当您使用 shell 进行工作时，可以使用一些方法改善您的工作流，本节课我们就来讨论这些方法。

我们已经使用 shell 一段时间了，但是到目前为止我们的关注点主要集中在使用不同的命令上面。现在，我们将会学习如何同时执行多个不同的进程并追踪它们的状态、如何停止或暂停某个进程以及如何使进程在后台运行。

我们还将学习一些能够改善您的 shell 及其他工具的工作流的方法，这主要是通过定义别名或基于配置文件对其进行配置来实现的。这些方法都可以帮您节省大量的时间。例如，仅需要执行一些简单的命令，我们就可以在所有的主机上使用相同的配置。我们还会学习如何使用 SSH 操作远端机器。

任务控制

某些情况下我们需要中断正在执行的任务，比如当一个命令需要执行很长时间才能完成时（假设我们在使用 `find` 搜索一个非常大的目录结构）。大多数情况下，我们可以使用 `Ctrl-C` 来停止命令的执行。但是它的工作原理是什么呢？为什么有的时候会无法结束进程？

结束进程

您的 shell 会使用 UNIX 提供的信号机制执行进程间通信。当一个进程接收到信号时，它会停止执行、处理该信号并基于信号传递的信息来改变其执行。就这一点而言，信号是一种 *软件中断*。

在上面的例子中，当我们输入 `Ctrl-C` 时，shell 会发送一个 `SIGINT` 信号到进程。

下面这个 Python 程序向您展示了捕获信号 `SIGINT` 并忽略它的基本操作，它并不会让程序停止。为了停止这个程序，我们需要使用 `SIGQUIT` 信号，通过输入 `Ctrl-\` 可以发送该信号。

```
#!/usr/bin/env python
import signal, time

def handler(signum, time):
    print("\nI got a SIGINT, but I am not stopping")

signal.signal(signal.SIGINT, handler)
i = 0
while True:
    time.sleep(.1)
    print("\r{}".format(i), end="")
    i += 1
```

如果我们向这个程序发送两次 `SIGINT` ，然后再发送一次 `SIGQUIT`，程序会有什么反应？注意 `^` 是我们在终端输入 `Ctrl` 时的表示形式：

```
$ python sigint.py
24^C
I got a SIGINT, but I am not stopping
26^C
I got a SIGINT, but I am not stopping
30^\[1]    39913 quit       python sigint.pyf
```

尽管 `SIGINT` 和 `SIGQUIT` 都常常用来发出和终止程序相关的请求。`SIGTERM` 则是一个更加通用的、也更加优雅地退出信号。为了发出这个信号我们需要使用 `kill` 命令，它的语法是： `kill -TERM <PID>`。

暂停和后台执行进程

信号可以让进程做其他的事情，而不仅仅是终止它们。例如，`SIGSTOP` 会让进程暂停。在终端中，键入 `Ctrl-Z` 会让 shell 发送 `SIGTSTP` 信号，`SIGTSTP` 是 Terminal Stop 的缩写（即 `terminal` 版本的 SIGSTOP）。

我们可以使用 `fg` 或 `bg` 命令恢复暂停的工作。它们分别表示在前台继续或在后台继续。

`jobs` 命令会列出当前终端会话中尚未完成的全部任务。您可以使用 pid 引用这些任务（也可以用 `pgrep` 找出 pid）。更加符合直觉的操作是您可以使用百分号 + 任务编号（`jobs` 会打印任务编号）来选取该任务。如果要选择最近的一个任务，可以使用 `$!` 这一特殊参数。

还有一件事情需要掌握，那就是命令中的 `&` 后缀可以让命令在直接在后台运行，这使得您可以直接在 shell 中继续做其他操作，不过它此时还是会使用 shell 的标准输出，这一点有时会比较恼人（这种情况可以使用 shell 重定向处理）。

让已经在运行的进程转到后台运行，您可以键入 `Ctrl-Z` ，然后紧接着再输入 `bg`。注意，后台的进程仍然是您的终端进程的子进程，一旦您关闭终端（会发送另外一个信号 `SIGHUP`），这些后台的进程也会终止。为了防止这种情况发生，您可以使用 `nohup`（一个用来忽略 `SIGHUP` 的封装）来运行程序。针对已经运行的程序，可以使用 `disown` 。除此之外，您可以使用终端多路复用器来实现，下一章节我们会进行详细地探讨。

