Git学习笔记

安装Git

1.使用yum安装

[root@docker ~]# yum install git

2.使用源码安装

安装依赖环境

[root@docker ~]# yum install curl-devel expat-devel gettext-devel \

openssl-devel zlib-devel

下载源码包

[root@docker ~]# wget https://mirrors.edge.kernel.org/pub/software/scm/git/[git-2.12.2.tar.gz](https://mirrors.edge.kernel.org/pub/software/scm/git/git-2.12.2.tar.gz)

进行编译安装

[root@docker ~]# tar -zxf git-2.0.0.tar.gz

[root@docker ~]# cd git-2.0.0

[root@docker ~]# make configure

[root@docker ~]# ./configure --prefix=/usr

[root@docker ~]# make all doc info

[root@docker ~]# sudo make install install-doc install-html install-info

查看是否安装成功

[root@docker ~]# git --version

初始化Git

当安装完 Git 应该做的第一件事就是设置你的用户名称与邮件地址。 这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到你的每一次提交中，不可更改：

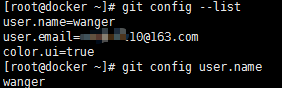
$ git config --global user.name "wanger"

$ git config --global user.email [wanger@163.com](mailto:wanger@163.com)

检查配置信息

$ git config –list //列出所有配置信息

也可以通过输入 git config <key>： 来检查 Git 的某一项配置



获取 Git 仓库

有两种取得 Git 项目仓库的方法。 第一种是在现有项目或目录下导入所有文件到 Git 中； 第二种是从一个服务器克隆一个现有的 Git 仓库。

**在现有目录中初始化仓库**

如果你打算使用 Git 来对现有的项目进行管理，你只需要进入该项目目录并输入：

git init

该命令将创建一个名为 .git 的子目录，这个子目录含有你初始化的 Git 仓库中所有的必须文件，这些文件是 Git 仓库的骨干。

**克隆现有的仓库**

如果你想获得一份已经存在了的 Git 仓库的拷贝，比如说，你想为某个开源项目贡献自己的一份力，这时就要用到 git clone 命令。 如果你对其它的 VCS 系统（比如说Subversion）很熟悉，请留心一下你所使用的命令是"clone"而不是"checkout"。 这是 Git 区别于其它版本控制系统的一个重要特性，Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。 当你执行 git clone 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。

克隆仓库的命令格式是 git clone [url]，比如，要克隆 Git 的可链接库 libgit2，可以用下面的命令：

$ git clone https://github.com/libgit2/libgit2

这会在当前目录下创建一个名为 “libgit2” 的目录，并在这个目录下初始化一个 .git 文件夹，从远程仓库拉取下所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。

 如果你想在克隆远程仓库的时候，自定义本地仓库的名字，你可以使用如下命令：

$ git clone https://github.com/libgit2/libgit2 mylibgit

这将执行与上一个命令相同的操作，不过在本地创建的仓库名字变为 mylibgit。

**记录每次更新到仓库**

**检查当前文件状态**

要查看哪些文件处于什么状态，可以用 git status 命令。 如果在克隆仓库后立即使用此命令，会看到类似这样的输出：



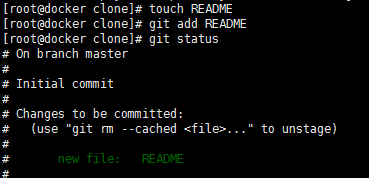
这说明你现在的工作目录相当干净。换句话说，所有已跟踪文件在上次提交后都未被更改过。

**跟踪新文件**

使用命令 git add 开始跟踪一个文件。 所以，要跟踪 README 文件，运行：

$ git add README

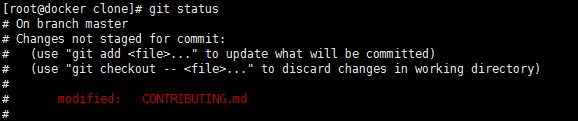
此时再运行 git status 命令，会看到 README 文件已被跟踪，并处于暂存状态



只要在 Changes to be committed 这行下面的，就说明是已暂存状态。 如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。 你可能会想起之前我们使用 git init 后就运行了 git add (files) 命令，开始跟踪当前目录下的文件。 git add 命令使用文件或目录的路径作为参数；如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。

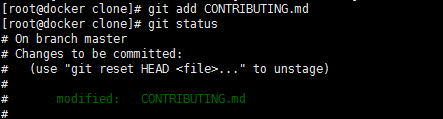
**暂存已修改文件**

现在我们来修改一个已被跟踪的文件。 如果你修改了一个名为 CONTRIBUTING.md 的已被跟踪的文件，然后运行 git status 命令，会看到下面内容：

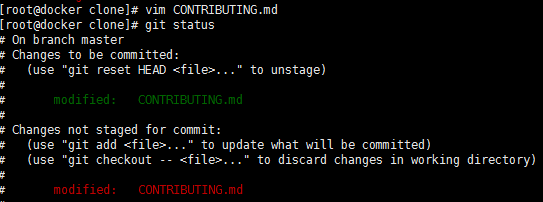


文件 CONTRIBUTING.md 出现在 Changes not staged for commit 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。 要暂存这次更新，需要运行 git add 命令。这是个多功能命令：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等。

我们运行 git add 将"CONTRIBUTING.md"放到暂存区，然后再看看 git status 的输出：



现在重新编辑CONTRIBUTING.Md文件，在使用git status命令查看



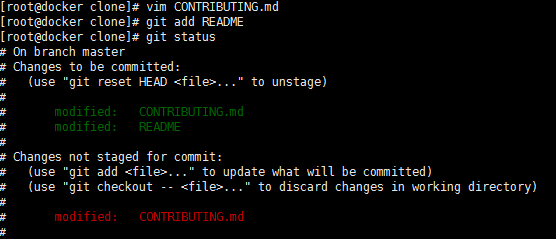
 现在 CONTRIBUTING.md 文件同时出现在暂存区和非暂存区。绿色的CONTRIBUTING.md在暂存区里，代表你运行 git add 命令时的版本，红色的CONTRIBUTING.md表示最近的一次修改，但还未使用git add将其放入暂存区

git status 命令的输出十分详细，但其用语有些繁琐。 如果你使用 git status -s 命令或 git status --short 命令，你将得到一种更为紧凑的格式输出。

**查看已暂存和未暂存的修改**

如果 git status 命令的输出对于你来说过于模糊，你想知道具体修改了什么地方，可以用 git diff 命令。 尽管 git status 已经通过在相应栏下列出文件名的方式回答了这个问题，git diff 将通过文件补丁的格式显示具体哪些行发生了改变。

假如再次修改 README 文件后暂存，然后编辑 CONTRIBUTING.md 文件后先不暂存， 运行 status 命令将会看到：



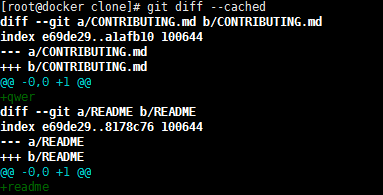
之前在暂存区里的CONTRIBUTING.md文件是我之前修改的

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 git diff：



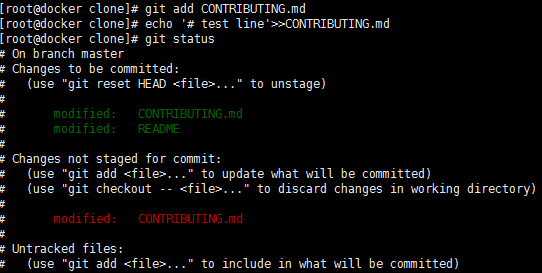
此命令比较的是修改之后还没有暂存起来的变化内容。可以看到我在里面添加了1234

若要查看已暂存的将要添加到下次提交里的内容，可以用 git diff --cached 命令。（Git 1.6.1 及更高版本还允许使用 git diff --staged，效果是相同的，但更好记些。）

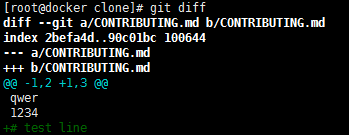


请注意，git diff 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。 所以有时候你一下子暂存了所有更新过的文件后，运行 git diff 后却什么也没有，就是这个原因。

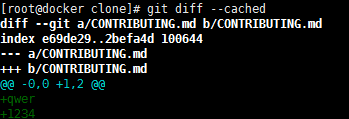
像之前说的，暂存 CONTRIBUTING.md 后再编辑，运行 git status 会看到暂存前后的两个版本。 如果我们的环境（终端输出）看起来如下：



现在运行 git diff 看暂存前后的变化：



然后用 git diff --cached 查看已经暂存起来的变化：

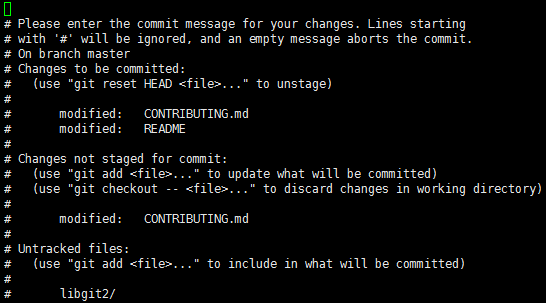


**提交更新**

现在的暂存区域已经准备妥当可以提交了。 在此之前，请一定要确认还有什么修改过的或新建的文件还没有 git add 过，否则提交的时候不会记录这些还没暂存起来的变化。 这些修改过的文件只保留在本地磁盘。 所以，每次准备提交前，先用 git status 看下，是不是都已暂存起来了， 然后再运行提交命令 git commit：

这种方式会启动文本编辑器以便输入本次提交的说明。 (默认会启用 shell 的环境变量 $EDITOR 所指定的软件，一般都是 vim 或 emacs。当然也可以使用 git config --global core.editor 命令设定你喜欢的编辑软件。）

可以看到，默认的提交消息包含最后一次运行 git status 的输出，放在注释行里，另外开头还有一空行，供你输入提交说明。 你完全可以去掉这些注释行，不过留着也没关系，多少能帮你回想起这次更新的内容有哪些。 (如果想要更详细的对修改了哪些内容的提示，可以用 -v 选项，这会将你所做的改变的 diff 输出放到编辑器中从而使你知道本次提交具体做了哪些修改。） 退出编辑器时，Git 会丢掉注释行，用你输入提交附带信息生成一次提交。



