源代码管理——以 Git 为例

王少东

二〇一二年十月

Part I

软件类项目的源代码管理

软件类项目

- 软件类项目的特点
- 新手惯用的工作方式
- 那种工作方式的弊端
- 更加有效的工作方式
- 这种工作方式的麻烦
- 获益
- 成本：学习某种源代码管理

软件类项目

- 软件类项目的特点
- 新手惯用的工作方式
- 那种工作方式的弊端
- 更加有效的工作方式
- 这种工作方式的麻烦
- 获益
- 成本：学习某种源代码管理

软件类项目

- 软件类项目的特点
- 新手惯用的工作方式
- 那种工作方式的弊端
- 更加有效的工作方式
- 这种工作方式的麻烦
- 获益
- 成本：学习某种源代码管理

软件类项目

- 软件类项目的特点
- 新手惯用的工作方式
- 那种工作方式的弊端
- 更加有效的工作方式
- 这种工作方式的麻烦
- 获益
- 成本：学习某种源代码管理

软件类项目

- 软件类项目的特点
- 新手惯用的工作方式
- 那种工作方式的弊端
- 更加有效的工作方式
- 这种工作方式的麻烦
- 获益
- 成本：学习某种源代码管理

软件类项目

- 软件类项目的特点
- 新手惯用的工作方式
- 那种工作方式的弊端
- 更加有效的工作方式
- 这种工作方式的麻烦
- 获益
- 成本：学习某种源代码管理

软件类项目

- 软件类项目的特点
- 新手惯用的工作方式
- 那种工作方式的弊端
- 更加有效的工作方式
- 这种工作方式的麻烦
- 获益
- 成本：学习某种源代码管理

源代码管理工具

- 源代码管理工具是什么?
- 我们为什么需要它?
- 不使用源代码管理会怎样?
- 常见的源代码管理工具：CVS，Subversion（SVN），Git

源代码管理工具

- 源代码管理工具是什么?
- 我们为什么需要它?
- 不使用源代码管理会怎样?
- 常见的源代码管理工具：CVS，Subversion（SVN），Git

源代码管理工具

- 源代码管理工具是什么?
- 我们为什么需要它?
- 不使用源代码管理会怎样?
- 常见的源代码管理工具：CVS，Subversion（SVN），Git

源代码管理工具

- 源代码管理工具是什么?
- 我们为什么需要它?
- 不使用源代码管理会怎样?
- 常见的源代码管理工具：CVS，Subversion（SVN），Git

以 Git 为例

- Git 是一种完全分布式的源代码管理工具
- Git 的使用非常适合多个线索的并行开发
- Git 仓库存在于每个工作目录当中，天然的备份
- Git 是自由软件，拥有广泛用户基础

以 Git 为例

- Git 是一种完全分布式的源代码管理工具
- Git 的使用非常适合多个线索的并行开发
- Git 仓库存在于每个工作目录当中，天然的备份
- Git 是自由软件，拥有广泛用户基础

以 Git 为例

- Git 是一种完全分布式的源代码管理工具
- Git 的使用非常适合多个线索的并行开发
- Git 仓库存在于每个工作目录当中，天然的备份
- Git 是自由软件，拥有广泛用户基础

以 Git 为例

- Git 是一种完全分布式的源代码管理工具
- Git 的使用非常适合多个线索的并行开发
- Git 仓库存在于每个工作目录当中，天然的备份
- Git 是自由软件，拥有广泛用户基础

使用 Git 所要涉及的几个基本词汇

- 仓库（repository）
- 工作目录（working directory）
- 提交（commit）
- 检出（checkout）
- 分支（branch）
- 合并（merge）
- 日志（log）

使用 Git 所要涉及的几个基本词汇

- 仓库（repository）
- 工作目录（working directory）
- 提交（commit）
- 检出（checkout）
- 分支（branch）
- 合并（merge）
- 日志（log）

使用 Git 所要涉及的几个基本词汇

- 仓库（repository）
- 工作目录（working directory）
- 提交（commit）
- 检出（checkout）
- 分支（branch）
- 合并（merge）
- 日志（log）

使用 Git 所要涉及的几个基本词汇

- 仓库（repository）
- 工作目录（working directory）
- 提交（commit）
- 检出（checkout）
- 分支（branch）
- 合并（merge）
- 日志（log）

使用 Git 所要涉及的几个基本词汇

- 仓库（repository）
- 工作目录（working directory）
- 提交（commit）
- 检出（checkout）
- 分支（branch）
- 合并（merge）
- 日志（log）

使用 Git 所要涉及的几个基本词汇

- 仓库（repository）
- 工作目录（working directory）
- 提交（commit）
- 检出（checkout）
- 分支（branch）
- 合并（merge）
- 日志（log）

使用 Git 所要涉及的几个基本词汇

- 仓库（repository）
- 工作目录（working directory）
- 提交（commit）
- 检出（checkout）
- 分支（branch）
- 合并（merge）
- 日志（log）

版本管理的工作情境

- 创建仓库
- 提交工作
- 推到远程
- 拖到本地
- 合并
- 提交
- 切换分支

版本管理的工作情境

- 创建仓库
- 提交工作
- 推到远程
- 拖到本地
- 合并
- 提交
- 切换分支

版本管理的工作情境

- 创建仓库
- 提交工作
- 推到远程
- 拖到本地
- 合并
- 提交
- 切换分支

版本管理的工作情境

- 创建仓库
- 提交工作
- 推到远程
- 拖到本地
- 合并
- 提交
- 切换分支

版本管理的工作情境

- 创建仓库
- 提交工作
- 推到远程
- 拖到本地
- 合并
- 提交
- 切换分支

版本管理的工作情境

- 创建仓库
- 提交工作
- 推到远程
- 拖到本地
- 合并
- 提交
- 切换分支

版本管理的工作情境

- 创建仓库
- 提交工作
- 推到远程
- 拖到本地
- 合并
- 提交
- 切换分支

Part II

Git 入门

检查 Git 是否已安装

- Linux 系统中，使用命令：

```
git --version
```

- Windows 系统下，可以看到如下图标：

- Mac 系统下（?）

检查 Git 是否已安装

- Linux 系统中，使用命令：

```
git --version
```

- Windows 系统下，可以看到如下图标：

- Mac 系统下（?）

检查 Git 是否已安装

- Linux 系统中，使用命令：

```
git --version
```

- Windows 系统下，可以看到如下图标：

- Mac 系统下（?）

安装 Git

- Linux 下的安装不必废话

```
sudo apt-get install git-core
```

- Windows 下载并安装 Git for Windows 或者 msysGit
- Mac 下的可选用 Git for OS X，或者 GitX

安装 Git

- Linux 下的安装不必废话

```
sudo apt-get install git-core
```

- Windows 下载并安装 Git for Windows 或者 msysGit
- Mac 下的可选用 Git for OS X，或者 GitX

安装 Git

- Linux 下的安装不必废话

```
sudo apt-get install git-core
```

- Windows 下载并安装 Git for Windows 或者 msysGit
- Mac 下的可选用 Git for OS X，或者 GitX

配置

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global core.editor vim
git config --global core.quotepash false
```

用起来

新建一个项目

```
mkdir xprj
cd xprj
vi main.c
```

为这个项目建立一个 Git 仓库并提交第一个版本

```
git init
git add .
git commit -m 'The first of first commit.'
```

用起来

新建一个项目

```
mkdir xprj
cd xprj
vi main.c
```

为这个项目建立一个 Git 仓库并提交第一个版本

```
git init
git add .
git commit -m 'The first of first commit.'
```

概念补充

- 仓库在哪里?
  本地仓库的概念
- “加入”的含义
  Staging Area, Index, Cache
  add, update, stage
- 查看 Git 状态

```
git status
```

概念补充

- 仓库在哪里?
  本地仓库的概念
- “加入”的含义
  Staging Area, Index, Cache
  add, update, stage
- 查看 Git 状态

```
git status
```

概念补充

- 仓库在哪里?
  本地仓库的概念
- “加入”的含义
  Staging Area, Index, Cache
  add, update, stage
- 查看 Git 状态

```
git status
```

修改与提交

修改一些内容

```
vi main.c
```

查看状态，提交修改

```
git status
git add main.c
git status
git commit
```

合并操作步骤

```
git commit -a -m 'Append a newline at the end of string'
```

修改与提交

修改一些内容

```
vi main.c
```

查看状态，提交修改

```
git status
git add main.c
git status
git commit
```

合并操作步骤

```
git commit -a -m 'Append a newline at the end of string'
```

修改与提交

修改一些内容

```
vi main.c
```

查看状态，提交修改

```
git status
git add main.c
git status
git commit
```

合并操作步骤

```
git commit -a -m 'Append a newline at the end of string'
```

增加文件

增加两个文件

```
vi common.c
vi common.h
```

查看状态，注意 Untracked files 提示

```
git status
```

加到 Git 仓库中

```
git add common.c common.h
git status
git commit -m 'Two files common.[ch] are added.'
git status
```

增加文件

增加两个文件

```
vi common.c
vi common.h
```

查看状态，注意 Untracked files 提示

```
git status
```

加到 Git 仓库中

```
git add common.c common.h
git status
git commit -m 'Two files common.[ch] are added.'
git status
```

增加文件

增加两个文件

```
vi common.c
vi common.h
```

查看状态，注意 Untracked files 提示

```
git status
```

加到 Git 仓库中

```
git add common.c common.h
git status
git commit -m 'Two files common.[ch] are added.'
git status
```

纳入/未纳入代码管理的文件

- git add <file> 纳入 Git 的控制之下
- git rm <file> 从 Git 管理中删除文件
- 特殊文件 .gitignore 专门设定无需 Git 管理的文件

纳入/未纳入代码管理的文件

- git add <file> 纳入 Git 的控制之下
- git rm <file> 从 Git 管理中删除文件
- 特殊文件 .gitignore 专门设定无需 Git 管理的文件

纳入/未纳入代码管理的文件

- git add <file> 纳入 Git 的控制之下
- git rm <file> 从 Git 管理中删除文件
- 特殊文件 .gitignore 专门设定无需 Git 管理的文件

选择性提交

继续修改文件

```
vi main.c
vi common.c
```

忘记了修改过哪些文件

```
git status
git diff
```

只想提交针对个别文件的修改

```
git add main.c
git commit
```

所有修改过的文件，一律提交

```
git commit -am 'fix the #755 bug'
```

选择性提交

继续修改文件

```
vi main.c
vi common.c
```

忘记了修改过哪些文件

```
git status
git diff
```

只想提交针对个别文件的修改

```
git add main.c
git commit
```

所有修改过的文件，一律提交

```
git commit -am 'fix the #755 bug'
```

选择性提交

继续修改文件

```
vi main.c
vi common.c
```

忘记了修改过哪些文件

```
git status
git diff
```

只想提交针对个别文件的修改

```
git add main.c
git commit
```

所有修改过的文件，一律提交

```
git commit -am 'fix the #755 bug'
```

选择性提交

继续修改文件

```
vi main.c
vi common.c
```

忘记了修改过哪些文件

```
git status
git diff
```

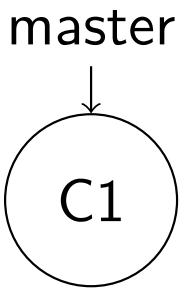
只想提交针对个别文件的修改

```
git add main.c
git commit
```

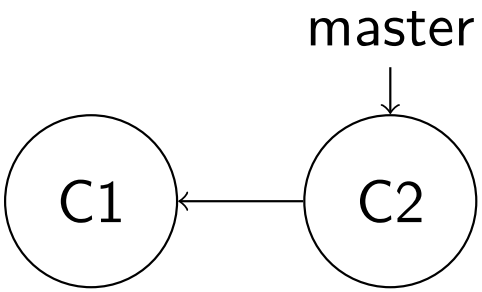
所有修改过的文件，一律提交

```
git commit -am 'fix the #755 bug'
```

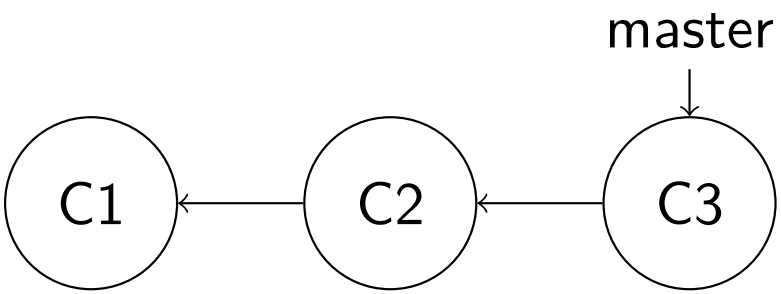
Git 仓库中包含每一次提交所形成的历史



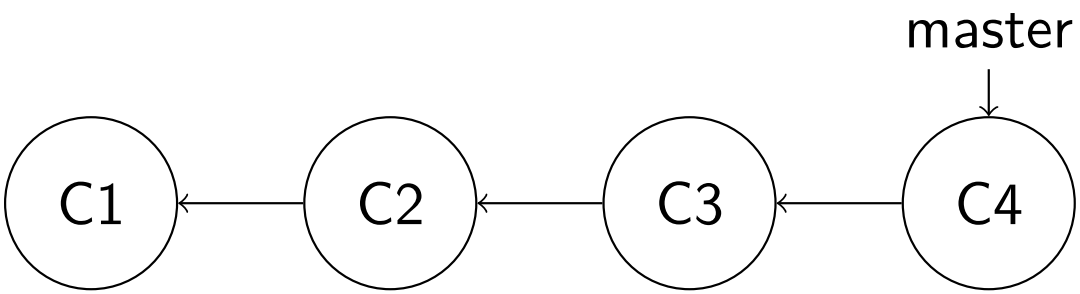
Git 仓库中包含每一次提交所形成的历史



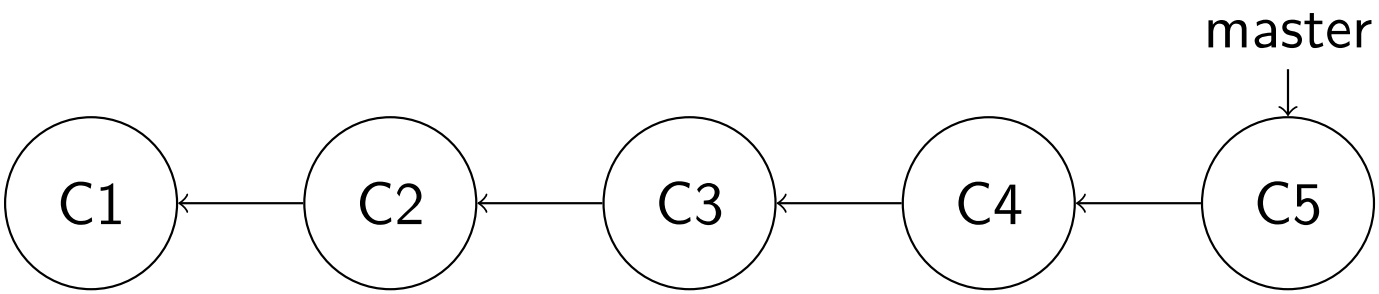
Git 仓库中包含每一次提交所形成的历史



Git 仓库中包含每一次提交所形成的历史



Git 仓库中包含每一次提交所形成的历史



观察提交历史

- 观察 Git 提交历史

```
git log
```

- 提交的名字
  - 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
  - 绝对提交名的简写：533e31
  - 参照名：master, dev, fixbug, v1.0
  - 符号参照名：HEAD
  - 相对提交名：master^

观察提交历史

- 观察 Git 提交历史

```
git log
```

- 提交的名字
  - 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
  - 绝对提交名的简写：533e31
  - 参照名：master, dev, fixbug, v1.0
  - 符号参照名：HEAD
  - 相对提交名：master^

观察提交历史

- 观察 Git 提交历史

```
git log
```

- 提交的名字
  - 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
  - 绝对提交名的简写：533e31
  - 参照名：master, dev, fixbug, v1.0
  - 符号参照名：HEAD
  - 相对提交名：master^

观察提交历史

- 观察 Git 提交历史

```
git log
```

- 提交的名字
  - 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
  - 绝对提交名的简写：533e31
  - 参照名：master, dev, fixbug, v1.0
  - 符号参照名：HEAD
  - 相对提交名：master^

观察提交历史

- 观察 Git 提交历史

```
git log
```

- 提交的名字
  - 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
  - 绝对提交名的简写：533e31
  - 参照名：master, dev, fixbug, v1.0
  - 符号参照名：`HEAD`
  - 相对提交名：`master^`

观察提交历史

- 观察 Git 提交历史

```
git log
```

- 提交的名字
  - 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
  - 绝对提交名的简写：533e31
  - 参照名：master, dev, fixbug, v1.0
  - 符号参照名：HEAD
  - 相对提交名：master^

观察提交历史

- 观察 Git 提交历史

```
git log
```

- 提交的名字
  - 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
  - 绝对提交名的简写：533e31
  - 参照名：master, dev, fixbug, v1.0
  - 符号参照名：`HEAD`
  - 相对提交名：`master^`

小结

使用 Git，线性进展，使用过程非常简单：

- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- ...

```
git add <file>
git commit
```

```
git commit -a
```

```
git commit -am 'The comment for this revision'
```

小结

使用 Git，线性进展，使用过程非常简单：

- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- ...

```
git add <file>
git commit
```

```
git commit -a
```

```
git commit -am 'The comment for this revision'
```

小结

使用 Git，线性进展，使用过程非常简单：

- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- ...

```
git add <file>
git commit
```

```
git commit -a
```

```
git commit -am 'The comment for this revision'
```

小结

使用 Git，线性进展，使用过程非常简单：

- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- ...

```
git add <file>
git commit
```

```
git commit -a
```

```
git commit -am 'The comment for this revision'
```

小结

使用 Git，线性进展，使用过程非常简单：

- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- 修改代码 ... 提交修改 ...
- ...

