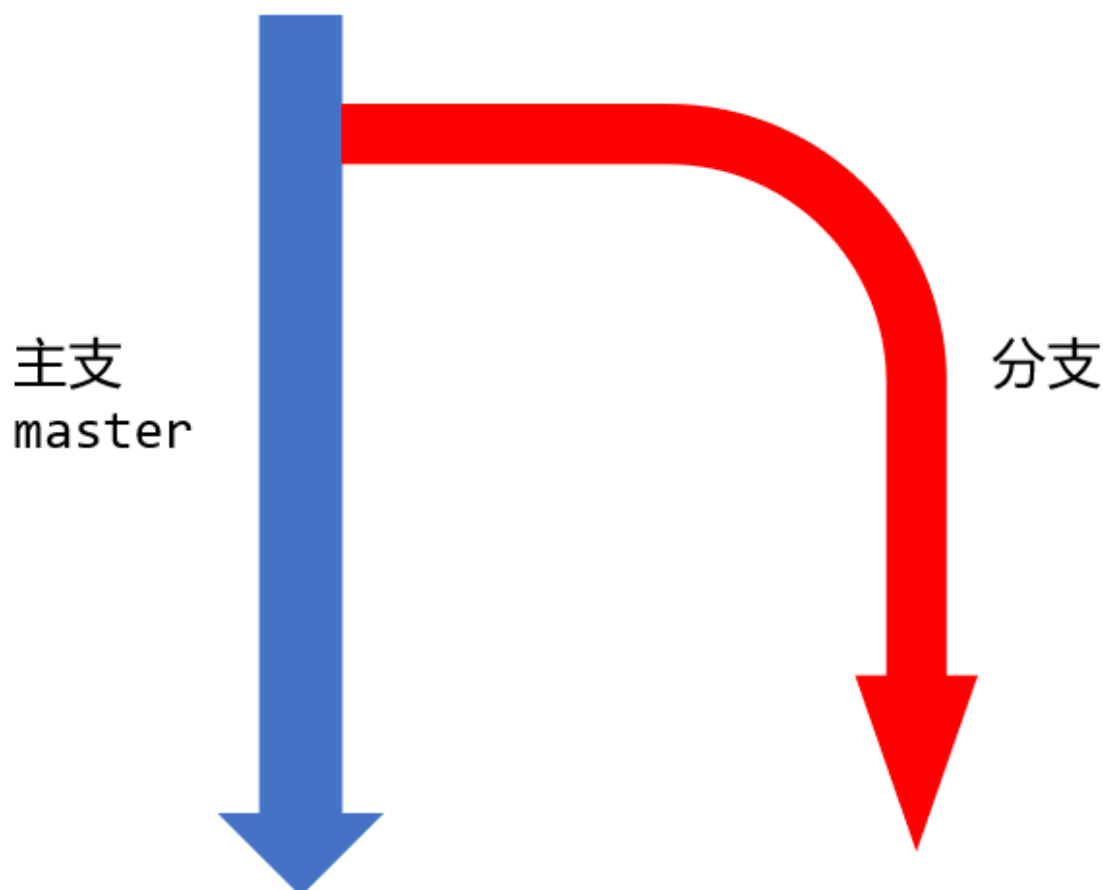


先分支管理

举一个例子: 现在公司要求写一个项目, 预计要求大约1个月完成, 但是你第一周就完成了70%, 接下来突然接收到调令需要临时出差, 你只能在酒店或是高铁上断断续续的写, 这时候为了防止代码丢失, 必须要提交到git上去. 但你写了一半的代码是没法运行的, 开发其他模块的同事一旦同步了你的代码, 那就是满屏幕的错误了, 要是等写完了再提交, 万一硬盘挂了, 那么所有代码都凉凉了.

所以这时候, 我们就需要一个临时单独空间, 可以让我们存储个人的代码, 别人看不到也无法同步你的临时代码, 等到项目完成了再一次性提交上去. 这个**临时空间就是分支**

