实验一 Git和Markdown基础

班级： 22物联1

学号： B20220305131

姓名： 黄志宇

Github地址：https://github.com/AYu1130/python_course

实验目的

1. Git基础，使用Git进行版本控制
2. Markdown基础，使用Markdown进行文档编辑
3. vscode的基本使用

实验环境

1. Git
2. VSCode
3. VSCode插件

实验过程记录

Git基础的学习

本部分是按照教材《Python编程从入门到实践》P440附录D进行的学习记录。

1. 安装配置Git

检查是否已经安装Git，可在终端使用命令`git --version`，如果已经安装将会输出版本号。

Git命令：

```
git --version
```

解释：

- 查看当前安装的Git版本

Git跟踪项目的修改是需要提供用户名和邮箱的，使用下面的命令来配置用户名和邮箱

Git命令：

```
git config --global user.name "用户名"
git config --global user.email "邮箱"
```

解释：

- 配置的用户名和邮箱将被用于标识用户在提交中的身份

如果要设置每个项目的主分支默认名称，可使用以下命令

```
git config --global init.defaultBranch main
```

解释：

- 全局配置，每个Git管理的新项目初始主分支为`main`

2. 项目以及忽略文件

进行版本控制的项目可能会有一些无需跟踪的文件，此时可创建名为`.gitignore`的特殊文件，可在文件中编写规则用以忽略特定的文件或目录

3. 初始化仓库

对于要使用Git进行版本控制的项目，首先需要在项目目录初始化仓库

Git命令：

```
git init
```

解释：

- 在项目目录初始化了一个空仓库，Git会在当前目录创建名为`.git`的子目录，其中包含了Git用来管理仓库的所有文件
- `.git`为隐藏目录

4. 检查状态

如果想知道哪些文件被修改、新增或删除，哪些文件已经暂存准备提交，可以使用以下命令检查状态

Git命令：

```
git status
```

解释：

- 用于显示工作区和暂存区的状态

5. 将文件加入仓库

要将文件加入仓库以此来跟踪的话，需要使用以下命令

```
git add .
```

解释：

- 将所有文件和修改添加到暂存区，不提交
- 没有在`.gitignore`文件中列出的
- 可将`.`替换成指定文件

6. 执行提交

已经将所有更改添加到暂存区，此时就要提交，需要执行以下命令

Git命令：

```
git commit -m "First commit"
```

解释：

- 将暂存区的更改提交到 Git 仓库
- `-m`用于直接指定本次的提交信息

7. 查看提交历史

如果要验证提交，可以使用以下来查看提交历史

Git命令：

```
git log
git log --pretty=oneline
```

解释：

- 输出提交历史，包括独一无二的引用ID，记录提交执行者，提交时间，提交指定的消息。
- `--pretty=oneline`用于指定显示：提交的引用ID和提交记录的消息

8. 第二次提交

在仓库中产生修改的文件如果没有添加到暂存区，使用`git commit`命令是无法提交这些修改的，此时可使用以下命令

Git命令：

```
git commit -am "second commit"
```

解释：

- 参数`-a`可让Git将仓库中所有修改了的文件都加入当前提交

9. 放弃修改

如果想要放弃最后一次提交之后的所有修改，变动很大的话手动撤销会很麻烦，想要恢复到最后一次提交的状态，可使用以下命令

Git命令：

```
git restore .
```

解释：

- 恢复到最后一次提交的状态

10. 检出以前的提交

检出并检查提交历史中的热河提交，既可以返回最后一次提交，也可以选择以前的提交而放弃此提交之后的所有更改

Git命令：

```
git log --pretty=oneline
git checkout %%%%%%
git switch -
```

解释：

- `git log`用于输出提交历史
- 使用`git checkout`并指定提交的引用ID前6个字符，分离头指针`HEAD`会移动到该提交
- `git switch -`能够回到`main`分支

