## git的使用

### 1 git简介与安装

1. Git简介

什么是版本控制

版本控制系统（Version Control System，简称VCS）是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统

按类型可以分为：

1：集中化的版本控制系统

2：分布式版本控制系统

Git是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。

Git是Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

Git与常用的版本控制工具CVS、Subversion等不同，它采用了分布式版本库的方式，不必依靠服务器端软件支持。

2、Git与SVN的区别

GIT不仅仅是个版本控制系统，它也是个内容管理系统(CMS),工作管理系统等。

SVN是集中式的版本控制系统，而Git是分布式版本控制系统。

集中式版本控制系统，版本库是集中存放在中央服务器的。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。

集中式版本控制系统的工作流程如下：

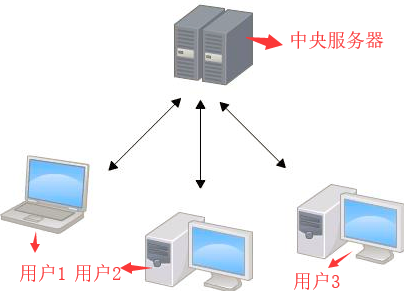


图8-1集中式版本控制系统工作流程

与集中式版本控制系统相比，分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，当你在工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。不同电脑之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。具体工作流程如下：

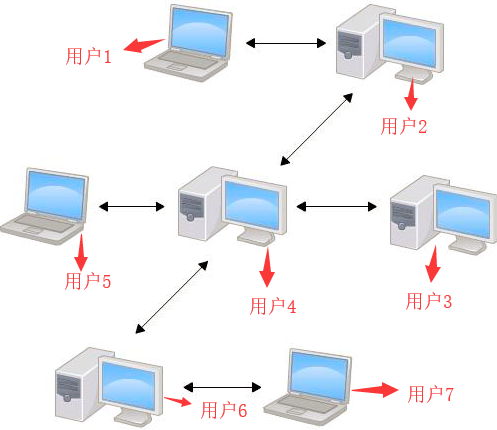


图8-2分布式版本控制系统工作流程

1. git的安装
2. Linux操作系统上安装

如果你使用的系统是 Centos/RedHat 安装命令为：

$ yum install curl-devel expat-devel gettext-devel \

openssl-devel zlib-devel

$ yum -y install git-core

$ git --version

git --version 1.8.3.1

1. Windows操作系统上安装

下载地址：https://git-scm.com/downloads

下载完成后，按照默认选项安装即可。

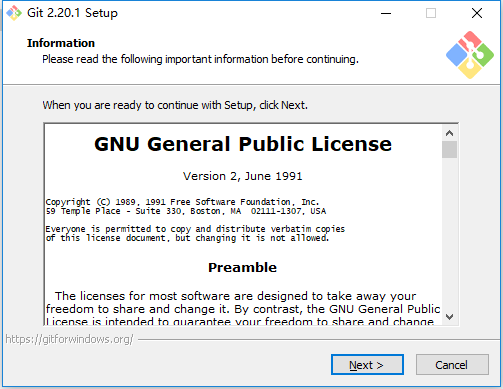


图8-3 Git安装界面

1. git配置

Git 提供了一个叫做 git config的工具，专门用来配置或读取相应的工作环境变量。

1. 用户信息

因为Git是分布式版本控制系统，所以，每个机器都必须设置用户名和Email地址信息。

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

1. 查看配置信息

要检查已有的配置信息，可以使用 git config --list 命令。

### 2 git工作区、暂存区和版本库

工作区（Working Directory）：就是你本地电脑的目录。

暂存区：英文叫stage, 或index。

一般存放在 ".git”目录下的index文件（.git/index）中，所以我们把暂存区有时也叫作索引。

版本库：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。你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

下面这个图展示了工作区、版本库中的暂存区和版本库之间的关系：



图8-4工作区、版本库中的暂存区和版本库关系图

图中左侧为工作区，右侧为版本库。在版本库中标记为 "index" 的区域是暂存区（stage, index），标记为 "master" 的是 master 分支所代表的目录树。

