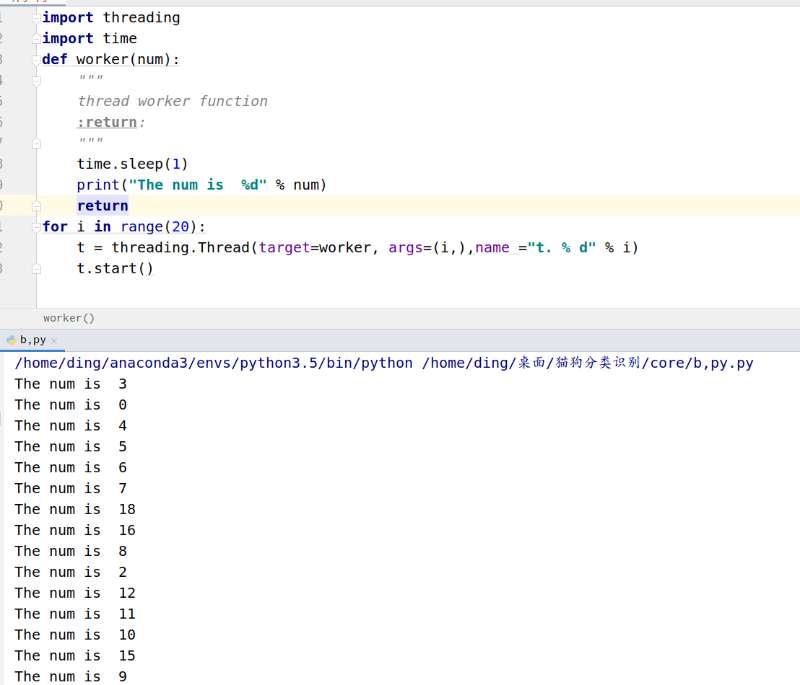
**苏建飞：**

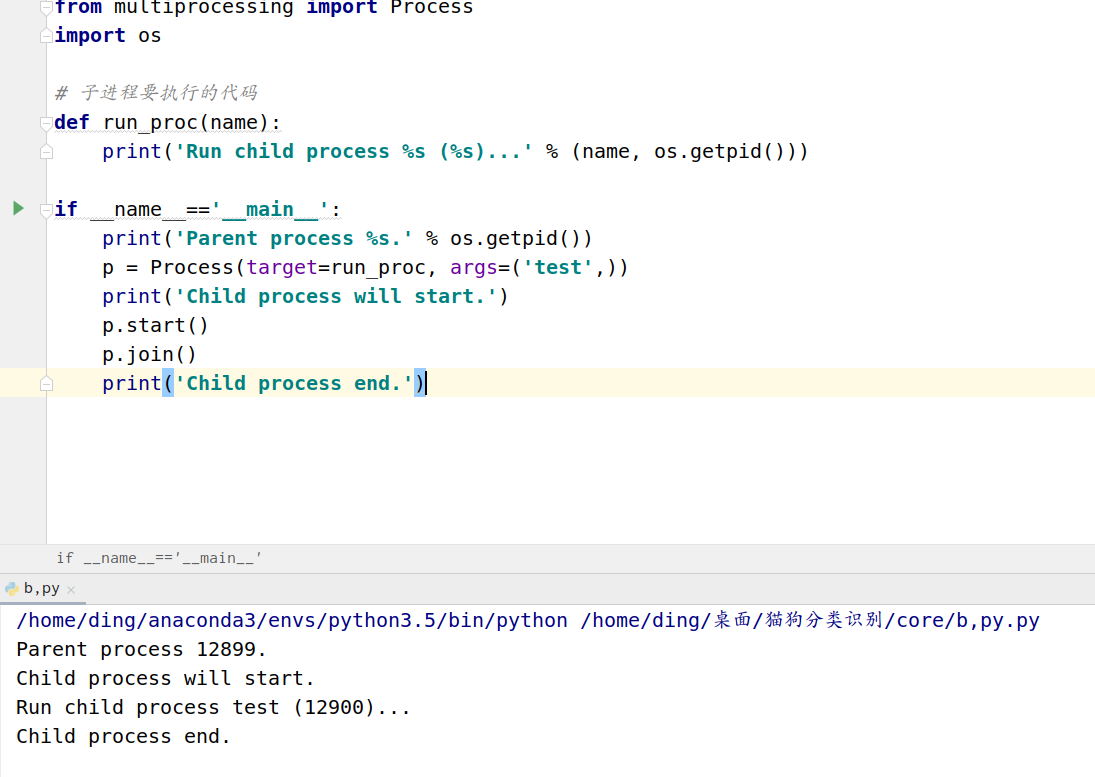
说说进程、线程、协程之间的区别？

gevent是一个第三方库，可以轻松通过gevent实现并发同步或异步编程，在gevent中用到的主要模式是greenlet，它是以C扩展模块形式接入Python的轻量级协程。greenlet全部运行在主程序操作系统进程的内部，但它们被协作式地调度。

