**git 常用命令**

目录

[一、 git 安装后基本的配置 1](#_Toc452452382)

[二、 版本库的创建 1](#_Toc452452383)

[三、 git 的状态 2](#_Toc452452384)

[四、 命令使用 2](#_Toc452452385)

[1. git add 2](#_Toc452452386)

[2. git commit 3](#_Toc452452387)

[3. git reset 3](#_Toc452452388)

[4. git checkout 3](#_Toc452452389)

[5. git revert 4](#_Toc452452390)

[五、 git分支 4](#_Toc452452391)

[1. 分支的创建 4](#_Toc452452392)

[2. 分支的合并 4](#_Toc452452393)

[直接合并： 4](#_Toc452452394)

[压合合并： 5](#_Toc452452395)

[拣选合并： 5](#_Toc452452396)

[git rebase 6](#_Toc452452397)

[3. 分支的删除 6](#_Toc452452398)

[4. 分支的查看 7](#_Toc452452399)

[5. 分支重命名 7](#_Toc452452400)

[六、 冲突处理 7](#_Toc452452401)

[1. 分支的合并和revert冲突处理 7](#_Toc452452402)

[2. rebase 7](#_Toc452452403)

[七、 远程分支 8](#_Toc452452404)

[1. 克隆远程仓库 8](#_Toc452452405)

[2. 查看当前的远程库 8](#_Toc452452406)

[查看现有的版本库 git remote 8](#_Toc452452407)

[添加git remote add [主机名] [地址] 添加远程版本库 8](#_Toc452452408)

[删除git remote rm [主机名] 删除远程版本库 9](#_Toc452452409)

[重命名 git remote rename命令用于远程主机的改名。 9](#_Toc452452410)

[3. 从远程仓库抓取数据 9](#_Toc452452411)

[git fetch 9](#_Toc452452412)

[git pull 10](#_Toc452452413)

[4. 推送数据到远程仓库 10](#_Toc452452414)

[git push命令 11](#_Toc452452415)

[push 时常见错误 12](#_Toc452452416)

[八、 查看历史 12](#_Toc452452417)

[1. git log 12](#_Toc452452418)

[2. git reflog 14](#_Toc452452419)

[3. git blame 14](#_Toc452452420)

[九、 Patch 15](#_Toc452452421)

[1. git format-patch 15](#_Toc452452422)

[pacth生成 15](#_Toc452452423)

[应用patch 15](#_Toc452452424)

[2. git diff 15](#_Toc452452425)

[3. git archive 16](#_Toc452452426)

[十、 **保存当前的工作现场** 17](#_Toc452452427)

[十一、 git的一些配置 17](#_Toc452452428)

[1. 忽略掉文件权限检查 17](#_Toc452452429)

[2. 取消设置 17](#_Toc452452430)

[十二、 git使用小技巧 17](#_Toc452452431)

[1. 修改提交内容 17](#_Toc452452432)

[修改最后一次提交的内容 17](#_Toc452452433)

[格式：git commit --amend 18](#_Toc452452434)

[git commit --amend -m "new message" //如果暂存去里没有新的内容这句话就是重新提交最新节点的备注信息 18](#_Toc452452435)

[修改前几次提交的内容 18](#_Toc452452436)

[2. 批量删除文件时提交的暂存区的方法 18](#_Toc452452437)

[3. 二进制的合并是的冲突解决 18](#_Toc452452438)

[十三、 错误处理 18](#_Toc452452439)

[1. Error pulling origin: error: The following untracked working tree files would be overwritten by 18](#_Toc452452440)

# git 安装后基本的配置

git安装后首先要配置名字和邮箱，每次提交都要调用到的

配置方法：

git config --globel user.name “your name”

git config --globel user.email “your email”

ui颜色默认为自动

git config --globel color.ui true

# 版本库的创建

要在哪个项目上使用git管理就在该项目的跟目录执行

git init

初始化之后，当前目录下会多.git的目录，里面存放在git需要的资源和数据初始化之后我们就要往版本库里添加文件，添加文件之前要先把原有的过滤文件都删了，

find -type d | grep .git$ | xargs rm –rf

find -name .gitignore | xargs rm

之后在.git同级目录新建一个文件.gitignore,建立的过滤规则

/out 过滤out目录下的文件

\*~ 过滤~结尾的文件

\*.log 过滤.log的文件

其他的按实际需求加，编译之后会变动及更换服务器之后编译会变动的都要加进去

接下来要把所有的文件都加入纳入版本控制

git add .

