

27 | 命令行：不只是酷，更重要的是能提高个人效能

2019-10-25 葛俊

研发效率破局之道

[进入课程 >](#)



讲述：葛俊

时长 18:03 大小 16.55M



你好，我是葛俊。今天，我要与你分享的主题是，命令行下的高效工作技巧。

我先和你讲一个有意思的话题吧。命令行工具常常会给人一种黑客的感觉，好莱坞的电影里面常常出现命令行窗口的使用。不知道你听说过没有，很多好莱坞电影在拍摄时使用的其实是一个叫作 [nmap](#) 的工具。这个工具是做安全扫描的，只不过因为它的显示特别花哨，所以被很多电影采用。在 [nmap](#) 官方网站上，还专门列出来了这些电影的名单。

类似这种可以让自己看起来很忙的工具还有很多，比如 [Genact](#)。下面是一个使用 [Genact](#) 的录屏，当然这里的命令并没有真正运行。这可能是整个专栏中，唯一一个让你看起来效率很高，实际上却是降低效率的工具，但说不定对你有用，你懂的。

```
genact-osx (genact-osx)
+ Sending login information... [done]
+ Sending command... [done]
>> Botnet update complete.
Establishing connections: 1737/1737
Cluster #00 (188 nodes) [online]
Cluster #01 (176 nodes) [online]
Cluster #02 (109 nodes) [online]
Cluster #03 (141 nodes) [online]
Cluster #04 (158 nodes) [online]
Cluster #05 (141 nodes) [online]
Cluster #06 (151 nodes) [online]
Cluster #07 (120 nodes) [online]
Cluster #08 (196 nodes) [online]
Cluster #09 (122 nodes) [online]
Cluster #10 (134 nodes) [online]
Cluster #11 (101 nodes) [online]
+ Synchronizing clocks... [done]
+ Sending login information... [done]
+ Sending command... [done]
>> Botnet update complete.
Establishing connections: 1725/1725
Cluster #00 (195 nodes) [online]
Cluster #01 (133 nodes) [online]
Cluster #02 (158 nodes) [online]
Cluster #03 (163 nodes) [online]
Cluster #04 (115 nodes) [online]
Cluster #05 (126 nodes) [online]
Cluster #06 (178 nodes) [online]
Cluster #07 (198 nodes) [online]
Cluster #08 (122 nodes) [online]
Cluster #09 (174 nodes) [online]
Cluster #10 (163 nodes) [online]
+ Synchronizing clocks... [done]
+ Sending login information... [done]
+ Sending command... [done]
>> Botnet update complete.
Establishing connections: 1234/1234
Cluster #00 (138 nodes) [online]
Cluster #01 (165 nodes) [booting]
Cluster #02 (121 nodes) [booting]
Cluster #03 (184 nodes) [booting]
Cluster #04 (132 nodes) [booting]
Cluster #05 (117 nodes) [booting]
Cluster #06 (127 nodes) [online]
Cluster #07 (130 nodes) [online]
Cluster #08 (120 nodes) [booting]

genact-osx
==> Starting build: 1.5.19(0.150/4/2)
-> Running build hook: [base]
-> Running build hook: [udev]
-> Running build hook: [autodetect]
-> Running build hook: [modconf]
-> Running build hook: [block]
-> Running build hook: [net]
-> Running build hook: [mdadm_udev]
-> Running build hook: [keymap]
-> Running build hook: [encrypt]
-> Running build hook: [fsck]
-> Running build hook: [filesystems]
==> Generating module dependencies
==> Creating lz4-compressed initcpio image: /boot/initramfs-linux.img
==> Image generation successful
==> Building image from preset: /etc/mkinitcpio.d/linux.preset: 'fallback'
-> -k /boot/vmlinuz-linux -c /etc/mkinitcpio.conf -g /boot/initramfs-linux-fallback.img
==> Starting build: 1.5.19(0.150/4/2)
-> Running build hook: [base]

genact-osx
2bffa63ba6f79fb0 4e a2 a0 36 d1 d0 30 ac ad 52 54 60 67 0e 9d 05 N..6..0..RT`g...
2bffa63ba6f79fc0 8b 4d e4 55 9f 6c bb 52 e3 ff a9 86 f1 28 41 12 .M.U.l.R....(A.
2bffa63ba6f79fd0 cb 51 f0 07 0a 91 d4 3d 15 d7 c5 7f e3 e2 20 0b .Q.....
2bffa63ba6f79fe0 5d e1 c4 41 ea 2e c6 4b c1 f2 da 4e 53 7b e0 60 ].A...K...NS[.
2bffa63ba6f79ff0 6f 62 f5 c6 7f 14 c6 27 02 e9 22 9e 8b ff c8 ec ob....."
2bffa63ba6f7a000 bc bc 41 23 f0 8b b6 52 b1 7d 1c 9a 3c 53 8f c ..A#...R.).<S.
2bffa63ba6f7a010 89 31 3f f9 c6 6e e2 83 9b 85 b9 3f e5 00 3a 88 .1?..f.....?..
2bffa63ba6f7a020 59 91 a9 a3 0a 22 93 06 bd a3 fb e9 60 d3 41 06 Y.....A.
2bffa63ba6f7a030 4c 00 08 f8 63 1f 02 7d 18 7a 9c 89 9e 69 83 6b L...c..).z...i.k
2bffa63ba6f7a040 70 1c 2f 00 25 2b db 3a 5c 9c 73 f7 b7 76 3a 4e p./.%+;|.s.v:N
2bffa63ba6f7a050 f5 2e a8 2b 5b ae a7 59 8d 9d 9e 3e 2e 7b be 49 .+.[.Y...>.{.I
2bffa63ba6f7a060 38 d5 18 4e c1 ec 9f 57 a8 ba 34 73 fb 6c 07 f9 8..N...W..4s.L..
2bffa63ba6f7a070 57 62 17 f5 ee 3f f6 01 63 47 65 5e aa e1 74 61 Wb...?..cGe^..ta
2bffa63ba6f7a080 57 1b c2 9c a3 3d e4 25 7b b3 3f 92 2b 2a 48 77 W....%{.?.+*Hw
2bffa63ba6f7a090 ff 58 46 72 c8 7d f5 65 00 a1 1f d9 5d a3 06 8e .XFR.}.e....]
2bffa63ba6f7a0a0 d8 3b fe fa 50 a2 c2 8b 84 d3 44 7f 73 c0 a5 2b .;.P...D.s.+
2bffa63ba6f7a0b0 dc 59 1f 42 20 f5 fd d1 f4 e9 75 5f 5d d8 05 94 .;B.....u]...
2bffa63ba6f7a0c0 b5 76 7f 6e 94 f9 45 80 1b b7 f3 3a b4 57 ac 87 .v.n..E.....:W.
2bffa63ba6f7a0d0 dd 82 dd 3b 57 cd 95 33 c7 6d 71 2c 73 92 58 22 ...;W..3.mq.s.X"
2bffa63ba6f7a0e0 88 67 2a 2d 7e e3 52 c5 10 a7 71 dc 82 f6 a3 8b .g*~..R...q.....
2bffa63ba6f7a0f0 97 22 db d2 59 07 16 5f 3b 48 04 e6 67 4d af f2 .Y...;H..gM..
2bffa63ba6f7a100 db a3 4c 69 e5 64 6a 6f 88 d5 7b 3b 0d 77 06 44 ..Li.djo.{};w.D.
2bffa63ba6f7a110 c3 6f ad 6c 34 c0 65 ed e0 21 7f d4 57 3c df 5c .o.l4.e...!.W<.\
2bffa63ba6f7a120 29 c3 3a 0f 99 a9 f4 50 41 a6 bc 80 44 71 00 4c ).:...PA...Dq.L
2bffa63ba6f7a130 1f c5 fa b6 35 ea 9a 5b 02 08 5b c2 f8 a6 60 7c ....5..[. [...|
2bffa63ba6f7a140 6f d3 11 56 82 cb 14 67 2d 85 20 32 c2 67 36 f4 o..V...g...2.g6.
```

