## MySQL 高级

### 视图

1.V\_good怎么生成的呢？然后连接sql语句?

视图的作用对基本表的引用

2.不管基本表怎么改，V表都是应用基本表数据

3.虚拟表作用在与python交互的时候通过视图来连接解决基本表的数据的改变如

字段的修改.

4.虚拟表重点！！！：不断对基本表的引用，基本表变虚拟表对应修改.

### 事务---innodb

**重要:引擎是innodb的才可以使用事务**

**2.python交互修改数据commit**

1.begin---开启事务

2.commit

3.rockback 回滚数据，放弃缓存中的数据回滚到原始数据

4.python操作数据库，系统默认开启事务

5.因为事务这个所以在python交互的时候修改数据就需要connect.commit

原子性，一致性:不commit原始数据库数据就不会被修改 or rollback

隔离性:对一个数据同时做修改的时候，一边开启后修改，另外一边也开启修改但是修改该数据的时候会堵塞不动。

持久性:数据一旦修改，所做的修改会永久保存到数据库.

### 索引

**实际数据占用空间**

commit

1.set profiling=1;

对数据库数据分组编号----索引

1.创建索引

2.索引随数据表数据添加然更新自己的索引文件

3.树结构好处？查询数据更快速分组存储数据

账户

use mysql---user

# Python\_提高

## Python\_提高1

### 1.GIL

**1. 什么是GIL锁?**

cpython解释器中---保证同一时刻只有一个线程使用CPU

并发

并行---多核CPU

1.多线程在python中并没有真正意义上把多个CPU使用起来,假装的多线程.

原因:(代码编译—>解释器--->CPU运行的流程) 使用了cpython解释器

**2.为什么还有多线程? 什么时候单线程比多线程快?**

2.1 科学计算—2\*100000次方 CPU一直被占用在计算是单线程快！

2.2 I/O操作(文件操作)堵塞等待操作 此时多线程快！---数据发收的间隙时间利用

**快在哪里？**1.线程1传输的时候占用CPU,等待数据回来的空隙 线程2就可以使用这段空闲传输数据

**3.解决办法GIL锁办法**: 1.更换python解释器 2.使用c语言 3.使用多进程

**注意:**

多任务---真正意义的多任务必须是多核—CPU

### 2.深浅拷贝

**1.深浅拷贝**

1.Import copy 浅拷贝:copy.copy 深拷贝:copy.deepcopy

2.id()----查看内存地址值

3.都会自己开辟新的内存空间来存储数据

**2.嵌套拷贝**

**是地址的引用而不是开辟新的数据空间**

-----**copy**.**copy**仅仅是对数据的**引用**并没有真正的数据空间的独立

-----**copy.deepcopy** **开辟新的空间**存储数据从而保证数据的独立空间

区别:是否开辟了新的空间存储数据(id()—查看内存地址)

切片拷贝----浅拷贝

字典拷贝----浅拷贝 a = 字典 b = a.copy()

**引用优点：**

1. 减少内存空间占用

2. 提升速度---因为开辟新空间需要时间，而引用就直接使用，所以会提升使用数据的速度

缺点：不能保证数据的独立性

**3.不同数据类型拷贝**

1.可变数据类型和不可变数据类型

2.不可变数据类型(like 元组)----深浅拷贝都不会开辟新的数据空间

注意:

浅拷贝 不管元组中的数据是什么 都不会开辟空间

深拷贝 只要数据含有**可变的类型**(like 嵌套) 都会开辟空间进行递归拷贝

### 3.私有化

1. \_x 左单下划线----防止被导入

2.\_\_xxx 左双下划线----私有化属性

3.xxx\_ 右单下划线----解决关键字重名情况(当使用关键字做变量的时候)

4. 类中的假私有化属性

使用魔法方法：**对象.\_\_dict\_\_** 就可以查看类中的私有属性

\_类\_\_属性---就可以从外部访问私有属性



### 2.1多继承以及MRO顺序

1.类名.\_\_init\_\_()

2. super().\_\_init\_\_()

参数列表：

1.多继承中需要在后面添加上(a, b, **\*args, \*\*kwargs**)

2.原因父类中不同的属性，子类继承传递上去不同的参数父类可以接收

|  |
| --- |
|  |

3.继承中super() 优势

原因:super()调用方法**会遵循\_\_mro\_\_**(c3算法编写)

|  |
| --- |
| 通过打印可以看到继承的一个流程，super()调用按照这个方法走  (<class '\_\_main\_\_.Grandson'>, <class '\_\_main\_\_.Son1'>, <class '\_\_main\_\_.Son2'>, <class '\_\_main\_\_.Parent'>, <class 'object'>) |

4.单继承 多继承

|  |
| --- |
| Class P: Class P:  pass pass  Class A(P): Class B:  pass pass  Class B(A): Class C(P, B):  pass pass |

查看继承顺序的方法:

**类名.\_\_mro\_\_**

|  |
| --- |
|  |

### 2.2 静态方法、类方法

类对象 存放类相关的数据空间

实例对象 存放实例相关的数据空间

类属性 存放类空间中的属性

实例属性 存放在实例空间中的属性

类方法 改变类属性的(cls默认类的空间地址) -**对象名(实例对象)和类名都可以访问类方法**

实例方法 改变实例属性

静态方法 不需要类对象也不实用实例对象

类和对象中间是怎么产生联系的呢？

通过cls和self记录的**内存地址值**,调用的时候搜索地址来产生通讯.

