**版本控制工具Git**

**使用指南**

**一品通电子商务有限公司技术中心**

**目录**

[1、 版本控制基础 3](#_Toc460251723)

[1.1什么是版本控制工具git？ 3](#_Toc460251724)

[1.2 版本控制基本概念 3](#_Toc460251725)

[1.2.1 git文件的状态变化 3](#_Toc460251726)

[1.3 Git常用命令 5](#_Toc460251727)

[1.3.1获得仓库 5](#_Toc460251728)

[1.3.2 记录每次更新到仓库 5](#_Toc460251729)

[1.3.3 获取信息 5](#_Toc460251730)

[1.3.4 分支与标记 5](#_Toc460251731)

[1.3.5 版本恢复 6](#_Toc460251732)

[1.3.6 插销操作 6](#_Toc460251733)

[1.3.7远程仓库的使用 6](#_Toc460251734)

[1.4 Git命令图解 7](#_Toc460251735)

[1.5 Git的工作流程 7](#_Toc460251736)

[1.6 Git的分支 7](#_Toc460251737)

[1.6.1 分支的概念 7](#_Toc460251738)

[1.6.2 Git分支简介 8](#_Toc460251739)

[1.6.3 分支的创建 9](#_Toc460251740)

[1.6.4分支切换 10](#_Toc460251741)

[1.6.5 分支创建和合并（实际例子） 13](#_Toc460251742)

[1.6.6 分支的管理 22](#_Toc460251743)

[1.6.7 分支开发流程 23](#_Toc460251744)

[1.6.8 远程分支 26](#_Toc460251745)

[2、 Git的安装 32](#_Toc460251746)

[2.1 Git服务器端安装和配置 32](#_Toc460251747)

[2.2 Git客户端安装： 32](#_Toc460251748)

[2）安装TortoiseGit（图形操作界面）和中文补丁包 32](#_Toc460251749)

[3、 Git客户端设置 33](#_Toc460251750)

[3.1 访问账号设置 33](#_Toc460251751)

[3.2设置用户名和电子邮箱 33](#_Toc460251752)

[3.3设置账户的SSH key： 33](#_Toc460251753)

[3.4 生成本地密钥文件 34](#_Toc460251754)

[3.5访问Git 命令行工具 35](#_Toc460251755)

[4、 Git操作演示 36](#_Toc460251756)

[4.1从当前目录初始化 36](#_Toc460251757)

[4.2.2 从现有仓库克隆 38](#_Toc460251758)

[4.3 推送本地库到服务器 39](#_Toc460251759)

# 版本控制基础

## 1.1什么是版本控制工具git？

1) Git是一个开源的分布式版本控制系统。

2）与SVN，CVS相比， Git不需要中心仓库

3）Git的分支与合并比较简单，版本对比快

4）GitHub是一个基于web与Git的一个项目托管服务平台，要求源代码必须公开

5）GitLab是一个用于仓库管理的开源项目，使用Git作为代码管理工具，可以部署在本地或私有服务器上

## 1.2 版本控制基本概念

1）Repository(本地库和远程库)

2）Pull/Push/Checkout/Feteh

3) Branch

4) Merge

5) Conflict

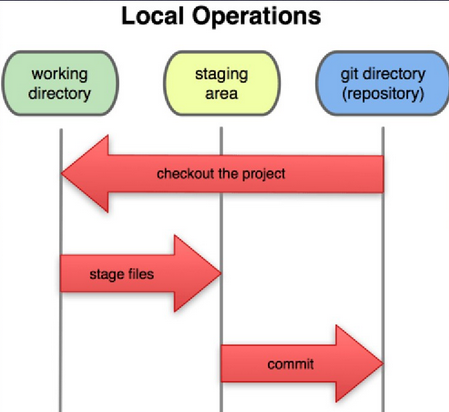
6) Revert

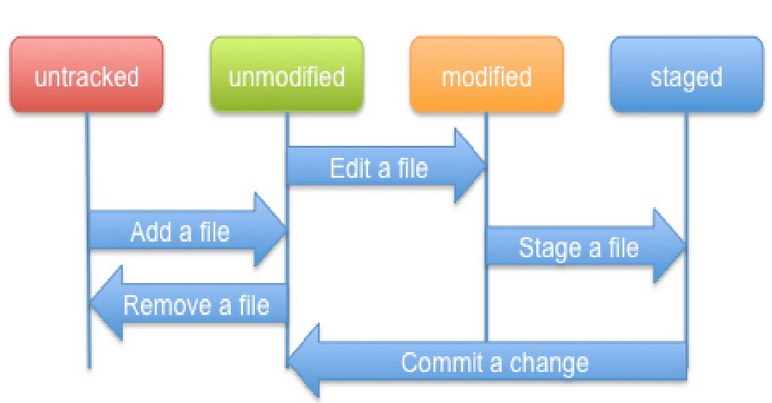
7) Working Directory

### 1.2.1 git文件的状态变化

文件的三种状态：

已提交(committed),已修改（modified，已暂存（staged）





1. 工作目录下的文件有两种状态：已跟踪或未跟踪(tracked or untracked)
2. 已跟踪的文件是指已被纳入版本控制管理的文件，在上次快照中有他们的记录，工作一段时间以后，工作一段时间后，它们的状态可能是未更新，已修改或已放入暂存区
3. 未跟踪的文件是指没有纳入版本控制的文件，它们既没有上次更新时的快照，也不在当前的暂存区
4. 初次克隆某个版本仓库时，工作目录中的所有文件都属于已跟踪文件，且状态为未修改
5. 通过git add将文件改为statged状态
6. 文件修改了后提交时加-a来载入

## 1.3 Git常用命令

### 1.3.1获得仓库

1. Git init
2. Git clone
3. Git pull与git fetch的区别
4. Get fetch从远程获取最新版本到本地，不会自动merge
5. Git pull 从远程获取最新版本并merge 到本地，相当于git fetch + Git merge

### 1.3.2 记录每次更新到仓库

* Git status 查看当前文件状态
* Git add 跟踪新文件， 暂存已修改过的文件
* Gitdiff，Git diff –staged 查看已暂存和未暂存的更新
* Git commit –m “更新说明”
* Git commit –a 跳过使用暂存区
* Git rm 移除文件
* Git mv 移动文件

