Git 的基本操作、开发流程、实用技巧总结

发表于：2018-4-09 10:14　作者：未至　　来源：腾讯云

Git 有哪些比较好的实践？

Git 有一些成熟的开发流程，比较主流的有两种：基于功能分支的开发流程 和 GitFlow开发流程。

相对来时，我更推荐前者，如果是复杂的大型项目，推荐GitFlow开发流程。

接下来，简单介绍下这两种协作模式。

基于功能分支的协作模式

基于功能分支的开发流程其实就是一句话：用分支来承载功能开发，开发结束之后就合并到 master 分支。

他的优点是能够保证master分支的整洁，同时还能让分支代码逻辑集中，也便于 CodeReview。

分支命名规范

推荐使用如下格式：ownerName/featureName。

这样既便于知道分支覆盖的功能，也便于找到分支的负责人。以后清理分支的时候也很方便。

开发流程

从 master 切出一个新分支

```
git checkout -b qixiu/newFeature
```

开发一些新功能，然后提交

建议较多频次的提交代码到本地仓库，以便能够更灵活的保存或撤销修改。

此外为了保证提交日志的清晰，建议备注清楚的注释。

```
git status
git add files // 挑选需要提交的文件，或者全部提交
git commit -m '提交备注'
git push origin qixiu/newFeature
```

如果功能开发完成，可以发起一个CodeReview流程

如果代码测试通过，合并到 master，然后准备上线

```
// 冗余版 合并到 master
git checkout master
git pull -r origin master
git checkout qixiu/newFeature
git rebase master // 处理冲突
git checkout master
git merge qixiu/newFeature
git push origin master
// 精简版 合并到 master
git checkout qixiu/newFeature
```

