

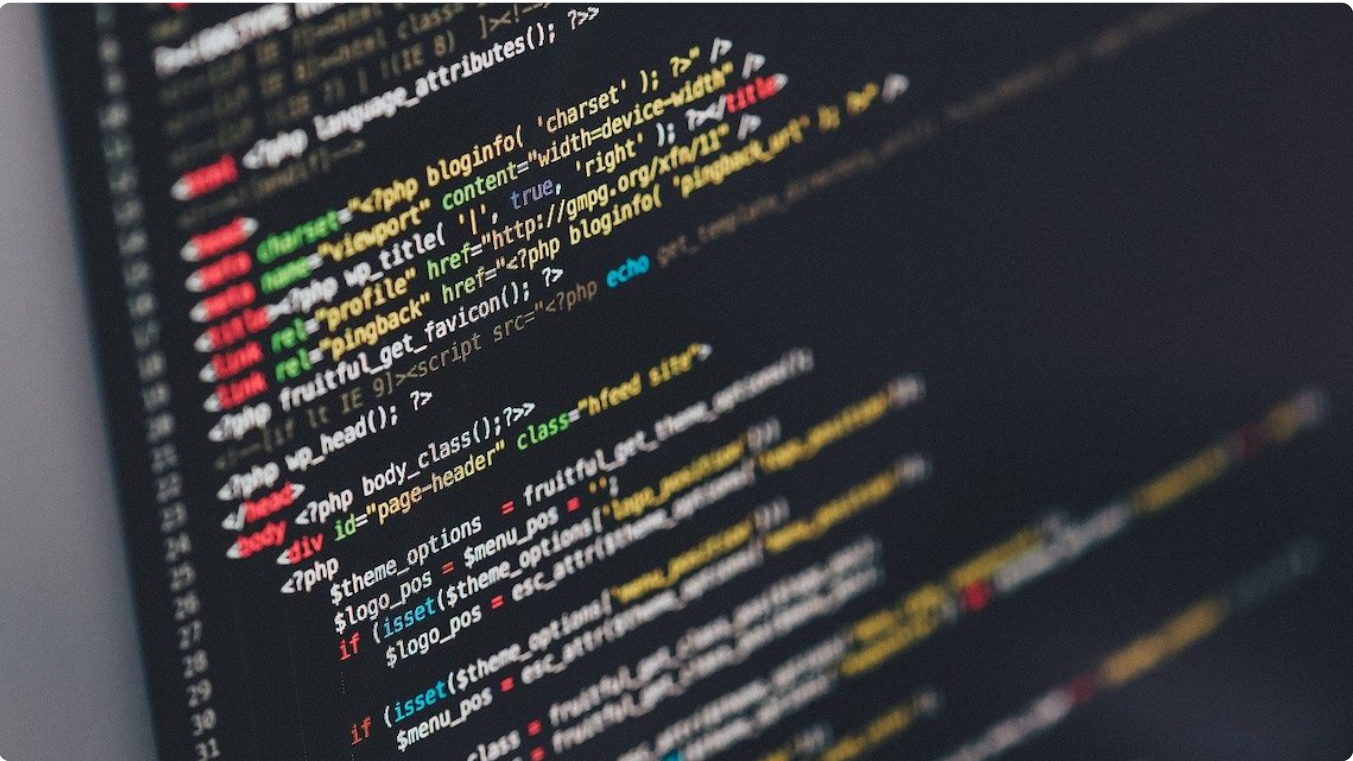


12 | 语法加亮和配色方案：颜即正义

2020-08-21 吴咏炜

Vim 实用技巧必知必会

[进入课程 >](#)



讲述：吴咏炜

时长 18:03 大小 16.53M



你好，我是吴咏炜。

语法加亮这个功能，我们都非常熟悉。和 vi 刚出现的时代不同，它现在已经成为编程的基本功能了。在我们使用的各种代码编辑器中，都有语法加亮的功能。我们甚至可以拿一句俗语反过来说：没见过猪跑，还能没吃过猪肉么？

但是，你有没有想过，语法加亮到底是怎么实现的呢？今天，我们就不仅要尝尝不同“风味”的猪肉，还要进一步看看猪到底是怎么跑的——这样，我们才能选择，然后调整出最符合自己口味的大菜。