git commit –m “frist version”

这样就把所有的文件都纳入管理了

还有一种方法就是克隆一个版本库

git clone [url]

如 git clone [user@192.168.0.26:/home/user/work/idh.code](mailto:user@192.168.0.26:/home/user/work/idh.code)

# git 的状态

一共有四个状态Untracked files（未跟踪）,nothing to be commit(未修改),changes not staged for commit（已修改）, changes to be commited（已暂存）

Untracked files（未跟踪）：指文件在工作目录下但没有加入版本库中。

nothing to be commit(未修改)：版本库中的文件没有做任何修改，就处于未修改状态。

changes not staged for commit（已修改）：文件已经修改过，可通过git add 加入暂存区

changes to be commited（已暂存）:git已经管理了这些改动，但是还没完成提交，可通过git commit 提交改动

# 命令使用

## git add

作用：将改动存入暂存区

用法：

1. **git add 文件名** 如git add package/apps/Settings/Android.mk 将package/apps/Settings/的android.mk的改动加入暂存区
2. **git add目录**如git add package/apps/Settings 将package/apps/Settings/下的改动和新加的文件加入暂存区
3. **git add .** 将所有的改动和新加的文件加入暂存区
4. **git add –A** 或**git ad –-all** 将所有的改动，新加的文件，及删除的文件都加入到暂存区

## git commit

作用：将内容提交到版本库之中

用法：

1. **git commit –m “这次改动的说明”** 如git commit –m “change softversion”,将暂存区的内容提交到版本库，改动的说明要尽量写的易懂，有代表行，完整，以方便日后查找
2. **git commit –a –m “这次改动的说明”**如 git commit –a –m “change softversion”, 将改动提交到版本库
3. git commit --amend -m "new message" //重新提交最新节点的备注信息

## git reset

作用：1.重置代码库

2.将暂存区的内容回退

用法：

1. **git reset 文件名** 如git reset package/apps/Settings/Android.mk，将暂存区的相应文件退回到changes not staged for commit（已修改）状态
2. **git reset目录** 如git reset package/apps/Settings 将暂存区中的package/apps/Settings/下的文件退回到changes not staged for commit（已修改）状态
3. **git reset .** 将暂存区所有的文件退回到changes not staged for commit（已修改）状态
4. **git reset 对应节点的哈希值** 将节点以后的改动都变为已修改状态

## git checkout

作用：1.将已修改的文件退回到未修改状态

2.切换分支

用法：

1. **git checkout文件名** 如git checkout package/apps/Settings/Android.mk，将相应文件的修改撤销，回到未修改的状态。
2. **git checkout目录** 如git checkout tpackage/apps/Settings 将package/apps/Settings/下的文件退回到未修改的状态
3. **git checkout.** 将所有的文件退回未修改的状态
4. **git checkout 分支名** 切换到相应分支

## git revert

作用：回退某一个节点的改动(git log里有记录)

