

9 Syncing With a Remote Written by Bhagat Singh & Chris Belanger

Up to this point in the book, you’ve worked pretty much exclusively on your local system, which isn’t to say that’s a bad thing — having a Git repository on your local machine can support a healthy development workflow, even when you are working by yourself. But where Git really shines is in managing distributed, concurrent development, and that’s what this chapter is all about. You’ve done lots of great work on your machine, and now it’s time to push it back to your remote repository and synchronize what you’ve done with what’s on the server. And there’s lots of reasons to have a remote repository somewhere, even if you are working on your own. If you ever need to restore your development environment, such as after a hard drive failure, or simply setting up another development machine, then all you have to do is clone your remote repository to your clean machine. And just because you’re working on your own now doesn’t mean that you won’t always want to maintain this codebase yourself. Down the road, you may want another maintainer for your project, or you may want to fully open-source your code. Having a remote hosted repository makes doing that trivial.

Pushing your changes

So many things in Git, as in life, depends on your perspective. Git has perspective standards when synchronizing local repositories with remote ones: **Pushing** is the act of taking your local changes and putting them up on the server, while **pulling** is the act of pulling any changes on the server into your local cloned repository. So you’re ready to push your changes, and that brings you to your next Git command, handily named `git push`.

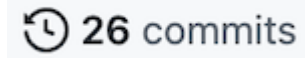
Execute the following command to push your changes up to the server:

```
git push origin main
```


This tells Git to take the changes from the `main` branch and synchronize the remote repository (`origin`) with your changes. You’ll see output similar to the following:

```
Enumerating objects: 50, done.
Counting objects: 100% (46/46), done.
Delta compression using up to 16 threads
Compressing objects: 100% (31/31), done.
Writing objects: 100% (36/36), 3.39 KiB | 579.00 KiB/s, done.
Total 36 (delta 17), reused 2 (delta 1), pack-reused 0
remote: Resolving deltas: 100% (17/17), completed with 4 local objects.
To https://www.github.com/belangerc/ideas.git
 c470849..f5c54f0  main -> main
```

Git’s given you a lot of output in this message, but essentially it’s telling you some high-level information about what it’s done, here: It’s synchronized 17 changed items from your local repository on the remote repository. **Note:** Wondering why Git didn’t prompt you for a commit message, here? That’s because a push is not really *committing* anything; what you’re doing is asking Git to take your changes and synchronize them onto the remote repository. You’re combining your commits with those already on the remote, not creating a new commit on top of what’s already on the remote. Want to see the effect of your changes? Head over to the URL for your repository on GitHub. If you’ve forgotten what that is, you can find it in the output of your `git push` command. In my case, it’s `https://www.github.com/belangerc/ideas`, but yours will have a different username in there. Once there, click the **26 commits** link near the top of your page:



You’ll be taken to a list of all of your synchronized changes in your remote repository, and you should recognize the commits that you’ve made in your local repository:

 **belangerc / ideas**

forked from [raywenderlich/ideas](#)

<> Code

Pull requests

Actions

Projects

Wiki

Security


Insights

Settings

main

Commits on Oct 20, 2020


Merge branch 'contact-details'

 **crispy8888** committed 13 minutes ago

a4eded5

<>


Adding README contact information

 **crispy8888** committed 13 minutes ago

37f4dc7

<>


Adding more detail to the README file

 **crispy8888** committed 18 minutes ago

23a1d9c

<>

Merge branch 'clickbait' into


 **crispy8888** committed 21 minutes ago

3ff6fbe

<>

Commits on Oct 18, 2020


Adding .gitignore files and HTML

 **crispy8888** committed 2 days ago

ac21935

<>


Adds all the good ideas about management

 **crispy8888** committed 2 days ago

3113a85

<>


Removes terrible live streaming ideas

 **crispy8888** committed 2 days ago

600f9d0

<>


Moves platform ideas to website directory

 **crispy8888** committed 2 days ago

21fcb3e

<>

Updates book ideas for Sumbian and MOS 6510

 **crispy8888** committed 2 days ago

7000000

<>

That's one half of the synchronization dance. And the yin to `git push`'s yang is, unsurprisingly, `git pull`.

Pulling changes

Pulling changes is pretty much the reverse scenario of pushing; Git takes the commits on the remote repo, and it integrates them all with your local commits.