```
git add <file>
git commit
```

```
git commit -a
```

```
git commit -am 'The comment for this revision'
```

小结（续）

其他的辅助命令

```
git init
```

```
git status
```

```
git log
```

获得在线帮助的方法：

```
git help
git help commit
git commit --help
```

小结（续）

其他的辅助命令

```
git init
```

```
git status
```

```
git log
```

获得在线帮助的方法：

```
git help
git help commit
git commit --help
```

小结（续）

其他的辅助命令

```
git init
```

```
git status
```

```
git log
```

获得在线帮助的方法：

```
git help
git help commit
git commit --help
```

小结（续）

其他的辅助命令

```
git init
```

```
git status
```

```
git log
```

获得在线帮助的方法：

```
git help
git help commit
git commit --help
```

小结（续）

其他的辅助命令

```
git init
```

```
git status
```

```
git log
```

获得在线帮助的方法：

```
git help
git help commit
git commit --help
```

Part III

分支与合并

新建分支

项目发布后，进入下一阶段开发，不想影响已经发布的版本

```
git branch dev
git checkout dev
```

已经另建了一个分支，并且转移到新的分支上。

```
git branch
```

修改源程序，并把修改过的文件提交到仓库

```
vi main.c
git add main.c
git commit -m 'Some function is added.'
```

这是在新的分支上的一个提交，不影响原来的分支。

新建分支

项目发布后，进入下一阶段开发，不想影响已经发布的版本

```
git branch dev
git checkout dev
```

已经另建了一个分支，并且转移到新的分支上。

```
git branch
```

修改源程序，并把修改过的文件提交到仓库

```
vi main.c
git add main.c
git commit -m 'Some function is added.'
```

这是在新的分支上的一个提交，不影响原来的分支。

新建分支

项目发布后，进入下一阶段开发，不想影响已经发布的版本

```
git branch dev
git checkout dev
```

已经另建了一个分支，并且转移到新的分支上。

```
git branch
```

修改源程序，并把修改过的文件提交到仓库

```
vi main.c
git add main.c
git commit -m 'Some function is added.'
```

这是在新的分支上的一个提交，不影响原来的分支。

新建分支

项目发布后，进入下一阶段开发，不想影响已经发布的版本

```
git branch dev
git checkout dev
```

已经另建了一个分支，并且转移到新的分支上。

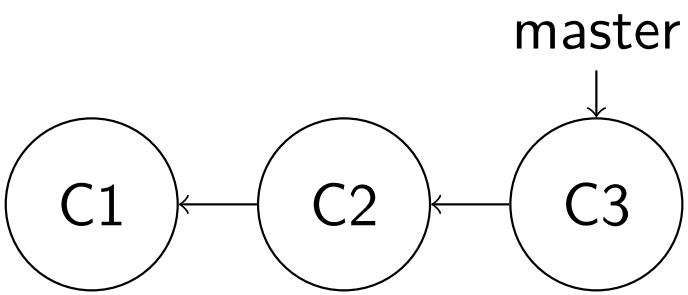
```
git branch
```

修改源程序，并把修改过的文件提交到仓库

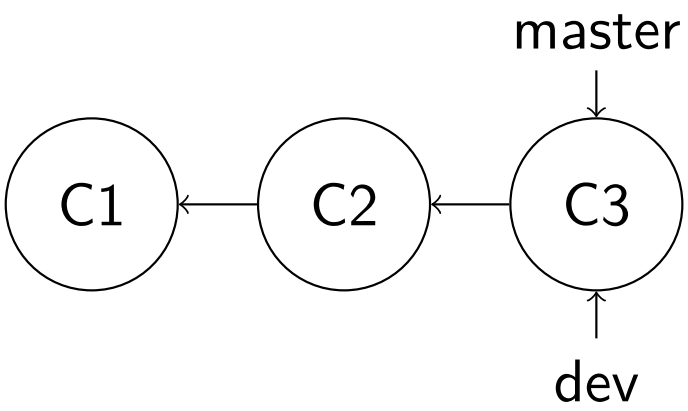
```
vi main.c
git add main.c
git commit -m 'Some function is added.'
```

这是在新的分支上的一个提交，不影响原来的分支。

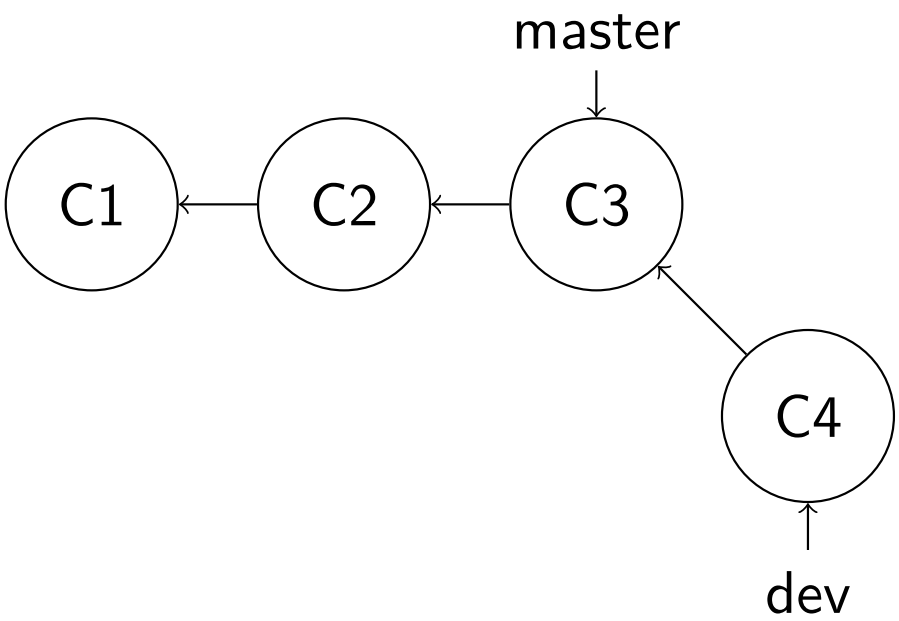
新的分支上的提交



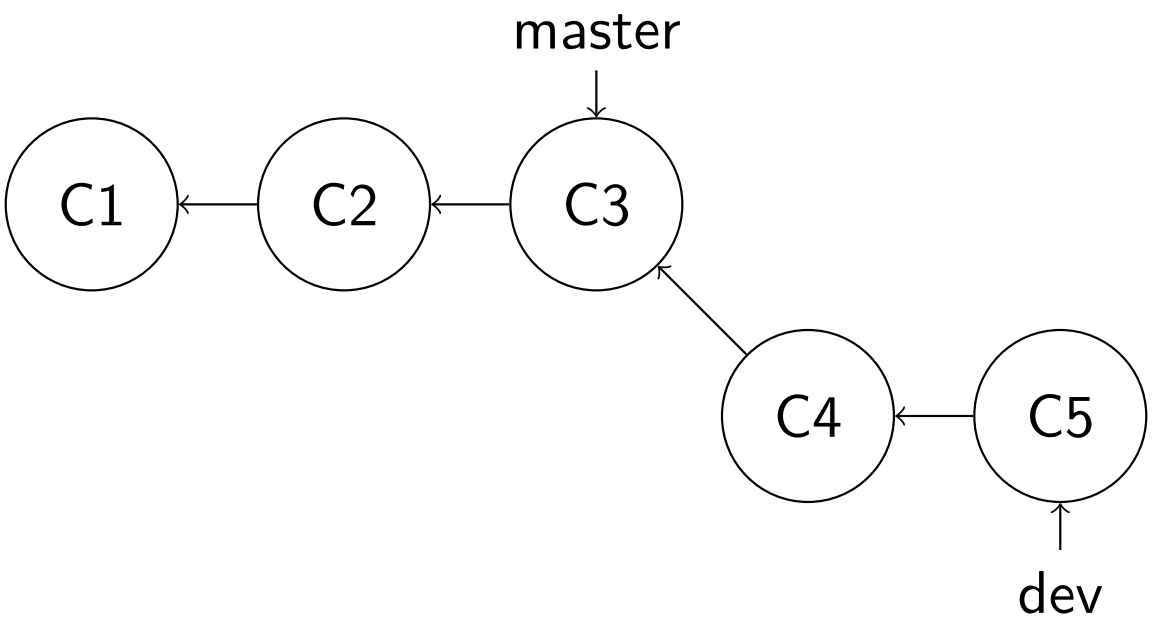
新的分支上的提交



新的分支上的提交



新的分支上的提交



切换分支

正在 dev 分支上工作，有用户报告已经发布的版本中的 bug，无法等到下一个版本发布，需要尽快解决。

当前工作先提交

```
git commit -a
```

切换到 master 分支，修复 bug 并提交

```
git checkout master
...
git commit -a
```

切换回来，继续 dev 分支上的开发工作

```
git checkout dev
...
git commit -a
```

切换分支

正在 dev 分支上工作，有用户报告已经发布的版本中的 bug，无法等到下一个版本发布，需要尽快解决。

当前工作先提交

```
git commit -a
```

切换到 master 分支，修复 bug 并提交

```
git checkout master
...
git commit -a
```

切换回来，继续 dev 分支上的开发工作

```
git checkout dev
...
git commit -a
```

切换分支

正在 dev 分支上工作，有用户报告已经发布的版本中的 bug，无法等到下一个版本发布，需要尽快解决。

当前工作先提交

```
git commit -a
```

切换到 master 分支，修复 bug 并提交

```
git checkout master
...
git commit -a
```

切换回来，继续 dev 分支上的开发工作

```
git checkout dev
...
git commit -a
```

切换分支

正在 dev 分支上工作，有用户报告已经发布的版本中的 bug，无法等到下一个版本发布，需要尽快解决。

当前工作先提交

```
git commit -a
```

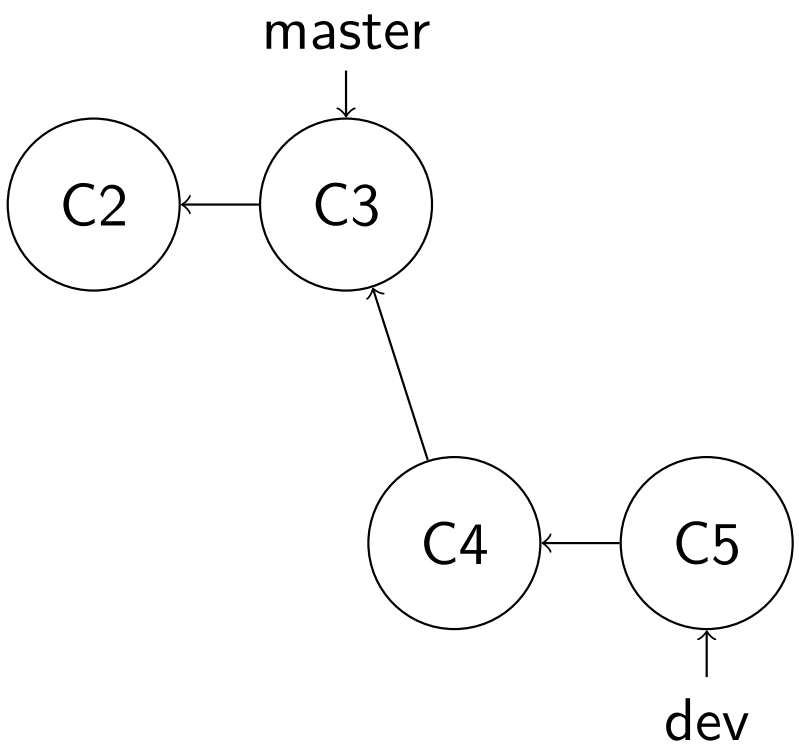
切换到 master 分支，修复 bug 并提交

```
git checkout master
...
git commit -a
```

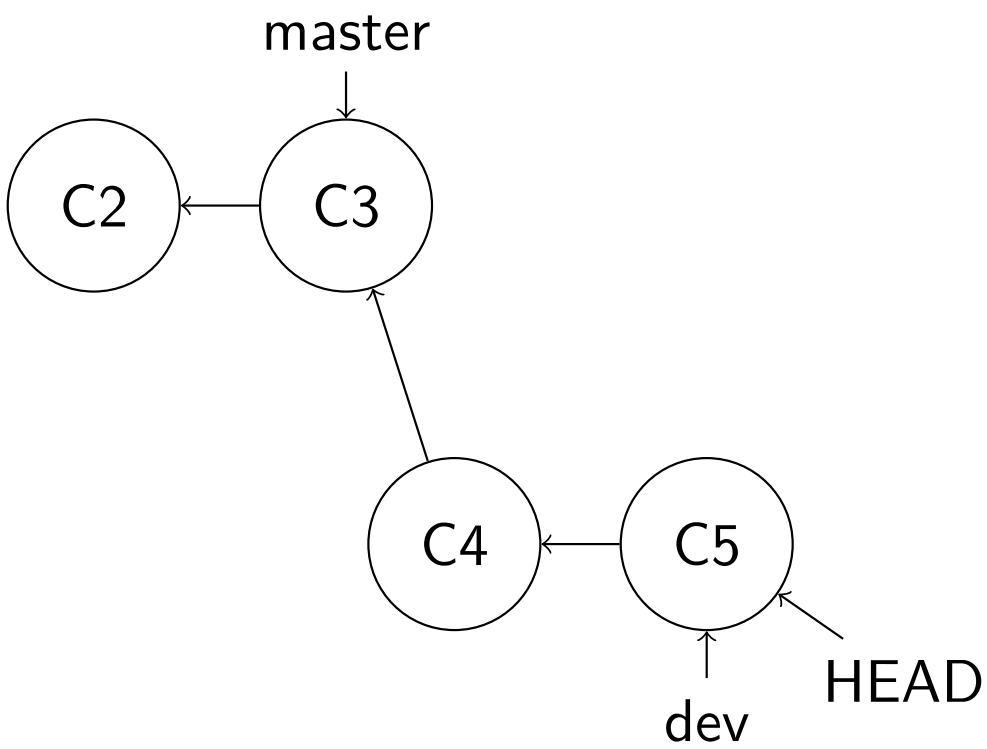
切换回来，继续 dev 分支上的开发工作

```
git checkout dev
...
git commit -a
```

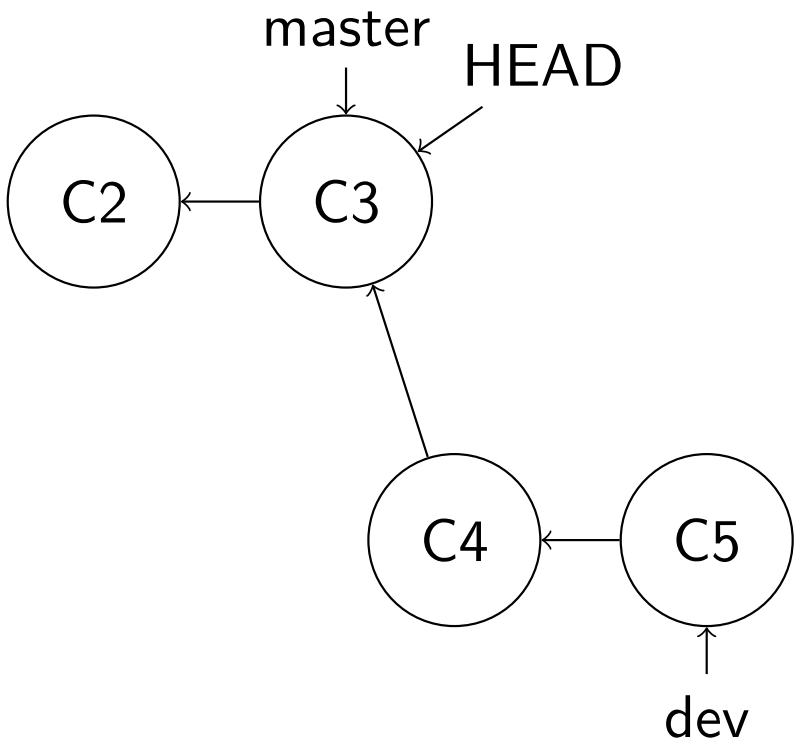
分支切换示意



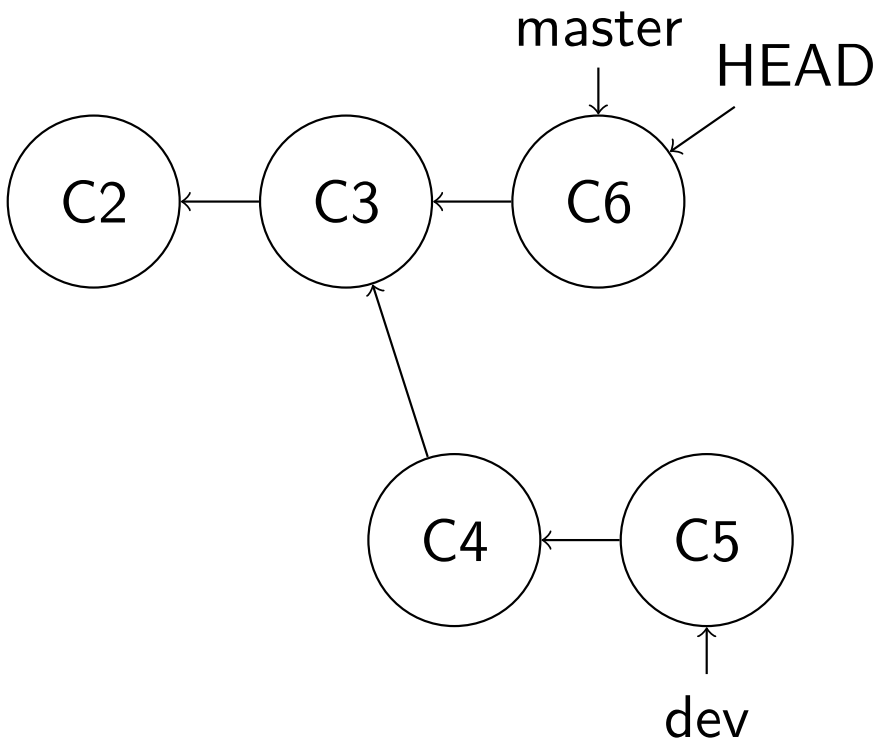
分支切换示意



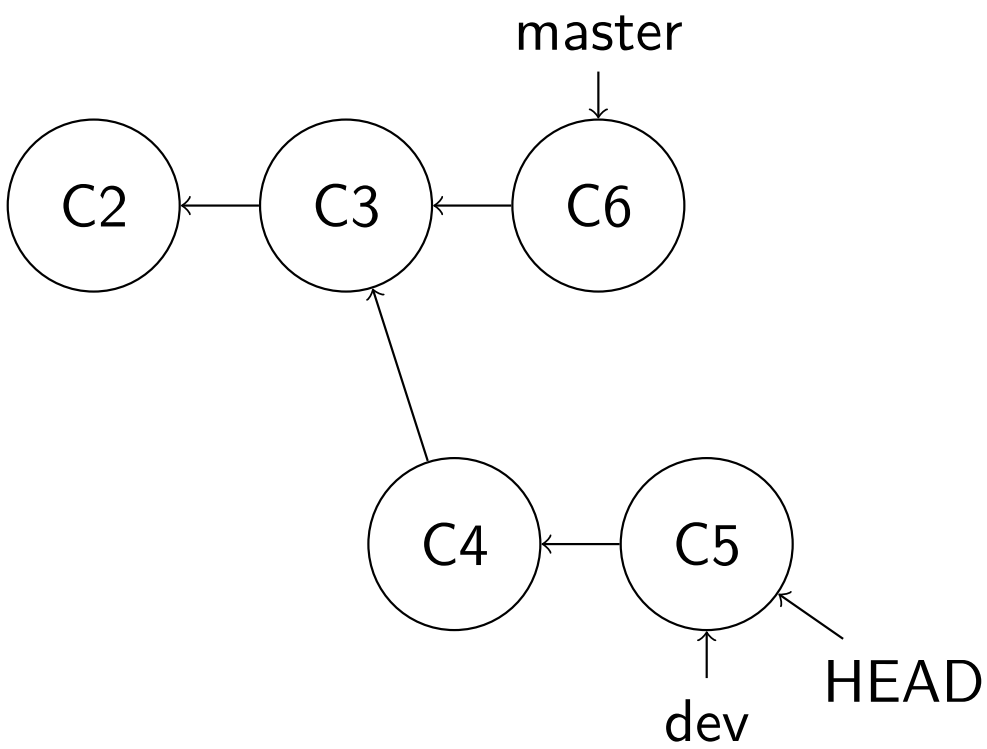
分支切换示意



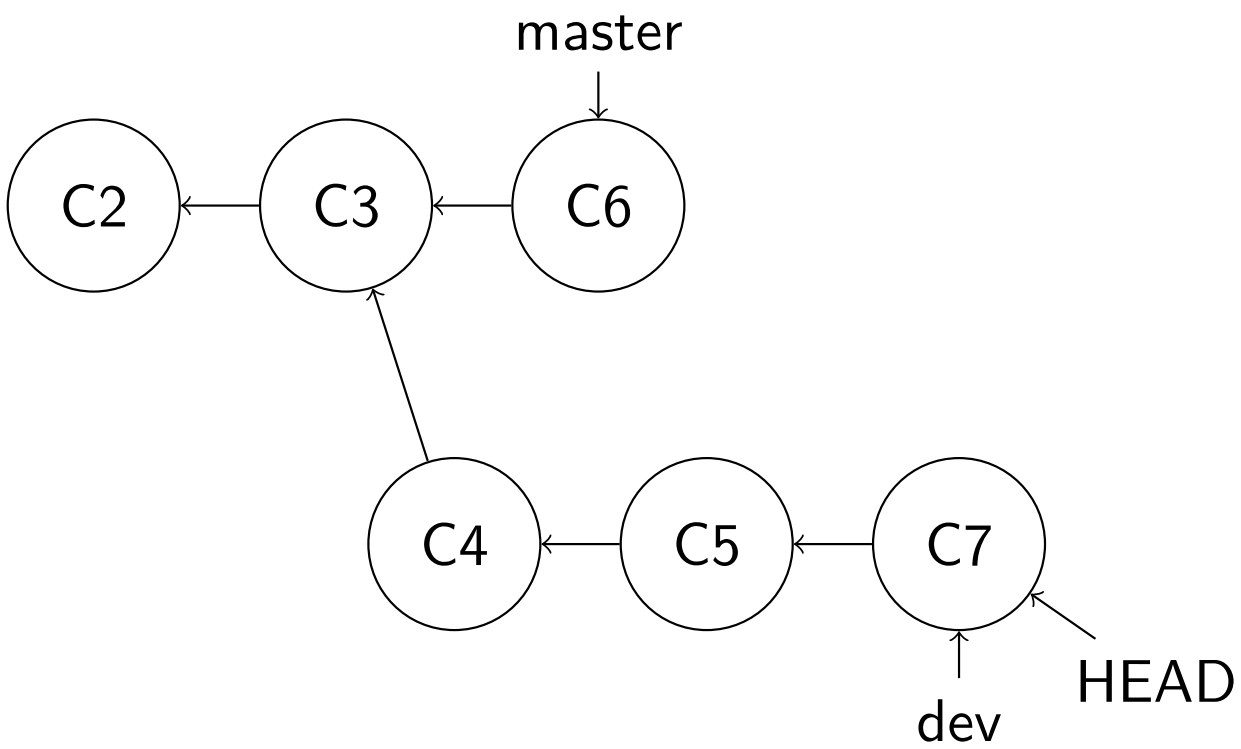
分支切换示意



分支切换示意



分支切换示意



分支合并

master 分支内所发现的 bug，在目前的 dev 分支内，也是存在的，希望把对那个 bug 的修复，合并到 dev 分支中。

