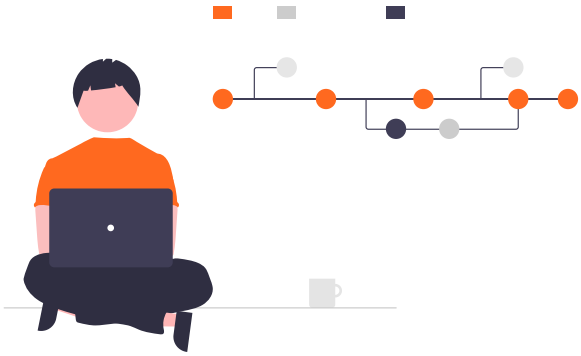


Git：合并、Cherry-Pick 和 Rebase

非常规指南

最后更新于2022 年 1 月 26 日 - 11 条评论

快速链接

↑ 顶部

您可以使用本指南深入了解 Git 的合并、变基和 cherry-picks 是如何在后台工作的，这样您就再也不会害怕它们了。

（**编者注**：大约 5500 字，您可能不想尝试在移动设备上阅读。将其加入书签，稍后再回来。即使在台式机上，也可以一次一口地读这头大象。）

介绍

当然，每个人和他们的祖母都使用 Git，而且似乎对它很满意。

但是你有没有搞砸合并然后你的解决方案是删除并重新克隆你的存储库？不太清楚哪里出了问题，为什么？

还是一个 rebase 突然让数十个合并冲突弹出，一个接一个，你不知道到底发生了什么？

简而言之，每当谈到*合并*、*变基*和*挑选*时，你是否有挥之不去的疑虑？

不要害怕，您来对地方了：本指南的其余部分将帮助您摆脱这些恐惧。

（预告片：到本文结束时，您将了解 a`git cherry-pick`本质上只是一个`git merge`。而 a`git rebase`本质上只是一个`git cherry-pick`？听起来很疯狂？请继续阅读！）

Git 存储内部结构

在您直接进入合并的具体细节之前，让我们看一下 Git 如何存储您的文件和提交。

从内部细节开始可能看起来有点奇怪，但请相信：这些内部结构是本指南中其他所有内容的基石，因此您需要先了解它们。

场景：提交两个文件

打开您的终端并执行以下命令。

```
# create a git repo in a directory of your liking

mkdir gitinternals
cd gitinternals
git init -b main

## add two .txt files and commit them

echo "-TODO-" > LICENSE.txt
echo "a marcobehler.com guide" > README.txt

git add LICENSE.txt
git add README.txt
```

```
git commit -m "Project Setup"

## update README.txt's contents

echo "a git guide" > README.txt

git add README.txt
git commit -m "Updated README"
```

您在第一次提交中创建了两个 .txt 文件，然后`README.txt`在第二次提交中更新了一个文件 ( ) 的内容。

这里有一个问题要问您：您认为 Git 将如何存储这两个提交，或者更确切地说是的两个版本`README.txt`？

- 它会在某处存储完整文件，即`a marcobehler.com guide`AND 吗？`a git guide`
- 它会存储增量，比如`a (-marcobehler.com)(+git) guide`（伪代码）吗？

奖励问题：这个问题的答案到底对合并或变基有何帮助？

让我们找出答案！

检查 Git 存储库：'git cat-file'

让我们在您的存储库中执行 a `git log`，您将获得类似于以下的输出：

```
# in your repository's directory
git log

# Project Setup

commit 142e5cf36d9f2047f24341883bd564b1d5170370 (HEAD -> main)
Author: Marco Behler <marco@marcobehler.com>
Date:   Tue Dec 28 09:54:44 2021 +0100

    Updated README

commit 715247c8426d3c16881539118e1eafeb38439b1c
Author: Marco Behler <marco@marcobehler.com>
Date:   Tue Dec 28 09:54:25 2021 +0100
```

```
    Project Setup
```

到目前为止，没有什么令人惊讶的——您将看到您的两个提交。您已经看到但可能多次忽略的东西是*commit ids*. 这是第二个提交的 ID。

```
commit 142e5cf36d9f2047f24341883bd564b1d5170370
```

更具体地说，*142e5cf36d9f2047f24341883bd564b1d5170370*不仅仅是一个随机 id，它是一个SHA-1 散列。

但是，这里*到底*散列了什么？

为了避免破坏答案，让我们使用另一个内置的 git 命令：*git cat-file*. *它基本上允许您*查看git 存储在存储库文件夹中*某处*.*git*的内容，前提是您碰巧知道它的 SHA1 哈希。听起来很有用，对吧？

执行以下命令（并确保使用您为提交获得的 SHA1 哈希值进行尝试）

```
# make sure to change the SHA1-hash!
git cat-file -p 142e5cf36d9f2047f24341883bd564b1d5170370
```

（注意：该*-p*选项确保漂亮地打印其输出。）

你会得到类似这样的输出：

```
# git cat-file's output
tree c4548e069652a6825894699ef7740a620ea0a6a8
parent 715247c8426d3c16881539118e1eafeb38439b1c
author Marco Behler <marco@marcobehler.com> 1641459065 +0100
committer Marco Behler <marco@marcobehler.com> 1641459065 +0100

Updated README
```

多田！这就是 Git 中的提交。这是一个包含……6 行的文本文件（5 行，还有一个空行将您的提交消息与其他行分隔开）。对真的。

如果你将这些行放入一个*sha1sum()*, 函数中，你将得到你的 SHA1 散列：*142e5cf36d9f2047f24341883bd564b1d5170370*！

> 对于高级读者，Git 并不完全执行 sha1sum(filecontent)，它实际上执行 sha1sum(header + filecontent) - 但我们稍后会对此进行介绍。

现在，您将熟悉您的提交（文件）中的一些行：

```
# who committed the file?
committer Marco Behler <marco@marcobehler.com> 1641459065 +0100

# what's the commit message?
Updated README
```

而提交的其他一些部分可能看起来不熟悉：

```
tree c4548e069652a6825894699ef7740a620ea0a6a8
parent 715247c8426d3c16881539118e1eafeb38439b1c
```

让我们（正确地）假设现在*parent(s)*只是引用当前提交之前的提交。那么，这*tree*条线代表什么？执行另一个*git cat-file*以找出答案！

```
# make sure to change the SHA1-hash to that of your tree!
git cat-file -p c4548e069652a6825894699ef7740a620ea0a6a8
```

看，这棵树似乎是另一个文本文件，引用了提交时存储库中所有文件的（快照）！

```
100644 blob ddd3b7b6335a636af9a9241096455e834f12f636    LICENSE.txt
100644 blob 773fc76fe191ceff24259d4e66efc90e86093b0c    README.txt
```

这是真的吗？好吧，你会通过做最后一次找到答案*git cat-file*，这次使用*README.txt's*散列。

```
git cat-file -p 773fc76fe191ceff24259d4e66efc90e86093b0c
```

这导致以下输出：

```
"a git guide"
```

这看起来很熟悉吗？是的，它是您*README.txt*文件的快照，在*第二次提*交时，即您更新自述文件时。这意味着 Git 确实为每次提交存储了完整的文件内容（假设内容已更改）？

好吧，可以肯定的是，让我们*git cat-file*为第一次提交重复这个游戏（这是一个很好的练习，所以请回头参考*git log*输出并重复这些步骤！）。你最终会得到这样的结果：

```
# cat'ing README.txt snapshotted during the first commit
git cat-file -p fe066d3f7568e13ef031b495e35c94be91b6366c

"a marcobehler.com guide"
```

要点：Git 不存储提交之间的增量，它总是为每次提交存储快照，即*完整文件*（只要文件发生更改并且其 SHA1 哈希值不在您的存储库中）。

这也是为什么 Git 对于具有（大部分）许多经常更改的二进制资产的项目来说不是一个很好的选择的原因。

.Git 文件夹

这是另一个练习。在您的项目文件夹中，转到您的*.git*子文件夹，更具体地说*.git/objects*。

```
# inside your project folder
cd .git/objects

# use 'dir' on Windows
ls -l
```

你会得到这样的输出：

```
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 11
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 14
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 71
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 77
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 c4
drwxrwxrwx 1 marco marco 512 Jan  6 09:50 dd
```

```
drwxrwxrwx 1 marco marco 512 Jan  6 09:51 fe
drwxrwxrwx 1 marco marco 512 Jan  6 09:50 info
drwxrwxrwx 1 marco marco 512 Jan  6 09:50 pack
```

现在，记下两次提交中任何一次提交的文件的 SHA1 哈希值*README.txt*，就像*142e5cf36d9f2047f24341883bd564b1d5170370*我的情况一样。更具体地说，看看前两个数字/字母，就*14*我而言。看，有一个*14*子目录！让我们进去吧。

```
cd 14
# use 'dir' on Windows
ls -l
```

输出为：

```
-r-xr-xr-x 1 marco marco 29 Jan  6 09:51 2e5cf36d9f2047f24341883bd564b1d5
```

文件夹内的一个文件。从*2e5...*. 稍等一下。如果你*14*从之前获取并将其与文件名（*2e5cf36d9f2047f24341883bd564b1d5170370*) 连接起来，你最终会得到...*正是*，你的 SHA1 哈希（*142e5cf36d9f2047f24341883bd564b1d5170370*)！

事实上，该文件*是**README.txt*您提交时文件的快照！

在过于兴奋并在您现有的遗留项目上尝试之前，请注意 Git 会压缩您的文件，因此您不能只打开该文件并查看它。但是，您可以查看此 Stackoverflow 答案以了解如何解压缩文件。

ew，那真是一段旅程！

Git SHA 总和

总结一下你刚刚学到的东西。当您提交新文件或更新文件时，即创建文件的新版本时，Git 将：

- ... 本质上是*sha1hash(header + filecontent)*对文件内容运行 a。阅读'How is git commit sha1 formed'以了解执行的功能。
- ... 将完整文件（不是增量文件）存储在 git 存储库的对象文件夹中，并将*散列*作为文件名。更具体地说：hash 的前两个字符作为子目录，hash.substring(2, length) 作为文件名。

- 请注意，Git 会压缩您的文件 (deflate/zlib)，而且，您的存储库越大，实际上会将文件合并在一起并更有效地存储它们，因此您的对象文件夹在更成熟的遗留项目中看起来会略有不同。尽管如此，相同的基础知识仍然适用。

哦，是的，提交和引用特定提交的所有文件的文件树也……只是文本文件。并以相同的方式存储和散列。

就这么简单。

Git 复习：提交、分支和头部

鉴于您每天都在处理这些概念，对提交和分支的良好理解似乎微不足道。

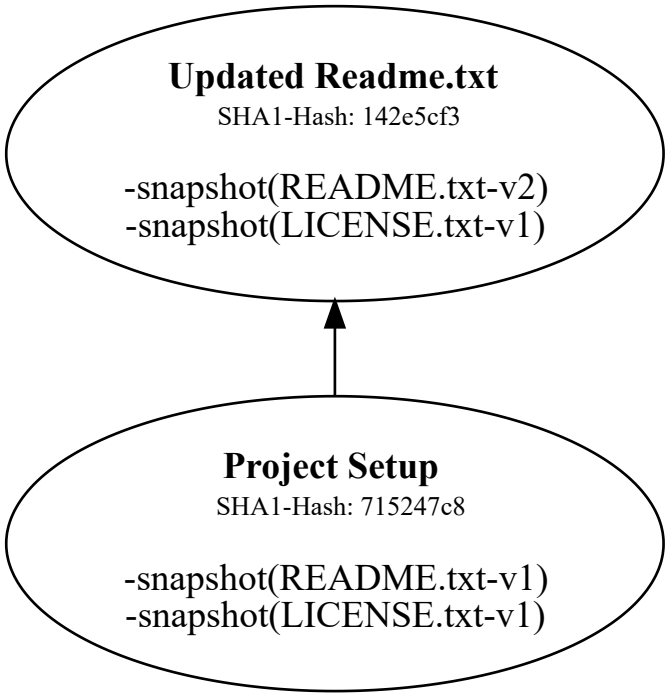
但是，你*真的*知道`HEAD`指的是什么吗？你舒服`detached HEADS`吗？

让我们快速复习一下。

您的存储库是什么样的

Git 存储库，或者更具体地说，您的提交历史被建模为Directed Acyclic Graph。

不要迷失在枯燥的计算机科学理论中，例如边、顶点和拓扑排序，让我们看一下当前提交历史的图表。



目前，您的存储库仅包含两个提交。正如您在上一节中了解到的，每个提交都引用了*所有*文件快照，这些快照在提交时有效。

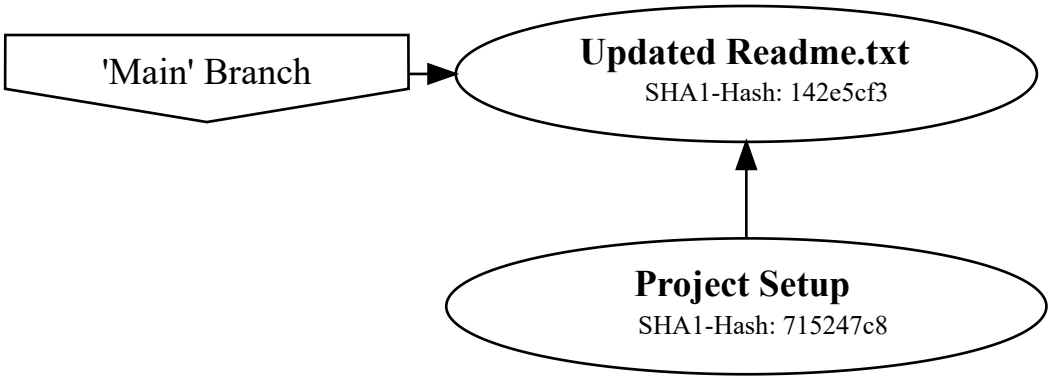
> CS 发言：我喜欢将 Git 提交历史视为从下到上或从左到右（定向），每个提交都有一个或多个父提交，但从不形成循环（非循环）。方向当然并不重要，重要的是例如 IntelliJ 的 Git 工具窗口如何表示流程。

什么是分支?

