Git的基本使用

初始化及初始配置

在使用git之前，需要先设置用户名和邮箱：

```
git config --global user.name <你的用户名>
git config --global user.email <你的邮箱>
```

这将会在每次提交时，告诉git，是谁提交的项目。在协作项目中非常有用。

在命令中，--global表示系统中当前用户的设置，也可以写成--system，表示系统的设置，还可以不写，表示本仓库的设置。本仓库的设置优先级最大，--global次之，--system最小。

创建仓库

创建仓库需要在一个目录下执行命令：

```
git init
```

这将把整个目录变成一个git仓库。

也可以执行：

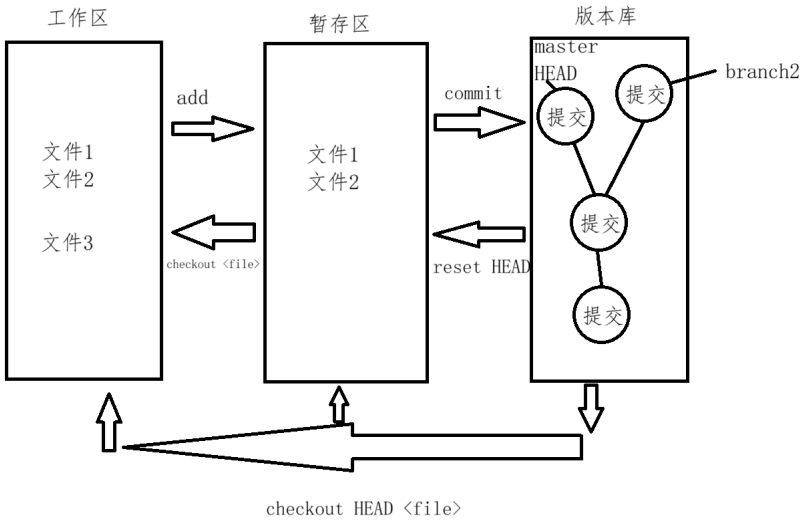
```
git init <目录名>
```

这将新建一个目录，并将这个目录变成一个git仓库。

git本地操作

示意图

git的示意图如下：



工作区

工作区就是我们进行开发的区域，当开发完毕后，可以将文件增加至暂存区

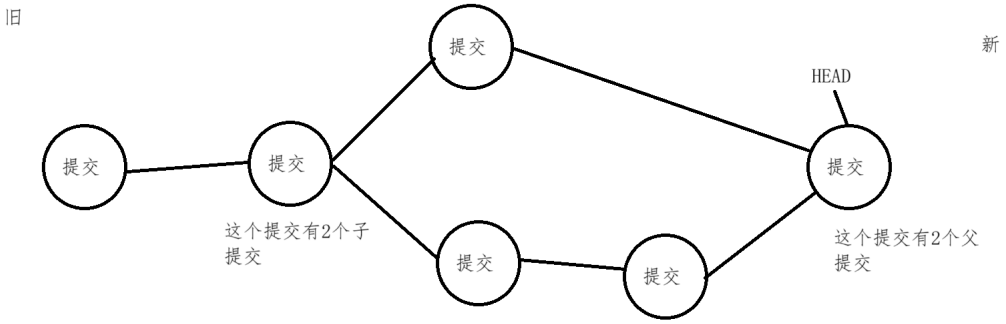
暂存区

接收工作区传入的文件，当开发好一步后，可以将文件提交至版本库

版本库

版本库存储的是提交链，由一个个提交组成，除了第一个提交外，每个提交都有一个或多个父提交，是这个提交之前的那个提交。每个提交都有经过SHA-1哈希函数计算的唯一ID。

git允许用户创建分支，分支是一个指针，指向该分支下最新的提交。分支的第一个提交的父提交可能有多个子提交，而分支合并意味着合并后的提交有多个父提交，分别来自合并的分支。



git的HEAD也是一个指针，指向的提交是下一个提交的父提交。

基本操作

打开一个空文件夹，创建一个版本库：

```
git init
```

然后在里面写一个hello.txt：

```
Hello World
```

我们可以通过git status查看工作区和暂存区的状态：

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        hello.txt

nothing added to commit but untracked files present (use "git add"
to track)
```

可以看到，hello.txt还没有被标记，暂存区里也没有这个文件。

我们也可以通过git status -s查看，这会输出更简易的结果：

