Step 系列：Prologue and Get-start

2024-02-22

LiChuang Huang

@ 立效研究院

Contents

List of Tables

1 Step 系列共性特征

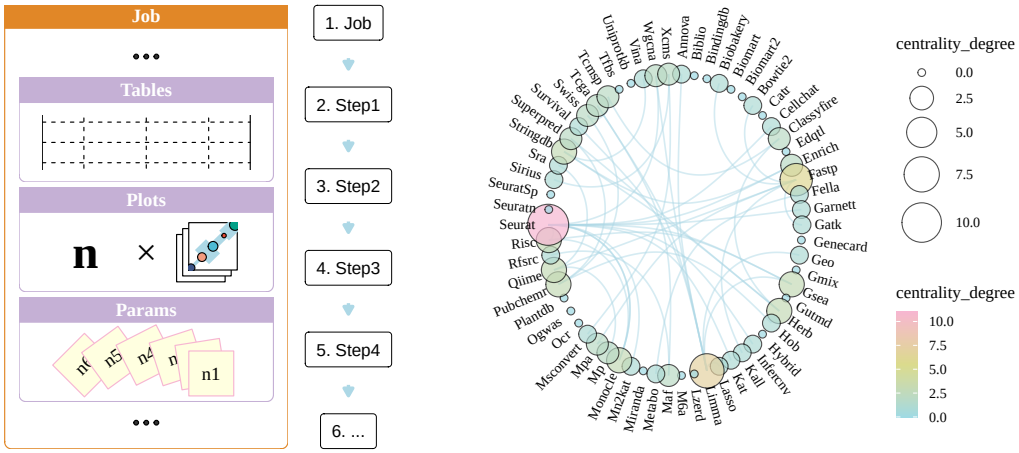

Figure 1: Workflow frame overview. 左图展示的是 Step 系列所有对象的框架结构和运行路线。右图展示的是，每一个圆球都代表一个方法或数据库或分析平台形成的数据对象，也就是左图中的 'job'，而它们之间的线，代表所有对象之间的转化或映射关系 (仅目前；还在不断拓展)；具体而言，我们可以简单的通过 'map' 或 'asjob' 这类的方法，将一个数据对象转化或映射到另一个对象，实现跨越多种方法或体系的联并分析。

1.1 宗旨

让分析变得更简单、更有效率。

1.2 理念

生物信息分析工具种类繁多，开发者出于各种原因，导致工具的适用性千差万别。学习和应用这些工具的成本可能是高昂的 (甚至它们可能有一些不为人知的漏洞)，想要把它们串连在一起分析，更是要付出一些分析之外的代价：像开发者一样调试这些程序。

Step 系列的分析方法为每一个制定好的方法、思路或工具 (一般是领域中的权威、经典或翘楚) 设立统一使用的标准，统一的数据存储，统一的应用流程，大幅度降低了学习成本、使用成本、应用成本。通过避免大量 "分析之外" 的繁琐工作，达到提高分析效率的目的。

Step 系列分析方法的一些基本特性：

- 统一的分析平台。Step 背后涵盖的工具可能涉及：R 包、Python 包、Java 包、Linux 命令行工具等。但最终用于分析的始终是 R 语言。Step 系列的方法通过在 R 中调用各种分析工具，实现不同分析平台之间的工具调用和数据对接 (降低了学习和应用成本)。
- 统一的数据存储单位。Step 系列所有的分析流程 (workflow) 都以一个对象 (Object) 存储。这避免了如果一个流程中涉及纷繁复杂的分析方法，人工存储数据 (中间数据、最终输出的图片、表格) 出错的可能 (提高效率)。必要时，通过统一的方法提取这些数据 (就像在图书馆的某一层的某一排的书架的某一个柜子取出一本书)。

