



下载APP



拓展3 | Python 程序员的 Vim 工作环境：完整的 Python 开发环境

2020-08-28 吴咏炜

Vim 实用技巧必知必会

[进入课程 >](#)**讲述：吴咏炜**

时长 12:45 大小 11.68M



你好，我是吴咏炜。

今天这一讲，我会介绍 Python 程序员定制 Vim 工作环境的完整方法。

Python 的流行程度越来越高，Python 程序员们对此一定是很高兴的。在 Stack Overflow 的 2020 年开发者调查里，Python 在最受爱戴（most loved）的语言里排名第三，而在最想要（most wanted）的语言里则已经连续四年排名第一！因此，它在 Vim 的生态系统



里受到了良好的支持，也不会是件令人吃惊的事。有开发者已经把 Python 开发所需要的若干插件揉到了一起，组成了一套开箱即用的工具，python-mode。

今天我们就以它为基础，讨论一下 Vim 对开发 Python 提供的支持。

功能简介

🔗 **Python-mode** 实际上是以 Vim 插件形式出现的一套工具，它包含了多个用于 Python 开发的工具。根据官网的介绍，它的主要功能点是：

支持 Python 3.6+

语法加亮

虚拟环境支持

运行 Python 代码 (<leader>r)

添加 / 删除断点 (<leader>b)

改善了的 Python 缩进

Python 的移动命令和操作符 ([], 3[[,]]M, vaC, viM, daC, ciM, ...)

改善了的 Python 折叠

同时运行多个代码检查器 (:PymodeLint)

自动修正 PEP 8 错误 (:PymodeLintAuto)

自动在 Python 文档里搜索 (K)

代码重构

智能感知的代码完成

跳转到定义 (<C-c>g)

.....

不过，还是要提醒一句，它的功能虽然挺多，但作为非商业软件，全靠志愿者来贡献代码，并不是所有功能的完成度都很高。有些功能做得尚不完善，有些功能则略显鸡肋，所以，我也不会全部都讲解。我们就择善而从之，在利用它不需要用户介入就能提供的功能外（如语法加亮和缩进），重点讲解它做得好的地方，以及可能有陷阱需要规避的地方。

安装

Python-mode 没有编译组件，全部由脚本代码组成，因而使用你的包管理器安装 python-mode/python-mode 即可，非常简单。

以 minpac 为例，你只需要在 vimrc 配置文件中 “Other plugins” 那行下面加入：

```
1 call minpac#add('python-mode/python-mode')
```

[复制代码](#)

然后执行 :PackUpdate 命令即可。

配置

在没有任何配置的情况下，python-mode 也是完全可用的。但如果你再做一些基本设置的话，就能够解决一些常见问题和规避一些常见陷阱。

我个人的设置是下面这个样子的：

```
1 function! IsGitRepo()
2     " This function requires GitPython
3     if has('pythonx')
4     pythonx << EOF
5     try:
6         import git
7     except ImportError:
8         pass
9     import vim
10
11 def is_git_repo():
12     try:
13         _ = git.Repo('.', search_parent_directories=True).git_dir
14         return 1
15     except:
16         return 0
17 EOF
18     return pyxeval('is_git_repo()')
19 else
20     return 0
21 endif
22 endfunction
```

[复制代码](#)

```
23 let g:pymode_rope = IsGitRepo()
24 let g:pymode_rope_completion = 1
25 let g:pymode_rope_complete_on_dot = 0
26 let g:pymode_syntax_print_as_function = 1
27 let g:pymode_syntax_string_format = 0
28 let g:pymode_syntax_string_templates = 0
29
```

稍微解释一下：

IsGitRepo 是利用 Python 代码检测当前是不是在 Git 库目录下的一个函数，它要求你在 Python 环境里安装了 GitPython (`pip3 install GitPython`)。

我们仅仅在当前目录是一个 Git 库下面才启用 rope 支持 (`pymode_rope`)。Rope 是 `python-mode` 里提供语义识别和自动完成的主要工具，它会扫描所有子目录并创建 rope 工程目录。如果你一不小心在你的主目录（或子目录非常多的地方）执行 `python-mode` 的命令，可能会导致 Vim 卡顿（`python-mode` 并不是一个异步的插件）。所以我们在这儿特别限制一下，防止误操作。

