
This book is available in English.

Full translation available in

azərbaycan dili,
български език,
Deutsch,
Español,

Français,
Ελληνικά,
日本語,
한국어,
Nederlands,
Русский,
Slovenščina,
Tagalog,
Українська
简体中文,

Partial translations available in

Čeština,
Македонски,
Polski,
Српски,
Ўзбекча,
繁體中文,

Translations started for

Беларуская,
فارسی,
Indonesian,
Italiano,
Bahasa Melayu,
Português (Brasil),
Português (Portugal),

Svenska,
Türkçe.

The source of this book is hosted on GitHub.
Patches, suggestions and comments are welcome.
Chapters ▾

1. **1. 起步**
    1. 1.1 关于版本控制
    2. 1.2 Git 简史
    3. 1.3 Git 是什么?
    4. 1.4 命令行
    5. 1.5 安装 Git
    6. 1.6 初次运行 Git 前的配置
    7. 1.7 获取帮助
    8. 1.8 总结
2. **2. Git 基础**
    1. 2.1 获取 Git 仓库
    2. 2.2 记录每次更新到仓库
    3. 2.3 查看提交历史
    4. 2.4 撤消操作
    5. 2.5 远程仓库的使用
    6. 2.6 打标签
    7. 2.7 Git 别名
    8. 2.8 总结
3. **3. Git 分支**

1. 3.1 分支简介
2. 3.2 分支的新建与合并
3. 3.3 分支管理
4. 3.4 分支开发工作流
5. 3.5 远程分支
6. 3.6 变基
7. 3.7 总结

4. 4. 服务器上的 Git

1. 4.1 协议
2. 4.2 在服务器上搭建 Git
3. 4.3 生成 SSH 公钥
4. 4.4 配置服务器
5. 4.5 Git 守护进程
6. 4.6 Smart HTTP
7. 4.7 GitWeb
8. 4.8 GitLab
9. 4.9 第三方托管的选择
10. 4.10 总结

5. 5. 分布式 Git

1. 5.1 分布式工作流程
2. 5.2 向一个项目贡献
3. 5.3 维护项目
4. 5.4 总结

1. 6. GitHub

1. 6.1 账户的创建和配置

2. 6.2 对项目做出贡献
3. 6.3 维护项目
4. 6.4 管理组织
5. 6.5 脚本 GitHub
6. 6.6 总结

2. **7. Git 工具**

1. 7.1 选择修订版本
2. 7.2 交互式暂存
3. 7.3 贮藏与清理
4. 7.4 签署工作
5. 7.5 搜索
6. 7.6 重写历史
7. 7.7 重置揭密
8. 7.8 高级合并
9. 7.9 Rerere
10. 7.10 使用 Git 调试
11. 7.11 子模块
12. 7.12 打包
13. 7.13 替换
14. 7.14 凭证存储
15. 7.15 总结

3. **8. 自定义 Git**

1. 8.1 配置 Git
2. 8.2 Git 属性
3. 8.3 Git 钩子
4. 8.4 使用强制策略的一个例子
5. 8.5 总结

4. 9. Git 与其他系统

    1. 9.1 作为客户端的 Git
    2. 9.2 迁移到 Git
    3. 9.3 总结

5. 10. Git 内部原理

    1. 10.1 底层命令与上层命令
    2. 10.2 Git 对象
    3. 10.3 Git 引用
    4. 10.4 包文件
    5. 10.5 引用规范
    6. 10.6 传输协议
    7. 10.7 维护与数据恢复
    8. 10.8 环境变量
    9. 10.9 总结

1. A1. 附录 A: 在其它环境中使用 Git

    1. A1.1 图形界面
    2. A1.2 Visual Studio 中的 Git
    3. A1.3 Visual Studio Code 中的 Git
    4. A1.4 Eclipse 中的 Git
    5. A1.5 IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine 中的 Git
    6. A1.6 Sublime Text 中的 Git
    7. A1.7 Bash 中的 Git
    8. A1.8 Zsh 中的 Git
    9. A1.9 Git 在 PowerShell 中使用 Git
    10. A1.10 总结

2. A2. 附录 B: 在你的应用中嵌入 Git

    1. A2.1 命令行 Git 方式
    2. A2.2 Libgit2
    3. A2.3 JGit
    4. A2.4 go-git
    5. A2.5 Dulwich

3. A3. 附录 C: Git 命令

    1. A3.1 设置与配置
    2. A3.2 获取与创建项目
    3. A3.3 快照基础
    4. A3.4 分支与合并
    5. A3.5 项目分享与更新
    6. A3.6 检查与比较
    7. A3.7 调试
    8. A3.8 补丁
    9. A3.9 邮件
    10. A3.10 外部系统
    11. A3.11 管理
    12. A3.12 底层命令

