3.2. 冲突解决

上一章介绍了Git协议，并且使用本地协议来模拟一个远程的版本库，以两个不同用户的身份检出该版本库，和该远程版本库进行交互——交换数据、协同工作。在上一章的协同中只遇到了一个小小的麻烦——非快进式推送，可以通过执行PULL（拉回）操作，成功完成合并后再推送。

但是在真实的运行环境中，用户间协同并不总是会一帆风顺，只要有合并就可能会有冲突。本章就重点介绍冲突解决机制。

3.2.1. 拉回操作中的合并

为了降低难度，上一章的实践中用户user1执行**git pull**操作解决非快进式推送问题似乎非常的简单，就好像直接把共享版本库中最新提交直接拉回到本地，然后就可以推送了，其他好像什么都没有发生一样。真的是这样么？

- 用户user1向共享版本库推送时，因为user2强制推送已经改变了共享版本库中的提交状态，导致user1推送失败，如图16-1所示。

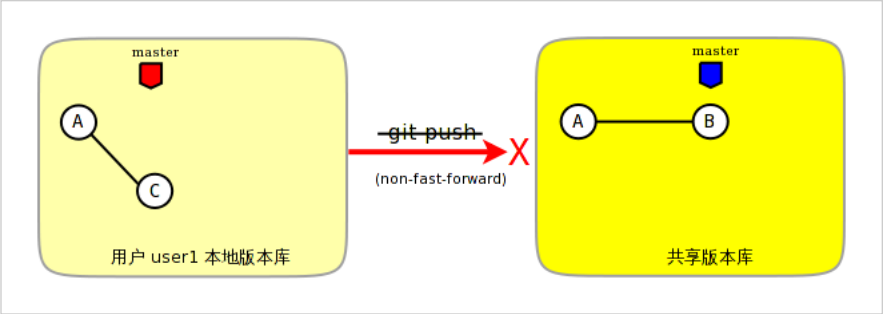

图 16-1：非快进式推送被禁止

- 用户user1执行PULL操作的第一阶段，将共享版本库master分支的最新提交拉回到本地，并更新到本地版本库特定的引用`refs/remotes/origin/master`（简称为`origin/master`），如图16-2所示。

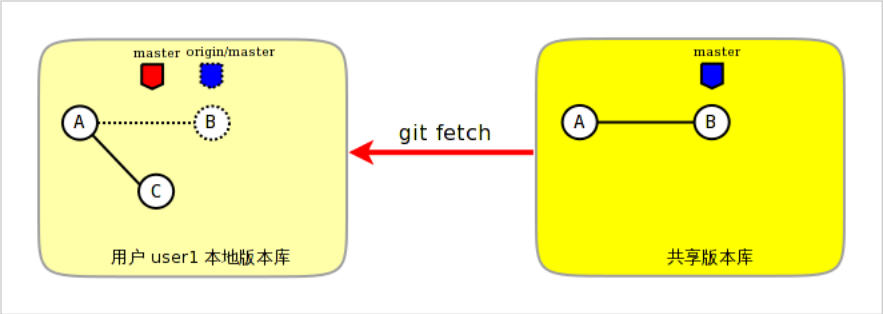

图 16-2：执行获取操作

- 用户user1执行PULL操作的第二阶段，将本地分支master和共享版本库本地跟踪分支`origin/master`进行合并操作，如图16-3所示。

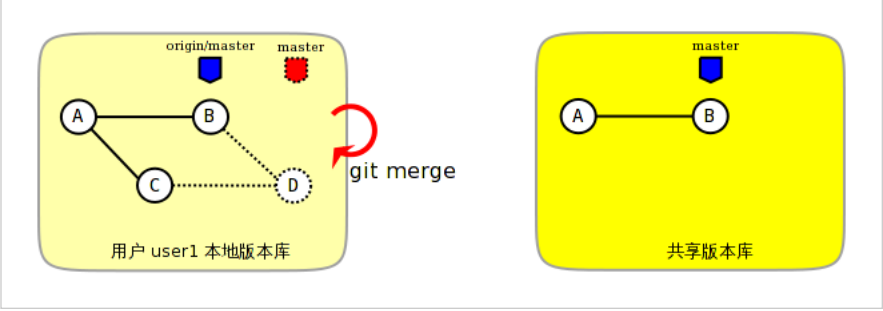

图 16-3：执行合并操作

- 用户user1执行PUSH操作，将本地提交推送到共享版本库中，如图16-4所示。

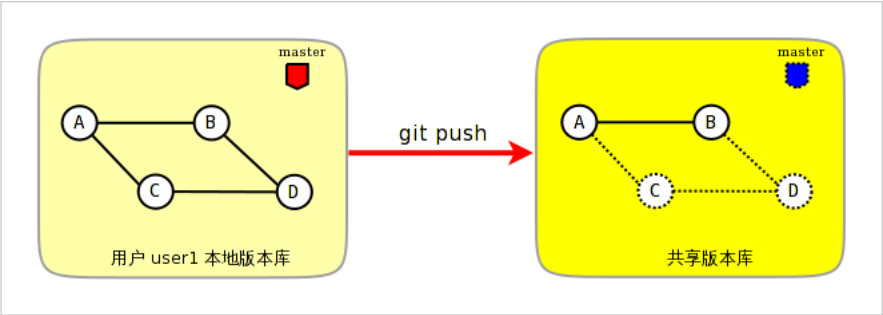

图 16-4：执行推送操作

实际上拉回（PULL）操作是由两个步骤组成的，一个是获取（FETCH）操作，一个是合并（MERGE）操作，即：

```
git pull = git fetch + git merge
```

图16-2示意的获取（FETCH）操作看似很简单，实际上要到第19章介绍远程版本库的章节才能够讲明白，现在只需要根据图示将获取操作理解为将远程的共享版本库的对象（提交、里程碑、分支等）复制到本地即可。