git pull -r origin master // 将master的代码更新下来，并且rebase处理冲突

git push origin master // 将本地代码更新到远端

有几点需要注意：

不要在master合并代码，保证master的可用性很重要。

确保在正确的分支执行正确的操作。

无论是处理冲突还是更新远端代码，请保有敬畏之心。

到此，一个正常的基于功能分支的开发流程就完成了。接下来看看另外一个开发流程。

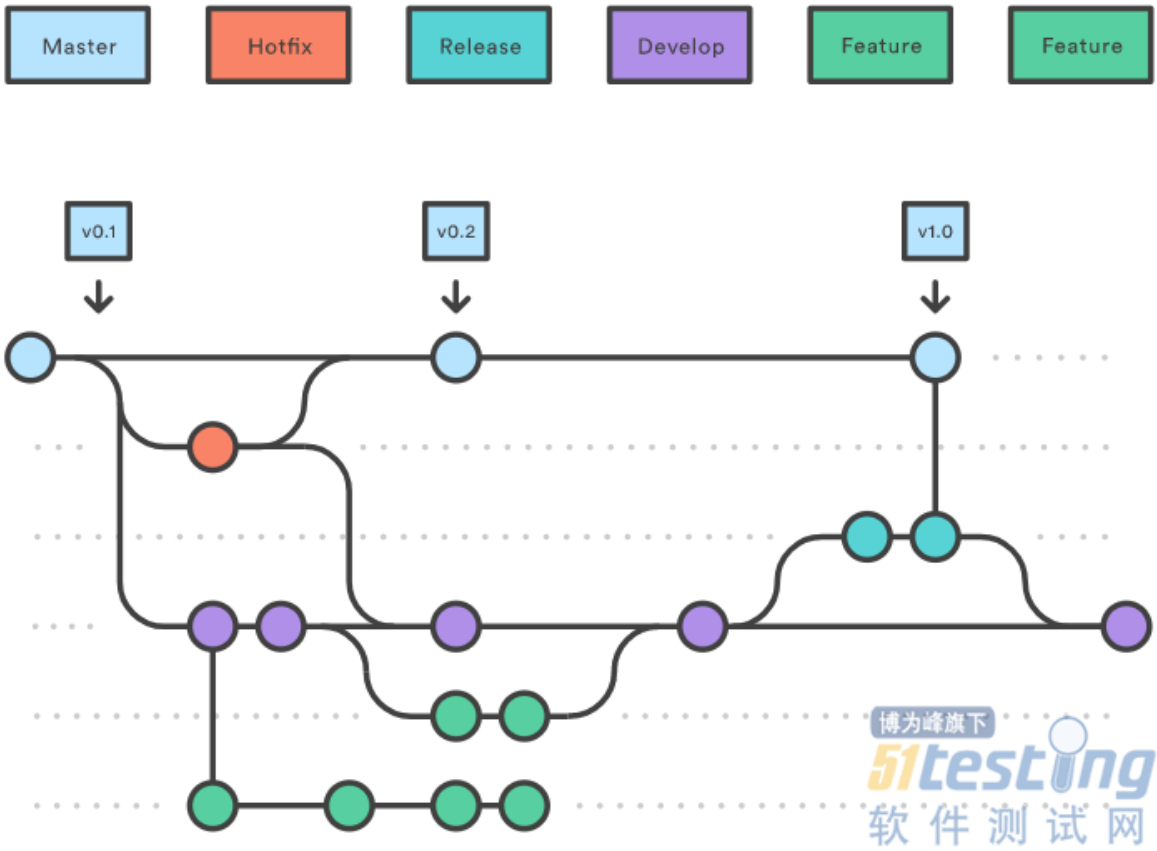
GitFlow 开发流程

GitFlow 比前文讲的基于功能分支的开发流程要复杂得多，它更适合大型的复杂项目。

它围绕项目发布流程定义了一个严格的分支模型，所有的开发流程都是围绕这个严格的分支模型进行。

而这个模型约定了每个分支的角色，以及他们如何沟通。

我们先来看看 GitFlow 开发流程中几个约定的分支，以及他们各自承担的角色是怎么样的？



Master分支：用于存放线上版本代码，可以方便的给代码打版本号。

Develop分支：用于整合 Feature 分支。

Feature分支：某个功能的分支，从 Develop 分支切出，并且功能完成时又合并回 Develop 分支，不直接和Master 分支交互。

Release分支：通常对应一个迭代。将一个版本的功能全部合并到 Develop 分支之后，从 Develop 切出一个Release 分支。这个分支不在追加新需求，可以完成 bug 修复、完善文档等工作。务必记住，代码发布后，需要将其合并到 Master 分支，同时也要合并到 Develop 分支。

