

版本控制入门--搬进GitHub（慕课网）视频教程记录

文章来自 yafey's Blog // Love Leanote!

主页 | 善用佳软 | About Me | 归档 | 标签

来源： 版本控制入门 - 搬进 Github （Github 客户端操作 - 视频教程）

⌚ 2016-05-02 00:10:05

👁 153 ⏺ 0 💬 1

第1章 课程介绍
第2章 浏览器中使用 Github
2-1 浏览器中使用Github (10:13)
第3章 Github 客户端的使用
3-1 Github客户端的使用 (11:11)
第4章 简单分支操作
4-1 简单分支操作 (09:41)
第5章 分支合并
5-1 合并分支 (上) (06:53)
5-2 合并分支 (下) (07:15)
第6章 团队协作流程
6-1 团队协作流程 (上) (06:23)
6-2 团队协作流程 (下) (07:12)
第7章 开源项目贡献流程
7-1 开源项目贡献流程 (12:26)
第8章 Github Issues
8-1 Github Issues (11:58)
第9章 Github Pages 搭建网站
9-1 Github Pages 搭建网站 (05:06)
第10章 Github 的秘密机关
10-1 Github的秘密机关 (05:37)
第11章 Until Next Time, Goodbye!

Content

Section 2 浏览器中使用Github
new Repositories
commit
版本树示意图
Section 3 Github客户端 (GitHub Desktop) 的使用
客户端创建仓库
添加文件到仓库中 (形成版本树)
回滚 (Undo 和 Revert ,
以及 roll back to this commit) 和发布 (publish)
Undo 操作： 只适合没有同步到 GitHub 的版本
Revert (抵消) this commit: 撤销此次的同步的 commit
roll back to this commit :
回滚到指定版本，废弃中间版本 (会新增一个版本)
publish:发布到 GitHub 网站上

Section 4 简单分支操作
什么是分支?
创建分支 和 切换当前分支
新建分支 和 master 分支
切换分支
同步分支到 远端仓库 (如 : GitHub)
删除分支
删除已发布的远端分支 (该功能在最新的 Desktop 中已消失)
切换 GitHub 的默认分支

Section 2 浏览器中使用Github

new Repositories

github 中是以 仓库 的概念存储代码。

- new repository 不是新项目，而是 新仓库。用于存储过往版本的仓库。
- 新建一个仓库
 1. 项目名称
 2. 描述
 3. 自动新增一个 readme 文件
 4. commit

commit

commit 是出现频率最高的一个单词。

- 点击 2 commits ， 可以查看 该仓库 所有 commit 的版本。

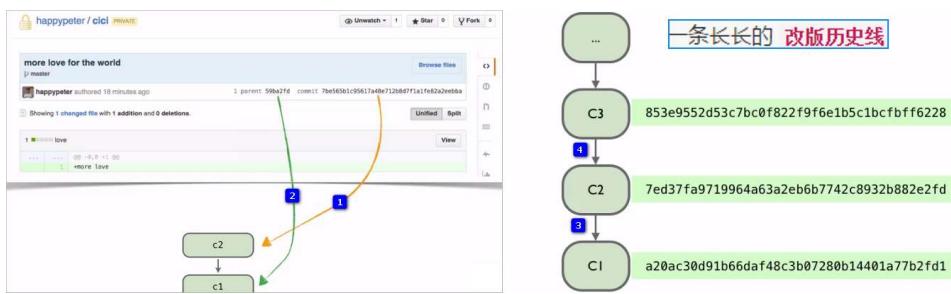
- 一个 commit 就是一个版本。
 - 一个版本中的信息：
 - Who** 在 **What time** 做了哪些修改, 为什么 (**Why**) 要修改 (**Commit 描述**)。
- 版本号使用一个 40 位的 (**SHA**) 字符串来标记。
 - 可以使用版本号的前几位 (貌似最少 4 位) 作为简写形式 (只要保证不会重复就可以)。
 - 父版本号使用前 7 位 版本号。
- 用 `命名/仓库名/commit/版本号`, 即可查到具体的过往版本
 - 比如: <https://github.com/yafey/cici/commit/91c7f80>
- 一个版本中, 会包含该版本的 (7位) 父版本的版本号。



版本树示意图

在 GitHub 网站上, 一个仓库的版本树的地址: <https://github.com/yafey/cici/commits/master> (即点击 **2 commits** 后的页面)

1. 假设知道了当前版本号, 用 **C2** 来表示。
2. 同时, 当前版本保存了父版本号 **C1** (上一个版本号), 就可以由 **C2** 指向 **C1**。
3. 这样 **C2** 和 **C1** 就连成了一条线。
4. 再做新版本, 又会和 **C2** 连成线, 一直做下去, 就会形成一条长长的 **改版历史线**。



一条长长的 改版历史线

Section 3 GitHub客户端 (GitHub Desktop) 的使用

见另一个笔记

客户端创建仓库

- `add`: 添加本地已有的仓库
- `creat`: 创建一个仓库, 原本没有
 - 在 GitHub Desktop 中创建的项目仓库, 是先创建在本地的, 这样我们就可以使用我们熟悉的 IDE 开发项目了, 开发完之后再提交到 GitHub 上去。
- `Clone`: 下载 GitHub 中的仓库到本地。

添加文件到仓库中 (形成版本树)

在新建的仓库中添加文件后, 在 GitHub Desktop 页面会在 `Changes` 有小蓝点提示有改动。

1. 点击 `Changes`。
2. 选择要 `commit` 的文件。
3. 可以点击文件中不想 `commit` 的行, 如图所示。
 - 默认是 `commit` 文件中的所有行。
 - `commit` 的行号显示的颜色是 **蓝色**, 非 `commit` 的行号显示的颜色是 **绿色**。
4. 编写 `commit comment`, 为什么修改。
5. `commit`。

Section 5 合并分支

本地两个分支合并

代码冲突 conflicts

在不同分支上修改不同的文件, 不会出现冲突
在不同分支上修改相同文件, 出现冲突需要手动 merge

合并远端分支

无冲突

冲突文件

Section 6 团队协作流程

什么是 GitHub Flow ?

创建一个分支

深度技巧

添加新版本

深度技巧

开启一个 Pull Request

深度技巧

讨论和代码审核

深度技巧

合并分支, 然后部署

深度技巧

示例: 下面是一个实际例子。

给队友添加写权限

开话题分支并在上面开发

Pull Request

讨论审核代码

快速 PR

总结

Section 7 开源项目贡献流程

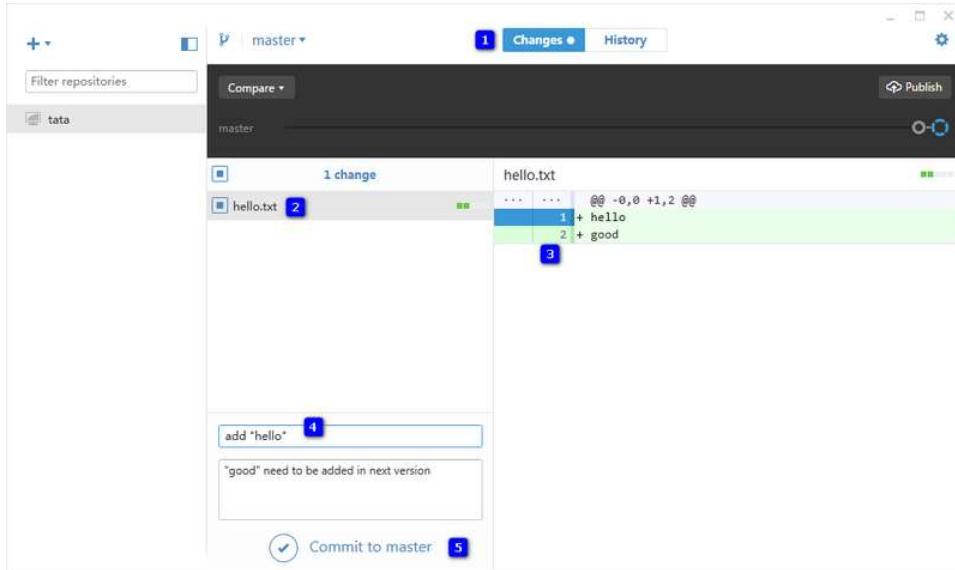
Section 8 Github Issues

Section 9 Github Pages 搭建网站

Section 10 Github的秘密机关



6. 多次 commit 形成版本树。



回滚 (Undo 和 Revert , 以及 roll back to this commit) 和发布 (publish)