2nd Edition

10.3 Git 内部原理 – Git 引用

Git 引用

如果你对仓库中从一个提交（比如 `1a410e`）开始往前的历史感兴趣，那么可以运行 `git log 1a410e` 这样的命令来显示历史，不过你需要记得 `1a410e` 是你查看历史的起点提交。 如果我们有一个文件来保存 SHA-1 值，而该文件有一个简单的名字， 然后用这个名字指针来替代原始的 SHA-

1 值的话会更加简单。

在 Git 中，这种简单的名字被称为“引用（references，或简写为 refs）”。 你可以在 `.git/refs` 目录下找到这类含有 SHA-1 值的文件。 在目前的项目中，这个目录没有包含任何文件，但它包含了一个简单的目录结构：

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

若要创建一个新引用来帮助记忆最新提交所在的位置，从技术上讲我们只需简单地做如下操作：

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master
```

现在，你就可以在 Git 命令中使用这个刚创建的新引用来代替 SHA-1 值了：

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

我们不提倡直接编辑引用文件。 如果想更新某个引用，Git 提供了一个更加安全的命令 `update-ref` 来完成此事：

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

这基本就是 Git 分支的本质：一个指向某一系列提交之首的指针或引用。 若想在第二个提交上创建一个分支，可以这么做：

```
$ git update-ref refs/heads/test cac0ca
```

这个分支将只包含从第二个提交开始往前追溯的记录：

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

至此，我们的 Git 数据库从概念上看起来像这样：

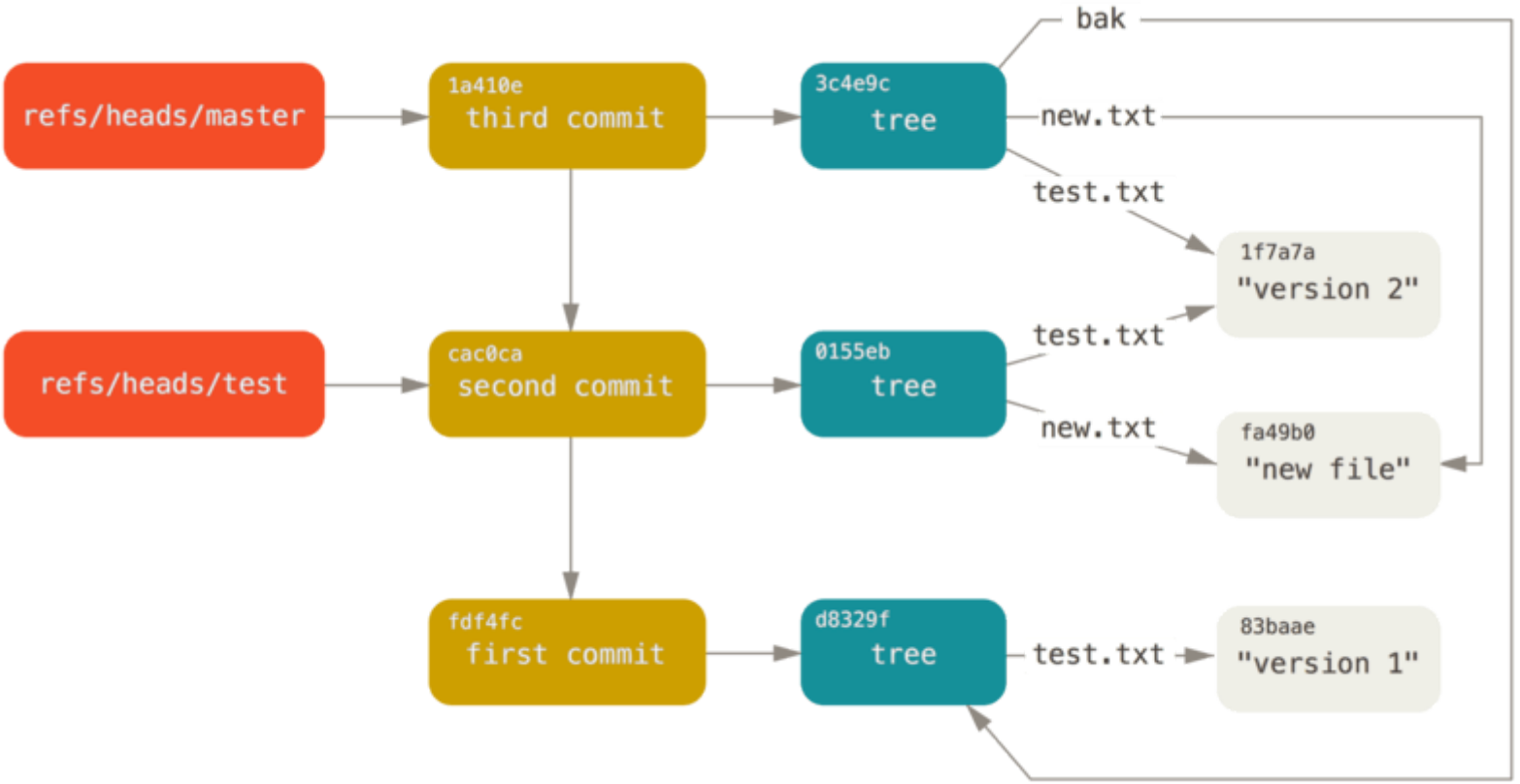

Figure 152. 包含分支引用的 Git 目录对象。

当运行类似于 `git branch <branch>` 这样的命令时，Git 实际上会运行 `update-ref` 命令， 取得当前所在分支最新提交对应的 SHA-1 值，并将其加入你想要创建的任何新引用中。

HEAD 引用

现在的问题是，当你执行 `git branch <branch>` 时，Git 如何知道最新提交的 SHA-1 值呢？ 答案是 HEAD 文件。

HEAD 文件通常是一个符号引用（symbolic reference），指向目前所在的分支。 所谓符号引用，表示它是一个指向其他引用的指针。

然而在某些罕见的情况下，HEAD 文件可能会包含一个 git 对象的 SHA-1 值。 当你在检出一个标签、提交或远程分支，让你的仓库变成 “分离 HEAD”状态时，就会出现这种情况。

如果查看 HEAD 文件的内容，通常我们看到类似这样的内容：

