

Git 之路

- 版本控制系统
- Git 独奏
- Git 和声

什么是版本控制

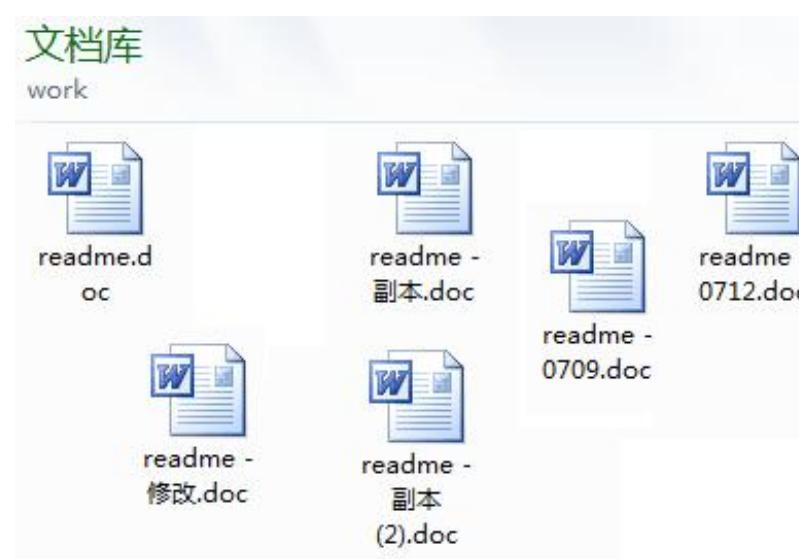
- 版本控制：对文档、源码、以及其他信息集合的变化的管理。
 - × 通常，这些变化形成了文档或源码的一个版本，版本通常用一个数字或者字母的编码来标识，被称为版本号 (vesion number 或 revision number)
- 版本控制系统 (VCS)：进行版本控制的软件，主要作用包括：
 - × 版本控制：用于备份、回滚到不同的源码或文档版本
 - × 多人协同开发：多人协同的软件开发有多种工作模式，通常会在某个服务器端保存最新的开发版本，多个开发人员使用版本控制系统可以在本地端同步版本、推送提交、合并冲突等等。
- 总之，个人开发软件时，VCS 可以帮助你维护软件的版本；团队开发软件时，VCS 不仅可以维护版本，还可以协同开发工作。

- 版本控制系统
- Git 独奏
- Git 和声

- 版本控制系统
 - ✗ 早期的版本控制
 - ✗ 版本控制系统的演进
 - ✗ Git 简介
 - ✗ 安装和初次运行时的配置

早期的版本控制

- 在没有版本控制系统的时期，许多人习惯用复制整个项目目录的方式在不同的开发主机上保存不同的版本，或许还会改名加上备份时间以示区别。有些开发团队还通过文件服务器的共享目录进行协同开发。
 - 这么做唯一的好处就是简单，不过坏处却不少，有时候会混淆所在的工作目录，弄错了文件丢了数据就没了后退的路，多人合作时常常会出现版本最终无法一致的情况。



早期的源码工具

- 在这种情况下，源码的版本控制非常容易出错，由于只是简单的复制，源码文件在开发过程中相互覆盖，非常混乱。
 - ✗ 在提交 / 查看源码的不同版本时，为了方便源码的合并和修订，软件工程师们逐渐开发出了一些源码比较和补丁工具： `diff` 和 `patch`，这些工具在今天生命力依然顽强。
 - ✗ 熟悉这些工具，对于理解和使用后续出现的版本控制系统仍然具有莫大的好处。

diff 源码比较工具

- 使用 diff 可以比较两个文本文件和目录的差异。例如，可以使用 diff 来比较下面两个文件，其中 world 是 hello 的修订版本

	hello
1	应该杜绝文章中的错别子。
2	
3	但是无论使用
4	*全拼，双拼
5	*还是五笔
6	
7	是人就有可能犯错，软件更是如此。
8	
9	犯了错，就要扣工资！
10	
11	改正的成本可能会很高。

	world
1	应该杜绝文章中的错别字。
2	
3	但是无论使用
4	*全拼，双拼
5	*还是五笔
6	
7	是人就有可能犯错，软件更是如此。
8	
9	改正的成本可能会很高。
10	
11	但是“只要眼球足够多，所有Bug都好捉”，
12	这就是开源的哲学之一。

diff 源码比较工具(续 1)

```
root ~ diff -u hello world > diff.txt
root ~ vi diff.txt
```

```
1 --- hello 2021-11-01 22:54:11.105103213 +0800
2 +++ world 2021-11-01 22:56:44.019332119 +0800
3 @@ -1,4 +1,4 @@
4 -应该杜绝文章中的错别子。
5 +应该杜绝文章中的错别字。
6
7 但是无论使用
8 *全拼，双拼
9 @@ -6,6 +6,7 @@
10
11 是人就有可能犯错，软件更是如此。
12
13 -犯了错，就要扣工资！
14 -
15 改正的成本可能会很高。
16 +
17 +但是“只要眼球足够多，所有Bug都好捉”，
18 +这就是开源的哲学之一。
```

```
~
```

```
~
```

```
~
```

```
"diff" 18L, 460B
```

diff 源码比较工具 (续 2)

- 第 1 行和第 2 行分别记录了原始文件和修改文件的文件名及时间戳。 --- 行标识的是原始文件， +++ 行标识的是修改文件。
- 在后面行中， - 行是原始文件中的行， + 行是修改文件中的行， 空格开始的行是原始文件和修改文件中相同的行。
- @@ -1,4 +1,4 @@ 是一个差异定位语句（前后分别有两个 @），标识一个差异小节。 -1,4 表示差异小节的内容包括了从原始文件第 1 行开始的 4 行； +1,4 则表示差异小节的内容包括了从修改文件第 1 行开始的 4 行。

patch 补丁工具

- 命令 patch 相当于 diff 的反向操作

有了 hello 和 diff.txt 文件，可以放心地将 world 文件删除，因为可以用下面的命令来还原 world 文件：

```
$cp hello world
```

```
$ patch world < diff.txt
```

- 也可以保留 world 和 diff.txt 文件，删除 hello 文件，使用下面的命令来还原 hello 文件：

```
$cp world hello
```

```
$patch -R hello < diff.txt
```

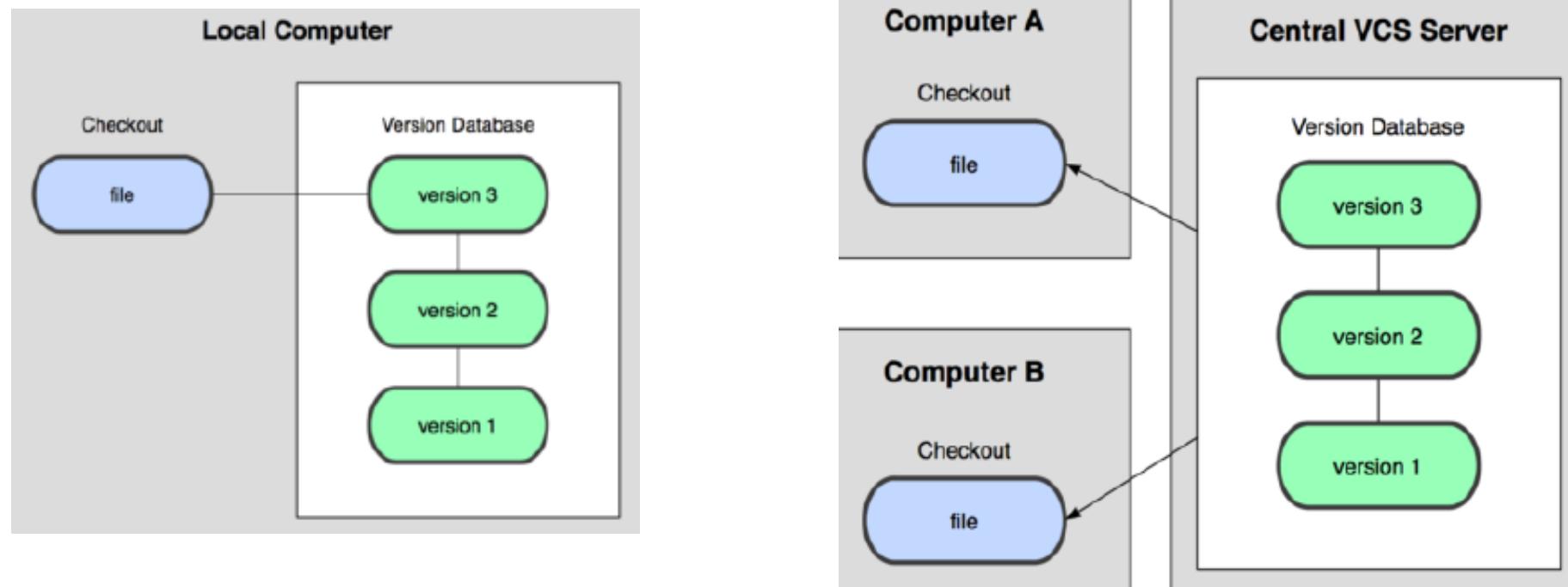
- 版本控制系统
- Git 基础
- Git 分支
- 服务器上的 Git
- 分布式 Git