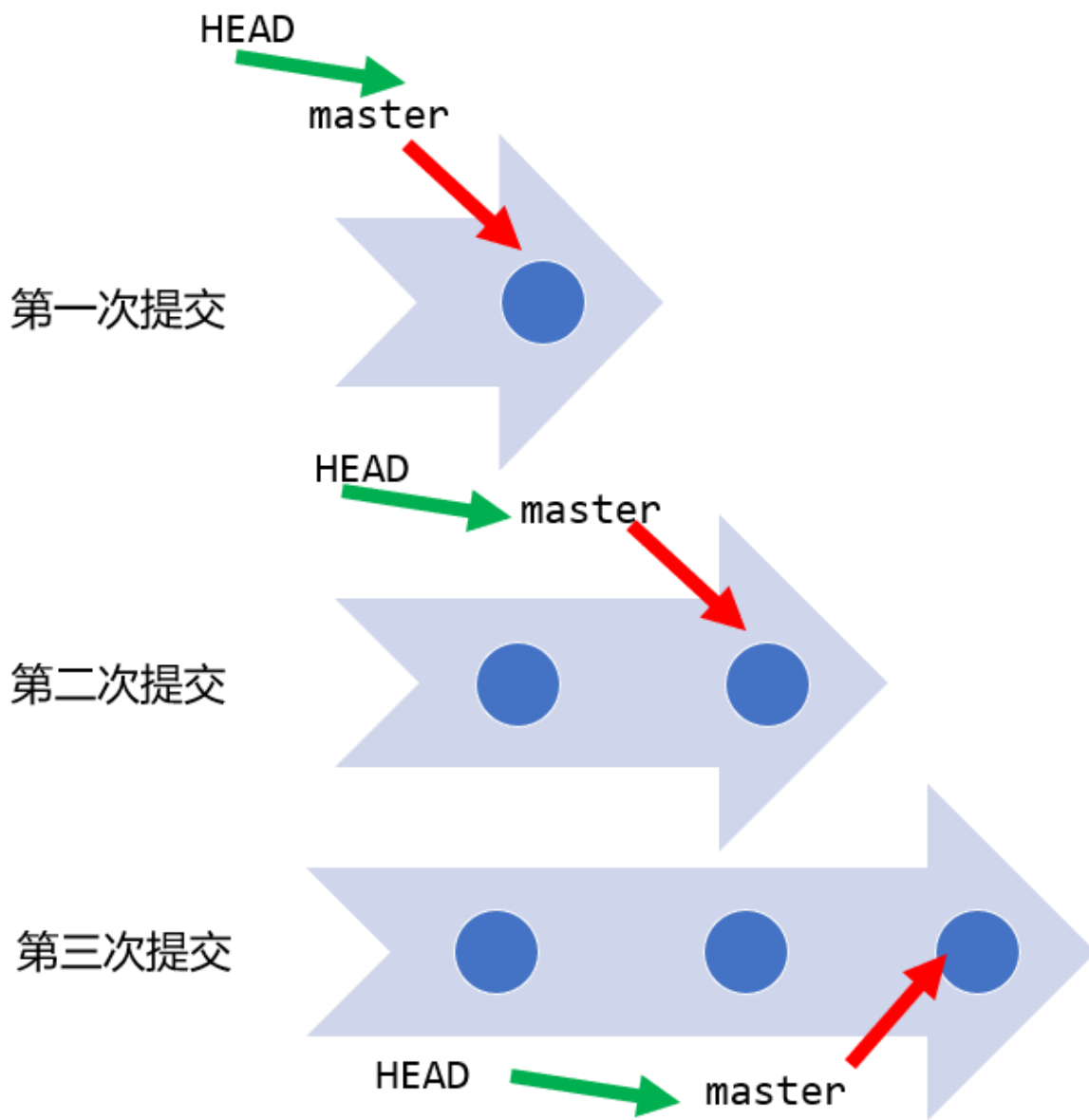


##创建和合并分支

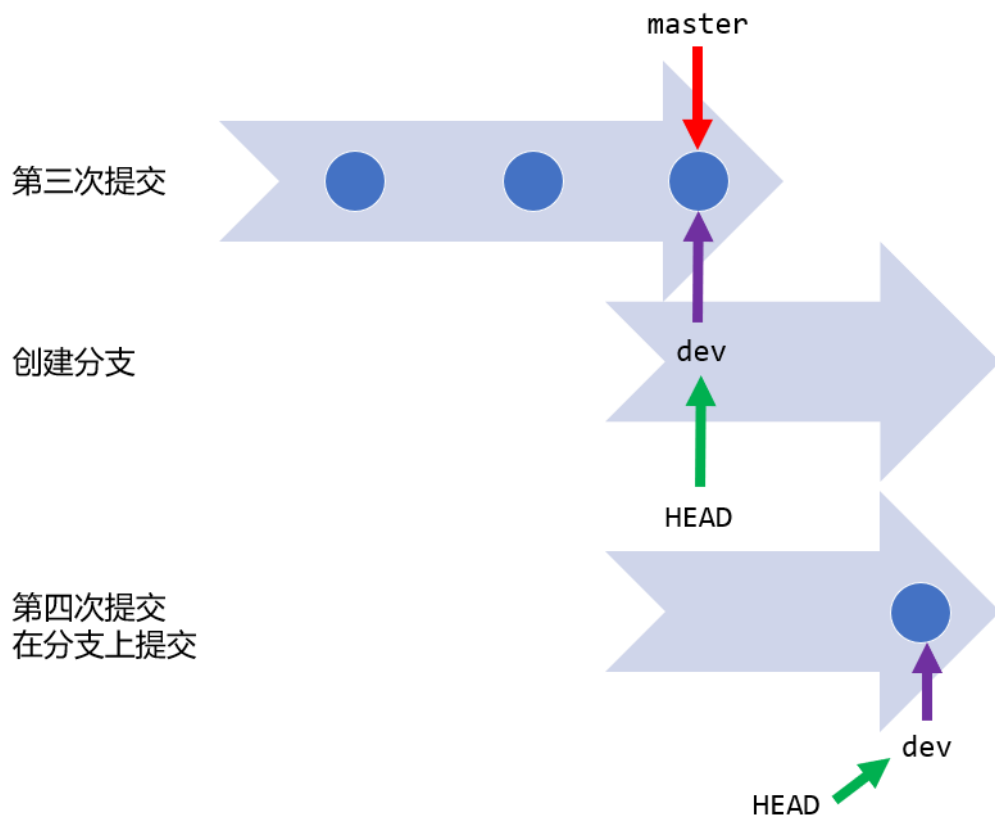
在**版本回滚**里, 你已经知道, 每次提交, Git都把它们串成一条时间线, 这条时间线就是一个分支。

###理论模型

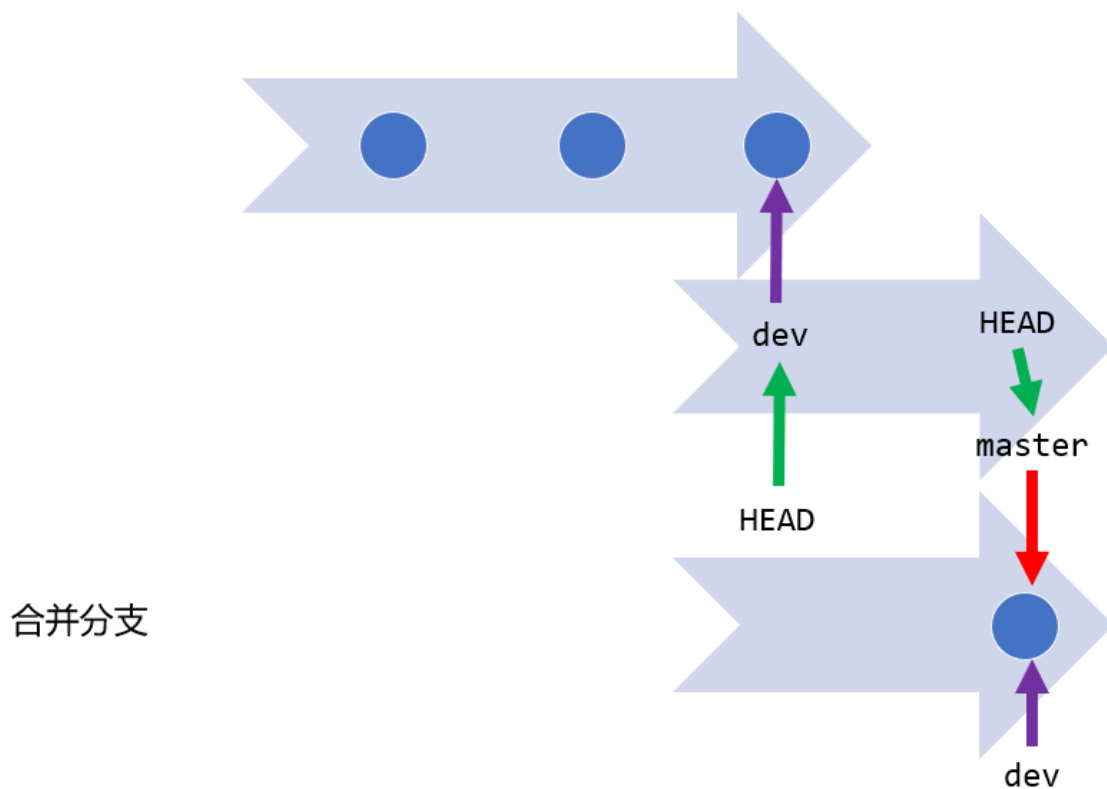
之前咱们知道 HEAD是指当前版本, 其实从实质上, HEAD相对于一个指针, 指向了master, 而master则指向了最新的提交, 就如同是 $a=b=1$;



