pro git读书笔记

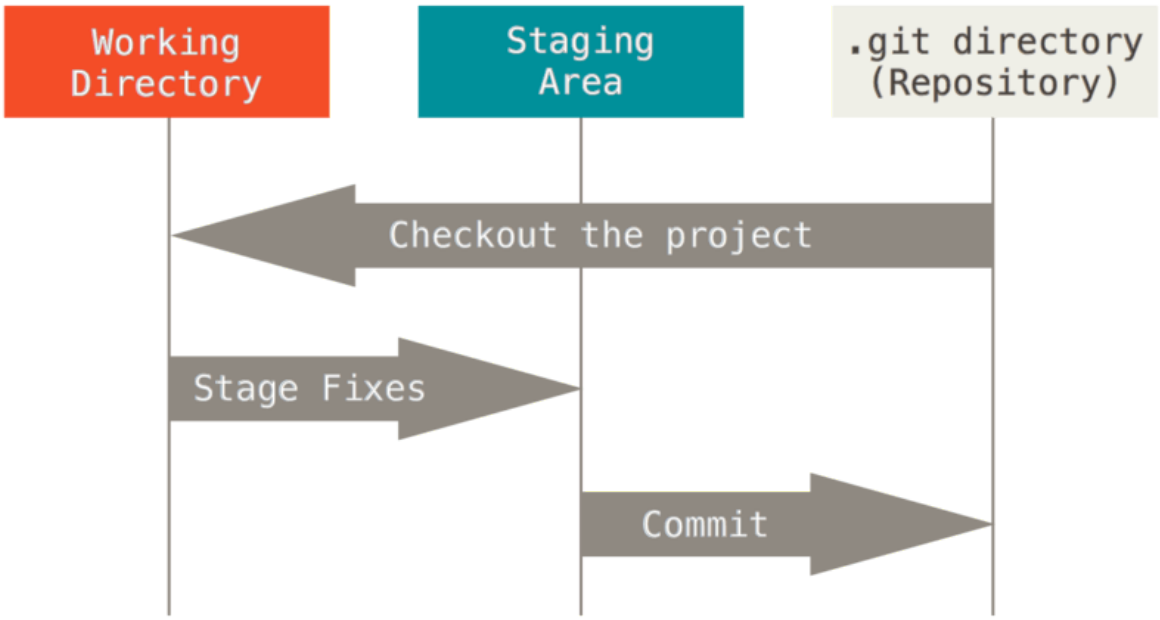
起步

三种区域以及三种状态

Git 项目有~~*三个工作区域*~~的概念：**工作目录**，**暂存区域**以及**Git 仓库**

- 工作目录是我们用来**修改文件**的目录，是对项目的某个版本独立提取出来的内容
- 暂存区域是用来**记录快照**的地方，暂存区域是一个文件，保存了下次将提交的文件列表信息，一般在 Git 仓库目录中，但是有必要单独提出来，因为其和仓库又有明显的区别， 也被称作`'索引''
- Git仓库是**存储文件**的地方，这个地方有一系列的快照，记录了文件的状态，是 Git 用来保存项目的元数据和对象数据库的地方

Git之中的文件有三种状态，文件可能处于其中之一：**已提交（committed）**、**已修改（modified）**和**已暂存（staged）**。*已提交表示数据已经安全的保存在本地数据库中*。 已修改表示修改了文件，但还没保存到数据库中。 这三种状态其实并不重要，重要的是三个工作区域的概念，已提交表示`git commit -m'xxx'`，文件保存到仓库了，已暂存表示文件已经执行了`git add file`，**已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中**，已修改表示对于本地工作目录之中的文件，进行了修改，是我们普通创作的基本操作



基本的 Git 工作流

- 在工作目录中修改文件
- 暂存文件，将文件的快照放入暂存区域
- 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录

初次运行 Git 前的配置

成功安装git后，会有一个`git config`的文件控制 Git 外观和行为的配置变量

用户信息

当安装完 Git 应该做的第一件事就是设置你的用户名称与邮件地址。 **每一次 Git 提交都会使用这些信息**，并且它会写入到每一次提交中，不可更改，需要说明的是，**如下的配置都是在bash命令行之中执行的，而不是在`git config`文件之中**

```
$ git config --global user.name "hello"
$ git config --global user.email 123456789@qq.com
```

如果使用了 `--global` 选项，那么该命令只需要运行一次，这样是一种全局配置，如果要针对特定项目使用不同的用户名称与邮件地址时，可以在那个项目目录下运行没有 `--global` 选项的命令来配置

文本编辑器

Git 会使用操作系统默认的文本编辑器，通常是 Vim。 如果你想使用不同的文本编辑器，例如 Emacs，可以使用如下配置：

```
$ git config --global core.editor emacs
```

检查配置信息

如果想要检查你的配置，可以使用 `git config --list` 命令来列出所有 Git 当时能找到的配置。

```
$ git config --list
user.name=hello
user.email=123456789@qq.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
```

可以通过输入 `git config <key>`： 来检查 Git 的某一项配置

```
$ git config user.name
hello
```

获取帮助

若使用 Git 时需要获取帮助，有三种方法可以找到 Git 命令的使用手册：

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

例如，要想获得 config 命令的手册，执行

```
$ git help config
```

Git 基础

获取 Git 仓库

有两种取得 Git 项目仓库的方法。 第一种是在现有项目或目录下导入所有文件到 Git 中； 第二种是从一个服务器克隆一个现有的 Git 仓库。

在现有目录中初始化仓库

使用 Git 来对现有的项目进行管理，需要进入该项目目录并输入：

```
$ git init
```

该命令将创建一个名为 `.git` 的子目录，这个子目录含有初始化的 Git 仓库中所有的必须文件，这些文件是 Git 仓库的骨干。此时仅仅是做了一个初始化的操作，项目里的文件还没有被跟踪。在一个已经存在文件的文件夹（而不是空文件夹）中初始化 Git 仓库来进行版本控制的话，**应该开始跟踪这些文件并提交。** 可通过 `git add` 命令来实现对指定文件的跟踪，然后执行 `git commit` 提交

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

克隆现有的仓库

如果想获得一份已经存在了的 Git 仓库的拷贝，比如说github上面的某一个项目，这时就要用到 `git clone` 命令。 **Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。** *当执行 `git clone` 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。* 克隆仓库的命令格式是 `git clone [url]` 。 比如，要克隆 Git 的可链接库 libgit2，可以用下面的命令

```
$ git clone https://github.com/libgit2/libgit2
```

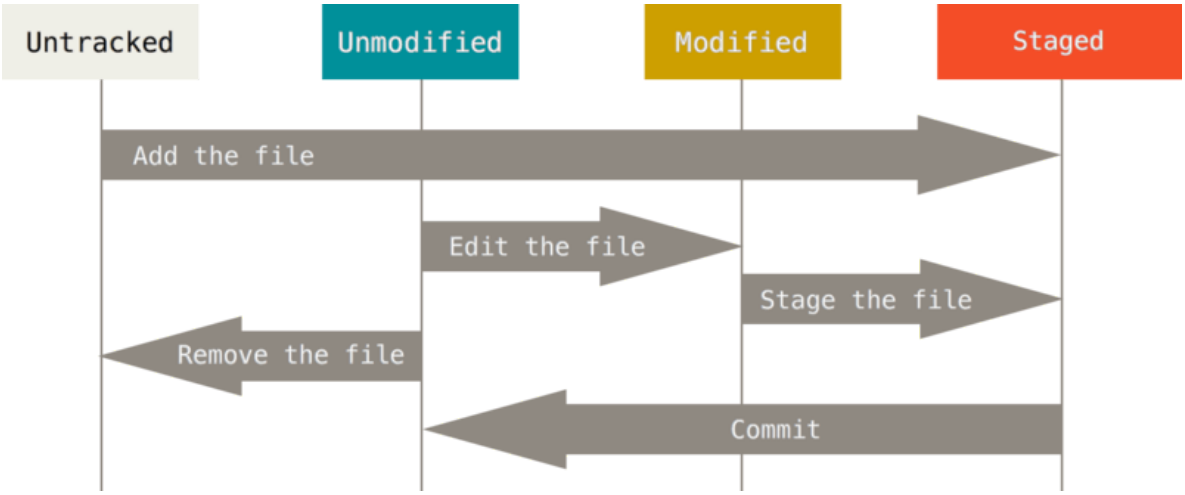
这会在当前目录下创建一个名为 “libgit2” 的目录，并在这个目录下初始化一个 `.git` 文件夹，从远程仓库拉取下所有数据放入 `.git` 文件夹，然后从中读取最新版本的文件的拷贝。 以上命令得到的本地仓库和远程仓库名称相同，如果想在克隆远程仓库的时候，**自定义本地仓库的名字**，可以使用如下命令

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

这将执行与上一个命令相同的操作，不过在本地创建的仓库名字变为 `mylibgit`。

文件状态

工作目录下的每一个文件都不外乎这两种状态：**已跟踪**或**未跟踪。 已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录**，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。 工作目录中除已跟踪文件以外的所有其它文件都属于**未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。** *初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。*编辑过某些文件之后，由于自上次提交后对它们做了修改，Git 将它们标记为已修改文件。 我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存了的修改，如此反复。所以使用 Git 时文件的生命周期如下：



要查看哪些文件处于什么状态，可以用 `git status` 命令。 如果在克隆仓库后立即使用此命令，会看到类似这样的输出

```
$ git status
On branch master
nothing to commit, working directory clean
```

这说明所有已跟踪文件在上次提交后都未被更改过。 此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，该命令还显示了当前所在分支，分支名是 “master”，这是默认的分支名。

现在在项目下创建一个新的 README 文件。 如果之前并不存在这个文件，使用 `git status` 命令，你将看到一个新的**未跟踪文件**

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

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

在状态报告中可以看到新建的 README 文件出现在 `Untracked files` 下面。 **未跟踪的文件意味着 Git 在之前的快照（提交）中没有这些文件**；*Git 不会自动将之纳入跟踪范围，除非使用 `git add README` 来说明我需要跟踪该文件，* 次吃才会将此文件纳入到跟踪范围之中，并且，此时此文件在暂存区，是一个快照，并未保存到Git 本地仓库之中永久存储

通常，我们使用 `git status` 得到的文件状态比较复杂，我们可以使用 `git status -s` 来获取简略的信息，通常有 `A` 和 `M` 两种， `A` 表示新添加的文件， `M` 表示修改过的文件

跟踪/暂存新文件

使用命令 `git add` 开始跟踪一个文件。 所以，要跟踪 README 文件，可以运行如下命令行

```
$ git add README
```

此时再运行 `git status` 命令，会看到 README 文件已被跟踪，并处于暂存状态

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

只要在 `Changes to be committed` 这行下面的，就说明是已暂存状态。 如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。**`git add` 后文件存放在暂存区**，`git add` 是个多功能命令：*可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等*。**这个命令可以理解为"添加内容到下一次提交中"。**

查看已暂存/未暂存已修改的文件

现在我们来修改一个已被跟踪的文件。 如果**修改**了一个名为 `CONTRIBUTING.md` 的**已被跟踪的文件**，然后运行 `git status` 命令，就可实现对已经暂存的文件的状态查看

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

文件 `CONTRIBUTING.md` 出现在 `Changes not staged for commit` 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。 要暂存这次更新，需要运行 `git add` 命令

查看已暂存

然后运行 `git add` 将"CONTRIBUTING.md"放到暂存区，现在两个文件都已暂存，下次提交时就会一并记录到仓库。输入 `git status` 可看到

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

假设此时，需要在 `CONTRIBUTING.md` 里加条注释， 然后保存，再运行 `git status` ，则有如下输出

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

此时的 `CONTRIBUTING.md` 文件同时出现在暂存区和非暂存区。 此时的操作是**1.刚 `git add` 过一次，2.然后立即又修改了文档，3.再又运行 `git status` 命令。**实际上此时的 Git 只不过暂存了运行 `git add` 命令时的版本（**1处的版本**）， 如果此时提交，`CONTRIBUTING.md` 的版本是你最后一次运行 `git add` 命令时的那个版本（**1处的版本**），而不是在工作目录中的当前版本（**2处的版本**）。 所以，**运行了 `git add` 之后又作了修订的文件，需要重新运行 `git add` 把最新版本重新暂存起来**

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

查看已暂存和未暂存的修改

虽然，现在我们可以使用`git status`来查看文档的状态，但是这只是针对与文档的，也就是说，我们可以看见那些文档是新增的，那些是修改的，但是具体文档之中修改了什么，新增了什么，我们无法知道，这种情况下，我们需要使用`git diff`命令来完成查看，**尽管 `git status` 已经通过在相应栏下列出文件名的方式回答了这个问题，`git diff` 将通过文件补丁的格式显示具体哪些行发生了改变**

- `git status`: 文件的增减，修改
- `git diff`: 文件内容的增加和修改
    - 查看**未暂存**修改：`git diff`，也就是**比较运行`git add`前后，修改的文件之间的差别**
    - 查看**已暂存**修改：`git diff --cached`，这个是**比较同一个文件，在本次`git add`和前一次`git add`，两次`git add`之间的差别，cached表示两个快照之间的差别**

假如再次修改 README 文件后暂存，然后编辑 `CONTRIBUTING.md` 文件后先不暂存， 运行 `status` 命令将会看到：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

查看未暂存的修改

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 `git diff`

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

此命令比较的是工作目录中当前文件和暂存区域快照之间的差异， 也就是修改之后还没有暂存起来的变化内容

若要查看已暂存的将要添加到下次提交里的内容，可以用 `git diff --cached` 命令。也就是**比较同一个文件，在本次`git add`和前一次`git add`，两次`git add`之间的差别，cached表示两个快照之间的差别**，Git 1.6.1 及更高版本还允许使用 `git diff --staged`，效果是相同的，但更好记些）

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

请注意，**git diff 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。** 所以有时候一下子暂存了所有更新过的文件后，运行 `git diff` 后却什么也没有，就是这个原因

查看已暂存的修改

引入新的例子，新建一个仓库，加入aaa.txt文档在其中，将其引入跟踪，运行一次`git add aaa.txt`，其中的内容为`aaa` ，然后修改aaa.txt，在第6行加入`bbb`，此时运行 `git status` ，**但是此时并未『再次运行』`git add`命令**，此时运行 `git status` 会看到暂存前后的两个版本。***此处要查看的是未暂存的修改***，效果如下

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   aaa.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   aaa.txt
```

现在运行 `git diff` 看暂存前后的变化，**此时查看的是未暂存的修改**

```
$ git diff
diff --git a/aaa.txt b/aaa.txt
index 7c4a013..6dad95f 100644
--- a/aaa.txt
+++ b/aaa.txt
@@ -1 +1,6 @@
-aaa
\ No newline at end of file
+aaa
+
+
+
+
+bbb
\ No newline at end of file
```

然后用 当运行了`git add`命令后，再次运行`git diff --cached` ，**查看的就是已暂存的修改，比较前后两个stage(暂存)之间的修改和变化**，--staged 和 --cached 是同义词

```
$ git diff --cached
diff --git a/aaa.txt b/aaa.txt
new file mode 100644
index 0000000..6dad95f
--- /dev/null
+++ b/aaa.txt
@@ -0,0 +1,6 @@
+aaa
+
+
+
+
+bbb
\ No newline at end of file
```

提交更新

当暂存区域已经准备妥当可以提交了。*在此之前，请一定要确认还有什么修改过的或新建的文件还没有 `git add` 过，否则提交的时候不会记录这些还没暂存起来的变化。* **这些修改过的文件只保留在本地磁盘。 所以，每次准备提交前，先用 `git status` 看下，是不是都已暂存起来了，** 然后再运行提交命令 `git commit`

```
$ git commit
```

如果想要更详细的对修改了哪些内容的提示，可以用 `-v` 选项，这会将所做的改变的 diff 输出放到编辑器中从而使你知道本次提交具体做了哪些修改。退出编辑器时，Git 会丢掉注释行，用你输入提交附带信息生成一次提交。

另外，也可以在 `commit` 命令后添加 `-m` 选项，将提交信息与命令放在同一行，如下所示：

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

可以看到，提交后它会告诉你，当前是在哪个分支（`master`）提交的，本次提交的完整 SHA-1 校验和是什么（`463dc4f`），以及在本次提交中，有多少文件修订过，多少行添加和删改过。

提交时记录的是放在暂存区域的快照。 *任何已经修改，但还未暂存(放到快照区(使用git add提交))的文件，在本次提交之中其所作的修改不会保存其中，而在下次添加到缓存区之后，才可以纳入版本管理。*
每一次运行提交操作，都是对项目作一次快照，以后可以回到这个状态，或者进行比较

跳过使用暂存区域

使用暂存区域的方式略显繁琐，Git 提供了一个跳过使用暂存区域的方式， 只要在提交的时候，给 `git commit` 加上 `-a` 选项，**Git 就会自动把所有已经跟踪过的文件暂存起来一并提交**，从而跳过 `git add` 步骤，而直接提交，**但是，这种操作只是针对已经加入追踪的文件，对于从未加入追踪的文件，是不会加入追踪的，所以仍然需要** `git add xxx`。如下，`111.txt`早已加入到追踪，修改之后，并未使用 `git add`，而是直接使用`git commit -a -m'commit without add step'`，而`11tt.txt`文件从未加入过追踪，所以直接提交的时候，对其不起作用，只能先加入追踪，以后才可以使用直接提交的命令，为了保险起见，建议使用add步骤，然后提交

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   111.txt

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

        11tt.txt

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

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit -a -m'commit without add step'
[master 9d5b684] commit without add step
 1 file changed, 3 insertions(+)
```

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        11tt.txt

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

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git log
commit 9d5b6842302a0b7d3fcb4d58f694cf19a9530972 (HEAD -> master)
Author: hello <123456789@qq.com>
Date:   Thu Dec 28 14:38:07 2017 +0800

    commit without add step
```

移除文件

要从 Git 仓库中移除某个文件，就必须要从已跟踪文件清单中移除（确切地说，是从暂存区域移除），然后提交。 可以用 `git rm` 命令完成此项工作，并连带从工作目录中删除指定的文件，**在下一次提交之后，删除的文档就不会出现在跟踪文件清单中，但是当删除文件未提交时，仍然受跟踪，并且是 `deleted` 状态。**注意的一点是，**当使用了 `git rm` 命令删除了某一文件之后，其会立即进入已追踪等待提交的状态，经过实验有无 `git add .` 都是可以的，『.』表示当前目录的所有文件**，。如下例子中，删除了`123.txt`文件，并且提交之后显示此分支是clean的

