PDF-XChange
Click to buy NOW!
www.docu-track.com

git与github学习

笔记本： INBOX
创建时间： 2015/2/5 14:21 **更新时间：** 2015/3/12 10:13
URL： http://www.worldhello.net/gotgithub/01-explore-github/010-what-is-github.html

. 概述

github是一个面向开源及私有软件项目的托管平台, 因采用了git托管方式而得名.现在已经超过Google和SourceForge成为全球最大的代码托管平台.

使用github不仅可以维护你的代码, 还可以在上面找到很多很有趣的项目, 让世界各地的人知道你的存在.

本文从下面几个方面来了解github和其所依赖的版本控制工具的方方面面:

. 什么是Github?什么是Git?两者什么关系?
    . 版本工具的发展历程
    . Git原理
    . 如何安装使用Git
. Github能做什么?
. Github上如何创建和维护一个项目
. 相关资源

本人在开始写这篇文章的时候, 是一个git菜鸟, 一方面希望用写文章这种方式去规划自己的学习计划, 另外也希望能留下一些总结, 对日后也有一些帮助.

. 什么是Github?什么是Git?两者什么关系

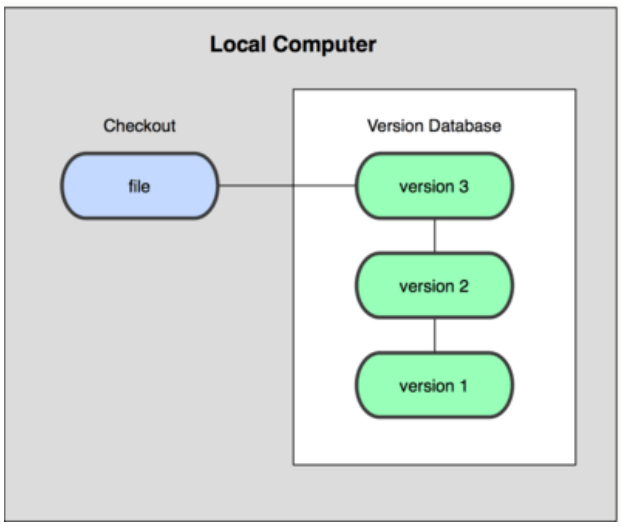
. Git

版本控制软件系统

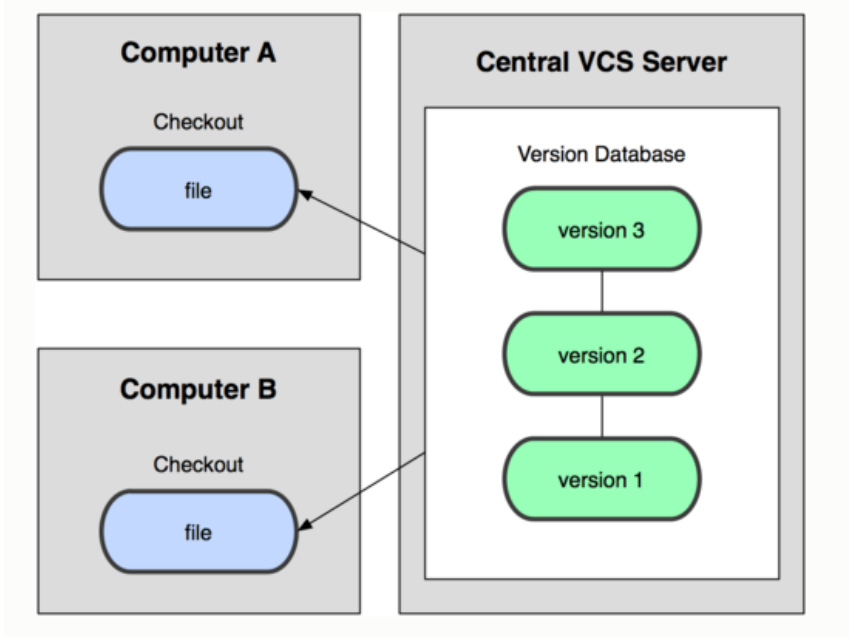
提到Git, 必然要提到版本控制软件的发展历史, 什么是版本控制? 为什么要关心它呢?版本控制是一种记录一个或若干文件内容变化, 以便将来查阅特定版本修订情况的系统. 在本文章所展示的例子中, 我们仅对保存着软件源代码的文本文件作版本控制管理, 但实际上, 你可以对任何类型的文件进行版本控制.

版本控制软件的作用明显可以极大的提高我们的工作效率, 在版本控制软件的发展初期, 人们用复制整个文件的方式加时间来保存不同的版本, 这种方法的唯一好处是非常简单, 但坏处是: 一旦混淆了工作目录, 弄丢了数据就无法恢复. 并且这种方法占用大量的硬盘空间, 并且很难理解不同版本的差异和当时人们为啥这样修改.

后来, 人们采用了一些数据库的技术, 开发了一种叫本地版本控制软件的控制版本软件这种软件使用数据库记录文件历次更替差异. 其中最著名的一种叫rcs, 现在在很多流行计算机系统都还在使用.



但是, 随着开发规模的增大, 软件开发不仅仅是个人的任务, 渐渐地, 以团队, 公司为单位的开发组织越来越多, 如何解决不同系统上的开发者协同工作, 成为了版本控制软件的新功能需求. 于是, 集中版本控制软件(CVCS)诞生了, 这类系统，诸如 CVS, Subversion 以及 Perforce 等, 都有一个单一的集中管理的服务器, 保存所有文件的修订版本, 而协同工作的人们都通过客户端连到这台服务器, 取出最新的文件或者提交更新. 多年以来, 这已成为版本控制系统的标准做法.

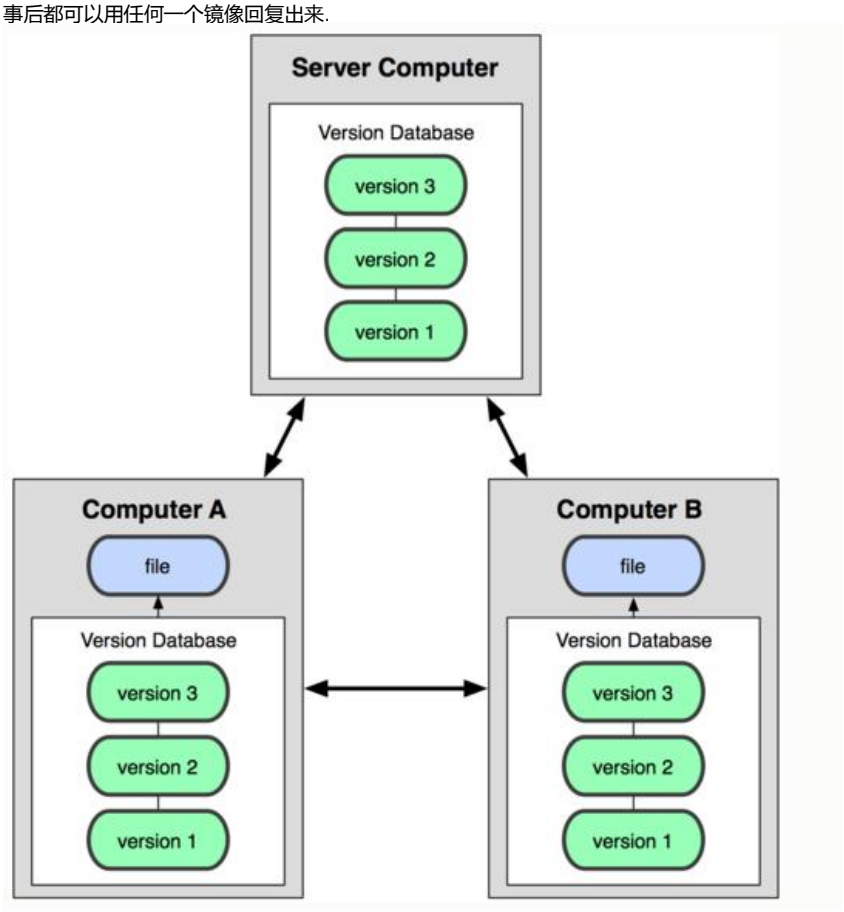


这种做法带来了许多好处, 特别是相较于老式的本地 VCS 来说。现在，每个人都可以在一定程度上看到项目中的其他人正在做些什么. 而管理员也可以轻松掌控每个开发者的权限, 并且管理一个 CVCS 要远比在各个客户端上维护本地数据库来得轻松容易.