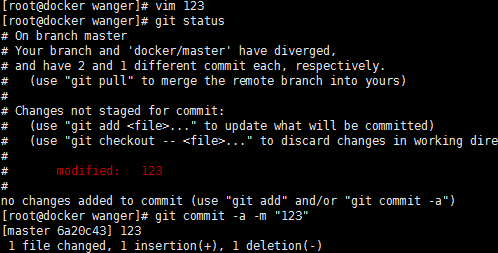
另外，你也可以在 commit 命令后添加 -m 选项，将提交信息与命令放在同一行，如下所示：



可以看到，提交后它会告诉你，当前是在哪个分支（master）提交的，本次提交的完整 SHA-1 校验和是什么（2c79c89），以及在本次提交中，有多少文件修订过，多少行添加和删改过。

**跳过使用暂存区域**

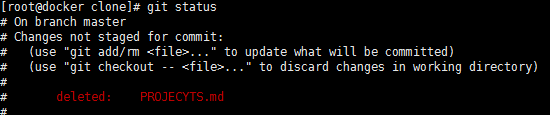
尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显繁琐。 Git 提供了一个跳过使用暂存区域的方式， 只要在提交的时候，给 git commit 加上 -a 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 git add 步骤：



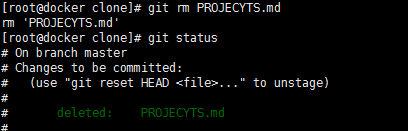
**移除文件**

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除（确切地说，是从暂存区域移除），然后提交。 可以用 git rm 命令完成此项工作，并连带从工作目录中删除指定的文件，这样以后就不会出现在未跟踪文件清单中了。

如果只是简单地从工作目录中手工删除文件，运行 git status 时就会在 “Changes not staged for commit” 部分（也就是 *未暂存清单*）看到：

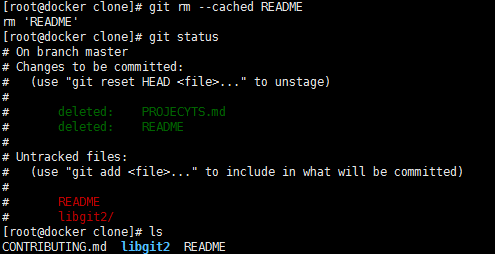


然后再运行 git rm 记录此次移除文件的操作：



下一次提交时，该文件就不再纳入版本管理了。 如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 -f（译注：即 force 的首字母）。 这是一种安全特性，用于防止误删还没有添加到快照的数据，这样的数据不能被 Git 恢复。

另外一种情况是，我们想把文件从 Git 仓库中删除（亦即从暂存区域移除），但仍然希望保留在当前工作目录中。 换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪。 当你忘记添加 .gitignore 文件，不小心把一个很大的日志文件或一堆 .a 这样的编译生成文件添加到暂存区时，这一做法尤其有用。 为达到这一目的，使用 --cached 选项：



可以看到，README文件已经从Git仓库中删除了，但还存在于工作目录中

git rm 命令后面可以列出文件或者目录的名字，也可以使用 glob 模式。 比方说：

$ git rm log/\\*.log

注意到星号 \* 之前的反斜杠 \， 因为 Git 有它自己的文件模式扩展匹配方式，所以我们不用 shell 来帮忙展开。 此命令删除 log/ 目录下扩展名为 .log 的所有文件。

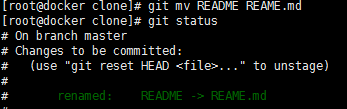
**移动文件**

不像其它的 VCS 系统，Git 并不显式跟踪文件移动操作。 如果在 Git 中重命名了某个文件，仓库中存储的元数据并不会体现出这是一次改名操作。 不过 Git 非常聪明，它会推断出究竟发生了什么

既然如此，当你看到 Git 的 mv 命令时一定会困惑不已。 要在 Git 中对文件改名，可以这么做：

$ git mv file\_from file\_to

它会恰如预期般正常工作。 实际上，即便此时查看状态信息，也会明白无误地看到关于重命名操作的说明：



其实，运行 git mv 就相当于运行了下面三条命令：

$ mv README README.md

$ git rm README

$ git add README.md

如此分开操作，Git 也会意识到这是一次改名，所以不管何种方式结果都一样。 两者唯一的区别是，mv 是一条命令而另一种方式需要三条命令，直接用 git mv 轻便得多。

**查看提交历史**

在提交了若干更新，又或者克隆了某个项目之后，你也许想回顾下提交历史。 完成这个任务最简单而又有效的工具是 git log 命令。

默认不用任何参数的话，git log 会按提交时间列出所有的更新，最近的更新排在最上面。 正如你所看到的，这个命令会列出每个提交的 SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明。

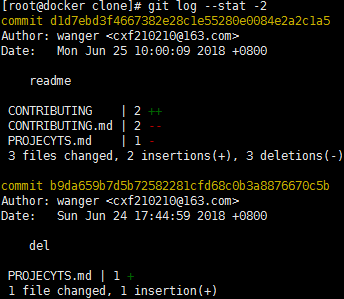
git log 有许多选项可以帮助你搜寻你所要找的提交， 接下来我们介绍些最常用的。

一个常用的选项是 -p，用来显示每次提交的内容差异。 你也可以加上 -2 来仅显示最近两次提交：

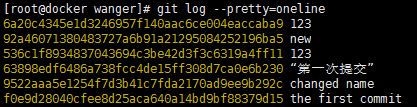


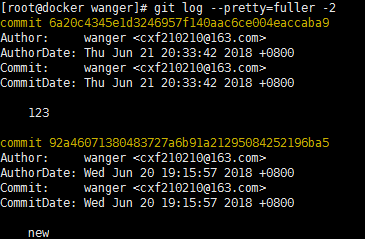
该选项除了显示基本信息之外，还附带了每次 commit 的变化。 当进行代码审查，或者快速浏览某个搭档提交的 commit 所带来的变化的时候，这个参数就非常有用了。 你也可以为 git log 附带一系列的总结性选项。 比如说，如果你想看到每次提交的简略的统计信息，你可以使用 --stat 选项：

正如你所--stat看到的，选项在每次提交的下面列出所有被修改过的文件，有多少文件被修改了以及被修改过的文件的哪些行被移除或是添加了。在每次提交的最后还有一个总结。



另外一个常用的选项是--pretty。这个选项可以指定使用不同于默认格式的方式展示提交历史。这个选项有一些内部的子选项供你使用。oneline比如用将每个提交放在一行显示，查看的提交数很大时非常有用。另外还有short，full和fuller可以用，展示的信息或多或少有些不同，请自己动手实践一下看看效果如何。





但最有意思的是格式，可以定制显示的记录格式。这样的输出对后期提取分析格外有用 ，而且输出的格式不会随着Git的更新而发生改变：

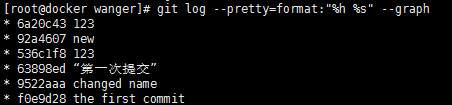


这里列一下git log --pretty=format常用的选项

| **选项** | **说明** |
| --- | --- |
| %H | 提交对象（提交）的完整哈希字串 |
| %h | 提交对象的简短哈希字串 |
| %T | 树对象（树）的完整哈希字串 |
| %t | 树对象的简短哈希字串 |
| %P | 父对象（父）的完整哈希字串 |
| %p | 父对象的简短哈希字串 |
| %an | 作者（作者）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（提交者）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 |

这里稍微解释一下作者和提交者之间的细微差别，作者指的是实际作出修改的人，提交者指的是最后将此工作成果提交到仓库的人。所以，当你为某个项目发布补丁，然后某个核心成员将你的补丁并入项目时，你就是作者，而那核心成员就是提交者。

当oneline或format与另一个log选项--graph结合使用时尤其有用。这个选项添加了一些ASCII字符串来形象地展示你的分支，合并历史：



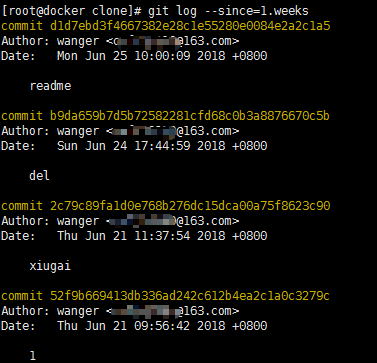
以上只是简单介绍了一些git log命令支持的选项。这里列出一下git log常用的选项

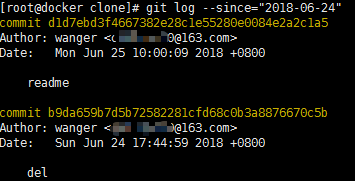
| **选项** |  | **说明** |
| --- | --- | --- |
| -p |  | 按补丁格式显示每个更新之间的差异。 |
| --stat |  | 显示每次更新的文件修改统计信息。 |
| --shortstat |  | 只显示--stat中最后的行数修改添加移除统计。 |
| --name-only |  | 仅在提交信息后显示已修改的文件清单。 |
| --name-status |  | 显示新增，修改，删除的文件清单。 |
| --abbrev-commit |  | 仅显示SHA-1的前几个字符，而非所有的40个字符。 |
| --relative-date |  | 使用较短的相对时间显示（比如，“2周前”）。 |
| --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）。 |

**限制输出长度**

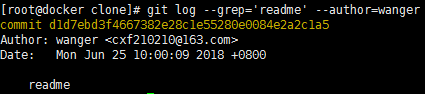
除了定制输出格式的选项之外，git log还有许多非常实用的限制输出长度的选项，也就是只输出部分提交信息。之前你已经看过过-2了，它只显示最近的两条提交，实际上，这是-<n>选项的写法，其中的n可以是任何整数，表示仅显示最近的若干条提交。另外还有按照时间作限制的选项，比如--since和--until也很有用。例如，下面的命令列出所有最近一周内的提交：

这个命令可以在多种格式下工作，比如说具体的某一天"2018-06-24"，或者是相对地多久以前"2 years 1 day 3 minutes ago"。





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，必须就用--all-match选项对话。否则，满足任意一个条件的提交都会被匹配出来）



另一个非常有用的筛选选项是-S，可以列出那些添加或移除某些字符串的提交。比如说，你想找出添加或移除某某个特定函数的引用的提交，你可以这样使用：$ git log -Sfunction\_name

最后一个很实用的git log选项是路径（路径），如果只关心某些文件或目录的历史提交，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。

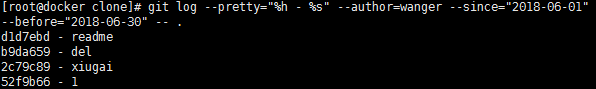
这里列出限制git log输出的常用选项：

|  |  |
| --- | --- |
| 选项 | 说明 |
| -(n) | 仅显示最近的n条提交 |
| --since， --afte | 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。 |
| --grep | 仅显示含指定关键字的提交 |
| -S | 仅显示添加或移除了某个关键字的提交 |

来看一个实际的例子，如果要查看Git仓库中，2018年6月期间，wanger提交的但未合并的测试文件，可以用下面的查询命令：

git log --pretty="%h - %s" --author=wanger --since="2018-06-01" \

--before="2018-06-30" -- .

