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脚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]})"

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 技巧是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) (8338c5) (c5b616)

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且具有迷惑性的；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。