0 写在开头

> 本文内容和所使用到的素材分享在 simpleRpackage: About how to develop a simple R package (github.com) 上。
>
> 本文案例提到的 R 包分享在 wangchengLi6/randomwalkli (github.com) 上。使用如下命令即可：`devtools::install_github("wangchengLi6/randomwalkli")`

如果预览过 CRAN 提供的 R 包开发官方文档，或者 Hadley 写作的 R 包开发教程，就不难意识到，R 包的开发是一个非常非常复杂的软件工程问题。作为一名刚入门的研究生，我对于该领域一知半解。因此这篇文章更接近于学习笔记，而非详尽的开发教程。我希望这篇文章能对以下两类读者产生相应帮助：

- 为仅需要制作一个 R 包并上传 GitHub 平台，来分享交流自身学术成果的读者，提供一个完整的中文教程。
- 为刚刚接触 R 包开发，但需要深入学习的读者，提供一个完整的思维框架。事实上，我认为进阶的内容大多是在该框架上的缝缝补补。

在开始之前，我们需要思考一个问题，即为什么需要做一个 R 包。

制作 R 包可以有很多种理由，但我的出发点是希望自己可能的某些学术成果能够更便捷地被其他人所使用，以实现推广和交流的目的。这意味着 R 包和单纯地公开代码并不一样，我们需要更多地从用户角度考虑。因此，制作一个 R 包至少要解决两个问题：

1. 将自己实现的功能打包在 R 包里。
2. 使得用户能够使用 R 包（包括知道怎么用）。

此外，考虑到 R 包的分发往往依赖于某些网络平台，因此我们还需要了解如何将开发好的 R 包，上传至某个平台（如常用的GitHub），以便于用户下载使用。

这分别对应了这篇文章的三个部分，即“创建一个可以被使用的 R 包”，“为 R 包的功能添加说明文档”，“将 R 包上传至 GitHub”。

参考材料

本文第一节和第二节的参考内容如下。需要说明的是，在此之前国内已经有一位用户在网络上上传过一份中文的 R 包开发极简教程以及相应的 b 站视频。因为我是在该教程的帮助下入门，所以全文的整体结构将类似于该教程。但进行了一些补充，并加入了个人理解。

1 国内某用户提供的中文版，R包开发教程，以及 B 站视频：RPubs - Document; https://www.bilibili.com/video/BV11U4y1u7aF/;

2 CRAN提供的官方文档：Writing R Extensions (r-project.org)

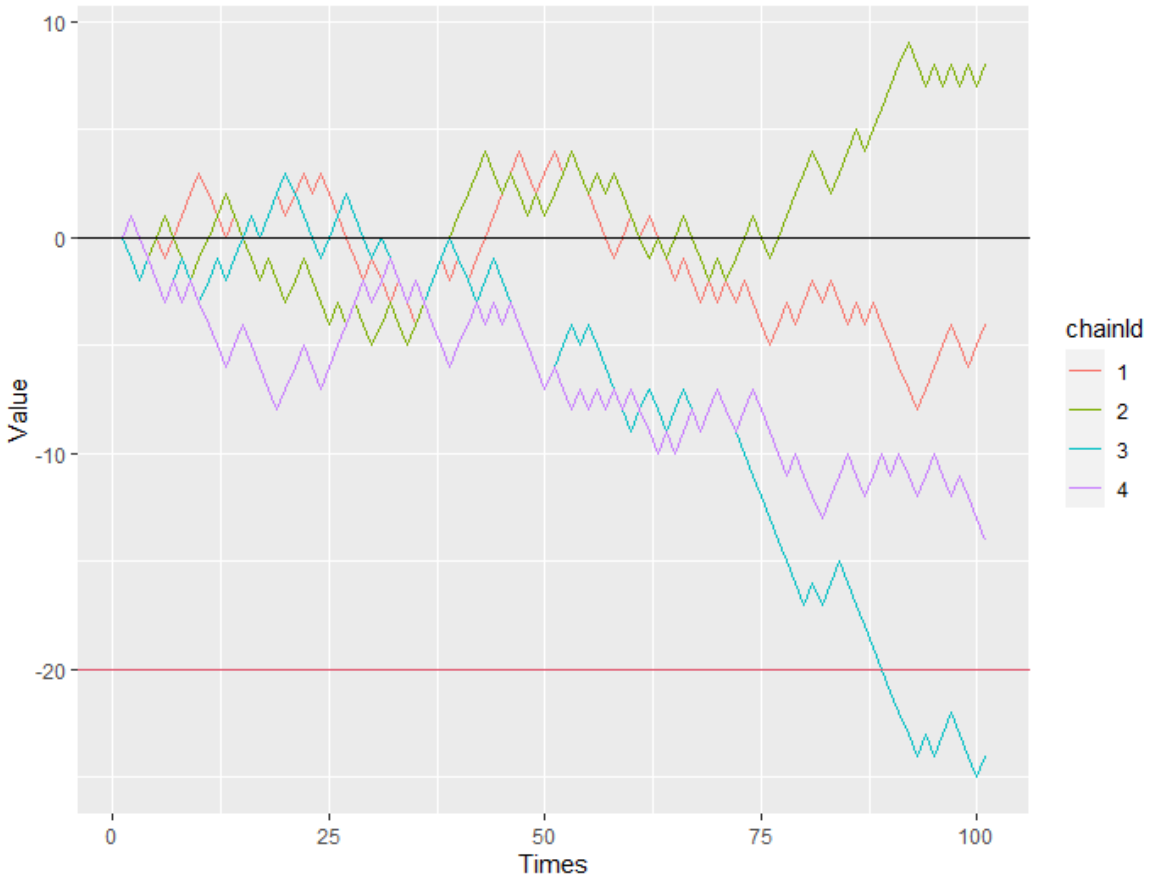
3 Hadley的一本R包开发教程，电子版: R Packages (2e) (r-pkgs.org)

1 创建一个简单的R包

H2 1.0 素材介绍

本案例实现的功能是绘制 `L` 条 长度为 `n`，单侧概率为 `p` 的随机游走链，此外有参数 `seed` 用于给定随机数种子。如下：

`main(L = 100, p = 0.5, n = 4, seed = 1)` ，



该 R 包已经上传至 GitHub，使用命令 `devtools::install_github("wangchengLi6/randomwalkli")` 即可安装。创建该 R 包所需的素材包括三个函数，分别是 `gene.chain` （用于生成绘制随机游走链的数据），`plot.chain` （使用 `gene.chain` 函数的数据，使用 `ggplot2` 包来生成一个 ggplot 对象）。 `main` 调用前两个函数，并最终绘制出随机游走链。

素材也已经上传至 GitHub simpleRpackage: About how to develop a simple R package (github.com) 中的 `code.R` 文件。

H2 1.1 准备工作

开发一个 R 包需要使用到四个 R 包：

```
install.packages(c("devtools","roxygen2","testthat","knitr"))
```

四个包的大概功能：

- devtools 和 usethis：用于构建 R 包的各种文件。
- roxygen2 和 knitr：用于生成 R 包所需要的各类文档。
- testthat：包含了测试 R 包所需要的方法。

> 一般 Rstudio 默认安装了这些包。此外 devtools 包还依赖于 "usethis"，因此理论上也会默认安装 usethis 包。

H2 1.2 创建一个项目（Project）

使用 Rstudio 菜单，我们可以非常便捷地创建一个 R 包项目：

File >> New project >> New Directory >> package

随后根据提示，输入你的包名（在本案例中，我们的包名是"randomwalkli"）

创建项目之后，会自动生成一系列的文件，其中我们会涉及到的是以下四个文件/文件夹

- 文件夹 `./R` ：包文件的核心，所有的R代码都会放在里面。
- 文件夹 `./man` ：之后所有的说明文档都会被储存在该文件夹中
- 文件 `DESCRIPTION` ：顾名思义，是 R 包的描述文件，将提供包的metadata。
- 文件 `NAMESPACE` ：声明需要从包中导出的函数（并非所有函数都需要导出），以及需要其它包中导入的函数。这个文件我们一般都不会自己编辑。

剩下的文件都不需要我们进行修改。

> 关于工作路径：
>
> 为了使 R 程序知道你正在修改的是哪一个项目，我们必须设定工作路径。
>
> 第一次使用 Rstudio 创建项目时， Rstudio 会自动将工作路径设定在项目所在的文件夹。而在之后再一次打开 Rstudio 的时候，工作路径可能已经改变。错误的工作路径将导致错误。因此在下一次打开 Rstudio 时，应当从 Rstudio 的顶部菜单栏选择 File >> Open project ，并选择你需要修改的项目文件夹下的 `<package>.Rproj` 文件。
>
> 或者也可以使用代码形式：使用 `getwd()` 确认当前工作目录，并使用 `setwd()` 修改工作目录到项目所在文件夹。

> 关于命令方法创建 R 包项目：
>
> 使用 Rstudio 创建 R 包项目，与在 Console 中输入命令 `devtools::create(<project path>)` 是一样的。之所以提到这件事情，并不是建议大家使用 函数命令 来创建 R 包，而是希望大家对 `devtools` 包的功能有所理解。

