# Git 基本操作

Git 的工作就是创建和保存你项目的快照及与之后的快照进行对比。本章将对有关创建与提交你的项目快照的命令作介绍。

# 获取与创建项目命令

# git init

用 git init 在目录中创建新的 Git 仓库。 你可以在任何时候、任何目录中这么做，完全是本地化的。

在目录中执行 git init，就可以创建一个 Git 仓库了。比如我们创建 runoob 项目：

$ mkdir runoob

$ cd runoob/

$ git init

Initialized empty Git repository in /Users/tianqixin/www/runoob/.git/

# 在 /www/runoob/.git/ 目录初始化空 Git 仓库完毕。

现在你可以看到在你的项目中生成了 .git 这个子目录。 这就是你的 Git 仓库了，所有有关你的此项目的快照数据都存放在这里。

ls -a

.    ..    .git

# git clone

使用 git clone 拷贝一个 Git 仓库到本地，让自己能够查看该项目，或者进行修改。

如果你需要与他人合作一个项目，或者想要复制一个项目，看看代码，你就可以克隆那个项目。 执行命令：

git clone [url]

[url] 为你想要复制的项目，就可以了。

例如我们克隆 Github 上的项目：

$ git clone git@github.com:schacon/simplegit.git

Cloning into 'simplegit'...

remote: Counting objects: 13, done.

remote: Total 13 (delta 0), reused 0 (delta 0), pack-reused 13

Receiving objects: 100% (13/13), done.

Resolving deltas: 100% (2/2), done.

Checking connectivity... done.

克隆完成后，在当前目录下会生成一个 simplegit 目录：

$ cd simplegit/

$ ls

README Rakefile lib

上述操作将复制该项目的全部记录。

$ ls -a

. .. .git README Rakefile lib

$ cd .git

$ ls

HEAD description info packed-refs

branches hooks logs refs

config index objects

默认情况下，Git 会按照你提供的 URL 所指示的项目的名称创建你的本地项目目录。 通常就是该 URL 最后一个 / 之后的项目名称。如果你想要一个不一样的名字， 你可以在该命令后加上你想要的名称。

# 基本快照

Git 的工作就是创建和保存你的项目的快照及与之后的快照进行对比。本章将对有关创建与提交你的项目的快照的命令作介绍。

# git add

git add 命令可将该文件添加到缓存，如我们添加以下两个文件：

$ touch README

$ touch hello.php

$ ls

README        hello.php

$ git status -s

?? README

?? hello.php

$

git status 命令用于查看项目的当前状态。

接下来我们执行 git add 命令来添加文件：

$ git add README hello.php

现在我们再执行 git status，就可以看到这两个文件已经加上去了。

$ git status -s

A README

A hello.php

$

新项目中，添加所有文件很普遍，我们可以使用 **git add .** 命令来添加当前项目的所有文件。

现在我们修改 README 文件：

$ vim README

在 README 添加以下内容：**# Runoob Git 测试**，然后保存退出。

再执行一下 git status：

$ git status -s

AM README

A hello.php

"AM" 状态的意思是，这个文件在我们将它添加到缓存之后又有改动。改动后我们再执行 **git add** 命令将其添加到缓存中：

$ git add .

$ git status -s

A README

A hello.php

当你要将你的修改包含在即将提交的快照里的时候，需要执行 git add。

# git status

git status 以查看在你上次提交之后是否有修改。

我演示该命令的时候加了 -s 参数，以获得简短的结果输出。如果没加该参数会详细输出内容：

$ git status

On branch master

Initial commit

Changes to be committed:

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

    new file: README

    new file: hello.php

# git diff

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

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

* 尚未缓存的改动：**git diff**
* 查看已缓存的改动： **git diff --cached**
* 查看已缓存的与未缓存的所有改动：**git diff HEAD**
* 显示摘要而非整个 diff：**git diff --stat**

在 hello.php 文件中输入以下内容：

<?php

echo '菜鸟教程：www.runoob.com';

?>

$ git status -s

A README

AM hello.php

$ git diff

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

index e69de29..69b5711 100644

--- a/hello.php

+++ b/hello.php

@@ -0,0 +1,3 @@

+<?php

+echo '菜鸟教程：www.runoob.com';

+?>

git status 显示你上次提交更新后的更改或者写入缓存的改动， 而 git diff 一行一行地显示这些改动具体是啥。

接下来我们来查看下 git diff --cached 的执行效果：

$ git add hello.php

$ git status -s

A README

A hello.php

$ git diff --cached

diff --git a/README b/README

new file mode 100644

index 0000000..8f87495

--- /dev/null

+++ b/README

@@ -0,0 +1 @@

+# Runoob Git 测试

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

new file mode 100644

index 0000000..69b5711

--- /dev/null

+++ b/hello.php

@@ -0,0 +1,3 @@

+<?php

+echo '菜鸟教程：www.runoob.com';