```
hello@PC-HELLO  MINGW64 /e/Codes/gittest (master)
$ ls
11tt.txt  123.txt  aaa.txt  odf.txt  ppp.txt  tt.py

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git rm 123.txt
rm '123.txt'

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit -m'delete'
[master 72d0b8b] delete
 1 file changed, 1 deletion(-)
 delete mode 100644 123.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
nothing to commit, working tree clean
```

移动文件

git中的移动文件，其实也就是改名字+移动两个功能，使用`mv`命令来完成，格式是`$ git mv file_from file_to`，可以同一层级改名，也可以不同层级改名+移动，运行 `git mv` 就相当于运行了如下3条命令

```
$ mv README.md README
$ git rm README.md
$ git add README
```

下面两个例子分别是不同层级的移动和同级移动，`s1`是git仓库下的一个文件夹

```
# eg.1
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ ls
11tt.txt  aaa.txt  hello.c  p1.txt  s1/  tt.py  tt.txt  wqe.py

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git mv aaa.txt s1/123.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    aaa.txt -> s1/123.txt

# eg.2
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit -m 'move action'
[master 63475c1] move action
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename aaa.txt => s1/123.txt (100%)

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git mv tt.txt tt1.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    tt.txt -> tt1.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit -m'mv rename'
[master e56502a] mv rename
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename tt.txt => tt1.txt (100%)
```

查看提交历史

最基本的查看提交历史的命令就是`git log`了，它会显示所有的历史，此时进入文本read-only模式(VIM)，需要使用基本的进入，退出等操作，这个不在话下。`git log`命令常有很多 的参数，一个常用的选项是`-p`，**用来显示每次提交的内容差异。** 加上`-2`来仅显示最近两次提交；`--stat`选项在每次提交的下面列出所有文件的增删改和文件内容的增删改情况，这是基本常用的命令，其他命令可以查询

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git log -p -2
```

```
commit e56502a144d8f157dfc60296636089ee3235337b (HEAD -> master)
Author: hello <123456789@qq.com>
Date:   Thu Dec 28 15:24:04 2017 +0800

    mv rename

diff --git a/tt.txt b/tt1.txt
similarity index 100%
rename from tt.txt
rename to tt1.txt

commit 63475c1e79f00f0b11c153a01bcb6fba26985d8a
Author: hello <123456789@qq.com>
Date:   Thu Dec 28 15:21:42 2017 +0800

    move action

diff --git a/aaa.txt b/s1/123.txt
similarity index 100%
rename from aaa.txt
rename to s1/123.txt


hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git log --stat -3
commit e56502a144d8f157dfc60296636089ee3235337b (HEAD -> master)
Author: hello <123456789@qq.com>
Date:   Thu Dec 28 15:24:04 2017 +0800

    mv rename

 tt.txt => tt1.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit 63475c1e79f00f0b11c153a01bcb6fba26985d8a
Author: hello <123456789@qq.com>
Date:   Thu Dec 28 15:21:42 2017 +0800

    move action

 aaa.txt => s1/123.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit b627e58c45f055b6e64dcf79096cf802cbe982a3
Author: hello <123456789@qq.com>
Date:   Thu Dec 28 15:20:46 2017 +0800

    213213

 s1/qqq.txt | 1 +
 1 file changed, 1 insertion(+)
```

git log 的常用选项

选项	说明
-p	按补丁格式显示每个更新之间的差异。
--stat	显示每次更新的文件修改统计信息。
--shortstat	只显示 --stat 中最后的行数修改添加移除统计。
--name-only	仅在提交信息后显示已修改的文件清单。
--name-status	显示新增、修改、删除的文件清单。
--abbrev-commit	仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。
--relative-date	使用较短的相对时间显示（比如，“2 weeks ago”）。
--graph	显示 ASCII 图形表示的分支合并历史。
--pretty	使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和 format（后跟指定格式）。

撤消操作

提交后(未修改快照)，修改提交信息

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 `--amend` 选项的提交命令尝试重新提交

```
$ git commit --amend
```

这个命令会将暂存区中的文件提交。 如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而修改的只是提交信息。如果提交后发现忘记了暂存某些需要的修改，可以像下面这样操作，(暂时没有演示，只展示命令)

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

最终只会有一个提交，也就是第二次提交将代替第一次提交的结果，如下

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git add mmm123.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit -m 'add mmm123.txt'
[master a575e22] add mmm123.txt
 1 file changed, 1 insertion(+)
 create mode 100644 mmm123.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git log -p -1
commit a575e2225c1b35411d133b1115bca6393290766e (HEAD -> master)
```

```
Author: hello <123456789@qq.com>
Date:   Thu Dec 28 17:18:20 2017 +0800

    add mmm123.txt     # 提交的信息

diff --git a/mmm123.txt b/mmm123.txt
new file mode 100644
index 0000000..6e9e156
--- /dev/null
+++ b/mmm123.txt
@@ -0,0 +1 @@
+1232132112312
\ No newline at end of file

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit --amend -m'add mmm123.txt hello'    # 修改提交的信息
[master c97ccc3] add mmm123.txt hello
 Date: Thu Dec 28 17:18:20 2017 +0800
 1 file changed, 1 insertion(+)
 create mode 100644 mmm123.txt


hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git log -p -1
commit c97ccc30c76e148bbc3ab2c07b64d10da9944f3e (HEAD -> master)
Author: hello <123456789@qq.com>
Date:   Thu Dec 28 17:18:20 2017 +0800

    add mmm123.txt hello    # 修改了提交的信息

diff --git a/mmm123.txt b/mmm123.txt
new file mode 100644
index 0000000..6e9e156
--- /dev/null
+++ b/mmm123.txt
@@ -0,0 +1 @@
+1232132112312
\ No newline at end of file
```

取消暂存文件

取消暂存文件，说明此时，文件已经被暂存起来了，也就是已经运行过了`git add xxx`命令，此时需要取消这次add，那么就需要使用`git reset HEAD <file>`命令，而且可如果一次存了多个文件，可以使用具体的文件名，一个一个的取消，使用例子如下

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        mg.txt
        pt.txt

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

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git add .

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   mg.txt
        new file:   pt.txt


hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git reset HEAD mg.txt  # 取消单个文件

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   pt.txt

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

        mg.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git add .

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   mg.txt
        new file:   pt.txt


hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git reset HEAD .    # 取消多个文件

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        mg.txt
        pt.txt

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

撤消对文件的修改

此处所讲的撤销对文件的修改指的是本地的**工作目录中的文件**，此时工作目录中的文件已经修改，但是此时还未提交到git仓库之中，此时如果我们想撤销所有对于工作目录中文件的修改，一个方法就是使用仓库中的文件，因为仓库中的文件没有修改过，所以我们可以使用它覆盖本地文件，但是这是一个很危险的动作，因为这样会让我们的工作成果，在不经意之间化为乌有，所以一定要**谨慎**！撤销的命令为`git checkout <file>`，**运行了此命令，本地文档的修改就被撤销。**比如，文档`sg.txt`的内容本来如下`123456`，提交之后，又修改`pt.txt`的内容，在第3行添加`654321 hello`

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        sg.txt

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

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git add .

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit -m'sg 123456'
[master 5257ed3] sg 123456
 1 file changed, 1 insertion(+)
 create mode 100644 sg.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   sg.txt

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

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git checkout sg.txt

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git status
On branch master
nothing to commit, working tree clean
```

远程仓库的使用

获取远程的仓库，可以使用`git clone giturl`来将远程的仓库**复制**到本地，查看远程仓库使用的是`git remote`，使用`git remote -v`，可以获取得到需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。 **origin**是 Git 给克隆的仓库服务器的默认名字。添加一个远程Git仓库使用`git remote add <shortname> <url>`命令，同时还可以指定一个轻松引用的简写，设定好简写之后，可以在命令行中使用**设定的字符串**来代替整个 URL，从远程服务器抓取仓库数据，可以使用`git fetch [remote-name]`，这样会**访问远程仓库，并且从中拉取本地还没有的数据**，获取其所有分支。如果使用`clone`

命令克隆了一个仓库，命令会自动将其添加为远程仓库并默认以 “origin” 为简写。 所以，`git fetch origin` 会抓取克隆（或上一次抓取）后新推送的所有工作。 **注意 `git fetch` 命令会将数据拉取到你的本地仓库，它并不会自动合并或修改你当前的工作。**有获取也有推送，当我们更新了本地代码，就可以将其推送到远程仓库，使用`git push [remote-name] [branch-name]`,比如想要将 master 分支推送到 `origin` 服务器时，那么运行`git push origin master`命令就可以将本地修改推送到远程服务器。如果想要查看某一个远程仓库的更多信息，可以使用 `git remote show [remote-name]` 命令。如果想要**重命名**引用的名字可以运行 `git remote rename oldname newname` 去修改一个远程仓库的简写名，远程**删除**仓库的分支可以使用`git remote rm` [branch-name]

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.

$ git remote
origin

$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)

$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)

$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit

$ git remote rename pb paul
$ git remote
origin
paul

$ git remote rm paul
$ git remote
origin
```

标签

Git 可以给历史中的某一个提交打上标签，以示重要。列出标签只需要输入 `git tag`。Git 使用两种主要类型的标签：**轻量标签**（lightweight）与**附注标签**（annotated）。一个轻量标签很像一个不会改变的分支，它只是一个**特定提交的引用。附注标签是存储在 Git 数据库中的一个完整对象。** 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；还有一个标签信息；并且可以使用 GNU Privacy Guard （GPG）签名与验证。 通常建议创建附注标签，这样就可以拥有以上所有信息；但是如果只是想用一个临时的标签，或者因为某些原因不想要保存那些信息，轻量标签也是可用的。 **轻**

量标签本质上是将提交校验和存储到一个文件中，没有保存任何其他信息。 创建轻量标签，不需要使用 `-a`、`-s` 或 `-m` 选项，只需要提供标签名字。创建**附注标签最简单的方式是当你在运行 `tag` 命令时指定 `-a` 选项，`-m` 选项指定了一条将会存储在标签中的信息**，通过使用 `git show` 命令可以看到标签信息与对应的提交信息

```
$ git tag
v0.1
v1.3


$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number


$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

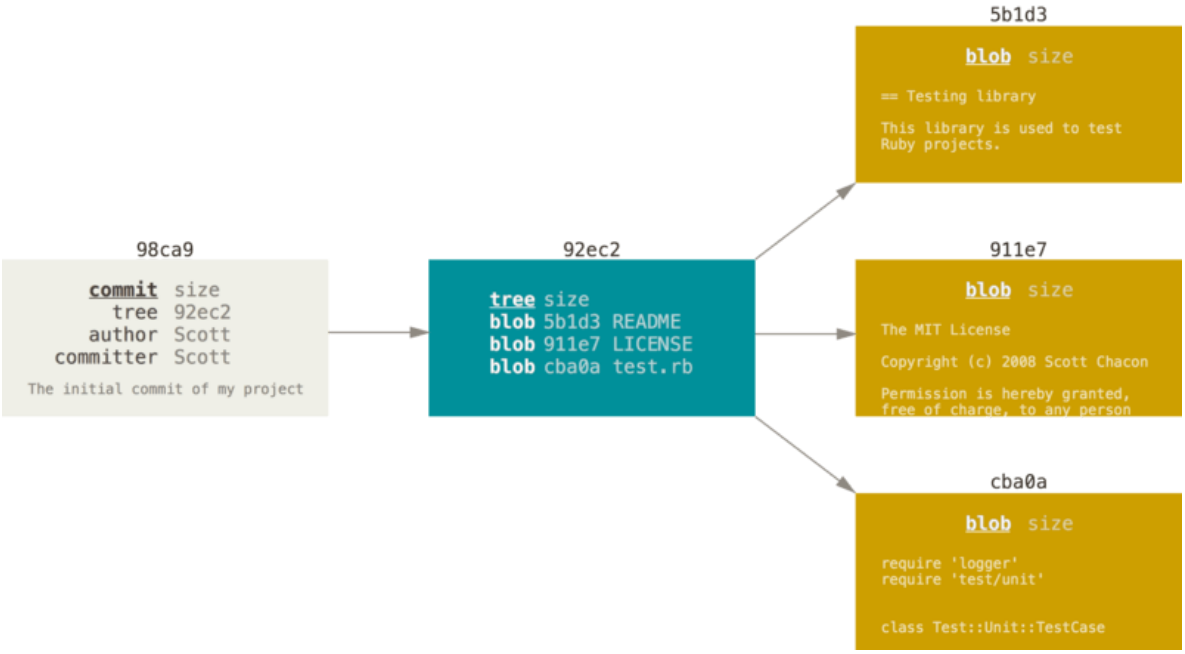
Git 分支

分支简介

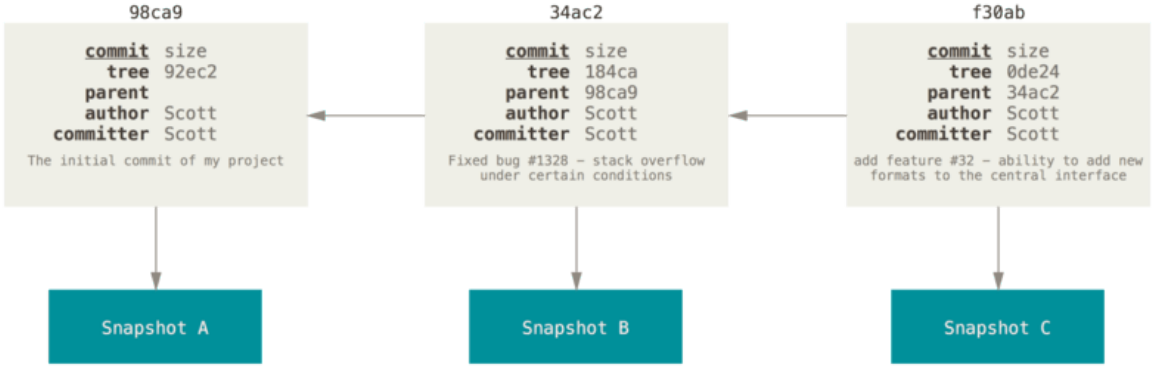
Git 保存的不是文件的变化或者差异，而是一系列不同时刻的文件快照。假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件。 暂存操作会为每一个文件计算校验和（SHA-1 哈希算法），然后会把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 对象来保存它们），最终将校验和加入到暂存区域等待提交：

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

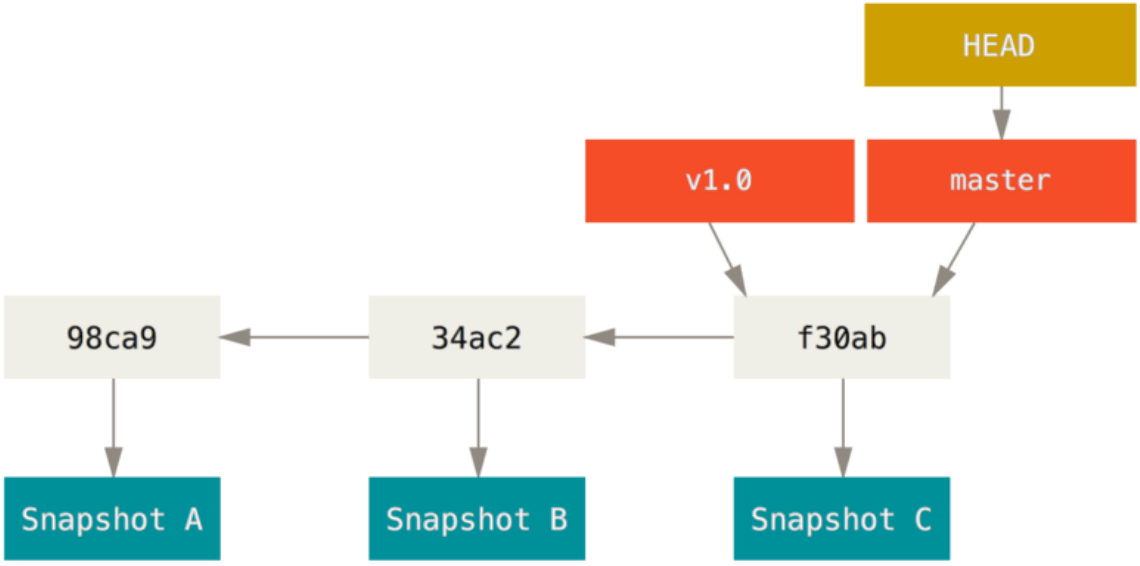
当使用 `git commit` 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和，然后在 Git 仓库中这些校验和保存为**树对象。** 随后，**Git 便会创建一个提交对象**，*它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的***指针。**如此一来，**Git 就可以在需要的时候重现此次保存的快照。**现在，Git 仓库中有五个对象：**三个 blob 对象**（保存着文件快照）、**一个树对象**（记录着目录结构和 blob 对象索引）以及**一个提交对象**（包含着指向前述树对象的指针和所有提交信息）



如果我们做些修改后再次提交，那么**这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针**

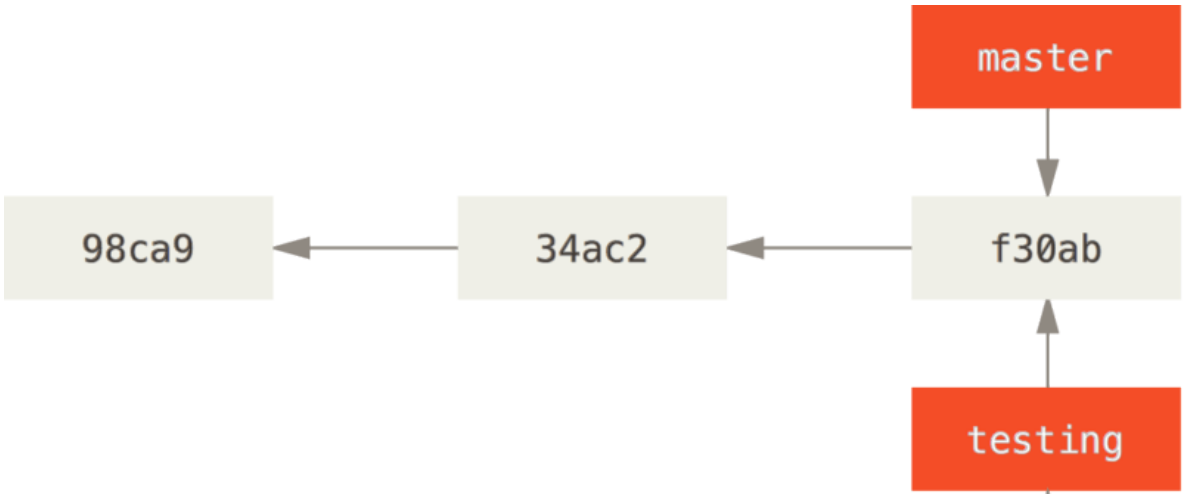


Git 的分支，其实本质上仅仅是指向提交对象的可变指针。 Git 的默认分支名字是 `master`。 在多次提交操作之后，*其实已经有一个指向最后那个提交对象的 `master` 分支。* 它会在每次的提交操作中自动向前移动

