The Command Pattern in R

Michael Lawrence

August 25, 2014

Contents

1 Introduction

Command pattern is a design pattern used in object-oriented programming, in this design, an object encapsulate commands and all other information needed for a later method call. *Command* pattern based examples include analysis pipeline, parallel processing, GUI action or do/undo.

A typical command pattern including some key features

- *Command* declare an interface for performing/executing operations.
- *ConcreteCoommand* implements the *Command* interface and its own concrete execute method, invoking operations on *Receiver*.
- *Client* creates *ConcreteCommand* object and sets the receiver.
- *Invoker* decides the time a method is called, ask command to perform a request.
- *Receiver* contains methods and knows how to perform operations.

commandr is a R package which use S4 representation to implement *Command* pattern in R. In this package, we follow the essential design of *Command* pattern to implemented a flexible analytical pipeline which including following components.

- Top level virtual class *Command*.
- Class *Pipeline* is a subclass of class *Command*, and a very important concept in this package. A *Pipeline* is composed of a sequence of *Protocols*. A `Pipeline` object is similar to *CommandManager* in *Command* pattern, accept different *Command*(the `Protocol`) in queued order. The *client* in *Command* pattern would be R user either through command line or a GUI.
- A *PipelineData* is also a virtual class, just represent a container for a dataset and an attached pipeline which described how this object generated from a sequence of protocols. For this implementation, it's equally applied to any data set have a slot or attribute called 'pipeline'. A `PipelineData` is the *receiver* in *Command* pattern.

- A *Stage* is an abstract step in a pipeline, represents a role to be played by protocols in a pipeline, each stage transforms one data in a particular way, multiple transformation could be defined in different *Protocols* for each *Stage*. Users could define new types of stages with method `setStage`. When a stage is set, specific input type or output type for this stage could be specified. So that a chained *Pipeline* could be validated which check the input/output data type. In this way, the development could fairly independent as long as we conform to data types for input/output, without worrying about transformation details.

- A *Protocol*(method) object performs a particular *Stage*, and it's a concrete step in a pipeline. Users could set different *Protocol*s by using method `setProtocol`. A *Protocol* implementation would be *ConcreteCommand* in *Command* pattern.

So as described above, a pipeline is composed of many *Stage*s (steps) and each stage coould implement as many *Protocols*(methods) as possible. So you can define (or chain) your pipeline with a sequence of different protocols.

The description in this section may be too abstract to understand the usage easily, so let's go through a very simple example in the following section.

2 Example pipelines

In this section, let's try a simple case, even though, once you get familiar yourself with the usage, it could be extended to more complicated real-world cases. For example, a complicated GC/MS processing pipeline(chroamtoplots) has been successfully built based on implementation of this package. You could explore the data by forming different pipeline and compare the methods you implemented.

Back to our mini-example, let's first set some goals about this pipeline before we move on.

- **Goal**: Processing a vector which may contains numeric values, missing value(NA), remove or replace some missing values and compute averaged summary on the processed data.

- **Stages**:

  - **Stage 1**: Remove missing value.
    - **Protocol 1**: simply remove the missing value.
    - **Protocol 2**: replace missing value with an arbitrary value, by default 0.
    - **Protocol 3**: remove missing value and get numeric value only from an arbitrary range.
  - **Stage 2**: Perform statistics computation(average) on processed data.
    - **Protocol 1**: compute mean.
    - **Protocol 2**: compute median.

