Git 从入门到放不下

gafish Java后端 2019-10-16

作者 | gafish

链接 | github.com/gafish/gafish.github.com

Git简介

Git 是一种分布式版本控制系统，它可以不受网络连接的限制，加上其它众多优点，目前已经成为程序开发人员做项目版本管理时的首选，非开发人员也可以用 Git 来做自己的文档版本管理工具。

2013年，淘宝前端团队开始全面采用 Git 来做项目管理，我也是那个时候开始接触和使用，从一开始的零接触到现在的重度依赖，真是感叹 Git 的强大。

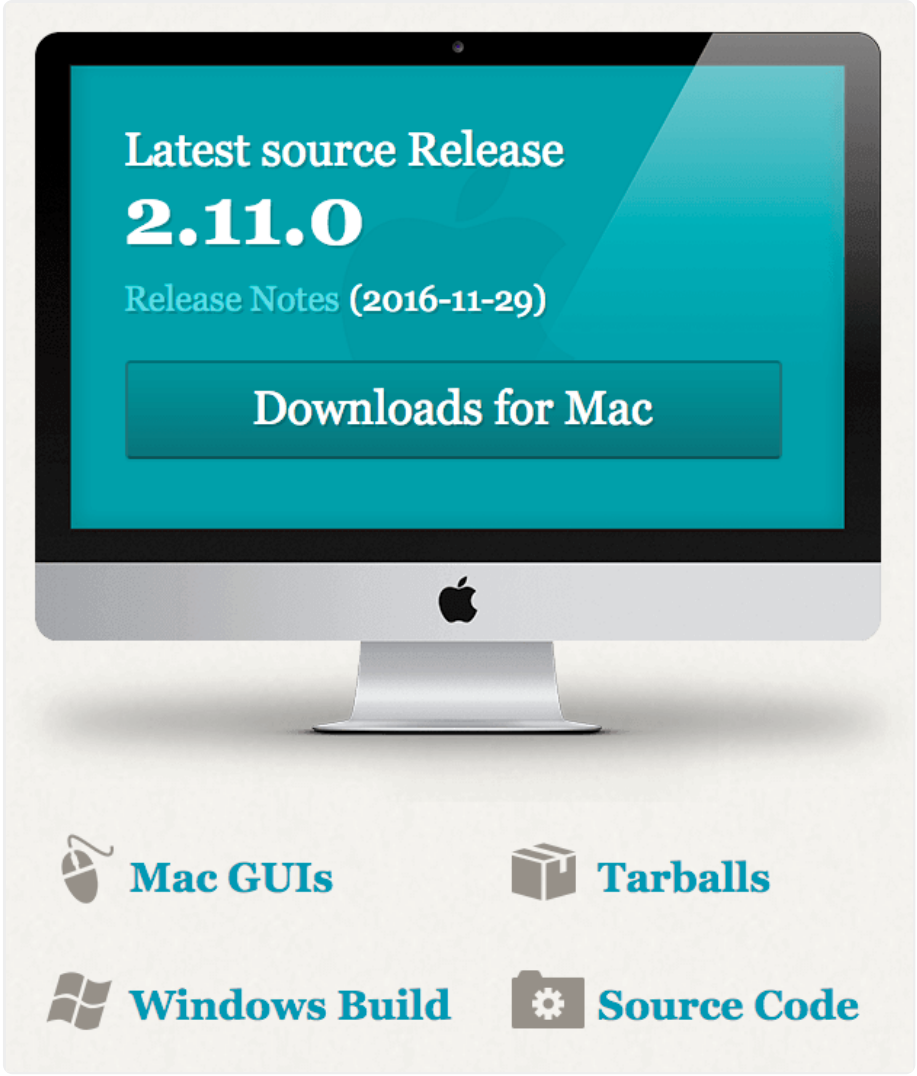
Git 的api很多，但其实平时项目中90%的需求都只需要用到几个基本的功能即可，所以本文将从 实用主义 和 深入探索 2个方面去谈谈如何在项目中使用 Git，一般来说，看完 实用主义 这一节就可以开始在项目中动手用。

> 说明：本文的操作都是基于 Mac 系统

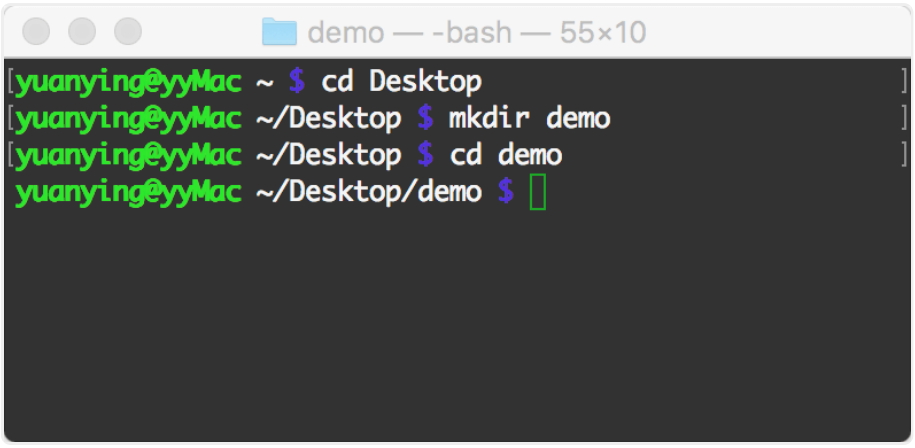
实用主义

准备阶段

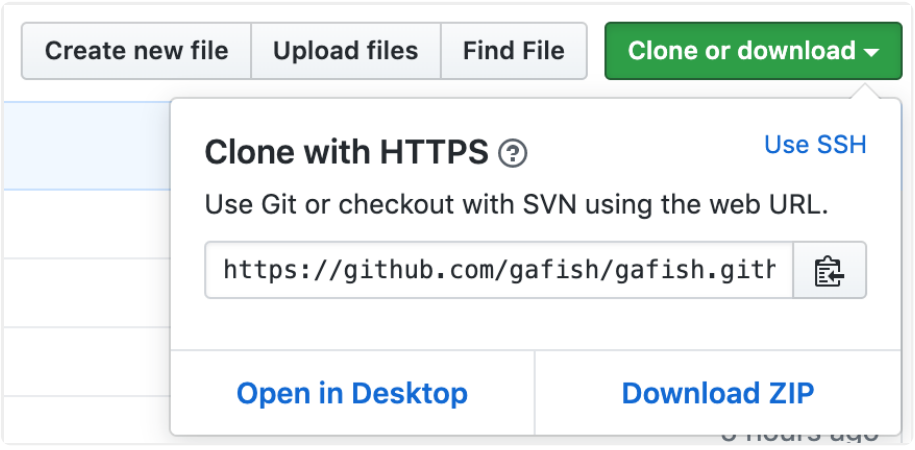
进入 Git官网 下载合适你的安装包，当前我下载到的版本是 2.11.0，本文也将在这个版本上演示效果。



安装好 Git 后，打开命令行工具，进入工作文件夹（为了便于理解我们在系统桌面上演示），创建一个新的demo文件夹。微信搜索 Java后端 关注，获取更多推送。



进入 Github网站 注册一个账号并登录，进入 我的博客，点击 Clone or download，再点击 Use HTTPS，复制项目地址 https://github.com/gafish/gafish.github.com.git 备用。微信搜索 Java后端 关注，获取更多推送。



再回到命令行工具，一切就绪，接下来进入本文的重点。

常用操作

所谓实用主义，就是掌握了以下知识就可以玩转 Git，轻松应对90%以上的需求。以下是实用主义型的Git命令列表，先大致看一下

- git clone
- git config
- git branch
- git checkout
- git status
- git add
- git commit
- git push
- git pull
- git log
- git tag

接下来，将通过对 我的博客 仓库进行实例操作，讲解如何使用 Git 拉取代码到提交代码的整个流程。

git clone

> 从git服务器拉取代码

```
git clone https://github.com/gafish/gafish.github.com.git
```

代码下载完成后在当前文件夹中会有一个 gafish.github.com 的目录，通过 cd gafish.github.com 命令进入目录。

git config

> 配置开发者用户名和邮箱

```
git config user.name gafish
git config user.email gafish@qqqq.com
```

每次代码提交的时候都会生成一条提交记录，其中会包含当前配置的用户名和邮箱。

git branch

> 创建、重命名、查看、删除项目分支，通过 Git 做项目开发时，一般都是在开发分支中进行，开发完成后合并分支到主干。

```
git branch daily/0.0.
0
```

创建一个名为 daily/0.0.0 的日常开发分支，分支名只要不包括特殊字符即可。

```
git branch -m daily/0.0.0 daily/0.0.
1
```

如果觉得之前的分支名不合适，可以为新建的分支重命名，重命名分支名为 daily/0.0.1

```
git branch
```

通过不带参数的branch命令可以查看当前项目分支列表

```
git branch -d daily/0.0.
1
```

如果分支已经完成使命则可以通过 -d 参数将分支删除，这里为了继续下一步操作，暂不执行删除操作

git checkout

> 切换分支

```
git checkout daily/0.0.
1
```

切换到 daily/0.0.1 分支，后续的操作将在这个分支上进行

git status

> 查看文件变动状态

通过任何你喜欢的编辑器对项目中的 README.md 文件做一些改动，保存。

```
git status
```

通过 git status 命令可以看到文件当前状态 Changes not staged for commit: （改动文件未提交到暂存区）

Onbranchdaily/0.0.1
Changesnotstagedforcommit:
 (use"gitadd<file>..."toupdatewhatwillbecommitted)
 (use"gitcheckout--<file>..."todiscardchangesinworkingdirectory)
  modified: README.md
nochangesaddedtocommit(use"gitadd"and/or"gitcommit-a")

git add

> 添加文件变动到暂存区

```
git add README.md
```

通过指定文件名 README.md 可以将该文件添加到暂存区，如果想添加所有文件可用 git add . 命令，这时候可通过 git status 看到文件当前状态 Changes to be committed: （文件已提交到暂存区）

Onbranchdaily/0.0.1
Changestobecommitted:
(use"gitresetHEAD<file>..."tounstage)
modified: README.md

git commit

> 提交文件变动到版本库

```
git commit -m '这里写提交原因
'
```

通过 -m 参数可直接在命令行里输入提交描述文本

git push

> 将本地的代码改动推送到服务器

```
git push origin daily/0.0.
1
```

origin 指代的是当前的git服务器地址，这行命令的意思是把 daily/0.0.1 分支推送到服务器，当看到命令行返回如下字符表示推送成功了。

Countingobjects:3,done.
Deltacompressionusingupto8threads.
Compressingobjects:100%(2/2),done.
Writingobjects:100%(3/3),267bytes|0bytes/s,done.
Total3(delta1),reused0(delta0)
remote:Resolvingdeltas:100%(1/1),completedwith1localobjects.
Tohttps://github.com/gafish/gafish.github.com.git
*[newbranch] daily/0.0.1->daily/0.0.1

现在我们回到Github网站的项目首页，点击 Branch:master 下拉按钮，就会看到刚才推送的 daily/00.1 分支了，微信搜索 Java后端 关注，获取更多推送。

git pull

> 将服务器上的最新代码拉取到本地

```
git pull origin daily/0.0.
1
```

如果其它项目成员对项目做了改动并推送到服务器，我们需要将最新的改动更新到本地，这里我们来模拟一下这种情况。

进入Github网站的项目首页，再进入 daily/0.0.1 分支，在线对 README.md 文件做一些修改并保存，然后在命令中执行以上命令，它将把刚才在线修改的部分拉取到本地，用编辑器打开 README.md ，你会发现文件已经跟线上的内容同步了。

如果线上代码做了变动，而你本地的代码也有变动，拉取的代码就有可能会跟你本地的改动冲突，一般情况下 Git 会自动处理这种冲突合并，但如果改动的是同一行，那就需要手动来合并代码，编辑文件，保存最新的改动，再通过 git add . 和 git commit -m 'xxx' 来提交合并。

git log

> 查看版本提交记录

```
git log
```

通过以上命令，我们可以查看整个项目的版本提交记录，它里面包含了提交人、日期、提交原因等信息，得到的结果如下：

```
commit c334730f8dba5096c54c8ac04fdc2b31ede7107a
Author: gafish <gafish@qqqq.com>
Date:   Wed Jan 11 09:44:13 2017 +0800
    Update README.md
commit ba6e3d21fcb1c87a718d2a73cdd11261eb672b2a
Author: gafish <gafish@qqqq.com>
Date:   Wed Jan 11 09:31:33 2017 +0800
    test
.....
```

提交记录可能会非常多，按 J 键往下翻，按 K 键往上翻，按 Q 键退出查看

git tag

> 为项目标记里程碑

```
git tag publish/0.0.1
git push origin publish/0.0.1
```

当我们完成某个功能需求准备发布上线时，应该将此次完整的项目代码做个标记，并将这个标记好的版本发布到线上，这里我们以 publish/0.0.1 为标记名并发布，当看到命令行返回如下内容则表示发布成功了

```
Total 0 (delta 0), reused 0 (delta 0
)
To https://github.com/gafish/gafish.github.com.git
 * [new tag]         publish/0.0.1 -> publish/0.0.
1
```

.gitignore

> 设置哪些内容不需要推送到服务器，这是一个配置文件

```
touch .gitignore
```

.gitignore 不是 Git 命令，而在项目中的一个文件，通过设置 .gitignore 的内容告诉 Git 哪些文件应该被忽略不需要推送到服务器，通过以上命令可以创建一个 .gitignore 文件，并在编辑器中打开文件，每一行代表一个要忽略的文件或目录，如：

```
demo.html
build/
```

以上内容的意思是 Git 将忽略 demo.html 文件 和 build/ 目录，这些内容不会被推送到服务器上

小结

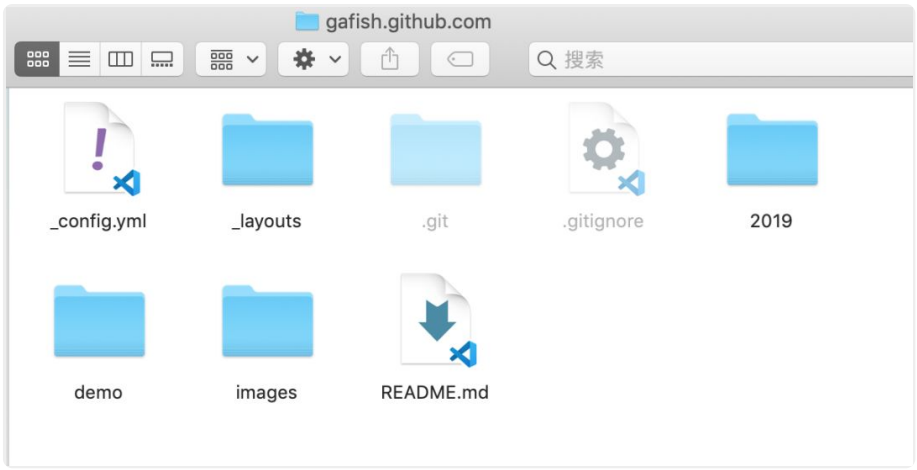
通过掌握以上这些基本命令就可以在项目中开始用起来了，如果追求实用，那关于 Git 的学习就可以到此结束了，偶尔遇到的问题也基本上通过 Google 也能找到答案，如果想深入探索 Git 的高阶功能，那就继续往下看 深入探索 部分。

深入探索

基本概念

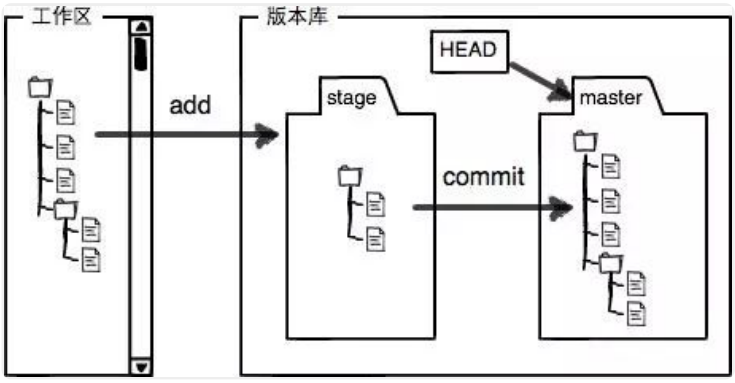
工作区（Working Directory）

就是你在电脑里能看到的目录，比如上文中的 gafish.github.com 文件夹就是一个工作区，微信搜索 Java后端 关注，获取更多推送。



本地版本库（Local Repository）

工作区有一个隐藏目录 .git，这个不算工作区，而是 Git 的版本库。

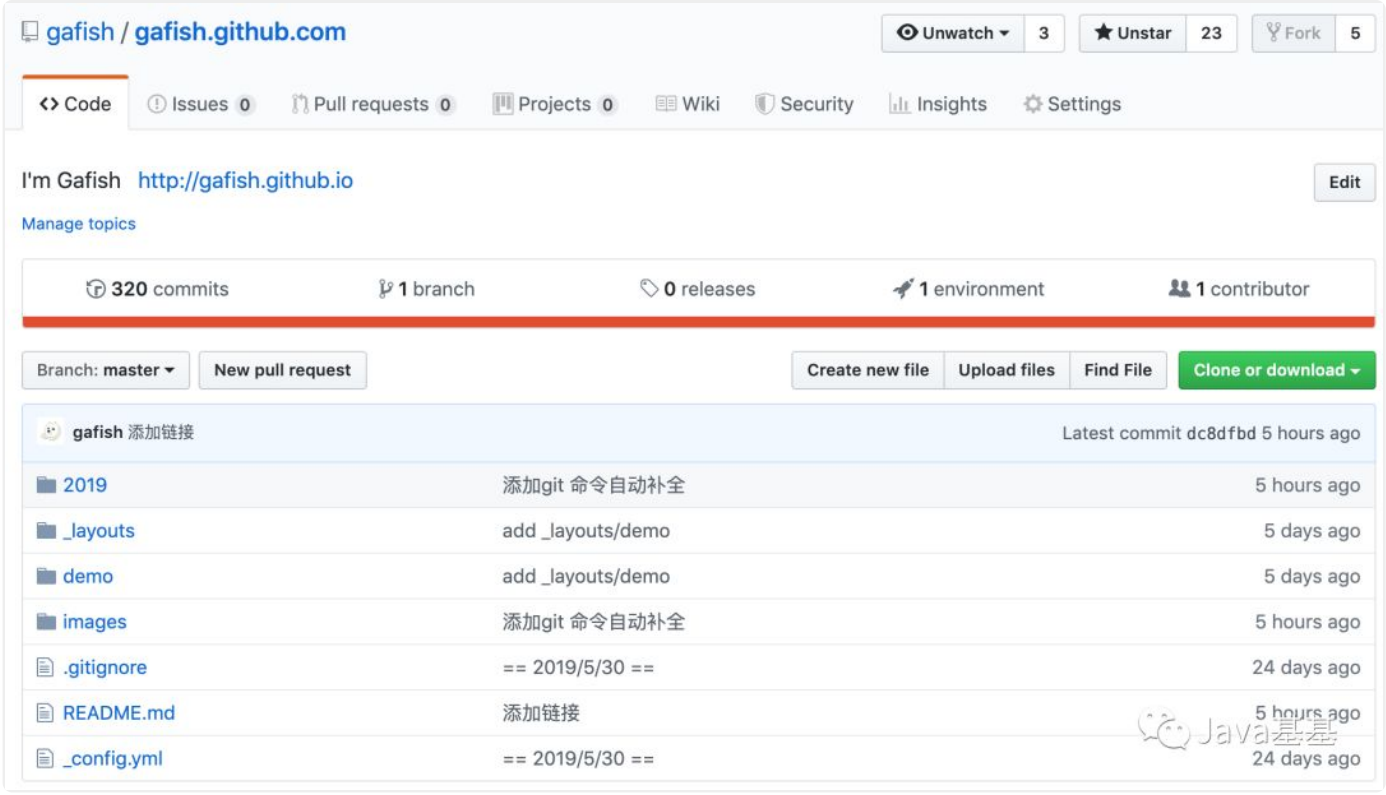


暂存区（stage）

本地版本库里存了很多东西，其中最重要的就是称为 stage（或者叫index）的暂存区，还有 Git 为我们自动创建的第一个分支 master，以及指向 master 的一个指针叫 HEAD。

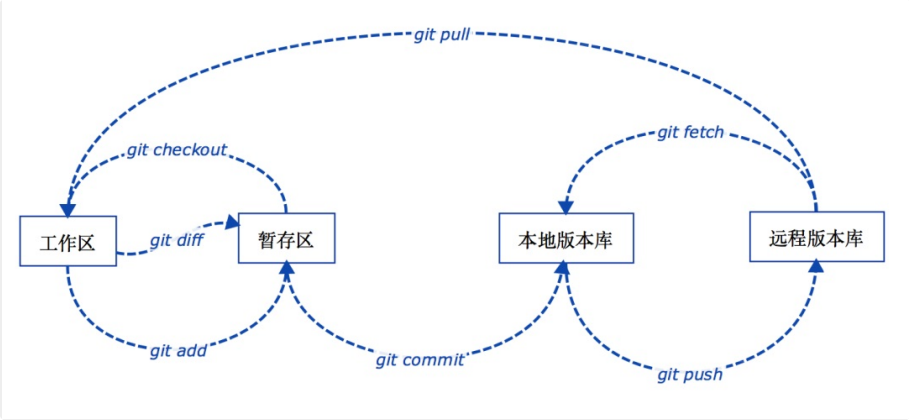
远程版本库（Remote Repository）

一般指的是 Git 服务器上所对应的仓库，本文的示例所在的github仓库就是一个远程版本库



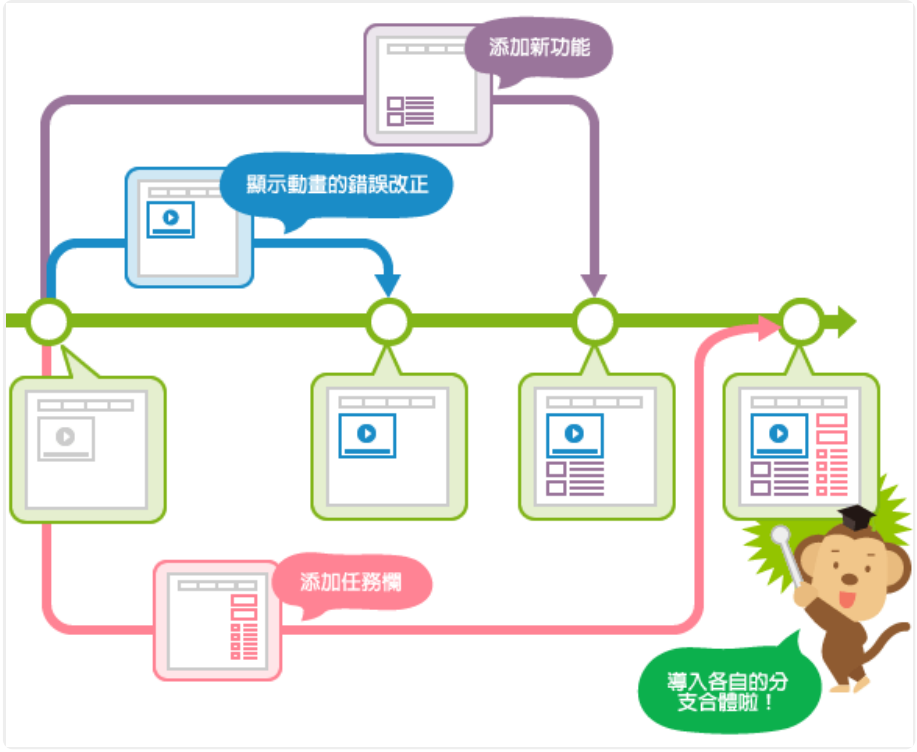
以上概念之间的关系

工作区、暂存区、本地版本库、远程版本库之间几个常用的 Git 操作流程如下图所示：



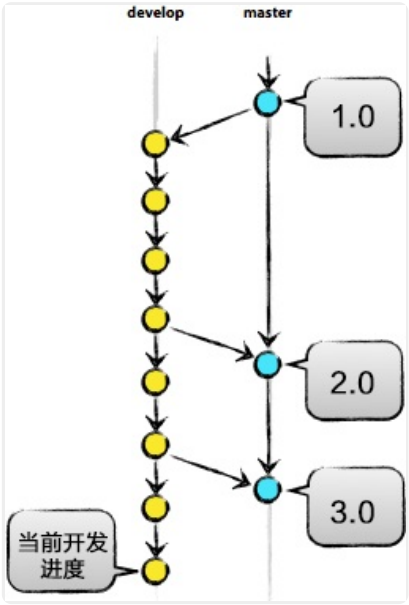
分支（Branch）

分支是为了将修改记录的整个流程分开存储，让分开的分支不受其它分支的影响，所以在同一个数据库里可以同时进行多个不同的修改，微信搜索 Java后端 关注，获取更多推送。



主分支（Master）

前面提到过 master 是 Git 为我们自动创建的第一个分支，也叫主分支，其它分支开发完成后都要合并到 master

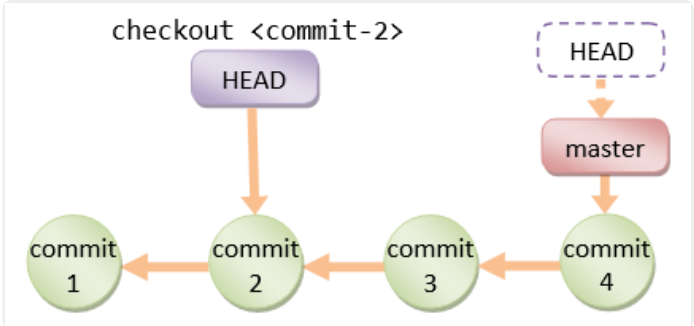


标签（Tag）

标签是用于标记特定的点或提交的历史，通常会用来标记发布版本的名称或版本号（如：publish/0.0.1），虽然标签看起来有点像分支，但打上标签的提交是固定的，不能随意的改动，参见上图中的1.0 / 2.0 / 3.0

HEAD

HEAD 指向的就是当前分支的最新提交



> 以上概念了解的差不多，那就可以继续往下看，下面将以具体的操作类型来讲解 Git 的高阶用法

操作文件

git add

> 添加文件到暂存区

```
git add -
i
```

通过此命令将打开交互式子命令系统，你将看到如下子命令

```
***Commands***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff      7: quit      8: help
```

通过输入序列号或首字母可以选择相应的功能，具体的功能解释如下：

- status：功能上和 git add -i 相似，没什么鸟用
- update：详见下方 git add -u
- revert：把已经添加到暂存区的文件从暂存区剔除，其操作方式和 update 类似
- add untracked：可以把新增的文件添加到暂存区，其操作方式和 update 类似
- patch：详见下方 git add -p
- diff：比较暂存区文件和本地版本库的差异，其操作方式和 update 类似
- quit：退出 git add -i 命令系统
- help：查看帮助信息

```
git add -
p
```

直接进入交互命令中最有用的 patch 模式

这是交互命令中最有用的模式，其操作方式和 update 类似，选择后 Git 会显示这些文件的当前内容与本地版本库中的差异，然后您可以自己决定是否添加这些修改到暂存区，在命令行 Stage deletion [y,n,q,a,d,/,?]? 后输入 y,n,q,a,d,/,? 其中一项选择操作方式，具体功能解释如下：

- y：接受修改
- n：忽略修改
- q：退出当前命令
- a：添加修改
- d：放弃修改
- /：通过正则表达式匹配修改内容
- ?：查看帮助信息

```
git add -
u
```

直接进入交互命令中的 update 模式

它会先列出工作区 修改 或 删除 的文件列表，新增 的文件不会被显示，在命令行 Update>> 后输入相应的列表序列号表示选中该项，回车继续选择，如果已选好，直接回车回到命令主界面，微信搜索 Java后端 关注，获取更多推送。

```
git add --ignore-removal .
```

添加工作区 修改 或 新增 的文件列表， 删除 的文件不会被添加

git commit

> 把暂存区的文件提交到本地版本库

```
git commit -m '第一行提交原因'  -m '第二行提交原因
'
```

不打开编辑器，直接在命令行中输入多行提交原因

```
git commit -am '提交原因
'
```

将工作区 修改 或 删除 的文件提交到本地版本库， 新增 的文件不会被提交

```
git commit --amend -m '提交原因
'
```

修改最新一条提交记录的提交原因

```
git commit -C HEA
D
```

将当前文件改动提交到 HEAD 或当前分支的历史ID

git mv

> 移动或重命名文件、目录

```
git mv a.md b.md -f
```

将 a.md 重命名为 b.md ，同时添加变动到暂存区，加 -f 参数可以强制重命名，相比用 mv a.md b.md 命令省去了 git add 操作，微信搜索 Java后端 关注，获取更多推送。

git rm

> 从工作区和暂存区移除文件

```
git rm b.md
```

从工作区和暂存区移除文件 b.md ，同时添加变动到暂存区，相比用 rm b.md 命令省去了 git add 操作

```
git rm src/ -r
```

允许从工作区和暂存区移除目录

git status

```
git status -s
```

以简短方式查看工作区和暂存区文件状态，示例如下：

Mdemo.html
??test.html

```
git status --ignored
```

查看工作区和暂存区文件状态，包括被忽略的文件

操作分支

git branch

> 查看、创建、删除分支

```
git branch -a
```

查看本地版本库和远程版本库上的分支列表

```
git branch -r
```

查看远程版本库上的分支列表，加上 -d 参数可以删除远程版本库上的分支

```
git branch -D
```

分支未提交到本地版本库前强制删除分支

```
git branch -vv
```

查看带有最后提交id、最近提交原因等信息的本地版本库分支列表

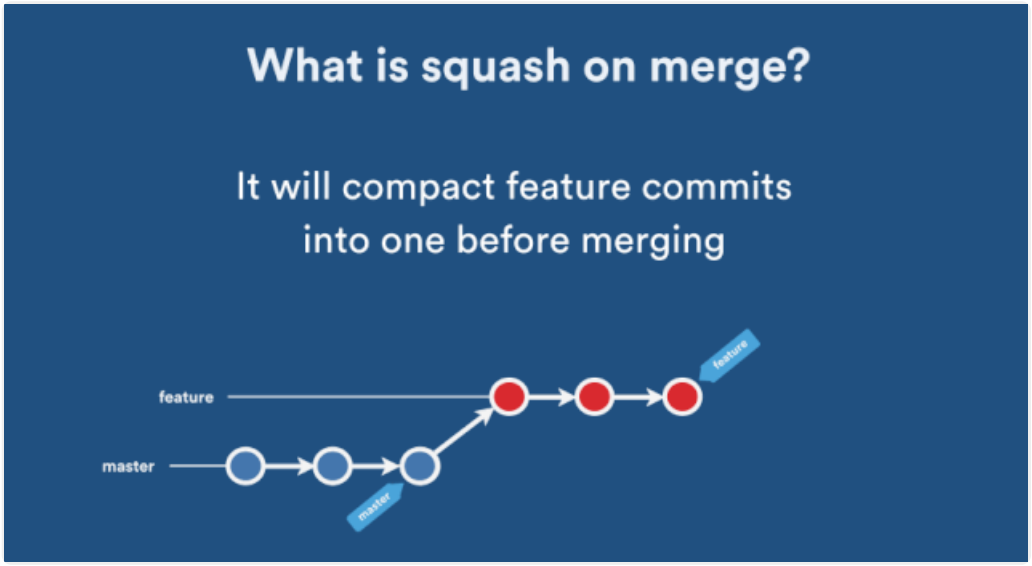
```
yuanying@yyMac ~/Desktop/demo/reading-list(daily/0.0.1) $ git branch -vv
* daily/0.0.1 2530d4b [origin/daily/0.0.1: ahead 16] mv
  master      22ff3cb [origin/master] Update README.md
```

git merge

> 将其它分支合并到当前分支

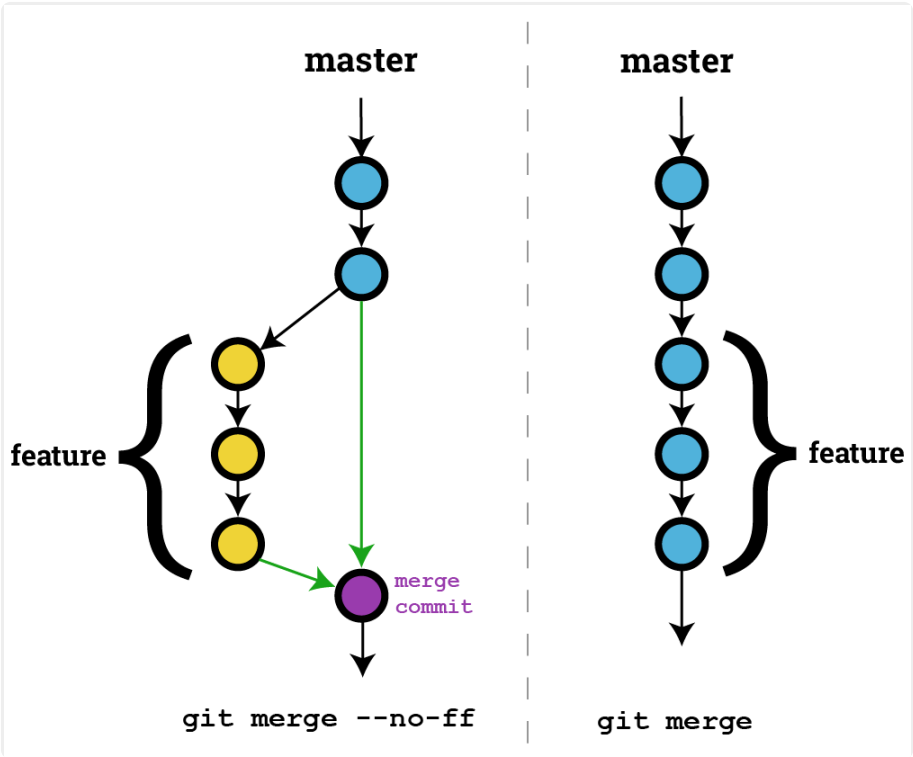
```
git merge --squash
```

将待合并分支上的 commit 合并成一个新的 commit 放入当前分支，适用于待合并分支的提交记录不需要保留的情况




```
git merge --no-f
f
```

默认情况下，Git 执行"快进式合并"（fast-farward merge），会直接将 Master 分支指向 Develop 分支，使用 --no-ff 参数后，会执行正常合并，在 Master 分支上生成一个新节点，保证版本演进更清晰。




```
git merge --no-edit
```

在没有冲突的情况下合并，不想手动编辑提交原因，而是用 Git 自动生成的类似 Merge branch 'test' 的文字直接提交

git checkout

> 切换分支

```
git checkout -b daily/0.0.
1
```

创建 daily/0.0.1 分支，同时切换到这个新创建的分支

```
git checkout HEAD demo.html
```

从本地版本库的 HEAD（也可以是提交ID、分支名、Tag名） 历史中检出 demo.html 覆盖当前工作区的文件，如果省略 HEAD 则是从暂存区检出

```
git checkout --orphan new_branch
```

这个命令会创建一个全新的，完全没有历史记录的新分支，但当前源分支上所有的最新文件都还在，真是强迫症患者的福音，但这个新分支必须做一次 git commit 操作后才会真正成为一个新分支。

```
git checkout -p other_branch
```

这个命令主要用来比较两个分支间的差异内容，并提供交互式的界面来选择进一步的操作，这个命令不仅可以比较两个分支间的差异，还可以比较单个文件的差异。

git stash

> 在 Git 的栈中保存当前修改或删除的工作进度，当你在一个分支里做某项功能开发时，接到通知把昨天已经测试完没问题的代码发布到线上，但这时你已经在这个分支里加入了其它未提交的代码，这个时候就可以把这些未提交的代码存到栈里。

```
git stash
```

将未提交的文件保存到Git栈中

```
git stash list
```

查看栈中保存的列表

```
git stash show stash@{0}
```

显示栈中其中一条记录

```
git stash drop stash@{0}
```

移除栈中其中一条记录

```
git stash pop
```

从Git栈中检出最新保存的一条记录，并将它从栈中移除

```
git stash apply stash@{0}
```

从Git栈中检出其中一条记录，但不从栈中移除

```
git stash branch new_banch
```

把当前栈中最近一次记录检出并创建一个新分支

```
git stash clear
```

清空栈里的所有记录

```
git stash create
```

为当前修改或删除的文件创建一个自定义的栈并返回一个ID，此时并未真正存储到栈里

```
git stash store xxxxxx
```

将 create 方法里返回的ID放到 store 后面，此时在栈里真正创建了一个记录，但当前修改或删除的文件并未从工作区移除

```
$ git stash create
09eb9a97ad632d0825be1ece361936d1d0bdb5c7
$ git stash store 09eb9a97ad632d0825be1ece361936d1d0bdb5c7
$ git stash list
stash@{0}: Created via "git stash store"
.
```

操作历史

git log

> 显示提交历史记录

```
git log -
p
```

显示带提交差异对比的历史记录

```
git log demo.html
```

显示 demo.html 文件的历史记录

```
git log --since="2 weeks ago"
```

显示2周前开始到现在的历史记录，其它时间可以类推

```
git log --before="2 weeks ago"
```

显示截止到2周前的历史记录，其它时间可以类推

```
git log -10
```

显示最近10条历史记录

```
git log f5f630a..HEAD
```

显示从提交ID f5f630a 到 HEAD 之间的记录，HEAD 可以为空或其它提交ID

```
git log --pretty=oneline
```

在一行中输出简短的历史记录

```
git log --pretty=format:"%h"
```

格式化输出历史记录

Git 用各种 placeholder 来决定各种显示内容，我挑几个常用的显示如下：

- %H: commit hash
- %h: 缩短的commit hash
- %T: tree hash
- %t: 缩短的 tree hash
- %P: parent hashes
- %p: 缩短的 parent hashes
- %an: 作者名字
- %aN: mailmap的作者名
- %ae: 作者邮箱
- %ad: 日期 (--date= 制定的格式)
- %ar: 日期, 相对格式(1 day ago)
- %cn: 提交者名字
- %ce: 提交者 email

- %cd: 提交日期 (--date= 制定的格式)
- %cr: 提交日期, 相对格式(1 day ago)
- %d: ref名称
- %s: commit信息标题
- %b: commit信息内容
- %n: 换行

git cherry-pick

> 合并分支的一条或几条提交记录到当前分支末梢

```
git cherry-pick 170a305
```

合并提交ID 170a305 到当前分支末梢

git reset

> 将当前的分支重设（reset）到指定的 <commit> 或者 HEAD

```
git reset --mixed <commit>
```

--mixed 是不带参数时的默认参数，它退回到某个版本，保留文件内容，回退提交历史

```
git reset --soft <commit>
```

暂存区和工作区中的内容不作任何改变，仅仅把 HEAD 指向 <commit>

```
git reset --hard <commit>
```

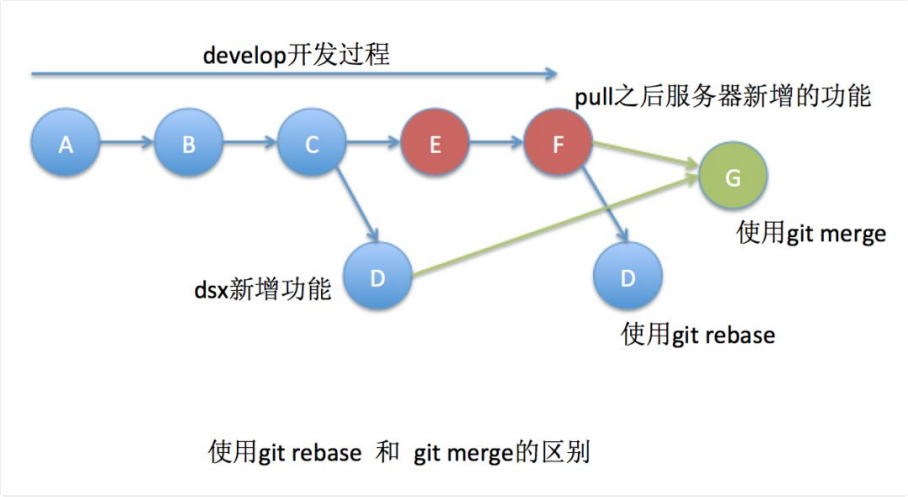
自从 <commit> 以来在工作区中的任何改变都被丢弃，并把 HEAD 指向 <commit>

git rebase

> 重新定义分支的版本库状态

```
git rebase branch_name
```

合并分支，这跟 merge 很像，但还是有本质区别，看下图：



使用git rebase 和 git merge的区别

合并过程中可能需要先解决冲突，然后执行 git rebase --continue

```
git rebase -i HEAD~~
```

打开文本编辑器，将看到从 HEAD 到 HEAD~~ 的提交如下

pick9a54fd4添加commit的说明
pick0d4a808添加pull的说明
#Rebase326fc9f..0d4a808ontod286baa
#
#Commands:
p,pick=usecommit
r,reword=usecommit,buteditthecommitmessage
e,edit=usecommit,butstopforamending
s,squash=usecommit,butmeldintopreviouscommit
f,fixup=like"squash",butdiscardthiscommit'slogmessage
x,exec=runcommand(therestoftheline)usingshell
#

将第一行的 pick 改成 Commands 中所列出来的命令，然后保存并退出，所对应的修改将会生效。

如果移动提交记录的顺序，将改变历史记录中的排序。

git revert

> 撤销某次操作，此次操作之前和之后的 commit 和 history 都会保留，并且把这次撤销作为一次最新的提交

```
git revert HEAD
```

撤销前一次提交操作

```
git revert HEAD --no-edit
```

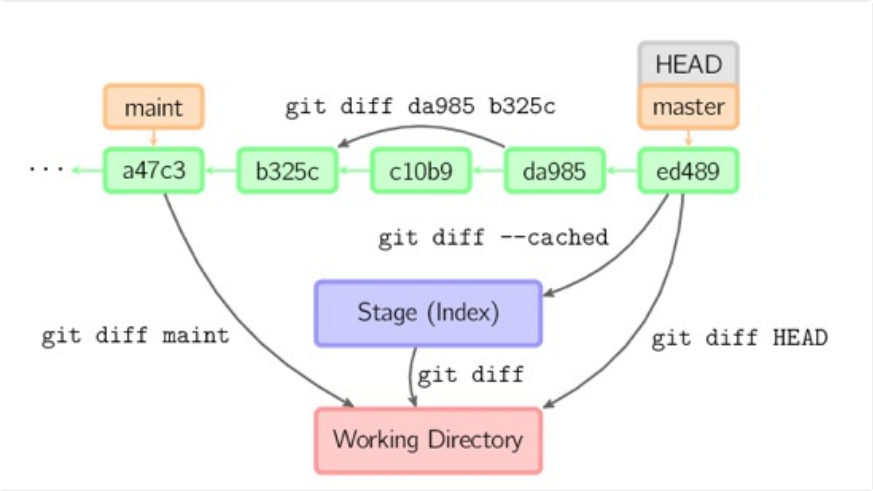
撤销前一次提交操作，并以默认的 Revert "xxx" 为提交原因

```
git revert -n HEAD
```

需要撤销多次操作的时候加 -n 参数，这样不会每次撤销操作都提交，而是等所有撤销都完成后一起提交，微信搜索 Java后端 关注，获取更多推送。

git diff

> 查看工作区、暂存区、本地版本库之间的文件差异，用一张图来解释



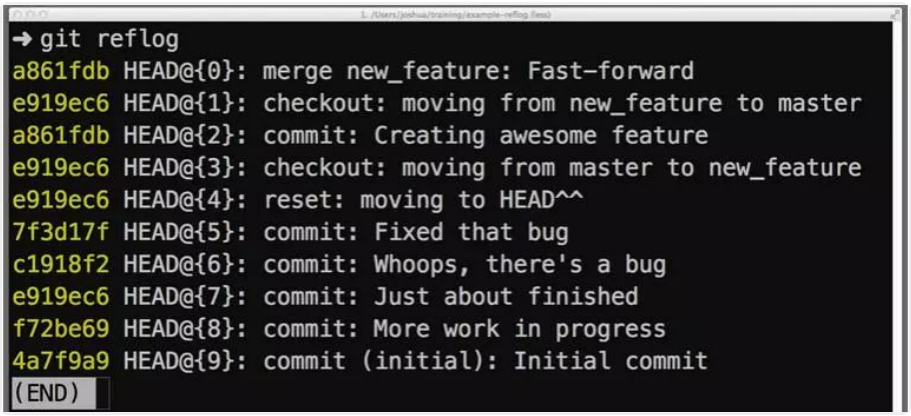

```
git diff --stat
```

通过 --stat 参数可以查看变更统计数据

```
 test.md | 1 -
 1 file changed, 1 deletion(-)
```

git reflog

reflog 可以查看所有分支的所有操作记录（包括commit和reset的操作、已经被删除的commit记录，跟git log 的区别在于它不能查看已经删除了的commit记录



