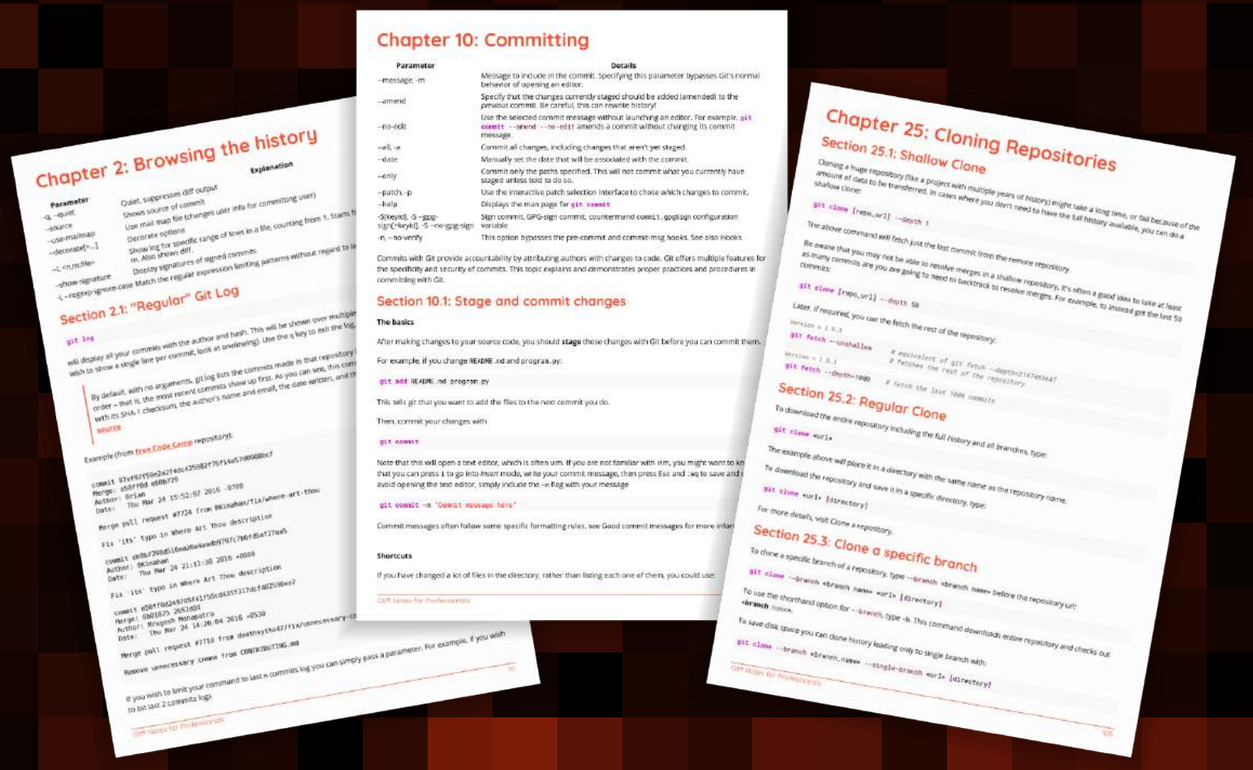
专业人士笔记

Git®

专业人士笔记



100+ 页

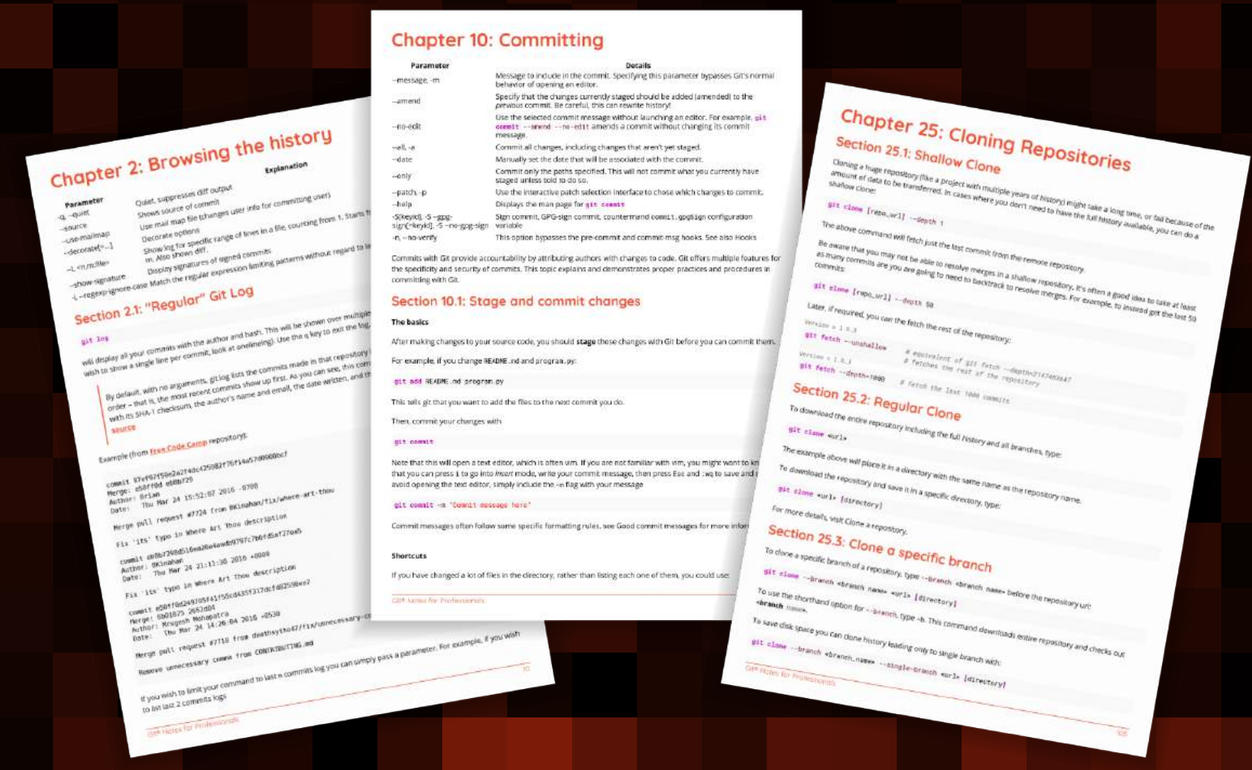
专业提示和技巧

GoalKicker.com

免费编程书籍

Git®

Notes for Professionals



100+ pages

of professional hints and tricks

GoalKicker.com

Free Programming Books

目录

Contents

关于

欢迎随意免费分享此PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/GitBook

本*Git® 专业人士笔记*一书汇编自Stack Overflow Documentation，内容由Stack Overflow的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费书籍，旨在教育用途，与官方Git®组织或公司及Stack Overflow无关。所有商标和注册商标均为其各自公司所有者所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/GitBook

This *Git® Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Git® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：Git入门

版本发布日期

2.13	2017-05-10
2.12	2017-02-24
2.11.1	2017-02-02
2.11	2016-11-29
2.10.2	2016-10-28
2.10	2016-09-02
2.9	2016-06-13
2.8	2016-03-28
2.7	2015-10-04
2.6	2015-09-28
2.5	2015-07-27
2.4	2015-04-30
2.3	2015-02-05
2.2	2014-11-26
2.1	2014-08-16
2.0	2014-05-28
1.9	2014-02-14
1.8.3	2013-05-24
1.8	2012-10-21
1.7.10	2012-04-06
1.7	2010-02-13
1.6.5	2009-10-10
1.6.3	2009-05-07
1.6	2008-08-17
1.5.3	2007-09-02
1.5	2007-02-14
1.4	2006-06-10
1.3	2006-04-18
1.2	2006-02-12
1.1	2006-01-08
1.0	2005-12-21
0.99	2005-07-11

第1.1节：创建你的第一个仓库，然后添加并提交文件

在命令行中，首先确认你已安装Git：

在所有操作系统上：

```
git --version
```

在类UNIX操作系统上：

Chapter 1: Getting started with Git

Version Release Date

2.13	2017-05-10
2.12	2017-02-24
2.11.1	2017-02-02
2.11	2016-11-29
2.10.2	2016-10-28
2.10	2016-09-02
2.9	2016-06-13
2.8	2016-03-28
2.7	2015-10-04
2.6	2015-09-28
2.5	2015-07-27
2.4	2015-04-30
2.3	2015-02-05
2.2	2014-11-26
2.1	2014-08-16
2.0	2014-05-28
1.9	2014-02-14
1.8.3	2013-05-24
1.8	2012-10-21
1.7.10	2012-04-06
1.7	2010-02-13
1.6.5	2009-10-10
1.6.3	2009-05-07
1.6	2008-08-17
1.5.3	2007-09-02
1.5	2007-02-14
1.4	2006-06-10
1.3	2006-04-18
1.2	2006-02-12
1.1	2006-01-08
1.0	2005-12-21
0.99	2005-07-11

Section 1.1: Create your first repository, then add and commit files

At the command line, first verify that you have Git installed:

On all operating systems:

```
git --version
```

On UNIX-like operating systems:

```
which git
```

如果没有返回任何内容，或者命令无法识别，您可能需要通过下载并运行安装程序来在系统上安装 Git。请参阅Git 主页，那里有非常清晰且易于理解的安装说明。

安装 Git 后，配置您的用户名和电子邮件地址。请在提交之前完成此操作。

Git 安装完成后，导航到您想要放入版本控制的目录，并创建一个空的 Git 仓库：

```
git init
```

这将创建一个隐藏文件夹.git，其中包含 Git 工作所需的底层结构。

接下来，检查 Git 将添加到新仓库中的文件；这一步需要特别注意：

```
git status
```

查看生成的文件列表；您可以告诉 Git 将哪些文件放入版本控制（避免添加包含密码等机密信息的文件，或仅会使仓库杂乱的文件）：

```
git add <文件/目录名 #1> <文件/目录名 #2> < … >
```

如果列表中的所有文件都应与所有有权访问仓库的人共享，使用一条命令即可添加当前目录及其子目录中的所有内容：

```
git add .
```

这将“暂存”所有文件，准备将它们添加到版本控制中，以便进行首次提交。

对于你不想纳入版本控制的文件，请在运行add命令之前，创建并填写一个名为.gitignore的文件。

提交所有已添加的文件，并附上提交信息：

```
git commit -m "Initial commit"
```

这会创建一个带有指定信息的新提交。提交就像是整个项目的保存点或快照。你现在可以将其推送（上传）到远程仓库，之后如果需要可以回到该提交。
如果省略了-m参数，默认编辑器将打开，你可以在其中编辑并保存提交信息。

添加远程仓库

要添加新的远程仓库，请在存放仓库的目录中使用终端执行git remote add命令。

git remote add命令需要两个参数：

1. 远程名称，例如origin
2. 远程 URL，例如 https://<your-git-service-address>/user/repo.git

```
git remote add origin https://<your-git-service-address>/owner/repository.git
```



```
which git
```

If nothing is returned, or the command is not recognized, you may have to install Git on your system by downloading and running the installer. See the Git homepage for exceptionally clear and easy installation instructions.

After installing Git, configure your username and email address. Do this *before* making a commit.

Once Git is installed, navigate to the directory you want to place under version control and create an empty Git repository:

```
git init
```

This creates a hidden folder, `.git`, which contains the plumbing needed for Git to work.

Next, check what files Git will add to your new repository; this step is worth special care:

```
git status
```

Review the resulting list of files; you can tell Git which of the files to place into version control (avoid adding files with confidential information such as passwords, or files that just clutter the repo):

```
git add <file/directory name #1> <file/directory name #2> < ... >
```

If all files in the list should be shared with everyone who has access to the repository, a single command will add everything in your current directory and its subdirectories:

```
git add .
```

This will "stage" all files to be added to version control, preparing them to be committed in your first commit.

For files that you want never under version control, create and populate a file named `.gitignore` before running the `add` command.

Commit all the files that have been added, along with a commit message:

```
git commit -m "Initial commit"
```

This creates a new commit with the given message. A commit is like a save or snapshot of your entire project. You can now push, or upload, it to a remote repository, and later you can jump back to it if necessary.
If you omit the `-m` parameter, your default editor will open and you can edit and save the commit message there.

Adding a remote

To add a new remote, use the `git remote add` command on the terminal, in the directory your repository is stored at.

The `git remote add` command takes two arguments:

1. A remote name, for example, `origin`
2. A remote URL, for example, `https://<your-git-service-address>/user/repo.git`

```
git remote add origin https://<your-git-service-address>/owner/repository.git
```

注意：在添加远程之前，您必须在您的 git 服务中创建所需的仓库，添加远程后您将能够推送/拉取提交。

第 1.2 节：克隆仓库

git clone 命令用于将服务器上的现有 Git 仓库复制到本地机器。

例如，要克隆一个 GitHub 项目：

```
cd <您希望克隆创建目录的路径>
git clone https://github.com/username/projectname.git
```

克隆一个 BitBucket 项目：

```
cd <您希望克隆创建目录的路径>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

这将在本地机器上创建一个名为`projectname`的目录，包含远程 Git 仓库中的所有文件。这包括项目的源文件，以及一个`.git`子目录，里面包含项目的完整历史记录和配置。

要指定不同的目录名称，例如MyFolder：

```
git clone https://github.com/username/projectname.git MyFolder
```

或者克隆到当前目录：

```
git clone https://github.com/username/projectname.git .
```

注意：

1. 当克隆到指定目录时，该目录必须为空或不存在。
2. 你也可以使用ssh版本的命令：

```
git clone git@github.com:username/projectname.git
```

https 版本和 ssh 版本是等效的。然而，一些托管服务如 GitHub 建议你使用 https 而不是 ssh。

第 1.3 节：共享代码

要共享你的代码，你需要在远程服务器上创建一个仓库，然后将本地仓库复制到该远程仓库。

为了最小化远程服务器上的空间使用，你创建一个裸仓库：只包含 .git 对象且不在文件系统中创建工作副本。作为额外好处，你将此远程设置为上游服务器，以便轻松与其他程序员共享更新。

在远程服务器上：

```
git init --bare /path/to/repo.git
```

在本地机器上：

NOTE: Before adding the remote you have to create the required repository in your git service, You'll be able to push/pull commits after adding your remote.

Section 1.2: Clone a repository

The `git clone` command is used to copy an existing Git repository from a server to the local machine.

For example, to clone a GitHub project:

```
cd <path where you would like the clone to create a directory>
git clone https://github.com/username/projectname.git
```

To clone a BitBucket project:

```
cd <path where you would like the clone to create a directory>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

This creates a directory called `projectname` on the local machine, containing all the files in the remote Git repository. This includes source files for the project, as well as a `.git` sub-directory which contains the entire history and configuration for the project.

To specify a different name of the directory, e.g. `MyFolder`:

```
git clone https://github.com/username/projectname.git MyFolder
```

Or to clone in the current directory:

```
git clone https://github.com/username/projectname.git .
```

Note:

1. When cloning to a specified directory, the directory must be empty or non-existent.
2. You can also use the `ssh` version of the command:

```
git clone git@github.com:username/projectname.git
```

The `https` version and the `ssh` version are equivalent. However, some hosting services such as GitHub recommend that you use `https` rather than `ssh`.

Section 1.3: Sharing code

To share your code you create a repository on a remote server to which you will copy your local repository.

To minimize the use of space on the remote server you create a bare repository: one which has only the `.git` objects and doesn't create a working copy in the filesystem. As a bonus you set this remote as an upstream server to easily share updates with other programmers.

On the remote server:

```
git init --bare /path/to/repo.git
```

On the local machine:

```
git remote add origin ssh://username@server:/path/to/repo.git
```

（注意 ssh: 只是访问远程仓库的一种可能方式。）现在将你的本地仓库复制到

远程：

```
git push --set-upstream origin master
```

添加--set-upstream（或-u）会创建一个上游（跟踪）引用，该引用被无参数的Git命令使用，例如git pull。

第1.4节：设置您的用户名和邮箱

```
您需要设置您的身份*在*创建任何提交之前。这样提交才会有正确的作者姓名和邮箱关联。
```

这与推送到远程仓库时的身份验证无关（例如使用您的GitHub、BitBucket或GitLab账户推送到远程仓库时）

要为*所有*仓库声明该身份，请使用`git config --global`
这会将设置存储在您的用户`.gitconfig`文件中：例如`$HOME/.gitconfig`，Windows系统中为
`%USERPROFILE%\.gitconfig`。

```
git config --global user.name "您的姓名"
git config --global user.email mail@example.com
```

要为单个仓库声明身份，请在仓库内使用git config。
这会将设置存储在单个仓库内的文件$GIT_DIR/config中。例如
/路径/到/您的/仓库/.git/config。

```
cd /路径/到/我的/仓库
git config user.name "您在工作时的登录名"
git config user.email mail_at_work@example.com
```

存储在仓库配置文件中的设置在使用该仓库时将优先于全局配置。

提示：如果你有不同的身份（一个用于开源项目，一个用于工作，一个用于私人仓库，……），并且你
不想忘记为你正在处理的每个不同仓库设置正确的身份：

- **移除全局身份**

```
git config --global --remove-section user.name
git config --global --remove-section user.email
```

版本 ≥ 2.8

- 强制 git 仅在仓库设置中查找你的身份，而不在全局配置中查找：

```
git config --global user.useConfigOnly true
```

这样，如果你忘记为某个仓库设置`user.name`和`user.email`并尝试提交，你
将看到：


```
git remote add origin ssh://username@server:/path/to/repo.git
```

(Note that `ssh:` is just one possible way of accessing the remote repository.)

Now copy your local repository to the remote:

```
git push --set-upstream origin master
```

Adding `--set-upstream` (or `-u`) created an upstream (tracking) reference which is used by argument-less Git commands, e.g. `git pull`.

Section 1.4: Setting your user name and email

```
You need to set who you are *before* creating any commit.  That will allow commits to have the
right author name and email associated to them.
```

It has nothing to do with authentication when pushing to a remote repository (e.g. when pushing to a remote repository using your GitHub, BitBucket, or GitLab account)

To declare that identity for *all* repositories, use `git config --global`
This will store the setting in your user's `.gitconfig` file: e.g. `$HOME/.gitconfig` or for Windows,
`%USERPROFILE%\.gitconfig`.

```
git config --global user.name "Your Name"
git config --global user.email mail@example.com
```

To declare an identity for a single repository, use `git config` inside a repo.
This will store the setting inside the individual repository, in the file `$GIT_DIR/config`. e.g.
`/path/to/your/repo/.git/config`.

```
cd /path/to/my/repo
git config user.name "Your Login At Work"
git config user.email mail_at_work@example.com
```

Settings stored in a repository's config file will take precedence over the global config when you use that repository.

Tips: if you have different identities (one for open-source project, one at work, one for private repos, ...), and you don't want to forget to set the right one for each different repos you are working on:

- **Remove a global identity**

```
git config --global --remove-section user.name
git config --global --remove-section user.email
```

Version ≥ 2.8

- To force git to look for your identity only within a repository's settings, not in the global config:

```
git config --global user.useConfigOnly true
```

That way, if you forget to set your `user.name` and `user.email` for a given repository and try to make a commit, you will see:

```
未提供名称且自动检测已禁用
未提供邮箱且自动检测已禁用
```

第1.5节：设置上游远程

如果你克隆了一个分支（例如 Github 上的开源项目），你可能没有权限推送到上游仓库，因此你需要既有自己的分支，也能拉取上游仓库的内容。

首先检查远程仓库名称：

```
$ git remote -v
origin    https://github.com/myusername/repo.git (fetch)
origin    https://github.com/myusername/repo.git (push)
upstream  # 这行可能存在也可能不存在
```

如果upstream已经存在（某些Git版本中会有），你需要设置URL（当前为空）：

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

如果upstream不存在，或者你还想添加朋友/同事的分支（当前不存在）：

```
$ git remote add upstream https://github.com/projectusername/repo.git
$ git remote add dave https://github.com/dave/repo.git
```

第1.6节：了解命令

要获取任何git命令的更多信息——即该命令的功能、可用选项及其他文档——使用--help选项或help命令。

例如，要获取关于git diff命令的所有可用信息，请使用：

```
git diff --help
git help diff
```

同样，要获取关于status命令的所有可用信息，请使用：

```
git status --help
git help status
```

如果你只想快速查看最常用命令行参数的含义，请使用-h：

```
git checkout -h
```

第1.7节：为Git设置SSH

如果你使用的是Windows，请打开Git Bash。如果你使用的是Mac或Linux，请打开你的终端。

在生成SSH密钥之前，你可以检查是否已有现有的SSH密钥。

列出你的~/.ssh目录内容：

```
$ ls -al ~/.ssh
# 列出你 ~/.ssh 目录中的所有文件
```



```
no name was given and auto-detection is disabled
no email was given and auto-detection is disabled
```

Section 1.5: Setting up the upstream remote

If you have cloned a fork (e.g. an open source project on Github) you may not have push access to the upstream repository, so you need both your fork but be able to fetch the upstream repository.

First check the remote names:

```
$ git remote -v
origin    https://github.com/myusername/repo.git (fetch)
origin    https://github.com/myusername/repo.git (push)
upstream  # this line may or may not be here
```

If `upstream` is there already (it is on *some* Git versions) you need to set the URL (currently it's empty):

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

If the upstream is **not** there, or if you also want to add a friend/colleague's fork (currently they do not exist):

```
$ git remote add upstream https://github.com/projectusername/repo.git
$ git remote add dave https://github.com/dave/repo.git
```

Section 1.6: Learning about a command

To get more information about any git command – i.e. details about what the command does, available options and other documentation – use the `--help` option or the **`help`** command.

For example, to get all available information about the **`git diff`** command, use:

```
git diff --help
git help diff
```

Similarly, to get all available information about the `status` command, use:

```
git status --help
git help status
```

If you only want a quick help showing you the meaning of the most used command line flags, use `-h`:

```
git checkout -h
```

Section 1.7: Set up SSH for Git

If you are using **Windows** open Git Bash. If you are using **Mac** or **Linux** open your Terminal.

Before you generate an SSH key, you can check to see if you have any existing SSH keys.

List the contents of your `~/.ssh` directory:

```
$ ls -al ~/.ssh
# Lists all the files in your ~/.ssh directory
```

检查目录列表，查看你是否已经有公用 SSH 密钥。默认情况下，公钥的文件名
通常是以下之一：

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

如果你看到已有的公钥和私钥对，并且想在你的 Bitbucket、GitHub（或类似）账户上使用，可以复制 id_*.pub
文件的内容。

如果没有，你可以使用以下命令创建一对新的公钥和私钥：

```
$ ssh-keygen
```

按回车键接受默认位置。根据提示输入并确认密码短语，或者留空。

确保你的 SSH 密钥已添加到 ssh-agent。如果 ssh-agent 尚未运行，请在后台启动它：

```
$ eval "$(ssh-agent -s)"
```

将你的 SSH 密钥添加到 ssh-agent。注意，你需要将命令中的 id_rsa 替换为你的私钥文件名：

```
$ ssh-add ~/.ssh/id_rsa
```

如果你想将现有仓库的上游地址从 HTTPS 改为 SSH，可以运行以下命令：

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

如果你想通过 SSH 克隆一个新的仓库，可以运行以下命令：

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

第1.8节：Git安装

让我们开始使用Git。首先——你必须先安装它。你可以通过多种方式获得它；主要有两种方式是从源码安装或安装适合你平台的现成软件包。

从源码安装

如果可以的话，通常建议从源码安装Git，因为你可以获得最新版本。每个版本的Git通常都会包含有用的用户界面改进，所以如果你熟悉从源码编译软件，获取最新版本通常是最佳选择。许多Linux发行版包含的包版本都很旧；除非你使用的是非常新的发行版或使用回溯包，否则从源码安装可能是最好的选择。

要安装Git，你需要安装Git依赖的以下库：curl、zlib、openssl、expat和libiconv。
例如，如果你使用的是带有yum（如Fedora）或apt-get（如基于Debian的系统）的系统，你可以使用以下命令之一来安装所有依赖项：

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


Check the directory listing to see if you already have a public SSH key. By default the filenames of the public keys are one of the following:

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

If you see an existing public and private key pair listed that you would like to use on your Bitbucket, GitHub (or similar) account you can copy the contents of the `id_*.pub` file.

If not, you can create a new public and private key pair with the following command:

```
$ ssh-keygen
```

Press the Enter or Return key to accept the default location. Enter and re-enter a passphrase when prompted, or leave it empty.

Ensure your SSH key is added to the ssh-agent. Start the ssh-agent in the background if it's not already running:

```
$ eval "$(ssh-agent -s)"
```

Add you SSH key to the ssh-agent. Notice that you'll need te replace `id_rsa` in the command with the name of your **private key file**:

```
$ ssh-add ~/.ssh/id_rsa
```

If you want to change the upstream of an existing repository from HTTPS to SSH you can run the following command:

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

In order to clone a new repository over SSH you can run the following command:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Section 1.8: Git Installation

Let's get into using some Git. First things first—you have to install it. You can get it a number of ways; the two major ones are to install it from source or to install an existing package for your platform.

Installing from Source

If you can, it's generally useful to install Git from source, because you'll get the most recent version. Each version of Git tends to include useful UI enhancements, so getting the latest version is often the best route if you feel comfortable compiling software from source. It is also the case that many Linux distributions contain very old packages; so unless you're on a very up-to-date distro or are using backports, installing from source may be the best bet.

To install Git, you need to have the following libraries that Git depends on: curl, zlib, openssl, expat, and libiconv. For example, if you're on a system that has yum (such as Fedora) or apt-get (such as a Debian based system), you can use one of these commands to install all of the dependencies:

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

```
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

当你安装好所有必要的依赖后，就可以去Git官网获取最新的快照版本：

http://git-scm.com/download 然后，编译并安装：

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

完成后，你也可以通过Git自身来获取更新：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

在Linux上安装

如果你想通过二进制安装程序在Linux上安装Git，通常可以通过你所使用发行版自带的基本包管理工具来完成。
如果你使用的是Fedora，可以使用yum：

```
$ yum install git
```

或者如果你使用的是基于Debian的发行版，比如Ubuntu，可以尝试apt-get：

```
$ apt-get install git
```

在Mac上安装

在Mac上安装Git有三种简单方法。最简单的是使用图形化的Git安装程序，你可以从SourceForge页面下载。

http://sourceforge.net/projects/git-osx-installer/

图1-7. Git OS X安装程序。另一种主要方式是通过MacPorts（http://www.macports.org）安装Git。如果你已经安装了MacPorts，可以通过以下命令安装Git：

```
$ sudo port install git +svn +doc +bash_completion +gitweb
```

你不必添加所有额外选项，但你可能会想包含+svn，以防将来需要使用Git与Subversion仓库一起工作（参见第8章）。

Homebrew（http://brew.sh/） 是安装 Git 的另一种选择。如果你已经安装了 Homebrew，可以通过以下命令安装 Git

```
$ brew install git
```

在 Windows 上安装

在 Windows 上安装 Git 非常简单。msysGit 项目提供了较为简便的安装流程。只需从 GitHub 页面下载安装程序 exe 文件，然后运行它：

```
http://msysgit.github.io
```



```
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

When you have all the necessary dependencies, you can go ahead and grab the latest snapshot from the Git web site:

http://git-scm.com/download Then, compile and install:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

After this is done, you can also get Git via Git itself for updates:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Installing on Linux

If you want to install Git on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora, you can use yum:

```
$ yum install git
```

Or if you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ apt-get install git
```

Installing on Mac

There are three easy ways to install Git on a Mac. The easiest is to use the graphical Git installer, which you can download from the SourceForge page.

http://sourceforge.net/projects/git-osx-installer/

Figure 1-7. Git OS X installer. The other major way is to install Git via MacPorts (http://www.macports.org). If you have MacPorts installed, install Git via

```
$ sudo port install git +svn +doc +bash_completion +gitweb
```

You don't have to add all the extras, but you'll probably want to include +svn in case you ever have to use Git with Subversion repositories (see Chapter 8).

Homebrew (http://brew.sh/) is another alternative to install Git. If you have Homebrew installed, install Git via

```
$ brew install git
```

Installing on Windows

Installing Git on Windows is very easy. The msysGit project has one of the easier installation procedures. Simply download the installer exe file from the GitHub page, and run it:

```
http://msysgit.github.io
```

安装完成后，你将同时拥有命令行版本（包括稍后会用到的 SSH 客户端）和标准的图形界面。

关于 Windows 使用的说明：你应该使用随附的 msysGit shell（类 Unix 风格）来使用 Git，这样可以运行本书中给出的复杂命令行。如果由于某些原因需要使用原生 Windows shell / 命令行控制台，则必须使用双引号替代单引号（用于包含空格的参数），并且如果参数是行尾且以脱字符（^）结尾，必须对其加引号，因为在 Windows 中该符号表示续行。

After it's installed, you have both a command-line version (including an SSH client that will come in handy later) and the standard GUI.

Note on Windows usage: you should use Git with the provided msysGit shell (Unix style), it allows to use the complex lines of command given in this book. If you need, for some reason, to use the native Windows shell / command line console, you have to use double quotes instead of single quotes (for parameters with spaces in them) and you must quote the parameters ending with the circumflex accent (^) if they are last on the line, as it is a continuation symbol in Windows.

第 2 章：浏览历史

参数	说明
-q, --quiet	安静模式，抑制差异输出
--source	显示提交的来源
--use-mailmap	使用邮件映射文件（更改提交用户的信息）
--decorate[=...]	装饰选项
--L <n,m:file>	显示文件中特定行范围的日志，行号从1开始。起始于第n行，直到第m行。也显示差异。
--show-signature	显示已签名提交的签名
-i, --regexp-ignore-case	匹配正则表达式时忽略字母大小写的限制模式

第2.1节：“常规”Git日志

```
git log
```

将显示所有提交及其作者和哈希。每个提交将显示多行。（如果你想每个提交显示一行，请查看单行显示）。使用q键退出日志。

> 默认情况下，无参数时，git log 会按时间倒序列出该仓库中的提交–即最新的提交最先显示。如你所见，该命令列出每个提交的SHA-1校验和、作者姓名和邮箱、提交日期以及提交信息。-来源

示例（来自Free Code Camp仓库）：

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
作者：Brian
日期：2016年3月24日星期四 15:52:07 -0700

合并拉取请求 #7724 来自 BKinahan/fix/where-art-thou

修正 Where Art Thou 描述中的“its”拼写错误

提交 eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
作者：BKinahan
日期：2016年3月24日 星期四 21:11:36 +0000

修正 Where Art Thou 描述中的“its”拼写错误

提交 e50ff0d249705f41f55cd435f317dcfd02590ee7
合并：6b01875 2652d04
作者：Mrugesh Mohapatra
日期：2016年3月24日 星期四 14:26:04 +0530

合并拉取请求 #7718 来自 deathsythe47/fix/unnecessary-comma

从 CONTRIBUTING.md 中移除多余的逗号
```

如果你想限制命令只显示最近的 n 次提交日志，可以简单地传入一个参数。例如，如果你想列出最近的2次提交日志

Chapter 2: Browsing the history

Parameter	Explanation
-q, --quiet	Quiet, suppresses diff output
--source	Shows source of commit
--use-mailmap	Use mail map file (changes user info for committing user)
--decorate[=...]	Decorate options
--L <n,m:file>	Show log for specific range of lines in a file, counting from 1. Starts from line n, goes to line m. Also shows diff.
--show-signature	Display signatures of signed commits
-i, --regexp-ignore-case	Match the regular expression limiting patterns without regard to letter case

Section 2.1: "Regular" Git Log

```
git log
```

will display all your commits with the author and hash. This will be shown over multiple lines per commit. (If you wish to show a single line per commit, look at onelineing). Use the q key to exit the log.

> By default, with no arguments, git log lists the commits made in that repository in reverse chronological order – that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message. - **source**

Example (from **Free Code Camp** repository):

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
Author: Brian
Date:   Thu Mar 24 15:52:07 2016 -0700

Merge pull request #7724 from BKinahan/fix/where-art-thou

Fix 'its' typo in Where Art Thou description

commit eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
Author: BKinahan
Date:   Thu Mar 24 21:11:36 2016 +0000

Fix 'its' typo in Where Art Thou description

commit e50ff0d249705f41f55cd435f317dcfd02590ee7
Merge: 6b01875 2652d04
Author: Mrugesh Mohapatra
Date:   Thu Mar 24 14:26:04 2016 +0530

Merge pull request #7718 from deathsythe47/fix/unnecessary-comma

Remove unnecessary comma from CONTRIBUTING.md
```

If you wish to limit your command to last n commits log you can simply pass a parameter. For example, if you wish to list last 2 commits logs

```
git log -2
```

第2.2节：更美观的日志

要以更美观的图形结构查看日志，请使用：

```
git log --decorate --oneline --graph
```

示例输出：

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
*   9b601ea 合并分支 'jk/difftool-in-subdir' 到 maint
|\
| * 32b8c58 difftool：使用 Git::* 函数代替传递状态
| * 98f917e difftool：避免使用 $GIT_DIR 和 $GIT_WORK_TREE
| * 9ec26e7 difftool：修正子目录中的参数处理
* |   f4fd627 合并分支 'jk/reset-ident-time-per-commit' 到 maint
...
```

由于这是一个相当大的命令，你可以分配一个别名：

```
git config --global alias.lol "log --decorate --oneline --graph"
```

使用别名版本：

```
# 当前分支的历史：
git lol

# 活跃分支（HEAD）、develop 和 origin/master 分支的合并历史：
git lol HEAD develop origin/master

# 仓库中所有内容的合并历史：
git lol --all
```

第2.3节：日志着色

```
git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)
%C(yellow)<%an>%Creset'
```

format 选项允许你指定自定义的日志输出格式：

参数	详情
%C(color_name)	选项为其后输出的内容着色
%h 或 %H	缩写提交哈希（完整哈希请使用 %H）
%Creset	重置颜色为默认终端颜色
%d	引用名称
%s	主题 [提交信息]
%cr	提交者日期，相对于当前日期
%an	作者姓名

第2.4节：单行日志

```
git log --oneline
```



```
git log -2
```

Section 2.2: Prettier log

To see the log in a prettier graph-like structure use:

```
git log --decorate --oneline --graph
```

sample output :

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
*   9b601ea Merge branch 'jk/difftool-in-subdir' into maint
|\
| * 32b8c58 difftool: use Git::* functions instead of passing around state
| * 98f917e difftool: avoid $GIT_DIR and $GIT_WORK_TREE
| * 9ec26e7 difftool: fix argument handling in subdirs
* |   f4fd627 Merge branch 'jk/reset-ident-time-per-commit' into maint
...
```

Since it's a pretty big command, you can assign an alias:

```
git config --global alias.lol "log --decorate --oneline --graph"
```

To use the alias version:

```
# history of current branch :
git lol

# combined history of active branch (HEAD), develop and origin/master branches :
git lol HEAD develop origin/master

# combined history of everything in your repo :
git lol --all
```

Section 2.3: Colorize Logs

```
git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)
%C(yellow)<%an>%Creset'
```

The `format` option allows you to specify your own log output format:

Parameter	Details
%C(color_name)	option colors the output that comes after it
%h or %H	abbreviates commit hash (use %H for complete hash)
%Creset	resets color to default terminal color
%d	ref names
%s	subject [commit message]
%cr	committer date, relative to current date
%an	author name

Section 2.4: Oneline log

```
git log --oneline
```

将显示所有提交，只显示哈希值的前半部分和提交信息。每个提交将占据一行，正如“oneline”标志所示。

> oneline 选项将每个提交打印在一行上，这在查看大量提交时非常有用。- 来源

示例（来自 Free Code Camp 仓库，与另一个示例中的相同代码部分）：

```
87ef97f 合并拉取请求 #7724 来自 BKinahan/fix/where-art-thou
eb8b729 修正 Where Art Thou 描述中的“its”拼写错误
e50ff0d 合并拉取请求 #7718 来自 deathsythe47/fix/unnecessary-comma
2652d04 从 CONTRIBUTING.md 中移除多余的逗号
6b01875 合并拉取请求 #7667 来自 zerkms/patch-1
766f088 修正赋值运算符术语
d1e2468 合并拉取请求 #7690 来自 BKinahan/fix/unsubscribe-crash
bed9de2 合并拉取请求 #7657 来自 Rafase282/fix/
```

如果你想限制命令只显示最近的 n 次提交日志，可以简单地传入一个参数。例如，如果你想列出最近的 2 次提交日志

```
git log -2 --oneline
```

第2.5节：日志搜索

```
git log -S"#define SAMPLES"
```

搜索特定字符串的添加或删除，或匹配提供的正则表达式（REGEXP）的字符串。在本例中，我们正在查找字符串`#define SAMPLES`的添加/删除。例如：

```
+#define SAMPLES   100000
```

或

```
-#define SAMPLES   100000
```

```
git log -G"#define SAMPLES"
```

搜索包含特定字符串的行的更改，或匹配提供的正则表达式（REGEXP）的字符串。例如：

```
-#define SAMPLES   100000
+#define SAMPLES   100000000
```

第2.6节：按作者姓名列出所有贡献

`git shortlog` 汇总 `git log` 并按作者分组

如果未提供参数，将按时间顺序显示每个提交者的所有提交列表。

```
$ git shortlog
提交者 1 (<提交次数>):
提交信息 1
    提交信息 2
```


will show all of your commits with only the first part of the hash and the commit message. Each commit will be in a single line, as the `oneline` flag suggests.

> The oneline option prints each commit on a single line, which is useful if you're looking at a lot of commits. - **source**

Example (from **Free Code Camp** repository, with the same section of code from the other example):

```
87ef97f Merge pull request #7724 from BKinahan/fix/where-art-thou
eb8b729 Fix 'its' typo in Where Art Thou description
e50ff0d Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
2652d04 Remove unnecessary comma from CONTRIBUTING.md
6b01875 Merge pull request #7667 from zerkms/patch-1
766f088 Fixed assignment operator terminology
d1e2468 Merge pull request #7690 from BKinahan/fix/unsubscribe-crash
bed9de2 Merge pull request #7657 from Rafase282/fix/
```

If you wish to limit you command to last n commits log you can simply pass a parameter. For example, if you wish to list last 2 commits logs

```
git log -2 --oneline
```

Section 2.5: Log search

```
git log -S"#define SAMPLES"
```

Searches for **addition** or **removal** of specific string or the string **matching** provided REGEXP. In this case we're looking for addition/removal of the string `#define SAMPLES`. For example:

```
+#define SAMPLES   100000
```

or

```
-#define SAMPLES   100000
```

```
git log -G"#define SAMPLES"
```

Searches for **changes** in **lines containing** specific string or the string **matching** provided REGEXP. For example:

```
-#define SAMPLES   100000
+#define SAMPLES   100000000
```

Section 2.6: List all contributions grouped by author name

`git shortlog` summarizes `git log` and groups by author

If no parameters are given, a list of all commits made per committer will be shown in chronological order.

```
$ git shortlog
Committer 1 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
```

```
    ...
提交者 2 (<提交次数>):
提交信息 1
    提交信息 2
    ...
```

若只想查看提交次数并隐藏提交描述，请使用摘要选项：

`-s`

`--summary`

```
$ git shortlog -s
<number_of_commits> 提交者 1
<number_of_commits> 提交者 2
```

要按提交次数而非提交者姓名的字母顺序排序输出，请传入编号选项：

`-n`

`--numbered`

要添加提交者的邮箱，请添加邮箱选项：

`-e`

`--email`

如果想显示除提交主题以外的信息，也可以提供自定义格式选项：

`--format`

这可以是任何被--format选项的git log接受的字符串。

有关此的更多信息，请参见上文的Colorizing Logs。

第2.7节：在git日志中搜索提交字符串

使用日志中的某个字符串搜索git日志：

```
git log [选项] --grep "搜索字符串"
```

示例：

```
git log --all --grep "removed file"
```

将在所有分支的所有日志中搜索removed file字符串。

从git 2.4+版本开始，可以使用--invert-grep选项反转搜索。

示例：

```
git log --grep="add file" --invert-grep
```

将显示所有不包含add file的提交。


```
    ...
Committer 2 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
```

To simply see the number of commits and suppress the commit description, pass in the summary option:

`-s`

`--summary`

```
$ git shortlog -s
<number_of_commits> Committer 1
<number_of_commits> Committer 2
```

To sort the output by number of commits instead of alphabetically by committer name, pass in the numbered option:

`-n`

`--numbered`

To add the email of a committer, add the email option:

`-e`

`--email`

A custom format option can also be provided if you want to display information other than the commit subject:

`--format`

This can be any string accepted by the `--format` option of `git log`.

See **Colorizing Logs** above for more information on this.

Section 2.7: Searching commit string in git log

Searching git log using some string in log:

```
git log [options] --grep "search_string"
```

Example:

```
git log --all --grep "removed file"
```

Will search for `removed file` string in **all logs** in **all branches**.

Starting from git 2.4+, the search can be inverted using the `--invert-grep` option.

Example:

```
git log --grep="add file" --invert-grep
```

Will show all commits that do not contain `add file`.

第2.8节：文件中某范围行的日志

```
$ git log -L 1,20:index.html
commit 6a57fde739de66293231f6204cbd8b2feca3a869
作者: 约翰·多<john@doe.com>
日期:   2016年3月22日 星期二 16:33:42 -0500

    提交信息

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
 <!DOCTYPE HTML>
 <html>
-       <head>
-        <meta charset="utf-8">

 <head>
+    <meta charset="utf-8">
      <meta http-equiv="X-UA-Compatible" content="IE=edge">
      <meta name="viewport" content="width=device-width, initial-scale=1">
```

第2.9节：过滤日志

```
git log --after '3天前'
```

具体日期也可以：

```
git log --after 2016-05-01
```

与其他接受日期参数的命令和标志一样，允许的日期格式是GNU date支持的格式（高度灵活）。

--after的别名是--since。

也存在相反的标志：--before和--until。

你也可以按作者过滤日志。例如：

```
git log --author=author
```

第2.10节：带内联更改的日志

要查看带内联更改的日志，请使用-p或--patch选项。

```
git log --patch
```

示例（来自Trello Scientist仓库）

```
提交 8ea1452aca481a837d9504f1b2c77ad013367d25
作者：Raymond Chou <info@raychou.io>
日期：2016年3月2日星期三 10:35:25 -0800

    修正readme错误 链接
```


Section 2.8: Log for a range of lines within a file

```
$ git log -L 1,20:index.html
commit 6a57fde739de66293231f6204cbd8b2feca3a869
Author: John Doe <john@doe.com>
Date:   Tue Mar 22 16:33:42 2016 -0500

    commit message

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
 <!DOCTYPE HTML>
 <html>
-       <head>
-        <meta charset="utf-8">
+
++<head>
+    <meta charset="utf-8">
      <meta http-equiv="X-UA-Compatible" content="IE=edge">
      <meta name="viewport" content="width=device-width, initial-scale=1">
```

Section 2.9: Filter logs

```
git log --after '3 days ago'
```

Specific dates work too:

```
git log --after 2016-05-01
```

As with other commands and flags that accept a date parameter, the allowed date format is as supported by GNU date (highly flexible).

An alias to `--after` is `--since`.

Flags exist for the converse too: `--before` and `--until`.

You can also filter logs by `author`. e.g.

```
git log --author=author
```

Section 2.10: Log with changes inline

To see the log with changes inline, use the `-p` or `--patch` options.

```
git log --patch
```

Example (from Trello Scientist repository)

```
ommit 8ea1452aca481a837d9504f1b2c77ad013367d25
Author: Raymond Chou <info@raychou.io>
Date:   Wed Mar 2 10:35:25 2016 -0800

    fix readme error link
```

```
diff --git a/README.md b/README.md
索引 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ 控制 函数 抛出异常，但 *在*测试其他函数并
准备
日志记录。匹配错误的标准基于构造函数和
消息。

-您可以在[examples/errors.js](examples/error.js)找到这个完整的示例。
+你可以在[examples/errors.js](examples/errors.js)找到这个完整示例。

 ## 异步行为


提交 d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
：
```

第2.11节：显示已提交文件的日志

```
git log --stat
```

示例：

```
提交 4ded994d7fc501451fa6e233361887a2365b91d1
作者：Manassés Souza <manasses.inatel@gmail.com>
日期：2016年6月6日 星期一 21:32:30 -0300

MercadoLibre java-sdk 依赖

 mltracking-poc/.gitignore |  1 +
 mltracking-poc/pom.xml    | 14 ++++++++++++--
 2文件已更改，13次插入(+)，2次删除(-)

提交 506fff56190f75bc051248770fb0bcd976e3f9a5
作者：Manassés Souza <manasses.inatel@gmail.com>
日期：2016年6月4日 星期六 12:35:16 -0300

    [manasses] 由 SpringBoot initializr 生成

.gitignore                                                    |  42
++++++++++++
mltracking-poc/mvnw                                           | 233
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
mltracking-poc/mvnw.cmd                                       | 145
++++++++++++++++++++++++++++++++++++++++
mltracking-poc/pom.xml                                        |  74
++++++++++++++++++++
mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java      |  12 ++++
 mltracking-poc/src/main/resources/application.properties                              |   0
 mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java |  18 +++++
 7文件已更改，524次插入(+)
```

第2.12节：显示单个提交的内容

使用git show可以查看单个提交

```
git show 48c83b3
```



```
diff --git a/README.md b/README.md
index 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ the control function threw, but *after* testing the other functions and
readying
 the logging. The criteria for matching errors is based on the constructor and
 message.

-You can find this full example at [examples/errors.js](examples/error.js).
+You can find this full example at [examples/errors.js](examples/errors.js).

 ## Asynchronous behaviors


commit d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
:
```

Section 2.11: Log showing commited files

```
git log --stat
```

Example:

```
commit 4ded994d7fc501451fa6e233361887a2365b91d1
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Mon Jun 6 21:32:30 2016 -0300

    MercadoLibre java-sdk dependency

 mltracking-poc/.gitignore |  1 +
 mltracking-poc/pom.xml    | 14 ++++++++++++--
 2 files changed, 13 insertions(+), 2 deletions(-)

commit 506fff56190f75bc051248770fb0bcd976e3f9a5
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Sat Jun 4 12:35:16 2016 -0300

    [manasses] generated by SpringBoot initializr

 .gitignore                                                    |  42
++++++++++++
 mltracking-poc/mvnw                                           | 233
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 mltracking-poc/mvnw.cmd                                       | 145
++++++++++++++++++++++++++++++++++++++++
 mltracking-poc/pom.xml                                        |  74
++++++++++++++++++++
 mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java      |  12 ++++
 mltracking-poc/src/main/resources/application.properties                              |   0
 mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java |  18 +++++
 7 files changed, 524 insertions(+)
```

Section 2.12: Show the contents of a single commit

Using `git show` we can view a single commit

```
git show 48c83b3
```

```
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

示例

```
提交 48c83b3690dfc7b0e622fd220f8f37c26a77c934
作者：Matt Clark <mrclark32493@gmail.com>
日期：2016年5月4日 星期三 18:26:40 -0400

提交信息将在此显示。

diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
索引 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {

colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
-                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#55FF55" ));
+                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#33CC33" ));
                        colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

第2.13节：两个分支之间的Git日志

git log master..foo 将显示在 foo 上但不在 master 上的提交。对于查看自分支以来你添加了哪些提交非常有用！

第2.14节：显示提交者姓名和提交时间的单行命令

```
tree = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
%C(yellow) %d %Creset %s"'  --all --graph
```

示例

```
*    40554ac  3个月前  亚历山大·佐尔托夫     合并拉取请求 #95 来自
gmandnepr/external_plugins
|\
| *  e509f61  3个月前  伊夫根·德格季连科     记录新属性
| *  46d4cb6  3个月前  伊夫根·德格季连科     使用外部插件运行想法
| *  6253da4  3个月前  伊夫根·德格季连科     解析外部插件类
| *  9fdb4e7  3个月前  伊夫根·德格季连科     保持原始工件名称因为这对Intellij可能很重要

| *  22e82e4  3个月前  伊夫根·德格季连科     在Intellij部分声明外部插件
|/
*  bc3d2cb  3个月前  亚历山大·佐尔托夫     忽略插件.xml中的DTD
```



```
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Example

```
commit 48c83b3690dfc7b0e622fd220f8f37c26a77c934
Author: Matt Clark <mrclark32493@gmail.com>
Date:   Wed May 4 18:26:40 2016 -0400

    The commit message will be shown here.

diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
index 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {

                        colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
-                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#55FF55" ));
+                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#33CC33" ));
                        colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

Section 2.13: Git Log Between Two Branches

`git log master..foo` will show the commits that are on `foo` and not on `master`. Helpful for seeing what commits you've added since branching!

Section 2.14: One line showing commiter name and time since commit

```
tree = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
%C(yellow) %d %Creset %s"'  --all --graph
```

example

```
*    40554ac  3 months ago  Alexander Zolotov    Merge pull request #95 from
gmandnepr/external_plugins
|\
| *  e509f61  3 months ago  Ievgen Degtiarenko    Documenting new property
| *  46d4cb6  3 months ago  Ievgen Degtiarenko    Running idea with external plugins
| *  6253da4  3 months ago  Ievgen Degtiarenko    Resolve external plugin classes
| *  9fdb4e7  3 months ago  Ievgen Degtiarenko    Keep original artifact name as this may be
important for intellij
| *  22e82e4  3 months ago  Ievgen Degtiarenko    Declaring external plugin in intellij section
|/
*  bc3d2cb  3 months ago  Alexander Zolotov    Ignore DTD in plugin.xml
```

第3章：远程操作

第3.1节：删除远程分支

在 Git 中删除远程分支的方法：

```
git push [远程名] --delete [分支名]
```

或

```
git push [远程名] :[分支名]
```

第3.2节：更改Git远程URL

检查现有远程

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

更改仓库URL

```
git remote set-url origin https://github.com/username/repo2.git
# 更改 'origin' 远程的URL
```

验证新的远程URL

```
git remote -v
# origin  https://github.com/username/repo2.git (fetch)
# origin  https://github.com/username/repo2.git (push)
```

第3.3节：列出现有的远程仓库

列出与此仓库关联的所有现有远程仓库：

```
git remote
```

详细列出与此仓库关联的所有现有远程仓库，包括fetch和push的URL：

```
git remote --verbose
```

或者简单地

```
git remote -v
```

第3.4节：删除远程分支的本地副本

如果远程分支已被删除，必须通知本地仓库修剪对该分支的引用。

从特定远程修剪已删除的分支：

Chapter 3: Working with Remotes

Section 3.1: Deleting a Remote Branch

To delete a remote branch in Git:

```
git push [remote-name] --delete [branch-name]
```

or

```
git push [remote-name] :[branch-name]
```

Section 3.2: Changing Git Remote URL

Check existing remote

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Changing repository URL

```
git remote set-url origin https://github.com/username/repo2.git
# Change the 'origin' remote's URL
```

Verify new remote URL

```
git remote -v
# origin  https://github.com/username/repo2.git (fetch)
# origin  https://github.com/username/repo2.git (push)
```

Section 3.3: List Existing Remotes

List all the existing remotes associated with this repository:

```
git remote
```

List all the existing remotes associated with this repository in detail including the `fetch` and `push` URLs:

```
git remote --verbose
```

or simply

```
git remote -v
```

Section 3.4: Removing Local Copies of Deleted Remote Branches

If a remote branch has been deleted, your local repository has to be told to prune the reference to it.

To prune deleted branches from a specific remote:

```
git fetch [remote-name] --prune
```

从所有远程仓库中修剪已删除的分支：

```
git fetch --all --prune
```

第3.5节：从上游仓库更新

假设你已经设置了上游（如"设置上游仓库"中所述）

```
git fetch 远程名
git merge 远程名/分支名
```

pull 命令结合了 fetch 和 merge。

```
git pull
```

带有 --rebase 标志的 pull 命令结合了 fetch 和 rebase，而不是 merge。

```
git pull --rebase 远程名 分支名
```

第3.6节：ls-remote

git ls-remote 是一个独特的命令，允许你查询远程仓库而无需先克隆/拉取它。

它将列出所述远程仓库的 refs/heads 和 refs/tags。

你有时会看到 refs/tags/v0.1.6 和 refs/tags/v0.1.6^{}：这里的 ^{} 用于列出被解引用的带注释标签（即标签指向的提交）

自 git 2.8（2016 年 3 月）起，你可以避免标签的重复条目，直接列出这些被解引用的标签，命令为：

```
git ls-remote --ref
```

当你有 "url.<base>.insteadOf" 配置时，它还能帮助解析远程仓库实际使用的 URL。
如果 `git remote --get-url <aremotename>` 返回 https://server.com/user/repo，且你设置了 `git config url.ssh://git@server.com:.insteadOf https://server.com/`：

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

第 3.7 节：添加新的远程仓库

```
git remote add upstream git-repository-url
```

将由 git-repository-url 表示的远程 git 仓库作为名为 upstream 的新远程添加到 git 仓库中

第3.8节：在新分支上设置上游

您可以使用以下命令创建一个新分支并切换到该分支

```
git checkout -b AP-57
```



```
git fetch [remote-name] --prune
```

To prune deleted branches from *all* remotes:

```
git fetch --all --prune
```

Section 3.5: Updating from Upstream Repository

Assuming you set the upstream (as in the "setting an upstream repository")

```
git fetch remote-name
git merge remote-name/branch-name
```

The `pull` command combines a `fetch` and a `merge`.

```
git pull
```

The `pull` with `--rebase` flag command combines a `fetch` and a `rebase` instead of `merge`.

```
git pull --rebase remote-name branch-name
```

Section 3.6: ls-remote

`git ls-remote` is one unique command allowing you to query a remote repo *without having to clone/fetch it first*.

It will list refs/heads and refs/tags of said remote repo.

You will see sometimes `refs/tags/v0.1.6` *and* `refs/tags/v0.1.6^{}`: the `^{}` to list the dereferenced annotated tag (ie the commit that tag is pointing to)

Since git 2.8 (March 2016), you can avoid that double entry for a tag, and list directly those dereferenced tags with:

```
git ls-remote --ref
```

It can also help resolve the actual url used by a remote repo when you have "`url.<base>.insteadOf`" config setting.
If `git remote --get-url <aremotename>` returns https://server.com/user/repo, and you have set `git config url.ssh://git@server.com:.insteadOf https://server.com/`:

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

Section 3.7: Adding a New Remote Repository

```
git remote add upstream git-repository-url
```

Adds remote git repository represented by `git-repository-url` as new remote named `upstream` to the git repository

Section 3.8: Set Upstream on a New Branch

You can create a new branch and switch to it using

```
git checkout -b AP-57
```

使用 git checkout 创建新分支后，您需要使用以下命令设置上游远程以便推送

```
git push --set-upstream origin AP-57
```

之后，您可以在该分支上使用 git push 命令。

第3.9节：入门

推送到远程分支的语法

`git push <remote_name> <branch_name>`

示例

`git push origin master`

第3.10节：重命名远程

要重命名远程仓库，使用命令 `git remote rename`

命令 `git remote rename` 需要两个参数：

- 一个已存在的远程名称，例如：**origin**
- 远程的新名称，例如：**destination**

获取已存在的远程名称

```
git remote
# origin
```

检查带有 URL 的已存在远程

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

重命名远程

```
git remote rename origin destination
# 将远程名称从 'origin' 改为 'destination'
```

验证新名称

```
git remote -v
# destination https://github.com/username/repo.git (fetch)
# destination https://github.com/usernam/repo.git (push)
```

=== 可能的错误 ===

1.无法将配置节 'remote.[旧名称]' 重命名为 'remote.[新名称]'

此错误表示您尝试使用的旧远程名称（origin）不存在。

After you use git checkout to create a new branch, you will need to set that upstream origin to push to using

```
git push --set-upstream origin AP-57
```

After that, you can use git push while you are on that branch.

Section 3.9: Getting Started

Syntax for pushing to a remote branch

`git push <remote_name> <branch_name>`

Example

`git push origin master`

Section 3.10: Renaming a Remote

To rename remote, use command `git remote rename`

The `git remote rename` command takes two arguments:

- An existing remote name, for example : **origin**
- A new name for the remote, for example : **destination**

Get existing remote name

```
git remote
# origin
```

Check existing remote with URL

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Rename remote

```
git remote rename origin destination
# Change remote name from 'origin' to 'destination'
```

Verify new name

```
git remote -v
# destination https://github.com/username/repo.git (fetch)
# destination https://github.com/usernam/repo.git (push)
```

=== Posible Errors ===

1. Could not rename config section 'remote.[old name]' to 'remote.[new name]'

   This error means that the remote you tried the old remote name (**origin**) doesn't exist.

2. 远程仓库【新名称】已存在。

   错误信息不言自明。

第3.11节：显示特定远程仓库的信息

输出已知远程仓库的一些信息：origin

```
git remote show origin
```

仅打印远程仓库的URL：

```
git config --get remote.origin.url
```

在2.7及以上版本中，也可以使用以下命令，这比使用config命令的上述方法更好。

```
git remote get-url origin
```

第3.12节：为特定远程设置URL

您可以通过以下命令更改现有远程的URL

```
git remote set-url 远程名称 URL
```

第3.13节：获取特定远程的URL

您可以使用以下命令获取现有远程的URL

```
git remote get-url <名称>
```

默认情况下，这将是

```
git remote get-url origin
```

第3.14节：更改远程仓库

要更改远程指向的仓库URL，您可以使用set-url选项，如下所示：

```
git remote set-url <远程名称> <远程仓库URL>
```

示例：

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```


2. Remote [new name] already exists.

   Error message is self explanatory.

Section 3.11: Show information about a Specific Remote

Output some information about a known remote: `origin`

```
git remote show origin
```

Print just the remote's URL:

```
git config --get remote.origin.url
```

With 2.7+, it is also possible to do, which is arguably better than the above one that uses the `config` command.

```
git remote get-url origin
```

Section 3.12: Set the URL for a Specific Remote

You can change the url of an existing remote by the command

```
git remote set-url remote-name url
```

Section 3.13: Get the URL for a Specific Remote

You can obtain the url for an existing remote by using the command

```
git remote get-url <name>
```

By default, this will be

```
git remote get-url origin
```

Section 3.14: Changing a Remote Repository

To change the URL of the repository you want your remote to point to, you can use the `set-url` option, like so:

```
git remote set-url <remote_name> <remote_repository_url>
```

Example:

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```

第4章：暂存

第4.1节：暂存所有文件更改

```
git add -A
```

版本 ≥ 2.0

```
git add .
```

在2.x版本中，`git add .` 会暂存当前目录及其所有子目录中对文件的所有更改。然而，在1.x版本中，它只会暂存新建和修改的文件，不包括已删除的文件。

使用git add -A，或其等效命令git add --all，可以在任何版本的git中暂存所有文件更改。

第4.2节：取消暂存包含更改的文件

```
git reset <filePath>
```

第4.3节：按块添加更改

你可以使用补丁标志查看将被暂存以供提交的“块”工作内容：

```
git add -p
```

或

```
git add --patch
```

这将打开一个交互式提示，允许你查看差异，并决定是否要包含它们。

```
暂存此块 [y,n,q,a,d,/,s,e,?]?
```

- y 将此块暂存到下一次提交
- n 不将此块暂存到下一次提交
- q 退出；不暂存此块及剩余的块
- a 暂存此块及文件中所有后续块
- d 不暂存此块及文件中所有后续块
- g 选择要跳转的块
- / 搜索匹配给定正则表达式的块
- j 保持此块未决定，查看下一个未决定的块
- J 保留此块未决定，查看下一块
- k 保留此块未决定，查看之前未决定的块
- K 保留此块未决定，查看之前的块
- s 将当前块拆分成更小的块
- e 手动编辑当前块
- ? 打印块帮助

这使得捕捉你不想提交的更改变得容易。

你也可以通过 git add --interactive 并选择 p 来打开此功能。

Chapter 4: Staging

Section 4.1: Staging All Changes to Files

```
git add -A
```

Version ≥ 2.0

```
git add .
```

In version 2.x, `git add .` will stage all changes to files in the current directory and all its subdirectories. However, in 1.x it will only stage new and modified files, not deleted files.

Use `git add -A`, or its equivalent command `git add --all`, to stage all changes to files in any version of git.

Section 4.2: Unstage a file that contains changes

```
git reset <filePath>
```

Section 4.3: Add changes by hunk

You can see what "hunks" of work would be staged for commit using the patch flag:

```
git add -p
```

or

```
git add --patch
```

This opens an interactive prompt that allows you to look at the diffs and let you decide whether you want to include them or not.

```
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

- y stage this hunk for the next commit
- n do not stage this hunk for the next commit
- q quit; do not stage this hunk or any of the remaining hunks
- a stage this hunk and all later hunks in the file
- d do not stage this hunk or any of the later hunks in the file
- g select a hunk to go to
- / search for a hunk matching the given regex
- j leave this hunk undecided, see next undecided hunk
- J leave this hunk undecided, see next hunk
- k leave this hunk undecided, see previous undecided hunk
- K leave this hunk undecided, see previous hunk
- s split the current hunk into smaller hunks
- e manually edit the current hunk
- ? print hunk help

This makes it easy to catch changes which you do not want to commit.

You can also open this via `git add --interactive` and selecting p.

第4.4节：交互式添加

git add -i（或 --interactive）将为你提供一个交互式界面，你可以编辑索引，准备你想要包含在下次提交中的内容。你可以添加和移除整个文件的更改，添加未跟踪的文件以及移除被跟踪的文件，还可以通过选择要添加的更改块、拆分这些更改块，甚至编辑差异，来选择部分更改放入索引。许多Git的图形化提交工具（例如

git gui）包含此功能；这可能比命令行版本更易于使用。

这非常有用，(1) 如果你在工作目录中有纠缠的更改，想要分开提交，而不是全部放在一个提交中，(2) 如果你正在进行交互式变基并想拆分过大的提交。

```
$ git add -i
已暂存       未暂存 路径
  1:     未更改          +4/-4 index.js
  2:         +1/-0        无 package.json

*** 命令 ***
  1：状态       2：更新       3：还原       4：添加未跟踪
  5：补丁       6：差异       7: 退出       8: 帮助
现在怎么办>
```

该输出的上半部分显示了索引的当前状态，分为暂存区和未暂存区两列：

1. index.js 已添加4行并删除4行。当前未暂存，状态报告显示“unchanged。” 当此文件变为暂存状态时，+4/-4 位将被转移到暂存列，未暂存列将显示“nothing”。

2. package.json 已添加一行并已暂存。自那以后没有进一步的更改
   如未暂存栏下的“无”线所示，已分阶段。

下半部分显示了您可以执行的操作。您可以输入一个数字（1-8）或一个字母（s、u、r、a、p、d、q、h）。

status 显示与上述输出顶部部分相同的输出。

update 允许你使用额外的语法对暂存的提交进行进一步修改。

revert 将暂存的提交信息还原到HEAD。

add untracked 允许你添加之前未被版本控制跟踪的文件路径。

patch 允许从类似于status的输出中选择一个路径进行进一步分析。

diff 显示将要提交的内容。

quit 退出命令。

help 提供关于使用此命令的更多帮助。

第4.5节：显示暂存的更改

显示已暂存以供提交的代码块：

```
git diff --cached
```


Section 4.4: Interactive add

`git add -i` (or `--interactive`) will give you an interactive interface where you can edit the index, to prepare what you want to have in the next commit. You can add and remove changes to whole files, add untracked files and remove files from being tracked, but also select subsection of changes to put in the index, by selecting chunks of changes to be added, splitting those chunks, or even editing the diff. Many graphical commit tools for Git (like e.g. `git gui`) include such feature; this might be easier to use than the command line version.

It is very useful (1) if you have entangled changes in the working directory that you want to put in separate commits, and not all in one single commit (2) if you are in the middle of an interactive rebase and want to split too large commit.

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +4/-4 index.js
  2:        +1/-0      nothing package.json

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

The top half of this output shows the current state of the index broken up into staged and unstaged columns:

1. `index.js` has had 4 lines added and 4 lines removed. It is currently not staged, as the current status reports "unchanged." When this file becomes staged, the `+4/-4` bit will be transferred to the staged column and the unstaged column will read "nothing."
2. `package.json` has had one line added and has been staged. There are no further changes since it has been staged as indicated by the "nothing" line under the unstaged column.

The bottom half shows what you can do. Either enter a number (1-8) or a letter (`s`, `u`, `r`, `a`, `p`, `d`, `q`, `h`).

`status` shows output identical to the top part of the output above.

`update` allows you to make further changes to the staged commits with additional syntax.

`revert` will revert the staged commit information back to HEAD.

`add untracked` allows you to add filepaths previously untracked by version control.

`patch` allows for one path to be selected out of an output similar to `status` for further analysis.

`diff` displays what will be committed.

`quit` exits the command.

`help` presents further help on using this command.

Section 4.5: Show Staged Changes

To display the hunks that are staged for commit:

```
git diff --cached
```

第4.6节：暂存单个文件

要将文件暂存以便提交，请运行

```
git add <filename>
```

第4.7节：暂存已删除的文件

```
git rm filename
```

要从git中删除文件但不从磁盘中移除，请使用--cached标志

```
git rm --cached filename
```


Section 4.6: Staging A Single File

To stage a file for committing, run

```
git add <filename>
```

Section 4.7: Stage deleted files

```
git rm filename
```

To delete the file from git without removing it from disk, use the `--cached` flag

```
git rm --cached filename
```

第5章：忽略文件和文件夹

本主题说明如何避免在Git仓库中添加不需要的文件（或文件更改）。有几种方法（全局或本地`.gitignore`，`.git/exclude`，`git update-index --assume-unchanged`，以及`git update-index --skip-tree`），但请记住Git管理的是*内容*，这意味着：忽略实际上是忽略文件夹的*内容*（即文件）。
空文件夹默认会被忽略，因为它本身无法被添加。

第5.1节：使用.gitignore文件忽略文件和目录

您可以通过在您的仓库中创建一个或多个.gitignore文件，使Git忽略某些文件和目录—也就是说，将它们排除在Git的跟踪之外。

在软件项目中，.gitignore 通常包含在构建过程中或运行时生成的文件和/或目录的列表。.gitignore 文件中的条目可能包括指向以下内容的名称或路径：

1. 临时资源，例如缓存、日志文件、已编译代码等。
2. 不应与其他开发者共享的本地配置文件
3. 包含秘密信息的文件，如登录密码、密钥和凭证

当在顶层目录创建时，规则将递归应用于整个仓库中的所有文件和子目录。当在子目录中创建时，规则将应用于该特定目录及其子目录。

当文件或目录被忽略时，它将不会被：

1. 被 Git 跟踪
2. 通过诸如git status或git diff等命令报告
3. 通过诸如git add -A等命令暂存

在需要忽略已跟踪文件的特殊情况下，应特别小心。详见：忽略已提交到 Git 仓库的文件。

示例

以下是基于.gitignore文件中glob文件模式的一些通用规则示例：

```
# 以`#`开头的行是注释。

# 忽略名为'file.ext'的文件
file.ext

# 注释不能与规则写在同一行！
# 以下行忽略名为'file.ext # not a comment'的文件
file.ext # not a comment

# 忽略带有完整路径的文件。
# 这会匹配根目录及子目录中的文件。
# 即其他文件otherfile.ext将在整个目录树中被忽略。
dir/otherdir/file.ext
otherfile.ext

# 忽略目录
# 目录本身及其内容都会被忽略。
bin/
gen/
```


Chapter 5: Ignoring Files and Folders

This topic illustrates how to avoid adding unwanted files (or file changes) in a Git repo. There are several ways (global or local `.gitignore`, `.git/exclude`, `git update-index --assume-unchanged`, and `git update-index --skip-tree`), but keep in mind Git is managing *content*, which means: ignoring actually ignores a folder *content* (i.e. files). An empty folder would be ignored by default, since it cannot be added anyway.

Section 5.1: Ignoring files and directories with a .gitignore file

You can make Git ignore certain files and directories — that is, exclude them from being tracked by Git — by creating one or more `.gitignore` files in your repository.

In software projects, `.gitignore` typically contains a listing of files and/or directories that are generated during the build process or at runtime. Entries in the `.gitignore` file may include names or paths pointing to:

1. temporary resources e.g. caches, log files, compiled code, etc.
2. local configuration files that should not be shared with other developers
3. files containing secret information, such as login passwords, keys and credentials

When created in the top level directory, the rules will apply recursively to all files and sub-directories throughout the entire repository. When created in a sub-directory, the rules will apply to that specific directory and its sub-directories.

When a file or directory is ignored, it will not be:

1. tracked by Git
2. reported by commands such as `git status` or `git diff`
3. staged with commands such as `git add -A`

In the unusual case that you need to ignore tracked files, special care should be taken. See: Ignore files that have already been committed to a Git repository.