Git 各种操作都是来回折腾各个版本 —— 应证了 —> Git 是「一个很强大的」版本控制工具。

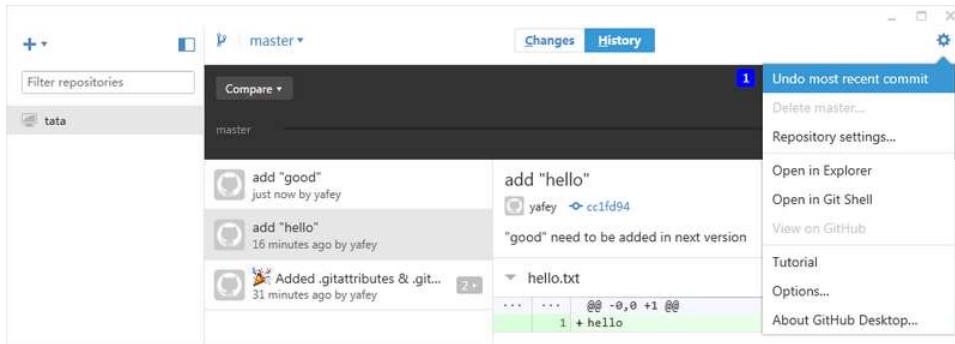
Undo 操作 : 只适合没有同步到 GitHub 的版本

Undo 操作 只适用于 没有同步的版本 (没有上传到 GitHub 网站上)。

- 什么是版本的同步呢?
 - 意思就是我们通过 GitHub Desktop 做的版本，只是在本地上，没有上传到 GitHub 的网站上去。
 - 那么对于上传到 GitHub 上的版本，我们可以通过 revert this commit 选项来撤销。
 - 我们还可以使用 Roll Back to this Commit 来回滚到当前版本，这样就撤销了之后的所有版本了。

说明:

- Undo 操作 (Undo most recent commit) 只能撤销最近一次的 commit .

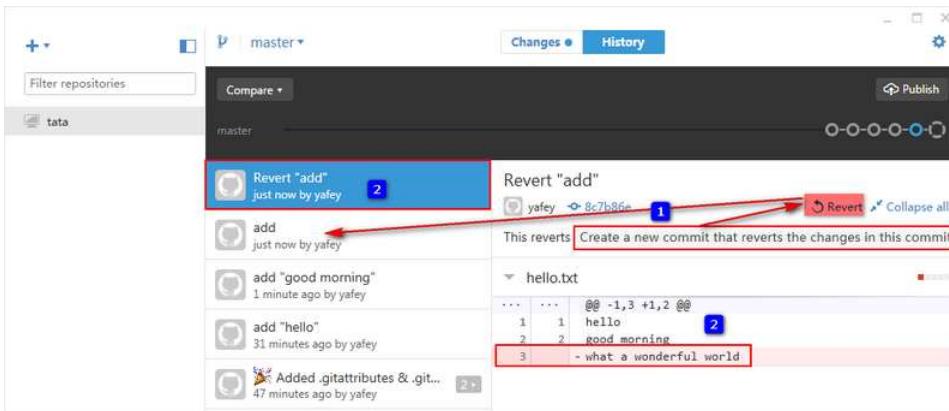


Revert (抵消) this commit : 撤销此次的 同步的 commit

已经发布的 commit 不能 Undo ， 是因为，队友已经看到了这次 commit ， 如果直接删除，会引起版本的 异常/混乱，所以，对于已经同步的 commit ， 需要使用 Revert this commit ， 会新增一个版本来标识将当前版本的代码回滚到了上一个版本。

说明:

1. 在需要 revert 的版本中 点击 Revert 即可。
2. 会新增一个版本号，显示此次的改动。

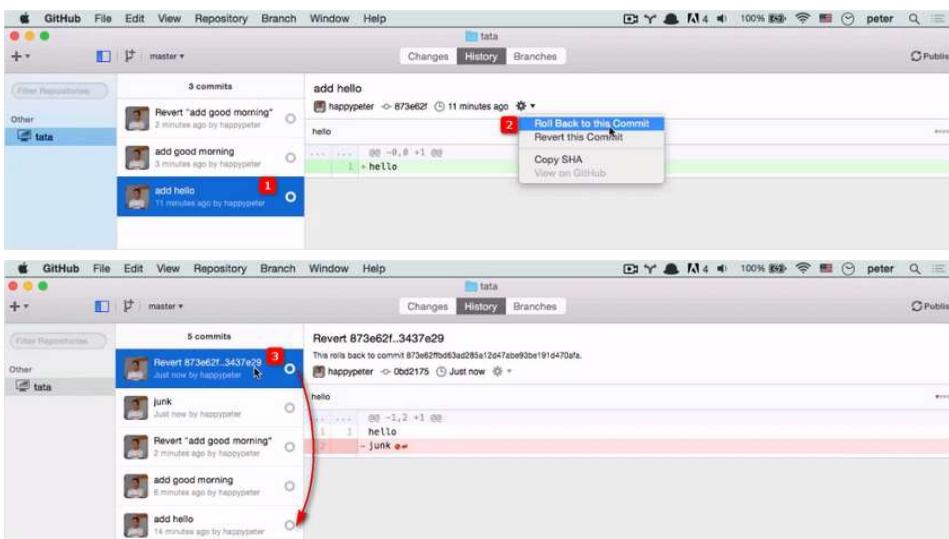


roll back to this commit : 回滚到指定版本，废弃中间版本（会新增一个版本）

之前某个版本的代码是正确的，在这之后进行了一些改动，现在想要还原到之前正确的版本。

说明：

1. 选中想要回滚的版本（之前正确的版本）。
2. 点击 `Roll back to this commit` 还原到正确的版本。
3. 将会新增一个版本来保存与之前正确的版本一致的内容（也适用于已经发布的版本）。



publish:发布到 GitHub 网站上

发布到 GitHub 网站上的好处：

1. 发布到 GitHub 上，代码有备份了。
2. 发布了才能和别人分享，进行协作。

说明：使用 GitHub 官方工具操作很简单。

1. 选择一个仓库，点击 `Publish`。
2. 可以修改在 GitHub 上的仓库名。
3. 在 GitHub 上对该仓库的描述。
4. 付费用户可以将仓库设置为 `private`，免费帐号不行。
5. 点击 `Publish Repository`。



Section 4 简单分支操作

- Git 最核心的 操作对象 是 版本 (commit)，最核心的操作技巧 就是 分支 (branch)。
- master (主分支，各个版本组成的历史线) 相当于指向最新版本的指针。
 - master 在 GitHub 上是 默认分支，是不能被删除的。

什么是分支？

仓库创建后，一旦有了新 commit，默认就会放到一个分支上，名字叫 master。

前面咱们一直看到的多个版本组成的一条历史线，就是 master 分支。

但是一个仓库内，用户可以自己创建其他的分支，可以有多条历史线。

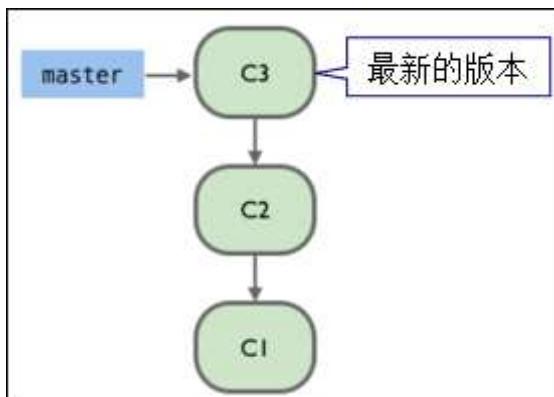
说说 master 这个名字，一般中文叫「主分支」，其实从技术底层来讲它跟其他我们自己要创建的分支没有区别，只不过它是天生的默认分支。实际工程项目中会人为的给它一个重要的使命，存放稳定代码。就像 github 公司倡导的。

master 分支上的所有代码都应该是可以部署的。

意思就是 master 分支上的代码是随时可以放到产品服务器上跑的代码。这样，如果想开发一个新功能，可以新开分支。想象一下历史线上有很多节，每个版本就是一节。一个分支相当于一根竹子，一节节的往上长。



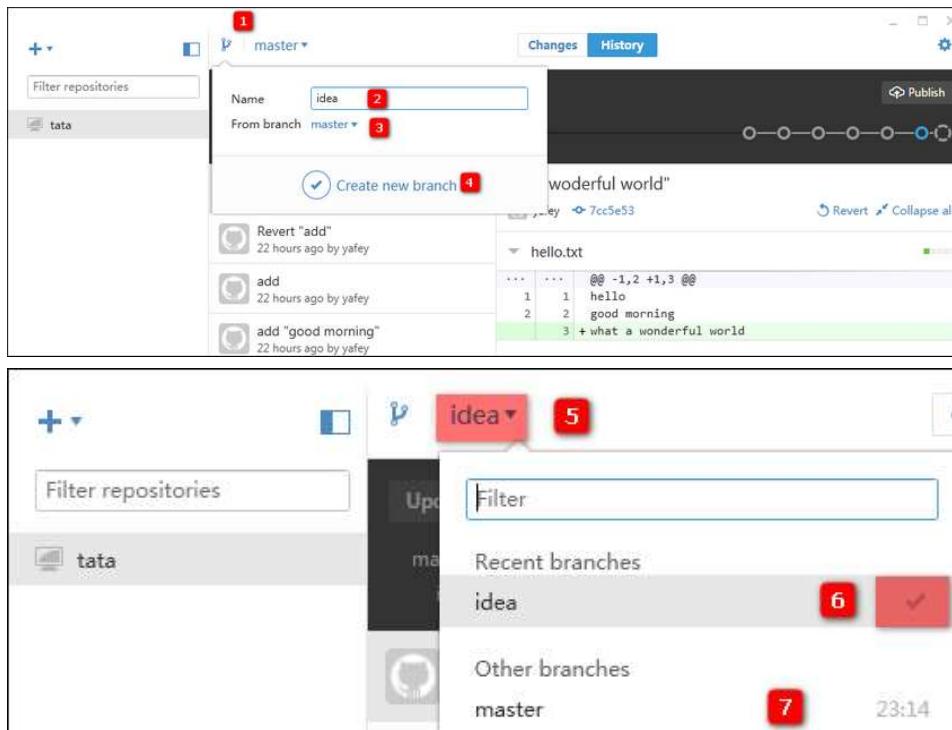
但是实际上，在底层并不是每个分支都拷贝出自己独立的一条历史线。其实 master 本身只是一个指针，指向 master 分支上最新的一个版本。这样由于每个 commit 都可以顺藤摸瓜找到自己的前一个 commit，那么这条历史线就可以确定了。