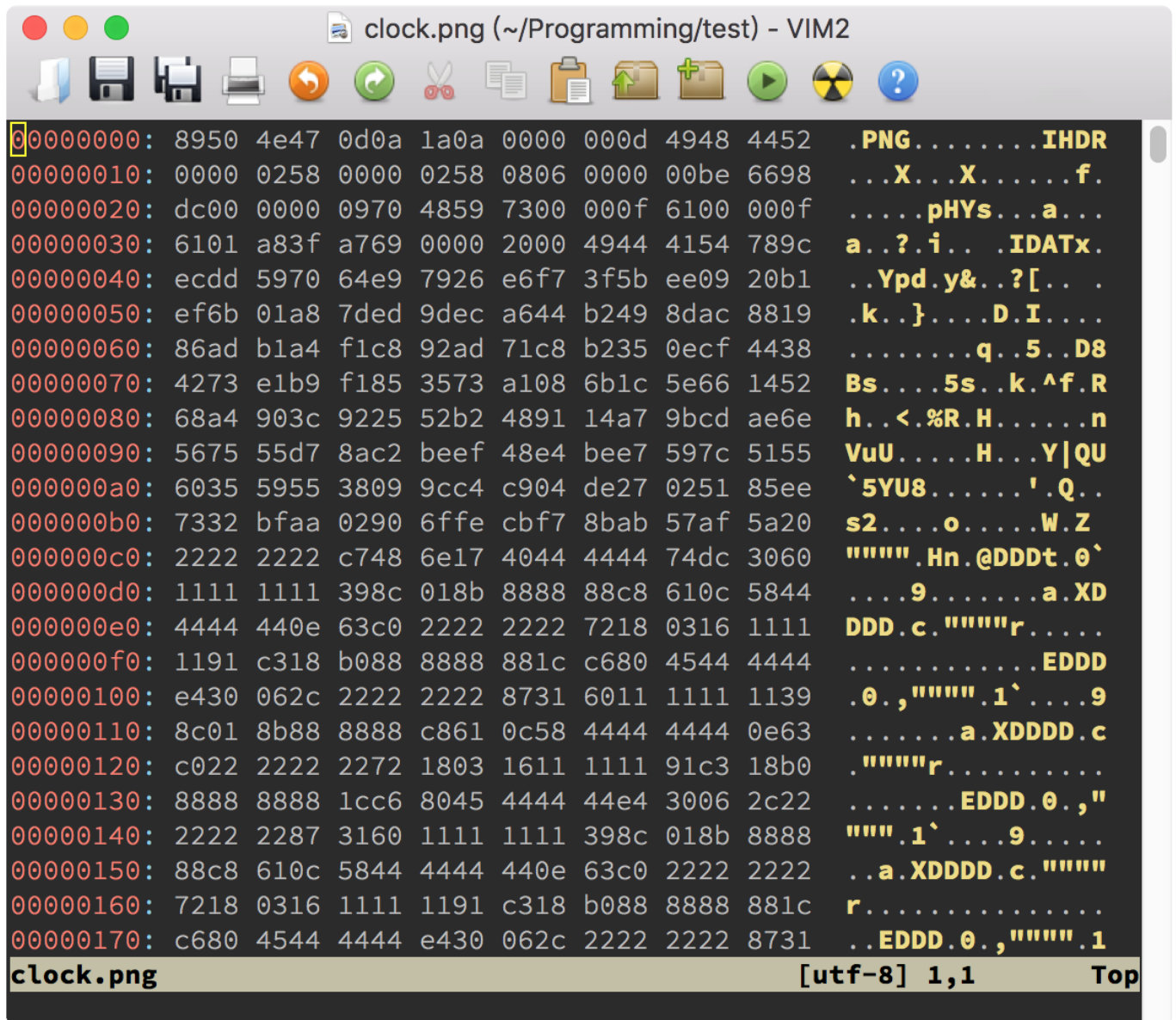


语法加亮

在 [第 8 讲](#) 里，我们已经提到，Vim 的语法加亮依靠的是在 `syntax` 目录下的运行支持文件。今天，我就通过例子给你解说一下，Vim 里如何实现语法加亮，然后语法加亮又如何映射到屏幕上的颜色和字体。

我们先来看一个比较简单的例子，`xxd`。

`xxd` 这个名字看起来，是不是有点陌生又有点熟悉？其实，我们在第 11 讲还刚讲过 `xxd`：它是一个把二进制文件转换成地址加十六进制数值再加可读 ASCII 文本的工具，它的输出格式在 Vim 里也被称作 `xxd`。不过，在用菜单项或 `:%!xxd` 命令转换之后，Vim 并不会自动使用 `xxd` 格式。要应用 `xxd` 格式的语法加亮，我们需要使用自动命令（可以参考 [`:help using-xxd`](#)），或者手工使用命令 `:setf xxd`。下图是对上次的二进制文件使用了 `xxd` 语法加亮的效果：



```
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452 .PNG.....IHDR
00000010: 0000 0258 0000 0258 0806 0000 00be 6698 ...X...X.....f.
00000020: dc00 0000 0970 4859 7300 000f 6100 000f .....pHYs...a...
00000030: 6101 a83f a769 0000 2000 4944 4154 789c a..?.i...IDATx.
00000040: ecdd 5970 64e9 7926 e6f7 3f5b ee09 20b1 ..Ypd.y&..? [...
00000050: ef6b 01a8 7ded 9dec a644 b249 8dac 8819 .k..}....D.I....
00000060: 86ad b1a4 f1c8 92ad 71c8 b235 0ecf 4438 .....q..5..D8
00000070: 4273 e1b9 f185 3573 a108 6b1c 5e66 1452 Bs....5s..k.^f.R
00000080: 68a4 903c 9225 52b2 4891 14a7 9bcd ae6e h..<.%R.H.....n
00000090: 5675 55d7 8ac2 beef 48e4 bee7 597c 5155 VuU.....H...Y|QU
000000a0: 6035 5955 3809 9cc4 c904 de27 0251 85ee `5YU8.....'.Q..
000000b0: 7332 bfaa 0290 6ffe cbf7 8bab 57af 5a20 s2....o.....W.Z
000000c0: 2222 2222 c748 6e17 4044 4444 74dc 3060 """".Hn.@DDDt.0`
000000d0: 1111 1111 398c 018b 8888 88c8 610c 5844 ....9.....a.XD
000000e0: 4444 440e 63c0 2222 2222 7218 0316 1111 DDD.c.""""r.....
000000f0: 1191 c318 b088 8888 881c c680 4544 4444 .....EDDD
00000100: e430 062c 2222 2222 8731 6011 1111 1139 .0.,""""1`....9
00000110: 8c01 8b88 8888 c861 0c58 4444 4444 0e63 .....a.XDDDD.c
00000120: c022 2222 2272 1803 1611 1111 91c3 18b0 .""""r.....
00000130: 8888 8888 1cc6 8045 4444 44e4 3006 2c22 .....EDDD.0.,
00000140: 2222 2287 3160 1111 1111 398c 018b 8888 """"1`....9.....
00000150: 88c8 610c 5844 4444 440e 63c0 2222 2222 ..a.XDDDD.c.""""
00000160: 7218 0316 1111 1191 c318 b088 8888 881c r.....
00000170: c680 4544 4444 e430 062c 2222 2222 8731 ..EDDD.0.,""""1

clock.png [utf-8] 1,1 Top
```

