目录

[0、 前言 1](#_Toc5423)

[1. python编程范式种类： 2](#_Toc18344)

[1.1 三种python编程方式 2](#_Toc25981)

[1.1.1 方式一，平铺指令。 2](#_Toc18993)

[1.1.2 方式二，面向过程函数式编程。 2](#_Toc1963)

[1.1.3 方式三，oop编程. 2](#_Toc23021)

[1.2 如何判断需不需要使用oop 2](#_Toc6849)

[1.3举一个实现人的各个功能的写法，具体详见代码附件里面的文件。 2](#_Toc27240)

[1.4 知乎上回答面向过程和oop的区别，参考一下？ 2](#_Toc23097)

[2. 设计模式 3](#_Toc28391)

[2.1 经典23种设计模式 3](#_Toc17198)

[2.2 总共36种设计模式 4](#_Toc4950)

[2.3 设计模式例子大全 5](#_Toc28688)

[2.3.1 菜鸟教程的，java实现的例子。 5](#_Toc5564)

[2.3.2 github的，python例子。 5](#_Toc18140)

[2.4 反模式 5](#_Toc3076)

[3.代码举例 5](#_Toc13861)

[3.1 使用无限复制粘贴扣字low模式来写代码真实例子，详见文件夹。 6](#_Toc353)

[3.2使用三种编程范式完成一个人的功能，详见文件夹。 6](#_Toc24447)

[4、 python里面的oop介绍 6](#_Toc28576)

[4.1 python self是什么，变量前什么时候要加self，方法名第一个参数什么时候加self 6](#_Toc16893)

[4.2 python什么时候用实例方法？ 什么时候用类方法？ 什么时候用静态方法？ 6](#_Toc31769)

[4.3 python怎么申明访问权限？什么时候使用公有 保护 私有属性/方法？为什么要分访问权限？ 6](#_Toc19937)

[5、 万能的oop转化公式，不需要学习任何设计模式和掌握封装技巧。 7](#_Toc23842)

[5.1、针对全局变量加函数无限复制粘贴扣字的写法， 7](#_Toc21183)

[5.2、针对频繁return和大量重复传参的写法 7](#_Toc13368)

[5.2.1、转化公式1 7](#_Toc912)

[5.2.2、转化公式2 7](#_Toc18992)

[5.3 对老代码和新代码的改造 7](#_Toc26765)

[6、 常见问题解答 7](#_Toc17965)

[6.1、是不是所有代码都要用oop？ 7](#_Toc5331)

[6.2、函数和类有什么区别？ 8](#_Toc32062)

[6.3 为什么要由面向过程转为写类，主要原因是写类是可以继承减少代码吗？ 8](#_Toc26132)

[6.4 感觉面向过程写代码更直观，不想用面向对象？ 8](#_Toc10175)

[6.5 讲oop为什么要怼代码中有return的写法? 8](#_Toc1763)

[6.6 面向过程转oop到底能有什么好处，为什么要改变？ 8](#_Toc29433)

[7、 类的封装技巧 9](#_Toc28770)

[7.1 封装时候应该声明哪些实例属性（成员变量）？ 9](#_Toc22679)

[7.2 类名的取名 9](#_Toc20056)

[8一些较好的类名命名方式，附录。 9](#_Toc13030)

[8.1 业务逻辑类 9](#_Toc248)

[8.2 操作DataSource的层 9](#_Toc31743)

[8.3 加工数据的层 10](#_Toc4578)

[8.4 拉取DataSource的层 10](#_Toc22342)

[8.5 进行异步处理的层 11](#_Toc24097)

[8.6集成访问FrameWork的层 11](#_Toc16361)

[8.7 操作View的类 11](#_Toc1337)

[8.8 包含UI上的动作的类 12](#_Toc19518)

[8.9 类命名总结 12](#_Toc28034)

[9、python 代码规范 、pep8规范 （与编程范式无关） 12](#_Toc13619)

[9.1 Python pep8规范 12](#_Toc28759)

[9.2 pycharm一键快速格式化，减少大部分不规范。 12](#_Toc2612)

[9.3 在pycharm中使用注释的方式临时抑制少数警告 12](#_Toc5361)

[9.4 不要在pycharm编辑器配置全局的警告抑制，到时候永远看不到该类型的警告了。 13](#_Toc32041)

[9.5 pycharm设置每行最大字母数 13](#_Toc14052)

# 前言

1. 最重要的事情要放在开头说，不然对文章的阐述会带有抵触情绪。事先声明，我从没有发表过反对面向过程和面向过程是垃圾的论断，我只是要反对在任何场景任何情况下都100%极端地纯面向过程的代码写法，这种编程思维方式在任何公司的python人 员里面都十分之普遍，对于其他语言的开发者这种过于极端的面向过程编程思维方式则不会这么严重。我不反对面向过程，而且我在本文也针对不同场景精确地提供了如何判断使用哪一种编程方式好；同时我要反对只有形没有魂的无效废物类和假oop。本文比较的是 oop + 面向过程混合开发 对比 100%纯极端面向过程；不要为了逃避学习oop，故意强奸民意歪曲成是我要表达 纯oop编程 比 面向过程 + oop混合开发好 这个结论，所以接下来大家千万别故意扯这个（例如故意拿 面向过程有时候也有好处 来作为论点，不理会这种伪观点，不值一驳），此文只对比oop + 面向过程混合开发 对比 100%纯极端面向过程。

1）这里面的编程范式主要是讲oop编程，这是最重要的。本文的经验和结论是经过我纵向比较和横向比较得出来的，就算是培训班老师都不可能有这么深刻的体会。

纵向比较：对比我以前写代码为什么很low 写得很慢 写得很重复 写代码情不自禁找个老代码复制写代码很难开头。

横向比较：使用新的思维方法对接新平台和重构老的对接平台，为什么可以动不动使解析减少60%行、使调度比价减少90%行、使取价格返回给前端减少90%行；为什么可以大幅度减少代码中的重复传参和return几十倍；为什么能使所写的代码更直观地看懂。