Hotfix分支：紧急修复的分支，是唯一可以从 Master 切出的分支，一旦修复了可以合并到 Master 分支和 Develop 分支。

从每个分支的功能和约定可以看出，它流程多约束多，对于小规模应用并不适合。

当然 GitFlow 有一些辅助工具 gitflow 可以自动化的完成这些任务，对于大型项目也很有帮助。

前面讲了 Git 有哪些基本操作，然后介绍了两个主流的工作流程。

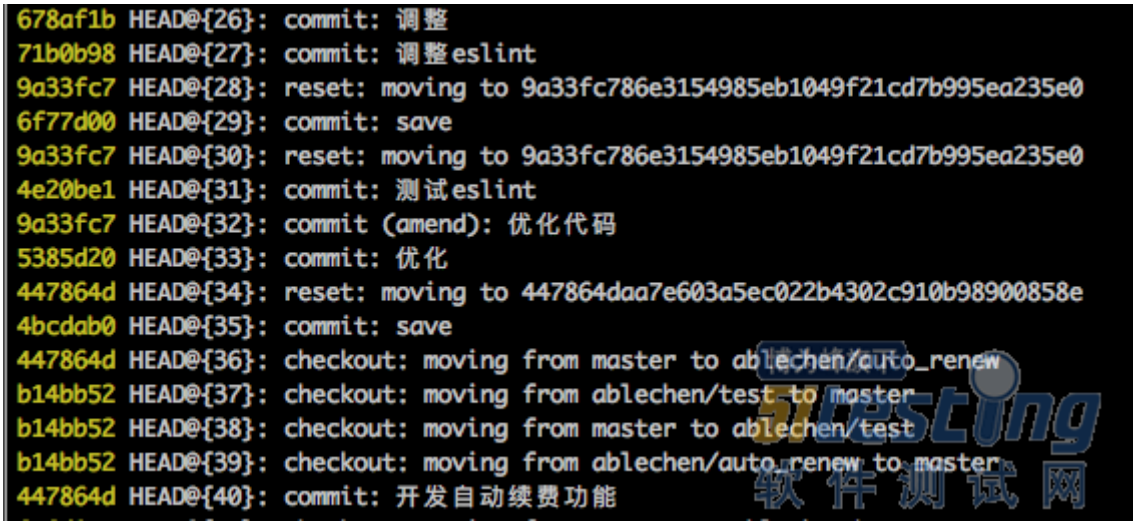
接下来我们看看 Git 有哪些特别的技巧值得一提。

Git 有哪些小技巧？

Git 操作除了基本的代码管理功能，还有一些小技巧能够让你眼前一亮。

git reflog，查看操作记录

这个我一定要放在第一个介绍，因为它曾经数次解救了我的代码



仔细看上图，reflog 记录了你所有的 git 命令操作，对于复原某些莫名其妙的场景或者回滚误操作有极大的帮助。

试想一个场景：你使用 git reset --hard commitID 把本地开发代码回滚到了一个之前的版本，而且还没有推到远端，怎么才能找回丢失的代码呢？

你如果使用 git log 查看提交日志，并不能找回丢弃的那些 commitID。

而 git reflog 却详细的记录了你每个操作的 commitID，可以轻易的让你复原当时的操作并且找回丢失的代码。

当然，如果你丢失的代码都没有提交记录，那么恭喜你，你的代码真的丢了。

压缩提交记录

这也是一个很实用的功能，前文提过，我们在开发中的时候尽量保持一个较高频率的代码提交，这样可以避免不小心代码丢失。但是真正合并代码的时候，我们并不希望有太多冗余的提交记录，而且 rebase 合并代码的时候，会把每个 commit 都处理一下，有时候会造成冗余的工作。

所以，压缩日志之后不经能让 commit 记录非常整洁，同时也便于使用 rebase 合并代码。

那么，如何压缩commit记录呢？

使用 git log 找到起始 commitID

git reset commitID，切记不要用 --hard参数

重新 git add && git commit

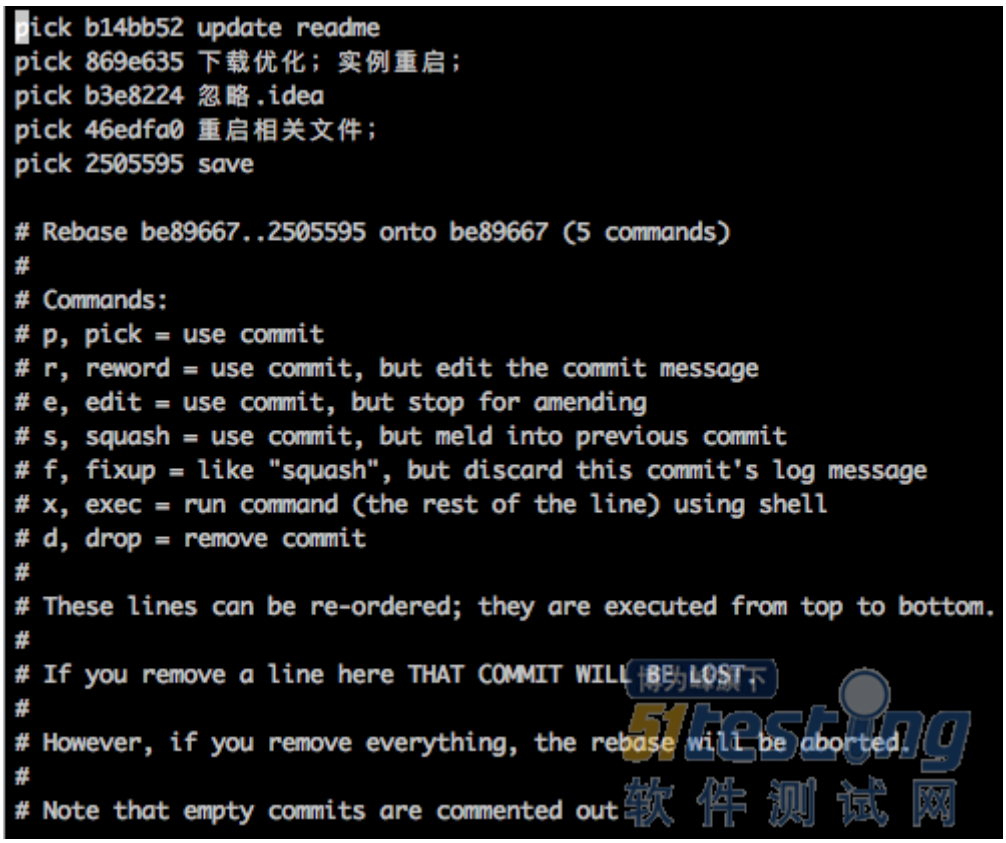
git push -f origin branchName，因为会有冲突，所以需要强制覆盖远端分支，请务必谨慎。