Git命令：

```
git log --pretty=oneline
git reset --hard %%%%%%
```

解释：

- `git reset --hard`并指定提交的引用ID，能够恢复到从该提交开始的状态

11. 删除仓库

如果仓库的历史记录被弄乱了，在尝试过方法但无法恢复的话，可以考虑删除`.git`目录，然后重新创建仓库来对修改重新跟踪。

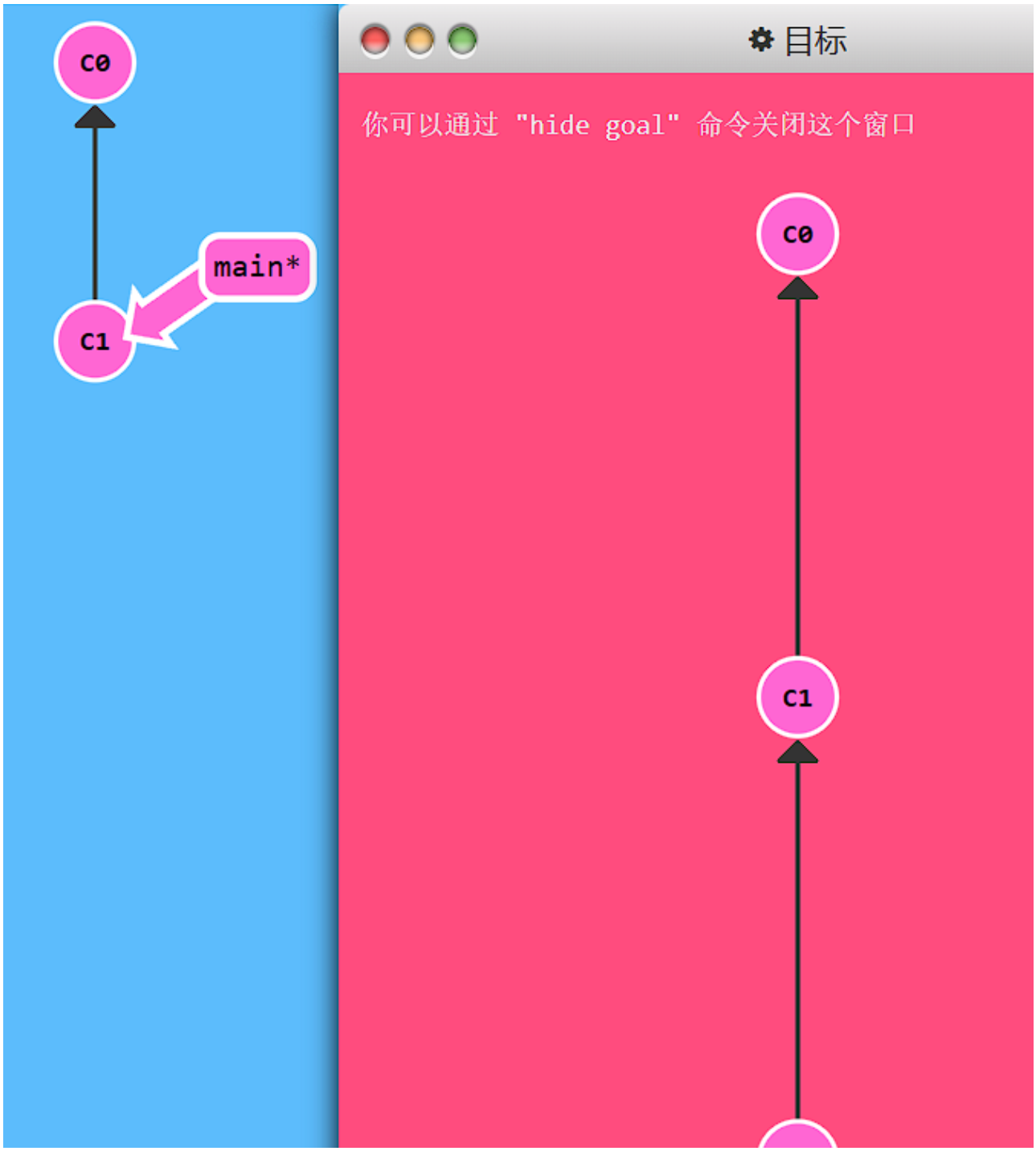
learngitbranching.js.org网站练习

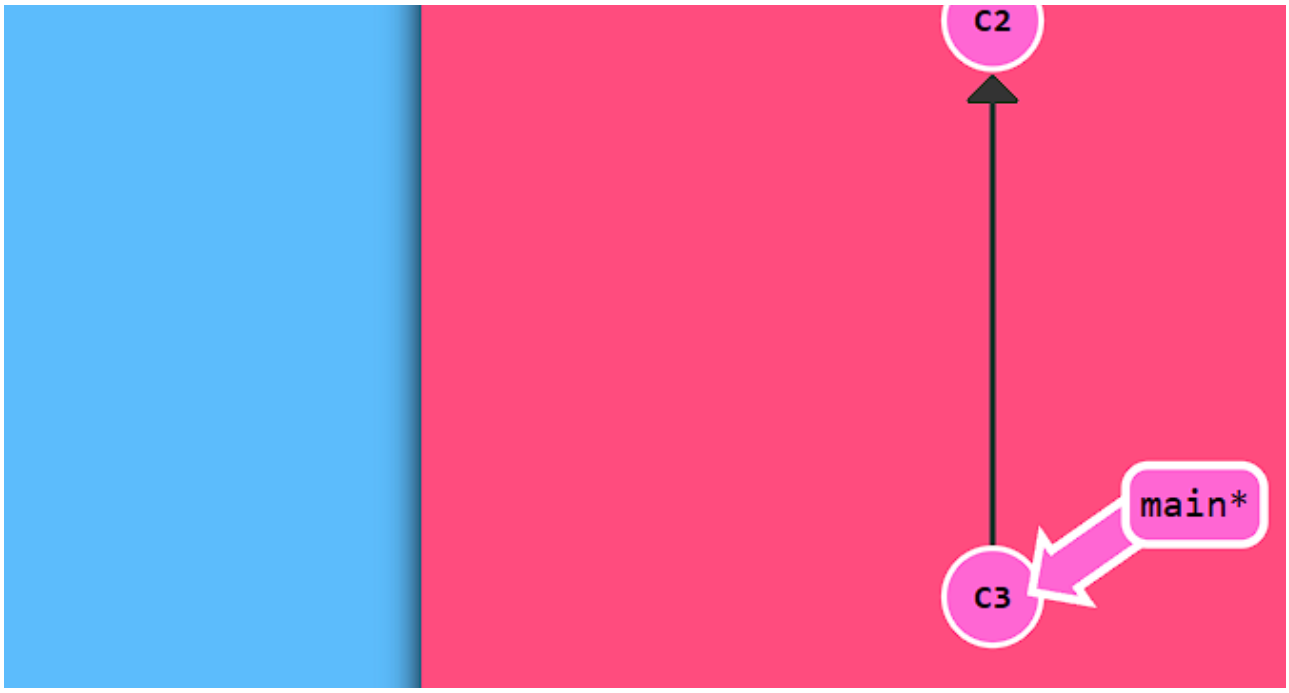
以下为在Learn Git Branching网站上的练习记录。

Git主要命令

1. Git Commit
   Git 希望尽可能地轻量提交记录，因此在每次进行提交时，它并不会盲目地复制整个目录。条件允许的情况下，它会将当前版本与仓库中的上一个版本进行对比，并把所有的差异打包到一起作为一个提交记录。

   任务目标：





Git命令：

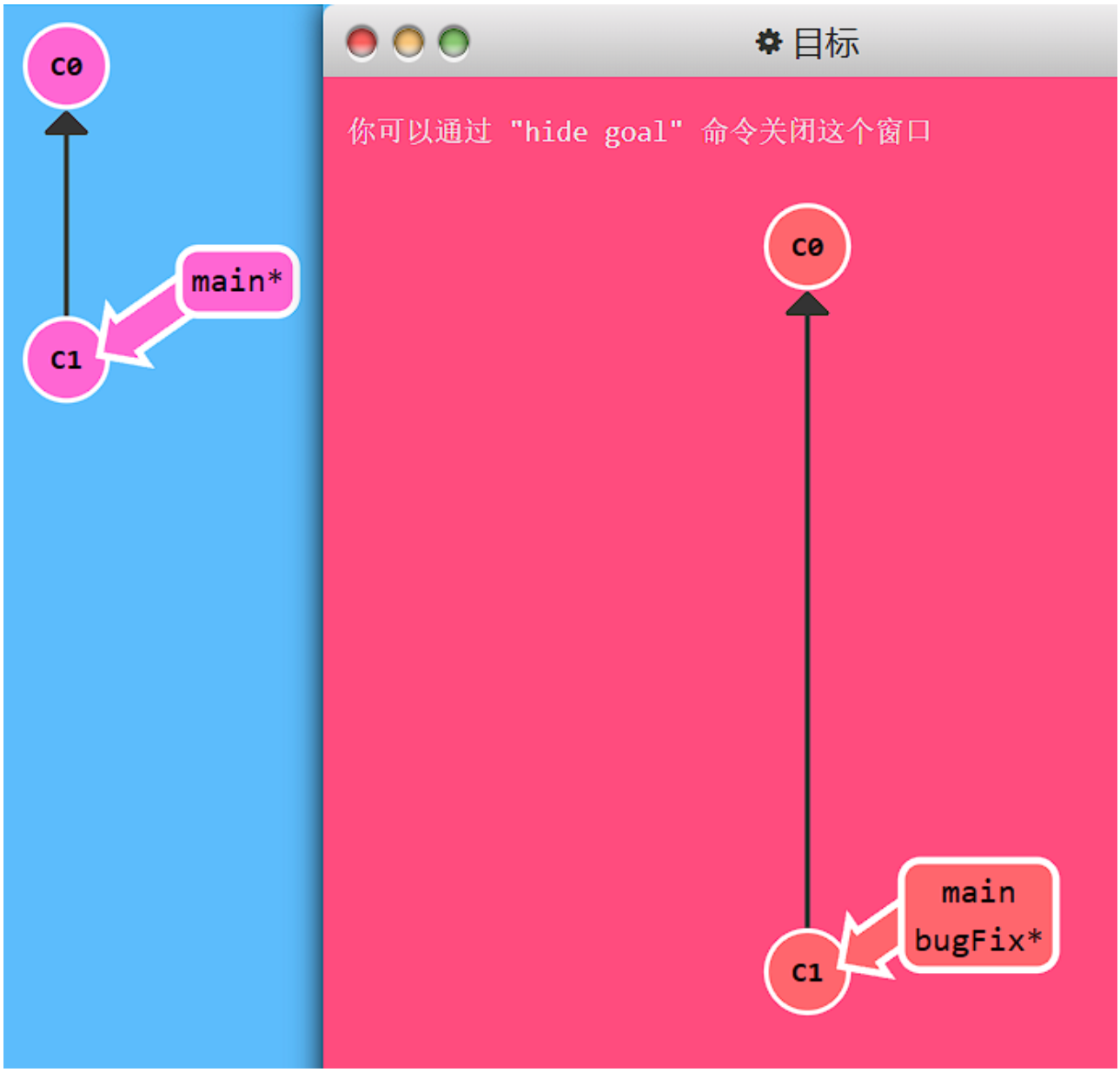
```
git commit
git commit
```

解释：

- 提交两次以达到任务关卡目标

2. Git Branch
Git 的分支也非常轻量。它们只是简单地指向某个提交纪录 —— 仅此而已。所以许多 Git 爱好者传颂：早建分支！多用分支！
这是因为即使创建再多的分支也不会造成储存或内存上的开销，并且按逻辑分解工作到不同的分支要比维护那些特别臃肿的分支简单多了。

任务目标：



Git命令：

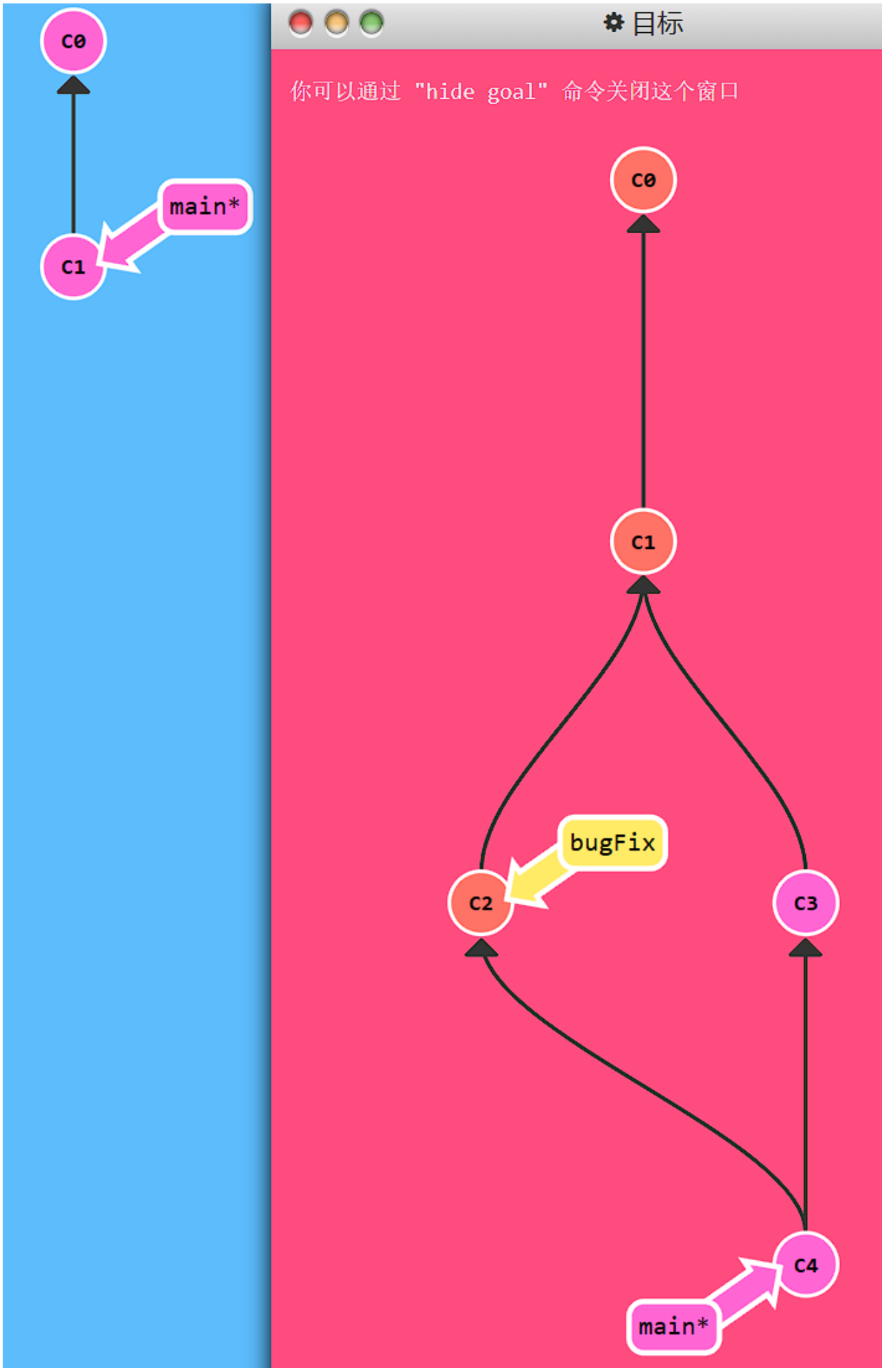
```
git checkout -b bugFix
```

解释：

- 创建并切换到分支bugFix
- 相当于git branch bugFix和git checkout bugFix两个命令

3. Git Merge 如何将两个分支合并到一起。就是说我们新建一个分支，在其上开发某个新功能，开发完成后再合并回主线。
在 Git 中合并两个分支时会产生一个特殊的提交记录，它有两个 parent 节点。翻译成自然语言相当于：“我要把这两个 parent 节点本身及它们所有的祖先都包含进来。”

任务目标：


C0
main*
C1
目标
你可以通过 "hide goal" 命令关闭这个窗口
C0
C1
bugFix
C2
C3
C4
main*

Git命令：

```
git checkout -b bugFix
git commit
git checkout main
git commit
git merge bugFix
```

解释：

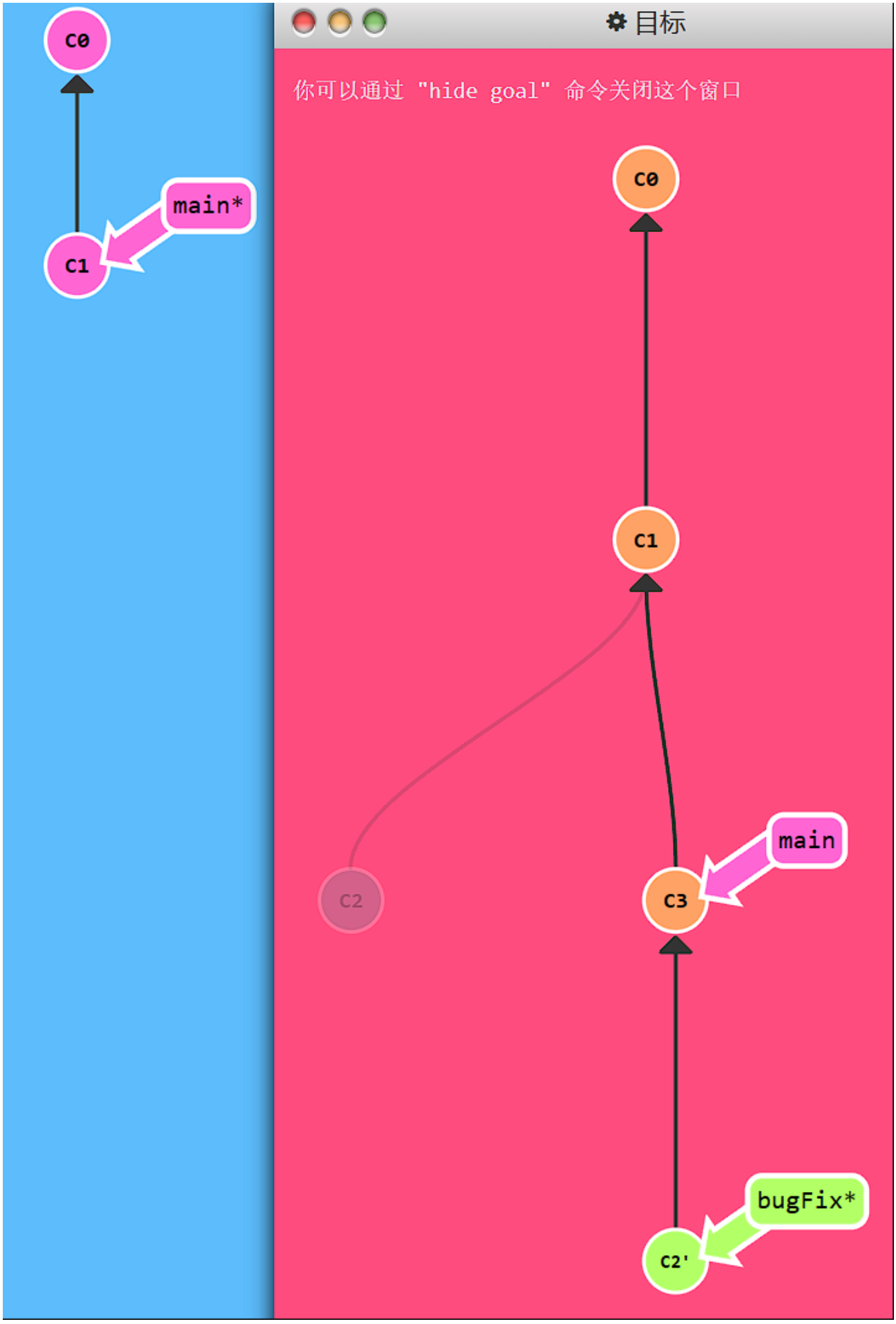
- `git checkout -b bugFix`创建并切换到分支`bugFix`
- `git commit`在分支`bugFix`中提交一次
- `git checkout main`切换到分支`main`
- `git commit`在分支`main`中提交一次。
- `git merge bugFix`将分支`branch`合并到分支`main`