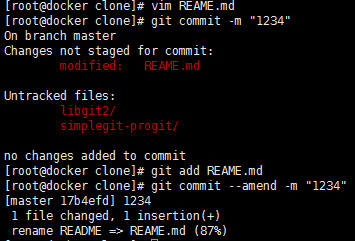


**撤消操作**

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 --amend 选项的提交命令尝试重新提交：

$ git commit --amend

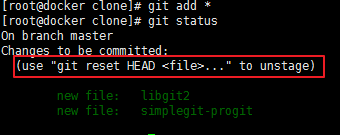
这个命令会将暂存区中的文件提交。 如果自上次提交以来你还未做任何修改（例如，在上次提交后马上执行了此命令），那么快照会保持不变，而你所修改的只是提交信息



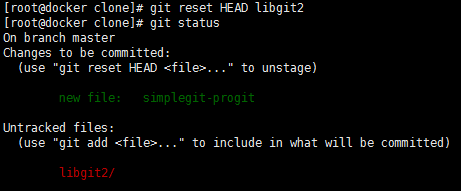
最终你只会有一个提交 - 第二次提交将代替第一次提交的结果。

**取消暂存的文件**

你已经修改了两个文件并且想要将它们作为两次独立的修改提交，但是却意外地输入了 git add \* 暂存了它们两个。 如何只取消暂存两个中的一个呢？ git status 命令提示了你：

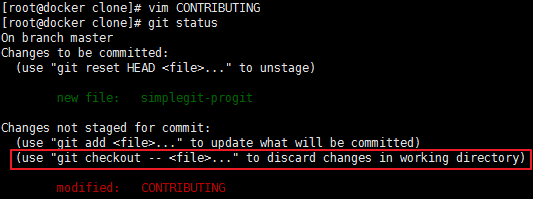


接下来使用命令“git reset HEAD libgit2”取消暂存文件libgit2

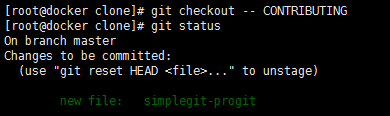


**撤消对文件的修改**

如果你并不想保留对 CONTRIBUTING文件的修改怎么办？ 你该如何方便地撤消修改 - 将它还原成上次提交时的样子（或者刚克隆完的样子，或者刚把它放入工作目录时的样子）？ 幸运的是，git status 也告诉了你应该如何做。 在最后一个例子中，未暂存区域是这样：



接下来使用命令“git checkout -- CONTRIBUTING”来撤销之前所做的修改



**远程仓库的使用**

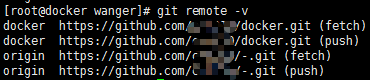
为了能在任意 Git 项目上协作，你需要知道如何管理自己的远程仓库。 远程仓库是指托管在因特网或其他网络中的你的项目的版本库。 你可以有好几个远程仓库，通常有些仓库对你只读，有些则可以读写。 与他人协作涉及管理远程仓库以及根据需要推送或拉取数据。 管理远程仓库包括了解如何添加远程仓库、移除无效的远程仓库、管理不同的远程分支并定义它们是否被跟踪等等。

**查看远程仓库**

如果想查看你已经配置的远程仓库服务器，可以运行 git remote 命令。 它会列出你指定的每一个远程服务器的简写。 如果你已经克隆了自己的仓库，那么至少应该能看到 origin - 这是 Git 给你克隆的仓库服务器的默认名字：

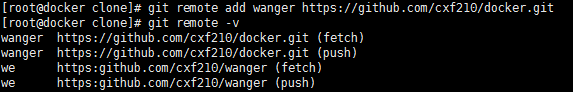


你也可以指定选项 -v，会显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。



**添加远程仓库**

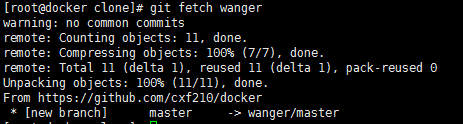
运行 git remote add <shortname> <url> 添加一个新的远程 Git 仓库，同时指定一个你可以轻松引用的简写：



**从远程仓库中抓取与拉取**

$ git fetch [remote-name]

这个命令会访问远程仓库，从中拉取所有你还没有的数据。 执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。

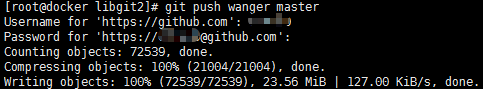


如果你使用 clone 命令克隆了一个仓库，命令会自动将其添加为远程仓库并默认以 “origin” 为简写。 所以，git fetch origin 会抓取克隆（或上一次抓取）后新推送的所有工作。 必须注意 git fetch 命令会将数据拉取到你的本地仓库 - 它并不会自动合并或修改你当前的工作。 当准备好时你必须手动将其合并入你的工作。

如果你有一个分支设置为跟踪一个远程分支，可以使用 git pull命令来自动的抓取然后合并远程分支到当前分支。

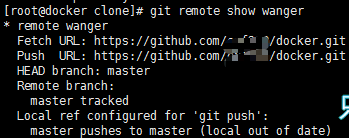
**推送到远程仓库**

当你想分享你的项目时，必须将其推送到上游。 这个命令很简单：git push [remote-name] [branch-name]。 当你想要将 master 分支推送到 wanger服务器时（再次说明，克隆时通常会自动帮你设置好那两个名字），那么运行这个命令就可以将你所做的备份到服务器(这里我将自己从GitHub上clone的仓库推送到远程仓库名为clone的仓库)：



**查看远程仓库**

如果想要查看某一个远程仓库的更多信息，可以使用 git remote show [remote-name] 命令。 如果想以一个特定的缩写名运行这个命令，例如 wanger，会得到像下面类似的信息：



**远程仓库的移除与重命名**

如果因为一些原因想要移除一个远程仓库 - 你已经从服务器上搬走了或不再想使用某一个特定的镜像了，又或者某一个贡献者不再贡献了 - 可以使用 git remote rm

如果想要重命名引用的名字可以运行 git remote rename 去修改一个远程仓库的简写名。 例如，想要将 wanger重命名为 we，可以用 git remote rename 这样做：

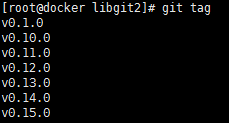


**打标签**

像其他版本控制系统（VCS）一样，Git 可以给历史中的某一个提交打上标签，以示重要。 比较有代表性的是人们会使用这个功能来标记发布结点（v1.0 等等）。

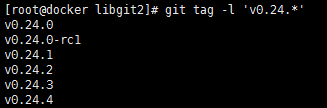
**列出标签**

在 Git 中列出已有的标签是非常简单直观的。 只需要输入 git tag：



这个命令以字母顺序列出标签；但是它们出现的顺序并不重要。

你也可以使用特定的模式查找标签。 例如，Git 自身的源代码仓库包含标签的数量超过 500 个。 如果只对 0.24系列感兴趣，可以运行：



**创建标签**

Git 使用两种主要类型的标签：轻量标签（lightweight）与附注标签（annotated）。

一个轻量标签很像一个不会改变的分支 - 它只是一个特定提交的引用。

然而，附注标签是存储在 Git 数据库中的一个完整对象。 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；还有一个标签信息；并且可以使用 GNU Privacy Guard （GPG）签名与验证。 通常建议创建附注标签，这样你可以拥有以上所有信息；但是如果你只是想用一个临时的标签，或者因为某些原因不想要保存那些信息，轻量标签也是可用的。

**附注标签**

在 Git 中创建一个附注标签是很简单的。 最简单的方式是当你在运行 tag 命令时指定 -a 选项：



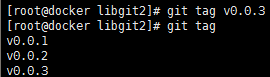
-m 选项指定了一条将会存储在标签中的信息。 如果没有为附注标签指定一条信息，Git 会运行编辑器要求你输入信息。

通过使用 git show 命令可以看到标签信息与对应的提交信息：



**轻量标签**

另一种给提交打标签的方式是使用轻量标签。 轻量标签本质上是将提交校验和存储到一个文件中 - 没有保存任何其他信息。 创建轻量标签，不需要使用 -a、-s 或 -m 选项，只需要提供标签名字：

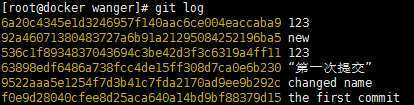


这时，如果在标签上运行 git show，你不会看到额外的标签信息。 命令只会显示出提交信息：

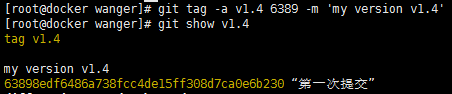


**后期打标签**

你也可以对过去的提交打标签。 假设提交历史是这样的：

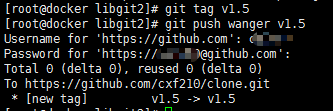


现在，假设在 v1.2 时你忘记给项目打标签，也就是在 “第一次提交” 提交。 你可以在之后补上标签。 要在那个提交上打标签，你需要在命令的末尾指定提交的校验和（或部分校验和）:



**共享标签**

默认情况下，git push 命令并不会传送标签到远程仓库服务器上。 在创建完标签后你必须显式地推送标签到共享服务器上。 这个过程就像共享远程分支一样 - 你可以运行 git push origin [tagname]。