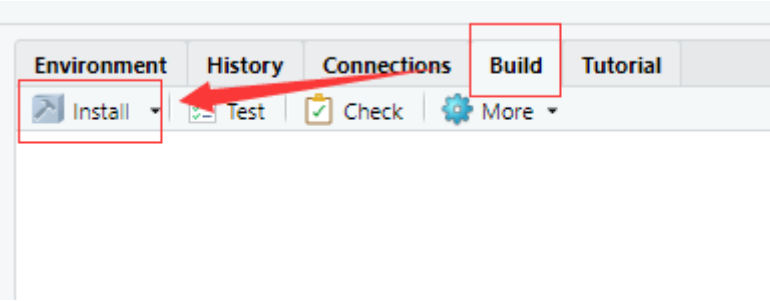
H2 1.3 创建所需要的函数

> R 包的函数（实际上是对象 objects）是一个包的基础，也是一个包的灵魂。

1.3.1 示例

创建函数是我们工作的核心。打开 R 文件夹，会发现里面已经有一个 `hello.R` 文件。这是创建 R 包时自动生成的示例文件。我们可以试着直接安装这个包：

方法一：图形化，Rstudio 界面右栏的两个窗口中，有一个小窗口选项是 `Build` ，单击进入后，点击 `Install` 按钮即可。



方法二：代码化，Console 中运行代码 `devtools::install(<projectpath>)` 。该函数可以编译安装由 `devtools` 包创建的 R 包项目。提到这件事情是因为，某些命令行系统下（如 linux服务器），当无法使用 Rstudio 时，可以使用该命令来安装储存在本地的 R 包。

> 关于 `devtools::install()` 的参数：
>
> 该函数如果未给定 `<projectpath>` 参数，则默认编译安装当前工作路径下的项目。这也是为什么前面强调了 工作路径 问题。更多参数请查看说明文档 `?devtools::install` 。

```
> devtools::install()
These packages have more recent versions available.
It is recommended to update all of them.
Which would you like to update?

 1: All
 2: CRAN packages only
 3: None
 4: cli         (3.3.0 -> 3.4.1) [CRAN]
 5: vctrs       (0.4.1 -> 0.5.1) [CRAN]
 6: viridisLite (0.4.0 -> 0.4.1) [CRAN]
 7: scales      (1.2.0 -> 1.2.1) [CRAN]
 8: isoband     (0.2.5 -> 0.2.6) [CRAN]
 9: gtable      (0.3.0 -> 0.3.1) [CRAN]
10: ggplot2     (3.3.6 -> 3.4.0) [CRAN]

Enter one or more numbers, or an empty line to skip updates:
✓  checking for file 'D:\OneDrive\WorkingTable\ProjectOriented\2022\20221119 MeetReport\rpack\pack\ran
omwalkli/DESCRIPTION' ...
─  preparing 'randomwalkli':
✓  checking DESCRIPTION meta-information ...
─  checking for LF line-endings in source and make files and shell scripts
─  checking for empty or unneeded directories
   Omitted 'LazyData' from DESCRIPTION
─  building 'randomwalkli_0.1.0.tar.gz'

Running "C:/PROGRA~1/R/R-42~1.0/bin/x64/Rcmd.exe" INSTALL \
  "C:\Users\22907\AppData\Local\Temp\RtmpGWyEgC/randomwalkli_0.1.0.tar.gz" --install-tests
* installing to library 'C:/Users/22907/AppData/Local/R/win-library/4.2'
* installing *source* package 'randomwalkli' ...
** using staged installation
** R
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (randomwalkli)
>
```

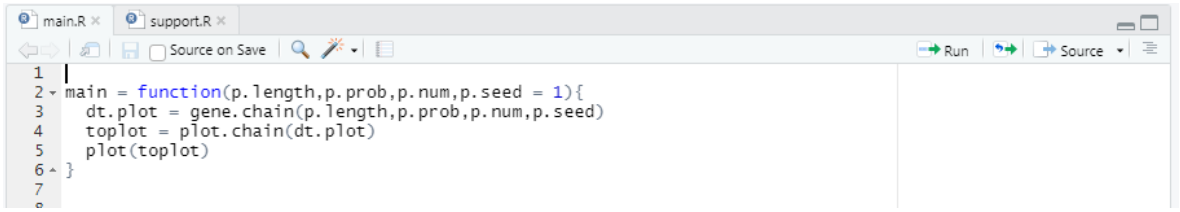
安装好 R 包后，我们需要重启 R Console 并 `library` 该包。其中 Rstudio 界面的 `Install` 按钮将自动完成这两步。而代码化方法则需要手动重启 R console （可以在顶部菜单栏 >> Session >> Restart R 完成），并手动导入这个包： `library("randomwalkli")` 。

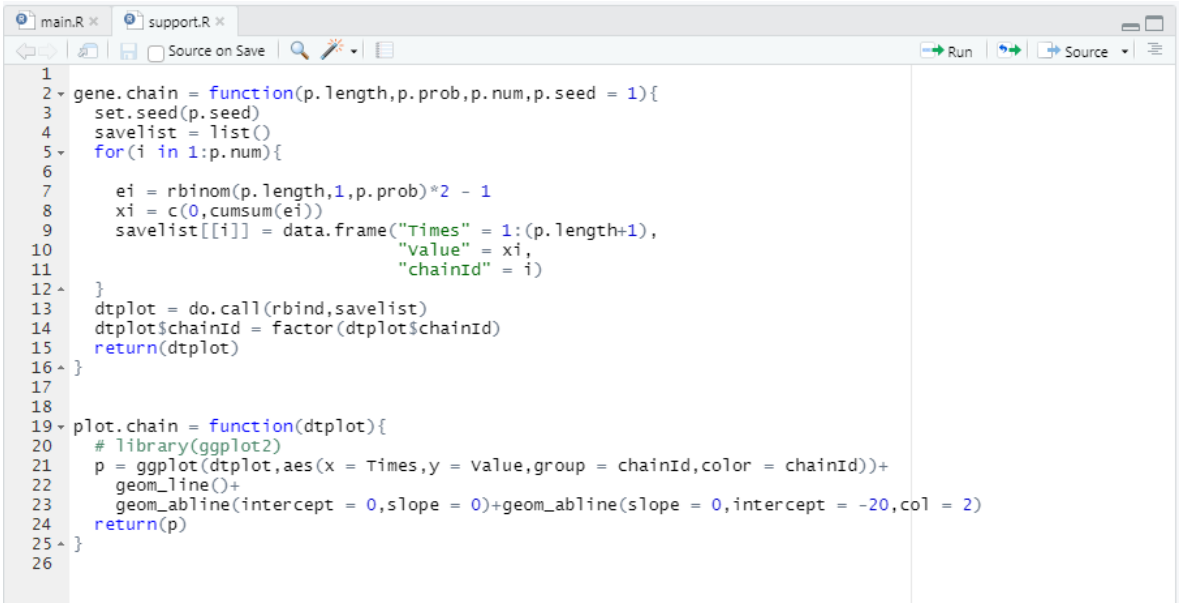
导入后，可以直接使用 `hello()` 。该函数没有返回值，而是直接打印字符串 `"hello world!"` 。

H3 1.3.2 自己的包函数

有了 `hello.R` 的参考，那么创建我们自己的 R 包也就非常简单。

删除原有的 `hello.R` 文件，并在 R 文件夹中，创建两个 R 文件用于存放我们自己的函数：





- 为什么是两个：这主要是一个习惯。
    - 建议不要将所有函数放在一起，最好是根据功能或者某种属性分开来。当然，函数少的时候没有太大影响，多的时候则便于管理。
- 为什么是函数：
    - `library` 一个R 包和 `sourse` 一个 `.R` 文件是不一样的：`sourse` 一个 R 文件，文件内每一行代码都将被执行。这与 Ctrl A 并 run 是一个效果。
    - 但包里面的 R 文件则只会在编译时（也就是你安装的时候）被执行一次，然后保存此时生成的所有对象。在 `library` R 包时，只会导入所有的对象。

> 关于对象和过程：
>
> 有关于对象和过程的区别，学过 Python 的读者大概会有所理解。我了解不深，也就不班门弄斧了。在这里，缺少相关知识也并不会影响你制作一个基础的 R 包，只需要记住 R 包中只放函数即可。

> 关于 R 包编译安装时发生的事情：
>
> 在 R 文件中写下 `time = Sys.time()`，即获取当前系统时间并赋予变量名 `time`。如果是 `source` 这个 R 文件，就会发现 `time` 变量的取值是执行 `source` 命令的时间。因为 `source` 一个 R 文件将逐行执行该文件内的代码。执行到 `time = Sys.time()` 时，`time` 变量将储存当前（执行时）的时间。