我们启用 rope 的完成功能 (`pymode_rope_completion`)。

我们禁用在输入 `.` 号时自动完成的功能 (`pymode_rope_complete_on_dot`)。这是因为 rope 提供的自动完成会侵入式地影响正常输入流，即如果我想不理睬自动完成是不行的。这一点就不如 YCM 了。因此，我们的自动完成仍然使用 YCM。不过，需要的话，我们仍可以通过 `<C-X><C-O>` 来使用 rope 的自动完成。

Python-mode 对 Python 语法的加亮改善还不错，但它的默认行为是把 `print` 作为保留字显示，而不是普通函数。在写 Python 3 时，还是需要修改一下它的行为

(`pymode_syntax_print_as_function`)。

Python-mode 会试图对字符串中出现的格式化字符串和模板替换字符串做特殊的加亮

(`pymode_syntax_string_format` 和

`pymode_syntax_string_templates`)。这儿主要的问题是，它会误匹配字符串中出现的 `{}` 和 `$` 序列。我个人不习惯错误的加亮，不过你可以根据自己的喜好，来决定是不是要启用这个功能。

使用

语法加亮

Python-mode 提供了自己的语法加亮文件。除了上面提到的可以选择对 `print` 如何加亮，以及在字符串内部进行特殊加亮的选项外，它还提供了很多改进，并且可以由用户通过选项来微调（`:help pymode-syntax`），如对赋值号（`=`）的特殊高亮和对 `self` 的特殊高亮，等等。这些改进我觉得还挺有用。