### 3 git基本操作

1、创建版本库

创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录，然后使用git init命令把这个目录变成Git可以管理的仓库。

$ cd D:

$ mkdir gitproject

$ cd gitproject

$ pwd

/d/gitproject

$ git init

Initialized empty Git repository in D:/gitproject/.git/

mkdir命令用于创建一个文件夹

cd 命令用于切换到指定的目录中

pwd命令用于显示当前目录

通过执行init命令，瞬间Git就把仓库建好了，而且告诉你是一个空的仓库（empty Git repository），这时在当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的。

2、添加文件到版本库

(1)创建文件

版本库创建好之后，在该文件夹下创建一个hello.txt文件，并且在该文件中写入一些内容。

$ touch hello.txt

$ echo 'hello git' > hello.txt

touch命令用于创建一个空文件

echo 'hello git' > hello.txt命令用于将指定的内容写入到文件中。

2.检查状态

文件创建好后，可以通过git status命令查看状态。

git status是另一个非常重要的命令，它反馈给我们仓库当前状态的信息：是否为最新代码，有什么更新等等。

$ git status

On branch master

No commits yet

Untracked files:

(use "git add <file>..." to include in what will be committed)

hello.txt

nothing added to commit but untracked files present (use "git add" to track)

反馈信息告诉我们，hello.txt尚未跟踪，这是说这个文件是新的，git不知道是应该跟踪它的变动还是直接忽略。为了跟踪我们的新文件，我们需要暂存它。

3.暂存文件

Git有个概念叫“暂存区“，你可以把它看成一块空白的画布，包裹着所有你可能会提交的变动。它一开始是空的，可以通过 git add 命令添加内容，最后使用 git commit 提交（创建一个快照）。

$ git add hello.txt

如果需要提交目录下的所有内容，可以这样做：

$ git add -A

如果只需要提交修改的内容，可以这样做：

$ git add .

再次使用git status查看状态

$ git status

On branch master

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: hello.txt

文件已经准备好可以提交了。状态信息还告诉我们暂存区文件发生了什么变动，这里我们新增了一个文件，同样可以做修改和删除。取决于我们在上一次git add之后发生了什么。

4.提交文件

一次提交代表着我们的仓库到了一个新的状态，就像是一个快照，允许我们像使用时光机一样回到之前的某个时间点。

创建提交，需要我们至少在到暂存区有一次修改（刚才我们做了git add），然后输入命令：

$ git commit -m "第一次提交文件"

[master (root-commit) 3a7e461] 第一次提交文件

1 file changed, 1 insertion(+)

create mode 100644 hello.txt

现在我们已经提交了一个hello.txt文件了，其中-m是指定提交信息的备注信息。

接下来可以通过命令git status来查看是否还有文件未提交：

$ git status

On branch master

nothing to commit, working tree clean

此时说明没有任何文件未提交。

3、管理文件

(1)对比文件差异

文件添加到版本库成功之后，如果还需要修改文件，则在修改文件之后，再次执行git status命令，可以查看被修改文件现在的状态

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: hello.txt

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

上面的命令告诉我们 hello.txt文件已被修改，但是未被提交的修改。

接下来我想看下hello.txt文件到底改了什么内容，可以使用如下命令：

$ git diff hello.txt

diff --git a/hello.txt b/hello.txt

index 8d0e412..aed03f6 100644

--- a/hello.txt

+++ b/hello.txt

@@ -1 +1,2 @@

hello git

+second modify

\ No newline at end of file

可以看到，hello.txt文件内容从一行hello git改成了两行，添加了一行second modify内容。知道了对hello.txt文件做了什么修改后，我们可以放心的提交到仓库了。