2）基于自己的比较和看一些面向过程和oop区别的回答，掌握了面向过程和面向对象编程的最本质区别，并针对区别提炼除了万能oop转化公式，即不需要改变现在代码的思维，只需要按照公式三步转化，就可以达到oo了。使用oop转化公式，写代码很快，编程基本不用想，不学封装技巧和设计模式的前提下，能写模块就能写类，能写项目就能写框架（简单的）。

3）如果继续使用无限复制粘贴扣字的极端面向过程编程的low模式写代码，要么会造成每天很忙，要么会造成半天的任务要花一天完成，虽然工作时间饱和了，工作任务量却不算很饱和。绝大部分工作是瞎忙活，没有用正确的方式。oop就是坦克核武器，面向过程在oop面前是关公耍大刀，会成为炮灰。

# python编程范式种类：

## 三种python编程方式

### 1.1.1 方式一，平铺指令。

从上往下叠加指令，适合简单的独立脚本。不需要和没机会被别的模块导入。

### 1.1.2 方式二，面向过程函数式编程。

适合实现独立的转化功能，基本原理是要实现转化 y = f(x)，适合函数无依赖状态（不需要在多个函数中频繁的传入和return相同意义的参数）。

### 1.1.3 方式三，oop编程.

适合多个函数间需要使用同一个变量，并且需要多实例（如果使在使用面向过程时候需要使用函数频繁的return各种状态/变量由类外使用多个参数来保存这些值和传入这些值，那就是也判断为需要多实例），必须同时满足这两个条件，才使用oop好，否则不需要oop。（但单例模式为了控制灵活的初始化传参和延迟初始化动作，一般也用类的方式）

## 1.2 如何判断需不需要使用oop

网上说的简单用面向过程，复杂的用面向对象，这简直是错误的废话。简单和复杂界定不了，即使是一个简单的查询价格，经过大量平台的才写对比，oop都能比面向过程减少70%行以上的代码，所以用这句话来判断用什么方式来写代码是错误的。只要严格使用上面描述的判断方式，就能很容易知道在什么场景什么时候使用哪种方式好了。具体的第二种和第三种方式在说什么，详细解释看下文，举一个人类的写法为例。

## 1.3举一个实现人的各个功能的写法，具体详见代码附件里面的文件。

第一种是：oop写法，类 加 方法

第二种是：全局变量加函数写法

第三种是：纯函数不要全局变量，依靠无限反复传参和return来实现

第四种是：废物无效类的写法，这是装x的写法，不是因为写了class就会高大上了，由于封装太少了（没有封装），依然是面向过程思维的类。

## 1.4 知乎上回答面向过程和oop的区别，参考一下？

面向对象和面向过程分别是什么？

<https://www.zhihu.com/question/28790424>

面向对象与面向过程最根本的不同是什么？

<https://www.zhihu.com/question/22679662>

面向对象和面向过程的区别有哪些？

<https://www.zhihu.com/question/19835050/answer/41011657>

面向对象和面向过程的相同点和区别是什么？

<https://www.zhihu.com/question/264967643/answer/288905292>

总结一下，大部分回答的都很宽泛，很抽象，不具体，对你从现在的极端面向过程转oop没有什么太大的帮助。

# 设计模式

不需要学习这部分，先按照oop转换公式完成从面向过程到oop的转化。等用多了，再看这里。oop面向对象是设计模式的基石。

## 2.1 经典23种设计模式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 频率 | 所属类型 | 模式名称 | 模式 | 简单定义 |
| 5 | 创建型 | Singleton | 单件 | 保证一个类只有一个实例，并提供一个访问它的全局访问点。 |
| 5 | 结构型 | Composite | 组合模式 | 将对象组合成树形结构以表示部分整体的关系，Composite使得用户对单个对象和组合对象的使用具有一致性。 |
| 5 | 结构型 | FAÇADE | 外观 | 为子系统中的一组接口提供一致的界面，facade提供了一高层接口，这个接口使得子系统更容易使用。 |
| 5 | 结构型 | Proxy | 代理 | 为其他对象提供一种代理以控制对这个对象的访问 |
| 5 | 行为型 | Iterator | 迭代器 | 提供一个方法顺序访问一个聚合对象的各个元素，而又不需要暴露该对象的内部表示。 |
| 5 | 行为型 | Observer | 观察者 | 定义对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知自动更新。 |
| 5 | 行为型 | Template Method | 模板方法 | 定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，Template Method使得子类可以不改变一个算法的结构即可以重定义该算法得某些特定步骤。 |
| 4 | 创建型 | Abstract Factory | 抽象工厂 | 提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们的具体类。 |
| 4 | 创建型 | Factory Method | 工厂方法 | 定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method使一个类的实例化延迟到了子类。 |
| 4 | 结构型 | Adapter | 适配器 | 将一类的接口转换成客户希望的另外一个接口，Adapter模式使得原本由于接口不兼容而不能一起工作那些类可以一起工作。 |
| 4 | 结构型 | Decorator | 装饰 | 动态地给一个对象增加一些额外的职责，就增加的功能来说，Decorator模式相比生成子类更加灵活。 |
| 4 | 行为型 | Command | 命令 | 将一个请求封装为一个对象，从而使你可以用不同的请求对客户进行参数化，对请求排队和记录请求日志，以及支持可撤销的操作。 |
| 4 | 行为型 | State | 状态 | 允许对象在其内部状态改变时改变他的行为。对象看起来似乎改变了他的类。 |
| 4 | 行为型 | Strategy | 策略模式 | 定义一系列的算法，把他们一个个封装起来，并使他们可以互相替换，本模式使得算法可以独立于使用它们的客户。 |
| 3 | 创建型 | Builder | 生成器 | 将一个复杂对象的构建与他的表示相分离，使得同样的构建过程可以创建不同的表示。 |
| 3 | 结构型 | Bridge | 桥接 | 将抽象部分与它的实现部分相分离，使他们可以独立的变化。 |
| 3 | 行为型 | China of Responsibility | 职责链 | 使多个对象都有机会处理请求，从而避免请求的送发者和接收者之间的耦合关系 |
| 2 | 创建型 | Prototype | 原型 | 用原型实例指定创建对象的种类，并且通过拷贝这些原型来创建新的对象。 |
| 2 | 结构型 | Flyweight | 享元 | 享元模式以共享的方式高效的支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。 |
| 2 | 行为型 | Mediator | 中介者 | 用一个中介对象封装一些列的对象交互。 |
| 2 | 行为型 | Visitor | 访问者模式 | 表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变各元素类的前提下定义作用于这个元素的新操作。 |
| 1 | 行为型 | Interpreter | 解释器 | 给定一个语言，定义他的文法的一个表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。 |
| 1 | 行为型 | Memento | 备忘录 | 在不破坏对象的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。 |