```
git checkout dev
git merge master
```

若合并成功，将会自动产生新的提交。

```
git log
```

可以观察到

分支合并

master 分支内所发现的 bug，在目前的 dev 分支内，也是存在的，希望把对那个 bug 的修复，合并到 dev 分支中。

```
git checkout dev
git merge master
```

若合并成功，将会自动产生新的提交。

```
git log
```

可以观察到

分支合并

master 分支内所发现的 bug，在目前的 dev 分支内，也是存在的，希望把对那个 bug 的修复，合并到 dev 分支中。

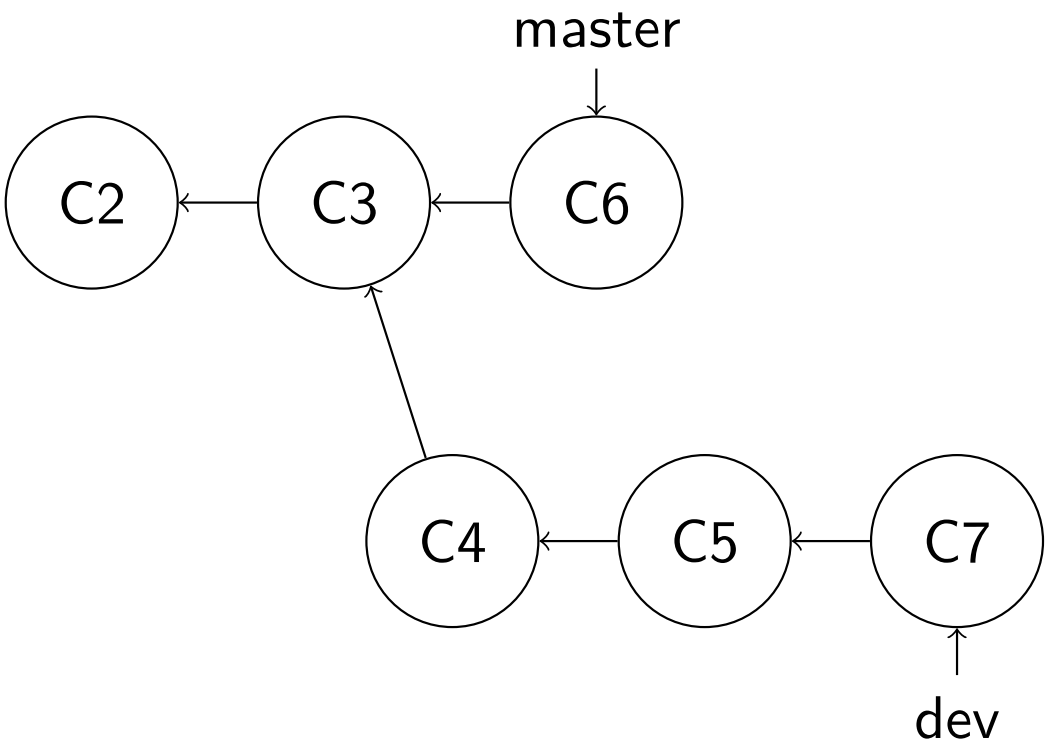
```
git checkout dev
git merge master
```

若合并成功，将会自动产生新的提交。

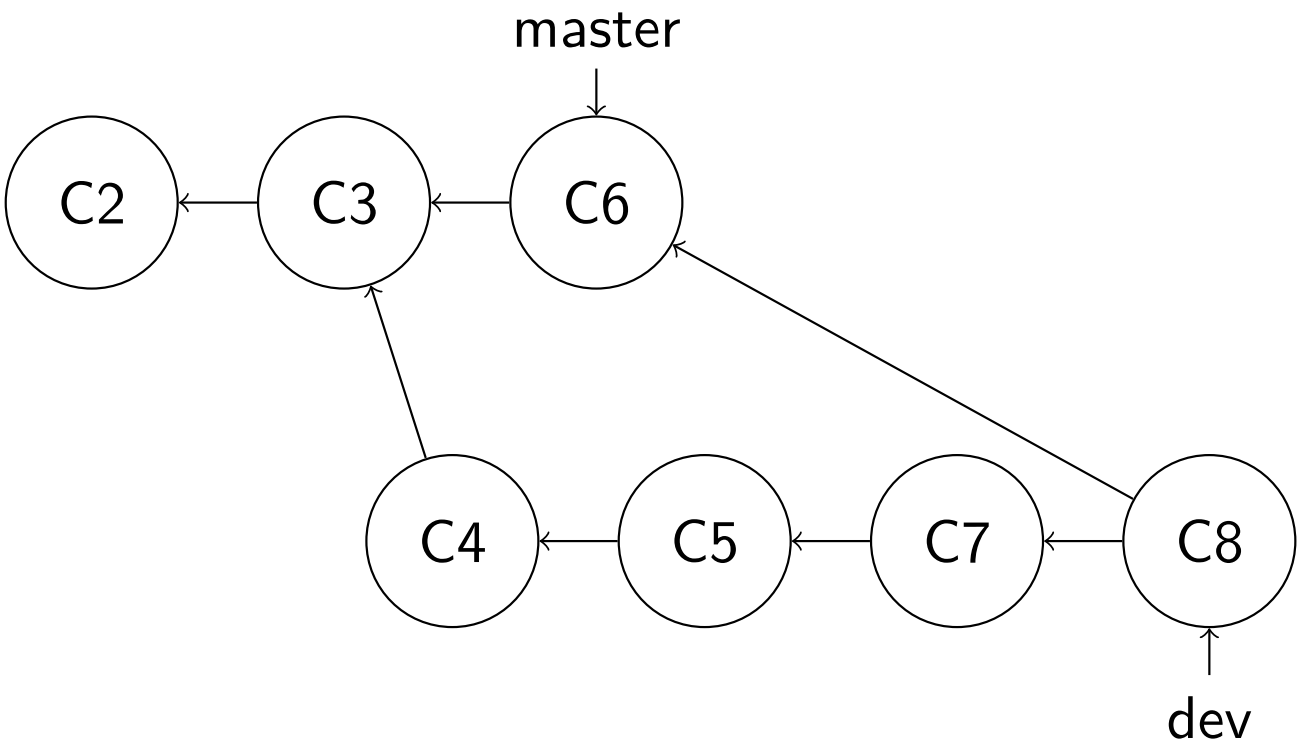
```
git log
```

可以观察到

master 合并到 dev



master 合并到 dev



更加专业的做法

- Git 分支与合并的成本很低
- 在新的分支上修改，然后合并回来的做法，更加专业
- 修复新发现 bug 的通行做法是

```
git checkout master
git branch fixbug
git checkout fixbug
# Fix bugs ...
git add ...
git commit
# Test
git checkout master
git merge fixbug
git branch -d fixbug
```

- 上面最后一条是删除分支的命令

更加专业的做法

- Git 分支与合并的成本很低
- 在新的分支上修改，然后合并回来的做法，更加专业
- 修复新发现 bug 的通行做法是

```
git checkout master
git branch fixbug
git checkout fixbug
# Fix bugs ...
git add ...
git commit
# Test
git checkout master
git merge fixbug
git branch -d fixbug
```

- 上面最后一条是删除分支的命令

更加专业的做法

- Git 分支与合并的成本很低
- 在新的分支上修改，然后合并回来的做法，更加专业
- 修复新发现 bug 的通行做法是