> 但如果该文件作为 R 包的 R 文件。则在安装该包时，这行代码将被执行一次，此时 `time` 变量将储存下当前时间（即安装的时间）。随后， `time` 变量将作为一个对象，和它的值一起被保留下来。在你 library 该包时， `time` 代表的对象会和它的值被一起导入。此时查看 `time` 变量的值，会发现它储存的时间是安装这个包的时间。
>
> 也就是说， R 包的 R 文件中的代码只会被执行一次，这意味着它无法达到我们预期的效果（比如说library某一个需要的包）。因此我们并不建议在 R 包的文件中写需要被执行的代码，而是只写可以被保留的对象（比如说函数，以及类、方法、范型等）。

> 补充：如果要使用别人的包，那么可以在函数内部写 `library` 。

H2 1.4 小结

在 R 文件夹中创建了相应的 `.R` 文件后，我们就可以安装这个 R 包来试试效果了。如果使用的是我提供的 `randomwalkli` 案例，就可以直接使用 `main()` 函数，来绘制随机游走的序列图。

此时，我们已经创建了一个简单的 R 包，它能够被正常的编译安装，并实现所需要的功能。但还需要解决的问题是，如何让你的包可以在别人的电脑环境下运行，以及如何让别人知道怎样使用你的函数。

> 关于运行环境：
>
> 在包的功能比较单一，涉及的函数比较少的时候，环境问题主要就是包的依赖问题。比如说，素材中的 `plot.chain` 函数，需要调用 `ggplot2` 包中的某些函数。那么，如果别人电脑上还没有安装 `ggplot2` 包，我们的函数在 `library` 时，就会报错。 R 包开发的很重要的一个问题就是解决这样的包依赖问题。解决方法在 2.1 节 (5) 中。

H1 2 修改说明文档

第一小节所构建的 R 包项目已经可以实现我们所需要的功能，这一小节主要是解决包的各种描述性信息以及环境依赖问题。这是为了保证新用户能够快速理解 R 包的使用方法，以及保证 R 包能够正常工作。

H2 2.1 DESCRIPTION

> 关于文件格式：
>
> 事实上，整个项目里面的所有文件都是文本形式的文件。例如 代码的 R 文件， 说明文档的 Rd 文件，DESCRIPTION 和 NAMESPACE 文件。

DESCRIPTION 是一个文本文件，每一行对应于一类特定信息。我们可以使用 Rstudio 打开，也可以使用 txt、VScode 等文本编辑器打开。打开后我们逐行来看其对应的信息。

```
Package: randomwalkli
Type: Package
Title: randomwalk
Version: 0.1.0
Authors@R: person('Hadley','Wickham',email = '...',role = c('aut','cre'))
Maintainer: The package maintainer <yourself@somewhere.net>
Description: to plot ....
```

```
License: CC0
Encoding: UTF-8
LazyData: true
Imports:
    ggplot2
RoxygenNote: 7.2.2
```

第一行是包的名称，类似于 "ggplot2" "MASS"。包名将完全取决于此，但不建议随便修改，建议和文件夹名称、项目名称，以及未来GitHub上的仓库名称对应。

第二行是项目类型，"Package" 是指这是一个包。不可随意修改。

(1) 标题和描述

第三行和第七行分别是 Title 和 Description，是对包的功能的一个描述。其中标题相对简短，描述更加详细。如下是包 ggplot2 的标题和描述，可以看到是对包功能的介绍。

```
Title: Create Elegant Data Visualisations Using the Grammar of Graphics
Description: A system for 'declaratively' creating graphics,
    based on "The Grammar of Graphics". You provide the data, tell
'ggplot2'
    how to map variables to aesthetics, what graphical primitives to use,
    and it takes care of the details.
```

这一部分手动修改并保存即可。

(2) Version

第四行 Version（版本），由三部分构成： 大版本，子版本，补丁版本，由点号隔开。

如果要上传到 CRAN 或其他的大型平台，版本号要求比较严格。例如 `ggplot2` 包的最新版本已经是 3.3.6 。而我们一般上传到 GitHub 上，版本就没有人进行规范，所以简单注意即可。

(3) Author and Maintainer

第五行作者，显而易见就是给出包的作者信息。作者信息有着非常标准的书写格式，即 Authors@R: c(person(), person(),...)。

事实上，这是一段 R 中的可执行代码（@R: 应该就是指在R中可以被 call 并执行），所以一位作者的信息可以由 R 的一个函数给定，其中包含的参数如下所示。因此如果是多位作者，就需要使用 `c()` 来创建一个向量。示例如下：

```
Authors@R:person('Hadley','Wickham',email = '...',role = c('aut','cre'))
或者多作者：
Authors@R:c(
    person('Si','Li',email = '...',role = c('aut','cre')),
    person('San','Zhang',email = '...',role = c('aut')))
```

```
person(given = NULL, family = NULL, middle = NULL,
       email = NULL, role = NULL, comment = NULL,
       first = NULL, last = NULL)
```

```
# given
# a character vector with the given names, or a list thereof.
#
# family
# a character string with the family name, or a list thereof.
#
# middle
# a character string with the collapsed middle name(s). Deprecated, see
Details.
#
# email
# a character string (or vector) giving an e-mail address (each), or a
list thereof.
#
# role
# a character vector specifying the role(s) of the person (see Details),
or a list thereof.
#
# comment
# a character string (or vector) providing comments, or a list thereof.
```

注意，这个地方需要**给且仅给**出一位维护者，即角色标签中包含 "cre" (`role = c("cre",...)`)，且维护者必须提供邮箱。

第六行是 Maintainer ，这个信息会在author中给出。根据参考资料 3 显示，该信息在后续会由 Author 中的信息自动生成。因此，保证 author 里面有且只有一个人的角色包含 cre ，且有邮箱即可。

Authors 部分的信息可以手动修改。

(4) License

第八行是包的版权许可声明，表明这个包的版权许可类型，即他人使用该包的信息时，必须遵守的版权规定。包括但不限于：

- MIT 其允许人们使用和发布包的代码，唯一的限制是许可证必须和代码一起发布
- GPL-2 意味着任何包含你的代码的包都必须使用GPL兼容的许可证来发布，此外，任何人发布代码的修改版本都必须公布源代码
- CC0 这个许可证放弃了你对代码和数据的所有权，任何人都可以自由的把它用于任何地方

可以手动敲上去，但建议使用代码 `usethis::use_mit_license()` 或者 `usethis::use_cc0_license()` 等。

(5) 依赖包

第十一行是在声明我们的 R 包所需要的依赖包。这是在解决 1.4 节 提到的环境问题。

该问题的起点是，你不能保证我们所需要的某些包，例如我们使用了的 ggplot2 包，在用户的电脑上已经安装过了。如果他没有安装 ggplot2 ，但你却在你的函数里面调用了 ggplot2 的函数，那么显然会报错。

所以我们需要在此处声明依赖的包。声明之后，当其他人安装这个 R 包时，会同时检查是否安装有依赖的包，如果没有则默认帮你自动安装。

对该部分进行修改，我们建议使用函数`usethis::use_package("packagename")`，例如`usethis::use_package("ggplot2")`，当然也可以按照格式，手动输入（不建议）。

> 关于`usethis::use_package("packagename")`的参数：
>
> 建议使用`?usethis::use_package`查看帮助文档，因为该函数提供了非常丰富的选项，以实现对依赖包的管理，比如说控制依赖包的版本。

> 关于依赖包和导入包的区别：
>
> 这里添加的依赖包，只意味着用户在安装这个R包时，会检查是否安装了依赖包，如果没有，则帮助安装。但是在函数内部调用的时候，仍然需要重新导入`library(ggplot2)`，或者在NAMESPACE 中声明。

H2 2.2 文档

一个好的文档应当帮助使用者更快地上手你的包的功能 / 函数。

H3 2.2.1 函数文档

几乎所有的 R 包的函数都有相应的函数文档，它能够帮助使用者了解函数实现的功能、传入参数、返回值，并提供案例、参考文献等诸多信息，以帮助使用者快速地了解该函数的用法。

以最简单的均值函数`mean`为例，使用`help(mean)`或者`?(mean)`代码，就可以找到它的帮助文档。其中给出了 描述(description)、 用法(usage)、 参数(arguments)、 返回值(value)、参考文献(references)、 相似内容(see also)、 案例(examples)。

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An `R` object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical evaluating to `TRUE` or `FALSE` indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`weighted.mean`, `mean.POSIXct`, `colMeans` for row and column means.