## 2.2 总共36种设计模式

过滤器模式：

过滤器模式（Filter Pattern）或标准模式（Criteria Pattern）是一种设计模式，这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。这种类型的设计模式属于结构型模式，它结合多个标准来获得单一标准。

责任链模式：

顾名思义，责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

解释器模式：

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文

空对象模式 ：

在空对象模式（Null Object Pattern）中，一个空对象取代 NULL 对象实例的检查。Null 对象不是检查空值，而是反应一个不做任何动作的关系。这样的 Null 对象也可以在数据不可用的时候提供默认的行为。

Mvc模式：

MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

前端控制器模式：

前端控制器模式（Front Controller Pattern）是用来提供一个集中的请求处理机制，所有的请求都将由一个单一的处理程序处理。该处理程序可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

拦截过滤器模式：

拦截过滤器模式（Intercepting Filter Pattern）用于对应用程序的请求或响应做一些预处理/后处理。定义过滤器，并在把请求传给实际目标应用程序之前应用在请求上。过滤器可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

发布订阅模式： 和观察者模式作用差不多。 虽然两种模式都存在订阅者和发布者（具体观察者可认为是订阅者、具体目标可认为是发布者），但是观察者模式是由具体目标调度的，而发布/订阅模式是统一由调度中心调的，所以观察者模式的订阅者与发布者之间是存在依赖的，而发布/订阅模式则不会。

比如一个常见的官方内置包logging，就是用的观察者模式。具体是logger 和 streamHandler/fileHandler/roatingFileHandler/roatingTimeFileHandler/stmpHandler/httpHandler等各种handler的关系就是被观察者和观察者，logger需要添加各种handler类型实例化后的对象，当触发日志日后会调用所添加的每个handler对象的emit方法，才实现了具体的各种各样的日志行为，不搞清楚这一基本本质，用日志功能时候就会蒙圈，日志的这种设计模式就是秒杀简单的print了，print最多也只是只完成了streamHandler的功能。在生产项目里面到处大量的print，就是很low（不好调整级别屏蔽和显示日志，以及不能显示打印日志所在行和实现各种日志扩展的handler的功能）。

缓存模式：也可以叫延迟计算模式。可以缓存方法的结果 指定时间或次数，而不是每次调用都进行计算。用这个可以代替实现一系列更细粒度的单例模式。

对象池模式：我们将对象存储在一个池中，当需要时再次使用，而不是每次都实例化一个新对象。（非享元模式，是类似mysql这样的连接池的实现），一般通过python上下文来实现借 和还的动作，避免每次使用前后都频繁摧毁和创建一个对象，达到节约大量cpu和io的目的。

委托授权模式：允许对象组合实现与继承相同的代码重用。重点是需要实现\_\_getattr\_\_方法，可以一键将被组合的对象的方法，转化成自己类的方法，无需大量重复自己写一个方法名，然后该方法名里面去调用被组合对象的方法，可以大幅度减少代码。但是是代码运行时改变代码行为，在ide补全方面很吃亏，无法自动提示和一键补全被组合对象的方法名和属性名，因为ide只能自动补全一些静态的死语法。

## 2.3 设计模式例子大全

2.3.1 菜鸟教程的，java实现的例子。 <https://www.runoob.com/design-pattern/design-pattern-tutorial.html>

### 2.3.2 github的，python例子。

<https://github.com/faif/python-patterns>

## 2.4 反模式

1）极端面向过程，从不写类，不了解oop或者排斥oop。

2）即使写了类，但所有场景都写的是一种几乎是纯静态工具类的类，说具体点就是没有或者只要很少的实例属性(成员变量) 没有或者很少有实例方法(成员函数)，所有全都是静态的，简直是个废物类，写得没有任何作用，因为没有封装的内涵，那就是没有面向对象的特性，违反oop。在py里面这样的类由类名加静态属性和静态方法名，没有任何作用，这种类 直接携程模块文件 加函数就可以替换了。

3)现在写得函数全部100%是公有类型，动不动一个文件写几百个函数，还全都是声明成公有的，调用的时候很难搞清楚哪些是入口函数或者需要被外部直接调用的，脑袋里没有访问权限意识。

4）非常喜爱经常把一个函数写几亿行，由于最小复用粒度是函数/方法，写太长了，有不同点但要复用就必须把整个函数复制扣字了。如果使用类的方式，能够把一组相关的方法放到一个类里面，同时区分好 私有 保护 公有，计时写再多的方法，也不会导致代码太难看懂。