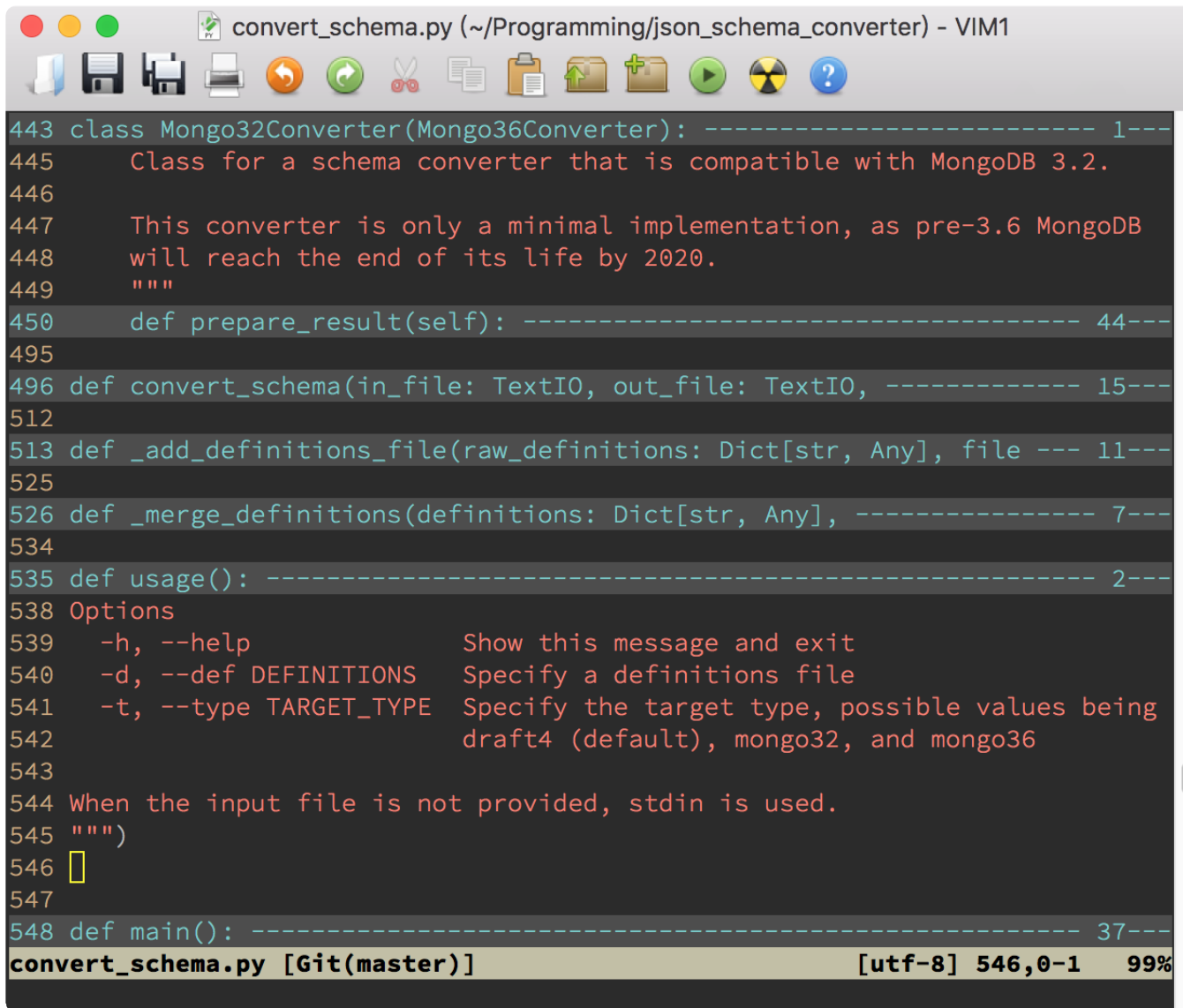
代码折叠

我个人一直不怎么喜欢代码折叠（主要是觉得额外展开这个步骤非常有干扰，而更愿意一目十行式地快速浏览），所以 Vim 的这个功能我基本不用。如果你喜欢折叠的话，你应该会很高兴 python-mode 能帮你自动折叠 Python 代码。你只需要在 `vimrc` 配置文件中加入下面这行即可：

```
1 let g:pymode_folding = 1
```

[复制代码](#)