合并到 master 中，然后更新远端 master。

此外还有两种压缩日志的办法：

git commit --amend：追加 commit 到上一个 commit 上。

git rebase -i：通过交互式的 rebase，提供对分支 commit 的控制，从而可以清理混乱的历史。



从实际应用来说，三种日志压缩都很优秀，git reset更简单，git rebase -i更细腻。

git rebase，合并代码

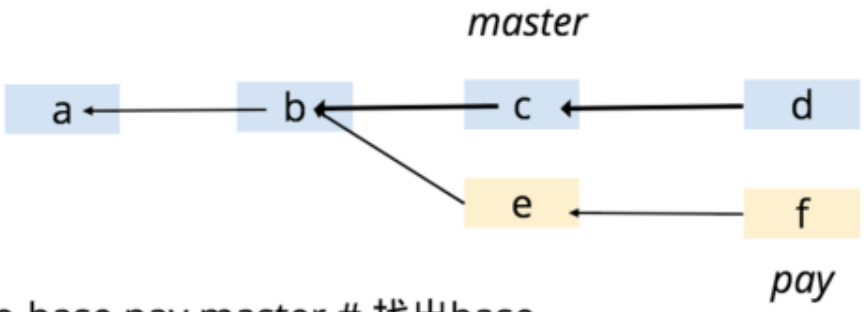
前文简单介绍了git rebase和 git merge 的区别，坦率讲，他们各有优劣。

git rebase能让你的 commit 记录非常整洁，无论是线上回滚还是 CodeReview 都更轻松；但却是一个有隐患的操作，使用时务必谨慎。

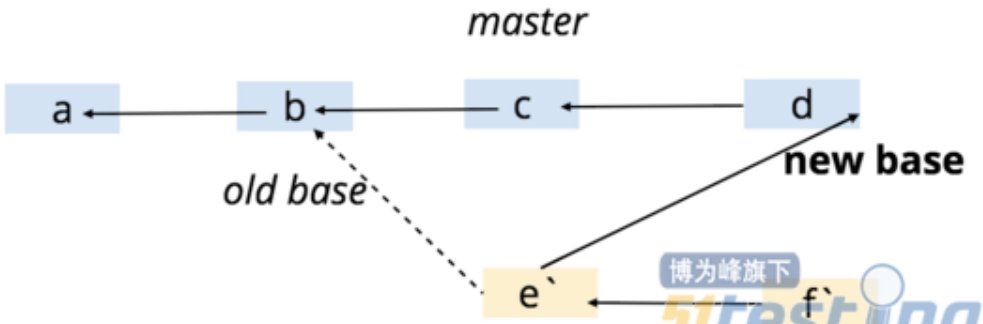
git merge 操作更安全，同时也更简单；但却会增加一些冗余的 commit 记录。