1. 由于不掌握oop，这样的写法很容易情不自禁的复制整个文件几百次然后扣字（因为大部分相同，但还有一些不同点，所以造成需要这样。要是完全相同，傻瓜都不会去无缘无故去全盘复制一次。文中做出解释，为什么会造成这种写法，以及贴出一些这种方式的真实的代码片段，描述如何使用oop转化公式轻松一键去重，只使用10% ~ 50% 的代码实现原来的功能），或者复制整个几百行片段然后扣字。给这种反模式取个最贴切的名字就是无限复制粘贴扣字low模式。
2. 没有封装复用的意识，最多是看了网上例子，所以只有redis mongo这种通用工具的类的封装，没有业务流程级别的复用，业务流程上的复用才能大幅度的减少几万行代码。

# 3.代码举例

## **3.1 使用无限复制粘贴扣字low模式来写代码真实例子，详见文件夹。**

## 3.2使用三种编程范式完成一个人的功能，详见文件夹。

# python里面的oop介绍

## 4.1 python self是什么，变量前什么时候要加self，方法名第一个参数什么时候加self

对于变量：

变量前面加了slef的变量就是实例属性，这样的属性可以在类里面的各个实例方法中通过访问self来访问这些变量，例如self.name。加了self的变量相当于模块级写法的全局变量，在类中每一个实例方法都能访问。可以少在一组方法里面反复传参和return，根据这条规则就能很容易判断变量前需不需要加self。类中的self的变量太少了就是封装不彻底，如果玩的极端，类里面没有任何self前缀的变量，就不是oop了。一定要多用self变量，但不能把本应该是一个局部变量的东西加self，这样很坑，因为别的地方不需要去使用这个变量，弄成这样会蛊惑人心。同时，实例属性应该首次出现在\_\_init\_\_方法中，不要突然的首次出现在其他方法中，这样如果没有调用该方法就使用这个属性，代码运行时候会发生严重错误，提示没有这个属性，这一规范可以通过代码静态检查来证实。

对于方法：

对于实例方法其他语言都是隐式的，不需要在方法第一个参数写上this，就能在方法内部使用this来访问实例属性，有的语言也可以不需要显式的通过this来访问实例属性。在python里面，如果方法里面需要访问实例属性，那么方法的第一个参数就必须是加个self，self只是表示对象本身，叫什么都不要紧，如果没有在方法上加上staticmethod或者classmethod装饰器，那么任何第一个参数就是代表了实例本身。这一点不需要死记硬背，先写方法内部，如果方法内部需要访问实例属性，自然会去想到self，如果方法体写完了，发现还没用到self，那方法应该申明为静态方法，静态方法的第一个参数不是代表实例本身。

## 4.2 python什么时候用实例方法？ 什么时候用类方法？ 什么时候用静态方法？

什么都不用想，先写函数体就是了。

写完后发现需要访问实例属性，那么就需要使用实例方法。

写完后发现只访问了类属性（类属性就是其他语言静态成员变量），没访问实例属性，那么方法就应该申明为类方法，加个classmethod装饰器申明，第一个参数代表类本身，一般默认叫cls，cls和self同理叫什么都不要紧，重要的是位置在第一个的形参，不是普通的方法传参，从第二个才是代表你要传的参数。

写完后发现没访问类属性也没访问实例属性，那就声明成静态方法，加个staticmethod声明。

类方法和静态方法是可以达到同样的目的的，静态方法的方法体中可以使用所在类的类名 加 来访问泪水属性，例如类方法中访问类属性xx是 cls.xx,假设类名是A，当使用静态方法时候在方法体中使用A.xx也可以访问类属性，主要区别是前者代表他需要访问类属性，有着个意图。另外就是cls永远代表当前类，就断把类名由A修改成B，也不需要修改方法体里面的代码。如果使用的是静态方法，类名变化了，方法体里面访问类属性的类名也需要修改。

## 4.3 python怎么申明访问权限？什么时候使用公有 保护 私有属性/方法？为什么要分访问权限？

1） 在方法或属性前面没有下划线是公有方法/属性，一个下划线是保护方法/属性，两个下划线是私有方法/属性

2）入口方法或需要被类外部直接调用的方法，声明成公有；不需要被类的外部直接调用，没有任何机会被外部直接调用的，声明成保护或私有。保护与私有的区别是，保护方法或属性不能被子类直接访问，如果你需要在子类中访问，不要弄成私有。

在类外部访问保护属性是a.\_x，能访问但是pycharm会对这样的访问产生一个警告；访问私有属性a.\_\_y是错误的会报属性不存在，应该是写a.\_\_A\_y才能找到私有属性y，这样的访问也会产生警告。如果在类外需要访问这里的x和y早就应该声明成公有访问权限。

3）python里面的访问权限是很弱的，不会像java一样强制访问会导致编译报错。但还是应该分权限的，不然你写100个方法，全都是公有的，调用这个类会非常抓狂，全都是公有意味着所有方法都希望被外部直接调用，而且pycharm自动补全会补全弹出这些所有方法，造成调用者很困惑，需要去认真的看类是怎么实现的，需要去思考他应该调用哪些方法。所以权限不是为了安全，而是为了给调用者一个非常清晰的代码意图。权限怎么可能是为了安全，如果有人故意想搞破坏直接删除代码乱改代码，光靠申明几个访问权限又能起什么用。

# 万能的oop转化公式，不需要学习任何设计模式和掌握封装技巧。

经过长期的实践和对比代码形式，得出的万能转换规律，结合附件代码文件的例子进行理解。

现在的面向过程编码思维不需要做任何变化，使用公式三步转化进行转换。

## 5.1、针对全局变量加函数无限复制粘贴扣字的写法，

1） 模块级降级为类名

2） 全局变量改成实例属性

3） 函数改为方法

## 5.2、针对频繁return和大量重复传参的写法

### 5.2.1、转化公式1

0）在脑袋中重构，把写法形式改成全局变量加函数的写法，此时不用担心全局变量是唯一的，不用大量重复传参和return，所有需要传参和return的都想象成使用全局变量和操作全局变量。