用法：

**git revert对应节点的哈希值**

# git分支

## 分支的创建

创建及切换到新分支上

**git branch newname** 在当前检出分支上新建分支，名叫newname。

**git checkout newname** 这样就切换到新的分上

也可以合并成一步

**git checkout –b newname**

在当前状态下创建分支newname分支并检出到该分支。

**git checkout –b newname master**

这个命令将上面两个命令合并：在master分支上创建分支newname分支并检出到该分支。

## 分支的合并

### 直接合并：

　　比如要把mmx分支上的所有改动合并到master分支：

　　首先先到master分支：**git checkout master**

　　然后把mmx给合并过来：**git merge mmx**

注意没参数的情况下merge是**fast-forward**的，即Git将master分支的指针直接移到dev的最前方。

换句话说，如果顺着一个分支走下去可以到达另一个分支的话，那么Git在合并两者时，只会简单移动指针，所以这种合并成为**快进式(Fast-forward)**。

### 压合合并：

　　将一条分支上的若干个提交条目压合成一个提交条目，提交到另一条分支的末梢。

　　把dev分支上的所有提交压合成主分支上的一个提交，即压合提交：

**git checkout master**

**git merge --squash dev**

　　此时，dev上的所有提交已经合并到当前工作区并暂存，但还没有作为一个提交，可以像其他提交一样，把这个改动提交到版本库中：

**git commit –m “something from dev”**

### 拣选合并：

　　拣选另一条分支上的某个提交条目的改动带到当前分支上。

　　每一次提交都会产生一个全局唯一的提交名称，利用这个名称就可以进行拣选提交。

　　比如在dev上的某个提交叫：321d76f

　　把它合并到master中：

**git checkout master**

**git cherry-pick 321d76f**

git cherry-pick <start-commit-id>..<end-commit-id>

或

git cherry-pick <start-commit-id>^..<end-commit-id>

前者表示把<start-commit-id>到<end-commit-id>之间(左开右闭，不包含start-commit-id)的提交cherry-pick到当前分支；

后者表示把<start-commit-id>到<end-commit-id>之间(闭区间，包含start-commit-id)的提交cherry-pick到当前分支。

其中，<start-commit-id>到<end-commit-id>只需要commit-id的前6位即可，并且<start-commit-id>在时间上必须早于<end-commit-id>

### git rebase

git rebase有点类似git merge，但是两者又有不同， git merge是一下子把所有的改动都和过来，最后在处理冲突。git rebase，是将分支上的改动一个一个合入，有冲突马上处理，

# 合并b

git rebase b

**# 处理完冲突继续合并**

git rebase --continue

# 跳过

git rebase --ski

# 取消合并

git rebase --abort

## 分支的删除

有些分支没有必要长期保存，比如分支中的代码已经打了标签并已发布，或者实验分支已经成功完成工作或中途废弃等等。

**注意**：打了标签的分支，Git在删除该分支时，从版本树起始到此标签间的全部历史轨迹均会保留，此时删除分支操作只是删除分支本身的名称，因此可以说该分支没有必要长期保存。

　　而在其他版本控制工具中，删除分支通常意味着删除分支上的所有历史轨迹，所以不能因为打了标签就认为其没有必要保存。

　　删除一个分支dev2：

**git branch –d dev2**

　　注意不能删除当前所在分支，需要转到别的分支上。

　　如果要删除的分支已经成功合并到当前分支，删除分支的操作会直接成功。

　　如果要删除的分支没有合并到当前所在分支，则会出现提示，如果确定无须合并而要直接删除，则执行命令：

**git branch –D dev2**

　　进行强删。

## 分支的查看

git branch 查看本地 当前分支 前面有\*表示当前分支

git branch –a 查看所有分支

git branch –r 查看远程分支

git branch –v 查看每个分支最后一个提交的节点

## 分支重命名

**git branch –m oldname newname** 重命名分支：

**-m**不会覆盖已有分支名称，即如果名为newname的分支已经存在，则会提示已经存在了。

　　如果改成**-M**就可以覆盖已有分支名称了，即会强制覆盖名为newname的分支，这种操作要谨慎。

# 冲突处理

分支的合并，rebase，revert等在执行的时候，有一些git无法自动合并，称之为冲突(conflict)。解决冲突人工处理

## 分支的合并和revert冲突处理

首先git status 查看状态，会有both modified的文件。

打开both modified 找到　**<<<<<<<，======= 和 >>>>>>>**这些标志

**<<<<<<<**标记冲突开始，后面跟的是当前分支中的内容。

**HEAD**指向当前分支末梢的提交。

**=======**之后，**>>>>>>>**之前是要merge过来的另一条分支上的代码或revert的代码

对于简单的合并，手工编辑，然后去掉这些标记，最后像往常的提交一样

git add 需要提交的文件

git commit

## rebase

根据提示先判断有冲突的节点需不需要何如，如果不需要则git rebase –skip跳过这个改动如果需要合入首先git status 查看状态，会有both modified的文件。打开both modified 找到　**<<<<<<<，======= 和 >>>>>>>**这些标志