这儿简单说说 rebase 的合并流程和注意事项吧。看下图

rebase



git merge-base pay master # 找出base



把commit剪下来，再挨个打到new base上

有三个点需要注意：

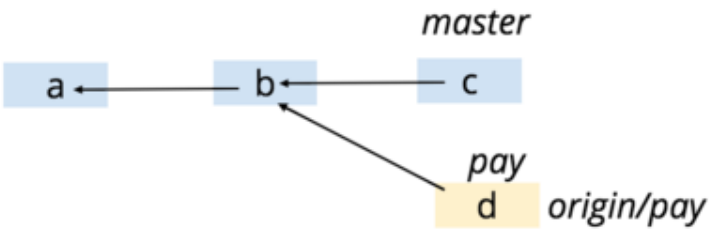
rebase 先找出共同的祖先节点

从祖先节点把 pay 分支的提交记录摘下来，然后 rebase 到 master 分支

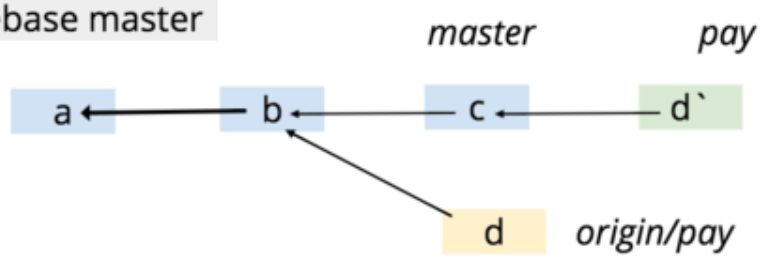
rebase 之后的 commitID 其实已经发生了变化

尤其是第三点，经常会让人误操作，所以务必注意。

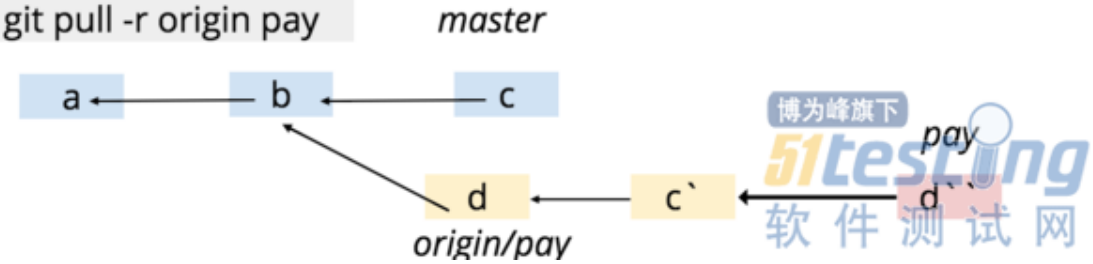
试想一下，开发过程中，如果我们频繁的 rebase master 分支，会有什么后果呢？



git rebase master

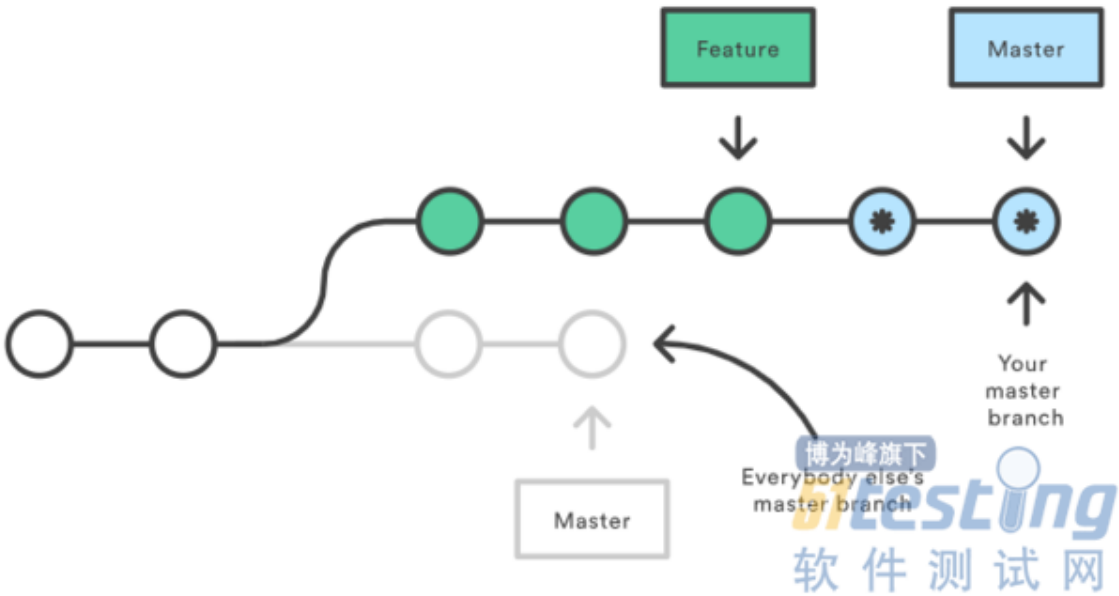


git pull -r origin pay



当你不断 rebase master 的时候，其实你本地的 d 都变成了 d’ ，再要和远端 pay 分支保持一致，你的本地分支 commit 记录已经不堪入目了。