1） 模块级降级为类名

2） 全局变量改成实例属性

3） 函数改为方法

后面三个步骤是一样的。全局变量变为实例属性后，每次实例化后每个对象的实例属性都是互不干扰的。每个对象可以看作为一个模块级写法的 模块文件的无限深拷贝。

### 5.2.2、转化公式2

1） 新增一个类

2）把重复传参和return的形参，全都去掉，改成实例属性

3） 函数改为方法。

## 5.3 对老代码和新代码的改造

对任何面向过程写法，使用转化公式，一定就可以修改成oop，然来的代码思维不需要做任何改变，只要按这个公式就可以改造。（前提是满足需要使用oop的两个条件，才需要改造）

对新写的代码，也可以按然来的想法写，说的是在脑袋里面那么写（不然真在ide中那么写，再修改，浪费花时间），然后按照此公式转化后写在ide里面。

最重要是理解： 命名空间 全局变量 实例属性 多实例需求 函数和方法 的关系，搞清楚了，写oop十分之简单，不会造成极端面向过程的曲折写法。

# 常见问题解答

## 6.1、是不是所有代码都要用oop？

答：不是，按照上面的方式判断用那种方式好，目的是要简单代码少就可以，便于维护扩展。反对无效废物面向过程的类和写成类后没有好处的代码。

## 6.2、函数和类有什么区别？

没有区别，就像问人和走路有什么区别，猪和吃饭有什么区别，问得牛头不对马嘴的伪问题，函数和方法才可以比较。类（对象）和模块才有可比性，必须要搞清楚原因，不然脑袋中有这个比较的想法那就不可能写得了oop。类的实例化后的每个对象可以看成是python模块的深拷贝，可以看成是对模块的无数次复制，但他们的全局变量互不干扰。

面向过程是 文件模块名.eat(狗的名字，sheet)

oop是 狗.eat(sheet)

## 6.3 为什么要由面向过程转为写类，主要原因是写类是可以继承减少代码吗？

1）oop最重要的特点是封装，封装可以减少无数return和重复传参；允许全局变量加函数的直观思维写代码，但通过对类的实例化很自然解决了多实例的需求，每个实例的属性属性都是互不干扰的。

2）一个类没有被继承不代表写它就没意义了，继承只是面向对象的第二个特点。在设计模式里面，继承用得也不是最多，主要是组合用的最多。继承是is-a ，组合是has-a。例如人用手机打电话发短信浏览网页，定义了一个电话有发送短信和浏览网页的功能，如果很执着于写类就是为了继承可以减少代码这个概念，会造成把人类继承手机类，人就可以发短信了，但这样做人成了手机的一个子类了，这很扯淡讲不通，子类继承父类，子类一定是父类的一个子集（mixin类除外,这是一种形式上继承，意义上是组合的），如界门纲目科属种范围一步步缩小。正常应该是将手机类（对象）组合到人类，人通过手机这个对象来实现打电话 浏览网页的目的。

## 6.4 感觉面向过程写代码更直观，不想用面向对象？

面向过程纯函数只是实现y=f（x）的转换，写起来才是不直观，需要很复杂的return和传参和内外保持各种状态，面向对象才是直观，写代码才叫快，典型区别的可以看看面向过程和oop来实现的 五子棋游戏、 蚂蚁和虫子游戏，里面的面向过程实现十分曲折，可以看一看这两个例子观察难度差别到底在哪。说面向过程写代码直观，主要是c语言中毒太深，情不自禁任何场景都极端面向过程，脑瓜转不过弯来造成的。

## 6.5 讲oop为什么要怼代码中有return的写法?

1）好好想想，代码中大量在一组函数间重复传参和频繁return，为什么要这样写？

2）为什么不可以弄成全局变量 + 函数的写法，每个函数都操作全局变量，就不需要吧东西return来return去，在函数间传来传去了。就仅仅因为 全局变量 + 函数解决不了动态初始化传参和同一代码块下（或者叫当前解释器，或python虚拟机）全局变量只有一份的矛盾，满足不了实现多实例的需求， 从而造成需要弄成反复return的写法绕开全局变量只有一个的问题，这样太low了。当你需要这么反复写return的时候，这种时候按照上文的 判断什么时候需要使用oop的法则，你已经需要使用oop了，只不过你在用low模式--靠反复的return来曲折的模拟多实例。

3)所以从这个角度判断需不需要使用oop，oop和面向过程哪个好，不需要从宏观上讲什么虚无缥缈大道理和概念，仅从代码的大致形式上就能判断出来，这很容易判断，一点都不需要纠结。

## 6.6 面向过程转oop到底能有什么好处，为什么要改变？

好处就是三大特点，封装 继承 多态，和使用设计模式更搭（如果极端排斥oop，绝大多数设计模式套不上，使用不了）。上文说明了详细原因和代码附件中列举了具体的例子，下面再说一次。