Examples

Here are some generic examples of rules in a `.gitignore` file, based on glob file patterns:

```
# Lines starting with `#` are comments.

# Ignore files called 'file.ext'
file.ext

# Comments can't be on the same line as rules!
# The following line ignores files called 'file.ext # not a comment'
file.ext # not a comment

# Ignoring files with full path.
# This matches files in the root directory and subdirectories too.
# i.e. otherfile.ext will be ignored anywhere on the tree.
dir/otherdir/file.ext
otherfile.ext

# Ignoring directories
# Both the directory itself and its contents will be ignored.
bin/
gen/
```

```
# 这里也可以使用通配符模式来忽略包含特定字符的路径。
# 例如，下面的规则将匹配 build/ 和 Build/
[bB]uild/

# 如果没有尾部斜杠，该规则将匹配文件和/或
# 目录，因此以下规则将忽略名为 `gen` 的文件
# 和名为 `gen` 的目录，以及该目录中的所有内容
bin
gen

# 按扩展名忽略文件
# 所有具有这些扩展名的文件将在
# 本目录及其所有子目录中被忽略。
*.apk
*.class

# 可以将两种形式结合起来，忽略某些目录中具有特定扩展名的文件。# 以下规则与上
面定义的通用规则是冗余的。

java/*.apk
gen/*.class

# 若只想忽略顶层目录中的文件，而不忽略其
# 子目录中的文件，请在规则前加上 `/` 前缀
/*.apk
/*.class

# 若要忽略任意深度中名为 DirectoryA 的目录，
# 请在 DirectoryA 前使用 **
# 别忘了最后的 /，
# 否则会忽略所有名为 DirectoryA 的文件，而不是目录
**/目录A/
# 这将忽略
# 目录A/
# 目录B/目录A/
# 目录C/目录B/目录A/
# 它不会忽略任何级别中名为目录A的文件

# 要忽略任何名为目录B且位于名为目录A的目录内的目录，中间可以有任意数量的目录，
# 使用目录之间的 **

目录A/**/目录B/
# 这将忽略
# 目录A/目录B/
# 目录A/目录Q/目录B/
# 目录A/目录Q/目录W/目录B/

# 要忽略一组文件，可以使用通配符，如上所示。
# 单独的 '*' 会忽略文件夹中的所有内容，包括你的 .gitignore 文件。
# 使用通配符时要排除特定文件，请对其取反。
# 这样它们就会被排除在忽略列表之外：
!.gitignore

# 使用反斜杠作为转义字符以忽略带有井号（#）的文件
#（自版本1.6.2.1起支持）
|#*#
```

大多数.gitignore文件在各种语言中是通用的，因此为了快速入门，这里提供了一组按语言分类的示例.gitignore文件，您可以克隆或复制/修改到您的项目中。或者，对于新项目，您也可以考虑使用在线工具自动生成一个起始文件。


```
# Glob pattern can also be used here to ignore paths with certain characters.
# For example, the below rule will match both build/ and Build/
[bB]uild/

# Without the trailing slash, the rule will match a file and/or
# a directory, so the following would ignore both a file named `gen`
# and a directory named `gen`, as well as any contents of that directory
bin
gen

# Ignoring files by extension
# All files with these extensions will be ignored in
# this directory and all its sub-directories.
*.apk
*.class

# It's possible to combine both forms to ignore files with certain
# extensions in certain directories. The following rules would be
# redundant with generic rules defined above.
java/*.apk
gen/*.class

# To ignore files only at the top level directory, but not in its
# subdirectories, prefix the rule with a `/`
/*.apk
/*.class

# To ignore any directories named DirectoryA
# in any depth use ** before DirectoryA
# Do not forget the last /,
# Otherwise it will ignore all files named DirectoryA, rather than directories
**/DirectoryA/
# This would ignore
# DirectoryA/
# DirectoryB/DirectoryA/
# DirectoryC/DirectoryB/DirectoryA/
# It would not ignore a file named DirectoryA, at any level

# To ignore any directory named DirectoryB within a
# directory named DirectoryA with any number of
# directories in between, use ** between the directories
DirectoryA/**/DirectoryB/
# This would ignore
# DirectoryA/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/

# To ignore a set of files, wildcards can be used, as can be seen above.
# A sole '*' will ignore everything in your folder, including your .gitignore file.
# To exclude specific files when using wildcards, negate them.
# So they are excluded from the ignore list:
!.gitignore

# Use the backslash as escape character to ignore files with a hash (#)
# (supported since 1.6.2.1)
\#*#
```

Most `.gitignore` files are standard across various languages, so to get started, here is set of sample `.gitignore` files listed by language from which to clone or copy/modify into your project. Alternatively, for a fresh project you may consider auto-generating a starter file using an online tool.

其他形式的.gitignore

.gitignore文件旨在作为仓库的一部分提交。如果您想忽略某些文件但不提交忽略规则，可以考虑以下选项：

- 编辑.git/info/exclude文件（使用与.gitignore相同的语法）。这些规则将在仓库范围内全局生效；
- 设置一个全局gitignore文件，将忽略规则应用于您所有的本地仓库：

此外，您可以在不更改全局git配置的情况下，忽略对已跟踪文件的本地更改，方法是：

- `git update-index --skip-worktree [<file>...]`：用于轻微的本地修改
- `git update-index --assume-unchanged [<file>...]`：用于生产就绪的、不变更的上游文件

有关后者标志之间的更多差异详情，请参见git update-index文档中的进一步选项。

清理被忽略的文件

你可以使用git clean -X来清理被忽略的文件：

```
git clean -Xn #显示被忽略文件的列表
git clean -Xf #删除之前显示的文件
```

注意：-X（大写）仅清理被忽略的文件。使用-x（小写）也会删除未跟踪的文件。

更多详情请参见git clean文档。

更多详情请参见Git手册。

第5.2节：检查文件是否被忽略

git check-ignore命令报告Git忽略的文件。

你可以在命令行传入文件名，git check-ignore会列出被忽略的文件名。例如：

```
$ cat .gitignore
*.o
$ git check-ignore example.o Readme.md
example.o
```

这里，只有 *.o 文件在 .gitignore 中定义，所以 Readme.md 不会出现在 git check-ignore 的输出中。

如果你想查看是 .gitignore 的哪一行导致文件被忽略，可以在 git check-ignore 命令中添加 -v 参数：

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o        example.o
```

从 Git 1.7.6 版本开始，你也可以使用 git status --ignored 来查看被忽略的文件。你可以在 official documentation 或“Finding files ignored by .gitignore”中找到更多信息。

Other forms of .gitignore

`.gitignore` files are intended to be committed as part of the repository. If you want to ignore certain files without committing the ignore rules, here are some options:

- Edit the `.git/info/exclude` file (using the same syntax as `.gitignore`). The rules will be global in the scope of the repository;
- Set up a global gitignore file that will apply ignore rules to all your local repositories:

Furthermore, you can ignore local changes to tracked files without changing the global git configuration with:

- `git update-index --skip-worktree [<file>...]`: for minor local modifications
- `git update-index --assume-unchanged [<file>...]`: for production ready, non-changing files upstream

See more details on differences between the latter flags and the `git update-index` documentation for further options.

Cleaning up ignored files

You can use `git clean -X` to cleanup ignored files:

```
git clean -Xn #display a list of ignored files
git clean -Xf #remove the previously displayed files
```

Note: `-X` (caps) cleans up *only* ignored files. Use `-x` (no caps) to also remove untracked files.

See the `git clean` documentation for more details.

See the Git manual for more details.

Section 5.2: Checking if a file is ignored

The `git check-ignore` command reports on files ignored by Git.

You can pass filenames on the command line, and `git check-ignore` will list the filenames that are ignored. For example:

```
$ cat .gitignore
*.o
$ git check-ignore example.o Readme.md
example.o
```

Here, only *.o files are defined in .gitignore, so Readme.md is not listed in the output of `git check-ignore`.

If you want to see line of which .gitignore is responsible for ignoring a file, add -v to the git check-ignore command:

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o        example.o
```

From Git 1.7.6 onwards you can also use `git status --ignored` in order to see ignored files. You can find more info on this in the official documentation or in Finding files ignored by .gitignore.

第 5.3 节：.gitignore 文件中的例外情况

如果你使用模式忽略文件但有例外情况，可以在例外前加上感叹号(!)。例如：

```
*.txt
!important.txt
```

上述示例指示 Git 忽略所有扩展名为 .txt 的文件，但不忽略名为 important.txt 的文件。

如果文件位于被忽略的文件夹中，你不能那么容易地重新包含它：

```
folder/
!folder/*.txt
```

在此示例中，文件夹中所有的 .txt 文件将继续被忽略。

正确的方法是在单独一行重新包含文件夹本身，然后通过 * 忽略 folder 中的所有文件，最后重新包含 folder 中的 *.txt，如下所示：

```
!folder/
folder/*
!folder/*.txt
```

注意：对于以感叹号开头的文件名，添加两个感叹号或使用 \ 字符进行转义：

```
!!includethis
\!excludethis
```

第5.4节：全局 .gitignore 文件

为了让 Git 在所有仓库中忽略某些文件，你可以在终端或命令提示符中使用以下命令创建一个全局 .gitignore：

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git 现在会在每个仓库自己的 .gitignore 文件之外，额外使用这个文件。其规则如下：

- 如果本地.gitignore文件明确包含某个文件，而全局.gitignore忽略该文件，则本地.gitignore优先（该文件将被包含）
- 如果仓库被克隆到多台机器上，那么全局.gigignore必须在所有机器上加载或者至少包含它，因为被忽略的文件会被推送到仓库，而拥有全局.gitignore的电脑不会更新它。这就是为什么如果项目由团队协作，使用仓库特定的.gitignore比全局的更好

该文件是存放平台、机器或用户特定忽略项的好地方，例如 OSX 的.DS_Store，Windows 的Thumbs.db或者 Vim 的*.ext~和*.ext.swp忽略项，如果你不想将这些文件保存在仓库中。所以一位使用 OS X 的团队成员可以添加所有.DS_STORE和_MACOSX（实际上无用），而另一位使用 Windows 的团队成员可以忽略所有thumbs.bd

第5.5节：忽略已经提交到

Section 5.3: Exceptions in a .gitignore file

If you ignore files by using a pattern but have exceptions, prefix an exclamation mark(!) to the exception. For example:

```
*.txt
!important.txt
```

The above example instructs Git to ignore all files with the `.txt` extension except for files named `important.txt`.

If the file is in an ignored folder, you can **NOT** re-include it so easily:

```
folder/
!folder/*.txt
```

In this example all .txt files in the folder would remain ignored.

The right way is re-include the folder itself on a separate line, then ignore all files in `folder` by `*`, finally re-include the `*.txt` in `folder`, as the following:

```
!folder/
folder/*
!folder/*.txt
```

Note: For file names beginning with an exclamation mark, add two exclamation marks or escape with the `\` character:

```
!!includethis
\!excludethis
```

Section 5.4: A global .gitignore file

To have Git ignore certain files across all repositories you can create a global .gitignore with the following command in your terminal or command prompt:

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git will now use this in addition to each repository's own .gitignore file. Rules for this are:

- If the local `.gitignore` file explicitly includes a file while the global `.gitignore` ignores it, the local `.gitignore` takes priority (the file will be included)
- If the repository is cloned on multiple machines, then the global `.gigignore` must be loaded on all machines or at least include it, as the ignored files will be pushed up to the repo while the PC with the global `.gitignore` wouldn't update it. This is why a repo specific `.gitignore` is a better idea than a global one if the project is worked on by a team

This file is a good place to keep platform, machine or user specific ignores, e.g. OSX `.DS_Store`, Windows `Thumbs.db` or Vim `*.ext~` and `*.ext.swp` ignores if you don't want to keep those in the repository. So one team member working on OS X can add all `.DS_STORE` and `_MACOSX` (which is actually useless), while another team member on Windows can ignore all `thumbs.bd`

Section 5.5: Ignore files that have already been committed to

Git仓库的文件

如果你已经将文件添加到 Git 仓库，现在想要停止跟踪它（使其不会出现在未来的提交中），你可以从索引中移除它：

```
git rm --cached <file>
```

这将从仓库中移除该文件，并防止 Git 继续跟踪其更改。选项`--cached`确保文件不会被物理删除。

请注意，之前添加的文件内容仍然可以通过 Git 历史记录查看。

请记住，如果其他人在您从索引中移除文件后从仓库拉取，**他们的副本将被物理删除**。

你可以让 Git 假装工作目录中的文件版本是最新的，而改为读取索引版本（从而忽略其中的更改），方法是使用 "skip worktree" 位：

```
git update-index --skip-worktree <file>
```

写操作不受此位影响，内容安全仍然是首要任务。你永远不会丢失宝贵的被忽略更改；另一方面，此位与暂存（stashing）冲突：要移除此位，请使用

```
git update-index --no-skip-worktree <file>
```

有时错误地建议对 Git 撒谎，让它假设文件未更改而不检查它。乍一看，这似乎是忽略对文件的任何进一步更改，而不将其从索引中移除：

```
git update-index --assume-unchanged <file>
```

这将强制 git 忽略对文件所做的任何更改（请记住，如果你拉取了该文件的任何更改，或者你暂存了它，**你被忽略的更改将会丢失**）

如果你想让 git 重新“关注”此文件，请运行以下命令：

```
git update-index --no-assume-unchanged <file>
```

第5.6节：在本地忽略文件而不提交忽略规则

`.gitignore` 在本地忽略文件，但它是用于提交到仓库并与其他贡献者和用户共享的。你可以设置一个全局的`.gitignore`，但那样所有仓库都会共享这些设置。

如果你想在仓库中本地忽略某些文件且不将该文件作为任何仓库的一部分，请编辑 .git/info/exclude 仓库内的该文件。

例如：

```
# 这些文件仅在此仓库中被忽略
# 这些规则不会与任何人共享
# 因为它们是个人的
gtk_tests.py
gui/gtk/tests/*
localhost
```


a Git repository

If you have already added a file to your Git repository and now want to **stop tracking it** (so that it won't be present in future commits), you can remove it from the index:

```
git rm --cached <file>
```

This will remove the file from the repository and prevent further changes from being tracked by Git. The `--cached` option will make sure that the file is not physically deleted.

Note that previously added contents of the file will still be visible via the Git history.

Keep in mind that if anyone else pulls from the repository after you removed the file from the index, **their copy will be physically deleted**.

You can make Git pretend that the working directory version of the file is up to date and read the index version instead (thus ignoring changes in it) with "skip worktree" bit:

```
git update-index --skip-worktree <file>
```

Writing is not affected by this bit, content safety is still first priority. You will never lose your precious ignored changes; on the other hand this bit conflicts with stashing: to remove this bit, use

```
git update-index --no-skip-worktree <file>
```

It is sometimes ***wrongly*** recommended to lie to Git and have it assume that file is unchanged without examining it. It looks at first glance as ignoring any further changes to the file, without removing it from its index:

```
git update-index --assume-unchanged <file>
```

This will force git to ignore any change made in the file (keep in mind that if you pull any changes to this file, or you stash it, **your ignored changes will be lost**)

If you want git to "care" about this file again, run the following command:

```
git update-index --no-assume-unchanged <file>
```

Section 5.6: Ignore files locally without committing ignore rules

`.gitignore` ignores files locally, but it is intended to be committed to the repository and shared with other contributors and users. You can set a global `.gitignore`, but then all your repositories would share those settings.

If you want to ignore certain files in a repository locally and not make the file part of any repository, edit `.git/info/exclude` inside your repository.

For example:

```
# these files are only ignored on this repo
# these rules are not shared with anyone
# as they are personal
gtk_tests.py
gui/gtk/tests/*
localhost
```

```
pushReports.py
server/
```

第5.7节：忽略对文件的后续更改（不删除文件）

有时你希望文件被Git管理，但忽略对其后续的更改。

使用update-index告诉Git忽略对文件或目录的更改：

```
git update-index --assume-unchanged my-file.txt
```

上述命令指示 Git 假设`my-file.txt`未被更改，并且不检查或报告更改。该文件仍然存在于仓库中。

这对于提供默认值并允许本地环境覆盖非常有用，例如：

```
# 创建一个包含一些值的文件
cat <<EOF
MYSQL_USER=app
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env

# 提交到 Git
git add .env
git commit -m "添加 .env 模板"

# 忽略对 .env 的未来更改
git update-index --assume-unchanged .env

# 更新你的密码
vi .env

# 没有更改！
git status
```

第5.8节：忽略任意目录中的文件

要忽略任意目录中的文件foo.txt，只需写出它的名称：

```
foo.txt # 匹配任意目录下所有名为 'foo.txt' 的文件
```

如果你只想忽略树的某部分中的文件，可以使用以下方式指定特定目录的子目录 `**` 模式：

```
bar/**/foo.txt # 匹配 'bar' 目录及其所有子目录中的所有 'foo.txt' 文件
```

或者你可以在bar/目录下创建一个**.gitignore**文件。与前面的例子等效的是创建文件`bar/.gitignore`，内容如下：

```
foo.txt # 匹配 bar/ 目录下任意子目录中的所有 'foo.txt' 文件
```

第5.9节：预填充的.gitignore模板

如果您不确定在您的.gitignore文件中应该列出哪些规则，或者您只是想为项目添加通用的例外规则

pushReports.py
server/

Section 5.7: Ignoring subsequent changes to a file (without removing it)

Sometimes you want to have a file held in Git but ignore subsequent changes.

Tell Git to ignore changes to a file or directory using `update-index`:

```
git update-index --assume-unchanged my-file.txt
```

The above command instructs Git to assume `my-file.txt` hasn't been changed, and not to check or report changes. The file is still present in the repository.

This can be useful for providing defaults and allowing local environment overrides, e.g.:

```
# create a file with some values in
cat <<EOF
MYSQL_USER=app
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env

# commit to Git
git add .env
git commit -m "Adding .env template"

# ignore future changes to .env
git update-index --assume-unchanged .env

# update your password
vi .env

# no changes!
git status
```

Section 5.8: Ignoring a file in any directory

To ignore a file `foo.txt` in **any** directory you should just write its name:

```
foo.txt # matches all files 'foo.txt' in any directory
```

If you want to ignore the file only in part of the tree, you can specify the subdirectories of a specific directory with `**` pattern:

```
bar/**/foo.txt # matches all files 'foo.txt' in 'bar' and all subdirectories
```

Or you can create a `.gitignore` file in the `bar/` directory. Equivalent to the previous example would be creating file `bar/.gitignore` with these contents:

```
foo.txt # matches all files 'foo.txt' in any directory under bar/
```

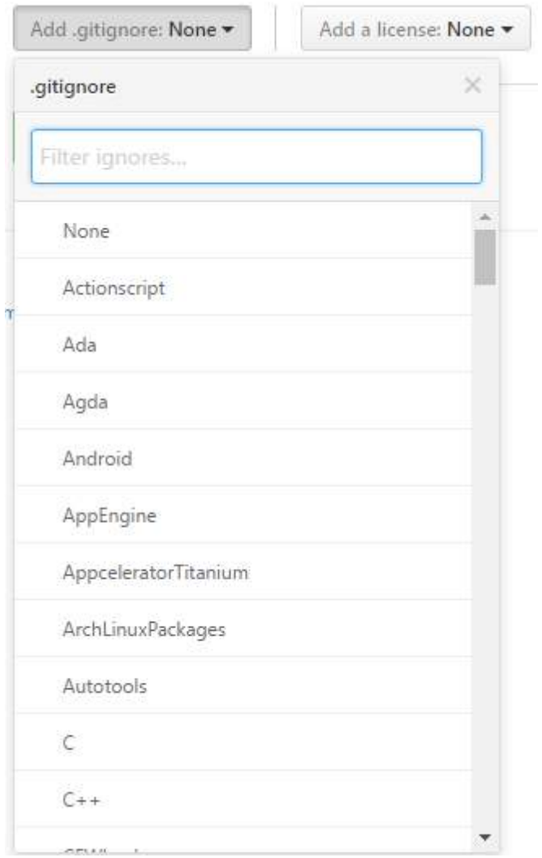
Section 5.9: Prefilled .gitignore Templates

If you are unsure which rules to list in your `.gitignore` file, or you just want to add generally accepted exceptions

您可以选择或生成一个.gitignore文件：

- https://www.gitignore.io/
- https://github.com/github/gitignore

许多托管服务如GitHub和BitBucket提供根据您可能使用的编程语言和IDE生成.gitignore文件的功能：



第5.10节：忽略子文件夹中的文件（多个gitignore文件）

假设您的仓库结构如下：

```
examples/
    output.log
src/
    <未显示的文件>
    output.log
README.md
```

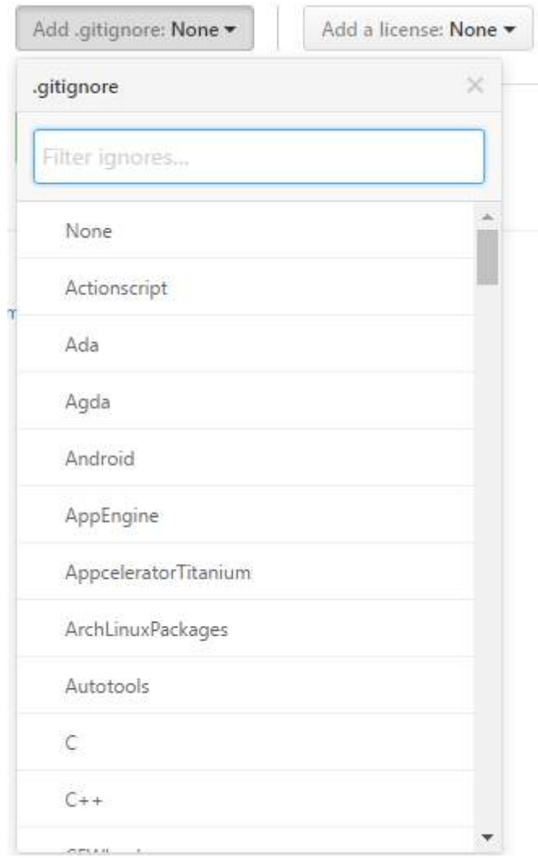
examples目录下的output.log是有效且项目所需的，用于收集理解信息，而位于src/下的output.log是在调试时创建的，不应出现在历史记录或仓库中。

有两种方法可以忽略此文件。您可以在工作目录根目录的.gitignore文件中放置绝对路径：

to your project, you can choose or generate a `.gitignore` file:

- https://www.gitignore.io/
- https://github.com/github/gitignore

Many hosting services such as GitHub and BitBucket offer the ability to generate `.gitignore` files based upon the programming languages and IDEs you may be using:



Section 5.10: Ignoring files in subfolders (Multiple gitignore files)

Suppose you have a repository structure like this:

```
examples/
    output.log
src/
    <files not shown>
    output.log
README.md
```

`output.log` in the examples directory is valid and required for the project to gather an understanding while the one beneath `src/` is created while debugging and should not be in the history or part of the repository.

There are two ways to ignore this file. You can place an absolute path into the `.gitignore` file at the root of the working directory:

```
# /.gitignore
src/output.log
```

或者，您可以在 src/ 目录下创建一个 **.gitignore** 文件，并忽略相对于该 `.gitignore` 的文件：

```
# /src/.gitignore
output.log
```

第5.11节：创建空文件夹

由于Git管理的是文件并附加其目录，这样可以减少提交内容并提高速度，因此无法添加和提交空文件夹。为了解决这个问题，有两种方法：

方法一：.gitkeep

一种解决方法是使用 .gitkeep 文件来注册该文件夹。具体做法是，先创建所需目录，然后在该文件夹中添加一个 .gitkeep 文件。该文件是空的，除了用于注册文件夹外没有其他作用。在Windows系统中（文件命名较为麻烦），只需在该目录下打开git bash，运行以下命令：

> $ touch .gitkeep

此命令仅在当前目录中创建一个空的`.gitkeep`文件

方法二：`dummy.txt`

另一种方法与上述非常相似，可以按照相同步骤操作，但不是使用.gitkeep，而是使用dummy.txt。这样还有一个额外的好处，就是可以通过Windows的右键菜单轻松创建该文件。你还可以在文件中留下有趣的信息。你也可以使用.gitkeep文件来跟踪空目录。.gitkeep通常是一个空文件，用于跟踪空目录。

第5.12节：查找被.gitignore忽略的文件

你可以使用以下命令列出当前目录中所有被git忽略的文件：

```
git status --ignored
```

所以如果我们的仓库结构如下：

```
.git
.gitignore
./example_1
./dir/example_2
./example_2
```

...以及.gitignore文件内容为：

```
example_2
```

...那么命令的结果将是：

```
$ git status --ignored
```



```
# /.gitignore
src/output.log
```

Alternatively, you can create a `.gitignore` file in the `src/` directory and ignore the file that is relative to this `.gitignore`:

```
# /src/.gitignore
output.log
```

Section 5.11: Create an Empty Folder

It is not possible to add and commit an empty folder in Git due to the fact that Git manages *files* and attaches their directory to them, which slims down commits and improves speed. To get around this, there are two methods:

Method one: `.gitkeep`

One hack to get around this is to use a `.gitkeep` file to register the folder for Git. To do this, just create the required directory and add a `.gitkeep` file to the folder. This file is blank and doesn't serve any purpose other than to just register the folder. To do this in Windows (which has awkward file naming conventions) just open git bash in the directory and run the command:

> $ touch .gitkeep

This command just makes a blank `.gitkeep` file in the current directory

Method two: `dummy.txt`

Another hack for this is very similar to the above and the same steps can be followed, but instead of a `.gitkeep`, just use a `dummy.txt` instead. This has the added bonus of being able to easily create it in Windows using the context menu. And you get to leave funny messages in them too.You can also use `.gitkeep` file to track the empty directory. `.gitkeep` normally is an empty file that is added to track the empty directory.

Section 5.12: Finding files ignored by .gitignore

You can list all files ignored by git in current directory with command:

```
git status --ignored
```

So if we have repository structure like this:

```
.git
.gitignore
./example_1
./dir/example_2
./example_2
```

...and .gitignore file containing:

```
example_2
```

...than result of the command will be:

```
$ git status --ignored
```

```
当前分支 master

初始提交

未跟踪的文件：
  (使用 "git add <file>..." 将文件包含到将要提交的内容中)

.gitignore
.example_1

被忽略的文件：
  (使用 "git add -f <file>..." 将文件包含到将要提交的内容中)

目录/
example_2
```

如果你想递归列出目录中被忽略的文件，必须使用额外参数 - --untracked-files=all

结果将如下所示：

```
$ git status --ignored --untracked-files=all
当前分支 master

初始提交

未跟踪的文件：
  (使用 "git add <file>..." 将文件包含到将要提交的内容中)

.gitignore
example_1

被忽略的文件：
  (使用 "git add -f <file>..." 将文件包含到将要提交的内容中)

dir/example_2
example_2
```

第5.13节：仅忽略文件的部分内容 [存根]

有时你可能希望在文件中保留本地更改，但又不想提交或发布。理想情况下，本地设置应集中在一个单独的文件中，该文件可以放入.gitignore，但有时作为短期解决方案，在已检入的文件中保留一些本地内容也是有帮助的。

你可以使用clean过滤器让Git“忽略”那些行。它们甚至不会出现在差异中。

假设这里有文件file1.c的代码片段：

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

你不想在任何地方发布NOCOMMIT行。

通过向Git配置文件（如.git/config）添加以下内容来创建“nocommit”过滤器：


```
On branch master

Initial commit

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

.gitignore
.example_1

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/
example_2
```

If you want to list recursively ignored files in directories, you have to use additional parameter - `--untracked-files=all`

Result will look like this:

```
$ git status --ignored --untracked-files=all
On branch master

Initial commit

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

.gitignore
example_1

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/example_2
example_2
```

Section 5.13: Ignoring only part of a file [stub]

Sometimes you may want to have local changes in a file you don't want to commit or publish. Ideally local settings should be concentrated in a separate file that can be placed into `.gitignore`, but sometimes as a short-term solution it can be helpful to have something local in a checked-in file.

You can make Git "unsee" those lines using clean filter. They won't even show up in diffs.

Suppose here is snippet from file `file1.c`:

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

You don't want to publish `NOCOMMIT` lines anywhere.

Create "nocommit" filter by adding this to Git config file like `.git/config`:

```
[filter "nocommit"]
    clean=grep -v NOCOMMIT
```

将此添加（或创建）到 .git/info/attributes 或 .gitmodules 中：

```
file1.c filter=nocommit
```

你的 NOCOMMIT 行将从 Git 中隐藏。

注意事项：

- 使用 clean 过滤器会减慢文件处理速度，尤其是在 Windows 上。
- 被忽略的行在 Git 更新文件时可能会消失。可以通过 smudge 过滤器来抵消，但这更为复杂。
- 未在 Windows 上测试

第 5.14 节：忽略已跟踪文件的更改。[草稿]

.gitignore 和 .git/info/exclude 仅对未跟踪文件生效。

要在被跟踪的文件上设置忽略标志，请使用命令update-index：

```
git update-index --skip-worktree myfile.c
```

要撤销此操作，请使用：

```
git update-index --no-skip-worktree myfile.c
```

你可以将此代码片段添加到全局 git config 中，以便更方便地使用 `git hide`、`git unhide` 和 `git hidden` 命令：

```
[alias]
hide   = update-index --skip-worktree
    unhide = update-index --no-skip-worktree
    hidden  = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

你也可以使用 update-index 命令的 --assume-unchanged 选项

```
git update-index --assume-unchanged <file>
```

如果你想再次监视此文件的更改，请使用

```
git update-index --no-assume-unchanged <file>
```

当指定 --assume-unchanged 标志时，用户承诺不更改该文件，并允许 Git 假设工作树中的文件与索引中记录的内容一致。Git 在需要修改索引中的该文件时（例如合并提交时）会失败；因此，如果上游更改了该假定未跟踪的文件，你需要手动处理这种情况。此时重点在于性能。

而 --skip-worktree 标志在你指示 git 永远不触碰某个特定文件时非常有用，因为该文件将在本地被更改，你不想意外提交这些更改（例如为特定环境配置的配置/属性文件）。当两者都设置时，skip-worktree 优先于 assume-un changed。


```
[filter "nocommit"]
    clean=grep -v NOCOMMIT
```

Add (or create) this to `.git/info/attributes` or `.gitmodules`:

```
file1.c filter=nocommit
```

And your NOCOMMIT lines are hidden from Git.

Caveats:

- Using clean filter slows down processing of files, especially on Windows.
- The ignored line may disappear from file when Git updates it. It can be counteracted with a smudge filter, but it is trickier.
- Not tested on Windows

Section 5.14: Ignoring changes in tracked files. [stub]

.gitignore and `.git/info/exclude` work only for untracked files.

To set ignore flag on a tracked file, use the command update-index:

```
git update-index --skip-worktree myfile.c
```

To revert this, use:

```
git update-index --no-skip-worktree myfile.c
```

You can add this snippet to your global git config to have more convenient `git hide`, `git unhide` and `git hidden` commands:

```
[alias]
    hide   = update-index --skip-worktree
    unhide = update-index --no-skip-worktree
    hidden  = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

You can also use the option --assume-unchanged with the update-index function

```
git update-index --assume-unchanged <file>
```

If you want to watch this file again for the changes, use

```
git update-index --no-assume-unchanged <file>
```

When --assume-unchanged flag is specified, the user promises not to change the file and allows Git to assume that the working tree file matches what is recorded in the index.Git will fail in case it needs to modify this file in the index e.g. when merging in a commit; thus, in case the assumed-untracked file is changed upstream, you will need to handle the situation manually.The focus lies on performance in this case.

While --skip-worktree flag is useful when you instruct git not to touch a specific file ever because the file is going to be changed locally and you don't want to accidentally commit the changes (i.e configuration/properties file configured for a particular environment). Skip-worktree takes precedence over assume-unchanged when both are set.

第5.15节：清除已提交但包含在.gitignore中的文件

有时会发生这样的情况：某个文件曾被git跟踪，但后来被添加到.gitignore中，
以停止跟踪它。这是一个非常常见的场景，常常忘记在将文件添加到.gitignore之前清理这些文件。
在这种情况下，旧文件仍会保留在仓库中。

为了解决这个问题，可以执行一次对仓库中所有内容的“模拟删除”，然后重新添加所有文件。只要你没有未提交的更改，并且传入了--cached参数，这个命令运行起来相当安全：

```
# 从索引中移除所有内容（文件会保留在文件系统中）
$ git rm -r --cached .

# 重新添加所有内容（它们将以当前状态被添加，包括更改）
$ git add .

# 提交，如果有任何更改。你应该只会看到删除操作
$ git commit -m '移除所有在.gitignore中的文件'

# 更新远程仓库
$ git push origin master
```


Section 5.15: Clear already committed files, but included in .gitignore

Sometimes it happens that a file was being tracked by git, but in a later point in time was added to .gitignore, in order to stop tracking it. It's a very common scenario to forget to clean up such files before its addition to .gitignore. In this case, the old file will still be hanging around in the repository.

To fix this problem, one could perform a "dry-run" removal of everything in the repository, followed by re-adding all the files back. As long as you don't have pending changes and the `--cached` parameter is passed, this command is fairly safe to run:

```
# Remove everything from the index (the files will stay in the file system)
$ git rm -r --cached .

# Re-add everything (they'll be added in the current state, changes included)
$ git add .

# Commit, if anything changed. You should see only deletions
$ git commit -m 'Remove all files that are in the .gitignore'

# Update the remote
$ git push origin master
```

第6章：Git 差异

参数	详情
-p，-u，--patch	生成补丁
-s，--no-patch	抑制差异输出。对于像`git show`这类默认显示补丁的命令非常有用，或者用于取消--patch的效果
--raw	以原始格式生成差异
--diff-algorithm=	选择差异算法。可选变体如下：myers，minimal，patience，histogram
--summary	输出扩展头信息的简要摘要，如创建、重命名和模式更改
--name-only	仅显示已更改文件的名称
--name-status	显示已更改文件的名称和状态。最常见的状态有 M（已修改）、A（已添加）和 D（已删除）
--check	如果更改引入冲突标记或空白错误，则发出警告。什么被视为空白错误由core.whitespace配置控制。默认情况下，行尾空白（包括仅由空白组成的行）以及行首缩进中紧跟着制表符的空格字符被视为空白错误。如果发现问题，则以非零状态退出。不兼容--exit-code
--full-index	在生成补丁格式输出时，在“index”行上显示完整的补丁前后blob对象名称，而不是仅显示前几个字符
--binary	除了--full-index之外，还输出可用git apply应用的二进制差异
-a, --text	将所有文件视为文本。
--color	设置颜色模式；例如，如果您想将差异输出传递给 less 并保持 git 的颜色显示，请使用 --color=always

第6.1节：显示工作分支中的差异

```
git diff
```

这将显示当前分支相对于上一个提交的未暂存更改。它只显示相对于索引的更改，意味着它显示您可以添加到下一个提交但尚未添加的内容。要添加（暂存）这些更改，可以使用git add。

如果文件已暂存，但在暂存后被修改，git diff将显示当前文件与暂存版本之间的差异。

第6.2节：显示两个提交之间的更改

```
git diff 1234abc..6789def    # 旧    新
```

例如：显示最近3次提交所做的更改：

```
git diff @~3..@    # HEAD -3    HEAD
```

注意：两个点（..）是可选的，但能增加清晰度。

这将显示提交之间的文本差异，无论它们在树中的位置如何。

第6.3节：显示暂存文件的差异

```
git diff --staged
```


Chapter 6: Git Diff

Parameter	Details
-p, -u, --patch	Generate patch
-s, --no-patch	Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`
--raw	Generate the diff in raw format
--diff-algorithm=	Choose a diff algorithm. The variants are as follows: `myers`, `minimal`, `patience`, `histogram`
--summary	Output a condensed summary of extended header information such as creations, renames and mode changes
--name-only	Show only names of changed files
--name-status	Show names and statuses of changed files The most common statuses are M (Modified), A (Added), and D (Deleted)
--check	Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code
--full-index	Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output
--binary	In addition to `--full-index`, output a binary diff that can be applied with `git apply`
-a, --text	Treat all files as text.
--color	Set the color mode; i.e. use `--color=always` if you would like to pipe a diff to less and keep git's coloring

Section 6.1: Show differences in working branch

```
git diff
```

This will show the *unstaged* changes on the current branch from the commit before it. It will only show changes relative to the index, meaning it shows what you *could* add to the next commit, but haven't. To add (stage) these changes, you can use `git add`.

If a file is staged, but was modified after it was staged, `git diff` will show the differences between the current file and the staged version.

Section 6.2: Show changes between two commits

```
git diff 1234abc..6789def    # old    new
```

E.g.: Show the changes made in the last 3 commits:

```
git diff @~3..@    # HEAD -3    HEAD
```

Note: the two dots (..) is optional, but adds clarity.

This will show the textual difference between the commits, regardless of where they are in the tree.

Section 6.3: Show differences for staged files

```
git diff --staged
```

这将显示上一次提交与当前暂存文件之间的更改。

注意：你也可以使用以下命令来实现相同的功能：

```
git diff --cached
```

这只是--staged的同义词，或者

```
git status -v
```

这将触发status命令的详细设置。

第6.4节：比较分支

显示new的尖端与original的尖端之间的变化：

```
git diff original new     # 等同于 original..new
```

显示自从从original分支出来后new上的所有更改：

```
git diff original...new     # 等同于 $(git merge-base original new)..new
```

仅使用一个参数，例如

git diff original

等同于

git diff original..HEAD

第6.5节：显示已暂存和未暂存的更改

要显示所有已暂存和未暂存的更改，请使用：

```
git diff HEAD
```

注意： 你也可以使用以下命令：

```
git status -vv
```

区别在于后者的输出实际上会告诉你哪些更改已暂存准备提交，哪些没有。

第6.6节：显示特定文件或目录的差异

```
git diff myfile.txt
```

显示指定文件（myfile.txt）上一次提交与尚未暂存的本地修改版本之间的差异。

这对目录同样适用：

```
git diff documentation
```


This will show the changes between the previous commit and the currently staged files.

NOTE: You can also use the following commands to accomplish the same thing:

```
git diff --cached
```

Which is just a synonym for `--staged` or

```
git status -v
```

Which will trigger the verbose settings of the `status` command.

Section 6.4: Comparing branches

Show the changes between the tip of **`new`** and the tip of **`original`**:

```
git diff original new     # equivalent to original..new
```

Show all changes on **`new`** since it branched from **`original`**:

```
git diff original...new     # equivalent to $(git merge-base original new)..new
```

Using only one parameter such as

git diff original

is equivalent to

git diff original..HEAD

Section 6.5: Show both staged and unstaged changes

To show all staged *and* unstaged changes, use:

```
git diff HEAD
```

NOTE: You can also use the following command:

```
git status -vv
```

The difference being that the output of the latter will actually tell you which changes are staged for commit and which are not.

Section 6.6: Show differences for a specific file or directory

```
git diff myfile.txt
```

Shows the changes between the previous commit of the specified file (`myfile.txt`) and the locally-modified version that has not yet been staged.

This also works for directories:

```
git diff documentation
```

上述命令显示指定目录（documentation/）中所有文件的上一次提交与尚未暂存的本地修改版本之间的差异。

要显示给定提交中某个文件版本与本地HEAD版本之间的差异，可以指定你想比较的提交：

```
git diff 27fa75e myfile.txt
```

或者如果你想查看两个不同提交之间的版本差异：

```
git diff 27fa75e ada9b57 myfile.txt
```

要显示由哈希值ada9b57指定的版本与分支
my_branchname上最新提交之间，仅针对名为my_changed_directory/的相对目录的差异，您可以这样做：

```
git diff ada9b57 my_branchname my_changed_directory/
```

第6.7节：查看长行的单词差异

```
git diff [HEAD|--staged...] --word-diff
```

此命令不会显示整行的更改，而是显示行内的差异。例如，不是显示：

```
-Hello world
+Hello world!
```

整行被标记为更改，word-diff会将输出改为：

```
Hello [-world-]{+world!+}
```

您可以通过指定--word-diff=color或--color-words来省略标记[-，-]、{+，+}。这样只会用颜色编码来标记差异：

```
@@ -1 +1 @@
Hello worldworld!
```

第6.8节：显示当前版本与
上一个版本之间的差异

```
git diff HEAD^ HEAD
```

这将显示上一次提交和当前提交之间的更改。

第6.9节：生成兼容patch的diff

有时你只需要一个可以用patch应用的diff。常规的git --diff不起作用。试试这个：

```
git diff --no-prefix > some_file.patch
```

然后在别处你可以反向应用它：

```
patch -p0 < some_file.patch
```


The above shows the changes between the previous commit of all files in the specified directory (`documentation/`) and the locally-modified versions of these files, that have not yet been staged.

To show the difference between some version of a file in a given commit and the local `HEAD` version you can specify the commit you want to compare against:

```
git diff 27fa75e myfile.txt
```

Or if you want to see the version between two separate commits:

```
git diff 27fa75e ada9b57 myfile.txt
```

To show the difference between the version specified by the hash `ada9b57` and the latest commit on the branch `my_branchname` for only the relative directory called `my_changed_directory/` you can do this:

```
git diff ada9b57 my_branchname my_changed_directory/
```

Section 6.7: Viewing a word-diff for long lines

```
git diff [HEAD|--staged...] --word-diff
```

Rather than displaying lines changed, this will display differences within lines. For example, rather than:

```
-Hello world
+Hello world!
```

Where the whole line is marked as changed, `word-diff` alters the output to:

```
Hello [-world-]{+world!+}
```

You can omit the markers `[-, -]`, `{+, +}` by specifying `--word-diff=color` or `--color-words`. This will only use color coding to mark the difference:

```
@@ -1 +1 @@
Hello worldworld!
```

Section 6.8: Show differences between current version and last version

```
git diff HEAD^ HEAD
```

This will show the changes between the previous commit and the current commit.

Section 6.9: Produce a patch-compatible diff

Sometimes you just need a diff to apply using patch. The regular `git --diff` does not work. Try this instead:

```
git diff --no-prefix > some_file.patch
```

Then somewhere else you can reverse it:

```
patch -p0 < some_file.patch
```

第6.10节：两个提交或分支之间的差异

查看两个分支之间的差异

```
git diff <branch1>..<branch2>
```

查看两个分支之间的差异

```
git diff <commitId1>..<commitId2>
```

查看与当前分支的差异

```
git diff <branch/commitId>
```

查看更改摘要

```
git diff --stat <branch/commitId>
```

查看某次提交后更改的文件

```
git diff --name-only <commitId>
```

查看与某分支不同的文件

```
git diff --name-only <branchName>
```

查看某次提交后文件夹中更改的文件

```
git diff --name-only <commitId> <folder_path>
```

第6.11节：使用meld查看工作目录中的所有修改

```
git difftool -t meld --dir-diff
```

将显示工作目录的更改。或者，

```
git difftool -t meld --dir-diff  [COMMIT_A] [COMMIT_B]
```

将显示两个特定提交之间的差异。

第6.12节：比较UTF-16编码的文本和二进制plist文件

您可以通过指定git如何比较这些文件来比较UTF-16编码的文件（iOS和macOS的本地化字符串文件是示例）。

将以下内容添加到您的~/.gitconfig文件中。

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

iconv 是一个用于转换不同编码的程序。

Section 6.10: difference between two commit or branch

To view difference between two branch

```
git diff <branch1>..<branch2>
```

To view difference between two branch

```
git diff <commitId1>..<commitId2>
```

To view diff with current branch

```
git diff <branch/commitId>
```

To view summary of changes

```
git diff --stat <branch/commitId>
```

To view files that changed after a certain commit

```
git diff --name-only <commitId>
```

To view files that are different than a branch

```
git diff --name-only <branchName>
```

To view files that changed in a folder after a certain commit

```
git diff --name-only <commitId> <folder_path>
```

Section 6.11: Using meld to see all modifications in the working directory

```
git difftool -t meld --dir-diff
```

will show the working directory changes. Alternatively,

```
git difftool -t meld --dir-diff  [COMMIT_A] [COMMIT_B]
```

will show the differences between 2 specific commits.

Section 6.12: Diff UTF-16 encoded text and binary plist files

You can diff UTF-16 encoded files (localization strings file os iOS and macOS are examples) by specifying how git should diff these files.

Add the following to your `~/.gitconfig` file.

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

`iconv` is a program to convert different encodings.

然后在你想使用它的仓库根目录下编辑或创建一个`.gitattributes`文件。或者直接编辑`~/.gitattributes`。

```
*.strings  diff=utf16
```

这将会在git diff之前转换所有以.strings结尾的文件。

你也可以对其他可以转换为文本的文件做类似的操作。

对于二进制plist文件，你需要编辑.gitconfig

```
[diff "plist"]
textconv = plutil -convert xml1 -o -
```

以及.gitattributes

```
*.plist  diff=plist
```


Then edit or create a `.gitattributes` file in the root of the repository where you want to use it. Or just edit `~/.gitattributes`.

```
*.strings diff=utf16
```

This will convert all files ending in `.strings` before git diffs.

You can do similar things for other files, that can be converted to text.

For binary plist files you edit `.gitconfig`

```
[diff "plist"]
textconv = plutil -convert xml1 -o -
```

and `.gitattributes`

```
*.plist diff=plist
```

第7章：撤销操作

第7.1节：返回到之前的提交

要跳回到之前的提交，首先使用git log查找该提交的哈希值。

要临时跳回到该提交，使用以下命令分离你的HEAD：

```
git checkout 789abcd
```

这会将你置于提交789abcd。你现在可以在这个旧提交之上进行新的提交，而不会影响你当前HEAD所在的分支。任何更改都可以通过branch或checkout -b命令转入一个合适的分支。

要回滚到之前的提交，同时保留更改：

```
git reset --soft 789abcd
```

要回滚最后一次提交：

```
git reset --soft HEAD~
```

要永久丢弃在特定提交之后所做的任何更改，请使用：

```
git reset --hard 789abcd
```

要永久丢弃自上一次提交后的所有更改：

```
git reset --hard HEAD~
```

注意：虽然你可以使用reflog和reset恢复被丢弃的提交，但未提交的更改无法恢复。为了安全起见，请使用git stash；而不是git reset --hard。

第7.2节：撤销更改

撤销工作副本中对文件或目录的更改。

```
git checkout -- file.txt
```

在所有文件路径上使用，从当前目录递归执行，将撤销工作副本中的所有更改。

```
git checkout -- .
```

若只想撤销部分更改，请使用--patch。系统会针对每个更改询问是否撤销。

```
git checkout --patch -- dir
```

撤销已添加到index的更改。

```
git reset --hard
```

如果没有--hard标志，则执行软重置。

对于尚未推送到远程的本地提交，你也可以执行软重置。这样你可以重新处理文件

Chapter 7: Undoing

Section 7.1: Return to a previous commit

To jump back to a previous commit, first find the commit's hash using `git log`.

To temporarily jump back to that commit, detach your head with:

```
git checkout 789abcd
```

This places you at commit `789abcd`. You can now make new commits on top of this old commit without affecting the branch your head is on. Any changes can be made into a proper branch using either `branch` or `checkout -b`.

To roll back to a previous commit while keeping the changes:

```
git reset --soft 789abcd
```

To roll back the ***last*** commit:

```
git reset --soft HEAD~
```

To permanently discard any changes made after a specific commit, use:

```
git reset --hard 789abcd
```

To permanently discard any changes made after the ***last*** commit:

```
git reset --hard HEAD~
```

Beware: While you can recover the discarded commits using `reflog` and `reset`, uncommitted changes cannot be recovered. Use `git stash`; `git reset` instead of `git reset --hard` to be safe.

Section 7.2: Undoing changes

Undo changes to a file or directory in the **working copy**.

```
git checkout -- file.txt
```

Used over all file paths, recursively from the current directory, it will undo all changes in the working copy.

```
git checkout -- .
```

To only undo parts of the changes use `--patch`. You will be asked, for each change, if it should be undone or not.

```
git checkout --patch -- dir
```

To undo changes added to the **index**.

```
git reset --hard
```

Without the `--hard` flag this will do a soft reset.

With local commits that you have yet to push to a remote you can also do a soft reset. You can thus rework the files

然后重新提交。

```
git reset HEAD~2
```

上述示例将撤销你最近的两个提交，并将文件恢复到工作副本状态。然后你可以
进行进一步修改并提交新的更改。

注意： 除软重置外，所有这些操作都会永久删除你的更改。为了更安全，可以分别使用git stash -p或git stash。之后你可以用stash pop撤销，或用stash drop永久删除。

第7.3节：使用reflog

如果你在变基时搞砸了，一个重新开始的选项是回到变基之前的提交。你可以使用以下方法做到这一点
reflog（记录了你过去90天内所有操作的历史——此项可配置）：

```
$ git reflog
4a5cbb3 HEAD@{0}：rebase完成：返回到refs/heads/foo
4a5cbb3 HEAD@{1}：rebase：修复了某些问题
904f7f0 HEAD@{2}：rebase：检出upstream/master
3cbe20a HEAD@{3}：提交：修复了某些问题
...
```

你可以看到rebase之前的提交是HEAD@{3}（你也可以检出该哈希）：

```
git checkout HEAD@{3}
```

现在你可以创建一个新分支 / 删除旧分支 / 再次尝试rebase。

你也可以直接重置回reflog中的某个点，但只有在你100%确定这是你想做的操作时才执行：

```
git reset --hard HEAD@{3}
```

这将把你当前的 git 树设置为与当时的状态相匹配（参见撤销更改）。

如果你只是暂时查看一个分支在另一个分支上变基后的效果，但不想保留结果，可以使用此方法。

第7.4节：撤销合并

撤销尚未推送到远程的合并

如果你还没有将合并推送到远程仓库，那么可以按照撤销提交的相同步骤操作，尽管其中有一些细微差别。

重置是最简单的选项，因为它会撤销合并提交以及从该分支添加的任何提交。
但是，你需要知道要重置回哪个 SHA，这可能比较棘手，因为你的git log现在会显示来自两个分支的提交。如果你重置到了错误的提交（例如另一个分支上的提交），这可能会破坏已提交的工作。

```
> git reset --hard <你所在分支的最后一次提交>
```

或者，假设合并是你最近的一次提交。

and then the commits.

```
git reset HEAD~2
```

The above example would unwind your last two commits and return the files to your working copy. You could then make further changes and new commits.

Beware: All of these operations, apart from soft resets, will permanently delete your changes. For a safer option, use `git stash -p` or `git stash`, respectively. You can later undo with `stash pop` or delete forever with `stash drop`.

Section 7.3: Using reflog

If you screw up a rebase, one option to start again is to go back to the commit (pre rebase). You can do this using `reflog` (which has the history of everything you've done for the last 90 days - this can be configured):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase finished: returning to refs/heads/foo
4a5cbb3 HEAD@{1}: rebase: fixed such and such
904f7f0 HEAD@{2}: rebase: checkout upstream/master
3cbe20a HEAD@{3}: commit: fixed such and such
...
```

You can see the commit before the rebase was `HEAD@{3}` (you can also checkout the hash):

```
git checkout HEAD@{3}
```

Now you create a new branch / delete the old one / try the rebase again.

You can also reset directly back to a point in your `reflog`, but only do this if you're 100% sure it's what you want to do:

```
git reset --hard HEAD@{3}
```

This will set your current git tree to match how it was at that point (See Undoing Changes).

This can be used if you're temporarily seeing how well a branch works when rebased on another branch, but you don't want to keep the results.

Section 7.4: Undoing merges

Undoing a merge not yet pushed to a remote

If you haven't yet pushed your merge to the remote repository then you can follow the same procedure as in undo the commit although there are some subtle differences.

A reset is the simplest option as it will undo both the merge commit and any commits added from the branch. However, you will need to know what SHA to reset back to, this can be tricky as your `git log` will now show commits from both branches. If you reset to the wrong commit (e.g. one on the other branch) **it can destroy committed work.**

```
> git reset --hard <last commit from the branch you are on>
```

Or, assuming the merge was your most recent commit.

```
> git reset HEAD~
```

回滚更安全，因为它不会破坏已提交的工作，但需要更多操作，因为你必须先回滚回滚操作，才能再次合并该分支（见下一节）。

撤销推送到远程的合并

假设你合并了一个新功能（add-gremlins）

```
> git merge feature/add-gremlins
...
   #解决任何合并冲突
> git commit #提交合并
...
> git push
...
501b75d..17a51fd  master -> master
```

之后你发现刚合并的功能导致其他开发者的系统出现问题，必须立即撤销，修复该功能本身需要太长时间，所以你只想撤销合并。

```
> git revert -m 1 17a51fd
...
> git push
...
17a51fd..e443799  master -> master
```

此时，系统中的小故障已经消除，你的开发同事们也不再对你大喊大叫了。然而，我们还没有完成。一旦你修复了 add-gremlins 功能的问题，你需要撤销这次回退操作，才能合并回主分支。

```
> git checkout feature/add-gremlins
...
   #各种提交以修复该错误。
> git checkout master
...
> git revert e443799
...
> git merge feature/add-gremlins
...
   #修复由错误修复引入的任何合并冲突
> git commit #提交合并
...
> git push
```

此时，你的功能已经成功添加。然而，鉴于此类错误通常由合并冲突引起，一种稍有不同的工作流程有时更有帮助，因为它允许你在

自己的分支上修复合并冲突。

```
> git checkout feature/add-gremlins
...
   #合并主分支并立即撤销回退操作。这会使你的分支处于
   #与主分支之前相同的错误状态。
> git merge master
...
> git revert e443799
...
   #现在继续修复这个错误（这里会有各种提交）
```



```
> git reset HEAD~
```

A revert is safer, in that it won't destroy committed work, but involves more work as you have to revert the revert before you can merge the branch back in again (see the next section).

Undoing a merge pushed to a remote

Assume you merge in a new feature (add-gremlins)

```
> git merge feature/add-gremlins
...
   #Resolve any merge conflicts
> git commit #commit the merge
...
> git push
...
   501b75d..17a51fd  master -> master
```

Afterwards you discover that the feature you just merged in broke the system for other developers, it must be undone right away, and fixing the feature itself will take too long so you simply want to undo the merge.

```
> git revert -m 1 17a51fd
...
> git push
...
   17a51fd..e443799  master -> master
```

At this point the gremlins are out of the system and your fellow developers have stopped yelling at you. However, we are not finished just yet. Once you fix the problem with the add-gremlins feature you will need to undo this revert before you can merge back in.

```
> git checkout feature/add-gremlins
...
   #Various commits to fix the bug.
> git checkout master
...
> git revert e443799
...
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

At this point your feature is now successfully added. However, given that bugs of this type are often introduced by merge conflicts a slightly different workflow is sometimes more helpful as it lets you fix the merge conflict on your branch.

```
> git checkout feature/add-gremlins
...
   #Merge in master and revert the revert right away.  This puts your branch in
   #the same broken state that master was in before.
> git merge master
...
> git revert e443799
...
   #Now go ahead and fix the bug (various commits go here)
```

```
> git checkout master
...
   #此时不需要撤销之前的撤销操作，因为那已经完成了
> git merge feature/add-gremlins
...
   #修复由错误修复引入的任何合并冲突
> git commit #提交合并
...
> git push
```

第7.5节：撤销一些已有的提交

使用 git revert 来撤销已有的提交，尤其是当这些提交已经推送到远程仓库时。它会记录一些新的提交来逆转之前某些提交的效果，你可以安全地推送这些提交而无需重写历史。

不要使用git push--force，除非你想招致该仓库所有其他用户的谴责。
绝不要重写公共历史。

例如，如果你刚刚推送了一个包含错误的提交，需要撤销它，可以执行以下操作：

```
git revert HEAD~1
git push
```

现在你可以自由地在本地撤销撤销提交，修复代码，然后推送正确的代码：

```
git revert HEAD~1
工作 .. 工作 .. 工作 ..
git add -A .
git commit -m "更新错误代码"
git push
```

如果你想撤销的提交已经在历史中更早的位置，你可以直接传入提交哈希。Git 会创建一个反向提交来撤销你原来的提交，你可以安全地将其推送到远程仓库。

```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

第7.6节：撤销/重做一系列提交

假设你想撤销十几个提交，但只想撤销其中的一部分。

```
git rebase -i <较早的SHA>
```

-i 将 rebase 置于“交互模式”。它的开始方式与上面讨论的 rebase 类似，但在重放任何提交之前，它会暂停并允许你在重放每个提交时轻松修改它。rebase -i 会在你的默认文本编辑器中打开，显示正在应用的提交列表，如下所示：


```
> git checkout master
...
   #Don't need to revert the revert at this point since it was done earlier
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

Section 7.5: Revert some existing commits

Use git revert to revert existing commits, especially when those commits have been pushed to a remote repository. It records some new commits to reverse the effect of some earlier commits, which you can push safely without rewriting history.

Don't use `git push --force` unless you wish to bring down the opprobrium of all other users of that repository. Never rewrite public history.

If, for example, you've just pushed up a commit that contains a bug and you need to back it out, do the following:

```
git revert HEAD~1
git push
```

Now you are free to revert the revert commit locally, fix your code, and push the good code:

```
git revert HEAD~1
work .. work .. work ..
git add -A .
git commit -m "Update error code"
git push
```

If the commit you want to revert is already further back in the history, you can simply pass the commit hash. Git will create a counter-commit undoing your original commit, which you can push to your remote safely.

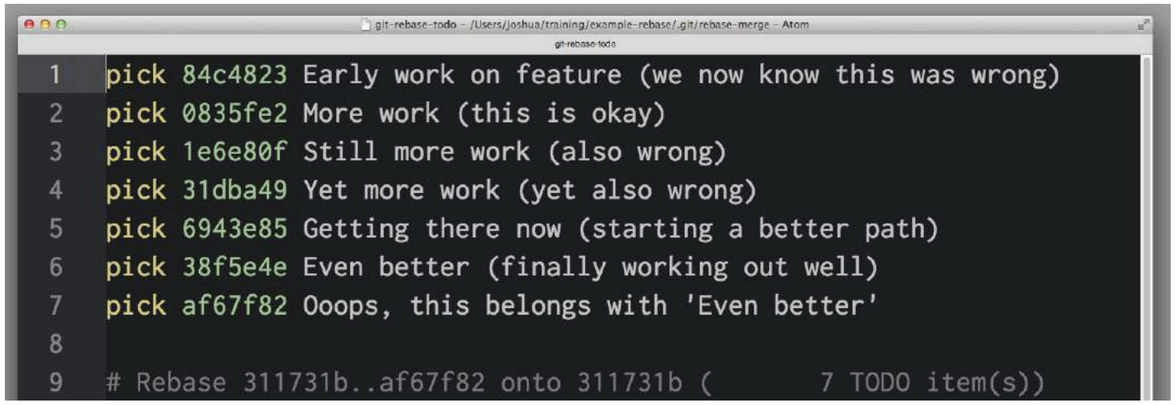
```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

Section 7.6: Undo / Redo a series of commits

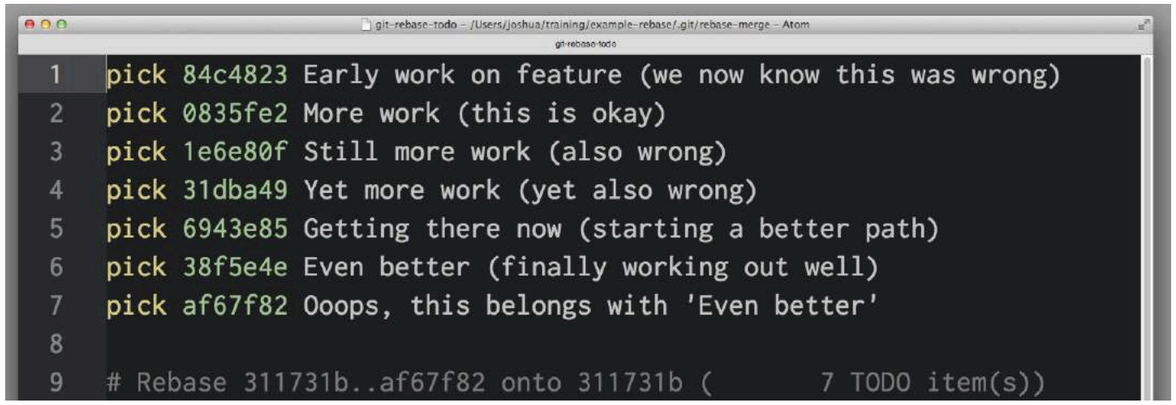
Assume you want to undo a dozen of commits and you want only some of them.

```
git rebase -i <earlier SHA>
```

-i puts rebase in "interactive mode". It starts off like the rebase discussed above, but before replaying any commits, it pauses and allows you to gently modify each commit as it's replayed.*`rebase -i`* will open in your default text editor, with a list of commits being applied, like this:



要丢弃一个提交，只需在编辑器中删除该行。如果你不再希望项目中存在错误的提交，你可以删除上面第1行和第3-4行。如果你想将两个提交合并，可以使用 squash或 fixup命令。



To drop a commit, just delete that line in your editor. If you no longer want the bad commits in your project, you can delete lines 1 and 3-4 above.If you want to combine two commits together, you can use the `squash` or `fixup` commands

第8章：合并

参数	详情
`-m`	包含在合并提交中的信息
`-v`	显示详细输出
`--abort`	尝试将所有文件恢复到原状态
`--ff-only`	当需要合并提交时立即中止
`--no-ff`	强制创建合并提交，即使不是强制要求
--no-commit	假装合并失败以便检查和调整结果
`--stat`	合并完成后显示差异统计
-n/--no-stat	不显示差异统计
`--squash`	允许在当前分支上进行一次包含合并更改的单一提交

第8.1节：自动合并

当两个分支上的提交不冲突时，Git可以自动合并它们：

```
~/Stack Overflow(branch:master) » git merge another_branch
自动合并 file_a
合并采用了“递归”策略。
file_a | 2 +-
 1 file 已更改，1 次插入(+)，1 次删除(-)
```

第8.2节：查找所有没有合并更改的分支

有时你可能会有一些分支已经将更改合并到主分支（master）中。
此命令查找所有不是`master`且与`master`相比没有独特提交的分支。这对于
查找在PR合并到master后未被删除的分支非常有用。

```
for branch in $(git branch -r) ; do
  [ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] && echo -
e `git show --pretty=format:"%ci %cr" $branch | head -n 1`\$branch
done | sort -r
```

第8.3节：中止合并

开始合并后，你可能想停止合并并将所有内容恢复到合并前的状态。使用--abort：

```
git merge --abort
```

第8.4节：带提交的合并

默认行为是在合并以快进方式解决时，仅更新分支指针，而不创建合并提交。使用--no-ff来解决。

```
git merge <branch_name> --no-ff -m "<commit message>"
```

第8.5节：仅保留合并一方的更改

在合并过程中，可以向git checkout传递--ours或--theirs，以从合并的一方获取文件的所有更改。

Chapter 8: Merging

Parameter	Details
`-m`	Message to be included in the merge commit
`-v`	Show verbose output
`--abort`	Attempt to revert all files back to their state
`--ff-only`	Aborts instantly when a merge-commit would be required
`--no-ff`	Forces creation of a merge-commit, even if it wasn't mandatory
`--no-commit`	Pretends the merge failed to allow inspection and tweaking of the result
`--stat`	Show a diffstat after merge completion
`-n/--no-stat`	Don't show the diffstat
`--squash`	Allows for a single commit on the current branch with the merged changes

Section 8.1: Automatic Merging

When the commits on two branches don't conflict, Git can automatically merge them:

```
~/Stack Overflow(branch:master) » git merge another_branch
Auto-merging file_a
Merge made by the 'recursive' strategy.
 file_a | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Section 8.2: Finding all branches with no merged changes

Sometimes you might have branches lying around that have already had their changes merged into master. This finds all branches that are not `master` that have no unique commits as compared to `master`. This is very useful for finding branches that were not deleted after the PR was merged into master.

```
for branch in $(git branch -r) ; do
  [ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] && echo -
e `git show --pretty=format:"%ci %cr" $branch | head -n 1`\\t$branch
done | sort -r
```

Section 8.3: Aborting a merge

After starting a merge, you might want to stop the merge and return everything to its pre-merge state. Use `--abort`:

```
git merge --abort
```

Section 8.4: Merge with a commit

Default behaviour is when the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. Use `--no-ff` to resolve.

```
git merge <branch_name> --no-ff -m "<commit message>"
```

Section 8.5: Keep changes from only one side of a merge

During a merge, you can pass `--ours` or `--theirs` to `git checkout` to take all changes for a file from one side or the other of a merge.

```
$ git checkout --ours   -- file1.txt # 使用我们的file1版本，删除他们的所有更改
$ git checkout --theirs -- file2.txt # 使用他们的file2版本，删除我们的所有更改
```

第8.6节：将一个分支合并到另一个分支

```
git merge incomingBranch
```

这会将分支`incomingBranch`合并到你当前所在的分支中。例如，如果你当前在`master`分支，则`incomingBranch`将被合并到`master`中。

合并在某些情况下可能会产生冲突。如果发生这种情况，你会看到消息自动合并失败；请修复冲突然后提交结果。你需要手动编辑冲突文件，或者要撤销合并尝试，运行：

```
git merge --abort
```



```
$ git checkout --ours   -- file1.txt # Use our version of file1, delete all their changes
$ git checkout --theirs -- file2.txt # Use their version of file2, delete all our changes
```

Section 8.6: Merge one branch into another

```
git merge incomingBranch
```

This merges the branch `incomingBranch` into the branch you are currently in. For example, if you are currently in `master`, then `incomingBranch` will be merged into `master`.

Merging can create conflicts in some cases. If this happens, you will see the message `Automatic merge failed; fix conflicts and then commit the result.` You will need to manually edit the conflicted files, or to undo your merge attempt, run:

```
git merge --abort
```

第9章：子模块

第9.1节：克隆包含子模块的Git仓库

当你克隆一个使用子模块的仓库时，需要初始化并更新这些子模块。

```
$ git clone --recursive https://github.com/username/repo.git
```

这将克隆引用的子模块并将它们放置在相应的文件夹中（包括子模块中的子模块）。这相当于在克隆完成后立即运行 git submodule update --init --recursive。

第9.2节：更新子模块

子模块引用另一个仓库中的特定提交。要检出所有子模块所引用的确切状态，请运行

```
git submodule update --recursive
```

有时你可能不想使用引用的状态，而是想将本地检出更新到远程子模块的最新状态。要使用单个命令将所有子模块检出到远程的最新状态，可以使用

```
git submodule foreach git pull <remote> <branch>
```

或者使用默认的 git pull 参数

```
git submodule foreach git pull
```

请注意，这只会更新您的本地工作副本。运行`git status`时，如果该命令导致子模块目录发生变化，子模块目录会被列为脏状态。要更新您的仓库以引用新的状态，您必须
提交更改：

```
git add <子模块目录>
git commit
```

如果您使用git pull，可能会有一些更改导致合并冲突，因此您可以使用git pull--rebase将您的更改回滚到最顶端，大多数情况下这会减少冲突的可能性。同时它会将所有分支拉取到本地。

```
git submodule foreach git pull --rebase
```

要检出特定子模块的最新状态，您可以使用：

```
git submodule update --remote <子模块目录>
```

第9.3节：添加子模块

您可以将另一个Git仓库作为项目中的一个文件夹包含进来，由Git进行跟踪：

```
$ git submodule add https://github.com/jquery/jquery.git
```


Chapter 9: Submodules

Section 9.1: Cloning a Git repository having submodules

When you clone a repository that uses submodules, you'll need to initialize and update them.

```
$ git clone --recursive https://github.com/username/repo.git
```

This will clone the referenced submodules and place them in the appropriate folders (including submodules within submodules). This is equivalent to running `git submodule update --init --recursive` immediately after the clone is finished.

Section 9.2: Updating a Submodule

A submodule references a specific commit in another repository. To check out the exact state that is referenced for all submodules, run

```
git submodule update --recursive
```

Sometimes instead of using the state that is referenced you want to update to your local checkout to the latest state of that submodule on a remote. To check out all submodules to the latest state on the remote with a single command, you can use

```
git submodule foreach git pull <remote> <branch>
```

or use the default `git pull` arguments

```
git submodule foreach git pull
```

Note that this will just update your local working copy. Running `git status` will list the submodule directory as dirty if it changed because of this command. To update your repository to reference the new state instead, you have to commit the changes:

```
git add <submodule_directory>
git commit
```

There might be some changes you have that can have merge conflict if you use `git pull` so you can use `git pull --rebase` to rewind your changes to top, most of the time it decreases the chances of conflict. Also it pulls all the branches to local.

```
git submodule foreach git pull --rebase
```

To checkout the latest state of a specific submodule, you can use :

```
git submodule update --remote <submodule_directory>
```

Section 9.3: Adding a submodule

You can include another Git repository as a folder within your project, tracked by Git:

```
$ git submodule add https://github.com/jquery/jquery.git
```

你应该添加并提交新的.gitmodules文件；这告诉Git在运行git submodule update时应该克隆哪些子模块。

第9.4节：设置子模块以跟随分支

子模块总是检出到特定的提交SHA1（“gitlink”，父仓库索引中的特殊条目）

但可以请求将该子模块更新到子模块远程仓库某个分支的最新提交。

与其进入每个子模块，执行git checkout一个分支--track origin/分支，git pull，不如直接在父仓库执行：

```
git submodule update --remote --recursive
```

由于子模块的SHA1会改变，之后仍需执行：

```
git add .
git commit -m "update submodules"
```

这假设子模块是：

- 要么是添加时指定了要跟随的分支：

```
git submodule -b 一个分支 -- /url/of/子模块/仓库
```

- 或配置（针对现有子模块）以跟踪某个分支：

```
cd /path/to/parent/repo
git config -f .gitmodules submodule.asubmodule.branch abranch
```

第9.5节：移动子模块

版本 > 1.8

运行：

$ git mv /path/to/module new/path/to/module
版本 ≤ 1.8

1. 编辑 .gitmodules 并适当更改子模块的路径，然后使用 git add 将其放入索引 .gitmodules。
2. 如有需要，创建子模块新位置的父目录（mkdir -p /path/to）。
3. 将所有内容从旧目录移动到新目录（mv -vi /path/to/module new/path/to/submodule）。
4. 确保 Git 跟踪该目录（git add /path/to）。
5. 使用 git rm --cached /path/to/module 删除旧目录。
6. 将目录 .git/modules//path/to/module 及其所有内容移动到 .git/modules//path/to/module。
7. 编辑 .git/modules//path/to/config 文件，确保 worktree 项指向新位置，因此在此示例中应为 worktree = ../../../../../path/to/module。通常在该位置的直接路径中应有另外两个 .. 目录。编辑文件 /path/to/module/.git，确保路径

You should add and commit the new .gitmodules file; this tells Git what submodules should be cloned when git submodule update is run.

Section 9.4: Setting a submodule to follow a branch

A submodule is always checked out at a specific commit SHA1 (the "gitlink", special entry in the index of the parent repo)

But one can request to update that submodule to the latest commit of a branch of the submodule remote repo.