That operation is pretty straightforward when you're working by yourself on a project; you pull the latest changes from the repository, and, most likely, the remote will always be synchronized with your local, since there's no one else but you to make any changes.

But the more common scenario is that you'll be working with others in the same repository, and they will be their own pushing changes to the repository. So most of the time, you won't have the luxury of pushing your changes onto an untouched repository, and you'll have to integrate the changes on the remote by pulling them into your repository before you can push your local changes.

To illustrate how this works, and to illustrate what `git pull` actually does to your repository, you'll simulate a scenario wherein someone else has made a change to the `main` branch and pushed their changes before you had a chance to push yours. You'll see how Git responds to this scenario, and you'll learn the steps required to solve this issue see how to solve this issue.

Moving the remote ahead

First, you have to simulate someone else making a change on the remote. Navigate to the main page on GitHub for your repository: <https://github.com/<username>/ideas>. Once there, click on the **tutorials** directory link of your project, and then click on **tutorial_ideas.md** to view it in your browser.

 **crispy8888 / ideas**

forked from [raywenderlich/ideas](#)

<> Code

Pull requests

Projects

Wiki

Security

Insights


Settings

Branch: master

ideas / tutorials / tutorial_ideas.md

Find file

Copy path

 **crispy8888** Adding PalmOS

2b7e8b1 42 minutes ago

1 contributor

5 lines (4 sloc) | 80 Bytes

Raw

Blame

History

📄


✏️

🗑️

Tutorial Ideas

[] Mastering PalmOS [] Mastering PalmOS [] Mastering PalmOS

Click the **edit** icon on the page (the little pencil icon), and GitHub will open a basic editor for you.

 **crispy8888 / ideas**

forked from [raywenderlich/ideas](#)

<> Code

Pull requests

Projects

Wiki

Security

Insights

Settings

ideas / tutorials /

tutorial_ideas.md

Cancel

<> Edit file

Preview changes

Spaces

2

Soft wrap

```
1 # Tutorial Ideas
2 [ ] Mastering PalmOS
3 [ ] Getting the most out of your TRS-80
4 [ ] Bridging SwiftUI and Visual Basic
5 |
```

Add the following idea to **tutorial_ideas.md** in the editor:
[] Blockchains with BASIC

Then, scroll down to the **Commit changes** section below the editor, add a commit message of your choice in the first field of that section, leave the radio button selection as **Commit directly to the main branch**, and click **Commit changes**. This creates a new commit on top of the existing `main` branch on the remote repository, just as if someone else on your development team had pushed the commits from their local system.

Now, you'll create a change to a different file in your **local** repository.

Return to your terminal program, and edit **books/book_ideas.md** and add the following line to the bottom of the file:

- [] Debugging with the Grace Hopper Method

Save your changes and exit.

```
Stage the change:
git add books/book_ideas.md
```

```
Now, create a commit on your local repository:
git commit -m "Adding debugging book idea"
```

You now have a commit on the head of your local main branch, and you also have a different commit on the head of your remote main branch. Now you want to push this change up to the remote. Well, that’s easy. Just execute the `git push` command as you normally would:

```
git push origin main
```

```
Git balks, and returns the following information to you:
! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://www.github.com/belangerc/ideas'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Well, that didn’t work as expected. Git is quite helpful sometimes in the hints it gives; in this case, it’s telling you that it detected changes on the remote that you don’t have locally. Since you’d probably want to make sure that your local changes meshed properly with the changes on the remote before you push, you’ll want to pull those changes down to your local system. Execute the following to pull the changes from the remote into your local:

```
git pull origin
```

```
Oh, heck, Git has opened up Vim, which means that it's creating a commit; in this case, it's creating a merge commit. Why, Git, why?
Merge branch 'main' of https://github.com/belangerc/ideas into main
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

You’ll explore what Git is doing shortly, but finish this commit first and let Git get on with whatever it’s doing. Git has already auto-created a commit message for you, so you might as well accept that and try and figure this mess out later. Press **z**, then type **wq** and then press **Enter** to save this commit message and exit out of Vim. You’re taken back to the command prompt, so execute the following to see what Git has done for you:

```
git log --oneline --graph
```

```
You'll see something similar to the following:
*   b495cc8 (HEAD -> main) Merge branch 'main' of https://github.com/belangerc/ideas
|\
| * 35054cc (origin/main, origin/HEAD) Update tutorial_ideas.md
* | 8648645 Adding debugging book idea
|/
*   a4eded5 Merge branch 'contact-details'
.
.
.
```