Examples

Run examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```


那么如何为函数创建一个帮助文档呢？该过程分为两步：书写注释，并根据注释创建对应的文档。

2.2.1.1 书写注释

在需要帮助文档的函数上方书写注释，注释与帮助文档的条目一一对应。此外，为了与 R 文件以 # 开头的注释相区分，函数的注释使用 #' 开头。例如：

```
#' Plot random walk chains
#'
#' It is an example of create a R package. This function is used to
#' make a set of random walk chains. We hope to
#' make it easy to plot random walk chains.
#' Thank you for using it.
```

```r
#'
#' @param p.length length of the generated chain.
#' @param p.prob probability to get positive in once walk.
#' @param p.num number of chains to generate.
#' @param p.seed random seed.
#'
#' @return  No return but plot the figure.
#' @examples
#' main(100,0.5,5,1)
#' main(100,0.51,5,1)
#' main(100,0.4,5,1)
#' main(100,0.6,5,1)
#' @export
#' @importFrom ggplot2 ggplot aes geom_line geom_abline
main = function(p.length,p.prob,p.num,p.seed = 1){
  dt.plot = gene.chain(p.length,p.prob,p.num,p.seed)
  toplot = plot.chain(dt.plot)
  plot(toplot)
}
```

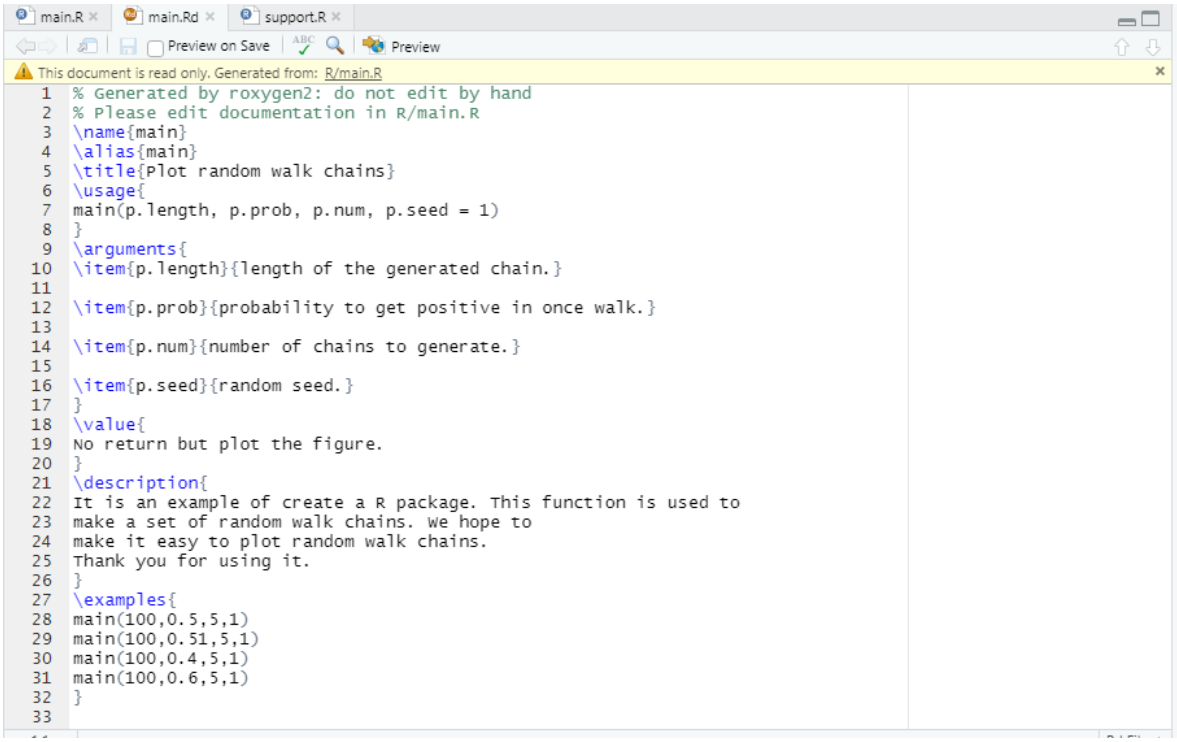
1. 注释的前两个非空部分分别对应于函数的标题和描述：
    - 第一行是函数的标题："Plot random walk chains"
    - 与标题间隔一个空行的是函数的描述："It is an example ... ... for using it."
2. 除了标题和描述外，其余项目使用 @ 来标识后面跟随的内容。比如说：
    - 参数：格式为 `#' @param <para.name> description of the parameter` 即 @param 后空一格是参数的名称，再空一格就可以书写该参数的描述。该部分对应于帮助文档中的 Arguments 的部分。
    - 返回值：格式为 `@return Description of the return value.` 即 @return 空一格后直接书写对于返回值的描述。该部分将对应于帮助文档的 Return 部分。
    - 余下还有 案例(@examples)，参考文献 (@references) 等诸多格式。具体参见官方说明文档。
3. 最后是 导出和导入 模块。该部分不用于生成帮助文档，而会被用于生成 NAMESPACE 文件。其中 `@export` 是指该函数将在 library 时被导入命名空间，即可以被用户使用；`@import <packagename>` 或者 `@importFrom <packagename> <function1> <function2> ...` 是指该函数依赖于其余包或者其余包中的某些函数，那么下一次在 library 我们的包时，这些包或者函数也会被导入。

> 关于导入和导出模块：
>
> 该模块并非制作一个 R 包必须掌握的内容。我们会在2.3节，有关NAMESPACE 的部分讲解如何绕过这两部分。

2.2.1.2 创建文档

在书写完某个函数的注释后，我们就可以使用命令 `devtools::document()` 来创建帮助文档。该函数将遍历所有的 R 文件，并自动创建（或更新）所有的帮助文档（以及 NAMESPACE 文件）。

所有的帮助文档都会被保存在 `./man` 路径下，文件名是函数名 `function.Rd` 。这是一个 Rd 格式的文本文件，使用 Rstudio 或 任何文本编辑器打开，然后会发现这是一个类似于 markdown的文件格式。



考虑到 R 也不会检查文档是怎么产生的。所以另外一种创建的文档的方式自然是，自己按照相同的格式书写注释，或者在已有的文档上进行修改。当然，我们并不建议如此操作，因为下一次运行 `devtools::document()` 函数时，会更新所有的帮助文档。

> 事实上，如果从 Rstudio 中打开，它会提示你该文档由函数 `devtools::document()` ，基于另一文件生成，因此在 Rstudio 中是只读 (read only) 的，以避免被误改。

通过不同的关键词 `@<keyword>` ，我们还可以添加其它很多类型的内容。例如， `?mean` 中我们可以看到的 reference （参考文献）, see also （相关指引）等。可以根据自己的需求，查看说明文档或教程。

2.2.2 包文档等非函数文档

事实上，不仅函数有自己的说明文档，包 (package) 自身也有说明文档。使用命令 `help(ggplot2)` ，我们就可以看到 ggplot2 的说明文档。那么如何创建包的文档呢?

包的文档创建类似于函数文档，也是写在 .R 文件中，然后使用命令 `devtools::document` 时，会自动生成相应的文档。因此我们只需要了解 包的说明文档的书写格式即可。

首先，在 R 文件夹中，创建一个空的 `.R` 文件，并写入一段同样以 `#'` 开头的注释。与函数相同，该部分注释内容的每一行同样对应于特定的信息。与函数文档的区别是，注释下方不再是某个函数，而是空值 `NULL` 。例如：

```
#' randomwalkli : package to learn how to develop R package
#'
```

```
#' It is an example for R package development.
#' The main function of it is to create a plot,
#' which is consist of several random walk chains.
#'
#'
#' @section 1 The purpose of package:
#' Function main is used to ... The key function is function main
..........
#'
#' @section 2 Conclusion:
#' We hope this case can help you learn about the development of a R
package.
#'
#' @docType package
#' @name randomwalkli

NULL
```

- 第一行是标题 (title)，与标题相隔一个空行的是描述 (description)。
- 之后的每一部分仍然使用 @ 加关键词指定内容属性：
    - 例如 `@section` ，类似 Latex 中的 section。其中关键词后空一格是这个 section 的标题，标题以冒号 `:` 结尾。换一行书写 section 的内容。
- 最后两行分别是文档类型 (docType) 和 名称 (name)：
    - 文档类型为 package，指这个说明文档给出的是 package 的信息。
    - 名称一般就是包的名称，也是搜索这个说明文档时所需要的名称。例如 `help("randomwalkli")` 。如果 name 位置书写的是 `rwl` ，则需要使用命令 `help("rwl")` 搜索该篇说明文档。

> 关于 文档的 name 属性：
>
> 如果仔细思考上述关于包的名称和文档类型的说明之后就会发现，并不是某个文档和包绑定了，只是而是恰好有一个和包名字相同的文档。因此，当我们用 `help` 命令搜索包的名称时，就刚好会找到这个文档。所以我们会在这个文档里面写包的说明文档。
>
> 同样的思路可以为其他对象添加相应的文档，比如说类(class)、数据集等。此外，文档也有更多的关键词以实现较为复杂的文本格式。

H2 2.3 NAMESPACE

H3 2.3.1 绕开 NAMESPACE 文件

> 关于NAMESPACE 的导入和导出：
>
> 事实上，在制作一个功能相对单一的 R 包时，我们可以不需要修改该文件。因为无非是
>
> (1) 不指定导出的函数，而是把所有函数导出。

> (2) 不指定导入的函数/包，而是在函数里面写 `library` 来导入，或者 `package::function()` 的形式来使用依赖包的函数。

导入依赖包的两种简单方法：

```
## 方法一
plot.chain = function(dtplot){
  library(ggplot2) # 直接在需要的函数里面写library
  p = ggplot(dtplot,
             aes(x = Times,y = Value,group = chainId,color = chainId))+
    geom_line()+
    geom_abline(intercept = 0,slope = 0)+
    geom_abline(slope = 0,intercept = -20,col = 2)
  return(p)
}
## 方法二
plot.chain = function(dtplot){
  p = ggplot2::ggplot(dtplot,
                      ggplot2::aes(x = Times,y = Value,
                                   group = chainId,color = chainId))+
    ggplot2::geom_line()+
    ggplot2::geom_abline(intercept = 0,slope = 0)+
    ggplot2::geom_abline(slope = 0,intercept = -20,col = 2)
  return(p)
}
```