Git 中的分支只是*指向*提交的指针。

因此，创建分支意味着向特定提交添加一个指针，实际上是一个标签。反之，删除分支意味着删除*指针*。

而如果你一直在一个分支上提交，分支指针会像影子一样跟在*最新*的提交（即所谓的）后面。`tip`



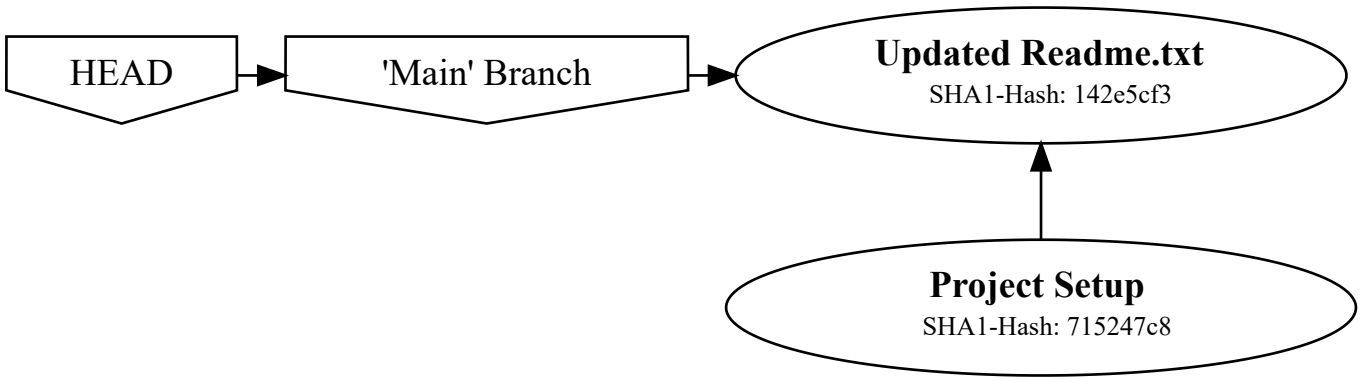
什么是头?

这几乎是一个棘手的问题，因为即使是Stackoverflow 上的正确答案也包含微妙的误解。

简短的回答是：

1. `HEAD`（全部大写）是指向您当前签出的内容的指针，并且在您添加新提交后将*跟随*。
2. 您当前签出的内容可以是：分支*或*提交*或*标记。
3. 通常，`HEAD`指向一个分支。

添加这条信息，这就是我们的存储库当前的样子：



如果你不信任图表，你也可以打开你的终端并执行以下命令：

```
# inside your project folder
cat .git/HEAD # use 'type .git\HEAD' on Windows, note the backslash

refs/heads/main
```

如您所见，您的*.git/HEAD*文件指向您的*main*分支（它又指向您的第二次提交）。

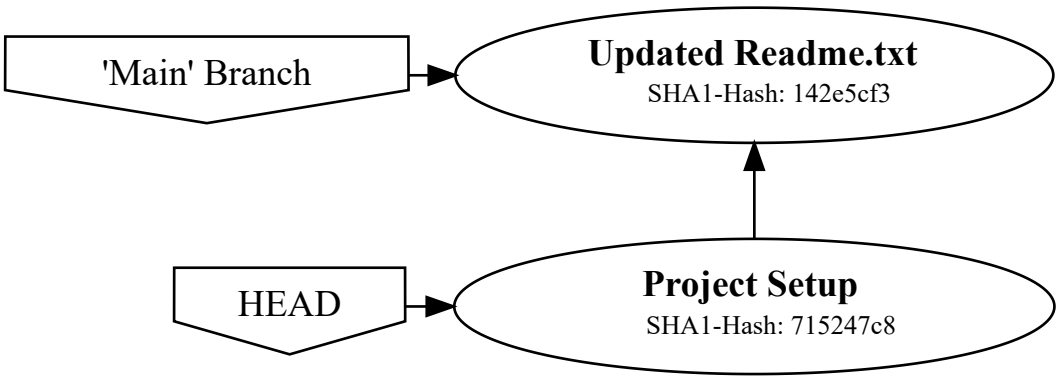
如果你继续在那个分支上提交，指针会像影子一样*HEAD*跟随指针。*branch*

什么是分离式 HEAD?

如果您现在直接签出修订版，会发生什么情况?

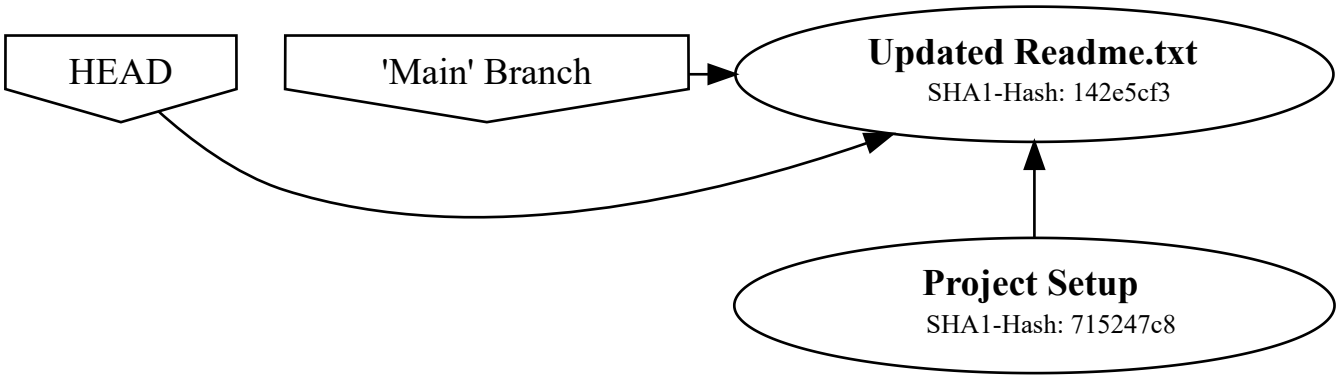
```
git checkout 715247c8426d3c16881539118e1eafeb38439b1c
cat .git/HEAD # use 'type .git\HEAD' on Windows, note the backslash

715247c8426d3c16881539118e1eafeb38439b1c
```

准确的说，now*HEAD*不是指向一个分支，而是直接指向一个commit!



事实上，您还可以直接检查第二个提交（具有分支指针的那个）。它具有相同的效果。

```
git checkout 142e5cf36d9f2047f24341883bd564b1d5170370
cat .git/HEAD # use 'type .git\HEAD' on Windows, note the backslash

142e5cf36d9f2047f24341883bd564b1d5170370
```



而以上两种情况，正是**detached HEAD**。这并不意味着您自己与 HEAD 分离，而是*HEAD*与**分支指针分离**。

这就是为什么每当您执行上面的两个检查时，Git 都会警告您您现在处于"分离的 HEAD"状态。

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:
...
```

为什么警告？因为你可以在这种状态下愉快地创建新的提交，但是当你切换分支时，你所有的提交都将消失/被垃圾收集。除非，您创建一个新分支来保留您的提交。

什么是 refs/heads?

*HEAD*这给我们留下了(uppercase) 和*heads*(lowercase)之间有什么区别的问题。

好吧，Git 存储库可以有（除其他外）、分支和标签。这些的总称是*references*or *refs*。

虽然*HEAD*指向您目前签出的任何内容，但 Git 存储库可以包含许多分支，并且每个分支都有一个*head*（小写），带有分支指针的提交。因此，您的"主要"分支的全名实际上是*refs/heads/main*，对于"feature-branch-1"，它应该是*refs/heads/feature-branch-1*等等。

实际上，转到您的终端并执行以下命令：

```
# inside your project folder
cd .git/refs
ls -ls # or dir on Windows

drwxrwxrwx 1 marco marco 512 Dec 28 10:56 heads
drwxrwxrwx 1 marco marco 512 Dec 28 09:54 tags
```

啊，我们的参考资料（分支、标签）！让我们检查一下 heads 文件夹。

```
cd heads
ls -ls # or dir on Windows

-rwxrwxrwx 1 marco marco 41 Dec 28 10:56 main
```

有我们的*main*分店！如果你愿意，继续创建其他几个分支，你会看到更多的文件弹出在这里（一个文件对应一个分支）。

Git 合并：幕后

最后，让我们仔细看看*merging*。

设想

在您的存储库中创建并切换到一个名为*merge_deep_dive*, 的新分支。然后，假设您最终确定了存储库（GPL）的软件许可证，并且您将*LICENSE.txt*使用它更新文件。

```
# create and checkout a new branch
git checkout -b merge_deep_dive

# update LICENSE.txt
```

```
echo "gpl-3.0" > LICENSE.txt
git add LICENSE.txt
git commit -m "Using a GNU General Public License v3.0 license"

[merge_deep_dive f05e046] Using a GNU General Public License v3.0 license
 1 file changed, 1 insertion(+), 1 deletion(-)
```

现在，有点做作，但切换回并使用另一个许可证（Apache*main*）更新同一文件。*LICENSE.txt*

```
# checkout branch
git checkout main

# update LICENSE.txt
echo "apache-2.0" > LICENSE.txt
git add LICENSE.txt
git commit -m "Using a Apache license 2.0 license"

[main 02d5da3] Using a Apache license 2.0 license
 1 file changed, 1 insertion(+), 1 deletion(-)
```

当然，当您现在尝试将*merge_deep_dive*分支合并到您的*main*分支中时，您会遇到冲突。

