rerere

Rerere ======

This document describes the rerere logic.

Conflict normalization ----------

To ensure recorded conflict resolutions can be looked up in the rerere database, even when branches are merged in a different order, different branches are merged that result in the same conflict, or when different conflict style settings are used, rerere normalizes the conflicts before writing them to the rerere database.

Different conflict styles and branch names are normalized by stripping the labels from the conflict markers, and removing the common ancestor version from the `diff3` or `zdiff3` conflict styles. Branches that are merged in different order are normalized by sorting the conflict hunks. More on each of those steps in the following sections.

Once these two normalization operations are applied, a conflict ID is calculated based on the normalized conflict, which is later used by rerere to look up the conflict in the rerere database.

Removing the common ancestor version
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Say we have three branches AB, AC and AC2. The common ancestor of these branches has a file with a line containing the string "A" (for brevity this is called "line A" in the rest of the document). In branch AB this line is changed to "B", in AC, this line is changed to "C", and branch AC2 is forked off of AC, after the line was changed to "C".

Forking a branch ABAC off of branch AB and then merging AC into it, we get a conflict like the following:

```
<<<<<<< HEAD
B
=======
C
>>>>>>> AC
```

Doing the analogous with AC2 (forking a branch ABAC2 off of branch AB and then merging branch AC2 into it), using the diff3 or zdiff3 conflict style, we get a conflict like the following:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
<<<<<<< HEAD
B
||||||| merged common ancestors
A
=======
C
>>>>>>> AC2
```

By resolving this conflict, to leave line D, the user declares:

```
After examining what branches AB and AC did, I believe that making
line A into line D is the best thing to do that is compatible with
what AB and AC wanted to do.
```

As branch AC2 refers to the same commit as AC, the above implies that this is also compatible with what AB and AC2 wanted to do.

By extension, this means that rerere should recognize that the above conflicts are the same. To do this, the labels on the conflict markers are stripped, and the common ancestor version is removed. The above examples would both result in the following normalized conflict:

```
<<<<<<<
B
=======
C
>>>>>>>
```

Sorting hunks ~~~~~~~~~~~~~~

As before, let’s imagine that a common ancestor had a file with line A in its early part, and line X in its late part. And then four branches are forked that do these things:

- AB: changes A to B
- AC: changes A to C
- XY: changes X to Y
- XZ: changes X to Z

Now, forking a branch ABAC off of branch AB and then merging AC into it, and forking a branch ACAB off of branch AC and then merging AB into it, would yield the conflict in a different order. The former would say "A became B or C, what now?" while the latter would say "A became C or B, what now?"

As a reminder, the act of merging AC into ABAC and resolving the conflict to leave line D means that the user declares:

```
After examining what branches AB and AC did, I believe that
making line A into line D is the best thing to do that is
compatible with what AB and AC wanted to do.
```

So the conflict we would see when merging AB into ACAB should be resolved the same way—it is the resolution that is in line with that declaration.

Imagine that similarly previously a branch XYXZ was forked from XY, and XZ was merged into it, and resolved "X became Y or Z" into "X became W".

Now, if a branch ABXY was forked from AB and then merged XY, then ABXY would have line B in its early part and line Y in its later part. Such a merge would be quite clean. We can construct 4 combinations using these four branches ((AB, AC) x (XY, XZ)).

Merging ABXY and ACXZ would make "an early A became B or C, a late X became Y or Z" conflict, while merging ACXY and ABXZ would make "an early A became C or B, a late X became Y or Z". We can see there are 4 combinations of ("B or C", "C or B") x ("X or Y", "Y or X").

By sorting, the conflict is given its canonical name, namely, "an early part became B or C, a late part became X or Y", and whenever any of these four patterns appear, and we can get to the same conflict and resolution that we saw earlier.

Without the sorting, we'd have to somehow find a previous resolution from combinatorial explosion.

Conflict ID calculation ~~~~~~~~~~~~~~~~~~~~~~~~

Once the conflict normalization is done, the conflict ID is calculated as the sha1 hash of the conflict hunks appended to each other, separated by <NUL> characters. The conflict markers are stripped out before the sha1 is calculated. So in the example above, where we merge branch AC which changes line A to line C, into branch AB, which changes line A to line C, the conflict ID would be SHA1('B<NUL>C<NUL>').

If there are multiple conflicts in one file, the sha1 is calculated the same way with all hunks appended to each other, in the order in which they appear in the file, separated by a <NUL> character.

Nested conflicts ~~~~~~~~~~~~~~~~~

Nested conflicts are handled very similarly to "simple" conflicts. Similar to simple conflicts, the conflict is first normalized by stripping the labels from conflict markers, stripping the common ancestor version, and sorting the conflict hunks, both for the outer and the inner conflict. This is done recursively, so any number of nested conflicts can be handled.

Note that this only works for conflict markers that "cleanly nest". If there are any unmatched conflict markers, rerere will fail to handle the conflict and record a conflict resolution.

The only difference is in how the conflict ID is calculated. For the inner conflict, the conflict markers themselves are not stripped out before calculating the sha1.

Say we have the following conflict for example:

```
<<<<<<< HEAD
1
=======
<<<<<<< HEAD
3
=======
2
>>>>>>> branch-2
>>>>>>> branch-3~
```

After stripping out the labels of the conflict markers, and sorting the hunks, the conflict would look as follows:

```
<<<<<<<
1
=======
<<<<<<<
2
=======
3
>>>>>>>
>>>>>>>
```

and finally the conflict ID would be calculated as: `sha1('1<NUL><<<<<<\n3\n=======\n2\n>>>>>><NUL>')`

□□□ #2
□ □□□ □□□ 2026-10-31 20:17:11 UTC
□ □□□ □□□ 2026-10-31 20:17:12 UTC