## 基本概念

对于任何一个文件，在 Git 内都只有三种状态：已提交（committed），已修改（modified）和已暂存（staged）。

已提交表示该文件已经被安全地保存在本地数据库 中了；

已修改表示修改了某个文件，但还没有提交保存；

已暂存表示把已修改的文件放在下次提交时要保存的清单中。  
由此我们看到 Git 管理项目时，文件流转的三个工作区域：Git 的工作目录，暂存区域，以及本地仓库。

每个项目都有一个 Git 目录（译注：如果 git clone 出来的话，就是其中 .git 的目录；如果git clone --bare 的话，新建的目录本身就是 Git 目录。），它是 Git 用来保存元数据和对象数据库的地方。该目录非常重要，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据。  
　　从项目中取出某个版本的所有文件和目录，用以开始后续工作的叫做工作目录。这些文件实际上都是从 Git 目录中的压缩对象数据库中提取出来的，接下来就可以在工作目录中对这些文件进行编辑。  
所谓的暂存区域只不过是个简单的文件，一般都放在 Git 目录中。有时候人们会把这个文件叫做索引文件，不过标准说法还是叫暂存区域。  
基本的 Git 工作流程如下：  
　1. 在工作目录中修改某些文件。   
　2. 对修改后的文件进行快照，然后保存到暂存区域。   
　3. 提交更新，将保存在暂存区域的文件快照永久转储到 Git 目录中。  
　　所以，我们可以从文件所处的位置来判断状态：如果是 Git 目录中保存着的特定版本文件，就属于已提交状态；如果作了修改并已放入暂存区域，就属于已暂存状态；如果自上次取出后，作了修改但还没有放到暂存区域，就 是已修改状态。到第二章的时候，我们会进一步了解其中细节，并学会如何根据文件状态实施后续操作，以及怎样跳过暂存直接提交。

### **在 Windows 上安装**

在 Windows 上安装 Git 同样轻松，有个叫做 msysGit 的项目提供了安装包，可以到 Google Code 的页面上下载 exe 安装文件并运行：  
<http://code.google.com/p/msysgit>完成安装之后，就可以使用命令行的 git 工具（已经自带了 ssh 客户端）了，另外还有一个图形界面的 Git 项目管理工具。

### 初次运行 Git 前的配置

一般在新的系统上，我们都需要先配置下自己的 Git 工作环境。配置工作只需一次，以后升级时还会沿用现在的配置。当然，如果需要，你随时可以用相同的命令修改已有的配置。  
Git 提供了一个叫做 git config 的工具（译注：实际是 git-config 命令，只不过可以通过 git 加一个名字来呼叫此命令。），专门用来配置或读取相应的工作环境变量。而正是由这些环境变量，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 $HOME 变量指定的目录，一般都是Chttp://www.qter.org/static/image/smiley/default/biggrin.gifocuments and Settings$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**  
　　第一个要配置的是你个人的用户名称和电子邮件地址。这两条配置很重要，每次 Git 提交时都会引用这两条信息，说明是谁提交了更新，所以会随更新内容一起被永久纳入历史记录：  
$ git config --global user.name "John Doe"

$ git config --global user.email [johndoe@example.com](mailto:johndoe@example.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**  
　　接下来要设置的是默认使用的文本编辑器。Git 需要你输入一些额外消息的时候，会自动调用一个外部文本编辑器给你用。默认会使用操作系统指定的默认编辑器，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 vimdiffGit   
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，具体怎么做可以参阅第七章。  
  
**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 --listuser.name=Scott Chaconuser.email=schacon@gmail.comcolor.status=autocolor.branch=autocolor.interactive=autocolor.diff=auto...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。  
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.nameScott Chacon  
  
**1.6 获取帮助**  
　　想了解 Git 的各式工具该怎么用，可以阅读它们的使用帮助，方法有三：  
$ git help  $ git --help  
$ man git-       
比如，要学习 config 命令可以怎么用，运行：  
$ git help config我们随时都可以浏览这些帮助信息而无需连网。不过，要是你觉得还不够，可以到 Frenode IRC 服务器（irc.freenode.net）上的 #git 或 #github 频道寻求他人帮助。这两个频道上总有着上百号人，大多都有着丰富的 git 知识，并且乐于助人。

## 操作

### 在工作目录中初始化新仓库

要对现有的某个项目开始用 Git 管理，只需到此项目所在的目录，执行：

|  |  |
| --- | --- |
| 1 | $ git init |

初始化后，在当前目录下会出现一个名为 .git 的目录，所有 Git 需要的数据和资源都存放在这个目录中。不过目前，仅仅是按照既有的结构框架初始化好了里边所有的文件和目录，但我们还没有开始跟踪管理项目中的任何一个文件。

### 当前目录下有几个文件想要纳入版本控制

如果当前目录下有几个文件想要纳入版本控制，需要先用 git add 命令告诉 Git 开始对这些文件进行跟踪，然后提交：

|  |  |
| --- | --- |
| 1 2 3 | $ git add \*.c $ git add README $ git commit -m 'initial project version' |