- 版本控制系统

- ✗ 早期的版本控制
 - ✗ 版本控制系统的演进
 - ✗ Git 简介
 - ✗ 安装和初次运行时的配置

不同的版本控制系统

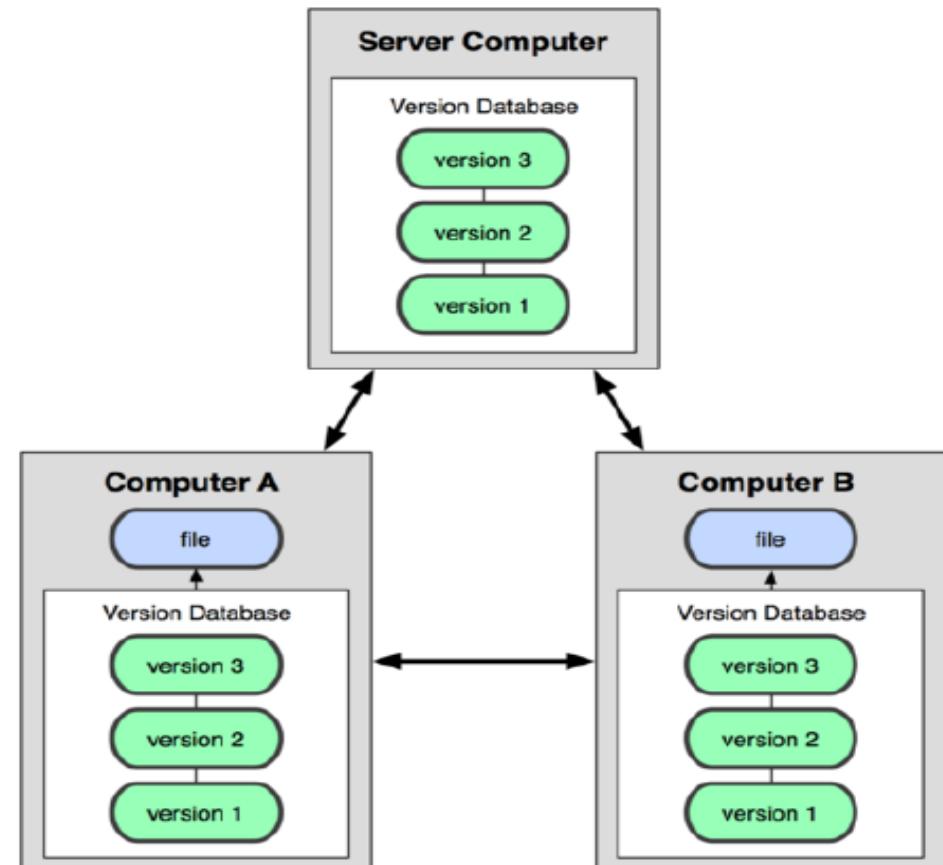
为了解决文件版本备份问题，人们很久以前就开发了许多种本地版本控制系统（左下图），大多都是采用某种简单的数据库来记录文件的历次更新差异；后来又出现了用于多人协作开发的集中式版本控制系统（右下图）。但这些系统的所有内容都严重依赖于中央数据库。



分布式版本控制系统

后来，分布式数据库的版本控制系统 (Distributed VCS) 面世了。在这类系统中，诸如 git、Mercurial、Bazaar 还有 Darcs 等，客户端并不只提取最新版本的文件快照，还将代码仓库完整地镜像下来。

这么一来，任何一处协同工作的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的提取操作，实际上都是一次对代码仓库的完整备份。其中，git 由 Linus Torvalds 于 2005 年为 Linux 内核开源社区所创建，已经成为最受欢迎的一个 DVCS。

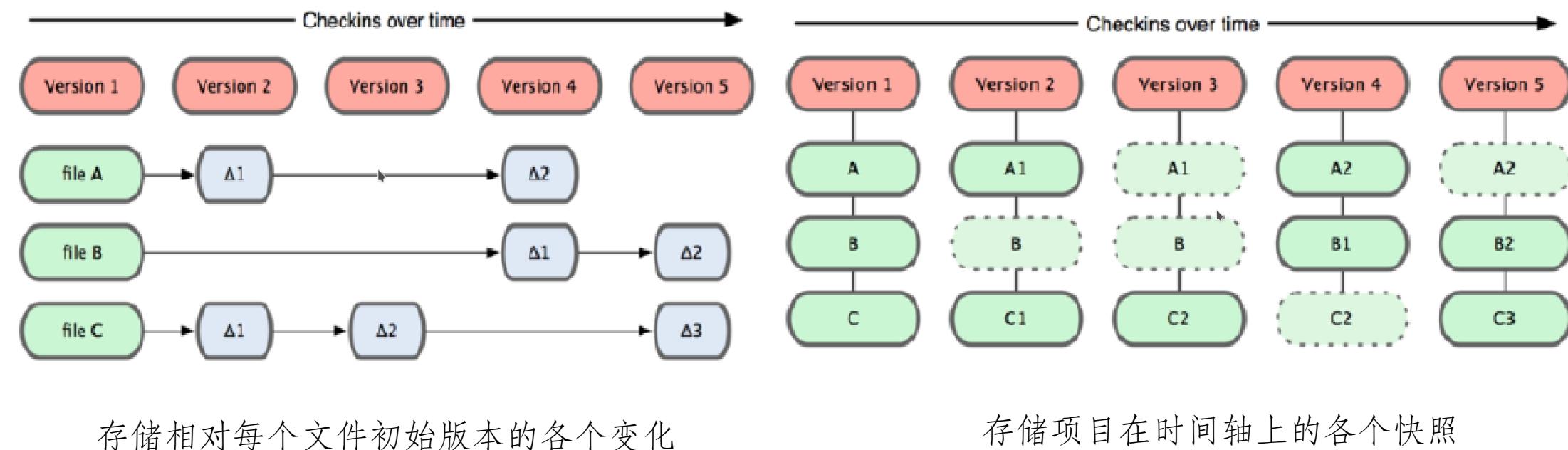


- 版本控制系统
- Git 独奏
- Git 和声

- 版本控制系统
 - ✗ 早期的版本控制
 - ✗ 版本控制系统的演进
 - ✗ Git 简介
 - ✗ 安装和初次运行时的配置

直接快照，而非比较差异

git 只关心文件数据的整体是否发生变化，而大多数其他系统则只关心文件内容的具体差异。



几乎所有操作都可本地执行

- git 中的绝大多数操作都只需要访问本地文件和资源，不用连网。但如果用 CVCS 的话，几乎所有操作都需要连接网络。因为 git 在本地磁盘上就保存着所有有关当前项目的历史更新，所以处理起来速度飞快。
- 举个例子，如果要浏览项目的历史更新摘要，git 不用跑到外面的服务器上去取数据回来，而直接从本地数据库读取后展示给你看。所以任何时候你都可以马上翻阅，无需等待。
- 如果想要看当前版本的文件和一个月前的版本之间有何差异，git 会取出一个月前的快照和当前文件作一次差异运算，而不用请求远程服务器来做这件事，或是把老版本的文件拉到本地来作比较。

时刻保持数据完整性

- 在保存到 git 之前，所有数据都要进行内容的校验和 (checksum) 计算，并将此结果作为数据的唯一标识和索引。
- 这项特性作为 git 的设计哲学，建在整体架构的最底层。所以如果文件在传输时变得不完整，或者磁盘损坏导致文件数据缺失，git 都能立即察觉。
- git 使用 SHA-1 算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个 SHA-1 哈希值 (40 个 16 进制字符) 作为指纹字符串
`24b9da6552252987aa493b52f8696cd6d3b00373`
- git 的工作完全依赖于这类指纹字串。实际上，所有保存在 git 数据库中的东西都是用此哈希值来作索引的，而不是靠文件名。

多数操作仅添加数据

- 常用的 git 操作大多仅仅是把数据添加到数据库。
- 因为任何一种不可逆的操作，比如删除数据，要回退或重现都会非常困难。在别的 VCS 中，若还未提交更新，就有可能丢失或者混淆一些修改的内容，但在 git 里，一旦提交快照之后就完全不用担心丢失数据，特别是在养成了定期推送至其他镜像仓库的习惯的话。
- 这种高可靠性令我们的开发工作安心不少，尽管去做各种试验性的尝试好了，再怎样也不会弄丢数据。

- 版本控制系统
- Git 独奏
- Git 和声

- 版本控制系统
 - ✗ 早期的版本控制
 - ✗ 版本控制系统的演进
 - ✗ Git 简介
 - ✗ 安装和初次运行时的配置

Git 源码安装

- 通过源码编译安装：若条件允许，从源代码安装有很多好处，可以安装最新的版本，因为 Git 一直在不断改进用户体验。基本步骤：
 - × 在 <http://git-scm.com/download> 下载最新的源码
 - + 可以通过 Git 克隆最新版本 `git clone https://github.com/git/git`
 - + 或者在 Web 接口下载 zip 压缩包：<https://github.com/git/git>
 - × 然后直接进入或者解压后进入 git 源码目录，阅读 `INSTALL` 文件进行编译安装
 - × 配置自动补全：
<https://github.com/markgandolfo/git-bash-completion>
 - × 配置命令行提示：
<http://volnitsky.com/project/git-prompt/>

配置 git-prompt

Install

Download `git-prompt.sh` or get it with GIT:

```
git clone git://github.com/lvv/git-prompt.git
```

Put following command at the end of your profile (`~/.bash_profile` or `~/.profile`)

```
[[ $- == *i* ]] && . /path/to/git-prompt.sh
```

There might be your old prompt defined too. You can comment it out. Some distros also have
`/etc/bashrc` or `/etc/bash/bashrc` with distro default prompt.

GIT config

GIT-PROMPT requires following GIT's option to be set:

```
git config [--global] core.quotepath off
git config [--global] --unset svn.pathnameencoding
git config [--global] --unset i18n.logoutputencoding
```

GIT-PROMPT config

Is optional. If config file is not found then git-prompt uses defaults. Defaults are listed in example

`git-prompt.conf`. Git-prompt looks (in listed order) for config file in following locations:

- `/etc/git-prompt.conf`
- `~/.git-prompt.conf`

Copy example config `git-prompt.conf` to any of above locations and customize as needed.

配置 git Completion

Git Completion 是 Git 的一个自动补全脚本

- 下载 `git-completion.bash`
- 将 `git-completion.bash` 复制到 `/etc/bash_completion.d/`。这将会自动运行。如果没有自动运行，可以在你的 `.bashrc` 文件中添加以下这些行：`. /etc/bash_completion.d/git-completion.bash`

Git 的软件源安装

- 如果要安装预编译好的 Git 二进制安装包，可以直接用系统提供的包管理工具。在 Fedora 和 mageia 上用 yum 和 urpmi 安装：

```
$ yum install git-core
```

```
#urpmi git
```

- 在 Ubuntu 这类 Debian 体系的系统上，可以用 apt-get 安装：

```
$ apt-get install git
```

- Manjaro / Arch linux:

```
#pacman -S git
```

Git 工作环境配置

- 在系统上开始使用 Git 时，需要先配置 Git 的环境。Git 带了一个称为 `git config` 的工具，用于配置或读取相应的配置变量。这些配置变量控制了 Git 的状态和操作，通常存放在三个地方：
 - × `/etc/gitconfig` 文件：系统中对所有用户都普遍适用的配置。若使用 `git config` 时用 `--system` 选项，读写的就是这个文件。
 - × `~/.gitconfig` 文件：用户目录下的配置文件只适用于该用户。若使用 `git config` 时用 `--global` 选项，读写的就是这个文件。
 - × 在当前使用仓库的 Git 目录中的配置文件（即 `.git/config` 文件）：这里的配置仅仅针对当前项目有效。
- 每一个级别的配置都会覆盖上层的相同配置，所以 `.git/config` 中的配置会覆盖 `/etc/gitconfig` 中的同名变量。