另外要注意，绝不要在公共的分支上使用 rebase！！！

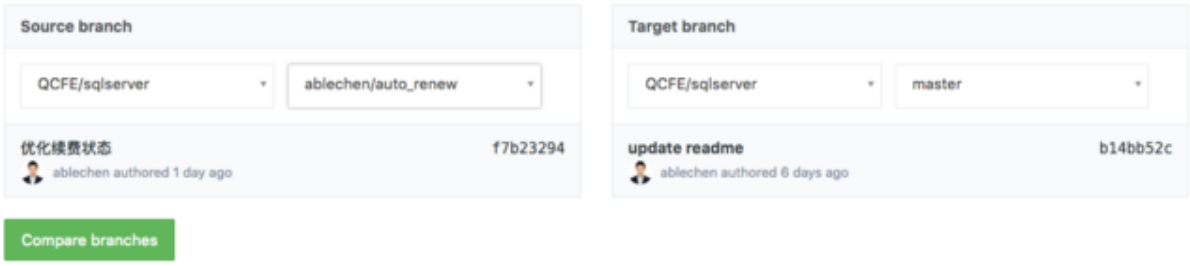


所以，为了安全，团队可以考虑采用 merge。

pull request，方便CodeReview

Git 不仅提供了代码托管以及代码开发的帮助，还提供了代码审核类似的功能。

当我们在功能分支开发完成之后，可以发起一个 pull request 请求，选择需要对比的两个分支



它会创建一个 pull request，制定相关人员来对代码进行 review。

通常情况下，团队应该鼓励交叉 review，涉及到公共代码的时候，一定要让相关人 review。

git hook，Git 的生命周期

这个大多数人应该都，听说过，git操作有它自身的生命周期，在不同的生命周期，我们可以做一些自动化的事情。

举两个简单的例子：

pre-commit的时候我们可以做 eslint

post-commit的时候，我们可以做利用 jenkins 类似的工具做持续集成

当然还有更多的声明周期，具体可以参考 Git 钩子

git submodule && git subtree，管理第三方模块

这两个命令通常用来管理公用的第三方模块。比如一些通用的底层逻辑、中间件、还有一些可能会频繁变化的通用业务组件。

当然，两者还是有区别的。

git submodule 主要用来管理一些单向更新的公共模块或底层逻辑。

git subtree 对于部分需要双向更新的可复用逻辑来说，特别适合管理。比如一些需要复用的业务组件代码。在我之前的实践中，我也曾用subtree来管理构建系统逻辑。

git alias，简化 Git 命令

我们可以通过配置 git alias 来简化需要输入的 Git 命令。

比如前文的 git subtree 需要输入很长的 Git 命令，我们可以配置 .git/config 文件来解决。

```
// git stpull appfe demo/xxx
// git stpush appfe demo/xxx
[alias]
stpull = !git subtree pull --prefix=$1 appfe $2 \
&& :
stpush = !git subtree pull --prefix=$1 appfe $2 \
&& git subtree split --rejoin --prefix=$1 $2 \
&& git subtree push --prefix=$1 appfe $2 \
&& :
```

总结说点啥？

该文首先介绍了 Git 常规操作

包括克隆代码、操作 commit、操作分支等。其实 Git 常规操作的命令并不多，请看第一部分的简单总结。

其次介绍了 Git 开发流程

该部分主要介绍了两种主流的开发模式：比较轻量的 基于功能分支的开发流程 和适合复杂项目的 GitFlow 开发流程 ，两种模式各有使用的场景，对于常规使用，前者就已经足够了。

最后介绍了一些 Git 实用技巧

主要包括：reflog 操作，压缩日志，rebase 的注意事项，利用 pull request 做 codeReview，利用 git hook 做一些自动化工作等。