(2)版本回退

如上，我们已经学会修改文件了，现在继续对hello.txt进行修改，增加一行内容为third modify，执行提交文件相关命令。

$ echo 'third modify' > hello.txt

$ git add hello.txt

$ git commit -m '第三次提交文件'

此时，我们已经对hello.txt文件进行了三次修改，可以通过git log命令查看历史记录

$ git log

commit 3efc530f20e1500799cb51ca64d9afc17bf2f95a (HEAD -> master)

Author: fmj <youremail@offcn.com>

Date: Wed Mar 13 20:56:18 2019 +0800

第三次提交文件

commit 5bc4e2de4c40867a916ea0403f69669f904e4725

Author: fmj <youremail@offcn.com>

Date: Wed Mar 13 20:42:20 2019 +0800

第二次提交文件

commit 3a7e461af83442a7ba76f0c6b36773f4cbf174cf

Author: fmj <youremail@offcn.com>

Date: Tue Mar 12 21:52:40 2019 +0800

第一次提交文件

git log命令显示从最近到最远的显示日志，我们可以看到最近三次提交。

现在如果想把版本回退到上一个版本，可以使用以下两种命令：

1)git reset --hard HEAD^

说明：此方法中HEAD^表示回退到上一个版本，如果要回退到上上个版本，则需要再增加一个^符号，即HEAD^^，以此类推，但是要回退到前100个版本的话，使用此方法就太不方便了。

以回退到上一个版本为例：

$ git reset --hard HEAD^

HEAD is now at 5bc4e2d 第二次提交文件

$ git log --pretty=oneline

5bc4e2de4c40867a916ea0403f69669f904e4725 (HEAD -> master) 第二次提交文件

3a7e461af83442a7ba76f0c6b36773f4cbf174cf 第一次提交文件

git log --pretty=oneline的作用是显示简要的日志信息。

通过查看日志，已经发现当前版本已经回退到上一个版本了。

2)git reset --hard HEAD~整数值

使用此方法，git reset --hard HEAD~100就可以非常方便的回退到前100个版本了。

3)git reset --hard 版本号

可以通过git reflog命令查看版本号

$ git reflog

5bc4e2d (HEAD -> master) HEAD@{0}: reset: moving to HEAD^

3efc530 HEAD@{1}: commit: 第三次提交文件

5bc4e2d (HEAD -> master) HEAD@{2}: reset: moving to HEAD^

cfb1b52 HEAD@{3}: commit: 第三次提交

5bc4e2d (HEAD -> master) HEAD@{4}: commit: 第二次提交文件

3a7e461 HEAD@{5}: commit (initial): 第一次提交文件

回退到指定的版本号：

$ git reset --hard 5bc4e2d

HEAD is now at 5bc4e2d 第二次提交文件

3)撤销修改

假如现在修改了文件内容还未提交版本，但是发现修改有误，所以现在需要马上恢复到以前的版本，可以通过git checkout命令进行操作。

$ echo 'third modify' > hello.txt

$ git checkout -- hello.txt

命令 git checkout -- hello.txt 意思就是，把hello.txt文件在工作区做的修改全部撤销。

注意：命令git checkout -- hello.txt 中的 -- 很重要，如果没有 -- 的话，那么命令变成创建分支了。

4)删除文件

一般情况下，可以直接在文件目录中把文件删了，或者使用rm命令：rm hello.txt ，如果想彻底从版本库中删掉了此文件的话，可以再执行commit命令提交。

4、分支管理

1)分支概念

几乎每一种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从开发主线上分离开来，然后在不影响主线的同时继续工作。

有人把 Git 的分支模型称为"必杀技特性"，而正是因为它，将 Git 从版本控制系统家族里区分出来。

一般企业中开发一个项目的分支策略：

主分支 master

开发分支 develop

功能分支 feature

预发布分支 release

bug 分支 fixbug

其它分支 other

主分支 master

代码库应该有一个、且仅有一个主分支。所有提供给用户使用的正式版本，都在这个主分支上发布。Git主分支的名字，默认叫做master。它是自动建立的，版本库初始化以后，默认就是在主分支在进行开发。