你的标识

使用 Git 首先要配置用户名和 email，这是每次提交都会引用的信息：

- `$ git config --global user.name "stardust"`
- `$ git config --global user.email "open-src@qq.com"`

查看配置：

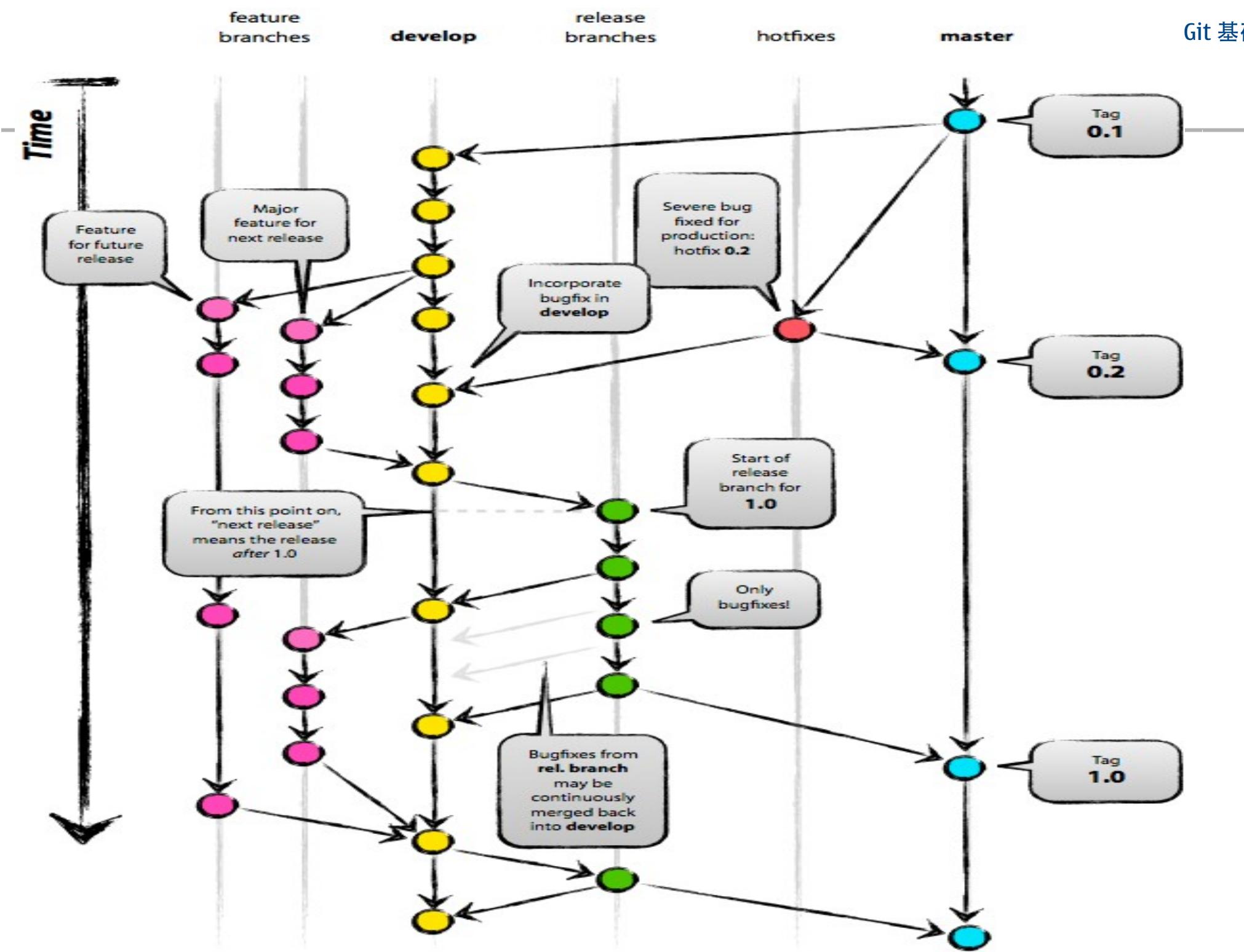
- `$ git config --list` # 列出当前所有的用户配置

查看 Git 各个工具的帮助，有三种方法：

- `$ git help <verb>`
- `$ git <verb> --help`
- `$ man git-<verb>`

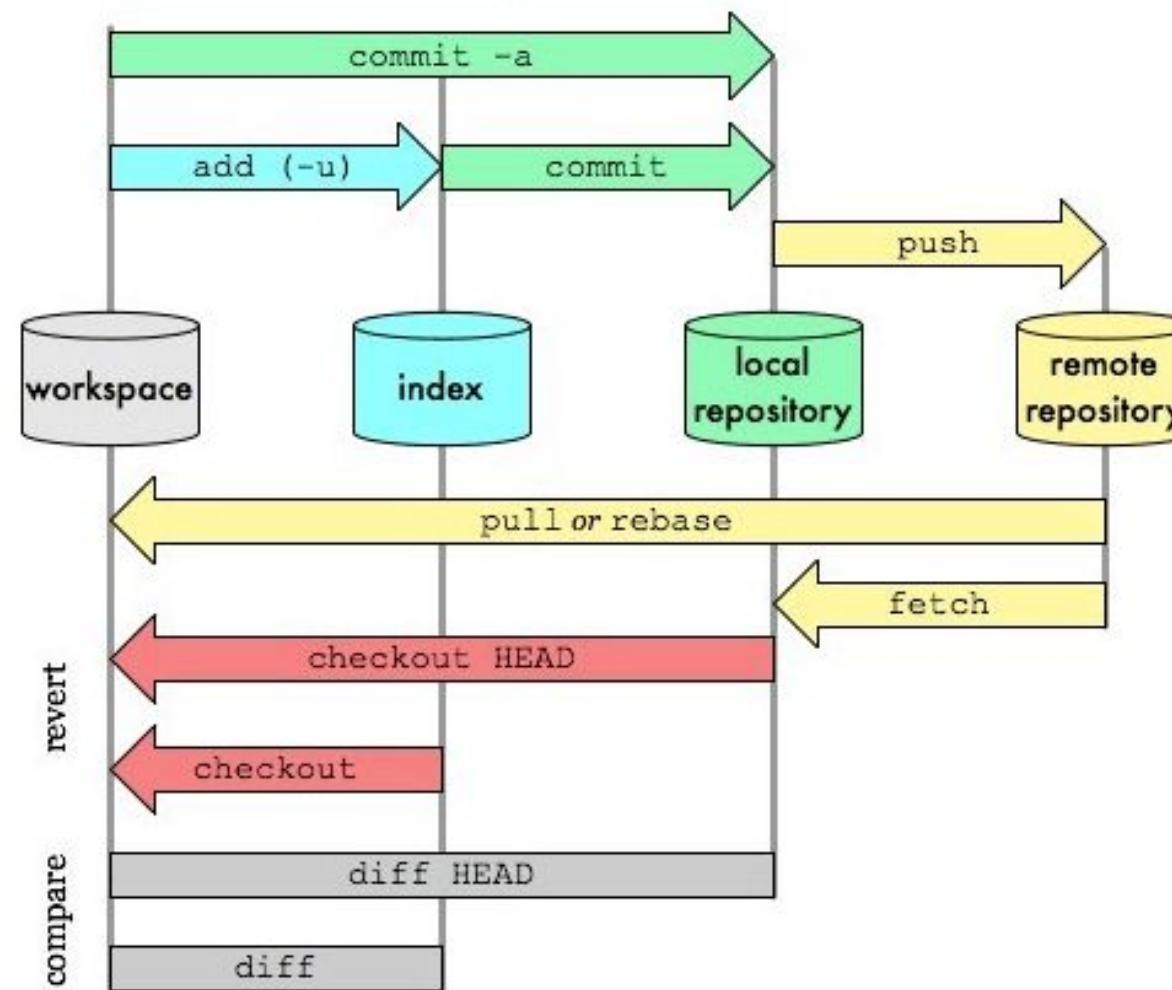
Git 之路

- 版本控制系统
- Git 独奏
- Git 和声



git 工作的模型

- 下面是 Git 工作的完整模型，但现在我们只需要首先关注本地单独使用 git 的情形，即除去使用下面的远程仓库的情形。



- 版本控制系统
- Git 独奏
- Git 和声

- Git 独奏
 - ✗ 三个区域与对象模型
 - ✗ 集结、提交
 - ✗ 重置、签出
 - ✗ Git 分支

使用 Git 管理项目

- 一个软件项目包括：源代码、开发文档、测试数据等等制品，我们通常将一个项目的所有制品存放在文件系统的某个目录下，并将该目录称为工作目录 (**workspace**)。通常，在工作目录中可以完成所有的开发工作。
- 当使用 **Git** 对项目进行管理时，需要创建该项目的仓库 **repository**，以存放项目在不同阶段的版本，提供项目的备份 / 回滚、分支 / 合并等功能。
- 为了更好地理解 **Git**，建议将 **shell** 环境变量 **LANGUAGE** 设置为 **en_GB**，方法是：将下面的语句添加到文件 **/etc/bashrc** 最后一行：

```
export LANGUAGE=en_GB
```