远程版本库连接

如果在GitHub项目初始化之前，文件已经存在于本地目录中，那可以在本地初始化本地版本库，再将本地版本库跟远程版本库连接起来

git init

> 在本地目录内部会生成.git文件夹

git remote

```
git remote -v
```

不带参数，列出已经存在的远程分支，加上 -v 列出详细信息，在每一个名字后面列出其远程url

```
git remote add origin https://github.com/gafish/gafish.github.com.git
```

添加一个新的远程仓库，指定一个名字，以便引用后面带的URL

git fetch

> 将远程版本库的更新取回到本地版本库

```
git fetch origin daily/0.0.
1
```

默认情况下，git fetch 取回所有分支的更新。如果只想取回特定分支的更新，可以指定分支名。

问题排查

git blame

> 查看文件每行代码块的历史信息

```
git blame -L 1,10 demo.html
```

截取 demo.html 文件1-10行历史信息

git bisect

> 二分查找历史记录，排查BUG

```
git bisect start
```

开始二分查找

```
git bisect bad
```

标记当前二分提交ID为有问题的点

```
git bisect good
```

标记当前二分提交ID为没问题的点

```
git bisect reset
```

查到有问题的提交ID后回到原分支

更多操作

git submodule

> 通过 Git 子模块可以跟踪外部版本库，它允许在某一版本库中再存储另一版本库，并且能够保持2个版本库完全独立

```
git submodule add https://github.com/gafish/demo.git demo
```

将 demo 仓库添加为子模块

```
git submodule update demo
```

更新子模块 demo

git gc

> 运行Git的垃圾回收功能，清理冗余的历史快照

git archive

> 将加了tag的某个版本打包提取

```
git archive -v --format=zip v0.1 > v0.1.zip
```

--format 表示打包的格式，如 zip，-v 表示对应的tag名，后面跟的是tag名，如 v0.1。

总结

本文只是对 Git 的所有功能中的部分实用功能做了一次探秘，Git非常强大，还有很多功能有待我们去发现，限于本文篇幅，咱就此打住吧，预知更多好用功能，请善用谷歌。

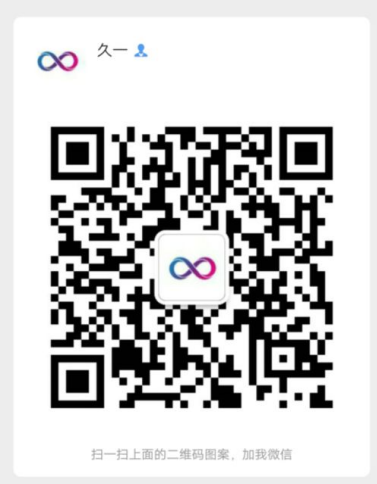
参考资料

- http://www.liaoxuefeng.com/wiki/001373951630592960dd18361248578c67b8067c8c017b000/00137453741517 82eb658c5a5ca454eaa451661275886c6000
- https://backlogtool.com/git-guide/tw/stepup/stepup1_1.html
- http://hubingforever.blog.163.com/blog/static/171040579201231421851 2/
- http://hubingforever.blog.163.com/blog/static/171040579201231110373 3821/
- http://www.cnblogs.com/hutaoer/archive/2013/05/07/git_checkout.html
- https://ruby-china.org/topics/939
- http://blog.csdn.net/hudashi/article/details/7664631/
- http://backlogtool.com/git-guide/cn/stepup/stepup7_6.html

-END-

码即可进入无广告交流群。

↓扫描二维码进群↓



推荐阅读

1. 感受 Lambda 之美，推荐收藏，需要时查阅
2. 如何优雅的导出 Excel
3. 文艺互联网公司 vs 二逼互联网公司
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

阅读原文

Git 代码防丢指南

joymufeng Java后端 2019-09-10

作者：joymufeng

我们在日常使用Git的过程中经常会发生一些意外情况，如果处理不当，则可能会出现代码丢失的假象。本文将针对IDEA&Git日常开发中的一些场景，为你层层拨开迷雾，解析常见的错误及其发生原因，让你从此不再惧怕代码冲突或丢失问题。

> 为简化问题，本文假设所有团队成员均在同一分支上开发。
> 文中更新操作是指在IDEA中单击菜单VCS-Update Project...。

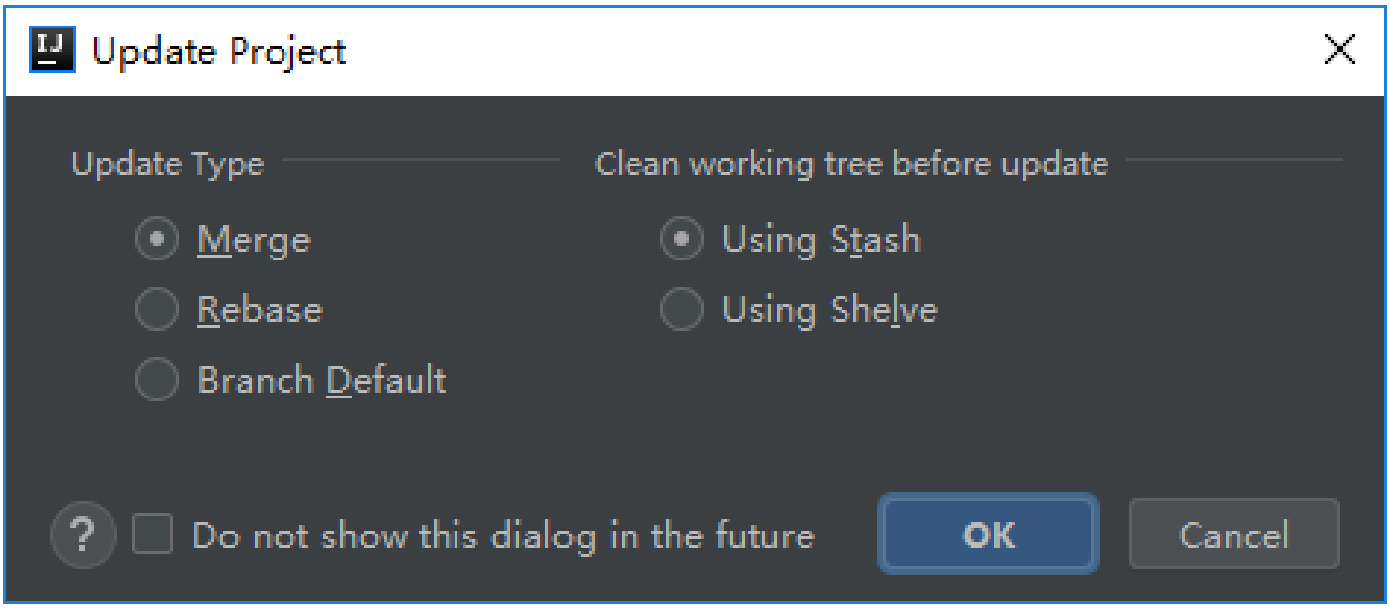
1、常见工作流程

通常当你早上到公司打开电脑，首先执行更新操作(单击IDEA菜单VCS-Update Project...)，然后开始愉快地编码。 编码完成后通常要执行以下几个操作：

- 更新操作
- 创建本次提交
- 推送远程分支

1.1 更新操作

为了保证Git拥有一个简洁的提交历史，在提交之前需要先执行更新操作，即在IDEA中依次单击菜单VCS-Update Project...，或者按下Ctrl+T，弹出如下窗口：



窗口左侧选择更新类型(Update Type)：

- Merge：更新时执行合并操作。等价于执行git fetch && git merge或者git pull --no-rebase。
- Rebase：更新时执行rebase操作。等价于执行git fetch && git rebase或者git pull --rebase。

- Branch Default：在.git/config文件中指定不同分支的更新类型。

窗口右侧选择在更新前工作目录(Working Directory)的清理方式：

- Using Stash：使用git stash储藏本地修改。
- Using Shelve：使用IDEA内置的Shelve功能储藏本地修改。

通常选择Merge和Using Stash即可，单击OK后，IDEA执行步骤如下：

- 第1步：使用git stash储藏本地修改
- 第2步：执行git fetch && git merge拉取远程分支并合并
- 第3步：执行git stash pop恢复储藏

> 有些同学可能更习惯先创建本地提交，然后在执行更新操作，这样会导致Git自动生成一个合并提交，导致提交历史不够简洁。

1.2 创建本次提交

更新完成后，在IDEA中单击菜单VCS-Commit...创建本次提交。

1.3 推送远程分支

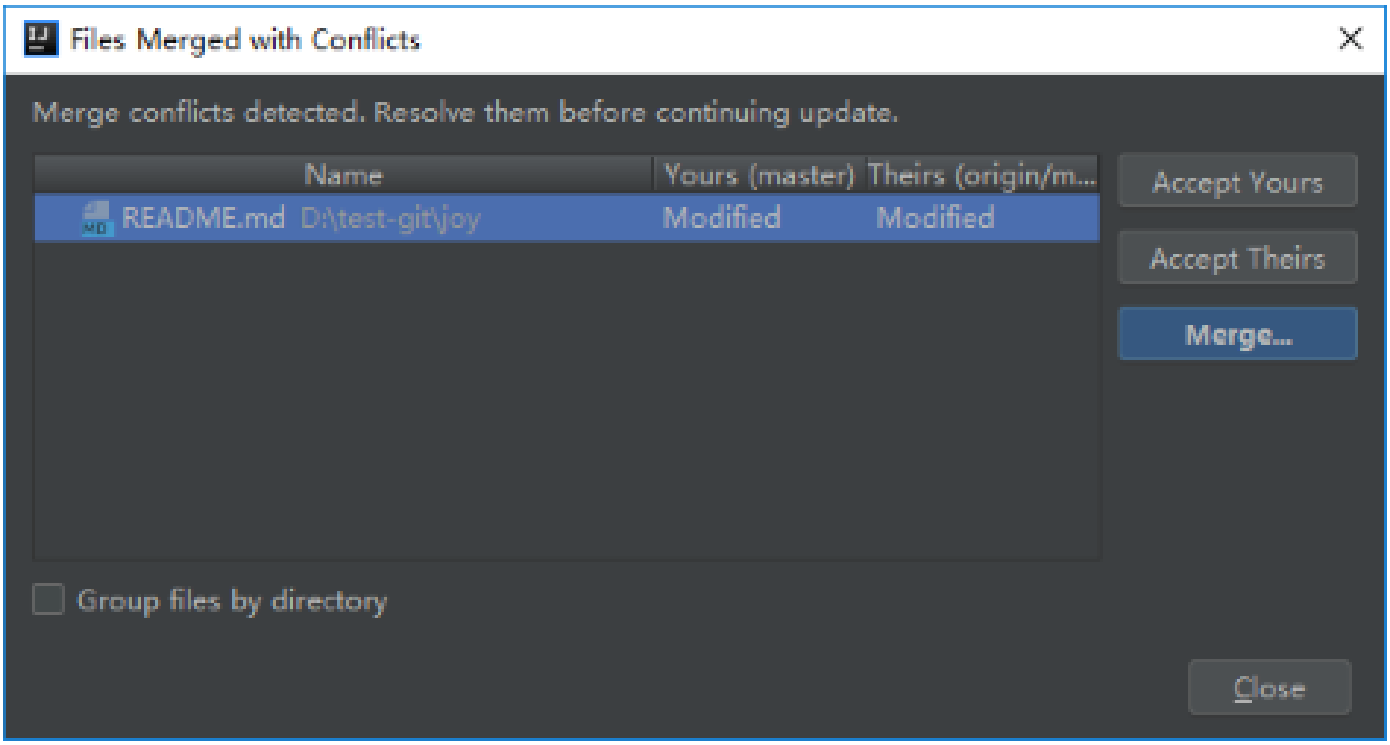
然后单击VCS-Git-Push...推送至远程分支。

2、常见问题分析

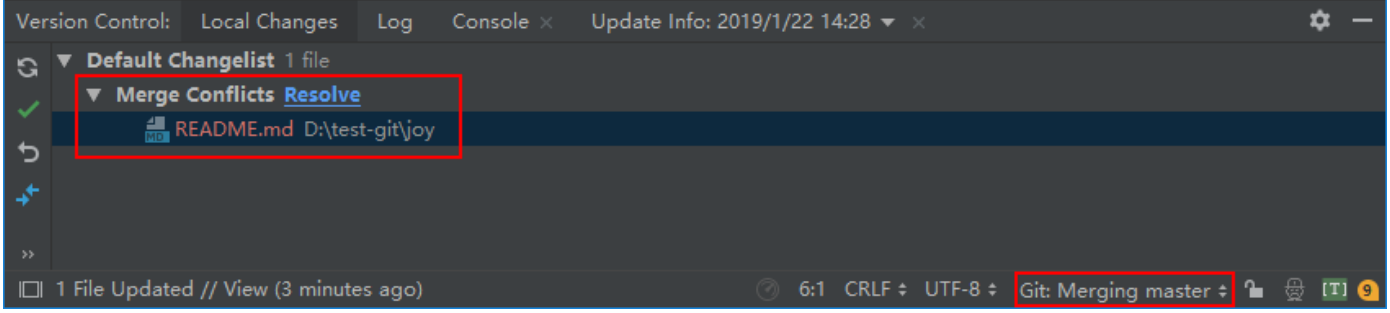
在上面的3步执行步骤中，第2步和第3步发生意外的风险最高，最常见的两种意外情况是冲突和文件占用，下面我们分别讨论。

2.1 合并远程分支冲突

如果在执行更新操作之前，你的本地分支已经创建过提交，并且尚未推送至远程分支，则在第2步执行git merge时很可能会发生冲突。



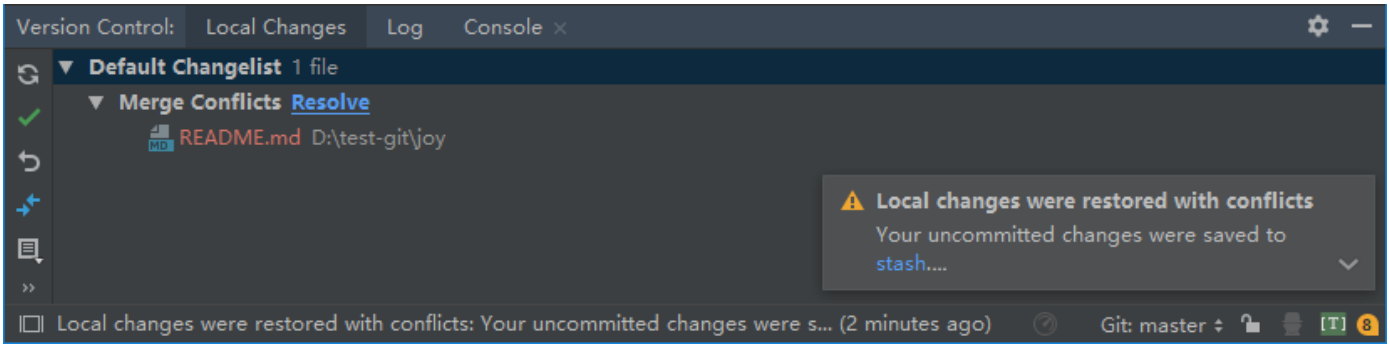
此时关闭上面的冲突窗口，Version Control工具窗口显示内容如下：



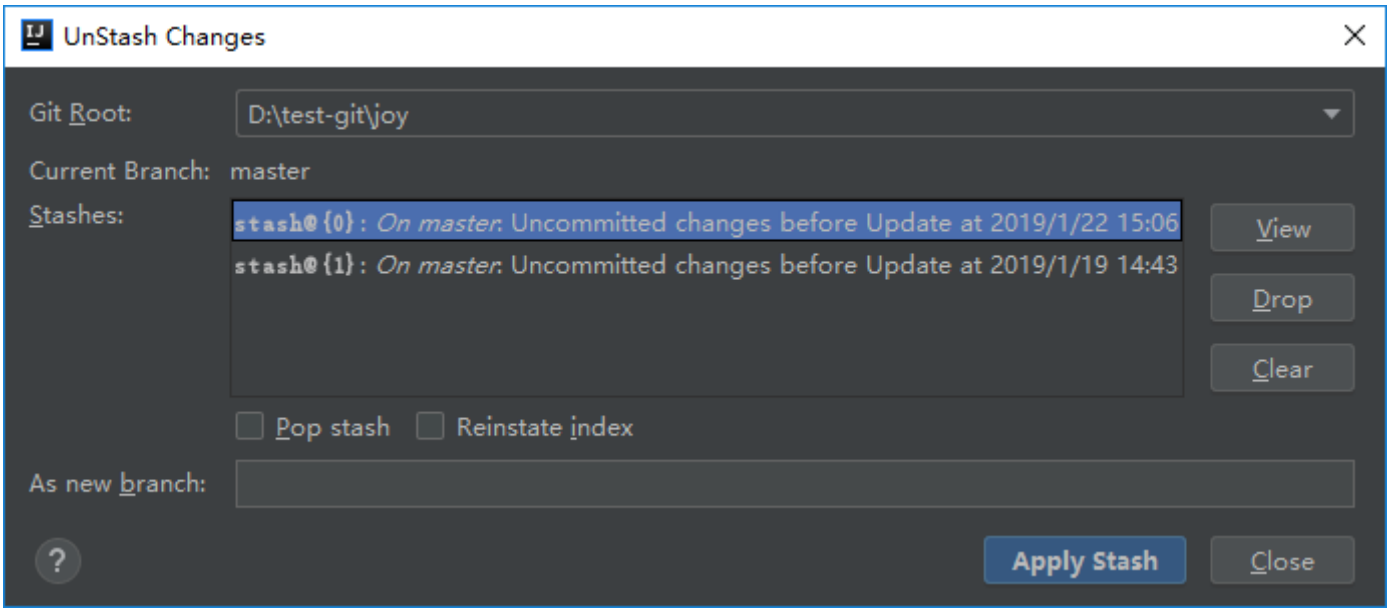
窗口右下角原本显示分支名称的位置变成了Merging master,表示本地分支master目前处于正在合并状态。 单击左侧红框内Resolve按钮可以再次调出处理冲突窗口。基于IDEA的图形界面手动解决冲突后，IDEA会自动将该文件加入暂存区(加入暂存区即表示冲突解决完成)，最后执行一次提交便可以完成冲突处理。

2.2 恢复储藏冲突

在更新操作的第3步执行git stash pop恢复储藏时，储藏内容可能与刚更新的内容发生冲突。

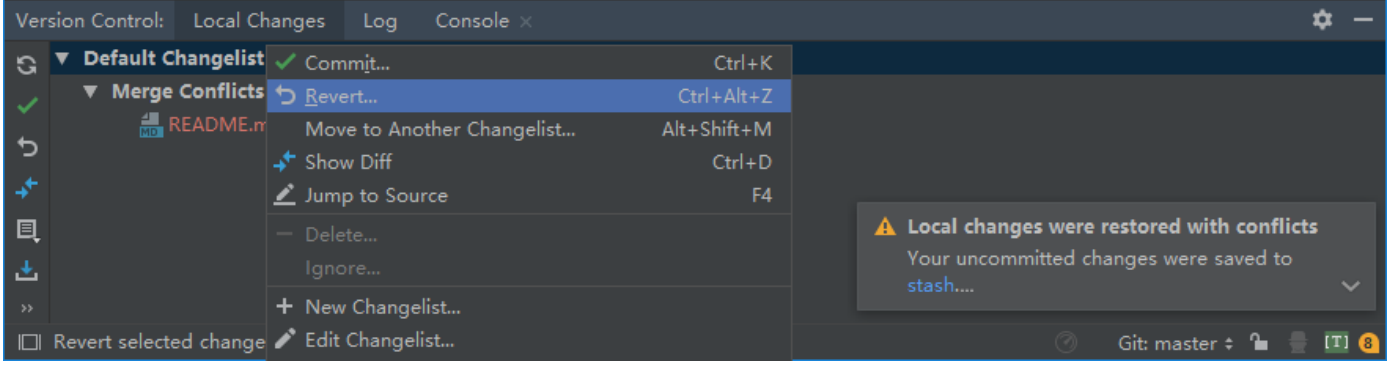


恢复储藏时发生的冲突跟上面的合并冲突稍微有些区别，首先是右下角的分支名称没有Merging字样，另外会在右下角额外弹出一个小窗提示恢复储藏失败，并且告诉你不用担心，所有的修改都在stash列表中，并没有丢失。查看stash列表的方式为单击菜单VCS-Git-UnStash Changes...:



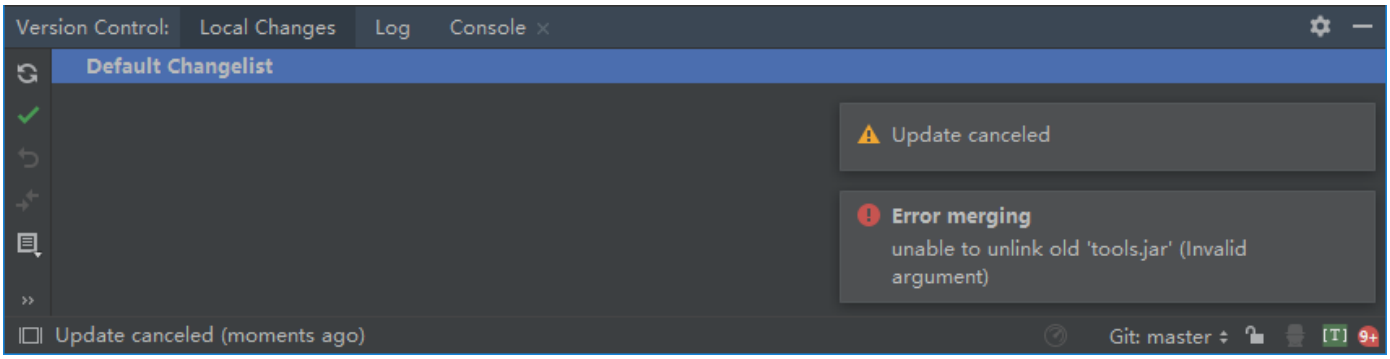
选中列表最上面的条目，然后单击Apply Stash，之前的修改就会重新回到工作目录。

我们继续回到冲突问题，手动解决冲突后执行一次提交就可以了。如果在解决冲突过程中发生了误操作,可以右击Default Changelist-Revert...清空当前工作目录内容，重新执行一次Apply Stash，然后重复解决冲突过程。



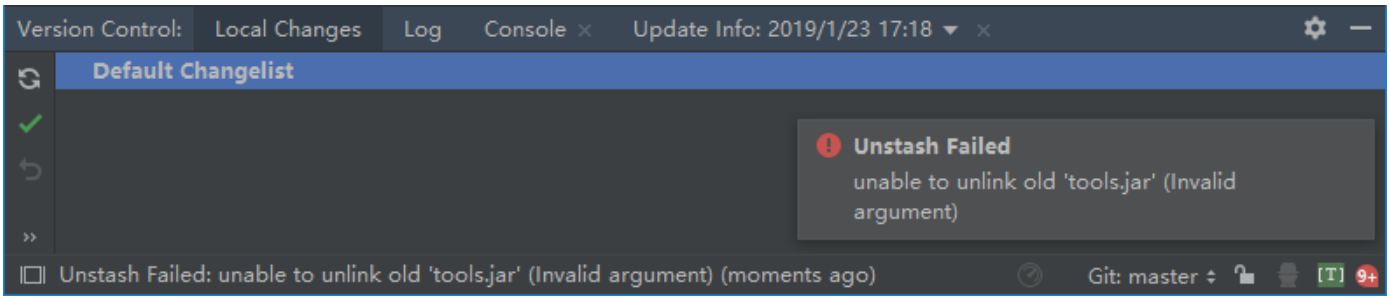
2.3 文件占用错误

在执行第2步git merge时，可能会因为文件被占用导致执行失败。 例如项目可能引入了一些jar文件，这些jar文件在本地已经被JVM动态加载了，如果有其它人更新了该jar文件并且推送到了远程分支，当你更新时便会遇到上述问题。关注微信公众号「web_resourc」,回复 Java 领取2019最新资源。



对于这种错误的解决方法很简单，首先解除文件的占用状态，例如终止本地JVM进程，然后再次点击VCS-Update。

在执行第3步git stash pop时，也会因为文件被占用导致执行失败。 例如你更新了某个jar文件，当恢复储藏时可能因为该jar文件被占用导致恢复失败。



对于这种错误，你需要首先解除文件占用状态，然后手动执行unstash操作。

3、先提交还是先更新？是个问题！

3.1 先提交后更新导致的问题

3.1.1 发生冲突时难以处理

如果先提交，但是在更新时却发生了冲突，这就意味着你刚刚创建的提交其实是有问题的，通常是团队沟通或是分工出了问题，但是不管这么说，别人已经抢先一步push了，你的提交便会被拒之门外。即便是手动解决了冲突，这个提交保留在历史中也会成为隐患，如果有其他人reset回这个提交继续工作，则在合并其它分支内容时发生冲突的概率会大大增加，所以最好处理方式是先撤销这个提交(reset --soft HEAD~)，然后更新并解决冲突，最后创建一个新的提交。

3.1.2 错误的处理冲突方式

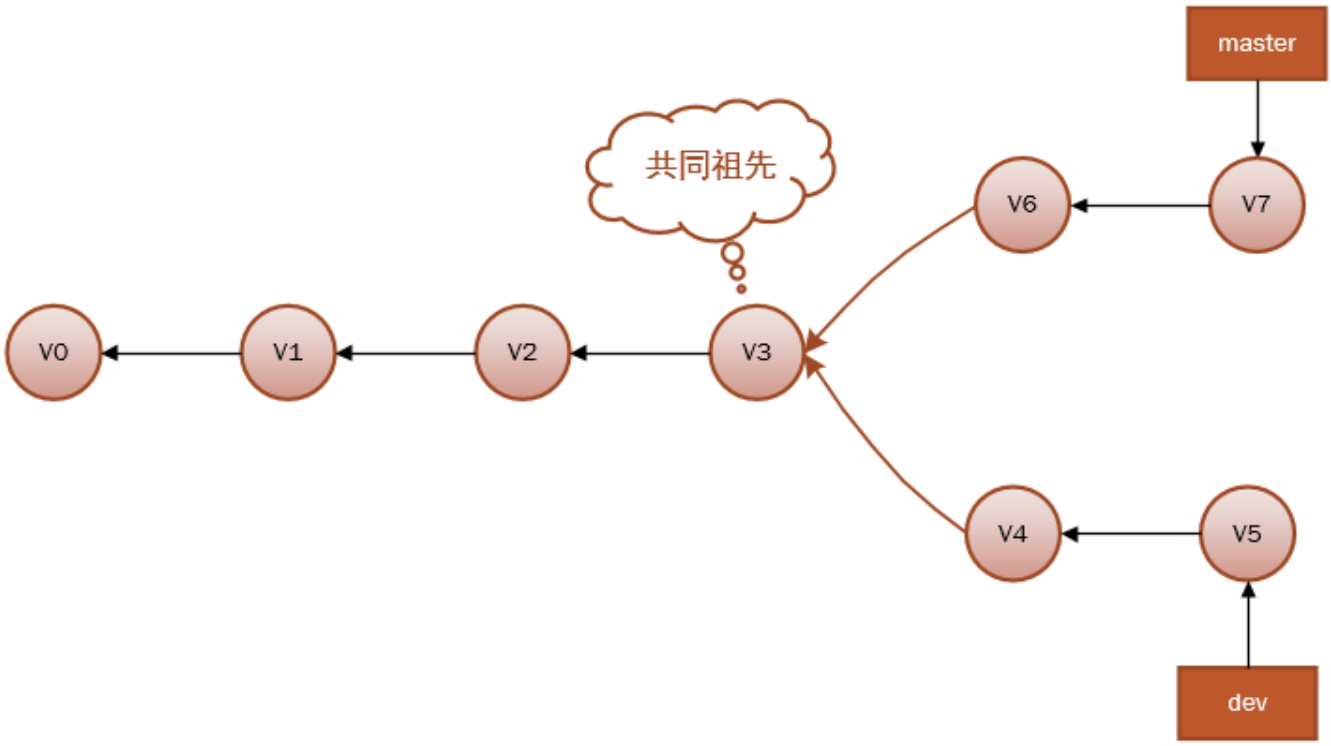
在发生冲突后，有些同学可能会想到下面的处理方式：

- 清空当前工作空间
- 调整冲突部分的代码
- 然后再次执行更新操作

上面的处理方式很明显是不可行的，因为你调整的代码首选会被IDEA储藏（stash）起来，然后在更新的第2步中仍然会发生冲突，并且发生冲突时，你的修改尚未恢复储藏(unstash)，导致看起来你调整的代码不见了，让人摸不着头脑。

3.1.3 Rebase会改写提交历史

如果在IDEA的更新窗口选择更新类型为Rebase，则等价于手动执行git fetch && git rebase或者git pull --rebase命令。这样的好处是不会生成一个自动合并提交，保持简洁的提交历史。但是需要注意的是，Rebase之后，你的本地提交会被改写，虽然提交信息一样，但是commit hash已经改变了，如下图所示：



在执行完如下的Rebase命令后，

```
$ git checkout dev
$ git rebase master
```

执行结果为：

请注意，结果中的v4和v5提交已经被改写了。

3.2 推荐先更新后提交

如果你事先知道会发生冲突，相信你一定不会选择先提交代码，但是冲突是不可避免的，这就要求我们平时养成良好的开发习惯。与其解决提交后的冲突，不如尽早地解决冲突然后提交，这样不仅可以减少一个无意义的自动合并提交，而且可以在冲突发生时简化处理过程。

3.3 养成良好习惯

为了尽量避免冲突发生，建议养成如下开发习惯：

- 编码前先更新
- 提交前先更新
- 提交前检查是否有编译错误
- 提交粒度尽可能小，描述尽可能准确
- 修改了公共文件，尽早通知其他成员更新
- 最后一条，也是最重要的，团队分工要明确

> **原文链接：**
> https://my.oschina.net/joymufeng/blog/3005221?from=singlemessage

---

推荐阅读

1. 史上最烂的项目：苦撑12年，600 多万行代码 ...
2. 请给 Spring Boot 多一些内存
3. 如何从零搭建百亿流量系统？
4. 惊了！原来 Web 发展历史是这样的

Web项目聚集地

微信扫描二维码，关注我的公众号

喜欢文章，点个在看

阅读原文

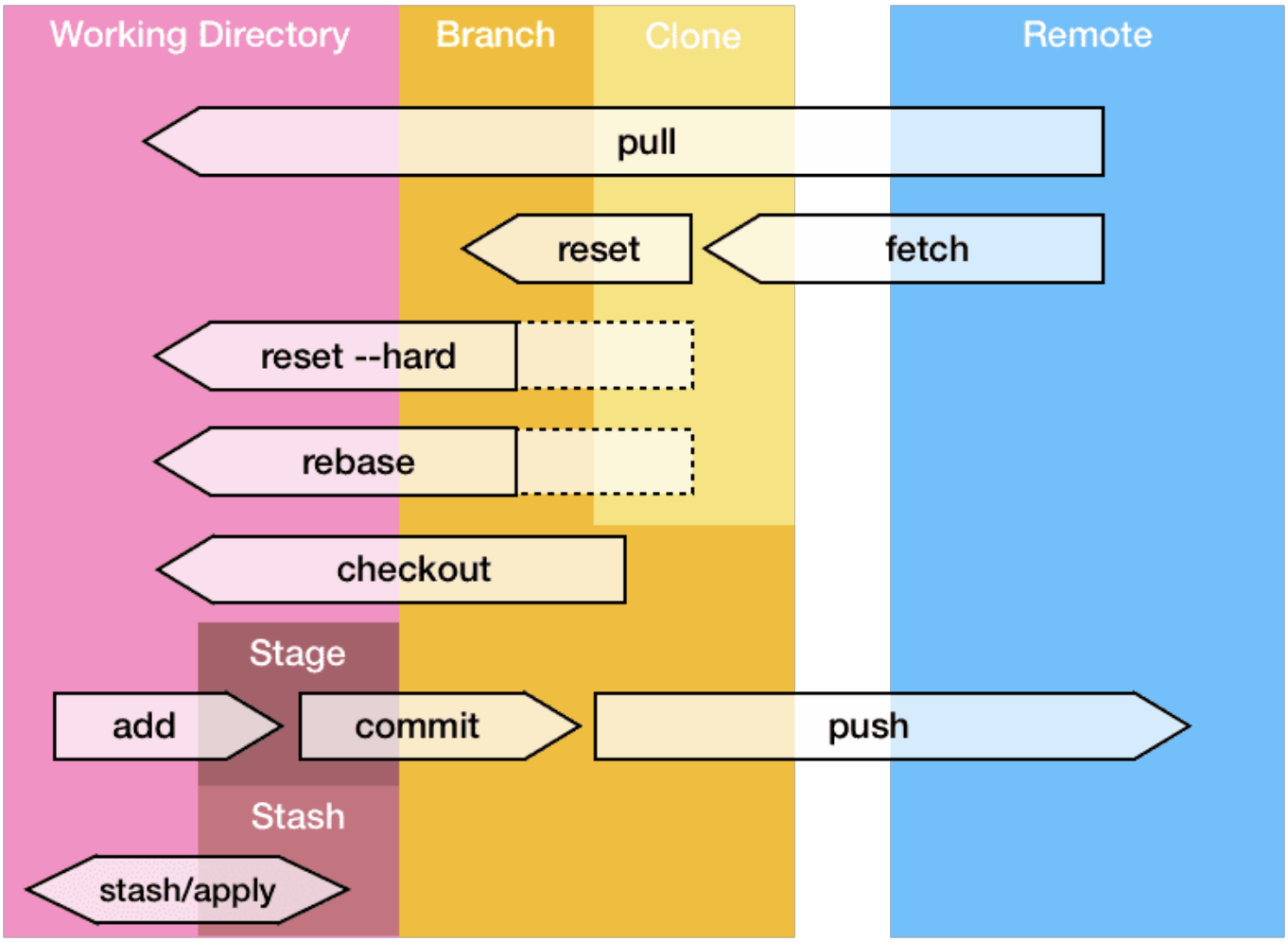
Git 高级用法小抄

Icarus Java后端 2月6日

作者 | Maxence Poutord

来源 | New Frontend 网站

如果你觉得 git 很迷惑人，那么这份小抄正是为你准备的！请注意我有意跳过了git commit、git pull/push 之类的基本命令，这份小抄的主题是 git 的一些「高级」用法。



导航 —— 跳到之前的分支

```
git checkout -
```

查看历史

```
# 每个提交在一行内显示
git log --oneline

# 在所有提交日志中搜索包含「homepage」的提交
git log --all --grep='homepage'

# 获取某人的提交日志
git log --author="Maxence"
```

哎呀：之前重置了一个不想保留的提交，但是现在又想要回滚？

```
# 获取所有操作历史
git reflog

# 重置到相应提交
git reset HEAD@{4}
# ……或者……
git reset --hard <提交的哈希值>
```

哎哟：我把本地仓库搞得一团糟，应该怎么清理？

```
git fetch origin
git checkout master
git reset --hard origin/master
```

查看我的分支和 master 的不同

```
git diff master..my-branch
```

定制提交

```
# 编辑上次提交
git commit --amend -m "更好的提交日志"

# 在上次提交中附加一些内容，保持提交日志不变git add . && git commit --amend --no-edit

# 空提交 —— 可以用来重新触发 CI 构建
git commit --allow-empty -m "chore: re-trigger build"
```

squash 提交

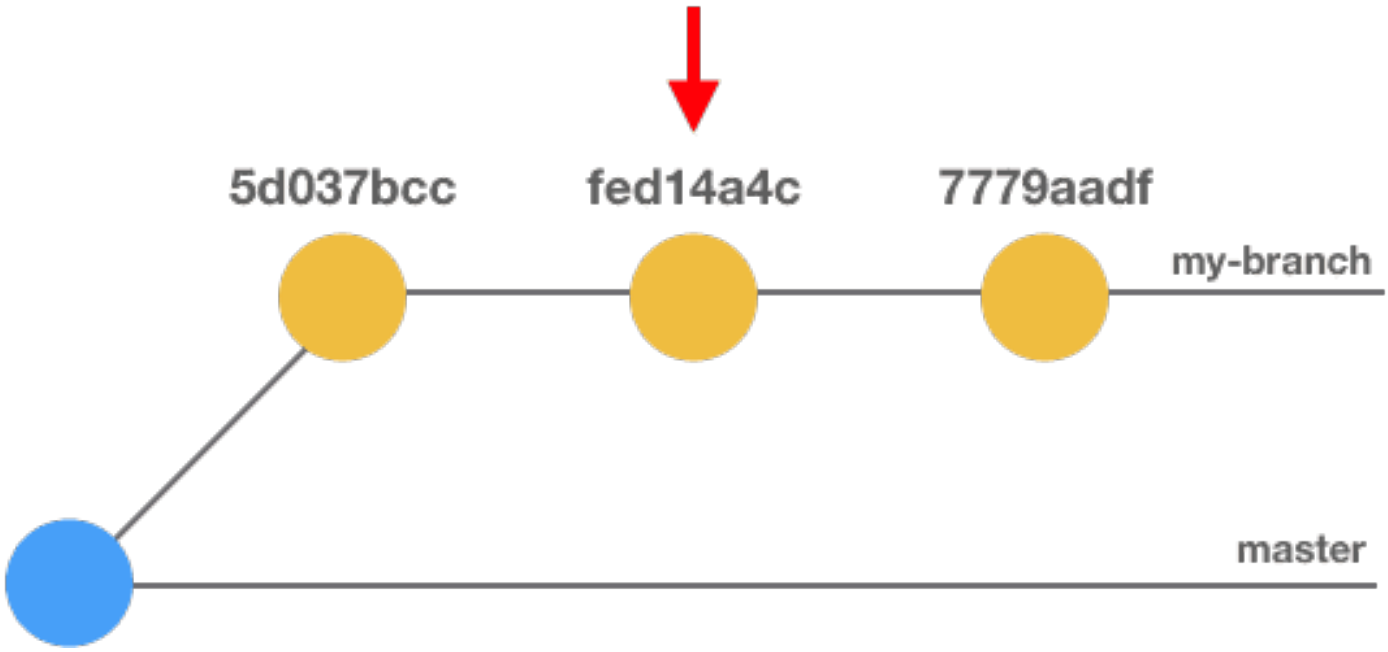
比方说我想要 rebase 最近 3 个提交：

- git rebase -i HEAD~3
- 保留第一行的 pick，剩余提交替换为 squash 或 s
- 清理提交日志并保存（vi 编辑器中键入 :wq 即可保存）

```
pick 64d26a1 feat: add index.js
s 45f0259 fix: update index.js
s 8b15b0a fix: typo in index.js
```

修正

比方说想在提交 fed14a4c 加上一些内容。



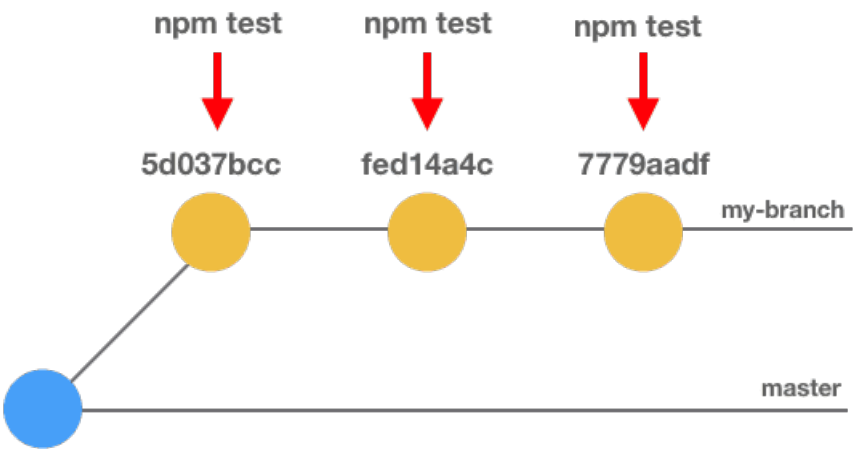
git 提交分支

```
git add .
git commit --fixup HEAD~1
# 或者也可以用提交的哈希值（fed14a4c）替换 HEAD~1

git rebase -i HEAD~3 --autosquash
# 保存并退出文件（VI 中输入 `:wq`）
```

rebase 的时候在每个提交上执行命令

如果特性很多，一个分支里可能有多个提交。如果测试失败了，你希望能找到导致测试失败的提交。这时候你可以使用 rebase --exec 命令在每个提交上执行命令。

```
# 在最近 3 个提交上运行 `npm test` 命令
git rebase HEAD~3 --exec "npm test"
```



暂存

暂存不止是 git stash 和 git stash pop ;)

```
# 保存所有正在追踪的文件
git stash save "日志信息"

# 列出所有的暂存项
git stash list

# 获取并删除暂存项
git stash apply stash@{1}
git stash drop stash@{1}
# ……或使用一条命令……
git stash pop stash@{1}
```

清理

```
# 移除远程仓库上不存在的分支
git fetch -p

# 移除所有包含 `greenkeeper` 的分支
git fetch -p && git branch --remote | fgrep greenkeeper | sed 's/^.\{9\}//' | xargs git push origin --delete
```

GitHub = Git + Hub

我把 Hub 当成 git 的一个封装来用。你如果也想这么做，可以设置一个别名：alias git='hub'

```
# 打开浏览器访问仓库 url（仅限 GitHub 仓库）git browse
```

额外福利：我最喜爱的 git 别名

```
alias g='git'
alias glog='git log --oneline --decorate --graph'
alias gst='git status'
alias gp='git push'
alias ga='git add'alias gc='git commit -v'

# 🤘
alias yolo='git push --force'

# 每周站会汇报工作时用
git-standup() {
  AUTHOR=${AUTHOR:="`git config user.name`"}

  since=yesterday
  if [[ $(date +%u) == 1 ]] ; then
    since="2 days ago"
  fi

  git log --all --since "$since" --oneline --author="$AUTHOR"
}
```

你最喜欢的 git 命令是哪个呢？

推荐阅读

1. 聊聊在阿里远程办公那点事儿
2. 你真的了解 volatile 吗?
3. 程序员才能看懂的动图
4. 如何获取靠谱的新型冠状病毒疫情

阅读原文

Git 高级用法：收藏了，以后肯定能用上！

Java后端 Java后端 1周前

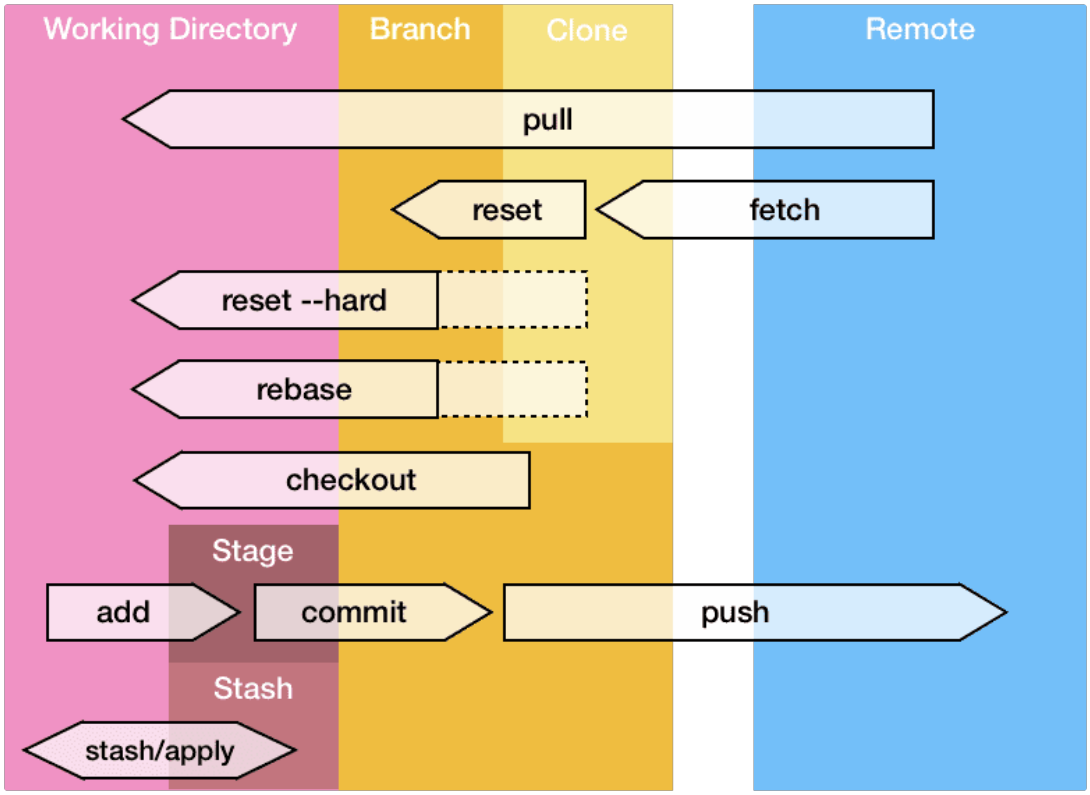
Java后端

作者:Maxence Poutord,

来源:New Frontend网站

如果你觉得 git 很迷惑人，那么这份小抄正是为你准备的！

请注意我有意跳过了 git commit、git pull/push 之类的基本命令，这份小抄的主题是 git 的一些「高级」用法。欢迎转发、点在看！



导航 —— 跳到之前的分支

```
git checkout -
```

查看历史

```
# 每个提交在一行内显示
git log --oneline

# 在所有提交日志中搜索包含「homepage」的提交
git log --all --grep='homepage'

# 获取某人的提交日志
git log --author="Maxence"
```