```
git checkout master
git branch fixbug
git checkout fixbug
# Fix bugs ...
git add ...
git commit
# Test
git checkout master
git merge fixbug
git branch -d fixbug
```

- 上面最后一条是删除分支的命令

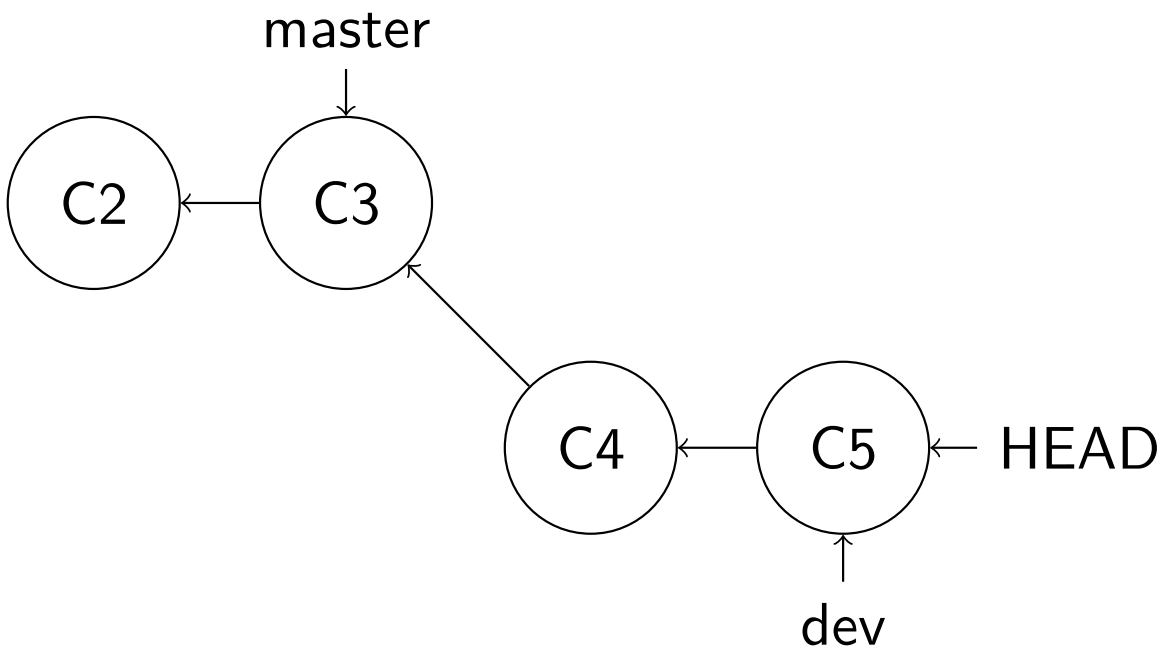
更加专业的做法

- Git 分支与合并的成本很低
- 在新的分支上修改，然后合并回来的做法，更加专业
- 修复新发现 bug 的通行做法是

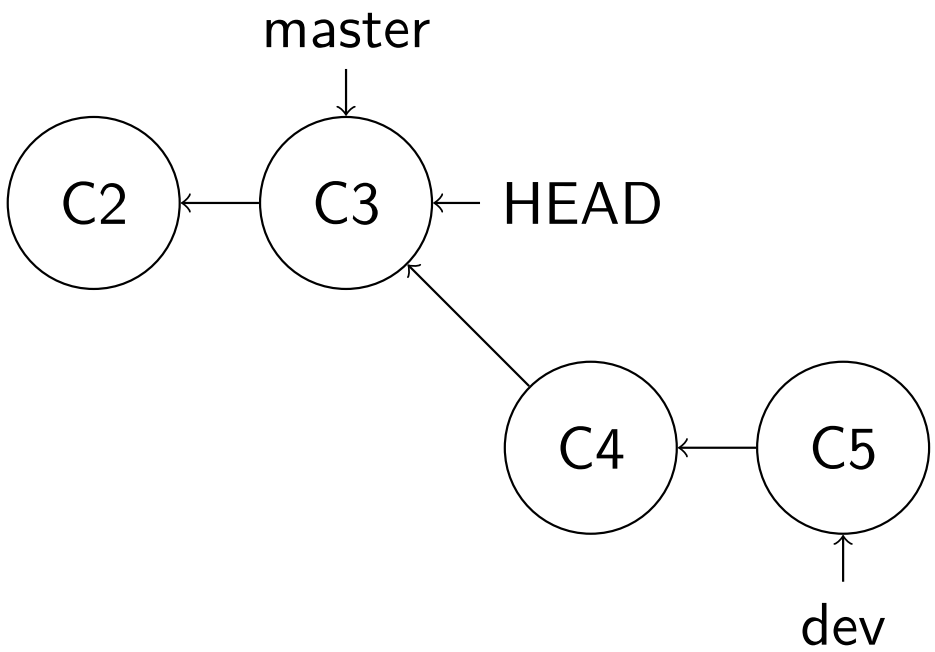
```
git checkout master
git branch fixbug
git checkout fixbug
# Fix bugs ...
git add ...
git commit
# Test
git checkout master
git merge fixbug
git branch -d fixbug
```

- 上面最后一条是删除分支的命令

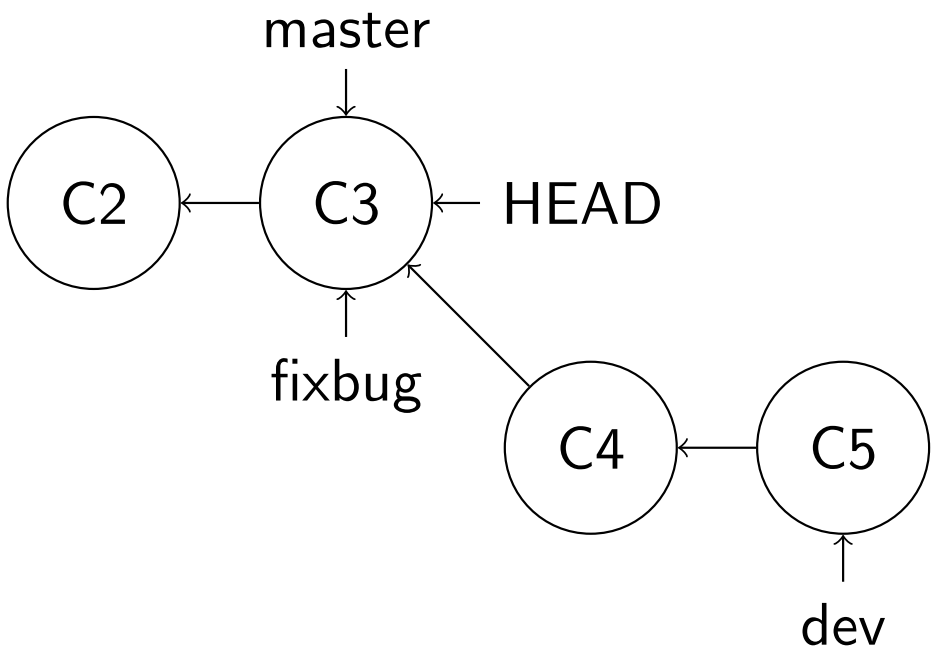
在新建的修复分支中处理缺陷



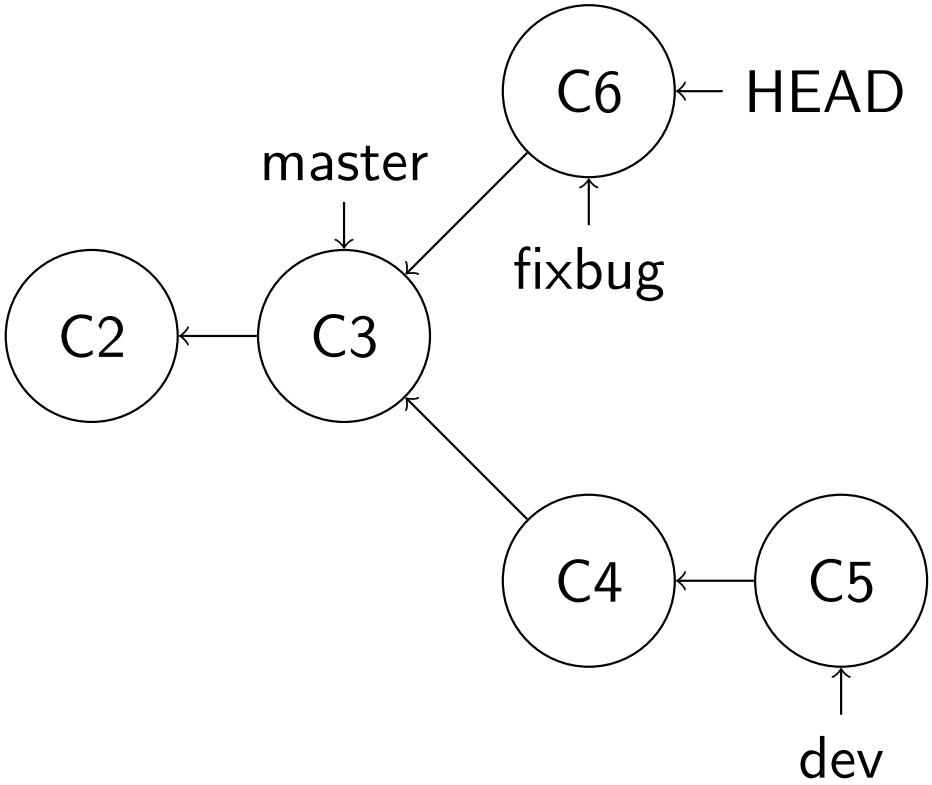