+?>

# git commit

使用 git add 命令将想要快照的内容写入缓存区， 而执行 git commit 将缓存区内容添加到仓库中。

Git 为你的每一个提交都记录你的名字与电子邮箱地址，所以第一步需要配置用户名和邮箱地址。

$ git config --global user.name 'runoob'

$ git config --global user.email test@runoob.com

接下来我们写入缓存，并提交对 hello.php 的所有改动。在首个例子中，我们使用 -m 选项以在命令行中提供提交注释。

$ git add hello.php

$ git status -s

A README

A hello.php

$ $ git commit -m '第一次版本提交'

[master (root-commit) d32cf1f] 第一次版本提交

2 files changed, 4 insertions(+)

create mode 100644 README

create mode 100644 hello.php

现在我们已经记录了快照。如果我们再执行 git status:

$ git status

# On branch master

nothing to commit (working directory clean)

以上输出说明我们在最近一次提交之后，没有做任何改动，是一个"working directory clean：干净的工作目录"。

如果你没有设置 -m 选项，Git 会尝试为你打开一个编辑器以填写提交信息。 如果 Git 在你对它的配置中找不到相关信息，默认会打开 vim。屏幕会像这样：

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: hello.php

#

~

~

".git/COMMIT\_EDITMSG" 9L, 257C

如果你觉得 git add 提交缓存的流程太过繁琐，Git 也允许你用 -a 选项跳过这一步。命令格式如下：

git commit -a

我们先修改 hello.php 文件为以下内容：

<?php

echo '菜鸟教程：www.runoob.com';

echo '菜鸟教程：www.runoob.com';

?>

再执行以下命令：

git commit -am '修改 hello.php 文件'

[master 71ee2cb] 修改 hello.php 文件

1 file changed, 1 insertion(+)

# git reset HEAD

git reset HEAD 命令用于取消已缓存的内容。

我们先改动文件 README 文件，内容如下：

# Runoob Git 测试

# 菜鸟教程

hello.php 文件修改为：

<?php

echo '菜鸟教程：www.runoob.com';

echo '菜鸟教程：www.runoob.com';

echo '菜鸟教程：www.runoob.com';

?>

现在两个文件修改后，都提交到了缓存区，我们现在要取消其中一个的缓存，操作如下：

$ git status -s

M README

M hello.php

$ git add .

$ git status -s

M README

M hello.php

$ git reset HEAD hello.php

Unstaged changes after reset:

M    hello.php

$ git status -s

M README

M hello.php

现在你执行 git commit，只会将 README 文件的改动提交，而 hello.php 是没有的。

$ git commit -m '修改'

[master f50cfda] 修改

1 file changed, 1 insertion(+)

$ git status -s

M hello.php

可以看到 hello.php 文件的修改并未提交。

这时我们可以使用以下命令将 hello.php 的修改提交：

$ git commit -am '修改 hello.php 文件'

[master 760f74d] 修改 hello.php 文件

1 file changed, 1 insertion(+)

$ git status

On branch master

nothing to commit, working directory clean

简而言之，执行 git reset HEAD 以取消之前 git add 添加，但不希望包含在下一提交快照中的缓存。

# git rm

如果只是简单地从工作目录中手工删除文件，运行 **git status** 时就会在 **Changes not staged for commit** 的提示。

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除，然后提交。可以用以下命令完成此项工作

git rm <file>

如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 **-f**

git rm -f <file>

如果把文件从暂存区域移除，但仍然希望保留在当前工作目录中，换句话说，仅是从跟踪清单中删除，使用 **--cached** 选项即可

git rm --cached <file>

如我们删除 hello.php文件：

$ git rm hello.php

rm 'hello.php'

$ ls

README

不从工作区中删除文件：

# $ git rm --cached README

rm 'README'

$ ls

README

可以递归删除，即如果后面跟的是一个目录做为参数，则会递归删除整个目录中的所有子目录和文件：

git rm –r \*

进入某个目录中，执行此语句，会删除该目录下的所有文件和子目录。

# git mv

git mv 命令用于移动或重命名一个文件、目录、软连接。

我们先把刚移除的 README 添加回来：

$ git add README

然后对其重名:

$ git mv README README.md

$ ls

README.md

# Git 分支管理

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。

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

创建分支命令：

git branch (branchname)