下面这个简单的会话中展示来了些概念的应用。

```
$ sleep 1000
^Z
[1]  + 18653 suspended  sleep 1000

$ nohup sleep 2000 &
[2] 18745
appending output to nohup.out

$ jobs
[1]  + suspended  sleep 1000
[2]  - running    nohup sleep 2000

$ bg %1
[1]  - 18653 continued  sleep 1000

$ jobs
[1]  - running    sleep 1000
[2]  + running    nohup sleep 2000

$ kill -STOP %1
[1]  + 18653 suspended (signal)  sleep 1000

$ jobs
[1]  + suspended (signal)  sleep 1000
[2]  - running    nohup sleep 2000

$ kill -SIGHUP %1
[1]  + 18653 hangup     sleep 1000

$ jobs
[2]  + running    nohup sleep 2000
```

```
$ kill -SIGHUP %2

$ jobs
[2]  + running    nohup sleep 2000

$ kill %2
[2]  + 18745 terminated  nohup sleep 2000

$ jobs
```

`SIGKILL` 是一个特殊的信号，它不能被进程捕获并且它会马上结束该进程。不过这样做会有一些副作用，例如留下孤儿进程。

您可以在 这里 或输入 `man signal` 或使用 `kill -l` 来获取更多关于信号的信息。

终端多路复用

当您在使用命令行时，您通常会希望同时执行多个任务。举例来说，您可以想要同时运行您的编辑器，并在终端的另外一侧执行程序。尽管再打开一个新的终端窗口也能达到目的，使用终端多路复用器则是一种更好的办法。

像 `tmux` 这类的终端多路复用器可以允许我们基于面板和标签分割出多个终端窗口，这样您便可以同时与多个 shell 会话进行交互。

不仅如此，终端多路复用使我们可以分离当前终端会话并在将来重新连接。

这让您操作远端设备时的工作流大大改善，避免了 `nohup` 和其他类似技巧的使用。

现在最流行的终端多路器是 `tmux`。`tmux` 是一个高度可定制的工具，您可以使用相关快捷键创建多个标签页并在它们间导航。

`tmux` 的快捷键需要我们掌握，它们都是类似 `<C-b> x` 这样的组合，即需要先按下 `Ctrl+b`，松开后再按下 `x`。`tmux` 中对象的继承结构如下：

- **会话** - 每个会话都是一个独立的工作区，其中包含一个或多个窗口
  - `tmux` 开始一个新的会话
  - `tmux new -s NAME` 以指定名称开始一个新的会话
  - `tmux ls` 列出当前所有会话
  - 在 `tmux` 中输入 `<C-b> d` ，将当前会话分离
  - `tmux a` 重新连接最后一个会话。您也可以通过 `-t` 来指定具体的会话
- **窗口** - 相当于编辑器或是浏览器中的标签页，从视觉上将一个会话分割为多个部分
  - `<C-b> c` 创建一个新的窗口，使用 `<C-d>` 关闭
  - `<C-b> N` 跳转到第 *N* 个窗口，注意每个窗口都是有编号的
  - `<C-b> p` 切换到前一个窗口
  - `<C-b> n` 切换到下一个窗口
  - `<C-b> ,` 重命名当前窗口
  - `<C-b> w` 列出当前所有窗口

- **面板** - 像 vim 中的分屏一样，面板使我们可以在一个屏幕里显示多个 shell
  - `<C-b> "` 水平分割
  - `<C-b> %` 垂直分割
  - `<C-b> <方向>` 切换到指定方向的面板，<方向> 指的是键盘上的方向键
  - `<C-b> z` 切换当前面板的缩放
  - `<C-b> [` 开始往回卷动屏幕。您可以按下空格键来开始选择，回车键复制选中的部分
  - `<C-b> <空格>` 在不同的面板排布间切换

扩展阅读：
这里 是一份 `tmux` 快速入门教程， 而这一篇 文章则更加详细，它包含了 `screen` 命令。您也许想要掌握 `screen` 命令，因为在大多数 UNIX 系统中都默认安装有该程序。

别名

输入一长串包含许多选项的命令会非常麻烦。因此，大多数 shell 都支持设置别名。shell 的别名相当于一个长命令的缩写，shell 会自动将其替换成原本的命令。例如，bash 中的别名语法如下：

```
alias alias_name="command_to_alias arg1 arg2"
```