使用了 `xxd` 语法加亮的效果

这个格式的语法加亮足够简单，我们就拿它来分析一下。不过，我有个小建议，你在看具体的语法加亮代码前，先花几秒钟的时间看一下图，自己分析一下里面有几种不同的语法加亮效果。

下面我们就来逐步看一下 `syntax/xxd.vim` 的内容。首先是开头和结尾部分：

[复制代码](#)

```
1 " quit when a syntax file was already loaded
2 if exists("b:current_syntax")
3     finish
4 endif
5
6 ...
7
8 let b:current_syntax = "xxd"
9
10 " vim: ts=4
```

最后一行的模式行，设定了这个文件使用的 `tab` 宽度。剩余部分基本上算是语法文件的固定格式了，有一个检查缓冲区变量（使用前缀 `b:`）、防止语法文件重复载入的条件判断，并在结尾设定这个缓冲区变量为语法的名称。

剩余部分可以分为两段。第一段是语法匹配：

[复制代码](#)

```
1 syn match xxdAddress      "[0-9a-f]\+:"      contains=xxdSep
2 syn match xxdSep          contained ":"
3 syn match xxdAscii        ".\{,16\}\r\=$"hs=s+2 contains=xxdDot
4 syn match xxdDot          contained "[.\r]"
```

这儿定义了 4 种不同的“语法项目”，其中 1、2 和 3、4 还互相有包含（“contains”）和被包含（“contained”）的关系。

1. **xxdAddress**。它是地址匹配，所以匹配条件是从行首开始的一个或更多的十六进制字符后面跟一个冒号。
2. **xxdSep**。它是分隔符，仅匹配 `xxdAddress` 中的冒号部分，也算是地址的一部分。

3. **xxdAscii**。它是右边的 ASCII 字符部分，条件是两个空格后面跟最多 16 个字符，然后是可选的 CR 字符（\= 和 \? 效果相同），然后必须是一行结束。
4. **xxdDot**。它是对 “.” 和 CR 字符的特殊匹配，可以留意一下上面图里 “.” 和其他字符的加亮效果的不同之处。同样，这个句点也属于 ASCII 字符部分。

上面的正则表达式都比较简单，唯一之前没出现过的是第 3 个正则表达式后面的 `hs=s+2`：它的含义是语法加亮的起始位置是模式匹配部分的开始位置再加 2（可查看 [@:help :syn-pattern-offset](#)），这是在语法加亮文件里的常用特殊语法。

上面的代码可以从 xxd 格式的内容中找出 4 种不同的语法格式。如何展示这些语法，就要看下面的第二段代码了：

[复制代码](#)

```
1 " Define the default highlighting.
2 if !exists("skip_xxd_syntax_inits")
3
4   hi def link xxdAddress Constant
5   hi def link xxdSep      Identifier
6   hi def link xxdAscii   Statement
7
8 endif
```

外面的条件语句不是惯用法，我们可以忽略。里面重要的是三个 `hi def link` 语句，拼写完整的话是 `highlight default link`（可参见帮助 [@:help :highlight-link](#)）。这三个语句建立了默认的语法加亮链接组，也就是，在用户没有自己在 `vimrc` 配置文件中 `highlight link` 来修改语法加亮时，默认的语法项目和加亮组之间的关系。目前，地址 `xxdAddress` 使用常数 `Constant` 的加亮方式，冒号分隔符 `xxdSep` 使用标识符 `Identifier` 的加亮方式，ASCII 文本 `xxdAscii` 使用语句 `Statement` 的加亮方式。

那 `xxdDot` 到哪儿去了呢？答案是，它没有加亮组，因为我们不需要对其进行特殊加亮。虽然 Vim 会认出它使用了特殊的语法格式，在显示上它和中间的十六进制数值一样，没有任何语法加亮效果。

`Constant`、`Identifier`、`Statement` 这些加亮组，又应该以何种方式展示呢？这就是配色方案要做的事情了。如果说语法加亮是逻辑问题的话，那配色方案就是个审美问题。你要个性化的话，就靠配色方案了。

配色方案

类似地，配色方案里包含的也是一些模板语句加上色彩的定义。比如，在配色方案 koehler 里，跟 xxd 相关的核心色彩定义是：

[复制代码](#)

```
1 set background=dark
2 hi Normal      guifg=white  guibg=black
3 hi Constant    term=underline cterm=bold ctermfg=magenta guifg=#ffa0a0
4 hi Identifier  term=underline ctermfg=brown  guifg=#40ffff
5 hi Statement   term=bold    cterm=bold ctermfg=yellow  gui=bold  guifg=#ffff6
```

首先，这个配色方案设定背景为 dark，深色（允许的另外一个值是 light，浅色背景）。这会调整缺省的颜色组，使得文字色彩在深色背景上显示比较友好。但这不会在终端里真正改变背景（仍要靠下面的背景色设定），因此，如果你在浅色背景的终端里使用这个配色方案，会显得不太友好。有些比较好的配色方案会采用相反的做法，根据目前是深色还是浅色背景，采用不同的配色。