哎呀：之前重置了一个不想保留的提交，但是现在又想要回滚？

```
# 获取所有操作历史
git reflog

# 重置到相应提交
git reset HEAD@{4}
# ……或者……
git reset --hard <提交的哈希值>
```

哎哟：我把本地仓库搞得一团糟，应该怎么清理？

```
git fetch origin
git checkout master
git reset --hard origin/master
```

查看我的分支和 master 的不同

```
git diff master..my-branch
```

定制提交

```
# 编辑上次提交
git commit --amend -m "更好的提交日志"

# 在上次提交中附加一些内容，保持提交日志不变git add . && git commit --amend --no-edit

# 空提交 —— 可以用来重新触发 CI 构建
git commit --allow-empty -m "chore: re-trigger build"
```

squash 提交

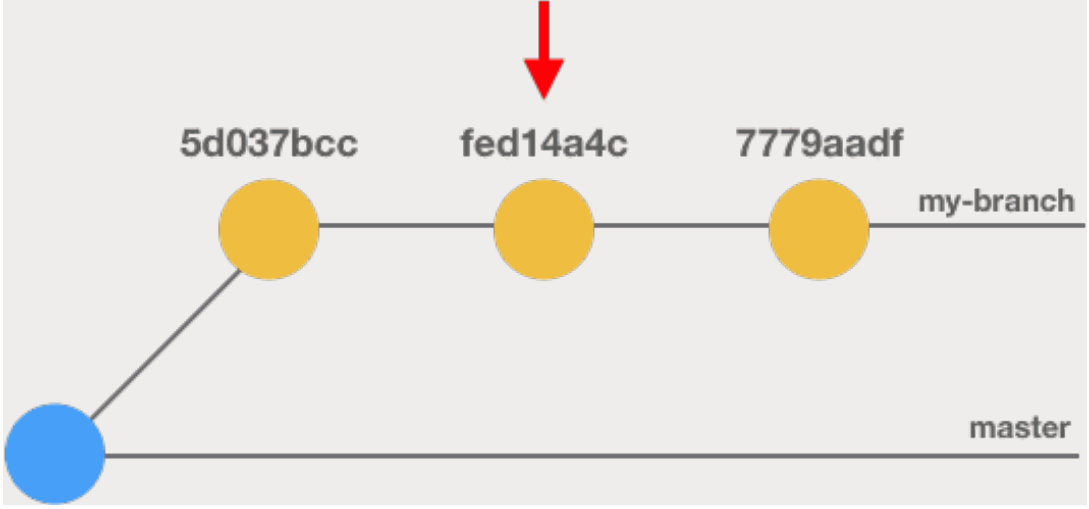
比方说我想要 rebase 最近 3 个提交：

- git rebase -i HEAD~3
- 保留第一行的 pick，剩余提交替换为 squash 或 s
- 清理提交日志并保存（vi 编辑器中键入 :wq 即可保存）

```
pick 64d26a1 feat: add index.js
s 45f0259 fix: update index.js
s 8b15b0a fix: typo in index.js
```

修正

比方说想在提交 fed14a4c 加上一些内容。



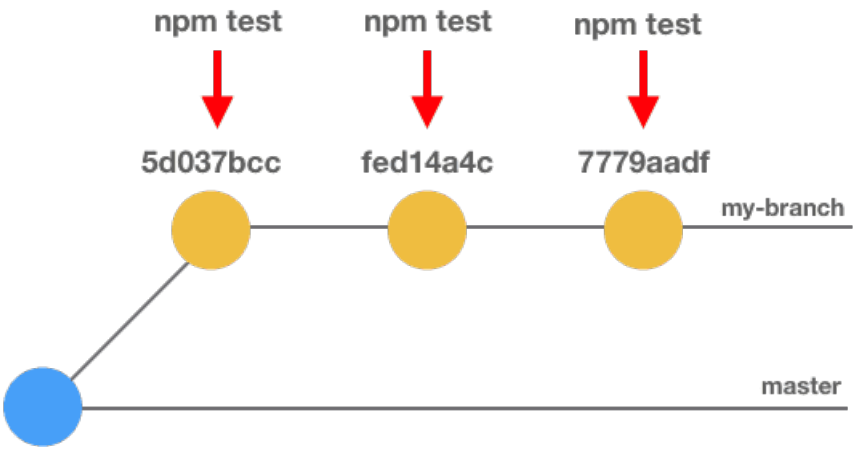
git 提交分支

```
git add .
git commit --fixup HEAD~1
# 或者也可以用提交的哈希值（fed14a4c）替换 HEAD~1

git rebase -i HEAD~3 --autosquash
# 保存并退出文件（VI 中输入 `:wq`）
```

rebase 的时候在每个提交上执行命令

如果特性很多，一个分支里可能有多个提交。如果测试失败了，你希望能找到导致测试失败的提交。这时候你可以使用 rebase --exec 命令在每个提交上执行命令。

```
# 在最近 3 个提交上运行 `npm test` 命令
git rebase HEAD~3 --exec "npm test"
```



暂存

暂存不止是 git stash 和 git stash pop ;)

```
# 保存所有正在追踪的文件
git stash save "日志信息"

# 列出所有的暂存项
git stash list

# 获取并删除暂存项
git stash apply stash@{1}
git stash drop stash@{1}
# ……或使用一条命令……
git stash pop stash@{1}
```

清理

```
# 移除远程仓库上不存在的分支
git fetch -p

# 移除所有包含 `greenkeeper` 的分支
git fetch -p && git branch --remote | fgrep greenkeeper | sed 's/^.\{9\}//' | xargs git push origin --delete
```

GitHub = Git + Hub

我把 Hub 当成 git 的一个封装来用。你如果也想这么做，可以设置一个别名：alias git='hub'

```
# 打开浏览器访问仓库 url（仅限 GitHub 仓库）git browse
```

额外福利：我最喜爱的 git 别名

```
alias g='git'
alias glog='git log --oneline --decorate --graph'
alias gst='git status'
alias gp='git push'
alias ga='git add'alias gc='git commit -v'

# 🤘
alias yolo='git push --force'

# 每周站会汇报工作时用
git-standup() {
  AUTHOR=${AUTHOR:="`git config user.name`"}

  since=yesterday
  if [[ $(date +%u) == 1 ]] ; then
    since="2 days ago"
  fi

  git log --all --since "$since" --oneline --author="$AUTHOR"
}
```

-END-

推荐阅读

1. Spring 中运用的 9 种设计模式吗？
2. Java 中的继承和多态(深入版)
3. 如何降低程序员的工资?
4. 编写 Spring MVC 的 14 个小技巧

喜欢文章，点个在看

IDEA 中使用 Git 图文教程

Java后端 2019-09-03

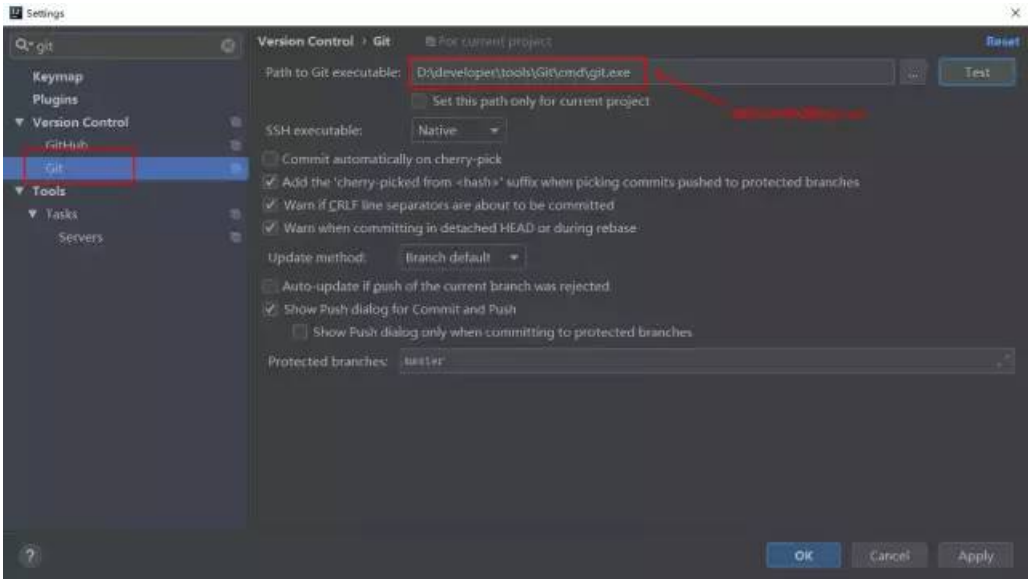
以下文章来源于macrozheng ，作者梦想de星空

摘要

大家在使用Git时，都会选择一种Git客户端，在IDEA中内置了这种客户端，可以让你不需要使用Git命令就可以方便地进行操作，本文将讲述IDEA中的一些常用Git操作。

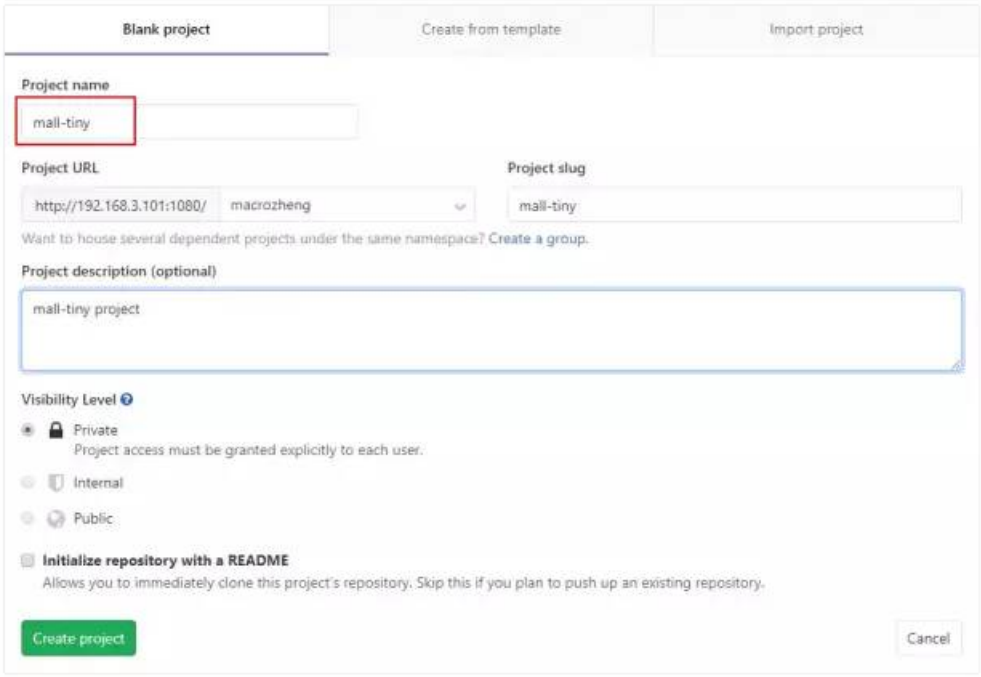
环境准备

- 使用前需要安装一个远程的Git仓库和本地的Git客户端，具体参考：10分钟搭建自己的Git仓库。
- 由于IDEA中的Git插件需要依赖本地Git客户端，所以需要进行如下配置：



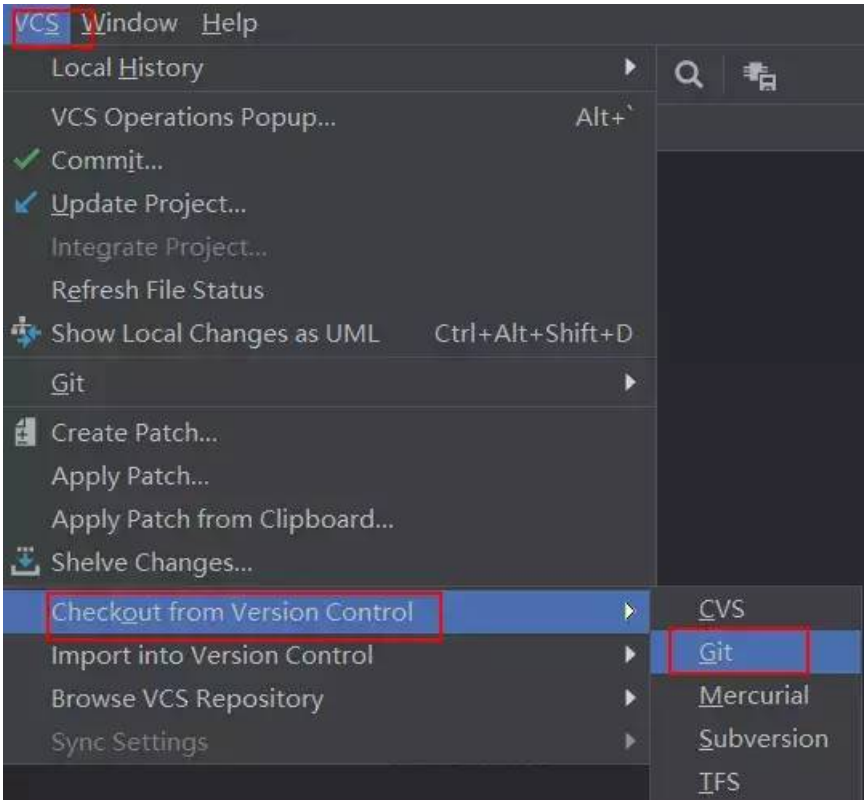
操作流程

在Gitlab中创建一个项目并添加README文件

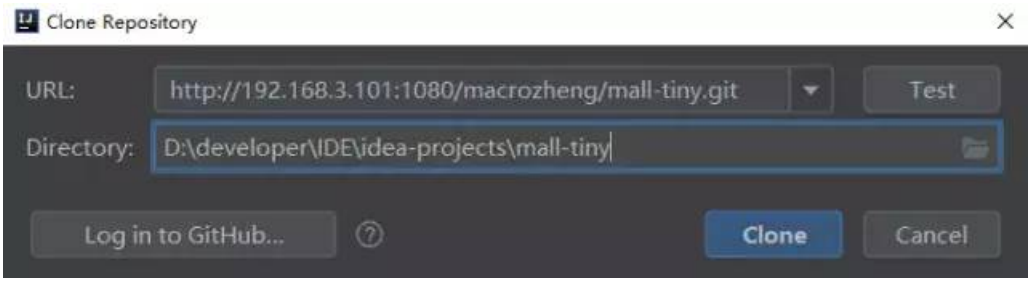


clone项目到本地

- 打开从Git检出项目的界面：



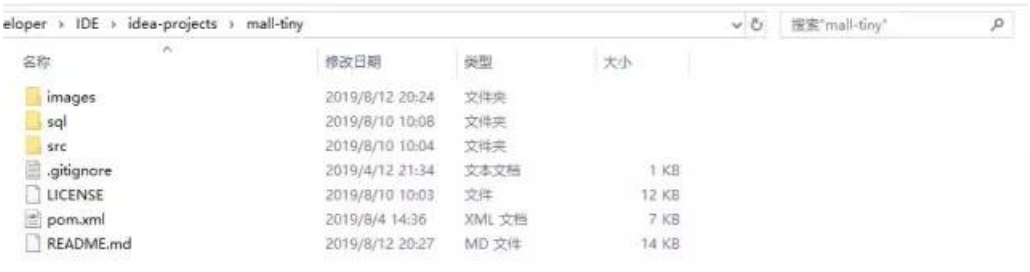
- 输入Git地址进行检出：



- 暂时不生成IDEA项目，因为项目还没初始化：

初始化项目并提交代码

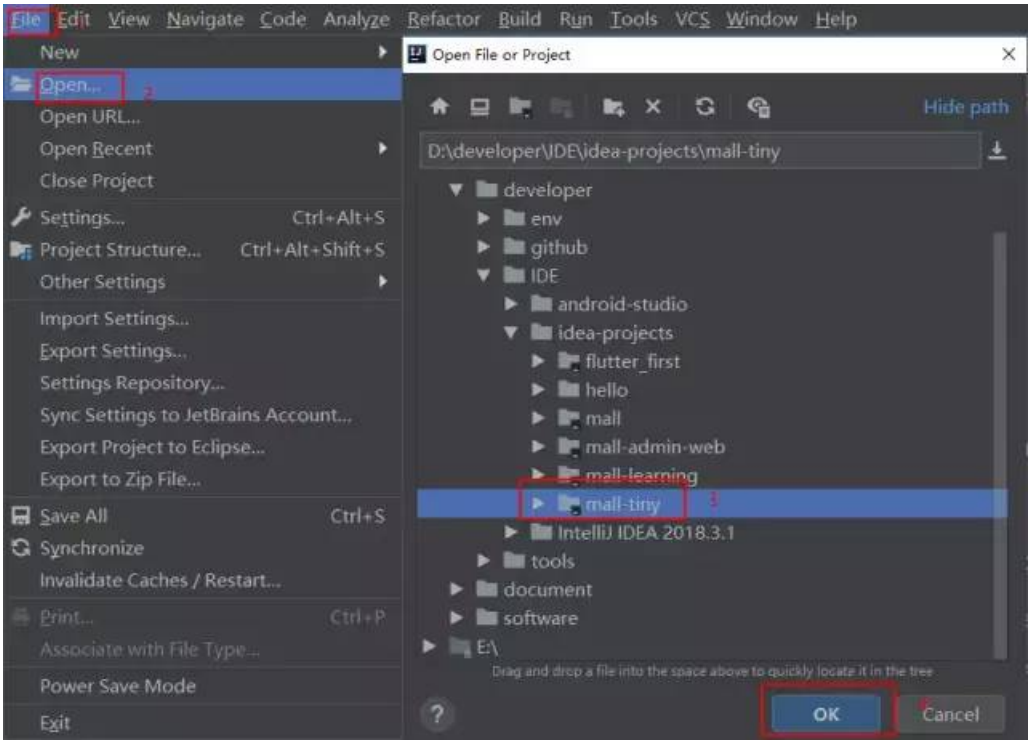
- 将mall-tiny的代码复制到该目录中：



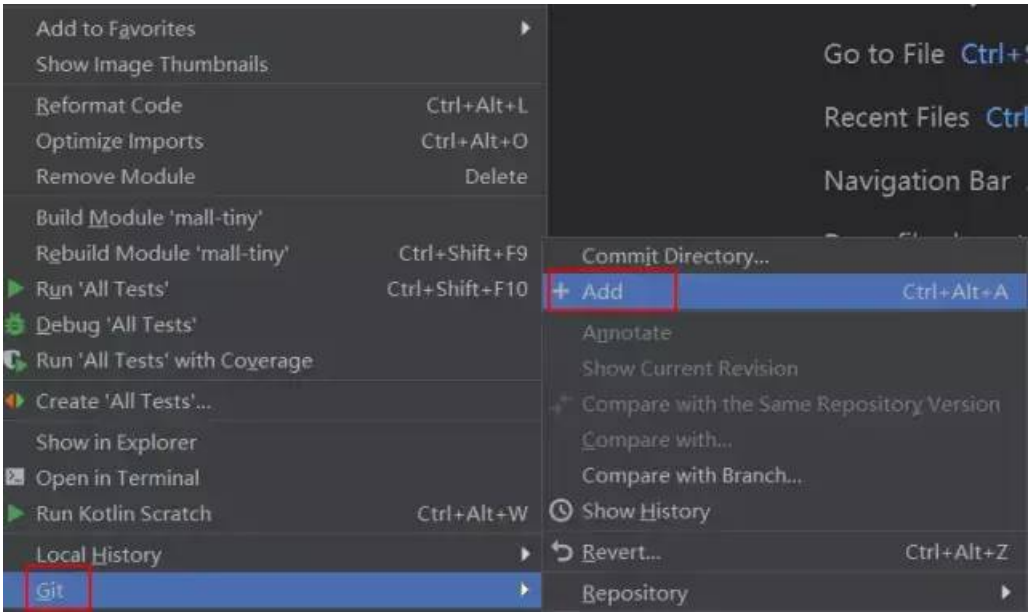
- 这里我们需要一个.gitignore文件来防止一些IDEA自动生成的代码被提交到Git仓库去：

```
# Maven
#
target/

# IDEA #
.idea/
*.iml

# Eclipse
#
.settings/
.classpath
.project
```

