Time Travel with Git, Uncommitting The Future

ITRabbitX

2756-12-22

Contents

Greetings, fellow code-navigators and time travelers! I send you these words through the intricate weavings of time and space from a world where technological marvels have made even the most tricky coding errors a distant memory. But I remember a time when things were not so streamlined – when repositories were chaotic, code was frequently committed, and unwanted files made our local repositories appear messy. Ah, the disarray!

Now, I could regale you with stories of the times when we had no choice but to discard whole repositories and start afresh, all because we couldn't neatly present our work. But why dwell on such disorganized times? Instead, I bring you a beacon of hope, a ray of light – a skill that, if mastered, could rewrite the very fabric of your project's history.

What is this wondrous skill, you ask? It's the art of Git time-travel, my friends! A method to mend the disarray in your code's timeline, to revert, to rebase, and to squash. With this in your toolkit, you will possess the power to tidy up the future by organizing the past.

Today, we embark on a daring mission! A time-bending adventure to rearrange an all-too-common issue. Unorganized, frequent commits to your repository during production processes. An action that, once done, may seem irreversible, leading us to a future filled with messy, unprofessional repositories.

Fear not, fellow voyagers! By manipulating the strands of time (and by that, I mean using handy Git commands), we will journey back, we will undo the mistakes of our past selves, and we will rewrite a future where our repositories are tidy, organized, and ready to be showcased in a public repository.

So gear up, brave coders! The future of well-organized code repositories awaits! But first, we must journey into the past. It's time to delve into the exciting

world of time-travel with Git.

The Unruly Repository

In our tale, the challenge we face can be equated to the chaos during the production stage of a project. This might seem minor, yet it contributes to a cluttered and untidy repository.

Task for the Time-Travel Initiate: Embark on the construction of a mock project, incorporating some messy commits, as you do when working on projects. Consider this your initiation into the world of 'temporal correction missions', a preparatory step that stages the ground for your forthcoming tidying expedition.

1. Create a new directory for your mock project

   ```
   mkdir git-time-travel
   cd git-time-travel
   ```

2. Initialize a new git repo

   ```
   git init
   ```

3. Create a few files

   ```
   echo "file_1" > file1.txt
   echo "file_2" > file2.txt
   echo "file_3" > file3.txt
   echo "file_4" > file4.txt
   ```

4. Commit the files

   ```
   git add file1.txt
   git commit -m "Add file1"
   git add file2.txt
   git commit -m "Add file2"
   git add file3.txt
   git commit -m "Add file3"
   git add file4.txt
   git commit -m "Add file4"
   ```

In the next part of our story, you will embark on a deeper exploration of this repository chaos. Be prepared to tidy up your mistakes, armed with the knowledge and tools we shall equip you with. Steady your nerves, for you are about to take a plunge into the intricate fabric of time.

Crafting the Repository Organizer

This chapter unfolds a comprehensive overview of **git checkout**, and how to hop between commits. This command is the toolkit of your repository organization - they give you the power to navigate through your code's timeline, helping you manage each file and commit in a structured and tidy manner.

Task for the Temporal Wayfinder: Embark on a mission of creating a series of additional commits within your mock project. Experiment with the versatility

of **git checkout** as it allows you to effortlessly hop between them, like an organizer setting the order of a series of documents.

1. Create a few text files and commit them separately.

```
echo "Hello, World!" > hello.txt
git add hello.txt
git commit -m "Add hello.txt"

echo "This is a repository organization exercise." > repo-organize.txt
git add repo-organize.txt
git commit -m "Add repo-organize.txt"
```

2. Move between your commits using **git checkout <commit hash>**. The actual hashes of your commits can be found with **git log**.

As we conclude this part of the story, you have now forged your repository organizing toolkit, ready for your first expedition to the past. Brace yourself for an adventure unlike any other, as we journey to restructure and tidy up the past.

Voyage Through the Commit History

In this interesting section, we shall traverse through the commit history using **git log** to analyze previous commits. This can be compared to a careful examination of a series of documents in a filing cabinet. We shall then familiarize ourselves with **git rebase** in its interactive mode, an essential maneuver, akin to rearranging the sequence of our documents for a better organization.

Task for the Time Navigator: Utilize the **git log** command to pinpoint the commit made before the addition of the excess commit or commits. Once identified, employ **git rebase** to retrace your steps, navigating back in time to that specific commit. With these steps, you are delving deep into the layers of the past, all set to tidy up a messy local repository.

1. Use **git log** to find the commit hash before the commit was added.
2. Use **git rebase -i <commit-hash>** to start an interactive rebase to the commit before the extra commit was added.

As this part of our story draws to a close, you stand at the edge of the past, the mess within your reach. Next we dive into the heart of our repository organization mission: The Great Cleanup. Brace yourself for this critical task.

The Great Cleanup

In this segment, we delve into the process of streamlining a cluttered repository, one that included an unnecessary and excess commit. Before embarking on this endeavor, however, we must ensure that any data we need is secured and stowed away safely. Consider this preliminary step as an important assignment: maintaining the integrity of useful data before The Great Cleanup.

Task for the Time Travel Data Protection Agent: Prepare your command line tools to relocate the unnecessary file to a different location. Following this

successful operation, with our data preserved, we are prepared to start a total removal of the excess commits from our timeline.

1. Copy the file to a safe location (where you can later find it)

```
cp file1.txt ~/backup_file1.txt
```

Congratulations, the file now resides in a safe haven.