合并（MERGE）操作是本章要介绍的重点。合并操作可以由拉回操作（**git pull**）隐式的执行，将其他版本库的提交和本地版本库的提交进行合并。还可以针对本版本库中的其他分支（将在第18章中介绍）进行显示的合并操作，将其他分支的提交和当前分支的提交进行合并。

合并操作的命令行格式如下：

```
git merge [选项...] <commit>...
```

合并操作的大多数情况，只须提供一个`<commit>`（提交ID或对应的引用：分支、里程碑等）作为参数。合并操作将`<commit>`对应的目录树和当前工作分支的目录树的内容进行合并，合并后的提交以当前分支的提交作为第一个父提交，以`<commit>`为第二个父提交。合并操作还支持将多个`<commit>`代表的分支和当前分支进行合并，过程类似。合并操作的选项很多，这会在本章及第24章"子树合并"中予以介绍。

默认情况下，合并后的结果会自动提交，但是如果提供`--no-commit`选项，则合并后的结果会放入暂存区，用户可以对合并结果进行检查、更改，然后手动提交。

合并操作并非总会成功，因为合并的不同提交可能同时修改了同一文件相同区域的内容，导致冲突。冲突会造成合并操作的中断，冲突的文件被标识，用户可以对标识为冲突的文件进行冲突解决操作，然后更新暂存区，再提交，最终完成合并操作。

根据合并操作是否遇到冲突，以及不同的冲突类型，可以分为以下几种情况：成功的自动合并、逻辑冲突、真正的冲突和树冲突。下面分别予以介绍。

3.2.2. 合并一：自动合并

Git的合并操作非常智能，大多数情况下会自动完成合并。不管是修改不同的文件，还是修改相同的文件（文件的不同位置），或者文件名变更。

3.2.2.1. 修改不同的文件

如果用户user1和user2各自的本地提交中修改了不同的文件，当一个用户将改动推送到服务器后，另外一个用户推送就遇到非快进式推送错误，需要先合并再推送。因两个用户修改了不同的文件，合并不会遇到麻烦。

在上一章的操作过程中，两个用户的本地版本库和共享版本库可能不一致，为确保版本库状态的一致性以便下面的实践能够正常执行，分别在两个用户的本地版本库中执行下面的操作。

```
$ git pull
$ git reset --hard origin/master
```

下面的实践中，两个用户分别修改不同的文件，其中一个用户要尝试合并操作将本地提交和另外一个用户的提交合并。

- 用户user1修改`team/user1.txt`文件，提交并推送到共享服务器。

```
$ cd /path/to/user1/workspace/project/
$ echo "hack by user1 at `date -R`" >> team/user1.txt
$ git add -u
$ git commit -m "update team/user1.txt"
$ git push
```

- 用户user2修改`team/user2.txt`文件，提交。

```
$ cd /path/to/user2/workspace/project/
$ echo "hack by user2 at `date -R`" >> team/user2.txt
$ git add -u
$ git commit -m "update team/user2.txt"
```

- 用户user2在推送的时候，会遇到非快进式推进的错误而被终止。

```
$ git push
To file:///path/to/repos/shared.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/shared.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

- 用户user2执行获取（**git fetch**）操作。获取到的提交更新到本地跟踪共享版本库master分支的本地引用`origin/master`中。

```
$ git fetch
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/shared
   bccc620..25fce74  master     -> origin/master
```

- 用户user2执行合并操作，完成自动合并。

```
$ git merge origin/master
Merge made by recursive.
 team/user1.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

- 用户user2推送合并后的本地版本库到共享版本库。

```
$ git push
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 747 bytes, done.
Total 7 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
To file:///path/to/repos/shared.git
   25fce74..0855b86  master -> master
```

- 通过提交日志，可以看到成功合并的提交和其两个父提交的关系图。

```
$ git log -3 --graph --stat
*   commit 0855b86678d1cf86ccdd13adaaa6e735715d6a7e
|\  Merge: f53acdf 25fce74
| | Author: user2 <user2@moon.ossxp.com>
| | Date:   Sat Dec 25 23:00:55 2010 +0800
| |
| |     Merge remote branch 'origin/master'
| |
| * commit 25fce74b5e73b960c42e4a463d03d462919b674d
| | Author: user1 <user1@sun.ossxp.com>
| | Date:   Sat Dec 25 22:54:53 2010 +0800
| |
| |     update team/user1.txt
| |
| |  team/user1.txt |    1 +
| |  1 files changed, 1 insertions(+), 0 deletions(-)
| |
* | commit f53acdf6a76e0552b562f5aaa4d40ff19e8e2f77
|/  Author: user2 <user2@moon.ossxp.com>
|   Date:   Sat Dec 25 22:56:49 2010 +0800
|
|       update team/user2.txt
|
|    team/user2.txt |    1 +
|    1 files changed, 1 insertions(+), 0 deletions(-)
```

3.2.2.2. 修改相同文件的不同区域

当用户user1和user2在本地提交中修改相同的文件，但是修改的是文件的不同位置时，则两个用户的提交仍可成功合并。

- 为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

- 用户user1在自己的工作区中修改README文件，在文件的第一行插入内容，更改后的文件内容如下。

```
User1 hacked.
Hello.
```

- 用户user1对修改进行本地提交并推送到共享版本库。

```
$ git add -u
$ git commit -m "User1 hack at the beginning."
$ git push
```

- 用户user2在自己的工作区中修改README文件，在文件的最后插入内容，更改后的文件内容如下。

```
Hello.
User2 hacked.
```

- 用户user2对修改进行本地提交。

```
$ git add -u
$ git commit -m "User2 hack at the end."
```

