串，倒叙输出

'''

import collections

def func1(one\_str):

'''

直接使用字符串切片功能逆转字符串

'''

return one\_str[::-1]

def func2(one\_str):

'''

将字符串转换为列表使用reverse函数

'''

one\_str\_list = list(one\_str)

one\_str\_list.reverse()

return ''.join(one\_str\_list)

def func3(one\_str):

'''

新建一个列表，从后往前添加元素

'''

one\_list=[]

for i in range(len(one\_str)-1,-1,-1):

one\_list.append(one\_str[i])

return ''.join(one\_list)

def func4(one\_str):

'''

借助于collections模块现成的方法extendleft

'''

deque1=collections.deque(one\_str)

deque2=collections.deque()

for one\_char in deque1:

deque2.extendleft(one\_char)

return ''.join(deque2)

def func5(one\_str):

'''

递归实现

'''

if len(one\_str)<=1:

return one\_str

return one\_str[-1]+func5(one\_str[:-1])

def func6(one\_str):

'''

借助基本的Swap操作，以中间为基准交换对称位置的字符

'''

one\_str\_list=list(one\_str)

if len(one\_str\_list)==0 or len(one\_str\_list)==1:

return one\_str\_list

i=0

length=len(one\_str\_list)

while i < length/2:

one\_str\_list[i], one\_str\_list[length-i-1]=one\_str\_list[length-i-1], one\_str\_list[i]

i+=1

return ''.join(one\_str\_list)

def main\_func(str\_list):

'''

主调用函数

'''

for one\_str in str\_list:

one\_list=[]

one=[]

one\_list.append(func1(one\_str))

one\_list.append(func2(one\_str))

one\_list.append(func3(one\_str))

one\_list.append(func4(one\_str))

one\_list.append(func5(one\_str))

one\_list.append(func6(one\_str))

print '字符串{0}逆序为：'.format(one\_str)

print one\_list

if \_\_name\_\_ == '\_\_main\_\_':

str\_list=['123456','abcdefg','zyxvuw','Together\_CZ']

main\_func(str\_list)

结果如下：

字符串123456逆序为：['654321', '654321', '654321', '654321', '654321', '654321']

字符串abcdefg逆序为：['gfedcba', 'gfedcba', 'gfedcba', 'gfedcba', 'gfedcba', 'gfedcba']

字符串zyxvuw逆序为：['wuvxyz', 'wuvxyz', 'wuvxyz', 'wuvxyz', 'wuvxyz', 'wuvxyz']

字符串Together\_CZ逆序为：['ZC\_rehtegoT', 'ZC\_rehtegoT', 'ZC\_rehtegoT', 'ZC\_rehtegoT', 'ZC\_rehtegoT', 'ZC\_rehtegoT']

**058、判断括号是否合法，类似于leetcode上的，[](){}，每个括号都要成对？**

python判断括号是否成对出现且顺序合法

第一种方法

def isBracketPair(a):

bracketAll = {")": "(", "]": "[", "}": "{"}

b = []

for i in a:

if i in bracketAll.values():

b.append(i)

elif len(b) > 0 and b[-1] == bracketAll.get(i):

b.pop()

else:

return "bad"

if len(b) == 0:

return "ok"

else:

return "bad"

第二种方法

class Solution(object):

def isValid(self, s):

"""

:type s: str

:rtype: bool

"""

# 判断是否是奇数或空字符

if len(s)<2 or len(s)%2!=0:

if s=='':

return True

else:

return False

count = 0

length = len(s)

# 将其中的(){}[] 都换掉，然后判断是否有剩余

while(count<length/2):

s = s.replace("{}","").replace("[]","").replace("()","")

count+=1

if len(s)>0:

return False

else:

return True

第三种方法

{}[]（) 是合法的括号。

   {}[} 是不合法的括号。

# 判断是否为合法的匹配括号。

class Solution():

def validBracket(self, nums):

dic2 = {"(":")","{":"}","[":"]"}

stack = []

for i in range(len(nums)):

if nums[i] in dic2:

stack.append(nums[i])

elif len(stack)!=0 and dic2[stack[-1]]==nums[i]:

stack.pop()

else:

return False

return stack==[]

if \_\_name\_\_ == "\_\_main\_\_":

s = Solution()

str2 = "(){}"

print(s.validBracket(str2))

**060、X是什么类型?**

X= (fo ri in ramg(10))

X是 generator类型

**061、请用一行代码 实现将1-N 的整数列表以3为单位分组？**

print ([[x for x in range(1,100)] [i:i+3] for i in range(0,len(list\_a),3)])

**062、Python中yield的用法？**

yield就是保存当前程序执行状态。你用for循环的时候，每次取一个元素的时候就会计算一次。用yield的函数叫generator,和iterator一样，它的好处是不用一次计算所有元素，而是用一次算一次，可以节省很多空间，generator每次计算需要上一次计算结果，所以用yield,否则一return，上次计算结果就没了

**063、进程总结？**

进程：程序运行在操作系统上的一个实例，就称之为进程。进程需要相应的系统资源：内存、时间片、pid。

创建进程： 首先要导入multiprocessing中的Process： 创建一个Process对象; 创建Process对象时，可以传递参数;

p = Process(target=XXX,args=(tuple,),kwargs={key:value})

target = XXX 指定的任务函数，不用加(),

args=(tuple,)kwargs={key:value}给任务函数传递的参数

使用start()启动进程 结束进程 给子进程指定函数传递参数Demo

**import** os

**from** mulitprocessing **import** Process

**import** time

**def** **pro\_func**(name,age,\*\*kwargs):

**for** i **in** range(5):

print("子进程正在运行中，name=%s,age=%d,pid=%d"%(name,age,os.getpid()))

print(kwargs)

time.sleep(0.2)

**if** \_\_name\_\_ =="\_\_main\_\_":

*#创建Process对象*

p = Process(target=pro\_func,args=('小明',18),kwargs={'m':20})

*#启动进程*

p.start()

time.sleep(1)

*#1秒钟之后，立刻结束子进程*

p.terminate()

p.join()

注意：进程间不共享全局变量 进程之间的通信-Queue

在初始化Queue()对象时（例如q=Queue(),若在括号中没有指定最大可接受的消息数量，获数量为负值时，那么就代表可接受的消息数量没有上限一直到内存尽头）

Queue.qsize():返回当前队列包含的消息数量；

Queue.empty():如果队列为空，返回True，反之False；

Queue.full():如果队列满了，返回True,反之False；

Queue.get([block[,timeout]]):获取队列中的一条消息，然后将其从队列中移除，block默认值为True。 如果block使用默认值，且没有设置timeout（单位秒),消息队列如果为空，此时程序将被阻塞（停在读中状态），直到消息队列读到消息为止，如果设置了timeout，则会等待timeout秒，若还没读取到任何消息，则抛出“Queue.Empty"异常；

Queue.get\_nowait()相当于Queue.get(False)；

Queue.put(item,[block[,timeout]]):将item消息写入队列，block默认值为True; 如果block使用默认值，且没有设置timeout（单位秒），消息队列如果已经没有空间可写入，此时程序将被阻塞（停在写入状态），直到从消息队列腾出空间为止，如果设置了timeout，则会等待timeout秒，若还没空间，则抛出”Queue.Full"异常 如果block值为False，消息队列如果没有空间可写入，则会立刻抛出"Queue.Full"异常;

Queue.put\_nowait(item):相当Queue.put(item,False) 进程间通信Demo:

from multiprocessing import Process.Queue

import os,time,random

#写数据进程执行的代码：

def write(q):

for value in ['A','B','C']:

print("Put %s to queue...",%value)

q.put(value)

time.sleep(random.random())

#读数据进程执行的代码

def read(q):

while True:

if not q.empty():

value = q.get(True)

print("Get %s from queue.",%value)

time.sleep(random.random())

else:

break

if \_\_name\_\_=='\_\_main\_\_':

#父进程创建Queue，并传给各个子进程

q = Queue()

pw = Process(target=write,args=(q,))

pr = Process(target=read,args=(q,))

#启动子进程pw ，写入：

pw.start()

#等待pw结束

pw.join()

#启动子进程pr，读取：

pr.start()

pr.join()

#pr 进程里是死循环，无法等待其结束，只能强行终止:

print('')

print('所有数据都写入并且读完')

进程池Pool

#coding:utf-8

from multiprocessing import Pool

import os,time,random

def worker(msg):

t\_start = time.time()

print("%s 开始执行，进程号为%d"%(msg,os.getpid()))

# random.random()随机生成0-1之间的浮点数

time.sleep(random.random()\*2)

t\_stop = time.time()

print(msg,"执行完毕，耗时%0.2f”%（t\_stop-t\_start))

po = Pool(3)#定义一个进程池，最大进程数3

for i in range(0,10):

po.apply\_async(worker,(i,))

print("---start----")

po.close()

po.join()

print("----end----")

进程池中使用Queue 如果要使用Pool创建进程，就需要使用multiprocessing.Manager()中的Queue(),而不是multiprocessing.Queue(),否则会得到如下的错误信息： RuntimeError： Queue objects should only be shared between processs through inheritance

from multiprocessing import Manager,Pool

import os,**time**,random

def reader(**q**):

**print**("reader 启动(%s),父进程为（%s)"%(os.getpid(),os.getpid()))

**for** i in range(q.qsize()):

**print**("reader 从Queue获取到消息:%s"%q.get(True))

def writer(**q**):

**print**("writer 启动（%s),父进程为(%s)"%(os.getpid(),os.getpid()))

**for** i ini "itcast":

q.put(i)

**if** \_\_name\_\_ == "\_\_main\_\_":

**print**("(%s)start"%os.getpid())

**q** = Manager().Queue()*#使用Manager中的Queue*

po = Pool()

po.apply\_async(wrtier,(**q**,))

time.sleep(1)

po.apply\_async(reader,(**q**,))

po.close()

po.join()

**print**("(%s)End"%os.getpid())

**网络编程**

**000、网络编程知识点？**

多进程

关键字

开启进程的俩种方式

进程之间的内存空间是隔离的

进程中的join方法使用

进程对象的其它属性和方法

守护进程的使用

进程的互斥锁

进程实现队列

生产者和消费者模型

多线程

使用多线程实现tcp并发

开启线程的俩种方式

进程和线程的俩种区别

区别一：启动数据快

区别二：线程间资源共享，进程间资源独立

守护线程的使用

线程的互斥锁

线程的GIL锁（解释器锁）

paramiko模块实现ssh登录

死锁和递归锁RLock

信号量Semaphore

Event事件

定时器

线程的queue

进程池ProcessPoolExecutor

线程池ThreadPoolExecutor

协程

yield实现协程

greenley模块

gevent模块

**001、实现线程、进程的模块？**

Python的标准库提供了两个模块：thread和threading，thread是低级模块，threading是高级模块，对thread进行了封装。绝大多数情况下，我们只需要使用threading这个高级模块。

启动一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行：

使用Thread类，可以有多种方法创建线程：

1、创建Thread类的实例，传递一个函数；

2、创建Thread类的实例，传递一个可调用的类实例；

3、派生Thread类的子类，并创建子类的实例；

一般的，我们会采用第一种或者第三种方法。

前面也说了python多线程的弊端和GIL的内容，适合IO密集型，而如果解决计算密集型时候的多线程呢？那就是多进程。

multiprocessing模块

1、python中多线程无法利用多核优势，如果想要充分地使用多核cpu的资源（os.cpu\_count()），在python中大部分情况需要使用多进程，python提供了multiprocessing。

2、multiprocessing并非是python的一个模块，而是python中多进程管理的一个包；

3、multiprocessing模块用来开启子进程，并在子进程中执行我们定制的任务（比如函数），该模块与多线程模块threading的编程接口类似。

4、multiprocessing模块的功能众多：支持子进程，通信和共享数据，执行不同形式的同步，提供了process、Queue、Lock等组件。

需要再次强调的一点是：与线程不同，进程没有任何共享状态，进程修改的数据，改动仅限与该进程内。

**009.简述什么是进程、线程、协程？**

1、进程是系统进行资源分配和调度的一个独立单位，一个程序至少有一个进程,一个进程至少有一个线程。

2、线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一3、组寄存器和栈),但它可与同属一个进程的其他的线程共享进程所拥有的全部资源。线程不能够独立执行，必须依存在进程中

4、协程是一种比线程更加轻量级的存在，一个线程也可以拥有多个协程。

**010.简述进程、线程、协程三者的区别？**

1、进程切换需要的资源很大，效率相对低；

2、线程切换需要的资源一般，效率比进程高；

3、协程切换任务资源很小，三者中效率最高；

4、多进程、多线程根据CPU核数不一样可能是并行，但是协程是在一个线程中所以是并发；

**011.简述进程、线程、协程适用于那种应用场景类型？**

计算密集型：用进程；

IO密集型：用线程或协程

**012.协程为何比线程还快？**

高并发+高扩展性+低成本：一个CPU支持上万的协程都不是问题。所以很适合用于高并发处理协程能保留上一次调用时的状态，不管是进程还是线程，每次阻塞、切换都需要陷入系统调用，使用线程时需要非常小心地处理同步问题，而协程完全不存在这个问题。

**017、什么是线程安全？**

Python线程安全：就是对于多线程同时操作是是安全的而不会发生写冲突,比如python的Queue；

Python非线程安全：就是多线成同时操作时会发生写冲突,比如python的list,set,dict ；

**018、标准库有哪些线程安全的队列？**

Python Python Queue模块有三种队列:

1、FIFO队列先进先出.(线程安全)；

2、LifoQueue类似于堆,即先进后出(线程安全)；

3、PriorityQueue优先级队列,级别越低,越先出来(线程安全)；

对应的构造函数:

1、class Queue.Queue(maxsize) FIFO；

2、class Queue.LifoQueue(maxsize) LIFO；

3、class Queue.PriorityQueue(maxsize) 优先级队列；

**019、什么是GIL？**

GIL是python中的全局解释器锁，是不可控的，同一个进程中，假如有多个线程在运行，那么其中一个线程在运行的时候就会霸占GIL锁，就使得其他线程无法运行，等该线程运行结束以后，其他线程才能运行。如果线程中遇到耗时操作(I/O密集型任务)，则解释器锁会解开，使得其他线程运行，所以说在多线程中，线程的运行仍是有先后顺序的，并不是同时进行。

**020.什么时候释放GIL锁？**

1、时间片耗尽(cpu时间)；

2、任务遇到I/O等待时；

3、执行任务结束；

4、执行到字节码阈值时；

**021、互斥锁与GIL的区别？**

1、互斥锁是在多线程的情况下，确保当前线程执行完成后，再执行下一个任务，当前任务没有结束，下个任务会阻塞。

2、GIL是保证同一时间只有1个线程在执行，但是该线程让出GIL的时，有可能并没完成该线程的任务，该线程的任务分多少次执行完成这个会安装GIL的默认策略。

**022.python高并发解决方案有哪些？**

1、gevent 代码看起来好看一些,但是维护比较差,patch没有规律,而且里面封装了C,对python3的支持最差.

2、twisted 稳定性是最好的,但是需要较长时间的学习.对python3的支持较差.

3、tornado 兼容性最好.但是过于简单了,功能不强,另外没有python数据库适配器能和tornado无缝对接,因此调用数据库很麻烦,而且只支持web.

**064、简述进程，线程，协程的区别以及应用场景，项目是否用？谈谈你对多进程，多线程，以及协程的理解？**

这个问题被问的概念相当之大。

1.进程是操作系统资源分配的最小单位，拥有独立的资源和地址空间

2.线程是CPU调度的单位

3.统一进程中的线程是资源共享的。

4.协程是用户级别的，程序之间的切换由用户自行处理，节省了CPU的调度时间。

区别：

1、进程是资源分配的单位；

2、线程是操作系统调度的单位；

3、进程切换需要的资源很大，效率很低；

4、线程切换需要的资源一般，效率一般(在不考虑GIL的情况下协程切换任务资源很小，效率高；多进程，多线程根据cpu内核数的运行不同，不一定是并行的，但是协程是在一个线程中，所以是并发。)

应用场景

1、协程：当程序中存在大量不需要cpu的操作时，适用协程；

2、计算密集型，用进程。

3、IO密集型，用线程。

进程：一个运行的程序（代码）就是一个进程，没有运行的代码叫程序，进程是系统资源分配的最小单位，进程拥有自己独立的内存空间，所有进程间数据不共享，开销大。

线程: cpu调度执行的最小单位，也叫执行路径，不能独立存在，依赖进程存在，一个进程至少有一个线程，叫主线程，而多个线程共享内存（数据共享，共享全局变量),从而极大地提高了程序的运行效率。

协程: 是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操中栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。简单点说协程是进程和线程的升级版,进程和线程都面临着内核态和用户态的切换问题而耗费许多切换时间,而协程就是用户自己控制切换的时机,不再需要陷入系统的内核态.Python里最常见的yield就是协程的思想!

**065.曾经在哪里使用过: 线程、进程、协程?**

1.在写高并发的服务端代码时。2.在写高性能爬虫的时候。

**066、协程？**

协程是一种用户态的轻量级线程，又称微线程。

协程拥有自己的寄存器上下文和栈，调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

高并发+高扩展性+低成本：一个CPU支持上万的协程都不是问题。所以很适合用于高并发处理。

所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch （切换到另一个线程）。

原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序是不可以被打乱，或者切割掉只执行部分。视作整体是原子性的核心。

**1、协程的优点：**

1、无需线程上下文切换的开销造成的资源的浪费。

2、无需原子操作锁定及同步的开销。

3、方便切换控制流，简化编程模型。

高并发及高扩展性加低成本：一个CPU支持上万的协程都可以，于高并发处理。

**2、协程的缺点：**

1. 1、无法利用多核资源，协程的本质是单个线程，它不能同时将单个CPU 的多个核用上（不能同时使用多核CPU）,协程需要和进程配合才能运行在多CPU上.当然我们日常所编写的绝大部分应用都没有这个必要，除非是cpu密集型应用。

2、协程需要与进程配合才能运行在多CPU上。

3、程序一旦阻塞，会阻塞整个代码段（进行阻塞（Blocking）操作（如IO时）会阻塞掉整个程序）。

**3、协程实现方式：**

1) yield；  
2) Greenlet；  
3) Gevent；

**4、yield实现的协程：**

传统的生产者－消费者模型是一个线程生成消息，一个线程取得消息，能过锁机制控制队列和等待，但一不小心就有可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换加生产者继续生产，效率较高。

整个流程是由一个线程执行，producer和consumer协作完成任务，所以称为协程，而不是线程中的抢占式多任务。

基于协程的定义，刚才使用yield实现的协程并不算合格的协程。

yield 是类似于return的关键字，不同的是，return关键字是直接结束了函数的执行。而yield 可以通过调用 send(), \_\_next\_\_() 让函数继续从yield关键字下一条语句继续执行。  
send()函数的用法 是可以传参的，send(param) 。即传一个参数给yield生成器。  
\_\_next\_\_() 函数和send()不同的是它不传参。

**5、由greenlet模块实现的协程：**

greenlet机制的主要思想是：生成器函数或者协程函数中的yield语句挂起函数的执行，直到稍后使用next()或send()操作进行恢复主止。可以使用一个调度器循环在一组生成器函数之间协作多个任务。greenlet是python中实现协程的一个模块。

greenlet不是一种真正的并发机制，而是在同一线程内，在不同函数的执行代码块之间切换  
当出现阻塞时，就显式切换到没有被阻塞的代码段执行，直到另一端代码再显示的切换到本段代码时，这段代码才会继续执行。

**6、基于greenlet框架，gevent模块实现协程：**

python通过yield提供了对协程的基本支持，但是不完全。第三方的gevent模块提供了协程支持。

gevent是第三方库，通过greenlet实现协程。  
当一个greenlet遇到IO操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。由于IO操作非常耗时，经常使程序处于等待状态，有了gevent自动切换协程，就保证总有greenlet在运行，而不是等待IO。

gevent.spawn()方法spawn一些任务，然后通过gevent.joinall将任务加入协程执行队列中等待执行。

gevent的核心greenlet，并试用 epoll机制使线程实现自动切换，并保证始终有greenlet在运行，而不是等待io。  
gevent.spawn()方法会创建并运行一个新的greenlet协程对象。  
gevent.joinall()方法的参数是一个协程对象列表，等待所有的协程都执行完成后退出

**网络编程之HTTP系列**

**001.HTTP是什么？**

1、HTTP协议是Hyper Text Transfer Protocol（超文本传输协议）的缩写,是用于从万维网（WWW:World Wide Web ）服务器传输超文本到本地浏览器的传送协议。

2、HTTP协议是一个基于TCP/IP通信协议来传递数据（HTML 文件, 图片文件, 查询结果等）。

3、HTTP协议是一个属于应用层的面向对象的协议。

4、HTTP协议工作于客户端和服务端架构上。浏览器作为HTTP客户端通过URL向HTTP服务器发送请求，服务器根据接收到的请求后，向客户端发送响应信息。

**基于TCP/IP：**

双方建立通信的顺序,以及Web页面显示需要 处理的步骤,等等。像这样把与互联网相关联的协议集合起来总称为 TCP/IP。而HTTP协议是基于TCP/IP协议之上的应用层协议。

**基于请求－响应模式：**

HTTP协议规定,请求从客户端发出,最后服务器响应该请求并返回

**无状态保存：**

HTTP是一种无状态协议。HTTP协议不对请求和响应之间的通信状态进行保存，不做持久化处理。这是为了更快地处理大量事务,确保协议的可伸缩性,而特意把HTTP协议设计成 如此简单的。

可是,随着Web的不断发展,因无状态而导致业务处理变得棘手的情况增多了。比如,用户登录到一家购物网站,即使他跳转到该站的 其他页面后,也需要能继续保持登录状态。

针对这个实例,网站为了能够掌握是谁送出的请求,需要保存用户的状态。HTTP/1.1虽然是无状态协议,但为了实现期望的保持状态功能, 于是引入了Cookie技术。有了Cookie再用HTTP协议通信,就可以管理状态了。

**002.HTTP请求报文与响应报文格式？**

**请求报文包含三部分：**

1、请求行(包含请求方法、URL、HTTP版本)；

2、请求首部字段；

3、空格（\r\n）；

4、请求内容实体；

* **响应报文包含三部分：**

1、状态行(包含状态码、状态码的原因短语、HTTP版本)；

2、响应首部字段；

3、空格（\r\n）；

4、响应内容实体 ；

**003.HTTP常见首部字段有哪些？**

1. **通用首部字段（请求报文与响应报文都会使用的首部字段）**

| **字段** | **字段描述** |
| --- | --- |
| Date | 创建报文时间 |
| Connection | 连接的管理 |
| Cache-Control | 缓存的控制 |
| Transfer-Encoding | 报文主体的传输编码方式 |

1. **请求首部字段（请求报文会使用的首部字段）**

| **字段** | **字段描述** |
| --- | --- |
| Host | 请求资源所在服务器 |
| Accept | 可处理的媒体类型 |
| Accept-Charset | 可接收的字符集 |
| Accept-Encoding | 可接受的内容编码 |
| Accept-Language | 可接受的自然语言 |

1. **响应首部字段（响应报文会使用的首部字段）**

| **字段** | **字段描述** |
| --- | --- |
| Accept-Ranges | 可接受的字节范围 |
| Location | 令客户端重新定向到的URI |
| Server | HTTP服务器的安装信息 |

1. **实体首部字段（请求报文与响应报文的实体部分使用的首部字段）**

| **字段** | **字段描述** |
| --- | --- |
| Allow | 资源可支持的HTTP方法 |
| Content-Type | 实体主类的类型 |
| Content-Encoding | 实体主体适用的编码方式 |
| Content-Language | 实体主体的自然语言 |
| Content-Length | 实体主体的的字节数 |
| Content-Range | 实体主体的位置范围，一般用于发出部分请求时使用 |

**004.HTTP与HTTPS的区别？**

1、HTTP协议传输的数据都是明文、未加密的。因此使用HTTP协议传输隐私信息非常不安全。

2、HTTPS协议是由HTTP+SSL构建的网络协议，可进行加密传输、身份认证，要比HTTP协议安全。

| **不同点** | **HTTP** | **HTTPS** |
| --- | --- | --- |
| 安全证书 | 不需要 | 需要到ca申请证书（免费/付费可选） |
| 是否加密 | 不加密，信息是明文传输 | 具有安全性的ssl加密传输协议 |
| 验证数据 | 不验证数据，可能遭到伪装 无法验证报文完整性，可能被篡改 | 验证数据且对数据完整性保护 |
| 连接方式 | 无状态连接 | HTTPS协议是由HTTP+SSL协议构建的 可进行加密传输、身份认证的网络协议，比HTTP协议安全。 |
| 默认端口 | 80 | 443 |
| 安全性 | 不安全 | 安全 |

**005.列举HTTP请求中常见的请求方式？**

| **方法名称** | **定义** |
| --- | --- |
| GET | 向特定的路径资源发出请求 |
| POST | 向指定路径资源提交数据进行处理请求（一般用于提交表单或者上传文件） |
| PUT | 从客户端向服务器传送数据更新指定的资源 |
| PATCH | 从客户端向服务器传送数据更新部分指定的资源 |
| DELETE | 请求服务器删除指定的资源 |
| HEAD | 向服务器索要与GET一样的请求，但是不返回返回体。 这个方法可以在不必传输整个响应内容的情况下，获取包含在响应消息头中的元信息 |
| OPTIONS | 查询相应URL支持的HTTP方法 |
| TRACE | 返回服务器收到的请求，主要用于测试或诊断 |
| CONNECT | HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务 |

**006.GET方法与POST方法的区别？**

| **不同点** | **GET** | **POST** |  |
| --- | --- | --- | --- |
| 本质 | 产生1个TCP数据包，只跑1次 浏览器会把HTTP header和data一并发送出去，服务器响应200（返回数据） | 产生2个TCP数据包，来回各1次（共2次） 浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据） |  |
| 请求形式 | 参数通过URL传递 | 把数据放在Request的body中 |  |
| 安全性 | 因为URL是可见的，可能会泄露私密信息， 如账号密码等，所以不安全 | 数据在request中，用户不可见，较get安全性较高 |  |
| 传输数据量 | 传输的数据量小，因为受URL长度最多是1024字节，但效率较高； | 可以传输大量数据，所以上传文件时只能用Post方式； |  |
| 支持编码 | 只能进行url编码 | 支持多种编码 |  |
| 字符格式 | 只能支持ASCII字符，向服务器传的中文字符可能会乱码。 | 支持标准字符集，可以正确传递中文字符。 |  |
| 浏览器处理 | 请求会被浏览器主动cache，请求参数会被完整保留在浏览器历史记录里 | 浏览器不会主动cache，参数不会被保留，除非手动设置 |  |
| 业务应用 | 常用在从服务器上获取数据 | 常用在向服务器发送数据 |  |

**GET只需要跑一趟就把信息送到了，而POST得跑两趟。第一趟，先去和服务器打个招呼“嗨，我等下要送一消息来，你们打开门迎接我”，然后再回头把数据送过去。**因为POST需要两步，时间上消耗的要多一点。所以看起来GET比POST更有效,但事实上不是，在网络环境好的情况下，发一次包的时间和发两次包的时间差别基本可以无视。而在网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点。而且并非所有浏览器都会在POST中发送两次包，Firefox浏览器就只发送一次。

**007.常见的HTTP相应状态码？**

| **状态码** | **描述** |
| --- | --- |
| 1xx | 指示信息--表示请求已接收，继续处理 |
| 2xx | 成功--表示请求已被成功接收、理解、接受 |
| 3xx | 重定向--要完成请求必须进行更进一步的操作 |
| 4xx | 客户端错误--请求有语法错误或请求无法实现 |
| 5xx | 服务器端错误--服务器未能实现合法的请求 |

| **状态码** | **描述** |
| --- | --- |
| 200 | 请求被正常处理 |
| 204 | 请求被受理但没有资源可以返回 |
| 206 | 客户端只是请求资源的一部分，服务器只对请求的部分资源执行GET方法， 相应报文中通过Content-Range指定范围的资源。 |
| 301 | 永久性重定向 |
| 302 | 临时重定向 |
| 303 | 与302状态码有相似功能，只是它希望客户端在请求一个URI的时候， 能通过GET方法重定向到另一个URI上 |
| 304 | 发送附带条件的请求时，条件不满足时返回，与重定向无关 |
| 307 | 临时重定向，与302类似，只是强制要求使用POST方法 |
| 400 | 请求报文语法有误，服务器无法识别 |
| 401 | 请求需要认证 |
| 403 | 请求的对应资源禁止被访问 |
| 404 | 服务器无法找到对应资源 |
| 500 | 服务器内部错误 |
| 503 | 服务器正忙 |

**008.如何对HTTP进行优化？**

**1. TCP复用：**

TCP连接复用是将多个客户端的HTTP请求复用到一个服务器端的TCP连接上，HTTP复用是指一个客户端的多个HTTP请求通过一个TCP连接进行处理；

**2. 内容缓存：**

将经常用到的内容进行缓存到浏览器中，那么客户端就可以直接在内存中获取响应的数据；

**3. 压缩：**

将文本数据进行压缩，减少带宽；

**4. SSL加速：**

使用SSL协议对HTTP协议进行加密，在通道内加密并加速；

**009.TCP与UDP的区别？**

**UDP协议和TCP协议都是传输层协议。**

  TCP（Transmission ControlProtocol，传输控制协议）提供的是面向连接，可靠的字节流服务。客户和服务器交换数据前，必须现在双方之间建立一个TCP连接，之后才能传输数据。并提供超时重发、丢弃重复数据，检验数据，流量控制等功能，保证数据能完整地从一端传到另一端。简单说就是必须要建立连接后才能传输数据，确保传输完整性，类比现实当中的打电话。

  UDP（User Data Protocol，用户数据报协议）是一个简单的面向数据报的运输层协议。它不提供可靠性，只是把应用程序传给IP层的数据报发送出去，但是不能保证它们能到达目的地。由于UDP在传输数据报前不用再客户和服务器之间建立一个连接，且没有超时重发等机制，所以传输速度很快。简单说就是单向把程序中的信息发送了，但也不知道对方收到没有，类比现实当中的寄信。

| **类型** | **不同点描述** |
| --- | --- |
| TCP | 面向连接，可靠的，速度慢，效率低。 |
| UDP | 无连接、不可靠、速度快、效率高。 |

**010.请介绍TCP的3次握手和4次挥手流程？**

SYN:请求字段；ACK:应答字段；



**==建立双工通信，确保双方都能收到对方的信息，所以需要3次握手==**

第1次握手：建立连接时，客户端发送syn包（syn=x）到服务器，并进入同步已发送状态，等待服务器确认；SYN：同步序列编号。

例如：客户端SYN=1; 客户端向服务器发送建立通信请求把客户端SYN=1的包发到服务器 客户端：我发了！

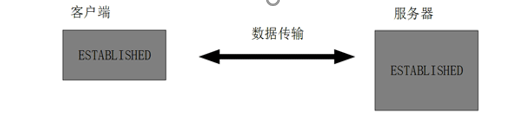
第2次握手：服务器收到syn包，必须确认客户的SYN（ack=x+1），同时自己也发送一个SYN包（syn=y），即SYN+ACK包，此时服务器进入同步已接受状态；

例如：服务端ACK=客户端SYN+1=2,服务端SYN=10；意思是服务端已经收到客户端请求，并在客户端SYN请求值上加1作为ACK值返回给客户端，告知客户端自己已收到，同时发送服务端自己的SYN值给客户端。（注意此处会同时返回SYN和ACK包给客户端） 服务器：你发来吧，我收到你的SYN了！

第3次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=y+1），此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。

例如：客户端AC包=服务端SYN+1=11；客户端在收到服务器的SYN包，并在服务器SYN请求值上加1作为ACK返回给服务端，告知客服端已收到服务端的SYN包，完成第三次握手。 客户端：我收到你的SYN了！

数据传输.....



**==全双工关闭需要客户端和服务器发送和接受都关闭，但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，只能先回复一个ACK报文，所以需要4次挥手。==**

第1次挥手：客户端进程发出连接释放报文，并且停止发送数据。此时，客户端进入FIN-WAIT-1（终止等待1）状态。

客户端：我不发了！停止发送，等待你的确认

第2次挥手： 服务器收到连接释放报文，发出确认报文。服务端就进入了CLOSE-WAIT（关闭等待）状态。TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。

服务器：我知道了！进入等待程序关闭状态

客户端收到服务器的确认请求后，此时，客户端就进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。

客户端：好，我知道你已经收到了！继续等待你的关闭确认

第3次挥手：服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为seq=w，此时，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。

服务器：我不发了！关闭发送，等待你的确认

第4次挥手： 客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是seq=u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。

客户端：我已经收到你的确认了，关闭接收

服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

服务器：我已经收到你的确认了，关闭发送

**011.为什么TCP连接的时候是3次握手，关闭的时候却是4次握手？**

因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。

但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。

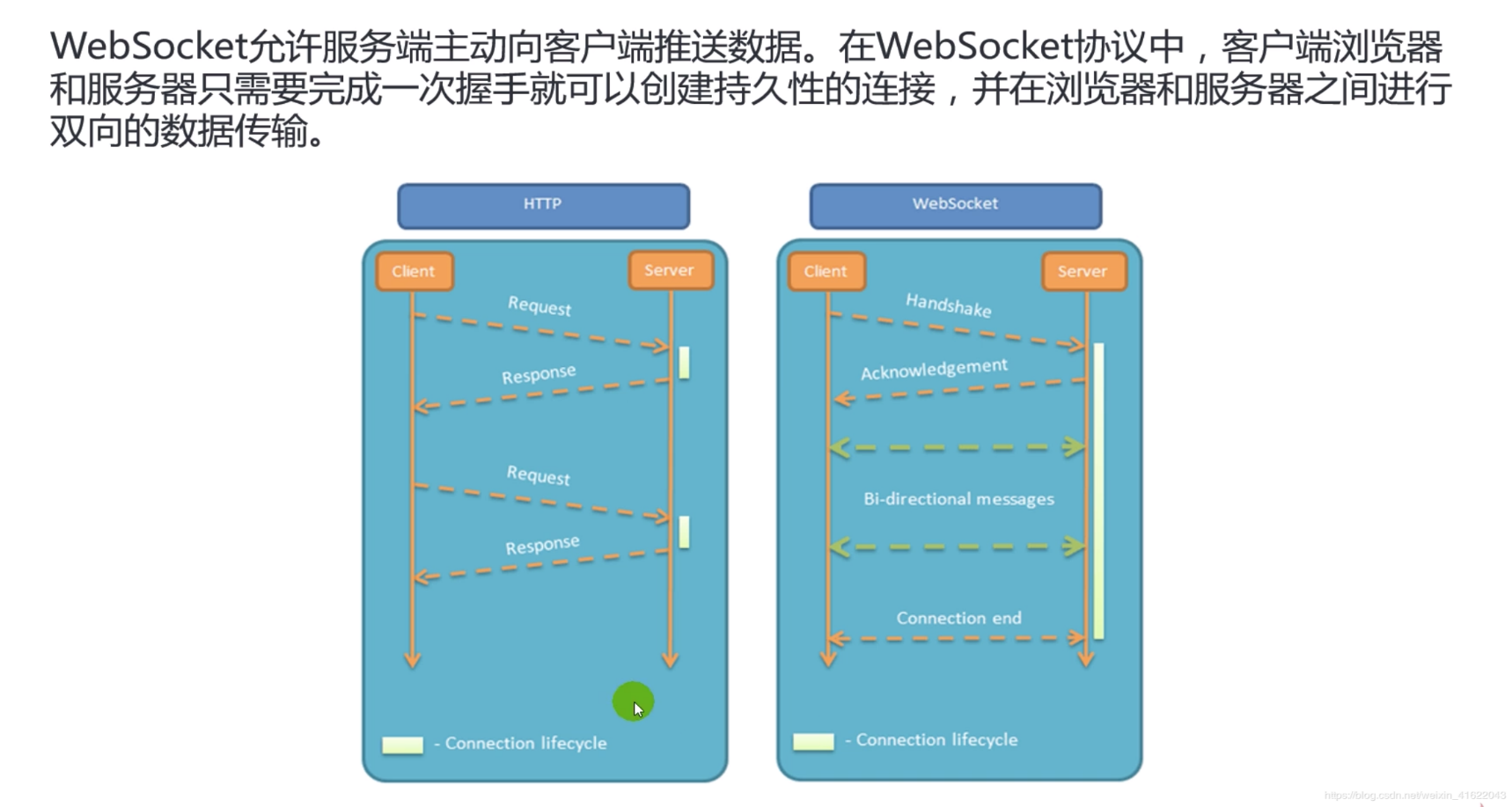
只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

**012.TCP为什么会分包和粘包？**

1、TCP是以段（Segment）为单位发送数据的,建立TCP链接后,有一个最大消息长度（MSS）.如果应用层数据包超过MSS,就会把应用层数据包拆分,分成两个段来发送.这个时候接收端的应用层就要拼接这两个TCP包,才能正确处理数据.

2、有时候,TCP为了提高网络的利用率,会使用一个叫做Nagle的算法.该算法是指,发送端即使有要发送的数据,如果很少的话,会延迟发送.如果应用层给TCP传送数据很快的话,就会把两个应用层数据包“粘”在一起,TCP最后只发一个TCP数据包给接收端.

**013.HTTP和websocket的区别？**



* ==**最大的区别就是HTTP只能由客户端推送信息给被动的服务端，而websocket既可以让客户端发送消息给服务端，也可以让服务端主动推送消息到客户端，实现双工通信**==
* **http协议是用在应用层的协议，他是基于tcp协议的，http协议建立链接也必须要有三次握手才能发送信息。**

  http链接分为短链接，长链接，短链接是每次请求都要三次握手才能发送自己的信息。即每一个request对应一个response。长链接是在一定的期限内保持链接。保持TCP连接不断开。客户端与服务器通信，必须要由客户端发起然后服务器返回结果。客户端是主动的，服务器是被动的。

* **WebSocket是为解决客户端与服务端实时通信。浏览器和服务器只需要做1个握手的动作，在建立连接之后，双方可以在任意时刻，相互推送信息。同时，服务器与客户端之间交换的头信息很小。**

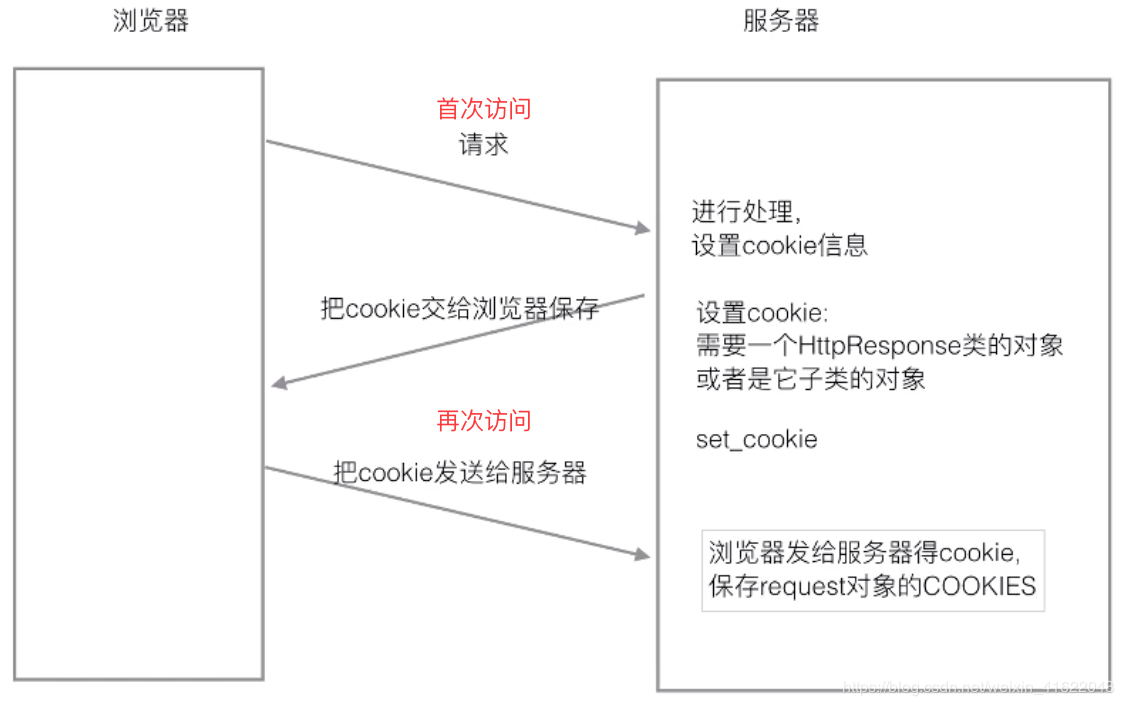
  建立了WebSocket之后服务器不必在浏览器发送request请求之后才能发送信息到浏览器。这时的服务器已有主动权想什么时候发就可以发送信息到服务器。而且信息当中不必在带有head的部分信息了与http的长链接通信来说，这种方式，不仅能降低服务器的压力。而且信息当中也减少了部分多余的信息。

**014.HTTP的长连接与websocket的持久连接的区别**

**1、HTTP1.1的连接默认使用长连接（persistent connection）**：HTTP 1.1默认进行持久连接。在1次 TCP 连接中可以完成多个 HTTP 请求，但是对每个请求仍然要单独发 header，Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Nginx）中设定这个时间。这种长连接是一种“伪链接”；

**2、WebSocket的持久连接**：WebSocket 是一个持久化的协议，只需建立1次Request/Response消息对，之后都是TCP连接，==避免了需要多次建立Request/Response消息对而产生的冗余头部信息。

**015.什么是cookie？其特点和应用场景有哪些？其如何获取、设置？**



* **cookie是一种会话跟踪技术，由服务器生成，然后通过响应发送给客户端的一个键值对。**

特点：

1、cookie是一个标准的python字典，以键值对方式进行存储，键值都是字符串；

2、通过浏览器访问一个网站时，浏览器会将存储该网站相关的所有cookie信息发送给该网站的服务器，request.COOKIES；

3、cookie是基于域名安全的；

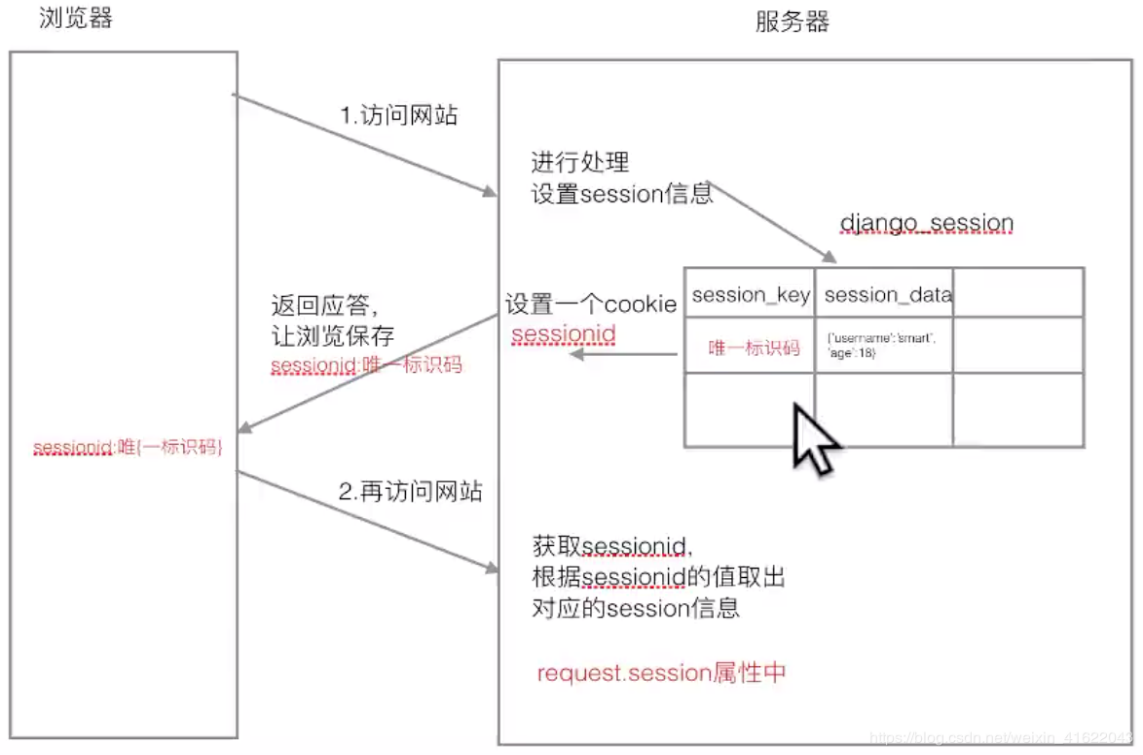
4、cookie是有过期时间的，如果不指定，默认关闭浏览器后cookie就会过期；

5、单个Cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个Cookie。

* **应用场景：记住用户名和密码，安全性要求不高**

response.set\_cookie("is\_login",True)  # 获取  
request.COOKIES.get("is\_login")  # 设置

**016.什么是session？其特点和应用场景有哪些？其如何设置、获取、清空？**



* **Session是服务器端技术，它保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是 Session。客户端浏览器再次访问时只需要从该 Session 中查找该客户的状态就可以了。**

特点：

1、它是以已键值对进行存储；

2、它依赖cookie，唯一的标识码session ID保存在cookie中；

3、它也是有过期时间，如果不指定，默认两周就会过期；

4、它是用base64加密；

5、应用场景：涉及到安全性要求比较高的数据，银行卡账户、密码；

request.session["is\_login"] = True  # 设置  
is\_login = request.session.get("is\_login")  # 获取  
request.session.flush()  # 清空

**017.session和cookie的相同点和区别？**

* **相同点**

||cookie/session| ---|---|---

跟踪技术|都是踪浏览器用户身份的绘画方式|

生成地点|都在服务器|

存储形式|都是键值对|

过期时间|都可自定义|

* **不同点**

||cookie|session| ---|---|---

存放地点|浏览器|服务器|

安全性|不安全，他人可通过分析存放在本地的cookie并进行Cookie欺骗|比较安全，因数据在服务器中，且用base64加密；

依赖性|无|依赖cookie，唯一的标识码session ID保存在cookie中 过期时间|默认关闭浏览器后就会过期|默认两周就会过期；

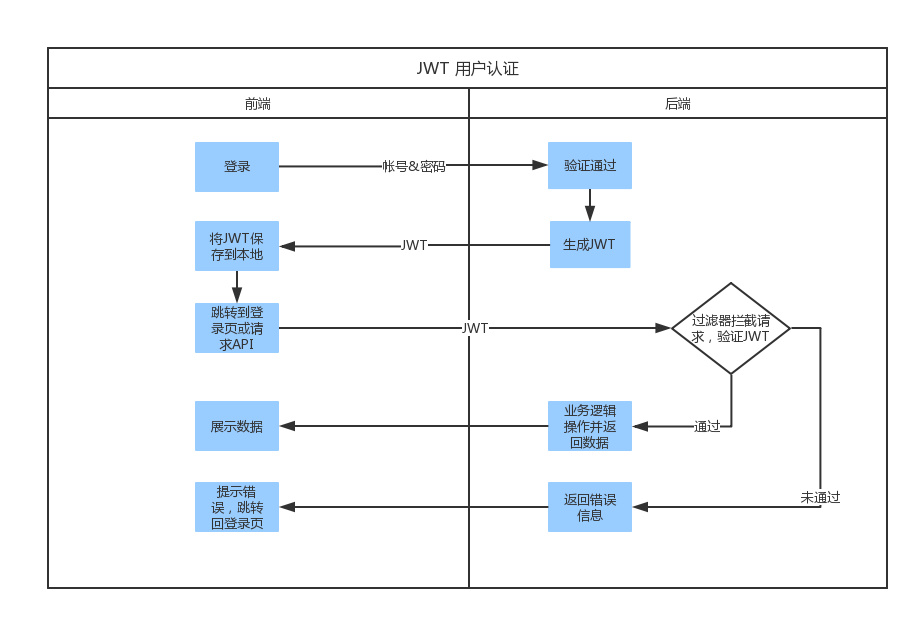
**018.什么是JWT，有什么特点？**

JWT 全称 JSON Web Tokens，它定义了一种以JSON 对象形式的安全通信方法。它由头部、负载和签名3部分组成。它具有2个特点：

简洁：可以通过URL, POST 参数或者在 HTTP header 发送，因为数据量小，传输速度快；

自包含：负载中包含了所有用户所需要的信息，避免了多次查询数据库。

**019.简述JWT的工作原理？**



1、客户端通过Web表单将正确的用户名和密码发送到服务端的接口。这一过程一般是POST请求。建议的方式是通过SSL加密的传输（https协议），从而避免敏感信息被嗅探。

2、服务端核对用户名和密码成功后，将用户的id等其他信息作为JWT Payload（负载），将其与头部分别进行Base64编码拼接后签名，形成一个JWT。形成的JWT就是一个形同lll.zzz.xxx的字符串，并设置有效时间。

3、服务端将JWT字符串作为登录成功的返回结果返回给客户端。

4、客户端将返回的JWT以cookie的形式保存在浏览器上，并设置cookie的有效时间（建议客户端cookie和服务端JWT的有效时间设置为一致），用户登出时客户端需删除cookie。

5、客户端在每次请求时将JWT放入HTTP Header中的Authorization位。(解决XSS和XSRF问题)；

6、服务端对收到的JWT进行解密和校验，如检查签名是否正确、Token是否过期、Token的接收方是否是自己等。

7、验证通过后服务端使用JWT中包含的用户信息进行其他逻辑操作，返回相应结果，否则返回401。

**020.传统Session方式存储ID和JWT的区别？**

1、Session是保存在服务端的，而JWT是保存在客户端的。

2、Session方式存储用户id的最大弊病在于Session是存储在服务器端的，所以需要占用大量服务器内存，对于较大型应用而言可能还要保存许多的状态。一般而言，大型应用还需要借助一些KV数据库和缓存机制来实现Session的存储。

JWT方式将用户状态分散到了客户端中，可以减少服务器查询数据库的次数和减轻服务端的内存压力。

除了用户id之外，还可以存储其他的和用户相关的信息，例如该用户是否是管理员、用户所在的分组等。虽说JWT方式让服务器有一些计算压力（例如加密、编码和解码），但是这些压力相比磁盘存储而言可能就不算什么了。具体是否采用，需要在不同场景下用数据说话。

**065、python多线程与多进程的区别？**

在UNIX平台上，当某个进程终结之后，该进程需要被其父进程调用wait，否则进程成为僵尸进程(Zombie)。所以，有必要对每个Process对象调用join()方法 (实际上等同于wait)。对于多线程来说，由于只有一个进程，所以不存在此必要性。

多进程应该避免共享资源。

在多线程中，我们可以比较容易地共享资源，比如使用全局变量或者传递参数。在多进程情况下，由于每个进程有自己独立的内存空间，以上方法并不合适。此时我们可以通过共享内存和Manager的方法来共享资源。但这样做提高了程序的复杂度，并因为同步的需要而降低了程序的效率。

**065、Python异常使用场景有那些？**

异步的使用场景:

1、 不涉及共享资源，获对共享资源只读，即非互斥操作；

2、 没有时序上的严格关系；

3、 不需要原子操作，或可以通过其他方式控制原子性；

4、 常用于IO操作等耗时操作，因为比较影响客户体验和使用性能；

5、 不影响主线程逻辑；

**066、多线程共同操作同一个数据互斥锁同步？**

**import** threading

**import** time

**class** **MyThread**(threading.Thread):

**def** **run**(self):

**global** num

time.sleep(1)

**if** mutex.acquire(1):

num +=1

msg = self.name + 'set num to ' +str(num)

**print** msg

mutex.release()

num = 0

mutex = threading.Lock()

**def** **test**():

**for** i **in** range(5):

t = MyThread()

t.start()

**if** \_\_name\_\_=="\_\_main\_\_":

test()

**067、什么是多线程竞争？**

线程是非独立的，同一个进程里线程是数据共享的，当各个线程访问数据资源时会出现竞争状态即：数据几乎同步会被多个线程占用，造成数据混乱，即所谓的线程不安全；

那么怎么解决多线程竞争问题？---锁；

锁的好处： 确保了某段关键代码（共享数据资源）只能由一个线程从头到尾完整地执行能解决多线程资源竞争下的原子操作问题。

锁的坏处： 阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了；

锁的致命问题: 死锁

**068、请介绍一下Python的线程同步？**

一、 setDaemon(False) 当一个进程启动之后，会默认产生一个主线程，因为线程是程序执行的最小单位，当设置多线程时，主线程会创建多个子线程，在Python中，默认情况下就是setDaemon(False),主线程执行完自己的任务以后，就退出了，此时子线程会继续执行自己的任务，直到自己的任务结束。 例子

**import** threading

**import** time

**def** **thread**():

time.sleep(2)

print('---子线程结束---')

**def** **main**():

t1 = threading.Thread(target=thread)

t1.start()

print('---主线程--结束')

**if** \_\_name\_\_ =='\_\_main\_\_':

main()

*#执行结果*

---主线程--结束

---子线程结束---

二、 setDaemon（True) 当我们使用setDaemon(True)时，这是子线程为守护线程，主线程一旦执行结束，则全部子线程被强制终止 例子

import threading

import time

def thread():

time.sleep(2)

print(’---子线程结束---')

def main():

t1 = threading.Thread(target=thread)

t1.setDaemon(True)#设置子线程守护主线程

t1.start()

print('---主线程结束---')

if \_\_name\_\_ =='\_\_main\_\_':

main()

*#执行结果*

**---主线程结束---** #只有主线程结束，子线程来不及执行就被强制结束

三、 join（线程同步) join 所完成的工作就是线程同步，即主线程任务结束以后，进入堵塞状态，一直等待所有的子线程结束以后，主线程再终止。

当设置守护线程时，含义是主线程对于子线程等待timeout的时间将会杀死该子线程，最后退出程序，所以说，如果有10个子线程，全部的等待时间就是每个timeout的累加和，简单的来说，就是给每个子线程一个timeou的时间，让他去执行，时间一到，不管任务有没有完成，直接杀死。

没有设置守护线程时，主线程将会等待timeout的累加和这样的一段时间，时间一到，主线程结束，但是并没有杀死子线程，子线程依然可以继续执行，直到子线程全部结束，程序退出。 例子

**import** threading

**import** time

**def** **thread**():

time.sleep(2)

print('---子线程结束---')

**def** **main**():

t1 = threading.Thread(target=thread)

t1.setDaemon(**True**)

t1.start()

t1.join(timeout=1)*#1 线程同步，主线程堵塞1s 然后主线程结束，子线程继续执行*

*#2 如果不设置timeout参数就等子线程结束主线程再结束*

*#3 如果设置了setDaemon=True和timeout=1主线程等待1s后会强制杀死子线程，然后主线程结束*

print('---主线程结束---')

**if** \_\_name\_\_=='\_\_main\_\_\_':

main()

**069、解释以下什么是锁，有哪几种锁？**

锁(Lock)是python提供的对线程控制的对象。有互斥锁，可重入锁（RLock），死锁；

python threading模块有两类锁：互斥锁（threading.Lock ）和可重用锁（threading.RLock）。

**070、什么是GIL锁?**

全局解释锁，每次只能一个线程获得cpu的使用权：为了线程安全，也就是为了解决多线程之间的数据完整性和状态同步而加的锁，因为我们知道线程之间的数据是共享的。

**071、gil锁的局限性和打破方式？**

局限性: 在Cpython解释器中，同一个进程下开启的多线程，同一时刻只能有一个线程执行，无法利用多核优势；

GIL存在原因： CPython在执行多线程的时候并不是线程安全的，所以为了程序的稳定性，加一把全局解释锁，能够确保任何时候都只有一个Python线程执行。

GIL的弊端：GIL对计算密集型的程序会产生影响。因为计算密集型的程序，需要占用系统资源。GIL的存在，相当于始终在进行单线程运算，这样自然就慢了。

IO密集型影响不大的原因在于，IO，input/output，这两个词就表明程序的瓶颈在于输入所耗费的时间，线程大部分时间在等待，所以它们是多个一起等（多线程）还是单个等（单线程）无所谓的。  这就好比，你在公交站等公交时，你们排队等公交（单线程）还是沿着马路一字排开等（多线程）是无所谓的。公交车（即input，即输入的资源）没来，哪种方式都是瞎折腾。

解决方案(打破方式)：

　　　multiprocessing：

　　　multiprocessing是一个多进程模块，开多个进程，每个进程都带一个GIL，就相当于多线程来用了。

　　　multiprocessing的弊端，多线程与多进程一个不同点在于：

  　　　　　多线程是共享内存的，即这些线程共用一个内存地址。好处在于便于线程间数据通信和数据同步。

　　　　　　多进程，各个进程地址之间是独立的内存地址。这样不存内存地址之间通信就麻烦了。

　　　　　　综上所述，如果是IO密集型且对数据通信有需求，使用python 的threading模块也是可以的。

　 解决方法：

1. 使用多进程执行，此将要面临解决共享数据的问题，多用queue或pipe解决；

2. 使用Python多线程load C的module执行。

 　　　　　　　　 from ctypes import \*   form threading import Thread

  　　　　　　　　#加载动态库

lib = cdll.LoadLibrary("./libdeadloop.so")

 　　　　　　　　 #创建一个子线程，让其执行c语言编写的函数，此函数是一个死循环

t = Thread(target=lib.DeadLoop)   t.start()

  　　　　　　　　 while True:    pass

 网友博客理解：

　　GIL是限制同一个进程中只有一个线程进入Python解释器。。。。。

　　而线程锁是由于在线程进行数据操作时保证数据操作的安全性(同一个进程中线程之间可以共用信息，如果同时对数据进行操作，则会出现公共数据错误) ；

　　其实线程锁完全可以替代GIL，但是Python的后续功能模块都是加在GIL基础上的，所以无法更改或去掉GIL,这就是Python语言最大的bug…只能用多进程或协程改善，或者直接用其他语言写这部分；

　　追问 ：

　　GIL本质就是一把互斥锁，既然是互斥锁，所有互斥锁的本质都一样，都是将并发运行变成串行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安全。保护不同的数据的安全，就应该加不同的锁。每执行一个python程序，就是开启一个进程，在一个python的进程内，不仅有其主线程或者由该主线程开启的其他线程，还有解释器开启的垃圾回收等解释器级别的线程，所有的线程都运行在这一个进程内，

　　　　所以：

　　　　1、所有数据都是共享的，这其中，代码作为一种数据也是被所有线程共享的（test.py的所有代码以及Cpython解释器的所有代码）；

　　　　2、所有线程的任务，都需要将任务的代码当做参数传给解释器的代码去执行，即所有的线程要想运行自己的任务，首先需要解决的是能够访问到解释器的代码。

　　在python的原始解释器CPython中存在着GIL（Global Interpreter Lock，全局解释器锁），因此在解释执行python代码时，会产生互斥锁来限制线程对共享资源的访问，直到解释器遇到I/O操作或者操作次数达到一定数目时才会释放GIL。 所以，虽然CPython的线程库直接封装了系统的原生线程，但CPython整体作为一个进程，同一时间只会有一个获得GIL的线程在跑，其他线程则处于等待状态。 这就造成了即使在多核CPU中，多线程也只是做着分时切换而已。 不过muiltprocessing的出现，已经可以让多进程的python代码编写简化到了类似多线程的程度了

我对 GIL的理解：

　　解决多线程之间数据完整性和状态同步的最简单方法自然就是加锁。

　　GIL锁开始运作主线程做操作，主线程完成操作GIL锁释放资源，所以多线程共同操作共享资源的时候，有一个线程竞得了资源，它就被GIL锁保护起来，其他线程只能是在那里等着，

　　但是这个时候，线程的休眠唤醒，全部会消耗CPU资源，所以嘞，就会慢。Python语言和GIL解释器锁没有关系，它是在实现Python解析器(CPython)时所引入的一个概念，同样一段代码可以通过CPython，PyPy，Psyco等不同的Python执行环境来执行，然而因为CPython是大部分环境下默认的Python执行环境。所以在很多人的概念里CPython就是Python，也就想当然的把GIL归结为Python语言的缺陷，所有GIL并不是python的特性，仅仅是因为历史原因在Cpython解释器中难以移除。

  　  GIL保证同一时刻只有一个线程执行代码，每个线程在执行过程中都要先获取GIL；

  　 线程释放GIL锁的情况：在IO操作等可能会引起阻塞的system call之前,可以暂时释放GIL,但在执行完毕后,必须重新获取GIL Python 3.x使用计时器（执行时间达到阈值后，当前线程释放GIL）或Python 2.x，tickets计数达到100；

  　　Python使用多进程是可以利用多核的CPU资源的。 多线程爬取比单线程性能有提升，因为遇到IO阻塞会自动释放GIL锁；

　　　  GIL只对计算密集型的程序有作用，对IO密集型的程序并没有影响，因为遇到IO阻塞会自动释放GIL锁   当需要执行计算密集型的程序时，

　　　  可以选择：1.换解释器，2.扩展C语言，3.换多进程等方案；

　　　 GIL（Global Interpreter Lock）：全局解释器锁，python解释器在执行python字节码的时候会锁住解释器，导致其它的线程不能使用解释器，从而多线程情况下CPU上不去。

　　　 lupa是一个python调用lua的第三方库（https://pypi.python.org/pypi/lupa），lua\_code是一段纯CPU计算的lua代码片段，之后开启了3个线程，可以发现CPU利用率达到了300%  之前在写一些python程序的时候，如果是cpu密集的常常会使用多进程的方式，但是这样会有一些缺点：

　　　　　　1. 进程间共享数据特别麻烦，虽然multiprocessing库提供了很多进程间共享数据的方法，但是这些方法最后自己会成为瓶颈；

 　　　　　 2. 编程复杂度比较高；

  　　　　　3. 主进程和子进程必然需要通信，进程间数据隔离，所以数据需要内存拷贝，成本高；相应的python代码：

 　　　　　　　　 #python lupa load

  　　　　　　　　import lupa   lua = lupa.LuaRuntime()

 　　　　　　　　 LIBS = [    "./scripts/foo.lua",   ]

　　　　　　　　  llibs = {}

 　　　　　　　　 def get\_file\_name(filename):

　　　　　　　　　　import os

　　　　　　　　　　(\_, tmp) = os.path.split(filename)

　　　　　　　　　　(f\_name, ext) = os.path.splittext(tmp)

　　　　　　　　　　return f\_name

　　　　　　　　  def load\_libs():

　　　　　　　　　　global LIBS, llibs    for lib\_p in LIBS:     f = open(lib\_p, 'r')

　　　　　　　　　　code\_str = f.readlines()     filename = get\_file\_name(lib\_p)

　　　　　　　　　　llibs[filename] = lua.execute('\n'.join(code\_str))

　　　　　　　　 if \_\_name\_\_ == '\_\_main\_\_':

　　　　　　　　　　 load\_libs()

　　　　　　　　　　print llibs['foo'].sayhi()

　　　　　　　　　　print llibs['foo'].callback(100, 200, 300, 400)

　　　　　　　　　　--foo.lualibfoo = {}

  　　　　　　　　　  function libfoo.sayhi()

　　　　　　　　　　return "hi from lupa"   end  function libfoo.callback(a, b, c, d)    return a \* b + c - d   endreturn libfoo

　　这样做有几点好处：

 　　　　1. python写框架，lua写回调，每次调用一遍load\_libs就相当于一次热更新

 　　　　2. lua代码本身特别简单，可以交给策划配置，与热更新结合效果更好

 　　　　3. python多线程结合lua使用可以突破python GIL的限制，后面补充一个demo

**073、线程安全？**

线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。   不会出现数据不一致或者数据污染。

线程不安全：就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。

**074、你要怎么来开多线程？**

答：直接用Threading模块开启多线程 或者用线程池的方式。

线程有 **就绪、阻塞、运行** 三种基本状态。  
1. 就绪状态是指线程具备运行的所有条件，逻辑上可以运行，在等待处理机；  
2. 运行状态是指线程占有处理机正在运行；  
3. 阻塞状态是指线程在等待一个事件（如某个信号量），逻辑上不可执行。

GIL 本质就是一把互斥锁，既然是互斥锁，所有互斥锁的本质都一样，都是将并发运行变成串行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安全。

Python 提供多线程编程的模块有以下几个：  
- \_thread  
- threading  
- Queue  
- multiprocessing

一般来说，使用线程有两种模式，一种是创建线程要执行的函数，把这个函数传递进Thread对象里，让它来执行；另一种是直接从**threading.Thread**继承，创建一个新的class，把线程执行的代码放到这个新的 class里。

**075、讲一下python的Gil？**

答：讲了多线程不能利用多核，多进程可以，gil是用来解决垃圾回收机制问题的。

在进行GIL讲解之前，我们可以先回顾一下并行和并发的区别：

并行：多个CPU同时执行多个任务，就好像有两个程序，这两个程序是真的在两个不同的CPU内同时被执行。

并发：CPU交替处理多个任务，还是有两个程序，但是只有一个CPU，会交替处理这两个程序，而不是同时执行，只不过因为CPU执行的速度过快，而会使得人们感到是在“同时”执行，执行的先后取决于各个程序对于时间片资源的争夺。

并行和并发同属于多任务，目的是要提高CPU的使用效率。这里需要注意的是，一个CPU永远不可能实现并行，即一个CPU不能同时运行多个程序，但是可以在随机分配的时间片内交替执行（并发），就好像一个人不能同时看两本书，但是却能够先看第一本书半分钟，再看第二本书半分钟，这样来回切换。

多线程并不会充分调用两个CPU，而是会像在一个CPU上充分运转，而多进程则是会完全调用两个CPU，同时执行；

Guido van Rossum（吉多·范罗苏姆）创建python时就只考虑到单核cpu，解决多线程之间数据完整性和状态同步的最简单方法自然就是加锁， 于是有了GIL这把超级大锁。因为cpython解析只允许拥有GIL全局解析器锁才能运行程序，这样就保证了保证同一个时刻只允许一个线程可以使用cpu。由于大量的程序开发者接收了这套机制,现在代码量越来越多,已经不容易通过c代码去解决这个问题。

**什么是GIL？**

即全局解释器所（global interpreter lock），每个线程在执行时候都需要先获取GIL，保证同一时刻只有一个线程可以执行代码，即同一时刻只有一个线程使用CPU，也就是说多线程并不是真正意义上的同时执行。

那么，我们改如何解决GIL锁的问题呢？

1.更换cpython为jpython(不建议)

2.使用多进程完成多线程的任务

3.在使用多线程可以使用c语言去实现

以下是几个面试会遇到的问题，希望对大家有所帮助：

**问题1: 什么时候会释放Gil锁？**

答 :

1 遇到像 i/o操作这种 会有时间空闲情况 造成cpu闲置的情况会释放Gil；

2 会有一个专门ticks进行计数 一旦ticks数值达到100 这个时候释放Gil锁 线程之间开始竞争Gil锁(说明:ticks这个数值可以进行设置来延长或者缩减获得Gil锁的线程使用cpu的时间)；

**问题2: 互斥锁和Gil锁的关系？**

Gil锁  : 保证同一时刻只有一个线程能使用到cpu；

互斥锁 : 多线程时,保证修改共享数据时有序的修改,不会产生数据修改混乱；

首先假设只有一个进程,这个进程中有两个线程 Thread1,Thread2, 要修改共享的数据date, 并且有互斥锁；

执行以下步骤

(1)多线程运行，假设Thread1获得GIL可以使用cpu，这时Thread1获得 互斥锁lock,Thread1可以改date数据(但并没有开始修改数据)；

(2)Thread1线程在修改date数据前发生了 i/o操作 或者 ticks计数满100 (注意就是没有运行到修改data数据),这个时候 Thread1 让出了Gil,Gil锁可以被竞争；

(3) Thread1 和 Thread2 开始竞争 Gil (注意:如果Thread1是因为 i/o 阻塞 让出的Gil Thread2必定拿到Gil,如果Thread1是因为ticks计数满100让出Gil 这个时候 Thread1 和 Thread2 公平竞争)；

(4)假设 Thread2正好获得了GIL, 运行代码去修改共享数据date,由于Thread1有互斥锁lock，所以Thread2无法更改共享数据date,这时Thread2让出Gil锁 , GIL锁再次发生竞争 ；

(5)假设Thread1又抢到GIL，由于其有互斥锁Lock所以其可以继续修改共享数据data,当Thread1修改完数据释放互斥锁lock,Thread2在获得GIL与lock后才可对data进行修改；

以上描述了 互斥锁和Gil锁的 一个关系。

IO密集型的python比计算密集型的程序更能利用多线程环境带来的便利。

**问题3：GIL对线程执行的影响？**

多线程环境中，python虚拟机按照以下方式执行：

1、设置GIL

2、切换到一个线程去执行；

3、运行代码，这里有两种机制：

1、指定数量的字节码指令（100个）；

2、固定时间15ms线程主动让出控制；

4、把线程设置为睡眠状态；

5、解锁GIL；

6、再次重复以上步骤；

**GIL介绍：**

python全局解释器锁(global interpreter lock, GIL)限制了任何时候只能有一个thread处于运行状态，这对于cpu密集型和多线程程序并不友好，会带来性能瓶颈。

**GIL解决的问题：**

python用引用计数来管理内存对象。当对象的引用计数变量为0的时候，对象占用的内存方可释放。引用计数变量是一个竞态条件，多个线程同时访问的时候需要进行互斥。如果不互斥，可能导致内存泄漏。

这个问题可以通过所有对象加锁来解决，但是这会导致死锁，性能等其他更复杂的问题。

GIL是给解释器自身加锁，任何python代码的执行都需要先获得解释器锁。这就解决了死锁问题(只有一个锁)，并且不会带来额外的性能问题。但是却导致cpu型的任务，任意时刻只能同时运行一个thread。

GIL并不是这个问题的唯一解决方案。线程安全的内存管理除了引用计数，也可以通过垃圾回收机制解决。但是这样会移除GIL带来的优势，单线程程序和IO型多线程程序的性能损失。

**为什么选择GIL：**

python的设计就是简单易用，快速开发，让更多的开发者参与其中。  
很多extensions需要GIL的线程安全内存管理。一些不是线程安全的C libraries可以很容易的集成到python中。并且GIL的实现很简单。对于单线程的程序也有性能的提升, GIL也是促使python如此流行的一个因素。

**对于多线程程序的影响：**

在cpu型多线程程序中，GIL阻止了线程的并行执行。对于IO型的程序，GIL并没有太大的影响，因为当等待IO操作的时候，会进行线程切换，锁是在线程之间共享的。

**GIL为什么没有被移除：**

移除GIL存在遗留的兼容性问题，还有很多C extensions依赖于GIL的方案。新的方案替代GIL，也会损失单线程程序和IO型多线程程序的性能，没人会希望新的版本反而导致已有程序的性能下降。

**怎么解决GIL带来的影响：**

利用多进程模块multiprocessing。多进程会带来显著的性能提升，但是不是成倍的，因为进程比线程更重，有其他开销。GIL存在于CPython，如果条件允许，也可以尝试用其他语言实现的python版本，譬如java实现的版本Jython。

**深入理解GIL：**

python线程是真正的系统线程，posix threads(pthreads), windows threads。  
完全由os管理。线程在运行时候持有GIL，在等待IO操作的时候会释放GIL。  
python并没有自己的线程调度机制，所有的线程调度依赖于OS。这里会有另一个问题，就是signal的处理，signal只能在main thread中被处理，而python解释器无法控制线程调度，所以只能期望更快的切换线程，让主线程得以运行。

cpu型任务：

对于cpu型的任务，解释器会定时的执行check动作，进行线程的切换。这里的定时单位是tick，tick是python解释器的一个指令运行时间。python指令可以通过dis模块查看。

对于广大写Python的人来说，GIL(Global Interpreter Lock, 全局解释器锁)肯定不陌生，但未必清楚GIL的历史和全貌是怎样的，今天我们就来梳理一下GIL。

1. 什么是GIL？

GIL的全称是 Global Interpreter Lock，全局解释器锁。之所以叫这个名字，是因为Python的执行依赖于解释器。Python最初的设计理念在于，为了解决多线程之间数据完整性和状态同步的问题，设计为在任意时刻只有一个线程在解释器中运行。而当执行多线程程序时，由GIL来控制同一时刻只有一个线程能够运行。即Python中的多线程是表面多线程，也可以理解为fake多线程，不是真正的多线程。

可能有的同学会问，同一时刻只有一个线程能够运行，那么是怎么执行多线程程序的呢？其实原理很简单：解释器的分时复用。即多个线程的代码，轮流被解释器执行，只不过切换的很频繁很快，给人一种多线程“同时”在执行的错觉。聊的学术化一点，其实就是“并发”。

再拓展一点“并发”和“并行”的概念：

普通解释：  
并发：交替做不同事情的能力  
并行：同时做不同事情的能力  
专业术语：  
并发：不同的代码块交替执行  
并行：不同的代码块同时执行

那么问题来了，Python为什么要如此设计呢？即为什么要保证同一时刻只有一个线程在解释器中运行呢？

答案是为了Python解释器中原子操作的线程安全。

2. 什么是线程安全，什么又是原子操作？

2.1 线程安全？

我们首先要搞清楚什么是进程，什么是线程。进程是系统资源分配的最小单位，线程是程序执行的最小单位。

举一个例子，如果我们把跑程序比作吃饭，那么进程就是摆满了饭菜的桌子，线程就是吃饭的那个人。

在多线程环境中，当各线程不共享数据的时候，那么一定是线程安全的。问题是这种情况并不多见，在多数情况下需要共享数据，这时就需要进行适当的同步控制了。

线程安全一般都涉及到synchronized，就是多线程环境中，共享数据同一时间只能有一个线程来操作 不然中间过程可能会产生不可预制的结果。

接着刚才的例子，桌子上有三碗米饭，一个人正在吃，吃了两碗米饭，但是还没有吃完，因此桌子上米饭的数量还没有更新；此时第二个人也想吃米饭，如果没有线程安全方面的考虑，第二个人要是想直接拿三碗米饭吃，就会出错。

2.2 原子操作？

2.2.1 什么是原子操作

原子操作就是不会因为进程并发或者线程并发而导致被中断的操作。原子操作的特点就是要么一次全部执行，要么全不执行。不存在执行了一半而被中断的情况。

当对全局资源存在写操作时，如果不能保证写入过程的原子性，会出现脏读脏写的情况。

3. GIL的优点与缺点？

GIL的优点是显而易见的，GIL可以保证我们在多线程编程时，无需考虑多线程之间数据完整性和状态同步的问题。

GIL缺点是：我们的多线程程序执行起来是“并发”，而不是“并行”。因此执行效率会很低，会不如单线程的执行效率。

网上很多人都提到过这样的疑问：”为什么我多线程Python程序运行得比其只有一个线程的时候还要慢?“显然，大家觉得一个具有两个线程的程序要比其只有一个线程时要快。事实上,这个问题是确实存在的，原因在于GIL的存在使得Python多线程程序的执行效率甚至比不上单线程的执行效率。很简单，由于GIL使得同一时刻只有一个线程在运行程序，再加上切换线程和竞争GIL带来的开销，显然Python多线程的执行效率就比不上单线程的执行效率了。

4. 为什么会有GIL，GIL的历史？

大家显然会继续思考，为什么GIL需要保证只有一个线程在某一时刻处于运行中?难道不可以添加细粒度的锁来阻止多个独立对象的同时访问?并且为什么之前没有人去尝试过类似的事情?