注意， `=` 两边是没有空格的，因为 `alias` 是一个 shell 命令，它只接受一个参数。

别名有许多很方便的特性:

```
# 创建常用命令的缩写
alias ll="ls -lh"

# 能够少输入很多
alias gs="git status"
alias gc="git commit"
alias v="vim"

# 手误打错命令也没关系
alias sl=ls

# 重新定义一些命令行的默认行为
alias mv="mv -i"           # -i prompts before overwrite
alias mkdir="mkdir -p"     # -p make parent dirs as needed
alias df="df -h"           # -h prints human readable format

# 别名可以组合使用
alias la="ls -A"
alias lla="la -l"

# 在忽略某个别名
\ls
# 或者禁用别名
unalias la

# 获取别名的定义
```

```
alias ll
# 会打印 ll='ls -lh'
```

值得注意的是，在默认情况下 shell 并不会保存别名。为了让别名持续生效，您需要将配置放进 shell 的启动文件里，像是 `.bashrc` 或 `.zshrc`，下一节我们就会讲到。

配置文件（Dotfiles）

很多程序的配置都是通过纯文本格式的被称作 *点文件* 的配置文件来完成的（之所以称为点文件，是因为它们的文件名以 `.` 开头，例如 `~/.vimrc`。也正因为此，它们默认是隐藏文件，`ls` 并不会显示它们）。

shell 的配置也是通过这类文件完成的。在启动时，您的 shell 程序会读取很多文件以加载其配置项。根据 shell 本身的不同，您从登录开始还是以交互的方式完成这一过程可能会有很大的不同。关于这一话题，这里 有非常好的资源

对于 `bash` 来说，在大多数系统下，您可以通过编辑 `.bashrc` 或 `.bash_profile` 来进行配置。在文件中您可以添加需要在启动时执行的命令，例如上文我们讲到过的别名，或者是您的环境变量。

实际上，很多程序都要求您在 shell 的配置文件中包含一行类似 `export PATH="$PATH:/path/to/program/bin"` 的命令，这样才能确保这些程序能够被 shell 找到。

还有一些其他的工具也可以通过 *点文件* 进行配置：

- `bash` - `~/.bashrc`, `~/.bash_profile`
- `git` - `~/.gitconfig`
- `vim` - `~/.vimrc` 和 `~/.vim` 目录
- `ssh` - `~/.ssh/config`
- `tmux` - `~/.tmux.conf`

我们应该如何管理这些配置文件呢，它们应该在它们的文件夹下，并使用版本控制系统进行管理，然后通过脚本将其 **符号链接** 到需要的地方。这么做有如下好处：

- **安装简单**: 如果您登录了一台新的设备，在这台设备上应用您的配置只需要几分钟的时间；
- **可移植性**: 您的工具在任何地方都以相同的配置工作
- **同步**: 在一处更新配置文件，可以同步到其他所有地方
- **变更追踪**: 您可能要在整个程序员生涯中持续维护这些配置文件，而对于长期项目而言，版本历史是非常重要的

配置文件中需要放些什么？您可以通过在线文档和 帮助手册 了解所使用工具的设置项。另一个方法是在网上搜索有关特定程序的文章，作者们在文章中会分享他们的配置。还有一种方法就是直接浏览其他人的配置文件：您可以在这里找到无数的 dotfiles 仓库 —— 其中最受欢迎的那些可以在 这里 找到（我们建议您不要直接复制别人的配置）。这里 也有一些非常有用的资源。

本课程的老师们也在 GitHub 上开源了他们的配置文件：
Anish,
Jon,
Jose.

可移植性

配置文件的一个常见的痛点是它可能并不能在多种设备上生效。例如，如果您在不同设备上使用的操作系统或者 shell 是不同的，则配置文件是无法生效的。或者，有时您仅希望特定的配置只在某些设备上生效。

有一些技巧可以轻松达成这些目的。如果配置文件 if 语句，则您可以借助它针对不同的设备编写不同的配置。例如，您的 shell 可以这样做：

```
if [[ "$(uname)" == "Linux" ]]; then {do_something}; fi

# 使用和 shell 相关的配置时先检查当前 shell 类型
if [[ "$SHELL" == "zsh" ]]; then {do_something}; fi

# 您也可以针对特定的设备进行配置
if [[ "$(hostname)" == "myServer" ]]; then {do_something}; fi
```