```
$ git status -s
?? hello.txt
```

这其中，??表示没有被标记，暂存区里也没有这个文件。

接下来我们用git add命令提交一下：

```
git add hello.txt
```

然后查看一下状态：

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   hello.txt
```

这就说明，hello.txt是要被提交的文件，即在暂存区里的文件，通过`git rm --cached <file>...`可以将其从暂存区中删除。

用git status -s查看：

```
$ git status -s
A  hello.txt
```

可以看到，hello.txt显示了一个A，说明已进入暂存区，且是一个新增加的文件。

在git status -s命令下，第一列表示在暂存区的文件，第二列表示在工作区的文件。

接下来修改hello.txt：

```
Hello World
Hello Git
```

这时候在查看一下状态：

```
$ git status -s
AM hello.txt
```

这里面，AM表示hello.txt是一个新增加的文件，工作区的文件有改动。

接下来进行提交，提交时提交的hello.txt是第一版的，只有一个Hello World的那一个：

```
git commit -m "Write 'Hello World'."
```

在这里面，-m后面指定的是对这个提交的描述。如果不指定，git则会开启一个编辑器（如vim），让用户自己写。

接下来我们再查看一下状态：

```
$ git status -s
 M hello.txt
```

可以看到，暂存区中的文件和版本库中的一样，工作区中的有修改。接下来我们add一下，并查看状态：

```
$ git add .
$ git status -s
M  hello.txt
```

可以看到，M从第二列移到了第一列，说明工作区中的文件与暂存区的文件没有变化，而暂存区的文件同版本库刚提交的文件有变化。

接下来我们将这个也提交一下：

```
git commit -m "Write 'Hello Git'."
```

此时查看一下状态：

```
$ git status -s
```

可以发现没有输出，这是因为工作区和暂存区的文件和版本库中的一样，没有改动。

我们可以通过git log命令查看版本库的提交：

```
$ git log
commit ed42a3faa1a0f4c45549d44f497645f1e867e78b (HEAD -> master)
Author: wtz <momomomo6508@qq.com>
Date:   Thu Nov 14 12:34:45 2024 +0800

    Write 'Hello Git'.

commit 284d779a683754d8956a42f3afdf7aa9218536cb
Author: wtz <momomomo6508@qq.com>
Date:   Thu Nov 14 12:31:16 2024 +0800

    Write 'Hello World'.
```

commit后面的就是提交ID，当实际使用时，只需要写出前6位，并不冲突即可。

我们可以通过git log --graph命令看到提交链：

```
$ git log --graph
* commit ed42a3faa1a0f4c45549d44f497645f1e867e78b (HEAD -> master)
| Author: wtz <momomomo6508@qq.com>
| Date:   Thu Nov 14 12:34:45 2024 +0800
|
|     Write 'Hello Git'.
|
* commit 284d779a683754d8956a42f3afdf7aa9218536cb
  Author: wtz <momomomo6508@qq.com>
  Date:   Thu Nov 14 12:31:16 2024 +0800

      Write 'Hello World'.
```

此时，HEAD指针和分支master指针都指向了提交ed42a3。

接下来我们通过git checkout命令将工作区和暂存区设置到284d77提交时的状态：

```
git checkout 284d77
```

git checkout命令会将HEAD指针指向指定的提交，同时更改工作区和暂存区的文件。

在这时候，git会提示我们进入分离头指针状态，这种状态下，HEAD指针不指向任何分支指向的提交，在HEAD指针到分支指针之间就会有提交不可见了：

```
$ git log
commit 284d779a683754d8956a42f3afdf7aa9218536cb (HEAD)
Author: wtz <momomomo6508@qq.com>
Date:   Thu Nov 14 12:31:16 2024 +0800

    Write 'Hello World'.
```

可以发现，此时提交ed42a3已经不见了，而工作区中的hello.txt也只有了一行Hello World.