效果见下图：



```

443 class Mongo32Converter(Mongo36Converter): ----- 1---
444     Class for a schema converter that is compatible with MongoDB 3.2.
445
446     This converter is only a minimal implementation, as pre-3.6 MongoDB
447     will reach the end of its life by 2020.
448     """
449
450     def prepare_result(self): ----- 44---
451
452
453 def convert_schema(in_file: TextIO, out_file: TextIO, ----- 15---
454
455
456 def _add_definitions_file(raw_definitions: Dict[str, Any], file --- 11---
457
458
459 def _merge_definitions(definitions: Dict[str, Any], ----- 7---
460
461
462 def usage(): ----- 2---
463
464 Options
465 -h, --help          Show this message and exit
466 -d, --def DEFINITIONS Specify a definitions file
467 -t, --type TARGET_TYPE Specify the target type, possible values being
468                        draft4 (default), mongo32, and mongo36
469
470 When the input file is not provided, stdin is used.
471 """
472
473
474 def main(): ----- 37---
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

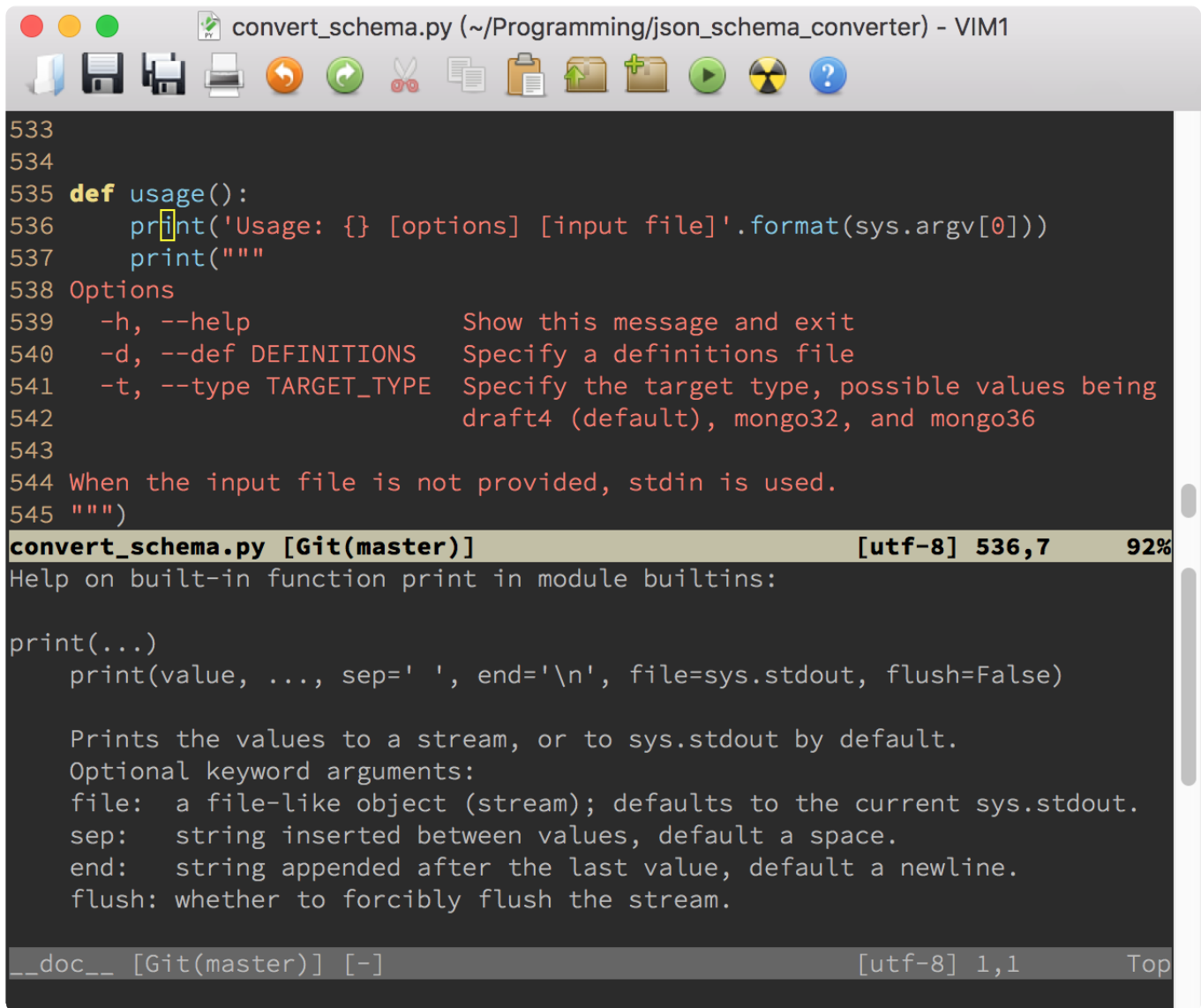
```

代码折叠效果

这个功能会导致打开 Python 文件变慢。你可以试试，斟酌一下自己是否希望使用这个功能。

快速文档查阅

Python-mode 默认映射了 `K` 对光标下的单词进行文档查阅。跟其他查阅文档的方式比起来，这还是非常快捷方便的。



The screenshot shows a Vim editor window titled 'convert_schema.py (~/.Programming/json_schema_converter) - VIM1'. The editor displays a Python script with a `usage()` function. The cursor is positioned at line 536, column 7, where the `print` function is called. Below the script, the help text for the `print` function is shown, detailing its arguments and usage. The status bar at the bottom indicates the current file is `convert_schema.py` on the `Git(master)` branch, with the cursor at line 536, column 7, and the file is 92% complete. The status bar also shows the current file is `__doc__` on the `Git(master)` branch, with the cursor at line 1, column 1, and the file is 100% complete.

```

533
534
535 def usage():
536     print('Usage: {} [options] [input file]'.format(sys.argv[0]))
537     print("""
538 Options
539 -h, --help            Show this message and exit
540 -d, --def DEFINITIONS Specify a definitions file
541 -t, --type TARGET_TYPE Specify the target type, possible values being
542                        draft4 (default), mongo32, and mongo36
543
544 When the input file is not provided, stdin is used.
545 """)
convert_schema.py [Git(master)] [utf-8] 536,7 92%
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

__doc__ [Git(master)] [-] [utf-8] 1,1 Top