开发分支 develop

主分支只用来分布重大版本，日常开发应该在另一条分支上完成。我们把开发用的分支，叫做develop。这个分支可以用来生成代码的最新代码版本。如果想正式对外发布，就在master分支上，对develop分支进行"合并"（merge）。

功能分支 feature

功能分支，它是为了开发某种特定功能，从Develop分支上面分出来的。开发完成后，要再并入Develop。功能分支的名字，可以采用feature-\*的形式命名。

预发布分支 release

预发布分支，它是指发布正式版本之前（即合并到Master分支之前），我们可能需要有一个预发布的版本进行测试。预发布分支是从Develop分支上面 分出来的，预发布结束以后，必须合并进Develop和Master分支。它的命名，可以采用release-\*的形式。

bug 分支 fixbug

bug分支。软件正式发布以后，难免会出现bug。这时就需要创建一个分支，进行bug修补。修补bug分支是从Master分支上面分出来的。修补结束以后，再合并进Master和Develop分支。它的命名，可以采用fixbug-\*的形式。

其它分支 other

还有就是其它分支了，可以根据需要创建即可。

2)创建分支

创建分支命令：

git branch branchname

创建一个dev分支为例：

$ git branch dev

3)查看分支

查看分支命令：

$ git branch

dev

\* master

命令会列出所有分支，当前分支前面会标一个\*号。

此例的意思就是，我们有一个叫做"master"的分支和一个dev，并且master分支是当前分支。

当你执行 git init 的时候，缺省情况下 Git 就会为你创建"master"分支。

4)切换分支

切换分支命令：

$ git checkout dev

Switched to branch 'dev'

再来查看分支

$ git branch

\* dev

master

接下来，我们在创建一个hello.html文件并进行提交

$ touch hello.html

$ git add hello.html

$ git commit -m 'dev分支提交文件hello.html'

然后再切换到master分支

$ git checkout master

Switched to branch 'master'

$ ls

hello.txt

在master分支上并没有出现刚才创建的hello.html文件，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。

5)合并分支

把dev分支上的工作成果合并到master分支上

$ git merge dev

Updating 5bc4e2d..63146eb

Fast-forward

hello.html | 0

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 hello.html

再来查看master分支的内容

$ ls

hello.html hello.txt

git merge命令用于合并指定分支到当前分支。合并后，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。

6)删除分支

分支合并完成后，就可以dev分支删除了

$ git branch -d dev

Deleted branch dev (was 63146eb).

7.解决冲突

创建dev分支

$ git checkout -b dev

Switched to a new branch 'dev'

git checkout -b dev命令表示创建dev分支并切换到dev分支

在dev分支下给hello.html增加两行内容，并提交。

$ git add hello.html

$ git commit -m 'dev分支提交文件'

[dev e28198a] dev分支提交文件

1 file changed, 1 insertion(+)

切换到master分支，并在hello.html文件中增加一行内容，并提交。

$ git checkout master

Switched to branch 'master'

$ git add hello.html

$ git commit -m 'master分支第二次提交文件'

[master d3e69da] master分支第二次提交文件

1 file changed, 2 insertions(+)

在master分支和dev分支分别对hello.html文件进行了修改，接下来合并dev分支到master分支。

$ git merge dev

Auto-merging hello.html

CONFLICT (content): Merge conflict in hello.html

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

Git告诉我们，hello.html文件存在冲突，必须手动解决冲突后再提交。

通过git status命令也可以告诉我们冲突的文件：

$ git status

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: hello.html

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

直接查看hello.html文件的内容：

$ cat hello.html

1111111111111

2222222222222

<<<<<<< HEAD

3333333333333

=======

>>>>>>> dev

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容。

当Git无法自动合并分支时，就必须首先解决冲突，解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

### 4 git远程仓库

目前我们使用到的 Git 命令都是在本地执行，如果你想通过Git分享你的代码或者与其他开发人员合作。那么你就需要将数据放到一台其他开发人员能够连接的服务器上。

这里我们以Github作为远程仓库，所以需要在Github网站上https://github.com注册一个账号，这里就不赘述注册过程了。

1. 生成SSH Key

由于本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以我们需要配置验证信息，通过Git命令生成SSH Key：

ssh-keygen -t rsa -C "youremail@example.com"