这些实用的问题有着十分有趣的回答。首先要明确一点, Python解释器的实现是有多个版本的：CPython, Jpython等。CPython就是用C语言实现Python解释器，JPython是用Java实现Python解释器。那么 GIL的问题实际上是存在于CPython中的。GIL的问题得不到解决，一方面是因为CPython中一开始就使用GIL的设计理念，并且很多Package依赖于CPython甚至依赖于GIL。因此造成尾大不掉，实际上是个历史问题。

为了利用多核，Python开始支持多线程。而解决多线程之间数据完整性和状态同步的最简单方法自然就是加锁。 于是有了GIL这把超级大锁，而当越来越多的代码库开发者接受了这种设定后，他们开始大量依赖这种特性（即默认python内部对象是thread-safe的，无需在实现时考虑额外的内存锁和同步操作）。

慢慢的这种实现方式被发现是蛋疼且低效的。但当大家试图去拆分和去除GIL的时候，发现大量库代码开发者已经重度依赖GIL而非常难以去除了。有多难？做个类比，像MySQL这样的“小项目”为了把Buffer Pool Mutex这把大锁拆分成各个小锁也花了从5.5到5.6再到5.7多个大版为期近5年的时间，本且仍在继续。MySQL这个背后有公司支持且有固定开发团队的产品走的如此艰难，那又更何况Python这样核心开发和代码贡献者高度社区化的团队呢？

GIL对诸如当前线程状态和为垃圾回收而用的堆分配对象这样的东西的访问提供着保护。这是该实现的一种典型产物。现在也有其它的Python解释器(和编译器)并不使用GIL。虽然,对于CPython来说,自其出现以来已经有很多不使用GIL的解释器。

那么为什么不抛弃GIL呢?许多人也许不知道,在1999年,针对Python 1.5,一个经常被提到但却不怎么理解的“free threading”补丁已经尝试实现了这个想法,该补丁来自Greg Stein。在这个补丁中,GIL被完全的移除,且用细粒度的锁来代替。然而,GIL的移除给单线程程序的执行速度带来了一定的代价。当用单线程执行时,速度大约降低了40%。使用两个线程展示出了在速度上的提高,但除了这个提高,这个收益并没有随着核数的增加而线性增长。由于执行速度的降低,这一补丁被拒绝了,并且几乎被人遗忘。

不过,“free threading”这个补丁是有启发性意义的,其证明了一个关于Python解释器的基本要点:移除GIL是非常困难的。由于该补丁发布时所处的年代,解释器变得依赖更多的全局状态,这使得想要移除当今的GIL变得更加困难。值得一提的是,也正是因为这个原因,许多人对于尝试移除GIL变得更加有兴趣。困难的问题往往很有趣。

但是这可能有点被误导了。让我们考虑一下:如果我们有了一个神奇的补丁,其移除了GIL,并且没有对单线程的Python代码产生性能上的下降,那么我们将会获得我们一直想要的:一个线程API可能会同时利用所有的处理器。但这确实是一个好事吗?

基于线程的编程毫无疑问是困难的。在编码过程中，总是会悄无声息的出现一些新的问题。因此有一些非常知名的语言设计者和研究者已经总结得出了一些线程模型。就像某个写过多线程应用的人可以告诉你的一样,不管是多线程应用的开发还是调试都会比单线程的应用难上数倍。程序员通常所具有的顺序执行的思维模恰恰就是与并行执行模式不相匹配。GIL的出现无意中帮助了开发者免于陷入困境。在使用多线程时仍然需要同步的情况下,GIL事实上帮助我们保持不同线程之间的数据一致性问题。

所以简单的说GIL的存在更多的是历史原因。如果推到重来，多线程的问题依然还是要面对，但是至少会比目前GIL这种方式会更优雅。

5. 如何规避GIL带来的影响？

用multiprocess（多进程）替代Thread（推荐）

multiprocess库的出现很大程度上是为了弥补thread库因为GIL而低效的缺陷。它完整的复制了一套thread所提供的接口方便迁移。唯一的不同就是它使用了多进程而不是多线程。每个进程有自己的独立的GIL，因此也不会出现进程之间的GIL争抢。

当然multiprocess也不是万能良药。它的引入会增加程序实现时线程间数据通讯和同步的困难。就拿计数器来举例子，如果我们要多个线程累加同一个变量，对于thread来说，申明一个global变量，用thread.Lock的context包裹住三行就搞定了。而multiprocess由于进程之间无法看到对方的数据，只能通过在主线程申明一个Queue，put再get或者用share memory的方法。这个额外的实现成本使得本来就非常痛苦的多线程程序编码，变得更加痛苦了。

用其他解析器（不推荐）

之前也提到了既然GIL只是CPython的产物，那么其他解析器是不是更好呢？没错，像JPython和IronPython这样的解析器由于实现语言的特性，他们不需要GIL的帮助。然而由于用了Java/C#用于解析器实现，他们也失去了利用社区众多C语言模块有用特性的机会。所以这些解析器也因此一直都比较小众。毕竟功能和性能大家在初期都会选择前者，Done is better than perfect。

**GIL与互斥锁？**

值得注意的是GIL 并不会保护开发者自己编写的代码。这是因为同一时刻固然只能有一个 Python 线程得到执行，但是，当这个线程正在操作某个数据结构的时候，其他线程可能会打断它，一旦发生这种现象，就会破坏程序的状态，从而使相关的数据结构无法保持其一致性。为了保证所有线程能够得到公平地执行，Python 解释器会给每个线程分配大致相等的处理器时间。为了达到这样的分配策略，Python 系统可能当某个线程正在执行的时候将其暂停，然后使另一个线程继续往下执行。由于我们无法提前获知 Python 系统会在何时暂停这些线程，所以我们无法控制程序中某些操作是原子操作。

为了防止线程中出现数据竞争的行为，使开发者可以保护自己的数据结构不受破坏，Python 在 threading 模块中提供了最简单、最有用的工具：Lock 类，该类相当于互斥锁。

在开发中我们可以使用互斥锁来保护某个对象，使得在多线程同时访问某个对象的时候，不会将该对象破坏。因为同一时刻，只有一个线程能够获得这把锁。也就是说对将要访问的对象进行隔离，那么使用线程隔离的意义在于：是当前线程能够正确的引用到它自己创造的对象，而不是引用到其它线程锁创建的对象。

**总结：**

Python GIL其实是功能和性能之间权衡后的产物，它尤其存在的合理性，也有较难改变的客观因素。我们可以做以下一些简单的总结：

因为GIL的存在，只有IO Bound场景下得多线程会得到较好的性能

如果对并行计算性能较高的程序可以考虑把核心部分也成C模块，或者索性用其他语言实现；

在Python编程中，如果想利用计算机的多核提高程序执行效率，用多进程代替多线程；

即使有GIL存在，由于GIL只保护Python解释器的状态，所以对于非原子操作，在Python进行多线程编程时也需要使用互斥锁（如thread中的lock）保证线程安全。

GIL在较长一段时间内将会继续存在，但是会不断对其进行改进；

**070、什么是死锁？简述线程死锁是怎么造成的，如何避免？GIL锁？**

死锁概念处理策略详细介绍:<https://wizardforcel.gitbooks.io/wangdaokaoyan-os/content/10.html>

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

**GIL锁（有时候，面试官不问，你自己要主动说，增加b格，尽量别一问一答的尬聊，不然最后等到的一句话就是：你还有什么想问的么？）**

**GIL锁 全局解释器锁（只在cpython里才有）；**

**作用：限制多线程同时执行，保证同一时间只有一个线程执行，所以cpython里的多线程其实是伪多线程!**

**所以Python里常常使用协程技术来代替多线程，协程是一种更轻量级的线程，**

**进程和线程的切换时由系统决定，而协程由我们程序员自己决定，而模块gevent下切换是遇到了耗时操作才会切换。**

**三者的关系：进程里有线程，线程里有协程。**

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

死锁的产生原因？

1、系统资源的竞争（竞争资源）；

2、进程运行推进顺序不当；

必要条件:

1、互斥条件；

2、请求和保持条件；

3、不剥夺条件；

4、环路等待条件；

解决死锁（大部分语言的处理方法：1、预防死锁(摒弃除1以外的条件)；2、避免死锁(银行家算法)；3、检测死锁(资源分配图)；4、解除死锁 ：01、剥夺资源；02、撤销进程：）

1、加锁顺序：线程按照一定的顺序加锁；

2、加锁时限：线程尝试获取锁的时候加上一定的时限，超过时限，则放弃对该锁的请求，并释放自己占有的锁；

3、死锁检测；

三者的关系：进程里有线程，线程里有协程。

GIL锁：全局解释器锁（只在cython里才有）（线程全局锁(Global Interpreter Lock)）：作用： 即Python为了保证线程安全而采取的独立线程运行的限制,说白了就是限制多线程同时执行，保证同一时间（一个时间点）cpu的一个内核只能运行一个线程（只有一个线程执行）。所以cython里的多线程其实是伪多线程！ 所以python里常常使用协程技术来代替多线程，协程是一种更轻量级的线程。

进程和线程的切换时由系统决定，而协程由我们程序员自己决定，而模块gevent下切换是遇到了耗时操作时才会切换 。

对于io密集型任务，python的多线程起到作用，但对于cpu密集型任务，python的多线程几乎占不到任何优势，还有可能因为争夺资源而变慢。解决办法就是多进程和协程(协程也只是单CPU,但是能减小切换代价提升性能).

**详细版：**

一、死锁的定义

多线程以及多进程改善了系统资源的利用率并提高了系统 的处理能力。然而，并发执行也带来了新的问题——死锁。所谓死锁是指多个线程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。

下面我们通过一些实例来说明死锁现象。

先看生活中的一个实例，2个人一起吃饭但是只有一双筷子，2人轮流吃（同时拥有2只筷子才能吃）。某一个时候，一个拿了左筷子，一人拿了右筷子，2个人都同时占用一个资源，等待另一个资源，这个时候甲在等待乙吃完并释放它占有的筷子，同理，乙也在等待甲吃完并释放它占有的筷子，这样就陷入了一个死循环，谁也无法继续吃饭。。。

在计算机系统中也存在类似的情况。例如，某计算机系统中只有一台打印机和一台输入 设备，进程P1正占用输入设备，同时又提出使用打印机的请求，但此时打印机正被进程P2 所占用，而P2在未释放打印机之前，又提出请求使用正被P1占用着的输入设备。这样两个进程相互无休止地等待下去，均无法继续执行，此时两个进程陷入死锁状态。

二、死锁产生的原因：

1) 系统资源的竞争：

通常系统中拥有的不可剥夺资源，其数量不足以满足多个进程运行的需要，使得进程在 运行过程中，会因争夺资源而陷入僵局，如磁带机、打印机等。只有对不可剥夺资源的竞争 才可能产生死锁，对可剥夺资源的竞争是不会引起死锁的。

2) 进程推进顺序非法：

进程在运行过程中，请求和释放资源的顺序不当，也同样会导致死锁。例如，并发进程 P1、P2分别保持了资源R1、R2，而进程P1申请资源R2，进程P2申请资源R1时，两者都 会因为所需资源被占用而阻塞。

信号量使用不当也会造成死锁。进程间彼此相互等待对方发来的消息，结果也会使得这 些进程间无法继续向前推进。例如，进程A等待进程B发的消息，进程B又在等待进程A 发的消息，可以看出进程A和B不是因为竞争同一资源，而是在等待对方的资源导致死锁。

3) 死锁产生的必要条件：

产生死锁必须同时满足以下四个条件，只要其中任一条件不成立，死锁就不会发生。

1、互斥条件：进程要求对所分配的资源（如打印机）进行排他性控制，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。

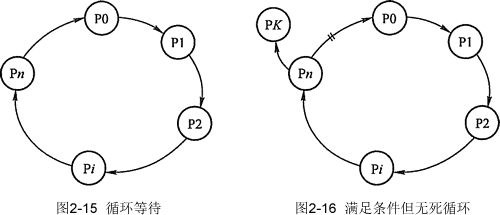
2、不剥夺条件：进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。

3、请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

4、循环等待条件：存在一种进程资源的循环等待链，链中每一个进程已获得的资源同时被 链中下一个进程所请求。即存在一个处于等待状态的进程集合{Pl, P2, ..., pn}，其中Pi等 待的资源被P(i+1)占有（i=0, 1, ..., n-1)，Pn等待的资源被P0占有，如图2-15所示。

直观上看，循环等待条件似乎和死锁的定义一样，其实不然。按死锁定义构成等待环所 要求的条件更严，它要求Pi等待的资源必须由P(i+1)来满足，而循环等待条件则无此限制。 例如，系统中有两台输出设备，P0占有一台，PK占有另一台，且K不属于集合{0, 1, ..., n}。

Pn等待一台输出设备，它可以从P0获得，也可能从PK获得。因此，虽然Pn、P0和其他 一些进程形成了循环等待圈，但PK不在圈内，若PK释放了输出设备，则可打破循环等待, 如图2-16所示。因此循环等待只是死锁的必要条件。



资源分配图含圈而系统又不一定有死锁的原因是同类资源数大于1。但若系统中每类资 源都只有一个资源，则资源分配图含圈就变成了系统出现死锁的充分必要条件。

产生死锁的一个例子

/\*\*

\* 一个简单的死锁类

\* 当DeadLock类的对象flag==1时（td1），先锁定o1,睡眠500毫秒

\* 而td1在睡眠的时候另一个flag==0的对象（td2）线程启动，先锁定o2,睡眠500毫秒

\* td1睡眠结束后需要锁定o2才能继续执行，而此时o2已被td2锁定；

\* td2睡眠结束后需要锁定o1才能继续执行，而此时o1已被td1锁定；

\* td1、td2相互等待，都需要得到对方锁定的资源才能继续执行，从而死锁。

\*/

public class DeadLock implements Runnable {

public int flag = 1;

//静态对象是类的所有对象共享的

private static Object o1 = new Object(), o2 = new Object();

@Override

public void run() {

System.out.println("flag=" + flag);

if (flag == 1) {

synchronized (o1) {

try {

Thread.sleep(500);

} catch (Exception e) {

e.printStackTrace();

}

synchronized (o2) {

System.out.println("1");

}

}

}

if (flag == 0) {

synchronized (o2) {

try {

Thread.sleep(500);

} catch (Exception e) {

e.printStackTrace();

}

synchronized (o1) {

System.out.println("0");

}

}

}

}

public static void main(String[] args) {

DeadLock td1 = new DeadLock();

DeadLock td2 = new DeadLock();

td1.flag = 1;

td2.flag = 0;

//td1,td2都处于可执行状态，但JVM线程调度先执行哪个线程是不确定的。

//td2的run()可能在td1的run()之前运行

new Thread(td1).start();

new Thread(td2).start();

}

}

**三、如何避免死锁**

在有些情况下死锁是可以避免的。三种用于避免死锁的技术：

加锁顺序（线程按照一定的顺序加锁）

加锁时限（线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁）

死锁检测

加锁顺序

当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。

如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。看下面这个例子：

Thread 1:

lock A

lock B

Thread 2:

wait for A

lock C (when A locked)

Thread 3:

wait for A

wait for B

wait for C

如果一个线程（比如线程3）需要一些锁，那么它必须按照确定的顺序获取锁。它只有获得了从顺序上排在前面的锁之后，才能获取后面的锁。

例如，线程2和线程3只有在获取了锁A之后才能尝试获取锁C(译者注：获取锁A是获取锁C的必要条件)。因为线程1已经拥有了锁A，所以线程2和3需要一直等到锁A被释放。然后在它们尝试对B或C加锁之前，必须成功地对A加了锁。

按照顺序加锁是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁(译者注：并对这些锁做适当的排序)，但总有些时候是无法预知的。

加锁时限

另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。若一个线程没有在给定的时限内成功获得所有需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试。这段随机的等待时间让其它线程有机会尝试获取相同的这些锁，并且让该应用在没有获得锁的时候可以继续运行(译者注：加锁超时后可以先继续运行干点其它事情，再回头来重复之前加锁的逻辑)。

以下是一个例子，展示了两个线程以不同的顺序尝试获取相同的两个锁，在发生超时后回退并重试的场景：

Thread 1 locks A

Thread 2 locks B

Thread 1 attempts to lock B but is blocked

Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out

Thread 1 backs up and releases A as well

Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out

Thread 2 backs up and releases B as well

Thread 2 waits randomly (e.g. 43 millis) before retrying.

在上面的例子中，线程2比线程1早200毫秒进行重试加锁，因此它可以先成功地获取到两个锁。这时，线程1尝试获取锁A并且处于等待状态。当线程2结束时，线程1也可以顺利的获得这两个锁（除非线程2或者其它线程在线程1成功获得两个锁之前又获得其中的一些锁）。

需要注意的是，由于存在锁的超时，所以我们不能认为这种场景就一定是出现了死锁。也可能是因为获得了锁的线程（导致其它线程超时）需要很长的时间去完成它的任务。

此外，如果有非常多的线程同一时间去竞争同一批资源，就算有超时和回退机制，还是可能会导致这些线程重复地尝试但却始终得不到锁。如果只有两个线程，并且重试的超时时间设定为0到500毫秒之间，这种现象可能不会发生，但是如果是10个或20个线程情况就不同了。因为这些线程等待相等的重试时间的概率就高的多（或者非常接近以至于会出现问题）。 (译者注：超时和重试机制是为了避免在同一时间出现的竞争，但是当线程很多时，其中两个或多个线程的超时时间一样或者接近的可能性就会很大，因此就算出现竞争而导致超时后，由于超时时间一样，它们又会同时开始重试，导致新一轮的竞争，带来了新的问题。)

这种机制存在一个问题，在Java中不能对synchronized同步块设置超时时间。你需要创建一个自定义锁，或使用Java5中java.util.concurrent包下的工具。写一个自定义锁类不复杂，但超出了本文的内容。后续的Java并发系列会涵盖自定义锁的内容。

死锁检测

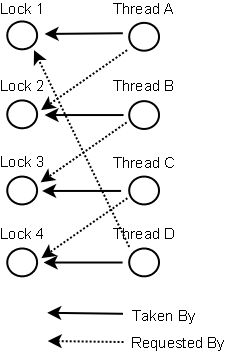
死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生。例如，线程A请求锁7，但是锁7这个时候被线程B持有，这时线程A就可以检查一下线程B是否已经请求了线程A当前所持有的锁。如果线程B确实有这样的请求，那么就是发生了死锁（线程A拥有锁1，请求锁7；线程B拥有锁7，请求锁1）。

当然，死锁一般要比两个线程互相持有对方的锁这种情况要复杂的多。线程A等待线程B，线程B等待线程C，线程C等待线程D，线程D又在等待线程A。线程A为了检测死锁，它需要递进地检测所有被B请求的锁。从线程B所请求的锁开始，线程A找到了线程C，然后又找到了线程D，发现线程D请求的锁被线程A自己持有着。这是它就知道发生了死锁。

下面是一幅关于四个线程（A,B,C和D）之间锁占有和请求的关系图。像这样的数据结构就可以被用来检测死锁。

[](http://ifeve.com/?attachment_id=3873)

那么当检测出死锁时，这些线程该做些什么呢？

一个可行的做法是释放所有锁，回退，并且等待一段随机的时间后重试。这个和简单的加锁超时类似，不一样的是只有死锁已经发生了才回退，而不会是因为加锁的请求超时了。虽然有回退和等待，但是如果有大量的线程竞争同一批锁，它们还是会重复地死锁（编者注：原因同超时类似，不能从根本上减轻竞争）。

一个更好的方案是给这些线程设置优先级，让一个（或几个）线程回退，剩下的线程就像没发生死锁一样继续保持着它们需要的锁。如果赋予这些线程的优先级是固定不变的，同一批线程总是会拥有更高的优先级。为避免这个问题，可以在死锁发生的时候设置随机的优先级。

**071、多线程交互访问数据，如果访问到了就不访问了(怎么避免重读)？**

创建一个已访问数据列表，用于存储已经访问过的数据，并加上互斥锁，在多线程访问数据的时候先查看数据是否在已访问的列表中，若已存在就直接跳过。

**072、当做个线程竞争一个对象且该对象并非线程安全的时候应该怎么办？**

**进行多线程编程，同步控制是非常重要的，而同步控制就涉及到了锁。**

对代码进行同步控制我们可以选择同步方法，也可以选择同步块，这两种方式各有优缺点，至于具体选择什么方式，就见仁见智了，同步块不仅可以更加精确的控制对象锁，也就是控制锁的作用域。

**何谓锁的作用域？**锁的作用域就是从锁被获取到其被释放的时间。而且可以选择要获取哪个对象的对象锁。但是如果在使用同步块机制时，如果使用过多的锁也会容易引起死锁问题，同时获取和释放所也有代价，而同步方法，它们所拥有的锁就是该方法所属的类的对象锁，换句话说，也就是this对象，而且锁的作用域也是整个方法，这可能导致其锁的作用域可能太大，也有可能引起死锁，同时因为可能包含了不需要进行同步的代码块在内，也会降低程序的运行效率。而不管是同步方法还是同步块，我们都不应该在他们的代码块内包含无限循环，如果代码内部要是有了无限循环，那么这个同步方法或者同步块在获取锁以后因为代码会一直不停的循环着运行下去，也就没有机会释放它所获取的锁，而其它等待这把锁的线程就永远无法获取这把锁，这就造成了一种死锁现象。

**详细解说一下同步方法的锁，同步方法分为静态同步方法与非静态同步方法。**

所有的非静态同步方法用的都是同一把锁——实例对象本身，也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，可是别的实例对象的非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，所以毋须等待该实例对象已获取锁的非静态同步方法释放锁就可以获取他们自己的锁。

而所有的静态同步方法用的也是同一把锁——类对象本身，这两把锁是两个不同的对象，所以静态同步方法与非静态同步方法之间是不会有竞态条件的。但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不管是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象！

**\*总之，对与同一个实例的非静态同步方法只对非静态方法有效，静态同步方法只对静态同步方法有效，静态同步方法和非静态同步方法是互不干扰的\*。**

而对于同步块，由于其锁是可以选择的，所以只有使用同一把锁的同步块之间才有着竞态条件，这就得具体情况具体分析了，但这里有个需要注意的地方，同步块的锁是可以选择的，但是不是可以任意选择的！！！！这里必须要注意一个物理对象和一个引用对象的实例变量之间的区别！使用一个引用对象的实例变量作为锁并不是一个好的选择，因为同步块在执行过程中可能会改变它的值，其中就包括将其设置为null，而对一个null对象加锁会产生异常，并且对不同的对象加锁也违背了同步的初衷！这看起来是很清楚的，但是一个经常发生的错误就是选用了错误的锁对象，因此必须注意：同步是基于实际对象而不是对象引用的！多个变量可以引用同一个对象，变量也可以改变其值从而指向其他的对象，因此，当选择一个对象锁时，我们要根据实际对象而不是其引用来考虑！作为一个原则，不要选择一个可能会在锁的作用域中改变值的实例变量作为锁对象！！！！

**072、什么是线程安全，什么是互斥锁？**

每个对象都对应于一个可称为’互斥锁‘的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。

同一进程中的多线程之间是共享系统资源的，多个线程同时对一个对象进行操作，一个线程操作尚未结束，另一线程已经对其进行操作，导致最终结果出现错误，此时需要对被操作对象添加互斥锁，保证每个线程对该对象的操作都得到正确的结果。

**073、说说下面几个概念：同步，异步，阻塞，非阻塞？**

同步： 多个任务之间有先后顺序执行，一个执行完下个才能执行。

异步： 多个任务之间没有先后顺序，可以同时执行，有时候一个任务可能要在必要的时候获取另一个同时执行的任务的结果，这个就叫回调！

阻塞： 如果卡住了调用者，调用者不能继续往下执行，就是说调用者阻塞了。

非阻塞： 如果不会卡住，可以继续执行，就是说非阻塞的。 同步异步相对于多任务而言，阻塞非阻塞相对于代码执行而言。

**074、什么是僵尸进程和孤儿进程？怎么避免僵尸进程？**

孤儿进程： 父进程退出，子进程还在运行的这些子进程都是孤儿进程，孤儿进程将被init 进程（进程号为1）所收养，并由init 进程对他们完成状态收集工作。

僵尸进程： 进程使用fork 创建子进程，如果子进程退出，而父进程并没有调用wait 获waitpid 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中的这些进程是僵尸进程。

避免僵尸进程的方法： 1.fork 两次用孙子进程去完成子进程的任务 2.用wait()函数使父进程阻塞 3.使用信号量，在signal handler 中调用waitpid,这样父进程不用阻塞；

**075、python中进程与线程的使用场景？**

多进程适合在CPU密集操作（cpu操作指令比较多，如位多的的浮点运算）。 多线程适合在IO密性型操作（读写数据操作比多的的，比如爬虫）

**076、线程是并发还是并行，进程是并发还是并行？**

线程是并发，进程是并行; 进程之间互相独立，是系统分配资源的最小单位，同一个线程中的所有线程共享资源。

**077、并行(parallel)和并发（concurrency)?**

并行： 同一时刻多个任务同时在运行;

并发: 在同一时间间隔内多个任务都在运行，但是并不会在同一时刻同时运行，存在交替执行的情况。

实现并行的库有： multiprocessing; CPU运算量大的程序，使用并行会更好

实现并发的库有: threading 程序需要执行较多的读写、请求和回复任务的需要大量的IO操作，IO密集型操作使用并发更好。

**078、IO密集型和CPU密集型区别？**

IO密集型：系统运行，大部分的状况是CPU在等 I/O（硬盘/内存）的读/写；

CPU密集型：大部分时间用来做计算，逻辑判断等CPU动作的程序称之CPU密集型。

**079、select,poll和epoll？**

I/O多路复用的本质就是用select/poll/epoll，去监听多个socket对象，如果其中的socket对象有变化，只要有变化，用户进程就知道了。

select是不断轮询去监听的socket，socket个数有限制，一般为1024个；

poll还是采用轮询方式监听，只不过没有个数限制；

epoll并不是采用轮询方式去监听了，而是当socket有变化时通过回调的方式主动告知用户进程。

其实所有的I/O都是轮询的方法，只不过实现的层面不同罢了。

这个问题可能有点深入了，但相信能回答出这个问题是对I/O多路复用有很好的了解了，其中tornado使用的就是 epoll。

[selec,poll和epoll区别总结](http://www.cnblogs.com/Anker/p/3265058.html)：select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。

但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

select、poll、epoll模型的区别？

1、支持一个进程所能打开的最大连接数：

select的最大连接数大概3232，或者3264；

poll本质和select没区别，但是它没有最大连接数限制；

epoll大概10万左右(1G的机器)；

2、FD剧增后带来的IO效率问题：

select和poll每次调用都会对连接进行线性遍历，所以会随着FD的增加会造成遍历速度慢的“线性下降性能问题”；

epoll没有前两个的线性下降的性能问题，但是当socket都很活跃的情况下，可能会有性能问题；

3、消息传递方式

select和poll内核需要将消息传递到用户空间，都需要内核拷贝动作；

epoll通过内核和用户空间共享一块内存来实现；

(1)select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

(2)poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的限制，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）；

epoll跟select都能提供多路I/O复用的解决方案。在现在的Linux内核里有都能够支持，其中epoll是Linux所特有，而select则应该是POSIX所规定，一般操作系统均有实现

**基本上select有3个缺点:**

1、连接数受限；

2、查找配对速度慢；

3、数据由内核拷贝到用户态；

poll改善了第一个缺点，epoll改了三个缺点。

关于epoll的: <http://www.cnblogs.com/my_life/articles/3968782.html>

**080、调度算法？**

1、先来先服务(FCFS, First Come First Serve)；

2、短作业优先(SJF, Shortest Job First)；

3、最高优先权调度(Priority Scheduling)；

4、时间片轮转(RR, Round Robin)；

5、多级反馈队列调度(multilevel feedback queue scheduling)；

常见的调度算法总结:<http://www.jianshu.com/p/6edf8174c1eb>

**08、实时调度算法:**

1、最早截至时间优先 EDF；2、最低松弛度优先 LLF；

### 080、什么是asyncio？python asyncio的原理？

python高并发模块；

asyncio是并发的一种方式，是一个协程相关的库。也叫异步IO；

asyncio这个库就是使用python的yield这个可以打断保存当前函数的上下文的机制， 封装好了selector 摆脱掉了复杂的回调关系；

### 081、程序编译与链接？

推荐: <http://www.ruanyifeng.com/blog/2014/11/compiler.html>

Bulid过程可以分解为4个步骤:预处理(Prepressing), 编译(Compilation)、汇编(Assembly)、链接(Linking)；

以c语言为例:

1、预处理

预编译过程主要处理那些源文件中的以“#”开始的预编译指令。

主要处理规则有：

1、将所有的“#define”删除，并展开所用的宏定义；

2、处理所有条件预编译指令，比如“#if”、“#ifdef”、 “#elif”、“#endif”；

3、处理“#include”预编译指令，将被包含的文件插入到该编译指令的位置，注：此过程是递归进行的；

4、删除所有注释；

5、添加行号和文件名标识，以便于编译时编译器产生调试用的行号信息以及用于编译时产生编译错误或警告时可显示行号；

6、保留所有的#pragma编译器指令。

2、编译

编译过程就是把预处理完的文件进行一系列的词法分析、语法分析、语义分析及优化后生成相应的汇编代码文件。这个过程是整个程序构建的核心部分。

3、汇编

汇编器是将汇编代码转化成机器可以执行的指令，每一条汇编语句几乎都是一条机器指令。经过编译、链接、汇编输出的文件成为目标文件(Object File)

4、链接

链接的主要内容就是把各个模块之间相互引用的部分处理好，使各个模块可以正确的拼接。

主要过程包块地址和空间的分配（Address and Storage Allocation）、符号决议(Symbol Resolution)和重定位(Relocation)等步骤。

### 082、静态链接和动态链接？

静态链接方法：静态链接的时候，载入代码就会把程序会用到的动态代码或动态代码的地址确定下来。静态库的链接可以使用静态链接，动态链接库也可以使用这种方法链接导入库。

动态链接方法：使用这种方式的程序并不在一开始就完成动态链接，而是直到真正调用动态库代码时，载入程序才计算(被调用的那部分)动态代码的逻辑地址，然后等到某个时候，程序又需要调用另外某块动态代码时，载入程序又去计算这部分代码的逻辑地址，所以，这种方式使程序初始化时间较短，但运行期间的性能比不上静态链接的程序。

### 083、虚拟内存技术？

虚拟存储器是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储系统.

### 084、分页和分段？

分页: 用户程序的地址空间被划分成若干固定大小的区域，称为“页”，相应地，内存空间分成若干个物理块，页和块的大小相等。可将用户程序的任一页放在内存的任一块中，实现了离散分配。

分段: 将用户程序地址空间分成若干个大小不等的段，每段可以定义一组相对完整的逻辑信息。存储分配时，以段为单位，段与段在内存中可以不相邻接，也实现了离散分配。

分页与分段的主要区别：

1、页是信息的物理单位,分页是为了实现非连续分配,以便解决内存碎片问题,或者说分页是由于系统管理的需要.段是信息的逻辑单位,它含有一组意义相对完整的信息,分段的目的是为了更好地实现共享,满足用户的需要；

2、页的大小固定,由系统确定,将逻辑地址划分为页号和页内地址是由机器硬件实现的.而段的长度却不固定,决定于用户所编写的程序,通常由编译程序在对源程序进行编译时根据信息的性质来划分；

3、分页的作业地址空间是一维的.分段的地址空间是二维的.

### 085、页面置换算法？

1、最佳置换算法OPT:不可能实现；

2、先进先出FIFO；

3、最近最久未使用算法LRU:最近一段时间里最久没有使用过的页面予以置换；

4、clock算法；

### 086、边缘触发和条件(水平)触发？

边缘触发是指每当状态变化时发生一个 io 事件，条件触发是只要满足条件就发生一个 io 事件。

### 087、unix进程间通信方式(IPC)

1、管道（Pipe）：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。

2、命名管道（named pipe）：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令mkfifo或系统调用mkfifo来创建。

3、信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。

4、消息（Message）队列：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺

5、共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

6、内存映射（mapped memory）：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。

7、信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。

8、套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

### 088、ARP协议？

1、ARP全称“Address Resolution Protocol”，地址解析协议。

2、实现局域网内通过IP地址获取主机的MAC地址。

3、MAC地址48位主机的物理地址，局域网内唯一。

4、ARP协议类似DNS服务，但不需要配置服务。

5、ARP协议是三层协议

地址解析协议(Address Resolution Protocol)，用ARP协议可实现通过IP地址获得对应主机的物理地址（MAC地址），其基本功能为透过目标设备的IP地址，查询目标的MAC地址，以保证通信的顺利进行。它是IPv4网络层必不可少的协议，不过在IPv6中已不再适用，并被邻居发现协议（NDP）所替代。

### 089、urllib和urllib2的区别？

这个面试官确实问过，当时答的urllib2可以Post而urllib不可以。

1、urllib提供urlencode方法用来GET查询字符串的产生，而urllib2没有。这是为何urllib常和urllib2一起使用的原因。

2、urllib2可以接受一个Request类的实例来设置URL请求的headers，urllib仅可以接受URL。这意味着，你不可以伪装你的User Agent字符串等。

post: [RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1](http://tools.ietf.org/html/rfc2616#section-9.5)

### 090、apache和nginx的区别？

nginx 相对 apache 的优点：

1、轻量级，同样起 web 服务，比 apache 占用更少的内存及资源；

2、抗并发，nginx 处理请求是异步非阻塞的，支持更多的并发连接，而 apache 则是阻塞型的，在高并发下 nginx 能保持低资源低消耗高性能；

3、配置简洁；

4、高度模块化的设计，编写模块相对简单；

5、社区活跃；

apache 相对 nginx 的优点：

1、rewrite ，比 nginx 的 rewrite 强大；

2、模块超多，基本想到的都可以找到；

3、少 bug ，nginx 的 bug 相对较多；

4、超稳定；

### 091、网站用户密码保存？

1、明文保存；

2、明文hash后保存,如md5；

3、MD5+Salt方式,这个salt可以随机；

4、知乎使用了Bcrypy(好像)加密；

### 093、幂等 Idempotence？

HTTP方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用(注意是副作用)。

1、GET http://www.bank.com/account/123456，不会改变资源的状态，不论调用一次还是N次都没有副作用。请注意，这里强调的是一次和N次具有相同的副作用，而不是每次GET的结果相同。

GET http://www.news.com/latest-news这个HTTP请求可能会每次得到不同的结果，但它本身并没有产生任何副作用，因而是满足幂等性的。

2、DELETE方法用于删除资源，有副作用，但它应该满足幂等性。比如：DELETE http://www.forum.com/article/4231，调用一次和N次对系统产生的副作用是相同的，即删掉id为4231的帖子；因此，调用者可以多次调用或刷新页面而不必担心引起错误。

3、POST所对应的URI并非创建的资源本身，而是资源的接收者。比如：POST http://www.forum.com/articles的语义是在http://www.forum.com/articles下创建一篇帖子，HTTP响应中应包含帖子的创建状态以及帖子的URI。两次相同的POST请求会在服务器端创建两份资源，它们具有不同的URI；所以，POST方法不具备幂等性。

4、PUT所对应的URI是要创建或更新的资源本身。比如：PUT http://www.forum/articles/4231的语义是创建或更新ID为4231的帖子。对同一URI进行多次PUT的副作用和一次PUT是相同的；因此，PUT方法具有幂等性。

### 097、CGI和WSGI？

CGI是通用网关接口，是连接web服务器和应用程序的接口，用户通过CGI来获取动态数据或文件等。

CGI程序是一个独立的程序，它可以用几乎所有语言来写，包括perl，c，lua，python等等。

WSGI, Web Server Gateway Interface，是Python应用程序或框架和Web服务器之间的一种接口，WSGI的其中一个目的就是让用户可以用统一的语言(Python)编写前后端。

**官方说明：**[**PEP-3333**](https://www.python.org/dev/peps/pep-3333/)

### 098、中间人攻击？

中间人攻击（Man-in-the-middle attack，通常缩写为MITM）是指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话，但事实上整个会话都被攻击者完全控制。

### 099、c10k问题？

所谓c10k问题，指的是服务器同时支持成千上万个客户端的问题，也就是concurrent 10 000 connection（这也是c10k这个名字的由来）。

推荐: <https://my.oschina.net/xianggao/blog/664275>

### 100、socket？

推荐: <http://www.360doc.com/content/11/0609/15/5482098_122692444.shtml>

Socket=Ip address+ TCP/UDP + port

### 101、浏览器缓存？

推荐: <http://www.cnblogs.com/skynet/archive/2012/11/28/2792503.html>

浏览器缓存机制：

浏览器缓存机制，其实主要就是HTTP协议定义的缓存机制（如： Expires； Cache-control等）。但是也有非HTTP协议定义的缓存机制，如使用HTML Meta 标签，Web开发者可以在HTML页面的<head>节点中加入<meta>标签等。

下面我主要介绍HTTP协议定义的缓存机制。

Expires策略：Expires是Web服务器响应消息头字段，在响应http请求时告诉浏览器在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。

Cache-control策略（重点关注）：Cache-Control与Expires的作用一致，都是指明当前资源的有效期，控制浏览器是否直接从浏览器缓存取数据还是重新发请求到服务器取数据。只不过Cache-Control的选择更多，设置更细致，如果同时设置的话，其优先级高于Expires。

### 102、HTTP1.0和HTTP1.1的请求方法？

推荐: <http://blog.csdn.net/elifefly/article/details/3964766>

1、请求头Host字段,一个服务器多个网站；

2、长链接；

3、文件断点续传；

4、身份认证,状态管理,Cache缓存；

HTTP请求8种方法介绍 .

根据HTTP 标准，HTTP 请求可以使用多种请求方法。

HTTP1.0 定义了三种请求方法： GET， POST 和HEAD 方法。

HTTP1.1 新增了五种请求方法：OPTIONS， PUT， DELETE， TRACE 和CONNECT 方法。HTTP1.1协议共定义了8种HTTP请求方法，HTTP请求方法也被叫做“请求动作”，不同的方法规定了不同的操作指定的资源方式。服务端也会根据不同的请求方法做不同的响应。

1、GET(请求指定的页面信息，并返回实体主体)：

GET请求会显示请求指定的资源。一般来说GET方法应该只用于数据的读取，而不应当用于会产生副作用的非幂等的操作中。

GET会方法请求指定的页面信息，并返回响应主体，GET被认为是不安全的方法，因为GET方法会被网络蜘蛛等任意的访问。

**2、HEAD(类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头)：**

HEAD方法与GET方法一样，都是向服务器发出指定资源的请求。但是，服务器在响应HEAD请求时不会回传资源的内容部分，即：响应主体。这样，我们可以不传输全部内容的情况下，就可以获取服务器的响应头信息。HEAD方法常被用于客户端查看服务器的性能。

**3、POST：**

POST请求会向指定资源提交数据，请求服务器进行处理，如：表单数据提交、文件上传等，请求数据会被包含在请求体中。POST方法是非幂等的方法，因为这个请求可能会创建新的资源或/和修改现有资源。

**4、PUT(从客户端向服务器传送的数据取代指定的文档的内容)：**

PUT请求会身向指定资源位置上传其最新内容，PUT方法是幂等的方法。通过该方法客户端可以将指定资源的最新数据传送给服务器取代指定的资源的内容。

**5、DELETE(请求服务器删除指定的页面)：**

DELETE请求用于请求服务器删除所请求URI（统一资源标识符，Uniform Resource Identifier）所标识的资源。DELETE请求后指定资源会被删除，DELETE方法也是幂等的。

6、CONNECT：

CONNECT方法是HTTP/1.1协议预留的，能够将连接改为管道方式的代理服务器。通常用于SSL加密服务器的链接与非加密的HTTP代理服务器的通信。

**7、OPTIONS：**

OPTIONS请求与HEAD类似，一般也是用于客户端查看服务器的性能。 这个方法会请求服务器返回该资源所支持的所有HTTP请求方法，该方法会用’\*’来代替资源名称，向服务器发送OPTIONS请求，可以测试服务器功能是否正常。JavaScript的XMLHttpRequest对象进行CORS跨域资源共享时，就是使用OPTIONS方法发送嗅探请求，以判断是否有对指定资源的访问权限。

**8、TRACE(回显服务器收到的请求，主要用于测试或诊断)：**

TRACE请求服务器回显其收到的请求信息，该方法主要用于HTTP请求的测试或诊断。

HTTP/1.1之后增加的方法

在HTTP/1.1标准制定之后，又陆续扩展了一些方法。其中使用中较多的是 PATCH 方法：

**PATCH：**

PATCH方法出现的较晚，它在2010年的RFC 5789标准中被定义。PATCH请求与PUT请求类似，同样用于资源的更新。二者有以下两点不同：

1、PATCH一般用于资源的部分更新，而PUT一般用于资源的整体更新。

2、当资源不存在时，PATCH会创建一个新的资源，而PUT只会对已在资源进行更新。

### 103、Ajax？

**AJAX,Asynchronous JavaScript and XML（异步的 JavaScript 和 XML）, 是与在不重新加载整个页面的情况下，与服务器交换数据并更新部分网页的技术。**

### 104、UDP总结？

**创建客户端套接字socket；**

**发送/接收数据；**

**关闭套接字socket；**

**1．import socket**

**2．def main():**

**3． # 1、创建udp 套接字**

**4． # socket.AF\_INET 表示IPv4 协议AF\_INET6 表示IPv6 协议**

**5． # socket.SOCK\_DGRAM 数据报套接字，只要用于udp 协议**

**6． udp\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)**

**8． # 2、准备接收方的地址**

**9． # 元组类型ip 是字符串类型端口号是整型**

**10． dest\_addr = ('192.168.113.111', 8888)**

**11． # 要发送的数据**

**12． send\_data = "我是要发送的数据"**

**13． # 3、发送数据**

**14． udp\_socket.sendto(send\_data.encode("utf-8"), dest\_addr)**

**15． # 4、等待接收方发送的数据如果没有收到数据则会阻塞等待，直到收到数据**

**16． # 接收到的数据是一个元组(接收到的数据, 发送方的ip 和端口)**

**17． # 1024 表示本次接收的最大字节数**

**18． recv\_data, addr = udp\_socket.recvfrom(1024)**

**19． # 5、关闭套接字**

**20． udp\_socket.close()**

**21．if \_\_name\_\_ == '\_\_main\_\_':**

**22． main()**

**编码的转换  
str--->bytes:  encode编码；bytes--->str;  decode编码；**

**udp绑定端口号  
创建socket；  
绑定端口号；  
接收/发送数据；  
关闭套接字；**

**1．import socket**

**2．def main():**

**3． # 1、创建udp 套接字**

**4． # socket.AF\_INET 表示IPv4 协议AF\_INET6 表示IPv6 协议**

**5． # socket.SOCK\_DGRAM 数据报套接字，只要用于udp 协议**

**6． udp\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)**

**7． # 2、绑定端口**

**8． # 元组类型ip 一般不写表示本机的任何的一个ip**

**9． local\_addr = ('', 7777)**

**10． udp\_socket.bind(local\_addr)**

**11． # 3、准备接收方的地址**

**12． # 元组类型ip 是字符串类型端口号是整型**

**13． dest\_addr = ('192.168.113.111', 8888)**

**14． # 要发送的数据**

**15． send\_data = "我是要发送的数据"**

**16． # 4、发送数据**

**17． udp\_socket.sendto(send\_data.encode("utf-8"), dest\_addr)**

**18． # 5、等待接收方发送的数据如果没有收到数据则会阻塞等待，直到收到数据**

**19． # 接收到的数据是一个元组(接收到的数据, 发送方的ip 和端口)**

**20． # 1024 表示本次接收的最大字节数**

**21． recv\_data, addr = udp\_socket.recvfrom(1024)**

**22． # 6、关闭套接字**

**23． udp\_socket.close()**

**24．if \_\_name\_\_ == '\_\_main\_\_':**

**25． main()**

**注意点：绑定端口要在发送数据之前进行绑定。**

### 105、TCP总结？

**1、tcp客户端的创建流程：  
创建tcp的套接字  
链接服务器  
发送数据给服务器端  
接收服务器端发送的消息  
关闭套接字**

**1．import socket**

**2．def main():**

**3． # 1、创建客户端的socket**

**4． # socket.AF\_INET 表示IPv4 协议AF\_INET6 表示IPv6 协议**

**5． # socket.SOCK\_STREAM 流式套接字，只要用于TCP 协议**

**6． client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)**

**7． # 2、构建目标地址**

**8． server\_ip = input("请输入服务器端的IP 地址：")**

**9． server\_port = int(input("请输入服务器端的端口号："))**

**10． # 3、连接服务器**

**11． # 参数：元组类型ip 是字符串类型端口号是整型**

**12． client\_socket.connect((server\_ip, server\_port))**

**13． # 要发送给服务器端的数据**

**14． send\_data = "我是要发送给服务器端的数据"**

**15． # 4、发送数据**

**16． client\_socket.send(send\_data.encode("gbk"))**

**17． # 5、接收服务器端恢复的消息， 没有消息会阻塞**

**18． # 1024 表示接收的最大字节数**

**19． recv\_date= client\_socket.recv(1024)**

**20． print("接收到的数据是：", recv\_date.decode('gbk'))**

**21． # 6、关闭套接字**

**22． client\_socket.close()**

**23．if \_\_name\_\_ == '\_\_main\_\_':**

**24． main()**

**2、TCP服务器的船舰流程**

**1.创建TCP 服务端的socket  
2.bing 绑定ip 地址和端口号  
3.listen 使套接字变为被动套接字  
4.accept 取出一个客户端连接，用于服务  
5.recv/send 接收和发送消息  
6.关闭套接字**

1．**import socket**

**2．**

**3．def main():**

**4． # 1、创建tcp 服务端的socket**

**5． server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)**

**6．**

**7． # 2、绑定**

**8． server\_socket.bind(('', 8888))**

**9．**

**10． # 3、listen 使套接字变为被动套接字**

**11． server\_socket.listen(128)**

**12．**

**13． # 4、如果有新的客户端来链接服务器，那么就产生一个新的套接字专门为这个客户端服务**

**14． # client\_socket 用来为这个客户端服务**

**15． # tcp\_server\_socket 就可以省下来专门等待其他新客户端的链接**

**16． client\_socket, client\_addr = server\_socket.accept()**

**17．**

**18． # 5、接收客户端发来的消息**

**19． recv\_data = client\_socket.recv(1024)**

**20． print("接收到客户端%s 的数据：%s" % (str(client\_addr), recv\_data.decode('gbk')))**

**22． # 6、回复数据给客户端**

**23． client\_socket.send("收到消息".encode('gbk'))**

**24．**

**25． # 7、关闭套接字**

**26． client\_socket.close()**

**27． server\_socket.close()**

**28．**

**29．if \_\_name\_\_ == '\_\_main\_\_':**

**30． main()**

• **当客户端需要链接服务器时，就需要使用connect 进行链接，udp 是不需要链接的而是直接发送，但是tcp 必须先链接，只有链接成功才能通信；**

**• 当一个tcp 客户端连接服务器时，服务器端会有1 个新的套接字，这个套接字用来标记这个客户端，单独为这个客户端服务；**

**• listen 后的套接字是被动套接字，用来接收新的客户端的连接请求的，而accept 返回的新套接字是标识这个新客户端的；**

**• 关闭listen 后的套接字意味着被动套接字关闭了，会导致新的客户端不能够链接服务器，但是之前已经链接成功的客户端正常通信；**

**• 关闭accept 返回的套接字意味着这个客户端已经服务完毕；**

**• 当客户端的套接字调用close 后，服务器端会recv 解阻塞，并且返回的长度为0，因此服务器可以通过返回数据的长度来区别客户端是否已经下线；同理当服务器断开tcp 连接的时候客户端，同样也会收到0 字节数据。**

### 106、怎么实现强行关闭客户端和服务器之间的连接?

在socket 通信过程中不断循环检测一个全局变量(开关标记变量)，一旦标记变量变为关闭，则调用socket 的close 方法，循环结束，从而达到关闭连接的目的。

### 107、简述TCP 和UDP 的区别以及优缺点?

TCP 是面向连接的通讯协议，通过三次握手建立连接，通讯完成时四次挥手。  
优点：TCP 在数据传递时，有确认、窗口、重传、阻塞等控制机制，能保证数据正确性，较为可靠（可靠，稳定，TCP在传输之前都会经过三次握手来建立连接，数据传输完以后会自动断开连接释放资源）。  
缺点：TCP 相对于UDP 速度慢一点，要求系统资源较多（效率低，占用资源高，易被攻击。传输数据必须先建立连接，在传输数据的时候，确认机制、重传机制、拥塞机制等都会消耗大量的时间。TCP三次握手机制也会导致TCP容易被利用，实现DOS、DDOS、CC等攻击）。

UDP 是面向无连接的通讯协议，UDP 数据包括目的端口号和源端口号信息。  
优点：UDP 速度快、操作简单、要求系统资源较少，由于通讯不需要连接，可以实现广播发送；  
缺点：UDP 传送数据前并不与对方建立连接，对接收到的数据也不发送确认信号，发送端不知道数据是否会正确接收，也不重复发送，不可靠。

### 108、简述浏览器通过WSGI 请求动态资源的过程?

1.发送http 请求动态资源给web 服务器；  
2.web 服务器收到请求后通过WSGI 调用一个属性给应用程序框架；  
3.应用程序框架通过引用WSGI 调用web 服务器的方法，设置返回的状态和头信息；  
4.调用后返回，此时web 服务器保存了刚刚设置的信息；  
5.应用程序框架查询数据库，生成动态页面的body 的信息；  
6.把生成的body 信息返回给web 服务器；  
7.web 服务器吧数据返回给浏览器；

### 109、域名解析？描述用浏览器访问www.baidu.com 的过程?

域名解析详细版在本文档，此处为简写版。

在互联网上，所有的地址都是ip地址，现阶段主要是IPv4（比如：110.110.110.110）。但是这些ip地址太难记了，所以就出现了域名（比如http://baidu.com）。  
域名解析就是将域名，转换为ip地址的这样一种行为。是把域名指向网站空间IP，让人们通过注册的域名可以方便地访问到网站一种服务。IP地址是网络上标识站点的数字地址，为方便记忆，采用域名来代替IP地址标识站点地址。域名解析就是域名到IP地址的转换过程。

1. 域名解析：  
   1、浏览器向DNS获取web服务器 www.baidu.com这个域名的 的ip地址；  
   2、建立TCP连接：浏览器与对应ip地址的服务器进行TCP链接，端口为80；  
   3、浏览器执行HTTP协议，发送GET请求，读取对应文件；  
   4、服务器接收到请求后,返回网页信息 ；  
   5.客户端浏览器将这些信息组织成用户可以查看的网页形式；

2、先要解析出baidu.com 对应的ip 地址：  
• 要先使用arp 获取默认网关的mac 地址；  
• 组织数据发送给默认网关(ip 还是dns 服务器的ip，但是mac 地址是默认网关的mac 地址)；  
• 默认网关拥有转发数据的能力，把数据转发给路由器；  
• 路由器根据自己的路由协议，来选择一个合适的较快的路径转发数据给目的网关；  
• 目的网关(dns 服务器所在的网关)，把数据转发给dns 服务器；  
• dns 服务器查询解析出baidu.com 对应的ip 地址，并原路返回请求这个域名的client得到了baidu.com 对应的ip 地址之后，会发送tcp 的3 次握手，进行连接；

• 使用http 协议发送请求数据给web 服务器；  
• web 服务器收到数据请求之后，通过查询自己的服务器得到相应的结果，原路返回给浏览器；  
• 浏览器接收到数据之后通过浏览器自己的渲染功能来显示这个网页；  
• 浏览器关闭tcp 连接，即4 次挥手结束，完成整个访问过程；

### 110、Get与Post请求的区别?

**简写版：**

1、GET是从服务器上获取数据，POST是向服务器传送数据 ；

2、GET的安全性非常低，POST安全性高；

3、在客户端，GET方式在通过URL提交数据，数据在URL中可以看到，POST方式，数据放置在HTML——HEADER内提交（GET是把参数添加到URL中传递给服务端，POST是将数据放在HTML的body中发送给服务端的，用户看不到该过程）；

4、对于GET方式，服务器端用Request.QueryString获取变量的值，对于POST方式，服务器端用Request.Form获取提交的数据；

**详细版：**

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。URL 的编码格式采用的是ASCII 编码，而不是uniclde，即是说所有的非ASCII 字符都要编码之后再传输。  
POST 请求：POST 请求会把请求的数据放置在HTTP 请求包的包体中。上面的item=bandsaw 就是实际的传输数据。  
因此，GET 请求的数据会暴露在地址栏中，而POST 请求则不会。  
2、传输数据的大小（GET的参数有长度限制不能大于2kB，POST传送的数据量较大，一般默认不限制 GET参数为ASCII编码，POST无编码限制. GET可以缓存收藏为书签，POST不能被缓存）：  
• 在HTTP 规范中，没有对URL 的长度和传输的数据大小进行限制。但是在实际开发过程中，对于GET，特定的浏览器和服务器对URL 的长度有限制。因此，在使用GET 请求时，传输数据会受到URL 长度的限制。  
• 对于POST，由于不是URL 传值，理论上是不会受限制的，但是实际上各个服务器会规定对POST提交数据大小进行限制，Apache、IIS 都有各自的配置。  
3、安全性：  
• POST 的安全性比GET 的高。这里的安全是指真正的安全，而不同于上面GET 提到的安全方法中的安全，上面提到的安全仅仅是不修改服务器的数据。比如，在进行登录操作，通过GET 请求，用户名和密码都会暴露再URL 上，因为登录页面有可能被浏览器缓存以及其他人查看浏览器的历史记录的原因，此时的用户名和密码就很容易被他人拿到了。除此之外，GET 请求提交的数据还可能会造成Cross-site request frogery 攻击。  
4、效率：GET 比POST 效率高。  
POST 请求的过程：  
1.浏览器请求tcp 连接（第一次握手）  
2.服务器答应进行tcp 连接（第二次握手）  
3.浏览器确认，并发送post 请求头（第三次握手，这个报文比较小，所以http 会在此时进行  
第一次数据发送）  
4.服务器返回100 continue 响应  
5.浏览器开始发送数据  
6.服务器返回200 ok 响应  
GET 请求的过程：  
1.浏览器请求tcp 连接（第一次握手）  
2.服务器答应进行tcp 连接（第二次握手）  
3.浏览器确认，并发送get 请求头和数据（第三次握手，这个报文比较小，所以http 会在此时  
进行第一次数据发送）  
4.服务器返回200 OK 响应

## 111、cookie 和session 的区别？他们能单独用吗？

1、储存位置：cookie 数据存放在客户的浏览器上，session 数据放在服务器上。  
2、安全性：cookie 不是很安全，别人可以分析存放在本地的cookie 并进行cookie 欺骗考虑到安全应当使用session；session较cookie更安全；session技术是要使用到cookie的，之所以出现session技术，主要是为了安全。

3、目的：Cookie 目的：跟踪会话，也可以保存用户偏好设置或者保存用户名密码等；session 目的：跟踪会话，会在一定时间内保存在服务器上。当访问增多，会比较占用服务器的性能考虑到减轻服务器性能方面，应当使用cookie。  
4、单个cookie 保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20 个cookie。

5、建议： 将登陆信息等重要信息存放为SESSION 其他信息如果需要保留，可以放在cookie 中；

Session采用的是在服务器端保持状态的方案，而Cookie采用的是在客户端保持状态的方案。但是禁用Cookie就不能得到Session。因为Session是用Session ID来确定当前对话所对应的服务器Session，而Session ID是通过Cookie来传递的，禁用Cookie相当于SessionID,也就得不到Session（Cookie是一个事件存储在客户端（浏览器），Session是一个对象存储在服务端；Session依赖于Cookie，用户每次请求服务器请求头都会携带Cookie，Cookie中保存了session标识符，服务器通过该标识符获取到跟请求对应的session对象）。

### 112、HTTP 协议响应状态码有什么用，列出你知道的HTTP 协议的状态码，然后讲出他们都表示什么意思？

通过状态码告诉客户端服务器的执行状态，以判断下一步该执行什么操作。  
常见的状态机器码有：  
1xx：100-199：指示信息–表示服务器成功接收客户端部分请求，要求客户端继续提交其余请求才能完成整个处理过程(请求已接收，继续处理)。  
2xx：200-299：成功–表示服务器成功接收请求并理解、接受已完成处理过程，常用200：响应ok成功，请求被正常处理；204：请求被受理但是没有任何资源返回。  
3xx：300-399：重定向–为完成请求，客户端需要更进一步细化请求操作。301：永久重定向；302：临时重定向，所有请求页面已经临时转移到新的url；304、307：使用缓存资源。  
4xx：400-499：客户端(请求有)错误–请求有语法错误或请求无法实现(请求包括错的顺序或不能完成)，常用400：Bad Request，客户端请求有语法错误，不能被服务器所理解)；401:Unauthorized，请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用；403:Forbidden，服务器收到请求，但是拒绝提供服务权限不够（请求的资源禁止访问）；404:Not Found，请求资源不存在（请求的资源找不到），eg：输入了错误的 URL(服务器无法找到被请求页面）；405：请求的方式不被允许。  
5xx：500-599：服务器端(出现)错误–服务器未能实现合法的请求(服务器无法完成显然有效的请求)。常用500: Internal Server Error,请求未完成，服务器（内部服务器错误）遇到不可预知的错误（情况）； 501：服务器繁忙；503：Server Unavailable，服务器当前不能处理客户端的请求，一段时间后可能恢复正常。

**python基础之socket编程：**

**一、客户端和服务器架构**

c/s架构和b/s架构

1. c:client(客户端) s:server(服务器端) 比如电脑上安装的qq,微信
2. b:browser(浏览器) s:server(服务器端) 比如，京东，天猫网页（这里不上app）
3. s(服务器)端：有固定的ip,稳定一直在运行，支持并发（多线程）
4. b/s架构也是c/s架构的一种
5. c/s架构分为：硬件c/s架构（打印机）和软件c/s架构

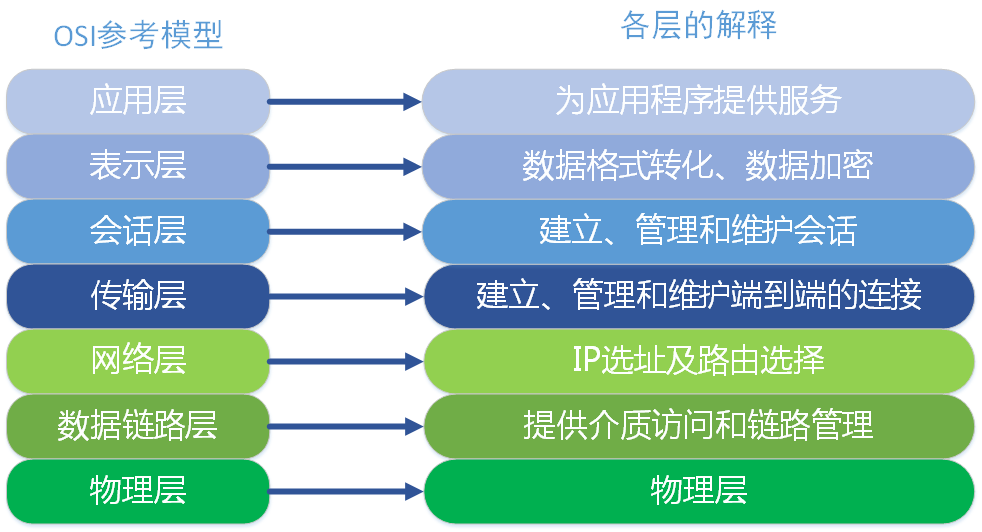
**1.1 学习网络编程的目的**

学习网络编程的目的就是为了写一个cs架构的软件

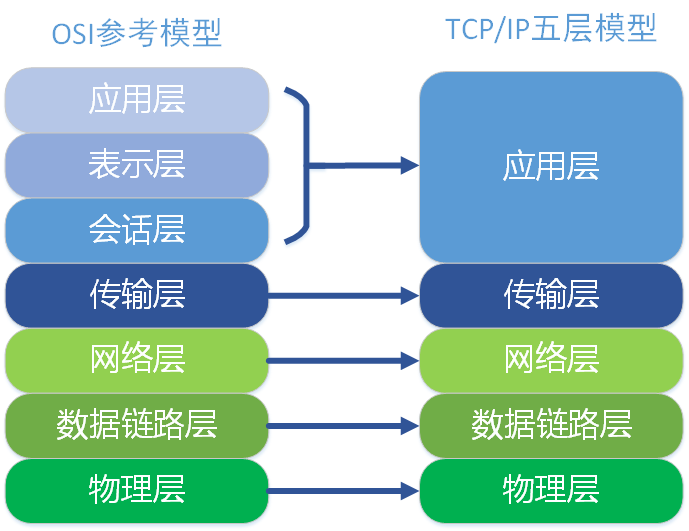
1. c(客户端)----------------网络-----------------s(服务器端)
2. 就是完成c/s架构的开发

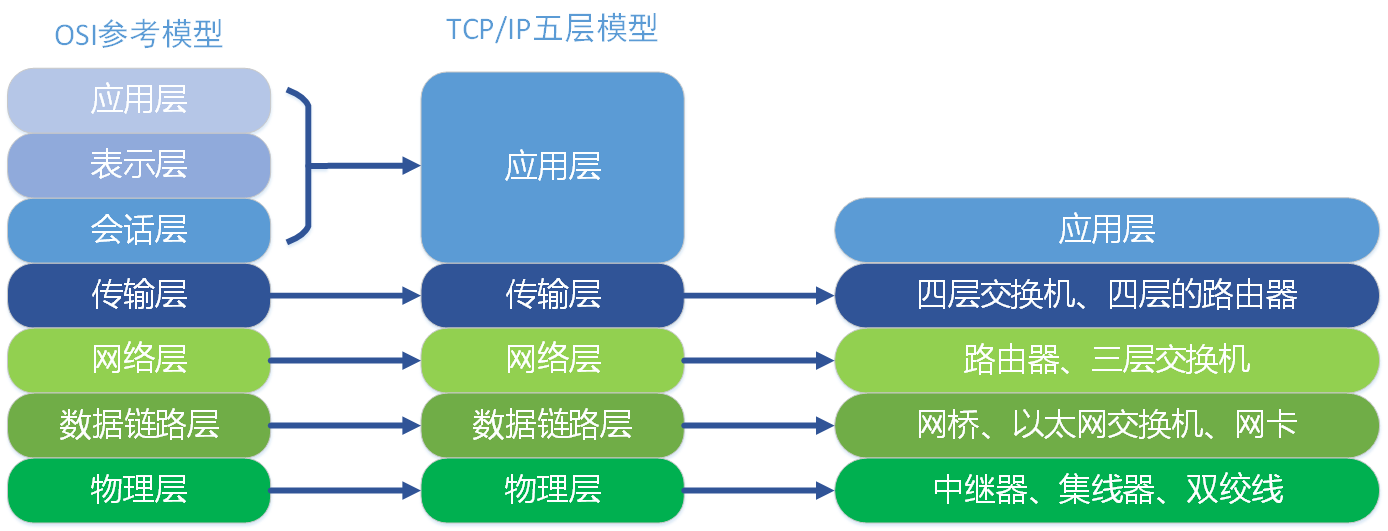
**二、osi七层协议**

1. 网络：网络连接介质+网络协议（osi七层）
2. 网络协议：osi七层协议：应用层，表示层，会话层，传输层，网络层，数据链路层，物理层



1. 五层协议：应用层，传输层，网络层，数据层，物理层



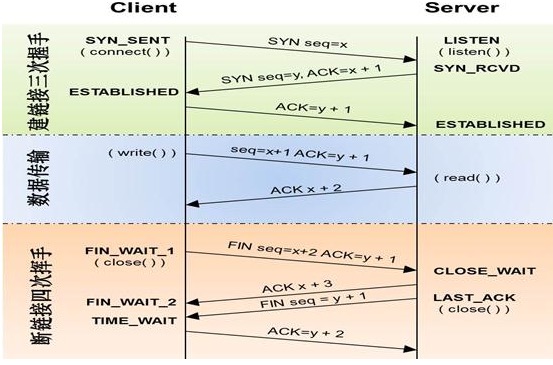
1. 四层协议

**2.1五层协议分析**

1. 物理层：010101这种电信号
2. 数据链路层：
   1. 把物理层的电信号分组，每一组叫一个数据报/数据帧。每一个数据帧分成：报头head和数据data俩部分
   2. 每一个数据报，都有报头和数据部分。
   3. 报头：固定的18个字节 ，6个字节是发送者的地址，6个字节是接收者的地址，6个字节是数据类型
   4. mac地址：发送者、接收者的地址，就是mac地址
   5. 每块网卡都有一个唯一的mac地址，12位16进制数表示（前6位是厂商编号，后六位是流水编号）
   6. 广播:同一个局域网里进行通信，容易产生广播风暴
3. 网络层
   1. ip:ipv4:32位2进制表示，点分十进制表示，从0.0.0.0到255.255.255.255，范围是有限的，不能表示出所有的网路设备，所以出现了ipv6
   2. 子网掩码：通过子网掩码和ip判断俩个ip是否处于同一个网段，通过ip地址和子网掩码做按位与运算
   3. ip地址： 172.16.10.1： 10101100.00010000.00001010.000000001
   4. 子网掩码：255.255.255.0: 11111111.11111111.11111111.000000000
   5. 按位与运算：172.16.10.0 10101100.00010000.00001010.000000000
   6. 172.16.10.0/24包含了ip地址和子网掩码
   7. ip跟mac有转换的关系
   8. 主机176.16.10.10.10/24访问172.16.10.11/24
   9. ARP协议：广播的方式发售那个数据包，获取目标主机的mac地址
   10. mac地址：查找学习mac地址和ip地址的映射表
   11. 第一次接收到就会在ip/mac映射表中添加一条数据{’172.16.10.11“：ddsadfgegsdgsdg}
   12. 任何一种协议都是由头部和内容的
4. 传输层

tcp协议：

* 1. 三次握手，四次挥手（重点），连接如何建立，发送数据如何保证可靠，断开如何断开，在建立连接过程中由状态。



* 1. 服务的如果看到大量的syn\_rcvd状态，请求同步收到状态
  2. dos和ddos攻击：拒绝服务攻击，分布式的拒绝服务攻击
  3. 端口号：端口号的范围0~65535,0-1023位系统占用端口

udp协议：

* 1. 发送，不需要响应，所以数据不可靠（比如看视频的 时候会花屏）
  2. 端口
     1. 通过ip和子网掩码唯一确定一台设备
     2. 通过ip和子网掩码维一以及端口号唯一确定一个软件。

1. 应用层

**三、上网流程分析**

1. 在浏览器输入<www.baidu.com>
2. 会取dns服务器通过域名解析成ip地址
3. 向ip+端口号这个地址发送请求，就会访问到百度的服务器

socket:在应用层和传输层之间的一个抽象层：它把TCP/TP层复杂的操作抽象为几个简单的接口供应用层调用已经实进程在网络中的通信。

**四、简单的通信**

#服务端

#导入一个socket模块

import socket

#想象成买手机打电话:socket.SOCK\_STREAM 表示建立tcp连接 ,udp连接socket.SOCK\_DGRAM

#买了个手机

soc=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

#插电话卡:绑定ip地址 传元组:(ip,端口号)

soc.bind(('192.168.11.146',8080)) #如果写本机ip，局域网外部可以访问

# soc.bind(('127.0.0.1',8080)) #如果写127.0.0.1，只能自己访问

#开机，监听,这个5 是半连接池的大小

soc.listen(5)

#等待别人给我打电话

print('xxxx')

conn,addr=soc.accept()

print('yyyy')

print(addr)

# conn 就是通路

#接收1024个字节

data=conn.recv(1024)

print('我收到客户端发的',data)

#conn.send 发送数据，数据必须是bytes格式

conn.send(b'xxxxx')

#挂断电话

conn.close()

#销毁手机

soc.close()

#客户端

import socket

#创建一个socket对象

soc=socket.socket()

#连接服务端

soc.connect(('192.168.11.146',8080))

#发送消息

soc.send(b'xxx')

data=soc.recv(1024)

print('我收到服务端回的',data)

#关闭连接

soc.close()

**进程**

**二、什么是进程**

**==进程就是资源的集合，程序执行的过程。==**

**三、进程的概念**

1. 进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

**五、进程的特征**

==**进程之间具有隔离性，进程必须等待子进程执行完毕**==

* 动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生，动态消亡的。
* 并发性：任何进程都可以同其他进程一起并发执行
* 独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位；
* 异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进
* 结构特征：进程由程序、数据和进程控制块三部分组成。

多个不同的进程可以包含相同的程序：一个程序在不同的数据集里就构成不同的进程，能得到不同的结果；但是执行过程中，程序不能发生改变。

**六、进程与程序中的区别**

程序是指令和数据的有序集合，其本身没有任何运行的含义，是一个静态的概念。而进程是程序在处理机上的一次执行过程，它是一个动态的概念。

程序可以作为一种软件资料长期存在，而进程是有一定生命期的。

程序是永久的，进程是暂时的。

注意：同一个程序执行两次，就会在操作系统中出现两个进程，所以我们可以同时运行一个软件，分别做不同的事情也不会混乱。

**一、进程的并发和并行**

**并行:** 并行是指两者同时执行，比如赛跑，两个人都在不停的往前跑；（资源够用，比如三个线程，四核的CPU ）；

**并发:** 并发是指资源有限的情况下，两者交替轮流使用资源，比如一段路(单核CPU资源)同时只能过一个人，A走一段后，让给B，B用完继续给A ，交替使用，目的是提高效率。

**二、并行和并发的区别**

并行是从微观上，也就是在一个精确的时间片刻，有不同的程序在执行，这就要求必须有多个处理器。

并发是从宏观上，在一个时间段上可以看出是同时执行的，比如一个服务器同时处理多个session。

**开启子进程的方式1**

from multiprocessing import Process

import time

def task():

print('进程 start')

time.sleep(2)

print('进程 end')

if \_\_name\_\_ == '\_\_main\_\_':#当开启子进程的时候，执行main内的代码

# 不会出现像递归一样的方式，一直创建子进程，在main内只会创建一个子进程

p = Process(target=task)#开子进程 申请新的内存空间 把父进程的所有代码完整拷贝一份过去

p.start()#告诉操作系统要开启子进程

time.sleep(3)

print('主进程')

###############

进程 start

进程 end

主进程

**方式1开启多个子进程**

from multiprocessing import Process

import time

def task(x):

print(f'子进程{x}-————>start')

time.sleep(2)

print(f'进程{x}-————>end')

if \_\_name\_\_ == '\_\_main\_\_':#当开启子进程的时候，执行main内的代码

# 不会出现像递归一样的方式，一直创建子进程，在main内只会创建一个子进程

p1 = Process(target=task,args=('ocean',))#开子进程 申请新的内存空间 把父进程的所有代码完整拷贝一份过去

p2 = Process(target=task,args=('sky',))

p1.start()

p2.start()

time.sleep(2)

print('主进程')

#################

子进程ocean-————>start

子进程sky-————>start

主进程

进程ocean-————>end

进程sky-————>end

**开启子进程的方式2**

from multiprocessing import Process

import time

class Test(Process):

def \_\_init\_\_(self,sex):

super().\_\_init\_\_()

self.sex = sex

def run(self):

print(f'子进程的性别是{self.sex}------>start')

time.sleep(1)

print(f'子进程end')

if \_\_name\_\_ == '\_\_main\_\_':#当开启子进程的时候，执行main内的代码

# 不会出现像递归一样的方式，一直创建子进程，在main内只会创建一个子进程

p = Test('女')#开子进程 申请新的内存空间 把父进程的所有代码完整拷贝一份过去

p.start()

print('主进程')

##############################

主进程

子进程的性别是女------>start

子进程end

**验证隔离性（内存空间隔离）**

from multiprocessing import Process  
import time  
x = 0  
def task():  
   global x  
   x = 100  
   print('子进程的x修改为{}'.format(x))  
if \_\_name\_\_ == '\_\_main\_\_':  
   p = Process(target=task)  
   p.start()  
   time.sleep(3)  
   print(x)  
#################  
子进程的x修改为100  
0

**僵尸进程和孤儿进程**

1. 僵尸进程：父进程的子进程结束的时候父进程没有wait（）情况下子进程会变成僵尸进程。父进程等着所有的子进程结束才会结束。
2. 孤儿进程（无害）一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
3. 有害的情况
   1. 情况1 无害父进程等着子进程都死，回收僵尸进程。
      1. 情况2 无害父进程死了，子进程活着，都要被init进程接管并且回收。
      2. 情况3父进程一直不死，造成了大量僵尸进程。占用了大量的pid号pid号是有限的。
      3. 解决方案：最直接的办法就是杀死父进程 。