如果想要一次性推送很多标签，也可以使用带有 --tags 选项的 git push 命令。 这将会把所有不在远程仓库服务器上的标签全部传送到那里。

**检出标签**

在 Git 中你并不能真的检出一个标签，因为它们并不能像分支一样来回移动。 如果你想要工作目录与仓库中特定的标签版本完全一样，可以使用 git checkout -b [branchname] [tagname] 在特定的标签上创建一个新分支：



当然，如果在这之后又进行了一次提交，version2 分支会因为改动向前移动了，那么 version2 分支就会和 v2.0.0 标签稍微有些不同，这时就应该当心了。

**Git 别名**

Git 并不会在你输入部分命令时自动推断出你想要的命令。 如果不想每次都输入完整的 Git 命令，可以通过 git config 文件来轻松地为每一个命令设置一个别名。 这里有一些例子你可以试试：

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

这意味着，当要输入 git commit 时，只需要输入 git ci。 随着你继续不断地使用 Git，可能也会经常使用其他命令，所以创建别名时不要犹豫。

在创建你认为应该存在的命令时这个技术会很有用。 例如，为了解决取消暂存文件的易用性问题，可以向 Git 中添加你自己的取消暂存别名：

$ git config --global alias.unstage 'reset HEAD --'

会使下面的两个命令等价：

$ git unstage fileA

$ git reset HEAD -- fileA

这样看起来更清楚一些。 通常也会添加一个 last 命令，像这样：

$ git config --global alias.last 'log -1 HEAD'

**Git分支简介**

几乎所有的版本控制系统都以某种形式支持分支。 使用分支意味着你可以把你的工作从开发主线上分离开来，以免影响开发主线。。 在很多版本控制系统中，这是一个略微低效的过程——常常需要完全创建一个源代码目录的副本。对于大项目来说，这样的过程会耗费很多时间。

或许你还记得 [起步](https://git-scm.com/book/zh/v2/ch00/ch01-introduction) 的内容，Git 保存的不是文件的变化或者差异，而是一系列不同时刻的文件快照。

在进行提交操作时，Git 会保存一个提交对象（commit object）。知道了 Git 保存数据的方式，我们可以很自然的想到——该提交对象会包含一个指向暂存内容快照的指针。 但不仅仅是这样，该提交对象还包含了作者的姓名和邮箱、提交时输入的信息以及指向它的父对象的指针。首次提交产生的提交对象没有父对象，普通提交操作产生的提交对象有一个父对象，而由多个分支合并产生的提交对象有多个父对象。

我们假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件。 暂存操作会为每一个文件计算校验和（使用我们在 [起步](https://git-scm.com/book/zh/v2/ch00/ch01-introduction) 中提到的 SHA-1 哈希算法），然后会把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 对象来保存它们），最终将校验和加入到暂存区域等待提交：

当使用 git commit 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和，然后在 Git 仓库中这些校验和保存为树对象。 随后，Git 便会创建一个提交对象，它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。如此一来，Git 就可以在需要的时候重现此次保存的快照。

现在，Git 仓库中有五个对象：三个 blob 对象（保存着文件快照）、一个树对象（记录着目录结构和 blob 对象索引）以及一个提交对象（包含着指向前述树对象的指针和所有提交信息）。



上图为首次提交对象及其树结构

做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针



Git 的分支，其实本质上仅仅是指向提交对象的可变指针。 Git 的默认分支名字是 master。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。 它会在每次的提交操作中自动向前移动。

以上的内容我并也不是特别理解，借鉴了廖雪峰的博客之后，我的理解是这样的：分支就是一条绳子，这条绳子的默认名字叫master，每次提交就相当于在绳子上打一个结，而进行每一次提交都会有一个指针指向最新的那个结，随着不断提交，绳子也就不断变长

**分支创建**

Git 是怎么创建新分支的呢？ 很简单，它只是为你创建了一个可以移动的新的指针。 比如，创建一个 testing 分支， 你需要使用 git branch 命令：

$ git branch testing

这会在当前所在的提交对象上创建一个指针。



那么，Git 又是怎么知道当前在哪一个分支上呢？ 也很简单，它有一个名为 HEAD 的特殊指针。 请注意它和许多其它版本控制系统（如 Subversion 或 CVS）里的 HEAD 概念完全不同。 在 Git 中，它是一个指针，指向当前所在的本地分支（译注：将 HEAD 想象为当前分支的别名）。 在本例中，你仍然在 master 分支上。 因为 git branch 命令仅仅 *创建* 一个新分支，并不会自动切换到新分支中去。Git创建分支很快，只是增加一个指向当前提交的指针，如果要切换分支，只需要将HEAD的指向从master分支移到新分支上就行了，工作区的文件没有什么变化。如下图，如果要切换到testing分支只要将HEAD的指针从master移到testing就OK了，当继续有新的提交出现，master指针也不再向前移动了，testing的指针开始指向新的提交，如果要合并分支，只要将master的指针指向testing的当前提交，就完成了合并。合并完分支，可以将testing分支删除，删除testing分支就是将testing的指针删除。



可以简单地使用 git log 命令查看各个分支当前所指的对象。 提供这一功能的参数是 --decorate。



**分支切换**

要切换到一个已存在的分支，你需要使用 git checkout 命令。 我们现在切换到新创建的 testing 分支去：

$ git checkout testing

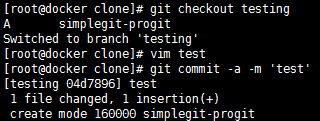


 想要新建一个分支并同时切换到那个分支上，你可以运行一个带有 -b 参数的 git checkout 命令

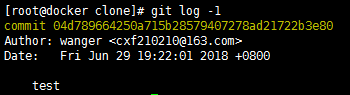
这样 HEAD 就指向 testing 分支了。



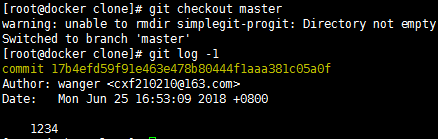
我们在testing分支编辑一些东西并提交



查看提交历史

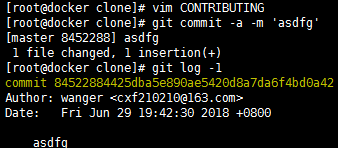


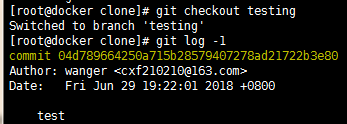
然后切换回master分支，并查看提交历史



可以看到切换回master分支以后，原来在testing分支上提交的内容已经不见了，也就是说切换回master分支后，指针还是停在之前未切换分支的提交上，

我们在master分支上再编辑一些东西，并切换回testing分支上，



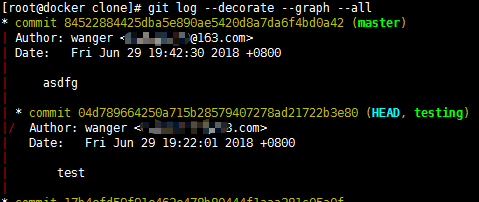


由上图可以看到，在master分支上做的提交也并没有出现在testing分支上

现在，这个项目的提交历史已经产生了分叉。因为刚才你创建了一个新分支，并切换过去进行了一些工作，随后又切换回 master 分支进行了另外一些工作。 上述两次改动针对的是不同分支：你可以在不同分支间不断地来回切换和工作，并在时机成熟时将它们合并起来。



你可以简单地使用 git log 命令查看分叉历史。 运行 git log --oneline --decorate --graph --all ，它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况。

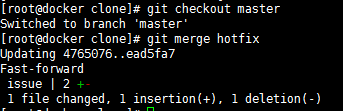


**分支的合并**

你可以运行你的测试，确保你的修改是正确的，然后将其合并回你的 master 分支来部署到线上。 你可以使用 git merge 命令来达到上述目的：

$ git checkout master

$ git merge hotfix



在合并的时候，你应该注意到了"快进（fast-forward）"这个词。 由于当前 master 分支所指向的提交是你当前提交（有关 hotfix 的提交）的直接上游，所以 Git 只是简单的将指针向前移动。 换句话说，当你试图合并两个分支时，如果顺着一个分支走下去能够到达另一个分支，那么 Git 在合并两者的时候，只会简单的将指针向前推进（指针右移），因为这种情况下的合并操作没有需要解决的分歧——这就叫做 “快进

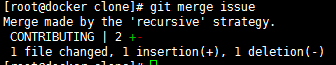
下图是master分支的指针向右快进到了hotfix的当前提交的指针上：



当合并到hotfix分支后，hotfix分支就没有利用价值了，就可以将它删除了， 你可以使用带 -d 选项的 git branch 命令来删除分支：



如果现在issue分支上的问题已经解决了，要将master分支与issue分支合并，运行git merge命令：



这和我之前合并 hotfix 分支的时候看起来有一点不一样。因为，master 分支所在提交并不是 iss53 分支所在提交的直接祖先，Git 不得不做一些额外的工作。 出现这种情况的时候，Git 会使用两个分支的末端所指的快照（C4 和 C5）以及这两个分支的工作祖先（C2），做一个简单的三方合并。



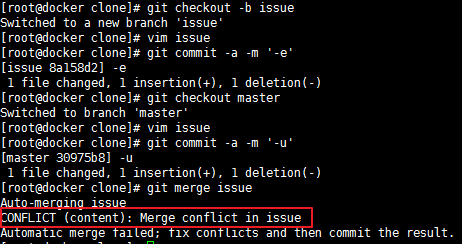
和之前将分支指针向前推进所不同的是，Git 将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。 这个被称作一次合并提交，它的特别之处在于他有不止一个父提交（也就是C4和C5）。



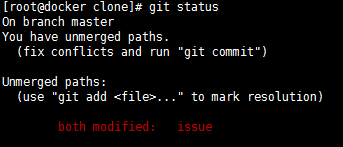
**遇到冲突时的分支合并**

有时候合并操作不会如此顺利。 如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们。 如果你对issue分支的修改和master分支 的修改都涉及到同一个文件的同一处，在合并它们的时候就会产生合并冲突：

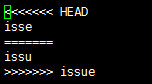
这里我分别在master分支和issue分支上都修改了issue文件，并进行了提交，可以看到我合并issue分支的时候明确提示了issue文件产生了冲突



此时 Git 做了合并，但是没有自动地创建一个新的合并提交。 Git 会暂停下来，等待你去解决合并产生的冲突。 你可以在合并冲突后的任意时刻使用 git status 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：