创建分支 和 切换当前分支

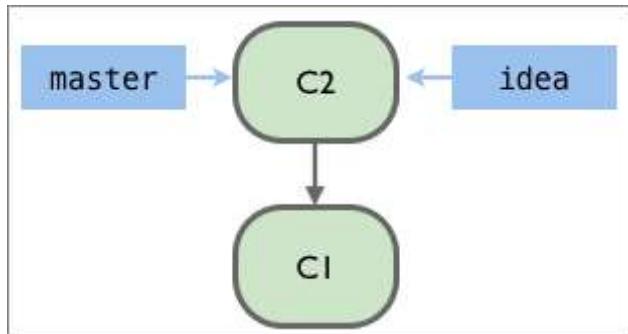
开分支的目的是，我们有好的想法想要实现，但是我们又不想污染原来的项目（master这个分支），那么我们就可以自己开一个新的分支。

1. 点击「创建分支图标」。
2. 创建 `idea` 分支（Create a new branch off master），
3. 表示新创建的 `idea` 分支是基于 `master` 分支的。
4. 创建后的 `idea` 分支并不为空，而是拥有和 `master` 一样的历史线。
5. 创建分支后，「当前分支」（current branch）自动切换到了 `idea` 分支。
6. 小对勾表示「当前分支」（current branch）。
7. 点击「其他分支」中的分支（如：`master` 分支）可切换到其他分支。



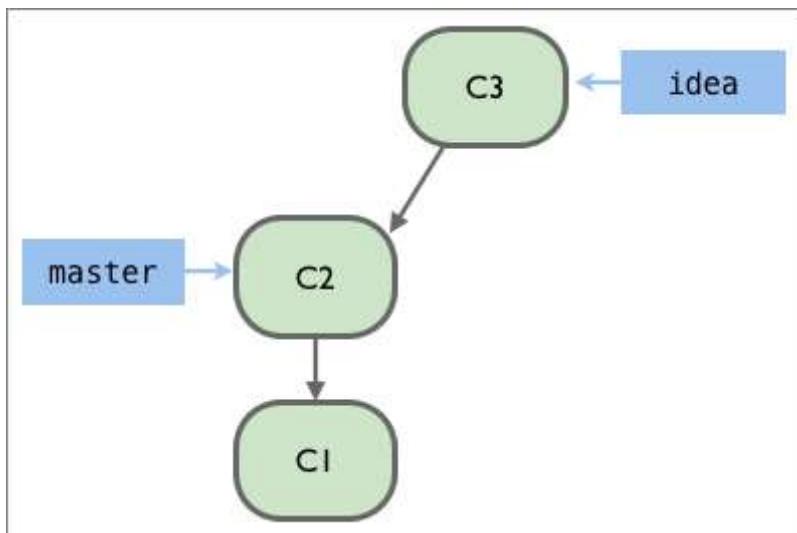
新建分支 和 master 分支

在底层，新建分支（此处为：`idea` 分支）的实现是非常巧妙的，就是又创建一个新的 `idea` 指针，跟 `master` 指针指向同一个版本，根本没有拷贝历史线。



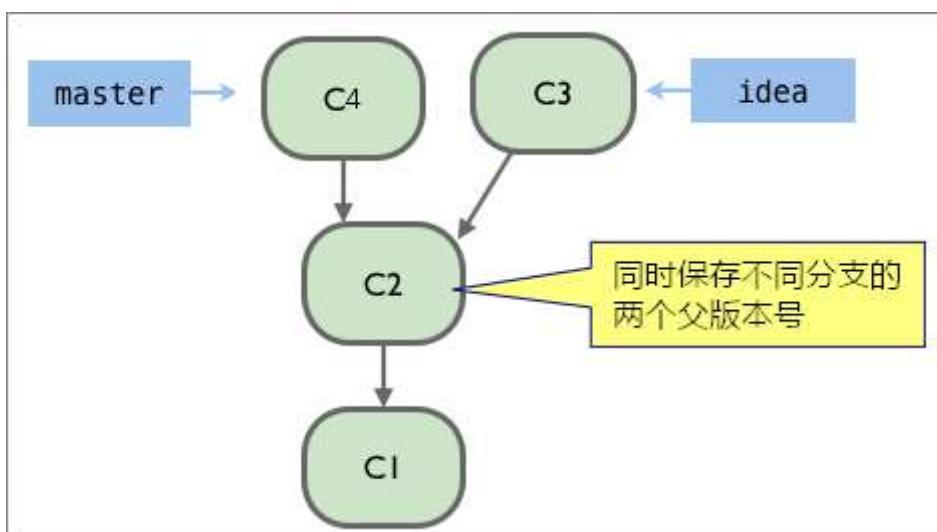
如果现在我对项目做一下修改，然后 commit 了。那么移动的只是 `idea` 指针，`master` 不变。就成了这样：（现在 `master` 分支包含两个版本 C1 和 C2，`idea` 分支包含三个版本 C

1, C2, C3。)



如果此时 master 分支上同时也进行了修改，然后 commit 了。那么移动的也只是 master 的指针，idea 不变，就成了这样：

- 这里需要注意的是：C2 版本将会保存 2 个父版本的（一个是 master 分支的 C4 版本，另一个是 idea 分支的 C3 版本）。



切换分支

在上面图中的 步骤5，相中哪个分支了，双击一下就切换过去了。

时间长了你会觉得这个也不够快，还是纯键盘操作快。敲 Cmd-B 可以打开分支切换框，输入名字回车，就切换成功了。

如果你在 idea 分支上有了修改但是还没有来得及 commit，这时候如果切换分支，那么 git 会帮你保存这部分修改，也就是在 切换到的分支 上是看不到这部分修改的。但是不要担心，只要你切换回老分支，修改内容又回来了。

注意，每次切换分支，项目代码，术语叫 工作树（Working Tree）是会随着变化的，在编辑器中看看就知道了。

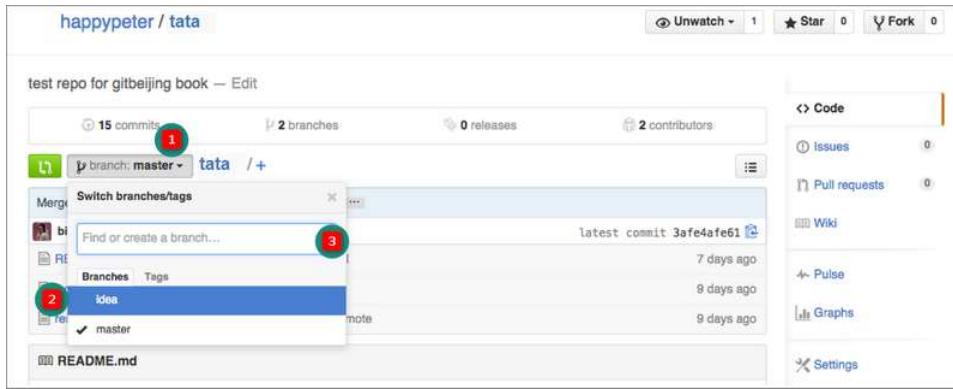
同步分支到远端仓库（如：GitHub）

默认情况下这个 idea 分支只是存在于本地，如果想在远端仓库上发布这个分支，就点一下 idea 分支右侧的「Publish」按钮。

这样，到远端仓库看一下，如下图：

1. 点击此处 查看分支。

2. 发现果然多了一个 idea 分支。
3. 此处的输入框中，不但能搜索已有分支，还能创建新分支。（很多操作在本地客户端（GitHub Desktop）和 github.com 上都能进行。）