事分两面, 有好有坏. 这么做最显而易见的缺点是中央服务器的单点故障. 如果宕机一小时, 那么在这一小时内, 谁都无法提交更新, 也就无法协同工作. 要是中央服务器的磁盘发生故障, 碰巧没做备份, 或者备份不够及时, 就会有丢失数据的风险. 最坏的情况是彻底丢失整个项目的所有历史更改记录, 而被客户端偶然提取出来的保存在本地的某些快照数据就成了恢复数据的希望. 但这样的话依然是个问题, 你不能保证所有的数据都已经有人事先完整提取出来过. 本地版本控制系统也存在类似问题, 只要整个项目的历史记录被保存在单一位置, 就有丢失所有历史更新记录的风险.

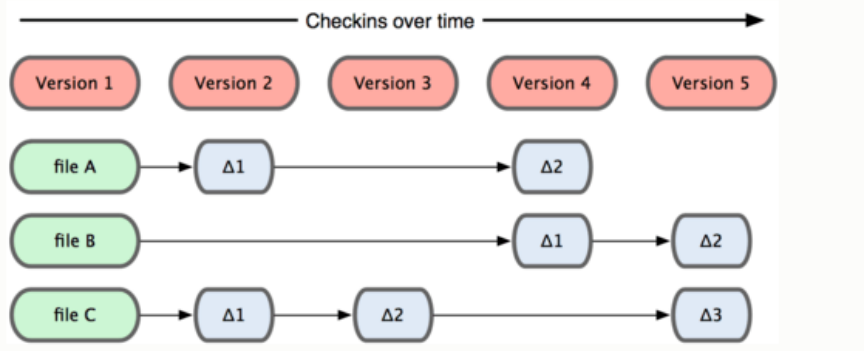
为了解决这个问题, 分布式版本控制系统面世了, 这类系统中, 客户端不仅仅提取最新的版本快照, 而是把整个代码仓库完整地镜像下来. 这么一来, 任何一处协同工作的服务器发生了问题,

事后都可以用任何一个镜像回复出来.



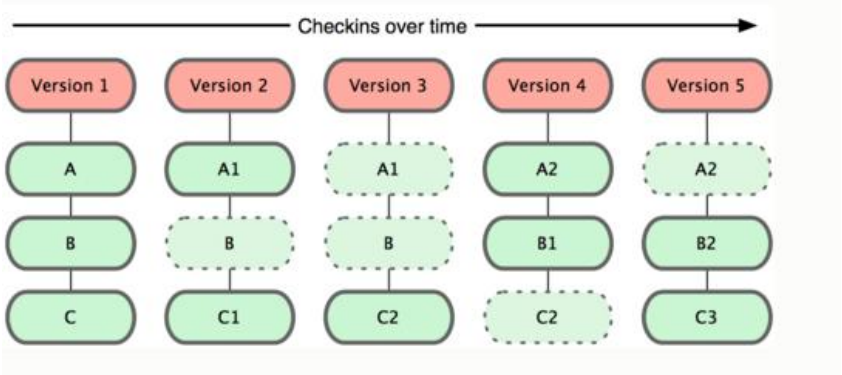
Git是分布式版本控制系统的一种, 这类系统都可以指定和若干不同的远端代码仓库进行交互. 籍此, 你就可以在同一个项目中, 分别和不同工作小组的人相互协作.
你可以根据需要设定不同的协作流程, 比如层次模型式的工作流, 而这在以前的集中式系统中是无法实现的.

Git基础
. Git目标
    . 速度
    . 简单的设计
    . 对非线性开发模式的支持
    . 完全分布式
    . 有能力对Linux内核这种项目进行高效管理
. Git内容
    . 尽管Git看起来操作和其他的版本控制软件非常相同, 但事实上, git和其他的版本控制软件的做法非常不同.
    . **直接记录快照, 而不是差异对比**.
    Git只关心文件数据整体是否发生变化, 而其他的系统只关心文件内容的具体差异. 他们每次都记录有哪些文件做了更新, 如下图:



PDF-XChange
Click to buy NOW!
www.docu-track.com

但Git并不记录和保存这些前后变化的差异数据. Git 更像是把变化的文件作快照后, 记录在一个微型的文件系统中. 每次提交更新时, 它会纵览一遍所有文件的指纹信息并对文件作一快照, 然后保存一个指向这次快照的索引. 为提高性能, 若文件没有变化, Git 不会再次保存, 而只对上次保存的快照作一链接. Git 的工作方式如下图所示:



这是 Git 同其他系统的重要区别. 它完全颠覆了传统版本控制的套路, 并对各个环节的实现方式作了新的设计. Git 更像是个小型的文件系统, 但它同时还提供了许多以此为基础的超强工具, 而不只是一个简单的 VCS.

关于快照的概念可以看下下面两篇文章

* http://www.v2ex.com/t/120979
* http://stackoverflow.com/questions/8198105/how-does-git-store-files/8198276#8198276

. Git**几乎所有的操作都是在本地操作的**, 而CVCS基本上所有的操作都和网络有关. Git在本地就保存所有的版本历史更新, 处理速度飞快

. Git时刻保持数据的完整性, 在保存到 Git 之前, 所有数据都要进行内容的校验和(checksum) 计算, 并将此结果作为数据的唯一标识和索引. 换句话说, 不可能在你修改了文件或目录之后, Git 一无所知. 这项特性作为 Git 的设计哲学, 建在整体架构的最底层. 所以如果文件在传输时变得不完整, 或者磁盘损坏导致文件数据缺失, Git 都能立即察觉.

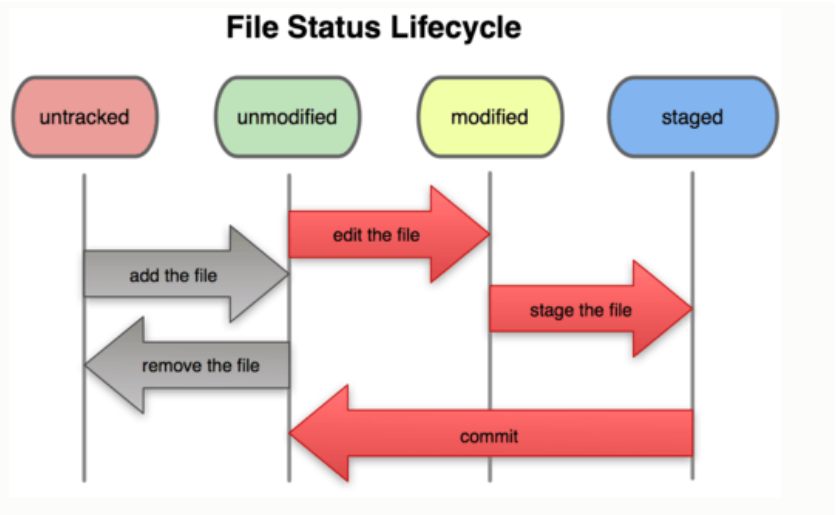
. Git多数操作仅添加数据. 常用的 Git 操作大多仅仅是把数据添加到数据库. 因为任何一种不可逆的操作, 比如删除数据, 都会使回退或重现历史版本变得困难重重. 在别的 VCS 中, 若还未提交更新, 就有可能丢失或者混淆一些修改的内容, 但在 Git 里, 一旦提交快照之后就完全不用担心丢失数据, 特别是养成定期推送到其他仓库的习惯的话.

