## **一. 基本信息**

## **项目介绍**

我们经常会遇到需要解析命令行参数的场景。如果没有趁手的工具，我们可以自己写一个，自己想办法处理传给main函数的参数。

Args是一个经典的编程练习题，复杂度与我们日常开发中接触到的需求大致相当，能很好地体现出程序员的熟练度。

## **教学目标**

* 准确框定需求范围
* 把需求拆解成明细的任务列表
* 练熟测试驱动开发的节奏

## **学习时长**

7小时

## **技能点**

* 单元测试
* 任务拆分
* 测试驱动开发
* 面向对象设计
* 重构

## **能力目标**

* 用测试用例描述和沟通需求
* 把需求拆解成可以逐步开发的任务列表
* 基本的面向对象设计
* 识别代码坏味道
* 重构的基本节奏和原则

## **二. 项目导学**

## **项目介绍**

“Args”也是一个经典的编程练习题。“鲍勃大叔”Robert Martin在《Clean Code》的第14章里介绍过这个题目。

### **初始需求**

我们经常会遇到需要解析命令行参数的场景。如果没有趁手的工具，我们可以自己写一个，自己想办法处理传给main函数的参数。

传入一个程序的参数包含了“标记”（flag）和“值”（value）。标记都是一个字母，前面加上“-”号（例如“-p”这样）。每个标记可以有一个值与之对应，也可以没有对应的值。

我们要开发一个解析器（parser）来处理这些参数。这个解析器需要一个参数结构（schema）来描述“这个程序应该接受哪些参数”的信息，包括：

* 应该有几个标记；
* 每个标记应该是什么类型；
* 每个标记的缺省值是什么。

参数结构指定好以后，就可以把实际接收到的参数列表传给参数解析器。解析器会首先验证参数列表是否与参数结构相匹配。然后，程序就可以向参数解析器查询每个参数的值（根据参数的标记名）。返回值的类型应该与参数结构中规定的类型相一致。

例如程序接收到的参数是这样：

-l -p 8080 -d /usr/logs

那么对应的参数结构应该规定3个标记：l、p、d

* “l”（logging，是否记录日志）标记没有与之对应的值，这是一个布尔型的标记，如果传入了“-l”就为True，否则为False。
* “p”（port，端口号）标记的值是整数型。
* "d"（directory，目录）标记的值是字符串型。

如果参数结构中规定了的标记在实际的参数列表中没有出现，那么就应该返回合理的缺省值，例如布尔型的缺省值可以是False，数值型的缺省值可以是0，字符串型的缺省值可以是空字符串。

如果实际给出的参数与参数结构不匹配，需要给出良好的错误信息，解释清楚出错的原因。

### **扩展需求**

扩展代码，支持列表类型的参数。例如下列参数中：

-g this,is,a,list -d 1,2,-3,5

“g”标记对应的是字符串类型的列表（[“this”, “is”, “a”, “list”]），“d”标记对应的是整数类型的列表（[1, 2, -3, 5]）。

代码应该具有良好的可扩展性，这样在添加新的值类型时才会简单明了。

## **教学目标**

* 准确框定需求范围
* 把需求拆解成明细的任务列表
* 练熟测试驱动开发的节奏

## **学习周期**

7小时

## **配套资料**

* 《测试驱动开发》，第一部分
* 《重构》，第三章
* 在线文章：需求都没整清楚，你写啥代码（[https://mp.weixin.qq.com/s/DNrZ5OJlmwjfIT97Cu9TSg](https://link.zhihu.com/?target=https://mp.weixin.qq.com/s/DNrZ5OJlmwjfIT97Cu9TSg" \t "https://zhuanlan.zhihu.com/p/_blank)）
* 在线文章：不读《实现模式》就不要写代码（[https://mp.weixin.qq.com/s/fLJIj0WdDYJcPfjYWJLGPw](https://link.zhihu.com/?target=https://mp.weixin.qq.com/s/fLJIj0WdDYJcPfjYWJLGPw" \t "https://zhuanlan.zhihu.com/p/_blank)）
* 在线文章：像机器一样思考（[http://insights.thoughtworkers.org/think-as-a-machine/](https://link.zhihu.com/?target=http://insights.thoughtworkers.org/think-as-a-machine/" \t "https://zhuanlan.zhihu.com/p/_blank)）

