


- [精选](#)
- [逛逛](#)

搜索

- [登录](#)
- [申请iG客](#)



'[UGeeker](#)' 发布于2016-08-08，阅读1079次，热度[2次](#)

## 30分钟教你学会Git

[#精选技术编程](#)[#精选技术编程](#)



Git近些年的火爆程度非同一般，这个版本控制系统被广泛地用在大型开源项目（比如Linux），不同规模的团队开发，以及独立开发者，甚至学生之中。

初学者非常容易被git里的[各种命令、参数](#)吓得不愿意继续去学。但实际上刚上手的时候，你并不需要了解所有命令的用途。你可以从掌握一些简单、常用又强大的命令开始，然后逐步去学习。这就是我们这篇文章要讲的内容。让我们快开始吧！



# 基本了解

Git是一些命令行工具的集合，可以用来跟踪、记录文件的变动，经常用于开源代码。比如你可以进行旧版本恢复、比对、分析、合并等等。这个过程被称之为版本控制。已经有一系列的版本控制系统，比如SVN、Mercurial、Perforce、CVS、Bitkeepe等等。

Git是分布式的，这意味着它并不依赖于中心服务器来保存你文件的旧版本。任何一台机器都可以有一个本地版本的控制系统，其实就是一个硬盘上的文件，我们称之为仓库（**repository**）。如果是多人协作的话，你还需要一个线上仓库，用来同步代码等信息。这就是[Github](#)、[BitBucket](#)等网站做的工作。

## 1.安装Git

在你的机器上安装git非常简单：

- Linux – 打开终端，然后通过包管理安装，在Ubuntu上命令是： *sudo apt-get install git*
- Windows – 推荐使用[git for windows](#)，它包括了图形工具以及命令行模拟器。
- OS X – 最简单的方式是使用[homebrew](#)安装，命令行执行*brew install git*

如果你是新手，推荐使用图形工具[Github desktop](#)和[Sourcetree](#)。不过即使使用图形界面的应用，知道一些基本的git命令依然很重要。接下来的内容我们集中在命令行控制上。

## 2.配置Git

安装完git，首要任务是做一些简单的配置，最重要的是用户名及邮箱，打开终端，执行以下命令。

```
$ git config --global user.name "My Name"
$ git config --global user.email myEmail@example.com
```

配置好这两项，Git就能记录下来是谁做的动作，一切都更有组织性了。

## 3.创建一个新仓库 – git init

git会把所有文件以及历史记录直接记录成一个文件夹保存在你的项目中。创建一个新的仓库，首先要去到项目路径下，执行*git init*。这时Git会创建一个隐藏的文件夹*.git*，所有的历史和配置信息都储存在其中。

比如我们在桌面创建一个文件夹 *git\_exercise*, 打开终端，输入：

```
$ cd Desktop/git_exercise/
$ git init
```

命令行会出现

```
Initialized empty Git repository in /home/user/Desktop/git_exercise/.git/
```

这说明我们的仓库已经建立好了，但现在是空的，试着新建一个*hello.txt*文本文件到这个文件夹里。

## 4.检查状态 – git status

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。在我们新建的仓库中执行*git status*会得到以下内容：

```
$ git status
On branch master
Initial commit
Untracked files:
  (use "git add ..." to include in what will be committed)
hello.txt
```

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。

5.暂存 - git add

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）。

这个例子中只有一个文件，让我们先add它：

```
$ 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
```

再次使用git status查看状态试试：

```
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached ..." to unstage)
    new file:   hello.txt
```

我们的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。

6.提交 - git commit

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 "Initial commit."
```

这就创建了一次从暂存区的提交（加入hello.txt），*-m “Initial commit.”*是用户对这次提交的描述，建议写成有意义的描述性信息。



远程仓库

到目前为止，我们的操作都是在本地的——只存在于*.git*文件中。为了能够协同开发，我们需要把代码部署到远程仓库服务器上。

1.链接远程仓库 - git remote add

为了能够上传到远程仓库，我们需要先建立起链接。在这篇教程中，我们远程仓库的地址为：*https://github.com/igeekbar/awesome-*



*project*。但你应该自己在Github、或BitBucket上搭建仓库，自己一步一步尝试。

把本地仓库链接到Github上，在命令行执行以下内容：