### 1.3.3 获取信息

1）get help

2) git status

3) git diff

4) git log 查看提交历史

5) git show

### 1.3.4 分支与标记

* 创建分支

Git branch <name>

Git branch <name> <commit-id>

* 删除分支

Git branch –d <name>

* 查看分支

Git branch

Git branch –r 查看远程服务器上的分支

* 转移到某个分支

Git checkout <commit-id>

Git checkout –b <name><commit-id>

* 分支合并

Git merge <name> 合并制定分支到当前分支

* 标记

标记用于标识某次比较重要的提交

Git tag <tag-name>

### 1.3.5 版本恢复

* Reset

将当前的工作目录完全回滚到指定的版本号

* Revert

还原一个版本的修改，必须提供一个具体的git版本号

例如 git revert bbaf6f

* Reset 和Revert的区别

Reset是将当前head的内容重置，不会留任何痕迹

Revert是撤销某次提交，但是这次撤销也会作为一次提交进行保存

### 1.3.6 插销操作

Git commit –amend 修改最后一次提交

Git reset HEAD <file> 取消已经暂存的文件

Git checkout --<file> 取消对文件的修改

### 1.3.7远程仓库的使用

Git remote 查看当前的远程库

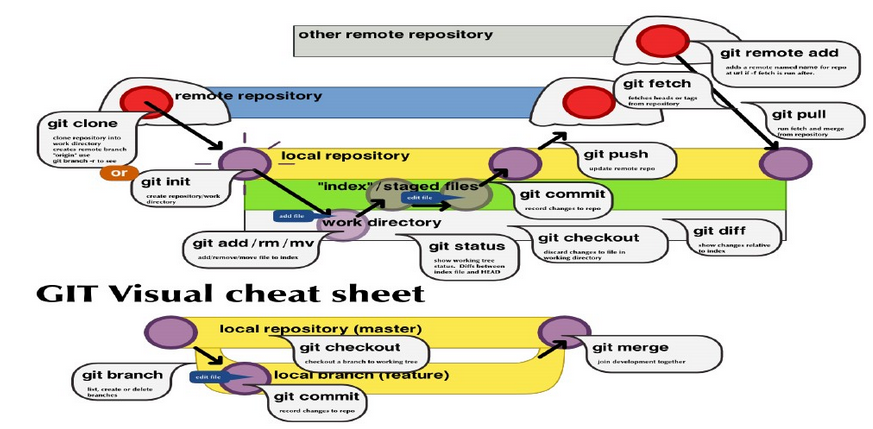
Git remote add添加远程仓库

Git fetch [remote –name] 从远程仓库抓取数据

Git push [remote –name] 推送数据到远程仓库

Git remote show [remote –name] 查看远程仓库信息

## 1.4 Git命令图解



## 1.5 Git的工作流程

* 与远程仓库同步（pull，fetch）
* 修改文件
* 查看变更（show，status）
* 载入变更（add or-a）
* 提交载入的变更（commit）
* 重复
* 上传（push）

## 1.6 Git的分支

### 1.6.1 分支的概念

分支是指在软件开发中每个开发人员在已有版本的基础上按照各自任务的不同创建个人的软件库副本，每个人可以互不影响地工作在自己的副本里，等工作完成后再把工作成果提交到公共的软件库，如果有冲突，就解决冲突后再提交。

几乎所有的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把你的工作从开发主线上分离开来，以免影响开发主线。

### 1.6.2 Git分支简介

假设现在有一个工作目录，里面包含了三个将要被暂存和提交的文件。 暂存操作会为每一个文件计算校验和（使用SHA-1 哈希算法），然后会把当前版本的文件快照保存到 Git 仓库中（Git 使用 blob 对象来保存它们），最终将校验和加入到暂存区域等待提交：

git add README test.rb

git commit -m 'The initial commit of my project'

当使用 git commit 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和，然后在 Git 仓库中这些校验和保存为树对象。 随后，Git 便会创建一个提交对象，它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。如此一来，Git 就可以在需要的时候重现此次保存的快照。

现在，Git 仓库中有五个对象：三个 blob 对象（保存着文件快照）、一个树对象（记录着目录结构和 blob 对象索引）以及一个提交对象（包含着指向前述树对象的指针和所有提交信息）。

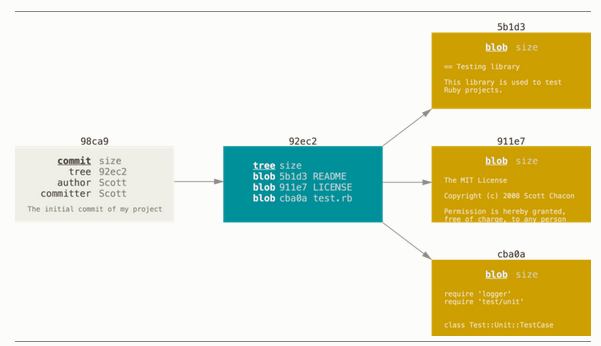


Figure 1-1. 首次提交对象及其树结构

做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。

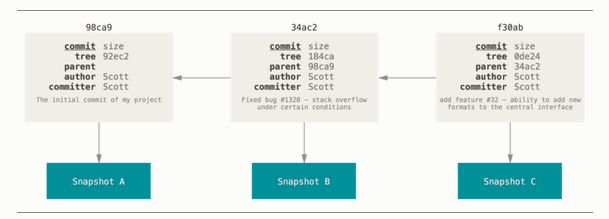


Figure 1-2. 提交对象及其父对象

Git 的分支，其实本质上仅仅是指向提交对象的可变指针。 Git 的默认分支名字是 master。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。 它会在每次的提交操作中自动向前移动。

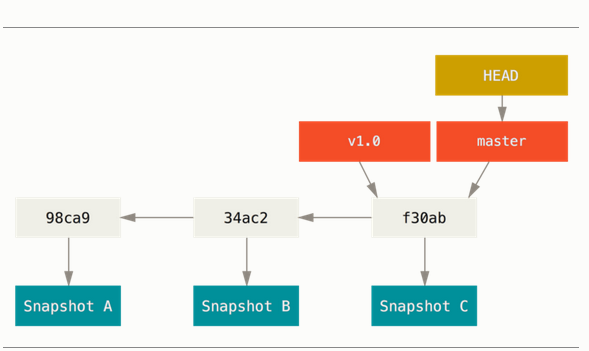


Figure 1-3. 分支及其提交历史

### 1.6.3 分支的创建

Git创建新分支的方式很简单，它只是为你创建了一个可以移动的新的指针。 比如，创建一个 testing 分支， 你需要使用 git branch 命令：

git branch testing

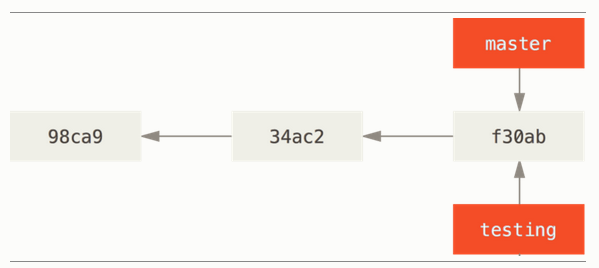


Figure1-4. 两个指向相同提交历史的分支

Git 通过HEAD确定当前在哪一个分支上工作， HEAD 是一个特殊指针。 请注意它和许多其它版本控制系统（如 Subversion 或 CVS）里的 HEAD 概念完全不同。 在 Git 中，它是一个指针，指向当前所在的本地分支（译注：将 HEAD 想象为当前分支的别名）。 在本例中，你仍然在 master 分支上。 因为 git branch 命令仅仅 创建 一个新分支，并不会自动切换到新分支中去。

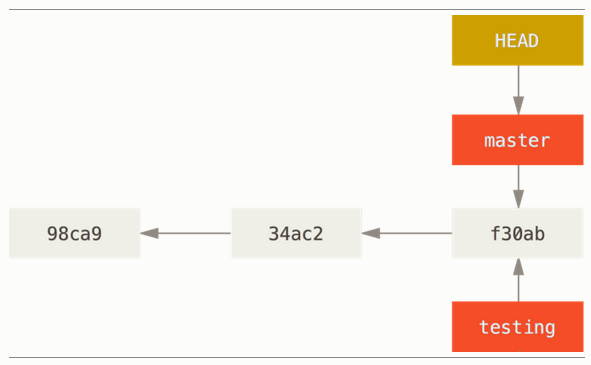


Figure 1-5. HEAD 指向当前所在的分支

可以简单地使用 git log 命令查看各个分支当前所指的对象。 提供这一功能的参数是 --decorate。

$ git log --oneline --decorate

f30ab (HEAD, master, testing) add feature #32 - ability to add new

34ac2 fixed bug #1328 - stack overflow under certain conditions

98ca9 initial commit of my project

### 1.6.4分支切换

要切换到一个已存在的分支，你需要使用 git checkout 命令。 我们现在切换到新创建的 testing 分支去：

$ git checkout testing

这样 HEAD 就指向 testing 分支了。

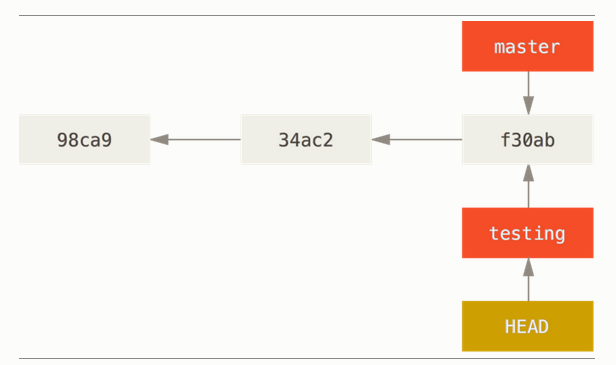


Figure 1-6. HEAD 指向当前所在的分支

这种切换分支有什么实质意义呢？ 现在不妨再提交一次：

$ vim test.rb

$ git commit -a -m 'made a change'