## **1.项目剖析**

## **项目解读**

Args的难度比FizzBuzz高得多。如果没有做好基本功的准备，贸然开始这道题目，恐怕很多人会感到很受挫。据我们的观察，大多数程序员首次练习完成这道题需要4小时以上，甚至用了一整天还完不成的情况也绝非罕见。

Args这道题目有意地在很多地方对需求描述不够清晰，例如参数结构的定义方式、参数的输入格式等。这就需要我们学会用测试来准确地表达自己对需求的理解，用测试作为与客户沟通的桥梁，用测试框定需求的范围。

一个有一定规模的需求，不仅要知道建造完成后的样子，还要知道如何一步步建造出来的过程。这个过程设计得好，整个需求的开发会很平稳，工作量评估会很准确，出错的可能性会很低。合理拆分任务，是良好建造过程的基础。

在开发遇到困难的时候，要学会把测试的步伐放得小一点、再小一点，从最简单的问题开始，每次专注写一个测试、让一个测试通过。

虽说作为练习题显得有些难度，其实这道题的复杂度跟日常工作中拆分良好的用户故事差不多，也就是大多数程序员需要一到两天开发出来的规模。练功房的小伙伴已经练到能在半小时内完成这道题。这就是基本功带来的效率差距。

## **技能要求**

* 用测试用例描述和沟通需求
* 把需求拆解成可以逐步开发的任务列表
* 基本的面向对象设计
* 识别代码坏味道
* 重构的基本节奏和原则

## **项目拆解**

本项目分为7个任务：

* Args初体验
* 框定需求范围
* Part I：Schema
* Part II：读入参数
* 封装所有的容器
* Part III：取出参数值
* 大类和长函数

## **任务1：Args初体验**

第一个任务很简单：做一遍Args这道题。

**做题的要求：**

* 看懂题目，开始编码之前先花10分钟拆解任务，把任务清单写下来
* 计时，开始编码
* 要求用TDD的方式实现：先写测试，后写代码

如果超过90分钟还没完成，就先暂停。我们还需要留出一点时间来反思。

**反思：**

* 做完整道题用了多长时间？
* 代码质量怎么样？
* TDD的方式顺畅吗？
* 实际做的步骤，和一开始拆分的任务是否相同？

## **任务2：框定需求范围**

如果在任务一遇到了困难，有很大可能是对需求的范围框得不够清晰，不知不觉中把需求扩大化了。（当然，也可能漏掉了一些重要的需求。）

比如说，有很多同学会花大量精力思考如何做字符串的解析。但是我们是否先问过客户，输入参数的格式应该是什么样？我们是否首先思考过，这个需求可以分解成几个大块？各个大块的优先级顺序是什么？

**在动手写代码之前，我们要首先弄清需求的范围和优先级，并用测试的形式把它记录下来。**

**做题的要求：**

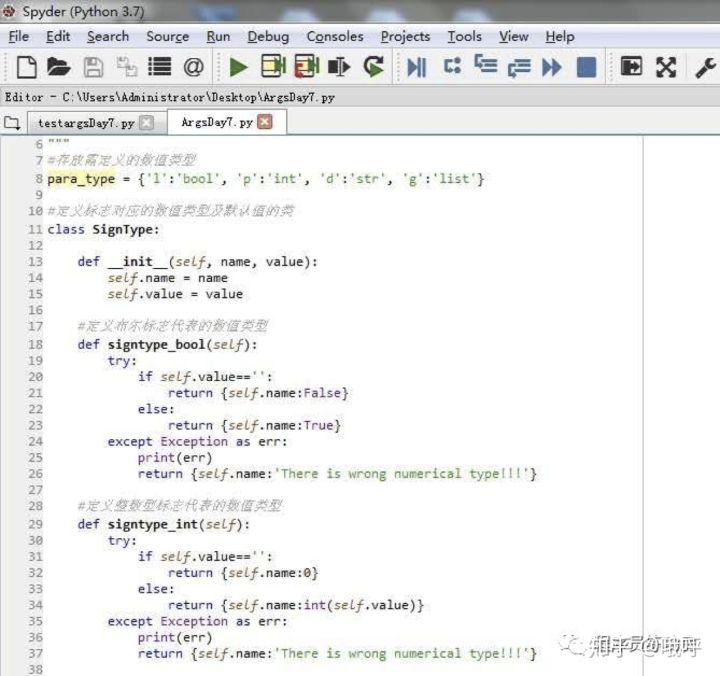
* 看懂题目，仔细想想，最终的用户会如何使用你开发的程序
* 只写测试，用测试来描述最终用户如何使用你开发的程序
* 写完测试之后，阅读这篇文章，看看你写的测试是否合理
* 再写一遍测试