删除分支

- 首先当前分支是不能删除的。
什么意思？
到客户端的 `Branches` 标签下，左侧有对勾的就是当前分支，打开右侧小箭头的下拉菜单，可以看到 `delete` 这一项是禁用的。想删除它，就先要切换到其他分支，例如 `master`。这样就可以删除 `idea` 分支了，如果执行本地删除 `github.com` 上对应分支也会同时被删除。（在新版本中「(3.0.11.0) 3b1518a」，暂时没有找到怎么删除分支。）
- 默认分支是不能被删除的。（默认为 `master` 分支，可在 GitHub 上修改）
在客户端把分支切换到 `idea` 分支，现在试图去删除 `master`。点开 `master` 分支的小箭头，发现 `delete` 一项可以点，所以点一下，但是报错了：“`“master” is the repository’s default branch and cannot be deleted.` 要到 `github.com` 上修改默认分支（`default branch` 注意跟当前分支是不同的），就像这样。（新版客户端中没找到删除分支的地方。）

删除已发布的远端分支（该功能在最新的 Desktop 中已消失）

原先，在分支上可以点击 `Unpublish`，就可以只删除远端的分支不删除本地分支。

（该功能在最新的 Desktop 中已消失。）

切换 GitHub 的默认分支

在远端仓库，也就是 `github.com` 上如何切换默认分支呢？到 `Settings` 下面就更改 `Default branch` 就可以了。

1. 默认为 `master` 分支。
2. 可以选择其他分支。（可选的分支已经在这里列出。）
3. （新增）现在在 GitHub 上还可以设置将某些分支保护起来（删除等操作将受到权限控制。）

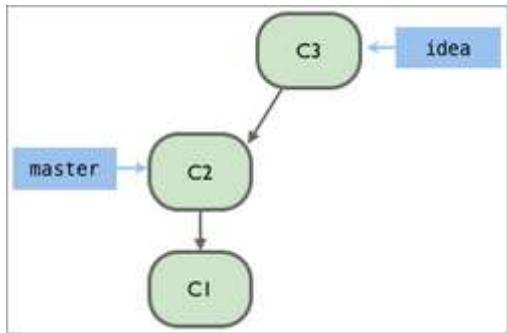
Section 5 合并分支

在一个 idea 分支上实现了一个想法，今天我想把这些代码放回 master 分支，这就涉及到了两个分支合并的技巧。

今天找几种实际情形，看看合并分支都有那些应用。

本地两个分支合并

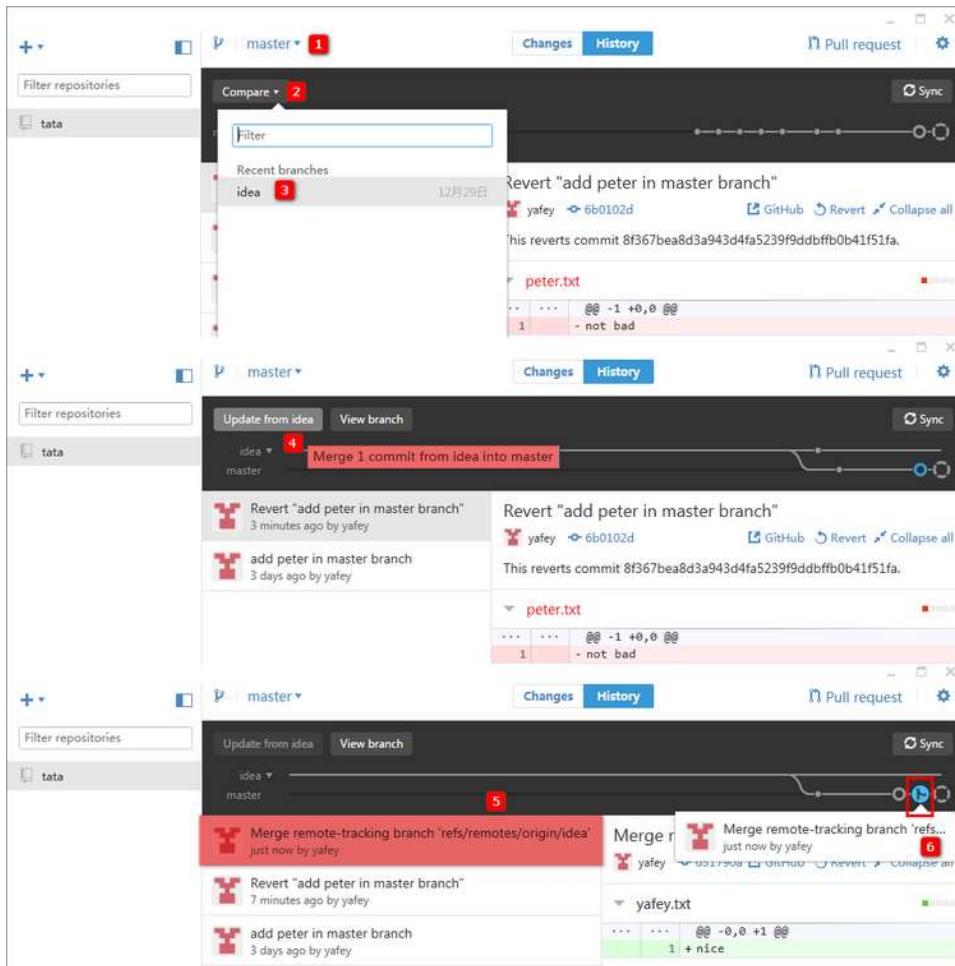
先从最简单的一种情况着手。现在项目只有一个 master 分支，我来新建一个 idea 分支，实现自己的想法，commit 一下。那现在仓库内的情况就是这样的：



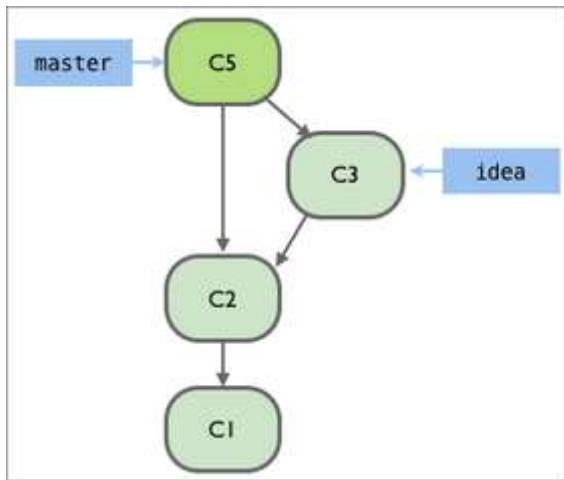
这个是前面已经见过的情形了。那如何让 idea 的代码并入 master 呢？

需要「融合」（**merge**）一下。

1. 点击到「master」分支。（如果是 idea 分支，动作将是 idea 分支 rebase { 将 master 分支上的改动更新到 idea 分支上 }）
2. 点击「Compare」，会列出其他分支。
3. 点击选择一条「其他分支」（此处的其他分支为 idea 分支）。
4. 将会列出下图的界面，鼠标放在「Update from idea」，会提示「Merge 1 commit from idea into master」（将 idea 分支上的提交合并到 master 分支上）。
5. 点击该按钮，将自动进行分支合并，并且会创建一个融合版本。
6. 在历史线上，将采用特殊图标标识。



底层历史线变成了这样：



新生成了一个 C5，这是一个「融合版本」（Merge Commit）。

- 这个合并挺特殊，里面一般没有修改内容，它的作用主要是把两个分支合并起来。
- 怎么合并的呢？
 - 把 master 的内容 sync 到 github.com 上，然后查看一下这个 merge commit，会发现它有两个 parent。



- merge 之后，master 分支指针指向了 merge commit，也就自动拥有了 idea 分支上的 C3 这个版本了。idea 分支一般这儿就可以删除了。

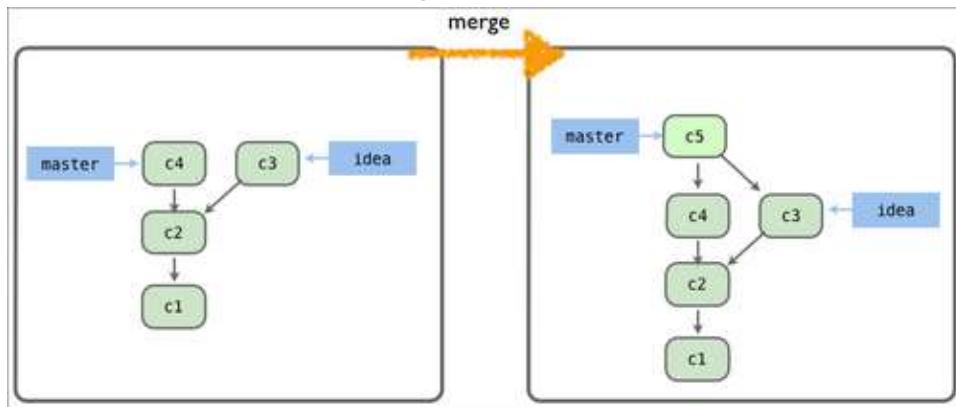
代码冲突 conflicts

实际（并行）开发的过程难免会遇到在不同的分支上修改同一个文件。

在不同分支上修改不同的文件，不会出现冲突

实际中经常有这样的情况，我正在 `idea` 分支上开发一个比较大的功能。但是这个时候突然发现了一个紧急的问题需要修复，所以我会直接到 `master` 分支上，做一个 `commit` 来解决这个紧急的问题。然后会来继续到 `idea` 上开发。

