GithubIntroductionNoPNG

November 9, 2019

1 A Very Quick Introduction to Git/Github for Julia Users

Julia's package system and Github are very closely intertwined:

- Julia's package management system (METADATA) is a Github repository
- The packages are hosted as Github repositories
- Julia packages are normally referred to with the ending ".jl"
- Repositories register to become part of the central package management by sending a pull request to METADATA.jl
- The packages can be found / investigated at Github.com
- Julia's error messages are hyperlinks to the page in Github

Because of this, it's very useful for everyone using Julia to know a little bit about Git/Github.

1.1 Git Basics

- Git is a common Version Control System (VCS)
- A project is a **repository** (repos)
- After one makes changes to a project, they **commit** the changes
- Changes are **pulled** to the main repository hosted online
- To download the code, you **clone** the repository
- Instead of editing the main repository, one edits a **branch**
- To get the changes of the main branch in yours, you **fetch**, **pull**, or **rebase**
- One asks the owner of the repository to add their changes via a **pull request**
- Stable versions are cut to **releases**

1.2 Github Basics

- The major online server for git repositories is Github
- Github is a free service
- Anyone can get a Github account
- The code is hosted online, free for everyone to view
- Users can open **Issues** to ask for features and give bug reports to developers
- Many projects are brought together into **organizations** (JuliaMath, JuliaDiffEq, JuliaStats, etc.)

An example Github repository for a Julia package is is DifferentialEquations.jl: https://github.com/JuliaDiffEq/DifferentialEquations.jl

1.3 Using Julia's Package Manager

1.3.1 Adding a Package

Julia's package manager functions are mirror the Git functions. Julia's package system is similar to R/Python in that a large number of packages are freely available. You search for them in places like Julia's Package Genie, or from the Julia Package Listing. Let's take a look at the Plots.jl package by Tom Breloff. To add a package, use `Pkg.add`

```
In [ ]: Pkg.update() # You may need to update your local packages first
        Pkg.add("Plots")
```

This will install the package to your local system. However, this will only work for registered packages. To add a non-registered package, go to the Github repository to find the clone URL and use `Pkg.clone`. For example, to install the `ParameterizedFunctions` package, we can use:

```
In [ ]: Pkg.clone("https://github.com/JuliaDiffEq/ParameterizedFunctions.jl")
```

1.3.2 Importing a Package

To use a package, you have to import the package. The `import` statement will import the package without exporting the functions to the namespace. (Note that the first time a package is run, it will precompile a lot of the functionality.) For example:

```
In [5]: import Plots
        Plots.plot(rand(4,4))
```

1.3.3 Exporting Functionality

To instead export the functions (of the developers choosing) to the namespace, we can use the `using` statement. Since Plots.jl exports the `plot` command, we can then use it without reference to the package that it came from:

```
In [6]: using Plots
        plot(rand(4,4))
```

What really makes this possible in Julia but not something like Python is that namespace clashes are usually avoided by multiple dispatch. Most packages will define their own types in order to use dispatches, and so when they export the functionality, the methods are only for their own types and thus do not clash with other packages. Therefore it's common in Julia for concise syntax like `plot` to be part of packages, all without fear of clashing.

1.4 Getting on the Latest Version

Since Julia is currently under lots of development, you may wish to checkout newer versions. By default, `Pkg.add` is the "latest release", meaning the latest tagged version. However, the main version shown in the Github repository is usually the "master" branch. It's good development practice that the latest release is kept "stable", while the "master" branch is kept "working", and development takes place in another branch (many times labelled "dev"). You can choose which branch your local repository takes from. For example, to checkout the master branch, we can use:

```
In [ ]: Pkg.checkout("Plots")
```

This will usually gives us pretty up to date features (if you are using a "unreleased version of Julia" like building from the source of the Julia nightly, you may need to checkout master in order to get some packages working). However, to go to a specific branch we can give the branch as another argument:

```
In [ ]: Pkg.checkout("Plots","dev")
```

This is not advised if you don't know what you're doing (i.e. talk to the developer or read the pull requests (PR)), but this is common if you talk to a developer and they say "yes, I already implemented that. Checkout the dev branch and use `plot(...)`).