那么，我们应该如何回到原来的提交呢？其实就是把HEAD指针checkout到提交ed42a3就好了。但是，怎么确定一定是提交ed42a3呢？可以使用git reflog命令查看之前改变提交链的命令：

```
$ git reflog
284d779 (HEAD) HEAD@{0}: checkout: moving from master to 284d77
ed42a3f (master) HEAD@{1}: commit: Write 'Hello Git'.
284d779 (HEAD) HEAD@{2}: commit (initial): Write 'Hello World'.
```

可以看到，在commit: Write 'Hello Git'.这一行中，有ed42a3f，说明这次提交的ID是ed42a3f，因此可以用如下命令回到原来的提交：

```
git checkout ed42a3f
```

可以看到，我们成功的回到了原来的提交。

和checkout类似的命令是reset，它也用于设置HEAD指针的位置，如：

```
git reset --soft <提交>    # 仅设置HEAD指针，不改变其他区域
git reset --mixed <提交>   # 设置HEAD指针并更改暂存区，不改变工作区。这是reset的默认操作
git reset --hard <提交>    # 设置HEAD指针并更改暂存区和工作区
```

如我们的例子可以用以下命令代替：

```
git reset --hard 284d77
git reset --hard ed42a3f
```

另外，如果想表示一个提交的父提交，可以用^表示，如果有多个父提交，可以在^后面跟上数字，表示第几个父提交。如：

```
HEAD^     # HEAD的最近一次父提交
HEAD^2    # HEAD的第二个父提交
HEAD^^    # HEAD的父提交的父提交，也可以表示为HEAD~2
```

分支操作

分支的查看用git branch命令：

```
$ git merge master  # 将分离的头指针设置到master上
$ git branch
* master
```

前文已经说过，分支的本质是一个指针，指向不同的提交。我们可以试一下：

创建一个分支并检入：

```
git checkout -b hello
或
git branch hello
git checkout hello
```

此时我们就进入的hello分支。接下来我们在hello分支里提交一点东西：

```
echo "Hello Beijing." > hello2.txt
git add .
git commit -m "Write 'Hello Beijing.'."
```

此时我们查看一下提交链：

```
$ git log --graph --oneline
* 4cdb400 (HEAD -> hello) Write 'Hello Beijing.'.
* ed42a3f (master) Write 'Hello Git'.
* 284d779 Write 'Hello World'.
```

这里面--oneline用于将提交放在一行显示，简化输出。

此时我们再进入master分支，提交一点东西：

```
git checkout master
echo "Hello BDA" > hello3.txt
git add .
git commit -m "Write 'Hello BDA'."
```

然后查看提交链：

```
$ git log --graph --oneline
* 99d5108 (HEAD -> master) Write 'Hello BDA'.
* ed42a3f Write 'Hello Git'.
* 284d779 Write 'Hello World'.
```

因为此时HEAD指针所在的提交链无法向前追溯到hello分支，因此hello的提交我们看不到。

好了，现在有两个分支了，在实际开发中，通常一个分支写一个功能或改一个bug，最后将所有分支合并在一起，这也是分支常用于做的事。那么，如何将分支放在一起呢？

用git merge命令即可：

```
git merge hello -m "Merge branch 'hello'"
```

注意，git merge命令同样需要加描述，如果不指定-m，则git会打开一个编辑器。

合并完成后，可以查看提交链：

```
$ git log --graph --oneline
*   6cecfac (HEAD -> master) Merge branch 'hello'
|\
| * 4cdb400 (hello) Write 'Hello Beijing.'.
* | 99d5108 Write 'Hello BDA'.
|/
* ed42a3f Write 'Hello Git'.
* 284d779 Write 'Hello World'.
```

可以看到，提交链出现了一个分叉，最后又合并起来了。此时，hello分支仍然存在，如果我们到hello分支再写一个提交会发生什么：

```
git checkout hello
echo "Hello Haidian" > hello4.txt
git add .
git commit -m "Write 'Hello Haidian'."
```