```
$ cat .git/HEAD
ref: refs/heads/master
```

如果执行 `git checkout test`，Git 会像这样更新 HEAD 文件：

```
$ cat .git/HEAD
ref: refs/heads/test
```

当我们执行 `git commit` 时，该命令会创建一个提交对象，并用 HEAD 文件中那个引用所指向的 SHA-1 值设置其父提交字段。

你也可以手动编辑该文件，然而同样存在一个更安全的命令来完成此事：`git symbolic-ref`。 可以借助此命令来查看 HEAD 引用对应的值：

```
$ git symbolic-ref HEAD
refs/heads/master
```

同样可以设置 HEAD 引用的值：

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

不能把符号引用设置为一个不符合引用规范的值：

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

标签引用

前面我们刚讨论过 Git 的三种主要的对象类型（**数据对象**、**树对象** 和 **提交对象** ），然而实际上还有第四种。 **标签对象（tag object）** 非常类似于一个提交对象——它包含一个标签创建者信息、一个日期、一段注释信息，以及一个指针。 主要的区别在于，标签对象通常指向一个提交对象，而不是一个树对象。 它像是一个永不移动的分支引用——永远指向同一个提交对象，只不过给这个提交对象加上一个更友好的名字罢了。

正如 Git 基础 中所讨论的那样，存在两种类型的标签：附注标签和轻量标签。 可以像这样创建一个轻量标签：

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

这就是轻量标签的全部内容——一个固定的引用。 然而，一个附注标签则更复杂一些。 若要创建一个附注标签，Git 会创建一个标签对象，并记录一个引用来指向该标签对象，而不是直接指向提交对象。 可以通过创建一个附注标签来验证这个过程（使用 `-a` 选项）：

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

下面是上述过程所建标签对象的 SHA-1 值：

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

现在对该 SHA-1 值运行 `git cat-file -p` 命令：

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

我们注意到，object 条目指向我们打了标签的那个提交对象的 SHA-1 值。 另外要注意的是，标签对象并非必须指向某个提交对象；你可以对任意类型的 Git 对象打标签。 例如，在 Git 源码中，项目维护者将他们的 GPG 公钥添加为一个数据对象，然后对这个对象打了一个标签。 可以克隆一个 Git 版本库，然后通过执行下面的命令来在这个版本库中查看上述公钥：

```
$ git cat-file blob junio-gpg-pub
```

Linux 内核版本库同样有一个不指向提交对象的标签对象——首个被创建的标签对象所指向的是最初被引入版本库的那份内核源码所对应的树对象。

远程引用

我们将看到的第三种引用类型是远程引用（remote reference）。 如果你添加了一个远程版本库并对其执行过推送操作，Git 会记录下最近一次推送操作时每一个分支所对应的值，并保存在 `refs/remotes` 目录下。 例如，你可以添加一个叫做 `origin` 的远程版本库，然后把 `master` 分支推送上去：

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

此时，如果查看 `refs/remotes/origin/master` 文件，可以发现 `origin` 远程版本库的 `master` 分支所对应的 SHA-1 值，就是最近一次与服务器通信时本地 `master` 分支所对应的 SHA-1 值：

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

远程引用和分支（位于 `refs/heads` 目录下的引用）之间最主要的区别在于，远程引用是只读的。 虽然可以 `git checkout` 到某个远程引用，但是 Git 并不会将 HEAD 引用指向该远程引用。因此，你永远不能通过 `commit` 命令来更新远程引用。 Git 将这些远程引用作为记录远程服务器上各分支最后已知位置状态的书签来管理。