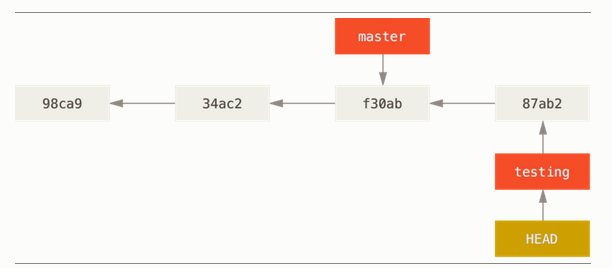


Figure 1-7. HEAD 分支随着提交操作自动向前移动

如图所示，你的 testing 分支向前移动了，但是 master 分支却没有，它仍然指向运行 git checkout 时所指的对象。 这就有意思了，现在我们切换回 master 分支看看：

$ git checkout master

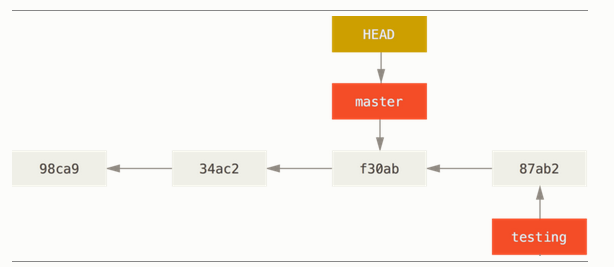


Figure 1-8. 检出时 HEAD 随之移动

这条命令做了两件事。 一是使 HEAD 指回 master 分支，二是将工作目录恢复成 master 分支所指向的快照内容。 也就是说，你现在做修改的话，项目将始于一个较旧的版本。 本质上来讲，这就是忽略 testing 分支所做的修改，以便于向另一个方向进行开发。

**分支切换会改变你工作目录中的文件**

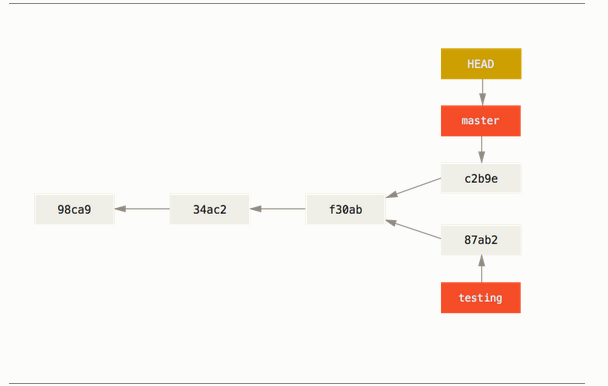
在切换分支时，一定要注意你工作目录里的文件会被改变。 如果是切换到一个较旧的分支，你的工作目录会恢复到该分支最后一次提交时的样子。 如果 Git 不能干净利落地完成这个任务，它将禁止切换分支。

如果你在当前分支进行修改和提交：

$ vim test.rb

$ git commit -a -m 'made other changes'

现在，这个项目的提交历史已经产生了分叉（参见下图 Figure1-9）。 因为刚才你创建了一个新分支，并切换过去进行了一些工作，随后又切换回 master 分支进行了另外一些工作。 上述两次改动针对的是不同分支：你可以在不同分支间不断地来回切换和工作，并在时机成熟时将它们合并起来。 而所有这些工作，你需要的命令只有 branch、checkout 和 commit。



你可以简单地使用 git log 命令查看分叉历史。 运行 git log --oneline --decorate --graph --all ，它会输出你的提交历史、各个分支的指向以及项目的分支分叉情况。

$ git log --oneline --decorate --graph --all

\* c2b9e (HEAD, master) made other changes

| \* 87ab2 (testing) made a change

|/

\* f30ab add feature #32 - ability to add new formats to the

\* 34ac2 fixed bug #1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

由于 Git 的分支实质上仅是包含所指对象校验和（长度为 40 的 SHA-1 值字符串）的文件，所以它的创建和销毁都异常高效。 创建一个新分支就像是往一个文件中写入 41 个字节（40 个字符和 1 个换行符），如此的简单能不快吗？

这与过去大多数版本控制系统形成了鲜明的对比，它们在创建分支时，将所有的项目文件都复制一遍，并保存到一个特定的目录。 完成这样繁琐的过程通常需要好几秒钟，有时甚至需要好几分钟。所需时间的长短，完全取决于项目的规模。而在 Git 中，任何规模的项目都能在瞬间创建新分支。 同时，由于每次提交都会记录父对象，所以寻找恰当的合并基础（译注：即共同祖先）也是同样的简单和高效。 这些高效的特性使得 Git 鼓励开发人员频繁地创建和使用分支。

### 1.6.5 分支创建和合并（实际例子）

让我们来看一个简单的分支新建与分支合并的例子，实际工作中你可能会用到类似的工作流。 你将经历如下步骤：

1. 开发某个网站。
2. 为实现某个新的需求，创建一个分支。
3. 在这个分支上开展工作。

正在此时，你突然接到一个电话说有个很严重的问题需要紧急修补。 你将按照如下方式来处理：

1. 切换到你的线上分支（production branch）。
2. 为这个紧急任务新建一个分支，并在其中修复它。
3. 在测试通过之后，切换回线上分支，然后合并这个修补分支，最后将改动推送到线上分支。
4. 切换回你最初工作的分支上，继续工作。

#### 1.6.5.1新建分支

首先，我们假设你正在你的项目上工作，并且已经有一些提交。

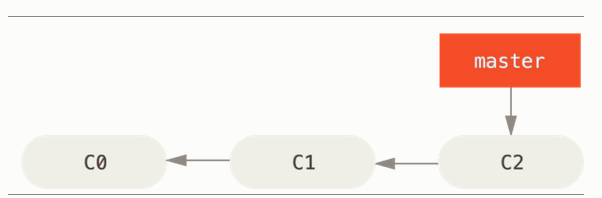
****

Figure 1-10. 一个简单提交历史

现在，你已经决定要解决你的公司使用的问题追踪系统中的 #53 问题。 想要新建一个分支并同时切换到那个分支上，你可以运行一个带有 -b 参数的 git checkout 命令：

$ git checkout -b iss53

Switched to a new branch "iss53"

它是下面两条命令的简写：

$ git branch iss53

$ git checkout iss53

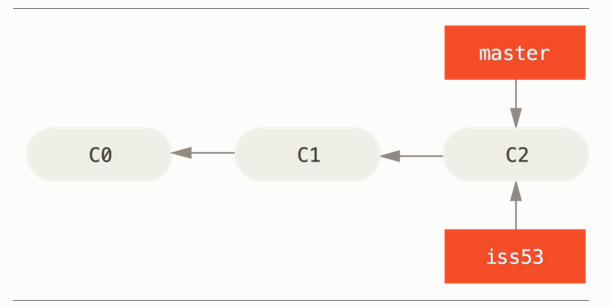
****

Figure 1-11. 创建一个新分支指针

你继续在 #53 问题上工作，并且做了一些提交。 在此过程中，iss53 分支在不断的向前推进，因为你已经检出到该分支（也就是说，你的 HEAD 指针指向了 iss53 分支）

$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'