```
# merging
git merge merge_deep_dive

# conflict time!
Auto-merging LICENSE.txt
CONFLICT (content): Merge conflict in LICENSE.txt
Automatic merge failed; fix conflicts and then commit the result.
```

为什么我们会发生这种冲突？作为人类，我们知道我们在试图合并的同一文件中编辑了同一行，但 Git 是如何知道的呢？

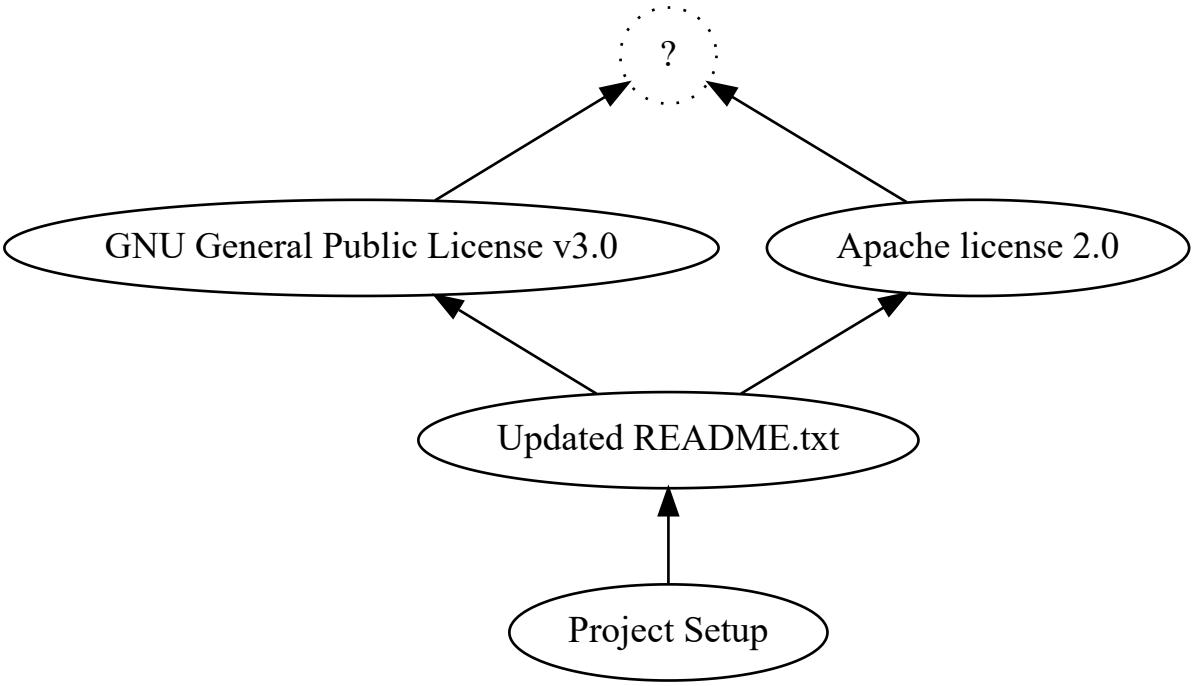
本质上，Git 必须逐行比较两次提交中您尝试合并的所有文件，并查看是否存在任何冲突的更改。（如果 SHA1 哈希匹配，比较可以短路）。

这有点高级，让我们详细看看它是如何工作的。

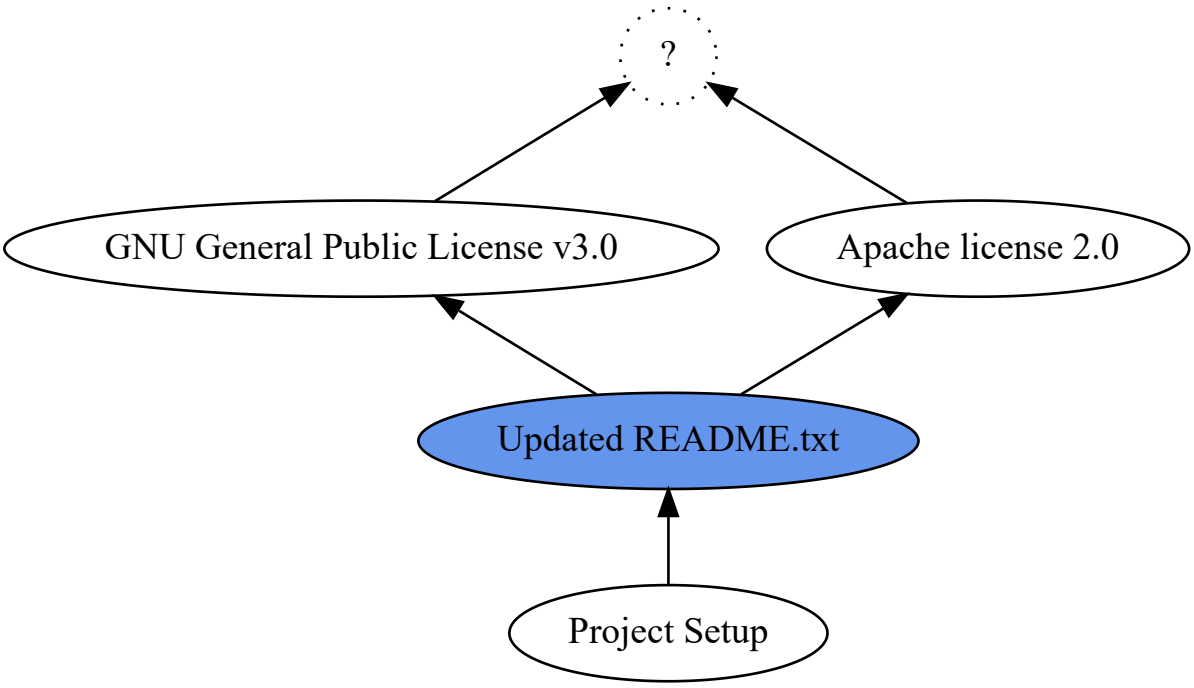
您不仅限于合并 2 个提交，所以实际上这可能是 n 个提交，例如合并 n 个分支。

Git 如何发现差异

这是合并时的提交图：您需要手动解决的内容 `license.txt`应该是什么。



当您触发该命令时，Git 将首先尝试为您尝试合并的两个提交`git merge`找到所谓的（最佳）共同祖先。`merge base`

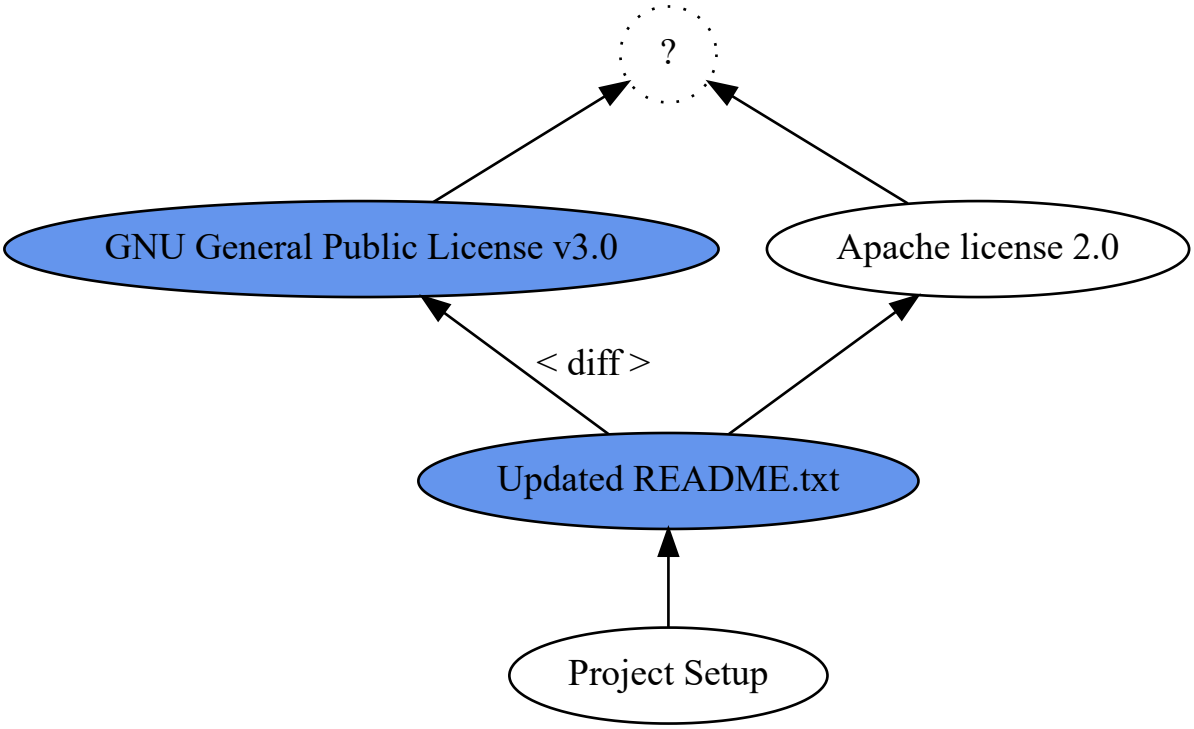


在我们的例子中，这就是`Updated readme.txt`提交。

然后，Git 将运行两个*diffs*，比较三个提交的所有文件和行 - 可能具有存储库文件的三个不同快照（base、branch1、branch2）。

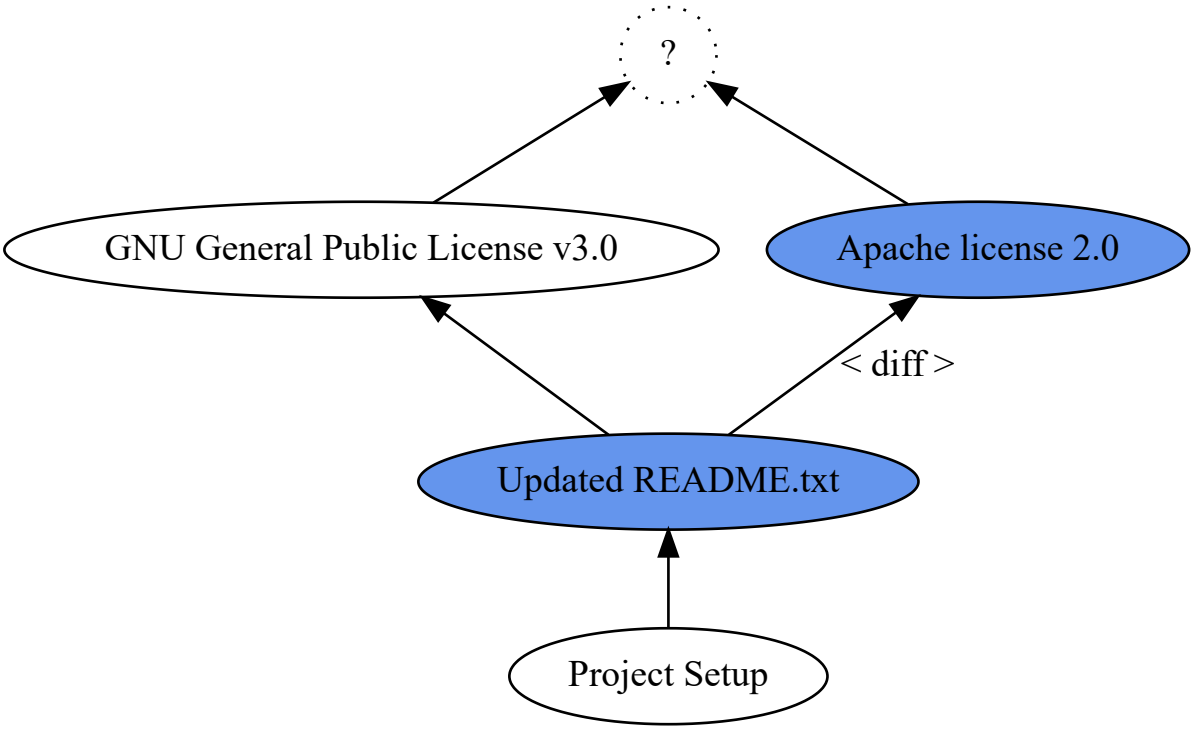
差异 1：合并基础与首次合并提交

Git 试图回答以下问题：提交`updated readme.txt`和`gnu general public license`提交之间发生了什么变化？

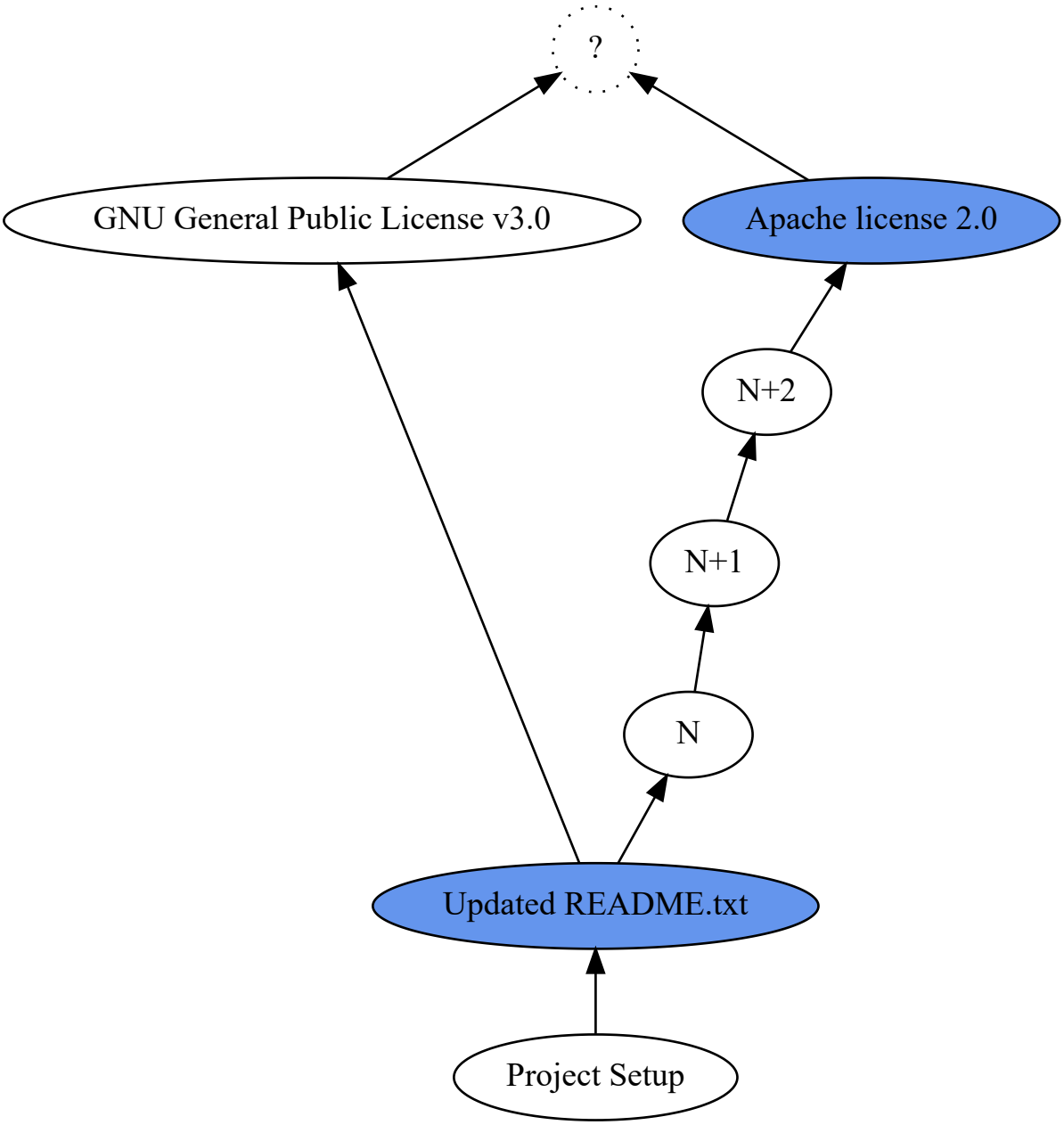


差异 2：合并基础与第二次合并提交

然后 Git 尝试回答下一个问题：提交`updated readme.txt`和`apache license`提交之间发生了什么变化？



请注意，共同祖先和您尝试合并的提交之间有多少提交并不重要，只有这两者的文件会有所不同。



Git差异

为了计算两个提交之间的差异，有一个方便的`git diff`命令，它基本上逐行比较两个提交的所有文件，并且还处理诸如文件重命名之类的情况。

查看上面的命令行历史记录，快速记下（速记或长格式）提交 ID，您最终会得到类似这样的内容：

- **合并基础（更新的自述文件）**：
  142e5cf36d9f2047f24341883bd564b1d5170370
- **提交一（Apache 许可证）**：
  02d5da3168da45d812679eb2736b73bdc7e8b3cf
- **提交二（GNU 公共许可证）**：
  f05e046cc5e29cc76d7414d3d0a75f2483693b5d

然后，执行*git diff*两次。