**检查当前文件状态**

要确定哪些文件当前处于什么状态，可以用 git status 命令。如果在克隆仓库之后立即执行此命令，会看到类似这样的输出：

|  |  |
| --- | --- |
| 1 | $ git status # On branch master nothing to commit (working directory clean) |

这说明你现在的工作目录相当干净。换句话说，当前没有任何跟踪着的文件，也没有任何文件在上次提交后更改过。此外，上面的信息还表明，当前目录下没 有出现任何处于未跟踪的新文件，否则 Git 会在这里列出来。最后，该命令还显示了当前所在的分支是 master，这是默认的分支名称，实际是可以修改的，现在先不用考虑。下一章我们就会详细讨论分支和引用。

现在让我们用 vim 编辑一个新文件 README，保存退出后运行 git status 会看到该文件出现在未跟踪文件列表中：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | $ vim README $ git status # On branch master # Untracked files: #   (use "git add        ..." to include in what will be committed) # #   README nothing added to commit but untracked files present (use "git add" to track) |

就是在“Untracked files”这行下面。Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”，因而不用担心把临时文件什么的也归入版本管理。不过现在的例子中，我们确实想要跟踪管理 README 这个文件。

**跟踪新文件**

使用命令 git add 开始跟踪一个新文件。所以，要跟踪 README 文件，运行：

|  |  |
| --- | --- |
| 1 | $ git add README |

此时再运行 git status 命令，会看到 README 文件已被跟踪，并处于暂存状态：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | $ git status # On branch master # Changes to be committed: #   (use "git reset HEAD        ..." to unstage) # #   new file:   README # |

只要在 “Changes to be committed” 这行下面的，就说明是已暂存状态。如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。你可能会想起之前我们使用git init 后就运行了 git add 命令，开始跟踪当前目录下的文件。在 git add 后面可以指明要跟踪的文件或目录路径。如果是目录的话，就说明要递归跟踪该目录下的所有文件。（译注：其实git add 的潜台词就是把目标文件快照放入暂存区域，也就是 add file into staged area，同时未曾跟踪过的文件标记为需要跟踪。这样就好理解后续 add 操作的实际意义了。）