- 用户user2执行获取（**git fetch**）操作。获取到的提交更新到本地跟踪共享版本库master分支的本地引用origin/master中。

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From file:///path/to/repos/shared
   0855b86..07e9d08  master     -> origin/master
```

- 用户user2执行合并操作，完成自动合并。

```
$ git merge refs/remotes/origin/master
Auto-merging README
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

- 用户user2推送合并后的本地版本库到共享版本库。

```
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 607 bytes, done.
Total 6 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (6/6), done.
To file:///path/to/repos/shared.git
   07e9d08..2a67e6f  master -> master
```

- 如果追溯一下README文件每一行的来源，可以看到分别是user1和user2更改的最前和最后的一行。

```
$ git blame README
07e9d082 (user1 2010-12-25 23:12:17 +0800 1) User1 hacked.
^5174bf3 (user1 2010-12-19 15:52:29 +0800 2) Hello.
bb0c74fa (user2 2010-12-25 23:14:27 +0800 3) User2 hacked.
```

3.2.2.3. 同时更改文件名和文件内容

如果一个用户将文件移动到其他目录（或修改文件名），另外一个用户针对重命名前的文件进行了修改，还能够实现自动合并么？这对于其他版本控制系统可能是一个难题，例如Subversion就不能很好地处理，还为此引入了一个“树冲

突”的新名词。Git对于此类冲突能够很好地处理，可以自动解决冲突实现自动合并。

- 为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

- 用户user1在自己的工作区中将文件README进行重命名，本地提交并推送到共享版本库。

```
$ cd /path/to/user1/workspace/project/
$ mkdir doc
$ git mv README doc/README.txt
$ git commit -m "move document to doc/."
$ git push
```

- 用户user2在自己的工作区中修改README文件，在文件的最后插入内容，并本地提交。

```
$ cd /path/to/user2/workspace/project/
$ echo "User2 hacked again." >> README
$ git add -u
$ git commit -m "User2 hack README again."
```

- 用户user2执行获取（**git fetch**）操作。获取到的提交更新到本地跟踪共享版本库master分支的本地引用origin/master中。

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From file:///path/to/repos/shared
   0855b86..07e9d08  master     -> origin/master
```

- 用户user2执行合并操作，完成自动合并。

```
$ git merge refs/remotes/origin/master
Merge made by recursive.
 README => doc/README.txt |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename README => doc/README.txt (100%)
```

- 用户user2推送合并后的本地版本库到共享版本库。

```
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 636 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To file:///path/to/repos/shared.git
   9c51cb9..f73db10  master -> master
```

- 使用-m参数可以查看合并操作所做出的修改。

```
$ git log -1 -m --stat
commit f73db106c820f0c6d510f18ae8c67629af9c13b7 (from 887488eee19300c566c272ec84b236026b0303c6)
Merge: 887488e 9c51cb9
Author: user2 <user2@moon.ossxp.com>
Date:   Sat Dec 25 23:36:57 2010 +0800

    Merge remote branch 'refs/remotes/origin/master'

 README         |    4 ----
 doc/README.txt |    4 ++++
 2 files changed, 4 insertions(+), 4 deletions(-)

commit f73db106c820f0c6d510f18ae8c67629af9c13b7 (from 9c51cb91bfe12654e2de1d61d722161db0539644)
Merge: 887488e 9c51cb9
Author: user2 <user2@moon.ossxp.com>
Date:   Sat Dec 25 23:36:57 2010 +0800

    Merge remote branch 'refs/remotes/origin/master'

 doc/README.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

3.2.3. 合并二：逻辑冲突

自动合并如果成功地执行，则大多数情况下就意味着完事大吉，但是在某些特殊情况下，合并后的结果虽然在Git看来是完美的合并，实际上却存在着逻辑冲突。

一个典型的逻辑冲突是一个用户修改了一个文件的文件名，而另外的用户在其他文件中引用旧的文件名，这样的合并虽然能够成功但是包含着逻辑冲突。例如：

- 一个C语言的项目中存在头文件`hello.h`，该头文件定义了一些函数声明。
- 用户user1将`hello.h`文件改名为`api.h`。
- 用户user2写了一个新的源码文件`foo.c`并在该文件中包含了`hello.h`文件。
- 两个用户的提交合并后，会因为源码文件`foo.c`找不到包含的`hello.h`文件而导致项目编译失败。

再举一个逻辑冲突的示例。假如一个用户修改了函数返回值而另外的用户使用旧的函数返回值，虽然成功合并但是存在逻辑冲突：

- 函数`compare(obj1, obj2)`用于比较两个对象`obj1`和`obj2`。返回`1`代表比较的两个对象相同，返回`0`代表比较的两个对象不同。
- 用户user1修改了该函数的返回值，返回`0`代表两个对象相同，返回`1`代表`obj1`大于`obj2`，返回`-1`则代表`obj1`小于`obj2`。
- 用户user2不知道user1对该函数的改动，仍以该函数原返回值判断两个对象的异同。
- 两个用户的提交合并后，不会出现编译错误，但是软件中会潜藏着重大的Bug。

上面的两个逻辑冲突的示例，尤其是最后一个非常难以捕捉。如果因此而贬低Git的自动合并，或者对每次自动合并的结果疑神疑鬼，进而花费大量精力去分析合并的结果，则是因噎废食、得不偿失。一个好的项目实践是每个开发人员都为自己的代码编写可运行的单元测试，项目每次编译时都要执行自动化测试，捕捉潜藏的Bug。在2010年OpenParty上的一个报告中，我介绍了如何在项目中引入单元测试及自动化集成，可以参考下面的链接：

- http://www.beijing-open-party.org/topic/9
- http://wenku.baidu.com/view/63bf7d160b4e767f5acfcef6.html

3.2.4. 合并三：冲突解决

如果两个用户修改了同一文件的同一区域，则在合并的时候会遇到冲突导致合并过程中断。这是因为Git并不能越俎代庖的替用户做出决定，而是把决定权交给用户。在这种情况下，Git显示为合并冲突，等待用户对冲突做出抉择。