Rather than going in each submodule, doing a git checkout abranch --track origin/abranch, git pull, you can simply do (from the parent repo) a:

```
git submodule update --remote --recursive
```

Since the SHA1 of the submodule would change, you would still need to follow that with:

```
git add .
git commit -m "update submodules"
```

That supposes the submodules were:

- either added with a branch to follow:

```
git submodule -b abranch -- /url/of/submodule/repo
```

- or configured (for an existing submodule) to follow a branch:

```
cd /path/to/parent/repo
git config -f .gitmodules submodule.asubmodule.branch abranch
```

Section 9.5: Moving a submodule

Version > 1.8

Run:

$ git mv /path/to/module *new/path/to/module*
Version ≤ 1.8

1. Edit .gitmodules and change the path of the submodule appropriately, and put it in the index with git add .gitmodules.
2. If needed, create the parent directory of the new location of the submodule (mkdir -p /path/to).
3. Move all content from the old to the new directory (mv -vi /path/to/module *new/path/to/submodule*).
4. Make sure Git tracks this directory (git add /path/to).
5. Remove the old directory with git rm --cached /path/to/module.
6. Move the directory .git/modules//path/to/module with all its content to .git/modules//path/to/module.
7. Edit the .git/modules//path/to/config file, make sure that worktree item points to the new locations, so in this example it should be worktree = ../../../../../path/to/module. Typically there should be two more .. then directories in the direct path in that place. . Edit the file /path/to/module/.git, make sure that the path

它指向主项目中正确的新位置.git文件夹，因此在此示例中gitdir: ../../../.git/modules//path/to/module.

git status的输出结果如下：

```
# 在master分支上
# 将要提交的更改：
#   （使用 "git reset HEAD <file>..." 来取消暂存）
#
#       修改：   .gitmodules
#       重命名：    old/path/to/submodule -> new/path/to/submodule
#
```

8. 最后，提交更改。

此示例来自Stack Overflow，作者Axel Beckert

第9.6节：移除子模块

版本 > 1.8

您可以通过调用以下命令来移除子模块（例如 the_submodule）：

```
$ git submodule deinit the_submodule
$ git rm the_submodule
```

- git submodule deinit the_submodule 会删除 .git/config 中 the_submodule 的条目。这会将 the_submodule 排除在 git submodule update、git submodule sync 和 git submodule foreach 调用之外，并删除其本地内容（source）。此外，这不会在你的父仓库中显示为更改。执行 git submodule init 和 git submodule update 会恢复子模块，同样不会在父仓库中产生可提交的更改。

- git rm the_submodule 会从工作区移除子模块。文件将被删除，同时子模块在 .gitmodules 文件中的条目也会被删除（source）。如果仅执行 git rm the_submodule（未先执行 git submodule deinit the_submodule ，则 .git/config 文件中的子模块条目仍然会保留。

版本 < 1.8

摘自 here：

1. 从 .gitmodules 文件中删除相关部分。
2. 暂存 .gitmodules 的更改 git add .gitmodules
3. 从 .git/config 中删除相关部分。
4. 运行 git rm --cached path_to_submodule（无尾部斜杠）。
5. 运行 rm -rf .git/modules/path_to_submodule
6. 提交 git commit -m "移除子模块 <name>"
7. 删除现在未被跟踪的子模块文件
8. rm -rf path_to_submodule

in it points to the correct new location inside the main project `.git` folder, so in this example `gitdir: ../../../.git/modules//path/to/module`.

`git status` output looks like this afterwards:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   .gitmodules
#       renamed:    old/path/to/submodule -> new/path/to/submodule
#
```

8. Finally, commit the changes.

This example from Stack Overflow, by Axel Beckert

Section 9.6: Removing a submodule

Version > 1.8

You can remove a submodule (e.g. `the_submodule`) by calling:

```
$ git submodule deinit the_submodule
$ git rm the_submodule
```

- `git submodule deinit the_submodule` deletes `the_submodules`' entry from .git/config. This excludes the_submodule from `git submodule update`, `git submodule sync` and `git submodule foreach` calls and deletes its local content (source). Also, this will not be shown as change in your parent repository. `git submodule init` and `git submodule update` will restore the submodule, again without commitable changes in your parent repository.

- `git rm the_submodule` will remove the submodule from the work tree. The files will be gone as well as the submodules' entry in the `.gitmodules` file (source). If only `git rm the_submodule` (without prior `git submodule deinit the_submodule` is run, however, the submodules' entry in your .git/config file will remain.

Version < 1.8

Taken from here:

1. Delete the relevant section from the `.gitmodules` file.
2. Stage the `.gitmodules` changes `git add .gitmodules`
3. Delete the relevant section from `.git/config`.
4. Run `git rm --cached path_to_submodule` (no trailing slash).
5. Run `rm -rf .git/modules/path_to_submodule`
6. Commit `git commit -m "Removed submodule <name>"`
7. Delete the now untracked submodule files
8. `rm -rf path_to_submodule`

第10章：提交

参数	详情
--message, -m	包含在提交中的信息。指定此参数将跳过Git正常打开编辑器的行为。
--amend	指定当前已暂存的更改应添加（修改）到之前的提交。请小心，这可能会重写历史！
--no-edit	使用选定的提交信息而不启动编辑器。例如，gitcommit --amend --no-edit 会修改提交但不更改其提交信息。
--all, -a	提交所有更改，包括尚未暂存的更改。
--date	手动设置将与提交关联的日期。
--only	仅提交指定的路径。除非另有指示，否则不会提交当前已暂存的内容。
--patch, -p	使用交互式补丁选择界面选择要提交的更改。
--help	显示git commit的手册页
-S[keyid], -S --gpg-签名[=keyid], -S --no-gpg-sign	签署提交，GPG签署提交，撤销commit.gpgSign配置变量
-n, --no-verify	此选项跳过 pre-commit 和 commit-msg 钩子。另见 钩子

使用 Git 提交通过将更改归因于作者来提供责任追踪。Git 提供多种功能以确保提交的特异性和安全性。本主题解释并演示了使用 Git 提交的正确做法和流程。

第10.1节：暂存并提交更改

基础知识

在对源代码进行更改后，您应先使用 Git暂存这些更改，然后才能提交它们。

例如，如果你修改了README.md和program.py：

```
git add README.md program.py
```

这告诉 git 你想将这些文件添加到下一次提交中。

然后，使用以下命令提交你的更改

```
git commit
```

注意，这会打开一个文本编辑器，通常是 vim。如果你不熟悉 vim，可能需要知道
你可以按 `i` 进入 *insert* 模式，写下你的提交信息，然后按 `Esc` 和 `:wq` 保存并退出。为了避免打开文本编辑器，只需在命令中包含 `-m` 标志和你的信息

```
git commit -m "Commit message here"
```

提交信息通常遵循一些特定的格式规则，更多信息请参见优秀的提交信息。

快捷方式

如果你在目录中更改了很多文件，而不想逐个列出，可以使用：

Chapter 10: Committing

Parameter	Details
--message, -m	Message to include in the commit. Specifying this parameter bypasses Git's normal behavior of opening an editor.
--amend	Specify that the changes currently staged should be added (amended) to the *previous* commit. Be careful, this can rewrite history!
--no-edit	Use the selected commit message without launching an editor. For example, `git commit --amend --no-edit` amends a commit without changing its commit message.
--all, -a	Commit all changes, including changes that aren't yet staged.
--date	Manually set the date that will be associated with the commit.
--only	Commit only the paths specified. This will not commit what you currently have staged unless told to do so.
--patch, -p	Use the interactive patch selection interface to chose which changes to commit.
--help	Displays the man page for `git commit`
-S[keyid], -S --gpg-sign[=keyid], -S --no-gpg-sign	Sign commit, GPG-sign commit, countermand `commit.gpgSign` configuration variable
-n, --no-verify	This option bypasses the pre-commit and commit-msg hooks. See also Hooks

Commits with Git provide accountability by attributing authors with changes to code. Git offers multiple features for the specificity and security of commits. This topic explains and demonstrates proper practices and procedures in committing with Git.

Section 10.1: Stage and commit changes

The basics

After making changes to your source code, you should **stage** those changes with Git before you can commit them.

For example, if you change `README.md` and `program.py`:

```
git add README.md program.py
```

This tells git that you want to add the files to the next commit you do.

Then, commit your changes with

```
git commit
```

Note that this will open a text editor, which is often vim. If you are not familiar with vim, you might want to know that you can press `i` to go into *insert* mode, write your commit message, then press `Esc` and `:wq` to save and quit. To avoid opening the text editor, simply include the `-m` flag with your message

```
git commit -m "Commit message here"
```

Commit messages often follow some specific formatting rules, see Good commit messages for more information.

Shortcuts

If you have changed a lot of files in the directory, rather than listing each one of them, you could use:

```
git add --all        # 等同于 "git add -a"
```

或者从顶层目录及子目录中添加所有更改，不包括已删除的文件：

```
git add .
```

或者只添加当前已被跟踪的文件（“更新”）：

```
git add -u
```

如果需要，查看已暂存的更改：

```
git status           # 显示已更改文件列表
git diff --cached    # 显示已暂存文件中的暂存更改
```

最后，提交更改：

```
git commit -m "Commit message here"
```

或者，如果您只修改了现有文件或删除了文件，且没有创建任何新文件，您可以将 `git add` 和 `git commit` 的操作合并为一个命令：

```
git commit -am "在此处填写提交信息"
```

请注意，这将以与 git add --all 相同的方式暂存所有已修改的文件。

敏感数据

您绝不应该提交任何敏感数据，例如密码甚至私钥。如果发生这种情况且更改已经推送到中央服务器，则应视所有敏感数据为已泄露。否则，可以在事后删除这些数据。一种快速且简便的解决方案是使用“BFG Repo-Cleaner”：

https://rtyley.github.io/bfg-repo-cleaner/.

命令 `bfg --replace-text passwords.txt my-repo.git` 会从 `passwords.txt` 文件中读取密码并将其替换为 `***REMOVED***`。此操作会考虑整个仓库的所有历史提交。

第10.2节：良好的提交信息

对于浏览 git log 的人来说，能够轻松理解每次提交的内容非常重要。
良好的提交信息通常包括任务编号或问题追踪器中的编号，以及对所做内容和原因的简明描述，有时还包括具体的实现方式。

更好的提交信息示例如下：

```
TASK-123: 实现通过OAuth登录
TASK-124: 添加JS/CSS文件的自动压缩
TASK-125: 修复名称超过200字符时压缩器错误
```

而以下信息则不太有用：

```
fix                          // 修复了什么？
```



```
git add --all        # equivalent to "git add -a"
```

Or to add all changes, *not including files that have been deleted*, from the top-level directory and subdirectories:

```
git add .
```

Or to only add files which are currently tracked ("update"):

```
git add -u
```

If desired, review the staged changes:

```
git status           # display a list of changed files
git diff --cached    # shows staged changes inside staged files
```

Finally, commit the changes:

```
git commit -m "Commit message here"
```

Alternately, if you have only modified existing files or deleted files, and have not created any new ones, you can combine the actions of `git add` and `git commit` in a single command:

```
git commit -am "Commit message here"
```

Note that this will stage **all** modified files in the same way as `git add --all`.

Sensitive data

You should never commit any sensitive data, such as passwords or even private keys. If this case happens and the changes are already pushed to a central server, consider any sensitive data as compromised. Otherwise, it is possible to remove such data afterwards. A fast and easy solution is the usage of the "BFG Repo-Cleaner": https://rtyley.github.io/bfg-repo-cleaner/.

The command `bfg --replace-text passwords.txt my-repo.git` reads passwords out of the `passwords.txt` file and replaces these with `***REMOVED***`. This operation considers all previous commits of the entire repository.

Section 10.2: Good commit messages

It is important for someone traversing through the `git log` to easily understand what each commit was all about. Good commit messages usually include a number of a task or an issue in a tracker and a concise description of what has been done and why, and sometimes also how it has been done.

Better messages may look like:

```
TASK-123: Implement login through OAuth
TASK-124: Add auto minification of JS/CSS files
TASK-125: Fix minifier error when name > 200 chars
```

Whereas the following messages would not be quite as useful:

```
fix                          // What has been fixed?
```

```
just a bit of a change      // 变更了什么？
TASK-371                     // 完全没有描述，读者需要自己查看跟踪器以获取解释

在 IBar 中实现了 IFoo     // 为什么需要这样做？
```

测试提交信息是否使用正确语气的一种方法是将空白处替换为该信息，看看是否通顺合理：

如果我添加这个提交，我将__到我的代码库中。

优秀 git 提交信息的七条规则

1. 用空行将主题行与正文分开
2. 主题行限制在 50 个字符以内
3. 主题行首字母大写
4. 主题行末尾不加句号
5. 主题行使用祈使语气
6. 手动将正文每行换行，限制在 72 个字符以内
7. 正文用来解释*什么*和*为什么*，而不是*如何*

来自 Chris Beam 博客的 7 条规则。

第 10.3 节：修改提交

如果您的**最新提交尚未发布**（未推送到上游仓库），则可以修改您的提交。

```
git commit --amend
```

这将把当前暂存的更改放到上一次提交中。

注意： 这也可以用来编辑错误的提交信息。它会调出默认编辑器（通常是 `vi` / **`vim`** / `emacs`），允许你修改之前的提交信息。

要在命令行中指定提交信息：

```
git commit --amend -m "新的提交信息"
```

或者使用之前的提交信息而不做更改：

```
git commit --amend --no-edit
```

修改提交会更新提交日期，但保持作者日期不变。你可以让 git 刷新该信息。

```
git commit --amend --reset-author
```

你也可以通过以下命令更改提交的作者：

```
git commit --amend --author "New Author <email@address.com>"
```

注意： 请注意，修改最近的提交会完全替换该提交，之前的提交会从分支历史中移除。在使用公共仓库和与其他协作者共同工作的分支时，应牢记这一点。


```
just a bit of a change      // What has changed?
TASK-371                    // No description at all, reader will need to look at the tracker
themselves for an explanation
Implemented IFoo in IBar    // Why it was needed?
```

A way to test if a commit message is written in the correct mood is to replace the blank with the message and see if it makes sense:

If I add this commit, I will ___ to my repository.

The seven rules of a great git commit message

1. Separate the subject line from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Manually wrap each line of the body at 72 characters
7. Use the body to explain *what* and *why* instead of *how*

7 rules from Chris Beam's blog.

Section 10.3: Amending a commit

If your **latest commit is not published yet** (not pushed to an upstream repository) then you can amend your commit.

```
git commit --amend
```

This will put the currently staged changes onto the previous commit.

Note: This can also be used to edit an incorrect commit message. It will bring up the default editor (usually `vi` / **`vim`** / `emacs`) and allow you to change the prior message.

To specify the commit message inline:

```
git commit --amend -m "New commit message"
```

Or to use the previous commit message without changing it:

```
git commit --amend --no-edit
```

Amending updates the commit date but leaves the author date untouched. You can tell git to refresh the information.

```
git commit --amend --reset-author
```

You can also change the author of the commit with:

```
git commit --amend --author "New Author <email@address.com>"
```

Note: Be aware that amending the most recent commit replaces it entirely and the previous commit is removed from the branch's history. This should be kept in mind when working with public repositories and on branches with other collaborators.

这意味着如果之前的提交已经被推送，修改后你将需要使用 push --force。

第10.4节：提交时不打开编辑器

当你运行 git commit 时，Git 通常会打开一个编辑器（如 vim 或 emacs）。使用 -m 选项可以直接从命令行指定提交信息：

```
git commit -m "Commit message here"
```

你的提交信息可以跨多行：

```
git commit -m "在这里写提交的'主题行'信息

更详细的描述写在这里（空一行后）。"
```

或者，你可以传入多个 -m 参数：

```
git commit -m "提交摘要" -m "更详细的描述写在这里"
```

参见 如何编写 Git 提交信息。

Udacity Git 提交信息风格指南

第10.5节：直接提交更改

通常，你需要使用`git add`或`git rm`将更改添加到索引中，然后才能`git commit`提交它们。传递`-a`或`--all`选项可以自动将每个更改（针对已跟踪的文件）添加到索引中，包括删除操作：

```
git commit -a
```

如果你还想添加提交信息，可以这样做：

```
git commit -a -m "your commit message goes here"
```

此外，你可以合并两个标志：

```
git commit -am "your commit message goes here"
```

你不一定要一次提交所有文件。省略-a或--all标志，直接指定你想要提交的文件：

```
git commit path/to/a/file -m "your commit message goes here"
```

如果要直接提交多个特定文件，也可以指定一个或多个文件、目录和模式：

```
git commit path/to/a/file path/to/a/folder/* path/to/b/file -m "your commit message goes here"
```

第10.6节：选择哪些行应被暂存以便提交

假设你在一个或多个文件中有许多更改，但你只想提交每个文件中的部分更改，你可以使用以下命令选择所需的更改：

This means that if the earlier commit had already been pushed, after amending it you will have to `push --force`.

Section 10.4: Committing without opening an editor

Git will usually open an editor (like `vim` or `emacs`) when you run `git commit`. Pass the `-m` option to specify a message from the command line:

```
git commit -m "Commit message here"
```

Your commit message can go over multiple lines:

```
git commit -m "Commit 'subject line' message here

More detailed description follows here (after a blank line)."
```

Alternatively, you can pass in multiple `-m` arguments:

```
git commit -m "Commit summary" -m "More detailed description follows here"
```

See How to Write a Git Commit Message.

Udacity Git Commit Message Style Guide

Section 10.5: Committing changes directly

Usually, you have to use `git add` or `git rm` to add changes to the index before you can `git commit` them. Pass the `-a` or `--all` option to automatically add every change (to tracked files) to the index, including removals:

```
git commit -a
```

If you would like to also add a commit message you would do:

```
git commit -a -m "your commit message goes here"
```

Also, you can join two flags:

```
git commit -am "your commit message goes here"
```

You don't necessarily need to commit all files at once. Omit the `-a` or `--all` flag and specify which file you want to commit directly:

```
git commit path/to/a/file -m "your commit message goes here"
```

For directly committing more than one specific file, you can specify one or multiple files, directories and patterns as well:

```
git commit path/to/a/file path/to/a/folder/* path/to/b/file -m "your commit message goes here"
```

Section 10.6: Selecting which lines should be staged for committing

Suppose you have many changes in one or more files but from each file you only want to commit some of the changes, you can select the desired changes using:

```
git add -p
```

或

```
git add -p [file]
```

你的每个更改都会被单独显示，对于每个更改，你将被提示选择以下选项之一：

```
y - 是，添加此块

n - 否，不添加此块

d - 否，不添加此块，也不添加该文件的任何剩余块。
如果你已经添加了想要的内容，并想跳过其余部分，这个选项很有用。

s - 如果可能，将该块拆分成更小的块

e - 手动编辑该块。这可能是最强大的选项。
它将在文本编辑器中打开该块，您可以根据需要进行编辑。
```

这将暂存你选择的文件部分。然后你可以像这样提交所有已暂存的更改：

```
git commit -m '提交信息'
```

未被暂存或提交的更改仍会出现在你的工作文件中，如果需要，可以稍后提交。或者如果剩余的更改不需要，可以使用以下命令丢弃：

```
git reset --hard
```

除了将大改动拆分成更小的提交之外，这种方法对于审查你即将提交的内容也很有用。通过逐个确认每个更改，你有机会检查自己写的内容，并且可以避免意外暂存不需要的代码，比如println/日志语句。

第10.7节：创建空提交

一般来说，空提交（或状态与父提交相同的提交）是错误的。

但是，在测试构建钩子、持续集成系统以及其他基于提交触发的系统时，能够轻松创建提交而无需编辑/触碰虚拟文件是很方便的。

--allow-empty 提交选项将绕过该检查。

```
git commit -m "这是一个空提交" --allow-empty
```

第10.8节：代表他人提交

如果代码是别人写的，你可以用--author选项来给他们署名：

```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

你也可以提供一个模式，Git 会用它来搜索之前的作者：

```
git commit -m "msg" --author "John"
```



```
git add -p
```

or

```
git add -p [file]
```

Each of your changes will be displayed individually, and for each change you will be prompted to choose one of he following options:

```
y - Yes, add this hunk

n - No, don't add this hunk

d - No, don't add this hunk, or any other remaining hunks for this file.
Useful if you've already added what you want to, and want to skip over the rest.

s - Split the hunk into smaller hunks, if possible

e - Manually edit the hunk.  This is probably the most powerful option.
It will open the hunk in a text editor and you can edit it as needed.
```

This will stage the parts of the files you choose. Then you can commit all the staged changes like this:

```
git commit -m 'Commit Message'
```

The changes that were not staged or committed will still appear in your working files, and can be committed later if required. Or if the remaining changes are unwanted, they can be discarded with:

```
git reset --hard
```

Apart from breaking up a big change into smaller commits, this approach is also useful for *reviewing* what you are about to commit. By individually confirming each change, you have an opportunity to check what you wrote, and can avoid accidentally staging unwanted code such as println/logging statements.

Section 10.7: Creating an empty commit

Generally speaking, empty commits (or commits with state that is identical to the parent) is an error.

However, when testing build hooks, CI systems, and other systems that trigger off a commit, it's handy to be able to easily create commits without having to edit/touch a dummy file.

The `--allow-empty` commit will bypass the check.

```
git commit -m "This is a blank commit" --allow-empty
```

Section 10.8: Committing on behalf of someone else

If someone else wrote the code you are committing, you can give them credit with the `--author` option:

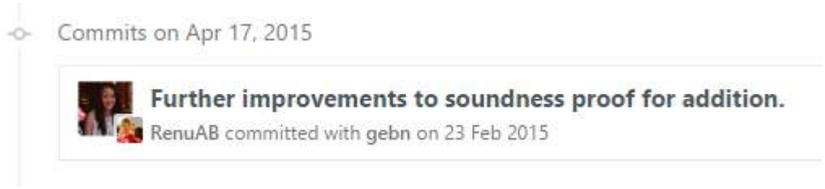
```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

You can also provide a pattern, which Git will use to search for previous authors:

```
git commit -m "msg" --author "John"
```

在这种情况下，将使用最近一次提交中作者信息包含“John”的作者信息。

在 GitHub 上，以上述任一方式提交的提交记录都会显示一个较大的作者缩略图，提交者的缩略图较小且位于前面：



第10.9节：GPG 签名提交

1. 确定你的密钥 ID

```
gpg --list-secret-keys --keyid-format LONG

/Users/davidcondrey/.gnupg/secring.gpg
------------------------------------
sec   2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]
```

您的ID是第一个斜杠后面的16位字母数字代码。

2. 在你的 git 配置中定义你的密钥 ID

```
git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
```

3. 从版本 1.7.9 开始，git commit 支持 -S 选项，用于为提交附加签名。使用此选项会提示输入你的 GPG 密码，并将签名添加到提交日志中。

```
git commit -S -m "你的提交信息"
```

第 10.10 节：提交特定文件的更改

你可以提交对特定文件所做的更改，而跳过使用 git add 进行暂存：

```
git commit file1.c file2.h
```

或者你可以先暂存文件：

```
git add file1.c file2.h
```

然后再提交它们：

```
git commit
```

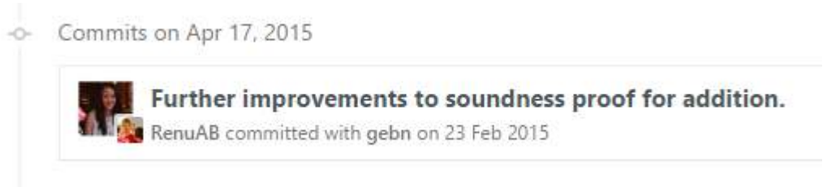
第10.11节：在特定日期提交

```
git commit -m '修复UI错误' --date 2016-07-01
```

--date参数设置作者日期（author date）。例如，该日期会出现在git log的标准输出中。

In this case, the author information from the most recent commit with an author containing "John" will be used.

On GitHub, commits made in either of the above ways will show a large author's thumbnail, with the committer's smaller and in front:



Section 10.9: GPG signing commits

1. Determine your key ID

```
gpg --list-secret-keys --keyid-format LONG

/Users/davidcondrey/.gnupg/secring.gpg
------------------------------------
sec   2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]
```

Your ID is a alphanumeric 16-digit code following the first forward-slash.

2. Define your key ID in your git config

```
git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
```

3. As of version 1.7.9, git commit accepts the -S option to attach a signature to your commits. Using this option will prompt for your GPG passphrase and will add your signature to the commit log.

```
git commit -S -m "Your commit message"
```

Section 10.10: Commiting changes in specific files

You can commit changes made to specific files and skip staging them using `git add`:

```
git commit file1.c file2.h
```

Or you can first stage the files:

```
git add file1.c file2.h
```

and commit them later:

```
git commit
```

Section 10.11: Committing at a specific date

```
git commit -m 'Fix UI bug' --date 2016-07-01
```

The `--date` parameter sets the *author date*. This date will appear in the standard output of `git log`, for example.

若要强制设置提交日期（commit date）也为该时间：

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m '修复UI错误' --date 2016-07-01
```

date参数支持GNU date所支持的灵活格式，例如：

```
git commit -m '修复UI错误' --date 昨天
git commit -m '修复UI错误' --date '3天前'
git commit -m '修复UI错误' --date '3小时前'
```

当日期未指定具体时间时，将使用当前时间，仅覆盖日期部分。

第10.12节：修改提交时间

你可以使用以下方式修改提交时间

```
git commit --amend --date="Thu Jul 28 11:30 2016 -0400"
```

甚至

```
git commit --amend --date="now"
```

第10.13节：修改提交的作者

如果你以错误的作者身份提交，可以更改它，然后进行修正

```
git config user.name "全名"
git config user.email "email@example.com"

git commit --amend --reset-author
```


To force the *commit date* too:

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Fix UI bug' --date 2016-07-01
```

The date parameter accepts the flexible formats as supported by GNU date, for example:

```
git commit -m 'Fix UI bug' --date yesterday
git commit -m 'Fix UI bug' --date '3 days ago'
git commit -m 'Fix UI bug' --date '3 hours ago'
```

When the date doesn't specify time, the current time will be used and only the date will be overridden.

Section 10.12: Amending the time of a commit

You cam amend the time of a commit using

```
git commit --amend --date="Thu Jul 28 11:30 2016 -0400"
```

or even

```
git commit --amend --date="now"
```

Section 10.13: Amending the author of a commit

If you make a commit as the wrong author, you can change it, and then amend

```
git config user.name "Full Name"
git config user.email "email@example.com"

git commit --amend --reset-author
```

第11章：别名

第11.1节：简单别名

在Git中创建别名有两种方式：

- 使用~/.gitconfig文件：

```
[alias]
ci = commit
    st = status
     co = checkout
```

- 使用命令行：

```
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.co "checkout"
```

别名创建后 - 输入：

- `git ci` 代替 `git commit`，
- `git st` 代替 `git status`，
- `git co` 代替 `git checkout`。

与常规 git 命令一样，别名可以与参数一起使用。例如：

```
git ci -m "提交信息..."
git co -b feature-42
```

第11.2节：列出 / 搜索现有别名

你可以使用 --get-regexp 来列出现有的 git 别名：

```
$ git config --get-regexp '^alias\.'
```

搜索别名

要搜索别名，请在你的.gitconfig文件的[alias]下添加以下内容：

```
aliases = !git config --list | grep ^alias\\. | cut -c 7- | grep -Ei --color \"$1\" "#"
```

然后你可以：

- `git aliases` - 显示所有别名
- `git aliases commit` - 仅显示包含“commit”的别名

第11.3节：高级别名

Git允许你在别名前加上!，以使用非git命令和完整的sh shell语法。

在你的~/.gitconfig文件中：

```
[alias]
```


Chapter 11: Aliases

Section 11.1: Simple aliases

There are two ways of creating aliases in Git:

- with the `~/.gitconfig` file:

```
[alias]
    ci = commit
    st = status
    co = checkout
```

- with the command line:

```
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.co "checkout"
```

After the alias is created - type:

- `git ci` instead of `git commit`,
- `git st` instead of `git status`,
- `git co` instead of `git checkout`.

As with regular git commands, aliases can be used beside arguments. For example:

```
git ci -m "Commit message..."
git co -b feature-42
```

Section 11.2: List / search existing aliases

You can list existing git aliases using `--get-regexp`:

```
$ git config --get-regexp '^alias\.'
```

Searching aliases

To search aliases, add the following to your `.gitconfig` under `[alias]`:

```
aliases = !git config --list | grep ^alias\\. | cut -c 7- | grep -Ei --color \"$1\" "#"
```

Then you can:

- `git aliases` - show ALL aliases
- `git aliases commit` - only aliases containing "commit"

Section 11.3: Advanced Aliases

Git lets you use non-git commands and full sh shell syntax in your aliases if you prefix them with `!`.

In your `~/.gitconfig` file:

```
[alias]
```

```
temp = !git add -A && git commit -m "Temp"
```

这些带前缀的别名支持完整的shell语法，这也意味着你可以使用shell函数来构建更复杂的别名，比如那些利用命令行参数的别名：

```
[alias]
ignore = "!f() { echo $1 >> .gitignore; }; f"
```

上述别名定义了f函数，然后用你传递给别名的任何参数运行它。所以运行`git ignore .tmp/`会将`.tmp/`添加到你的`.gitignore`文件中。

事实上，这种模式非常有用，以至于Git为你定义了$1、$2等变量，因此你甚至不必为此定义特殊函数。（但请记住，即使你通过这些变量访问，Git也会附加参数，所以你可能想在末尾添加一个虚拟命令。）

注意，以!为前缀的别名是从你的git检出根目录运行的，即使你当前目录在树的更深层。这是一个从根目录运行命令而不必显式cd过去的有用方法。

```
[alias]
ignore = "! echo $1 >> .gitignore"
```

第11.4节：临时忽略已跟踪的文件

要临时将文件标记为忽略（将文件作为参数传递给别名）- 输入：

```
unwatch = update-index --assume-unchanged
```

要重新开始跟踪文件 - 输入：

```
watch = update-index --no-assume-unchanged
```

要列出所有被临时忽略的文件 - 输入：

```
unwatched = "!git ls-files -v | grep '^[[:lower:]]'"
```

清除未关注列表 - 输入：

```
watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

别名使用示例：

```
git unwatch my_file.txt
git watch my_file.txt
git unwatched
git watchall
```

第11.5节：显示带分支图的美化日志

```
[alias]
  logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

lg =  log --graph --date-order --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
```



```
    temp = !git add -A && git commit -m "Temp"
```

The fact that full shell syntax is available in these prefixed aliases also means you can use shell functions to construct more complex aliases, such as ones which utilize command line arguments:

```
[alias]
    ignore = "!f() { echo $1 >> .gitignore; }; f"
```

The above alias defines the `f` function, then runs it with any arguments you pass to the alias. So running `git ignore .tmp/` would add `.tmp/` to your `.gitignore` file.

In fact, this pattern is so useful that Git defines $1, $2, etc. variables for you, so you don't even have to define a special function for it. (But keep in mind that Git will also append the arguments anyway, even if you access it via these variables, so you may want to add a dummy command at the end.)

Note that aliases prefixed with `!` in this way are run from the root directory of your git checkout, even if your current directory is deeper in the tree. This can be a useful way to run a command from the root without having to `cd` there explicitly.

```
[alias]
    ignore = "! echo $1 >> .gitignore"
```

Section 11.4: Temporarily ignore tracked files

To temporarily mark a file as ignored (pass file as parameter to alias) - type:

```
unwatch = update-index --assume-unchanged
```

To start tracking file again - type:

```
watch = update-index --no-assume-unchanged
```

To list all files that has been temporarily ignored - type:

```
unwatched = "!git ls-files -v | grep '^[[:lower:]]'"
```

To clear the unwatched list - type:

```
watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

Example of using the aliases:

```
git unwatch my_file.txt
git watch my_file.txt
git unwatched
git watchall
```

Section 11.5: Show pretty log with branch graph

```
[alias]
  logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

  lg =  log --graph --date-order --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
```

```
cyan)<%an>%Creset'
lgb = log --graph --date-order --branches --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
lga = log --graph --date-order --all \
   --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
```

这里是对--pretty格式中使用的选项和占位符的说明（完整列表可通过git help log 获取）

--graph - 绘制提交树

--date-order - 尽可能使用提交时间戳顺序

--first-parent - 在合并节点只跟踪第一个父节点

--branches - 显示所有本地分支（默认只显示当前分支）

--all - 显示所有本地和远程分支

%h - 提交的哈希值（缩写）

%ad - 日期戳（作者）

%an - 作者用户名

%an - 提交用户名

%C(auto) - 使用[color]部分定义的颜色

%Creset - 重置颜色

%d - --decorate（分支和标签名称）

%s - 提交信息

%ad - 作者日期（将遵循 --date 指令）（而非提交者日期）

%an - 作者姓名（可以用 %cn 表示提交者姓名）

第11.6节：查看哪些文件被你的.gitignore 配置忽略

```
[ alias ]

ignored = ! git ls-files --others --ignored --exclude-standard --directory \
            && git ls-files --others -i --exclude-standard
```

每个文件显示一行，这样你可以用 grep（仅目录）：

```
$ git ignored | grep '/$'
.yardoc/
doc/
```

或者计数：


```
cyan)<%an>%Creset'
  lgb = log --graph --date-order --branches --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
 lga = log --graph --date-order --all \
   --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
```

Here an explanation of the options and placeholder used in the `--pretty` format (exhaustive list are available with `git help log`)

--graph - draw the commit tree

--date-order - use commit timestamp order when possible

--first-parent - follow only the first parent on merge node.

--branches - show all local branches (by default, only current branch is shown)

--all - show all local and remotes branches

%h - hash value for commit (abbreviated)

%ad - Date stamp (author)

%an - Author username

%an - Commit username

%C(auto) - to use colors defined in [color] section

%Creset - to reset color

%d - --decorate (branch & tag names)

%s - commit message

%ad - author date (will follow --date directive) (and not commiter date)

%an - author name (can be %cn for commiter name)

Section 11.6: See which files are being ignored by your .gitignore configuration

```
[ alias ]

    ignored = ! git ls-files --others --ignored --exclude-standard --directory \
            && git ls-files --others -i --exclude-standard
```

Shows one line per file, so you can grep (only directories):

```
$ git ignored | grep '/$'
.yardoc/
doc/
```

Or count:

```
~$ git ignored | wc -l
199811                # 哎呀，我的主目录变得很拥挤了
```

第11.7节：在保持线性历史的同时更新代码

有时你需要保持代码提交的线性（非分支）历史。如果你在一个分支上工作了一段时间，执行常规的git pull可能会很棘手，因为这会记录一次与上游的合并。

```
[alias]
up = pull --rebase
```

这将更新你的上游源，然后将你尚未推送的工作重新应用到你拉取下来的内容之上。

使用方法：

```
git up
```

第11.8节：取消暂存已暂存的文件

通常，要使用git reset命令移除已暂存准备提交的文件，reset根据提供的参数有很多功能。要完全取消暂存所有已暂存的文件，我们可以利用git别名创建一个新别名，使用reset，这样就不需要记住为reset提供正确的参数了。

```
git config --global alias.unstage "reset --"
```

现在，每当你想取消暂存已暂存的文件时，只需输入git unstage即可。


```
~$ git ignored | wc -l
199811                # oops, my home directory is getting crowded
```

Section 11.7: Updating code while keeping a linear history

Sometimes you need to keep a linear (non-branching) history of your code commits. If you are working on a branch for a while, this can be tricky if you have to do a regular `git pull` since that will record a merge with upstream.

```
[alias]
  up = pull --rebase
```

This will update with your upstream source, then reapply any work you have not pushed on top of whatever you pulled down.

To use:

```
git up
```

Section 11.8: Unstage staged files

Normally, to remove files that are staged to be committed using the `git reset` commit, `reset` has a lot of functions depending on the arguments provided to it. To completely unstage all files staged, we can make use of git aliases to create a new alias that uses `reset` but now we do not need to remember to provide the correct arguments to `reset`.

```
git config --global alias.unstage "reset --"
```

Now, any time you want to **unstage** stages files, type `git unstage` and you are good to go.

第12章：变基（Rebasing）

参数	详情
--继续	解决合并冲突后重新启动变基过程。
--abort	中止变基操作并将 HEAD 重置到原始分支。如果在启动变基操作时提供了分支，则 HEAD 将重置到该分支。否则，HEAD 将重置到变基操作开始时的位置。
--keep-empty	保留那些相对于其父提交没有任何更改的提交。
--skip	跳过当前补丁，重新启动变基过程。
-m, --merge	使用合并策略进行变基。当使用递归（默认）合并策略时，这允许变基识别上游分支的重命名。 请注意，变基合并的工作方式是将工作分支的每个提交依次重放到上游分支之上。因此，当发生合并冲突时，被报告为“ours”的一方是迄今为止已变基的提交序列（从上游开始），而“theirs”则是工作分支。换句话说，双方角色互换。
--stat	显示自上次变基以来上游发生变化的差异统计。差异统计也受配置选项 rebase.stat 控制。

-x, --exec command 执行交互式变基，在每个提交之间暂停并执行 command

第12.1节：本地分支变基

变基会将一系列提交重新应用到另一个提交之上。

要变基一个分支，先切换到该分支，然后将其变基到另一个分支之上。

```
git checkout topic
git rebase master  # 将当前分支变基到master分支上
```

这将导致：

```
A---B---C topic
/
D---E---F---G master
```

变为：

```
A'--B'--C' topic
/
D---E---F---G master
```

这些操作可以合并为一个命令，该命令检出分支并立即对其进行变基：

```
git rebase master topic   # 将 topic 分支变基到 master 分支上
```

重要提示： 变基后，应用的提交将拥有不同的哈希值。你不应对已经推送到远程主机的提交进行变基。其后果可能是无法将本地变基后的分支通过 git push 推送到远程主机，唯一的选择可能是使用 git push --force。

第12.2节：变基：我们的和他们的，本地和远程

变基会切换“我们的”和“他们的”的含义：

Chapter 12: Rebasing

Parameter	Details
--continue	Restart the rebasing process after having resolved a merge conflict.
--abort	Abort the rebase operation and reset HEAD to the original branch. If branch was provided when the rebase operation was started, then HEAD will be reset to branch. Otherwise HEAD will be reset to where it was when the rebase operation was started.
--keep-empty	Keep the commits that do not change anything from its parents in the result.
--skip	Restart the rebasing process by skipping the current patch.
-m, --merge	Use merging strategies to rebase. When the recursive (default) merge strategy is used, this allows rebase to be aware of renames on the upstream side. Note that a rebase merge works by replaying each commit from the working branch on top of the upstream branch. Because of this, when a merge conflict happens, the side reported as ours is the so-far rebased series, starting with upstream, and theirs is the working branch. In other words, the sides are swapped.
--stat	Show a diffstat of what changed upstream since the last rebase. The diffstat is also controlled by the configuration option rebase.stat.

-x, --exec `command` Perform interactive rebase, stopping between each commit and executing `command`

Section 12.1: Local Branch Rebasing

Rebasing reapplies a series of commits on top of another commit.

To `rebase` a branch, checkout the branch and then `rebase` it on top of another branch.

```
git checkout topic
git rebase master  # rebase current branch onto master branch
```

This would cause:

```
      A---B---C topic
/
D---E---F---G master
```

To turn into:

```
              A'--B'--C' topic
/
D---E---F---G master
```

These operations can be combined into a single command that checks out the branch and immediately rebases it:

```
git rebase master topic   # rebase topic branch onto master branch
```

Important: After the rebase, the applied commits will have a different hash. You should not rebase commits you have already pushed to a remote host. A consequence may be an inability to `git push` your local rebased branch to a remote host, leaving your only option to `git push --force`.

Section 12.2: Rebase: ours and theirs, local and remote

A rebase switches the meaning of "ours" and "theirs":

```
git checkout topic
git rebase   master    # 将 topic 分支变基到 master 分支顶端
```

HEAD 指向的就是“我们的”

变基的第一步是将 HEAD 重置到 master；然后从旧分支 topic 中逐个挑选提交到新分支（旧的 topic 分支中的每个提交都会被重写，并且会有不同的哈希值）。

关于合并工具使用的术语（不要与 本地引用或远程引用 混淆）

```
=> 本地 是 master (“我们的”),
=> 远程是 topic (“他们的”)
```

这意味着合并/差异工具会将上游分支显示为本地（master：你正在rebase的基准分支），将工作分支显示为远程（topic：正在被rebase的分支）

```
+---------------------------------------+
| 本地:master |    基准    | 远程:topic |
+---------------------------------------+
|             已合并                    |
+---------------------------------------+
```

反转示意
在合并时：

```
c--c--x--x--x(*) <- 当前分支 topic（'*' = HEAD）
\
\
\--y--y--y <- 另一个要合并的分支
```

我们不改变当前分支 topic，所以我们仍然在处理之前的内容（并且从另一个分支进行合并）

```
c--c--x--x--x---------o(*)  合并，仍在分支 topic 上
\        ^         /
\     我们     /
\             /
--y--y--y--/
^
他们的
```

在变基时：

但是在变基时我们会切换阵营，因为变基的第一步是检出上游分支，将当前提交重新应用到其之上！

```
c--c--x--x--x(*) <- 当前分支 topic（'*' = HEAD）
\
\
\--y--y--y <- 上游分支
```



```
git checkout topic
git rebase   master    # rebase topic branch on top of master branch
```

Whatever HEAD's pointing to is "ours"

The first thing a rebase does is resetting the `HEAD` to `master`; before cherry-picking commits from the old branch `topic` to a new one (every commit in the former `topic` branch will be rewritten and will be identified by a different hash).

With respect to terminologies used by merge tools (not to be confused with local ref or remote ref)

```
=> local is master ("ours"),
=> remote is topic ("theirs")
```

That means a merge/diff tool will present the upstream branch as **`local`** (`master`: the branch on top of which you are rebasing), and the working branch as `remote` (`topic`: the branch being rebased)

```
+---------------------------------------+
| LOCAL:master |    BASE    | REMOTE:topic |
+---------------------------------------+
|             MERGED                    |
+---------------------------------------+
```

Inversion illustrated
On a merge:

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
\
\
\--y--y--y <- other branch to merge
```

We don't change the current branch `topic`, so what we have is still what we were working on (and we merge from another branch)

```
c--c--x--x--x---------o(*)  MERGE, still on branch topic
\       ^        /
\     ours     /
\            /
--y--y--y--/
^
theirs
```

On a rebase:

But **on a rebase** we switch sides because the first thing a rebase does is to checkout the upstream branch to replay the current commits on top of it!

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
\
\
\--y--y--y <- upstream branch
```

一次git rebase 上游分支会首先将HEAD设置到上游分支，因此相比之前的“当前”工作分支，“我们的”和“他们的”会发生切换。

```
c--c--x--x--x <- 之前的“当前”分支，新的“他们的”
\
\
\--y--y--y(*) <- 将HEAD设置到此提交，以在其上重放x的提交
^       这将成为新的“我们的”
|
上游
```

然后变基会将“他们的”提交重放到新的“我们的”topic分支上：

```
c--c..x..x..x <- 旧的“他们的”提交，现在称为“幽灵”，可通过“reflogs”访问
\
\
\--y--y--y--x'--x'--x'(*) <- 主题分支，一旦所有 x 都被重放，
^                        将主题分支指向此提交
|
上游分支
```

第12.3节：交互式变基

本示例旨在描述如何使用git rebase的交互模式。假设读者对git rebase的含义及其功能已有基本了解。

交互式变基通过以下命令启动：

```
git rebase -i
```

-i选项指的是交互模式。使用交互式变基，用户可以更改提交信息，以及重新排序、拆分和/或合并（压缩为一个）提交。

假设你想重新排列最近的三个提交。为此，你可以运行：

```
git rebase -i HEAD~3
```

执行上述指令后，您的文本编辑器中将打开一个文件，您可以在其中选择如何对提交进行变基。以本示例为例，只需更改提交的顺序，保存文件并关闭编辑器。这将启动一个按照您应用的顺序进行的变基。如果您查看git log，您将看到提交按照您指定的新顺序排列。

重写提交信息

现在，你已经决定其中一个提交信息过于模糊，想让它更具描述性。让我们使用相同的命令检查最近的三个提交。

```
git rebase -i HEAD~3
```

这次提交不会重新排序，而是会进行变基，我们将把默认的pick改为reword，用于你想要更改提交信息的提交。

当你关闭编辑器时，变基将开始，并在你想要重写的特定提交信息处暂停。

A `git rebase upstream` will first set `HEAD` to the upstream branch, hence the switch of 'ours' and 'theirs' compared to the previous "current" working branch.

```
c--c--x--x--x <- former "current" branch, new "theirs"
\
\
\--y--y--y(*) <- set HEAD to this commit, to replay x's on it
^       this will be the new "ours"
|
upstream
```

The rebase will then replay 'their' commits on the new 'our' `topic` branch:

```
c--c..x..x..x <- old "theirs" commits, now "ghosts", available through "reflogs"
\
\
\--y--y--y--x'--x'--x'(*) <- topic  once all x's are replayed,
^                        point branch topic to this commit
|
upstream branch
```

Section 12.3: Interactive Rebase

This example aims to describe how one can utilize `git rebase` in interactive mode. It is expected that one has a basic understanding of what `git rebase` is and what it does.

Interactive rebase is initiated using following command:

```
git rebase -i
```

The `-i` option refers to *interactive mode*. Using interactive rebase, the user can change commit messages, as well as reorder, split, and/or squash (combine to one) commits.

Say you want to rearrange your last three commits. To do this you can run:

```
git rebase -i HEAD~3
```

After executing the above instruction, a file will be opened in your text editor where you will be able to select how your commits will be rebased. For the purpose of this example, just change the order of your commits, save the file, and close the editor. This will initiate a rebase with the order you've applied. If you check `git log` you will see your commits in the new order you specified.

Rewording commit messages

Now, you've decided that one of the commit messages is vague and you want it to be more descriptive. Let's examine the last three commits using the same command.

```
git rebase -i HEAD~3
```

Instead of rearranging the order the commits will be rebased, this time we will change `pick`, the default, to `reword` on a commit where you would like to change the message.

When you close the editor, the rebase will initiate and it will stop at the specific commit message that you wanted to

这将允许你将提交信息更改为你想要的内容。更改完信息后，只需关闭编辑器即可继续。

更改提交内容

除了更改提交信息，你还可以修改提交所做的更改。为此，只需将
pick改为edit用于某个提交。Git 在到达该提交时会暂停，并将该提交的原始更改放入暂存区。你现在可以通过取消暂存或添加新更改来调整这些更改。

一旦暂存区包含了你想要在该提交中的所有更改，就提交这些更改。旧的提交信息会显示出来，你可以修改它以反映新的提交内容。

将单个提交拆分为多个

假设你已经提交了一个提交，但后来决定这个提交可以拆分成两个或更多的提交。
使用之前相同的命令，将pick替换为edit，然后按回车。

现在，git 会在你标记为编辑的提交处停止，并将其所有内容放入暂存区。从那时起，你可以运行git reset HEAD^将提交放入工作目录。然后，你可以以不同的顺序添加和提交文件——最终将一个提交拆分成 n 个提交。

将多个提交合并为一个

假设你已经完成了一些工作，并且有多个提交，你认为它们可以合并成一个提交。为此，你可以执行git rebase -i HEAD~3，将3替换为适当的提交数量。

这次将pick替换为squash。在变基过程中，你指示合并的提交将被合并到前一个提交上；从而将它们变成一个单一的提交。

第12.4节：变基到初始提交

自 Git 1.7.12 版本起，可以变基到根提交。根提交是仓库中创建的第一个提交，通常无法编辑。使用以下命令：

```
git rebase -i --root
```

第12.5节：配置自动暂存

Autostash 是在对本地更改使用 rebase 时非常有用的配置选项。通常情况下，你可能需要
引入上游分支的提交，但还没有准备好进行提交。

但是，如果工作目录不干净，Git 不允许开始变基。自动暂存来帮忙：

```
git config --global rebase.autostash    # 一次性配置
git rebase @{u}                         # 例如在上游分支上的变基
```

无论变基是成功完成还是中止，自动暂存都会被应用。变基是否成功并不影响自动暂存的应用。如果变基成功，基础提交因此发生了变化，那么自动暂存和新提交之间可能会产生冲突。在这种情况下，你需要在提交之前解决冲突。这与手动暂存然后应用没有区别，因此自动执行没有任何缺点。

reword. This will let you change the commit message to whichever you desire. After you've changed the message, simply close the editor to proceed.

Changing the content of a commit

Besides changing the commit message you can also adapt the changes done by the commit. To do so just change `pick` to `edit` for one commit. Git will stop when it arrives at that commit and provide the original changes of the commit in the staging area. You can now adapt those changes by unstaging them or adding new changes.

As soon as the staging area contains all changes you want in that commit, commit the changes. The old commit message will be shown and can be adapted to reflect the new commit.

Splitting a single commit into multiple

Say you've made a commit but decided at a later point this commit could be split into two or more commits instead. Using the same command as before, replace `pick` with `edit` instead and hit enter.

Now, git will stop at the commit you have marked for editing and place all of its content into the staging area. From that point you can run `git reset HEAD^` to place the commit into your working directory. Then, you can add and commit your files in a different sequence - ultimately splitting a single commit into *n* commits instead.

Squashing multiple commits into one

Say you have done some work and have multiple commits which you think could be a single commit instead. For that you can carry out `git rebase -i HEAD~3`, replacing `3` with an appropriate amount of commits.

This time replace `pick` with `squash` instead. During the rebase, the commit which you've instructed to be squashed will be squashed on top of the previous commit; turning them into a single commit instead.

Section 12.4: Rebase down to the initial commit

Since Git 1.7.12 it is possible to rebase down to the root commit. The root commit is the first commit ever made in a repository, and normally cannot be edited. Use the following command:

```
git rebase -i --root
```

Section 12.5: Configuring autostash

Autostash is a very useful configuration option when using rebase for local changes. Oftentimes, you may need to bring in commits from the upstream branch, but are not ready to commit just yet.

However, Git does not allow a rebase to start if the working directory is not clean. Autostash to the rescue:

```
git config --global rebase.autostash    # one time configuration
git rebase @{u}                         # example rebase on upstream branch
```

The autostash will be applied whenever the rebase is finished. It does not matter whether the rebase finishes successfully, or if it is aborted. Either way, the autostash will be applied. If the rebase was successful, and the base commit therefore changed, then there may be a conflict between the autostash and the new commits. In this case, you will have to resolve the conflicts before committing. This is no different than if you would have manually stashed, and then applied, so there is no downside to doing it automatically.

第12.6节：在变基过程中测试所有提交

在发起拉取请求之前，确保分支中每个提交都能成功编译并通过测试是很有用的。我们可以使用-x参数自动完成这一步。

例如：

```
git rebase -i -x make
```

将执行交互式变基，并在每个提交后暂停以执行make。如果make失败，git 会停止，给你机会修复问题并修改提交，然后再继续选择下一个提交。

第12.7节：代码审查前的变基

总结

目标是将你分散的提交重新组织成更有意义的提交，以便更容易进行代码审查。如果一次涉及太多文件的多层次更改，代码审查会更困难。如果你能将按时间顺序创建的提交重新组织成主题相关的提交，那么代码审查过程会更简单（并且可能减少代码审查过程中遗漏的错误）。

这个过于简化的例子并不是使用 git 进行更好代码审查的唯一策略。这是我使用的方法，
也是为了激励其他人思考如何让代码审查和 git 历史更简单/更好。

这也从教学角度展示了 rebase 的强大功能。

这个例子假设你了解交互式 rebase。

假设：

- 你正在基于 master 分支开发一个功能分支
- 你的功能有三个主要层次：前端、后端、数据库你在功能分支
- 上做了很多提交。每个提交同时涉及多个层次

- 你最终希望分支中只有三个提交
  - 一个包含所有前端更改
  - 一个包含所有后端更改
  - 一个包含所有数据库更改

策略：

- 我们将把按时间顺序的提交变成“主题”提交。
- 首先，将所有提交拆分成多个更小的提交——每个提交一次只包含一个主题（在我们的示例中，主题是前端、后端、数据库更改）
- 然后将我们的主题提交重新排序，并将它们“合并”为单个主题提交

示例：

```
$ git log --oneline master..
975430b 数据库添加工作：db.sql logic.rb
3702650尝试允许添加待办事项：page.html logic.rb
43b075a 初稿：page.html 和 db.sql
$ git rebase -i master
```

这将在文本编辑器中显示：

Section 12.6: Testing all commits during rebase

Before making a pull request, it is useful to make sure that compile is successful and tests are passing for each commit in the branch. We can do that automatically using `-x` parameter.

For example:

```
git rebase -i -x make
```

will perform the interactive rebase and stop after each commit to execute `make`. In case `make` fails, git will stop to give you an opportunity to fix the issues and amend the commit before proceeding with picking the next one.

Section 12.7: Rebasing before a code review

Summary

This goal is to reorganize all of your scattered commits into more meaningful commits for easier code reviews. If there are too many layers of changes across too many files at once, it is harder to do a code review. If you can reorganize your chronologically created commits into topical commits, then the code review process is easier (and possibly less bugs slip through the code review process).

This overly-simplified example is not the only strategy for using git to do better code reviews. It is the way I do it, and it's something to inspire others to consider how to make code reviews and git history easier/better.

This also pedagogically demonstrates the power of rebase in general.

This example assumes you know about interactive rebasing.

Assuming:

- you're working on a feature branch off of master
- your feature has three main layers: front-end, back-end, DB
- you have made a lot of commits while working on a feature branch. Each commit touches multiple layers at once
- you want (in the end) only three commits in your branch
  - one containing all front end changes
  - one containing all back end changes
  - one containing all DB changes

Strategy:

- we are going to change our chronological commits into "topical" commits.
- first, split all commits into multiple, smaller commits -- each containing only one topic at a time (in our example, the topics are front end, back end, DB changes)
- Then reorder our topical commits together and 'squash' them into single topical commits

Example:

```
$ git log --oneline master..
975430b db adding works: db.sql logic.rb
3702650 trying to allow adding todo items: page.html logic.rb
43b075a first draft: page.html and db.sql
$ git rebase -i master
```

This will be shown in text editor:

```
pick 43b075a 初稿：page.html 和 db.sql
pick 3702650 尝试允许添加待办事项：page.html logic.rb
pick 975430b 数据库添加工作：db.sql logic.rb
```

将其更改为：

```
e 43b075a 初稿：page.html 和 db.sql
e 3702650 尝试允许添加待办事项：page.html logic.rb
e 975430b 数据库添加功能完成：db.sql logic.rb
```

然后 git 会一次应用一个提交。每次提交后，它会显示一个提示，然后你可以执行以下操作：

```
已停止于 43b075a92a952faf999e76c4e4d7fa0f44576579... 初稿：page.html 和 db.sql
你现在可以使用以下命令修改提交

        git commit --amend

一旦你对更改满意，运行

        git rebase --continue

$ git status
rebase 进行中；基于 4975ae9
你当前正在编辑一个提交 同时 在分支 'feature' 上进行 rebase，基于 '4975ae9'。
  (使用 "git commit --amend" 来修改当前提交)
  (一旦你对更改满意，使用 "git rebase --continue" )

没有要提交的内容，工作目录干净
$ git reset HEAD^ #这会"取消提交"此提交中的所有更改。
$ git status -s
 M db.sql
 M page.html
$ git add db.sql  #现在我们将创建更小的主题提交
$ git commit -m "first draft: db.sql"
$ git add page.html
$ git commit -m "first draft: page.html"
$ git rebase --continue
```

然后你将对每个提交重复这些步骤。最后，你会得到如下结果：

```
$ git log --oneline
0309336 db adding works: logic.rb
06f81c9 db adding works: db.sql
3264de2 adding todo items: page.html
675a02b adding todo items: logic.rb
272c674 first draft: page.html
08c275d first draft: db.sql
```

现在我们再运行一次 rebase 来重新排序并合并提交：

```
$ git rebase -i master
```

这将在文本编辑器中显示：

```
pick 08c275d first draft: db.sql
pick 272c674 first draft: page.html
pick 675a02b adding todo items: logic.rb
```



```
pick 43b075a first draft: page.html and db.sql
pick 3702650 trying to allow adding todo items: page.html logic.rb
pick 975430b db adding works: db.sql logic.rb
```

Change it to this:

```
e 43b075a first draft: page.html and db.sql
e 3702650 trying to allow adding todo items: page.html logic.rb
e 975430b db adding works: db.sql logic.rb
```

Then git will apply one commit at a time. After each commit, it will show a prompt, and then you can do the following:

```
Stopped at 43b075a92a952faf999e76c4e4d7fa0f44576579... first draft: page.html and db.sql
You can amend the commit now, with

        git commit --amend

Once you are satisfied with your changes, run

        git rebase --continue

$ git status
rebase in progress; onto 4975ae9
You are currently editing a commit while rebasing branch 'feature' on '4975ae9'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
$ git reset HEAD^ #This 'uncommits' all the changes in this commit.
$ git status -s
 M db.sql
 M page.html
$ git add db.sql  #now we will create the smaller topical commits
$ git commit -m "first draft: db.sql"
$ git add page.html
$ git commit -m "first draft: page.html"
$ git rebase --continue
```

Then you will repeat those steps for every commit. In the end, you have this:

```
$ git log --oneline
0309336 db adding works: logic.rb
06f81c9 db adding works: db.sql
3264de2 adding todo items: page.html
675a02b adding todo items: logic.rb
272c674 first draft: page.html
08c275d first draft: db.sql
```

Now we run rebase one more time to reorder and squash:

```
$ git rebase -i master
```

This will be shown in text editor:

```
pick 08c275d first draft: db.sql
pick 272c674 first draft: page.html
pick 675a02b adding todo items: logic.rb
```

```
pick 3264de2 添加待办事项：page.html
pick 06f81c9 数据库添加工作：db.sql
pick 0309336 数据库添加工作：logic.rb
```

将其更改为：

```
pick 08c275d 初稿：db.sql
s 06f81c9 数据库添加工作：db.sql
pick 675a02b 添加待办事项：logic.rb
s 0309336 数据库添加工作：logic.rb
pick 272c674 初稿：page.html
s  3264de2 添加待办事项：page.html
```

注意：确保你告诉 git rebase 按照它们被提交的时间顺序应用/合并较小的专题提交。否则你可能会遇到虚假的、不必要的合并冲突需要处理。

当交互式变基完成后，你会得到如下结果：

```
$ git log --oneline master..
74bdd5f 添加待办事项：GUI 层
e8d8f7e 添加待办事项：业务逻辑层
121c578 添加待办事项：数据库层
```

总结

你现在已经将按时间顺序的提交变基成了专题提交。在实际工作中，你可能不需要每次都这样做，但当你想要或需要这样做时，现在你可以了。希望你也学到了更多关于 git rebase 的知识。

第12.8节：中止交互式变基

你已经开始了一个交互式变基。在选择提交的编辑器中，你发现出现了问题（例如缺少某个提交，或者选择了错误的变基目标），你想要中止变基。

为此，只需删除所有提交和操作（即所有不以#符号开头的行），变基操作将被中止！

编辑器中的帮助文本实际上提供了这个提示：

```
# 将 36d15de..612f2f7 变基到 36d15de（3 条命令）
#
# 命令：
# p, pick = 使用提交
# r, reword = 使用提交，但编辑提交信息
# e, edit = 使用提交，但暂停以便修改
# s, squash = 使用提交，但合并到前一个提交
# f, fixup = 类似“squash”，但丢弃该提交的日志信息
# x, exec = 使用 shell 运行命令（该行剩余部分）
#
# 这些行可以重新排序；它们从上到下依次执行。
#
# 如果你删除这里的一行，该提交将会丢失。
#
# 但是，如果你删除所有内容，变基将会中止。
#          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# 注意空提交已被注释掉
```



```
pick 3264de2 adding todo items: page.html
pick 06f81c9 db adding works: db.sql
pick 0309336 db adding works: logic.rb
```

Change it to this:

```
pick 08c275d first draft: db.sql
s 06f81c9 db adding works: db.sql
pick 675a02b adding todo items: logic.rb
s 0309336 db adding works: logic.rb
pick 272c674 first draft: page.html
s  3264de2 adding todo items: page.html
```

NOTICE: make sure that you tell git rebase to apply/squash the smaller topical commits *in the order they were chronologically commited*. Otherwise you might have false, needless merge conflicts to deal with.

When that interactive rebase is all said and done, you get this:

```
$ git log --oneline master..
74bdd5f adding todos: GUI layer
e8d8f7e adding todos: business logic layer
121c578 adding todos: DB layer
```

Recap

You have now rebased your chronological commits into topical commits. In real life, you may not need to do this every single time, but when you do want or need to do this, now you can. Plus, hopefully you learned more about git rebase.

Section 12.8: Aborting an Interactive Rebase

You have started an interactive rebase. In the editor where you pick your commits, you decide that something is going wrong (for example a commit is missing, or you chose the wrong rebase destination), and you want to abort the rebase.

To do this, simply delete all commits and actions (i.e. all lines not starting with the # sign) and the rebase will be aborted!

The help text in the editor actually provides this hint:

```
# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Note that empty commits are commented out
```

第12.9节：设置git-pull以自动执行变基而非合并

如果你的团队采用基于变基的工作流程，设置git使每个新创建的分支在执行git pull时进行变基操作而非合并操作，可能会更有利。

要设置每个新分支自动变基，请将以下内容添加到你的.gitconfig或.git/config中：

```
[branch]
autosetuprebase = always
```

命令行：`git config [--global] branch.autosetuprebase always`

或者，你也可以设置`git pull`命令始终表现得像传入了`--rebase`选项：

```
[pull]
rebase = true
```

命令行：`git config [--global] pull.rebase true`

第12.10节：变基后的推送

有时你需要通过 rebase 重写历史，但 `git push` 会因为你重写了历史而报错。

这可以通过 git push --force 解决，但建议考虑使用 git push --force-with-lease，表示如果本地远程跟踪分支与远程分支不同，例如有人在你上次拉取后推送了代码，则推送会失败。这可以避免无意中覆盖他人最近的推送。

注意：git push --force —— 甚至 --force-with-lease —— 都是危险的命令，因为它会重写分支的历史。如果其他人在强制推送之前已经拉取了该分支，他/她的 gitpull 或 git fetch 会出现错误，因为本地历史和远程历史已经分叉。这可能导致该用户遇到意外错误。虽然通过查看 reflogs 可以恢复其他用户的工作，但这会浪费大量时间。如果必须对有其他贡献者的分支进行强制推送，尽量与他们协调，避免他们遇到错误。

Section 12.9: Setup git-pull for automatically perform a rebase instead of a merge

If your team is following a rebase-based workflow, it may be a advantageous to setup git so that each newly created branch will perform a rebase operation, instead of a merge operation, during a `git pull`.

To setup every *new* branch to automatically rebase, add the following to your `.gitconfig` or `.git/config`:

```
[branch]
autosetuprebase = always
```

Command line: `git config [--global] branch.autosetuprebase always`

Alternatively, you can setup the `git pull` command to always behave as if the option `--rebase` was passed:

```
[pull]
rebase = true
```

Command line: `git config [--global] pull.rebase true`

Section 12.10: Pushing after a rebase

Sometimes you need rewrite history with a rebase, but `git push` complains about doing so because you rewrote history.

This can be solved with a `git push --force`, but consider `git push --force-with-lease`, indicating that you want the push to fail if the local remote-tracking branch differs from the branch on the remote, e.g., someone else pushed to the remote after the last fetch. This avoids inadvertently overwriting someone else's recent push.

Note: `git push --force` - and even `--force-with-lease` for that matter - can be a dangerous command because it rewrites the history of the branch. If another person had pulled the branch before the forced push, his/her `git pull` or `git fetch` will have errors because the local history and the remote history are diverged. This may cause the person to have unexpected errors. With enough looking at the reflogs the other user's work can be recovered, but it can lead to a lot of wasted time. If you must do a forced push to a branch with other contributors, try to coordinate with them so that they do not have to deal with errors.

第13章：配置

参数	详情
`--system`	编辑系统范围的配置文件，适用于所有用户（在 Linux 上，该文件位于 `$(prefix)/etc/gitconfig`）
`--global`	编辑全局配置文件，适用于你操作的所有仓库（在 Linux 上，该文件位于 ~/.gitconfig
`--local`	编辑仓库特定的配置文件，位于你仓库中的 .git/config；这是默认设置

第13.1节：设置使用哪个编辑器

有几种方法可以设置用于提交、变基等操作的编辑器。

- 更改core.editor配置设置。

```
$ git config --global core.editor nano
```

- 设置GIT_EDITOR环境变量。

  针对单个命令：

```
$ GIT_EDITOR=nano git commit
```

  或者针对终端中运行的所有命令。 注意： 这只在关闭终端之前有效。

```
$ export GIT_EDITOR=nano
```

- 要更改所有终端程序（不仅仅是Git）的编辑器，请设置VISUAL或EDITOR环境变量。（参见视觉与编辑器。）

```
$ export EDITOR=nano
```

  注意： 如上所述，这仅适用于当前终端；您的 shell 通常会有一个配置文件，允许您永久设置它。（例如，在 **bash** 中，将上述行添加到您的 ~/.bashrc 或 ~/.bash_profile.）

一些文本编辑器（主要是图形界面编辑器）一次只能运行一个实例，且如果你已经打开了一个实例，通常会自动退出。如果你的文本编辑器是这种情况，Git 会显示消息由于提交信息为空，终止提交。且不会先让你编辑提交信息。如果遇到这种情况，请查阅你的文本编辑器文档，看看它是否有--wait标志（或类似功能），可以让编辑器暂停，直到文档关闭。

第13.2节：自动更正拼写错误

```
git config --global help.autocorrect 17
```

这使得 git 中启用自动更正功能，可以原谅你的一些小错误（例如输入git stats而不是git status）。您提供给 help.autocorrect 的参数决定了系统在自动应用自动更正命令之前应等待多长时间，单位为十分之一秒。在上述命令中，17 表示 git

Chapter 13: Configuration

Parameter	Details
`--system`	Edits the system-wide configuration file, which is used for every user (on Linux, this file is located at `$(prefix)/etc/gitconfig`)
`--global`	Edits the global configuration file, which is used for every repository you work on (on Linux, this file is located at `~/.gitconfig`
`--local`	Edits the respository-specific configuration file, which is located at `.git/config` in your repository; this is the default setting

Section 13.1: Setting which editor to use

There are several ways to set which editor to use for committing, rebasing, etc.

- Change the `core.editor` configuration setting.

```
$ git config --global core.editor nano
```

- Set the `GIT_EDITOR` environment variable.

  For one command:

```
$ GIT_EDITOR=nano git commit
```

  Or for all commands run in a terminal. **Note:** This only applies until you close the terminal.

```
$ export GIT_EDITOR=nano
```

- To change the editor for *all* terminal programs, not just Git, set the `VISUAL` or `EDITOR` environment variable. (See VISUAL vs EDITOR.)

```
$ export EDITOR=nano
```

  Note: As above, this only applies to the current terminal; your shell will usually have a configuration file to allow you to set it permanently. (On **bash**, for example, add the above line to your `~/.bashrc` or `~/.bash_profile`.)

Some text editors (mostly GUI ones) will only run one instance at a time, and generally quit if you already have an instance of them open. If this is the case for your text editor, Git will print the message `Aborting commit due to empty commit message.` without allowing you to edit the commit message first. If this happens to you, consult your text editor's documentation to see if it has a `--wait` flag (or similar) that will make it pause until the document is closed.

Section 13.2: Auto correct typos

```
git config --global help.autocorrect 17
```

This enables autocorrect in git and will forgive you for your minor mistakes (e.g. **`git stats`** instead of **`git status`**). The parameter you supply to `help.autocorrect` determines how long the system should wait, in tenths of a second, before automatically applying the autocorrected command. In the command above, 17 means that git

应等待1.7秒后再应用自动更正的命令。

但是，更大的错误将被视为缺失的命令，因此输入类似git testingit的内容会导致 testingit 不是一个 git 命令。

第13.3节：列出并编辑当前配置

Git 配置允许你自定义 git 的工作方式。它通常用于设置你的姓名和邮箱或喜欢的编辑器
或合并的方式。

查看当前配置。

```
$ git config --list
...
core.editor=vim
credential.helper=osxkeychain
...
```

编辑配置：

```
$ git config <key> <value>
$ git config core.ignorecase true
```

如果你希望更改对所有仓库生效，使用 --global

```
$ git config --global user.name "你的名字"
$ git config --global user.email "你的邮箱"
$ git config --global core.editor vi
```

你可以再次列出以查看更改。

第13.4节：用户名和电子邮件地址

安装 Git 后，您首先应该做的是设置您的用户名和电子邮件地址。在终端中输入：

```
git config --global user.name "Mr. Bean"
git config --global user.email mrbean@example.com
```

- git config 是获取或设置选项的命令--global 表示
- 将编辑特定于您用户账户的配置文件user.name 和 user.email 是配置变量的键；user
- 是配置文件的部分。name 和 email 是变量的名称。
- "Mr. Bean" 和 mrbean@example.com 是您存储在这两个变量中的值。注意 "Mr. Bean" 周围的引号，因为您存储的值包含空格，所以必须加引号。

第13.5节：多个用户名和电子邮件地址

自 Git 2.13 起，可以通过使用文件夹过滤器配置多个用户名和电子邮件地址。

Windows 示例：
.gitconfig

编辑：`git config --global -e`

添加：

should wait 1.7 seconds before applying the autocorrected command.

However, bigger mistakes will be considered as missing commands, so typing something like `git testingit` would result in `testingit is not a git command.`

Section 13.3: List and edit the current configuration

Git config allows you to customize how git works. It is commonly used to set your name and email or favorite editor or how merges should be done.

To see the current configuration.