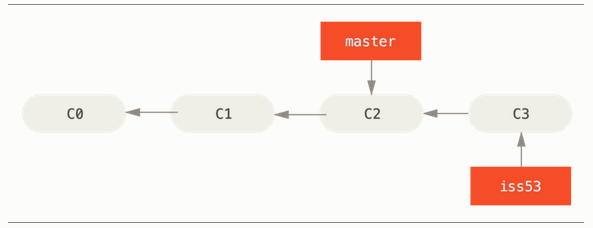
****

Figure 1-12. iss53 分支随着工作的进展向前推进

现在你接到那个电话，有个紧急问题等待你来解决。 有了 Git 的帮助，你不必把这个紧急问题和 iss53 的修改混在一起，你也不需要花大力气来还原关于 53# 问题的修改，然后再添加关于这个紧急问题的修改，最后将这个修改提交到线上分支。 你所要做的仅仅是切换回 master 分支。

但是，在你这么做之前，要留意你的工作目录和暂存区里那些还没有被提交的修改，它可能会和你即将检出的分支产生冲突从而阻止 Git 切换到该分支。 最好的方法是，在你切换分支之前，保持好一个干净的状态。 有一些方法可以绕过这个问题（即，保存进度（stashing） 和 修补提交（commit amending））。 现在，我们假设你已经把你的修改全部提交了，这时你可以切换回 master 分支了：

$ git checkout master

Switched to branch 'master'

这个时候，你的工作目录和你在开始 #53 问题之前一模一样，现在你可以专心修复紧急问题了。 请牢记：当你切换分支的时候，Git 会重置你的工作目录，使其看起来像回到了你在那个分支上最后一次提交的样子。 Git 会自动添加、删除、修改文件以确保此时你的工作目录和这个分支最后一次提交时的样子一模一样。

接下来，你要修复这个紧急问题。 让我们建立一个针对该紧急问题的分支（hotfix branch），在该分支上工作直到问题解决：

$ git checkout -b hotfix

Switched to a new branch 'hotfix'

$ vim index.html

$ git commit -a -m 'fixed the broken email address'

[hotfix 1fb7853] fixed the broken email address

1 file changed, 2 insertions(+)

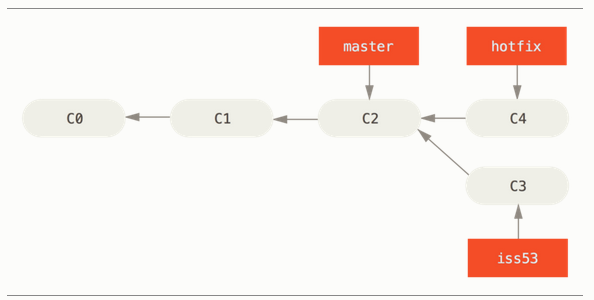


Figure 1-13. 基于 master 分支的紧急问题分支 hotfix branch

你可以运行你的测试，确保你的修改是正确的，然后将其合并回你的 master 分支来部署到线上。 你可以使用 git merge 命令来达到上述目的：

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)

在合并的时候，你应该注意到了"快进（fast-forward）"这个词。 由于当前 master 分支所指向的提交是你当前提交（有关 hotfix 的提交）的直接上游，所以 Git 只是简单的将指针向前移动。 换句话说，当你试图合并两个分支时，如果顺着一个分支走下去能够到达另一个分支，那么 Git 在合并两者的时候，只会简单的将指针向前推进（指针右移），因为这种情况下的合并操作没有需要解决的分歧——这就叫做 “快进（fast-forward）”。

现在，最新的修改已经在 master 分支所指向的提交快照中，你可以着手发布该修复了。

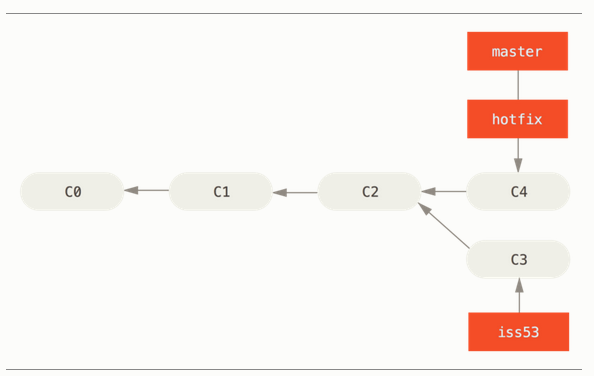


Figure 1-14. master 被快进到 hotfix

关于这个紧急问题的解决方案发布之后，你准备回到被打断之前时的工作中。 然而，你应该先删除 hotfix 分支，因为你已经不再需要它了 —— master 分支已经指向了同一个位置。 你可以使用带 -d 选项的 git branch 命令来删除分支：

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

现在你可以切换回你正在工作的分支继续你的工作，也就是针对 #53 问题的那个分支（iss53 分支）。

$ git checkout iss53

Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'finished the new footer [issue 53]'

[iss53 ad82d7a] finished the new footer [issue 53]

1 file changed, 1 insertion(+)

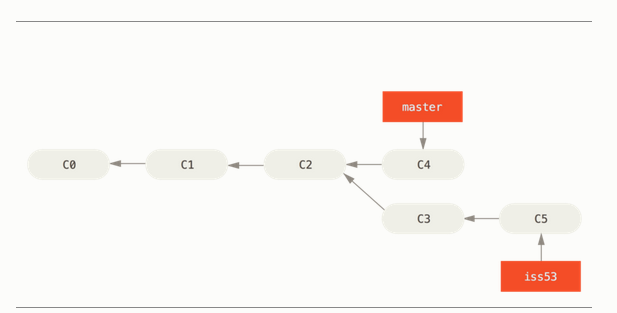


Figure 3-15. 继续在 iss53 分支上的工作

你在 hotfix 分支上所做的工作并没有包含到 iss53 分支中。 如果你需要拉取 hotfix 所做的修改，你可以使用 git merge master 命令将 master 分支合并入 iss53 分支，或者你也可以等到 iss53 分支完成其使命，再将其合并回 master 分支。

#### 1.6.5.2分支的合并

假设你已经修正了 #53 问题，并且打算将你的工作合并入 master 分支。 为此，你需要合并 iss53 分支到 master 分支，这和之前你合并 hotfix 分支所做的工作差不多。 你只需要检出到你想合并入的分支，然后运行 git merge 命令：

$ git checkout master

Switched to branch 'master'

$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

这和你之前合并 hotfix 分支的时候看起来有一点不一样。 在这种情况下，你的开发历史从一个更早的地方开始分叉开来（diverged）。 因为，master 分支所在提交并不是 iss53 分支所在提交的直接祖先，Git 不得不做一些额外的工作。 出现这种情况的时候，Git 会使用两个分支的末端所指的快照（C4 和 C5）以及这两个分支的工作祖先（C2），做一个简单的三方合并。

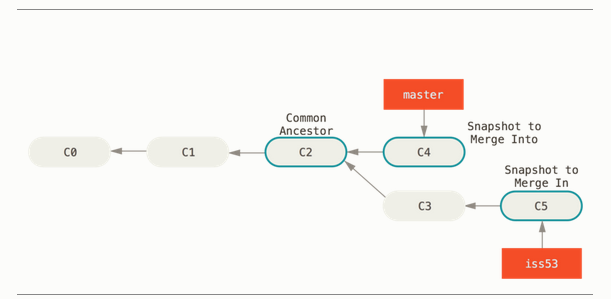


Figure 1-16. 一次典型合并中所用到的三个快照

和之前将分支指针向前推进所不同的是，Git 将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。 这个被称作一次合并提交，它的特别之处在于他有不止一个父提交。