创建项目仓库

- **创建新仓库：**在项目的工作目录下执行：

\$ git init // 在当前目录下创建一个仓库，该仓库是一个隐藏目录 .git

- **克隆现有仓库：**从现有 Git 仓库克隆一个新的镜像仓库（注意是克隆 clone，而不是签出 checkout），同时在工作目录下签出最新版本

- × 获知某个仓库的 URL，通过 git clone 命令克隆一个完整仓库：

\$ git clone https://github.com/git/git.git

\$ git clone https://github.com/git/git.git git2 // 克隆 git 仓库，并重命名为 git2。

- × git 有 4 种主要传输协议：本地、git、http(s)、ssh

\$ git clone [ssh://]linus@github.com:/git/git.git // ssh 协议克隆

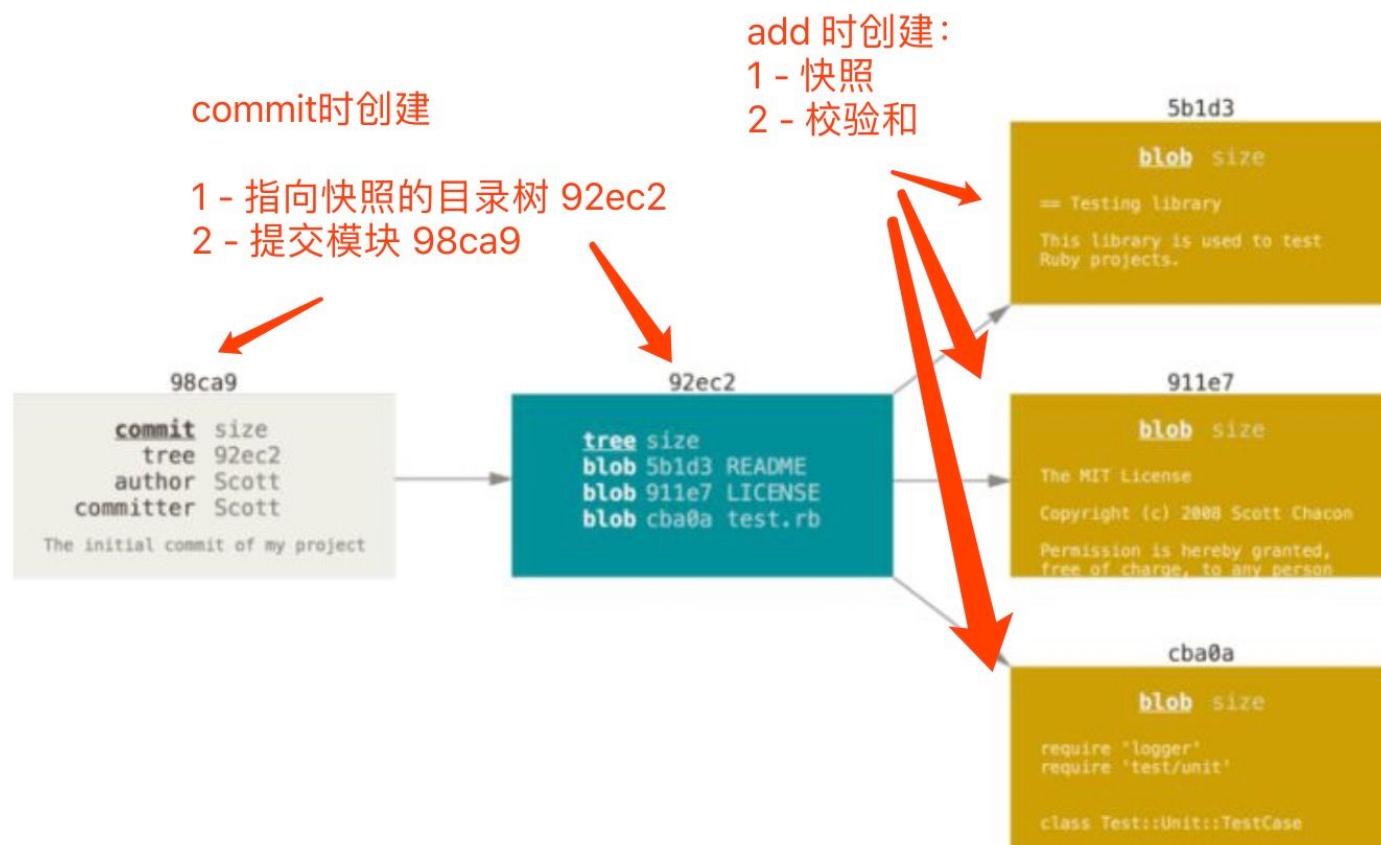
\$ git clone git://github.com/git/git.git // 使用 git 协议克隆

Git 对象

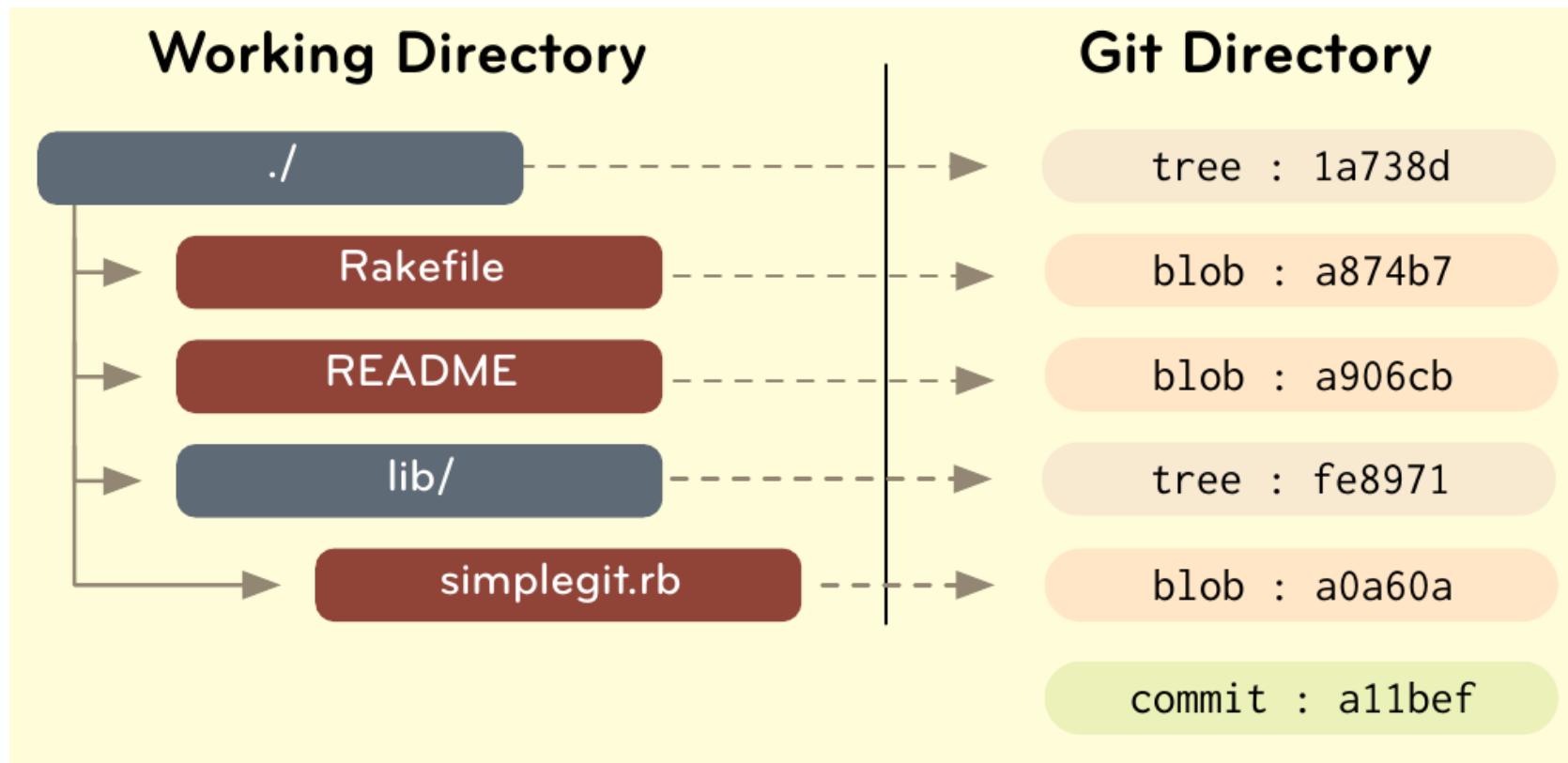
- Git 依赖于 4 类数据对象工作：:"blob"、"tree"、"commit" 和 "tag"。每个对象 (object) 包括三个部分：类型、大小和内容。大小就是指内容的大小，内容取决于对象的类型。
 - × **blob**: 是某个文件数据的快照，称为文件快照对象。blob 仅仅存储被跟踪文件不同版本的内容，并不记录文件的名字、权限等属性。stage
 - × **tree**: 即某个目录的快照。tree 对象存储了对应目录所包含的子目录 tree 对象和文件 blob 对象的指纹字符串 (索引) 等信息。commit
 - × **commit**: 包含根目录 tree 对象的索引和其他提交信息元数据，用来标记项目某一个特定时间点的状态。commit
 - × **tag**: 标签，是某个 blob/ tree/commit 对象的快捷名字。可以包含签名，最常见的是指向 commit 的 GPG 签名的标签。tag

Git 对象的存储

- 下图展示了某个项目的工作目录下有 README、LICENSE 和 test.rb 三个文件时，进行 git add/stage 和 git commit 操作时所产生的 git 对象



Git 对象的存储



- 所有的 git 对象都存储为一个文件，文件名为索引的后 38 个字符，存放在以它的前 2 个字符命名的目录中。该目录存放在对象数据库目录 .git/objects 中。

文件的状态

- 任何时候，git 项目中的文件处于以下 4 种状态中的一种：未跟踪、未修改、已修改和已集结，后面 3 种是被跟踪文件的状态。
- 当把某个文件纳入 git 管理时，将会直接对文件作快照，文件将会从 Untracked 状态变为 staged 状态。
- 当完成某次提交时，被跟踪的文件将会处于未修改状态。