任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。 Git 会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。 出现冲突的文件会包含一些特殊区段，看起来像下面这个样子：



这表示 HEAD 所指示的版本（也就是你的 master 分支所在的位置，因为你在运行 merge 命令的时候已经检出到了这个分支）在这个区段的上半部分（======= 的上半部分），而 iss53 分支所指示的版本在 ======= 的下半部分。 为了解决冲突，你必须选择使用由 ======= 分割的两部分中的一个，或者你也可以自行合并这些内容。 例如，你可以通过把这段内容换成下面的样子来解决冲突：



上述的冲突解决方案仅保留了其中一个分支的修改，并且 <<<<<<< , ======= , 和 >>>>>>> 这些行被完全删除了。 在你解决了所有文件里的冲突之后，对冲突的文件进行提交之后再合并就可以了。



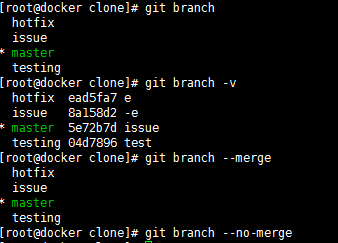
**分支管理**

git branch 命令不只是可以创建与删除分支。 如果不加任何参数运行它，会得到当前所有分支的一个列表：

注意 master 分支前的 \* 字符：它代表现在检出的那一个分支（也就是说，当前 HEAD 指针所指向的分支）。 这意味着如果在这时候提交，master 分支将会随着新的工作向前移动。 如果需要查看每一个分支的最后一次提交，可以运行 git branch -v 命令：

--merged 与 --no-merged 这两个有用的选项可以过滤这个列表中已经合并或尚未合并到当前分支的分支。 如果要查看哪些分支已经合并到当前分支，可以运行 git branch --merged：

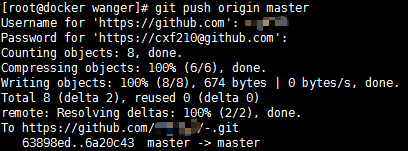
已经合并了的分支通常可以使用 git branch -d 删除掉；你已经将它们的工作整合到了另一个分支，所以并不会失去任何东西。



**推送**

当你想要公开分享一个分支时，需要将其推送到有写入权限的远程仓库上。 本地的分支并不会自动与远程仓库同步 - 你必须显式地推送想要分享的分支。 这样，你就可以把不愿意分享的内容放到私人分支上，而将需要和别人协作的内容推送到公开分支。

如果希望和别人一起在名为master 的分支上工作，你可以像推送第一个分支那样推送它。 运行 git push (remote) (branch):



你也可以运行 git push origin master:master，它会做同样的事 - 相当于它说，“推送本地的master分支，将其作为远程仓库的master分支” 可以通过这种格式来推送本地分支到一个命名不相同的远程分支。如果并不想让远程仓库上的分支叫做 master，可以运行 git push origin master:wanger 来将本地的 master分支推送到远程仓库的wanger分支。

**如果不想在每一次推送时都输入用户名与密码，你可以设置一个 “credential cache”。 最简单的方式就是将其保存在内存中几分钟，可以简单地运行 git config --global credential.helper cache 来设置它。**

下一次其他协作者从服务器上抓取数据时，他们会在本地生成一个远程分支 origin/master，指向服务器的 master分支的引用：

要特别注意的一点是当抓取到新的远程跟踪分支时，本地不会自动生成一份可编辑的副本（拷贝）。 换一句话说，这种情况下，不会有一个新的 master 分支 - 只有一个不可以修改的 origin/master 指针。

可以运行 git merge origin/master将这些工作合并到当前所在的分支。 如果想要在自己的 master 分支上工作，可以将其建立在远程跟踪分支之上：



这会给你一个用于工作的本地分支，并且起点位于 origin/master。

**跟踪分支**

从一个远程跟踪分支检出一个本地分支会自动创建一个叫做 “跟踪分支”（有时候也叫做 “上游分支”）。 跟踪分支是与远程分支有直接关系的本地分支。 如果在一个跟踪分支上输入 git pull，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

当克隆一个仓库时，它通常会自动地创建一个跟踪 origin/master 的 master 分支。 然而，如果你愿意的话可以设置其他的跟踪分支 - 其他远程仓库上的跟踪分支，或者不跟踪 master 分支。 最简单的就是之前看到的例子，运行 git checkout -b [branch] [remotename]/[branch]。 这是一个十分常用的操作所以 Git 提供了 --track 快捷方式：

如果想要将本地分支与远程分支设置为不同名字，你可以轻松地增加一个不同名字的本地分支的上一个命令：



现在，本地分支 sf 会自动从 origin/serverfix 拉取。

设置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支，你可以在任意时间使用 -u 或 --set-upstream-to 选项运行 git branch 来显式地设置。



如果想要查看设置的所有跟踪分支，可以使用 git branch 的 -vv 选项。 这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。



这里可以看到master分支正在跟踪origin/master分支

**拉取**

当 git fetch 命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。 它只会获取数据然后让你自己合并。 然而，有一个命令叫作 git pull 在大多数情况下它的含义是一个 git fetch 紧接着一个 git merge 命令。 如果有一个像之前章节中演示的设置好的跟踪分支，不管它是显式地设置还是通过 clone 或 checkout 命令为你创建的，git pull 都会查找当前分支所跟踪的服务器与分支，从服务器上抓取数据然后尝试合并入那个远程分支。

由于 git pull 的魔法经常令人困惑所以通常单独显式地使用 fetch 与 merge 命令会更好一些。

**删除远程分支**

假设你已经通过远程分支做完所有的工作了 - 也就是说你和你的协作者已经完成了一个特性并且将其合并到了远程仓库的 master 分支（或任何其他稳定代码分支）。 可以运行带有 --delete 选项的 git push 命令来删除一个远程分支。 如果想要从服务器上删除 server 分支，运行下面的命令：



当然也可以使用git push origin –delete server

基本上这个命令做的只是从服务器上移除这个指针。 Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。

**变基**

在 Git 中整合来自不同分支的修改主要有两种方法：merge 以及 rebase。 在

**变基的基本操作**

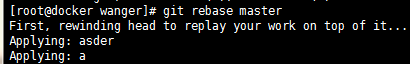
请回顾之前在 [分支的合并](https://git-scm.com/book/zh/v2/ch00/r_basic_merging) 中的一个例子，你会看到开发任务分叉到两个不同分支，又各自提交了更新。



之前介绍过，整合分支最容易的方法是 merge 命令。 它会把两个分支的最新快照（C3 和 C4）以及二者最近的共同祖先（C2）进行三方合并，合并的结果是生成一个新的快照（并提交）。



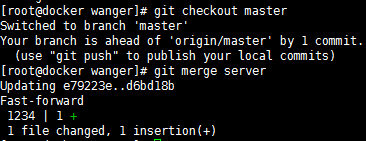
其实，还有一种方法：你可以提取在 C4 中引入的补丁和修改，然后在 C3 的基础上应用一次。 在 Git 中，这种操作就叫做 *变基*。 你可以使用 rebase 命令将提交到某一分支上的所有修改都移至另一分支上，就好像“重新播放”一样。



它的原理是首先找到这两个分支（即当前分支 experiment、变基操作的目标基底分支 master）的最近共同祖先 C2，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件，然后将当前分支指向目标基底 C3, 最后以此将之前另存为临时文件的修改依序应用。



现在回到 master 分支，进行一次快进合并。





现在两个指针指向了同一次的提交

此时，C4' 指向的快照就和上面使用 merge 命令的例子中 C5 指向的快照一模一样了。 这两种整合方法的最终结果没有任何区别，但是变基使得提交历史更加整洁。 你在查看一个经过变基的分支的历史记录时会发现，尽管实际的开发工作是并行的，但它们看上去就像是串行的一样，提交历史是一条直线没有分叉。

无论是通过变基，还是通过三方合并，整合的最终结果所指向的快照始终是一样的，只不过提交历史不同罢了。 变基是将一系列提交按照原有次序依次应用到另一分支上，而合并是把最终结果合在一起。

关于变基的更多操作可以输入命令man git rebase获取

**变基的风险**

呃，奇妙的变基也并非完美无缺，要用它得遵守一条准则：

**不要对在你的仓库外有副本的分支执行变基。**

如果你遵循这条金科玉律，就不会出差错。 否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你。

变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。

4.1 服务器上的 Git - 协议

架设一台 Git 服务器并不难。 首先，选择你希望服务器使用的通讯协议。 在本章第一节将介绍可用的协议以及各自优缺点。 下面一节将解释使用那些协议的典型设置及如何在你的服务器上运行。 最后，如果你不介意托管你的代码在其他人的服务器，且不想经历设置与维护自己服务器的麻烦，可以试试我们介绍的几个仓库托管服务。

**协议**

Git 可以使用四种主要的协议来传输资料：本地协议（Local），HTTP 协议，SSH（Secure Shell）协议及 Git 协议。

**本地协议**

最基本的就是 *本地协议（Local protocol）* ，其中的远程版本库就是硬盘内的另一个目录。 这常见于团队每一个成员都对一个共享的文件系统（例如一个挂载的 NFS）拥有访问权，或者比较少见的多人共用同一台电脑的情况。

如果你使用共享文件系统，就可以从本地版本库克隆（clone）、推送（push）以及拉取（pull）。 像这样去克隆一个版本库或者增加一个远程到现有的项目中，使用版本库路径作为 URL。 例如，克隆一个本地版本库，可以执行如下的命令：

$ git clone /opt/git/project.git

或你可以执行这个命令：

$ git clone file:///opt/git/project.git

如果在 URL 开头明确的指定 file://，那么 Git 的行为会略有不同。 如果仅是指定路径，Git 会尝试使用硬链接（hard link）或直接复制所需要的文件。 如果指定 file://，Git 会触发平时用于网路传输资料的进程，那通常是传输效率较低的方法。

要增加一个本地版本库到现有的 Git 项目，可以执行如下的命令：

$ git remote add local\_proj /opt/git/project.git

然后，就可以像在网络上一样从远端版本库推送和拉取更新了。

**优点**

基于文件系统的版本库的优点是简单，并且直接使用了现有的文件权限和网络访问权限。 如果你的团队已经有共享文件系统，建立版本库会十分容易。 只需要像设置其他共享目录一样，把一个裸版本库的副本放到大家都可以访问的路径，并设置好读/写的权限，就可以了。

