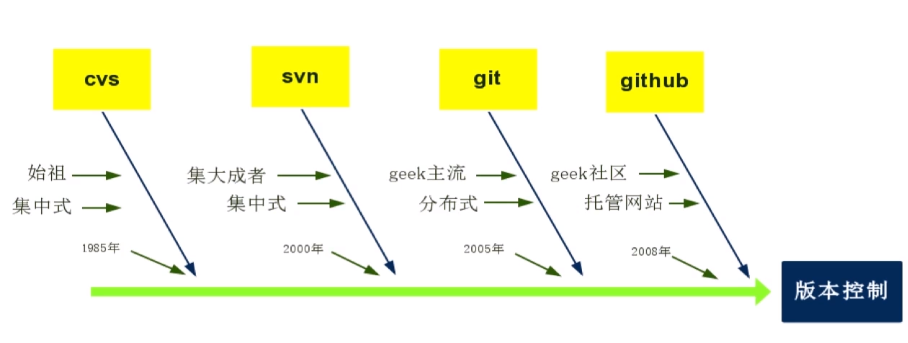
Git多人协作控制

1. 版本管理工具

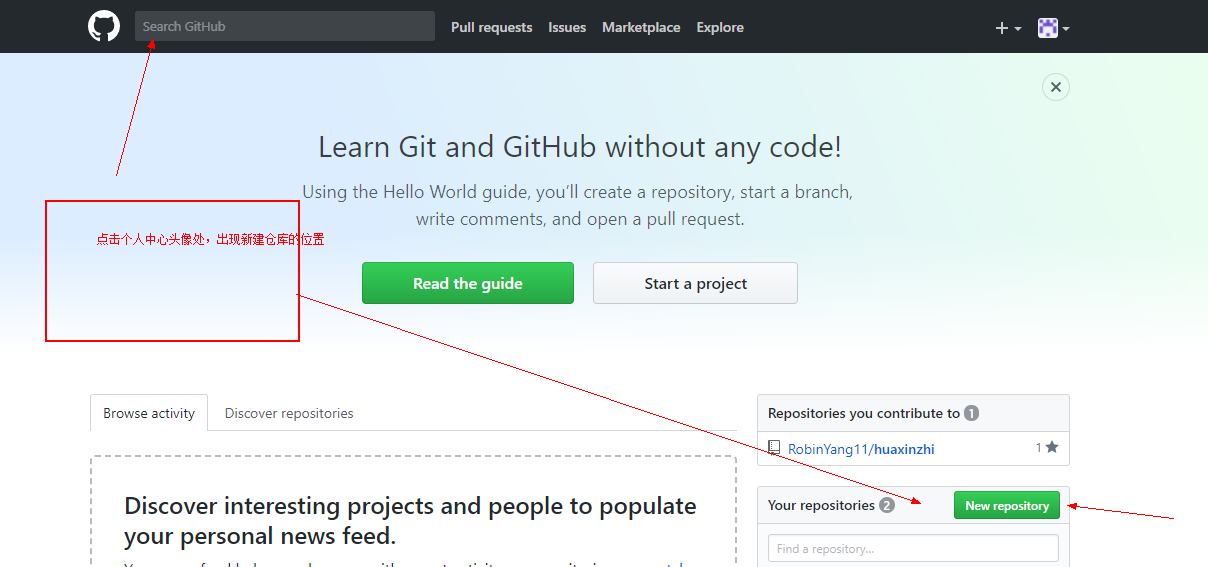


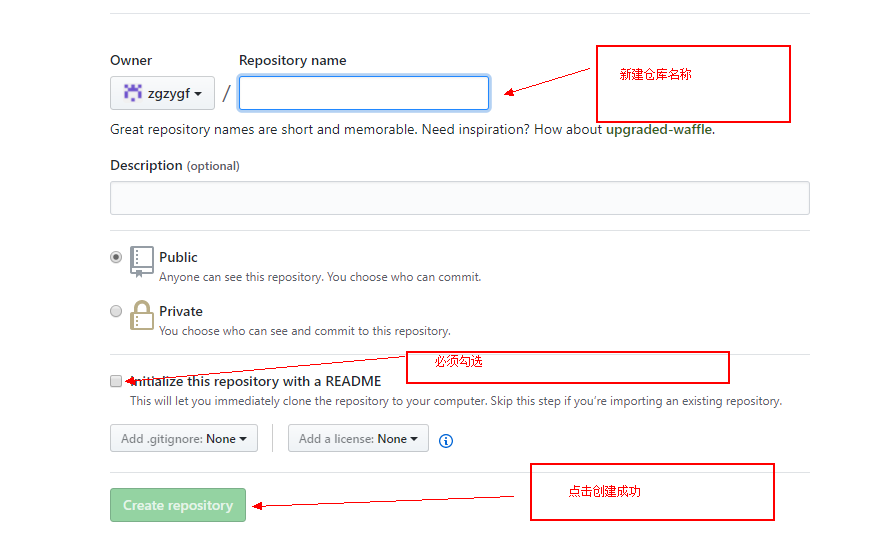
集中式：需要一个中心服务器来放置最新文件，不联网就没法提交没法查看操作等。

分布式：每一个拥有版本库的人都可以在不联网的情况下快速的完成文件的提交查看记录等操作。

GitHub windows下官网下载：<https://desktop.github.com/>

Github 注册：<https://github.com>

1. 在GitHub中新建一个知识库



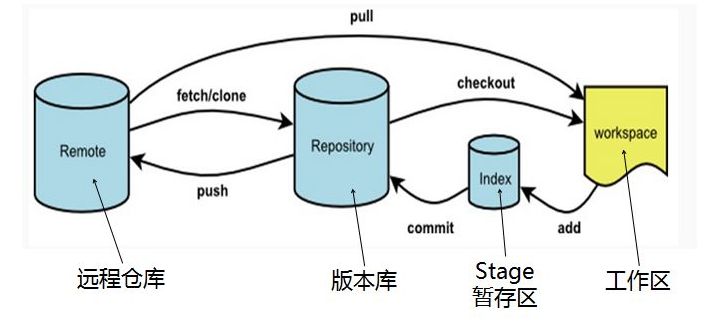
git 工作流程

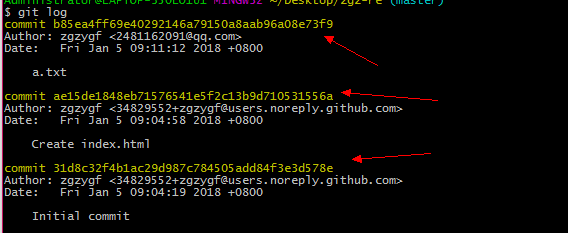
图 1 git工作流程图

1. 从远程仓库把需要的项目pull(clone)下来，工作区结束后add操作把文件添加到stage(暂存区)，【可以使用 $ git reset 撤出暂存区】，然后可以git status查看提交的文件状态，确认提交的文件全部进入暂存区就可以进行git commit提交了，有冲突的就在对应位置修改，修改后把修改的文件git add 然后再提交，以上步骤结束无误成功之后把文档版本库整个push(推到)远程仓库就ok。