. Git内的文件只三种状态:
  . Committed(已提交)
  . modified(已修改)
  . staged(已暂存)

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
  . 本地仓库

PDF-XChange Click to buy NOW! www.docu-track.com



每个项目都有一个 Git 目录 (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:

. 在工作目录中修改某些文件
. 对修改后的文件进行快照, 然后保存到暂存区域.
. 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, 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.

Git安装

Git安装极其简单:下面提供一些安装的方法:

. Linux

Linux下面安装git应该是最为简单明了的。使用yum或apt-get工具均可很方便的安装。

Git也提供源代码的安装方式。也一样很简单明了。

Git源码可以在下面网站下载：

http://git-scm.com/download

. Mac

Mac上面有两种安装Git的方式，最简单的当然是使用GUI安装。下载地址在：

http://sourceforge.net/projects/git-osx-installer/

另一种是通过Macport安装， 使用下面命令：

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

. Window

在Window下有个叫 msysGit 的项目提供了安装包，可以到 GitHub 的页面上下载安装文件并运行：

http://msysgit.github.com/

PDF-XChange
Click to buy NOW!
www.docu-track.com

> 给 Windows 用户的敬告：应该在msysGit 提供的 Unix 风格的 shell 来运行Git。在 Unix 风格的 shell 中，可以使用复杂多行的命令。对于那些需要在 Windows 命令行中使用 Git 的用户，必须注意：在参数中间有空格的时候，必须使用双引号将参数括起来（在 Linux 中是单引号）；另外，如果扬抑符（^）作为参数的结尾，并且作为这一行的最后一个字符，则这个参数也需要用双引号括起来。因为扬抑符在 Windows 命令行中表示续行。

Git运行初始化
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.

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
```

之后是编辑器, 当然这个不设也没啥问题.

```
$ git config --global core.editor emacs
```

差异分析工具:

```
$ git config --global merge.tool vimdiff
```

查看配置信息:

```
$ git config --list
```

直接查询某个变量, 如user.name:

```
$ git config user.name
```

获得帮助:

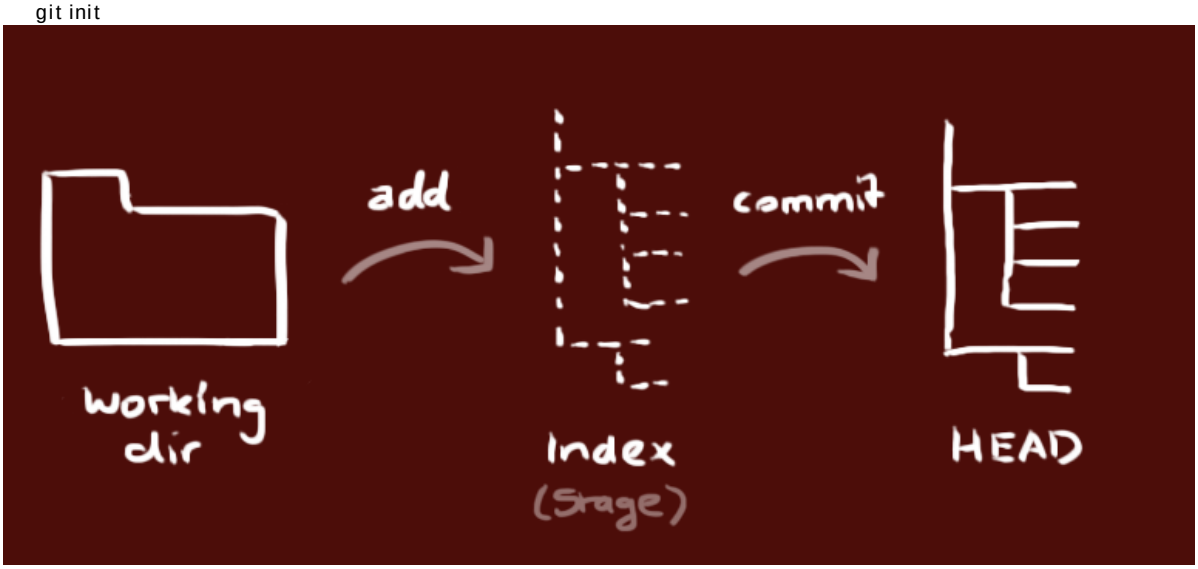
```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

也可以上irc.freenode.net上的 `#git` 或 `#github` 频道寻求他人帮助

工作目录中初始化新仓库:

PDF-XChange
Click to buy NOW!
www.docu-track.com

git init



从工作目录到暂存目录, 使用add命令.

git add *

从暂存目录到本地仓库, 使用commit命令.

git commit -m "*******"

git add这个命令的语义是个比较容易引起疑问的语义, 在 subversion 中的 svn add 动作是将某个文件加入版本控制，而 git add的意义完全不同。

同时，git diff --cached是比较 stage 的文件的差异的，也是一个不直观的命令。

在git的后续版本, 就做了两个修改：

git stage 作为 git add 的一个同义词

git diff --staged 作为 git diff --cached 的相同命令

* stage area的意义
  . 分批提交
  . 分阶段提交
  . 文件快照, 方便回退

git也提供直接跳过stage area的方法:

git commit -a

从现有仓库克隆:

git clone [url]

查看提交记录:

git log

PDF-XChange Click to buy NOW! www.docu-track.com

选项	说明
-p	按补丁格式显示每个更新之间的差异。
--word-diff	按 word diff 格式显示差异。
--stat	显示每次更新的文件修改统计信息。
--shortstat	只显示 --stat 中最后的行数修改添加移除统计。
--name-only	仅在提交信息后显示已修改的文件清单。
--name-status	显示新增、修改、删除的文件清单。
--abbrev-commit	仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。
--relative-date	使用较短的相对时间显示（比如，“2 weeks ago”）。
--graph	显示 ASCII 图形表示的分支合并历史。
--pretty	使用其他格式显示历史提交信息。可用的选项包括 oneline，short，full，fuller 和 format（后跟指定格式）。
--oneline	--pretty=oneline --abbrev-commit 的简化用法。

查看工作目录状态:

```
git status
```

重新提交:

```
git commit --amend
```

取消暂存:

```
git reset HEAD <filename>
```

取消修改:

```
git checkout -- <filename>
```

远端仓库:

查看远端仓库:

```
git remote
```

添加远程仓库:

```
git remote add <shortname> <url>
```

从远程仓库抓取数据:

```
git fetch <remote-name>
```

此命令会到远程仓库中拉取所有你本地仓库中还没有的数据。运行完成后，你就可以在本地访问该远程仓库中的所有分支，将其中某个分支合并到本地，或者只是取出某个分支，一探究竟。

推送数据到远程仓库:

```
git push <remote-name> <brach-name>
```

查看远程仓库信息:

```
git remote show <remote-name>
```

远程仓库删除和重命名:

```
git remote rename
```

远端仓库合并到本地仓库:

```
git pull
```

Git标签:

打标签, 是基本上所有VCS都带有的功能, 标签表示在某个时间的版本, 在发布版本的

PDF-XChange
Click to buy NOW!
www.docu-track.com

时候, 常常这样做.GIT也不例外地提供了这样的功能.