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ****马上注册，查看详细内容！注册请先查看：****[注册须知](http://www.qter.org/forum.php?mod=viewthread&tid=5938) 您需要 [登录](http://www.qter.org/member.php?mod=logging&action=login) 才可以下载或查看，没有帐号？[立即注册](http://www.qter.org/member.php?mod=register)  x    原文：[《Pro Git》](http://git-scm.com/book/zh)  **Git 基础** 读完本章你就能上手使用 Git 了（[伯乐](http://www.jobbole.com/)在线注：如果你对Git还不了解，建议从本[Git](http://blog.jobbole.com/25775/)系列[第一篇文章](http://blog.jobbole.com/25775/)开始阅读）。本章将介绍几个最基本的，也是最常用的 Git 命令，以后绝大多数时间里用到的也就是这几个命令。读完本章，你就能初始化一个新的代码仓库，做一些适当配置；开始或停止跟踪某些文件；暂存或提交某些更 新。我们还会展示如何让 Git 忽略某些文件，或是名称符合特定模式的文件；如何既快且容易地撤消犯下的小错误；如何浏览项目的更新历史，查看某两次更新之间的差异；以及如何从远程仓库 拉数据下来或者推数据上去。  **2.1  取得项目的 Git 仓库**  有两种取得 Git 项目仓库的方法。第一种是在现存的目录下，通过导入所有文件来创建新的 Git 仓库。第二种是从已有的 Git 仓库克隆出一个新的镜像仓库来。  **在工作目录中初始化新仓库**  要对现有的某个项目开始用 Git 管理，只需到此项目所在的目录，执行：   |  |  | | --- | --- | | 1 | $ git init |   初始化后，在当前目录下会出现一个名为 .git 的目录，所有 Git 需要的数据和资源都存放在这个目录中。不过目前，仅仅是按照既有的结构框架初始化好了里边所有的文件和目录，但我们还没有开始跟踪管理项目中的任何一个文件。（在第九章我们会详细说明刚才创建的.git 目录中究竟有哪些文件，以及都起些什么作用。） 如果当前目录下有几个文件想要纳入版本控制，需要先用 git add 命令告诉 Git 开始对这些文件进行跟踪，然后提交：   |  |  | | --- | --- | | 1 2 3 | $ git add \*.c $ git add README $ git commit -m 'initial project version' |   稍后我们再逐一解释每条命令的意思。不过现在，你已经得到了一个实际维护着若干文件的 Git 仓库。  **从现有仓库克隆**  如果想对某个开源项目出一份力，可以先把该项目的 Git 仓库复制一份出来，这就需要用到 git clone 命令。如果你熟悉其他的 VCS 比如 Subversion，你可能已经注意到这里使用的是 clone 而不是 checkout。这是个非常重要的差别，Git 收取的是项目历史的所有数据（每一个文件的每一个版本），服务器上有的数据克隆之后本地也都有了。实际上，即便服务器的磁盘发生故障，用任何一个克隆出来 的客户端都可以重建服务器上的仓库，回到当初克隆时的状态（虽然可能会丢失某些服务器端的挂钩设置，但所有版本的数据仍旧还在，有关细节请参考第四章）。[GitHub](http://blog.jobbole.com/6492/) 克隆仓库的命令格式为 git clone [http://blog.jobbole.com/wp-conte ... 0201121432\_845.png]](http://blog.jobbole.com/wp-content/uploads/2012/08/20120201121432_845.png%5d)  图 2-1. 文件的状态变化周期  **检查当前文件状态**  要确定哪些文件当前处于什么状态，可以用 git status 命令。如果在克隆仓库之后立即执行此命令，会看到类似这样的输出：   |  |  | | --- | --- | | 1 | $ git status # On branch master nothing to commit (working directory clean) |   这说明你现在的工作目录相当干净。换句话说，当前没有任何跟踪着的文件，也没有任何文件在上次提交后更改过。此外，上面的信息还表明，当前目录下没 有出现任何处于未跟踪的新文件，否则 Git 会在这里列出来。最后，该命令还显示了当前所在的分支是 master，这是默认的分支名称，实际是可以修改的，现在先不用考虑。下一章我们就会详细讨论分支和引用。 现在让我们用 vim 编辑一个新文件 README，保存退出后运行 git status 会看到该文件出现在未跟踪文件列表中：   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 9 10 | $ vim README $ git status # On branch master # Untracked files: #   (use "git add        ..." to include in what will be committed) # #   README nothing added to commit but untracked files present (use "git add" to track) |   就是在“Untracked files”这行下面。Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”，因而不用担心把临时文件什么的也归入版本管理。不过现在的例子中，我们确实想要跟踪管理 README 这个文件。  **跟踪新文件**  使用命令 git add 开始跟踪一个新文件。所以，要跟踪 README 文件，运行：   |  |  | | --- | --- | | 1 | $ git add README |   此时再运行 git status 命令，会看到 README 文件已被跟踪，并处于暂存状态：   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 9 | $ git status # On branch master # Changes to be committed: #   (use "git reset HEAD        ..." to unstage) # #   new file:   README # |   只要在 “Changes to be committed” 这行下面的，就说明是已暂存状态。如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。你可能会想起之前我们使用git init 后就运行了 git add 命令，开始跟踪当前目录下的文件。在 git add 后面可以指明要跟踪的文件或目录路径。如果是目录的话，就说明要递归跟踪该目录下的所有文件。（译注：其实git add 的潜台词就是把目标文件快照放入暂存区域，也就是 add file into staged area，同时未曾跟踪过的文件标记为需要跟踪。这样就好理解后续 add 操作的实际意义了。）  **暂存已修改文件**  现在我们修改下之前已跟踪过的文件 benchmarks.rb，然后再次运行 status 命令，会看到这样的状态报告：   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | $ git status # On branch master # Changes to be committed: #   (use "git reset HEAD        ..." to unstage) # #   new file:   README # # Changed but not updated: #   (use "git add         ..." to update what will be committed) # #   modified:   benchmarks.rb # |   文件 benchmarks.rb 出现在 “Changed but not updated” 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。要暂存这次更新，需要运行git add 命令（这是个多功能命令，根据目标文件的状态不同，此命令的效果也不同：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等）。现在让我们运行git add 将 benchmarks.rb 放到暂存区，然后再看看 git status 的输出：   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 9 10 11 | $ git add benchmarks.rb $ git status # On branch master # Changes to be committed: #   (use "git reset HEAD        ..." to unstage) # #   new file:   README #   modified:   benchmarks.rb # |   现在两个文件都已暂存，下次提交时就会一并记录到仓库。假设此时，你想要在 benchmarks.rb 里再加条注释，重新编辑存盘后，准备好提交。不过且慢，再运行git status 看看：   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | $ vim benchmarks.rb $ git status # On branch master # Changes to be committed: #   (use "git reset HEAD        ..." to unstage) # #   new file:   README #   modified:   benchmarks.rb # # Changed but not updated: #   (use "git add         ..." to update what will be committed) # #   modified:   benchmarks.rb # |   怎么回事？benchmarks.rb 文件出现了两次！一次算未暂存，一次算已暂存，这怎么可能呢？好吧，实际上 Git 只不过暂存了你运行 git add 命令时的版本，如果现在提交，那么提交的是添加注释前的版本，而非当前工作目录中的版本。所以，运行了git add 之后又作了修订的文件，需要重新运行 git add 把最新版本重新暂存起来：   |  |  | | --- | --- | | 1 2 3 4 5 6 7 8 9 10 11 | $ git add benchmarks.rb $ git status # On branch master # Changes to be committed: #   (use "git reset HEAD        ..." to unstage) # #   new file:   README #   modified:   benchmarks.rb # |   **忽略某些文件** 一般我们总会有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。我们可以创建一个名为 .gitignore 的文件，列出要忽略的文件模式。来看一个实际的例子：   |  |  | | --- | --- | | 1 | $ cat .gitignore \*.[oa] \*~ | |