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

你可以使用 Git 属性来有效地比较两个二进制文件（binary files，译注：指非文本文件）。那么第一步要做的是，告诉 Git 怎么把你的二进制文件转化为纯文本格式，从而让普通的 diff 命令可以进行文本对比。但是，我们怎么把*二进制文件*转化为文本呢？最好的解决方法是找到一个转换工具帮助我们进行转化。但是，大部分的二进制文件不能表示为可读的文本，例如语音文件就很难转化为文本文件。如果你遇到这些情况，比较简单的解决方法是从这些二进制文件中获取元数据。虽然这些元数据并不能完全描述一个二进制文件，但大多数情况下，都是能够概括文件情况的。

下面，我们将会展示，如何使用转化工具进行二进制文件的比较。

边注：有一些二进制文件虽然包含文字，但是却难以转换。（译注：例如 Word 文档。）在这些情况，你可以尝试使用 strings 工具来获取其中的文字。但如果当这些文档包含 UTF-16 编码，或者其他代码页（codepages），strings 也可能无补于事。strings 在大部分的 Mac 和 Linux 下都有安装。当遇到有二进制文件需要转换的时候，你可以试试这个工具。

Word文档

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 catdoc

这个命令会在你的 .git/config 文件中增加一节：

[diff "word"]

textconv = catdoc

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

@@ -128,7 +128,7 @@ and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use

and yet retain these initial qualities. It’s incredibly fast, it’s

very efficient with large projects, and it has an incredible branching

-system for non-linear development.

+system for non-linear development (See Chapter 3).

Git 成功且简洁地显示出我增加的文本"(See Chapter 3)"。工作的很完美！

OpenDocument文本文档

我们用于处理Word文档（\*.doc）的方法同样适用于处理OpenOffice.org创建的OpenDocument文本文档（\*.odt）。

把下面这行添加到.gitattributes文件：

\*.odt diff=odt

然后在.git/config 文件中设置odt过滤器：

[diff "odt"]

binary = true

textconv = /usr/local/bin/odt-to-txt

OpenDocument文档实际上是多个文件（包括一个XML文件和表格、图片等文件）的压缩包。我们需要写一个脚本来提取其中纯文本格式的内容。创建一个文件/usr/local/bin/odt-to-txt（你也可以放到其他目录下），写入下面内容：

#! /usr/bin/env perl

# Simplistic OpenDocument Text (.odt) to plain text converter.

# Author: Philipp Kempgen

if (! defined($ARGV[0])) {

print STDERR "No filename given!\n";

print STDERR "Usage: $0 filename\n";

exit 1;

}

my $content = '';

open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;

{

local $/ = undef; # slurp mode

$content = <$fh>;

}

close $fh;

$\_ = $content;

s/<text:span\b[^>]\*>//g; # remove spans

s/<text:h\b[^>]\*>/\n\n\*\*\*\*\* /g; # headers

s/<text:list-item\b[^>]\*>\s\*<text:p\b[^>]\*>/\n -- /g; # list items

s/<text:list\b[^>]\*>/\n\n/g; # lists

s/<text:p\b[^>]\*>/\n /g; # paragraphs

s/<[^>]+>//g; # remove all XML tags

s/\n{2,}/\n\n/g; # remove multiple blank lines

s/\A\n+//; # remove leading blank lines

print "\n", $\_, "\n\n";

然后把它设为可执行文件

chmod +x /usr/local/bin/odt-to-txt

现在git diff命令就可以显示.odt文件的变更了。

图像文件

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 : 2009:04:17 10:12:35-07:00

+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 test.txt

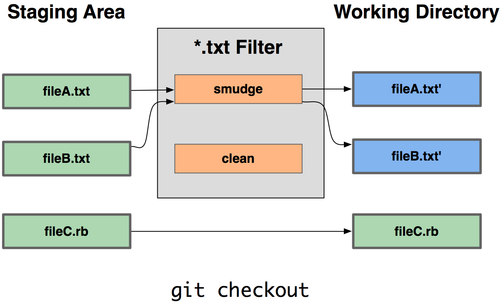
$ git checkout -- test.txt

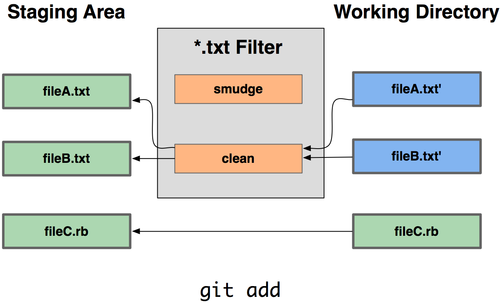
$ cat test.txt

$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。