- 统一的方法名称。Step 系列的所有 Workflow 的方法名称都是统一的 (step1、step2、step3…… map、vis 等)，但不会因为一同调用而出错。这是为了减免分析者的负担而设计的 (如果一次完整分析涉及十几种工具，每个工具又有十几种方法名称，那对分析者的记忆量和细心度的考验是惊人的) (提高效率)。
- 规范的分析流程。大多数的分析工具本身具备多种方法以适用灵活分析，但也提高了学习、应用成本。Step 系列的各个 Workflow 的流程都是单向的，对分析方法或思路的组合应用大都是建立在官方指南或教程的基础上，又或者是泛用性 (降低了学习成本)。如果有不同思路，可以通过关键参数调整方法，或者视情况搭建一个额外的 Workflow。因此，每个 Workflow 是以分析思路为分门别类的，而不是工具本身。
- 提供权威、经典、泛用的分析方法。Workflow 创建前，会广泛查阅时下文献，对工具择优而选、择新而用 (比如，更趋向于选择更新的来自于 Nature Biotechnology 的分析方法)。
- 提供泛用的组合思路。通过组合各种数据库、分析工具，应对千奇百怪的分析需求。同时，一些适当的组合，会发挥超出单一工具的价值 (Fig. 1 的右图提供了现如今存在的许多组合思路；同时，每个 Workflow 内部又存在一些思路组合)。
- 不断开发进化。每一个完成的 Workflow 不是固定不变的，在面临新的分析环境、新的分析需求，更多的功能会被加入到 "step" 方法中，让每一次的进步都直接应用于今后所有的同类分析。另外，一些大大小小的创新 (比如，更严谨或更酷炫的绘图) 也会不断追加到方法中。
- 效率至上。所有分析方法以输入命令形式实现，允许批量处理。
- 附带实用工具。分析流程相对固定，但通过提供一些小工具 (比如用于额外的绘图)，可以将分析更加灵活。

1.3 局限性

- 适配性。由于多数生信工具都基于 Linux，此外 Step 系列在编写时，也只在 Linux 上调试过；所以，这些工具将只适用于 Linux 系统。
- 灵活性。Step 系列的所有的方法都是封装过后的，这虽然简化了代码，降低了学习和使用成本，但是也很大程度上降低了灵活性。如果使用者打算开展一种与我所构想的分析框架截然不同的分析思路，那么这一系列的方法将不再适用。当然，对于另有奇思妙想，但缺乏某一标准分析的基本了解的使用者，这一系列的方法或函数，仍然具有参考价值，尤其是对于不同分析方法之间的数据格式转换。任何人都可以直接查看 (也能直接本地修改) 'utils.tool' 这个 R 包中每一个方法或函数。
- 片面性。受限于工作经验和认知的局限性，或许有更好的分析思路、更好的参数或者分支选择，或者有更好的分析方法的选择；在未来，可能会发现过去所封装的方法有所缺陷，甚至包含错误，这都在所难免。如果有建议或新的思路，可以直接向我提供，以更快的过时的内容或纠正其中的错误。

1.4 写在实际代码的前面

如果你熟知并且能使用 R 代码，但是对 R 的 S4 OOP (面向对象编程) 以及 '参数化多态' 并不了解 (可能多数的 R 的使用者都不知道)，那么这个系列的所有内容都会让你感到困惑。

我可以例举一些应用了 OOP 的经典 R 包：`Seurat`, `limma`, `dplyr` ……

你可能很熟悉 `dplyr`，但从没有听说过它是所谓 `OOP`，然而，它其实应用了 S3 形式的 OOP; 而单细胞分析经典工具 `Seurat`, 它的设计正是 S4 OOP。

S3 OOP 相对简单，S4 OOP 更加复杂。分析组学数据的 R 包往往都会应用 OOP 搭建分析框架，这是因为组学数据繁杂的数据体系和复杂的处理过程，正需要 OOP 来设计好各个数据的存储方式以及如何提取、转化，这样才能最大程度减轻学习、应用的负担；还因为有很多中间数据不是面向用户层面的，但是对于下一步的程序运行又是必须的，以 OOP 的 '类' 来自动化存储和利用显然是一种名智的选择。

在 Step 系列中，S4 OOP 和 '参数化多态' 被应用到了极致 (夸张的地步) 。

以下，我举例说明 S4 OOP 的创建以及应用，以便于对此陌生的使用者理解：

R input

```
.demo1 <- setClass("obj_1", contains = "numeric")
.demo2 <- setClass("obj_2", contains = "character")
.demo3 <- setClass("obj_3", representation(slot1 = "list"))
```