2.3.2 不绕开 NAMESPACE 文件

(1) 内容

该文件包含两部分内容：

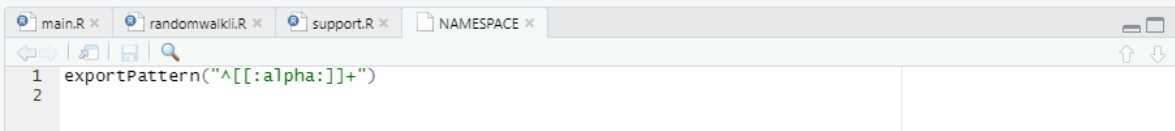
- 你希望从自己的包里面导出去 (export) 的函数
    - 不是所有的函数都需要公开给使用者，某些函数可能只是会被下一个函数调用。
    - 为什么只导出部分函数：
        - 某些函数只是过程量，用户用不到。如果这样的函数比较多，而且全部导进去的话，不方便用户使用。
        - 这些函数往往我们不会为其书写说明文档。
    - 比如素材中，我们只希望用户使用 `main` 函数来绘图，而 `gene.chain` 和 `plot.chain` 函数并不会被使用。因此我们只导出 `main` 函数。
- 你希望从别人的包里面导进来 (import) 的函数
    - 为什么不直接在函数里面使用 `library` 来导入包：
        - 我们只是用到包的一部分函数，全部导入容易产生不必要的错误。
        - 不需要每一个函数里面都写一次 `library` ，这样不方便管理。

- 比如说，素材中只使用了 `aes` `ggplot` `geom_line` `geom_abline` 四个函数，那我们可以只导入这四个函数。

(2) 修改方法

如何修改 NAMESPACE 文件：

首先，在项目创建时，会默认生成一个 NAMESPACE 文件：



这一行命令的意思是，将所有的函数都展示给用户。比如说我的 `gene.chain` 和 `gene.plot` 。同时也不导入任何需要的函数。因此我们需要做的步骤是：

- 确认哪些函数是需要被用户使用的，并进行标识 `#' @export` 。（参见2.2.1.1 节的注释写法）
- 确认哪些函数是需要被我们的函数所引用的，并告诉我们的系统 `#' @importfrom pck fun1 fun2 ...` ，或者是某些包 `#' @import pck` 。（参见2.2.1.1 节的注释写法）
- 使用 `devtools::document()` 函数，更新 NAMESPACE 文件。（由于该函数只会修改由它创建的文档，因此需要把之前默认生成的 NAMESPACE 文件删除）

2.4 小结

在第一节，我们实现了 R 包的功能；这一节，我们实现了各类文档信息的修改。至此，一个简单的完整的 R 包就已经开发好了。随后，我们就可以把整个项目文件发送给用户，让他们使用 `devtools::install()` 函数在自己的电脑上安装这个包。但这显然并不方便。因此我们需要把开发好的 R 包项目上传至 GitHub，使得用户只需要一行命令就可以从 GitHub 上获取并安装我们的 R 包：

```
devtools::install_github("username/packagename")
```

因此下一节就是如何将项目上传至 GitHub。

3 上传到GitHub

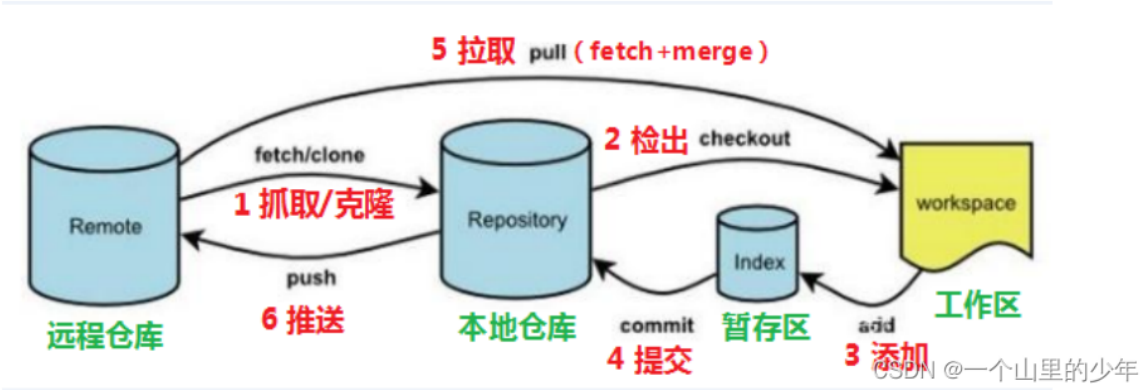
3.0 不依赖于 git 系统的上传方法

将一个 R 包上传至 GitHub 平台的本质，是将项目的所有文件上传上去。显然，在创建完 GitHub 仓库后（如何创建仓库参见 3.2.1 节），直接把文件复制上去也是可行的。这样做的好处是非常简单，不需要再学习 git 系统的使用。但缺点是，一旦需要多次修改，那么这件事情就会变得非常麻烦。所以，是否学习 git 系统的内容取决于自身需求。

就我目前所学习到的， git 系统是一个非常好用的文本文档的版本管理系统。所以即使不是为了上传 R 包，学习它也是有意义的。基础部分的学习参考自以下链接：

> git 命令相关资料部分参考自以下Blog：
>
> 1. (20条消息) 史上最简单Git入门教程孤单品尝寂寞的博客-CSDN博客git入门
> 2. (20条消息) Git入门(建议收藏)一个山里的少年的博客-CSDN博客git的入门介绍
>
> 其中第一条blog 有对 git 系统原理的解释。

3.1 git 系统简介



Git是一个分布式版本控制系统，其中包含了非常丰富的功能。仅我所了解的就包括 **版本管理** 和 **多人协作**。虽然将一个制作好的 R 包上传到 GitHub 上，需要涉及的知识其实非常少。但考虑到整个过程不可能绕开 **添加(add)** 和 **提交(commit)** 操作，因此我们仍将简单介绍一下版本管理相关的内容。

3.1.1 安装git软件

第一件事情是安装 git 软件。直接从官方网站安装即可 Git (git-scm.com) ，选择合适自己电脑的版本。安装过程一路继续即可，因为其中大部分内容我也不清楚具体含义和区别。

在安装之后，需要设定用户名和邮箱。这是因为未来的所有提交操作都需要记录是由谁进行的。

设置方法：打开命令行cmd，并执行以下两行命令：

```
git config --global user.name <username>
git config --global user.email <email>
// 例如
git config --global user.name "liwch"
git config --global user.email "22xxxxxxxx@qq.com"
```

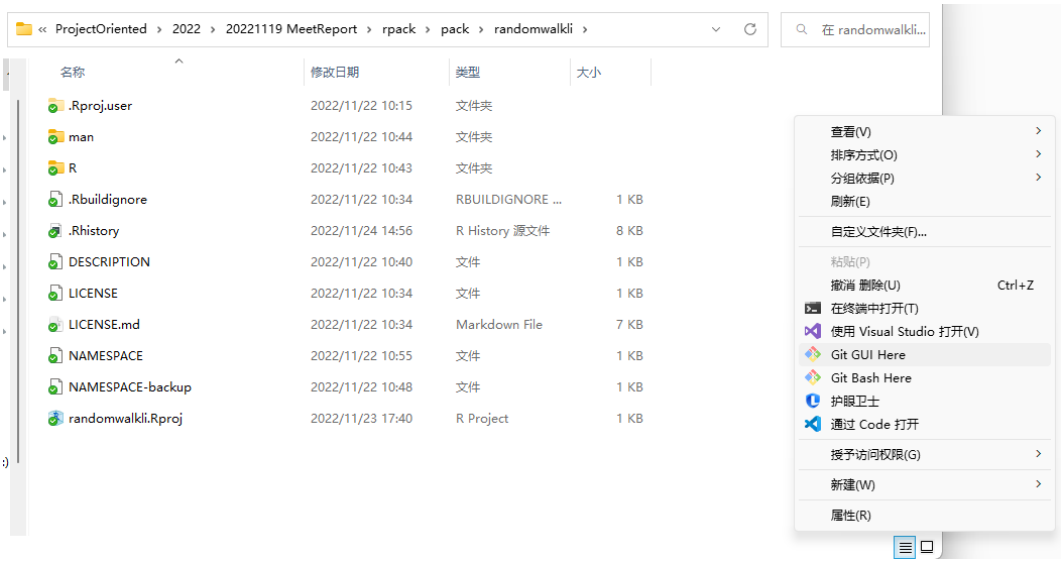
> 如果提示找不到 git 命令，则大概率是没有添加环境变量，网上搜索如何添加环境变量即可。