**反思：**

* 你对用户需求的理解合理吗？
* 从你编写的测试来看，用户使用你开发的程序方便吗？
* 你理解的需求可以分为几个大块？
* 各个大块需求的优先级顺序是什么？ 最好能再练一遍，验证反思的结果。

## **任务3：Part I：Schema**

在处理Args的需求时，一个常见的错误是没有考虑必要的灵活性。很多同学把“-l参数是布尔型”、“-p参数是整数型”这些信息硬编码写在代码中，就像这样：



于是这个程序就与例子中这个使用场景绑死了。比如说，假如另一个使用者要使用我们开发的程序，然而在他的场景中，要处理的是“-w”这个参数，我们怎么办呢？难道每增加一个使用者我们就要去改一遍代码吗？很多不可维护的焦油坑软件就是这么来的。

题目里已经给出了“参数结构（schema）”这个概念。今天我们的任务就是，开发Schema输入和解析的逻辑。

请注意：今天我们只处理“将Schema读入对象结构”的逻辑。至于怎么使用Schema，暂时不考虑。

**做题的要求：**

* 先想一想，与Schema相关的对象结构是什么样的，用测试描述出来
* 然后考虑，用户会如何输入Schema信息，用测试描述出来
* 编写实现代码，让测试通过

**反思：**

* 你的测试是否一次性考虑了太多的事？
* 如果再来一遍，如何可以缩小测试-开发的步伐？

最好能再练一遍，验证反思的结果。

## **任务4：Part II：读入参数**

除了Schema之外，Args的另一个大块功能，就是从用户的输入中读入参数。今天我们的任务就是实现这部分逻辑。

请注意：今天我们只处理“将参数读入对象结构”的逻辑。至于怎么使用参数，今天暂不考虑。

**做题的要求：**

* 先想一想，与参数相关的对象结构是什么样的，用测试描述出来
* 然后考虑，用户会如何输入参数信息，用测试描述出来
* 编写实现代码，让测试通过

**反思：**

* 你的测试是否一次性考虑了太多的事？
* 如果再来一遍，如何可以缩小测试-开发的步伐？

最好能再练一遍，验证反思的结果。

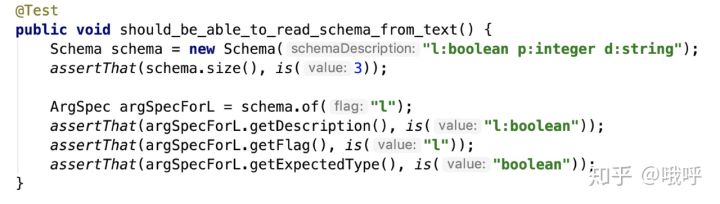
## **任务5：封装所有的容器**

[https://mp.weixin.qq.com/s/YojJITzYIycG6PyGCrRxPw](https://link.zhihu.com/?target=https://mp.weixin.qq.com/s/YojJITzYIycG6PyGCrRxPw" \t "https://zhuanlan.zhihu.com/p/_blank) 《用一次Map就失去一个对象》