对于“正常”（Normal）的加亮组，这个配色方案采用了最直截了当的前景白、背景黑。可以预见，这个配色会比较醒（cì）目（yǎn）。

对于 Constant 加亮组，这个配色方案就稍微复杂点了，分了单色终端、色彩终端和图形界面的不同配色。古老的单色终端里使用下划线（应该已经没人用吧，所以以后我就忽略这种设定了）；色彩终端下使用粗体和紫色前景；图形界面指定了前景色为 RGB 色彩 #ffa0a0，亮棕色。

Identifier 加亮组也类似，色彩终端下使用棕色前景，图形界面下前景色则是 RGB 色彩 #40ffff，亮青色。

Statement 加亮组在色彩终端和图形界面下都使用粗体，色彩终端使用黄色前景色，图形界面使用前景色是 RGB 色彩 #ffff60，亮黄色。

使用这个配色方案在图形界面和色彩终端下的效果，如下面的截图所示：

```
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452 .PNG.....IHDR
00000010: 0000 0258 0000 0258 0806 0000 00be 6698 ...X...X.....f.
00000020: dc00 0000 0970 4859 7300 000f 6100 000f .....pHYs...a...
00000030: 6101 a83f a769 0000 2000 4944 4154 789c a..?.i...IDATx.
```

图形界面下 koehler 配色方案的效果

```
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452 .PNG.....IHDR
00000010: 0000 0258 0000 0258 0806 0000 00be 6698 ...X...X.....f.
00000020: dc00 0000 0970 4859 7300 000f 6100 000f .....pHYs...a...
00000030: 6101 a83f a769 0000 2000 4944 4154 789c a..?.i...IDATx.
```

色彩终端下 koehler 配色方案的效果

配色方案在终端下的优化

说到这里，我们有必要来讨论一下 Vim 里允许使用的色彩数量。在图形界面 Vim 里，色彩是 Vim 本身调用系统的编程接口来控制的，可以使用 RGB 的所有 16,777,216 种不同颜色。但在终端里，Vim 会受到终端能力的限制，只能根据终端的能力来显示色彩。根据终端的类型，我们可以分为 4 种情况：

第 1 种是最古老的是单色终端，没有颜色，只能使用下划线、粗体等效果。效果定义使用 `term=...` 的形式。今天，我们应该基本碰不到这样的环境了。

第 2 种是 8/16 色终端，允许使用最基本的八种颜色（黑、红、绿、黄、蓝、紫、青、白），以及这些颜色的较亮变体（即使 8 色终端一般也能在前景色上使用加亮的变体）。我们可以使用 `cterm=...` 定义粗体等效果（由于兼容性问题，不常用），用 `ctermfg=...` 和 `ctermbg=...` 定义前景和背景色，其中可以使用英文色彩名称或序号（见 [🔗:help cterm-colors](#)）。鉴于序号在不同的环境里可能是不同的，我们一般使用色彩名称。如果你使用非图形界面终端，可能会遇到这种情况，但这应当也很不常见了吧。

这些颜色虽然是标准的，但很多终端允许用户调整这些颜色，以达到最好的色彩组合效果。比如，下图是 macOS 里终端应用的一个设置界面，其中的“ANSI 颜色”就是用户可以调整的 16 种“标准色”：



macOS 终端应用的文本设置界面

第 3 种是 256 色终端，用户可以选择预先定义的 256 种颜色之一，这在目前的终端里是非常主流的方式了。你可以在网上很方便地找到 [脚本](#) 来输出这些颜色，效果如下图所示：

System colors:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15

6x6x6 color cube and greyscale:

16	22	28	34	40	46	52	58	64	70	76	82	88	94	100	106	112	118
17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113	119
18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120
19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109	115	121
20	26	32	38	44	50	56	62	68	74	80	86	92	98	104	110	116	122
21	27	33	39	45	51	57	63	69	75	81	87	93	99	105	111	117	123
124	130	136	142	148	154	160	166	172	178	184	190	196	202	208	214	220	226
125	131	137	143	149	155	161	167	173	179	185	191	197	203	209	215	221	227
126	132	138	144	150	156	162	168	174	180	186	192	198	204	210	216	222	228
127	133	139	145	151	157	163	169	175	181	187	193	199	205	211	217	223	229
128	134	140	146	152	158	164	170	176	182	188	194	200	206	212	218	224	230
129	135	141	147	153	159	165	171	177	183	189	195	201	207	213	219	225	231
232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249
250	251	252	253	254	255												

在 iTerm2 下使用脚本输出的 256 种颜色

要选择这 256 种颜色中的一种，方式不太直观：你需要使用 `ctermfg=...` 和 `ctermbg=...`，并直接写出这 256 种颜色之一的编号。

这 256 种颜色都可以算是标准的，它们的标准 RGB 值有明确的定义。头 16 种颜色就是上面的“ANSI 颜色”，在终端里常常可以直接调整，图中也可以看到和前面图里的颜色已经有明显的不同。虽然界面只提供了头 16 种颜色的调整，但为了达到最佳的显示效果，你也可以编程修改这 256 种颜色的调色板。

第 4 种是支持真彩 (truecolor) 的终端，跟编程修改 256 色的调色板相比，这是更简单的做法。下面是部分比较常见的支持 RGB 真彩的终端 ([此处](#)是一个更完整的列表)：

GNOME-Terminal (Linux)

iTerm2 (macOS)

mintty (Windows)

命令提示符 (Windows 10 版本 1703 及以后；在命令提示符里使用 Vim，如果不启用真彩支持，颜色可能完全错误！)

