Git

Basi, workflow e concetti avanzati (pt3)

Andrea Fornaia, Ph.D.
Department of Mathematics and Computer Science
University of Catania
Viale A.Doria, 6 - 95125 Catania Italy
fornaia@dmi.unict.it
http://www.cs.unict.it/~fornaia/
Advanced Topics
Fast Forward Merge

- Using FF you lose any reference in the history about the branch!
- Even if FF is possible, **you may avoid it**
  - commit history will be a record of what actually happened
- FF **makes the history linear** when is possible
- To force linear history, use **rebase** instead of merge
Rebase vs Merge

C1 → C2 → C3
  ↓  ↓  ↓
C4 ← C5 ← master

MERGE

C1 → C2 → C3 → C4 → C5
  ↓  ↓  ↓  ↓
master ↑ ↑ ↑ ↑ feature

REBASE

C1 → C2 → C3
  ↓  ↓  ↓
C4 ← C5 ← feature

C6 → C4 → C5
  ↓  ↓
master ↑ ↑

C6 → C4 → C5
  ↓  ↓
feature ↑ ↑
Rebase vs Merge

**MERGE**

```markdown
- git checkout master
- git merge feature
- git branch -d feature (optional)
```

**PRO**
commit history is a record of what actually happened

**CONS**
Non-linear history can be hard to understand

**REBASE**

```markdown
- git checkout feature
- git rebase master
  (move feature commits on top of master)
- git checkout master
- git merge feature (is ff)
- git branch -d feature (optional)
```

**PRO**
Linear and clean history

**CONS**
changing the commit history you’re *lying* about what actually happened

Take the best from both:
rebase local changes before pushing them, to clean up your history but never rebase anything you’ve pushed somewhere
Rebase Conflicts

error: could not apply fa39187... something to add to patch A

When you have resolved this problem, run "git rebase --continue". If you prefer to skip this patch, run "git rebase --skip" instead. To check out the original branch and stop rebasing, run "git rebase --abort". Could not apply fa39187f3c3dfd2ab5faa38ac01cf3de7ce2e841... Change fake file

• You can run **git rebase --abort** to completely undo the rebase.
  – Git will return you to your branch's state as it was before git rebase was called.
• You can run **git rebase --skip** to completely skip the commit.
  – None of the changes of by the problematic commit will be included.
  – It is very rare that you would choose this option (**don't do it!**)
• You can fix the conflict.

[https://help.github.com/articles/resolving-merge-conflicts-after-a-git-rebase/]
Stashing

• When you want to **switch branches**
• But you **don’t want to commit** what you’ve been working on yet
  – You can’t change branch if changes couldn’t be applied without conflicts with the new branch

• Takes the **dirty state** of your working directory
  – **staged** changes (index)
  – modified **tracked** files (working dir)
• Saves it on a **stack of unfinished changes** that you can reapply at any time

Stashing

$ git status
On branch master
Changes to be committed: new file: bar.txt
Changes not staged for commit: modified: foo.txt
Untracked files: baz.txt

$ git stash
Saved working directory and index state WIP on master: e0ad866 C1
HEAD is now at e0ad866 C1

$ git status
On branch master
Untracked files: baz.txt

$ git stash list
stash@{0}: WIP on master: e0ad866 C1

... do something else ...

$ git stash pop
(index and working dir status restored; stash dropped)
Detached HEAD

- With "git checkout" you determine which revision of your project you want to work on.
- Normally, you use a **branch name** to communicate with "git checkout" (if none, **HEAD** is assumed).
- This **move also HEAD** to point the specific branch.
- When a **specific commit is checked out instead of a branch** you will have a "detached HEAD": HEAD is not pointing to branch!

```
$ git graph
* 76d9801 (HEAD, master) C3
* f7c2337 C2
* 1026c96 C1

$ git checkout f7c2337
Note: checking out 'f7c2337'.
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.
...```
If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

```bash
git checkout -b new_branch_name
```

```bash
$ git graph
* 76d9801 (master) C3
* f7c2337 (HEAD) C2
* 1026c96 C1

$ touch C4.txt && git add C4.txt && git commit -m C4

$ git graph -all
* 024c5e5 (HEAD) C4
  * 76d9801 (master) C3
  /
* f7c2337 c2
* 1026c96 c1
Detached HEAD

Option 1: SAVE CHANGES

$ git checkout -b mod
Switched to a new branch 'mod'

$ git graph --all
* 024c5e5 (HEAD, mod) C4
| * 76d9801 (master) C3
|/
* f7c2337 c2
* 1026c96 c1

git reflog

Usefull to recover the name (SHA1) of a recently deleted commit! (024c5e5)

It shows the history of all your pointer operations (checkout, branch...)