3.1.2 建立本地仓库

在工作区建立本地仓库：

打开Git命令行或者图形化界面，这里使用命令行以方便演示：

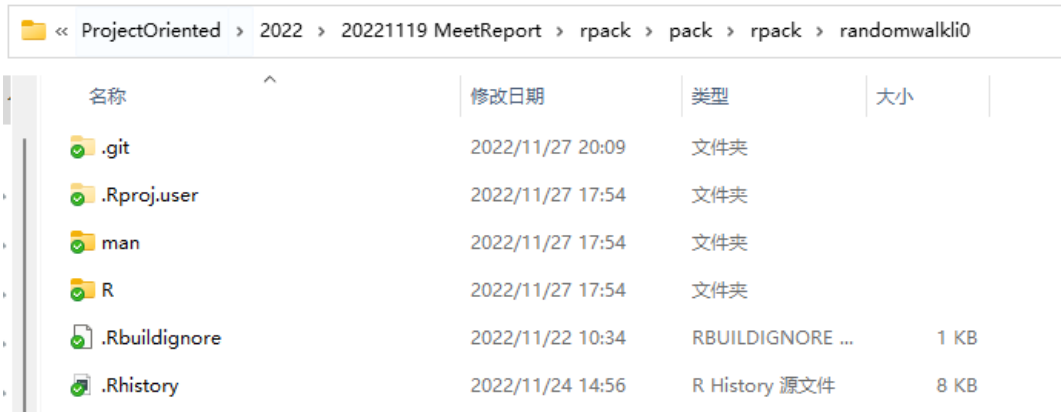
1. 在项目所在路径下右键菜单，会出现两个选项（Git bash here / Git GUI here），分别是以命令行形式打开 和 以图形化界面形式打开。



2. 执行 `git init` ，其中 `init` 是初始化命令。

```
22907@DESKTOP-74C8EBN MINGW64 /d/OneDrive/WorkingTable/ProjectOriented/2022/2022
1119 MeetReport/rpack/pack/rpack/randomwalkli0
$ git init
Initialized empty Git repository in D:/OneDrive/WorkingTable/ProjectOriented/202
2/20221119 MeetReport/rpack/pack/rpack/randomwalkli0/.git/
```

3. 而后在工作目录下会生成一个隐藏文件夹 .git，这就是本地仓库和暂存区所在的文件夹。这个工作目录除了 .git 文件夹之外的部分就是工作区。



H3 3.1.X 工作区、暂存区 和 本地仓库

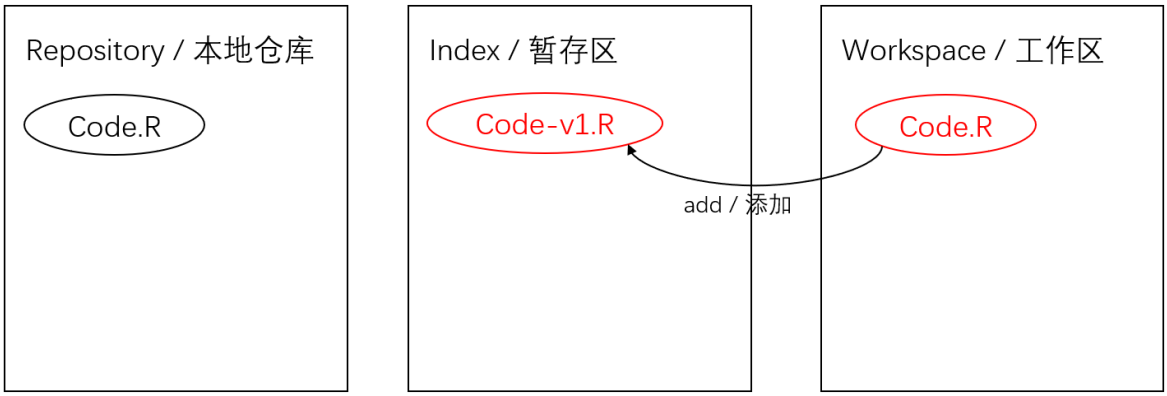
在开始之前，我们需要简单介绍一下工作区和本地仓库之间的关系：

在实际工作时，我们往往会将代码储存在一个文件夹（如上图，即本地仓库）内。当我们需要修改代码时，我们当然可以直接在 **本地仓库** 里面的 .R 文件上修改。这样做的缺点是，如果修改出现了问题，一般是不能后悔的。

为了安全起见，我们会把 **本地仓库** 中的 .R 文件复制到 **工作区**，而后修改 **工作区** 中复制的文件。把文件从本地仓库中复制到工作区的过程叫做 **checkout（检出）**。（如下图）

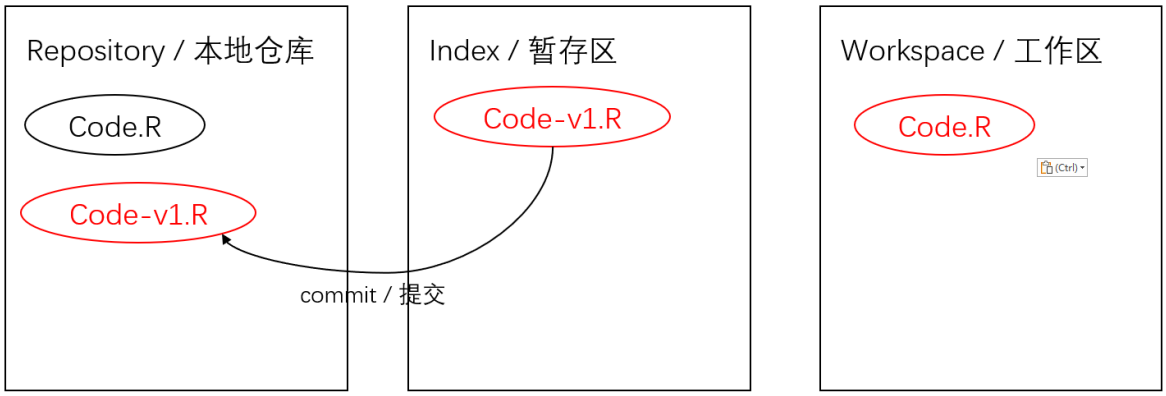
在修改之后，我们还需要把修改好的文件复制回去。当然，我们可以直接覆盖掉原文件，但这意味着我们还是失去了后悔的机会。比较合理的思路是把两个版本的文件都保存下来。那么我们可以给修改后的文件做一个备注，与原始文件区分开来，然后再复制回去。

这个过程中，我们会先将修改后的文件复制进 **暂存区**，这个过程叫做 **add（添加）**。（如下图）



暂存区 的文件仅仅是临时保存你的修改，如果确认修改需要被保存下来，那么需要把 **暂存区** 的文件复制到 **本地仓库**，这个过程就是 **commit（提交）**。

而且在提交时，我们还会为暂存区的文件提供一些额外信息。比如版本号和备注信息。那么提交完成后，仓库中将保存有 原始版本 和 修改后的 v1。（如下图）



这之后，我们可以在 **工作区** 已经有的文件上进行修改，并再次执行 **添加** 和 **提交** 的步骤，使得 **本地仓库** 中增加新的版本。但也有可能说，我们不小心把工作区的文件删除了，或者我们想基于原始版本进行修改，那么我们就可以从本地仓库中 **检出** 原始版本 或者 v1版本 进行修改。而后再执行添加和提交步骤。

此外，在添加和提交步骤中，我们并没有要求工作区的文档必须来源于本地仓库。因此，我们也可以在工作区新建一个 **文件**，并通过 **添加** 和 **提交** 步骤，将新文件储存在 **本地仓库** 中。

> 关于 git 系统：
>
> Git 系统就是将这个过程自动化了，只需要 `checkout` `add` `commit` 这三个命令，就可以自动复制并记录相关的版本信息。而 .git 文件夹就是把 本地仓库 和 暂存区 的合体，并且额外储存了日志文件等额外信息。为了避免用户的手动修改，使得版本信息被破坏， .git 文件夹默认对用户进行隐藏。当然，我们也不建议手动修改其中的任何内容。

> 补充一点，如果细看会发现， R 包项目中的所有文件都是以文本形式进行存储，类似于 .txt / .tex / .R / .md。所以我们只需要研究明白一个 文本文件 的情况即可。似乎，Git系统大多数时候只处理文本文件。

H3 3.1.3 将工作区的内容提交到本地仓库

虽然我们讲解了检出，但是提交代码实际上只需要 add commit push 三步。当我们在工作区已经将所有的项目文件准备好后，我们只需要执行 **添加** 和 **提交** 操作，就可以把所有文件保存在本地仓库中。