```
$ git config --list
...
core.editor=vim
credential.helper=osxkeychain
...
```

To edit the config:

```
$ git config <key> <value>
$ git config core.ignorecase true
```

If you intend the change to be true for all your repositories, use `--global`

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email"
$ git config --global core.editor vi
```

You can list again to see your changes.

Section 13.4: Username and email address

Right after you install Git, the first thing you should do is set your username and email address. From a shell, type:

```
git config --global user.name "Mr. Bean"
git config --global user.email mrbean@example.com
```

- `git config` is the command to get or set options
- `--global` means that the configuration file specific to your user account will be edited
- `user.name` and `user.email` are the keys for the configuration variables; `user` is the section of the configuration file. `name` and `email` are the names of the variables.
- `"Mr. Bean"` and `mrbean@example.com` are the values that you're storing in the two variables. Note the quotes around `"Mr. Bean"`, which are required because the value you are storing contains a space.

Section 13.5: Multiple usernames and email address

Since Git 2.13, multiple usernames and email addresses could be configured by using a folder filter.

Example for Windows:
.gitconfig

Edit: `git config --global -e`

Add:

```
[includeIf "gitdir:D:/work"]
  path = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]
  path = .gitconfig-opensource.config
```

说明

- 顺序是有依赖的，最后一个匹配的规则“生效”。
- 末尾需要/，例如"gitdir:D:/work"不起作用。
- 必须有gitdir:前缀。

.gitconfig-work.config

与.gitconfig同目录下的文件

```
[user]
name = Money
  email = work@somewhere.com
```

.gitconfig-opensource.config

与.gitconfig同目录下的文件

```
[user]
姓名 = Nice
email = cool@opensource.stuff
```

Linux 示例

```
[includeIf "gitdir:~/work/"]
  path = .gitconfig-work
[includeIf "gitdir:~/opensource/"]
  path = .gitconfig-opensource
```

Windows 部分下的文件内容和注释。

第13.6节：多个 git 配置

你最多可以有5个 git 配置来源：

- 6 个文件：
  - **%ALLUSERSPROFILE%\Git\Config**（仅限 Windows）
  - （系统）**<git>/etc/gitconfig**，其中 **<git>** 是 git 安装路径。
    （在 Windows 上，是 **<git>\mingw64\etc\gitconfig**）
  - （系统）**$XDG_CONFIG_HOME/git/config**（仅限 Linux/Mac）
  - （全局）**~/**.gitconfig（Windows：%USERPROFILE%\.gitconfig）
  - （本地）.git/config（在 git 仓库内 $GIT_DIR）
  - 一个专用文件（使用git config -f），例如用于修改子模块的配置：git config -f .gitmodules ...
- **使用命令行中的git -c**：git -c core.autocrlf=false fetch 会覆盖*任何*其他core.autocrlf为false，仅针对该fetch命令。

顺序很重要：在一个来源中设置的任何配置都可以被其下方列出的来源覆盖。

git config --system/global/local 是列出这三个来源的命令，但只有 git config -l 会列出*所有解析后的*配置。
“解析后”意味着它只列出最终被覆盖的配置值。


```
[includeIf "gitdir:D:/work"]
  path = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]
  path = .gitconfig-opensource.config
```

Notes

- The order is depended, the last one who matches "wins".
- the / at the end is needed - e.g. "gitdir:D:/work" won't work.
- the gitdir: prefix is required.

.gitconfig-work.config

File in the same directory as *.gitconfig*

```
[user]
  name = Money
  email = work@somewhere.com
```

.gitconfig-opensource.config

File in the same directory as *.gitconfig*

```
[user]
  name = Nice
  email = cool@opensource.stuff
```

Example for Linux

```
[includeIf "gitdir:~/work/"]
  path = .gitconfig-work
[includeIf "gitdir:~/opensource/"]
  path = .gitconfig-opensource
```

The file content and notes under section Windows.

Section 13.6: Multiple git configurations

You have up to 5 sources for git configuration:

- 6 files:
  - **%ALLUSERSPROFILE%\Git\Config** (Windows only)
  - (system) **<git>/etc/gitconfig**, with **<git>** being the git installation path.
    (on Windows, it is **<git>\mingw64\etc\gitconfig**)
  - (system) **$XDG_CONFIG_HOME/git/config** (Linux/Mac only)
  - (global) **~/**.gitconfig (Windows: %USERPROFILE%\.gitconfig)
  - (local) .git/config (within a git repo $GIT_DIR)
  - a **dedicated file** (with **git config -f**), used for instance to modify the config of submodules: **git config -f** .gitmodules ...
- **the command line with git -c**: git -c core.autocrlf=false fetch would override *any* other core.autocrlf to **false**, *just* for that fetch command.

The order is important: any config set in one source can be overridden by a source listed below it.

git config --system/global/local is the command to list 3 of those sources, but only git config -l would list *all resolved* configs.
"resolved" means it lists only the final overridden config value.

自 git 2.8 起，如果你想查看哪个配置来自哪个文件，可以输入：

```
git config --list --show-origin
```

第13.7节：配置行结束符

描述

当与使用不同操作系统（OS）的团队合作时，有时在处理行结束符时可能会遇到问题。

微软 Windows

在使用微软Windows操作系统（OS）时，行结束符通常是回车符加换行符（CR+LF）的形式。打开一个使用Unix机器（如Linux或OSX）编辑过的文件时，可能会出现问题，导致文本看起来根本没有行结束符。这是因为Unix系统只使用换行符（LF）作为行结束符。

为了解决这个问题，你可以运行以下指令

```
git config --global core.autocrlf=true
```

在**检出（checkout）**时，该指令将确保行结束符按照微软Windows操作系统的要求配置（LF -> CR+LF）

基于Unix的系统（Linux/OSX）

同样，当基于Unix的操作系统用户尝试读取在微软Windows操作系统上编辑的文件时，可能会出现问题。为防止任何意外问题，请运行

```
git config --global core.autocrlf=input
```

在提交（commit）时，这将把行结束符从CR+LF转换为LF

第13.8节：仅针对单个命令的配置

你可以使用-c <name>=<value>来仅为单个命令添加配置。

要以其他用户身份提交，而无需更改 .gitconfig 中的设置：

```
git -c user.email = mail@example commit -m "some message"
```

注意：在该示例中，您无需同时指定 user.name 和 user.email，git 会从之前的提交中补全缺失的信息。

第13.9节：设置代理

如果您处于代理服务器后面，必须告诉 git：

```
git config --global http.proxy http://my.proxy.com:portnumber
```

如果您不再使用代理：

Since git 2.8, if you want to see which config comes from which file, you type:

```
git config --list --show-origin
```

Section 13.7: Configuring line endings

Description

When working with a team who uses different operating systems (OS) across the project, sometimes you may run into trouble when dealing with line endings.

Microsoft Windows

When working on Microsoft Windows operating system (OS), the line endings are normally of form - carriage return + line feed (CR+LF). Opening a file which has been edited using Unix machine such as Linux or OSX may cause trouble, making it seem that text has no line endings at all. This is due to the fact that Unix systems apply different line-endings of form line feeds (LF) only.

In order to fix this you can run following instruction

```
git config --global core.autocrlf=true
```

On **checkout**, This instruction will ensure line-endings are configured in accordance with Microsoft Windows OS (LF -> CR+LF)

Unix Based (Linux/OSX)

Similarly, there might be issues when the user on Unix based OS tries to read files which have been edited on Microsoft Windows OS. In order to prevent any unexpected issues run

```
git config --global core.autocrlf=input
```

On **commit**, this will change line-endings from CR+LF -> +LF

Section 13.8: configuration for one command only

you can use `-c <name>=<value>` to add a configuration only for one command.

To commit as an other user without having to change your settings in .gitconfig :

```
git -c user.email = mail@example commit -m "some message"
```

Note: for that example you don't need to precise both `user.name` and `user.email`, git will complete the missing information from the previous commits.

Section 13.9: Setup a proxy

If you are behind a proxy, you have to tell git about it:

```
git config --global http.proxy http://my.proxy.com:portnumber
```

If you are no more behind a proxy:

```
git config --global --unset http.proxy
```



```
git config --global --unset http.proxy
```

第14章：分支管理

参数	详情
-d，--delete	删除分支。该分支必须已完全合并到其上游分支，或者如果未使用--track或--set-upstream设置上游，则合并到HEAD
-D	--delete --force的快捷方式
-m，--move	移动/重命名分支及相应的引用日志
-M	--move --force的快捷方式
-r，--remotes	列出或删除（如果与-d一起使用）远程跟踪分支
-a，--all	列出远程跟踪分支和本地分支
--list	激活列表模式。 `git branch <pattern>` 会尝试创建分支，使用 `git branch --list <pattern>` 来列出匹配的分支
--set-upstream	如果指定的分支尚不存在或已使用 --force，则行为与 --track 完全相同。否则设置配置，类似于创建分支时的 --track，但不会更改分支指向的位置

第14.1节：创建和切换新分支

要创建新分支，同时保持在当前分支上，使用：

```
git branch <name>
```

通常，分支名称不得包含空格，并且需遵守 here 中列出的其他规范。要切换到现有分支：

```
git checkout <name>
```

要创建一个新分支并切换到该分支：

```
git checkout -b <name>
```

要在当前分支的最后一次提交（也称为 HEAD）以外的某个点创建分支，可以使用以下任一命令：

```
git branch <name> [<start-point>]
git checkout -b <name> [<start-point>]
```

<start-point> 可以是 git 已知的任何 revision（例如另一个分支名、提交 SHA，或符号引用如 HEAD 或标签名）：

```
git checkout -b <name> some_other_branch
git checkout -b <name> af295
git checkout -b <name> HEAD~5
git checkout -b <name> v1.0.5
```

要从远程分支创建分支（默认的 <remote_name> 是 origin）：

```
git branch <name> <remote_name>/<branch_name>
git checkout -b <name> <remote_name>/<branch_name>
```

如果某个分支名只存在于一个远程仓库，你可以直接使用

Chapter 14: Branching

Parameter	Details
-d, --delete	Delete a branch. The branch must be fully merged in its upstream branch, or in HEAD if no upstream was set with `--track` or `--set-upstream`
-D	Shortcut for `--delete --force`
-m, --move	Move/rename a branch and the corresponding reflog
-M	Shortcut for `--move --force`
-r, --remotes	List or delete (if used with -d) the remote-tracking branches
-a, --all	List both remote-tracking branches and local branches
--list	Activate the list mode. `git branch <pattern>` would try to create a branch, use `git branch --list <pattern>` to list matching branches
--set-upstream	If specified branch does not exist yet or if `--force` has been given, acts exactly like `--track`. Otherwise sets up configuration like --track would when creating the branch, except that where branch points to is not changed

Section 14.1: Creating and checking out new branches

To create a new branch, while staying on the current branch, use:

```
git branch <name>
```

Generally, the branch name must not contain spaces and is subject to other specifications listed here. To switch to an existing branch :

```
git checkout <name>
```

To create a new branch and switch to it:

```
git checkout -b <name>
```

To create a branch at a point other than the last commit of the current branch (also known as HEAD), use either of these commands:

```
git branch <name> [<start-point>]
git checkout -b <name> [<start-point>]
```

The **<start-point>** can be any revision known to git (e.g. another branch name, commit SHA, or a symbolic reference such as HEAD or a tag name):

```
git checkout -b <name> some_other_branch
git checkout -b <name> af295
git checkout -b <name> HEAD~5
git checkout -b <name> v1.0.5
```

To create a branch from a remote branch (the default **<remote_name>** is origin):

```
git branch <name> <remote_name>/<branch_name>
git checkout -b <name> <remote_name>/<branch_name>
```

If a given branch name is only found on one remote, you can simply use

```
git checkout -b <branch_name>
```

等同于

```
git checkout -b <branch_name> <remote_name>/<branch_name>
```

有时你可能需要将最近的几个提交移动到一个新分支。这可以通过
创建分支并“回滚”来实现，操作如下：

```
git branch <new_name>
git reset --hard HEAD~2 # 回退2个提交，未提交的工作将丢失。
git checkout <new_name>
```

下面是该技巧的示意说明：

```
初始状态        执行 git branch <new_name> 后     执行 git reset --hard HEAD~2
                                 newBranch                          newBranch
                                     ↓                                  ↓
A-B-C-D-E (HEAD)          A-B-C-D-E (HEAD)                 A-B-C-D-E (HEAD)
        ↑                         ↑                                ↑
     master                    master                           master
```

第14.2节：列出分支

Git 提供了多条命令用于列出分支。所有命令都使用git branch功能，根据命令行中使用的选项，提供某些分支的列表。Git 会在可能的情况下，用星号标记当前选中的分支。

目标	命令
列出本地分支	`git branch`
详细列出本地分支	`git branch -v`
列出远程和本地分支	`git branch -a 或者 git branch --all`
列出远程和本地分支（详细）	`git branch -av`
列出远程分支	`git branch -r`
列出带有最新提交的远程分支	`git branch -rv`
列出已合并的分支	`git branch --merged`
列出未合并的分支	`git branch --no-merged`
列出包含指定提交的分支	`git branch --contains [<commit>]`

注意:

- 向 -v 添加额外的 v，例如 $ git branch -avv 或 $ git branch -vv 也会打印上游分支的名称。
- 以红色显示的分支是远程分支

第14.3节：删除远程分支

要删除 origin 远程仓库上的分支，Git 1.5.0及更高版本可以使用

```
git push origin :<branchName>
```

从Git 1.7.0版本开始，可以使用以下命令删除远程分支


```
git checkout -b <branch_name>
```

which is equivalent to

```
git checkout -b <branch_name> <remote_name>/<branch_name>
```

Sometimes you may need to move several of your recent commits to a new branch. This can be achieved by branching and "rolling back", like so:

```
git branch <new_name>
git reset --hard HEAD~2 # Go back 2 commits, you will lose uncommitted work.
git checkout <new_name>
```

Here is an illustrative explanation of this technique:

```
 Initial state       After git branch <new_name>     After git reset --hard HEAD~2
                              newBranch                          newBranch
                                  ↓                                  ↓
A-B-C-D-E (HEAD)          A-B-C-D-E (HEAD)                 A-B-C-D-E (HEAD)
        ↑                         ↑                                ↑
     master                    master                           master
```

Section 14.2: Listing branches

Git provides multiple commands for listing branches. All commands use the function of `git branch`, which will provide a list of a certain branches, depending on which options are put on the command line. Git will if possible, indicate the currently selected branch with a star next to it.

Goal	Command
List local branches	`git branch`
List local branches verbose	`git branch -v`
List remote and local branches	`git branch -a` OR `git branch --all`
List remote and local branches (verbose)	`git branch -av`
List remote branches	`git branch -r`
List remote branches with latest commit	`git branch -rv`
List merged branches	`git branch --merged`
List unmerged branches	`git branch --no-merged`
List branches containing commit	`git branch --contains [<commit>]`

Notes:

- Adding an additional `v` to `-v` e.g. `$ git branch -avv` or `$ git branch -vv` will print the name of the upstream branch as well.
- Branches shown in red color are remote branches

Section 14.3: Delete a remote branch

To delete a branch on the `origin` remote repository, you can use for Git version 1.5.0 and newer

```
git push origin :<branchName>
```

and as of Git version 1.7.0, you can delete a remote branch using

```
git push origin --delete <branchName>
```

删除本地远程跟踪分支：

```
git branch --delete --remotes <remote>/<branch>
git branch -dr <remote>/<branch> # 更简短

git fetch <remote> --prune # 删除多个过时的跟踪分支
git fetch <remote> -p      # 更简短
```

在本地删除分支。注意，如果该分支有未合并的更改，则不会删除该分支：

```
git branch -d <branchName>
```

要删除一个分支，即使它有未合并的更改：

```
git branch -D <branchName>
```

第14.4节：快速切换到上一个分支

你可以使用以下命令快速切换到上一个分支

```
git checkout -
```

第14.5节：检出跟踪远程分支的新分支

有三种方法可以创建一个新的分支feature，该分支跟踪远程分支origin/feature：

- `git checkout --track -b feature origin/feature`,
- `git checkout -t origin/feature`,
- `git checkout feature` - 假设本地没有feature分支，并且只有一个远程仓库有feature分支。

要设置上游以跟踪远程分支，请输入：

- `git branch --set-upstream-to=<remote>/<branch> <branch>`
- `git branch -u <remote>/<branch> <branch>`

说明：

- **<remote>** 可以是：`origin`、`develop` 或用户创建的分支，
- **<branch>** 是用户在远程跟踪的分支。

要验证本地分支跟踪了哪些远程分支：

- `git branch -vv`

第14.6节：在本地删除分支

```
$ git branch -d dev
```

如果名为 dev 的分支的更改已合并到另一个分支且不会丢失，则删除该分支。如果 dev 分支包含尚未合并且会丢失的更改，git branch -d 将失败：


```
git push origin --delete <branchName>
```

To delete a local remote-tracking branch:

```
git branch --delete --remotes <remote>/<branch>
git branch -dr <remote>/<branch> # Shorter

git fetch <remote> --prune # Delete multiple obsolete tracking branches
git fetch <remote> -p      # Shorter
```

To delete a branch locally. Note that this will not delete the branch if it has any unmerged changes:

```
git branch -d <branchName>
```

To delete a branch, even if it has unmerged changes:

```
git branch -D <branchName>
```

Section 14.4: Quick switch to the previous branch

You can quickly switch to the previous branch using

```
git checkout -
```

Section 14.5: Check out a new branch tracking a remote branch

There are three ways of creating a new branch `feature` which tracks the remote branch `origin/feature`:

- `git checkout --track -b feature origin/feature`,
- `git checkout -t origin/feature`,
- `git checkout feature` - assuming that there is no local `feature` branch and there is only one remote with the `feature` branch.

To set upstream to track the remote branch - type:

- `git branch --set-upstream-to=<remote>/<branch> <branch>`
- `git branch -u <remote>/<branch> <branch>`

where:

- **<remote>** can be: `origin`, `develop` or the one created by user,
- **<branch>** is user's branch to track on remote.

To verify which remote branches your local branches are tracking:

- `git branch -vv`

Section 14.6: Delete a branch locally

```
$ git branch -d dev
```

Deletes the branch named `dev` *if* its changes are merged with another branch and will not be lost. If the `dev` branch does contain changes that have not yet been merged that would be lost, `git branch -d` will fail:

```
$ git branch -d dev
错误：分支 'dev' 尚未完全合并。
如果确定要删除它，请运行 'git branch -D dev'。
```

根据警告信息，您可以使用-D标志强制删除该分支（并且会丢失该分支中所有未合并的更改）：

```
$ git branch -D dev
```

第14.7节：创建一个孤立分支（即没有父提交的分支）

```
git checkout --orphan new-orphan-branch
```

> 在此新分支上进行的第一次提交将没有父提交，它将成为一个全新历史的根节点，完全与其他所有分支和提交断开连接。

source

第14.8节：重命名分支

重命名您当前检出的分支：

```
git branch -m new_branch_name
```

重命名其他分支：

```
git branch -m branch_you_want_to_rename new_branch_name
```

第14.9节：在分支中搜索

列出包含特定提交或标签的本地分支

```
git branch --contains <commit>
```

列出包含特定提交或标签的本地和远程分支

```
git branch -a --contains <commit>
```

第14.10节：推送分支到远程

用于将本地分支上的提交推送到远程仓库。

git push 命令接受两个参数：

- 远程名称，例如 `origin`
- 分支名称，例如 `master`

例如：

```
git push  <REMOTENAME> <BRANCHNAME>
```



```
$ git branch -d dev
error: The branch 'dev' is not fully merged.
If you are sure you want to delete it, run 'git branch -D dev'.
```

Per the warning message, you can force delete the branch (and lose any unmerged changes in that branch) by using the `-D` flag:

```
$ git branch -D dev
```

Section 14.7: Create an orphan branch (i.e. branch with no parent commit)

```
git checkout --orphan new-orphan-branch
```

> The first commit made on this new branch will have no parents and it will be the root of a new history totally disconnected from all the other branches and commits.

source

Section 14.8: Rename a branch

Rename the branch you have checked out:

```
git branch -m new_branch_name
```

Rename another branch:

```
git branch -m branch_you_want_to_rename new_branch_name
```

Section 14.9: Searching in branches

To list local branches that contain a specific commit or tag

```
git branch --contains <commit>
```

To list local and remote branches that contain a specific commit or tag

```
git branch -a --contains <commit>
```

Section 14.10: Push branch to remote

Use to push commits made on your local branch to a remote repository.

The `git push` command takes two arguments:

- A remote name, for example, `origin`
- A branch name, for example, `master`

For example:

```
git push  <REMOTENAME> <BRANCHNAME>
```

例如，你通常运行git push origin master将本地更改推送到你的在线仓库。

使用-u（--set-upstream的缩写）将在推送时设置跟踪信息。

```
git push -u <REMOTENAME> <BRANCHNAME>
```

默认情况下，git会将本地分支推送到同名的远程分支。例如，如果你有一个名为new-feature的本地分支，推送该本地分支时也会创建一个名为new-feature的远程分支。如果你想为远程分支使用不同的名称，可以在本地分支名称后面加上远程名称，用:分隔：

```
git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

第14.11节：将当前分支HEAD移动到任意提交

分支只是指向某个提交的指针，因此你可以自由移动它。要使分支指向提交aabbcc，执行命令

```
git reset --hard aabbcc
```

请注意，这将覆盖你分支当前的提交及其整个历史。执行此命令可能会丢失一些工作。如果发生这种情况，你可以使用reflog来恢复丢失的提交。建议在新分支上执行此命令，而不是当前分支。

然而，当进行变基或其他大规模历史修改时，此命令特别有用。

As an example, you usually run `git push origin master` to push your local changes to your online repository.

Using `-u` (short for `--set-upstream`) will set up the tracking information during the push.

```
git push -u <REMOTENAME> <BRANCHNAME>
```

By default, `git` pushes the local branch to a remote branch with the same name. For example, if you have a local called `new-feature`, if you push the local branch it will create a remote branch `new-feature` as well. If you want to use a different name for the remote branch, append the remote name after the local branch name, separated by `:`:

```
git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

Section 14.11: Move current branch HEAD to an arbitrary commit

A branch is just a pointer to a commit, so you can freely move it around. To make it so that the branch is referring to the commit `aabbcc`, issue the command

```
git reset --hard aabbcc
```

Please note that this will overwrite your branch's current commit, and as so, its entire history. You might loose some work by issuing this command. If that's the case, you can use the reflog to recover the lost commits. It can be advised to perform this command on a new branch instead of your current one.

However, this command can be particularly useful when rebasing or doing such other large history modifications.

第15章：修订列表

参数	详情
--oneline	以单行显示提交及其标题。

第15.1节：列出master分支中但不在origin/master中的提交

```
git rev-list --oneline master ^origin/master
```

Git rev-list 会列出一个分支中而另一个分支中没有的提交。当你想要确定代码是否已经合并到某个分支时，这是一个非常有用的工具。

- 使用--oneline选项将显示每个提交的标题。
- ^操作符会将指定分支中的提交从列表中排除。
- 如果需要，你可以传入两个以上的分支。例如，git rev-list foo bar ^baz 会列出foo和bar中的提交，但不包括baz中的提交。

Chapter 15: Rev-List

Parameter	Details
--oneline	Display commits as a single line with their title.

Section 15.1: List Commits in master but not in origin/master

```
git rev-list --oneline master ^origin/master
```

Git rev-list will list commits in one branch that are not in another branch. It is a great tool when you're trying to figure out if code has been merged into a branch or not.

- Using the `--oneline` option will display the title of each commit.
- The `^` operator excludes commits in the specified branch from the list.
- You can pass more than two branches if you want. For example, `git rev-list foo bar ^baz` lists commits in foo and bar, but not baz.

第16章：合并提交（Squashing）

第16.1节：不使用变基（Rebasing）合并最近的提交

如果你想将之前的 x 次提交合并成一次提交，可以使用以下命令：

```
git reset --soft HEAD~x
git commit
```

将 x 替换为你想合并的之前提交的数量。

请注意，这将创建一个新的提交，实际上会丢失之前 x 次提交的信息，包括作者、提交信息和日期。你可能想要先复制粘贴一个已有的提交信息。

第16.2节：合并时压缩提交

你可以使用git merge --squash将一个分支引入的更改压缩成一个提交。不会创建实际的提交。

```
git merge --squash <branch>
git commit
```

这在某种程度上等同于使用git reset，但当被合并的更改有一个符号名称时更方便。比较：

```
git checkout <branch>
git reset --soft $(git merge-base master <branch>)
git commit
```

第16.3节：在变基过程中合并提交

提交可以在git rebase过程中合并。建议在尝试以这种方式合并提交之前，先了解rebase的相关知识。

1.确定你想要从哪个提交开始rebase，并记下该提交的哈希值。

2. **运行git rebase -i [commit hash]。**

   或者，你可以输入HEAD~4代替提交哈希，以查看最新的提交及其之前的4个提交。

3.在运行该命令后打开的编辑器中，确定你想要合并的提交。
将这些行开头的pick替换为 squash，以将它们合并到前一个提交中。

4.选择好要合并的提交后，系统会提示你编写提交信息。

记录提交以确定rebase的位置

```
> git log --oneline
612f2f7 这个提交不应被合并
d84b05d 这个提交应被合并
ac60234 另一个提交
36d15de 从这里开始变基
```


Chapter 16: Squashing

Section 16.1: Squash Recent Commits Without Rebasing

If you want to squash the previous `x` commits into a single one, you can use the following commands:

```
git reset --soft HEAD~x
git commit
```

Replacing `x` with the number of previous commits you want to be included in the squashed commit.

Mind that this will create a *new* commit, essentially forgetting information about the previous `x` commits including their author, message and date. You probably want to *first* copy-paste an existing commit message.

Section 16.2: Squashing Commit During Merge

You can use `git merge --squash` to squash changes introduced by a branch into a single commit. No actual commit will be created.

```
git merge --squash <branch>
git commit
```

This is more or less equivalent to using `git reset`, but is more convenient when changes being incorporated have a symbolic name. Compare:

```
git checkout <branch>
git reset --soft $(git merge-base master <branch>)
git commit
```

Section 16.3: Squashing Commits During a Rebase

Commits can be squashed during a `git rebase`. It is recommended that you understand rebasing before attempting to squash commits in this fashion.

1. Determine which commit you would like to rebase from, and note its commit hash.
2. Run `git rebase -i [commit hash]`.

   Alternatively, you can type `HEAD~4` instead of a commit hash, to view the latest commit and 4 more commits before the latest one.

3. In the editor that opens when running this command, determine which commits you want to squash. Replace `pick` at the beginning of those lines with `squash` to squash them into the previous commit.
4. After selecting which commits you would like to squash, you will be prompted to write a commit message.

Logging Commits to determine where to rebase

```
> git log --oneline
612f2f7 This commit should not be squashed
d84b05d This commit should be squashed
ac60234 Yet another commit
36d15de Rebase from here
```

```
17692d1 做了一些更多的事情
e647334 另一个提交
2e30df6 初始提交

> git rebase -i 36d15de
```

此时会弹出你选择的编辑器，你可以在其中描述你想对提交做的操作。Git
在注释中提供帮助。如果保持原样，则不会发生任何变化，因为每个提交都会被保留，
且它们的顺序与变基前相同。在本例中，我们应用以下命令：

```
pick ac60234 又一个提交
squash d84b05d 这个提交应该被合并
pick 612f2f7 这个提交不应该被合并

# 将 36d15de..612f2f7 变基到 36d15de（3 条命令）
#
# 命令：
# p, pick = 使用提交
# r, reword = 使用提交，但编辑提交信息
# e, edit = 使用提交，但暂停以便修改
# s, squash = 使用提交，但合并到前一个提交
# f, fixup = 类似“squash”，但丢弃该提交的日志信息
# x, exec = 使用 shell 运行命令（该行剩余部分）
#
# 这些行可以重新排序；它们从上到下依次执行。
#
# 如果你删除这里的一行，该提交将会丢失。
#
# 但是，如果你删除所有内容，变基将会中止。
#
# 注意空提交会被注释掉
```

写入提交信息后的Git日志

```
> git log --oneline
77393eb 这个提交不应该被合并
e090a8c 又一个提交
36d15de 从这里开始变基
17692d1 做了一些更多的事情
e647334 另一个提交
2e30df6 初始提交
```

第16.4节：自动压缩和修正

提交更改时，可以指定该提交将来会被压缩合并到另一个提交中
，操作方法如下，

```
git commit --squash=[要合并到的提交的commit hash]
```

也可以使用--fixup=[commit hash]作为替代来进行修正提交。

还可以使用提交信息中的关键词代替提交哈希，方法如下，

```
git commit --squash :/things
```

这里会使用最近的包含“things”一词的提交。

这些提交的信息会以'fixup!'或'squash!'开头，后面跟着将被合并的提交的剩余提交信息。


```
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit

> git rebase -i 36d15de
```

At this point your editor of choice pops up where you can describe what you want to do with the commits. Git provides help in the comments. If you leave it as is then nothing will happen because every commit will be kept and their order will be the same as they were before the rebase. In this example we apply the following commands:

```
pick ac60234 Yet another commit
squash d84b05d This commit should be squashed
pick 612f2f7 This commit should not be squashed

# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Git log after writing commit message

```
> git log --oneline
77393eb This commit should not be squashed
e090a8c Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit
```

Section 16.4: Autosquashing and fixups

When committing changes it is possible to specify that the commit will in future be squashed to another commit and this can be done like so,

```
git commit --squash=[commit hash of commit to which this commit will be squashed to]
```

One might also use, `--fixup=[commit hash]` alternatively to fixup.

It is also possible to use words from the commit message instead of the commit hash, like so,

```
git commit --squash :/things
```

where the most recent commit with the word 'things' would be used.

These commits' message would begin with `'fixup!'` or `'squash!'` followed by the rest of the commit message to which these commits will be squashed to.

在执行变基时，应使用--autosquash标志来启用自动压缩/修正功能。

第16.5节：自动压缩：在变基过程中提交你想要压缩的代码

假设有如下历史记录，想象你做了一个更改，想将其压缩合并到提交bbb2222 A second commit中：

```
$ git log --oneline --decorate
ccc3333 (HEAD -> master) 第三个提交
bbb2222 第二个提交
aaa1111 第一个提交
9999999初始提交
```

一旦你完成了修改，可以像往常一样将它们添加到索引中，然后使用带有 --fixup 参数的命令提交，参数后跟你想要合并的提交的引用：

```
$ git add .
$ git commit --fixup bbb2222
[my-feature-branch ddd4444] fixup! 第二个提交
```

这将创建一个新的提交，Git 在交互式变基时可以识别其提交信息：

```
$ git log --oneline --decorate
ddd4444 (HEAD -> master) fixup! 第二个提交
ccc3333 第三个提交
bbb2222 第二个提交
aaa1111 第一个提交
9999999 初始提交
```

接下来，使用 --autosquash 参数进行交互式变基：

```
$ git rebase --autosquash --interactive HEAD~4
```

Git 会建议你将使用 commit --fixup 创建的提交压缩合并到正确的位置：

```
pick aaa1111 第一次提交
pick bbb2222 第二次提交
fixup ddd4444 fixup! 第二次提交
pick ccc3333 第三次提交
```

为了避免每次变基时都要输入 --autosquash，你可以默认启用此选项：

```
$ git config --global rebase.autosquash true
```


When rebasing --autosquash flag should be used to use the autosquash/fixup feature.

Section 16.5: Autosquash: Committing code you want to squash during a rebase

Given the following history, imagine you make a change that you want to squash into the commit bbb2222 A second commit:

```
$ git log --oneline --decorate
ccc3333 (HEAD -> master) A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Once you've made your changes, you can add them to the index as usual, then commit them using the --fixup argument with a reference to the commit you want to squash into:

```
$ git add .
$ git commit --fixup bbb2222
[my-feature-branch ddd4444] fixup! A second commit
```

This will create a new commit with a commit message that Git can recognize during an interactive rebase:

```
$ git log --oneline --decorate
ddd4444 (HEAD -> master) fixup! A second commit
ccc3333 A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Next, do an interactive rebase with the --autosquash argument:

```
$ git rebase --autosquash --interactive HEAD~4
```

Git will propose you to squash the commit you made with the commit --fixup into the correct position:

```
pick aaa1111 A first commit
pick bbb2222 A second commit
fixup ddd4444 fixup! A second commit
pick ccc3333 A third commit
```

To avoid having to type --autosquash on every rebase, you can enable this option by default:

```
$ git config --global rebase.autosquash true
```

第17章：挑拣提交（Cherry Picking）

参数	详情
-e, --edit	使用此选项时，git cherry-pick 会让你在提交前编辑提交信息。
-x	在记录提交时，会在原始提交信息后附加一行“(cherry picked from commit ...)”以表明该更改是从哪个提交挑拣过来的。此操作仅针对无冲突的挑拣提交进行。
--ff	如果当前 HEAD 与被 cherry-pick 的提交的父提交相同，则会对该提交执行快进操作。
--继续	使用 .git/sequencer 中的信息继续正在进行的操作。可用于在解决失败的 cherry-pick 或 revert 冲突后继续操作。
--quit	放弃当前正在进行的操作。可用于在失败的 cherry-pick 或 revert 后清除 sequencer 状态。
--abort	取消操作并返回到序列操作之前的状态。

cherry-pick 会将某个提交中引入的补丁提取出来，尝试重新应用到你当前所在的分支上。

来源：Git SCM 书籍

第 17.1 节：将提交从一个分支复制到另一个分支

git cherry-pick <commit-hash> 会将现有提交中的更改应用到另一个分支，同时记录一个新的提交。基本上，你可以在分支之间复制提交。

给定以下树（来源）

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
          \
76cada - 62ecb3 - b886a0 [feature]
```

假设我们想把 b886a0 复制到 master（放在 5a6057 之上）。

我们可以运行

```
git checkout master
git cherry-pick b886a0
```

现在我们的树状结构大致如下：

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [master]
          \
76cada - 62ecb3 - b886a0 [feature]
```

其中新的提交 a66b23 具有与 b886a0 相同的内容（源代码差异，提交信息），但父提交不同。注意，cherry-pick 只会挑选该提交（本例中为 b886a0 ）的更改，而不是 feature 分支上的所有更改（如果要复制所有更改，需要使用 rebase 或 merge）。

第17.2节：从一个分支复制一系列提交到另一个分支

git cherry-pick <commit-A>..<commit-B> 会将从 A 之后到包括 B 的每个提交放到当前检出的分支顶部。

Chapter 17: Cherry Picking

Parameters	Details
-e, --edit	With this option, `git cherry-pick` will let you edit the commit message prior to committing.
-x	When recording the commit, append a line that says "(cherry picked from commit …)" to the original commit message in order to indicate which commit this change was cherry-picked from. This is done only for cherry picks without conflicts.
--ff	If the current HEAD is the same as the parent of the cherry-pick'ed commit, then a fast forward to this commit will be performed.
--continue	Continue the operation in progress using the information in .git/sequencer. Can be used to continue after resolving conflicts in a failed cherry-pick or revert.
--quit	Forget about the current operation in progress. Can be used to clear the sequencer state after a failed cherry-pick or revert.
--abort	Cancel the operation and return to the pre-sequence state.

A cherry-pick takes the patch that was introduced in a commit and tries to reapply it on the branch you're currently on.

Source: Git SCM Book

Section 17.1: Copying a commit from one branch to another

`git cherry-pick <commit-hash>` will apply the changes made in an existing commit to another branch, while recording a new commit. Essentially, you can copy commits from branch to branch.

Given the following tree (Source)

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
          \
           76cada - 62ecb3 - b886a0 [feature]
```

Let's say we want to copy `b886a0` to master (on top of `5a6057`).

We can run

```
git checkout master
git cherry-pick b886a0
```

Now our tree will look something like:

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [master]
          \
           76cada - 62ecb3 - b886a0 [feature]
```

Where the new commit `a66b23` has the same content (source diff, commit message) as `b886a0` (but a different parent). Note that cherry-picking will only pick up changes on that commit(`b886a0` in this case) not all the changes in feature branch (for this you will have to either use rebasing or merging).

Section 17.2: Copying a range of commits from one branch to another

`git cherry-pick <commit-A>..<commit-B>` will place every commit *after* A and up to and including B on top of the currently checked-out branch.

git cherry-pick <commit-A>^..<commit-B> 会将提交 A 及从 A 到包括 B 的所有提交放到当前检出的分支之上。

第17.3节：检查是否需要进行 cherry-pick

在开始 cherry-pick 过程之前，你可以检查想要 cherry-pick 的提交是否已经存在于目标分支中，如果存在，则无需任何操作。

git branch --contains <commit> 列出包含指定提交的本地分支。

git branch -r --contains <commit> 还会在列表中包含远程跟踪分支。

第17.4节：查找尚未应用到上游的提交

命令 git cherry 显示尚未被 cherry-pick 的更改。

示例：

```
git checkout master
git cherry development
```

... 并看到类似如下的输出：

```
+ 492508acab7b454eee8b805f8ba906056eede0ff
- 5ceb5a9077ddb9e78b1e8f24bfc70e674c627949
+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b
+ b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9
```

以+开头的提交将是尚未被挑选到development分支中的提交。

语法：

git cherry [-v] [<upstream> [<head> [<limit>]]]

选项：

-v 显示 SHA1 旁的提交主题。

< upstream > 用于搜索等效提交的上游分支。默认为 HEAD 的上游分支。

< head > 工作分支；默认为 HEAD。

< limit > 不报告直到（包括）limit 的提交。

更多信息请查看 git-cherry 文档。

git cherry-pick <commit-A>^..<commit-B> will place commit A and every commit up to and including B on top of the currently checked-out branch.

Section 17.3: Checking if a cherry-pick is required

Before you start the cherry-pick process, you can check if the commit you want to cherry-pick already exists in the target branch, in which case you don't have to do anything.

git branch --contains <commit> lists local branches that contain the specified commit.

git branch -r --contains <commit> also includes remote tracking branches in the list.

Section 17.4: Find commits yet to be applied to upstream

Command **git cherry** shows the changes which haven't yet been cherry-picked.

Example:

```
git checkout master
git cherry development
```

... and see output a bit like this:

```
+ 492508acab7b454eee8b805f8ba906056eede0ff
- 5ceb5a9077ddb9e78b1e8f24bfc70e674c627949
+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b
+ b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9
```

The commits that being with + will be the ones that haven't yet cherry-picked into `development`.

Syntax:

git cherry [-v] [<upstream> [<head> [<limit>]]]

Options:

-v Show the commit subjects next to the SHA1s.

< upstream > Upstream branch to search for equivalent commits. Defaults to the upstream branch of HEAD.

< head > Working branch; defaults to HEAD.

< limit > Do not report commits up to (and including) limit.

Check git-cherry documentation for more info.

第18章：恢复

第18.1节：从重置中恢复

使用 Git，您几乎总能将时间倒回去

不要害怕尝试那些会重写历史的命令*。Git 默认不会删除您的提交，保存期为90天，在此期间您可以轻松地从 reflog 中恢复它们：

```
$ git reset @~3   # 回退3次提交
$ git reflog
c4f708b HEAD@{0}: reset: 移动到 @~3
2c52489 HEAD@{1}: commit: 更多更改
4a5246d HEAD@{2}: commit: 进行重要更改
e8571e4 HEAD@{3}: commit: 进行一些更改
... 更早的提交 ...
$ git reset 2c52489
... 您回到了起点
```

** 但要注意像 --hard 和 --force 这样的选项 — 它们可能会丢弃数据。*
** 此外，避免在您正在协作的分支上重写历史。*

第18.2节：从 git stash 恢复

运行 git stash 后，要获取您最近的 stash，请使用

```
git stash apply
```

要查看你的存储列表，请使用

```
git stash list
```

你将得到类似如下的列表

```
stash@{0}: 在 master 上的未完成工作（WIP）：67a4e01 将测试合并到 develop
stash@{1}: 在 master 上的未完成工作（WIP）：70f0d95 在用户登录时将用户角色添加到 localStorage
```

使用显示的编号选择要恢复的不同 git 存储

```
git stash apply stash@{2}
```

你也可以选择 'git stash pop'，它的作用和 'git stash apply' 一样，比如..

```
git stash pop
```

或

```
git stash pop stash@{2}
```

git stash apply 和 git stash pop 的区别...

git stash pop：stash 数据将从 stash 列表的栈中移除。

示例：

Chapter 18: Recovering

Section 18.1: Recovering from a reset

With Git, you can (almost) always turn the clock back

Don't be afraid to experiment with commands that rewrite history*. Git doesn't delete your commits for 90 days by default, and during that time you can easily recover them from the reflog:

```
$ git reset @~3   # go back 3 commits
$ git reflog
c4f708b HEAD@{0}: reset: moving to @~3
2c52489 HEAD@{1}: commit: more changes
4a5246d HEAD@{2}: commit: make important changes
e8571e4 HEAD@{3}: commit: make some changes
... earlier commits ...
$ git reset 2c52489
... and you're back where you started
```

** Watch out for options like `--hard` and `--force` though — they can discard data.*
** Also, avoid rewriting history on any branches you're collaborating on.*

Section 18.2: Recover from git stash

To get your most recent stash after running git stash, use

```
git stash apply
```

To see a list of your stashes, use

```
git stash list
```

You will get a list that looks something like this

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Choose a different git stash to restore with the number that shows up for the stash you want

```
git stash apply stash@{2}
```

You can also choose 'git stash pop', it works same as 'git stash apply' like..

```
git stash pop
```

or

```
git stash pop stash@{2}
```

Difference in git stash apply and git stash pop...

git stash pop: stash data will be remove from stack of stash list.

Ex:

```
git stash list
```

你将得到类似如下的列表

```
stash@{0}: 在 master 上的未完成工作（WIP）：67a4e01 将测试合并到 develop
stash@{1}: 在 master 上的未完成工作（WIP）：70f0d95 在用户登录时将用户角色添加到 localStorage
```

现在使用命令弹出 stash 数据

```
git stash pop
```

再次检查 stash 列表

```
git stash list
```

你将得到类似如下的列表

```
stash@{0}：在 master 分支上的工作进行中（WIP）：70f0d95 在用户登录时将用户角色添加到 localStorage
```

你可以看到一个 stash 数据已从 stash 列表中被移除（弹出），stash@{1} 变成了 stash@{0}。

第18.3节：从丢失的提交中恢复

如果你回退到了之前的提交并丢失了较新的提交，可以通过运行

以下命令来恢复丢失的提交

```
git reflog
```

然后找到丢失的提交，并通过执行以下命令重置回该提交

```
git reset HEAD --hard <sha1-of-commit>
```

第18.4节：在提交后恢复已删除的文件

如果你不小心提交了删除文件的操作，后来又发现需要恢复该文件。

首先找到删除该文件的提交ID。

```
git log --diff-filter=D --summary
```

将为你提供一个按顺序排列的删除文件的提交摘要。

然后通过以下命令恢复文件

```
git checkout 81eeccf~1 <your-lost-file-name>
```

（将81eeccf替换为你自己的提交ID）

第18.5节：将文件恢复到之前的版本

要将文件恢复到之前的版本，可以使用reset命令。

```
git reset <sha1-of-commit> <file-name>
```



```
git stash list
```

You will get a list that looks something like this

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Now pop stash data using command

```
git stash pop
```

Again Check for stash list

```
git stash list
```

You will get a list that looks something like this

```
stash@{0}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

You can see one stash data is removed (popped) from stash list and stash@{1} became stash@{0}.

Section 18.3: Recovering from a lost commit

In case you have reverted back to a past commit and lost a newer commit you can recover the lost commit by running

```
git reflog
```

Then find your lost commit, and reset back to it by doing

```
git reset HEAD --hard <sha1-of-commit>
```

Section 18.4: Restore a deleted file after a commit

In case you have accidentally commited a delete on a file and later realized that you need it back.

First find the commit id of the commit that deleted your file.

```
git log --diff-filter=D --summary
```

Will give you a sorted summary of commits which deleted files.

Then proceed to restore the file by

```
git checkout 81eeccf~1 <your-lost-file-name>
```

(Replace 81eeccf with your own commit id)

Section 18.5: Restore file to a previous version

To restore a file to a previous version you can use `reset`.

```
git reset <sha1-of-commit> <file-name>
```

如果你已经对文件做了本地修改（但你不需要这些修改！），你也可以使用--hard选项

第18.6节：恢复已删除的分支

要恢复已删除的分支，你需要通过运行命令找到被删除分支的头指针对应的提交

```
git reflog
```

然后你可以通过运行以下命令重新创建该分支

```
git checkout -b <branch-name> <sha1-of-commit>
```

如果git的垃圾回收器删除了悬挂提交（即没有引用的提交），你将无法恢复已删除的分支。始终备份你的仓库，尤其是在小团队或专有项目中工作时

If you have already made local changes to the file (that you do not require!) you can also use the `--hard` option

Section 18.6: Recover a deleted branch

To recover a deleted branch you need to find the commit which was the head of your deleted branch by running

```
git reflog
```

You can then recreate the branch by running

```
git checkout -b <branch-name> <sha1-of-commit>
```

You will not be able to recover deleted branches if git's garbage collector deleted dangling commits - those without refs. Always have a backup of your repository, especially when you work in a small team / proprietary project

第19章：Git Clean

参数	详情
-d	除了未跟踪的文件外，还会删除未跟踪的目录。如果未跟踪的目录由另一个Git仓库管理，默认情况下不会被删除。如果你确实想删除这样的目录，请使用-f选项两次。
-f，--force	如果 Git 配置变量 clean.requireForce 未设置为 false，git clean 将拒绝删除文件或目录，除非使用了 -f、-n 或 -i。Git 会拒绝删除包含 .git 子目录或文件的目录，除非再次使用 -f。
-i, --interactive	交互式提示删除每个文件。
-n, --dry-run	仅显示将被删除的文件列表，而不实际删除它们。
-q,--quiet	仅显示错误，不显示成功删除的文件列表。

第19.1节：交互式清理

```
git clean -i
```

将打印出要删除的项目，并通过如下命令请求确认：

```
将移除以下项目：
folder/file1.py
folder/file2.py
*** 命令 ***
1: 清理        2: 按模式过滤        3: 按数字选择        4: 逐个询问
5: 退出        6: 帮助
现在怎么办>
```

交互选项 `i` 可以与其他选项如 `X`、`d` 等一起使用。

第19.2节：强制删除未跟踪文件

```
git clean -f
```

将删除所有未跟踪的文件。

第19.3节：清理被忽略的文件

```
git clean -fX
```

将删除当前目录及所有子目录中所有被忽略的文件。

```
git clean -Xn
```

将预览所有将被清理的文件。

第19.4节：清理所有未跟踪的目录

```
git clean -fd
```

将删除所有未跟踪的目录及其中的文件。它将从当前工作目录开始，
遍历所有子目录。

Chapter 19: Git Clean

Parameter	Details
-d	Remove untracked directories in addition to untracked files. If an untracked directory is managed by a different Git repository, it is not removed by default. Use -f option twice if you really want to remove such a directory.
-f, --force	If the Git configuration variable clean. `requireForce` is not set to false, git clean will refuse to delete files or directories unless given -f, -n or -i. Git will refuse to delete directories with .git sub directory or file unless a second -f is given.
-i, --interactive	Interactively prompts the removal of each file.
-n, --dry-run	Only displays a list of files to be removed, without actually removing them.
-q,--quiet	Only display errors, not the list of successfully removed files.

Section 19.1: Clean Interactively

```
git clean -i
```

Will print out items to be removed and ask for a confirmation via commands like the follow:

```
Would remove the following items:
folder/file1.py
folder/file2.py
*** Commands ***
1: clean        2: filter by pattern        3: select by numbers        4: ask each
5: quit         6: help
What now>
```

Interactive option `i` can be added along with other options like `X`, `d`, etc.

Section 19.2: Forcefully remove untracked files

```
git clean -f
```

Will remove all untracked files.

Section 19.3: Clean Ignored Files

```
git clean -fX
```

Will remove all ignored files from the current directory and all subdirectories.

```
git clean -Xn
```

Will preview all files that will be cleaned.

Section 19.4: Clean All Untracked Directories

```
git clean -fd
```

Will remove all untracked directories and the files within them. It will start at the current working directory and will iterate through all subdirectories.

```
git clean -dn
```

将预览所有将被清理的目录。


```
git clean -dn
```

Will preview all directories that will be cleaned.

第20章：使用 .gitattributes 文件

第20.1节：自动换行符规范化

在项目根目录创建一个.gitattributes文件，内容如下：

```
* text=auto
```

这将导致所有文本文件（由Git识别）以LF格式提交，但检出时根据主机
操作系统默认设置进行。

这相当于推荐的core.autocrlf默认值：

- Linux/macOS 上为 input
- **Windows 上为 true**

第20.2节：识别二进制文件

Git在识别二进制文件方面表现不错，但你可以明确指定哪些文件是二进制文件。创建一个
.gitattributes文件在项目根目录，内容如下：

```
*.png 二进制
```

binary 是一个内置宏属性，相当于 -diff -merge -text。

第20.3节：预填充的 .gitattribute 模板

如果您不确定在 .gitattributes 文件中应列出哪些规则，或者您只是想向项目添加公认的
属性，可以在以下位置选择或生成一个 .gitattributes 文件：

- https://gitattributes.io/
- https://github.com/alexkaratarakis/gitattributes

第20.4节：禁用行尾规范化

在项目根目录创建一个.gitattributes文件，内容如下：

```
* -text
```

这相当于设置 core.autocrlf = false。

Chapter 20: Using a .gitattributes file

Section 20.1: Automatic Line Ending Normalization

Create a `.gitattributes` file in the project root containing:

```
* text=auto
```

This will result in all text files (as identified by Git) being committed with LF, but checked out according to the host operating system default.

This is equivalent to the recommended `core.autocrlf` defaults of:

- `input` on Linux/macOS
- **`true`** on Windows

Section 20.2: Identify Binary Files

Git is pretty good at identifying binary files, but you can explicitly specify which files are binary. Create a `.gitattributes` file in the project root containing:

```
*.png binary
```

`binary` is a built-in macro attribute equivalent to `-diff -merge -text`.

Section 20.3: Prefilled .gitattribute Templates

If you are unsure which rules to list in your `.gitattributes` file, or you just want to add generally accepted attributes to your project, you can shoose or generate a `.gitattributes` file at:

- https://gitattributes.io/
- https://github.com/alexkaratarakis/gitattributes

Section 20.4: Disable Line Ending Normalization

Create a `.gitattributes` file in the project root containing:

```
* -text
```

This is equivalent to setting `core.autocrlf = false`.

第21章：.mailmap 文件：关联贡献者和邮箱别名

第21.1节：通过别名合并贡献者以在 shortlog 中显示提交计数

当贡献者从不同的机器或操作系统向项目添加内容时，可能会使用不同的电子邮件地址或姓名，这会导致贡献者列表和统计数据分散。

运行git shortlog-sn以获取贡献者列表及其提交次数，可能会得到如下输出：

```
帕特里克·罗斯福斯 871
伊丽莎白·穆恩 762
E. Moon 184
罗斯福斯，帕特里克 90
```

这种分散/脱节可以通过提供一个纯文本文件.mailmap来调整，该文件包含电子邮件映射关系。

同一行中列出的所有姓名和电子邮件地址将分别关联到第一个命名实体。

以上例子中的映射文件可能如下所示：

```
帕特里克·罗斯福斯<fussy@kingkiller.com>罗斯福斯，帕特里克<fussy@kingkiller.com>
伊丽莎白·穆恩<emoon@marines.mil>E. Moon<emoon@scifi.org>
```

一旦该文件存在于项目根目录，重新运行git shortlog-sn将得到一个合并后的列表：

```
帕特里克·罗斯福斯 961
伊丽莎白·穆恩 946
```


Chapter 21: .mailmap file: Associating contributor and email aliases

Section 21.1: Merge contributers by aliases to show commit count in shortlog

When contributors add to a project from different machines or operating systems, it may happen that they use different email addresses or names for this, which will fragment contributor lists and statistics.

Running `git shortlog -sn` to get a list of contributors and the number of commits by them could result in the following output:

```
Patrick Rothfuss 871
Elizabeth Moon 762
E. Moon 184
Rothfuss, Patrick 90
```

This fragmentation/disassociation may be adjusted by providing a plain text file `.mailmap`, containing email mappings.

All names and email addresses listed in one line will be associated to the first named entity respectively.

For the example above, a mapping could look like this:

```
Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>
Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>
```

Once this file exists in the project's root, running `git shortlog -sn` again will result in a condensed list:

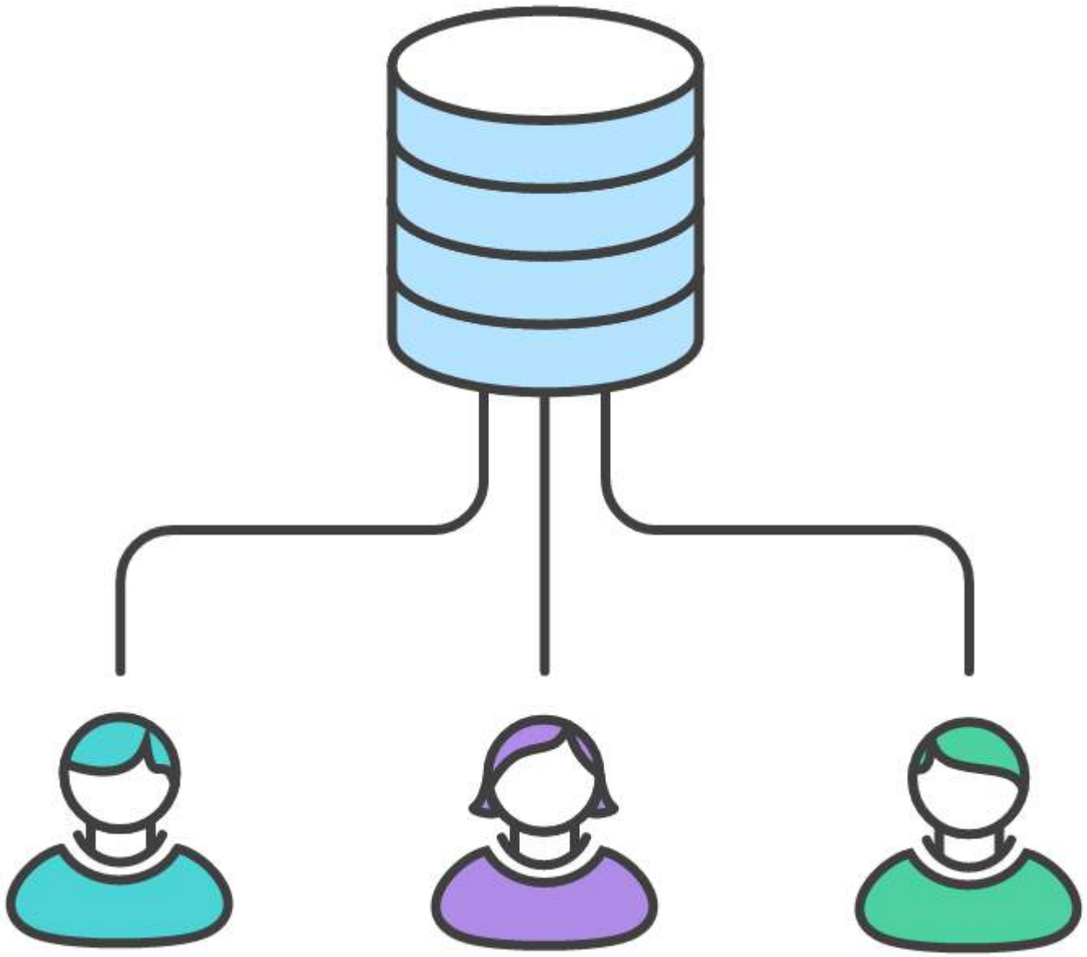
```
Patrick Rothfuss 961
Elizabeth Moon 946
```

第22章：分析工作流程类型

第22.1节：集中式工作流

在这个基本的工作流模型中，master 分支包含所有活跃的开发。贡献者在继续开发之前需要特别确保拉取最新的更改，因为该分支会快速变化。每个人都可以访问这个仓库，并且可以直接提交更改到 master 分支。

该模型的可视化表示：



这是经典的版本控制范式，早期的系统如 Subversion 和 CVS 就是基于此构建的。
以这种方式工作的软件被称为集中式版本控制系统（Centralized Version Control Systems，简称 CVCS）。虽然 Git 也能以这种方式工作，但存在显著的缺点，比如每次拉取之前都必须先进行合并。团队完全可以采用这种方式工作，但频繁的合并冲突解决可能会消耗大量宝贵时间。

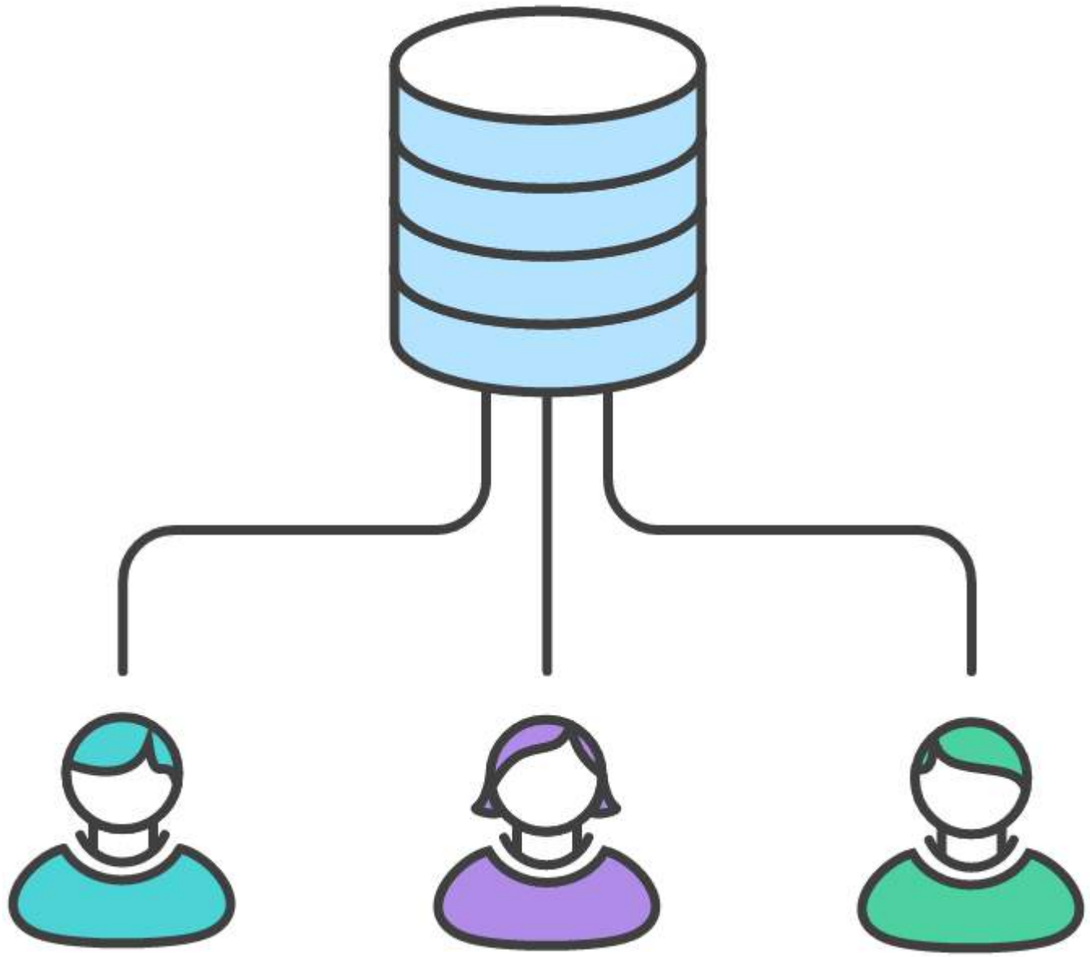
这就是 Linus Torvalds 创建 Git 的原因，他并没有将其设计为 CVCS，而是作为一个DVCS，或称为分布式版本控制系统，类似于 Mercurial。这种新方式的优势在于本页其他示例中展示的灵活性。

Chapter 22: Analyzing types of workflows

Section 22.1: Centralized Workflow

With this fundamental workflow model, a `master` branch contains all active development. Contributors will need to be especially sure they pull the latest changes before continuing development, for this branch will be changing rapidly. Everyone has access to this repo and can commit changes right to the master branch.

Visual representation of this model:



This is the classic version control paradigm, upon which older systems like Subversion and CVS were built. Softwares that work this way are called Centralized Version Control Systems, or CVCS's. While Git is capable of working this way, there are notable disadvantages, such as being required to precede every pull with a merge. It's very possible for a team to work this way, but the constant merge conflict resolution can end up eating a lot of valuable time.

This is why Linus Torvalds created Git not as a CVCS, but rather as a *DVCS*, or *Distributed Version Control System*, similar to Mercurial. The advantage to this new way of doing things is the flexibility demonstrated in the other examples on this page.

第22.2节：Gitflow 工作流

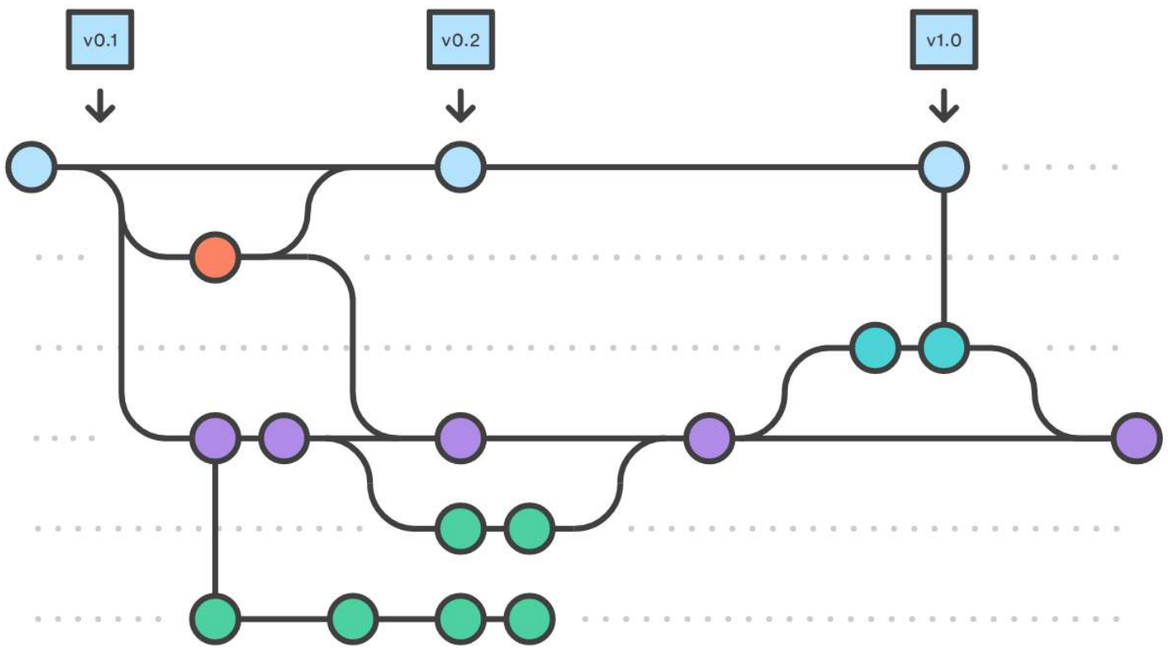
Gitflow 最初由Vincent Driessen提出，是一种使用 git 和多个预定义分支的开发工作流。这可以看作是功能分支工作流的一个特例。

其核心思想是为开发中的特定部分保留独立的分支：

- master 分支始终是最新的生产代码。实验性代码不应放在这里。
- develop分支包含所有最新的开发内容。这些开发变更几乎可以是任何内容，但较大的功能会保留在它们自己的分支中。这里的代码始终在开发中，并在发布/部署前合并到release分支。

- hotfix分支用于无法等待下一个版本发布的轻微错误修复。 hotfix分支从master分支分出，并合并回master和develop分支。
- release分支用于将develop的新开发发布到master。任何最后时刻的更改，例如版本号的提升，都会在release分支中完成，然后合并回master和develop。部署新版本时，master应打上当前版本号的标签（例如使用语义化版本控制）以便将来参考和轻松回滚。
- feature分支保留给较大的功能开发。这些功能专门在指定分支中开发，完成后与develop合并。专用的feature分支有助于分离开发，并能够独立部署已完成的功能。

该模型的可视化表示：



该模型的原始表示：

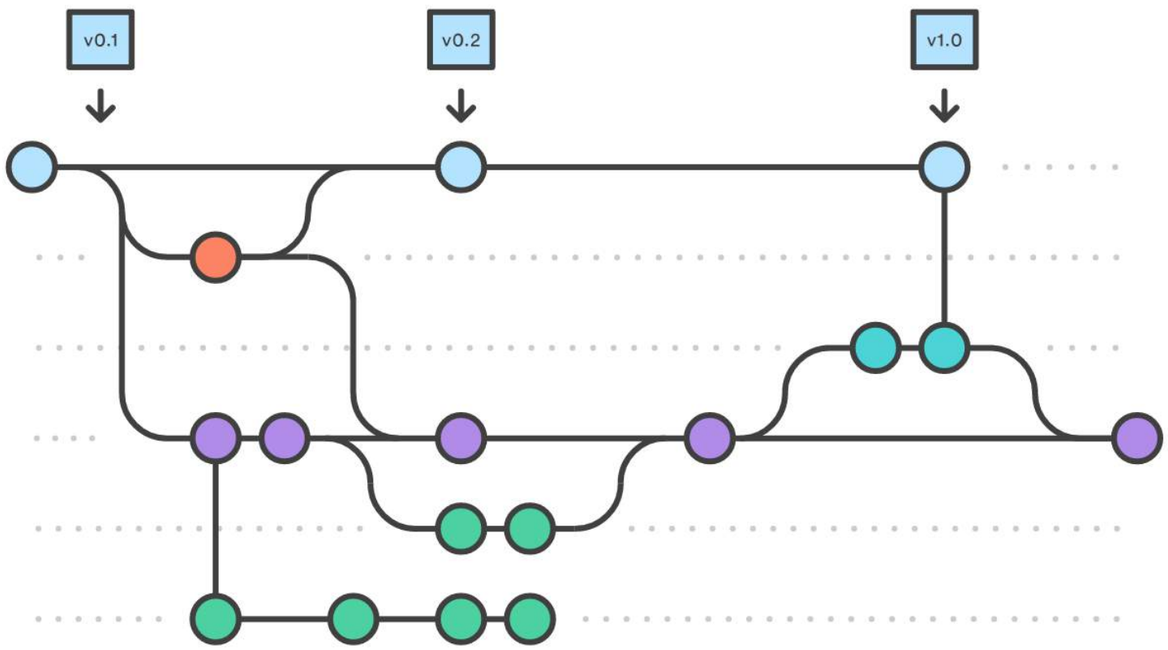
Section 22.2: Gitflow Workflow

Originally proposed by Vincent Driessen, Gitflow is a development workflow using git and several pre-defined branches. This can seen as a special case of the Feature Branch Workflow.