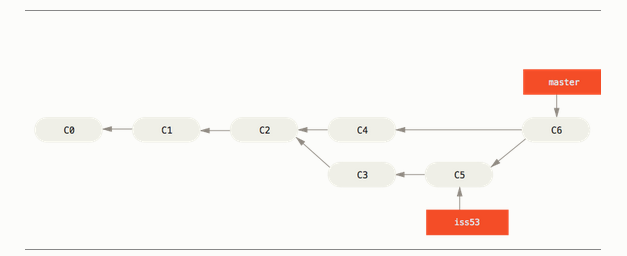


Figure 1-17. 一个合并提交

需要指出的是，Git 会自行决定选取哪一个提交作为最优的共同祖先，并以此作为合并的基础；这和更加古老的 CVS 系统或者 Subversion （1.5 版本之前）不同，在这些古老的版本管理系统中，用户需要自己选择最佳的合并基础。 Git 的这个优势使其在合并操作上比其他系统要简单很多。

既然你的修改已经合并进来了，你已经不再需要 iss53 分支了。 现在你可以在任务追踪系统中关闭此项任务，并删除这个分支。

$ git branch -d iss53

#### 1.6.5.3 遇到冲突时的分支合并

有时候合并操作不会如此顺利。 如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们。 如果你对 #53 问题的修改和有关 hotfix 的修改都涉及到同一个文件的同一处，在合并它们的时候就会产生合并冲突：

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

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

此时 Git 做了合并，但是没有自动地创建一个新的合并提交。 Git 会暂停下来，等待你去解决合并产生的冲突。 你可以在合并冲突后的任意时刻使用 git status 命令来查看那些因包含合并冲突而处于未合并（unmerged）状态的文件：

$ git status

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")

任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。 Git 会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。 出现冲突的文件会包含一些特殊区段，看起来像下面这个样子：

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

这表示 HEAD 所指示的版本（也就是你的 master 分支所在的位置，因为你在运行 merge 命令的时候已经检出到了这个分支）在这个区段的上半部分（======= 的上半部分），而 iss53 分支所指示的版本在 ======= 的下半部分。 为了解决冲突，你必须选择使用由 ======= 分割的两部分中的一个，或者你也可以自行合并这些内容。 例如，你可以通过把这段内容换成下面的样子来解决冲突：

<div id="footer">

please contact us at email.support@github.com

</div>

上述的冲突解决方案仅保留了其中一个分支的修改，并且 <<<<<<< , ======= , 和 >>>>>>> 这些行被完全删除了。 在你解决了所有文件里的冲突之后，对每个文件使用 git add 命令来将其标记为冲突已解决。 一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决。

你可以再次运行 git status 来确认所有的合并冲突都已被解决：

$ git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: index.html

如果你对结果感到满意，并且确定之前有冲突的的文件都已经暂存了，这时你可以输入 git commit 来完成合并提交。 默认情况下提交信息看起来像下面这个样子：

Merge branch 'iss53'

Conflicts:

index.html

#

# It looks like you may be committing a merge.

# If this is not correct, please remove the file

# .git/MERGE\_HEAD

# and try again.

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

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

# On branch master

# All conflicts fixed but you are still merging.

#

# Changes to be committed:

# modified: index.html

#

如果你觉得上述的信息不够充分，不能完全体现分支合并的过程，你可以修改上述信息，添加一些细节给未来检视这个合并的读者一些帮助，告诉他们你是如何解决合并冲突的，以及理由是什么。

### 1.6.6 分支的管理

现在已经创建、合并、删除了一些分支，让我们看看一些常用的分支管理工具。

git branch 命令不只是可以创建与删除分支。 如果不加任何参数运行它，会得到当前所有分支的一个列表：

$ git branch

iss53

\* master

testing

注意 master 分支前的 \* 字符：它代表现在检出的那一个分支（也就是说，当前 HEAD 指针所指向的分支）。 这意味着如果在这时候提交，master 分支将会随着新的工作向前移动。 如果需要查看每一个分支的最后一次提交，可以运行 git branch -v 命令：

$ git branch -v

iss53 93b412c fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 add scott to the author list in the readmes

--merged 与 --no-merged 这两个有用的选项可以过滤这个列表中已经合并或尚未合并到当前分支的分支。 如果要查看哪些分支已经合并到当前分支，可以运行 git branch --merged：

$ git branch --merged

iss53

\* master

因为之前已经合并了 iss53 分支，所以现在看到它在列表中。 在这个列表中分支名字前没有 \* 号的分支通常可以使用 git branch -d 删除掉；你已经将它们的工作整合到了另一个分支，所以并不会失去任何东西。

查看所有包含未合并工作的分支，可以运行 git branch --no-merged：

$ git branch --no-merged

testing

这里显示了其他分支。 因为它包含了还未合并的工作，尝试使用 git branch -d 命令删除它时会失败：

$ git branch -d testing

error: The branch 'testing' is not fully merged.

If you are sure you want to delete it, run 'git branch -D testing'.

如果真的想要删除分支并丢掉那些工作，如同帮助信息里所指出的，可以使用 -D 选项强制删除它。

### 1.6.7 分支开发流程

由于分支管理的便捷，已经衍生出一些典型的工作模式，我们可以根据项目实际情况选择一种实用。

#### 1.6.7.1 长期分支模式

因为 Git 使用简单的三方合并，所以就算在一段较长的时间内，反复把一个分支合并入另一个分支，也不是什么难事。 也就是说，在整个项目开发周期的不同阶段，你可以同时拥有多个开放的分支；你可以定期地把某些特性分支合并入其他分支中。

许多使用 Git 的开发者都喜欢使用这种方式来工作，比如只在 master 分支上保留完全稳定的代码——有可能仅仅是已经发布或即将发布的代码。 他们还有一些名为 develop 或者 testing 的平行分支，被用来做后续开发或者测试稳定性——这些分支不必保持绝对稳定，但是一旦达到稳定状态，它们就可以被合并入 master 分支了。 这样，在确保这些已完成的特性分支（短期分支，比如之前的 iss53 分支）能够通过所有测试，并且不会引入更多 bug 之后，就可以合并入主干分支中，等待下一次的发布。

事实上我们刚才讨论的，是随着你的提交而不断右移的指针。 稳定分支的指针总是在提交历史中落后一大截，而前沿分支的指针往往比较靠前。

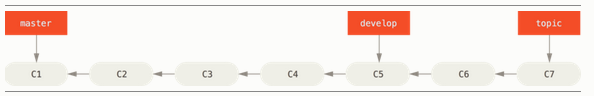


Figure 1-18. 渐进稳定分支的线性图

通常把他们想象成流水线（work silos）可能更好理解一点，那些经过测试考验的提交会被遴选到更加稳定的流水线上去。

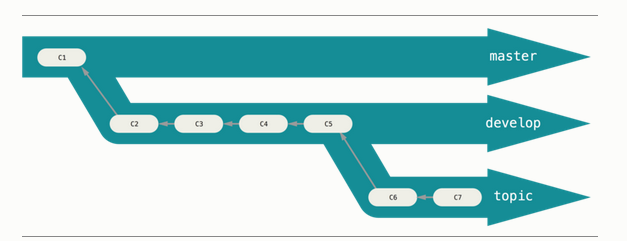


Figure 1-19. 渐进稳定分支的流水线（“silo”）视图

可以用这种方法维护不同层次的稳定性。 一些大型项目还有一个 proposed（建议） 或 pu: proposed updates（建议更新）分支，它可能因包含一些不成熟的内容而不能进入 next 或者 master 分支。 这么做的目的是使你的分支具有不同级别的稳定性；当它们具有一定程度的稳定性后，再把它们合并入具有更高级别稳定性的分支中。 再次强调一下，使用多个长期分支的方法并非必要，但是这么做通常很有帮助，尤其是当你在一个非常庞大或者复杂的项目中工作时。

#### 1.6.7.2 特性分支

特性分支对任何规模的项目都适用。 特性分支是一种短期分支，它被用来实现单一特性或其相关工作。 也许你从来没有在其他的版本控制系统（VCS）上这么做过，因为在那些版本控制系统中创建和合并分支通常很费劲。 然而，在 Git 中一天之内多次创建、使用、合并、删除分支都很常见。