切换分支命令:

git checkout (branchname)

当你切换分支的时候，Git 会用该分支的最后提交的快照替换你的工作目录的内容， 所以多个分支不需要多个目录。

合并分支命令:

git merge

你可以多次合并到统一分支， 也可以选择在合并之后直接删除被并入的分支。

# Git 分支管理

# 列出分支

列出分支基本命令：

git branch

没有参数时，git branch 会列出你在本地的分支。

$ git branch

\* master

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

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

如果我们要手动创建一个分支。执行 git branch (branchname) 即可。

$ git branch testing

$ git branch

\* master

testing

现在我们可以看到，有了一个新分支 testing。

当你以此方式在上次提交更新之后创建了新分支，如果后来又有更新提交， 然后又切换到了"testing"分支，Git 将还原你的工作目录到你创建分支时候的样子

接下来我们将演示如何切换分支，我们用 git checkout (branch) 切换到我们要修改的分支。

$ ls

README

$ echo 'runoob.com' > test.txt

$ git add .

$ git commit -m 'add test.txt'

[master 048598f] add test.txt

2 files changed, 1 insertion(+), 3 deletions(-)

delete mode 100644 hello.php

create mode 100644 test.txt

$ ls

README        test.txt

$ git checkout testing

Switched to branch 'testing'

$ ls

README        hello.php

当我们切换到"testing"分支的时候，我们添加的新文件test.txt被移除了, 原来被删除的文件hello.php文件又出现了。切换回"master"分支的时候，它们有重新出现了。

$ git checkout master

Switched to branch 'master'

$ ls

README        test.txt

我们也可以使用 git checkout -b (branchname) 命令来创建新分支并立即切换到该分支下，从而在该分支中操作。

$ git checkout -b newtest

Switched to a new branch 'newtest'

$ git rm test2.txt

rm 'test2.txt'

$ ls

README        test.txt

$ git commit -am 'removed test2.txt'

[newtest 556f0a0] removed test2.txt

1 file changed, 1 deletion(-)

delete mode 100644 test2.txt

$ git checkout master

Switched to branch 'master'

$ ls

README        test.txt    test2.txt

如你所见，我们创建了一个分支，在该分支的上下文中移除了一些文件，然后切换回我们的主分支，那些文件又回来了。

使用分支将工作切分开来，从而让我们能够在不同上下文中做事，并来回切换。

# 删除分支

删除分支命令：

git branch -d (branchname)

例如我们要删除"testing"分支：

$ git branch

\* master

testing

$ git branch -d testing

Deleted branch testing (was 85fc7e7).

$ git branch

\* master

# 分支合并

一旦某分支有了独立内容，你终究会希望将它合并回到你的主分支。 你可以使用以下命令将任何分支合并到当前分支中去：

git merge

$ git branch

\* master

newtest

$ ls

README        test.txt    test2.txt

$ git merge newtest

Updating 2e082b7..556f0a0

Fast-forward

test2.txt | 1 -

1 file changed, 1 deletion(-)

delete mode 100644 test2.txt

$ ls

README        test.txt

以上实例中我们将 newtest 分支合并到主分支去，test2.txt 文件被删除。

# 合并冲突

合并并不仅仅是简单的文件添加、移除的操作，Git 也会合并修改。

$ git branch

\* master

$ cat test.txt

runoob.com

首先，我们创建一个叫做"change\_site"的分支，切换过去，我们将内容改为 www.runoob.com 。

$ git checkout -b change\_site

Switched to a new branch 'change\_site'

$ vim test.txt

$ head -1 test.txt

www.runoob.com

$ git commit -am 'changed the site'

[change\_site d7e7346] changed the site

1 file changed, 1 insertion(+), 1 deletion(-)

将修改的内容提交到 "change\_site" 分支中。 现在，假如切换回 "master" 分支我们可以看内容恢复到我们修改前的，我们再次修改test.txt文件。

$ git checkout master

Switched to branch 'master'

$ head -1 test.txt

runoob.com

$ vim test.txt

$ cat test.txt

runoob.com

新增加一行

$ git diff

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

index 704cce7..f84c2a4 100644

--- a/test.txt

+++ b/test.txt

@@ -1 +1,2 @@

runoob.com

+新增加一行

$ git commit -am '新增加一行'

[master 14b4dca] 新增加一行

1 file changed, 1 insertion(+)