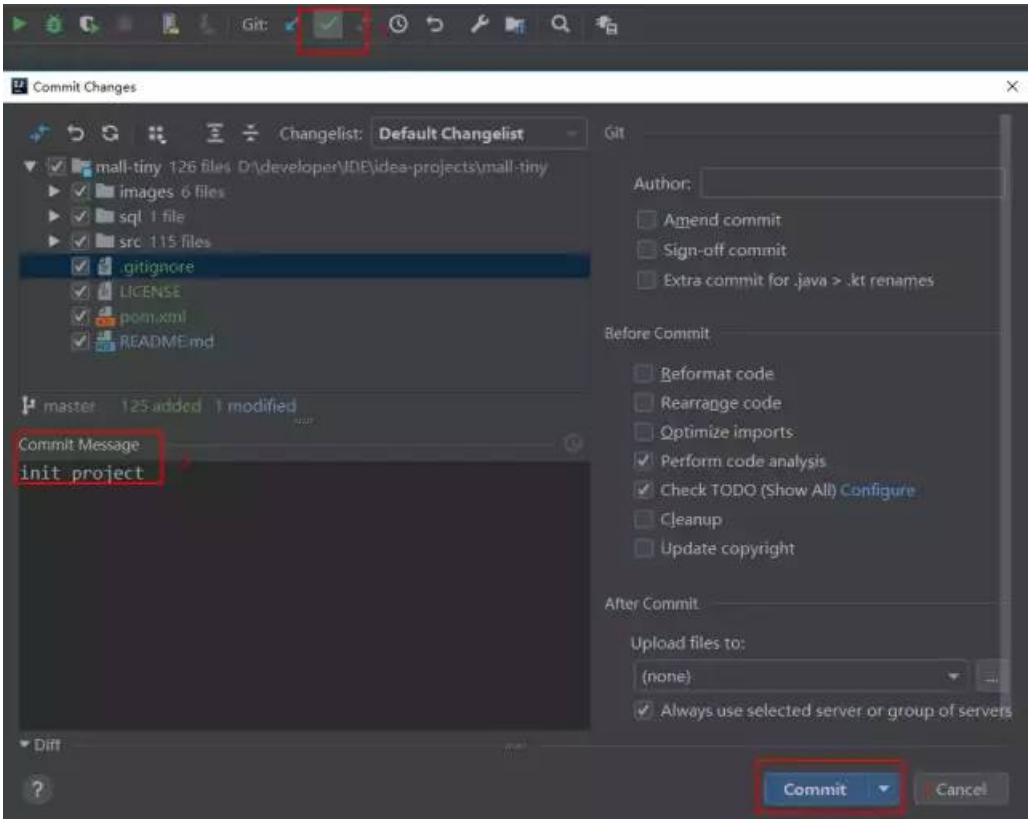
- 使用IDEA打开项目：



- 右键项目打开菜单，将所有文件添加到暂存区中：

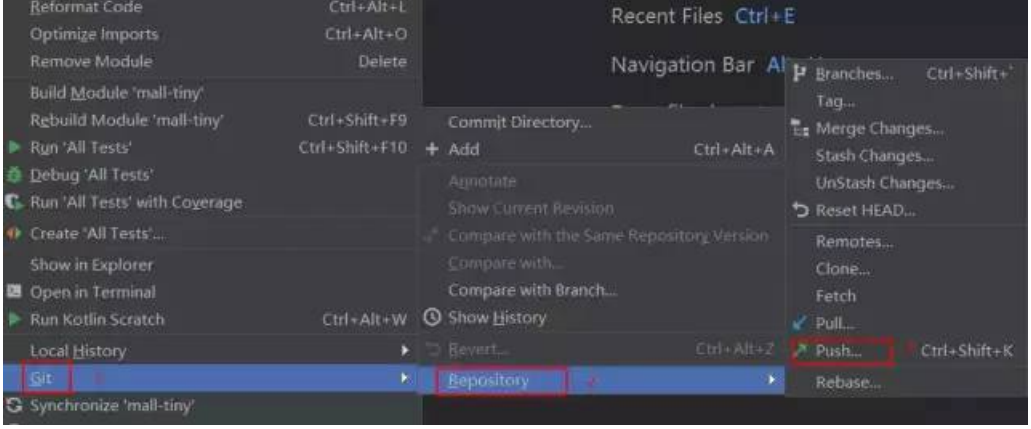


- 添加注释并提交代码：

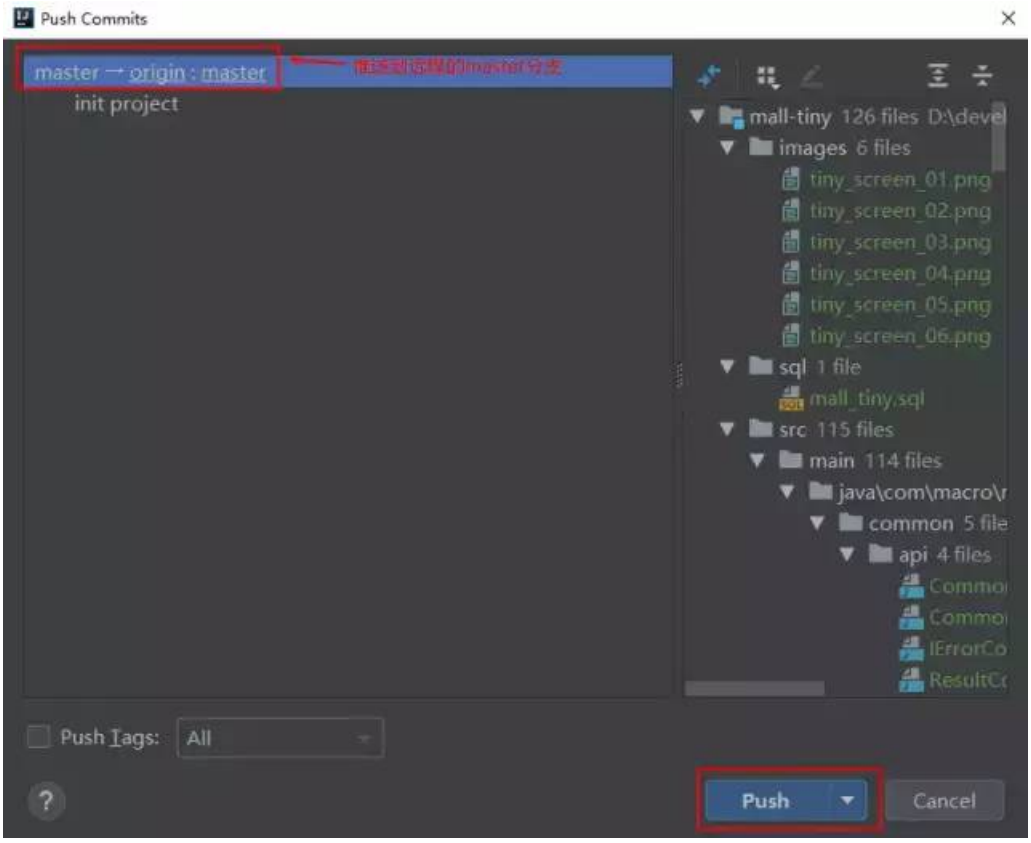


将代码推送到远程仓库

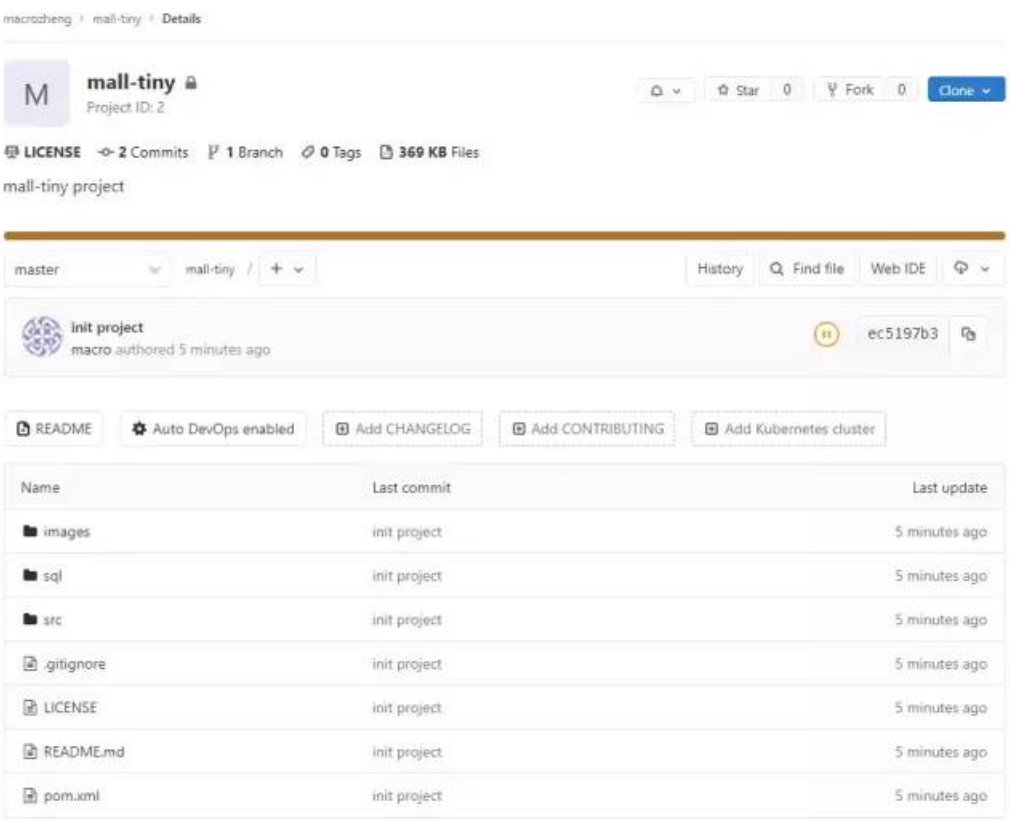
- 点击push按钮推送代码：



- 确认推送内容：

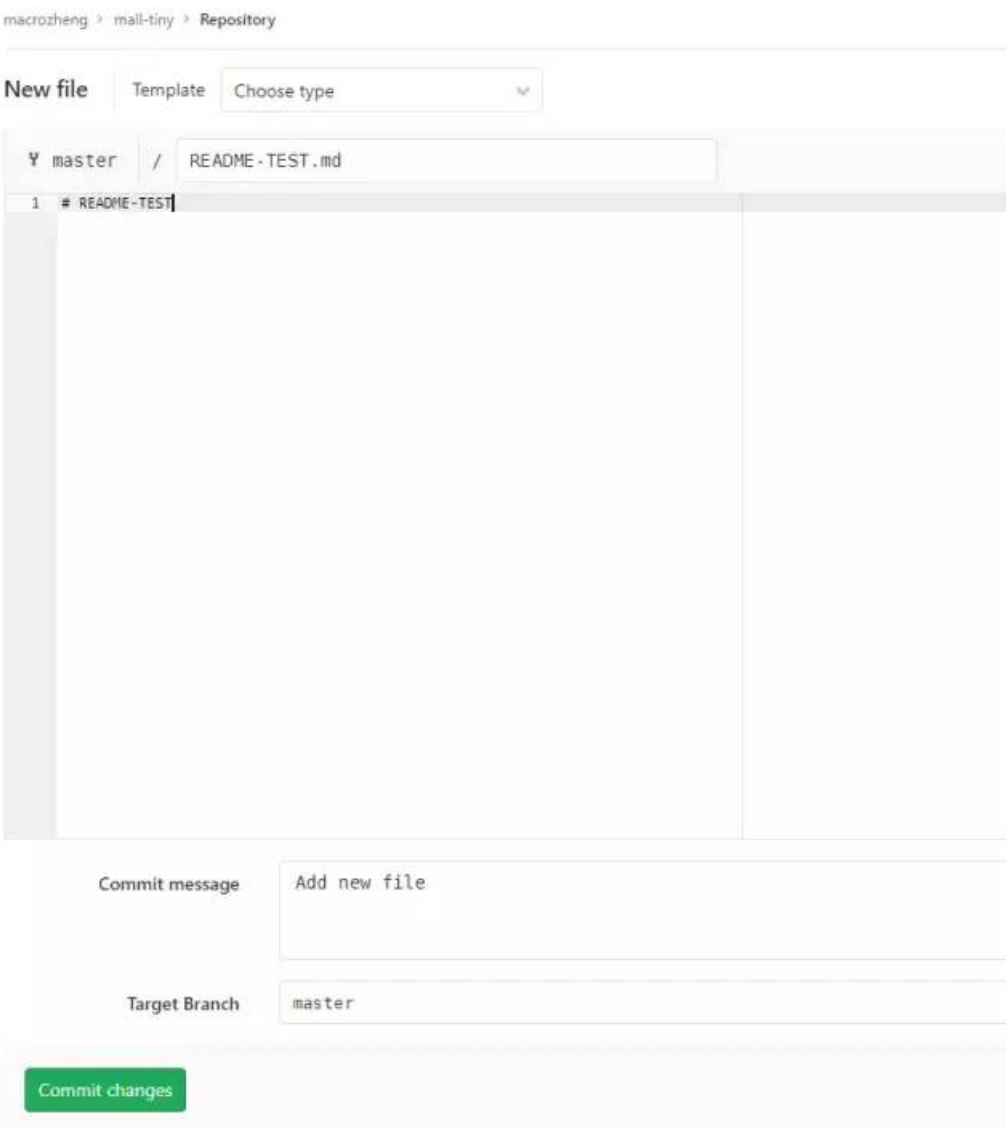


- 查看远程仓库发现已经提交完成：

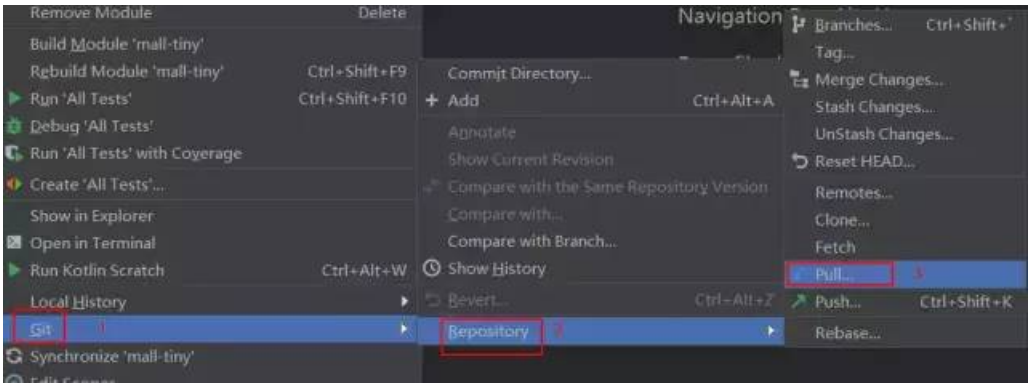


从远程仓库拉取代码

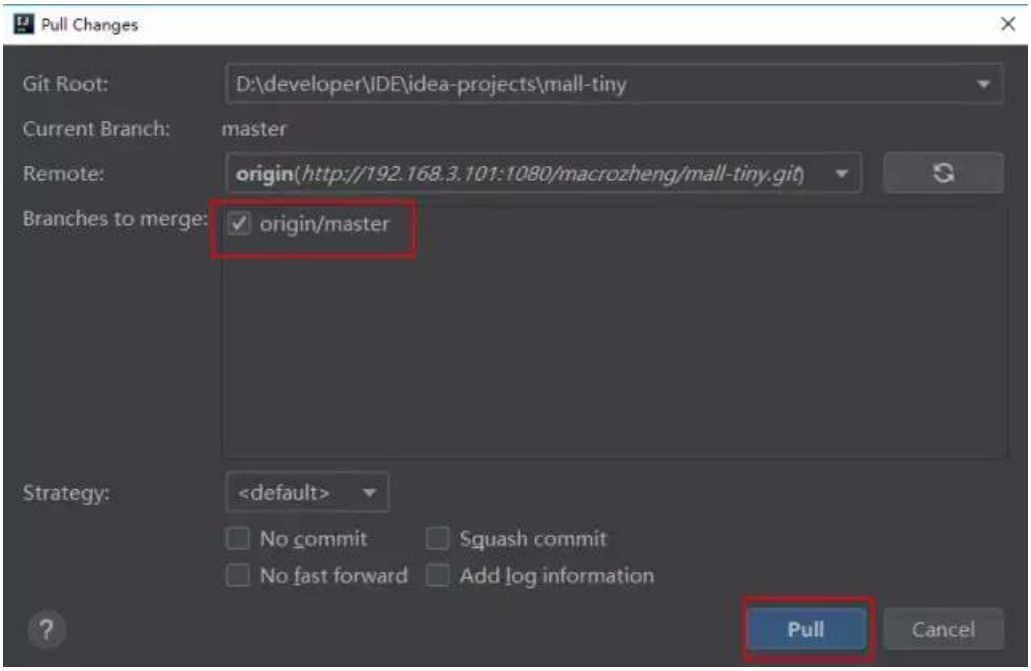
- 在远程仓库添加一个README-TEST.md文件：



- 从远程仓库拉取代码：

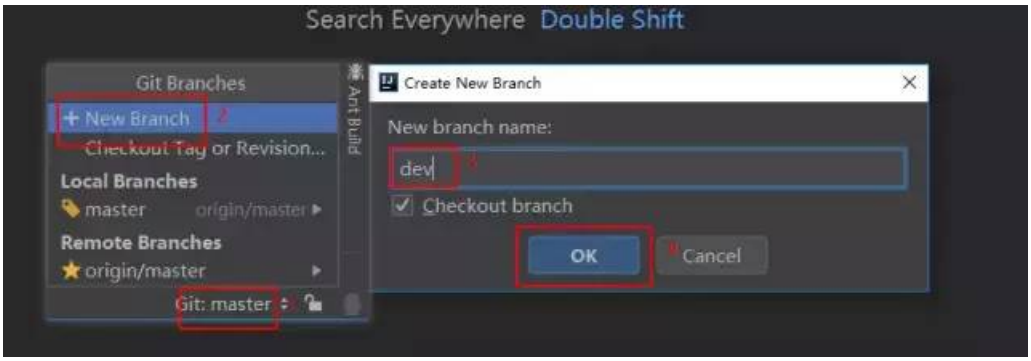


- 确认拉取分支信息：

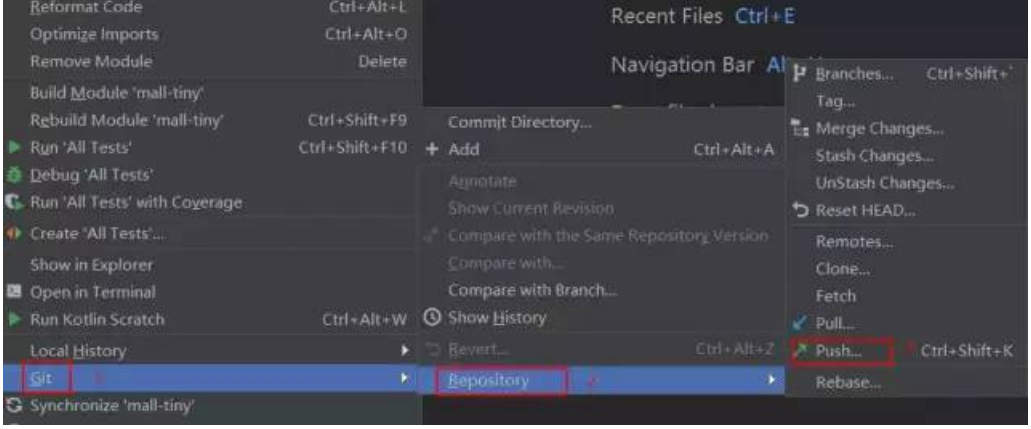


从本地创建分支并推送到远程

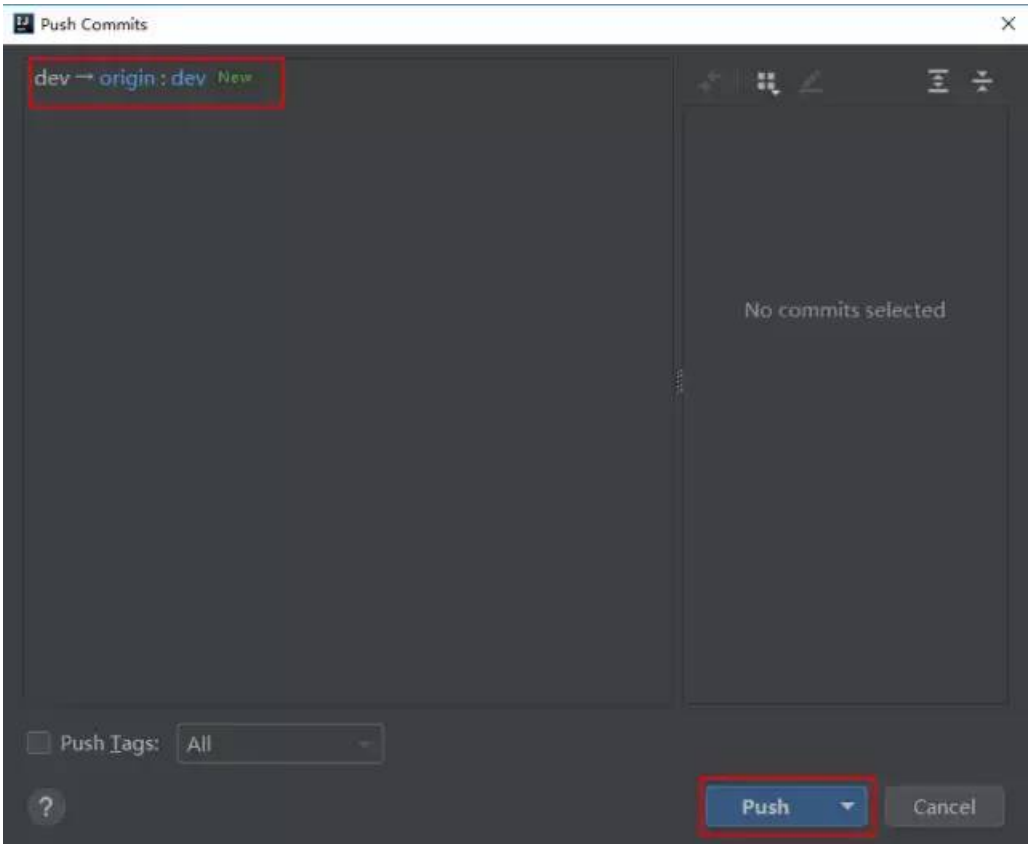
- 在本地创建dev分支，点击右下角的Git:master按钮：



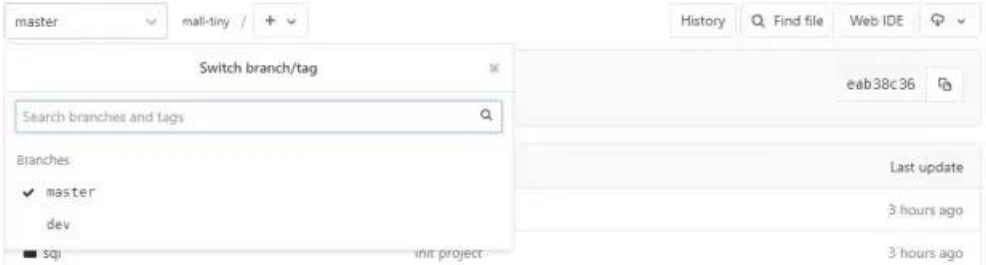
- 使用push将本地dev分支推送到远程：



- 确认推送内容：

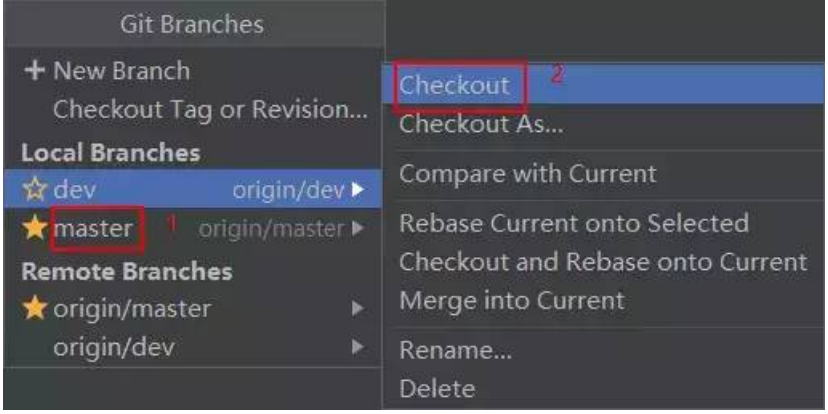


- 查看远程仓库发现已经创建了dev分支：



分支切换

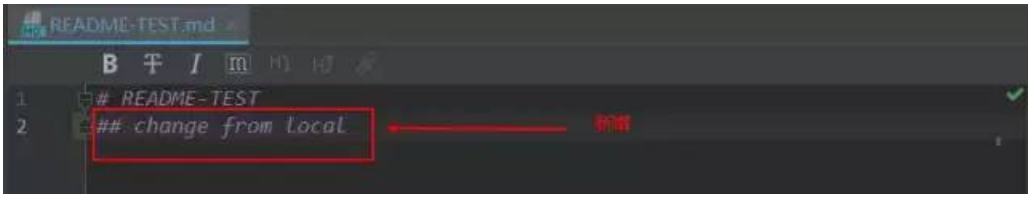
- 从dev分支切换回master分支：



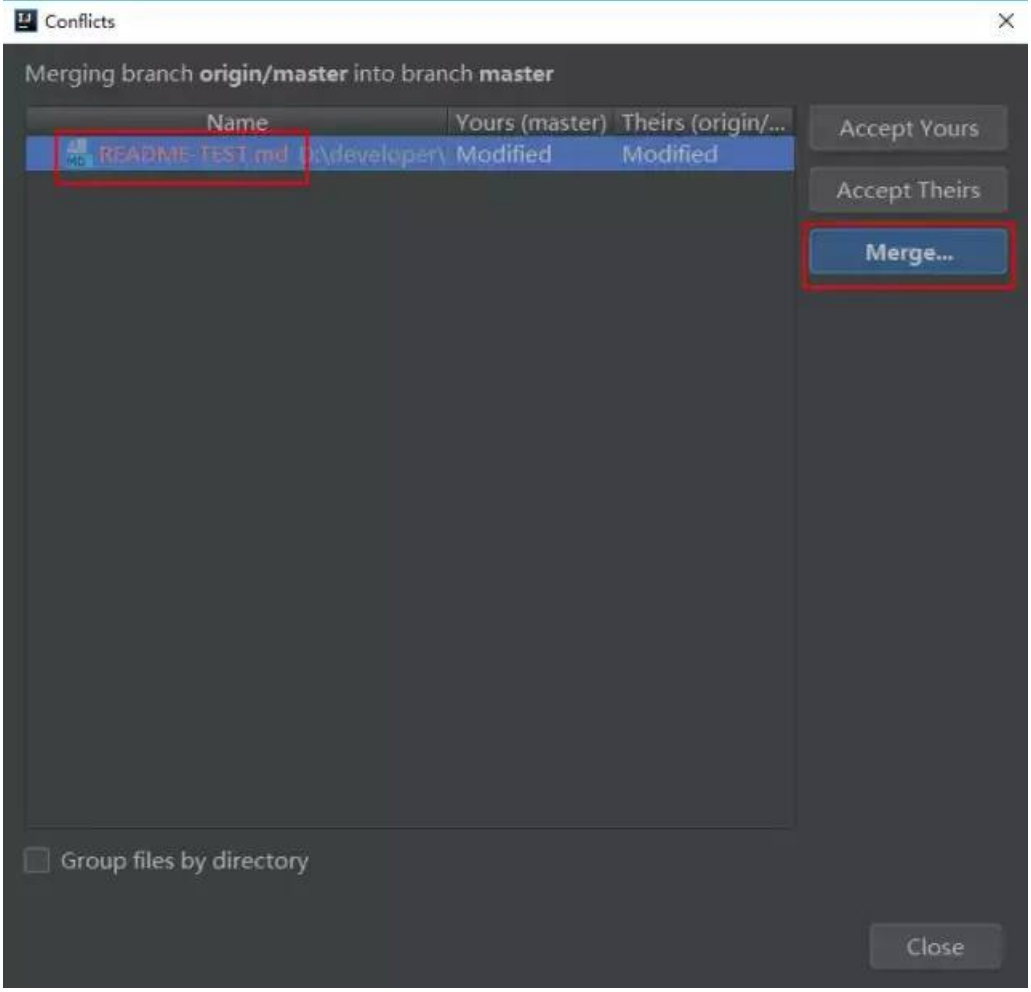
Git文件冲突问题解决

- 修改远程仓库代码：

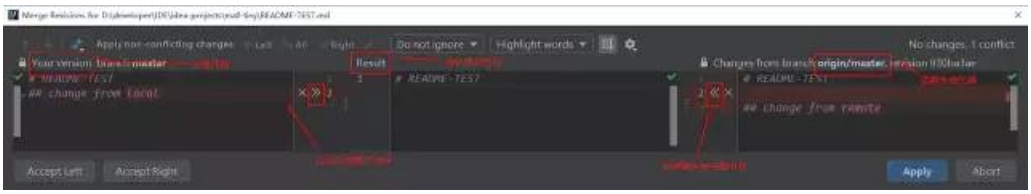
- 修改本地仓库代码：



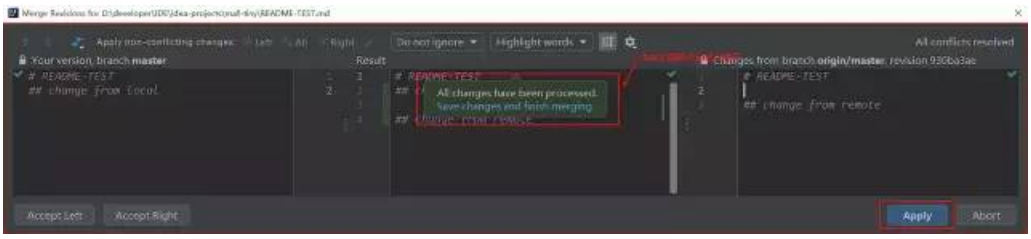
- 提交本地仓库代码并拉取，发现代码产生冲突，点击Merge进行合并：



- 点击箭头将左右两侧代码合并到中间区域：



- 冲突合并完成后，点击Apply生效：

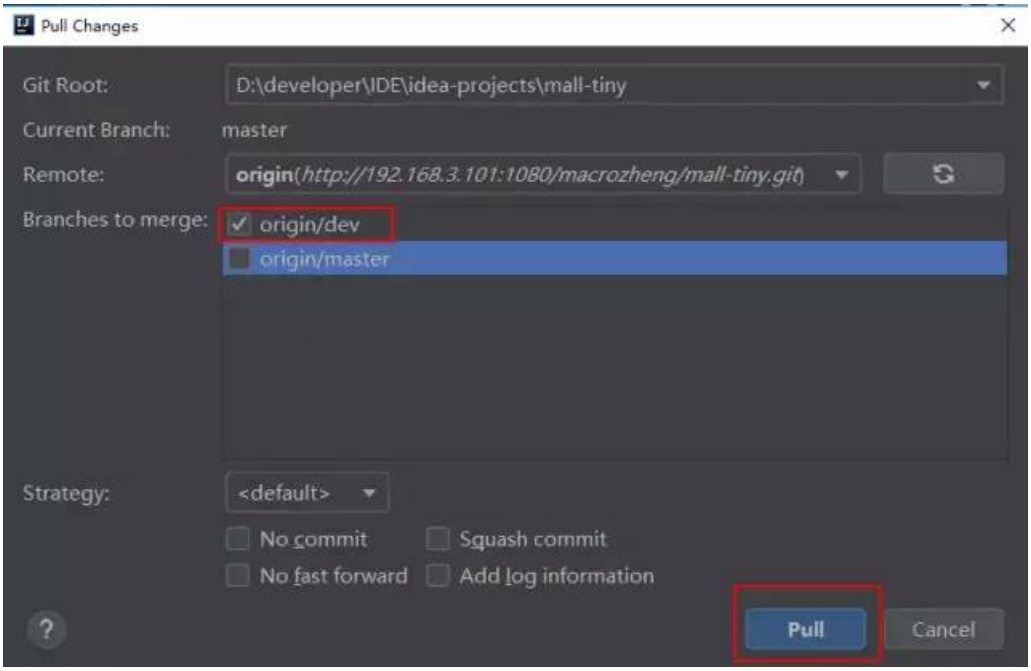


- 提交代码并推送到远程。

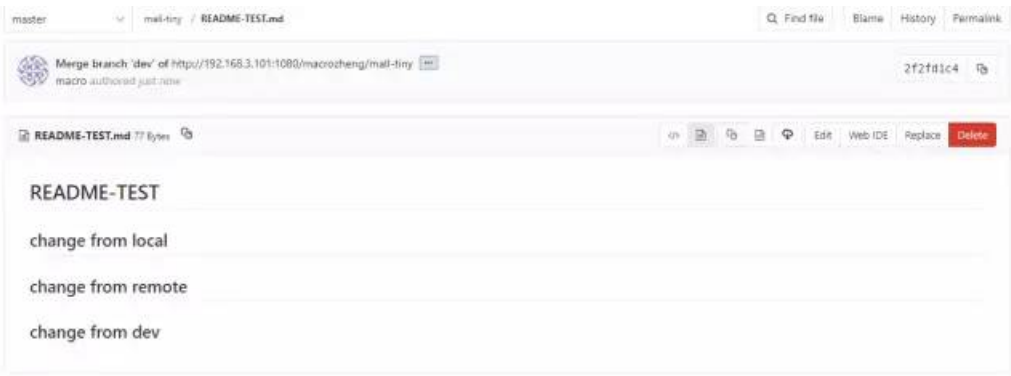
从dev分支合并代码到master

- 在远程仓库修改dev分支代码：

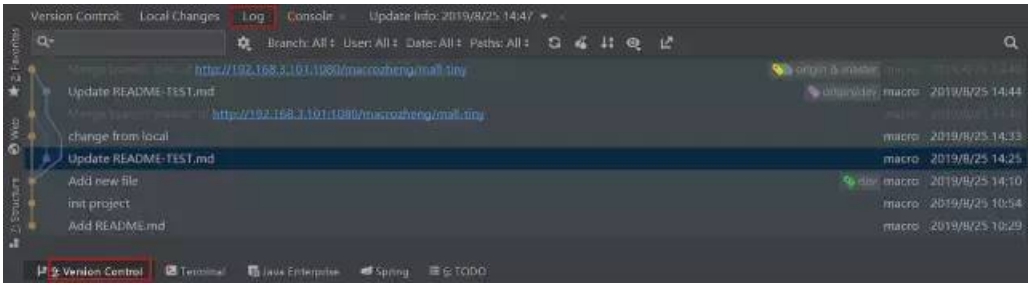
- 在本地仓库拉取代码，选择从dev分支拉取并进行合并：



- 发现产生冲突，解决后提交并推送到远程仓库即可。



查看Git仓库提交历史记录



作者：MacroZheng

链接：https://juejin.im/post/5d667fc6e51d453b5d4d8da5

如果喜欢本篇文章，欢迎转发、点赞。关注订阅号「Web项目聚集地」，回复「全栈」即可获取 2019 年最新 Java、Python、前端学习视频资源。

推荐阅读

1. 这代码写的，狗屎一样

2. 这代码写的，狗屎一样（下）

3. 除了负载均衡，Nginx 还可以做很多

4. 快来薅当当的羊毛！

5. 聊一聊 Java 泛型中的通配符

6. 数据库不使用外键的 9 个理由

Web项目聚集地

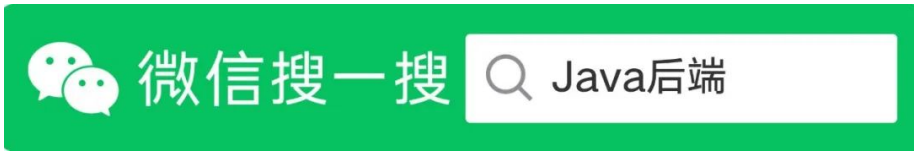
微信扫描二维码，关注我的公众号

喜欢文章，点个在看

阅读原文

IntelliJ IDEA下的使用 Git

J'KYO Java后端 3月11日



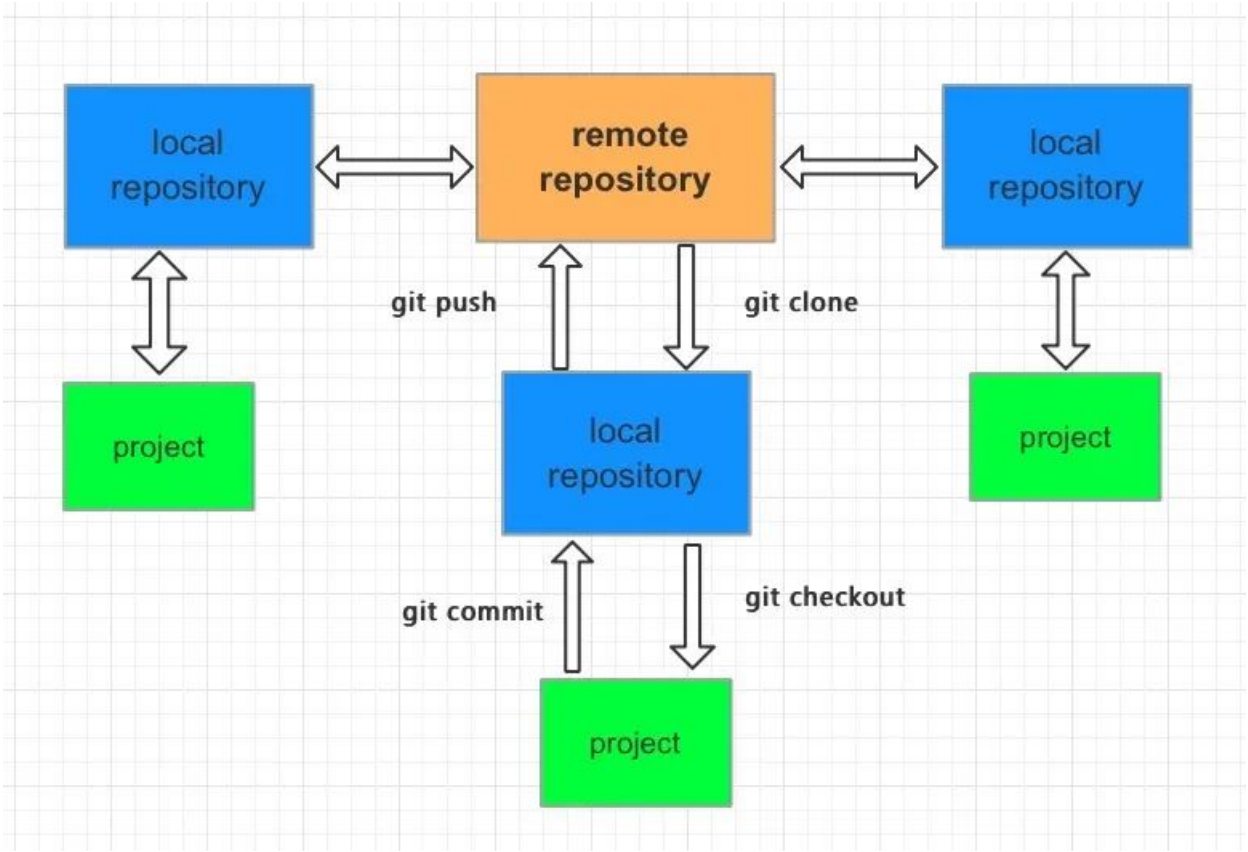
作者| J'KYO

链接 | cnblogs.com/pejsidney/p/9199115.html

1、Git 简介

Git是目前流行的分布式版本管理系统。它拥有两套版本库，本地库和远程库，在不进行合并和删除之类的操作时这两套版本库互不影响。也因此其近乎所有的操作都是本地执行，所以在断网的情况下任然可以提交代码，切换分支。git又使用了SHA-1哈希算法确保了在文件传输时变得不完整、磁盘损坏导致数据丢失时能立即察觉到。

git的基本工作流程：



- git clone：将远程的Master分支代码克隆到本地仓库
- git checkout：切出分支出来开发
- git add：将文件加入库跟踪区
- git commit：将库跟踪区改变的代码提交到本地代码库中
- git push： 将本地仓库中的代码提交到远程仓库

Git 分支

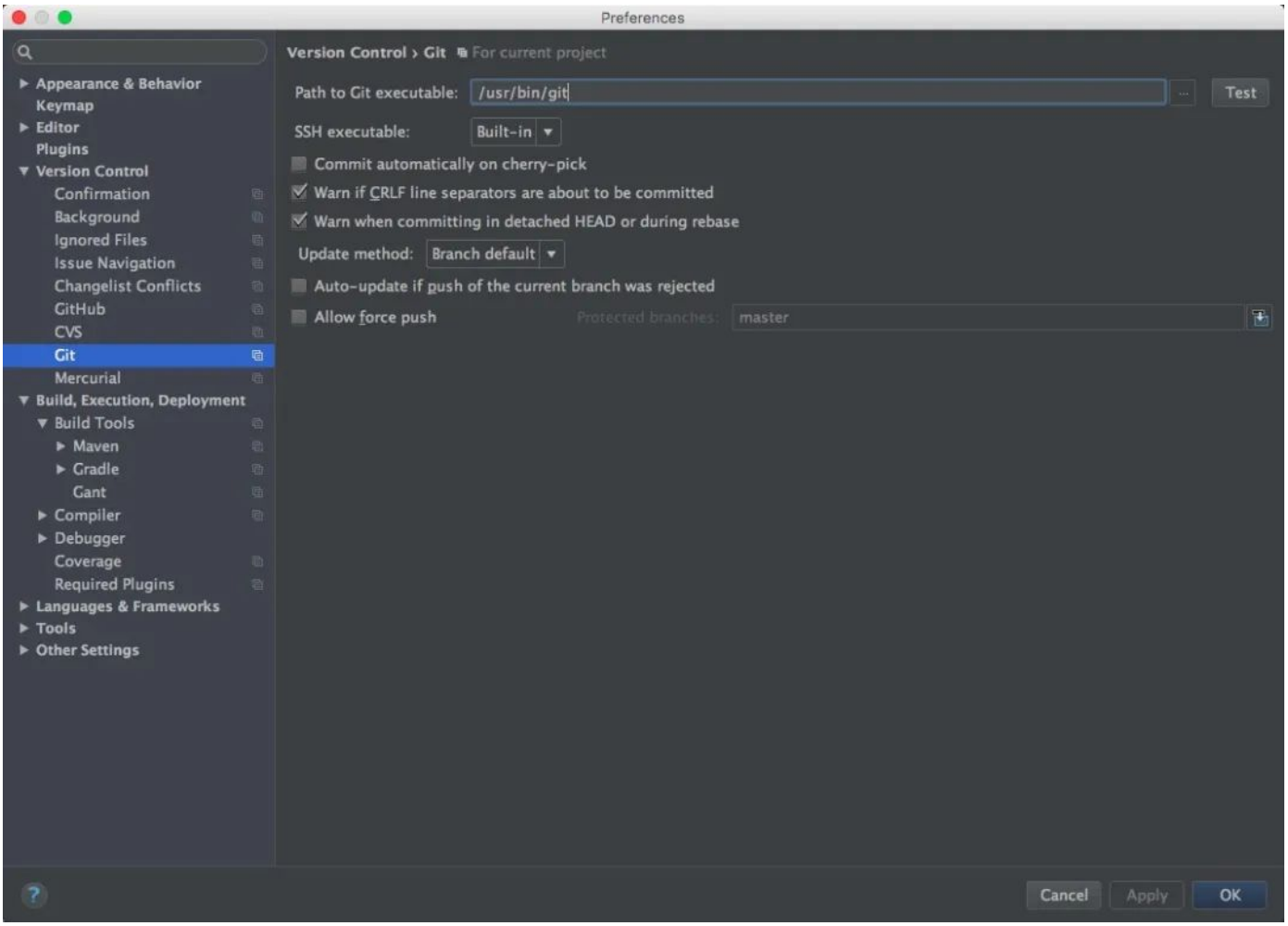
- 主分支
  - master分支：存放随时可供生产环境中的部署的代码

    - develop分支：存放当前最新开发成果的分支，当代码足够稳定时可以合并到master分支上去。
- 辅助分支
    - feature分支：开发新功能使用，最终合并到develop分支或抛弃掉
    - release分支：做小的缺陷修正、准备发布版本所需的各项说明信息
    - hotfix分支：代码的紧急修复工作

2、Git在IntelliJ IDEA下的使用

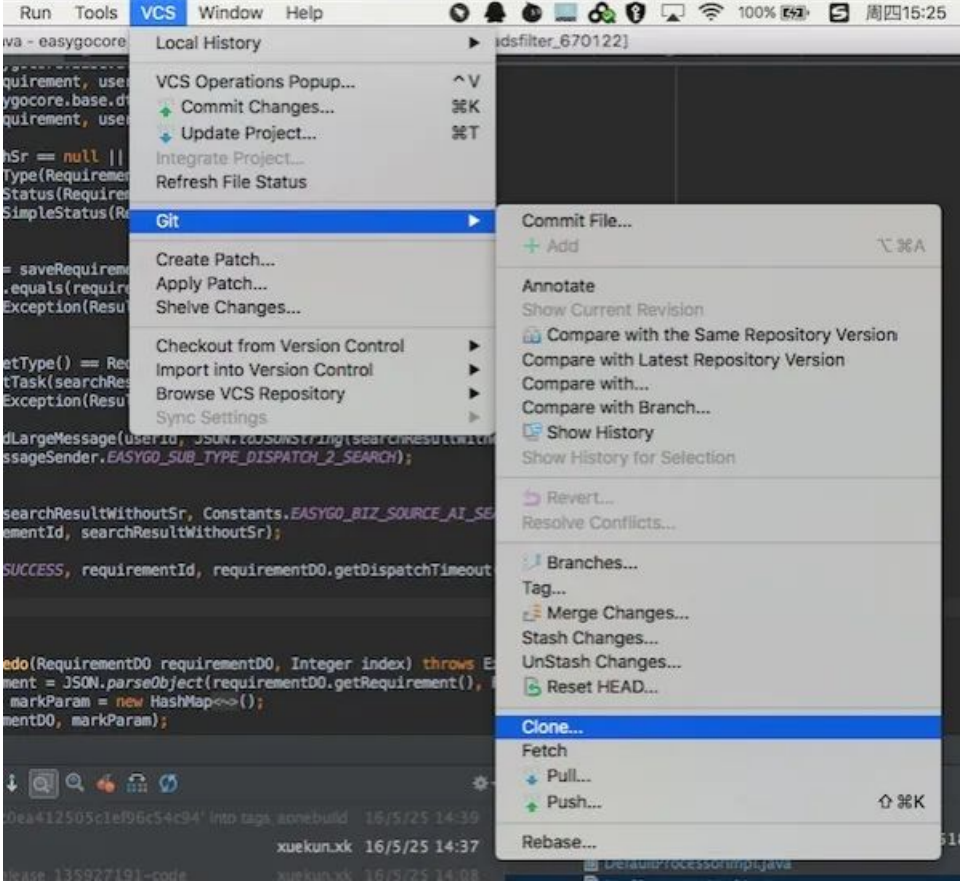
2.1、IntelliJ IDEA下配置 Git

- 本地安装好git，并配置合理的SSH key，具体看这里
- IntelliJ IDEA->Performance->Version Control->git 将自己安装git的可执行文件路径填入Path to Git executable，点击Test测试一下

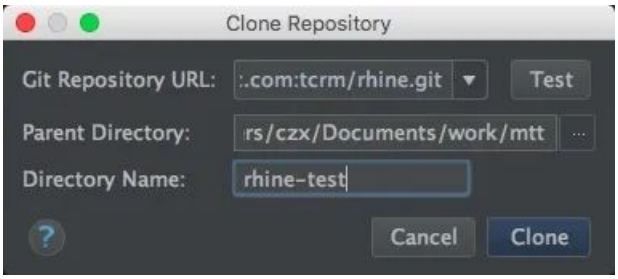


2.2、git clone

- VCS->Git->Clone

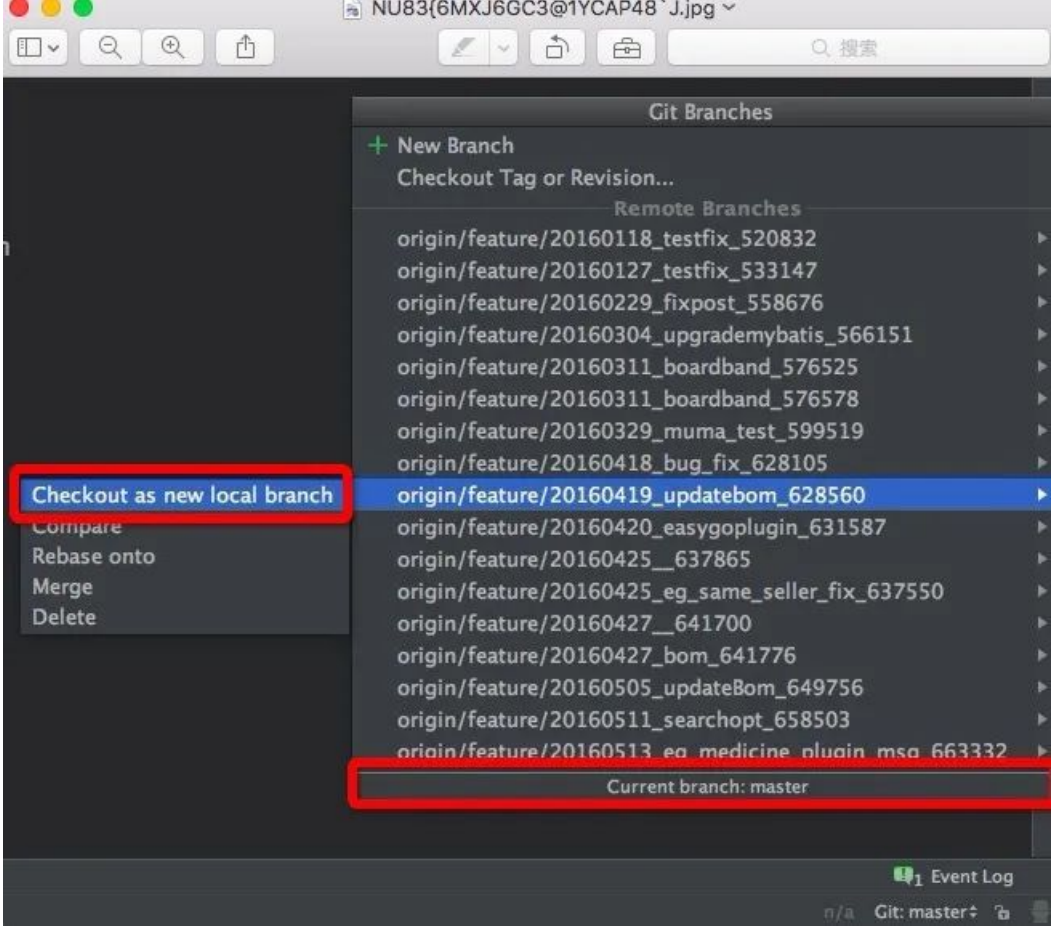


- 输入你的远程仓库地址,点击测试一下地址是否正确

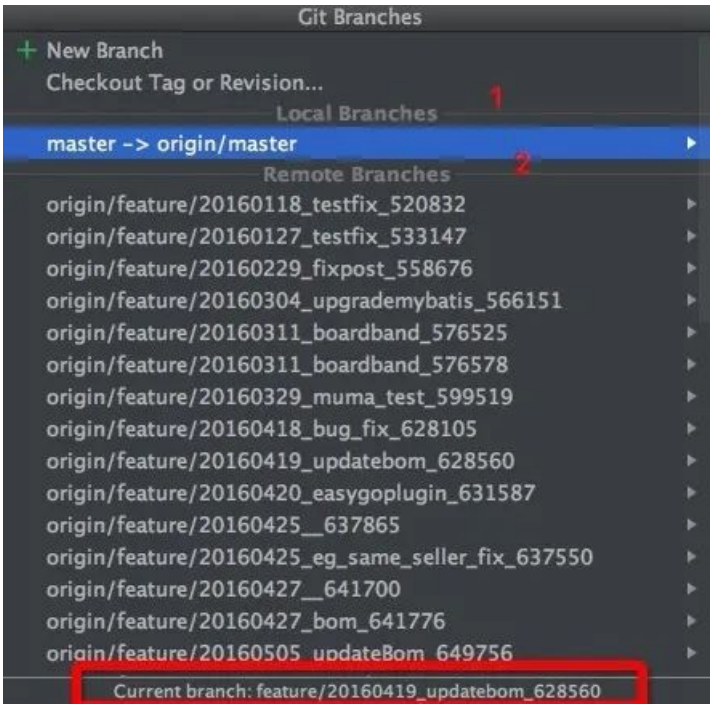


2.3、git checkout

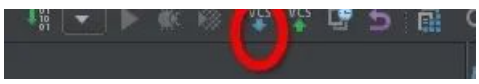
- 在IntelliJ IDEA右下角有一个git的分支管理，点击。选择自己需要的分支，checkout出来



- checkout出来，会在底端显示当前的分支。其中1显示的为本地仓库中的版本，2为远程仓库中的版本

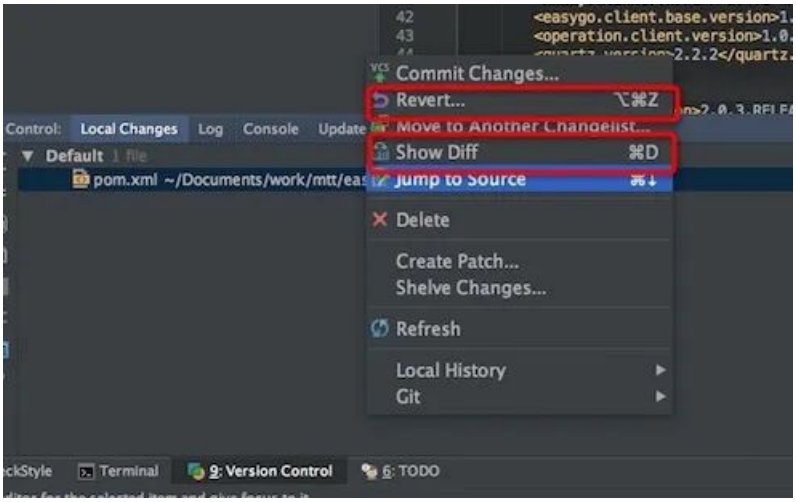


- 点击IDE的右上角的向下箭头的VCS，将分支的变更同步到本地



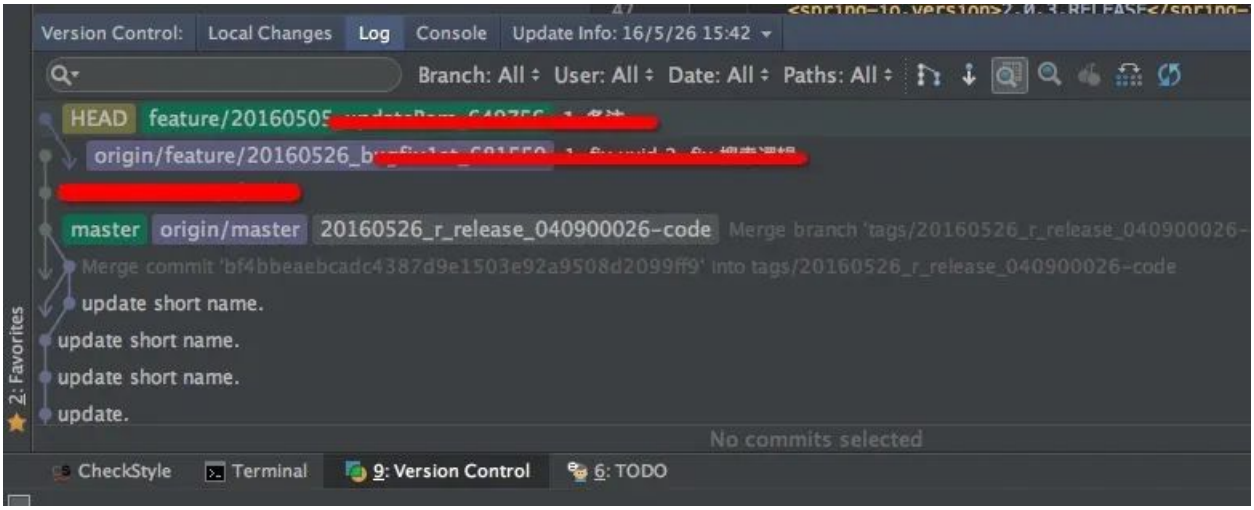
2.4、git diff

- 在local changes 中选中要比对的文件，右键选择show diff 便可以查看文件的变动。或者选择Revert放弃文件的改动



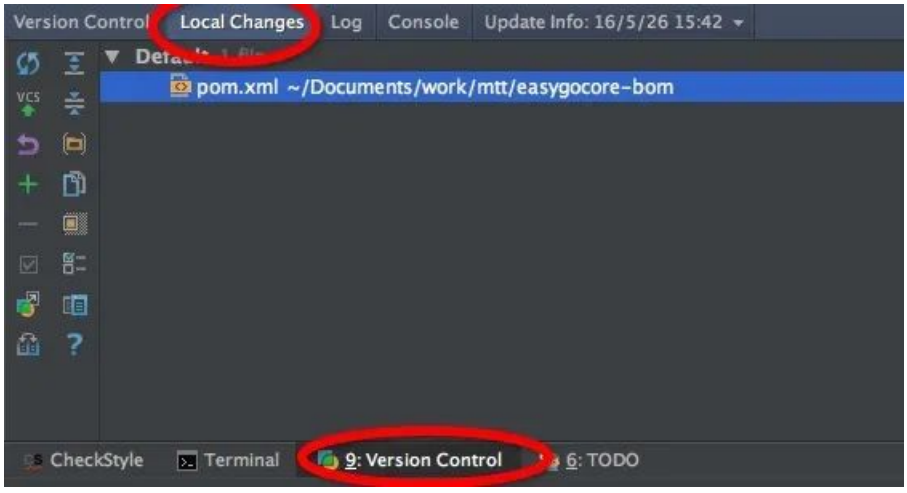
2.5、git log

- 在Version Control下选择Log，可以查看提交历史

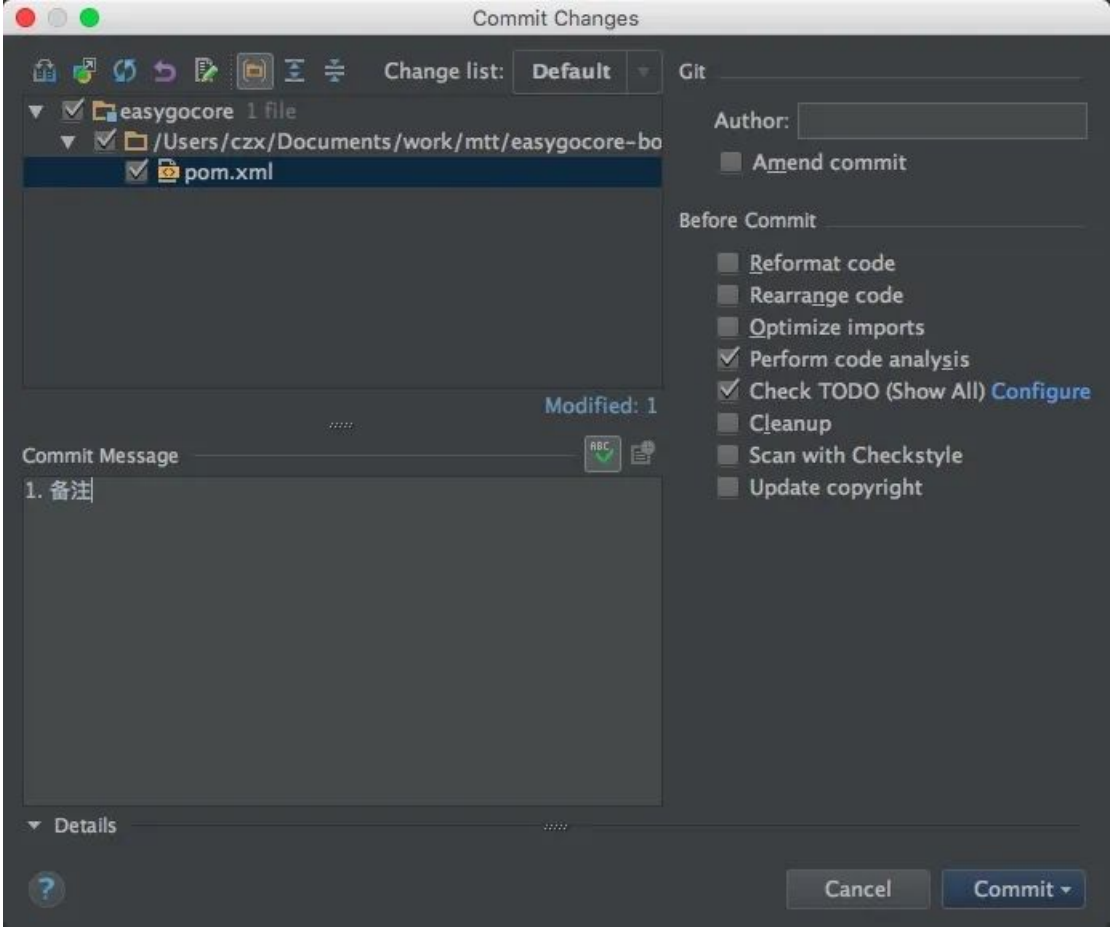


2.6、git commit

- 默认导入的工程已经git add加入库跟踪区了
- 随便修改一下pom.xml文件，其修改的文件会显示在Version Control中的local changes下

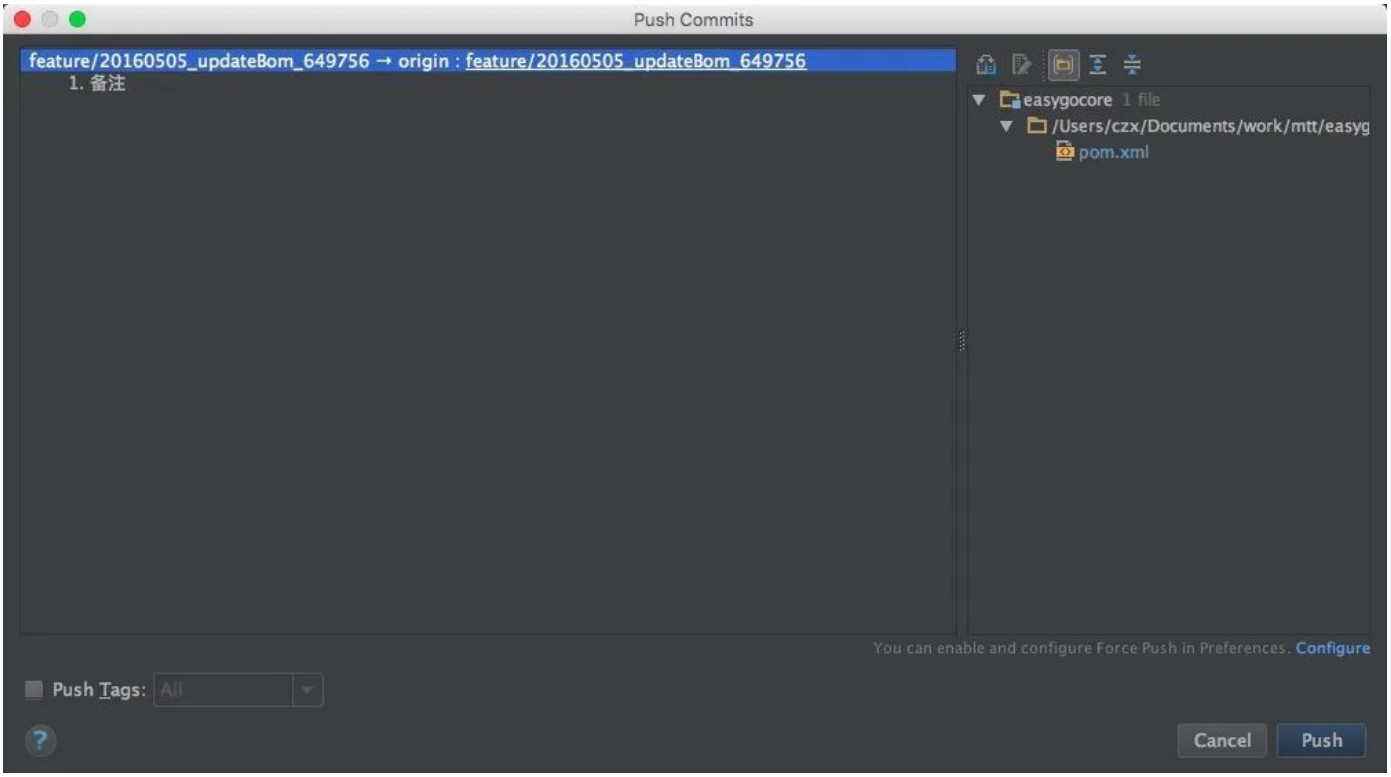


- 点击IDE右上角的向上箭头的VCS， git commit, 写上日志提交到本地代码库中



2.7、git push

- VCS->Git->Push 将本地代码提交到远程仓库



2.8、在Idea命令行使用git

常用命令请参考:

收藏了！IntelliJ IDEA 快捷键 Windows 版本

IntelliJ IDEA 常用快捷键 - Mac版本

-END-

推荐阅读

1. 如何写出让同事无法维护的代码?
2. 用好 Git 和 SVN，轻松驾驭版本管理
3. 如何使用 Java 灵活读取 Excel 内容？
4. IntelliJ IDEA 快捷键 Windows 版本

阅读原文

不给队友拖后腿！团队开发中 Git 最佳实践

欧雷 Java后端 2019-10-09

来源｜欧雷

编辑｜GitHubDaily（id：GitHubDaily）

出处｜ourai.ws/posts/working-with-git-in-team/

前言

> 在 2005 年的某一天，Linux 之父 Linus Torvalds 发布了他的又一个里程碑作品——Git。它的出现改变了软件开发流程，大大地提高了开发流畅度！直到现在仍十分流行，完全没有衰退的迹象。

本文要从具体实践角度，尤其是在团队协作中，阐述如何去好好地应用 Git。既然是讲在团队中的应用实践，我就尽可能地结合实际场景来讲述。

1.习惯养成

如果一个团队在使用 Git 时没有一些规范，那么将是一场难以醒来的噩梦！然而，规范固然重要，但更重要的是个人素质，在使用 Git 时需要自己养成良好的习惯。

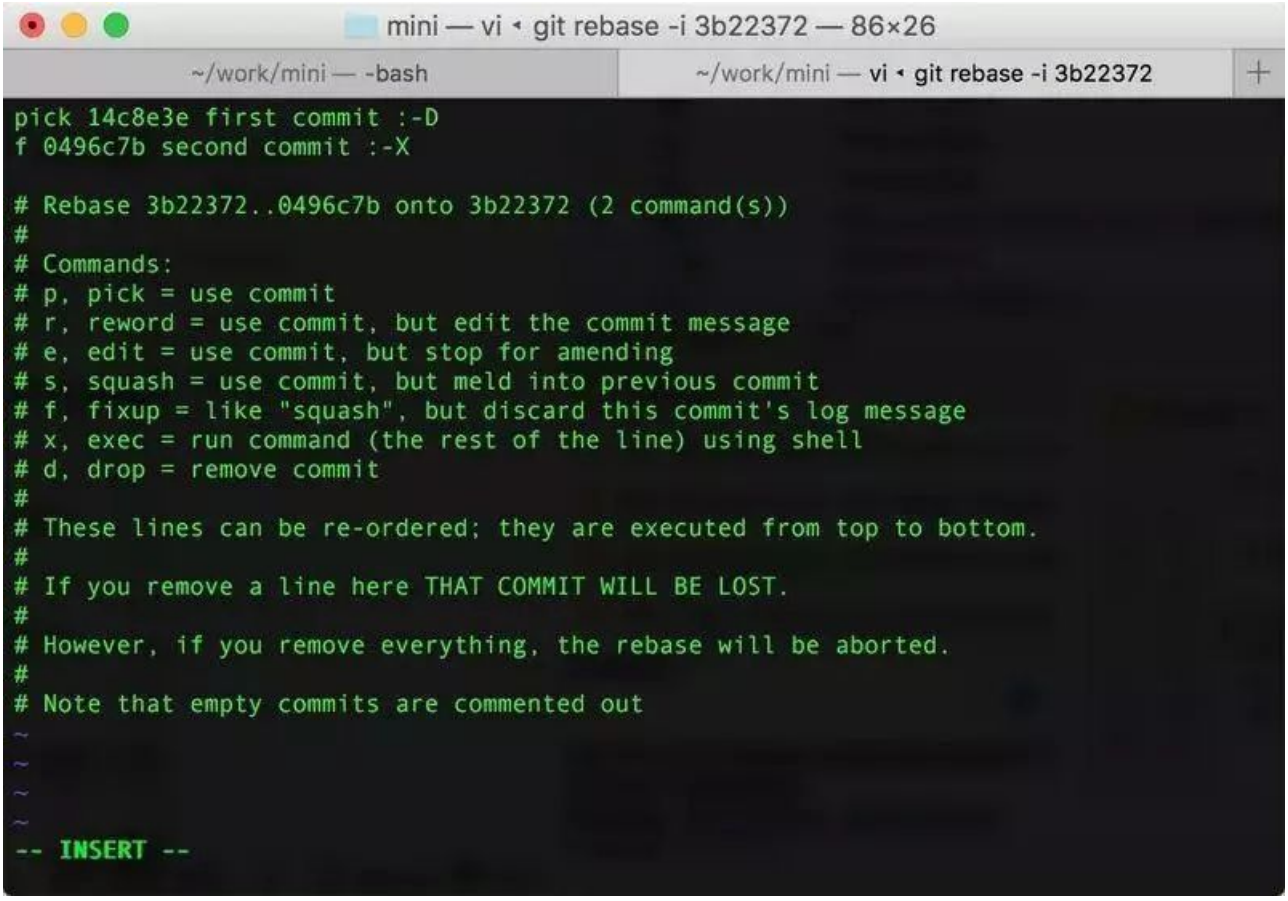
1.1 提交

如何去写一个提交信息，在具体开发工作中主要需要遵守的原则就是「使每次提交都有质量」，只要坚持做到以下几点就 OK 了：

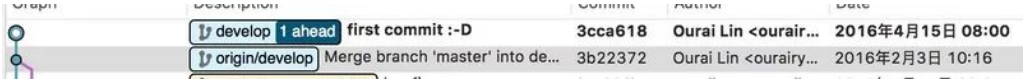
- 提交时的粒度是一个小功能点或者一个 bug fix，这样进行恢复等的操作时能够将「误伤」减到最低；
- 用一句简练的话写在第一行，然后空一行稍微详细阐述该提交所增加或修改的地方；
- 不要每提交一次就推送一次，多积攒几个提交后一次性推送，这样可以避免在进行一次提交后发现代码中还有小错误。