2. 开发时在创建的这个库里面复制地址然后自己新建文件夹，然后开启命令，把地址colne到本地文件夹开发。
3. $ cd zgz-re(创建的仓库名)， 添加到仓库， $ git pul 从仓库把项目拉下来， $ git fecth 取回所有分之的更新。l不能在主分支上开发，需要自己新建一个分支（或者在自己已有的分支）开发。$ git branch zgz新建分支，$ git checkout zgz更改到分支，更改到分支之后再 $ git pull +远程地址，把项目取下来，然后进行开发，开发后也可以$ git diff对比查看不同之处（此命令比较的是工作目录(Working tree)和暂存区域快照(index)之间的差异，也就是修改之后还没有暂存起来的变化内容。），之后就可以进行$ git add \*(也可$ git add 改动的文件)把文件放入暂存区，然后查看更改状态$ git status ， 再进行$ git commit提交文件到版本库，要先切换到主分支，然后进行$ git merge+自己开发的分支，合并分支操作，再$ git push 到远程仓库。
4. $ git log 可以查看提交历史，以便确定要回退到哪个版本。

要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

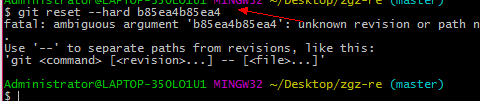
上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit\_id。