讲完这个娱乐性的话题，我们进入正题吧。

为什么要使用命令行？

GUI 图形界面的出现是计算机技术的变革，极大方便了用户。但在这么多年后的今天，命令行工具为什么仍然有如此强大的生命力呢？

在我看来，对软件工程师来说，想要高效开发就必须掌握命令行，主要原因包括：

虽然鼠标的移动和点击比较直观，但要完成重复性的工作，使用键盘会快捷得多。这一点从超市的结算人员就可以看出来，使用键盘系统的收银员总是噼里啪啦地很快就可以完成结算，而使用鼠标点击的话明显慢很多。

作为开发人员，可以比较容易地使用命令行的脚本，对自己的工作进行自动化，以及其他系统工具联动。但使用 GUI 的话，就会困难得多。

命令行通常可以暴露更完整的功能，让使用者对整个系统有更透彻的理解。Git 就是一个典型的例子，再好的 GUI Git 系统都只能封装一部分 Git 命令行功能，要想真正了解 Git，我们必须使用命令行。

有一些情况是必须使用命令行的，比如 SSH 到远程服务器上工作的时候。

为了演示命令行的强大功能给我们带来的方便，下面是一个在本地查看文件并上传到服务器的流程的录屏。

```
1. jasonge@Juns-MacBook-Pro-2: ~ (zsh)
Error: ENOENT: no such file or directory, stat '/Users/jasonge/Downloads/Mailbox
(Beta) 2.app/Contents/Frameworks/Sparkle.framework/Resources/fr_CA.lproj'
  at Object.statSync (fs.js:855:3)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:136:22)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at Object.<anonymous> (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/alder.js:193:1)
  at Module._compile (internal/modules/cjs/loader.js:776:30)
^R
jasong@Juns-MacBook-Pro-2 ~ 1541 13:56:38
```

```
1. jasonge@Juns-MacBook-Pro-2: ~ (zsh)
Error: ENOENT: no such file or directory, stat '/Users/jasonge/Downloads/Mailbox
(Beta) 2.app/Contents/Frameworks/Sparkle.framework/Resources/fr_CA.lproj'
  at Object.statSync (fs.js:855:3)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:136:22)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at buildTree (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/
alder.js:184:7)
  at Object.<anonymous> (/Users/jasonge/.config/yarn/global/node_modules/@aweary/alder/alder.js:193:1)
  at Module._compile (internal/modules/cjs/loader.js:776:30)
^R
jasong@Juns-MacBook-Pro-2 ~ 1541 13:56:38
```

通过这个案例，你可以看到命令行的以下几个功能：

在提示行会高亮显示时间、机器名、用户名、Git 代码仓的分支和状态，以及上一个命令的完成状态。

输入命令的时候，高亮显示错误并自动纠错。

使用交互的形式进行文件夹的跳转，并方便查找文件，还可以直接在命令行里显示图片。

使用交互的工具，把文件上传到远端的服务器，并快速连接到远端查看传输是否成功。

整个流程全部都是在命令行里完成的，速度非常快，用户体验也非常好。正因为如此，我看到的硅谷特别高效的开发人员，绝大多数都大量使用命令行。那，面对成百上千的命令行工具，我们**怎样才能高效地学习和使用**呢？

我将高效学习使用命令行的过程，归纳为两大步：

1. 配置好环境；
2. 针对自己最常使用命令行的场景，有的放矢地选择工具。

今天，我就与你详细讲述**环境配置**这个话题。而关于选择工具的话题，我会在下一篇文章中与你详细介绍。总结来讲，环境配置主要包括以下四步：

1. 选择模拟终端；
2. 选择 Shell；
3. 具体的 Shell 配置；
4. 远程 SSH 的处理。

这里需要注意的是，在命令行方面，macOS 和 Linux 系统比 Windows 系统要强大许多，所以我主要以 macOS 和 Linux 系统来介绍，而关于 Windows 的环境配置，我只会捎带提一下。不过，macOS 和 Linux 系统中的工具选择和配置思路，你可以借鉴到 Windows 系统中。