1. 封装能以很直观的思维写代码，方法来操作成员变量好似函数来操作全局变量一样简单，写代码时候思维很直观，但通过类的实例化解决了全局变量只有一份的问题。
2. 继承能使用很少代码来完成不同点，多态能加强调用处的统一性，而不是面向过程为了实现不同点，新的函数老是起一些杂七杂八的函数名，造成调用形式没有统一。
3. 使用oop不光可以使实现代码本身更为简单，在调用处也能更为简单。代码本身实现上，由于操作实例属性类似是在操作全局变量，不需要思考传参和return，写法和想法都非常直观。在代码被调用处，由于oop只需要在类外写一个变量代表实例化后的对象操作各种方法就可以，不需要在面向过程时候，需要在调用处写一堆中间状态的变量然后传给各个函数然后保存新的结果后传给下一个函数，很烦人。
4. 几乎可以使任何代码减少50%行以上（亲身实践对比过，重写多个对接平台的价格计划解析），特殊的场景和地方，例如多平台对接开发，有的文件最高能使代码行数减少90%行。
5. 使命名空间能多一级，不用oop，代码的命名空间只有模块级，使用oop代码里面命名空间能增加一级，模块级和对象级。模块级做命名空间缺点很明显，模块在解释器中只有一份，模块里面的全局变量在解释器也只有一份，导致你经常不敢写也不能写全局变量，会造成曲折的频繁return传参的方式来实现业务。对象级的命名空间至关重要，第一点是他能把一组函数挂在一个类名下，是代码函数看起来更紧凑，不然一个文件一下子写几百个函数，不知道哪些函数式有关系的一组；第二点是最重要的，对类的每次实例化后，每一个实例都有独立平等的命名空间，改变每个实例的成员变量都不会影响其他实例的成员变量，使用类＋ 多实例，给予你使用全局变量 + 函数的直观编程思维方式，这样写法非常快，代码思维不会很曲折。
6. 类就是类型，类型就是类，写几十万行代码还只有0个类，那你代码除了python内置的 数字类型、 字符串、字典、列表、元祖这些最基本的类型外，如果你导入了三方库，会增加例如Session类 Response类，除此之外，没有任何自己的类型。对类型不敏感，爱造成代码错误；没有类型意识，会造成在使用三方库时候稀里糊涂的，python不需要显式申明类型，知道每一个变量的类型就更为重要了。

7）python有大量丰富的三方包，但哪个包是绝对极端的函数写的，常见的三方包哪一个包里面低于了10个类，几乎没有；有了c语言，还发明c++语言那不是闲着没事干了；现在最好的几种语言都是非纯面向过程编程的，要么强制类要么是混合的。所以从这些最牛的大神来看，大家不需要质疑这个，没有理由需要顽固的坚持纯面向过程。

# 类的封装技巧

前提还是要判断需不需要使用类，不需要就不要写类了。

要先理解好oop和面向过程比解决了哪些弊端，oop后有什么好处，就能封装。初步的封装依然是按上面的转化公式来封装就满足绝大多数要求了。

一些具体的技巧：

## 7.1 封装时候应该声明哪些实例属性（成员变量）？

实际编程写得类，肯定不是像我举例的描述一个人，有身高 体重 姓名 年龄 性别 这些东西，把他们弄成实例属性，实际编程中没这么直观的对象来很容易抽象。

所以不需要思考这个，不需要想想有什么实例属性。直接先写方法就是了，如果涉及到变量需要跨方法访问就声明为成员变量。（不要通过反复return 传参来弄这种变量）；但不要全部把局部变量都弄成实例属性了，这样也很坑。

## 7.2 类名的取名

类名和python模块名一般都应该是名词，有的携程动词了，例如类名叫GetPrice,这样写第一行时候就已经失败了50%了，导致代码很难写下去。类是对象的模板应该是一个名词，方法才用动词。类名一般上多使用名词，方法名一般多用动词和助动词，这是因为类名要抽象出拥有某种职责的某物，所以多使用名词。

# 8一些较好的类名命名方式，附录。

链接：<http://qiita.com/KeithYokoma/items/ee21fec6a3ebb5d1e9a8>

原作者：KeithYokoma

译者：dssunxun

除了名词，另一方面，声明接口时，我们一般使用形容词来作为接口的名称（例如：Iterable、Closeable等）。

通过使用形容词，可以较好的注明类所拥有的特性。

## 8.1 业务逻辑类

即一般所谓的Model层，该层有着各种不同功能的类。

一般性的，给该层的类起名都是Model或是Manager这样的名字，而随着业务的增多，该层也会变的越来越肥大。

实际上在Model层中也有各种各样的层，通过给它们起好名字明确它们的功能，然后组合起来建立起业务逻辑

## 8.2 操作DataSource的层

大致上就是拥有某些I/O的基础功能(增删查改、request等)的类，比如说，拥有操作DB逻辑的类、拥有通过

HTTP通信获取Response的类

|  |  |  |
| --- | --- | --- |
| 名称 | 补充说明 | 例 |
| Client | 类似HttpClient这样的，在有Server-Client的含义的情况下使用 | TwitterApiClient, QiitaApiClient |
| Gateway | 作为访问API时的网关时使用 | TwitterTimelineGateway, QiitaAccountGateway |
| Store，Storage、Registry | 访问数据库，在磁盘进行数据持久化时使用 | FavoriteSettingStore, DataStorage, ConfigRegistry |
| Cache | 缓存时使用 | TimelineCache |
| Log | 日志。或是存储操作历史记录的路径 | UsageLog |
| History | 存储历  史记录的路径 | UsageHistory |
| Configuration, Preference, Setting | 存储设定数据的路径 | TimelineConfiguration |

有必要的话，可将I/O的基本功能各自分开实现（可提供一个统一管理的接口或是一开始就创建个统一管理的类来操作）

|  |  |  |
| --- | --- | --- |
| 名称 | 补充说明 | 例 |
| Logger | 执行日志操作 | UsageLogger |
| Cleaner, Sweeper | 清除数据时使用 | CacheCleaner, CacheSweeper |

## 8.3 加工数据的层

|  |  |  |
| --- | --- | --- |
| 名称 | 补充说明 | 例 |
| Filter | 筛选数据时使用 | TimelineFilter |
| Extractor | 从某个数据中抽出其他数据 | MessageExtractor |
| Formatter | 格式化某个数据输出为其他数据 | MessageFormatter |
| Collector | 收集数据 | AnalyticsDataCollector |

