Git Remotes

In this chapter, we're going to be looking at **remote repositories**, or as we call them in Git, simply **remotes**.

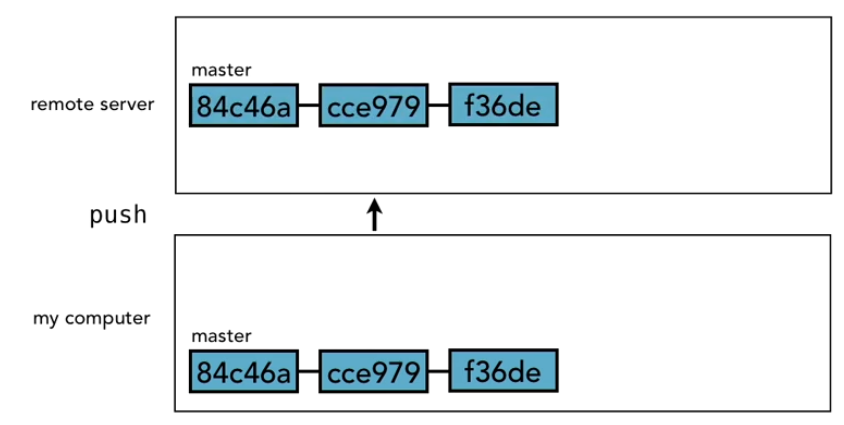
In everything we've done up until now is on our own computer. We can do version control just on our local files and not share them with anyone else, and that's very effective. But Git becomes even more powerful when we're able to collaborate with others, and that's what remotes allow us to do.

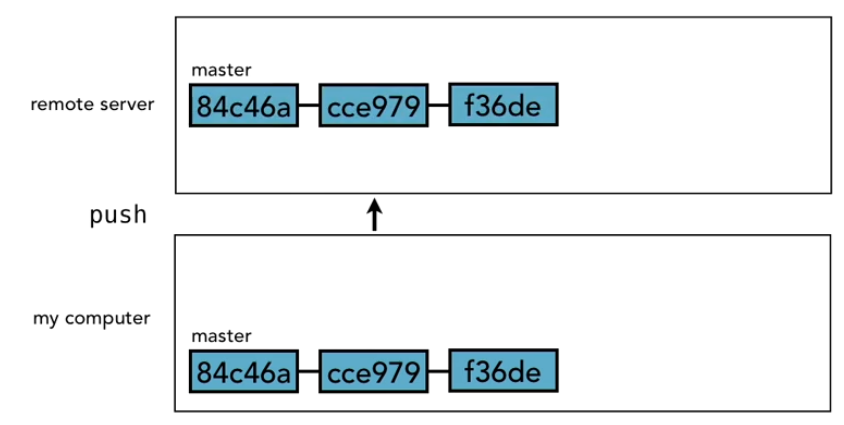
The concept is that there is a remote server, and we'll take our changes that we've made and put them on that remote server so that other people can then see them. They can then download the changes that we've made to their repositories, they can make changes of their own, upload those back to the remote server, and we can pull those back down into our repository to get their changes.

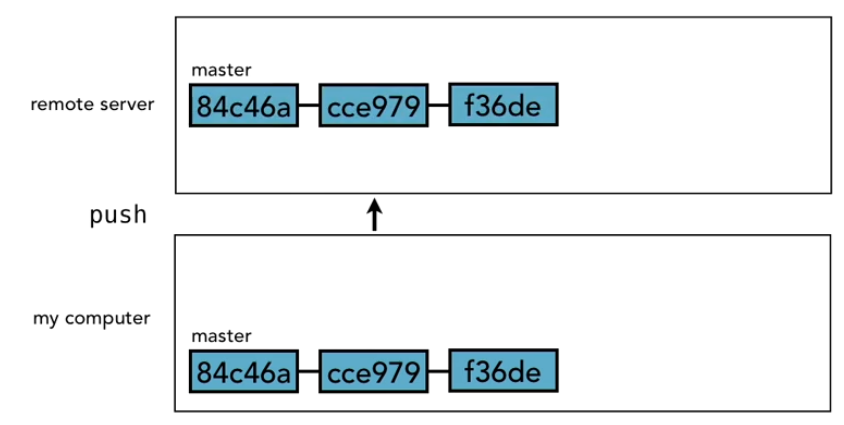
It makes this remote repository sort of a central clearing house for these different changes that are going on, and that remote server is just simply a Git repository. Remember that Git is distributed version control. There's no real difference between the different repositories, so there's not a big difference between the **server** and our computer or the client. The only difference really is that the server is running some Git software that allows it to communicate with lots of different Git **clients** at the same time, but the actual repository where those changes are being stored, is just simply a Git repository; it has commits, it has branches, it has a HEAD pointer.