第一步，选择模拟终端

我将一个好的终端应该具有的特征，归纳为 4 个：

快，稳定；

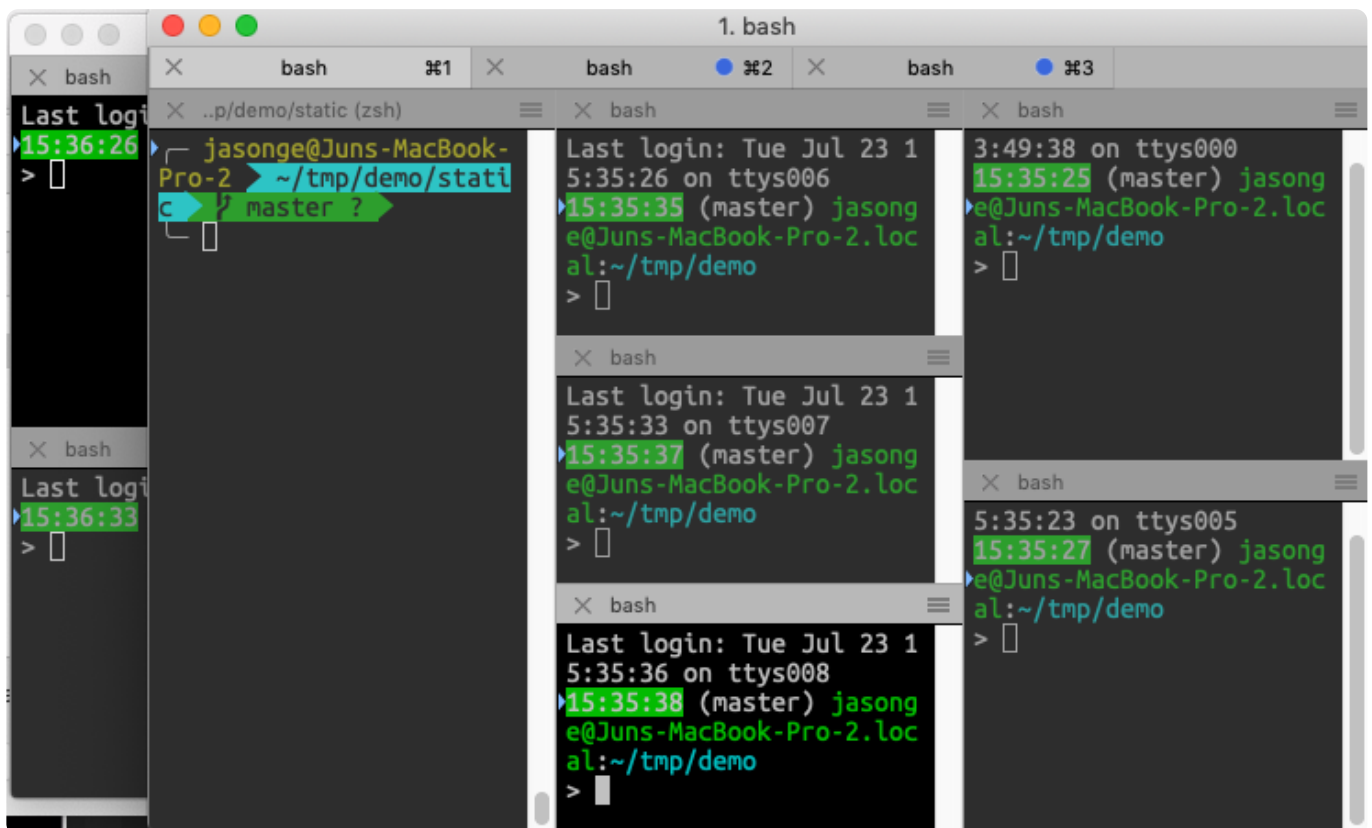
支持多终端，比如可以方便地水平和纵向分屏，有 tab 等；

方便配置字体颜色等；

方便管理常用 SSH 的登录。

macOS 系统自带的终端不太好用，常见的替代工具有 iTerm2、Terminator、Hyper 和 Uterm。我平时使用的 iTerm2，是一个免费软件，功能强大，具备上面提到的 4 个特征。下面我就以 iTerm2 为例展开介绍。其他几个工具上也有类似功能，所以你不必担心。

在多终端的场景方面，iTerm2 支持多窗口、多标签页，同一窗口中可以进行多次水平和纵向分屏。这些操作以及窗口的跳转都有快捷键支持，你可以很方便地在网络上搜索到。



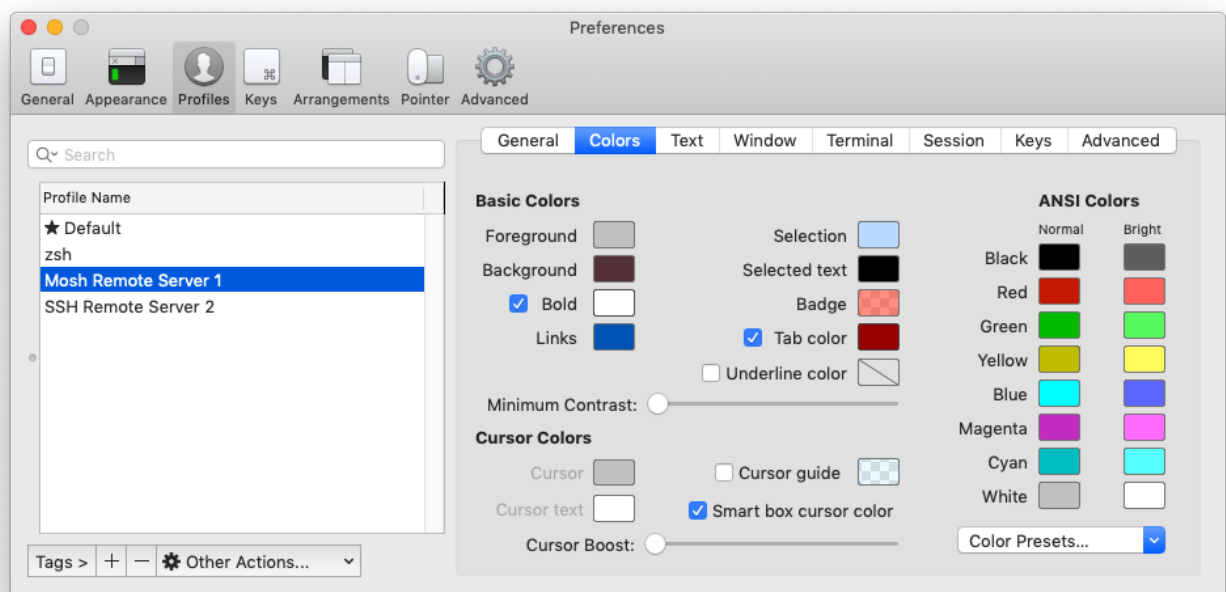
在管理常用 SSH 的登录方面，iTerm2 使用 Profile（用户画像）来控制。比如，下面是一个连接到远程服务器案例的录屏。

可以看到，在我的工作环境里常会用到 4 个 Profile，其中有两个是连接到远端服务器的，包括 Mosh Remote Server 1 和 SSH Remote Server 2。工作时，我使用 Cmd+O，然

后选择 Server 1 这个 Profile，就可以打开一个新窗口，连接到这个远程服务器上。



每一个 Profile 都可以定义自己的字体、颜色、shell 命令等。比如，Server 1 是类生产服务器，我就把背景设置成了棕红色，提醒自己在这个机器上工作时一定要小心。所以在上面的录屏中你可以看到，连接到远端的 SSH 标签页，它的背景、标签页都是棕红色。另外，下面是如何对 Profile 颜色进行设置的截屏。



除了这些基础功能外，iTerm2 还有很多贴心的设计。比如：

在屏幕中显示运行历史 (Cmd+Opt+B/F) 。有些情况下，向上滚动终端并不能看到之前的历史，比如运行 VIM 或者 Tmux 的时候。这时，浏览显示历史就特别有用了。