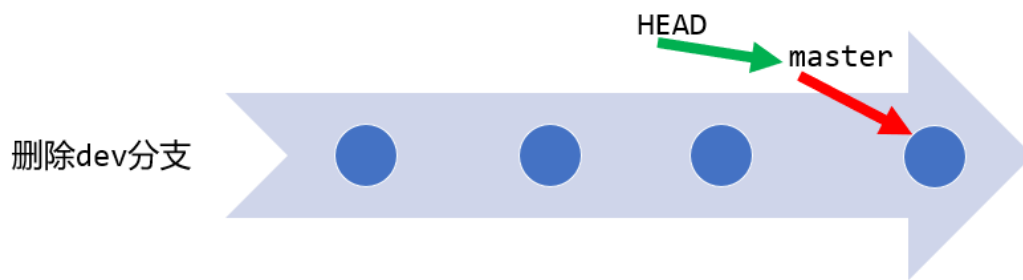
当我们创建新的分支，例如 `dev` 时，Git 新建了一个指针叫 `dev`，指向 `master` 相同的提交，再把 `HEAD` 指向 `dev`，就表示当前分支在 `dev` 上。现在开始，对工作区的修改和提交就是针对 `dev` 分支了，比如新提交一次后，`dev` 指针往前移动一步，而 `master` 指针不变：



假如我们在 `dev` 上的工作完成了，就可以把 `dev` 合并到 `master` 上。Git怎么合并呢？最简单的方法，就是直接把 `master` 指向 `dev` 的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！合并完分支后，甚至可以删除 `dev` 分支。删除 `dev` 分支就是把 `dev` 指针给删掉，删掉后，我们就剩下了一条 `master` 分支：



示例代码

Mode	LastWriteTime		Length	Name
----	-----		-----	----
-a----	2019-08-15	12:49	0	demo.txt
-a----	2019-08-14	21:40	0	key.txt
-a----	2019-08-14	22:59	125	readme.txt

还是在老项目的基础上来进行实践

1. 创建分支

```
1 | $ git branch dev
2 | // git branch "分支名"
```

2. 切换分支

```
1 | $ git checkout dev
```

```
λ git branch
E:\learngit [master ≡]
λ git checkout dev
Switched to branch 'dev'
```

3. 查看分支

```
1 | $ git branch
```

```
λ git branch
* dev
master
```

`git branch` 命令会列出所有分支，当前分支前面会标一个*号。

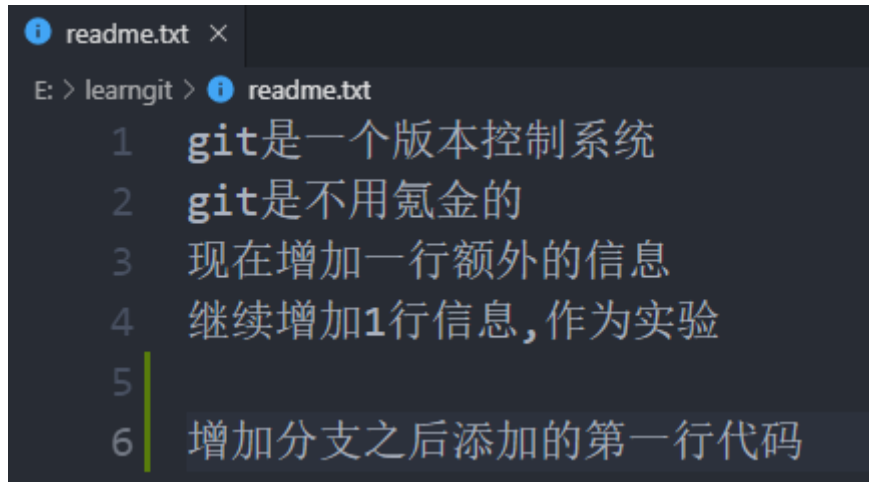
4. 联合指令 创建并切换

```
1 | $ git checkout -b dev
```

`git checkout` 命令加上 `-b` 参数表示创建并切换，相当于以下两条命令：

```
1 $ git branch dev
2 $ git checkout dev
```

5. 更改代码



```
readme.txt x
E: > learngit > readme.txt
1 git是一个版本控制系统
2 git是不用氮金的
3 现在增加一行额外的信息
4 继续增加1行信息,作为实验
5
6 增加分支之后添加的第一行代码
```

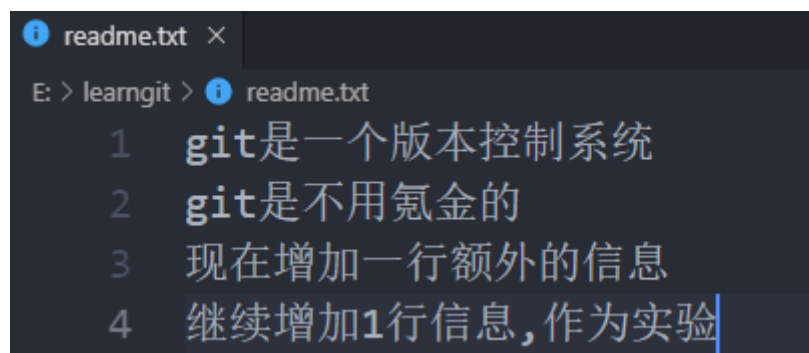
6. 提交

```
1 $ git add readme.txt
2 $ git commit -m "在增加分支后添加了一些文字"
```

```
λ git add .\readme.txt
E:\learngit [dev +0 ~1 -0 ~]
λ git commit -m "在新的分支上更改了readme的内容"
[dev 123ae2b] 在新的分支上更改了readme的内容
1 file changed, 3 insertions(+), 1 deletion(-)
```