回退到上个版本 $ git reset --hard HEAD^

再回退到最初的版本 $ git reset --hard commit\_id



输入前几位就可以 $ git reset --hard b85ea

6.$ git checkout -- readme.txt

命令git checkout -- readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销，这里有两种情况：

一种是readme.txt自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是readme.txt已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次git commit或git add时的状态。

#### 7.删除文件

在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git并且提交：

$ git add test.txt

$ git **commit** -m "add test.txt"

一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：

$ rm test.txt

这个时候，Git知道你删除了文件，因此，工作区和版本库就不一致了，git status命令会立刻告诉你哪些文件被删除了：

$ git status

*deleted: test.txt#*

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

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：

$ git rm test.txt

rm 'test.txt'

$ git **commit** -m "remove test.txt"

[master d17efd8] remove test.txt

1 file changed, 1 deletion(-)

**delete** mode 100644 test.txt

现在，文件就从版本库中被删除了。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

$ git checkout -- test.txt

8.提交中，分支切换问题

首先，我们创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$git checkout dev

Switched to branch 'dev'

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

git branch命令会列出所有分支，当前分支前面会标一个\*号。

然后，我们就可以在dev分支上正常提交，比如对readme.txt做个修改，加上一行：

Creating a **new** branch is quick.

然后提交：

$ git add readme.txt

$ git commit -m "branch test"

现在，dev分支的工作完成，我们就可以切换回master分支：

$ git checkout master

Switched to branch 'master'

切换回master分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：



现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev

Updating d17efd8..fec145a

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

合并完成后，就可以放心地删除dev分支了：

$ git branch -d dev

Deleted branch dev (was fec145a).

删除后，查看branch，就只剩下master分支了：

$ git branch

\* master

因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

#### 创建与合并分支

在[版本回退](https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000" \t "https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/_blank)里，已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长：

当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：



你看，Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

不过，从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支：



真是太神奇了，你看得出来有些提交是通过分支完成的吗？

下面开始实战。

首先，我们创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

git branch命令会列出所有分支，当前分支前面会标一个\*号。

然后，我们就可以在dev分支上正常提交，比如对readme.txt做个修改，加上一行：

Creating a **new** branch is quick.

然后提交：

$ git add readme.txt

$ git commit -m "branch test"

[dev fec145a] branch test

1 file changed, 1 insertion(+)

现在，dev分支的工作完成，我们就可以切换回master分支：

$ git checkout master

Switched to branch 'master'

切换回master分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：



现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev

Updating d17efd8..fec145a

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

### 小结

Git鼓励大量使用分支：

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

创建+切换分支：git checkout -b <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

补充：

git commit -m ""  只会提交添加到缓存区的文件（只提交添加的）

git commit -a -m ""  能提交修改过，但是没有添加到缓存区的文件（修改过的就能提交）

使用命令：git log  能查看提交历史，后面加上  --pretty=oneline  能使内容单行显示

使用命令：git status  能让我们时刻掌握仓库当前的状态

<http://blog.csdn.net/dengdengda/article/details/50903176>

多人开发创建分支合并中的冲突问题：

我们都知道git是一个分布式的代码管理工具，因为是分布式每个人都可以在自己的本地仓库进行操作以及提交，最后可以push到remote repository。但是我们在push之前，应该pull一下，因为如果有相同的文件修改了会出现问题，其实git pull 这个操作还是不太好的，因为他不够直观

git pull //相当于下面两个操作

git fetch

git merge

git fetch 会把此分支上面所有的更新拉下来，但是没有和本地的Head分支进行和合并。

如果执行git fetch后再去执行 git merge就会把更新的部门和当前的Head进行合并，如果有冲突就会产生冲突文件，然后把冲突文件修改一下就可以了。

#### 探索

我们用本地的一个仓库去模拟一下git merge产生冲突时候的情况。

git init //初始化一个本地仓库echo "master" > info.txt

git add .