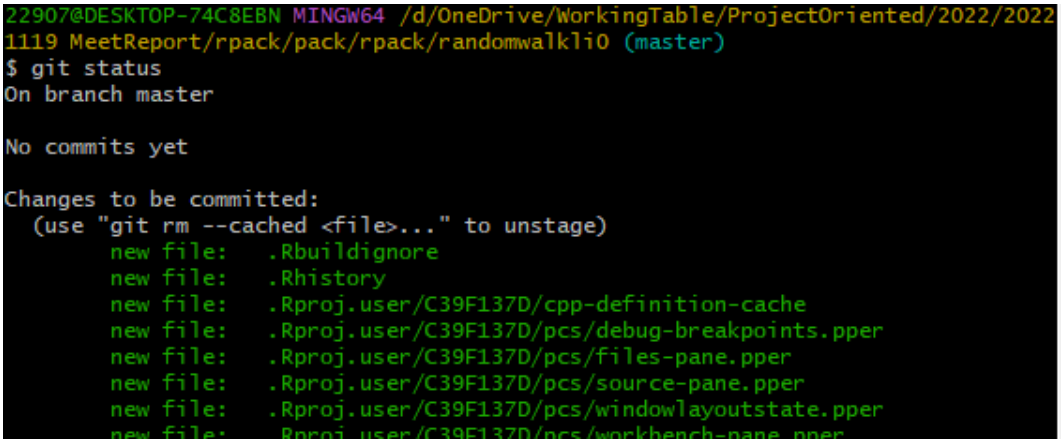
1. 将工作区的内容添加至暂存区
    - 查看目前文件状态 `git status`

```
22907@DESKTOP-74C8EBN MINGW64 /d/OneDrive/WorkingTable/ProjectOriented/2022/2022
1119 MeetReport/rpack/pack/rpack/randomwalkli0 (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .Rbuildignore
        .Rhistory
        .Rproj.user/
        DESCRIPTION
        LICENSE
        LICENSE.md
        NAMESPACE
        NAMESPACE-backup
        R/
        man/
        randomwalkli.Rproj

nothing added to commit but untracked files present (use "git add" to track)
```

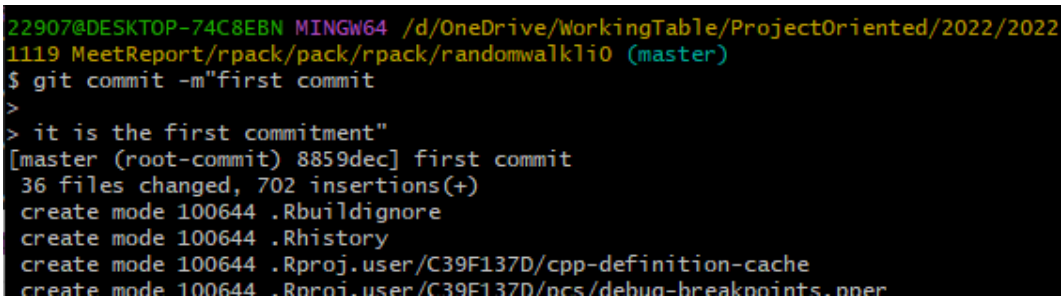
    - 提示内容表明，标红的文件均是新文件，本地仓库中没有相关信息。随后我们就可以使用命令将工作区内容添加到暂存区：
        - 添加特定文件 `git add <filename>`
        - 添加全部文件 `git add .`

```
22907@DESKTOP-74C8EBN MINGW64 /d/OneDrive/WorkingTable/ProjectOriented/2022/2022
1119 MeetReport/rpack/pack/rpack/randomwalkli0 (master)
$ git add .
warning: in the working copy of '.Rbuildignore', LF will be replaced by CRLF the
 next time Git touches it
warning: in the working copy of '.Rhistory', LF will be replaced by CRLF the nex
t time Git touches it
warning: in the working copy of '.Rproj.user/C39F137D/pcs/debug-breakpoints.pper
```

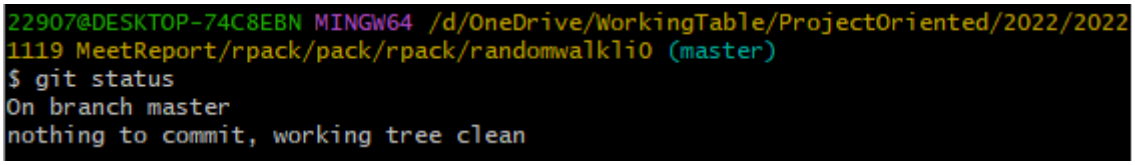
2. 将暂存区的内容正式提交到本地仓库
    - 查看目前文件状态 `git status`



- 提示内容表明，标绿的文件是新添加的文件，正在等待被提交。随后，我们可以使用命令将暂存区内容添加到本地仓库：`git commit -m<description>`
    - 第一，提交步骤必须给定注释参数。这部分信息不会影响文件内容，但会被记录在日志里面。
    - 第二，该命令是将暂存区 **所有的** 未提交内容全部提交给本地仓库。



3. 最后再查看一次状态，提示内容表示并不存在未提交的文件，而且工作区内容较上一次添加，并没有发生修改。

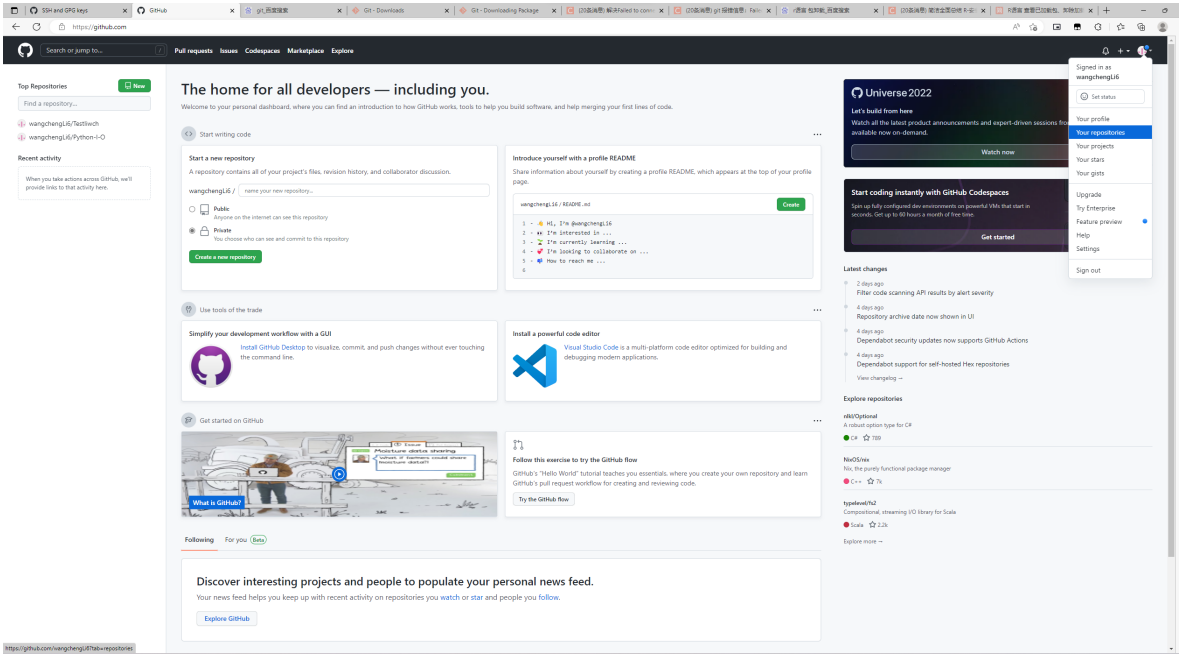


H2 3.2 GitHub 仓库

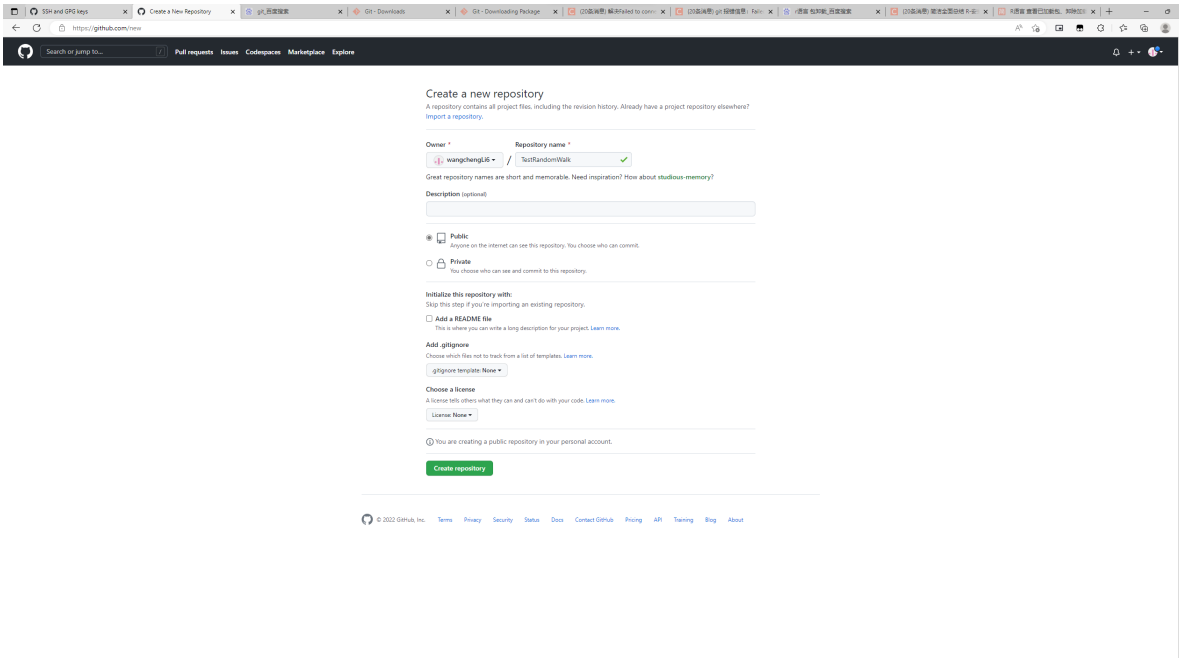
当我们把项目文件全部存入 **本地仓库** 后，我们还需要做的事情就是，把本地仓库的内容全部 **推送(push)** 到**远程仓库**（如GitHub）。首先，我们创建一个空白的 GitHub 仓库。