在新建的修复分支中处理缺陷



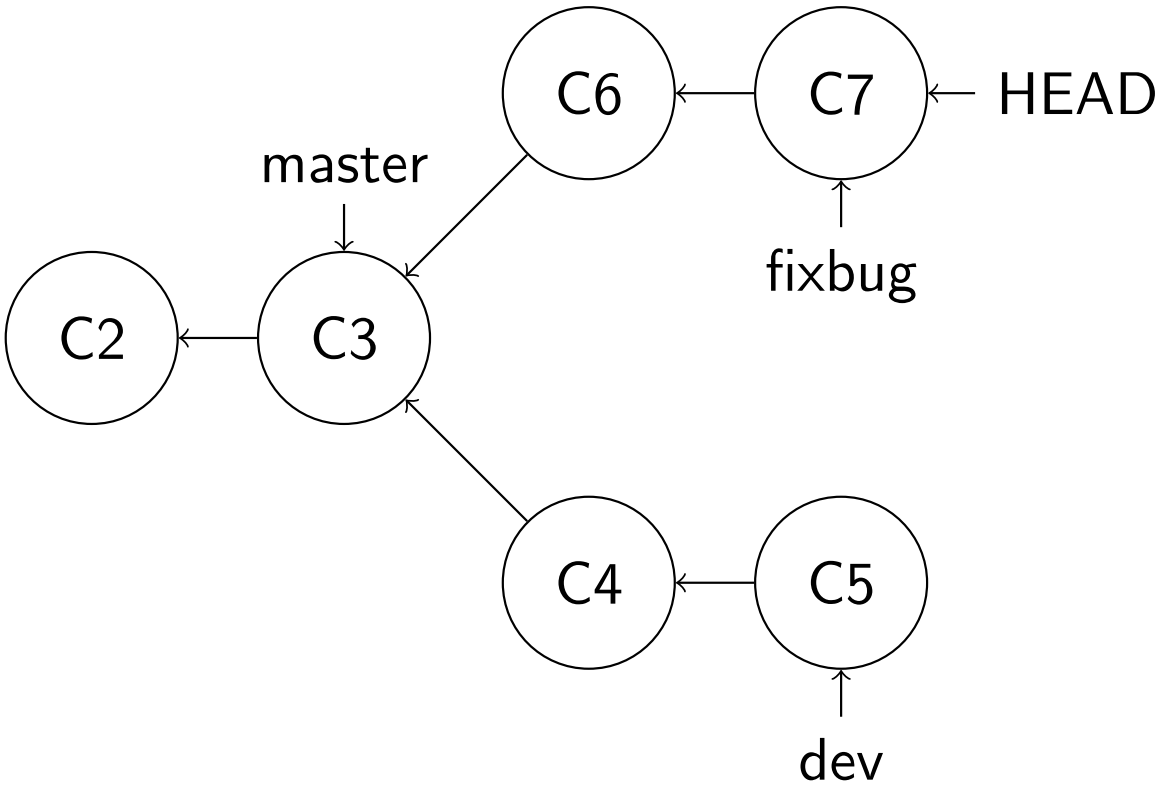
在新建的修复分支中处理缺陷



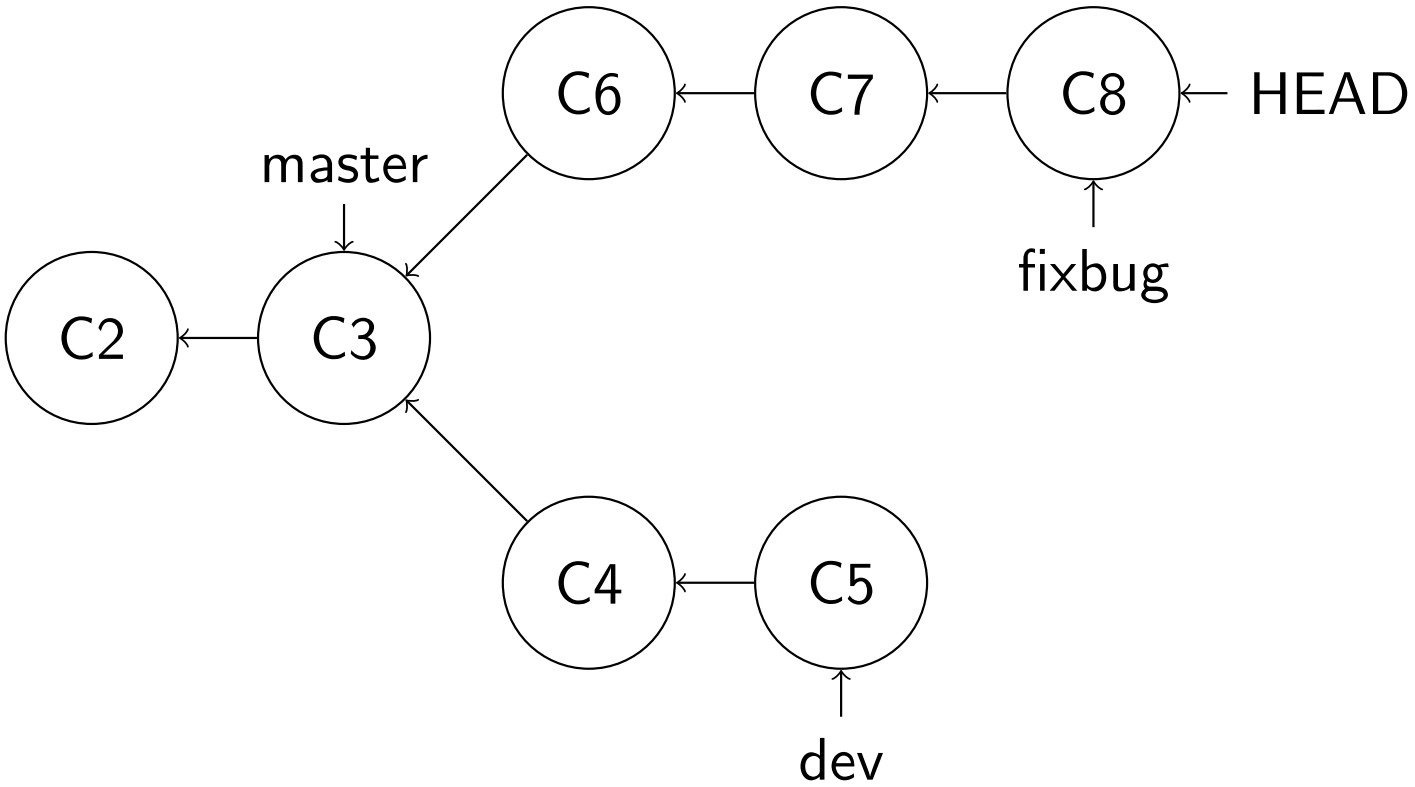
在新建的修复分支中处理缺陷



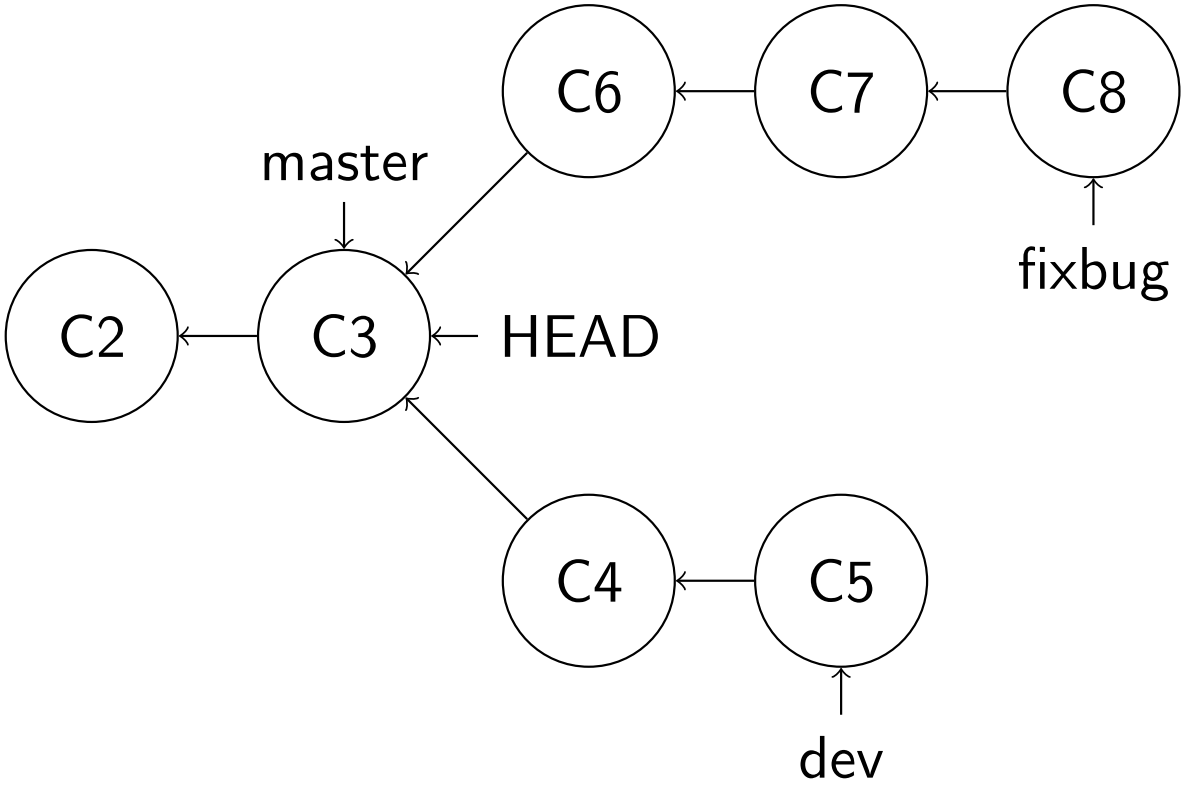
在新建的修复分支中处理缺陷



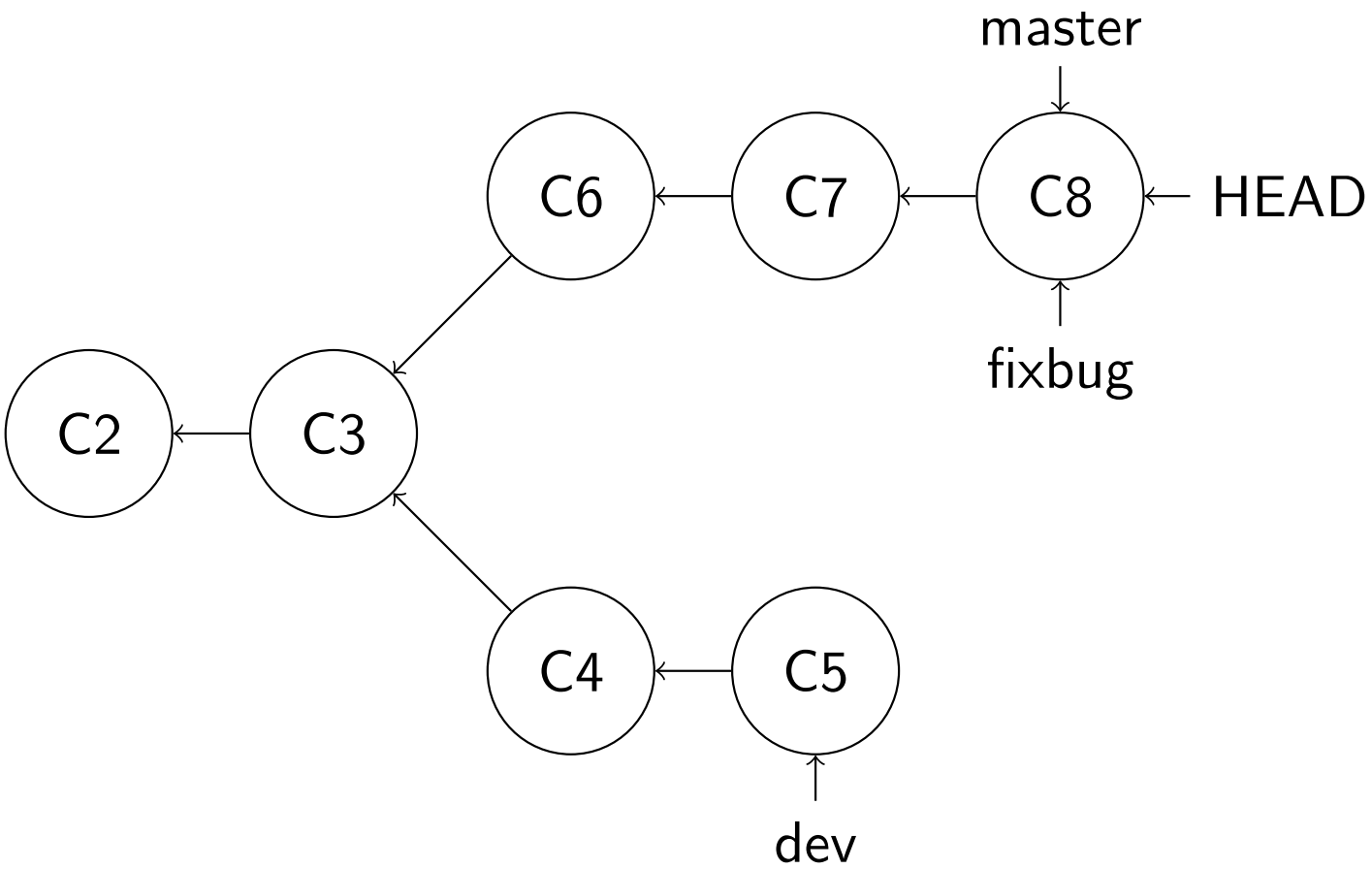
在新建的修复分支中处理缺陷



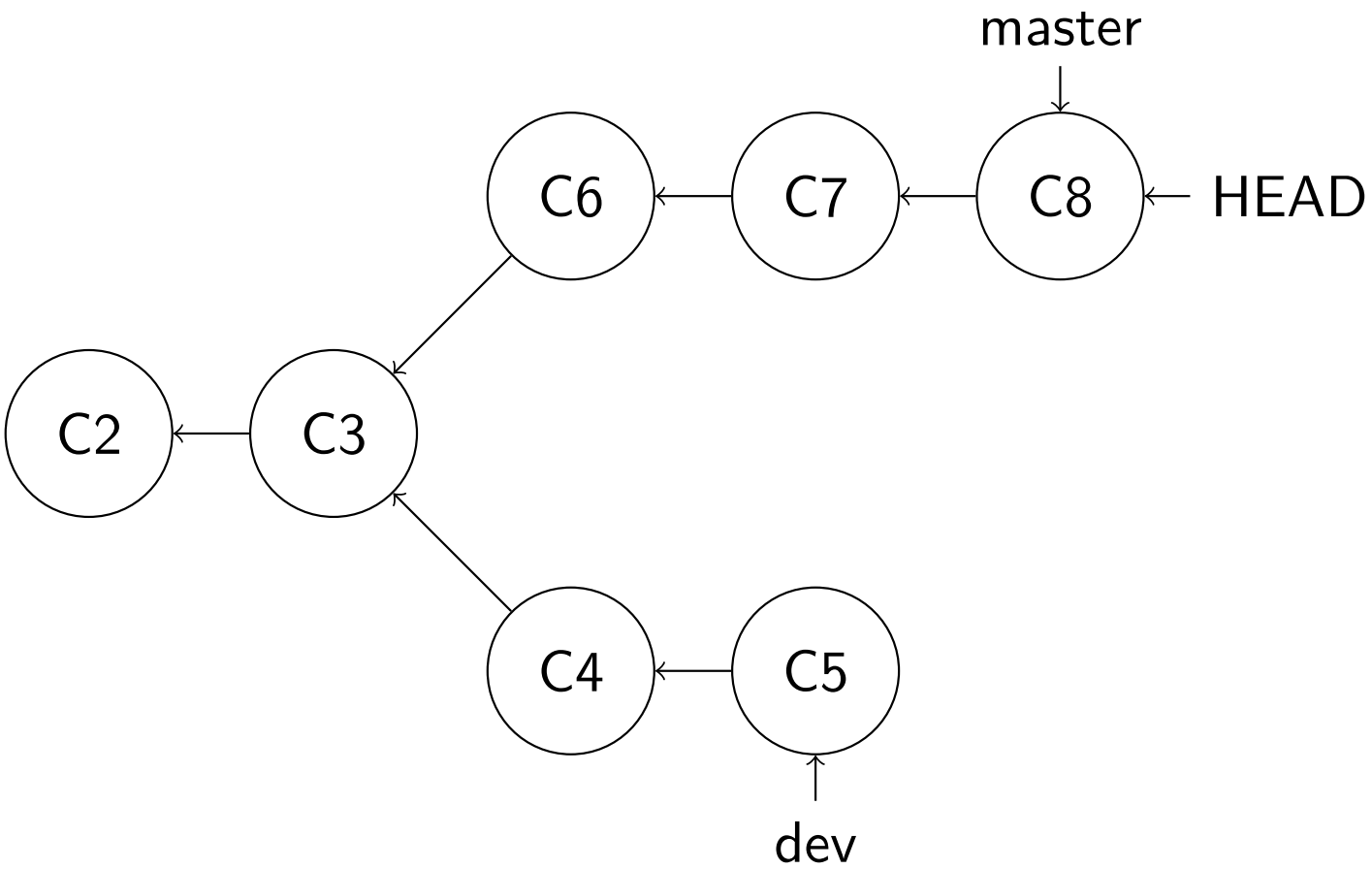
在新建的修复分支中处理缺陷



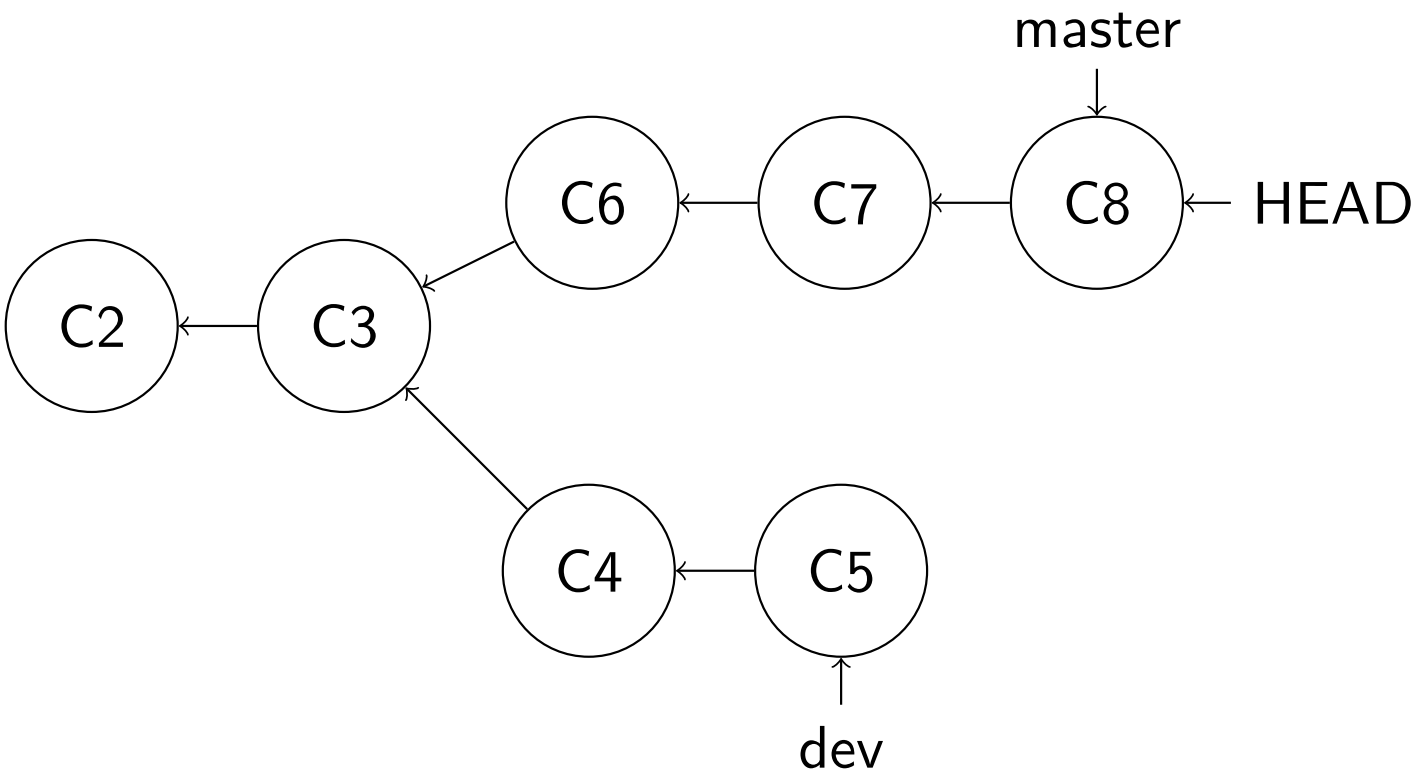
在新建的修复分支中处理缺陷



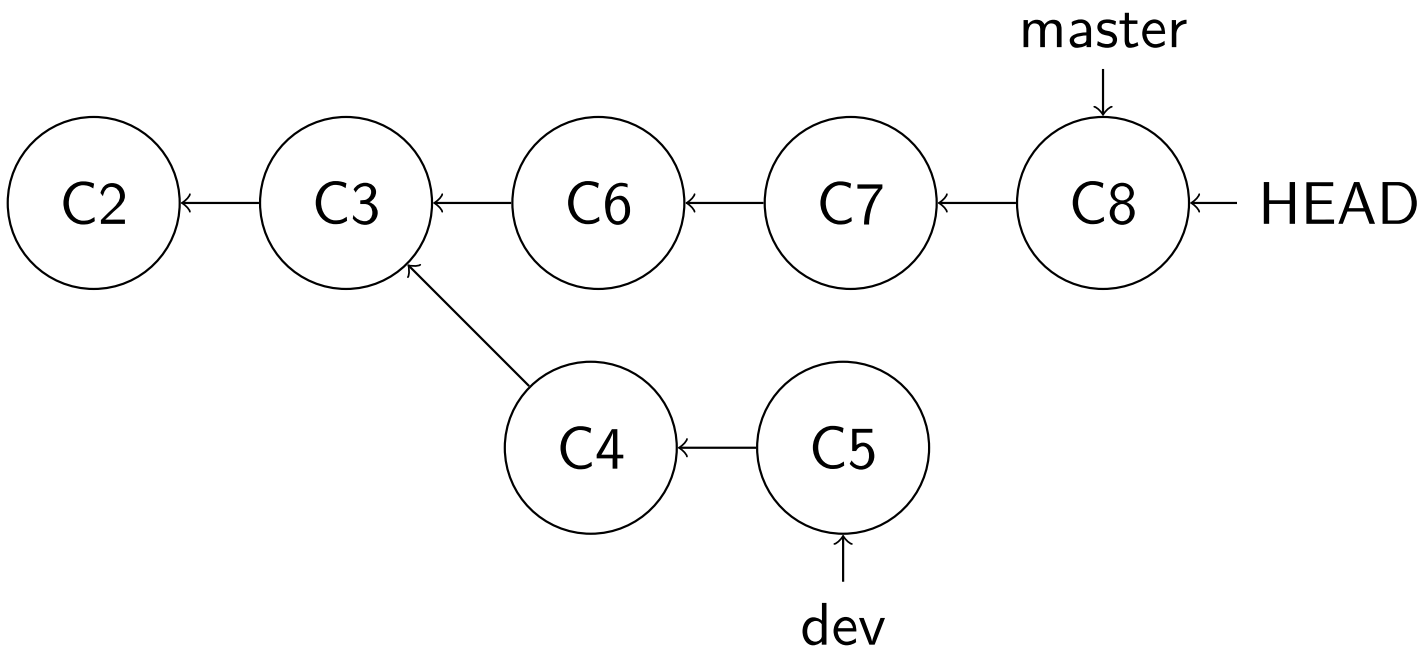
在新建的修复分支中处理缺陷



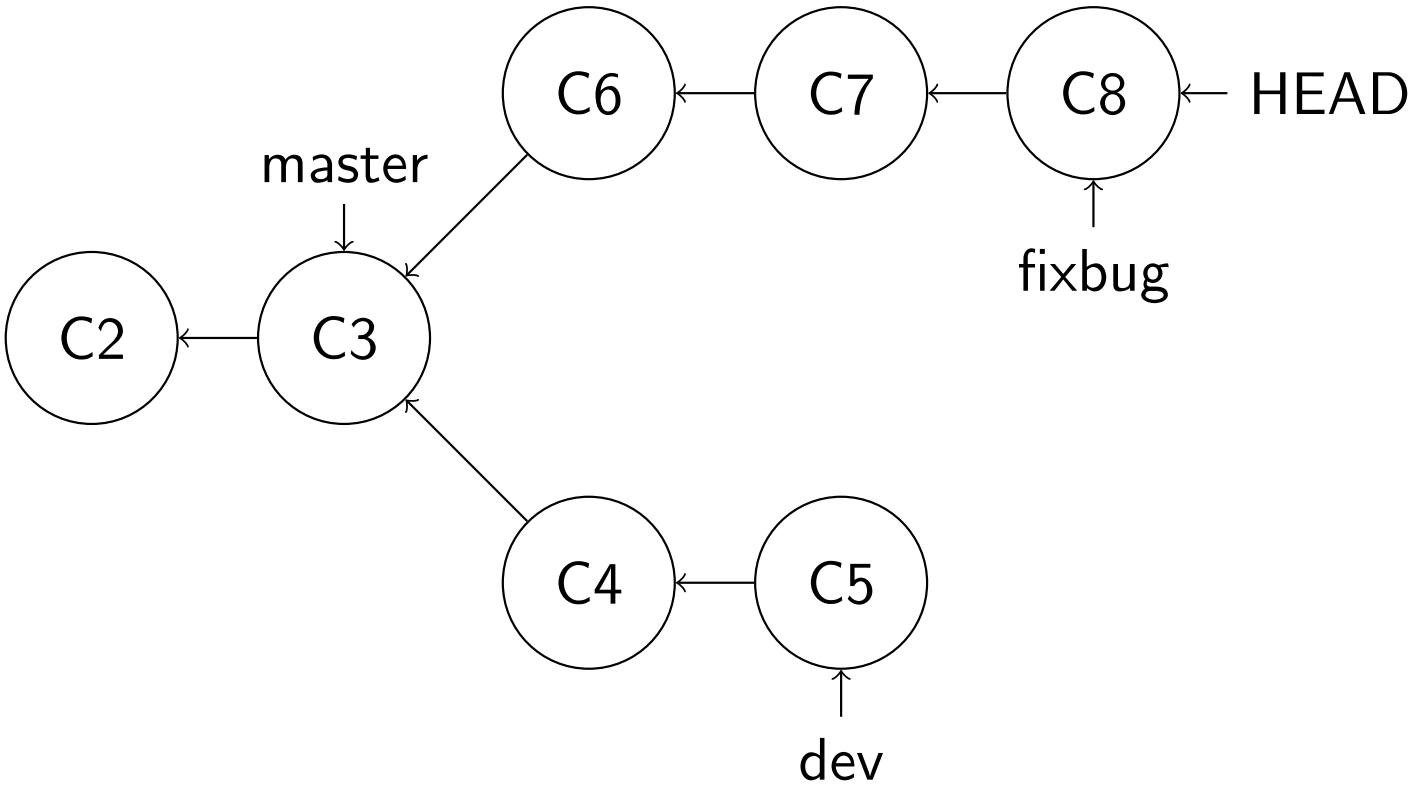
在新建的修复分支中处理缺陷



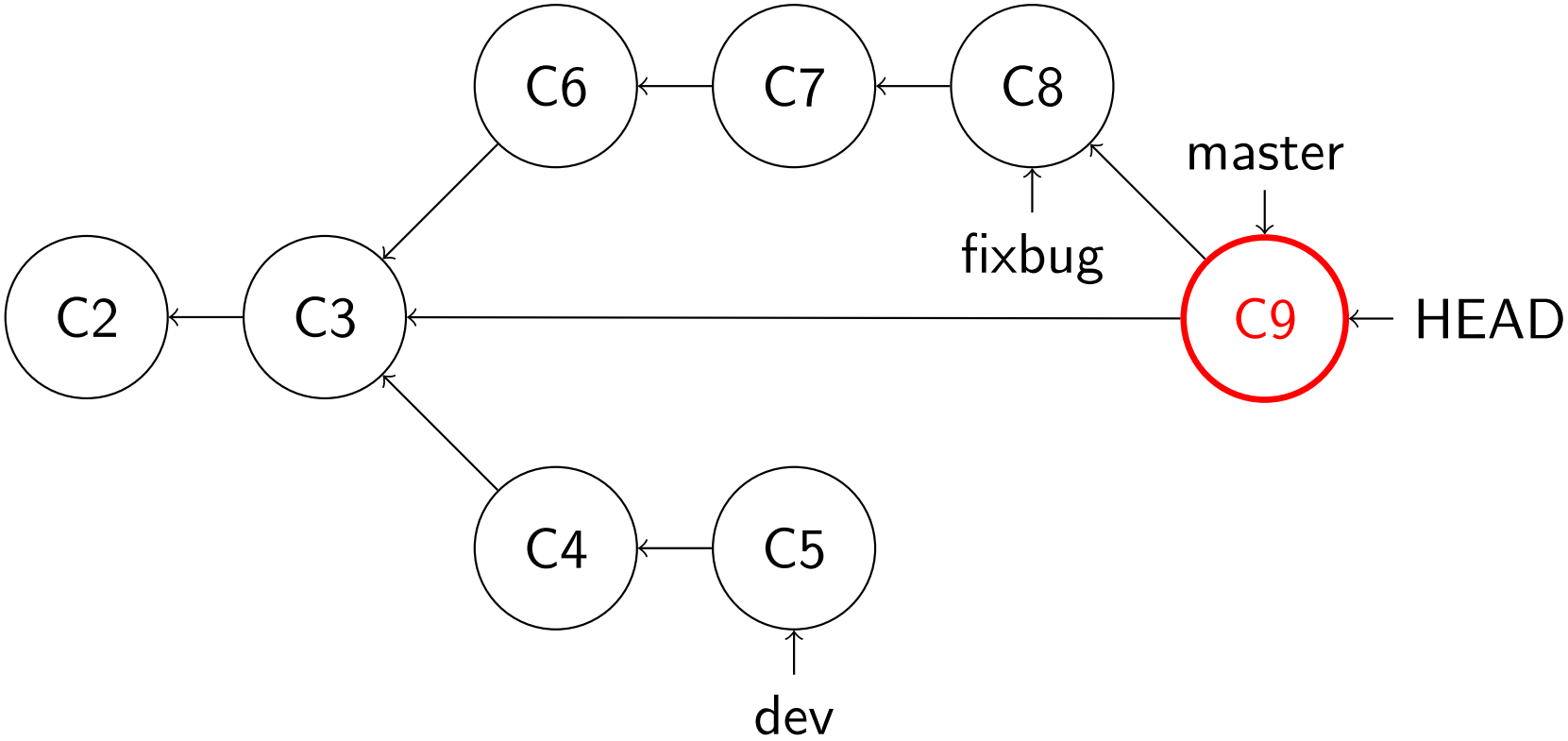
在新建的修复分支中处理缺陷



在新建的修复分支中处理缺陷



在新建的修复分支中处理缺陷



更简洁的命令

- 创建新分支并切换过去