以上，我构造了 3 个类，并且让前两个类分别继承了 'numeric' 和 'character'。

下面，我创建了它们的示例：

R input

```
test1 <- .demo1(1)
test2 <- .demo2("This is a object of class 'obj_2'")
test3 <- .demo3(slot1 = list(1, 2, test1))
```

之后，我打算专门为它们设计属于它们的方法，让这个方法可以运用到它们任何一个实例上；但是，另一方面，我又打算让这个方法采取不同的处理策略应对它们。

或许，一种广为孰知的解决方法是：

R input

```
my_fun <- function(x) {
  if (is(x, "obj_1")) {
    message("Herein for obj_1")
  } else if (is(x, "obj_2")) {
    message("Herein for obj_2")
  } else if (is(x, "obj_3")) {
    message("Herein for obj_3")
  }
}
```

将各自处理的代码放在它们的 'if' 模块下。这当然可以，问题是，如果有成百上千个这样的类呢？……

下面演示的是一种将它们拆分开来的设计：

R input

```
# 这是整体设计框架，可以称之为 '接口'
setGeneric("my_method", function(x, ...) standardGeneric("my_method"))

setMethod("my_method", signature = c(x = "obj_1"),
  function(x){
    message("Herein for obj_1")
  })

setMethod("my_method", signature = c(x = "obj_2"),
  function(x){
    message("Herein for obj_2")
  })

setMethod("my_method", signature = c(x = "obj_3"),
  function(x){
    message("Herein for obj_3")
  })
```

你能看明白吗?

那么，以下三行代码各自会发生什么?

R input

```
my_method(test1)
my_method(test2)
my_method(test3)
```

还可以设计一种更复杂、更灵活的:

R input

```
setGeneric("my_method2", function(x, y, ...) standardGeneric("my_method2"))

setMethod("my_method2", signature = c(x = "obj_1", y = "obj_2"),
  function(x, y, sep = "\n"){
    message(x, sep, y)
  })
```

可以试试看它的效果:

R input

```
my_method2(test1, test2)
```

如果像这样输入：

R input

```
my_method2(test1, test3)
```

那么它一定会报错，因为我没有定义属于它们两个组合的方法。

如果你明白了上述的所有示例，那么，也就等于理解了 Step 系列的全部设计。

1.5 获取了 R 包之后，才能开始任何演示

1.5.1 一些额外可能需要的系统依赖工具

如果你使用的是 Ubuntu 发行版，据我的经验，安装 `devtools`, `BiocManager` 等工具之前，估计需要先安装以下：

Bash input

```
## Libraries for installing 'usethis' and 'devtools'.
sudo apt install -y libssl-dev libcurl4-openssl-dev libblas-dev
sudo apt install -y liblapack-dev libgfortran-11-dev gfortran libharfbuzz-dev libfribidi-dev
## Libraries for installing 'BiocManager' and its some packages.
sudo apt install -y libnetcdf-dev libopenbabel-dev libeigen3-dev
## Libraries For installing other graphic packages.
sudo apt install -y libfontconfig1-dev librsvg2-dev libmagick++-dev
```

1.5.2 主体：`utils.tool`

Bash input

```
git clone https://github.com/shaman-yellow/utils.tool.git ~/utils.tool
```

`utils.tool` 是标准的 R 包结构形式，这意味着，即使你不用 `git` 获取它，单纯用：

- `remotes::install_github("shaman-yellow/utils.tool")`

也能成功获取并直接安装完成。但是，这包里面大多数的方法都没有导出 (export) 到用户层次 (这是因为，这个包的改动情形太多了，我一直在创建新的方法或者调整旧的方法)，你即使 `library(utils.tool)` 加载了它，也会出现使用不了许多方法的情况。万无一失的做法是：

R input

```r
if (!requireNamespace("devtools", quietly = TRUE))
  install.packages("devtools")

devtools::load_all("~/utils.tool")
```

1.6 泛用方法

像我在 1.4 中展示的，Step 系列中有许多 '参数化多态' 的方法设计，你可能会经常见到类似下方的代码：

R input

```r
sr <- job_seurat("./data")
sr <- step1(sr)
sr <- step2(sr)
sr <- step3(sr)
sr <- step4(sr)
sr <- step5(sr)
sr <- step6(sr)

mn <- asjob_monocle(sr)
mn <- step1(mn)
mn <- step2(mn)
mn <- step3(mn)
```