如果配置文件支持 include 功能，您也可以多加利用。例如：`~/.gitconfig` 可以这样编写：

```
[include]
    path = ~/.gitconfig_local
```

然后我们可以在日常使用的设备上创建配置文件 `~/.gitconfig_local` 来包含与该设备相关的特定配置。您甚至应该创建一个单独的代码仓库来管理这些与设备相关的配置。

如果您希望在不同的程序之间共享某些配置，该方法也适用。例如，如果您想要在 `bash` 和 `zsh` 中同时启用一些别名，您可以把它们写在 `.aliases` 里，然后在这两个 shell 里应用：

```
# Test if ~/.aliases exists and source it
if [ -f ~/.aliases ]; then
    source ~/.aliases
fi
```

远端设备

对于程序员来说，在他们的日常工作中使用远程服务器已经非常普遍了。如果您需要使用远程服务器来部署后端软件或您需要一些计算能力强大的服务器，您就会用到安全 shell（SSH）。和其他工具一样，SSH 也是可以高度定制的，也值得我们花时间学习它。

通过如下命令，您可以使用 `ssh` 连接到其他服务器：

```
ssh foo@bar.mit.edu
```

这里我们尝试以用户名 `foo` 登录服务器 `bar.mit.edu`。服务器可以通过 URL 指定（例如 `bar.mit.edu`），也可以使用 IP 指定（例如 `foobar@192.168.1.42`）。后面我们会介绍如何修改 ssh 配置文件使我们可以用类似 `ssh bar` 这样的命令来登录服务器。

执行命令

`ssh` 的一个经常被忽视的特性是它可以直接远程执行命令。
`ssh foobar@server ls` 可以直接在用 foobar 的命令下执行 `ls` 命令。
想要配合管道来使用也可以，`ssh foobar@server ls | grep PATTERN` 会在本地查询远端 `ls` 的输出而 `ls | ssh foobar@server grep PATTERN` 会在远端对本地 `ls` 输出的结果进行查询。

SSH 密钥

基于密钥的验证机制使用了密码学中的公钥，我们只需要向服务器证明客户端持有对应的私钥，而不需要公开其私钥。这样您就可以避免每次登录都输入密码的麻烦了秘密就可以登录。不过，私钥(通常是 `~/.ssh/id_rsa` 或者 `~/.ssh/id_ed25519`) 等效于您的密码，所以一定要好好保存它。

密钥生成

使用 `ssh-keygen` 命令可以生成一对密钥：

```
ssh-keygen -o -a 100 -t ed25519 -f ~/.ssh/id_ed25519
```

您可以为密钥设置密码，防止有人持有您的私钥并使用它访问您的服务器。您可以使用 `ssh-agent` 或 `gpg-agent`，这样就不需要每次都输入该密码了。

如果您曾经配置过使用 SSH 密钥推送到 GitHub，那么可能您已经完成了 这里 介绍的这些步骤，并且已经有了一个可用的密钥对。要检查您是否持有密码并验证它，您可以运行 `ssh-keygen -y -f /path/to/key`.

基于密钥的认证机制

`ssh` 会查询 `.ssh/authorized_keys` 来确认那些用户可以被允许登录。您可以通过下面的命令将一个公钥拷贝到这里：

```
cat .ssh/id_ed25519.pub | ssh foobar@remote 'cat >> ~/.ssh/authorized_keys'
```

如果支持 `ssh-copy-id` 的话，可以使用下面这种更简单的解决方案：

```
ssh-copy-id -i .ssh/id_ed25519.pub foobar@remote
```

