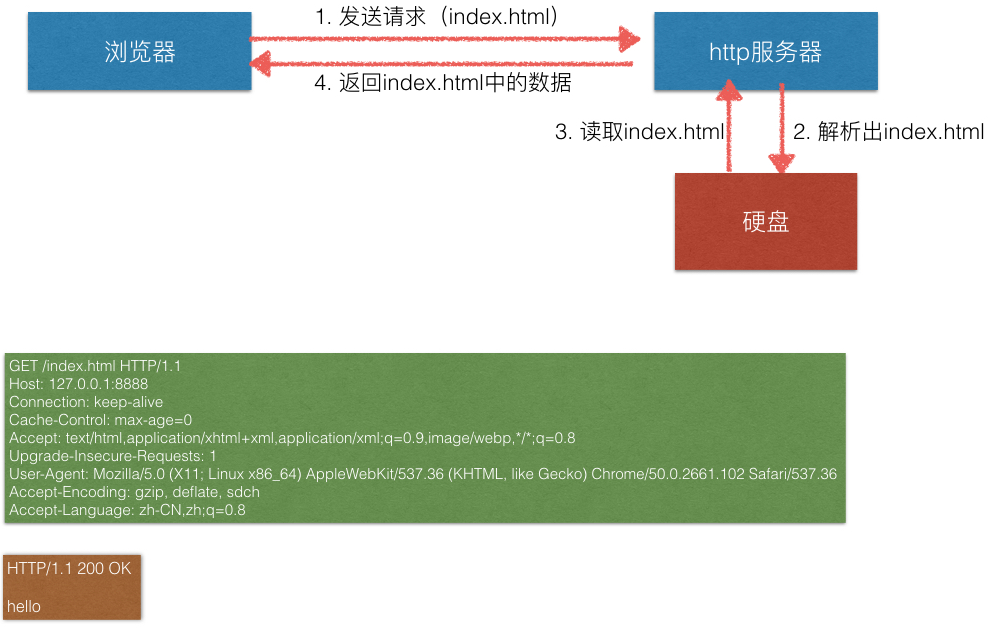
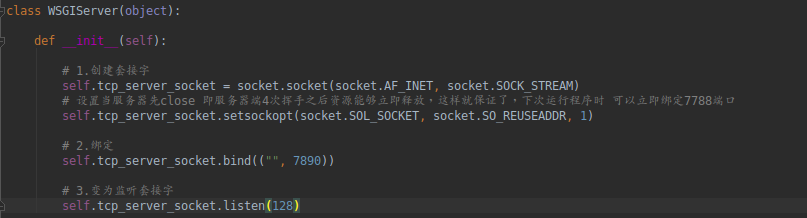