**<<<<<<<**标记冲突开始，后面跟的是当前分支中的内容。

**HEAD**指向当前分支末梢的提交。

**=======**之后，**>>>>>>>**之前节点的改动

对于简单的合并，手工编辑，然后去掉这些标记，最后像往常的提交一样

git add 需要提交的文件

git rebase -- continue

# 远程分支

## 克隆远程仓库

版本库的创建除了自己创建之后还可以克隆现有的远程仓库

克隆命令：

git clone <版本库的网址>

git clone <版本库的网址> <本地目录名>

如git clone [user@192.168.0.26:/home/user/work/idh.code](mailto:user@192.168.0.26:/home/user/work/idh.code)

## 查看当前的远程库

　要查看当前配置有哪些远程仓库,可以用 git remote 命令,它会列出每个远程库的简短名字.在克隆完某个项目后,至少可以看到一个名为 origin 的远程库,Git 默认使用这个名字来标识你所克隆的原始仓库

### 查看现有的版本库 git remote

查看现有的版本库，-v 选项(译注:此为 –verbose 的简写,取首字母),显示对应的克隆地址 git remote –v

### 添加git remote add [主机名] [地址] 添加远程版本库

如git remote add pb git://github.com/paulboone/ticgit.git pb表示 git://github.com/paulboone/ticgit.git这个地址

### 删除git remote rm [主机名] 删除远程版本库

如git remote rm pb 在本地删除pb这个远程仓库

### 重命名 git remote rename命令用于远程主机的改名。

git remote rename [原主机名] [新主机名]

## 从远程仓库抓取数据

### git fetch

一旦远程主机的版本库有了更新（Git术语叫做commit），需要将这些更新取回本地，这时就要用到git fetch命令。

git fetch [远程主机名]

上面命令将某个远程主机的更新，全部取回本地。

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

git fetch [远程主机名] [分支名]

如，取回origin主机的master分支。

git fetch origin master

所取回的更新，在本地主机上要用"远程主机名/分支名"的形式读取。比如origin主机的master，就要用origin/master读取。

取回远程主机的更新以后，可以在它的基础上，使用git checkout命令创建一个新的分支。

git checkout -b newBrach origin/master

上面命令表示，在origin/master的基础上，创建一个新分支。

此外，也可以使用git merge命令或者git rebase命令，在本地分支上合并远程分支。

git merge origin/master

或者

git rebase origin/master

上面命令表示在当前分支上，合并origin/master。

### git pull

git pull命令，作用是取回远程主机某个分支的更新，再与本地的指定分支合并。它的完整格式稍稍有点复杂。

$ git pull <远程主机名> <远程分支名>:<本地分支名>

比如，取回origin主机的next分支，与本地的master分支合并，需要写成下面这样。

git pull origin next:master

如果远程分支是与当前分支合并，则冒号后面的部分可以省略。

git pull origin next

上面命令表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再做git merge。

git fetch origin

git merge origin/next

$ git pull origin

上面命令表示，本地的当前分支自动与对应的origin主机"追踪分支"（remote-tracking branch）进行合并。

如果当前分支只有一个追踪分支，连远程主机名都可以省略。

$ git pull

上面命令表示，当前分支自动与唯一一个追踪分支进行合并。

如果合并需要采用rebase模式，可以使用--rebase选项。

$ git pull --rebase <远程主机名> <远程分支名>:<本地分支名>

## 推送数据到远程仓库

项目进行到一个阶段,要同别人分享目前的成果,可以将本地仓库中的数据推送到远程仓库.实现这个任务的命令很简单: git push [remote-name] [branch-name].如果要把本地的 master 分支推送到 origin [服务器](http://server.chinaitlab.com/)上(再次说明下,克隆操作会自动使用默认的 master 和 origin 名字),可以运行下面的命令:

$ git push origin master只有在所克隆的[服务器](http://server.chinaitlab.com/)上有写权限,或者同一时刻没有其他人在推数据,这条命令才会如期完成任务.如果在你推数据前,已经有其他人推送了若干更新,那 你的推送操作就会被驳回.你必须先把他们的更新抓取到本地,并到自己的项目中,然后才可以再次推送.