通过 SSH 复制文件

使用 ssh 复制文件有很多方法：

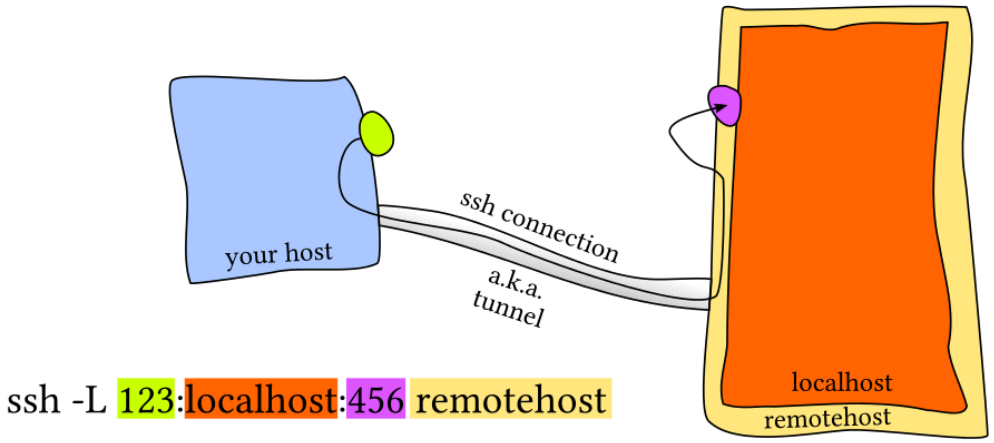
- `ssh+tee`, 最简单的方法是执行 `ssh` 命令，然后通过这样的方法利用标准输入实现 `cat localfile | ssh remote_server tee serverfile`。回忆一下，`tee` 命令会将标准输出写入到一个文件；
- `scp` ：当需要拷贝大量的文件或目录时，使用 `scp` 命令则更加方便，因为它可以方便的遍历相关路径。语法如下：`scp path/to/local_file remote_host:path/to/remote_file`；
- `rsync` 对 `scp` 进行了改进，它可以检测本地和远端的文件以防止重复拷贝。它还可以提供一些诸如符号连接、权限管理等精心打磨的功能。甚至还可以基于 `--partial` 标记实现断点续传。`rsync` 的语法和 `scp` 类似；

端口转发

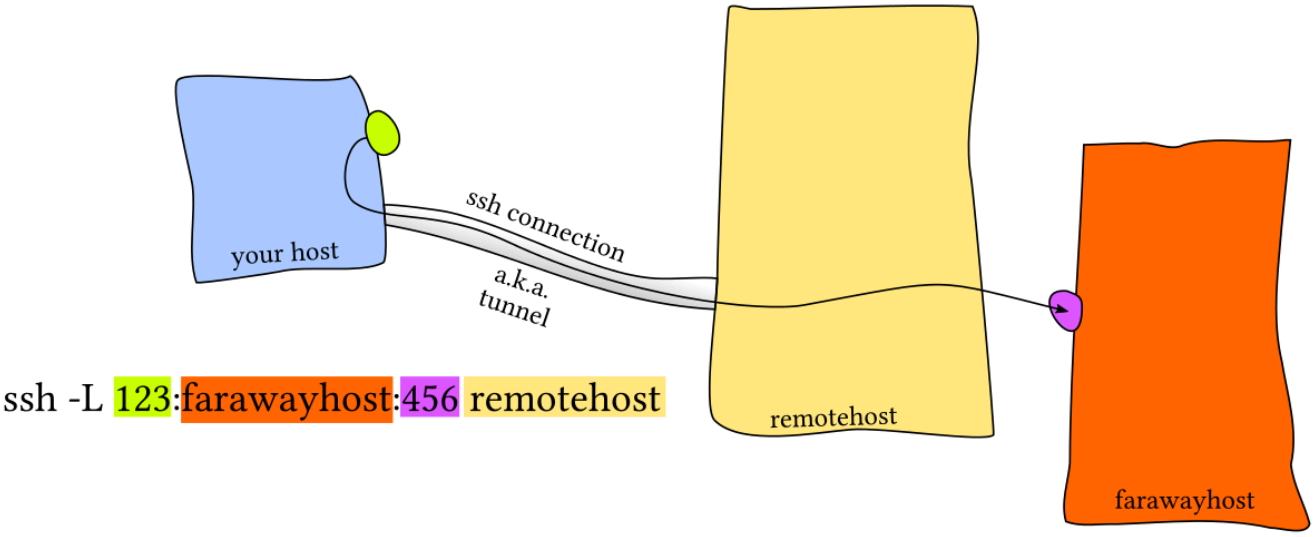
很多情况下我们都会遇到软件需要监听特定设备的端口。如果是在您的本机，可以使用 `localhost:PORT` 或 `127.0.0.1:PORT`。但是如果需要监听远程服务器的端口该如何操作呢？这种情况下远端的端口并不会直接通过网络暴露给您。