后面的your\_email@youremail.com改为你在github上注册的邮箱，之后会要求确认路径和输入密码，使用默认的一路回车就行。生成的文件，如下图：

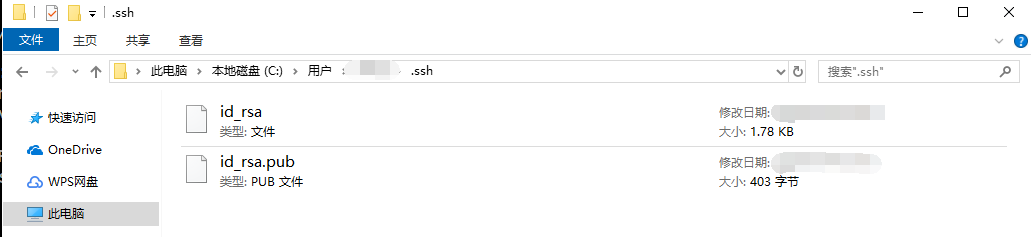


图8-5 SSH Key文件

1. Github设置

打开github网站并登陆，登陆后将用Git生成的SSH Key粘贴到指定的位置。

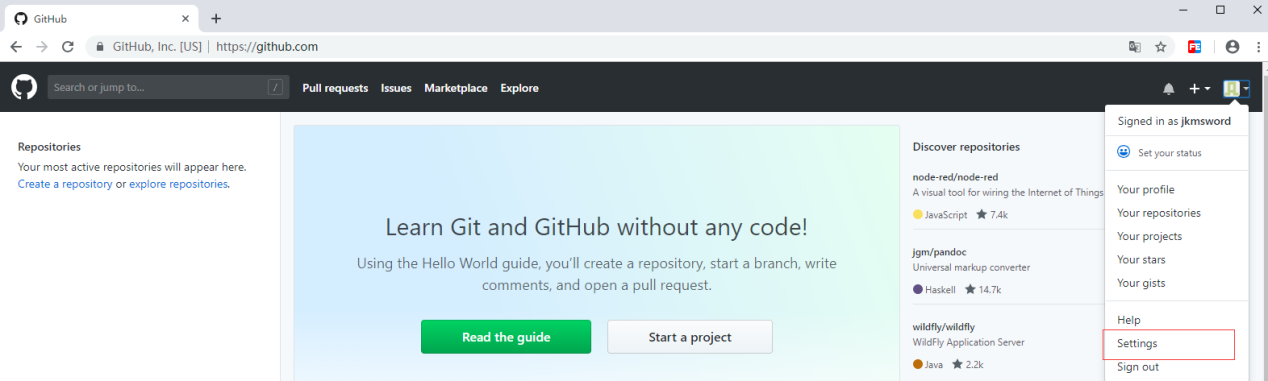


图8-6 Github用户中心设置