- **Form different pipline**

This pipeline is implemented in the following code trunk.

```
> library(commandr)
> ## let's name the first stage 'trim' and specify input types
> setStage("trim", intype = "numeric")

[1] "trim"

> ## a class called 'StageTrim' is defined automatically
> ## then we implemnted protocol 1, 2, 3  for this stage
> setProtocol("remove", fun = function(x){x[!is.na(x)]}, parent = "trim")
```

```
[1] "ProtoTrimRemove"
```

```
> setProtocol("replace", representation = list(val = "numeric"),
+             fun = function(x, val = 0){
+               x[is.na(x)] <- val
+               x
+ }, parent = "trim")
```

```
[1] "ProtoTrimReplace"
```

```
> setProtocol("range", representation = list(low = "numeric", high = "numeric"),
+             fun = function(x, low = 0, high = Inf) x[x >= low & x <= high & !is.na(x)],
+             parent = "trim")
```

```
[1] "ProtoTrimRange"
```

```
> ## let's name the second stage 'average' and specify input types
> setStage("average", intype = "numeric")
```

```
[1] "average"
```

```
> ## implement protocol 1 and 2 for stage 1
> setProtocol("mean", fun = mean, parent = "average")
```

```
[1] "ProtoAverageMean"
```

```
> setProtocol("quantile", representation = list(probs = "numeric"),
+             fun = quantile, parent = "average")
```

```
[1] "ProtoAverageQuantile"
```

```
> d <- c(1, 2, 3, NA,  5, 6, 7)
> ## First pipepine: 1. remove missing value 2. compuate mean
> ## by default, use default protocol for each stage
> p <- Pipeline("trim", "average")
> perform(p, d)
```

```
[1] 4
attr(,"pipeline")
Pipeline with 2 protocol(s):
[1] trim (remove)
[2] average (mean)
```

```
> ## Second pipepine: 1. replcae missing value with 100. 2. compuate mean
> ## make another pipeline easily
> p <- Pipeline(Protocol("trim", "replace", val = 100),
+               "average")
> perform(p, d)
```

```
[1] 17.71429
attr(,"pipeline")
Pipeline with 2 protocol(s):
[1] trim (replace)
[2] average (mean)
```

```
> ## Third pipepine: 1. remove missing value and get value above 2. 2. compuate quantile
> p <- Pipeline(Protocol("trim", "range", low = 2),
+               Protocol("average", "quantile", probs = 0.75),
+               displayName = "Filter and Average")
> perform(p, d)

75%
  6
attr(,"pipeline")
Pipeline with 2 protocol(s):
[1] trim (range)
[2] average (quantile)
```

Now you basically get an idea about how to form pipelines easily and flexibly with defined protocols. So basically, you need to define stages for your pipeline, and then implement as many protocols as you want for each stage. Then construct your pipeline by using constructor `Pipeline`, and pass protocols in the right order. Please pay attention to that

- The default stage protocol method is the one first defined for the stage.
- When use method `Protocol`, make sure they try to use default protocol first unless you pass the protocol name as second argument.

Some accessor are provided for object `Pipeline`.

```
> ## accessor
> inType(p)

[1] "numeric"

> outType(p)

[1] "numeric"

> parameters(p)

[[1]]
[[1]]$low
[1] 2

[[1]]$high
[1] Inf

[[1]]$active
[1] TRUE


[[2]]
[[2]]$probs
[1] 0.75

[[2]]$active
[1] TRUE

> protocol(p, "average")
```

```
average (quantile)
```

```
> displayName(p)
```

```
[1] "Filter and Average"
```

```
> ## find a protocol via protocol name
> findProtocols(p, "average")
```

```
[1] 2
```

It's very **important** to keep in mind that a pipline require your input/output data type match each other in the sequence of protocols. So when define your protocol for each stage, make sure you follow rules made in the stage definition.

`Pipeline` is essentially a *list*, so you can use typical method to subset/re-order pipeline or insert a protocol. But again, keep in mind, an validation step for data input type and output type is always performed when do so. More subseting method are defined via `tail,head`, in a slightly different way which you specify a specific input type or output type to subset the pipeline. More information coudl be find under the manual for function `Pipeline`(run `help('Pipeline')`). Or just get an idea about it by run through following examples.