我们已经在上一节中你创建的 iss53 和 hotfix 特性分支中看到过这种用法。 你在上一节用到的特性分支（iss53 和 hotfix 分支）中提交了一些更新，并且在它们合并入主干分支之后，你又删除了它们。 这项技术能使你快速并且完整地进行上下文切换（context-switch）——因为你的工作被分散到不同的流水线中，在不同的流水线中每个分支都仅与其目标特性相关，因此，在做代码审查之类的工作的时候就能更加容易地看出你做了哪些改动。 你可以把做出的改动在特性分支中保留几分钟、几天甚至几个月，等它们成熟之后再合并，而不用在乎它们建立的顺序或工作进度。

考虑这样一个例子，你在 master 分支上工作到 C1，这时为了解决一个问题而新建 iss91 分支，在 iss91 分支上工作到 C4，然而对于那个问题你又有了新的想法，于是你再新建一个 iss91v2 分支试图用另一种方法解决那个问题，接着你回到 master 分支工作了一会儿，你又冒出了一个不太确定的想法，你便在 C10 的时候新建一个 dumbidea 分支，并在上面做些实验。 你的提交历史看起来像下面这个样子：

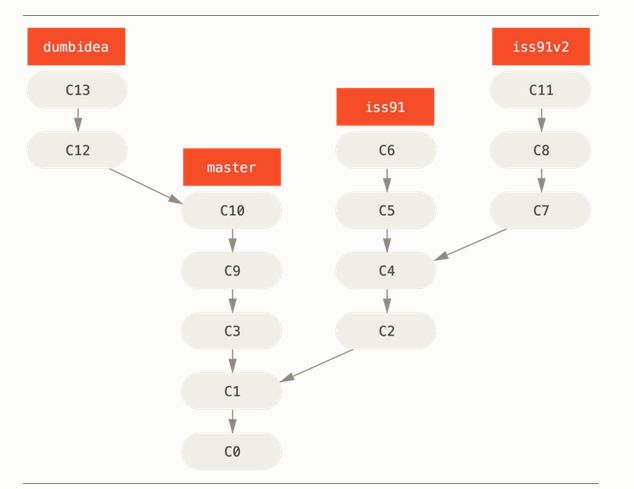


Figure 1-20. 拥有多个特性分支的提交历史

现在，我们假设两件事情：你决定使用第二个方案来解决那个问题，即使用在 iss91v2 分支中方案；另外，你将 dumbidea 分支拿给你的同事看过之后，结果发现这是个惊人之举。 这时你可以抛弃 iss91 分支（即丢弃 C5 和 C6 提交），然后把另外两个分支合并入主干分支。 最终你的提交历史看起来像下面这个样子：

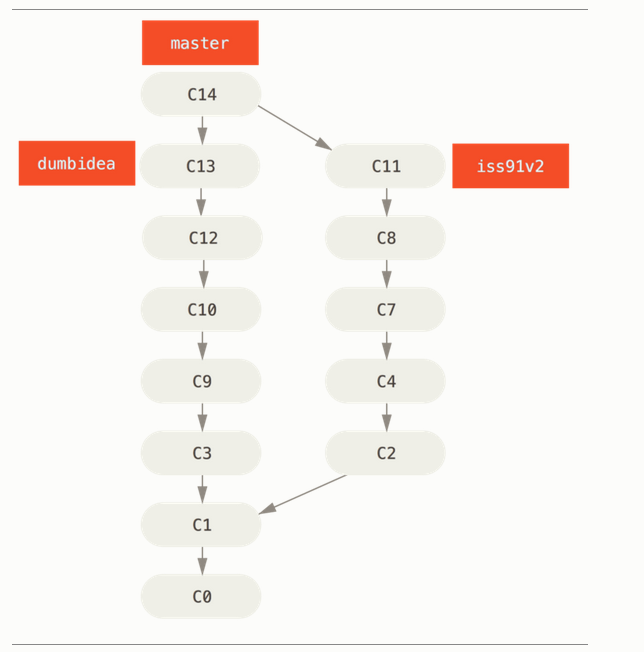


Figure 1-21. 合并了 dumbidea 和 iss91v2 分支之后的提交历史

### 1.6.8 远程分支

#### 1.6.8.1 远程分支

远程引用是对远程仓库的引用（指针），包括分支、标签等等。 你可以通过 git ls-remote (remote) 来显式地获得远程引用的完整列表，或者通过 git remote show (remote) 获得远程分支的更多信息。 然而，一个更常见的做法是利用远程跟踪分支。

远程跟踪分支是远程分支状态的引用。 它们是你不能移动的本地引用，当你做任何网络通信操作时，它们会自动移动。 远程跟踪分支像是你上次连接到远程仓库时，那些分支所处状态的书签。

它们以 (remote)/(branch) 形式命名。 例如，如果你想要看你最后一次与远程仓库 origin 通信时 master 分支的状态，你可以查看 origin/master 分支。 你与同事合作解决一个问题并且他们推送了一个 iss53 分支，你可能有自己的本地 iss53 分支；但是在服务器上的分支会指向 origin/iss53 的提交。

这可能有一点儿难以理解，让我们来看一个例子。 假设你的网络里有一个在 git.ourcompany.com 的 Git 服务器。 如果你从这里克隆，Git 的 clone 命令会为你自动将其命名为 origin，拉取它的所有数据，创建一个指向它的 master 分支的指针，并且在本地将其命名为 origin/master。 Git 也会给你一个与 origin 的 master 分支在指向同一个地方的本地 master 分支，这样你就有工作的基础。

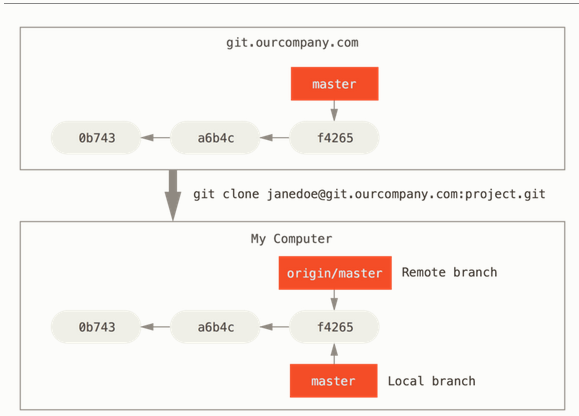


Figure 1-22. 克隆之后的服务器与本地仓库

如果你在本地的 master 分支做了一些工作，然而在同一时间，其他人推送提交到 git.ourcompany.com 并更新了它的 master 分支，那么你的提交历史将向不同的方向前进。 也许，只要你不与 origin 服务器连接，你的 origin/master 指针就不会移动。

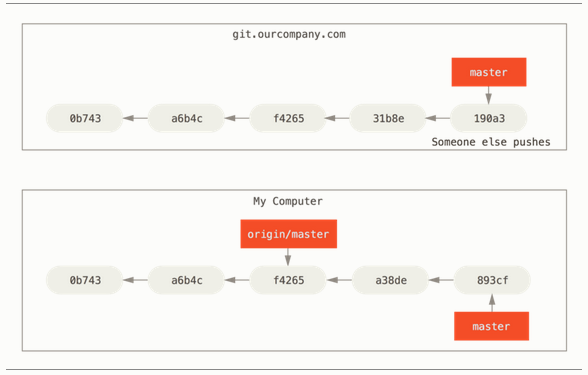


Figure 1-23. 本地与远程的工作可以分叉

如果要同步你的工作，运行 git fetch origin 命令。 这个命令查找 “origin” 是哪一个服务器（在本例中，它是 git.ourcompany.com），从中抓取本地没有的数据，并且更新本地数据库，移动 origin/master 指针指向新的、更新后的位置。