|  |
| --- |
| class Province(object):  # 类属性  country = '中国'  def \_\_init\_\_(**self**, name):  **# 实例属性 self—实例对象**  self.name = name  def fun1(self):  print(“实例方法”)  @classmethod  def class\_func(cls):  print('类方法')  @staticmethod  def static\_func():  print('静态方法') |

### 2.3 property属性-应用

1.作用: 简化私有属性的修改 调用的流程

2.property使用时注意点:1.@property装饰器 2.调用的时候**不需要()**

3.使用Property的两种方式

|  |
| --- |
| 三种装饰器  第一种装饰器: @property  第二种装饰器: @property方法名.setter  第三种装饰器: @property方法名.deleter |

|  |
| --- |
| 1.使用peoperty的第一种方式—**装饰器**  **class Goods:  def \_\_init\_\_(**self**):** # 原价  self.org\_price **=** 100  self.discount **=** 0.8  **@**property  **def price(**self**):** value **=** self.org\_price **\*** self.discount  **return** value   **@price.setter**  **def price(**self, values**):** self.org\_price **=** values   **@price.deleter**  **def price(**self**):  del** self.org\_price  p **=** Goods**()** # print(p.org\_price) # 设置商品价格 p.org\_price **=** 1000 print**(**p.org\_price**)** # 删除商品价格 # del p.price # print(p.org\_price)  2.使用property属性的第二种方式--**类属性**  **class Goods(**object**):  def \_\_init\_\_(**self**):** self.original\_price **=** 100  self.discount **=** 0.8   **def get\_price(**self**):** # 谨记返回值  new\_price **=** self.original\_price **\*** self.discount  **return** new\_price   **def set\_price(**self, value**):** self.original\_price **=** value   **def del\_price(**self**):  del** self.original\_price  **# 类属性**   **price = property(get\_price, set\_price, del\_price)** g1 **=** Goods**()** # 原价 print**(**g1.price**)** # 设置新价格 g1.price **=** 1000 print**(**g1.price**)** # 删除商品原价 **del** g1.price print**(**g1.price**)** |

\*\*\*只该私有属性仅返回数值的时候使用property优化私有属性代码

### 2.4 魔法属性

1. \_\_doc\_\_ 打印类的描述信息(就是““”xxx”””注释内的信息)

2.\_\_moudle\_\_---查看当前操作的对象在那个模块

\_\_class\_\_ ---查看当前操作对象的类是什么

3.\_\_dict\_\_ ---查看类中所有属性 or 查看对象内部的属性

4.\_\_init\_\_ ---初始化属性 \_\_new\_\_ -----创建内存空间

5. \_\_del\_\_ 当对象在内存中被释放时，自动触发执行

6. \_\_str\_\_ 类中定义了该方法后，创建对象打印时会**默认输入该方法内的返回值**

7.\_\_call\_\_ ---对象后面加括号,触发执行.

|  |
| --- |
| **class Foo:  def \_\_init\_\_(**self**):  pass  def \_\_call\_\_(**self, **\***args, **\*\***kwargs**):** print**('\_\_call\_\_')** obj **=** Foo**()** # 创建对象触发执行 \_\_init\_\_ obj**()** # 对象添加()触发执行 \_\_call\_\_ |

8. \_\_getitem\_\_ \_\_setitem\_\_ \_\_delitem\_\_

**用于索引操作如字典，以上分别表示获取，设置，删除数据**

|  |
| --- |
| **class Foo(**object**):  def \_\_getitem\_\_(**self, key**):** print**("\_\_getitem\_\_"**, key**)   def \_\_setitem\_\_(**self, key, value**):** print**("\_\_setitem\_\_"**, key, value**)  def \_\_delitem\_\_(**self, key**):** print**("\_\_delitem\_\_"**, key**)** f **=** Foo**()** # 传入参数 result **=** f**["k1"]** # 设置值 f**["k2"] = "laowang"** # 删除 **del** f**["k1"]** |

9.\_\_getslice\_\_ \_\_setslice\_\_ \_\_delslice\_\_

该三个方法用于切片操作 如:列表  **Python2，3已经没有.**

|  |
| --- |
| class Foo(object):  def \_\_getslice\_\_(self, i, j):  print('\_\_getslice\_\_', i, j)  def \_\_setslice\_\_(self, i, j, sequence):  print('\_\_setslice\_\_', i, j)  def \_\_delslice\_\_(self, i, j):  print('\_\_delslice\_\_', i, j)  obj = Foo()  obj[-1:1] # 自动触发执行 \_\_getslice\_\_  obj[0:1] = [11,22,33,44] # 自动触发执行 \_\_setslice\_\_  del obj[0:2] # 自动触发执行 \_\_delslice\_\_ |

### 2.5 面向对象设计

缺省参数



### 2.6 With 与“上下文管理器”

1.为什么With ----自动调用close()?

1）任何实现了\_\_enter\_\_()和\_\_exit\_\_()方法的对象都可称之为上下文管理器

2）with 语法用于简化资源操作的后续**清除操作**，是 try/finally 的替代方法，实现原理建立在上下文管理器之上

|  |
| --- |
| with open(“1.txt”, “w”) as f:  f.write(“Sunday”) |

2.什么是上下文管理器

任何实现了 \_\_enter\_\_() 和 \_\_exit\_\_() 方法的对象都可称之为上下文管理器

|  |
| --- |
| **class File(**object**):  def \_\_init\_\_(**self, filename, mode**):** self.filename **=** filename  self.mode **=** mode   **def \_\_enter\_\_(**self**):** self.f **=** open**(**self.filename, self.mode**)  return** self.f   **def \_\_exit\_\_(**self, **\***args**):** self.f.close**()  with** File**("1.txt"**, **"a+") as** f**:** f.write**("\nhello, python")** |