其他的情形也有，总之这样就会出现，两个不同分支上并行开发，同时都有新的 `commit`，这个一般没有问题，一样可以直接 `merge`，如下图



在不同分支上修改相同文件，出现冲突需要手动 merge

但是如果在两个分支上改动了同一个地方，合并的就会出现代码冲突。因为 git 不知道该听哪个分支的，所以只能报出冲突的位置，让开发者手动解决。

来具体操作一下。在 `idea` 分支上，改动 `README.md` 文件中的一行，比如改成 `AAA`，`commit` 了，然后切换到 `master` 分支上，把这一行的内容改为 `BBB`，也一样做 `commit`。这样再到客户端，打开 `merge view` 把 `idea` 分支 `merge` 到 `master` 之中，操作不会直接成功，而是会看到下面的代码冲突界面。

1. 在「其他分支」页面，点击「Update from master」。（新版本只能从「其他分支」中合并冲突，并在「其他分支」修改，这样可以确保 `master` 分支上的代码是稳定的。）
2. 点击「View conflict」可以看到冲突的内容。
3. 我们看到冲突的文件内容。下面将进行详细解释。

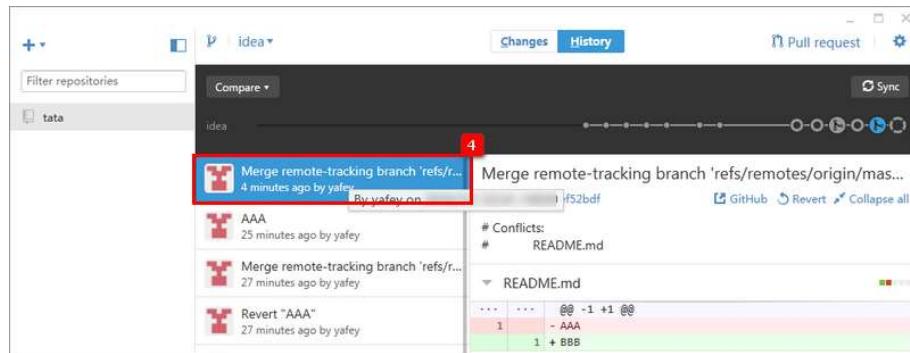
冲突文件的内容如下：

```
1. @@ -1 +1,5 @@
2. <<<<< HEAD
3. AAA
4. =====
5. BBB
6. >>>>> refs/remotes/origin/master
```

- 上面的 `HEAD` 是代表当前分支，此刻对应我的情形就是 `idea`。
- `=====` 就是两个冲突代码块的分界线。上面的代码就是 `idea` 分支上的，下面的代码是 `master` 分支的。
- 解决冲突就是把上面的三行「冲突标示符」都删掉，然后修改代码。（具体采用哪个分支上的代码视具体情况而定，一般（此处）采用 `master` 分支上的代码）。
 - 修改后的文件内容如下所示：

```
1. BBB
```

- 回到客户端，点击 `Commit`，这样，这次分支合并就完成了，也会生成一个「`merge commit`」。



~~合并分支除了融合（merge）还有另外一种形式叫“变基”（rebase）这里暂时用不上，先不管。~~

合并远端分支

~~在特定条件下，点 sync 按钮两个分支合并不使用融合（merge）方式，而采用“变基”（rebase）方式，这样最终不会生成一个 merge commit。但是达成的效果是一样的，也是实现了两个分支代码的合并，处理冲突的方式也一样，所以暂时不必深究。~~

现在我本地仓库叫 coco，github.com 上托管了这个仓库。那么自然就有本地一个 master 分支，和远端一个 master 分支，这两个分支虽然名字都叫 master，但是本质上也是两个分支，也存在分支合并的问题。

无冲突

比如这样，我在 github.com 网页上，修改一下项目，把修改内容 commit 到 master 分支之上。这样，远端的 master 就比我本地的 master 分支多了一个 commit。此时我到客户端，点 sync 按钮执行同步，这样这个 commit 就会直接被拉（pull）到我本地，这个是前面提过的。

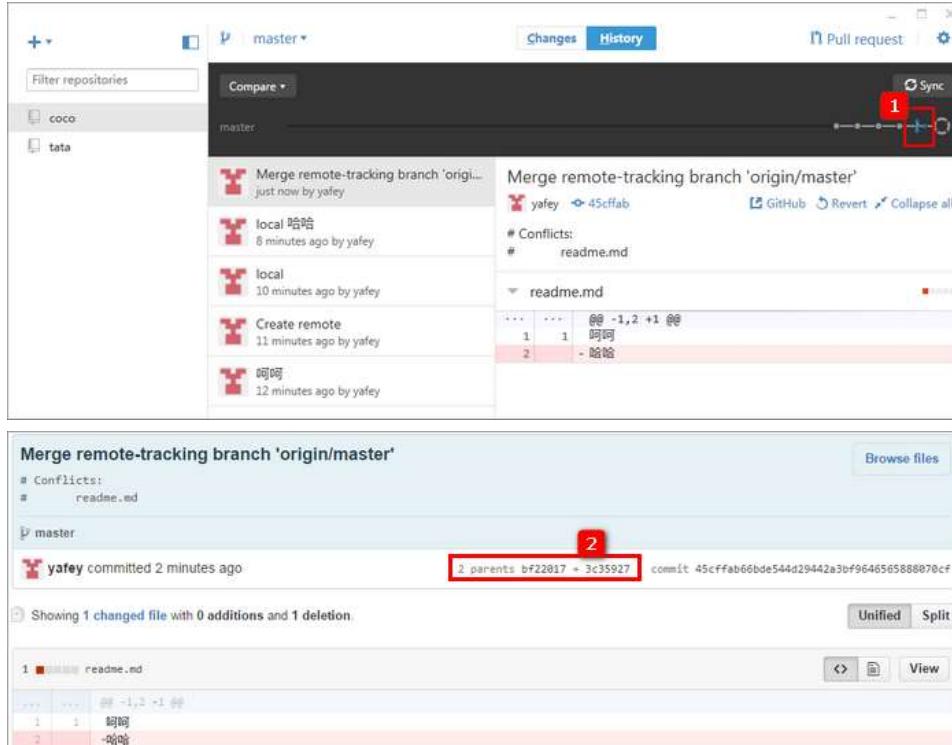
冲突文件

另外一种情况，假设通过客户端发布了一个文件 `readme.txt`（此时本地和远端都有一个 `readme.txt`，假设现在里面没有内容。）。

- 远端为这个文件添加内容 `remote`，并且 Commit。
- 我自己在本地添加内容 `local`，也做了一个 commit。

也就是本地的 `master` 和远端 `master` 出现了并行开发的情况，这种情况是非常常见的。这个时候我执行 `sync`，会发生什么呢？

这样我执行 `sync`，跟本地两个分支冲突合并是一样的，一般也会生成一个 `merge commit`，在本地客户端（1）和 `github.com` 上（2）的历史线都可以看到。



Section 6 团队协作流程

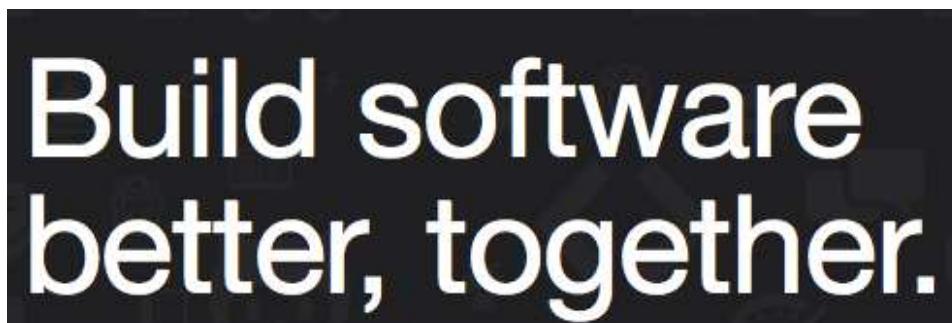
前几天还都是一个开发者唱独角戏。但是尽管如此也可以看出 git 带来的便利了，比如代码写错了可以回滚，为了新功能开发可以开新分支等等。但是 `git` 和 `github` 更大的威力在于协作。

聊了这么多天的 `github`，有必要稍微停下来，再想想究竟什么是 `github`。没错，前面的使用中也看到了，`github` 是 `git` 仓库的托管平台，让我们的项目仓库可以方便的备份同步。但是其实也许比这个还要重要的是，`github` 是一个大家一起协作做项目的平台，是一种开发者的工作方式，引导一种看着不像流程的一种真正健康轻便的开发流程。

在 [How Github Use Github To Build Github](#) 的演讲中，Zach Holman 说，

多年来我一直在寻找做软件正确的方式，现在我想我找到了，`github` 就是这种方式。

究竟 `Github` 是什么的问题从它的标语中也可以看出



Github 多年来总结出来一套自己的团队协作流程，简单而且强大，叫做 **Github Flow**，网站上的各个功能都是围绕着这个流程来开发的。另，中文版的 **Github Flow** 在[这里](#)（参考 [Github Flow 中文版 github 地址](#)）。

要了解一个流程，没有什么比跑一个最简单的实际例子更好的方式了，官方给出的 [Hello World](#) 就是服务于这个目的。

不过这个 [Hello World](#) 用的是纯粹的网页来实现整个流程。咱们今天用网页配合 [Github For Mac](#) 客户端来完成这个流程。

什么是 Github Flow ?