在这些终端里，终端 Vim 就能显示跟图形界面 Vim 同样多的颜色数，因而能达到最佳色彩效果。你仍需手工打开（默认关闭的）Vim 选项 `termguicolors`。此后，Vim 就会使用你在 `guifg` 和 `guibg` 中写的 RGB 色彩，也就是说，把终端当图形界面一样看待（在色彩方面）。

鉴于真彩终端的一个惯例是设置环境变量 `COLORTERM` 为 `truecolor` 或 `24bit`，我们可以在 `vimrc` 配置文件中进行检查：

[复制代码](#)

```
1 if has('termguicolors') &&  
2     \($COLORTERM == 'truecolor' || $COLORTERM == '24bit')  
3     set termguicolors  
4 endif
```

不过，这个检查方式仅限于类 Unix 平台。对于 Windows，Vim 提供了另外一个专门的特性检查项：

[复制代码](#)

```
1 if has('vcon')  
2     set termguicolors  
3 endif
```

推荐配色方案

上面我们分析的 `koehler`，算是 Vim 内置的配色方案中比较中规中矩的一个，可用，但不那么好看。如果你想要一个漂亮的配色方案，还是不应该在内置配色方案里寻找。

一个广受好评的 Vim 配色方案是 [gruvbox](#)（包管理器中的安装名称是 `"morhetz/gruvbox"`）。它不仅是一个支持深色背景和浅色背景的配色方案，而且还特意确保自己能 and Vim 的一些最流行插件兼容。下面是这个配色方案的示意截图：

```

functional.h (~/Programming/nvwa/nvwa) - VIM

/**
 * Applies a function cumulatively to all elements of a tuple.
 *
 * @param f      the function to apply
 * @param value  the first argument to be passed to the function
 * @param args   the input tuple
 * @pre         \a f shall take one argument of the result type, and
 *              one argument of the type of the elements in \a args.
 */
template <typename _Rs, typename _Fn, typename... _Targs>
constexpr auto reduce(_Fn&& f,
                      const std::tuple<_Targs...>& args,
                      _Rs&& value)
{
    return detail::tuple_reduce_impl(std::forward<_Fn>(f),
                                     std::forward<_Rs>(value),
                                     args,
                                     std::index_sequence_for<_Targs...>());
}
functional.h [Git(master)] [utf-8] 683,1 67%

```

浅色背景下的 gruvbox 效果图

```

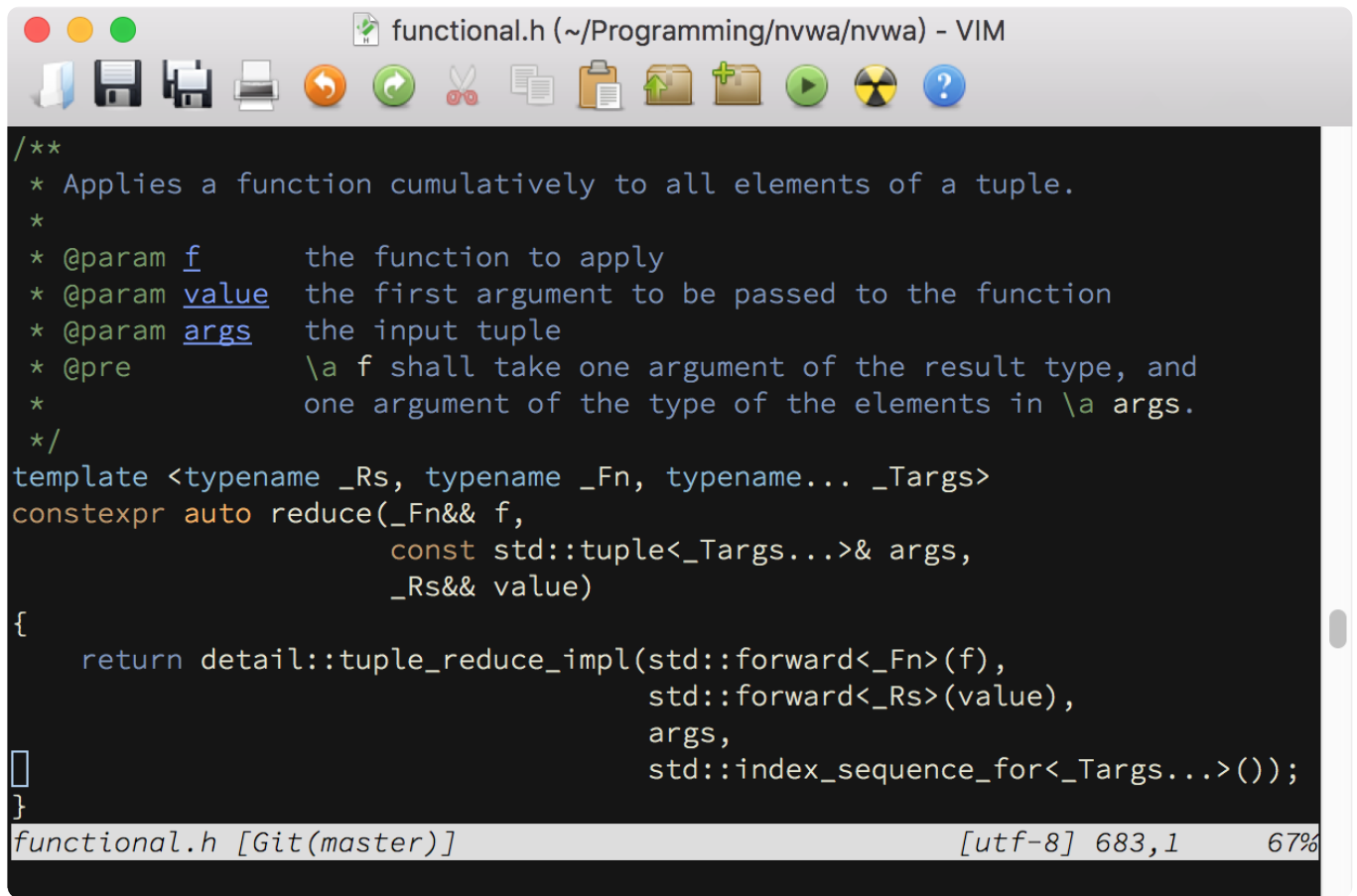
functional.h (~/Programming/nvwa/nvwa) - VIM

/**
 * Applies a function cumulatively to all elements of a tuple.
 *
 * @param f      the function to apply
 * @param value  the first argument to be passed to the function
 * @param args   the input tuple
 * @pre         \a f shall take one argument of the result type, and
 *              one argument of the type of the elements in \a args.
 */
template <typename _Rs, typename _Fn, typename... _Targs>
constexpr auto reduce(_Fn&& f,
                      const std::tuple<_Targs...>& args,
                      _Rs&& value)
{
    return detail::tuple_reduce_impl(std::forward<_Fn>(f),
                                     std::forward<_Rs>(value),
                                     args,
                                     std::index_sequence_for<_Targs...>());
}
functional.h [Git(master)] [utf-8] 683,1 67%