```
git checkout -b fixbug
```

- 被修改的文件，加入暂存区并提交

```
git commit -a
```

- 切换到新分支的同时，把当前修改过的文件一起合并过去

```
git checkout -m master
```

要注意合并是否出现冲突

更简洁的命令

- 创建新分支并切换过去

```
git checkout -b fixbug
```

- 被修改的文件，加入暂存区并提交

```
git commit -a
```

- 切换到新分支的同时，把当前修改过的文件一起合并过去

```
git checkout -m master
```

要注意合并是否出现冲突

更简洁的命令

- 创建新分支并切换过去

```
git checkout -b fixbug
```

- 被修改的文件，加入暂存区并提交

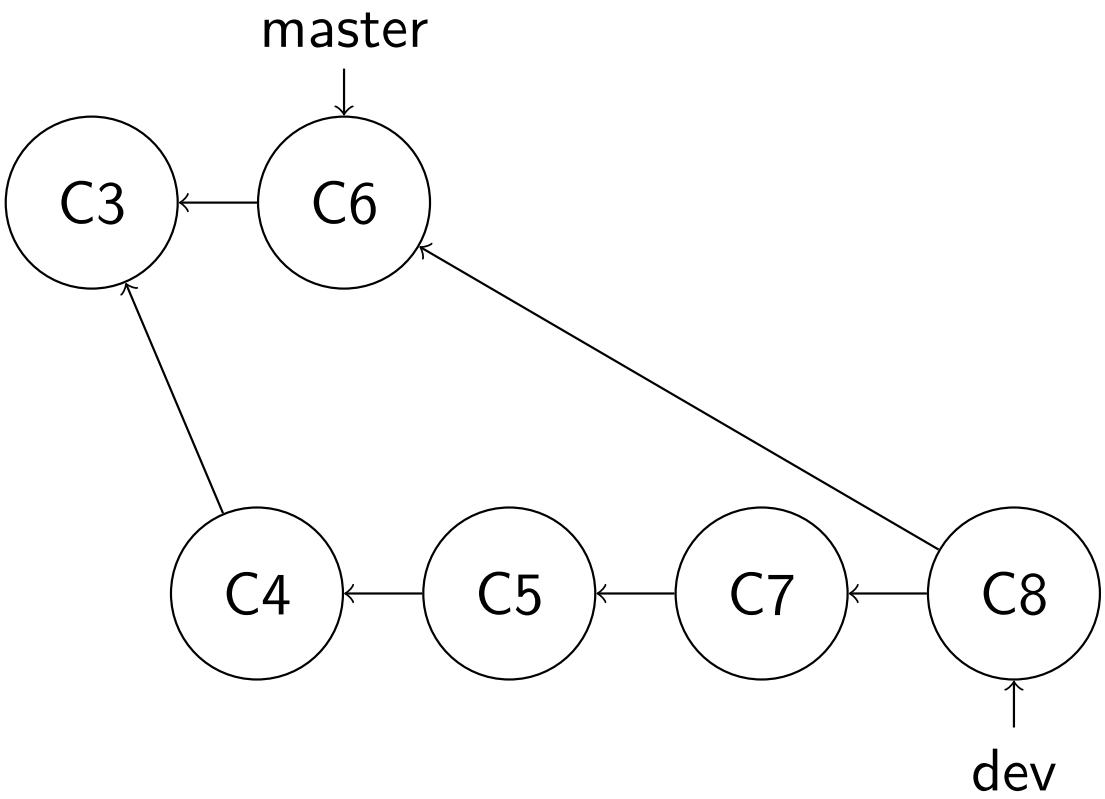
```
git commit -a
```

- 切换到新分支的同时，把当前修改过的文件一起合并过去

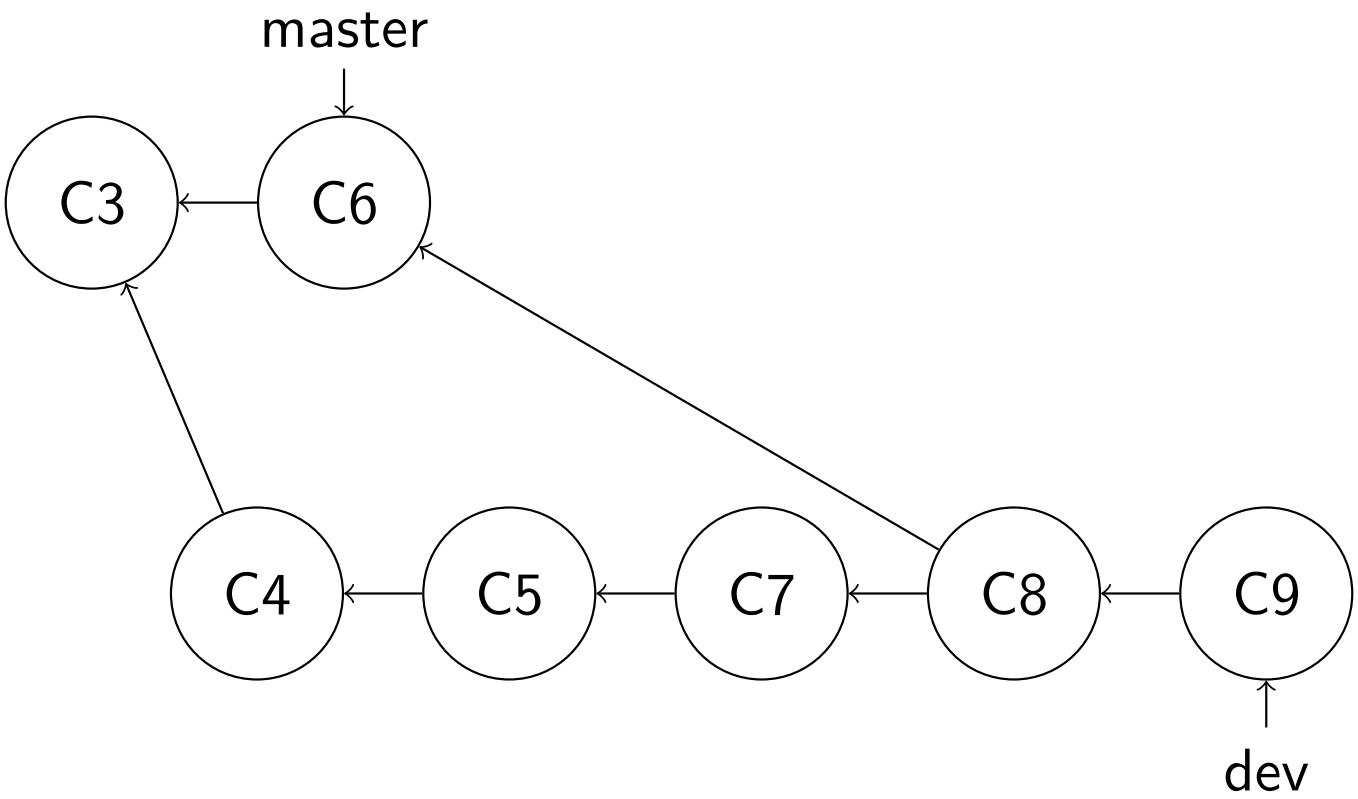
```
git checkout -m master
```

要注意合并是否出现冲突

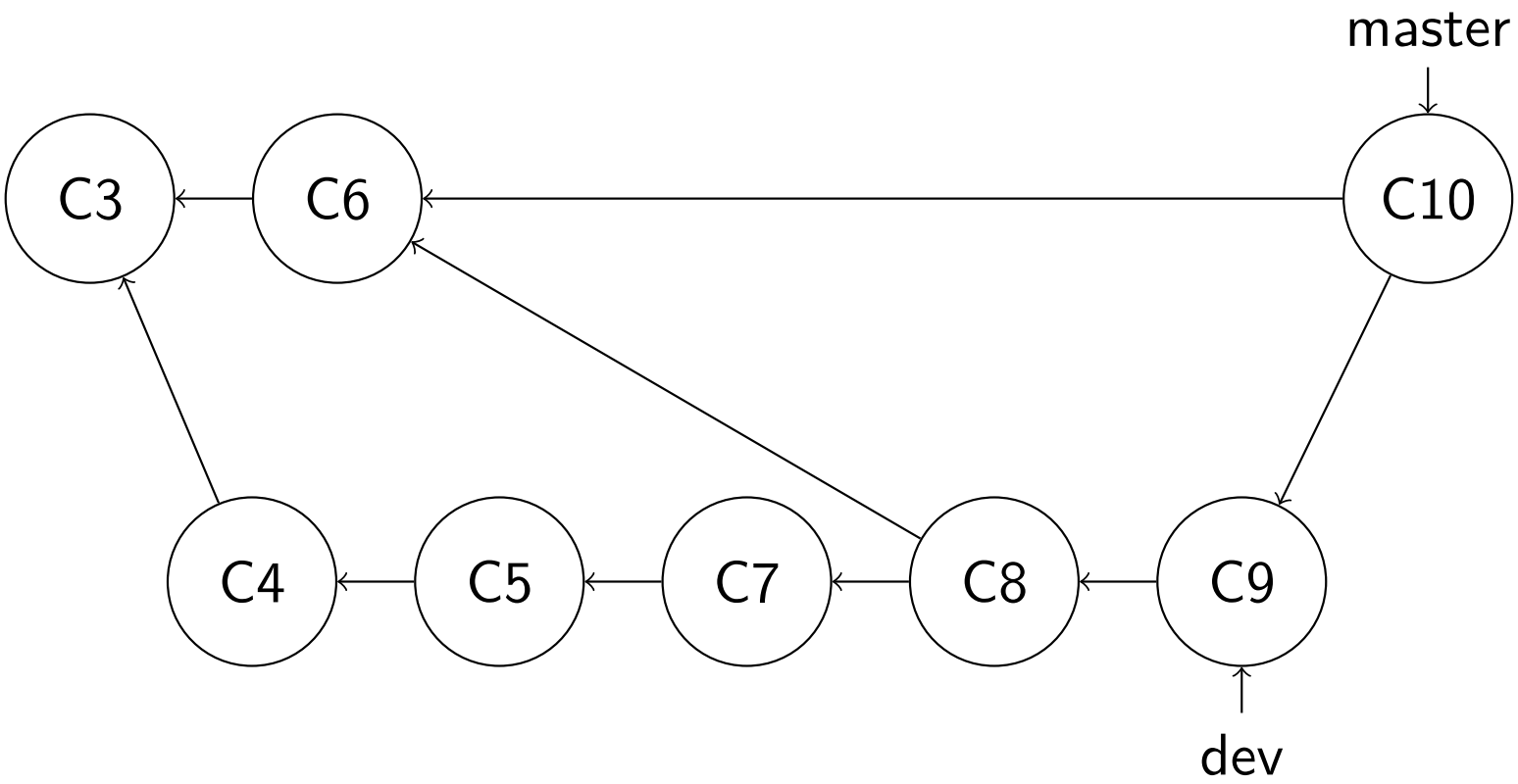
dev 合并到 master



dev 合并到 master



dev 合并到 master



分支合并冲突

不同分支上，对同一部分的修改不一致，合并时会产生冲突

```
$ git merge dev
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts
and then commit the result.
```

合并发生冲突时，不会自动产生新的提交。

分支合并冲突

不同分支上，对同一部分的修改不一致，合并时会产生冲突

```
$ git merge dev
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts
and then commit the result.
```

合并发生冲突时，不会自动产生新的提交。

分支合并冲突（续）

产生冲突的文件内容变成

```
#include<stdio.h>
int main()
{
        printf("Hello, World!\n");
<<<<<<< HEAD
        printf("This is second line.\n");
=======
        printf("The 2nd line is here.\n");
>>>>>>> dev
}
```

解决合并时的冲突

手工解决冲突，然后提交

```
vi main.c
...
git add main.c
git commit
```

小结

人们在使用 Git 作为源代码控制的情况，绝大多数工作流程如下：

- 修改代码 . . . 提交修改 . . .
- 创建分支，切换分支
- 修改代码 . . . 提交修改 . . .
- 切换分支
- 修改代码 . . . 提交修改 . . .
- 切换分支
- 合并分支
- . . .

Part IV

多 Git 一点

查看分支

- 当前所在分支

```
git branch
```

- 了解分支情况

```
git show-branch
```

- 绘出分支历史关系图

```
git log --graph --all --pretty=oneline
```

查看分支

- 当前所在分支

```
git branch
```

- 了解分支情况

```
git show-branch
```

- 绘出分支历史关系图

```
git log --graph --all --pretty=oneline
```

查看分支

- 当前所在分支

```
git branch
```

- 了解分支情况

```
git show-branch
```

- 绘出分支历史关系图

```
git log --graph --all --pretty=oneline
```

观察历史

- 每一次提交都被保存在仓库中，用一串 sha1 校验和表示
- 观察每一次的提交：

```
git log
```

- 或者

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

观察历史

- 每一次提交都被保存在仓库中，用一串 sha1 校验和表示
- 观察每一次的提交：

```
git log
```

- 或者

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

观察历史

- 每一次提交都被保存在仓库中，用一串 sha1 校验和表示
- 观察每一次的提交：

```
git log
```

- 或者

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

标识历史上的每一个提交

- 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
- 绝对提交名的简写：533e31
- 参照名：master, dev, fixbug, v1.0, remotes/origin/master
- 符号参照名：
  `HEAD`, `ORIG_HEAD`, `FETCH_HEAD`, `MERGE_HEAD`
- 相对提交名：
  `master^`, `master^^`, `master~2`, `master~10^2`

标识历史上的每一个提交

- 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
- 绝对提交名的简写：533e31
- 参照名：master, dev, fixbug, v1.0, remotes/origin/master
- 符号参照名：
  `HEAD`, `ORIG_HEAD`, `FETCH_HEAD`, `MERGE_HEAD`
- 相对提交名：
  `master^`, `master^^`, `master~2`, `master~10^2`

标识历史上的每一个提交

- 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
- 绝对提交名的简写：533e31
- 参照名：master, dev, fixbug, v1.0, remotes/origin/master
- 符号参照名：
  `HEAD, ORIG_HEAD, FETCH_HEAD, MERGE_HEAD`
- 相对提交名：
  `master^, master^^, master~2, master~10^2`

标识历史上的每一个提交

- 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
- 绝对提交名的简写：533e31
- 参照名：master, dev, fixbug, v1.0, remotes/origin/master
- 符号参照名：
  `HEAD, ORIG_HEAD, FETCH_HEAD, MERGE_HEAD`
- 相对提交名：
  `master^, master^^, master~2, master~10^2`

标识历史上的每一个提交

- 绝对提交名：533e3140bffee43b02c5648c8fcc3e63232739a6
- 绝对提交名的简写：533e31
- 参照名：master, dev, fixbug, v1.0, remotes/origin/master
- 符号参照名：
  `HEAD, ORIG_HEAD, FETCH_HEAD, MERGE_HEAD`
- 相对提交名：
  `master^, master^^, master~2, master~10^2`

相对提交名

- 尝试

```
git rev-parse master
```

- 尝试

```
git show-branch --more=3 --all
```

- 尝试

```
git rev-parse master~2
```

- 尝试

```
git rev-parse HEAD^
git rev-parse HEAD^^
git rev-parse HEAD^2
```

相对提交名

- 尝试

```
git rev-parse master
```

- 尝试

```
git show-branch --more=3 --all
```

- 尝试

```
git rev-parse master~2
```

- 尝试

```
git rev-parse HEAD^
git rev-parse HEAD^^
git rev-parse HEAD^2
```

相对提交名

- 尝试

```
git rev-parse master
```

- 尝试

```
git show-branch --more=3 --all
```

- 尝试

```
git rev-parse master~2
```

- 尝试

```
git rev-parse HEAD^
git rev-parse HEAD^^
git rev-parse HEAD^2
```

相对提交名

- 尝试

```
git rev-parse master
```

- 尝试

```
git show-branch --more=3 --all
```

- 尝试

```
git rev-parse master~2
```

- 尝试

```
git rev-parse HEAD^
git rev-parse HEAD^^
git rev-parse HEAD^2
```

标签

- 给当前的版本加标签

```
git tag -a v1.0
```

- 查看标签

```
git tag
```

- 了解标签所对应的提交名

```
git rev-parse v1.0
```

- 在提交历史中显示标签

```
git log --decorate
```

标签

- 给当前的版本加标签

```
git tag -a v1.0
```

- 查看标签

```
git tag
```

- 了解标签所对应的提交名

```
git rev-parse v1.0
```

- 在提交历史中显示标签

```
git log --decorate
```

标签

- 给当前的版本加标签

```
git tag -a v1.0
```

- 查看标签

```
git tag
```

- 了解标签所对应的提交名

```
git rev-parse v1.0
```

- 在提交历史中显示标签

```
git log --decorate
```

标签

- 给当前的版本加标签

```
git tag -a v1.0
```

- 查看标签

```
git tag
```

- 了解标签所对应的提交名

```
git rev-parse v1.0
```

- 在提交历史中显示标签

```
git log --decorate
```

标签（续）

- 给历史上的版本加标签

```
git checkout fd311e
git tag -a v1.0 -m 'first release'
git checkout dev
```

- 或者

```
git tag -a v1.1 9feb0 -m 'improved version'
```

- 轻量标签

```
git tag birth-day-revision
```

标签（续）

- 给历史上的版本加标签

```
git checkout fd311e
git tag -a v1.0 -m 'first release'
git checkout dev
```

- 或者

```
git tag -a v1.1 9feb0 -m 'improved version'
```

- 轻量标签

```
git tag birth-day-revision
```

标签（续）

- 给历史上的版本加标签

```
git checkout fd311e
git tag -a v1.0 -m 'first release'
git checkout dev
```

- 或者

```
git tag -a v1.1 9feb0 -m 'improved version'
```

- 轻量标签

```
git tag birth-day-revision
```

穿越：回到从前

- 要检查某个历史上的版本

```
git checkout 533e31
```

- 或者

```
git checkout v1.0
git checkout master~3
```

- 试试

```
git checkout :/"some strings to find in comment"
```

- 回到当下

```
git checkout dev
```

穿越：回到从前

- 要检查某个历史上的版本

```
git checkout 533e31
```

- 或者

```
git checkout v1.0
git checkout master~3
```

- 试试

```
git checkout :/"some strings to find in comment"
```

- 回到当下

```
git checkout dev
```

穿越：回到从前

- 要检查某个历史上的版本

```
git checkout 533e31
```

- 或者

```
git checkout v1.0
git checkout master~3
```

- 试试

```
git checkout :/"some strings to find in comment"
```

- 回到当下

```
git checkout dev
```

穿越：回到从前

- 要检查某个历史上的版本

```
git checkout 533e31
```

- 或者

```
git checkout v1.0
git checkout master~3
```

- 试试

```
git checkout :/"some strings to find in comment"
```

- 回到当下

```
git checkout dev
```

基于历史版本的修改

- 'detached HEAD' state 概念
- 查看/修改/试验/提交/抛弃
- 另建分支以便将来可以找到它

```
...
git checkout -b new_branch_name
...
git commit
git checkout dev
```

基于历史版本的修改

- 'detached HEAD' state 概念
- 查看/修改/试验/提交/抛弃
- 另建分支以便将来可以找到它

```
...
git checkout -b new_branch_name
...
git commit
git checkout dev
```

基于历史版本的修改

- 'detached HEAD' state 概念
- 查看/修改/试验/提交/抛弃
- 另建分支以便将来可以找到它

```
...
git checkout -b new_branch_name
...
git commit
git checkout dev
```

干净的工作目录

- 已修改未更新
- 已更新待提交
- 提交后又修改
- 不干净的工作目录，妨碍你切换到其他版本

干净的工作目录

- 已修改未更新
- 已更新待提交
- 提交后又修改
- 不干净的工作目录，妨碍你切换到其他版本

干净的工作目录

- 已修改未更新
- 已更新待提交
- 提交后又修改
- 不干净的工作目录，妨碍你切换到其他版本

干净的工作目录

- 已修改未更新
- 已更新待提交
- 提交后又修改
- 不干净的工作目录，妨碍你切换到其他版本

抛弃与撤销

- 抛弃所做的修改，强制切换

```
git checkout -f dev
```

- 撤销所做的修改，留在当前分支

```
git checkout -- main.c
```

- 修改已经更新到暂存区，尚未提交

```
git reset HEAD main.c
git checkout -- main.c
```

- 两步合并成一步

```
git reset --hard HEAD main.c
```

抛弃与撤销

- 抛弃所做的修改，强制切换

```
git checkout -f dev
```

- 撤销所做的修改，留在当前分支

```
git checkout -- main.c
```

- 修改已经更新到暂存区，尚未提交

```
git reset HEAD main.c
git checkout -- main.c
```

- 两步合并成一步

```
git reset --hard HEAD main.c
```

抛弃与撤销

- 抛弃所做的修改，强制切换

```
git checkout -f dev
```

- 撤销所做的修改，留在当前分支

```
git checkout -- main.c
```

- 修改已经更新到暂存区，尚未提交

```
git reset HEAD main.c
git checkout -- main.c
```

- 两步合并成一步

```
git reset --hard HEAD main.c
```

抛弃与撤销

- 抛弃所做的修改，强制切换

```
git checkout -f dev
```

- 撤销所做的修改，留在当前分支

```
git checkout -- main.c
```

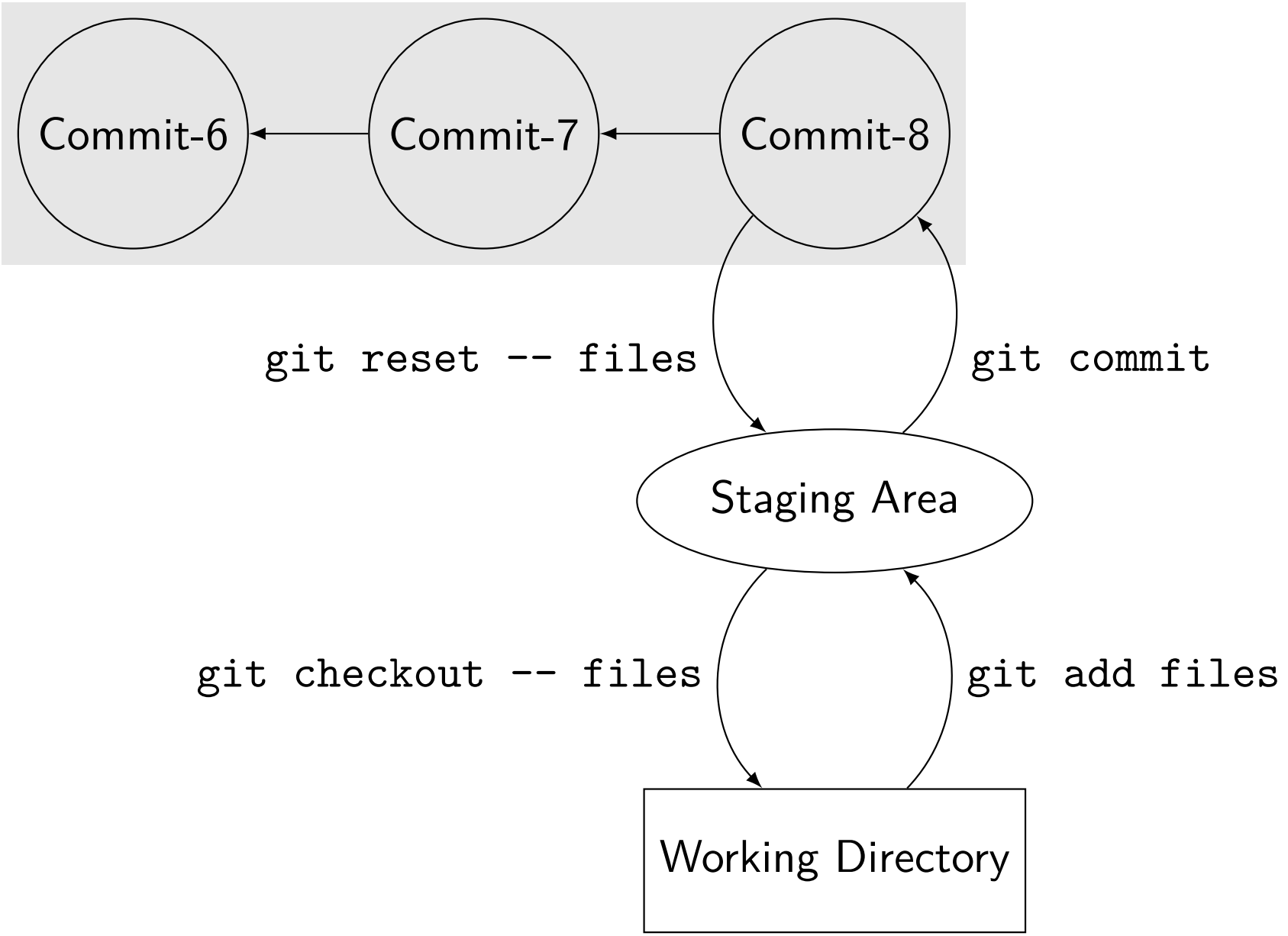
- 修改已经更新到暂存区，尚未提交