假如已经把代码提交了，对这次提交的内容进行检查时发现里面有个变量单词拼错了或者其他失误，只要还没有推送到远程，就有一个不被他人发觉你的疏忽的补救方法——首先，把失误修正之后提交，可以用与上次提交同样的信息。

然后，终端中执行命令 git rebase -i [SHA]，其中 SHA 是上一次提交之前的那次提交的，在这里是 3b22372。



最后，这样就将两次提交的节点合并成一个，甚至能够修改提交信息！



谁说历史不可篡改了？**前提是，想要合并的那几次提交还没有推送到远程！**

1.2 推送

当自己一个人进行开发时，在功能完成之前不要急着创建远程分支。

1.3 拉取

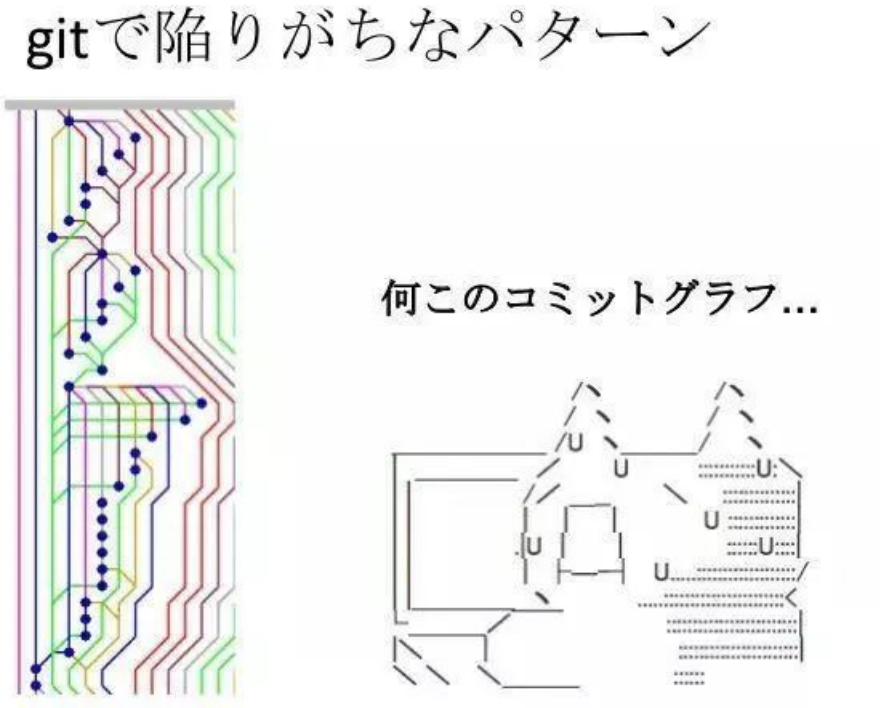
请读张文钿所写的《使用 git rebase 避免無謂的 merge》：https://ihower.tw/blog/archives/3843。

1.4 合并

在将其他分支的代码合并到当前分支时，如果那个分支是当前分支的父分支，为了保持图表的可读性和可追踪性，可以考虑用 git rebase 来代替 git merge；反过来或者不是父子关系的两个分支以及互相已经 git merge 过的分支，就不要采用 git rebase 了，避免出现重复的冲突和提交节点。

2.分支管理

Git 的一大特点就是可以创建很多分支并行开发。正因为它的灵活性，团队中如果没有一个成熟的分支模型的话，那将会是一团糟。



要是谁真把这么乱的提交图表摆在我面前，就给他一个上勾拳！

2.1 分支模型

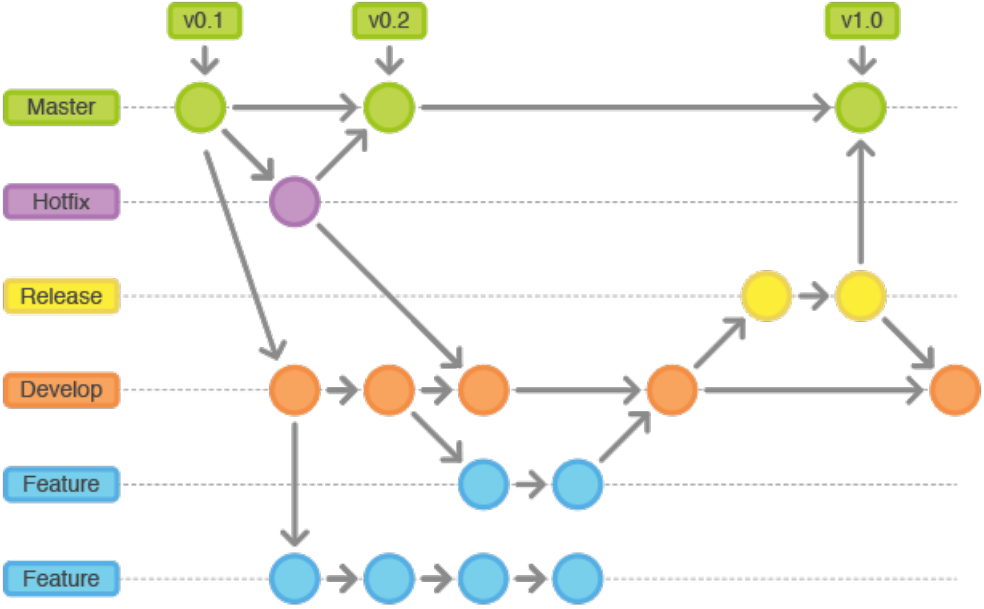
有个很成熟的叫 Git Flow 的分支模型，它能够应对 99% 的场景，剩下的那 1% 留给几乎不存在的极度变态的场景。

需要注意的是，**它只是一个模型，而不是一个工具；你可以用工具去应用这个模型，也可以用最朴实的命令行。所以，重要的是理解概念，不要执着于实行的手段。**

简单说来，Git Flow 就是给原本普普通通的分支赋予了不同的「职责」：

- **master**——最为稳定功能最为完整的随时可发布的代码；
- hotfix——修复线上代码的 bug；
- **develop**——永远是功能最新最全的分支；
- feature——某个功能点正在开发阶段；
- release——发布定期要上线的功能。

看到上面的「master」和「develop」加粗了吧?代表它们是「主要分支」，其他的分支是基于它们派生出来的。主要分支每种类型只能有一个，派生分支每个类型可以同时存在多个。各类型分支之间的关系用一张图来体现就是：



更多信息可参考 xirong 所整理的《Git工作流指南》:https://github.com/xirong/my-git/blob/master/git-workflow-tutorial.md

2.2 工具选择

一直不喜欢「**最好用」这种命题，主观性太强，不会有一个结论。对于工具的选择，我一直都是秉承「哪个能更好地解决问题就用哪个」这个原则。所以，只要不影响到团队，用什么工具都是可以接受的。但根据多数开发人员的素质情况来看，建议使用图形化工具，例如 SourceTree（https://www.sourcetreeapp.com）。如果想用命令行，可以啊！先在心里问下自己：「我 Git 牛逼不？会不会惹麻烦给别人？」

在团队中应用 Git Flow 时，推荐使用 SourceTree 与 GitLab（https://gitlab.com/）配合的形式：

- 用 SourceTree 创建 feature 等分支以及本地的分支合并、删除；
- 用 GitLab 做代码审核和远程的分支合并、删除。

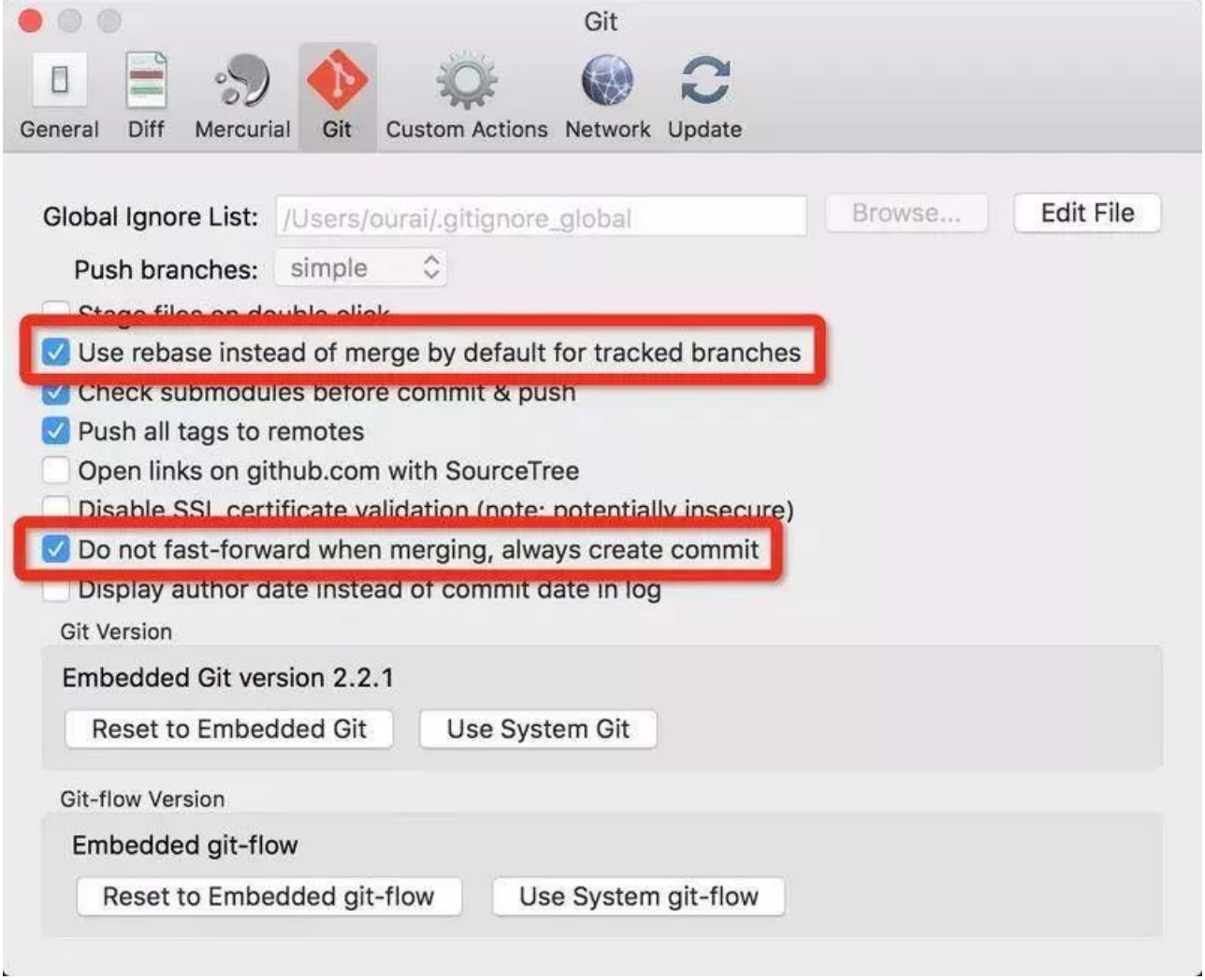
SourceTree 和 GitLab 应该是相辅相成的存在，而不是互相取代。

3.事前准备

为了将一些规范性的东西和 Git Flow 的部分操作自动化处理，要对 SourceTree 和 GitLab 进行一下配置。

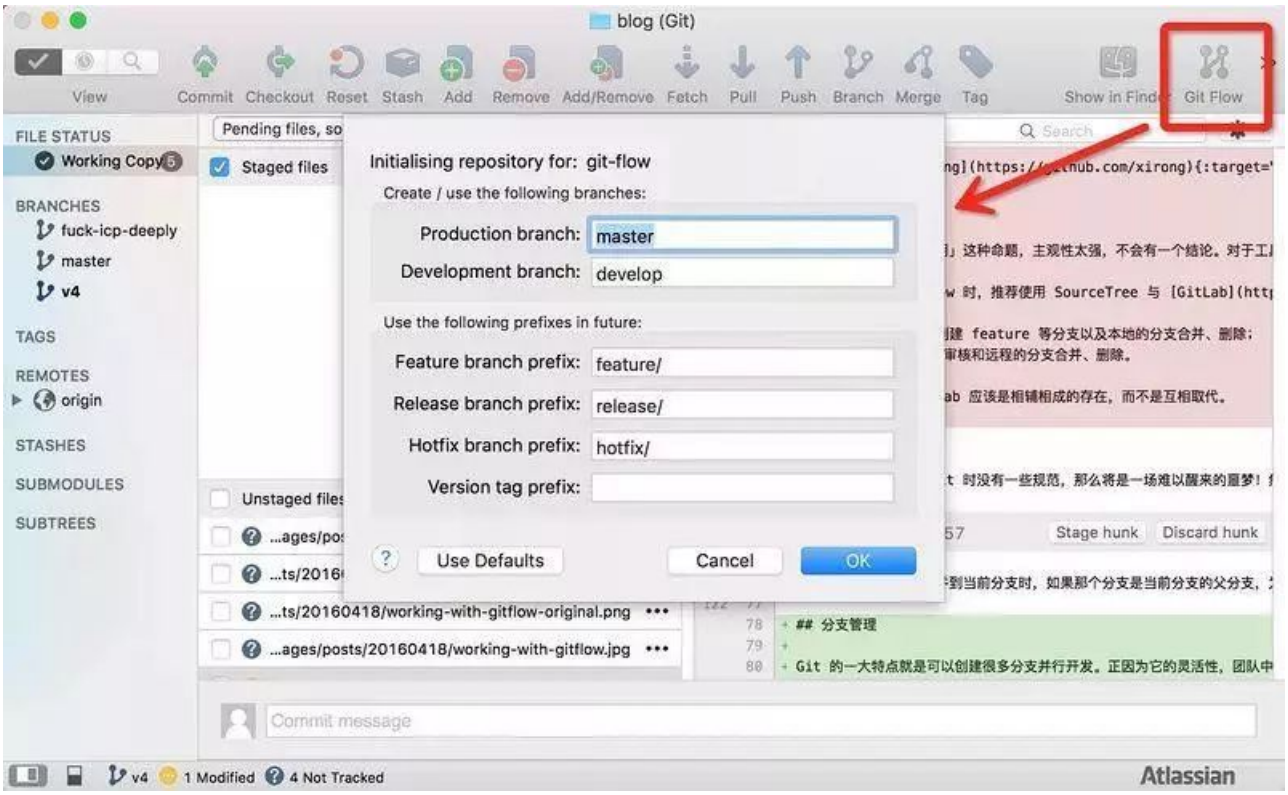
3.1 SourceTree

按下 command + ，调出「Preferences」界面并切换到「Git」标签，勾选「Use rebase instead of merge by default for tracked branches」和「Do not fast-forward when merging, always create commit」。



这样设置之后，在点「Pull」按钮拉取代码时会自动执行 git pull —rebase；并且，每次合并时会自动创建新的包含分支信息的提交节点。

接下来，点击工具栏中的「Git Flow」按钮将相关的流程自动化。如果没有特殊需求，直接按下对话框中的「OK」就好了。初始化完成后会自动切换到 develop 分支。



这下再点「Git Flow」按钮所弹出的对话框就是选择创建分支类型的了。

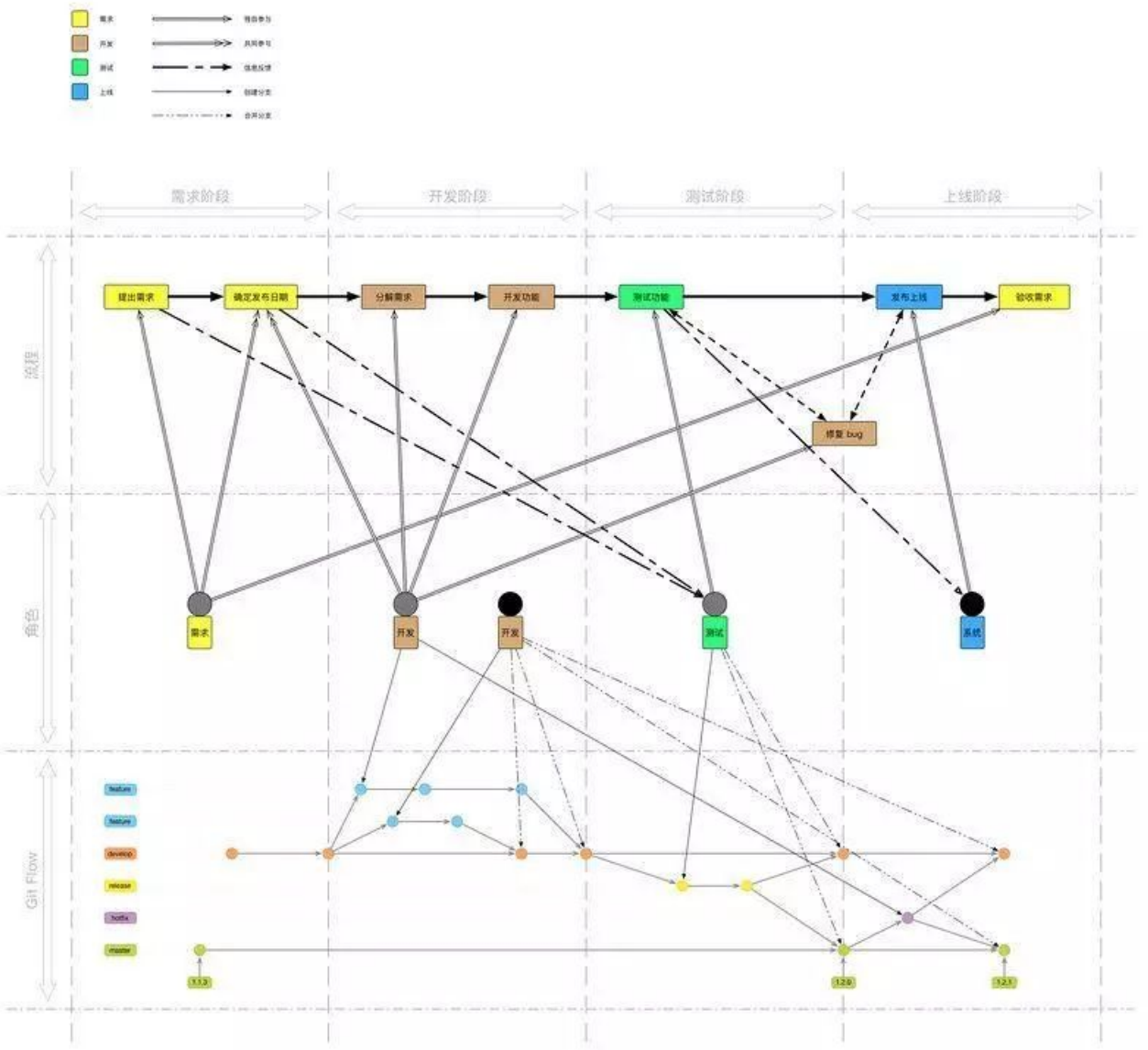
3.2 GitLab

在创建项目仓库后一定要把主要分支，也就是 master 和 develop 给保护起来。为它们设置权限，只有项目负责人可以进行推送和删除等操作。

被保护的分支在列表中会有特殊的标记进行区分。

4.开发流程

在引入 Git Flow 之后，所有工作都要围绕着它来展开，将原本的流程与之结合形成「基于 Git Flow 的开发流程」。



4.1 开发功能

在确定发布日期之后，将需要完成的内容细分一下分配出去，负责某个功能的开发人员利用 SourceTree 所提供的 Git Flow 工具创建一个对应的 feature 分支。如果是多人配合的话，创建分支并做一些初始化工作之后就推送创建远程分支；否则，直到功能开发完毕要合并进 develop 前，不要创建远程分支。

功能开发完并自测之后，先切换到 develop 分支将最新的代码拉取下来，再切换回自己负责的 feature 分支把 develop 分支的代码合并进来。合并方式参照上文中的「合并」，如果有冲突则自己和配合的人一起解决。

然后，到 GitLab 上的项目首页创建合并请求（merge request）。

「来源分支」选择要被合并的 feature 分支且「目标分支」选择 develop 分支后点击「比较分支」按钮，在出现的表单中将处理人指派为项目负责人。

项目负责人在收到合并请求时，应该先做下代码审核看看有没有明显的严重的错误；有问题就找负责开发的人去修改，没有就接受请求并删除对应的 feature 分支。

在将某次发布的所需功能全部开发完成时，就可以交付测试了。

4.2 测试功能

负责测试的人创建一个 release 分支部署到测试环境进行测试；若发现了 bug，相应的开发人员就在 release 分支上或者基于 release 分支创建一个分支进行修复。

4.3 发布上线

当确保某次发布的功能可以发布时，负责发布的人将 release 分支合并进 master 和 develop 并打上 tag，然后打包发布到线上环境。

建议打 tag 时在信息中详细描述这次发布的内容，如：添加了哪些功能，修复了什么问题。

4.4 修复问题

当发现线上环境的代码有小问题或者做些文案修改时，相关开发人员就在本地创建 hotfix 分支进行修改，具体操作参考「开发功能」。

如果是相当严重的问题，可能就得回滚到上一个 tag 的版本了。

5.额外说明

这里所提到的事情，虽非必需，但知道之后却会如虎添翼。

5.1 分支命名

除了主要分支的名字是固定的之外，派生分支是需要自己命名的，这里就要有个命名规范了。强烈推荐用如下形式：

- feature——按照功能点（而不是需求）命名；
- release——用发布时间命名，可以加上适当的前缀；
- hotfix——GitLab 的 issue 编号或 bug 性质等。

另外还有 tag，用语义化的版本号（http://semver.org/lang/zh-CN/）命名。

5.2 发布日期

发布频率是影响开发人员与测试人员的新陈代谢和心情的重要因素之一，频繁无规律的发布会导致内分泌失调、情绪暴躁，致使爆粗口、砸电脑等状况出现。所以，确保一个固定的发布周期至关重要！

在有一波或几波需求来临之时，想挡掉是不太可能的，但可以在评审时将它（们）分期，在某个发布日之前只做一部分。这是必须要控制住的！不然任由着需求方说「这个今天一定要上」「那个明天急着用」的话，技术人员就等着进医院吧！

-END-

推荐阅读

1. Java后端优质文章整理
2. JDK 13 新特性一览
3. 14 个实用的数据库设计技巧
4. 面试官：Redis 内存满了怎么办？
5. 如何设计 API 接口，实现统一格式返回？

Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看

团队开发中 Git 最佳实践，不给队友拖后腿

Java后端 2019-09-15

本文地址：https://segmentfault.com/a/1190000004963641

> 在 2005 年的某一天，Linux 之父 Linus Torvalds 发布了他的又一个里程碑作品——Git。它的出现改变了软件开发流程，大大地提高了开发流畅度！直到现在仍十分流行，完全没有衰退的迹象。

本文不是一篇 Git 入门教程，Git 入门教程大家可以参考:Git 教程合集。

本文要从具体实践角度，尤其是在团队协作中，阐述如何去好好地应用 Git。既然是讲在团队中的应用实践，我就尽可能地结合实际场景来讲述。

1.习惯养成

如果一个团队在使用 Git 时没有一些规范，那么将是一场难以醒来的噩梦！然而，规范固然重要，但更重要的是个人素质，在使用 Git 时需要自己养成良好的习惯。

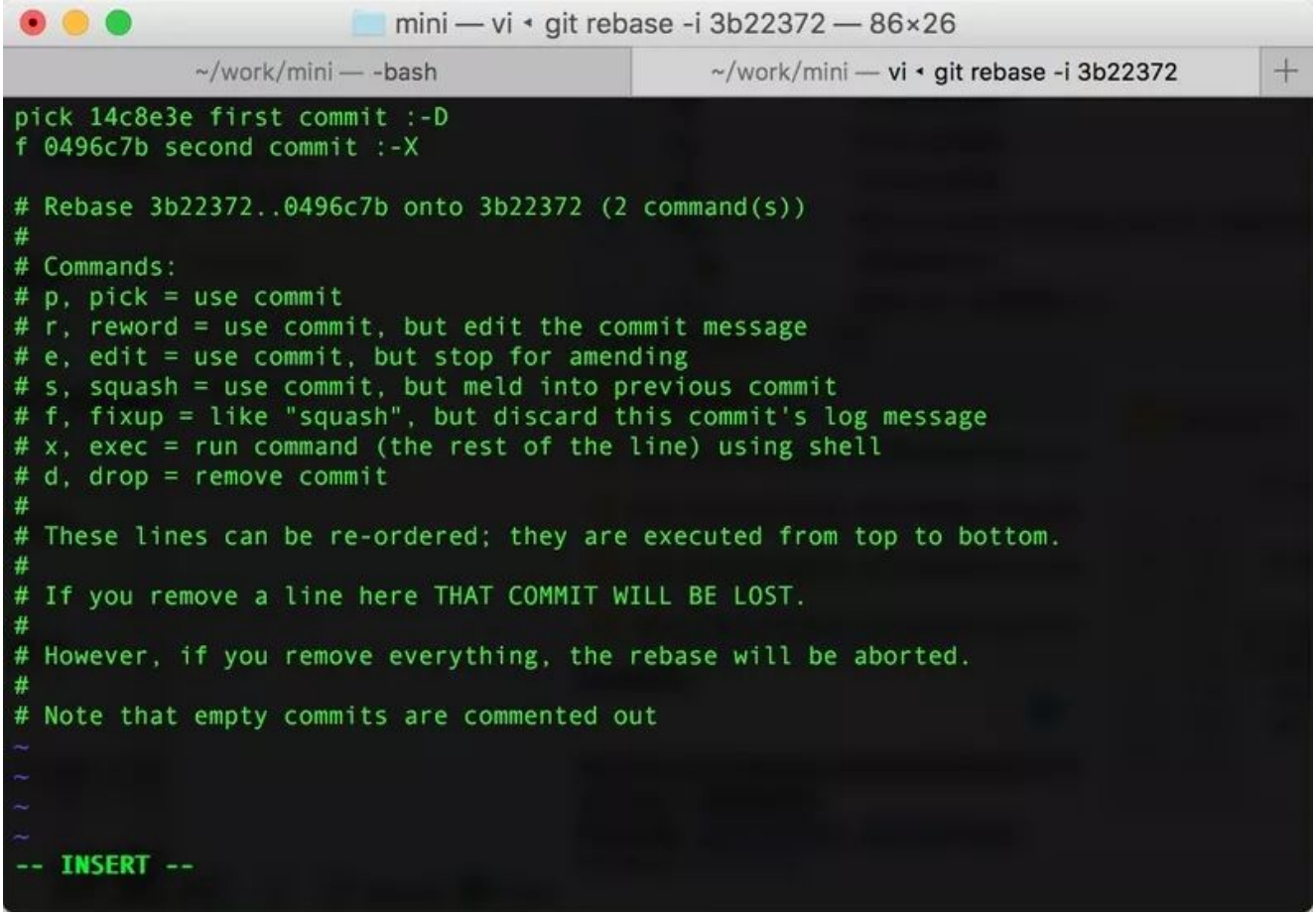
1.1 提交

如何去写一个提交信息，在具体开发工作中主要需要遵守的原则就是「使每次提交都有质量」，只要坚持做到以下几点就 OK 了：

- 提交时的粒度是一个小功能点或者一个 bug fix，这样进行恢复等的操作时能够将「误伤」减到最低；
- 用一句简练的话写在第一行，然后空一行稍微详细阐述该提交所增加或修改的地方；
- 不要每提交一次就推送一次，多积攒几个提交后一次性推送，这样可以避免在进行一次提交后发现代码中还有小错误。

假如已经把代码提交了，对这次提交的内容进行检查时发现里面有个变量单词拼错了或者其他失误，只要还没有推送到远程，就有一个不被他人发觉你的疏忽的补救方法——首先，把失误修正之后提交，可以用与上次提交同样的信息。

然后，终端中执行命令 `git rebase -i [SHA]`，其中 SHA 是上一次提交之前的那次提交的，在这里是 3b22372。



最后，这样就将两次提交的节点合并成一个，甚至能够修改提交信息！

谁说历史不可篡改了？**前提是，想要合并的那几次提交还没有推送到远程！**

1.2 推送

当自己一个人进行开发时，在功能完成之前不要急着创建远程分支。

1.3 拉取

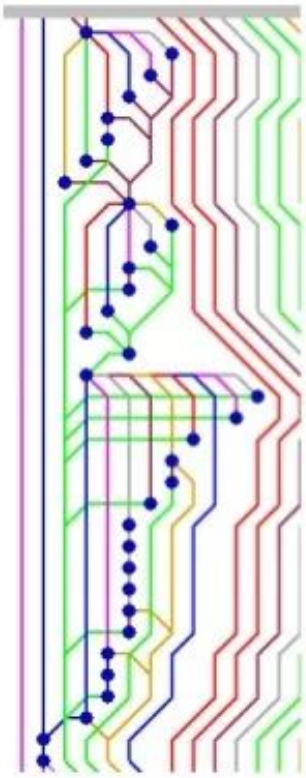
请读张文钿所写的《使用 git rebase 避免無謂的 merge》：https://ihower.tw/blog/archives/3843。

1.4 合并

在将其他分支的代码合并到当前分支时，如果那个分支是当前分支的父分支，为了保持图表的可读性和可追踪性，可以考虑用 git rebase 来代替 git merge；反过来或者不是父子关系的两个分支以及互相已经 git merge 过的分支，就不要采用 git rebase 了，避免出现重复的冲突和提交节点。

2.分支管理

Git 的一大特点就是可以创建很多分支并行开发。正因为它的灵活性，团队中如果没有一个成熟的分支模型的话，那将会是一团糟。



要是谁真把这么乱的提交图表摆在我面前，就给他一个上勾拳！

2.1 分支模型

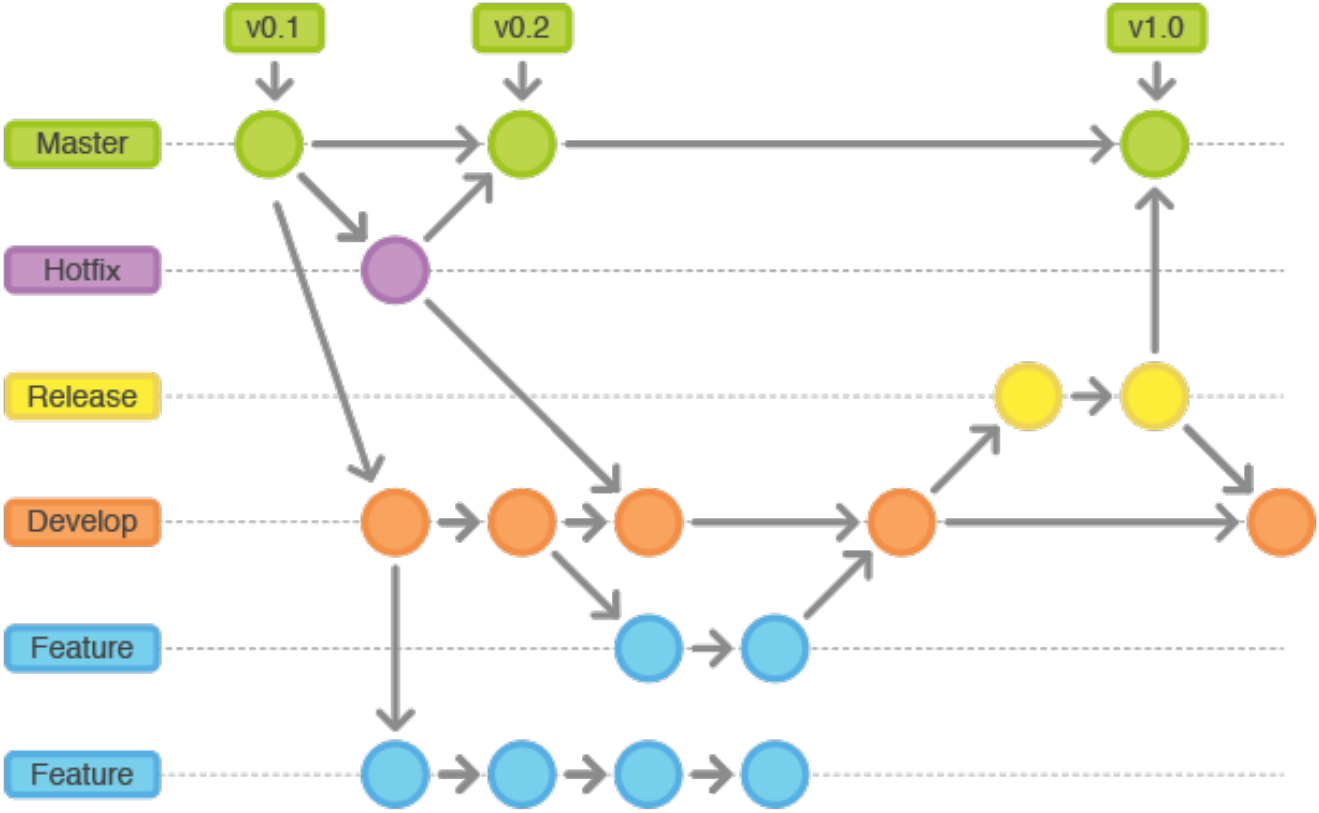
有个很成熟的叫 Git Flow 的分支模型，它能够应对 99% 的场景，剩下的那 1% 留给几乎不存在的极度变态的场景。

需要注意的是，**它只是一个模型，而不是一个工具；你可以用工具去应用这个模型，也可以用最朴实的命令行。所以，重要的是理解概念，不要执着于实行的手段。**

简单说来，Git Flow 就是给原本普普通通的分支赋予了不同的「职责」：

- **master**——最为稳定功能最为完整的随时可发布的代码；
- hotfix——修复线上代码的 bug；
- **develop**——永远是功能最新最全的分支；
- feature——某个功能点正在开发阶段；
- release——发布定期要上线的功能。

看到上面的「master」和「develop」加粗了吧？代表它们是「主要分支」，其他的分支是基于它们派生出来的。主要分支每种类型只能有一个，派生分支每个类型可以同时存在多个。各类型分支之间的关系用一张图来体现就是：



更多信息可参考 xirong 所整理的《Git工作流指南》:https://github.com/xirong/my-git/blob/master/git-workflow-tutorial.md

2.2 工具选择

一直不喜欢「＊＊最好用」这种命题，主观性太强，不会有一个结论。对于工具的选择，我一直都是秉承「哪个能更好地解决问题就用哪个」这个原则。所以，只要不影响到团队，用什么工具都是可以接受的。但根据多数开发人员的素质情况来看，建议使用图形化工具，例如 SourceTree（https://www.sourcetreeapp.com）。如果想用命令行，可以啊！先在心里问下自己：「我 Git 牛逼不？会不会惹麻烦给别人？」

在团队中应用 Git Flow 时，推荐使用 SourceTree 与 GitLab （https://gitlab.com/）配合的形式：

- 用 SourceTree 创建 feature 等分支以及本地的分支合并、删除；
- 用 GitLab 做代码审核和远程的分支合并、删除。

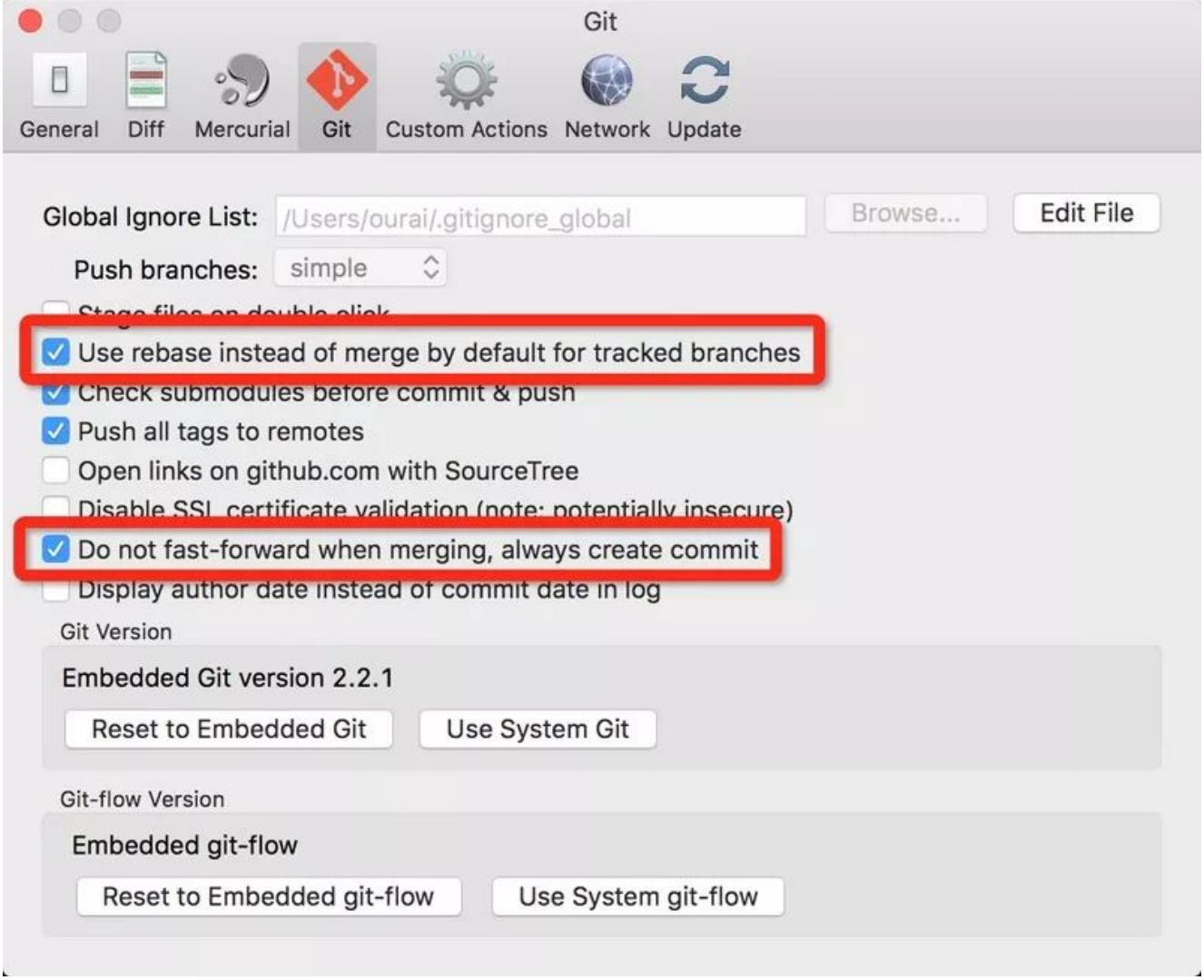
SourceTree 和 GitLab 应该是相辅相成的存在，而不是互相取代。

3.事前准备

为了将一些规范性的东西和 Git Flow 的部分操作自动化处理，要对 SourceTree 和 GitLab 进行一下配置。

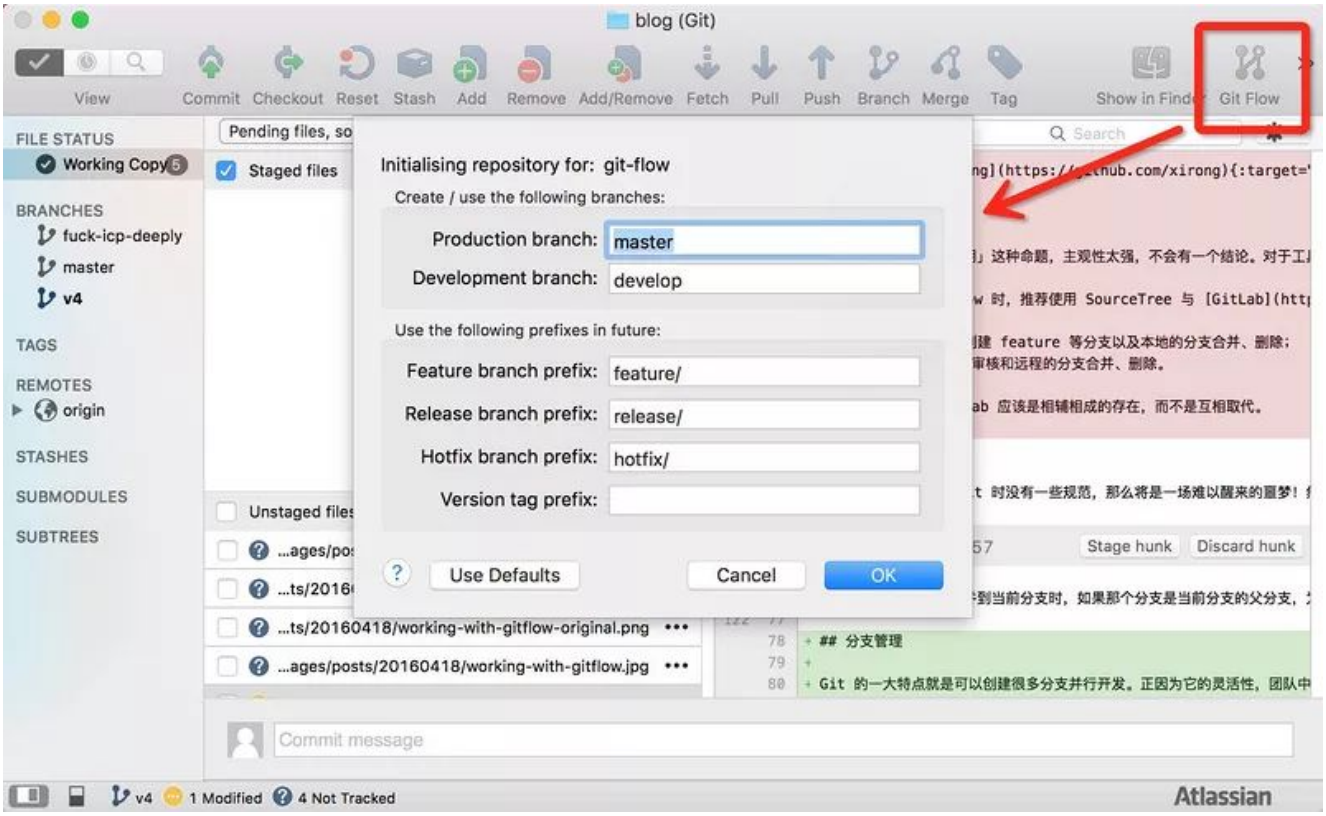
3.1 SourceTree

按下 command + , 调出「Preferences」界面并切换到「Git」标签，勾选「Use rebase instead of merge by default for tracked branches」和「Do not fast-forward when merging, always create commit」。



这样设置之后，在点「Pull」按钮拉取代码时会自动执行 git pull --rebase；并且，每次合并时会自动创建新的包含分支信息的提交节点。

接下来，点击工具栏中的「Git Flow」按钮将相关的流程自动化。如果没有特殊需求，直接按下对话框中的「OK」就好了。初始化完成后会自动切换到 develop 分支。