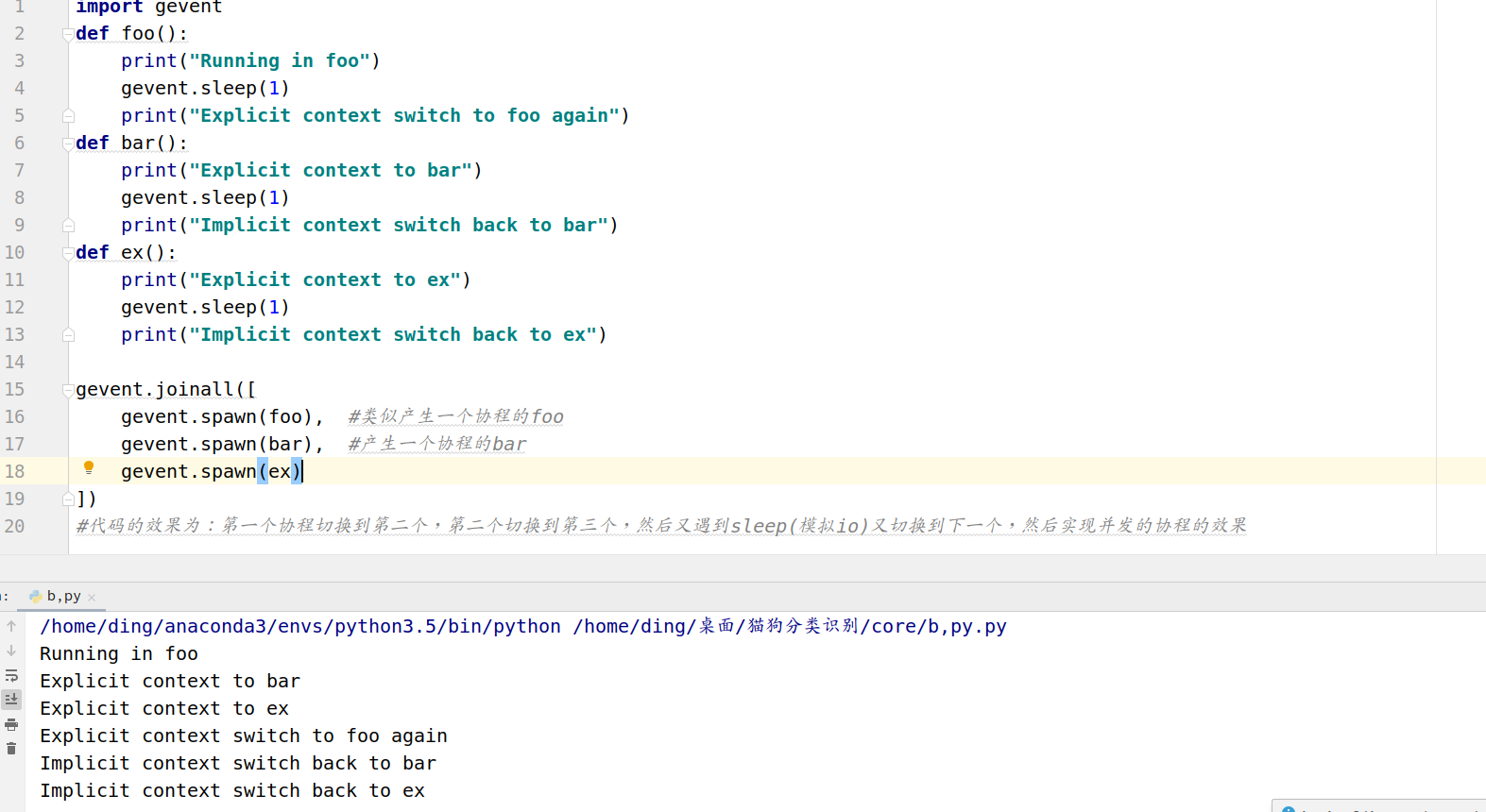
线程：

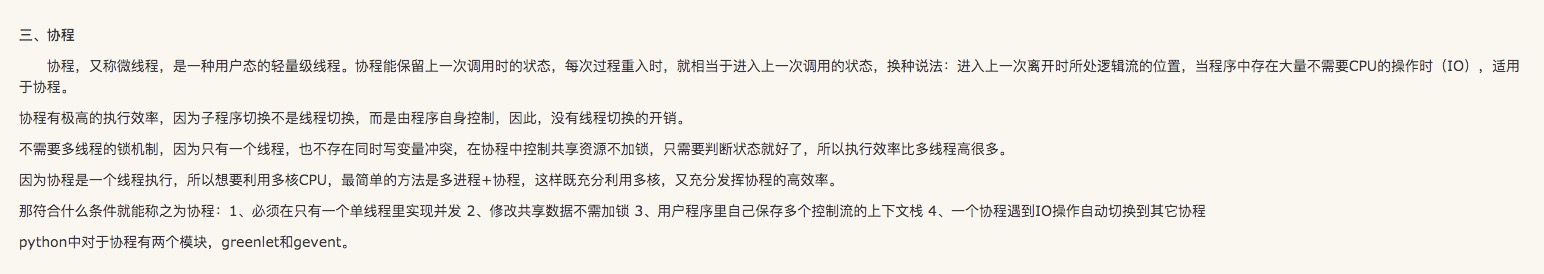
****

**进程：**

****

**协程：**

****

****

补充一点：线程和协程，线程是系统调度，去分配，但是协程是可以用代码控制的。单线程实现异步

**丁昌茂**：

简单介绍一下python的装饰器

装饰器是用于拓展已有函数功能的\_语法，为已有函数添加\_额外的功能，装饰器的作用是在不用更改原函数的代码前提下给函数增加新的功能。

1. 装饰器的几种形式

根据被装饰函数定义的参数和返回值定义形式不同，装饰器也对应几种变形。

无参无返回值

def setFunc(func):

def wrapper():

print('Start')

func()

print('End')

return wrapper

@setFunc

def show():

print('show')

show()

无参有返回值

def setFunc(func):

def wrapper():

print('Start')

return func()

return wrapper

@setFunc # show = setFunc(show)

def show():

return 100

print(show() \* 100)

有参无返回值

def setFunc(func):

def wrapper(s):

print('Start')