```
# This is only an example. Replace the URI with your own repository address.
$ git remote add origin https://github.com/igeekbar/awesome-project.git
```

一个项目可以同时拥有好几个远程仓库，为了区分通常会起不同的名字。通常主要的远程仓库被称为*origin*。

## 2.上传到服务器 - git push

把本地的提交传送到服务器的动作叫做*push*。每次我们要提交修改到服务器上时，都会使用到*git push*。

*git push*命令有两个参数，远程仓库的名字，以及分支的名字：

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/igeekbar/awesome-project.git
* [new branch]    master -> master
```

取决于你使用的服务器，push过程中你可能需要验证身份（输入用户名、密码，请先在网站上进行注册）。如果没有出差错，现在用浏览器看你的远程仓库上已经有*hello.txt*了。

## 3.克隆仓库 - git clone

其他人可以看到你放在Github上的开源项目，他们可以用*git clone*命令下载到本地。

```
$ git clone https://github.com/igeekbar/awesome-project.git
```

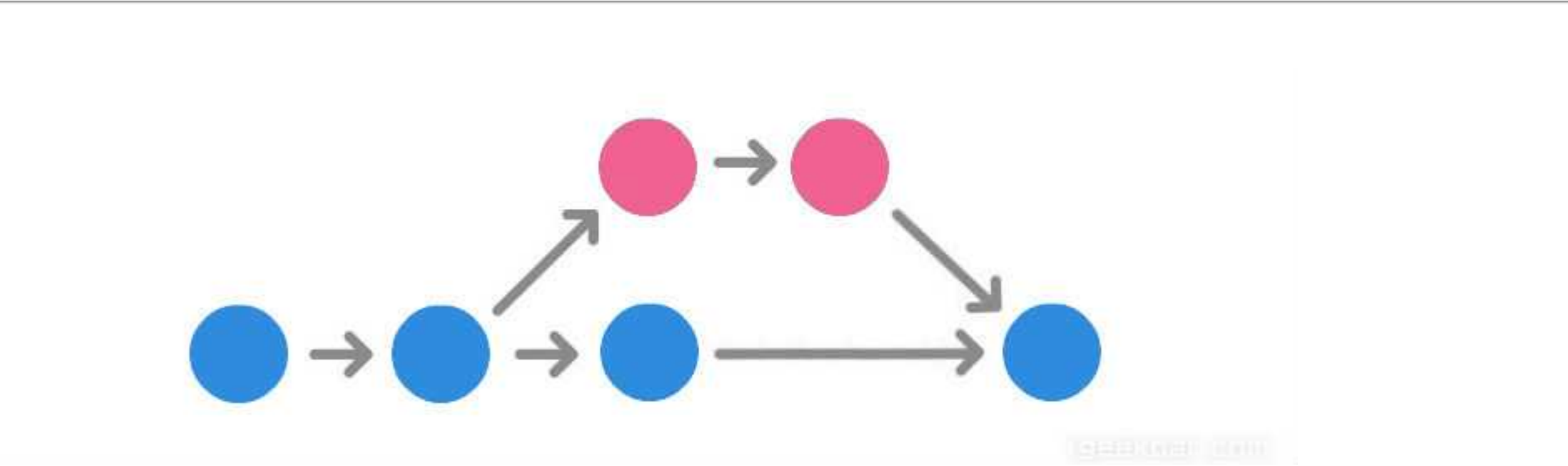
本地也会创建一个新的仓库，并自动将github上的版本设为远程仓库。

## 4.从服务器上获得修改 - git pull

如果你更新了远程仓库上的内容，其他人可以通过*git pull*命令拉取你的变动：

```
$ git pull origin master
From https://github.com/igeekbar/awesome-project
* branch      master    -> FETCH_HEAD
Already up-to-date.
```