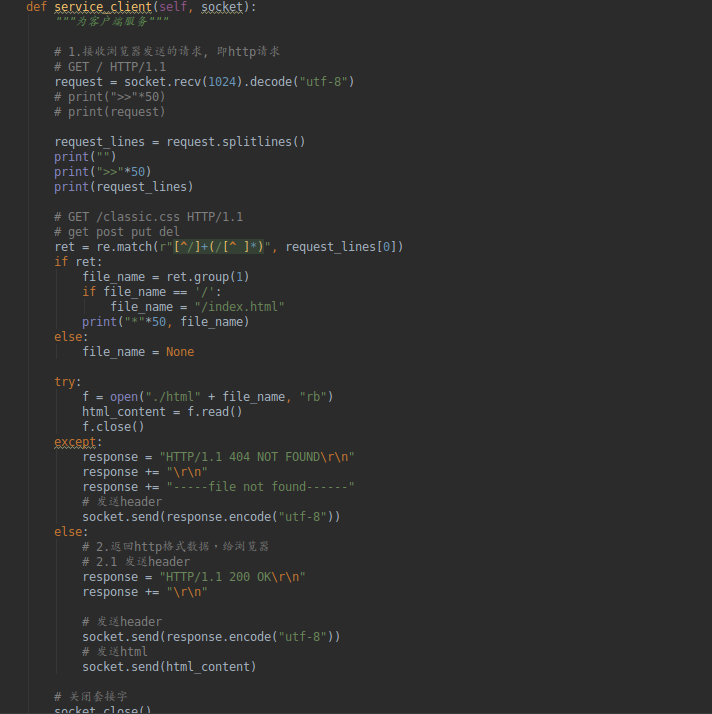
**一、WSGI-mini-web框架**

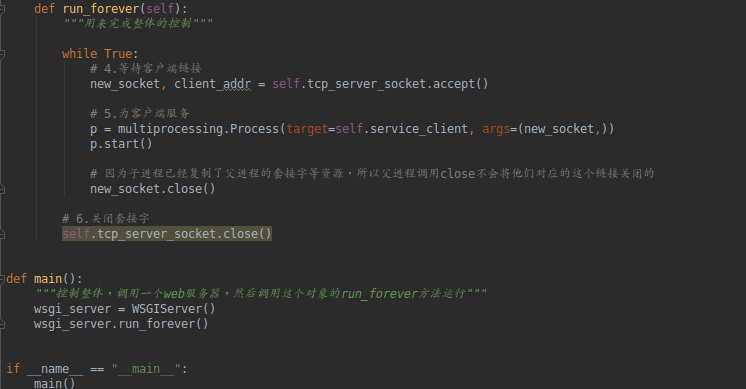
**1. 静态服务器**



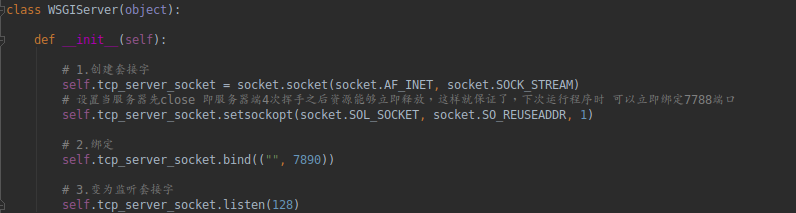
浏览器直接申请资源，服务器提取url信息，读取后发送数据。

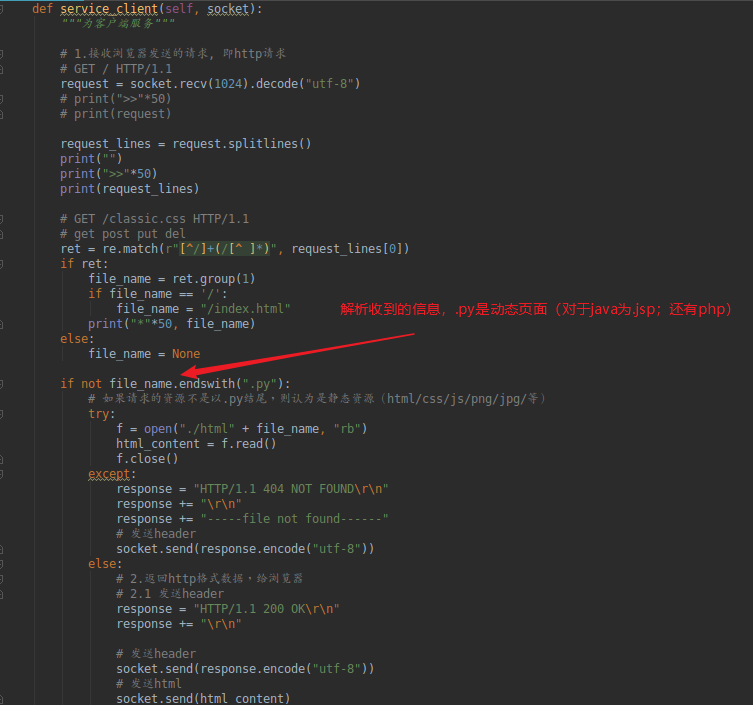


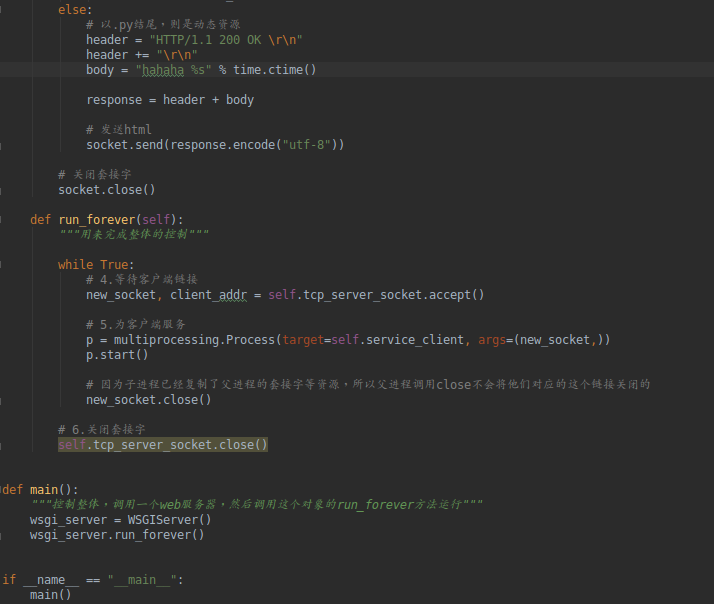




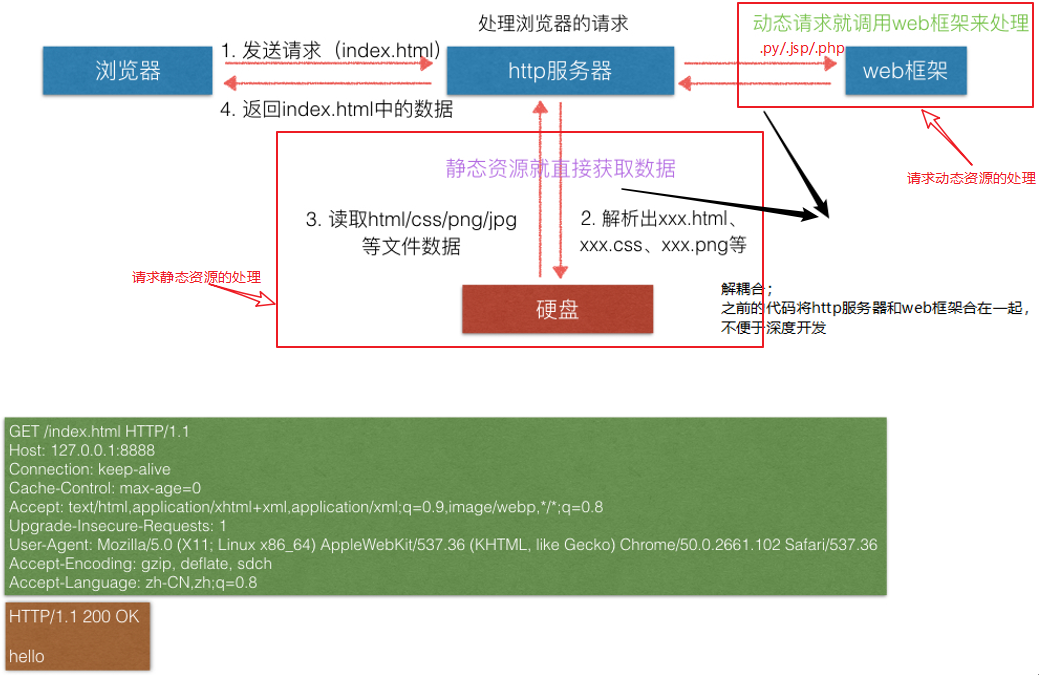
**2. 动态服务器**



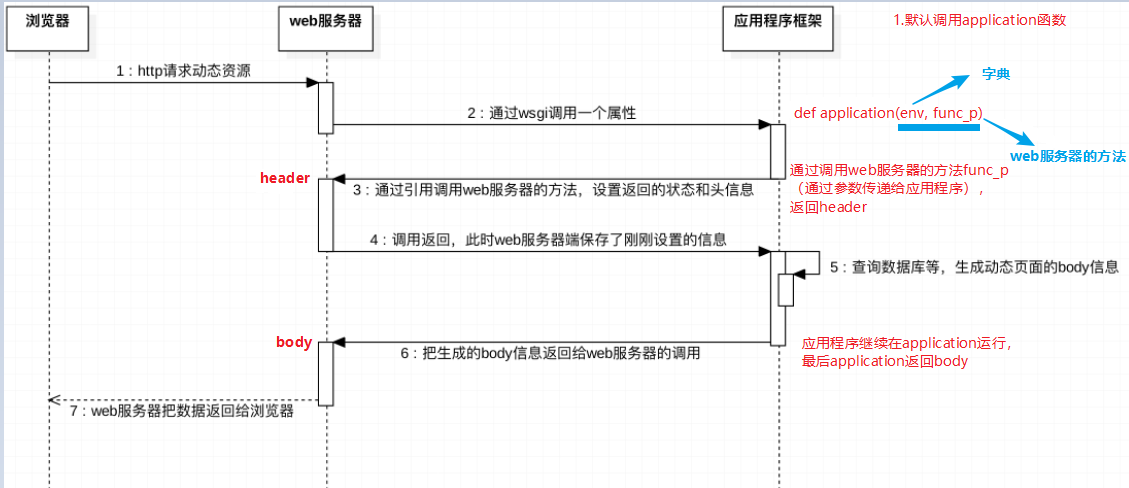




静态服务器是静态的资源，如js/css/html/jpg/bnp等，动态的是根据代码要求进行变化；



* 1. **服务器动态资源请求**

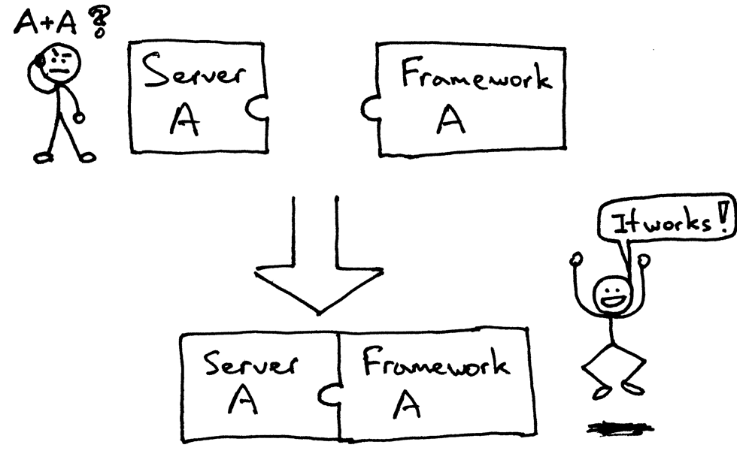


应用程序框架通过调用application，application传递两个参数，第一个是字典，第二个是web服务器的方法，应用程序通过调用web服务器方法func\_p返回header，再继续运行application返回body。

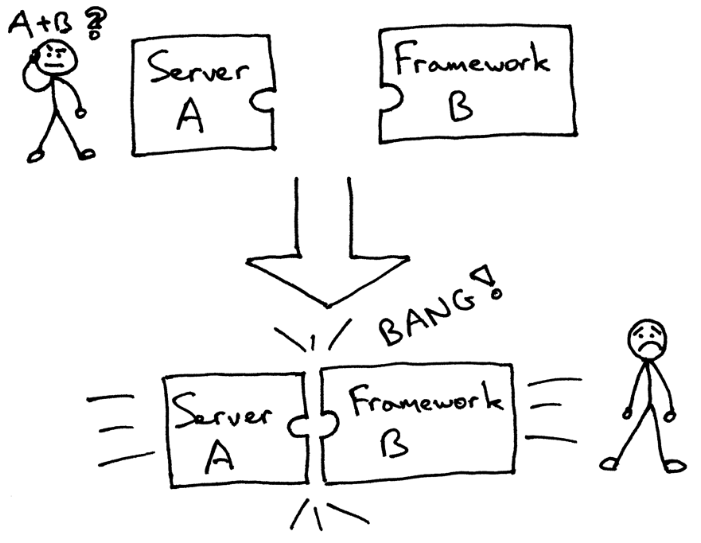
* 1. **WSGI**

怎么在你刚建立的Web服务器上运行一个Django应用和Flask应用，如何不做任何改变而适应不同的web架构呢？

在以前，选择 Python web 架构会受制于可用的web服务器，反之亦然。如果架构和服务器可以协同工作，那就好了：

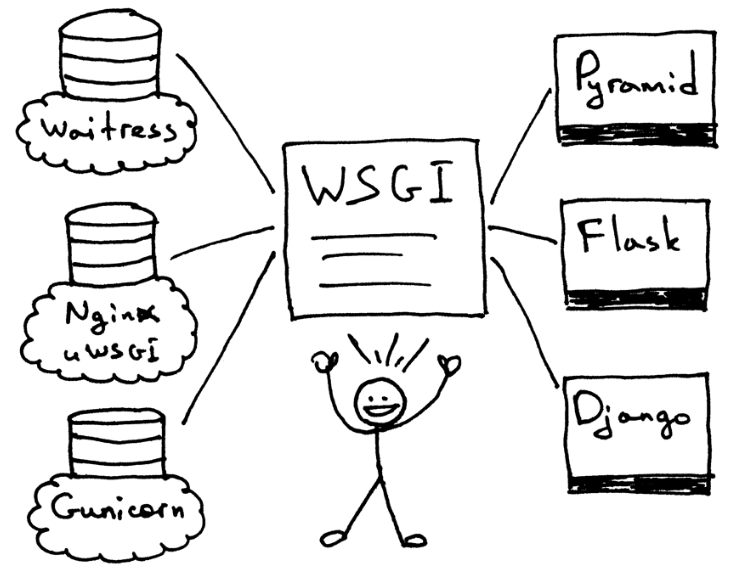


但有可能面对（或者曾有过）下面的问题，当要把一个服务器和一个架构结合起来时，却发现他们不是被设计成协同工作的：



那么，怎么可以不修改服务器和架构代码而确保可以在多个架构下运行web服务器呢？答案就是 Python Web Server Gateway Interface (或简称 WSGI，读作“wizgy”)。

WSGI允许开发者将选择web框架和web服务器分开。可以混合匹配web服务器和web框架，选择一个适合的配对。比如,可以在Gunicorn 或者 Nginx/uWSGI 或者 Waitress上运行 Django, Flask, 或 Pyramid。真正的混合匹配，得益于WSGI同时支持服务器和架构：



web服务器必须具备WSGI接口，所有的现代Python Web框架都已具备WSGI接口，它让你不对代码作修改就能使服务器和特点的web框架协同工作。

WSGI由web服务器支持，而web框架允许你选择适合自己的配对，但它同样对于服务器和框架开发者提供便利使他们可以专注于自己偏爱的领域和专长而不至于相互牵制。其他语言也有类似接口：java有Servlet API，Ruby 有 Rack。

* 1. **定义WSGI接口**

WSGI接口定义非常简单，它只要求Web开发者实现一个函数，就可以响应HTTP请求。我们来看一个最简单的Web版本的“Hello World!”：

def application(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

return 'Hello World!'

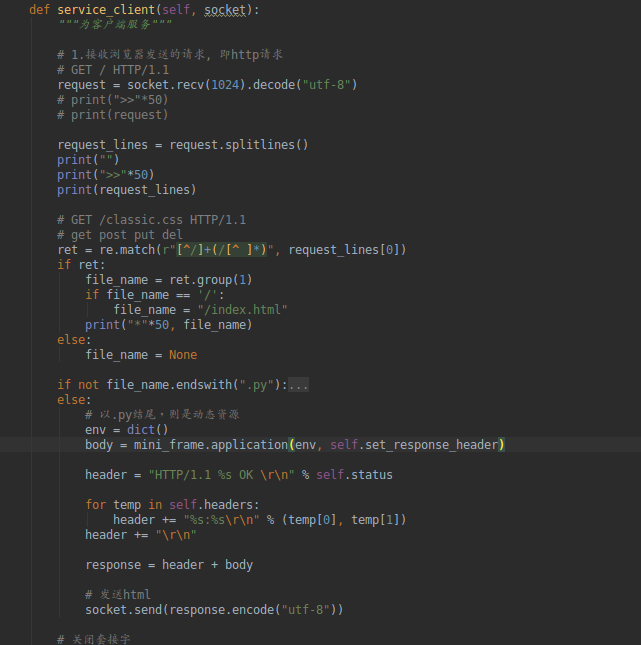
上面的application()函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数：

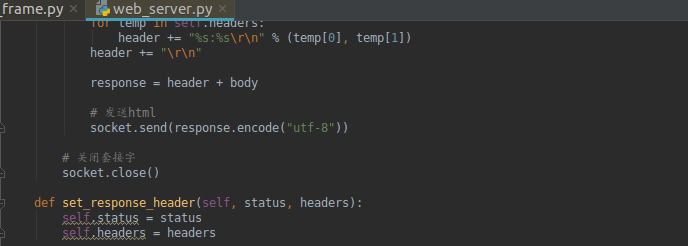
* environ：一个包含所有HTTP请求信息的dict对象；
* start\_response：一个发送HTTP响应的函数。

整个application()函数本身没有涉及到任何解析HTTP的部分，也就是说，把底层web服务器解析部分和应用程序逻辑部分进行了分离，这样开发者就可以专心做一个领域了

不过，等等，这个application()函数怎么调用？如果我们自己调用，两个参数environ和start\_response我们没法提供，返回的str也没法发给浏览器。

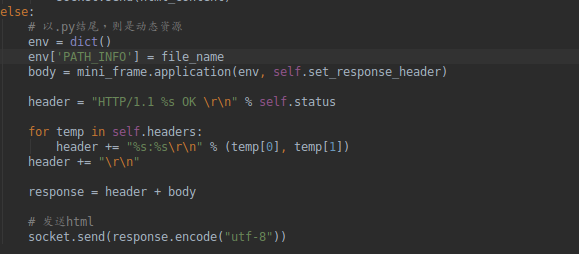
所以application()函数必须由WSGI服务器来调用。有很多符合WSGI规范的服务器。而我们此时的web服务器项目的目的就是做一个既能解析静态网页还可以解析动态网页的服务器