Note: Wondering what those asterisks (*) mean in the graphical representation of your tree? Since commits from different branches are shown stacked one on top of the other, the asterisks simply show you on which branch this commit was made. In this case, you can see the book idea was committed on one branch (your local main branch), and the other commit was created on the remote origin branch. Working up the tree, you have a common ancestor of `a4eded5 Merge branch 'contact-details'`. Then you have commit `8648645`, which is the commit you made on your local repository, followed by `35054cc`, your remote commit on the GitHub repository page. And *also*, there’s this `b495cc8 Merge branch 'main'` stuff at the top. And *also also*, Git shows your remote “Update tutorial_ideas.md” on a branch. But you didn’t create a branch. You chose the option on the GitHub edit page to commit directly to main. Where did that come from? **Note:** It’s seemingly simple scenarios like this — non-conflicting changes to distinct files resulting in a merge commit — that causes newcomers to Git to throw up their hands and say, “What the heck, Git?” This is why learning Git on the command line can be instructive, as opposed to using a Git GUI client that hides details like this. Seeing what Git is doing under the hood, and, more importantly, understanding *why*, is what will help you navigate these types of scenarios like a pro. To understand what Git’s doing, you need to dissect the `git pull` command first, since `git pull` is not one, but *two* commands in disguise. Press **Q** to exit out of the git log viewer.

First step: Git fetch

`git pull` is really *two* commands in one: `git fetch`, followed by `git merge`. You haven’t run across `git fetch` yet. Fetching updates your local repository’s hidden **.git** directory with all of the commits for this repository, both local and remote. Then, Git can figure out what to do with what it’s fetched from the remote; maybe it can fast-forward merge it, maybe it can’t, or maybe there’s a conflict preventing Git from going any further until you fix the conflict. Generally, it’s a good idea to execute `git fetch` before pushing your changes to the remote, if you suspect that someone else may have been committing changes to that same particular branch on the remote, and you want to check out what they’ve done before you integrate it with your work. When Git fetches the remote commits and brings them down to your local system, it creates a temporary reference to the tip of the remote repository’s branch. Think back to when you explored a little of the Git internal file structure, and you found the file **.git/refs/heads/main** that simply contained a reference to the hash of the commit that was at the tip of the current branch (i.e., HEAD).

You can see this reference in your own local hidden **.git** directory. Execute the following command:

```
ls .git
```

In the results, you should see a file named **FETCH_HEAD**. That’s the temporary reference to the tip of your remote branches. Want to see what’s inside? Sure thing! Execute the following command to see the contents of **FETCH_HEAD**:

```
cat .git/FETCH_HEAD
```

You’ll see a hash, along with a note of where this commit came from. In my case, I see the following at the top of that file:

```
8909ec5feb674be351d99f19c51a6981930ba285      branch 'main' of https://github.com/belangerc/ideas
```

Second step: Git merge

So once Git has fetched all of the commits to your local system, you’re essentially in a position in which you have a commit from one source — your local commit — that Git needs to combine with another commit: the remote commit. Sounds like merging a branch, doesn’t it? In fact, that’s pretty much how Git views the situation. Take a look back at the state of the repository graph before you merged, reproduced here:

```
| * 35054cc (origin/main, origin/HEAD) Update tutorial_ideas.md
* | 8648645 Adding debugging book idea
|/
*   a4eded5 Merge branch 'contact-details'
.
.
.
```

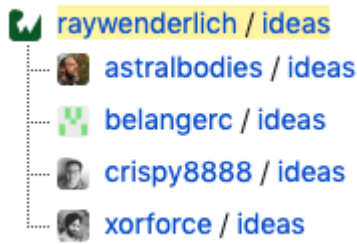
Merging two commits, regardless of where they came from, is essentially what you did when you merged your branches back to main in the previous chapter. The difference here is that Git creates a virtual “branch” that points to the commit from the remote repository, as you can see in the graphical representation of the repository tree above. There is a way around creating a messy merge commit, that involves the Git mechanism of **rebasing**. That is not in the scope for this book, but, for now, you’ll simply push your changes to the remote and live with the merge commit for now. Execute the following command to push your changes up to the remote:

```
git push origin main
```

Head over to the main GitHub page for your repository, click on the **29 commits** link, and you’ll see your changes up there on the remote.