4. Git Rebase

`git rebase`同样能合并分支。Rebase 实际上就是取出一系列的提交记录，“复制”它们，然后在另外一个地方逐个的放下去。

`Rebase`的优势就是可以创造更线性的提交历史。如果只允许使用 Rebase 的话，代码库的提交历史将会变得异常清晰。

任务目标：


C0
main*
C1
目标
你可以通过 "hide goal" 命令关闭这个窗口
C0
C1
C2
main
C3
bugFix*
C2'

Git命令：

```
git checkout -b bugFix
git commit
git checkout main
git commit
git checkout bugFix
git rebase main
```

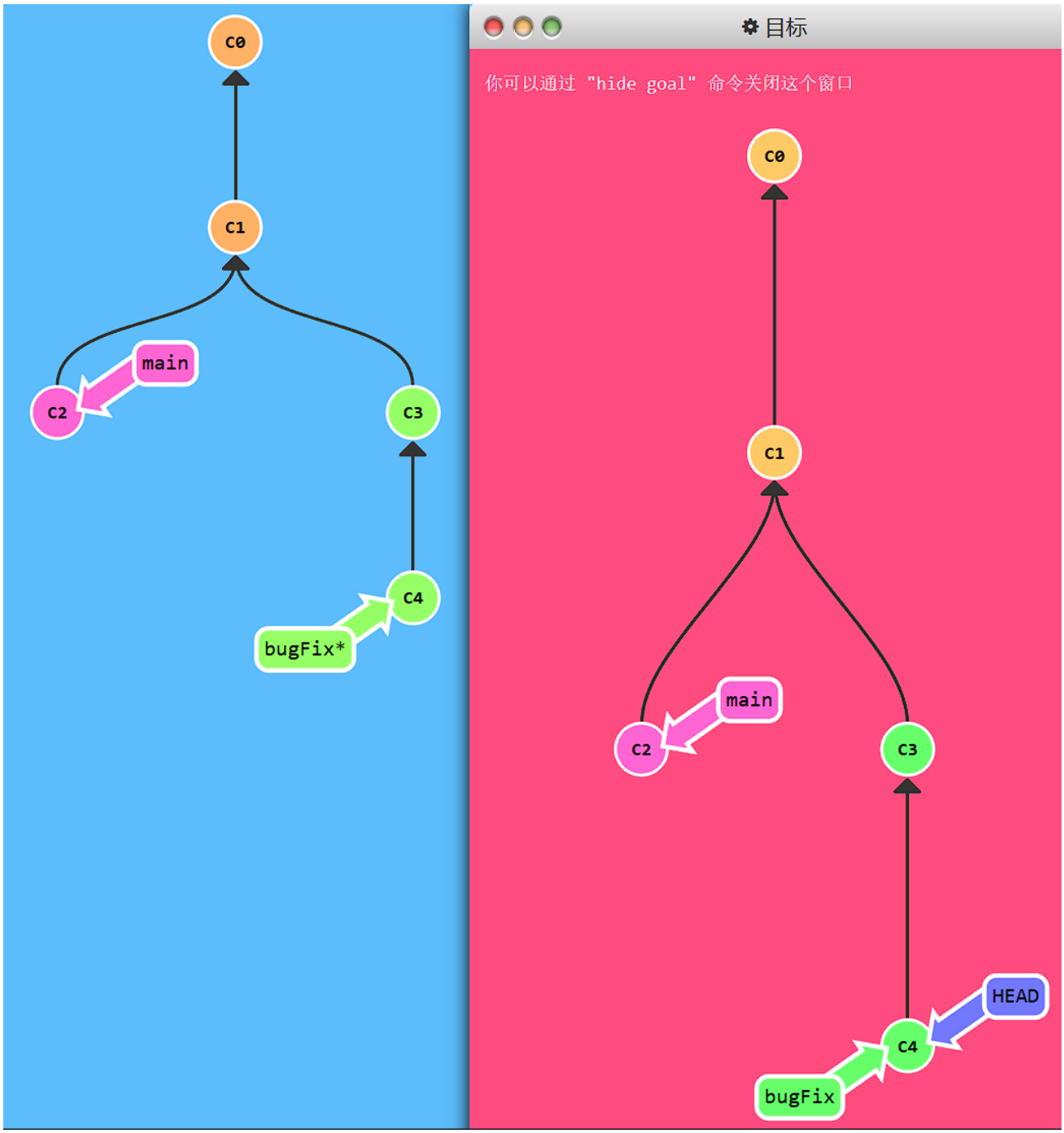
解释：

- `git checkout -b bugFix` 创建并切换到分支 `bugFix`
- `git commit` 在分支`bugFix` 中进行一次提交
- `git checkout main` 切换到分支 `main`
- `git commit` 在分支`main` 中进行一次提交
- `git checkout bugFix` 切换到分支 `bugFix`，在使用`rebase`进行合并前需要切换到分支`bugFix`
- `git rebase main` 将当前分支`bugFix`合并到分支`main`

Git 超棒特性

1. 分离 HEAD
   在接触 Git 更高级功能之前，有必要先学习在项目的提交树上前后移动的几种方法。
   HEAD 是一个对当前所在分支的符号引用 —— 也就是指向你正在其基础上进行工作的提交记录。
   HEAD 总是指向当前分支上最近一次提交记录。大多数修改提交树的 Git 命令都是从改变 HEAD 的指向开始的。
   HEAD 通常情况下是指向分支名的（如 bugFix）。在你提交时，改变了 bugFix 的状态，这一变化通过 HEAD 变得可见。
   分离 HEAD 意味着将其附加到提交而不是分支。如果在 HEAD 分离时提交，则提交将不属于任何分支，并且将完全无法访问（提交哈希除外）。

   任务目标：



Git命令：

```
git checkout C4
```

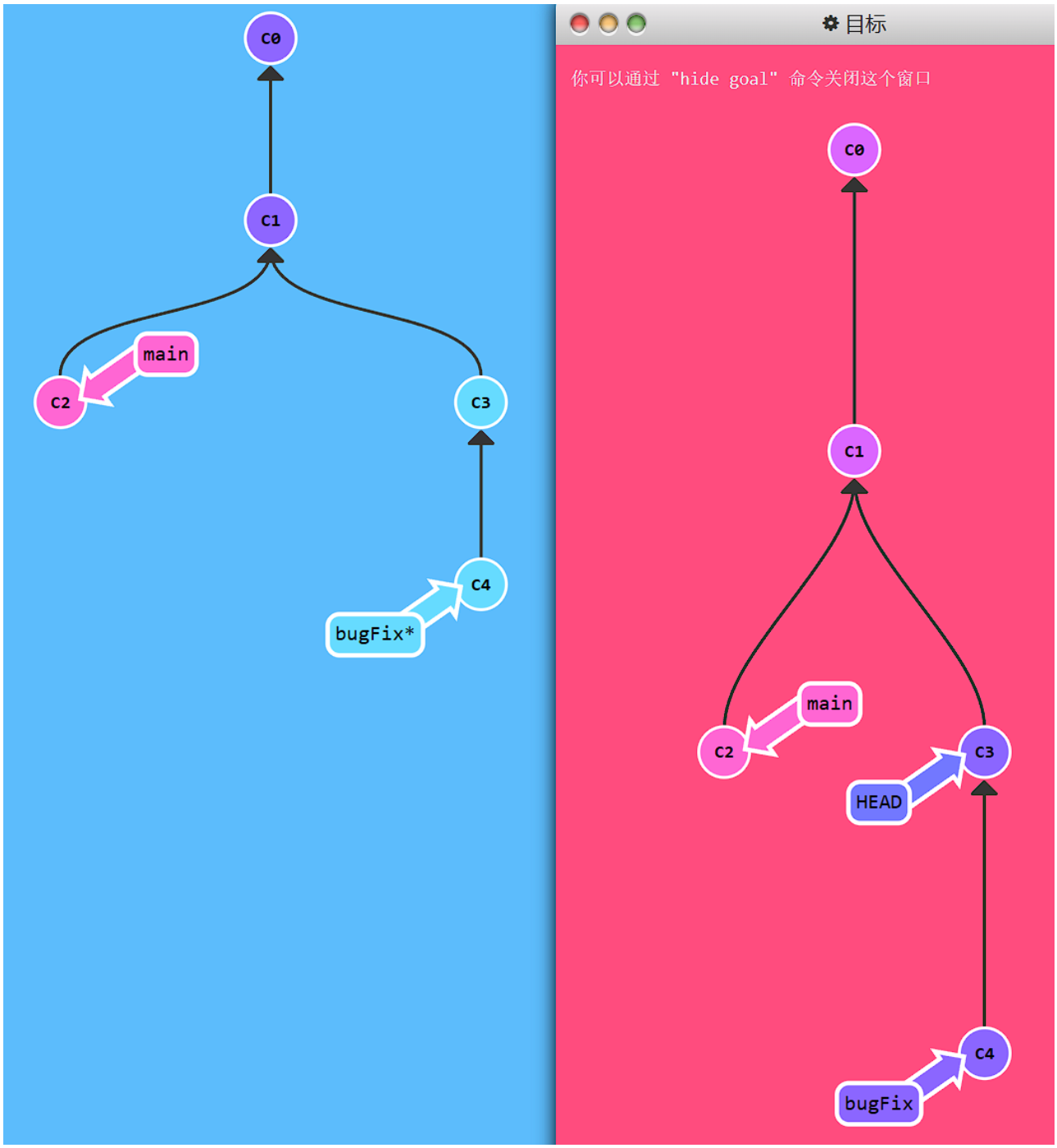
解释：

- `git checkout c4`将`HEAD`分离指向提交记录 C4

2. 相对引用(^) 通过哈希值指定提交记录很不方便，所以 Git 引入了相对引用。
使用相对引用的话，你就可以从一个易于记忆的地方（比如 bugFix 分支或 HEAD）开始计算。
这里有两个简单的用法：