Git的标签有两种类型:

. 轻量级标签

轻量级标签像是个不会变化的分支，实际上, 轻量级标签就是个指向特定提交对象的引用

. 含标注标签

含附注标签，实际上是存储在仓库中的一个独立对象，它有自身的校验和信息，包含着标签的名字，电子邮件地址和日期，以及标签说明，标签本身也允许使用 GNU Privacy Guard (GPG) 来签署或验证。

. 添加标签

添加轻量级标签:

```
git tag <tag-name>
```

添加含标注标签:

```
git tag -a <tag-name> -m <commit>
```

签署标签:

```
git tag -s <tag-name> -m <commit>
```

后期加注标签:

```
git tag -a <tag-name> <sumup of version>
```

. 查看标签

```
git tag
```

. 验证标签

```
git tag -v <tag-name>
```

. 分享标签

```
git push origin <tag-name>
```

一次推送所有标签

```
git push origin --tags
```

Git分支

. 分支概念

建立分支对很多版本工具来说都是花费非常大的操作, 但是, Git的版本控制是难以想象的轻量级, 它的新建操作几乎可以在一瞬间完成.

Git鼓励在工作中频繁使用分支与合并. 这是Git和其他版本控制工具分开的地方.

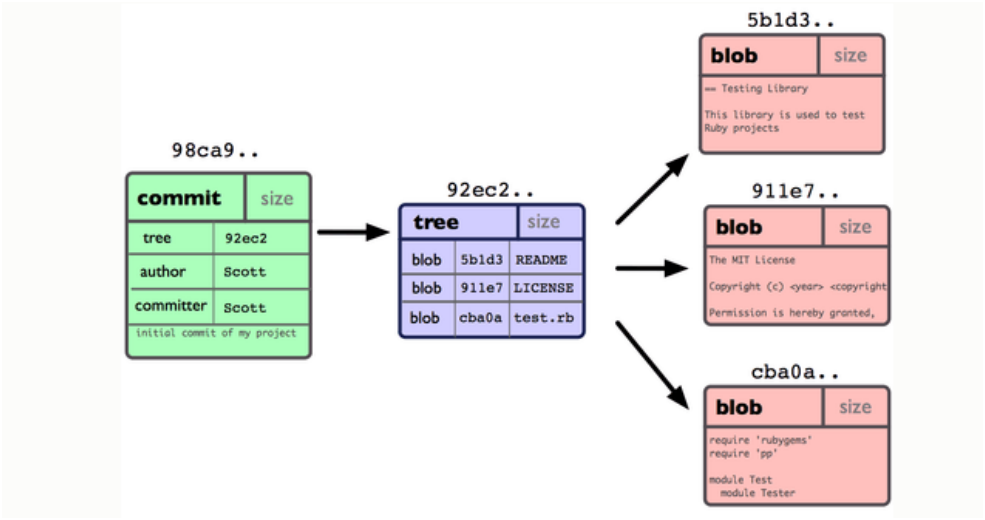
. 分支概念

我们先来看下修改提交的时候发生的事情:

. 当使用git commit提交一个对象的时候, Git会先计算每一个子目录的校验和

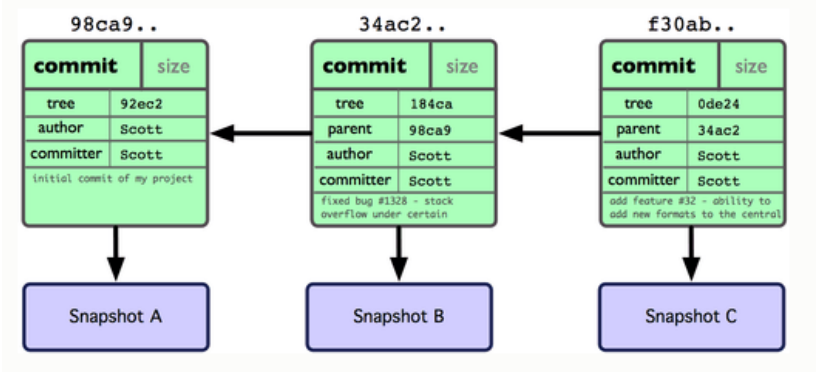
. 在Git仓库中, Git将这个目录保存为tree的对象, tree对象是一个关于其他提交信息元素的commit对象

. Git创建的commit对象中, 除了提交信息外, 还包含指向这个树对象的指针.从概念上讲如下图, 这样就可以很方便的找到这个快照的内容.



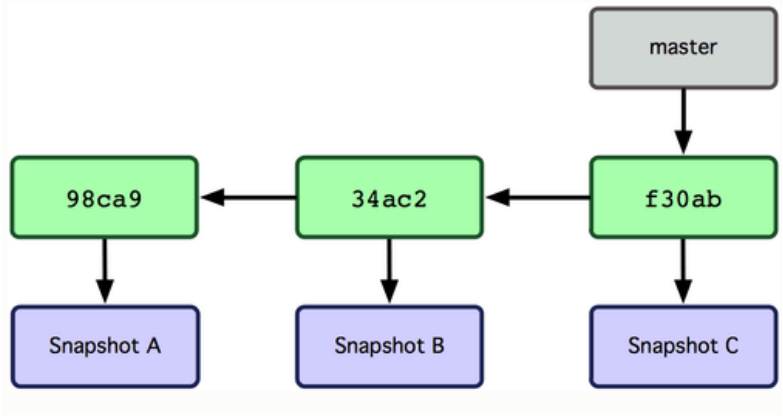
这是一次提交中发送的事情,多次提交会发生什么事情呢?我们继续往下看

. 在多次修改后, 原先的commit对象中会增加一个指向上次提交的commit的指针, 如下图:



好的, 现在开始我们可以讨论下分支了:

. 所谓的分支, 本质上来说就是一个指向commit的指针



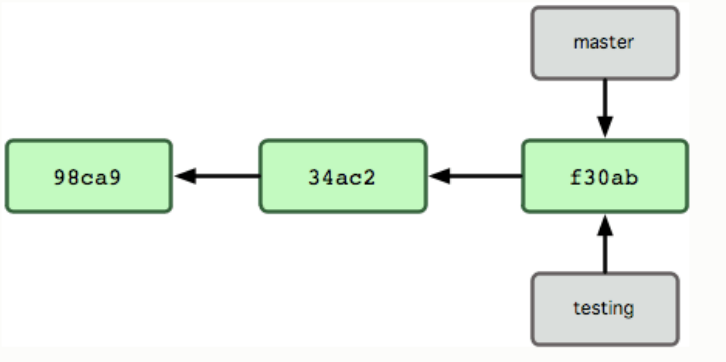
这里的master分支, 指向的就是最后提交的一次commit.