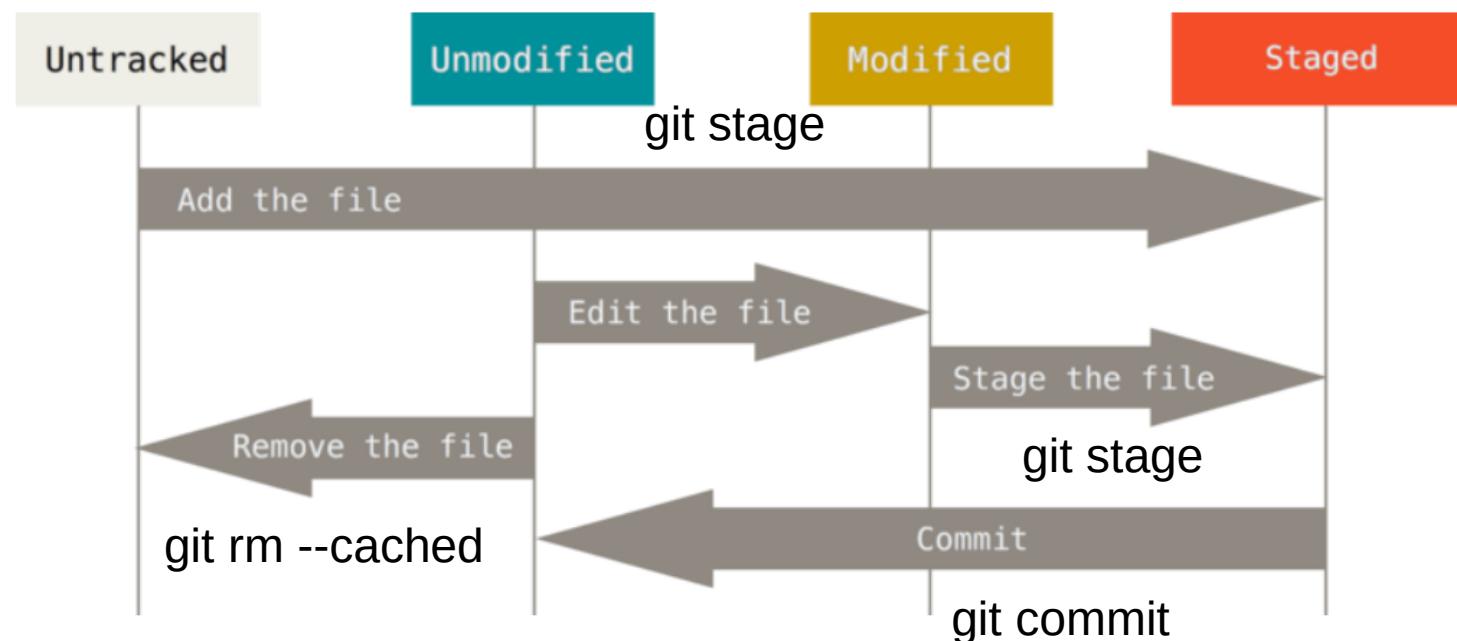
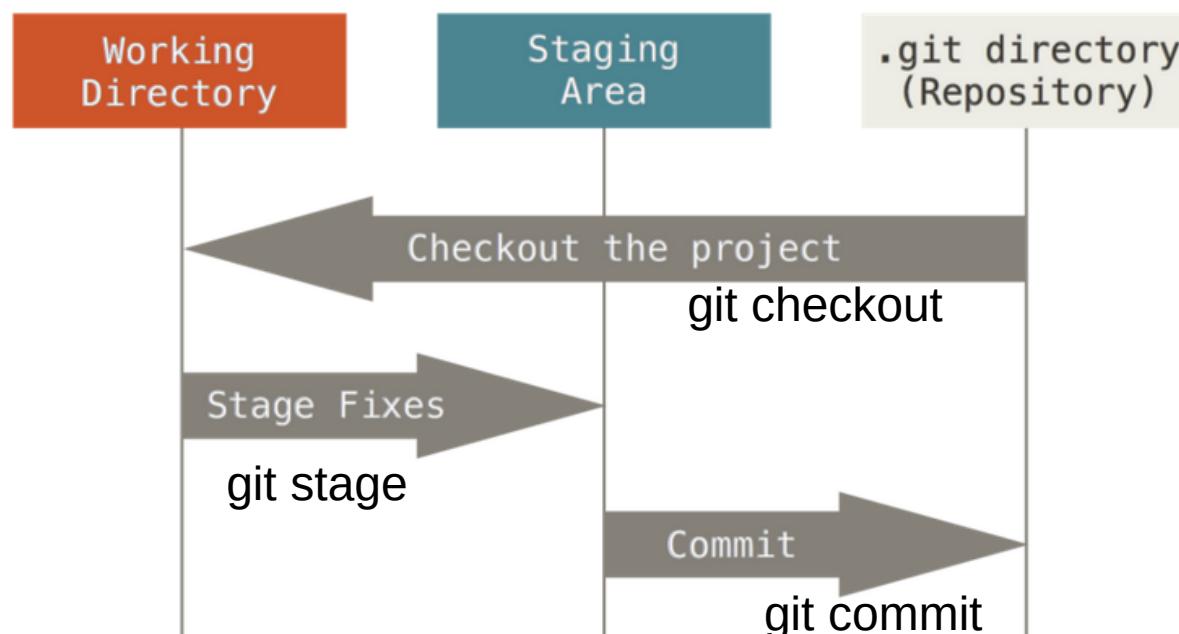


FIGURE 2-1
The lifecycle of the status of your files.

三个区域

- git 的本地工作依赖于以下三个区域：
 - × 工作目录 (workspace) 是项目根目录，里面的项目文件会被修改；
 - × staging area 是 .git 下的名为 index 的文件（集结区），该文件集结（即记录）了将被作为新提交的那些文件快照的索引；
 - × .git 目录就是仓库，将集结区的记录信息作为一个版本提交到仓库中



Working directory, staging area, and Git directory.

Git 提交版本的方式

- Git 按照“集结提交”的方式提交项目版本：“当 Git 跟踪的某个文件有变化时，将会对该文件做一个快照，然后集结这个快照的索引到 index 索引文件中；当认为集结的所有快照足以形成下一个版本时，则进行一次提交。”
- 假设某个项目 demo 包含了 foo.h、foo.cpp、bar.h、bar.cpp 等文件，并使用 Git 进行了管理。在刚提交了某个版本之后，.git 仓库中保存了：
 - × 最新的 4 个，称为 **blob 对象** 的文件快照
 - × 记录 4 个 blob 对象指纹字符串（索引）的 **tree 对象**：demo 目录的快照
 - × 记录了 tree 对象指纹字符串（索引）、提交人、作者等信息的 **commit 对象**集结区 (index 文件) 与仓库同步，保存了仓库最新 blob/tree 对象的信息
- 此时，工作区的文件、index 文件和仓库最新的 blob/tree 对象文件的时间戳相同，都处于相互同步的干净状态，可以进行下阶段的开发了。

Blob 对象

- 一个 "blob 对象" 就是一个二进制数据，是某个文件的内容在某个时候的快照，并不存储该文件的任何属性和信息。
- `git show <blob object>/git cat-file -p <blob object>` 可以来查看一个 blob 对象的内容；
- `git cat-file -t <object>` 可以用来查看对象的类型。

5b1d3..

blob	size
<pre>#ifndef REVISION_H #define REVISION_H #include "parse-options.h" #define SEEN (1u<<0) #define UNINTERESTING (1u #define TREESAME (1u<<2)</pre>	

```
$ git show 6ff87c4664
```

Note that the only valid version of the GPL as far as this project is concerned is this particular version of the license (ie v2, not v2.2 or v3.x or whatever), unless explicitly otherwise stated.

...

Tree 对象

- Tree 对象记录了指向 blob 对象或是其它 tree 对象的索引，一般用来表示内容之间的目录层次关系。
- 可以使用 git show 命令来查看一个 Tree 对象，但是 git ls-tree 能够让你看到更多的细节。

c36d4..

tree	size
blob	blob
blob	5b1d3
tree	03e78
tree	cdc8b
blob	cba0a
blob	911e7

```
$ git ls-tree fb3a8bdd0ce
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c .
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d .gitignore
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3 .mailmap
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745 COPYING
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200 Documentation
100644 blob 289b046a443c0647624607d471289b2c7dcd470b GIT-VERSION-GEN
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1 INSTALL
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52 Makefile
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52 README
...
```

Commit 对象

- Commit 对象指向一个 tree 对象，并且带有相关的描述信息。
- 可以用 `--pretty=raw` 参数配合 `git show` 或 `git log` 命令去查看某个 commit：

ae668..

commit	size
tree	c4ec5
parent	a149e
author	Scott
committer	Scott
my commit message goes here and it is really, really cool	

```
$ git show -s --pretty=raw 2be7fcb476
commit 2be7fcb4764f2dbcee52635b91fdb1b3dcf7ab4
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
parent 257a84d9d02e90447b149af58b271c19405edb6a
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700

Fix misspelling of 'suppress' in docs

signed-off-by: Junio C Hamano <gitster@pobox.com>
```

3 种对象的关联

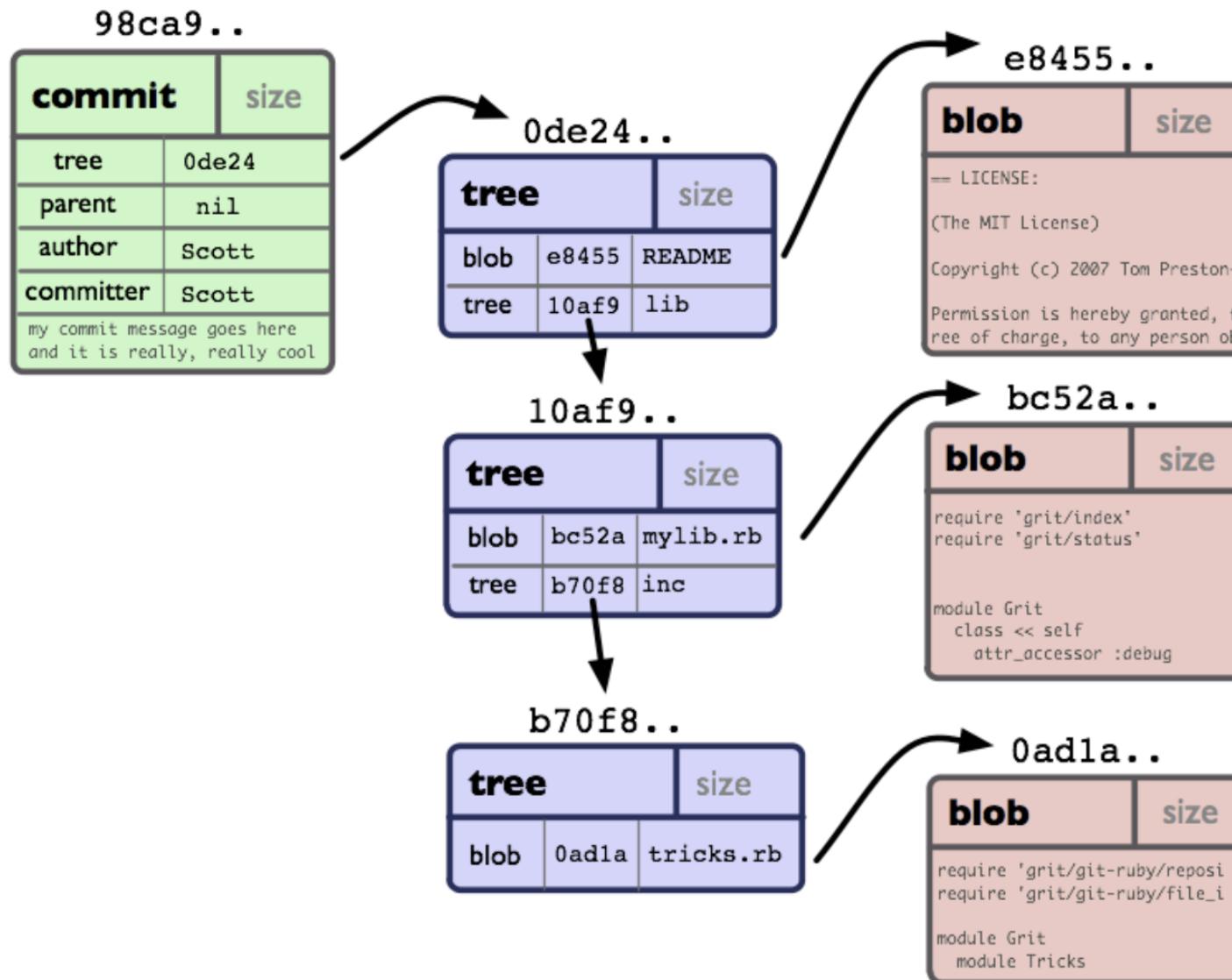
- 那么， blob 、 tree 和 commit 三种对象有什么关联呢？
- 假设有一个小项目 hellogit ，具有如下的目录结构：

```
root ➔ ~ ➔ temp ➔ hellogit ➔ tree -r  
.  
└── README  
└── lib  
    └── mylib.rb  
    └── inc  
        └── tricks.rb
```