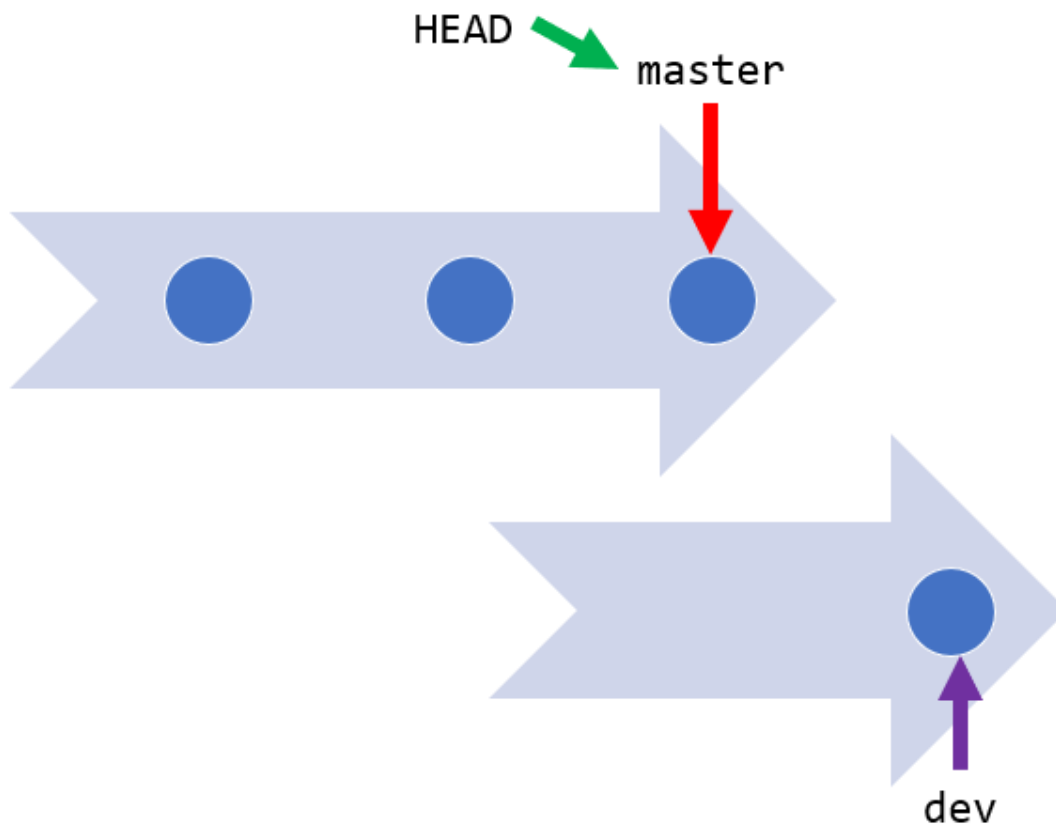
7. 切换分支

```
1 $ git branch master
```



```
readme.txt x
E: > learngit > readme.txt
1 git是一个版本控制系统
2 git是不用氮金的
3 现在增加一行额外的信息
4 继续增加1行信息,作为实验
```

切换回 `master` 分支后, 再查看一个 `readme.txt` 文件, 刚才添加的内容不见了! 因为那个提交是在 `dev` 分支上, 而 `master` 分支此刻的提交点并没有变:



8. 合并分支

```
1 | $ git merge dev
```

`git merge` 命令用于**合并指定分支到当前分支**。合并后，再查看 `readme.txt` 的内容，就可以看到，和 `dev` 分支的最新提交是完全一样的。

```
λ git merge dev
Updating 59b6d5e..123ae2b
Fast-forward
 readme.txt | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

注意到上面的 `Fast-forward` 信息，Git 告诉我们，这次合并是“快进模式”，也就是直接把 `master` 指向 `dev` 的当前提交，所以合并速度非常快

9. 删除分支

```
1 | $ git branch -d dev
```

```
λ git branch -d dev
Deleted branch dev (was 123ae2b).
```

```
λ git branch
* master
```

删除分支之后再来看就能发现，此时就剩下 `master` 分支了

因为创建、合并和删除分支非常快，所以 Git 鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在 `master` 分支上工作效果是一样的，但过程更安全。

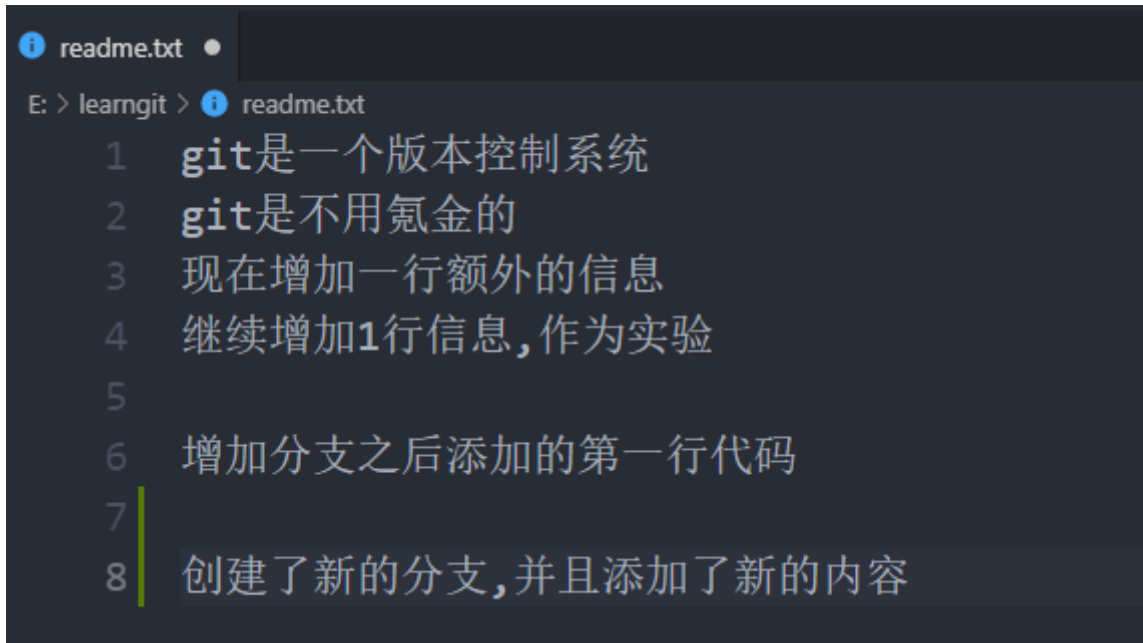
解决冲突

分支的合并有时候会出现一些意想不到的问题, 下面我们来几个例子

1. 准备新的 feature1 分支, 继续我们的新分支开发:

```
1 $ git checkout -b feature1
2 Switched to a new branch 'feature1'
```

2. 新建了分支之后, 再修改一下readme.txt文档



```
readme.txt
E: > learnngit > readme.txt
1 git是一个版本控制系统
2 git是不用氮金的
3 现在增加一行额外的信息
4 继续增加1行信息, 作为实验
5
6 增加分支之后添加的第一行代码
7
8 创建了新的分支, 并且添加了新的内容
```

3. 在 feature1 分支上提交:

```
λ git commit -m "新增了readme文件的文字信息"
[feature1 badf697] 新增了readme文件的文字信息
1 file changed, 3 insertions(+), 1 deletion(-)
```

4. 切换分支

```
1 $ git checkout master
```

```
λ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
```

Git还会自动提示我们当前 master 分支比远程的 master 分支要超前1个提交。

在 master 分支上把 readme.txt 文件的最后一行改为: 最后一行

```
readme.txt
E: > learn git > readme.txt
1 git是一个版本控制系统
2 git是不用氮金的
3 现在增加一行额外的信息
4 继续增加1行信息,作为实验
5
6 增加分支之后添加的第一行代码
7
8 最后一行
```

5. 在master分支上提交

```
1 $ git add readme.txt
2 $ git commit -m "增加 最后一行信息"
```

```
λ git add .\readme.txt
E:\learngit [master ↑1]
λ git commit
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Git还会自动提示我们当前 `master` 分支比远程的 `master` 分支要超前1个提交。

这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突

```
λ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

6. 冲突处理

告诉我们，`readme.txt` 文件存在冲突，必须手动解决冲突后再提交。

`git status`可以告诉我们冲突的文件


```

λ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   readme.txt

no changes added to commit (use "git add" and/or "git commit -a")

```

查看了readme.txt

```

E:\> learn git > readme.txt
1  git是一个版本控制系统
2  git是不用氪金的
3  现在增加一行额外的信息
4  继续增加1行信息,作为实验
5
6  增加分支之后添加的第一行代码
7
8  Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
9  <<<<<<< HEAD (Current Change)
10 2
11 =====
12 创建新的分支,并且添加了新的内容
13 1
14 >>>>>> feature1 (Incoming Change)
15

```

Git用<<<<<<<, =====, >>>>>>> 标记出不同分支的内容, 我们修改如下后保存:

```

Untitled-1  readme.txt x
E:\> learn git > readme.txt
1  git是一个版本控制系统
2  git是不用氪金的
3  现在增加一行额外的信息
4  继续增加1行信息,作为实验
5
6  增加分支之后添加的第一行代码
7
8  Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
9  <<<<<<< HEAD (Current Change)
10 2
11 =====
12 创建新的分支,并且添加了新的内容
13 1
14 >>>>>> feature1 (Incoming Change)
15

```

```

λ git add readme.txt
E:\> learn git [master ↑1 +0 ~1 -0 ~]
λ git commit -m "2"
[master 9ecfc52] 2

```

再次提交就可以了, 主要一个核心哲学, 每个人在自己的分支里面完成模块的编写, 不要没事跑到master上更改

7. 删除分支

```
1 $ git branch -d feature1
2 Deleted branch feature1 (was 06f00ec).
```

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

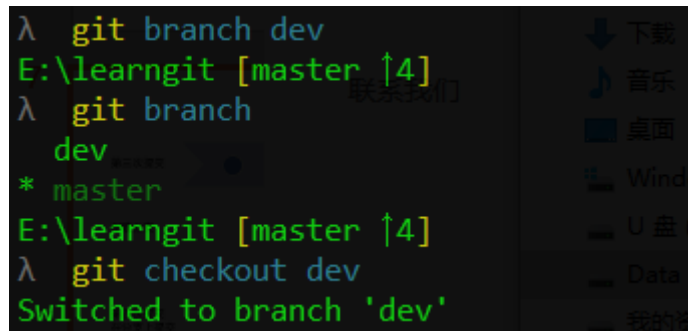
git log --graph命令可以看到分支合并图。

分支管理策略

通常，合并分支时，如果可能，Git会用 Fast forward 模式(速度优先模式)，但这种模式下，删除分支后，会丢掉分支信息。如果要强制禁用 Fast forward 模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息

首先，仍然创建并切换 dev 分支：

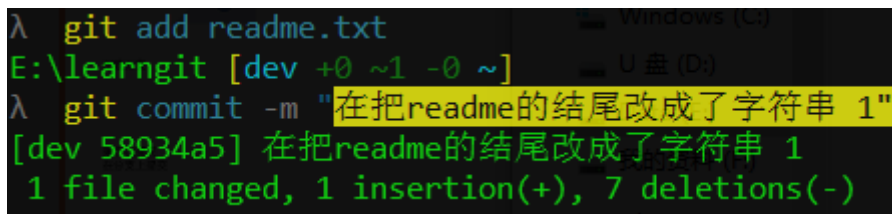
```
1 $ git checkout -b dev
```



```
λ git branch dev
E:\learngit [master ↑4]
λ git branch
dev
* master
E:\learngit [master ↑4]
λ git checkout dev
Switched to branch 'dev'
```

修改readme.txt文件，并提交一个新的commit：

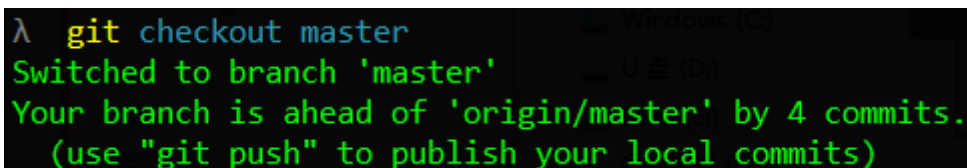
```
1 $ git add readme.txt
2 $ git commit -m "在把readme的结尾改成了字符串 1"
```



```
λ git add readme.txt
E:\learngit [dev +0 ~1 -0 ~]
λ git commit -m "在把readme的结尾改成了字符串 1"
[dev 58934a5] 在把readme的结尾改成了字符串 1
1 file changed, 1 insertion(+), 7 deletions(-)
```

现在，我们切换回 master：

```
1 $ git checkout master
```



```
λ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 4 commits.
(use "git push" to publish your local commits)
```

准备合并 dev 分支，请注意 --no-ff 参数，表示禁用 Fast forward：

```
1 $ git merge --no-ff -m "merge with no-ff" dev
```

```
λ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 8 +-----
 1 file changed, 1 insertion(+), 7 deletions(-)
```

因为本次合并要创建一个新的commit，所以加上 `-m` 参数，把commit描述写进去。

合并后，我们用 `git log --graph` 看看分支历史：