此时就需要进行 *端口转发*。端口转发有两种，一种是本地端口转发和远程端口转发（参见下图，该图片引用自这篇 StackOverflow 文章）中的图片。

本地端口转发

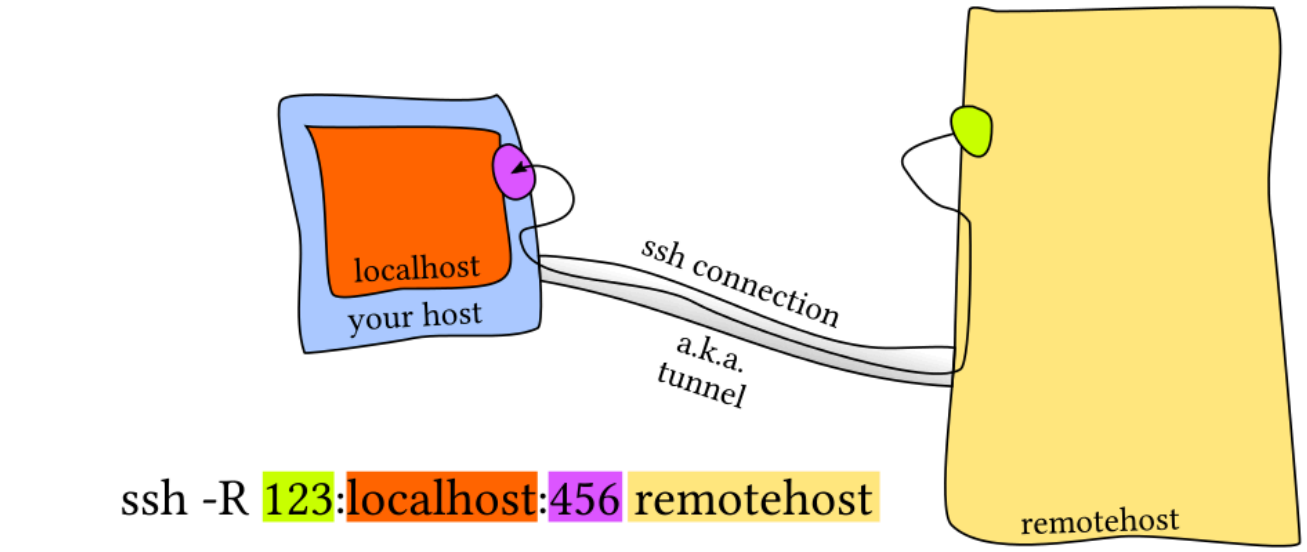


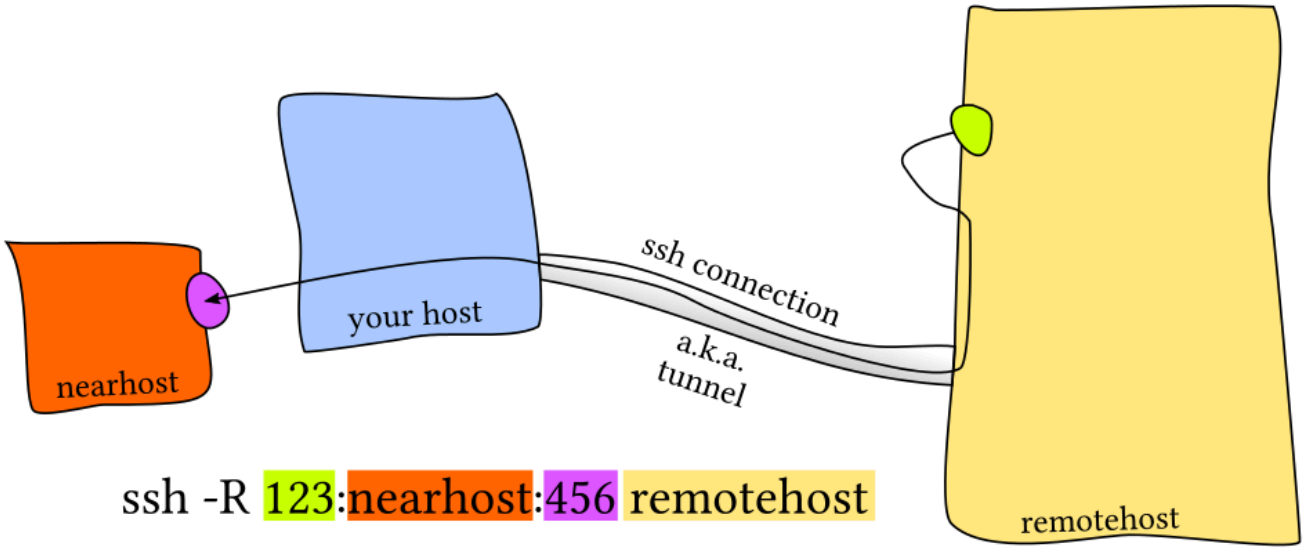
ssh -L 123:localhost:456 remotehost



ssh -L 123:farawayhost:456 remotehost

远程端口转发





常见的情景是使用本地端口转发，即远端设备上的服务监听一个端口，而您希望在本地设备上的一个端口建立连接并转发到远程端口上。例如，我们在远端服务器上运行 Jupyter notebook 并监听 `8888` 端口。 然后，建立从本地端口 `9999` 的转发，使用 `ssh -L 9999:localhost:8888 foobar@remote_server` 。这样只需要访问本地的 `localhost:9999` 即可。

SSH 配置

我们已经介绍了很多参数。为它们创建一个别名是个好想法，我们可以这样做：

```
alias my_server="ssh -i ~/.id_ed25519 --port 2222 -L 9999:localhost:8888
foobar@remote_server
```

不过，更好的方法是使用 `~/.ssh/config`.

```
Host vm
    User foobar
    HostName 172.16.174.141
    Port 2222
    IdentityFile ~/.ssh/id_ed25519
    LocalForward 9999 localhost:8888

# 在配置文件中也可以使用通配符
Host *.mit.edu
    User foobaz
```

这么做的好处是，使用 `~/.ssh/config` 文件来创建别名，类似 `scp`、`rsync` 和 `mosh` 的这些命令都可以读取这个配置并将设置转换为对应的命令行选项。

注意，`~/.ssh/config` 文件也可以被当作配置文件，而且一般情况下也是可以被导入其他配置文件的。不过，如果您将其公开到互联网上，那么其他人都将会看到您的服务器地址、用户名、开放端口等等。这些信息可能会帮助到那些企图攻击您系统的黑客，所以请务必三思。