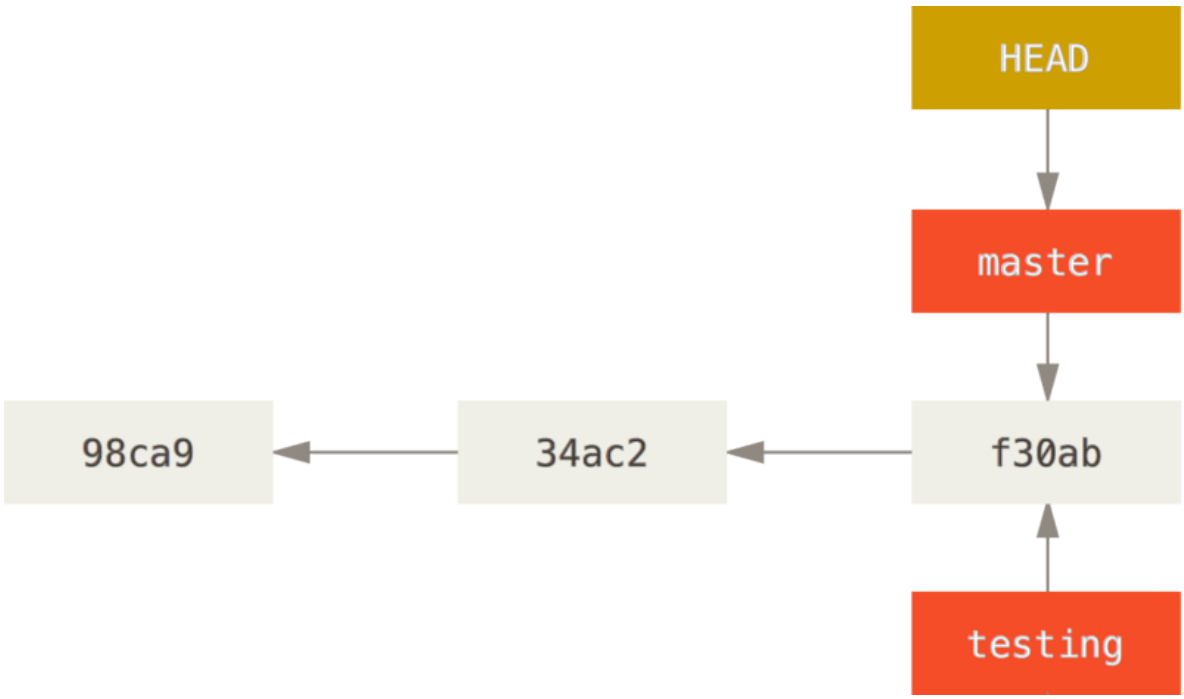


分支创建

Git 创建新分支实际相当于它只是创建了一个可以移动的新的指针。 比如，创建一个 testing 分支， 只需要使用 `git branch testing` 命令，即可创建一个新的分支，这会**在当前所在的提交对象上创建一个指针。**这会在当前所在的提交对象上创建一个指针



当有多个分支时，Git 是怎么知道当前在哪一个分支上呢? **它使用名为 `HEAD` 的特殊指针。**在 Git 中，`HEAD` **指向当前所在的本地分支**（将 `HEAD` 想象为当前分支的别名）。 在本例中，此时仍然在 `master` 分支上。 `git branch` 命令仅仅 *创建* 一个新分支，并不会自动切换到新分支中去。**此时，当前 “master” 和 “testing” 分支均指向校验和以 `f30ab` 开头的提交对象**

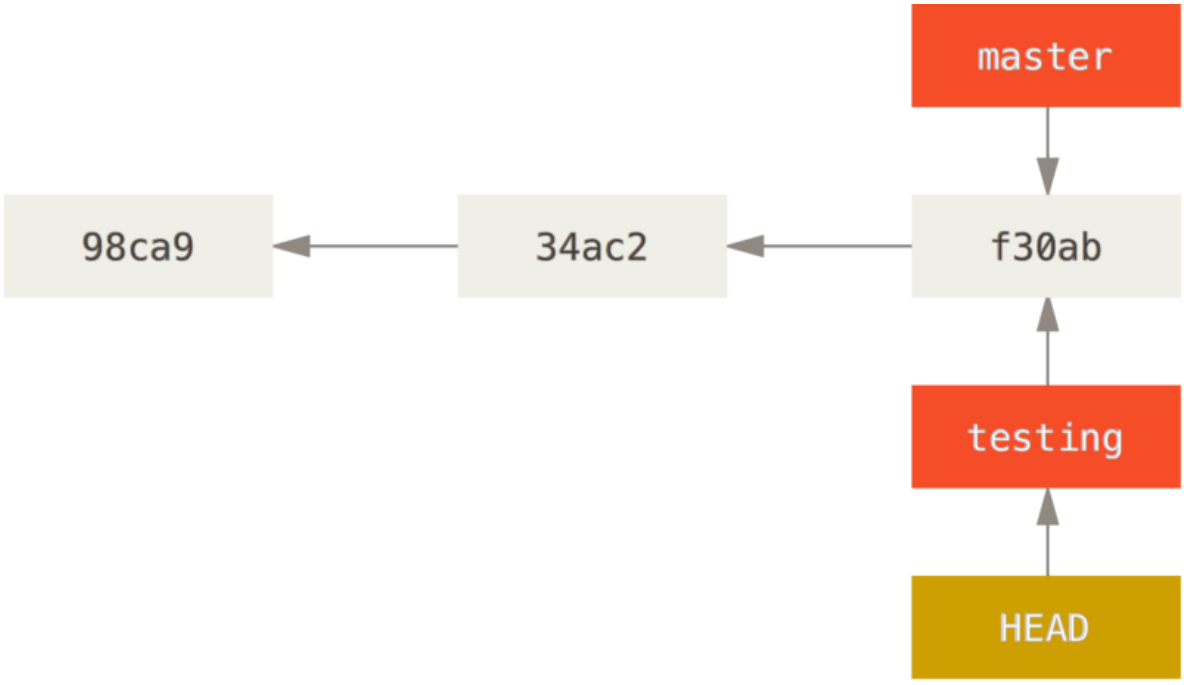


你可以简单地使用 `git log` 命令查看各个分支当前所指的对象。 提供这一功能的参数是 `--decorate`。

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

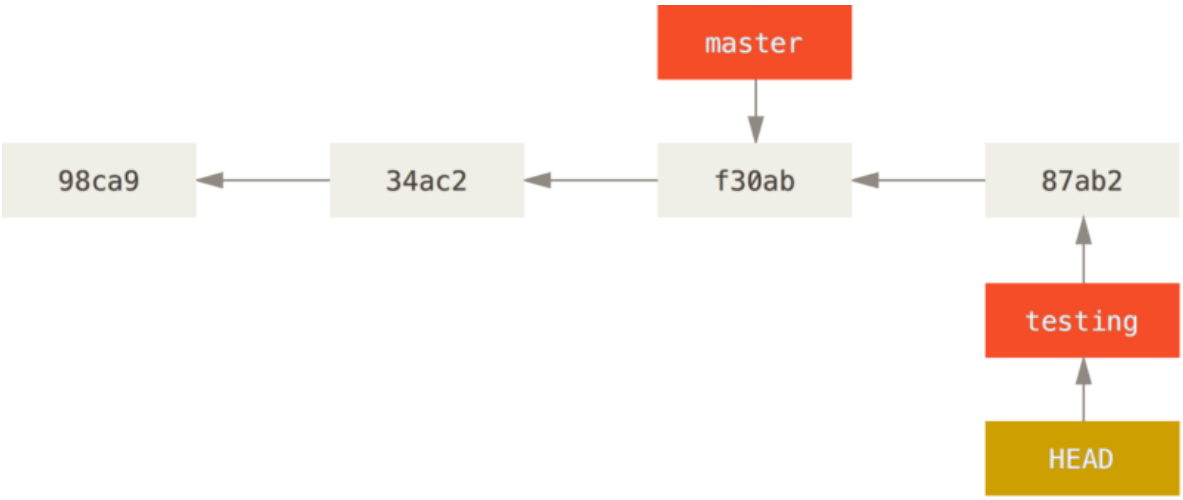
分支切换

要切换到一个已存在的分支，需要使用 `git checkout [branche name]` 命令。 `git checkout testing` 将分支切换到新创建的 `testing` 分支上，这样 `HEAD` 就指向 `testing` 分支了。就是说，**切换分支，其实是将HEAD指针的位置切换在对应的分支**，而 `HEAD` 指针本身指向当前所在的本地分支，就将分支切换过去了

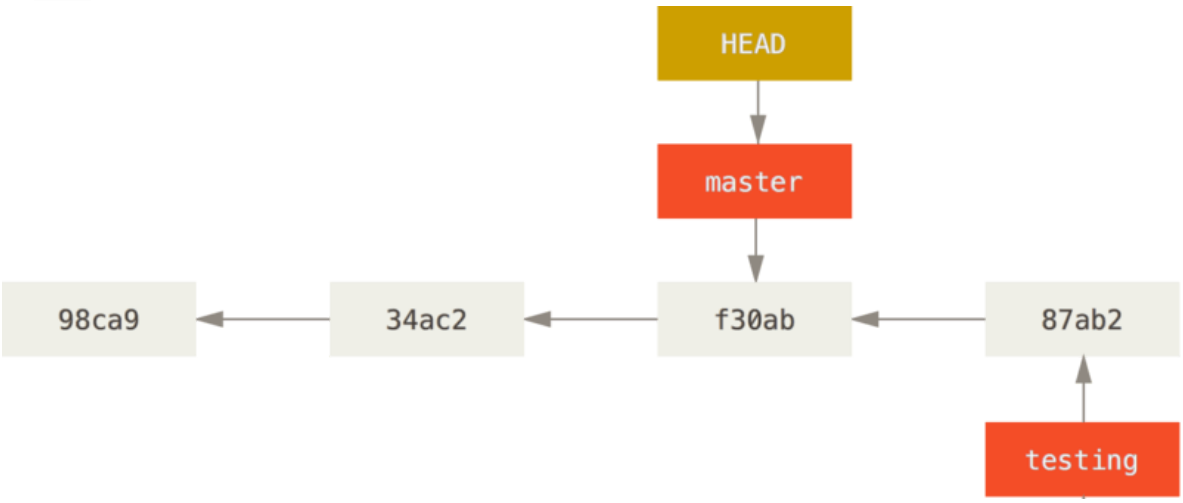


此时，我们在 `testing` 分支，修改其中的文件内容，然后再次提交

```
$ vim test.rb
$ git commit -a -m 'made a change'
```



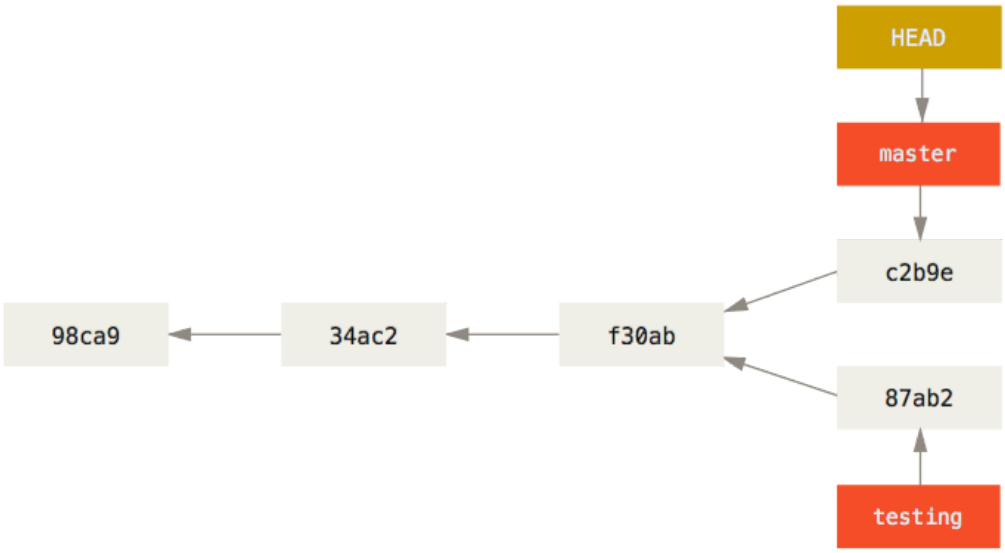
如图所示， `testing` 分支向前移动了，但是 `master` 分支却没有，它仍然指向运行 `git checkout` 时所指的对象。现在我们运行 `$ git checkout master`，切换回 `master` 分支，有如下情况，显示了 `HEAD` 指针的移动情况



这条命令做了两件事： **一是使 HEAD 指回 `master` 分支，二是将工作目录恢复成 `master` 分支所指向的快照内容。** 也就是说，现在做修改的话，项目将始于一个较旧的版本。**本质上来讲，这就是忽略 `testing` 分支所做的修改，以便于向另一个方向进行开发。** *分支切换会改变工作目录中的文件，在切换分支时，一定要注意工作目录里的文件会被改变。 如果是切换到一个较旧的分支，你的工作目录会恢复到该分支最后一次提交时的样子。* 此时我们在 `master` 分支上做些修改并提交

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

现在，**此项目的提交历史已经产生了分叉。** 因为我们在两个分支上切换，并且进行了一些修改和提交。上述两次改动针对的是不同分支，所以导致提交历史有了分叉。我们可以在不同分支间不断地来回切换和工作，在时机成熟时可以将它们合并起来。 完成这些工作需要的命令有 `branch`、`checkout` 和 `commit`



项目分叉历史

可以简单地使用 `git log` 命令查看分叉历史。 运行 `git log --oneline --decorate --graph --all` ，它会输出提交历史、各个分支的指向以及项目的分支分叉情况

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

由于 Git 的分支实质上仅是包含所指对象校验和（长度为 40 的 SHA-1 值字符串）的文件，所以它的创建和销毁都异常高效。 创建一个新分支就相当于往一个文件中写入 41 个字节（40 个字符和 1 个换行符） ，这与过去大多数版本控制系统形成了鲜明的对比，它们在创建分支时，将所有的项目文件都复制一遍，并保存到一个特定的目录。 完成这样繁琐的过程通常需要好几秒钟，有时甚至需要好几分钟。所需时间的长短，完全取决于项目的规模。而在 Git 中，任何规模的项目都能在瞬间创建新分支。 同时，由于每次提交都会记录父对象，所以寻找恰当的合并基础（即**共同祖先**）也是同样的简单和高效。 这些高效的特性使得 Git 鼓励开发人员频繁地创建和使用分支

分支的新建与合并

新建分支

新建一个分支并同时切换到那个分支上，可以运行一个带有 `-b` 参数的 `git checkout` 命令

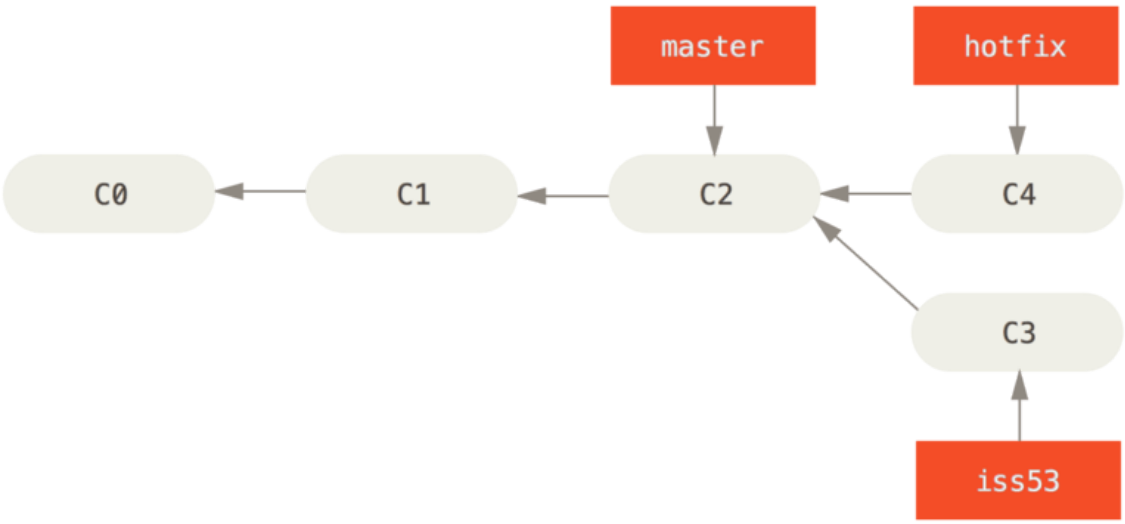
```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

它是下面两条命令的简写：

