返回 码云 首页(目录) 章节列表 ▾ Pro Git（中文版）

7 自定义 Git

到目前为止，我阐述了 Git 基本的运作机制和使用方式，介绍了 Git 提供的许多工具来帮助你简单且有效地使用它。 在本章，我将会介绍 Git 的一些重要的配置方法和钩子机制以满足自定义的要求。通过这些工具，它会和你和公司或团队配合得天衣无缝。

7.1 配置 Git

如第一章所言，用`git config`配置 Git，要做的第一件事就是设置名字和邮箱地址：

```
$ git config --global user.name "John Doe"
    $ git config --global user.email johndoe@example.com
```

从现在开始，你会了解到一些类似以上但更为有趣的设置选项来自定义 Git。

先过一遍第一章中提到的 Git 配置细节。Git 使用一系列的配置文件来存储你定义的偏好，它首先会查找`/etc/gitconfig`文件，该文件含有 对系统上所有用户及他们所拥有的仓库都生效的配置值（译注：gitconfig是全局配置文件）， 如果传递`--system`选项给`git config`命令， Git 会读写这个文件。

接下来 Git 会查找每个用户的`~/.gitconfig`文件，你能传递`--global`选项让 Git读写该文件。

最后 Git 会查找由用户定义的各个库中 Git 目录下的配置文件（`.git/config`），该文件中的值只对属主库有效。 以上阐述的三层配置从一般到特殊层层推进，如果定义的值有冲突，以后面层中定义的为准，例如：在`.git/config`和`/etc/gitconfig`的较量中， `.git/config`取得了胜利。虽然你也可以直接手动编辑这些配置文件，但是运行`git config`命令将会来得简单些。

客户端基本配置

Git 能够识别的配置项被分为了两大类：客户端和服务器端，其中大部分基于你个人工作偏好，属于客户端配置。尽管有数不尽的选项，但我只阐述 其中经常使用或者会对你的工作流产生巨大影响的选项，如果你想观察你当前的 Git 能识别的选项列表，请运行

```
$ git config --help
```

`git config`的手册页（译注：以man命令的显示方式）非常细致地罗列了所有可用的配置项。

core.editor

Git默认会调用你的环境变量editor定义的值作为文本编辑器，如果没有定义的话，会调用Vi来创建和编辑提交以及标签信息， 你可以使用`core.editor`改变默认编辑器：

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

现在无论你的环境变量editor被定义成什么，Git 都会调用Emacs编辑信息。

commit.template

如果把此项指定为你系统上的一个文件，当你提交的时候， Git 会默认使用该文件定义的内容。 例如：你创建了一个模板文件`$HOME/.gitmessage.txt`，它看起来像这样：

```
subject line

    what happened

    [ticket: X]
```

设置`commit.template`，当运行`git commit`时， Git 会在你的编辑器中显示以上的内容， 设置`commit.template`如下：

```
$ git config --global commit.template $HOME/.gitmessage.txt
    $ git commit
```

然后当你提交时，在编辑器中显示的提交信息如下：

```
subject line

    what happened

    [ticket: X]
    # 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: lib/test.rb
    #
    ~
    ~
    ".git/COMMIT_EDITMSG" 14L, 297C
```

如果你有特定的策略要运用在提交信息上，在系统上创建一个模板文件，设置 Git 默认使用它，这样当提交时，你的策略每次都会被运用。

core.pager

core.pager指定 Git 运行诸如`log`、`diff`等所使用的分页器，你能设置成用`more`或者任何你喜欢的分页器（默认用的是`less`）， 当然你也可以什么都不用，设置空字符串：

```
$ git config --global core.pager ''
```

这样不管命令的输出量多少，都会在一页显示所有内容。

user.signingkey

如果你要创建经签署的含附注的标签（正如第二章所述），那么把你的GPG签署密钥设置为配置项会更好，设置密钥ID如下：

```
$ git config --global user.signingkey <gpg-key-id>
```

现在你能够签署标签，从而不必每次运行`git tag`命令时定义密钥：

```
$ git tag -s <tag-name>
```

core.excludesfile

正如第二章所述，你能在项目库的.gitignore文件里头用模式来定义那些无需纳入 Git 管理的文件，这样它们不会出现在未跟踪列表， 也不会在你运行git add后被暂存。然而，如果你想用项目库之外的文件来定义那些需被忽略的文件的话，用core.excludesfile 通知 Git 该文件所处的位置，文件内容和.gitignore类似。

help.autocorrect

该配置项只在 Git 1.6.1及以上版本有效，假如你在Git 1.6中错打了一条命令，会显示：

```
$ git com
    git: 'com' is not a git-command. See 'git --help'.

    Did you mean this?
    commit
```

如果你把help.autocorrect设置成1（译注：启动自动修正），那么在只有一个命令被模糊匹配到的情况下，Git 会自动运行该命令。

Git中的着色

Git能够为输出到你终端的内容着色，以便你可以凭直观进行快速、简单地分析，有许多选项能供你使用以符合你的偏好。

color.ui

Git会按照你需要自动为大部分的输出加上颜色，你能明确地规定哪些需要着色以及怎样着色，设置color.ui为true来打开所有的默认终端着色。

```
$ git config --global color.ui true
```

设置好以后，当输出到终端时，Git 会为之加上颜色。其他的参数还有false和always，false意味着不为输出着色，而always则表明在任何情况下都要着色，即使 Git 命令被重定向到文件或管道。Git 1.5.5版本引进了此项配置，如果你拥有的版本更老，你必须对颜色有关选项各自进行详细地设置。

你会很少用到color.ui = always，在大多数情况下，如果你想在被重定向的输出中插入颜色码，你能传递--color标志给 Git 命令来迫使它这么做，color.ui = true应该是你的首选。

color.*

想要具体到哪些命令输出需要被着色以及怎样着色或者 Git 的版本很老，你就要用到和具体命令有关的颜色配置选项，它们都能被置为true、false或always：

```
color.branch
    color.diff
    color.interactive
    color.status
```

除此之外，以上每个选项都有子选项，可以被用来覆盖其父设置，以达到为输出的各个部分着色的目的。例如，让diff输出的改变信息以粗体、蓝色前景和黑色背景的形式显示：

```
$ git config --global color.diff.meta “blue black bold”
```

你能设置的颜色值如：normal、black、red、green、yellow、blue、magenta、cyan、white，正如以上例子设置的粗体属性，想要设置字体属性的话，可以选择如：bold、dim、ul、blink、reverse。

如果你想配置子选项的话，可以参考`git config`帮助页。

外部的合并与比较工具

虽然 Git 自己实现了diff,而且到目前为止你一直在使用它，但你能够用一个外部的工具替代它，除此以外，你还能用一个图形化的工具来合并和解决冲突从而不必自己手动解决。有一个不错且免费的工具可以被用来做比较和合并工作，它就是P4Merge（译注：Perforce图形化合并工具），我会展示它的安装过程。

P4Merge可以在所有主流平台上运行，现在开始大胆尝试吧。对于向你展示的例子，在Mac和Linux系统上，我会使用路径名，在Windows上，`/usr/local/bin`应该被改为你环境中的可执行路径。