服务器侧的配置通常放在 `/etc/ssh/sshd_config`。您可以在这里配置免密认证、修改 ssh 端口、开启 X11 转发等等。 您也可以为每个用户单独指定配置。

杂项

连接远程服务器的一个常见痛点是遇到由关机、休眠或网络环境变化导致的掉线。如果连接的延迟很高也很让人讨厌。Mosh（即 mobile shell ）对 ssh 进行了改进，它允许连接漫游、间歇连接及智能本地回显。

有时将一个远端文件夹挂载到本地会比较方便， sshfs 可以将远端服务器上的一个文件夹挂载到本地，然后您就可以使用本地的编辑器了。

Shell & 框架

在 shell 工具和脚本那节课中我们已经介绍了 `bash` shell，因为它是目前最通用的 shell，大多数的系统都将其作为默认 shell。但是，它并不是唯一的选项。

例如，`zsh` shell 是 `bash` 的超集并提供了一些方便的功能：

- 智能替换, `**`
- 行内替换/通配符扩展
- 拼写纠错
- 更好的 tab 补全和选择
- 路径展开 (`cd /u/lo/b` 会被展开为 `/usr/local/bin`)

框架 也可以改进您的 shell。比较流行的通用框架包括 prezto 或 oh-my-zsh。还有一些更精简的框架，它们往往专注于某一个特定功能，例如 zsh 语法高亮 或 zsh 历史子串查询。 像 fish 这样的 shell 包含了很多用户友好的功能，其中一些特性包括：

- 向右对齐
- 命令语法高亮
- 历史子串查询
- 基于手册页面的选项补全

- 更智能的自动补全
- 提示符主题

需要注意的是，使用这些框架可能会降低您 shell 的性能，尤其是如果这些框架的代码没有优化或者代码过多。您随时可以测试其性能或禁用某些不常用的功能来实现速度与功能的平衡。

终端模拟器

和自定义 shell 一样，花点时间选择适合您的 **终端模拟器** 并进行设置是很有必要的。有许多终端模拟器可供您选择（这里有一些关于它们之间 比较 的信息）

您会花上很多时间在使用终端上，因此研究一下终端的设置是很有必要的，您可以从下面这些方面来配置您的终端：

- 字体选择
- 彩色主题
- 快捷键
- 标签页/面板支持
- 回退配置
- 性能（像 Alacritty 或者 kitty 这种比较新的终端，它们支持 GPU 加速）。