- 使用 ^ 向上移动 1 个提交记录
- 使用 ~num 向上移动多个提交记录

任务目标：



Git命令：

```
git checkout bugFix^
```

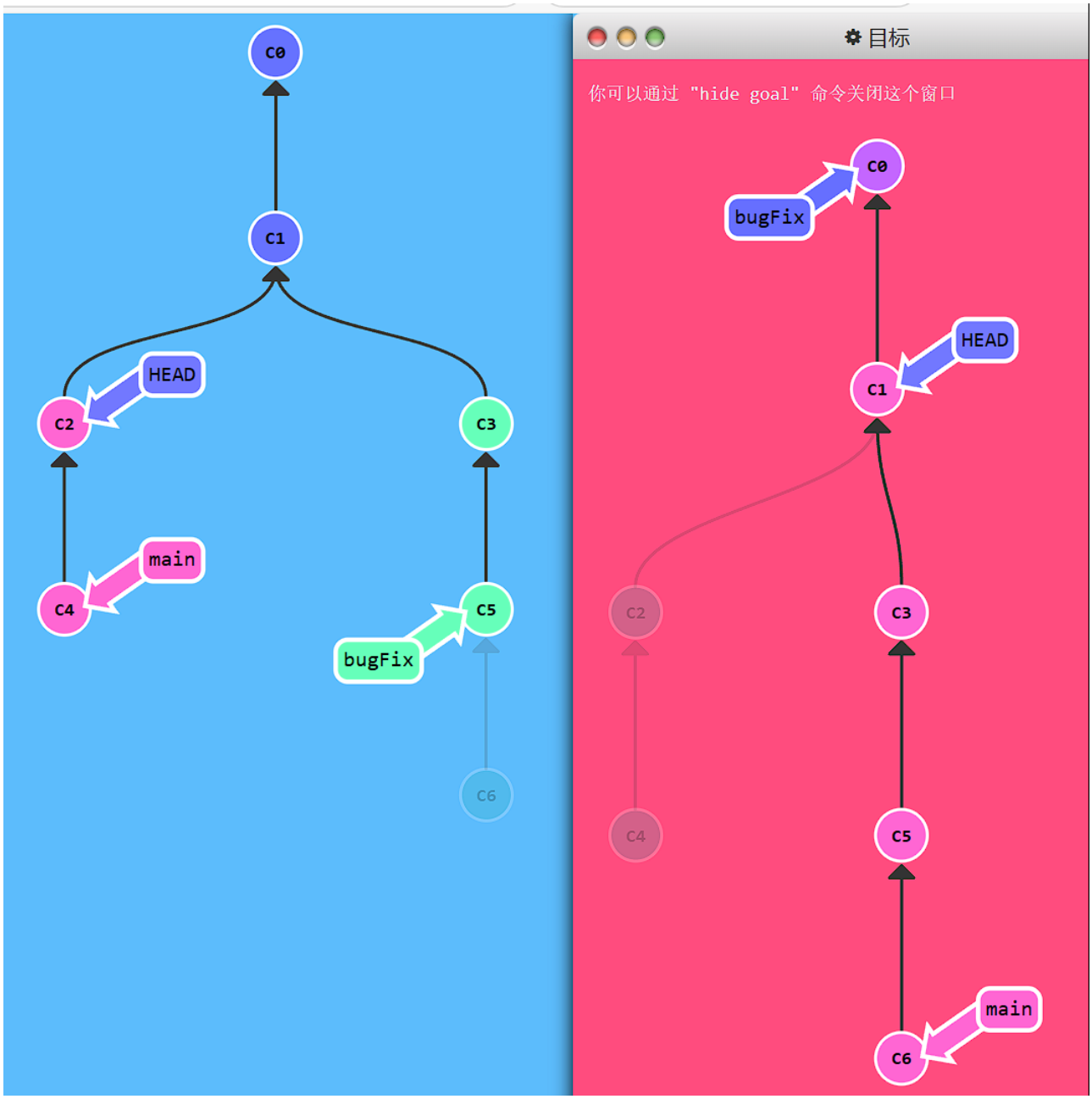
解释：

- `git checkout bugFix^`将`HEAD`分离指向分支`bugFix`的父节点。

3. 相对引用2(~)
如果想在提交树中向上移动很多步的话，敲那么多 ^ 貌似也挺烦人的，Git 当然也考虑到了这一点，于是又引入了操作符 ~。

该操作符后面可以跟一个数字（可选，不跟数字时与 ^ 相同，向上移动一次），指定向上移动多少次。相对引用最多的就是移动分支。可以直接使用 -f 选项让分支指向另一个提交。

任务目标：



Git命令：

```
git branch -f `main` c6
git branch -f bugFix c0
git checkout c1
```

解释：

- 第一个命令强制将 main 分支指向 c6 提交。
- 第二个命令强制将 bugFix 分支指向 c0 提交。
- 第三个命令将工作目录切换到 c1 提交。

4. 撤销变更

在 Git 中有很多方法可以撤销更改。就像提交代码一样，在 Git 中撤销更改既有低级操作部分（暂存单独的文件或代码块），也有高级操作部分（实际如何撤销更改）。我们的应用程序将专注于后者。

在 Git 中有两种主要的方法来撤销更改,一种是使用 `git reset`，另一种是使用 `git revert`。

`git reset` 通过将分支引用移动到时间线上的较早提交来撤销更改。从这个意义上说，你可以将它视为“重写历史”；它会将分支向后移动，就像那个提交从未发生过一样。

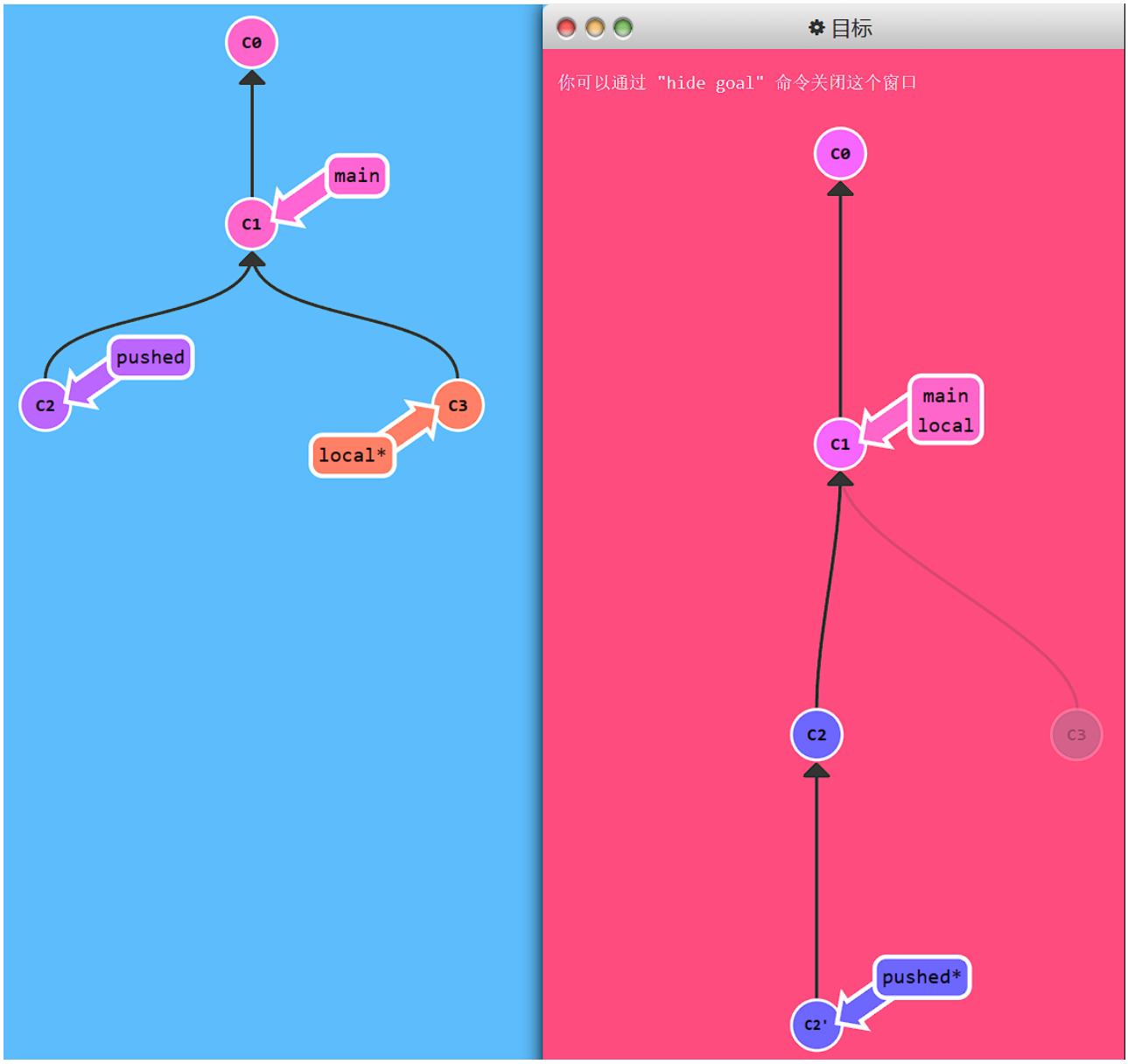
`git reset HEAD~1` 是最常用的撤销最近一次提交的方法。它通过将分支向后移动一个提交来工作。`--soft` 选项会向后移动，但会保持索引和工作目录（磁盘上的文件）不变。这样你可以重新提交这些更改，并且你有机会在重新提交之前重新排列它们。而 `--hard` 选项则会使你的工作目录与它移动到的提交状态相匹配。这是一种撤销提交的好方法，但它也可能导致未提交的工作丢失——使用时要小心！

虽然 `reset` 在本地分支上工作得很好，但其“重写历史”的方式不适用于他人正在使用的远程分支。

为了撤销更改并与其他人分享这些撤销的更改，我们需要使用 `git revert`。让我们看看它的实际操作。

`git revert HEAD` 将创建一个提交，该提交会反转最后一次提交引入的所有更改。这样你就可以将新的提交推送到服务器，你的同事也可以获取到它。

任务目标：



Git命令：

```
git reset HEAD~1
git checkout pushed
git revert HEAD
```

解释：

- git reset HEAD~1 用于将分支向后移动一个提交。
- git checkout pushed 用于切换到 pushed 分支。
- git revert HEAD 用于创建一个提交，该提交反转最后一次提交引入的所有内容。