然后查看提交链：

```
$ git log --graph --oneline
* b2bf7ca (HEAD -> hello) Write 'Hello Haidian'.
* 4cdb400 Write 'Hello Beijing.'.
* ed42a3f Write 'Hello Git'.
* 284d779 Write 'Hello World'.
```

可以看到，这里面的提交链并没有分支，因为master合并hello后，hello仍指向Hello Beijing的那个提交，当进行新的提交后，git创建了一个提交，和合并hello的那个“提交”（其实是合并）一样，它们的父提交都是Hello Beijing的那个提交。

接下来我们合并一下master：

```
git mergr master -m "Merge branch 'master' into hello"
```

查看提交链：

```
$ git log --graph --oneline
*   634ab4e (HEAD -> hello) Merge branch 'master' into hello
|\
| *   6cecfac (master) Merge branch 'hello'
| |\
| * | 99d5108 Write 'Hello BDA'.
* | | b2bf7ca Write 'Hello Haidian'.
| |/
|/|
* | 4cdb400 Write 'Hello Beijing.'.
|/
* ed42a3f Write 'Hello Git'.
* 284d779 Write 'Hello World'.
```

可以看到，这个提交链稍显复杂。

冲突解决

好了，不看这么复杂的提交链了，我们新建一个仓库：

```
cd ..
mkdir demo
cd demo
git init
```

然后创建hello.txt并提交：

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

接下来创建一个分支并进入：

```
git checkout -b b1
```

b1分支想要把在hello.txt上加一行：

```
echo "Hello Java" >> hello.txt
git add .
git commit -m "Add Java line."
```

切到master分支，master分支也想加上一行：

```
git checkout master
echo "Hello Kotlin" >> hello.txt
git add .
git commit -m "Add Kotlin line."
```

这时候两个分支想要合并，结果：

```
$ git merge b1
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.
```

为啥呢？因为分支冲突。我们可以打开hello.txt看一下：

```
$ cat hello.txt
Hello World
<<<<<<< HEAD
Hello Kotlin
=======
Hello Java
>>>>>>> b1
```

现在hello.txt已经被改成了这样，`Hello Kotlin`部分是master分支（HEAD指针所在分支）的内容，而`Hello Java`是bi分支的内容，git都给我们整理对比出来了。我们必须修改hello.txt的内容，才能继续合并。

如果你使用vscode等IDE，它会给你一些处理冲突的选项。也可以直接编辑hello.txt，如把它修改成这样：

```
Hello World
Hello Kotlin
Hello Java
```

然后添加到暂存区并提交：

```
git add .
git commit -m "solve merge"
```

此时就完成了合并：

```
$ git log --graph --oneline
*   794102a (HEAD -> master) solve merge
|\
| * 1d6b4b0 (b1) Add Java line.
* | 1c0dc70 Add Kotlin line.
|/
* 19a12e1 Hello World
```

git远程操作

选择一个远程git网站，如GitCode，登录或注册账号后新建一个项目，起个名字，然后点击头像-个人设置-访问令牌，新建一个访问令牌。这个访问令牌就是git访问的密码。

如果你想使用已有项目，直接从那个项目中获取链接即可。

根据提示，用git clone命令创建对应的本地仓库：

```
git clone <远程仓库链接>
```

此时会提示输入用户名和密码，用户名是GitCode用户名，用户密码是访问令牌。

注意克隆完之后，git会新建一个目录，仓库在这里面。

接下来简单创建一个文件并提交：

```
echo "Hello GitCode" > hello.txt
git add .
git commit -m "Hello GitCode"
```

这只是提交到了本地版本库，还需要通过git push命令提交到远程：

```
git push
```

通过git pull将远程仓库内容拉取到本地：

```
git pull
```

git pull其实是两个命令的结合，分别是：

```
git fetch
git merge
```

git fetch是从远程服务器获取内容的命令，git merge是合并分支命令。git fetch会将远程仓库的分支设置到origin/下，如master设置到origin/master，因此需要进行合并操作。

参考资料

[1]蒋鑫. Git 权威指南[M]. 北京：机械工业出版社