课后练习

习题解答

任务控制

1. 我们可以使用类似 `ps aux | grep` 这样的命令来获取任务的 pid ，然后您可以基于 pid 来结束这些进程。但我们其实有更好的方法来做这件事。在终端中执行 `sleep 10000` 这个任务。然后用 `Ctrl-Z` 将其切换到后台并使用 `bg` 来继续允许它。现在，使用 `pgrep` 来查找 pid 并使用 `pkill` 结束进程而不需要手动输入 pid。(提示：：使用 `-af` 标记)。
2. 如果您希望某个进程结束后再开始另外一个进程， 应该如何实现呢？在这个练习中，我们使用 `sleep 60 &` 作为先执行的程序。一种方法是使用 `wait` 命令。尝试启动这个休眠命令，然后待其结束后再执行 `ls` 命令。

   但是，如果我们在不同的 bash 会话中进行操作，则上述方法就不起作用了。因为 `wait` 只能对子进程起作用。之前我们没有提过的一个特性是， `kill` 命令成功退出时其状态码为 0 ，其他状态则是非 0。 `kill -0` 则不会发送信号，但是会在进程不存在时返回一个不为 0 的状态码。请编写一个 bash 函数 `pidwait` ，它接受一个 pid 作为输入参数，然后一直等待直到该进程结束。您需要使用 `sleep` 来避免浪费 CPU 性能。

终端多路复用

1. 请完成这个 `tmux` 教程 参考 这些步骤 来学习如何自定义 `tmux`。

别名

1. 创建一个 `dc` 别名，它的功能是当我们错误的将 `cd` 输入为 `dc` 时也能正确执行。
2. 执行 `history | awk '{$1="";print substr($0,2)}' | sort | uniq -c | sort -n | tail -n 10` 来获取您最常用的十条命令，尝试为它们创建别名。注意：这个命令只在 Bash 中生效，如果您使用 ZSH，使用 `history 1` 替换 `history`。

配置文件

让我们帮助您进一步学习配置文件：

1. 为您的配置文件新建一个文件夹，并设置好版本控制
2. 在其中添加至少一个配置文件，比如说您的 shell，在其中包含一些自定义设置（可以从设置 `$PS1` 开始）。
3. 建立一种在新设备进行快速安装配置的方法（无需手动操作）。最简单的方法是写一个 shell 脚本对每个文件使用 `ln -s`，也可以使用 专用工具
4. 在新的虚拟机上测试该安装脚本。
5. 将您现有的所有配置文件移动到项目仓库里。
6. 将项目发布到 GitHub。

远端设备

进行下面的练习需要您先安装一个 Linux 虚拟机（如果已经安装过则可以直接使用），如果您对虚拟机尚不熟悉，可以参考 这篇教程 来进行安装。

1. 前往 `~/.ssh/` 并查看是否已经存在 SSH 密钥对。如果不存在，请使用 `ssh-keygen -o -a 100 -t ed25519` 来创建一个。建议为密钥设置密码然后使用 `ssh-agent`，更多信息可以参考 这里；
2. 在 `.ssh/config` 加入下面内容：

```
Host vm
    User username_goes_here
    HostName ip_goes_here
    IdentityFile ~/.ssh/id_ed25519
    LocalForward 9999 localhost:8888
```

3. 使用 `ssh-copy-id vm` 将您的 ssh 密钥拷贝到服务器。
4. 使用 `python -m http.server 8888` 在您的虚拟机中启动一个 Web 服务器并通过本机的 `http://localhost:9999` 访问虚拟机上的 Web 服务器
5. 使用 `sudo vim /etc/ssh/sshd_config` 编辑 SSH 服务器配置，通过修改 `PasswordAuthentication` 的值来禁用密码验证。通过修改 `PermitRootLogin` 的值来禁用 root 登录。然后使用 `sudo service sshd restart` 重启 `ssh` 服务器，然后重新尝试。
6. (附加题) 在虚拟机中安装 `mosh` 并启动连接。然后断开服务器/虚拟机的网络适配器。mosh 可以恢复连接吗？
7. (附加题) 查看 `ssh` 的 `-N` 和 `-f` 选项的作用，找出在后台进行端口转发的命令是什么？