### git push命令

git push命令用于将本地分支的更新，推送到远程主机。它的格式与git pull命令相仿。

git push <远程主机名> <本地分支名>:<远程分支名>

注意，分支推送顺序的写法是<来源地>:<目的地>，所以git pull是<远程分支>:<本地分支>，而git push是<本地分支>:<远程分支>。

如果省略远程分支名，则表示将本地分支推送与之存在"追踪关系"的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。

$ git push origin master

上面命令表示，将本地的master分支推送到origin主机的master分支。如果后者不存在，则会被新建。

如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支。

$ git push origin :master

# 等同于

$ git push origin --delete master

上面命令表示删除origin主机的master分支。

如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。

$ git push origin

上面命令表示，将当前分支推送到origin主机的对应分支。

如果当前分支只有一个追踪分支，那么主机名都可以省略。

$ git push

如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用git push。

$ git push -u origin master

上面命令将本地的master分支推送到origin主机，同时指定origin为默认主机，后面就可以不加任何参数使用git push了。

不带任何参数的git push，默认只推送当前分支，这叫做simple方式。此外，还有一种matching方式，会推送所有有对应的远程分支的本地分支。Git 2.0版本之前，默认采用matching方法，现在改为默认采用simple方式。如果要修改这个设置，可以采用git config命令。

$ git config --global push.default matching

# 或者

$ git config --global push.default simple

还有一种情况，就是不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要使用--all选项。

$ git push --all origin

上面命令表示，将所有本地分支都推送到origin主机。

如果远程主机的版本比本地版本更新，推送时Git会报错，要求先在本地做git pull合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用--force选项。

$ git push --force origin

上面命令使用--force选项，结果导致远程主机上更新的版本被覆盖。除非你很确定要这样做，否则应该尽量避免使用--force选项。

最后，git push不会推送标签（tag），除非使用--tags选项。

$ git push origin –tags

### push 时常见错误

一般通过git init创建出来的版本库是不允许push 需要修改config里的receive.denyCurrentBranch

git config receive.denyCurrentBranch **ignore**

在初始化远程仓库时最好使用 git--bare init   而不要使用：git init

      如果使用了gitinit初始化，则远程仓库的目录下，也包含worktree，当本地仓库向远程仓库push时,

 如果远程仓库正在push的分支上（如果当时不在push的分支，就没有问题）, 那么push后的结果不会反应在worktree上,  也即在远程仓库的目录下对应的文件还是之前的内容，必须得使用git reset--hard才能看到push后的内容.

一般来说要作为可以push 的一般要创建裸仓库，即在初始化的时候用git init –bare，

使用”git init –bare”方法创建一个所谓的裸仓库，之所以叫裸仓库是因为这个仓库只保存git历史提交的版本信息，而不允许用户在上面进行各种git操作，如果你硬要操作的话，只会得到下面的错误（”This operation must be run in a work tree”）

# 查看历史

## git log

例子

git log --pretty =oneline//在一行显示 git log.

git log --oneline --stat //显示每个文件的变化行数，其中后面跟的--start参数是用来统计哪些文件被改动，有多少行被改动。

git log --oneline --graph，则可以图形化地显示 branch 的变化（方便查看 merge 变化）。git log --until=1.minute.ago // 一分钟之前的所有 log

git log --since=1.day.ago //一天之内的log

git log --since=1.hour.ago //一个小时之内的 log

git log --since=`.month.ago --until=2.weeks.ago //一个月之前到半个月之前的log

git log --since ==2013-08.01 --until=2013-09-07 //某个时间段的 log

git log --pretty=%s –since=2014-08-20//输出2014-08-20之后的commit的内容

format，可以定制要显示的记录格式，这样的输出便于后期编程提取分析，像这样：

$ git log --pretty=format:"%h - %an, %ar : %s"