```
1 | $ git log --graph
```

```
λ git log --graph
* commit 36b48848d5beef1eaaa5e26fc8009fe6ea8ea02b (HEAD -> master)
Merge: 9ecfc52 58934a5
Author: 万章 <3003436226@qq.com>
Date: Thu Aug 15 23:39:26 2019 +0800
    merge with no-ff
* commit 58934a54f446492abe9e51e8fb798b1a4715a56b (dev)
Author: 万章 <3003436226@qq.com>
Date: Thu Aug 15 23:38:59 2019 +0800
    在把readme的结尾改成了字符串 1
* commit 9ecfc529a4361180a6198ba984fa84c8d8c56e10
Merge: 8d59b46 06f00ec
Author: 万章 <3003436226@qq.com>
Date: Thu Aug 15 22:39:35 2019 +0800
    2
* commit 06f00ec8991530769c581711ab20b82767087450
Author: 万章 <3003436226@qq.com>
Date: Thu Aug 15 22:19:29 2019 +0800
    1
```

分支策略

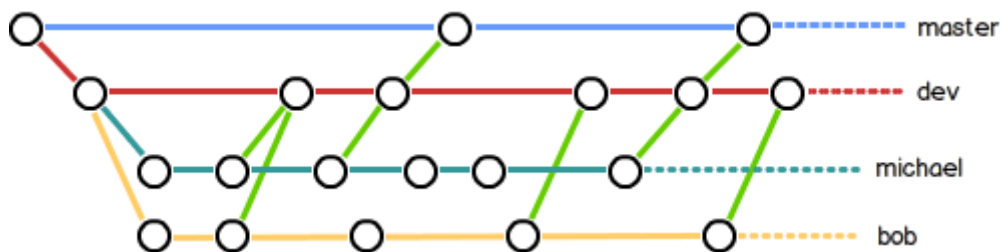
在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，`master` 分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在 `dev` 分支上，也就是说，`dev` 分支是不稳定的，到某个时候，比如1.0版本发布时，再把 `dev` 分支合并到 `master` 上，在 `master` 分支发布1.0版本；

你和你的小伙伴们每个人都在 `dev` 分支上干活，每个人都有自己的分支，时不时地往 `dev` 分支上合并就可以了。

所以，团队合作的分支看起来就像这样：



合并分支时，加上 `--no-ff` 参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而 `fast forward` 合并就看不出曾经做过合并。

Bug分支

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

预设一下代码：

```

main.js > ...
1  function add(a,b){
2      return a+b;
3  }
4
5  let obj={
6      0:"hello",
7      1:"world",
8      2:"万章",
9      3:"银时"
10 }
11
12 console.log(add(1));

```

其运行结果就是NaN, 对于我们正常运行的需求来说, 这就是个bug, 现在我们要想修改这个bug, 但是当前的dev分支又没有把工作提交

```

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

    main.js

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

```

如上所示, 我们还没提交, 但并不是你不想提交, 而是工作只进行到一半, 还没法提交。

###git stash

这是我们可以用到 Git给我们提供的 stash功能, 可以把当前的工作区先预存起来

```
λ git add main.js
E:\learngit [dev +1 ~0 -0 ~]
λ git stash
Saved working directory and index state WIP on dev: 58934a5 在把readme的结尾改成了字符串 1
```

此时再查看git status的状态

```
λ git status
On branch dev
nothing to commit, working tree clean
```

就是一个非常干净的状态

再新建 bug分区

```
λ git branch issue-101
E:\learngit [dev]
λ git checkout issue-101
Switched to branch 'issue-101'
```

修改代码

```
main.js > ...
1 function add(a,b){
2     if(a!=undefined&&b!=undefined){
3         return a+b;
4     }else{
5         return 0;
6     }
7
8 }
9
10 let obj={
11     0:"hello",
12     1:"world",
13     2:"万章",
14     3:"银时"
15 }
16
17 console.log(add(1));
```

提交代码

```
λ git commit -m "bug修复"
[issue-101 ff363ca] bug修复
1 file changed, 17 insertions(+)
create mode 100644 main.js
```

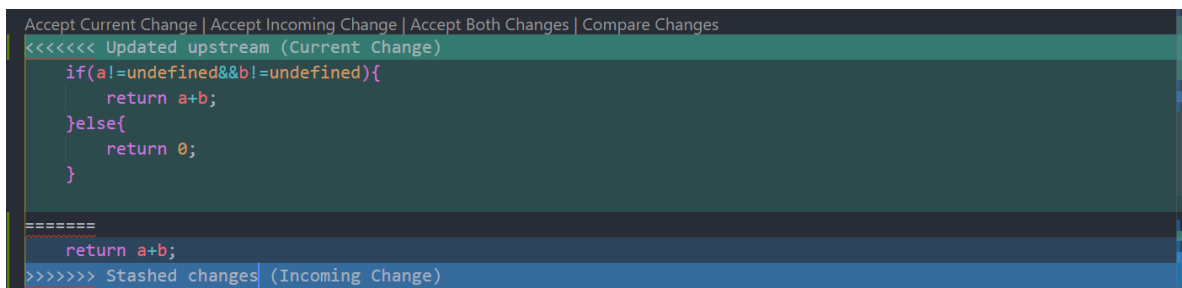
切换回 dev区, 进行合并