git commit -m "master"//在上面master的基础上创建两个新的分支

git branch issue1 //创建一个issue1 本地分支

git branch issue2 //创建一个issue2 本地分支

这里要注意点，这两个分支都是从master分支产生的，此时这三个分支都是在一个点，对于git来说，这三个分支的指针是一样的，这里需要支出的是HEAD指的就是当前的分支，每当我们git checkout 分支的时候，HEAD也会自动的修改指向切换后的分支。

我们可以这样去查看HEAD:

cat .git/HEAD

ref: refs/heads/master //输出

接下来进入正题，首先切换到issue1分支:

git checkout issue1echo "issue1" >> info.txt

git add .

git commit -m "issue1"

//下面是issue1操作结果

[issue1 4e5bb48] issue1

1 file changed, 1 insertion(+)

然后切换到issue2:

git checkout issue2echo "issue2" >> info.txt

git add .

git commit -m "issue2"

//下面是issue2的操作

[issue2 bcf0fb0] issue2

1 file changed, 1 insertion(+)

然后我们再切换到master分支把issue1和issue2merge到master分支,不过在我们merge之前我们想思考一下，假设我们先merge issue1分支，然后再merge issue2分支会出现什么样的结果呢?

因为issue1 和 issue2都是从master分支创建出来的，所以master分支merge issue1的时候是可以成功的，因为在master分支在merge issue1的时候，master分支并没有改动过，但在master　merge issue2的时候，就会有问题，因为此时master以及修改了，而且修改的地方还都存在冲突，冲突就是第二行。

master合并issue1分支:

Updating e84213a..4e5bb48

Fast-forward

info.txt | 1 +

1 file changed, 1 insertion(+)

上面的Fast-forward指的就是把master当前的指针直接移动到issue1　也就是ｍaster此时和issue1指向一个位置。

master再合并issue2分支:

Auto-merging info.txt

CONFLICT (content): Merge conflict in info.txt

Automatic merge failed; fix conflicts and then commit the result.

从上面可以看出，出现了冲突。

cat info.txt//结果如下

master

<<<<<<< HEAD //HEAD到分界线是当前分支的(master)

issue1

======= //这个是分界线

issue2

>>>>>>> issue2　　//分界线到此处是合并分支的(issue2)

因为第二行出现了冲突,master merge issue1后第二行为issue1

然后此时merge issue2的时候　第二行为issue2，所以出现了冲突，

修改一下，把刚修改冲突的文件git add+文件名，然后提交（git commit）就可以了。

树冲突  
  
文件名修改造成的冲突，称为树冲突。  
比如，a用户把文件改名为a.c，b用户把同一个文件改名为b.c，那么b将这两个commit合并时，会产生冲突。  
$ git status  
 added by us: b.c  
 both deleted: origin-name.c  
 added by them: a.c  
如果最终确定用b.c，那么解决办法如下：  
git rm a.c  
git rm origin-name.c  
git add b.c  
git commit  
执行前面两个git rm时，会告警“file-name : needs merge”，可以不必理会。

补充：

查看自带的版本：git --version

查询环境变量的设定：$ git config user name



查看已有的配置信息：$ git config --list

git stash把当前的工作暂存起来，然后切换到主分支master,再新建一个分支做紧急的事，做完后，切换回之前工作的分支（git checkout 分知名），使用git stash list查看工作现场存放到哪里了，

$ git stash list

stash@{0}: WIP on dev: 6224937 add merge

然后用git stash apply恢复到原来位置。

$ git stash apply stash@{0}

另一种方式是用git stash pop，恢复的同时把stash内容也删了：

Reset:版本回退。 git reset --hard commit\_id

git fetch ：会把此分支上面所有的更新拉下来（当前环境主分支），但是没有和本地的Head分支进行和合并（新建的开发工作的子分支）。

### **git diff**

执行 git diff 来查看执行 git status 的结果的详细信息。

git diff 命令显示已写入缓存与已修改但尚未写入缓存的改动的区别。git diff 有两个主要的应用场景。

尚未缓存的改动：**git diff**

查看已缓存的改动： **git diff --cached**

查看已缓存的与未缓存的所有改动：**git diff HEAD**

显示摘要而非整个 diff：**git diff --stat**

**Git mv**

git mv 命令用于移动或重命名一个文件、目录、软连接。

我们先把刚移除的 README 添加回来：

$ git add README

然后对其重名:

$ git mv README README.md