现在这些改变已经记录到我的 "master" 分支了。接下来我们将 "change\_site" 分支合并过来。

$ git merge change\_site

Auto-merging test.txt

CONFLICT (content): Merge conflict in test.txt

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

$ cat test.txt

<<<<<<< HEAD

runoob.com

新增加一行

=======

www.runoob.com

>>>>>>> change\_site

我们将前一个分支合并到 "master" 分支，一个合并冲突就出现了，接下来我们需要手动去修改它。

$ vim test.txt

$ cat test.txt

www.runoob.com

新增加一行

$ git diff

diff --cc test.txt

index f84c2a4,bccb7c2..0000000

--- a/test.txt

+++ b/test.txt

@@@ -1,2 -1,1 +1,2 @@@

- runoob.com

+ www.runoob.com

+新增加一行

在 Git 中，我们可以用 git add 要告诉 Git 文件冲突已经解决

$ git status -s

UU test.txt

$ git add test.txt

$ git status -s

M test.txt

$ git commit

[master 88afe0e] Merge branch 'change\_site'

现在我们成功解决了合并中的冲突，并提交了结果。

# Git 查看提交历史

在使用 Git 提交了若干更新之后，又或者克隆了某个项目，想回顾下提交历史，我们可以使用 git log 命令查看。

针对我们前一章节的操作，使用 git log 命令列出历史提交记录如下：

$ git log

commit 88afe0e02adcdfea6844bb627de97da21eb10af1

Merge: 14b4dca d7e7346

Author: runoob <runoob@runoob.com>

Date: Sun Mar 1 15:03:42 2015 +0800

Merge branch 'change\_site'

Conflicts:

test.txt

commit 14b4dcadbdc847207651d5a9fae0d315057f346e

Author: runoob <runoob@runoob.com>

Date: Sun Mar 1 14:53:15 2015 +0800

新增加一行

commit d7e734640da06055e107eaf29cf350b3f1de1c2c

Author: runoob <runoob@runoob.com>

Date: Sun Mar 1 14:48:57 2015 +0800

changed the site

commit 556f0a0637978097b82287ac665a717623b21f3f

Author: runoob <runoob@runoob.com>

Date: Sun Mar 1 14:40:34 2015 +0800

removed test2.txt

我们可以用 --oneline 选项来查看历史记录的简洁的版本。

$ git log --oneline

88afe0e Merge branch 'change\_site'

14b4dca 新增加一行

d7e7346 changed the site

556f0a0 removed test2.txt

2e082b7 add test2.txt

048598f add test.txt

85fc7e7 test comment from runoob.com

这告诉我们的是，此项目的开发历史。

我们还可以用 --graph 选项，查看历史中什么时候出现了分支、合并。以下为相同的命令，开启了拓扑图选项：

$ git log --oneline --graph

\* 88afe0e Merge branch 'change\_site'

|\

| \* d7e7346 changed the site

\* | 14b4dca 新增加一行

|/

\* 556f0a0 removed test2.txt

\* 2e082b7 add test2.txt

\* 048598f add test.txt

\* 85fc7e7 test comment from runoob.com

现在我们可以更清楚明了地看到何时工作分叉、又何时归并。

你也可以用 '--reverse'参数来逆向显示所有日志。

$ git log --reverse --oneline

85fc7e7 test comment from runoob.com

048598f add test.txt

2e082b7 add test2.txt

556f0a0 removed test2.txt

d7e7346 changed the site

14b4dca 新增加一行

88afe0e Merge branch 'change\_site'

如果只想查找指定用户的提交日志可以使用命令：git log --author , 例如，比方说我们要找 Git 源码中 Linus 提交的部分：

$ git log --author=Linus --oneline -5

81b50f3 Move 'builtin-\*' into a 'builtin/' subdirectory

3bb7256 make "index-pack" a built-in

377d027 make "git pack-redundant" a built-in

b532581 make "git unpack-file" a built-in

112dd51 make "mktag" a built-in

如果你要指定日期，可以执行几个选项：--since 和 --before，但是你也可以用 --until 和 --after。

例如，如果我要看 Git 项目中三周前且在四月十八日之后的所有提交，我可以执行这个（我还用了 --no-merges 选项以隐藏合并提交）：

$ git log --oneline --before={3.weeks.ago} --after={2010-04-18} --no-merges

5469e2d Git 1.7.1-rc2

d43427d Documentation/remote-helpers: Fix typos and improve language

272a36b Fixup: Second argument may be any arbitrary string

b6c8d2d Documentation/remote-helpers: Add invocation section

5ce4f4e Documentation/urls: Rewrite to accomodate transport::address

00b84e9 Documentation/remote-helpers: Rewrite description

03aa87e Documentation: Describe other situations where -z affects git diff

77bc694 rebase-interactive: silence warning when no commits rewritten

636db2c t3301: add tests to use --format="%N"