**HTTP 协议**

Git 通过 HTTP 通信有两种模式。 在 Git 1.6.6 版本之前只有一个方式可用，十分简单并且通常是只读模式的。 Git 1.6.6 版本引入了一种新的、更智能的协议，让 Git 可以像通过 SSH 那样智能的协商和传输数据。 之后几年，这个新的 HTTP 协议因为其简单、智能变的十分流行。

**智能（Smart） HTTP 协议**

“智能” HTTP 协议的运行方式和 SSH 及 Git 协议类似，只是运行在标准的 HTTP/S 端口上并且可以使用各种 HTTP 验证机制，这意味着使用起来会比 SSH 协议简单的多，比如可以使用 HTTP 协议的用户名／密码的基础授权，免去设置 SSH 公钥。

智能 HTTP 协议或许已经是最流行的使用 Git 的方式了，它即支持像 git:// 协议一样设置匿名服务，也可以像 SSH 协议一样提供传输时的授权和加密。 而且只用一个 URL 就可以都做到，省去了为不同的需求设置不同的 URL。 如果你要推送到一个需要授权的服务器上（一般来讲都需要），服务器会提示你输入用户名和密码。 从服务器获取数据时也一样。

**优点**

不同的访问方式只需要一个 URL 以及服务器只在需要授权时提示输入授权信息，这两个简便性让终端用户使用 Git 变得非常简单。 相比 SSH 协议，可以使用用户名／密码授权是一个很大的优势，这样用户就不必须在使用 Git 之前先在本地生成 SSH 密钥对再把公钥上传到服务器。 对非资深的使用者，或者系统上缺少 SSH 相关程序的使用者，HTTP 协议的可用性是主要的优势。 与 SSH 协议类似，HTTP 协议也非常快和高效。

你也可以在 HTTPS 协议上提供只读版本库的服务，如此你在传输数据的时候就可以加密数据；或者，你甚至可以让客户端使用指定的 SSL 证书。

另一个好处是 HTTP/S 协议被广泛使用，一般的企业防火墙都会允许这些端口的数据通过。

**缺点**

在一些服务器上，架设 HTTP/S 协议的服务端会比 SSH 协议的棘手一些。 除了这一点，用其他协议提供 Git 服务与 “智能” HTTP 协议相比就几乎没有优势了。

如果你在 HTTP 上使用需授权的推送，管理凭证会比使用 SSH 密钥认证麻烦一些。

**SSH 协议**

架设 Git 服务器时常用 SSH 协议作为传输协议。 因为大多数环境下已经支持通过 SSH 访问 —— 即时没有也比较很容易架设。 SSH 协议也是一个验证授权的网络协议；并且，因为其普遍性，架设和使用都很容易。

通过 SSH 协议克隆版本库，你可以指定一个 ssh:// 的 URL：

$ git clone ssh://user@server/project.git

或者使用一个简短的 scp 式的写法：

$ git clone user@server:project.git

你也可以不指定用户，Git 会使用当前登录的用户名。

**优势**

用 SSH 协议的优势有很多。 首先，SSH 架设相对简单 —— SSH 守护进程很常见，多数管理员都有使用经验，并且多数操作系统都包含了它及相关的管理工具。 其次，通过 SSH 访问是安全的 —— 所有传输数据都要经过授权和加密。 最后，与 HTTP/S 协议、Git 协议及本地协议一样，SSH 协议很高效，在传输前也会尽量压缩数据。

**缺点**

SSH 协议的缺点在于你不能通过他实现匿名访问。 即便只要读取数据，使用者也要有通过 SSH 访问你的主机的权限，这使得 SSH 协议不利于开源的项目。 如果你只在公司网络使用，SSH 协议可能是你唯一要用到的协议。 如果你要同时提供匿名只读访问和 SSH 协议，那么你除了为自己推送架设 SSH 服务以外，还得架设一个可以让其他人访问的服务。

**Git 协议**

接下来是 Git 协议。 这是包含在 Git 里的一个特殊的守护进程；它监听在一个特定的端口（9418），类似于 SSH 服务，但是访问无需任何授权。 要让版本库支持 Git 协议，需要先创建一个 git-daemon-export-ok 文件 —— 它是 Git 协议守护进程为这个版本库提供服务的必要条件 —— 但是除此之外没有任何安全措施。 要么谁都可以克隆这个版本库，要么谁也不能。 这意味着，通常不能通过 Git 协议推送。 由于没有授权机制，一旦你开放推送操作，意味着网络上知道这个项目 URL 的人都可以向项目推送数据。 不用说，极少会有人这么做。

**优点**

目前，Git 协议是 Git 使用的网络传输协议里最快的。 如果你的项目有很大的访问量，或者你的项目很庞大并且不需要为写进行用户授权，架设 Git 守护进程来提供服务是不错的选择。 它使用与 SSH 相同的数据传输机制，但是省去了加密和授权的开销。

**缺点**

Git 协议缺点是缺乏授权机制。 把 Git 协议作为访问项目版本库的唯一手段是不可取的。 一般的做法里，会同时提供 SSH 或者 HTTPS 协议的访问服务，只让少数几个开发者有推送（写）权限，其他人通过 git:// 访问只有读权限。 Git 协议也许也是最难架设的。 它要求有自己的守护进程，这就要配置 xinetd或者其他的程序，这些工作并不简单。 它还要求防火墙开放 9418 端口，但是企业防火墙一般不会开放这个非标准端口。 而大型的企业防火墙通常会封锁这个端口。

**在服务器上搭建 Git**(需要提前安装Git)

1.首先创建一个git用户和git组

[root@docker ~]# groupadd git

[root@docker ~]# useradd -g git git

禁止 git 用户 ssh 登录服务器，修改/etc/passwd文件 ，git-shell 是git的受限 shell 工具，你可以方便地将用户 git 的活动限制在与 Git 相关的范围内。该工具随 Git 软件包一同提供。

将git:x:1001:1002::/home/git:/bin/bash修改为

git:x:1001:1002::/home/git:/usr/bin/git-shell

这样，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。

2.创建Git版本仓库

[root@docker ~]# cd /home/git/

[root@docker git]# mkdir study.git

[root@docker git]# git init --bare study.git

这一步是将study.git初始化成git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，裸仓库没有工作区，服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区。

[root@docker git]# chown -R git:git study.git

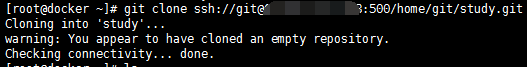
把仓库所属用户改为git

3.克隆远程仓库

git clone [git@173.168.16.222:/home/git/study.git](mailto:git@173.168.16.222:/home/git/study.git)

如果SSH用的不是默认的22端口，则需要使用以下的命令

git clone ssh://git@173.168.16.222:500/home/git/study.git



**注意：这里我在克隆时出现了一点错误，后面的路径写错了，git用户没有权限访问远程仓库都会报错，出现下面的报错信息建议检查以上两点是否有错误**



4.创建并配置密钥登陆

创建密钥是为了在进行git clone操作的时候免密登入，

首先编辑ssh的配置文件/etc/ssh/sshd\_config添加如下三行内容，修改完成之后重启ssh服务：

RSAAuthentication yes

PubkeyAuthentication yes

AuthorizedKeysFile .ssh/authorized\_keys

管理 Git 服务的用户是 git，所以实际存放公钥的路径是 /home/git/.ssh/authorized\_keys

在git服务器上创建存放公钥的目录

[root@docker git]# mkdir .ssh

[root@docker git]# chmod 700 .ssh

[root@docker git]# touch .ssh/authorized\_keys

[root@docker git]#chmod 600 .ssh/authorized\_keys

最后在客户端生成密钥对并将公钥传到服务端的/home/git/.ssh/authorized\_keys文件里，使用ssh-keygen生成密钥对，可以使用scp命令将生成的公钥传到服务端

这样Git服务器就搭建完成了

**在centos7上搭建GitLab**

1.安装所需依赖环境

[root@localhost ~]# yum -y install policycoreutils openssh-server openssh-clients postfix

2.启动postfix并让他开机自启

[root@localhost ~]# systemctl enable postfix && systemctl start postfix

3.下载并安装gitlab

[root@localhost ~]# wget <https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el7/gitlab-ce-8.9.5-ce.0.el7.x86_64.rpm>

[root@localhost ~]# rpm -i gitlab-ce-8.9.5-ce.0.el7.x86\_64.rpm

4.修改配置文件，定义服务器ip

Gitlab的配置文件位于/etc/gitlab/gitlab.rb，因为gitlab是由Ruby语言编写的，所以配置文件后缀是rb，将配置文件里的external\_url项修改为自己的ip，比如：external\_url 'http://192.168.188.222’