Dealing with multiple remotes

There’s another somewhat common synchronization scenario in which you have not one, but *two* remotes to deal with. You’ve been working on your own fork of the **ideas** repository for some time, but what if there were a few changes in someone else’s forked repository that you wanted to pull down to your own local system, and merge from whatever branch that user has them in, into your main branch? Head over to the original **ideas** repository at <https://github.com/raywenderlich/ideas>. Click on the number next to the **Fork** button, and you’ll see a list of all the forks that have been created from this repository:



This mysterious `crispy8888` user has created an update on his copy of the repository that you’d like to pull down and incorporate into your local repository. Click on the **ideas** link next to the `crispy8888` username, and you’ll be taken to the `crispy8888` fork. Get the URL of this fork using the **Clone or Download** button. Back in your terminal program, execute the following to add a new remote to your repository:

```
git remote add crispy8888 https://github.com/crispy8888/ideas.git
```

This creates a new remote reference in your repository, named `crispy8888`, that points to the `crispy8888`’s fork at the above URL. Execute the following command to see that your local repository now has another remote added to it:

```
git remote -v
```


You'll see something similar to the following:

```
crispy8888      https://github.com/crispy8888/ideas.git (fetch)
crispy8888      https://github.com/crispy8888/ideas.git (push)
origin https://www.github.com/belangerc/ideas (fetch)
origin https://www.github.com/belangerc/ideas (push)
```

There you are: another remote that points to someone else's fork. Now you can work with that remote, just as you did with `origin`. Remember, the name of your first remote, `origin`, is nothing more than a convention. There's nothing special about `origin`; it's just another remote, no different than the `crispy8888` one you just created. And you don't have to name your new remote the same as the account that created it; I could easily have named that remote `whatshisname` instead of `crispy8888` and things would have worked just as well.

At this point, you only have a *reference* to the remote in your local repository; you don't actually have any of the new remote's content yet. To see this, execute the following command to see the graphical view of your repository:

```
git log --oneline --graph --all
```

It looks the same as before. But didn't you just add a remote, and then use the `--all` switch above? Even though you've instructed Git to look at all of the branches, you still can't see the changes on the `crispy8888` remote. That's because you haven't **fetch**ed any of the content yet from that fork; it's all still up on the server. Execute the following command to pull down the contents of the `crispy8888` remote:

```
git fetch crispy8888
```

At the end of the output from that command, you'll see the following two lines:

```
* [new branch]      clickbait -> crispy8888/clickbait
* [new branch]      master   -> crispy8888/master
```

Now you can look at the graphical representation of this repository with the following command:

```
git log --oneline --graph --all
```

Scroll down until you find the most recent entry that references the `crispy8888` remote, and , you'll see where this remote has diverged from the original:

```
* 3ff6fbe Merge branch 'clickbait'
|
| |
| | * fbe86a2 (crispy8888/clickbait) Added another clickbait idea
| | /
| | * e69a76a (origin/clickbait, clickbait) Adding suggestions from Mic
| | * 5096c54 Adding first batch of clickbait ideas
| | * 22d9abd (crispy8888/master) Merge branch 'master' of https://github.com/crispy8888/ideas into master
| | \
| | | * f550fed Update tutorial_ideas.md
| | | /
| | | _
| | | /
| | | |
| | | * f9278e6 Adding debugging book idea
| | | /
| | | /
| | |
```

ASCII graphing tools have their limitations, to be sure! But you get the point: there is a commit on `crispy8888/clickbait` that you'd like to pull into your own repository.