那么，当我运行 `step1`，到底会发生什么？这取决于你传入 `step1` 的参数属于哪一个类 (这里主要是第一个参数) 。

我为 `step1` 和其它方法定义了近百种 (今后还在不断增加) 不同的形态，大多数时候，连我自己都忘记运行它们会发生什么。因此，快速了解对应的 `step1` 或其他方法的特定形态，是极其必要的。

你可以通过下面形式查看某种特定形态的 `step1` 的默认参数 (在 1.6.6 中对此做了演示)：

R input

```r
# 查看 'job_seurat' 对应形态的 step1
not(.job_seurat())
step1
```

注意，如果不输入 `not(.job_seurat())` 限制，你会直接看见所有 step1 的形态。

你可能会问，这只显示了它的传入参数，那如何查看它的具体的函数体呢？

像这样：

R input

```
selectMethod(step1, "job_seurat")
```

1.6.1 获取结果

所有的方法，像 step1, step2, step3 这些，都会处理传入的数据对象，并形成新的数据存储于原数据对象中。例如：

- `sr <- step1(sr)`

新的数据会被存储在 sr 的数据槽中。

统一的，一般来说，Figure 主要被存储于 'plots' 中，而 Table (data.frame) 被存储在 'tables' 中; 然而也有例外，一些中间数据或不太重要的数据，被设计存储于 'params' 中。

下面具体说明。

1.6.1.1 提取数据

示例提取 step1、step2、step3 等的结果：

R input

```
# 不要运行，仅用于说明
sr@plots$step1$p.qc
sr@plots$step2$p.pca_rank
sr@plots$step3$p.umap

sr@tables$step4$anno_SingleR
sr@tables$step6$scsa_res

sr@params$group.by
# 以上也可以直接这么写：
sr$group.by
```

总而言之，在 x@tables, x@plots, x@params 中，基本能找到所有的运行结果。

还有一种情况需要说明：例如单细胞分析中，在中途，你打算用 '原生' 的 Seurat 的代码自主分析了，而不是用封装后不太灵活的 Step 系列风格的代码。

这时候，只要这个流程封装的是其它 R 包的代码，那么基本能直接通过以下方式取得：

R input

```
# 还是以 `sr` 为示例
seurat_object <- object(sr)
# 或者这样:
seurat_object <- sr@object
```

1.6.1.2 存储（输出）数据

那么，1.6.1.1 中示例的数据提取得到后，又如何存储下来呢？保存为数据表格，.csv, .tsv, .xlsx 等，或者.pdf，对于经验丰富的 R 使用者应该是个简单的问题。这里，我主要提供了一种更快捷的方式来把它们保存下来(同统一的函数保存它们)：

R input

```
# 通过 option 设置它们的存储文件夹 (否则，会被存储在额外创建的 figs, tabs 文件夹中)
options(savedir = list(figs = "Figure+Table", tabs = "Figure+Table"))

# 然后就能保存它们了
autosv(sr@plots$step1$p.qc, "this-is-figure-1")
autosv(sr@plots$step2$p.pca_rank, "this-is-figure-2")
autosv(sr@plots$step3$p.umap, "this-is-figure-3")

autosv(sr@tables$step4$anno_SingleR, "this-is-table-1")
autosv(sr@tables$step6$scsa_res, "this-is-table-2")
```

1.6.1.3 调整图片长宽比例，然后输出

你可能会对默认的图片输出的长宽比例不满意 (虽然很多时候，我已经周到的调整过它们的比例了)，想要再做调整。

你可以这样做：

R input

```
p_my_plot <- sr@plots$step1$p.qc
p_my_plot <- wrap(p_my_plot, 20, 20)

# 或者按照相对比例
p_my_plot <- zoom(p_my_plot, 1.5, 1.5)

# 然后再输出图片
autosv(p_my_plot, "my-plot")
```

1.6.2 存储 Session (以便下次从当前状态继续分析)