**守护进程**

会随着主进程的结束而结束。

主进程创建守护进程

　　其一：守护进程会在主进程代码执行结束后就终止

　　其二：守护进程内无法再开启子进程,否则抛出异常：

**线程**

**一、线程**

1. 初识线程 在传统操作系统种，每个进程有一个地址空间，而且默认就有一个控制线程，cpu真正的执行单位是线程。 在工厂中，每一个车间都有房子，而且每个车间默认就有一个流水线。
2. 做一个比较：相当于 操作系统------->工厂 进程------->车间 线程------->流水线 cpu------->电源
3. 线程：cpu最小的执行单位 进程：资源集合/资源单位 线程运行 ------->代码运行 进程运行------->各种资源 + 线程
4. 右键运行会发生的事情：申请内存空间，先将解释器代码丢进去并且把python代码丢进去（进程做的）（内存运行的时候不识别python文件代码，只有解释器才能识别python文件代码，所有先把解释器放入内存），运行代码（线程）
5. 进程和线程的区别： 过程描述的区别 线程------->单指代码的执行过程 进程------->资源的申请与销毁的过程
6. 进程内存空间彼此隔离（**内存共享or隔离**）

多个进程内存空间彼此隔离。

同一进程下的多个线程共享该进程内的数据。同一个进程下的线程共享资源

1. 进程和线程的创造速度 进程需要申请资源开辟空间------->慢 只是告诉操作系统一个执行方案------->快
2. **1 进程的启动和销毁（空间开始和释放）**

只要一说一个进程启动了，那你知道了一个事情，是关于这个程序的一块空间造好了。

只要一说进程销毁了，那你也知道了一个事情，是关于这个程序的空间释放了。

**2 线程的启动和销毁（进程里的一段代码运行起来了，进程里的一段代码运行结束）**

1. **进程之间是竞争关系，线程之间是协作关系？（了解）**

车间直接是竞争/抢电源的关系，竞争（不同的进程直接是竞争关系，是不同的程序员写的程序运行的，迅雷抢占其他进程的网速，360把其他进程当做病毒干死） 一个车间的不同流水线式协同工作的关系（同一个进程的线程之间是合作关系，是同一个程序写的程序内开启动，迅雷内的线程是合作关系，不会自己干自己）

**多线程**

**1.3解决死锁方法（递归锁）**

解决方法，递归锁，在Python中为了支持在同一线程中多次请求同一资源，python提供了可重入锁RLock。

**二、信号量Semaphore**

同进程的一样

**实现：定制了锁的个数，也就意味着最多有几个线程可以抢到锁头**

Semaphore管理一个内置的计数器， 每当调用acquire()时内置计数器+1； 调用release() 时内置计数器-1； 计数器不能小于0；当计数器为0时，acquire()将阻塞线程直到其他线程调用release()。

实例：(同时只有5个线程可以获得semaphore,即可以限制最大连接数为5)：

**3.1 什么是GIL**

1. 在cpython解释器中有一把GIL锁（全局解释锁），GIL锁本质是一把互斥锁；
2. 导致了同一个进程下，同一时间只能运行一个线程，无法利用多核优势；
3. 同一个进程下的多个线程只能实现并发不能实现并行。

**3.2为什么要有GIL**

因为cpython自带的垃圾回收机制不是线程安全的，所以要有GIL锁。导致了同一个进程下，同一时间只能由一个线程，无法利用多核优势。

分析：

1. 我们四个任务要处理，处理方式并发
2. 解决方案：
   1. 开启四个进程
   2. 一个进程下开启四个线程

**多进程**

**七、进程池和线程池**

**7.1进程池和线程池的意义**

1. 进程池和线程池
   1. 进程池和线程池的功能是限制进程数或者线程数。
2. 什么时候限制？

当并发的任务数量远远大于计算机所能承受的范围的时候，即无法一次性开启过多的任务数量，就应该考虑去限制进程数或者线程数，从而保证服务器不会崩溃。（荡机）

**协程**

1. python的线程用的是操作系统原生的线程

**8.1什么是协程**

1. 协程：单线程下实现并发
2. 并发：切换+保存状态
3. 多线程：操作系统帮助实现，如果遇到io切换，执行时间过程也会切换，实现一个雨露均沾的效果

* 什么样的协程是有有意义的？
  1. 遇到io切换的时候是有意义的
  2. 具体：协程的概念本质是程序猿抽象出来的，操作系统根本不知道协程的存在，也就是说来了一个线程我自己遇到io，我自己线程内部直接切到自己的另一个线程任务上，操作系统根本不知道这个过程。
* 优点：
  1. 自己控制切换要比操作系统控制切换的快的多
* 缺点：
  1. 对比多线程

自己要检测所有的io,只要有一个阻塞整体都会跟着阻塞

* 1. 对比多进程

无法利用多核优势

* 为什么要有协程？

自己控制切换的速度比操作系统控制切换的快，降低了单个线程的io时间。

### 113、请简单说一下三次握手和四次挥手？

**三次握手过程：**  
**第一次握手：**客户端的应用进程主动打开，并向服务端发送一个带有SYN 标志，以及随机生成的序号100(0 字节)的请求报文段。其首部中：SYN=1,seq=x（客户端通过向服务器端发送一个SYN来创建一个主动打开，作为三次握手的一部分。客户端把这段连接的序号设定为随机数 A。）。

**第二次握手：**服务器应用进程被动打开，若服务端收到报文后同意客户端的请求，则返回一个报文(SYN200(0 字节)，ACk1001(字节+1))给客户端，其报文首部中：SYN=1,ACK=1,ack=x+1,seq=y（服务器端应当为一个合法的SYN回送一个SYN/ACK。ACK 的确认码应为 A+1，SYN/ACK 包本身又有一个随机序号 B。）。  
**第三次握手：**客户端收到确认报文之后，通知上层应用进程连接已建立，客户端再次向服务器发送带有ACk 标志201(字节+)序号的确认报文给服务端，其报文首部：ACK=1,ack=y+1。当服务器收到客户端的确认报文之后，也通知其上层应用进程连接已建立（最后，客户端再发送一个ACK。当服务端受到这个ACK的时候，就完成了三路握手，并进入了连接创建状态。此时包序号被设定为收到的确认号 A+1，而响应则为 B+1。）。

至此三次握手过程结束，客户端开始向服务端发送数据。

1 客户端向服务端发起请求：我想给你通信，你准备好了么？  
2 服务端收到请求后回应客户端：I'ok，你准备好了么？  
3 客户端礼貌的再次回一下客户端：准备就绪，咱们开始通信吧！  
整个过程跟打电话的过程一模一样:1 喂，你在吗2 在，我说的你听得到不3 恩，听得到(接下来请开始你的表演)；  
补充：SYN：请求询问，ACk：回复，回应。

**四次挥手过程：**

注意: 中断连接端可以是客户端，也可以是服务器端. 下面仅以客户端断开连接举例, 反之亦然；  
由于TCP 连接是可以双向通信的（全双工），因此每个方向都必须单独进行关闭（这句话才是精辟，后面四个挥手过程都是其具体实现的语言描述）；  
四次挥手过程，客户端和服务端都可以先开始断开连接；  
**第一次挥手：**数据传输结束以后，客户端的应用进程发出带有fin 标识的连接释放报文段给服务端，并停止发送数据，请求通信关闭，其首部：FIN=1,seq=u（客户端发送一个数据分段, 其中的 FIN 标记设置为1. 客户端进入 FIN-WAIT 状态. 该状态完成后进入第二次挥手，此后除第四步挥手外客户端只接收数据, 不再发送数据）；  
**第二次挥手：**服务端收到连接释放报文段信息后，发出回复ACK 确认报文答应关闭客户端通信(连接)请求，其首部：ack=u+1,seq=v。此时本次连接就进入了半关闭状态，客户端不再向服务器发送数据，而服务器端仍会继续发送。（服务器接收到带有 FIN = 1 的数据分段, 发送带有 ACK = 1 的剩余数据分段, 确认收到客户端发来的 FIN 信息）；  
**第三次挥手：**这个阶段若服务器已经没有要向客户端发送的数据，其应用进程就通知服务器发出最后一个带有fin 标识的报文给客户端，也请求关闭通信，释放TCP连接。报文的首部应为：FIN=1,ACK=1,seq=w,ack=u+1（服务器等到所有数据传输结束, 向客户端发送一个带有 FIN = 1 的数据分段, 并进入 CLOSE-WAIT 状态, 等待客户端发来带有 ACK = 1 的确认报文）；  
**第四次挥手：**客户端收到连接释放报文段之后，客户端必须发出确认回应ack 给服务端：ACK=1,seq=u+1,ack=w+1，答应关闭服务端的通信(连接)请求。 再经过2MSL(最长报文端寿命)后，本次TCP连接真正结束，通信双方完成了他们的告别。（客户端收到服务器发来带有 FIN = 1 的报文, 返回 ACK = 1 的报文确认, 为了防止服务器端未收到需要重发, 进入 TIME-WAIT 状态. 服务器接收到报文后关闭连接. 客户端等待 2MSL 后未收到回复, 则认为服务器成功关闭, 客户端关闭连接）；

图解: <http://blog.csdn.net/whuslei/article/details/6667471>

### 114、说一下什么是tcp 的2MSL？

主动发送fin 关闭的一方，在4 次挥手最后一次要等待一段时间我们称这段时间为2MSL；  
TIME\_WAIT 状态的存在有两个理由：  
1.让4 次挥手关闭流程更加可靠；  
2.防止丢包后对后续新建的正常连接的传输造成破坏；

### 115、为什么客户端在TIME-WAIT 状态必须等待2MSL 的时间？

1、为了保证客户端发送的最后一个ACK 报文段能够达到服务器。这个ACK 报文段可能丢失，因而使处在LAST-ACK 状态的服务器收不到确认。服务器会超时重传FIN+ACK 报文段，客户端就能在2MSL 时间内收到这个重传的FIN+ACK 报文段，接着客户端重传一次确认，重启计时器。最好，客户端和服务器都正常进入到CLOSED 状态。如果客户端在TIME-WAIT 状态不等待一段时间，而是再发送完ACK 报文后立即释放连接，那么就无法收到服务器重传的FIN+ACK 报文段，因而也不会再发送一次确认报文。这样，服务器就无法按照正常步骤进入CLOSED 状态。  
2、防止已失效的连接请求报文段出现在本连接中。客户端在发送完最后一个ACK 确认报文段后，再经过时间2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段。

### 116、说说HTTP 和HTTPS 区别？

HTTP 协议传输的数据都是未加密的，也就是明文的，因此使用HTTP 协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP 协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS 协议是由SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比http 协议安全。  
HTTPS 和HTTP 的区别主要如下：  
1、https 协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。  
2、http 是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl 加密传输协议(HTTPS握手,对称加密,非对称加密,TLS/SSL,RSA)。  
3、http 和https 使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http 的连接很简单，是无状态的；HTTPS 协议是由SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比http 协议安全。

### 118、使用Socket 套接字需要传入哪些参数？

Address Family 和Type，分别表示套接字应用场景和类型。  
family 的值可以是AF\_UNIX(Unix 域，用于同一台机器上的进程间通讯)，也可以是AF\_INET（对于IPV4 协议的TCP 和UDP），至于type 参数，SOCK\_STREAM（流套接字）或者SOCK\_DGRAM（数据报文套接字）,SOCK\_RAW（raw 套接字）。

### 121、url的形式？

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

scheme：协议(例如：http， https， ftp)；

host：服务器的IP 地址或者域名；

port：服务器的端口（如果是走协议默认端口，80 or 443）；

path：访问资源的路径；

query-string：参数，发送给http 服务器的数据；

anchor：锚（跳转到网页的指定锚点位置）；

<http://localhost:4000/file/part01/1.2.html>；

## 122、CORS 和 CSRF的区别？

什么是CORS？ CORS是一个W3C标准,全称是“跨域资源共享"(Cross-origin resoure sharing). 它允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而客服了AJAX只能同源使用的限制。

什么是CSRF？ CSRF主流防御方式是在后端生成表单的时候生成一串随机token,内置到表单里成为一个字段，同时，将此串token置入session中。每次表单提交到后端时都会检查这两个值是否一致，以此来判断此次表单提交是否是可信的，提交过一次之后，如果这个页面没有生成CSRF token,那么token将会被清空,如果有新的需求，那么token会被更新。 攻击者可以伪造POST表单提交，但是他没有后端生成的内置于表单的token，session中没有token都无济于事。

### 122、CSRF和XSS的区别？

1、CSRF(Cross-site request forgery)跨站请求伪造；XSS(Cross Site Scripting)跨站脚本攻击；2、CSRF 重点在请求，XSS 重点在脚本。

## 123、Session,Cookie,JWT的理解（cookie与session区别，本文档已写）

为什么要使用会话管理？众所周知，HTTP协议是一个无状态的协议，也就是说每个请求都是一个独立的请求，请求与请求之间并无关系。但在实际的应用场景，这种方式并不能满足我们的需求。举个大家都喜欢用的例子，把商品加入购物车，单独考虑这个请求，服务端并不知道这个商品是谁的，应该加入谁的购物车？因此这个请求的上下文环境实际上应该包含用户的相关信息，在每次用户发出请求时把这一小部分额外信息，也做为请求的一部分，这样服务端就可以根据上下文中的信息，针对具体的用户进行操作。所以这几种技术的出现都是对HTTP协议的一个补充，使得我们可以用HTTP协议+状态管理构建一个的面向用户的WEB应用。

**Session 和Cookie的区别？**这里我想先谈谈session与cookies,因为这两个技术是做为开发最为常见的。

**那么session与cookies的区别是什么？**个人认为session与cookies最核心区别在于额外信息由谁来维护。利用cookies来实现会话管理时，用户的相关信息或者其他我们想要保持在每个请求中的信息，都是放在cookies中,而cookies是由客户端来保存，每当客户端发出新请求时，就会稍带上cookies,服务端会根据其中的信息进行操作。 当利用session来进行会话管理时，客户端实际上只存了一个由服务端发送的session\_id,而由这个session\_id,可以在服务端还原出所需要的所有状态信息，从这里可以看出这部分信息是由服务端来维护的。

除此以外，session与cookies都有一些自己的缺点： cookies的安全性不好，攻击者可以通过获取本地cookies进行欺骗或者利用cookies进行CSRF攻击。使用cookies时,在多个域名下，会存在跨域问题。 session 在一定的时间里，需要存放在服务端，因此当拥有大量用户时，也会大幅度降低服务端的性能，当有多台机器时，如何共享session也会是一个问题.(redis集群)也就是说，用户第一个访问的时候是服务器A，而第二个请求被转发给了服务器B，那服务器B如何得知其状态。实际上，session与cookies是有联系的，比如我们可以把session\_id存放在cookies中的。

**JWT是如何工作的：**首先用户发出登录请求，服务端根据用户的登录请求进行匹配，如果匹配成功，将相关的信息放入payload中，利用算法，加上服务端的密钥生成token，这里需要注意的是secret\_key很重要，如果这个泄露的话，客户端就可以随机篡改发送的额外信息，它是信息完整性的保证。生成token后服务端将其返回给客户端，客户端可以在下次请求时，将token一起交给服务端，一般是说我们可以将其放在Authorization首部中，这样也就可以避免跨域问题。

**jwt加密**

权限，认证，频率

  def validate(self, attrs):  
      user = self.\_get\_user(\*\*attrs)  
      # 签发token，并将user和token存放到序列化对象中  
      payload = jwt\_payload\_handler(user)  
      token = jwt\_encode\_handler(payload)  
      self.user = user  
      self.token = token  
      return attrs

### 124、jwt是干嘛的，怎么实现的？

答：用户认证，加密，三个部分加密（最重要），校验。

一、优势简介  
JSON Web Tokens简称jwt，是rest接口的一种安全策略。本身有很多的优势：

解决跨域问题：这种基于Token的访问策略可以克服cookies的跨域问题。  
服务端无状态可以横向扩展，Token可完成认证，无需存储Session。  
系统解耦，Token携带所有的用户信息，无需绑定一个特定的认证方案，只需要知道加密的方法和密钥就可以进行加密解密，有利于解耦。  
防止跨站点脚本攻击，没有cookie技术，无需考虑跨站请求的安全问题。  
二、原理简介  
JSON Web Tokens的格式组成，jwt是一段被base64编码过的字符序列，用点号分隔，一共由三部分组成，头部header，消息体playload和签名sign。

要如何用php实现JWT认证，那我们首先就来认识一下什么是JWT。  
什么是JWTJWT(json web token)是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准。  
JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源。比如用在用户登录上。  
JWT定义了一种用于简洁，自包含的用于通信双方之间以 JSON 对象的形式安全传递信息的方法。JWT 可以使用 HMAC 算法或者是 RSA 的公钥密钥对进行签名。

JWT有两个特点：

自包含(Self-contained)：负载中包含了所有用户所需要的信息，避免了多次查询数据库

简洁(Compact)：可以通过URL, POST 参数或者在 HTTP header 发送，因为数据量小，传输速度快

JWT组成

JWT由header，payload，signature三个部分，下面我们用官网的实例先来讲解一个这三个部分的用法。

header部分：

jwt的头部承载两部分信息：

声明类型，这里是jwt

声明加密的算法 通常直接使用 HMAC SHA256

完整的头部就像下面这样的JSON：

{  
 "alg": "HS256",  
 "typ": "JWT"  
}

对应base64UrlEncode编码为：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

说明：该字段为json格式。alg字段指定了生成signature的算法，默认值为HS256，typ默认值为JWT

payload部分：

载荷就是存放有效信息的地方。

标准中注册的声明 (建议但不强制使用) ：

iss: jwt签发者

sub: jwt所面向的用户

aud: 接收jwt的一方

exp: jwt的过期时间，这个过期时间必须要大于签发时间

nbf: 定义在什么时间之前，该jwt都是不可用的

iat: jwt的签发时间

jti: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。

{  
 "sub": "1234567890",  
 "name": "John Doe",  
 "iat": 1516239022  
}

对应base64UrlEncode编码为：

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

说明：该字段为json格式，表明用户身份的数据，可以自己自定义字段，很灵活。sub 面向的用户，name 姓名 ,iat 签发时间。例如可自定义示例如下：

{  
  "iss": "admin",     //该JWT的签发者  
  "iat": 1535967430,    //签发时间  
  "exp": 1535974630,    //过期时间  
  "nbf": 1535967430,     //该时间之前不接收处理该Token  
  "sub": "www.admin.com",  //面向的用户  
  "jti": "9f10e796726e332cec401c569969e13e"  //该Token唯一标识  
}

signature部分：

jwt的第三部分是一个签证信息，这个签证信息由三部分组成：

header (base64后的)

payload (base64后的)

secret

HMACSHA256(  
 base64UrlEncode(header) + "." +  
 base64UrlEncode(payload),  
 123456  
)

对应的签名为：

keH6T3x1z7mmhKL1T3r9sQdAxxdzB6siemGMr\_6ZOwU

最终得到的JWT的json为(header.payload.signature)：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.keH6T3x1z7mmhKL1T3r9sQdAxxdzB6siemGMr\_6ZOwU

说明：对header和payload进行base64UrlEncode编码后进行拼接。通过key（这里是123456）进行HS256算法签名。

JWT的使用流程：

初次登录：用户初次登录，输入用户名密码

密码验证：服务器从数据库取出用户名和密码进行验证

生成JWT：服务器端验证通过，根据从数据库返回的信息，以及预设规则，生成JWT

返还JWT：服务器的HTTP RESPONSE中将JWT返还

带JWT的请求：以后客户端发起请求，HTTP REQUEST

HEADER中的Authorizatio字段都要有值，为JWT

服务器验证JWT

 起源

说起JWT，我们应该来谈一谈基于token的认证和传统的session认证的区别。

传统的session认证

我们知道，http协议本身是一种无状态的协议，而这就意味着如果用户向我们的应用提供了用户名和密码来进行用户认证，那么下一次请求时，用户还要再一次进行用户认证才行，因为根据http协议，我们并不能知道是哪个用户发出的请求，所以为了让我们的应用能识别是哪个用户发出的请求，我们只能在服务器存储一份用户登录的信息，这份登录信息会在响应时传递给浏览器，告诉其保存为cookie,以便下次请求时发送给我们的应用，这样我们的应用就能识别请求来自哪个用户了,这就是传统的基于session认证。

但是这种基于session的认证使应用本身很难得到扩展，随着不同客户端用户的增加，独立的服务器已无法承载更多的用户，而这时候基于session认证应用的问题就会暴露出来.

基于session认证所显露的问题

Session: 每个用户经过我们的应用认证之后，我们的应用都要在服务端做一次记录，以方便用户下次请求的鉴别，通常而言session都是保存在内存中，而随着认证用户的增多，服务端的开销会明显增大。

扩展性: 用户认证之后，服务端做认证记录，如果认证的记录被保存在内存中的话，这意味着用户下次请求还必须要请求在这台服务器上,这样才能拿到授权的资源，这样在分布式的应用上，相应的限制了负载均衡器的能力。这也意味着限制了应用的扩展能力。

CSRF: 因为是基于cookie来进行用户识别的, cookie如果被截获，用户就会很容易受到跨站请求伪造的攻击。

基于token的鉴权机制

基于token的鉴权机制类似于http协议也是无状态的，它不需要在服务端去保留用户的认证信息或者会话信息。这就意味着基于token认证机制的应用不需要去考虑用户在哪一台服务器登录了，这就为应用的扩展提供了便利。

流程上是这样的：

用户使用用户名密码来请求服务器

服务器进行验证用户的信息

服务器通过验证发送给用户一个token

客户端存储token，并在每次请求时附送上这个token值

服务端验证token值，并返回数据

这个token必须要在每次请求时传递给服务端，它应该保存在请求头里， 另外，服务端要支持CORS(跨来源资源共享)策略，一般我们在服务端这么做就可以了Access-Control-Allow-Origin: \*。

那么我们现在回到JWT的主题上。

JWT长什么样？

JWT是由三段信息构成的，将这三段信息文本用.链接一起就构成了Jwt字符串。就像这样:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

JWT的构成

第一部分我们称它为头部（header),第二部分我们称其为载荷（payload, 类似于飞机上承载的物品)，第三部分是签证（signature).

header

jwt的头部承载两部分信息：

声明类型，这里是jwt

声明加密的算法 通常直接使用 HMAC SHA256

完整的头部就像下面这样的JSON：

{

'typ': 'JWT',

'alg': 'HS256'

}

然后将头部进行base64加密（该加密是可以对称解密的),构成了第一部分.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

playload

载荷就是存放有效信息的地方。这个名字像是特指飞机上承载的货品，这些有效信息包含三个部分

标准中注册的声明

公共的声明

私有的声明

标准中注册的声明 (建议但不强制使用) ：

iss: jwt签发者

sub: jwt所面向的用户

aud: 接收jwt的一方

exp: jwt的过期时间，这个过期时间必须要大于签发时间

nbf: 定义在什么时间之前，该jwt都是不可用的.

iat: jwt的签发时间

jti: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。

公共的声明 ：  
公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息.但不建议添加敏感信息，因为该部分在客户端可解密.

私有的声明 ：  
私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为base64是对称解密的，意味着该部分信息可以归类为明文信息。

定义一个payload:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

然后将其进行base64加密，得到Jwt的第二部分。

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9

signature

jwt的第三部分是一个签证信息，这个签证信息由三部分组成：

header (base64后的)

payload (base64后的)

secret

这个部分需要base64加密后的header和base64加密后的payload使用.连接组成的字符串，然后通过header中声明的加密方式进行加盐secret组合加密，然后就构成了jwt的第三部分。

// javascript

var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);

var signature = HMACSHA256(encodedString, 'secret'); // TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

将这三部分用.连接成一个完整的字符串,构成了最终的jwt:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

注意：secret是保存在服务器端的，jwt的签发生成也是在服务器端的，secret就是用来进行jwt的签发和jwt的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。一旦客户端得知这个secret, 那就意味着客户端是可以自我签发jwt了。

如何应用

一般是在请求头里加入Authorization，并加上Bearer标注：

fetch('api/user/1', {

headers: {

'Authorization': 'Bearer ' + token

}

})

服务端会验证token，如果验证通过就会返回相应的资源。整个流程就是这样的:

jwt-diagram

总结

优点

因为json的通用性，所以JWT是可以进行跨语言支持的，像JAVA,JavaScript,NodeJS,PHP等很多语言都可以使用。

因为有了payload部分，所以JWT可以在自身存储一些其他业务逻辑所必要的非敏感信息。

便于传输，jwt的构成非常简单，字节占用很小，所以它是非常便于传输的。

它不需要在服务端保存会话信息, 所以它易于应用的扩展

安全相关

不应该在jwt的payload部分存放敏感信息，因为该部分是客户端可解密的部分。

保护好secret私钥，该私钥非常重要。

如果可以，请使用https协议

### 124、什么是cookie,token和session?它们之间有什么关系？

**token令牌，是用户身份的验证方式。最简单的token组成:uid(用户唯一的身份标识)、time（当前时间的时间戳）、sign（签名）。**

**对Token认证的五点认识：**

**1、一个Token就是一些信息的集合；在Token中包含足够多的信息，以便在后续请求中减少查询数据库的几率；**

**2、服务端需要对cookie和HTTP Authrorization Header进行Token信息的检查；**

**3、基于上一点，你可以用一套token认证代码来面对浏览器类客户端和非浏览器类客户端；**

**4、因为token是被签名的，所以我们可以认为一个可以解码认证通过的token是由我们系统发放的，其中带的信息是合法有效的；**

**5、session会话，代表服务器与浏览器的一次会话过程，这个过程是连续的，也可以时断时续。cookie中存放着一个sessionID，请求时会发送这个ID；session因为请求（request对象）而产生；session是一个容器，可以存放会话过程中的任何对象；session的创建与使用总是在服务端，浏览器从来都没有得到过session对象；session是一种http存储机制，目的是为武装的http提供持久机制。**

**cookie储存在用户本地终端上的数据，服务器生成，发送给浏览器，下次请求统一网站给服务器。**

**cookie与session区别cookie数据存放在客户端上，session数据放在服务器上；cookie不是很安全，且保存数据有限；session一定时间内保存在服务器上,当访问增多，占用服务器性能。**

**session与token作为身份认证，token安全行比session好；Session 认证只是简单的把User 信息存储到Session 里，因为SID 的不可预测性，暂且认为是安全的。这是一种认证手段。 而Token ，如果指的是OAuth Token 或类似的机制的话，提供的是认证和授权，认证是针对用户，授权是针对App 。其目的是让 某App有权利访问某用户的信息。**

**token与cookie：Cookie是不允许垮域访问的，但是token是支持的， 前提是传输的用户认证信息通过HTTP头传输； token就是令牌，比如你授权（登录）一个程序时，他就是个依据，判断你是否已经授权该软件；cookie就是写在客户端的一个txt文件，里面包括你登录信息之类的，这样你下次在登录某个网站，就会自动调用cookie自动登录用户名；session和cookie差不多，只是session是写在服务器端的文件，也需要在客户端写入cookie文件，但是文件里是你的浏览器编号.Session的状态是存储在服务器端，客户端只有session id；而Token的状态是存储在客户端。**

## 125、nginx,tomcat,apach都是什么？

**Nginx**（engine x)是一个高性能的HTTP和反向代理服务器，也是一个IMAP/POP3/SMTP服务器，工作在OSI七层，负载的实现方式：轮询，IP\_HASH,fair,session\_sticky；

**Apache HTTP Serve**r是一个模块化的服务器，源于NCSAhttpd服务器；

**Tomcat** 服务器是一个免费的开放源代码的Web应用服务器，属于轻量级应用服务器，是开发和调试JSP程序的首选。

## 126、简述QQ登陆过程？

qq登录，在我们的项目中分为了三个接口，

第一个接口是请求qq服务器返回一个qq登录的界面;

第二个接口是通过扫码或账号登陆进行验证，qq服务器返回给浏览器一个code和state,利用这个code通过本地服务器去向qq服务器获取access\_token覆返回给本地服务器，凭借access\_token再向qq服务器获取用户的openid(openid用户的唯一标识) ；

第三个接口是判断用户是否是第一次qq登录，如果不是的话直接登录返回的jwt-token给用户，对没有绑定过本网站的用户，对openid进行加密生成token进行绑定。

## 128、项目中日志的作用？

一、日志相关概念 ：

1.日志是一种可以追踪某些软件运行时所发生事件的方法；

2.软件开发人员可以向他们的代码中调用日志记录相关的方法来表明发生了某些事情 ；

3.一个事件可以用一个包含可选变量数据的消息来描述 ；

4.此外，事件也有重要性的概念，这个重要性也可以被成为严重性级别(level)

二、日志的作用：

1.通过log的分析，可以方便用户了解系统或软件、应用的运行情况;

2.如果你的应用log足够丰富，可以分析以往用户的操作行为、类型喜好，地域分布或其他更多信息;

3.如果一个应用的log同时也分了多个级别，那么可以很轻易地分析得到该应用的健康状况，及时发现问题并快速定位、解决问题，补救损失。

4.简单来讲就是我们通过记录和分析日志可以了解一个系统或软件程序运行情况是否正常，也可以在应用程序出现故障时快速定位问题。不仅在开发中，在运维中日志也很重要，日志的作用也可以简单。

总结为以下几点： 1.程序调试； 2.了解软件程序运行情况，是否正常； 3,软件程序运行故障分析与问题定位；4,如果应用的日志信息足够详细和丰富，还可以用来做用户行为分析。

## 129、谈一下你对uWSGI和nginx的理解？

wsgi是一种通用的接口标准或者接口协议，实现了python web程序与服务器之间交互的通用性。

uwsgi：同WSGI一样是一种通信协议。

uwsgi协议是一个'uWSGI服务器'自有的协议，它用于定义传输信息的类型，'uWSGI'是实现了uwsgi和WSGI两种协议的Web服务器，负责响应python的web请求。

Nginx中HttpUwsgiModule的作用是与uWSGI服务器进行交换。

WSGI是一种通信协议，WSGI不是一种Web服务器网关接口。它是一个Web服务器（如nginx，uWSGI等服务器）与web应用（如用Flask框架写的程序）通信的一种规范。

要注意WSGI/uwsgi/uWSGI这三个概念的区分。

uwsgi其实是一种线路协议而不是通信协议，在此常用于在uWSGI服务器与其他网络服务器的数据通信。

uWSGI是实现WSGI通信协议、uwsgi线路协议、http协议的Web服务器。

nginx 是一个开源的高性能的HTTP服务器和反向代理：

1.作为web服务器，它处理静态文件和索引文件效果非常高 ；

2.它的设计非常注重效率，最大支持5万个并发连接，但只占用很少的内存空间；

3.稳定性高，配置简洁；

4.强大的反向代理和负载均衡功能，平衡集群中各个服务器的负载压力应用。

## 130、说说nginx和uWISG 服务器之间如何配合工作的？

首先浏览器发起http请求到nginx服务器，Nginx根据接收到请求包，进行url分析，判断访问的资源类型，如果是静态资源，直接读取静态资源返回给浏览器，如果请求的是动态资源就转交给uwsgi服务器，uwsgi服务器根据自身的uwsgi和WSGI协议，找到对应的Django框架，Django框架下的应用进行逻辑处理后，将返回值发送到uwsgi服务器，然后uwsgi服务器再返回给nginx，最后nginx将返回值返回给浏览器进行渲染显示给户。 如果可以，画图讲解效果更佳，可以将下面的图画给面试官。



**131、python的底层网络交互模块有哪些？**

socket，urllib，,urllib3、requests，grab，pycurl；

**132、简述OSI七层协议？IP ，TCP/UDP ，HTTP ，RTSP ，FTP 分别在哪层？**

为了实现计算机系统的互连，OSI参考模型把整个网络的通信功能划分为7个层次，同时也定义了层次之间的相互关系以及各层所包括的服务及每层的功能。

OSI的七层由低到高依次为：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。

下三层（物理层、数据链路层、网络层）面向数据通信，而上三层（会话层、表示层、应用层）则面向资源子网，而传输层则是七层中最为重要的一层。它位于上层和下层中间，起承上启下的作用。

IP： 网络层；  
TCP/UDP： 传输层；  
HTTP、RTSP、FTP： 应用层协议；

**133、什么是C/S和B/S架构？**

软件系统体系结构:

1、C/S 架构是一种典型的两层架构，其全称是Client/Server，即客户端服务器端架构，其客户端包含一个或多个在用户的电脑上运行的程序，而服务器端有两种，一种是数据库服务器端，客户端通过数据库连接访问服务器端的数据；另一种是Socket服务器端，服务器端的程序通过Socket与客户端的程序通信。

2、B/S架构的全称为Browser/Server，即浏览器/服务器结构。Browser指的是Web浏览器，极少数事务逻辑在前端实现，但主要事务逻辑在服务器端实现，Browser客户端，WebApp服务器端和DB端构成所谓的三层架构。B/S架构的系统无须特别安装，只有Web浏览器即可。

C/S体系结构:指的是客户端/服务端，例如;QQ

B(browser)/S体系结构:指的是浏览器/服务端，例如12306(网站);购物网站

两者区别:

C/S :优点:交互性好,对服务器压力小,安全 ;缺点:服务器更新时需要同步更新客户端；

B/S:优点:不需要更新客户端 缺点:交互性差,安全性低；

**135、如果已经建立了TCP连接，但是客户端突然出现故障了怎么办？**

TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

**137、TCP和UDP的区别？**

1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接；

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付；

3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信；

4、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的，UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）；

5、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信；

6、TCP对系统资源要求较多，UDP对系统资源要求较少；TCP首部开销20字节;UDP的首部开销小，只有8个字节.

7、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道；

**138、为什么基于tcp协议的通信比基于udp协议的通信更可靠？**

TCP是面向连接的传输协议，每次都需要建立一个可以相互信任的连接，中间有个三次握手过程。而UDP是面向无连接的传输协议，不需要建立安全的连接，

tcp:可靠 对方给了确认收到信息，才发下一个，如果没收到确认信息就重发；

udp:不可靠 一直发数据，不需要对方回应；

**139、什么是局域网和广域网？**

局域网（Local Area Network），简称LAN，是指在某一区域内由多台计算机互联成的计算机组。“某一区域”指的是同一办公室、同一建筑物、同一公司和同一学校等，一般是方圆几千米以内。局域网可以实现文件管理、应用软件共享、打印机共享、扫描仪共享、工作组内的日程安排、电子邮件和传真通信服务等功能。局域网是封闭型的，可以由办公室内的两台计算机组成，也可以由一个公司内的上千台计算机组成。

广域网（Wide Area Network），简称WAN，是一种跨越大的、地域性的计算机网络的集合。通常跨越省、市，甚至一个国家。广域网包括大大小小不同的子网，子网可以是局域网，也可以是小型的广域网。

两者区别：

1、两者范围不一样，广域网比局域网广:

局域网就是在固定的一个地理区域内由2台以上的电脑用网线和其他网络设备搭建而成的一个封闭的计算机组，范围在几千米以内；

广域网是一种地域跨度非常大的网络集合，范围在几十公里到几千公里。

2、两者的IP地址设置不一样:-\*+

局域网里面，必须在网络上有一个唯一的IP地址，这个IP地址是唯一的，在另外一个局域网，这个IP地址仍然能够使用。

广域网上的每一台电脑（或其他网络设备）都有一个或多个广域网IP地址，而且不能重复。

3、接口类型不同；

4、速率不同；

5、协议不同；

**140、什么是socket？简述基于tcp协议的socket通信流程？**

Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部。

服务端：创建socket对象，绑定ip端口bind(), 设置最大链接数listen(), accept()与客户端的connect()创建双向管道， send(), recv(),close()；

客户端：创建socket对象，connect()与服务端accept()创建双向管道, send(),recv(),close()；

socket通常也称作"套接字"，用于描述IP地址和端口，是一个通信链的句柄。

通信流程：

1、服务端创建一个ServerSocket对象,指定端口号,ServerSocket对象等待客户端的连接请求。

2、客户端创建一个Socket对象,指定主机地址和端口号,向服务端发出连接请求。

3、服务端接收到客户端的连接请求,建立一条TCP连接,再创建一个Socket对象与客户端的Socket对象进行通信。

4、服务端和客户端分别创建字节输入流和字节输出流,通过字节输入流获得对方发来的数据,通过字节输出流向对方发送数据。

5、当一方决定结束通信时,向对方发送结束信息;另一方接收到结束信息后,双方分别关闭各自的TCP连接。

6、ServerSocket对象停止等待客户端的连接请求。

**141、什么是粘包？出现粘包的原因？**

**粘包：**多个数据包被连续存储于连续的缓存中，在对数据包进行读取时由于无法确定发生方的发送边界，而采用某一估测值大小来进行数据读出，若双方的size不一致时就会使指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾(粘包：数据粘在一起，主要因为：接收方不知道消息之间的界限，不知道一次性提取多少字节的数据造成的数据量比较小，时间间隔比较短，就合并成了一个包，这是底层的一个优化算法（Nagle算法）)。

**出现粘包现象的原因是多方面的**，它既可能由发送方造成，也可能由接收方造成。

**发送方引起的粘包**是由TCP协议本身造成的，TCP为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据。若连续几次发送的数据都很少，通常TCP会根据优化算法把这些数据合成一包后一次发送出去，这样接收方就收到了粘包数据。

**接收方引起的粘包**是由于接收方用户进程不及时接收数据，从而导致粘包现象。这是因为接收方先把收到的数据放在系统接收缓冲区，用户进程从该缓冲区取数据，若下一包数据到达时前一包数据尚未被用户进程取走，则下一包数据放到系统接收缓冲区时就接到前一包数据之后，而用户进程根据预先设定的缓冲区大小从系统接收缓冲区取数据，这样就一次取到了多包数据。

**142、发生粘包现象如何处理？**

1、对于发送方引起的粘包现象，用户可通过编程设置来避免，TCP提供了强制数据立即传送的操作指令push，TCP软件收到该操作指令后，就立即将本段数据发送出去，而不必等待发送缓冲区满；

2、对于接收方引起的粘包，则可通过优化程序设计、精简接收进程工作量、提高接收进程优先级等措施，使其及时接收数据，从而尽量避免出现粘包现象；

3、由接收方控制，将一包数据按结构字段，人为控制分多次接收，然后合并，通过这种手段来避免粘包。

**143、IO多路复用的作用？**

I/O多路复用是用于提升效率，单个进程可以同时监听多个网络连接IO。

举例：通过一种机制，可以监视多个文件描述符，一旦描述符就绪（读就绪和写就绪），能通知程序进行相应的读写操作，I/O多路复用避免阻塞在io上，原本为多进程或多线程来接收多个连接的消息变为单进程或单线程保存多个socket的状态后轮询处理。

与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

**145、什么是防火墙？防火墙的作用是什么？**

在互联网上防火墙是一种非常有效的网络安全模型，通过它可以隔离风险区域(即Internet或有一定风险的网络)与安全区域(局域网)的连接，同时不会妨碍人们对风险区域的访问。所以它一般连接在核心交换机与外网之间。

防火墙的作用：

1、过滤进出网络的数据 ；

2、管理进出访问网络的行为 ；

3、封堵某些禁止业务；

4、记录通过防火墙信息内容和活动；

5、对网络攻击检测和告警；

**149、python中如何使用进程池和线程池？**

from concurrent.futures import ThreadPoolExecutor,ProcessPoolExecutor  
import os,time,random  
from multiprocessing import Pool  
def task(n):  
   print('%s is runing' %os.getpid())  
   time.sleep(random.randint(1,3))  
   return n\*\*2  
if \_\_name\_\_ == '\_\_main\_\_':  
   # 多进程方式一  
   pool2=Pool()  
   pool2.map(task,range(10))  
   # 多进程方式二，下面这种多进程和多线程的用法一模一样  
   executor=ThreadPoolExecutor(max\_workers=3)  
   futures=[]  
   for i in range(11):  
       future=executor.submit(task,i)  
       futures.append(future)  
   executor.shutdown(True)  
   print('+++>')  
   for future in futures:  
       print(future.result())

**150、threading.local的作用？**

为每个线程创建一个独立的空间，使得线程对自己的空间中的数据进行操作(数据隔离)。

**151、进程之间如何进行通信？**

1、共享内存；

2、通过mmap模块实现；

3、信号；

4、通过Queue队列；

5、通过Pipe管道；

6、通过socket；

python提供了多种进程通信的方式，主要Queue和Pipe这两种方式，Queue用于多个进程间实现通信，Pipe是两个进程的通信。

**152、什么是并发和并行？**

1、并发:指应用能够交替执行不同的任务,其实并发有点类似于多线程的原理,多线程并非是同时执行多个任务,如果你开两个线程执行,就是在你几乎不可能察觉到的速度不断去切换这两个任务,已达到"同时执行效果",其实并不是的,只是计算机的速度太快,我们无法察觉到而已.

2、并行:指应用能够同时执行不同的任务,

3、并发是多个事件在同一时间段执行，并行是多个事件在统一时间点执行。

并发：同一时刻只能处理一个任务，但可以交替处理多个任务。(一个处理器同时处理多个任务)；

并行：同一时刻可以处理多个任务。(多个处理器或者是多核的处理器同时处理多个不同的任务)；

类比：并发是一个人同时吃三个馒头，而并行是三个人同时吃三个馒头。

**153、解释什么是同步、异步、阻塞与非阻塞？**

**同步和异步：**

同步：就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。

异步：当一个异步过程调用发出后，调用者不会立刻得到结果。实际处理这个调用的部件是在调用发出后，通过状态、通知来通知调用者，或通过回调函数处理这个调用。

**阻塞和非阻塞**

阻塞：阻塞调用是指调用结果返回之前，当前线程会被挂起。函数只有在得到结果之后才会返回。非阻塞：指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

**阻塞，非阻塞：**进程/线程要访问的数据是否就绪，进程/线程是否需要等待；

**同步，异步：**访问数据的方式，同步需要主动读写数据，在读写数据的过程中还是会阻塞；异步只需要I/O操作完成的通知，并不主动读写数据，由操作系统内核完成数据的读写。

**154、同步和异步，阻塞和非阻塞的区别？**

同步：执行一个操作之后，需要主动等待返回结果；异步：执行一个操作之后，不需要主动等待返回结果，若接收到结果通知，再回来执行刚才没执行完的操作。

同步和异步关心的问题是：要不要主动等待结果。

阻塞：在执行一个操作时，不能做其他操作；非阻塞：在执行一个操作时，能做其他操作。

阻塞和非阻塞关心的问题是：能不能做其他操作。

**154、路由器和交换机的区别？**

交换机是一种用于电信号转发的网络设备。路由器是链接因特网中各局域网和广域网的设备。

区别

1、交换机工作在第二层，数据链

路层，路由器工作在第三层，网络层。

2、路由器提供防火墙服务。

3、传统交换机只能风格冲突域，不能分割广播域，二路由器可以分割广播域。

**157、如何修改本地hosts文件？**

进入c:\windows\system32\drivers\etc进行修改；

Hosts是一个没有扩展名的系统文件，可以用记事本等工具打开，其作用就是将一些常用的网址域名与其对应的IP地址建立一个关联“数据库”，当用户在浏览器中输入一个需要登录的网址时，系统会首先自动从Hosts文件中寻找对应的IP地址，一旦找到，系统会立即打开对应网页，如果没有找到，则系统会再将网址提交给DNS域名解析服务器进行IP地址的解析。

文件路径：C:\WINDOWS\system32\drivers\etc。将127.0.0.1 www.163.com 添加在最下面，修改后用浏览器访问“www.163.com”会被解析到127.0.0.1，导致无法显示该网页。

**158、生产者消费者模型的应用场景？**

说明：生产者只在仓库未满时进行生产，仓库满时生产者进程被阻塞；消费者只在仓库非空时进行消费，仓库为空时消费者进程被阻塞；

应用场景：处理数据比较消耗时间，线程独占，生产数据不需要即时的反馈等。比如说写入日志，将多线程产生的日志放在队列中，然后写入。

**159、什么是CDN？**

cdn全称是内容分发网络。其目的是让用户能够更快速的得到请求的数据。

cdn就是用来加速的，他能让用户就近访问数据，这样就更快的获取到需要的数据。目的是使用户可以就近到服务器取得所需内容，解决 Internet网络拥挤的状况，提高用户访问网站的响应速度。cdn 即内容分发网络

**160、traceroute使用哪种网络协议？**

traceroute (Windows 系统下是tracert) 命令利用ICMP 协议定位您的计算机和目标计算机之间的所有路由器。

**161、守护线程，守护进程是什么？**

**主进程创建守护进程**

1、守护进程会在主进程代码运行结束的情况下，立即挂掉。

2、守护进程本身就是一个子进程。

3、主进程在其代码结束后就已经算运行完毕了（守护进程在此时就被回收）,然后主进程会一直等非守护的子进程都运行完毕后回收子进程的资源(否则会产生僵尸进程)，才会结束，

**守护线程**

1、守护线程会在该进程内所有非守护线程全部都运行完毕后,守护线程才会挂掉。并不是主线程运行完毕后守护线程挂掉。这一点是和守护进程的区别之处！

2、守护线程守护的是：当前进程内所有的子线程！

3、主线程在其他非守护线程运行完毕后才算运行完毕（守护线程在此时就被回收）。因为主线程的结束意味着进程的结束，进程整体的资源都将被回收，而进程必须保证非守护线程都运行完毕后才能结束。

**162、简述多进程开发中join和deamon的区别？**

join：当子线程调用join时，主线程会被阻塞，当子线程结束后，主线程才能继续执行。

deamon：当子进程被设置为守护进程时，主进程结束，不管子进程是否执行完毕，都会随着主进程的结束而结束。

**163、GIL锁对python性能的影响？**

会降低多线程的效率。可以说python就是个单线程的程序。

如何避免：

1、用多进程代替多线程；

2、使用其他解释器；

GIL：全局解释器锁。每个线程在执行的过程都需要先获取GIL，保证同一时刻只有一个线程可以执行字节码。

线程释放GIL锁的情况：

在IO操作等可能会引起阻塞的system call之前,可以暂时释放GIL,但在执行完毕后,必须重新获取GIL;

Python 3.x使用计时器（执行时间达到阈值后，当前线程释放GIL）或Python 2.x，tickets计数达到100;

Python使用多进程是可以利用多核的CPU资源的。

多线程爬取比单线程性能有提升，因为遇到IO阻塞会自动释放GIL锁

**164、使用yield实现一个协程？**

def consumer():

r = ''

while True:

n = yield r

if n is None:

return

print('[CONSUMER] Consuming %s...' % n)

r = '200 OK'

def produce(c):

c.send(None)

n = 0

while n < 5:

n = n + 1

print('[PRODUCER] Producing %s...' % n)

r = c.send(n)

print('[PRODUCER] Consumer return: %s' % r)

c.close()

c = consumer()

produce(c)

**165、使用async语法实现一个协程？**

import asyncio  
import time  
now = lambda : time.time()  
async def hello():  
   print("hello")  
   await asyncio.sleep(2)  
   return "done"  
     
start = now()  
# 协程对象  
h1 = hello()  
h2 = hello()  
h3 = hello()  
​  
# 创建一个事件loop  
loop = asyncio.get\_event\_loop()  
# 任务（task）对象  
tasks = [  
asyncio.ensure\_future(h1),  
asyncio.ensure\_future(h2),  
asyncio.ensure\_future(h3),  
]  
# 将协程加入到事件循环loop  
loop.run\_until\_complete(asyncio.wait(tasks))  
for task in tasks:  
   print(task.result())  
print(now()-start)

**168、什么是gevent？**

gevent是一个python网络框架，它为各种并发和网络相关的任务提供了整洁的API；

gevent是第三方库，通过greenlet实现协程，其基本思想是：当一个greenlet遇到IO操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。由于IO操作非常耗时，经常使程序处于等待状态，有了gevent为我们自动切换协程，就保证总有greenlet在运行，而不是等待IO。

**169、什么是twisted框架？**

twisted是用python实现的基于事件驱动的网络引擎框架。

**170、什么是LVS？**

LVS是linux虚拟服务器，是一个虚拟的linux集群系统；

**171、什么是Nginx？**

nginx是一个高性能的HTTP和反向代理服务，也是一个IMAP/POP3/SMTP服务；Nginx是一款自由的、开源的、高性能的HTTP服务器和反向代理服务器，同时也是一个IMAP、POP3、SMTP代理服务器。可以用作HTTP服务器、方向代理服务器、负载均衡。

**172、什么是keepalived？**

Keepalived是Linux下一个轻量级别的高可用解决方案；

**173、什么是haproxy？**

HAProxy是一个使用C语言编写的自由及开放源代码软件，其提供高可用性、负载均衡，以及基于TCP和HTTP的应用程序代理；

**174、什么是负载均衡？**

1、负载均衡建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

2、负载均衡其意思就是分摊到多个操作单元上进行执行。

系统的扩展可分为纵向（垂直）扩展和横向（水平）扩展。纵向扩展，是从单机的角度通过增加硬件处理能力，比如CPU处理能力，内存容量，磁盘等方面，实现服务器处理能力的提升，不能满足大型分布式系统（网站），大流量，高并发，海量数据的问题。因此需要采用横向扩展的方式，通过添加机器来满足大型网站服务的处理能力。比如：一台机器不能满足，则增加两台或者多台机器，共同承担访问压力。这就是典型的集群和负载均衡架构。

**175、是否使用过Nginx实现负载均衡？用过哪些负载均衡算法？**

**一、关于Nginx的负载均衡：**

　　在服务器集群中，Nginx起到一个代理服务器的角色（即反向代理），为了避免单独一个服务器压力过大，将来自用户的请求转发给不同的服务器。详情请查看我的另一篇博客。

**二、Nginx负载均衡策略：**

　　负载均衡用于从“upstream”模块定义的后端服务器列表中选取一台服务器接受用户的请求。一个最基本的upstream模块是这样的，模块内的server是服务器列表：

#动态服务器组

upstream dynamic\_zuoyu {

server localhost:8080; #tomcat 7.0

server localhost:8081; #tomcat 8.0

server localhost:8082; #tomcat 8.5

server localhost:8083; #tomcat 9.0

}

　　在upstream模块配置完成后，要让指定的访问反向代理到服务器列表：

#其他页面反向代理到tomcat容器

location ~ .\*$ {

index index.jsp index.html;

proxy\_pass http://dynamic\_zuoyu;

}

这就是最基本的负载均衡实例，但这不足以满足实际需求；

**目前Nginx服务器的upstream模块支持6种方式的分配：**

**负载均衡策略：**

轮询 默认方式

weight 权重方式

ip\_hash 依据ip分配方式

least\_conn 最少连接方式

fair（第三方） 响应时间方式

url\_hash（第三方） 依据URL分配方式

在这里，只详细说明Nginx自带的负载均衡策略，第三方不多描述。

**1、轮询**

　　最基本的配置方法，上面的例子就是轮询的方式，它是upstream模块默认的负载均衡默认策略。每个请求会按时间顺序逐一分配到不同的后端服务器。

　　有如下参数：

fail\_timeout 与max\_fails结合使用。

max\_fails

设置在fail\_timeout参数设置的时间内最大失败次数，如果在这个时间内，所有针对该服务器的请求都失败了，那么认为该服务器会被认为是停机了，

fail\_time 服务器会被认为停机的时间长度,默认为10s。

backup 标记该服务器为备用服务器。当主服务器停止时，请求会被发送到它这里。

down 标记服务器永久停机了。

　　注意：

在轮询中，如果服务器down掉了，会自动剔除该服务器。

缺省配置就是轮询策略。

此策略适合服务器配置相当，无状态且短平快的服务使用。

**2、weight**

　　权重方式，在轮询策略的基础上指定轮询的几率。例子如下：

#动态服务器组

upstream dynamic\_zuoyu {

server localhost:8080 weight=2; #tomcat 7.0

server localhost:8081; #tomcat 8.0

server localhost:8082 backup; #tomcat 8.5

server localhost:8083 max\_fails=3 fail\_timeout=20s; #tomcat 9.0

}

　　在该例子中，weight参数用于指定轮询几率，weight的默认值为1,；weight的数值与访问比率成正比，比如Tomcat 7.0被访问的几率为其他服务器的两倍。

　　注意：

权重越高分配到需要处理的请求越多。

此策略可以与least\_conn和ip\_hash结合使用。

此策略比较适合服务器的硬件配置差别比较大的情况。

**3、ip\_hash**

　　指定负载均衡器按照基于客户端IP的分配方式，这个方法确保了相同的客户端的请求一直发送到相同的服务器，以保证session会话。这样每个访客都固定访问一个后端服务器，可以解决session不能跨服务器的问题。

#动态服务器组

upstream dynamic\_zuoyu {

ip\_hash; #保证每个访客固定访问一个后端服务器

server localhost:8080 weight=2; #tomcat 7.0

server localhost:8081; #tomcat 8.0

server localhost:8082; #tomcat 8.5

server localhost:8083 max\_fails=3 fail\_timeout=20s; #tomcat 9.0

}

　　注意：

在nginx版本1.3.1之前，不能在ip\_hash中使用权重（weight）。

ip\_hash不能与backup同时使用。

此策略适合有状态服务，比如session。

当有服务器需要剔除，必须手动down掉。

**4、least\_conn**

　　把请求转发给连接数较少的后端服务器。轮询算法是把请求平均的转发给各个后端，使它们的负载大致相同；但是，有些请求占用的时间很长，会导致其所在的后端负载较高。这种情况下，least\_conn这种方式就可以达到更好的负载均衡效果。

#动态服务器组

upstream dynamic\_zuoyu {

least\_conn; #把请求转发给连接数较少的后端服务器

server localhost:8080 weight=2; #tomcat 7.0

server localhost:8081; #tomcat 8.0

server localhost:8082 backup; #tomcat 8.5

server localhost:8083 max\_fails=3 fail\_timeout=20s; #tomcat 9.0

}

　　注意：

此负载均衡策略适合请求处理时间长短不一造成服务器过载的情况。

**5、第三方策略**

　　第三方的负载均衡策略的实现需要安装第三方插件。

①fair

　　按照服务器端的响应时间来分配请求，响应时间短的优先分配。

#动态服务器组

upstream dynamic\_zuoyu {

server localhost:8080; #tomcat 7.0

server localhost:8081; #tomcat 8.0

server localhost:8082; #tomcat 8.5

server localhost:8083; #tomcat 9.0

fair; #实现响应时间短的优先分配

}

②url\_hash

　　按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，要配合缓存命中来使用。同一个资源多次请求，可能会到达不同的服务器上，导致不必要的多次下载，缓存命中率不高，以及一些资源时间的浪费。而使用url\_hash，可以使得同一个url（也就是同一个资源请求）会到达同一台服务器，一旦缓存住了资

#动态服务器组

upstream dynamic\_zuoyu {

hash $request\_uri; #实现每个url定向到同一个后端服务器

server localhost:8080; #tomcat 7.0

server localhost:8081; #tomcat 8.0

server localhost:8082; #tomcat 8.5

server localhost:8083; #tomcat 9.0

}

**三、总结**

　　以上便是6种负载均衡策略的实现方式，其中除了轮询和轮询权重外，都是Nginx根据不同的算法实现的。在实际运用中，需要根据不同的场景选择性运用，大都是多种策略结合使用以达到实际需求。

**176、简述什么是正向代理和反向代理？**

**正向代理代理客户端(客户端找一个代理去访问服务器，服务器不知道你的真实IP)，反向代理代理服务器(服务器找一个代理给你响应，你不知道服务器的真实IP)；**

**正向代理：**正向代理类似一个跳板机，代理访问外部资源。

正向代理 是一个位于客户端和原始服务器(origin server)之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端必须要进行一些特别的设置才能使用正向代理。

**正向代理作用：**

1、访问原来无法访问的资源，如google ；

2、可以做缓存，加速访问资源；

3、对客户端访问授权，上网进行认证；

4、代理可以记录用户访问记录（上网行为管理），对外隐藏用户信息；

**反向代理：**

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

**反向代理的作用：**

1、保证内网的安全，可以使用反向代理提供WAF功能，阻止web攻击；

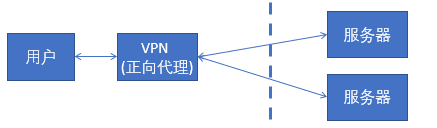
2、负载均衡，通过反向代理服务器来优化网站的负载；

**176、详述什么是正向代理和反向代理?**

1. 正向代理

正向代理，是在用户端的。比如需要访问某些国外网站，我们可能需要购买vpn。并且vpn是在我们的用户浏览器端设置的(并不是在远端的服务器设置)。

浏览器先访问vpn地址，vpn地址转发请求，并最后将请求结果原路返回来。

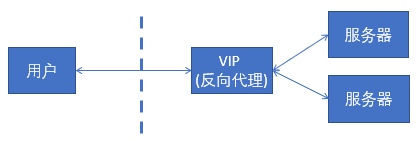


2. 反向代理

有正向代理，就有反向代理。(哈哈，因为起名字的人也会考虑，为什么不叫代理，而是取名"正向"代理)。

反向代理是作用在服务器端的，是一个虚拟ip(VIP)。对于用户的一个请求，会转发到多个后端处理器中的一台来处理该具体请求。

大型网站都有DNS(域名解析服务器)，load balance(负载均衡器)等。



总结，nginx作为软件能支持反向代理，也就是说nginx可以作为负载均衡器。

(负载均衡器可以提高网站性能，支持更高并发请求)

分类: 概念浏览器先访问vpn地址，vpn地址转发请求，并最后将请求结果原路返回来。

反向代理，这个词相信搞网络的朋友都很熟悉的，但是具体是什么意思呢？说实话，复杂的我也不懂，就我个人理解而言，反向代理有很多用途，比如说保护真实服务器不被外界攻击，加速网络等等。今天我们要介绍的就是加速网络的一种。

**177、阐述Django项目中一个请求的生命周期？**

简述Django请求生命周期：

一般是用户通过浏览器向我们的服务器发起一个请求(request),这个请求会去访问视图函数，如果不涉及到数据调用，那么这个时候视图函数返回一个模板也就是一个网页给用户） 视图函数调用模型去数据库查找数据，然后逐级返回，视图函数把返回的数据填充到模板中空格中，最后返回网页给用户。

1.wsgi ,请求封装后交给web框架（Flask，Django)   
2.中间件，对请求进行校验或在请求对象中添加其他相关数据，例如：csrf,request.session

3.路由匹配 根据浏览器发送的不同url去匹配不同的视图函数

4.视图函数，在视图函数中进行业务逻辑的处理，可能涉及到：orm，templates

5.中间件，对响应的数据进行处理 6.wsgi，将响应的内容发送给浏览器

详述Django请求生命周期：

1、用户通过浏览器请求一个页面；

2、请求到达Request Middlewares中间件，中间件对request做一些预处理或者直接response请求；

3、URLConf通过urls.py文件和请求的URL找到相应的View；

4、View Middlewares被访问，它同样可以对request做一些处理或者直接返回response；

5、调用View中的函数；

6、View中的方法可以选择性的通过Models访问底层的数据；

7、所有的Model-to-DB的交互都是通过manager完成的；

8、如果需要，Views可以使用一个特殊的Context；

9、Context被传给Template用来生成页面；

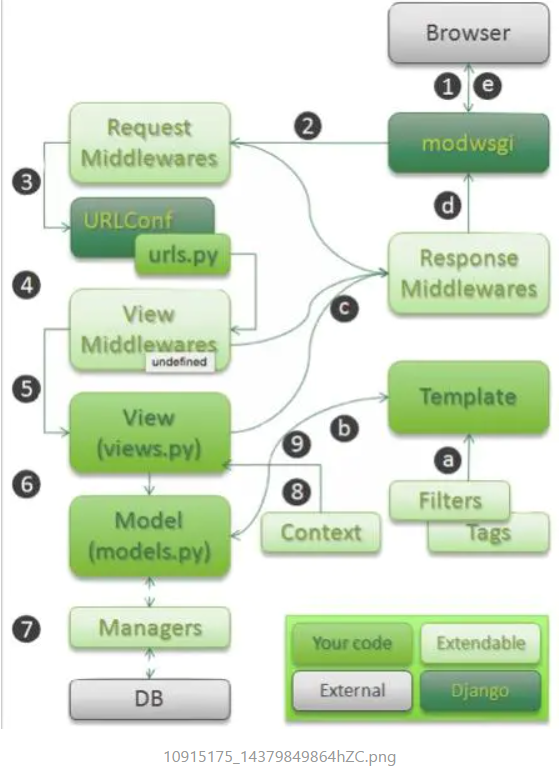
a. Template使用Filters和Tags去渲染输出

b. 输出被返回到View

c. HTTPResponse被发送到Response Middlewares

d. 任何Response Middlewares都可以丰富response或者返回一个完全不同的response

e. Response返回到浏览器，呈现给用户



**178、接口评审？**

针对同一个接口，根据实际业务需求，为解决接口兼容性问题，可以对接口进行版本扩展，命名规范为“名词”+“动词”+“版本号”形式，版本号采用v1、v2、v3形式命名，接口中统一使用小写字母，根据RFC3986定义，URI是对大小写敏感的，所以为了避免歧义，我们尽量用小写字符。

接口管理规范；

架构管理规范；

代码管理规范；

　　无规矩不成方圆，微服务架构下，代码的管理一般采用git进行管控，因此，在使用git进行版本控制时，应遵循一些原则及规范。

代码管理原则：代码管理的原则，用于确保代码管理过程中不出现原则性错误，出现原则性错误，则会出现许多无用的操作，基本原则如下：

模式确实后，必须严格遵循执行；

提交代码时，禁止代码强制提交；

代码提交时，必须进行注释说明；

代码提交时，必须按照规范执行；

出现冲突时，必须进行确认处理；

**数据库**

**MySQL数据库**

**000、了解过Hbase，DB2，SQLServer，Access吗？**

1、Hbase：HBase是一个分布式的、面向列的开源数据库；

2、DB2：一套关系型数据库管理系统；

3、SQLServer：SQL Server是由Microsoft开发和推广的关系数据库管理系统；

4、Sccess：Access是由微软发布的关系数据库管理系统。

**001、列举常见的关系型数据库和非关系型数据库？**

关系型数据库：Oracle、DB2、Microsoft SQL Server、Microsoft Access、MySQL；

非关系型数据库：mongodb，redis，elasticsearch，NoSql；

**002、讲一下数据库的连表有哪些？**

答：左连 右脸 内联 自关联

1.左右连接：以哪个表为主，结果集为“主表”的全部记录百+“副表”与“主表”相匹配的记录，如果“副表”中没有和“主表”相匹配的记录，则相对应的记录显示为NULL  
2.左连接：左边表全部行+右边表相匹配的行，如果左边表中度的某一行，在右边表中没有匹配的行，则显示NULL（left join 或者left outer join）  
3.右连接：和左连接相反。（right join 或者right outer join）  
4.内连接：它返回字段ID（连接条件）同时存在于两个表中的记录，也就是说，仅当至少有一个同属于两表的行符合联接条件时，内联接才返回行，内联接消除与另一个表中的任何行不匹配的行。（inner　join或者join）  
5.全连接：不管匹配不匹配，全部都显示出来回。（full join或者full outer join）  
6.交叉连接：没有WHERE 子句的交叉联接将产生答联接所涉及的表的笛卡尔积。第一个表的行数乘以第二个表的行数等于笛卡尔积结果集的大小。（cross join不带where）  
7.自连接：给自己取个别名，一个表当两个表来使用。

**002、Mysql常见数据库引擎及区别？**

1、InnoDB：用于事务处理应用程序，具有众多特性，包括ACID事务支持。(提供行级锁)；

2、MyISAM：默认的MySQL插件式存储引擎，它是在Web、数据仓储和其他应用环境下最常使用的存储引擎之一。注意，通过更改STORAGE\_ENGINE配置变量，能够方便地更改MySQL服务器的默认存储引擎；

3、Memory：将所有数据保存再RAM中；

**003、简述事务及其四大特性？**

彻底理解数据库事务: <http://www.hollischuang.com/archives/898>

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作（用户定义的一个数据库操作序列），要么完全地执行要么完全地不执行（这些操作要么全做要么全不做），是一个不可分割的工作单位；

事务是一个不可分割的数据库操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。事务是逻辑上的一组操作，要么都执行，要么都不执行。

**事物的四大特性(ACID)介绍一下**：原子性、一致性、隔离性和持续性。

1、原子性：事务是数据库的逻辑工作单位，事务中包括的诸操作要么都做，要么都不做。事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；

2、一致性：事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；

3、隔离性：一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能互相干扰。并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；

4、持续性：持续性也称永久性，指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的，接下来的其他操作或故障不应该对其执行结果有任何影响。

事务机制可以确保数据一致性。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

01、原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。

02、一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

03、隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

04、持久性（durability）。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

对于支持事务的数据库， 在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。commit()方法游标的所有更新操作，rollback（）方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。（在上一个笔记上面写了mysql语句在python中的使用）

事务的用户隔离级别: 数据库使用者可以控制数据库工作在哪个级别下,就可与防止不同的隔离性问题 read uncommitted --不做任何隔离,可能脏读,幻读 read committed --可以防止脏读,不能防止不可重复读,和幻读, Repeatable read --可以防止脏读,不可重复读,不能防止幻读 Serializable --数据库运行在串行化实现,所有问题都没有,就是性能低

脏读、不可重复读、幻读： 也许有很多读者会对上述隔离级别中提及到的 脏读、不可重复读、幻读 的理解有点吃力，我在这里尝试使用通俗的方式来解释这三种语义：

脏读：所谓的脏读，其实就是读到了别的事务回滚前的脏数据。比如事务B执行过程中修改了数据X，在未提交前，事务A读取了X，而事务B却回滚了，这样事务A就形成了脏读。

也就是说，当前事务读到的数据是别的事务想要修改成为的但是没有修改成功的数据。

不可重复读：事务A首先读取了一条数据，然后执行逻辑的时候，事务B将这条数据改变了，然后事务A再次读取的时候，发现数据不匹配了，就是所谓的不可重复读了。

也就是说，当前事务先进行了一次数据读取，然后再次读取到的数据是别的事务修改成功的数据，导致两次读取到的数据不匹配，也就照应了不可重复读的语义。

幻读：事务A首先根据条件索引得到N条数据，然后事务B改变了这N条数据之外的M条或者增添了M条符合事务A搜索条件的数据，导致事务A再次搜索发现有N+M条数据了，就产生了幻读。

也就是说，当前事务读第一次取到的数据比后来读取到数据条目少。

不可重复读和幻读比较： 两者有些相似，但是前者针对的是update或delete，后者针对的insert。

## 004、请给出你熟悉关系数据库三大范式，有什么作用？

在进行数据库的设计时，所遵循的一些规范，只要按照设计规范进行设计，就能设计出没有数据冗余和数据维护异常的数据库结构。

简写版：

数据库三大范式是什么？

第一范式：每个列都不可以再拆分。

第二范式：在第一范式的基础上，非主键列完全依赖于主键，而不能是依赖于主键的一部分。

第三范式：在第二范式的基础上，非主键列只依赖于主键，不依赖于其他非主键。

在设计数据库结构的时候，要尽量遵守三范式，如果不遵守，必须有足够的理由。比如性能。事实上我们经常会为了性能而妥协数据库的设计。

详细版：

数据库的设计的规范有很多，通常来说我们在设是数据库时只要达到其中一些规范就可以了，这些规范又称之为数据库的三范式，一共有三条，也存在着其他范式，我们只要做到满足前三个范式的要求，就能设陈出符合我们的数据库了，我们也不能全部来按照范式的要求来做，还要考虑实际的业务使用情况，所以有时候也需要做一些违反范式的要求。

**1.数据库设计的第一范式(最基本)**，基本上所有数据库的范式都是符合第一范式的，符合第一范式的表具有以下几个特点： 数据库表中的所有字段都只具有单一属性，单一属性的列是由基本的数据类型（整型，浮点型，字符型等）所构成的设计出来的表都是简单的二比表；第一范式（1NF）：在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。所谓第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，第一范式就是无重复的列。

**2.数据库设计的第二范式(是在第一范式的基础上设计的)**，要求一个表中只具有一个业务主键，也就是说符合第二范式的表中不能存在非主键列对只对部分主键的依赖关系；第二范式（2NF）：第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被惟一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为主关键字或主键、主码。第二范式（2NF）要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。简而言之，第二范式就是非主属性非部分依赖于主关键字。

**3.数据库设计的第三范式，**指每一个非主属性既不部分依赖与也不传递依赖于业务主键，也就是第二范式的基础上消除了非主属性对主键的传递依赖。第三范式（3NF）：满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。例如，存在一个部门信息表，其中每个部门有部门编号（dept\_id）、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。简而言之，第三范式就是属性不依赖于其它非主属性。（我的理解是消除冗余）

## 005、简述触发器、函数、视图和存储过程？

1、触发器：触发器是一个特殊的存储过程，它是MySQL在insert、update、delete的时候自动执行的代码块；

2、函数：MySQL中提供了许多内置函数，还可以自定义函数（实现程序员需要sql逻辑处理）；

3、视图：视图是由查询结果形成的一张虚拟表，是表通过某种运算得到的一个投影；

4、存储过程：把一段代码封装起来，当要执行这一段代码的时候，可以通过调用该存储过程来实现（经过第一次编译后再次调用不需要再次编译，比一个个执行sql语句效率高）；

## 006、数据库索引？索引的原理与作用？

* 推荐: <http://tech.meituan.com/mysql-index.html>
* [MySQL索引背后的数据结构及算法原理](http://blog.codinglabs.org/articles/theory-of-mysql-index.html)

聚集索引,非聚集索引,B-Tree,B+Tree,最左前缀原理。

作用：索引是关系型数据库中用来提升查询性能最为重要的手段，索引就好比是一本书的目录，如果需要查询某个知识点，只需要在目录中查找。

原理：索引的底层算法是B-tree（B树），B树是一种自平衡树，能够保持数据有序，这种数据结构能够让查询数据，顺序访问、插入数据删除数据都在对数时间内完成。

## 007、mysql索引种类？

1、普通索引：仅加速查询；

2、唯一索引：加速查询 + 列值唯一（可以有null）；

3、主键索引：加速查询 + 列值唯一（不可以有null）+ 表中只有一个；

4、组合索引：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并；

5、全文索引：对文本的内容进行分词，进行搜索；

**009、Mysql常见的函数？**

1、ABS(x)：返回x的绝对值；

2、ROUND(x)：返回参数x的四舍五入的一个整数；

3、TRIM(str)：去除字符串两边的空白；

4、COUNT()：返回值的个数；

5、AVG()：返回平均值；

6、SUM()：求和；

**010、列举创建索引但是无法命中索引的情况？**

1、如果条件中有or，即使其中有条件带索引也不会使用(这也是为什么尽量少用or的原因）；

2、对于多列索引，不是使用的第一部分(第一个)，则不会使用索引；

3、like查询是以%开头；

4、如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引；

5、如果mysql估计使用全表扫描要比使用索引快,则不使用索引；

**011、数据库的导入与导出命令？**

导出(mysqldump)：

导出数据和表结构：mysqldump -uroot -p dbname > dbname .sql

只导出表结构：mysqldump -uroot -p -d dbname > dbname .sql

导入：mysql -u用户名 -p密码 数据库名 < 数据库名.sql

**023、讲一下left join 和inner join的区别？**

答：一个左连是左边所有数据都展示出来，内联两个表都展示。

1 WHERE子句中使用的连接语句，在数据库语言中，被称为隐性连接。INNER JOIN……ON子句产生的连接称为显性连接。（其他JOIN参数也是显性连接）WHERE和INNER JOIN产生的连接关系，没有本质区别，结果也一样。但是！隐性连接随着数据库语言的规范和发展，已经逐渐被淘汰，比较新的数据库语言基本上已经抛弃了隐性连接，全部采用显性连接了。  
a inner join：理解为“有效连接”，两张表中都有的数据才会显示left join：理解为“有左显示”，比如on a.field=b.field，则显示a表中存在的全部数据及a\\b中都有的数据，A中有、B没有的数据以null显示  
b right join：理解为“有右显示”，比如on a.field=b.field，则显示B表中存在的全部数据及a\\b中都有的数据，B中有、A没有的数据以null显示  
c full join：理解为“全连接”，两张表中所有数据都显示，实际就是inner (left-inner) (right-inner)  
3 join可以分主次表 外联接有三种类型：完全外联，左联，右联．  
完全外联包含两张表的所有记录．  
左联是以左边的表为主，右边的为辅，右联则相反  
4一般要使得数据库查询语句性能好点遵循一下原则:  
在做表与表的连接查询时，大表在前，小表在后  
不使用表别名，通过字段e799bee5baa6e59b9ee7ad9431333363393063前缀区分不同表中的字段  
查询条件中的限制条件要写在表连接条件前  
尽量使用索引的字段做为查询条件

**027、简述left join和right join的区别？**

左连接和右连接很相似，只是左右表位置的不同罢了。

1、left join(左连接) 返回包括左表中的所有记录和右表中连接字段相等的记录

2、right join(右连接) 返回包括右表中的所有记录和左表中连接字段相等的记录

**024、知道union 和union all 的区别吗** ？

union和union all的区别是,union会自动压缩多个结果集合中的重复结果，而union all则将所有的结果全部显示出来，不管是不是重复。

Union：对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序；

UNION在进行表链接后会筛选掉重复的记录，所以在表链接后会对所产生的结果集进行排序运算，删除重复的记录再返回结果。

实际大部分应用中是不会产生重复的记录，最常见的是过程表与历史表UNION

Union All：对两个结果集进行并集操作，包括重复行，不进行排序；

如果返回的两个结果集中有重复的数据，那么返回的结果集就会包含重复的数据了。

Union：对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序；

Union All：对两个结果集进行并集操作，包括重复行，不进行排序；

**013、char和varchar的区别？**

1、保存方式不一样；

2、char有固定的长度，而varchar属于可变长的字符类型；

3、char的效率比varchar高；

**014、MySQL执行计划的作用和使用方法？**

作用：用来进行查询分析，比如整个查询涉及多少防，使用哪些索引，运行时间等；

使用方法：使用explain关键字，如explain select xxx from xxx;