```
λ git merge --no-ff -m "将修改后的代码合并至dev分支" issue-101
Merge made by the 'recursive' strategy.
main.js | 17 ++++++
1 file changed, 17 insertions(+)
create mode 100644 main.js
```

此时再恢复 stash 状态

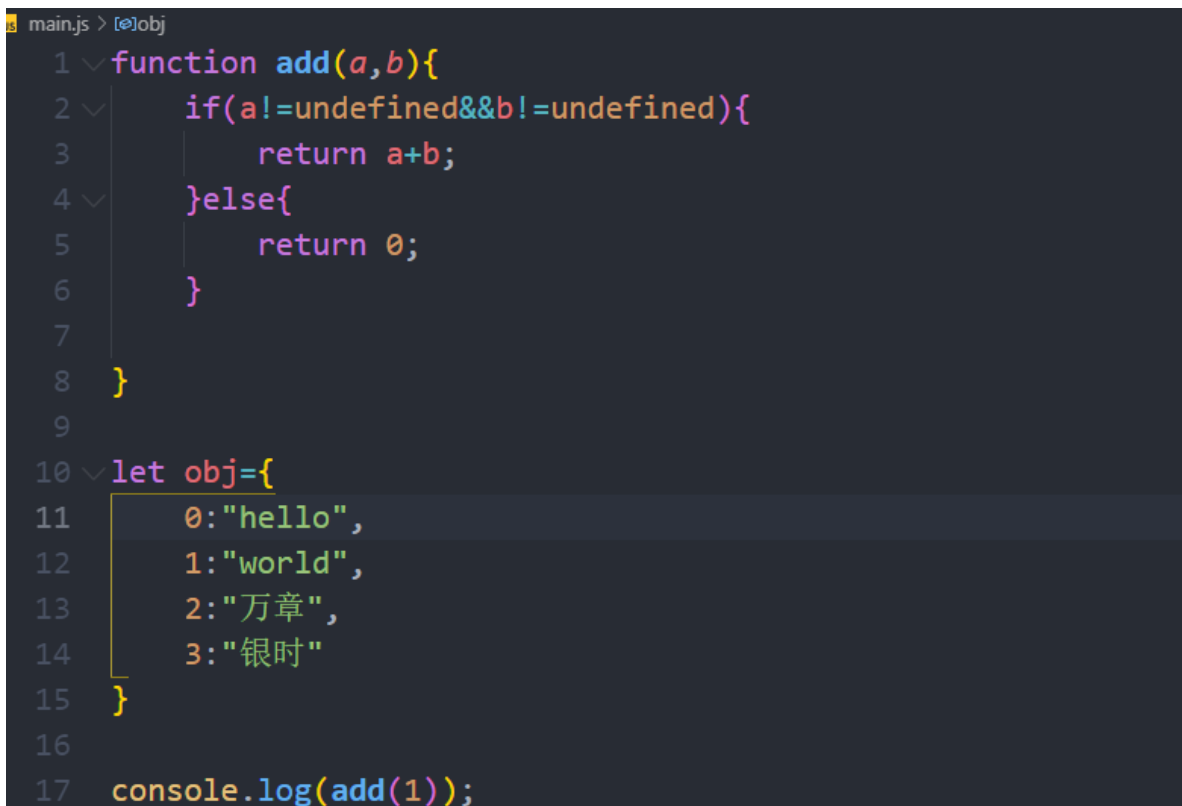
```
λ git stash list
stash@{0}: WIP on dev: 58934a5 在把readme的结尾改成了字符串 1
stash@{1}: WIP on master: 0124d98 更改readme.txt, 新增加一个key.txt文件
E:\learngit [dev]
λ git stash apply
CONFLICT (add/add): Merge conflict in main.js
Auto-merging main.js
```

就会提示 代码冲突, 因为我们恢复的状态还是之前错误版本, 而我们又添加了一个新的版本进去



```
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<< Updated upstream (Current Change)
    if(a!=undefined&&b!=undefined){
        return a+b;
    }else{
        return 0;
    }
=====
    return a+b;
>>>>>> Stashed changes (Incoming Change)
```

所以代码就冲突了, 此时我们就需要手动来进行调整, 点击接受当前变化即可



```
main.js > [Obj]
1 function add(a,b){
2     if(a!=undefined&&b!=undefined){
3         return a+b;
4     }else{
5         return 0;
6     }
7
8 }
9
10 let obj={
11     0:"hello",
12     1:"world",
13     2:"万章",
14     3:"银时"
15 }
16
17 console.log(add(1));
```

然后再次进行提交

```

λ git status
On branch dev
Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

        both added:      main.js

no changes added to commit (use "git add" and/or "git commit -a")

```

```

λ git add main.js
E:\learngit [dev +0 ~1 -0 ~]
λ git commit -m "修改后的main.js"
[dev 2e39ef4] 修改后的main.js
 1 file changed, 1 insertion(+), 1 deletion(-)
E:\learngit [dev]
λ git status
On branch dev
nothing to commit, working tree clean

```

最后回到master, 进行最后的合并

```

λ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
  (use "git push" to publish your local commits)
E:\learngit [master ↑6]
λ git merge --no-ff -m "最终合并" dev
Merge made by the 'recursive' strategy.
  main.js | 17 ++++++
  1 file changed, 17 insertions(+)
  create mode 100644 main.js
E:\learngit [master ↑10]
λ git status
On branch master
Your branch is ahead of 'origin/master' by 10 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

```

搞定

注意: 恢复工作现场时 要对 stash的内容进行删除

一是用 `git stash apply` 恢复, 但是恢复后, stash内容并不删除, 你需要用 `git stash drop` 来删除;

另一种方式是用 `git stash pop`, 恢复的同时把stash内容也删了:

feature 分支

软件开发中, 总有无穷无尽的新的功能要不断添加进来。添加一个新功能时, 你肯定不希望因为一些实验性质的代码, 把主分支搞乱了, 所以, 每添加一个新功能, 最好新建一个feature分支, 在上面开发, 完成后, 合并, 最后, 删除该feature分支。

比如我们现在决定要给当前项目加一个新功能, 就叫做Gundam

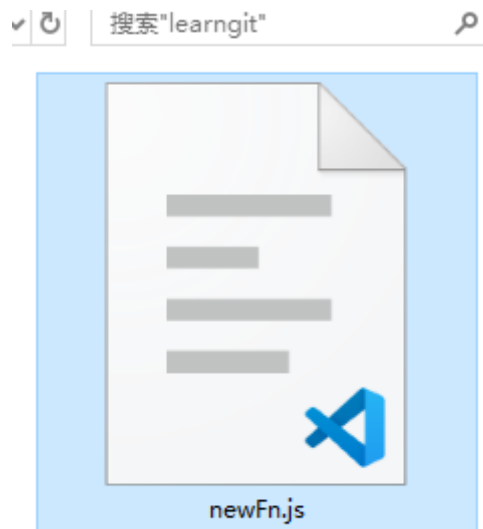
1. 开发准备

新建分支并切换分支

```
1 | $ git branch Gundam
2 | $ git checkout Gundam
```

2. 开发新功能, 新建新文件

```
1 | $ git add newFn.js
2 | $ git commit -m "开发了新功能, 增加新的newFn.js"
```



```
λ git add newFn.js
E:\learngit [Gundam +1 ~0 -0 ~]
λ git commit -m "开发了新功能, 增加新的newFn.js"
[Gundam 8b2bd25] 开发了新功能, 增加新的newFn.js
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 newFn.js
```

3. 切换回dev 进行合并即可, 就如上述的bug修复一般

```
1 | $ git checkout dev
```

但是,此时产品经理跟你说, 新功能不加了, 赶紧给我删掉, 虽说你连砍死他的心都有了, 但是还是得照着做

4. 销毁分支

```
1 | $ git branch -d Gundam
```

```
λ git branch -d Gundam
error: The branch 'Gundam' is not fully merged.
If you are sure you want to delete it, run 'git branch -D Gundam'.
```

但是此时可以看到, git给我们报了个错, 说Gundam分支没有被合并, 这也是为了防止用户啥时候手欠不小心删掉了新功能分支

如果我们确实要在未合并之前就删除掉, 那么此时我们就需要用大写的-D参数