这下再点「Git Flow」按钮所弹出的对话框就是选择创建分支类型的了。

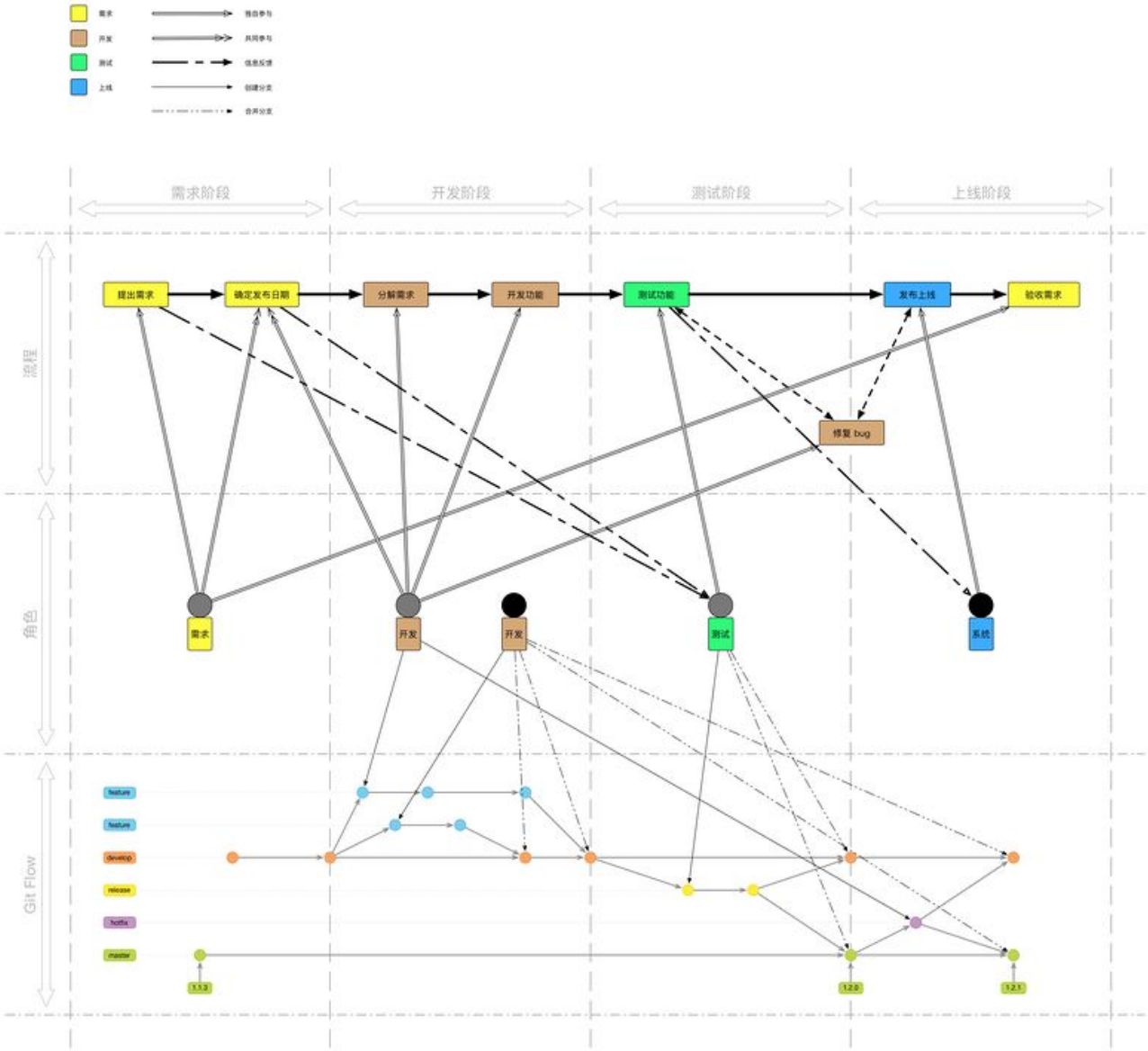
3.2 GitLab

在创建项目仓库后一定要把主要分支，也就是 master 和 develop 给保护起来。为它们设置权限，只有项目负责人可以进行推送和删除等操作。

被保护的分支在列表中会有特殊的标记进行区分。

4.开发流程

在引入 Git Flow 之后，所有工作都要围绕着它来展开，将原本的流程与之结合形成「基于 Git Flow 的开发流程」。



4.1 开发功能

在确定发布日期之后，将需要完成的内容细分一下分配出去，负责某个功能的开发人员利用 SourceTree 所提供的 Git Flow 工具创建一个对应的 feature 分支。如果是多人配合的话，创建分支并做一些初始化工作之后就推送创建远程分支；否则，直到功能开发完毕要合并进 develop 前，不要创建远程分支。

功能开发完并自测之后，先切换到 develop 分支将最新的代码拉取下来，再切换回自己负责的 feature 分支把 develop 分支的代码合并进来。合并方式参照上文中的「合并」，如果有冲突则自己和配合的人一起解决。

然后，到 GitLab 上的项目首页创建合并请求（merge request）。

「来源分支」选择要被合并的 feature 分支且「目标分支」选择 develop 分支后点击「比较分支」按钮，在出现的表单中将处理人指派为项目负责人。

项目负责人在收到合并请求时，应该先做下代码审核看看有没有明显的严重的错误；有问题就找负责开发的人去修改，没有就接受请求并删除对应的 feature 分支。

在将某次发布的所需功能全部开发完成时，就可以交付测试了。

4.2 测试功能

负责测试的人创建一个 release 分支部署到测试环境进行测试；若发现了 bug，相应的开发人员就在 release 分支上或者基于 release 分支创建一个分支进行修复。

4.3 发布上线

当确保某次发布的功能可以发布时，负责发布的人将 release 分支合并进 master 和 develop 并打上 tag，然后打包发布到线上环境。

建议打 tag 时在信息中详细描述这次发布的内容，如：添加了哪些功能，修复了什么问题。

4.4 修复问题

当发现线上环境的代码有小问题或者做些文案修改时，相关开发人员就在本地创建 hotfix 分支进行修改，具体操作参考「开发功能」。

如果是相当严重的问题，可能就得回滚到上一个 tag 的版本了。

5.额外说明

这里所提到的事情，虽非必需，但知道之后却会如虎添翼。

5.1 分支命名

除了主要分支的名字是固定的之外，派生分支是需要自己命名的，这里就要有个命名规范了。强烈推荐用如下形式：

- feature——按照功能点（而不是需求）命名；
- release——用发布时间命名，可以加上适当的前缀；
- hotfix——GitLab 的 issue 编号或 bug 性质等。

另外还有 tag，用语义化的版本号（http://semver.org/lang/zh-CN/）命名。

5.2 发布日期

发布频率是影响开发人员与测试人员的新陈代谢和心情的重要因素之一，频繁无规律的发布会导致内分泌失调、情绪暴躁，致使爆粗口、砸电脑等状况出现。所以，确保一个固定的发布周期至关重要！

在有一波或几波需求来临之时，想挡掉是不太可能的，但可以在评审时将它（们）分期，在某个发布日之前只做一部分。这是必须要控制住的！不然任由着需求方说「这个今天一定要上」「那个明天急着用」的话，技术人员就等着进医院吧！

推荐阅读

1. 仅推荐一个置顶的程序员公众号
2. 寓教于乐，用玩游戏的方式学习 Git
3. 在浏览器输入 URL 回车之后发生了什么
4. 在阿里干了 5 年，面试个小公司挂了…

如何在 IntelliJ IDEA 中使用 Git

Java后端 2019-12-26

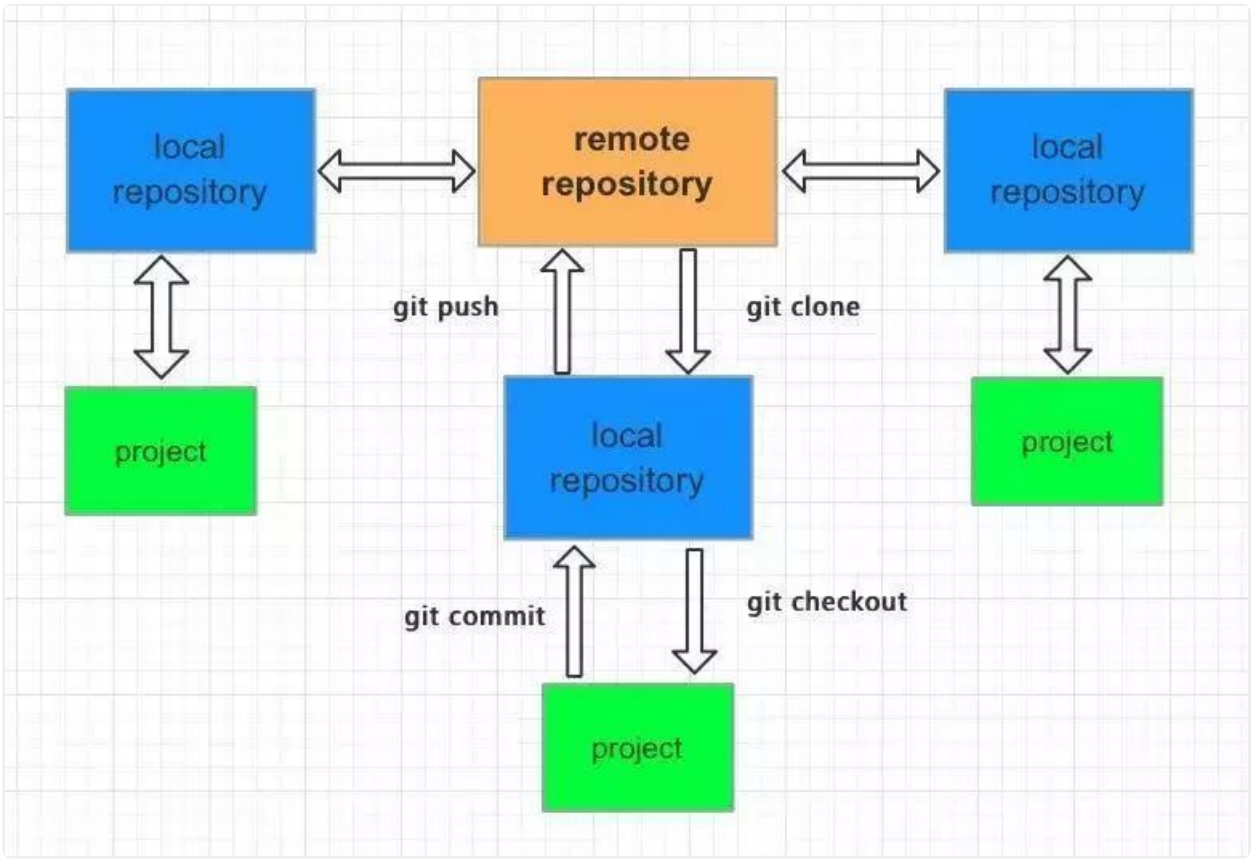
作者：J'KYO

来源：https://urlify.cn/2YjiEj

1、Git简介

Git是目前流行的分布式版本管理系统。它拥有两套版本库，本地库和远程库，在不进行合并和删除之类的操作时这两套版本库互不影响。也因此其近乎所有的操作都是本地执行，所以在断网的情况下任然可以提交代码，切换分支。Git又使用了SHA-1哈希算法确保了在文件传输时变得不完整、磁盘损坏导致数据丢失时能立即察觉到。

Git的基本工作流程：



- git clone：将远程的Master分支代码克隆到本地仓库
- git checkout：切出分支出来开发
- git add：将文件加入库跟踪区
- git commit：将库跟踪区改变的代码提交到本地代码库中
- git push： 将本地仓库中的代码提交到远程仓库

git 分支

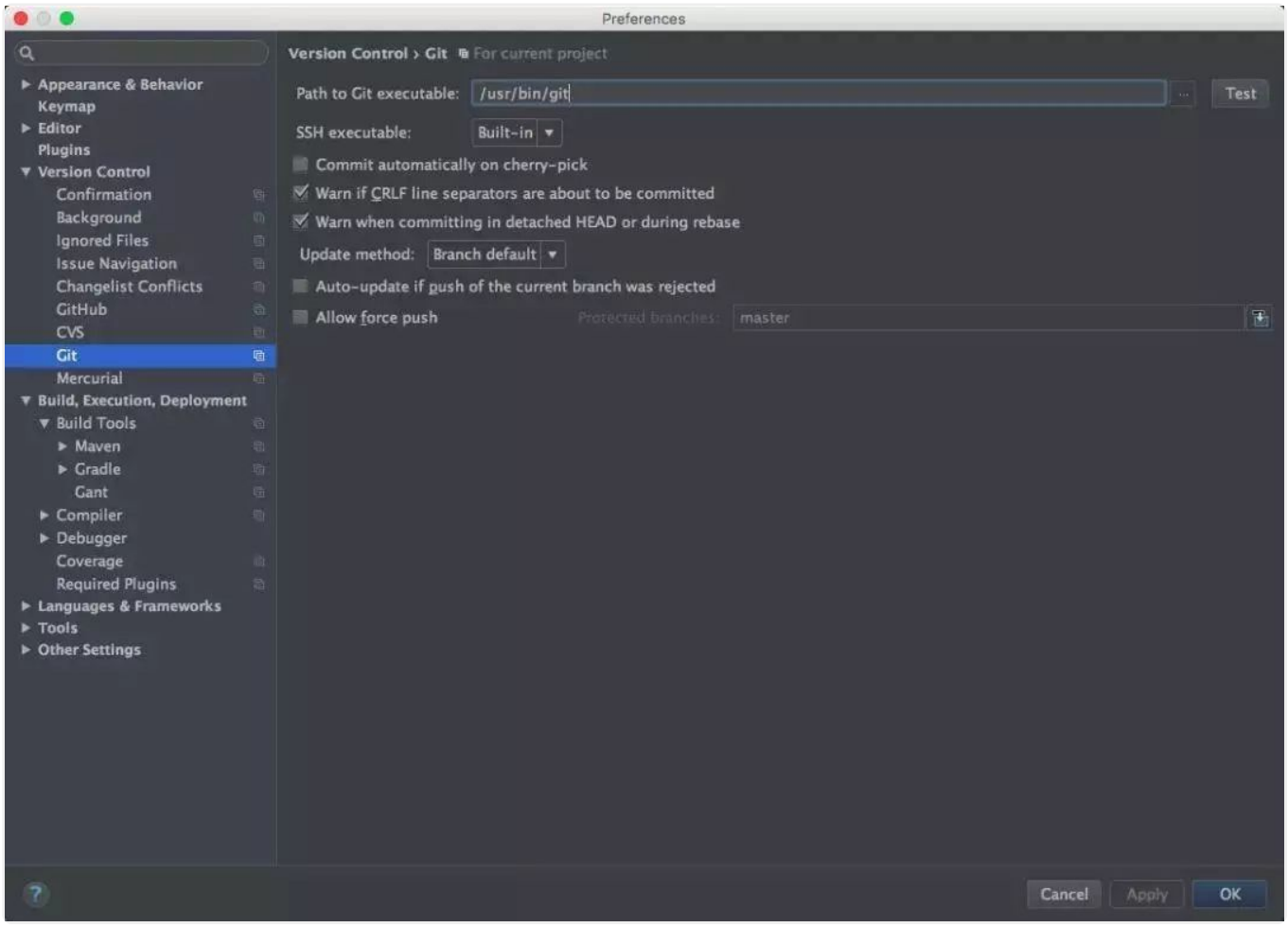
- 主分支
- master分支：存放随时可供生产环境中的部署的代码
- develop分支：存放当前最新开发成果的分支，当代码足够稳定时可以合并到master分支上去。

- 辅助分支
- feature分支：开发新功能使用，最终合并到develop分支或抛弃掉
- release分支：做小的缺陷修正、准备发布版本所需的各项说明信息
- hotfix分支：代码的紧急修复工作

2、Git在IntelliJ IDEA下的使用

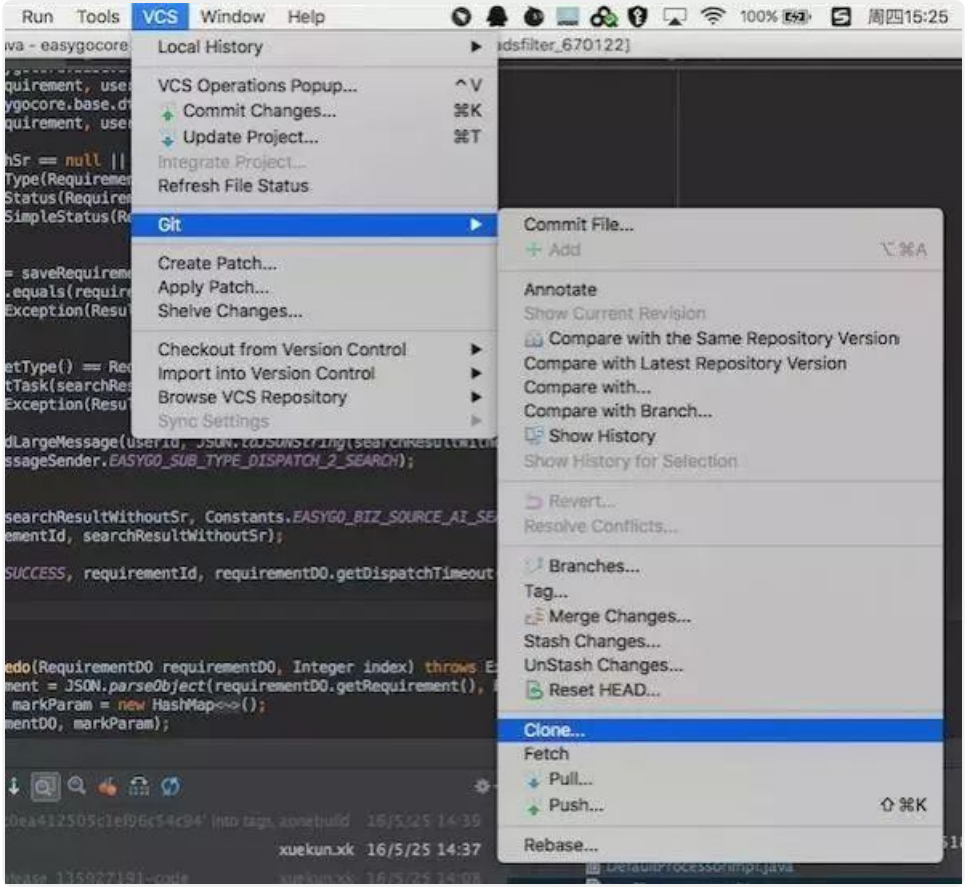
2.1、IntelliJ IDEA下配置Git

本地安装好git，并配置合理的SSH key，具体看这里 IntelliJ IDEA->Performance->Version Control->git 将自己安装git的可执行文件路径填入Path to Git executable，点击 Test测试一下

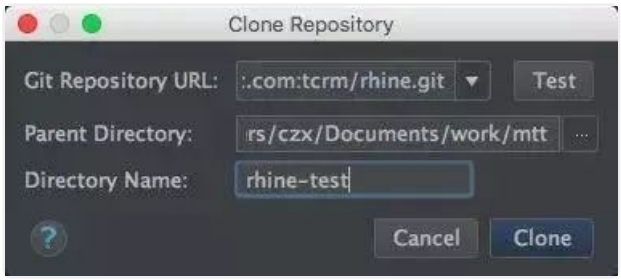


2.2、git clone

VCS->Git->Clone

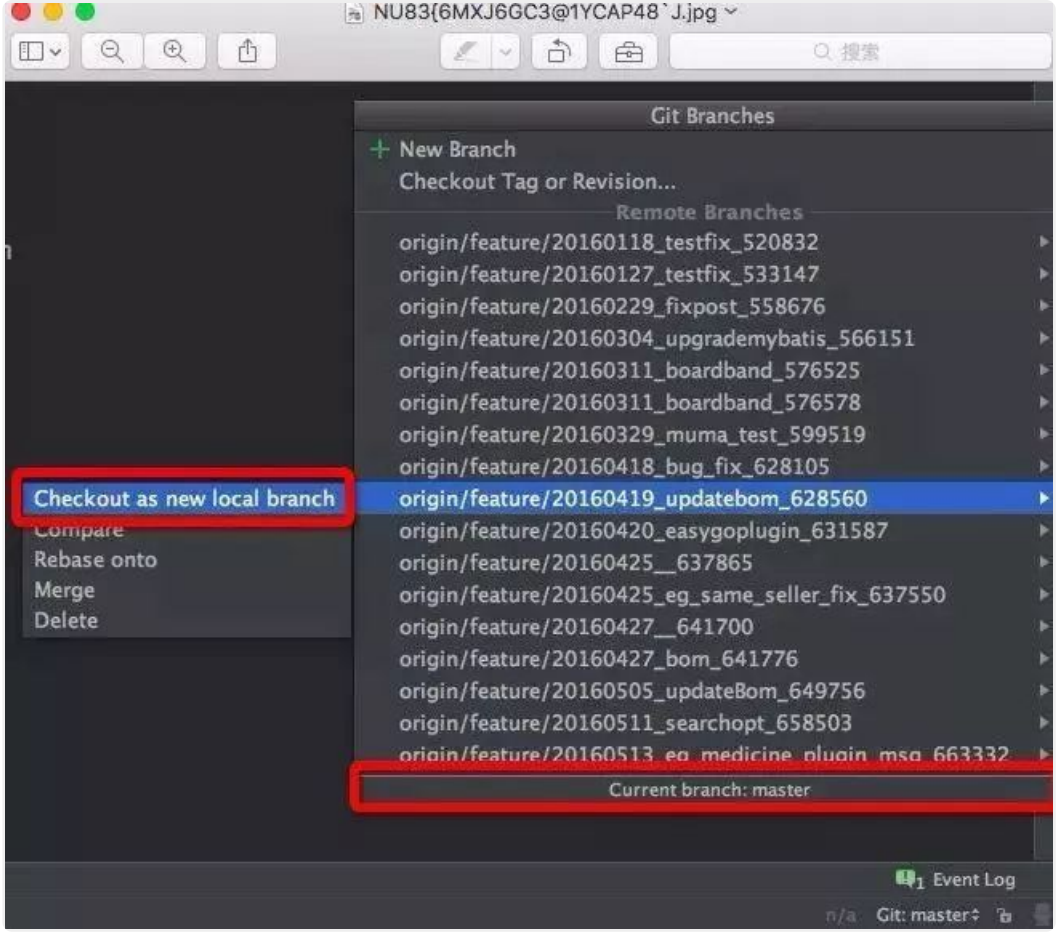


输入你的远程仓库地址,点击测试一下地址是否正确

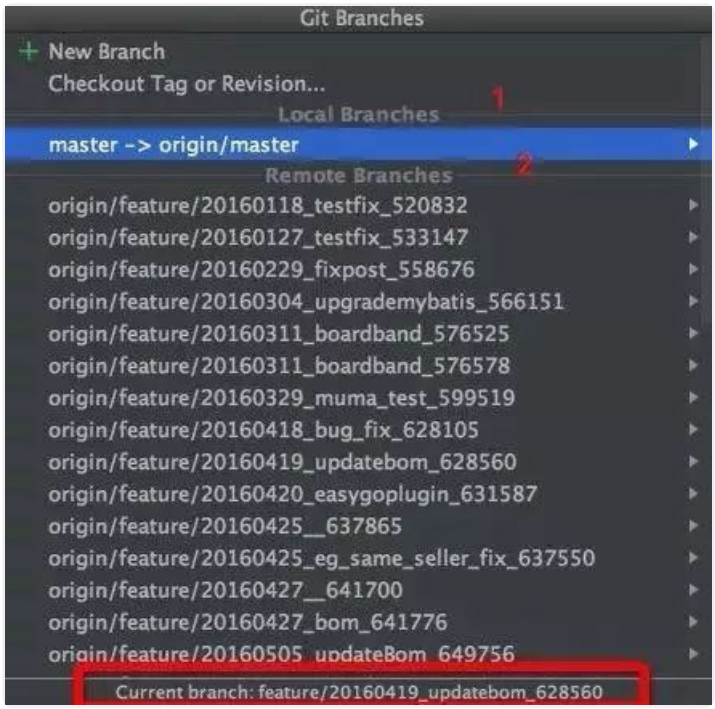


2.3、git checkout

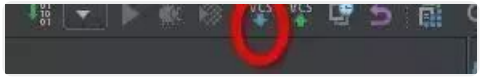
在IntelliJ IDEA右下角有一个git的分支管理，点击。选择自己需要的分支，checkout出来



checkout出来，会在底端显示当前的分支。其中1显示的为本地仓库中的版本，2为远程仓库中的版本

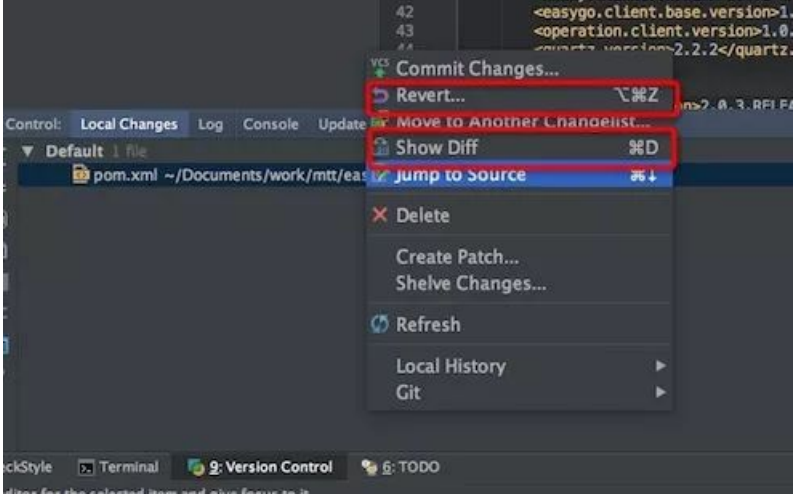


点击IDE的右上角的向下箭头的VCS，将分支的变更同步到本地



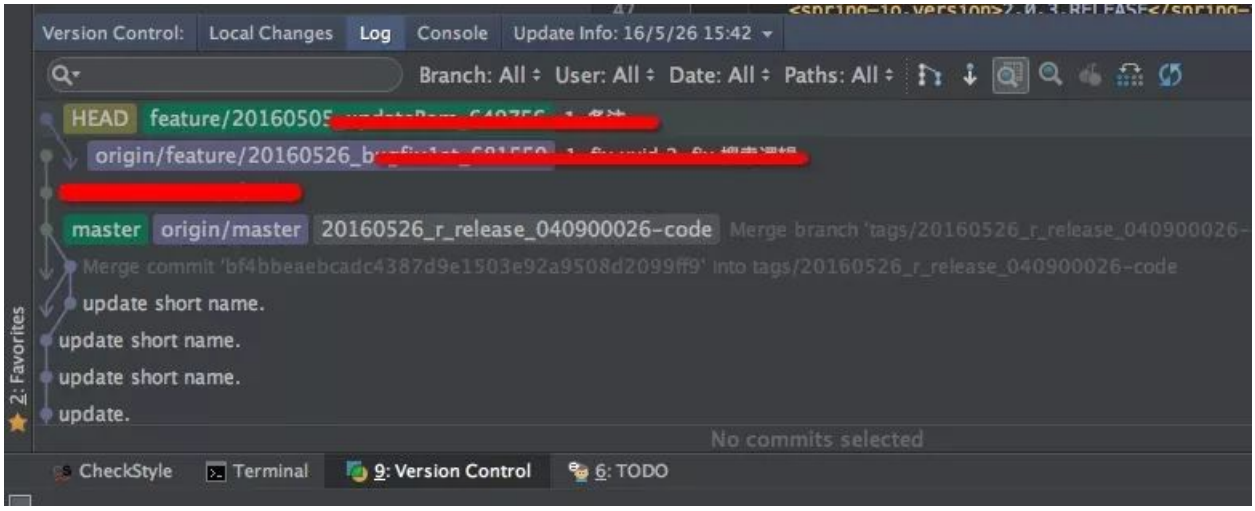
2.4、git diff

在local changes 中选中要比对的文件，右键选择show diff 便可以查看文件的变动。或者选择Revert放弃文件的改动



2.5、git log

在Version Control下选择Log，可以查看提交历史

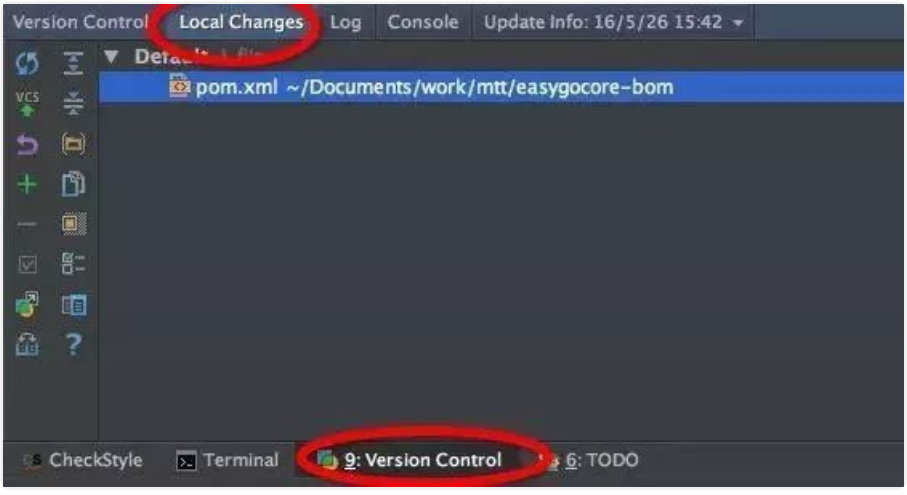


2.6、git commit

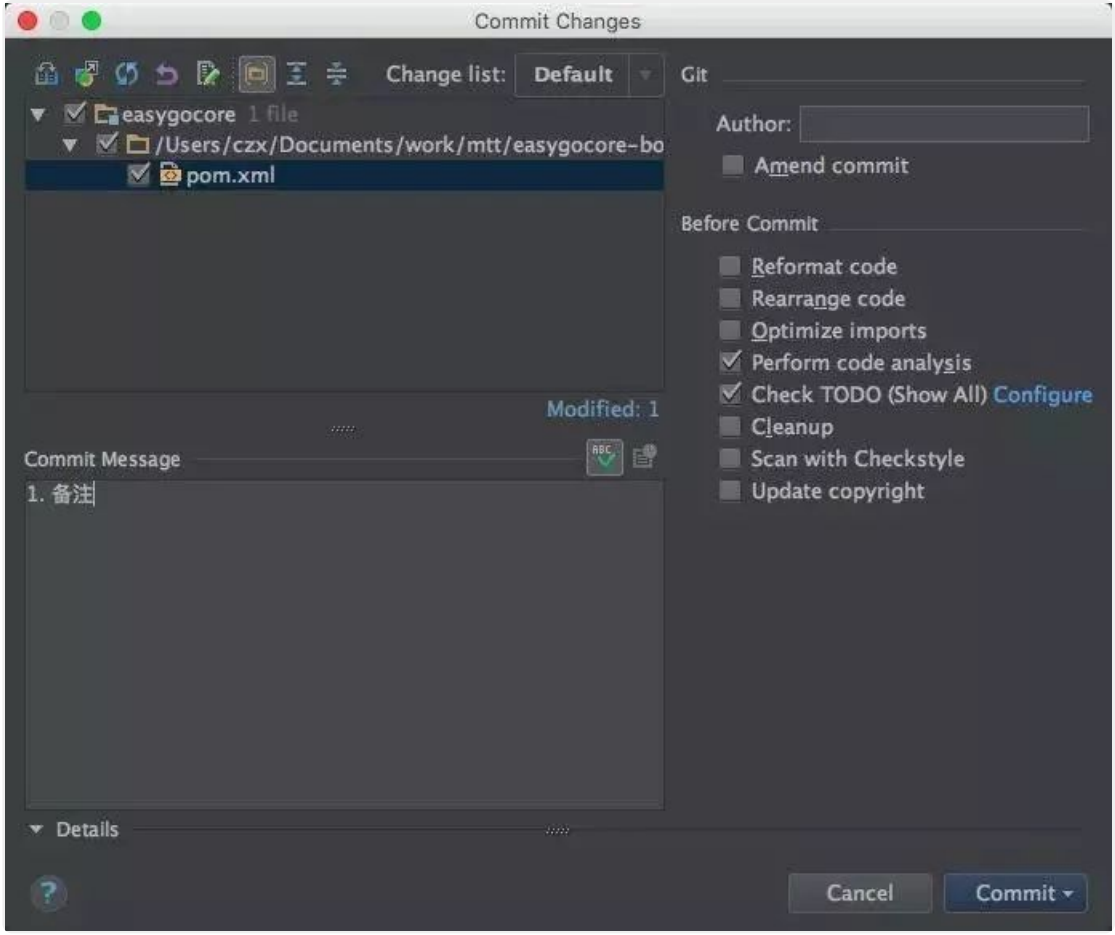
默认导入的工程已经git add加入库跟踪区了

随便修改一下pom.xml文件，其修改的文件会显示在Version Control中的local changes下

欢迎关注公众号：Java后端

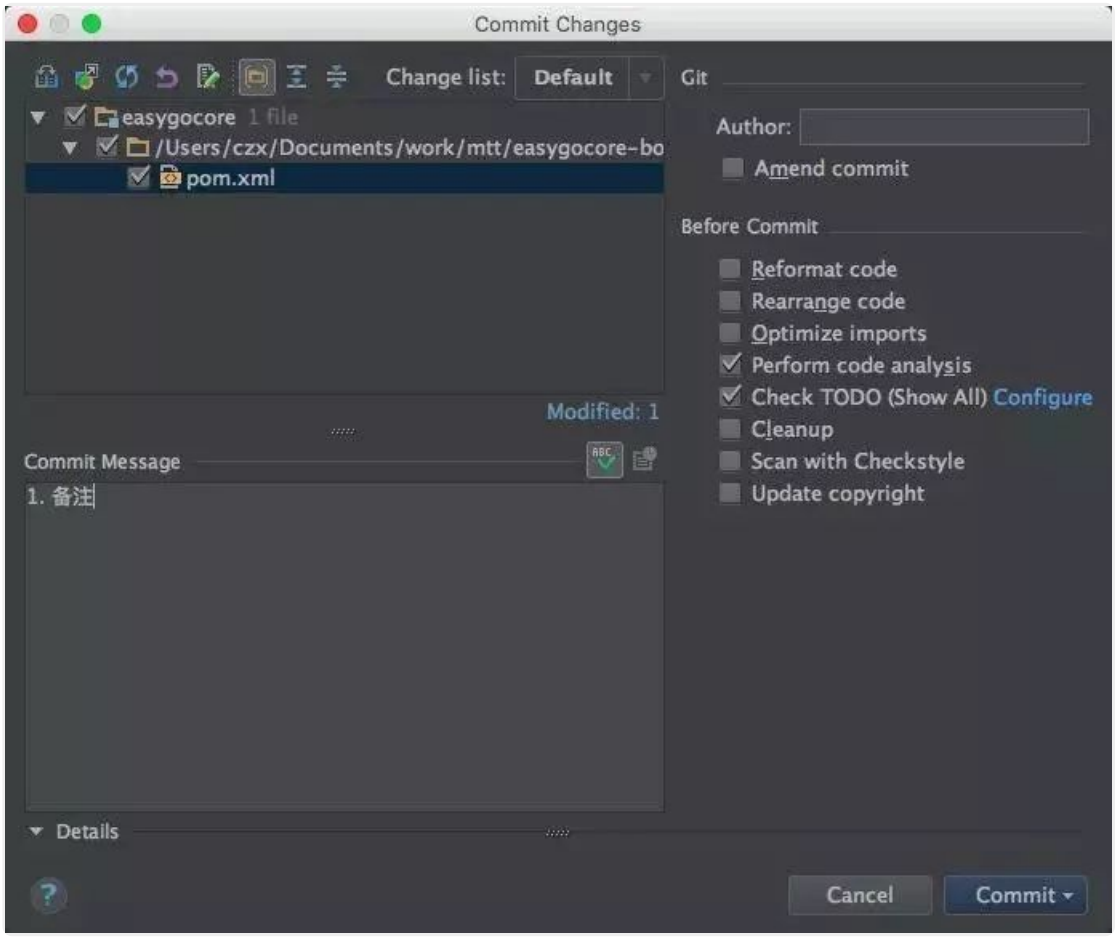


点击IDE右上角的向上箭头的VCS， git commit, 写上日志提交到本地代码库中



2.7、git push

VCS->Git->Push 将本地代码提交到远程仓库



2.8、在Idea命令行使用git

mac下同时按alt+F12,进入idea命令行

常见的命令：

- clone项目 git clone xxxxxx
- 检查项目状态 git status
- 切换分支并和远程的分支关联 git checkout -b xxx -t origin/xxx
- 拉最新更新 git pull
- 提交更新 git commit -am "备注"
- 合并分支到当前分支，首先切换到需要被合并的分支 git checkout xxx, 再合并 git merge yyyy
- 提交 git push

- END -

推荐阅读

1. Spring 的 Bean 生命周期
2. 12306 又崩溃，买张车票怎么就这么难
3. 发布没有答案的面试题，都是耍流氓
4. 什么是一致性 Hash 算法？
5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

学到了！用 Git 和 Github 提高效率的 10 个技巧

张伯函 Java后端 2019-09-25

上一篇 | 刷屏了！请给我一面国旗@微信官方

作　者 | 张伯函

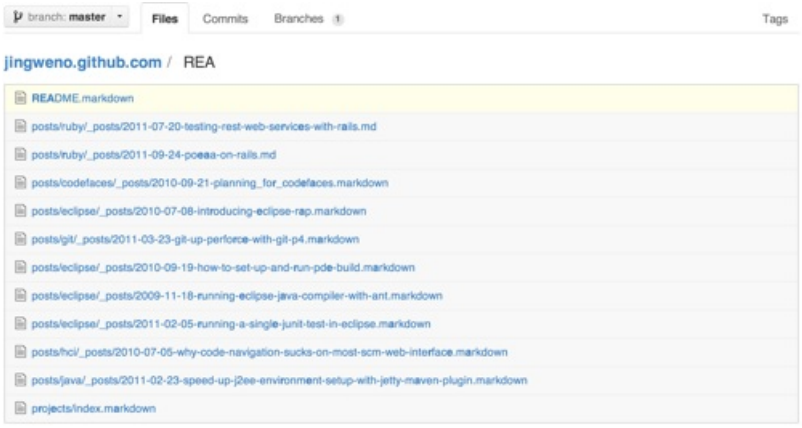
链　接 | segmentfault.com/a/1190000003830252

Git 和 GitHub都是非常强大的工具。即使你已经使用他们很长时间，你也很有可能不知道每个细节。我整理了Git和GitHub可能提高日常效率的10个常用技巧。

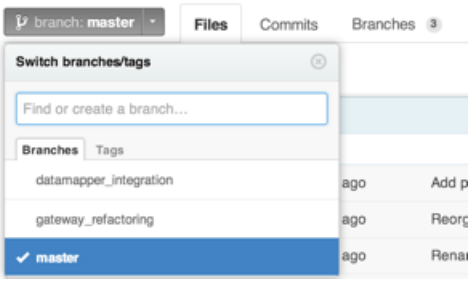
GitHub

快捷键: t 和 w

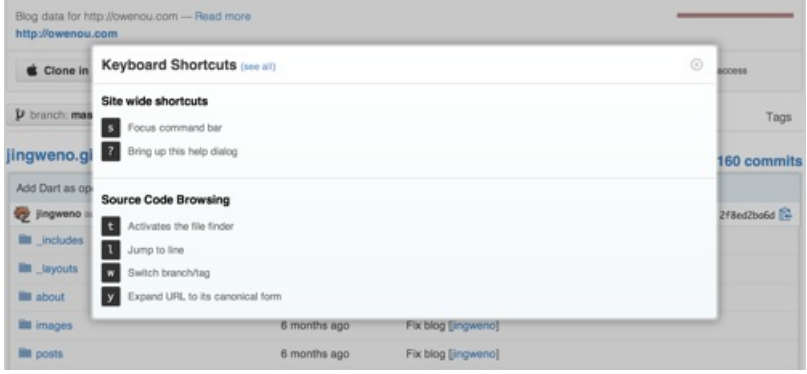
在你的源码浏览页面，按t可以快速进入模糊文件名搜索模式：



在你仓库主页，按w可以快速进行分支过滤：



在任意GitHub页面中，按?展示当前页面可用的快捷键：



忽略空格: ?w=1

在任意的diff URL添加**?w=1**用来整理缩进:

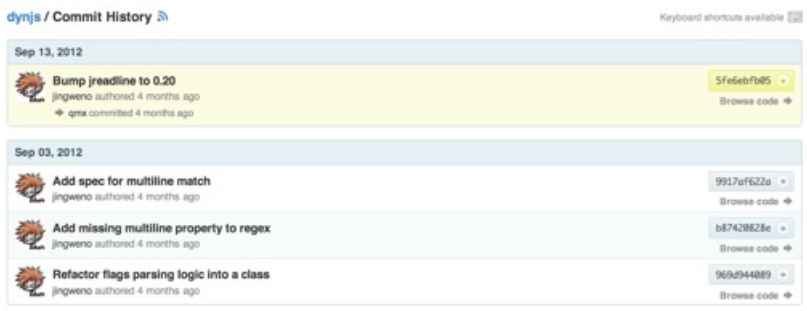
按范围过滤提交记录: master@{time}..master

你可以创建一个对比页面通过使用URL github.com/user/repo/compare/{range}。范围(range)可以是两个SHA例如sha1…sha2或者两个分支名称例如master…my-branch。范围同时也非常智能的支持使用时间作为关注点。

你可以通过master@{1.day.ago}…master过滤从昨天开始的提交。例如：链接，https://github.com/rails/rails/compare/master@{1.day.ago}…master显示Rails项目中全部昨天开始的提交记录和变化：

按作者过滤提交记录: ?author=github_handle

你可以通过在对比页面URL中增加?author=github_handle来按作者过滤提交记录。例如：链接https://github.com/dynjs/dynjs/commits/master?author=jingweno显示jingweno对Dynjs 的提交记录：



.diff 和 .patch

在比较页面、合并请求页面或者评论页面的URL后增加.diff或者.patch，可以得到diff或者patch的文本格式。例如：链接https://github.com/rails/rails/compare/master@{1.day.ago}…master.patch显示Rails项目中全部昨天开始的提交记录和变化的文本格式：

```
diff --git a/actionpack/test/template/form_helper_test.rb b/actionpack/test/template/form_helper_test.rb
index 247068d..14518f5 100644
--- a/actionpack/test/template/form_helper_test.rb
+++ b/actionpack/test/template/form_helper_test.rb
@@ -2163,14 +2163,14 @@ def test_nested_fields_for_with_child_index_option_override_on_a_nested_attribut
     assert_dom_equal expected, output_buffer
   end

-  class FakeAssociatonProxy
+  class FakeAssociationProxy
     def to_ary
       [1, 2, 3]
     end
   end

   def test_nested_fields_for_with_child_index_option_override_on_a_nested_attributes_collection_association_with_proxy
-    @post.comments = FakeAssociatonProxy.new
+    @post.comments = FakeAssociationProxy.new

     form_for(@post) do |f|
       concat f.fields_for(:comments, Comment.new(321), :child_index => 'abc') { |cf|
diff --git a/guides/source/documents.yaml b/guides/source/documents.yaml
index 13f982d..c73bbeb 100644
--- a/guides/source/documents.yaml
+++ b/guides/source/documents.yaml
@@ -102,7 +102,6 @@
     description: This guide documents the asset pipeline.
   -
     name: Working with JavaScript in Rails
-    work_in_progress: true
     url: working_with_javascript_in_rails.html
     description: This guide covers the built-in Ajax/JavaScript functionality of Rails.
   -
```

邮件回复

你可以直接在收到的GitHub通知邮件进行评论，不必在网站页面中评论。GitHub会正确的处理你的评论：

链接行

在文件展示页面，点击某行或者通过按SHIFT选择多行，URL会有相应的改变。如果你要给你的队友分享一段代码是非常方便的：

关注用户

在合并请求、问题或者任何评论中中提到用户会使用户关注全部的后续通知：

自动链接

在合并请求、问题、或者任何评论中，sha和问题码(例如：#1)会被自动链接。并且，你也可以链接其它仓库的sha或者问题码，格式：user/repo@sha1或者user/repo#1。下面是一个评论中通过sha自动链接的例子：

hub

Hub 是 GitHub的命令行。它提供了Git和Github之间的集成。一个最有用的命令就是在命令行输入hub pull-request创建pull request。详见readme.