高亮显示当前编辑位置，包括高亮显示当前行 (Cmd+Opt+;) 高亮显示光标所在位置 (Cmd+/) 。

与上一次运行命令相关的操作，包括显示上一次运行命令的地方 (Cmd+Shift+up) ，选中上一个命令的输出 (Cmd+Shift+A) 。

其中第 2、3 项功能是由一组 [macOS 的集成工具](#) 提供的。这个工具集还包括显示图片的命令 `imgls`、`imgcat`，显示自动补全命令，显示时间、注释，以及在主窗口旁显示额外信息等。这些设计虽然很小，但非常实用。

关于 Windows 系统，2019 年 5 月微软推出了 [Windows Terminal](#)，支持多 Tab，定制性很强，据说体验很不错。

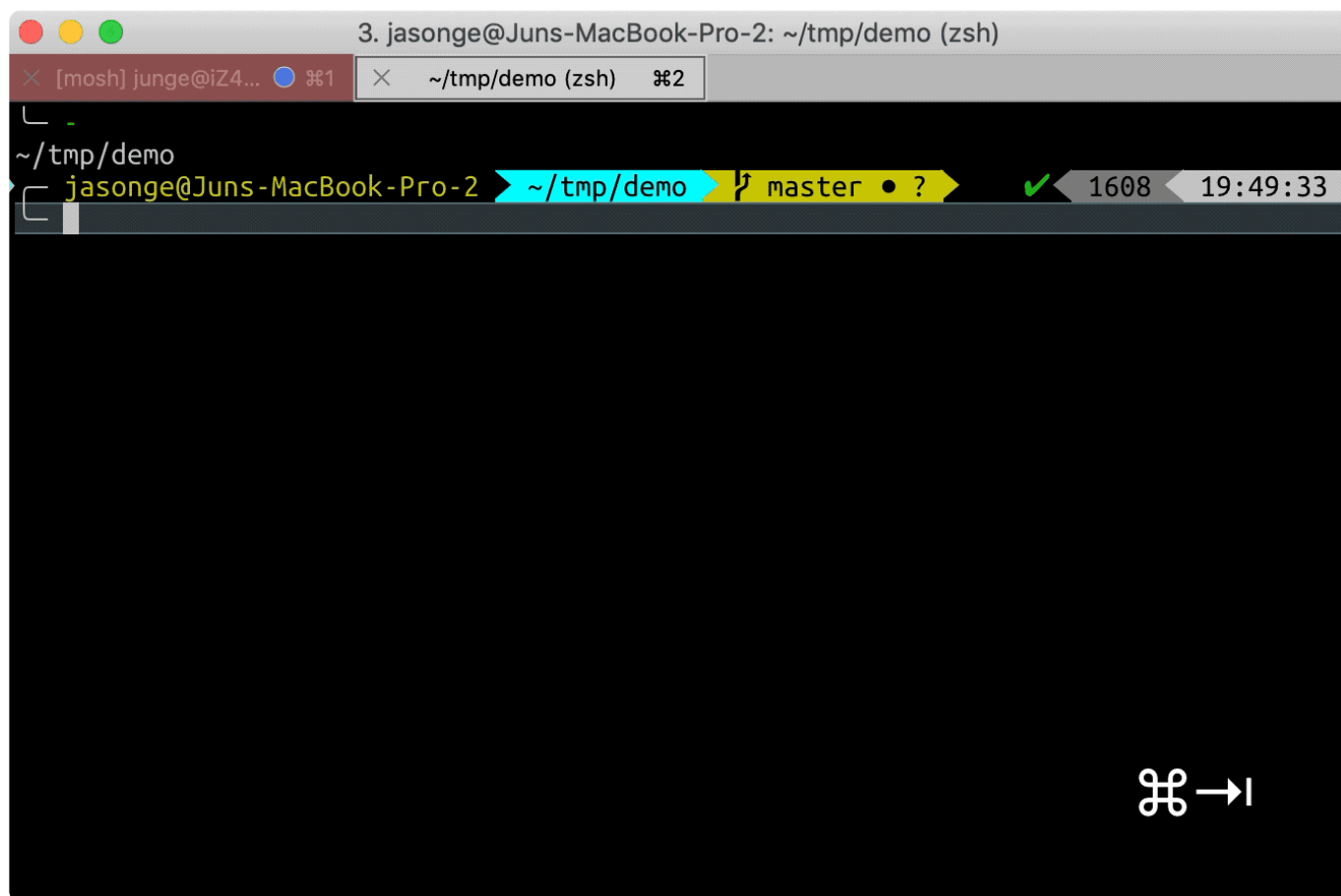
选择好了终端，环境设置的第二步就是选择 Shell。

第二步，选择 Shell

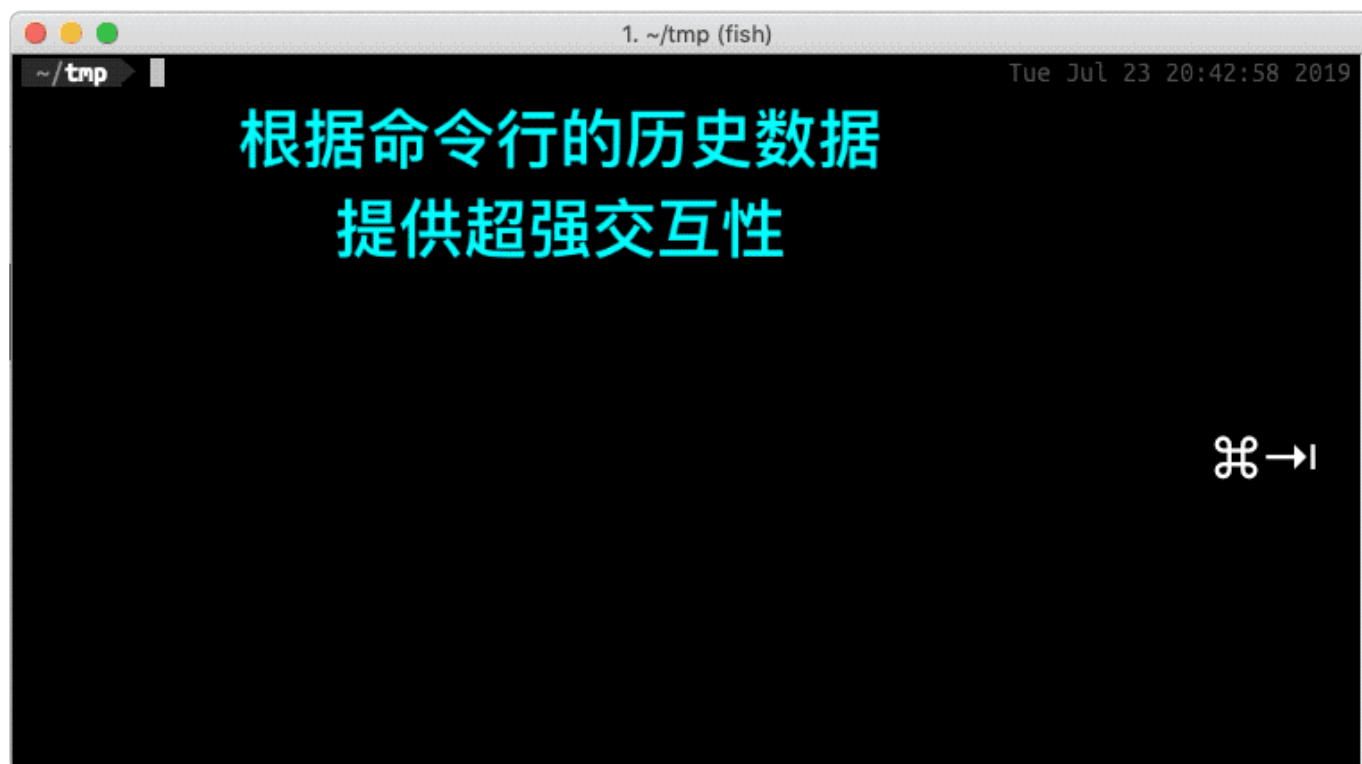
在我看来，选择 Shell 主要有普遍性和易用性这两条原则。

Linux/Unix 系统下，**Bash** 最普遍、用户群最广，但是易用性不是很好。常用来替代 Bash 的工具 **Zsh** 和 **Fish**，它们的易用性都很好。下面是两张图片，用于展示 Zsh 和 Fish 在易用性方面的一些功能。

Zsh:



Fish:



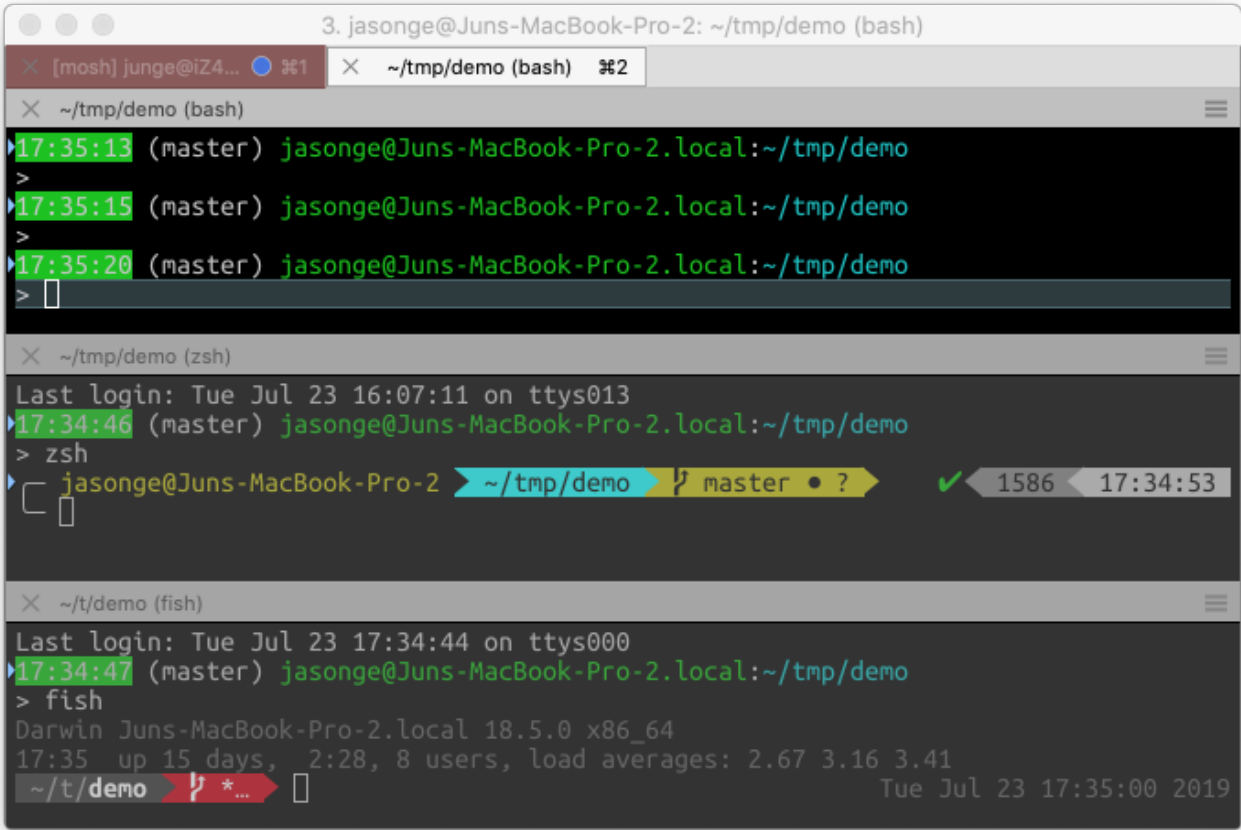
我个人觉得 Fish 比 Zsh 更方便。事实上，Fish 是 Friendly Interactive Shell 的简称。所以，交互是 Fish 的强项。可惜的是，Fish 不严格遵循 POSIX 的语法，与 Bash 的脚本不兼容，而 Zsh 则兼容，所以我目前主要使用的是 Zsh。

选好了模拟终端和 Shell 之后，便是配置环境的第三步，具体的 Shell 配置了。

第三步，具体的 Shell 配置

接下来，我以我自己使用的设置为例，向你介绍 Bash、Zsh、Fish 的具体配置吧。这里，主要包括**命令行提示符**的配置和其他配置两个方面。

之所以把命令行提示符单独提出来，是因为它一直展现在界面上，能提供很有用的价值，对命令行高效工作至关重要。下面是一张图片，展示了 Bash、Zsh 和 Fish 的命令行提示符。