To be diligent, you should probably follow a branching workflow here so your actions are easily traceable in the log. Move to your own `clickbait` branch:

```
git checkout clickbait
```

Now you'd like to merge those two changes into your new branch. That's done in just the same way that you merge any other branch. The only difference is that you have to explicitly specify the remote that you want to merge from:

```
git merge crispy8888/clickbait
```

Git narrates every step of what it's doing like any good, modern YouTube star:

```
Updating e69a76a..9ff4582
Fast-forward
 articles/clickbait_ideas.md | 1 +
 1 file changed, 1 insertion(+)
```

Oh, that's nice — Git performed a clean fast-forward merge for you, since there were no other changes on the forked `clickbait` branch since you created your own fork. That's quite a change from your previous attempt, where you ended up with a merge commit for a simple change.

To check that Git actually created a fast-forward merge, check the first few lines of `git log --oneline --graph` (don't use the `--all` switch, so you'll just see your current branch):

```
* fbe86a2 (HEAD -> clickbait, crispy8888/clickbait) Added another clickbait idea
* e69a76a (origin/clickbait) Adding suggestions from Mic
* 5096c54 Adding first batch of clickbait ideas
```

Are you done, yet? No, you've only merged this into your local `clickbait` branch. You still need to merge this into main.

First, switch to the branch you'd like to merge into:

```
git checkout main
```

Now, merge in your local `clickbait` branch as follows:

```
git merge clickbait
```

Vim opens up, so either accept the default merge message, or press **I** to enter Insert mode to improve it yourself. When done, **Escape + Colon + w + q** will get you out of there.

Pull up the log again, with `git log --oneline --graph` to see the current state of affairs:

```
* 72670be (HEAD -> main) Merge branch 'clickbait'
|
| |
| | * fbe86a2 (crispy8888/clickbait, clickbait) Added another clickbait idea
* | b495cc8 (origin/main, origin/HEAD) Merge branch 'main' of https://github.com/belangerc/ideas
| \ \
| * | 35054cc Update tutorial_ideas.md
* | | 8648645 Adding debugging book idea
| / /
.
.
.
```

At the top is your merge commit, and below that is your work done merging from the `crispy8888` remote. You can tell that Git is pushing its ASCII art graphing skills to the limit here with just three branches at play, but `git log` does nicely in a pinch when you don't have access to your usual GUI tools.

You're done, here, so all that's left is to push this merge to `origin`. Do that as you normally would with the following command:

```
git push origin main
```

You've done a *tremendous* amount in this chapter, so there's no challenge for you. You've covered more here than any average developer would likely see in the course of a few years' worth of simple pushing, pulling, branching and merging.

Key points

Git has two mechanisms for synchronization: **pushing** and **pulling**.

`git push` takes your local commits and synchronizes the remote repository with those commits.

`git pull` brings the commits from the remote repository and merges them with your local commits.

`git pull` is actually two commands in disguise: `git fetch` and `git merge`.

`git fetch` pulls all of the commits down from the remote repository to your local one.

`git merge` merges the commits from the remote into your local repository.

You can't push to a remote that has any commits that you don't have locally, and that Git can't fast-forward merge.

You can pull commits from multiple remotes into your local repository and merge them as you would commits from any other branch or remote.

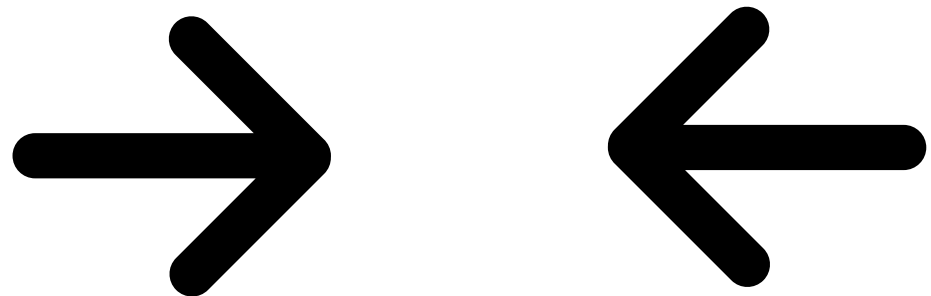
Where to go from here?

You've accomplished quite a bit, here, so now that you know how to work in a powerful fashion with Git repositories, it's time to loop back around and answer two questions:

“How do I create a Git repository from scratch?”

“How to I create a remote repository from a local one?”

You'll answer those two questions in the next two chapters that will close out this Beginning Git section of the book, and lead you nicely into the Intermediate Git chapters to come.



Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.
© 2022 Razeware LLC