下载P4Merge：

```
http://www.perforce.com/perforce/downloads/component.html
```

首先把你要运行的命令放入外部包装脚本中，我会使用Mac系统上的路径来指定该脚本的位置，在其他系统上，它应该被放置在二进制文件`p4merge`所在的目录中。创建一个merge包装脚本，名字叫作`extMerge`，让它带参数调用`p4merge`二进制文件：

```
$ cat /usr/local/bin/extMerge
    #!/bin/sh
    /Applications/p4merge.app/Contents/MacOS/p4merge $*
```

diff包装脚本首先确定传递过来7个参数，随后把其中2个传递给merge包装脚本，默认情况下， Git传递以下参数给diff：

```
path old-file old-hex old-mode new-file new-hex new-mode
```

由于你仅仅需要`old-file`和`new-file`参数，用diff包装脚本来传递它们吧。

```
$ cat /usr/local/bin/extDiff
    #!/bin/sh
    [ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

确认这两个脚本是可执行的：

```
$ sudo chmod +x /usr/local/bin/extMerge
    $ sudo chmod +x /usr/local/bin/extDiff
```

现在来配置使用你自定义的比较和合并工具吧。这需要许多自定义设置：`merge.tool`通知 Git 使用哪个合并工具；`mergetool.*.cmd`规定命令运行的方式；`mergetool.trustExitCode`会通知 Git 程序的退出是否指示合并操作成功；`diff.external`通知 Git 用什么命令做比较。因此，你能运行以下4条配置命令：

```
$ git config --global merge.tool extMerge
    $ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
    $ git config --global mergetool.trustExitCode false
    $ git config --global diff.external extDiff
```

或者直接编辑~/.gitconfig文件如下：

```
[merge]
    tool = extMerge
    [mergetool "extMerge"]
    cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
    trustExitCode = false
    [diff]
    external = extDiff
```

设置完毕后，运行diff命令：

```
$ git diff 32d1776b1^ 32d1776b1
```

命令行居然没有发现diff命令的输出，其实，Git 调用了刚刚设置的P4Merge，它看起来像图7-1这样：

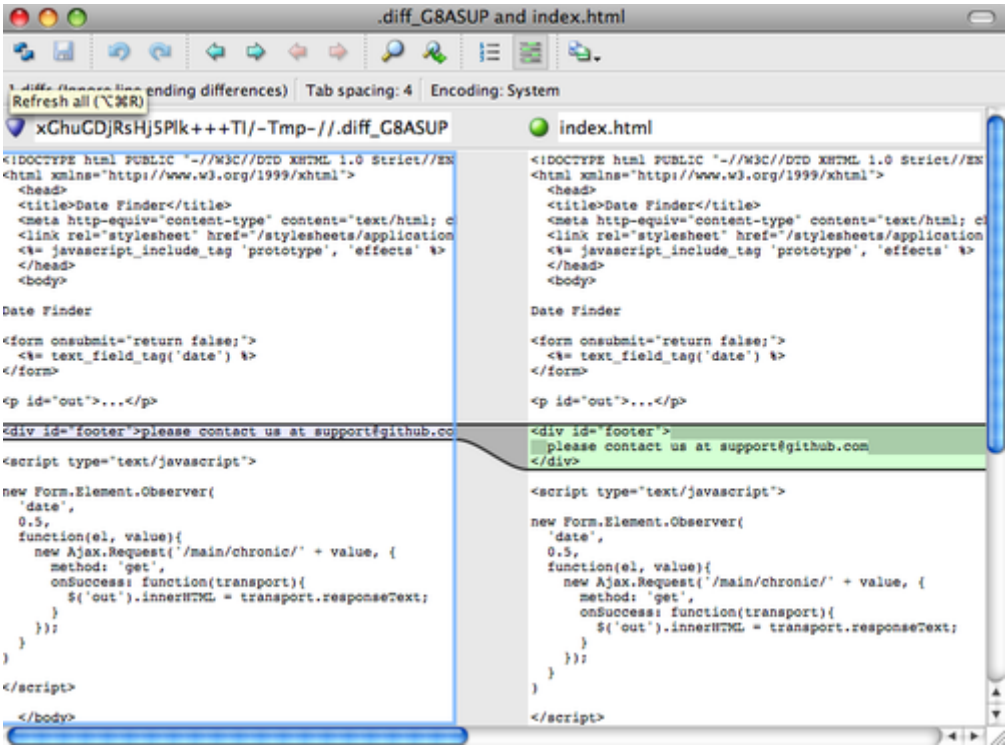

Figure 7-1. P4Merge.

当你设法合并两个分支，结果却有冲突时，运行`git mergetool`，Git 会调用P4Merge让你通过图形界面来解决冲突。

设置包装脚本的好处是你能简单地改变diff和merge工具，例如把extDiff和extMerge改成KDiff3，要做的仅仅是编辑extMerge脚本文件：

```
$ cat /usr/local/bin/extMerge
    #!/bin/sh
    /Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

现在 Git 会使用KDiff3来做比较、合并和解决冲突。

Git预先设置了许多其他的合并和解决冲突的工具，而你不必设置cmd。可以把合并工具设置为：kdiff3、opendiff、tkdiff、meld、xxdiff、emerge、vimdiff、gvimdiff。如果你不想用到KDiff3的所有功能，只是想用它来合并，那么kdiff3正符合你的要求，运行：

```
$ git config --global merge.tool kdiff3
```

如果运行了以上命令，没有设置extMerge和extDiff文件，Git 会用KDiff3做合并，让通常内设的比较工具来做比较。

格式化与空白

格式化与空白是许多开发人员在协作时，特别是在跨平台情况下，遇到的令人头疼的细小问题。由于编辑器的不同或者Windows程序员在跨平台项目中的文件行尾加入了回车换行符，一些细微的空格变化会不经意地进入大家合作的工作或提交的补丁中。不用怕，Git 的一些配置选项会帮助你解决这些问题。

core.autocrlf

假如你正在Windows上写程序，又或者你正在和其他人合作，他们在Windows上编程，而你却在其他系统上，在这些情况下，你可能会遇到行尾结束符问题。这是因为Windows使用回车和换行两个字符来结束一行，而Mac和Linux只使用换行一个字符。虽然这是小问题，但它会极大地扰乱跨平台协作。

Git可以在你提交时自动地把行结束符CRLF转换成LF，而在签出代码时把LF转换成CRLF。用core.autocrlf来打开此项功能，如果是在Windows系统上，把它设置成true，这样当签出代码时，LF会被转换成CRLF：

```
$ git config --global core.autocrlf true
```

Linux或Mac系统使用LF作为行结束符，因此你不想 Git 在签出文件时进行自动的转换；当一个以CRLF为行结束符的文件不小心被引入时你肯定想进行修正，把core.autocrlf设置成input来告诉 Git 在提交时把CRLF转换成LF，签出时不转换：

```
$ git config --global core.autocrlf input
```