```
# git diff --find-renames <hash-of-X> <hash-of-Y>

git diff 142e5cf36d9f2047f24341883bd564b1d5170370 02d5da3168da45d812679eb
```

您将获得以下输出。

```
diff --git a/LICENSE.txt b/LICENSE.txt
index ddd3b7b..30d05dd 100644
--- a/LICENSE.txt
+++ b/LICENSE.txt
@@ -1 +1 @@
-"-TODO-"
+"apache-2.0"
```

再次针对第二个差异。

```
# git diff --find-renames <hash-of-X> <hash-of-Y>

git diff 142e5cf36d9f2047f24341883bd564b1d5170370 f05e046cc5e29cc76d7414d
```

这导致以下输出。

```
diff --git a/LICENSE.txt b/LICENSE.txt
index ddd3b7b..0da1e62 100644
--- a/LICENSE.txt
+++ b/LICENSE.txt
@@ -1 +1 @@
-"-TODO-"
+"gpl-3.0"
```

要完全理解 diff 的输出，您可以阅读这个很棒的Stackoverflow Answer。

要点是，两次提交都更改了第一*LICENSE.txt*行的文件。他们删除了该*-TODO-*行，并各自添加了自己的不同行。

Git 将如何处理差异结果

Git 的最终工作是生成一个合并提交，这意味着它将获取您`merge base`的更改列表并将其应用于该基础。

问题：Git 执行了两个差异，因此它现在有两个更改列表，它需要将这些更改合并到一个最终列表中。这是它的工作原理：

- 如果两个 diff 输出中都存在更改，则需要确保仅应用这些更改一次。删除`-TODO-`线在我们的示例中就是这样的候选者，它不能被删除两次。
- 如果更改仅出现在一个差异中并且不冲突，则只需应用更改即可。
- 但是，如果两个差异中存在冲突的更改（在我们的示例中，`+"apache-2.0"`vs`+"gpl-3.0"`在同一行的同一文件中），那么 Git 无法合理地做任何事情，它需要提示用户（您！）进行手动解决。

它是如何做到的？

合并标记

Git 通知您有关特定文件中的合并冲突的方式是将*合并标记*分别放入您的冲突文件中：`<<<<<<<`和`>>>>>>>`。

让我们在合并期间再看一下您的`LICENSE.txt`文件。

```
## or 'type LICENSE.txt' on Windows
cat LICENSE.txt
```

这将导致如下输出：

```
<<<<<<< HEAD
"apache-2.0"
=======
"gpl-3.0"
>>>>>>> merge_deep_dive
```

过去我看到奇怪的同事（包括我自己）在看到合并标记 (`<<<<<<<`或`>>>>>>>`) 时惊慌失措，但让我们逐行浏览这个版本的文件。

```
<<<<<<< HEAD
(...)
```

```
>>>>>>> merge_deep_dive
```

被合并标记包围的整个块仅表示在特定行（或周围行，不同分支之间：*HEAD / (pointing to main)*和*merge_deep_dive*.

```
"apache-2.0"
=======
"gpl-3.0"
```

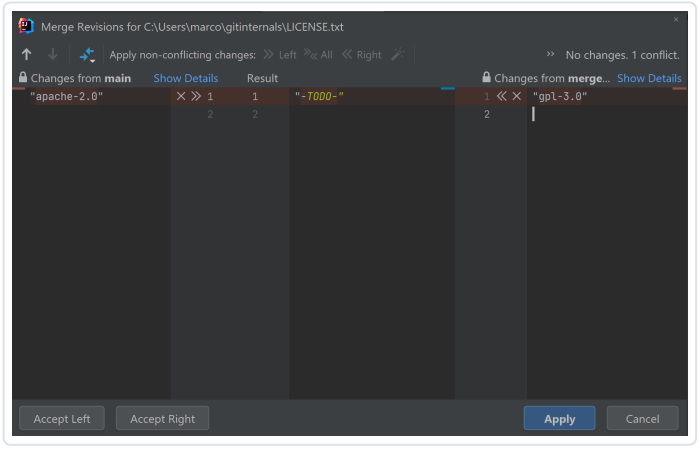
该行的两个不同版本由*=======*.

就是这样。合并标记本身没有什么特别之处，除了你，作为一个人，知道有一个你应该修复的冲突，IDE 可以采用这些标记并将它们转换为更具视觉吸引力的东西。这意味着什么？

好吧，例如 IntelliJ（或您最喜欢的 IDE）的实际工作是：

```
<<<<<<< HEAD
"apache-2.0"
=======
"gpl-3.0"
>>>>>>> merge_deep_dive
```

进入一个漂亮的小对话窗口，让您选择更改。



因此，无论何时您不小心关闭了 Intellij 的合并窗口并看到*>>>>>>>*合并标记，您都不应该惊慌。*什么都没发生*。

只需重新打开“解决”窗口，让 IntelliJ 重新解析标记并为您可视化它们，这样您就可以单击几个按钮，而不必在命令行上手动解决合并问题。

没错，我们还没谈及解决冲突呢。让我们现在就这样做。

> 如果你一直想知道“git merge --abort 做了什么”，答案（简化）：删除所有合并标记并将你的文件恢复到你最近提交的状态。

解决合并

要*解决*合并问题，您需要在`<<<<<< >>>>>>>`合并后将标记块（）替换为您希望该行读取的任何内容。

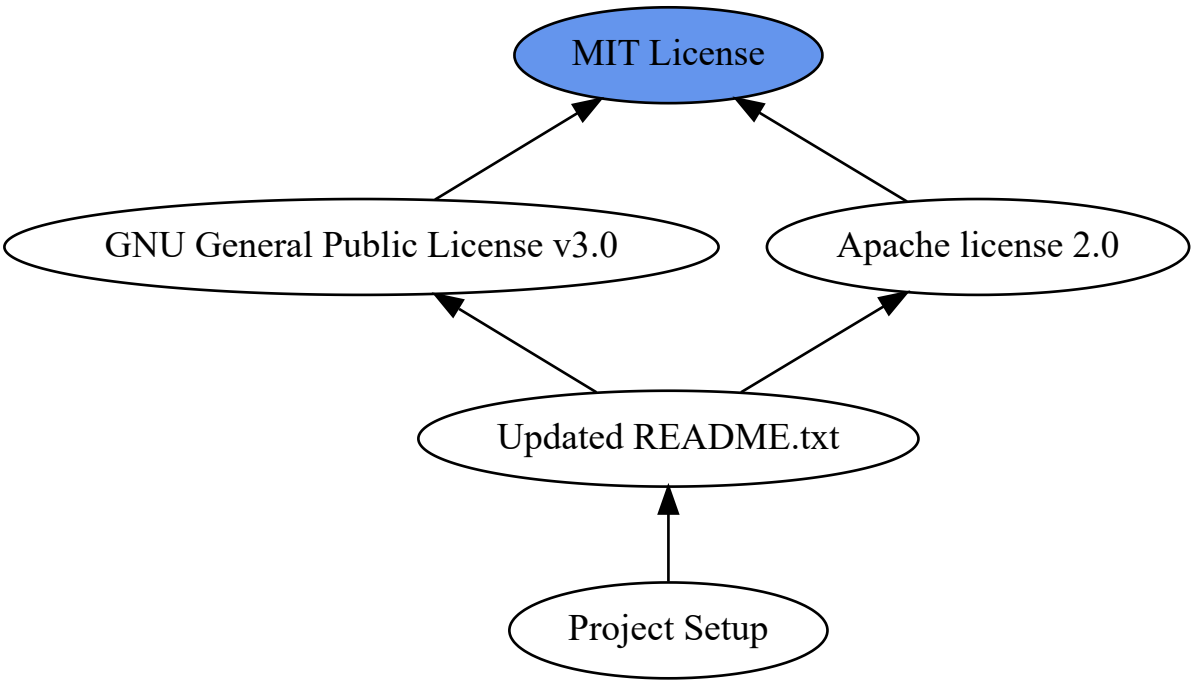
作为旁注（我已经在现实世界中看到过这种情况™）：合并标记并不特殊。他们只是几个角色。这意味着您*也可以*按原样提交您的文件，即在其中使用合并标记——虽然这很荒谬，但 Git 不会抱怨。（你的同事会喜欢的。）

由于您的`LICENSE.txt`文件仅包含一行，因此您只需在此处用全新的文本覆盖整个文件，从而“解决”合并冲突。

请记住，您不必*接受*之前提交的更改，您也可以选择忽略它们。在任何情况下，Git 都会简单地将文件的完整新版本存储在您的存储库中。

```
echo "MIT No Attribution License" > LICENSE.txt
git add LICENSE.txt
git commit -m "solved the license challenge merge"