**015、为什么数据很大的时候使用limit offset分页时，越往后翻速度越慢，如何优化？**

使用limit分页时，比如limit 10000，20的意思是扫描满足条件的10020行，扔掉前面的10000行，最后返回20行，问题就出在这，当数据量大的时候，大量时间花在了扫描上面了。

优化方法：

1、当一个数据库表过于庞大，LIMIT offset, length中的offset值过大，则SQL查询语句会非常缓慢，你需增加order by，并且order by字段需要建立索引。

2、如果使用子查询去优化LIMIT的话，则子查询必须是连续的，某种意义来讲，子查询不应该有where条件，where会过滤数据，使数据失去连续性

**016、什么是索引合并？**

1、索引合并是把几个索引的范围扫描合并成一个索引。

2、索引合并的时候，会对索引进行并集，交集或者先交集再并集操作，以便合并成一个索引。

3、这些需要合并的索引只能是一个表的。不能对多表进行索引合并。

**017、什么是覆盖索引？**

覆盖索引又可以称为索引覆盖。

解释一： 就是select的数据列只用从索引中就能够取得，不必从数据表中读取，换句话说查询列要被所使用的索引覆盖。

解释二： 索引是高效找到行的一个方法，当能通过检索索引就可以读取想要的数据，那就不需要再到数据表中读取行了。如果一个索引包含了（或覆盖了）满足查询语句中字段与条件的数据就叫做覆盖索引。

解释三： 是非聚集组合索引的一种形式，它包括在查询里的Select、Join和Where子句用到的所有列（即建立索引的字段正好是覆盖查询语句[select子句]与查询条件[Where子句]中所涉及的字段，也即，索引包含了查询正在查找的所有数据）。

**018、简述数据库的读写分离？**

读写分离就是在主服务器上修改，数据会同步到从服务器，从服务器只能提供读取数据，不能写入，实现备份的同时也实现了数据库性能的优化，以及提升了服务器安全。

**019、简述数据库分库分表？**

垂直切分

垂直分库：

就是根据业务耦合性，将关联度低的不同表存储在不同的数据库

垂直分表：

基于数据库中的"列"进行，某个表字段较多，可以新建一张扩展表，将不经常用或字段长度较大的字段拆分出去到扩展表中。

横向切分：

水平切分分为库内分表和分库分表，是根据表内数据内在的逻辑关系，将同一个表按不同的条件分散到多个数据库或多个表中，每个表中只包含一部分数据，从而使得单个表的数据量变小，达到分布式的效果。

**020、数据库锁的作用？**

锁分为三种：乐观锁，悲观锁和共享锁；

数据库和操作系统一样，是一个多用户使用的共享资源。当多个用户并发地存取数据时，在数据库中就会产生多个事务同时存取同一数据的情况。若对并发操作不加控制就可能会读取和存储不正确的数据，破坏数据库的一致性。

加锁是实现数据库并发控制的一个非常重要的技术。在实际应用中经常会遇到的与锁相关的异常情况，当两个事务需要一组有冲突的锁，而不能将事务继续下去的话，就会出现死锁，严重影响应用的正常执行。

**021、MySQL的半同步复制原理？**

**半同步复制，**介于异步复制和全同步复制之间，主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接收到并写到relay log中才返回给客户端。相对于异步复制，半同步复制牺牲了一定的性能，提高了数据的安全性。

**异步复制，**MySQL默认的复制是异步的，主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理。原理最简单，性能最好，但是主从之间数据不一致的概率很大。

**全同步复制，**指当主库执行完一个事务，所有的从库都执行了该事务才返回给客户端。因为需要等待所有从库执行完该事务才能返回，所以全同步复制的性能必然会收到严重的影响。

**022、MySQL的增删改查？**

**增**

指定字段名

语法：INSERT INTO 表名（字段名1，字段名2，…）VALUES（值1，值2，…）；

举例：INSERT INTO student(id,name,grade) VALUES(1,'zhangshan',98);

不指定字段名

语法：INSERT INTO 表名 VALUES(值11，值2，…）；

举例：INSERT INTO student VALUES (2,'lisi',62);

其他写法

语法：INSERT INTO 表名 SET 字段名1=值1[,字段名2=值2，…]

举例：INSERT INTO student SET id=4，name='zhaoliu',grade=72;

同时添加多条数据

语法：INSERT INTO 表名[(字段名1，字段名2，…)] VALUES （值1，值2，…），（值1，值2，…），…（值1，值2，…）

举例：INSERT INTO student VALUES (5，‘lilei’,99), (6,'hanmeimei',87), (8,'poly',76);

**删**

删除部分数据

语法：DELETE FROM 表名 [WHERE 条件表达式]

命令：DELETE  FROM student WHERE id=7;

删除全部数据

语法：DELETE FROM 表名

命令：DELETE FROM student；

推荐的删除全部数据

语法：TRUNCTE [TABLE ] 表名

举例：TRUNCATE TABLE student

**改**

更新部分数据

语法：UPDATE 表名 SET 字段名1=值1，[ ，字段名2=值2，…] [ WHERE 条件表达式 ]

命令：UPDATE student SET name=‘caocao’,grade=50 WHERE id=1;

更新全部数据

语法：UPDATE 表名 SET 字段名=值

命令：UPDATE student SET grade=80；

**查**

查询所有字段

语法：SELECT 字段名1，字段名2，… FROM 表名 （该语法也可以查询部分字段）

语法：SELECT \* FROM 表名；

**按条件查询**

语法：SELECT 字段名1，字段名2，… FROM 表名 WHERE 条件表达式

命令：SELECT id，name FROM student2  WHERE id=4;

**带IN关键字的查询**

语法：SELECT \* | 字段名1，字段名2，… FROM 表名 WHERE 字段名 [ NOT ]  IN （元素1，元素2，…）

命令：SELECT \* FROM student2 WHERE id IN （1,2,3）；

带 BETWEEN AND  关键字的查询

语法：SELECT \* | { 字段名1，字段名2，… } FROM  表名 WHERE 字段名 [ NOT ] BETWEEN  值1  AND  值2；

命令：SELECT id,name FROM students WHERE id BETWEEN 2 AND 5;

**空值查询**

语法：SELECT \* | 字段名1，字段名2，… FROM 表名 WHERE 字段名 IS [ NOT ] NULL

命令：SELECT \* FROM student2 WHERE gender IS NULL;

**带 DISTINCT 关键字的查询**

语法：SELECT DISTINCT 字段名 FROM 表名；

命令：SELECT DISTINCT gender FROM student2;

**带 LIKE 关键字的查询**

语法：SELECT \* | 字段名1，字段名2，… FROM 表名 WHERE 字段名 [ NOT ] LIKE ‘匹配字符串’;

注意：%表示匹配任意长度的字符串，\_表示匹配单个字符串

命令：SELECT id,name FROM student2  WHERE name LIKE "S%";

命令：SELECT id,name FROM student2 WHERE name LIKE 'w%g';

命令：SELECT id,name FROM student2 WHERE name NOT LIKE '%y%';

命令：SELECT \* FROM student2 WHERE name LIKE 'wu\_ong';

**带 AND 关键字的多条件查询**

语法：SELECT \* | 字段名1，字段名2，… FROM 表名 WHERE 条件表达式1 AND 条件表达式2 [ … AND 条件表达式 n ];

命令：SELECT id,name FROM student2 WHERE id<5 AND gender='女';

**带 OR 关键字的多条件查询**

语法：SELECT \* | 字段名1，字段名2，… FROM 表名 WHERE 条件表达式1 OR 条件表达式2 [ … OR 条件表达式 n ];

命令：SELECT id,name ,gender FROM student2 WHERE id<3 OR gender='女';

AND和OR一起使用时，AND的优先级高于OR

**聚合函数**

**COUNT()函数：统计记录的条数**

语法：SELECT COUNT(\*) FROM 表名举例：

命令：SELECT COUNT(\*) FROM student2;

**SUM()函数：求出表中某个字段所有值的总和**

语法：SELECT  SUM(字段名) FROM 表名；

命令：SELECT SUM(grade) FROM student2;

**AVG()函数：求出表中某个字段所有值的平均值**

语法：SELECT AVG(字段名) FROM 表名；

命令：SELECT AVG(grade) FROM student2;

**MAX()函数：求出表中某个字段所有值的最大值**

语法：SELECT MAX(字段名) FROM 表名；

命令：SELECT MAX(grade) FROM student2;

**MIN()函数：求出表中某个字段所有值的最小值**

语法：SELECT MIN(字段名) FROM 表名；

命令：SELECT MIN(grade) FROM student2;

**对查询结果进行排序**

语法：SELECT 字段名1，字段名2，… FROM 表名 ORDER BY 字段名1 [ ASC | DESC ],字段名2 [ ASC | DESC ]

(升序)命令：SELECT \* FROM student2 ORDER BY grade;

(降序)命令：命令：SELECT \* FROM student2 ORDER BY grade DESC;

**分组查询**

语法：SELECT  字段名1，字段名2，… FROM 表名 GROUP BY 字段名1，字段名2，… [ HAVING 条件表达式 ];

单独使用 GROUP BY 进行分组

命令：SELECT \* FROM student2 GROUP BY gender;

**GROUP BY 和聚合函数一起使用**

命令：SELECT COUNT(\*) ,gender FROM student2 GROUP BY gender;

**GROUP BY 和 HAVING 关键字一起使用**

命令：SELECT sum(grade),gender FROM student2 GROUP BY gender HAVING SUM(grade) < 300;

**使用 LIMIT 限制查询结果的数量**

语法：SELECT 字段名2，字段名2，… FROM 表名 LIMIT [ OFFSET ,] 记录数

(从0开始的4条)命令：SELECT \* FROM student LIMIT 4;

(从第五条开始的4条)命令：SELECT \* FROM student2 ORDER BY grade DESC LIMIT 4,4;

**为表和字段取别名**

语法：SELECT \* FROM 表名 [ AS ] 别名；

命令：SELECT \* FROM student2 AS s WHERE s.gender='女';

**为字段取别名**

语法：SELECT 字段名 [ AS ] 别名 [ ,字段名 [AS] 别名，…]  FROM 表名 ；

命令：SELECT name AS stu\_name,gender AS stu\_gender FROM student2;

**023、MySQL的建表语句？**

#创建表，例子  
#所谓的建表就是声明列的过程,所以要首先分析列  
create table member(  
  id int unsigned auto\_increment primary key,  
  username varchar(20) not null default '',  
  gender char(1) not null default '',  
  weight tinyint unsigned not null default 0,  
  birth date not null default '0000-00-00',  
  salary decimal(8,2) not null default 0.00,  
  lastlogin int unsigned not null default 0  
)engine myisam charset utf8;

**024、MySQL如何创建索引？**

**ALTER TABLE：**

ALTER TABLE用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。

ALTER TABLE table\_name ADD INDEX index\_name (column\_list)

ALTER TABLE table\_name ADD UNIQUE (column\_list)

ALTER TABLE table\_name ADD PRIMARY KEY (column\_list)

**CREATE INDEX：**

CREATE INDEX可对表增加普通索引或UNIQUE索引。

CREATE INDEX index\_name ON table\_name (column\_list)

CREATE UNIQUE INDEX index\_name ON table\_name (column\_list)

**025、简述SQL注入原理，以及如何在代码层面房子sql注入？**

通俗点讲，SQL注入的根本原因是: "用户输入数据"意外变成了代码被执行。 "用户输入数据"我这里可以指Web前端$\_POST,$\_GET获取的数据，也可以指从数据库获取的数据，当然也不排除程序猿无意中使用的特殊字符串。

在SQL语句的拼接中，一些含特殊字符的变量在拼接时破坏了SQL语句的结构，导致"用户输入数据"意外变成了代码被执行。

规避方法：

1、理语句法  (解析协议层面上完全规避SQL注入)；

2、字符串转义（不要在sql中拼接字符） # query作为sql模板，args为将要传入的参数 execute(query, args=None)；

**026、使用python将数据库的student表中的数据写入db.txt？**

import pymysql  
connect=pymysql.Connect(  
   host='',  
   port=,  
   user='',  
   passwd='',  
   db='',  
   charset='',  
)  
  
cursor=connect.cursor()  
sql='select \* from student'  
cursor.execute(sql)  
students=cursor.fetchall()  
​  
with open('db.txt','w') as f:  
   for student in students:  
       f.write(student)  
         
cursor.close()  
connect.close()

**028、索引有什么作用，有哪些分类，有什么好处和坏处？**

作用：为了增加查询速度，提高系统性能；

分类：

唯一索引：不允许其中任何两行具有相同索引值的索引。

非唯一索引：允许其中任何两行具有相同索引值的索引。

主键索引：有一列或列组合，其值唯一标识表中的每一行。

聚集索引：表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。

好处：1、帮助用户提高查询速度；2、利用索引的唯一性来控制记录的唯一性；3、可以加速表与表之间的连接；4、降低查询中分组和排序的时间；

坏处：

1、存储索引占用磁盘空间；2、执行数据修改操作(INSERT、UPDATE、DELETE)产生索引维护；

**029、mysql慢日志？**

MySQL的慢查询日志是MySQL提供的一种日志记录，它用来记录在MySQL中响应时间超过阀值的语句，具体指运行时间超过long\_query\_time值的SQL，则会被记录到慢查询日志中。

**030、数据库的乐观锁和悲观锁是什么？举例怎么实现的？**

**简写版：**

1、数据库管理系统（DBMS）中的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时不破坏事务的隔离性和统一性以及数据库的统一性。乐观并发控制（乐观锁）和悲观并发控制（悲观锁）是并发控制主要采用的技术手段。

2、悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。在查询完数据的时候就把事务锁起来，直到提交事务。实现方式：使用数据库中的锁机制；

3、乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。在修改数据的时候把事务锁起来，通过version的方式来进行锁定。实现方式：乐一般会使用

本号机制或CAS算法实现。

**详写版：**

悲观锁和乐观锁是什么？

**(1)悲观锁（Pessimistic Lock）：**

悲观锁的特点是先获取锁，再进行业务操作，即“悲观”的认为获取锁是非常有可能失败的，因此要先确保获取锁成功再进行业务操作。通常所说的“一锁二查三更新”即指的是使用悲观锁。通常来讲在数据库上的悲观锁需要数据库本身提供支持，即通过常用的select … for update操作来实现悲观锁。当数据库执行select for update时会获取被select中的数据行的行锁，因此其他并发执行的select for update如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。select for update获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。

这里需要注意的一点是不同的数据库对select for update的实现和支持都是有所区别的，例如oracle支持select for update no wait，表示如果拿不到锁立刻报错，而不是等待，mysql就没有no wait这个选项。另外mysql还有个问题是select for update语句执行中所有扫描过的行都会被锁上，这一点很容易造成问题。因此如果在mysql中用悲观锁务必要确定走了索引，而不是全表扫描。

**(2)乐观锁（Optimistic Lock）：**

乐观锁的特点先进行业务操作，不到万不得已不去拿锁。即“乐观”的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。

乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。一般的做法是在需要锁的数据上增加一个版本号，或者时间戳，然后按照如下方式实现：

复制代码

1. SELECT data AS old\_data, version AS old\_version FROM …;

2. 根据获取的数据进行业务操作，得到new\_data和new\_version

3. UPDATE SET data = new\_data, version = new\_version WHERE version = old\_version

if (updated row > 0) {

    // 乐观锁获取成功，操作完成

} else {

    // 乐观锁获取失败，回滚并重试

}

乐观锁是否在事务中其实都是无所谓的，其底层机制是这样：在数据库内部update同一行的时候是不允许并发的，即数据库每次执行一条update语句时会获取被update行的写锁，直到这一行被成功更新后才释放。因此在业务操作进行前获取需要锁的数据的当前版本号，然后实际更新数据时再次对比版本号确认与之前获取的相同，并更新版本号，即可确认这之间没有发生并发的修改。如果更新失败即可认为老版本的数据已经被并发修改掉而不存在了，此时认为获取锁失败，需要回滚整个业务操作并可根据需要重试整个过程。

**总结：**

乐观锁在不发生取锁失败的情况下开销比悲观锁小，但是一旦发生失败回滚开销则比较大，因此适合用在取锁失败概率比较小的场景，可以提升系统并发性能

乐观锁还适用于一些比较特殊的场景，例如在业务操作过程中无法和数据库保持连接等悲观锁无法适用的地方。

何谓悲观锁与乐观锁

乐观锁对应于生活中乐观的人总是想着事情往好的方向发展，悲观锁对应于生活中悲观的人总是想着事情往坏的方向发展。这两种人各有优缺点，不能不以场景而定说一种人好于另外一种人。

**悲观锁：**

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

**乐观锁：**

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

**两种锁的使用场景：**

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

乐观锁常见的两种实现方式

乐观锁一般会使用版本号机制或CAS算法实现。

版本号机制 一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

举一个简单的例子：

假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ；而当前帐户余额字段（ balance ）为 $100 。当需要对账户信息表进行更新的时候，需要首先读取version字段。

操作员 A 此时将其读出（ version=1 ），并从其帐户余额中扣除 $50（ $100-$50 ）。 在操作员 A 操作的过程中，操作员B 也读入此用户信息（ version=1 ），并从其帐户余额中扣除 $20 （ $100-$20 ）。 操作员 A 完成了修改工作，提交更新之前会先看数据库的版本和自己读取到的版本是否一致，一致的话，就会将数据版本号加1（ version=2 ），连同帐户扣除后余额（ balance=$50 ），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2 。 操作员 B 完成了操作，提交更新之前会先看数据库的版本和自己读取到的版本是否一致，但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2 ，而自己读取到的版本号为1 ，不满足 “ 当前最后更新的version与操作员第一次读取的版本号相等 “ 的乐观锁策略，因此，操作员 B 的提交被驳回。 这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

CAS算法 即compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS算法涉及到三个操作数

需要读写的内存值 V 进行比较的值 A 拟写入的新值 B 当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

**乐观锁的缺点：**

ABA 问题是乐观锁一个常见的问题

1 ABA 问题 如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA"问题。

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

2 循环时间长开销大 自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3 只能保证一个共享变量的原子操作 CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作。

CAS与synchronized的使用情景 简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）

对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。 对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。 补充： Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为 “重量级锁” 。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 偏向锁 和 轻量级锁 以及其它各种优化之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠 Lock-Free 的队列，基本思路是 自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

悲观锁

悲观锁（Pessimistic Lock），顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。

Java synchronized 就属于悲观锁的一种实现，每次线程要修改数据时都先获得锁，保证同一时刻只有一个线程能操作数据，其他线程则会被block。

乐观锁

乐观锁（Optimistic Lock），顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交更新的时候会判断一下在此期间别人有没有去更新这个数据。乐观锁适用于读多写少的应用场景，这样可以提高吞吐量。

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。

乐观锁一般来说有以下2种方式：

1. 使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。
2. 使用时间戳（timestamp）。乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

Java JUC中的atomic包就是乐观锁的一种实现，AtomicInteger 通过CAS（Compare And Set）操作实现线程安全的自增。

MySQL隐式和显示锁定

MySQL InnoDB采用的是两阶段锁定协议（two-phase locking protocol）。在事务执行过程中，随时都可以执行锁定，锁只有在执行 COMMIT或者ROLLBACK的时候才会释放，并且所有的锁是在同一时刻被释放。前面描述的锁定都是隐式锁定，InnoDB会根据事务隔离级别在需要的时候自动加锁。

另外，InnoDB也支持通过特定的语句进行显示锁定，这些语句不属于SQL规范：

* SELECT ... LOCK IN SHARE MODE
* SELECT ... FOR UPDATE
* 乐观锁与悲观锁的区别
* 乐观锁的思路一般是表中增加版本字段，更新时where语句中增加版本的判断，算是一种CAS（Compare And Swep）操作，商品库存场景中number起到了版本控制（相当于version）的作用（ AND number=#{number}）。
* 悲观锁之所以是悲观，在于他认为本次操作会发生并发冲突，所以一开始就对商品加上锁（SELECT ... FOR UPDATE），然后就可以安心的做判断和更新，因为这时候不会有别人更新这条商品库存。
* 小结
* 这里我们通过 MySQL 乐观锁与悲观锁 解决并发更新库存的问题，当然还有其它解决方案，例如使用 分布式锁。目前常见分布式锁实现有两种：基于Redis和基于Zookeeper，基于这两种 业界也有开源的解决方案，例如 [Redisson Distributed locks](https://link.jianshu.com?t=https:/github.com/redisson/redisson/wiki/8.-distributed-locks-and-synchronizers) 、 [Apache Curator Shared Lock](https://link.jianshu.com?t=http:/curator.apache.org/curator-recipes/shared-lock.html) ，这里就不细说，网上Google 一下就有很多资料。

悲观锁：正如其名，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）的修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

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。

如何选择乐观锁还是悲观锁:

1.响应速度：如果需要非常高的响应速度，建议采用乐观锁方案，成功就执行，不成功就失败，不需要等待其他并发去释放锁

2.冲突频率：如果冲突频率非常高，建议采用悲观锁，保证成功率，如果冲突频率大，乐观锁会需要多次重试才能成功，代价比较大

3.重试代价：如果重试代价大，建议采用悲观锁

乐观锁

总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或CAS操作实现。

悲观锁

总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。  
注：要使用数据库的悲观锁，我们必须关闭mysql数据库的自动提交属性，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。

悲观锁分为两种：共享锁和排它锁

共享锁是其它事务可以读但是不能写  
排他锁是只有自己得事务有权限对此数据进行读写

**举例讲一下具体应用场景（乐观锁）？**

答：讲了一个订单。

俗话说金三银四，不管是为了升职加薪还是打算换个环境，现在正是广大程序员们的跳槽季，IT行业正暗流涌动腥风血雨，互联网寒冬似乎并没有过去，很多企业依然在裁员，还有部分企业光明正大的实行着"996"，这一行可能正面临着最大的竞争压力和生存压力。我们无法改变环境，也无法左右市场，但是我们能改变和左右的是自己的知识深度和广度，只有武装到牙齿，不断的充实自己，才能在面试这场江湖纷争里游刃有余。

你是不是经常被面试官突如其来的问题问的语无伦次或者毫无底气？进而在一整场面试里都失去了信心。最近有个群员就遇到这个问你，面试官让他自我介绍以后，就突然问他mysql的悲观锁和乐观锁区别，而这正好是他的盲区，第一个问题就被堵住了，这场面试的结果也可想而知。下面就从面试角度来浅谈一下，该怎样回答这个问题。

一、概念上区别

乐观锁（\*\* Optimistic Locking\*\*）：顾名思义，对加锁持有一种乐观的态度，即先进行业务操作，不到最后一步不进行加锁，"乐观"的认为加锁一定会成功的，在最后一步更新数据的时候再进行加锁。

悲观锁\*\*（Pessimistic Lock）\*\*：正如其名字一样，悲观锁对数据加锁持有一种悲观的态度。因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

二、实现方式：

乐观锁：

* version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

sql实现代码：

update table set x=x+1, version=version+1 where id=#{id} and version=#{version};

* CAS（定义见后）操作方式：即compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。（这种方式作者也是最近刚知道，惭愧惭愧）

悲观锁：是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了（原理：共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程），如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。

三、使用场景

* 乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。
* 悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

四、特点

* 乐观锁：乐观锁的特点先进行业务操作，不到万不得已不去拿锁。即“乐观”的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。
* 悲观锁：悲观锁的特点是先获取锁，再进行业务操作，即“悲观”的认为获取锁是非常有可能失败的，因此要先确保获取锁成功再进行业务操作。通常所说的“一锁二查三更新”即指的是使用悲观锁。

五、sql实现代码和案例

因为本文主要是讲解的是面试的时候怎样回答，所以就不将具体sql代码和案例贴出来了，感兴趣的同学可以自行去看一下，如果以后有时间，我也会单独写一篇详细的实现案例出来。

总结：以上，基本回答完前四点后，这道题已经能在面试官心里达到八九十分了。但是我们的人生不仅仅是为了面试，希望同学们一定要弄懂其中的原理和区别，悲观锁和乐观锁都有自己的优缺点，简而言之记得一句话：读取频繁使用乐观锁，写入频繁使用悲观锁。乐观锁不能解决脏读的问题。

如果觉得本文有用，请推荐给更多有需要的人，谢谢！如果发现问题，欢迎留言，请随时批评改正，谢谢！

现在我有一个购买商品的需求，我们知道当我们购买商品时，后台会进行减库存和增加购买记录的操作。我们分别在无锁和乐观锁和悲观锁进行相应的代码演示来说明问题。

ySql的乐观锁：

    我们在使用乐观锁时会假设在极大多数情况下不会形成冲突，只有在数据提交的时候，才会对数据是否产生冲突进行检验。如果数据产生冲突了，则返回错误信息，进行相应的处理。

    实现：MySql最经常使用的乐观锁时进行版本控制，也就是在数据库表中增加一列，记为version，当我们将数据读出时，将版本号一并读出，当数据进行更新时，会对这个版本号进行加1，当我们提交数据时，会判断数据库表中当前的version列值和当时读出的version是否相同，若相同说明没有进行更新的操作，不然，则取消这次的操作。

三、悲观锁

    MySql的悲观锁就是打开事务，当启动事务时，如果事务中的sql语句涉及到索引并用索引进行了条件判断，那么会使用行级锁锁定所要修改的行，否则使用表锁锁住整张表。

我们可以在不同的场合使用不同的处理方法，乐观锁并发高并且性能也很好，而悲观锁虽然并发不是很高，但是它不允许脏读，所以各有各的优点。

乐观锁与悲观锁的具体区别: <http://www.cnblogs.com/Bob-FD/p/3352216.html>

**031、MVCC？**

全称是Multi-Version Concurrent Control，即多版本并发控制，在MVCC协议下，每个读操作会看到一个一致性的snapshot，并且可以实现非阻塞的读。MVCC允许数据具有多个版本，这个版本可以是时间戳或者是全局递增的事务ID，在同一个时间点，不同的事务看到的数据是不同的。

**032、**[**MySQL**](http://lib.csdn.net/base/mysql)**的innodb引擎是如何实现MVCC的？**

innodb会为每一行添加两个字段，分别表示该行创建的版本和删除的版本，填入的是事务的版本号，这个版本号随着事务的创建不断递增。在repeated read的隔离级别（[事务的隔离级别请看这篇文章](http://blog.csdn.net/chosen0ne/article/details/10036775)）下，具体各种数据库操作的实现：

01、select：满足以下两个条件innodb会返回该行数据：

1、该行的创建版本号小于等于当前版本号，用于保证在select操作之前所有的操作已经执行落地。

2、该行的删除版本号大于当前版本或者为空。删除版本号大于当前版本意味着有一个并发事务将该行删除了。

02、insert：将新插入的行的创建版本号设置为当前系统的版本号。

03、delete：将要删除的行的删除版本号设置为当前系统的版本号。

04、update：不执行原地update，而是转换成insert + delete。将旧行的删除版本号设置为当前版本号，并将新行insert同时设置创建版本号为当前版本号。

其中，写操作（insert、delete和update）执行时，需要将系统版本号递增。

​由于旧数据并不真正的删除，所以必须对这些数据进行清理，innodb会开启一个后台线程执行清理工作，具体的规则是将删除版本号小于当前系统版本的行删除，这个过程叫做purge。通过MVCC很好的实现了事务的隔离性，可以达到repeated read级别，要实现serializable还必须加锁。

参考：[MVCC浅析](http://blog.csdn.net/chosen0ne/article/details/18093187)

**033、你用的 mysql 是哪个引擎，各引擎间有什么区别？MyISAM和InnoDB？**

主要 MyISAM 与 InnoDB 两个引擎，其主要区别如下：

01、InnoDB 支持事务，MyISAM 不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MyISAM就不可以了;

02、MyISAM 适合查询以及插入为主的应用，InnoDB 适合频繁修改以及涉及到安全性较高的应用;

03、InnoDB 支持外键，MyISAM 不支持;

04、MyISAM 是默认引擎，InnoDB 需要指定;

05、InnoDB 不支持 FULLTEXT 类型的索引;

06、InnoDB 中不保存表的行数，如 select count(\*) from table 时，InnoDB;需要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count(\*)语句包含 where 条件时 MyISAM 也需要扫描整个表;

07、对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM表中可以和其他字段一起建立联合索引;

08、清空整个表时，InnoDB 是一行一行的删除，效率非常慢。MyISAM 则会重建表;

09、InnoDB 支持行锁(某些情况下还是锁整表，如 update table set a=1 where user like ‘%lee%’

10、InnoDB索引是聚簇索引，MyISAM索引是非聚簇索引。

11、InnoDB的主键索引的叶子节点存储着行数据，因此主键索引非常高效。

12、MyISAM索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。

14、InnoDB非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要update一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到读操作完成。另外，MyISAM 对于 SELECT COUNT(\*) 这类的计算是超快无比的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。他是它支持“行锁” ，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

|| InnoDB |MyISAM ---|---|---

事务|支持|不支持

必须有主键|是|否

外键|支持|不支持

表级锁|支持|不支持

行级锁|支持|支持

全文索引| 支持(V5.5+)|支持

表格被压缩后可查询|不支持|支持

保存表的具体行数| 不保存|保存

存储格式|frm/ibd|frm/MYD/MYI

**如何选择：**

是否要支持事务，如果要请选择innodb，如果不需要可以考虑MyISAM；

如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读也有写，请使用InnoDB。

系统奔溃后，MyISAM恢复起来更困难，能否接受；

MySQL5.5版本开始Innodb已经成为Mysql的默认引擎(之前是MyISAM)，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

**051、InnoDB引擎的4大特性？**

1、插入缓冲（insert buffer)；

2、二次写(double write)；

3、自适应哈希索引(ahi)；

4、预读(read ahead)；

**005.InnoDB为什么推荐使用自增ID作为主键？**

自增ID可以保证每次插入时B+索引是从右边扩展的，可以避免B+树和频繁合并和分裂（对比使用UUID）。如果使用字符串主键和随机主键，会使得数据随机插入，效率比较差

mysql 数据库引擎: <http://www.cnblogs.com/0201zcr/p/5296843.html>  
MySQL存储引擎－－MyISAM与InnoDB区别: <https://segmentfault.com/a/1190000008227211>

**沉默是金** [**https://blog.markhoo.com**](https://blog.markhoo.com)

**037、主键、超键、候选键、外键？**

主键：数据库表中对存储数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值(Null)。

超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

候选键：是最小超键，即没有冗余元素的超键。

外键：在一个表中存在的另一个表的主键称此表的外键。

**038、视图的作用，视图可以更改么？**

1、视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询;不包含任何列或数据。使用视图可以简化复杂的sql操作，隐藏具体的细节，保护数据;视图创建后，可以使用与表相同的方式利用它们。

2、视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有order by则对视图再次order by将被覆盖。

3、创建视图： create view xxx as xxxxxx；对于某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的，对视图的更新将对基表进行更新;但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

**039、drop,delete与truncate的区别？**

drop直接删掉表，truncate删除表中数据，再插入时自增长id又从1开始，delete删除表中数据，可以加where字句。

1.delete 语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行回滚操作。truncate table则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器，执行速度快。

2.表和索引所占空间。当表被truncate后，这个表和索引所占用的空间会恢复到初始大小，而delete操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

3.一般而言，drop>truncate>delete

4.应用范围。truncate只能对table，delete可以是table和view

5.truncate和delete只删除数据，而drop则删除整个表（结构和数据)

6.truncate与不带where的delete:只删除数据，而不删除表的结构（定义）；drop语句将删除表的结构被依赖的约束(constrain),触发器（trigger)索引(index);依赖于该表的存储过程/函数将被保留，但其状态会变为:invalid.

**040、索引的工作原理及其种类？**

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询，更新数据库表中数据。索引的实现通常使用B树以其变种B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间（因为索引也要随之变动）

**041、连接的种类？**

查询分析器中执行：  
--建表table1,table2：  
create table table1(id int,name varchar(10))  
create table table2(id int,score int)  
insert into table1 select 1,'lee'  
insert into table1 select 2,'zhang'  
insert into table1 select 4,'wang'  
insert into table2 select 1,90  
insert into table2 select 2,100  
insert into table2 select 3,70  
如表  
-------------------------------------------------  
table1 | table2 |  
-------------------------------------------------  
id name |id score |  
1 lee |1 90|  
2 zhang| 2 100|  
4 wang| 3 70|  
-------------------------------------------------  
以下均在查询分析器中执行  
**一、外连接**  
1.概念：包括左向外联接、右向外联接或完整外部联接  
2.左连接：left join 或 left outer join  
(1)左向外联接的结果集包括 LEFT OUTER 子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值(null)。  
**(2)sql 语句**  
select \* from table1 left join table2 on table1.id=table2.id  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang2100  
4wangNULLNULL  
------------------------------  
注释：包含table1的所有子句，根据指定条件返回table2相应的字段，不符合的以null显示  
**3.右连接：right join 或 right outer join**  
(1)右向外联接是左向外联接的反向联接。将返回右表的所有行。如果右表的某行在左表中没有匹配行，则将为左表返回空值。  
(2)sql 语句  
select \* from table1 right join table2 on table1.id=table2.id  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang2100  
NULLNULL370  
------------------------------  
注释：包含table2的所有子句，根据指定条件返回table1相应的字段，不符合的以null显示  
**4.完整外部联接:full join 或 full outer join**  
(1)完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。  
(2)sql 语句  
select \* from table1 full join table2 on table1.id=table2.id  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang2100  
4wangNULLNULL  
NULLNULL370  
------------------------------  
注释：返回左右连接的和（见上左、右连接）  
**二、内连接**  
1.概念：内联接是用比较运算符比较要联接列的值的联接  
2.内连接：join 或 inner join  
3.sql 语句  
select \* from table1 join table2 on table1.id=table2.id  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang2100  
------------------------------  
注释：只返回符合条件的table1和table2的列  
4.等价（与下列执行效果相同）  
A:select a.\*,b.\* from table1 a,table2 b where a.id=b.id  
B:select \* from table1 cross join table2 where table1.id=table2.id (注：cross join后加条件只能用where,不能用on)  
**三、交叉连接(完全)**  
1.概念：没有 WHERE 子句的交叉联接将产生联接所涉及的表的笛卡尔积。第一个表的行数乘以第二个表的行数等于笛卡尔积结果集的大小。（table1和table2交叉连接产生3\*3=9条记录）  
2.交叉连接：cross join (不带条件where...)  
3.sql语句  
select \* from table1 cross join table2  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang190  
4wang190  
1lee2100  
2zhang2100  
4wang2100  
1lee370  
2zhang370  
4wang370  
------------------------------  
注释：返回3\*3=9条记录，即笛卡尔积  
4.等价（与下列执行效果相同）  
A:select \* from table1,table2

**042、数据库优化的思路？**

这个我借鉴了慕课上关于数据库优化的课程。

**1.SQL语句优化**

1）应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。  
2）应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num is null；  
可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：select id from t where num=0；  
3）很多时候用 exists 代替 in 是一个好的选择；  
4）用Where子句替换HAVING 子句 因为HAVING 只会在检索出所有记录之后才对结果集进行过滤；

**2.索引优化**

看上文索引

**3.数据库结构优化**

1）范式优化： 比如消除冗余（节省空间。）

2）反范式优化：比如适当加冗余等（减少join）

3）拆分表：

一个好的数据库设计方案对于数据库的性能往往会起到事半功倍的效果。

需要考虑数据冗余、查询和更新的速度、字段的数据类型是否合理等多方面的内容。

**将字段很多的表分解成多个表：**对于字段较多的表，如果有些字段的使用频率很低，可以将这些字段分离出来形成新表。因为当一个表的数据量很大时，会由于使用频率低的字段的存在而变慢。

**增加中间表：**对于需要经常联合查询的表，可以建立中间表以提高查询效率。通过建立中间表，将需要通过联合查询的数据插入到中间表中，然后将原来的联合查询改为对中间表的查询。

**增加冗余字段：**设计数据表时应尽量遵循范式理论的规约，尽可能的减少冗余字段，让数据库设计看起来精致、优雅。但是，合理的加入冗余字段可以提高查询速度。表的规范化程度越高，表和表之间的关系越多，需要连接查询的情况也就越多，性能也就越差。

分区将数据在物理上分隔开，不同分区的数据可以制定保存在处于不同磁盘上的数据文件里。这样，当对这个表进行查询时，只需要在表分区中进行扫描，而不必进行全表扫描，明显缩短了查询时间，另外处于不同磁盘的分区也将对这个表的数据传输分散在不同的磁盘I/O，一个精心设置的分区可以将数据传输对磁盘I/O竞争均匀地分散开。对数据量大的时时表可采取此方法。可按月自动建表分区。  
4）拆分其实又分垂直拆分和水平拆分：

案例： 简单购物系统暂设涉及如下表： 1.产品表（数据量10w，稳定） 2.订单表（数据量200w，且有增长趋势） 3.用户表 （数据量100w，且有增长趋势）。以[mysql](http://www.2cto.com/database/MySQL/)为例讲述下水平拆分和垂直拆分，mysql能容忍的数量级在百万静态数据可以到千万。

垂直拆分： 解决问题：表与表之间的io竞争，不解决问题：单表中数据量增长出现的压力。方案： 把产品表和用户表放到一个server上，订单表单独放到一个server上 。

水平拆分： 解决问题：单表中数据量增长出现的压力 ，不解决问题：表与表之间的io争夺。  
方案： 用户表通过性别拆分为男用户表和女用户表 订单表通过已完成和完成中拆分为已完成订单和未完成订单产品表未完成订单放一个server上 已完成订单表盒男用户表放一个server上女用户表放一个server上(女的爱购物，哈哈)；

**4.服务器硬件优化：**这个么多花钱咯！

**面试回答数据库优化问题从以下几个层面入手**

（1）、根据服务层面：配置mysql性能优化参数；

（2）、从系统层面增强mysql的性能：优化数据表结构、字段类型、字段索引、分表，分库、读写分离等等。

（3）、从数据库层面增强性能：优化SQL语句，合理使用字段索引。

（4）、从代码层面增强性能：使用缓存和NoSQL数据库方式存储，如MongoDB/Memcached/Redis来缓解高并发下数据库查询的压力。

（5）、减少数据库操作次数，尽量使用数据库访问驱动的批处理方法。

（6）、不常使用的数据迁移备份，避免每次都在海量数据中去检索。

（7）、提升数据库服务器硬件配置，或者搭建数据库集群。

（8）、编程手段防止SQL注入：使用JDBC PreparedStatement按位插入或查询；正则表达式过滤（非法字符串过滤）；

**013、简述数据库有做过优化吗（详述数据库优化在后面）？**

答：业务量大的时候，建立主从，实现读写分离。

优化手段:

a)  创建索引:创建合适的索引，我们就可以现在索引中查询，查询到以后直接找对应的记录。

b)  分表   ：当一张表的数据比较多或者一张表的某些字段的值比较多并且很少使用时，采用水平分表和垂直分表来优化

c)  读写分离：当一台服务器不能满足需求时，采用读写分离的方式进行集群。

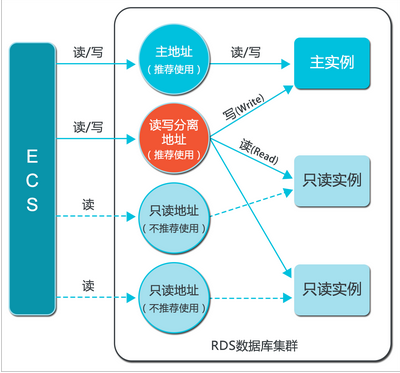
d)  缓存:使用  redis来进行缓存等等。

在互联网项目中，当业务规模越来越大，数据也越来越多，随之而来的就是数据库压力会越来越大。

我们可能会采取各种方式去优化，比如之前文章提到的缓存方案，SQL优化等等，除了这些方式以外，这里再分享几个针对数据库优化的常规手段：「数据读写分离」与「数据库Sharding」。这两点基本上是大中型互联网项目中应用的非常普遍的方案了。

下面我们来详细看一看，

**一、从读写分离到CQRS**

  
（图片来源阿里云）

由于互联网业务场景，大多数是读多写少，因此进行数据库的读写分离是一件非常简单且有效率的方案。   
读写分离简单点来说就是把对数据的读操作和写操作进行分开来，让这两种操作去访问不同的数据库，这样的话，就可以减轻数据库的压力了。

例如上图中，数据库会有一个「主实例」，这个主要用来提供写操作的（偶尔也会承担一点读操作），除了「主实例」以外，还会有多个「从实例」（在图中显示的是 只读实例），「从实例」的功能只是用来承担读操作的。

那上面就出现了多个数据库了，在多个数据库之间的数据是怎么保证一致性的呢？

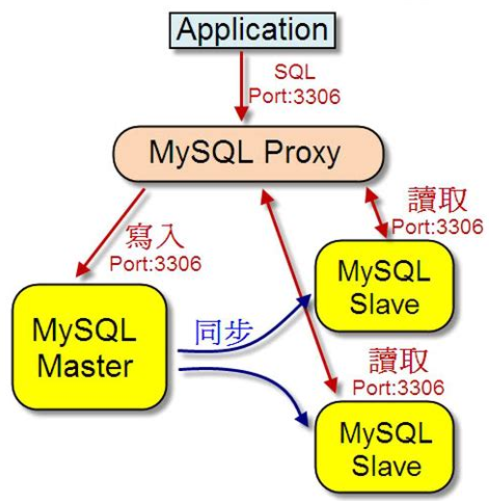
其实，我们常用的数据库就自带这类同步功能，比如 Mysql，它自己有一个master-slave功能，可以实现主库与从库数据的自动同步，是基于二进制日志复制来实现的。在主库进行的写操作，会形成二进制日志，然后Mysql会把这个日志异步的同步到从库上，从库再自动执行一遍这个二进制日志，那么数据就跟主库一致了。

除了Mysql以外，像Oracle等商业数据库都有类似的功能，甚至是网络上还有很多开源的第三方数据同步工具，也有很多成熟好用的。

好了，「主实例」与「从实例」之间的数据同步问题解决了，那现在还有一个问题就是，项目中是怎样让 写请求 去访问「主实例」，让 读请求 去访问「从实例」的，这个路由规则是怎么实现的呢？

常规的有2种方式：

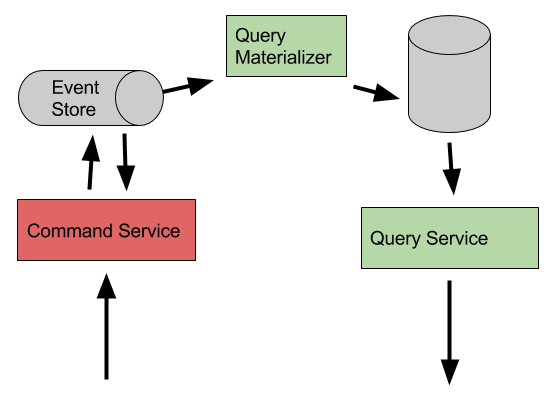
1. **使用编码方式** ：  
   这个方式主要是靠开发同学在编码的时候，根据读写不同的操作需求，去调用不同的数据源。例如在数据操作层（DAO层）将读数据与写数据分开为两个方法（函数），然后为这两个方法分别指定不同的数据库即可。   
   但是这种方式有点硬编码的味道了，而且对开发同学而言还得额外关注这个事情，多了一个编码成本且容易不小心忽略掉。
2. **使用中间件：**   
   这种方式就是在后端数据库的前面，前置一个 数据库代理服务，如下图的：MySQL-Proxy 是Mysql提供的一个中间件，用于实现读写分离请求，但这个组件实际用的人不多，我们可以选择其它的一些开源的组件替代，例如：MyCat、ProxySQL 等等，但大致的原理比较类似，通过这个图很容易理解这个模式。



好了，基础的读写分离就讲完了，但感觉这个方式虽然实用是实用，就是不怎么有逼格。

OK，想要有逼格是吧，满足你，那我们就来聊一聊另一个有逼格的读写分离概念: 「 CQRS 」

**CQRS**：Command Query Responsibility Segregation   
**命令（增删改）和查询的责任分离**



我们还是先看图，通过上图可以简单的理解一下CQRS

CQRS 重点强调的就是 Query（读） 和 Command（写）的分离，在业务上将职责分离清晰，Command 主要做业务逻辑的执行，Query来负责数据查询和展示。同时 这两种操作是基于不同的数据源，甚至是一个是数据库，另外一个是NoSQL都可以，Query去查询的数据源可以直接按照领域模型进行存储，而并不是按照数据模型去存储，这样查询出来就立即可以展示，而不用转换，且查询效率高。

其实CQRS是由鼎鼎大名的 Martin Fowler 提出，搞计算机的应该都认识。想要更深入的去学习CQRS，可以翻看Martin Fowler公开的资料。

**二、Sharding（分库分表）**

上面讲完了数据库的读写分离，现在我们来聊一下数据库的Sharding。

随着数据库里的数据越来越大，单表查询的性能已经不能满足业务要求了，这个时候就需要进行分表处理了，将大表拆分为若干个小表，不同的分表中数据也不一样，这样可以分散查询压力，提高处理效率。

然而，当表越来越多，所有的数据都在一个数据库上时，网络IO以及文件IO也都会集中在一个数据库上，可能会超过单台服务器的容量， CPU、内存、文件IO、网络IO 都会成为系统的瓶颈，QPS/TPS也会超过单数据库实例的处理极限。那么这个时候就需要对数据库进行分片处理。

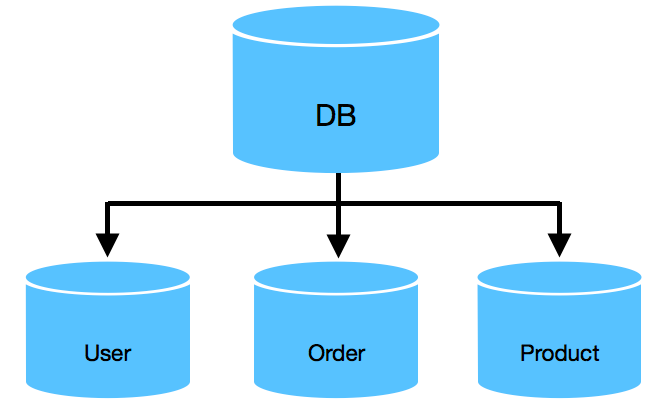
因为分表和分库的思路类似，因此下面统一来聊技术方案。   
其实分库分表只是我们通俗的便于理解的说话，正确的描述应该是：数据分片

数据的分片主要有2种模式：

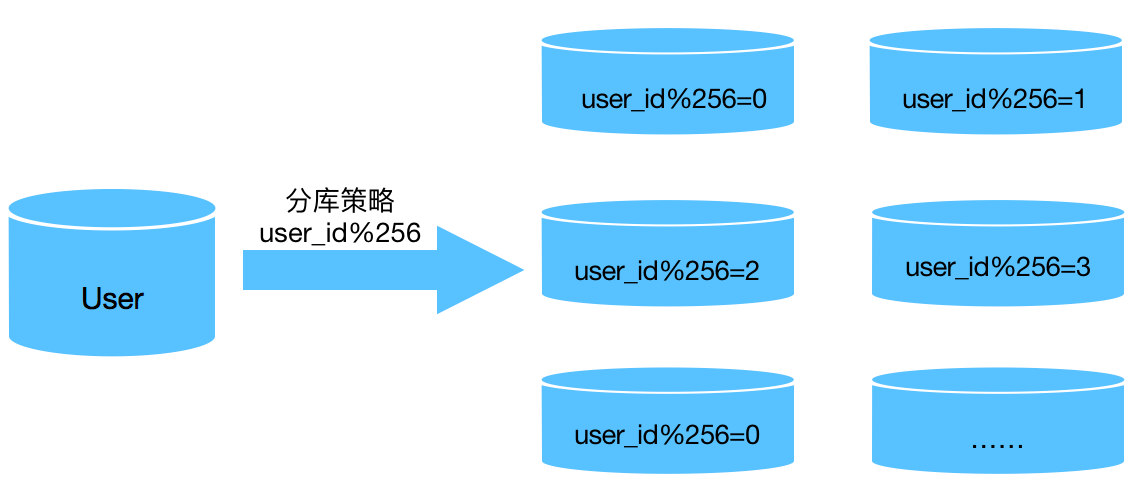
* **垂直拆分**
* **水平拆分**

两种拆分应用的场景是不同的：

1. **垂直拆分**，是指按照业务模块进行拆分。简单来讲，就是把业务紧密的模块的字段/表放在一起，放在同一个数据库或者服务器上。将不同业务的字段/表进行独立，拆到不同的数据库或者服务器上。比如一个游戏系统中，可以将玩家基本信息与道具公会等信息进行拆分。   
   如图示例：

  
（图片来源网络）

1. **水平拆分**，是指纯粹的按照某种数据规则/格式进行拆分。例如 按照数据唯一ID的哈希散列拆分、按照数据的日期拆分、按照某种范围拆分等等。水平拆分需要注意的是，随着数据动态的变化，分片数量可能需要随之动态调整，另外就是水平分片是没有考虑业务特征的，因此在进行业务汇总查询或者分片中事物处理的时候就比较麻烦一些。   
   如图示例：



另外，在实际应用中，两种拆分模式一般会结合在一起使用，效果更佳。

以上就是数据库性能优化之「数据读写分离」与「数据库Sharding」方法，欢迎大家一起交流。

**043、存储过程与触发器的区别？**

1、触发器与存储过程非常相似，触发器也是SQL语句集，两者唯一的区别是触发器不能用EXECUTE语句调用，而是在用户执行Transact-SQL语句时自动触发（激活）执行。

2、触发器是在一个修改了指定表中的数据时执行的存储过程。通常通过创建触发器来强制实现不同表中的逻辑相关数据的引用完整性和一致性。由于用户不能绕过触发器，所以可以用它来强制实施复杂的业务规则，以确保数据的完整性。

3、触发器不同于存储过程，触发器主要是通过事件执行触发而被执行的，而存储过程可以通过存储过程名称名字而直接调用。

4、当对某一表进行诸如UPDATE、INSERT、DELETE这些操作时，SQLSERVER就会自动执行触发器所定义的SQL语句，从而确保对数据的处理必须符合这些SQL语句所定义的规则。

**044、你常用的mysql引擎有哪些? 各引擎间有什么区别?**

**（1）MyISAM存储引擎：**

不支持事务，也不支持外键，优势是访问速度快，对事务完整性没有要求或者以选择，插入的为主应用基本上可以用这个引擎来创建³³表；

1、可以配合锁，实现操作系统下的复制备份，迁移；

2、使用表级锁，并发性差；

3、支持全文检索（MySQL InonoDB在5.6以后也支持全文检索）；

4、主机宕机后，MyISAM表易损坏，灾难恢复性不佳；

5、无事务支持；

6、只缓存索引，数据的缓存是利用操作系统缓冲区来实现的。

7、数据紧凑存储，因此可获得更小的索引和更快的全表扫描性能。

**（2）InnoDB存储引擎（从mysql 5.5版本开始，innodb是mysql默认的存储引擎。）：**

支持事务，灾难恢复性好;支持外键.;支持热备份 ；

1、支持事务，默认的事务隔离级别是可重复读，其事务控制是通过多版本并发控制来实现的。

2、使用行级锁 ；

3、对于InnoDB引擎中的表，其数据的物理组织形式是簇表（Cluster Table），数据按主键来组织，也就是说主键索引数据表的另一种常见形式是非簇表，其索引是有序的，而数据是无序的；

4、实现了缓冲管理，不仅能缓冲索引也能缓冲数据，并且会自动创建散列索引以加快数据的获取。相比之下，MyISAM数据只是缓存了索引；

**（3）存储器存储引擎：**

存储器存储引擎使用存在于内存中的内容来创建表。

每个存储器表只实际对应一个磁盘文件，格式是.frm.memory类型的表访问非常的快，因为它的数据是放在内存中的，并且默认使用HASH索引，但是一旦服务关闭，表中的数据就会丢失掉。

**记忆体存储引擎的表可以选择使用BTREE索引或者HASH索引，两种不同类型的索引有其不同的使用范围。**

Hash索引优点： Hash索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位，不像B-Tree索引需要从根节点到枝节点，最后才能访问到页节点这样多次的IO访问，所以散列索引的查询效率要远高于B树索引。

哈希索引缺点：那么不精确查找呢，也很明显，因为哈希算法是基于等值计算的，所以对于“喜欢”等范围查找哈希索引无效，不支持;

MEMORY存储引擎提供“内存”表，也不支持事务，外键。

1、使用内存表（内存引擎）可以显着提高访问数据的速度，可用于缓存会频繁访问的，可以重构的数据，计算结果，统计值，中间结果，但也有如下这些不足之处。

2、使用的是表级锁，虽然内存访问快，但如果频繁地读读，表级锁可能会成为瓶颈所在。

3、只支持固定大小的行.VARCHAR类型的字段会存储为固定长度的CHAR类型，浪费空间。不支持TEXT，BLOB字段。

4、当有些查询需要使用到临时表（使用的也是MEMORY存储引擎）时如果表中有TEXT，BLOB字段，那么会转化为基于磁盘的MyISAM表，严重降低性能。 由于内存资源成本昂贵，一般不建议设置过大的内存表，如果内存表满了，就会在MySQL错误日志里发现类似“Thetable'table\_name'已满“这样的错误，可通过清除数据或调整内存表参数来避免报错。

5、服务器重启数据会丢失；

**（4）MERGE引擎存储：**

合并存储引擎的英文一组的MyISAM的表的组合，这些的MyISAM的表必须结构完全相同，合并表本身并没有数据，对合并类型的表可以进行查询，更新，删除操作，这些操作实际上是对内部的的MyISAM数据表进行的。

**（5）ARCHIVE引擎：**

1、存储引擎的英文被设计用来存储企业中的大量流水数据存储引擎。

2、存档引擎使用的zlib无损数据压缩，让数据都保存在压缩的存储表中。

3、当数据被插入时，它们被压缩。

4、它只支持INSERT和SELECT，支持自增键及其上的索引，不支持其他索引。

5、它适合做日志记录，用户行为分析，不需要更新，删除和索引的数据

**045、什么是SQL？**

1、结构化查询语言(Structured Query Language)简称SQL，是一种数据库查询语言。

2、作用：用于存取数据、查询、更新和管理关系数据库系统。

**046、什么是MySQL?**

MySQL是一个[关系型数据库](https://cloud.tencent.com/product/cdb-overview?from=10680)管理系统，由瑞典MySQL AB 公司开发，属于 Oracle 旗下产品。MySQL 是最流行的关系型数据库管理系统之一，在 WEB 应用方面，MySQL是最好的 RDBMS (Relational Database Management System，关系数据库管理系统) 应用软件之一。在Java企业级开发中非常常用，因为 MySQL 是开源免费的，并且方便扩展。

**047、mysql有关权限的表都有哪几个？**

MySQL服务器通过权限表来控制用户对数据库的访问，权限表存放在mysql数据库里，由mysql\_install\_db脚本初始化。这些权限表分别user，db，table\_priv，columns\_priv和host。

下面分别介绍一下这些表的结构和内容：

1、user权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。

2、db权限表：记录各个帐号在各个数据库上的操作权限。

3、table\_priv权限表：记录数据表级的操作权限。

4、columns\_priv权限表：记录数据列级的操作权限。

5、host权限表：配合db权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受GRANT和REVOKE语句的影响。

**048、MySQL的binlog有几种录入格式？分别有什么区别？**

有三种格式，statement，row和mixed。

1、statement模式下，每一条会修改数据的sql都会记录在binlog中。不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。由于sql的执行是有上下文的，因此在保存的时候需要保存相关的信息，同时还有一些使用了函数之类的语句无法被记录复制。

2、row级别下，不记录sql语句上下文相关信息，仅保存哪条记录被修改。记录单元为每一行的改动，基本是可以全部记下来但是由于很多操作，会导致大量行的改动(比如alter table)，因此这种模式的文件保存的信息太多，日志量太大。

3、mixed，一种折中的方案，普通操作使用statement记录，当无法使用statement的时候使用row。

此外，新版的MySQL中对row级别也做了一些优化，当表结构发生变化的时候，会记录语句而不是逐行记录。

**049、mysql有哪些数据类型？**

1、整数类型，包括TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT，分别表示1字节、2字节、3字节、4字节、8字节整数。任何整数类型都可以加上UNSIGNED属性，表示数据是无符号的，即非负整数。

长度：整数类型可以被指定长度，例如：INT(11)表示长度为11的INT类型。长度在大多数场景是没有意义的，它不会限制值的合法范围，只会影响显示字符的个数，而且需要和UNSIGNED ZEROFILL属性配合使用才有意义。

例子：假定类型设定为INT(5)，属性为UNSIGNED ZEROFILL，如果用户插入的数据为12的话，那么数据库实际存储数据为00012。

2、实数类型，包括FLOAT、DOUBLE、DECIMAL。DECIMAL可以用于存储比BIGINT还大的整型，能存储精确的小数。而FLOAT和DOUBLE是有取值范围的，并支持使用标准的浮点进行近似计算。计算时FLOAT和DOUBLE相比DECIMAL效率更高一些，DECIMAL你可以理解成是用字符串进行处理。

3、字符串类型，包括VARCHAR、CHAR、TEXT、BLOB

VARCHAR用于存储可变长字符串，它比定长类型更节省空间。

VARCHAR使用额外1或2个字节存储字符串长度。列长度小于255字节时，使用1字节表示，否则使用2字节表示。

VARCHAR存储的内容超出设置的长度时，内容会被截断。

CHAR是定长的，根据定义的字符串长度分配足够的空间。

CHAR会根据需要使用空格进行填充方便比较。

CHAR适合存储很短的字符串，或者所有值都接近同一个长度。

CHAR存储的内容超出设置的长度时，内容同样会被截断。

4、枚举类型（ENUM），把不重复的数据存储为一个预定义的集合。

有时可以使用ENUM代替常用的字符串类型。

ENUM存储非常紧凑，会把列表值压缩到一个或两个字节。

ENUM在内部存储时，其实存的是整数。

尽量避免使用数字作为ENUM枚举的常量，因为容易混乱。

排序是按照内部存储的整数

5、日期和时间类型，尽量使用timestamp，空间效率高于datetime，

用整数保存时间戳通常不方便处理。

如果需要存储微妙，可以使用bigint存储。

看到这里，这道真题是不是就比较容易回答了。

**052、什么是索引？**

1、索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。

2、索引是一种数据结构。数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种B+树。

3、更通俗的说，索引就相当于目录。为了方便查找书中的内容，通过对内容建立索引形成目录。索引是一个文件，它是要占据物理空间的。

**054、MySQL建立索引的优点和缺点？**

建立索引的优缺点:

为什么要创建索引呢？

        这是因为，创建索引可以大大提高系统的性能。   
        第一、通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。   
        第二、可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。   
        第三、可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。   
        第四、在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。   
        第五、通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

        也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？这种想法固然有其合理性，然而也有其片面性。

虽然，索引有许多优点， 但是，为表中的每一个列都增加索引，是非常不明智的。

这是因为，增加索引也有许多不利的一个方面:

        第一、创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

        第二、索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间。如果要建立聚簇索引，那么需要的空间就会更大。

        第三、当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

**055、什么样的字段适合创建索引？**

 索引是建立在数据库表中的某些列的上面。因此，在创建索引的时候，应该仔细考虑在哪些列上可以创建索引，在哪些列上不能创建索引。

       一般来说，应该在这些列上创建索引，例如：

       第一、在经常需要搜索的列上，可以加快搜索的速度；

       第二、在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；

       第三、在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；

       第四、在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；

       第五、在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；

       第六、在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

       建立索引，一般按照select的where条件来建立，比如： select的条件是where f1 and f2，那么如果我们在字段f1或字段f2上简历索引是没有用的，只有在字段f1和f2上同时建立索引才有用等。

**056、什么样的字段不适合创建索引？**

同样，对于有些列不应该创建索引。一般来说，不应该创建索引的的这些列具有下列特点：

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。   
第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比 例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。   
第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。   
第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。

因此，当修改性能远远大于检索性能时，不应该创建索引。

**057、创建索引的方法？**

1、创建索引，例如 create index <索引的名字> on table\_name (列的列表);   
2、修改表，例如 alter table table\_name add index[索引的名字] (列的列表);   
3、创建表的时候指定索引，例如create table table\_name ( [...], INDEX [索引的名字] (列的列表) );

查看表中索引的方法:

show index from table\_name; 查看索引

索引的类型及创建例子::

1.PRIMARY KEY （主键索引）

mysql> alter table table\_name add primary key ( `column` )

2.UNIQUE 或 UNIQUE KEY (唯一索引)

mysql> alter table table\_name add unique (`column`)  
3.FULLTEXT (全文索引)  
mysql> alter table table\_name add fulltext (`column` )  
4.INDEX (普通索引)  
mysql> alter table table\_name add index index\_name ( `column` )

5.多列索引 (聚簇索引)  
mysql> alter table `table\_name` add index index\_name ( `column1`, `column2`, `column3` )

**053、索引有哪些优缺点？**

索引的优点：

1、可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

2、通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

索引的缺点：

1、时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，会降低增/改/删的执行效率；

2、空间方面：索引需要占物理空间。

**054、索引有哪几种类型？**

**主键索引:** 数据列不允许重复，不允许为NULL，一个表只能有一个主键。

**唯一索引:** 数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。

可以通过 ALTER TABLE table\_name ADD UNIQUE (column); 创建唯一索引

可以通过 ALTER TABLE table\_name ADD UNIQUE (column1,column2); 创建唯一组合索引

**普通索引:** 基本的索引类型，没有唯一性的限制，允许为NULL值。

可以通过ALTER TABLE table\_name ADD INDEX index\_name (column);创建普通索引

可以通过ALTER TABLE table\_name ADD INDEX index\_name(column1, column2, column3);创建组合索引。

**全文索引：** 是目前搜索引擎使用的一种关键技术。

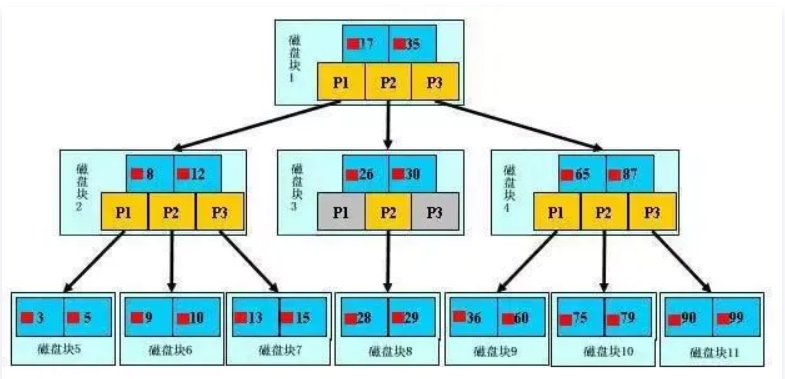
可以通过ALTER TABLE table\_name ADD FULLTEXT (column);创建全文索引

**055、索引的数据结构（b树，hash）？**

索引的数据结构和具体存储引擎的实现有关，在MySQL中使用较多的索引有Hash索引，B+树索引等，而我们经常使用的InnoDB存储引擎的默认索引实现为：B+树索引。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

**1. B树索引**

mysql通过存储引擎取数据，基本上90%的人用的就是InnoDB了，按照实现方式分，InnoDB的索引类型目前只有两种：BTREE（B树）索引和HASH索引。B树索引是Mysql数据库中使用最频繁的索引类型，基本所有存储引擎都支持BTree索引。通常我们说的索引不出意外指的就是（B树）索引（实际是用B+树实现的，因为在查看表索引时，mysql一律打印BTREE，所以简称为B树索引）



**2. B+tree性质**

1、n棵子tree的节点包含n个关键字，不用来保存数据而是保存数据的索引。

2、所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

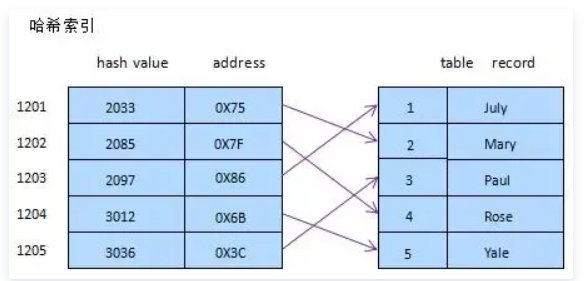
3、所有的非终端结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。

4、B+ 树中，数据对象的插入和删除仅在叶节点上进行。

5、B+树有2个头指针，一个是树的根节点，一个是最小关键码的叶节点。

**3. 哈希索引**

简要说下，类似于数据结构中简单实现的HASH表（散列表）一样，当我们在mysql中用哈希索引时，主要就是通过Hash算法（常见的Hash算法有直接定址法、平方取中法、折叠法、除数取余法、随机数法），将数据库字段数据转换成定长的Hash值，与这条数据的行指针一并存入Hash表的对应位置；如果发生Hash碰撞（两个不同关键字的Hash值相同），则在对应Hash键下以链表形式存储。当然这只是简略模拟图。



**056、索引的基本原理？**

1、索引用来快速地寻找那些具有特定值的记录。如果没有索引，一般来说执行查询时遍历整张表。

2、索引的原理很简单，就是把无序的数据变成有序的查询：

1、把创建了索引的列的内容进行排序；

2、对排序结果生成倒排表；

3、在倒排表内容上拼上数据地址链；

4、在查询的时候，先拿到倒排表内容，再取出数据地址链，从而拿到具体数据；

**057、索引算法有哪些？**

索引算法有 BTree算法和Hash算法；

**1. BTree算法**

BTree是最常用的mysql数据库索引算法，也是mysql默认的算法。因为它不仅可以被用在=,>,>=,<,<=和between这些比较操作符上，而且还可以用于like操作符，只要它的查询条件是一个不以通配符开头的常量。

**2. Hash算法**

Hash Hash索引只能用于对等比较，例如=,<=>（相当于=）操作符。由于是一次定位数据，不像BTree索引需要从根节点到枝节点，最后才能访问到页节点这样多次IO访问，所以检索效率远高于BTree索引。

**058、索引设计的原则？**

1、适合索引的列是出现在where子句中的列，或者连接子句中指定的列。

2、基数较小的类，索引效果较差，没有必要在此列建立索引；

3、使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间；

4、不要过度索引。索引需要额外的磁盘空间，并降低写操作的性能。在修改表内容的时候，索引会进行更新甚至重构，索引列越多，这个时间就会越长。所以只保持需要的索引有利于查询即可。

**059、创建索引的原则？**

索引虽好，但也不是无限制的使用，最好符合一下几个原则

1、最左前缀匹配原则，组合索引非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

2、较频繁作为查询条件的字段才去创建索引；

3、更新频繁字段不适合创建索引；

4、若是不能有效区分数据的列不适合做索引列(如性别，男女未知，最多也就三种，区分度实在太低)；

5、尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

6、定义有外键的数据列一定要建立索引。

7、对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。

8、对于定义为text、image和bit的数据类型的列不要建立索引。

**060、创建索引时需要注意什么？**

1、非空字段：应该指定列为NOT NULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；

2、取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；

3、索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大效率越高。

**061、使用索引查询一定能提高查询的性能吗？**

1、通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。

2、索引需要空间来存储，也需要定期维护， 每当有记录在表中增减或索引列被修改时，索引本身也会被修改。这意味着每条记录的INSERT，DELETE，UPDATE将为此多付出4，5 次的磁盘I/O。 因为索引需要额外的存储空间和处理，那些不必要的索引反而会使查询反应时间变慢。使用索引查询不一定能提高查询性能，索引范围查询(INDEX RANGE SCAN)适用于两种情况:

1、基于一个范围的检索，一般查询返回结果集小于表中记录数的30%；

2、基于非唯一性索引的检索；

**062、百万级别或以上的数据如何删除？**

关于索引：由于索引需要额外的维护成本，因为索引文件是单独存在的文件,所以当我们对数据的增加,修改,删除,都会产生额外的对索引文件的操作,这些操作需要消耗额外的IO,会降低增/改/删的执行效率。所以，在我们删除数据库百万级别数据的时候，查询MySQL官方手册得知删除数据的速度和创建的索引数量是成正比的。

1、所以我们想要删除百万数据的时候可以先删除索引（此时大概耗时三分多钟）；

2、然后删除其中无用数据（此过程需要不到两分钟）；

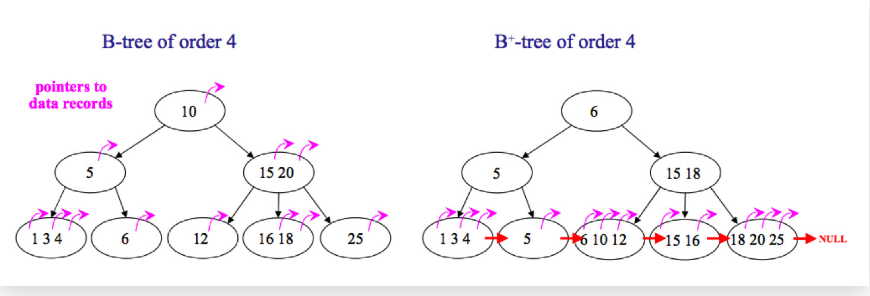
3、删除完成后重新创建索引(此时数据较少了)创建索引也非常快，约十分钟左右；

4、与之前的直接删除绝对是要快速很多，更别说万一删除中断,一切删除会回滚。那更是坑了。

**064、B树和B+树的区别？**

1、在B树中，你可以将键和值存放在内部节点和叶子节点；但在B+树中，内部节点都是键，没有值，叶子节点同时存放键和值。

2、B+树的叶子节点有一条链相连，而B树的叶子节点各自独立。



**065、使用B树的好处？**

B树可以在内部节点同时存储键和值，因此，把频繁访问的数据放在靠近根节点的地方将会大大提高热点数据的查询效率。这种特性使得B树在特定数据重复多次查询的场景中更加高效。

**066、使用B+树的好处？**

由于B+树的内部节点只存放键，不存放值，因此，一次读取，可以在内存页中获取更多的键，有利于更快地缩小查找范围。 B+树的叶节点由一条链相连，因此，当需要进行一次全数据遍历的时候，B+树只需要使用O(logN)时间找到最小的一个节点，然后通过链进行O(N)的顺序遍历即可。而B树则需要对树的每一层进行遍历，这会需要更多的内存置换次数，因此也就需要花费更多的时间

**067、什么是聚簇索引？何时使用聚簇索引与非聚簇索引？**

1、聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据；

2、非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key\_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因。

**068、非聚簇索引一定会回表查询吗？**

1、不一定，这涉及到查询语句所要求的字段是否全部命中了索引，如果全部命中了索引，那么就不必再进行回表查询。

2、举个简单的例子，假设我们在员工表的年龄上建立了索引，那么当进行select age from employee where age < 20的查询时，在索引的叶子节点上，已经包含了age信息，不会再次进行回表查询。

**069、联合索引是什么？为什么需要注意联合索引中的顺序？**

1、MySQL可以使用多个字段同时建立一个索引，叫做联合索引。在联合索引中，如果想要命中索引，需要按照建立索引时的字段顺序挨个使用，否则无法命中索引。

2、MySQL使用索引时需要索引有序，假设现在建立了”name，age，school”的联合索引，那么索引的排序为: 先按照name排序，如果name相同，则按照age排序，如果age的值也相等，则按照school进行排序。

3、当进行查询时，此时索引仅仅按照name严格有序，因此必须首先使用name字段进行等值查询，之后对于匹配到的列而言，其按照age字段严格有序，此时可以使用age字段用做索引查找，以此类推。因此在建立联合索引的时候应该注意索引列的顺序，一般情况下，将查询需求频繁或者字段选择性高的列放在前面。此外可以根据特例的查询或者表结构进行单独的调整。

**072、什么是脏读？幻读？不可重复读？**

1、脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

2、不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

3、幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

**073、什么是事务的隔离级别？MySQL的默认隔离级别是什么？**

为了达到事务的四大特性，数据库定义了4种不同的事务隔离级别，由低到高依次为Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

**SQL 标准定义了四个隔离级别：**

READ-UNCOMMITTED(读取未提交)： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。

READ-COMMITTED(读取已提交)： 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。

REPEATABLE-READ(可重复读)： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

SERIALIZABLE(可串行化)： 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

* **Mysql 默认采用的 REPEATABLE\_READ隔离级别 Oracle 默认采用的 READ\_COMMITTED隔离级别；**

**074、隔离级别与锁的关系？**

1、在Read Uncommitted级别下，读取数据不需要加共享锁，这样就不会跟被修改的数据上的排他锁冲突；

2、在Read Committed级别下，读操作需要加共享锁，但是在语句执行完以后释放共享锁；

3、在Repeatable Read级别下，读操作需要加共享锁，但是在事务提交之前并不释放共享锁，也就是必须等待事务执行完毕以后才释放共享锁。

4、SERIALIZABLE 是限制性最强的隔离级别，因为该级别锁定整个范围的键，并一直持有锁，直到事务完成。

**075、按照锁的粒度分数据库锁有哪些？**

**行级锁:**行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁 和 排他锁。特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

**表级锁:** 表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。特点：开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

**页级锁:**页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

**076、从锁的类别上分MySQL都有哪些锁呢？**

从锁的类别上来讲，有共享锁和排他锁。

共享锁: 又叫做读锁。 当用户要进行数据的读取时，对数据加上共享锁。共享锁可以同时加上多个。

排他锁: 又叫做写锁。 当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

**077、InnoDB存储引擎的锁的算法有哪三种？**

Record lock：单个行记录上的锁；

Gap lock：间隙锁，锁定一个范围，不包括记录本身；

Next-key lock：record+gap 锁定一个范围，包含记录本身；

**078、什么是死锁？怎么解决？**

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。

常见的解决死锁的方法

1、如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。

2、在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；

3、对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

* 如果业务处理不好可以用分布式事务锁或者使用乐观锁

**079、大表数据查询，怎么优化？**

1、优化shema、sql语句+索引；

2、第二加缓存，memcached, redis；

3、主从复制，读写分离；

4、垂直拆分，根据你模块的耦合度，将一个大的系统分为多个小的系统，也就是分布式系统；

5、水平切分，针对数据量大的表，这一步最麻烦，最能考验技术水平，要选择一个合理的sharding key, 为了有好的查询效率，表结构也要改动，做一定的冗余，应用也要改，sql中尽量带sharding key，将数据定位到限定的表上去查，而不是扫描全部的表

**080、超大分页怎么处理？**

超大的分页一般从两个方向上来解决:

1、数据库层面,这也是我们主要集中关注的(虽然收效没那么大),类似于select from table where age > 20 limit 1000000,10这种查询其实也是有可以优化的余地的. 这条语句需要load1000000数据然后基本上全部丢弃,只取10条当然比较慢. 当时我们可以修改为select from table where id in (select id from table where age > 20 limit 1000000,10).这样虽然也load了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,所以速度会很快. 同时如果ID连续的好,我们还可以select \* from table where id > 1000000 limit 10,效率也是不错的,优化的可能性有许多种,但是核心思想都一样,就是减少load的数据

2、从需求的角度减少这种请求…主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止ID泄漏且连续被人恶意攻击

**081、为什么要尽量设定一个主键？**

主键是数据库确保数据行在整张表唯一性的保障，即使业务上本张表没有主键，也建议添加一个自增长的ID列作为主键。设定了主键之后，在后续的删改查的时候可能更加快速以及确保操作数据范围安全。

**082、自增ID还是UUID？**

推荐使用自增ID，不要使用UUID。

1、因为在InnoDB存储引擎中，主键索引是作为聚簇索引存在的，也就是说，主键索引的B+树叶子节点上存储了主键索引以及全部的数据(按照顺序)，如果主键索引是自增ID，那么只需要不断向后排列即可，如果是UUID，由于到来的ID与原来的大小不确定，会造成非常多的数据插入，数据移动，然后导致产生很多的内存碎片，进而造成插入性能的下降。

2、总之，在数据量大一些的情况下，用自增主键性能会好一些。

3、关于主键是聚簇索引，如果没有主键，InnoDB会选择一个唯一键来作为聚簇索引，如果没有唯一键，会生成一个隐式的主键。

**083、字段为什么要求定义为not null？**

null值会占用更多的字节，且会在程序中造成很多与预期不符的情况。

**084、如果要存储用户的密码散列，应该使用什么字段进行存储？**

密码散列，盐，用户身份证号等固定长度的字符串应该使用char而不是varchar来存储，这样可以节省空间且提高检索效率。

**085、MySQL数据库cpu飙升到500%的话他怎么处理？**

1、当 cpu 飙升到 500%时，先用操作系统命令 top 命令观察是不是 mysqld 占用导致的，如果不是，找出占用高的进程，并进行相关处理。

2、如果是 mysqld 造成的， show processlist，看看里面跑的 session 情况，是不是有消耗资源的 sql 在运行。找出消耗高的 sql，看看执行计划是否准确， index 是否缺失，或者实在是数据量太大造成。

3、一般来说，肯定要 kill 掉这些线程(同时观察 cpu 使用率是否下降)，等进行相应的调整(比如说加索引、改 sql、改内存参数)之后，再重新跑这些 SQL。

4、也有可能是每个 sql 消耗资源并不多，但是突然之间，有大量的 session 连进来导致 cpu 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整，比如说限制连接数等。

**086、主从复制的作用？**

1、主数据库出现问题，可以切换到从数据库。

2、可以进行数据库层面的读写分离。

3、可以在从数据库上进行日常备份。

**087、MySQL主从复制解决的问题？**

1、数据分布：随意开始或停止复制，并在不同地理位置分布数据备份；

2、[负载均衡](https://cloud.tencent.com/product/clb?from=10680)：降低单个服务器的压力；

3、高可用和故障切换：帮助应用程序避免单点失败；

4、升级测试：可以用更高版本的MySQL作为从库；

**088、MySQL主从复制工作原理？**

1、在主库上把数据更高记录到二进制日志；

2、从库将主库的日志复制到自己的中继日志；

3、从库读取中继日志的事件，将其重放到从库数据中；。

**089、基于rbac的权限管理？**

RBAC（Role-Based Access Control，基于角色的访问控制），就是用户通过角色与权限进行关联。简单地说，一个用户拥有若干角色，一个角色拥有若干权限。这样，就构造成“用户-角色-权限”的授权模型。在这种模型中，用户与角色之间，角色与权限之间都是多对多的关系。

**090、数据库表的引擎：驱动数据的方式-数据库优化？**

前提: 引擎是建表是规定, 提供给表使用的, 不是数据库

1.展示所有引擎

mysql> show engines;

# 重点:

mysql> use db1;选择数据库

2. innodb(默认): 支持事务, 行级锁, 外键（安全性比较高，加锁）

mysql>: create table t1(id int)engine=innodb;

3. myisam: 查询效率要优于innodb, 当不需要支持事务, 行级锁, 外键, 可以通过设置myisam来优化数据库（速度比较快，无锁）

mysql>: create table t2(id int)engine=myisam;

4. blackhole：黑洞，存进去的数据都会消失（可以理解不存数据）

mysql>: create table t3(id int)engine=blackhole;

5. memory：表结构是存储在硬盘上的，但是表数据全部存储在内存中，重启清空

mysql>: create table t4(id int)engine=memory;

insert into t1 values(1);

insert into t2 values(1);

insert into t3 values(1);

insert into t4 values(1);

select \* from t1; ...