这样会在Windows系统上的签出文件中保留CRLF，会在Mac和Linux系统上，包括仓库中保留LF。

如果你是Windows程序员，且正在开发仅运行在Windows上的项目，可以设置false取消此功能，把回车符记录在库中：

```
$ git config --global core.autocrlf false
```

core.whitespace

Git预先设置了一些选项来探测和修正空白问题，其4种主要选项中的2个默认被打开，另2个被关闭，你可以自由地打开或关闭它们。

默认被打开的2个选项是trailing-space和space-before-tab，trailing-space会查找每行结尾的空格，space-before-tab会查找每行开头的制表符前的空格。

默认被关闭的2个选项是indent-with-non-tab和cr-at-eol，indent-with-non-tab会查找8个以上空格（非制表符）开头的行，cr-at-eol让 Git 知道行尾回车符是合法的。

设置core.whitespace，按照你的意图来打开或关闭选项，选项以逗号分割。通过逗号分割的链中去掉选项或在选项前加-来关闭，例如，如果你想要打开除了cr-at-eol之外的所有选项：

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

当你运行git diff命令且为输出着色时，Git 探测到这些问题，因此你也许在提交前能修复它们，当你用git apply打补丁时同样也会从中受益。如果正准备运用的补丁有特别的空白问题，你可以让 Git 发警告：

```
$ git apply --whitespace=warn <patch>
```

或者让 Git 在打上补丁前自动修正此问题：

```
$ git apply --whitespace=fix <patch>
```

这些选项也能运用于衍合。如果提交了有空白问题的文件但还没推送到上流，你可以运行带有--whitespace=fix选项的rebase来让Git在重写补丁时自动修正它们。

服务器端配置

Git服务器端的配置选项并不多，但仍有一些饶有生趣的选项值得你一看。

receive.fsckObjects

Git默认情况下不会在推送期间检查所有对象的一致性。虽然会确认每个对象的有效性以及是否仍然匹配SHA-1检验和，但 Git 不会在每次推送时都检查一致性。对于 Git 来说，库或推送的文件越大，这个操作代价就相对越高，每次推送会消耗更多时间，如果想在每次推送时 Git 都检查一致性，设置 receive.fsckObjects 为true来强迫它这么做：

```
$ git config --system receive.fsckObjects true
```

现在 Git 会在每次推送生效前检查库的完整性，确保有问题的客户端没有引入破坏性的数据。

receive.denyNonFastForwards

如果对已经被推送的提交历史做衍合，继而再推送，又或者以其它方式推送一个提交历史至远程分支，且该提交历史没在这个远程分支中，这样的推送会被拒绝。这通常是个很好的禁止策略，但有时你在做衍合并确定要更新远程分支，可以在push命令后加`-f`标志来强制更新。

要禁用这样的强制更新功能，可以设置`receive.denyNonFastForwards`：

```
$ git config --system receive.denyNonFastForwards true
```

稍后你会看到，用服务器端的接收钩子也能达到同样的目的。这个方法可以做更细致的控制，例如：禁用特定的用户做强制更新。

receive.denyDeletes

规避`denyNonFastForwards`策略的方法之一就是用户删除分支，然后推回新的引用。在更新的 Git 版本中（从1.6.1版本开始），把`receive.denyDeletes`设置为true：

```
$ git config --system receive.denyDeletes true
```

这样会在推送过程中阻止删除分支和标签 — 没有用户能够这么做。要删除远程分支，必须从服务器手动删除引用文件。通过用户访问控制列表也能这么做，在本章结尾将会介绍这些有趣的方式。

7.2 Git属性

一些设置项也能被运用于特定的路径中，这样，Git 以对一个特定的子目录或子文件集运用那些设置项。这些设置项被称为 Git 属性，可以在你目录中的`.gitattributes`文件内进行设置（通常是你项目的根目录），也可以当你不想让这些属性文件和项目文件一同提交时，在`.git/info/attributes`进行设置。

使用属性，你可以对个别文件或目录定义不同的合并策略，让 Git 知道怎样比较非文本文件，在你提交或签出前让 Git 过滤内容。你将在这部分了解到能在自己的项目中使用的属性，以及一些实例。

二进制文件

你可以用 Git 属性让其知道哪些是二进制文件（以防 Git 没有识别出来），以及指示怎样处理这些文件，这点很酷。例如，一些文本文件是由机器产生的，而且无法比较，而一些二进制文件可以比较 — 你将会了解到怎样让 Git 识别这些文件。

识别二进制文件

一些文件看起来像是文本文件，但其实是作为二进制数据被对待。例如，在Mac上的Xcode项目含有一个以`.pbxproj`结尾的文件，它是由记录设置项的IDE写到磁盘的JSON数据集（纯文本javascript数据类型）。虽然技术上看它是由ASCII字符组成的文本文件，但你并不认为如此，因为它确实是一个轻量级数据库 — 如果有2人改变了它，你通常无法合并和比较内容，只有机器才能进行识别和操作，于是，你想把它当成二进制文件。

让 Git 把所有`pbxproj`文件当成二进制文件，在`.gitattributes`文件中设置如下：

```
*.pbxproj -crlf -diff
```

现在，Git 会尝试转换和修正CRLF（回车换行）问题，也不会当你在项目中运行git show或git diff时，比较不同的内容。在Git 1.6及之后的版本中，可以用一个宏代替`-crlf -diff`：

```
*.pbxproj binary
```

比较二进制文件

在Git 1.6及以上版本中，你能利用 Git 属性来有效地比较二进制文件。可以设置 Git 把二进制数据转换成文本格式，用通常的diff来比较。

这个特性很酷，而且鲜为人知，因此我会结合实例来讲解。首先，要解决的是最令人头疼的问题：对Word文档进行版本控制。很多人对Word文档又恨又爱，如果想对其进行版本控制，你可以把文件加入到 Git 库中，每次修改后提交即可。但这样做没有一点实际意义，因为运行`git diff`命令后，你只能得到如下的结果：

```
$ git diff
    diff --git a/chapter1.doc b/chapter1.doc
    index 88839c4..4afcb7c 100644
    Binary files a/chapter1.doc and b/chapter1.doc differ
```

你不能直接比较两个不同版本的Word文件，除非进行手动扫描，不是吗？ Git 属性能很好地解决此问题，把下面的行加到`.gitattributes`文件：

```
*.doc diff=word
```

当你要看比较结果时，如果文件扩展名是"doc"，Git 调用"word"过滤器。什么是"word"过滤器呢？其实就是 Git 使用`strings` 程序，把Word文档转换成可读的文本文件，之后再进行比较：

```
$ git config diff.word.textconv strings
```

现在如果在两个快照之间比较以`.doc`结尾的文件，Git 对这些文件运用"word"过滤器，在比较前把Word文件转换成文本文件。

下面展示了一个实例，我把此书的第一章纳入 Git 管理，在一个段落中加入了一些文本后保存，之后运行`git diff`命令，得到结果如下：

```
$ git diff
    diff --git a/chapter1.doc b/chapter1.doc
    index c1c8a0a..b93c9e4 100644
    --- a/chapter1.doc
    +++ b/chapter1.doc
    @@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
    re going to cover how to get it and set it up for the first time if you don
    t already have it on your system.
    In Chapter Two we will go over basic Git usage - how to use Git for the 80%
    -s going on, modify stuff and contribute changes. If the book spontaneously
    +s going on, modify stuff and contribute changes. If the book spontaneously
    +Let's see if this works.
```