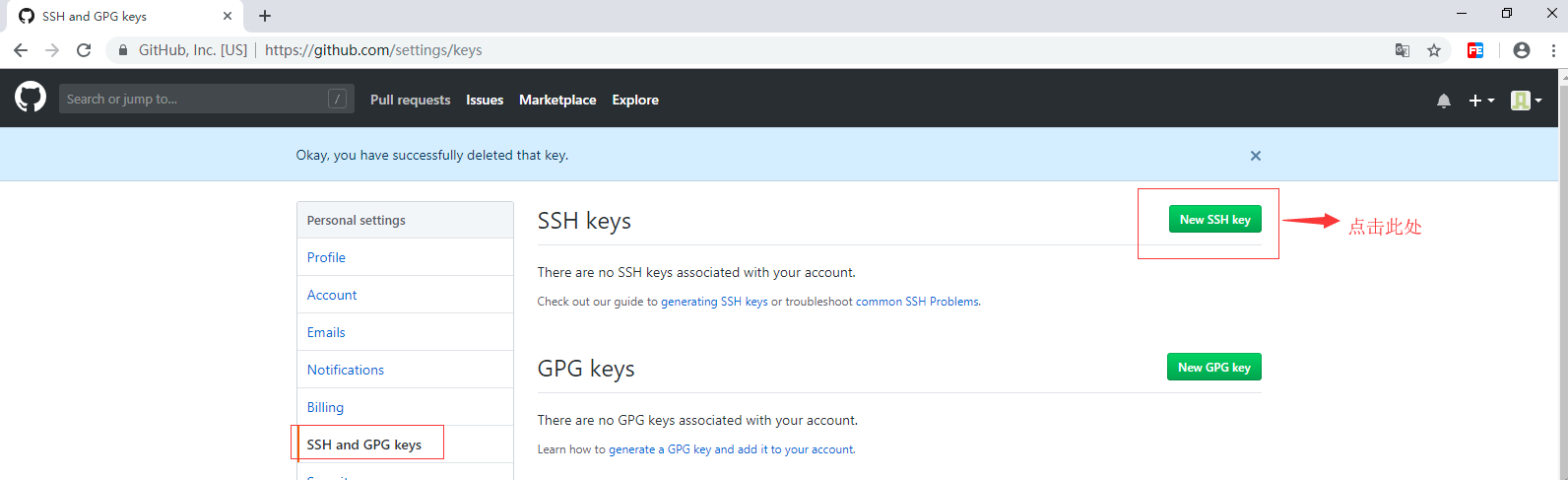


图8-7 创建SSH Key

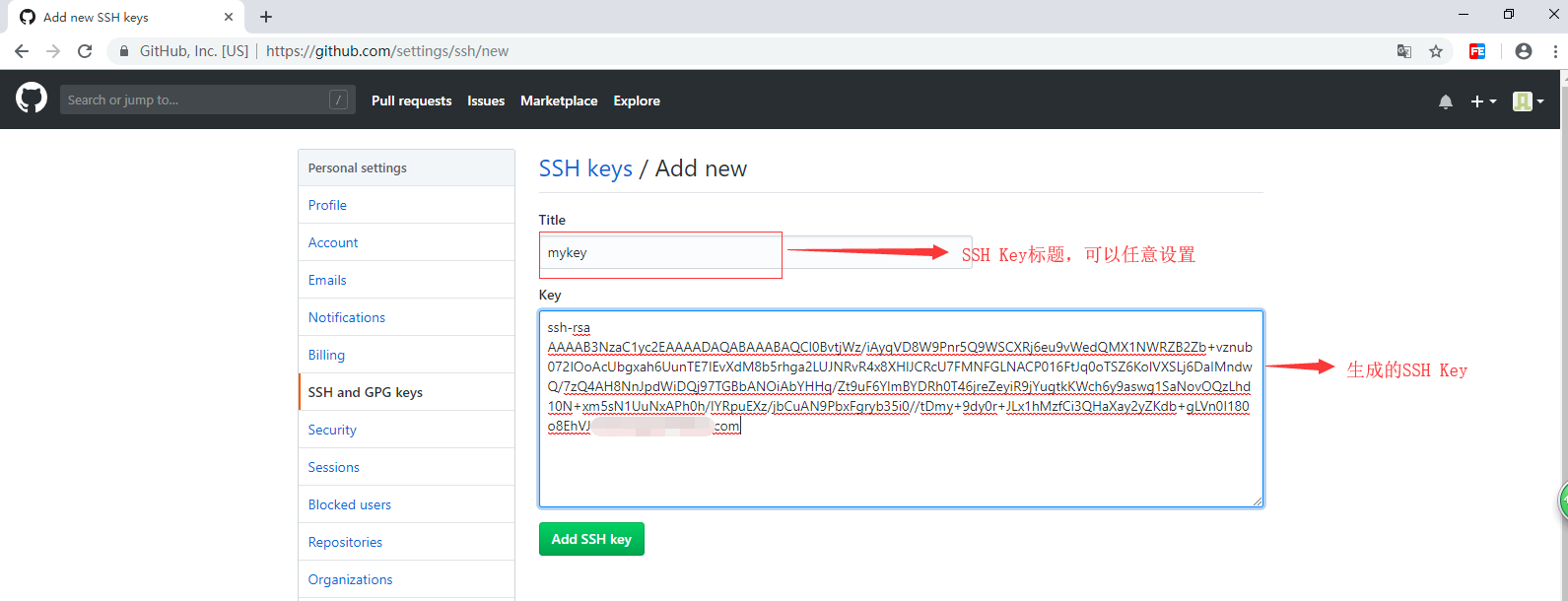


图8-8填写SSH Key信息

注意：需要将生成的文件id\_rsa.pub中的内容拷贝到Key对应的内容栏中。

验证是否设置成功，在Git中输入一下命令：

$ ssh -T git@github.com

Hi your name! You've successfully authenticated, but GitHub does not provide shell access.