这样, Git创建一个分支是一件非常简单的事情:

假设我们要创建一个叫testing的分支, 我们只需要输入一个这样的命令:

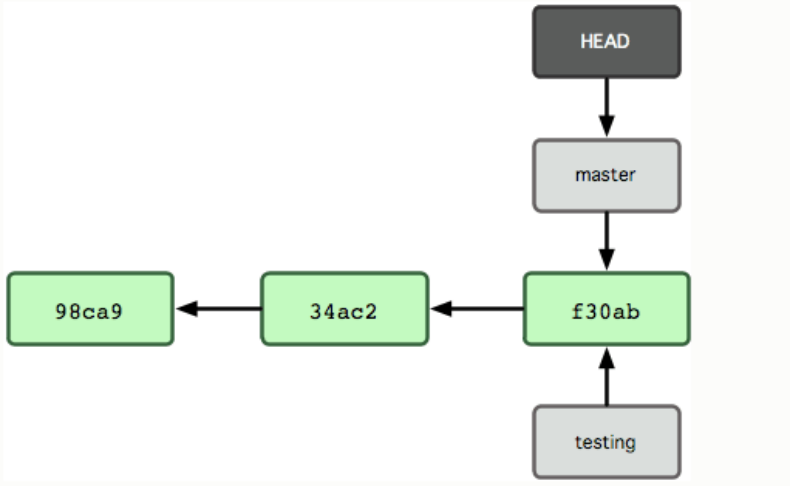
```
git branch testing
```

PDF-XChange
Click to buy NOW!
www.docu-track.com



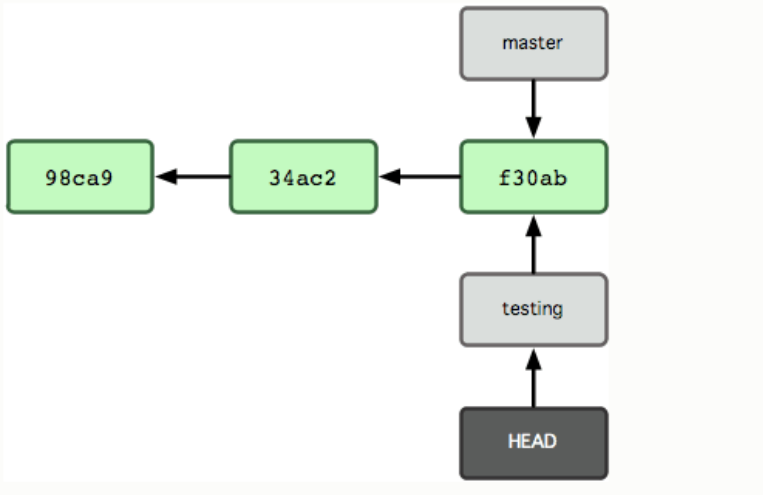
这会在当前的commit对象上创建一个新的指针, 这样分支就建立完成了!

另外一个问题: 如何指定在哪个分支上工作?
Git提供了一个很简单的方法, 它保存了一个叫HEAD的指针, 他是一个指向你正在工作的本地版本的指针.



切换到其他分支:

```
git checkout testing
```



这样HEAD就指向了testing.
分支间的切换, 用branch和checkout两个命令就可以. 切换只需要修改指针就可以.
同时因为每次提交都记录了parent对象, 将来要合并分支的时候, 寻找适合的合并基础的工作就完成了.

分支合并, 只需要输入:

```
git merge <branch-name>
```

git会自动查找最合适的合并点去进行合并

PDF-XChange
Click to buy NOW!
www.docu-track.com

. 分支管理

列举所有分支:

```
git branch
```

查看分支的最后一个提交信息:

```
git branch -v
```

筛选出未合并(或已经合并)分支, 可以使用 --merged或--no-merged

删除分支, 使用 -d 选项

. git分支开发流程

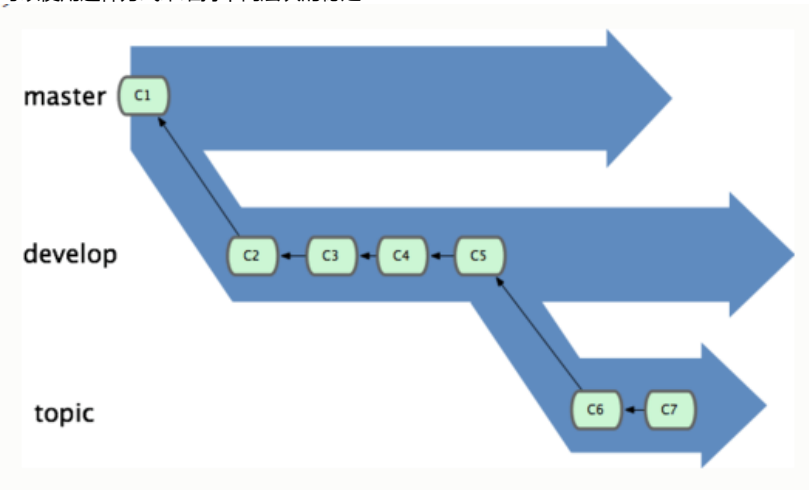
git的分支管理很有特点, 由此衍生出一些利用分支开发的工作流程:

. 长期分支

仅在 `master` 分支中保留完全稳定的代码，即已经发布或即将发布的代码。与此同时，他们还有一个名为 `develop` 或 `next` 的平行分支，专门用于后续的开发，或仅用于稳定性测试 — 当然并不是说一定要绝对稳定，不过一旦进入某种稳定状态，便可以把它合并到 `master` 里。这样，在确保这些已完成的特性分支能够通过所有测试，并且不会引入更多错误之后，就可以并到主干分支中，等待下一次的发布。

随着提交对象不断右移的指针。稳定分支的指针总是在提交历史中落后一大截，而前沿分支总是比较靠前

可以使用这种方式来维持不同层次的稳定



. 特性分支

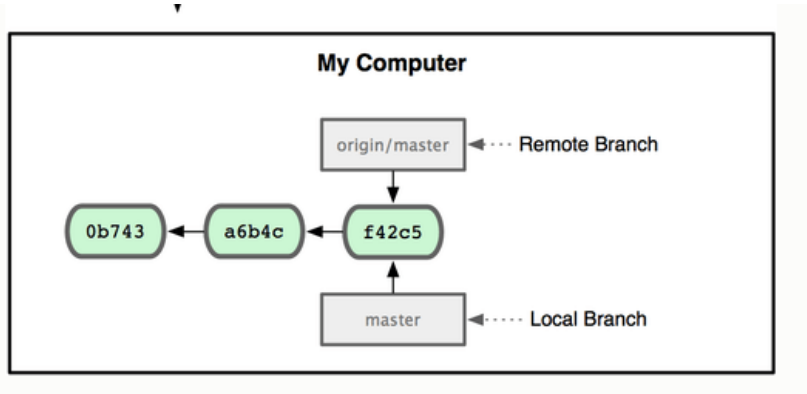
在任何规模的项目中都可以使用特性（Topic）分支。一个特性分支是指一个短期的，用来实现单一特性或与其相关工作的分支。可能你在以前的版本控制系统里从未做过类似这样的事情，因为通常创建与合并分支消耗太大。然而在 Git 中，一天之内建立、使用、合并再删除多个分支是常见的事。

. 远程分支

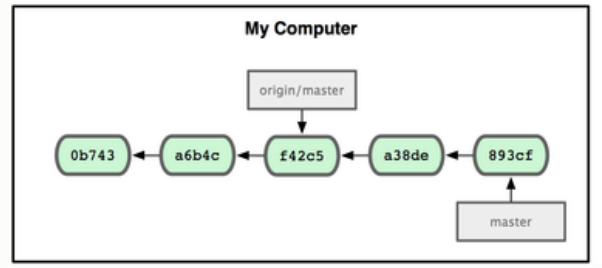
远程分支（remote branch）是对远程仓库中的分支的索引。它们是一些无法移动的本地分支；只有在 Git 进行网络交互时才会更新。远程分支就像是书签，提醒着你上次连接远程仓库时上面各分支的位置。

Git中使用(远程仓库名/分支名)这样的命名方式来表示远程分支,

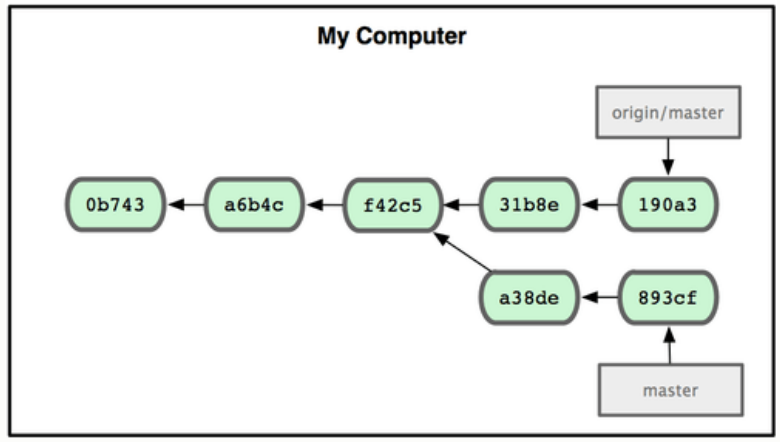
第一次clone后, git会在本地建立master和origin/master分支, 并将它们都指向origin/master:



如果在本地master分支上做了改动, 那master的指针就会不断地向前推进, 与此同时服务器上的origin/master也在不断地更新, 但只要不和服务器通讯, 本地的origin/master不会前进.