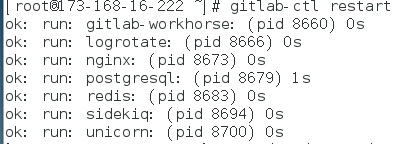
5.重置gitlab

[root@localhost ~]# gitlab-ctl reconfigure

运行此命令其实就是运行了一个自动化配置的脚本，它会自动安装gitlab需要的组件

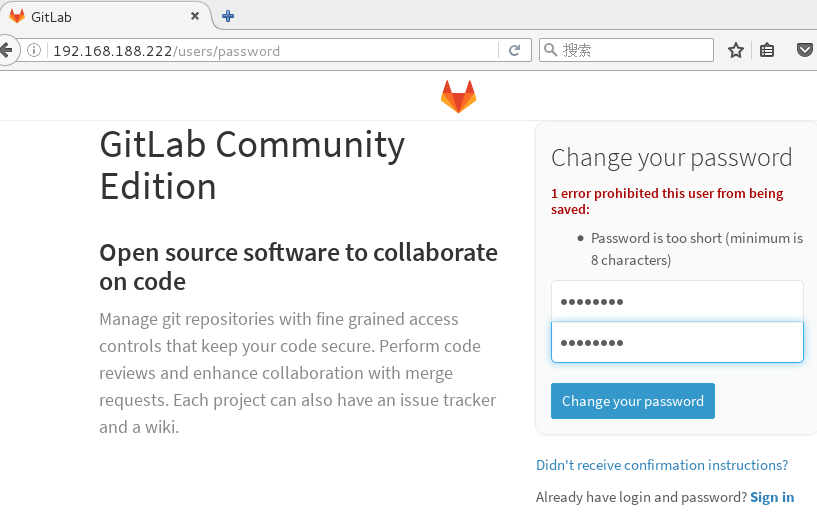
[root@localhost ~]# gitlab-ctl restart

重启之后可以看到相关的组件已经安装完成并成功运行

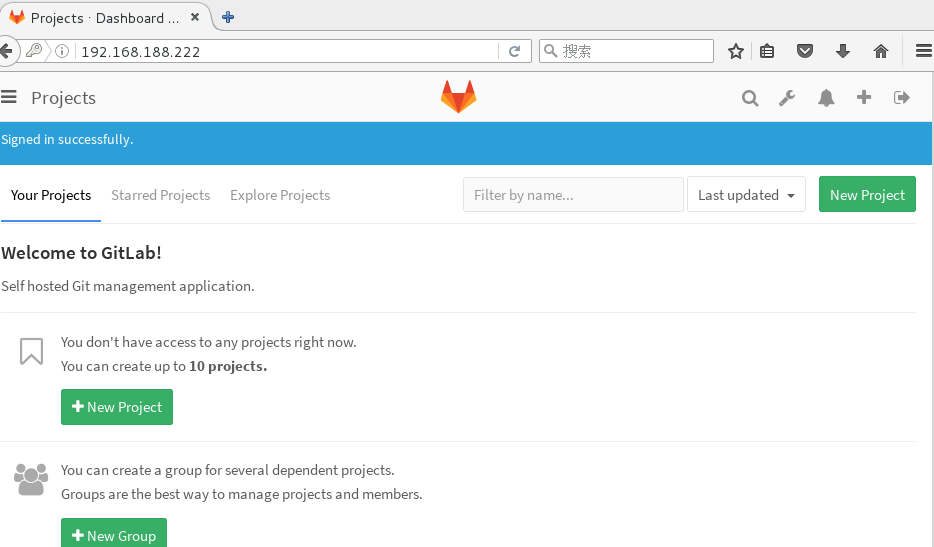


6.进入gitlab

输入ip地址即可进入gitlab页面，他会提示你去设置一个用于登录的密码

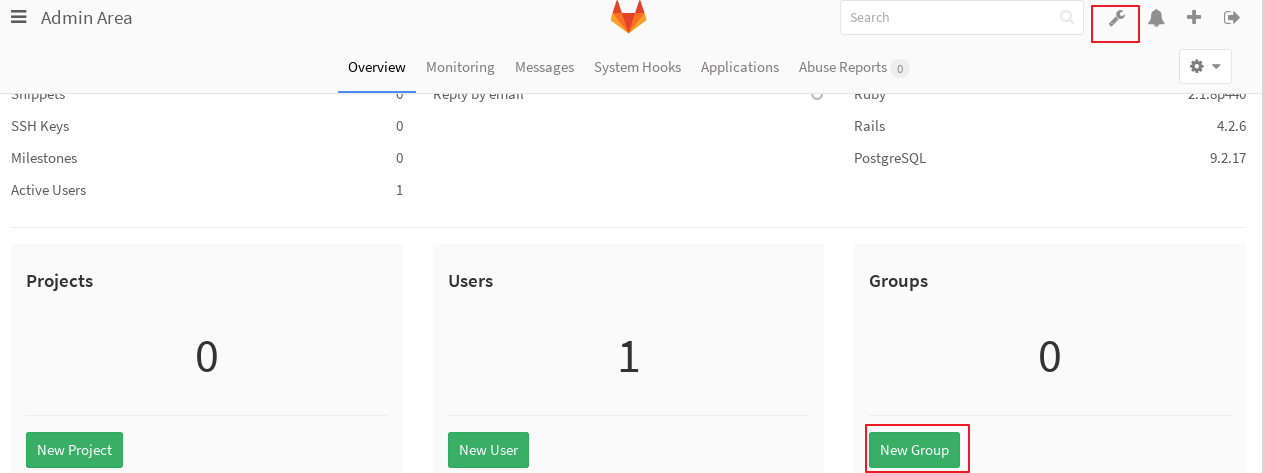


这样Gitlab的搭建就完成了

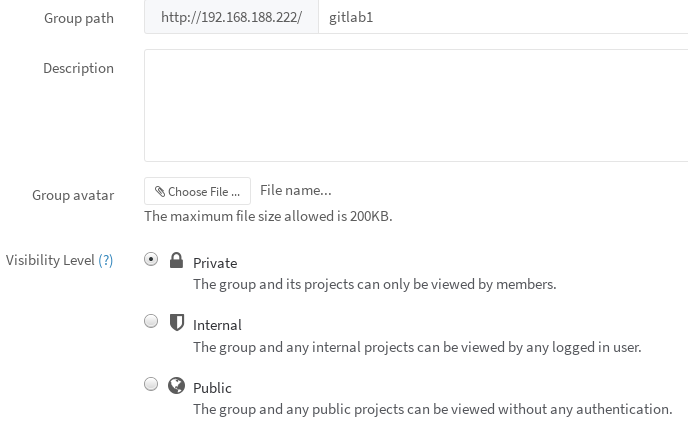


**Gitlab的使用**

单击屏幕右上角的小扳手可以进入管理区域，单击‘New Group’可以创建一个组，公司里的一个项目可能会有好几个开发团队来进行开发，可以为每一个团队创建一个项目组



在‘Group path’栏里可以输入组的路径，在‘Visibility Level’栏里可以设置安全级别，第一级private表示只有授权过的人才能访问项目，第二级internal表示只有能登陆gitlab的人才能够访问项目，第三级public表示所有能访问到gitlab上的人才能访问项目，点击左下角‘Create group’即可创建组



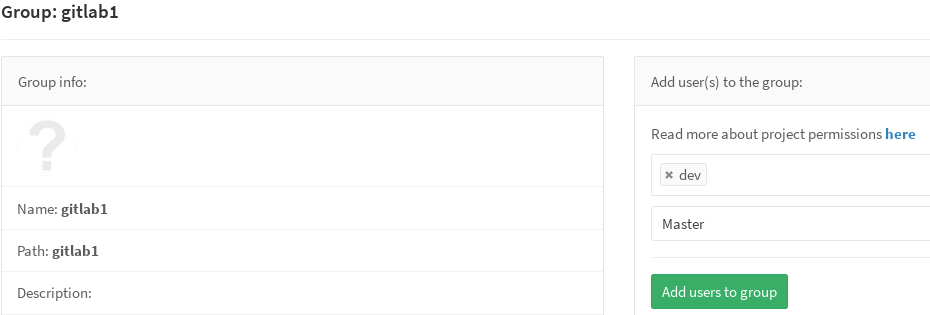
单击‘Add users to group’可以将用户添加进组里

再次单击右上角的小扳手就又回到了管理区域，单击‘New User’可以添加用户，在‘Account’处可以设置用户信息，在‘Access’处可以设置用户的权限，比如‘Project limit’可以设置该用户允许创建私人库的最大数目，‘Can create group’表示该用户是否可以创建组，‘Admin’表示是否将其设置为管理员，单击左下角‘Create user’即可创建用户

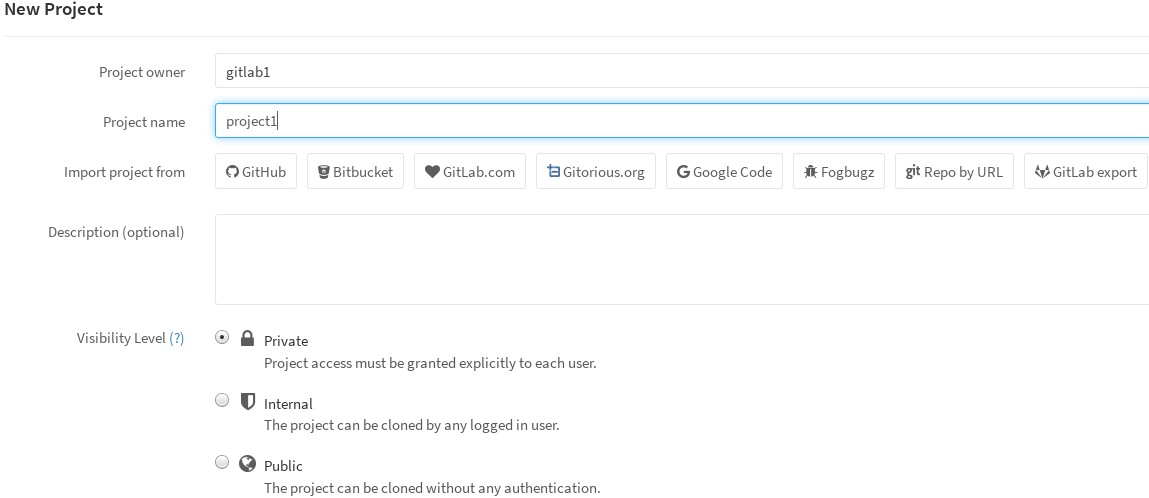


我在这里创建了两个用户，一个叫dev,另一个叫ops，现在我要把这两个用户都加到刚才新建的组里，单击‘Groups’下的数字1就可以进入新建的组里，在右边表单的第一列可以选择要添加的用户，第二列选择用户的权限，单击‘Add users to group’即可添加用户到组，下面完整列出了Guest,Reporter,Developer,Master,Owner对应的权限。