下面的实践非常简单，两个用户都修改`doc/README.txt`文件，在第二行"Hello."的后面加上自己的名字。

- 为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

- 用户user1在自己的工作区修改`doc/README.txt`文件（仅改动了第二行）。修改后内容如下：

```
User1 hacked.
Hello, user1.
User2 hacked.
User2 hacked again.
```

- 用户user1对修改进行本地提交并推送到共享版本库。

```
$ git add -u
$ git commit -m "Say hello to user1."
$ git push
```

- 用户user2在自己的工作区修改`doc/README.txt`文件（仅改动了第二行）。修改后内容如下：

```
User1 hacked.
Hello, user2.
User2 hacked.
User2 hacked again.
```

- 用户user2对修改进行本地提交。

```
$ git add -u
$ git commit -m "Say hello to user2."
```

- 用户user2执行拉回操作，遇到冲突。

`git pull`操作相当于`git fetch`和`git merge`两个操作。

```
$ git pull
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/shared
   f73db10..a123390  master     -> origin/master
Auto-merging doc/README.txt
CONFLICT (content): Merge conflict in doc/README.txt
Automatic merge failed; fix conflicts and then commit the result.
```

执行**git pull**时所做的合并操作由于遇到冲突导致中断。来看看处于合并冲突状态时工作区和暂存区的状态。

执行**git status**命令，可以从状态输出中看到文件`doc/README.txt`处于未合并的状态，这个文件在两个不同的提交中都做了修改。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      doc/README.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

那么Git是如何记录合并过程及冲突的呢？实际上合并过程是通过`.git`目录下的几个文件进行记录的：

- 文件`.git/MERGE_HEAD`记录所合并的提交ID。
- 文件`.git/MERGE_MSG`记录合并失败的信息。
- 文件`.git/MERGE_MODE`标识合并状态。

版本库暂存区中则会记录冲突文件的多个不同版本。可以使用**git ls-files**命令查看。

```
$ git ls-files -s
100644 ea501534d70a13b47b3b4b85c39ab487fa6471c2 1       doc/README.txt
100644 5611db505157d312e4f6fb1db2e2c5bac2a55432 2       doc/README.txt
100644 036dbc5c11b0a0cefc8247cf0e9a3e678f8de060 3       doc/README.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

在上面的输出中，每一行分为四个字段，前两个分别是文件的属性和SHA1哈希值。第三个字段是暂存区编号。当合并冲突发生后，会用到0以上的暂存区编号。

- 编号为1的暂存区用于保存冲突文件修改之前的副本，即冲突双方共同的祖先版本。可以用`:1:<filename>`访问。

  ```
  $ git show :1:doc/README.txt
  User1 hacked.
  Hello.
  User2 hacked.
  User2 hacked again.
  ```

- 编号为2的暂存区用于保存当前冲突文件在当前分支中修改的副本。可以用`:2:<filename>`访问。

  ```
  $ git show :2:doc/README.txt
  User1 hacked.
  Hello, user2.
  User2 hacked.
  User2 hacked again.
  ```

- 编号为3的暂存区用于保存当前冲突文件在合并版本（分支）中修改的副本。可以用`:3:<filename>`访问。

  ```
  $ git show :3:doc/README.txt
  User1 hacked.
  Hello, user1.
  User2 hacked.
  User2 hacked again.
  ```

对暂存区中冲突文件的上述三个副本无须了解太多，这三个副本实际上是提供冲突解决工具，用于实现三向文件合并的。

工作区的版本则可能同时包含了成功的合并及冲突的合并，其中冲突的合并会用特殊的标记（<<<<<<< ======= >>>>>>>）进行标识。查看当前工作区中冲突的文件：

```
$ cat doc/README.txt
User1 hacked.
<<<<<<< HEAD
Hello, user2.
=======
Hello, user1.
>>>>>>> a123390b8936882bd53033a582ab540850b6b5fb
User2 hacked.
User2 hacked again.
```

特殊标识<<<<<<<（七个小于号）和=======（七个等号）之间的内容是当前分支所更改的内容。在特殊标识=======（七个等号）和>>>>>>>（七个大于号）之间的内容是所合并的版本更改的内容。

冲突解决的实质就是通过编辑操作，将冲突标识符所标识的冲突内容替换为合适的内容，并去掉冲突标识符。编辑完毕后执行**git add**命令将文件添加到暂存区（标号0），然后再提交就完成了冲突解决。

当工作区处于合并冲突状态时，无法再执行提交操作。此时有两个选择：放弃合并操作，或者对合并冲突进行冲突解决操作。放弃合并操作非常简单，只须执行**git reset**将暂存区重置即可。下面重点介绍如何进行冲突解决的操作。有两个方法进行冲突解决，一个是对少量冲突非常适合的手工编辑操作，另外一个是使用图形化冲突解决工具。

3.2.4.1. 手工编辑完成冲突解决

先来看看不使用工具，直接手动编辑完成冲突解决。打开文件doc/README.txt，将冲突标识符所标识的文字替换为Hello, user1 and user2.。修改后的文件内容如下：

```
User1 hacked.
Hello, user1 and user2.
User2 hacked.
User2 hacked again.
```

然后添加到暂存区，并提交：

```
$ git add -u
$ git commit -m "Merge completed: say hello to all users."
```

查看最近三次提交的日志，会看到最新的提交就是一个合并提交：

```
$ git log --oneline --graph -3
*   bd3ad1a Merge completed: say hello to all users.
|\
| * a123390 Say hello to user1.
* | 60b10f3 Say hello to user2.
|/
```

提交完成后，会看到.git目录下与合并相关的文件.git/MERGE_HEAD、.git/MERGE_MSG、.git/MERGE_MODE文件都自动删除了。

如果查看暂存区，会发现冲突文件在暂存区中的三个副本也都清除了（实际在对编辑完成的冲突文件执行**git add**后就已经清除了）。