这个窗口分为三部分，最上面是 Bash，中间是 Zsh，最下面是 Fish，都配置了文件路径、Git 信息和时间戳等信息。接下来，我带你一起看看这 3 个工具应该如何配置吧。

Bash 比较麻烦，配置文件包括定义颜色和命令行提示符的两部分：

[复制代码](#)

```
1 ## 文件 $HOME/.bash/term_colors, 定义颜色
2 # Basic aliases for bash terminal colors
```

```

3 N="\[\033[0m\]"      # unsets color to term's fg color
4
5 # regular colors
6 K="\[\033[0;30m\]"    # black
7 R="\[\033[0;31m\]"    # red
8 G="\[\033[0;32m\]"    # green
9 Y="\[\033[0;33m\]"    # yellow
10 B="\[\033[0;34m\]"    # blue
11 M="\[\033[0;35m\]"    # magenta
12 C="\[\033[0;36m\]"    # cyan
13 W="\[\033[0;37m\]"    # white
14
15 # empahsized (bolded) colors
16 MK="\[\033[1;30m\]"
17 MR="\[\033[1;31m\]"
18 MG="\[\033[1;32m\]"
19 MY="\[\033[1;33m\]"
20 MB="\[\033[1;34m\]"
21 MM="\[\033[1;35m\]"
22 MC="\[\033[1;36m\]"
23 MW="\[\033[1;37m\]"
24
25 # background colors
26 BGK="\[\033[40m\]"
27 BGR="\[\033[41m\]"
28 BGG="\[\033[42m\]"
29 BGY="\[\033[43m\]"
30 BGB="\[\033[44m\]"
31 BGM="\[\033[45m\]"
32 BGC="\[\033[46m\]"
33 BGW="\[\033[47m\]"