```

深色背景下的 gruvbox 效果图

我觉得 gruvbox 在深色背景下还是挺漂亮的。它的颜色总体偏暖, 而你如果喜欢深色背景下较为清冷的色彩, 我觉得 [jellybeans](#) (包管理器中的安装名称是 “nanotech/jellybeans.vim”) 还不错:



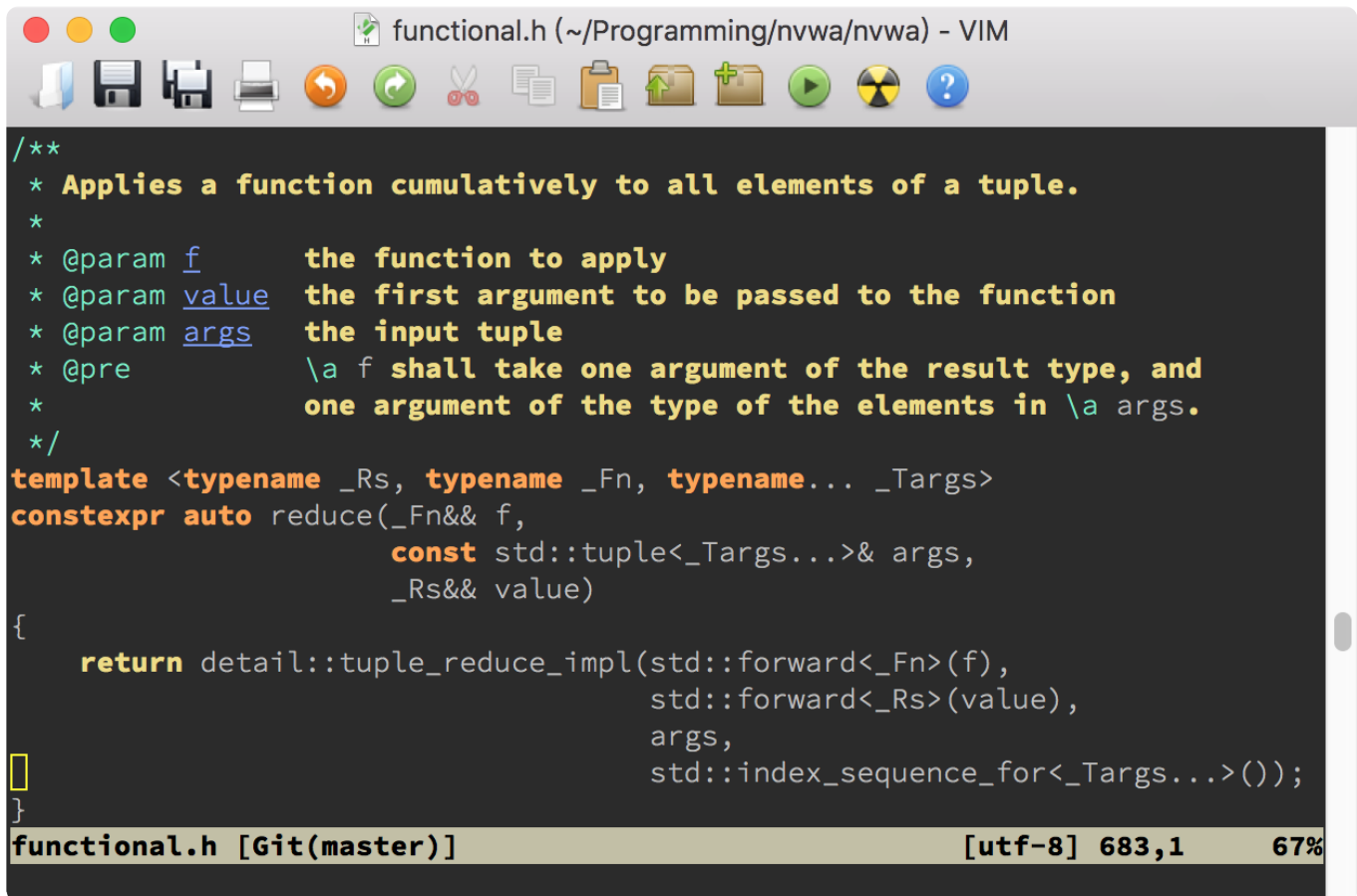
```
functional.h (~/Programming/nvwa/nvwa) - VIM

/**
 * Applies a function cumulatively to all elements of a tuple.
 *
 * @param f      the function to apply
 * @param value  the first argument to be passed to the function
 * @param args   the input tuple
 * @pre         \a f shall take one argument of the result type, and
 *              one argument of the type of the elements in \a args.
 */
template <typename _Rs, typename _Fn, typename... _Targs>
constexpr auto reduce(_Fn&& f,
                     const std::tuple<_Targs...>& args,
                     _Rs&& value)
{
    return detail::tuple_reduce_impl(std::forward<_Fn>(f),
                                     std::forward<_Rs>(value),
                                     args,
                                     std::index_sequence_for<_Targs...>());
}

functional.h [Git(master)] [utf-8] 683,1 67%
```