```

使用 K 查看 Python 的文档

缩进支持

在 Vim 的运行支持文件中，本来就包含了对 Python 缩进的支持，但默认的支持并没有把像 PEP 8 这样的 Python 编程规范考虑进去，缩进风格并不十分正确。安装了 `python-mode` 后，缩进就能更好地自动遵循 PEP 8 规范了。

代码检查

不管 Vim 的缩进对不对，如果你在其他编辑器里编辑了 Python 代码，Vim 是不会修正其中的缩进或其他问题的——除非你启用代码检查器。

Python-mode 里带了好几个代码检查器，默认启用的是下面三个：

`pyflakes`，一个很轻量的代码检查器，检查常见的 Python 编码问题，如未使用的变量和导入

pep8, 一个专门检查代码是否符合 PEP 8 的检查器

mccabe, 一个专门检查圈复杂度的代码检查器

默认启用哪些检查器, 是通过下面的全局变量来控制的:

复制代码

```
1 let g:pymode_lint_checkers = ['pyflakes', 'pep8', 'mccabe']
```

你可以自己在 vimrc 配置文件里定义这个变量, 调节希望使用的代码检查器。我觉得默认的代码检查器还比较合适, 因为执行真的很快, 基本上可以在执行检查的瞬间帮你检查完代码并标记出问题。你可以手工执行 :PymodeLint 来检查代码, python-mode 也会自动在你保存文件时进行检查。

```
1 #!/usr/bin/env python
2 #coding: utf-8
3
4 from __future__ import print_function
5 from enum import Enum
6 import re
7 import sys
8
9 class BlockMode(Enum):
10     none = 0
11     single_quot = 1
12     double_quot = 2
13     comment = 3
14     one_line_comment = 4
15
16 block_start_c = {
17     """ : BlockMode.single_quot,
18     """ : BlockMode.double_quot,
19     '/*' : BlockMode.comment,
20     '//': BlockMode.one_line_comment,
21 }
22
23 def skip_a_char(string):
24     return string[1:]
25
26 def proc_block_c(line, block_mode):
27     """
28     Processes the next block in the code line.
29
30 count_sloc1.py  [+]  [utf-8] 1,1  Top
```

我几年前写的不符合 PEP 8 的代码存盘试验

可以看到, 检查的结果会在屏幕的左侧标记出来, 表示不同的问题类型; 并且光标移到这样的行上, Vim 底部还会显示问题的描述信息。同时, python-mode 检查出问题时会自动

动打开一个位置列表，我们在第 13 讲提过，这是跟窗口关联的类似于快速修复窗口的信息窗口。由于我们可能在多个窗口 / 标签页编辑多个文件，位置列表确实比较合适。当 python-mode 认为你修复了所有问题时，这个位置列表也会自动关闭。

顺便提醒你注意一下屏幕右侧的红线（在某些配色方案里可能是其他颜色）。这条线在第 80 列上，也是提醒你写代码不能到那个位置，因为 PEP 8 规定 Python 代码行最长是 79 个字符。如果到达红线位置的话，那 pep8 检查的时候，一定跑不了，会报错的。

上面图中的错误都是 PEP 8 问题，绝大部分可以简单地执行 `:PymodeLintAuto` 命令来自动解决，用不着我们自己去动手修改代码。

Python-mode 还有两个没有默认启用的检查器：

pylint，一个功能很强的代码检查器，它可以嗅出你的代码中的坏味道，除了性能，可以说是全面强于 pyflakes（使用它你得擦亮眼睛，做好被它虐的准备）

pep257，一个检查文档串（docstring）是否符合 PEP 257 的工具（这个工具我个人感觉不成熟，给出的建议有点混乱）

由于 pylint 执行比较慢，我觉得还是先写完代码再专门来扫描并解决其报告的问题比较合适。上面的这个示例代码，跑 pylint 需要超过一秒才能执行完成，在存盘时自动执行检查基本属于不可忍受。这当然也是因为 python-mode 没有异步执行外部命令造成的。我们最后还会再看一下执行慢和异步的问题。