1. 创建远程仓库

SSH Key设置好之后，需要在Github上创建一个仓库

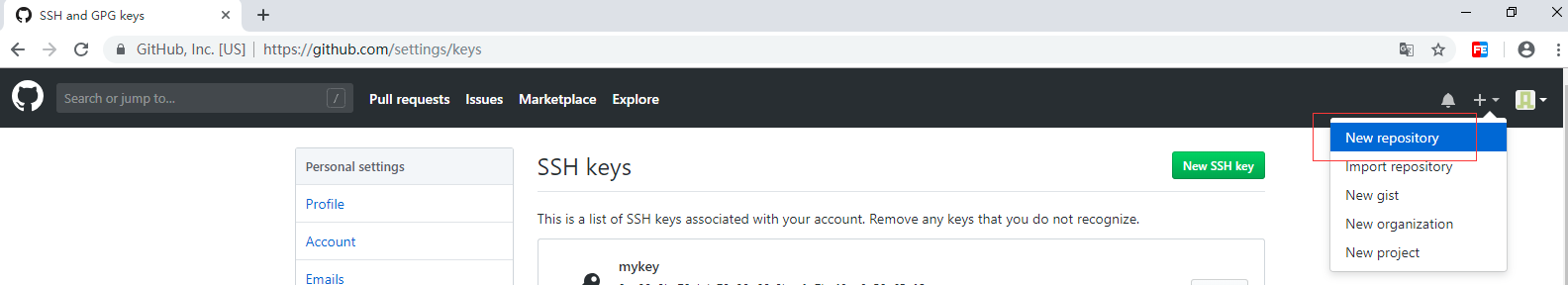


图8-9 创建Github远程仓库

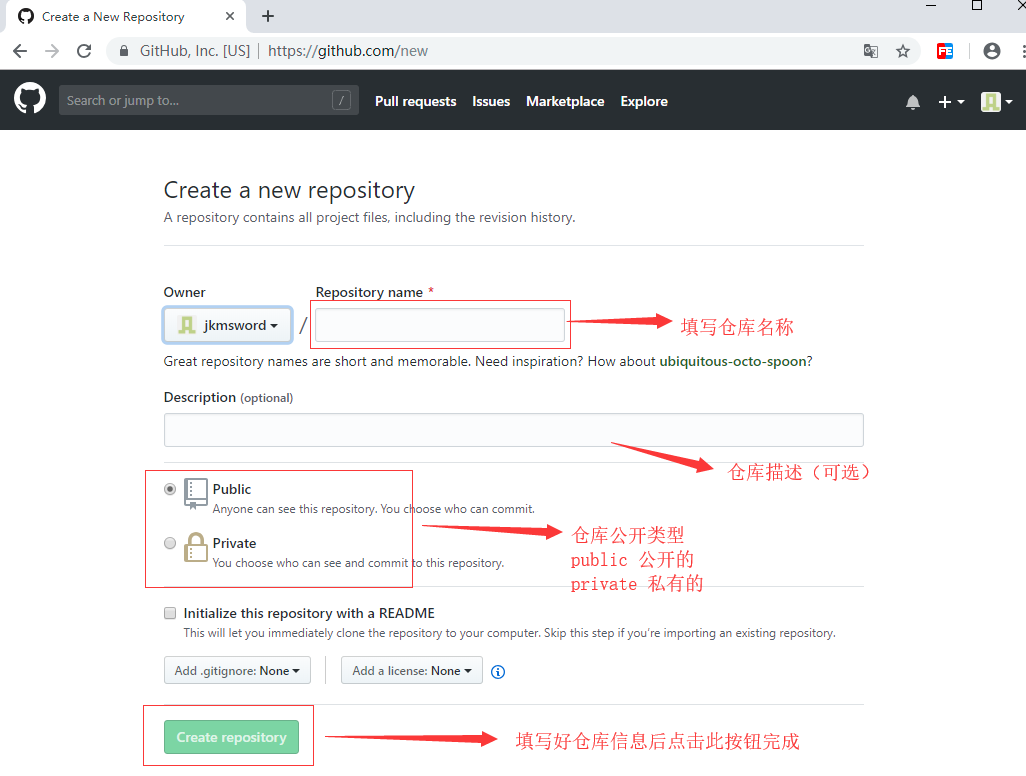


图8-10 填写Github仓库信息

1. git远程仓库操作

1)添加远程仓库

在Git中执行命令进行添加远程仓库

$ git remote add origin https://github.com/Github用户名/仓库名.git

2)推送版本至远程仓库

在本地把文件提交之后，执行以下命令将已提交的文件推送至Github远程仓库：

$ git push -u origin master

执行此命令后，会立刻弹出一个窗口，用于填写Github网站的账号和密码