func(s)

print('End')

return wrapper

@setFunc

def show(s):

print('Hello %s' % s)

show('Tom')

有参有返回值

def setFunc(func):

def wrapper(x, y):

print('Start')

return func(x, y)

return wrapper

@setFunc

def myAdd(x, y):

return x + y

print(myAdd(1, 2))

2. 万能装饰器

根据被\_\_装饰函数的定义不同，上面细分了四种形式。

但是这四种形式依然不能实现对\_任何函数进行装饰的目的。

思考： 那么能否利用学\_过的技术，实现一种适用于任何形式函数定义的装饰器呢？

def setFunc(func):

def wrapper(\*args, \*\*kwargs): # 接收不同的参数

print('wrapper context')

return func(\*args, \*kwargs) # 再原样传回给被装饰的函数

return wrapper

@setFunc

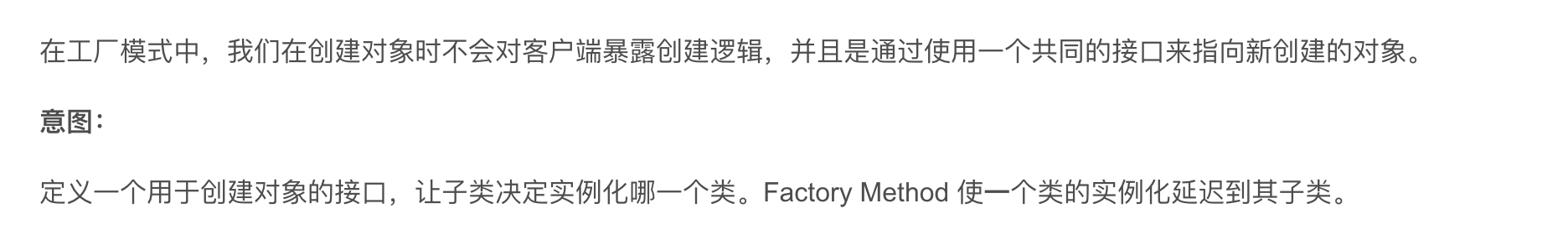
def show(name, age):

print(name,age)

show('tom',12)

\_通过可变参数和关键字参数来接收不同的参数类型。

**陈中原：**

****

\_\_author\_\_ = "Burgess Zheng"