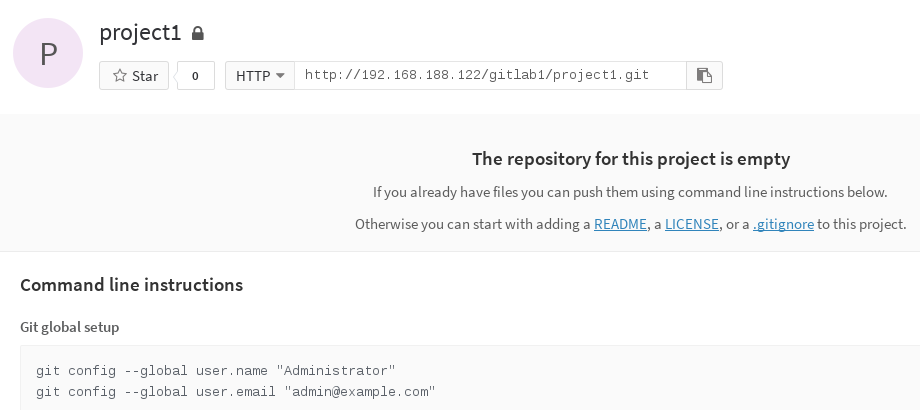
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Guest | Reporter | Developer | Master | Owner |
| Create new issues | \* | \* | \* | \* | \* |
| Leave comments | \* | \* | \* | \* | \* |
| Pull the project code |  | \* | \* | \* | \* |
| Download a project |  | \* | \* | \* | \* |
| Create code snippets |  | \* | \* | \* | \* |
| Create new merge requests |  |  | \* | \* | \* |
| Push changes to nonprotected branches |  |  | \* | \* | \* |
| Remove nonprotected branches |  |  | \* | \* | \* |
| Add tags |  |  | \* | \* | \* |
| Write a wiki |  |  | \* | \* | \* |
| Manage the issue tracker |  |  | \* | \* | \* |
| Add new team members |  |  |  | \* | \* |
| Push changes to protected branches |  |  |  | \* | \* |
| Manage the branch protection |  |  |  | \* | \* |
| Manage Git tags |  |  |  | \* | \* |
| Edit the project |  |  |  | \* | \* |
| Add deploy keys to the project |  |  |  | \* | \* |
| Configure the project hooks |  |  |  | \* | \* |



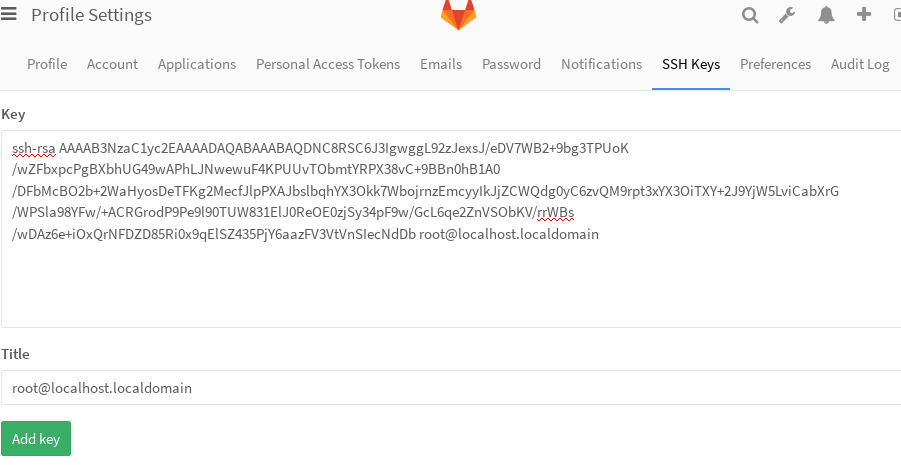
接下来开始创建一个新项目，再次点击小扳手图标进入到管理区域，单击‘New Project’创建一个新项目，单击‘Project owner’可以选择一个项目的属主，在‘Project name’一栏设置项目名称，‘Visibility Level’会自动继承属主的权限，单击左下角的‘Create project’可以创建新项目



可以看到新项目已经创建成功了

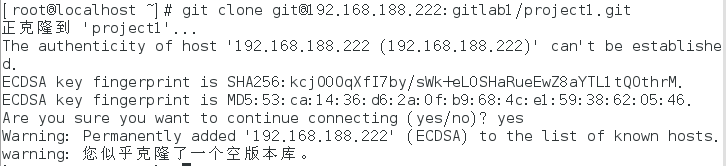


对gitlab进行权限的配置，单击左侧菜单栏打开侧边栏，找到‘Profile Settings’单击‘SSH Keys’为开发者添加ssh-key，添加ssh-key可以使开发者下的所有项目都赋予可以让开发者上传的权限，如何生成密钥对我在搭建Git服务器那一部分中提到过，进入/root/.ssh/目录下查看公钥的内容，并将内容复制到gitlab上，单击‘Add key’即可成功添加ssh-key

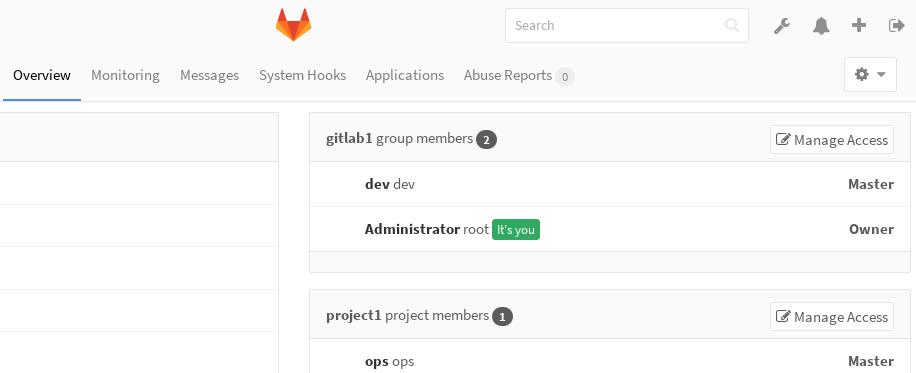


添加完ssh-key我们进行clone测试

[root@localhost ~]# git clone git@192.168.188.222:gitlab1/project1.git

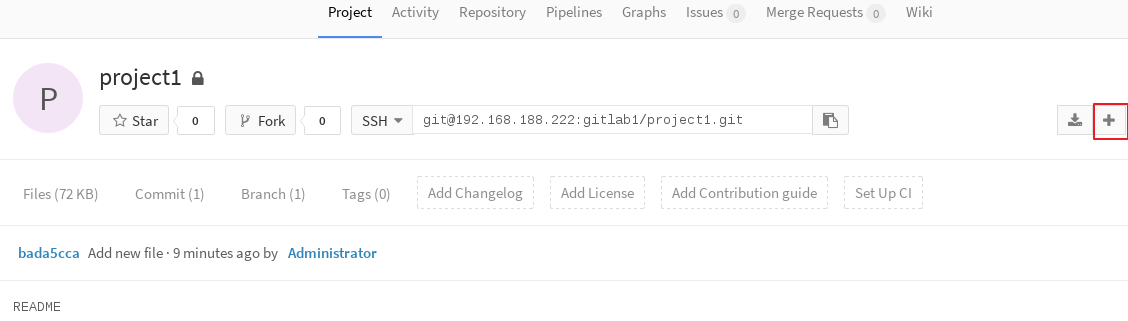


打开新建的项目可以看到项目的概览，项目会自动继承所属组里的用户权限，当然也可以单独在项目那里添加用户，在project members处单击‘Manage Access’可以为项目单独添加用户并授权，可以看到这个项目的组里有了两个用户，一个是超级管理员，一个是我在组里添加的用户dev,而在项目用户那里我新添加了一个ops用户



新建一次提交

在所在的项目页面单击右上角的加号按钮，有一个‘New file’选项，单击它会创建一个新文件，按照表单输入文件名，文件内容，提交描述信息，单击‘Commit Changes’就可以新建一个提交了，初次提交会创建一个master的主分支，可以看到我这里提交了一个README的文件

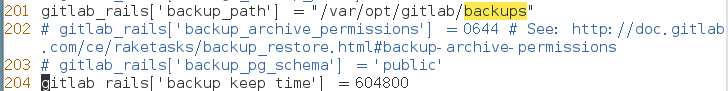


在客户端使用git pull测试，可以看到目录下面多了一个README文件

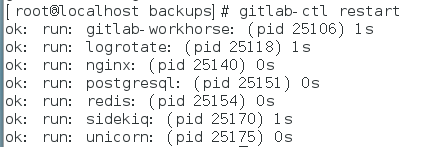


**Gitlab备份**

首先编辑配置文件/etc/gitlab/gitlab.rb，找到配置文件里的备份目录，存在于201行，这行默认是注释的，需要取消注释，备份文件路径默认存在于/var/opt/gitlab/backups目录，当然也可以在配置文件中自己定义备份文件的路径，除此之外还要设置备份文件存在时间，默认是7天，也就是604800秒，关于备份文件存在时间的配置在配置文件的204行，也是需要取消注释的，修改完之后保存退出

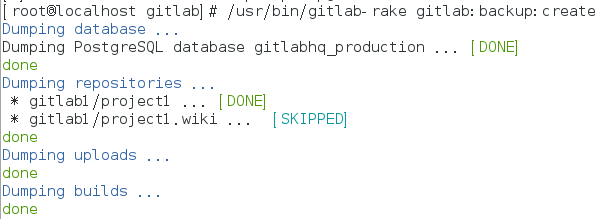


运行gitlab-ctl reconfigure重置gitlab的配置，不然修改的配置不会生效，重置完之后重启



运行下面的命令即可将gitlab进行备份

[root@localhost gitlab]# /usr/bin/gitlab-rake gitlab:backup:create，如果想让gitlab定时备份，可以将命令放到定时任务里去



我们可以进入备份目录里查看备份的结果

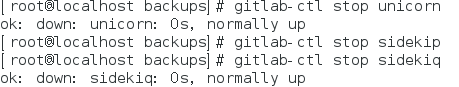


**Gitlab恢复**

在恢复之前需要停止数据写入，运行下面的两个命令

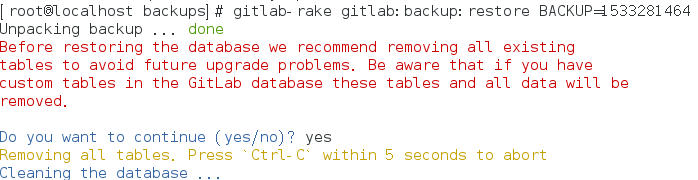
[root@localhost backups]# gitlab-ctl stop unicorn

[root@localhost backups]# gitlab-ctl stop sidekiq



恢复命令如下

[root@localhost backups]# gitlab-rake gitlab:backup:restore BACKUP=1533281464 #BACKUP后面加的是备份文件前面的时间戳



恢复完成后重启之前停掉的那两个服务

[root@localhost backups]# gitlab-ctl start unicorn

[root@localhost backups]# gitlab-ctl start sidekiq

也可以重启所有服务

[root@localhost backups]# gitlab-ctl restart

这里我之前把项目删除了，进行恢复测试了一下，项目成功恢复