2015-11-08 发布

git merge是怎样判定冲突的？

在解决git merge的冲突时，有时我总忍不住吐槽git实在太不智能了，明明仅仅是往代码里面插入几行，没想到合并就失败了，只能手工去一个个确认。真不知道git的合并冲突是怎么判定的。

在一次解决了涉及几十个文件的合并冲突后（整整花了我一个晚上和一个早上的时间！），我终于下定决心，去看一下git merge代码里面冲突判定的具体实现。正所谓冤有头债有主，至少下次遇到同样的问题时就可以知道自己栽在谁的手里了。于是就有了这样一篇文章，讲讲git merge内部的冲突判定机制。

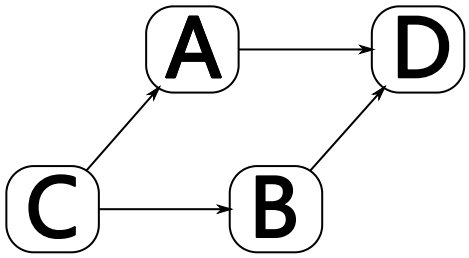
recursive three-way merge和ancestor

git的源码
先用 `merge` 作关键字搜索，看看涉及的相关代码。
找了一段时间，找到了git merge的时候，比较待合并文件的函数入口：ll_merge。另外还有一份文档，它也指出 `ll_merge` 正是合并实现的入口。

从函数签名可以看到，mmfile_t应该就代表了待合并的文件。有趣的是，这里待合并的文件并不是两份，而是三份。

```
int ll_merge(mmbuffer_t *result_buf,
             const char *path,
             mmfile_t *ancestor, const char *ancestor_label,
             mmfile_t *ours, const char *our_label,
             mmfile_t *theirs, const char *their_label,
             const struct ll_merge_options *opts)
```

看过 `git help merge` 的读者应该知道， `ours` 表示当前分支， `theirs` 表示待合并分支。看得出来，这个函数就是把某个文件在不同分支上的版本合并在一起。那么 `ancestor` 又是位于哪个分支呢？倒过来从调用方开始阅读代码，可以看出大体的流程是这样的， `git merge` 会找出三个commit，然后对每个待合并的文件调用 `ll_merge` ，生成最终的合并结果。按注释的说法， `ancestor` 是后面两个commit（ `ours` 和 `theirs` ）的公共祖先（ancestor）。另外前面提到的文档也说明，git合并的时候使用的是 `recursive three-way merge` 。



关于 `recursive three-way merge` ， wikipedia上有个相关的介绍#Recursive_three-way_merge)。就是在合并的时候，将ours，theirs和ancestor三个版本的文件进行比较，获取ours和ancestor的diff，以及theirs和ancestor的diff，这样做能够发现两个不同的分支到底做了哪些改动。毕竟后面git需要判定冲突的内容，如果没有原初版本的信息，只是简单地比较两个文件，是做不到的。

鉴于我的目标是发掘git判定冲突的机制，所以没有去看git里面查找ancestor的实现。不过只需肉眼在图形化界面里瞅上一眼，就可以找到ancestor commit。（比如在gitlab的network界面中，回溯两个分支的commit线，一直到岔路口）

有一点需要注意的是，revert一个commit不会改变它的ancestor。所谓的revert，只是在当前commit的上面添加了新的undo commit，并没有改变“岔路口”的位置。不要想当然地认为，revert之后ancestor就变成上一个commit的ancestor了。尤其是在revert merge commit的时候，总是容易忘掉这个事实。假如你revert了一个merge commit，在重新merge的时候，git所参照的ancestor将不是merge之前的ancestor，而是revert之后的ancestor。于是就掉到坑里去了。建议所有读者都看一下git官方对于 `revert merge commit` 潜在后果的说法：https://github.com/git/git/blob/master/Documentation/howto/revert-a-faulty-merge.txt
结论是，如果一个merge commit引入的bug容易修复，请不要轻易revert一个merge commit。

剖析xdiff

从 `ll_merge` 往下追，可以看到后面出了一条旁路： `ll_binary_merge` 。这个函数专门处理bin类型文件的合并。它的实现简单粗暴，如果你没有指定合并策略（theris或ours），直接报Cannot merge binary files错误。看来在git看来，二进制文件并没有diff的价值。

主路径从 `ll_xdl_merge` 到 `xdl_merge` ，进到一个叫xdiff的库中。终于找到git merge的具体实现了。

平心而论，xdiff的代码风格十分糟糕，不仅注释太少，而且结构体成员变量居然使用类似i1、i2这样的命名，看得我头昏脑胀、心烦意燥。

吐槽结束，先讲下 `xdl_merge` 的流程。 `xdl_merge` 做了下面四件事：

1. 由 `xdl_do_diff` 完成two-way diff（ours和ancestor，theirs和ancestor）,生成修改记录，存储到 `xdfenv_t` 中。
2. `xdl_change_compact` 压缩相邻的修改记录，再用 `xdl_build_script` 建立xdchange_t链表，记录双方修改。xdchange_t主要包括了修改的起始行号和修改范围。
3. 这时候分三种情况，其中两种是只有一方有修改（只有ours或theirs一条链表），直接退出。最后一种是双方都有修改，需要合并修改记录。由于修改记录是按行号有序排列的，所以直接合并两个链表。修改记录如果没有重叠部分，按先后顺序标记为我方修改/他方修改。如果发生了重叠，就表示发生了冲突。之后会重新过一遍两个待合并链表，对于那些标记为冲突的部分，比较它们是否相等的，如果是，标记为双方修改。
4. 由 `xdl_fill_merge_buffer` 输出合并结果。如果有冲突，调用 `fill_conflict_hunk` 输出冲突情况。如果没有冲突（标记为我方修改/他方修改/双方修改），则合并ancestor的原内容和修改记录，按标记的类型取修改后的内容，并输出。

输出冲突情况的代码位于 `fill_conflict_hunk` 中。它的实现很简单，毕竟此时我们已经有了双方修改的内容，现在只需要同时输出冲突内容，供用户取舍。（这便是那次花了一个晚上和一个早上改掉的冲突的源头，凶手就是你，哼）。

输出格式恐怕大家都很熟悉。该函数会先打印若干个 `<` ，个数由DEFAULT_CONFLICT_MARKER_SIZE决定，也即是7个。然后是ours分支名。接着输出我方的修改，然后输出若干个 `=` 。最后是他方的修改，以及若干个 `>` 。这个就是折磨人的合并冲突了：

```
<<<<<<< HEAD
3
=======
2
>>>>>>> branch1
```

总结

git merge的冲突判定机制如下：先寻找两个commit的公共祖先，比较同一个文件分别在ours和theirs下对于公共祖先的差异，然后合并这两组差异。如果双方同时修改了一处地方且修改内容不同，就判定为合并冲突，依次输出双方修改的内容。