```
$ git branch iss53
$ git checkout iss53
```

当切换分支的时候，Git 会重置工作目录，使其看起来像回到了在那个分支上最后一次提交的样子。Git 会自动添加、删除、修改文件以确保此时的工作目录和这个分支最后一次提交时的样子一模一样。现在我们假如在master分支上遇到了问题，切换到master分支，并且新建一个针对此问题的分支hotfix，其形式如下

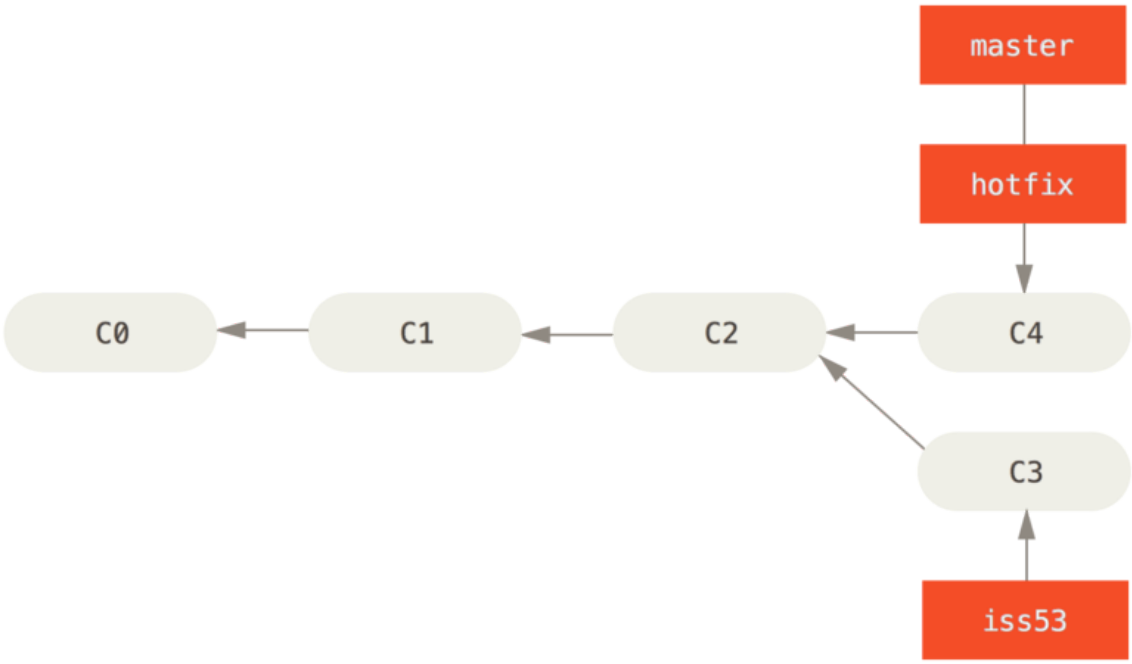
```
$ git checkout master
Switched to branch 'master'

$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```



快进合并

当问题处理好了之后，可以将其合并到master 分支上，使用`git merge`就可以完成操作。**合并分为快进合并，普通合并和有冲突的合并三种。***快进合并最为简单*，**由于`master`分支所指向的提交是当前提交（有关 hotfix 的提交）的直接上游，所以 Git 只是简单的将指针向前移动**，这种就是快进合并。换句话说，当试图合并两个分支时，*如果顺着一个分支走下去能够到达另一个分支，那么 Git 在合并两者的时候，只会简单的将指针向前推进（指针右移），因为这种情况下的合并操作没有需要解决的分歧——这就叫做“快进（fast-forward）”*。**当前分支为`A`，`git merge B`，此操作将`B`分支合并到`A`分支**，这是分支合并的方向

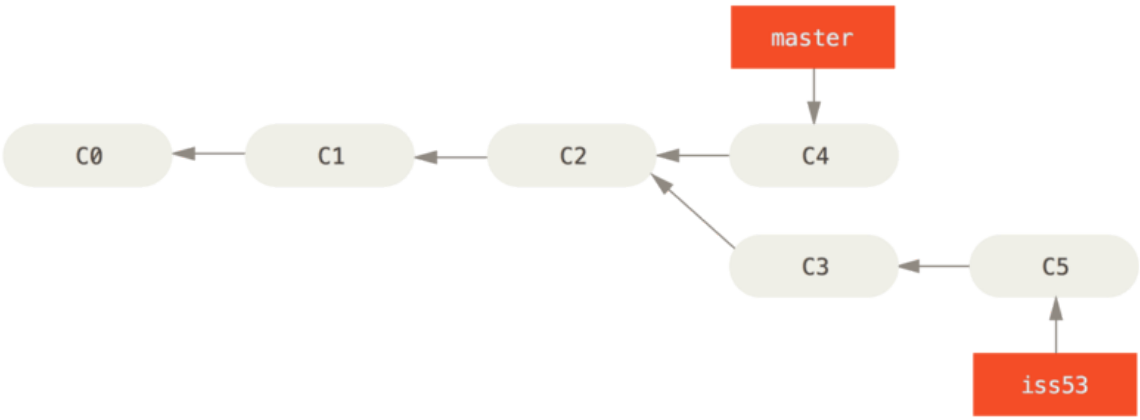
```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```



删除分支

如果要删除某一个分支，则可以使用`$ git branch -d [branch name]`语句，运行过后，提交的情况如下图

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```



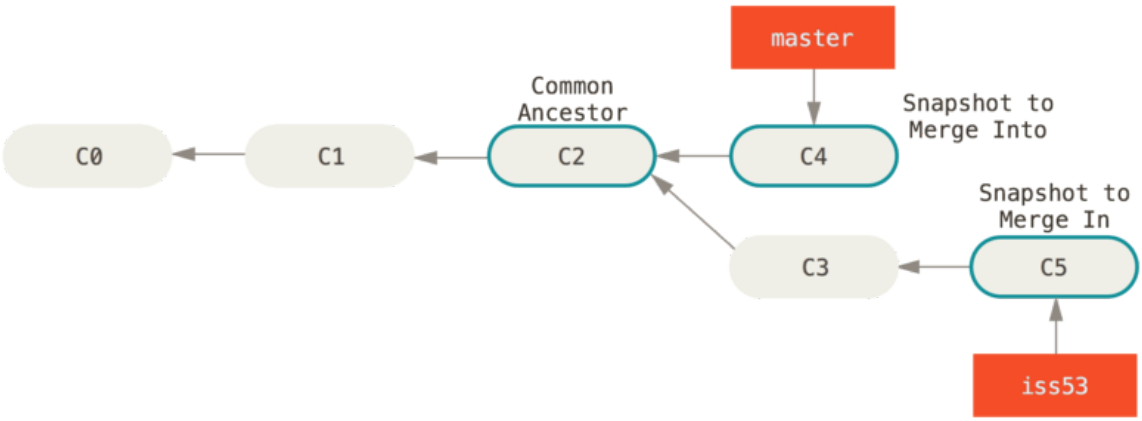
`hotfix`分支已经删除，在`hotfix`分支上所做的工作并没有包含到`iss53`分支中。 如果需要拉取`hotfix`所做的修改，可以使用`git merge master`命令将`master`分支合并入`iss53`分支，或者也可以等到`iss53`分支完成其使命，再将其合并回`master`分支。

普通合并

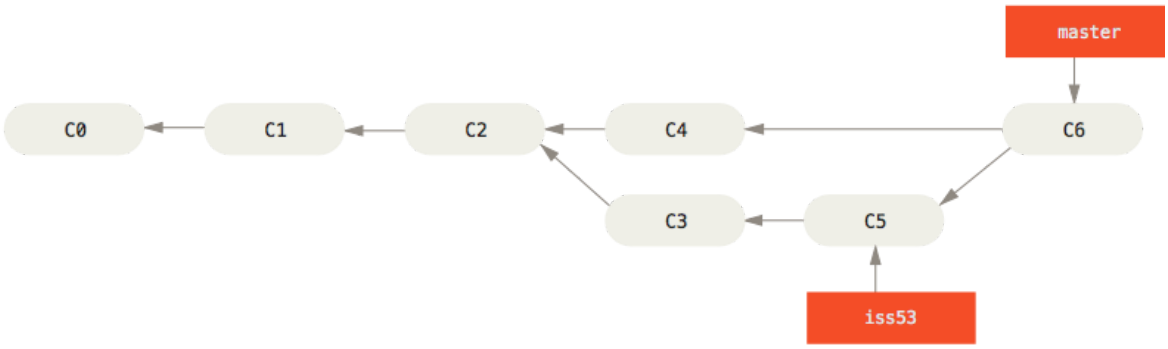
假设你已经修正了`iss53`问题，并且打算将工作合并入`master`分支。 为此需要合并`iss53`分支到`master`分支，这和之前合并`hotfix`分支所做的工作差不多。**只需要checkout想合并入的分支，然后运行`git merge`命令**，即可运行

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

这和之前合并 `hotfix` 分支的时候看起来有一点不一样。 在这种情况下，开发历史从一个更早的地方开始分叉开来（diverged）。 因为，`master` 分支所在提交并不是 `iss53` 分支所在提交的直接祖先，Git 不得不做一些额外的工作。 **出现这种情况的时，Git 会使用两个分支的末端所指的快照（`C4` 和 `C5`）以及这两个分支的工作祖先（`C2`），做一个简单的三方合并。**



和之前将分支指针向前推进所不同的是，**Git 将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。 这个被称作一次合并提交，它的特别之处在于他有不止一个父提交**

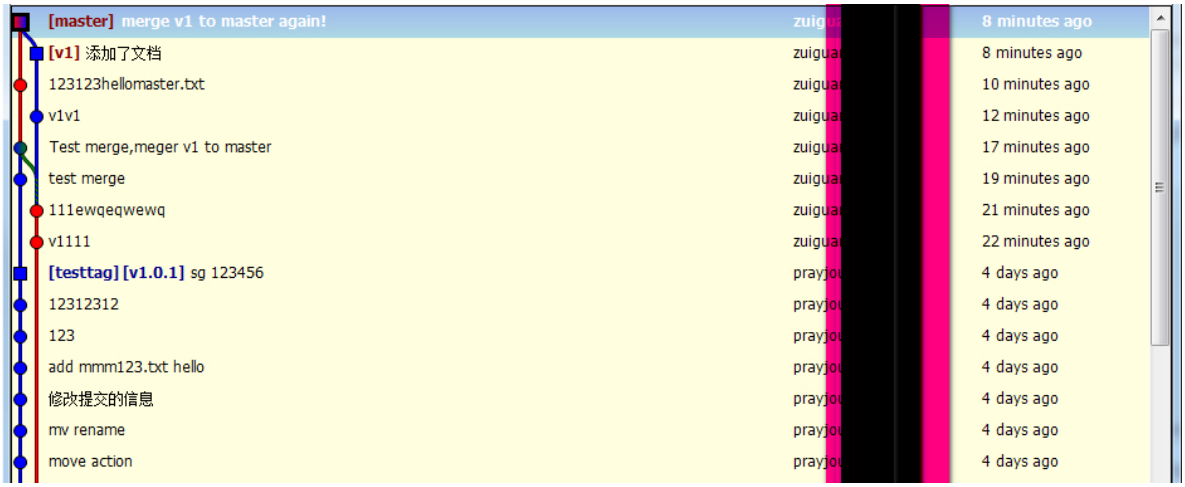


需要指出的是，**Git 会自行决定选取哪一个提交作为最优的共同祖先，并以此作为合并的基础**；这和更加古老的 CVS 系统或者 Subversion （1.5 版本之前）不同，在这些古老的版本管理系统中，用户需要自己选择最佳的合并基础。 Git 的这个优势使其在合并操作上比其他系统要简单很多，在自己的电脑上的例子如下

```
# 打印master之中的文档列表
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ ls
111333.txt  fsdafsaf.txt  hellomaster.txt  p1.txt   s1/     tt1.txt
11tt.txt    ganggang.txt  m1.txt           ppp.txt  sg.txt  v1.txt
222.txt     hello.c       mmm123.txt       pt.txt   tt.py   wqe.py
```

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git checkout v1
Switched to branch 'v1'

# 切换分支后，添加新的文档，我是一个新建的文档.txt是要添加的文档
hello@PC-HELLO MINGW64 /e/Codes/gittest (v1)
$ ls
111333.txt  222.txt       ppp.txt  v1.txt    我是一个新建的文档.txt
123.txt     fsdafsaf.txt  tt.py    v1v1.txt


hello@PC-HELLO MINGW64 /e/Codes/gittest (v1)
$ git add .
hello@PC-HELLO MINGW64 /e/Codes/gittest (v1)
$ git commit -m '添加了文档'
[v1 939bc2b] 娣诲姞浜嗘枃妗
 1 file changed, 1 insertion(+)
 create mode 100644
"\346\210\221\346\230\257\344\270\200\344\270\252\346\226\260\345\273\272\347\23
2\204\346\226\207\346\241\243.txt"

hello@PC-HELLO MINGW64 /e/Codes/gittest (v1)
$ git checkout master
Switched to branch 'master'

# 将v1合并到master，checkout master，然后将v1合并到当前的master分支
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git merge v1
Merge made by the 'recursive' strategy.
 v1v1.txt                                                                  | 1 +
 ...\252\346\226\260\345\273\272\347\232\204\346\226\207\346\241\243.txt" | 1 +
 2 files changed, 2 insertions(+)
 create mode 100644 v1v1.txt
 create mode 100644
"\346\210\221\346\230\257\344\270\200\344\270\252\346\226\260\345\273\272\347\23
2\204\346\226\207\346\241\243.txt"

# 打印master合并之后的文档列表
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ ls
111333.txt    ganggang.txt     mmm123.txt  s1/      v1.txt
11tt.txt      hello.c          p1.txt      sg.txt   v1v1.txt
222.txt       hellomaster.txt  ppp.txt     tt.py    wqe.py
fsdafsaf.txt  m1.txt           pt.txt      tt1.txt  我是一个新建的文档.txt
```

分支与合并的情况如图显示，此图之中有两次合并，最新的一次是当前的，此时这两个分支都是处于HEAD处的，但是其实，此时仍然可以在v1和master分支上面各自提交文档或者其他内容，但是一般而言，此时会选择将合并后的分支删除掉，或者保留不再去有新的更新



遇到冲突时的分支合并

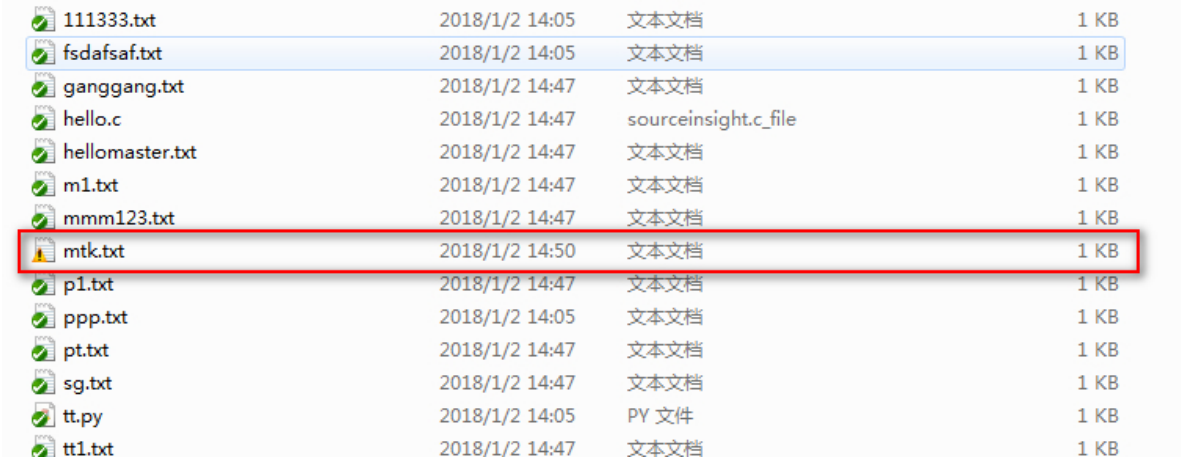
有时候合并操作不会如此顺利。 **如果在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们**

```
# master分支上面添加了mtk.txt
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ ls
111333.txt    hello.c          p1.txt   tt.py     我是一个新建的文档.txt
11tt.txt      hellomaster.txt  ppp.txt  tt1.txt
222.txt       m1.txt           pt.txt   v1.txt
fsdafsaf.txt  mmm123.txt       s1/      v1v1.txt
ganggang.txt  mtk.txt          sg.txt   wqe.py

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git add .
g
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit -m'add mtk.txt'
[master 1f58e1e] add mtk.txt
 1 file changed, 6 insertions(+)
 create mode 100644 mtk.txt

# 创建v3分支
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git branch v3

# 查看所有分支
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git branch
* master
  v1
  v3

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git checkout v3
Switched to branch 'v3'

# 修改v3分支上的mtk.txt文件
hello@PC-HELLO MINGW64 /e/Codes/gittest (v3)
$ git status
On branch v3
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   mtk.txt

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

hello@PC-HELLO MINGW64 /e/Codes/gittest (v3)
$ git add .

hello@PC-HELLO MINGW64 /e/Codes/gittest (v3)
$ git commit -m'update mtk.txt'
[v3 c5baa2c] update mtk.txt
 1 file changed, 15 insertions(+), 6 deletions(-)

# 修改master分支上的mtk.txt文件，并且提交
hello@PC-HELLO MINGW64 /e/Codes/gittest (v3)
$ git checkout master
Switched to branch 'master'

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git add .
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git commit -m 'update mtk.txt'
[master be8df15] update mtk.txt
 1 file changed, 1 insertion(+), 1 deletion(-)

# 合并v3分支到master分支
hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$ git merge v3
Auto-merging mtk.txt
CONFLICT (content): Merge conflict in mtk.txt
Automatic merge failed; fix conflicts and then commit the result.

# 此时出现冲突，用(master|MERGING)表示
hello@PC-HELLO MINGW64 /e/Codes/gittest (master|MERGING)
$
```

以上的操作，Git 做了合并，但是没有自动地创建一个新的合并提交。 Git 会暂停下来，**等待我们手动去解决合并产生的冲突。** *可以在合并冲突后的任意时刻使用 `git status` 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件*。使用`git status`来查看冲突情况，有冲突的文件使用**unmerged**来标记，且会显示其路径，其在库之中会标记出来，如下图

```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master|MERGING)
$ git status
On branch master
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:   mtk.txt

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



任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。 Git 会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。 出现冲突的文件会包含一些特殊区段，看起来像下面这个样子

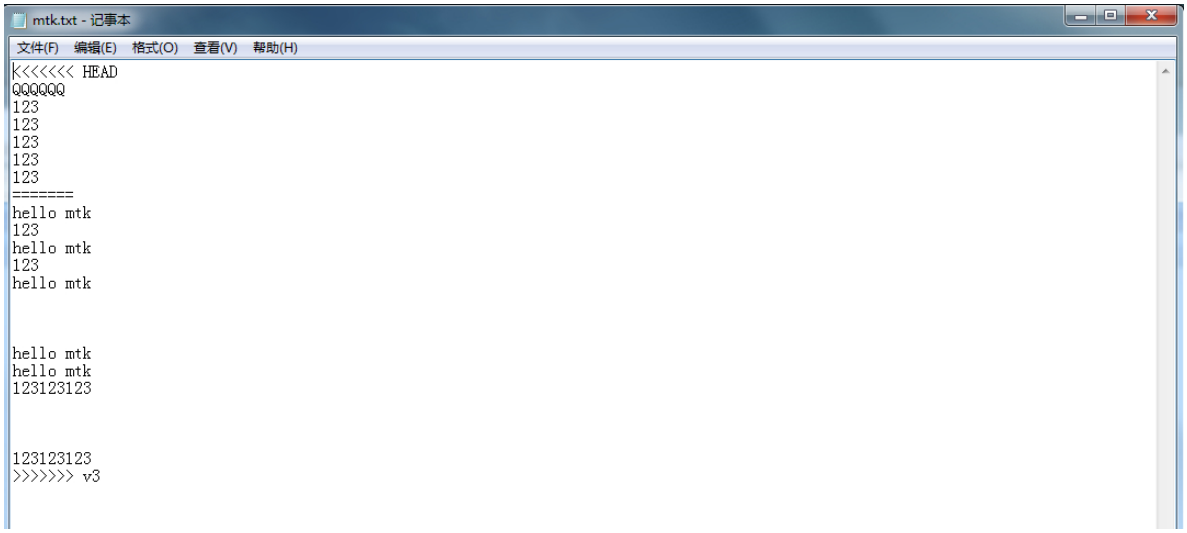
```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