Git

```
git log -p FILE
```

查看README.md的修改历史，例如：

```
git log -p README.md
git log -S'PATTERN'
```

例如，搜索修改符合stupid的历史：

```
git log -S'stupid'
git add -p
```

交互式的保存和取消保存变化，使用：

```
git add -p
git rm –cached FILE
```

这个命令只删除远程文件，例如：

```
git rm --cached database.yml
```

删除database.yml被保存的记录，但是不影响本地文件。这对删除已经推送过的忽略文件记录而且不影响本地文件是非常的方便的。

```
git log ..BRANCH
```

这个命令返回某个非HEAD分支的提交记录。假如你在一个功能分支，输入：

```
 git log ..master
```

返回全部master分支的历史记录，包括未被合并到当前分支的提交记录。

```
git branch –merged & git branch –no-merge
d
```

这个命令返回已合并分支列表或未合并的分支列表。这个命令对合并前检查非常有用。例如，在一个功能分支，输入

```
git branch --no-merged
```

返回未合并到该分支的分支列表。

```
git branch –contains SHA
```

返回包含某个指定sha的分支列表。例如：

```
git branch --contains 2f8e2b
```

显示全部包含提交2f832b的分支。这个命令对于验证git cherry-pick完成非常有帮助。

```
git status -s
```

返回一个简单版的git status。我设置这个命令为默认git status来减少噪音。

```
git reflog
```

显示你在本地已完成的操作列表。

```
git shortlog -sn
```

显示提交记录的参与者列表。和GitHub的参与者列表相同。

Summary

Git是一个设计良好的工具。了解它可以直接让你更有效率并成为一个更有才华的程序员。GitHub，在另一个方面，在Git基础上提供便利的团队合作特性。有能力使用GitHub也会提高你日常效率。

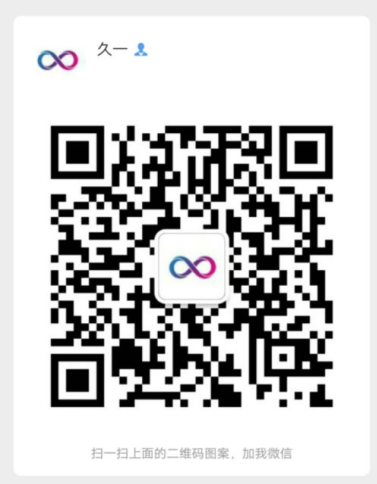
为了更好的加深你对的Git和Github了解，我推荐一些资料：

- ProGit, 最好的Git指南
- Advanced Git
- Git and GitHub Secrets

-END-

如果看到这里，说明你喜欢这篇文章，帮忙**转发**一下吧，感谢。微信搜索「web_resource」，关注后回复「进群」即可进入无广告交流群。

↓扫描二维码进群↓



推荐阅读

1. Java后端优质文章整理
2. 全网最牛掰的12306抢票神器
3. 面试官：说一说 Spring Boot 自动配置原理
4. 在浏览器输入 URL 回车之后发生了什么？
5. 接私活必备的 10 个开源项目

Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看

阅读原文

寓教于乐，用玩游戏的方式学习 Git

少数派 Java后端 2019-09-13

作者 | mozlingyu
来源 | 少数派

用游戏的方式来学习，是一种有趣而高效的方式。

从刚接触电脑时的打字练习软件 金山打字通，到程序猿写代码的利器 Vim 都有小游戏（金山打字通游戏、VIM Adventures）来帮助我们入门。

当你的目标从**掌握技能**转变为**打通游戏**之后，学习本身也就不再痛苦。在完成每一关的过程中，都能增加我们的技能熟练度和成就感。

而 Learn Git Branching，就是一个用小游戏带你入门的 Git 的网站。

为什么要学习 Git

Git 是一种分布式的版本管理系统，作用和网盘有点类似，但是功能性和灵活性都更强大。

如果你是一个计算机专业的学生或从业者，Git 的重要性不言而喻；但是对普通人而言，Git 也有**备份数据、保存历史记录**等重要作用。

不怕断电、断网

数据无价，但很多时候我们无法保证自己的电脑不出问题。有很多时候我们会怀念上一个小时、前天或不久以前自己写的那些文字、画过的那些图，如何完好的保存数据是个永远的话题。将工作保存在云端也许会增加我们内心的安全感，这也是很多人会用网盘进行备份的原因。

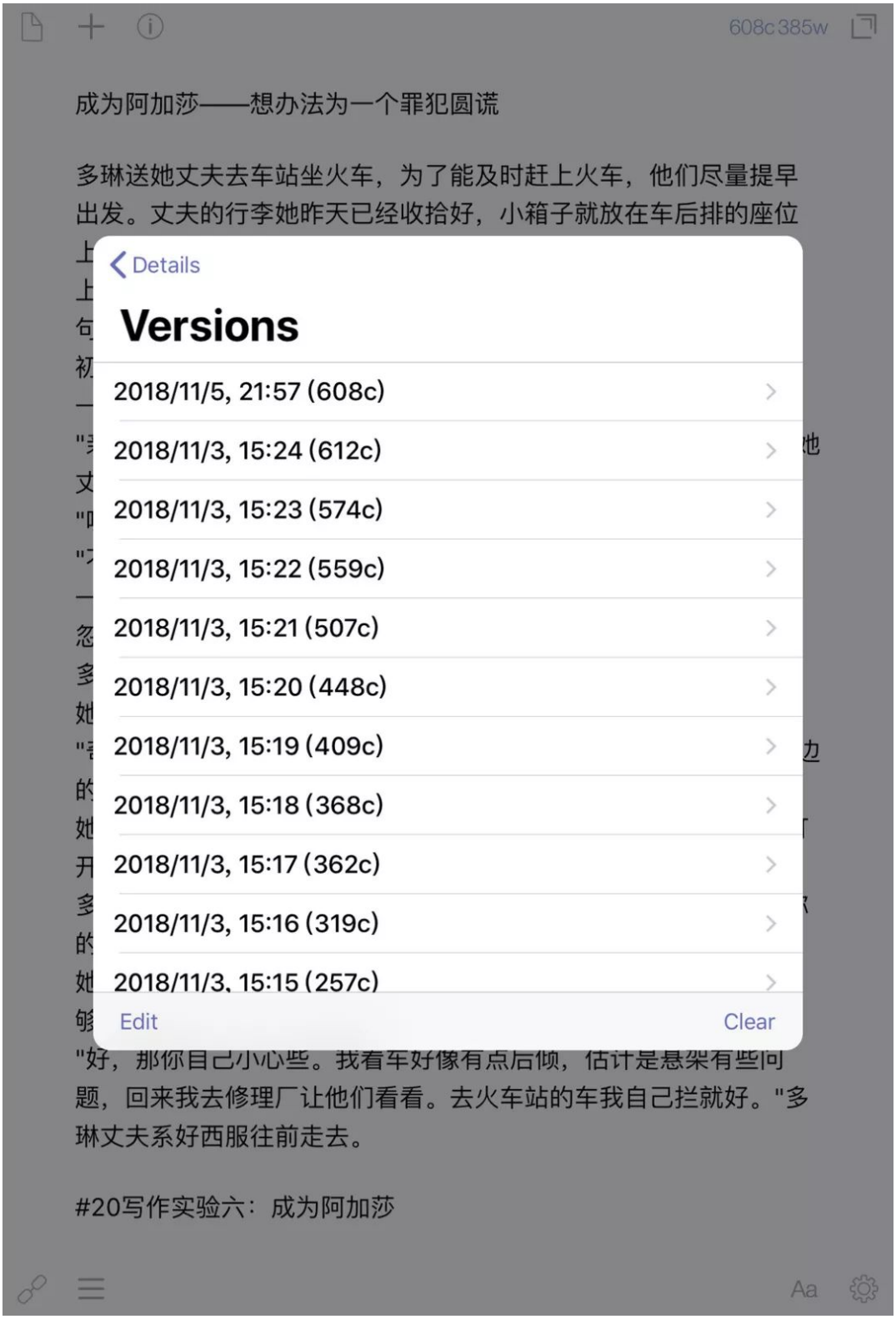
Git 的「本地提交」较好地解决了这个问题，它的工作方式大概是这样：**即使没网络，我们也可以先工作，等到连上网络后再打包上传**。每个文档的名字都写有主要的工作（修改）内容，而且我们还能清楚地看到文档之间的区别和改动，Git 还会自动把历史纪录保存下来。

对于越来越多开始从事移动办公、远程工作的人来说，由于网络环境不稳定，即使他不编程，也能从 Git 中受益。

严肃写作的一个个节点

iOS 上的知名笔记工具 Drafts 会在你写作时，每隔一分钟帮你保存一份当前文档的「快照」，这样就可以方便地找回之前的某个写作版本了。你觉得不满意而删除的一段文字，总可以在历史版本中找回，很贴心。

保存写作的历史版本其实就是建立写作节点的过程。而这，也是 Git 所擅长的。



对于论文、书籍这些严肃的写作内容来说，我们在写作的时候不会频繁的更改。深思熟虑之后，在一个节点时，我们有必要保存一下这一阶段的成果。同时，我们在对以前工作进行修改之后，也希望能便利地显示和以往的不同。

Git 最原本的使用方式

Git 是一个分布式版本控制软件，于 2005 年以 GPL 发布。它最初是为更好地管理 Linux内核 开发而设计。它不需要服务器端软件就可以运作版本控制，使得源代码的发布和交流极其方便。

GitHub 是通过 Git 进行版本控制的源代码托管服务，我们每个人都可以把自己的代码托管在上面。同时，也可以看看别人写的代码，相互交流，极大方便了软件项目的多人协作开发，也推动了开源软件社区的壮大。

GitHub 所具有的社交性让很多人戏称为最大的同性交友网站（男性用户占绝大多数）。如果你愿意，可以把别人的代码下载到本地随意修改，放心这不会影响他人。

总之，在 GitHub 上，大家都用同一种语言进行代码仓库的操作，那就是本文的主角：Git。

如果你想开始学编程，一定想记录一下自己成长的路径。想必也一定会去 GitHub 看一看，这时学一下 Git 不是顺理成章的吗？

怎么学习 Git

曾经我在好奇心的驱使下去学习最浅显易懂的 Git 教程《廖雪峰的 Git 教程》。

但经过一段时间之后，我发现自己仅仅会使用 `git clone`，`git add`，`git commit`，`git push` 等简单语句。之后多次查看 `git rebase` 的用法，也一直没有理解。

我们来看看最基本的几条命令有什么作用：

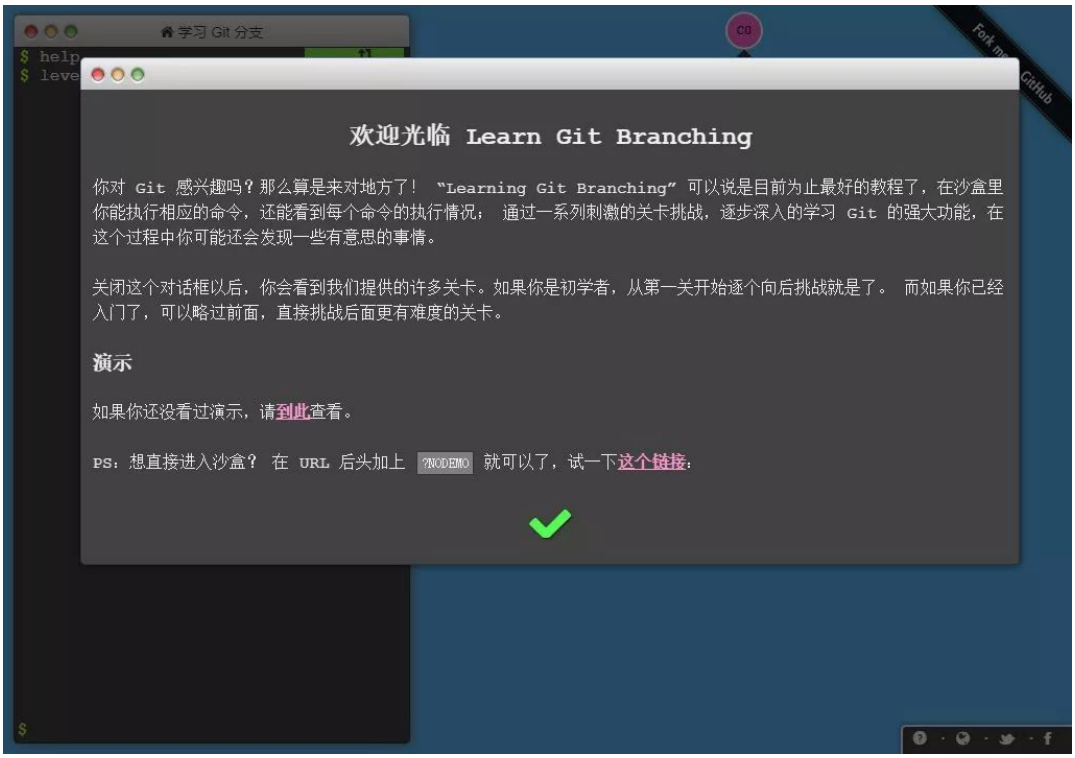
> git clone 克隆：下载远程代码仓库到本地；
>
> git add 添加：添加文件、修改后的文件到暂存区；
>
> git commit 提交：建立本地仓库的工作节点；
>
> git push 推送：将本地仓库推送到远程代码托管服务、网站。

就像很多次把学过的知识还给老师的经历，这次只记住几个简单命令的过程依然没让我失望。这些命令是我平时使用最多的，所以记得最牢，而其他命令只能用到时再去查了。**没有经过实践的知识好像无法停留在脑子里。**

前几天在 twitter 上闲逛，发现了一个可以交互式学习 Git 的网站 Learn Git Branching。尝试了一下发现效果真的很棒，有一种在玩编程游戏的感觉。

回想啃着玩 Human Resource Machine 的经历，证明**不在于知识本身多艰深，只要学习曲线不陡峭，咱都能爬上去。**

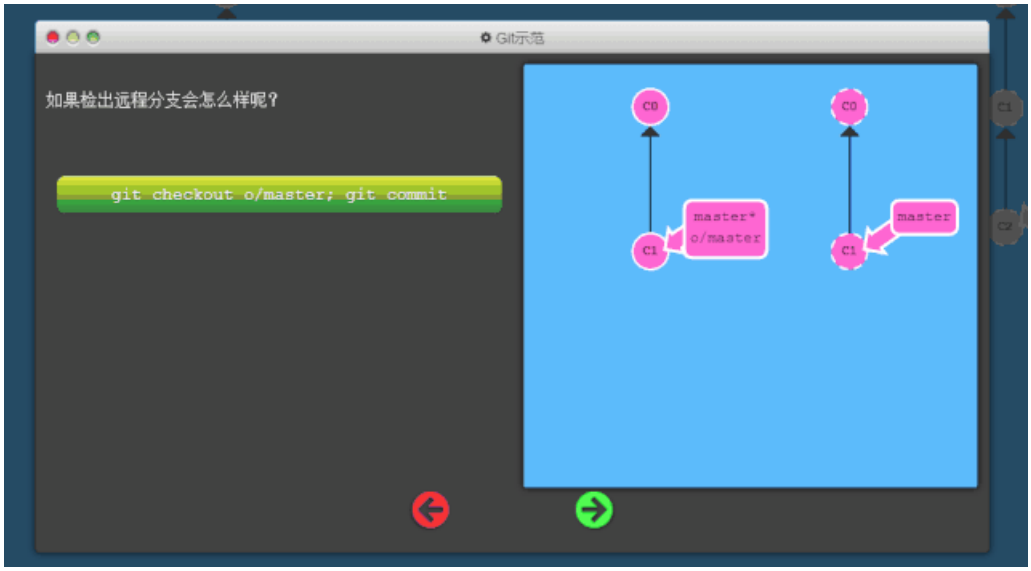
Learn Git Branching



https://learngitbranching.js.org

网站是有中文的，而且界面很不错，给人的第一印象很友好。

关卡选择界面可以看到，有「主要」、「远程」两部分。每一部分的内容不多，覆盖了基本的用法。每一关都是一个模拟的小项目，通过上手操作很容易获得代入感。代码本身的用法在玩游戏的过程中领会到，这种感觉很棒，让人上瘾。



教程的每一关都有引导，告诉你示范动作是怎样的、会有什么结果。稍作了解之后就可以自己上手探索。模仿之前教的动作，做错了可以 reset 命令从头开始。完成之后，可以用 show solution 命令查看答案。这种即时反馈的学习让过程变得有趣。

从 Git 到 GitHub

说了这么多，简单讲一下具体的流程：

1、git clone 下载代码到本地。

2、创建了自己的文件，或者进行了修改，可以用 git add . 把所有文件加入暂存区，等待建立节点。

3、git commit -m "这一阶段工作描述" 尽量细化你的节点，别做了很多工作才提交一次呀。

4、git push 把之前建立的一系列节点推送到 GitHub 发布、保存。

5、git pull 将 GitHub 的改动同步到本地，比如你在办公室电脑的改动同步到家里，或者多人协作项目中他人的改动同步到本地。

总结一下就是：**「记录修改、本地提交、传到云端」**的过程。

其他学习资源

除了之前文章中提到的 廖雪峰的 Git 教程 外，还有很多学习资料。比如：

Pro Git 第二版

http://bit.ly/2H7A7Lg

这本书被誉为 Git 学习的圣经，作者是 Scott Chacon 和 Ben Straub。Scott Chacon 在 GitHub 工作，自称 Git 的布道者。你可以在网站上免费阅读这本书，也可以下载他们提供的电子版本。

git-recipes

http://bit.ly/2Z4jw0M

它 童仲毅 (geeeeeeeeek@github) 对很多英文资料进行翻译、整理的集合教程。包含入门基础、进阶知识和应用范例。这些英文资料主要包括 GitHub 竞争者 Bitbucket 的 Git 教程。

Udacity Git 课程

http://bit.ly/2H5PZhi

谷歌无人车之父 Sebastian Thrun 创办的 优达学城（Udacity）上面的 免费 Git 课程。这门课程由优达学城与 GitHub 共同制作，介绍进行版本控制的基础知识，重点讲解 Git 版本控制系统以及 GitHub 协作平台。如果你更喜欢这种上课方式，这门课程随时都可以开始学习。

happypeter1983 的 Git 视频教程

http://bit.ly/2Z6rS87

这份教程就更进阶了一些。讲到了一些高级命令的用法。当然还有其他学习资源。

以上，便是今天的分享，觉得内容对你有所帮助的，还请点个「**在看**」支持，谢谢各位。

如果喜欢本篇文章，欢迎转发、点赞。关注订阅号「Web项目聚集地」，回复「进群」即可进入无广告技术交流。

推荐阅读

1. Java 实现 QQ 登陆
2. 在阿里干了 5 年，面试个小公司挂了…
3. 如何写出让同事无法维护的代码?
4. 史上最烂的项目：苦撑12年，600 多万行代码 ...

Web项目聚集地

微信扫描二维码，关注我的公众号

喜欢文章，点个在看

最常用的 Git 命令总结

命中水 Java后端 2019-10-27

来源 | www.cxiansheng.cn/daily/490

分支操作

1. git branch 创建分支
2. git branch -b 创建并切换到新建的分支上
3. git checkout 切换分支
4. git branch 查看分支列表
5. git branch -v 查看所有分支的最后一次操作
6. git branch -vv 查看当前分支
7. git brabch -b 分支名 origin/分支名 创建远程分支到本地
8. git branch --merged 查看别的分支和当前分支合并过的分支
9. git branch --no-merged 查看未与当前分支合并的分支
10. git branch -d 分支名 删除本地分支
11. git branch -D 分支名 强行删除分支
12. git branch origin :分支名 删除远处仓库分支
13. git merge 分支名 合并分支到当前分支上

暂存操作

1. git stash 暂存当前修改
2. git stash apply 恢复最近的一次暂存
3. git stash pop 恢复暂存并删除暂存记录
4. git stash list 查看暂存列表
5. git stash drop 暂存名(例：stash@{0}) 移除某次暂存
6. git stash clear 清除暂存

回退操作

1. git reset --hard HEAD^ 回退到上一个版本
2. git reset --hard ahdhs1(commit_id) 回退到某个版本
3. git checkout -- file撤销修改的文件(如果文件加入到了暂存区，则回退到暂存区的，如果文件加入到了版本库，则还原至加入版本库之后的状态)
4. git reset HEAD file 撤回暂存区的文件修改到工作区

标签操作

1. git tag 标签名 添加标签(默认对当前版本)
2. git tag 标签名 commit_id 对某一提交记录打标签
3. git tag -a 标签名 -m '描述' 创建新标签并增加备注
4. git tag 列出所有标签列表
5. git show 标签名 查看标签信息
6. git tag -d 标签名 删除本地标签
7. git push origin 标签名 推送标签到远程仓库
8. git push origin --tags 推送所有标签到远程仓库
9. git push origin :refs/tags/标签名 从远程仓库中删除标签

常规操作

1. git push origin test 推送本地分支到远程仓库
2. git rm -r --cached 文件/文件夹名字 取消文件被版本控制
3. git reflog 获取执行过的命令
4. git log --graph 查看分支合并图
5. git merge --no-ff -m '合并描述' 分支名 不使用Fast forward方式合并，采用这种方式合并可以看到合并记录
6. git check-ignore -v 文件名 查看忽略规则
7. git add -f 文件名 强制将文件提交

git创建项目仓库

1. git init 初始化
2. git remote add origin url 关联远程仓库
3. git pull
4. git fetch 获取远程仓库中所有的分支到本地

忽略已加入到版本库中的文件

1. git update-index --assume-unchanged file 忽略单个文件
2. git rm -r --cached 文件/文件夹名字 (. 忽略全部文件)

取消忽略文件

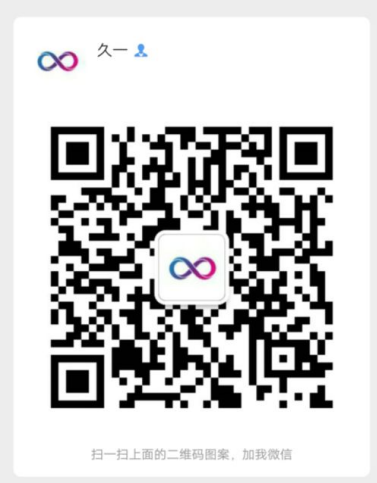
1. git update-index --no-assume-unchanged file

拉取、上传免密码

1. git config --global credential.helper store

-END-

↓扫描二维码进群↓



推荐阅读

1. 10 个让你笑的合不拢嘴的 GitHub 项目
2. 当我遵循了这 16 条规范写代码
3. 理解 IntelliJ IDEA 的项目配置和 Web 部署
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践

学Java，请关注公众号：Java后端

喜欢文章，点个在看

用好 Git 和 SVN ,轻松驾驭版本管理

凌承一 Java后端 1月15日

链接 | www.bubuko.com/infodetail-2844306.html

本文从 Git 与 SVN 的对比入手，介绍如何通过 Git-SVN 开始使用 Git，并总结平时工作高频率使用到的 Git 常用命令。

一、Git vs SVN

> Git 和 SVN 孰优孰好，每个人有不同的体验。
> Git是分布式的，SVN是集中式的

这是 Git 和 SVN 最大的区别。若能掌握这个概念，两者区别基本搞懂大半。因为 Git 是分布式的，所以 Git 支持离线工作，在本地可以进行很多操作，包括接下来将要重磅推出的分支功能。而 SVN 必须联网才能正常工作。

Git复杂概念多，SVN简单易上手

所有同时掌握 Git 和 SVN 的开发者都必须承认，Git 的命令实在太多了，日常工作需要掌握add,commit,status,fetch,push,rebase等，若要熟练掌握，还必须掌握rebase和merge的区别，fetch和pull的区别等，除此之外，还有cherry-pick，submodule，stash等功能，仅是这些名词听着都很绕。

在易用性这方面，SVN对于新手来说会更有好一些。但是从另外一方面看，Git 命令多意味着功能多，若我们能掌握大部分 Git 的功能，体会到其中的奥妙，会发现再也回不去 SVN 的时代了。

Git分支廉价，SVN分支昂贵

在版本管理里，分支是很常使用的功能。在发布版本前，需要发布分支，进行大需求开发，需要 feature 分支，大团队还会有开发分支，稳定分支等。在大团队开发过程中，常常存在创建分支，切换分支的求。

Git 分支是指针指向某次提交，而 SVN 分支是拷贝的目录。这个特性使 Git 的分支切换非常迅速，并且创建成本非常低。

而且 Git 有本地分支，SVN 无本地分支。在实际开发过程中，经常会遇到有些代码没写完，但是需紧急处理其他问题，若我们使用 Git，便可以创建本地分支存储没写完的代码，待问题处理完后，再回到本地分支继续完成代码。

二、Git 核心概念

Git 最核心的一个概念就是工作流。

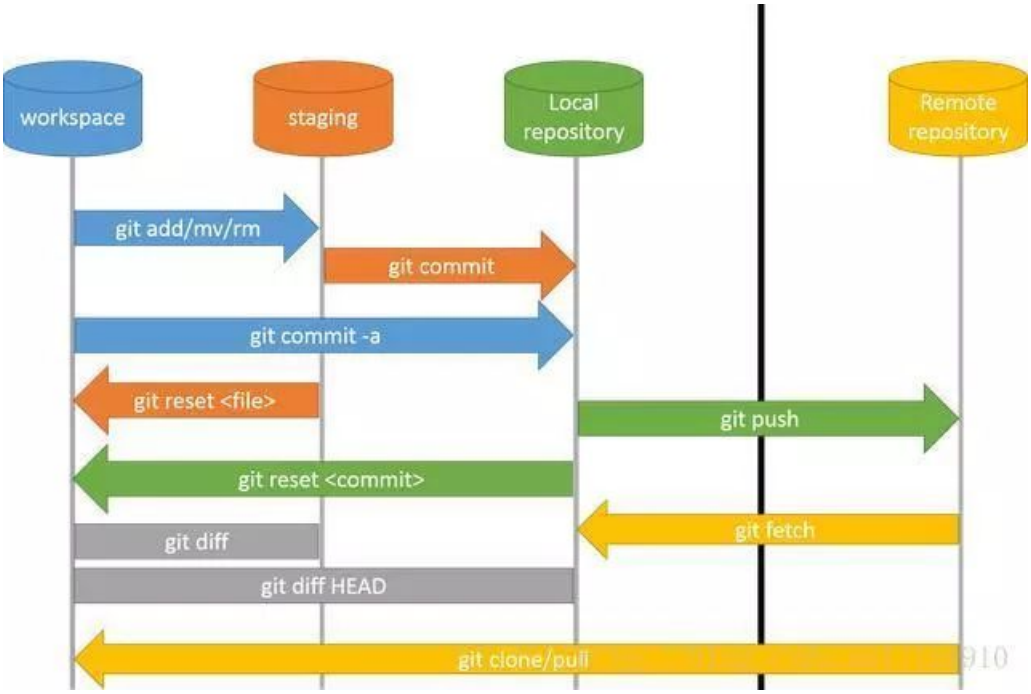
- 工作区(Workspace)是电脑中实际的目录。
- 暂存区(Index)类似于缓存区域，临时保存你的改动。
- 仓库区(Repository)，分为本地仓库和远程仓库。

从 SVN 切换到 Git，最难理解并且最不能理解的是暂存区和本地仓库。熟练使用 Git 后，会发现这简直是神设计，由于这两者的存在，使许多工作变得易管理。

通常提交代码分为几步：

1. git add从工作区提交到暂存区
2. git commit从暂存区提交到本地仓库
3. git push或git svn dcommit从本地仓库提交到远程仓库

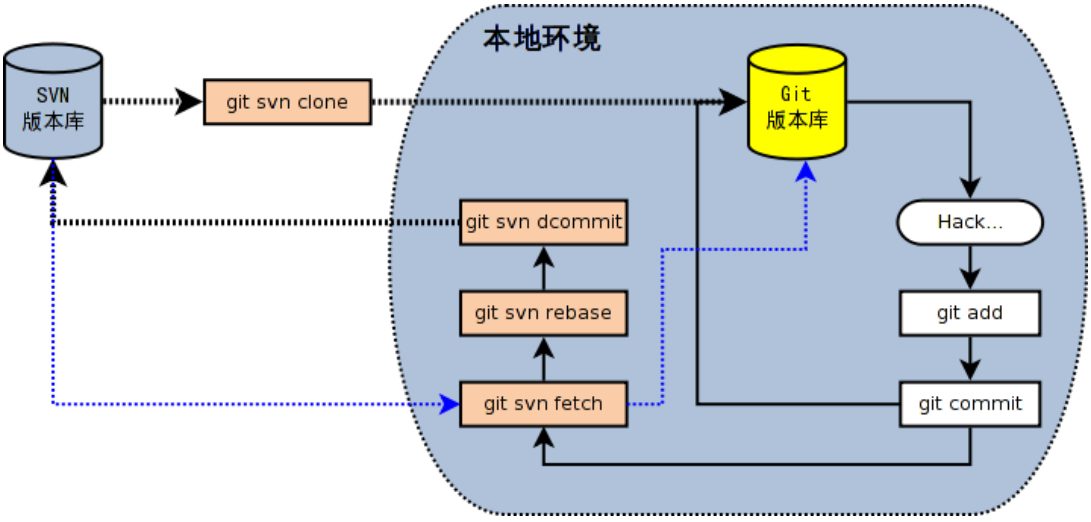
一般来说，记住以下命令，便可进行日常工作了（图片来源于网络）：



三、Git-SVN常用命令

若服务器使用的 SVN，但是本地想要体验 Git 的本地分支，离线操作等功能，可以使用 Git-SVN功能。

常用操作如下（图片来源于网络）：



[Git-SVN]

```
# 下载一个 SVN 项目和它的整个代码历史，并初始化为 Git 代码库
$ git svn clone -s [repository]
# 查看当前版本库情况
$ git svn info
# 取回远程仓库所有分支的变化
$ git svn fetch
# 取回远程仓库当前分支的变化，并与本地分支变基合并
$ git svn rebase
# 上传当前分支的本地仓库到远程仓库
$ git svn dcommit
# 拉取新分支，并提交到远程仓库
$ svn copy [remote_branch] [new_remote_branch] -m [message]
# 创建远程分支对应的本地分支
$ git checkout -b [local_branch] [remote_branch]
```

四、初始化

```
从本节开始，除特殊说明，以下命令均适用于 Git 与 Git-SVN。

# 在当前目录新建一个Git代码库
$ git init
# 下载一个项目和它的整个代码历史 [Git only]
$ git clone [url]
```

五、配置

```
# 列举所有配置
$ git config -l
# 为命令配置别名
$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.st status
$ git config --global alias.br branch
# 设置提交代码时的用户信息
$ git config [--global] user.name "[name]"
$ git config [--global] user.email "[email address]"
```

Git 用户的配置文件位于 ~/.gitconfig

Git 单个仓库的配置文件位于 ~/$PROJECT_PATH/.git/config

六、增删文件

```
# 添加当前目录的所有文件到暂存区
$ git add .
# 添加指定文件到暂存区
$ git add <file1> <file2> ...
# 添加指定目录到暂存区，包括其子目录
$ git add <dir>
# 删除工作区文件，并且将这次删除放入暂存区
$ git rm [file1] [file2] ...
# 停止追踪指定文件，但该文件会保留在工作区
$ git rm --cached [file]
# 改名文件，并且将这个改名放入暂存区
$ git mv [file-original] [file-renamed]
```

把文件名 file1 添加到 .gitignore 文件里，Git 会停止跟踪 file1 的状态。

七、分支

```
# 列出所有本地分支
$ git branch
# 列出所有本地分支和远程分支
$ git branch -a
# 新建一个分支，但依然停留在当前分支
$ git branch [branch-name]
# 新建一个分支，并切换到该分支
$ git checkout -b [new_branch] [remote-branch]
# 切换到指定分支，并更新工作区
$ git checkout [branch-name]
# 合并指定分支到当前分支
$ git merge [branch]
# 选择一个 commit，合并进当前分支
$ git cherry-pick [commit]
# 删除本地分支，-D 参数强制删除分支
$ git branch -d [branch-name]
# 删除远程分支
$ git push [remote] :[remote-branch]
```

八、提交

```
# 提交暂存区到仓库区
$ git commit -m [message]
# 提交工作区与暂存区的变化直接到仓库区
$ git commit -a
# 提交时显示所有 diff 信息
$ git commit -v
# 提交暂存区修改到仓库区，合并到上次修改，并修改上次的提交信息
$ git commit --amend -m [message]
# 上传本地指定分支到远程仓库
$ git push [remote] [remote-branch]
```

九、拉取

```
# 下载远程仓库的所有变动 (Git only)
$ git fetch [remote]
# 显示所有远程仓库 (Git only)
$ git remote -v
# 显示某个远程仓库的信息 (Git only)
$ git remote show [remote]
# 增加一个新的远程仓库，并命名 (Git only)
$ git remote add [remote-name] [url]
# 取回远程仓库的变化，并与本地分支合并，(Git only), 若使用 Git-SVN，请查看第三节
$ git pull [remote] [branch]
# 取回远程仓库的变化，并与本地分支变基合并，(Git only), 若使用 Git-SVN，请查看第三节
$ git pull --rebase [remote] [branch]
```

十、撤销

```
# 恢复暂存区的指定文件到工作区
$ git checkout [file]
# 恢复暂存区当前目录的所有文件到工作区
$ git checkout .
# 恢复工作区到指定 commit
$ git checkout [commit]
# 重置暂存区的指定文件，与上一次 commit 保持一致，但工作区不变
$ git reset [file]
# 重置暂存区与工作区，与上一次 commit 保持一致
$ git reset --hard
# 重置当前分支的指针为指定 commit，同时重置暂存区，但工作区不变
$ git reset [commit]
# 重置当前分支的HEAD为指定 commit，同时重置暂存区和工作区，与指定 commit 一致
$ git reset --hard [commit]
# 新建一个 commit，用于撤销指定 commit
$ git revert [commit]
# 将未提交的变化放在储藏区
$ git stash
# 将储藏区的内容恢复到当前工作区
$ git stash pop
```

十一、查询

```
# 查看工作区文件修改状态
$ git status
# 查看工作区文件修改具体内容
$ git diff [file]
# 查看暂存区文件修改内容
$ git diff --cached [file]
# 查看版本库修改记录
$ git log
# 查看某人提交记录
$ git log --author=someone
# 查看某个文件的历史具体修改内容
$ git log -p [file]
# 查看某次提交具体修改内容
$ git show [commit]
```

实际环境，Git的使用的确比SVN要多一点，毕竟功能要强大一些。**你的工作环境是使用svn or git ？欢迎分享！**

- END -

推荐阅读

1. 彻底征服 Spring AOP

2. 使用 ThreadLocal 一次解决老大难问题

3. 一场近乎完美基于 Dubbo 的微服务改造实践

4. GitHb 项目搜索技巧

5. Spring Boot 常见错误及解决方法

喜欢文章，点个在看

用好 Git 和 SVN，轻松驾驭版本管理

Java后端 3月10日

来源 | 凌承一

链接 | bubuko.com/infodetail-2844306.html

本文从 Git 与 SVN 的对比入手，介绍如何通过 Git-SVN 开始使用 Git，并总结平时工作高频率使用到的 Git 常用命令。

一、Git vs SVN

> Git 和 SVN 孰优孰好，每个人有不同的体验。
> Git是分布式的，SVN是集中式的

这是 Git 和 SVN 最大的区别。若能掌握这个概念，两者区别基本搞懂大半。因为 Git 是分布式的，所以 Git 支持离线工作，在本地可以进行很多操作，包括接下来将要重磅推出的分支功能。而 SVN 必须联网才能正常工作。

Git复杂概念多，SVN简单易上手

所有同时掌握 Git 和 SVN 的开发者都必须承认，Git 的命令实在太多了，日常工作需要掌握 add,commit,status,fetch,push,rebase等，若要熟练掌握，还必须掌握rebase和merge的区别，fetch和pull的区别等，除此之外，还有cherry-pick，submodule，stash等功能，仅是这些名词听着都很绕。

在易用性这方面，SVN对于新手来说会更有好一些。但是从另外一方面看，Git 命令多意味着功能多，若我们能掌握大部分 Git 的功能，体会到其中的奥妙，会发现再也回不去 SVN 的时代了。

Git分支廉价，SVN分支昂贵

在版本管理里，分支是很常使用的功能。在发布版本前，需要发布分支，进行大需求开发，需要 feature 分支，大团队还会有开发分支，稳定分支等。在大团队开发过程中，常常存在创建分支，切换分支的求。

Git 分支是指针指向某次提交，而 SVN 分支是拷贝的目录。这个特性使 Git 的分支切换非常迅速，并且创建成本非常低。

而且 Git 有本地分支，SVN 无本地分支。在实际开发过程中，经常会遇到有些代码没写完，但是需紧急处理其他问题，若我们使用 Git，便可以创建本地分支存储没写完的代码，待问题处理完后，再回到本地分支继续完成代码。

二、Git 核心概念

Git 最核心的一个概念就是工作流。

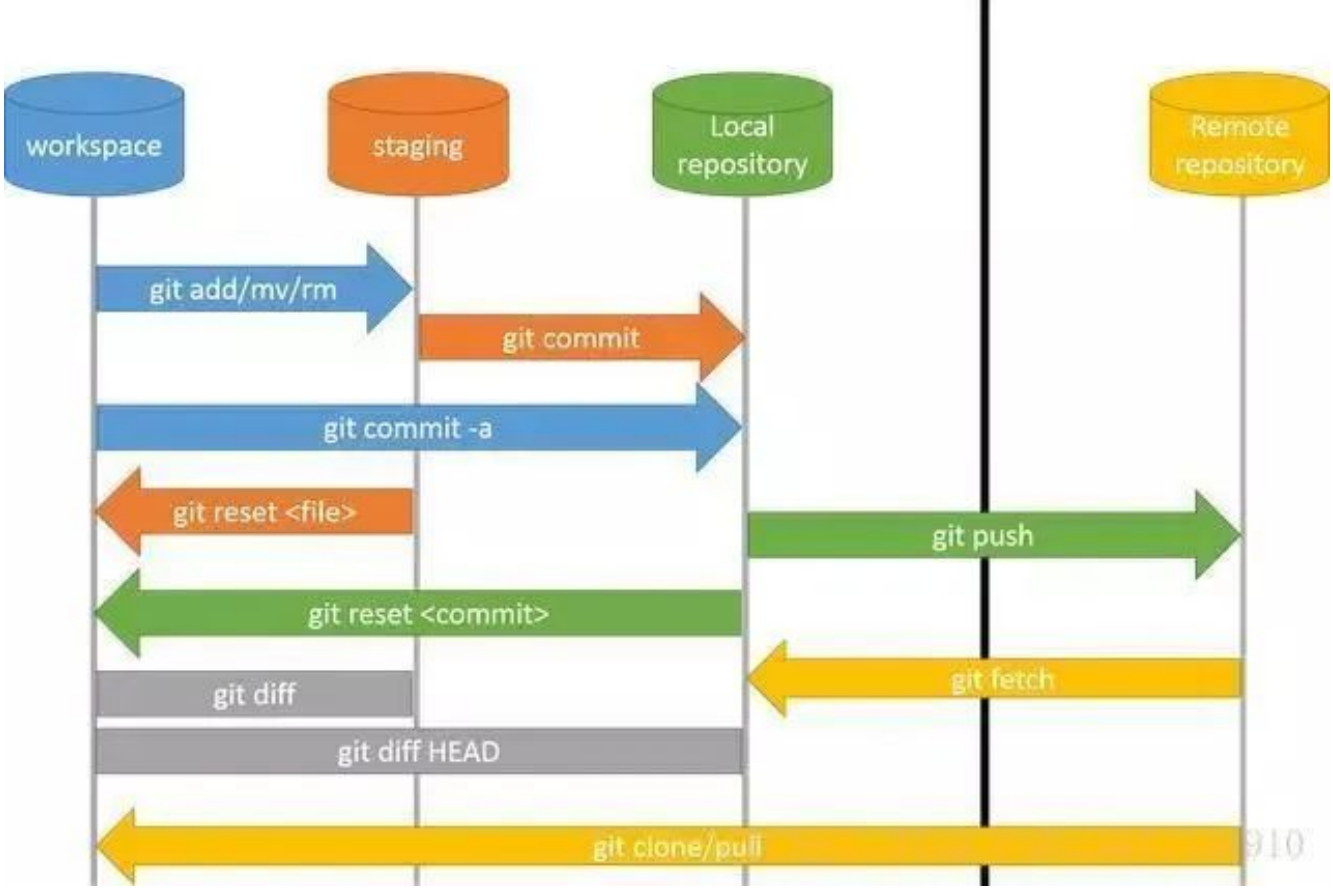
- 工作区(Workspace)是电脑中实际的目录。
- 暂存区(Index)类似于缓存区域，临时保存你的改动。
- 仓库区(Repository)，分为本地仓库和远程仓库。

从 SVN 切换到 Git，最难理解并且最不能理解的是暂存区和本地仓库。熟练使用 Git 后，会发现这简直是神设计，由于这两者的存在，使许多工作变得易管理。

通常提交代码分为几步：

1. git add从工作区提交到暂存区
2. git commit从暂存区提交到本地仓库
3. git push或git svn dcommit从本地仓库提交到远程仓库

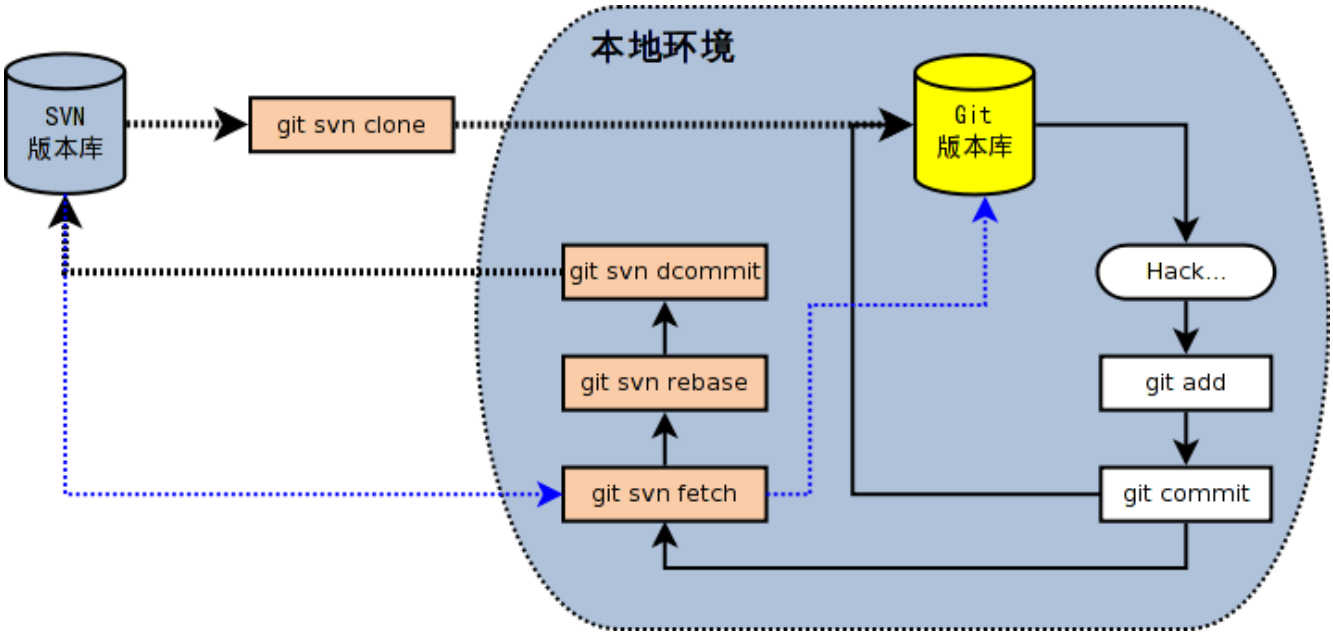
一般来说，记住以下命令，便可进行日常工作了（图片来源于网络）：



三、Git-SVN常用命令

若服务器使用的 SVN，但是本地想要体验 Git 的本地分支，离线操作等功能，可以使用 Git-SVN功能。

常用操作如下（图片来源于网络）：



[Git-SVN]

```
# 下载一个 SVN 项目和它的整个代码历史，并初始化为 Git 代码库
$ git svn clone -s [repository]
# 查看当前版本库情况
$ git svn info
# 取回远程仓库所有分支的变化
$ git svn fetch
# 取回远程仓库当前分支的变化，并与本地分支变基合并
$ git svn rebase
# 上传当前分支的本地仓库到远程仓库
$ git svn dcommit
# 拉取新分支，并提交到远程仓库
$ svn copy [remote_branch] [new_remote_branch] -m [message]
# 创建远程分支对应的本地分支
$ git checkout -b [local_branch] [remote_branch]
```

四、初始化

```
从本节开始，除特殊说明，以下命令均适用于 Git 与 Git-SVN。

# 在当前目录新建一个Git代码库
$ git init
# 下载一个项目和它的整个代码历史 [Git only]
$ git clone [url]
```

五、配置

```
# 列举所有配置
$ git config -l
# 为命令配置别名
$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.st status
$ git config --global alias.br branch
# 设置提交代码时的用户信息
$ git config [--global] user.name "[name]"
$ git config [--global] user.email "[email address]"
```

Git 用户的配置文件位于 ~/.gitconfig

Git 单个仓库的配置文件位于 ~/$PROJECT_PATH/.git/config

六、增删文件

```
# 添加当前目录的所有文件到暂存区
$ git add .
# 添加指定文件到暂存区
$ git add <file1> <file2> ...
# 添加指定目录到暂存区，包括其子目录
$ git add <dir>
# 删除工作区文件，并且将这次删除放入暂存区
$ git rm [file1] [file2] ...
# 停止追踪指定文件，但该文件会保留在工作区
$ git rm --cached [file]
# 改名文件，并且将这个改名放入暂存区
$ git mv [file-original] [file-renamed]
```

