使用 Git 调试

Git 同样提供了一些工具来帮助你调试项目中遇到的问题。由于 Git 被设计为可应用于几乎任何类型的项目，这些工具是通用型，但是在遇到问题时可以经常帮助你查找缺陷所在。

文件标注

如果你在追踪代码中的缺陷想知道这是什么时候为什么被引进来的，文件标注会是你的最佳工具。它会显示文件中对每一行进行修改的最近一次提交。因此，如果你发现自己代码中的一个方法存在缺陷，你可以用git blame来标注文件，查看那个方法的每一行分别是由谁在哪一天修改的。下面这个例子使用了-L选项来限制输出范围在第12至22行：

$ git blame -L 12,22 simplegit.rb

^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')

^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13) command("git show #{tree}")

^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end

^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)

9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')

79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17) command("git log #{tree}")

9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end

9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)

42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)

42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21) command("git blame #{path}")

42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end

请注意第一个域里是最后一次修改该行的那次提交的 SHA-1 值。接下去的两个域是从那次提交中抽取的值——作者姓名和日期——所以你可以方便地获知谁在什么时候修改了这一行。在这后面是行号和文件的内容。请注意^4832fe2提交的那些行，这些指的是文件最初提交的那些行。那个提交是文件第一次被加入这个项目时存在的，自那以后未被修改过。这会带来小小的困惑，因为你已经至少看到了Git使用^来修饰一个提交的SHA值的三种不同的意义，但这里确实就是这个意思。

另一件很酷的事情是在 Git 中你不需要显式地记录文件的重命名。它会记录快照然后根据现实尝试找出隐式的重命名动作。这其中有一个很有意思的特性就是你可以让它找出所有的代码移动。如果你在git blame后加上-C，Git会分析你在标注的文件然后尝试找出其中代码片段的原始出处，如果它是从其他地方拷贝过来的话。最近，我在将一个名叫GITServerHandler.m的文件分解到多个文件中，其中一个是GITPackUpload.m。通过对GITPackUpload.m执行带-C参数的blame命令，我可以看到代码块的原始出处：

$ git blame -C -L 141,153 GITPackUpload.m

f344f58d GITServerHandler.m (Scott 2009-01-04 141)

f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC

f344f58d GITServerHandler.m (Scott 2009-01-04 143) {

70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"GATHER COMMI

ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)

ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString \*parentSha;

ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit \*commit = [g

ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)

ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) //NSLog(@"GATHER COMMI

ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)

56ef2caf GITServerHandler.m (Scott 2009-01-05 151) if(commit) {

56ef2caf GITServerHandler.m (Scott 2009-01-05 152) [refDict setOb

56ef2caf GITServerHandler.m (Scott 2009-01-05 153)

这真的非常有用。通常，你会把你拷贝代码的那次提交作为原始提交，因为这是你在这个文件中第一次接触到那几行。Git可以告诉你编写那些行的原始提交，即便是在另一个文件里。

二分查找

标注文件在你知道问题是哪里引入的时候会有帮助。如果你不知道，并且自上次代码可用的状态已经经历了上百次的提交，你可能就要求助于bisect命令了。bisect会在你的提交历史中进行二分查找来尽快地确定哪一次提交引入了错误。

例如你刚刚推送了一个代码发布版本到产品环境中，对代码为什么会表现成那样百思不得其解。你回到你的代码中，还好你可以重现那个问题，但是找不到在哪里。你可以对代码执行bisect来寻找。首先你运行git bisect start启动，然后你用git bisect bad来告诉系统当前的提交已经有问题了。然后你必须告诉bisect已知的最后一次正常状态是哪次提交，使用git bisect good [good\_commit]：

$ git bisect start

$ git bisect bad

$ git bisect good v1.0

Bisecting: 6 revisions left to test after this

[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo

Git 发现在你标记为正常的提交(v1.0)和当前的错误版本之间有大约12次提交，于是它检出中间的一个。在这里，你可以运行测试来检查问题是否存在于这次提交。如果是，那么它是在这个中间提交之前的某一次引入的；如果否，那么问题是在中间提交之后引入的。假设这里是没有错误的，那么你就通过git bisect good来告诉 Git 然后继续你的旅程：

$ git bisect good

Bisecting: 3 revisions left to test after this

[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing

现在你在另外一个提交上了，在你刚刚测试通过的和一个错误提交的中点处。你再次运行测试然后发现这次提交是错误的，因此你通过git bisect bad来告诉Git：

$ git bisect bad

Bisecting: 1 revisions left to test after this

[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table

这次提交是好的，那么 Git 就获得了确定问题引入位置所需的所有信息。它告诉你第一个错误提交的 SHA-1 值并且显示一些提交说明以及哪些文件在那次提交里修改过，这样你可以找出缺陷被引入的根源：

$ git bisect good

b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit

commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04

Author: PJ Hyett <pjhyett@example.com>

Date: Tue Jan 27 14:48:32 2009 -0800

secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730

f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M config

当你完成之后，你应该运行git bisect reset来重设你的HEAD到你开始前的地方，否则你会处于一个诡异的地方：

$ git bisect reset

这是个强大的工具，可以帮助你检查上百的提交，在几分钟内找出缺陷引入的位置。事实上，如果你有一个脚本会在工程正常时返回0，错误时返回非0的话，你可以完全自动地执行git bisect。首先你需要提供已知的错误和正确提交来告诉它二分查找的范围。你可以通过bisect start命令来列出它们，先列出已知的错误提交再列出已知的正确提交：

$ git bisect start HEAD v1.0

$ git bisect run test-error.sh

这样会自动地在每一个检出的提交里运行test-error.sh直到Git找出第一个破损的提交。你也可以运行像make或者make tests或者任何你所拥有的来为你执行自动化的测试。