Git 成功且简洁地显示出我增加的文本"Let’ s see if this works"。虽然有些瑕疵，在末尾显示了一些随机的内容，但确实可以比较了。如果你能找到或自己写个Word到纯文本的转换器的话，效果可能会更好。 `strings`可以在大部分Mac和Linux系统上运行，所以它是处理二进制格式的第一选择。

你还能用这个方法比较图像文件。当比较时，对JPEG文件运用一个过滤器，它能提炼出EXIF信息 — 大部分图像格式使用的元数据。如果你下载并安装了`exiftool`程序，可以用它参照元数据把图像转换成文本。比较的不同结果将会用文本向你展示：

```
$ echo '*.png diff=exif' >> .gitattributes
    $ git config diff.exif.textconv exiftool
```

如果在项目中替换了一个图像文件，运行`git diff`命令的结果如下：

```
diff --git a/image.png b/image.png
    index 88839c4..4afcb7c 100644
    --- a/image.png
    +++ b/image.png
    @@ -1,12 +1,12 @@
    ExifTool Version Number : 7.74
    -File Size : 70 kB
    -File Modification Date/Time : 2009:04:21 07:02:45-07:00
    +File Size : 94 kB
    +File Modification Date/Time : 2009:04:21 07:02:43-07:00
    File Type : PNG
    MIME Type : image/png
    -Image Width : 1058
    -Image Height : 889
    +Image Width : 1056
    +Image Height : 827
    Bit Depth : 8
    Color Type : RGB with Alpha
```

你会发现文件的尺寸大小发生了改变。

关键字扩展

使用SVN或CVS的开发人员经常要求关键字扩展。在 Git 中，你无法在一个文件被提交后修改它，因为 Git 会先对该文件计算校验和。然而，你可以在签出时注入文本，在提交前删除它。 Git 属性提供了2种方式这么做。

首先，你能够把blob的SHA-1校验和自动注入文件的`$Id$`字段。如果在一个或多个文件上设置了此字段，当下次你签出分支的时候，Git 用blob的SHA-1值替换那个字段。注意，这不是提交对象的SHA校验和，而是blob本身的校验和：

```
$ echo '*.txt ident' >> .gitattributes
    $ echo '$Id$' > test.txt
```

下次签出文件时，Git 入了blob的SHA值：

```
$ rm text.txt
    $ git checkout -- text.txt
    $ cat test.txt
    $Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

然而，这样的显示结果没有多大的实际意义。这个SHA的值相当地随机，无法区分日期的前后，所以，如果你在CVS或Subversion中用过关键字替换，一定会包含一个日期值。

因此，你能写自己的过滤器，在提交文件到暂存区或签出文件时替换关键字。有2种过滤器，"clean"和"smudge"。在`.gitattributes`文件中，你能对特定的路径设置一个过滤器，然后设置

处理文件的脚本，这些脚本会在文件签出前（"smudge"，见图 7-2）和提交到暂存区前（"clean"，见图7-3）被调用。这些过滤器能够做各种有趣的事。

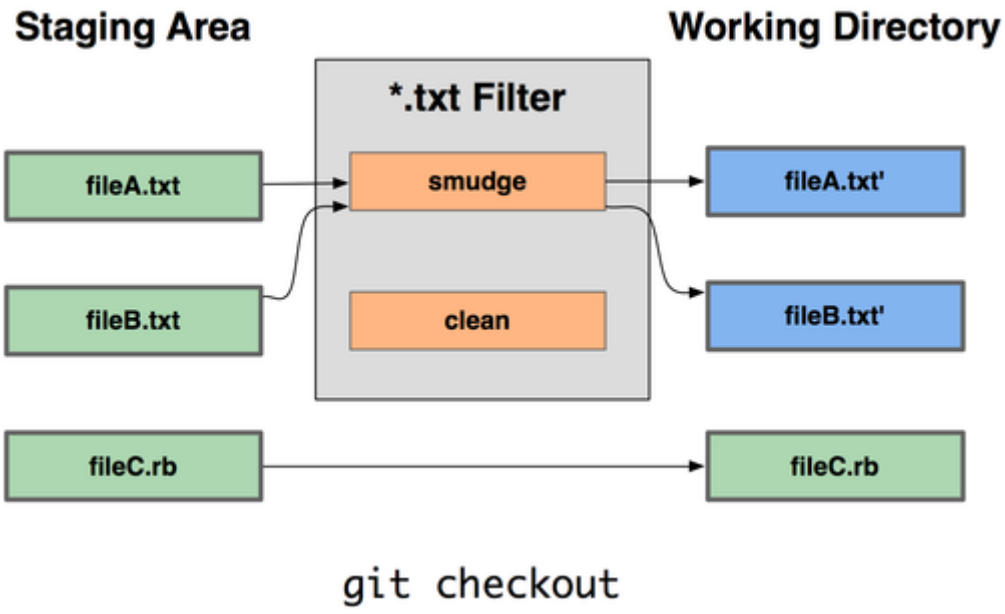

图7-2. 签出时，“smudge”过滤器被触发。

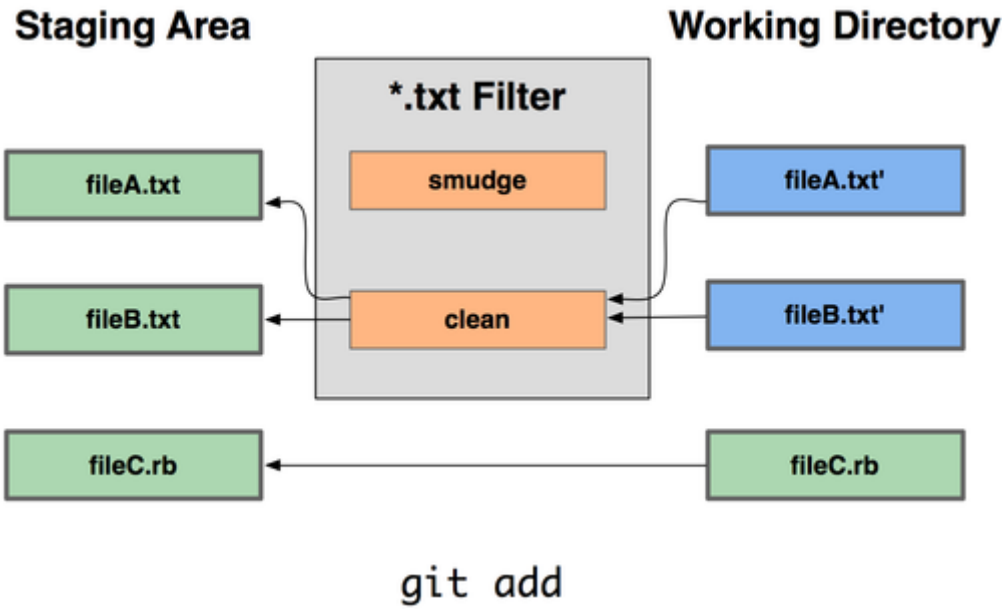

图7-3. 提交到暂存区时，“clean”过滤器被触发。

这里举一个简单的例子：在暂存前，用`indent`（缩进）程序过滤所有C源代码。在`.gitattributes`文件中设置"indent"过滤器过滤`*.c`文件：