```

 复制代码

```

1 ## 文件 $HOME/.bashrc, 设置提示符及其解释
2 ##### PROMPT #####
3 # Set up the prompt colors
4 source $HOME/.bash/term_colors
5 PROMPT_COLOR=$G
6 if [ ${UID} -eq 0 ]; then
7     PROMPT_COLOR=$R ### root is a red color prompt
8 fi
9
10 #t Some good thing about this prompt:
11 # (1) The time shows when each command was executed, when I get back to my teri
12 # (2) Git information really important for git users
13 # (3) Prompt color is red if I'm root
14 # (4) The last part of the prompt can copy/paste directly into an SCP command
15 # (5) Color highlight out the current directory because it's important
16 # (6) The export PS1 is simple to understand!

```


```
17 # (7) If the prev command error codes, the prompt '>' turns red
18 export PS1="\e[42m\t\e[m$N $W"'$(_git_ps1 "(%s) ")'"$N$PROMPT_COLOR\u@\H$N:$C'
19 export PROMPT_COMMAND='if [ $? -ne 0 ]; then CURSOR_COLOR=`echo -e "\033[0;31m`'
```

命令行提示符之外的其他方面的配置，在 Bash 方面，我主要设置了一些命令行补全 (completion) 和别名设置 (alias)：

 复制代码

```
1 ## git alias
2 alias g=git
3 alias gro='git r origin/master'
4 alias grio='git r -i origin/master'
5 alias gric='git r --continue'
6 alias grio='git r --abort'
7
8
9 ## ls aliases
10 alias ls='ls -G'
11 alias la='ls -la'
12 alias ll='ls -l'
13
14
15 ## git completion, 请参考 https://github.com/git/git/blob/master/contrib/complet
16 source ~/.git-completion.bash
```

Zsh 的配置就容易得多了，而且是模块化的。基本上就是安装一个配置的框架，然后选择插件和主题即可。具体来说，我的 Zsh 命令行提示符配置步骤包括以下三步。

第一， **安装 oh-my-zsh**。这是一个对 Zsh 进行配置的常用开源框架。

 复制代码


```
1 brew install zsh
```

第二， **安装 powerline 字体**，供下一步使用。

 复制代码


```
1 brew install powerlevel9k
```

第三，在 ~/.zshrc 中配置 ZSH_THEME，指定使用 powerlevel9k 这个主题。

 复制代码

```
1 ZSH_THEME="powerlevel9k/powerlevel9k"
```

命令行提示符以外的其他配置，主要是通过安装和使用 oh-my-zsh 插件的方式来完成。下面是我使用的各种插件，供你参考。

 复制代码

```
1 ## 文件 ~/.zshrc.sh 中关 oh-my-zsh 的插件列表，具体插件细节请参考 https://github.com/r
2 plugins=(
3   git
4   z
5   vi-mode
6   zsh-syntax-highlighting
7   zsh-autosuggestions
8   osx
9   colored-man-pages
10  cating
11  web-search
12  vscode
13  docker
14  docker-compose
15  copydir
16  copyfile
17  npm
18  yarn
19  extract
20  fzf-z
21 )
22
23 source $ZSH/oh-my-zsh.sh
```


至于 Fish 的配置，和 Zsh 差不多，也是安装一个配置的框架，然后选择插件和主题即可。在配置命令行提示符时，主要步骤包括以下两步。

第一，安装配置管理框架 oh-my-fish：

 复制代码

```
1 curl -L https://get.oh-my.fish | fish
```

第二，查看、安装、使用 oh-my-fish 的某个主题，主题会自动配置好命令行提示符：

 复制代码

```
1 omf theme
2 omf install <theme>
3 omf theme <theme>
4
5 ## 我使用的是 bobthefish 主题
6 omf theme bobthefish
```

这里有一篇不错的关于 [使用 oh-my-fish 配置的文章](#)，供你参考。

Fish 的其他方面的配置，也是使用 oh-my-fish 配置会比较方便。关于具体的配置方法，建议你参照 [官方文档](#)。

关于环境的最后一个配置，是远程 SSH 的处理。

第四步，远程 SSH 的处理

SSH 到其他机器，是开发人员的常见操作，最大的痛点是，怎样保持多次连接的持久性。也就是说，连接断开以后，远端的 SSH 进程被杀死，之前的工作记录、状态丢失，导致下一次连接进去需要重新设置，交易花销太大。有两类工具可以很好地解决这个问题。

第一类工具是 Tmux 或者 Screen，这两个工具比较常见，用来管理一组窗口。

接下来，我以 Tmux 为例，与你描述其工作流程：首先 SSH 到远程服务器，然后用远程机器上的 Tmux Client 连接到已经运行的 Tmux Session 上。SSH 断开之后，Tmux Client 被杀死，但 Tmux Session 仍然保持运行，意味着命令的运行状态继续存在，下次 SSH 过去再使用 Tmux Client 连接即可。

如果你想深入了解 Tmux 的概念和搭建过程，可以参考 [这篇文章](#)。

第二类是一个保持连接不中断的工具，移动 Shell (Mobile Shell)。这也是我目前唯一见到的一个。这个工具是 MIT 做出来的，知道的人不多，是针对移动设备的网络经常断开设计的。

它的具体原理是，每次初始登录使用 SSH，之后就不再使用 SSH 了，而是使用一个基于 UDP 的 SSP 协议，能够在网络断开重连的时候自动重新连接，所以从使用者的角度来看就像从来没有断开过一样。

接下来，我以阿里云 ECS 主机、运行 Ubuntu18.04 为例，与你分享 Mosh+Tmux 的具体安装和设置方法。

第一，服务器端安装并运行 Mosh Server。

```
1 junge@iZ4i3zrhuhpdbhZ:~$ sudo apt-get install mosh
```