说白了，就是给一个项目开发新功能要走的几步。整个过程的核心是「拉取请求（Pull Request）」（已 master 角度来考虑，master 把其他分支的代码拉到 master 分支上）。

第一步，创建新的话题分支。如下图所示

创建一个分支

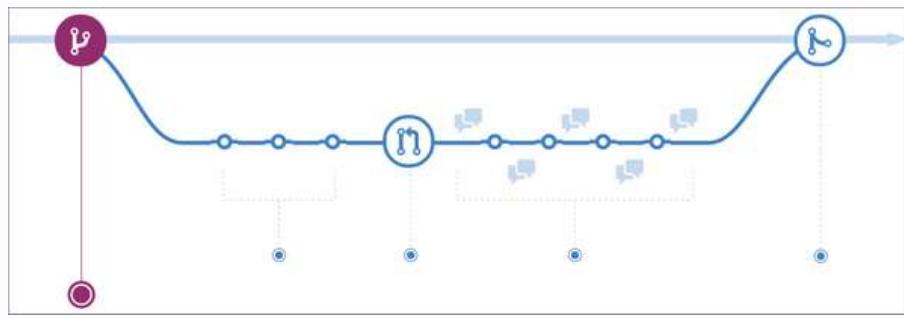
当你在开发一个项目的时候，一般在同一时刻你会同时开展多个想法，其中一些比较成熟了，另一些还是很初级。有了分支就可以很好的来进行管理了。

当你在项目中创建一个分支的时候，你正在搭建一个可以尝试新想法的环境。你在新分支上所做的修改不会影响到 [主（master）](#) 分支，所以你能自由实验和提交修改。在被你的同事审核之前，你的分支是不会被合并的，所以一切都是安全的。

深度技巧

分支是 Git 的一个核心概念，整个 GitHub Flow 也是基于它的。最重要的规则只有一个：[主（master）](#) 分支上的任何内容都要保证是可部署的。

正因为如此，当开发一个新功能或者修复错误的时候，你的新分支独立于主分支是极其重要的。你的分支名应该见名知义（例如，[refactor-authentication](#), [user-content-cache-key](#), [make-retina-avatars](#)），以便让其他人知道你正在做什么工作。



第二步，不断实现功能，做成一个个新版本。

添加新版本

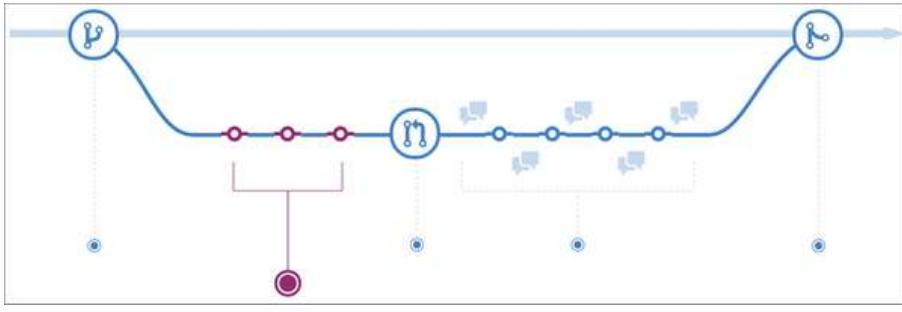
一旦你的分支创建好之后，就可以开始做些修改了。不论何时你添加，编辑或者删除一个文件，你就做一个版本，然后把它们添加到你的分支中。

添加版本的过程就跟踪了你在一个功能分支上的工作进展。

版本也为你的工作创建了一个透明的历史，其他人可以跟随着去理解你所做的工作以及为什么要这样做。每一个版本都有一条相关联的版本信息，版本信息是一段描述，解释了为什么要做出这样一个特定的修改。此外，每一个版本都可以看作是一个独立的修改单元。这样如果你不小心改错了，或者是改变了开发思路的时候，就可以来回滚修改了。

深度技巧

版本信息很重要，因为一旦你的修改被推送到服务器上，它们会以一个一个版本的形式显示。通过书写清楚的版本信息，你可以更容易让其他人跟上你的思路并提供反馈。



第三步，发起“拉取请求”（**Pull Request**），后面简称**PR**吧。

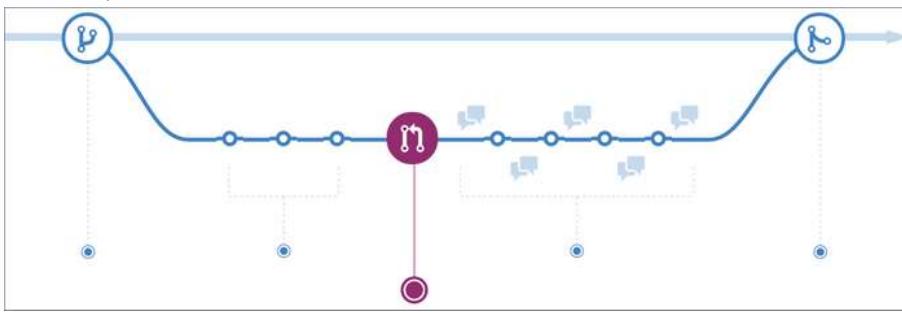
开启一个 Pull Request

Pull Request 用来发起对你做的各个版本的讨论。因为 **Pull Request** 与底层的 Git 仓库代码是紧密相关的，任何人都能确切地看到一旦他接受了你的 **Pull Request** 会有那些代码合并进来。

在开发过程中的任意时点，你都可以开启一个 **Pull Request**: 当你有很少或没有代码，但想要分享一些截图或基本想法的时候，当你陷入困境需要帮助和建议的时候，或者当你准备好让他人审核你工作的时候。通过在你的 **Pull Request** 信息中使用 **Github** 的 **@mention** 系统，你可以向特定的人或团队请求反馈，不管他们就在大厅的那边，还是离你有十个时区之遥。

深度技巧

Pull Requests 对贡献开源项目和管理共享仓库的变动是非常有用的。若你正使用 **For k & Pull** 模式，**Pull Request** 提供了一种方式来通知项目维护者你希望他们考虑一下你提交的修改。若你正使用一个共享仓库模式，在提议修改被合并到主分支中之前，**Pull Request** 可以启动对修改代码的审核和讨论。



第四步，大家讨论。这是一个代码审核的过程。

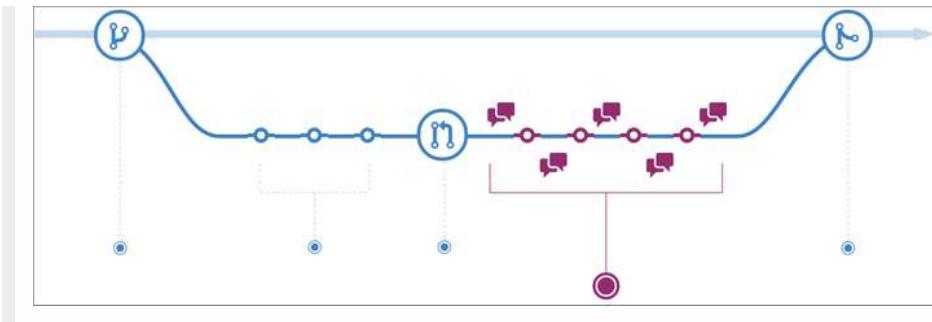
讨论和代码审核

一旦开启了一个 **Pull Request**，审核你修改的人或团队会来提出问题和评论。有可能是代码风格不符合项目规范，也或者代码忘了单元测试，也可能各方面都没问题。**Pull Request** 就是为了鼓励这种类型的讨论而设计的。

根据对你所做版本的讨论和反馈，你也可以继续往你的分支上推送代码。若有人评论说你忘记做某件事或你的代码中存在 bug，你可以在你的分支中修复它，并提交这些修改。**Github** 将会在同一个 **Pull Request** 中展示你的新修改和任何新的反馈。

深度技巧

Pull Request 中评论是用 **Markdown** 格式书写的，所以你可以在评论中嵌入图片和 emoji 表情符号，使用带有格式的文本块，和其它轻量级格式。