```
*.c filter=indent
```

然后，通过以下配置，让 Git 知道"indent"过滤器在遇到"smudge"和"clean"时分别该做什么：

```
$ git config --global filter.indent.clean indent
    $ git config --global filter.indent.smudge cat
```

于是，当你暂存`*.c`文件时，`indent`程序会被触发，在把它们签出之前，`cat`程序会被触发。但`cat`程序在这里没什么实际作用。这样的组合，使C源代码在暂存前被`indent`程序过滤，非常有效。

另一个例子是类似RCS的$Date$关键字扩展。为了演示，需要一个小脚本，接受文件名参数，得到项目的最新提交日期，最后把日期写入该文件。下面用Ruby脚本来实现：

```
#! /usr/bin/env ruby
    data = STDIN.read
    last_date = `git log --pretty=format:"%ad" -1`
    puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

该脚本从git log命令中得到最新提交日期，找到文件中的所有$Date$字符串，最后把该日期填充到$Date$字符串中 — 此脚本很简单，你可以选择你喜欢的编程语言来实现。把该脚本命名为expand_date，放到正确的路径中，之后需要在 Git 中设置一个过滤器（dater），让它在签出文件时调用expand_date，在暂存文件时用Perl清除之：

```
$ git config filter.dater.smudge expand_date
    $ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

这个Perl小程序会删除$Date$字符串里多余的字符，恢复$Date$原貌。到目前为止，你的过滤器已经设置完毕，可以开始测试了。打开一个文件，在文件中输入$Date$关键字，然后设置 Git 属性：

```
$ echo '# $Date$' > date_test.txt
    $ echo 'date*.txt filter=dater' >> .gitattributes
```

如果暂存该文件，之后再签出，你会发现关键字被替换了：

```
$ git add date_test.txt .gitattributes
    $ git commit -m "Testing date expansion in Git"
    $ rm date_test.txt
    $ git checkout date_test.txt
    $ cat date_test.txt
    # $Date: Tue Apr 21 07:26:52 2009 -0700$
```

虽说这项技术对自定义应用来说很有用，但还是要小心，因为.gitattributes文件会随着项目一起提交，而过滤器（例如：dater）不会，所以，过滤器不会在所有地方都生效。当你在设计这些过滤器时要注意，即使它们无法正常工作，也要让整个项目运作下去。

导出仓库

Git属性在导出项目归档时也能发挥作用。

export-ignore

当产生一个归档时，可以设置 Git 不导出某些文件和目录。如果你不想在归档中包含一个子目录或文件，但想他们纳入项目的版本管理中，你能对应地设置export-ignore属性。

例如，在test/子目录中有一些测试文件，在项目的压缩包中包含他们是没有意义的。因此，可以增加下面这行到 Git 属性文件中：

```
test/ export-ignore
```

现在，当运行git archive来创建项目的压缩包时，那个目录不会在归档中出现。

export-subst

还能对归档做一些简单的关键字替换。在第2章中已经可以看到，可以以`--pretty=format`形式的简码在任何文件中放入`$Format:$` 字符串。例如，如果想在项目中包含一个叫作`LAST_COMMIT`的文件，当运行`git archive`时，最后提交日期自动地注入进该文件，可以这样设置：

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
    $ echo "LAST_COMMIT export-subst" >> .gitattributes
    $ git add LAST_COMMIT .gitattributes
    $ git commit -am 'adding LAST_COMMIT file for archives'
```

运行`git archive`后，打开该文件，会发现其内容如下：

```
$ cat LAST_COMMIT
    Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

合并策略

通过 Git 属性，还能对项目中的特定文件使用不同的合并策略。一个非常有用的选项就是，当一些特定文件发生冲突，Git 会尝试合并他们，而使用你这边的合并。

如果项目的一个分支有歧义或比较特别，但你想从该分支合并，而且需要忽略其中某些文件，这样的合并策略是有用的。例如，你有一个数据库设置文件database.xml，在2个分支中他们是不同的，你想合并一个分支到另一个，而不弄乱该数据库文件，可以设置属性如下：