```
git reset HEAD main.c
git checkout -- main.c
```

- 两步合并成一步

```
git reset --hard HEAD main.c
```

示意图


Commit-6
Commit-7
Commit-8
git reset -- files
git commit
Staging Area
git checkout -- files
git add files
Working Directory

git checkout 操作的多义性

- 切换分支

```
git checkout branch_name
```

- 单纯改变工作目录

```
git checkout -- main.c
```

- 区别：参数是“提交名”，还是“文件名”
- 应用范例：“拎出”特定文件的历史版本到当前工作目录

```
git checkout 7fba82 main.c
git checkout fixbug~4 main.c
git checkout v1.1   main.c
```

git checkout 操作的多义性

- 切换分支

```
git checkout branch_name
```

- 单纯改变工作目录

```
git checkout -- main.c
```

- 区别：参数是“提交名”，还是“文件名”
- 应用范例：“拎出”特定文件的历史版本到当前工作目录

```
git checkout 7fba82 main.c
git checkout fixbug~4 main.c
git checkout v1.1   main.c
```

git checkout 操作的多义性

- 切换分支

```
git checkout branch_name
```

- 单纯改变工作目录

```
git checkout -- main.c
```

- 区别：参数是“提交名”，还是“文件名”
- 应用范例：“拎出”特定文件的历史版本到当前工作目录

```
git checkout 7fba82 main.c
git checkout fixbug~4 main.c
git checkout v1.1   main.c
```

git checkout 操作的多义性

- 切换分支

```
git checkout branch_name
```

- 单纯改变工作目录

```
git checkout -- main.c
```

- 区别：参数是“提交名”，还是“文件名”
- 应用范例：“拎出”特定文件的历史版本到当前工作目录

```
git checkout 7fba82 main.c
git checkout fixbug~4 main.c
git checkout v1.1   main.c
```

逆转与回退

- 逆转（回滚）

```
git revert HEAD
```

- 回退

```
git reset --hard HEAD~1
```

- 已经向外公布的提交，不应该改变其历史

逆转与回退

- 逆转（回滚）

```
git revert HEAD
```

- 回退

```
git reset --hard HEAD~1
```

- 已经向外公布的提交，不应该改变其历史

逆转与回退

- 逆转（回滚）

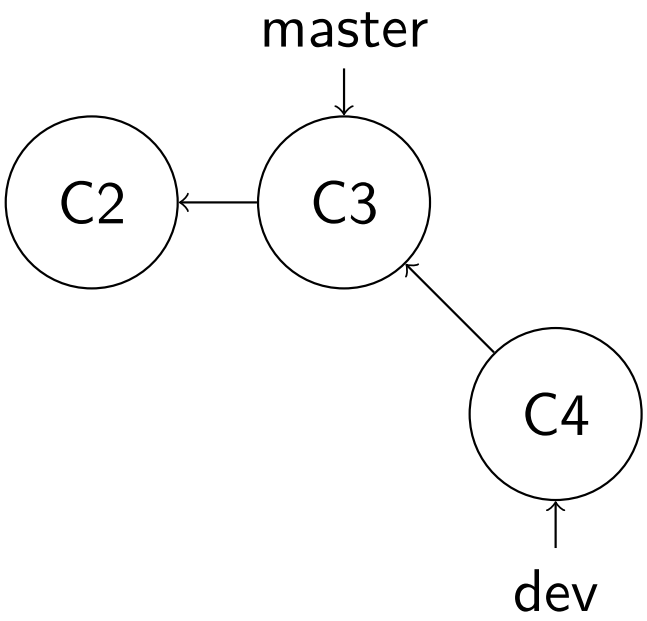
```
git revert HEAD
```

- 回退

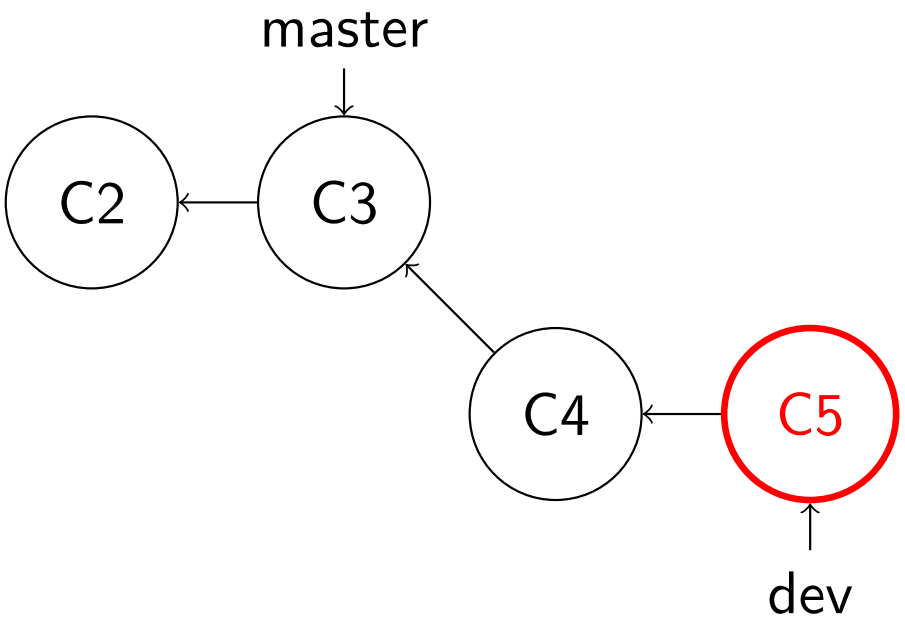
```
git reset --hard HEAD~1
```

- 已经向外公布的提交，不应该改变其历史

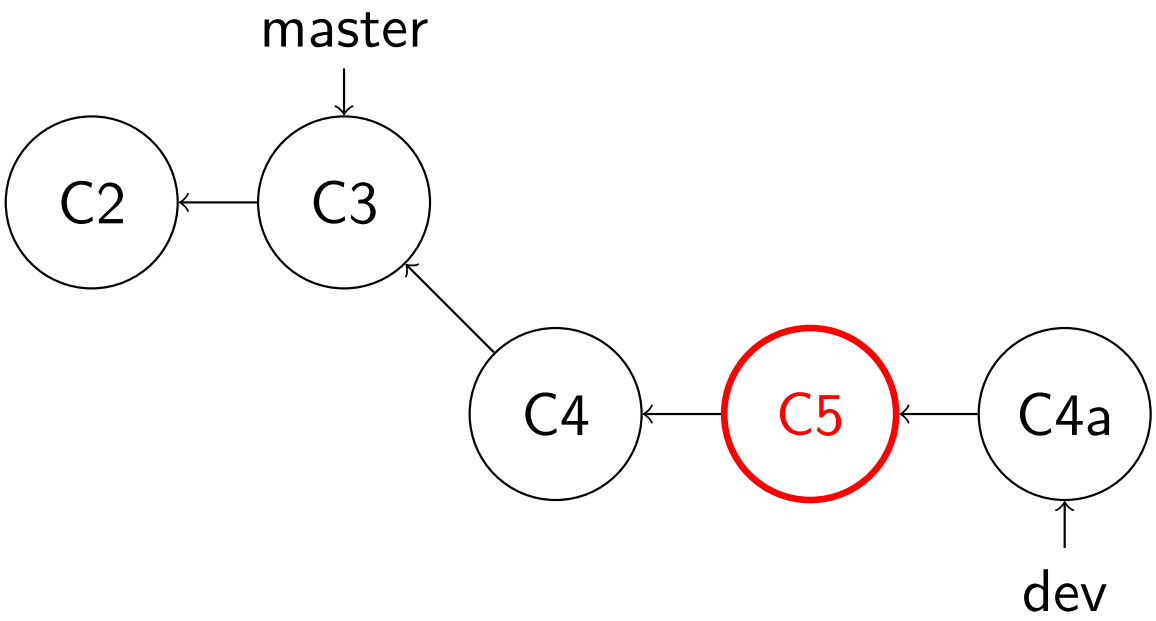
图示



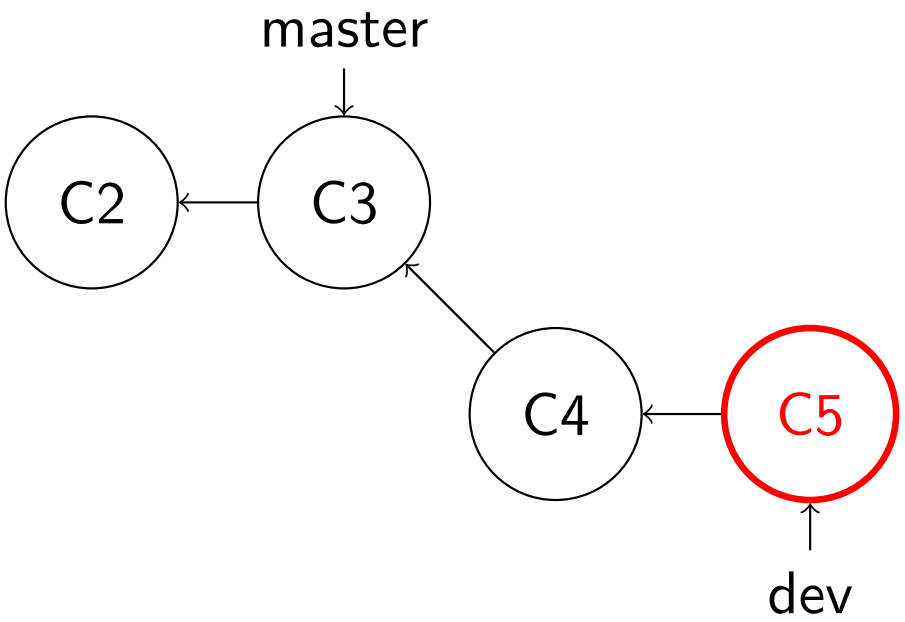
图示



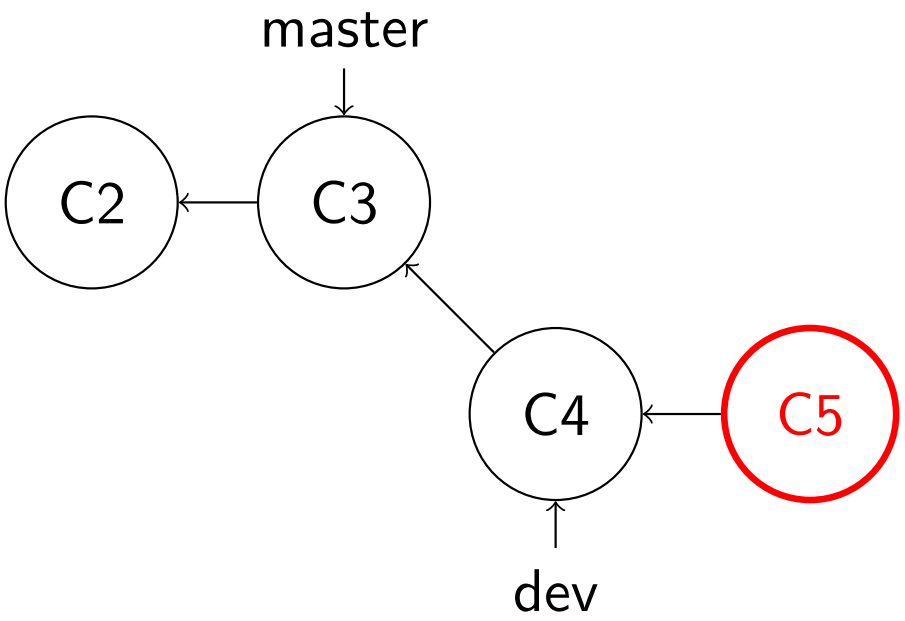
图示



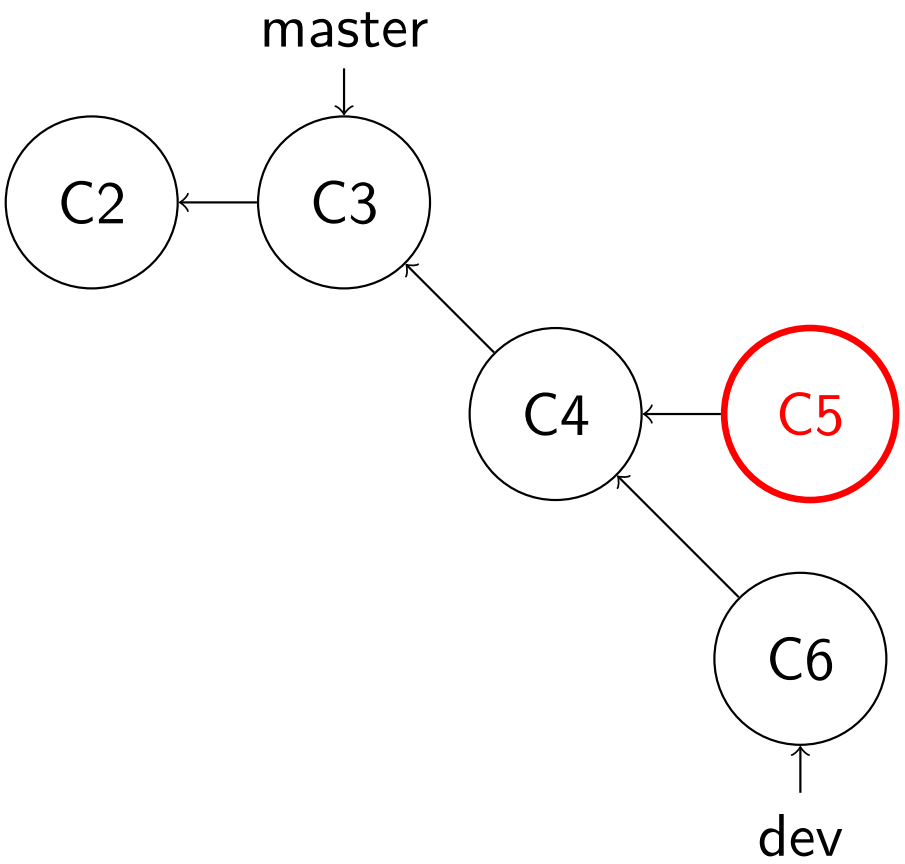
图示



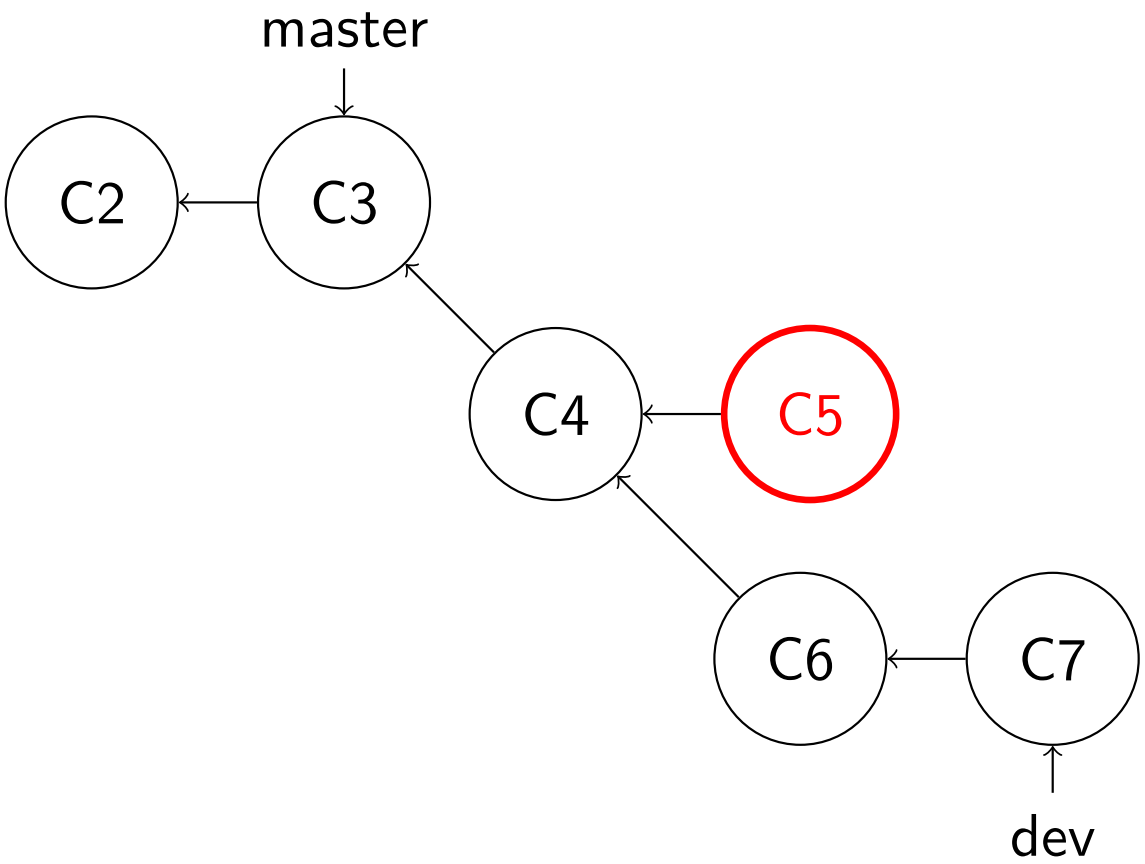
图示



图示



图示



再谈合并

- “快进型”合并，只移动分支指针
- 一般的常规合并，自动向仓库产生新的提交
- 合并有冲突时，不产生提交，等待处理

```
# Edit the file ...
git add main.c
git commit
```

冲突解决后的 git commit 操作，产生新的提交

再谈合并

- “快进型”合并，只移动分支指针
- 一般的常规合并，自动向仓库产生新的提交
- 合并有冲突时，不产生提交，等待处理

```
# Edit the file ...
git add main.c
git commit
```

冲突解决后的 git commit 操作，产生新的提交

再谈合并

- “快进型”合并，只移动分支指针
- 一般的常规合并，自动向仓库产生新的提交
- 合并有冲突时，不产生提交，等待处理