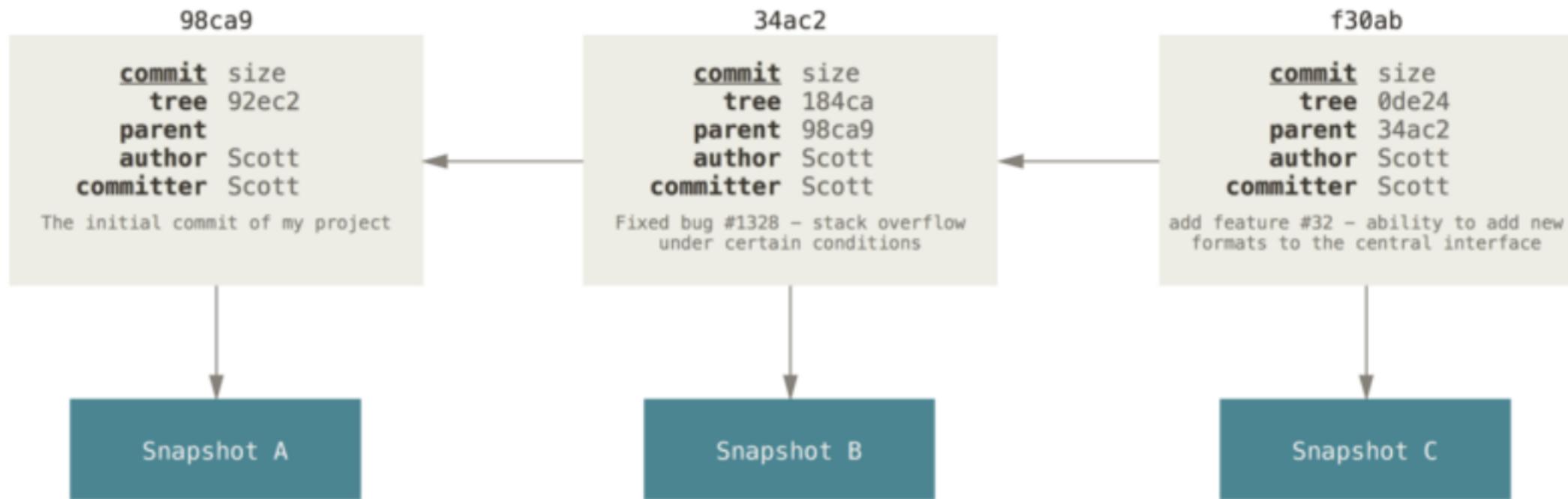
2 directories, 3 files

3 种对象的关联 (续)

- 如果把它用 Git 管理，那么这一次的提交可能就如下图所示：

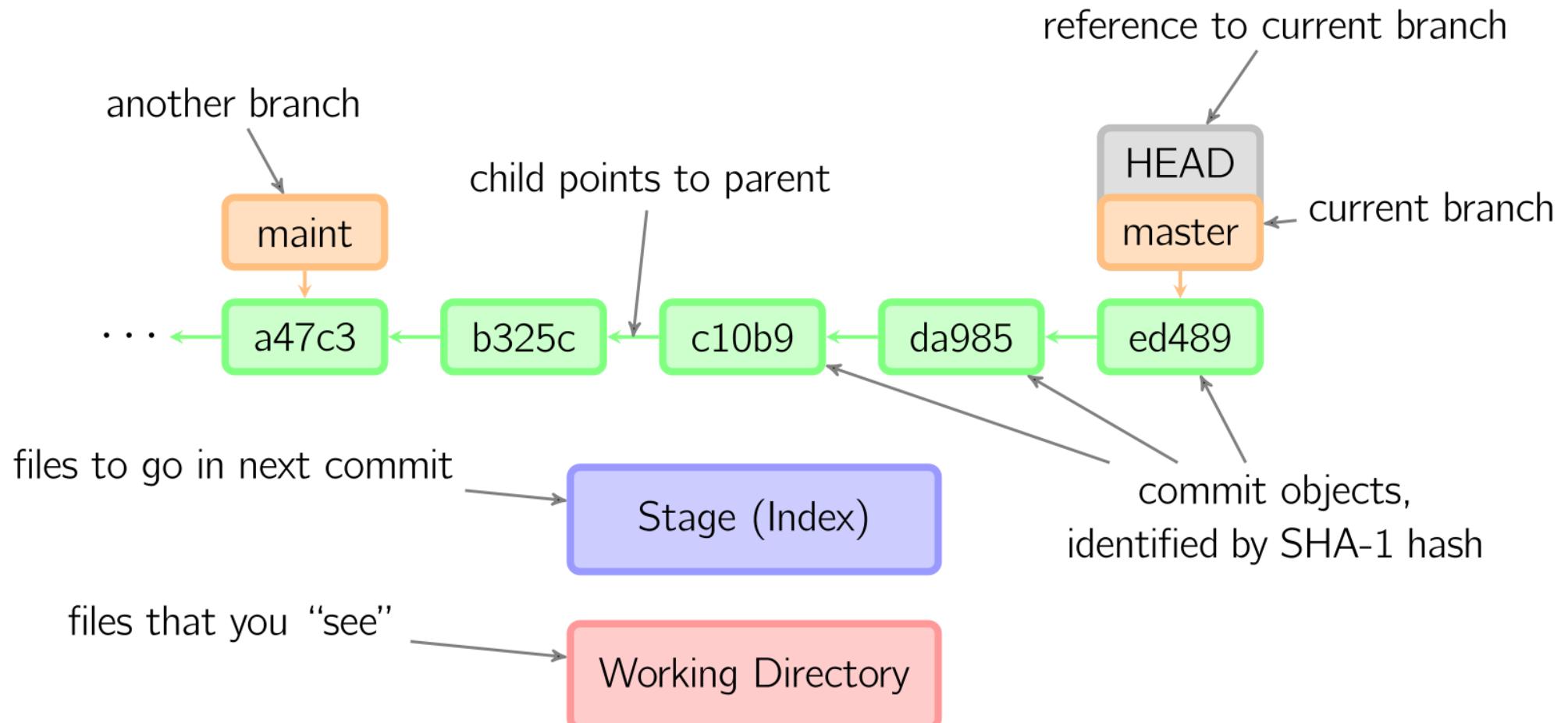


多次提交后的 Git 对象



Commits and their parents

工作目录、集结区(索引)和 Git 仓库的提交链



打标签

- 同大多数 VCS 一样，git 可以对某一时间点上的版本打上标签，主要是给某次 commit 起一个好记的名字。人们在发布某个软件版本（如 v1.0 等等）的时候，经常这么做。
- git 标签分为两种：lightweight 和 annotated。常见命令如下：

```
# git tag [-l 'v1.7.*'] # 列出所有标签 [列出所有 v1.7.* 的标签]
```

```
# git tag v5.5 5b888 // 以后可以用 v5.5 代替索引 5b888...
```

```
# git tag -a v1.4 -m 'my version 1.4' # 创建一个含附注的标签
```

```
# git show v1.4
```

```
# git push origin [tagname][--tags]
```

Tag 对象

- 由于 blob、tree 和 commit 对象都使用指纹字符串，使用时不利于记忆。为此，Git 提供了 Tag 标签对象：
 - 为上述 3 类对象取一个容易记忆的名字，方便以后引用它们
 - 通过标记某次的 commit，用于表示软件到达了某个开发版本（v1.0, v2.1, 或 α, β, RC 等版本）

49e11..

tag	size
object	ae668
type	commit
tagger	Scott
my tag message that explains this tag	

查看 Tag 对象

- 一个标签对象包括一个对象索引 (object)、标记对象类型 (type)、标签创建人 (tagger)，以及一条可能包含签名 (signature) 的消息。