把文件名 file1 添加到 .gitignore 文件里，Git 会停止跟踪 file1 的状态。

七、分支

```
# 列出所有本地分支
$ git branch
# 列出所有本地分支和远程分支
$ git branch -a
# 新建一个分支，但依然停留在当前分支
$ git branch [branch-name]
# 新建一个分支，并切换到该分支
$ git checkout -b [new_branch] [remote-branch]
# 切换到指定分支，并更新工作区
$ git checkout [branch-name]
# 合并指定分支到当前分支
$ git merge [branch]
# 选择一个 commit，合并进当前分支
$ git cherry-pick [commit]
# 删除本地分支，-D 参数强制删除分支
$ git branch -d [branch-name]
# 删除远程分支
$ git push [remote] :[remote-branch]
```

八、提交

```
# 提交暂存区到仓库区
$ git commit -m [message]
# 提交工作区与暂存区的变化直接到仓库区
$ git commit -a
# 提交时显示所有 diff 信息
$ git commit -v
# 提交暂存区修改到仓库区，合并到上次修改，并修改上次的提交信息
$ git commit --amend -m [message]
# 上传本地指定分支到远程仓库
$ git push [remote] [remote-branch]
```

九、拉取

```
# 下载远程仓库的所有变动 (Git only)
$ git fetch [remote]
# 显示所有远程仓库 (Git only)
$ git remote -v
# 显示某个远程仓库的信息 (Git only)
$ git remote show [remote]
# 增加一个新的远程仓库，并命名 (Git only)
$ git remote add [remote-name] [url]
# 取回远程仓库的变化，并与本地分支合并，(Git only), 若使用 Git-SVN，请查看第三节
$ git pull [remote] [branch]
# 取回远程仓库的变化，并与本地分支变基合并，(Git only), 若使用 Git-SVN，请查看第三节
$ git pull --rebase [remote] [branch]
```

十、撤销

```
# 恢复暂存区的指定文件到工作区
$ git checkout [file]
# 恢复暂存区当前目录的所有文件到工作区
$ git checkout .
# 恢复工作区到指定 commit
$ git checkout [commit]
# 重置暂存区的指定文件，与上一次 commit 保持一致，但工作区不变
$ git reset [file]
# 重置暂存区与工作区，与上一次 commit 保持一致
$ git reset --hard
# 重置当前分支的指针为指定 commit，同时重置暂存区，但工作区不变
$ git reset [commit]
# 重置当前分支的HEAD为指定 commit，同时重置暂存区和工作区，与指定 commit 一致
$ git reset --hard [commit]
# 新建一个 commit，用于撤销指定 commit
$ git revert [commit]
# 将未提交的变化放在储藏区
$ git stash
# 将储藏区的内容恢复到当前工作区
$ git stash pop
```

十一、查询

```
# 查看工作区文件修改状态
$ git status
# 查看工作区文件修改具体内容
$ git diff [file]
# 查看暂存区文件修改内容
$ git diff --cached [file]
# 查看版本库修改记录
$ git log
# 查看某人提交记录
$ git log --author=someone
# 查看某个文件的历史具体修改内容
$ git log -p [file]
# 查看某次提交具体修改内容
$ git show [commit]
```

实际环境，Git的使用的确比SVN要多一点，毕竟功能要强大一些。**你的工作环境是使用svn or git ？ 欢迎分享！**

-END-

推荐阅读

1. 如何使用 Java 灵活读取 Excel 内容？

2. 贼好用的 Java 工具类库

3. IntelliJ IDEA 快捷键 Windows 版本

4. 盘点那些改变过世界的代码

阅读原文

花 20 分钟，再来梳理一下 Git 基础知识

Java后端 2019-09-28

作者 | 你喜欢吃青椒么

来源 | juejin.im/post/5d157bf3f265da1bcc1954e6

上篇 | 在 Spring Boot 中，如何干掉 if else

前言

本文是参考廖雪峰老师的Git资料再加上我自己对Git的理解，记录我的Git学习历程，作下此文是为以后学习，工作，开发中如果遇到问题可以回过头来参考参考。因为水平有限，难免会有出错的地方，欢迎指正。

Git是什么

官方话：Git是一个免费的开源分布式版本控制系统，旨在快速高效地处理从小型到大型项目的所有事务。

引用廖雪峰老师的话，它能自动帮我记录每次文件的改动，还可以让同事协作编辑，这样就不用自己管理一堆类似的文件了，也不需要把文件传来传去。如果想查看某次改动，只需要在软件里瞄一眼就可以。

为什么要学习Git

- 面试要被问。可以应付面试。
- 很多公司开发都用Git来处理项目。现在不学，以后肯定还要学。
- 在我看来Git是现如今所有程序员都要掌握的，以后与同事共同开发项目必定要用到的，熟练掌握Git命令，可以提高开发的效率。

安装Git

Windows

直接在官网上去下载。下载完成后，随便在某个文件下右键如果有Git Bash Here就安装成功。安装后，还要在命令行输入

```
$git config --global user.name "你的名字
"
$git config --global user.email "你的邮箱
"
```

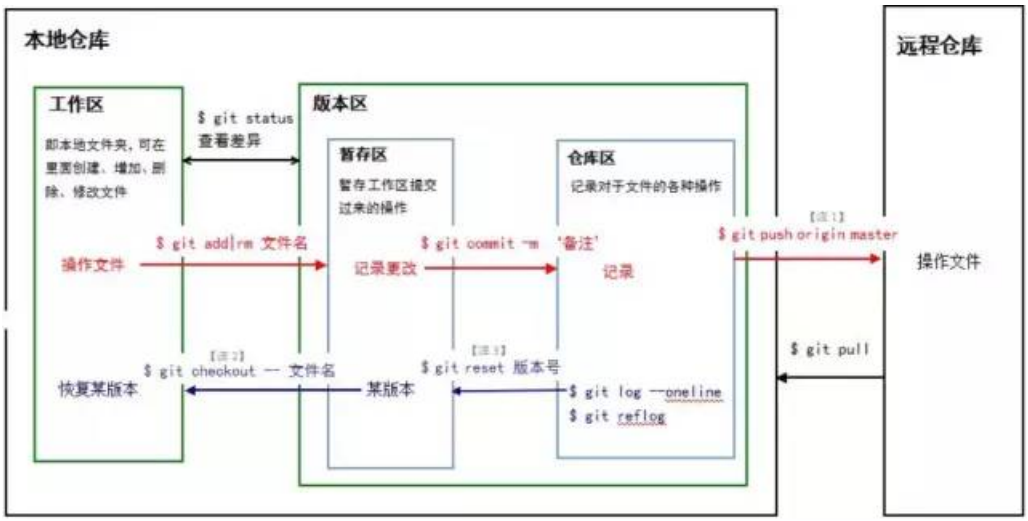
global表示全局，这台机器所有的Git仓库都会使用这个配置。允许单个仓库使用其他的名字和邮箱。

Mac

> Mac也可以像Windows一样，按上面的步骤安装。

也可以直接从AppStore安装Xcode，Xcode集成了Git，不过默认没有安装，你需要运行Xcode，选择菜单“Xcode”->“Preferences”，在弹出窗口中找到“Downloads”，选择“Command Line Tools”，点“Install”就可以完成安装了。

仓库



本地仓库是对于远程仓库而言的。本地仓库 = 工作区 + 版本区。

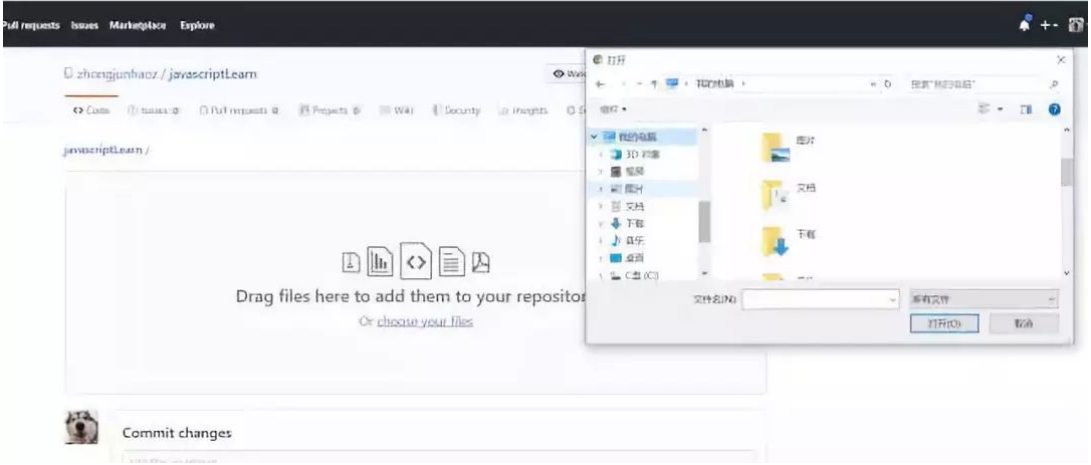
- 工作区即磁盘上的文件集合。
- 版本区(版本库)即.git文件。
- 版本库 = 暂存区(stage) + 分支(master) + 指针Head。

以我使用最频繁的git命令为例，即提交到github为例。

- git init 原本本地仓库只包含着工作区，这是最常见的工作状态。此时，git init一下，表示在本地区域创建了一个.git文件，版本区建立。
- git add . 表示把工作区的所有文件全部提交到版本区里面的暂存区
- 当然你也可以通过 git add ./xxx/ 一条一条分批添加到暂存区。
- git commit -m "xxx" 把暂存区的所有文件提交到仓库区，暂存区空空荡荡。
- git remote add origin https://github.com/name/name_cangku.git 把本地仓库与远程仓库连接起来。
- git push -u origin master 把仓库区的文件提交到远程仓库里。
- 一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的。会有这样的信息nothing to commit, working tree clean

提交到GitHub

以前不熟悉git命令的时候，我提交项目到github上都是直接在网页上直接拉取文件提交上去的。有点羞耻。



- git init .初始化，表示把这个文件变成Git可以管理的仓库。初始化后打开隐藏的文件可以看到有一个.git文件。
- git add . 后面的一个点表示把这个文件全部提交到暂存区。
- git add ./readme.md/ 表示把这个文件下面的readme.md文件提交到暂存区。
- git commit -m "你要评论一点什么东西" git commit的意思是把暂存区的全部文件提交到本地仓库。-m后接评论。
- git remote add origin https://github.com/name/name_cangku.git表示把你本地的仓库与GitHub上的远程仓库连接起来。只需要连接一次，以后提交的时候就可以不用谢这条命令了。name是你的github名字，name_cangku是你的仓库名。注意不要把后面的.git给漏掉了。因为我前面就是这么走过来的，绕了很多弯路。至于如何在GitHub上新建仓库，网上有很多教程，这里不再赘述了。
- git push -u origin master 把本地仓库提交到远程仓库。(最后一步)在你的远程仓库上刷新一下就可以看到你提交的文件了。
- 最后提到的是，在git commit -m ""之前，可以重复git add到暂存区。但是git commit会把你之前存放在暂存区的全部文件一次性全部提交到本地仓库。

版本的回溯与前进

提交一个文件，有时候我们会提交很多次，在提交历史中，这样就产生了不同的版本。每次提交，Git会把他们串成一条时间线。如何回溯到我们提交的上一个版本，用git reset --hard + 版本号即可。版本号可以用git log来查看，每一次的版本都会产生不一样的版本号。

回溯之后，git log查看一下发现离我们最近的那个版本已经不见了。但是我还想要前进到最近的版本应该如何？只要git reset --hard + 版本号就行。退一步来讲，虽然我们可以通过git reset --hard + 版本号,靠记住版本号来可以在不同的版本之间来回穿梭。

但是,有时候把版本号弄丢了怎么办？git reflog帮你记录了每一次的命令，这样就可以找到版本号了，这样你又可以通过git reset来版本穿梭了。

撤销

场景1：在工作区时，你修改了一个东西，你想撤销修改，git checkout -- file。廖雪峰老师指出撤销修改就回到和版本库一模一样的状态，即用版本库里的版本替换工作区的版本。

场景2：你修改了一个内容，并且已经git add到暂存区了。想撤销怎么办？回溯版本，git reset --hard + 版本号,再git checkout

-- file,替换工作区的版本。

场景3：你修改了一个内容，并且已经git commit到了master。跟场景2一样，版本回溯，再进行撤销。

删除

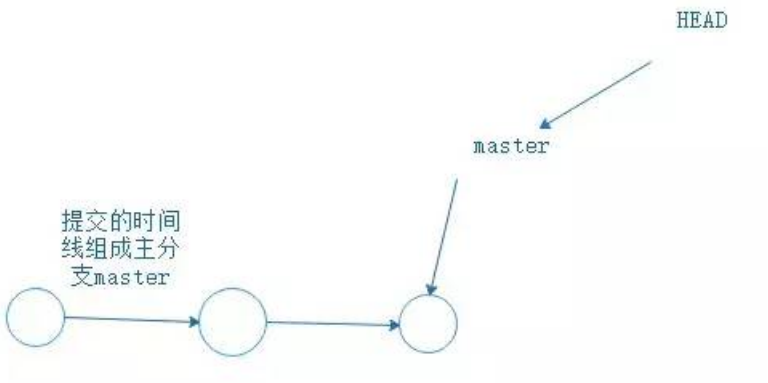
如果你git add一个文件到暂存区，然后在工作区又把文件删除了，Git会知道你删除了文件。如果你要把版本库里的文件删除，git rm 并且git commit -m "xxx".

如果你误删了工作区的文件，怎么办？使用撤销命令，git checkout --就可以。这再次证明了撤销命令其实就是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

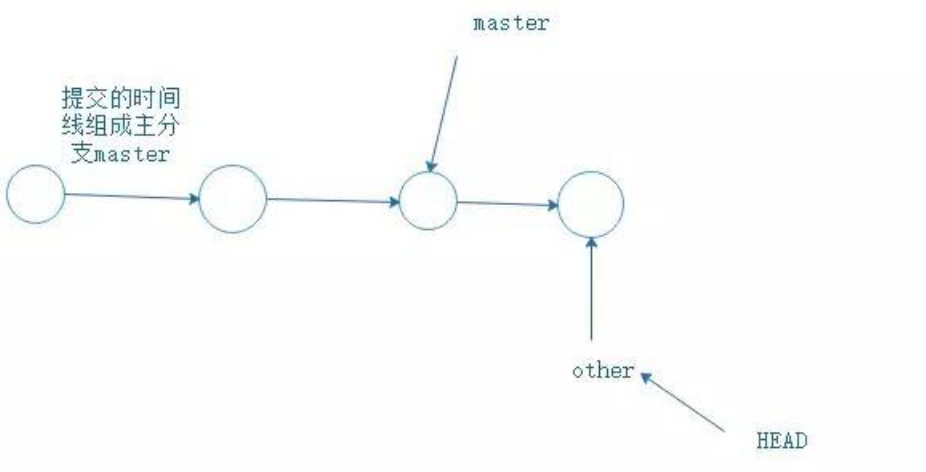
分支

分支，就像平行宇宙，廖雪峰老师如是说。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。用 Git 和 Github 提高效率的 10 个技巧！这篇也推荐看下。

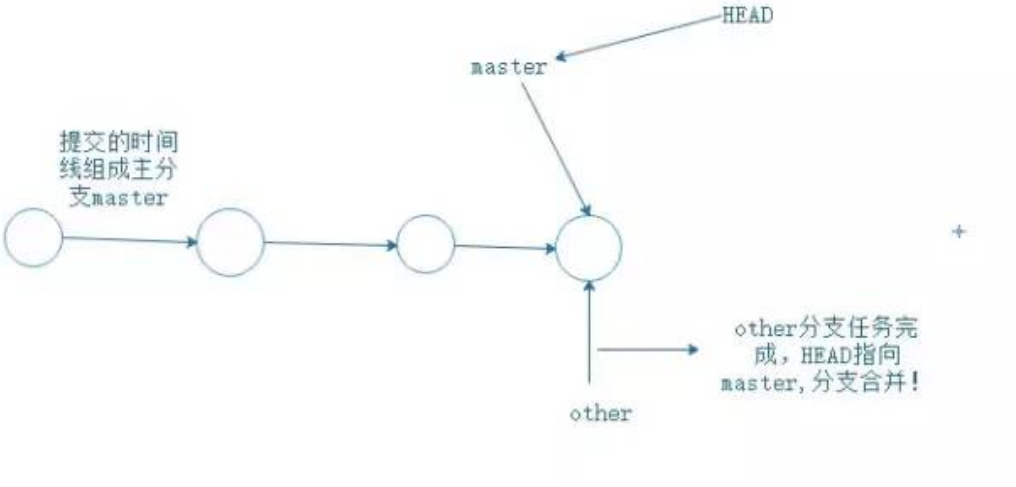
创建与合并分支



在没有其他分支插进来时，只有一个master主分支。每次你git push -u origin master 提交就是增加一条时间轴，master也会跟着移动。



创建一个other的分支，通过other提交，虽然时间轴向前走了，但是主分支master还在原来的位置。



理论分析完，看一下命令怎么写。

创建分支other,切换到other分支。

```
git branch other
git checkout other
```

查看当前所有分支

```
git branch
```

```
* other
  master
```

当前的分支会有一个*

用other提交

```
git add ./xxx/
git commit -m "xxx"
```

other分支完成，切换回master

```
git checkout master
```

此时，master分支上并没有other的文件，因为分支还没有合并。

合并分支

```
git merge other
```

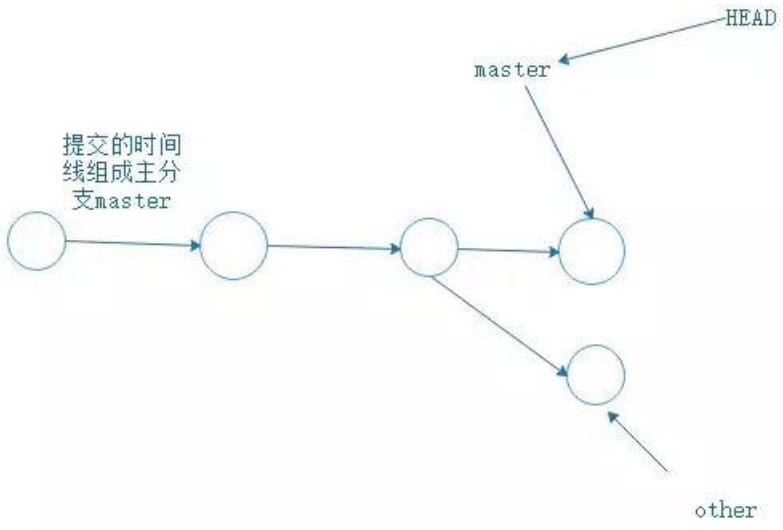
合并完成之后，就可以在master分支上查看到文件了。

删除other分支

```
git branch -d other
```

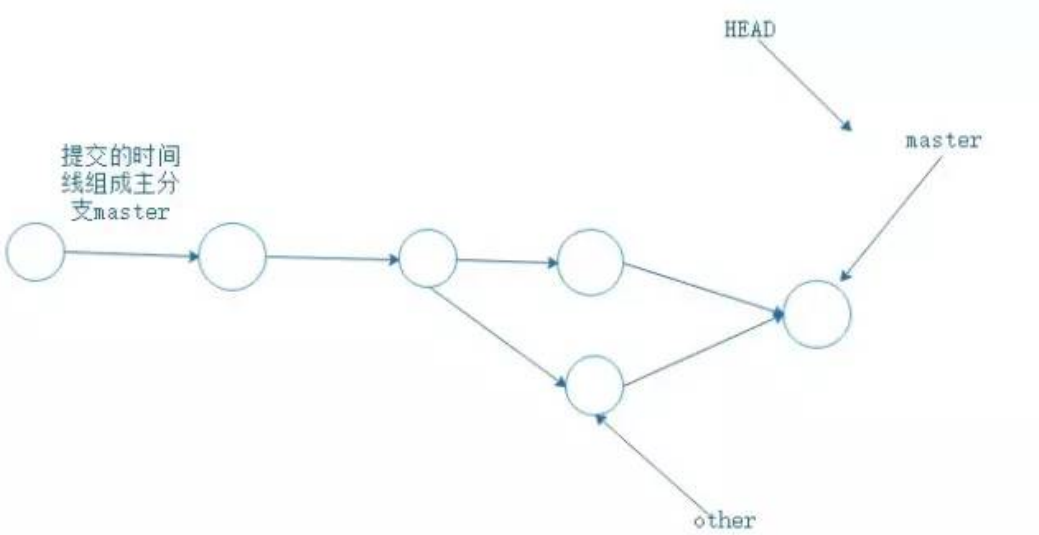
我由此想到，在以后工作中，应该是一个开放小组共同开发一个项目，组长会创建很多分支，每一个分支可以交给一个人去开发某一个功能，一个小组共同开发而且不会相互干扰。谁的功能完成了，可以由组长合并一下完成了的分支。哦，完美！*关注微信公众号 Java后端 获取更多推送。*

解决合并分支问题



假如有这样一种情况，分支other已经commit了，但是此时指针指回master时，并且master没有合并，而是git add / commit 提交了。这样，就产生了冲突，主分支master文件内容与other分支的内容不一样。合并不起来！所以，

- 修改文件的内容，让其保持一致。
- git add git commit 提交。
- 分支合并了。



- git log --graph 查看分支合并图
- git branch -d other 删除分支，任务结束。

分支管理策略

git merge --no-ff other 禁用Fast forward模式，因为使用Fast forward模式，删除分支后，分支历史信息会丢失。

BUG分支

> 廖雪峰老师提到，工作中每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。但如果你手上有分支在工作中，你的上级要你改另外的分支的BUG。
>
> 你要把现在正在工作的分支保存下来，git stash,把当前工作现场“存储”起来，等以后恢复后继续工作。当你解决BUG后，git checkout other回到自己的分支。用git stash list查看你刚刚“存放”起来的工作去哪里了。

此时你要恢复工作：

- git stash apply恢复却不删除stash内容，git stash drop删除stash内容。
- git stash pop恢复的同时把stash内容也删了.
- 此时，用git stash list查看，看不到任何stash 内容。

总结：修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；当手头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场

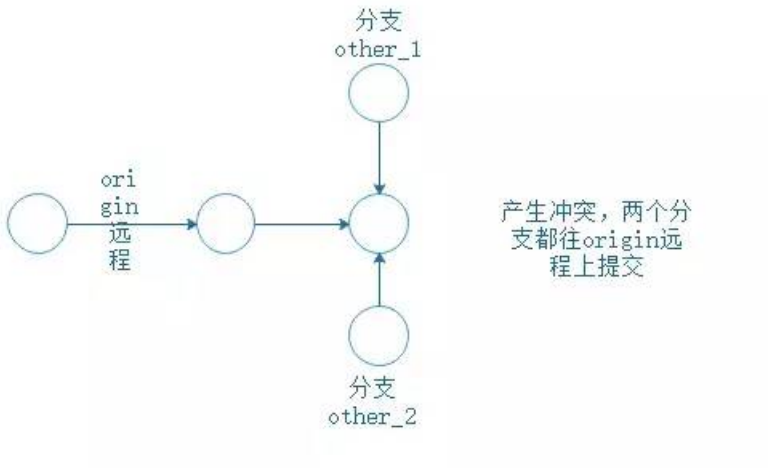
删除分支

- git branch -d + 分支有可能会删除失败，因为Git会保护没有被合并的分支。
- git branch -D + 分支 强行删除，丢弃没被合并的分支。

多人协作

- git remote 查看远程库的信息，会显示origin，远程仓库默认名称为origin
- git remote -v显示更详细的信息
- git push -u origin master推送master分支到origin远程仓库。
- git push -u origin other 推送other到origin远程仓库。

抓取分支



产生上图的冲突时，

- git pull 把最新的提交从远程仓库中抓取下来，在本地合并，解决冲突。在进行git pull
- 如果git pull 也失败了，还要指定分支之间的链接，这一步Git会提醒你怎么做。然后再git pull。

廖雪峰老师的总结：多人协作的工作模式通常是这样：

- 首先，可以试图用git push origin

  推送自己的修改；
- 如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；
- 如果合并有冲突，则解决冲突，并在本地提交；
- 没有冲突或者解决掉冲突后，再用git push origin

  推送就能成功！
- 如果git pull提示no tracking information，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream-to origin/。

Rebase

git rebase 把分叉的提交历史“整理”成一条直线，看上去更直观.缺点是本地的分叉提交已经被修改过了。

最后在进行git push -u origin master

rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比。

标签管理

比如一个APP要上线，通常在版本库中打一个标签(tag),这样，就确定了打标签的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。*关注微信公众号 Java后端 获取更多推送。*

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针。

tag其实就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。比如tag v2.1就是把历史上的一个版本的东西叫做v2.1

创建标签

步骤：

- git branch查看当前分支,git checkout master切换到master分支。
- git tag <name> 打标签，默认为HEAD。比如git tag v1.0
- 默认标签是打在最新提交的commit上的。如果想要打标签在以前的commit上，要git log找到历史提交的commit id.
- 如果一个commt id是du2n2d9,执行git tag v1.0 du2n2d9就把这个版本打上了v1.0的标签了。
- git tag 查看所有标签，可以知道历史版本的tag
- 标签不是按时间顺序列出，而是按字母排序的。
- git show <tagname> 查看标签信息。
- git tag -a <标签名> -m "<说明>",创建带说明的标签。-a指定标签名，-m指定说明文字。用show可以查看说明。

操作标签

- git tag -d v1.0 删除标签。因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。
- git push origin <tagname> 推送某个标签到远程
- git push origin --tags 一次性推送全部尚未推送到远程的本地标签
- 如果标签推送到远程。git tag -d v1.0 先删除本地标签v1.0。git push origin :refs/tags/v1.0删除远程标签v1.0

自定义Git

- git config --global color.ui true让Git显示颜色，会让命令输出看起来更醒目
- 忽略特殊文件 创建一个.gitignore文件，把需要忽略的文件名填进去。Git就会自动忽略这些文件。我也在学习中遇到过这样的问题，比如node_modules文件就可以忽略。

> 忽略文件原则：忽略操作系统自动生成的文件，比如缩略图等；忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

- 强制提交已忽略的的文件。git add -f <file>
- git check-ignore -v <file>检查为什么Git会忽略该文件。
- 给Git命令配别名,这个有点骚，就是你以后想输入git rebase时,你给它一个“外号”，就叫它git nb。以后你可以通过git nb来代替git rebase。

常用Git命令总结

- git config --global user.name "你的名字" 让你全部的Git仓库绑定你的名字
- git config --global user.email "你的邮箱" 让你全部的Git仓库绑定你的邮箱

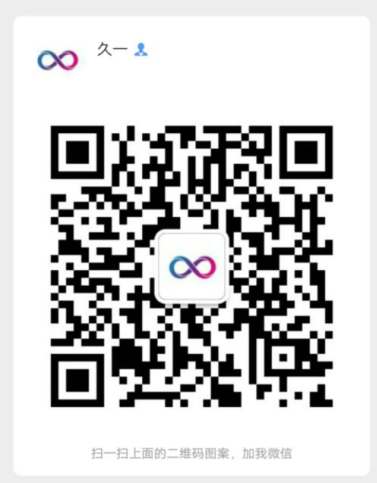
- git init 初始化你的仓库
- git add . 把工作区的文件全部提交到暂存区
- git add ./<file>/ 把工作区的<file>文件提交到暂存区
- git commit -m "xxx" 把暂存区的所有文件提交到仓库区，暂存区空空荡荡
- git remote add origin https://github.com/name/name_cangku.git 把本地仓库与远程仓库连接起来
- git push -u origin master 把仓库区的主分支master提交到远程仓库里
- git push -u origin <其他分支> 把其他分支提交到远程仓库
- git status查看当前仓库的状态
- git diff 查看文件修改的具体内容
- git log 显示从最近到最远的提交历史
- git clone + 仓库地址下载克隆文件
- git reset --hard + 版本号 回溯版本，版本号在commit的时候与master跟随在一起
- git reflog 显示命令历史
- git checkout -- <file> 撤销命令，用版本库里的文件替换掉工作区的文件。我觉得就像是Git世界的ctrl + z
- git rm 删除版本库的文件
- git branch 查看当前所有分支
- git branch <分支名字> 创建分支
- git checkout <分支名字> 切换到分支
- git merge <分支名字> 合并分支
- git branch -d <分支名字> 删除分支,有可能会删除失败，因为Git会保护没有被合并的分支
- git branch -D + <分支名字> 强行删除，丢弃没被合并的分支
- git log --graph 查看分支合并图
- git merge --no-ff <分支名字> 合并分支的时候禁用Fast forward模式,因为这个模式会丢失分支历史信息
- git stash 当有其他任务插进来时，把当前工作现场“存储”起来,以后恢复后继续工作
- git stash list 查看你刚刚“存放”起来的工作去哪里了
- git stash apply 恢复却不删除stash内容
- git stash drop 删除stash内容
- git stash pop 恢复的同时把stash内容也删了
- git remote 查看远程库的信息，会显示origin，远程仓库默认名称为origin
- git remote -v 显示更详细的信息
- git pull 把最新的提交从远程仓库中抓取下来，在本地合并,和git push相反

- git rebase 把分叉的提交历史“整理”成一条直线，看上去更直观
- git tag 查看所有标签，可以知道历史版本的tag
- git tag <name> 打标签，默认为HEAD。比如git tag v1.0
- git tag <tagName> <版本号> 把版本号打上标签，版本号就是commit时，跟在旁边的一串字母数字
- git show <tagName> 查看标签信息
- git tag -a <tagName> -m "<说明>" 创建带说明的标签。-a指定标签名，-m指定说明文字
- git tag -d <tagName> 删除标签
- git push origin <tagname> 推送某个标签到远程
- git push origin --tags 一次性推送全部尚未推送到远程的本地标签
- git push origin :refs/tags/<tagname> 删除远程标签<tagname>
- git config --global color.ui true 让Git显示颜色，会让命令输出看起来更醒目
- git add -f <file> 强制提交已忽略的的文件
- git check-ignore -v <file> 检查为什么Git会忽略该文件

-END-

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓扫描二维码进群↓



推荐阅读

1. Java后端优质文章整理
2. 在 Spring Boot 中，如何干掉 if else
3. Java 性能优化：教你提高代码运行的效率
4. 在浏览器输入 URL 回车之后发生了什么？
5. 接私活必备的 10 个开源项目

Java后端

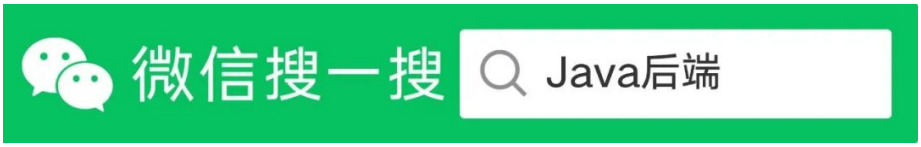
长按识别二维码，关注我的公众号

喜欢文章，点个在看

阅读原文

高频使用的 Git 命令

CRPER Java后端 2月29日



前言

汇总下我在项目中高频使用的git命令及姿势。

不是入门文档，官方文档肯定比我全面，这里是结合实际业务场景输出。

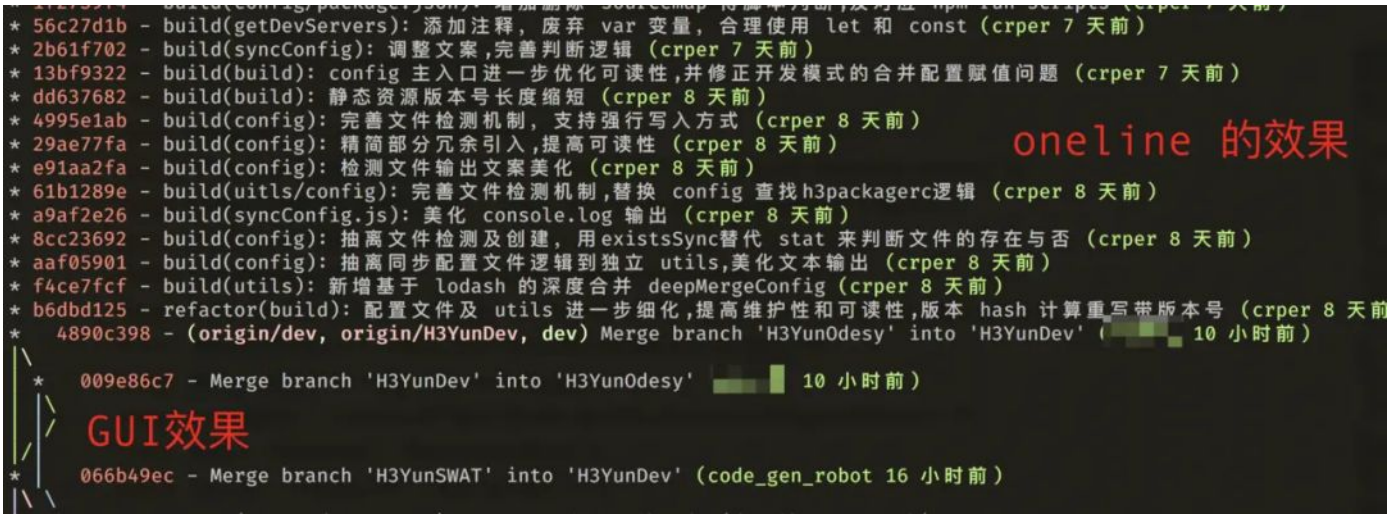
使用的 Git版本：**git version 2.24.0**

命令

git log

```
# 输出概要日志,这条命令等同于
# git log --pretty=oneline --abbrev-commit
git log --oneline

# 指定最近几个提交可以带上 - + 数字
git log --oneline -5

# 提供类似 GUI 工具的 log 展示
git log --graph --date=relative --pretty=tformat:'%Cred%h%Creset -%C(auto)%d%Creset %s %Cgreen(%an %ad)%Creset'
```

查看日志，常规操作，必备



git status

查看工作区状态的东东，不如GUI直观，但是命令行也有一些用的

```
git status
git status -s
git status --show-stash
git checkout
```

用来切换到对应记录的,可以基于分支,提交,标签。

切提交和标签一般用来热修复或者老版本需要加新特性。

```
git checkout dev

git checkout origin/test

git checkout --track origin/feature-test

git checkout -b testbranch

git checkout -- file

git checkout .

git checkout -
```

git commit

天天打交道的命令，这里说一些很常见的姿势

```
git commit --amend --no-edit

git commit --no-verify -m "xxx"

git commit -m "xxx"

git commit -t templateFile

git commit -F
```

git reset

不得不说，代码回滚中这个命令也是用的很多，而且是 --hard

```
git reset --hard commit_sha1
git reset --soft commit_sha1
git reset --soft HEAD~1
git reset --mixed commit_sha1
git reset --merge commit_sha1
git reset --keep commit_sha1
```

git revert

一般用于master 的代码回滚，因为多人在上面协作，

revert 可以平稳的回滚代码,但却保留提交记录,不会让协作的人各种冲突

```
git revert commit-sha1
```

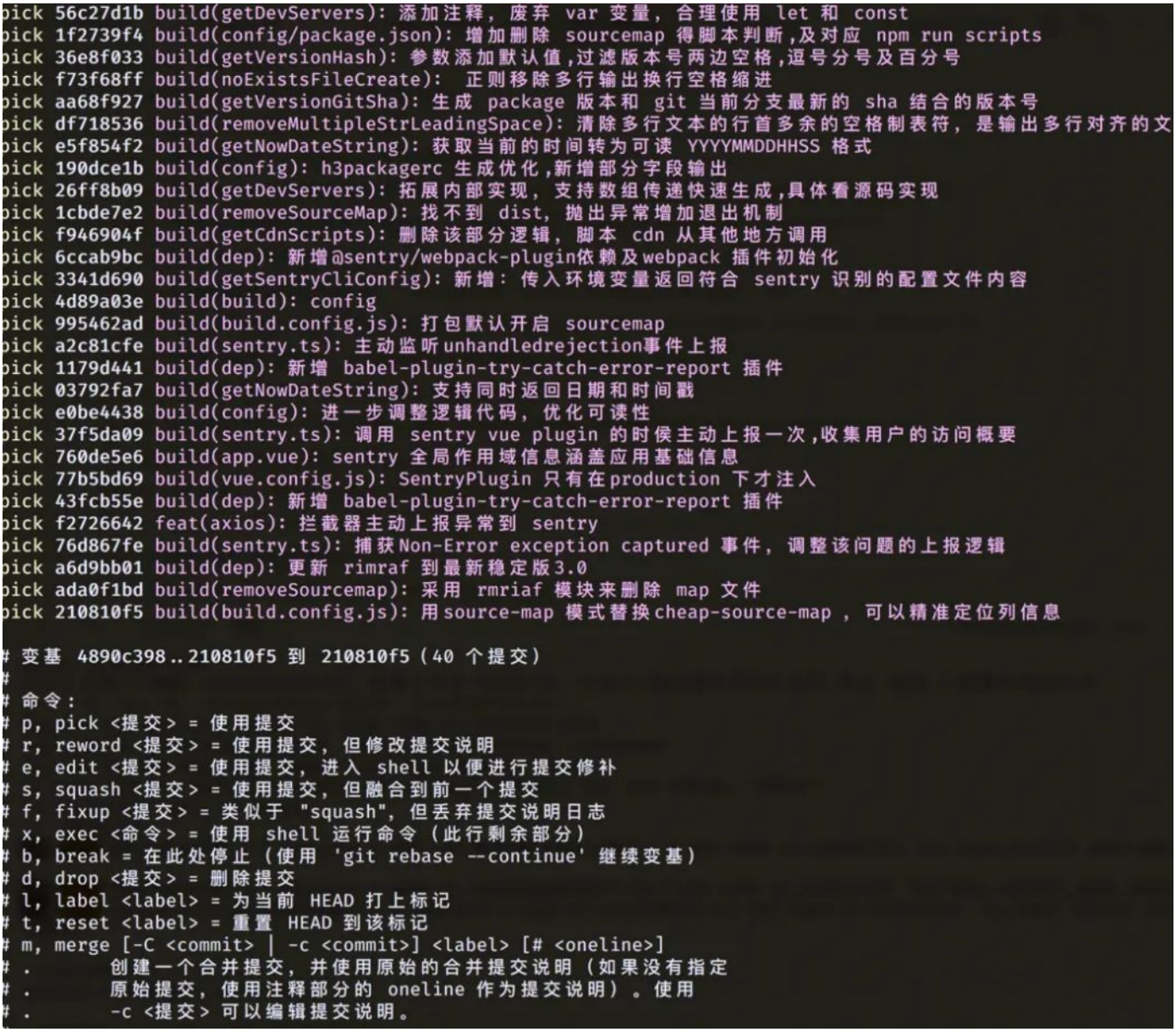
git rebase

变基在项目中算是很频繁的，为什么这么说。

比如你开发一个新的 feature, 遵循最小化代码提交的理念。

在整个功能开发完毕的时侯，会有非常多的 commit，用 rebase 可以让我们的commit记录很干净

```
git rebase -i git-sha1|branch(HEAD)
git rebase --continue
git rebase --skip
git rebase --abort
```



- pick: 是保留该 commit(采用)
- edit: 一般你提交的东西多了,可以用这个把东东拿回工作区拆分更细的 commit
- reword: 这个可以重新修改你的 commit msg
- squash: 内容保留，把提交信息往上一个 commit 合并进去
- fixup: 保留变动内容，但是抛弃 commit msg
- drop: 用的比较少，无用的改动你会提交么！！！

突然发现截图还有几个新的行为，估计是新版本带来的，

从字面上就可以看出来大体的意思, 就是把回滚和打标签这些放到变基中简化操作。

温馨提示：

- 本地提交之前，最好把基准点变为需要合并的分支，这样提交 PR/MR 的时侯就不会冲突(本地来解决冲突)
- 不要在公共分支上变基！！！一变其他协作者基本都一堆冲突！除非你们有很清晰的分支管理机制

git merge

```
git merge --no-ff branchName
```

git pull

git pull中用的最多是带--rebase(-r)的方式(变基形式拉取合并代码),保持分支一条线。

默认的pull会走ff模式,多数情况会产生新的commit,部分参数与 merge提供一致。

git push

当本地分支存在，远程分支不存在的时候，可以这样推送关联的远程分支

```
git push origin localbranch
git push -d origin branchName
git push --tags
git push --follow-tags
git push -f origin branchName
git push --force-with-lease
```

git remote

这个东西用在你需要考虑维护多个地方仓库的时候会考虑，或者修改仓库源的时候

```
git remote add origin url
git remote add github url
git remote set-url  origin(或者其他上游域) url
```

git branch

该命令用的最多的就是删除本地分支，重命名分支，删除远程分支了

```
git branch -d branchName
git branch -M oldBranch newNameBranch
git branch --set-upstream-to=origin/xxx
git branch --set-upstream-to origin xxx
```

git stash

暂存用的最多时侯就是你撸代码撸到一半，突然说有个紧急 BUG 要修正。

或者别人在你这里需要帮忙排查代码，你这时候也会用到。

强烈建议给每个 stash 添加描述信息！！！

```
git stash save stashName
git stash -u save stashName
git stash push -m "更改了 xx"
git stash apply stash@{0}
git stash pop stash@{0}
git stash list
git stash clear
git stash drop stash@{0}
git stash show stash@{0}
```

git reflog

这个命令的强大之处，是记录了所有行为，包括你 rebase,merge, reset 这些

当我们不小心硬回滚的时侯,或变基错了都可以在这里找到行为之前的commit，然后回滚。

当然这个时间回溯也只在本地有用，你推送到远程分支的破坏性改动,该凉还是得凉。

```
git reflog -5
```

git cherry-pick

这个东西你可以理解为你去买橘子，你会专门挑一些符合心意的橘子放到购物篮中。

你可以从多个分支同时挑取部分需要的 commit 合并到同一个地方去，是不是贼骚。

这货和变基有点类似，但是仅仅类似，挑过来的 commit 若是没有冲突则追加。

有冲突会中断，解决后 --continue

```
git cherry-pick commit-sha1
git cherry-pick master~4 master~2
git cherry-pick startGitSha1..endGitSha1
```

git rm

这个命令在旧的版本用的比较最多的姿势是为了重新索引.gitignore 的范围

```
git rm --cache -- file
git rm -r --cached .
git add .
git commit -m "xxx"
```

git rev-parse

这个估计一般人用的不是很多，可以通过这个快速获取部分git 仓库的信息

我在弄脚本的时侯就会从这里拿东西

```
git rev-parse --short HEAD --verify
git rev-parse --show-toplevel
git rev-parse --git-dir
git rev-parse --all
```

git diff

对于这个命令，在终端比对用的不是很频繁，除了少量改动的时侯可能会用这个看看。

其他情况下我更倾向于用 GUI 工具来看，因为比对更加直观。

总结

git 的常用命令其实很好掌握，很多命令都有 Linux 的影子。

列出来的命令都是高频使用的，或许有一些更骚的姿势没有摸索到，

有更好建议的，或者发现不对之处的请留言，会及时修正，谢谢阅读。

作者：CRPER

链接：juejin.im/post/5de8d849e51d455808332166

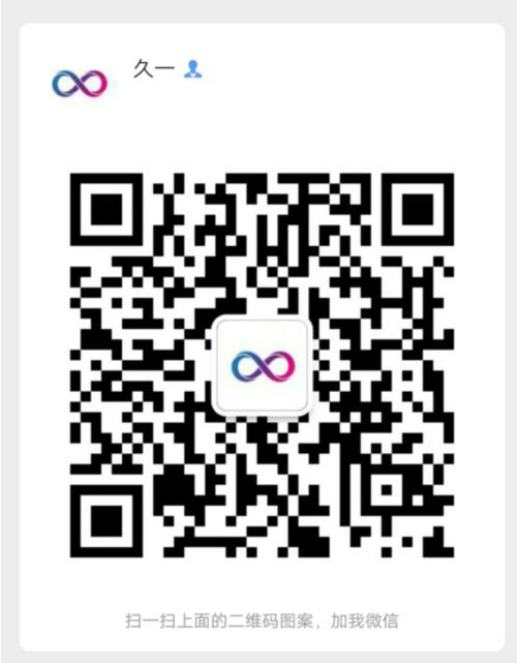
欢迎点击阅读原文访问作者博客。本文由公众号（Java后端）编辑，转载请著名且保留出处。

有偿投稿： 欢迎投稿原创技术博文，一旦采用将给予 50元 - 200元 不等稿费，要求个人原创，图文并茂。可以是职场经验、面试经历，也可是技术教程，学习笔记。投稿请联系微信「web527zsd」，备注投稿。

-END-

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓扫描二维码添加小编↓



推荐阅读

1. 一个依赖搞定 Session 共享

2. 用好 Java 中的枚举，真的没有那么简单！

3. 互联网公司的技术架构

4. 浅谈 Web 网站架构演变过程

阅读原文