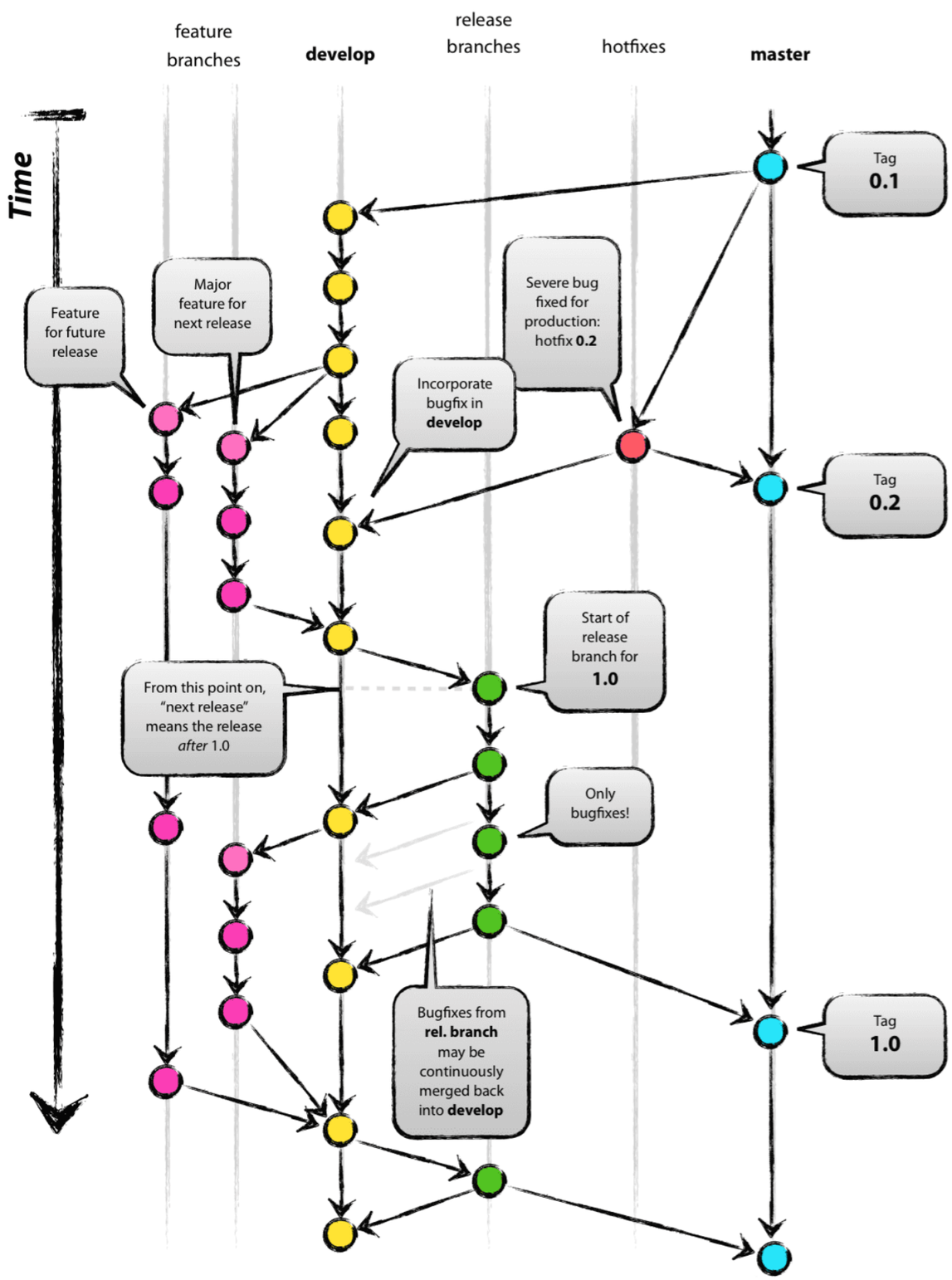
The idea of this one is to have separate branches reserved for specific parts in development:

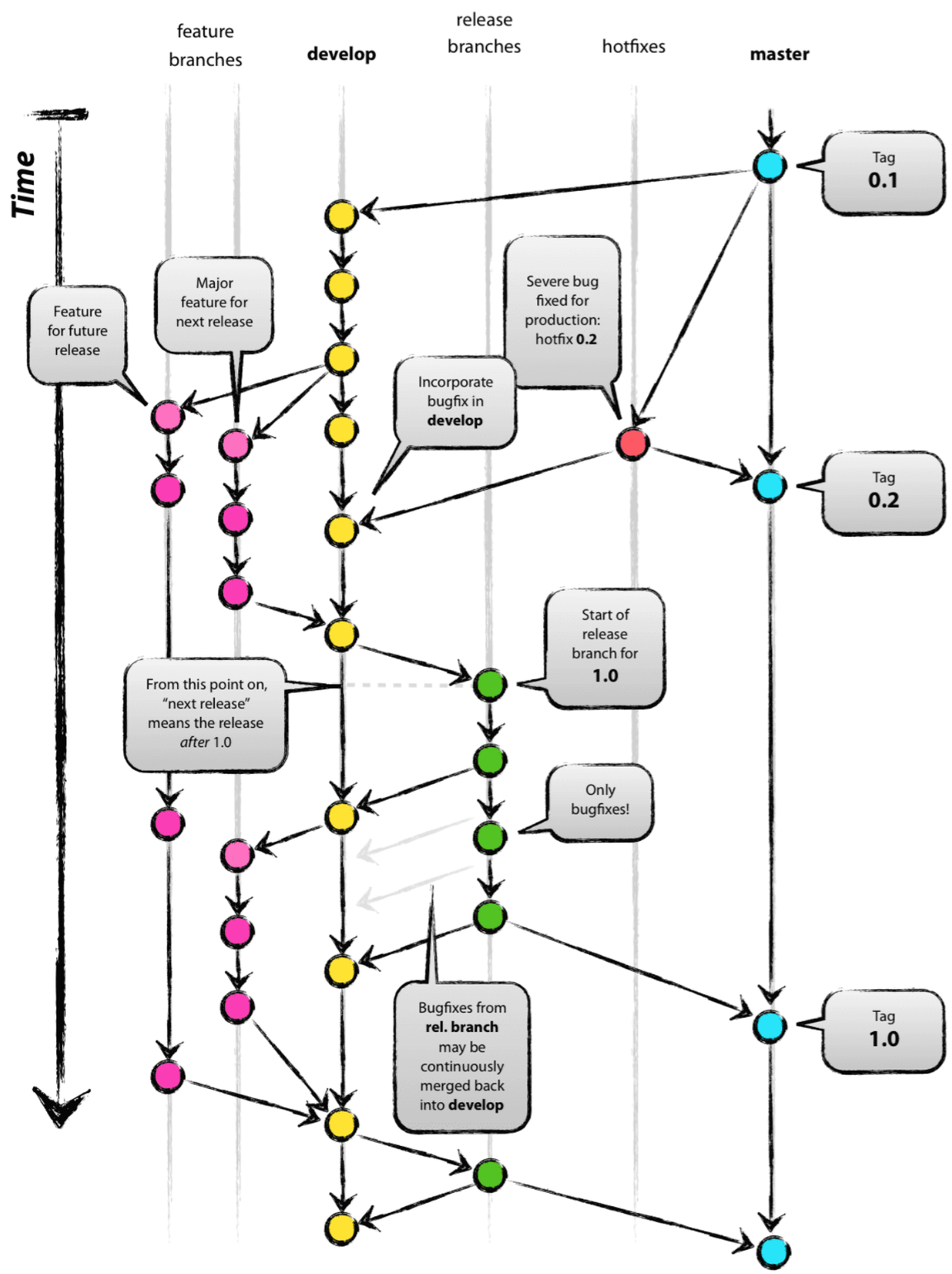
- `master` branch is always the most recent *production* code. Experimental code does not belong here.
- `develop` branch contains all of the latest *development*. These developmental changes can be pretty much anything, but larger features are reserved for their own branches. Code here is always worked on and merged into `release` before release / deployment.
- `hotfix` branches are for minor bug fixes, which cannot wait until the next release. `hotfix` branches come off of `master` and are merged back into both `master` and `develop`.
- `release` branches are used to release new development from `develop` to `master`. Any last minute changes, such as bumping version numbers, are done in the release branch, and then are merged back into `master` and `develop`. When deploying a new version, `master` should be tagged with the current version number (e.g. using semantic versioning) for future reference and easy rollback.
- `feature` branches are reserved for bigger features. These are specifically developed in designated branches and integrated with `develop` when finished. Dedicated `feature` branches help to separate development and to be able to deploy *done* features independently from each other.

A visual representation of this model:



The original representation of this model:


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

第22.3节：功能分支工作流

功能分支工作流的核心思想是所有功能开发都应在专用分支中进行，而不是在master分支中。此封装使多个开发者能够在不干扰主代码库的情况下共同开发特定功能。这也意味着master分支永远不会包含损坏的代码，这对于持续集成环境是一个巨大优势。

封装功能开发还使得利用拉取请求成为可能，拉取请求是一种围绕分支发起讨论的方式。它们为其他开发者提供了在功能被集成到官方项目之前进行审核的机会。或者，如果你在开发某个功能的过程中遇到困难，你可以打开一个拉取请求，向同事寻求建议。关键是，拉取请求让你的团队能够非常方便地相互评论彼此的工作。

基于Atlassian教程。

第22.4节：GitHub流程

在许多开源项目中很受欢迎，但不限于此。

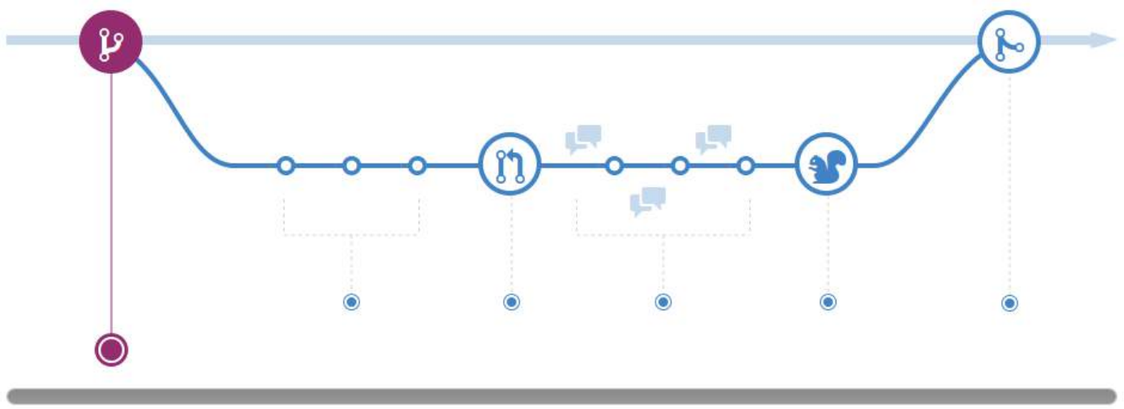
特定位置（Github、Gitlab、Bitbucket、本地服务器）的主分支包含最新的可发布版本。
对于每个新功能/错误修复/架构更改，每个开发者都会创建一个分支。

更改发生在该分支上，可以在拉取请求、代码审查等中讨论。一旦被接受，它们会合并到主分支。

Scott Chacon的完整流程：

- 主分支中的任何内容都是可部署的
- 要开始新工作，从主分支创建一个描述性命名的分支（例如：new-oauth2-scopes）
- 在本地提交到该分支，并定期将你的工作推送到服务器上同名分支
- 当你需要反馈或帮助，或者认为分支已准备好合并时，打开一个拉取请求
- 在其他人审查并批准该功能后，你可以将其合并到主分支
- 一旦合并并推送到'master'，你可以且应该立即部署

最初发布于Scott Chacon的个人网站。



图片来源于GitHub Flow参考资料

Section 22.3: Feature Branch Workflow

The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the `master` branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the `master` branch will never contain broken code, which is a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues. The point is, pull requests make it incredibly easy for your team to comment on each other's work.

based on Atlassian Tutorials.

Section 22.4: GitHub Flow

Popular within many open source projects but not only.

Master branch of a specific location (Github, Gitlab, Bitbucket, local server) contains the latest shippable version. For each new feature/bug fix/architectural change each developer creates a branch.

Changes happen on that branch and can be discussed in a pull request, code review, etc. Once accepted they get merged to the master branch.

Full flow by Scott Chacon:

- Anything in the master branch is deployable
- To work on something new, create a descriptively named branch off of master (ie: new-oauth2-scopes)
- Commit to that branch locally and regularly push your work to the same named branch on the server
- When you need feedback or help, or you think the branch is ready for merging, open a pull request
- After someone else has reviewed and signed off on the feature, you can merge it into master
- Once it is merged and pushed to 'master', you can and should deploy immediately

Originally presented on Scott Chacon's personal web site.

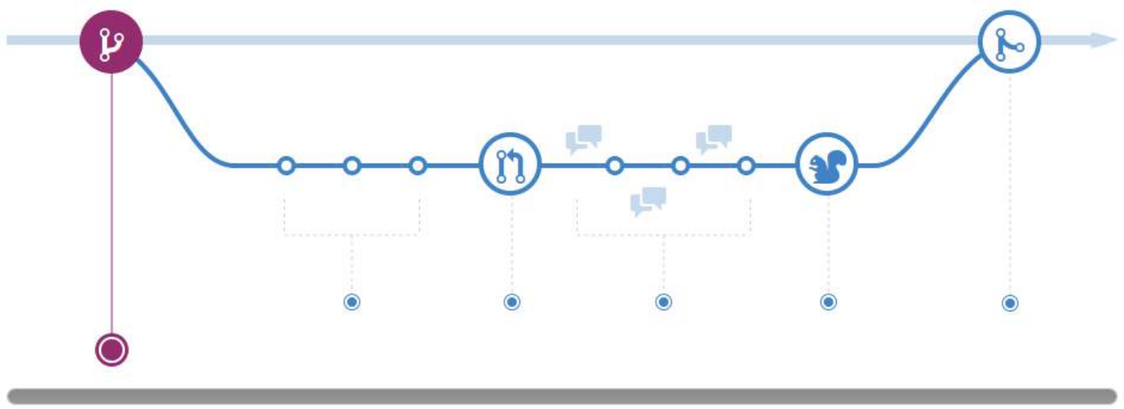
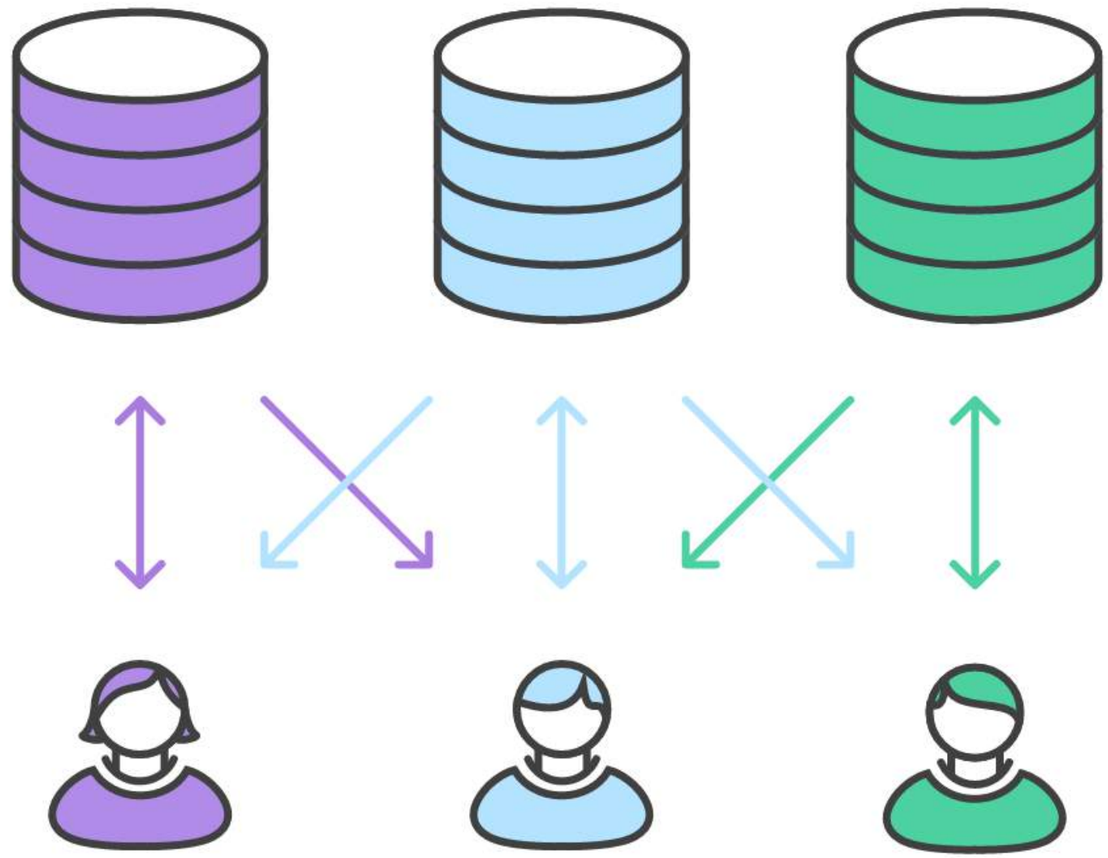


Image courtesy of the GitHub Flow reference

第22.5节：分叉工作流

这种工作流类型与本主题中提到的其他类型有根本区别。开发者不是访问一个
集中式仓库，而是每个开发者都有自己从
主仓库分叉出来的仓库。这样做的好处是开发者可以向自己的仓库提交，而不是向共享仓库提交，
维护者可以在合适的时候将分叉仓库中的更改合并到原始仓库中。

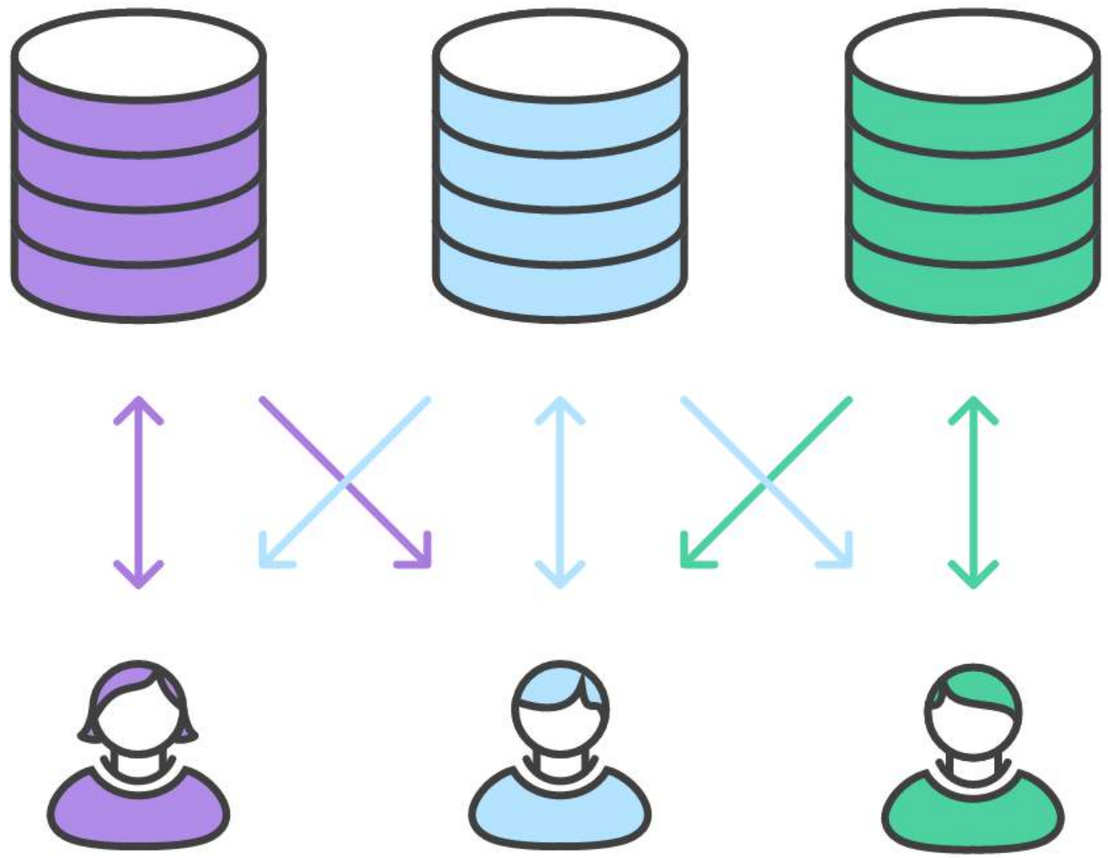
该工作流的可视化表示如下：



Section 22.5: Forking Workflow

This type of workflow is fundamentally different than the other ones mentioned on this topic. Instead of having one centralized repo that all developers have access to, each developer has his/her *own* repo that is forked from the main repo. The advantage of this is that developers can post to their own repos rather than a shared repo and a maintainer can integrate the changes from the forked repos into the original whenever appropriate.

A visual representation of this workflow is as follows:



第23章：拉取

参数	详情
`--quiet`	无文本输出
`-q`	--quiet的简写
`--verbose`	详细文本输出。分别传递给 fetch 和 merge/rebase 命令。
`-v`	--verbose 的缩写
--[no-]递归子模块[=是|按需|否]	是否获取子模块的新提交？（这并不是拉取/检出）

与使用 Git 推送时将本地更改发送到中央仓库服务器不同，使用 Git 拉取是将服务器上的当前代码“拉取”到本地机器中。本文介绍了使用 Git 从仓库拉取代码的过程，以及在将不同代码拉取到本地副本时可能遇到的情况。

第 23.1 节：将更改拉取到本地仓库

简单拉取

当你与他人在远程仓库（例如 GitHub）上协作时，某个时刻你会想要与他们共享你的更改。一旦他们将更改推送到远程仓库，你就可以通过从该仓库拉取来获取这些更改。

```
git pull
```

在大多数情况下会这样做。

从不同的远程或分支拉取

你可以通过指定远程或分支的名称来从不同的远程或分支拉取更改

```
git pull origin feature-A
```

将分支feature-A从origin拉取到你的本地分支。注意，你可以直接提供一个URL代替远程名称，也可以用一个对象名称（如提交SHA）代替分支名称。

手动拉取

为了模拟git pull的行为，你可以先使用git fetch然后使用git merge

```
git fetch origin # 从origin检索对象并更新引用
git merge origin/feature-A # 实际执行合并
```

这可以让你获得更多控制权，并允许你在合并之前检查远程分支。实际上，拉取后，你可以用git branch -a查看远程分支，并用以下命令检出它们

```
git checkout -b local-branch-name origin/feature-A # 检出远程分支
# 检查分支，进行提交、合并、修改提交信息或其他操作
git checkout merging-branches # 切换到目标分支
```


Chapter 23: Pulling

Parameters	Details
`--quiet`	No text output
`-q`	shorthand for `--quiet`
`--verbose`	verbose text output. Passed to fetch and merge/rebase commands respectively.
`-v`	shorthand for `--verbose`
`--[no-]recurse-submodules[=yes|on-demand|no]`	Fetch new commits for submodules? (Not that this is not a pull/checkout)

Unlike pushing with Git where your local changes are sent to the central repository's server, pulling with Git takes the current code on the server and 'pulls' it down from the repository's server to your local machine. This topic explains the process of pulling code from a repository using Git as well as the situations one might encounter while pulling different code into the local copy.

Section 23.1: Pulling changes to a local repository

Simple pull

When you are working on a remote repository (say, GitHub) with someone else, you will at some point want to share your changes with them. Once they have pushed their changes to a remote repository, you can retrieve those changes by *pulling* from this repository.

```
git pull
```

Will do it, in the majority of cases.

Pull from a different remote or branch

You can pull changes from a different remote or branch by specifying their names

```
git pull origin feature-A
```

Will pull the branch `feature-A` form `origin` into your local branch. Note that you can directly supply an URL instead of a remote name, and an object name such as a commit SHA instead of a branch name.

Manual pull

To imitate the behavior of a git pull, you can use `git fetch` then `git merge`

```
git fetch origin # retrieve objects and update refs from origin
git merge origin/feature-A # actually perform the merge
```

This can give you more control, and allows you to inspect the remote branch before merging it. Indeed, after fetching, you can see the remote branches with `git branch -a`, and check them out with

```
git checkout -b local-branch-name origin/feature-A # checkout the remote branch
# inspect the branch, make commits, squash, ammend or whatever
git checkout merging-branches # moving to the destination branch
```

```
git merge local-branch-name # 执行合并
```

这在处理拉取请求时非常方便。

第23.2节：使用本地更改进行更新

当存在本地更改时，git pull 命令会中止并报告：

> 错误：您对以下文件的本地更改将被合并覆盖

为了更新（类似于 svn update 对 subversion 的操作），您可以运行：

```
git stash
git pull --rebase
git stash pop
```

一个方便的方法是使用以下命令定义别名：

版本 < 2.9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

版本 ≥ 2.9

```
git config --global alias.up 'pull --rebase --autostash'
```

接下来您只需简单使用：

```
git up
```

第23.3节：拉取，覆盖本地

```
git fetch
git reset --hard origin/master
```

注意： 使用 `reset --hard` 丢弃的提交可以通过 `reflog` 和 `reset` 恢复，但未提交的更改将被永久删除。

如果远程和分支名称不同，请将 origin 和 master 分别更改为你想强制拉取的远程和分支。

第23.4节：从远程拉取代码

```
git pull
```

第23.5节：拉取时保持线性历史

拉取时变基

如果你正在从远程仓库拉取新的提交，并且当前分支有本地更改，
git 会自动合并远程版本和你的版本。如果你想减少分支上的合并次数，
可以告诉 git 在远程分支版本上变基你的提交。

```
git pull --rebase
```



```
git merge local-branch-name # performing the merge
```

This can be very handy when processing pull requests.

Section 23.2: Updating with local changes

When local changes are present, the `git pull` command aborts reporting :

> error: Your local changes to the following files would be overwritten by merge

In order to update (like svn update did with subversion), you can run :

```
git stash
git pull --rebase
git stash pop
```

A convenient way could be to define an alias using :

Version < 2.9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

Version ≥ 2.9

```
git config --global alias.up 'pull --rebase --autostash'
```

Next you can simply use :

```
git up
```

Section 23.3: Pull, overwrite local

```
git fetch
git reset --hard origin/master
```

Beware: While commits discarded using `reset --hard` can be recovered using `reflog` and `reset`, uncommitted changes are deleted forever.

Change `origin` and `master` to the remote and branch you want to forcibly pull to, respectively, if they are named differently.

Section 23.4: Pull code from remote

```
git pull
```

Section 23.5: Keeping linear history when pulling

Rebasing when pulling

If you are pulling in fresh commits from the remote repository and you have local changes on the current branch then git will automatically merge the remote version and your version. If you would like to reduce the number of merges on your branch you can tell git to rebase your commits on the remote version of the branch.

```
git pull --rebase
```

将其设为默认行为

要将此设置为新创建分支的默认行为，请输入以下命令：

```
git config branch.autosetuprebase always
```

要更改现有分支的行为，请使用此命令：

```
git config branch.BRANCH_NAME.rebase true
```

并且

```
git pull --no-rebase
```

执行正常的合并拉取操作。

检查是否可以快进合并

若只允许本地分支快进合并，可以使用：

```
git pull --ff-only
```

当本地分支无法快进时，将显示错误，需要对其进行变基或与上游合并。

第23.6节：拉取，“权限被拒绝”

如果.git文件夹权限错误，可能会出现一些问题。通过设置整个.git文件夹的所有者来解决此问题。有时会发生其他用户拉取并更改.git文件夹或文件的权限。

解决此问题的方法：

```
chown -R youruser:yourgroup .git/
```


Making it the default behavior

To make this the default behavior for newly created branches, type the following command:

```
git config branch.autosetuprebase always
```

To change the behavior of an existing branch, use this:

```
git config branch.BRANCH_NAME.rebase true
```

And

```
git pull --no-rebase
```

To perform a normal merging pull.

Check if fast-forwardable

To only allow fast forwarding the local branch, you can use:

```
git pull --ff-only
```

This will display an error when the local branch is not fast-forwardable, and needs to be either rebased or merged with upstream.

Section 23.6: Pull, "permission denied"

Some problems can occur if the `.git` folder has wrong permission. Fixing this problem by setting the owner of the complete `.git` folder. Sometimes it happen that another user pull and change the rights of the `.git` folder or files.

To fix the problem:

```
chown -R youruser:yourgroup .git/
```

第24章：钩子

第24.1节：推送前钩子

适用于Git 1.8.2及以上版本。

版本 ≥ 1.8

推送前钩子可用于阻止推送操作。这样做的原因包括：阻止意外手动推送到特定分支，或在既定检查失败时（单元测试、语法检查）阻止推送。

预推送钩子只需在.git/hooks/目录下创建一个名为pre-push的文件即可，（注意事项），并确保该文件具有可执行权限：chmod +x ./git/hooks/pre-push。

这是Hannah Wolfe提供的一个示例，用于阻止推送到master分支：

```
#!/bin/bash

protected_branch='master'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*/\(.*\),\1,')

if [ $protected_branch = $current_branch ]
then
    read -p "你正准备推送到master分支，确定要这样做吗？[y|n] " -n 1 -r < /dev/tty
    echo
    if echo $REPLY | grep -E '^[Yy]$' > /dev/null
    then
        exit 0 # 推送将执行
    fi
    exit 1 # 推送将不会执行
else
    exit 0 # 推送将执行
fi
```

这是Volkan Unsal的一个示例，它确保RSpec测试通过后才允许推送：

```
#!/usr/bin/env ruby
require 'pty'
html_path = "rspec_results.html"
begin
PTY.spawn( "rspec spec --format h > rspec_results.html" ) do |stdin, stdout, pid|
  begin
stdin.each { |line| print line }
  rescue Errno::EIO
  end
end
rescue PTY::ChildExited
  puts "子进程退出！"
end

# 查找是否有任何错误
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
errors = html.match(/(\d+) errors/)[0].to_i rescue 0
if errors == 0 then
errors = html.match(/(\d+) failure/)[0].to_i rescue 0
end
pending = html.match(/(\d+) pending/)[0].to_i rescue 0
```


Chapter 24: Hooks

Section 24.1: Pre-push

Available in Git 1.8.2 and above.

Version ≥ 1.8

Pre-push hooks can be used to prevent a push from going though. Reasons this is helpful include: blocking accidental manual pushes to specific branches, or blocking pushes if an established check fails (unit tests, syntax).

A pre-push hook is created by simply creating a file named `pre-push` under `.git/hooks/`, and (**gotcha alert**), making sure the file is executable: `chmod +x .git/hooks/pre-push`.

Here's an example from Hannah Wolfe that blocks a push to master:

```
#!/bin/bash

protected_branch='master'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*/\(.*\),\1,')

if [ $protected_branch = $current_branch ]
then
    read -p "You're about to push master, is that what you intended? [y|n] " -n 1 -r < /dev/tty
    echo
    if echo $REPLY | grep -E '^[Yy]$' > /dev/null
    then
        exit 0 # push will execute
    fi
    exit 1 # push will not execute
else
    exit 0 # push will execute
fi
```

Here's an example from Volkan Unsal which makes sure RSpec tests pass before allowing the push:

```
#!/usr/bin/env ruby
require 'pty'
html_path = "rspec_results.html"
begin
  PTY.spawn( "rspec spec --format h > rspec_results.html" ) do |stdin, stdout, pid|
  begin
    stdin.each { |line| print line }
  rescue Errno::EIO
  end
end
rescue PTY::ChildExited
  puts "Child process exit!"
end

# find out if there were any errors
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
errors = html.match(/(\d+) errors/)[0].to_i rescue 0
if errors == 0 then
  errors = html.match(/(\d+) failure/)[0].to_i rescue 0
end
pending = html.match(/(\d+) pending/)[0].to_i rescue 0
```

```
if errors.zero?
  puts "0 failed! #{examples} run, #{pending} pending"
  # HTML Output when tests ran successfully:
  # puts "View spec results at #{File.expand_path(html_path)}"
  sleep 1
  exit 0
else
  puts "\a提交失败！！"
  puts "在 #{File.expand_path(html_path)} 查看你的 rspec 结果"
  puts
  puts "#{errors} 失败！#{examples} 运行，#{pending} 待处理"
  # 测试失败时打开 HTML 输出
  # `open #{html_path}`
  exit 1
end
```

如你所见，有很多可能性，但核心是如果一切顺利则 exit 0，如果出现问题则 exit 1。每当你 exit 1 时，推送将被阻止，代码将保持在运行 git push... 之前的状态。

使用客户端钩子时，请记住用户可以通过在推送时使用选项 "--no-verify" 来跳过所有客户端钩子。如果你依赖钩子来强制执行流程，可能会吃亏。

文档：https://git-scm.com/docs/githooks#_pre_push
官方示例：
https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample

第24.2节：提交前验证 Maven 构建（或其他构建系统）

.git/hooks/pre-commit

```
#!/bin/sh
if [ -s pom.xml ]; then
    echo "正在运行 mvn verify"
    mvn clean verify
    if [ $? -ne 0 ]; then
        echo "Maven 构建失败"
        exit 1
    fi
fi
```

第24.3节：自动将某些推送转发到其他仓库

post-receive 钩子可以用来自动将接收到的推送转发到另一个仓库。

```
$ cat .git/hooks/post-receive

#!/bin/bash

IFS=' '
while read local_ref local_sha remote_ref remote_sha
do

  echo "$remote_ref" | egrep '^refs\/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
    ref=`echo $remote_ref | \ tsed -e 's/^refs\/heads\///'`
```



```
if errors.zero?
  puts "0 failed! #{examples} run, #{pending} pending"
  # HTML Output when tests ran successfully:
  # puts "View spec results at #{File.expand_path(html_path)}"
  sleep 1
  exit 0
else
  puts "\aCOMMIT FAILED!!"
  puts "View your rspec results at #{File.expand_path(html_path)}"
  puts
  puts "#{errors} failed! #{examples} run, #{pending} pending"
  # Open HTML Ooutput when tests failed
  # `open #{html_path}`
  exit 1
end
```

As you can see, there are lots of possibilities, but the core piece is to `exit 0` if good things happened, and `exit 1` if bad things happened. Anytime you `exit 1` the push will be prevented and your code will be in the state it was before running `git push...`.

When using client side hooks, keep in mind that users can skip all client side hooks by using the option "--no-verify" on a push. If you're relying on the hook to enforce process, you can get burned.

Documentation: https://git-scm.com/docs/githooks#_pre_push
Official Sample:
https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample

Section 24.2: Verify Maven build (or other build system) before committing

.git/hooks/pre-commit

```
#!/bin/sh
if [ -s pom.xml ]; then
    echo "Running mvn verify"
    mvn clean verify
    if [ $? -ne 0 ]; then
        echo "Maven build failed"
        exit 1
    fi
fi
```

Section 24.3: Automatically forward certain pushes to other repositories

post-receive hooks can be used to automatically forward incoming pushes to another repository.

```
$ cat .git/hooks/post-receive

#!/bin/bash

IFS=' '
while read local_ref local_sha remote_ref remote_sha
do

  echo "$remote_ref" | egrep '^refs\/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
    ref=`echo $remote_ref | sed -e 's/^refs\/heads\///'`
```

```
    echo 正在将功能分支转发到其他仓库： $ref
    git push -q --force other_repos $ref
  }

完成
```

在此示例中，egrep 正则表达式查找特定的分支格式（此处为用于命名 Jira 任务的 JIRA-12345）。当然，如果你想转发所有分支，可以省略这部分。

第24.4节：Commit-msg

此钩子类似于prepare-commit-msg钩子，但它在用户输入提交信息之后调用，而不是之前。通常用于警告开发者提交信息格式不正确。

传递给此钩子的唯一参数是包含提交信息的文件名。如果你不喜欢用户输入的信息，可以直接修改该文件（与prepare-commit-msg相同），或者通过非零状态退出来完全中止提交。

以下示例用于检查提交信息中是否包含单词 ticket 后跟数字

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)

if [[ -z $isPresent ]]
  then echo "提交信息错误，$word 缺失"; exit 1;
  else echo "提交信息正确"; exit 0;
fi
```

第24.5节：本地钩子

本地钩子只影响它们所在的本地仓库。每个开发者都可以修改自己的本地钩子，因此它们不能被可靠地用作强制提交策略的手段。它们的设计目的是让开发者更容易遵守某些指导原则，避免将来可能出现的问题。

本地钩子有六种类型：pre-commit、prepare-commit-msg、commit-msg、post-commit、post-checkout和pre-rebase。

前四种钩子与提交相关，允许你对提交生命周期的每个部分进行一定的控制。
最后两种钩子让你可以对git checkout和git rebase命令执行一些额外操作或安全检查。

所有“pre-”钩子允许你修改即将执行的操作，而“post-”钩子主要用于通知。

第24.6节：post-checkout

这个钩子的工作方式类似于post-commit钩子，但它在你成功使用git checkout检出一个引用时被调用。它可以作为一个有用的工具，用于清理工作目录中那些自动生成的文件，避免引起混淆。

该钩子接受三个参数：

1. 上一个HEAD的引用，
2. 新的 HEAD 的引用，和
3. 一个标志，指示这是分支检出还是文件检出（分别为1或0）。


```
    echo Forwarding feature branch to other repository: $ref
    git push -q --force other_repos $ref
  }

done
```

In this example, the `egrep` regexp looks for a specific branch format (here: JIRA-12345 as used to name Jira issues). You can leave this part off if you want to forward all branches, of course.

Section 24.4: Commit-msg

This hook is similar to the `prepare-commit-msg` hook, but it's called after the user enters a commit message rather than before. This is usually used to warn developers if their commit message is in an incorrect format.

The only argument passed to this hook is the name of the file that contains the message. If you don't like the message that the user has entered, you can either alter this file in-place (same as `prepare-commit-msg`) or you can abort the commit entirely by exiting with a non-zero status.

The following example is used to check if the word ticket followed by a number is present on the commit message

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)

if [[ -z $isPresent ]]
  then echo "Commit message KO, $word is missing"; exit 1;
  else echo "Commit message OK"; exit 0;
fi
```

Section 24.5: Local hooks

Local hooks affect only the local repositories in which they reside. Each developer can alter their own local hooks, so they can't be used reliably as a way to enforce a commit policy. They are designed to make it easier for developers to adhere to certain guidelines and avoid potential problems down the road.

There are six types of local hooks: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout, and pre-rebase.

The first four hooks relate to commits and allow you to have some control over each part in a commit's life cycle. The final two let you perform some extra actions or safety checks for the git checkout and git rebase commands.

All of the "pre-" hooks let you alter the action that's about to take place, while the "post-" hooks are used primarily for notifications.

Section 24.6: Post-checkout

This hook works similarly to the `post-commit` hook, but it's called whenever you successfully check out a reference with `git checkout`. This could be a useful tool for clearing out your working directory of auto-generated files that would otherwise cause confusion.

This hook accepts three parameters:

1. the ref of the previous HEAD,
2. the ref of the new HEAD, and
3. a flag indicating if it was a branch checkout or a file checkout (`1` or `0`, respectively).

其退出状态不会影响git checkout命令。

第24.7节：提交后钩子

此钩子在commit-msg钩子之后立即调用。它不能改变`git commit`操作的结果，因此主要用于通知目的。

该脚本不接受参数，其退出状态不会以任何方式影响提交。

第24.8节：接收后钩子

此钩子在成功推送操作后调用。它通常用于通知目的。

该脚本不接受参数，但通过标准输入接收与pre-receive相同的信息：

```
<old-value> <new-value> <ref-name>
```

第24.9节：预提交

每次运行`git commit`时都会执行此钩子，用于验证即将提交的内容。您可以使用此钩子检查即将提交的快照。

此类钩子适用于运行自动化测试，以确保即将提交的内容不会破坏项目的现有功能。此类钩子还可以检查空白字符或行尾（EOL）错误。

pre-commit 脚本不会接收任何参数，且以非零状态退出会中止整个提交过程。

第24.10节：Prepare-commit-msg

此钩子在`pre-commit`钩子之后调用，用于在文本编辑器中填充提交信息。通常用于修改自动生成的压缩（squash）或合并（merge）提交信息。

此钩子会接收一到三个参数：

- 包含提交信息的临时文件名。
- 提交类型，可能是
  - message（`-m`或`-F`选项）、
  - template（`-t`选项）、
  - merge（如果是合并提交），或
  - squash（如果是压缩其他提交）。
- 相关提交的 SHA1 哈希值。仅在使用-c、-C或--amend选项时提供。

类似于pre-commit，退出时返回非零状态会中止提交。

第24.11节：预变基（Pre-rebase）

此钩子在git rebase开始更改代码结构之前调用。此钩子通常用于确保变基操作是合适的。

此钩子接受2个参数：

1. 系列分支分叉的上游分支，
2. 正在变基的分支（变基当前分支时为空）。

Its exit status has no affect on the `git checkout` command.

Section 24.7: Post-commit

This hook is called immediately after the `commit-msg` hook. It cannot alter the outcome of the `git commit` operation, therefore it's used primarily for notification purposes.

The script takes no parameters, and its exit status does not affect the commit in any way.

Section 24.8: Post-receive

This hook is called after a successful push operation. It is typically used for notification purposes.

The script takes no parameters, but is sent the same information as `pre-receive` via standard input:

```
<old-value> <new-value> <ref-name>
```

Section 24.9: Pre-commit

This hook is executed every time you run `git commit`, to verify what is about to be committed. You can use this hook to inspect the snapshot that is about to be committed.

This type of hook is useful for running automated tests to make sure the incoming commit doesn't break existing functionality of your project. This type of hook may also check for whitespace or EOL errors.

No arguments are passed to the pre-commit script, and exiting with a non-zero status aborts the entire commit.

Section 24.10: Prepare-commit-msg

This hook is called after the `pre-commit` hook to populate the text editor with a commit message. This is typically used to alter the automatically generated commit messages for squashed or merged commits.

One to three arguments are passed to this hook:

- The name of a temporary file that contains the message.
- The type of commit, either
  - message (`-m` or `-F` option),
  - template (`-t` option),
  - merge (if it's a merge commit), or
  - squash (if it's squashing other commits).
- The SHA1 hash of the relevant commit. This is only given if `-c`, `-C`, or `--amend` option was given.

Similar to `pre-commit`, exiting with a non-zero status aborts the commit.

Section 24.11: Pre-rebase

This hook is called before `git rebase` begins to alter code structure. This hook is typically used for making sure a rebase operation is appropriate.

This hook takes 2 parameters:

1. the upstream branch that the series was forked from, and
2. the branch being rebased (empty when rebasing the current branch).

你可以通过退出时返回非零状态来中止变基操作。

第24.12节：预接收（Pre-receive）

此钩子在每次有人使用git push将提交推送到仓库时执行。它始终位于推送目标的远程仓库中，而不在发起推送的（本地）仓库中。

该钩子在任何引用更新之前运行。它通常用于执行任何类型的开发策略。

脚本不接受参数，但每个被推送的引用会以以下格式通过标准输入的单独一行传递给脚本：

```
<old-value> <new-value> <ref-name>
```

第24.13节：更新

此钩子在pre-receive之后调用，工作方式相同。它在实际更新任何内容之前调用，但是针对每个被推送的引用单独调用，而不是一次性针对所有引用调用。

此钩子接受以下3个参数：

- 被更新的引用名称，
- 引用中存储的旧对象名称，和
- 引用中存储的新对象名称。

这些信息与传递给pre-receive的信息相同，但由于update是针对每个引用单独调用的，您可以拒绝某些引用，同时允许其他引用。

You can abort the rebase operation by exiting with a non-zero status.

Section 24.12: Pre-receive

This hook is executed every time somebody uses `git push` to push commits to the repository. It always resides in the remote repository that is the destination of the push and not in the originating (local) repository.

The hook runs before any references are updated. It is typically used to enforce any kind of development policy.

The script takes no parameters, but each ref that is being pushed is passed to the script on a separate line on standard input in the following format:

```
<old-value> <new-value> <ref-name>
```

Section 24.13: Update

This hook is called after `pre-receive`, and it works the same way. It's called before anything is actually updated, but is called separately for each ref that was pushed rather than all of the refs at once.

This hook accepts the following 3 arguments:

- name of the ref being updated,
- old object name stored in the ref, and
- new object name stored in the ref.

This is the same information passed to `pre-receive`, but since `update` is invoked separately for each ref, you can reject some refs while allowing others.

第25章：克隆仓库

第25.1节：浅克隆

克隆一个庞大的仓库（例如包含多年历史的项目）可能需要很长时间，或者因为需要传输的数据量过大而失败。在不需要完整历史的情况下，您可以进行浅克隆：

```
git clone [repo_url] --depth 1
```

上述命令将只从远程仓库获取最后一次提交。

请注意，您可能无法在浅仓库中解决合并问题。通常建议至少获取您需要回溯以解决合并的提交数量。例如，要获取最近的50次提交：

```
git clone [repo_url] --depth 50
```

之后，如有需要，可以获取仓库的其余部分：

版本 ≥ 1.8.3

```
git fetch --unshallow      # 等同于 git fetch —depth=2147483647
                           # 获取仓库的其余部分
```

版本 < 1.8.3

```
git fetch --depth=1000     # 获取最近的1000次提交
```

第25.2节：常规克隆

要下载整个仓库，包括完整历史和所有分支，请输入：

```
git clone <url>
```

上述示例会将其放置在与仓库名称相同的目录中。

要下载仓库并将其保存到指定目录，请输入：

```
git clone <url> [directory]
```

更多详情，请访问 克隆仓库。

第25.3节：克隆特定分支

要克隆仓库的特定分支，请在仓库URL前输入 --branch <分支名称>：

```
git clone --branch <分支名称> <url> [directory]
```

要使用 --branch 的简写选项，请输入 -b。该命令会下载整个仓库并检出 **<branch name>**。

为了节省磁盘空间，你可以只克隆通向单个分支的历史，命令如下：

```
git clone --branch <branch_name> --single-branch <url> [directory]
```


Chapter 25: Cloning Repositories

Section 25.1: Shallow Clone

Cloning a huge repository (like a project with multiple years of history) might take a long time, or fail because of the amount of data to be transferred. In cases where you don't need to have the full history available, you can do a shallow clone:

```
git clone [repo_url] --depth 1
```

The above command will fetch just the last commit from the remote repository.

Be aware that you may not be able to resolve merges in a shallow repository. It's often a good idea to take at least as many commits are you are going to need to backtrack to resolve merges. For example, to instead get the last 50 commits:

```
git clone [repo_url] --depth 50
```

Later, if required, you can the fetch the rest of the repository:

Version ≥ 1.8.3

```
git fetch --unshallow      # equivalent of git fetch —depth=2147483647
                           # fetches the rest of the repository
```

Version < 1.8.3

```
git fetch --depth=1000     # fetch the last 1000 commits
```

Section 25.2: Regular Clone

To download the entire repository including the full history and all branches, type:

```
git clone <url>
```

The example above will place it in a directory with the same name as the repository name.

To download the repository and save it in a specific directory, type:

```
git clone <url> [directory]
```

For more details, visit Clone a repository.

Section 25.3: Clone a specific branch

To clone a specific branch of a repository, type `--branch <branch name>` before the repository url:

```
git clone --branch <branch name> <url> [directory]
```

To use the shorthand option for `--branch`, type `-b`. This command downloads entire repository and checks out **<branch** `name`**>**.

To save disk space you can clone history leading only to single branch with:

```
git clone --branch <branch_name> --single-branch <url> [directory]
```

如果命令中未添加 --single-branch，所有分支的历史都会被克隆到 [directory]。这在大型仓库中可能会成为问题。

要稍后撤销--single-branch标志并获取仓库的其余部分，请使用以下命令：

```
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*"
git fetch origin
```

第25.4节：递归克隆

版本 ≥ 1.6.5

```
git clone <url> --recursive
```

克隆仓库的同时也会克隆所有子模块。如果子模块本身还包含其他子模块，Git也会克隆那些子模块。

第25.5节：使用代理克隆

如果需要在代理环境下使用git下载文件，设置系统范围的代理服务器可能不够。你也可以尝试以下方法：

```
git config --global http.proxy http://<proxy-server>:<port>/
```


If `--single-branch` is not added to the command, history of all branches will be cloned into `[directory]`. This can be issue with big repositories.

To later undo `--single-branch` flag and fetch the rest of repository use command:

```
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*"
git fetch origin
```

Section 25.4: Clone recursively

Version ≥ 1.6.5

```
git clone <url> --recursive
```

Clones the repository and also clones all submodules. If the submodules themselves contain additional submodules, Git will also clone those.

Section 25.5: Clone using a proxy

If you need to download files with git under a proxy, setting proxy server system-wide couldn't be enough. You could also try the following:

```
git config --global http.proxy http://<proxy-server>:<port>/
```

第26章：暂存

参数	详情
显示	显示存储的更改，作为存储状态与其原始父状态之间的差异。当未指定 <stash> 时，显示最新的存储。
列表	列出您当前拥有的存储。每个存储都会列出其名称（例如 stash@{0} 是最新的存储，stash@{1} 是之前的一个，依此类推）、创建存储时所在的分支名称，以及存储所基于的提交的简短描述。
弹出	从存储列表中移除单个存储状态，并将其应用到当前工作树状态之上。
应用	类似于弹出，但不从存储列表中移除该状态。
清除	移除所有存储状态。请注意，这些状态随后将被修剪，可能无法恢复。
丢弃	从存储列表中移除单个存储的状态。当未指定 <stash> 时，移除最新的一个。即 stash@{0}，否则 <stash> 必须是形式为 stash@{<revision>} 的有效 stash 日志引用。
创建	创建一个暂存（这是一个普通的提交对象）并返回其对象名称，但不进行存储在 ref 命名空间中的任何位置。这旨在对脚本有用。它可能不是你想使用的命令；请参见上面的“save”。
商店	将通过 git stash create 创建的给定暂存（这是一个悬挂的合并提交）存储到暂存引用中，并更新暂存引用日志。此命令旨在对脚本有用。它可能不是你想使用的命令；请参见上面的“save”。

第26.1节：什么是暂存？

当你在一个项目上工作时，可能正处于功能分支更改的中途，这时主分支出现了一个错误。你还没准备好提交代码，但也不想丢失你的更改。这时，gitstash 就派上用场了。

在分支上运行git status以显示您未提交的更改：

```
(master) $ git status
当前分支是 master
您的分支与 'origin/master' 是同步的。
未暂存以供提交的更改：
  (使用 "git add <file>..." 来更新将要提交的内容)
  (使用 "git checkout -- <file>..." 来放弃工作目录中的更改)

修改：   business/com/test/core/actions/Photo.c

没有添加到提交的更改 (使用 "git add" 和/或 "git commit -a")
```

然后运行 git stash 将这些更改保存到堆栈中：

```
(master) $ git stash
已保存工作目录和索引状态，WIP 在 master 上：
2f2a6e1 合并拉取请求 #1 来自 test/test-branch
HEAD 现在位于 2f2a6e1 合并拉取请求 #1 来自 test/test-branch
```

如果你已经向工作目录添加了文件，这些文件也可以被暂存。你只需要先将它们暂存（stage）即可。

```
(master) $ git stash
已保存工作目录和索引状态，WIP 在 master 上：
(master) $ git status
当前分支为 master
未跟踪的文件：
  (使用 "git add <file>..." 将文件包含到将要提交的内容中)
```


Chapter 26: Stashing

Parameter	Details
show	Show the changes recorded in the stash as a diff between the stashed state and its original parent. When no <stash> is given, shows the latest one.
list	List the stashes that you currently have. Each stash is listed with its name (e.g. stash@{0} is the latest stash, stash@{1} is the one before, etc.), the name of the branch that was current when the stash was made, and a short description of the commit the stash was based on.
pop	Remove a single stashed state from the stash list and apply it on top of the current working tree state.
apply	Like `pop`, but do not remove the state from the stash list.
clear	Remove all the stashed states. Note that those states will then be subject to pruning, and may be impossible to recover.
drop	Remove a single stashed state from the stash list. When no <stash> is given, it removes the latest one. i.e. stash@{0}, otherwise <stash> must be a valid stash log reference of the form stash@{<revision>}.
create	Create a stash (which is a regular commit object) and return its object name, without storing it anywhere in the ref namespace. This is intended to be useful for scripts. It is probably not the command you want to use; see "save" above.
store	Store a given stash created via git stash create (which is a dangling merge commit) in the stash ref, updating the stash reflog. This is intended to be useful for scripts. It is probably not the command you want to use; see "save" above.

Section 26.1: What is Stashing?

When working on a project, you might be half-way through a feature branch change when a bug is raised against master. You're not ready to commit your code, but you also don't want to lose your changes. This is where **`git stash`** comes in handy.

Run **`git status`** on a branch to show your uncommitted changes:

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/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:   business/com/test/core/actions/Photo.c

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

Then run **`git stash`** to save these changes to a stack:

```
(master) $ git stash
Saved working directory and index state WIP on master:
2f2a6e1 Merge pull request #1 from test/test-branch
HEAD is now at 2f2a6e1 Merge pull request #1 from test/test-branch
```

If you have added files to your working directory these can be stashed as well. You just need to stage them first.

```
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
        NewPhoto.c

没有要提交的内容，但存在未跟踪的文件 (使用 "git add" 来跟踪)
(master) $ git stage NewPhoto.c
(master) $ git stash
已保存工作目录和索引状态，WIP 在 master 上：
(master) $ git status
当前分支为 master
没有要提交的内容，工作树干净
(master) $
```

你的工作目录现在已经没有任何更改。你可以通过重新运行 git status 来查看：

```
(master) $ git status
当前分支是 master
您的分支与 'origin/master' 是同步的。
没有要提交的内容，工作目录干净
```

要应用最新的存储，运行 git stash apply（此外，你还可以使用 git stash pop 来应用并移除最后一次存储的更改）：

```
(master) $ git stash apply
当前分支 master
您的分支与 'origin/master' 是同步的。
未暂存以供提交的更改：
   (使用 "git add <file>..." 来更新将要提交的内容)
   (使用 "git checkout -- <file>..." 来放弃工作目录中的更改)

修改：   business/com/test/core/actions/Photo.c

没有添加到提交的更改 (使用 "git add" 和/或 "git commit -a")
```

但是请注意，stash 不会记住你之前工作的分支。在上述示例中，用户
是在master分支上执行 stash 的。如果他们切换到dev分支，dev，并运行git stash apply，最后一次的 stash 会被应用到**dev**分支上。

```
(master) $ git checkout -b dev
切换到新分支 'dev'
(dev) $ git stash apply
当前分支 dev
未暂存以供提交的更改：
   (使用 "git add <file>..." 来更新将要提交的内容)
   (使用 "git checkout -- <file>..." 来放弃工作目录中的更改)

修改：   business/com/test/core/actions/Photo.c

没有添加到提交的更改 (使用 "git add" 和/或 "git commit -a")
```

第26.2节：创建 stash

将当前工作目录和索引（也称为暂存区）的状态保存到一个 stash 栈中。

```
git stash
```

要将所有未跟踪的文件包含在 stash 中，请使用--include-untracked或-u标志。

```
git stash --include-untracked
```



```
        NewPhoto.c

nothing added to commit but untracked files present (use "git add" to track)
(master) $ git stage NewPhoto.c
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
nothing to commit, working tree clean
(master) $
```

Your working directory is now clean of any changes you made. You can see this by re-running `git status`:

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

To apply the very last stash, run `git stash apply` (additionally, you can apply *and* remove the last stashed changed with `git stash pop`):

```
(master) $ git stash apply
On branch master
Your branch is up-to-date with 'origin/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:   business/com/test/core/actions/Photo.c

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

Note, however, that stashing does not remember the branch you were working on. In the above examples, the user was stashing on **master**. If they switch to the **dev** branch, **dev**, and run `git stash apply` the last stash is put on the **dev** branch.

```
(master) $ git checkout -b dev
Switched to a new branch 'dev'
(dev) $ git stash apply
On branch dev
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:   business/com/test/core/actions/Photo.c

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

Section 26.2: Create stash

Save the current state of working directory and the index (also known as the staging area) in a stack of stashes.

```
git stash
```

To include all untracked files in the stash use the `--include-untracked` or `-u` flags.

```
git stash --include-untracked
```

为你的 stash 添加消息，以便以后更容易识别

```
git stash save "<whatever message>"
```

要在stash后保持暂存区当前状态，请使用--keep-index或-k参数。

```
git stash --keep-index
```

第26.3节：应用并移除stash

要应用最近的stash并将其从栈中移除，请输入：

```
git stash pop
```

要应用特定的stash并将其从栈中移除，请输入：

```
git stash pop stash@{n}
```

第26.4节：应用stash但不移除

应用最近的stash但不将其从栈中移除

```
git stash apply
```

或者特定的stash

```
git stash apply stash@{n}
```

第26.5节：显示暂存

显示最后一次暂存中保存的更改

```
git stash show
```

或者特定的stash

```
git stash show stash@{n}
```

显示特定暂存中保存的更改内容

```
git stash show -p stash@{n}
```

第26.6节：部分暂存

如果您只想暂存工作区中的部分差异，可以使用部分暂存。

```
git stash -p
```

然后交互式选择要暂存的代码块。

从版本 2.13.0 开始，你也可以避免交互模式，使用新的 push 关键字通过路径规范创建部分暂存。

To include a message with your stash to make it more easily identifiable later

```
git stash save "<whatever message>"
```

To leave the staging area in current state after stash use the `--keep-index` or `-k` flags.

```
git stash --keep-index
```

Section 26.3: Apply and remove stash

To apply the last stash and remove it from the stack - type:

```
git stash pop
```

To apply specific stash and remove it from the stack - type:

```
git stash pop stash@{n}
```

Section 26.4: Apply stash without removing it

Applies the last stash without removing it from the stack

```
git stash apply
```

Or a specific stash

```
git stash apply stash@{n}
```

Section 26.5: Show stash

Shows the changes saved in the last stash

```
git stash show
```

Or a specific stash

```
git stash show stash@{n}
```

To show content of the changes saved for the specific stash

```
git stash show -p stash@{n}
```

Section 26.6: Partial stash

If you would like to stash only *some* diffs in your working set, you can use a partial stash.

```
git stash -p
```

And then interactively select which hunks to stash.

As of version 2.13.0 you can also avoid the interactive mode and create a partial stash with a pathspec using the new **push** keyword.

```
git stash push -m "My partial stash" -- app.config
```

第 26.7 节：列出已保存的暂存

```
git stash list
```

这将按时间倒序列出堆栈中的所有暂存。
你将得到类似如下的列表：

```
stash@{0}: 在 master 上的 WIP：67a4e01 将测试合并到 develop
stash@{1}: 在 master 上的 WIP：70f0d95 用户登录时将用户角色添加到 localStorage
```

你可以通过名称引用特定的暂存，例如 stash@{1}。

第 26.8 节：将你的未完成工作移到另一个分支

如果在工作时发现自己在错误的分支上且尚未创建任何提交，可以使用暂存轻松地将工作移到正确的分支：

```
git stash
git checkout correct-branch
git stash pop
```

请记住，`git stash pop` 会应用最后一次存储并将其从存储列表中删除。若想保留存储在列表中并仅应用到某个分支，可以使用：

```
git stash apply
```

第26.9节：移除暂存

移除所有暂存

```
git stash clear
```

移除最后一个暂存

```
git stash drop
```

或者特定的stash

```
git stash drop stash@{n}
```

第26.10节：使用checkout应用部分暂存内容

你已经创建了一个暂存，并希望只检出该暂存中的部分文件。

```
git checkout stash@{0} -- myfile.txt
```

第26.11节：从暂存恢复早期更改

运行 git stash 后，要获取您最近的 stash，请使用


```
git stash push -m "My partial stash" -- app.config
```

Section 26.7: List saved stashes

```
git stash list
```

This will list all stashes in the stack in reverse chronological order.
You will get a list that looks something like this:

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

You can refer to specific stash by its name, for example `stash@{1}`.

Section 26.8: Move your work in progress to another branch

If while working you realize you're on wrong branch and you haven't created any commits yet, you can easily move your work to correct branch using stashing:

```
git stash
git checkout correct-branch
git stash pop
```

Remember `git stash pop` will apply the last stash and delete it from the stash list. To keep the stash in the list and only apply to some branch you can use:

```
git stash apply
```

Section 26.9: Remove stash

Remove all stash

```
git stash clear
```

Removes the last stash

```
git stash drop
```

Or a specific stash

```
git stash drop stash@{n}
```

Section 26.10: Apply part of a stash with checkout

You've made a stash and wish to checkout only some of the files in that stash.

```
git checkout stash@{0} -- myfile.txt
```

Section 26.11: Recovering earlier changes from stash

To get your most recent stash after running git stash, use

```
git stash apply
```

要查看你的存储列表，请使用

```
git stash list
```

你将得到类似如下的列表

```
stash@{0}: 在 master 上的 WIP：67a4e01 将测试合并到 develop
stash@{1}: 在 master 上的 WIP：70f0d95 用户登录时将用户角色添加到 localStorage
```

使用显示的编号选择要恢复的不同 git 存储

```
git stash apply stash@{2}
```

第26.12节：交互式暂存

暂存会将工作目录的脏状态–即已修改的已跟踪文件和已暂存的更改–保存到一个未完成更改的堆栈中，您可以随时重新应用这些更改。

仅暂存已修改的文件：

假设您不想暂存已暂存的文件，只想暂存已修改的文件，可以使用：

```
git stash --keep-index
```

这将只暂存已修改的文件。

暂存未跟踪的文件：

暂存从不保存未跟踪的文件，它只暂存已修改和已暂存的文件。所以假设您也需要暂存未跟踪的文件，可以使用：

```
git stash -u
```

这将跟踪未跟踪、已暂存和已修改的文件。

仅暂存某些特定更改：

假设您只需要从所有已修改和已暂存的文件中暂存文件的一部分代码或仅部分文件，那么您可以这样操作：

```
git stash --patch
```

Git 不会将所有修改的内容都存入暂存区，而是会以交互方式提示你选择哪些更改想要存入暂存区，哪些想保留在工作目录中。

第26.13节：恢复被丢弃的暂存

如果你刚刚执行了弹出操作且终端仍然打开，屏幕上仍会显示由 `git stash pop` 打印的哈希值：

```
$ git stash pop
```



```
git stash apply
```

To see a list of your stashes, use

```
git stash list
```

You will get a list that looks something like this

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Choose a different git stash to restore with the number that shows up for the stash you want

```
git stash apply stash@{2}
```

Section 26.12: Interactive Stashing

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Stashing only modified files:

Suppose you don't want to stash the staged files and only stash the modified files so you can use:

```
git stash --keep-index
```

Which will stash only the modified files.

Stashing untracked files:

Stash never saves the untracked files it only stashes the modified and staged files. So suppose if you need to stash the untracked files too then you can use this:

```
git stash -u
```

this will track the untracked, staged and modified files.

Stash some particular changes only:

Suppose you need to stash only some part of code from the file or only some files only from all the modified and stashed files then you can do it like this:

```
git stash --patch
```

Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

Section 26.13: Recover a dropped stash

If you have only just popped it and the terminal is still open, you will still have the hash value printed by `git stash pop` on screen:

```
$ git stash pop
```

```
[...]
丢弃的引用/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)
```

（注意 git stash drop 也会产生相同的行。）

否则，你可以使用以下命令找到它：

```
git fsck --no-reflog | awk '/dangling commit/ {print $3}'
```

这将显示你提交图中所有不再被任何分支或标签引用的提交——每一个丢失的提交，包括你曾经创建的每一个暂存提交，都会出现在该图中。

找到你想要的暂存提交最简单的方法可能是将该列表传递给gitk：

```
gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )
```

这将启动一个仓库浏览器，向你展示仓库中曾经存在的每一个提交，无论它是否可达。

如果你更喜欢在控制台上查看漂亮的图形而不是单独的图形界面应用，可以将gitk替换为类似git log --graph --oneline --decorate的命令。

要识别暂存提交，请查找如下形式的提交信息：

WIP on somebranch: commithash 一些旧的提交信息

一旦你知道了想要的提交的哈希值，就可以将其作为暂存应用：

git stash apply sh_hash

或者你也可以使用gitk中的上下文菜单，为任何你感兴趣的不可达提交创建分支。
之后，你可以使用所有常规工具随意处理它们。完成后，只需再次将那些分支吹走即可。


```
[...]
Dropped refs/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)
```

(Note that git stash drop also produces the same line.)

Otherwise, you can find it using this:

```
git fsck --no-reflog | awk '/dangling commit/ {print $3}'
```

This will show you all the commits at the tips of your commit graph which are no longer referenced from any branch or tag – every lost commit, including every stash commit you've ever created, will be somewhere in that graph.

The easiest way to find the stash commit you want is probably to pass that list to `gitk`:

```
gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )
```

This will launch a repository browser showing you *every single commit in the repository ever*, regardless of whether it is reachable or not.

You can replace `gitk` there with something like `git log --graph --oneline --decorate` if you prefer a nice graph on the console over a separate GUI app.

To spot stash commits, look for commit messages of this form:

WIP on *somebranch*: *commithash Some old commit message*

Once you know the hash of the commit you want, you can apply it as a stash:

`git stash apply` sh_hash

Or you can use the context menu in `gitk` to create branches for any unreachable commits you are interested in. After that, you can do whatever you want with them with all the normal tools. When you're done, just blow those branches away again.

第27章：子树

第27.1节：创建、拉取和回溯子树

创建子树

添加一个名为plugin的新远程，指向插件的仓库：

```
git remote add plugin https://path.to/remotes/plugin.git
```

然后创建一个子树，指定新的文件夹前缀为`plugins/demo`。`plugin`是远程名称，`master`指的是子树仓库的主分支：

```
git subtree add --prefix=plugins/demo plugin master
```

拉取子树更新

拉取插件中正常提交的更新：

```
git subtree pull --prefix=plugins/demo plugin master
```

回移子树更新

1. 指定在超级项目中需要回移的提交：

   ```
   git commit -am "new changes to be backported"
   ```

2. 检出用于合并的新分支，设置为跟踪子树仓库：

   ```
   git checkout -b backport plugin/master
   ```

3. 选择性应用回移提交：

   ```
   git cherry-pick -x --strategy=subtree master
   ```

4. 将更改推送回插件源：

   ```
   git push plugin backport:master
   ```


Chapter 27: Subtrees

Section 27.1: Create, Pull, and Backport Subtree

Create Subtree

Add a new remote called `plugin` pointing to the plugin's repository:

```
git remote add plugin https://path.to/remotes/plugin.git
```

Then Create a subtree specifying the new folder prefix `plugins/demo`. `plugin` is the remote name, and `master` refers to the master branch on the subtree's repository:

```
git subtree add --prefix=plugins/demo plugin master
```

Pull Subtree Updates

Pull normal commits made in plugin:

```
git subtree pull --prefix=plugins/demo plugin master
```

Backport Subtree Updates

1. Specify commits made in superproject to be backported:

   ```
   git commit -am "new changes to be backported"
   ```

2. Checkout new branch for merging, set to track subtree repository:

   ```
   git checkout -b backport plugin/master
   ```

3. Cherry-pick backports:

   ```
   git cherry-pick -x --strategy=subtree master
   ```

4. Push changes back to plugin source:

   ```
   git push plugin backport:master
   ```

第28章：重命名

参数	详情
-f 或 --force	即使目标存在，也强制重命名或移动文件

第28.1节：重命名文件夹

将文件夹从oldName重命名为newName

```
git mv directoryToFolder/oldName directoryToFolder/newName
```

然后执行`git commit`和/或`git push`

如果出现此错误：

> fatal: 重命名 'directoryToFolder/oldName' 失败：无效的参数

请使用以下命令：

```
git mv directoryToFolder/oldName temp && git mv temp directoryToFolder/newName
```

第28.2节：重命名本地和远程分支

最简单的方法是先切换到本地分支：

```
git checkout old_branch
```

然后重命名本地分支，删除旧的远程分支，并将新重命名的分支设置为上游分支：

```
git branch -m new_branch
git push origin :old_branch
git push --set-upstream origin new_branch
```

第28.3节：重命名本地分支

您可以使用以下命令重命名本地仓库中的分支：

```
git branch -m old_name new_name
```


Chapter 28: Renaming

Parameter	Details
`-f` or `--force`	Force renaming or moving of a file even if the target exists

Section 28.1: Rename Folders

To rename a folder from `oldName` to `newName`

```
git mv directoryToFolder/oldName directoryToFolder/newName
```

Followed by `git commit` and/or `git push`

If this error occurs:

> fatal: renaming 'directoryToFolder/oldName' failed: Invalid argument

Use the following command:

```
git mv directoryToFolder/oldName temp && git mv temp directoryToFolder/newName
```

Section 28.2: rename a local and the remote branch

the easiest way is to have the local branch checked out:

```
git checkout old_branch
```

then rename the local branch, delete the old remote and set the new renamed branch as upstream:

```
git branch -m new_branch
git push origin :old_branch
git push --set-upstream origin new_branch
```

Section 28.3: Renaming a local branch

You can rename branch in local repository using this command:

```
git branch -m old_name new_name
```

第29章：推送

参数	详情
--force	覆盖远程引用以匹配您的本地引用。 *可能导致远程仓库丢失提交，因此请谨慎使用。*
--verbose	以详细模式运行。
<remote>	推送操作的目标远程仓库。
<refspec>...	指定用哪个本地引用或对象更新远程引用。

更改、暂存并提交代码后，需要推送（push）以使您的更改对他人可见，并将本地更改传输到仓库服务器。本文将介绍如何正确使用 Git 进行代码推送。

第29.1节：将特定对象推送到远程分支

通用语法

```
git push <远程名> <对象>:<远程分支名>
```

示例

```
git push origin master:wip-yourname
```

将您的 master 分支推送到 origin 的 `wip-yourname` 分支（通常是您克隆的仓库）。

删除远程分支

删除远程分支相当于向该分支推送一个空对象。

```
git push <远程名> :<远程分支名>
```

示例

```
git push origin :wip-yourname
```

将远程分支wip-yourname删除

除了使用冒号，你也可以使用--delete标志，在某些情况下这样更易读。

示例

```
git push origin --delete wip-yourname
```

推送单个提交

如果你的分支中只有一个提交想推送到远程，而不推送其他内容，
你可以使用以下命令

```
git push <远程名> <提交SHA>:<远程分支名>
```

示例

假设有如下git历史

Chapter 29: Pushing

Parameter	Details
--force	Overwrites the remote ref to match your local ref. *Can cause the remote repository to lose commits, so use with care*.
--verbose	Run verbosely.
<remote>	The remote repository that is destination of the push operation.
<refspec>...	Specify what remote ref to update with what local ref or object.

After changing, staging, and committing code with Git, pushing is required to make your changes available to others and transfers your local changes to the repository server. This topic will cover how to properly push code using Git.

Section 29.1: Push a specific object to a remote branch

General syntax

```
git push <remotename> <object>:<remotebranchname>
```

Example

```
git push origin master:wip-yourname
```

Will push your master branch to the `wip-yourname` branch of origin (most of the time, the repository you cloned from).

Delete remote branch

Deleting the remote branch is the equivalent of pushing an empty object to it.

```
git push <remotename> :<remotebranchname>
```

Example

```
git push origin :wip-yourname
```

Will delete the remote branch `wip-yourname`

Instead of using the colon, you can also use the --delete flag, which is better readable in some cases.