```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0       doc/README.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

3.2.4.2. 图形工具完成冲突解决

上面介绍的通过手工编辑完成冲突解决并不复杂，对于简单的冲突是最快捷的解决方法。但是如果冲突的区域过多、过大，并且缺乏原始版本作为参照，冲突解决过程就会显得非常的不便，这种情况下使用图形工具就显得非常有优势。

还以上面的冲突解决为例介绍使用图形工具进行冲突解决的方法。为了制造一个冲突，首先把user2辛辛苦苦完成的冲突解决提交回滚，再执行合并进入冲突状态。

- 将冲突解决的提交回滚，强制重置到前一个版本。

  ```
  $ git reset --hard HEAD^
  ```

- 这时查看状态，会显示当前工作分支的最新提交和共享版本库的master分支的最新提交出现了偏离。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
nothing to commit (working directory clean)
```

- 那么执行合并操作吧。冲突发生了。

```
$ git merge refs/remotes/origin/master
Auto-merging doc/README.txt
CONFLICT (content): Merge conflict in doc/README.txt
Automatic merge failed; fix conflicts and then commit the result.
```

下面就演示使用图形工具如何解决冲突。使用图形工具进行冲突解决需要事先在操作系统中安装相关的工具软件，如：kdiff3、meld、tortoisemerge、araxis等。而启动图形工具进行冲突解决也非常简单，只须执行命令**git mergetool**即可。

```
$ git mergetool
merge tool candidates: opendiff kdiff3 tkdiff xxdiff meld tortoisemerge
gvimdiff diffuse ecmerge p4merge araxis emerge vimdiff
Merging:
doc/README.txt

Normal merge conflict for 'doc/README.txt':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (kdiff3):
```

运行**git mergetool**命令后，会显示支持的图形工具列表，并提示用户选择可用的冲突解决工具。默认会选择系统中已经安装的工具软件，如**kdiff3**。直接按下回车键，自动打开**kdiff3**进入冲突解决界面：

启动**kdiff3**后，如图16-5，上方三个窗口由左至右显示冲突文件的三个版本，分别是：

A. 暂存区1中的版本（共同祖先版本）。
B. 暂存区2中的版本（当前分支更改的版本）。
C. 暂存区3中的版本（他人更改的版本）。

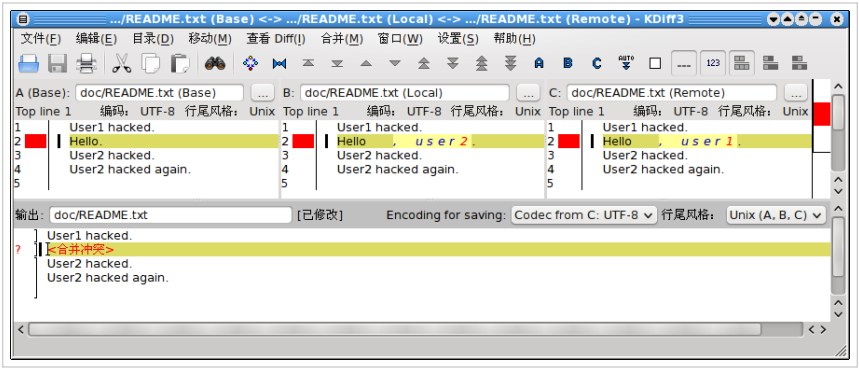

图 16-5：kdiff3 冲突解决界面

kdiff3下方的窗口是合并后文件的编辑窗口。如图16-6所示，点击标记为"合并冲突"的一行，在弹出菜单中出现A、B、C三个选项，分别代表从A、B、C三个窗口拷贝相关内容到当前位置。

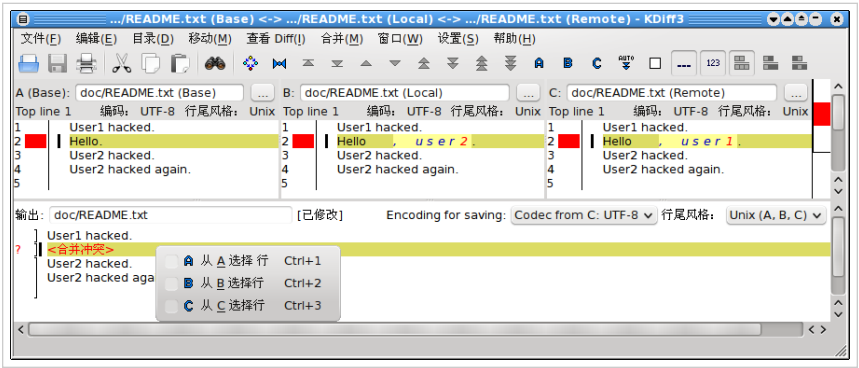

图 16-6：kdiff3 合并冲突行的弹出菜单

当通过图16-6显示的弹出菜单选择了B和C后，可以在图16-7中看到在合并窗口出现了标识B和C的行，分别代表user2和user1对该行的修改。

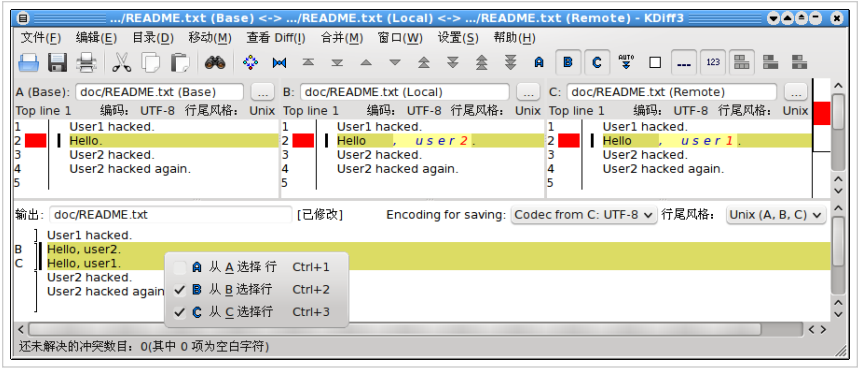

图 16-7：在 kdiff3 冲突区域同时选取B和C的修改

在合并窗口进行编辑，将“Hello, user1.”修改为“Hello, user1 and user2.”，如图16-8。修改后，可以看到该行的标识由C改变为m，含义是该行是经过手工修改的行。

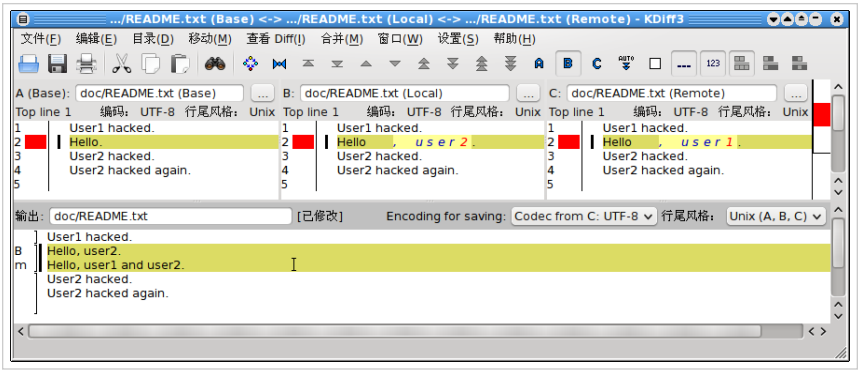

图 16-8：在 kdiff3 的冲突区域编辑内容

在合并窗口删除标识为从B窗口引入的行“Hello, user2.”，如图16-9。保存退出即完成图形化冲突解决。

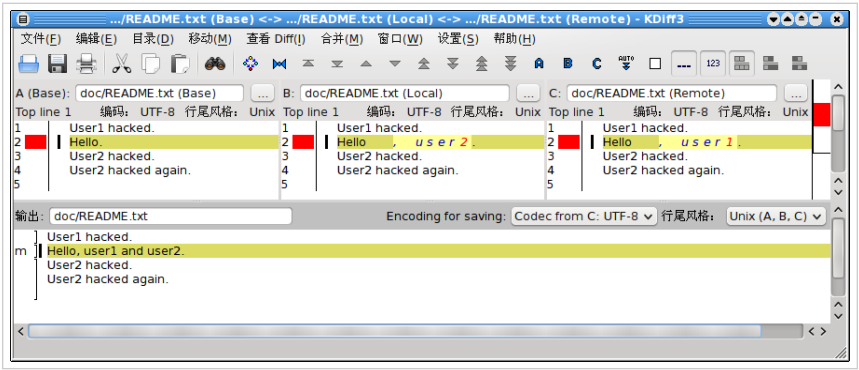

图 16-9：完成 kdiff3 冲突区域的编辑

图形工具保存退出后，显示工作区状态，会看到冲突已经解决。在工作区还会遗留一个以.orig结尾的合并前文件副本。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Changes to be committed:
#
#       modified:   doc/README.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       doc/README.txt.orig
```