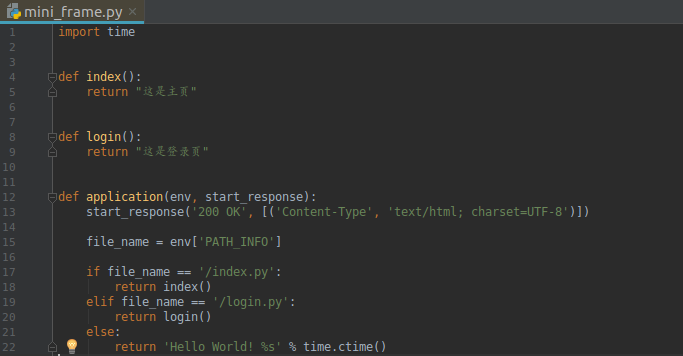




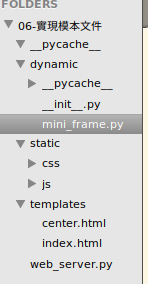


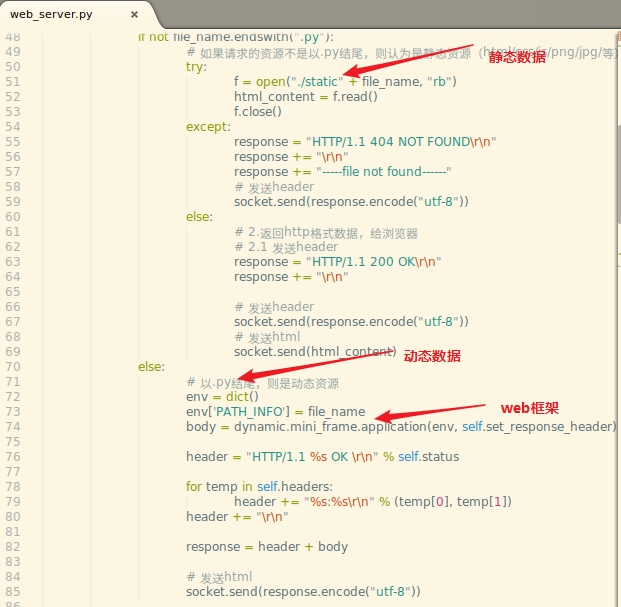
* 1. **设置传输字典**





* 1. **获取页面模板**

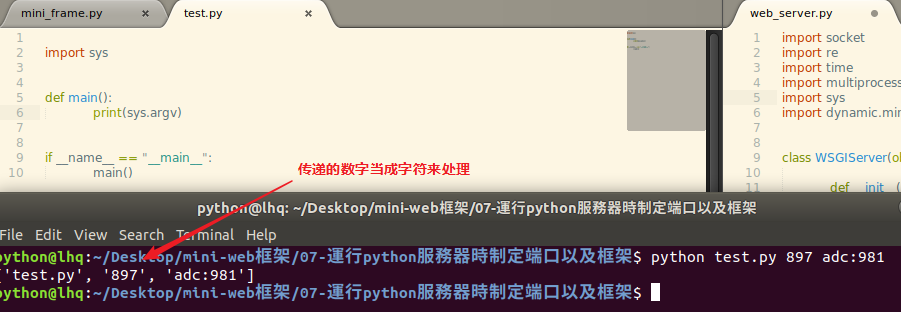




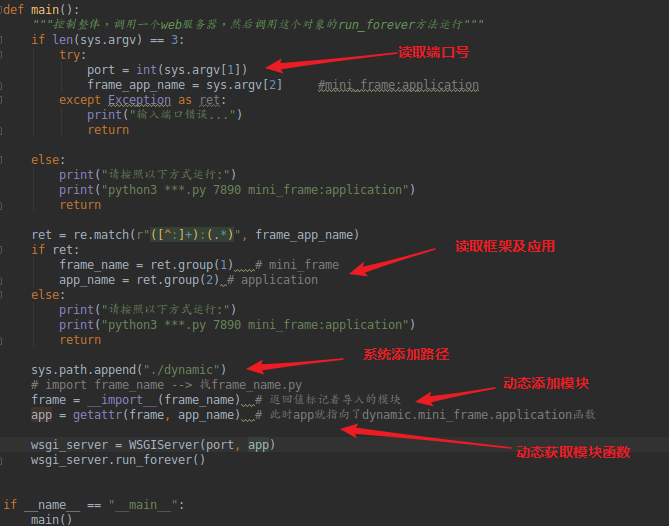


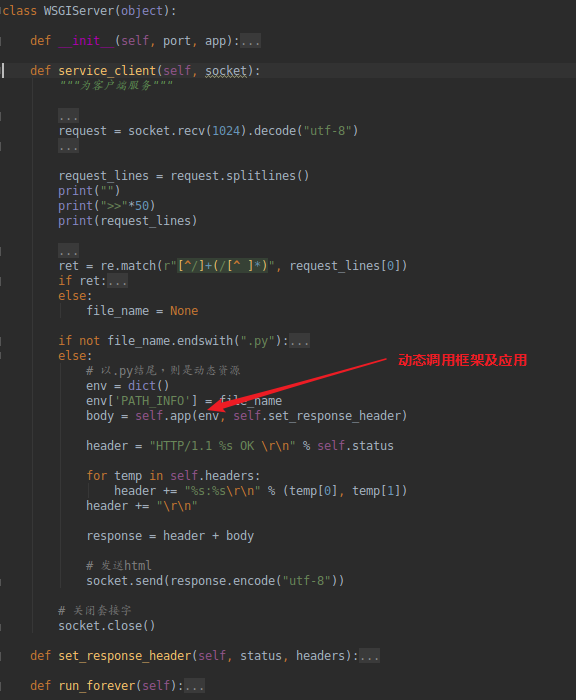
* 1. **传递参数**

**1.测试参数传递**



**2.添加web服务器的端口及应用**

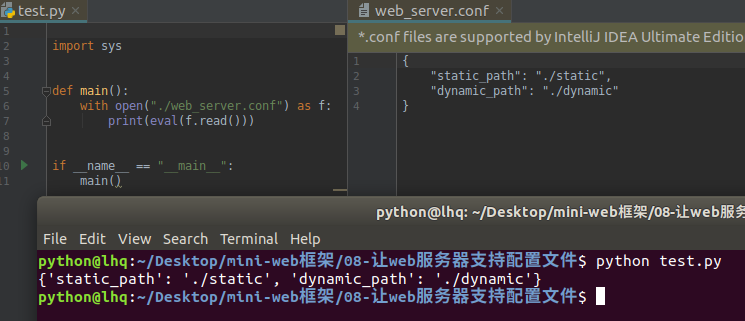




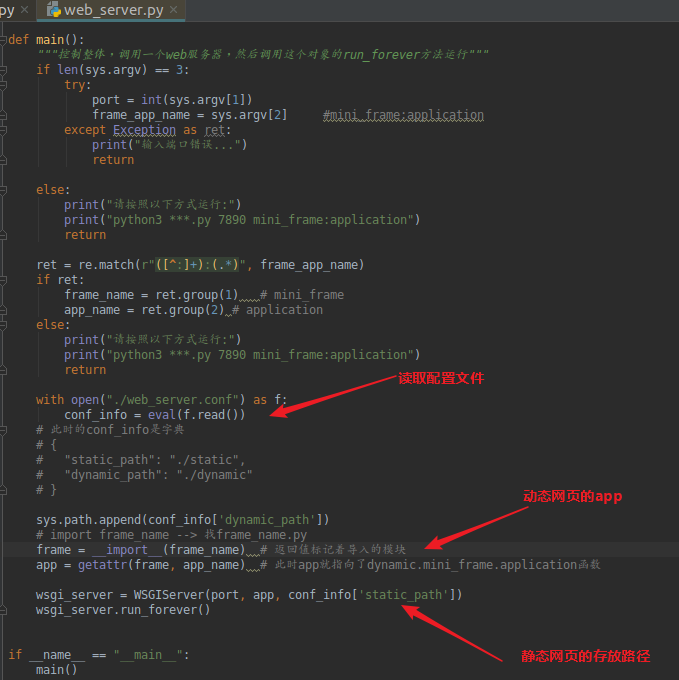


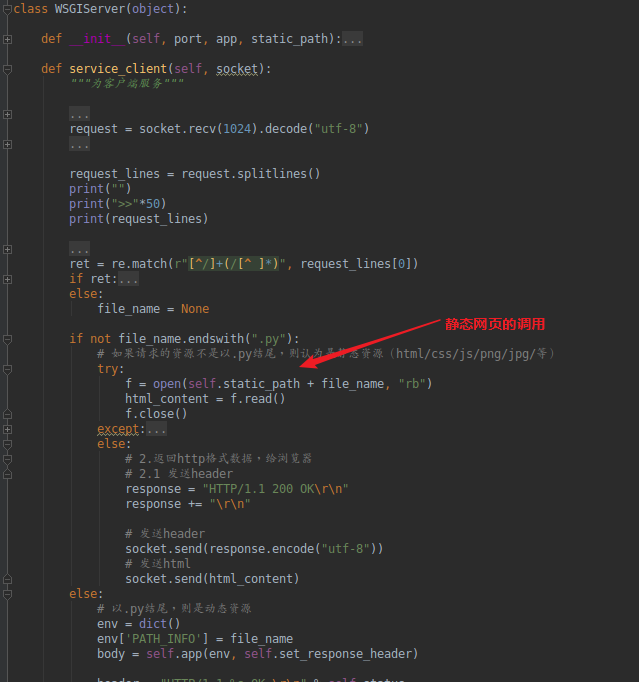
* 1. **给web服务器配置文件**

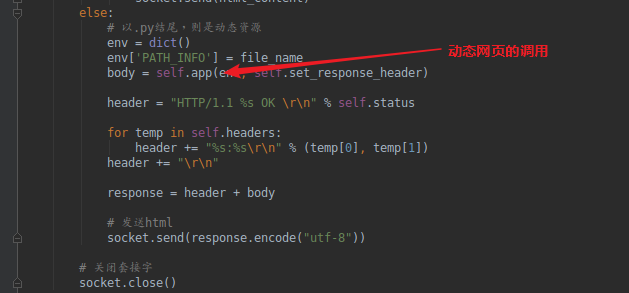
**1.测试配置文件的读取**



**2.添加web服务器的配置文件**







* 1. **添加shell功能**

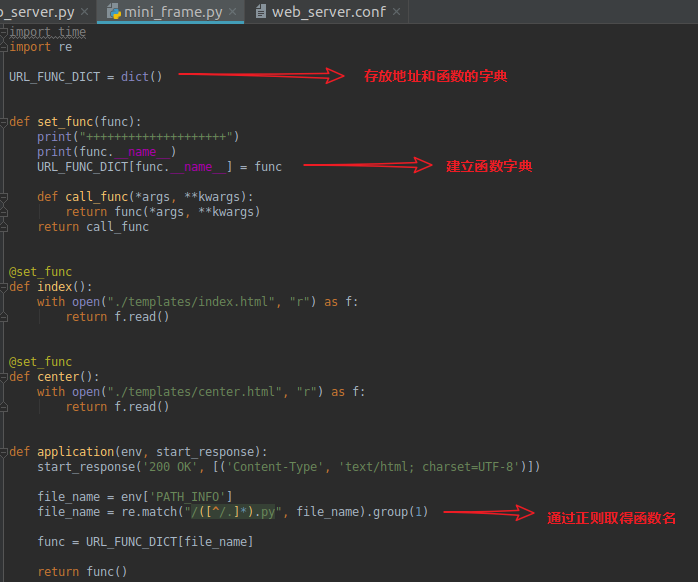


**二、mini-web框架添加路由和MySQL功能**

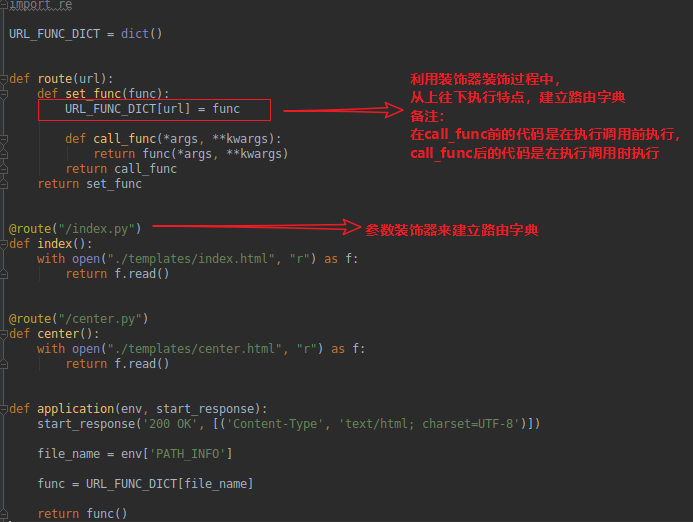
**2.1 用装饰器实现网址路由功能**

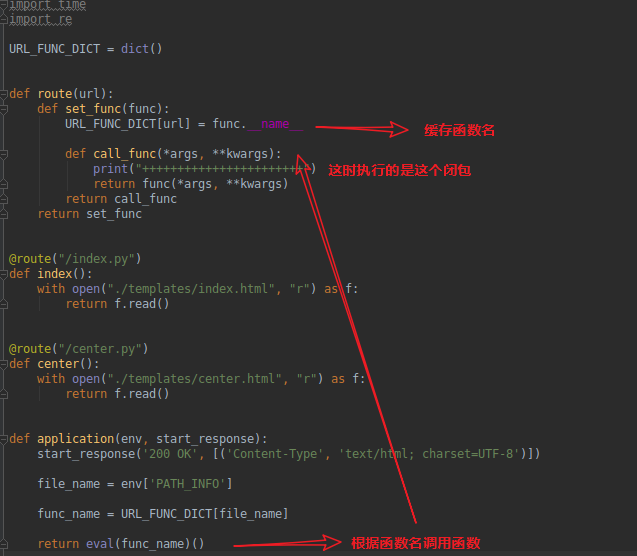
网络路由：根据网址不一样，调用的内容就不一样。

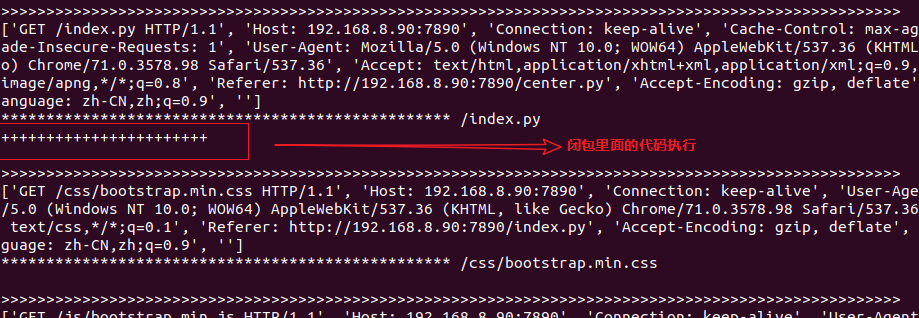
方案1：采用正则表达式解析网址



方案2：采用带参数的装饰器实现

  
理解：url\_func\_dict缓存的地址是直接函数的地址，没有经过第二层装饰器，调用时直接指向函数不会执行call\_func里面的代码；如果需要执行call\_func里面的代码，则需要如下使用，缓存函数名称，根据函数名称去指向闭包函数。





**2.2 伪静态、静态和动态的区别**

目前开发的网站其实真正意义上都是动态网站，只是URL上有些区别，一般URL分为静态URL、动态URL、伪静态URL，他们的区别是什么?