自由修改提交树

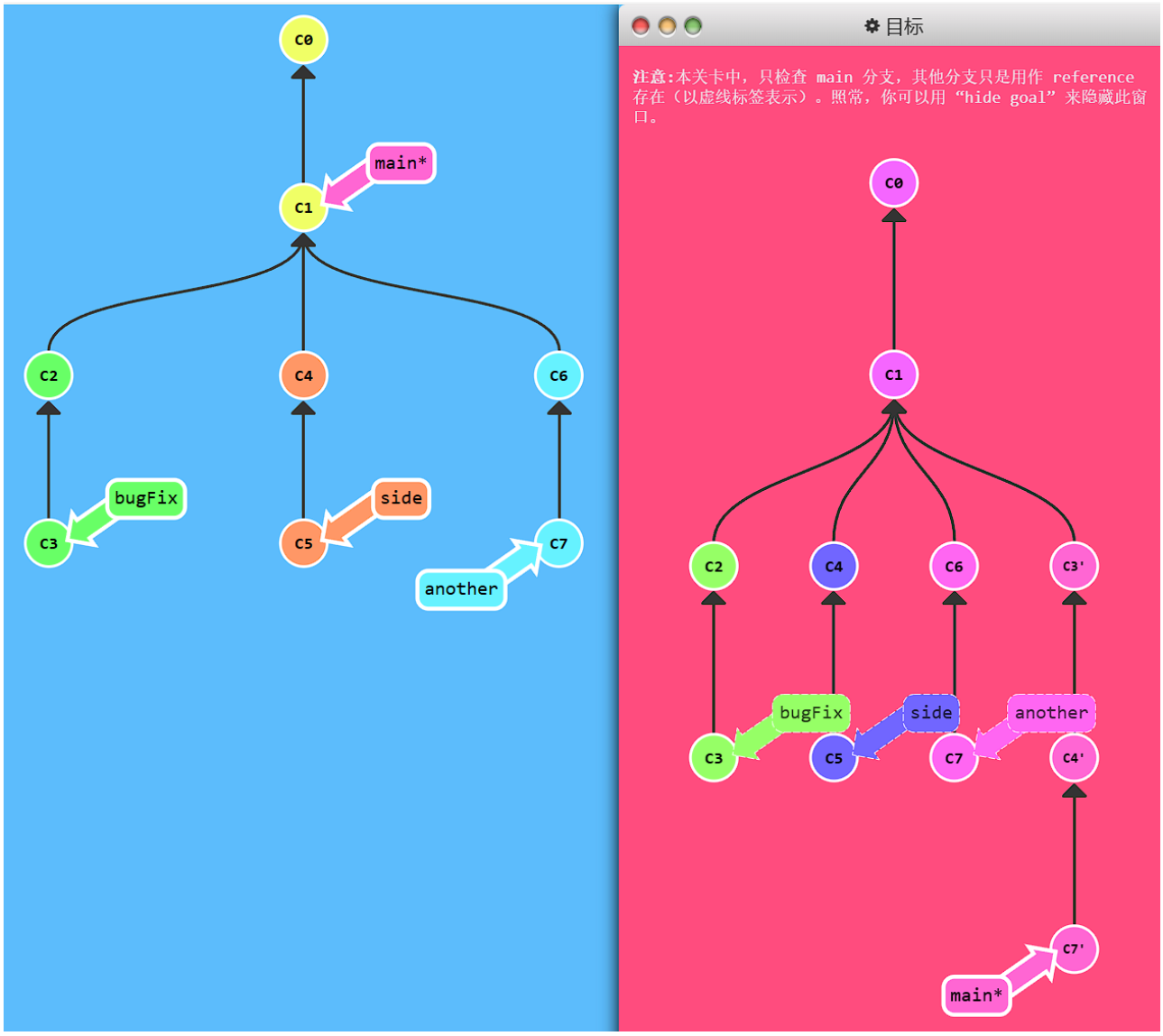
1. Git Cherry-pick
   “移动工作”——它是开发者用精确、优雅、灵活的方式来表达“我想把这份工作放在这里，那份工作放在那里”的方式。

   这一系列命令中的第一个是 `git cherry-pick`。它的形式如下：

   ```
   git cherry-pick <Commit1> <Commit2> <...>
   ```

   它是一种非常直接的方式，表示你希望将一系列提交复制到你当前位置（`HEAD`）之下。

   任务目标：



Git命令:

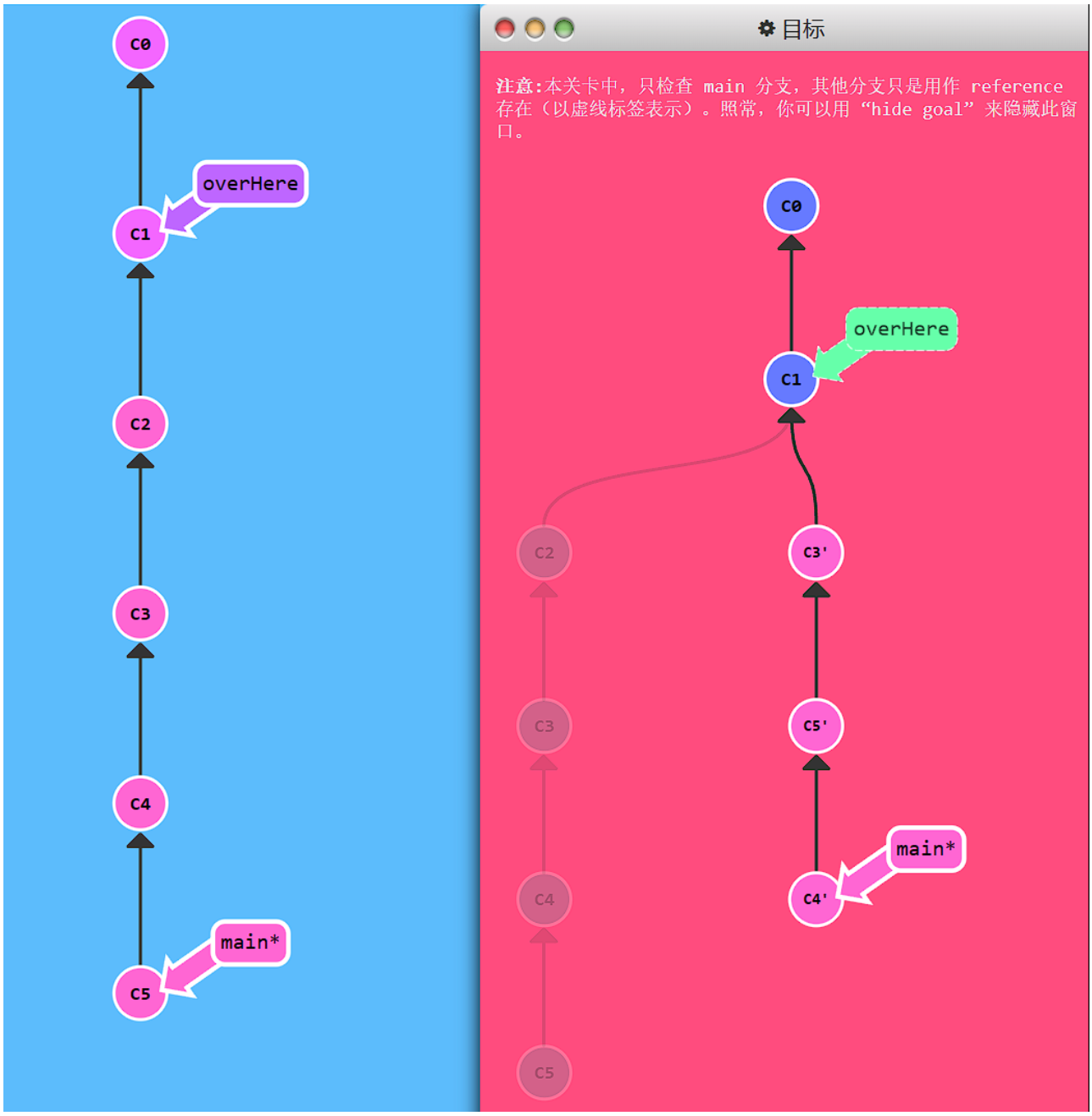
```
git cherry-pick c3 c4 c7
```

解释：

- 使用 `git cherry-pick c3 c4 c7` 来复制一系列提交，并将它们放在当前提交（HEAD）之下。

2. 交互式 rebase
如果不知道自己想要哪些提交呢？可以使用交互式 rebase 来处理这种情况——这是审查一系列即将变基的提交的最佳方法。
交互式 rebase 就是指 Git 使用带有 -i 选项的 rebase 命令。
如果加上这个选项，Git 将打开一个界面，显示哪些提交即将在变基目标之下被复制。它还会显示这些提交的哈希值和信息，这对于了解具体内容非常有帮助。

任务命令：



Git命令：

```
git rebase -i HEAD~4
```

解释：

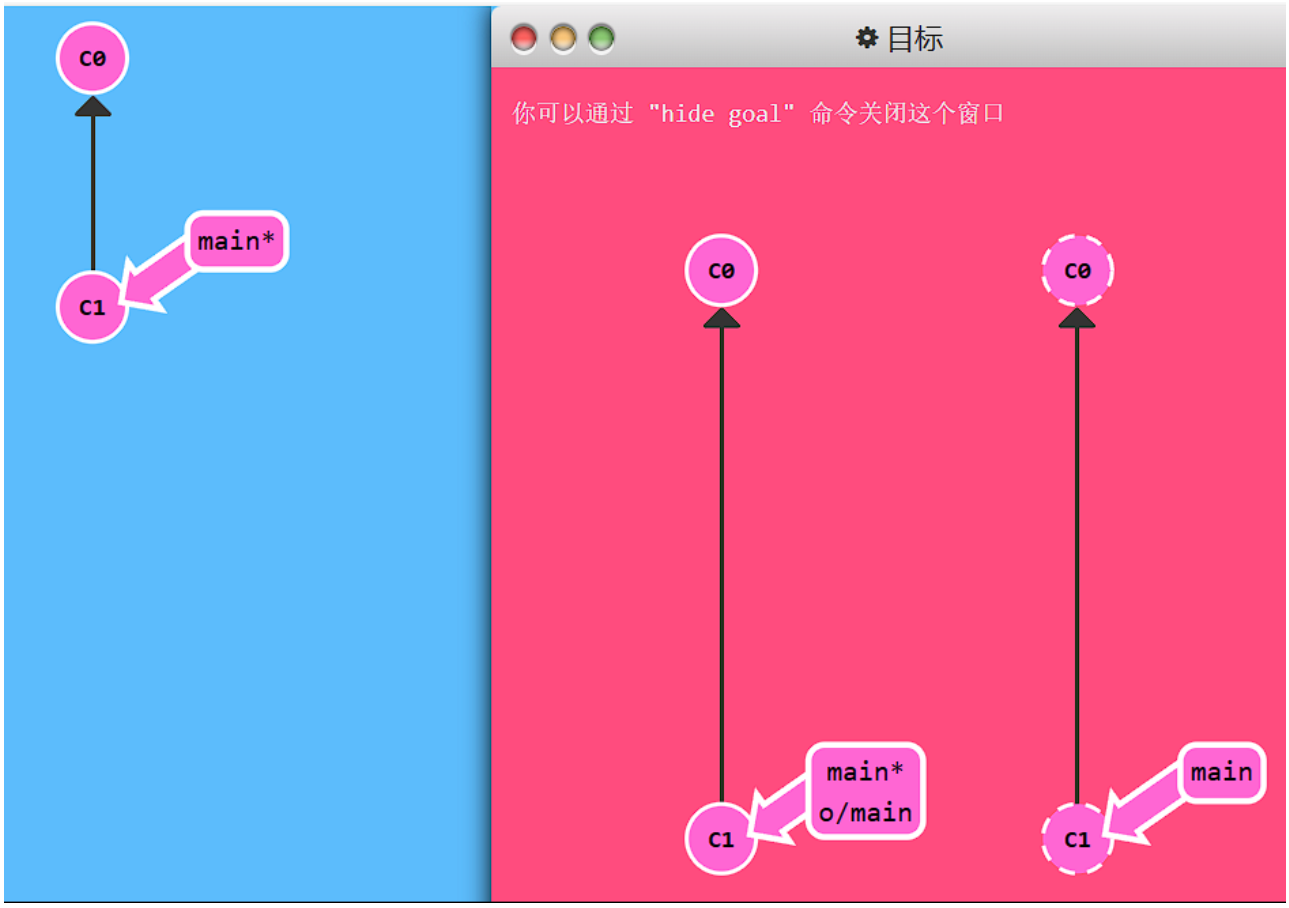
- `git rebase -i HEAD~4` 来打开一个界面，显示哪些提交即将在`rebase`目标之下被复制