H3 3.2.1 建立GitHub仓库

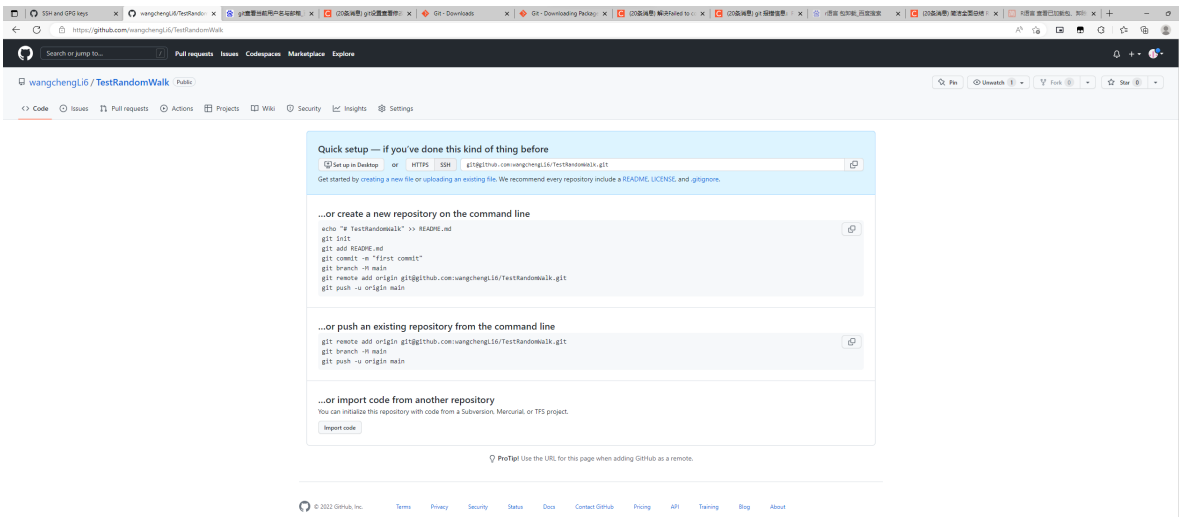
注册账号并建立一个空白仓库



选择新建仓库，并给定仓库名称即可。（选择公共仓库才可以被别人看到）



新建完成后，GitHub会提示你如何进行操作：



H3 3.2.2 将本地仓库推送至远程仓库（GitHub仓库）

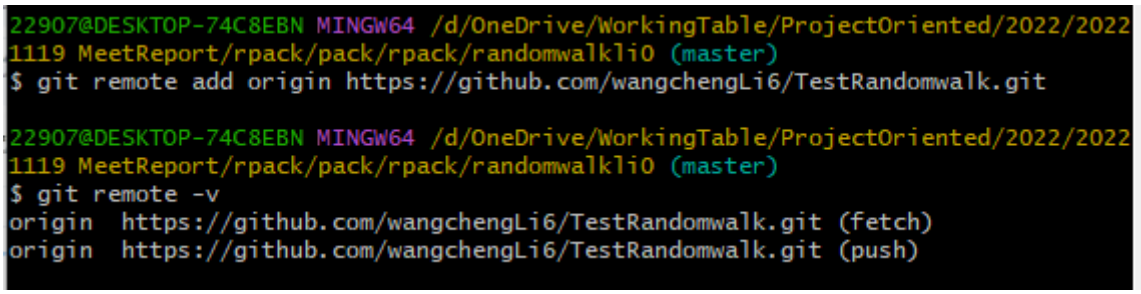
1. 将本地仓库与远程仓库相连接

执行命令：

```
git remote add origin <仓库链接>
\\ 例如：
git remote add origin https://github.com/wangchengLi6/randomwalkli.git
```

其他相关命令包括：

- `git remote -v` 查看远程仓库地址
- `git remote rm origin` 删除远程仓库地址
- `git remote add origin https://github.com/wangchengLi6/randomwalkli.git` 添加远程仓库地址
- `git remote set-url origin https://github.com/wangchengLi6/randomwalkli.git` 修改远程仓库地址

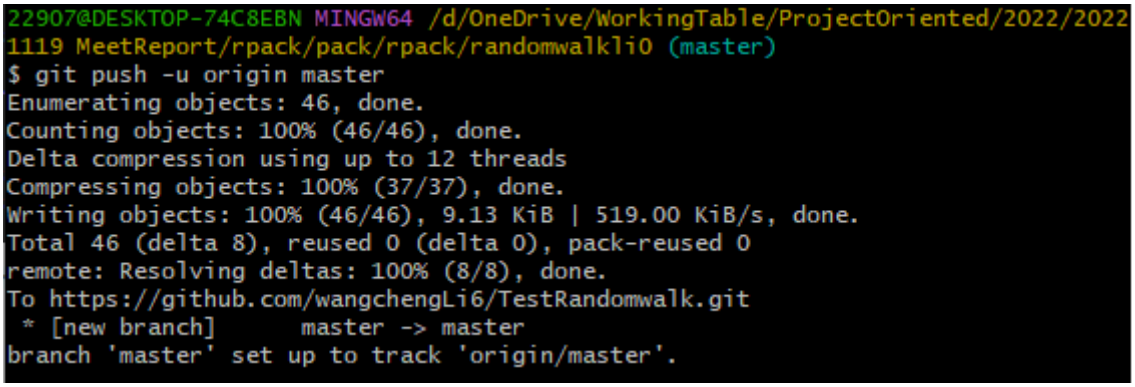


2. 连接之后，我们只需要将本地仓库的内容 **推送(push)** 至远程仓库（你所给定的远程仓库链接）即可。

执行命令：

```
git push -u origin <branch name>
\\ 例如：
git push -u origin master
```

其中 master 是需要推送的分支名。 `-u` 命令是第一次push 的时候添加的参数，这会默认将本地仓库和远程仓库的 master 分支联系起来。



> 关于身份认证：
>
> 显然 GitHub 不会让人随便修改别人的仓库，所以在第一次push的时候，GitHub 会需要你提交身份认证，可能是图形化的登录界面，也可能是代码行中输入用户名和密码。按照提示操作即可。

> 补充：默认主干分支的名称是 master。可以使用命令 `git branch` ，来查看所有已有的分支。由于并未讲到创建分支和修改分支，因此并不做更多叙述。

然后就可以去 GitHub 上查看相关情况。此时，别人就可以通过 `devtools::install_github("username/repositoryname")` 命令来安装你的 R 包了。

如果之后对 项目里的相关文件进行了修改，只需要先将修改后的文件 **添加(add)** 并 **提交(commit)** 到本地仓库，再 **推送(push)** 到远程仓库即可。第二次 `push` 不需要 `-u` 参数

> 关于科学上网：
>
> 在 push 的时候可能会出现的问题是：如果不使用科学上网的方法，不一定能够成功登录GitHub。但是使用了科学上网的方法后，有可能出现 DNS 解析错误导致连接超时的问题。这个问题有一些不同的解决办法，可以不科学上网而是强行连接，也可以使用命令行添加科学上网的相关信息。这里不做过多介绍，如果出现报错，只需要复制错误信息百度搜索即可。

4 结语

不出意外的话，前面三节的内容能且仅能帮助读者制作一个简单的 R 包并上传至 GitHub 平台。这离一个合格的 R 包还有很远距离，但满足科研的交流和推广目标也基本足够。同时，它也可以为初学者搭建一个 R 包开发的大体框架，剩下的内容大多是在这个框架上的缝缝补补。如果需要更加详细或者进阶的内容，读者可以阅读学习第0节中给出的后两个参考资料。最后，我们在这里给出可能的进阶功能以供借鉴：

- 提供示例数据集
- 版本管理和版本迭代
- 更加谨慎地处理包的依赖
- 兼容性问题：包括自身的不同版本、依赖包的不同版本、R 的不同版本
- 完整的错误处理机制，包括参数错误、运行错误等

- 兼容、可靠的并行运算
- 借助 c/c++ 的算法优化
- 创建更加复杂的对象：如类、方法等
- 与已有方法的嵌入
    - 例如惩罚回归相关的ncvreg包能够直接调用plot函数来绘制路径图。

感谢阅读，并希望能够有所帮助。