```
database.xml merge=ours
```

如果合并到另一个分支，database.xml文件不会有合并冲突，显示如下：

```
$ git merge topic
    Auto-merging database.xml
    Merge made by recursive.
```

这样，database.xml会保持原样。

7.3 Git挂钩

和其他版本控制系统一样，当某些重要事件发生时，Git 以调用自定义脚本。有两组挂钩：客户端和服务器端。客户端挂钩用于客户端的操作，如提交和合并。服务器端挂钩用于 Git 服务器端的操作，如接收被推送的提交。你可以随意地使用这些挂钩，下面会讲解其中一些。

安装一个挂钩

挂钩都被存储在 Git 目录下的`hooks`子目录中，即大部分项目中的`.git/hooks`。 Git 默认会放置一些脚本样本在这个目录中，除了可以作为挂钩使用，这些样本本身是可以独立使用的。所有的样本都是shell脚本，其中一些还包含了Perl的脚本，不过，任何正确命名的可执行脚本都可以正常使用 — 可以用Ruby或Python，或其他。在Git 1.6版本之后，这些样本名都是以.sample结尾，因此，你必须重新命名。在Git 1.6版本之前，这些样本名都是正确的，但这些样本不是可执行文件。

把一个正确命名且可执行的文件放入 Git 目录下的`hooks`子目录中，可以激活该挂钩脚本，因此，之后他一直会被 Git 调用。随后会讲解主要的挂钩脚本。

客户端挂钩

有许多客户端挂钩，以下把他们分为：提交工作流挂钩、电子邮件工作流挂钩及其他客户端挂钩。

提交工作流挂钩

有 4个挂钩被用来处理提交的过程。`pre-commit`挂钩在键入提交信息前运行，被用来检查即将提交的快照，例如，检查是否有东西被遗漏，确认测试是否运行，以及检查代码。当从该挂钩返回非零值时，Git 放弃此次提交，但可以用`git commit --no-verify`来忽略。该挂钩可以被用来检查代码错误（运行类似lint的程序），检查尾部空白（默认挂钩是这么做的），检查新方法（译注：程序的函数）的说明。

`prepare-commit-msg`挂钩在提交信息编辑器显示之前，默认信息被创建之后运行。因此，可以有机会在提交作者看到默认信息前进行编辑。该挂钩接收一些选项：拥有提交信息的文件路径，提交类型，如果是一次修订的话，提交的SHA-1校验和。该挂钩对通常的提交来说不是很有用，只在自动产生的默认提交信息的情况下有作用，如提交信息模板、合并、压缩和修订提交等。可以和提交模板配合使用，以编程的方式插入信息。

`commit-msg`挂钩接收一个参数，此参数是包含最近提交信息的临时文件的路径。如果该挂钩脚本以非零退出，Git 放弃提交，因此，可以用来在提交通过前验证项目状态或提交信息。本章上一小节已经展示了使用该挂钩核对提交信息是否符合特定的模式。

`post-commit`挂钩在整个提交过程完成后运行，他不会接收任何参数，但可以运行`git log -1 HEAD`来获得最后的提交信息。总之，该挂钩是作为通知之类使用的。

提交工作流的客户端挂钩脚本可以在任何工作流中使用，他们经常被用来实施某些策略，但值得注意的是，这些脚本在clone期间不会被传送。可以在服务器端实施策略来拒绝不符合某些策略的推送，但这完全取决于开发者在客户端使用这些脚本的情况。所以，这些脚本对开发者是有用的，由他们自己设置和维护，而且在任何时候都可以覆盖或修改这些脚本。

E-mail工作流挂钩

有3个可用的客户端挂钩用于e-mail工作流。当运行`git am`命令时，会调用他们，因此，如果你没有在工作流中用到此命令，可以跳过本节。如果你通过e-mail接收由`git format-patch`产生的补丁，这些挂钩也许对你有用。

首先运行的是`applypatch-msg`挂钩，他接收一个参数：包含被建议提交信息的临时文件名。如果该脚本非零退出，Git 放弃此补丁。可以使用这个脚本确认提交信息是否被正确格式化，或让脚本编辑信息以达到标准化。

下一个在`git am`运行期间调用是`pre-applypatch`挂钩。该挂钩不接收参数，在补丁被运用之后运行，因此，可以被用来在提交前检查快照。你能用此脚本运行测试，检查工作树。如果有些什么遗漏，或测试没通过，脚本会以非零退出，放弃此次`git am`的运行，补丁不会被提交。

最后在`git am`运行期间调用的是`post-applypatch`挂钩。你可以用他来通知一个小组或获取的补丁的作者，但无法阻止打补丁的过程。

其他客户端挂钩

`pre-rebase`挂钩在衍合前运行，脚本以非零退出可以中止衍合的过程。你可以使用这个挂钩来禁止衍合已经推送的提交对象，Git `pre-rebase`挂钩样本就是这么做的。该样本假定next是你定义的分支名，因此，你可能要修改样本，把next改成你定义过且稳定的分支名。

在`git checkout`成功运行后，`post-checkout`挂钩会被调用。他可以用来为你的项目环境设置合适的工作目录。例如：放入大的二进制文件、自动产生的文档或其他一切你不想纳入版本控制的文件。

最后，在`merge`命令成功执行后，`post-merge`挂钩会被调用。他可以用来在 Git 无法跟踪的工作树中恢复数据，诸如权限数据。该挂钩同样能够验证在 Git 控制之外的文件是否存在，因此，当工作树改变时，你想这些文件可以被复制。

服务器端挂钩

除了客户端挂钩，作为系统管理员，你还可以使用两个服务器端的挂钩对项目实施各种类型的策略。这些挂钩脚本可以在提交对象推送到服务器前被调用，也可以在推送到服务器后被调用。推送到服务器前调用的挂钩可以在任何时候以非零退出，拒绝推送，返回错误消息给客户端，还可以如你所愿设置足够复杂的推送策略。

pre-receive 和 post-receive

处理来自客户端的推送（push）操作时最先执行的脚本就是 `pre-receive` 。它从标准输入（stdin）获取被推送引用的列表；如果它退出时的返回值不是0，所有推送内容都不会被接受。利用此挂钩脚本可以实现类似保证最新的索引中不包含非fast-forward类型的这类效果；抑或检查执行推送操作的用户拥有创建，删除或者推送的权限或者他是否对将要修改的每一个文件都有访问权限。

`post-receive` 挂钩在整个过程完结以后运行，可以用来更新其他系统服务或者通知用户。它接受与 `pre-receive` 相同的标准输入数据。应用实例包括给某邮件列表发信，通知实时整合数据的服务器，或者更新软件项目的问题追踪系统 —— 甚至可以通过分析提交信息来决定某个问题是否应该被开启，修改或者关闭。该脚本无法组织推送进程，不过客户端在它完成运行之前将保持连接状态；所以在用它作一些消耗时间的操作之前请三思。

update

update 脚本和 `pre-receive` 脚本十分类似。不同之处在于它会为推送者更新的每一个分支运行一次。假如推送者同时向多个分支推送内容，`pre-receive` 只运行一次，相比之下 update 则会为每一个更新的分支运行一次。它不会从标准输入读取内容，而是接受三个参数：索引的名字（分支），推送前索引指向的内容的 SHA-1 值，以及用户试图推送内容的 SHA-1 值。如果 update 脚本以退出时返回非零值，只有相应的那一个索引会被拒绝；其余的依然会得到更新。

7.4 Git 强制策略实例

在本节中，我们应用前面学到的知识建立这样一个Git 工作流程：检查提交信息的格式，只接受纯fast-forward内容的推送，并且指定用户只能修改项目中的特定子目录。我们将写一个客户端角本来提示开发人员他们推送的内容是否会被拒绝，以及一个服务端脚本来实际执行这些策略。

这些脚本使用 Ruby 写成，一半由于它是作者倾向的脚本语言，另外作者觉得它是最接近伪代码的脚本语言；因而即便你不使用 Ruby 也能大致看懂。不过任何其他语言也一样适用。所有 Git 自带的样例脚本都是用 Perl 或 Bash 写的。所以从这些脚本中能找到相当多的这两种语言的挂钩样例。

服务端挂钩

所有服务端的工作都在hooks（挂钩）目录的 update（更新）脚本中制定。update 脚本为每一个得到推送的分支运行一次；它接受推送目标的索引，该分支原来指向的位置，以及被推送的新内容。如果推送是通过 SSH 进行的，还可以获取发出此次操作的用户。如果设定所有操作都通过公匙授权的单一帐号（比如 " git " ）进行，就有必要通过一个 shell 包装依据公匙来判断用户的身份，并且设定环境变量来表示该用户的身份。下面假设尝试连接的用户储存在 `$USER` 环境变量里，我们的 update 脚本首先搜集一切需要的信息：

```
#!/usr/bin/env ruby

    $refname = ARGV[0]
    $oldrev = ARGV[1]
    $newrev = ARGV[2]
    $user = ENV['USER']

    puts "Enforcing Policies... \n(#{$refname}) (#{$oldrev[0,6]}) (#{$newrev[0,6]})"
```

没错，我在用全局变量。别鄙视我——这样比较利于演示过程。

指定特殊的提交信息格式

我们的第一项任务是指定每一条提交信息都必须遵循某种特殊的格式。作为演示，假定每一条信息必须包含一条形似 "ref: 1234" 这样的字符串，因为我们需要把每一次提交和项目的问题追踪系统。我们要逐一检查每一条推送上来的提交内容，看看提交信息是否包含这么一个字符串，然后，如果该提交里不包含这个字符串，以非零返回值退出从而拒绝此次推送。

把 `$newrev` 和 `$oldrev` 变量的值传给一个叫做 `git rev-list` 的 Git plumbing 命令可以获取所有提交内容的 SHA-1 值列表。`git rev-list` 基本类似 `git log` 命令，但它默认只输出 SHA-1 值而已，没有其他信息。所以要获取由 SHA 值表示的从一次提交到另一次提交之间的所有 SHA 值，可以运行：