[main 2a7be29] solved the license challenge merge
```

这会将您的存储库的 DAG 更改为如下所示，引入合并提交`MIT license`。



同样，请注意合并提交（`MIT license`）本身并不特殊，除了它有 2 个祖先或"父母"。如果您选择合并 5 个分支，您将有 5 个"父级"——是的，使用 Git，您可以同时合并任意数量的分支。

概括

这就是 Git 的合并魔法！总结：

- 查找合并基础（共同祖先）。
- 逐行比较 3 个（或更多）提交的所有文件（如果它们具有不同的 SHA1 和）。
- 生成最终的更改列表并将其应用于合并基础，以创建新的提交。

毕竟不是太棘手。

有趣的是：`git merge`它也用于 cherry-picks 和 rebases，所以我们可以在下一节中直接重用我们新发现的知识。

Git Cherry-Pick：幕后花絮

Cherry-Picks 很有趣，因为他们*觉得*您可以随机提交，然后以某种方式将其*克隆*到当前提交之上。

但这并不是樱桃挑选的工作方式。事实上，如果您*还没有*阅读Git Merge：幕后花絮，请先阅读。

为什么？

因为 a`git cherry-pick`本质上是 a `git merge`。莫名其妙？我也是。

设想

返回您的终端窗口并检查`updated readme.txt revision`. 你会得到什么？对，一个**独立的 HEAD**。

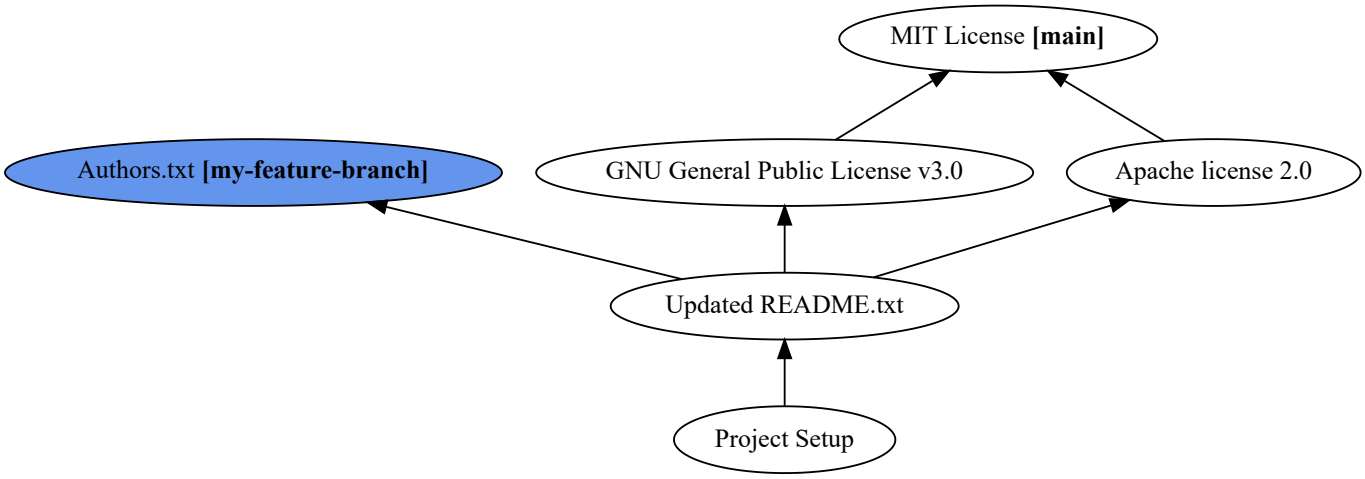
添加一个新文件并再次`AUTHORS.txt`更改的内容。`LICENSE.txt`

```
# switching into detached head state
git checkout 142e5cf36d9f2047f24341883bd564b1d5170370

# adding a new commit
echo "Marco" > AUTHORS.txt
git add AUTHORS.txt
```

```
echo "proprietary" > LICENSE.txt
git add LICENSE.txt

git commit -m "added authors.txt and updated license"

[detached HEAD 6db2a07] added authors.txt and updated license
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 AUTHORS.txt

# creates a new branch to retain the commits you just created
git switch -c my-feature-branch
```

除了文件之外，您还创建了一个名为 的新分支*my-feature-branch*，以摆脱分离的 HEAD 状态。

您的存储库现在看起来像这样：



切换回您的*main*分支，然后挑选您的提交。

```
git checkout main

git cherry-pick 6db2a07

Auto-merging LICENSE.txt
CONFLICT (content): Merge conflict in LICENSE.txt
error: could not apply 6db2a07... added authors.txt and updated license
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

啊哈，所以我们遇到了合并冲突（因为我们的 LICENSE.txt 文件），所以我们不只是盲目地从精心挑选的提交中复制更改。

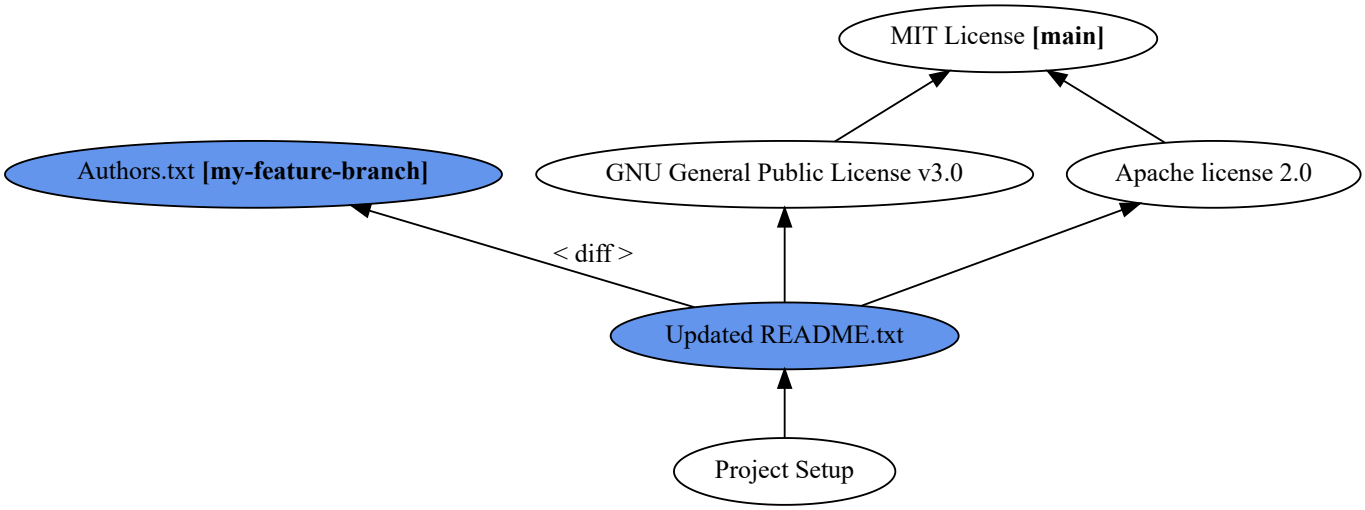
回想一章，我们知道我们需要（至少）两次提交来进行合并，但我们还需要一个*合并基础*，一个共同的祖先。

我们有精心挑选的提交和我们当前的提交，但什么是合理的合并基础？

答案：是樱桃采摘的父母！

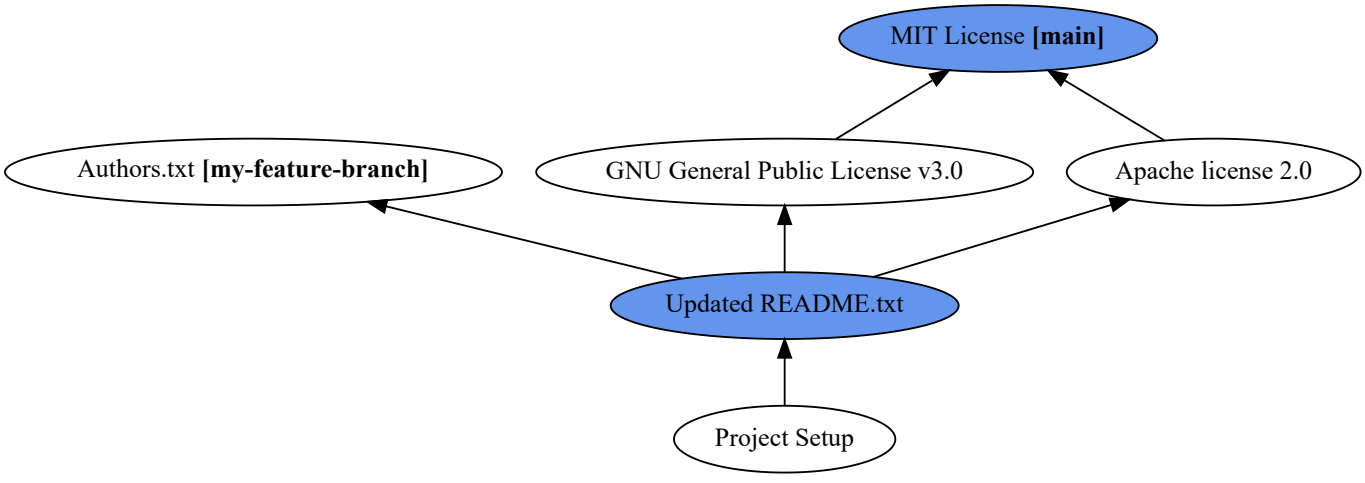
使困惑？这是将要执行的两个 git-diff。

差异 1：*updated readme.txt*对比*Authors.txt*



这个有点简单。我们得到一个差异列表，其中有人添加了一个*authors.txt*文件并修改了该*LICENSE.txt*文件。

差异 2：*updated readme.txt*对比*MIT license*



这有点令人困惑。合并基础和我们当前提交之间的差异，本质上会为我们提供从合并基础到.....我们当前提交的所有更改。

Git 的 Cherry-Pick 算法

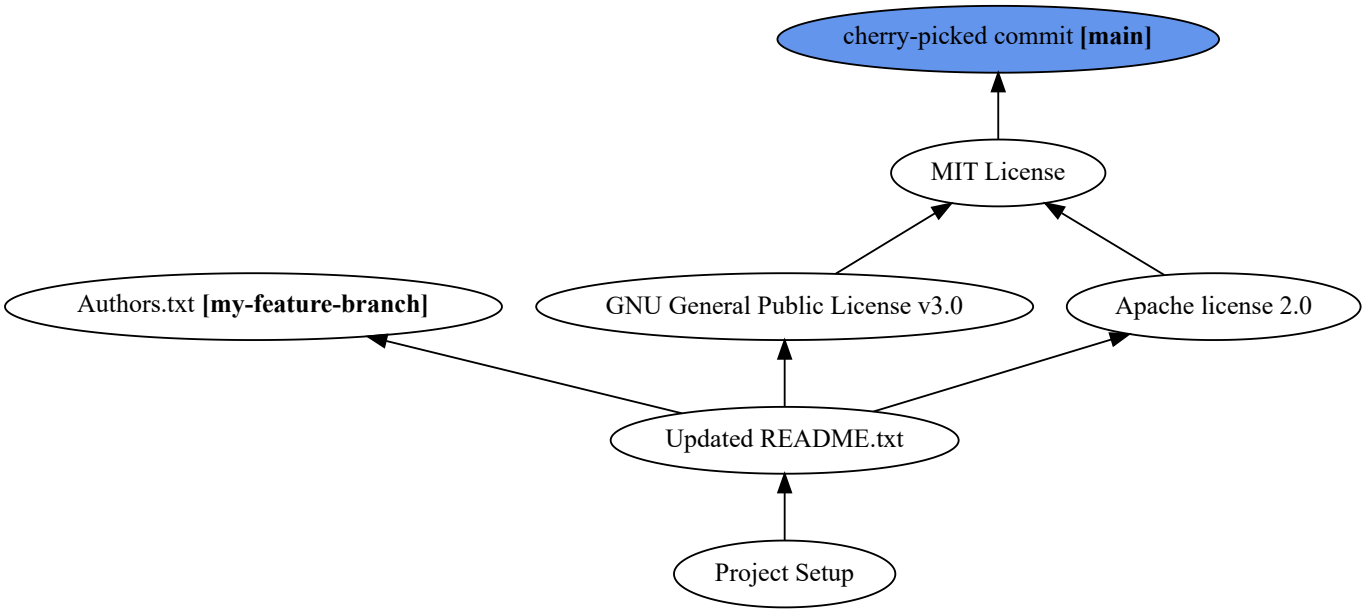
花点时间考虑清楚。这是发生了什么：

1. cherry-picked 的父项和 cherry-picked 提交之间的差异，将为您提供您尝试 cherry-pick 的更改。
2. cherry-picked 的父项和你当前的提交之间的差异，将为你提供 Git 需要应用的所有更改，从父项到你的当前状态。
3. 如果您有这两个差异列表，那么您可以简单地重新使用 git 的合并功能：首先应用 (2) 中的所有更改。然后应用您真正感兴趣的更改 (1)。这样，您将自动获得在您精心挑选的提交中发生冲突（删除、更改、修改）的文件的合并冲突。
4. Git 不会生成合并提交（合并多个祖先），而是只会根据合并的差异创建一个普通提交。