Option 2: DISCARD CHANGES

$ git checkout master
Warning: you are leaving 1 commit behind, not connected to any of your branches: 024c5e5 C4

If you want to keep them by creating a new branch, this may be a good time to do so with:

git branch new_branch_name 024c5e5

Switched to a new branch 'master'

$ git graph --all
* 76d9801 (HEAD, master) C3
* f7c2337 c2
* 1026c96 c1
$ git reflog
76d9801 HEAD@{0}: checkout: moving from
024c5e5e57aff2d53531dc0e1c58dc7a34d2807f to master
024c5e5 HEAD@{1}: commit: C4
f7c2337 HEAD@{2}: checkout: moving from master to f7c2337
76d9801 HEAD@{3}: commit: C3
f7c2337 HEAD@{4}: commit: C2
1026c96 HEAD@{5}: commit (initial): C1

$ git branch mod 024c5e57aff2d53531dc0e1c58dc7a34d2807f
$ git graph --all
* 024c5e5 (mod) C4
| * 76d9801 (HEAD, master) C3
|/
* f7c2337 C2
* 1026c96 C1
Reset vs Checkout

• `git checkout HEAD -- <path>`
  revert the dir/file content from *working tree* to a specific commit version
  – you can use `<commit>` or `<branch>` instead of HEAD

• `git reset HEAD -- <path>`
  remove file/dir from *index tree* (staging area)
  – used to unstage modifications
**Reset vs Checkout**

- **git checkout `<commit>`** modifies the working tree content to a specific commit (typically a branch) and updates HEAD.

- **git reset `<commit>`** moves the current branch pointer to the given commit (HEAD will be also updated).

```
$ git graph --all
* 024c5e5 (mod) C4
  * 76d9801 (HEAD, master) C3
  /
* f7c2337 C2
* 1026c96 C1
```

```
$ git checkout mod
$ git graph --all
* 024c5e5 (HEAD, mod) C4
  * 76d9801 (master) C3
  /
* f7c2337 C2
* 1026c96 C1
```

```
$ git reset f7c2337
$ git graph --all
* 76d9801 (master) C3
* f7c2337 (HEAD, mod) C2
* 1026c96 C1
```

```
$ git status
C4.txt untracked
```
GitHub
• A home for free public git repositories
• Interface for exploring git repositories
• Real open source
  – immediate, easy access to the code
• Fork it, try it, learn it
• Social Coding
Why use GitHub

- It takes care of the **server aspects of git**
- Web interface for git
  - Exploring code and its history
  - Tracking issues
  - Pull Requests
  - Much more...
- Facilitates:
  - Learning from others
  - Seeing what people are up to
  - Contributing to others' code
- Lowers the barrier to collaboration
  - "There's a typo in your documentation" vs.
  - "Here's a correction for your documentation"
- **Improve your CV** contributing to open-source projects
Cloning Existing Project

Design patterns implemented in Java [http://java-design-patterns.com](http://java-design-patterns.com)
Forking Projects

• Contribute to an existing project to which you don’t have push access

• GitHub will make a copy of the project that is entirely yours

• No need to add users to projects as collaborators to give them push access

• People can push their changes back to the original repository by Pull Requests (PR)
1. Create a **topic branch** from master
2. Make some commits to **improve** the project
3. Open a **Pull Request**
4. Developers **discuss**, and optionally continue committing
5. Then **pull** and test with modifications
6. Time to **merge** feature branch to master
• **Requests** for contribution:
  – Bugfix
  – Enhancement
  – New Feature
• It opens an **issue discussion**
• A **good way to start** with GitHub open-source projects:
  – fork the project
  – work on one of the issues
  – propose your solution with a Pull Request