```
$ git rev-list 538c33..d14fc7
    d14fc7c847ab946ec39590d87783c69b031bdfb7
    9f585da4401b0a3999e84113824d15245c13f0be
    234071a1be950e2a8d078e6141f5cd20c1e61ad3
    dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
    17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

截取这些输出内容，循环遍历其中每一个 SHA 值，找出与之对应的提交信息，然后用正则表达式来测试该信息包含的格式话的内容。

下面要搞定如何从所有的提交内容中提取出提交信息。使用另一个叫做 `git cat-file` 的 Git plumbing 工具可以获得原始的提交数据。我们将在第九章了解到这些 plumbing 工具的细节；现在暂时先看一下这条命令的输出：

```
$ git cat-file commit ca82a6
    tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
    parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
    author Scott Chacon <schacon@gmail.com> 1205815931 -0700
    committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

    changed the version number
```

通过 SHA-1 值获得提交内容中的提交信息的一个简单办法是找到提交的第一行，然后取从它往后的所有内容。可以使用 Unix 系统的 `sed` 命令来实现该效果：

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
    changed the version number
```

这条咒语从每一个待提交内容里提取提交信息，并且会在提取信息不符合要求的情况下退出。为了退出脚本和拒绝此次推送，返回一个非零值。整个脚本大致如下：

```
$regex = /\[ref: (\d+)\]/

    # 指定提交信息格式
    def check_message_format
    missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
    missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
    puts "[POLICY] Your message is not formatted correctly"
    exit 1
    end
    end
    end
    check_message_format
```

把这一段放在 `update` 脚本里，所有包含不符合指定规则的提交都会遭到拒绝。

实现基于用户的访问权限控制列表（ACL）系统

假设你需要添加一个使用访问权限控制列表的机制来指定哪些用户对项目的哪些部分有推送权限。某些用户具有全部的访问权，其他人只对某些子目录或者特定的文件具有推送权限。要搞定这一点，所有的规则将被写入一个位于服务器的原始 Git 仓库的 `acl` 文件。我们让 `update` 挂钩检阅这些规则，审视推送的提交内容中需要修改的所有文件，然后决定执行推送的用户是否对所有这些文件都有权限。

我们首先要创建这个列表。这里使用的格式和 CVS 的 ACL 机制十分类似：它由若干行构成，第一项内容是 `avail` 或者 `unavail`，接着是逗号分隔的规则生效用户列表，最后一项是规则生效的目录（空白表示开放访问）。这些项目由 | 字符隔开。

下例中，我们指定几个管理员，几个对 `doc` 目录具有权限的文档作者，以及一个对 `lib` 和 `tests` 目录具有权限的开发人员，相应的 ACL 文件如下：

```
avail|nickh,pjhyett,defunkt,tpw
    avail|usinclair,cdickens,ebronte|doc
    avail|schacon|lib
    avail|schacon|tests
```

首先把这些数据读入你编写的数据结构。本例中，为保持简洁，我们暂时只实现 `avail` 的规则（译注：也就是省略了 `unavail` 部分）。下面这个方法生成一个关联数组，它的主键是用户名，值是一个该用户有写权限的所有目录组成的数组：

```
def get_acl_access_data(acl_file)
    # read in ACL data
    acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
    access = {}
    acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
    access[user] ||= []
    access[user] << path
    end
```

```
    end
    access
    end
```

针对之前给出的 ACL 规则文件，这个 `get_acl_access_data` 方法返回的数据结构如下：

```
{"defunkt"=>[nil],
    "tpw"=>[nil],
    "nickh"=>[nil],
    "pjhyett"=>[nil],
    "schacon"=>["lib", "tests"],
    "cdickens"=>["doc"],
    "usinclair"=>["doc"],
    "ebronte"=>["doc"]}
```

搞定了用户权限的数据，下面需要找出哪些位置将要被提交的内容修改，从而确保试图推送的用户对这些位置有全部的权限。

使用 `git log` 的 `--name-only` 选项（在第二章里简单的提过）我们可以轻而易举的找出一次提交里修改的文件：

```
$ git log -1 --name-only --pretty=format:'' 9f585d

    README
    lib/test.rb
```

使用 `get_acl_access_data` 返回的 ACL 结构来一一核对每一次提交修改的文件列表，就能找出该用户是否有权限推送所有的提交内容:

```
# 仅允许特定用户修改项目中的特定子目录
    def check_directory_perms
    access = get_acl_access_data('acl')

    # 检查是否有人在向他没有权限的地方推送内容
    new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
    new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || # 用户拥有完全访问权限
    (path.index(access_path) == 0) # 或者对此位置有访问权限
    has_file_access = true
    end
    end
    if !has_file_access
    puts "[POLICY] You do not have access to push to #{path}"
    exit 1
    end
    end
    end
    end

    check_directory_perms
```

以上的大部分内容应该都比较容易理解。通过 `git rev-list` 获取推送到服务器内容的提交列表。然后，针对其中每一项，找出它试图修改的文件然后确保执行推送的用户对这些文件具有权限。一个不太容易理解的 Ruby 技巧石 `path.index(access_path) ==0` 这句，它的返回真值如果路径以 `access_path` 开头——这是为了确保 `access_path` 并不是只在允许的路径之一，而是所有准许全选的目录都在该目录之下。

现在你的用户没法推送带有不正确的提交信息的内容，也不能在准许他们访问范围之外的位置做出修改。

只允许 Fast-Forward 类型的推送

剩下的最后一项任务是指定只接受 fast-forward 的推送。在 Git 1.6 或者更新版本里，只需要设定 `receive.denyDeletes` 和 `receive.denyNonFastForwards` 选项就可以了。但是通过挂钩的实现可以在旧版本的 Git 上工作，并且通过一定的修改它它可以做到只针对某些用户执行，或者更多以后可能用的到的规则。

检查这一项的逻辑是看看提交里是否包含从旧版本里能找到但在新版本里却找不到的内容。如果没有，那这是一次纯 fast-forward 的推送；如果有，那我们拒绝此次推送：

```
# 只允许纯 fast-forward 推送
    def check_fast_forward
    missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
    missed_ref_count = missed_refs.split("\n").size
    if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
    end
    end

    check_fast_forward
```

一切都设定好了。如果现在运行 `chmod u+x .git/hooks/update` —— 修改包含以上内容文件的权限，然后尝试推送一个包含非 fast-forward 类型的索引，会得到一下提示：

```
$ git push -f origin master
    Counting objects: 5, done.
    Compressing objects: 100% (3/3), done.
    Writing objects: 100% (3/3), 323 bytes, done.
    Total 3 (delta 1), reused 0 (delta 0)
    Unpacking objects: 100% (3/3), done.
    Enforcing Policies...
    (refs/heads/master) (8338c5) (c5b616)
    [POLICY] Cannot push a non-fast-forward reference
    error: hooks/update exited with error code 1
    error: hook declined to update refs/heads/master
    To git@gitserver:project.git
    ! [remote rejected] master -> master (hook declined)
    error: failed to push some refs to 'git@gitserver:project.git'
```

这里有几个有趣的信息。首先，我们可以看到挂钩运行的起点：