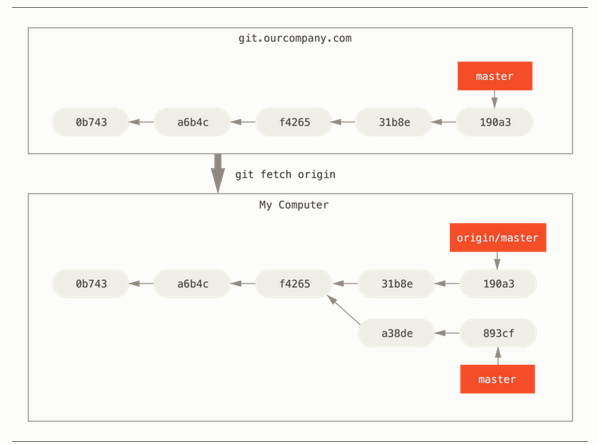


Figure 1-24. git fetch 更新你的远程仓库引用

#### 1.6.8.2 推送

当你想要公开分享一个分支时，需要将其推送到有写入权限的远程仓库上。 本地的分支并不会自动与远程仓库同步 - 你必须显式地推送想要分享的分支。 这样，你就可以把不愿意分享的内容放到私人分支上，而将需要和别人协作的内容推送到公开分支。

如果希望和别人一起在名为 serverfix 的分支上工作，你可以像推送第一个分支那样推送它。 运行 git push (remote) (branch):

$ git push origin serverfix

Counting objects: 24, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.

Total 24 (delta 2), reused 0 (delta 0)

To https://github.com/schacon/simplegit

\* [new branch] serverfix -> serverfix

也可以运行 git push origin serverfix:serverfix，它会做同样的事 - 相当于它说，“推送本地的 serverfix 分支，将其作为远程仓库的 serverfix 分支” 可以通过这种格式来推送本地分支到一个命名不相同的远程分支。 如果并不想让远程仓库上的分支叫做 serverfix，可以运行 git push origin serverfix:awesomebranch 来将本地的 serverfix 分支推送到远程仓库上的 awesomebranch 分支。

下一次其他协作者从服务器上抓取数据时，他们会在本地生成一个远程分支 origin/serverfix，指向服务器的 serverfix 分支的引用：

$ git fetch origin

remote: Counting objects: 7, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 3 (delta 0)

Unpacking objects: 100% (3/3), done.

From https://github.com/schacon/simplegit

\* [new branch] serverfix -> origin/serverfix

要特别注意的一点是当抓取到新的远程跟踪分支时，本地不会自动生成一份可编辑的副本（拷贝）。 换一句话说，这种情况下，不会有一个新的 serverfix 分支 - 只有一个不可以修改的 origin/serverfix 指针。

可以运行 git merge origin/serverfix 将这些工作合并到当前所在的分支。 如果想要在自己的 serverfix 分支上工作，可以将其建立在远程跟踪分支之上：

$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

这会给你一个用于工作的本地分支，并且起点位于 origin/serverfix。

#### 1.6.8.3 跟踪分支

从一个远程跟踪分支检出一个本地分支会自动创建一个叫做 “跟踪分支”（有时候也叫做 “上游分支”）。 跟踪分支是与远程分支有直接关系的本地分支。 如果在一个跟踪分支上输入 git pull，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。

当克隆一个仓库时，它通常会自动地创建一个跟踪 origin/master 的 master 分支。 然而，如果你愿意的话可以设置其他的跟踪分支 - 其他远程仓库上的跟踪分支，或者不跟踪 master 分支。 最简单的就是之前看到的例子，运行 git checkout -b [branch] [remotename]/[branch]。 这是一个十分常用的操作所以 Git 提供了 --track 快捷方式：

$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch serverfix from origin.

Switched to a new branch 'serverfix'

如果想要将本地分支与远程分支设置为不同名字，你可以轻松地增加一个不同名字的本地分支的上一个命令：

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch serverfix from origin.

Switched to a new branch 'sf'

现在，本地分支 sf 会自动从 origin/serverfix 拉取。

如果想要查看设置的所有跟踪分支，可以使用 git branch 的 -vv 选项。 这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。

$ git branch -vv

iss53 7e424c3 [origin/iss53: ahead 2] forgot the brackets

master 1ae2a45 [origin/master] deploying index fix

\* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it

testing 5ea463a trying something new

这里可以看到 iss53 分支正在跟踪 origin/iss53 并且 “ahead” 是 2，意味着本地有两个提交还没有推送到服务器上。 也能看到 master 分支正在跟踪 origin/master 分支并且是最新的。 接下来可以看到 serverfix 分支正在跟踪 teamone 服务器上的 server-fix-good 分支并且领先 2 落后 1，意味着服务器上有一次提交还没有合并入同时本地有三次提交还没有推送。 最后看到 testing 分支并没有跟踪任何远程分支。

需要重点注意的一点是这些数字的值来自于你从每个服务器上最后一次抓取的数据。 这个命令并没有连接服务器，它只会告诉你关于本地缓存的服务器数据。 如果想要统计最新的领先与落后数字，需要在运行此命令前抓取所有的远程仓库。 可以像这样做：$ git fetch --all; git branch -vv

#### 1.6.8.4 拉取

当 git fetch 命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。 它只会获取数据然后让你自己合并。 然而，有一个命令叫作 git pull 在大多数情况下它的含义是一个 git fetch 紧接着一个 git merge 命令。 如果有一个像之前章节中演示的设置好的跟踪分支，不管它是显式地设置还是通过 clone 或 checkout 命令为你创建的，git pull 都会查找当前分支所跟踪的服务器与分支，从服务器上抓取数据然后尝试合并入那个远程分支。

由于 git pull 的魔法经常令人困惑所以通常单独显式地使用 fetch 与 merge 命令会更好一些。

#### 1.6.8.5 删除远程分支

假设你已经通过远程分支做完所有的工作了 - 也就是说你和你的协作者已经完成了一个特性并且将其合并到了远程仓库的 master 分支（或任何其他稳定代码分支）。 可以运行带有 --delete 选项的 git push 命令来删除一个远程分支。 如果想要从服务器上删除 serverfix 分支，运行下面的命令：

$ git push origin --delete serverfix

To https://github.com/schacon/simplegit

- [deleted] serverfix

基本上这个命令做的只是从服务器上移除这个指针。 Git 服务器通常会保留数据一段时间直到垃圾回收运行，所以如果不小心删除掉了，通常是很容易恢复的。

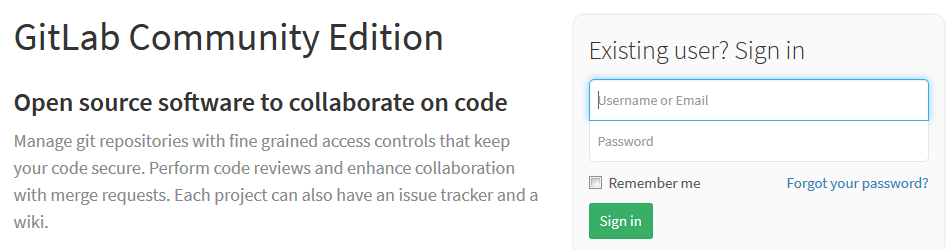
# Git的安装

## 2.1 Git服务器端安装和配置

1）GitLab服务器端已经在公司服务器安装好，访问地址如下：

<http://192.168.1.212:10080/>

2）目前已经为技术中心\_开发部的同事创建了访问账号



3）目前在服务器端已创建了项目：ypt，并把开发部的同事加入到该项目

访问地址：http://192.168.1.212:10080/root/ypt.git

## 2.2 Git客户端安装：

1）安装msysgit（Git客户端-命令行界面）

下载地址：<http://msysgit.github.io/>

根据自己操作系统版本情况下载合适的客户端版本，我这里有下载好的windows 64位操作系统的客户端版本，需要的联系我

## 2）安装TortoiseGit（图形操作界面）和中文补丁包

我这里有下载好的安装程序，有需要的联系我



# Git客户端设置

## 3.1 访问账号设置

1）访问Git命令行客户端：

在安装好的程序组里选择Git-🡪git Bash即可访问git命令行工具，如下图：



## 3.2设置用户名和电子邮箱

个人的用户名和电子邮件地址很重要，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

在Git Bash窗口输入一下命令设置你的用户名和电子邮箱地址：

git config –global user.name “zhangyi”

git config –global user.email 252782519@qq.com

查看已有的配置信息：

Git config –list

## 3.3设置账户的SSH key：

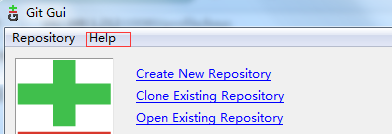
Git使用SSH加密传输更新提交数据，确保每次的更新提交不被非法篡改

1）在git bash窗口输入以下命令，并输入3个回车：

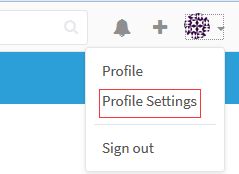
ssh-keygen -t rsa -C “252782519@qq.com”