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

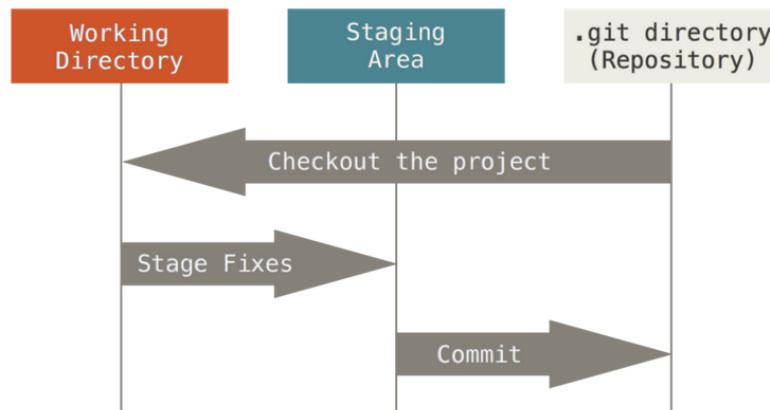
iD8DBQBFO1GqwMbZpPMRm5oRAuRiaJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----
```

- 版本控制系统
- Git 独奏
- Git 和声

- Git 独奏
 - ✗ 三个区域与对象模型
 - ✗ 集结、提交
 - ✗ 重置、签出
 - ✗ Git 分支

many stages 、 one commit

- Git 按照 “many stages 、 one commit” 的方式提交版本：“当 Git 跟踪的某个文件有变化时，可以使用 git stage 对已修改文件进行集结，集结包括两个操作：
 1. 对修改后的文件作 blob 快照；
 2. 将这个 blob 快照的索引记录到 index 索引文件中。
- 当经过多次 stage 之后，如果认为集结的所有文件快照足以形成新的版本时，则进行一次提交。提交时，将依据 index 索引文件的信息产生 tree 对象和 commit 对象，存储在仓库中。”



一次提交的场景

- 采用前面的项目 demo，假设首先修改 foo.cpp。在完成一些修改后，如果想暂时将 foo.cpp 的修改做一个备份，可以使用命令 `git stage foo.cpp` 创建该文件的快照，存放在“对象数据库”中，并在 index 中集结 (stage) 这个快照索引 (即更新该文件的快照索引)，然后继续开发。
 - × 在修改了文件之后，进行文件的 stage 之前，可以使用 `git diff` 查看工作区与集结区的文件差异；
 - × 也可以使用 `git diff HEAD` 查看工作区与仓库的最新提交的文件差异。
 - × 在 stage 文件之后，可以使用 `git diff --staged` 查看集结区与仓库最新提交的文件差异
 - × 如果你不 `stage` 这个快照的索引，可以使用 `git restore --staged foo.cpp`，删除 index 中文件快照的索引，但 .git/objects 中仍然保留了原来的文件快照。

一次提交的场景（续）

- 在继续开发 foo.cpp 时：
 - × 如果后续的修改弄乱了，想要恢复到当前集结的快照状态，可以使用 `git restore foo.cpp` (等价于 `git checkout -- foo.cpp`)。
 - × 如果后续的修改到达了一个里程碑，又想将此时的代码 stage 起来，此时仍然使用 `git stage foo.cpp` 进行新的文件快照并 **集结新的快照索引**；此时 `index` 不再记录该文件的前一个快照索引，因为此时有了新的快照版本了
- 如果 foo.cpp 修改暂时完毕，接下来交替修改 bar.cpp、include 目录下的 bar.h 文件，添加修改 main.cpp 文件