**092、mysql支持的数据类型？**

mysql数据库支持存在那些数据：整型、浮点型、字符型、时间类型、枚举类型、集合类型；

**095、简述Mysql数据库存储的原理？**

储存过程是一个可编程的函数，它在数据库中创建并保存。实际上是用户通过创建存储过程创建好的系统对象，它具有指定的名字、类型（存储过程/函数）及要执行的语句序列等。它可以有SQL语句和一些特殊的控制结构组成。当希望在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟。它允许控制数据的访问方式。 存储过程通常有以下优点：

1、存储过程能实现较快的执行速度 。

2、存储过程允许标准组件是编程。

3、存储过程可以用流程控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

4、存储过程可被作为一种安全机制来充分利用。

5、存储过程能够减少网络流量。

**006、数据库索引？**

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B\_TREE。 B\_TREE 索引加速了数据访问，因为存储引擎不会再去扫描整张表得到需要的数据；相反，它从根节点开始，根节点保存了子节点的指针，存储引擎会根据指针快速寻找数据。

**096、数据库索引原理与种类？**

index的底层原理：

数据库的index的底层原理b+树，速度快，时间复杂度是O(logn)。索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，更通俗的说，数据库索引好比是一本书前面的目录，能加快数据库的查询速度 MySQL索引的类型：

1. 普通索引：这是最基本的索引，它没有任何限制

2. 唯一索引：索引列的值必须唯一，但允许有空值，如果是组合索引，则列值的组合必须唯一

3. 全文索引：全文索引仅可用于 MyISAM 表，可以从CHAR、VARCHAR或TEXT列中作为CREATE TABLE语句的一部分被创建，或是随后使用ALTER TABLE 或CREATE INDEX被添加（切记对于大容量的数据表，生成全文索引是一个非常消耗时间非常消耗硬盘空间的做法）

4. 单列索引、多列索引：多个单列索引与单个多列索引的查询效果不同，因为执行查询时，MySQL只能使用一个索引，会从多个索引中选择一个限制最为严格的索引。

5. 组合索引（最左前缀）：简单的理解就是只从最左面的开始组合（实在单列索引的基础上进一步压榨索引效率的一种方式）

**097、索引在什么情况下遵循最左前缀的规则？**

mysql在使用组合索引查询的时候（在多字段进行索引的时候）需要遵循“最左前缀”规则；

**099、索引有什么作用,有那些分类,有什么好处和坏处?**

作用：索引提供指向存储在表的指定列中的数据值的指针，然后根据您指定的排序顺序对这些指针排序。数据库使用索引以找到特定值，然后顺指针找到包含该值的行。这样可以使对应于表的SQL语句执行得更快，可快速访问数据库表中的特定信息。

分类：

1、唯一索引：唯一索引是不允许其中任何两行具有相同索引值的索引。当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。

2、主键索引：数据库表经常有一列或多列组合，其值唯一标识表中的每一行。该列称为表的主键。在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

3、聚集索引：在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

4、索引列：可以基于数据库表中的单列或多列创建索引。多列索引可以区分其中一列可能有相同值的行。如果经常同时搜索两列或多列或按两列或多列排序时，索引也很有帮助。例如，如果经常在同一查询中为姓和名两列设置判据，那么在这两列上创建多列索引将很有意义。

优点：

1、大大加快数据的检索速度。

2、创建唯一性索引，保证数据库表中每一行数据的唯一性。

3、加速表和表之间的连接。

4、在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间。

缺点：

1、索引需要占物理空间。

2、当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度。

**002 事务的特性？**

1、原子性：事务中的全部操作在数据库中是不 可分割的，要么全部完成，要么均不执行。

2、一致性：几个并行执行的事务，其执行结果必 须与按某一顺序串行执行的结果相一致。

3、隔离性：事务的执行不受其他事务的干扰，事务 执行的中间结果对其他事务必须是透明的。

4、持久性：对于任意已提交事务，系统必须保证该 事务对数据库的改变不被丢失，即使数据库出现故障

**003、readis 和 mysql 的区别？**

readis 是内存数据库，数据保存在内存中，速度快。 mysql 是关系型数据库，持久化存储，存放在磁盘里面，功能强 大。检索的话，会涉及到一定的 IO，数据访问也就慢。

**004、redis 受攻击怎么办？**

1 设置主从配置；

2持久化存储；

3 Redis 不以 root 账户启动；

3 设置复杂密码；

4不允许 key 方式登录；

**005、MongoDB？**

MongoDB 是一个面向文档的数据库系统。使用 C++编写，不支持 SQL，但有自己功能强大的查询语法。

MongoDB 使用 BSON 作为数据存储和传输的格式。BSON 是一种类似JSON 的二进制序列化文档，支持嵌套对象和数组。

**9  Redis、mongodb 优缺点？**

MongoDB 建议集群部署，更多的考虑到 集群方案，Redis 更偏重于进程顺序写入，虽然支持集群，也仅限于 主-从模式.

Redis 优点：

         1 读写性能优异 2 支持数据持久化3 支持主从复制

缺点：

        1 Redis 不具备自动容错和恢复功能

2 Redis 较难支持在线扩容

3 Redis 的主从复制采用全量复制

mongodb优点：

         1弱一致性，更能保证用户的访问速度 文档结构的存储方式，能够更便捷的获取数

2高效存储二进制大对象 (比如照片和视频)

3支持复制集、主备、互为主备、自动分片等特性 动态查询 全索引支持,扩展到内部对象和内嵌数组

缺点：不支持事务 MongoDB 占用空间过大 维护工具不够成熟

**10 怎样解决海量数据的存储和访问造成系统设计瓶颈的问题？**

水平切分数据库：可以降低单台机器的负载，同时最大限度的降 低了宕机造成的损失；

负载均衡策略：可以降低单台机器的访问负载，降低宕机的可能性；

集群方案：解决了数据库宕机带来的单点数据库不能访问的问题；

读写分离策略：最大限度了提高了应用中读取数据的速度和并发量；

**011、MySQL 集群的优缺点？**

优点：

1 99.999%的高可用性

2快速的自动失效切换

3灵活的分布式体系结构，没有单点故障

4高吞吐量和低延迟

5可扩展性强，支持在线扩容

缺点：

1存在很多限制，比如：不支持外键

2部署、管理、配置很复杂

3占用磁盘空间大，内存大

4备份和恢复不方便

5重启的时候，数据节点将数据 load 到内存需要很长时间

**013、redis 基本类型、相关方法？**

Redis支持五种数据类型：string（字符串）、hash（哈希）、list（列表）、set （集合）及zset(sorted set：有序集合)。

String常用命令: set,get,decr,incr,mget 等

hash常用命令：hget,hset,hgetall 等。

list常用命令：lpush,rpush,lpop,rpop,lrange等

set常用命令：sadd,spop,smembers,sunion等

zset常用命令：zadd,zrange,zrem,zcard 等

**014、怎样解决数据库高并发的问题？**

1.分表分库；2. 数据库索引；3. redis 缓存数据库；4. 读写分离；5. 负载均衡集群：

**015 、sqlserver，MySQL ，Oracle http，redis，https 默认端 口号？**

         Sqlserver:1433  MySQL:3306  Oracle:1521  http:40  https:443  redis:6379

## 详述[Mysql优化](https://www.cnblogs.com/jiaxinzhu/p/12466817.html)大全

* [1.1 Mysql数据库的优化技术](https://www.cnblogs.com/jiaxinzhu/p/12466817.html" \l "_label0)
* [1.2 数据库表设计](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label1)
* [1.3 SQL优化](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label2)
* [1、为查询缓存优化你的查询](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label3)
* [2、EXPLAIN 你的 SELECT 查询](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label4)
* [3、 当只要一行数据时使用 LIMIT 1](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label5)
* [4、建立适当的索引](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label6)
* [1.4 MySQL中like模糊匹配为何低效](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label7)
* [1.5 数据库优化方案](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label8)
* [1.6 数据库怎么优化查询效率](https://www.cnblogs.com/jiaxinzhu/p/12466817.html#_label9)
  1. **Mysql数据库的优化技术**

**0、要保证数据库的效率，要做好以下四个方面的工作？**

01. 数据库设计；  
02. sql语句优化；  
03. 数据库参数配置；  
04. 恰当的硬件资源和操作系统；

此外，使用适当的存储过程，也能提升性能。  
这个顺序也表现了这四个工作对性能影响的大小；

**1.2 数据库表设计**

**1、通俗地理解三个范式**

　　　　　　第一范式： 1NF是对属性的原子性约束，要求属性(列)具有原子性，不可再分解；(只要是关系型数据库都满足1NF)

　　　　　　第二范式： 2NF是对记录的惟一性约束，要求记录有惟一标识，即实体的惟一性；

　　　　　　第三范式： 3NF是对字段冗余性的约束，它要求字段没有冗余。 没有冗余的数据库设计可以做到。

**2、第一范式（1NF）**

　　　　　　1. 即表的列的具有原子性,不可再分解，即列的信息，不能分解, 只要数据库是关系型数据库(mysql/oracle/db2/informix/sysbase/sql server)，就自动的满足1NF。  
　　　　　　2. 数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。  
　　　　　　3. 如果实体中的某个属性有多个值时，必须拆分为不同的属性 。通俗理解即一个字段只存储一项信息。

**3、第二范式（2NF）**

　　　　　　1、要求数据库表中的每个实例或行必须可以被惟一地区分。为实现区分通常需要我们设计一个主键来实现(这里的主键不包含业务逻辑)。

**4、第三范式（3NF）**

　　　　　　1、要求一个数据库表中不包含已在其它表中已包含的非主键字段。  
　　　　　　2、如果能够被推导出来，就不应该单独的设计一个字段来存放(能尽量外键join就用外键join)。  
　　　　　　3、很多时候，我们为了满足第三范式往往会把一张表分成多张表。

**1.3 SQL优化**

1、为查询缓存优化你的查询

　　　　1、大多数的MySQL服务器都开启了查询缓存。这是提高性最有效的方法之一，而且这是被MySQL的数据库引擎处理的。  
　　　　2、当有很多相同的查询被执行了多次的时候，这些查询结果会被放到一个缓存中，这样，后续的相同的查询就不用操作表而直接访问缓存结果了。  
　　　　3、像 NOW() 和 RAND() 或是其它的诸如此类的SQL函数都不会开启查询缓存，因为这些函数的返回是会不定的易变的。

2、EXPLAIN 你的 SELECT 查询

**1、explain关键字作用**

　　　　　　1、使用 EXPLAIN 关键字可以让你知道MySQL是如何处理你的SQL语句的。这可以帮你分析你的查询语句或是表结构的性能瓶颈。  
　　　　　　2、EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的……等等，等等。

**2、explain使用举例**

　　　　　　Explain select \* from emp where ename=“wsrcla” 会产生如下信息：  
　　　　　　　select\_type: 表示查询的类型。  
　　　　　　　table:  输出结果集的表  
　　　　　　　type:  表示表的连接类型  
　　　　　　　possible\_keys:   表示查询时，可能使用的索引  
　　　　　　　key:   表示实际使用的索引  
　　　　　　　key\_len:   索引字段的长度  
　　　　　　　rows:   扫描出的行数(估算的行数)  
　　　　　　　Extra:   执行情况的描述和说明

**3、EXPLAIN信息详解**

**1. id**

SELECT识别符。这是SELECT的查询序列号

**2. select\_type**

　　　　　　　　PRIMARY :   子查询中最外层查询  
　　　　　　　　SUBQUERY :    子查询内层第一个SELECT，结果不依赖于外部查询  
　　　　　　　　DEPENDENT SUBQUERY:   子查询内层第一个SELECT，依赖于外部查询  
　　　　　　　　UNION :   UNION语句中第二个SELECT开始后面所有SELECT，  
　　　　　　　　SIMPLE：  简单的 select 查询,不使用 union 及子查询  
　　　　　　　　UNION ：  UNION 中的第二个或随后的 select 查询,不依赖于外部查询的结果集

**3. Table ：**显示这一步所访问数据库中表名称

**4. Type ： 对表访问方式**

　　　　　　　　ALL：  SELECT \* FROM emp \G 完整的表扫描 通常不好  
　　　　　　　　SELECT \* FROM (SELECT \* FROM emp WHERE empno = 1) a ;  
　　　　　　　　system：  表仅有一行(=系统表)。这是const联接类型的一个特  
　　　　　　　　const：  表最多有一个匹配行

**5. Possible\_keys ：**该查询可以利用的索引，如果没有任何索引显示  null

**6. Key ：**Mysql 从 Possible\_keys 所选择使用索引

**7. Rows ：**估算出结果集行数

**8. Extra查询细节信息**

No tables ：  Query语句中使用FROM DUAL 或不含任何FROM子句；  
Using filesort ：  当Query中包含 ORDER BY 操作，而且无法利用索引完成排序，  
Impossible WHERE noticed after reading const tables: MYSQL Query Optimizer  
通过收集统计信息不可能存在结果  
Using temporary：  某些操作必须使用临时表，常见 GROUP BY ; ORDER BY；  
Using where：  不用读取表中所有信息，仅通过索引就可以获取所需数据;

3、 当只要一行数据时使用 LIMIT 1

　　　　1、当你查询表的有些时候，你已经知道结果只会有一条结果，但因为你可能需要去fetch游标，或是你也许会去检查返回的记录数。  
　　　　2、在这种情况下，加上 LIMIT 1 可以增加性能。这样一样，MySQL数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据。

4、建立适当的索引

**1、索引为什会使查找变快**

　　　　　　1、btree类型的索引，就是使用的二分查找法，肯定快啊，算法复杂度是log2N，也就是说16条数据查4次，32条数据查5次，64条数据查6次....依次类推。  
　　　　　　2、使用索引跟没使用索引的区别，就跟我们使用新华字典查字，一个是根据拼音或者笔画查找，一个是从头到尾一页一页翻。

**2、索引的代价**

　　　　　　1、磁盘占用  
　　　　　　2、对dml(update delete insert)语句的效率影响

**3、索引使用原则**

　　　　　　1、较频繁的作为查询条件字段应该创建索引  
　　　　　　　　　　select \* from emp where empno = 1;

　　　　　　2、唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件  
　　　　　　　　　　select \* from emp where sex = '男'

　　　　　　3、更新非常频繁的字段不适合创建索引  
　　　　　　　　　　select \* from emp where logincount = 1

　　　　　　4、不会出现在WHERE子句中的字段不该创建索引

**4、mysql四种索引的区别**

1、主键索引，主键自动的为主索引 (类型Primary)  
2、唯一索引 (UNIQUE)  
3、普通索引 (INDEX)  
4、全文索引 (FULLTEXT) [适用于MyISAM] ——》sphinx + 中文分词 coreseek [sphinx 的中文版 ]

**5、索引的使用**

**1. 建立索引**

　　　　　　　　1、create [UNIQUE|FULLTEXT] index index\_name on tbl\_name (col\_name [(length)] [ASC | DESC] , …..);  
　　　　　　　　2、alter table table\_name ADD INDEX [index\_name] (index\_col\_name,...)

**2.  删除索引**

　　　　　　　　1、DROP INDEX index\_name ON tbl\_name;  
　　　　　　　　2、alter table table\_name drop index index\_name;

　　　　　　　　注：删除主键(索引)比较特别: alter table t\_b drop primary key;

**3. 创建普通索引方法**

#1 查看student表中有哪些索引  
　　　　　　　　　　mysql> show index from student;                                   #查看student表中有哪些索引

　　　　　　#2 创建最基本的的索引  
　　　　　　　　　　mysql> create index index\_name on student(name(32));              #将student中字段name创建成索引

　　　　　　#3 删除索引的语法  
　　　　　　　　　　mysql> drop index index\_name on student;

**4. 创建唯一索引**

注： 它与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值；

#1创建索引：  
mysql>  create unique index index\_name on student(name(32));

**6、使用或不使用索引的情况**

**1. 下列几种情况下有可能使用到索引**

　　　　　　　　1，对于创建的多列索引，只要查询条件使用了最左边的列，索引一般就会被使用。  
　　　　　　　　2，对于使用like的查询，查询如果是  '%aaa' 不会使用到索引， 'aaa%' 会使用到索引。

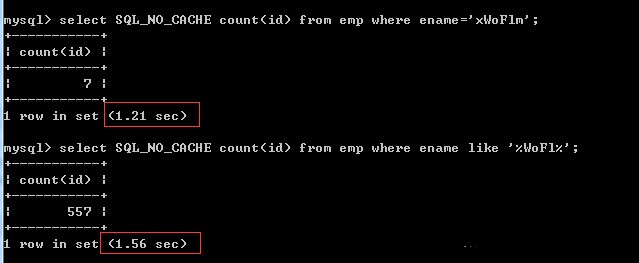
**2. 下列的表将不使用索引**

　　　　　　　　1， 如果条件中有or，即使其中有条件带索引也不会使用。  
　　　　　　　　2， 对于多列索引，不是使用的第一部分，则不会使用索引。  
　　　　　　　　3， like查询是以%开头  
　　　　　　　　4， 如果列类型是字符串，那一定要在条件中将数据使用引号引用起来。否则不使用索引。(添加时,字符串必须'')  
　　　　　　　　5， 如果mysql估计使用全表扫描要比使用索引快，则不使用索引。

**1.4 MySQL中like模糊匹配为何低效？**

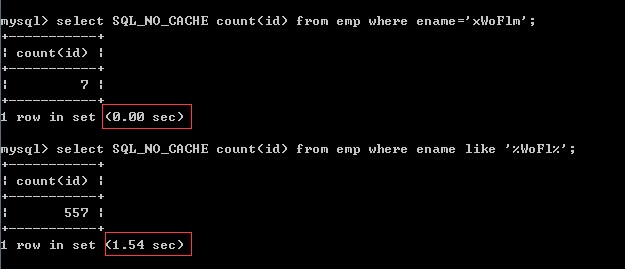
**1、都不使用索引**

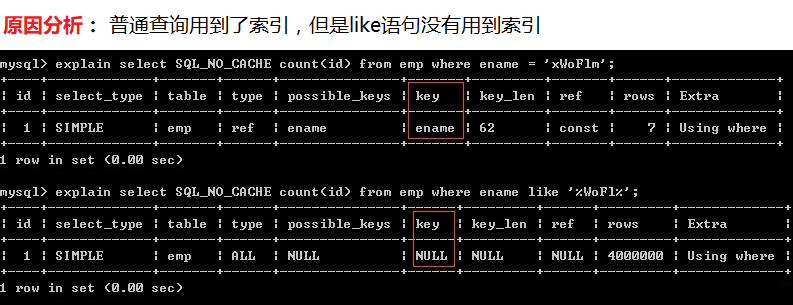
**说明：**不使用索引的时候普通查询与like查询的耗时相当，like略长，这也是必然的，因为它要进行额外的算法。



**2、like不使用索引、普通查询使用索引**

**说明：**使用索引后，普通查询的耗时基本算是秒查，非常快；而like查询还是耗时一秒多。

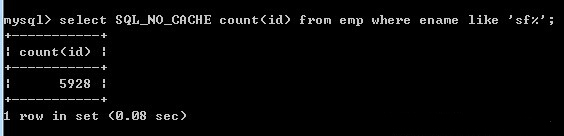




**3、like查询起始也可以使用索引**

　　　　　　1）如果我们查询的时候写成“like 'dd\_'或者like 'dd%'”，这样是可以用到索引的，此时的查询速度也会相对的快一点。

　　　　　　2）如果查询的时候写成“ like '%dd'”,查询字符串最前面就是模糊的无法使用索引



**1.5 数据库优化方案？**

01、优化索引、SQL 语句、分析慢查询；

02. 表的设计合理化(符合3NF)，设计表的时候严格根据数据库的设计范式来设计数据库；

03. 使用缓存，把经常访问到的数据而且不需要经常变化的数据放在缓存中，能节约磁盘IO；

04. 优化硬件，mysql服务器硬件升级：采用SSD，使用磁盘队列技术(RAID0,RAID1,RDID5)等；

05. 采用MySQL 内部自带的表分区技术，把数据分层不同的文件，能够提高磁盘的读取效率；

06. 分表技术(水平分割、垂直分割)，垂直分表；把一些不经常读的数据放在一张表里，节约磁盘I/O；

07. 主从分离读写[写: update/delete/add]分离；采用主从复制把数据库的读操作和写入操作分离开来；

08. 分库分表分机器（数据量特别大），主要的的原理就是数据路由；

09. 选择合适的表引擎，参数上的优化；

10. 进行架构级别的缓存，静态化和分布式；

11. 不采用全文索引；

12、减少数据访问-创建并正确使用索引；

13、返回更少的数据；

14、数据分页处理；

15、只返回需要的字段；

16、减少交互次数；

17、使用存储过程[模块化编程，可以提高速度]，采用更快的存储方式，例如 NoSQL存储经常访问的数据\*\*；

18、优化业务逻辑；

19、减少服务器的cpu运算；

20、使用绑定变量；

21、合理使用排序；

22、减少比较操作；

23、大量复杂运算在客户端处理；

24、利用更多资源；

25、数据库并行处理，客户端多进程访问；  
26、添加适当索引(index) [四种: 普通索引、主键索引、唯一索引unique、全文索引]；  
27、对mysql配置优化 [配置最大并发数my.ini, 调整缓存大小 ]；  
28、定时的去清除不需要的数据,定时进行碎片整理(MyISAM)；

**016、数据库查询优化？**

主要通过三种途径提高数据库查询效率：1.SQL的性能优化，2.数据库的性能优化，3.查询优化

光说SQL语句,就要注意:  
1、不要写效率低的语句，特别是有些复杂的，如果效率低就坚决不用，考虑用其他方法代替。  
2、注意配合索引；  
3、存储过程和触发器的效率是由存储过程和触发器内的语句的效率决定的，特别是触发器，如果效率不好会影响整个服务器的性能。

01、储存引擎选择：如果数据表需要事务处理，应该考虑使用InnoDB，因为它完全符合ACID特性。 如果不需要事务处理，使用默认存储引擎MyISAM是比较明智的

02、分表分库，主从。

03、对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引；

04、应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描 ；

05、应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描 ；

06、应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描 ；

07、Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志 ；

08、对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

09、尽量避免大事务操作，提高系统并发能力。

10、尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

**017、mysql存储过程：MySQL PROCEDURE？**

存储过程（PROCEDURE）是存储在服务器中的一组指定的 SQL 语句，客户机无需不断重新发出单个语句，而可以引用存储过程。存储过程类型分为存储过程 (PROCEDURE) 和存储函数 (FUNCTION) 。

Ø   存储过程：通过 CALL 语句调用过程。这些过程可以使用输出变量或结果集传回值。

Ø   存储函数：在语句中调用函数。这些函数可返回标量值。

l   使用存储过程的好处：

1)          客户机功能

通过存储过程，您可以在数据库中集中创建一个语句或一系列语句，以供使用不同编程语言编写或在不同平台上运行的多个客户机应用程序使用。

2)          安全性

存储过程为需要最高安全级别的应用程序提供了一个解决方法。例如，银行针对所有常用操作均使用存储过程和函数。这提供了一致、安全的环境。可对存储过程进行编码，以确保正确记录了每个操作。在此类设置中，应用程序和用户无法直接访问数据库表，只能执行特定的存储过程。

3)          性能

因为服务器和客户机之间需要发送的信息变少了，所以存储过程可提升性能。客户机按名称调用存储过程，而不是传递该存储过程所包含的所有语句。

4)          函数库

通过存储过程，可以在数据库服务器中使用函数库。这些库用作数据库的 API 。

l   使用存储过程的问题：

1)          增加了服务器负载

在数据库自身中执行存储过程可增加服务器负载并降低应用程序的性能。可以运行测试并运用常识来确保在数据库本身中包含逻辑所带来的方便比可能引发的性能问题更为显著。

2)          开发工具有限

MySQL 中支持存储过程的开发工具不像在更通用的编程语言中那样成熟和明确。此局限性会使存储过程的编写和调试过程更加困难，在决策过程中需要加以考虑。

3)          语言功能和速度有限

虽然在许多情况下在数据库本身中包含逻辑具有很大的优势，但是与其他编程语言相比，在可实现的内容方面仍有局限。存储过程在数据库上下文中执行，与客户机应用程序中的存储过程相比，在处理大量数据时性能较好，但是客户机应用程序语言可能具有更强大、更通用的处理、集成或其他库功能。您必须考虑所需功能的范围，以确保针对每个存储过程使用最佳的可能解决方案。

4)          调试和概要分析功能有限

1.1.         执行存储过程

用于调用存储过程的命令与 MySQL 中的其他命令非常类似。使用 CALL 语句来调用存储过程（ procedure ）。存储过程使用输出变量或结果集传回值。执行 FUNCTION ，像其他任何函数一样，从语句内部调用函数（即，通过调用相应函数的名称），函数返回标量值。

每个存储过程均与特定数据库相关联。这有多重含义：

Ø   USE <database> ：调用存储过程时， MySQL 会在该存储过程运行期间执行隐式 USE <database> 。不能在存储过程内发出 USE 语句。

Ø   限定名称：可使用存储过程的数据库名称限定存储过程名称。执行此操作可引用当前数据库以外的存储过程。例如，要调用与 test 数据库相关联的存储过程 p 或函数 f ，请使用 CALL test.p() 或 test.f() 。

Ø   数据库删除：删除数据库时，也会删除与其关联的所有存储过程。

MySQL 允许在存储过程内使用常规 SELECT 语句。此类查询的结果集将直接发送到客户机。

1)          存储过程示例

mysql> DELIMITER //

mysql> CREATE PROCEDURE record\_count ()

-> BEGIN

-> SELECT 'Country count ', COUNT(\*) FROM Country;

-> SELECT 'City count ', COUNT(\*) FROM City;

-> SELECT 'CountryLanguage count', COUNT(\*) FROM CountryLanguage;

-> END//

mysql> DELIMITER ;

Ø   复合语句

通过在存储过程中使用 BEGIN … END 语法并使用触发器，可以创建复合语句。 BEGIN … END 块可包含零个或多个语句。空复合语句是合法的，而且复合语句中的语句数量没有上限。

Ø   分隔符

在 BEGIN … END 语法中，必须使用分号 (;) 终止每个语句。由于 mysql 客户机使用分号作为 SQL 语句的默认终止字符，在以交互方式或针对批处理使用 mysql 命令行客户机时，必须使用 DELIMITER 语句更改此设置。

示例中，第一个 DELIMITER 语句用于将 SQL 语句终止字符更改为两个正斜杠 (//) 。此更改可确保客户机不会将复合语句中的分号解释为语句分隔符，并确保客户机不会过早地将 CREATE PROCEDURE 语句发送到服务器。当创建存储过程的语句以 // 终止时，客户机会先将该语句发送到服务器，然后再发出第二个 DELIMITER 语句将语句分隔符重置为分号。

2)          存储函数：示例

mysql> DELIMITER //

mysql> CREATE FUNCTION pay\_check (gross\_pay FLOAT(9,2), tax\_rate FLOAT (3,2))

-> RETURNS FLOAT(9,2)

-> NO SQL

-> BEGIN

-> DECLARE net\_pay FLOAT(9,2)

-> DEFAULT 0;

-> SET net\_pay=gross\_pay - gross\_pay \* tax\_rate;

-> RETURN net\_pay;

-> END//

mysql> DELIMITER ;

Ø   RETURNS 子句

RETURNS 子句用于确定此函数要返回的值的类型。

Ø   特征

通过多个特征，可确定有关存储函数所使用的数据的性质。在 MySQL 中，这些特征仅供参考。服务器不会使用这些特征来限制允许存储函数执行的语句种类。

l   CONTAINS SQL 表示存储函数包含用于读取或写入数据的语句。如果未显式提供以上任何特征，则此为默认值。

l   NO SQL 表示存储函数不包含任何 SQL 语句。

l   READS SQL DATA 表示存储函数包含用于读取数据的语句（例如， SELECT ）而不包含用于写入数据的语句。

l   MODIFIES SQL DATA 表示存储过程包含用于写入数据的语句（例如， INSERT 或 DELETE ）。

注：在启用了二进制日志记录后，如果创建函数时未指定以下项之一，则 MySQL 会产生一个错误： NO SQL 、 READS SQL DATA 或 DETERMINISTIC 。

Ø   DECLARE 语句

在存储过程中使用 DECLARE 语句来声明本地变量并初始化用户变量。可将 DEFAULT 子句添加到 DECLARE 语句的结尾，以便为用户变量指定初始值。如果省去 DEFAULT 子句，则用户变量的初始值为 NULL 。

Ø   SET 语句

通过 SET 语句，您可以使用 = 或 := 作为赋值运算符来向定义的变量赋值。

Ø   RETURN 语句

RETURN 语句用于终止存储函数的执行，并将值表达式返回给函数调用方。

1.2.         检查存储过程

Ø   SHOW CREATE PROCEDURE 和 SHOW CREATE FUNCTION

这些语句为 MySQL 扩展，类似于 SHOW CREATE TABLE 。这些语句返回可用于重新创建指定存储过程的具体字符串。这些语句的主要限制之一是您必须知道过程或函数的名称，并且必须确定其为过程或函数，然后才能尝试查看相应信息。

Ø   SHOW PROCEDURE STATUS 和 SHOW FUNCTION STATUS

这些语句特定于 MySQL 。它们可返回存储过程的特征，如数据库、名称、类型、创建者以及创建和修改日期。这些语句有一个优点：可基于 LIKE 模式显示特定存储过程。如果未指定任何模式，则会根据所使用的语句，列出所有存储过程或所有存储函数的信息。例如，以下语句显示名称以“ film ”开头的过程的相关信息：

SHOW PROCEDURE STATUS LIKE 'film%'\G

Ø   INFORMATION\_SCHEMA.ROUTINES

INFORMATION\_SCHEMA.ROUTINES 表包含存储过程（过程和函数）的相关信息，并返回可同时在 SHOW CREATE … 和 SHOW … STATUS 语句中找到的大部分详细信息，以包含用于创建存储过程的实际语法。在这三个选项中，此表可完整地呈现数据库中的可用存储过程。

示例：

mysql> SELECT routine\_name, routine\_schema, routine\_type, definer

> FROM INFORMATION\_SCHEMA.ROUTINES

> WHERE routine\_name LIKE 'film%';

+-------------------+----------------+--------------+----------------+

| routine\_name | routine\_schema | routine\_type | definer |

+-------------------+----------------+--------------+----------------+

| film\_in\_stock | sakila | PROCEDURE | root@localhost |

| film\_not\_in\_stock | sakila | PROCEDURE | root@localhost |

+-------------------+----------------+--------------+----------------+

2 rows in set (0.00 sec)

Ø   mysql 系统数据库中与编程组件关联的表

mysql 系统数据库中包含的一些表可提供与 MySQL 存储过程功能相关的信息。这些表包括：

l   mysql.event 表，包含 MySQL 服务器中所存储事件的相关信息；

l   mysql.proc 表，包含 MySQL 服务器中的存储过程和函数的相关信息；

l   mysql.procs\_priv 表，为引用存储过程的用户提供访问控制授予详细信息；

1.3.         存储过程和执行安全性

存储过程和函数的使用涉及多个权限。

默认操作：创建存储过程时， MySQL 会自动向您的帐户授予对该存储过程的 EXECUTE 和 ALTER ROUTINE 权限。拥有撤消权限以及 GRANT OPTION 权限的用户稍后可撤消或删除这些权限。在创建存储过程后，可以通过发出 SHOW GRANTS 语句来验证这些权限。

授予权限：当在全局级别或数据库级别授予所有权限时， GRANT ALL 语句包括除 GRANT OPTION 之外的所有存储过程权限。要授予 GRANT OPTION 权限，请在该语句结尾包含 WITH GRANT OPTION 子句。您可以在单个存储过程级别授予 EXECUTE 、 ALTER ROUTINE 和 GRANT OPTION 权限，但仅限于已经存在的存储过程。要授予对单个存储过程的权限，可使用其数据库名称限定存储过程，并提供关键字 PROCEDURE 或 FUNCTION 以指示存储过程类型，如以下示例中所示：

mysql> GRANT EXECUTE, ALTER ROUTINE ON PROCEDURE world\_innodb.record\_count TO 'magellan'@'localhost' WITH GRANT OPTION;

mysql> GRANT ALL ON world\_innodb.\* TO 'magellan'@'localhost';

mysql> GRANT EXECUTE, ALTER ROUTINE ON PROCEDURE world\_innodb.record\_count TO 'magellan'@'localhost';

权限对应允许的操作

CREATE ROUTINE ：创建存储过程。

ALTER ROUTINE ：更改或删除存储过程。

EXECUTE ：执行存储过程。

GRANT OPTION ：将权限授予其他帐户。

**mysql存储过程详解：**

1.     存储过程简介

我们常用的操作数据库语言SQL语句在执行的时候需要要先编译，然后执行，而存储过程（Stored Procedure）是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

一个存储过程是一个可编程的函数，它在数据库中创建并保存。它可以有SQL语句和一些特殊的控制结构组成。当希望在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟。它允许控制数据的访问方式。

存储过程通常有以下优点：

(1).存储过程增强了SQL语言的功能和灵活性。存储过程可以用流控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

(2).存储过程允许标准组件是编程。存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。

(3).存储过程能实现较快的执行速度。如果某一操作包含大量的Transaction-SQL代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在每次运行时都要进行编译和优化，速度相对要慢一些。

(4).存储过程能过减少网络流量。针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织程存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，从而大大增加了网络流量并降低了网络负载。

(5).存储过程可被作为一种安全机制来充分利用。系统管理员通过执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全。

2.     关于MySQL的存储过程

存储过程是数据库存储的一个重要的功能，但是MySQL在5.0以前并不支持存储过程，这使得MySQL在应用上大打折扣。好在MySQL 5.0终于开始已经支持存储过程，这样即可以大大提高数据库的处理速度，同时也可以提高数据库编程的灵活性。

3.     MySQL存储过程的创建

(1). 格式

MySQL存储过程创建的格式：CREATE PROCEDURE 过程名 ([过程参数[,...]])  
[特性 ...] 过程体

这里先举个例子：

mysql> DELIMITER //

mysql> CREATE PROCEDURE proc1(OUT s int)

    -> BEGIN

    -> SELECT COUNT(\*) INTO s FROM user;

    -> END

    -> //

mysql> DELIMITER ;

注：

（1）这里需要注意的是DELIMITER //和DELIMITER ;两句，DELIMITER是分割符的意思，因为MySQL默认以";"为分隔符，如果我们没有声明分割符，那么编译器会把存储过程当成SQL语句进行处理，则存储过程的编译过程会报错，所以要事先用DELIMITER关键字申明当前段分隔符，这样MySQL才会将";"当做存储过程中的代码，不会执行这些代码，用完了之后要把分隔符还原。

（2）存储过程根据需要可能会有输入、输出、输入输出参数，这里有一个输出参数s，类型是int型，如果有多个参数用","分割开。

（3）过程体的开始与结束使用BEGIN与END进行标识。

这样，我们的一个MySQL存储过程就完成了，是不是很容易呢?看不懂也没关系，接下来，我们详细的讲解。

(2). 声明分割符

其实，关于声明分割符，上面的注解已经写得很清楚，不需要多说，只是稍微要注意一点的是：如果是用MySQL的Administrator管理工具时，可以直接创建，不再需要声明。

(3). 参数

MySQL存储过程的参数用在存储过程的定义，共有三种参数类型,IN,OUT,INOUT,形式如：

CREATE PROCEDURE([[IN |OUT |INOUT ] 参数名 数据类形...])

IN 输入参数:表示该参数的值必须在调用存储过程时指定，在存储过程中修改该参数的值不能被返回，为默认值

OUT 输出参数:该值可在存储过程内部被改变，并可返回

INOUT 输入输出参数:调用时指定，并且可被改变和返回

Ⅰ. IN参数例子

创建:

mysql > DELIMITER //

mysql > CREATE PROCEDURE demo\_in\_parameter(IN p\_in int)

-> BEGIN

-> SELECT p\_in;

-> SET p\_in=2;

-> SELECT p\_in;

-> END;

-> //

mysql > DELIMITER ;   
执行结果:

mysql > SET @p\_in=1;

mysql > CALL demo\_in\_parameter(@p\_in);

+------+

| p\_in |

+------+

|   1  |

+------+

+------+

| p\_in |

+------+

|   2  |

+------+

mysql> SELECT @p\_in;

+-------+

| @p\_in |

+-------+

|  1    |

+-------+    
以上可以看出，p\_in虽然在存储过程中被修改，但并不影响@p\_id的值

Ⅱ.OUT参数例子

创建:

mysql > DELIMITER //

mysql > CREATE PROCEDURE demo\_out\_parameter(OUT p\_out int)

-> BEGIN

-> SELECT p\_out;

-> SET p\_out=2;

-> SELECT p\_out;

-> END;

-> //

mysql > DELIMITER ;   
执行结果:

mysql > SET @p\_out=1;

mysql > CALL sp\_demo\_out\_parameter(@p\_out);

+-------+

| p\_out |

+-------+

| NULL  |

+-------+

+-------+

| p\_out |

+-------+

|   2   |

+-------+

mysql> SELECT @p\_out;

+-------+

| p\_out |

+-------+

|   2   |

+-------+    
Ⅲ. INOUT参数例子

创建:

mysql > DELIMITER //

mysql > CREATE PROCEDURE demo\_inout\_parameter(INOUT p\_inout int)

-> BEGIN

-> SELECT p\_inout;

-> SET p\_inout=2;

-> SELECT p\_inout;

-> END;

-> //

mysql > DELIMITER ;

执行结果:

mysql > SET @p\_inout=1;

mysql > CALL demo\_inout\_parameter(@p\_inout) ;

+---------+

| p\_inout |

+---------+

|    1    |

+---------+

+---------+

| p\_inout |

+---------+

|    2    |

+---------+

mysql > SELECT @p\_inout;

+----------+

| @p\_inout |

+----------+

|    2     |

+----------+

(4). 变量

Ⅰ. 变量定义

DECLARE variable\_name [,variable\_name...] datatype [DEFAULT value];

其中，datatype为MySQL的数据类型，如:int, float, date, varchar(length)

例如:

DECLARE l\_int int unsigned default 4000000;

DECLARE l\_numeric number(8,2) DEFAULT 9.95;

DECLARE l\_date date DEFAULT '1999-12-31';

DECLARE l\_datetime datetime DEFAULT '1999-12-31 23:59:59';

DECLARE l\_varchar varchar(255) DEFAULT 'This will not be padded';

Ⅱ. 变量赋值

 SET 变量名 = 表达式值 [,variable\_name = expression ...]

Ⅲ. 用户变量

ⅰ. 在MySQL客户端使用用户变量

mysql > SELECT 'Hello World' into @x;

mysql > SELECT @x;

+-------------+

|   @x        |

+-------------+

| [Hello World](http://www.chengxusheji.com/) |

+-------------+

mysql > SET @y='Goodbye Cruel World';

mysql > SELECT @y;

+---------------------+

|     @y              |

+---------------------+

| Goodbye Cruel World |

+---------------------+

mysql > SET @z=1+2+3;

mysql > SELECT @z;

+------+

| @z   |

+------+

|  6   |

+------+

ⅱ. 在存储过程中使用用户变量

mysql > CREATE PROCEDURE GreetWorld( ) SELECT CONCAT(@greeting,' World');

mysql > SET @greeting='Hello';

mysql > CALL GreetWorld( );

+----------------------------+

| CONCAT(@greeting,' World') |

+----------------------------+

|  Hello World               |

+----------------------------+

ⅲ. 在存储过程间传递全局范围的用户变量

mysql> CREATE PROCEDURE p1()   SET @last\_procedure='p1';

mysql> CREATE PROCEDURE p2() SELECT CONCAT('Last procedure was ',@last\_proc);

mysql> CALL p1( );

mysql> CALL p2( );

+-----------------------------------------------+

| CONCAT('Last procedure was ',@last\_proc |

+-----------------------------------------------+

| Last procedure was p1                         |

+-----------------------------------------------+

注意:

①用户变量名一般以@开头

②滥用用户变量会导致程序难以理解及管理

(5). 注释

MySQL存储过程可使用两种风格的注释

双模杠：--

该风格一般用于单行注释

c风格： 一般用于多行注释

例如：

mysql > DELIMITER //

mysql > CREATE PROCEDURE proc1 --name存储过程名

-> (IN parameter1 INTEGER)

-> BEGIN

-> DECLARE variable1 CHAR(10);

-> IF parameter1 = 17 THEN

-> SET variable1 = 'birds';

-> ELSE

-> SET variable1 = 'beasts';

-> END IF;

-> INSERT INTO table1 VALUES (variable1);

-> END

-> //

mysql > DELIMITER ;

4.     MySQL存储过程的调用

用call和你过程名以及一个括号，括号里面根据需要，加入参数，参数包括输入参数、输出参数、输入输出参数。具体的调用方法可以参看上面的例子。

5.     MySQL存储过程的查询

我们像知道一个数据库下面有那些表，我们一般采用show tables;进行查看。那么我们要查看某个数据库下面的存储过程，是否也可以采用呢？答案是，我们可以查看某个数据库下面的存储过程，但是是令一钟方式。

我们可以用

select name from mysql.proc where db=’数据库名’;

或者

select routine\_name from information\_schema.routines where routine\_schema='数据库名';

或者

show procedure status where db='[数据库](http://www.redisfans.com/)名';

进行查询。

如果我们想知道，某个存储过程的详细，那我们又该怎么做呢？是不是也可以像操作表一样用describe 表名进行查看呢？

答案是：我们可以查看存储过程的详细，但是需要用另一种方法：

SHOW CREATE PROCEDURE 数据库.存储过程名;

就可以查看当前存储过程的详细。

6.     MySQL存储过程的修改

ALTER PROCEDURE

更改用CREATE PROCEDURE 建立的预先指定的存储过程，其不会影响相关存储过程或存储功能。

7.     MySQL存储过程的删除

删除一个存储过程比较简单，和删除表一样：

DROP PROCEDURE

从MySQL的表格中删除一个或多个存储过程。

8.     MySQL存储过程的控制语句

(1). 变量作用域

内部的变量在其作用域范围内享有更高的优先权，当执行到end。变量时，内部变量消失，此时已经在其作用域外，变量不再可见了，应为在存储  
过程外再也不能找到这个申明的变量，但是你可以通过out参数或者将其值指派  
给会话变量来保存其值。

mysql > DELIMITER //

mysql > CREATE PROCEDURE proc3()

     -> begin

     -> declare x1 varchar(5) default 'outer';

     -> begin

     -> declare x1 varchar(5) default 'inner';

     -> select x1;

     -> end;

     -> select x1;

     -> end;

     -> //

mysql > DELIMITER ;

 (2). 条件语句

Ⅰ. if-then -else语句

mysql > DELIMITER //

mysql > CREATE PROCEDURE proc2(IN parameter int)

     -> begin

     -> declare var int;

     -> set var=parameter+1;

     -> if var=0 then

     -> insert into t values(17);

     -> end if;

     -> if parameter=0 then

     -> update t set s1=s1+1;

     -> else

     -> update t set s1=s1+2;

     -> end if;

     -> end;

     -> //

mysql > DELIMITER ;    
Ⅱ. case语句：

mysql > DELIMITER //

mysql > CREATE PROCEDURE proc3 (in parameter int)

     -> begin

     -> declare var int;

     -> set var=parameter+1;

     -> case var

     -> when 0 then

     -> insert into t values(17);

     -> when 1 then

     -> insert into t values(18);

     -> else

     -> insert into t values(19);

     -> end case;

     -> end;

     -> //

mysql > DELIMITER ;

(3). 循环语句

Ⅰ. while ···· end while：

mysql > DELIMITER //

mysql > CREATE PROCEDURE proc4()

     -> begin

     -> declare var int;

     -> set var=0;

     -> while var<6 do

     -> insert into t values(var);

     -> set var=var+1;

     -> end while;

     -> end;

     -> //

mysql > DELIMITER ;

Ⅱ. repeat···· end repeat：

它在执行操作后检查结果，而while则是执行前进行检查。

mysql > DELIMITER //

mysql > CREATE PROCEDURE proc5 ()

     -> begin

     -> declare v int;

     -> set v=0;

     -> repeat

     -> insert into t values(v);

     -> set v=v+1;

     -> until v>=5

     -> end repeat;

     -> end;

     -> //

mysql > DELIMITER ;    
Ⅲ. loop ·····end loop:

loop循环不需要初始条件，这点和while 循环相似，同时和repeat循环一样不需要结束条件, leave语句的意义是离开循环。

mysql > DELIMITER //

mysql > CREATE PROCEDURE proc6 ()

     -> begin

     -> declare v int;

     -> set v=0;

     -> LOOP\_LABLE:loop

     -> insert into t values(v);

     -> set v=v+1;

     -> if v >=5 then

     -> leave LOOP\_LABLE;

     -> end if;

     -> end loop;

     -> end;

     -> //

mysql > DELIMITER ;

Ⅳ. LABLES 标号：

标号可以用在begin repeat while 或者loop 语句前，语句标号只能在合法的语句前面使用。可以跳出循环，使运行指令达到复合语句的最后一步。

(4). ITERATE迭代

Ⅰ. ITERATE:

通过引用复合语句的标号,来从新开始复合语句

mysql > DELIMITER //

mysql > CREATE PROCEDURE proc10 ()

     -> begin

     -> declare v int;

     -> set v=0;

     -> LOOP\_LABLE:loop

     -> if v=3 then

     -> set v=v+1;

     -> ITERATE LOOP\_LABLE;

     -> end if;

     -> insert into t values(v);

     -> set v=v+1;

     -> if v>=5 then

     -> leave LOOP\_LABLE;

     -> end if;

     -> end loop;

     -> end;

     -> //

mysql > DELIMITER ;

9.     MySQL存储过程的基本函数

(1).字符串类

CHARSET(str) //返回字串字符集  
CONCAT (string2 [,... ]) //连接字串  
INSTR (string ,substring ) //返回substring首次在string中出现的位置,不存在返回0  
LCASE (string2 ) //转换成小写  
LEFT (string2 ,length ) //从string2中的左边起取length个字符  
LENGTH (string ) //string长度  
LOAD\_FILE (file\_name ) //从文件读取内容  
LOCATE (substring , string [,start\_position ] ) 同INSTR,但可指定开始位置  
LPAD (string2 ,length ,pad ) //重复用pad加在string开头,直到字串长度为length  
LTRIM (string2 ) //去除前端空格  
REPEAT (string2 ,count ) //重复count次  
REPLACE (str ,search\_str ,replace\_str ) //在str中用replace\_str替换search\_str  
RPAD (string2 ,length ,pad) //在str后用pad补充,直到长度为length  
RTRIM (string2 ) //去除后端空格  
STRCMP (string1 ,string2 ) //逐字符比较两字串大小,  
SUBSTRING (str , position [,length ]) //从str的position开始,取length个字符,  
注：mysql中处理字符串时，默认第一个字符下标为1，即参数position必须大于等于1

mysql> select substring('abcd',0,2);

+-----------------------+

| substring('abcd',0,2) |

+-----------------------+

|                       |

+-----------------------+

1 row in set (0.00 sec)

mysql> select substring('abcd',1,2);

+-----------------------+

| substring('abcd',1,2) |

+-----------------------+

|     ab                |

+-----------------------+

1 row in set (0.02 sec)

TRIM([[BOTH|LEADING|TRAILING] [padding] FROM]string2) //去除指定位置的指定字符  
UCASE (string2 ) //转换成大写  
RIGHT(string2,length) //取string2最后length个字符  
SPACE(count) //生成count个空格

(2).数学类

ABS (number2 ) //绝对值  
BIN (decimal\_number ) //十进制转二进制  
CEILING (number2 ) //向上取整  
CONV(number2,from\_base,to\_base) //进制转换  
FLOOR (number2 ) //向下取整  
FORMAT (number,decimal\_places ) //保留小数位数  
HEX (DecimalNumber ) //转十六进制  
注：HEX()中可传入字符串，则返回其ASC-11码，如HEX('DEF')返回4142143  
也可以传入十进制整数，返回其十六进制编码，如HEX(25)返回19  
LEAST (number , number2 [,..]) //求最小值  
MOD (numerator ,denominator ) //求余  
POWER (number ,power ) //求指数  
RAND([seed]) //随机数  
ROUND (number [,decimals ]) //四舍五入,decimals为小数位数]

注：返回类型并非均为整数，如：  
(1)默认变为整形值

mysql> select round(1.23);

+-------------+

| round(1.23) |

+-------------+

|           1 |

+-------------+

1 row in set (0.00 sec)

mysql> select round(1.56);

+-------------+

| round(1.56) |

+-------------+

|           2 |

+-------------+

1 row in set (0.00 sec)

(2)可以设定小数位数，返回浮点型数据

mysql> select round(1.567,2);

+----------------+

| round(1.567,2) |

+----------------+

|           1.57 |

+----------------+

1 row in set (0.00 sec)

SIGN (number2 ) //

(3).日期时间类

ADDTIME (date2 ,time\_interval ) //将time\_interval加到date2  
CONVERT\_TZ (datetime2 ,fromTZ ,toTZ ) //转换时区  
CURRENT\_DATE ( ) //当前日期  
CURRENT\_TIME ( ) //当前时间  
CURRENT\_TIMESTAMP ( ) //当前时间戳  
DATE (datetime ) //返回datetime的日期部分  
DATE\_ADD (date2 , INTERVAL d\_value d\_type ) //在date2中加上日期或时间  
DATE\_FORMAT (datetime ,FormatCodes ) //使用formatcodes格式显示datetime  
DATE\_SUB (date2 , INTERVAL d\_value d\_type ) //在date2上减去一个时间  
DATEDIFF (date1 ,date2 ) //两个日期差  
DAY (date ) //返回日期的天  
DAYNAME (date ) //英文星期  
DAYOFWEEK (date ) //星期(1-7) ,1为星期天  
DAYOFYEAR (date ) //一年中的第几天  
EXTRACT (interval\_name FROM date ) //从date中提取日期的指定部分  
MAKEDATE (year ,day ) //给出年及年中的第几天,生成日期串  
MAKETIME (hour ,minute ,second ) //生成时间串  
MONTHNAME (date ) //英文月份名  
NOW ( ) //当前时间  
SEC\_TO\_TIME (seconds ) //秒数转成时间  
STR\_TO\_DATE (string ,format ) //字串转成时间,以format格式显示  
TIMEDIFF (datetime1 ,datetime2 ) //两个时间差  
TIME\_TO\_SEC (time ) //时间转秒数]  
WEEK (date\_time [,start\_of\_week ]) //第几周  
YEAR (datetime ) //年份  
DAYOFMONTH(datetime) //月的第几天  
HOUR(datetime) //小时  
LAST\_DAY(date) //date的月的最后日期  
MICROSECOND(datetime) //微秒  
MONTH(datetime) //月  
MINUTE(datetime) //分返回符号,正负或0  
SQRT(number2) //开平方

本文主要记录了本人学习mysql存储过程时的笔记，文档中用到的表结构在文档结尾处有说明。

一、定义

       存储过程（Stored Procedure）是在大型数据库系统中，一组为了完成特定功能的SQL 语句集，存储在数据库中，经过第一次编译后调用不需要再次编译，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过程是数据库中的一个重要对象。

二、存储过程的特点

    1、能完成较复杂的判断和运算

    2、可编程行强，灵活

    3、SQL编程的代码可重复使用

    4、执行的速度相对快一些

    5、减少网络之间的数据传输，节省开销；

**详述MySQL 存储过程**

MySQL 5.0 版本开始支持存储过程。

存储过程（Stored Procedure）是一种在数据库中存储复杂程序，以便外部程序调用的一种数据库对象。

存储过程是为了完成特定功能的SQL语句集，经编译创建并保存在数据库中，用户可通过指定存储过程的名字并给定参数(需要时)来调用执行。

存储过程思想上很简单，就是数据库 SQL 语言层面的代码封装与重用。

**优点**

存储过程可封装，并隐藏复杂的商业逻辑。

存储过程可以回传值，并可以接受参数。

存储过程无法使用 SELECT 指令来运行，因为它是子程序，与查看表，数据表或用户定义函数不同。

存储过程可以用在数据检验，强制实行商业逻辑等。

**缺点**

存储过程，往往定制化于特定的数据库上，因为支持的编程语言不同。当切换到其他厂商的数据库系统时，需要重写原有的存储过程。

存储过程的性能调校与撰写，受限于各种数据库系统。

**一、存储过程的创建和调用**

存储过程就是具有名字的一段代码，用来完成一个特定的功能。

创建的存储过程保存在数据库的数据字典中。

**创建存储过程**

CREATE [DEFINER = { user | CURRENT\_USER }] 　PROCEDURE sp\_name ([proc\_parameter[,...]]) [characteristic ...] routine\_body proc\_parameter: [ IN | OUT | INOUT ] param\_name type characteristic: COMMENT 'string' | LANGUAGE SQL | [NOT] DETERMINISTIC | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } | SQL SECURITY { DEFINER | INVOKER } routine\_body: 　　Valid SQL routine statement [begin\_label:] BEGIN 　　[statement\_list] 　　　　…… END [end\_label]

**MYSQL 存储过程中的关键语法**

声明语句结束符，可以自定义:

DELIMITER $$

或

DELIMITER //

声明存储过程:

CREATE PROCEDURE demo\_in\_parameter(IN p\_in int)

存储过程开始和结束符号:

BEGIN .... END

变量赋值:

SET @p\_in=1

变量定义:

DECLARE l\_int int unsigned default 4000000;

创建mysql存储过程、存储函数:

create procedure 存储过程名(参数)

存储过程体:

create function 存储函数名(参数)

**实例**

创建数据库，备份数据表用于示例操作：

mysql> create database db1; mysql> use db1; mysql> create table PLAYERS as select \* from TENNIS.PLAYERS; mysql> create table MATCHES as select \* from TENNIS.MATCHES;

下面是存储过程的例子，删除给定球员参加的所有比赛：

mysql> delimiter $$　　#将语句的结束符号从分号;临时改为两个$$(可以是自定义) mysql> CREATE PROCEDURE delete\_matches(IN p\_playerno INTEGER) -> BEGIN -> 　　DELETE FROM MATCHES -> WHERE playerno = p\_playerno; -> END$$ Query OK, 0 rows affected (0.01 sec) mysql> delimiter;　　#将语句的结束符号恢复为分号

**解析：**默认情况下，存储过程和默认数据库相关联，如果想指定存储过程创建在某个特定的数据库下，那么在过程名前面加数据库名做前缀。 在定义过程时，使用 **DELIMITER $$** 命令将语句的结束符号从分号 **;** 临时改为两个 **$$**，使得过程体中使用的分号被直接传递到服务器，而不会被客户端（如mysql）解释。

调用存储过程：

call sp\_name[(传参)];

mysql> select \* from MATCHES; +---------+--------+----------+-----+------+ | MATCHNO | TEAMNO | PLAYERNO | WON | LOST | +---------+--------+----------+-----+------+ | 1 | 1 | 6 | 3 | 1 | | 7 | 1 | 57 | 3 | 0 | | 8 | 1 | 8 | 0 | 3 | | 9 | 2 | 27 | 3 | 2 | | 11 | 2 | 112 | 2 | 3 | +---------+--------+----------+-----+------+ 5 rows in set (0.00 sec) mysql> call delete\_matches(57); Query OK, 1 row affected (0.03 sec) mysql> select \* from MATCHES; +---------+--------+----------+-----+------+ | MATCHNO | TEAMNO | PLAYERNO | WON | LOST | +---------+--------+----------+-----+------+ | 1 | 1 | 6 | 3 | 1 | | 8 | 1 | 8 | 0 | 3 | | 9 | 2 | 27 | 3 | 2 | | 11 | 2 | 112 | 2 | 3 | +---------+--------+----------+-----+------+ 4 rows in set (0.00 sec)

**解析：**在存储过程中设置了需要传参的变量p\_playerno，调用存储过程的时候，通过传参将57赋值给p\_playerno，然后进行存储过程里的SQL操作。

**存储过程体**

* 存储过程体包含了在过程调用时必须执行的语句，例如：dml、ddl语句，if-then-else和while-do语句、声明变量的declare语句等
* 过程体格式：以begin开始，以end结束(可嵌套)

BEGIN

　　BEGIN

　　　　BEGIN

　　　　　　statements;

　　　　END

　　END

END

**注意：**每个嵌套块及其中的每条语句，必须以分号结束，表示过程体结束的begin-end块(又叫做复合语句compound statement)，则不需要分号。

为语句块贴标签:

[begin\_label:] BEGIN

　　[statement\_list]

END [end\_label]

例如：

label1: BEGIN 　　label2: BEGIN 　　　　label3: BEGIN 　　　　　　statements; 　　　　END label3 ; 　　END label2; END label1

标签有两个作用：

1、增强代码的可读性

2、在某些语句(例如:leave和iterate语句)，需要用到标签

**二、存储过程的参数**

MySQL存储过程的参数用在存储过程的定义，共有三种参数类型,IN,OUT,INOUT,形式如：

CREATEPROCEDURE 存储过程名([[IN |OUT |INOUT ] 参数名 数据类形...])

IN 输入参数：表示调用者向过程传入值（传入值可以是字面量或变量）

OUT 输出参数：表示过程向调用者传出值(可以返回多个值)（传出值只能是变量）

INOUT 输入输出参数：既表示调用者向过程传入值，又表示过程向调用者传出值（值只能是变量）

**1、in 输入参数**

mysql> delimiter $$ mysql> create procedure in\_param(in p\_in int) -> begin -> 　　select p\_in; -> 　　set p\_in=2; -> select P\_in; -> end$$ mysql> delimiter ; mysql> set @p\_in=1; mysql> call in\_param(@p\_in); +------+ | p\_in | +------+ | 1 | +------+ +------+ | P\_in | +------+ | 2 | +------+ mysql> select @p\_in; +-------+ | @p\_in | +-------+ | 1 | +-------+

以上可以看出，p\_in 在存储过程中被修改，但并不影响 @p\_id 的值，因为前者为局部变量、后者为全局变量。

**2、out输出参数**

mysql> delimiter // mysql> create procedure out\_param(out p\_out int) -> begin -> select p\_out; -> set p\_out=2; -> select p\_out; -> end -> // mysql> delimiter ; mysql> set @p\_out=1; mysql> call out\_param(@p\_out); +-------+ | p\_out | +-------+ | NULL | +-------+ 　　#因为out是向调用者输出参数，不接收输入的参数，所以存储过程里的p\_out为null +-------+ | p\_out | +-------+ | 2 | +-------+ mysql> select @p\_out; +--------+ | @p\_out | +--------+ | 2 | +--------+ 　　#调用了out\_param存储过程，输出参数，改变了p\_out变量的值

**3、inout输入参数**

mysql> delimiter $$ mysql> create procedure inout\_param(inout p\_inout int) -> begin -> select p\_inout; -> set p\_inout=2; -> select p\_inout; -> end -> $$ mysql> delimiter ; mysql> set @p\_inout=1; mysql> call inout\_param(@p\_inout); +---------+ | p\_inout | +---------+ | 1 | +---------+ +---------+ | p\_inout | +---------+ | 2 | +---------+ mysql> select @p\_inout; +----------+ | @p\_inout | +----------+ | 2 | +----------+ #调用了inout\_param存储过程，接受了输入的参数，也输出参数，改变了变量

**注意：**

1、如果过程没有参数，也必须在过程名后面写上小括号例：

CREATE PROCEDURE sp\_name ([proc\_parameter[,...]]) ……

2、确保参数的名字不等于列的名字，否则在过程体中，参数名被当做列名来处理

**建议：**

输入值使用in参数。

返回值使用out参数。

inout参数就尽量的少用。

**三、变量**

**1. 变量定义**

局部变量声明一定要放在存储过程体的开始：

DECLAREvariable\_name [,variable\_name...] datatype [DEFAULT value];

其中，datatype 为 MySQL 的数据类型，如: int, float, date,varchar(length)

例如:

DECLARE l\_int int unsigned default 4000000; DECLARE l\_numeric number(8,2) DEFAULT 9.95; DECLARE l\_date date DEFAULT '1999-12-31'; DECLARE l\_datetime datetime DEFAULT '1999-12-31 23:59:59'; DECLARE l\_varchar varchar(255) DEFAULT 'This will not be padded';

**2. 变量赋值**

SET 变量名 = 表达式值 [,variable\_name = expression ...]