所以，让我们快速解决冲突（如果你愿意，看看里面的合并标记*LICENSE.txt*，看看它只是一个正常的合并过程）。

```
echo "Proprietary Wins" > LICENSE.txt
git add LICENSE.txt
git commit -m "cherry-picked commit"

[main 3081860] cherry-picked commit
 Date: Thu Jan 6 10:54:16 2022 +0100
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 AUTHORS.txt
```

合并冲突已解决，现在您将看到一个新的提交（不是多个祖先之间的合并提交，也不是克隆或其他东西）添加到您的提交历史记录中。



简而言之，这是一个樱桃选择：

- 与强制合并基础的合并，设置为 cherry-picked 的父级。
- 您将在当前分支中创建一个新提交，而不是合并提交，合并多个祖先。

> 你有没有试过挑选一个合并提交？在那种情况下会发生什么？合并基础是什么？在命令行上试试吧！

Git Rebase：幕后

想象一下，您有一些提交*L-M-N*要变基到 commit 上*A*。

前段时间，我认为这意味着将*L-M-N*提交作为一个单元，然后以某种方式*将*它们*A*按原样移动。

但正如您现在可能已经猜到的那样，这并不是 rebase 的工作原理。事实上，如果您还没有阅读Git Cherry-Pick：幕后花絮部分，现在就去读吧。

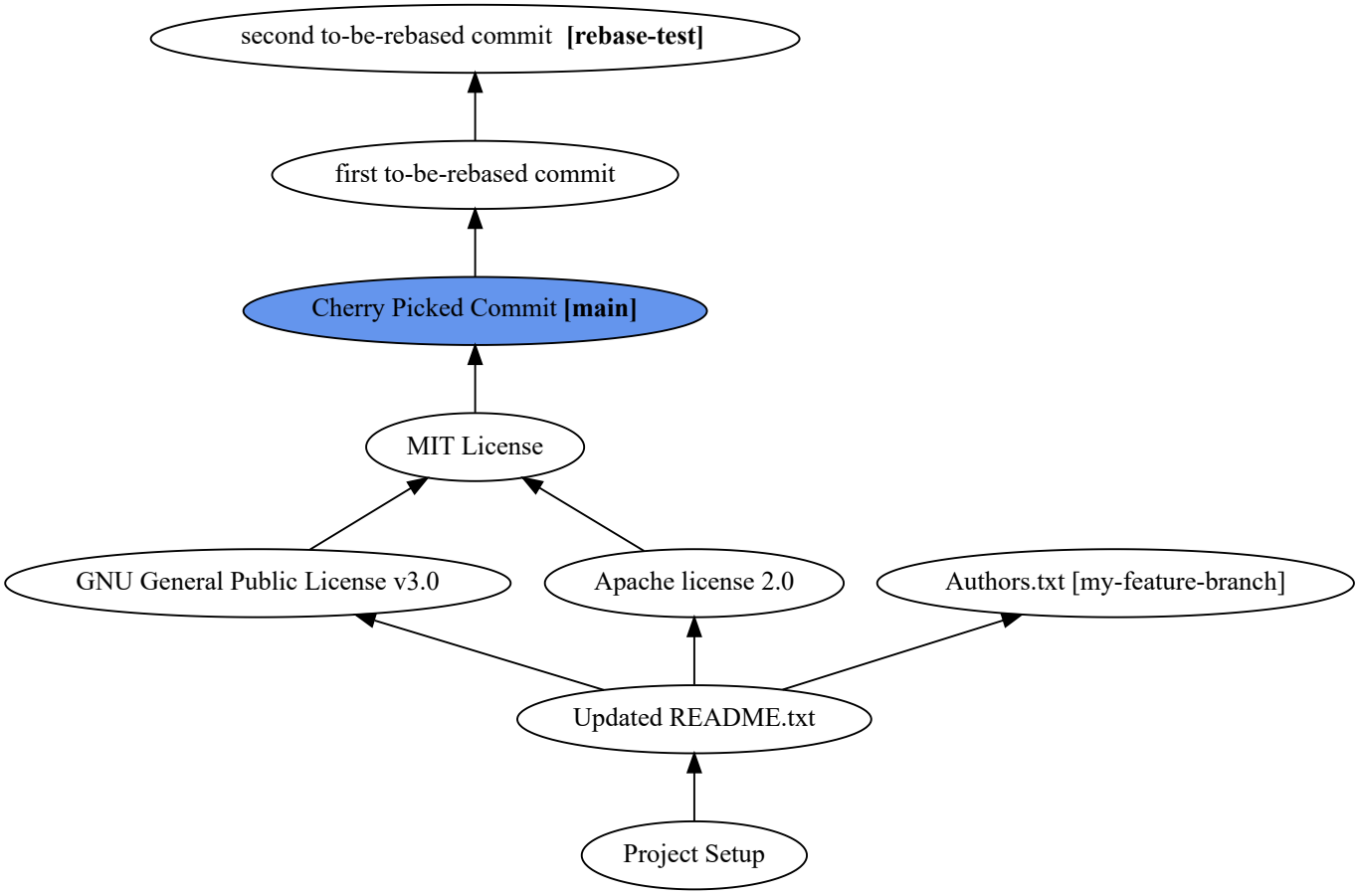
为什么？因为，变基实际上会在幕后进行挑选！让我们看看如何。

设想

```
# make sure you are on main branch and fork off a branch
git checkout main
git checkout -b rebase-test
```

```
## update LICENSE.txt
echo "first to-be-rebased commit" > LICENSE.txt
git add LICENSE.txt
git commit -m "first to-be-rebased commit"

[rebase-test afee899] first to-be-rebased commit
 1 file changed, 1 insertion(+), 1 deletion(-)

## update LICENSE.txt again
echo "second to-be-rebased commit" > LICENSE.txt
git add LICENSE.txt
git commit -m "second to-be-rebased commit"

[rebase-test 03da5d3] second to-be-rebased commit
 1 file changed, 1 insertion(+), 1 deletion(-)
```

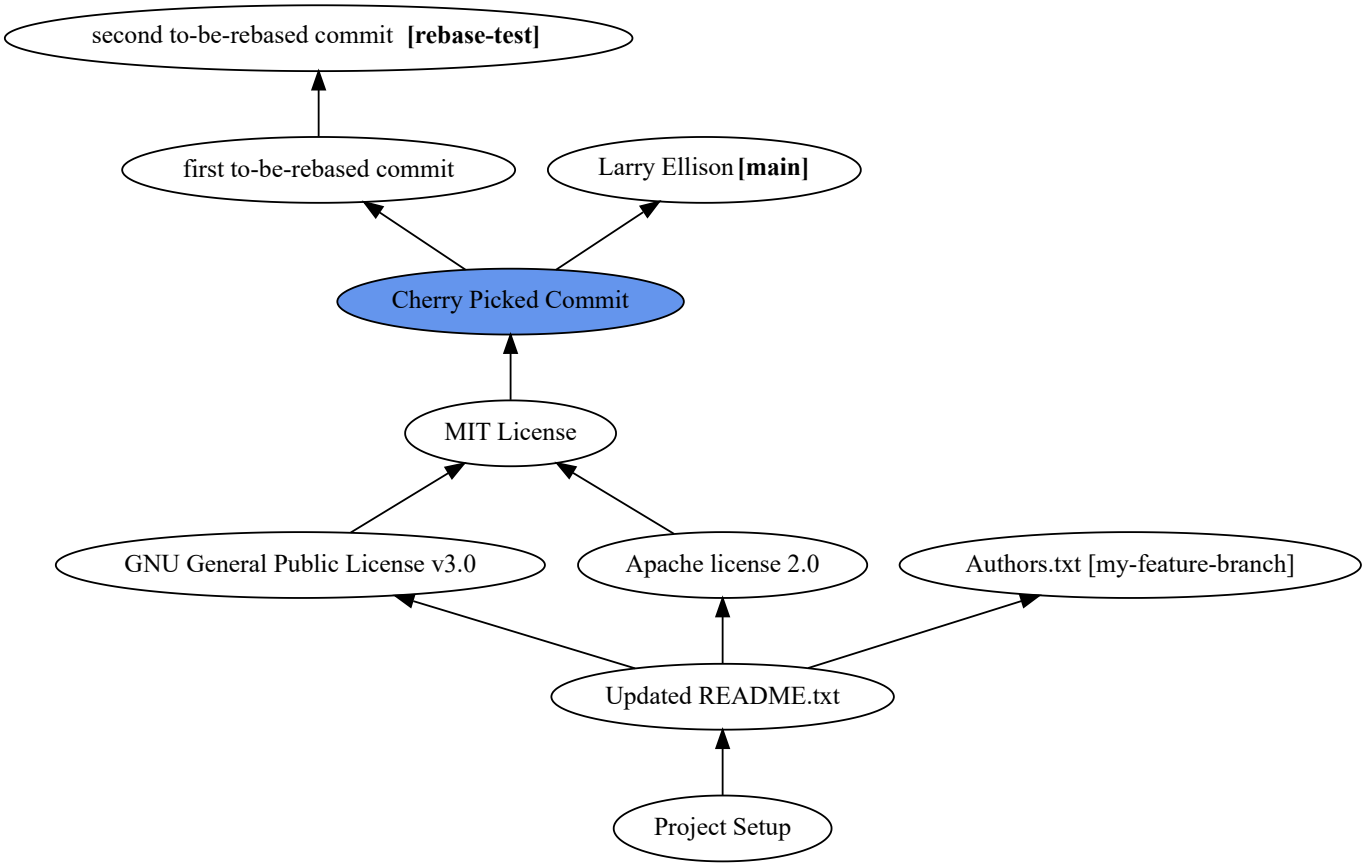
从*main*分支分支，您创建了一个名为 的新分支*rebase-test*。该分支包含两个提交，它们都更新*LICENSE.txt*文件（使用无意义的数据）。



现在我们有了那个分支，让我们回到*main**LICENSE.txt*并再次更新我们在那个分支中的内容。

```
# make sure to checkout the main branch again
git checkout main

# update license.txt
echo "Larry Ellison license" > LICENSE.txt
git add LICENSE.txt
git commit -m "Larry Ellison license"

[rebase-test ac6ed98] Larry Ellison license
 1 file changed, 1 insertion(+), 1 deletion(-)
```

这将导致这张图（点击放大）。



在您现在尝试将*rebase-test*分支变基之前*main*：您认为会发生什么？

你会得到任何合并冲突吗？如果有，有多少，为什么？

```
# make sure you are on rebase-test
git checkout rebase-test

# rebase the branch onto main
git rebase main
```

```
error: could not apply e03d753... added rebase-test branch
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase -
Could not apply e03d753... added rebase-test branch
Auto-merging LICENSE.txt
CONFLICT (content): Merge conflict in LICENSE.txt
```

好的，所以现在似乎只有一个合并冲突。*LICENSE.txt*让我们通过显示文件看看它是什么。

```
cat LICENSE.txt # type LICENSE.txt on Windows

<<<<<<< HEAD
"Larry Ellison license"
=======
"first to-be-rebased commit"
>>>>>>> ea02446 (first to-be-rebased commit)
```

所以，这与我们在分支上所做的*第一次*提交有冲突。*rebase-test*让我们使用这两行来修复合并冲突。

```
(echo Larry Ellison license && echo first to-be-rebased commit) > LICENSE
git add LICENSE.txt
```

当我们继续 rebase 时会发生什么？让我们找出来。

```
git rebase --continue
```

您将首先被提示为（看似）新的提交输入提交消息。选择您喜欢的消息，保存提交，然后您将直接返回到您的终端窗口，还有另一个合并冲突等待您解决。

```
[detached HEAD fee1098] first to-be-rebased commit
 1 file changed, 2 insertions(+), 1 deletion(-)
error: could not apply 49922e8... second to-be-rebased commit
```

```
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase -
Could not apply 49922e8... second to-be-rebased commit
Auto-merging LICENSE.txt
CONFLICT (content): Merge conflict in LICENSE.txt
```

这次怎么了？

```
cat LICENSE.txt # type LICENSE.txt on Windows

<<<<<<< HEAD
Larry Ellison license
first to-be-rebased commit
=======
"second to-be-rebased commit"
>>>>>>> 49922e8 (second to-be-rebased commit)
```