查看暂存区会发现暂存区中的冲突文件的三个副本都已经清除。

```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0       doc/README.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

执行提交和推送。

```
$ git commit -m "Say hello to all users."
[master 7f7bb5e] Say hello to all users.
$ git push
Counting objects: 14, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 712 bytes, done.
Total 8 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To file:///path/to/repos/shared.git
   a123390..7f7bb5e  master -> master
```

查看最近三次的提交日志，会看到最新的提交是一个合并提交。

```
$ git log --oneline --graph -3
*   7f7bb5e Say hello to all users.
|\
| * a123390 Say hello to user1.
* | 60b10f3 Say hello to user2.
|/
```

3.2.5. 合并四：树冲突

如果一个用户将某个文件改名，另外一个用户将同样的文件改为另外的名字，当这两个用户的提交进行合并操作时，Git显然无法替用户做出裁决，于是就产生了冲突。这种因为文件名修改造成的冲突，称为树冲突。这种树冲突的解决方式比较特别，因此专题介绍。

仍旧使用前面的版本库进行此次实践。为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

下面就分别以两个用户的身份执行提交，将同样的一个文件改为不同的文件名，制造一个树冲突。

- 用户user1将文件`doc/README.txt`改名为`readme.txt`，提交并推送到共享版本库。

  ```
  $ cd /path/to/user1/workspace/project
  $ git mv doc/README.txt readme.txt
  $ git commit -m "rename doc/README.txt to readme.txt"
  [master 615c1ff] rename doc/README.txt to readme.txt
   1 files changed, 0 insertions(+), 0 deletions(-)
   rename doc/README.txt => readme.txt (100%)
  $ git push
  Counting objects: 3, done.
  Delta compression using up to 2 threads.
  Compressing objects: 100% (2/2), done.
  Writing objects: 100% (2/2), 282 bytes, done.
  Total 2 (delta 0), reused 0 (delta 0)
  Unpacking objects: 100% (2/2), done.
  To file:///path/to/repos/shared.git
     7f7bb5e..615c1ff  master -> master
  ```

- 用户user2将文件`doc/README.txt`改名为`README`，并做本地提交。

  ```
  $ cd /path/to/user2/workspace/project
  $ git mv doc/README.txt README
  $ git commit -m "rename doc/README.txt to README"
  [master 20180eb] rename doc/README.txt to README
   1 files changed, 0 insertions(+), 0 deletions(-)
   rename doc/README.txt => README (100%)
  ```

- 用户user2执行**git pull**操作，遇到合并冲突。