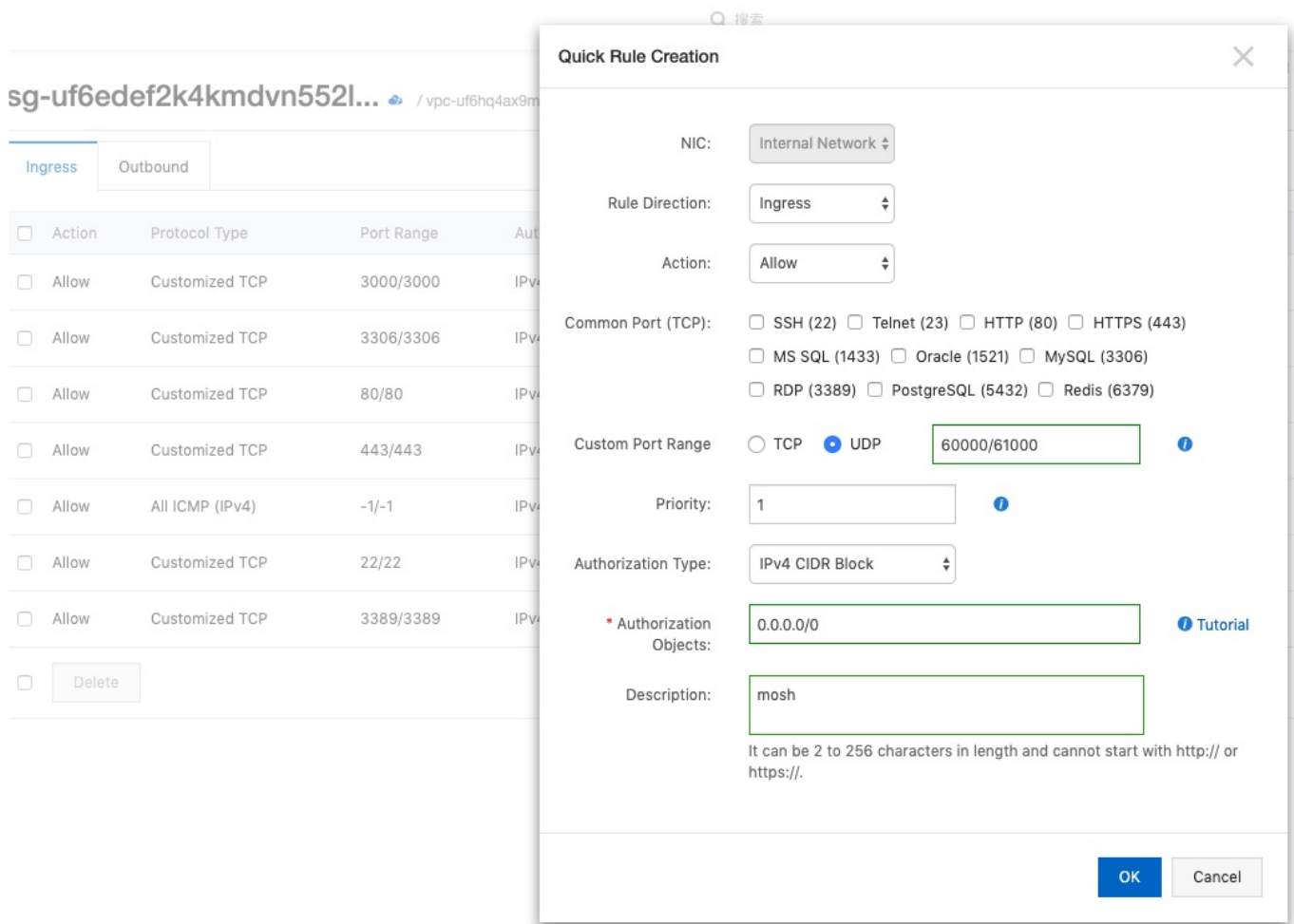
 复制代码

第二，打开服务器上的 UDP 端口 60000~61000。

```
1 junge@iZ4i3zrhuhpdbhZ:~$ sudo ufw allow 60000:61000/udp
```

 复制代码

第三，在阿里云的 Web 界面上修改主机的安全组设置，允许 UDP 端口 60000~61000。



第四，在客户端（比如 Mac 上），安装 Mosh Client。

```
1 jasonge@Juns-MacBook-Pro-2@l$ brew isntall mosh
```