Rope 支持

Rope 是一个 Python 库，提供对 Python 代码的分析、重构和自动完成功能。由于我们使用 YCM 来进行自动完成，也能完成像跳转到定义这样的任务，rope 就略显鸡肋了。不过，它有重命名重构功能，而 YCM 并不支持对 Python 的重命名重构，所以两者功能还不算完全重叠。

你如果决定要用一下 rope 的话，需要了解以下几点：

rope 会使用一个叫做 `.ropeproject`（默认名字）的目录，在里面缓存需要的信息；这个目录在当前目录下，或当前目录的一个父目录下；如果找不到，默认会在当前目录下创建这个目录

使用命令 `:PymodeRopeNewProject 路径` 可以在指定路径下创建这个 `.ropeproject` 目录

使用命令 `:PymodeRopeRegenerate` 可以重新产生项目数据的缓存

默认情况下 (`g:pymode_rope_regenerate_on_write` 等于 1)，在文件存盘时 `python-mode` 即会自动执行 `:PymodeRopeRegenerate` 命令

在启用 `rope` 之后，你就可以使用下面的命令了：

使用 `<C-X><C-O>` 来启用自动完成（我们把 . 还是交给 YCM 了）

使用 `<C-C>g` 来跳转到定义（跟 YCM 的 `\gt` 比，大部分情况下没区别；`rope` 跳转更好和 YCM 跳转更好的情况都有，但都不多见）

使用 `<C-C>d` 来查看光标下符号的文档；和 `K` 键不同，这个命令可以查看当前项目代码里的文档字串

重构 (refactor) 功能以 `<C-C>r` 开始，如 `<C-C>rr` 是重命名 (rename) 光标下的符号，这些功能还是比较强大的（可以使用 `:help pymode-rope-refactoring` 来查看完整的帮助信息）

下面的动图展示了 `rope` 的若干功能：

```
488         del flattened_result[stringized_path]['$exists']
489     if 'properties' in result:
490         for k, v in result['properties'].items():
491             if k in required_properties:
492                 Mongo32Converter._flatten_recursively(
493                     v, flattened_result, obj_path + [k])
494
495
496 def convert_schema(in_file: TextIO, out_file: TextIO,
497                   definitions: Dict[str, Any], target_type: str):
498     schema = json.load(in_file)
499     class_table = {
500         "draft4": Draft4Converter,
501         "mongo32": Mongo32Converter,
502         "mongo36": Mongo36Converter,
503     }
504     try:
505         converter = class_table[target_type](schema, definitions)
506     except KeyError:
507         raise RuntimeError('Unrecognized target type ' + target_type)
508     result = converter.result
509     json.dump(result, out_file, ensure_ascii=False, indent=2)
510     print('')
511
512
513 def _add_definitions_file(raw_definitions: Dict[str, Any], filename: str)
<json_schema_converter/convert_schema.py [Git(master)] [utf-8] 500,19 86%
```

在 rope 里查看文档、跳转到定义和重命名

替换方案

如果你对 python-mode 的某些功能不满意，可以禁用其部分功能，用其他插件来代替。

首先，如果你如果觉得 rope 提供的额外功能对你用处不大的话，我们可以完全禁用 rope (let g:pymode_rope = 0)，专心使用 YCM。这样，硬盘上也就不会出现 .ropeproject 那样的目录了。

其次，如果你真的希望能在写代码的时候自动进行 pylint 检查，那你也可以禁用 python-mode 里的代码检查器功能 (let g:pymode_lint = 0)，转而使用 [ALE](#) 来进行异步检查。你需要安装它（包管理器需要的名字是 dense-analysis/ale），并在 vimrc 配置文件中加入：