  ```
  $ git pull
  remote: Counting objects: 3, done.
  remote: Compressing objects: 100% (2/2), done.
  remote: Total 2 (delta 0), reused 0 (delta 0)
  Unpacking objects: 100% (2/2), done.
  From file:///path/to/repos/shared
     7f7bb5e..615c1ff  master     -> origin/master
  ```

```
CONFLICT (rename/rename): Rename "doc/README.txt"->"README" in branch "HEAD" rename "doc/README.txt"->"readme.tx
Automatic merge failed; fix conflicts and then commit the result.
```

因为两个用户同时更改了同一文件的文件名并且改成了不同的名字，于是引发冲突。此时查看状态会看到：

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       added by us:        README
#       both deleted:       doc/README.txt
#       added by them:      readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

此时查看一下用户user2本地版本库的暂存区，可以看到因为冲突在编号为1、2、3的暂存区出现了相同SHA1哈希值的对象，但是文件名各不相同。

```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 2       README
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 1       doc/README.txt
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 3       readme.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

其中在暂存区1中是改名之前的`doc/README.txt`，在暂存区2中是用户user2改名后的文件名`README`，而暂存区3是其他用户（user1）改名后的文件`readme.txt`。

此时的工作区中存在两个相同的文件`README`和`readme.txt`分别是用户user2和user1对`doc/README.txt`重命名之后的文件。

```
$ ls -l readme.txt README
-rw-r--r-- 1 jiangxin jiangxin 72 12月 27 12:25 README
-rw-r--r-- 1 jiangxin jiangxin 72 12月 27 16:53 readme.txt
```

3.2.5.1. 手工操作解决树冲突

这时user2应该和user1商量一下到底应该将该文件改成什么名字。如果双方最终确认应该采用user2重命名的名称，则user2应该进行下面的操作完成冲突解决。

- 删除文件`readme.txt`。

  在执行**git rm**操作过程会弹出三条警告，说共有三个文件待合并。

  ```
  $ git rm readme.txt
  README: needs merge
  doc/README.txt: needs merge
  readme.txt: needs merge
  rm 'readme.txt'
  ```

- 删除文件`doc/README.txt`。

  执行删除过程，弹出的警告少了一条，因为前面的删除操作已经将一个冲突文件撤出暂存区了。

  ```
  $ git rm doc/README.txt
  README: needs merge
  doc/README.txt: needs merge
  rm 'doc/README.txt'
  ```

- 添加文件`README`。

  ```
  $ git add README
  ```

- 这时查看一下暂存区，会发现所有文件都在暂存区0中。

  ```
  $ git ls-files -s
  100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0       README
  100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
  100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
  ```

- 提交完成冲突解决。

```
$ git commit -m "fixed tree conflict."
[master e82187e] fixed tree conflict.
```

- 查看一下最近三次提交日志，看到最新的提交是一个合并提交。

```
$ git log --oneline --graph -3 -m --stat
*   e82187e (from 615c1ff) fixed tree conflict.
|\
| |  README     |    4 ++++
| |  readme.txt |    4 ----
| |  2 files changed, 4 insertions(+), 4 deletions(-)
| * 615c1ff rename doc/README.txt to readme.txt
| |  doc/README.txt |    4 ----
| |  readme.txt     |    4 ++++
| |  2 files changed, 4 insertions(+), 4 deletions(-)
* | 20180eb rename doc/README.txt to README
|/
|    README         |    4 ++++
|    doc/README.txt |    4 ----
|    2 files changed, 4 insertions(+), 4 deletions(-)
```

3.2.5.2. 交互式解决树冲突

树冲突虽然不能像文件冲突那样使用图形工具进行冲突解决，但还是可以使用**git mergetool**命令，通过交互式问答快速解决此类冲突。

首先将user2的工作区重置到前一次提交，再执行**git merge**引发树冲突。

- 重置到前一次提交。

```
$ cd /path/to/user2/workspace/project
$ git reset --hard HEAD^
HEAD is now at 20180eb rename doc/README.txt to README
$ git clean -fd
```

- 执行**git merge**引发树冲突。

```
$ git merge refs/remotes/origin/master
CONFLICT (rename/rename): Rename "doc/README.txt"->"README" in branch "HEAD" rename "doc/README.txt"->"readme.tx
Automatic merge failed; fix conflicts and then commit the result.
$ git status -s
AU README
DD doc/README.txt
UA readme.txt
```

上面操作所引发的树冲突，可以执行**git mergetool**命令进行交互式冲突解决，会如下逐一提示用户进行选择。

- 执行**git mergetool**命令。忽略其中的提示和警告。

```
$ git mergetool
merge tool candidates: opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse ecmerge p4merge araxis
Merging:
doc/README.txt
README
readme.txt

mv: 无法获取"doc/README.txt" 的文件状态(stat): 没有那个文件或目录
cp: 无法获取"./doc/README.txt.BACKUP.13869.txt" 的文件状态(stat): 没有那个文件或目录
mv: 无法将".merge_file_I3gfzy" 移动至"./doc/README.txt.BASE.13869.txt": 没有那个文件或目录
```

- 询问对文件doc/README.txt的处理方式。输入d选择将该文件删除。

```
Deleted merge conflict for 'doc/README.txt':
  {local}: deleted
  {remote}: deleted
Use (m)odified or (d)eleted file, or (a)bort? d
```

- 询问对文件README的处理方式。输入c选择将该文件保留（创建）。

```
Deleted merge conflict for 'README':
  {local}: created
  {remote}: deleted
Use (c)reated or (d)eleted file, or (a)bort? c
```

- 询问对文件readme.txt的处理方式。输入d选择将该文件删除。

```
Deleted merge conflict for 'readme.txt':
  {local}: deleted
  {remote}: created
Use (c)reated or (d)eleted file, or (a)bort? d
```