**3. 用户变量**

**在MySQL客户端使用用户变量:**

mysql > SELECT 'Hello World' into @x; mysql > SELECT @x; +-------------+ | @x | +-------------+ | Hello World | +-------------+ mysql > SET @y='Goodbye Cruel World'; mysql > SELECT @y; +---------------------+ | @y | +---------------------+ | Goodbye Cruel World | +---------------------+ mysql > SET @z=1+2+3; mysql > SELECT @z; +------+ | @z | +------+ | 6 | +------+

**在存储过程中使用用户变量：**

mysql > CREATE PROCEDURE GreetWorld( ) SELECT CONCAT(@greeting,' World'); mysql > SET @greeting='Hello'; mysql > CALL GreetWorld( ); +----------------------------+ | CONCAT(@greeting,' World') | +----------------------------+ | Hello World | +----------------------------+

**在存储过程间传递全局范围的用户变量：**

mysql> CREATE PROCEDURE p1() SET @last\_procedure='p1'; mysql> CREATE PROCEDURE p2() SELECT CONCAT('Last procedure was ',@last\_procedure); mysql> CALL p1( ); mysql> CALL p2( ); +-----------------------------------------------+ | CONCAT('Last procedure was ',@last\_proc | +-----------------------------------------------+ | Last procedure was p1 | +-----------------------------------------------+

**注意:**

1、用户变量名一般以@开头

2、滥用用户变量会导致程序难以理解及管理

**四、注释**

MySQL 存储过程可使用两种风格的注释

两个横杆--：该风格一般用于单行注释。

**c 风格**： 一般用于多行注释。

例如：

mysql > DELIMITER // mysql > CREATE PROCEDURE proc1 --name存储过程名 -> (IN parameter1 INTEGER) -> BEGIN -> DECLARE variable1 CHAR(10); -> IF parameter1 = 17 THEN -> SET variable1 = 'birds'; -> ELSE -> SET variable1 = 'beasts'; -> END IF; -> INSERT INTO table1 VALUES (variable1); -> END -> // mysql > DELIMITER ;

**MySQL存储过程的调用**

用call和你过程名以及一个括号，括号里面根据需要，加入参数，参数包括输入参数、输出参数、输入输出参数。具体的调用方法可以参看上面的例子。

**MySQL存储过程的查询**

我们像知道一个数据库下面有那些表，我们一般采用 **showtables;** 进行查看。那么我们要查看某个数据库下面的存储过程，是否也可以采用呢？答案是，我们可以查看某个数据库下面的存储过程，但是是另一钟方式。

我们可以用以下语句进行查询：

selectname from mysql.proc where db='数据库名';

或者

selectroutine\_name from information\_schema.routines where routine\_schema='数据库名';

或者

showprocedure status where db='数据库名';

**如果我们想知道，某个存储过程的详细，那我们又该怎么做呢？是不是也可以像操作表一样用describe 表名进行查看呢？**

**答案是：**我们可以查看存储过程的详细，但是需要用另一种方法：

SHOWCREATE PROCEDURE 数据库.存储过程名;

就可以查看当前存储过程的详细。

**MySQL存储过程的修改**

ALTER PROCEDURE

更改用 CREATE PROCEDURE 建立的预先指定的存储过程，其不会影响相关存储过程或存储功能。

**MySQL存储过程的删除**

删除一个存储过程比较简单，和删除表一样：

DROPPROCEDURE

从 MySQL 的表格中删除一个或多个存储过程。

**MySQL存储过程的控制语句**

**(1). 变量作用域**

内部的变量在其作用域范围内享有更高的优先权，当执行到 end。变量时，内部变量消失，此时已经在其作用域外，变量不再可见了，应为在存储过程外再也不能找到这个申明的变量，但是你可以通过 out 参数或者将其值指派给会话变量来保存其值。

mysql > DELIMITER // mysql > CREATE PROCEDURE proc3() -> begin -> declare x1 varchar(5) default 'outer'; -> begin -> declare x1 varchar(5) default 'inner'; -> select x1; -> end; -> select x1; -> end; -> // mysql > DELIMITER ;

**(2). 条件语句**

1. if-then-else 语句

mysql > DELIMITER // mysql > CREATE PROCEDURE proc2(IN parameter int) -> begin -> declare var int; -> set var=parameter+1; -> if var=0 then -> insert into t values(17); -> end if; -> if parameter=0 then -> update t set s1=s1+1; -> else -> update t set s1=s1+2; -> end if; -> end; -> // mysql > DELIMITER ;

2. case语句：

mysql > DELIMITER // mysql > CREATE PROCEDURE proc3 (in parameter int) -> begin -> declare var int; -> set var=parameter+1; -> case var -> when 0 then -> insert into t values(17); -> when 1 then -> insert into t values(18); -> else -> insert into t values(19); -> end case; -> end; -> // mysql > DELIMITER ; case when var=0 then insert into t values(30); when var>0 then when var<0 then else end case

**(3). 循环语句**

1. while ···· end while

mysql > DELIMITER // mysql > CREATE PROCEDURE proc4() -> begin -> declare var int; -> set var=0; -> while var<6 do -> insert into t values(var); -> set var=var+1; -> end while; -> end; -> // mysql > DELIMITER ;

while 条件 do

--循环体

endwhile

2. repeat···· end repea

它在执行操作后检查结果，而 while 则是执行前进行检查。

mysql > DELIMITER // mysql > CREATE PROCEDURE proc5 () -> begin -> declare v int; -> set v=0; -> repeat -> insert into t values(v); -> set v=v+1; -> until v>=5 -> end repeat; -> end; -> // mysql > DELIMITER ;

repeat

--循环体

until 循环条件

end repeat;

3. loop ·····endloop

loop 循环不需要初始条件，这点和 while 循环相似，同时和 repeat 循环一样不需要结束条件, leave 语句的意义是离开循环。

mysql > DELIMITER // mysql > CREATE PROCEDURE proc6 () -> begin -> declare v int; -> set v=0; -> LOOP\_LABLE:loop -> insert into t values(v); -> set v=v+1; -> if v >=5 then -> leave LOOP\_LABLE; -> end if; -> end loop; -> end; -> // mysql > DELIMITER ;

4. LABLES 标号：

标号可以用在 begin repeat while 或者 loop 语句前，语句标号只能在合法的语句前面使用。可以跳出循环，使运行指令达到复合语句的最后一步。

**(4). ITERATE迭代**

ITERATE 通过引用复合语句的标号,来从新开始复合语句:

mysql > DELIMITER // mysql > CREATE PROCEDURE proc10 () -> begin -> declare v int; -> set v=0; -> LOOP\_LABLE:loop -> if v=3 then -> set v=v+1; -> ITERATE LOOP\_LABLE; -> end if; -> insert into t values(v); -> set v=v+1; -> if v>=5 then -> leave LOOP\_LABLE; -> end if; -> end loop; -> end; -> // mysql > DELIMITER ;

*参考文章：*

*https://www.cnblogs.com/geaozhang/p/6797357.html*

*http://blog.sina.com.cn/s/blog\_86fe5b440100wdyt.html*

**031、深入理解数据库事务？**

数据库事务（简称‘事务’）  
数据库事务是指作为单个逻辑工作单元执行的一系列数据库操作。

Transaction作为关系型数据库的核心组成，在数据安全方面有着非常重要的作用，本文会一步步解析事务的核心特性，以获得对事务更深的理解。

什么是事务

数据库几乎是所有系统的核心模块，它将数据有条理地保存在储存介质(磁盘）中，  
并在逻辑上，将数据以结构化的形态呈现给用户。支持数据的增、删、改、查，并在过程中保障数据的正确且可靠。

要做到这点并非易事，常见的例子就是银行转账，A账户给B账户转账一个亿(T1)，买一块地盖房子。在这种交易的过程中，有几个问题值得思考：

如何同时保证上述交易中，A账户总金额减少一个亿，B账户总金额增加一个亿？ A

A账户如果同时在和C账户交易(T2)，如何让这两笔交易互不影响？ I

如果交易完成时数据库突然崩溃，如何保证交易数据成功保存在数据库中？ D

如何在支持大量交易的同时，保证数据的合法性(没有钱凭空产生或消失) ？ C

要保证交易正常可靠地进行，数据库就得解决上面的四个问题，这也就是事务诞生的背景，它能解决上面的四个问题，对应地，它拥有四大特性：

原子性（Atomicity）: 事务要么全部完成，要么全部取消。 如果事务崩溃，状态回到事务之前（事务回滚）。

隔离性（Isolation）: 如果2个事务 T1 和 T2 同时运行，事务 T1 和 T2 最终的结果是相同的，不管 T1和T2谁先结束。

持久性（Durability）: 一旦事务提交，不管发生什么（崩溃或者出错），数据要保存在数据库中。

一致性（Consistency）: 只有合法的数据（依照关系约束和函数约束）才能写入数据库。

ACID

接下来详细地了解这四大特性：

原子性，确保不管交易过程中发生了什么意外状况（服务器崩溃、网络中断等），不能出现A账户少了一个亿，但B账户没到帐，或者A账户没变，但B账户却凭空收到一个亿（数据不一致）。A和B账户的金额变动要么同时成功，要么同时失败(保持原状)。

隔离性，如果A在转账1亿给B（T1），同时C又在转账3亿给A（T2），不管T1和T2谁先执行完毕，最终结果必须是A账户增加2亿，而不是3亿，B增加1亿，C减少3亿。

持久性，确保如果 T1 刚刚提交，数据库就发生崩溃，T1执行的结果依然会保持在数据库中。

一致性，确保钱不会在系统内凭空产生或消失， 依赖原子性和隔离性。

可以看出，原子性、隔离性、一致性的根本问题，是不同的事务同时对同一份数据(A账户)进行写操作(修改、删除、新增)，如果事务中都只是读数据的话，那么它们可以随意地同时进行，反正读到的数据都是一样的。

如果，几个互不知晓的事务在同时修改同一份数据，那么很容易出现后完成的事务覆盖了前面的事务的结果，导致不一致。 事务在最终提交之前都有可能会回滚，撤销所有修改：

如果T1事务修改了A账户的数据，

这时T2事务读到了更新后的A账户数据，并进行下一步操作，

但此时T1事务却回滚了，撤销了对A账户的修改，

那么T2读取到的A账户数据就是非法的，这会导致数据不一致。

这些问题都是事务需要避免的。

所以，在事务中更新某条数据获得的互斥锁，只有在事务提交或失败之后才会释放，在此之前，其他事务是只能读，不能写这条数据的。

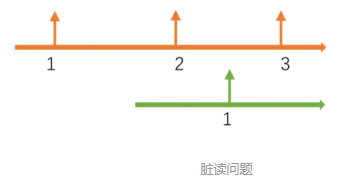
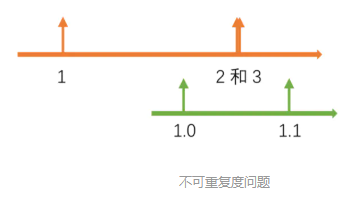
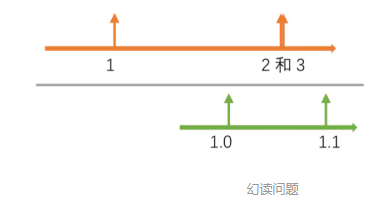
这就是隔离性的关键，针对隔离性的强度，有以下四的级别([引用自这篇文章](https://link.zhihu.com/?target=http%3A//blog.jobbole.com/100349/))：

读取未提交（Read uncommitted）：最低级别的隔离，是读取已提交+新的隔离突破。如果事务A读取了数据D，然后数据D被事务B修改（但并未提交，事务B仍在运行中），事务A再次读取数据D时，数据修改是可见的。如果事务B回滚，那么事务A第二次读取的数据D是无意义的，因为那是事务B所做的从未发生的修改（已经回滚了嘛）。这叫脏读（dirty read）。Read uncommit：（读未提交）有脏读问题

读取已提交（Read committed，Oracle、PostgreSQL、SQL Server默认模式）：可重复读+新的隔离突破。如果事务A读取了数据D，然后数据D被事务B修改（或删除）并提交，事务A再次读取数据D时数据的变化（或删除）是可见的。这叫不可重复读（non-repeatable read）。Read commit：（读已提交）解决脏读问题，有不可重复读问题

可重复读（Repeatable read，MySQL默认模式）：如果一个事务成功执行并且添加了新数据(事务提交)，这些数据对其他正在执行的事务是可见的。但是如果事务成功修改了一条数据，修改结果对正在运行的事务不可见。所以，事务之间只是在新数据方面突破了隔离，对已存在的数据仍旧隔离。Repeatable read：（可重复读）解决不可重复读问题，有幻读问题

串行化(Serializable，SQLite默认模式）：最高级别的隔离。两个同时发生的事务100%隔离，每个事务有自己的"世界", 串行执行。串行化）解决幻读问题，所有问题都解决

* 由于事务可以分为三个阶段：读取、修改和提交，那么下面根据事务的三阶段，通过进行分析对应问题（脏读等问题）和采用对应的策略（行级锁等策略）来学习。如下图所示，其中我们后面就将读取阶段简称为1，修改阶段简称为2，提交节点简称为3。横轴表示时间线
* 
* 事务的三个阶段
* 下面我们用两个事务的时间线来讲解没有事务的一些隔离级别的时候存在的一些问题。  
  事务A：橘色  
  事务B：绿色
* 脏读
* 问题：
* 如果事务B的读取阶段位于事务A的修改和提交阶段（其实就是读取到了其他事务修改但是没有提交的数据），紧接着这个时候事务A发生回滚，那么事务B读取到的数据就是脏数据——脏读。
* 
* 脏读问题
* **解决：**
* 在事务修改和提交（也就是2和3阶段）之间增加写锁（就是排它锁，在读取期间不允许其他事务读取事务），这样就解决了脏读问题——其实就是Read commit（提交读）。
* **不可重复度**
* **问题：**
* 脏读问题解决了，但是在脏读问题之上又存在新的问题，比如在事务B第一次读取和第二次读取期间，事务A发生了修改和提交（这里我们站在解决脏读的问题上来看，事务A的2和3加了排它锁，我这里将两者合在一起），导致事务B第一次和第二次读取到的数据不一样（数据本身的值发生了变化）——不可重复读。
* 
* 不可重复度问题
* 解决：
* 1.通过添加行级共享锁
* 通过在事务B的读取到提交（就是阶段1~3，上图我们就只显示了读取阶段）期间加上读锁，在读锁这个共享锁加锁期间，事务A是不能添加排它锁的，也就是不能进行修改，进而每次事务B读取到的都是最新提交的数据。但是这种仍然存在幻读（下面介绍）。
* 2.通过用mvcc策略
* mvcc 可参考相关文档
* 幻读
* 问题：
* 不可重复读问题解决了，但是这里还有幻读问题，就是事务第一次读取和第二次读取，发现读取到的数据量不一样——幻读（这个跟不可重复读相似，但是这个强调的是数量的变化）。  
  下图我用灰色线表示两个事务处理的是不同的数据行（因为可重复读的隔离级别要求了一个事务读取到结束都加上了行级共享锁，其他事务是无法在中间修改）。
* 
* 幻读问题
* 当其中事务A在某个时间修改了其他行的数据，而此修改之前事务B读取了一次，但是这个数据修改后却满足了事务B的查询要求，就会引起事务B在第一次和第二次查询期间数据量不一样，就产生了幻读的问题。
* 解决：
* 在一个事务的执行中，添加表级锁，读添加表级共享锁，写添加表级排它锁，这样事务之间的执行就会完全无交叉。但是这种表级锁对应引起的性能会比较低，因为所有的事务就是串行化的执行（这也许就是叫串行化的原因）。下面我们会讲解下，一个新的解决方法：mvcc。就是在解决幻读的情况下，能让隔离级别保持在不可重复度的阶段。

**032、如何保证MySQL数据库持久性？**

隔离性的问题解决了，但是如果在事务提交后，事务的数据还没有真正落到磁盘上，此时数据库奔溃了，事务对应的数据会不会丢？

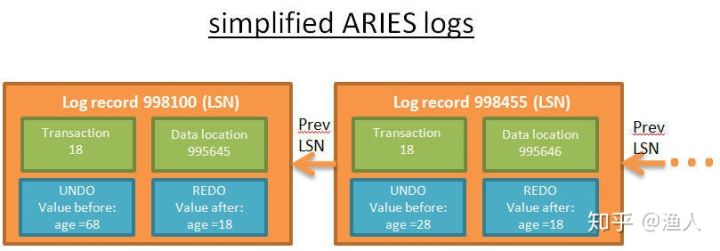
事务会保证数据不会丢，当数据库因不可抗拒的原因奔溃后重启，它会保证：

* 成功提交的事务，数据会保存到磁盘
* 未提交的事务，相应的数据会回滚

事务日志

数据库通过事务日志来达到这个目标。 事务的每一个操作（增/删/改）产生一条日志，内容组成大概如下：

* LSN：一个按时间顺序分配的唯一日志序列号，靠后的操作的LSN比靠前的大。
* TransID：产生操作的事务ID。
* PageID：被修改的数据在磁盘上的位置，数据以页为单位存储。
* PrevLSN：同一个事务产生的上一条日志记录的指针。
* UNDO：取消本次操作的方法，按照此方法回滚。
* REDO：重复本次操作的方法，如有必要，重复此方法保证操作成功。



磁盘上每个页（保存数据的，不是保存日志的）都记录着最后一个修改该数据操作的LSN。数据库会通过解析事务日志，将修改真正落到磁盘上(写盘)，随后清理事务日志(正常情况下)。

这也是数据库在保证数据安全和性能这两个点之前的折中办法：

* 如果每次更新都写盘，由于数据是随机的，会造成大量的随机IO，性能会非常差
* 如果每次更新不马上写盘，那一旦数据库崩溃，数据就会丢失

折中的办法就是：

* 将数据的变更以事务日志的方式，按照时间先后追加到日志缓冲区，由特定算法写入事务日志，这是顺序IO，性能较好
* 通过数据管理器解析事务日志，由特定的算法择机进行写盘

数据库恢复

当数据库从崩溃中恢复时，会有以下几个步骤：

* 解析存在的事务日志，分析哪些事务需要回滚，哪些需要写盘(还没来得及写盘，数据库就崩溃了)。
* Redo，进行写盘。检测对应数据所在数据页的LSN，如果数据页的LSN>=事务操作的LSN，说明已经写过盘，不然进行写盘操作。
* Undo, 按照LSN倒序进行回滚

经过这几个阶段，在数据库恢复后，可以达到奔溃前的状态，也保证了数据的一致性。

#传播行为

1、PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

2、PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

3、PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

4、PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

5、PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

6、PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

7、PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

**032、mysql 存储过程 教程？**

mysql是开源的关系型数据库，由于其体积小、开源、有条件免费，所以，使用mysql的用户还是很多的。

本文就从创建表、插入测试数据、创建存储过程、执行存储过程、修改存储过程、删除存储过程等方面，完整的演示mysql存储过程的使用

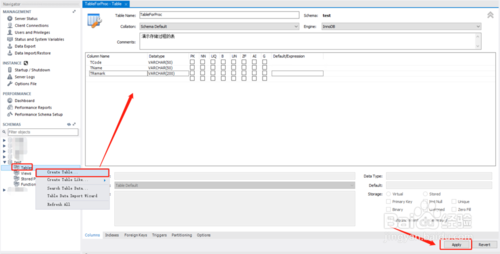
方法/步骤

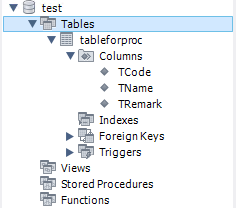
1. 创建表

     1）打开mysql的图形化管理工具【MySQL Workbench 6.3 CE】，登录mysql数据库

     2）在【test】测试数据库（自己创建的空数据库，实际场景使用具体数据库即可）的【Tables】菜单上，点击右键选择【Create Table】

     3）在新建表界面，输入表名称、备注、添加列，点击【Apply】按钮完成创建表

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=1)

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=2)

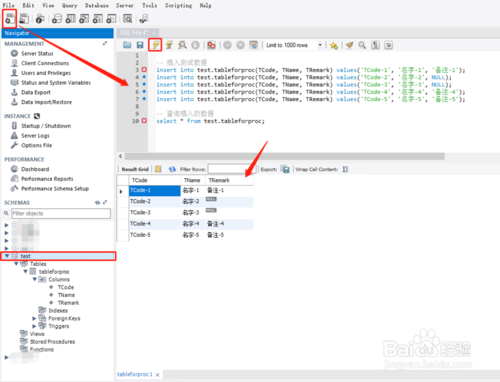
1. 插入测试数据

     1）选中test数据库，此步可选（因为mysql的sql语句都是要带数据库名称前缀的）

     2）点击菜单顶部的第一个图标菜单【Create a new tab ...】

     3）在sql语句窗口，输入插入语句与查询语句

     4）点击sql窗口上面的 闪电 图标，执行sql语句

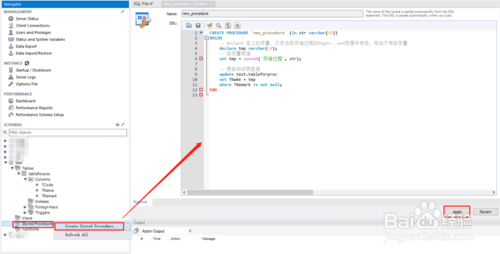
[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=3)

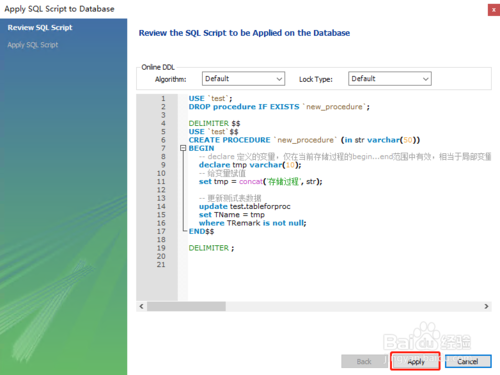
1. 创建存储过程

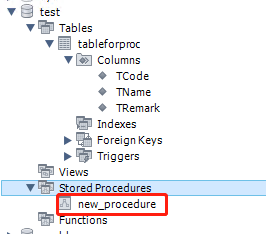
    1）在test数据库的【Stored Procedures】菜单中点击右键【Create Stored Procedures】

    2）在创建存储过程界面，输入如图所示的代码（虽然有一个输入参数str，但是没有使用，在存储过程中可以直接使用这个变量）

    3）点击【Apply】创建存储过程，在弹窗中都点击【Apply】即可

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=4)

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=5)

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=6)

1. 执行存储过程

     1）在SQL窗口中调用存储过格式【call new\_procedure('haha');】

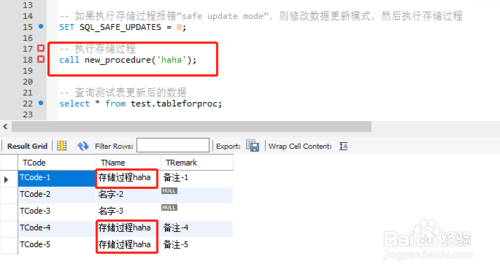
          A）call是调用存储过程的必须命令

          B）new\_procedure是要调用的存储过程名字

          C）('haha')是传入存储过程的参数

     2）如果执行存储过程报错“safe update mode”，则修改数据更新模式，然后执行存储过程

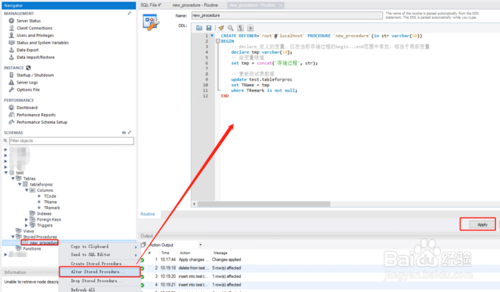
          SET SQL\_SAFE\_UPDATES = 0;

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=7)

1. **修改存储过程**

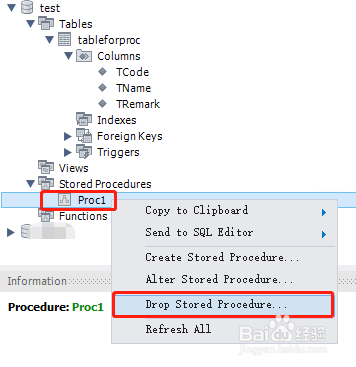
     1）在要修改的存储过程名字上点击右键，选择【Alter Stored Procedure...】

     2）在修改存储过程界面，直接修改后，Apply即可

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=8)

1. **删除存储过程**

     在要删除的存储过程名字上点击右键，选择【Drop Stored Procedure...】即可

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=9)

END

经验内容仅供参考，如果您需解决具体问题(尤其法律、医学等领域)，建议您详细咨询相关领域专业人士。

[举报](javascript:void(0))作者声明：本篇经验系本人依照真实经历原创，未经许可，谢绝转载。

**033、mysql复制--主从复制配置？**

当单台 MYSQL 服务器无法满足当前网站流量时的优化方案。需要搭建 mysql 集群技术。

复制功能：

数据分布

负载均衡（读）

备份

高可用和故障切换

MySQL升级测试

复制方式：

主–从复制

主–主复制

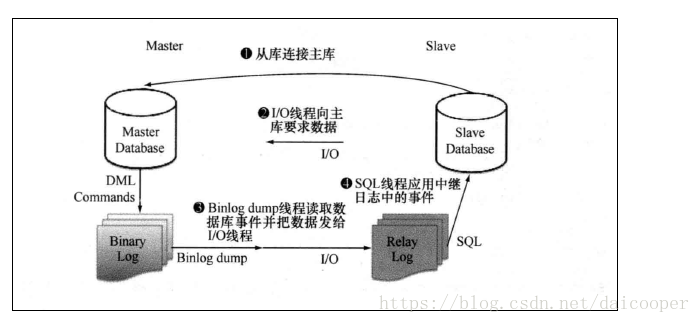
半同步复制

复制原理：

Mysql 中有一种日志叫做 bin 日志（二进制日志）。这个日志会记录下所有修改了数据库的SQL 语句（insert,update,delete,create/alter/drop table, grant 等等）。

主从复制的原理其实就是把主服务器上的 bin 日志复制到从服务器上执行一遍，这样从服务器上的数据就和主服务器上的数据相同了。

复制流程图：



复制过程：

主节点必须启用二进制日志，记录任何修改了数据库数据的事件。

从节点开启一个线程（I/O Thread)把自己扮演成 mysql 的客户端，通过 mysql 协议，请求主节点的二进制日志文件中的事件

主节点启动一个线程（dump Thread），检查自己二进制日志中的事件，跟对方请求的位置对比，如果不带请求位置参数，则主节点就会从第一个日志文件中的第一个事件一个一个发送给从节点。

从节点接收到主节点发送过来的数据把它放置到中继日志（Relay log）文件中。并记录该次请求到主节点的具体哪一个二进制日志文件内部的哪一个位置（主节点中的二进制文件会有多个，在后面详细讲解）。

从节点启动另外一个线程（sql Thread ），把 Relay log 中的事件读取出来，并在本地再执行一次。

复制中线程的作用：

从节点：

I/O Thread: 从 Master 节点请求二进制日志事件，并保存于中继日志中。

Sql Thread: 从Relay log 中读取日志事件并在本地完成重放。

主节点：

Dump Thread:为每个 Slave 的 I/O Thread 启动一个 dump 线程，用于向从节点发送二进制事件。

\*\*思考：\*\*从节点需要建立二进制日志文件吗？

看情况，如果从节点需要作为其他节点的主节点时，是需要开启二进制日志文件的。这种情况叫做级联复制。如果只是作为从节点，则不需要创建二进制文件。

Mysql复制特点：

异步复制：主节点中一个用户请求一个写操作时，主节点不需要把写的数据在本地操作完成同时发送给从服务器并等待从服务器反馈写入完成，再响应用户。主节点只需要把写入操作在本地完成，就响应用户。但是，从节点中的数据有可能会落后主节点，可以使用（很多软件来检查是否落后）主从数据不一致。

主从复制配置过程：

主节点：

启用二进制日志。

为当前节点设置一个全局唯一的server\_id。

创建有复制权限的用户账号 REPLIACTION SLAVE ,REPLIATION CLIENT。

从节点：

启动中继日志。

为当前节点设置一个全局唯一的server\_id。

使用有复制权限的用户账号连接至主节点，并启动复制线程。

配置演示：

测试环境：

主节点：192.168.0.196

从节点：192.168.0.198

mysql版本：mysql-5.5.17

linux：centos6.5

1：编辑主节点配置文件

2：启用二进制日志文件

添加：log-bin = mysql-bin

添加 :server-id =1

添加：innodb-file-per-table =ON

添加：skip\_name\_resolve=ON

3：启动mysql服务，并连接mysql

4：查看日志信息

1:)查看二进制日志是否开启

2:)查看主节点二进制日志列表

3:)查看主节点的server id

5：在主节点上创建有复制权限的用户。REPLIACTION SLAVE ，REOPLIATION CLIENT

6：刷新

主服务器配置完成。

注意：确保主服务器的iptables没有阻断3306的访问端口。

7：编辑从节点配置文件

8：开启中继日志

添加：relay-log=relay-log

添加：relay-log-index=relay-log.index

添加：server-id=2

添加：innodb\_file\_per\_table=ON

添加：skip\_name\_resolve=ON

9:启动从节点mysql服务，并登陆mysql

10:查看从节点日志信息

1:)查看中继日志是否开启

2:)产看server 相关信息

11：在从节点配置访问主节点的参数信息

添加 主节点主机，访问主节点的用户名及密码，主节点二进制文件信息。

注意：主节点的二进制文件一定要是二进制列表中的最后一个二进制文件。

CHANGE MASTER TO MASTER\_HOST='192.168.0.196',MASTER\_USER='repluser',MASTER\_PASSWORD='replpass', MASTER\_LOG\_FILE='mysql-bin.000042',MASTER\_LOG\_POS=362;

12：查看从节点的状态信息

因为没有启动 从节点的复制线程，IO线程 和 SQL 线程都为NO.

13：启动复制线程

START SLAVE 可以指定线程类型：IO\_THREAD ,SQL\_THREAD, 如果不指定，则两个都启动。

14:再次查看从节点状态信息

至此，mysql的主----从复制配置完成。

15：测试

1：）在主节点创建数据库，并查看主节点二进制日志信息

2：）在从节点查找二进制日志信息，并查看mydb数据库是否复制成功

主从复制架构中应注意的问题：

从节点要设置某些限定使得它不能进行写操作,才能保证复制当中的数据一致。

1：限制从服务器为只读

在从服务器上设置：

read\_only = ON,但是此限制对拥有SUPER权限 的用户均无效。

阻止所有用户：

mysq>FLUSH TABLES WITH READ LOCK;

2:如何保证主从复制时的事物安全？

1：在主节点设置参数

sync\_binlog=1： Mysql开启bin-log日志使用bin-log时，默认情况下，并不是每次执行写入就与硬盘同步，这样在服务器崩溃时，就可能导致bin-log最后的语句丢失。可以通过这个参数来调节，sync\_binlog=N，使执行N次写入后，与硬盘同步。1是最安全的，但是也是最慢的。

如果用到innode 存储引擎:

innodb\_flush\_logs\_at\_trx\_commit=ON（刷写日志：在事务提交时，要将内存中跟事务相关的数据立即刷写到事务日志中去。）

innodb\_support\_xa=ON (分布式事务：基于它来做两段式提交功能)

sync\_master\_info=1：每次给从节点dump一些事件信息之后，主节点的master info 信息会立即同步到磁盘上。让从服务器中的 master\_info 及时更新。

2：在每个slave节点

skip\_slave\_start =ON (跳过自动启动，使用手动启动。)

relay\_log也会在内从中先缓存，然后在同步到relay\_log中去，可以使用下面参数使其立即同步。

sync\_relay\_log =1 ，默认为10000，即每10000次sync\_relay\_log事件会刷新到磁盘。为0则表示不刷新，交由OS的cache控制。

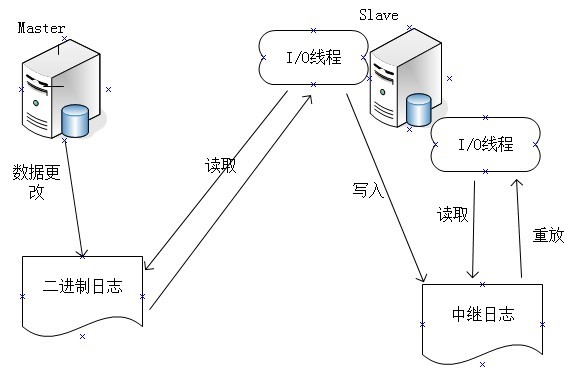
sync\_relay\_log\_info=1每间隔多少事务刷新relay-log.info，如果是table（innodb）设置无效，每个事务都会更新

注： 在从节点中 master.info是记录在主节点复制位置的文件。

relylog\_info: 本地将来至于主节点的哪一个二进制文件中position并且保存文本地哪一个中继日志中的哪一个postion. 从节点启动时也需要根据relay-log.info定位本地relay-log.

Mysql主从复制

工作原理图：



**主从复制的原理：**

分为同步复制和异步复制，实际复制架构中大部分为异步复制。 复制的基本过程如下：

1).Slave上面的IO进程连接上Master，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容；

2).Master接收到来自Slave的IO进程的请求后，通过负责复制的IO进程根据请求信息读取制定日志指定位置之后的日志信息，返回给Slave 的IO进程。返回信息中除了日志所包含的信息之外，还包括本次返回的信息已经到Master端的bin-log文件的名称以及bin-log的位置；

3).Slave的IO进程接收到信息后，将接收到的日志内容依次添加到Slave端的relay-log文件的最末端，并将读取到的Master端的 bin-log的文件名和位置记录到master-info文件中，以便在下一次读取的时候能够清楚的告诉Master“我需要从某个bin-log的哪个位置开始往后的日志内容，请发给我”；

4).Slave的Sql进程检测到relay-log中新增加了内容后，会马上解析relay-log的内容成为在Master端真实执行时候的那些可执行的内容，并在自身执行。

**环境描述**

操作系统：CentOS6.3\_x64

主服务器master：192.168.0.202

从服务器slave：192.168.0.203

一、mysql主从复制

**1、主从安装mysql，版本一致**

我们装的是mysql-5.5.30.tar.gz这里省略...请参考<http://going.blog.51cto.com/7876557/1290440>

**2、修改master，slave服务器**

master服务器配置：

vi /usr/local/mysql/etc/my.cnf

[mysqld]

server-id=202 #设置服务器唯一的id，默认是1，我们设置ip最后一段，slave设置203

log-bin=mysql-bin # 启用二进制日志

#binlog-ignore-db = mysql,information\_schema #忽略写入binlog的库

slave服务器配置：

vi /usr/local/mysql/etc/my.cnf

[mysqld]

server-id=203

replicate-do-db = abc #只同步abc库

slave-skip-errors = all #忽略因复制出现的所有错误

**3、重启主从服务器mysql**

/etc/init.d/mysqld restart

**4、在主服务器上建立帐户并授权slave**

mysql> mysql -u root -p123.com

mysql> GRANT REPLICATION SLAVE ON \*.\* to ‘sync’@‘192.168.1.2’ identified by ‘1234.com’; #replication:复制

**5、查看主数据库状态**

mysql> show master status;

+------------------+----------+--------------+------------------+

| File | Position | Binlog\_Do\_DB | Binlog\_Ignore\_DB |

+------------------+----------+--------------+------------------+

| mysql-bin.000002 | 263 | | |

+------------------+----------+--------------+------------------+

**6、配置从数据库**

mysql> change master to

 -> master\_host='192.168.0.202',

 -> master\_user='sync',

 -> master\_password='1234.com',

 -> master\_log\_file='mysql-bin.000002',

 -> master\_log\_pos=263;

#Log和pos是master上随机获取的。这段也可以写到my.cnf里面。

**7、启动slave同步进程并查看状态**

mysql> start slave;

[](https://s4.51cto.com/attachment/201310/103403849.jpg)

其中Slave\_IO\_Running 与 Slave\_SQL\_Running 的值都必须为YES，才表明状态正常。

**8、验证主从同步**

在主mysql创建数据库abc，再从mysql查看已经同步成功！

mysql> create database abc;

mysql> show databases;

+--------------------+

| Database |

+--------------------+

| information\_schema |

| abc |

| mysql |

| performance\_schema |

| test |

+--------------------+

在slave启动报错：

“Last\_IO\_Error: Got fatal error 1236 from master when reading data from binary log: ‘Could not find first log file name in binary log index file’”

解决：报错的原因主要是slave设置master的二进制文件名或pos值不对应！

先flush logs；清空日志，在查看下主数据库的状态 show master status;看下日志文件名字和position值；

再在slave中，执行：CHANGE MASTER TO MASTER\_LOG\_FILE=‘二进制日志名’,MASTER\_LOG\_POS=值;

最后启动同步进程：start slave；

**002、MySQL支持以下几种类型的索引？**

      （1）B-Tree索引

      （2）哈希索引

      （3）空间数据索引（R-Tree）

      （4）全文索引

      （5）其他索引类别

二、索引类型

Mysql目前主要有以下几种索引类型：FULLTEXT，HASH，BTREE，RTREE。

1. FULLTEXT

即为全文索引，目前只有MyISAM引擎支持。其可以在CREATE TABLE ，ALTER TABLE ，CREATE INDEX 使用，不过目前只有 CHAR、VARCHAR ，TEXT 列上可以创建全文索引。

全文索引并不是和MyISAM一起诞生的，它的出现是为了解决WHERE name LIKE “%word%"这类针对文本的模糊查询效率较低的问题。

2. HASH

由于HASH的唯一（几乎100%的唯一）及类似键值对的形式，很适合作为索引。

HASH索引可以一次定位，不需要像树形索引那样逐层查找,因此具有极高的效率。但是，这种高效是有条件的，即只在“=”和“in”条件下高效，对于范围查询、排序及组合索引仍然效率不高。

3. BTREE

BTREE索引就是一种将索引值按一定的算法，存入一个树形的数据结构中（二叉树），每次查询都是从树的入口root开始，依次遍历node，获取leaf。这是MySQL里默认和最常用的索引类型。

4. RTREE

RTREE在MySQL很少使用，仅支持geometry数据类型，支持该类型的存储引擎只有MyISAM、BDb、InnoDb、NDb、Archive几种。

相对于BTREE，RTREE的优势在于范围查找。

ps. 此段详细内容见此片博文：Mysql几种索引类型的区别及适用情况

三、索引种类

普通索引：仅加速查询

唯一索引：加速查询 + 列值唯一（可以有null）

主键索引：加速查询 + 列值唯一（不可以有null）+ 表中只有一个

组合索引：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并

全文索引：对文本的内容进行分词，进行搜索

ps.

索引合并，使用多个单列索引组合搜索

覆盖索引，select的数据列只用从索引中就能够取得，不必读取数据行，换句话说查询列要被所建的索引覆盖

从物理存储角度

1、聚集索引（clustered index）

2、非聚集索引（non-clustered index）

**003、索引是怎么加快查询速度的？**

答：支支吾吾，它是一种新的数据结构。

打个来比方，索引就是书的目录，有了目录，查询起来当然更方便。可以分为聚自族,非聚族索引 聚族索引一个表只能有一个，它查询速度快，占有空间小非聚族索百引可以有多个，但查询速度慢，占有空间大。所有的素度引都有一个缺点，如果表里面的数据频繁更新问的话，那建索引就要慎重，索引提答高查询速度，但降低更新速度

为什么能够提高查询速度？

索引就是通过事先排好序，从而在查找时可以应用二分查找等高效率的算法。  
一般的顺序查找，复杂度为O(n)，而二分查找复杂度为O(log2n)。当n很大时，二者的效率相差及其悬殊。  
举个例子：  
表中有一百万条数据，需要在其中寻找一条特定id的数据。如果顺序查找，平均需要查找50万条数据。而用二分法，至多不超过20次就能找到。二者的效率差了2.5万倍！

在一个或者一些字段需要频繁用作查询条件，并且表数据较多的时候，创建索引会明显提高查询速度，因为可由全表扫描改成索引扫描。

（无索引时全表扫描也就是要逐条扫描全部记录，直到找完符合条件的，索引扫描可以直接定位）

不管数据表有无索引，首先在SGA的[数据缓冲区](http://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E7%BC%93%E5%86%B2%E5%8C%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPh7-Pj6znWT1Phf3rAnk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH0drHf4PWbd)中查找所需要的数据，如果[数据缓冲区](http://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E7%BC%93%E5%86%B2%E5%8C%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPh7-Pj6znWT1Phf3rAnk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH0drHf4PWbd)中没有需要的数据时，服务器进程才去读磁盘。  
1、无索引，直接去读表数据存放的磁盘块，读到[数据缓冲区](http://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E7%BC%93%E5%86%B2%E5%8C%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPh7-Pj6znWT1Phf3rAnk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH0drHf4PWbd)中再查找需要的数据。  
2、有索引，先读入索引表，通过索引表直接找到所需数据的物理地址，并把数据读入数据缓冲区中。

索引有什么副作用吗？

（1）索引是有大量数据的时候才建立的，没有大量数据反而会浪费时间，因为索引是使用二叉树建立.

（2）当一个系统查询比较频繁，而新建，修改等操作比较少时，可以创建索引，这样查询的速度会比以前快很多，同时也带来弊端，就是新建或修改等操作时，比没有索引或没有建立覆盖索引时的要慢。

（3）索引并不是越多越好，太多索引会占用很多的索引表空间，甚至比存储一条记录更多。  
对于需要频繁新增记录的表，最好不要创建索引，没有索引的表，执行insert、append都很快，有了索引以后，会多一个维护索引的操作，一些大表可能导致insert 速度非常慢。

索引优点中，最重要的就是加快查询速度。

索引有如下3个特点，这3点虽然看上去，句句都是废话，但仔细想想就发现，这3个特点包含的东西，远远超过了3句话所涵盖的。

（1）索引有序

创建索引后，之所以查询速度会快，是由于创建索引时，会按照字段进行排序，如果有多个字段，那么会按照多个字段逐个排序。

在查找的时候，可以用类似二分查找的算法，快速判断要找的值在哪个B树的分支。

（2）索引高度不会太高

在数据量比较少的情况下，建不建索引，效果不大。

但当数据量越来越大，比如达到百万、甚至亿级的时候，查找数据就会越来越慢，为什么会那么慢呢？

在没有索引的情况下，要从头到尾的找，比如表里有1亿条数据，那么就得扫描这1亿条数据，最后可能只找到了10条满足要求的数据。

在这个过程中，大部分都是做的无用功。。。

而创建索引后，生成了B树，虽然有1亿条数据，但是这个B树不会提高，一般不超过7层，那么这个层数是如何影响查找效率的呢？

因为索引存储的值，都是在叶子节点上，所以要找到值，就必须要从B树的根开始找，如果有6层，那么相当于根开始到叶子，一共要进过6次的判断，最后找到想要的值。

所以，层数越少，比较次数少，需要读取的数据叶子数，也少，IO次数就少，那就会更快。

（3）索引中包含字段的值

给某个字段建索引，那么这个索引中就会包含这个字段的值，这句话看上去像废话，但很有用。

在某些情况下，索引不起过滤的所用，貌似这个索引好像就没什么用。

其实这个时候，对索引的访问方式，就不是原来的索引查找，而是索引扫描，也就是说sql要访问的数据，都可以直接从索引中出来，那么就没必要去查表的数据，索引起到的是覆盖查询的目的，相对于查整个表来说，索引页数更少，IO就少，所以查询时间就会小很多。

**004、索引底层是什么数据结构？**

答：不知道。

索引是帮助MySQL高效获取数据的排好序的数据结构（容易忽略的点：排好序）（形象点就是教科书的目录）

索引存储在文件里（也就是说有IO操作）

索引结构：

这里说说在几种数据结构中，mysql为什么选择hash，B+Tree

1. 二叉树
2. 红黑树
3. hash
4. BTree
5. B+Tree

首先，如果数据没有索引，那么我们读取数据是这样的

MySQL官方对索引的定义为:索引(Index)是帮助MySQL高效获取数据的数据结构

大家使用索引有没有想过这个问题？为什么索引能够帮助mysql高效获取数据？我一一给大家道来！在给大家讲之前，先更大家分享一些计算机基础知识，有助于理解

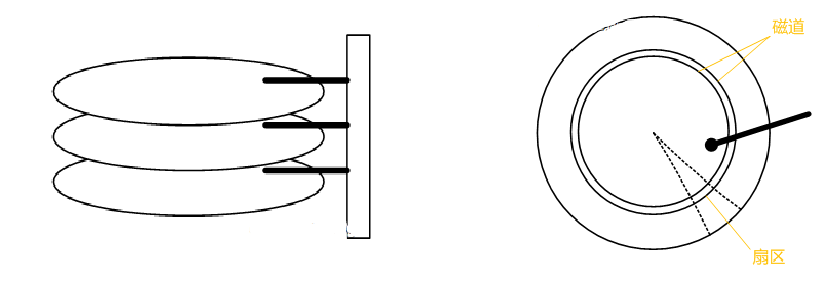
我们都知道mysql数据是已文件的形式存储在磁盘上的。

那磁盘是有一圈一圈的磁道组成的

磁头移动到不同磁道，磁盘旋转，这样就可以读取到数据

磁盘存取原理 1.寻道时间(速度慢，费时) 2.旋转时间(速度较快)

cpu读取数据都是从内存去读，内存去磁盘读取数据，内存读取磁盘数据大小都是已页的大小单位，一页=10kb



总结：

通过上面的一些知识，我们知道当磁头移动到另一个磁道读取诗句就是我们常说的一次I/O操作，但是我们知道mysql数据是分布到不同的磁道上的，每次读取数据都要把所有磁道读取一遍，那我们进行I/O次数就很多了，查询效率就很低。

那索引就是把索引字段数据的地址保存起来，来帮助mysql直接定位到哪个磁道的哪个扇区，这样就减少I/O操作了，自然查询效率就提高了

B-Tree和B+Tree

目前大部分数据库系统及文件系统都采用B-Tree或其变种B+Tree作为索引结构，在本文的下一节会结合存储器原理及计算机存取原理讨论为什么B-Tree和B+Tree在被如此广泛用于索引，这一节先单纯从数据结构角度描述它们。

a.二叉树

二叉树是n(n>=0)个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树组成。

不使用原因：会出现极端情况，一个节点只有一度，就是只有一个子节点，那读取树的一层就是一次I/O，那性能也不好

b.红黑树

红黑树即为平衡二叉树的一种

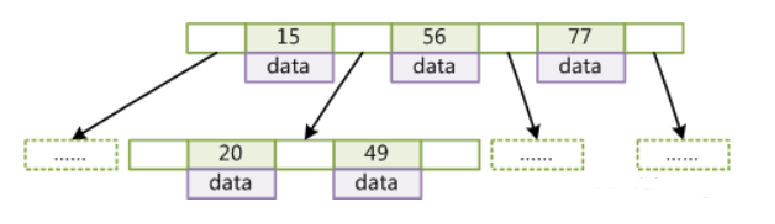
不使用原因：极端情况下，一个节点有2个子节点，那就出现一层只有2个节点的情况，这种性能也不好

c.Hash

不使用原因：Hash是把索引数据进行Hash算法对应一个地址，我们会发现这个好像性能很好啊，直接找到，但是我们想想，它能满足我们日常开发大部分情况吗？比如通过大于或者小于去筛选数据，所以说也不合适，当然mysql还是提供了Hash索引，毕竟有些场合还是用起来也不错

d.B Tree

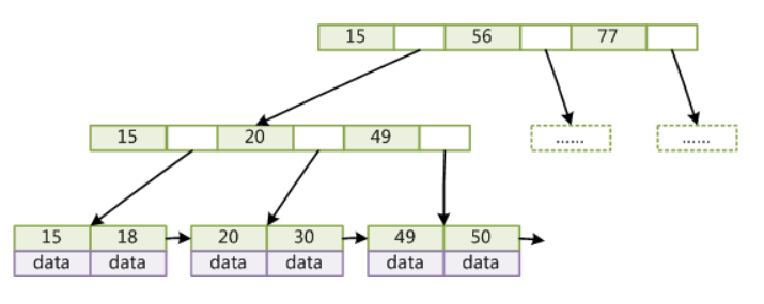
1.度(Degree)-节点的数据存储个数 2.叶节点具有相同的深度 3.叶节点的指针为空 4.节点中的数据key从左到右递增排列



不使用原因：虽然解决了每一层的节点数的极端情况下，但是我们会发现每个节点存储了索引和数据，那一层能存储的数据太多也不好,毕竟内存能读取的数据大小就是10kb

e.B+Tree

1.B+Tree(B-Tree变种) 2.非叶子节点不存储data，只存储key，可以增大度 3.叶子节点不存储指针 4.顺序访问指针，提高区间访问的性能



使用原因：设计有几个方面

1.非叶子节点不存储data，只存储key，可以增大度

2.叶子节点不存储指针

3.顺序访问指针，提高区间访问的性能

3.B+Tree索引的性能分析

一般使用磁盘I/O次数评价索引结构的优劣

预读：磁盘一般会顺序向后读取一定长度的数据(页的整数倍)放入内存

局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用

B+Tree节点的大小设为等于一个页，每次新建节点直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，就实现了一个节点的载入只需一次I/O

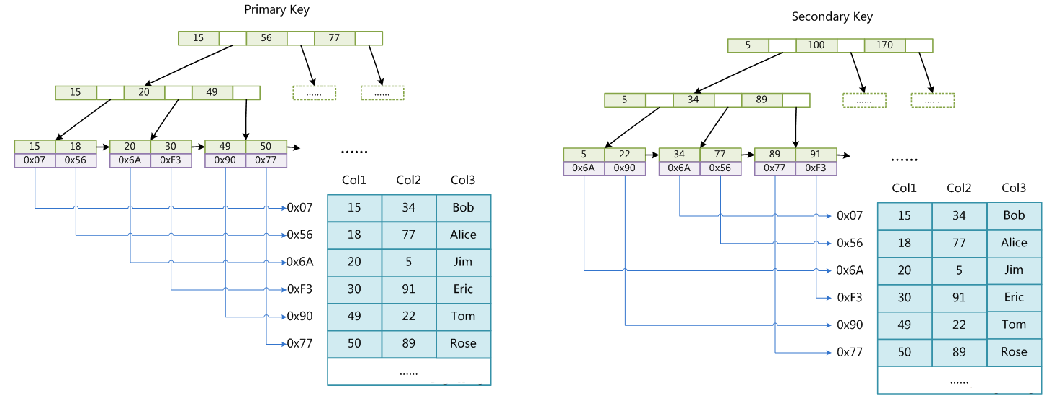
B+Tree的度d一般会超过100，因此h非常小(一般为3到5之间)

4.不同的存储引擎有不同的索引实现

1.MyISAM索引实现(非聚集) 2.InnoDB索引实现(聚集)

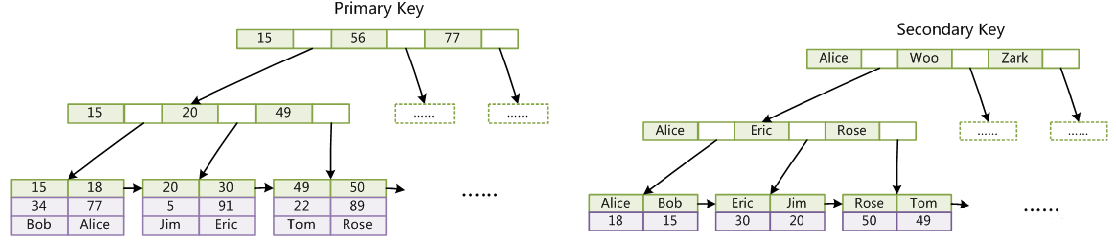
a.MyISAM索引实现(非聚集)

--->MyISAM索引文件和数据文件是分离的



b.InnoDB索引实现(聚集)

1.数据文件本身就是索引文件 2.表数据文件本身就是按B+Tree组织的一个索引结构文件 3.聚集索引-叶节点包含了完整的数据记录 4.为什么InnoDB表必须有主键，并且推荐使用整型的自增主键？ 5.为什么非主键索引结构叶子节点存储的是主键值？(一致性和节省存储空间)



5.联合索引结构

---->联合索引的底层存储结构长什么样？

6.索引最左前缀原理：

1、顾名思义，就是最左优先，在创建多列索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。

2、最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

3、=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

### 采用多线程对数据库读写引发冲突

[**多人操作sqlite3数据库冲突问题解决方法**](https://www.cnblogs.com/battlescars/p/cpp_sqlite3_op_race.html)**（用的不是python语言，看思路）**

**问题描述：**sqlite3数据放置在某一台电脑的某个共享文件夹下，操作数据库的应用程序安装在同一局域网下的很多台电脑上，由于存在多人同时使用该应用程序，所以存在多人同时操作数据库的情况。经过测试发现，最常见的情况是当两人或者多人往数据库中写入数据时，只有其中一个写入成功，其他数据都写入失败。

**解决方案分析：**

由于本人编写程序未MFC应用程序，所以尝试使用windows互斥量mutex，具体的使用方法如下：

bool CMFCApplication2Dlg::Lock()

{

　　m\_pMutex = CreateMutex(NULL, false, L"txt\_mutex");

　　if (NULL == m\_pMutex)

　　{

　　　　return false;

　　}

　　DWORD nRet = WaitForSingleObject(m\_pMutex, INFINITE);

　　if (nRet != WAIT\_OBJECT\_0)

　　{

　　　　return false;

　　}

　　return true;

}

bool CMFCApplication2Dlg::UnLock()

{

　　return ReleaseMutex(&m\_pMutex);

}

在某用户开始进行写入操作时，先调用Lock()获取mutxt，写入完成之后调用UnLock()释放mutxt。然并卵，该方法并不奏效。（可能由于本人对windows多线程/多进程编程这一块太过生疏，所以无法利用这方面的知识来解决这个问题，如果有大神知道解决方法，求不吝赐教）

所以经过一番思考之后，决定使用在共享盘的那台电脑上跑一个服务端小程序来防止数据库的操作冲突。

**具体实现方法如下：**

思路分析：在服务端每收到一个客户端的连接请求之后，都创建一个新的线程来处理相应的操作，新的线程不断的去获取客户端发来的消息，当客户端发来的消息是”开始操作”时，线程将尝试获取互斥量mutxt，获取成功之后将给客户端发送回复消息，当该线程接收到”操作结束”的消息时，线程将释放互斥量mutxt,并且会断开该客户端与服务端的socket连接。

服务端代码：

#include <WinSock2.h>

#include <Windows.h>

#include <iostream>

#include <string>

#pragma comment(lib, "ws2\_32.lib")

using namespace std;

static HANDLE m\_mutex = INVALID\_HANDLE\_VALUE;

DWORD WINAPI AnswerThread(LPVOID lparam)

{

SOCKET ClientSocket = (SOCKET)(LPVOID)lparam;

int bytesRecv;

while (1)

{

bytesRecv = SOCKET\_ERROR;

char sendbuff[3] = "ok";

char recvbuf[20] = "";

for (int i = 0; i<(int)strlen(recvbuf); i++)

{

recvbuf[i] = '\0';

}

while (bytesRecv == SOCKET\_ERROR)

{

bytesRecv = recv(ClientSocket, recvbuf, sizeof(recvbuf), 0);

}

string recved = recvbuf;

if (recved == "op\_begin")

{

WaitForSingleObject(m\_mutex, INFINITE);

cout << "op\_begin" << endl;

send(ClientSocket, sendbuff, sizeof(sendbuff), 0);

}

if (recved == "op\_end")

{

cout << "op\_end" << endl;

ReleaseMutex(&m\_mutex);

closesocket(ClientSocket);

return 0;

}

}

return 0;

}

int main()

{

WSADATA wsaData;

int iRet = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iRet != NO\_ERROR)

printf("Error at WSAStartup()\n");

SOCKET m\_socket;

m\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (m\_socket == INVALID\_SOCKET)

{

printf("Error at socket():%ld\n", WSAGetLastError());

WSACleanup();

return 0;

}

SOCKADDR\_IN service;

service.sin\_family = AF\_INET;

service.sin\_addr.S\_un.S\_addr = htonl(INADDR\_ANY);

service.sin\_port = htons(2501);

if (bind(m\_socket, (SOCKADDR\*)&service, sizeof(service)) == SOCKET\_ERROR)

{

printf("bind() failed.\n");

closesocket(m\_socket);

return 0;

}

else

printf("bind ok.\n");

if (listen(m\_socket, 20) == SOCKET\_ERROR)

printf("Error listening on socket.\n");

else

printf("listening ok.\n");

SOCKET AcceptSocket;

printf("waiting for a client to connect...\n");

m\_mutex = CreateMutex(NULL, FALSE, L"Mutex");

if (!m\_mutex)

{

cout << "Failed to CreateMutex !" << endl;

return 0;

}

int count = 0;

while (1)

{

AcceptSocket = SOCKET\_ERROR;

while (AcceptSocket == SOCKET\_ERROR)

{

AcceptSocket = accept(m\_socket, NULL, NULL);

}

count++;

printf("client num %d connected.\n", count);

DWORD dwThreadId;

HANDLE hThread;

hThread = CreateThread(NULL, NULL, AnswerThread, (LPVOID)AcceptSocket, 0, &dwThreadId);

if (hThread == NULL)

{

printf("CreatThread AnswerThread() failed.\n");

}

else

{

printf("create thread %d ok.\n", count);

}

CloseHandle(hThread);

}

closesocket(m\_socket);

WSACleanup();

}

客户端代码：(进入数据库操作前)

WSADATA wsaData;

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)

{

　　TRACE("Failed to load Winsock");

　　return;

}

string txtPath = save\_path + "\\ip.txt";

ifstream infile(txtPath);

string ip;

getline(infile, ip);

infile.close();

SOCKADDR\_IN addrSrv;

addrSrv.sin\_family = AF\_INET;

addrSrv.sin\_port = htons(2501);

addrSrv.sin\_addr.S\_un.S\_addr = inet\_addr(ip.c\_str());

sockClient = socket(AF\_INET, SOCK\_STREAM, 0);

if (SOCKET\_ERROR == sockClient){

　　TRACE("Socket() error:%d", WSAGetLastError());

　　return;

}

if (connect(sockClient, (SOCKADDR\*)&addrSrv, sizeof(addrSrv)) == INVALID\_SOCKET){

　　TRACE("Connect failed:%d", WSAGetLastError());

　　return;

}

char buff[9] = "op\_begin";

send(sockClient, buff, sizeof(buff), 0);

int bytesRecv = SOCKET\_ERROR;

char recvbuf[3] = "";

for (int i = 0; i<(int)strlen(recvbuf); i++)

{

　　recvbuf[i] = '\0';

}

while (bytesRecv == SOCKET\_ERROR)

{

　　CMessageDlg message;

　　message.DoModal();

　　bytesRecv = recv(sockClient, recvbuf, sizeof(recvbuf), 0);

}

客户端代码：(数据库操作完成之后)

char buff[7] = "op\_end";

send(sockClient, buff, sizeof(buff), 0);

closesocket(sockClient);

注：由于经验不足，本方法可能有很多内存释放，资源利用等细节没有考虑到，所以本方法仅供参考。另外有关socket编程部分的代码参考：

SQL多用户访问数据库怎样解决冲突？

sql多用户访问数据库其实就是事务并发，会引起如下问题：  
1、脏读：一个事务读取到了另外一个事务没有提交的数据  
事务1：更新一条e799bee5baa6e997aee7ad94e59b9ee7ad9431333363396466数据  
事务2：读取事务1更新的记录  
事务1：调用commit进行提交  
此时事务2读取到的数据是保存在数据库内存中的数据，称为脏读。  
读到的数据为脏数据  
详细解释：  
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。因为这个数据是还没有提交的数据，那么另外一个  
事务读到的这个数据是脏数据，依据脏数据所做的操作可能是不正确的。  
2、不可重复读：在同一事务中，两次读取同一数据，得到内容不同  
事务1：查询一条记录  
事务2：更新事务1查询的记录  
事务2：调用commit进行提交  
事务1：再次查询上次的记录  
此时事务1对同一数据查询了两次，可得到的内容不同，称为不可重复读。  
3、幻读：同一事务中，用同样的操作读取两次，得到的记录数不相同  
事务1：查询表中所有记录  
事务2：插入一条记录  
事务2：调用commit进行提交  
事务1：再次查询表中所有记录  
此时事务1两次查询到的记录是不一样的，称为幻读  
详细解释：  
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。  
处理以上隔离级别的问题，采用如下方是：  
事务隔离五种级别：  
TRANSACTION\_NONE 不使用事务。  
TRANSACTION\_READ\_UNCOMMITTED 允许脏读。  
TRANSACTION\_READ\_COMMITTED 防止脏读，最常用的隔离级别,并且是大多数数据库的默认隔离级别  
TRANSACTION\_REPEATABLE\_READ 可以防止脏读和不可重复读，  
TRANSACTION\_SERIALIZABLE 可以防止脏读，不可重复读取和幻读，（事务串行化）会降低数据库的效率  
以上的五个事务隔离级别都是在Connection接口中定义的静态常量，  
使用setTransactionIsolation(int level) 方法可以设置事务隔离级别。  
如:con.setTransactionIsolation(Connection.REPEATABLE\_READ);  
注意：事务的隔离级别受到数据库的限制，不同的数据库支持的的隔离级别不一定相同  
1 脏读：修改时加排他锁，直到事务提交后才释放，读取时加共享锁，读取完释放事务1读取数据时加上共享锁后（这 样在事务1读取数据的过程中，其他事务就不会修改该数据），不允许任何事物操作该数据，只能读取，之后1如果有更新操作，那么会转换为排他锁，其他事务更 无权参与进来读写，这样就防止了脏读问题。  
但是当事务1读取数据过程中，有可能其他事务也读取了该数据，读取完毕后共享锁释放，此时事务1修改数据，修改 完毕提交事务，其他事务再次读取数据时候发现数据不一致，就会出现不可重复读问题，所以这样不能够避免不可重复读问题。  
2 不可重复读：读取数据时加共享锁，写数据时加排他锁，都是事务提交才释放锁。读取时候不允许其他事物修改该数据，不管数据在事务过程中读取多少次，数据都是一致的，避免了不可重复读问题  
3 幻读问题：采用的是范围锁RangeS RangeS\_S模式，锁定检索范围为只读，这样就避免了幻影读问题。

**常用mysql命令**

**mysql 常用命令**

mysql 导入导出的命令。

使用mysqldump命令  
mysqldump 是命令行工具，主要用于mysql备份和还原数据。这个命令使用起来方便，直接在终端执行。

Dumping structure and contents of MySQL databases and tables.  
Usage: mysqldump [OPTIONS] database [tables]  
OR mysqldump [OPTIONS] --databases [OPTIONS] DB1 [DB2 DB3...]  
OR mysqldump [OPTIONS] --all-databases [OPTIONS]

mysqldump 的主要参数：

-h, --host=ip 主机地址  
-u, --user=name 登录用户名  
-P, --port=# Mysql连接端口  
-p, --password[=name] Mysql密码。如果不给定值，直接回车，会提示输入密码  
-d, --no-data 不包含行信息，只导出表结构

这些命令都不用刻意去记，执行 mysqldump --help ， 就可以查询使用帮助。

导出整个表(包含数据)  
mysqldump -u 用户名 -p [密码] 数据库名 > 导出的文件名  
导出表

**包含数据**

mysqldump -u 用户名 -p 数据库名 表名 > 导出的文件名

**只导出表结构**

mysqldump -u 用户名 -d -p 数据库名 表名 > 导出的文件名

导入到远程服务器  
使用管道命令，还可以将导出的数据直接导入到远程的服务器上，前提是服务器可以相互访问。

mysqldump -u 用户名 -p 数据库名 | mysql -h 远程服务器HOST 远程数据库名  
SQL文件压缩备份&还原  
mysqldump -u 用户名 -d -p 数据库名 | gzip > 文件名.sql.gz

gunzip < 文件名.sql.gz | mysql -u 用户名 -p 数据库  
示例SQL

create database test;  
use test;

create table tb1(  
f1 int auto\_increment,  
f2 varchar(50),  
primary key(f1)  
)ENGINE= INNODB default charset = utf8;

insert into tb1 values(null, 'a');  
insert into tb1 values(null, 'b');  
insert into tb1 values(null, 'c');  
insert into tb1 values(null, 'd');

create table tb2(  
t1 int auto\_increment,  
t2 varchar(50),  
primary key(t1)  
)ENGINE= INNODB default charset = utf8;

insert into tb2 values(null, 'a');  
insert into tb2 values(null, 'b');  
insert into tb2 values(null, 'c');  
insert into tb2 values(null, 'd');

select \* from tb1;  
select \* from tb2;  
mysqldump -uhomestead -p testdb > test\_ddl.sql  
mysqldump -uhomestead -p test tb1 > ~/Code/tmp/db1\_ddl.sql  
mysqldump -uhomestead -p test -d tb1 > ~/Code/tmp/db1\_ddl.sql

mysqldump -uhomestead -d -p test | gzip > backup-file.sql.gz  
gunzip < backup-file.sql.gz | mysql -uhomestead -p test  
使用source 命令  
可以在mysql控制台里，加载执行SQL文件。

# 多数据更新冲突处理机制

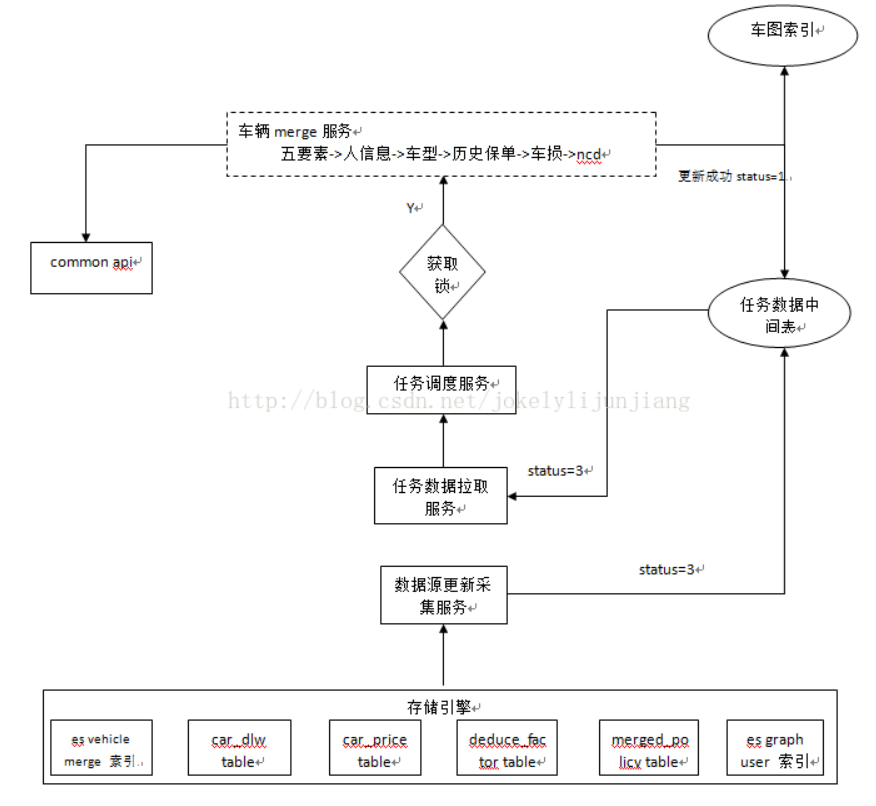
一 背景

        在做一个数据合并项目的时候，遇到了多数据源更新同一数据，导致并发更新冲突的问题。具体场景是，有一个数据实时采集服务，然后通过数据merge服务，完成最终的数据合并操作。但是在做合并操作时，有可能会存在并发冲突，后一个线程将前一个线程刚处理后的字段给覆盖。当时的存储引擎是es，虽然es本身自带版本控制功能，但是由于es自身的刷新时间间隔，以及项目本身设置的刷新时间较长（场景所需），会导致数据的不可见性，所以不能依赖es本身的冲突控制机制

二 解决方案

         为了解决冲突问题，想法是引入任务调度系统，负责具体的调度任务。首先，数据实时采集系统会将采集到的数据以标准格式存入到一张任务中间表中，然后任务调度系统会多线程并发从中间表中获取未处理的任务数据，此时引入分布式锁机制，根据设置的锁粒度，线程去尝试获取锁，获取到锁的数据则往下执行，失败则做具体的失败兼容逻辑。在锁的选择方面，我们选用的是redis，当然还有其他的方案，比如memcached，zookeeper等，各有千秋吧。当然，锁的超时时间很重要，这个要具体问题具体分析，但一定要记得有兜底机制释放你的锁。

三 简单流程图



**登入Mysql**

mysql -u root -p

mysql> set names utf8; (防止乱码，先设置字符集)  
mysql> use test;  
mysql> source ~/Code/tmp/db1\_ddl.sql  
使用into outfile 和 load data infile命令  
有时候，我们需要根据查询条件导出SQL，可以使用mysql语句：into outfile 和 load data infile。

同source命令， 都需要先登入mysql。

导出

select \* from tb1 into outfile '/var/lib/mysql-files/outfile1' fields terminated by '|' enclosed by '"' lines terminated by '\r\n' ;  
导入

load data infile "/var/lib/mysql-files/outfile1" replace into table tb1 fields terminated by '|' enclosed by '"' lines terminated by '\r\n' ;  
用这两条命令还是有注意事项：

分隔符参数

fields关键字指定了文件记段的分割格式，如果用到这个关键字，MySQL剖析器希望看到至少有下面的一个选项：

terminated by 分隔符：意思是以什么字符作为分隔符  
enclosed by 字段括起字符  
escaped by 转义字符  
terminated by 描述字段的分隔符，默认情况下是tab字符（ ）  
enclosed by 描述的是字段的括起字符。  
escaped by 描述的转义字符。默认的是反斜杠（backslash： ）  
字符集设置

load data infile '/var/lib/mysql-files/outfile1' replace into table tb1 character set utf8 fields terminated by '|' enclosed by '"' lines terminated by '\r\n' ;  
目录限制

mysqld 的secure\_file\_priv配置 ，用来限制LOAD\_FILE（）和LOAD DATA和SELECT … ，INTO OUTFILE报表到指定的目录。

导入导出一定是在设定的目录文件，否则会报错：ERROR 1290 (HY000):The MySQL server is running with the --secure-file-priv option so it cannot execute this statement

查看配置

SELECT @@secure\_file\_priv;

常用参数  
-A, --all-databases  
导出所有数据库  
--add-drop-database  
在创建数据库之前先DROP数据库  
--add-drop-table  
在创建表之前先drop表，默认true  
--add-locks  
在insert语句前加锁，默认true  
--compact  
导出更少的输出信息(用于调试)。去掉注释和头尾等结构  
-c, --complete-insert  
使用完整的insert语句(包含列名称)。  
-B, --databases  
导出几个数据库。参数后面所有名字参量都被看作数据库名。导出文件中会有'USE db\_name;'  
--default-character-set=name  
设置默认字符集  
-e, --extended-insert  
使用具有多个VALUES列的INSERT语法。这样使导出文件更小，并加速导入时的速度，默认true  
\*\*--ignore-table=name \*\*  
忽略指定的表不导出  
--insert-ignore  
在插入行时使用insert ignore  
-l, --lock-tables  
开始导出前，锁定所有表  
--dump-slave  
将主库的binlog位置和文件名追加到导出数据的文件中,该参数在在从服务器上执行，相当于执行show slave status.当设置为1时，将会以CHANGE MASTER命令输出到数据文件；设置为2时，会在change前加上注释。  
--master-data[=#]  
该选项将当前服务器的binlog的位置和文件名追加到输出文件中(show master status)。如果为1，将会输出CHANGE MASTER 命令；如果为2，输出的CHANGE MASTER命令前添加注释信息。  
-n, --no-create-db  
只导出数据，不添加create database语句，默认false  
-t, --no-create-info  
只导出数据，不添加create table语句，默认false  
-d, --no-data  
不导出任何数据，只导出数据库表结构  
-q, --quick  
不缓冲查询，直接导出到标准输出。默认true  
\*\*--replace \*\*  
插入数据时用replace  
--set-gtid-purged[=name]  
是否添加SET@@GLOBAL.GTID\_PURGED，默认auto  
--single-transaction  
使用一致性的快照来导出数据，不锁表，默认false  
--tables  
覆盖--databases (-B)参数，指定需要导出的表名  
-w, --where=name  
按照条件导出，只能导出单个表  
-u -h -P -p  
用户名，主机名，端口，密码

常用方法  
1、导出所有库  
mysqldump -uroot -p123 --single-transaction --all-databases >all.sql  
该命令会导出包括系统数据库在内的所有数据库  
2、导出单个库  
mysqldump -uroot -p123 --single-transaction test > test.sql  
3、导出多个数据库  
mysqldump -uroot -p123 --single-transaction --databases db1 db2> db.sql  
导出文件中会有'USE db1;USE db2'  
4、导出单表  
mysqldump -uroot -p123 --single-transaction test table\_1 > table\_1.sql  
5、导出多表  
mysqldump -uroot -p123 --single-transaction test --tables table\_1 table\_2> table.sql  
6、只导表结构  
mysqldump -uroot -p123 --single-transaction test -d > test\_structure.sql  
7、只导出数据  
mysqldump -uroot -p123 --no-create-info --single-transaction --skip-add-locks --compact -c --set-gtid-purged=OFF test t2 >t2.sql  
8、按照where条件导出  
mysqldump -uroot -p123 --single-transaction test table\_1 -w "id<1000000" > table\_1.sql

1. 增

* 增加一张表

CREATE TABLE `table\_name`(

...

)ENGINE=InnoDB DEFAULT CHARSET=utf8;

* 增加记录

INSERT INTO `your\_table\_name`(`column\_name`)

VALUES

('your\_value\_one'),

('your\_value\_two');

* 增加字段

ALTER TABLE `your\_table\_name`

ADD `your\_column\_name` ...

AFTER `column\_name`;

* 增加索引
  + 主键
* ALTER TABLE `your\_table\_name`
* ADD PRIMARY KEY your\_index\_name(your\_column\_name);
  + 唯一索引
* ALTER TABLE `your\_table\_name`
* ADD UNIQUE your\_index\_name(your\_column\_name);
  + 普通索引
* ALTER TABLE `your\_table\_name`
* ADD INDEX your\_index\_name(your\_column\_name);
  + 全文索引
* ALTER TABLE `your\_table\_name`
* ADD FULLTEXT your\_index\_name(your\_column\_name);

1. 删

* 逐行删除

DELETE FORM `table\_name`

WHERE ...;

* 清空整张表

TRUNCATE TABLE `your\_table\_name`;

* 删除表

DROP TABLE `your\_table\_name`;

* 删除字段

ALTER TABLE `your\_table\_name`

DROP `column\_name`;

* 删除索引

ALTER TABLE `your\_table\_name`

DROP INDEX your\_index\_name(your\_column\_name);

1. 改

* 变更数据

UPDATE `table\_name`

SET column\_name=your\_value

WHERE ...;

* 变更字段

ALTER TABLE `your\_table\_name`

CHANGE `your\_column\_name` `your\_column\_name` ...(变更);

* 变更字段值为另一张表的某个值

UPDATE `your\_table\_name`

AS a

JOIN `your\_anther\_table\_name`

AS b

SET a.column = b.anther\_column

WHERE a.id = b.a\_id...;

1. 查

* 普通查询

SELECT `column\_name\_one`, `column\_name\_two`

FROM `table\_name`;

* 关联查询

SELECT \*

FROM `your\_table\_name`

AS a

JOIN `your\_anther\_table\_name`

AS b

WHERE a.column\_name = b.column\_name...;

* 合计函数条件查询：WHERE 关键字无法与合计函数一起使用

SELECT aggregate\_function(column\_name)

FROM your\_table\_name

GROUP BY column\_name

HAVING aggregate\_function(column\_name)...;

* 同一个实例下跨库查询

SELECT \*

FROM database\_name.your\_table\_name

AS a

JOIN another\_database\_name.your\_another\_table\_name

AS b

WHERE a.column\_name = b.column\_name...;

1. 复制一张表结构

CREATE TABLE `your\_table\_name`

LIKE `destination\_table\_name`;

1. 完全复制一张表：表结构+全部数据

CREATE TABLE `your\_table\_name`

LIKE `destination\_table\_name`;

INSERT INTO `your\_table\_name`

SELECT \*

FROM `destination\_table\_name`;

**附录：mysql常用命令**

* 登陆： mysql -h host -u username -p
* 列出数据库：SHOW DATABESES;
* 列出表:SHOW TABLES;
* 列出表结构:DESC table\_name
* 使用一个数据库：USE database\_name;
* 导入：source 'file';
* 导出：mysqldump -h 127.0.0.1 -u root -p "database\_name" "table\_name" --where="condition" > file\_name.sql;
* 查看慢日志：mysqldumpslow -s [c:按记录次数排序/t:时间/l:锁定时间/r:返回的记录数] -t [n:前n条数据] -g "正则"　/path
* 新增用户： insert into user(Host, User, authentication\_string) value('localhost', 'username', password('pwd'))

**mysql 5.7 新增用户**

// 插入新用户

insert into mysql.user(Host, User, authentication\_string, ssl\_cipher, x509\_issuer, x509\_subject

value('localhost', 'username', password('password'), '', '', '');

// 数据库授权

grant all privileges on dbname.name.\* to username@localhost identified by 'password';

// 刷新权限信息

FLUSH PRIVILEGES;

写得不错的：

**002、数据库连接是单例吗？ 有必要实现多例吗？**

 　   单例数据库连接在连接池中只有一个实例，对系统资源的开销比较少，甚至可以长时间保持连接不回收，以节省创建连接和回收连接的时间。

　　 但这样的连接在多用户并发时不能提供足够的效率，形象的来讲就是大家要排队。  初级程序员的作法是每个用户需求过来都打开一次连接，用完回收掉。

　 　进阶的做法是建立一个连接池，连接池里面给定一些已打开的连接，用程序控制这些连接的分配与调度

　     数据库链接用单例模式的原因：

        单例只保留一个对象，可以减少系统资源开销。

 　　提高创建速度，每次都获取已经存在的对象因此提高创建速度全局共享对象。

 　　单例在系统中只存在一个对象实例，因此任何地方使用此对象都是一个对象避免多实例创建使用时产生的逻辑错误。

　　 例模式是一种常用的软件设计模式，它的核心结构只包含一个被称为单例的特殊类。它的目的是保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。

　　 单例模式有3种实现方式：懒汉式、饿汉式和双重锁的形式。

　　单例模式优点 只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等  因为类控制了实例化过程，所以类可以灵活更改实例化过程

　　单例模式会阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例。  单例的缺点  就是不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，

　　单例就会引起数据的错误，不能保存彼此的状态。 用单例模式，就是在适用其优点的状态下使用。

　　　　比如：   要进一个房间（数据库），就为这个房间开了一扇门（数据库类），一般情况下是一个人开一扇门，   不管你进出（数据库操作）这个房间多少次，门就这一扇（单例），

　　　　当然一个人也可以开很多扇门（非单例），   但你知道一个房间能开的门的数量是有限的，因此你不使用单例的话，一是性能慢一些，二是走别人的门，让别人无门可进。。

**MySQL**

**01.NoSQL和SQL数据库之间有什么不同？**

* SQL数据库主要称为关系数据库;
* NoSQL数据库主要称为非关系数据库或分布式数据库;
* **结构**

  SQL数据库是最广泛使用的选项之一，使其成为安全的选择，尤其适用于复杂的查询。但它具有一定的限制性。SQL数据库是基于表的，它要求您在使用之前使用预定义模式来确定数据的结构。此外，您的所有数据都必须遵循相同的结构。这可能需要大量的前期准备，这意味着结构的变化既困难又对整个系统造成破坏。

  NoSQL数据库具有非结构化数据的动态模式。他是键值对，基于文档的，图形数据库或宽列存储。数据以多种方式存储，这意味着它可以是面向文档，面向列，基于图形或组织为KeyValue存储。这种灵活性意味着可以在没有首先定义结构的情况下创建文档。每个文档也可以有自己独特的结构。语法因数据库而异，您可以随时添加字段。

* **可伸缩性**

  SQL数据库是可垂直扩展的。这意味着您可以通过增加RAM，CPU或SSD等功能来增加单个服务器的负载。

  NoSQL数据库可以横向扩展。这意味着您可以通过分片或在NoSQL数据库中添加更多服务器来处理更多流量。

* **遵循的属性**

SQL数据库遵循ACID属性（原子性，一致性，隔离性和持久性）

NoSQL数据库遵循Brewers CAP定理（一致性，可用性和分区容差）。

| **SQL** | **NoSQL** |
| --- | --- |
| 关系数据库管理系统 | 非关系或分布式数据库系统 |
| 静态或预定义的架构 | 动态架构 |
| 不适用于分层数据存储 | 适合分层数据存储 |
| 适合复杂查询 | 不太适合复杂的查询 |
| 可垂直扩展 | 可横向扩展 |
| Mysql/Oracle/PostgreSQL | MongoDB/Redis/RavenDB |

**02.MySQL支持哪些存储引擎?**

MySQL支持多种存储引擎,比如InnoDB,MyISAM,Memory,Archive等等.在大多数的情况下,直接选择使用InnoDB引擎都是最合适的,InnoDB也是MySQL的默认存储引擎.