Example

```
git push origin --delete wip-yourname
```

Push a single commit

If you have a single commit in your branch that you want to push to a remote without pushing anything else, you can use the following

```
git push <remotename> <commit SHA>:<remotebranchname>
```

Example

Assuming a git history like this

```
eeb32bc 提交1 - 已推送
347d700 提交2 - 想推送
e539af8 提交3 - 仅本地
5d339db 提交4 - 仅本地
```

要仅将提交347d700推送到远程master，使用以下命令

```
git push origin 347d700:master
```

第29.2节：推送

```
git push
```

将代码推送到现有的上游仓库。根据推送配置，它会推送当前分支的代码（Git 2.x 的默认行为）或所有分支的代码（Git 1.x 的默认行为）。

指定远程仓库

在使用 git 时，拥有多个远程仓库会很方便。要指定要推送到的远程仓库，只需在命令后附加其名称。

```
git push origin
```

指定分支

要推送到特定分支，比如feature_x：

```
git push origin feature_x
```

设置远程跟踪分支

除非你正在操作的分支最初来自远程仓库，否则首次使用git push命令是无效的。你必须执行以下命令，告诉 git 将当前分支推送到特定的远程/分支组合。

```
git push --set-upstream origin master
```

这里，master 是远程 origin 上的分支名称。你可以使用 -u 作为 --set-upstream 的简写。

推送到新仓库

要推送到尚未创建或为空的仓库：

1. 在 GitHub 上创建仓库（如果适用）
2. 复制给你的 URL，格式为 `https://github.com/用户名/仓库名.git`
3. 进入你的本地仓库，执行 git remote add origin URL
    - 要验证是否添加成功，运行 `git remote -v`
4. 运行 `git push origin master`

你的代码现在应该已经在 GitHub 上了


```
eeb32bc Commit 1 - already pushed
347d700 Commit 2 - want to push
e539af8 Commit 3 - only local
5d339db Commit 4 - only local
```

to push only commit *347d700* to remote *master* use the following command

```
git push origin 347d700:master
```

Section 29.2: Push

```
git push
```

will push your code to your existing upstream. Depending on the push configuration, it will either push code from you current branch (default in Git 2.x) or from all branches (default in Git 1.x).

Specify remote repository

When working with git, it can be handy to have multiple remote repositories. To specify a remote repository to push to, just append its name to the command.

```
git push origin
```

Specify Branch

To push to a specific branch, say `feature_x`:

```
git push origin feature_x
```

Set the remote tracking branch

Unless the branch you are working on originally comes from a remote repository, simply using `git push` won't work the first time. You must perform the following command to tell git to push the current branch to a specific remote/branch combination

```
git push --set-upstream origin master
```

Here, `master` is the branch name on the remote `origin`. You can use `-u` as a shorthand for `--set-upstream`.

Pushing to a new repository

To push to a repository that you haven't made yet, or is empty:

1. Create the repository on GitHub (if applicable)
2. Copy the url given to you, in the form `https://github.com/USERNAME/REPO_NAME.git`
3. Go to your local repository, and execute `git remote add origin URL`
    - To verify it was added, run `git remote -v`
4. Run `git push origin master`

Your code should now be on GitHub

更多信息请参见 添加远程仓库

说明

推送代码意味着 git 会分析你本地提交和远程的差异，并将它们发送到上游进行写入。当推送成功时，你的本地仓库和远程仓库将同步，其他用户可以看到你的提交。

关于“上游”和“下游”概念的更多细节，请参见备注。

第29.3节：强制推送

有时，当你本地的更改与远程的更改不兼容时（即，当你无法快进远程分支，或者远程分支不是你本地分支的直接祖先），唯一推送更改的方法就是强制推送。

```
git push -f
```

或

```
git push --force
```

重要说明

这将覆盖任何远程更改，且你的远程仓库将与本地保持一致。

注意：使用此命令可能导致远程仓库丢失提交。此外，如果你与他人共享此远程仓库，强烈建议不要执行强制推送，因为他们的历史记录将保留所有被覆盖的提交，从而导致他们的工作与远程仓库不同步。

经验法则是，只有在以下情况下才进行强制推送：

- 除了你之外，没有人拉取你试图覆盖的更改你可以强制所有人在强制
- 推送后重新克隆一份新的副本，并让每个人将他们的更改应用到新副本上（这样做可能会让别人讨厌你）。

第29.4节：推送标签

```
git push --tags
```

推送本地仓库中所有远程仓库不存在的git标签。

第29.5节：更改默认推送行为

Current 更新远程仓库中与当前工作分支同名的分支。

```
git config push.default current
```

Simple 推送到上游分支，但如果上游分支名称不同则无法工作。

```
git config push.default simple
```


For more information view Adding a remote repository

Explanation

Push code means that git will analyze the differences of your local commits and remote and send them to be written on the upstream. When push succeeds, your local repository and remote repository are synchronized and other users can see your commits.

For more details on the concepts of "upstream" and "downstream", see Remarks.

Section 29.3: Force Pushing

Sometimes, when you have local changes incompatible with remote changes (ie, when you cannot fast-forward the remote branch, or the remote branch is not a direct ancestor of your local branch), the only way to push your changes is a force push.

```
git push -f
```

or

```
git push --force
```

Important notes

This will **overwrite** any remote changes and your remote will match your local.

Attention: Using this command may cause the remote repository to **lose commits**. Moreover, it is strongly advised against doing a force push if you are sharing this remote repository with others, since their history will retain every overwritten commit, thus rending their work out of sync with the remote repository.

As a rule of thumb, only force push when:

- Nobody except you pulled the changes you are trying to overwrite
- You can force everyone to clone a fresh copy after the forced push and make everyone apply their changes to it (people may hate you for this).

Section 29.4: Push tags

```
git push --tags
```

Pushes all of the `git tags` in the local repository that are not in the remote one.

Section 29.5: Changing the default push behavior

Current updates the branch on the remote repository that shares a name with the current working branch.

```
git config push.default current
```

Simple pushes to the upstream branch, but will not work if the upstream branch is called something else.

```
git config push.default simple
```

Upstream 推送到上游分支，无论其名称是什么。

```
git config push.default upstream
```

Matching 推送所有本地和远程匹配的分支 git config push.default upstream

设置好首选样式后，使用

```
git push
```

更新远程仓库。

Upstream pushes to the upstream branch, no matter what it is called.

```
git config push.default upstream
```

Matching pushes all branches that match on the local and the remote git config push.default upstream

After you've set the preferred style, use

```
git push
```

to update the remote repository.

第30章：内部结构

第30.1节：仓库（Repo）

一个git仓库是一个存储一组文件和目录元数据的磁盘数据结构。

它存在于你项目的`.git/`文件夹中。每次你向git提交数据时，数据都会存储在这里。反过来，`.git/`包含每一个提交记录。

它的基本结构如下：

```
.git/
objects/
    refs/
```

第30.2节：对象（Objects）

git本质上是一个键值存储。当你向git添加数据时，它会构建一个对象，并使用该对象内容的SHA-1哈希作为键。

因此，git中的任何内容都可以通过其哈希值进行查找：

```
git cat-file -p 4bb6f98
```

有4种类型的对象：

- `blob`
- **`tree`**
- `commit`
- `tag`

第30.3节：HEAD引用

`HEAD` 是一个特殊的引用。它总是指向当前对象。

你可以通过查看.git/HEAD文件来查看它当前指向的位置。

通常，HEAD指向另一个引用：

```
$cat .git/HEAD
ref: refs/heads/mainline
```

但它也可以直接指向一个对象：

```
$ cat .git/HEAD
4bb6f98a223abc9345a0cef9200562333
```

这就是所谓的“分离头指针”（detached head）——因为HEAD没有附着（指向）任何ref，而是直接指向一个object。

第30.4节：引用（Refs）

引用（ref）本质上是一个指针。它是一个指向object的名称。例如，

Chapter 30: Internals

Section 30.1: Repo

A **`git repository`** is an on-disk data structure which stores metadata for a set of files and directories.

It lives in your project's `.git/` folder. Every time you commit data to git, it gets stored here. Inversely, `.git/` contains every single commit.

It's basic structure is like this:

```
.git/
    objects/
    refs/
```

Section 30.2: Objects

`git` is fundamentally a key-value store. When you add data to **`git`**, it builds an `object` and uses the SHA-1 hash of the `object`'s contents as a key.

Therefore, any content in **`git`** can be looked up by it's hash:

```
git cat-file -p 4bb6f98
```

There are 4 types of `Object`:

- `blob`
- **`tree`**
- `commit`
- `tag`

Section 30.3: HEAD ref

`HEAD` is a special `ref`. It always points to the current object.

You can see where it's currently pointing by checking the `.git/HEAD` file.

Normally, `HEAD` points to another `ref`:

```
$cat .git/HEAD
ref: refs/heads/mainline
```

But it can also point directly to an `object`:

```
$ cat .git/HEAD
4bb6f98a223abc9345a0cef9200562333
```

This is what's known as a "detached head" - because `HEAD` is not attached to (pointing at) any `ref`, but rather points directly to an `object`.

Section 30.4: Refs

A `ref` is essentially a pointer. It's a name that points to an `object`. For example,

```
“master” --> 1a410e...
```

它们存储在`.git/refs/heads/`目录下的纯文本文件中。

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

这通常被称为branches（分支）。然而，你会注意到在git中并不存在真正的branch——只有ref。

现在，可以通过直接跳转到不同objects的哈希值来纯粹地浏览git。但这将非常不方便。引用（ref）为你提供了一个方便的名称来引用objects。通过名称而不是哈希值来让git定位到特定位置要容易得多。

第30.5节：提交对象

提交（commit）可能是Git用户最熟悉的对象类型，因为他们习惯于使用git commit命令来创建它。

然而，提交并不直接包含任何更改的文件或数据。相反，它主要包含元数据和指向包含提交实际内容的其他对象的指针。

提交包含以下几项内容：

- 树（tree）的哈希值
- 父提交（commit）的哈希值
- 作者姓名/邮箱，提交者姓名/邮箱
- 提交信息

你可以这样查看任何提交的内容：

```
$ git cat-file commit 5bac93
tree 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
commiter Neil Peart <npeart@rush.com>

第一次提交！
```

树

一个非常重要的注意事项是，tree 对象存储了项目中的每一个文件，并且它存储的是完整的文件，而不是差异。这意味着每个 commit 都包含了整个项目的快照*。

**从技术上讲，只有更改过的文件会被存储。但这更多是为了效率的实现细节。从设计角度来看，`commit` 应被视为包含项目完整副本的对象。*

父对象

parent 行包含另一个 commit 对象的哈希值，可以被视为指向“上一个提交”的“父指针”。这隐式地形成了一个称为 commit graph 的提交图。具体来说，它是一个有向无环图（DAG）。


```
"master" --> 1a410e...
```

They are stored in `.git/refs/heads/ in plain text files.

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

This is commonly what are called `branches`. However, you'll note that in **`git`** there is no such thing as a `branch` - only a `ref`.

Now, it's possible to navigate **`git`** purely by jumping around to different `objects` directly by their hashes. But this would be terribly inconvenient. A `ref` gives you a convenient name to refer to `objects` by. It's much easier to ask **`git`** to go to a specific place by name rather than by hash.

Section 30.5: Commit Object

A `commit` is probably the `object` type most familiar to **`git`** users, as it's what they are used to creating with the **`git commit`** commands.

However, the `commit` does not directly contain any changed files or data. Rather, it contains mostly metadata and pointers to other `objects` which contain the actual contents of the `commit`.

A `commit` contains a few things:

- hash of a **`tree`**
- hash of a parent `commit`
- author name/email, commiter name/email
- commit message

You can see the contents of any commit like this:

```
$ git cat-file commit 5bac93
tree 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
commiter Neil Peart <npeart@rush.com>

First commit!
```

Tree

A very important note is that the **`tree`** objects stores EVERY file in your project, and it stores whole files not diffs. This means that each `commit` contains a snapshot of the entire project*.

**Technically, only changed files are stored. But this is more an implementation detail for efficiency. From a design perspective, a `commit` should be considered as containing a complete copy of the project.*

Parent

The `parent` line contains a hash of another `commit` object, and can be thought of as a "parent pointer" that points to the "previous commit". This implicitly forms a graph of commits known as the **commit graph**. Specifically, it's a directed acyclic graph (or DAG).

第30.6节：Tree对象

tree 基本上代表传统文件系统中的文件夹：嵌套的文件或其他文件夹的容器。

tree 包含：

- 0个或多个 blob 对象
- 0个或多个 tree 对象

正如你可以使用 ls 或 dir 来列出文件夹的内容一样，你也可以列出 tree 对象的内容。

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b    .gitignore
100644 blob cc0956f1    Makefile
040000 tree 92e1ca7e    src
...
```

你可以通过先找到commit中tree的哈希值，然后查看该ree，来查找commit中的文件：

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
parent ...
author ...
commiter ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b    .gitignore
100644 blob cc0956f1    Makefile
040000 tree 92e1ca7e    src
...
```

第30.7节：Blob对象

一个blob包含任意的二进制文件内容。通常，它是原始文本，如源代码或博客文章。
但它也可以是PNG文件的字节或其他任何内容。

如果你有一个blob的哈希值，你可以查看它的内容。

```
$ git cat-file -p d429810
package com.example.project

class Foo {
 ...
}
...
```

例如，你可以像上面那样浏览一个ree，然后查看其中的一个blob。

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b    .gitignore
100644 blob cc0956f1    Makefile
040000 tree 92e1ca7e    src
100644 blob cae391ff    Readme.txt

$ git cat-file -p cae391ff
欢迎使用我的项目！这是自述文件
```


Section 30.6: Tree Object

A **tree** basically represents a folder in a traditional filesystem: nested containers for files or other folders.

A **tree** contains:

- 0 or more blob objects
- 0 or more **tree** objects

Just as you can use ls or **dir** to list the contents of a folder, you can list the contents of a **tree** object.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b    .gitignore
100644 blob cc0956f1    Makefile
040000 tree 92e1ca7e    src
...
```

You can look up the files in a commit by first finding the hash of the **tree** in the commit, and then looking at that **tree**:

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
parent ...
author ...
commiter ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b    .gitignore
100644 blob cc0956f1    Makefile
040000 tree 92e1ca7e    src
...
```

Section 30.7: Blob Object

A blob contains arbitrary binary file contents. Commonly, it will be raw text such as source code or a blog article.
But it could just as easily be the bytes of a PNG file or anything else.

If you have the hash of a blob, you can look at it's contents.

```
$ git cat-file -p d429810
package com.example.project

class Foo {
 ...
}
...
```

For example, you can browse a **tree** as above, and then look at one of the blobs in it.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b    .gitignore
100644 blob cc0956f1    Makefile
040000 tree 92e1ca7e    src
100644 blob cae391ff    Readme.txt

$ git cat-file -p cae391ff
Welcome to my project! This is the readmefile
```

...

第30.8节：创建新的提交

git commit 命令执行以下几项操作：

1. 创建 blobs 和 trees 来表示你的项目目录——存储在 .git/objects
2. 创建一个新的 commit 对象，包含你的作者信息、提交信息，以及步骤1中的根 tree——同样存储在 .git/objects
3. 更新 .git/HEAD 中的 HEAD 引用为新创建的 commit 的哈希值

这会导致一个新的项目快照被添加到 git 中，并且与之前的状态相连接。

第30.9节：移动 HEAD

当你对某个提交（通过哈希或引用指定）运行 git checkout 时，你是在告诉 git 让你的工作目录看起来像快照被创建时的样子。

1. 更新工作目录中的文件以匹配 commit 内的 tree
2. 更新HEAD以指向指定的哈希或引用

第30.10节：移动引用

运行 git reset --hard 会将引用移动到指定的哈希值/引用。

将 MyBranch 移动到 b8dc53：

```
$ git checkout MyBranch      # 将 HEAD 移动到 MyBranch
$ git reset --hard b8dc53    # 使 MyBranch 指向 b8dc53
```

第30.11节：创建新引用

运行 git checkout -b <refname> 会创建一个指向当前 commit 的新引用。

```
$ cat .git/head
1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch
1f324a
```


...

Section 30.8: Creating new Commits

The `git commit` command does a few things:

1. Create `blobs` and `trees` to represent your project directory - stored in `.git/objects`
2. Creates a new `commit` object with your author information, commit message, and the root `tree` from step 1 - also stored in `.git/objects`
3. Updates the `HEAD` ref in `.git/HEAD` to the hash of the newly-created `commit`

This results in a new snapshot of your project being added to `git` that is connected to the previous state.

Section 30.9: Moving HEAD

When you run `git checkout` on a commit (specified by hash or ref) you're telling `git` to make your working directory look like how it did when the snapshot was taken.

1. Update the files in the working directory to match the `tree` inside the `commit`
2. Update `HEAD` to point to the specified hash or ref

Section 30.10: Moving refs around

Running `git reset --hard` moves refs to the specified hash/ref.

Moving `MyBranch` to `b8dc53`:

```
$ git checkout MyBranch      # moves HEAD to MyBranch
$ git reset --hard b8dc53    # makes MyBranch point to b8dc53
```

Section 30.11: Creating new Refs

Running `git checkout -b <refname>` will create a new ref that points to the current `commit`.

```
$ cat .git/head
1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch
1f324a
```

第31章：git-tfs

第31.1节：git-tfs 克隆

这将创建一个与项目同名的文件夹，即 /My.Project.Name

```
$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

第31.2节：从裸git仓库克隆git-tfs

从git仓库克隆比直接从TFVS克隆快十倍，并且在团队环境中效果良好。至少一名团队成员需要先通过常规的git-tfs克隆创建裸git仓库。然后可以引导新仓库与TFVS协同工作。

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

第31.3节：通过Chocolatey安装git-tfs

以下假设您将使用kdiff3进行文件差异比较，虽然不是必需的，但这是个好主意。

```
C:\> choco install kdiff3
```

可以先安装Git，这样您可以设置任何所需参数。这里还安装了所有Unix工具，'NoAutoCrlf'表示检出时保持原样，提交时保持原样。

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'
```

这就是你安装 git-tfs 通过 chocolatey 所真正需要的全部内容。

```
C:\> choco install git-tfs
```

第31.4节：git-tfs 签入

启动 TFVS 的签入对话框。

```
$ git tfs checkintool
```

这将把你所有的本地提交合并成一次单独的签入。

第31.5节：git-tfs 推送

将所有本地提交推送到 TFVS 远程仓库。

```
$ git tfs rcheckin
```

注意：如果需要签入备注，此操作将失败。可以通过在提交信息中添加 `git-tfs-force: rcheckin` 来绕过此限制。

Chapter 31: git-tfs

Section 31.1: git-tfs clone

This will create a folder with the same name as the project, i.e. /My.Project.Name

```
$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

Section 31.2: git-tfs clone from bare git repository

Cloning from a git repository is ten times faster than cloning directly from TFVS and works well in a team environment. At least one team member will have to create the bare git repository by doing the regular git-tfs clone first. Then the new repository can be bootstrapped to work with TFVS.

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

Section 31.3: git-tfs install via Chocolatey

The following assumes you will use kdiff3 for file diffing and although not essential it is a good idea.

```
C:\> choco install kdiff3
```

Git can be installed first so you can state any parameters you wish. Here all the Unix tools are also installed and 'NoAutoCrlf' means checkout as is, commit as is.

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'
```

This is all you really need to be able to install git-tfs via chocolatey.

```
C:\> choco install git-tfs
```

Section 31.4: git-tfs Check In

Launch the Check In dialog for TFVS.

```
$ git tfs checkintool
```

This will take all of your local commits and create a single check-in.

Section 31.5: git-tfs push

Push all local commits to the TFVS remote.

```
$ git tfs rcheckin
```

Note: this will fail if Check-in Notes are required. These can be bypassed by adding `git-tfs-force: rcheckin` to the commit message.

第32章：Git中的空目录

第32.1节：Git不跟踪目录

假设你已经初始化了一个项目，目录结构如下：

```
/build
app.js
```

然后你添加了到目前为止创建的所有内容并提交：

```
git init
git add .
git commit -m "Initial commit"
```

Git只会跟踪文件app.js。

假设你为应用程序添加了构建步骤，并依赖"build"目录作为输出目录（且你不想让每个开发者都必须遵循这个设置说明），一种惯例是在该目录中包含一个".gitkeep"文件，让Git跟踪该文件。

```
/build
  .gitkeep
app.js
```

然后添加这个新文件：

```
git add build/.gitkeep
git commit -m "保留构建目录"
```

Git 现在会跟踪文件 build/.gitkeep，因此构建文件夹将在检出时可用。

再次说明，这只是一个约定，而不是 Git 的功能。

Chapter 32: Empty directories in Git

Section 32.1: Git doesn't track directories

Assume you've initialized a project with the following directory structure:

```
/build
app.js
```

Then you add everything so you've created so far and commit:

```
git init
git add .
git commit -m "Initial commit"
```

Git will only track the file app.js.

Assume you added a build step to your application and rely on the "build" directory to be there as the output directory (and you don't want to make it a setup instruction every developer has to follow), a *convention* is to include a ".gitkeep" file inside the directory and let Git track that file.

```
/build
  .gitkeep
app.js
```

Then add this new file:

```
git add build/.gitkeep
git commit -m "Keep the build directory around"
```

Git will now track the file build/.gitkeep file and therefore the build folder will be made available on checkout.

Again, this is just a convention and not a Git feature.

第33章：git-svn

第33.1节：克隆SVN仓库

你需要使用以下命令创建仓库的新本地副本

```
git svn clone SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b
BRANCHES_REPO_PATH
```

如果你的SVN仓库遵循标准布局（trunk、branches、tags文件夹），你可以少输入一些内容：

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

git svn clone 会逐个检出每个SVN修订版本，并在你的本地仓库中进行git提交，以重建历史记录。如果SVN仓库有很多提交，这将花费一些时间。

命令完成后，你将拥有一个完整的git仓库，带有一个名为master的本地分支，该分支跟踪SVN仓库中的trunk分支。

第33.2节：将本地更改推送到SVN

命令

```
git svn dcommit
```

将为你的每个本地git提交创建一个SVN修订版本。与SVN一样，你的本地git历史必须与SVN仓库中的最新更改保持同步，因此如果命令失败，先尝试执行git svn rebase。

第33.3节：本地工作

只需像使用普通git仓库一样使用你的本地git仓库，使用常规git命令：

- git add FILE 和 git checkout -- FILE 用于暂存/取消暂存文件git commi
- t 用于保存更改。这些提交将是本地的，不会“推送”到SVN仓库，就像在普通git仓库中一样

- `git stash` 和 `git stash pop` 允许使用暂存区
- `git reset HEAD --hard` 撤销所有本地更改
- `git log` 访问仓库中的所有历史记录
- `git rebase -i` 以便你可以自由重写本地历史
- `git branch` 和 `git checkout` 用于创建本地分支

正如git-svn文档所述，“Subversion是一个远不如Git复杂的系统”，因此你不能在不破坏Subversion服务器历史的情况下使用git的全部功能。幸运的是规则非常简单：保持历史线性

这意味着你几乎可以执行任何 git 操作：创建分支、删除/重新排序/合并提交、移动历史记录、删除提交等。除了合并操作。如果你需要重新整合本地分支的历史，请使用git rebase。

当你执行合并时，会创建一个合并提交。合并提交的特别之处在于它有两个父提交，这使得历史记录变成非线性。在你将合并提交“推送”到仓库时，非线性历史会让SVN感到困惑。

不过不用担心：如果你将git合并提交“推送”到SVN，不会破坏任何东西。如果你这么做，

Chapter 33: git-svn

Section 33.1: Cloning the SVN repository

You need to create a new local copy of the repository with the command

```
git svn clone SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b
BRANCHES_REPO_PATH
```

If your SVN repository follows the standard layout (trunk, branches, tags folders) you can save some typing:

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` checks out each SVN revision, one by one, and makes a git commit in your local repository in order to recreate the history. If the SVN repository has a lot of commits this will take a while.

When the command is finished you will have a full fledged git repository with a local branch called master that tracks the trunk branch in the SVN repository.

Section 33.2: Pushing local changes to SVN

The command

```
git svn dcommit
```

will create a SVN revision for each of your local git commits. As with SVN, your local git history must be in sync with the latest changes in the SVN repository, so if the command fails, try performing a `git svn rebase` first.

Section 33.3: Working locally

Just use your local git repository as a normal git repo, with the normal git commands:

- `git add` FILE and `git checkout --` FILE To stage/unstage a file
- `git commit` To save your changes. Those commits will be local and will not be "pushed" to the SVN repo, just like in a normal git repository
- `git stash` and `git stash` pop Allows using stashes
- `git reset HEAD --hard` Revert all your local changes
- `git log` Access all the history in the repository
- `git rebase -i` so you can rewrite your local history freely
- `git branch` and `git checkout` to create local branches

As the git-svn documentation states "Subversion is a system that is far less sophisticated than Git" so you can't use all the full power of git without messing up the history in the Subversion server. Fortunately the rules are very simple: **Keep the history linear**

This means you can make almost any git operation: creating branches, removing/reordering/squashing commits, move the history around, delete commits, etc. Anything *but merges*. If you need to reintegrate the history of local branches use `git rebase` instead.

When you perform a merge, a merge commit is created. The particular thing about merge commits is that they have two parents, and that makes the history non-linear. Non-linear history will confuse SVN in the case you "push" a merge commit to the repository.

However do not worry: **you won't break anything if you "push" a git merge commit to SVN**. If you do so, when

当git合并提交发送到svn服务器时，它将包含该合并所有提交的所有更改，因此你会丢失那些提交的历史，但不会丢失代码中的更改。

第33.4节：从SVN获取最新更改

相当于git pull的命令是

```
git svn rebase
```

该命令会从SVN仓库检索所有更改，并将它们应用到你当前分支的本地提交之上。

你也可以使用命令

```
git svn fetch
```

从SVN仓库检索更改并将其带到本地机器，但不会将其应用到你的本地分支。

第33.5节：处理空文件夹

git 不识别文件夹的概念，它只处理文件及其文件路径。这意味着 git 不会跟踪空文件夹。然而，SVN 会。使用 git-svn 时，默认情况下，你在 git 中对空文件夹所做的任何更改都不会传播到 SVN。

在提交时使用--rmdir标志可以解决此问题，如果你在本地删除了空文件夹中的最后一个文件，SVN 中的空文件夹也会被删除：

```
git svn dcommit --rmdir
```

不幸的是，它不会删除已存在的空文件夹：你需要手动删除。

为了避免每次执行 dcommit 时都添加该标志，或者如果你使用的是 git GUI 工具（如 SourceTree）想要确保安全，可以通过以下命令将此行为设置为默认：

```
git config --global svn.rmdir true
```

这会修改你的 .gitconfig 文件，并添加以下内容：

```
[svn]
rmdir = true
```

要删除所有未被跟踪且应保持为空以供SVN使用的文件和文件夹，请使用以下git命令：

```
git clean -fd
```

请注意：之前的命令将删除所有未被跟踪的文件和空文件夹，即使是那些应该被SVN跟踪的文件夹！如果您需要重新生成被SVN跟踪的空文件夹，请使用以下命令

```
git svn mkdirs
```

实际上，这意味着如果您想清理工作区中的未跟踪文件和文件夹，应该始终使用这两个命令来重新创建被SVN跟踪的空文件夹：

```
git clean -fd && git svn mkdirs
```


the git merge commit is sent to the svn server it will contain all the changes of all commits for that merge, so you will lose the history of those commits, but not the changes in your code.

Section 33.4: Getting the latest changes from SVN

The equivalent to `git pull` is the command

```
git svn rebase
```

This retrieves all the changes from the SVN repository and applies them *on top* of your local commits in your current branch.

You can also use the command

```
git svn fetch
```

to retrieve the changes from the SVN repository and bring them to your local machine but without applying them to your local branch.

Section 33.5: Handling empty folders

git does not recognice the concept of folders, it just works with files and their filepaths. This means git does not track empty folders. SVN, however, does. Using git-svn means that, by default, *any change you do involving empty folders with git will not be propagated to SVN*.

Using the `--rmdir` flag when issuing a comment corrects this issue, and removes an empty folder in SVN if you locally delete the last file inside it:

```
git svn dcommit --rmdir
```

Unfortunately **it does not removes existing empty folders**: you need to do it manually.

To avoid adding the flag each time you do a dcommit, or to play it safe if you are using a git GUI tool (like SourceTree) you can set this behaviour as default with the command:

```
git config --global svn.rmdir true
```

This changes your .gitconfig file and adds these lines:

```
[svn]
rmdir = true
```

To remove all untracked files and folders that should be kept empty for SVN use the git command:

```
git clean -fd
```

Please note: the previous command will remove all untracked files and empty folders, even the ones that should be tracked by SVN! If you need to generate againg the empty folders tracked by SVN use the command

```
git svn mkdirs
```

In practices this means that if you want to cleanup your workspace from untracked files and folders you should always use both commands to recreate the empty folders tracked by SVN:

```
git clean -fd && git svn mkdirs
```

第34章：归档

参数	详情
--format=<fmt>	生成归档的格式：tar 或 zip。如果未指定此选项且指定了输出文件，则格式将根据文件名推断（如果可能）。否则，默认使用 tar。
-l, --list	显示所有可用格式。
-v, --verbose	将进度报告输出到标准错误。
--prefix=<prefix>/	在归档中的每个文件名前加上 <prefix>/。
-o <file>, --output=<file>	将归档写入 <file>，而不是标准输出。
--worktree-attributes	在工作树中的.gitattributes文件中查找属性。
<extra>	这可以是归档后端理解的任何选项。对于zip后端，使用-0将存储文件而不进行压缩，而-1到-9可以用来调整压缩速度和比率。
--remote=<repo>	从远程仓库<repo>检索tar归档文件，而不是本地仓库。
--exec=<git-upload-archive>	与 --remote 一起使用，用于指定远程的 <git-upload-archive 路径。
<tree-ish>	用于生成归档的树或提交。
<path>	如果没有可选参数，归档中将包含当前工作目录中的所有文件和目录。如果指定了一个或多个路径，则只包含这些路径。

第34.1节：创建git仓库的归档

使用 git archive 可以创建仓库的压缩归档，例如用于发布分发。

创建当前 HEAD 版本的tar归档：

```
git archive --format tar HEAD | cat > archive-HEAD.tar
```

创建当前 HEAD 版本的gzip压缩tar归档：

```
git archive --format tar HEAD | gzip > archive-HEAD.tar.gz
```

这也可以通过（将使用内置的 tar.gz 处理）来完成：

```
git archive --format tar.gz HEAD > archive-HEAD.tar.gz
```

创建当前 HEAD 版本的 zip 压缩包：

```
git archive --format zip HEAD > archive-HEAD.zip
```

或者也可以只指定一个带有有效扩展名的输出文件，格式和压缩类型将从中推断：

```
git archive --output=archive-HEAD.tar.gz HEAD
```

第34.2节：创建带目录前缀的 git 仓库归档

创建 git 归档时使用前缀被认为是良好实践，这样解压时所有文件都会放在一个目录内

Chapter 34: Archive

Parameter	Details
--format=<fmt>	Format of the resulting archive: `tar` or `zip`. If this options is not given and the output file is specified, the format is inferred from the filename if possible. Otherwise, defaults to `tar`.
-l, --list	Show all available formats.
-v, --verbose	Report progress to stderr.
--prefix=<prefix>/	Prepend <prefix>/ to each filename in the archive.
-o <file>, --output=<file>	Write the archive to <file> instead of stdout.
--worktree-attributes	Look for attributes in `.gitattributes` files in the working tree.
<extra>	This can be any options that the archiver backend understands. For `zip` backend, using `-0` will store the files without deflating them, while `-1` through `-9` can be used to adjust compression speed and ratio.
--remote=<repo>	Retrieve a tar archive from a remote repository **<repo>** rather than the local repository.
--exec=<git-upload-archive>	Used with `--remote` to specify the path to the **<git-upload-archive** on the remote.
<tree-ish>	The tree or commit to produce an archive for.
<path>	Without an optional parameter, all files and directories in the current working directory are included in the archive. If one or more paths are specified, only these are included.

Section 34.1: Create an archive of git repository

With `git archive` it is possible to create compressed archives of a repository, for example for distributing releases.

Create a tar archive of current `HEAD` revision:

```
git archive --format tar HEAD | cat > archive-HEAD.tar
```

Create a tar archive of current `HEAD` revision with gzip compression:

```
git archive --format tar HEAD | gzip > archive-HEAD.tar.gz
```

This can also be done with (which will use the in-built tar.gz handling):

```
git archive --format tar.gz HEAD > archive-HEAD.tar.gz
```

Create a zip archive of current `HEAD` revision:

```
git archive --format zip HEAD > archive-HEAD.zip
```

Alternatively it is possible to just specify an output file with valid extension and the format and compression type will be inferred from it:

```
git archive --output=archive-HEAD.tar.gz HEAD
```

Section 34.2: Create an archive of git repository with directory prefix

It is considered good practice to use a prefix when creating git archives, so that extraction will place all files inside a

目录。要创建带有目录前缀的HEAD归档：

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

解压后，所有文件将被解压到当前目录下名为 src-directory-name 的文件夹中。

第34.3节：基于特定分支、修订、标签或目录创建git仓库归档

也可以创建除 HEAD 之外的其他项目的归档，如分支、提交、标签和目录。

创建本地分支 dev 的归档：

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

创建远程分支 origin/dev 的归档：

```
git archive --output=archive-dev.zip --prefix=src-directory-name origin/dev
```

创建标签 v.01 的归档：

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

在修订版本HEAD的特定子目录（sub-dir）内创建文件归档：

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```


directory. To create an archive of `HEAD` with a directory prefix:

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

When extracted all the files will be extracted inside a directory named `src-directory-name` in the current directory.

Section 34.3: Create archive of git repository based on specific branch, revision, tag or directory

It is also possible to create archives of other items than `HEAD`, such as branches, commits, tags, and directories.

To create an archive of a local branch `dev`:

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

To create an archive of a remote branch `origin/dev`:

```
git archive --output=archive-dev.zip --prefix=src-directory-name origin/dev
```

To create an archive of a tag `v.01`:

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

Create an archive of files inside a specific sub directory (`sub-dir`) of revision `HEAD`:

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

第35章：使用filter-branch重写历史

第35.1节：更改提交的作者

您可以使用环境过滤器来更改提交的作者。只需修改并导出脚本中的$GIT_AUTHOR_NAME，即可更改提交的作者。

创建一个名为filter.sh的文件，内容如下：

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
    export GIT_AUTHOR_NAME="Author to Change To"
    export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

然后从命令行运行filter-branch：

```
chmod +x ./filter.sh
git filter-branch --env-filter ./filter.sh
```

第35.2节：将git提交者设置为与提交作者相同

该命令，给定一个提交范围 commit1..commit2，重写历史，使git提交作者也成为git提交者：

```
git filter-branch -f --commit-filter \
   'export GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
    export GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
    export GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
    git commit-tree $@' \
    -- commit1..commit2
```


Chapter 35: Rewriting history with filter-branch

Section 35.1: Changing the author of commits

You can use an environment filter to change the author of commits. Just modify and export `$GIT_AUTHOR_NAME` in the script to change who authored the commit.

Create a file `filter.sh` with contents like so:

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
    export GIT_AUTHOR_NAME="Author to Change To"
    export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Then run `filter-branch` from the command line:

```
chmod +x ./filter.sh
git filter-branch --env-filter ./filter.sh
```

Section 35.2: Setting git committer equal to commit author

This command, given a commit range `commit1..commit2`, rewrites history so that git commit author becomes also git committer:

```
git filter-branch -f --commit-filter \
   'export GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
    export GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
    export GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
    git commit-tree $@' \
    -- commit1..commit2
```

第36章：迁移到Git

第36.1节：SubGit

SubGit 可用于对SVN仓库进行一次性导入到git。

```
$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git
```

第36.2节：使用Atlassian转换工具从SVN迁移到Git

下载Atlassian转换工具 here。该工具需要Java，因此请确保您在计划进行转换的机器上安装了Java运行环境 JRE。

使用命令 `java -jar svn-migration-scripts.jar verify` 检查您的机器是否缺少完成转换所需的任何程序。具体来说，该命令会检查Git、Subversion和 `git-svn` 工具。它还会验证您是否在区分大小写的文件系统上执行迁移。迁移到Git 应该在区分大小写的文件系统上进行，以避免损坏仓库。

接下来，您需要生成一个作者文件。Subversion仅通过提交者的用户名跟踪更改。而Git则使用两个信息来区分用户：真实姓名和电子邮件地址。以下命令将生成一个文本文件，将Subversion用户名映射到其Git对应名称：

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

其中 **<svn-repo>** 是您希望转换的Subversion仓库的URL。运行此命令后，贡献者的身份信息将映射到 `authors.txt` 中。电子邮件地址的格式将是 **<username>**`@mycompany.com`。在作者文件中，您需要手动将每个人的默认名称（默认情况下为其用户名）更改为其真实姓名。请确保在继续之前检查所有电子邮件地址的正确性。

以下命令将把一个svn仓库克隆为Git仓库：

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

其中 <svn-repo> 是上面使用的相同仓库URL，<git-repo-name> 是当前目录中用于克隆仓库的文件夹名称。使用此命令前有几点需要注意：

- 上面的 `--stdlayout` 标志告诉Git您使用的是包含 `trunk`、`branches` 和 `tags` 文件夹的标准布局。非标准布局的Subversion仓库需要您指定 `trunk` 文件夹、所有 `branch` 文件夹和 `tags` 文件夹的位置。可以通过以下示例完成： `git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors-file=authors.txt <svn-repo> <git-repo-name>`。
- 此命令可能需要数小时才能完成，具体取决于您的仓库大小。
- 为了缩短大型代码库的转换时间，可以直接在托管Subversion代码库的服务器上运行转换，以消除网络开销。

`git svn clone` 会将Subversion的分支（以及主干）作为远程分支导入，包括Subversion标签（远程分支前缀为 tags/）。要将这些转换为实际的分支和标签，请在Linux机器上按顺序运行以下命令。运行后， `git branch -a` 应显示正确的分支名称， `git tag -l` 应显示代码库标签。

Chapter 36: Migrating to Git

Section 36.1: SubGit

SubGit may be used to perform a one-time import of an SVN repository to git.

```
$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git
```

Section 36.2: Migrate from SVN to Git using Atlassian conversion utility

Download the Atlassian conversion utility here. This utility requires Java, so please ensure that you have the Java Runtime Environment JRE installed on the machine you plan to do the conversion.

Use the command `java -jar svn-migration-scripts.jar verify` to check if your machine is missing any of the programs necessary to complete the conversion. Specifically, this command checks for the Git, subversion, and `git-svn` utilities. It also verifies that you are performing the migration on a case-sensitive file system. Migration to Git should be done on a case-sensitive file system to avoid corrupting the repository.

Next, you need to generate an authors file. Subversion tracks changes by the committer's username only. Git, however, uses two pieces of information to distinguish a user: a real name and an email address. The following command will generate a text file mapping the subversion usernames to their Git equivalents:

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

where **<svn-repo>** is the URL of the subversion repository you wish to convert. After running this command, the contributors' identification information will be mapped in `authors.txt`. The email addresses will be of the form **<username>**`@mycompany.com`. In the authors file, you will need to manually change each person's default name (which by default has become their username) to their actual names. Make sure to also check all of the email addresses for correctness before proceeding.

The following command will clone an svn repo as a Git one:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

where **<svn-repo>** is the same repository URL used above and **<git-repo-name>** is the folder name in the current directory to clone the repository into. There are a few considerations before using this command:

- The `--stdlayout` flag from above tells Git that you're using a standard layout with `trunk`, `branches`, and `tags` folders. Subversion repositories with non-standard layouts require you to specify the locations of the `trunk` folder, any/all `branch` folders, and the `tags` folder. This can be done by following this example: `git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors-file=authors.txt <svn-repo> <git-repo-name>`.
- This command could take many hours to complete depending on the size of your repo.
- To cut down the conversion time for large repositories, the conversion can be run directly on the server hosting the subversion repository in order to eliminate network overhead.

`git svn clone` imports the subversion branches (and trunk) as remote branches including subversion tags (remote branches prefixed with `tags/`). To convert these to actual branches and tags, run the following commands on a Linux machine in the order they are provided. After running them, `git branch -a` should show the correct branch names, and `git tag -l` should show the repository tags.

```
git for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname; do git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git branch "$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname"; done
```

从svn到Git的转换现在完成了！只需将本地仓库push到服务器，即可继续使用Git进行贡献，同时完整保留了svn的版本历史。

第36.3节：从Mercurial迁移到Git

可以使用以下方法将Mercurial代码库导入到Git中：

1. 使用fast_export：

```
cd
git clone git://repo.or.cz/fast-export.git
git init git_repo
cd git_repo
~/fast-export/hg-fast-export.sh -r /path/to/old/mercurial_repo
git checkout HEAD
```

2. 使用Hg-Git：这里有一个非常详细的回答：https://stackoverflow.com/a/31827990/5283213
3. 使用GitHub的导入工具：请按照GitHub上的（详细）说明操作。

第36.4节：从团队基础版本控制（TFVC）迁移到Git

您可以使用一个名为Git-TF的开源工具将团队基础版本控制迁移到Git。迁移还会通过将TFS提交转换为Git提交来传输您现有的历史记录。

要使用Git-TF将您的解决方案放入Git，请按照以下步骤操作：

下载Git-TF

您可以从Codeplex下载（并安装）Git-TF：Git-TF @ Codeplex

克隆您的TFVC解决方案

启动 PowerShell（Windows）并输入命令

```
git-tf clone http://my.tfs.server.address:port/tfs/mycollection '$/myproject/mybranch/mysolution' --deep
```

--deep 参数是关键字，因为它告诉 Git-Tf 复制你的提交历史。现在你在调用 clone 命令的文件夹中拥有了一个本地 git 仓库。

清理

- 添加一个 .gitignore 文件。如果你使用的是 Visual Studio，编辑器可以帮你完成，否则你可以通过从 github/gitignore 下载完整文件来手动完成。
- 从解决方案中移除 TFS 源代码控制绑定（删除所有 *.vssscc 文件）。你也可以通过修改解决方案文件，删除 GlobalSection(TeamFoundationVersionControl).....EndGlobalSection 来实现。

提交并推送


```
git for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname; do git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git branch "$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname"; done
```

The conversion from svn to Git is now complete! Simply push your local repo to a server and you can continue to contribute using Git as well as having a completely preserved version history from svn.

Section 36.3: Migrating Mercurial to Git

One can use the following methods in order to import a `Mercurial` Repo into `Git`:

1. Using fast export:

```
cd
git clone git://repo.or.cz/fast-export.git
git init git_repo
cd git_repo
~/fast-export/hg-fast-export.sh -r /path/to/old/mercurial_repo
git checkout HEAD
```

2. Using Hg-Git: A very detailed answer here: https://stackoverflow.com/a/31827990/5283213
3. Using GitHub's Importer: Follow the (detailed) instructions at GitHub.

Section 36.4: Migrate from Team Foundation Version Control (TFVC) to Git

You could migrate from team foundation version control to git by using an open source tool called Git-TF. Migration will also transfer your existing history by converting tfs checkins to git commits.

To put your solution into Git by using Git-TF follow these steps:

Download Git-TF

You can download (and install) Git-TF from Codeplex: Git-TF @ Codeplex

Clone your TFVC solution

Launch powershell (win) and type the command

```
git-tf clone http://my.tfs.server.address:port/tfs/mycollection '$/myproject/mybranch/mysolution' --deep
```

The --deep switch is the keeyword to note as this tells Git-Tf to copy your checkin-history. You now have a local git repository in the folder from which you called your cloe command from.

Cleanup

- Add a .gitignore file. If you are using Visual Studio the editor can do this for you, otherwise you could do this manually by downloading a complete file from github/gitignore.
- RemoveTFS source control bindings from solution (remove all *.vssscc files). You could also modify your solution file by removing the GlobalSection(TeamFoundationVersionControl).....EndClobalSection

Commit & Push

通过提交并推送你的本地仓库到远程仓库，完成转换。

```
git add .
git commit -a -m "Coverted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master
```

第36.5节：使用 svn2git 从 SVN 迁移到 Git

svn2git 是一个基于 Ruby 的封装工具，利用 git 原生的 SVN 支持工具 git-svn，帮助你将项目从 Subversion 迁移到 Git，同时保留历史记录（包括主干、标签和分支的历史）。

示例

要迁移具有标准布局的svn仓库（即分支、标签和主干位于仓库根目录）：

```
$ svn2git http://svn.example.com/path/to/repo
```

要迁移非标准布局的svn仓库：

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches
branches-dir
```

如果您不想迁移（或没有）分支、标签或主干，可以使用选项 --notrunk、--nobranches 和 --notags。

例如，`$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches` 将只迁移主干历史。

为了减少新仓库所需的空间，您可能想排除曾经添加但不应该存在的目录或文件（例如构建目录或归档文件）：

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

迁移后优化

如果您在新创建的git仓库中已有几千次（或更多）提交，您可能想在将仓库推送到远程之前减少所用空间。可以使用以下命令完成此操作：

```
$ git gc --aggressive
```

注意： 在大型仓库（数万个提交

和/或数百兆字节的历史）上，前面的命令可能需要几个小时才能完成。

Complete your conversion by committing and pushing your local repository to your remote.

```
git add .
git commit -a -m "Coverted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master
```

Section 36.5: Migrate from SVN to Git using svn2git

svn2git is a Ruby wrapper around git's native SVN support through git-svn, helping you with migrating projects from Subversion to Git, keeping history (incl. trunk, tags and branches history).

Examples

To migrate a svn repository with the standard layout (ie. branches, tags and trunk at the root level of the repository):

```
$ svn2git http://svn.example.com/path/to/repo
```

To migrate a svn repository which is not in standard layout:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches
branches-dir
```

In case you do not want to migrate (or do not have) branches, tags or trunk you can use options `--notrunk`, `--nobranches`, and `--notags`.

For example, `$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches` will migrate only trunk history.

To reduce the space required by your new repository you may want to exclude any directories or files you once added while you should not have (eg. build directory or archives):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

Post-migration optimization

If you already have a few thousand of commits (or more) in your newly created git repository, you may want to reduce space used before pushing your repository on a remote. This can be done using the following command:

```
$ git gc --aggressive
```

Note: The previous command can take up to several hours on large repositories (tens of thousand of commits and/or hundreds of megabytes of history).

第37章：显示

第37.1节：概述

git show 显示各种Git对象。

对于提交：

显示提交信息和引入更改的差异。

命令	描述
`git show`	显示上一个提交
`git show @~3`	显示倒数第3个提交

对于树和blob：

显示树或blob。

命令	描述
`git show @~3:`	显示项目根目录在3次提交之前的状态（一个树对象）
git show @~3:src/program.js	显示 src/program.js 在3次提交之前的状态（一个blob对象）
git show @:a.txt @:b.txt	显示当前提交中 `a.txt` 与 `b.txt` 的拼接内容

对于标签：

显示标签信息和引用的对象。

Chapter 37: Show

Section 37.1: Overview

`git show` shows various Git objects.

For commits:

Shows the commit message and a diff of the changes introduced.

Command	Description
`git show`	shows the previous commit
`git show @~3`	shows the 3rd-from-last commit

For trees and blobs:

Shows the tree or blob.

Command	Description
`git show @~3:`	shows the project root directory as it was 3 commits ago (a tree)
`git show @~3:src/program.js`	shows `src/program.js` as it was 3 commits ago (a blob)
`git show @:a.txt @:b.txt`	shows `a.txt` concatenated with `b.txt` from current commit

For tags:

Shows the tag message and the referenced object.

第38章：解决合并冲突

第38.1节：手动解决

在执行 `git merge` 时，你可能会发现git报告“合并冲突”错误。它会告诉你哪些文件存在冲突，你需要解决这些冲突。

在任何时候执行 git status 都能帮助你查看哪些内容仍需编辑，并给出类似的提示信息

```
在master分支
你有未合并的路径。
  (解决冲突并运行"git commit")

未合并的路径：
  (使用"git add <file>..."来标记解决)

双方均修改：       index.html

未添加更改以提交(使用"git add"和/或"git commit -a")
```

Git 会在文件中留下标记，告诉你冲突发生的位置：

```
<<<<<<<<< HEAD: index.html #表示你当前分支的状态
<div id="footer">联系：email@somedomain.com</div>
========= #表示冲突之间的分隔
<div id="footer">
请通过 email@somedomain.com 联系我们
</div>
>>>>>>>>> iss2: index.html #表示另一个分支（iss2）的状态
```

为了解决冲突，您必须适当编辑 <<<<<< 和 >>>>>>> 标记之间的区域，完全删除状态行（即 <<<<<<<、>>>>>> 和 ======== 行）。然后使用git add index.html 标记为已解决，接着使用git commit 完成合并。

Chapter 38: Resolving merge conflicts

Section 38.1: Manual Resolution

While performing a `git merge` you may find that git reports a "merge conflict" error. It will report to you which files have conflicts, and you will need to resolve the conflicts.

A `git status` at any point will help you see what still needs editing with a helpful message like

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

    both modified:      index.html

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

Git leaves markers in the files to tell you where the conflict arose:

```
<<<<<<<<< HEAD: index.html #indicates the state of your current branch
<div id="footer">contact : email@somedomain.com</div>
========= #indicates break between conflicts
<div id="footer">
please contact us at email@somedomain.com
</div>
>>>>>>>>> iss2: index.html #indicates the state of the other branch (iss2)
```

In order to resolve the conflicts, you must edit the area between the <<<<<< and >>>>>>> markers appropriately, remove the status lines (the <<<<<<<, >>>>>>>, and ======== lines) completely. Then `git add index.html` to mark it resolved and `git commit` to finish the merge.

第39章：捆绑包

第39.1节：在本地机器上创建git捆绑包并在另一台机器上使用

有时您可能希望在没有网络连接的机器上维护git仓库的版本。
捆绑包允许您将一个机器上仓库中的git对象和引用打包，并导入到另一台机器上的仓库中。

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

以某种方式将**changes_between_tags.bundle**文件传输到远程机器；例如，通过U盘。一旦您将其放到那里：

```
git bundle verify changes_between_tags.bundle  # 确保捆绑包完整无损
git checkout [some branch]       # 在远程机器的仓库中
git bundle list-heads changes_between_tags.bundle # 列出捆绑包中的引用
git pull changes_between_tags.bundle [来自捆绑包的引用，例如last字段，
来自之前的输出]
```

反过来也可以。一旦你在远程仓库进行了更改，就可以将增量打包；例如，将更改放到U盘中，然后合并回本地仓库，这样两者就能保持同步，而无需机器之间直接使用git、ssh、rsync或http协议访问。

Chapter 39: Bundles

Section 39.1: Creating a git bundle on the local machine and using it on another

Sometimes you may want maintain versions of a git repository on machines that have no network connection. Bundles allow you to package git objects and references in a repository on one machine and import those into a repository on another.

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

Somehow transfer the **changes_between_tags.bundle** file to the remote machine; e.g., via thumb drive. Once you have it there:

```
git bundle verify changes_between_tags.bundle  # make sure bundle arrived intact
git checkout [some branch]       # in the repo on the remote machine
git bundle list-heads changes_between_tags.bundle # list the references in the bundle
git pull changes_between_tags.bundle [reference from the bundle, e.g. last field from the previous output]
```

The reverse is also possible. Once you've made changes on the remote repository you can bundle up the deltas; put the changes on, e.g., a thumb drive, and merge them back into the local repository so the two can stay in sync without requiring direct `git`, `ssh`, `rsync`, or `http` protocol access between the machines.

第40章：使用Gitk图形化显示提交历史

第40.1节：显示单个文件的提交历史

```
gitk path/to/myfile
```

第40.2节：显示两个提交之间的所有提交

假设你有两个提交 d9e1db9 和 5651067，想查看它们之间发生了什么。 d9e1db9 是最早的祖先， 5651067 是提交链中的最终后代。

```
gitk --ancestry-path d9e1db9 5651067
```

第40.3节：显示自版本标签以来的提交

如果你有版本标签 v2.3，可以显示自该标签以来的所有提交。

```
gitk v2.3..
```


Chapter 40: Display commit history graphically with Gitk

Section 40.1: Display commit history for one file

```
gitk path/to/myfile
```

Section 40.2: Display all commits between two commits

Let's say you have two commits `d9e1db9` and `5651067` and want to see what happened between them. `d9e1db9` is the oldest ancestor and `5651067` is the final descendant in the chain of commits.

```
gitk --ancestry-path d9e1db9 5651067
```

Section 40.3: Display commits since version tag

If you have the version tag `v2.3` you can display all commits since that tag.

```
gitk v2.3..
```

第41章：二分查找/定位错误提交

第41.1节：二分查找（git bisect）

git bisect 允许你使用二分查找来找到引入错误的提交。

通过提供两个提交引用开始二分查找会话：一个是错误出现之前的良好提交，另一个是错误出现之后的错误提交。
通常，错误提交是HEAD。

```
# 开始 git bisect 会话
$ git bisect start

# 提供一个不存在错误的提交
$ git bisect good 49c747d

# 提供一个存在错误的提交
$ git bisect bad HEAD
```

git 开始二分查找：它将修订版本分成两半，并将仓库切换到中间的修订版本。
检查代码以确定该修订版本是良好还是错误：

```
# 告诉 git 该修订是好的，
# 意味着它不包含该错误
$ git bisect good

# 如果该版本包含错误，
# 则告诉 git 它是坏的
$ git bisect bad
```

git 将根据你的指令继续对剩余的坏版本子集进行二分查找。git 会呈现一个单一的版本，除非你的标记有误，否则该版本正是引入错误的版本。

之后记得运行 git bisect reset 来结束 bisect 会话并返回到 HEAD。

```
$ git bisect reset
```

如果你有一个可以检测错误的脚本，可以用以下命令自动化该过程：

```
$ git bisect run [script] [arguments]
```

其中 [script] 是你的脚本路径，[arguments] 是传递给脚本的参数。

运行此命令将自动执行二分查找，根据脚本的退出码在每一步执行 `git bisect good` 或 `git bisect bad`。退出码为 0 表示 good，退出码为 1-124、126 或 127 表示坏。退出码为 125 表示脚本无法测试该版本（这将触发 `git bisect skip`）。

第41.2节：半自动查找错误提交

假设你当前在master分支上，某些功能没有按预期工作（引入了回归），但你不知道具体在哪个提交。你只知道它在上一个版本中是正常的（例如，该版本有标签，或者你知道提交哈希，这里假设为old-rel）。

Chapter 41: Bisecting/Finding faulty commits

Section 41.1: Binary search (git bisect)

`git bisect` allows you to find which commit introduced a bug using a binary search.

Start by bisecting a session by providing two commit references: a good commit before the bug, and a bad commit after the bug. Generally, the bad commit is `HEAD`.

```
# start the git bisect session
$ git bisect start

# give a commit where the bug doesn't exist
$ git bisect good 49c747d

# give a commit where the bug exist
$ git bisect bad HEAD
```

`git` starts a binary search: It splits the revision in half and switches the repository to the intermediate revision. Inspect the code to determine if the revision is good or bad:

```
# tell git the revision is good,
# which means it doesn't contain the bug
$ git bisect good

# if the revision contains the bug,
# then tell git it's bad
$ git bisect bad
```

`git` will continue to run the binary search on each remaining subset of bad revisions depending on your instructions. `git` will present a single revision that, unless your flags were incorrect, will represent exactly the revision where the bug was introduced.

Afterwards remember to run `git bisect reset` to end the bisect session and return to HEAD.

```
$ git bisect reset
```

If you have a script that can check for the bug, you can automate the process with:

```
$ git bisect run [script] [arguments]
```

Where `[script]` is the path to your script and `[arguments]` is any arguments that should be passed to your script.

Running this command will automatically run through the binary search, executing `git bisect good` or `git bisect bad` at each step depending on the exit code of your script. Exiting with 0 indicates good, while exiting with 1-124, 126, or 127 indicates bad. 125 indicates that the script cannot test that revision (which will trigger a `git bisect skip`).

Section 41.2: Semi-automatically find a faulty commit

Imagine you are on the `master` branch and something is not working as expected (a regression was introduced), but you don't know where. All you know is, that is was working in the last release (which was e.g., tagged or you know the commit hash, lets take `old-rel` here).

Git可以帮助你通过非常少的步骤（二分查找）找到引入回归的错误提交。

首先开始二分查找：

```
git bisect start master old-rel
```

这会告诉Git，master是一个坏的修订版本（或第一个坏的版本），而old-rel是最后一个已知的正常版本。

Git现在会检出这两个提交中间的一个分离头（detached head）。现在，你可以进行测试。根据测试结果是否正常，执行

```
git bisect good
```

或

```
git bisect bad
```

如果此提交无法测试，您可以轻松地git reset并测试那个，git会处理这个。

经过几步操作，git 将输出有问题的提交哈希值。

要中止二分查找过程，只需执行

```
git bisect reset
```

git 将恢复到之前的状态。

Git has help for you, finding the faulty commit which introduced the regression with a very low number of steps (binary search).

First of all start bisecting:

```
git bisect start master old-rel
```

This will tell git that `master` is a broken revision (or the first broken version) and `old-rel` is the last known version.

Git will now check out a detached head in the middle of both commits. Now, you can do your testing. Depending on whether it works or not issue

```
git bisect good
```

or

```
git bisect bad
```

. In case this commit cannot be tested, you can easily `git reset` and test that one, git willl take care of this.

After a few steps git will output the faulty commit hash.

In order to abort the bisect process just issue

```
git bisect reset
```

and git will restore the previous state.

第42章：追踪责任

参数	详情
文件名	需要检查详情的文件名
-f	显示原始提交中的文件名
-e	显示作者邮箱而非作者姓名
-w	忽略空白字符，在比较子版本和父版本时不考虑空白
-L start,end	仅显示指定的行范围 例如：git blame -L 1,2 [filename]
--show-stats	在 blame 输出末尾显示额外统计信息
-l	显示完整的修订号（默认：关闭）
-t	显示原始时间戳（默认：关闭）
-reverse	向前浏览历史而非向后
-p, --porcelain	供机器处理的输出
-M	检测文件内移动或复制的行
-C	除了 -M 外，还检测在同一次提交中从其他文件移动或复制且被修改的行
-h	显示帮助信息
-c	使用与 git-annotate 相同的输出模式（默认：关闭）
-n	显示原始提交中的行号（默认：关闭）

第42.1节：仅显示特定行

可以通过指定行范围来限制输出，如

```
git blame -L <start>,<end>
```

其中 <start> 和 <end> 可以是：

- 行号

  ```
  git blame -L 10,30
  ```

- /正则表达式/

  git blame -L /void main/, git blame -L 46,/void foo/

- +offset，-offset（仅适用于 <end>）

  git blame -L 108,+30, git blame -L 215,-15

可以指定多个行范围，且允许范围重叠。

git blame -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40

第42.2节：查找是谁修改了文件

```
// 显示指定文件每行的作者和提交信息
git blame test.c
```


Chapter 42: Blaming

Parameter	Details
filename	Name of the file for which details need to be checked
-f	Show the file name in the origin commit
-e	Show the author email instead of author name
-w	Ignore white spaces while making a comparison between child and parent's version
-L start,end	Show only the given line range Example: `git blame -L 1,2 [filename]`
--show-stats	Shows additional statistics at end of blame output
-l	Show long rev (Default: off)
-t	Show raw timestamp (Default: off)
-reverse	Walk history forward instead of backward
-p, --porcelain	Output for machine consumption
-M	Detect moved or copied lines within a file
-C	In addition to -M, detect lines moved or copied from other files that were modified in the same commit
-h	Show the help message
-c	Use the same output mode as git-annotate (Default: off)
-n	Show the line number in the original commit (Default: off)

Section 42.1: Only show certain lines

Output can be restricted by specifying line ranges as

```
git blame -L <start>,<end>
```

Where **<start>** and **<end>** can be:

- line number

  ```
  git blame -L 10,30
  ```

- /regex/

  ```
  git blame -L /void main/, git blame -L 46,/void foo/
  ```

- +offset, -offset (only for **<end>**)

  ```
  git blame -L 108,+30, git blame -L 215,-15
  ```

Multiple line ranges can be specified, and overlapping ranges are allowed.

```
git blame -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40
```

Section 42.2: To find out who changed a file

```
// Shows the author and commit per line of specified file
git blame test.c
```

```
// 显示指定文件每行的作者邮箱和提交信息
git blame -e test.c file

// 通过指定范围限制行的选择
git blame -L 1,10 test.c
```

第42.3节：显示最后修改某行的提交

`git blame <file>`

将显示文件，每行标注最后修改该行的提交。

第42.4节：忽略仅空白字符的更改

有时仓库中会有只调整空白字符的提交，例如修正缩进或在制表符和空格之间切换。这会使得查找代码实际编写的提交变得困难。

`git blame -w`

将忽略仅空白字符的更改，以查找该行的真实来源。


```
// Shows the author email and commit per line of specified
git blame -e test.c file

// Limits the selection of lines by specified range
git blame -L 1,10 test.c
```

Section 42.3: Show the commit that last modified a line

`git blame <file>`

will show the file with each line annotated with the commit that last modified it.

Section 42.4: Ignore whitespace-only changes

Sometimes repos will have commits that only adjust whitespace, for example fixing indentation or switching between tabs and spaces. This makes it difficult to find the commit where the code was actually written.

`git blame -w`

will ignore whitespace-only changes to find where the line really came from.

第43章：Git修订语法

第43.1节：通过对象名称指定修订版本

```
$ git show dae86e1950b1277e545cee180551750029cfe735
$ git show dae86e19
```

您可以使用SHA-1对象名称指定修订版本（实际上也可以是任何对象：标签、树即目录内容、blob即文件内容），可以是完整的40字节十六进制字符串，或在仓库中唯一的子字符串。

第43.2节：符号引用名称：分支、标签、远程跟踪分支

```
$ git log master    # 指定分支
$ git show v1.0     # 指定标签
$ git show HEAD     # 指定当前分支
$ git show origin   # 指定远程 'origin' 的默认远程跟踪分支
```

您可以使用符号引用名称指定修订版本，其中包括分支（例如“master”、“next”、“maint”）、标签（例如“v1.0”、“v0.6.3-rc2”）、远程跟踪分支（例如“origin”、“origin/master”）以及特殊引用，如表示当前分支的“HEAD”。

如果符号引用名称有歧义，例如你同时有名为“fix”的分支和标签（不建议分支和标签同名），你需要指定想要使用的引用类型：

```
$ git show heads/fix      # 或者 'refs/heads/fix'，用于指定分支
$ git show tags/fix       # 或者 'refs/tags/fix'，用于指定标签
```

第43.3节：默认修订版：HEAD

```
$ git show        # 等同于 'git show HEAD'
```

“HEAD”表示你基于其进行工作树更改的提交，通常是当前分支的符号名称。许多（但不是全部）接受修订参数的命令，如果缺省则默认使用“HEAD”。

第43.4节：Reflog引用：<refname>@{<n>}

```
$ git show @{1}           # 使用当前分支的reflog
$ git show master@{1}     # 使用分支 'master' 的reflog
$ git show HEAD@{1}       # 使用 'HEAD' 的reflog
```

一个引用，通常是分支或HEAD，后跟后缀@和用大括号括起来的序号（例如{1}, {15}）指定该引用在本地仓库中的第n个之前的值。你可以使用git reflog命令查看最近的reflog条目，或者使用--walk-reflogs / -g选项配合git log命令。

```
$ git reflog
08bb350 HEAD@{0}: reset: 移动到 HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): 文档查询参数
08bb350 HEAD@{2}: pull: 快进
f34be46 HEAD@{3}：检出：从 af40944bda352190f05d22b7cb8fe88beb17f3a7 移动到 master 分支
af40944 HEAD@{4}：检出：从 master 移动到 v2.6.3

$ git reflog gitweb-docs
```


Chapter 43: Git revisions syntax

Section 43.1: Specifying revision by object name

```
$ git show dae86e1950b1277e545cee180551750029cfe735
$ git show dae86e19
```

You can specify revision (or in truth any object: tag, tree i.e. directory contents, blob i.e. file contents) using SHA-1 object name, either full 40-byte hexadecimal string, or a substring that is unique to the repository.

Section 43.2: Symbolic ref names: branches, tags, remote-tracking branches

```
$ git log master    # specify branch
$ git show v1.0     # specify tag
$ git show HEAD     # specify current branch
$ git show origin   # specify default remote-tracking branch for remote 'origin'
```

You can specify revision using a symbolic ref name, which includes branches (for example 'master', 'next', 'maint'), tags (for example 'v1.0', 'v0.6.3-rc2'), remote-tracking branches (for example 'origin', 'origin/master'), and special refs such as 'HEAD' for current branch.

If the symbolic ref name is ambiguous, for example if you have both branch and tag named 'fix' (having branch and tag with the same name is not recommended), you need to specify the kind of ref you want to use:

```
$ git show heads/fix      # or 'refs/heads/fix', to specify branch
$ git show tags/fix       # or 'refs/tags/fix', to specify tag
```

Section 43.3: The default revision: HEAD

```
$ git show        # equivalent to 'git show HEAD'
```

'HEAD' names the commit on which you based the changes in the working tree, and is usually the symbolic name for the current branch. Many (but not all) commands that take revision parameter defaults to 'HEAD' if it is missing.

Section 43.4: Reflog references: <refname>@{<n>}

```
$ git show @{1}           # uses reflog for current branch
$ git show master@{1}     # uses reflog for branch 'master'
$ git show HEAD@{1}       # uses 'HEAD' reflog
```

A ref, usually a branch or HEAD, followed by the suffix `@` with an ordinal specification enclosed in a brace pair (e.g. `{1}`, `{15}`) specifies the n-th prior value of that ref *in your **local** repository*. You can check recent reflog entries with `git reflog` command, or `--walk-reflogs` / `-g` option to `git log`.

```
$ git reflog
08bb350 HEAD@{0}: reset: moving to HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{2}: pull: Fast-forward
f34be46 HEAD@{3}: checkout: moving from af40944bda352190f05d22b7cb8fe88beb17f3a7 to master
af40944 HEAD@{4}: checkout: moving from master to v2.6.3

$ git reflog gitweb-docs
```

```
4ebf58d gitweb-docs@{0}：分支：从 master 创建
```

注意：使用 reflogs 实际上取代了旧的利用 ORIG_HEAD 引用（大致相当于 HEAD@{1}）的方法。

第 43.5 节：Reflog 引用：<refname>@{<date>}

```
$ git show master@{昨天}
$ git show HEAD@{5 分钟前}    # 或 HEAD@{5.minutes.ago}
```

一个引用后跟后缀 @ 以及用大括号括起来的日期说明（例如 {昨天}，{1 个月 2 周 3 天 1 小时 1 秒前} 或 {1979-02-26 18:30:00}）指定该引用在之前某个时间点（或最接近该时间点）的值。注意，这会查找你本地引用在给定时间的状态；例如，你本地的内容是什么上周的'master'分支。

你可以使用git reflog并带上日期说明符来查找在本地仓库中对指定引用所做操作的确切时间。

```
$ git reflog HEAD@{now}
08bb350 HEAD@{2016年7月23日 星期六 19:48:13 +0200}: reset: 移动到 HEAD^
4ebf58d HEAD@{2016年7月23日 星期六 19:39:20 +0200}: commit: gitweb(1): 文档查询参数
08bb350 HEAD@{2016年7月23日 星期六 19:26:43 +0200}: pull: 快进合并
```

第43.6节：跟踪/上游分支：<branchname>@{upstream}

```
$ git log @{upstream}..       # 本地已做但尚未发布的操作，当前分支
$ git show master@{upstream}  # 显示分支 'master' 的上游
```

附加在分支名后的后缀@{upstream}（简写为<branchname>@{u}）指的是该分支名指定的分支所基于的上游分支（通过branch.<name>.remote和branch.<name>.merge配置，或通过git branch --set-upstream-to=<branch>设置）。如果缺少分支名，则默认为当前分支。

结合修订范围的语法，非常有用来查看你的分支领先上游的提交（本地仓库中尚未推送到上游的提交），以及落后于上游的提交（上游中尚未合并到本地分支的提交），或者两者兼有：

```
$ git log --oneline @{u}..
$ git log --oneline ..@{u}
$ git log --oneline --left-right @{u}...  # 与 ...@{u} 相同
```

第43.7节：提交祖先链：<rev>^，<rev>~<n>，等

```
$ git reset --hard HEAD^             # 丢弃最后一次提交
$ git rebase --interactive HEAD~5    # 交互式变基最近4次提交
```

修订参数后缀 ^ 表示该提交对象的第一个父提交。 ^<n> 表示第<n>个父提交（即 <rev>^ 等同于 <rev>^1）。

修订参数后缀 ~<n> 表示该命名提交对象的第<n>代祖先提交对象，仅沿第一个父提交路径。 例如，<rev>~3 等同于 <rev>^^^。作为快捷方式，<rev>~ 表示 <rev>~1，等同于 <rev>^1，简写为 <rev>^。


```
4ebf58d gitweb-docs@{0}: branch: Created from master
```

Note: using reflogs practically replaced older mechanism of utilizing `ORIG_HEAD` ref (roughly equivalent to `HEAD@{1}`).

Section 43.5: Reflog references: <refname>@{<date>}

```
$ git show master@{yesterday}
$ git show HEAD@{5 minutes ago}   # or HEAD@{5.minutes.ago}
```

A ref followed by the suffix `@` with a date specification enclosed in a brace pair (e.g. `{yesterday}`, `{1 month 2 weeks 3 days 1 hour 1 second ago}` or `{1979-02-26 18:30:00}`) specifies the value of the ref at a prior point in time (or closest point to it). Note that this looks up the state of your **local** ref at a given time; e.g., what was in your local *'master'* branch last week.

You can use `git reflog` with a date specifier to look up exact time where you did something to given ref in the local repository.

```
$ git reflog HEAD@{now}
08bb350 HEAD@{Sat Jul 23 19:48:13 2016 +0200}: reset: moving to HEAD^
4ebf58d HEAD@{Sat Jul 23 19:39:20 2016 +0200}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{Sat Jul 23 19:26:43 2016 +0200}: pull: Fast-forward
```

Section 43.6: Tracked / upstream branch: <branchname>@{upstream}

```
$ git log @{upstream}..       # what was done locally and not yet published, current branch
$ git show master@{upstream}  # show upstream of branch 'master'
```

The suffix `@{upstream}` appended to a branchname (short form **`<branchname>`**`@{u}`) refers to the branch that the branch specified by branchname is set to build on top of (configured with `branch.<name>.remote` and `branch.<name>.merge`, or with `git branch --set-upstream-to=<branch>`). A missing branchname defaults to the current one.