```
Enforcing Policies...
    (refs/heads/master) (fb8c72) (c56860)
```

注意这是从 update 脚本开头输出到标准你输出的。所有从脚本输出的提示都会发送到客户端，这点很重要。

下一个值得注意的部分是错误信息。

```
[POLICY] Cannot push a non fast-forward reference
    error: hooks/update exited with error code 1
    error: hook declined to update refs/heads/master
```

第一行是我们的脚本输出的，在往下是 Git 在告诉我们 update 脚本退出时返回了非零值因而推送遭到了拒绝。最后一点：

```
To git@gitserver:project.git
    ! [remote rejected] master -> master (hook declined)
    error: failed to push some refs to 'git@gitserver:project.git'
```

我们将为每一个被挂钩拒之门外的索引受到一条远程信息，解释它被拒绝是因为一个挂钩的原因。

而且，如果那个 ref 字符串没有包含在任何的提交里，我们将看到前面脚本里输出的错误信息：

```
[POLICY] Your message is not formatted correctly
```

又或者某人想修改一个自己不具备权限的文件然后推送了一个包含它的提交，他将看到类似的提示。比如，一个文档作者尝试推送一个修改到 `lib` 目录的提交，他会看到

```
[POLICY] You do not have access to push to lib/test.rb
```

全在这了。从这里开始，只要 `update` 脚本存在并且可执行，我们的仓库永远都不会遭到回转或者包含不符合要求信息的提交内容，并且用户都被锁在了沙箱里面。

客户端挂钩

这种手段的缺点在于用户推送内容遭到拒绝后几乎无法避免的抱怨。辛辛苦苦写成的代码在最后时刻惨遭拒绝是十分悲剧切具迷惑性的；更可怜的是他们不得不修改提交历史来解决问题，这怎么也算不上王道。

逃离这种两难境地的法宝是给用户一些客户端的挂钩，在他们作出可能悲剧的事情的时候给以警告。然后呢，用户们就能在提交--问题变得更难修正之前解除隐患。由于挂钩本身不跟随克隆的项目副本分发，所以必须通过其他途径把这些挂钩分发到用户的 .git/hooks 目录并设为可执行文件。虽然可以在相同或单独的项目内 容里加入并分发它们，全自动的解决方案是不存在的。

首先，你应该在每次提交前核查你的提交注释信息，这样你才能确保服务器不会因为不合条件的提交注释信息而拒绝你的更改。为了达到这个目的，你可以增加'commit-msg'挂钩。如果你使用该挂钩来阅读作为第一个参数传递给git的提交注释信息，并且与规定的模式作对比，你就可以使git在提交注释信息不符合条件的情况下，拒绝执行提交。

```
#!/usr/bin/env ruby
    message_file = ARGV[0]
    message = File.read(message_file)

    $regex = /\[ref: (\d+)\]/

    if !$regex.match(message)
    puts "[POLICY] Your message is not formatted correctly"
    exit 1
    end
```

如果这个脚本放在这个位置 (`.git/hooks/commit-msg`) 并且是可执行的, 并且你的提交注释信息不是符合要求的，你会看到：

```
$ git commit -am 'test'
    [POLICY] Your message is not formatted correctly
```

在这个实例中，提交没有成功。然而如果你的提交注释信息是符合要求的，git会允许你提交：

```
$ git commit -am 'test [ref: 132]'
    [master e05c914] test [ref: 132]
    1 files changed, 1 insertions(+), 0 deletions(-)
```

接下来我们要保证没有修改到 ACL 允许范围之外的文件。加入你的 .git 目录里有前面使用过的 ACL 文件，那么以下的 pre-commit 脚本将把里面的规定执行起来：

```
#!/usr/bin/env ruby

    $user = ENV['USER']

    # [ insert acl_access_data method from above ]

    # 只允许特定用户修改项目重特定子目录的内容
    def check_directory_perms
    access = get_acl_access_data('.git/acl')

    files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
    files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
    has_file_access = true
    end
    if !has_file_access
    puts "[POLICY] You do not have access to push to #{path}"
    exit 1
    end
    end
    end

    check_directory_perms
```

这和服务端的脚本几乎一样，除了两个重要区别。第一，ACL 文件的位置不同，因为这个脚本在当前工作目录运行，而非 Git 目录。ACL 文件的目录必须从

```
access = get_acl_access_data('acl')
```

修改成：

```
access = get_acl_access_data('.git/acl')
```

另一个重要区别是获取被修改文件列表的方式。在服务端的时候使用了查看提交纪录的方式，可是目前的提交都还没被记录下来呢，所以这个列表只能从暂存区域获取。和原来的

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

不同，现在要用

```
files_modified = `git diff-index --cached --name-only HEAD`
```

不同的就只有这两点——除此之外，该脚本完全相同。一个小陷阱在于它假设在本地运行的账户和推送到远程服务端的相同。如果这二者不一样，则需要手动设置一下 `$user` 变量。

最后一项任务是检查确认推送内容中不包含非 fast-forward 类型的索引，不过这个需求比较少见。要找出一个非 fast-forward 类型的索引，要么衍合超过某个已经推送过的提交，要么从本地不同分支推送到远程相同的分支上。

既然服务器将给出无法推送非 fast-forward 内容的提示，而且上面的挂钩也能阻止强制的推送，唯一剩下的潜在问题就是衍合一次已经推送过的提交内容。

下面是一个检查这个问题的 pre-rabase 脚本的例子。它获取一个所有即将重写的提交内容的列表，然后检查它们是否在远程的索引里已经存在。一旦发现某个提交可以从远程索引里衍变过来，它就放弃衍合操作：

```
#!/usr/bin/env ruby

    base_branch = ARGV[0]
    if ARGV[1]
    topic_branch = ARGV[1]
    else
    topic_branch = "HEAD"
    end

    target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
    remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

    target_shas.each do |sha|
    remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
    puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
    exit 1
    end
    end
    end
```

这个脚本利用了一个第六章“修订版本选择”一节中不曾提到的语法。通过这一句可以获得一个所有已经完成推送的提交的列表：

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

`SHA^@` 语法解析该次提交的所有祖先。这里我们从检查远程最后一次提交能够衍变获得但从所有我们尝试推送的提交的 SHA 值祖先无法衍变获得的提交内容——也就是 fast-forward 的内容。

这个解决方案的硬伤在于它有可能很慢而且常常没有必要——只要不用 `-f` 来强制推送，服务器会自动给出警告并且拒绝推送内容。然而，这是个不错的练习而且理论上能帮助用户避免一次将来不得不折回来修改的衍合操作。

7.5 总结

你已经见识过绝大多数通过自定义 Git 客户端和服务端来来适应自己工作流程和项目内容的方式了。无论你创造出了什么样的工作流程，Git 都能用的顺手。

下一节上一节首页（目录）| 返回 码云