Push & Pull —— Git 远程仓库

1. Git Clone
   远程仓库有很多很棒的特性：
   首先也是最重要的，远程仓库是一个很好的备份！本地 Git 仓库能够将文件恢复到以前的状态，但所有这些信息都是本地存储的。通过在其他计算机上拥有Git 仓库的副本，即使你丢失了所有本地数据，仍然可以继续之前的工作。

更重要的是，远程仓库让编程变得社交化！现在项目副本托管在其他地方，同伴可以很轻松地为项目做出贡献（或者拉取最新的更改）。
使用可视化远程仓库活动的网站（如 GitHub）已经变得非常流行，但远程仓库始终是这些工具的基础支撑。因此，理解它们非常重要！
现在要学习远程仓库操作，需要一个命令来为这些课程设置环境。这个命令就是 git clone。
git clone 是用来创建远程仓库的本地副本的命令（例如从 GitHub）。

任务目标：



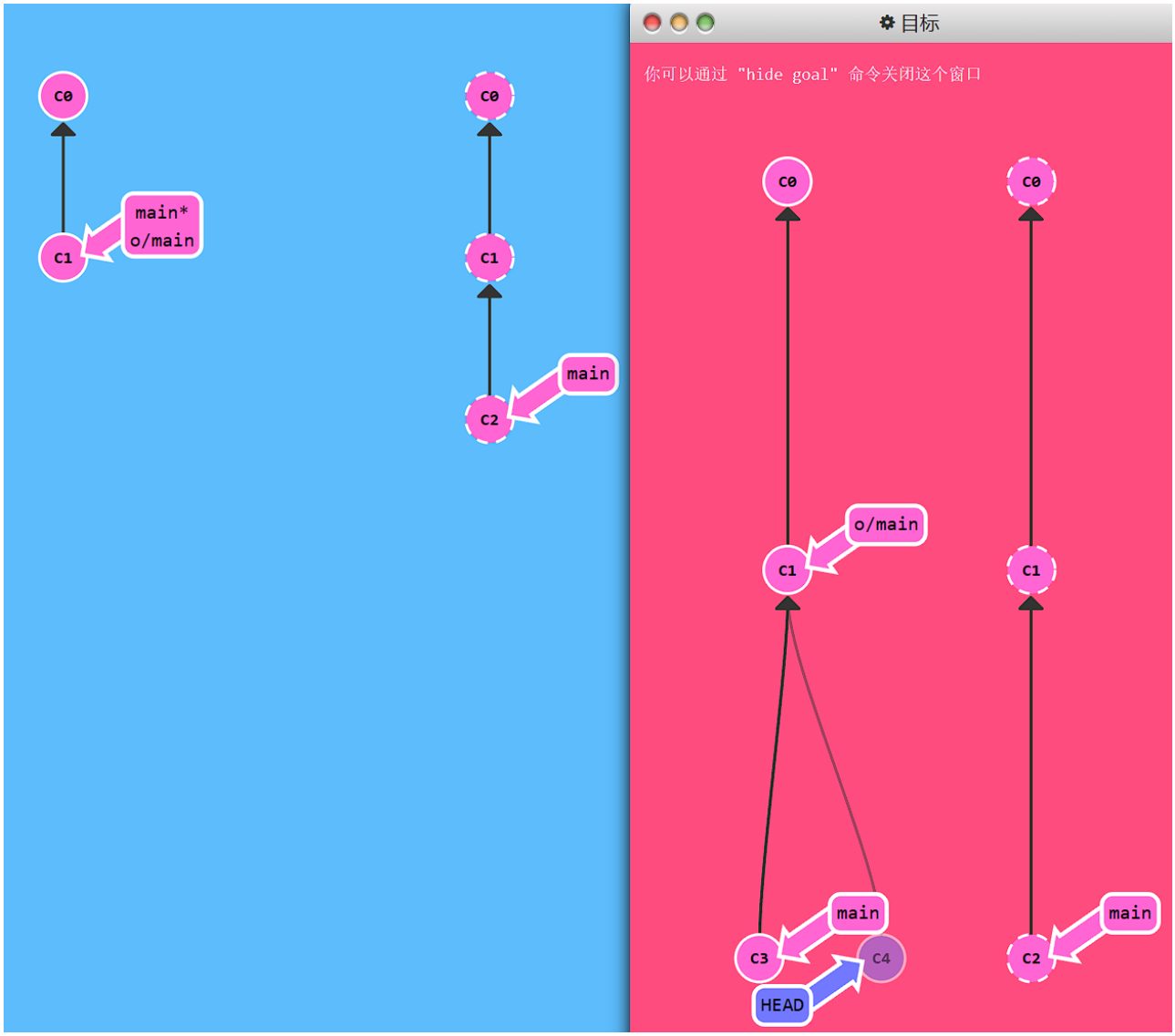
Git命令： git clone

解释：

- 使用 `git clone` 可以将你的本地仓库变成一个远程仓库。练习是与真实的Git相反的

2. 远程分支
在本地仓库中，o/main 是一个远程分支，它反映了远程仓库的状态。远程分支有特殊属性，比如检出时会进入分离的 HEAD 模式，因为不能直接在这些分支上工作。通常，远程的标准命名是 origin，但在某些界面中会简写为 o。这些分支帮助了解本地与远程之间的差异，确保在分享工作前，清楚地理解这些区别。

任务目标：



Git命令：

```
git commit
git checkout o/main
git commit
```

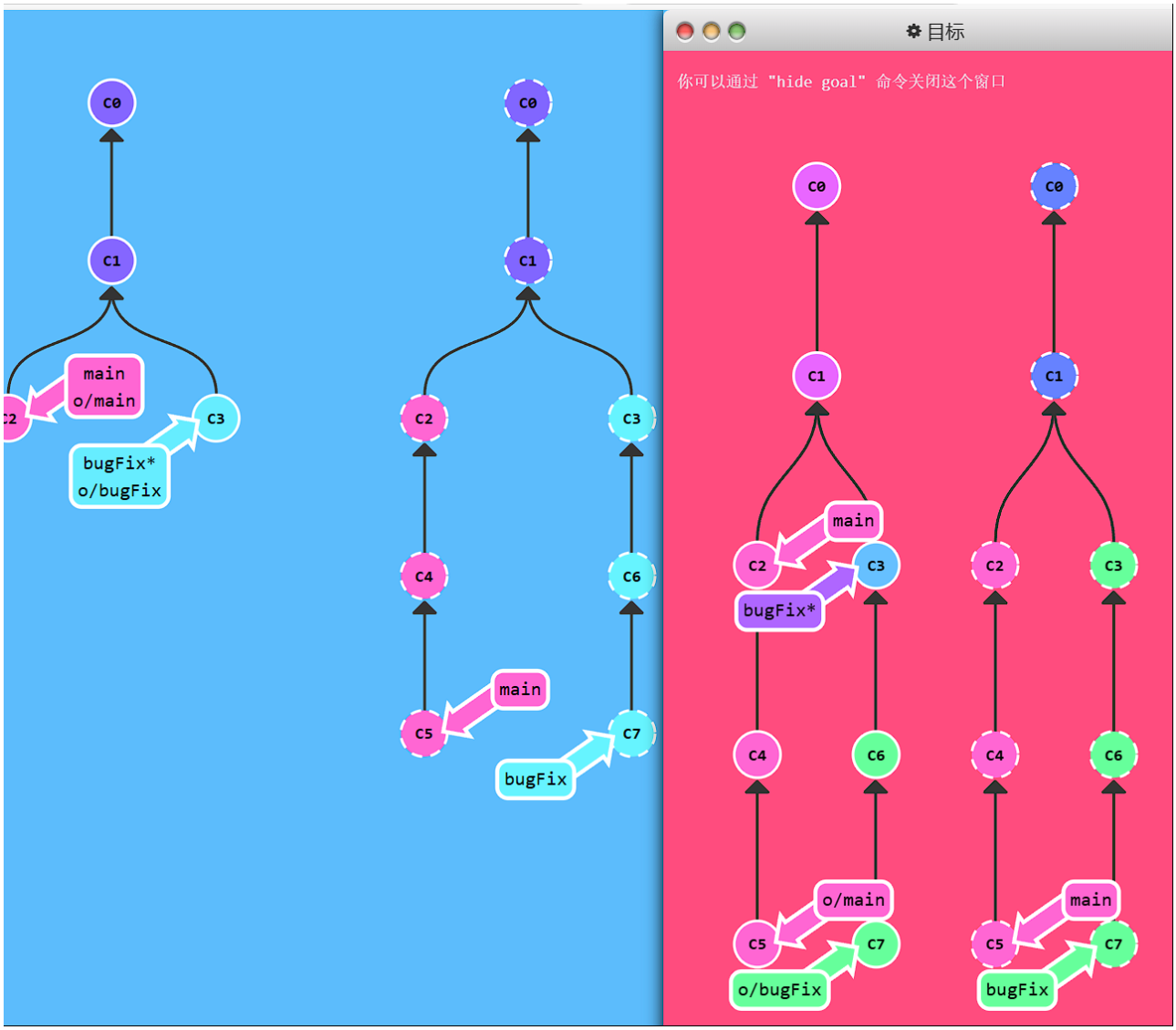
解释：

- 使用 `git commit` 在分支 `main` 上进行一次提交。
- 使用 `git checkout o/main` 切换到分支 `o/main`。
- 使用 `git commit` 尝试在分支上进行提交。注意：现在 `HEAD` 是分离状态，因为不能直接在远程分支上工作。

3. Git Fetch
使用 Git 远程仓库的本质在于在不同仓库之间传输数据，通过 git fetch 可以从远程仓库获取数据，但不改变本地状态。
git fetch 完成两件事：下载远程仓库中本地缺失的提交，更新本地对远程分支的引用。它与远程仓库通过互联网通信，但不会更新本地分支或文件系统。