这表示 `HEAD` 所指示的版本（也就是 `master` 分支所在的位置，**即当前所在的分支**，因为在运行 merge 命令的时候已经检出到了这个分支）在这个区段的上半部分（`=======` 的上半部分），而 `iss53` 分支所指示的版本在 `=======` 的下半部分。 **为了解决冲突，必须选择使用由 `=======` 分割的两部分中的一个，或者你也可以自行合并这些内容。** 例如，可以通过把这段内容换成下面的样子来解决冲突

```
<div id="footer">
please contact us at email.support@github.com
```

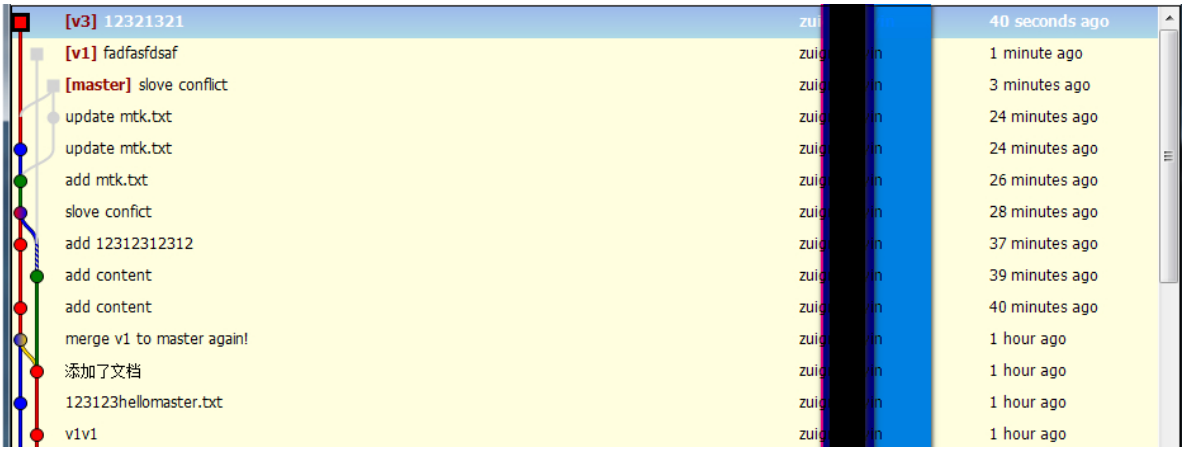
上述的冲突解决方案仅保留了其中一个分支的修改，并且 `<<<<<<<` ，`=======` ，和 `>>>>>>>` 这些行被完全删除了。 解决了所有文件里的冲突之后，对每个文件使用 `git add` 命令来将其标记为冲突已解决。 一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决

此时冲突文档`mtk.txt`之中的内容如下图，**需要我们手动解决冲突，保留 `=======` 的以上或者以下的部分，也可以自己修改，添加其他内容，然后存到暂存状态(stash)`git add`，就会消除冲突状态**




```
hello@PC-HELLO MINGW64 /e/Codes/gittest (master|MERGING)
$ git status
On branch master
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:   mtk.txt

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

# 修改完了mtk.txt,然后暂存，暂存之后，就会消除冲突状态
hello@PC-HELLO MINGW64 /e/Codes/gittest (master|MERGING)
$ git add .
# 此时已经不是冲突状态，而是修改状态了
hello@PC-HELLO MINGW64 /e/Codes/gittest (master|MERGING)
$  git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   mtk.txt

# 提交
hello@PC-HELLO MINGW64 /e/Codes/gittest (master|MERGING)
$ git commit -m'slove conflict'
[master 85157c0] slove conflict

hello@PC-HELLO MINGW64 /e/Codes/gittest (master)
$
```

当前的状态如下，此时已经又在各个分支上做了提交操作，表示各个分支已经又改变了，而非仅仅是合并时的情况



如果想使用图形化工具来解决冲突，你可以运行 `git mergetool`，该命令会为启动一个合适的可视化合并工具，并一步一步解决这些冲突。解决了冲突之后，如果对结果感到满意，并且确定之前有冲突的的文件都已经暂存了，这时你可以输入 `git commit` 来完成合并提交

分支管理

现在已经创建、合并、删除了一些分支。可以在此基础上进行分支管理。分支管理的基础是`git branch`命令。 `git branch` **命令不只是可以创建与删除分支。** 如果不加任何参数运行它，会得到当前所有分支的一个列表

```
$ git branch
  iss53
* master
  testing
```

注意 `master` 分支前的 `*` 字符：它代表现在检出的那一个分支（也就是说，当前 `HEAD` 指针所指向的分支）。 这意味着如果在这时候提交，`master` 分支将会随着新的工作向前移动。 如果需要查看每一个分支的最后一次提交，可以运行 `git branch -v` 命令

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

`--merged` 与 `--no-merged` 这两个有用的选项可以过滤这个列表中已经合并或尚未合并到当前分支的分支。 如果要查看哪些分支已经合并到当前分支，可以运行 `git branch --merged`

```
$ git branch --merged
  iss53
* master
```

因为之前已经合并了 `iss53` 分支，所以现在看到它在列表中。 在这个列表中分支名字前没有 `*` 号的分支通常可以使用 `git branch -d` 删除掉；因为我们已经将它们的工作整合到了另一个分支，所以并不会失去任何东西。查看所有包含未合并工作的分支，可以运行 `git branch --no-merged`

```
$ git branch --no-merged
  testing
```

这里显示了其他分支。 因为它包含了还未合并的工作，尝试使用 `git branch -d` 命令删除它时会失败

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

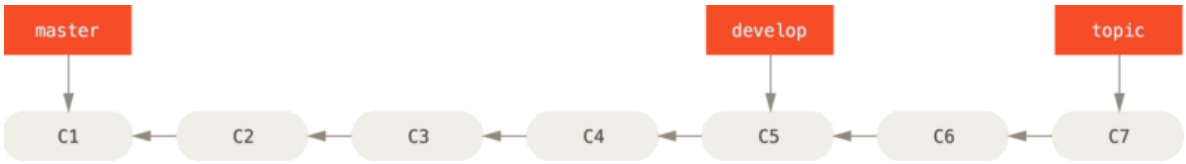
如果真的想要删除分支并丢掉那些工作，如同帮助信息里所指出的，可以使用 `-D` 选项强制删除它

分支开发工作流

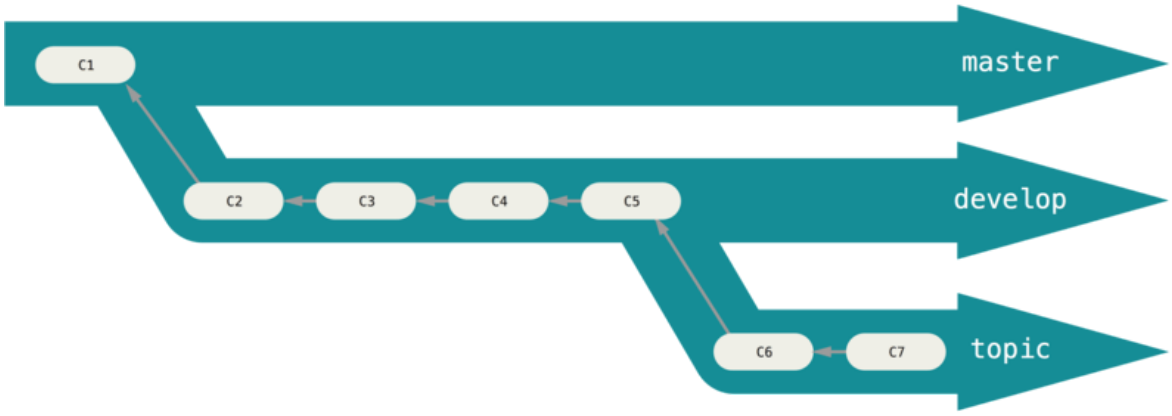
使用分支开发工作流，可以减少开发中因为代码版混乱而带来的问题，同时可以发挥git强大的分支管理功能，对于开发十分有帮助，此处介绍的是git 原始的分支开发工作流，更专业的还有git flow，可以完成对于主版本，发布版本，开发版本，修复版本等的区分，简化了开发的代码管理问题

长期分支

一般开发之中，都会有bug产生，然后需要我们去处理和修复这些问题，因此稳定分支的指针总是在提交历史中落后一大截，而前沿分支的指针往往比较靠前。一般而言，我们会fork多个分支，比如`master`,`develop`,作为平行分支，当开发的功能达到一个比较稳定的状态时，就将其合并到`master`分支，这样，在确保这些已完成的特性分支（短期分支，比如之前的 `iss53` 分支）能够通过所有测试，并且不会引入更多 bug 之后，就可以合并入主干分支中，等待下一次的发布



通常可以把各个分支的流，想象成流水线（work silos），那些经过测试考验的提交会被遴选到更加稳定的流水线上去

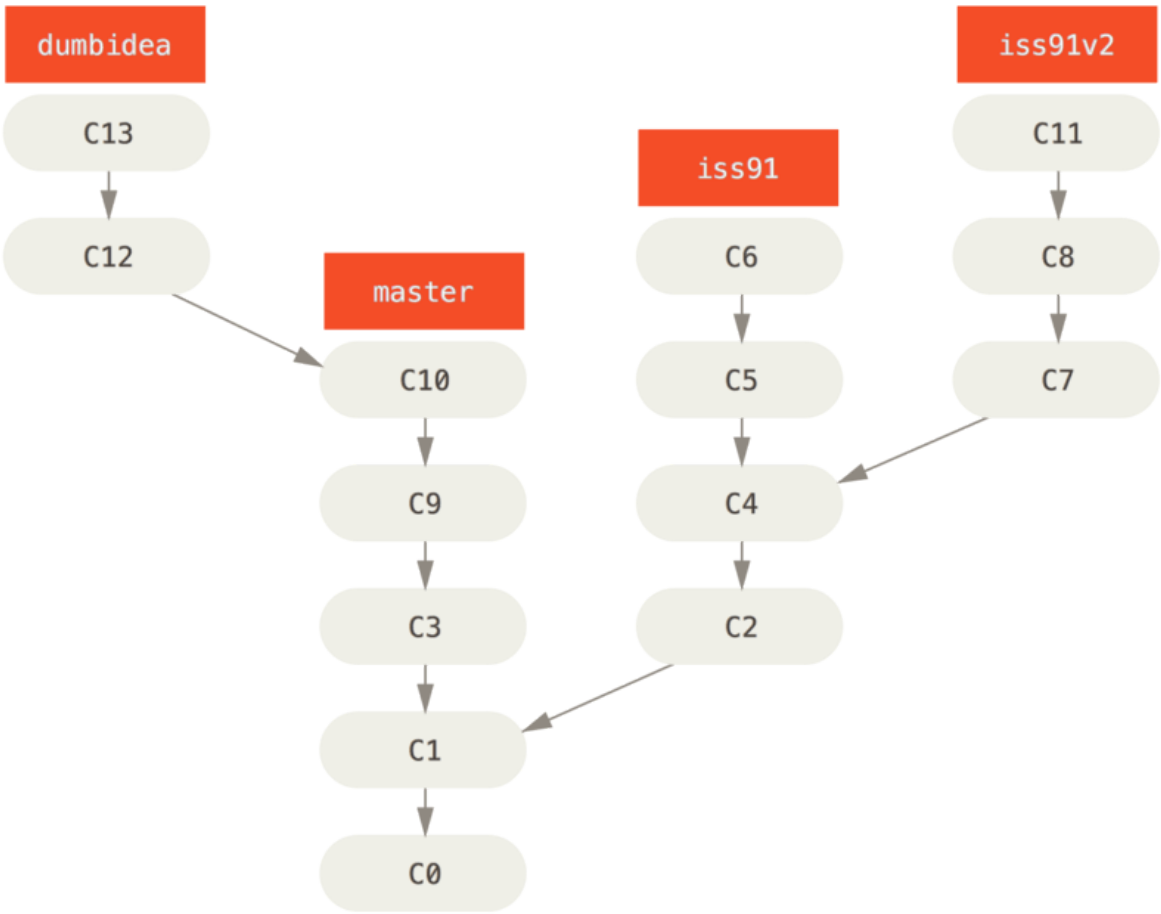


使用多个长期分支的方法并非必要，但是这么做通常很有帮助，尤其是当你在一个非常庞大或者复杂的项目中工作时，因此而言，使用`git flow`将会是一个有益的尝试

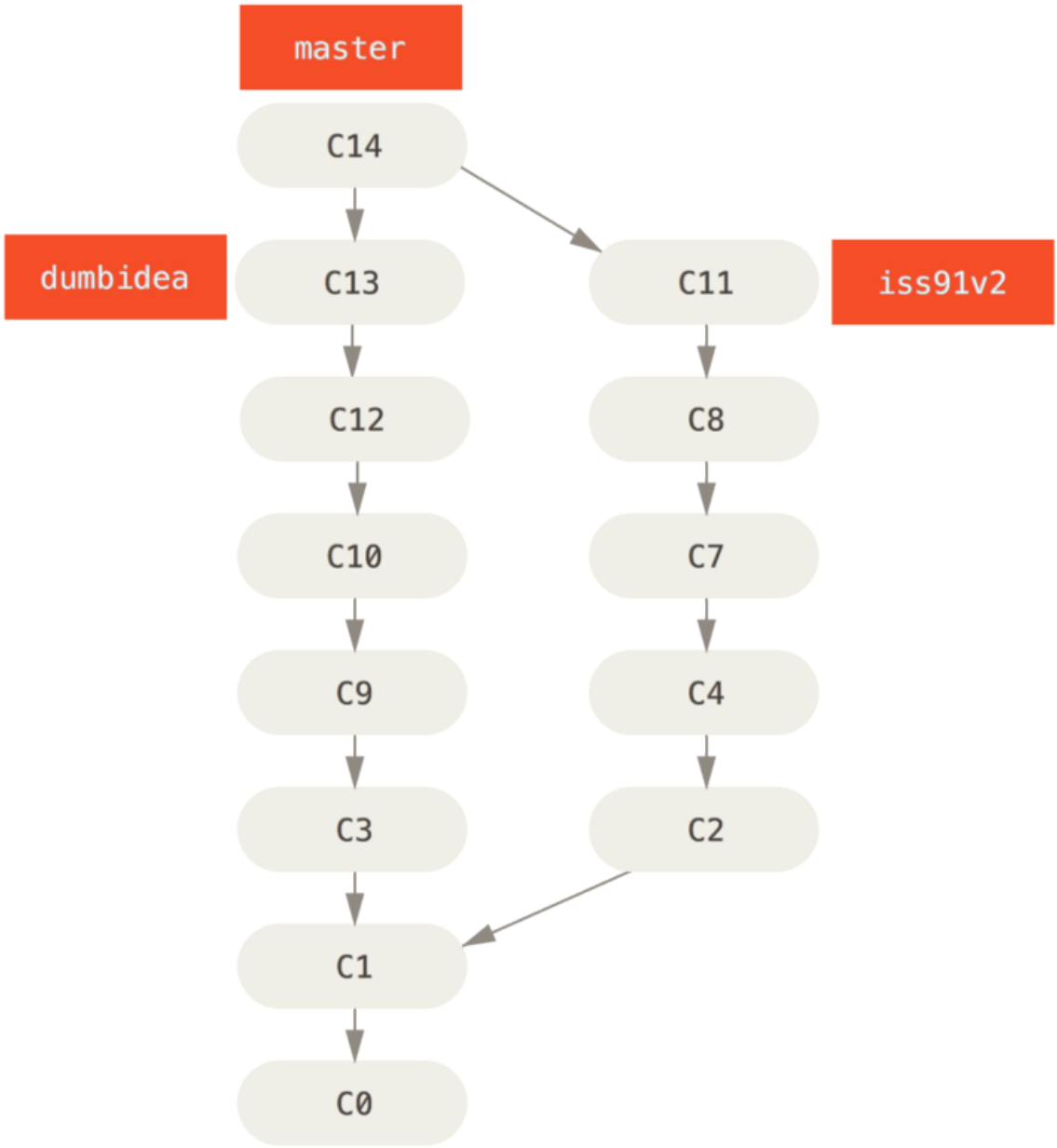
特性分支

特性分支对任何规模的项目都适用。 **特性分支是一种短期分支，它被用来实现单一特性或其相关工作。** *通常我们在一个特性分支上完成相关的功能，经过测试达到比较稳定的状态之后，会将他们合并到主干分支，然后删除删除了它们。*这在做代码审查之类的工作的时候能更加容易地看出做了哪些改动。当然特性分支也可以保留，等它们成熟之后再合并，而不用在乎它们建立的顺序或工作进度

假如，在`master`分支上工作到`C1`，这时为了解决一个问题而新建`iss91`分支，在`iss91`分支上工作到`C4`，然而对于那个问题我们又有了新的想法，于是可以再新建一个`iss91v2`分支试图用另一种方法解决那个问题，接着回到`master`分支工作了一会儿，又冒出了一个不太确定的想法，便在`C10`的时候新建一个`dumbidea`分支，并在上面做些实验。 我们的提交记录可以描述如下



现在，我们假设两件事情：决定使用第二个方案来解决那个问题，即使用在 `iss91v2` 分支中方案；另外，当我们将 `dumbidea` 分支拿给其他同事看过之后，结果发现这是个惊人之举。 这时我们可以抛弃 `iss91` 分支（即丢弃 `C5` 和 `C6` 提交），然后把另外两个分支合并入主干分支。 最终我们的提交历史看起来像下