#!/usr/bin/env python

#-\*- coding:utf-8 -\*-

'''

simple Factory Method

'''

class Shape(object):

'''

父类

'''

def draw(self):

raise NotImplementedError

class Circle(Shape):

'''

Shape子类

'''

def draw(self):

print('draw circle')

class Rectangle(Shape):

'''

Shape的子类

'''

def draw(self):

print('draw Rectangle')

class ShapeFactory(object):

'''

工厂模式：暴露给用户去调用的，

用户可通过该类进行选择Shape的子类进行实例化

'''

def create(self, shape):

if shape == 'Circle':

return Circle()

elif shape == 'Rectangle':

return Rectangle()

else:

return None

fac = ShapeFactory() #实例化工厂类

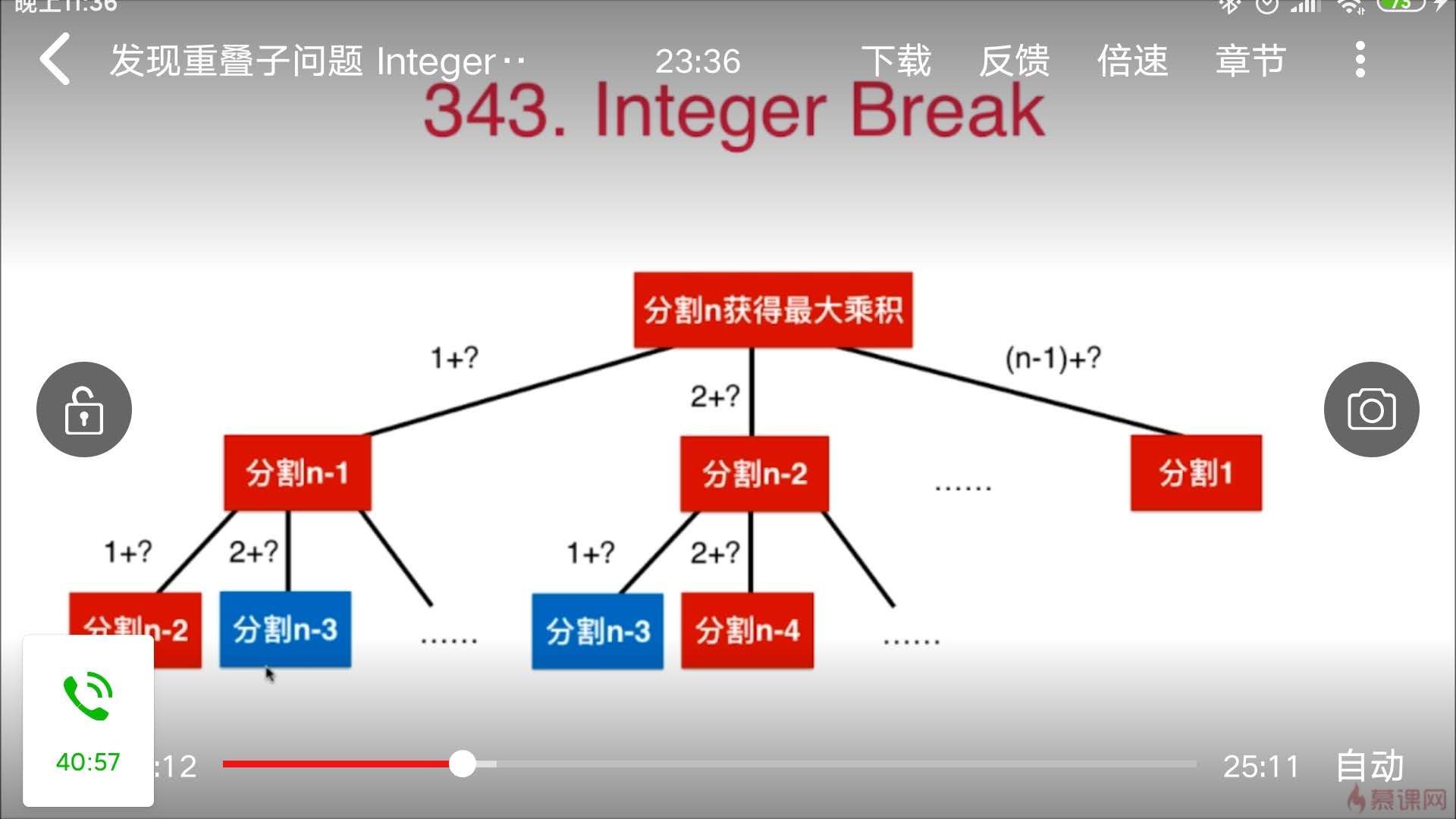
obj = fac.create('Circle') #实例化Shape的Circle子类