填写上账号和密码无误后点击login按钮，就会出现如下图提示：

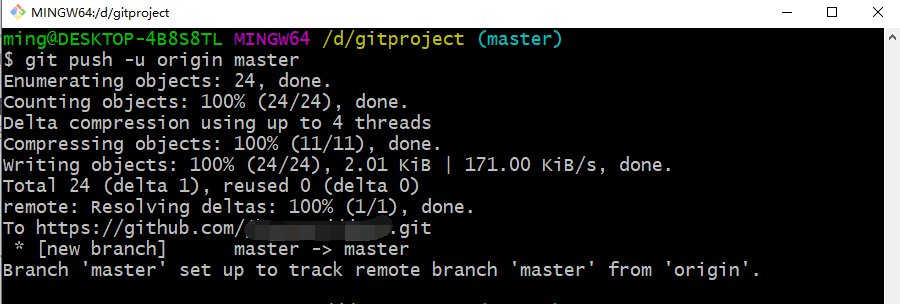


图8-11 Github推送成功提示

3)拉取远程仓库

Git 有两个命令用来提取远程仓库的更新。

①从远程仓库下载新分支与数据

$ git fetch

该命令执行完后需要执行git merge 远程分支到你所在的分支。

②从远端仓库提取数据并尝试合并到当前分支

$ git pull

该命令就是在执行 git fetch 之后紧接着执行 git merge 远程分支到你所在的任意分支。

4)删除远程仓库

如果要删除远程仓库，可以使用如下命令：

git remote rm [别名]

$ git remote rm origin

5)克隆远程仓库

如果要下载github上的项目，执行clone

$ git clone 远程仓库地址