## 8.4 拉取DataSource的层

取得数据、存储为Cache、然后再将其返回给Controller和Presenter层的层。该层的Model从使用者看来，无需在意获得

的数据是否是从Cache而来。

|  |  |  |
| --- | --- | --- |
| 名称 | 补充说明 | 例 |
| Provider | 将上面提到过的DB、http通信、Cache等封装化后的上位层。或是Android中的ContentProvider | TwitterTimelineProvider |
| Manager | 管理数据 | AccountManager |
| Loader | 读取数据 | TimelineLoader |
| Logger | 写日志、或是提供访问Log的抽象层 | RecentUsageLogger |
| Configurator | 存储设定的默认值、将某种数据自动的保存到设定 | FirstSettingConfigurator |
| Migrator | 处理当版本升级等数据结构发生变化时的逻辑 | UserDataMigrator |

## 8.5 进行异步处理的层

|  |  |  |
| --- | --- | --- |
| 名称 | 补充说明 | 例 |
| Job, Task, Runnable, Executable | 统一处理异步操作 | UploadJob, MigrationTask |
| Runner, Executor, Worker | 执行被给予的Job和Task | UploadJobRunner, MigrationTaskExecutor |
| Aware | 拥有同步操作相关的某些Context，表示在其管理下的接口（Spring Framework有使用） | ApplicationContextAware |

## 8.6集成访问FrameWork的层

基于Facade模式、提供通向其他的Framework和SubSystem接口的层

|  |  |  |
| --- | --- | --- |
| 名称 | 补充说明 | 例 |
| Facade | 实现Facade模式 | BoundServiceFacade |
| Service | 将兼容性的实现给封装化，并实现对各种功能的访问的层 | ApplicationControllerService |
| Resolver | 根据用户环境进行Routing处理的层 | ContentResolver |

## 8.7 操作View的类

即一般上的Controller层，根据所用的Framework有所不同，大致上一般会命名为如下。

严格来讲，与Presenter有些许的不同，在此将其包含进去

|  |  |
| --- | --- |
| 名称 | 补充说明 |
| Activity | Android中使用 |
| Fragment | Android中使用 |
| ViewController | iOS中使用 |
| Controller | MVC中的Controller |
| Screen.Presenter | MVP中的Presenter。Mortar中使用 |
| Window |  |

## 8.8 包含UI上的动作的类

抽象特定的操作，并将该操作想执行的处理综合起来的类。也可作为接口名称使用

|  |  |  |
| --- | --- | --- |
| 名称 | 补充说明 | 例 |
| Action | 表示操作 | SubmitAction, CancelAction |
| Dispatcher, Handler | 接受操作执行处理 | SubmitActionDispatcher, UserActionHandler |
| Listener, Watcher | 监视操作。Observer模式的实现名 | ClickListener, TextEditWatcher |

## 8.9 类命名总结

因语言和框架带来的作法和规则的不同，并没有什么一个最好的方法。设计和重构时，在想好了

如何分层后，如何命名也非常重要。在上面我列举了很多，但是直接用有名的设计模式和框架中用到的名字

有可能也比较好。所以尽可能的清楚地起个能把一个类的作用给表现出来的名字吧

# 9、python 代码规范 、pep8规范 （与编程范式无关）

## 9.1 Python pep8规范

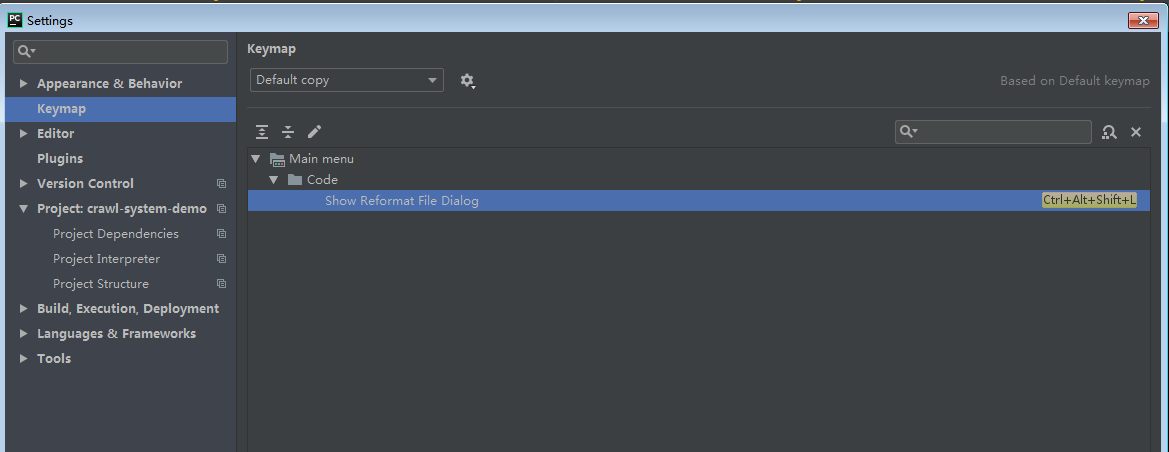
Pep8规范网址 <https://blog.csdn.net/ratsniper/article/details/78954852>

可以看这些规范介绍，但不需要强行记住。

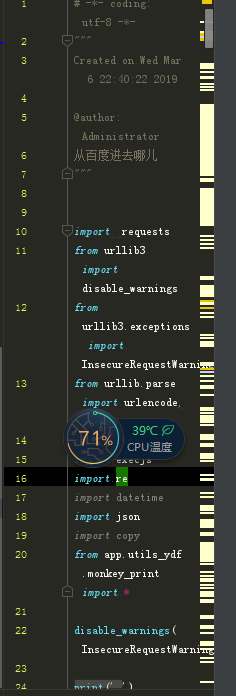
在pycharm ide中，右边滚动条有黄色,就是不规范，pycharm就包括pep8。