使用深色背景的 jellybeans 效果图

不过, 这两种方案我都是偶尔使用, 我使用更多的还是明白 (mbbill) 设计的 [desertEx](#) (包管理器中的安装名称是 “mbbill/desertEx”) 的一个版本。它的特点是无加亮的普通文字对比度设得比较低, 读起来比较轻松 (但加亮部分效果仍然比较强烈)。如果你想试试这个方案的话, 可以自己安装尝试一下:



```
functional.h (~/Programming/nvwa/nvwa) - VIM

/**
 * Applies a function cumulatively to all elements of a tuple.
 *
 * @param f      the function to apply
 * @param value  the first argument to be passed to the function
 * @param args   the input tuple
 * @pre         \a f shall take one argument of the result type, and
 *              one argument of the type of the elements in \a args.
 */
template <typename _Rs, typename _Fn, typename... _Targs>
constexpr auto reduce(_Fn&& f,
                     const std::tuple<_Targs...>& args,
                     _Rs&& value)
{
    return detail::tuple_reduce_impl(std::forward<_Fn>(f),
                                     std::forward<_Rs>(value),
                                     args,
                                     std::index_sequence_for<_Targs...>());
}

functional.h [Git(master)] [utf-8] 683,1 67%
```

我用的 desertEx 效果图 (非最新版本)

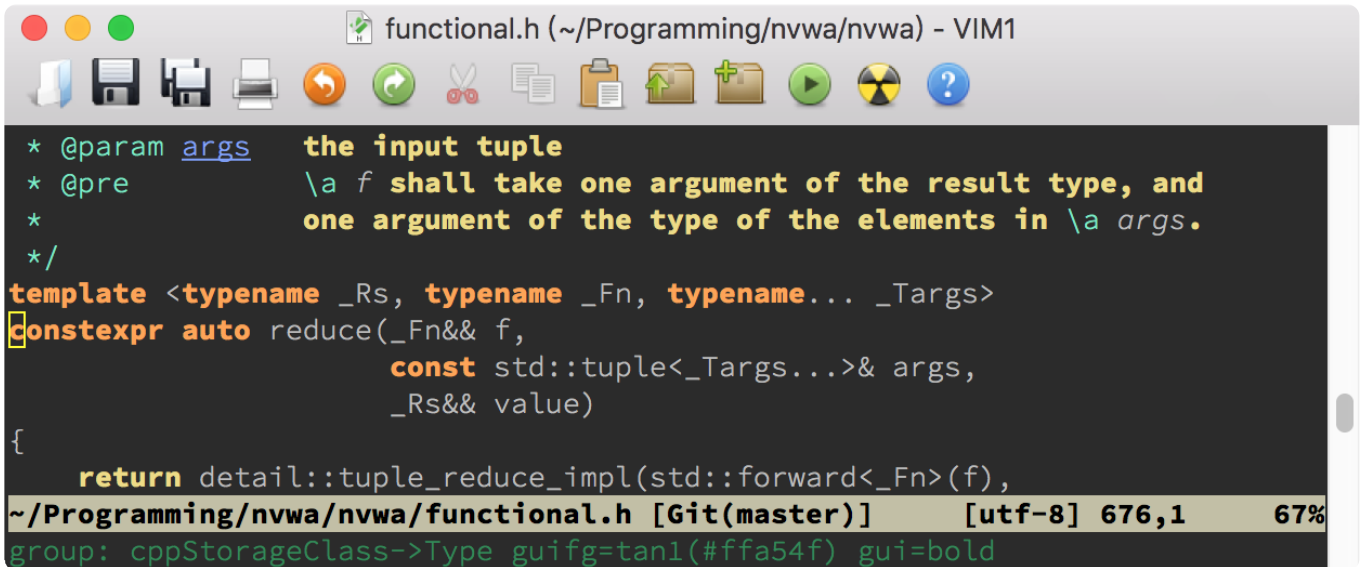
检查 / 调试配色方案

如果你想自己对配色方案进行调整的话, 有一个小工具肯定会非常有用, 那就是 vim-scripts/SyntaxAttr.vim。不过, 这个插件不会自己添加键映射, 需要你在用包管理器安装之后, 自己在 vimrc 配置文件中加入类似下面的语句:

```
1 noremap <Leader>a :call SyntaxAttr()<CR>
```