记得把邮箱换成自己的

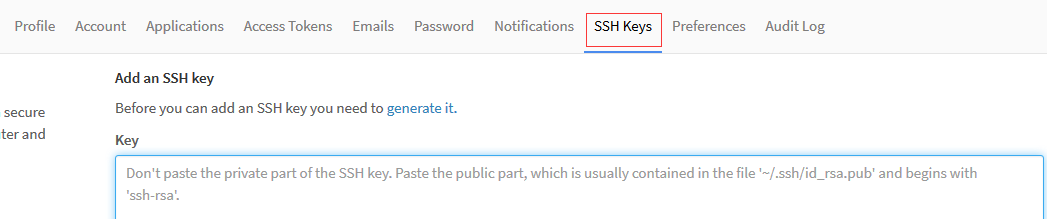
2）打开“Git GUI”（Git安装后的另一部分），点击菜单的help－ssh，然后copy当前的“key”



3）用自己账号登录Git服务端，点击“profile setting”：

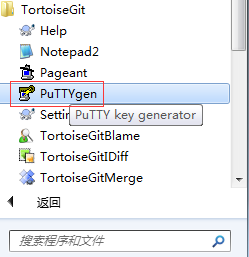


4）在“ssh key”标签页把刚生成的公钥粘贴过来并生称自己的sshkey:

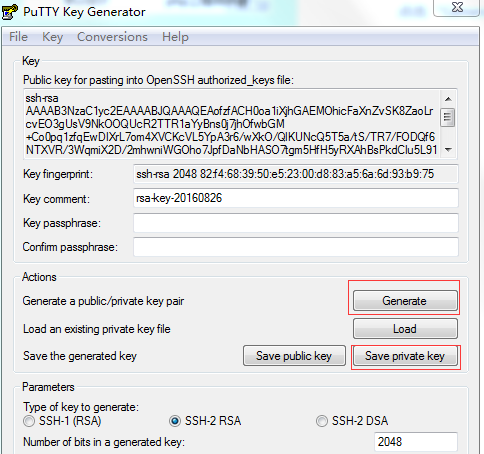


## 3.4 生成本地密钥文件

在Windows开始菜单中点“TortoiseGit”程序组，然后选择“puTTY gen”，如下图：



在“key generator“窗口生成并保存密钥文件，如下图：



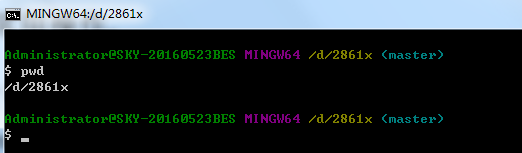
## 3.5访问Git 命令行工具

Git系统可以通过命令行工具来访问，也可以通过GUI工具来访问，本文主要介绍命令行工具

1）选择或创建一个工作目录

2）在该目录点击鼠标右键，在弹出菜单中选择”Git Bash Here”菜单项

Git会打开Bash命令行窗口，并把当前目录设置为你的工作目录



# Git操作演示

## 4.1从当前目录初始化

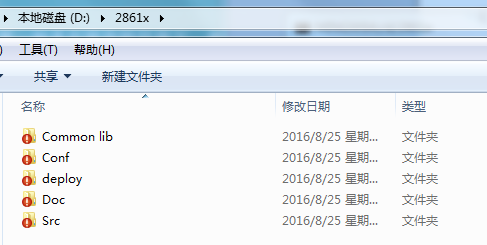
1) 要对现有的某个项目开始用Git进行管理，只需要在此项目所在的目录执行命令 Git init

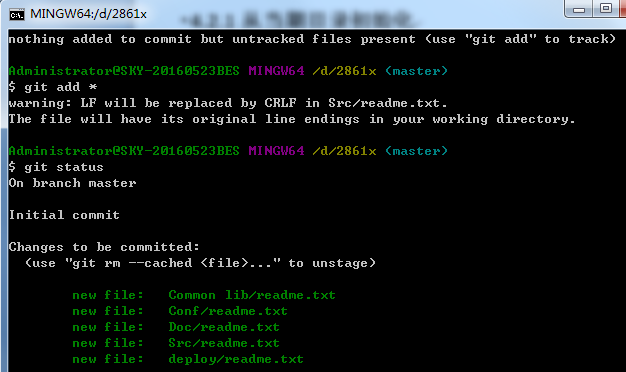
初始化后，在当前目录下会出现一个名为.git的隐藏子目录，所有Git需要的数据和资源都存放在这个目录中。

1. 使用 git add 命令把当前目录下的文件加入版本控制：

git add \* 把全部文件和目录加入版本控制

4) 使用git status 查看文件的状态





1. 使用git commit命令提交本地版本库：

git commit –m “initial project version”

或者使用GUI提交：鼠标右键单击项目目录，在弹出菜单中选”Git提交”

在提交界面中填写日志信息，设置提交日期和作者信息，然后点击确定即可，如下图



提交后，文件目录图标发生改变，如下图：



## 4.2 从现有仓库克隆

如果想把已有项目的git仓库复制一份出来，需要用到Git clone命令

Git clone <http://localhost:10080/root/ypt.git>

或者使用GUI来操作

鼠标右键你的工作目录，在弹出菜单中选择“Git克隆”即可

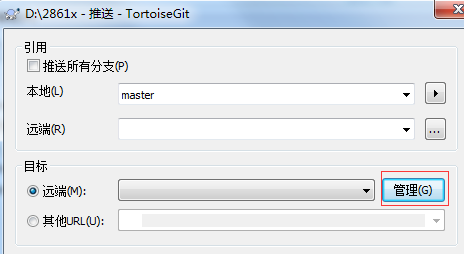
## 4.3 推送本地库到服务器

在把项目文件提交到本地版本库后，就可以吧本地库推送到服务器

用鼠标右键点击项目目录，在弹出菜单中选“推送”，如下图：

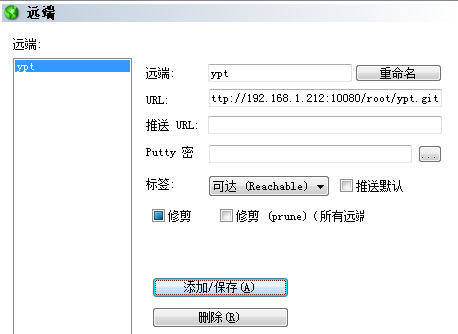


在弹出窗口选择“管理”，如下图：

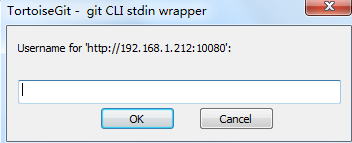


在弹出窗口中填写远端服务器Git项目的访问地址，如下图：

项目地址为：http://192.168.1.212:10080/root/ypt.git



关闭“管理”窗口，在“推送”窗口选择刚设置的“ypt”远端服务器进行推送



输入你的登录账号名和密码，即可完成推送，如下图：



查看服务器端，文件推送到服务器，如下图：