**2.2.1 静态URL**

静态URL类似 域名/news/2012-5-18/110.html 我们一般称为真静态URL，每个网页有真实的物理路径，也就是真实存在服务器里的。

* 优点是：

网站打开速度快，因为它不用进行运算；另外网址结构比较友好，利于记忆。

* 缺点是：

最大的缺点是如果是中大型网站，则产生的页面特别多，不好管理。至于有的开发者说占用硬盘空间大，我觉得这个可有忽略不计，占用不了多少空间的，况且目前硬盘空间都比较大。还有的开发者说会伤硬盘，这点也可以忽略不计。

* 一句话总结:

静态网站对SEO的影响：静态URL对SEO肯定有加分的影响，因为打开速度快，这个是本质。

SEO（Search Engine Optimization）：汉译为搜索引擎优化。 是一种方式：利用搜索引擎的规则提高网站在有关搜索引擎内的自然排名。

**2.2.2 动态URL**

动态URL类似 域名/NewsMore.asp?id=5 或者 域名/DaiKuan.php?id=17，带有？号的URL，我们一般称为动态网址，每个URL只是一个逻辑地址，并不是真实物理存在服务器硬盘里的。

* 优点是：

适合中大型网站，修改页面很方便，因为是逻辑地址，所以占用硬盘空间要比纯静态网站小。

* 缺点是：

因为要进行运算，所以打开速度稍慢，不过这个可有忽略不计，目前有服务器缓存技术可以解决速度问题。最大的缺点是URL结构稍稍复杂，不利于记忆。

* 一句话总结:

动态URL对SEO的影响：目前百度SE已经能够很好的理解动态URL，所以对SEO没有什么减分的影响（特别复杂的URL结构除外）。所以你无论选择动态还是静态其实都无所谓，看你选择的程序和需求了。

**2.2.3 伪静态URL**

伪静态URL类似 域名/course/74.html 这个URL和真静态URL类似。他是通过伪静态规则把动态URL伪装成静态网址。也是逻辑地址，不存在物理地址。

* 优点是：

URL比较友好，利于记忆。非常适合大中型网站，是个折中方案。

* 缺点是：

设置麻烦，服务器要支持重写规则，小企业网站或者玩不好的就不要折腾了。另外进行了伪静态网站访问速度并没有变快，因为实质上它会额外的进行运算解释，反正增加了服务器负担，速度反而变慢，不过现在的服务器都很强大，这种影响也可以忽略不计。还有可能会造成动态URL和静态URL都被搜索引擎收录，不过可以用robots禁止掉动态地址。

* 一句话总结:

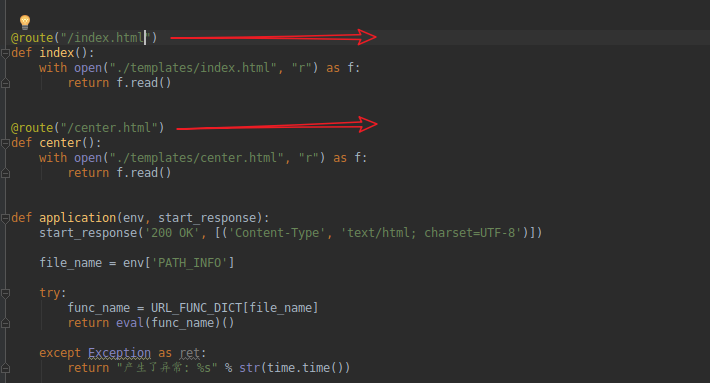
对SEO的影响：和动态URL一样，对SEO没有什么减分影响。

**2.3 mini-web框架-实现伪静态url**

修改web服务器：



修改应用：



修改模板中的超链接，将“.py”->“.html”

**2.4 准备数据**

**1. 创建数据库**

create database stock\_db charset=utf8;

**2. 选择数据库**

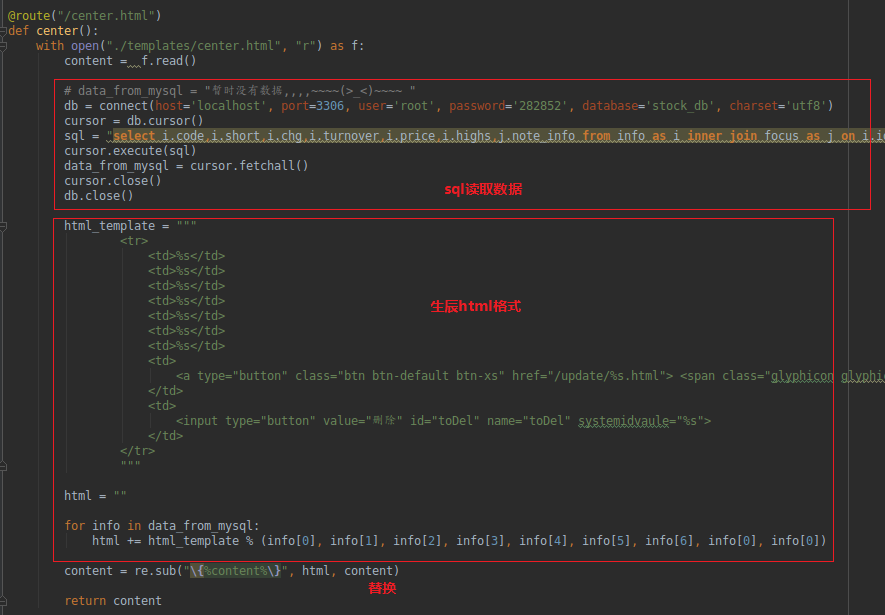
use stock\_db;

