## Git 配置

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

这些环境变量，决定了 Git 在各个环节的具体工作方式和行为。这些变量可以存放在以下三个不同的地方：

* /etc/gitconfig 文件：系统中对所有用户都普遍适用的配置。若使用 git config 时用 --system 选项，读写的就是这个文件。
* ~/.gitconfig 文件：用户目录下的配置文件只适用于该用户。若使用 git config 时用 --global 选项，读写的就是这个文件。
* 当前项目的 Git 目录中的配置文件（也就是工作目录中的 .git/config 文件）：这里的配置仅仅针对当前项目有效。每一个级别的配置都会覆盖上层的相同配置，所以 .git/config 里的配置会覆盖 /etc/gitconfig 中的同名变量。

在 Windows 系统上，Git 会找寻用户主目录下的 .gitconfig 文件。主目录即 $HOME 变量指定的目录，一般都是 C:\Documents and Settings\$USER。

此外，Git 还会尝试找寻 /etc/gitconfig 文件，只不过看当初 Git 装在什么目录，就以此作为根目录来定位。

### **用户信息**

配置个人的用户名称和电子邮件地址：

$ git config --global user.name "runoob"\_x000D\_$ git config --global user.email test@runoob.com

如果用了 **--global** 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这里配置的用户信息。

如果要在某个特定的项目中使用其他名字或者电邮，只要去掉 --global 选项重新配置即可，新的设定保存在当前项目的 .git/config 文件里。

### **文本编辑器**

设置Git默认使用的文本编辑器, 一般可能会是 Vi 或者 Vim。如果你有其他偏好，比如 Emacs 的话，可以重新设置：:

$ git config --global core.editor emacs

### **差异分析工具**

还有一个比较常用的是，在解决合并冲突时使用哪种差异分析工具。比如要改用 vimdiff 的话：

$ git config --global merge.tool vimdiff

Git 可以理解 kdiff3，tkdiff，meld，xxdiff，emerge，vimdiff，gvimdiff，ecmerge，和 opendiff 等合并工具的输出信息。

当然，你也可以指定使用自己开发的工具.

### **查看配置信息**

要检查已有的配置信息，可以使用 git config --list 命令：

$ git config --list\_x000D\_http.postbuffer=2M\_x000D\_user.name=runoob\_x000D\_user.email=test@runoob.com

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如 /etc/gitconfig 和 ~/.gitconfig），不过最终 Git 实际采用的是最后一个。

这些配置我们也可以在 **~/.gitconfig** 或 **/etc/gitconfig** 看到，如下所示：

vim ~/.gitconfig

显示内容如下所示：

[http]\_x000D\_ postBuffer = 2M[user]\_x000D\_ name = runoob\_x000D\_ email = test@runoob.com

也可以直接查阅某个环境变量的设定，只要把特定的名字跟在后面即可，像这样：

$ git config user.name\_x000D\_runoob

# Git 工作流程

本章节我们将为大家介绍 Git 的工作流程。

一般工作流程如下：

* 克隆 Git 资源作为工作目录。
* 在克隆的资源上添加或修改文件。
* 如果其他人修改了，你可以更新资源。
* 在提交前查看修改。
* 提交修改。
* 在修改完成后，如果发现错误，可以撤回提交并再次修改并提交。

# Git 工作区、暂存区和版本库

## 基本概念

我们先来理解下Git 工作区、暂存区和版本库概念

* **工作区：**就是你在电脑里能看到的目录。
* **暂存区：**英文叫stage, 或index。一般存放在"git目录"下的index文件（.git/index）中，所以我们把暂存区有时也叫作索引（index）。
* **版本库：**工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

# 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\_x000D\_$ cd runoob/\_x000D\_$ git initInitialized 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.gitCloning into 'simplegit'...\_x000D\_remote: Counting objects: 13, done.\_x000D\_remote: Total 13 (delta 0), reused 0 (delta 0), pack-reused 13Receiving 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\_x000D\_$ cd .git\_x000D\_$ ls\_x000D\_HEAD description info packed-refs\_x000D\_branches hooks logs refs\_x000D\_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\_x000D\_$ touch hello.php\_x000D\_$ ls\_x000D\_README hello.php\_x000D\_$ git status -s?? README?? hello.php\_x000D\_$

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\_x000D\_A README\_x000D\_A hello.php\_x000D\_$

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<pre><p>在 README 添加以下内容：<b># Runoob Git 测试</b>，然后保存退出。</p><p>再执行一下 git status：</p>\_x000D\_$ git status -s\_x000D\_AM README\_x000D\_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。改动后我们在执行 git add 命令将其添加到缓存中：

$ git add .\_x000D\_$ git status -s\_x000D\_A README\_x000D\_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 statusOn branch master\_x000D\_Initial commit\_x000D\_Changes to be committed:\_x000D\_ (use "git rm --cached <file>..." to unstage)\_x000D\_\_x000D\_ new file: README\_x000D\_ 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\_x000D\_echo '菜鸟教程：www.runoob.com';?>

$ git status -s\_x000D\_A README\_x000D\_AM hello.php\_x000D\_$ git diff\_x000D\_diff --git a/hello.php b/hello.php\_x000D\_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 \_x000D\_$ git status -s\_x000D\_A README\_x000D\_A hello.php\_x000D\_$ git diff --cached\_x000D\_diff --git a/README b/READMEnew file mode 100644\_x000D\_index 0000000..8f87495--- /dev/null+++ b/README@@ -0,0 +1 @@+# Runoob Git 测试\_x000D\_diff --git a/hello.php b/hello.phpnew file mode 100644\_x000D\_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'\_x000D\_$ 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\_x000D\_