复制代码

这样, 我们就能用 \a 来检查光标下面的语法高亮详情了。下面是一个示例:



```
* @param args    the input tuple
* @pre           \a f shall take one argument of the result type, and
*               one argument of the type of the elements in \a args.
*/
template <typename _Rs, typename _Fn, typename... _Targs>
constexpr auto reduce(_Fn&& f,
                     const std::tuple<_Targs...>& args,
                     _Rs&& value)
{
    return detail::tuple_reduce_impl(std::forward<_Fn>(f),
    ~/Programming/nvwa/nvwa/functional.h [Git(master)] [utf-8] 676,1 67%
group: cppStorageClass->Type guifg=tan1(#ffa54f) gui=bold
```

检查光标下文本的加亮组

从上面可以看到，`constexpr` 属于 `cppStorageClass` 语法加亮组（这是在 `syntax/cpp.vim` 中定义的），并且被链接到了 `Type` 加亮组。后面的 `guifg` 和 `gui` 设定就是 `Type` 加亮组的内容：使用色彩 `tan1`（RGB 值为 `#ffa54f`），特殊效果为粗体（`bold`）。

在你自己设计、调试语法文件或配色方案时，你会发现这个工具非常有用。

输出加亮效果

作为一个文本编辑器，Vim 只接受文本的复制和粘贴。如果你想要在一个（非 Markdown）文档中展示有语法加亮的代码，Vim 也是可以用来产生这样的代码的——通过 HTML 输出。

Vim 默认就提供了 `:TOhtml` 命令（参见 [第 4 讲](#)中讨论的系统内置插件），可以把当前展示的语法加亮效果输出为一个 HTML 文件。你可以根据最终文档的要求，选择合适的深色或浅色配色方案，然后使用该命令来输出 HTML。这个命令默认输出整个文件，你也可以自己在可视模式下选定范围，或者用逗号隔开的行号选定范围，这样 `:TOhtml` 命令就只会输出选定范围的 HTML 代码。

这个方式灵活是挺灵活，但不能直接把带色彩加亮的文本贴到文档里去，终究还是不太方便。令人欣慰的是，早就有人找到了在各个平台上把 HTML 代码转成剪贴板富文本的方法。我最近对一个 Vim 插件 [vim-copy-as-rtf](#) 作了点改造，使其可以在我们现在讲的三大主流平台（macOS、Linux 和 Windows）上都可以直接复制出带语法加亮的代码。在

macOS 和 Windows 上，没有特别的配置要求；在 Linux 桌面环境下，我们要求系统必须装有 xclip 工具。这样，我们只需要在使用 TOhtml 的地方，把命令改成 CopyRTF 就能把加亮的代码复制到系统的剪贴板中。

内容小结

好了，我们现在来总结一下今天学到的内容：

Vim 使用正则表达式来匹配代码，把代码分到不同的“语法项目”里。

语法项目可以跟加亮组进行链接，加亮组可以针对终端和图形界面定义不同的显示方式。

主流的终端可以显示 256 色，某些终端已经可以跟图形界面一样显示 RGB 真彩。

我推荐了 gruvbox、jellybeans 和 desertEx 三个配色方案；当然，你也可以自己在网上找其他好的配色方案。

SyntaxAttr.vim 可以显示语法加亮组的细节，可以用来帮助调试语法加亮和配色方案。

Vim 的加亮效果可以输出成 HTML 文件，也可以复制到剪贴板中成为带语法加亮的富文本，方便在办公文档中使用。

本讲我们的配置文件有一些改动，对应的标签是 l12-unix 和 l12-windows。

课后练习

想一想，如果希望颜色在 8/16 色终端下和 256 色终端下都有效，并且在 256 色终端下使用 256 色的配置，该怎么做？

在自己尝试解决这个问题之后，可以看一下 jellybeans 里面的颜色定义，了解它是如何解决在不同终端下面的颜色一致性问题的。

如果有任何问题，欢迎留言和我交流。

我是吴咏炜，我们下一讲再见！

提建议

更多课程推荐

Java 业务开发 常见错误 100 例

>>> 全面避坑 + 最佳实践 = 健壮代码

朱晔

贝壳金服资深架构师



涨价倒计时 🕒

今日秒杀 **¥79**，8月8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 11 | 文本的细节：关于字符、编码、行你所需要知道的一切

下一篇 拓展1 | 纯文本编辑：使用 Vim 书写中英文文档

精选留言 (3)

写留言



我来也

2020-08-21

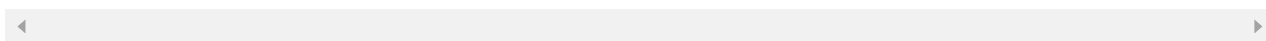
今天收获最大的就是这个`termguicolors`参数了。
有了它,vim的显示效果立马提升了好几个档次.💎💎💎💎💎

最近也折腾过256色,但总感觉配置的颜色跟实际显示的有些偏差.

...

展开 ∨

作者回复: 一直坐沙发.....:-)



1

3



我来也

2020-08-21

课后思考题:

与想象的差不多,根据当前环境来调整配色的参数.

[jellybeans](<https://github.com/nanotech/jellybeans.vim/blob/master/colors/jellybeans.vim>)...

展开 ▾

1

1



AirY

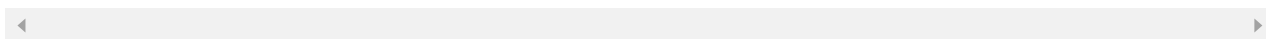
2020-08-24

大佬，求帮助，我配置Vundle插件后，sudo vim会报错

作者回复: 建议你先去查阅一下，怎样问问题.....

我完全不知道你的问题出在哪里，怎么回答你？我只能说的，我的 sudo vim 是没问题的。

但我一般也不这么用，因为容易出权限问题。如果我需要用 root 权限编辑的话，我更愿意先 sudo su -l，再进行编辑。



1

1