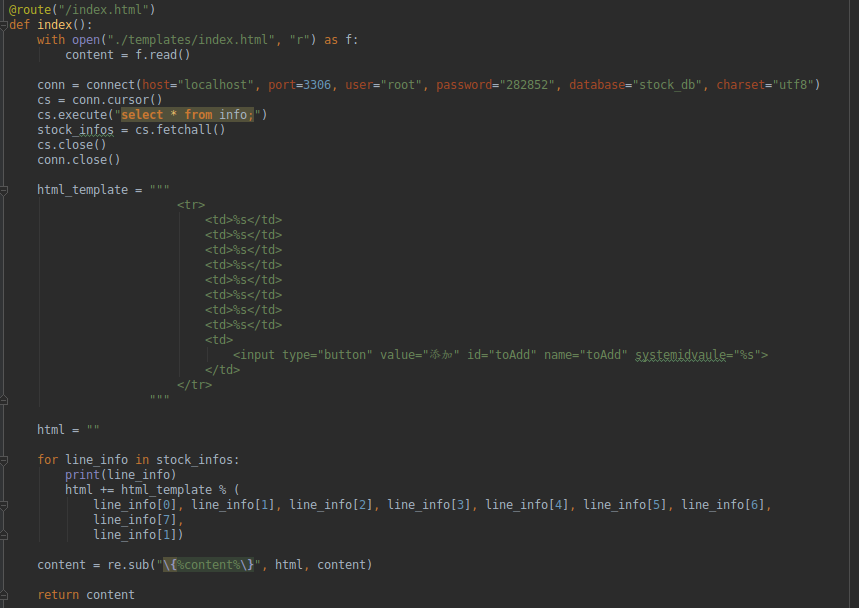
**3. 导入数据**

stock\_db.sql在课件中

source stock\_db.sql

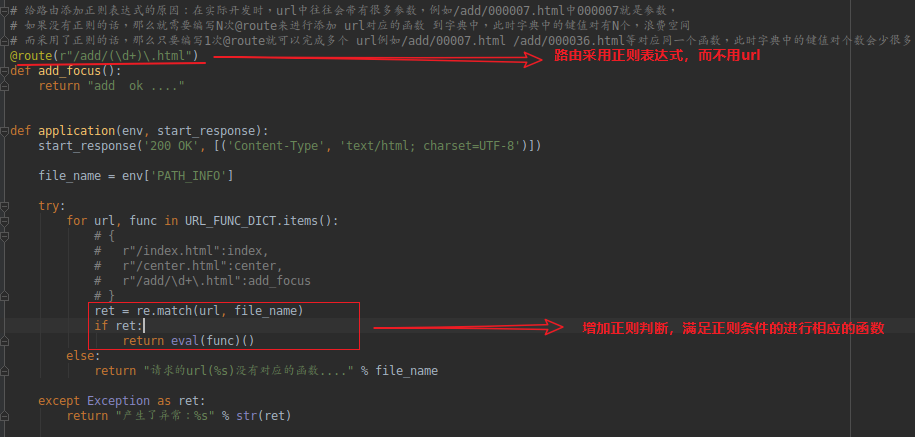
**2.5 mini-web框架-组装数据为html格式**





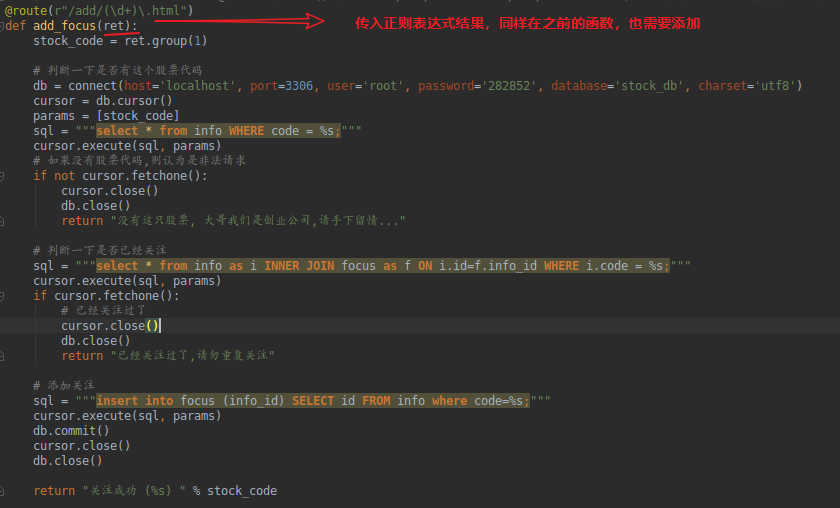
**三、mini-web框架添加log和路由正则**

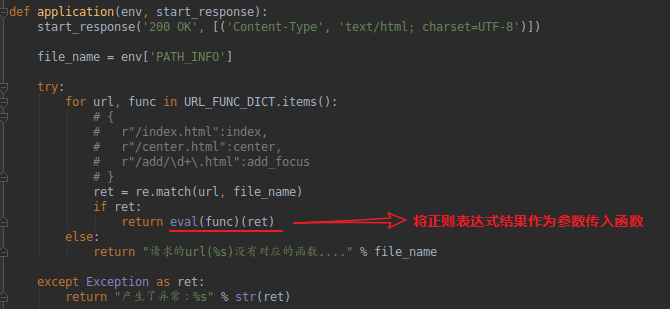
**3.1 路由支持正则**

需要将课件里面的模板更新到项目中，更新index.html、center.html、update.html

**3.2 mini-web框架-mysql-增加**

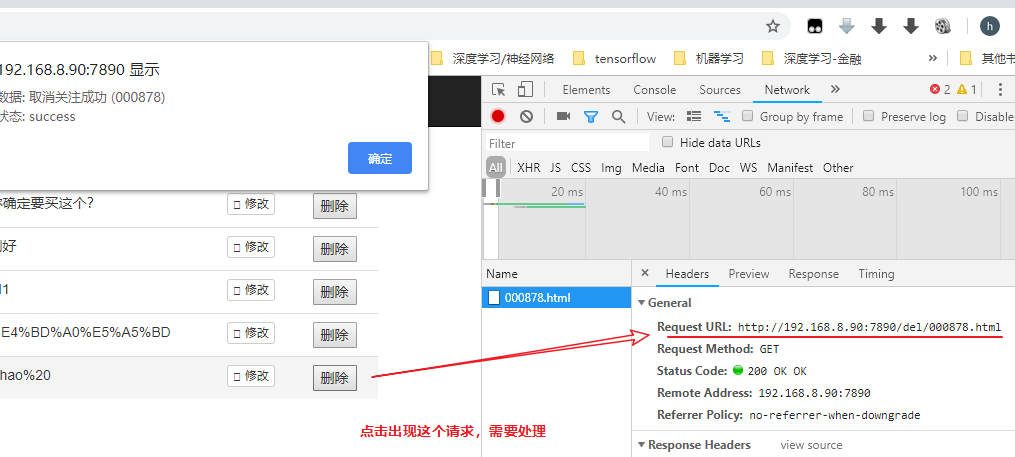


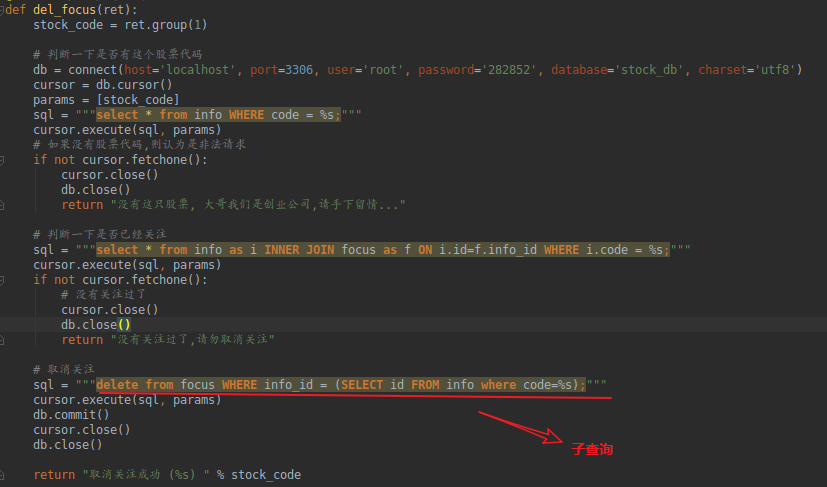




**3.3 mini-web框架-mysql-删除**

同添加类似

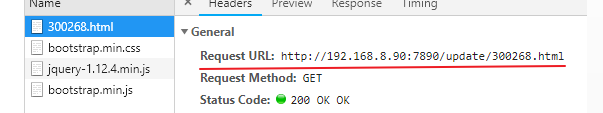


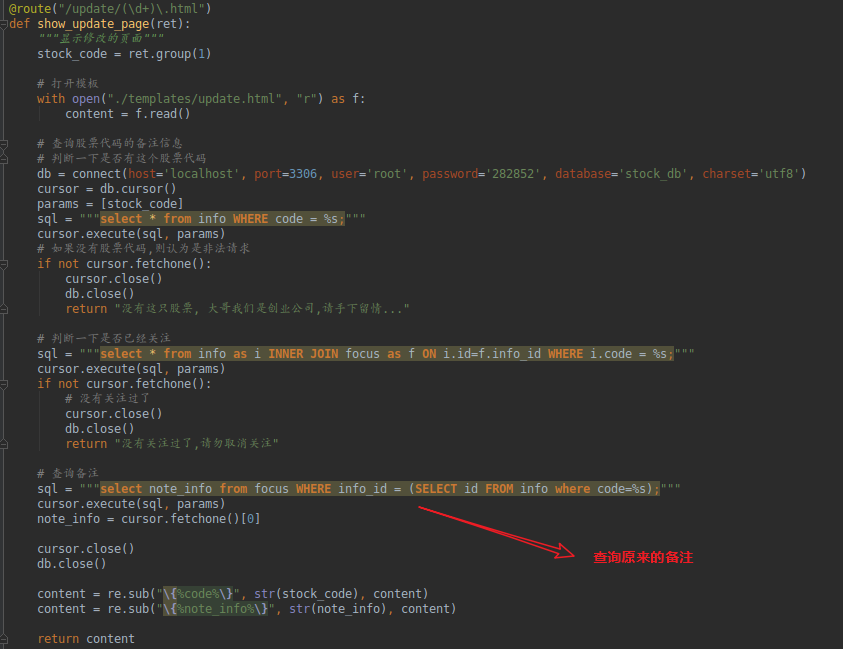


**3.4 mini-web框架-mysql-修改**

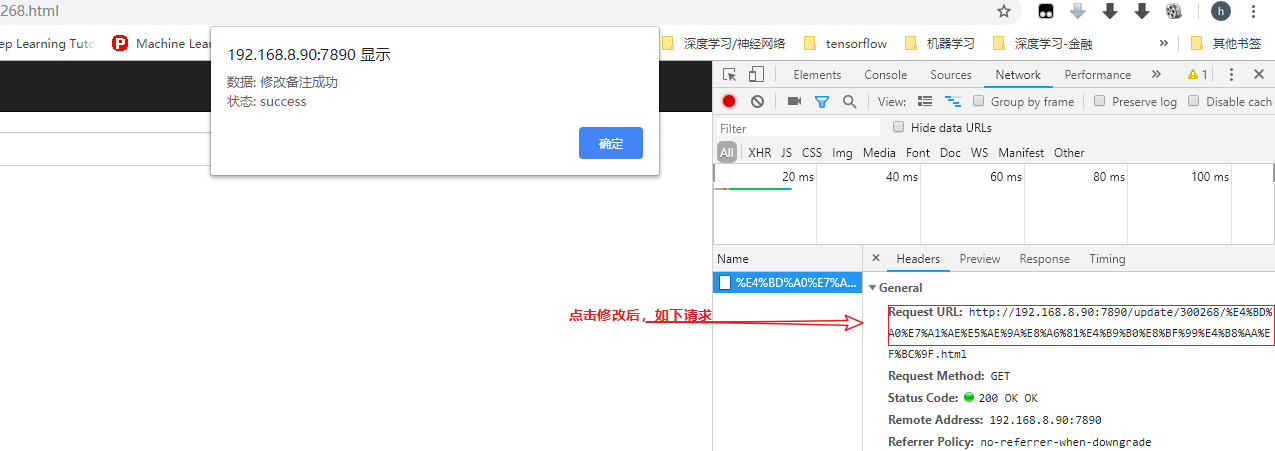
显示新的修改页面：

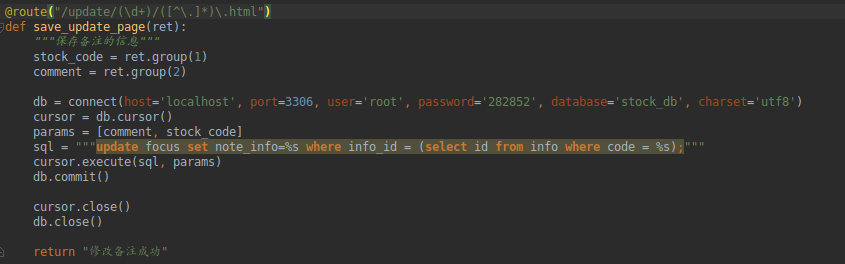
点击修改出现请求如下：





提交修改备注信息：

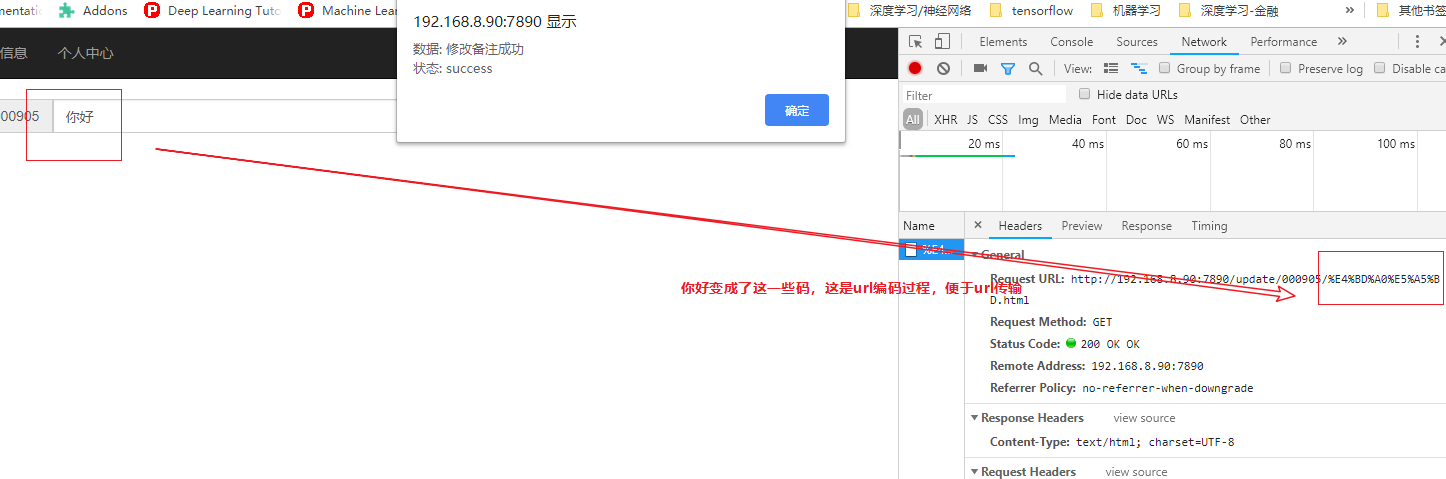




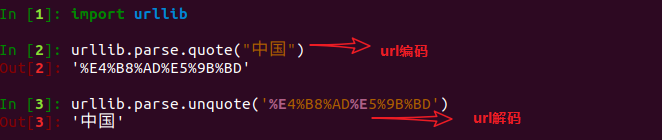
但是会出现以下乱码：

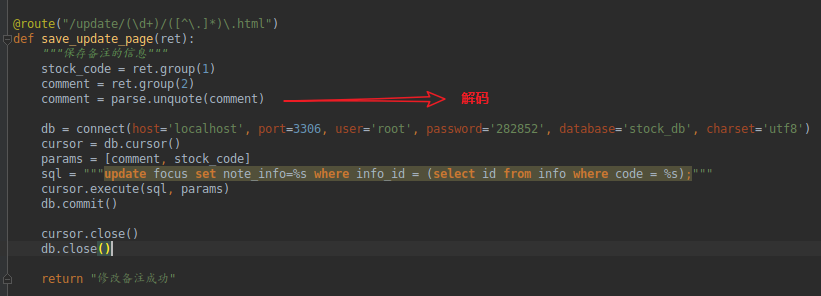


**3.5 mini-web框架-url编码**

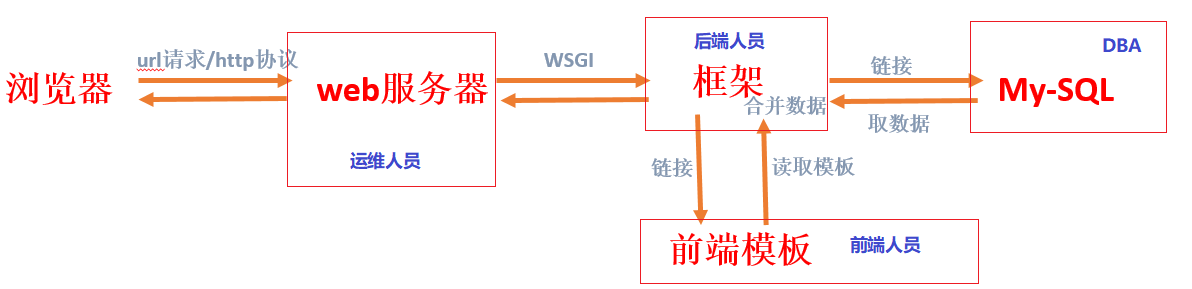


url对“你好”进行了编码，将编码传输给框架(即处理函数save\_update\_page)，如果直接存储则是这一推编码，需要在函数里面进行解码。





**3.6 web开发过程及结构**



**3.7 logging日志模块**

Python中有个logging模块可以完成相关信息的记录，在debug时用它往往事半功倍

**3.7.1. 日志级别**

日志一共分成5个等级，从低到高分别是：

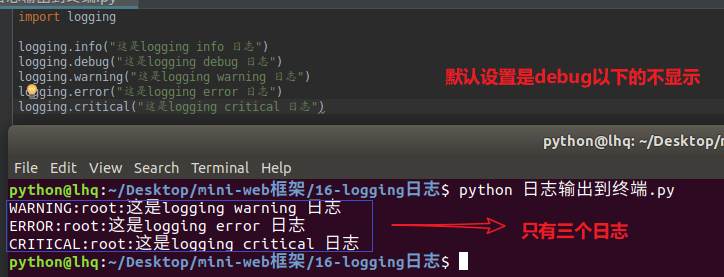
1. DEBUG
2. INFO
3. WARNING
4. ERROR
5. CRITICAL

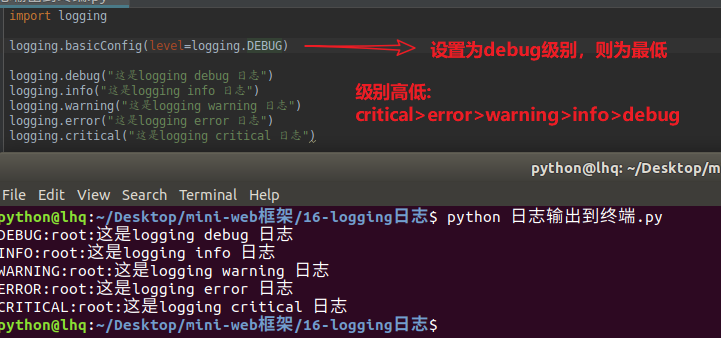
说明:

* DEBUG：详细的信息,通常只出现在诊断问题上
* INFO：确认一切按预期运行
* WARNING：一个迹象表明,一些意想不到的事情发生了,或表明一些问题在不久的将来(例如。磁盘空间低”)。这个软件还能按预期工作。
* ERROR：更严重的问题,软件没能执行一些功能
* CRITICAL：一个严重的错误,这表明程序本身可能无法继续运行

这5个等级，也分别对应5种打日志的方法： debug 、info 、warning 、error 、critical。默认的是WARNING，当在WARNING或之上时才被跟踪。

**3.7.2 将日志输出到控制台（print）**



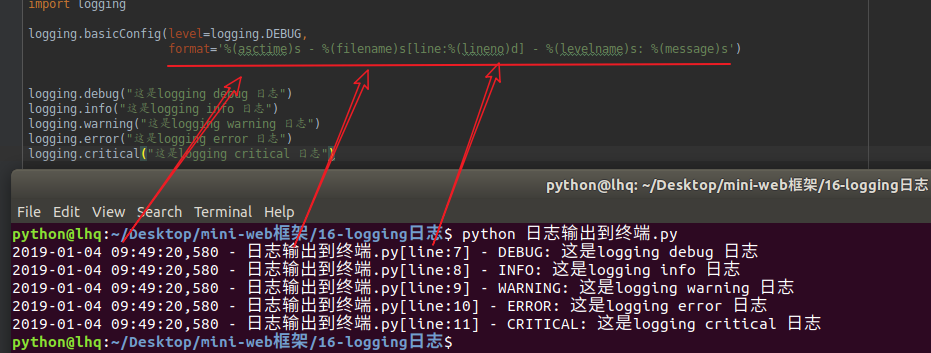


通过logging.basicConfig函数对日志的输出格式及方式做相关配置，上面代码设置日志的输出等级是DEBUG级别，意思是DEBUG级别以上的日志才会输出。

logging.basicConfig(level=logging.WARNING,

format='%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s')

**一般加上这一行，表示时间、文件名、行数，方便快速查找问题代码位置。**



注意，只要用过一次log功能再次设置格式时将失效，实际开发中格式肯定不会经常变化，所以刚开始时需要设定好格式

**3.7.3 将日志输出到文件**

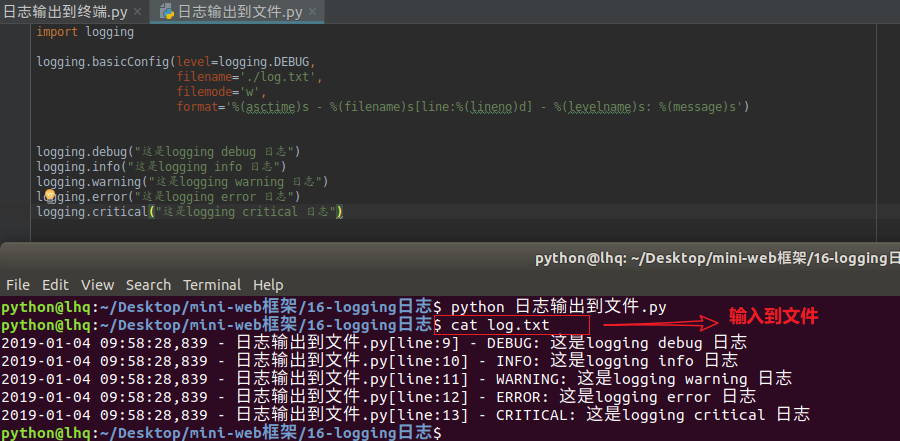
logging.basicConfig(level=logging.WARNING,

**filename='./log.txt',**

**filemode='w',**

format='%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s')

备注：这里的filemode可以是“a”（添加模式），“w”（覆盖模式）



**3.7.4 将日志输出到文件和终端**

import logging

# 第一步，创建一个logger

logger = logging.getLogger()

logger.setLevel(logging.INFO) # Log等级总开关

# 第二步，创建一个handler，用于写入日志文件

logfile = './log.txt'

fh = logging.FileHandler(logfile, mode='a') # open的打开模式这里可以进行参考

fh.setLevel(logging.DEBUG) # 输出到file的log等级的开关

# 第三步，再创建一个handler，用于输出到控制台

ch = logging.StreamHandler()

ch.setLevel(logging.WARNING) # 输出到console的log等级的开关

# 第四步，定义handler的输出格式

formatter = logging.Formatter("%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s")

fh.setFormatter(formatter)

ch.setFormatter(formatter)

# 第五步，将logger添加到handler里面

logger.addHandler(fh)

logger.addHandler(ch)

# 日志

logger.debug('这是 logger debug message')

logger.info('这是 logger info message')

logger.warning('这是 logger warning message')

logger.error('这是 logger error message')

logger.critical('这是 logger critical message')

**3.7.5 日志格式说明**

logging.basicConfig函数中，可以指定日志的输出格式format，这个参数可以输出很多有用的信息，如下:

* %(levelno)s: 打印日志级别的数值
* %(levelname)s: 打印日志级别名称
* %(pathname)s: 打印当前执行程序的路径，其实就是sys.argv[0]
* %(filename)s: 打印当前执行程序名
* %(funcName)s: 打印日志的当前函数
* %(lineno)d: 打印日志的当前行号
* %(asctime)s: 打印日志的时间
* %(thread)d: 打印线程ID
* %(threadName)s: 打印线程名称
* %(process)d: 打印进程ID
* %(message)s: 打印日志信息

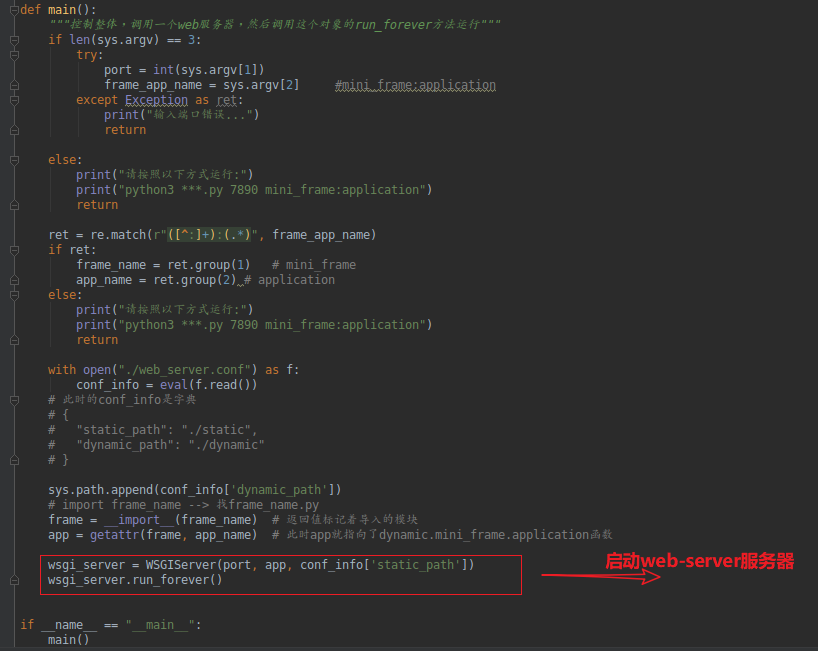
在工作中给的常用格式如下:

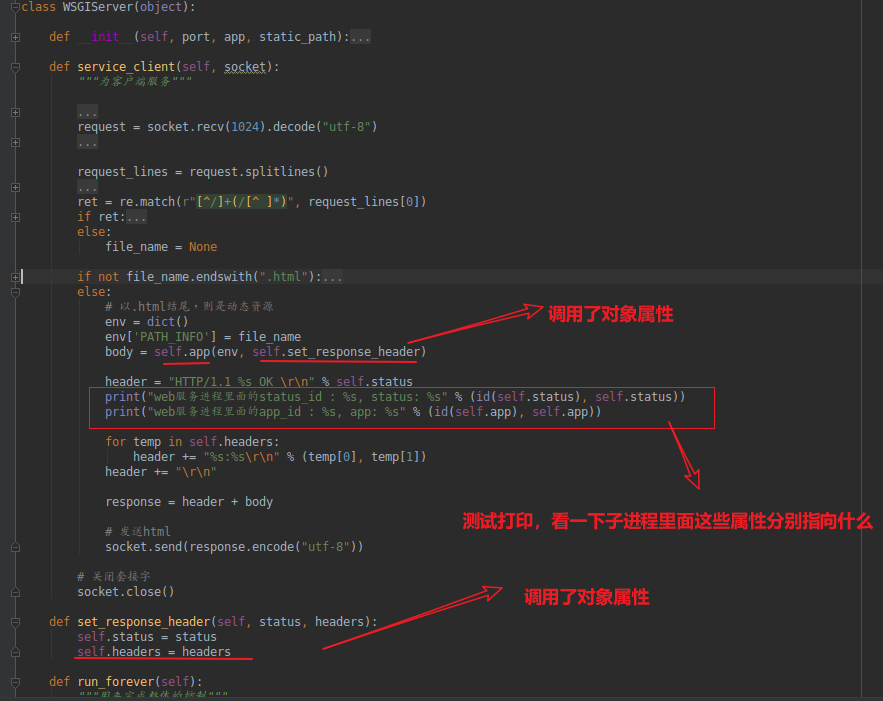
format='%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s: %(message)s'

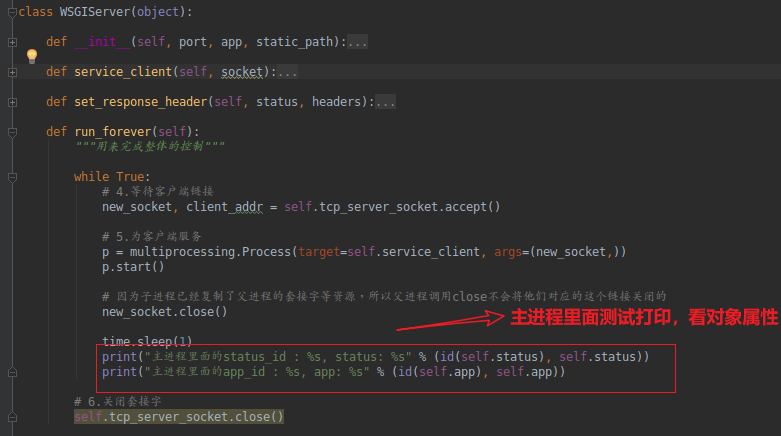
这个格式可以输出日志的打印时间，是哪个模块输出的，输出的日志级别是什么，以及输入的日志内容。

**四、再议多进程**

之前的代码中用到了mini-web-server，服务器中采用的多进程进行web服务。但是建立多进程时，传输的是service\_client函数和socket，没有传输web-server对象中的属性。但是实际服务中，调用了对象的属性，那多进程是怎么调用的？为什么没有向进程传输参数就能调用对象的属性？





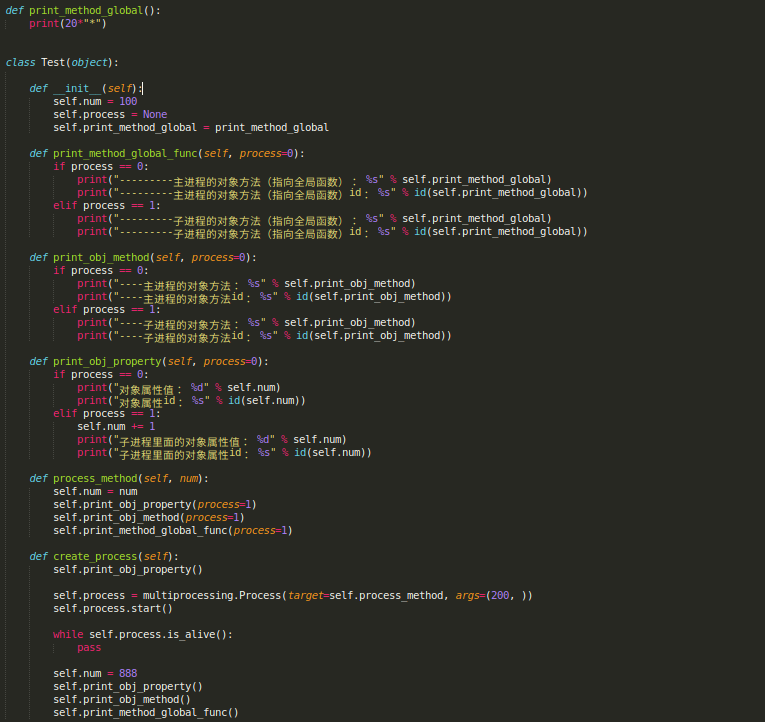


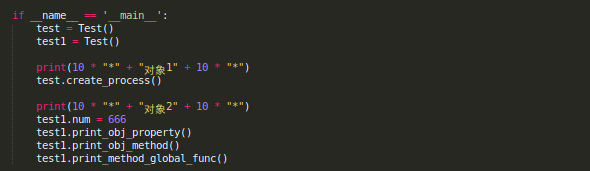


分析：

从以上结果来看，在实例里面进行多进程创建，实例属性进行了复制，同时指向一致；当给实例属性重新复制时，实例属性指向不一致，因此上文中self.status在子进程中重新复制，导致主/子进程id不一致；而self.app本质上是属性，指向外部函数，同时在主/子进程里面，没有重新复制，因此id一致，指向的都是外部函数。