因为在我们git clone之后还没有提交过修改，所有没有任何变动。



# 分支

当你在做一个新功能的时候，最好是在一个独立的区域上开发（原始项目的拷贝），通常称之为分支。分支之间相互独立，并且拥有自己的历史记录，直到你决定把他们合并到一起。这样做的原因是：

- 已经可以运行的稳定版本的代码不会被破坏
- 不同的功能可以由不同开发者同时开发
- 开发者可以专注于自己的分支，不用担心被其他人破坏
- 在不确定哪个版本更好之前，同一个特性可以在不同的分支上创建多个版本，便于比较

## 1.创建新分支 - git branch

每一个仓库的默认分支都叫**master**, 创建新分支可以用`git branch <name>`命令：

```
$ git branch amazing_new_feature
```

创建了一个名为`amazing_new_feature`的新分支，它目前和**master**分支是一样的内容。

## 2.切换分支 - git checkout

使用`git branch`，可以查看分支状态：

```
$ git branch
  amazing_new_feature
* master
```

\* 号表示当前活跃分支为`master`，现在我想在新分支上开发新的特性，使用`git checkout`切换分支。有一个参数表示要切换到的分支。

```
$ git checkout amazing_new_feature
```

## 3.合并分支 - git merge

我们在“`amazing_new_feature`”分支想添加一个`feature.txt`。和之前一样我们来创建文件、添加到暂存区、提交。

```
$ git add feature.txt
$ git commit -m "New feature complete."
```

新分支任务完成了，回到`master`分支。

```
$ git checkout master
```

现在去查看文件夹内容，你会惊奇地发现之前刚刚创建的`feature.txt`文件不见了，因为我们现在回到了`master`分支上，这里并没有`feature.txt`。想把文件添加到这里，我们需要使用`git merge`把`amazing_new_feature`分支合并到`master`上。

```
$ git merge amazing_new_feature
```

现在`master`分支是最新的了，`amazing_new_feature`分支可以删掉了。

```
$ git branch -d amazing_new_feature
```

---



## 进阶功能

在这篇教程的最后一节，我们来看一些高级并且实用的技巧。

### 1. 比对两个不同提交之间的差别

每次提交都有一个标识id，查看所有历史提交和他们的id，可以使用 *git log*:

```
$ git log
commit ba25c0ff30e1b2f0259157b42b9f8f5d174d80d7
Author: igeekbar
Date:  Fri July 29 17:15:28 2016 +0300
    New feature complete
commit b10cc1238e355c02a044ef9f9860811ff605c9b4
Author: igeekbar
Date:  Fri July 29 16:30:04 2016 +0300
    Added content to hello.txt
commit 09bd8cc171d7084e78e4d118a2346b7487dca059
Author: igeekbar
Date:  Thu July 28 17:52:14 2016 +0300
    Initial commit
```

id很长，但是当使用它的时候你并不需要复制整个字符串，前几个字符就够了。

查看某一次提交更新了什么，使用 *git show [commit]*:

```
$ git show b10cc123
commit b10cc1238e355c02a044ef9f9860811ff605c9b4
Author: igeekbar
Date:  Fri July 29 16:30:04 2016 +0300
    Added content to hello.txt
diff --git a/hello.txt b/hello.txt
index e69de29..b546a21 100644
--- a/hello.txt
+++ b/hello.txt
@@ -0,0 +1 @@
+Nice weather today, isn't it?
```

查看两次提交的不同，可以使用*git diff [commit-from]..[commit-to]*:

```
$ git diff 09bd8cc..ba25c0ff
diff --git a/feature.txt b/feature.txt
new file mode 100644
index 0000000..e69de29
diff --git a/hello.txt b/hello.txt
index e69de29..b546a21 100644
--- a/hello.txt
+++ b/hello.txt
@@ -0,0 +1 @@
+Nice weather today, isn't it?
```

比较首次提交和最后一次提交，我们可以看到中间所有的更改。使用*git difftool*命令可以用图形化界面查看所有更改。

2.回滚某个文件到之前的版本

Git允许我们将某个特定的文件回滚到特定的提交，使用的也是 *git checkout*命令。

下面我们将*hello.txt*回滚到最初的状态，需要指定回滚到哪个提交（以id作为参数），以及文件的全路径。

```
$ git checkout 09bd8cc1 hello.txt
```