最后，把话题分支的内容合并到 master。

合并分支，然后部署

一旦大家审核了你的 Pull Request 并且所有代码通过了测试，就是可以把你的代码合并到主分支了。

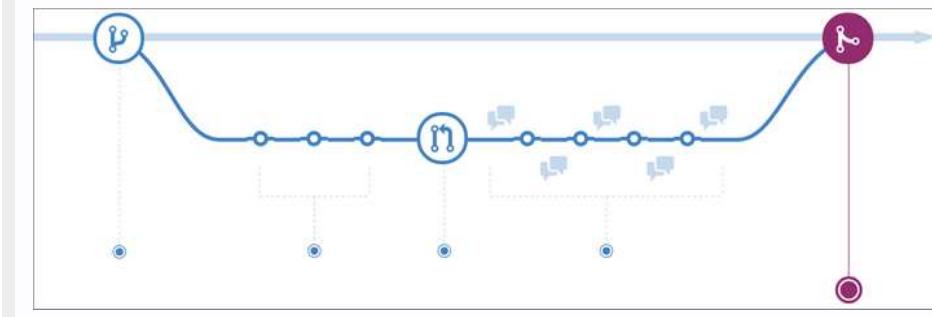
在把代码合并到 GitHub 上的仓库之前，如果你想测试代码，你可以先在本地执行合并操作。在你没有仓库的推送权限的情况下，这也是很方便的。

一旦合并之后，Pull Request 会保留代码的历史修改记录。因为它们是可搜索的，它们让人可以回到过去，去理解为什么做这个决定以及怎样做的决定。

深度技巧

通过在你的 Pull Request 中包含某些特定关键词，你就能用代码关联 issues。在你的 Pull Request 被合并的时候，与其相关的 issues 也会关闭。

例如，输入这个短语 Closes #32 将会关闭仓库中编号为32的 issue。更多信息，查看我们的 [帮助文档](#)。



示例：下面是一个实际例子。

给队友添加写权限

现在我和 @billi66 要合作开发一个新项目。于是我就来创建这个项目。

下面就继续在 coco 这个项目上做演示。

现在我要把 @billi66 添加进来，让她也具有项目的修改权限了。如何来做呢？把她添加成项目的“协作者”（ collaborator ）就行了。coco 本身是一个私有仓库，但是对于开源项目也是一样可以添加 collaborator 的。

首先到项目页面，点击 `Settings` 一项，

A screenshot of a GitHub repository page for 'happypeter / coco'. The repository is private. At the top right, there are buttons for 'Unwatch', 'Star', 'Fork', and 'Code'. Below the repository name, it says 'test repo for gitbeijing book — Edit'. It shows 11 commits, 1 branch, 0 releases, and 1 contributor. A dropdown menu shows 'branch: master'. The commit history lists several commits, with one from 'happypeter' being the latest. On the right side, there's a sidebar with links for 'Issues', 'Pull requests', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. An orange arrow points to the 'Settings' link.

到项目的 `Settings` 页面，可以看到如下图所示的 `collaborator` 选项，输入框中输入 `bi` 就可以自动补齐出 `billie66` 了

A screenshot of the 'Collaborators' section of the GitHub 'Settings' page for the 'coco' repository. The sidebar on the left has 'Collaborators' selected, indicated by an orange arrow. The main area shows a search bar with 'bi' typed in, and a list of suggestions below it. One suggestion, 'billie66 Billie Zhang', is highlighted with a blue background, indicated by another orange arrow. To the right of the suggestions is a button labeled 'Add collaborator', also indicated by an orange arrow.

回车选中，然后点 `Add collaborator` 按钮，这样就完成了。

开话题分支并在上面开发

现在我和 @billie66 都对 coco 项目有写权限，对于非常有把握的代码，可以直接在本地 `master` 开发然后 `sync` 到远端 `master` 分支上面。但是如果是比较重要的功能，还是要发单独开“话题分支”（Topic Branch），这个是后面发 PR 的前提。

尽管所有的流程操作都可以在浏览器中完成。但是更为常见的情形是我和 @billie 会把代码 `clone` 到本地并进行新功能的开发，因为这样可以使用自己的编辑器以及测试工具。

现在我要对项目开发一个很大很大的功能，所以就先来开一个分支叫做 `describe-project`。名字是越清楚越好的，这样队友比较能一眼看出我在干什么。注意开新分支一定要在刚刚更新过的 `master` 的基础上开。开好之后把这个分支发布到远端，以后这个分支上每次实现一点小功能就 `publish` 到远端，这样的好处是队友可以随时看到我的进展。

这样我做了两个版本，而且都同步到远端仓库了。所以到 `describe-project` 分支的历史上可以看到多了下面两个新 `commit`。

The screenshot shows a GitHub repository named 'happypeter / coco'. It displays two sets of commits:

- Commits on Mar 20, 2015:**
 - add emoji** (authored 3 minutes ago)
 - add tagline** (authored 8 minutes ago)
- Commits on Mar 18, 2015:**
 - Merge remote-tracking branch 'origin/master'** (authored 2 days ago)
 - add local** (authored 2 days ago)

Two specific commits are highlighted with green arrows pointing to their detailed diff views:

- The commit **'add emoji'** has a red box around its diff view, which shows changes to README.md. The diff output is:

```
4 3 1 README.md
...
1 -这是世界上最牛的 demo 项目。
2 +这是世界上最牛的 demo 项目。
3 +:+1: :octocat:
```
- The commit **'add tagline'** has a red box around its diff view, which shows changes to README.md. The diff output is:

```
3 2 1 README.md
...
1 -test project for gitbeijing book
2 -AABB
3 +这是一个世界上最牛的 demo 项目。
```

功能实现了，可以发 PR 了。

发 Pull Request

PR 在整个 Github Flow 流程中占有核心位置。其实 PR 的目的就是讨论，且整个讨论过程是围绕着实打实的代码。

先到仓库页面，找到发 PR 的大绿按钮

The screenshot shows a GitHub repository page for 'test repo for gitbeijing book'. It displays the following information:

- 13 commits
- 2 branches
- 0 releases
- 1 contributor

Your recently pushed branches:

- describe-project** (25 minutes ago)

Branches:

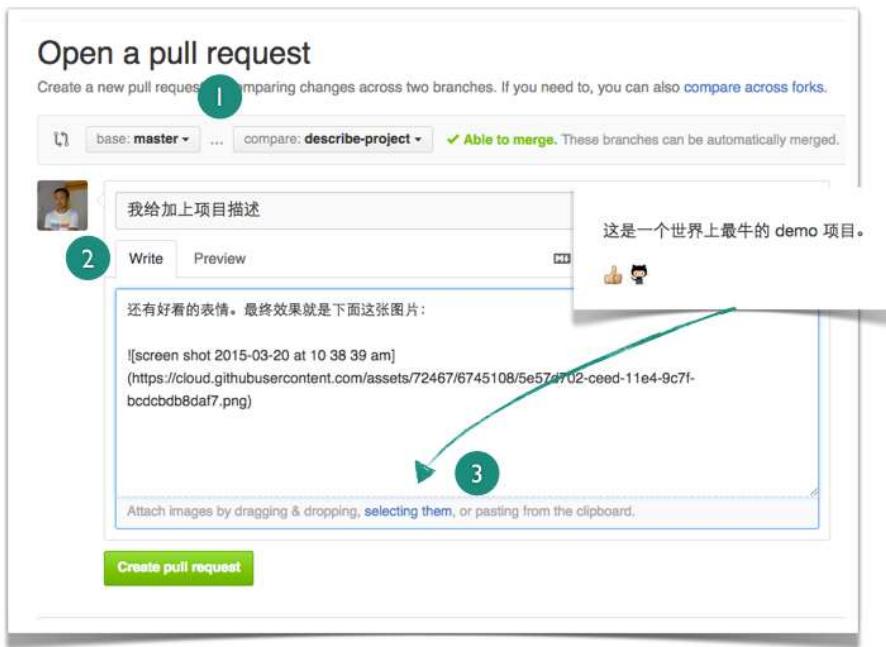
- coco / +**

This branch is 2 commits ahead of master

Buttons at the bottom:

- Pull Request
- Compare
- Compare & pull request (highlighted with an orange arrow)