详见以下实例：

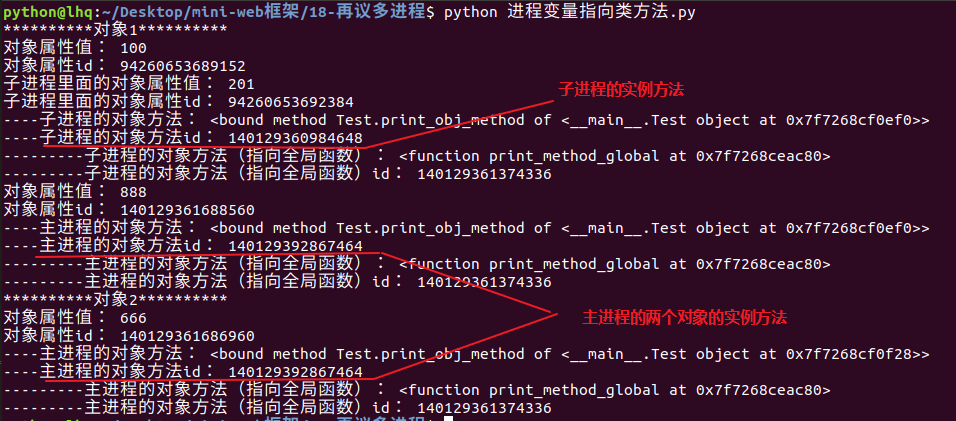






先观察实例属性的变化，实例生成后，实例属性初始化id都指向类里面定义的对象属性id，当实例属性修改后，该实例属性更改，修改之后每个实例属性id都是独立的。

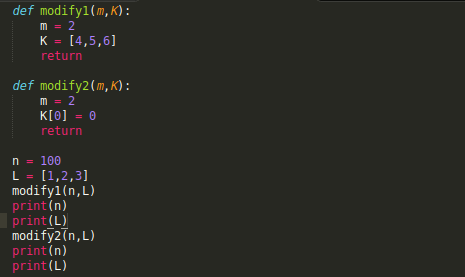
后面会认识到类本身就是对象，因此类本身定义的实例属性是占有内存空间的。当通过类创建实例时，对象的实例属性首先指向类定义里面的实例属性内存空间，当对象重新赋值实例属性时，对象的实例对象指向新地址。对于任何新创建的对象，该对象的初始实例属性指向都是一致的。

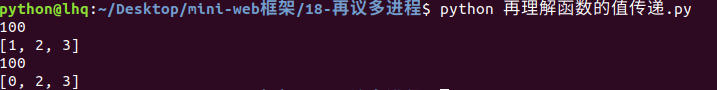


对于实例方法，创建进程时，子进程创建了实例方法指针存放的内存空间即id，但是该指针指向一致的实例方法。

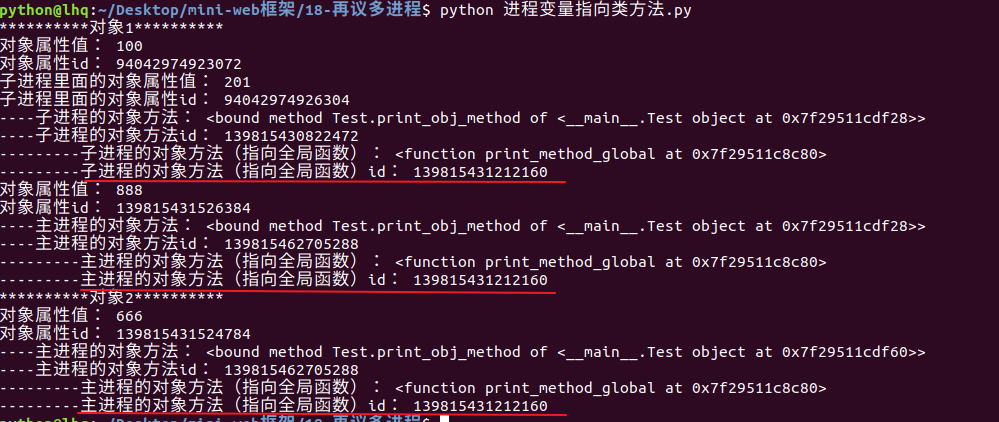
对于id的在理解，id(变量)=指针地址，变量=\*指针地址(即指针->指向的值)

**（插播）再理解函数的值传递：**





从结果可以看出，执行modify1( )之后，n和L都没有发生任何改变；执行modify2( )后，n还是没有改变，L发生了改变。因为在Python中参数传递采用的是值传递方式，在执行函数modify1时，先获取n和L的id( )值，然后为形参m和K分配空间，让m和K分别指向对象100和对象[1,2,3]。m、L只是指向n和L的内存空间，当整体改变时，m、L就指向另外的地方。因此m=2这句让m重新指向对象2，而K=[4,5,6]这句让K重新指向对象[4,5,6]。这种改变并不会影响到实参n和L，所以在执行modify1之后，n和L没有发生任何改变；在执行函数modify2时，同理，让m和K分别指向对象2和对象[1,2,3]，然而K[0]=0让K[0]重新指向了对象0（注意这里K和L指向的是同一段内存），所以对K指向的内存数据进行的任何改变也会影响到L，因此在执行modify2后，L发生了改变。



在看类里面的self.print\_method\_global，这个其实只是一个实例属性，指向一个外部函数，因为全程没有改变这个实例属性，因此id和指向一直保持一致。

进行多进程创建时，self指向一个实例对象，因此在子进程里面会复制该对象，在子进程里面，形成新的对象。当子进程修改复制实例属性时，子进程内的实例属性改变（实例属性指向新的地址），但不影响主进程的实例属性；实例方法在复制时，就自动创建新的指针内存空间，但是指针指向一致，均是该实例方法。

**五、元类、实现ORM**

**5.1 类也是对象**

在大多数编程语言中，类就是一组用来描述如何生成一个对象的代码段。在Python中这一点仍然成立：

>>> class ObjectCreator(object):

… pass

…

>>> my\_object = ObjectCreator()

>>> print(my\_object)

<\_\_main\_\_.ObjectCreator object at 0x8974f2c>

但是，Python中的类还远不止如此。类同样也是一种对象。只要你使用关键字class，Python解释器在执行的时候就会创建一个对象。

下面的代码段：

>>> class ObjectCreator(object):

… pass

…

将在内存中创建一个对象，名字就是ObjectCreator。这个对象（类对象ObjectCreator）拥有创建对象（实例对象）的能力。但是，它的本质仍然是一个对象，于是乎你可以对它做如下的操作：

* 你可以将它赋值给一个变量
* 你可以拷贝它
* 你可以为它增加属性
* 你可以将它作为函数参数进行传递

下面是示例：

>>> print(ObjectCreator) # 你可以打印一个类，因为它其实也是一个对象

<class '\_\_main\_\_.ObjectCreator'>

>>> def echo(o):

… print(o)

…

>>> echo(ObjectCreator) # 你可以将类做为参数传给函数

<class '\_\_main\_\_.ObjectCreator'>

>>> print(hasattr(ObjectCreator, 'new\_attribute'))

Fasle

>>> ObjectCreator.new\_attribute = 'foo' # 你可以为类增加属性

>>> print(hasattr(ObjectCreator, 'new\_attribute'))

True

>>> print(ObjectCreator.new\_attribute)

foo

>>> ObjectCreatorMirror = ObjectCreator # 你可以将类赋值给一个变量

>>> print(ObjectCreatorMirror())

<\_\_main\_\_.ObjectCreator object at 0x8997b4c>

**5.2 使用type创建类**

type还有一种完全不同的功能，动态的创建类。

type可以接受一个类的描述作为参数，然后返回一个类。（要知道，根据传入参数的不同，同一个函数拥有两种完全不同的用法是一件很傻的事情，但这在Python中是为了保持向后兼容性）

type可以像这样工作：

**type(类名, 由父类名称组成的元组（针对继承的情况，可以为空），包含属性的字典（名称和值）)**

比如下面的代码：

In [2]: class Test: #定义了一个Test类

...: pass

...:

In [3]: Test() # 创建了一个Test类的实例对象

Out[3]: <\_\_main\_\_.Test at 0x10d3f8438>

可以手动像这样创建：

Test2 = type("Test2", (), {}) # 定了一个Test2类

In [5]: Test2() # 创建了一个Test2类的实例对象

Out[5]: <\_\_main\_\_.Test2 at 0x10d406b38>

我们使用"Test2"作为类名，并且也可以把它当做一个变量来作为类的引用。类和变量是不同的，这里没有任何理由把事情弄的复杂。即type函数中第1个实参，也可以叫做其他的名字，这个名字表示类的名字

In [23]: MyDogClass = type('MyDog', (), {})

In [24]: print(MyDogClass)

<class '\_\_main\_\_.MyDog'>

**5.2.1 使用type创建带有属性的类**

type 接受一个字典来为类定义属性，因此

>>> Foo = type('Foo', (), {'bar': True})

可以翻译为：

>>> class Foo(object):

… bar = True

并且可以将Foo当成一个普通的类一样使用：

>>> print(Foo)

<class '\_\_main\_\_.Foo'>

>>> print(Foo.bar)

True

>>> f = Foo()

>>> print(f)

<\_\_main\_\_.Foo object at 0x8a9b84c>

>>> print(f.bar)

True

当然，你可以继承这个类，代码如下：

>>> class FooChild(Foo):

… pass

就可以写成：

>>> FooChild = type('FooChild', (Foo,), {})

>>> print(FooChild)

<class '\_\_main\_\_.FooChild'>

>>> print(FooChild.bar) # bar属性是由Foo继承而来

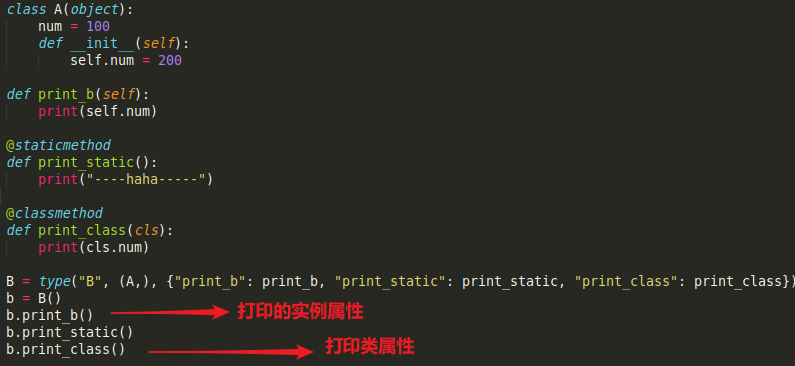
True

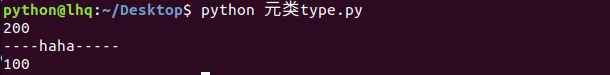
注意：

* type的第2个参数，元组中是父类的名字，而不是字符串
* 添加的属性是类属性，并不是实例属性

**5.2.2 使用type创建带有方法的类**

可以创建实例方法、类方法、静态方法。





**5.3 到底什么是元类**

类创建实例对象，元类创建类；



元类就是用来创建类的“东西”。你创建类就是为了创建类的实例对象，不是吗？但是我们已经学习到了Python中的类也是对象。

元类就是用来创建这些类（对象）的，元类就是类的类，你可以这样理解为：

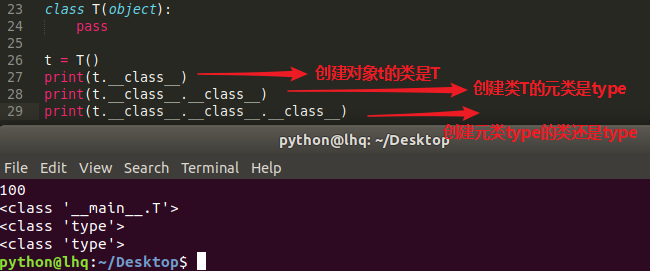
MyClass = MetaClass() # 使用元类创建出一个对象，这个对象称为“类”

my\_object = MyClass() # 使用“类”来创建出实例对象

你已经看到了type可以让你像这样做：

MyClass = type('MyClass', (), {})