[https://github.com mauricioaniche repodriller]
Issue Example

README could give links to competitors with explanation of the differences #89

Closed davisjam opened this issue on Oct 11, 2017 · 1 comment

davisjam commented on Oct 11, 2017

e.g. Boa, Alitehia Core, etc.

davisjam added a commit to davisjam/repodriller that referenced this issue on Oct 18, 2017

davisjam referenced this issue on Oct 18, 2017

Add links to other options in README #99

mauricioaniche added a commit that referenced this issue on Oct 19, 2017

Add links to other options in README (#99)  

[https://github.com/mauricioaniche/repodriller/issues/89]
Pull Request Example

Add links to other options in README #99

[https://github.com/mauricioaniche/repodriller/pull/99]
PR and Code Coverage

• Code coverage (test coverage):
  – percentage of source code executed by an automated test suite (hits / total_lines)
  – a code with high coverage has lesser chances of containing undetected bugs

• Include coverage report into the CI pipeline:
  – focus on sound integrations
    • refine unit tests to increase coverage before merge
  – promote healthy pull requests
    • where new features and bug fixes commonly occur
How does it work

• **Source code instrumentation:**  
  – adds instrumentation statements to the source code and use a normal compiler to produce an instrumented application

• **Intermediate code instrumentation:**  
  – adds instrumentation bytecode to the compiled files, generating a new instrumented application

• **Runtime information collection:**  
  – collects information from the runtime environment during (non-instrumented) code execution to determine coverage
CodeCov

- Code coverage tool
  - coverage report
  - supports different languages
- Easy to integrate:
  - GitHub
  - Travis CI
- Check coverage for:
  - commits
  - branches
  - files and folders
  - pull requests

[https://codecov.io]
import Foundation

class MoviesManager: ListDisplayableDataProvider {
    private var cachedItems: [ListDisplayable]?
    var moviesDataProvider: MoviesDataProvider

    init(withDataProvider dataProvider: MoviesDataProvider = MoviesDataSource()) {
        self.moviesDataProvider = dataProvider
    }

    func getListItem(onCompleted: ([ListDisplayable]) -> ())?
    { self.moviesDataProvider.getMovies {
        (movies) in
        let listItems = movies.map({ ListDisplayableItem(withMovieTitle: $0) })
        DispatchQueue.main.async(execute: {
            onCompleted?(listItems)
        })
        self.cachedItems = listItems
    }

    func searchListItems(searchTerm: String, onCompleted: ([ListDisplayable]) -> ())? {
        if let cached = cachedItems {
            onCompleted?(self.filter(cached, term: searchTerm))
        } else {
            getListItem(onCompleted: { (items) in
                onCompleted?(self.filter(items, term: searchTerm))
            })
        }
    }

    private func filter(_ list: [ListDisplayable], term: String) -> [ListDisplayables] {
        return list.filter({ $0.listTitle.lowercased().contains(term.lowercased()) })
    }

    // MARK: ListDisplayable
    private struct ListDisplayableItem: ListDisplayable {
        var listTitle: String
        var listSubtitle: String?

        init(withMovieItem item: MovieItem) {
            self.listTitle = item.title
            self.listSubtitle = item.movieDescription
        }
    }
}
Fix your code to pass unit tests

Fix your unit tests to pass your code
Question Time? Are you sure?

ONE DOES NOT SIMPLY ASK A GIT QUESTION
References

• K. Broman and S.G. Younkin: A brief introduction to git & GitHub. [https://www.biostat.wisc.edu/~kbroman/talks/GitPrimer.pdf]
• V. Driessen: A successful Git branching model. [http://nvie.com/posts/a-successful-git-branching-model/]
• Understanding the GitHub Flow. [https://guides.github.com/introduction/flow/]
• R. E. Silverman: Git Pocket Guide: A Working Introduction
• Parth Shandilya: Integrating Travis CI and Codecov into a Python-based Project. [https://hackernoon.com/integrating-travis-ci-and-codecov-into-a-python-based-project-6f658074ff63]