Together with syntax for revision ranges it is very useful to see the commits your branch is ahead of upstream (commits in your local repository not yet present upstream), and what commits you are behind (commits in upstream not merged into local branch), or both:

```
$ git log --oneline @{u}..
$ git log --oneline ..@{u}
$ git log --oneline --left-right @{u}...  # same as ...@{u}
```

Section 43.7: Commit ancestry chain: <rev>^, <rev>~<n>, etc

```
$ git reset --hard HEAD^             # discard last commit
$ git rebase --interactive HEAD~5    # rebase last 4 commits
```

A suffix `^` to a revision parameter means the first parent of that commit object. `^<n>` means the <n>-th parent (i.e. **`<rev>`**`^` is equivalent to **`<rev>`**`^1`).

A suffix `~<n>` to a revision parameter means the commit object that is the <n>-th generation ancestor of the named commit object, following only the first parents. This means that for example **`<rev>`**`~3` is equivalent to **`<rev>`**`^^^`. As a shortcut, **`<rev>`**`~` means **`<rev>`**`~1`, and is equivalent to **`<rev>`**`^1`, or **`<rev>`**`^` in short.

此语法是可组合的。

要查找此类符号名称，可以使用 git name-rev 命令：

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

请注意，以下示例中必须使用--pretty=oneline，而不是--oneline

```
$ git log --pretty=oneline | git name-rev --stdin --name-only
master 第六批主题 for 2.10
master~1 合并分支 'ls/p4-tmp-refs'
master~2 合并分支 'js/am-call-theirs-theirs-in-fallback-3way'
[...]
master~14^2 sideband.c：对 strbuf 使用的小优化
master~16^2 connect：从配置文件读取$GIT_SSH_COMMAND
[...]
master~22^2~1 t7810-grep.sh：修复空白不一致问题
master~22^2~2 t7810-grep.sh：修复重复的测试名称
```

第43.8节：解引用分支和标签：<rev>^0, <rev>^{<type>}

在某些情况下，命令的行为取决于它是接收分支名、标签名还是任意修订版本。如果需要后者，可以使用“解引用”语法。

后缀^后跟用大括号括起来的对象类型名（tag、commit、tree、blob）（例如v0.99.8^{commit}）表示递归解引用<rev>处的对象，直到找到类型为<type>的对象，或者无法再解引用。<rev>^0是<rev>^{commit}的简写。

```
$ git checkout HEAD^0    # 在现代 Git 中等同于 'git checkout --detach'
```

后缀^后跟空的大括号（例如v0.99.8^{}）表示递归解引用标签，直到找到非标签对象。

比较

```
$ git show v1.0
$ git cat-file -p v1.0
$ git replace --edit v1.0
```

with

```
$ git show v1.0^{}
$ git cat-file -p v1.0^{}
$ git replace --edit v1.0^{}
```

第43.9节：最年轻的匹配提交：<rev>^{/<text>}, :/<text>

```
$ git show HEAD^{/修复严重错误}    # 从HEAD开始查找
$ git show ':/修复严重错误'        # 从任意分支开始查找
```

冒号（':'），后跟斜杠（'/'），再后跟文本，表示一个提交，其提交信息匹配指定的正则表达式。该名称返回从任意引用可达的最年轻匹配提交。

This syntax is composable.

To find such symbolic names you can use the `git name-rev` command:

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

Note that `--pretty=oneline` and not `--oneline` must be used in the following example

```
$ git log --pretty=oneline | git name-rev --stdin --name-only
master Sixth batch of topics for 2.10
master~1 Merge branch 'ls/p4-tmp-refs'
master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'
[...]
master~14^2 sideband.c: small optimization of strbuf usage
master~16^2 connect: read $GIT_SSH_COMMAND from config file
[...]
master~22^2~1 t7810-grep.sh: fix a whitespace inconsistency
master~22^2~2 t7810-grep.sh: fix duplicated test name
```

Section 43.8: Dereferencing branches and tags: <rev>^0, <rev>^{<type>}

In some cases the behavior of a command depends on whether it is given branch name, tag name, or an arbitrary revision. You can use "de-referencing" syntax if you need the latter.

A suffix `^` followed by an object type name (`tag`, `commit`, `tree`, `blob`) enclosed in brace pair (for example `v0.99.8^{commit}`) means dereference the object at **`<rev>`** recursively until an object of type **`<type>`** is found or the object cannot be dereferenced anymore. **`<rev>^0`** is a short-hand for **`<rev>`**`^{commit}`.

```
$ git checkout HEAD^0    # equivalent to 'git checkout --detach' in modern Git
```

A suffix `^` followed by an empty brace pair (for example `v0.99.8^{}`) means to dereference the tag recursively until a non-tag object is found.

Compare

```
$ git show v1.0
$ git cat-file -p v1.0
$ git replace --edit v1.0
```

with

```
$ git show v1.0^{}
$ git cat-file -p v1.0^{}
$ git replace --edit v1.0^{}
```

Section 43.9: Youngest matching commit: <rev>^{/<text>}, :/<text>

```
$ git show HEAD^{/fix nasty bug}   # find starting from HEAD
$ git show ':/fix nasty bug'       # find starting from any branch
```

A colon (':'), followed by a slash ('/'), followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from *any* ref.

正则表达式可以匹配提交信息的任意部分。要匹配以某字符串开头的信息，可以使用例如:/^foo。特殊序列:/!保留用于匹配修饰符。 :/!-foo执行负匹配，而:/!!foo匹配字面上的!字符，后跟foo。

修订参数后缀^，后跟一对大括号，内含以斜杠开头的文本，与下面的:/<text>语法相同，返回^之前的<rev>可达的最年轻匹配提交。

The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. `:/^foo`. The special sequence `:/!` is reserved for modifiers to what is matched. `:/!-foo` performs a negative match, while `:/!!foo` matches a literal ! character, followed by `foo`.

A suffix `^` to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the `:/<text>` syntax below that it returns the youngest matching commit which is reachable from the **<rev>** before `^`.

第44章：工作树

参数	详情
-f --force	默认情况下，当<branch>已被另一个工作树签出时，add命令拒绝创建新的工作树。此选项可覆盖该保护措施。
-b <new-branch> -B <new-branch>	使用add命令，创建一个名为<new-branch>的新分支，起点为<branch>，并签出<new-branch>到新的工作树中。如果省略<branch>，则默认为HEAD。默认情况下，-b如果分支已存在则拒绝创建新分支。使用-B可覆盖此保护，将<new-branch>重置为<branch>。
--detach	使用add命令，在新的工作树中分离HEAD。
--[no-] checkout	默认情况下，add会签出<branch>，但可以使用--no-checkout来抑制签出，以便进行自定义操作，例如配置稀疏签出（sparse-checkout）。
-n --dry-run	使用 prune 时，不删除任何内容；仅报告将要删除的内容。
--porcelain	使用 list 时，以便于脚本解析的格式输出。该格式将在不同 Git 版本及用户配置下保持稳定。
-v --verbose	使用 prune 时，报告所有删除操作。
--expire <time>	使用 prune 时，仅过期未使用且早于<time>的工作树。

第44.1节：使用工作树

你正处于开发新功能的关键阶段，老板突然进来要求你立即修复某个问题。你通常会想使用git stash暂时保存你的更改。然而，此时你的工作树处于混乱状态（有新增、移动和删除的文件，以及其他零散的内容），你不想打扰你的进度。

通过添加工作树，您可以创建一个临时的关联工作树来进行紧急修复，完成后将其移除，
然后继续之前的编码工作：

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# … 工作 工作 工作 …
$ git commit -a -m '给老板的紧急修复'
$ popd
$ rm -rf ../temp
$ git worktree prune
```

注意：在此示例中，修复仍在 emergency-fix 分支。此时你可能想要执行 `git merge` 或 `git format-patch`，之后再删除 emergency-fix 分支。

第44.2节：移动工作树

目前（截至版本2.11.0）没有内置功能可以移动已存在的工作树。这被列为官方缺陷（见 https://git-scm.com/docs/git-worktree#_bugs）。

为绕过此限制，可以直接在 .git 引用文件中手动操作。

在此示例中，仓库的主副本位于 /home/user/project-main，次级工作树位于 /home/user/project-1，我们想将其移动到 /home/user/project-2。

在这些步骤之间不要执行任何 git 命令，否则可能会触发垃圾回收，导致对次级工作树的引用丢失。请从开始到结束连续执行以下步骤，不要中断：

Chapter 44: Worktrees

Parameter	Details
-f --force	By default, add refuses to create a new working tree when **<branch>** is already checked out by another working tree. This option overrides that safeguard.
-b **<new-branch>** -B **<new-branch>**	With add, create a new branch named **<new-branch>** starting at **<branch>**, and check out **<new-branch>** into the new working tree. If **<branch>** is omitted, it defaults to HEAD. By default, -b refuses to create a new branch if it already exists. -B overrides this safeguard, resetting **<new-branch>** to **<branch>**.
--detach	With add, detach HEAD in the new working tree.
--[no-] checkout	By default, add checks out **<branch>**, however, --no-checkout can be used to suppress checkout in order to make customizations, such as configuring sparse-checkout.
-n --dry-run	With prune, do not remove anything; just report what it would remove.
--porcelain	With list, output in an easy-to-parse format for scripts. This format will remain stable across Git versions and regardless of user configuration.
-v --verbose	With prune, report all removals.
--expire **<time>**	With prune, only expire unused working trees older than **<time>**.

Section 44.1: Using a worktree

You are right in the middle of working on a new feature, and your boss comes in demanding that you fix something immediately. You may typically want use `git stash` to store your changes away temporarily. However, at this point your working tree is in a state of disarray (with new, moved, and removed files, and other bits and pieces strewn around) and you don't want to disturb your progress.

By adding a worktree, you create a temporary linked working tree to make the emergency fix, remove it when done, and then resume your earlier coding session:

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... work work work ...
$ git commit -a -m 'emergency fix for boss'
$ popd
$ rm -rf ../temp
$ git worktree prune
```

NOTE: In this example, the fix still is in the emergency-fix branch. At this point you probably want to `git merge` or `git format-patch` and afterwards remove the emergency-fix branch.

Section 44.2: Moving a worktree

Currently (as of version 2.11.0) there is no built-in functionality to move an already existing worktree. This is listed as an official bug (see https://git-scm.com/docs/git-worktree#_bugs).

To get around this limitation it is possible to perform manual operations directly in the `.git` reference files.

In this example, the main copy of the repo is living at `/home/user/project-main` and the secondary worktree is located at `/home/user/project-1` and we want to move it to `/home/user/project-2`.

Don't perform any git command in between these steps, otherwise the garbage collector might be triggered and the references to the secondary tree can be lost. Perform these steps from the start until the end without interruption:

1. 将工作树的.git文件更改为指向主树内的新位置。该文件 /home/user/project-1/.git 现在应包含以下内容：

```
gitdir: /home/user/project-main/.git/worktrees/project-2
```

2. 通过移动工作树的目录，重命名主项目中.git目录内的工作树存在于那里：

```
$ mv /home/user/project-main/.git/worktrees/project-1 /home/user/project-
main/.git/worktrees/project-2
```

3. 将引用更改为指向/home/user/project-main/.git/worktrees/project-2/gitdir内新位置。在此示例中，文件将包含以下内容：

```
/home/user/project-2/.git
```

4.最后，将你的工作树移动到新位置：

```
$ mv /home/user/project-1 /home/user/project-2
```

如果你操作正确，列出现有的工作树应该指向新的位置：

```
$ git worktree list
/home/user/project-main  23f78ad [master]
/home/user/project-2     78ac3f3 [branch-name]
```

现在运行 git worktree prune 也应该是安全的。

1. Change the worktree's `.git` file to point to the new location inside the main tree. The file `/home/user/project-1/.git` should now contain the following:

```
gitdir: /home/user/project-main/.git/worktrees/project-2
```

2. Rename the worktree inside the `.git` directory of the main project by moving the worktree's directory that exists in there:

```
$ mv /home/user/project-main/.git/worktrees/project-1 /home/user/project-
main/.git/worktrees/project-2
```

3. Change the reference inside `/home/user/project-main/.git/worktrees/project-2/gitdir` to point to the new location. In this example, the file would have the following contents:

```
/home/user/project-2/.git
```

4. Finally, move your worktree to the new location:

```
$ mv /home/user/project-1 /home/user/project-2
```

If you have done everything correctly, listing the existing worktrees should refer to the new location:

```
$ git worktree list
/home/user/project-main  23f78ad [master]
/home/user/project-2     78ac3f3 [branch-name]
```

It should now also be safe to run `git worktree prune`.

第45章：Git 远程

参数	详情
-v, --verbose	详细运行。
-m <master>	设置 head 指向远程的 <master> 分支
--mirror=fetch	引用不会存储在 refs/remotes 命名空间中，而是在本地仓库中镜像
--mirror=push	git push 将表现得像传递了 --mirror 参数
--no-tags	git fetch <name> 不会从远程仓库导入标签
-t <branch>	指定远程仓库仅跟踪 <branch>
-f	git fetch <name> 会在远程设置完成后立即运行
--tags	git fetch <name> 会从远程仓库导入所有标签
-a, --auto	符号引用的 HEAD 被设置为与远程 HEAD 相同的分支
-d，--delete	所有列出的引用都会从远程仓库中删除
--add	将 <name> 添加到当前跟踪的分支列表中（set-branches）
--add	不是更改某个 URL，而是添加新的 URL（set-url）
--all	推送所有分支。
--delete	删除所有匹配 <url> 的 URL。（set-url）
--push	操作推送 URL 而非拉取 URL
-n	远程分支不会先通过 `git ls-remote <name>` 查询，而是使用缓存的信息代替
--dry-run	报告将要被修剪的分支，但实际上不修剪它们
--prune	删除没有本地对应分支的远程分支

第45.1节：显示远程仓库

要列出所有配置的远程仓库，使用git remote。

它显示你配置的每个远程句柄的简称（别名）。

```
$ git remote
premium
premiumPro
origin
```

要显示更详细的信息，可以使用--verbose或-v标志。输出将包括远程的URL和类型（push或pull）：

```
$ git remote -v
premiumPro    https://github.com/user/CatClickerPro.git (fetch)
premiumPro    https://github.com/user/CatClickerPro.git (push)
premium    https://github.com/user/CatClicker.git (fetch)
premium    https://github.com/user/CatClicker.git (push)
origin    https://github.com/ud/starter.git (fetch)
origin    https://github.com/ud/starter.git (push)
```

第45.2节：更改你的Git仓库的远程URL

如果远程仓库已迁移，您可能需要执行此操作。更改远程 URL 的命令是：

```
git remote set-url
```


Chapter 45: Git Remote

Parameter	Details
-v, --verbose	Run verbosely.
-m <master>	Sets head to remote's <master> branch
--mirror=fetch	Refs will not be stored in refs/remotes namespace, but instead will be mirrored in the local repo
--mirror=push	`git push` will behave as if --mirror was passed
--no-tags	`git fetch <name>` does not import tags from the remote repo
-t <branch>	Specifies the remote to track *only* <branch>
-f	`git fetch <name>` is run immediately after remote is set up
--tags	`git fetch <name>` imports every tag from the remote repo
-a, --auto	The symbolic-ref's HEAD is set to the same branch as the remote's HEAD
-d, --delete	All listed refs are deleted from the remote repository
--add	Adds <name> to list of currently tracked branches (set-branches)
--add	Instead of changing some URL, new URL is added (set-url)
--all	Push all branches.
--delete	All urls matching <url> are deleted. (set-url)
--push	Push URLS are manipulated instead of fetch URLS
-n	The remote heads are not queried first with `git ls-remote <name>`, cached information is used instead
--dry-run	report what branches will be pruned, but do not actually prune them
--prune	Remove remote branches that don't have a local counterpart

Section 45.1: Display Remote Repositories

To list all configured remote repositories, use `git remote`.

It shows the short name (aliases) of each remote handle that you have configured.

```
$ git remote
premium
premiumPro
origin
```

To show more detailed information, the `--verbose` or `-v` flag can be used. The output will include the URL and the type of the remote (`push` or `pull`):

```
$ git remote -v
premiumPro    https://github.com/user/CatClickerPro.git (fetch)
premiumPro    https://github.com/user/CatClickerPro.git (push)
premium    https://github.com/user/CatClicker.git (fetch)
premium    https://github.com/user/CatClicker.git (push)
origin    https://github.com/ud/starter.git (fetch)
origin    https://github.com/ud/starter.git (push)
```

Section 45.2: Change remote url of your Git repository

You may want to do this if the remote repository is migrated. The command for changing the remote url is:

```
git remote set-url
```

它需要两个参数：一个已存在的远程名称（origin，upstream）和URL。

检查你当前的远程URL：

```
git remote -v
origin    https://bitbucket.com/develop/myrepo.git (fetch)
origin    https://bitbucket.com/develop/myrepo.git (push)
```

更改你的远程URL：

```
git remote set-url origin https://localserver/develop/myrepo.git
```

再次检查你的远程URL：

```
git remote -v
origin    https://localserver/develop/myrepo.git (fetch)
origin    https://localserver/develop/myrepo.git (push)
```

第45.3节：删除远程仓库

删除名为`<name>`的远程。所有远程跟踪分支和该远程的配置设置都会被删除。

要删除远程仓库 dev：

```
git remote rm dev
```

第45.4节：添加远程仓库

要添加远程仓库，请在本地仓库根目录使用 git remote add 命令。

要将远程Git仓库 <url> 添加为简短名称 <name>，请使用

```
git remote add <name> <url>
```

然后可以使用命令 `git fetch <name>` 来创建和更新远程跟踪分支 `<name>/<branch>`。

第45.5节：显示远程仓库的更多信息

你可以通过 git remote show <远程仓库别名> 查看远程仓库的更多信息

```
git remote show origin
```

结果：

```
远程源
获取 URL:  https://localserver/develop/myrepo.git
推送 URL:  https://localserver/develop/myrepo.git
HEAD 分支: master
远程分支：
master       已跟踪
本地分支配置为执行'git pull'：
master       与远程 master 合并
```


It takes 2 arguments: an existing remote name (origin, upstream) and the url.

Check your current remote url:

```
git remote -v
origin    https://bitbucket.com/develop/myrepo.git (fetch)
origin    https://bitbucket.com/develop/myrepo.git (push)
```

Change your remote url:

```
git remote set-url origin https://localserver/develop/myrepo.git
```

Check again your remote url:

```
git remote -v
origin    https://localserver/develop/myrepo.git (fetch)
origin    https://localserver/develop/myrepo.git (push)
```

Section 45.3: Remove a Remote Repository

Remove the remote named `<name>`. All remote-tracking branches and configuration settings for the remote are removed.

To remove a remote repository dev:

```
git remote rm dev
```

Section 45.4: Add a Remote Repository

To add a remote, use `git remote add` in the root of your local repository.

For adding a remote Git repository <url> as an easy short name <name> use

```
git remote add <name> <url>
```

The command `git fetch <name>` can then be used to create and update remote-tracking branches `<name>/<branch>`.

Section 45.5: Show more information about remote repository

You can view more information about a remote repository by `git remote show <remote repository alias>`

```
git remote show origin
```

result:

```
remote origin
Fetch URL:  https://localserver/develop/myrepo.git
Push  URL:  https://localserver/develop/myrepo.git
HEAD branch: master
Remote branches:
  master      tracked
Local branches configured for 'git pull':
  master      merges with remote master
```

```
本地引用配置为执行'git push'：
master       推送到 master       （最新）
```

第 45.6 节：重命名远程仓库

将名为 <old> 的远程重命名为 <new>。所有远程跟踪分支和远程的配置设置都会被更新。

将远程分支名 dev 重命名为 dev1 ：

```
git remote rename dev dev1
```



```
Local refs configured for 'git push':
  master      pushes to master      (up to date)
```

Section 45.6: Rename a Remote Repository

Rename the remote named **<old>** to **<new>**. All remote-tracking branches and configuration settings for the remote are updated.

To rename a remote branch name dev to dev1 :

```
git remote rename dev dev1
```

第46章：Git大文件存储（LFS）

第46.1节：声明某些文件类型以外部存储

使用Git LFS的常见工作流程是通过基于规则的系统声明哪些文件被拦截，就像.gitignore文件一样。

通常使用通配符来选择某些文件类型进行统一跟踪。

例如：git lfs track "*.psd"

当添加并提交与上述模式匹配的文件后，推送到远程时，该文件将被单独上传，远程仓库中的文件将被指针替代。

文件被LFS跟踪后，你的.gitattributes文件将相应更新。Github建议提交本地的.gitattributes文件，而不是使用全局的.gitattributes文件，以帮助确保在处理不同项目时不会出现问题。

第46.2节：为所有克隆设置LFS配置

要设置适用于所有克隆的LFS选项，请在仓库根目录创建并提交名为.lfsconfig的文件。该文件可以以与.git/config中允许的方式相同的方式指定LFS选项。

例如，要默认排除某个文件不被LFS拉取，请创建并提交包含以下内容的.lfsconfig文件：

```
[lfs]
    fetchexclude = ReallyBigFile.wav
```

第46.3节：安装LFS

通过Homebrew或从网站下载并安装。

对于Brew，

```
brew install git-lfs
git lfs install
```

通常你还需要对托管远程仓库的服务进行一些设置，以使其支持lfs。这对于每个主机都会有所不同，但通常只需勾选一个表示你想使用git lfs的选项。

Chapter 46: Git Large File Storage (LFS)

Section 46.1: Declare certain file types to store externally

A common workflow for using Git LFS is to declare which files are intercepted through a rules-based system, just like `.gitignore` files.

Much of time, wildcards are used to pick certain file-types to blanket track.

e.g. `git lfs track "*.psd"`

When a file matching the above pattern is added them committed, when it is then pushed to the remote, it will be uploaded separately, with a pointer replacing the file in the remote repository.

After a file has been tracked with lfs, your `.gitattributes` file will be updated accordingly. Github recommends committing your local `.gitattributes` file, rather than working with a global `.gitattributes` file, to help ensure you don't have any issues when working with different projects.

Section 46.2: Set LFS config for all clones

To set LFS options that apply to all clones, create and commit a file named `.lfsconfig` at the repository root. This file can specify LFS options the same way as allowed in `.git/config`.

For example, to exclude a certain file from LFS fetches be default, create and commit `.lfsconfig` with the following contents:

```
[lfs]
    fetchexclude = ReallyBigFile.wav
```

Section 46.3: Install LFS

Download and install, either via Homebrew, or from website.

For Brew,

```
brew install git-lfs
git lfs install
```

Often you will also need to do some setup on the service that hosts your remote to allow it to work with lfs. This will be different for each host, but will likely just be checking a box saying you want to use git lfs.

第47章：Git补丁

参数	详情
(<mbox>|<Maildir>)...	要读取补丁的邮箱文件列表。如果你不提供此参数，命令将从标准输入读取。如果你提供目录，它们将被视为Maildir。
-s，--signoff	在提交信息中添加一行 Signed-off-by:，使用你自己的提交者身份。
-q, --quiet	保持安静。只打印错误信息。
-u，--utf8	向 git mailinfo 传递 -u 标志。从电子邮件中获取的拟提交日志信息将重新编码为 UTF-8 编码（配置变量 i18n.commitencoding 可用于指定项目首选编码，如果不是 UTF-8）。你可以使用 --no-utf8 来覆盖此设置。
--no-utf8	向 git mailinfo 传递 -n 标志。
-3，--3way	当补丁无法干净地应用时，如果补丁记录了它应该应用的 blob 的身份且我们本地有这些 blob，则回退到三方合并。
--ignore-date，--ignore-space-change，--ignore-whitespace，--whitespace=<option>，-C<n>，-p<n>，--directory=<dir>，--exclude=<path>，--include=<path>，--reject	这些标志会传递给执行补丁的 git apply 程序。
--patch-format	默认情况下，命令会尝试自动检测补丁格式。此选项允许用户绕过自动检测，指定补丁应被解释为的格式。有效格式包括mbox、stgit、stgit-series和hg。
-i, --interactive	以交互方式运行。
--committer-date-is-author-date	默认情况下，命令会将电子邮件中的日期记录为提交作者日期，并使用提交创建时间作为提交者日期。这允许用户通过使用与作者日期相同的值来伪造提交者日期。
--ignore-date	默认情况下，命令会将电子邮件中的日期记录为提交作者日期，并使用提交创建时间作为提交者日期。这允许用户通过使用与提交者日期相同的值来伪造作者日期。
--skip	跳过当前补丁。此操作仅在重新启动中止的补丁时有意义。
-S[<keyid>]，--gpg-sign[=<keyid>]	对提交进行GPG签名。
--continue，-r，--resolved	在补丁失败后（例如尝试应用冲突的补丁），用户已手动应用补丁，且索引文件存储了应用结果。使用从电子邮件消息中提取的作者信息和提交日志以及当前索引文件进行提交，并继续操作。
--resolvemsg=<msg>	当补丁失败发生时，<msg> 会在退出前打印到屏幕上。此信息将覆盖标准提示，告知您使用--continue或--skip来处理失败。此功能仅供git rebase和git am之间的内部使用。
--abort	恢复原始分支并中止补丁操作。

第47.1节：创建补丁

创建补丁有两个步骤。

1. 进行更改并提交。

Chapter 47: Git Patch

Parameter	Details
(<mbox>|<Maildir>)...	The list of mailbox files to read patches from. If you do not supply this argument, the command reads from the standard input. If you supply directories, they will be treated as Maildirs.
-s, --signoff	Add a Signed-off-by: line to the commit message, using the committer identity of yourself.
-q, --quiet	Be quiet. Only print error messages.
-u, --utf8	Pass `-u` flag to **`git mailinfo`**. The proposed commit log message taken from the e-mail is re-coded into UTF-8 encoding (configuration variable `i18n.commitencoding` can be used to specify project's preferred encoding if it is not UTF-8). You can use `--no-utf8` to override this.
--no-utf8	Pass -n flag to git mailinfo.
-3, --3way	When the patch does not apply cleanly, fall back on 3-way merge if the patch records the identity of blobs it is supposed to apply to and we have those blobs available locally.
--ignore-date, --ignore-space-change, --ignore-whitespace, --whitespace=<option>, -C<n>, -p<n>, --directory=<dir>, --exclude=<path>, --include=<path>, --reject	These flags are passed to the git apply program that applies the patch.
--patch-format	By default the command will try to detect the patch format automatically. This option allows the user to bypass the automatic detection and specify the patch format that the patch(es) should be interpreted as. Valid formats are `mbox`, `stgit`, `stgit-series`, and `hg`.
-i, --interactive	Run interactively.
--committer-date-is-author-date	By default the command records the date from the e-mail message as the commit author date, and uses the time of commit creation as the committer date. This allows the user to lie about the committer date by using the same value as the author date.
--ignore-date	By default the command records the date from the e-mail message as the commit author date, and uses the time of commit creation as the committer date. This allows the user to lie about the author date by using the same value as the committer date.
--skip	Skip the current patch. This is only meaningful when restarting an aborted patch.
-S[<keyid>], --gpg-sign[=<keyid>]	GPG-sign commits.
--continue, -r, --resolved	After a patch failure (e.g. attempting to apply conflicting patch), the user has applied it by hand and the index file stores the result of the application. Make a commit using the authorship and commit log extracted from the e-mail message and the current index file, and continue.
--resolvemsg=<msg>	When a patch failure occurs, **`<msg>`** will be printed to the screen before exiting. This overrides the standard message informing you to use `--continue` or `--skip` to handle the failure. This is solely for internal use between **`git rebase`** and **`git am`**.
--abort	Restore the original branch and abort the patching operation.

Section 47.1: Creating a patch

To create a patch, there are two steps.

1. Make your changes and commit them.

2. 运行git format-patch<commit-reference>以转换自提交<commit-reference>以来的所有提交（不包括它）到补丁文件中。

例如，如果应从最近的两个提交生成补丁：

```
git format-patch HEAD~~
```

这将创建两个文件，每个提交一个，自HEAD~~以来，如下所示：

```
0001-hello_world.patch
0002-beginning.patch
```

第47.2节：应用补丁

我们可以使用git apply some.patch将.patch文件中的更改应用到当前工作目录。它们将处于未暂存状态，需要提交。

要将补丁作为提交（及其提交信息）应用，请使用

```
git am some.patch
```

将所有补丁文件应用到代码树：

```
git am *.patch
```


2. Run `git format-patch <commit-reference>` to convert all commits since the commit <commit-reference> (not including it) into patch files.

For example, if patches should be generated from the latest two commits:

```
git format-patch HEAD~~
```

This will create 2 files, one for each commit since HEAD~~, like this:

```
0001-hello_world.patch
0002-beginning.patch
```

Section 47.2: Applying patches

We can use `git apply some.patch` to have the changes from the `.patch` file applied to your current working directory. They will be unstaged and need to be committed.

To apply a patch as a commit (with its commit message), use

```
git am some.patch
```

To apply all patch files to the tree:

```
git am *.patch
```

第48章：Git统计

参数	详情
-n, --numbered	根据每个作者的提交次数排序输出，而非按字母顺序
-s, --summary	仅提供提交计数摘要
-e, --email	显示每个作者的电子邮件地址
--format[=<format>]	不使用提交主题，而是使用其他信息来描述每个提交。<format> 可以是 gitlog 的 --format 选项接受的任何字符串。
-w[<宽度>[,<缩进1>[,<缩进2>]]]	通过在width处换行来换行输出。每个条目的第一行缩进indent1个空格，后续行缩进indent2个空格。 每个条目的第一行缩进indent1个空格，后续行缩进indent2个空格。
<revision range>	仅显示指定修订范围内的提交。默认显示直到当前提交的整个历史记录。
[--] <path>	仅显示解释匹配path文件来源的提交。 路径可能需要以"-- "为前缀，以将其与选项或修订范围分开。

第48.1节：每个开发者的代码行数

```
git ls-tree -r HEAD | sed -Ee 's/^.{53}//' | \
while read filename; do file "$filename"; done | \
grep -E ': .*text' | sed -E -e 's/: .*//' | \
while read filename; do git blame --line-porcelain "$filename"; done | \
sed -n 's/^author //p' | \
sort | uniq -c | sort -rn
```

第48.2节：列出每个分支及其最后修订日期

```
for k in `git branch -a | sed s/^..//`; do echo -e `git log -1 --pretty=format:"%Cgreen%ci
%Cblue%cr%Creset" $k --\"$k";done | sort
```

第48.3节：每个开发者的提交记录

Git shortlog 用于总结 git 日志输出并按作者分组提交记录。

默认情况下，会显示所有提交信息，但参数 --summary 或 -s 会跳过提交信息，列出作者及其提交总数。

--numbered 或 -n 会将排序方式从按作者字母升序改为按提交数降序。

```
git shortlog -sn        #作者姓名及提交次数

git shortlog -sne       #作者姓名及邮箱和提交次数
```

或

```
git log --pretty=format:%ae \
| gawk -- '{ ++c[$0]; } END { for(cc in c) printf "%5d %s",c[cc],cc; }'
```

注意：同一人的提交如果姓名和/或邮箱拼写不同，可能不会被归为一组。例如 John Doe 和 Johnny Doe 会分别出现在列表中。为了解决这个问题，

Chapter 48: Git statistics

Parameter	Details
-n, --numbered	Sort output according to the number of commits per author instead of alphabetic order
-s, --summary	Only provide a commit count summary
-e, --email	Show the email address of each author
--format[=<format>]	Instead of the commit subject, use some other information to describe each commit. <format> can be any string accepted by the --format option of git log.
-w[<width>[,<indent1>[,<indent2>]]]	Linewrap the output by wrapping each line at width. The first line of each entry is indented by indent1 number of spaces, and subsequent lines are indented by indent2 spaces.
<revision range>	Show only commits in the specified revision range. Default to the whole history until the current commit.
[--] <path>	Show only commits that explain how the files matching path came to be. Paths may need to be prefixed with "-- " to separate them from options or the revision range.

Section 48.1: Lines of code per developer

```
git ls-tree -r HEAD | sed -Ee 's/^.{53}//' | \
while read filename; do file "$filename"; done | \
grep -E ': .*text' | sed -E -e 's/: .*//' | \
while read filename; do git blame --line-porcelain "$filename"; done | \
sed -n 's/^author //p' | \
sort | uniq -c | sort -rn
```

Section 48.2: Listing each branch and its last revision's date

```
for k in `git branch -a | sed s/^..//`; do echo -e `git log -1 --pretty=format:"%Cgreen%ci
%Cblue%cr%Creset" $k --`\\t"$k";done | sort
```

Section 48.3: Commits per developer

Git shortlog is used to summarize the git log outputs and group the commits by author.

By default, all commit messages are shown but argument --summary or -s skips the messages and gives a list of authors with their total number of commits.

--numbered or -n changes the ordering from alphabetical (by author ascending) to number of commits descending.

```
git shortlog -sn        #Names and Number of commits

git shortlog -sne       #Names along with their email ids and the Number of commits
```

or

```
git log --pretty=format:%ae \
| gawk -- '{ ++c[$0]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'
```

Note: Commits by the same person may not be grouped together where their name and/or email address has been spelled differently. For example John Doe and Johnny Doe will appear separately in the list. To resolve this,

请参阅.mailmap功能。

第48.4节：按日期统计提交次数

```
git log --pretty=format:"%ai" | awk '{print " : "$1}' | sort -r | uniq -c
```

第48.5节：分支中的提交总数

```
git log  --pretty=oneline |wc -l
```

第48.6节：以美观格式列出所有提交

```
git log --pretty=format:"%Cgreen%ci %Cblue%cn  %Cgreen%cr%Creset %s"
```

这将以每行一条的形式，显示所有提交的日期、用户和提交信息，便于查看。

--pretty 选项有许多占位符，每个都以%开头。所有选项可以在这里找到

第48.7节：查找计算机上的所有本地Git仓库

要列出计算机上所有的git仓库位置，可以运行以下命令

```
find $HOME -type d -name ".git"
```

假设你有 locate，这样会快得多：

```
locate .git |grep git$
```

如果你有 gnu locate 或 mlocate，这将只选择 git 目录：

```
locate -ber \\.git$
```

第48.8节：显示每个作者的提交总数

为了获取每个开发者或贡献者在仓库中的提交总数，你可以
简单地使用 `git shortlog`：

```
git shortlog -s
```

该命令会显示作者名称及其提交次数。

此外，如果你想统计所有分支的结果，可以在命令中添加 --all 参数：

```
git shortlog -s --all
```


refer to the `.mailmap` feature.

Section 48.4: Commits per date

```
git log --pretty=format:"%ai" | awk '{print " : "$1}' | sort -r | uniq -c
```

Section 48.5: Total number of commits in a branch

```
git log  --pretty=oneline |wc -l
```

Section 48.6: List all commits in pretty format

```
git log --pretty=format:"%Cgreen%ci %Cblue%cn  %Cgreen%cr%Creset %s"
```

This will give a nice overview of all commits (1 per line) with date, user and commit message.

The `--pretty` option has many placeholders, each starting with `%`. All options can be found here

Section 48.7: Find All Local Git Repositories on Computer

To list all the git repository locations on your you can run the following

```
find $HOME -type d -name ".git"
```

Assuming you have `locate`, this should be much faster:

```
locate .git |grep git$
```

If you have `gnu locate` or `mlocate`, this will select only the git dirs:

```
locate -ber \\.git$
```

Section 48.8: Show the total number of commits per author

In order to get the total number of commits that each developer or contributor has made on a repository, you can simply use the `git shortlog`:

```
git shortlog -s
```

which provides the author names and number of commits by each one.

Additionally, if you want to have the results calculated on all branches, add `--all` flag to the command:

```
git shortlog -s --all
```

第49章：git send-email

第49.1节：使用git send-email和Gmail

背景：如果你参与像Linux内核这样的项目，不是通过拉取请求，而是需要将你的提交发送到邮件列表进行审核。本文介绍如何使用git-send-email配合Gmail。

将以下内容添加到你的.gitconfig文件中：

```
[sendemail]
smtpserver = smtp.googlemail.com
    smtpencryption = tls
smtpserverport = 587
     smtpuser = name@gmail.com
```

然后在网页上操作：进入Google -> 我的账户 -> 已连接的应用和网站 -> 允许不够安全的应用 -> 开启

创建补丁集：

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

然后将补丁发送到邮件列表：

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

创建并发送补丁的更新版本（此例中为版本2）：

```
git format-patch -v 2 HEAD~~~~  ......
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

第49.2节：撰写

```
--from                   * 发件人邮箱：
--[no-]to               * 收件人邮箱：
--[no-]cc               * 抄送邮箱：
--[no-]bcc              * 密送邮箱：
--subject               * 邮件主题：
--in-reply-to           * 电子邮件 "In-Reply-To:"
--[no-]xmailer               * 添加 "X-Mailer:" 头部（默认）。
--[no-]annotate              * 在编辑器中审查将要发送的每个补丁。
--compose                    * 打开编辑器以编写引言。
--compose-encoding         * 假定引言的编码。
--8bit-encoding         * 如果未声明，假定8位邮件的编码。
--transfer-encoding     * 使用的传输编码（quoted-printable，8bit，base64）。
```

第49.3节：通过邮件发送补丁

假设你对一个项目（这里是ulogd2，官方分支是git-svn）有很多提交，并且你想将你的补丁集发送到邮件列表devel@netfilter.org。为此，只需在git目录的根目录打开一个终端，并使用：

```
git format-patch --stat -p --raw --signoff  --subject-prefix="ULOGD PATCH" -o /tmp/ulogd2/ -n git-svn
```


Chapter 49: git send-email

Section 49.1: Use git send-email with Gmail

Background: if you work on a project like the Linux kernel, rather than make a pull request you will need to submit your commits to a listserv for review. This entry details how to use git-send email with Gmail.

Add the following to your .gitconfig file:

```
[sendemail]
    smtpserver = smtp.googlemail.com
    smtpencryption = tls
    smtpserverport = 587
    smtpuser = name@gmail.com
```

Then on the web: Go to Google -> My Account -> Connected Apps & Sites -> Allow less secure apps -> Switch ON

To create a patch set:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

Then send the patches to a listserv:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

To create and send updated version (version 2 in this example) of the patch:

```
git format-patch -v 2 HEAD~~~~  ......
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

Section 49.2: Composing

```
--from                   * Email From:
--[no-]to                * Email To:
--[no-]cc                * Email Cc:
--[no-]bcc               * Email Bcc:
--subject                * Email "Subject:"
--in-reply-to            * Email "In-Reply-To:"
--[no-]xmailer               * Add "X-Mailer:" header (default).
--[no-]annotate              * Review each patch that will be sent in an editor.
--compose                    * Open an editor for introduction.
--compose-encoding       * Encoding to assume for introduction.
--8bit-encoding          * Encoding to assume 8bit mails if undeclared
--transfer-encoding      * Transfer encoding to use (quoted-printable, 8bit, base64)
```

Section 49.3: Sending patches by mail

Suppose you've got a lot of commit against a project (here ulogd2, official branch is git-svn) and that you wan to send your patchset to the Mailling list devel@netfilter.org. To do so, just open a shell at the root of the git directory and use:

```
git format-patch --stat -p --raw --signoff  --subject-prefix="ULOGD PATCH" -o /tmp/ulogd2/ -n git-svn
```

```
git send-email --compose --no-chain-reply-to --to devel@netfilter.org /tmp/ulogd2/
```

第一个命令将从 /tmp/ulogd2/ 中的补丁创建一系列带有统计报告的邮件，第二个命令将启动你的编辑器来撰写补丁集的介绍邮件。为了避免糟糕的邮件线程，可以使用：

```
git config sendemail.chainreplyto false
```

source

```
git send-email --compose --no-chain-reply-to --to devel@netfilter.org /tmp/ulogd2/
```

First command will create a serie of mail from patches in /tmp/ulogd2/ with statistic report and second will start your editor to compose an introduction mail to the patchset. To avoid awful threaded mail series, one can use :

```
git config sendemail.chainreplyto false
```

source

第50章：Git 图形界面客户端

第50.1节：gitk 和 git-gui

当你安装 Git 时，也会获得它的可视化工具 gitk 和 git-gui。

gitk 是一个图形化的历史查看器。可以把它看作是 git log 和 git grep 的强大 GUI 外壳。当你想查找过去发生的某些事情，或可视化你的项目历史时，这是你应该使用的工具。

从命令行调用 gitk 最简单。只需进入一个 Git 仓库目录，然后输入：

```
$ gitk [git log 选项]
```

Gitk 接受许多命令行选项，其中大多数会传递给底层的 git log 操作。可能最有用的是 --all 标志，它告诉 gitk 显示从任何引用（ref）可达的提交，而不仅仅是 HEAD。Gitk 的界面如下所示：

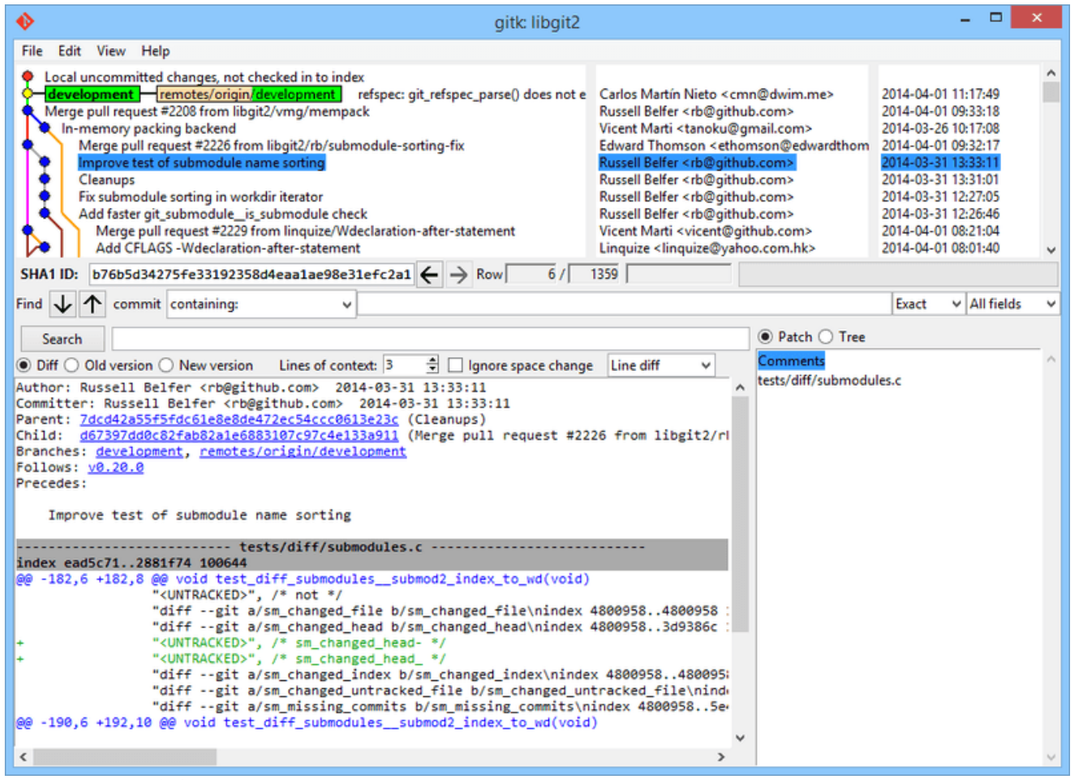


图1-1. gitk 历史查看器。

顶部看起来有点像 git log --graph 的输出；每个点代表一次提交，线条表示父子关系，引用以彩色框显示。黄色点代表 HEAD，红色点代表尚未成为提交的更改。底部是所选提交的视图；左侧是注释和补丁，右侧是摘要视图。中间是一组用于搜索历史的控件。

你可以通过右键点击分支名称或提交信息来访问许多与 git 相关的功能。例如，切换到不同的分支或挑选提交都可以通过一次点击轻松完成。

Chapter 50: Git GUI Clients

Section 50.1: gitk and git-gui

When you install Git, you also get its visual tools, gitk and git-gui.

`gitk` is a graphical history viewer. Think of it like a powerful GUI shell over git log and git grep. This is the tool to use when you're trying to find something that happened in the past, or visualize your project's history.

Gitk is easiest to invoke from the command-line. Just cd into a Git repository, and type:

```
$ gitk [git log options]
```

Gitk accepts many command-line options, most of which are passed through to the underlying git log action. Probably one of the most useful is the `--all` flag, which tells gitk to show commits reachable from any ref, not just HEAD. Gitk's interface looks like this:

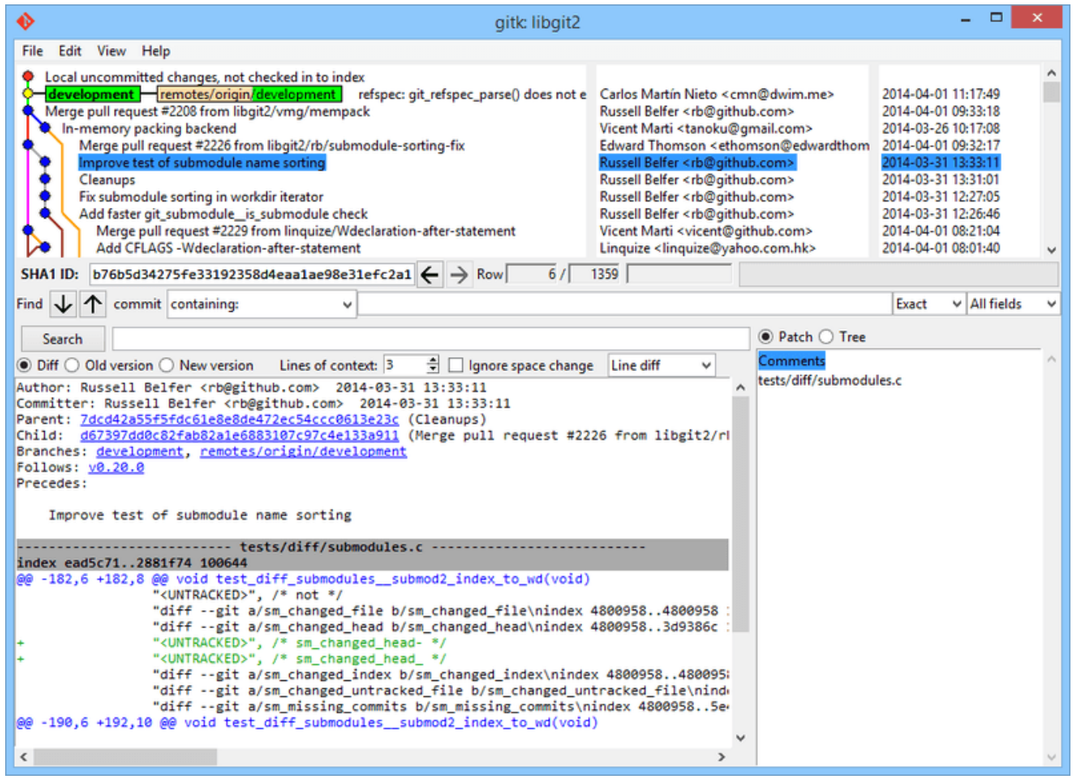


Figure 1-1. The gitk history viewer.

On the top is something that looks a bit like the output of git log --graph; each dot represents a commit, the lines represent parent relationships, and refs are shown as colored boxes. The yellow dot represents HEAD, and the red dot represents changes that are yet to become a commit. At the bottom is a view of the selected commit; the comments and patch on the left, and a summary view on the right. In between is a collection of controls used for searching history.

You can access many git related functions via right-click on a branch name or a commit message. For example checking out a different branch or cherry pick a commit is easily done with one click.

另一方面，git-gui 主要是一个用于制作提交的工具。它同样最容易通过命令行调用：

```
$ git gui
```

它看起来大致如下：

git-gui 提交工具。

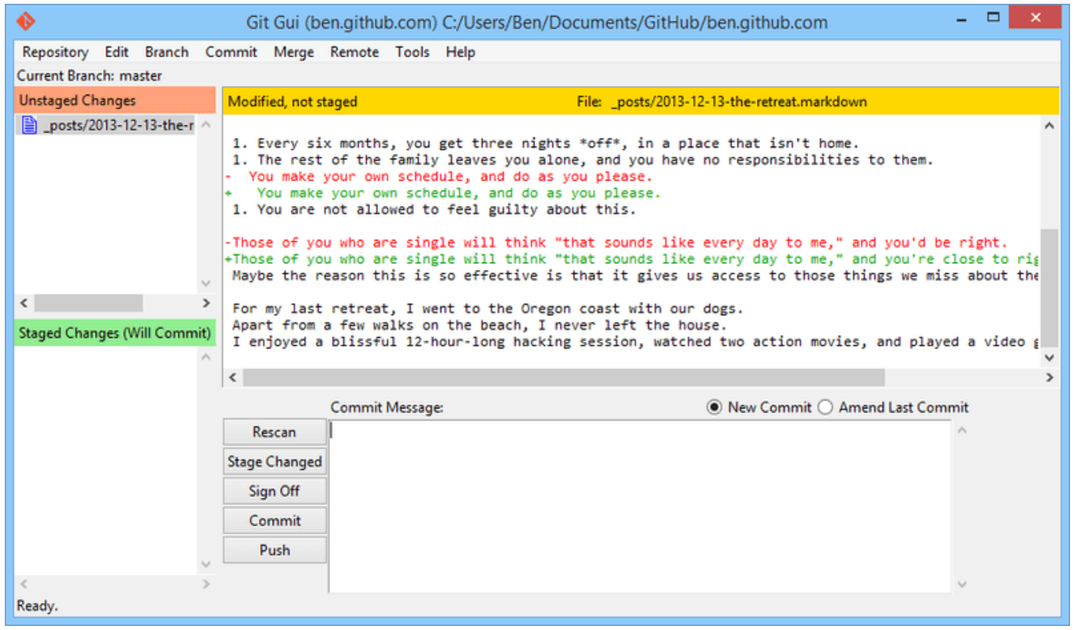


图 1-2. git-gui 提交工具。

左侧是索引；未暂存的更改在上方，已暂存的更改在下方。你可以通过点击图标在两种状态间移动整个文件，或者点击文件名选择文件进行查看。

右上方是差异视图，显示当前选中文件的更改。你可以通过右键点击此区域来暂存单个块（或单行）。

右下方是消息和操作区域。在文本框中输入你的提交信息，然后点击“提交”类似于 git commit 的操作。你也可以选择通过选择“修正”单选按钮来修改上一次提交，这会用上一次提交的内容更新“暂存的更改”区域。然后你可以简单地暂存或取消暂存一些更改，修改提交信息，点击“提交”再次替换旧的提交为新的提交。

gitk 和 git-gui 是面向任务的工具示例。它们各自针对特定用途（分别是查看历史和创建提交）进行定制，省略了该任务不需要的功能。

来源：https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces

第50.2节：GitHub Desktop

网站：https://desktop.github.com
价格：免费

`git-gui`, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

```
$ git gui
```

And it looks something like this:

The `git-gui` commit tool.

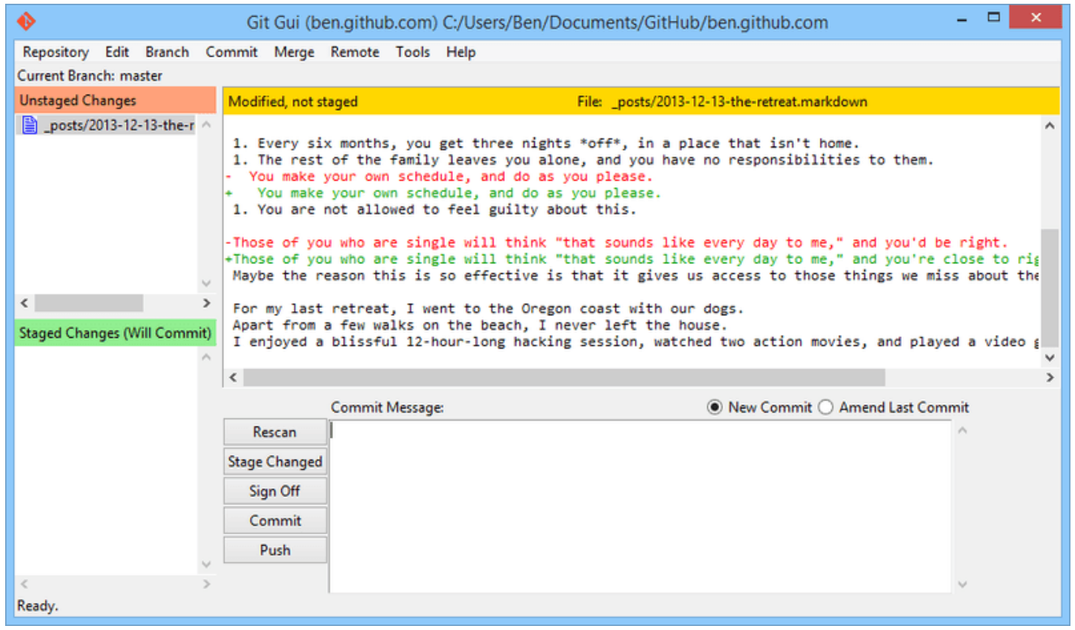


Figure 1-2. The git-gui commit tool.

On the left is the index; unstaged changes are on top, staged changes on the bottom. You can move entire files between the two states by clicking on their icons, or you can select a file for viewing by clicking on its name.

At top right is the diff view, which shows the changes for the currently-selected file. You can stage individual hunks (or individual lines) by right-clicking in this area.

At the bottom right is the message and action area. Type your message into the text box and click “Commit” to do something similar to git commit. You can also choose to amend the last commit by choosing the “Amend” radio button, which will update the “Staged Changes” area with the contents of the last commit. Then you can simply stage or unstage some changes, alter the commit message, and click “Commit” again to replace the old commit with a new one.

gitk and git-gui are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.

Source: https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces

Section 50.2: GitHub Desktop

Website: https://desktop.github.com
Price: free

平台：OS X 和 Windows
开发者：GitHub

第50.3节：Git Kraken

网站：https://www.gitkraken.com
价格：60美元/年（对开源、教育、非营利、初创或个人使用免费）
平台：Linux、OS X、Windows
开发者：Axosoft

第50.4节：SourceTree

网站：https://www.sourcetreeapp.com
价格：免费（需要账户）
平台：OS X 和 Windows
开发者：Atlassian

第50.5节：Git Extensions

网站：https://gitextensions.github.io
价格：免费
平台：Windows

第50.6节：SmartGit

网站：http://www.syntevo.com/smartgit/
价格：仅限非商业用途免费。永久许可证价格为99美元
平台：Linux、OS X、Windows
开发者：syntevo

Platforms: OS X and Windows
Developed by: GitHub

Section 50.3: Git Kraken

Website:https://www.gitkraken.com
Price: $60/years (free for For open source, education, non-profit, startups or personal use)
Platforms: Linux, OS X, Windows
Developed by: Axosoft

Section 50.4: SourceTree

Website: https://www.sourcetreeapp.com
Price: free (account is necessary)
Platforms: OS X and Windows
Developer: Atlassian

Section 50.5: Git Extensions

Website: https://gitextensions.github.io
Price: free
Platform: Windows

Section 50.6: SmartGit

Website: http://www.syntevo.com/smartgit/
Price: Free for non-commercial use only. A perpetual license costs 99 USD
Platforms: Linux, OS X, Windows
Developed by: syntevo

第51章：Reflog - 恢复git log中未显示的提交

第51.1节：从错误的变基中恢复

假设你开始了一个交互式变基：

```
git rebase --interactive HEAD~20
```

但不小心将一些不想丢失的提交压缩或丢弃了，然后完成了变基。要恢复，执行 `git reflog`，可能会看到如下输出：

```
aaaaaaa HEAD@{0} rebase -i (finish): 返回到 refs/head/master
bbbbbbb HEAD@{1} rebase -i (squash): 修复解析错误
...
ccccccc HEAD@{n} rebase -i (start): 检出 HEAD~20
ddddddd HEAD@{n+1} ...
...
```

在这种情况下，最后的提交 `ddddddd`（或 `HEAD@{n+1}`）是你*变基前*分支的最新提交。因此，要恢复该提交（以及所有父提交，包括那些意外压缩或丢弃的），执行：

```
$ git checkout HEAD@{n+1}
```

然后你可以使用 git checkout -b [branch] 在该提交处创建一个新分支。更多信息请参见分支章节。

Chapter 51: Reflog - Restoring commits not shown in git log

Section 51.1: Recovering from a bad rebase

Suppose that you had started an interactive rebase:

```
git rebase --interactive HEAD~20
```

and by mistake, you squashed or dropped some commits that you didn't want to lose, but then completed the rebase. To recover, do `git reflog`, and you might see some output like this:

```
aaaaaaa HEAD@{0} rebase -i (finish): returning to refs/head/master
bbbbbbb HEAD@{1} rebase -i (squash): Fix parse error
...
ccccccc HEAD@{n} rebase -i (start): checkout HEAD~20
ddddddd HEAD@{n+1} ...
...
```

In this case, the last commit, `ddddddd` (or `HEAD@{n+1}`) is the tip of your *pre-rebase* branch. Thus, to recover that commit (and all parent commits, including those accidentally squashed or dropped), do:

```
$ git checkout HEAD@{n+1}
```

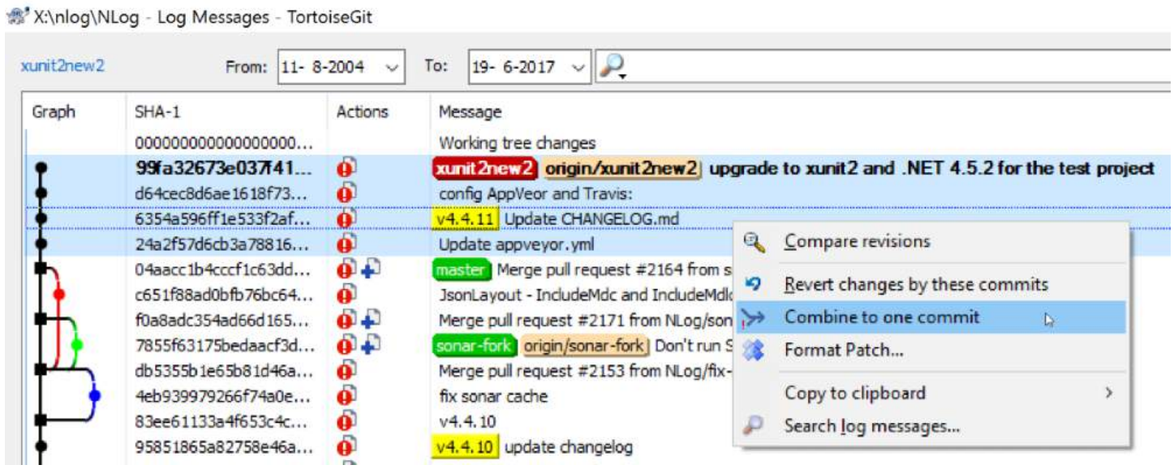
You can then create a new branch at that commit with `git checkout -b [branch]`. See Branching for more information.

第52章：TortoiseGit

第52.1节：压缩提交

简单方法

如果您的选择中有合并提交，则此方法无效



高级方法

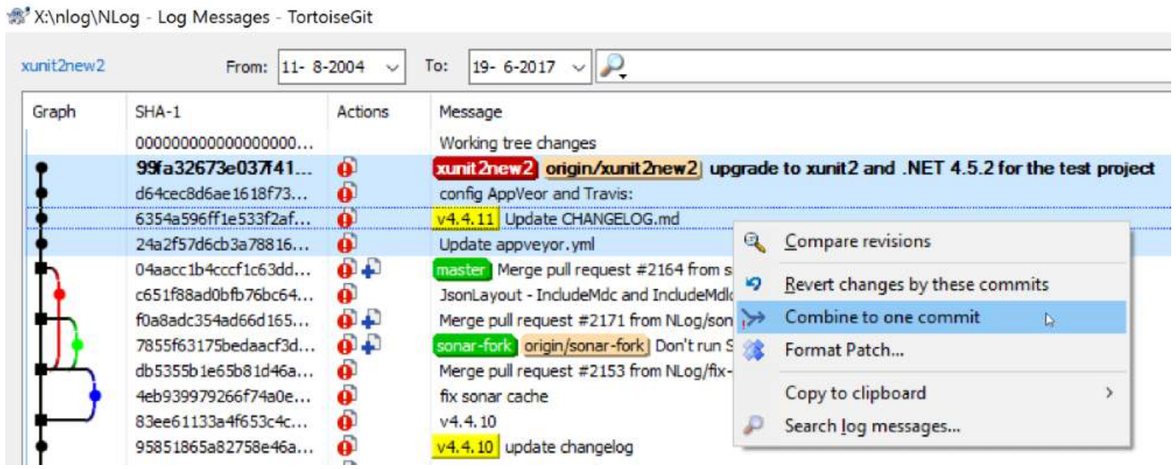
启动变基对话框：

Chapter 52: TortoiseGit

Section 52.1: Squash commits

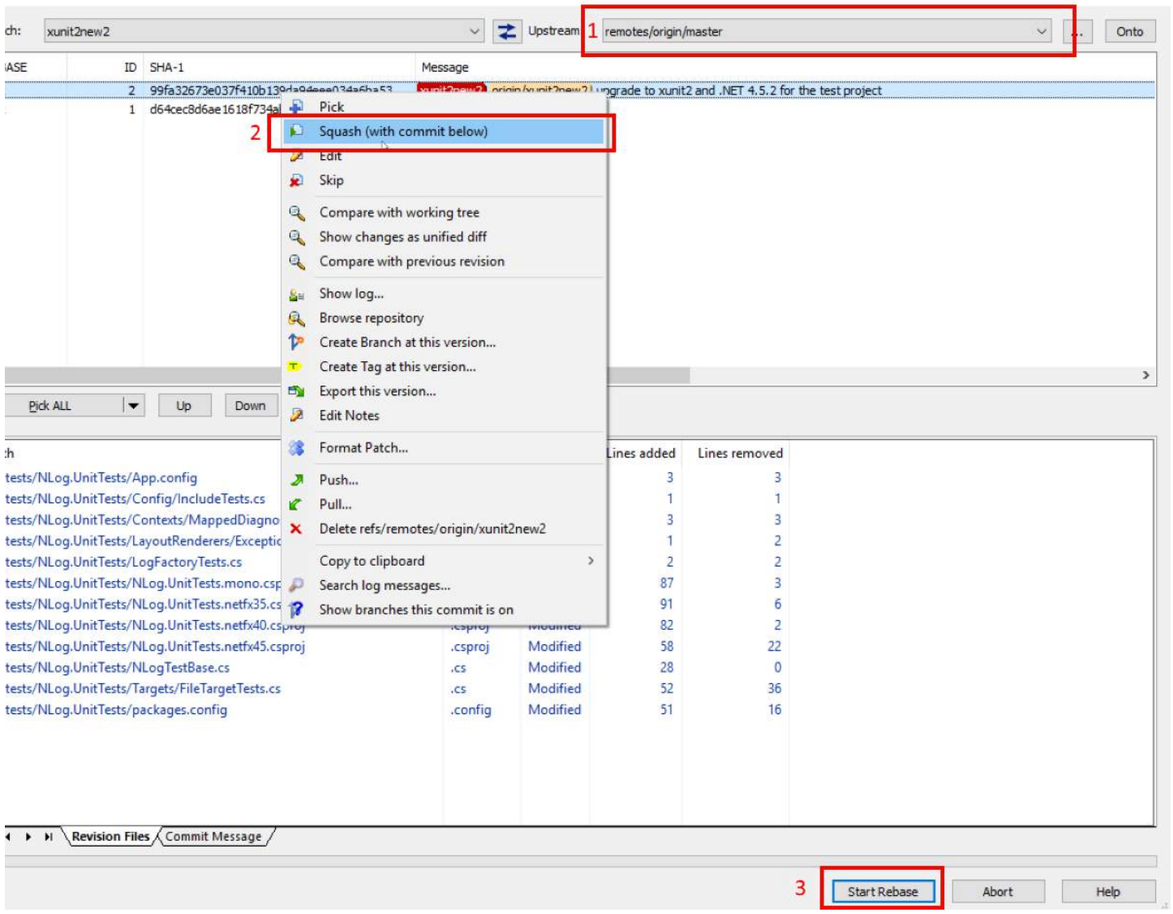
The easy way

This won't work if there are merge commits in your selection



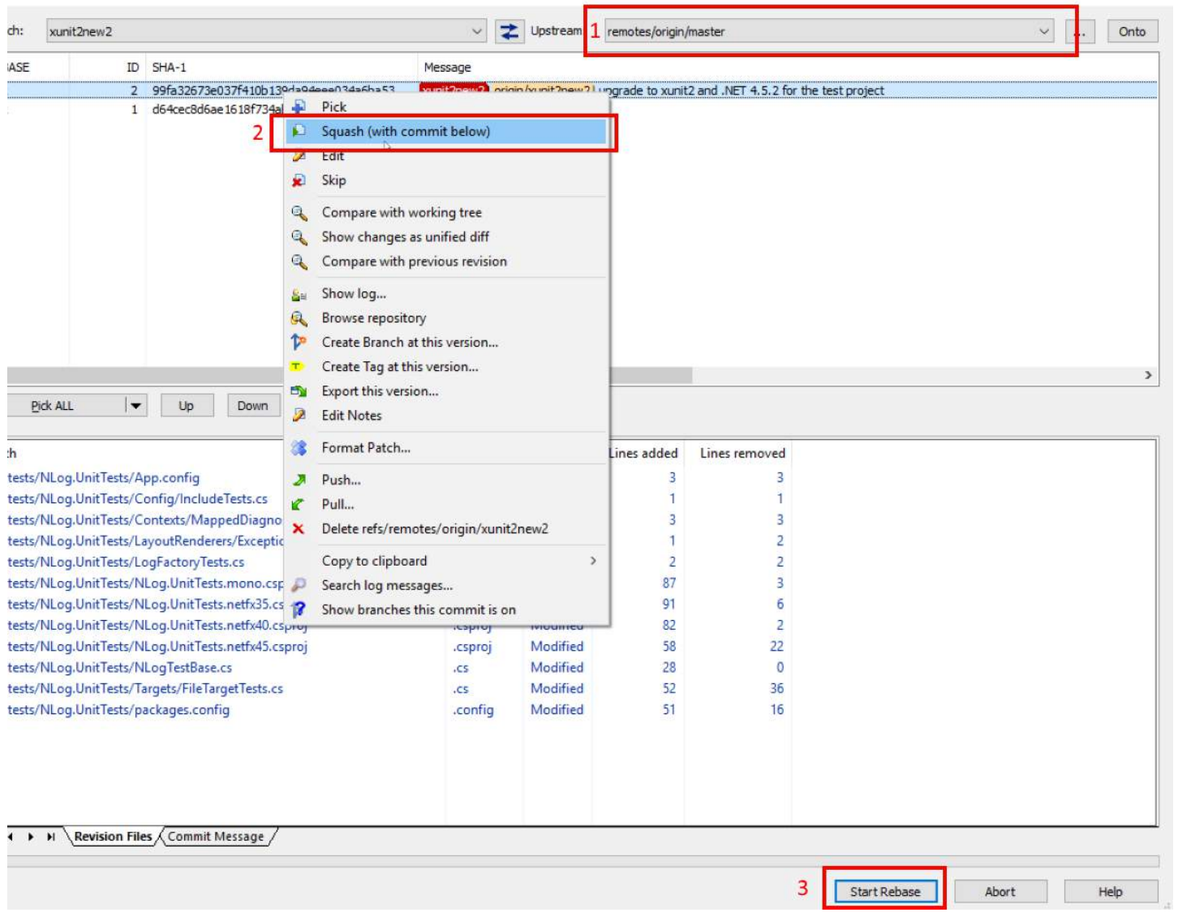
The advanced way

Start the rebase dialog:



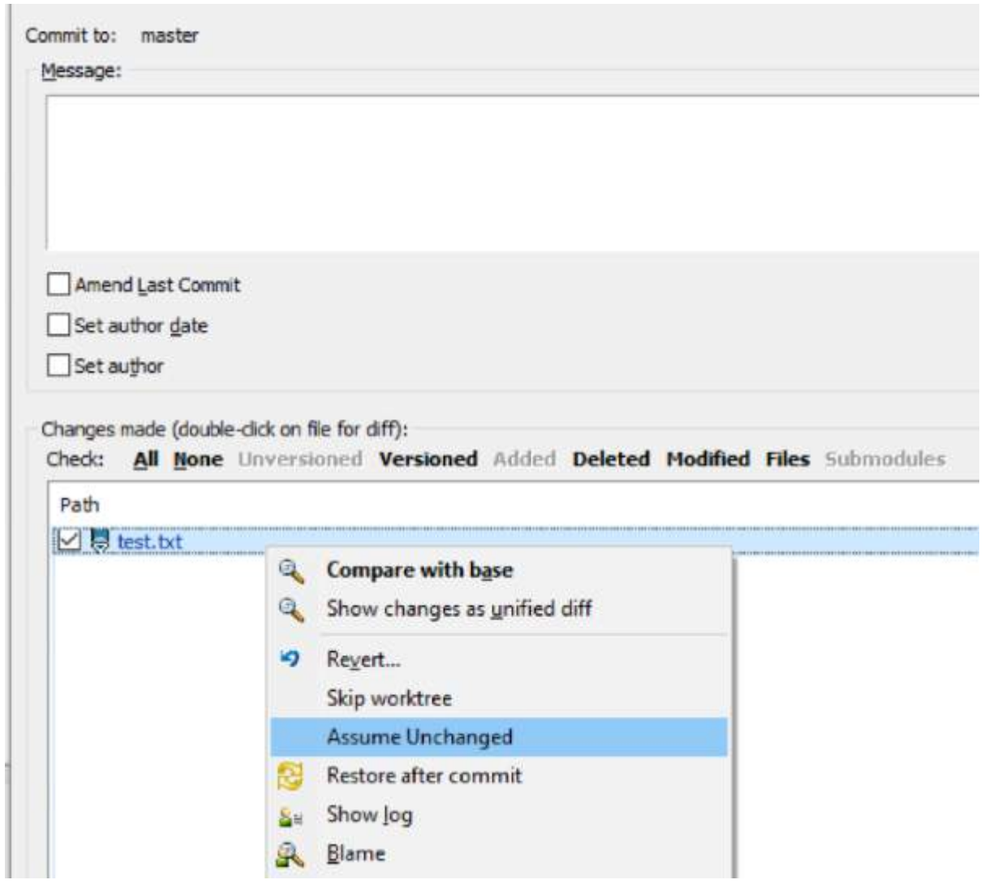
第52.2节：假设未更改

如果文件被更改，但您不想提交它，请将该文件设置为“假设未更改”