Task for the Experienced Time Cleanup Agent: Use your command line skills to remove the extra commit from the repository.

Our next endeavor aims to eradicate these remaining parts, leaving not a single trace within the commit history. This necessitates squashing the commit wherein the file was added, joined with the commit where it was removed, essentially neutralizing their combined effect.

2. Remove the extra commit/file and commit the removal

```
git rm file1.txt
git commit -m "Remove excess file1"
```

The commit has effectively been removed from the repository, although the traces of its existence still linger.

Task for the Time Travel Commit Historian: Travel back through the commit history to the commit preceding the addition of the extra commit. Within the text editor that appears, find the lines symbolizing the commit that added the excess file and its subsequent removal. Replace **'pick'** with **'squash'** for the line representing the 'Remove excess commit' commit, save, and then close the document. Finally, in the new text editor space that appears, write a commit message that describes the newly squashed commit.

3. Start an interactive rebase to the commit before the extra commit was added

```
git rebase -i <commit-hash>
```

Having reached this point in our mission, we have successfully restructured the repository – the unneeded commit, and therefore messy commit histories, were never a part of the repository. As if the inclusion of the commit was a mere figment of imagination. However, our journey through the commit history isn't at its end yet - there are still proactive measures we can undertake to improve the structure and clarity of our repository.

The next adventure calls as your repository organizing journey continues.

The Art of Repository Recomposition

In this chapter we delve into the finer details of using **git commit** and **git push** to commit and dispatch your adjustments, painting a clear image of a careful organizer restructuring a project's repository. We will also highlight the importance of keeping unnecessary files out of your project using a .gitignore file, an action comparable to erecting a well-defined boundary around your workspace.

Task for the Time-Mending Artist: Initiate the transformation of your repository by committing and pushing your adjustments. Next, you may have some files you wish git to ignore. Create a .gitignore file and add the file to it, effectively ensuring it will stay out of your repository in the future. Conclude this stage by committing and pushing this change. Through these maneuvers, you streamline your project's repository, forging a clear and concise workspace.

1. Add surplus-file.txt to a .gitignore file and commit the change:

```
echo "surplus_file.txt" > .gitignore
git add .gitignore
git commit -m "Ignore surplus file"
```

2. Push your changes to the remote repository:

```
git push
```

At the end of this story, you've not only mastered the art of repository reorganization but also established a mess-free environment for the future. With these tools at your disposal, you are well-equipped to keep any project's repository well-structured and reader-friendly. However, our journey isn't over yet. Hold onto your tools as we venture further.

The Final Adventure: A Repository Polished

As we conclude this captivating expedition through the realms of git manipulation, let us pause and reflect on the lessons etched in our journey. We ventured back in the commit history to rectify a past disarray, and in the process, changed the course of the future of our project.

The consequences of our initial lack of organization serve as a stark reminder to remain vigilant in our future endeavors. Each commit we make should be carefully reviewed, each push meticulously checked. The clarity and maintainability of our repository (or in this case, our project's structure) depends on it. With the skills acquired from this journey, it is my hope that we have empowered you to keep your repositories well-organized and reader-friendly in the future.

This brings our repository-cleaning expedition to an end, but remember, every ending paves the way for a new beginning. We have a few more git techniques to explore in the bonus adventure...

Bonus Adventure: Mastering the Organizational Arts

For those who seek to delve deeper into the mysteries of Git, we cover additional advanced techniques that provide even more control over your project's organization. Consider these maneuvers as expert-level repository management, deployed under special circumstances to deftly navigate the maze of code.

Git Cherry-pick: Consider this as your ability to pluck a specific event from the history, irrespective of its chronological place. With **git cherry−pick**, you can apply changes from any commit onto your current branch, useful when specific changes need to be extracted from the commit history.

Git Bisect: This tool allows you to locate the exact moment in time when things went awry. It's akin to a time travel detective's investigation tool, pinpointing the precise commit that introduced an issue.

Git Stash: Imagine having the power to temporarily set aside some changes, perform other tasks, and then reintroduce those changes back into the project. **git stash** is your temporary pocket to hold changes, freeing you to work on different branches without losing your progress.

Remember, these are powerful tools, and with great power comes great responsibility. Use them wisely to navigate your project's history, and you'll find that the control you have over the past, present, and future of your project can make all the difference in your world of code.

Fast travelling through time with Git (Steps only)

In case you need the answer fast, here are the relevant commands for git. Please be aware that this is intended for before you have pushed something to your public repostory, and remember to replace the example file names with yours, if needed.

1. Use *git log* to find the commit hash before the surplus file was added:

   ```
   git log
   ```

2. Use *git rebase* to start an interactive rebase to the commit before the unneeded file was added:

   ```
   git rebase -i <commit-hash>
   ```

3. Copy the unneeded file(or whatever information you wish to keep) to a safe location:

   ```
   cp file1.txt ~/backup_file1.txt
   ```

4. Remove the unneeded file and commit the removal:

   ```
   git rm file1.txt
   git commit -m "Remove excess file"
   ```

5. Start an interactive rebase to the commit before the surplus file was added:

   ```
   git rebase -i <commit-hash>
   ```

Step 6 is optional

6. Add extra_file.txt to a .gitignore file and commit the change:

   ```
   echo "extra_file.txt" > .gitignore
   git add .gitignore
   git commit -m "Ignore extra file"
   ```

7. Push your changes to your remote repository:

   ```
   git push
   ```