3.回滚提交

如果你发现最新的一次提交忘记加入某个文件，或是信息输入的不正确，你可以通过 *git commit --amend*来改正，它会把最新的提交打回暂存区，并尝试重新提交。

如果是更复杂的情况，比如不是最新的提交除了问题，你可以使用*git revert*。

最新的一次提交别名也叫HEAD。

```
$ git revert HEAD
```

其他提交需要指明id:

```
$ git revert b10cc123
```

回滚提交时，发生冲突是非常频繁的。比如文件被指定回滚的提交之后的某次提交修改过，git不能正确回滚。

4.解决合并冲突

冲突经常出现在合并分支或者是拉取别人的代码。有些时候git能自动处理冲突，其他时候需要我们手动处理。

我们来看以下的例子，John 和 Tim 分别在各自的分支上写了一段代码，来显示一个数组中所有的元素。

John使用了for循环:

```
// Use a for loop to console.log contents.
for(var i=0; i<arr.length; i++) {
  console.log(arr[i]);
}
```

Tim使用forEach:

```
// Use forEach to console.log contents.
arr.forEach(function(item) {
  console.log(item);
});
```

它们都提交了代码到各自的分支上，现在假设John尝试合并Tim的代码:

```
$ git merge tim_branch
Auto-merging print_array.js
CONFLICT (content): Merge conflict in print_array.js
Automatic merge failed; fix conflicts and then commit the result.
```

这时候git并不能自动解决冲突，于是它在代码中插入冲突标记。

```
<<<<<< HEAD
// Use a for loop to console.log contents.
for(var i=0; i<arr.length; i++) {
  console.log(arr[i]);
}
=====
// Use forEach to console.log contents.
```

```
arr.forEach(function(item) {  
    console.log(item);  
});  
>>>>>> Tim's commit.
```

==== 号上方是当前最新一次提交，下方是冲突的代码。这样我们可以清晰地看出区别，决定使用哪一个版本，或者重新写一个。假设我们对于这两个版本都不满意，我们把代码改成以下代码：

```
// Not using for loop or forEach.  
// Use Array.toString() to console.log contents.  
console.log(arr.toString());
```

好了，再提交一下：

```
$ git add -A  
$ git commit -m "Array printing conflict resolved."
```

在大型项目中，我们可能在合并过程中出现很多冲突，大部分开发者会借助[GUI工具](#)来获得帮助，运行推行界面可以使用*git mergetool*命令。

## 5.配置 .gitignore

大部分项目中，会有些文件、文件夹是我们不想提交的。为了防止使用*git add -A*时不小心提交，我们可以利用*.gitignore*文件：

- 在项目根目录创建.gitignore文件
- 在文件中列出不需要提交的文件名、文件夹名，每个一行
- .gitignore文件需要像普通文件一样add、commit和push

通常会被ignore的文件有：

- log文件
- task runner builds
- node.js项目中的node\_modules文件夹
- IDEs比如NetBeans和IntelliJ生成的文件
- 个人笔记

以下是一个.gitignore文件的例子：

```
*.log  
build/  
node_modules/  
.idea/  
my_notes.txt
```

“/”说明是一个文件夹，里面的所有内容都被递归忽略

---

## 总结

Git教程到这里就结束啦！

Git很复杂，还有很多的特性和技巧等着你去挖掘，这篇教程只涵盖了冰山一角，希望你不要因为太多繁琐的命令而停下前进的脚步！加油！





本文二维码  
[#精选技术编程#精选技术编程](#)  
0 2 1079



UGeeker 设计其实是一种发自内心的艺术修行~~

设计灵感来自于梦境和现实的夹层中 ~~

全部评论

-  Cindy - 非典型程序媛

教程讲的很清楚~再推荐一个学习git分支的交互网站哈 <http://learngitbranching.js.org/>  
2016年08月1日 10:33:46 [回复](#)

[TERRi](#)(2016年08月9日 07:12:24): 这个教程也非常不错 推荐

[评论前请先申请个人账号或登录](#)