许多开发者误以为 git fetch 会自动更新本地工作区，其实并不会，后续可学习其他命令来完成实际更新。

任务目标：



Git命令：

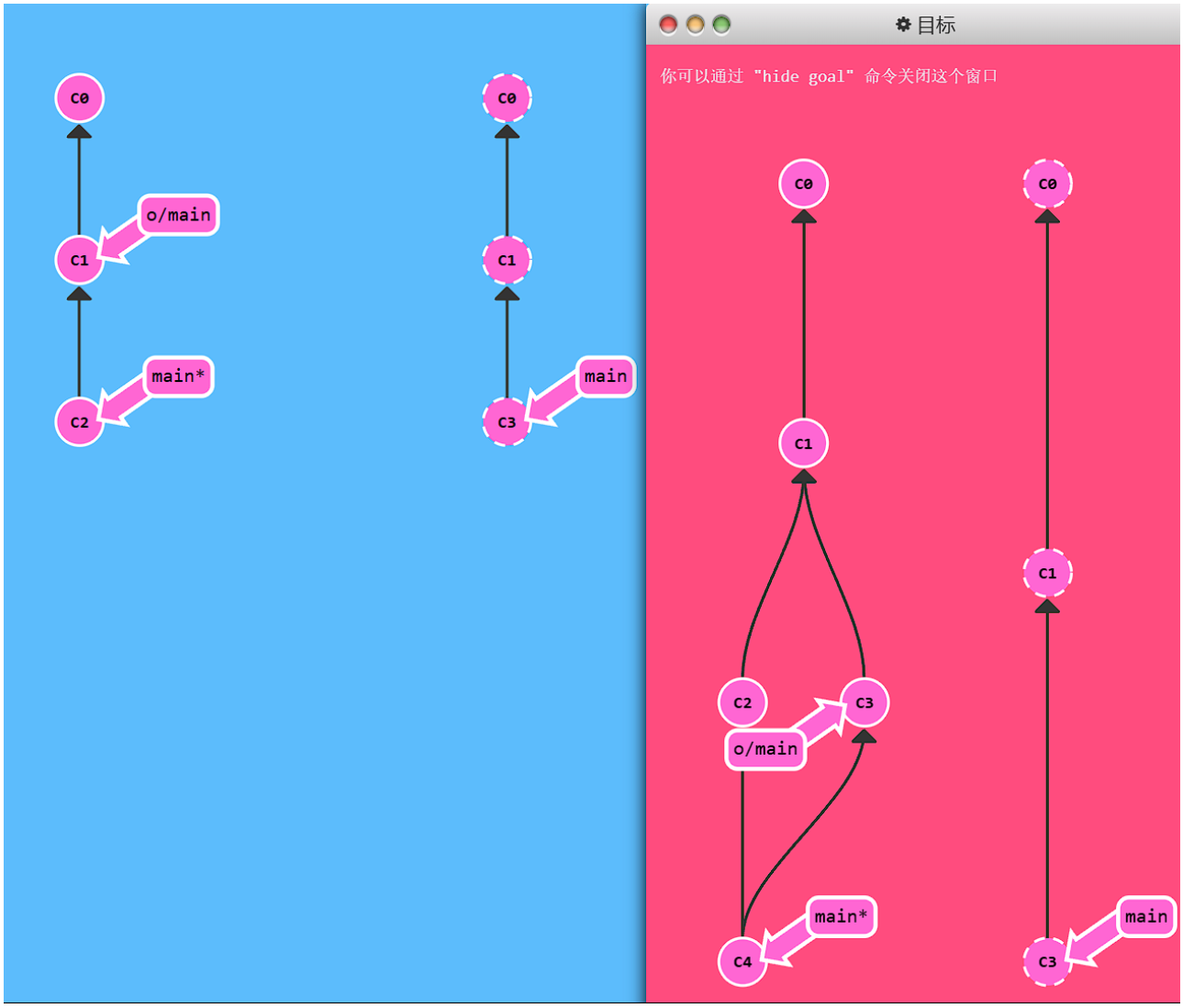
```
git fetch
```

解释：

- 使用 `git fetch` 下载远程仓库中本地缺失的提交，并更新本地对远程分支的指向位置。

4. Git Pull
在使用 `git fetch` 获取远程仓库的数据后，可以通过多种方式将这些新提交整合到本地分支，例如 `git cherry-pick`、`git rebase` 和 `git merge`。其中，`git pull` 是一个常用命令，它可以同时完成获取和合并两项操作。

任务目标：



Git命令：

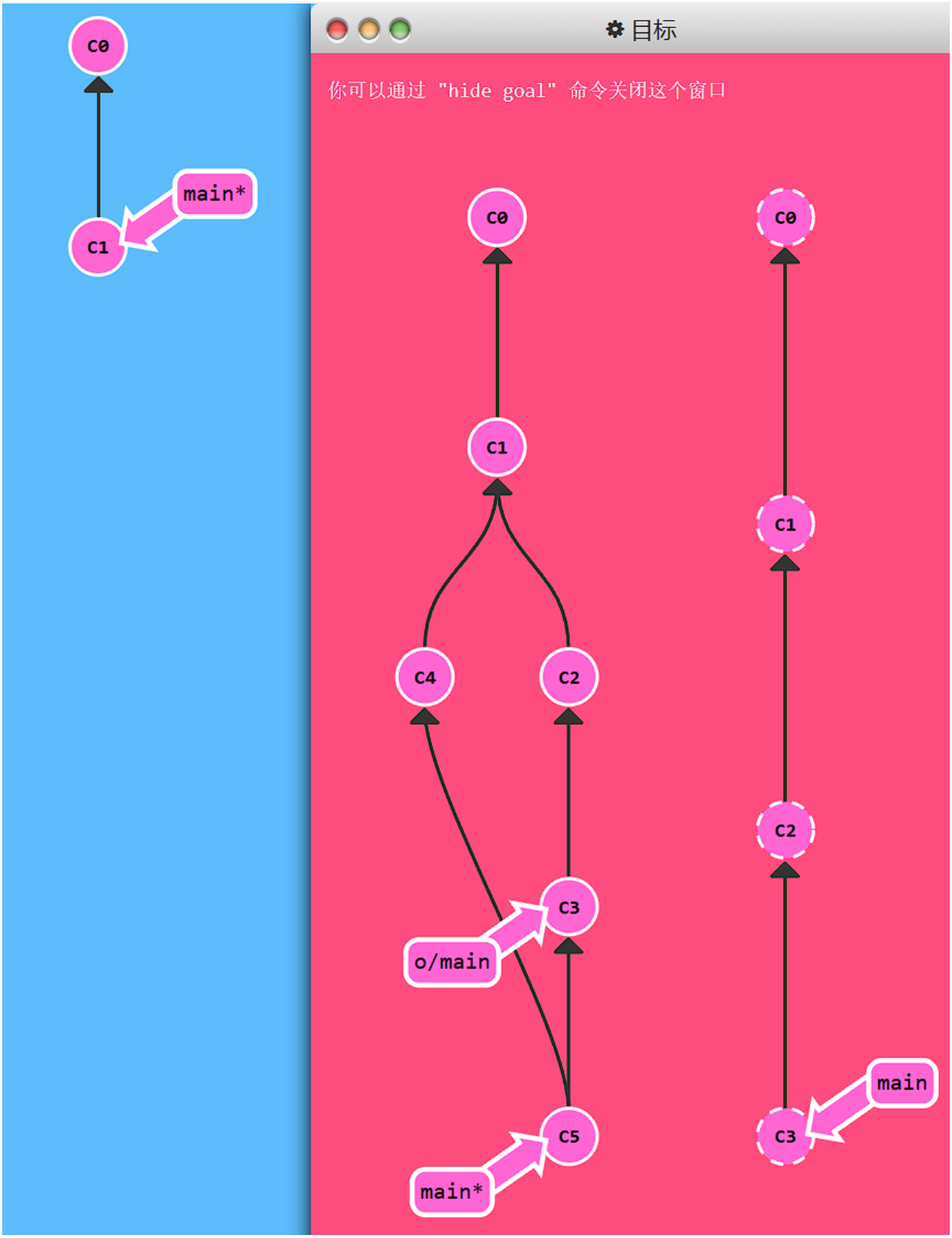
```
git pull
```

解释：

- `git pull`以获取远程更改，然后合并它们。

5. 模拟团队合作
学习如何拉取远程引入的更改。
这意味着我们基本上需要“假装”远程是由你的同事、朋友或合作者更新的，有时是在特定分支或某个特定数量的提交上。
为了做到这一点，我们引入了一个恰如其分命名的命令：`git fakeTeamwork`！

任务目标：



Git命令：

```
git clone
git fakeTeamwork `main` 2
git fetch
git commit
git merge o/main
```

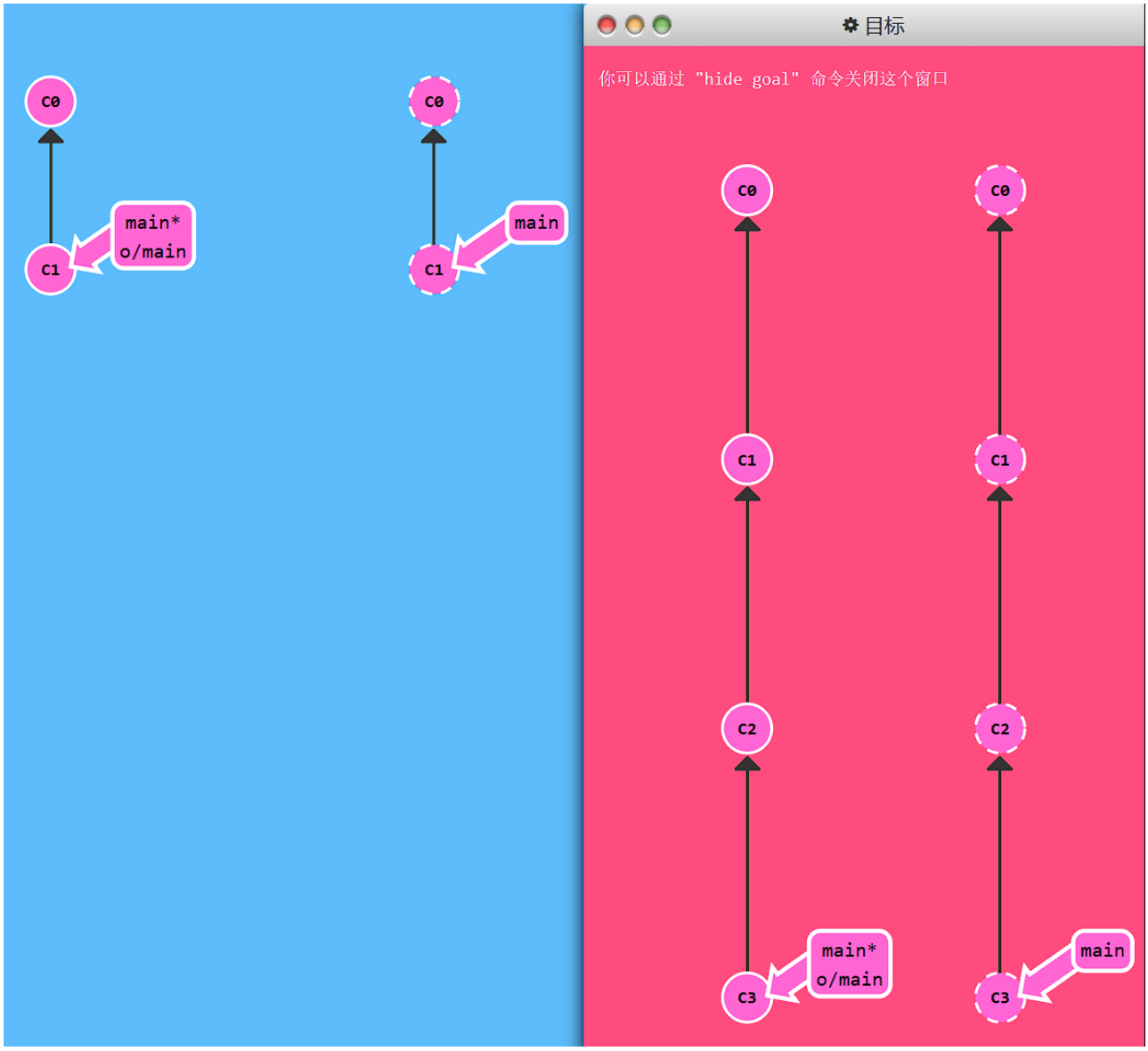
解释：

- 使用 `git clone` 将你的本地仓库变成一个远程仓库。
- 使用 `git fakeTeamwork main` 模拟远程仓库的 `main` 分支被你的某个合作者更新了两个提交。
- 使用 `git fetch` 下载远程仓库有但本地缺失的提交，并更新本地远程分支的指向。
- 使用 `git commit` 在 `main` 分支中创建一个提交。
- 使用 `git merge o/main` 合并 `main` 分支中的远程更改。