以上的所有这一切都只发生在本地的 Git 版本库中，暂时没有和服务器发生交互

远程分支

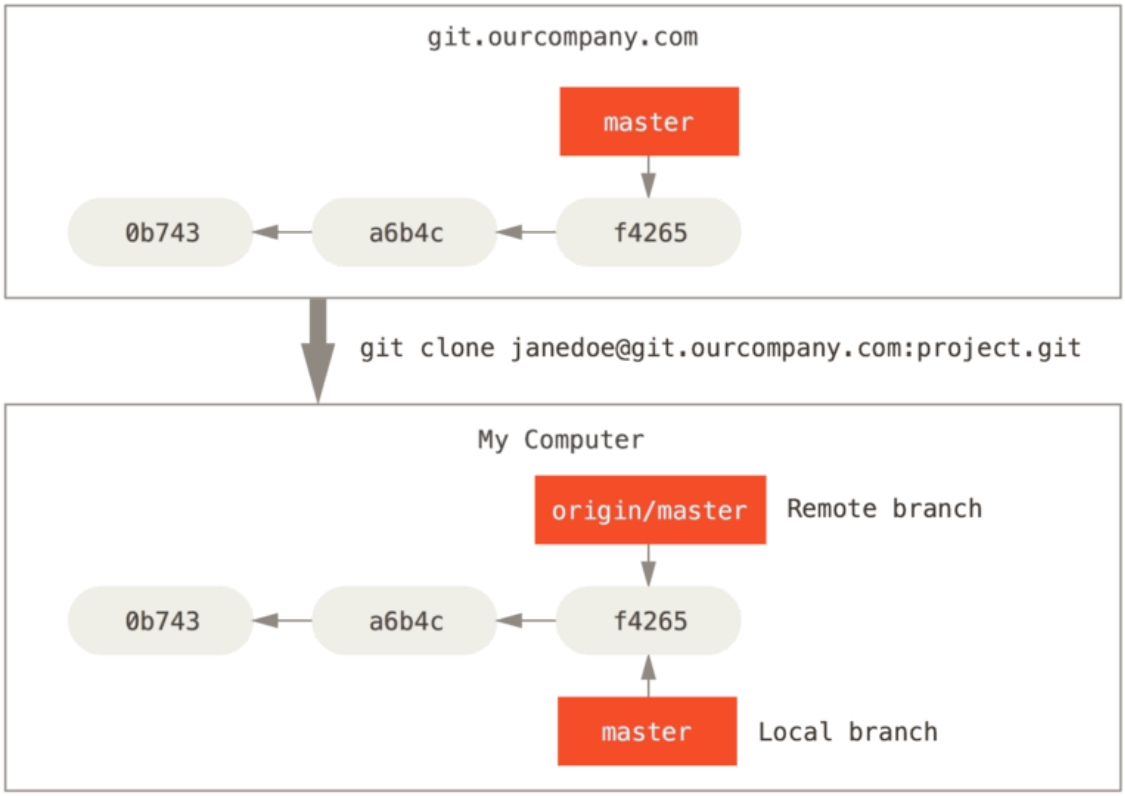
远程引用是对远程仓库的引用（指针），包括分支、标签等等。 可以通过 `git ls-remote (remote)` 来显式地获得远程引用的完整列表，或者通过 `git remote show (remote)` 获得远程分支的更多信息。一个更常见的做法是利用远程跟踪分支，**远程跟踪分支是远程分支状态的引用**，其实就是*将本地的修改，更新等一系列本地的操作，提交到远程仓库之中去*

在本地情况下，对于远程仓库的分支，**以 `(remote)/(branch)` 形式命名**，这是在本地的git系统对于远程git仓库的理解的命名。如`origin/master`。**就我的理解而言，本地仓库的分支是`master`，远程仓库的分支也是`master`,而这个`origin/master`更像是为了区分本地仓库的分支和远程仓库的分支的一个flag，是远程仓库在本地仓库的分支**

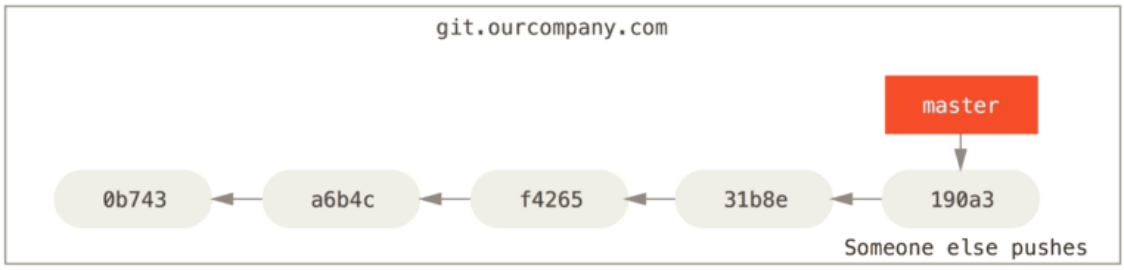
假设网络里有一个在 `git.ourcompany.com` 的 Git 服务器。**服务器上的分支为`master`，如果从这里克隆，Git 的 `clone` 命令会在本地建立两个分支，但是在未改变仓库分支内容时，这两个分支指向是一致的，一个是代表本地仓库分支的`master`，一个是代表远程仓库分支的`origin/master`**，这就有工作的基础

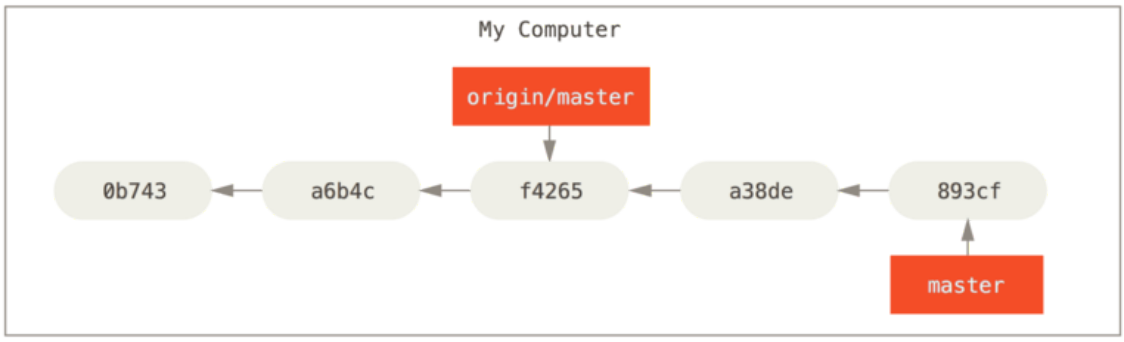
需要说明的是，可以从下图看出，本地的分支和远程的分支，仍然都是叫`master`，而这个`origin/master`的作用，就是为了标记出来本地和远程的区别，当了一个flag，相当于一个中间的标记。本地提交到远程，可以通过这个特殊的指针，区分那些文件更新了，那些是新的内容，借此来实现更新，此时这个指针会向前移动，和当前的本地分支`master`的头指针处于同一个位置处，当然，从远程拉取更新到本地，也会该表这个指针的位置

~~`“origin”`并无特殊含义远程仓库名字`“origin”`与分支名字`“master”`一样，在 Git 中并没有任何特别的含义一样。同时`“master”`是当我们运行`git init`时默认的起始分支名字，原因仅仅是它的广泛使用，`“origin”`是当你运行`git clone`时默认的远程仓库名字。如果你运行`git clone -o booyah`，那么你默认的远程分支名字将会是`booyah/master`。~~从远程`clone`仓库的示意图如下



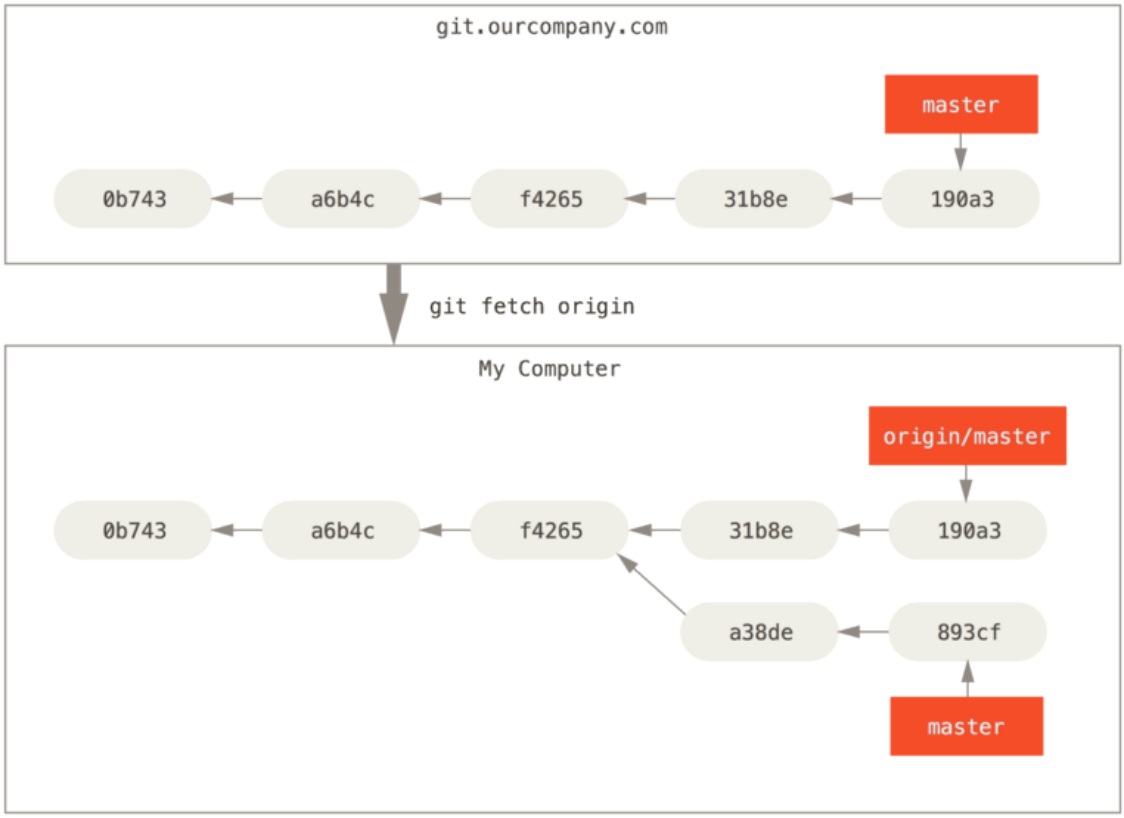
如果在本地的`master`分支做了一些工作，然而在同一时间，其他人推送提交到`git.ourcompany.com`并更新了它的`master`分支，那么我们的提交历史将向不同的方向前进。 但是，**只要不与origin 服务器连接`origin/master`指针就不会移动，`origin/master`是远程仓库在本地的分支，`master`是本地仓库的分支**

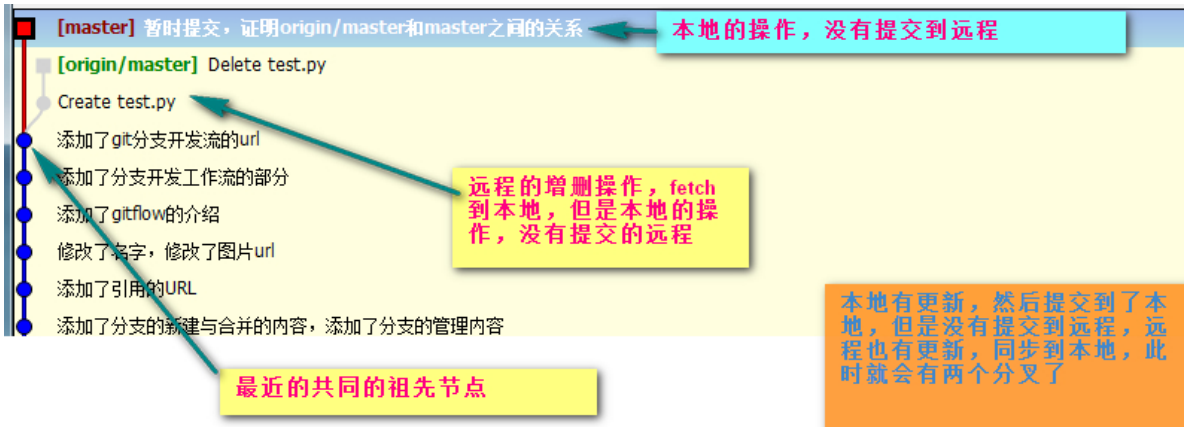




获取数据

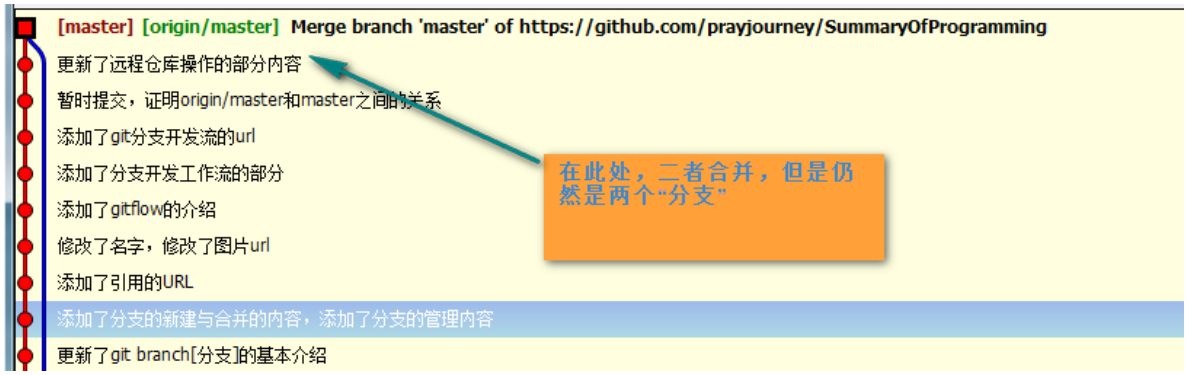
如果远程的仓库有更新，我们要将其同步到本地仓库，可以运行 `git fetch origin` 命令来获取更新。 这个命令查找 “origin” 是哪一个服务器（在本例中，它是 `git.ourcompany.com`），从中抓取本地没有的数据，并且更新本地数据库，移动 `origin/master` 指针指向新的、更新后的位置。**可以看出，当本地和远程有不同的更新时候，其会在最近的共同祖先节点(`f4265`)处分叉**,下面第二张图片说明了`origin/master`和`master`之间的关系





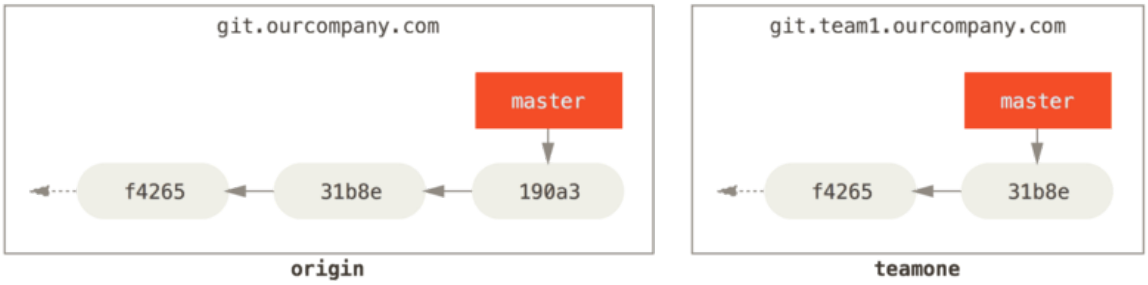
origin/master和master的关系

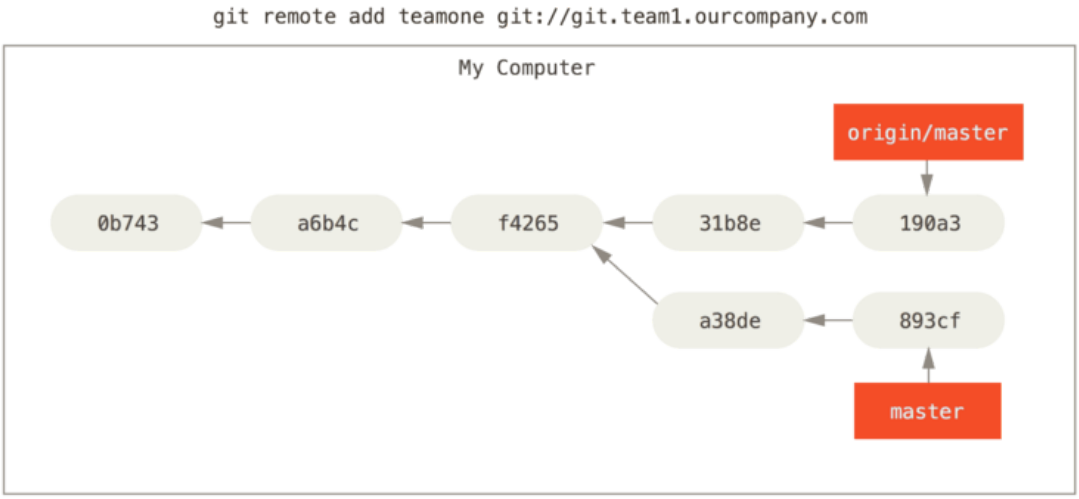
从远程仓库克隆到本地，Git 的 `clone` 命令会在本地建立两个分支，一个是代表本地仓库分支的 `master`，一个是代表远程仓库分支的 `origin/master`，当本地修改，提交到远程时，一般这两个指针指向同一个HEAD，但是当本地和远程有不同的操作时，这两个分支的头会有不同，此时就会产生分叉，简而言之，**就是两个不同的分支，一个代表本地分支，一个代表远程分支。**在未改变仓库分支内容时，这两个分支指向是一致的。详细可见**跟踪分支**部分。下图是本地提交，二者合并后的情况