**06.简单描述mysql中，索引，主键，唯一索引，联合索引的区别，对数据库的性能有什么影响（从读写两方面）？**

* 索引是一种特殊的文件，它们包含着对数据表里所有记录的引用指针。
* 普通索引(由关键字KEY或INDEX定义的索引)的唯一任务是加快对数据的访问速度。普通索引允许被索引的数据列包含重复的值。
* 唯一索引可以保证数据记录的唯一性。在数据列创建索引的时候添加关键字UNIQUE把它定义为唯一索引，能确保定某个数据列只包含彼此各不相同的值。
* 主键，是一种特殊的唯一索引，在一张表中只能定义一个主键索引，主键用于唯一标识一条记录，使用关键字 PRIMARY KEY 来创建。
* 联合索引可以涵盖多个数据列联合查询。
* 索引可以极大的提高数据的查询速度，但是会降低插入、删除、更新表的速度，因为在执行这些写操作时，还要操作索引文件。

**07.主键、外键和索引的区别？**

**定义：**

* 主键：唯一标识一条记录，不能有重复的，不允许为空
* 外键：表的外键是另一表的主键, 外键可以有重复的, 可以是空值
* 索引：该字段没有重复值，但可以有一个空值

**作用：**

* 主键：用来保证数据完整性
* 外键：用来和其他表建立联系用的
* 索引：是提高查询排序的速度

**个数：**

* 主键：主键只能有一个
* 外键：一个表可以有多个外键
* 索引：一个表可以有多个唯一索引

**08.索引的底层实现原理？**

实现原理：B+树，经过优化的B+树。主要是在所有的叶子结点中增加了指向下一个叶子节点的指针，因此InnoDB建议为大部分表使用默认自增的主键作为主索引。

**09.建立索引有什么原则？**

1. 选择唯一性建立索引
2. 为经常需要排序、分组和联合操作的字段建立索引
3. 为常作为查询条件的字段建立索引
4. 限制索引的数目
5. 尽量使用数据量少的索引
6. 尽量使用前缀来索引
7. 删除不再使用或者很少使用的索引

**10.索引的优点和缺点是什么？**

优点

1. 快速访问数据表中的特定信息，提高检索速度
2. 创建唯一性索引，保证数据库表中每一行数据的唯一性。
3. 加速表和表之间的连接
4. 减少查询中分组和排序的时间

缺点：

1. 创建索引和维护索引需要耗费时间，随着数据量增加而增加；
2. 每个索引都需要占用物理空间，随着数据量增加而增加
3. 当对表进行增、删、改、的时候索引也要动态维护，这样就降低了数据的维护速度。

**10.什么情况下不宜建立索引？**

* 对于查询中很少涉及的列或者重复值比较多的列，不宜建立索引。
* 对于一些特殊的数据类型，不宜建立索引，比如文本字段（text）等

**11.什么是事务，其有哪些特性？**

* 事务是一系列的数据库操作，是数据库应用的基本逻辑单位。事务是被绑定在一起作为一个逻辑工作单元的SQL语句分组，如果任何一个语句操作失败那么整个操作就被失败，以后操作就会回滚到操作前状态，或者是上有个节点。为了确保要么执行，要么不执行，就可以使用事务。要将有组语句作为事务考虑，就需要通过ACID测试，即原子性，一致性，隔离性和持久性。==简单一句话：事务就是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。==

事务特性：

1. **原子性**：即不可分割性，事务要么执行，要么不执行。
2. **一致性**：事务的执行使得数据库从一种一致性的正确状态转换成另一种一致性的正确状态
3. **隔离性**：事务操作之间彼此独立和透明互不影响。在事务正确提交之前，不允许把该事务对数据的任何修改提供给其他事务，
4. **持久性**：事务正确提交后，其结果将永久保存在数据库中。即使系统崩溃、修改的数据也不会丢失。

**12.事务有哪些隔离级别？**

1. **未提交读(Read Uncommitted)**：允许脏读，其他事务只要修改了数据，即使未提交，本事务也能看到修改后的数据值。也就是可能读取到其他会话中未提交事务修改的数据。
2. **已提交读(Read Committed)**：如果一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，那么这种隔离级别就称之为已提交读。Oracle等多数数据库默认都是该级别 (不重复读)。
3. **可重复读(Repeated Read)**：可重复读。一个事务只能读到另一个已经提交的事务修改过的数据，但是第一次读过某条记录后，即使其他事务修改了该记录的值并且提交，该事务之后再读该条记录时，读到的仍是第一次读到的值，而不是每次都读到不同的数据。那么这种隔离级别就称之为可重复读。它是mysql默认的隔离级别。
4. **串行读(Serializable)**：如果一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先的事务再次按照该条件查询时，能把另一个事务插入的记录也读出来，那就意味着发生了幻读。

**13.Mysql中的事务回滚机制概述？**

事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位，事务回滚是指将该事务已经完成的对数据库的更新操作撤销。如要同时修改数据库中两个不同表时，如果它们不是一个事务的话，当第一个表修改完，可能第二个表修改过程中出现了异常而没能修改，此时就只有第二个表依旧是未修改之前的状态，而第一个表已经被修改完毕。而当你把它们设定为同一个事务的时候，当第一个表修改完，第二表修改出现异常而没能修改，第一个表和第二个表都要回到未修改的状态，这就是所谓的事务回滚

**14.SQL注入的主要特点？**

变种极多，攻击简单，危害极大

主要危害：

1. 未经授权操作数据库的数据
2. 恶意纂改网页
3. 私自添加系统账号或者使数据库使用者账号
4. 网页挂木马

**15.SQL语言包括哪几部分？每部分都有哪些操作关键字？**

**SQL语言包括数据定义(DDL)、数据操纵(DML),数据控制(DCL)和数据查询（DQL）四个部分。**

* 数据定义：Create Table,Alter Table,Drop Table, Craete/Drop Index等
* 数据操纵：Select ,insert,update,delete,
* 数据控制：grant,revoke
* 数据查询：select

**16.完整性约束包括哪些？**

数据完整性是指数据的精确和可靠性。

分为以下四类：

1. **实体完整性**：规定表的每一行在表中是惟一的实体。
2. **域完整性**：是指表中的列必须满足某种特定的数据类型约束，其中约束又包括取值范围、精度等规定。
3. **参照完整性**：是指两个表的主关键字和外关键字的数据应一致，保证了表之间的数据的一致性，防止了数据丢失或无意义的数据在数据库中扩散。
4. **用户定义的完整性**：不同的关系数据库系统根据其应用环境的不同，往往还需要一些特殊的约束条件。用户定义的完整性即是针对某个特定关系数据库的约束条件，它反映某一具体应用必须满足的语义要求。

**与表有关的约束**：包括列约束(NOT NULL（非空约束）)和表约束(PRIMARY KEY、foreign key、check、UNIQUE) 。

**17.什么是锁？**

数据库是一个多用户使用的共享资源。当多个用户并发地存取数据时，在数据库中就会产生多个事务同时存取同一数据的情况。若对并发操作不加控制就可能会读取和存储不正确的数据，破坏数据库的一致性。

加锁是实现数据库并发控制的一个非常重要的技术。当事务在对某个数据对象进行操作前，先向系统发出请求，对其加锁。加锁后事务就对该数据对象有了一定的控制，在该事务释放锁之前，其他的事务不能对此数据对象进行更新操作。

**基本锁类型：锁包括行级锁和表级锁**

**18.什么叫视图？游标是什么？**

* 视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，视图通常是有一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易，相比多表查询。
* 游标是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

**19.NULL是什么意思？**

NULL这个值表示UNKNOWN(未知):它不表示“”(空字符串)。对NULL这个值的任何比较都会生产一个NULL值。您不能把任何值与一个 NULL值进行比较，并在逻辑上希望获得一个答案。

**20.SQL语句中‘相关子查询’与‘非相关子查询’有什么区别？**

子查询：嵌套在其他查询中的查询称之。子查询又称内部，而包含子查询的语句称之外部查询（又称主查询）。所有的子查询可以分为两类，即相关子查询和非相关子查询

1. 非相关子查询是独立于外部查询的子查询，子查询总共执行一次，执行完毕后将值传递给外部查询。
2. 相关子查询的执行依赖于外部查询的数据，外部查询执行一行，子查询就执行一次。

故非相关子查询比相关子查询效率高

**21.char和varchar的区别？**

**char**是一种固定长度的类型，每个值都占用N个字节，如果某个长度小于N，MySQL就会在它的右边用空格字符补足。合适存储具有近似得长度（md5值,身份证，手机号）,长度比较短小的字符串。

**varchar**则是一种可变长度的类型，每个值只占用刚好够用的字节再加上一个用来记录其长度的字节。适合经常更新得字符串，更新时不会出现页分裂得情况，避免出现存储碎片，获得更好的io性能。

**22.谈谈三个范式？**

* 第一范式: 每个列都不可以再拆分
* 第二范式: 非主键列完全依赖于主键,而不能是依赖于主键的一部分.
* 第三范式: 非主键列只依赖于主键,不依赖于其他非主键.

在设计数据库结构的时候,要尽量遵守三范式,如果不遵守,必须有足够的理由.比如性能. 事实上我们经常会为了性能而妥协数据库的设计.

**23.如何进行SQL优化？**

1. **选择正确的存储引擎**

以 MySQL为例，包括有两个存储引擎 MyISAM 和 InnoDB，每个引擎都有利有弊。 MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要update一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到读操作完成。另外，MyISAM 对于 SELECT COUNT(\*) 这类的计算是超快无比的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。但是它支持“行锁” ，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

1. **优化字段的数据类型**

记住一个原则，越小的列会越快。如果一个表只会有几列罢了（比如说字典表，配置表），那么，我们就没有理由使用 INT 来做主键，使用 MEDIUMINT, SMALLINT 或是更小的 TINYINT 会更经济一些。如果你不需要记录时间，使用 DATE 要比 DATETIME 好得多。当然，你也需要留够足够的扩展空间。

1. **为搜索字段添加索引**

索引并不一定就是给主键或是唯一的字段。如果在你的表中，有某个字段你总要会经常用来做搜索，那么最好是为其建立索引，除非你要搜索的字段是大的文本字段，那应该建立全文索引。

1. **避免使用Select \*从数据库里读出越多的数据，那么查询就会变得越慢。**

如果你的数据库服务器和WEB服务器是两台独立的服务器的话，这还会增加网络传输的负载。即使你要查询数据表的所有字段，也尽量不要用\*通配符，善用内置提供的字段排除定义也许能给带来更多的便利。

1. **使用 ENUM 而不是 VARCHAR**

ENUM 类型是非常快和紧凑的。在实际上，其保存的是 TINYINT，但其外表上显示为字符串。这样一来，用这个字段来做一些选项列表变得相当的完美。例如，性别、民族、部门和状态之类的这些字段的取值是有限而且固定的，那么，你应该使用 ENUM 而不是 VARCHAR。

1. **尽可能的使用 NOT NULL**

除非你有一个很特别的原因去使用 NULL 值，你应该总是让你的字段保持 NOT NULL。 NULL其实需要额外的空间，并且，在你进行比较的时候，你的程序会更复杂。 当然，这里并不是说你就不能使用NULL了，现实情况是很复杂的，依然会有些情况下，你需要使用NULL值。

1. **固定长度的表会更快**

如果表中的所有字段都是“固定长度”的，整个表会被认为是 “static” 或 “fixed-length”。 例如，表中没有如下类型的字段： VARCHAR，TEXT，BLOB。只要你包括了其中一个这些字段，那么这个表就不是“固定长度静态表”了，这样，MySQL 引擎会用另一种方法来处理。固定长度的表会提高性能，因为MySQL搜寻得会更快一些，因为这些固定的长度是很容易计算下一个数据的偏移量的，所以读取的自然也会很快。而如果字段不是定长的，那么，每一次要找下一条的话，需要程序找到主键。

**24.说说对SQL语句优化有哪些方法？**

1. Where子句中：where表之间的连接必须写在其他Where条件之前，那些可以过滤掉最大数量记录的条件必须写在Where子句的末尾.HAVING最后。
2. 用EXISTS替代IN、用NOT EXISTS替代NOT IN。
3. 避免在索引列上使用计算；
4. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by ；
5. 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描；
6. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描；

**25.实践中如何优化MySQL？**

1. SQL语句及索引的优化
2. 数据库表结构的优化
3. 系统配置的优化
4. 硬件的优化

**26.如何设计一个高并发的系统？**

1. 数据库的优化，包括合理的事务隔离级别、SQL语句优化、索引的优化
2. 使用缓存，尽量减少数据库IO
3. 分布式数据库、分布式缓存
4. 服务器的负载均衡

**27.锁的优化策略？**

1. 读写分离
2. 分段加锁
3. 减少锁持有的时间
4. 多个线程尽量以相同的顺序去获取资源

**28.MySQL数据库作发布系统的存储，一天五万条以上的增量，预计运维三年,怎么优化？**

1. 设计良好的数据库结构，允许部分数据冗余，尽量避免join查询，提高效率。
2. 选择合适的表字段数据类型和存储引擎，适当的添加索引。
3. mysql库主从读写分离。
4. 找规律分表，减少单表中的数据量提高查询速度。
5. 添加缓存机制，比如memcached，apc等。
6. 不经常改动的页面，生成静态页面。
7. 书写高效率的SQL。比如不使用 SELECT \*

**29.对于大流量的网站,您采用什么样的方法来解决各页面访问量统计问题？**

1. 确认服务器是否能支撑当前访问量。
2. 优化数据库访问。
3. 禁止外部访问链接（盗链）, 比如图片盗链。
4. 控制文件下载。
5. 使用不同主机分流。
6. 使用浏览统计软件，了解访问量，有针对性的进行优化。

**31.超大分页怎么处理?**

超大的分页一般从三个方向上来解决.

* **优化SQL查询语句** 数据库层面,这也是我们主要集中关注的(虽然收效没那么大),类似于select \* from table where age > 20 limit 1000000,10这种查询其实也是有可以优化的余地的. 这条语句需要load1000000数据然后基本上全部丢弃,只取10条当然比较慢. 当时我们可以修改为select \* from table where id in (select id from table where age > 20 limit 1000000,10).这样虽然也load了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,所以速度会很快. 同时如果ID连续的好,我们还可以select \* from table where id > 1000000 limit 10,效率也是不错的,优化的可能性有许多种,但是==核心思想都一样,就是减少load的数据。==
* **优化产品设计** 从需求的角度减少这种请求….主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止ID泄漏且连续被人恶意攻击。
* **使用缓存** 解决超大分页,其实主要是靠缓存,可预测性的提前查到内容,缓存至redis等k-V数据库中,直接返回即可。

**32.关心过业务系统里面的sql耗时吗?统计过慢查询吗?对慢查询都怎么优化过?**

在业务系统中,除了使用主键进行的查询,其他的我都会在测试库上测试其耗时,慢查询的统计主要由运维在做,会定期将业务中的慢查询反馈给我们。 慢查询的优化首先要搞明白慢的原因是什么？ 是查询条件没有命中索引？ 是load了不需要的数据列？ 还是数据量太大？

所以优化也是针对这三个方向来的,

1. 首先分析语句,看看是否load了额外的数据,可能是查询了多余的行并且抛弃掉了，可能是加载了许多结果中并不需要的列，对语句进行分析以及重写。
2. 分析语句的执行计划,然后获得其使用索引的情况,之后修改语句或者修改索引，使得语句可以尽可能的命中索引。
3. 如果对语句的优化已经无法进行，可以考虑表中的数据量是否太大，如果是的话可以进行横向或者纵向的分表。
4. 推荐使用自增ID,不要使用UUID.

**33.Modelfirst和DBfirst区别**

**Model First** 就是代表model优先，那么前提也就是先创建model，然后根据model自动建立数据库。 **Datebase First** 就是代表数据库优先，那么前提就是先创建数据库。

**区别** 当数据结构发生变化的时候， **Datebase First**选择的是从数据库生成。 **Model First**选择的是从模型生成数据库。

**原因** 在数据结构发生变化的时候，**Datebase First**编程方式中是选择从数据库更新模型，因此就导致了**Datebase First**是以数据库为主；而**Model First**选择的是空模型生成。

**Redis模块**

**01. 什么是Redis？**

Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据flush到硬盘上进行保存。因为是纯内存操作，Redis的性能非常出色，每秒可以处理超过 10万次读写操作，是已知性能最快的Key-Value DB。 Redis的出色之处不仅仅是性能，Redis最大的魅力是支持保存多种数据结构，此外单个value的最大限制是1GB，不像 memcached只能保存1MB的数据，因此Redis可以用来实现很多有用的功能，比方说用他的List来做FIFO双向链表，实现一个轻量级的高性 能消息队列服务，用他的Set可以做高性能的tag系统等等。另外Redis也可以对存入的Key-Value设置expire时间，因此也可以被当作一 个功能加强版的memcached来用。 Redis的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

**02. Redis相比memcached有哪些优势？**

1. memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型
2. redis的速度比memcached快很多
3. redis可以持久化其数据

**03. Redis支持哪几种数据类型？** String、List、Set、Sorted Set、hashes

**04. Redis主要消耗什么物理资源？**redis是一种基于内存的高性能数据库--- 主要消耗于内存

**05. Redis的全称是什么？**Remote Dictionary Server。

**06. Redis有哪几种数据淘汰策略？**

* noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但DEL和几个例外）
* allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。
* volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。
* allkeys-random: 回收随机的键使得新添加的数据有空间存放。
* volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。
* volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

**07. Redis官方为什么不提供Windows版本？**

因为目前Linux版本已经相当稳定，而且用户量很大，无需开发windows版本，反而会带来兼容性等问题。

**08. 一个字符串类型的值能存储最大容量是多少？**512M

**09. 为什么Redis需要把所有数据放到内存中？**

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。在内存越来越便宜的今天，redis将会越来越受欢迎。 如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

**10. Redis集群方案应该怎么做？都有哪些方案？**

1. twemproxy，大概概念是，它类似于一个代理方式，使用方法和普通redis无任何区别，设置好它下属的多个redis实例后，使用时在本需要连接redis的地方改为连接twemproxy，它会以一个代理的身份接收请求并使用一致性hash算法，将请求转接到具体redis，将结果再返回twemproxy。使用方式简便(相对redis只需修改连接端口)，对旧项目扩展的首选。 问题：twemproxy自身单端口实例的压力，使用一致性hash后，对redis节点数量改变时候的计算值的改变，数据无法自动移动到新的节点。
2. codis，目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在节点数量改变情况下，旧节点数据可恢复到新hash节点。
3. redis cluster3.0自带的集群，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。
4. 在业务代码层实现，起几个毫无关联的redis实例，在代码层，对key 进行hash计算，然后去对应的redis实例操作数据。 这种方式对hash层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的自动脚本恢复，实例的监控，等等。

**11. Redis集群方案什么情况下会导致整个集群不可用？**

有A，B，C三个节点的集群,在没有复制模型的情况下,如果节点B失败了，那么整个集群就会以为缺少5501-11000这个范围的槽而不可用。

**12. MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据？**

redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

**13. Redis有哪些适合的场景？**

1. 会话缓存（Session Cache）

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？ 幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

1. 全页缓存（FPC）

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。 再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。 此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

1. 队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。

如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从这里去查看。

1. 排行榜/计数器

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可： 当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行： ZRANGE user\_scores 0 10 WITHSCORES Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

1. 发布/订阅

最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

**14.Redis为什么快？除了他是内存型数据库外，还有什么原因?**

* [1、采用了多路复用io阻塞机制](https://blog.csdn.net/qq_44472134/article/details/104252693#31io_25)
* [2、数据结构简单，操作节省时间](https://blog.csdn.net/qq_44472134/article/details/104252693#32_26)
* [3、是C语言编写的，运行在内存中，自然速度快](https://blog.csdn.net/qq_44472134/article/details/104252693#33C_27)
* [采用单线程，](https://blog.csdn.net/qq_44472134/article/details/104252693#_28)
* [使用底层模型不同](https://blog.csdn.net/qq_44472134/article/details/104252693#_30)

**15. Redis和Redisson有什么关系？**

Redisson是一个高级的分布式协调Redis客服端，能帮助用户在分布式环境中轻松实现一些Java的对象 (Bloom filter, BitSet, Set, SetMultimap, ScoredSortedSet, SortedSet, Map, ConcurrentMap, List, ListMultimap, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock, ReadWriteLock, AtomicLong, CountDownLatch, Publish / Subscribe, HyperLogLog)。

**16. Redis与Redisson对比有什么优缺点？**

Jedis是Redis的Java实现的客户端，其API提供了比较全面的Redis命令的支持；Redisson实现了分布式和可扩展的Java数据结构，和Jedis相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等Redis特性。Redisson的宗旨是促进使用者对Redis的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

**17. Redis如何设置密码及验证密码？**

设置密码：config set requirepass 123456

授权密码：auth 123456

**18. 说说Redis哈希槽的概念？**

Redis集群没有使用一致性hash,而是引入了哈希槽的概念，Redis集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

**19. Redis集群的主从复制模型是怎样的？**

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有N-1个复制品.

**20. Redis集群会有写操作丢失吗？为什么？**Redis并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

**21. Redis集群之间是如何复制的？**异步复制

**22. Redis集群最大节点个数是多少？**16384个。

**23. Redis集群如何选择数据库？**Redis集群目前无法做数据库选择，默认在0数据库。

**24. 怎么测试Redis的连通性？**ping

**25. Redis中的管道有什么用？**

一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应。这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。

这就是管道（pipelining），是一种几十年来广泛使用的技术。例如许多POP3协议已经实现支持这个功能，大大加快了从服务器下载新邮件的过程。

**26. 怎么理解Redis事务？**

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

**27. Redis事务相关的命令有哪几个？**MULTI、EXEC、DISCARD、WATCH

**28. Redis key的过期时间和永久有效分别怎么设置？**EXPIRE和PERSIST命令。

**29. Redis如何做内存优化？**

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key,而是应该把这个用户的所有信息存储到一张散列表里面.

**30. Redis回收进程如何工作的？**

一个客户端运行了新的命令，添加了新的数据。

Redi检查内存使用情况，如果大于maxmemory的限制, 则根据设定好的策略进行回收。

一个新的命令被执行，等等。

所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地回收回到边界以下。

如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越。

**31. Redis回收使用的是什么算法？** LRU算法

**32. Redis如何做大量数据插入？**Redis2.6开始redis-cli支持一种新的被称之为pipe mode的新模式用于执行大量数据插入工作。

**33. 为什么要做Redis分区？**

分区可以让Redis管理更大的内存，Redis将可以使用所有机器的内存。如果没有分区，你最多只能使用一台机器的内存。分区使Redis的计算能力通过简单地增加计算机得到成倍提升,Redis的网络带宽也会随着计算机和网卡的增加而成倍增长。

**34. 你知道有哪些Redis分区实现方案？**

客户端分区就是在客户端就已经决定数据会被存储到哪个redis节点或者从哪个redis节点读取。大多数客户端已经实现了客户端分区。

代理分区：意味着客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。代理根据分区规则决定请求哪些Redis实例，然后根据Redis的响应结果返回给客户端。redis和memcached的一种代理实现就是Twemproxy。

查询路由(Query routing) 的意思是客户端随机地请求任意一个redis实例，然后由Redis将请求转发给正确的Redis节点。

Redis Cluster实现了一种混合形式的查询路由，但并不是直接将请求从一个redis节点转发到另一个redis节点，而是在客户端的帮助下直接redirected到正确的redis节点。

**35. Redis分区有什么缺点？**

涉及多个key的操作通常不会被支持。例如你不能对两个集合求交集，因为他们可能被存储到不同的Redis实例（实际上这种情况也有办法，但是不能直接使用交集指令）。

同时操作多个key,则不能使用Redis事务.

分区使用的粒度是key，不能使用一个非常长的排序key存储一个数据集（The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set）.

当使用分区的时候，数据处理会非常复杂，例如为了备份你必须从不同的Redis实例和主机同时收集RDB / AOF文件。

分区时动态扩容或缩容可能非常复杂。Redis集群在运行时增加或者删除Redis节点，能做到最大程度对用户透明地数据再平衡，但其他一些客户端分区或者代理分区方法则不支持这种特性。然而，有一种预分片的技术也可以较好的解决这个问题。

**36. Redis持久化数据和缓存怎么做扩容？**

如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容。

如果Redis被当做一个持久化存储使用，必须使用固定的keys-to-nodes映射关系，节点的数量一旦确定不能变化。否则的话(即Redis节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。

**37. 分布式Redis是前期做还是后期规模上来了再做好？为什么？**

既然Redis是如此的轻量（单实例只使用1M内存）,为防止以后的扩容，最好的办法就是一开始就启动较多实例。即便你只有一台服务器，你也可以一开始就让Redis以分布式的方式运行，使用分区，在同一台服务器上启动多个实例。

一开始就多设置几个Redis实例，例如32或者64个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看做这点牺牲是值得的。

这样的话，当你的数据不断增长，需要更多的Redis服务器时，你需要做的就是仅仅将Redis实例从一台服务迁移到另外一台服务器而已（而不用考虑重新分区的问题）。一旦你添加了另一台服务器，你需要将你一半的Redis实例从第一台机器迁移到第二台机器。

**38. Twemproxy是什么？**

Twemproxy是Twitter维护的（缓存）代理系统，代理Memcached的ASCII协议和Redis协议。它是单线程程序，使用c语言编写，运行起来非常快。它是采用Apache 2.0 license的开源软件。

Twemproxy支持自动分区，如果其代理的其中一个Redis节点不可用时，会自动将该节点排除（这将改变原来的keys-instances的映射关系，所以你应该仅在把Redis当缓存时使用Twemproxy)。

Twemproxy本身不存在单点问题，因为你可以启动多个Twemproxy实例，然后让你的客户端去连接任意一个Twemproxy实例。

Twemproxy是Redis客户端和服务器端的一个中间层，由它来处理分区功能应该不算复杂，并且应该算比较可靠的。

**39. 支持一致性哈希的客户端有哪些？**Redis-rb、Predis等。

**40. Redis与其他key-value存储有什么不同？**

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，应为数据量不能大于硬件内存。在内存数据库方面的另一个优点是， 相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。 同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

**41. Redis的内存占用情况怎么样？**

给你举个例子： 100万个键值对（键是0到999999值是字符串“hello world”）在我的32位的Mac笔记本上 用了100MB。同样的数据放到一个key里只需要16MB， 这是因为键值有一个很大的开销。 在Memcached上执行也是类似的结果，但是相对Redis的开销要小一点点，因为Redis会记录类型信息引用计数等等。

当然，大键值对时两者的比例要好很多。

64位的系统比32位的需要更多的内存开销，尤其是键值对都较小时，这是因为64位的系统里指针占用了8个字节。 但是，当然，64位系统支持更大的内存，所以为了运行大型的Redis服务器或多或少的需要使用64位的系统。

**42. 都有哪些办法可以降低Redis的内存使用情况呢？**

如果你使用的是32位的Redis实例，可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。

**43. 查看Redis使用情况及状态信息用什么命令？** Info命令

**44. Redis的内存用完了会发生什么？**

如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以将Redis当缓存来使用配置淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

**45. Redis是单线程的，如何提高多核CPU的利用率？**

可以在同一个服务器部署多个Redis的实例，并把他们当作不同的服务器来使用，在某些时候，无论如何一个服务器是不够的， 所以，如果你想使用多个CPU，你可以考虑一下分片（shard）。

**46. 一个Redis实例最多能存放多少的keys？List、Set、Sorted Set他们最多能存放多少元素？**

理论上Redis可以处理多达232的keys，并且在实际中进行了测试，每个实例至少存放了2亿5千万的keys。我们正在测试一些较大的值。

任何list、set、和sorted set都可以放232个元素。换句话说，Redis的存储极限是系统中的可用内存值。

**47. Redis常见性能问题和解决方案？**

1. Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件
2. 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次
3. 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内
4. 尽量避免在压力很大的主库上增加从库
5. 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

**48. Redis提供了哪几种持久化方式？**

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储.

AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF命令以redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大.

如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.

你也可以同时开启两种持久化方式, 在这种情况下, 当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整.

最重要的事情是了解RDB和AOF持久化方式的不同,让我们以RDB持久化方式开始。

**49. 如何选择合适的持久化方式？**

一般来说， 如果想达到足以媲美PostgreSQL的数据安全性， 你应该同时使用两种持久化功能。如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失，那么你可以只使用RDB持久化。

有很多用户都只使用AOF持久化，但并不推荐这种方式：因为定时生成RDB快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比AOF恢复的速度要快，除此之外， 使用RDB还可以避免之前提到的AOF程序的bug。

**50. 修改配置不重启Redis会实时生效吗？**

针对运行实例，有许多配置选项可以通过 CONFIG SET 命令进行修改，而无需执行任何形式的重启。 从 Redis 2.2 开始，可以从 AOF 切换到 RDB 的快照持久性或其他方式而不需要重启 Redis。检索 ‘CONFIG GET \* 命令获取更多信息。但偶尔重新启动是必须的，如为升级 Redis 程序到新的版本，或者当你需要修改某些目前 CONFIG 命令还不支持的配置参数的时候。

**一、为什么要使用redis呢？**

原因一、避免反复请求数据库造成效率低下

举个例子，假如系统中有2千万用户信息，用户信息基本固定，一旦录入很少变动，那么你每次加载所有用户信息时，如果都要请求数据库，数据库编译并执行你的查询语句，这样效率就会低下很多，针对这种信息不经常变动并且数据量。

较大的情况，通常做法，就是把他加入缓存，每次取数前先去判断，如果缓存不为空，那么就从缓存取值，如果为空，再去请求数据库，并将数据加入缓存，这样大大提高系统访问效率。

原因二、需要在 秒杀，热点数据 和DB数据库之间 加入缓存中间件

项目的DB遇到了瓶颈，特别是秒杀和热点数据这样的场景DB基本上就扛不住了，那就需要缓存中间件的加入了，目前市面上有的缓存中间件有 Redis 和 Memcached ，他们的优缺点

https://blog.csdn.net/qq\_44472134/article/details/104416741

综合这些然后再结合我们项目特点，最后我们在技术选型的时候选谁。

**二、以下单线程仅指Redis负责存取这块的线程只有一个，而非Redis中只有一个进程**

我先给个我的结论，单线程的Redis在瓶颈是cpu的io时（这不是大多数应用的实际应用场景），确实速度会比多线程慢。但是，我们实际应用场景中很少会遇到瓶颈是CPU的io的情况，这时候单线程优势就凸显出来了。

实现很简单！性能又不会比多线程差，并且，单线程确实不用处理上下文的切换，cpu利用率会比多线程高，这时候采用单线程实现是一种很划算的做法。当然，如果你的宽带和内存牛逼到了使得你的io成为瓶颈，这时候也只能使用多线程了。

面试时考官让我挑一种自己熟悉的NoSQL数据库讲一讲，我当场就蒙了，我就用过sql server，mysql和Oracle这几种，这几种就算从名字看也知道是sql数据库嘛，绞尽脑汁，我福至心灵，答出，Redis！

先说说Redis是什么吧小老弟？

Redis嘛，就是一种运行速度很快，并发很强的跑在内存上的NoSql数据库，支持键到五种数据类型的映射，（string、list、set、zset、hash）

**三、讲一讲为什么Redis这么快？**

3.1、采用了多路复用io阻塞机制

3.2、数据结构简单，操作节省时间

3.3.、是C语言编写的，运行在内存中，自然速度快

采用单线程，

避免了不必要的上下文切换和竞争条件，安全；也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

使用底层模型不同

它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求

**四、Redis的常见问题**

4.1、Redis为什么是单线程的？

因为Redis的瓶颈不是cpu的运行速度，而往往是网络带宽和机器的内存大小。再说了，单线程切换开销小，容易实现既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了。

4.2、如果万一CPU成为你的Redis瓶颈了，或者，你就是不想让服务器其他核闲置，那怎么办？

那也很简单，你多起几个Redis进程就好了。Redis是key-value数据库，又不是关系数据库，数据之间没有约束。只要客户端分清哪些key放在哪个Redis进程上就可以了。redis-cluster可以帮你做的更好。

4.3、单线程可以处理高并发请求吗？

当然可以了，Redis都实现了。有一点概念需要澄清，并发并不是并行。

（相关概念：并发性I/O流，意味着能够让一个计算单元来处理来自多个客户端的流请求。并行性，意味着服务器能够同时执行几个事情，具有多个计算单元）

4.4、我们使用单线程的方式是无法发挥多核CPU 性能，有什么办法发挥多核CPU的性能嘛？

我们可以通过在单机开多个Redis 实例来完善！

警告：这里我们一直在强调的单线程，只是在处理我们的网络请求的时候只有一个线程来处理，一个正式的Redis Server运行的时候肯定是不止一个线程的，这里需要大家明确的注意一下！

例如Redis进行持久化的时候会以子进程或者子线程的方式执行（具体是子线程还是子进程待读者深入研究）

**六、有序集合的实现方式是哪种数据结构？**跳跃表。

**七、请列举几个用得到Redis的常用使用场景?**

7.1、缓存，

毫无疑问这是Redis当今最为人熟知的使用场景。再提升服务器性能方面非常有效；

7.2、排行榜，

在使用传统的关系型数据库（mysql oracle 等）来做这个事儿，非常的麻烦，而利用Redis的SortSet(有序集合)数据结构能够简单的搞定；

7.3、计算器/限速器，

利用Redis中原子性的自增操作，我们可以统计类似用户点赞数、用户访问数等，这类操作如果用MySQL，频繁的读写会带来相当大的压力；限速器比较典型的使用场景是限制某个用户访问某个API的频率，常用的有抢购时，防止用户疯狂点击带来不必要的压力；

7.4、好友关系，

利用集合的一些命令，比如求交集、并集、差集等。可以方便搞定一些共同好友、共同爱好之类的功能；

7.5、简单消息队列，

除了Redis自身的发布/订阅模式，我们也可以利用List来实现一个队列机制，比如：到货通知、邮件发送之类的需求，不需要高可靠，但是会带来非常大的DB压力，完全可以用List来完成异步解耦；

7.6、Session共享，

以PHP为例，默认Session是保存在服务器的文件中，如果是集群服务，同一个用户过来可能落在不同机器上，这就会导致用户频繁登陆；采用Redis保存Session后，无论用户落在那台机器上都能够获取到对应的Session信息。

7.7、一些频繁被访问的数据，

经常被访问的数据如果放在关系型数据库，每次查询的开销都会很大，而放在redis中，因为redis 是放在内存中的可以很高效的访问

**八、简述Redis的数据淘汰机制？**

volatile-lru 从已设置过期时间的数据集中挑选最近最少使用的数据淘汰

volatile-ttl 从已设置过期时间的数据集中挑选将要过期的数据淘汰

volatile-random从已设置过期时间的数据集中任意选择数据淘汰

allkeys-lru从所有数据集中挑选最近最少使用的数据淘汰

allkeys-random从所有数据集中任意选择数据进行淘汰

noeviction禁止驱逐数据

**九、Redis怎样防止异常数据不丢失？**

9.1、RDB 持久化 （快照的形式，周期性的持久化）\*\* 是Redis的默认持久化方法 适合冷备

将某个时间点的所有数据都存放到硬盘上。

可以将快照复制到其它服务器从而创建具有相同数据的服务器副本。

如果系统发生故障，将会丢失最后一次创建快照之后的数据。

如果数据量很大，保存快照的时间会很长。

9.1.1、RDB的配置方式

找到Redis的配置文件：redis.conf

1） 设置触发条件：略

2) 设置rdb文件路径：略

默认rdb文件存放路径是当前目录，文件名是：dump.rdb。可以在配置文件中修改路径和文件名，分别是dir和dbfilename

Redis启动后会读取RDB快照文件，将数据从硬盘载入到内存，一般情况下1GB的快照文件载入到内存的时间大约20-30秒钟。

除了自动快照，还可以手动发送SAVE或BGSAVE命令让Redis执行快照，两个命令的区别在于，前者是由主进程进行快照操作，会阻塞住其他请求，后者会通过fork子进程进行快照操作。

通过RDB方式实现持久化，一旦Redis异常退出，就会丢失最后一次快照以后更改的所有数据。这就需要开发者根据具体的应用场合，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受的范围。如果数据很重要以至于无法承受任何损失，则可以考虑使用AOF方式进行持久化

9.1.2、RDB 優點

1.RDB是一个非常紧凑(compact)的文件，它保存了redis 在某个时间点上的数据集。这种文件非常适合用于进行备份和灾难恢复（将持久化到硬盘中的文件恢复即可）。

2.生成RDB文件的时候，redis主进程会fork()一个子进程来处理所有保存工作，主进程不需要进行任何磁盘IO操作。

3.RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

9.1.3、RDB缺點

1.如果你需要尽量避免在服务器故障时丢失数据，那么RDB 不适合你。 虽然Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据（最后一次的数据）。

2.每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

9.1.4、RDB 保存過程

当条件满足，redis需要执行RDB的时候，服务器会执行以下操作：

1、 redis调用系统函数fork() ，创建一个子进程进行持久化。

2.子进程将数据集写入到一个临时 RDB 文件中（持久化，也就是将内存中的数据写入临时文件）。

3.当子进程完成对临时RDB文件的写入时，redis 用新的临时RDB 文件替换原来的RDB 文件，并删除旧 RDB 文件。

注：fork的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程。

在执行fork的时候操作系统（类Unix操作系统）会使用写时复制（copy-on-write）策略，即fork函数发生的一刻父子进程共享同一内存数据，当父进程要更改其中某片数据时（如执行一个写命令 ），操作系统会将该片数据复制一份以保证子进程的数据不受影响，所以新的RDB文件存储的是执行fork那一刻的内存数据。

Redis在进行快照的过程中不会修改RDB文件，只有快照结束后才会将旧的文件替换成新的，也就是说任何时候RDB文件都是完整的。这使得我们可以通过定时备份RDB文件来实 现Redis数据库备份。RDB文件是经过压缩（可以配置rdbcompression参数以禁用压缩节省CPU占用）的二进制格式，所以占用的空间会小于内存中的数据大小，更加利于传输。

9.2、AOF 持久化 （追加到AOF文件的末尾，最后以不同的频次保存到到磁盘）适合热备

将写命令添加到 AOF 文件（Append Only File）的末尾。

使用 AOF 持久化需要设置同步选项，从而确保写命令同步到磁盘文件上的时机。这是因为对文件进行写入并不会马上将内容同步到磁盘上，而是先存储到缓冲区，然后由操作系统决定什么时候同步到磁盘。有以下同步选项：

选项同步频率always每个写命令都同步everysec每秒同步一次no让操作系统来决定何时同步

always 选项会严重减低服务器的性能；

everysec 选项比较合适，可以保证系统崩溃时只会丢失一秒左右的数据，并且 Redis 每秒执行一次同步对服务器性能几乎没有任何影响；

no 选项并不能给服务器性能带来多大的提升，而且也会增加系统崩溃时数据丢失的数量

随着服务器写请求的增多，AOF 文件会越来越大。Redis 提供了一种将 AOF 重写的特性，能够去除 AOF 文件中的冗余写命令。

用户可以向 Redis 发送 BGREWRITEAOF 命令，这个命令会移除 AOF 文件中冗余的命令来重写 AOF 文件，使 AOF 文件的体积变得尽可能地小

BGREWRITEAOF 的工作原理和快照持久化命令 BGSAVE 的工作原理类似，Redis 会创建一个子进程来负责对 AOF 文件进行重写。

值得注意的是，进行 AOF 文件重写时，如果原来的 AOF 文件体积已经非常大，那么重写 AOF 并删除旧 AOF 文件的过程将会对 Redis 的性能造成较大的影响。

为此，Redis 提供auto-aof-rewrite-percentage和auto-aof-rewrite-min-size两个配置选项来对 AOF 重写进行配置。auto-aof-rewrite-percentage和auto-aof-rewrite-min-size两个配置选项的含义可以参考 redis.conf 配置中的详细说明，具体来说，auto-aof-rewrite-percentage配置当 AOF 文件需要比旧 AOF 文件增大多少时才进行 AOF 重写，而auto-aof-rewrite-min-size则配置当 AOF 文件需要达到多大体积时才进行 AOF 重写。只有这两个配置的条件都达到时，才会进行 AOF 重写。

9.2.1、AOF的设置方法：

为了打开 AOF 持久化的功能，我们只需要将 redis.conf 配置文件中的appendonly配置选项设置为yes即可。涉及 AOF 持久化的几个常用配置如下所示：

appendonly yes

appendfilename "appendonly.aof"

appendfsync everysec

appendonly：是否打开 AOF 持久化功能

appendfilename：AOF 文件名称

appendfsync：同步频率

appendfsync 选项及同步频率

选项 同步频率

always 每个 Redis 命令都要同步写入硬盘。这样会严重降低 Redis 的性能

everysec 每秒执行一次同步，显式地将多个写命令同步到硬盘

no 让操作系统来决定应该何时进行同步

9.2.2、AOF的优点

AOF 持久化的方法提供了多种的同步频率，即使使用默认的同步频率每秒同步一次，Redis 最多也就丢失 1 秒的数据而已。

AOF 文件使用 Redis 命令追加的形式来构造，因此，即使 Redis 只能向 AOF 文件写入命令的片断，使用 redis-check-aof 工具也很容易修正 AOF 文件。

AOF 文件的格式可读性较强，这也为使用者提供了更灵活的处理方式。例如，如果我们不小心错用了 FLUSHALL 命令，在重写还没进行时，我们可以手工将最后的 FLUSHALL 命令去掉，然后再使用 AOF 来恢复数据。

9.2.3、AOF 的缺点

对于具有相同数据的的 Redis，AOF 文件通常会比 RDF 文件体积更大。

虽然 AOF 提供了多种同步的频率，默认情况下，每秒同步一次的频率也具有较高的性能。但在 Redis 的负载较高时，RDB 比 AOF 具好更好的性能保证。

RDB 使用快照的形式来持久化整个 Redis 数据，而 AOF 只是将每次执行的命令追加到 AOF 文件中，因此从理论上说，RDB 比 AOF 方式更健壮。官方文档也指出，AOF 的确也存在一些 BUG，这些 BUG 在 RDB 没有存在。

**十、缓存常见问题及缓存穿透，缓存雪崩以及缓存击穿 的含义及原理？**

10.1、缓存更新方式

这是决定在使用缓存时就该考虑的问题。

缓存的数据在数据源发生变更时需要对缓存进行更新，数据源可能是 DB，也可能是远程服务。更新的方式可以是主动更新。数据源是 DB 时，可以在更新完 DB 后就直接更新缓存。

当数据源不是 DB 而是其他远程服务，可能无法及时主动感知数据变更，这种情况下一般会选择对缓存数据设置失效期，也就是数据不一致的最大容忍时间。

这种场景下，可以选择失效更新，key 不存在或失效时先请求数据源获取最新数据，然后再次缓存，并更新失效期。

但这样做有个问题，如果依赖的远程服务在更新时出现异常，则会导致数据不可用。改进的办法是异步更新，就是当失效时先不清除数据，继续使用旧的数据，然后由异步线程去执行更新任务。这样就避免了失效瞬间的空窗期。另外还有一种纯异步更新方式，定时对数据进行分批更新。实际使用时可以根据业务场景选择更新方式。

10.2、数据不一致

产生的原因

一般是主动更新失败，例如更新 DB 后，更新 Redis 因为网络原因请求超时；或者是异步更新失败导致。

解决的办法

如果服务对耗时不是特别敏感可以增加重试；如果服务对耗时敏感可以通过异步补偿任务来处理失败的更新，或者短期的数据不一致不会影响业务，那么只要下次更新时可以成功，能保证最终一致性就可以。

10.3、缓存穿透：redis和数据库中都没有数据

打个比方，你是个很有钱的人，开满了百度云，腾讯视频各种杂七杂八的会员，但是你就是没有netflix的会员，然后你把这些账号和密码发布到一个你自己做的网站上，然后你有一个朋友每过十秒钟就查询你的网站，发现你的网站没有Netflix的会员后打电话向你要。你就相当于是个数据库，网站就是Redis。这就是缓存穿透。

10.4、缓存击穿：就是一个很热门的数据，突然失效，大量请求到服务器数据库中

大家都喜欢看腾讯视频上的《水果传》，但是你的会员突然到期了，大家在你的网站上看不到腾讯视频的账号，纷纷打电话向你询问，这就是缓存击穿

10.5、缓存雪崩：大面积的缓存失效，打崩了DB

你的各种会员突然同一时间都失效了，那这就是缓存雪崩了。

10.6、 解决方法

缓存穿透：数据库和缓存中都没有

1.接口层增加校验，对传参进行个校验，比如说我们的id是从1开始的，那么id<=0的直接拦截；

2.缓存中取不到的数据，在数据库中也没有取到，这时可以将key-value对写为key-null，这样可以防止攻击用户反复用同一个id暴力攻击,等再訪問這個key的時候，就提示位置错误、稍后重试这样的值具体取啥问产品，或者將缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。

3.这样可以防止攻击用户反复用同一个id暴力攻击，但是我们要知道正常用户是不会在单秒内发起这么多次请求的，可以让运维大大对单个IP每秒访问次数超出阈值的IP都拉黑。

4、还有一个高级用法布隆过滤器（Bloom Filter）这个也能很好的防止缓存穿透的发生，他的原理也很简单就是利用高效的数据结构和算法快速判断出你这个Key是否在数据库中存在，不存在你return就好了，存在你就去查了DB刷新KV再return

缓存击穿： 单个key失效

1、最好的办法就是设置热点数据永不过期，拿到刚才的比方里，那就是你买腾讯一个永久会员

缓存雪崩： 所有缓存同时失效

1.缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。

2.如果缓存数据库是分布式部署，将热点数据均匀分布在不同得缓存数据库中。

3.redis数据库常用数据类型已经相关操作

**十一、过期时间设置常用操作：**

11.1、过期时间设置

EXPIRE 将key的生存时间设置为ttl秒

PEXPIRE 将key的生成时间设置为ttl毫秒

EXPIREAT 将key的过期时间设置为timestamp所代表的的秒数的时间戳

PEXPIREAT 将key的过期时间设置为timestamp所代表的的毫秒数的时间戳

其实以上几种处理方式都是根据PEXPIREAT来实现的，设置生存时间的时候是redis内部计算好时间之后在内存处理的，最终的处理都会转向PEXPIREAT。

1、2两种方式是设置一个过期的时间段，就是咱们处理验证码最常用的策略，设置三分钟或五分钟后失效，把分钟数转换成秒或毫秒存储到redis中。

3、4两种方式是指定一个过期的时间 ，比如优惠券的过期时间是某年某月某日，只是单位不一样

11.2、键过期时间查看

ttl：以秒为单位，返回键的剩余生存时间。

pttl：以毫秒为单位，返回键的剩余生存时间。

11.3、键过期判定

检查当前Unix（当前时间）时间戳是否大于键的过期时间，是则过期，否则不过期。

11.4、 过期键的理论删除策略

（1）定时删除：

　　设置一个键过期时间的同时，创建一个定时器。每个带有过期时间的键都对应着一个定时器。

　　这种策略对内存是最友好的，但对CPU时间是最不友好的。创建一个定时器需要用到Redis服务器中的时间事件，而当前时间事件的实现方式为无序链表，查找一个事件的时间复杂度为O(N)，并不能高效地处理大量时间事件。

（2）惰性删除：

　　访问一个键的时候再检测该键是否过期，是则删除之。

　　这种策略对CPU时间是最友好的，但对内存是最不友好的。没被访问到的过期键永远不会被删除，可以看做内存泄露。对于运行状态非常依赖于内存的Redis来说，这种策略显然会影响到Redis的性能。

（3）定期删除：

　　这种策略是对前两种策略的整合与折中方案。使用这种策略需要控制好删除操作每次执行的时长和执行的频率，否则会退化为前两种策略的其中一种。

（4）内存淘汰机制

11.5、 Redis采用的过期键删除策略

Redis服务器实际使用的是惰性删除和定期删除两种策略配合使用的方案。

（1）惰性删除策略的实现：

　　所有读写数据库的Redis命令在执行之前都会先检查输入键是否已过期，过期则删除之。

（2）定期删除策略的实现：

　　在规定时间内，分多次遍历服务器中的各个数据库，从数据库的过期字典中随机检查一部分键的过期时间，并删除其中的过期键。

<1>定期删除程序每次运行时，都会从一定数量的数据库中取出一定数量的随机键进行检查，并删除其中的过期键。

<2>使用一个全局变量记录当前删除程序检查的是第几个数据库，下一次运行都会接着上一次的进度进行处理。

<3>随着删除程序的不断执行，服务器中所有的数据库都会被检查一遍，然后这个全局变量被重置为0，开始新一轮的检查工作。

11.6、 AOF、RDB和复制功能对过期键的处理

（1）生成RDB文件：

在执行save或bgsave命令创建一个新的RDB文件时，程序会对数据库中的键进行检查，已过期的键不会被保存到新创建的RDB文件中。

（2）载入RDB文件：

<1>主服务器模式：载入RDB文件时，程序会对文件中保存的键进行检查，只有未过期的键会被载入到数据库中。

<2>从服务器模式：文件中保存的所有键都会被载入到数据库中。不过因为主从服务器在进行数据同步的时候，从服务器的数据库会被清空，所以过期键对载入RDB文件的从服务器也不会造成影响。

（3）AOF文件写入：

当过期键被惰性删除或定期删除之后，程序会向AOF文件追加一条del命令，来显式地记录该键已被删除。

（4）AOF重写：

程序会对数据库中的键进行检查，已过期的键不会被保存到重写后的AOF文件中。

（5）复制：

当服务器运行在复制模式下时，从服务器的过期键删除动作由主服务器控制：

<1>主服务器在删除一个过期键之后，会显式地向所有从服务器发送一个del命令，告知从服务器删除这个过期键。

<2>从服务器在执行客户端发送的读命令时，即使碰到过期键也不会将其删除，而是将过期键的值继续返回给客户端。

<3>从服务器只有在接到主服务器发送来的del命令之后，才会删除过期键。

说一下Redis中的Master-Slave模式

**十二、连接过程？**

主服务器创建快照文件，发送给从服务器，并在发送期间使用缓冲区记录执行的写命令。快照文件发送完毕之后，开始向从服务器发送存储在缓冲区中的写命令；

从服务器丢弃所有旧数据，载入主服务器发来的快照文件，之后从服务器开始接受主服务器发来的写命令；

主服务器每执行一次写命令，就向从服务器发送相同的写命令。

**十三、主从链(图略)？**

随着负载不断上升，主服务器可能无法很快地更新所有从服务器，或者重新连接和重新同步从服务器将导致系统超载。为了解决这个问题，可以创建一个中间层来分担主服务器的复制工作。中间层的服务器是最上层服务器的从服务器，又是最下层服务器的主服务器。

Sentinel（哨兵）可以监听集群中的服务器，并在主服务器进入下线状态时，自动从从服务器中选举出新的主服务器。

**十四、分片？**

分片是将数据划分为多个部分的方法，可以将数据存储到多台机器里面，这种方法在解决某些问题时可以获得线性级别的性能提升。

假设有 4 个 Redis 实例 R0，R1，R2，R3，还有很多表示用户的键 user:1，user:2，... ，有不同的方式来选择一个指定的键存储在哪个实例中。

最简单的方式是范围分片，例如用户 id 从 0~1000 的存储到实例 R0 中，用户 id 从 1001~2000 的存储到实例 R1 中，等等。但是这样需要维护一张映射范围表，维护操作代价很高。

还有一种方式是哈希分片，使用 CRC32 哈希函数将键转换为一个数字，再对实例数量求模就能知道应该存储的实例。

根据执行分片的位置，可以分为三种分片方式：

客户端分片：客户端使用一致性哈希等算法决定键应当分布到哪个节点。

代理分片：将客户端请求发送到代理上，由代理转发请求到正确的节点上。

服务器分片：Redis Cluster

**十五、Redis是怎么持久化的？服务主从数据怎么交互的？**

RDB做镜像全量持久化，AOF做增量持久化。因为RDB会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要AOF来配合使用。在redis实例重启时，会使用RDB持久化文件重新构建内存，再使用AOF重放近期的操作指令来实现完整恢复重启之前的状态。

这里很好理解，把RDB理解为一整个表全量的数据，AOF理解为每次操作的日志就好了，服务器重启的时候先把表的数据全部搞进去，但是他可能不完整，你再回放一下日志，数据不就完整了嘛。不过Redis本身的机制是 AOF持久化开启且存在AOF文件时，优先加载AOF文件；AOF关闭或者AOF文件不存在时，加载RDB文件；加载AOF/RDB文件城后，Redis启动成功； AOF/RDB文件存在错误时，Redis启动失败并打印错误信息

[**redis数据库常见的面试问题**](https://www.cnblogs.com/Survivalist/p/8119891.html)

**020、Redis是什么？**

1、是一个完全开源免费的key-value内存数据库；

2、通常被认为是一个数据结构服务器，主要是因为其有着丰富的数据结构 strings、map、 list、sets、 sorted sets；

​通常局限点来说，Redis也以消息队列的形式存在，作为内嵌的List存在，满足实时的高并发需求。在使用缓存的时候，redis比memcached具有更多的优势，并且支持更多的数据类型，把redis当作一个中间存储系统，用来处理高并发的数据库操作；

**Redis优点：**

1、速度快：使用标准C写，所有数据都在内存中完成，读写速度分别达到10万/20万；

2、持久化：对数据的更新采用Copy-on-write技术，可以异步地保存到磁盘上，主要有两种策略，一是根据时间，更新次数的快照（save 300 10 ）二是基于语句追加方式(Append-only file，aof)；

3、自动操作：对不同数据类型的操作都是自动的，很安全；

4、快速的主--从复制，官方提供了一个数据，Slave在21秒即完成了对Amazon网站10G key set的复制；

5、Sharding技术： 很容易将数据分布到多个Redis实例中，数据库的扩展是个永恒的话题，在关系型数据库中，主要是以添加硬件、以分区为主要技术形式的纵向扩展解决了很多的应用场景，但随着web2.0、移动互联网、云计算等应用的兴起，这种扩展模式已经不太适合了，所以近年来，像采用主从配置、数据库复制形式的，Sharding这种技术把负载分布到多个特理节点上去的横向扩展方式用处越来越多。

**Redis缺点：**

1、是数据库容量受到物理内存的限制,不能用作海量数据的高性能读写,因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。

2、Redis较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

**001、redis和memcached的区别？**

存储方式：Memecache把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。Redis有部份存在硬盘上，这样能保证数据的持久性。

数据支持类型：Memcache对数据类型支持相对简单。Redis有复杂的数据类型。

使用底层模型不同：它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

value大小：redis最大可以达到1GB，而memcache只有1MB

**002、如何高效的找到redis中所有以felix开头的key？**

scan 0 match felix\* count 5；表示从游标0开始查询felix开头的key，每次返回5条，但是这个5条不一定；

**003、什么是一致性哈希？**

一致性哈希简称DHT,是麻省理工学院提出的一种算法，目前主要应用于分布式缓存当中。一致性哈希可以有效地解决分布式存储结构下动态增加和删除节点所带来的问题。

**004、redis是单进程单线程的吗？**

Redis采用的是基于内存的采用的是单进程单线程模型的KV数据库；

**005、redis默认多少个db？**

默认有16个数据库；

**006、如果redis中的某个列表中的数据量非常大，如何实现循环显示每一个值？**

使用生成器一个一个取；

**007、redis如何实现主从复制？以及数据同步机制？**

redis从主从结构可以采用一主多从或者级联结构，主从复制可以根据是否全量分为全量同步和增量同步。

全量同步：redis全量复制一般发生在slave初始化阶段，这时slave需要将master上的所有数据都复制一份步骤如下：

1、从服务器连接主服务器，发送SYNC命令；

2、主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；

3、主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；

4、从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；

5、主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；

6、从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；

增量同步：

1、Redis增量复制是指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。 增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

Redis主从同步策略：

1、主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步

**008、redis中sentinel的作用？**

Redis Sentinel是一个用来监控redis集群中节点的状态，不用来存储数据。当集群中的某个节点有故障时，可以自动的进行故障转移的操作。通常为了保证sentinel的高可用，sentinel也会部署多个；

**009、如何实现redis集群？**

1、Twitter开发的twemproxy；2、豌豆荚开发的codis；3、redis官方的redis-cluster；

**010、redis中默认有多少个哈希槽？**

2^14个；Redis 集群没有使用一致性hash, 而是引入了哈希槽的概念。

**011、简述redis的有几种持久化策略以及比较？**

1、RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照。

2、AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写(rewrite)，使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小。

区别：

1、RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

2、AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

**012、列举redis支持的过期策略？**

1、定时删除：在设置key的过期时间的同时，为该key创建一个定时器，让定时器在key的过期时间来临时，对key进行删除；

2、惰性删除：key过期的时候不删除，每次从数据库获取key的时候去检查是否过期，若过期，则删除，返回null；

3、定期删除：每隔一段时间执行一次删除(在redis.conf配置文件设置hz，1s刷新的频率)过期key操作；

**013、如何保证redis中的数据都是热点数据？**

redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。redis 提供 6种数据淘汰策略：

1、volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰；

2、volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰；

3、volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰；

4、allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰；

5、allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰；

6、no-enviction（驱逐）：禁止驱逐数据；

**014、如何基于redis实现发布和订阅？**

# 发布者  
#coding:utf-8import time  
import redis  
number\_list = ['300033', '300032', '300031', '300030']  
signal = ['1', '-1', '1', '-1']  
rc = redis.StrictRedis(host='\*\*\*', port='6379', db=3, password='\*\*\*\*\*\*\*\*')  
for i in range(len(number\_list)):  
   value\_new = str(number\_list[i]) + ' ' + str(signal[i])  
   rc.publish("liao", value\_new)  #发布消息到liao

# 订阅者  
#coding:utf-8import time  
import redis  
rc = redis.StrictRedis(host='\*\*\*\*', port='6379', db=3,     password='\*\*\*\*\*\*')  
ps = rc.pubsub()  
ps.subscribe('liao')  #从liao订阅消息for item in ps.listen():       #监听状态：有消息发布了就拿过来  
   if item['type'] == 'message':  
       print item['channel']  
       print item['data']

**015、什么是codis？**

Codis 是一个分布式 Redis 解决方案, 对于上层的应用来说, 连接到 Codis Proxy 和连接原生的Redis Server 没有明显的区别 (有一些命令不支持), 上层应用可以像使用单机的 Redis 一样使用, Codis 底层会处理请求的转发, 不停机的数据迁移等工作, 所有后边的一切事情, 对于前面的客户端来说是透明的, 可以简单的认为后边连接的是一个内存无限大的 Redis 服务，当然，前段时间redis官方的3.0出了稳定版，3.0支持集群功能，codis的实现原理和3.0的集群功能差不多。

**016、什么是Twemproxy？**

Twemproxy是一种代理分片机制，由Twitter开源。Twemproxy作为代理，可接受来自多个程序的访问，按照路由规则，转发给后台的各个Redis服务器，再原路返回。

该方案很好的解决了单个Redis实例承载能力的问题。当然，Twemproxy本身也是单点，需要用Keepalived做高可用方案。

通过Twemproxy可以使用多台服务器来水平扩张redis服务，可以有效的避免单点故障问题。

虽然使用Twemproxy需要更多的硬件资源和在redis性能有一定的损失（twitter测试约20%），但是能够提高整个系统的HA也是相当划算的。

不熟悉twemproxy的同学，如果玩过nginx反向代理或者mysql proxy，那么你肯定也懂twemproxy了。其实twemproxy不光实现了redis协议

**017、redis如何实现事务？**

Redis事务机制

严格意义来讲,Redis的事务和我们理解的传统数据库(如mysql)的事务是不一样的；Redis的事务实质上是命令的集合，在一个事务中要么所有命令都被执行，要么所有事物都不执行。  
一个事务从开始到执行会经历以下三个阶段：

1、开始事务。

2、命令入队。

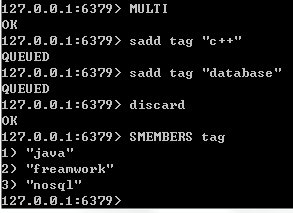
3、执行事务。

在MySQL中我们使用START TRANSACTION 或 BEGIN开启一个事务，使用COMMIT提交一个事务；而在Redis中我们使用MULTI 开始一个事务，由 EXEC 命令触发事务， 一并执行事务中的所有命令。



可以看到，MULTI 开始到 EXEC结束前，中间所有的命令都被加入到一个命令队列中；当执行 EXEC命令后，将QUEUE中所有的命令执行。

此外我们可以使用DISCARD取消事务。



需要注意的是:  
1.Redis的事务没有关系数据库事务提供的回滚（rollback），所以开发者必须在事务执行失败后进行后续的处理；  
2.如果在一个事务中的命令出现错误，那么所有的命令都不会执行；  
3.如果在一个事务中出现运行错误，那么正确的命令会被执行。

WATCH：研究过java的J.U.C包的人应该都知道CAS，CAS是一种保证原子性的操作。那么redis也提供了这样的一个机制，就是利用watch命令来实现的。

WATCH命令可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。

**018、分布式锁？**

什么是分布式锁？

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此干扰来保证一致性，在这种情况下，便需要使用到分布式锁。

实现分布式锁有很多实现方式和工具，如Zookeeper、Redis等。

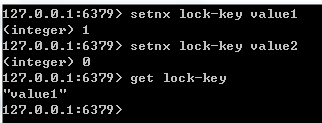
使用Redis实现分布式锁原理：

Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系，基于此，Redis中可以使用SETNX命令实现分布式锁。

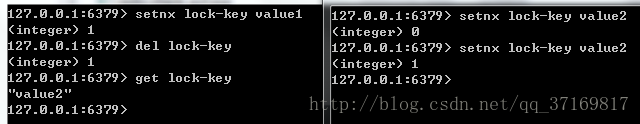
SETNX——SET if Not eXists（如果不存在，则设置）：

setnx key value

将 key 的值设为 value ，当且仅当 key 不存在。  
若给定的 key 已经存在，则 SETNX 不做任何动作。



如果需要解锁，使用 del key 命令就能释放锁：



左图首先使用setnx对键加锁成功返回1，右图再次使用setnx命令对键加锁失败返回0，说明有客户端持有锁。使用del释放锁以后，右图就可以使用setnx命令对键加锁。

**解决死锁：**

如果一个持有锁的客户端失败或崩溃了不能释放锁，该怎么解决？

答：给锁设置一个过期时间，可以通过两种方法实现：通过命令 “setnx 键名 过期时间 “；或者通过设置锁的expire时间，让Redis去删除锁。

**第一种实现方式：**  
使用 setnx key “当前系统时间+锁持有的时间”和getset key “当前系统时间+锁持有的时间”组合的命令就可以实现。  
具体做法如下：

客户端2发送SETNX lock.test 想要获得锁，由于之前的客户端1还持有锁，所以Redis返回一个0  
客户端2发送GET lock.test 以检查锁是否超时了，如果没超时，则等待或重试。  
反之，如果已超时，客户端2通过下面的操作来尝试获得锁：  
GETSET lock.test 过期的时间  
通过GETSET，客户端2拿到的时间戳如果仍然是超时的，那就说明，客户端2如愿以偿拿到锁了。  
如果在客户端2之前，有个客户端3比客户端2快一步执行了上面的操作，那么客户端2拿到的时间戳是个未超时的值，这时，说明客户端2没有如期获得锁，需要再次等待或重试。  
尽管客户端2没拿到锁，但它改写了客户端3设置的锁的超时值，不过这一点非常微小的误差带来的影响可以忽略不计。

**第二种就非常简单了：**  
通过Redis中expire()给锁设定最大持有时间，如果超过，则Redis来帮我们释放锁。

1.客户端1使用setnx获得了锁，并且使用expire设定一个过期时间，假定是10ms

2.过了4ms后，客户端1不幸运的宕机了，此时客户端2想要通过setnx尝试获得锁，但是锁还没有过期，任然被客户端1所持有。

3.到了11ms时，锁过期了，Redis帮我们删除了锁，此时客户端2想要通过setnx尝试获得锁，此时就能成功获得锁。

在实际过程中，我们可以设定一个时间T，用来表示客户端在初次尝试获得锁失败以后，在多次尝试获得锁所花的时间。如果次时间为0，表示除此尝试获得锁失败以后就不会再去尝试获得锁了。

**019、redis中watch的作用？**

watch 用于在进行事务操作的最后一步也就是在执行exec 之前对某个key进行监视 如果这个被监视的key被改动，那么事务就被取消，否则事务正常执行. 一般在MULTI 命令前就用watch命令对某个key进行监控.如果想让key取消被监控，可以用unwatch命令；

**020、Django项目中如何对接Redis高速缓存服务？**

settings.py文件中配置

CACHES = {

'default': {

'BACKEND': 'django\_redis.cache.RedisCache',

'LOCATION': [

'redis://120.77.222.217:6379/0',

],

'KEY\_PREFIX': '生成key的前缀', # 这里随便写

'OPTIONS': {

'CLIENT\_CLASS': 'django\_redis.client.DefaultClient',

'CONNECTION\_POOL\_KWARGS': {

'max\_connections': 256,

},

'PASSWORD': '密码',

}

},

"my\_redis": {

"BACKEND": "django\_redis.cache.RedisCache",

"LOCATION": "redis://127.0.0.1:6379/1",

"OPTIONS": {

"CLIENT\_CLASS": "django\_redis.client.DefaultClient",

'CONNECTION\_POOL\_KWARGS': {

'max\_connections': 256,

},

'PASSWORD': '123456',

}

},

}

# 使用引擎

SESSION\_ENGINE = 'django.contrib.sessions.backends.cache'

# 设置过期时间86400秒

SESSION\_COOKIE\_AGE = 86400

**021、Redis宕机怎么解决?**

宕机:服务器停止服务

如果只有一台redis，肯定 会造成数据丢失，无法挽救

多台redis或者是redis集群，宕机则需要分为在主从模式下区分来看：

slave从redis宕机，配置主从复制的时候才配置从的redis，从的会从主的redis中读取主的redis的操作日志，在redis中从库重新启动后会自动加入到主从架构中，自动完成同步数据;

如果从数据库实现了持久化，此时千万不要立马重启服务，否则可能会造成数据丢失，正确的操作如下：在slave数据上执行SLAVEOF ON ONE,来断开主从关系并把slave升级为主库，此时重新启动主数据库，执行SLAVEOF，把它设置为从库，连接到主的redis上面做主从复制，自动备份数据。

以上过程很容易配置错误，可以使用redis提供的哨兵机制来简化上面的操作。简单的方法:redis的哨兵(sentinel)的功能

**022、redis和mecached的区别，以及使用场景？**

区别

1、redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可以用于缓存其他东西，例如图片，视频等等；

2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list,set,hash等数据结构的存储；

3、虚拟内存-redis当物流内存用完时，可以将一些很久没用的value交换到磁盘；

4、过期策略-memcache在set时就指定，例如set key1 0 0 8，即永不过期。Redis可以通过例如expire设定，例如expire name 10；

5、分布式-设定memcache集群，利用magent做一主多从，redis可以做一主多从。都可以一主一丛；

6、存储数据安全-memcache挂掉后，数据没了，redis可以定期保存到磁盘(持久化)；

7、灾难恢复-memcache挂掉后，数据不可恢复，redis数据丢失后可以通过aof恢复；

8、Redis支持数据的备份，即master-slave模式的数据备份；

9、应用场景不一样，redis除了作为NoSQL数据库使用外，还能用做消息队列，数据堆栈和数据缓存等;Memcache适合于缓存SQL语句，数据集，用户临时性数据，延迟查询数据和session等；

使用场景

1、如果有持久方面的需求或对数据类型和处理有要求的应该选择redis；

2、如果简单的key/value存储应该选择memcached.；

**023、Redis集群方案该怎么做?都有哪些方案?**

1、redis 目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在节点数量改变情况下，旧节点数据客恢复到新hash节点；

2、redis cluster3.0自带的集群，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。具体看官方介绍；

3、在业务代码层实现，起几个毫无关联的redis实例，在代码层，对key进行hash计算，然后去对应的redis实例操作数据。这种方式对hash层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的字典脚本恢复，实例的监控，等等；

**024、Redis回收进程是如何工作的？**

1、一个客户端运行了新的命令，添加了新的数据。

2、redis检查内存使用情况，如果大于maxmemory的限制，则根据设定好的策略进行回收。

3、一个新的命令被执行等等，所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断回收回到边界以下。

4、如果一个命令的结果导致大量内存被使用(例如很大的集合的交集保存到一个新的键)，不用多久内存限制就会被这个内存使用量超越。

**Redis**

string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）

**简介**

redis是一个key-value存储系统。和Memcached（也是一个数据库）类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步

**使用Redis有哪些好处？**

(1) 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

(2) 支持丰富数据类型，支持string，list，set，sorted set，hash

(3) 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行

(4) 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

**redis相比memcached有哪些优势？**

(1) memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型；

(2) redis的速度比memcached快很多；

(3) redis可以持久化其数据；

**redis常见性能问题和解决方案：**

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件；

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次；

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内；

(4) 尽量避免在压力很大的主库上增加从库；

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...；

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

**MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据**

相关知识：redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。redis 提供 6种数据淘汰策略：

voltile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据

**Memcache与Redis的区别都有哪些？**

1)、存储方式：

Memecache把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。Redis有部份存在硬盘上，这样能保证数据的持久性。

2)、数据支持类型：

Memcache对数据类型支持相对简单。Redis有复杂的数据类型。

3），value大小：redis最大可以达到1GB，而memcache只有1MB

**Redis 常见的性能问题都有哪些？如何解决？**

1).Master写内存快照，save命令调度rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以Master最好不要写内存快照。

2).Master AOF持久化，如果不重写AOF文件，这个持久化方式对性能的影响是最小的，但是AOF文件会不断增大，AOF文件过大会影响Master重启的恢复速度。Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化,如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。

3).Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。

4). Redis主从复制的性能问题，为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内

**redis 最适合的场景**

Redis最适合所有数据in-momory的场景，虽然Redis也提供持久化功能，但实际更多的是一个disk-backed的功能，跟传统意义上的持久化有比较大的差别，那么可能大家就会有疑问，似乎Redis更像一个加强版的Memcached，那么何时使用Memcached,何时使用Redis呢?