Now we're going to introduce a remote server. The process that we used to do that is called a **push**. We **push** our changes to the remote server.

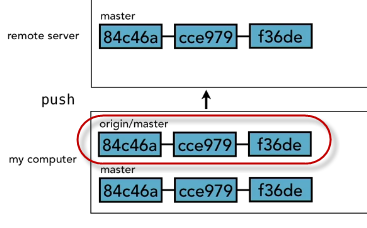
At that point, the remote server creates the same branch with the same commits with the exact same commit IDs referring to all of them.

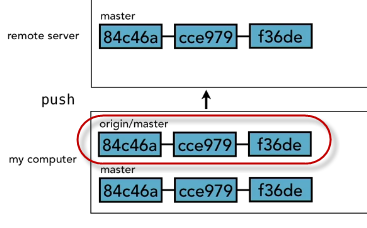






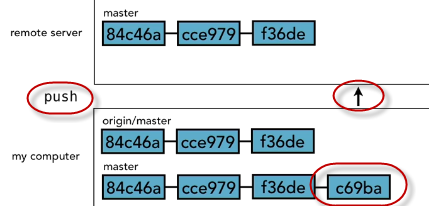
At that point our collaborators can see our commits. At the same time, **Git also makes another branch on our local computer** that is typically called **origin** **slash** and then whatever the **branch name** is. Now that's by convention, we'll talk about how we can change that name, it doesn't have to be origins, but in this case, we'll stick with the

default.

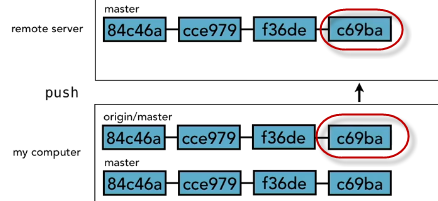


So, **origin/master** is a branch on our local machine that references the remote server branch, and it always tries to stay and sync with that. Right now, it looks like all **three of them are in sync**, but it doesn't always have to be that way.

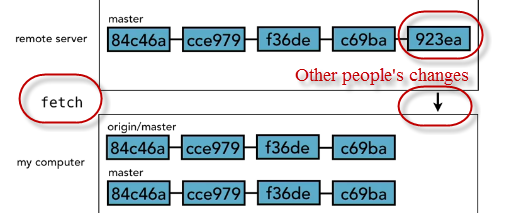
We continue developing, we make some more commits on our master branch when we're ready to share those commits, again, we do a push,



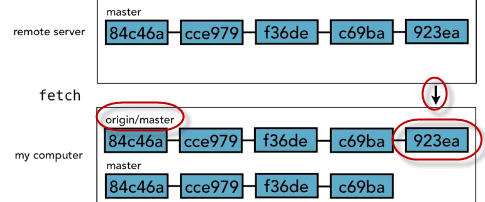
so we **push** that code up to the remote server, and **it makes note of the change also in our origin/master branch**, the one that tries to stay in sync with the remote one.



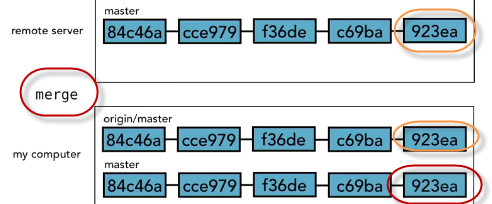
When other people make changes to the remote server and contribute them there, we need to pull those changes down so that we know about them, and the way that we do that is with what's called a **fetch**.



We fetch the changes, at that point they come into our origin master branch, because what we're doing is keeping those in sync. Fetch is essentially saying sync up my origin master with the remote server version, but it does not bring it into our master branch.

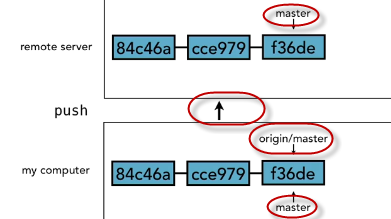


Now our computer knows about the change, we have it locally, if we get on an airplane and fly somewhere while we're on the airplane, we'll have a copy of that commit that 923ea, but **it won't be in our master branch until we do a merge**, and at that point then it'll be brought in to our master branch, and we'll be back in sync.