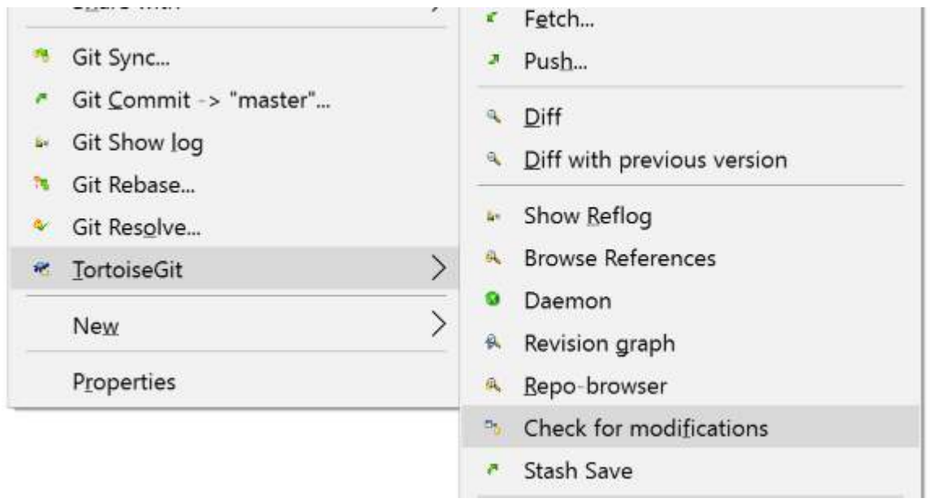
Section 52.2: Assume unchanged

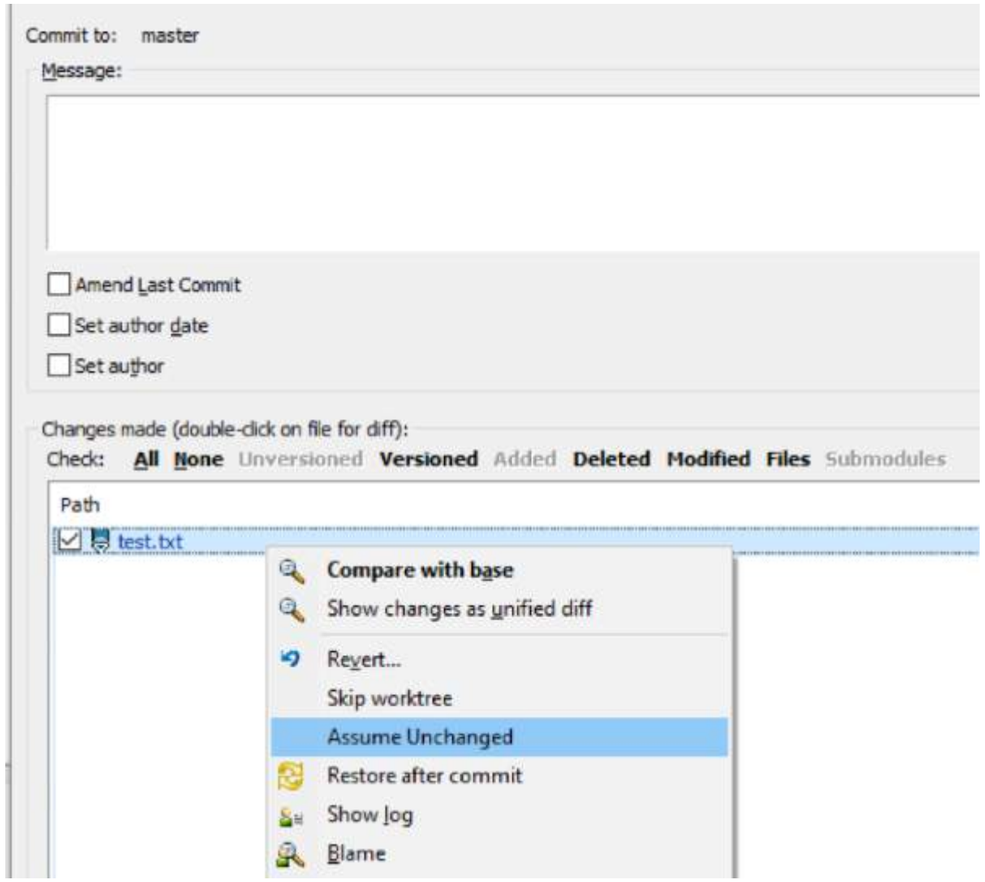
If a file is changed, but you don't like to commit it, set the file as "Assume unchanged"



撤销“假设未更改”

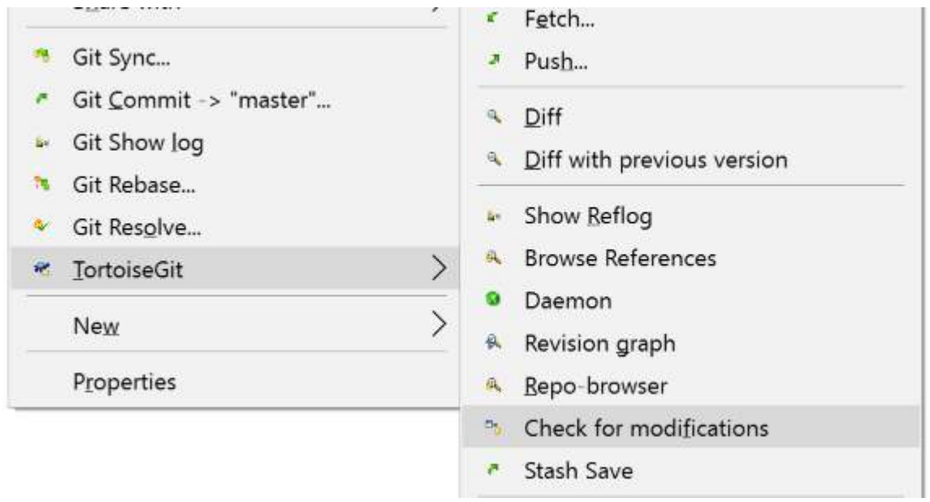
需要一些步骤：

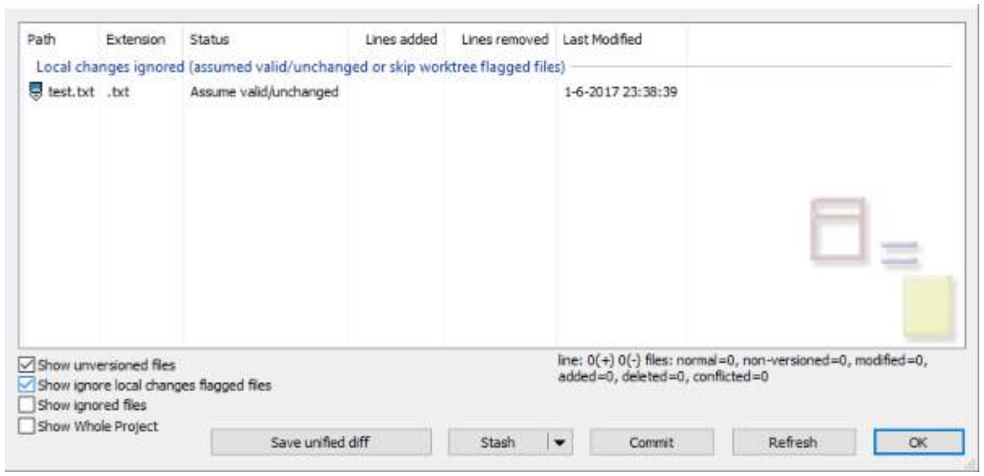


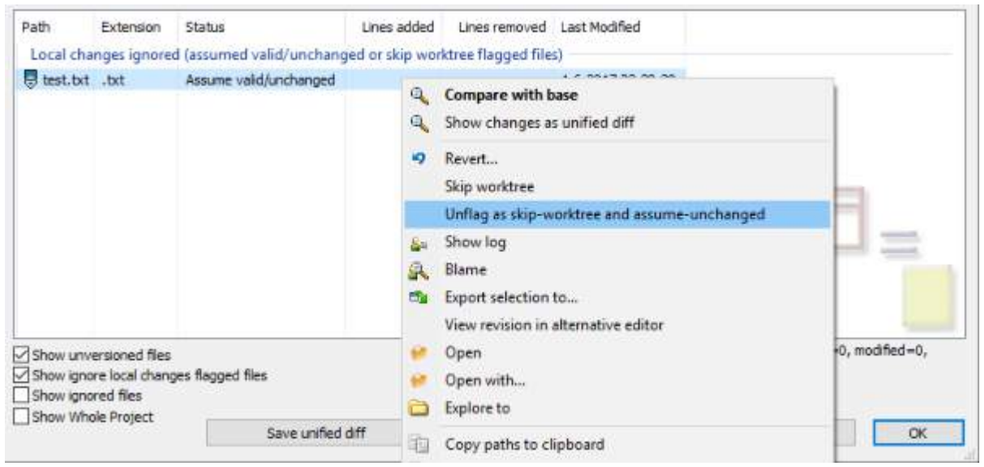


Revert "Assume unchanged"

Need some steps:

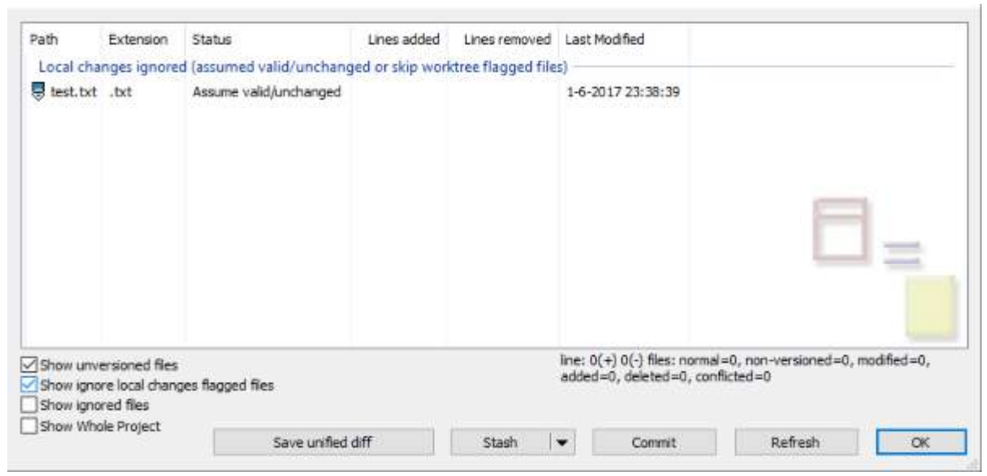


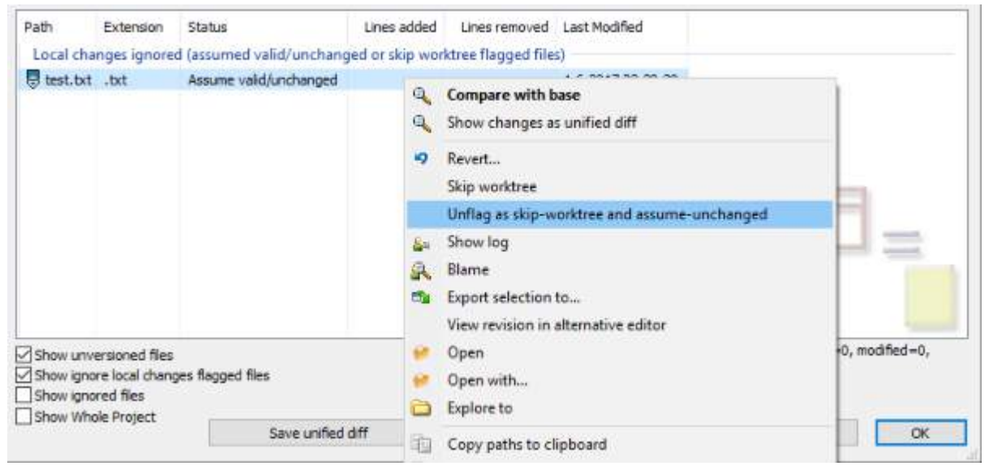




第52.3节：忽略文件和文件夹

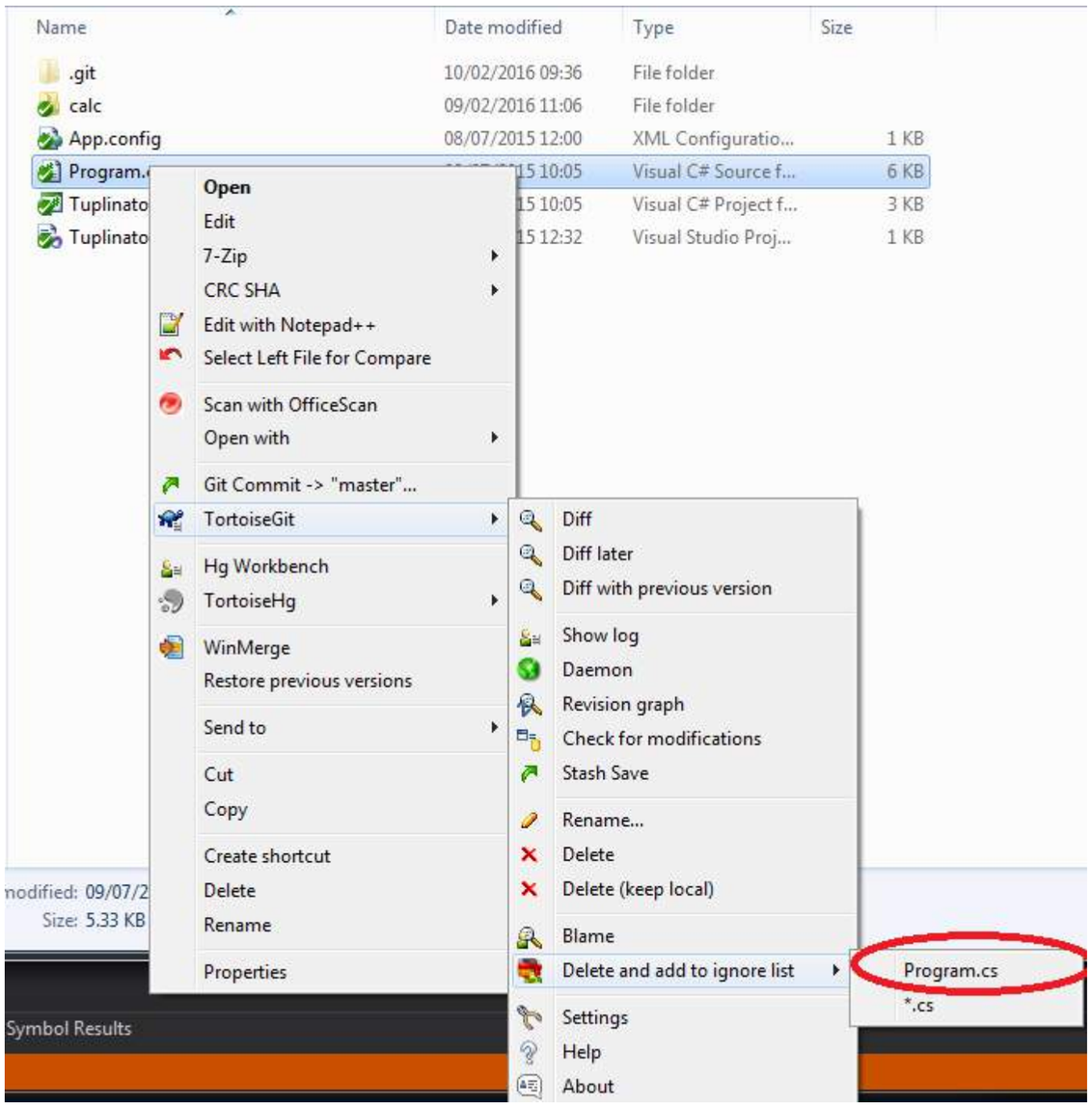
使用TortoiseGit图形界面的用户点击 右键 在你想忽略的文件（或文件夹）上 -> TortoiseGit -> 删除并添加到忽略列表，这里你可以选择忽略该类型的所有文件或仅忽略这个特定文件 -> 会弹出对话框 点击确定，操作完成。





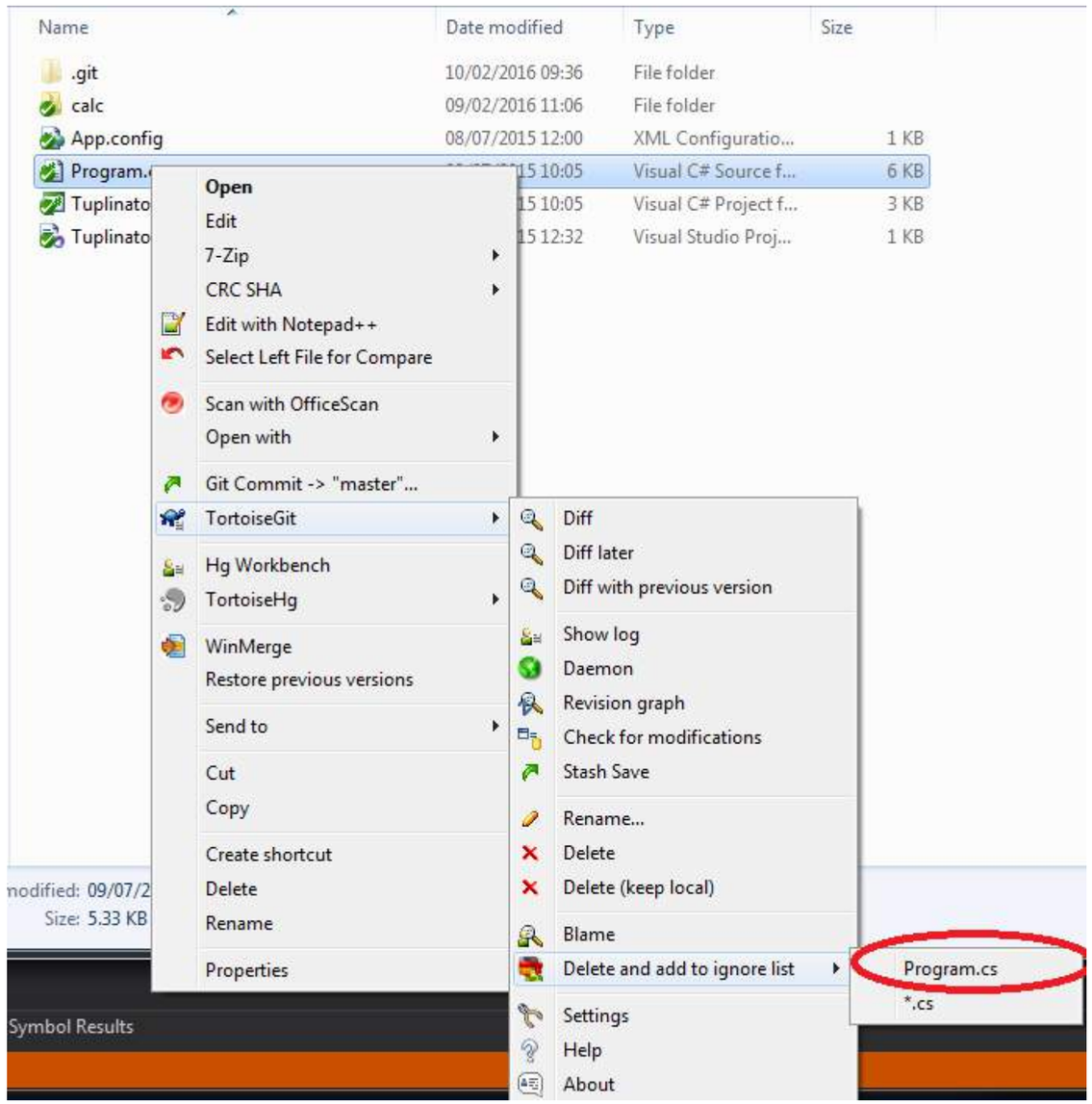
Section 52.3: Ignoring Files and Folders

Those that are using TortioseGit UI click `Right Mouse` on the file (or folder) you want to ignore -> `TortoiseGit -> Delete and add to ignore list`, here you can choose to ignore all files of that type or this specific file -> dialog will pop out Click `Ok` and you should be done.



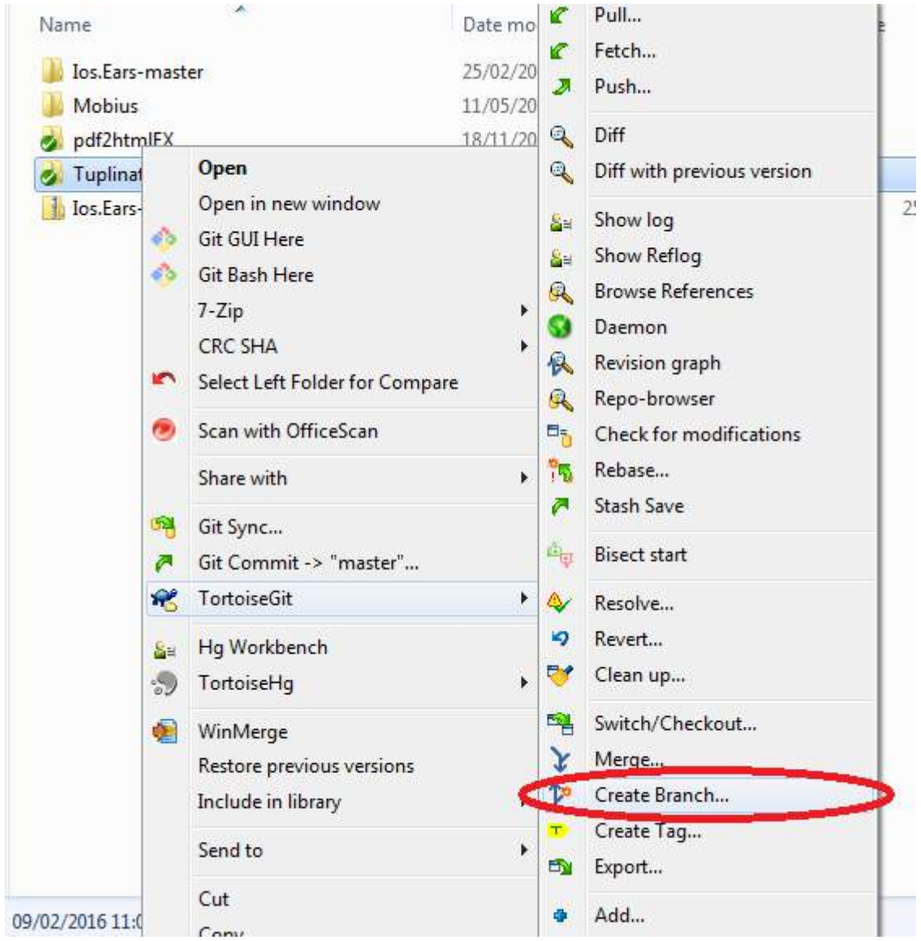
第52.4节：分支

使用图形界面进行分支的用户点击 右键 在仓库上点击，然后选择TortoiseGit -> 创建分支...

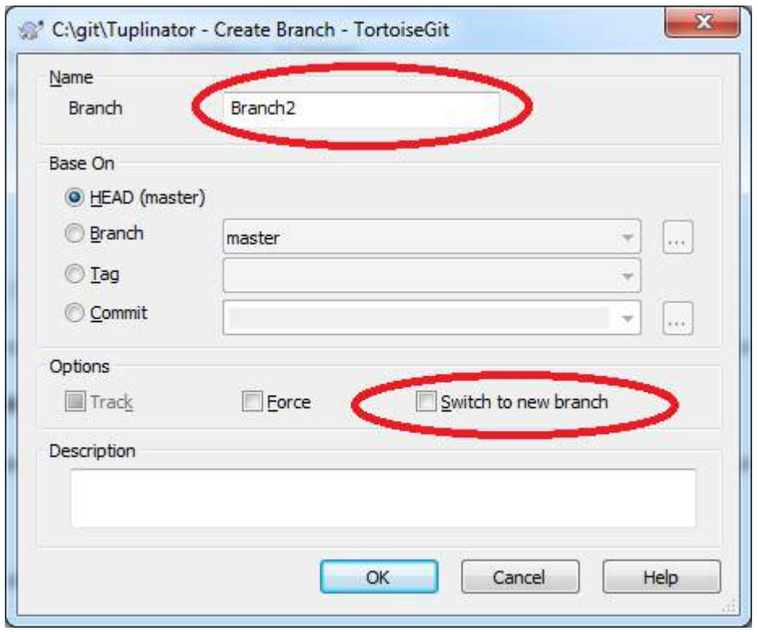


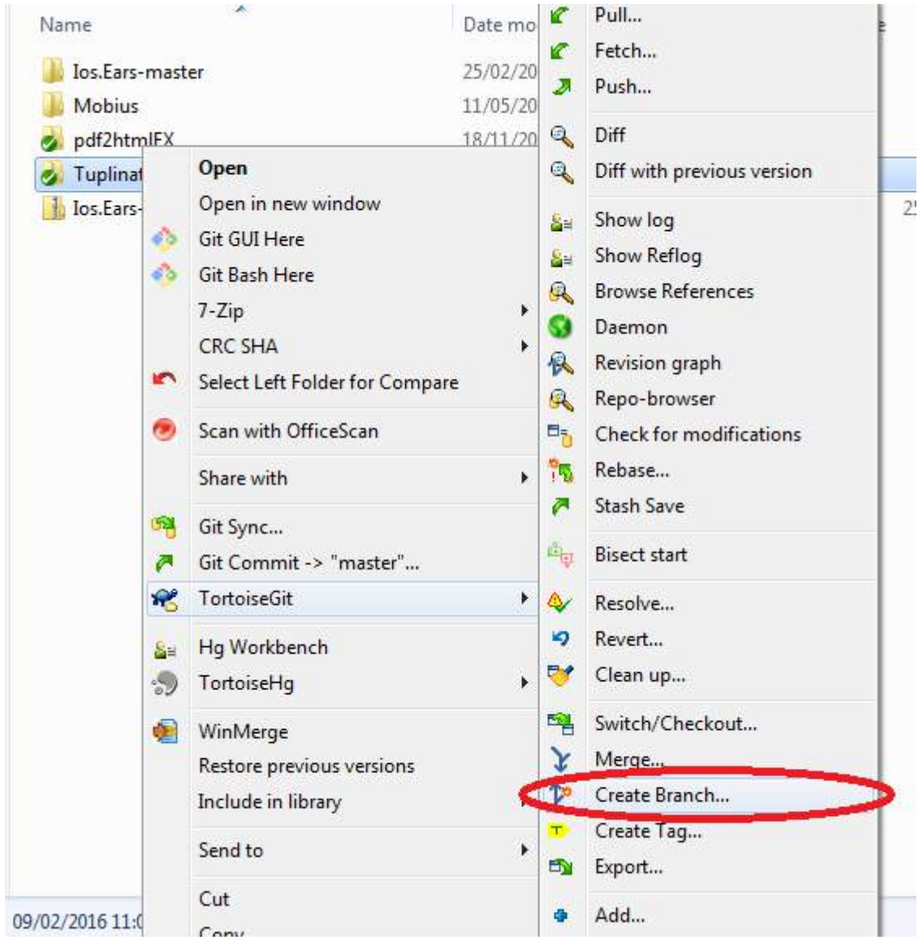
Section 52.4: Branching

For those that are using UI to branch click `Right Mouse` on repository then `Tortoise Git` -> `Create Branch...`

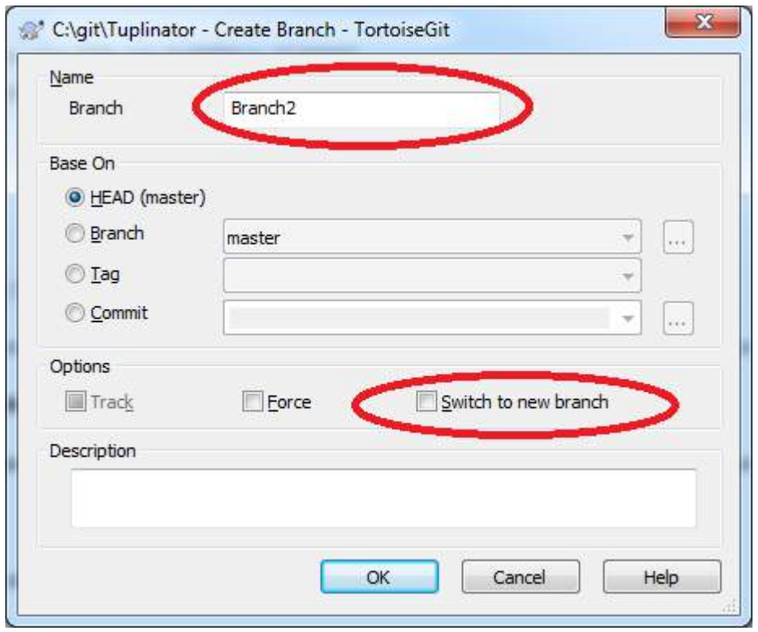


会打开新窗口 -> 为分支命名 -> 勾选切换到新分支（通常你会在分支后开始工作）。-> 点击确定，操作完成。





New window will open -> `Give branch a name` -> Tick the box `Switch to new branch` (Chances are you want to start working with it after branching). -> Click `OK` and you should be done.



第53章：外部合并和差异工具

第53.1节：将KDi3设置为合并工具

以下内容应添加到您的全局.gitconfig文件中

```
[merge]
    tool = kdiff3
[mergetool "kdiff3"]
path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    keepBackup = false
keepbackup = false
    trustExitCode = false
```

请记得将path属性设置为指向您安装KDiff3的目录

第53.2节：设置KDiff3作为差异工具

```
[diff]
tool = kdiff3
    guitool = kdiff3
[difftool "kdiff3"]
path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

第53.3节：设置IntelliJ IDE作为合并工具（Windows）

```
[merge]
    tool = intellij
[mergetool "intellij"]
cmd = cmd \"/C D:\\workspace\ools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE") $(cd
$(dirname "$BASE") && pwd)/$(basename "$BASE") $(cd $(dirname "$MERGED") && pwd)/$(basename
"$MERGED")\"
    keepBackup = false
    keepbackup = false
    trustExitCode = true
```

这里唯一需要注意的是，cmd 属性不接受路径中包含任何奇怪字符。如果你的IDE安装位置有奇怪字符（例如安装在Program Files (x86)中），你需要创建一个符号链接

第53.4节：设置IntelliJ IDE作为差异工具（Windows）

```
[diff]
tool = intellij
    guitool = intellij
[difftool "intellij"]
path = D:/程序文件(x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
    cmd = cmd \"/C D:\\workspace\ools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname "$LOCAL")
&& pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")\"
```

这里唯一需要注意的是，cmd 属性不接受路径中包含任何奇怪字符。如果你的IDE安装位置有奇怪字符（例如安装在Program Files (x86)中），你需要创建一个符号链接

Chapter 53: External merge and difftools

Section 53.1: Setting up KDiff3 as merge tool

The following should be added to your global `.gitconfig` file

```
[merge]
    tool = kdiff3
[mergetool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    keepBackup = false
    keepbackup = false
    trustExitCode = false
```

Remember to set the `path` property to point to the directory where you have installed KDiff3

Section 53.2: Setting up KDiff3 as diff tool

```
[diff]
    tool = kdiff3
    guitool = kdiff3
[difftool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

Section 53.3: Setting up an IntelliJ IDE as merge tool (Windows)

```
[merge]
    tool = intellij
[mergetool "intellij"]
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE") $(cd
$(dirname "$BASE") && pwd)/$(basename "$BASE") $(cd $(dirname "$MERGED") && pwd)/$(basename
"$MERGED")\"
    keepBackup = false
    keepbackup = false
    trustExitCode = true
```

The one gotcha here is that this `cmd` property does not accept any weird characters in the path. If your IDE's install location has weird characters in it (e.g. it's installed in `Program Files (x86)`, you'll have to create a symlink

Section 53.4: Setting up an IntelliJ IDE as diff tool (Windows)

```
[diff]
    tool = intellij
    guitool = intellij
[difftool "intellij"]
    path = D:/Program Files (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname "$LOCAL")
&& pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")\"
```

The one gotcha here is that this `cmd` property does not accept any weird characters in the path. If your IDE's install location has weird characters in it (e.g. it's installed in `Program Files (x86)`, you'll have to create a symlink

第53.5节：设置Beyond Compare

您可以设置bcomp.exe的路径

```
git config --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'
```

并将bc3配置为默认

```
git config --global diff.tool bc3
```


Section 53.5: Setting up Beyond Compare

You can set the path to `bcomp.exe`

```
git config --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'
```

and configure `bc3` as default

```
git config --global diff.tool bc3
```

第54章：更新引用中的对象名称

第54.1节：更新引用中的对象名称

使用

更新存储在引用中的对象名称

概要

```
git update-ref [-m <原因>] (-d <引用> [<旧值>] | [--no-deref] [--create-reflog] <引用>
<新值> [<旧值>] | --stdin [-z])
```

通用语法

1. 解引用符号引用，将当前分支头更新为新对象。

```
git update-ref HEAD <新值>
```

2. 在验证引用的当前值匹配旧值后，将新值存储到引用中。

```
git update-ref refs/head/master <新值> <旧值>
```

上述语法仅当master分支头的当前值为旧值时，将其更新为新值。

使用-d标志在验证<引用>仍包含<旧值>后删除该引用。

使用--create-reflog，update-ref将为每个引用创建reflog，即使通常不会创建。

使用-z标志以NUL结尾的格式指定，其值包括update、create、delete、verify。

更新

在验证了<oldvalue>（如果提供）后，将<ref>设置为<newvalue>。指定零值<newvalue>以确保更新后该引用不存在，和/或指定零值<oldvalue>以确保更新前该引用不存在。

创建

在确认<ref>不存在后，使用<newvalue>创建。所给的<newvalue>不得为零。

删除

在确认<ref>存在且值为<oldvalue>（如果提供）后删除。如果提供，<oldvalue>不得为零。

验证

验证<ref>是否为<oldvalue>，但不进行更改。如果<oldvalue>为零或缺失，则该引用必须不存在。

Chapter 54: Update Object Name in Reference

Section 54.1: Update Object Name in Reference

Use

Update the object name which is stored in reference

SYNOPSIS

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref>
<newvalue> [<oldvalue>] | --stdin [-z])
```

General Syntax

1. Dereferencing the symbolic refs, update the current branch head to the new object.

```
git update-ref HEAD <newvalue>
```

2. Stores the `newvalue` in `ref`, after verify that the current value of the `ref` matches `oldvalue`.

```
git update-ref refs/head/master <newvalue> <oldvalue>
```

above syntax updates the master branch head to `newvalue` only if its current value is `oldvalue`.

Use -d flag to deletes the named **<ref>** after verifying it still contains **<oldvalue>**.

Use `--create-reflog`, update-ref will create a reflog for each ref even if one would not ordinarily be created.

Use -z flag to specify in NUL-terminated format, which has values like update, create, delete, verify.

Update

Set **<ref>** to **<newvalue>** after verifying **<oldvalue>**, if given. Specify a zero **<newvalue>** to ensure the ref does not exist after the update and/or a zero **<oldvalue>** to make sure the ref does not exist before the update.

Create

Create **<ref>** with **<newvalue>** after verifying it does not exist. The given **<newvalue>** may not be zero.

Delete

Delete **<ref>** after verifying it exists with **<oldvalue>**, if given. If given, **<oldvalue>** may not be zero.

Verify

Verify **<ref>** against **<oldvalue>** but do not change it. If **<oldvalue>** zero or missing, the ref must not exist.

第55章：Bash Ubuntu上的Git分支名称

本文件介绍了在bash终端中git的分支名称。我们开发者经常需要查找git分支名称。我们可以将分支名称与当前目录路径一起添加。

第55.1节：终端中的分支名称

什么是PS1

PS1表示提示符字符串1。它是Linux/UNIX shell中可用的提示符之一。当你打开终端时，bash提示符会显示PS1变量中定义的内容。为了在bash提示符中添加分支名称，我们需要编辑PS1变量（在~/.bash_profile中设置PS1的值）。

显示git分支名称

将以下内容添加到你的~/.bash_profile中

```
git_branch() {
 git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git_branch)\[\033[00m\] $ "
```

这个git_branch函数会找到我们当前所在的分支名称。完成这些更改后，我们可以在终端中进入git仓库，并能看到分支名称。

Chapter 55: Git Branch Name on Bash Ubuntu

This documentation deals with the **branch name** of the git on the **bash** terminal. We developers need to find the git branch name very frequently. We can add the branch name along with the path to the current directory.

Section 55.1: Branch Name in terminal

What is PS1

PS1 denotes Prompt String 1. It is the one of the prompt available in Linux/UNIX shell. When you open your terminal, it will display the content defined in PS1 variable in your bash prompt. In order to add branch name to bash prompt we have to edit the PS1 variable (set value of PS1 in ~/.bash_profile).

Display git branch name

Add following lines to your ~/.bash_profile

```
git_branch() {
 git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git_branch)\[\033[00m\] $ "
```

This git_branch function will find the branch name we are on. Once we are done with this changes we can navigate to the git repo on the terminal and will be able to see the branch name.

第56章：Git客户端钩子

像许多其他版本控制系统一样，Git 也有一种在某些重要操作发生时触发自定义脚本的方法。这些钩子分为两类：客户端钩子和服务器端钩子。客户端钩子由提交和合并等操作触发，而服务器端钩子则在接收推送的提交等网络操作时运行。你可以出于各种原因使用这些钩子。

第56.1节：Git 预推送钩子

pre-push脚本由git push调用，在检查远程状态后，但在任何内容被推送之前。如果该脚本以非零状态退出，则不会推送任何内容。

该钩子调用时带有以下参数：

```
$1 -- 正在推送的远程名称（例如：origin）
$2 -- 正在推送的URL（例如：https://://.git）
```

有关正在推送的提交的信息以以下格式作为行传递到标准输入：

```
<local_ref> <local_sha1> <remote_ref> <remote_sha1>
```

示例值：

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

以下示例 pre-push 脚本取自默认的 pre-push.sample，该文件是在使用git init 初始化新仓库时自动创建的

```
# 此示例展示了如何阻止推送提交，其日志信息以 "WIP"（进行中的工作）开头。


remote="$1"
url="$2"

z40000000000000000000000000000000000000000

while read local_ref local_sha remote_ref remote_sha
do
    if [ "$local_sha" = $z40 ]
    then
        # 处理删除操作
        :
    else
        if [ "$remote_sha" = $z40 ]
        then
            # 新分支，检查所有提交
            range="$local_sha"
        else
            # 更新现有分支，检查新的提交
            range="$remote_sha..$local_sha"
        fi
```


Chapter 56: Git Client-Side Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

Section 56.1: Git pre-push hook

pre-push script is called by `git push` after it has checked the remote status, but before anything has been pushed. If this script exits with a non-zero status nothing will be pushed.

This hook is called with the following parameters:

```
 $1 -- Name of the remote to which the push is being done (Ex: origin)
$2 -- URL to which the push is being done (Ex: https://://.git)
```

Information about the commits which are being pushed is supplied as lines to the standard input in the form:

```
<local_ref> <local_sha1> <remote_ref> <remote_sha1>
```

Sample values:

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

Below example pre-push script was taken from default pre-push.sample which was automatically created when a new repository is initialized with `git init`

```
# This sample shows how to prevent push of commits where the log message starts
# with "WIP" (work in progress).

remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000

while read local_ref local_sha remote_ref remote_sha
do
    if [ "$local_sha" = $z40 ]
    then
        # Handle delete
        :
    else
        if [ "$remote_sha" = $z40 ]
        then
            # New branch, examine all commits
            range="$local_sha"
        else
            # Update to existing branch, examine new commits
            range="$remote_sha..$local_sha"
        fi
```

```
        # 检查是否有WIP提交
        commit=`git rev-list -n 1 --grep '^WIP' "$range"`
        if [ -n "$commit" ]
        then
            echo >&2 "在 $local_ref 中发现WIP提交，未推送"
            exit 1
        fi
    fi
完成

exit 0
```

第56.2节：安装钩子

钩子全部存储在Git目录的 hooks 子目录中。在大多数项目中，该目录是 .git/hooks。

要启用钩子脚本，请将一个文件放入 .git 目录的 hooks 子目录中，文件名应适当命名（无扩展名）且具有可执行权限。


```
        # Check for WIP commit
        commit=`git rev-list -n 1 --grep '^WIP' "$range"`
        if [ -n "$commit" ]
        then
            echo >&2 "Found WIP commit in $local_ref, not pushing"
            exit 1
        fi
    fi
done

exit 0
```

Section 56.2: Installing a Hook

The hooks are all stored in the `hooks` sub directory of the Git directory. In most projects, that's `.git/hooks`.

To enable a hook script, put a file in the `hooks` subdirectory of your `.git` directory that is named appropriately (without any extension) and is executable.

第57章：Git rerere

rerere（重用已记录的解决方案）允许你告诉git记住你如何解决冲突块。这使得当git下次遇到相同冲突时，能够自动解决。

第57.1节：启用rerere

要启用 rerere，请运行以下命令：

```
$ git config --global rerere.enabled true
```

这可以在特定仓库中执行，也可以全局执行。

Chapter 57: Git rerere

`rerere` (reuse recorded resolution) allows you to tell git to remember how you resolved a hunk conflict. This allows it to be automatically resolved the next time that git encounters the same conflict.

Section 57.1: Enabling rerere

To enable `rerere` run the following command:

```
$ git config --global rerere.enabled true
```

This can be done in a specific repository as well as globally.

第58章：更改git仓库名称

如果你在远程端（例如github或bitbucket）更改了仓库名称，当你推送现有代码时，会看到错误：致命错误，未找到仓库**。

第58.1节：更改本地设置

打开终端，

```
cd projectFolder
git remote -v （它会显示之前的git地址）
git remote set-url origin https://username@bitbucket.org/username/newName.git
git remote -v （再次确认，它会显示新的git地址）
git push （随意操作。）
```


Chapter 58: Change git repository name

If you change repository name on the remote side, such as your github or bitbucket, when you push your exisiting code, you will see error: Fatal error, repository not found**.

Section 58.1: Change local setting

Go to terminal,

```
cd projectFolder
git remote -v (it will show previous git url)
git remote set-url origin https://username@bitbucket.org/username/newName.git
git remote -v (double check, it will show new git url)
git push (do whatever you want.)
```

第59章：Git标签

像大多数版本控制系统（VCS）一样，Git 具有将历史中特定点标记为重要的能力。
通常人们使用此功能来标记发布点（v1.0 等）。

第59.1节：列出所有可用标签

使用命令 git tag 可以列出所有可用的标签：

```
$ git tag
<输出如下>
v0.1
v1.3
```

> 注意：标签 是按字母顺序 输出的。

也可以搜索可用的标签：

```
$ git tag -l "v1.8.5*"
<输出如下>
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

第59.2节：在GIT中创建并推送标签

创建标签：

- 在当前分支上创建标签：

  ```
  git tag < 标签名 >
  ```

  这将在你所在分支的当前状态下创建一个本地标签。

- 使用某个提交创建标签：

  ```
  git tag 标签名 提交标识符
  ```

  这将在你所在分支的提交标识符处创建一个本地标签。

在GIT中推送提交：

- 推送单个标签：

Chapter 59: Git Tagging

Like most Version Control Systems (VCSs), `Git` has the ability to `tag` specific points in history as being important. Typically people use this functionality to mark release points (`v1.0`, and so on).

Section 59.1: Listing all available tags

Using the command **`git tag`** lists out all available tags:

```
$ git tag
<output follows>
v0.1
v1.3
```

> **Note**: the `tags` are output in an **alphabetical** order.

One may also `search` for available `tags`:

```
$ git tag -l "v1.8.5*"
<output follows>
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Section 59.2: Create and push tag(s) in GIT

Create a tag:

- To create a tag on your current branch:

  ```
  git tag < tagname >
  ```

  This will create a local `tag` with the current state of the branch you are on.

- To create a tag with some commit:

  ```
  git tag tag-name commit-identifier
  ```

  This will create a local `tag` with the commit-identifier of the branch you are on.

Push a commit in GIT:

- Push an individual tag:

```
git push origin tag-name
```

- 一次推送所有标签

```
git push origin --tags
```



```
git push origin tag-name
```

- Push all the tags at once

```
git push origin --tags
```

第60章：整理你的本地和远程仓库

第60.1节：删除远程已删除的本地分支

用于本地与已删除远程分支之间的远程跟踪

```
git fetch -p
```

然后你可以使用

```
git branch -vv
```

查看哪些分支不再被跟踪。

不再被跟踪的分支将以下列形式显示，包含“gone”

```
分支                12345e6 [origin/branch: gone] 修复了错误
```

然后你可以结合上述命令，查找“git branch -vv”返回“gone”的分支，
再使用“-d”删除这些分支

```
git fetch -p && git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d
```


Chapter 60: Tidying up your local and remote repository

Section 60.1: Delete local branches that have been deleted on the remote

To remote tracking between local and deleted remote branches use

```
git fetch -p
```

you can then use

```
git branch -vv
```

to see which branches are no longer being tracked.

Branches that are no longer being tracked will be in the form below, containing 'gone'

```
 branch              12345e6 [origin/branch: gone] Fixed bug
```

you can then use a combination of the above commands, looking for where 'git branch -vv' returns 'gone' then using '-d' to delete the branches

```
git fetch -p && git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d
```

第61章：diff-tree

比较通过两个树对象找到的blob的内容和模式。

第61.1节：查看特定提交中更改的文件

git diff-tree --no-commit-id --name-only -r COMMIT_ID

第61.2节：用法

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-diff-options>] <tree-ish> [<tree-ish>] [<path>...]
```

选项	说明
-r	递归比较差异
--root	将初始提交作为与 /dev/null 的差异包含在内

第61.3节：常用的diff选项

选项	说明
-z	以diff-raw格式输出，行以NUL结尾。
-p	输出补丁格式。
-u	-p的同义词。
--patch-with-raw	输出补丁和 diff-raw 格式。
--stat	显示 diffstat 而非补丁。
--numstat	显示数字 diffstat 而非补丁。
--patch-with-stat	输出补丁并在前面添加其 diffstat。
--name-only	仅显示更改文件的名称。
--name-status	显示更改文件的名称和状态。
--full-index	在索引行上显示完整的对象名称。
--abbrev=<n>	在 diff-tree 头部和 diff-raw 中缩写对象名称。
-R	交换输入文件对。
-B	检测完整重写。
-M	检测重命名。
-C	检测复制。
--find-copies-harder	尝试将未更改的文件作为复制检测的候选。
-l<n>	限制重命名尝试的路径数量。
-O	根据重新排序差异。
-S	查找仅一侧包含该字符串的文件对。
--pickaxe-all	当使用 -S 并找到匹配时，显示所有文件的差异。
-a --text	将所有文件视为文本处理。

Chapter 61: diff-tree

Compares the content and mode of blobs found via two tree objects.

Section 61.1: See the files changed in a specific commit

git diff-tree --no-commit-id --name-only -r COMMIT_ID

Section 61.2: Usage

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-diff-options>] <tree-ish> [<tree-ish>] [<path>...]
```

Option	Explanation
-r	diff recursively
--root	include the initial commit as diff against /dev/null

Section 61.3: Common diff options

Option	Explanation
-z	output diff-raw with lines terminated with NUL.
-p	output patch format.
-u	synonym for -p.
--patch-with-raw	output both a patch and the diff-raw format.
--stat	show diffstat instead of patch.
--numstat	show numeric diffstat instead of patch.
--patch-with-stat	output a patch and prepend its diffstat.
--name-only	show only names of changed files.
--name-status	show names and status of changed files.
--full-index	show full object name on index lines.
--abbrev=<n>	abbreviate object names in diff-tree header and diff-raw.
-R	swap input file pairs.
-B	detect complete rewrites.
-M	detect renames.
-C	detect copies.
--find-copies-harder	try unchanged files as candidate for copy detection.
-l<n>	limit rename attempts up to paths.
-O	reorder diffs according to the .
-S	find filepair whose only one side contains the string.
--pickaxe-all	show all files diff when -S is used and hit is found.
-a --text	treat all files as text.

致谢

非常感谢所有来自 Stack Overflow Documentation 的人员提供此内容，
更多更改可发送至 web@petercv.com 以发布或更新新内容。

亚伦·克里奇利	第6章
亚伦·斯科姆拉	第49章
aavrug	第26章
阿卜杜拉	第2章
阿比杰特·卡苏尔德	第6章
adarsh	第16章
阿迪·莱斯特	第6章
AER	第12章、第25章和第29章
AesSedai101	第4章、第11章和第53章
艾哈迈德·梅特瓦利	第2章
Ajedi32	第11章
阿拉·埃丁·杰巴利	第1章
艾伦	第10章
亚历克斯·斯塔基	第46章
亚历山大·伯德	第12章
艾伦·伯尔森	第1章
铝	第50章
安贝斯	第45章
阿米泰·斯特恩	第1章和第10章
安德拉斯	第6章和第12章
安迪狗	第16章
安迪普拉	第44章
安德里亚·罗马尼奥利	第25章
安德鲁·斯克利亚列夫斯基	第10章
安迪·海登	第1、4、7、10和25章
AnimiVulpis	第1、5和14章
AnoE	第24章
安东尼·斯汤顿	第11章
APerson	第10和13章
apidae	第6章
Aratz	第2章
亚萨夫	第4章和第26章
阿塞纳尔	第11章和第13章
阿特斯·戈拉尔	第32章
阿图尔·坎杜里	第17章和第59章
巴德	第14章
班迪	第10章和第16章
本	第5章
笨拙的哲学家	第25章
鲍勃·塔克曼	第14章
博金	第1章、第7章、第31章和第36章
博žo 斯托伊科维ć	第5章
bpoiss	第5章
布赖姆	第5章
brentonstrine	第7章和第8章
Brett	第2章
Brian	第1章和第7章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Aaron Critchley	Chapter 6
Aaron Skomra	Chapter 49
aavrug	Chapter 26
Abdullah	Chapter 2
Abhijeet Kasurde	Chapter 6
adarsh	Chapter 16
Adi Lester	Chapter 6
AER	Chapters 12, 25 and 29
AesSedai101	Chapters 4, 11 and 53
Ahmed Metwally	Chapter 2
Ajedi32	Chapter 11
Ala Eddine JEBALI	Chapter 1
Alan	Chapter 10
Alex Stuckey	Chapter 46
Alexander Bird	Chapter 12
Allan Burleson	Chapter 1
Alu	Chapter 50
ambes	Chapter 45
Amitay Stern	Chapters 1 and 10
anderas	Chapters 6 and 12
AndiDog	Chapter 16
andipla	Chapter 44
Andrea Romagnoli	Chapter 25
Andrew Sklyarevsky	Chapter 10
Andy Hayden	Chapters 1, 4, 7, 10 and 25
AnimiVulpis	Chapters 1, 5 and 14
AnoE	Chapter 24
Anthony Staunton	Chapter 11
APerson	Chapters 10 and 13
apidae	Chapter 6
Aratz	Chapter 2
Asaph	Chapters 4 and 26
Asenar	Chapters 11 and 13
Ates Goral	Chapter 32
Atul Khanduri	Chapters 17 and 59
Bad	Chapter 14
bandi	Chapters 10 and 16
Ben	Chapter 5
Blundering Philosopher	Chapter 25
BobTuckerman	Chapter 14
Boggin	Chapters 1, 7, 31 and 36
Božo Stojković	Chapter 5
bpoiss	Chapter 5
Braiam	Chapter 5
brentonstrine	Chapters 7 and 8
Brett	Chapter 2
Brian	Chapters 1 and 7

Brian Hinchey	第26章
bstpierre	第11章
bud	第17章、第26章和第28章
Cache Staheli	第5章、第10章和第13章
卡勒布·布林克曼	第3章和第16章
查理·伊根	第6章
黄勤	第20章
克里斯蒂安·马克斯	第24章
科迪·古尔德纳	第10章和第29章
科林·M	第5章
ComicSansMS	第9章
配置	第4、22、24、36和44章
cormacrelf	第10章
克雷格·布雷特	第1章
创意约翰	第18章
尴尬	第29章
丹尼尔·基什	第45章
dahlbyk	第20章
dan	第14章
丹·赫尔姆	第1章
丹尼尔·凯弗	第12、14和50章
丹尼尔·斯特拉多夫斯基	第14章
达特茅斯	第5、25、34、43、45、47和48章
大卫·本·诺布尔	第38章
davidcondrey	第2章和第10章
深度	第10章和第26章
迪帕克·班萨尔	第14章
德维什·赛尼	第5章
迪拉杰·瓦茨	第5章
迪米特里奥斯·米斯特里奥蒂斯	第22章
董昶	第49章
杜贝克	第17章
邓肯·X·辛普森	第14章
e.doroskevic	第12章和第13章
埃德·科特雷尔	第5章
幻影	第24章
伊丽莎白	第45章
enrico.bacis	第5章
ericdwang	第10章
eush77	第6章、第11章和第16章
eykanal	第1章
以斯拉·弗里（Ezra Free）	第25章
法比奥	第2章
法哈德·法吉希（Farhad Faghihi）	第48章
FeedTheWeb	第48章
流程	第2章和第24章
forevergenin	第3章、第14章和第34章
forresthopkinsa	第22章
fracz	第5章、第14章和第26章
弗雷德·巴克莱	第1章、第10章和第14章
frlan	第29章
Functino	第5章
fybw id	第49章和第61章

Brian Hinchey	Chapter 26
bstpierre	Chapter 11
bud	Chapters 17, 26 and 28
Cache Staheli	Chapters 5, 10 and 13
Caleb Brinkman	Chapters 3 and 16
Charlie Egan	Chapter 6
Chin Huang	Chapter 20
Christiaan Maks	Chapter 24
Cody Guldner	Chapters 10 and 29
Collin M	Chapter 5
ComicSansMS	Chapter 9
Configure	Chapters 4, 22, 24, 36 and 44
cormacrelf	Chapter 10
Craig Brett	Chapter 1
Creative John	Chapter 18
cringe	Chapter 29
Dániel Kis	Chapter 45
dahlbyk	Chapter 20
dan	Chapter 14
Dan Hulme	Chapter 1
Daniel Käfer	Chapters 12, 14 and 50
Daniel Stradowski	Chapter 14
Dartmouth	Chapters 5, 25, 34, 43, 45, 47 and 48
David Ben Knoble	Chapter 38
davidcondrey	Chapters 2 and 10
Deep	Chapters 10 and 26
Deepak Bansal	Chapter 14
Devesh Saini	Chapter 5
Dheeraj vats	Chapter 5
Dimitrios Mistriotis	Chapter 22
Dong Thang	Chapter 49
dubek	Chapter 17
Duncan X Simpson	Chapter 14
e.doroskevic	Chapters 12 and 13
Ed Cottrell	Chapter 5
Eidolon	Chapter 24
Elizabeth	Chapter 45
enrico.bacis	Chapter 5
ericdwang	Chapter 10
eush77	Chapters 6, 11 and 16
eykanal	Chapter 1
Ezra Free	Chapter 25
Fabio	Chapter 2
Farhad Faghihi	Chapter 48
FeedTheWeb	Chapter 48
Flows	Chapters 2 and 24
forevergenin	Chapters 3, 14 and 34
forresthopkinsa	Chapter 22
fracz	Chapters 5, 14 and 26
Fred Barclay	Chapters 1, 10 and 14
frlan	Chapter 29
Functino	Chapter 5
fybw id	Chapters 49 and 61

ganesshkumar	第25章
gavv	第14章和第35章
乔治·布莱顿	第10章
georgebrock	第16章
GingerPlusPlus	第26章
格伦·史密斯	第35章
gnis	第19章
格雷格·布雷	第50章
纪尧姆	第29章
纪尧姆·帕斯卡尔	第5章和第36章
古勒里亚	第2章
哈杜穆斯	第22章
海托尔特塞尔让	第3章
恩里克·巴塞洛斯	第1章
霍伦	第22章
雨果·巴夫	第48章
雨果·费雷拉	第12章
伊戈尔·伊万查	第10章
英德雷加德	第36章
intboolstring	第2、4、5、6、9、10、12、17和29章
伊萨克·康布林克	第57章
J F	第9章
杰克·瑞安	第6章
杰克D	第6章
雅库布·纳尔特布斯基	第4、5、6、26和43章
詹姆斯·拉奇	第14章
詹姆斯·泰勒	第10章
雅诺什	第1、2和10章
贾里德	第26章
杰森	第14章
贾夫 洛克	第1、45和49章
杰夫·帕克特	第27章
jeffdill2	第1、3、6和26章
延斯	第5章
jkdev	第4章
joaquinlpereyra	第5章
乔尔·科内特	第14章
joeytwiddle	第10章
JonasCz	第5章
乔纳森	第14章
乔纳森·拉姆	第1章
乔丹·诺特	第10章
约瑟夫·达森布罗克	第1章和第14章
约瑟夫·K·斯特劳斯	第6章和第12章
joshng	第5章
jpkrohling	第16章
jready	第2章
jtbandes	第11章和第12章
朱利安	第13章和第52章
朱莉·大卫	第3章、第18章和第26章
jwd630	第39章
Kačer	第5章
Kageetai	第1章

ganesshkumar	Chapter 25
gavv	Chapters 14 and 35
George Brighton	Chapter 10
georgebrock	Chapter 16
GingerPlusPlus	Chapter 26
Glenn Smith	Chapter 35
gnis	Chapter 19
Greg Bray	Chapter 50
Guillaume	Chapter 29
Guillaume Pascal	Chapters 5 and 36
guleria	Chapter 2
hardmooth	Chapter 22
heitortsergent	Chapter 3
Henrique Barcelos	Chapter 1
Horen	Chapter 22
Hugo Buff	Chapter 48
Hugo Ferreira	Chapter 12
Igor Ivancha	Chapter 10
Indregaard	Chapter 36
intboolstring	Chapters 2, 4, 5, 6, 9, 10, 12, 17 and 29
Isak Combrinck	Chapter 57
J F	Chapter 9
Jack Ryan	Chapter 6
JakeD	Chapter 6
Jakub Narębski	Chapters 4, 5, 6, 26 and 43
james large	Chapter 14
James Taylor	Chapter 10
janos	Chapters 1, 2 and 10
Jarede	Chapter 26
Jason	Chapter 14
Jav_Rock	Chapters 1, 45 and 49
Jeff Puckett	Chapter 27
jeffdill2	Chapters 1, 3, 6 and 26
Jens	Chapter 5
jkdev	Chapter 4
joaquinlpereyra	Chapter 5
Joel Cornett	Chapter 14
joeytwiddle	Chapter 10
JonasCz	Chapter 5
Jonathan	Chapter 14
Jonathan Lam	Chapter 1
Jordan Knott	Chapter 10
Joseph Dasenbrock	Chapters 1 and 14
Joseph K. Strauss	Chapters 6 and 12
joshng	Chapter 5
jpkrohling	Chapter 16
jready	Chapter 2
jtbandes	Chapters 11 and 12
Julian	Chapters 13 and 52
Julie David	Chapters 3, 18 and 26
jwd630	Chapter 39
Kačer	Chapter 5
Kageetai	Chapter 1

mpromonet	第2章、第9章、第17章和第23章
MrTux	第41章
mwarsco	第24章
mystarrocks	第3章
n0shadow	第19章
纳拉扬·阿查里亚	第5章
内森·亚瑟	第4章
内森尼尔·福特	第6章和第7章
内曼贾·博里奇	第12章
内曼贾·特里富诺维奇	第50章
nepda	第14章
Neui	第1章和第5章
nighthawk454	第30章和第42章
尼辛·K·阿尼尔	第7章
诺亚	第2章、第8章和第14章
努沙德·PP	第14章
努里·塔斯德米尔	第5章
nus	第11章和第38章
ob1	第1章
食人魔诗篇33	第6章
夹竹桃	第2章
olegtaranenko	第14章
orkoden	第6章和第40章
奥尔托马拉·洛克尼	第6章、第12章、第13章和第26章
奥扎伊尔·卡弗雷	第14章
P.J.Meisch	第28章
佩斯	第7章
PaladiN	第5章、第9章和第14章
帕特里克	第26章
pcm	第29章
佩德罗·皮涅罗	第2章和第50章
penguincoder	第6章、第8章和第11章
彼得·阿米登	第51章
彼得·米特拉诺	第12、25和26章
PhotometricStereo	第28章
pkowalczyk	第25章
pktangyue	第5章
Pod	第1章和第10章
pogosama	第29章
poke	第5章
普里扬舒·谢卡尔	第13、14、19和42章
pylang	第5章和第12章
拉加夫	第3章
拉尔夫·拉斐尔·弗里克斯	第3章、第14章、第19章和第26章
红绿代码	第17章
瑞斯·欧	第5章
里卡多·阿莫雷斯	第33章
理查德	第12章
理查德·达利	第4章
理查德·汉密尔顿	第14章
里克	第5章、第7章、第10章、第25章和第36章
riyadhalnur	第11章
罗尔德·内夫斯	第1章

mpromonet	Chapters 2, 9, 17 and 23
MrTux	Chapter 41
mwarsco	Chapter 24
mystarrocks	Chapter 3
n0shadow	Chapter 19
Narayan Acharya	Chapter 5
Nathan Arthur	Chapter 4
Nathaniel Ford	Chapters 6 and 7
Nemanja Boric	Chapter 12
Nemanja Trifunovic	Chapter 50
nepda	Chapter 14
Neui	Chapters 1 and 5
nighthawk454	Chapters 30 and 42
Nithin K Anil	Chapter 7
Noah	Chapters 2, 8 and 14
Noushad PP	Chapter 14
Nuri Tasdemir	Chapter 5
nus	Chapters 11 and 38
ob1	Chapter 1
Ogre Psalm33	Chapter 6
Oleander	Chapter 2
olegtaranenko	Chapter 14
orkoden	Chapters 6 and 40
Ortomala Lokni	Chapters 6, 12, 13 and 26
Ozair Kafray	Chapter 14
P.J.Meisch	Chapter 28
Pace	Chapter 7
PaladiN	Chapters 5, 9 and 14
Patrick	Chapter 26
pcm	Chapter 29
Pedro Pinheiro	Chapters 2 and 50
penguincoder	Chapters 6, 8 and 11
Peter Amidon	Chapter 51
Peter Mitrano	Chapters 12, 25 and 26
PhotometricStereo	Chapter 28
pkowalczyk	Chapter 25
pktangyue	Chapter 5
Pod	Chapters 1 and 10
pogosama	Chapter 29
poke	Chapter 5
Priyanshu Shekhar	Chapters 13, 14, 19 and 42
pylang	Chapters 5 and 12
Raghav	Chapter 3
Ralf Rafael Frix	Chapters 3, 14, 19 and 26
RedGreenCode	Chapter 17
RhysO	Chapter 5
Ricardo Amores	Chapter 33
Richard	Chapter 12
Richard Dally	Chapter 4
Richard Hamilton	Chapter 14
Rick	Chapters 5, 7, 10, 25 and 36
riyadhalnur	Chapter 11
Roald Nefs	Chapter 1

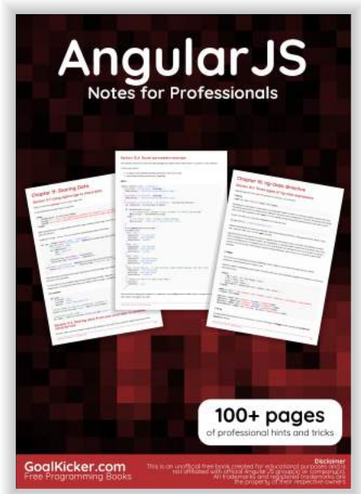
罗宾	第14章
rokonoid	第5章
ronnyfm	第1章
萨拉赫·埃丁·拉尼切	第3章
saml	第3章
萨达斯里昂	第22章
萨沙	第5章
萨沙·沃尔夫	第5章
SashaZd	第48章
萨扎德·希赛因·汗	第1章
斯科特·韦尔登	第3、5、22、23、41和51章
塞巴斯蒂安b	第5和26章
SeeuD1	第5章
shoelzer	第46章
Shog9	第23章
西蒙娜·卡莱蒂	第14章和第41章
sjas	第5章
SommerEngineering	第10章
sonali	第45章
索尼·金	第10章
spikeheap	第5章
斯托尼	第23章
strangeqargo	第18章
SurDin	第6章
Tall Sam	第16章
textshell	第7章
Thamilan	第23章
thanksd	第11章
the12	第14章
TheDarkKnight	第36章和第59章
theJollySin	第5章
托马斯·克劳利	第60章
tinlyx	第9章
托比	第5章和第20章
托比·艾伦	第2章
汤姆·吉塞林克	第5章
汤姆·黑尔	第11章
托马斯·卡尼巴诺	第26章和第29章
托马什·Bąk	第5章
特拉维斯	第12章
泰勒·齐卡	第1章
tymspy	第1章
撤销	第8、9、10和25章
乌韦	第14章
第六章	第5章
维克多·施罗德	第5、12和44章
维文·乔治	第3章
弗拉德	第14章
弗拉基米尔·F	第10章
Vogel612	第8章
VonC	第1、3、5、9、12和13章
芥末爱好者	第12章
威尔弗雷德·休斯	第5章

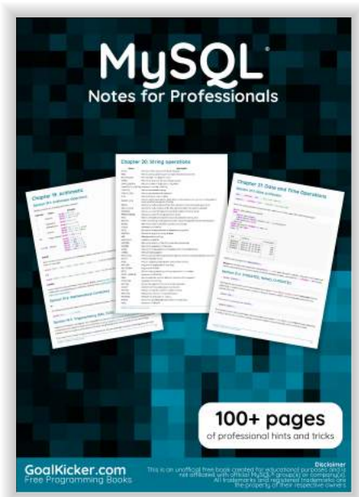
Robin	Chapter 14
rokonoid	Chapter 5
ronnyfm	Chapter 1
Salah Eddine Lahniche	Chapter 3
saml	Chapter 3
Sardathrion	Chapter 22
Sascha	Chapter 5
Sascha Wolf	Chapter 5
SashaZd	Chapter 48
Sazzad Hissain Khan	Chapter 1
Scott Weldon	Chapters 3, 5, 22, 23, 41 and 51
Sebastianb	Chapters 5 and 26
SeeuD1	Chapter 5
shoelzer	Chapter 46
Shog9	Chapter 23
Simone Carletti	Chapters 14 and 41
sjas	Chapter 5
SommerEngineering	Chapter 10
sonali	Chapter 45
Sonny Kim	Chapter 10
spikeheap	Chapter 5
Stony	Chapter 23
strangeqargo	Chapter 18
SurDin	Chapter 6
Tall Sam	Chapter 16
textshell	Chapter 7
Thamilan	Chapter 23
thanksd	Chapter 11
the12	Chapter 14
TheDarkKnight	Chapters 36 and 59
theJollySin	Chapter 5
Thomas Crowley	Chapter 60
tinlyx	Chapter 9
Toby	Chapters 5 and 20
Toby Allen	Chapter 2
Tom Gijselinck	Chapter 5
Tom Hale	Chapter 11
Tomás Cañibano	Chapters 26 and 29
Tomasz Bąk	Chapter 5
Travis	Chapter 12
Tyler Zika	Chapter 1
tymspy	Chapter 1
Undo	Chapters 8, 9, 10 and 25
Uwe	Chapter 14
Vi.	Chapter 5
Victor Schröder	Chapters 5, 12 and 44
Vivin George	Chapter 3
Vlad	Chapter 14
Vladimir F	Chapter 10
Vogel612	Chapter 8
VonC	Chapters 1, 3, 5, 9, 12 and 13
Wasabi Fan	Chapter 12
Wilfred Hughes	Chapter 5

威尔	第6章
沃伊切赫·卡齐奥尔	第11、25和26章
沃尔夫冈	第4、5、8、13和14章
WPrecht	第42章
xiaoyaoworm	第58章
ydaetskcoR	第5章
耶尔科·帕尔马	第14章
尤里·费多罗夫	第5章和第14章
Zaz	第6、7、10、18、23和37章
zebediah49	第41章
zygimantus	第14章
□ N YI IS	第18章
□	第6章和第12章

Will	Chapter 6
Wojciech Kazior	Chapters 11, 25 and 26
Wolfgang	Chapters 4, 5, 8, 13 and 14
WPrecht	Chapter 42
xiaoyaoworm	Chapter 58
ydaetskcoR	Chapter 5
Yerko Palma	Chapter 14
Yury Fedorov	Chapters 5 and 14
Zaz	Chapters 6, 7, 10, 18, 23 and 37
zebediah49	Chapter 41
zygimantus	Chapter 14
□□N□YI□□IS	Chapter 18
□□□□□	Chapters 6 and 12

你可能也喜欢





You may also like