在前面两天的练习里，你很可能用了不少的容器，尤其是Map。例如此前有个同学有这样一段测试：



同样的逻辑，我写的测试就没有暴露任何容器：



我在一篇[公众号文章](https://link.zhihu.com/?target=https://mp.weixin.qq.com/s/YojJITzYIycG6PyGCrRxPw" \t "https://zhuanlan.zhihu.com/p/_blank)里这样说：

“Map的出现几乎一定意味着有个对象的概念被错过了” “Map即对象”

从内容来看，Map和对象真的是一样的：Map的key就是对象的字段名；Map的value就是对象的字段值。JavaScript早年间没有那么明确的面向对象特征的时候，大家真的就是把Map和对象混用的。

但是Map和对象最大的区别在于：你是否给它一个合适的名字。老话说名不正则言不顺。没有一个合适的名字说“这是一个Flag对象”，你就倾向于不会多想想，到底哪些行为应该属于这个对象。

其实还不仅限于Map。再比如List（甚至数组），很多时候也是被过度使用的。当一个对象给另一个对象提供一个List，很多时候就意味着暴露了自己的内部信息。所以我的观点是，List和Map（以及其他集合结构）不应该在对象之间传递，只应该在对象内部使用。

这些本来应该封装成对象的集合结构没有得到合适封装， 其结果就是我昨晚上说的： “我注意到这是大家的代码另一个常见的问题” “对象太少” “带来的结果就是很多逻辑在一个函数里一层套一层”

所以下次一旦发现自己开始用Map，不妨多想一想：你需要的真的是一个Map吗？或者，其实你需要的是一个正等着被定义的对象？

感受一下。把前面做过的题目再练一遍。

**做题的要求：**

* 把前面做过的“读入Schema”和“读入参数”的逻辑再实现一遍，遵循TDD的节奏，先写测试，再写代码，步伐要小。
* 注意：封装所有集合结构（例如Map和List）。集合结构只在你自己定义的对象内部使用，不能传递到对象之外。

**反思：**

* 封装了集合结构之后，代码产生了什么变化？
* 得到了新的对象吗？
* 是否有更好的方式对这些新的对象进行测试？

最好能再练一遍，验证反思的结果。

## **任务6：Part III：取出参数值**

有了前面的对象做基础，完成Args的最后一步“取出参数值”应该很简单了。这就是我们今天的任务。

**做题的要求：**

* 完整实现Args的需求。如果时间来不及，可以先不考虑边界异常情况。
* 开始编码之前先花10分钟拆解任务，把任务清单写下来
* 要求用TDD的方式实现：先写测试，后写代码
* 计时，开始编码

**反思：**

* 做完整道题用了多长时间？
* TDD的方式顺畅吗？每行代码都是由测试驱动出来的吗？
* 实际做的步骤，和一开始拆分的任务是否相同？

最好能再练一遍，验证反思的结果。

## **任务7：大类和长函数**

代码质量不是主观的审美判断。它是有相当客观的判断依据的。这个判断依据就是Martin Fowler在《重构》里说的“坏味道”（bad smell）。

这堂课，我们翻开《重构》（第二版），翻到第三章“代码的坏味道”。这里面列举了24种坏味道，今天我们专注看其中的两个坏味道：Long Method（过长函数）和Large Class（过大的类）。

多长的函数就算“过长”？多大的类就算“过大”？我有一个简单的经验法则。对于Args这样一个典型的需求场景：

* 如果任何一个函数的实现体（不含函数签名和外围的大括号）超过5行代码，这个函数就过长了
* 如果任何一个类的实现体（不含import、类签名和外围的大括号）超过50行代码，这个类就过大了

你的代码中是否已经出现了这两个坏味道？《重构》第三章给出了对应的重构手法，试着照书里写的重构手法，重构一下看看吧。

**练习的要求：**

* 从前一个任务的代码开始，查看有没有“过长函数”和“过大的类”坏味道
* 如果发现了坏味道，看看对应的重构手法是什么
* 重构，直到坏味道消失

**反思：**

* 重构的感觉怎么样？是否有信心？
* 重构之后的代码质量怎么样？