下表 列出了常用的格式占位符写法及其代表的意义。

选项 说明

%H 提交对象（commit）的完整哈希字串

**%h 提交对象的简短哈希字串**

%T 树对象（tree）的完整哈希字串

%t 树对象的简短哈希字串

%P 父对象（parent）的完整哈希字串

%p 父对象的简短哈希字串

**%an 作者（author）的名字**

%ae 作者的电子邮件地址

%ad 作者修订日期（可以用 -date= 选项定制格式）

%ar 作者修订日期，按多久以前的方式显示

%cn 提交者(committer)的名字

%ce 提交者的电子邮件地址

%cd 提交日期

%cr 提交日期，按多久以前的方式显示

**%s 提交说明**

用 oneline 或 format 时结合 --graph 选项，可以看到开头多出一些 ASCII 字符串表示的简单图形，形象地展示了每个提交所在的分支及其分化衍合情况。

git log 命令支持的选项。表 2-2 还列出了一些其他常用的选项及其释义。

选项 说明

**-p 按补丁格式显示每个更新之间的差异**。

**--stat 显示每次更新的文件修改统计信息**。

--shortstat 只显示 --stat 中最后的行数修改添加移除统计。

--name-only 仅在提交信息后显示已修改的文件清单。

--name-status 显示新增、修改、删除的文件清单。

--abbrev-commit 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。

--relative-date 使用较短的相对时间显示（比如，“2 weeks ago”）。

--graph 显示 ASCII 图形表示的分支合并历史。

--pretty 使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和 format（后跟指定格式）。

还可以给出若干搜索条件，列出符合的提交。用 --author 选项显示指定作者的提交，用 --grep 选项搜索提交说明中的关键字。（请注意，如果要得到同时满足这两个选项搜索条件的提交，就必须用--all-match 选项。）

如果只关心某些文件或者目录的历史提交，可以在 git log 选项的最后指定它们的路径。因为是放在最后位置上的选项，所以用两个短划线（--）隔开之前的选项和后面限定的路径名。

表 2-3 还列出了其他常用的类似选项。

选项 说明

-(n) 仅显示最近的 n 条提交

--since, --after 仅显示指定时间之后的提交。

--until, --before 仅显示指定时间之前的提交。

--author 仅显示指定作者相关的提交。

--committer 仅显示指定提交者相关的提交。

## git reflog

可以查看所有的commit ，reset，cherry-pick，merge，checkout操作

## git blame

查看文件的每一行最后有谁改动

git blame -w # 忽略移除空白这类改动

git blame -M # 忽略移动文本内容这类改动

git blame -C # 忽略移动文本内容到其它文件这类改动

命令格式：

git blame [file\_name]

# Patch

## git format-patch

### pacth生成

使用git format-patch生成所需要的patch:

格式 git format-patch -1 哈希值

git format-patch -M master         // 当前分支所有超前master的提交

git format-patch -s 4e16                // 某次提交以后的所有patch, --4e16指的是SHA1 ID

# git format-patch -1                     //  单次提交

# git format-patch -3                    // 从master往前3个提交的内容，可修改为你想要的数值

# git format-patch –n 07fe            // -n指patch数，07fe对应提交的名称, 某次提交（含）之前的几次提交

# git format-patch -s --root origin     // 从origin到指定提交的所有patch

### 应用patch

git am

格式 git am patch名称

## git diff

作用：查看文件的修改

git diff 查看尚未暂存的文件更新了哪些部分，不加参数直接输入，修改之后还没有暂存起来的变化内容

git diff 哈希值 查看对应节点的文件修改

git diff topic master 或 git diff topic..master 直接将两个分支上最新的提交做diff

 git diff –stat 查看简单的diff结果，可以加上--stat参数

 git diff test 查看当前目录和另外一个分支的差别 ，显示当前目录和另一个叫'test'分支的差别

 git diff HEAD -- ./lib 显示当前目录下的lib目录和上次提交之间的差别（更准确的说是在当前分支下）

git diff HEAD^ HEAD 比较上次提交commit和上上次提交

 git diff SHA1 SHA2 比较两个历史版本之间的差异

git diff生成patch

 git diff test >1.pacth

应用pacth

git apply 1.pacth

## git archive

作用：1.文件归档

格式：git archive 哈希值 --prefix=保存路径 要生成的文件

git archive -o ../updated.zip HEAD $(git diff --name-only HEAD^) 导出最后一次提交修改过的文件

git archive 8e8765 -- prefix=patch/ $(git diff –name-only 8e8765..25601f0.)|tar –xf - 在8e8765节点的状态下将8e8765和25601f0之间不改动过的文件归档保存到patch/下，并解压

例子：

git archive命令，可以对任意提交对应的目录树建立归档。

$ git archive -o latest.zip HEAD  基于最新提交建立归档文件latest.zip

$ git archive -o partial.tar HEAD src doc  只把目录src和doc建立到归档partial.tar中

$ git archive --format=tar --prefix=1.0/ v1.0 | gzip > foo-1.0.tar.gz  基于里程碑v1.0建立归档，并且为归档中的文件添加目录前缀1.0

$git archive 8e8765 --prefix=updated/ $(git diff --name-only -b 7.19 7.20) | tar -xf - 在8e8765 归档7.19～7.20之后变化的文件，保存在updated/并解压

在建立归档时，如果使用树对象ID进行归档，则使用当前时间作为归档中文件的修改时间，而如果使用提交ID或里程碑等，则使用提交建立的时间作为归档中文件的修改时间。

如果使用tar格式建立归档，并且使用提交ID或里程碑ID，还会把提交ID记录在归档文件的文件头中。记录在文件头中的提交ID可以通过git tar-commit-id命令获取。

如果希望在建立归档时忽略某些文件或目录，可以通过为相应文件或目录建立export-ignore属性加以实现。

# **保存当前的工作现场**

适用情况：做了修改后，还没有add commit等等后续工作，现在突然要切换分支做其它事情，默认情况下你在这个分支修改的代码会被带到切换过去的分支中。可以先把你修改的保存起来。这些修改可以再还原过来。

$ git stash -u

$ xxxx 随便你的操作

$ git stash pop

注意:-u是代表是也把添加的新文件（术语是未跟踪）也藏起来，一般是要有这个u的。

# git的一些配置

## 忽略掉文件权限检查

old mode 100755

new mode 100644

让git忽略掉文件权限检查：

git config --add core.filemode false

## 取消设置

eg

$git config --global --unset color.ui "auto" //取消颜色设置

# git使用小技巧

## 修改提交内容

### 修改最后一次提交的内容

## 格式：git commit --amend

eg：

## git commit --amend -m "new message" //如果暂存去里没有新的内容这句话就是重新提交最新节点的备注信息

### 修改前几次提交的内容

git rebase -i 哈希值

这打开的打开文本编辑器，将看到从当前到选定的提交

将将要修改节点前的“pick”改成“edit”，然后保存并退出。将会显示以下内容，修改过的提交呈现退出状态。（如果只要修改节点名称将pick改为reword）

git commit –amend

git rebase –continue

## 批量删除文件时提交的暂存区的方法

git add –A 或git add –all

## 二进制的合并是的冲突解决

对于二进制文件的冲突，你肯定不想通过编辑二进制文件来解决冲突，那是不可能完成的事情。你要做的就是：要么选择对方的修改，要么选择自己的修改。你可以用git checkout的--theirs或--ours选项

git checkout –theirs ：选择对方保留的修改，放弃自己的修改

git checkout –ours：选择自己的修改，放弃对方的修改

# 错误处理

## Error pulling origin: error: The following untracked working tree files would be overwritten by

其实你看错误提示就知道了，是因为一些untracked working tree files引起的问题。所以只要解决了这些untracked的文件就能解决这个问题。

解决

方式二：通过命令行，你直接执行git clean -d -fx即可。