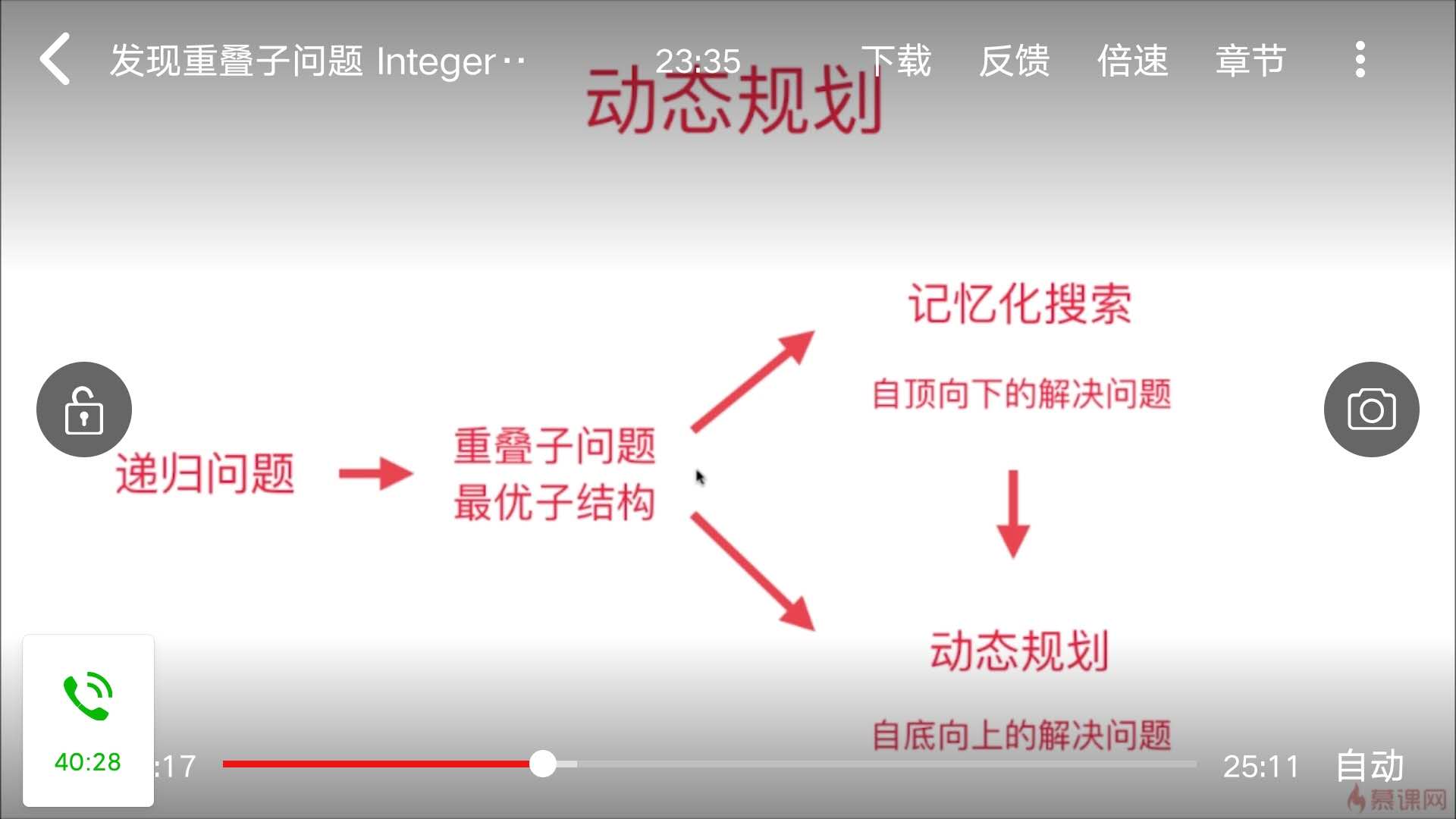
obj.draw()

解释一波：工厂模式，说白了，就是有一个工厂类，工厂类，可以生产出各种类的对象。只要实例化一个工厂类的对象，根据多态，产生不同多想，如果可以封装配合 接口 或者抽象类。会让操作统一，而且客户端不用更改代码。我理解的有点类似数据库表的操作。

**王福成：**

****

****

****