R 语言自带封装好的函数 `save.image()` 可以快速存储当前会话。

你可以使用它，但也可以选择使用我专门为 Step 系列提供的函数 `saves()`。`saves()` 与 `save.image()` 仅有些许不同 (但在将来可能会有更多不同)，那就是额外存储一个 `getOption("internal_job")` 获取的对象，可以用于 `auto_method()` 快速展示当前使用的 `job` 类对象 (`getOption("internal_job")` 获取的，还包含一部分在 `step` 系列函数内部调用的 `job`)。

总之，使用 `saves` 是推荐的做法：

R input

```
# 不指定文件名，保存为 'workflow.rds'
saves()
```

下次重新加载，仅需：

R input

```
loads()
```

1.6.3 存储单个 job 对象

可以直接用 R 自带的 `saveRDS()` 存储。例如：

R input

```
# 假设我有以下一个数据对象：
sr <- .job_seurat()
# 然后存储它 (当然，它基本是空的)
saveRDS(sr, "sr.rds")
```

更复杂的情况 (我会这么做)，例如，我想要把 `sr` 这个对象存储下来，并且清空 `sr` 中我不再使用并且极其占用内存的部分——只保留必要的 Figure、Table 结果用于展示——这种情形，我在单细胞数据集分析常常遇到。你可以尝试使用：

R input

```
sr <- clear(sr)
```

完整的 `sr` 会被存储下来，而返回的 `sr` 对象将仅保留绘图或表格结果。之后，再使用 `saves()` 函数保存会话和以后加载会话，会变得轻便许多。

1.6.4 空间管理

某些 R 包生成的数据或加载的数据可能极其占用运行内存 (RAM)。便捷地，可以用 `space()` 函数查看当前内存占用。

例如，我当前的 R session 下，运行 `space()`：

R input

```
space()
```

```
## # A tibble: 17 x 3
##    name                          size  value
##    <chr>                         <chr> <dbl>
##  1 .__C__obj_1                   0 Mb      0
##  2 .__C__obj_2                   0 Mb      0
##  3 .__C__obj_3                   0 Mb      0
##  4 .__T__my_method:.GlobalEnv    0 Mb      0
##  5 .__T__my_method2:.GlobalEnv   0 Mb      0
##  6 .demo1                        0 Mb      0
##  7 .demo2                        0 Mb      0
##  8 .demo3                        0 Mb      0
##  9 .Random.seed                  0 Mb      0
## 10 .requireCachedGenerics        0 Mb      0
## 11 autoRegisters                 0 Mb      0
## 12 my_method                     0 Mb      0
## 13 my_method2                    0 Mb      0
## 14 pkgs                          0 Mb      0
## 15 test1                         0 Mb      0
## 16 test2                         0 Mb      0
## 17 test3                         0 Mb      0
```

如果你的机器的 RAM 有 32 GB 或以上，那么多数情况下一般都不需要太担心内存过载；但也有特殊情形，比如，你用 `Seurat` 加载了十多批单细胞数据集，打算将它们集成分析 (单细胞数据集一般都比较占用内存)

1.6.5 版本更新

1.6.5.1 更新主体 R 包

在 1.5.2 部分，你已经知道如何获取 `utils.tool` 这个 R 包了。

需要提醒的是，这个 R 包在频繁地被我更新，有时候是新的工具，有时候是对旧的工具的重写或改写 (一般不会太彻底)。这时候，可能需要你在本地加载最新的版本 (如果你不需要新的工具，那么也没必要一定更新)。

例如：

Bash input

```
cd ~/utils.tool
git pull orgin master
```

这样，你运行：

R input

```
devtools::load_all("~/utils.tool")
```

使用的就是最新的 utils.tool 了。

1.6.5.2 更新已存储的 job 对象

可能会出现一种极端情况：

你已经用 Step 系列完成了某一次的分析，并且将 job 对象本地存储下来；然而由于新的需求，或者发现过去分析中的错误，你打算对某个部分进行修改；可能整个计算耗时较长，让你不想从头开始；而这时候，我 (utils.tool 的编写者) 修改了这个 job 对应的代码部分，而你也早已跟新到了较新的版本。