下面图中显示的界面中，看1处，注意一下是拿出哪两个分支来进行对比。2处，我要填写一些内容，解释一下我的修改内容。3处，可以上传图片。



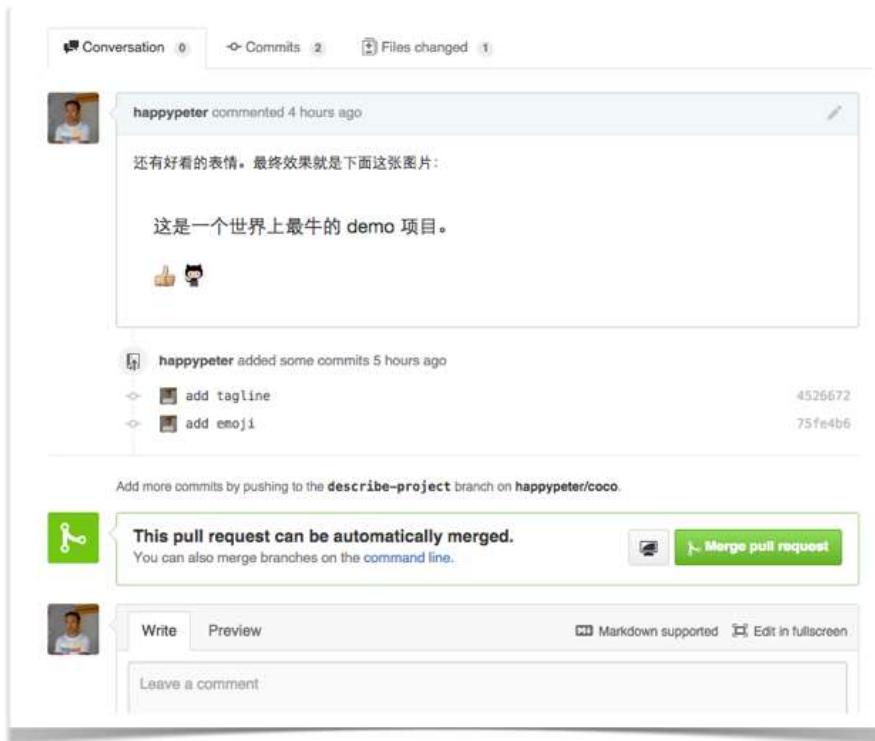
同样在这个页面上，滑动到下方还可以看到这次 PR 的具体对比出来的代码内容



点击 `Create Pull Request` 按钮，这样发 PR 就成功了。

补充一句。实际上，客户端中也可以发 PR，达成的效果跟网页中发是一样的，这里就不演示了。

讨论审核代码

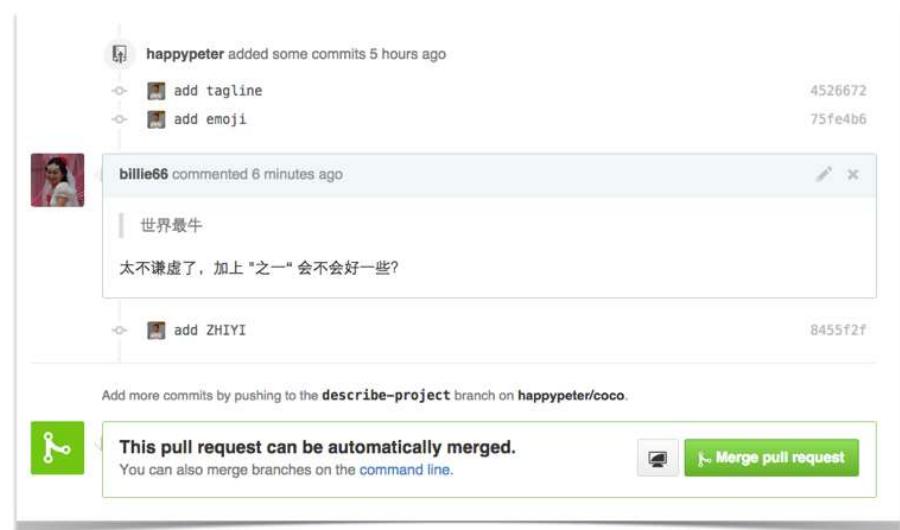


PR 的讨论过程也算是代码审核。不一定是一个老大审核小弟们的代码，可以是队友之间的互相审核。

@billie66 看到这个 PR 之后，就会发表她的意见了。

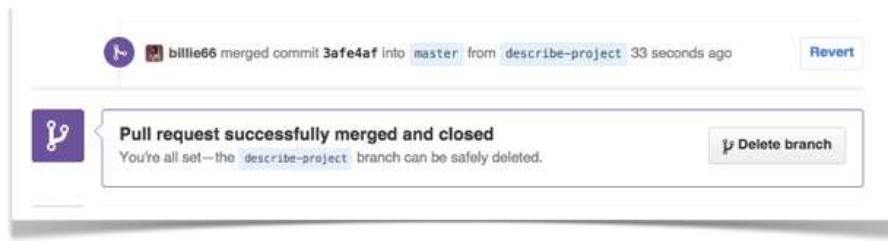


看到之后我觉得有道理，代码需要调整，那我现在是不是要撤销这 PR 重新发呢？不用。我只需要继续在 describe-project 分支上改代码然后再同步上来。



上面的图中可以看到，讨论不断继续，会形成一条由评论和代码穿插而成的一条线。最后达成一致，我或者 @billie 其中之一可以点一下上面的大大的 Merge Pull Request (融合拉取请求) 的按钮，这样话题分支上的代码就合并到 master 之上了。接下来 describe-

project 这个话题分支也就可以删掉了。然后关闭这个 PR 了。

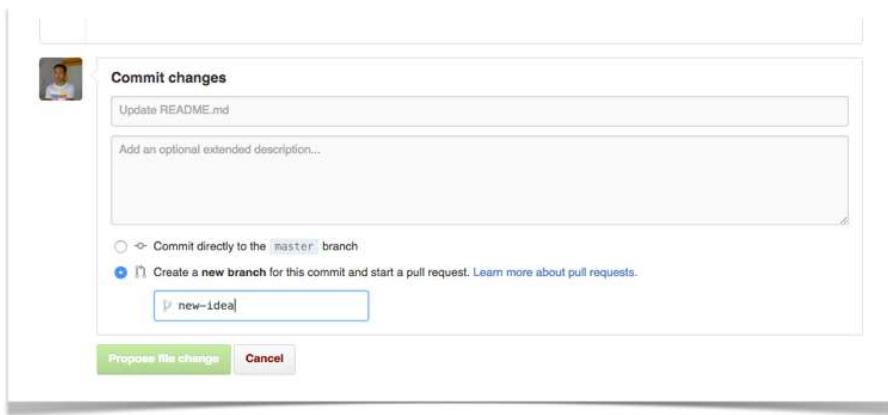


每一个 PR 都是开发历史上的一次小事件，很长时间过去之后，再看看当时的 PR 就可以看到当时为什么要开发这个功能，大家都是什么意见，都写了哪些代码，所以是项目发展的珍贵资料。从这个角度来说，即使一个 PR 没有被 merge 进 master，那它里面的代码和讨论的内容也会是非常有意义的尝试，也可能在未来会有很大的参考价值。

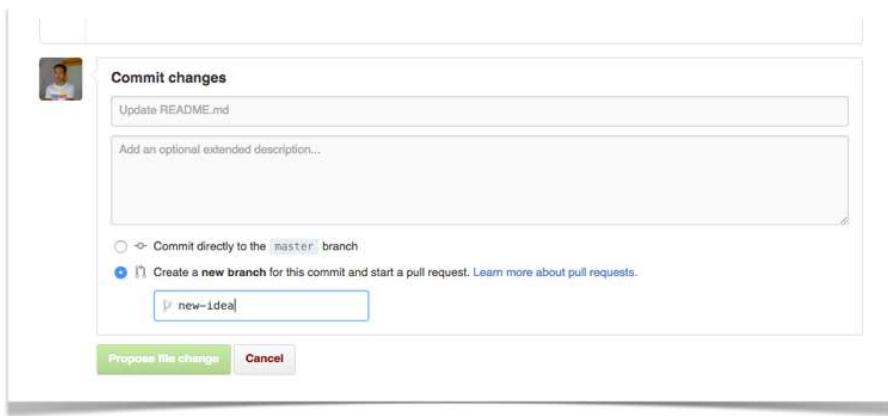
快速 PR

走一遍 Github Flow 其实方式并不唯一。前面讨论的，在自己的机器上改代码，用客户端作 commit，然后在网页上发 PR 是一种常见的方法。如果我只是改一个文件中的一个小地方，完全可以使用 github 网页功能提供的 快速 PR 这种方法。来演示一下。

网页界面中，找到我要修改的文件，点击 edit



然后在下面的界面中，可以直接填写一个 Topic 分支名，创建这个分支，并 commit 到这个分支上发 PR 了



说实话要只是一个小改动，即使是老手，你让我切换到编辑器和客户端，再跑一遍整个的这个发 PR 的流程，我也会觉得挺麻烦挺分心的。快速 PR 方法真的是非常方便。

总结

其实，每一个打开的 PR 都很类似于后面我们要讲的 Issue (事务卡片)，比如二者都可以用 Markdown 格式来写评论。这些技巧会在 Issue 相关的那一部分来介绍。

Section 7 开源项目贡献流程

Section 8 Github Issues

Section 9 Github Pages 搭建网站

Section 10 Github的秘密机关

Pre: 如何选择开源许可证?

Next: (转)(未整理)Git 教程

[Sign in to leave a comment.](#)

No Leanote account? [Sign up now.](#)

1 comments



weibo-ne0oo

点赞.

9 month ago