6. Git Push

上传共享工作的方法是使用 `git push`，这与下载共享工作的 `git pull` 相反。`git push` 用于将你的更改上传到指定的远程仓库，并允许他人从远程仓库下载你的工作。需要注意的是，`git push` 在没有参数时的行为取决于 Git 的设置 push.default，在我们的课程中会使用 upstream 作为默认设置。在自己的项目中推送之前，建议检查这个设置。

任务目标：



Git命令：

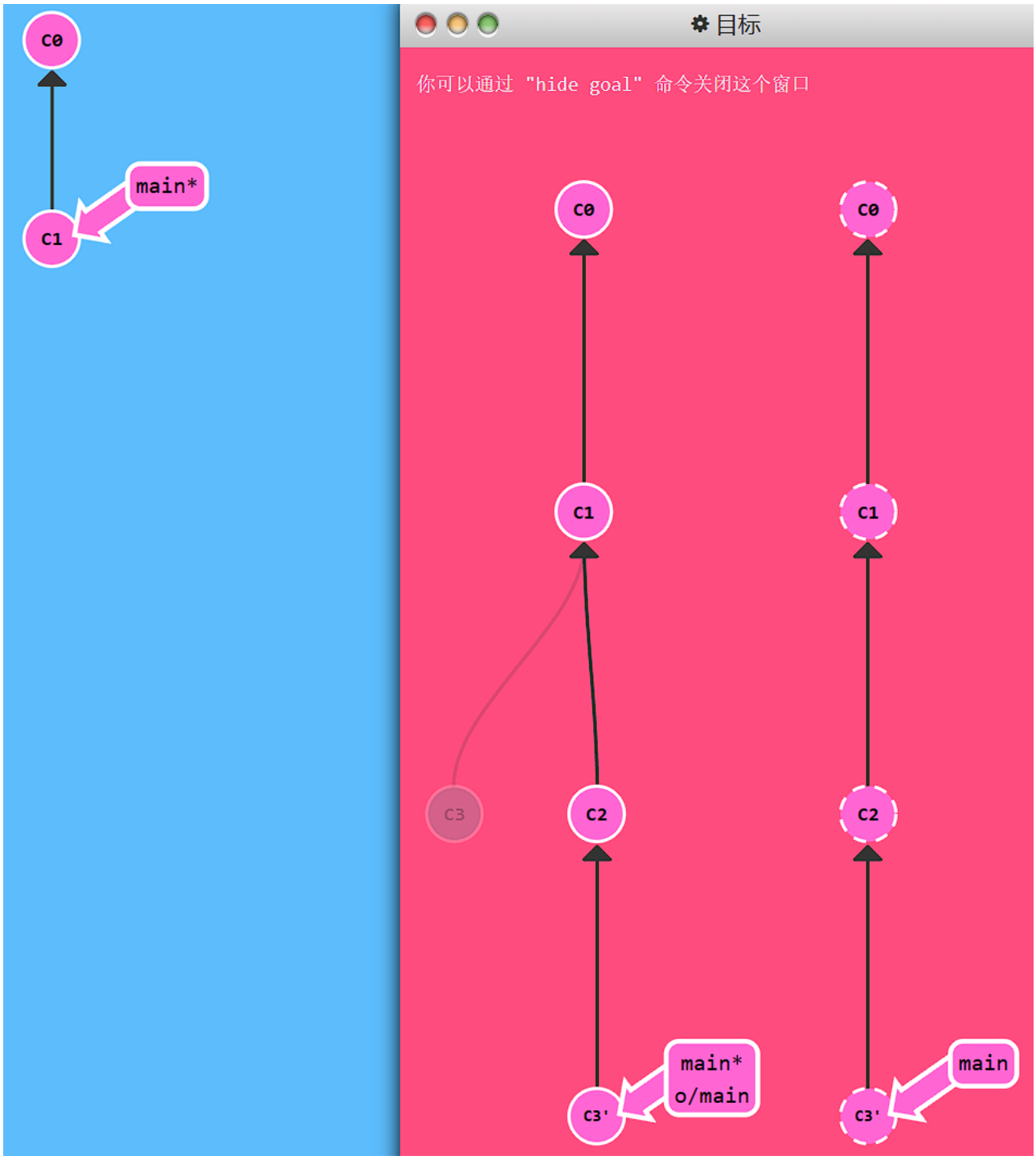
```
git commit
git commit
git push
```

解释：

- 在本地仓库提交两次，并将这些提交推送到远程仓库，同时更改为跟踪远程仓库的 `o/main`。

7. 偏离的提交历史
   学习了如何拉取和推送代码，但当仓库历史分歧时可能会导致困惑。假设在开发一个新功能，但同事们在这期间更新了很多代码，导致自己的版本过时。此时直接 `git push` 是不允许的，因为 Git 强制你在推送前更新本地状态以与远程同步。这可以通过 `git pull --rebase` 来实现，它是 `fetch` 和 `rebase` 的简写。在解决分歧时，常用的工作流程是：克隆仓库、模拟团队协作、进行自己的提交，然后通过 `rebase`发布工作。

   任务目标：



Git命令：

```
git clone
git fakeTeamwork `main` 1
git commit
git pull --rebase
git push
```

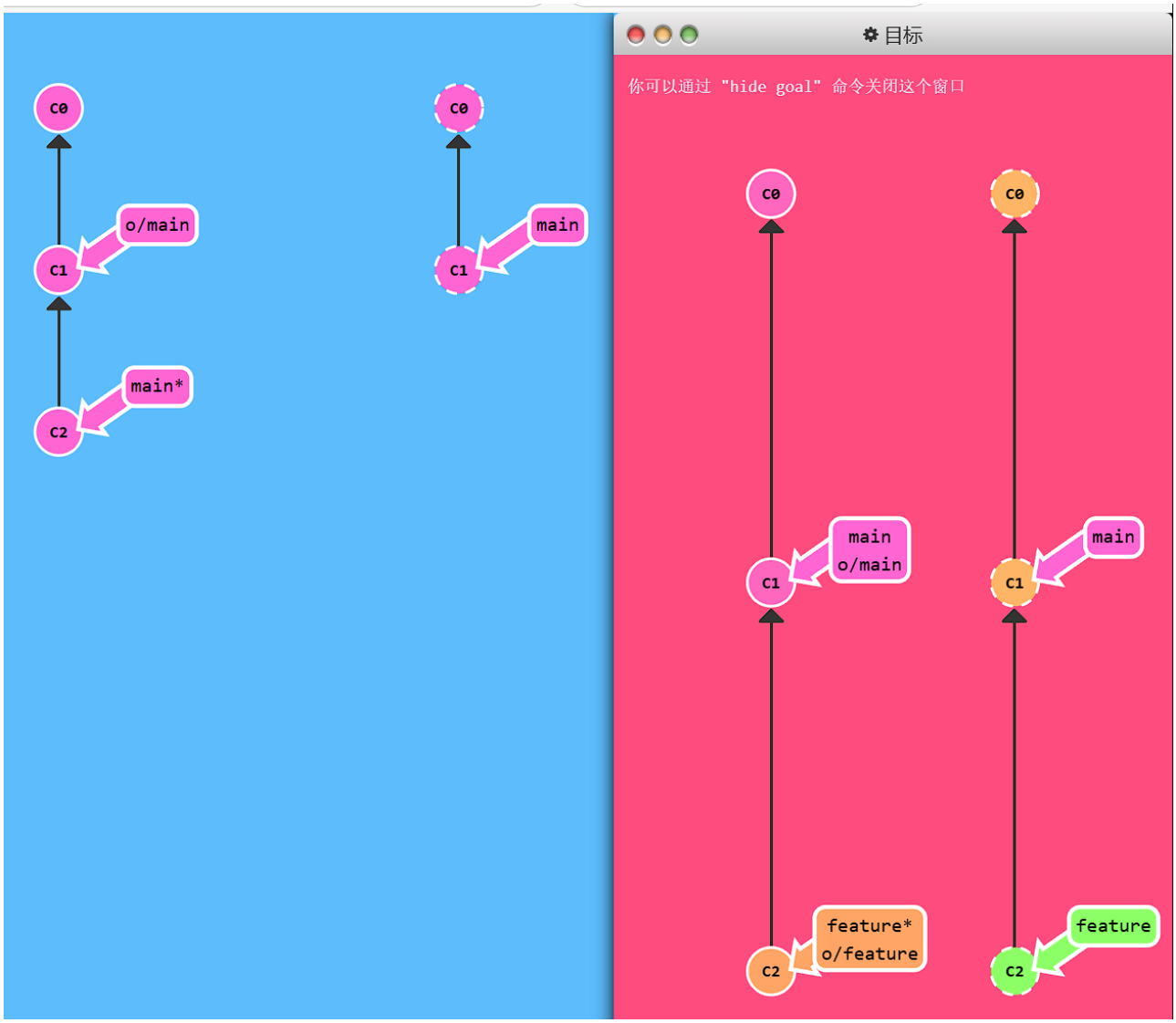
解释：

- 使用 `git clone` 把本地仓库变成一个远程仓库。

- 使用 `git fakeTeamwork main` 来模拟在 `main` 分支上，你的某个合作者更新了远程库（增加了一个提交）。
- 使用 `git commit` 在 `main` 分支上进行一次提交。
- 使用 `git pull --rebase` 获取远程的更改，然后将你的本地更改基于远程更改重新整理。
- 使用 `git push` 将本地仓库的提交推送到远程仓库。

8. 锁定的Main(Locked Main)
在大型协作团队中，`main` 分支通常需要通过拉取请求来合并更改，无法直接推送。如果不小心直接在 `main` 上提交，需要创建一个 `feature` 分支并推送，同时将 `main` 分支重置以与远程同步，以避免将来拉取时发生冲突。

任务目标：



Git命令：

```
git checkout -b feature
git branch -f `main` HEAD~
git push
```

解释：

- 使用 `git checkout -b feature` 创建一个新分支 `feature` 并切换到该分支。
- 使用 `git branch -f main HEAD~` 将 `main` 移动到 `HEAD` 的父节点。这一步很重要，因为需要重置你的 `main` 分支与远程同步，否则下次执行 `pull` 时，如果他人的提交与你的提交发生冲突，可能会出现问题。
- 使用 `git push` 将本地仓库中 `feature` 分支的提交推送到远程仓库。

实验总结

1. 了解学习并实际操作了如何安装和配置Git，学习了VSCode的基本使用方法以及安装方便实用的插件。

2. 在练习网站上学习了Git进行版本控制的基础知识和命令，包括初始化仓库、检查状态、提交更改、查看提交历史、合并分支、和使用rebase等操作。此外，还了解了如何使用.gitignore文件来忽略不需要跟踪的文件。

3. 学习了Markdown的基础，用于文档编辑，提升了文档编写的效率和可读性。

4. 遇到的问题与解决方案：

    - Markdown文件转PDF分享后插入的图片无法显示的问题。
        - 查阅资料，找到了PicGo + Gitee 搭建图床来插入网络图片url的方案，教程：https://zhuanlan.zhihu.com/p/361226602