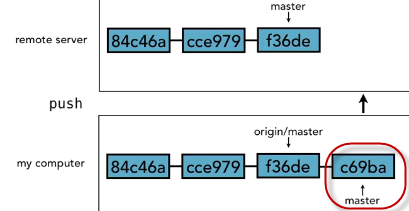
Now this is an over simplified example, because you might notice that on my computer I have lots of duplicates. You can see that I have those same Git objects listed twice on my own computer. In realty, it doesn't store them twice. Git is smart about reusing these objects, because they're exactly the same.

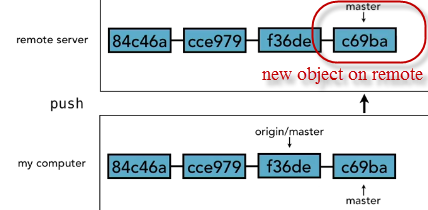
So, Git uses pointers instead, to point to that, and we know about the HEAD pointer. So let's take a quick look at how that works just to make sure that that's clear. I think for illustrative purposes, it was good to see it as two separate branches with two sets of commits. But, we have a HEAD pointer for master that points to the third commit,



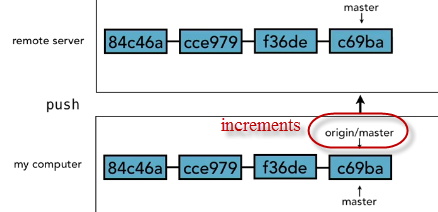
when we do a push to remote server, it creates the commits there, and moves the HEAD pointer for master to the third commit, and then also adds another pointer that points to the same commit on our local computer. It doesn't duplicate those three objects.

When we then make a new commit, it moves our master pointer to the commit,

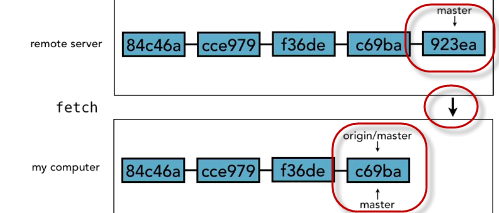




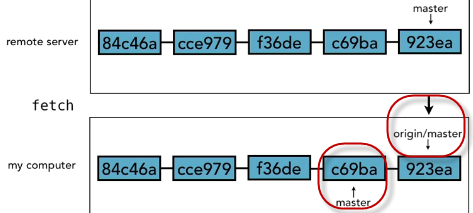
when we do a **push** it creates that new object on the remote server and moves the pointer, and **also increments origin/master**.

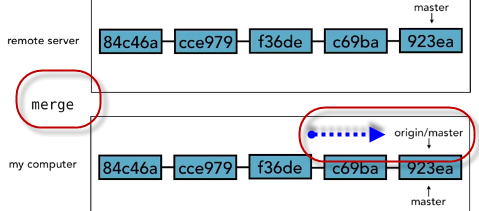


Then when new changes come into the remote server from someone else, of course, it moves the master pointer to point to that latest commit. When we do a fetch, it brings that new Git object down onto our computer and moves the origin pointer to point to it, but our (local) master ones still doesn't.



We have to first do a fast-forward merge, and we know that would be a fast-forward merge, that's what we would call that, in this case where it just moves all the way to point to the same commit.

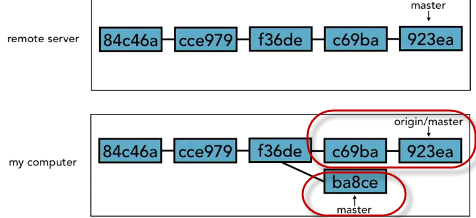




So as you can probably see, origin/master really is just a branch, it's just like the other branches that we're working with. The only difference is that it tries to stay in sync with what's on the remote server.

The reason that matters is because it is possible for someone else (a third dev) to make a commit on the remote server while we're in the process of making a commit on our local machine.

All right, it happens all the time. I'm making changes to one part of the project, my collaborator is making changes to their part of the project, they put theirs up on the remote server, my origin master, once I do a fetch, the state of our two servers would look something like this.



You'll see that origin/master does still include those objects (c69ba and 923ea) that are in the remote branch, it did stay in sync with that, but in the meantime, I've got a **new** commit called **ba8ce**. So now that our two branches have diverged, we need to do a **merge** to bring them back together again, and that process works exactly the same way as we saw when we were working with emerging branches. Origin/master is just a branch, so we can merge them together and then the next time we do a push, our master changes will be merged in and sent off to the remote server.

So , the process that you go through when you're working with a remote, is that you'll do your commits locally, then you'll fetch the latest from the remote server, get your origin branch in sync (with remote?), then merge any of the new work you did into what just came down from the server and then push the result back up to the remote server. If the process seems at all unclear now, it'll become second nature soon enough.