如果简单地比较Redis与Memcached的区别，大多数都会得到以下观点：

1 、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存储。 2 、Redis支持数据的备份，即master-slave模式的数据备份。 3 、Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

（1）、会话缓存（Session Cache）

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

（2）、全页缓存（FPC）

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。

再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。

此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

（3）、队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。

如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从这里去查看。

（4），排行榜/计数器

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user\_scores 0 10 WITHSCORES

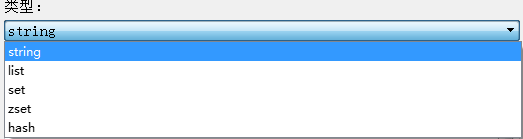
Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

（5）、发布/订阅

最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

Redis提供的所有特性中，我感觉这个是喜欢的人最少的一个，虽然它为用户提供如果此多功能。

**支持的数据类型（5大数据类型）**



redis={

k1:'123', 字符串

k2:[1,2,3,4], 列表/数组

k3:{1,2,3,4} 集合

k4:{name:lqz,age:12} 字典/哈希表

k5:{('lqz',18),('egon',33)} 有序集合

}

**特点：**

可以持久化

单线程，单进程

import redis

pool = redis.ConnectionPool(host='127.0.0.1', port=6379)

r = redis.Redis(connection\_pool=pool)

r.set('foo', 'Bar')

print(r.get('foo'))

**4 发布订阅，观察者模式**

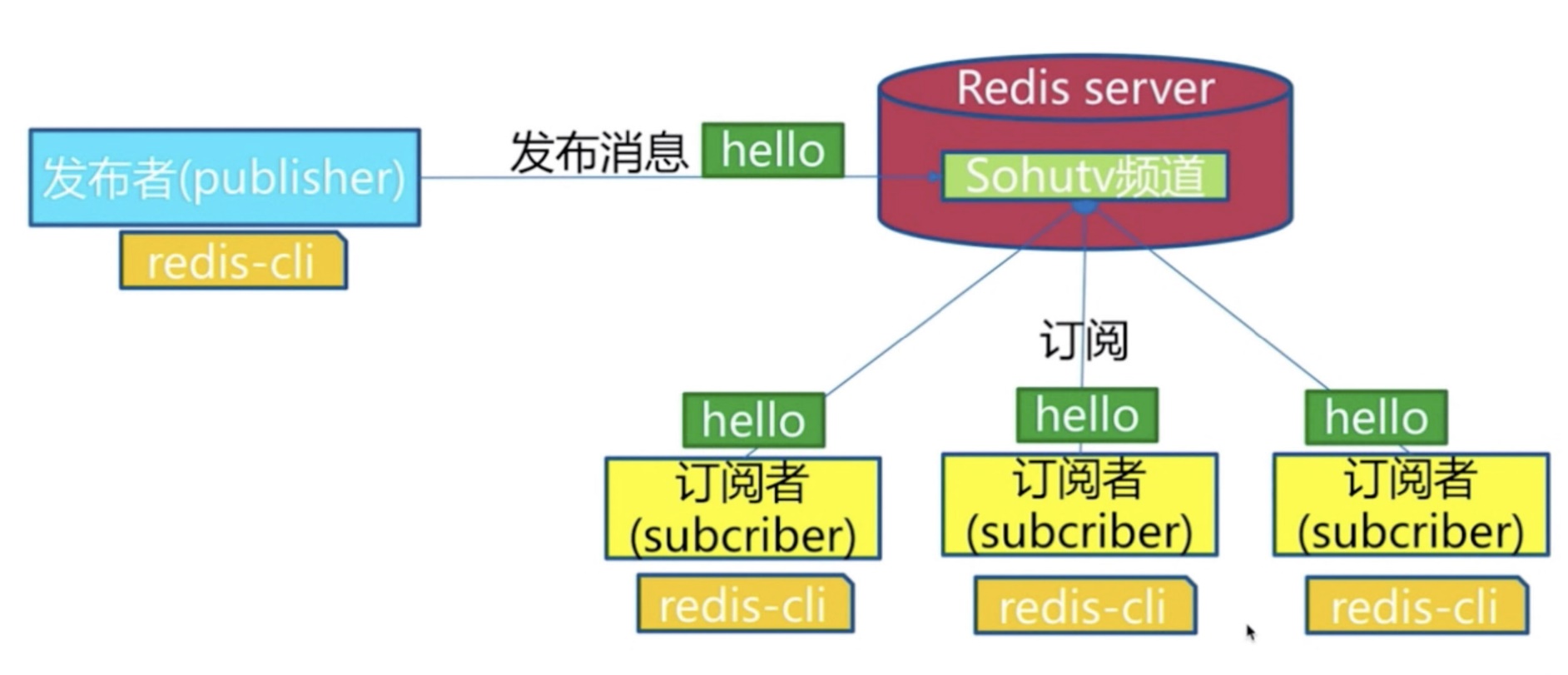
**发布订阅**

**3.1 角色**

**发布者/订阅者/频道**

发布者发布了消息，所有的订阅者都可以收到，就是生产者消费者模型（后订阅了，无法获取历史消息）

**3.2 模型**



###

**5 GEO 地理位置信息**

附近的人，查找附近的商铺，计算两个点的直线距离

**6 持久化方案**

**RDB和AOF的选择**

**4.1 rdb和aof的比较**

| **命令** | **rdb** | **aof** |
| --- | --- | --- |
| 启动优先级 | 低 | 高(挂掉重启，会加载aof的数据) |
| 体积 | 小 | 大 |
| 恢复速度 | 快 | 慢 |
| 数据安全性 | 丢数据 | 根据策略决定 |
| 轻重 | 重 | 轻 |

**4.2 rdb最佳策略**

rdb关掉，主从操作时；

集中管理：按天，按小时备份数据；

主从配置，从节点打开；

**4.3 aof最佳策略**

开：缓存和存储，大部分情况都打开，

aof重写集中管理；

everysec：通过每秒刷新的策略；

**4.4 最佳策略**

小分片：每个redis的最大内存为4g；

缓存或存储：根据特性，使用不通策略；

时时监控硬盘，内存，负载网络等；

有足够内存

**7 主从复制**

一台机器上可以启动多个redis服务，监听的端口不同；

./src/redis-cli -p 6378 连接6378

**通过命令来做**

slaveof 127.0.0.1 6379

**通过配置文件（写到配置文件）**

slaveof ip port #配置从节点ip和端口 slave-read-only yes #从节点只读，因为可读可写，数据会乱

**8 哨兵高可用**

**一 主从复制高可用**

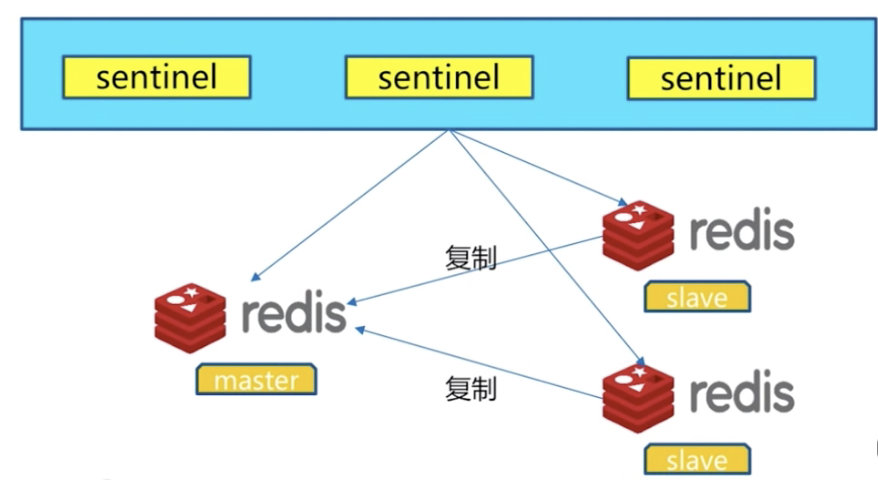
#主从复制存在的问题：

#1 主从复制，主节点发生故障，需要做故障转移，可以手动转移：让其中一个slave变成master；

#2 主从复制，只能主写数据，所以写能力和存储能力有限；

**二 架构说明**

可以做故障判断，故障转移，通知客户端（其实是一个进程），客户端直接连接sentinel的地址



1 多个sentinel发现并确认master有问题；

2 选举触一个sentinel作为领导；

3 选取一个slave作为新的master；

4 通知其余slave成为新的master的slave；

5 通知客户端主从变化；

6 等待老的master复活成为新master的slave；

**1 搭一个一主两从**

#创建三个配置文件：

#第一个是主配置文件

daemonize yes

pidfile /var/run/redis.pid

port 6379

dir "/opt/soft/redis/data"

logfile “6379.log”

#第二个是从配置文件

daemonize yes

pidfile /var/run/redis2.pid

port 6378

dir "/opt/soft/redis/data2"

logfile “6378.log”

slaveof 127.0.0.1 6379

slave-read-only yes

#第三个是从配置文件

daemonize yes

pidfile /var/run/redis3.pid

port 6377

dir "/opt/soft/redis/data3"

logfile “6377.log”

slaveof 127.0.0.1 6379

slave-read-only yes

#把三个redis服务都启动起来

./src/redis-server redis\_6379.conf

./src/redis-server redis\_6378.conf

./src/redis-server redis\_6377.conf

2 搭建哨兵

# sentinel.conf这个文件

# 把哨兵也当成一个redis服务器

创建三个配置文件分别叫sentinel\_26379.conf sentinel\_26378.conf sentinel\_26377.conf

#内容如下(需要修改端口，文件地址日志文件名字)

port 26379

daemonize yes

dir ./data3

protected-mode no

bind 0.0.0.0

logfile "redis\_sentinel3.log"

sentinel monitor mymaster 127.0.0.1 6379 2

sentinel down-after-milliseconds mymaster 30000

sentinel parallel-syncs mymaster 1

sentinel failover-timeout mymaster 180000

#启动三个哨兵

./src/redis-sentinel sentinel\_26379.conf

./src/redis-sentinel sentinel\_26378.conf

./src/redis-sentinel sentinel\_26377.conf

# 主动停掉主redis 6379，哨兵会自动选择一个从库作为主库

#等待原来的主库启动，该主库会变成从库

redis五大数据类型 答：巴拉巴拉

redis如果存多了要怎么办 答：持久化存储。1. 增加内存，2. 使用淘汰策略，3. 使用集群。

重点介绍下23；

第2点：

我们知道，redis设置配置文件的maxmemory参数，可以控制其最大可用内存大小（字节）。

那么当所需内存，超过maxmemory怎么办？

这个时候就该配置文件中的maxmemory-policy出场了。

其默认值是noeviction。

下面我将列出当可用内存不足时，删除redis键具有的淘汰规则。

规则名称

规则说明

volatile-lru

使用LRU算法删除一个键（只对设置了生存时间的键）

allkeys-lru

使用LRU算法删除一个键

volatile-random

随机删除一个键（只对设置了生存时间的键）

allkeys-random

随机删除一个键

volatile-ttl

删除生存时间最近的一个键

noeviction

不删除键，只返回错误

LRU算法，least RecentlyUsed，最近最少使用算法。也就是说默认删除最近最少使用的键。

但是一定要注意一点！redis中并不会准确的删除所有键中最近最少使用的键，而是随机抽取3个键，删除这三个键中最近最少使用的键。

那么3这个数字也是可以设置的，对应位置是配置文件中的maxmeory-samples.

 第3点：

3.集群怎么做

Redis仅支持单实例，内存一般最多10~20GB。对于内存动辄100~200GB的系统，就需要通过集群来支持了。

Redis集群有三种方式：客户端分片、代理分片、RedisCluster(在之后一篇文章详细说一下。)

·        客户端分片

通过业务代码自己实现路由

优势：可以自己控制分片算法、性能比代理的好

劣势：维护成本高、扩容/缩容等运维操作都需要自己研发

·        代理分片

代理程序接收到来自业务程序的数据请求，根据路由规则，将这些请求分发给正确的Redis实例并返回给业务程序。使用类似Twemproxy、Codis等中间件实现。

优势：运维方便、程序不用关心如何链接Redis实例

劣势：会带来性能消耗（大概20%）、无法平滑扩容/缩容，需要执行脚本迁移数据，不方便(Codis在Twemproxy基础上优化并实现了预分片来达到Auto Rebalance)。

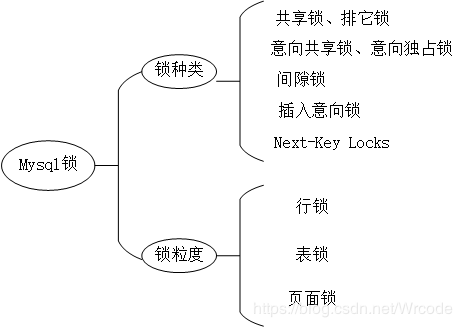
·        Redis Cluster

优势：官方集群解决方案、无中心节点，和客户端直连，性能较好

劣势：方案太重、无法平滑扩容/缩容，需要执行相应的脚本，不方便、太新，没有相应成熟的解决案例

秒杀用什么锁 答：悲观锁，其实应该用乐观锁

简单的秒杀实现方式比较多，常见的有可重入锁、redis分布式锁、线程同步锁、数据库锁等，其中最简单也最容易实现的就是数据库乐观锁了，下面的demo以springboot+Data JPA 框架为基础，利用mysql乐观锁实现了一个简单的秒杀场景，乐观锁本质并不会加锁，只是在进行数据修改做版本校验，校验通过更新数据，否则不更新。本demo仅供了解并发学习之用，还有很多未完善的地方，请大家多多包涵。



背景：有库存数要求的秒杀活动。

基于redis解决方案

1、基于watch的乐观锁方案  
watch用于监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。这种方案跟mysql中的乐观锁方案类似，具体表现也是一样的。

2、基于list的队列方案  
基于队列的方案利用了redis出队操作的原子性，抢购开始之前首先将商品编号放入响应的队列中，在抢购时依次从队列中弹出操作，这样可以保证每个商品只能被一个进程获取并操作，不存在超发的情况。该方案的优点是理解和实现起来都比较简单，缺点是当商品数量较多是，需要将大量的数据存入到队列中，并且不同的商品需要存入到不同的消息队列中

**Mongodb**

1、在概念上，MongoDB的文档与Javascript的对象相近，因而可以认为它类似于JSON。JSON（<http://www.json.org>）是一种简单的数据表示方式：其规范仅用一段文字就能描述清楚（其官网证明了这点），且仅包含六种数据类型。

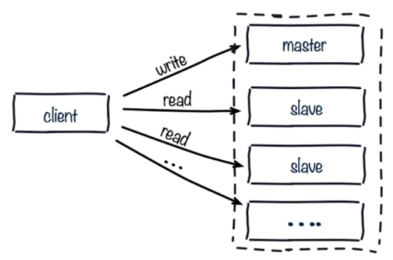
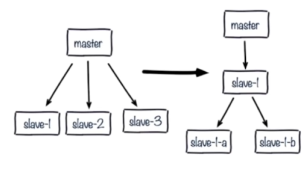
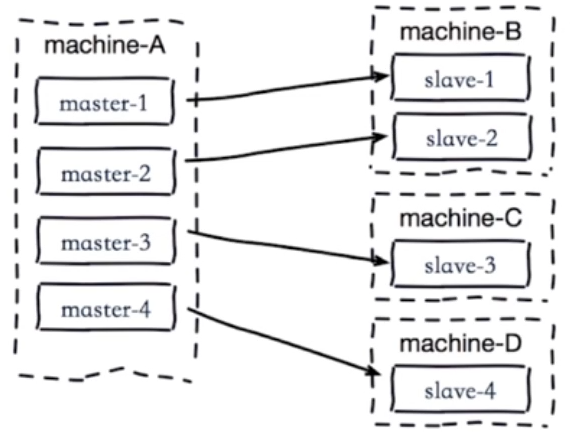
2、这样有很多好处：易于理解、易于解析、易于记忆。然而从另一方面说，因为只有null、布尔、数字、字符串、数字和对象这几种数据类型，所以JSON的表达能力有一定的局限。

3、虽然JSON具备的这些类型已经具有很强的表现力，但绝大数应用（尤其是在于数据库打交道时）都还需要其他一些重要的类型。例如，JSON没有日期类型，这使得原本容易日期处理变得烦人。另外，JSON只有一种数字类型，无法区分浮点数和整数，更别区分32位和64位了。再者JSON无法表示其他一些通用类型，如正则表达式或函数。

4、MongoDB在保留了JSON基本键/值对特性的基础上，添加了其他一些数据类型。在不同的编程语言下，这些类型的确切表示有些许差异。下面说明了MongoDB支持的其他通用类型，以及如何正在文档中使用它们

**TCP/IP协议**

TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议）是指能够在多个不同网络间实现信息传输的协议簇。TCP/IP协议不仅仅指的是[TCP](https://baike.baidu.com/item/TCP/33012) 和[IP](https://baike.baidu.com/item/IP/224599)两个协议，而是指一个由[FTP](https://baike.baidu.com/item/FTP/13839)、[SMTP](https://baike.baidu.com/item/SMTP/175887)、TCP、[UDP](https://baike.baidu.com/item/UDP/571511)、IP等协议构成的协议簇， 只是因为在TCP/IP协议中TCP协议和IP协议最具代表性，所以被称为TCP/IP协议。

* TCP（transport control protocol，传输控制协议）是面向连接的，面向流的，提供高可靠性服务。收发两端（客户端和服务器端）都要有一一成对的socket，因此，发送端为了将多个发往接收端的包，更有效的发到对方，使用了优化方法（Nagle算法），将多次间隔较小且数据量小的数据，合并成一个大的数据块，然后进行封包。这样，接收端，就难于分辨出来了，必须提供科学的拆包机制。 即面向流的通信是无消息保护边界的。
* UDP（user datagram protocol，用户数据报协议）是无连接的，面向消息的，提供高效率服务。不会使用块的合并优化算法，, 由于UDP支持的是一对多的模式，所以接收端的skbuff(套接字缓冲区）采用了链式结构来记录每一个到达的UDP包，在每个UDP包中就有了消息头（消息来源地址，端口等信息），这样，对于接收端来说，就容易进行区分处理了。 即面向消息的通信是有消息保护边界的。
* TCP是基于数据流的，于是收发的消息不能为空，这就需要在客户端和服务端都添加空消息的处理机制，防止程序卡住，而udp是基于数据报的，即便是你输入的是空内容（直接回车），那也不是空消息，udp协议会帮你封装上消息头，实验略；
* 1、为什么要搭建主从系统
* 这已经是业界很多应用的通用做法了，跟mysql一样搭建主从常见的就是可以实现读写分离，至于为什么要做读写分离从redis的本身来讲：redis的操作具有原子性，同时使用的是单线程(这样可以避免多线程之间切换消耗)操作,过多的写会阻塞线程影响redis的整体性能同时也会影响redis读性能。
* 此外当我们集群中或者单机的主节点宕机了或者无法跟集群的其它节点就行通讯的话，如果没有从节点则这部分缓存数据丢失，如果有从节点时我们可以用节点数据继续服务因为几乎所有的数据主节点都同步到从节点啦，可以正常使用。
* 2、主从特性
* 1、一个 master 可以拥有一个或者多个 slave。
* 2、slave 可以接收来自其他 slave 的连接。意思是不是就是说 slave 在接收其他的slave的连接之后成为 master ？等下我们来验证。
* 3、redis 复制在 master 这一端是非阻塞的，也就是说在和 slave 同步数据的时候，master 仍然可以执行客户端的操作命令而不受其影响。
* 4、redis 复制在 slave 这一端也是非阻塞的。在配置文件里面有 slave-serve-stale-data 这一项，如果它为 yes ，slave 在执行同步时，它可以使用老版本的数据来处理查询请求，如果是 no ，slave 将返回一个错误。在完成同步后，slave 需要删除老数据，加载新数据，在这个阶段，slave 会阻止连接进来。
* 5、使用复制可以避免 master 因为需要把全部的数据集写入磁盘而造成的开销，因此可以把 master 中 save 配置项全部注释掉，不让它进行保存，然后配置 slave ，让 slave 保存。虽然有这个特性，但是我们好像一般不这么做。
* 3、实现过程略。
* 主从实现原理剖析：
* 当你设置了主从关系后，slave 在第一次连接或者重新连接 master 时，slave 都会发送一条同步指令给 master ；master 接到指令后，开始启动后台保存进程保存数据，接着收集所有的数据修改指令。后台保存完了，master 就把这份数据发送给 slave，slave 先把数据保存到磁盘，然后把它加载到内存中，master 接着就把收集的数据修改指令一行一行的发给 slave，slave 接收到之后重新执行该指令，这样就实现了数据同步。slave 在与 master 失去联系后，自动的重新连接。如果 master 收到了多个 slave 的同步请求，它会执行单个后台保存来为所有的 slave 服务。
* 主从的缺点是什么 答：主库挂了就完蛋了。
* **读写分离的问题**
* 
* 1.数据复制的延迟
* 读写分离时，master会异步的将数据复制到slave，如果这是slave发生阻塞，则会延迟master数据的写命令，造成数据不一致的情况
* 解决方法：可以对slave的偏移量值进行监控，如果发现某台slave的偏移量有问题，则将数据读取操作切换到master，但本身这个监控开销比较高，所以关于这个问题，大部分的情况是可以直接使用而不去考虑的。
* 2.读到过期的数据
* 我们知道redis在删除过期key的时候，是有两种策略，第一种是懒惰型策略，即只有当redis操作这个key的时候，发现这个key过期，就会把这个key删除。第二种是定期采样一些key进行删除。
* 针对上面说的两种过期策略，会有个问题，即如果我们过期key的数量非常多，而采样速度根本比不上过期key的生成速度时会造成很多过期数据没有删除，但在redis里master和slave达成一种协议，slave是不能处理数据的（即不能删除数据）而我们的客户端没有及时读到到过期数据同步给master将key删除，就会导致slave读到过期的数据（这个问题已经在redis3.2版本中解决）
* 主从配置不一致
* 这个问题一般很少见，但如果有，就会发生很多诡异的问题
* 例如：
* 1. maxmemory配置不一致：这个会导致数据的丢失
* 原因：例如master配置4G，slave配置2G，这个时候主从复制可以成功，但，如果在进行某一次全量复制的时候，slave拿到master的RDB加载数据时发现自身的2G内存不够用，这时就会触发slave的maxmemory策略，将数据进行淘汰。更可怕的是，在高可用的集群环境下，如果我们将这台slave升级成master的时候，就会发现数据已经丢失了。
* 2. 数据结构优化参数不一致（例如hash-max-ziplist-entries）：这个就会导致内存不一致
* 原因：例如在master上对这个参数进行了优化，而在slave没有配置，就会造成主从节点内存不一致的诡异问题。
* 规避全量复制
* 首先，我们知道，redis复制有全量复制和部分复制两种（这个我前面博客有写到）而全量复制的开销是很大的。那么我们来看看，如何尽量去规避全量复制。
* 1.第一次全量复制
* 当我们某一台slave第一次去挂到master上时，是不可避免要进行一次全量复制的，那么，我们如何去想办法降低开销呢？
* 方案1：小主节点，例如我们把redis分成2G一个节点，这样一来，会加速RDB的生成和同步，同时还可以降低我们fork子进程的开销（master会fork一个子进程来生成同步需要的RDB文件，而fork是要拷贝内存快的，如果主节点内存太大，fork的开销就大）。
* 方案2：既然第一次不可以避免，那我们可以选在集群低峰的时间（凌晨）进行slave的挂载。
* 2.节点RunID不匹配
* 例如我们主节点重启（RunID发生变化），对于slave来说，它会保存之前master节点的RunID，如果它发现了此时master的RunID发生变化，那它会认为这是master过来的数据可能是不安全的，就会采取一次全量复制
* 解决办法：对于这类问题，我们只有是做一些故障转移的手段，例如master发生故障宕掉，我们选举一台slave晋升为master（哨兵或集群）
* 3.复制积压缓冲区不足
* 我在全量复制与部分复制那篇文章提到过，master生成RDB同步到slave，slave加载RDB这段时间里，master的所有写命令都会保存到一个复制缓冲队列里（如果主从直接网络抖动，进行部分复制也是走这个逻辑），待slave加载完RDB后，拿offset的值到这个队列里判断，如果在这个队列中，则把这个队列从offset到末尾全部同步过来，这个队列的默认值为1M。而如果发现offset不在这个队列，就会产生全量复制。
* 解决办法：增大复制缓冲区的配置 rel\_backlog\_size 默认1M，我们可以设置大一些，从而来加大我们offset的命中率。这个值，我们可以假设，一般我们网络故障时间一般是分钟级别，那我们可以根据我们当前的QPS来算一下每分钟可以写入多少字节，再乘以我们可能发生故障的分钟就可以得到我们这个理想的值。
* 规避复制风暴
* 什么是复制风暴？举例：我们master重启，其master下的所有slave检测到RunID发生变化，导致所有从节点向主节点做全量复制。尽管redis对这个问题做了优化，即只生成一份RDB文件，但需要多次传输，仍然开销很大。
* 1.单主节点复制风暴：主节点重启，多从节点全量复制
* 解决：更换复制拓扑如下图：
* 
* 1.我们将原来master与slave中间加一个或多个slave，再在slave上加若干个slave，这样可以分担所有slave对master复制的压力。（这种架构还是有问题：读写分离的时候，slave1也发生了故障，怎么去处理？）
* 2.如果只是实现高可用，而不做读写分离，那当master宕机，直接晋升一台slave即可。
* 2.单机器复制风暴：机器宕机后的大量全量复制，如下图：
* 
* 当machine-A这个机器宕机重启，会导致该机器所有master下的所有slave同时产生复制。（灾难）
* 解决：
* 1.主节点分散多机器（将master分散到不同机器上部署）
* 2.还有我们可以采用高可用手段（slave晋升master）就不会有类似问题了。
* 转载于:https://my.oschina.net/u/3371837/blog/1789452

**002、redis你都用来做了些什么？**

* 答：做消息队列，做缓存。
* 一、redis是什么
* Redis（全称：Remote Dictionary Server 远程字典服务）是一个开源的使用ANSI [C语言](https://baike.baidu.com/item/C%E8%AF%AD%E8%A8%80)编写、支持网络、可基于内存亦可持久化的日志型、Key-Value nosql [数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/103728)，并提供多种语言的API。
* 二、redis的应用场景
* 1、缓存
* 缓存现在几乎是所有中大型网站都在用的必杀技，合理的利用缓存不仅能够提升网站访问速度，还能大大降低数据库的压力。Redis提供了键过期功能，也提供了灵活的键淘汰策略，所以，现在Redis用在缓存的场合非常多。
* 2、排行榜
* 很多网站都有排行榜应用的，如京东的月度销量榜单、商品按时间的上新排行榜等。Redis提供的有序集合数据类构能实现各种复杂的排行榜应用。
* 3、计数器
* 什么是计数器，如电商网站商品的浏览量、视频网站视频的播放数等。为了保证数据实时效，每次浏览都得给+1，并发量高时如果每次都请求数据库操作无疑是种挑战和压力。
* Redis提供的incr命令来实现计数器功能，内存操作，性能非常好，非常适用于这些计数场景。
* 4、分布式会话
* 集群模式下，在应用不多的情况下一般使用容器自带的session复制功能就能满足，当应用增多相对复杂的系统中，一般都会搭建以Redis等内存数据库为中心的session服务，session不再由容器管理，而是由session服务及内存数据库管理。
* 5、分布式锁
* 在很多互联网公司中都使用了分布式技术，分布式技术带来的技术挑战是对同一个资源的并发访问，如全局ID、减库存、秒杀等场景，并发量不大的场景可以使用数据库的悲观锁、乐观锁来实现，但在并发量高的场合中，利用数据库锁来控制资源的并发访问是不太理想的，大大影响了数据库的性能。可以利用Redis的setnx功能来编写分布式的锁，如果设置返回1说明获取锁成功，否则获取锁失败，实际应用中要考虑的细节要更多。
* 6、 社交网络
* 点赞、踩、关注/被关注、共同好友等是社交网站的基本功能，社交网站的访问量通常来说比较大，而且传统的关系数据库类型不适合存储这种类型的数据，Redis提供的哈希、集合等数据结构能很方便的的实现这些功能。
* 7、最新列表
* Redis列表结构，LPUSH可以在列表头部插入一个内容ID作为关键字，LTRIM可用来限制列表的数量，这样列表永远为N个ID，无需查询最新的列表，直接根据ID去到对应的内容页即可。
* 8、消息系统(不推荐使用)
* 消息队列是大型网站必用中间件，如ActiveMQ、RabbitMQ、Kafka等流行的消息队列中间件，主要用于业务解耦、流量削峰及异步处理实时性低的业务。Redis提供了发布/订阅及阻塞队列功能，能实现一个简单的消息队列系统。
* 三、redis的优点和缺点
* 优点：
* （1）速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O1)
* （2）支持丰富数据类型，支持 string，list，set，Zset，hash 等
* （3）支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
* （4）丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除，支持数据的持久化，支持数据的备份，即master-slave模式的数据备份。
* 缺点：
* （1）由于 Redis 是内存数据库，所以，单台机器，存储的数据量，跟机器本身的内存大小。虽然 Redis 本身有 Key 过期策略，但是还是需要提前预估和节约内存。如果内存增长过快，需要定期删除数据。
* （2）redis是单线程的，单台服务器无法充分利用多核服务器的CPU。
* 四、redis为什么快？
* 1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；
* 2、数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；
* 3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
* 4、使用多路I/O复用模型，非阻塞IO；
* 5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；
* 1. 获取最新的n个数据，2. 获取 TOP N 的数据，3. 设置精准的抢购时间，4. 轻松实现计数器
* ，5. 去除重复值，6. 利用set命令制作反垃圾系统，7. 利用发布订阅的中的SUBSCRIBE和PUBLISH命令构建实时消息推送系统，8. 构建队列系统，
* redis可以用来做什么？以博客帖子为例大概说说，当然redis的其他功能还有很多，后续在做介绍。
* Redis 是互联网技术领域使用最为广泛的存储中间件，它是「Remote Dictionary Service」的首字母缩写，也就是「远程字典服务」。
* 记录帖子的点赞数、评论数和点击数 (hash) --针对帖子的一对多记录， key为数名称，value为各种数的值。
* 记录用户的帖子 ID 列表 (排序)，便于快速显示用户的帖子列表 (zset) -- 记录id而不是文章是为了需要减少内存，同时zset 的score可以用来排序。
* 记录帖子的标题、摘要、作者和封面信息，用于列表页展示 (hash) -- 针对帖子的一对多记录，key为必要信息的名称，value为各种值。
* 记录帖子的点赞用户 ID 列表，评论 ID 列表，用于显示和去重计数 (zset) -- 单个字段记录，不重复。
* 缓存近期热帖内容 (帖子内容空间占用比较大)，减少数据库压力 (hash) -- 热帖数量不多，所以内容全部缓存，key为id，value为帖子内容。
* 记录帖子的相关文章 ID，根据内容推荐相关帖子 (list) -- 相关文章，有相关度的概念，所以需要有序，而list可以作为队列和栈来使用。
* 如果帖子 ID 是整数自增的，可以使用 Redis 来分配帖子 ID(计数器)。 收藏集和帖子之间的关系 (zset) -- 有序不重复。
* 记录热榜帖子 ID 列表，总热榜和分类热榜 (zset) -- 有序不重复。
* 缓存用户行为历史，进行恶意行为过滤 (zset,hash) -- 有序不重复，针对用户一对多行为记录。

**003、redis数据怎么持久化存储？**

* 答：忘记了。
* 从严格意义上说，Redis服务提供四种持久化存储方案：RDB、AOF、虚拟内存（VM）和　DISKSTORE。虚拟内存（VM）方式，从Redis Version 2.4开始就被官方明确表示不再建议使用，Version 3.2版本中更找不到关于虚拟内存（VM）的任何配置范例，Redis的主要作者Salvatore Sanfilippo还专门写了一篇论文，来反思Redis对虚拟内存（VM）存储技术的支持问题。
* 至于DISKSTORE方式，是从Redis Version 2.8版本开始提出的一个存储设想，到目前为止Redis官方也没有在任何stable版本中明确建议使用这用方式。在Version 3.2版本中同样找不到对于这种存储方式的明确支持。从网络上能够收集到的各种资料来看，DISKSTORE方式和RDB方式还有着一些千丝万缕的联系，不过各位读者也知道，除了官方文档以外网络资料很多就是大抄。
* **> 为什么要做持久化存储?**
* 持久化存储是将 Redis 存储在内存中的数据存储在硬盘中，实现数据的永久保存。我们都知道 Redis 是一个基于内存的 nosql 数据库，内存存储很容易造成数据的丢失，因为当服务器关机等一些异常情况都会导致存储在内存中的数据丢失。  
  **> 持久化存储分类**
* 在 Redis 中，持久化存储分为两种AOF和RDB。一种是 aof 日志追加的方式，另外一种是 rdb 数据快照的方式。  
  **> RDB持久化存储**
* **什么是RDB持久化存储**  
  RDB持久化存储即是将redis存在内存中的数据以快照的形式保存在本地磁盘中。
* .RDB持久化存储分为自动备份和手动备份  
  1.手动备份通过 save 命令和 bgsave 命令。save是同步阻塞，而 bgsave 是非阻塞(阻塞实际发生在 fork 的子进程中)。因此，在我们实际过程中大多是使用bgsave命令实现备份.
* RDB持久化的优势与劣势  
  优势:  
  1.文件实现的数据快照，全量备份，便于数据的传输.比如我们需要把A服务器上的备份文件传输到B服务器上面，直接将rdb文件拷贝即可.  
  2.文件采用压缩的二进制文件，当重启服务时加载数据文件，比aof方式更快.  
  劣势:  
  1.rbd采用加密的二进制格式存储文件，由于Redis各个版本之间的兼容性问题也导致rdb由版本兼容问题导致无法再其他的Redis版本中使用.  
  2.时效性差，容易造成数据的不完整性.因为rdb并不是实时备份，当某个时间段Redis服务出现异常，内存数据丢失，这段时间的数据是无法恢复的，因此易导致数据的丢失.

**004、redis的哨兵讲一下 ？**

* 答：哨兵用来实现高可用巴拉巴拉。
* sentinel是redis高可用的解决方案，sentinel系统（N个sentinel实例，N >= 1）可以监视一个或者多个redis master服务，以及这些master服务的所有从服务；当某个master服务下线时，自动将该master下的某个从服务升级为master服务替代已下线的master服务继续处理请求。
* Redis的主从[架构](http://lib.csdn.net/base/16)，如果master发现故障了，还得手动将slave切换成master继续服务，手动的方式容易造成失误，导致数据丢失，那Redis有没有一种机制可以在master和slave进行监控，并在master发送故障的时候，能自动将slave切换成master呢？有的，那就是哨兵。
* 哨兵的作用：   
  1、监控redis进行状态，包括master和slave   
  2、当master down机，能自动将slave切换成master
* 为了达到redis的高可用，有两种部署方式：主从复制+哨兵机制；集群模式。哨兵机制是redis2.8开始支持。集群模式是redis3.0开始支持。
* 主从复制的意义：
* 主从复制可以把主节点的数据复制给从节点。从节点可以备份主节点的数据，起到主节点down调，顶上来接替主节点工作的作用。也可以起到分担主节点读压力的作用。
* 没有哨兵机制的时候，主从复制结构部署存在的问题是什么？也可以说redis主节点发生故障如何解决？
* 如果主节点down调，主从切换需要人工介入。
* 主从切换步骤为：
* 1、启用从节点为主节点。命令：slaveof no one
* 2、旧主节点的其他从节点变成新主节点的从节点。命令：slaveof new master
* 3、通知应用方redis主节点变成了新主节点。 修改客户端调用的地址并重启客户端。
* 4、旧主节点变成新主节点的从节点。  命令：slaveof new master
* 哨兵机制存在的意义：
* 为了实现redis故障转移的自动化。自动发现，自动转移。不需要人工参与。
* 功能作用
* 监控(monitoring)：Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。
* 提醒(Notifation)：当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
* 自动故障转移(Automatic failover)：当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作， 它会将失效主服务器的其中一个从服务器升级为新的主服务器， 并让失效主服务器的其他从服务器改为复制新的主服务器； 当客户端试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。
* 之前有说到redis的主从分离架构以实现提高redis的高水平扩展能力，但是单单是这样的主从架构是存在着一些问题的：
* master（主）节点挂了会发生什么？
* master挂了，那么master下的从节点同样的处于不可用状态了。即master那一片都挂了。因为slave（从）节点不能再接收到新的数据
* slave节点挂了会怎么样？
* 如果是一个slave节点挂了，那么还有其他的slave节点对外提供服务，不至于所有的请求都发向后台的数据库中导致数据库压力突然增大。
* redis进程挂掉了会怎么样？
* 这就比较厉害了，进程挂掉了redis就没用了，有的时候redis进程挂掉了，而你有恰好没有给redis设置数据的持久化，redis可能会丢失较多的数据，除此外这台机器的这个进程的redis肯定也不能对外提供服务了。带来的问题就和前面两个一样的了，如果是master的进程，同master节点挂了；如果是slave的进程，同slave节点挂了。
* 解决办法：
* 哨兵，对zookeeper有了解的话会很容易理解这个机制，在这里简要的说明一下哨兵的工作。这里以zookeeper来做哨兵模式的讲解。首先zookeeper是一个分布式锁的服务，根据CPA理论，在分布式系统中，系统只能达到CAP理论中的两个要求，无法满足全部三个。
* 这个不难理解。C：数据的一致性，P：数据的分区容错性，A:数据的可用性。zookeeper实现的主要是CAP中的一致性（C）和分区容错性（P）不过你就算姑且认为zookeeper主要实现的是一致性（C）也ok，zookeeper在A方面的确不算强调的地方。同时由于zookeeper的非高可用性，zookeeper被认为不适合作为服务发现的系统。当然并非说zookeeper不能用，只是他的可用性不算好而已。zookeeper实现其强一致
* Redis Sentinel（哨兵集群）是一种高可用的redis部署方案。在集群中的redis-master服务挂掉时，无需人为干预，即可通过哨兵集群的自我调整，实现redis服务的持续可用。
* 哨兵集群的功能:
* 监控：持续监测集群中的master和slave是否可用。
* 通知：通知系统管理员，集群中的部分服务不可用。
* 故障切换：若master挂掉，哨兵集群会启动故障切换，选出其中一个slave作为新的master。使用redis的应用程序也会被告知使用新的master。
* 提供配置信息：哨兵集群为redis客户提供服务发现功能，客户端访问哨兵集群，会被告知当前的master地址。
* 哨兵集群的分布式特性：多个哨兵进程协同工作。
* 多个哨兵进程的好处：
* 故障需要多个进程同时确认，降低了发现故障时出错的概率。
* 部分哨兵进程挂掉后，集群同样可以维持工作，从而使系统更加健壮。
* 工作原理：
* 首先来了解哨兵集群中两个重要的基本概念：
* SDOWN(主观宕机)：哨兵实例的PING命令没有返回+PONG，-LOADING error，-MASTERDOWN error。
* ODOWN(客观宕机)：足够数量（大于等于sentinel.conf文件中的配置）的哨兵实例判断出发生了SDOWN。
* 发生客观宕机后开始故障切换。
* 哨兵集群可以自动发现各个哨兵实例以及redis master的slave

**005、Redis是否保证事务的一致性？**

　　　　网友遇到问题：

　　　　　　”redis设计之初是简单高效，所以说在事务操作时命令是不会出错的，出错的可能性就是程序的问题“   那这样的意思就是把锅抛给程序咯？

　　　　　　如果程序能保证百分之百不出错那么关系型数据库还要啥事务呢？

　　　　　　redis事务报错时仍会执行所有命令，这样怎么保证一致性呢？

　　　　　　或者说白了redis根本就不支持事务只是冠以事务的名号而已。以上纯属个人见解   又专业人士可以解释大家讨论。

　　　　网友理解：

　　　　　　1、Redis事务主要用于不间断执行多条命令，即是存在引发错误的命令。Redis先执行命令，命令执行成功后才会记录日志，所以出现错误时无法回滚。  支持完整的acid会让Redis变得复杂也可能导致性能较低。 此外，使用lua脚本也可以保证Redis不间断执行多条命令。

　　　　　　2、找到网上这个解释比较到位：单个 Redis 命令的执行是原子性的，但 Redis 没有在事务上增加任何维持原子性的机制，所以 Redis 事务的执行并不是原子性的。 事务可以理解为一个打包的批量执行脚本，但批量指令并非原子化的操作，中间某条指令的失败不会导致前面已做指令的回滚，也不会造成后续的指令不做。

 redis事务

  　　　　1、基本概念

  　　　　　　1）什么是redis的事务？

 　　　　　　　　 简单理解，可以认为redis事务是一些列redis命令的集合，并且有如下两个特点：

 　　　　　　　　 a）事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

 　　　　　　　　 b）事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

 　　　　　　 2）事务的性质ACID

  　　　　　　　　一般来说，事务有四个性质称为ACID，分别是原子性，一致性，隔离性和持久性。

 　　　　　　　　 a）原子性atomicity：redis事务保证事务中的命令要么全部执行要不全部不执行。有些文章认为redis事务对于执行错误不回滚违背了原子性，是偏颇的。

 　　　　　　　　 b）一致性consistency：redis事务可以保证命令失败的情况下得以回滚，数据能恢复到没有执行之前的样子，是保证一致性的，除非redis进程意外终结。

  　　　　　　　　c）隔离性Isolation：redis事务是严格遵守隔离性的，原因是redis是单进程单线程模式，可以保证命令执行过程中不会被其他客户端命令打断。

 　　　　　　　　 d）持久性Durability：redis事务是不保证持久性的，这是因为redis持久化策略中不管是RDB还是AOF都是异步执行的，不保证持久性是出于对性能的考虑。

　　　　　　  3）redis事务的错误

  　　　　　　　　使用事务时可能会遇上以下两种错误：

  　　　　　　　　　　a）入队错误：事务在执行 EXEC 之前，入队的命令可能会出错。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），

　　　　　　　　　　　　　　或者其他更严重的错误，比如内存不足（如果服务器使用 maxmemory 设置了最大内存限制的话）。

  　　　　　　　　　　b）执行错误：命令可能在 EXEC 调用之后失败。举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类。

 　　　　　　　　　　 注：第三种错误，redis进程终结，本文并没有讨论这种错误。

 　　　　2、redis事务的用法

  　　　　　　　redis事务是通过MULTI，EXEC，DISCARD和WATCH四个原语实现的。

  　　　　　　　MULTI命令用于开启一个事务，它总是返回OK。

  　　　　　　　MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

  　　　　　　　另一方面，通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务。

**006、Redis的持久化方式？**

　　　　两种持久化方式：

　　　　　　RDB和AOF  深入了解RDB和AOF的作用原理，剩下的就是根据实际情况来制定合适的策略了，再复杂一点，也就是定制一个高可用的，数据安全的策略了。

　　　　　　在RDB方式下，你有两种选择，一种是手动执行持久化数据命令来让redis进行一次数据快照，另一种则是根据你所配置的配置文件 的 策略，达到策略的某些条件时来自动持久化数据。

　　　　　　而手动执行持久化命令，你依然有两种选择，那就是save命令和bgsave命令。   save操作在Redis主线程中工作，因此会阻塞其他请求操作，应该避免使用。

　　　　　　bgSave则是调用Fork,产生子进程，父进程继续处理请求。子进程将数据写入临时文件，并在写完后，替换原有的.rdb文件。

　　　　　　Fork发生时，父子进程内存共享，所以为了不影响子进程做数据快照，在这期间修改的数据，将会被复制一份，而不进共享内存。

　　　　　　所以说，RDB所持久化的数据，是Fork发生时的数据。在这样的条件下进行持久化数据，如果因为某些情况宕机，则会丢失一段时间的数据。

　　　　　　如果你的实际情况对数据丢失没那么敏感，丢失的也可以从传统数据库中获取或者说丢失部分也无所谓，那么你可以选择RDB持久化方式。

　　　　　　再谈一下配置文件的策略，实际上它和bgsave命令持久化原理是相同的。

　　　　　　AOF持久化方式：

　　　　　　配置文件中的appendonly修改为yes。开启AOF持久化后，你所执行的每一条指令，都会被记录到appendonly.aof文件中。

　　　　　　但事实上，并不会立即将命令写入到硬盘文件中，而是写入到硬盘缓存，在接下来的策略中，配置多久来从硬盘缓存写入到硬盘文件。

　　　　　　所以在一定程度一定条件下，还是会有数据丢失，不过你可以大大减少数据损失。  这里是配置AOF持久化的策略。

　　　　　　redis默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

　　　　　　而no则是不主动进行同步操作，是默认30s一次。当然always一定是效率最低的，个人认为everysec就够用了，数据安全性能又高。

　　　Redis也允许我们同时使用两种方式，再重启redis后会从aof中恢复数据，因为aof比rdb数据损失小嘛。

　　　　　　深入理解Redis的两种持久化方式:  RDB每次进行快照方式会重新记录整个数据集的所有信息。RDB在恢复数据时更快，可以最大化redis性能，子进程对父进程无任何性能影响。

　　　　　　AOF有序的记录了redis的命令操作。意外情况下数据丢失甚少。他不断地对aof文件添加操作日志记录，你可能会说，这样的文件得多么庞大呀。

　　　　　　是的，的确会变得庞大，但redis会有优化的策略，比如你对一个key1键的操作，set key1 001 ,  set key1 002, set key1 003。那优化的结果就是将前两条去掉咯，

　　　　　　那具体优化的配置在配置文件中对应的是  https://images2015.cnblogs.com/blog/686162/201608/686162-20160809211516715-145676984.png

　　　　　　前者是指超过上一次aof重写aof文件大小的百分之多少，会再次优化，如果没有重写过，则以启动时为主。

　　　　　　后者是限制了允许重写的最小aof文件大小。bgrewriteaof命令是手动重写命令，会fork子进程，在临时文件中重建数据库状态，对原aof无任何影响，

　　　　　　当重建旧的状态后，也会把fork发生后的一段时间内的数据一并追加到临时文件，最后替换原有aof文件，新的命令继续向新的aof文件中追加。

　　Redis数据库简介：

　　　　　　Redis是一种高级key-value数据库。它跟memcached类似，不过数据可以持久化，而且支持的数据类型很丰富。

　　　　　　有字符串，链表，集 合和有序集合。支持在服务器端计算集合的并，交和补集(difference)等，还支持多种排序功能。

　　　　　　所以Redis也可以被看成是一个数据结构服务 器。   Redis的所有数据都是保存在内存中，然后不定期的通过异步方式保存到磁盘上(这称为“半持久化模式”)；

　　　　　　也可以把每一次数据变化都写入到一个append only file(aof)里面(这称为“全持久化模式”)。

　　Redis数据库简介：

　　　　　　默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

　　　　　　默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

　　　　　　由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁盘上，

　　　　　　当redis重启后，可以从磁盘中恢复数据。redis提供两种方式进行持久化，一种是RDB持久化（原理是将Reids在内存中的数据库记录定时dump到磁盘上的RDB持久化），

　　　　　　另外一种是AOF（append only file）持久化（原理是将Reids的操作日志以追加的方式写入文件）。

　　　　　　那么这两种持久化方式有什么区别呢，改如何选择呢？网上看了大多数都是介绍这两种方式怎么配置，怎么使用，就是没有介绍二者的区别，在什么应用场景下使用。

**007、二者的区别？RDB存在哪些优势呢？**

 　　　　　　1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。

　　　　　　　　比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。

　　　　　　　　通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

 　　　　　　2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

 　　　　　　3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

 　　　　　　4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

 　　　　RDB又存在哪些劣势呢？

 　　　　　　1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

 　　　　　　2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

 　　　　AOF的优势有哪些呢？

 　　　　　　1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。

　　　　　　　　事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。

　　　　　　　　而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

 　　　　　　2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。

　　　　　　　　然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

 　　　　　　3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。

　　　　　　　　因此在进行rewrite切换时可以更好的保证数据安全性。

　　　　　　 4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

 　　　　AOF的劣势有哪些呢？

 　　　　　　1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

 　　　　　　2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

 　　　　　　二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。

　　　　 　　 rdb这个就更有些 eventually consistent的意思了。

**008、RDB与AOF常用配置？**

 　　　　RDB持久化配置：

 　　　　Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

 　　　　　　save 900 1              #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

 　　　　　　save 300 10            #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

 　　　　　　save 60 10000        #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

 　　　　AOF持久化配置

 　　　　在Redis的配置文件中存在三种同步方式，它们分别是：

 　　　　　　appendfsync always     #每次有数据修改发生时都会写入AOF文件。

 　　　　　　appendfsync everysec  #每秒钟同步一次，该策略为AOF的缺省策略。

 　　　　　　appendfsync no          #从不同步。高效但是数据不会被持久化。

同理celery也是只能用3.1.25 不能用4版本以上的，django1.11才可以用4以上的

**MongoDB面试题**

**001、MongoDB中对多条记录做更新操作命令是什么？**

Mongodb更新有两个命令：update、save。

**1.1update命令：**

update命令格式：

db.collection.update(criteria,objNew,upsert,multi)

参数说明：

criteria：查询条件

objNew：update对象和一些更新操作符

upsert：如果不存在update的记录，是否插入objNew这个新的文档，true为插入，默认为false，不插入。

multi：默认是false，只更新找到的第一条记录。如果为true，把按条件查询出来的记录全部更新。

1.2save命令

Mongodb另一个更新命令是save，格式如下：

db.collection.save(obj)

obj代表需要更新的对象，如果集合内部已经存在一个和obj相同的"\_id"的记录，Mongodb会把obj对象替换集合内已存在的记录，如果不存在，则会插入obj对象。

这条命令比较简单，示例就省略了。

**二、数据更新操作符：**

1.$inc：用法：{$inc:{field:value}}；作用：对一个数字字段的某个field增加value

2.$set：用法：{$set:{field:value}}；作用：把文档中某个字段field的值设为value

3.$unset：用法：{$unset:{field:1}}；作用：删除某个字段field

4.$push：用法：{$push:{field:value}}；作用：把value追加到field里。注：field只能是数组类型，如果field不存在，会自动插入一个数组类型

5.pushAll：用法：{$pushAll:{field:value\_array}}：作用：用法同$push一样，只是$pushAll可以一次追加多个值到一个数组字段内。

6.$addToSet：用法：{$addToSet:{field:value}}；作用：加一个值到数组内，而且只有当这个值在数组中不存在时才增加。

7.$pop：用法：删除数组内第一个值：{$pop:{field:-1}}、删除数组内最后一个值：{$pop:{field:1}}；作用：用于删除数组内的一个值

8.$pull：用法：{$pull:{field:\_value}}；作用：从数组field内删除一个等于\_value的值

9.$pullAll：用法：{$pullAll:value\_array}；作用：用法同$pull一样，可以一次性删除数组内的多个值。

10.$rename：用法：{$rename:{old\_field\_name:new\_field\_name}}；作用：对字段进行重命名

**002、MongoDB如何才会拓展到多个shard里？**

mongodb 分片是基于区域(range)的.所以一个集合(collection)中的所有的对象都被存放到一个块(chunk)中.只有当存在多余一个块的时候,才会有多个分片获取数据的选项.现在,每个默认块的大小是 64mb,所以你需要至少 64 mb 空间才可以实施一个迁移.

**003、为什么要使用Nosql？**

[关系型数据库](https://cloud.tencent.com/product/cdb-overview?from=10680)采用的结构化的数据，NoSQL采用的是键值对的方式存储数据。

**004、mongo使用场景？**

在处理非结构化/半结构化的大数据时； 在水平方向上进行扩展时； 随时应对动态增加的数据项时可以优先考虑使用NoSQL数据库。 高并发解决方案时；

**005、以下特点使得**[**MongoDB**](https://cloud.tencent.com/product/mongodb?from=10680)**成为最好的NoSQL数据库？**

面向文件的、高性能、 高可用性、 易扩展性、 丰富的查询语言、 完全索引、

**006、mongoDB的基本结构？**

MongoDB的最基本的数据单元，叫document，类似于关系式数据库中的行 row。一系列documents，组成了一个collection，相当于关系式数据库中的table。当一个 collection 数据量太大时，可以把该collection按documents切分，分成多个数据块，每个数据块叫做一个chunk，多个chunks聚集在一起，组成了一个shard。

Sharding 的意义，不仅保障了数据库的扩容（scalability），同时也保障了系统的负载均衡（load balance）。

**007、MongoDB支持存储过程吗？如果支持的话，怎么用？**

MongoDB支持存储过程，它是JavaScript写的，保存在db.system.js表中。

**008、如何执行事务/加锁?**

MongoDB没有使用传统的锁或者复杂的带回滚的事务，因为它设计的宗旨是轻量，快速以及可预计的高性能。可以把它类比成[MySQL](https://cloud.tencent.com/product/cdb?from=10680) MylSAM的自动提交模式。通过精简对事务的支持，性能得到了提升，特别是在一个可能会穿过多个服务器的系统里。

**009、什么是“片键”？**

片键是拆分集合的依据，管理员设置的“片键”将数据分摊到自己管理的mongodb集群，数据和片的对应关系以及相应的配置信息保存在”config服务器”上。

**010、什么是master或primary?**

它是当前备份集群(replica set)中负责处理所有写入操作的主要节点/成员。在一个备份集群中，当失效备援(failover)事件发生时，一个另外的成员会变成primary。

主从复制模式：master slave 副本集 ：primary（活跃） secondary（备份）

**011、什么是secondary或slave?**

Seconday从当前的primary上复制相应的操作。它是通过跟踪复制oplog(local.oplog.rs)做到的。

**012、数据库的整体结构？**

键值对–》文档–》集合–》数据库

**013、mongodb的结构介绍？**

数据库中存储的对象设计bson，一种类似json的二进制文件，由键值对组成

**014、为什么mongodb的数据文件那么庞大？**

mongodb会积极的预分配预留空间，防止文件系统碎片

**015、名字空间（namespace）是什么？**

在collection中，数据库名+集合名叫做名字空间。也就是一个集合的完整名；

**016、数据在什么时候才会扩展到多个分片（shard）里？**

mongodb分片是基于区域的，所以一个集合的所有对象都放置在同一个块中，只有当存在多余一个块的时候，才会有多个分片获取数据的选项；

**017、当我试图更新一个正在被迁移的块（chunk）上的文档时会发生什么？**

会立即更新旧的分片，然后更改才会在所有权转移前复制到新的分片上

**018、能否使用日志特征进行安全备份？**

是的。

**019、更新操作立刻fsync到磁盘？**

一般磁盘的写操作都是延迟执行的；

**020、如果用户移除对象的属性，该属性是否从存储层中删除？**

是的，用户移除属性然后对象会重新保存（re-save()）。

**021、分析器在MongoDB中的作用是什么?**

分析器就是explain 显示每次操作性能特点的数据库分析器。通过分析器可能查找比预期慢的操作

**022、什么是MongoDB？**

MongoDB是一个文档数据库，提供好的性能，领先的非关系型数据库。采用BSON存储文档数据。

BSON（）是一种类json的一种二进制形式的存储格式，简称Binary JSON.

相对于json多了date类型和二进制数组。

**023、MongoDB的优势有哪些？**

面向文档的存储：

以 JSON 格式的文档保存数据。

任何属性都可以建立索引。

复制以及高可扩展性。

自动分片。

丰富的查询功能。

快速的即时更新。

**024、什么是数据库？**

数据库可以看成是一个电子化的文件柜,用户可以对文件中的数据运行新增、检索、更新、删除等操作。数据库是一个所有集合的容器，在文件系统中每一个数据库都有一个相关的物理文件。

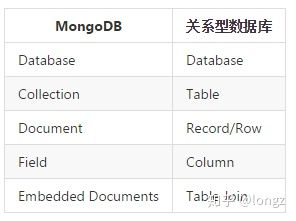
**025、什么是集合(表)？**

集合就是一组 MongoDB 文档。它相当于关系型数据库（RDBMS）中的表这种概念。集合位于单独的一个数据库中。一个集合内的多个文档可以有多个不同的字段。一般来说，集合中的文档都有着相同或相关的目的。

**026、什么是文档(记录)？**

文档由一组key value组成。文档是动态模式,这意味着同一集合里的文档不需要有相同的字段和结构。在关系型数据库中table中的每一条记录相当于MongoDB中的一个文。

**027、 MongoDB和关系型数据库术语对比图？**



**028、什么是非关系型数据库？**

非关系型数据库的显著特点是不使用SQL作为查询语言，数据存储不需要特定的表格模式。

**029、为什么用MOngoDB？**

架构简单，没有复杂的连接，深度查询能力、MongoDB支持动态查询。容易调试，容易扩展，不需要转化/映射应用对象到数据库对象，使用内部内存作为存储工作区,以便更快的存取数据。

**030、在哪些场景使用MongoDB？**

大数据、内容管理系统、移动端Apps、数据管理。

**031、MongoDB中的命名空间是什么意思?**

mongodb存储bson对象在丛集(collection)中.数据库名字和丛集名字以句点连结起来叫做名字空间(namespace).

　　一个集合命名空间又有多个数据域(extent)，集合命名空间里存储着集合的元数据，比如集合名称，集合的

第一个数据域和最后一个数据域的位置等等。而一个数据域由若干条文档(document)组成，每个数据域都有一个

头部，记录着第一条文档和最后一条文档的为知，以及该数据域的一些元数据。extent之间，document之间通过

双向链表连接。

索引的存储数据结构是B树，索引命名空间存储着对B树的根节点的指针。

**032、monogodb 中的分片什么意思？**

　　分片是将数据水平切分到不同的物理节点。当应用数据越来越大的时候，数据量也会越来越大。当数据量增长

时，单台机器有可能无法存储数据或可接受的读取写入吞吐量。利用分片技术可以添加更多的机器来应对数据量增加

以及读写操作的要求。

**033、为什么要在MongoDB中使用分析器？**

mongodb中包括了一个可以显示数据库中每个操作性能特点的数据库分析器.通过这个分析器你可以找到比预期慢的查询(或写操作);利用这一信息,比如,可以确定是否需要添加索引.

**034、MongoDB支持主键外键关系吗？**

默认MongoDB不支持主键和外键关系。 用Mongodb本身的API需要硬编码才能实现外键关联，不够直观且难度较大

**035、MongoDB支持哪些数据类型？**

String

Integer

Double

Boolean

Object

Object ID

Arrays

Min/Max Keys

Datetime

Code

Regular Expression等

**036、为什么要在MongoDB中用"Code"数据类型？**

"Code"类型用于在文档中存储 JavaScript 代码；

**037、为什么要在MongoDB中用"Regular Expression"数据类型？**

"Regular Expression"类型用于在文档中存储正则表达式；

**038、为什么在MongoDB中使用"Object ID"数据类型？**

"ObjectID"数据类型用于存储文档id；

**039、"ObjectID"有哪些部分组成？**

一共有四部分组成:时间戳、客户端ID、客户进程ID、三个字节的增量计数器；

**040、在MongoDb中什么是索引？**

索引用于高效的执行查询,没有索引的MongoDB将扫描整个集合中的所有文档,这种扫描效率很低,需要处理大量的数据.

索引是一种特殊的数据结构,将一小块数据集合保存为容易遍历的形式.索引能够存储某种特殊字段或字段集的值,并按照索引指定的方式将字段值进行排序.

**041、如何添加索引？**

使用db.collection.createIndex()在集合中创建一个索引

**042、如何查询集合中的文档？**

db.collectionName.find({key:value})

**043、用什么方法可以格式化输出结果？**

db.collectionName.find().pretty()

**044、如何使用"AND"或"OR"条件循环查询集合中的文档？**

db.mycol.find(

{

$or: [

{key1: value1}, {key2:value2}

]

}

).pretty()

**045、更新数据？**

db.collectionName.update({key:value},{$set:{newkey:newValue}})

**046、如何删除文档？**

db.collectionName.remove({key:value})

**047、在MongoDB中如何排序？**

并使用 1 和 -1 来指定排序方式，其中 1 表示升序，而 -1 表示降序。

db.connectionName.find({key:value}).sort({columnName:1})

**048、什么是聚合？**

聚合操作能够处理数据记录并返回计算结果。聚合操作能将多个文档中的值组合起来，对成组数据执行各种操作，返回单一的结果。它相当于 SQL 中的 count(\*) 组合 group by。对于 MongoDB 中的聚合操作，应该使用aggregate()方法。

db.COLLECTION\_NAME.aggregate(AGGREGATE\_OPERATION)

**049、在MongoDB中什么是副本集（避免单点故障）？**

在MongoDB中副本集由一组MongoDB实例组成，包括一个主节点多个次节点，MongoDB客户端的所有数据都写入主节点(Primary),副节点从主节点同步写入数据，以保持所有复制集内存储相同的数据，提高数据可用性。

**050、什么是NoSQL数据库？NoSQL和RDBMS有什么区别？在哪些情况下使用和不使用NoSQL数据库？**

　NoSQL是非关系型数据库，NoSQL = Not Only SQL。

关系型数据库采用的结构化的数据，NoSQL采用的是键值对的方式存储数据。

在处理非结构化/半结构化的大数据时；在水平方向上进行扩展时；随时应对动态增加的数据项时可以优先考虑

使用NoSQL数据库。

在考虑数据库的成熟度；支持；分析和商业智能；管理及专业性等问题时，应优先考虑关系型数据库。

**051、MongoDB支持存储过程吗？如果支持的话，怎么用？**

MongoDB支持存储过程，它是javascript写的，保存在db.system.js表中。

**052、如何理解MongoDB中的GridFS机制，MongoDB为何使用GridFS来存储文件？**

GridFS是一种将大型文件存储在MongoDB中的文件规范。使用GridFS可以将大文件分隔成多个小文档存放，这样我们能够有效的保存大文档，而且解决了BSON对象有限制的问题。

**053、为什么MongoDB的数据文件很大？**

MongoDB采用的预分配空间的方式来防止文件碎片。

**054、当更新一个正在被迁移的块（Chunk）上的文档时会发生什么？**

更新操作会立即发生在旧的块（Chunk）上，然后更改才会在所有权转移前复制到新的分片上。

**055、MongoDB在A:{B,C}上建立索引，查询A:{B,C}和A:{C,B}都会使用索引吗？**

不会，只会在A:{B,C}上使用索引。

**056、mongodb成为最好nosql数据库的原因是什么?**

面向文件的、 高性能、高可用性、 易扩展性、 丰富的查询语言。

**057、如果用户移除对象的属性,该属性是否从存储层中删除?**

是的,用户移除属性然后对象会重新保存(re-save()).

**058、允许空值null吗?**

对于对象成员而言,是的.然而用户不能够添加空值(null)到数据库丛集(collection)因为空值不是对象.然而用户能够添加空对象{}.

**059、更新操作立刻fsync到磁盘?**

不会,磁盘写操作默认是延迟执行的.写操作可能在两三秒(默认在60秒内)后到达磁盘.例如,如果一秒内数据库收到一千个对一个对象递增的操作,仅刷新磁盘一次.

**060、如何执行事务/加锁?**

　　mongodb没有使用传统的锁或者复杂的带回滚的事务,因为它设计的宗旨是轻量,快速以及可预计的高性能.可以把它类比成mysql mylsam的自动提交模式.通过精简对事务的支持,性能得到了提升,特别是在一个可能会穿过多个服务器的系统里.

**061、启用备份故障恢复需要多久?**

　　从备份数据库声明主数据库宕机到选出一个备份数据库作为新的主数据库将花费10到30秒时间.这期间在主数据库上的操作将会失败–包括写入和强一致性读取(strong consistent read)操作.然而,你还能在第二数据库上执行最终一致性查询(eventually consistent query)(在slaveok模式下),即使在这段时间里.

**062、什么是master或primary?**

　　它是当前备份集群(replica set)中负责处理所有写入操作的主要节点/成员.在一个备份集群中,当失效备援(failover)事件发生时,一个另外的成员会变成primary.

**063、我应该启动一个集群分片(sharded)还是一个非集群分片的 mongodb 环境?**

　　(数据量大用集群分片,数据量小用非集群)

　　为开发便捷起见,我们建议以非集群分片(unsharded)方式开始一个 mongodb 环境,除非一台服务器不足以存放你的初始数据集.从非集群分片升级到集群分片(sharding)是无缝的,所以在你的数据集还不是很大的时候没必要考虑集群分片(sharding).

**064、分片(sharding)和复制(replication)是怎样工作的?**

　　每一个分片(shard)是一个分区数据的逻辑集合.分片可能由单一服务器或者集群组成,我们推荐为每一个分片(shard)使用集群.

**065、数据在什么时候才会扩展到多个分片(shard)里?**

　　mongodb 分片是基于区域(range)的.所以一个集合(collection)中的所有的对象都被存放到一个块(chunk)中.只有当存在多余一个块的时候,才会有多个分片获取数据的选项.现在,每个默认块的大小是 64mb,所以你需要至少 64 mb 空间才可以实施一个迁移.

**066、当我试图更新一个正在被迁移的块(chunk)上的文档时会发生什么?**

　　更新操作会立即发生在旧的分片(shard)上,然后更改才会在所有权转移(ownership transfers)前复制到新的分片上.

**067、如果在一个分片(shard)停止或者很慢的时候,我发起一个查询会怎样?**

　　如果一个分片(shard)停止了,除非查询设置了“partial”选项,否则查询会返回一个错误.如果一个分片(shard)响应很慢,mongodb则会等待它的响应.

**068、可以把movechunk目录里的旧文件删除吗?**

　　没问题,这些文件是在分片(shard)进行均衡操作(balancing)的时候产生的临时文件.一旦这些操作已经完成,相关的临时文件也应该被删除掉.但目前清理工作是需要手动的,所以请小心地考虑再释放这些文件的空间.

**069、如果块移动操作(movechunk)失败了,我需要手动清除部分转移的文档吗?**

 　　不需要,移动操作是一致(consistent)并且是确定性的(deterministic);一次失败后,移动操作会不断重试;当完成后,数据只会出现在新的分片里(shard).

**070、mongodb是否支持事务**

　　MongoDB 4.0的新特性——事务（Transactions）：MongoDB 是不支持事务的，因此开发者在需要用到事务的时候，不得不借用其他工具，在业务代码层面去弥补数据库的不足。

　　事务和会话(Sessions)关联，一个会话同一时刻只能开启一个事务操作，当一个会话断开，这个会话中的事务也会结束。

**129、简述 vitualenv 及应用场景?**

'vitualenv'是一个独立的python虚拟环境。

如：当前项目依赖的是一个版本，但是另一个项目依赖的是另一个版本，这样就会造成依赖冲突，而virtualenv就是解决这种情况的，virtualenv通过创建一个虚拟化的python运行环境，将我们所需的依赖安装进去的，不同项目之间相互不干扰。

**130、简述 pipreqs 及应用场景？**

可以通过对项目目录扫描，自动发现使用了那些类库，并且自动生成依赖清单。pipreqs ./ 生成requirements.txt

**131、在Python中使用过什么代码检查工具？**

1）PyFlakes：静态检查Python代码逻辑错误的工具。

2）Pep8： 静态检查PEP8编码风格的工具。

3）NedBatchelder’s McCabe script：静态分析Python代码复杂度的工具。

Python代码分析工具：PyChecker、Pylint

**132、B Tree和B+ Tree的区别？**

1.B树中同一键值不会出现多次，并且有可能出现在叶结点，也有可能出现在非叶结点中。

而B+树的键一定会出现在叶结点中，并有可能在非叶结点中重复出现，以维持B+树的平衡。

2.因为B树键位置不定，且在整个树结构中只出现一次，

**133、请列举你熟悉的设计模式？**

工厂模式/单例模式等。

**134、有没有刷过leetcode？**

leetcode是个题库，里面有多很编程题目，可以在线编译运行。

https://leetcode-cn.com/problemset/all/

**135、列举熟悉的的Linux命令。**

1创建目录

mkdir /data

cd /

mkdir data

2查看目录

ls

ls -l 显示详细信息

**136、公司线上服务器是什么系统？**

Linux/Centos，ubuntu

**137、解释 PV、UV 的含义？**

PV访问量（Page View），即页面访问量，每打开一次页面PV计数+1，刷新页面也是。

UV访客量（Unique Visitor）指独立访客访问数，一台电脑终端为一个访客。

**138、解释 QPS的含义？**

'QPS(Query Per Second)'每秒查询率，是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。

**140、supervisor的作用？**

# Supervisor：

是一款基于Python的进程管理工具，可以很方便的管理服务器上部署的应用程序。

是C/S模型的程序，其服务端是supervisord服务,客户端是supervisorctl命令

# 主要功能:

1 启动、重启、关闭包括但不限于python进程。

2 查看进程的运行状态。

3 批量维护多个进程。

**142、简述SSH的整个过程？**

SSH 为 'Secure Shell' 的缩写，是建立在应用层基础上的安全协议。SSH 是目前较可靠，为远程登录会话和其他网络服务提供的安全性协议。利用SSH 协议可以有效防止远程管理过程中的信息泄露问题。

**142、让你开发微信二维码电脑端登录，你要怎么实现？**

python 网站实现扫描二维码关注微信公众号，实现自动登陆。

微信公众平台接口测试帐号：https://mp.weixin.qq.com/debug/cgi-bin/sandbox?t=sandbox/login

1，微信提供生成带参数的二维码的接口，参数就是唯一值（场景值，我用的时间戳）：参考 微信文档 生成带参数的二维码；

2，网站调用微信系统，获取生成的二维码图片：

获取到 ret\_url 返回给前端 直接显示 二维码图片，用 scene\_id 创建一个websocket 连接；

3 、微信公众号配置：设置好微信 回调的地址 httl.../wx/ 用户扫码后会请求此地址；

4、处理推送事件 httl.../wx/；

**143、有问题都去那些找解决方案？**

起初是百度，发现搜到的答案不精准，净广告，转战谷歌，但墙了；捣鼓怎么FQ，还会去知乎、stackoverfloow、必应、思否(segmentfault)

**144、是否有关注什么技术类的公众号？**

python之禅(主要专注Python相关知识，作者：刘志军)；

码农翻身(主要是Java的，但不光是java，涵盖面很广，作者：刘欣)；

实验楼(在线练项目)；

and so on；

**145、最近在研究什么新技术？**

Numpy；pandas(金融量化分析、聚宽)；百度AI；图灵API；智能玩具

**云服务模块**

**01、你使用过哪些云服务技术？（**基础组件、隔离策略、备份与恢复策略、容器化、资源管理**）**

云计算的核心技术有哪些？

云计算是一种以数据和处理能力为中心的密集型计算模式，它融合了多项ICT技术，是传统技术“平滑演进”的产物。其中以虚拟化技术、分布式数据存储技术、编程模型、大规模数据管理技术、分布式资源管理、信息安全、云计算平台管理技术、绿色节能技术最为关键。

1、虚拟化技术：

　虚拟化是云计算最重要的核心技术之一，它为云计算服务提供基础架构层面的支撑，是ICT服务快速走向云计算的最主要驱动力。可以说，没有虚拟化技术也就没有云计算服务的落地与成功。随着云计算应用的持续升温，业内对虚拟化技术的重视也提到了一个新的高度。与此同时，我们的调查发现，很多人对云计算和虚拟化的认识都存在误区，认为云计算就是虚拟化。事实上并非如此，虚拟化是云计算的重要组成部分但不是全部。

　　从技术上讲，虚拟化是一种在软件中仿真计算机硬件，以虚拟资源为用户提供服务的计算形式。旨在合理调配计算机资源，使其更高效地提供服务。它把应用系统各硬件间的物理划分打破，从而实现架构的动态化，实现物理资源的集中管理和使用。虚拟化的最大好处是增强系统的弹性和灵活性，降低成本、改进服务、提高资源利用效率。

　　从表现形式上看，虚拟化又分两种应用模式。一是将一台性能强大的服务器虚拟成多个独立的小服务器，服务不同的用户。二是将多个服务器虚拟成一个强大的服务器，完成特定的功能。这两种模式的核心都是统一管理，动态分配资源，提高资源利用率。在云计算中，这两种模式都有比较多的应用。

2、分布式数据存储技术：

　　云计算的另一大优势就是能够快速、高效地处理海量数据。在数据爆炸的今天，这一点至关重要。为了保证数据的高可靠性，云计算通常会采用分布式存储技术，将数据存储在不同的物理设备中。这种模式不仅摆脱了硬件设备的限制，同时扩展性更好，能够快速响应用户需求的变化。

　　分布式存储与传统的网络存储并不完全一样，传统的网络存储系统采用集中的存储服务器存放所有数据，存储服务器成为系统性能的瓶颈，不能满足大规模存储应用的需要。分布式网络存储系统采用可扩展的系统结构，利用多台存储服务器分担存储负荷，利用位置服务器定位存储信息，它不但提高了系统的可靠性、可用性和存取效率，还易于扩展。

　　在当前的云计算领域，Google的GFS和Hadoop开发的开源系统HDFS是比较流行的两种云计算分布式存储系统。

　　GFS（Google File System）技术：谷歌的非开源的GFS（GoogleFile System）云计算平台满足大量用户的需求，并行地为大量用户提供服务。使得云计算的数据存储技术具有了高吞吐率和高传输率的特点。

　　HDFS（Hadoop Distributed File System）技术：大部分ICT厂商，包括Yahoo、Intel的“云”计划采用的都是HDFS的数据存储技术。未来的发展将集中在超大规模的数据存储、数据加密和安全性保证、以及继续提高I/O速率等方面。

3、编程模式：

　　从本质上讲，云计算是一个多用户、多任务、支持并发处理的系统。高效、简捷、快速是其核心理念，它旨在通过网络把强大的服务器计算资源方便地分发到终端用户手中，同时保证低成本和良好的用户体验。在这个过程中，编程模式的选择至关重要。云计算项目中分布式并行编程模式将被广泛采用。

　　分布式并行编程模式创立的初衷是更高效地利用软、硬件资源，让用户更快速、更简单地使用应用或服务。在分布式并行编程模式中，后台复杂的任务处理和资源调度对于用户来说是透明的，这样用户体验能够大大提升。MapReduce是当前云计算主流并行编程模式之一。MapReduce模式将任务自动分成多个子任务，通过Map和Reduce两步实现任务在大规模计算节点中的高度与分配。

　　MapReduce是Google开发的java、Python、C++编程模型，主要用于大规模数据集（大于1TB）的并行运算。MapReduce模式的思想是将要执行的问题分解成Map（映射）和Reduce（化简）的方式，先通过Map程序将数据切割成不相关的区块，分配（调度）给大量计算机处理，达到分布式运算的效果，再通过Reduce程序将结果汇整输出。

4、大规模数据管理：

　　处理海量数据是云计算的一大优势。那么如何处理则涉及到很多层面的东西，因此高效的数据处理技术也是云计算不可或缺的核心技术之一。对于云计算来说，数据管理面临巨大的挑战。云计算不仅要保证数据的存储和访问，还要能够对海量数据进行特定的检索和分析。由于云计算需要对海量的分布式数据进行处理、分析，因此，数据管理技术必需能够高效的管理大量的数据。

　　Google的BT（BigTable）数据管理技术和Hadoop团队开发的开源数据管理模块HBase是业界比较典型的大规模数据管理技术。

　　BT（BigTable）数据管理技术：BigTable是非关系的数据库，是一个分布式的、持久化存储的多维度排序Map.BigTable建立在 GFS,Scheduler, Lock Service和MapReduce之上，与传统的关系数据库不同，它把所有数据都作为对象来处理，形成一个巨大的表格，用来分布存储大规模结构化数据。 Bigtable的设计目的是可靠的处理PB级别的数据，并且能够部署到上千台机器上。

　　开源数据管理模块HBase:HBase是Apache的Hadoop项目的子项目，定位于分布式、面向列的开源数据库。HBase不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。另一个不同的是HBase基于列的而不是基于行的模式。作为高可靠性分布式存储系统，HBase在性能和可伸缩方面都有比较好的表现。利用HBase技术可在廉价PC Server上搭建起大规模结构化存储集群。

5、分布式资源管理：

　　云计算采用了分布式存储技术存储数据，那么自然要引入分布式资源管理技术。在多节点的并发执行环境中，各个节点的状态需要同步，并且在单个节点出现故障时，系统需要有效的机制保证其它节点不受影响。而分布式资源管理系统恰是这样的技术，它是保证系统状态的关键。

　　另外，云计算系统所处理的资源往往非常庞大，少则几百台服务器，多则上万台，同时可能跨跃多个地域。且云平台中运行的应用也是数以千计，如何有效地管理这批资源，保证它们正常提供服务，需要强大的技术支撑。因此，分布式资源管理技术的重要性可想而知。

　　全球各大云计算方案/服务提供商们都在积极开展相关技术的研发工作。其中Google内部使用的Borg技术很受业内称道。另外，微软、IBM、Oracle/Sun等云计算巨头都有相应解决方案提出。

6、信息安全：

　　调查数据表明，安全已经成为阻碍云计算发展的最主要原因之一。数据显示，32%已经使用云计算的组织和45%尚未使用云计算的组织的ICT管理将云安全作为进一步部署云的最大障碍。因此，要想保证云计算能够长期稳定、快速发展，安全是首要需要解决的问题。

　　事实上，云计算安全也不是新问题，传统互联网存在同样的问题。只是云计算出现以后，安全问题变得更加突出。在云计算体系中，安全涉及到很多层面，包括网络安全、服务器安全、软件安全、系统安全等等。因此，有分析师认为，云安全产业的发展，将把传统安全技术提到一个新的阶段。

　　现在，不管是软件安全厂商还是硬件安全厂商都在积极研发云计算安全产品和方案。包括传统杀毒软件厂商、软硬防火墙厂商、IDS/IPS厂商在内的各个层面的安全供应商都已加入到云安全领域。相信在不久的将来，云安全问题将得到很好的解决.

7、云计算平台管理：

　　云计算资源规模庞大，服务器数量众多并分布在不同的地点，同时运行着数百种应用，如何有效地管理这些服务器，保证整个系统提供不间断的服务是巨大的挑战。云计算系统的平台管理技术，需要具有高效调配大量服务器资源，使其更好协同工作的能力。其中，方便地部署和开通新业务、快速发现并且恢复系统故障、通过自动化、智能化手段实现大规模系统可靠的运营是云计算平台管理技术的关键。

　　对于提供者而言，云计算可以有三种部署模式，即公共云、私有云和混合云。三种模式对平台管理的要求大不相同。对于用户而言，由于企业对于ICT资源共享的控制、对系统效率的要求以及ICT成本投入预算不尽相同，企业所需要的云计算系统规模及可管理性能也大不相同。因此，云计算平台管理方案要更多地考虑到定制化需求，能够满足不同场景的应用需求。

　　包括Google、IBM、微软、Oracle/Sun等在内的许多厂商都有云计算平台管理方案推出。这些方案能够帮助企业实现基础架构整合、实现企业硬件资源和软件资源的统一管理、统一分配、统一部署、统一监控和统一备份，打破应用对资源的独占，让企业云计算平台价值得以充分发挥。

8、绿色节能技术：

　　节能环保是全球整个时代的大主题。云计算也以低成本、高效率着称。云计算具有巨大的规模经济效益，在提高资源利用效率的同时，节省了大量能源。绿色节能技术已经成为云计算必不可少的技术，未来越来越多的节能技术还会被引入云计算中来。

　　Carbon Disclosure Project（碳排放披露项目，简称CDP）近日发布了一项有关云计算有助于减少碳排放的研究报告。报告指出，迁移至云的美国公司每年就可以减少碳排放8570万吨，这相当于2亿桶石油所排放出的碳总量。

　　总之，云计算服务提供商们需要持续改善技术，让云计算更绿色。

**02、常用的云技术有哪些，请列举4个?**

* ECS实例
* OSS
* RDS
* CDN