可以运行 `git fetch origin` 来同步远程服务器上的数据到本地。该命令首先找到 `origin` 是哪个服务器，从上面获取你尚未拥有的数据，更新你本地的数据库，然后把`origin/master` 的指针移到它最新的位置上。



推送本地分支:

```
git push <remote-name> <branch-name>
```

跟踪远程分支:

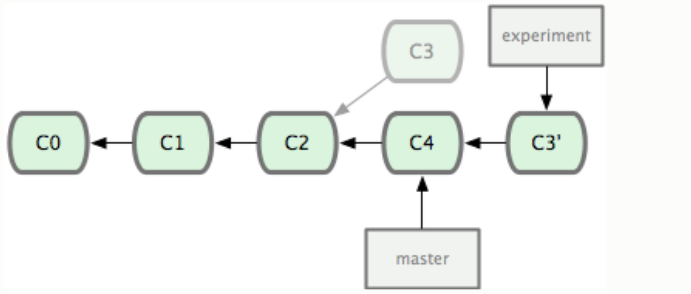
从远程分支 `checkout` 出来的本地分支，称为 *跟踪分支* (tracking branch)。跟踪分支是一种和某个远程分支有直接联系的本地分支。在跟踪分支里输入 git push，Git 会自行推断应该向哪个服务器的哪个分支推送数据。同样，在这些分支里运行 `git pull` 会获取所有远程索引，并把它们的数据都合并到本地分支中来。

删除分支:

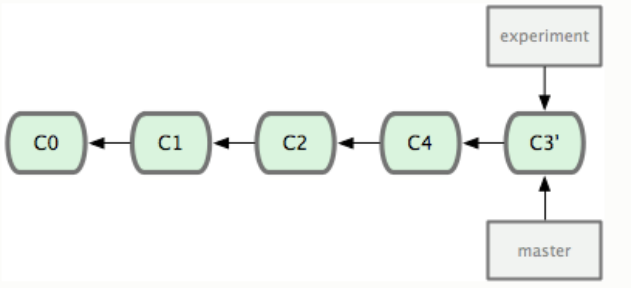
如果不再需要某个远程分支了，比如搞定了某个特性并把它合并进了远程的 `master` 分支（或任何其他存放稳定代码的分支），可以用这个非常无厘头的语法来删除它：`git push [远程名] :[分支名]`。

分支衍合:

分支合并有两种方式, 一种是之前提过的merge(合并), 另一种是rebase(衍合)
衍合的原理是回到两个分支最近的共同祖先，根据当前分支（也就是要进行衍合的分支 `experiment`）后续的历次提交对象（这里只有一个 C3），生成一系列文件补丁，然后以基底分支（也就是主干分支`master`）最后一个提交对象（C4）为新的出发点，逐个应用之前准备好的补丁文件，最后会生成一个新的合并提交对象（C3'），从而改写 `experiment` 的提交历史，使它成为 `master` 分支的直接下游.



回到 `master` 分支，进行一次快进合并



现在的 C3' 对应的快照，其实和普通的三方合并，即上个例子中的 C5 对应的快照内容一模一样了。虽然最后整合得到的结果没有任何区别，但衍合能产生一个更为整洁的提交历史。如果视察一个衍合过的分支的历史记录，看起来会更清楚：仿佛所有修改都是在一根线上先后进行的，尽管实际上它们原本是同时并行发生的。

使用衍合要遵守一点规则:

一旦分支中的提交对象发布到公共仓库，就千万不要对该分支进行衍合操作。

服务器上的Git

. Git上的协议

本地传输

. 存储位置

本地硬盘

. 应用场景

. 多人在同一服务器上通过NFS共同开发

. 团队共用一台电脑

. 操作

. 本地创建

git clone /opt/git/project.git

git clone file:///opt/git/project.git

这两种方式有一点点不同, 只给出路径，Git 会尝试使用硬链接或直接复制它所需要的文件。如果使用了 `file://` ，Git 会调用它平时通过网络来传输数据的工序，而这种方式的效率相对较低。使用 `file://` 前缀的主要原因是当你需要一个不包含无关引用或对象的干净仓库副本的时候 — 一般指从其他版本控制系统导入的，或类似情形。

. 特点

优点

. 创建简单

缺点

PDF-XChange Click to buy NOW! www.docu-track.com

. 难以控制不同位置的访问控制

SSH协议

. 存储位置

远程SSH服务器

. 应用场景

SSH是架设比较容易的网络协议, 同时SSH是GIT支持的三种网络协议中唯一同时支持读写的, HTTP和GIT都只支持读操作. 基本上所有网络上的仓库都需要支持SSH.

. 操作

. 本地创建

git clone ssh://user@server/project.git

git clone user@server/project.git不指定时默认也是使用SSH

. 特点

优点

. SSH架设简单

. SSH支持读写

. SSH是安全的协议, 所有的数据和授权都是加密的

. SSH是高效的协议, 在传输的时候会压缩数据

缺点

. 不能通过SSH实现仓库的匿名访问。即使仅为读取数据，人们也必须在能通过 SSH 访问主机的前提下才能访问仓库，这使得 SSH 不利于开源的项目。

Git协议

. 存储位置

远程服务器

. Git协议简介

Git协议是一个包含在 Git 软件包中的特殊守护进程； 它会监听一个提供类似于 SSH 服务的特定端口（9418），而无需任何授权。打算支持 Git 协议的仓库，需要先创建 `git-daemon-export-ok` 文件 — 它是协议进程提供仓库服务的必要条件 — 但除此之外该服务没有什么安全措施。

. 应用场景

Git协议没有授权机制, 通常不能用在需要推送的项目

. 特点

优点

. Git 协议是现存最快的传输协议。如果你在提供一个有很大访问量的公共项目，或者一个不需要对读操作进行授权的庞大项目，架设一个 Git 守护进程来供应仓库是个不错的选择。它使用与 SSH 协议相同的数据传输机制，但省去了加密和授权的开销。

缺点

. Git 协议消极的一面是缺少授权机制。用 Git 协议作为访问项目的唯一方法通常是不可取的。一般的做法是，同时提供 SSH 接口，让几个开发者拥有推送（写）权限，其他人通过 `git://` 拥有只读权限。 Git 协议可能也是最难架设的协议。它要求有单独的守护进程，需要定制 — 我们将在本章的“ Gitosis ”一节详细介绍它的架设 — 需要设定 `xinetd` 或类似的程序，而这些工作就没那么轻松了。该协议还要求防火墙开放 9418 端口，而企业级防火墙一般不允许对这个非标准端口的访问。大型企业级防火墙通常会封锁这个少见的端口。