## 9.2 pycharm一键快速格式化，减少大部分不规范。

使用这个组合键可以见见减少大部分pep8的规范警告



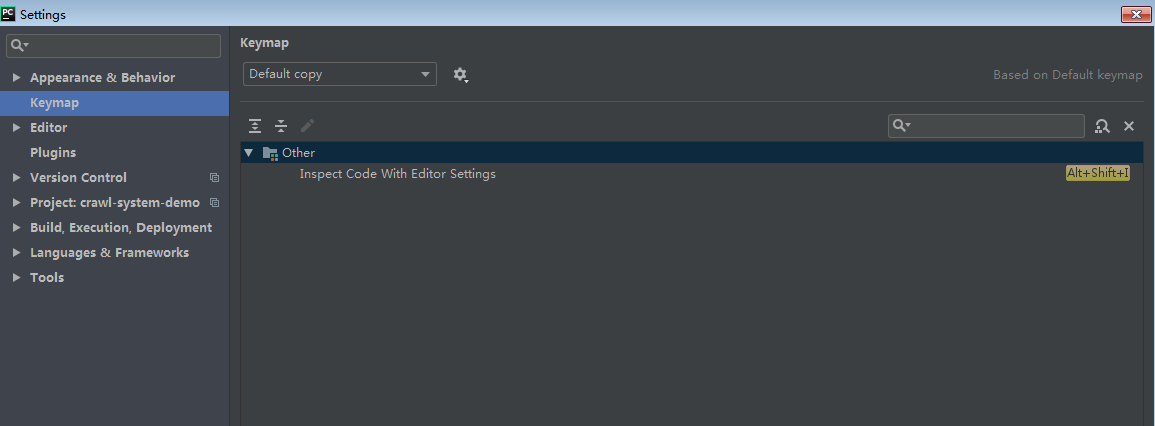
下面这样的吓人，全是黄色连成一片。



## 

## 9.3 在pycharm中使用注释的方式临时抑制少数警告

例如try except，一般要精确捕获异常类型，不建议直接捕获最宽泛级的Exception错误，这是不好的习惯。但有时候出错的类型可能有很多种，而代码又确实是不需要不想关心到底是怎么造成错误的，是可以捕获最宽泛的Exception错误，这时候可以使用临时的注释来抑制警告，右边的滚动条黄色会消失。

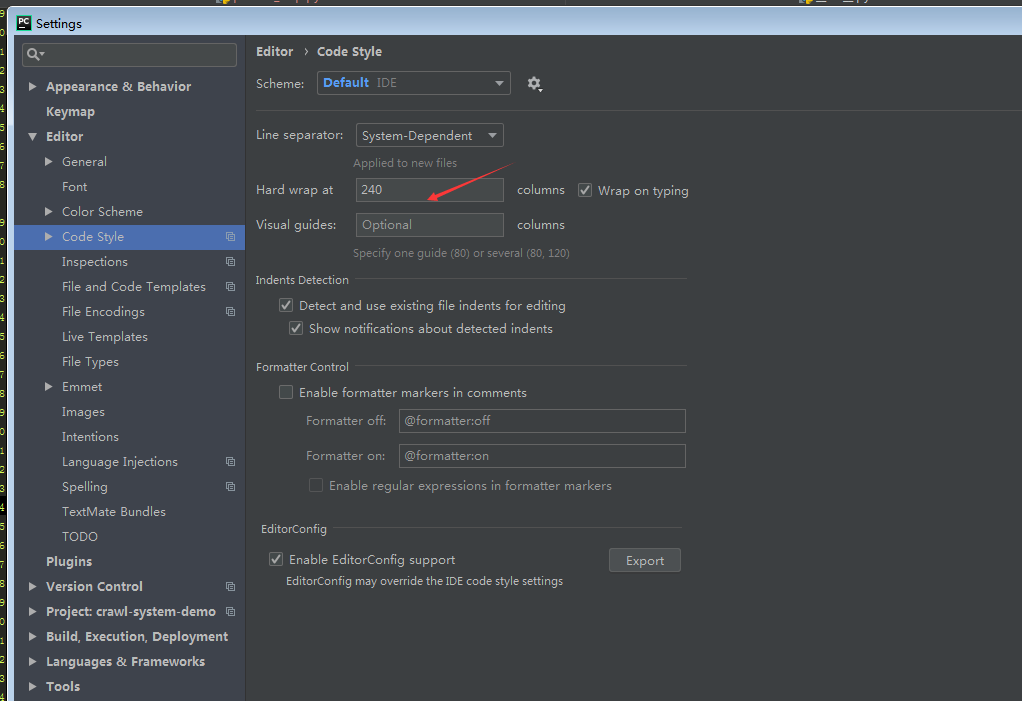
如 # noinspection PyBroadException 可以临时抑制这个警告。 

## 9.4 不要在pycharm编辑器配置全局的警告抑制，到时候永远看不到该类型的警告了。

有些警告是规范问题，有些警告可以加深对语言的理解，例如没有访问成员变量和调用其他实例方法，却没有使用静态方法，这会产生警告，加深对oop的理解。例如 成员变量首次出现不是在\_\_init\_\_里面，而是其他方法里面，这样的成员变量很危险，如果没有调用那个方法就开始访问这个成员变量，会造成严重错误，报属性不存在。

## 9.5 pycharm设置每行最大字母数

有时候，代码的一行很长（例如变量代表一个url字符串），使用\折行后，再次复制它还原为原来的内容很麻烦。修改为最大240个。

只要不把一行代码逻辑写很长，故意使用分号一行写多个语句，故意极限压缩代码行数，只有少数变量内容超过80字母没太大毛病。

目前在不设置全局的配置来使警告消失，通过一键格式化和临时抑制小部分警告，和把最大行数设为240，做到了所有代码0 warning。其他代码都是100多个警告，吓人。警告太多了就会破罐子破摔，养成无视警告的习惯了。