 复制代码

```
1 let g:ale_linters = {
2     \ 'python': ['pylint'],
3     \ }
```

别忘了这种情况下，你需要自己用 pip 安装 pylint。这不像 python-mode 的情况，所有工具都已经打包在那一个套件里面了。

内容小结

在这一讲，我们通过介绍 python-mode，介绍了一个比较适用于 Python 程序员的 Vim 开发环境。这个工具集成了对 Python 的语法加亮、代码折叠、文档查阅、代码检查、自动完成等多方面的功能，对 Python 开发者非常适用。我们同时也讨论了 Vim 之外的一些代码检查工具，以及当你对 python-mode 不满意时，如何部分替换其功能。

课后练习

同样地，学完今天这一讲之后，你的主要任务就是把 python-mode 装起来、配置好、用一下。如果遇到什么问题，欢迎留言和我讨论。

我是吴咏炜，我们下一讲再见！

提建议

更多课程推荐

程序员的数学基础课

在实战中重新理解数学

黄申

LinkedIn 资深数据科学家



涨价倒计时 🕒

今日秒杀 **¥79**, 9月11日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 拓展2 | C 程序员的 Vim 工作环境：C 代码的搜索、提示和自动完成

下一篇 拓展4 | 插件样例分析：自己动手改进插件

精选留言 (4)

写留言



YouCompleteMe

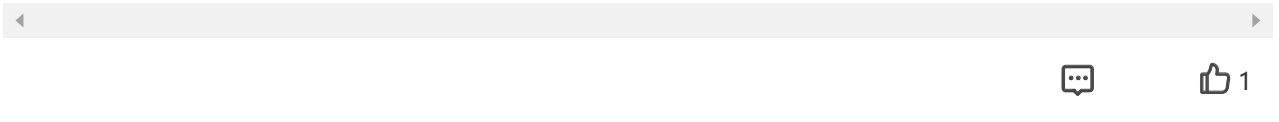
2020-09-04

测了下--startuptime, IsGitRepo在我的电脑上带来140ms的耗时, 换成 let g:pymode_rope = !empty(finddir('.git', ':')) ,耗时较少。同时看到colorscheme 设置语句, 会带来30ms的加载时间, 这个有办法优化吗~

作者回复: 谢谢, 你不说, 我都没意识到这个问题。

不过, 有趣的是, 你说的方法能大大加快开启空 Vim 的速度或者打开非 Python 文件的速度, 但对同时打开一个 .py 文件效果提升很不明显。后面加载 Python-mode 插件似乎把这部份开销抵消了。

色彩方案的耗时要看色彩方案的复杂度。我在我的机器上比了一下，我用的 desertEx 比 gruvbo x 耗时要短一半以上。好看的代价吧，哈哈。



小铁匠

2020-09-01

使用pyenv和portry来管理依赖，怎么为不同项目配置不同虚拟目录

作者回复: 我觉得不需要配置。你进入这个虚拟环境后，再用 Vim 打开项目里的文件，自然就使用这个虚拟环境的配置了。



我来也

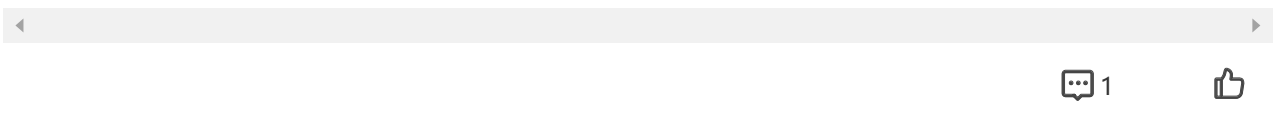
2020-08-29

学习了。

平常工作中，python用的不多，我就用coc.nvim应付一下算了。
有基本的语法高亮、补全、跳转就行了。

展开 ∨

作者回复: 哈哈，难怪这一讲你不是沙发了。



YouCompleteMe

2020-08-28

又到了纠结用ale 还是 pymode 和 YouCompleteMe提供的诊断功能了-_-

作者回复: 如果开用ALE和pylint，诊断可以让它干吧。或者懒得手工装工具，还让pymode管比较快的检查，不过pyflakes功能重复了，可以关掉不用。