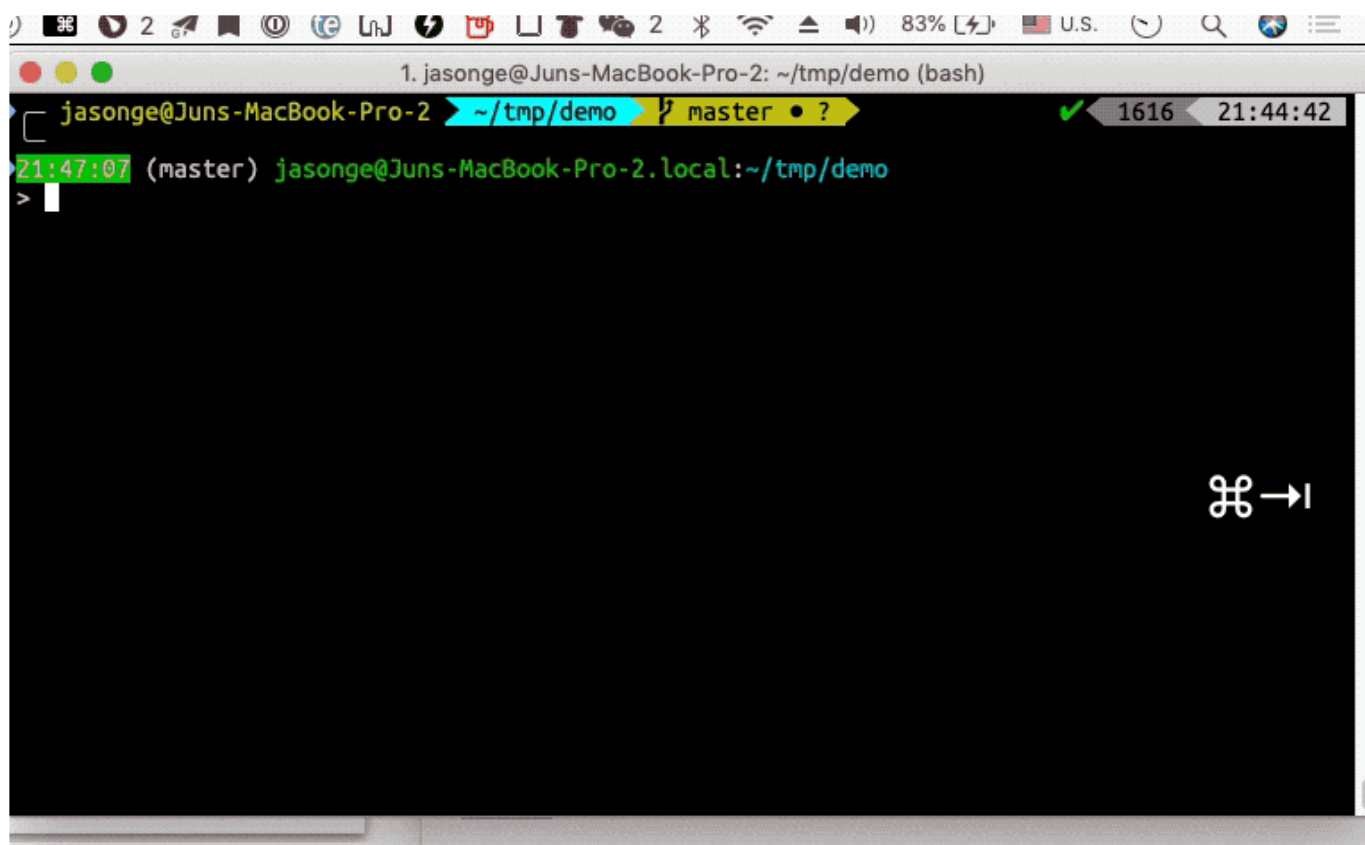
复制代码

第五，客户端使用 Mosh，用与 SSH 一样的命令行连接到服务器。

```
1 jasonge@Juns-MacBook-Pro-2@l$ mosh junge@<server-ip-or-name>
```

复制代码

下面这个录屏演示的是，我日常工作中使用 Mosh + Tmux 的流程。期间我会断开无线网，你可以看到 Mosh 自动连接上了，就好像来没有断过一样。



小结

今天，我与你分享的是使用命令行工具工作时，涉及的环境配置问题。

首先，我与你介绍了选择模拟终端、选择和配置 Shell 的重要准则，并结合案例给出了具体的工具和配置方法，其中涉及的工具包括 iTerm2、Bash、Zsh、Fish 等。然后，我结合远程 SSH 这一常见工作场景，给出了使用 Tmux 和 Mosh 的优化建议。

掌握了关于环境配置的这些内容以后，在下一篇文章中，我将与你介绍具体命令行工具的选择和使用。

其实，我推荐开发者多使用命令行工具，并不是因为它们看起来炫酷，而是它们确实可以帮助我们节省时间、提高个人的研发效能。而高效使用命令行工具的前提，是配置好环境。

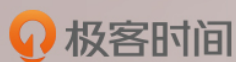
以 Mosh 为例，我最近经常会使用 iPad SSH 到远端服务器做一些开发工作。在这种移动开发的场景下，iPad 的网络经常断开，每次重新连接开销太大，基本上没办法工作。于是，我最终发现了 Mosh，并针对开发场景进行了设置。现在，每次我重新打开 iPad 的终端时，远程连接自动恢复，好像网络从没有断开过一样。这样一来，我就可以在移动端高效地开发了。

而对研发团队来说，如果能够对命令行工作环境进行优化和统一，毫无疑问会节省个人选择和配置工具的时间，进而提升团队的研究效能。

思考题

你觉得 Tmux 和 Screen 的最大区别是什么？是否有什么场景，我们必须使用其中的一个吗？

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎你把这篇文章分享给更多的朋友一起阅读。我们下期再见！



研发效率破局之道

Facebook 研发效率工作法

葛俊

前 Facebook 内部工具团队 Tech Lead



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 26 | Facebook怎样实现代码提交的原子性？

下一篇 28 | 从工作场景出发，寻找炫酷且有效的命令行工具

精选留言 (4)

 写留言



我来也

2019-10-26

这篇文章我也有些想说的.

我用这些工具可以做到的事情有:

- 1.每次即使是重启了系统(虽然mac不长重启),也能在有限的4步后恢复到工作状态. 之前的各种连接,打开的文件,执行的程序都还在,路径都不用切换,相对位置都一样....

展开 ∨

作者回复: 这个回复和@Johnson的回复都非常高质量。在进行讨论的时候该给出了很好的建议和工具。

关键字: Workspace as Code! !

tmuxinator 我还是第一次听说。看了一下，确实对保持Tmux Session做了进一步的工作。总的来说，你这样的配置可以说是把工作环境作为Code。Workspace as Code :) 对减少环境的设置很有效!! 推荐你写一篇博文把这个分享给大家。

另外几个链接也不错。推荐大家都看看 :)



💬 1

👍 2



Johnson

2019-10-26

tmux和screen之间的区别还真没怎么注意过，不过从开始用的时候感觉screen就比较老了，tmux更新好像更积极一些，可能区别在于tmux更强调强大的交互？session，window，pane 反正tmux给我的感觉就是分屏好用强大。

Terminal...

展开 ∨

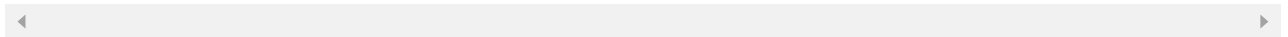
作者回复: 这个回复和“我来也”的回复都非常棒。下面是我的一点反馈：

1. 这个关于Tmux vs. Mosh的解释的确很清楚：“Well, first off, if you want the ability to attach to a session from multiple clients (or after the client dies), you should use screen or tmux. Mosh is a substitute (in some cases) for SSH, not for screen. Many Mosh users use it together with screen and like it that way.”

2. “我想让spacemacs在terminal中的光标的形状和颜色跟着模式变，这样非常棒，如果是iTerm2的话我就没法用xterm的控制字符序列来改变光标的形状和颜色，得用它自己的控制序列。所以我现在就直接用MacOS自带的terminal+MacForge(扩展管理)+MouseTerm plus(实现了xterm的OSC 12/112),仅供大家参考。”

这个推荐写一篇博文解释一下。我没有看太明白。

3. dotfiles 这种方法我有使用。我没有使用GitHub宫内宫开仓，用的是GitLab的一个私有仓中。效果还不错。后面可以考虑公开出来：)



💬 2

👍 2



hexinzhe

2019-10-25

请葛大展开讲讲ipad ssh 工作流

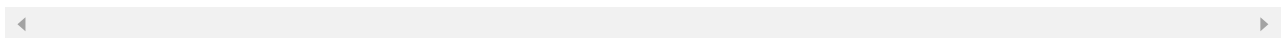
展开 ▼

作者回复: iPad上SSH，我使用的是Termius(<https://apps.apple.com/us/app/termius-ssh-client/id549039908>)。它支持Mosh。

这里我放了一张截图：

<https://sharemyfiles.s3-ap-southeast-1.amazonaws.com/iPad-termius.png>

我后面专门写一篇iPad上的工作流的文章。



💬

👍 1

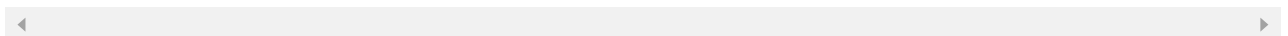


Jxin

2019-10-25

🔪手贱买的surface。压着重装黑苹果的想法，周末去研究下win的窗口先。

作者回复: 哈哈，研究回来给大家介绍一下：)



💬

👍