HTTP协议

. HTTP协议简介

HTTP 或 HTTPS 协议的优美之处在于架设的简便性。基本上，只需要把 Git 的裸仓库文件放在 HTTP 的根目录下，配置一个特定的 `post-update` 挂钩（hook）就可以搞定。此后，每个能访问 Git 仓库所在服务器上 web 服务的人都可以进行克隆操作。下面的操作可以允许通过 HTTP 对仓库进行读取：其他人就可以用下面的命令来克隆仓库：

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

更多请参照网页:

Pro Git Chapter4.5

. HTTP特点

优点

. 使用 HTTP 协议的好处是易于架设。几条必要的命令就可以让全世界读取到仓库的内容。花费不过几分钟。HTTP 协议不会占用过多服务器资源。因为它一般只用到静态的 HTTP 服务提供所有数据，普通的 Apache 服务器平均每秒能支撑数千个文件的并发访问 — 哪怕让一个小型服务器超载都很难。

你也可以通过 HTTPS 提供只读的仓库，这意味着你可以加密传输内容；你甚至可以要求客户端使用特定签名的 SSL 证书。一般情况下，如果到了这一步，使用 SSH 公共密钥可能是更简单的方案；不过也存在一些特殊情况，这时通过 HTTPS 使用带签名的 SSL 证书或者其他基于 HTTP 的只读连接授权方式是更好的解决方案。

缺点

. HTTP 协议的消极面在于，相对来说客户端效率更低。克隆或者下载仓库内容可能会花费更多时间，而且 HTTP 传输的体积和网络开销比其他任何一个协议都大。因为它没有按需供应的能力 — 传输过程中没有服务端的动态计算

. 应用场景

Git协议没有授权机制, 通常不能用在需要推送的项目

. 特点

优点

. Git 协议是现存最快的传输协议。如果你在提供一个有很大访问量的公共项目，或者一个不需要对读操作进行授权的庞大项目，架设一个 Git 守护进程来供应仓库是个不错的选择。它使用与 SSH 协议相同的数据传输机制，但省去了加密和授权的开销。

缺点

. Git 协议消极的一面是缺少授权机制。用 Git 协议作为访问项目的唯一方法通常是不可取的。一般的做法是，同时提供 SSH 接口，让几个开发者拥有推送（写）权限，其他人通过 `git://` 拥有只读权限。 Git 协议可能也是最难架设的协议。它要求有单独的守护进程，需要定制 .

该协议还要求防火墙开放 9418 端口，而企业级防火墙一般不允许对这个非标准端口的访问。大型企业级防火墙通常会封锁这个少见的端口。

. 在服务器上部署Git

在服务上部署Git, 大致分为下面两步:

. 导出裸仓库

在开始架设服务器仓库的时候, 需要先把现有仓库导出为裸仓库:

```
git clone --bare my_project my_project.git
```

这一步相当于git init + git fetch

效果大致相当于:

```
cp -Rf my_project/.git my_project.git
```

. SSH

在搭建了SSH服务的服务器上面架设Git是再简单不过的事情了, 只需要把目录复制过去就可以了.

复制之后, 所有对这个服务器有访问权限的机器都可以用下面的命令对项目进行复制:

```
gii clone user@git.example.com:/sshroot/my_project.git
```

如果某个用户对这个目录有写权限, 那么他就对这个项目有推送权限

架设git很简单, 难的是对Git进行访问控制, 但git可以使用服务器操作系统的文件访问机制来解决这样的问题.

. Github托管

PDF-XChange
Click to buy NOW!
www.docu-track.com

分四步:

. 注册账号

. 建立仓库

. 初始化本地仓库并添加远程仓库

. 推送本地到远端

具体看Pro Git Chapter4.10

Git内部原理

. 底层命令(Plumbing)和高层命令(Protecting)

Git开始的设计并不是一整套对用户友好的VCS, 而是供VCS使用的工具集, 很多Git命令是被设计成为VCS调用的底层命令, 这部分命令被称为"plumbing"而对用户友好的那些命令被称为"porcelain".

plumbing是为其他工具和自定义脚本服务的, 而porcelain是为用户服务

. git里的内容

当我们调用git init的时候, git会在当前目录中创建一个".git"的目录, 几乎所有的git存储和操作的内容都会在这个目录里面. 如果我们需要备份或复制一个库, 基本上将这个目录复制出来就可以了, 下面我们来讨论下这个目录里面的内容:

```
$ ls -la
total 4
drwxr-xr-x   14 Administ Administ        0 Mar 10 14:06 .
drwxr-xr-x    6 Administ Administ        0 Mar 10 14:06 ..
-rw-r--r--    1 Administ Administ       14 Mar 10 14:23 COMMIT_EDITMSG
-rw-r--r--    1 Administ Administ       99 Mar 10 14:31 FETCH_HEAD
-rw-r--r--    1 Administ Administ       23 Mar 10 14:06 HEAD
-rw-r--r--    1 Administ Administ       41 Mar 10 14:31 ORIG_HEAD
-rw-r--r--    1 Administ Administ      269 Mar 10 14:10 config
-rw-r--r--    1 Administ Administ       73 Mar 10 14:06 description
drwxr-xr-x   11 Administ Administ        0 Mar 10 14:06 hooks
-rw-r--r--    1 Administ Administ      278 Mar 10 14:23 index
drwxr-xr-x    3 Administ Administ        0 Mar 10 14:06 info
drwxr-xr-x    4 Administ Administ        0 Mar 10 14:09 logs
drwxr-xr-x   15 Administ Administ        0 Mar 10 14:06 objects
drwxr-xr-x    5 Administ Administ        0 Mar 10 14:06 refs
```

`description` 文件仅供 GitWeb 程序使用，所以不用关心这些内容

`config` 文件包含了项目特有的配置选项

`info` 目录保存了一份不希望在

`.gitignore` 文件中管理的忽略模式 (ignored patterns) 的全局可执行文件

`hooks` 目录保存了第七章详细介绍了的客户端或服务端钩子脚本。

`objects` 目录存储所有数据内容

`refs` 目录存储指向数据 (分支) 的提交对象的指针

`HEAD` 文件指向当前分支

`index` 文件保存了暂存区域信息

`COMMIT_EDITMSG` 文件保存了最近一次提交的修改记录

. Git对象

. Blob:

Git是一套内容寻址文件系统, 从核心上来看不过是简单地存储键值对（key-value）。它允许插入任意类型的内容，并会返回一个键值，通过该键值可以在任何时候再取出该内容。可以通过底层命令 `hash-object` 来示范这点，传一些数据给该命令，它会将数据保存在 `.git` 目录并返回表示这些数据的键值.这样每个文件的对象称为blob

git hash-object 会计算出一段文字的hash值, 并会根据后面的选项对其创建blob对象

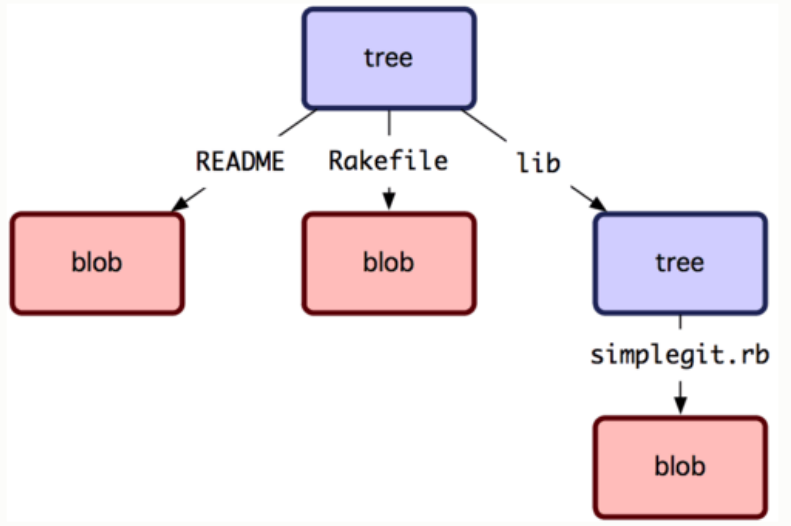
可以使用git cat-file 取回数据内容.