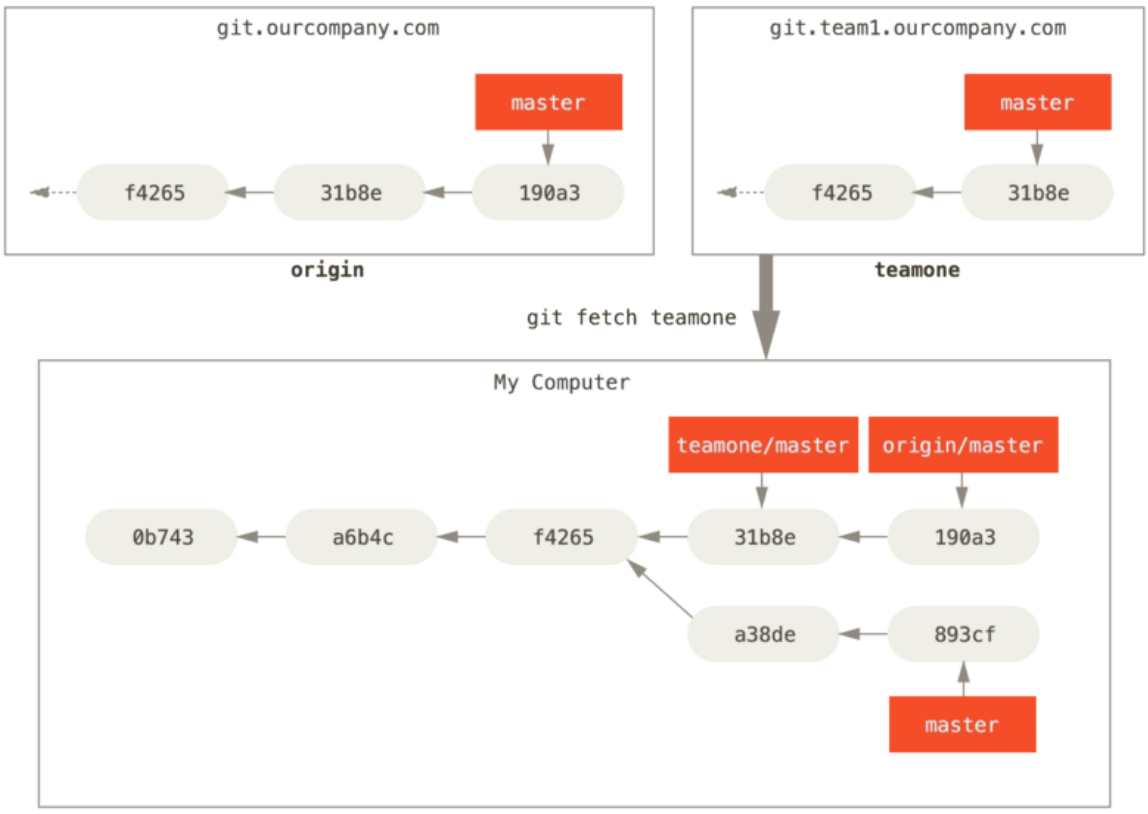
添加另一个远程仓库

为了演示有多个远程仓库与远程分支的情况，我们假定有另一个内部 Git 服务器，仅用于 sprint 小组的开发工作。 这个服务器位于 `git.team1.ourcompany.com`。 **我们可以运行 `git remote add` 命令添加一个新的远程仓库引用到当前的项目。** 将这个远程仓库命名为 `teamone`，将其作为整个 URL 的缩写





此时，*我们可以运行 `git fetch teamone` 来抓取远程仓库 `teamone` 有，而本地没有的数据。* **因为那台服务器上现有的数据是 `origin` 服务器上的一个子集**，~~所以 Git 并不会抓取数据而是会设置远程跟踪分支 `teamone/master` 指向 `teamone` 的 `master` 分支~~，这相当于新开了一个分支



推送

运行 `git push (remote) (branch)`，可以将本地的修改，推送到远程仓库之中。实验得知，如果远程仓库不存在某个分支，然后去推送，会报错，也就是说，**推送的时候，分支需要存在，即先要创建分支**

```
hello@HELLO MINGW64 /f/mydoc/forgit/gitextensions/SummaryOfProgramming/读书笔记
(master)
$ git push origin testpush
error: src refspec testpush does not match any.
error: failed to push some refs to
'https://github.com/prayjourney/SummaryOfProgramming.git'
```

跟踪分支

从一个远程跟踪分支检出一个本地分支会自动创建一个叫做“**跟踪分支**”（有时候也叫做“上游分支”）。跟踪分支是与远程分支有直接关系的本地分支。也就是`origin/master`和`master`的关系。**如果在一个跟踪分支上输入 `git pull`，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支**

当克隆一个仓库时，它通常会自动地创建一个跟踪 `origin/master` 的 `master` 分支。 也可以设置其他的跟踪分支 - 其他远程仓库上的跟踪分支，或者不跟踪 `master` 分支。 最简单的就是之前看到的例子，运行 `git checkout -b [branch] [remotename]/[branch]`。 这是一个十分常用的操作所以 Git 提供了 `--track` 快捷方式

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

如果想要将本地分支与远程分支设置为不同名字，可以轻松地增加一个不同名字的本地分支的上一个命令

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

现在，本地分支 `sf` 会自动从 `origin/serverfix` 拉取

想要让已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支，可以在任意时间使用 `-u` 或 `--set-upstream-to` 选项运行 `git branch` 来显式地设置

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

如果想要查看设置的所有跟踪分支，可以使用 `git branch` 的 `-vv` 选项。 这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do
it
  testing   5ea463a trying something new
```

这里可以看到 `iss53` 分支正在跟踪 `origin/iss53` 并且“ahead”是 2，意味着本地有两个提交还没有推送到服务器上。 也能看到 `master` 分支正在跟踪 `origin/master` 分支并且是最新的。 接下来可以看到 `serverfix` 分支正在跟踪 `teamone` 服务器上的 `server-fix-good` 分支并且领先 3 落后 1，意味着服务器上有一次提交还没有合并入同时本地有三次提交还没有推送。 最后看到 `testing` 分支并没有跟踪任何远程分支

需要重点注意的一点是这些数字的值来自于我们从每个服务器上最后一次抓取的数据。 *这个命令并没有连接服务器，* **它只会告诉我们关于本地缓存的服务器数据。** 如果想要统计最新的领先与落后数字，需要在运行此命令前抓取所有的远程仓库。 可以像这样做：`$ git fetch --all; git branch -vv`

拉取

当 `git fetch` 命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。 它只会获取数据然后让你自己合并。 **而有一个命令叫作 `git pull` 在大多数情况下它的含义是一个 `git fetch` 紧接着一个 `git merge` 命令。 由于 `git pull` 的魔法经常令人困惑所以通常单独显式地使用 `fetch` 与 `merge` 命令会更好一些。**就是说，`git pull`=`git fetch`+`git merge`，但是，通常情况下，我们还是最好使用`git fetch`

删除远程分支

假设我们已经通过远程分支做完所有的工作了，也就是说我们和协作者已经完成了一个特性并且将其合并到了远程仓库的 `master` 分支（或任何其他稳定代码分支）。 可以运行带有 `--delete` 选项的 `git push` 命令来删除一个远程分支。 如果想要从服务器上删除 `serverfix` 分支，运行下面的命令

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

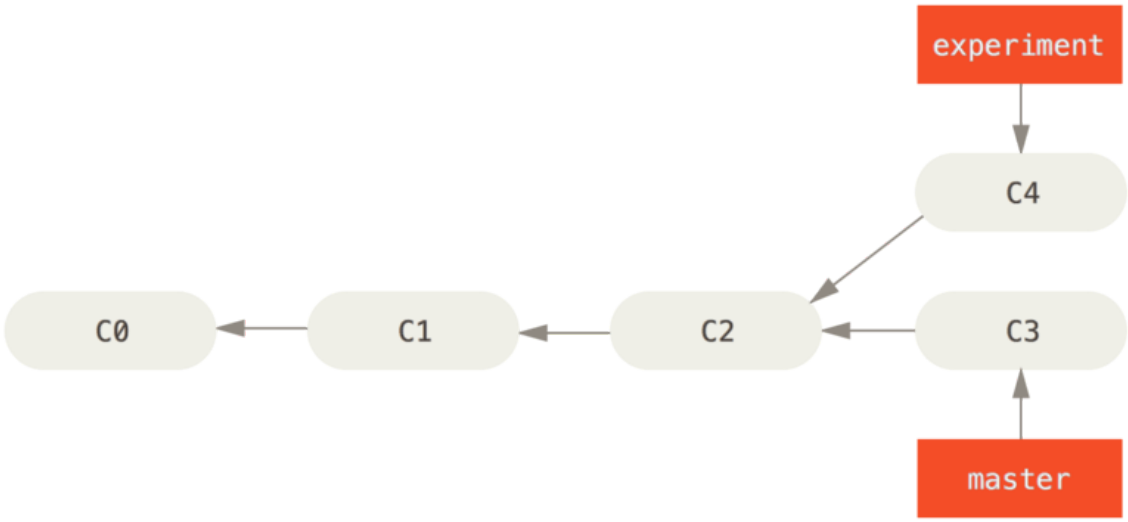
基本上这个命令做的只是从服务器上移除这个指针。 Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的

变基

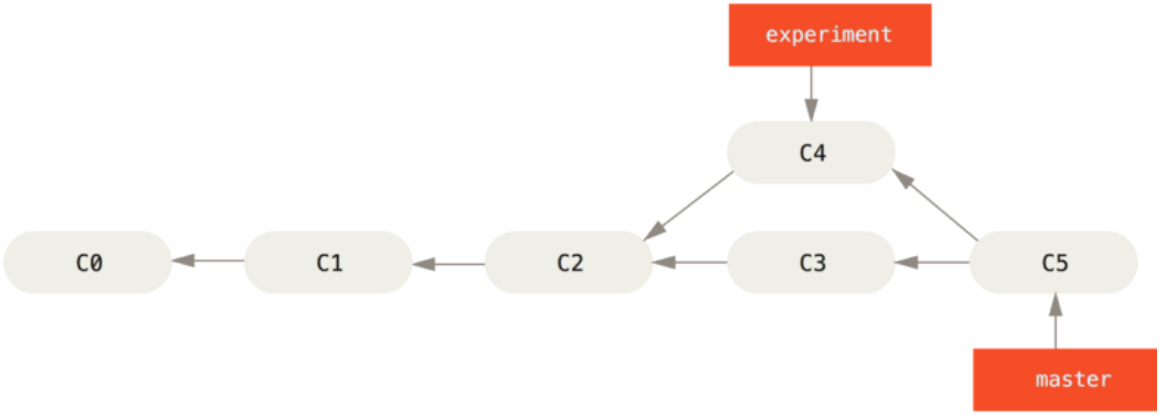
在 Git 中整合来自不同分支的修改主要有两种方法：`merge` 以及 `rebase`

变基的基本操作

在当前的git版本中，路径之中出现中文，变基会产生问题！可以考虑之前的例子，假若之前的开发任务分叉到两个不同分支，又各自提交了更新



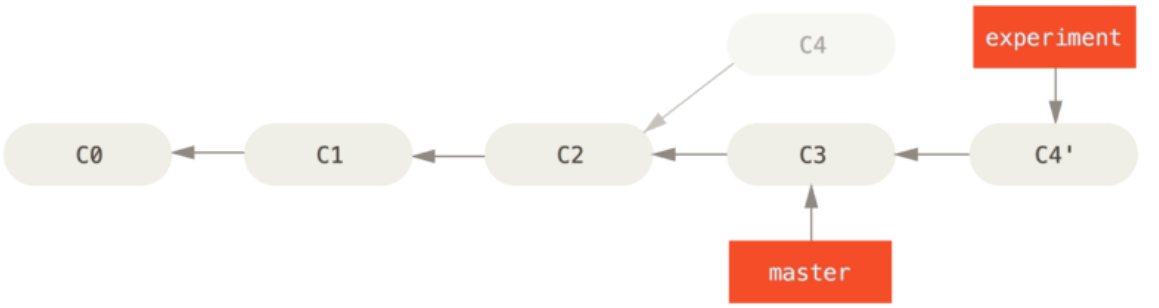
整合分支最容易的方法是 `merge` 命令。 它会把两个分支的最新快照（`C3` 和 `C4`）以及二者最近的共同祖先（`C2`）进行三方合并，合并的结果是生成一个新的快照（并提交）



其实，还有一种方法：**可以提取在 `C4` 中引入的补丁和修改，然后在 `C3` 的基础上应用一次。** 在 Git 中，这种操作就叫做***变基***。可以使用 `rebase` 命令将提交到某一分支上的所有修改都移至另一分支上，就好像"重新播放"一样。在上面这个例子中，运行

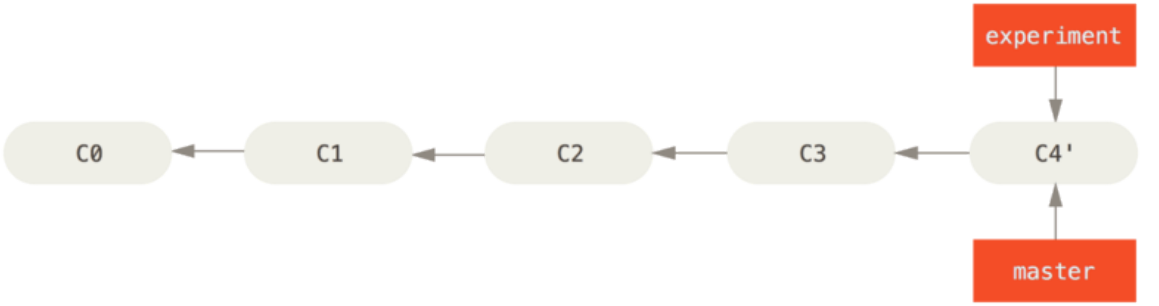
```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

它的原理是首先找到这两个分支（即当前分支 `experiment`、变基操作的目标基底分支 `master`）的最近共同祖先 `C2`，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件，然后将当前分支指向目标基底 `C3`，最后以此将之前另存为临时文件的修改依序应用。（写明了 commit id，以便理解，下同）



现在回到 `master` 分支，进行一次快进合并

```
$ git checkout master
$ git merge experiment
```



此时，`C4'` 指向的快照就和上面使用 `merge` 命令的例子中 `C5` 指向的快照一模一样了。 这两种整合方法的最终结果没有任何区别，但是**变基使得提交历史更加整洁。** 你在查看一个经过变基的分支的历史记录时会发现，*尽管实际的开发工作是并行的，但它们看上去就像是串行的一样，提交历史是一条直线没有分叉

一般我们这样做的目的是为了确保在向远程分支推送时能保持提交历史的整洁——例如向某个其他人维护的项目贡献代码时。 在这种情况下，我们首先在自己的分支里进行开发，当开发完成时需要先将你的代码变基到 `origin/master` 上，然后再向主项目提交修改。 这样的话，该项目的维护者就不再需要进行整合工作，只需要快进合并便可

需要注意的是，*无论是通过变基，还是通过三方合并，整合的最终结果所指向的快照始终是一样的，只不过提交历史不同罢了。* **变基是将一系列提交按照原有次序依次应用到另一分支上，而合并是把最终结果合在一起**

合并和变基的操作方向

合并需要现将某一个分支A检出（或者是当前分支A已经是检出状态），然后使用`git merge B`将B分支合并到A分支上

变基是使用某一分支A为基底，将当前的分支B（或者是当前分支B已经是检出状态），使用`git rebase A`，变基到A分支上。

需要注意，合并(merge)和变基(rebase)两个的方向正好相反，比如，当前分支为**A**，`git merge B`,是将B合并到A上，B--->A, 而`git rebase B`,是将A和B共同祖先之后的，A上面的变化，制作成补丁，打到B分支上，也就是说，该操作会舍弃 A分支上提取的 commit，同时**不会像 merge 一样生成一个合并修改内容的 commit，相当于把 A分支（当前所在分支）上的修改在 B 分支（目标分支）上原样复制了一遍，将A 分支指向 B 的最新提交（目标分支） 处**， B<---A

`rebase`**操作提取了当前分支的修改，将其复制在了目标分支的最新提交后面**,所以，*该操作会丢弃当前分支上的提交，将其作为补丁，在新的基分支上重新提交*

变基有冲突和无冲突的两种情况

第一种: 是`rebase`无冲突的情况

```
# 当前分支是v-rebase
hello@HELLO MINGW64 /f/gittest (v-rebase)
$ git rebase master  # 将v-rebase变基到master上
First, rewinding head to replay your work on top of it...
Applying: v-rebase operate
Using index info to reconstruct a base tree...
A       222.txt
Falling back to patching base and 3-way merge...
Applying: second time operate

# 查看当前分支状态
hello@HELLO MINGW64 /f/gittest (v-rebase)
$ git status
On branch v-rebase
nothing to commit, working tree clean

hello@HELLO MINGW64 /f/gittest (v-rebase)
$
```

第二种: 是`rebase`有冲突的情况。当`rebase`遇到了冲突时，需要**手动解决冲突。**手动解决了冲突之后，然后存储到stash缓存区域，再次继续变基`git rebase --continue`，就可以完成变基。当然此时也可以终止操作`git rebase --abort`，此操作会返回到变基前的状态

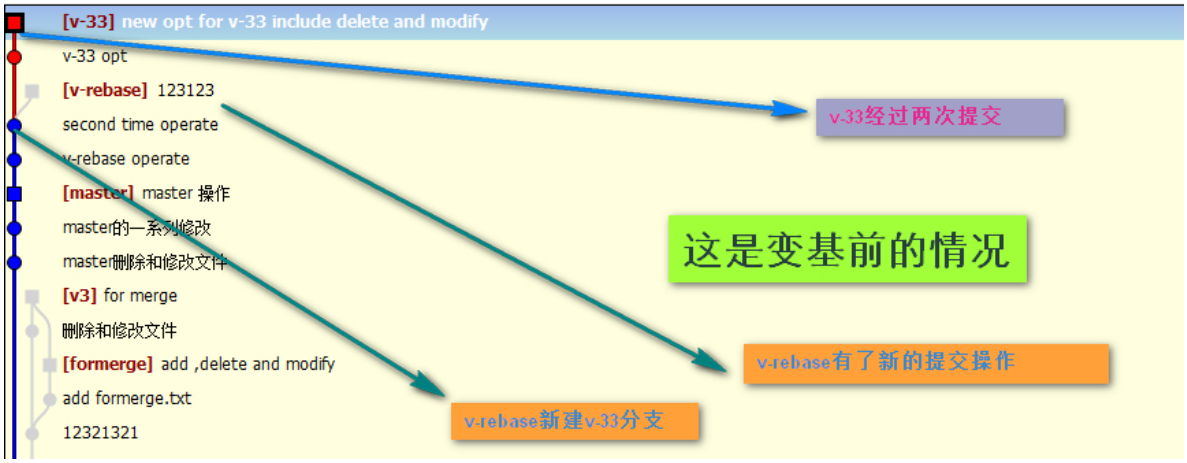
对于`rebase`产生冲突的情况和`merge`产生冲突的情况场景是一致的，都是如果对于同一个文件在两个分支上，产生了各自的操作，或者有了各自的增删情况，导致变基的时候，会产生冲突