- **Goal**: We cast a numeric value to character, then to factor, and finally to a list.

```
> # make a new example
> setStage("DemoCastN2C", intype = "numeric", outtype = "character")
```

```
[1] "demoCastN2C"
```

```
> setProtocol("cast", fun = function(x){
+                message("Convert from numeric to character")
+                as.character(x)
+            },
+            parent = "DemoCastN2C")
```

```
[1] "ProtoDemoCastN2CCast"
```

```
> setStage("DemoCastC2F", intype = "character", outtype = "factor")
```

```
[1] "demoCastC2F"
```

```
> setProtocol("cast", fun = function(x){
+                message("Convert from character to factor")
+                as.factor(x)
+            },
+            parent = "DemoCastC2F")
```

```
[1] "ProtoDemoCastC2FCast"
```

```
> setStage("DemoCastF2L", intype = "factor", outtype = "list")
```

```
[1] "demoCastF2L"
```

```
> setProtocol("cast", fun = function(x){
+                message("Convert from factor to list")
+                as.list(x)
+            },
+            parent = "DemoCastF2L")
```

```
[1] "ProtoDemoCastF2LCast"
```

```
> d <- 1:3
> p <- Pipeline(Protocol("DemoCastN2C"),
+               Protocol("DemoCastC2F"),
+               Protocol("DemoCastF2L"))
> p
```

```
Pipeline with 3 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)
[3] demoCastF2L (cast)
```

```
> perform(p, d)
```

```
[[1]]
[1] 1
Levels: 1 2 3

[[2]]
[1] 2
Levels: 1 2 3

[[3]]
[1] 3
Levels: 1 2 3

attr(,"pipeline")
Pipeline with 3 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)
[3] demoCastF2L (cast)
```

```
> # subsetting
> # convert to a factor
> p12 <- p[1:2]
> p12
```

```
Pipeline with 2 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)
```

```
> perform(p12, d)
```

```
[1] 1 2 3
attr(,"pipeline")
Pipeline with 2 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)
Levels: 1 2 3
```

```
> #
> p23 <- pipeline(p, intype = "character")
> p23
```

```
Pipeline with 2 protocol(s):
[1] demoCastC2F (cast)
[2] demoCastF2L (cast)
```

```
> perform(p23, as.character(d))
```

```
[[1]]
[1] 1
Levels: 1 2 3

[[2]]
[1] 2
Levels: 1 2 3

[[3]]
[1] 3
Levels: 1 2 3

attr(,"pipeline")
Pipeline with 2 protocol(s):
[1] demoCastC2F (cast)
[2] demoCastF2L (cast)
```

```
> #
> p12 <- head(p, 2)
> p12
```

```
Pipeline with 2 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)
```

```
> #or
> head(p, outtype = "factor")
```

```
Pipeline with 2 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)
```

```
> head(p, role = "DemoCastC2F")
```

```
Pipeline with 2 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)
```

```
> tail(p, 2)
```

```
Pipeline with 2 protocol(s):
[1] demoCastC2F (cast)
[2] demoCastF2L (cast)
```

```
> tail(p, intype = "character")
```

```
Pipeline with 2 protocol(s):
[1] demoCastC2F (cast)
[2] demoCastF2L (cast)
```

```
> tail(p, intype = "factor")

Pipeline with 3 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)
[3] demoCastF2L (cast)

> tail(p, role = "DemoCastC2F")

Pipeline with 1 protocol(s):
[1] demoCastF2L (cast)

> #combination
> p1 <- Pipeline(Protocol("DemoCastN2C"))
> p2 <- Pipeline(Protocol("DemoCastC2F"))
> p3 <- Pipeline(Protocol("DemoCastF2L"))
> c(p1 ,p2)

Pipeline with 2 protocol(s):
[1] demoCastN2C (cast)
[2] demoCastC2F (cast)

> p[2] <- p2
```

3 sessionInfo

```
> sessionInfo()

R version 3.1.0 (2014-04-10)
Platform: x86_64-apple-darwin10.8.0 (64-bit)

locale:
[1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods
[7] base

other attached packages:
[1] commandr_1.0.1

loaded via a namespace (and not attached):
[1] tools_3.1.0
```