好吧。似乎 Git 现在试图对 进行变基*second-to-be-rebased commit*，导致又一次合并冲突。

让我们通过将所有三行放在我们的*LICENSE.txt*文件中来修复它，然后继续（或者更确切地说完成）rebase。

```
(echo Larry Ellison license && echo first to-be-rebased commit && echo ec
git add LICENSE.txt

git rebase --continue
```

系统将再次提示您考虑提交消息。这样做，保存新的提交，你会得到以下输出：

```
[detached HEAD f61d31f] second to-be-rebased commit
 1 file changed, 2 insertions(+), 1 deletion(-)
Successfully rebased and updated refs/heads/rebase-test.
```

哇，rebase 成功了！但实际上，这个过程很混乱，不是吗？

究竟刚刚发生了什么？让我们找出答案！

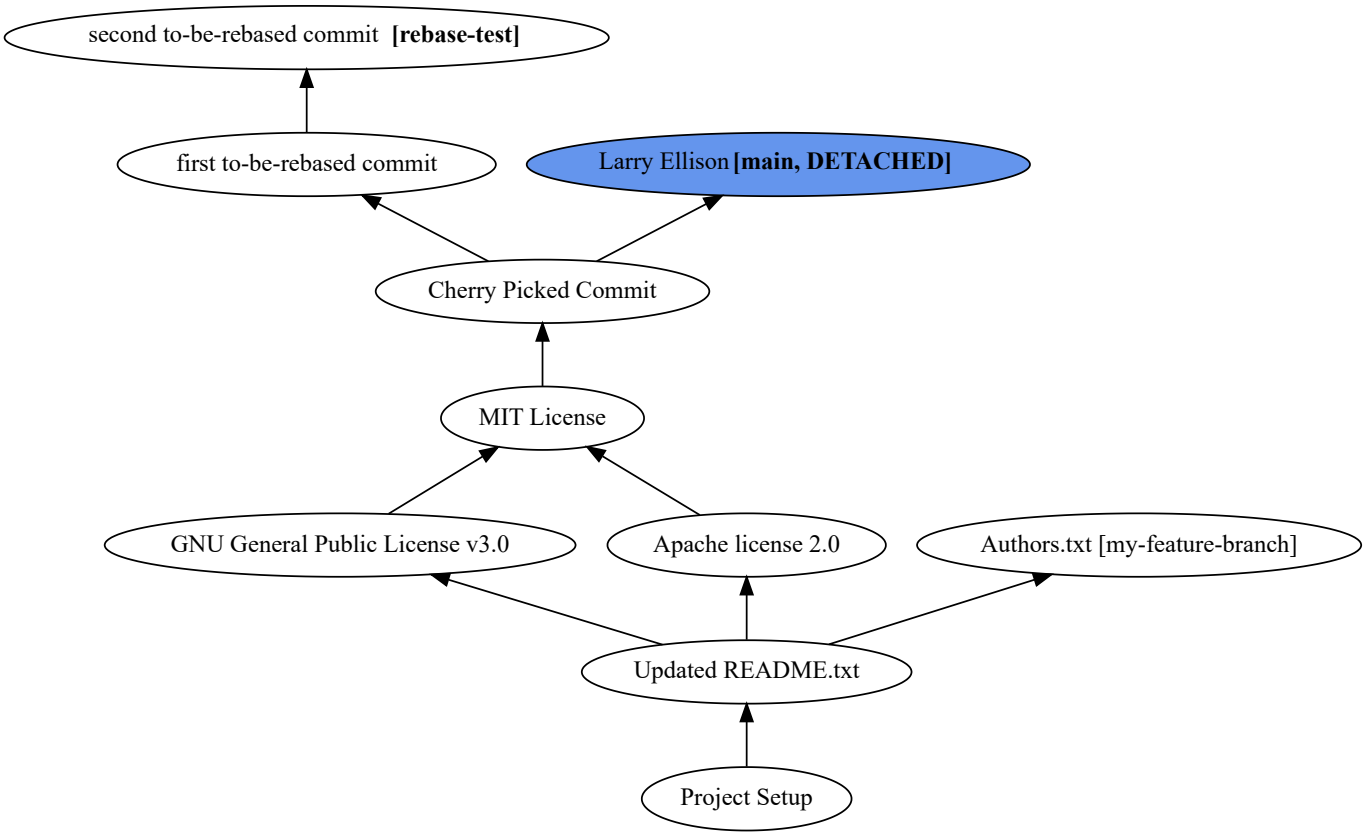
变基如何工作

首先，变基收集要变基的提交哈希列表，在我们的例子中是来自*`first-commit-to-be-rebased`*和的哈希*`second-commit-to-be-rebased`*。

在我的例子中，这些 id 是：

- **第一个被重新定位的提交：** afee899 ...
- **第二个被重新定位的提交：** 03da5d3d ...

然后，它检查我们的目标，将*`Larry Ellison`*提交作为*分离的 HEAD*。

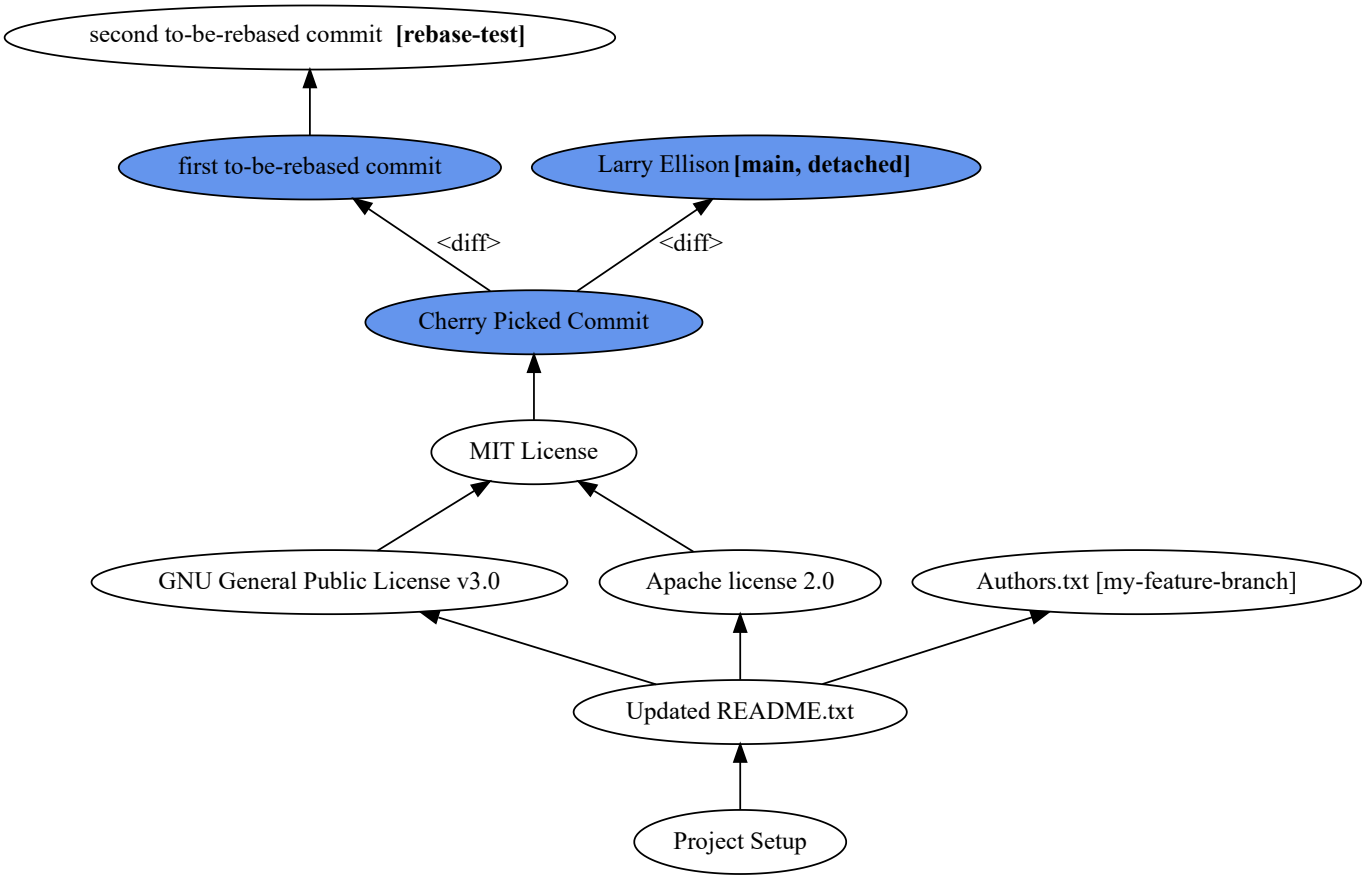


Git 现在将从该列表（afee899、03da5d3d）中挑选每个提交，一次一个，以适当的顺序放到*分离的 HEAD*上。

第一次变基

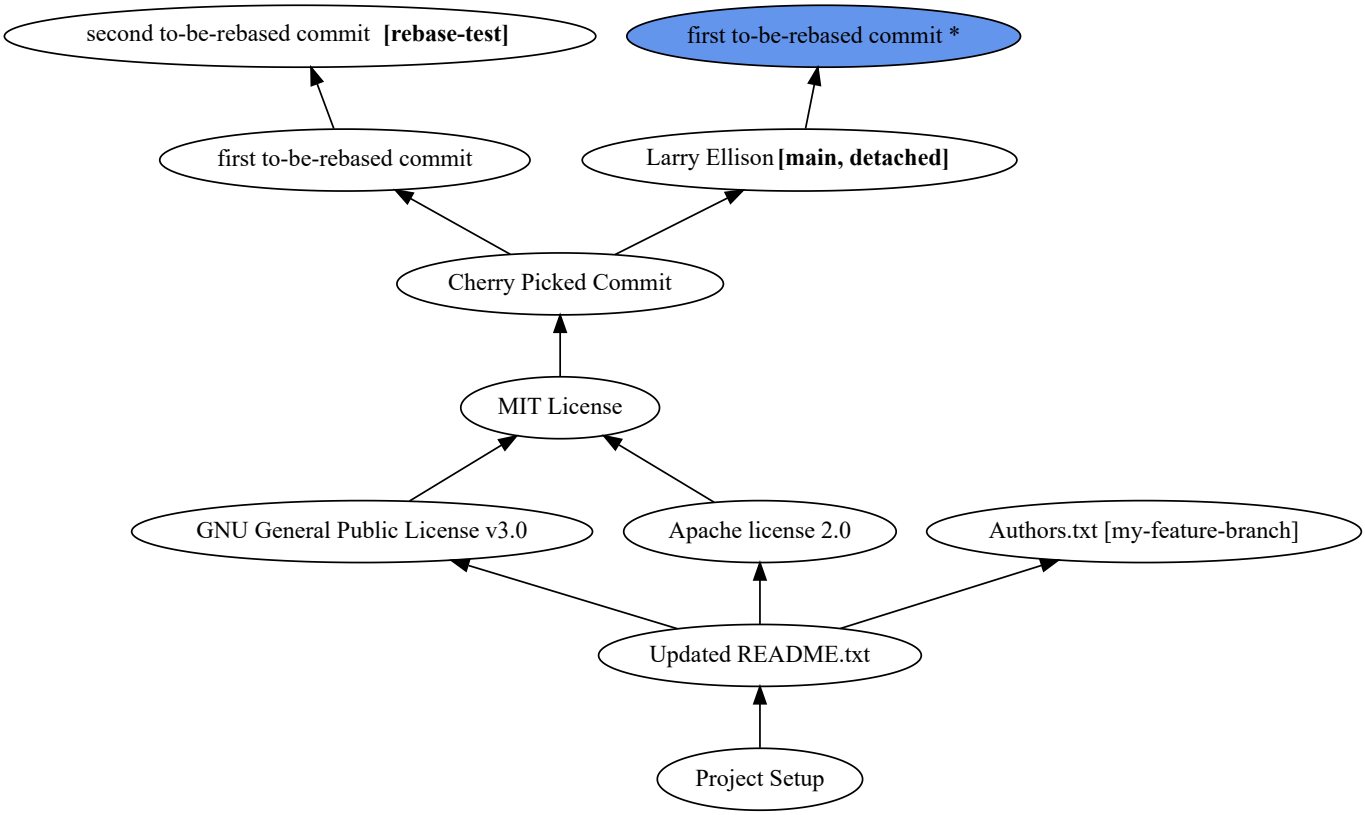
让我们从*`first-commit-to-be-rebased`*. 请记住，git cherry-pick 只是与特定合并基础的合并。所以，我们需要找到一个merge base，然后执行两个diffs。