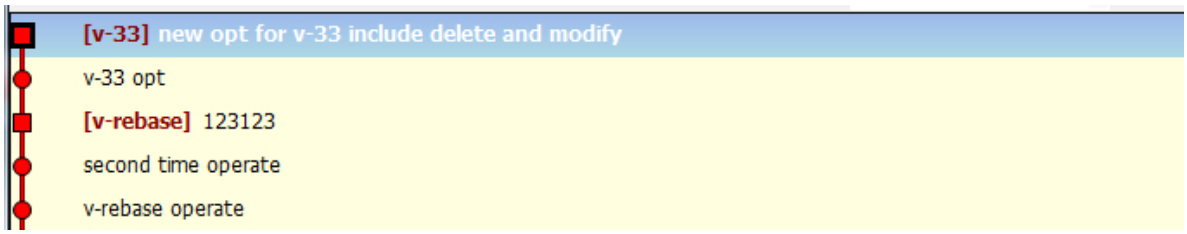
```
hello@HELLO MINGW64 /f/gittest (v-33)  # v-33是当前分支
$ git status
On branch v-33
nothing to commit, working tree clean

hello@HELLO MINGW64 /f/gittest (v-33)  # 将v-33分支变基到v-rebase分支上
$ git rebase v-rebase
First, rewinding head to replay your work on top of it...
Applying: v-33 opt
error: Failed to merge in the changes.   #变基遇到了冲突
Using index info to reconstruct a base tree...
A       v1.txt
M       v1v1.txt
Falling back to patching base and 3-way merge...
CONFLICT (modify/delete): v1v1.txt deleted in v-33 opt and modified in HEAD.
Version HEAD of v1v1.txt left in tree.
CONFLICT (modify/delete): v1.txt deleted in HEAD and modified in v-33 opt.
Version v-33 opt of v1.txt left in tree.
Removing m1.txt
Patch failed at 0001 v-33 opt
The copy of the patch that failed is found in: .git/rebase-apply/patch

Resolve all conflicts manually, mark them as resolved with  # 需要手动解决冲突
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --
abort".


hello@HELLO MINGW64 /f/gittest (v-33|REBASE 1/2)  # 存储到stash区域
$ git add rh.txt

hello@HELLO MINGW64 /f/gittest (v-33|REBASE 1/2)  # 删除冲突的文件
$ git rm v1v1.txt
v1v1.txt: needs merge           # 提示需要合并，也就意味着，首先得存到缓存区域
rm 'v1v1.txt'

hello@HELLO MINGW64 /f/gittest (v-33|REBASE 1/2)  # 查看状态
$ git status
rebase in progress; onto 9a45c03
You are currently rebasing branch 'v-33' on '9a45c03'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    m1.txt
        new file:   rh.txt
        deleted:    v1v1.txt

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
```

```
  (use "git add/rm <file>..." as appropriate to mark resolution)

        deleted by us:   v1.txt


hello@HELLO MINGW64 /f/gittest (v-33|REBASE 1/2)  # 添加到缓存区域
$ git add .

hello@HELLO MINGW64 /f/gittest (v-33|REBASE 1/2)
$ git status
rebase in progress; onto 9a45c03
You are currently rebasing branch 'v-33' on '9a45c03'.
  (all conflicts fixed: run "git rebase --continue")

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    m1.txt
        new file:   rh.txt
        new file:   v1.txt
        deleted:    v1v1.txt

hello@HELLO MINGW64 /f/gittest (v-33|REBASE 1/2)  #  解决了冲突，继续变基
$ git rebase --continue
Applying: v-33 opt
Applying: new opt for v-33 include delete and modify  #  此时，变基已经完成

hello@HELLO MINGW64 /f/gittest (v-33)  #  查看最近两次的操作日志
$ git log -p -2
commit 504de3cb8c27d08b27de6fc1398728d45b67d6bf (HEAD -> v-33)  # last 1
Author: zhello <224hello@qq.com>
Date:   Thu Jan 4 13:00:40 2018 +0800

    new opt for v-33 include delete and modify

diff --git a/ganggang.txt b/ganggang.txt
deleted file mode 100644
index 81581d3..0000000
--- a/ganggang.txt
+++ /dev/null
@@ -1 +0,0 @@
-ganggangjiade
\ No newline at end of file
diff --git a/rh.txt b/rh.txt
index ae81893..f6de5fd 100644
--- a/rh.txt
+++ b/rh.txt
@@ -1 +1,7 @@
-sfdafafafd
\ No newline at end of file
+sfdafafafd
+
+131231231
+
+
+
+3123123
\ No newline at end of file
```

```
diff --git a/tt1.txt b/tt1.txt
index b322b68..7259b92 100644
--- a/tt1.txt
+++ b/tt1.txt
@@ -1 +1,4 @@
-321312
\ No newline at end of file
+321312
+
+fdsgdsgfsggs
+gdfsg
\ No newline at end of file

commit 4a8afa139dd8291f8d5eef987dff08920c82d85a  # last 2
Author: zhello <224hello@qq.com>
Date:   Thu Jan 4 12:59:05 2018 +0800

    v-33 opt

diff --git a/m1.txt b/m1.txt
deleted file mode 100644
index 03079c2..0000000
--- a/m1.txt
+++ /dev/null
@@ -1,6 +0,0 @@
-fsadfasdfda
-
-
-f
-sa
-11
diff --git a/rh.txt b/rh.txt
new file mode 100644
index 0000000..ae81893
--- /dev/null
+++ b/rh.txt
@@ -0,0 +1 @@
+sfdafafafd
\ No newline at end of file
diff --git a/v1.txt b/v1.txt
new file mode 100644
index 0000000..48b2e3e
--- /dev/null
+++ b/v1.txt
@@ -0,0 +1,9 @@
+<D5><E2><CA><C7>v1
+
+V1
+V1  V2
+sdfafs
+fsadsaf
+fsdasfda
+fsdaaffaf
+fdsaf
\ No newline at end of file
diff --git a/v1v1.txt b/v1v1.txt
deleted file mode 100644
index bb4855f..0000000
```

```
--- a/v1v1.txt
+++ /dev/null
@@ -1,2 +0,0 @@
-v1v1123
-v1v1123123213123
\ No newline at end of file

hello@HELLO MINGW64 /f/gittest (v-33)
$
```

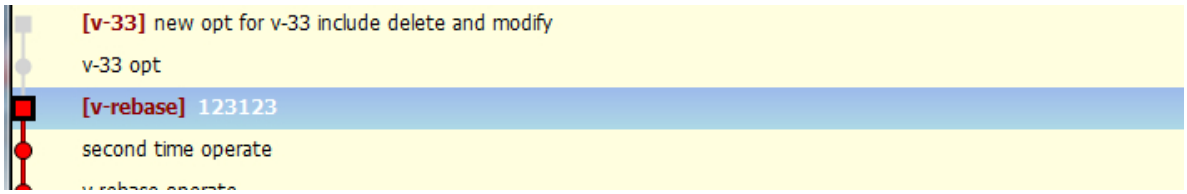




变基完成后

当完成了变基之后，虽然看上面的第3张分支图，好像是两个分支合二为一了，但是其实还是可以切换的，如果有新的添加或者操作，则两个分叉又会增长，这其实和 `rebase` 的含义是一致的，也就是，**首先找到这两个分支（即当前分支 `v-33`、变基操作的目标基底分支 `v-rebase`）的最近共同祖先，然后对比当前分支相对于该祖先的历次提交，提取 `v-33` 和 `v-rebase` 中从共同祖先之后的修改，并存为临时文件，然后将当前分支 `v-33` 指向一个新的目标基地，最后以此将之前另存为临时文件的修改依序应用到 `v-33`**，~~而我们的 `v-rebase` 分支还是没有在此过程之中受影响~~

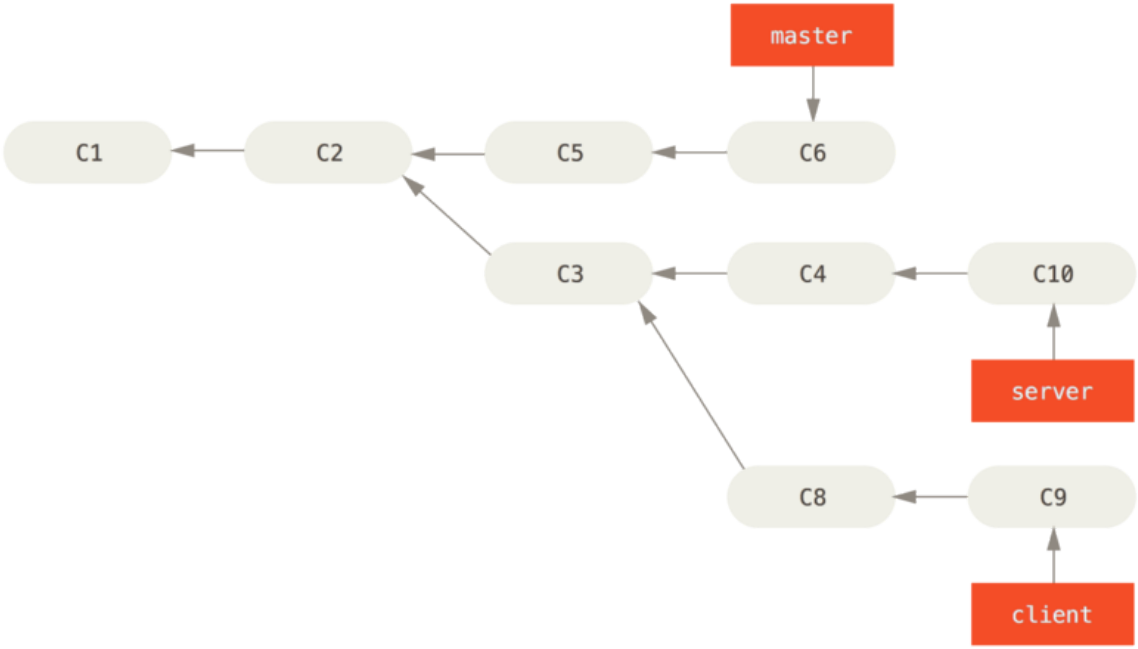
```
hello@HELLO MINGW64 /f/gittest (v-33)
$ git checkout v-rebase
Switched to branch 'v-rebase'

hello@HELLO MINGW64 /f/gittest (v-rebase)
$
```




```
hello@HELLO MINGW64 /f/gittest (v-rebase)
$ git status
On branch v-rebase
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   tt1.txt

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

hello@HELLO MINGW64 /f/gittest (v-rebase)
$ git add .
g
hello@HELLO MINGW64 /f/gittest (v-rebase)
$ git commit -m 'modify tt1.txt'
[v-rebase 89d71c0] modify tt1.txt
 1 file changed, 2 insertions(+), 1 deletion(-)

hello@HELLO MINGW64 /f/gittest (v-rebase)
$
```

更有趣的变基例子

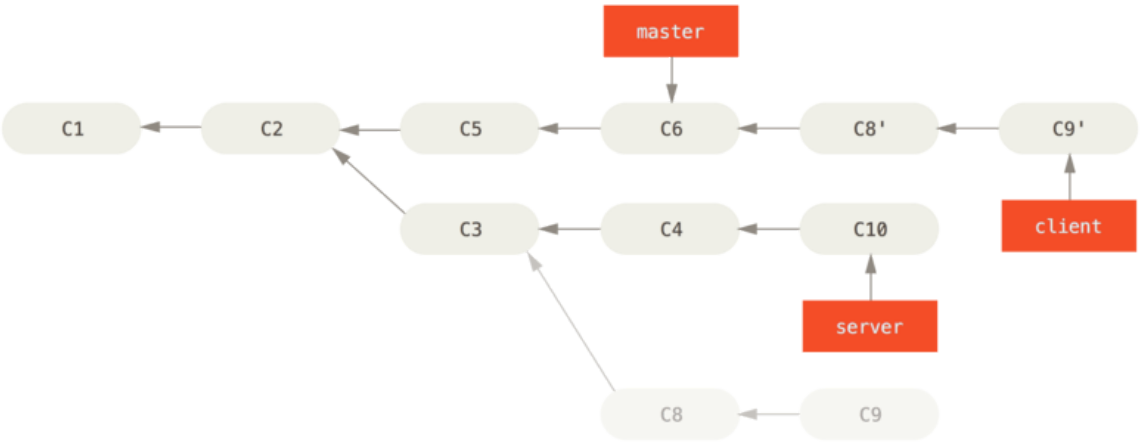
在对两个分支进行变基时，所生成的“重放”并不一定要在目标分支上应用，**也可以指定另外的一个分支进行应用。** 就像**从一个特性分支里再分出一个特性分支的提交历史中**的例子那样。 我们创建了一个特性分支 `server`，为服务端添加了一些功能，提交了 `C3` 和 `C4`。 然后从 `C3` 上创建了特性分支 `client`，为客户端添加了一些功能，提交了 `C8` 和 `C9`。 最后，回到 `server` 分支，又提交了 `C10`。



假设我们希望将 `client` 中的修改合并到主分支并发布，但暂时并不想合并 `server` 中的修改，因为它们还需要经过更全面的测试。 这时，就可以使用 `git rebase` 命令的 `--onto` 选项，选中在 `client` 分支里但不在 `server` 分支里的修改（即 `C8` 和 `C9`），将它们在 `master` 分支上重放

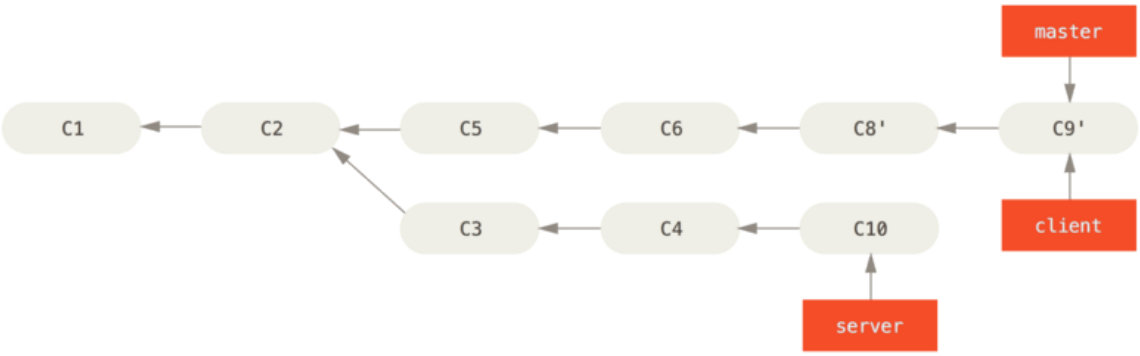
```
$ git rebase --onto master server client
```

以上命令的意思是："取出 `client` 分支，找出处于 `client` 分支和 `server` 分支的共同祖先之后的修改，然后把它们在 `master` 分支上重放一遍"。 这理解起来有一点复杂，不过效果非常酷



现在可以快进合并 `master` 分支了。（如下图：快进合并 master 分支，使之包含来自 client 分支的修改）

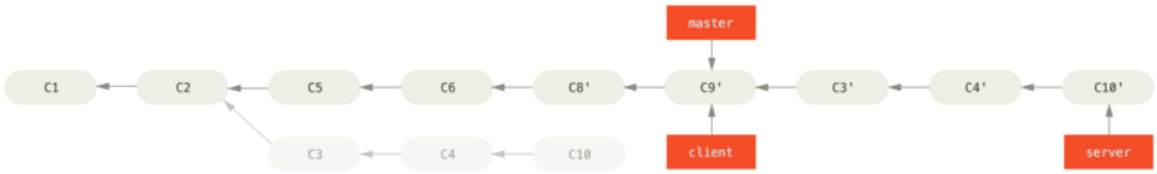
```
$ git checkout master
$ git merge client
```



接下来我们决定将 `server` 分支中的修改也整合进来。 使用 `git rebase [basebranch] [topicbranch]` 命令可以直接将特性分支（即本例中的 `server`）变基到目标分支（即 `master`）上。这样做能省去你先切换到 `server` 分支，再对其执行变基命令的多个步骤。

```
$ git rebase master server
```

如下图(将 server 中的修改变基到 master)上所示， `server` 中的代码被“续”到了 `master` 后面。



然后就可以快进合并主分支 master 了

```
$ git checkout master
$ git merge server
```

至此， `client` 和 `server` 分支中的修改都已经整合到主分支里了，我们就可以删除这两个分支，最终提交历史会变成下图中的样子

```
$ git branch -d client
$ git branch -d server
```

变基的风险

变基也并非完美无缺，要用它得遵守一条准则：**不要对在仓库外有副本的分支执行变基**!这一点很重要

变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。 如果已经将提交推送至某个仓库，而其他人也已经从该仓库拉取提交并进行了后续工作，此时，如果用 `git rebase` 命令重新整理了提交并再次推送，我们的同伴因此将不得不再次将他们手头的工作与我们刚才的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟，慎用变基

变基 vs. 合并

至此，我们已在实战中学习了变基和合并的用法，一定会想问，到底哪种方式更好。 在回答这个问题之前，让我们退后一步，讨论一下提交历史到底意味着什么

有一种观点认为，仓库的提交历史即是 **记录实际发生过什么。** 它是针对历史的文档，本身就有价值，不能乱改。 从这个角度看来，改变提交历史是一种亵渎，你使用*谎言*掩盖了实际发生过的事情。 如果由合并产生的提交历史是一团糟怎么办？ 既然事实就是如此，那么这些痕迹就应该被保留下来，让后人能够查阅

另一种观点则正好相反，他们认为提交历史是 **项目过程中发生的事。** 没人会出版一本书的第一版草稿，软件维护手册也是需要反复修订才能方便使用。 持这一观点的人会使用 `rebase` 及 `filter-branch` 等工具来编写故事，怎么方便后来的读者就怎么写

现在，让我们回到之前的问题上来，到底合并还是变基好？希望我们可以明白，这并没有一个简单的答案。 Git 是一个非常强大的工具，它允许我们对提交历史做许多事情，但每个团队、每个项目对此的需求并不相同。 既然我们已经分别学习了两者的用法，相信你能够根据实际情况作出明智的选择

总的原则是，**只对尚未推送或分享给别人的本地修改执行变基操作清理历史，从不对已推送至别处的提交执行变基操作，这样，我们才能享受到两种方式带来的便利**，也就是说,**本地变基，远程合并。**