- 查看当前状态，只有一些尚未清理的临时文件，而冲突已经解决。

```
$ git status -s
?? .merge_file_I3gfzy
?? README.orig
```

- 提交完成冲突解决。

```
$ git commit -m "fixed tree conflict."
[master e070bc9] fixed tree conflict.
```

- 向共享服务器推送。

```
$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 457 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To file:///path/to/repos/shared.git
   615c1ff..e070bc9  master -> master
```

3.2.6. 合并策略

Git合并操作支持很多合并策略，默认会选择最适合的合并策略。例如，和一个分支进行合并时会选择recursive合并策略，当和两个或两个以上的其他分支进行合并时采用octopus合并策略。可以通过传递参数使用指定的合并策略，命令行如下：

```
git merge [-s <strategy>] [-X <strategy-option>] <commit>...
```

其中参数-s用于设定合并策略，参数-X用于为所选的合并策略提供附加的参数。

下面分别介绍不同的合并策略：

- resolve

  该合并策略只能用于合并两个头（即当前分支和另外的一个分支），使用三向合并策略。这个合并策略被认为是最安全、最快的合并策略。

- recursive

  该合并策略只能用于合并两个头（即当前分支和另外的一个分支），使用三向合并策略。这个合并策略是合并两个头指针时的默认合并策略。

  当合并的头指针拥有一个以上的祖先的时候，会针对多个公共祖先创建一个合并的树，并以此作为三向合并的参照。这个合并策略被认为可以实现冲突的最小化，而且可以发现和处理由于重命名导致的合并冲突。

  这个合并策略可以使用下列选项。

  - ours

    在遇到冲突的时候，选择我们的版本（当前分支的版本），而忽略他人的版本。如果他人的改动和本地改动不冲突，会将他人改动合并进来。

    不要将此模式和后面介绍的单纯的ours合并策略相混淆。后面介绍的ours合并策略直接丢弃其他分支的变更，无论冲突与否。

  - theirs

    和ours选项相反，遇到冲突时选择他人的版本，丢弃我们的版本。

  - subtree[=path]

这个选项使用子树合并策略，比下面介绍的`subtree`（子树合并）策略的定制能力更强。下面的`subtree`合并策略要对两个树的目录移动进行猜测，而`recursive`合并策略可以通过此参数直接对子树目录进行设置。

- octopus

  可以合并两个以上的头指针，但是拒绝执行需要手动解决的复杂合并。主要的用途是将多个主题分支合并到一起。这个合并策略是对三个及三个以上头指针进行合并时的默认合并策略。

- ours

  可以合并任意数量的头指针，但是合并的结果总是使用当前分支的内容，丢弃其他分支的内容。

- subtree

  这是一个经过调整的recursive策略。当合并树A和B时，如果B和A的一个子树相同，B首先进行调整以匹配A的树的结构，以免两棵树在同一级别进行合并。同时也针对两棵树的共同祖先进行调整。

  关于子树合并会在第4篇的第24章"子树合并"中详细介绍。

3.2.7. 合并相关的设置

可以通过**git config**命令设置与合并相关的环境变量，对合并进行配置。下面是一些常用的设置。

- merge.conflictstyle

  该变量定义冲突文件的显示风格，有两个可用的风格，默认的"merge"或"diff3"。

  默认的"merge"风格使用标准的冲突分界符（`<<<<<<<`、`=======`、`>>>>>>>`）对冲突内容进行标识，其中的两个文字块分别是本地的修改和他人的修改。

  如果使用"diff3"风格，则会在冲突中出现三个文字块，分别是：`<<<<<<<`和`|||||||`之间的本地更改版本、`|||||||`和`=======`之间的原始（共同祖先）版本和`=======`和`>>>>>>>`之间的他人更改的版本。例如：

  ```
  User1 hacked.
  <<<<<<< HEAD
  Hello, user2.
  ||||||| merged common ancestors
  Hello.
  =======
  Hello, user1.
  >>>>>>> a123390b8936882bd53033a582ab540850b6b5fb
  User2 hacked.
  User2 hacked again.
  ```

- merge.tool

  执行**git mergetool**进行冲突解决时调用的图形化工具。变量`merge.tool`可以设置为如下内置支持的工具："kdiff3"、"tkdiff"、"meld"、"xxdiff"、"emerge"、"vimdiff"、"gvimdiff"、"diffuse"、"ecmerge"、"tortoisemerge"、"p

  ```
  $ git config --global merge.tool kdiff3
  ```

  如果将`merge.tool`设置为其他值，则使用自定义工具进行冲突解决。自定义工具需要通过`mergetool.<tool>.cmd`对自定义工具的命令行进行设置。

- mergetool.<tool>.path

  如果**git mergetool**支持的冲突解决工具安装在特殊位置，可以使用`mergetool.<tool>.path`对工具`<tool>`的安装位置进行设置。例如：

  ```
  $ git config --global mergetool.kdiff3.path /path/to/kdiff3
  ```

- mergetool.<tool>.cmd

  如果所用的冲突解决工具不在内置的工具列表中，还可以使用`mergetool.<tool>.cmd`对自定义工具的命令行进行设置，同时要将`merge.tool`设置为`<tool>`。

  自定义工具的命令行可以使用Shell变量。例如：

  ```
  $ git config --global merge.tool mykdiff3
  $ git config --global mergetool.mykdiff3.cmd '/usr/bin/kdiff3
              -L1 "$MERGED (Base)" -L2 "$MERGED (Local)" -L3 "$MERGED (Remote)"
              --auto -o "$MERGED" "$BASE" "$LOCAL" "$REMOTE"'
  ```

- merge.log

  是否在合并提交的提交说明中包含合并提交的概要信息。默认为`false`。