$ git status -s

\_x000D\_A README\_x000D\_A hello.php\_x000D\_$

$ git commit -m '第一次版本提交'

[master (root-commit) d32cf1f] 第一次版本提交\_x000D\_ 2 files changed, 4 insertions(+)\_x000D\_ create mode 100644 README\_x000D\_ 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\_x000D\_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\_x000D\_echo '菜鸟教程：www.runoob.com';\_x000D\_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\_x000D\_ 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\_x000D\_echo '菜鸟教程：www.runoob.com';\_x000D\_echo '菜鸟教程：www.runoob.com';\_x000D\_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\_x000D\_ M README\_x000D\_ M hello.php\_x000D\_$ git add .\_x000D\_$ git status -s\_x000D\_M README\_x000D\_M hello.php\_x000D\_$ git reset HEAD -- hello.php

Unstaged changes after reset:\_x000D\_M hello.php\_x000D\_$ git status -s\_x000D\_M README\_x000D\_ 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\_x000D\_ 1 file changed, 1 insertion(+)\_x000D\_$ git status -s\_x000D\_ 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\_x000D\_ 1 file changed, 1 insertion(+)\_x000D\_$ git status

On branch master\_x000D\_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**

git rm 会将条目从缓存区中移除。这与 git reset HEAD 将条目取消缓存是有区别的。 "取消缓存"的意思就是将缓存区恢复为我们做出修改之前的样子。

默认情况下，**git rm file** 会将文件从缓存区和你的硬盘中（工作目录）删除。

如果你要在工作目录中留着该文件，可以使用 **git rm --cached**：

如我们删除 hello.php文件：

$ git rm hello.php \_x000D\_rm 'hello.php'\_x000D\_$ ls\_x000D\_README

不从工作区中删除文件：

$ git rm --cached README \_x000D\_rm 'README'\_x000D\_$ ls\_x000D\_README

### **git mv**

git mv 命令做得所有事情就是 **git rm --cached** 命令的操作， 重命名磁盘上的文件，然后再执行 git add 把新文件添加到缓存区。

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\_x000D\_$ ls\_x000D\_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，并切换过去。执行 git branch (branchname) 即可。

$ git branch testing\_x000D\_$ git branch\* master\_x000D\_ 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

X`

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\_x000D\_README\_x000D\_$ echo 'w3cschool.cc' > test.txt\_x000D\_$ git add .\_x000D\_$ git commit -m 'add test.txt'[master 048598f] add test.txt\_x000D\_ 2 files changed, 1 insertion(+), 3 deletions(-)\_x000D\_ delete mode 100644 hello.php\_x000D\_ create mode 100644 test.txt\_x000D\_$ ls\_x000D\_README test.txt\_x000D\_$ git checkout testingSwitched to branch 'testing'\_x000D\_$ ls\_x000D\_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'\_x000D\_$ ls\_x000D\_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'\_x000D\_$ git rm test2.txt \_x000D\_rm 'test2.txt'\_x000D\_$ ls\_x000D\_README test.txt\_x000D\_$ git commit -am 'removed test2.txt'[newtest 556f0a0] removed test2.txt\_x000D\_ 1 file changed, 1 deletion(-)\_x000D\_ delete mode 100644 test2.txt\_x000D\_$ git checkout master

Switched to branch 'master'\_x000D\_$ ls\_x000D\_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\_x000D\_ testing\_x000D\_$ git branch -d testing

Deleted branch testing (was 85fc7e7).\_x000D\_$ 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\_x000D\_ newtest\_x000D\_$ ls\_x000D\_README test.txt test2.txt\_x000D\_$ git merge newtest

Updating 2e082b7..556f0a0

Fast-forward\_x000D\_ test2.txt | 1 -\_x000D\_ 1 file changed, 1 deletion(-)\_x000D\_ delete mode 100644 test2.txt\_x000D\_$ ls\_x000D\_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\_x000D\_$ cat test.txt\_x000D\_w3cschool.cc

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.w3cschool.cc 。

$ git checkout -b change\_site

Switched to a new branch 'change\_site'\_x000D\_$ vim test.txt \_x000D\_$ head -1 test.txt \_x000D\_www.w3cschool.cc\_x000D\_$ git commit -am 'changed the site'

[change\_site d7e7346] changed the site\_x000D\_ 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'\_x000D\_$ head -1 test.txt \_x000D\_w3cschool.cc\_x000D\_$ vim test.txt \_x000D\_$ cat test.txt\_x000D\_w3cschool.cc

新增加一行\_x000D\_$ git diff\_x000D\_diff --git a/test.txt b/test.txt\_x000D\_index 704cce7..f84c2a4 100644

--- a/test.txt

+++ b/test.txt

@@ -1 +1,2 @@\_x000D\_ w3cschool.cc

+新增加一行\_x000D\_$ git commit -am '新增加一行'

[master 14b4dca] 新增加一行\_x000D\_ 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\_x000D\_CONFLICT (content): Merge conflict in test.txt

Automatic merge failed; fix conflicts and then commit the result.\_x000D\_$ cat test.txt <<<<<<< HEAD\_x000D\_w3cschool.cc

新增加一行

=======\_x000D\_www.w3cschool.cc

>>>>>>> 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 \_x000D\_$ cat test.txt \_x000D\_www.w3cschool.cc

新增加一行\_x000D\_$ git diff\_x000D\_diff --cc test.txt\_x000D\_index f84c2a4,bccb7c2..0000000

--- a/test.txt

+++ b/test.txt

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

- w3cschool.cc

+ www.w3cschool.cc\_x000D\_ +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\_x000D\_UU test.txt\_x000D\_$ git add test.txt \_x000D\_$ git status -s\_x000D\_M test.txt\_x000D\_$ 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。