. tree对象

tree 对象可以存储文件名，同时也允许存储一组文件。Git 以一种类似 UNIX 文件系统但更简单的方式来存储内容。所有内容以 tree 或 blob 对象存储，其中 tree 对象对应于 UNIX 中的目录，blob 对象则大致对应于 inodes 或文件内容。一个单独的 tree 对象包含一条或多条 tree 记录，每一条记录含有一个指向 blob 或子 tree 对象

PDF-XChange
Click to buy NOW!
www.docu-track.com

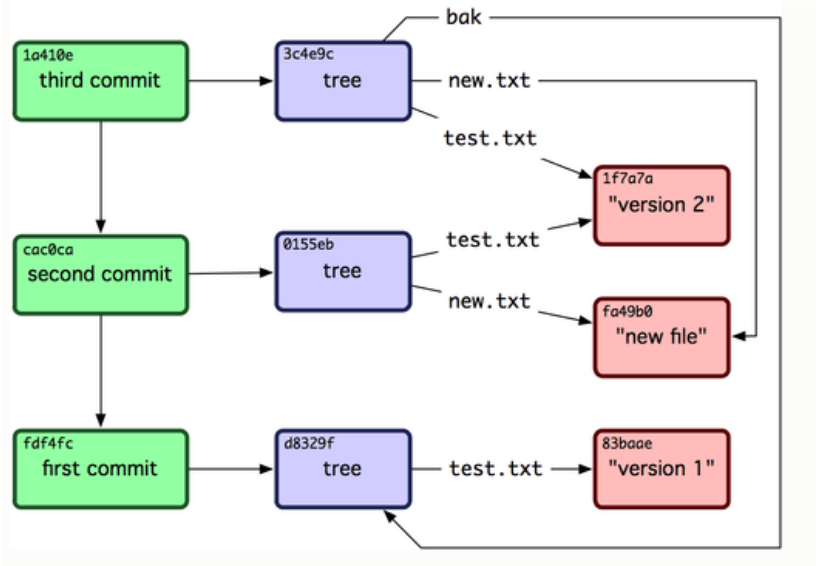
的 SHA-1 指针，并附有该对象的权限模式 (mode)、类型和文件名信息。
下面是Git对象的一个具体模型:



可以自己创建 tree 。通常 Git 根据你的暂存区域或 index 来创建并写入一个 tree 。因此要创建一个 tree 对象的话首先要通过将一些文件暂存从而创建一个 index 。可以使用 plumbing 命令 `update-index` 为一个单独文件 —— test.txt 文件的第一个版本 —— 创建一个 index 。通过该命令人为的将 test.txt 文件的首个版本加入到了一个新的暂存区域中。由于该文件原先并不在暂存区域中 (甚至就连暂存区域也还没被创建出来呢) ，必须传入 `--add` 参数;由于要添加的文件并不在当前目录下而是在数据库中，必须传入 `--cacheinfo` 参数。

. commit对象

commit 对象保存了每个对象的SHA-1 值, 以及何时保存, 保存者是谁, 为何保存的信息



这样, Git中commit, tree, blob三个对象的关系就出来了

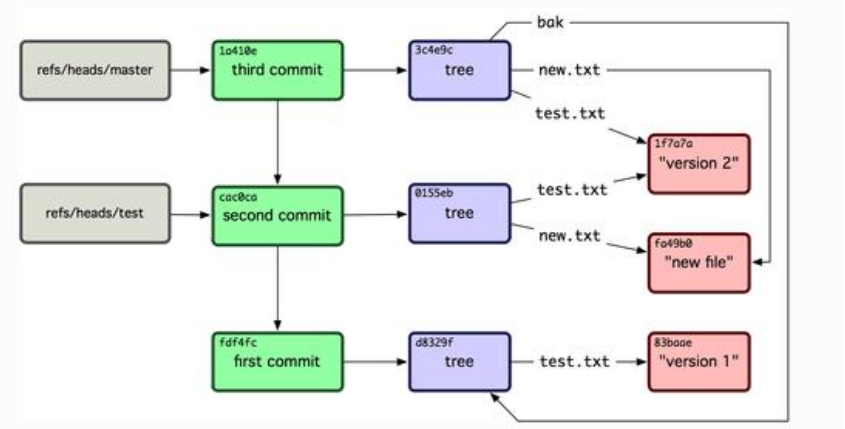
. 对象存储

Git中的对象都是用zlib对内容进行压缩存储

. Git References

. 我们可以用git log 19977这样的命令来查看完整的历史, 但是这样我们就要记得每次提交的SHA-1码, 这基本上是不可能的.所以我们需要用一个简单的名字来代替这些SHA-1值, 这种名字叫做引用.

PDF-XChange
Click to buy NOW!
www.docu-track.com



最终我们看到的应该是这样.

. Packfiles
之前我们说过, git是只保存文件快照, 不保存差异的. 但是这其实是在VCS的概念上描述这一点, 在存储的角度, GIT也是会保存差异来减少存储空间的.
打包的时机有三个:
. 手动调用git gc
. 推送远程服务器
. 自动调用git gc时调用

. 引用规则
在添加远端服务器的时候, 会在.git/config中添加一节:

```
[remote "origin"]
   url = git@github.com:schacon/simplegit-progit.git
   fetch = +refs/heads/master:refs/remotes/origin/master
```

fetch后面这一段就是引用规则, refspec.大致格式如上, 一个可选的 `+` 号，接着是 `<src>:<dst>` 的格式，这里 `<src>` 是远端上的引用格式，`<dst>` 是将要记录在本地的引用格式

. 维护与数据恢复
见git9.7
. Github
. GitHub是一个共享虚拟主机服务, 用于存放使用Git版本控制的软件代码和内容项目. 它由GitHub公司 (曾称Logical Awesome) 的开发者Chris Wanstrath, PJ Hyett和Tom Preston-Werner使用Ruby on Rails编写而成.

GitHub同时提供付费账户和为开源项目提供的免费账户. 根据在2009年的Git用户调查, GitHub是最流行的Git访问站点. 除了允许个人和组织创建和访问代码库以外, 它也提供了一些方便社会化软件开发的功能, 包括允许用户跟踪其他用户, 组织, 软件库的动态, 对软件代码的改动和bug 提出评论等. GitHub也提供了图表功能, 用于显示开发者们怎样在代码库上工作以及软件的开发活跃程度.

. WiKi资料
http://zh.wikipedia.org/wiki/GitHub

. GitHub**和**Git**的关系**
GitHub和Git之间的关系, 简而言之, GitHub是一个用Git进行版本控制的项目托管平台.

. **资源**
. Git 相关资料
. Pro Git book
http://git-scm.com/book/zh/v1
. git 简明指南 -- 相当有趣并易懂的教程
http://rogerdudler.github.io/git-guide/index.zh.html
. github上有趣的项目

PDF-XChange
Click to buy NOW!
www.docu-track.com

. 利用 Github 生成书籍
https://github.com/GitbookIO/gitbook
. 根据个人 Github 信息生成个人简历
https://github.com/resume/resume.github.com
. HTML5及游戏开发
https://github.com/mozilla/BrowserQuest
https://github.com/codecombat/codecombat
. 其他一些有意思的项目
http://www.zhihu.com/question/23498424