```
# Edit the file ...
git add main.c
git commit
```

冲突解决后的 git commit 操作，产生新的提交

对合并的撤销

- 合并成功，提交已自动产生，想要撤销它

```
git reset --hard ORIG_HEAD
```

- 合并有冲突，提交未产生，此刻不想解决冲突，想要撤销合并

```
git reset --hard HEAD
```

- 合并遇冲突，试图解决，但不满意，重回合并冲突时的情景

```
git checkout -m
```

对合并的撤销

- 合并成功，提交已自动产生，想要撤销它

```
git reset --hard ORIG_HEAD
```

- 合并有冲突，提交未产生，此刻不想解决冲突，想要撤销合并

```
git reset --hard HEAD
```

- 合并遇冲突，试图解决，但不满意，重回合并冲突时的情景

```
git checkout -m
```

对合并的撤销

- 合并成功，提交已自动产生，想要撤销它

```
git reset --hard ORIG_HEAD
```

- 合并有冲突，提交未产生，此刻不想解决冲突，想要撤销合并

```
git reset --hard HEAD
```

- 合并遇冲突，试图解决，但不满意，重回合并冲突时的情景

```
git checkout -m
```

Part V

远程仓库与协同工作

初始工作

- 克隆操作

```
git clone ssh://example.com/git/our-project.git
git clone git@example.com:git/our-project.git
```

- 远程仓库
- 本地仓库
  1. 工作目录（缺省：master 分支的最新本）
  2. 内含本地仓库（.git 子目录）
- Development Repository & Bare Repository

初始工作

- 克隆操作

```
git clone ssh://example.com/git/our-project.git
git clone git@example.com:git/our-project.git
```

- 远程仓库
- 本地仓库
  1. 工作目录（缺省：master 分支的最新本）
  2. 内含本地仓库（.git 子目录）
- Development Repository & Bare Repository

初始工作

- 克隆操作

```
git clone ssh://example.com/git/our-project.git
git clone git@example.com:git/our-project.git
```

- 远程仓库
- 本地仓库
  1. 工作目录（缺省： master 分支的最新本）
  2. 内含本地仓库（ .git 子目录）
- Development Repository & Bare Repository

初始工作

- 克隆操作

```
git clone ssh://example.com/git/our-project.git
git clone git@example.com:git/our-project.git
```

- 远程仓库
- 本地仓库
  1. 工作目录（缺省：master 分支的最新本）
  2. 内含本地仓库（.git 子目录）
- Development Repository & Bare Repository

初始工作

- 克隆操作

```
git clone ssh://example.com/git/our-project.git
git clone git@example.com:git/our-project.git
```

- 远程仓库
- 本地仓库
  1. 工作目录（缺省：master 分支的最新本）
  2. 内含本地仓库（ .git 子目录）
- Development Repository & Bare Repository

初始工作

- 克隆操作

```
git clone ssh://example.com/git/our-project.git
git clone git@example.com:git/our-project.git
```

- 远程仓库
- 本地仓库
  1. 工作目录（缺省：master 分支的最新本）
  2. 内含本地仓库（ .git 子目录）
- Development Repository & Bare Repository

本地仓库

- 本地仓库一旦建立，操作就针对本地仓库

```
git log
```

- 可以看到所有的提交历史，都被复制过来。
- 自动为你建立工作目录，缺省的分支是 master
- 自动为你的远程仓库，起一个名字，缺省是 origin

本地仓库

- 本地仓库一旦建立，操作就针对本地仓库

```
git log
```

- 可以看到所有的提交历史，都被复制过来。
- 自动为你建立工作目录，缺省的分支是 master
- 自动为你的远程仓库，起一个名字，缺省是 origin

本地仓库

- 本地仓库一旦建立，操作就针对本地仓库

```
git log
```

- 可以看到所有的提交历史，都被复制过来。
- 自动为你建立工作目录，缺省的分支是 master
- 自动为你的远程仓库，起一个名字，缺省是 origin

本地仓库

- 本地仓库一旦建立，操作就针对本地仓库

```
git log
```

- 可以看到所有的提交历史，都被复制过来。
- 自动为你建立工作目录，缺省的分支是 master
- 自动为你的远程仓库，起一个名字，缺省是 origin

回顾单独工作时的流程

- 基于当前分支建一个临时分支，并切换到那里工作
- 修改代码 . . . 提交
- 修改代码 . . . 提交
- 切换回工作分支（master, dev, . . .）
- 把临时分支合并过来
- 删除临时分支（可保留再用）
- . . .

回顾单独工作时的流程

- 基于当前分支建一个临时分支，并切换到那里工作
- 修改代码 . . . 提交
- 修改代码 . . . 提交
- 切换回工作分支（master, dev, . . .）
- 把临时分支合并过来
- 删除临时分支（可保留再用）
- . . .

回顾单独工作时的流程

- 基于当前分支建一个临时分支，并切换到那里工作
- 修改代码 . . . 提交
- 修改代码 . . . 提交
- 切换回工作分支（master, dev, . . .）
- 把临时分支合并过来
- 删除临时分支（可保留再用）
- . . .

回顾单独工作时的流程

- 基于当前分支建一个临时分支，并切换到那里工作
- 修改代码 . . . 提交
- 修改代码 . . . 提交
- 切换回工作分支（master, dev, . . .）
- 把临时分支合并过来
- 删除临时分支（可保留再用）
- . . .

回顾单独工作时的流程

- 基于当前分支建一个临时分支，并切换到那里工作
- 修改代码 . . . 提交
- 修改代码 . . . 提交
- 切换回工作分支（master, dev, . . .）
- 把临时分支合并过来
- 删除临时分支（可保留再用）
- . . .

回顾单独工作时的流程

- 基于当前分支建一个临时分支，并切换到那里工作
- 修改代码 . . . 提交
- 修改代码 . . . 提交
- 切换回工作分支（master, dev, . . .）
- 把临时分支合并过来
- 删除临时分支（可保留再用）
- . . .

回顾单独工作时的流程

- 基于当前分支建一个临时分支，并切换到那里工作
- 修改代码 . . . 提交
- 修改代码 . . . 提交
- 切换回工作分支（master, dev, . . .）
- 把临时分支合并过来
- 删除临时分支（可保留再用）
- . . .

本地仓库推出去

你的每一次提交，都存放在本地仓库中，别人如何知晓?

```
git push
```

将本地仓库的内容“推送”到远程仓库

自你上次和远程仓库同步以后，

远程仓库未曾受过别人的 git push 操作:
你的 git push 将成功

反之，你的推送操作将会失败

```
! [rejected]    master -> master (non-fast-forward)
```

本地仓库推出去

你的每一次提交，都存放在本地仓库中，别人如何知晓?

```
git push
```

将本地仓库的内容“推送”到远程仓库

自你上次和远程仓库同步以后，

远程仓库未曾受过别人的 git push 操作:
你的 git push 将成功

反之，你的推送操作将会失败

```
! [rejected]    master -> master (non-fast-forward)
```

本地仓库推出去

你的每一次提交，都存放在本地仓库中，别人如何知晓?

```
git push
```

将本地仓库的内容“推送”到远程仓库

自你上次和远程仓库同步以后，

远程仓库未曾受过别人的 git push 操作:
你的 git push 将成功

反之，你的推送操作将会失败

```
! [rejected]    master -> master (non-fast-forward)
```

本地仓库推出去

你的每一次提交，都存放在本地仓库中，别人如何知晓?

```
git push
```

将本地仓库的内容“推送”到远程仓库

自你上次和远程仓库同步以后，

远程仓库未曾受过别人的 git push 操作:
你的 git push 将成功

反之，你的推送操作将会失败

```
! [rejected]    master -> master (non-fast-forward)
```

本地仓库推出去

你的每一次提交，都存放在本地仓库中，别人如何知晓?

```
git push
```

将本地仓库的内容“推送”到远程仓库

自你上次和远程仓库同步以后，

远程仓库未曾受过别人的 git push 操作:
你的 git push 将成功

反之，你的推送操作将会失败

```
! [rejected]     master -> master (non-fast-forward)
```

合并远程仓库里的变更到本地

用远程仓库的内容更新本地仓库

```
git pull
```

以上操作暗中包含两个步骤

```
git fetch  (from the remote ropository)
git merge  (the tracking branch to the topic branch)
```

合并远程仓库里的变更到本地

用远程仓库的内容更新本地仓库

```
git pull
```

以上操作暗中包含两个步骤

```
git fetch  (from the remote ropository)
git merge  (the tracking branch to the topic branch)
```

合并只能在工作目录中进行

git 的逻辑是，合并永远不在远程仓库一端进行。必须先在某人的工作目录中合并，提交到本地仓库，再推送到远程。

如果本地合并一切顺利，合并之后，再次运行

```
git push
```

如果合并出现冲突，解决冲突，提交改变到本地仓库，再推到远程。

合并只能在工作目录中进行

git 的逻辑是，合并永远不在远程仓库一端进行。必须先在某人的工作目录中合并，提交到本地仓库，再推送到远程。

如果本地合并一切顺利，合并之后，再次运行

```
git push
```

如果合并出现冲突，解决冲突，提交改变到本地仓库，再推到远程。

合并只能在工作目录中进行

git 的逻辑是，合并永远不在远程仓库一端进行。必须先在某人的工作目录中合并，提交到本地仓库，再推送到远程。

如果本地合并一切顺利，合并之后，再次运行

```
git push
```

如果合并出现冲突，解决冲突，提交改变到本地仓库，再推到远程。

多个远程仓库

git 允许使用多个远程仓库。
第一次使用克隆命令时，远程仓库自动命名为 origin。
以后可以再添加、删除、换名、修改远程仓库。

```
git remote add github git://github.com/xxx/yyy.git
git remote rm github
git remote rename lib njit
git remote set-url xxlib tom@lib.xxyy.edu.cn:git/hw.git
```

Git 操作中特别指明目标仓库

Git 操作中，带上远程仓库名

```
git push github
git push github test_branch
git fetch github
git fetch github new_branch
```

本地新建的分支，必需在 git push 命令中，特别指明，才能推送到远程仓库。

```
git push origin my_new_local_branch
git push --tags
```

Git 操作中特别指明目标仓库

Git 操作中，带上远程仓库名

```
git push github
git push github test_branch
git fetch github
git fetch github new_branch
```

本地新建的分支，必需在 git push 命令中，特别指明，才能推送到远程仓库。

```
git push origin my_new_local_branch
git push --tags
```

工作分支与其跟踪分支

- 了解远程仓库的况

```
git remote
git remote show
git remote show origin
```

- 本地分支/远程分支
- 工作分支/跟踪分支

工作分支与其跟踪分支

- 了解远程仓库的况

```
git remote
git remote show
git remote show origin
```

- 本地分支/远程分支
- 工作分支/跟踪分支

工作分支与其跟踪分支

- 了解远程仓库的况

```
git remote
git remote show
git remote show origin
```

- 本地分支/远程分支
- 工作分支/跟踪分支

设定跟踪分支

- 设定跟踪分支

```
git clone    (will set up tracing branch automatically
git push -u other_remote my_branch
git branch -t new_branch github/dev
git branch --set-upstream l_br_name gitorious/master
git checkout -b sf origin/serverfix
git checkout -t origin/hack
```

- 了解分支跟踪情况

```
git remote show origin  (or with -n option)
git branch -avv
git config --list
cat .git/config
```

设定跟踪分支

- 设定跟踪分支

```
git clone    (will set up tracing branch automatically
git push -u other_remote my_branch
git branch -t new_branch github/dev
git branch --set-upstream l_br_name gitorious/master
git checkout -b sf origin/serverfix
git checkout -t origin/hack
```

- 了解分支跟踪情况

```
git remote show origin  (or with -n option)
git branch -avv
git config --list
cat .git/config
```

跟踪分支的意义

- 跟踪分支是远程仓库的本地代理
- 设定跟踪分支的目的是方便使用
- 已设定:

```
git push          (or with remote repository name)
git pull          (or with remote repository name)
```

- 未设定:

```
git push origin master
git fetch origin master
git checkout master
git merge origin/master
```

跟踪分支的意义

- 跟踪分支是远程仓库的本地代理
- 设定跟踪分支的目的是方便使用
- 已设定:

```
git push          (or with remote repository name)
git pull          (or with remote repository name)
```

- 未设定:

```
git push origin master
git fetch origin master
git checkout master
git merge origin/master
```

跟踪分支的意义

- 跟踪分支是远程仓库的本地代理
- 设定跟踪分支的目的是方便使用
- 已设定:

```
git push          (or with remote repository name)
git pull          (or with remote repository name)
```

- 未设定:

```
git push origin master
git fetch origin master
git checkout master
git merge origin/master
```

跟踪分支的意义

- 跟踪分支是远程仓库的本地代理
- 设定跟踪分支的目的是方便使用
- 已设定:

```
git push          (or with remote repository name)
git pull          (or with remote repository name)
```

- 未设定:

```
git push origin master
git fetch origin master
git checkout master
git merge origin/master
```

建立远程仓库

- 位于服务器上的远程仓库设成 Bare repository

```
git init --bare     (with or without a repos name)
```

- 依惯例仓库的名字用 .git 结尾
- 使用 ssh 方式访问是最佳选择
- 每个用户提供 ssh 公钥，放到服务器中

建立远程仓库

- 位于服务器上的远程仓库设成 Bare repository

```
git init --bare     (with or without a repos name)
```

- 依惯例仓库的名字用 .git 结尾
- 使用 ssh 方式访问是最佳选择
- 每个用户提供 ssh 公钥，放到服务器中

建立远程仓库

- 位于服务器上的远程仓库设成 Bare repository

```
git init --bare     (with or without a repos name)
```

- 依惯例仓库的名字用 .git 结尾
- 使用 ssh 方式访问是最佳选择
- 每个用户提供 ssh 公钥，放到服务器中

建立远程仓库

- 位于服务器上的远程仓库设成 Bare repository

```
git init --bare    (with or without a repos name)
```

- 依惯例仓库的名字用 .git 结尾
- 使用 ssh 方式访问是最佳选择
- 每个用户提供 ssh 公钥，放到服务器中

密钥对制作与部署

在你的电脑上，产生密钥对

```
cd ~/.ssh
# If id_rsa and id_rsa.pub have existed, backup them
#   or using id_rsa.pub as public key
ssh-keygen -t rsa -C "your_email@youremail.com"
# Give passphrase or leave it empty
# OK. Give id_rsa.pub to server administrator
```

Git 服务器设定

- 安装 ssh server
- 安装 git
- 建立一个项目用户比如 xuser，该用户的缺省 shell 设为 /usr/bin/git-shell （安全考虑）
- 以 xuser 身份建立项目的仓库 xprj.git ，确保 xuers 对该目录的读写权
- 添加授权访问用户的 ssh pulic key

```
cat id_rsa.pub >> ~xuser/.ssh/authorized_keys
```

- 在客户机上测试 ssh 服务

```
ssh -l xuser -T gitserver.example.com
```

Git 服务器设定

- 安装 ssh server
- 安装 git
- 建立一个项目用户比如 xuser，该用户的缺省 shell 设为 /usr/bin/git-shell （安全考虑）
- 以 xuser 身份建立项目的仓库 xprj.git ，确保 xuers 对该目录的读写权
- 添加授权访问用户的 ssh pulic key

```
cat id_rsa.pub >> ~xuser/.ssh/authorized_keys
```

- 在客户机上测试 ssh 服务

```
ssh -l xuser -T gitserver.example.com
```

Git 服务器设定

- 安装 ssh server
- 安装 git
- 建立一个项目用户比如 xuser，该用户的缺省 shell 设为 /usr/bin/git-shell （安全考虑）
- 以 xuser 身份建立项目的仓库 xprj.git ，确保 xuers 对该目录的读写权
- 添加授权访问用户的 ssh pulic key

```
cat id_rsa.pub >> ~xuser/.ssh/authorized_keys
```

- 在客户机上测试 ssh 服务

```
ssh -l xuser -T gitserver.example.com
```

Git 服务器设定

- 安装 ssh server
- 安装 git
- 建立一个项目用户比如 xuser，该用户的缺省 shell 设为 /usr/bin/git-shell （安全考虑）
- 以 xuser 身份建立项目的仓库 xprj.git ，确保 xuers 对该目录的读写权
- 添加授权访问用户的 ssh pulic key

```
cat id_rsa.pub >> ~xuser/.ssh/authorized_keys
```

- 在客户机上测试 ssh 服务

```
ssh -l xuser -T gitserver.example.com
```

Git 服务器设定

- 安装 ssh server
- 安装 git
- 建立一个项目用户比如 xuser，该用户的缺省 shell 设为 /usr/bin/git-shell （安全考虑）
- 以 xuser 身份建立项目的仓库 xprj.git ，确保 xuers 对该目录的读写权
- 添加授权访问用户的 ssh pulic key

```
cat id_rsa.pub >> ~xuser/.ssh/authorized_keys
```

- 在客户机上测试 ssh 服务

```
ssh -l xuser -T gitserver.example.com
```

Git 服务器设定

- 安装 ssh server
- 安装 git
- 建立一个项目用户比如 xuser，该用户的缺省 shell 设为 /usr/bin/git-shell （安全考虑）
- 以 xuser 身份建立项目的仓库 xprj.git ，确保 xuers 对该目录的读写权
- 添加授权访问用户的 ssh pulic key

```
cat id_rsa.pub >> ~xuser/.ssh/authorized_keys
```

- 在客户机上测试 ssh 服务

```
ssh -l xuser -T gitserver.example.com
```

用 ssh 访问远程 Git 仓库的 URL

```
ssh://xuser@git.example.com/path/to/xprj.git
ssh://xuser@git.example.com:305/path/to/xprj.git
xuser@git.example.com:/path/to/xprj.git
xuser@git.example.com:var/xprj.git
```

除 ssh 协议以外，Git 还支持：
本地目录路径、file、git、http、https、rsync 等访问协议。

仓库之间的主次

理论上，Git 是完全分布式的组织方式，技术上，没有哪个仓库更“正宗”。

事实上，人们会将某个仓库理解成“中心”仓库。

不同仓库中，每一个提交的时间戳，并不重要，重要的是其惟一的 sha1 名，以及它在提交历史中的位置。

仓库之间的主次

理论上，Git 是完全分布式的组织方式，技术上，没有哪个仓库更“正宗”。

事实上，人们会将某个仓库理解成“中心”仓库。

不同仓库中，每一个提交的时间戳，并不重要，重要的是其惟一的 sha1 名，以及它在提交历史中的位置。

仓库之间的主次

理论上，Git 是完全分布式的组织方式，技术上，没有哪个仓库更“正宗”。

事实上，人们会将某个仓库理解成“中心”仓库。

不同仓库中，每一个提交的时间戳，并不重要，重要的是其惟一的 sha1 名，以及它在提交历史中的位置。

Part VI

参考材料

参考材料

- https://help.github.com/
- http://gitready.com/
- http://progit.org/
- http://gitimmersion.com/

反馈

shaodongwang@njit.edu.cn