- **合并基础：** *`Cherry Pick Commit`* . 它是待选提交的父级！
- **差异 1 候选人：** *`first to-be-rebased-commit`*
- **Diff 2 Candidate:** *`Larry Ellison`*，分离的 HEAD。



由于两个提交更改了同一行，在同一个`LICENSE.txt`文件中，您会遇到合并冲突，您需要解决这个问题。但即使没有合并冲突，您最终也总是会得到一个*新*的提交（您不仅仅是移动或复制精心挑选的提交）。

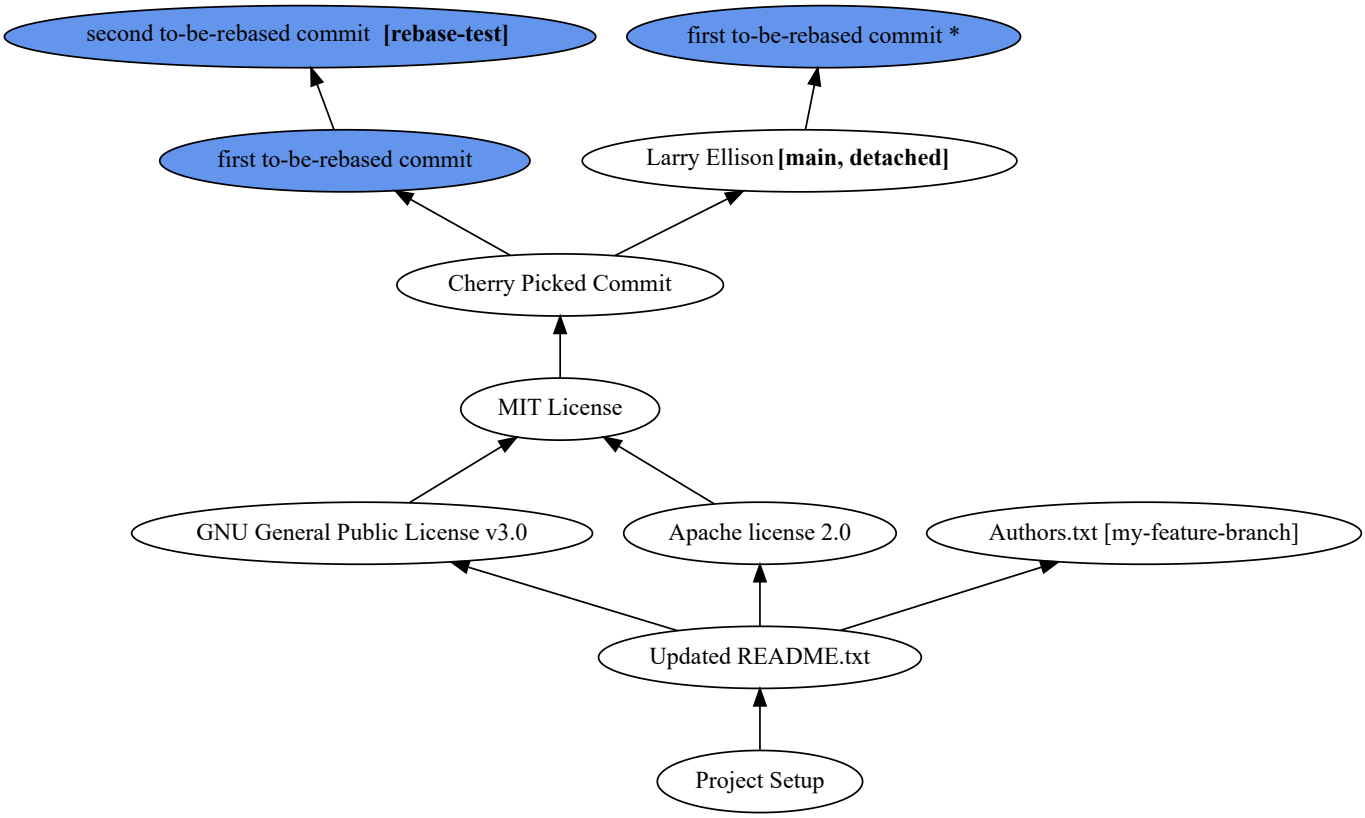
因此，在第一次挑选之后，我们的图表将如下所示。'*' 表示提交已被精心挑选。



第二次变基

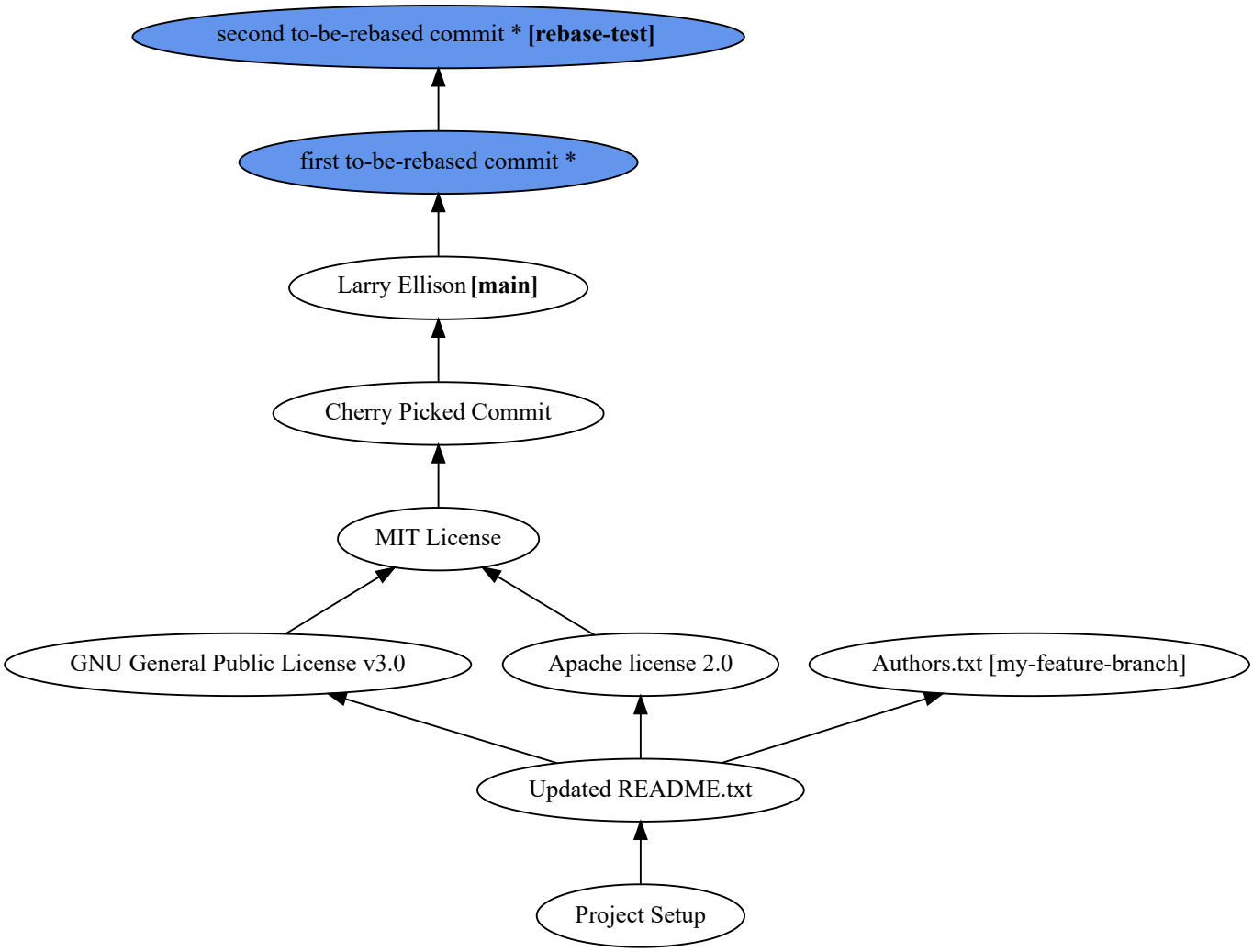
那是第二次挑选发生的时候。

- **Merge Base：** （*first-commit-to-be-rebased*要选择的提交的父级）
- **差异 1 候选人**：*second-to-be-picked-commit*
- **Diff 2 Candidate**：已经被精心挑选的*first-commit-to-be-rebased **



这是我们的第二次合并冲突发生的时候，因为两者*first-to-be-rebased-commits*已经不同并且*second-to-be-rebased-commit*再次更改同一文件中的相同行。

在解决了第二个合并冲突之后，我们的 rebase 已经完成，我们将得到这张图，没有分离的 HEAD。



实际发生的事情是 Git`rebase-test`从 our 中删除了分支名称`second-to-be-rebased commit`并将其粘贴到最终提交中，即“副本”，即`second-to-be-rebased commit *`.

然后它将 HEAD 指针重新附加到我们的“新”分支。这意味着，你不会再从`rebase-test`分支中看到你的原始提交，看起来它们已经移动了，但它们仍然在 Git 的 reflogs 中，你在新`rebase-test-branch`的中看到的其他提交也是全新的。

呸，很故事，是吧？

亲自在命令行上浏览示例几次，为期末考试做准备。

期末考试

这是你的最后一个问题。

`first_to_be_rebased`想象一下，在上面的 rebase 示例中，您会在第一次合并冲突期间完全接受来自提交的更改。

重新设置您的`second-to-be-rebased`提交是否会导致另一个合并冲突？为什么？

此外，您是否必须为这些重新定位的提交中的任何一个提供新的提交消息？为什么？

在评论中让我知道 :)

鳍

恭喜！那是很多。

到现在为止，您不仅应该对 Git 的合并、变基和 cherry-picks 工作原理有很好的理论理解，而且应该有很好的实践理解（您是在命令行上跟着做的，不是吗？）。

你最大的启示是什么？你最大的收获是什么？最重要的是：你现在对 Git 感觉更舒服了吗？

您希望本指南的下一次修订涵盖哪些主题？在下一节中查找想法列表。

在Twitter 上让我知道，或者直接发送电子邮件至marco@marcobehler.com。我很想听听你的消息！

谢谢阅读. Auf Wiedersehen。

未来话题

- 突出显示不同的合并策略（快速词、递归等）
- 对 squashes 和其他用例使用变基

致谢

非常感谢Chris Torek，他对 Git 的了解比我所知道的还要多，他在 Stackoverflow 上的回答对本文的形成有很大帮助。

还有 Joen Loeliger 和 Matthew McCullough，他们用 Git 编写了版本控制。

分享

注释

登录

添加评论

M↓ 降价

匿名评论 添加评论

赞成票 最新的 最旧的

Intelygenz
0分 · 10个月前

如果我接受来自 first-to-be-rebased 提交的更改，那么第二个 rebase 步骤将不会生成任何合并冲突作为 diff 2 候选者，即 first-to-be-rebased 提交和新挑选的第一个之间的差异- to-be-rebased commit *, 将是空的（两者中的 LICENSE.txt 文件是相同的）。

话虽如此，我不必为使用此策略的提交指定不同的名称。有什么我想念的吗？

米 **马可贝勒**
0分 · 10个月前

是的，正确 <拍手> :)

至于编辑提交消息，你的意思是什么时候有冲突或者什么时候没有冲突？

Intelygenz
0分 · 10个月前

两个都

米 **马可贝勒**
0分 · 10个月前

有趣的。如果我没记错的话，当我遇到冲突时，我肯定必须提供提交消息。如果没有，那就没有。在某些时候需要仔细检查。

? **匿名的**
0分 · 8个月前

我买了这个以为它是完整的。看起来它仍在进行中。那正确吗？是否会有可下载的 PDF 和视频课程？谢谢！

米 **马可贝勒**
0分 · 8个月前

您好，它实际上是完整的，而不是在作品中。在未来的修订中可能会添加一些额外的主题，但我想先等待用户的一些反馈。至于 PDF/视频，不，没有具体计划。你喜欢视频吗？

? **匿名的**
0分 · 10个月前

只是好奇这个提交来自哪里?

捕捉第一次提交的文件树/文件哈希

git 猫文件-p fe066d3f7568e13ef031b495e35c94be91b6366c

米 **马可贝勒**
0分 · 10个月前

哈，词不达意。这不是第一次提交的哈希值，而是在第一次提交期间截取的 README.txt 文件的哈希值。为此，您必须通过日志 -> 提交 -> 树 -> 文件。我会修正措辞，感谢您的反馈。