这是因为函数type实际上是一个元类。type就是Python在背后用来创建所有类的元类。Python中所有的东西，注意，我是指所有的东西——都是对象。这包括整数、字符串、函数以及类。它们全部都是对象，而且它们都是从一个类创建而来，这个类就是type。



**5.3.1 \_\_metaclass\_\_属性**

你可以在定义一个类的时候为其添加\_\_metaclass\_\_属性。

class Foo(Bar):

pass

Python做了如下的操作：

* Foo中有\_\_metaclass\_\_这个属性吗？如果是，Python会通过\_\_metaclass\_\_创建一个名字为Foo的类(对象)
* 如果Python没有找到\_\_metaclass\_\_，它会继续在Bar（父类）中寻找\_\_metaclass\_\_属性，并尝试做和前面同样的操作。
* 如果Python在任何父类中都找不到\_\_metaclass\_\_，它就会在模块层次中去寻找\_\_metaclass\_\_，并尝试做同样的操作。
* 如果还是找不到\_\_metaclass\_\_,Python就会用内置的type来创建这个类对象。

现在的问题就是，你可以在\_\_metaclass\_\_中放置些什么代码呢？答案就是：可以创建一个类的东西。那么什么可以用来创建一个类呢？type，或者任何使用到type或者子类化type的东东都可以



**5.3.2 自定义元类**

元类的主要目的就是为了当创建类时能够自动地改变类。

假想一个很傻的例子，你决定在你的模块里所有的类的属性都应该是大写形式。有好几种方法可以办到，但其中一种就是通过在模块级别设定\_\_metaclass\_\_。采用这种方法，这个模块中的所有类都会通过这个元类来创建，我们只需要告诉元类把所有的属性都改成大写形式就万事大吉了。



现在让我们再做一次，这一次用一个真正的class来当做元类。

#coding=utf-8

class UpperAttrMetaClass(type):

# \_\_new\_\_ 是在\_\_init\_\_之前被调用的特殊方法

# \_\_new\_\_是用来创建对象并返回之的方法

# 而\_\_init\_\_只是用来将传入的参数初始化给对象

# 你很少用到\_\_new\_\_，除非你希望能够控制对象的创建

# 这里，创建的对象是类，我们希望能够自定义它，所以我们这里改写\_\_new\_\_

# 如果你希望的话，你也可以在\_\_init\_\_中做些事情

# 还有一些高级的用法会涉及到改写\_\_call\_\_特殊方法，但是我们这里不用

def \_\_new\_\_(cls, class\_name, class\_parents, class\_attr):

# 遍历属性字典，把不是\_\_开头的属性名字变为大写

new\_attr = {}

for name, value in class\_attr.items():

if not name.startswith("\_\_"):

new\_attr[name.upper()] = value

# 方法1：通过'type'来做类对象的创建

return type(class\_name, class\_parents, new\_attr)

# 方法2：复用type.\_\_new\_\_方法

# 这就是基本的OOP编程，没什么魔法

# return type.\_\_new\_\_(cls, class\_name, class\_parents, new\_attr)

# python3的用法

class Foo(object, metaclass=UpperAttrMetaClass):

bar = 'bip'

就是这样，除此之外，关于元类真的没有别的可说的了。但就元类本身而言，它们其实是很简单的：

* 拦截类的创建
* 修改类
* 返回修改之后的类

究竟为什么要使用元类？

现在回到我们的大主题上来，究竟是为什么你会去使用这样一种容易出错且晦涩的特性？好吧，一般来说，你根本就用不上它：

“元类就是深度的魔法，99%的用户应该根本不必为此操心。如果你想搞清楚究竟是否需要用到元类，那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要做什么，而且根本不需要解释为什么要用元类。” —— Python界的领袖 Tim Peters

**5.4 元类实现ORM**

**5.4.1 ORM是什么**

ORM 是 python编程语言后端web框架 Django的核心思想，“Object Relational Mapping”，即对象-关系映射，简称ORM。

一个句话理解就是：创建一个实例对象，用创建它的类名当做数据表名，用创建它的类属性对应数据表的字段，当对这个实例对象操作时，能够对应MySQL语句



demo:

class User(父类省略):

uid = ('uid', "int unsigned")

name = ('username', "varchar(30)")

email = ('email', "varchar(30)")

password = ('password', "varchar(30)")

...省略...

u = User(uid=12345, name='Michael', email='test@orm.org', password='my-pwd')

u.save()

# 对应如下sql语句

# insert into User (username,email,password,uid)

# values ('Michael','test@orm.org','my-pwd',12345)

说明

* 所谓的ORM就是让开发者在操作数据库的时候，能够像操作对象时通过xxxx.属性=yyyy一样简单，这是开发ORM的初衷
* 只不过ORM的实现较为复杂，Django中已经实现了 很复杂的操作，本节知识 主要通过完成一个 insert相类似的ORM，理解其中的道理就就可以了

**5.4.2 通过元类简单实现ORM中的insert功能**

class ModelMetaclass(type):

def \_\_new\_\_(cls, name, bases, attrs):

mappings = dict()

# 判断是否需要保存

for k, v in attrs.items():

# 判断是否是指定的StringField或者IntegerField的实例对象

if isinstance(v, tuple):

print('Found mapping: %s ==> %s' % (k, v))

mappings[k] = v

# 删除这些已经在字典中存储的属性

for k in mappings.keys():

attrs.pop(k)

# 将之前的uid/name/email/password以及对应的对象引用、类名字

attrs['\_\_mappings\_\_'] = mappings # 保存属性和列的映射关系

attrs['\_\_table\_\_'] = name # 假设表名和类名一致

return type.\_\_new\_\_(cls, name, bases, attrs)

class User(metaclass=ModelMetaclass):

uid = ('uid', "int unsigned")

name = ('username', "varchar(30)")

email = ('email', "varchar(30)")

password = ('password', "varchar(30)")

# 当指定元类之后，以上的类属性将不在类中，而是在\_\_mappings\_\_属性指定的字典中存储

# 以上User类中有

# \_\_mappings\_\_ = {

# "uid": ('uid', "int unsigned")

# "name": ('username', "varchar(30)")

# "email": ('email', "varchar(30)")

# "password": ('password', "varchar(30)")

# }

# \_\_table\_\_ = "User"

def \_\_init\_\_(self, \*\*kwargs):

for name, value in kwargs.items():

setattr(self, name, value)

def save(self):

fields = []

args = []

for k, v in self.\_\_mappings\_\_.items():

fields.append(v[0])

args.append(getattr(self, k, None))

sql = 'insert into %s (%s) values (%s)' % (self.\_\_table\_\_, ','.join(fields), ','.join([str(i) for i in args]))

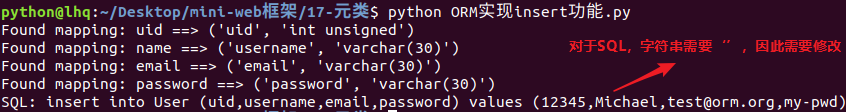
print('SQL: %s' % sql)

u = User(uid=12345, name='Michael', email='test@orm.org', password='my-pwd')

# print(u.\_\_dict\_\_)

u.save()

执行的效果:



**5.4.3 完善对数据类型的检测**

class ModelMetaclass(type):

def \_\_new\_\_(cls, name, bases, attrs):

mappings = dict()

# 判断是否需要保存

for k, v in attrs.items():

# 判断是否是指定的StringField或者IntegerField的实例对象

if isinstance(v, tuple):

print('Found mapping: %s ==> %s' % (k, v))

mappings[k] = v

# 删除这些已经在字典中存储的属性

for k in mappings.keys():

attrs.pop(k)

# 将之前的uid/name/email/password以及对应的对象引用、类名字

attrs['\_\_mappings\_\_'] = mappings # 保存属性和列的映射关系

attrs['\_\_table\_\_'] = name # 假设表名和类名一致

return type.\_\_new\_\_(cls, name, bases, attrs)

class User(metaclass=ModelMetaclass):

uid = ('uid', "int unsigned")

name = ('username', "varchar(30)")

email = ('email', "varchar(30)")

password = ('password', "varchar(30)")

# 当指定元类之后，以上的类属性将不在类中，而是在\_\_mappings\_\_属性指定的字典中存储

# 以上User类中有

# \_\_mappings\_\_ = {

# "uid": ('uid', "int unsigned")

# "name": ('username', "varchar(30)")

# "email": ('email', "varchar(30)")

# "password": ('password', "varchar(30)")

# }

# \_\_table\_\_ = "User"

def \_\_init\_\_(self, \*\*kwargs):

for name, value in kwargs.items():

setattr(self, name, value)

def save(self):

fields = []

args = []

for k, v in self.\_\_mappings\_\_.items():

fields.append(v[0])

args.append(getattr(self, k, None))

args\_temp = list()

for temp in args:

# 判断入如果是数字类型

if isinstance(temp, int):

args\_temp.append(str(temp))

elif isinstance(temp, str):

args\_temp.append("""'%s'""" % temp)

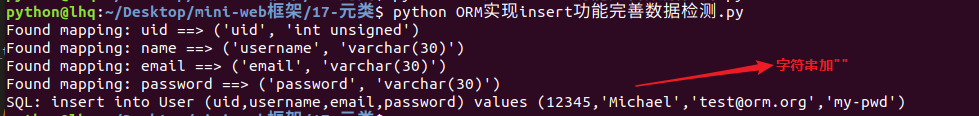
sql = 'insert into %s (%s) values (%s)' % (self.\_\_table\_\_, ','.join(fields), ','.join(args\_temp))

print('SQL: %s' % sql)

u = User(uid=12345, name='Michael', email='test@orm.org', password='my-pwd')

# print(u.\_\_dict\_\_)

u.save()



备注：对于字符串，需要在字符串里面显示’’，则采用3个“”将字符串放进去。

**5.4.4 抽取到基类中**

class ModelMetaclass(type):

def \_\_new\_\_(cls, name, bases, attrs):

mappings = dict()

# 判断是否需要保存

for k, v in attrs.items():

# 判断是否是指定的StringField或者IntegerField的实例对象

if isinstance(v, tuple):

print('Found mapping: %s ==> %s' % (k, v))

mappings[k] = v

# 删除这些已经在字典中存储的属性

for k in mappings.keys():

attrs.pop(k)

# 将之前的uid/name/email/password以及对应的对象引用、类名字

attrs['\_\_mappings\_\_'] = mappings # 保存属性和列的映射关系

attrs['\_\_table\_\_'] = name # 假设表名和类名一致

return type.\_\_new\_\_(cls, name, bases, attrs)

class Model(object, metaclass=ModelMetaclass):

def \_\_init\_\_(self, \*\*kwargs):

for name, value in kwargs.items():

setattr(self, name, value)

def save(self):

fields = []

args = []

for k, v in self.\_\_mappings\_\_.items():

fields.append(v[0])

args.append(getattr(self, k, None))

args\_temp = list()

for temp in args:

# 判断入如果是数字类型

if isinstance(temp, int):

args\_temp.append(str(temp))

elif isinstance(temp, str):

args\_temp.append("""'%s'""" % temp)

sql = 'insert into %s (%s) values (%s)' % (self.\_\_table\_\_, ','.join(fields), ','.join(args\_temp))

print('SQL: %s' % sql)

class User(Model):

uid = ('uid', "int unsigned")

name = ('username', "varchar(30)")

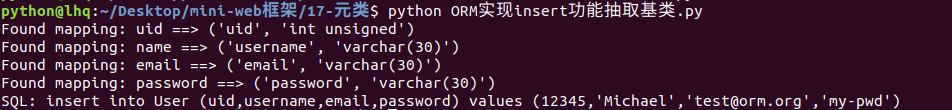
email = ('email', "varchar(30)")

password = ('password', "varchar(30)")

u = User(uid=12345, name='Michael', email='test@orm.org', password='my-pwd')

# print(u.\_\_dict\_\_)

u.save()



**创建类时，首先看自己定义元类没有；自己没有定义元类，看父类是否有元类，父类有元类，则子类用父类的元类；即元类会被继承；**