这样，你会遇见一种麻烦：代码无法顺利运行了。

这是可能是因为我修改了 job 的数据槽 (增加或减少)，而 S4 的数据检查是严格的，会判定你旧的 job 不满足条件，因此抛出错误。

如果遭遇这种情形，请尝试：

R input

```
# 例如，以下是你的旧的 `job` 对象
sr <- .job_seurat()
# 尝试更新它
sr <- upd(sr)
```

1.6.6 查看 step 方法的默认参数

step 方法的参数力求精简，一般只保留关键的参数用以控制分析。但这些参数可能会在将来被拓展 (添加额外的参数) 以适应新的分析需求。

可以通过类似以下方式查看默认参数：

R input

```
## 示例：seurat 工作流的 step1 的默认参数
not(.job_seurat())
step1
```

```
## job_seurat:
##     x
##
## -- Methods parameters ------------------------------------------------------------------------
```

R input

```
## 示例：monocle 工作流 step2 的默认参数
not(.job_monocle())
step2
```

```
## job_monocle:
##     x, roots
##
## -- Methods parameters ------------------------------------------------------------------------
```

1.6.7 额外的信息

工作流创建时参考的信息源参考文献信息

R input

```
# 创建一个空的 Seurat 工作流对象 'sr_1'
sr_1 <- .job_seurat()
# 查看方法说明
sr_1@method
```

```
## [1] "The R package `Seurat` used for scRNA-seq processing; `SCSA` (python) used for cell type annota
```

R input

```
# 查看官方网站或信息源网站
sr_1@info
```

```
## [1] "Tutorial: https://satijalab.org/seurat/articles/pbmc3k_tutorial.html"
```

1.6.8 试试一些交叉工具：map, asjob_* .....

map、asjob_* (可能还有 do_*) 这些方法，可能是 Step 系列带来的最大便捷，或者说惊喜。因为它们提供了便捷的接口，实现两种或多种分析或数据之间的快速转换。

基本上，我把我所遇到的需要转换的数据形式都以 map 或 asjob_* 的形式写下来了。

具体的内容，这里不便展开 (因为琐碎)，会在对应的 Step 系列文档中示例说明。

但是你可以试着探索有哪些可用，例如，直接输入：

R input

```
map
```

查看函数体：

R input

```
selectMethod(map, c("job_seurat", "job_seurat"))
selectMethod(map, c("job_seurat", "job_kat"))
```

1.6.9 完成整套分析之后

如果你已经用 Step 系列完成了一整套分析，但是还是不太明白整个过程发生了什么，例如，想知道整个过程到底使用了哪些方法，借鉴了哪些思路，可以参考哪些文献，那么你可以用如下方式查看：

R input

```
auto_method()
```

你可能会注意到类似：[@InferenceAndAJinS2021] 这样的字符，其实，这是参考文献来源的标记，以 pandoc、Latex 等工具自动注释。简而言之，通过它，可以在.bib 文件中找到对应的参考文献。

这一.bib 文件：

- https://github.com/shaman-yellow/utils.tool/blob/master/inst/extdata/library.bib

没错，其实它就存在于 utils.tool 这个 R 包中，当你获取使用它的时候，也就拥有这个.bib 文件了 (它也会很频繁的更新)。

1.7 加入我，和我一起编写新的 Step 系列!

一个编写好的 Step 工作流可以省下许多琐碎的事带来的精力耗费，例如，不需要在各个分析平台切换，而只用 R 语言一路写下代码 (如果你和我一样使用 Rmarkdown，就更能体会它带来的便利了) 。

如果你对此感兴趣，想尝试创建自己的 Step 系列方法，那么你可以在 github 上获取 utils.tool 的源代码，并尝试理解它的框架。

例如，job_seurat 系列方法的源代码：

- https://github.com/shaman-yellow/utils.tool/blob/master/R/workflow_01_seurat.R

整体框架设计 (大多数的接口) ：

- https://github.com/shaman-yellow/utils.tool/blob/master/R/workflow_00.R

1.8 关于安装配置

所有 Step 系列的方法的安装配置会尽可能详细的罗列，但由于笔者 (开发者) 仅在自身的 Linux (Ubuntu 发行版) 系统做过调试，并不确定在其它的机器上会遇到哪些安装的特殊问题。如有疑问，或安装上的困难，请联系：Huang Lichuang (huanglichuang@wie-biotech.com) 。