```
1 | $ git branch -D Gundam
```

多人协作编程

当你从远程仓库克隆时, 实际上Git自动把本地的 `master` 分支和远程的 `master` 分支对应起来了, 并且, 远程仓库的默认名称是 `origin`。

要查看远程库的信息, 用 `git remote`:

```
λ git remote
origin
```

或者, 用 `git remote -v` 显示更详细的信息:

```
λ git remote -v
origin https://github.com/naturewind19/LearnGit.git (fetch)
origin https://github.com/naturewind19/LearnGit.git (push)
```

上面显示了可以抓取和推送的 `origin` 的地址。如果没有推送权限, 就看不到push的地址。

推送分支

推送分支, 就是把该分支上的所有本地提交推送到远程库。推送时, 要指定本地分支, 这样, Git 就会把该分支推送到远程库对应的远程分支上:

```
1 | $ git push origin master // 把本地的master分支 推送到远程的master分支上
2 | $ git push origin dev //把本地的dev分支 推送到远程的dev分支上
```

但是, 并不是一定要把本地分支往远程推送, 那么, 哪些分支需要推送, 哪些不需要呢?

- `master` 分支是主分支, 因此要时刻与远程同步;
- `dev` 分支是开发分支, 团队所有成员都需要在上面工作, 所以也需要与远程同步;
- `bug` 分支只用于在本地修复bug, 就没必要推到远程了, 除非老板要看看你每周到底修复了几个bug;
- `feature` 分支是否推到远程, 取决于你是否和你的小伙伴合作在上面开发。

抓取分支

多人协作时, 大家都会往 `master` 和 `dev` 分支上推送各自的修改。

现在, 模拟一个你的小伙伴, 可以在另一台电脑 (注意要把SSH Key添加到GitHub) 或者同一台电脑的另一个目录下克隆:

```
E:\MyLibrary\gitLearning\anthorLeargit\LearnGit [master ≡]
```

当你的小伙伴从远程库clone时, 默认情况下, 你的小伙伴只能看到本地的 `master` 分支

```
E:\MyLibrary\gitLearning\anthorLeargit\LearnGit [master ≡]
λ git branch
* master
```

```
E:\learngit [dev]
λ git branch
* dev
  master
```

现在, 你的小伙伴要在 dev 分支上开发, 就必须创建远程 origin 的 dev 分支到本地, 于是他用这个命令创建本地 dev 分支: (记得先在另一个本地库里面把 dev 提交上去先)

```
1 | $ git checkout -b dev origin/dev
```

```
λ git checkout -b dev origin/dev
Switched to a new branch 'dev'
Branch 'dev' set up to track remote branch 'dev' from 'origin'.
```

现在两个人就都可以在 dev 分支就行开发, 但是如果两个人写的不一样的话这就好玩了

```
E:\MyLibrary\gitLearning\anthorLeargit\LearnGit [dev ≡ +1 ~0 -0 ~]
λ git commit -m "这个分支增加了一个antherUsers.txt"
[dev d900295] 这个分支增加了一个antherUsers.txt
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 antherUsers.txt
```

```
E:\learngit [dev +1 ~0 -0 ~]
λ git commit -m "这个分支增加了一个Users.txt"
[dev ea94e99] 这个分支增加了一个Users.txt
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 Users.txt
```

两个人都来向远程库提交

```
λ git push origin dev
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 289 bytes | 289.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:naturewind19/LearnGit.git
  82afe96..d900295 dev -> dev
E:\MyLibrary\gitLearning\anthorLeargit\LearnGit [dev ≡]
```

第一个提交的就没啥问题了, 但是第二个提交的就有问题了, 这时 git 系统就提示我们, 我们需要先把另一个用户提交的数据 pull 到本地进行合并之后, 再来提交

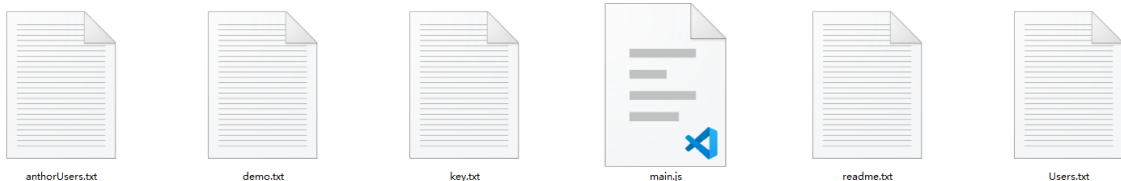
```
E:\learngit [dev]
λ git push origin dev
To https://github.com/naturewind19/LearnGit.git
 ! [rejected]      dev -> dev (fetch first)
error: failed to push some refs to 'https://github.com/naturewind19/LearnGit.git'
hint: Updates were rejected because the remote contains work that you do
hint: not 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.
```

先把内容扒下来

有多种方法

```
1 $ git pull origin "分支名" // 此时会从远程库里面抓取 符合"分支名"的库，并且与本地合并
2
3 $ git branch --set-upstream-to="远程分支名" "本地分支名"
4 ## git branch --set-upstream-to=origin/dev dev
```

```
λ git pull origin dev
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1), pack-reused 0
Unpacking objects: 100% (2/2), done.
From https://github.com/naturewind19/LearnGit
* branch            dev            -> FETCH_HEAD
   82afe96..d900295  dev            -> origin/dev
Merge made by the 'recursive' strategy.
 authorUsers.txt | 0
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 authorUsers.txt
```



再提交

```
λ git push origin dev
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 550 bytes | 550.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/naturewind19/LearnGit.git
   d900295..b7d5d8e  dev -> dev
```

因此，多人协作的工作模式通常是这样：

1. 首先，可以试图用 `git push origin <branch-name>` 推送自己的修改；
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 没有冲突或者解决掉冲突后，再用 `git push origin <branch-name>` 推送就能成功！

如果 `git pull` 提示 `no tracking information`，则说明本地分支和远程分支的链接关系没有创建，用命令 `git branch --set-upstream-to <branch-name> origin/<branch-name>`。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

小结

- 查看远程库信息，使用 `git remote -v`；

- 本地新建的分支如果不推送到远程，对其他人就是不可见的；
- 从本地推送分支，使用 `git push origin branch-name`，如果推送失败，先用 `git pull` 抓取远程的新提交；
- 在本地创建和远程分支对应的分支，使用 `git checkout -b branch-name origin/branch-name`，本地和远程分支的名称最好一致；（或是直接）
- 建立本地分支和远程分支的关联，使用 `git branch --set-upstream="远程分支名" "本地分支名"`；
- 从远程抓取分支，使用 `git pull`，如果有冲突，要先处理冲突。