 - × 如果此时修改暂时到达一个里程碑，可以仍然使用 `git stage main.cpp bar.cpp include/bar.h` 创建这三个文件的快照 blob 对象并在 `index` 中 **集结这三个快照的索引**，同时 **记录 include 目录与 bar.h 的关系**。

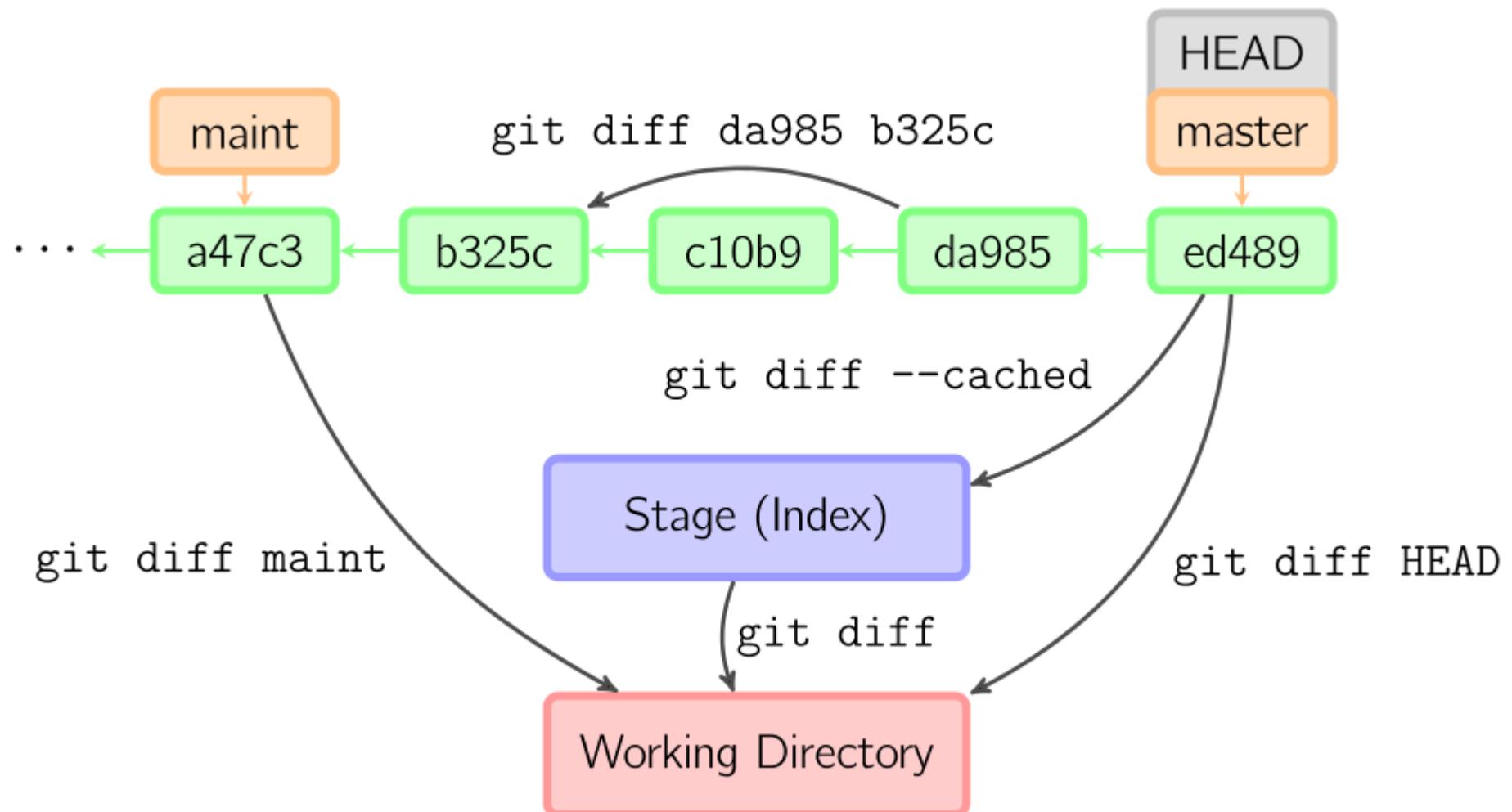
一次提交的场景（续）

- 如果所有的开发工作暂时都告一段落，准备进行一次提交。此时工作目录中，文件的变化集结完成：
 - × 被修改文件的新快照索引已被更新记录在 `index` 中；
 - × 未修改文件的快照索引仍然在 `index` 中保持不变。

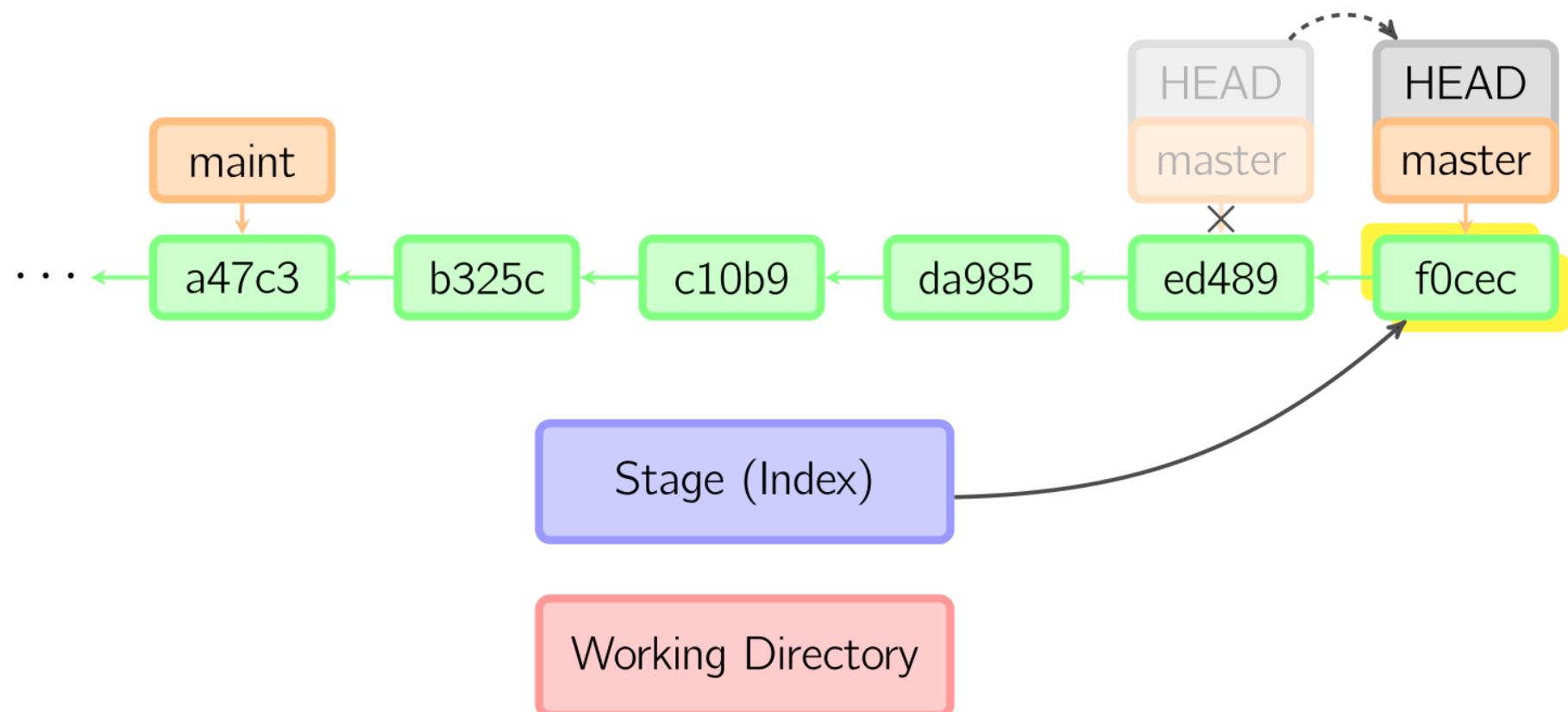
此时，`index` 文件中集结了这次提交的所有文件索引，当执行 `git commit -m "some comments"` 时：

- × 就会根据 `index` 文件中记录的目录结构关系创建若干 `tree` 对象：工作目录及其子目录的 `tree` 对象。其中，父 `tree` 对象记录了子 `tree` 对象的索引。
- × 根据提交信息和工作目录 `tree` 对象的索引生成 `commit` 对象。

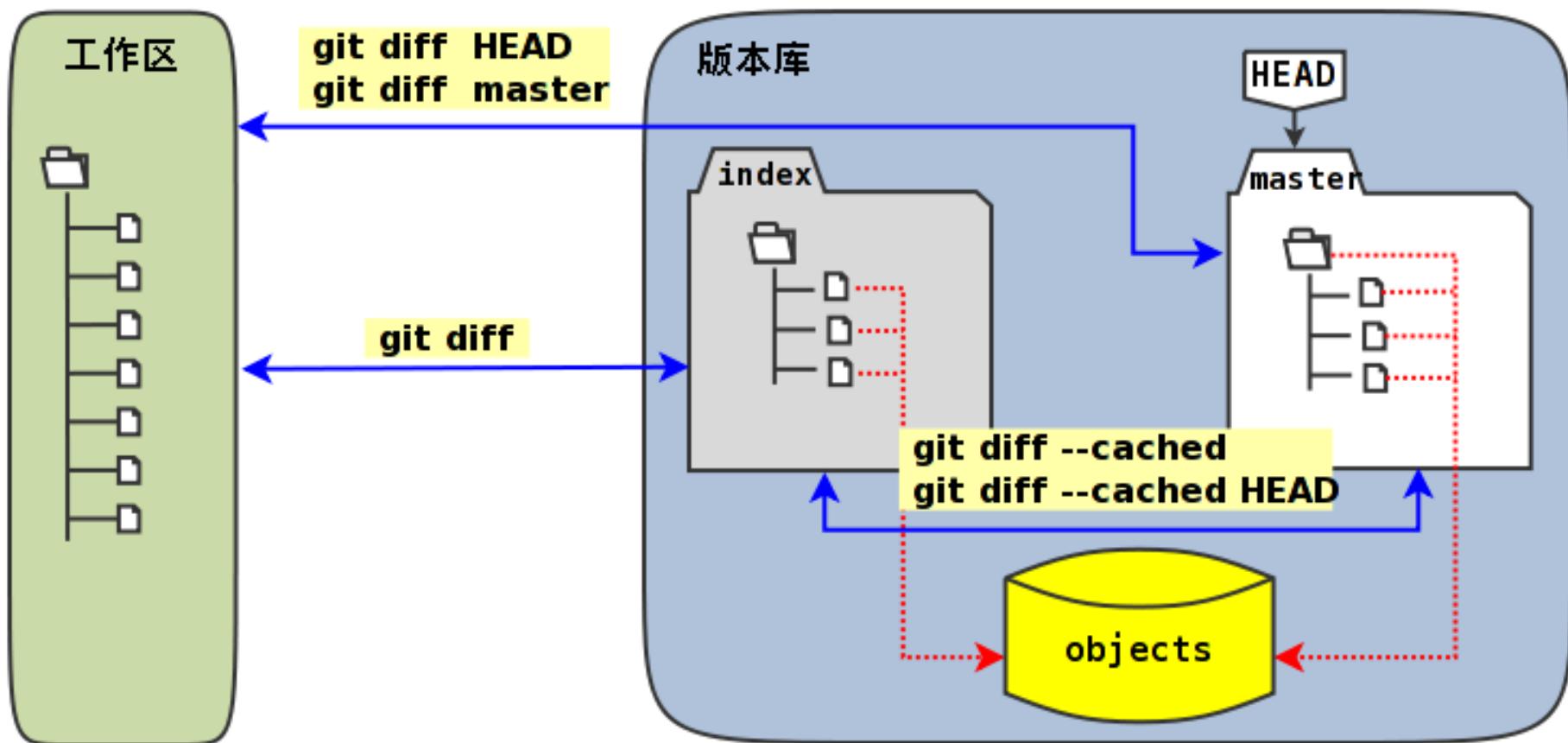
使用 git diff 进行各种差异比较图解



git commit 图解



git diff 示意图

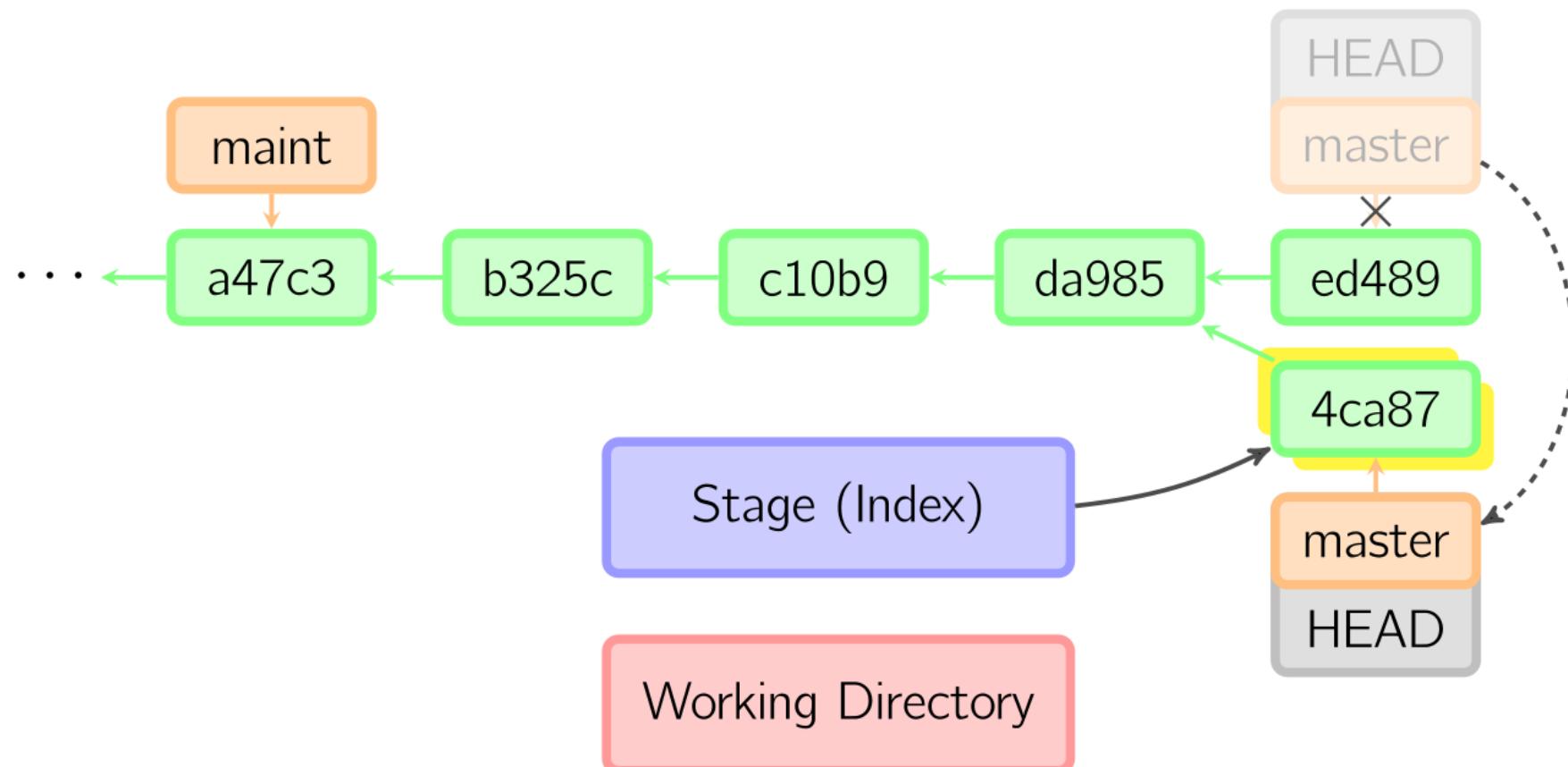


修改最新提交

- 有时候提交完了才发现漏掉了几个文件没有加，或者提交信息写错了。想要修改刚才的提交操作，可以使用 `--amend` 选项重新提交：
 - × `$ git commit -m 'initial commit'`
 - × `$ git stage forgotten_file`
 - × `$ git commit --amend`

此命令将使用当前的集结区的快照提交。如果刚才提交忘了集结某些修改，可以先补上集结操作，然后再运行 `--amend` 提交

git commit --amend 示意图



撤消最新提交、查看分支最新提交信息

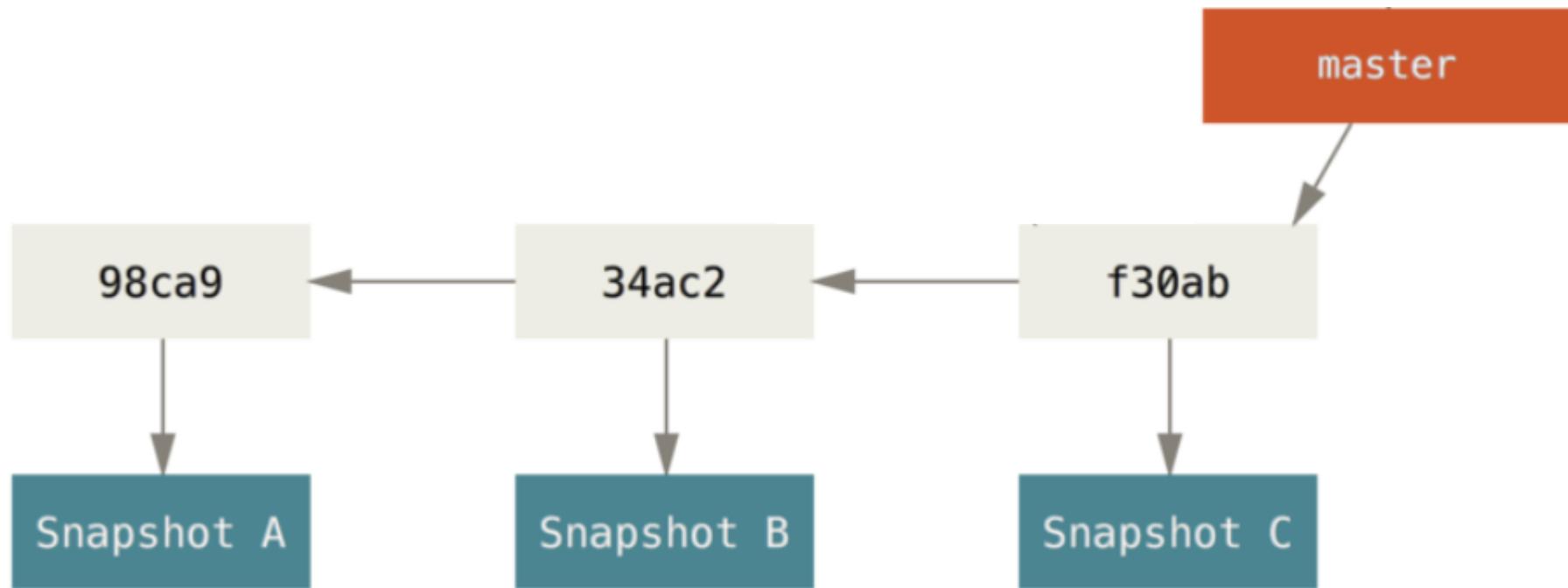
- 如果想要撤消刚才的提交操作，可以使用 git revert 命令：
 - × \$ git revert HEAD
 - 该命令会撤消最新的提交对象，你也可以撤消更早期的修改
 - × \$ git revert HEAD^
- 如果想查看所有分支最新的提交信息，可以使用 git log 命令：
 - × git log --graph --all //--all 显示所有分支， --graph 显示提交链

- 版本控制系统
- Git 独奏
- Git 和声

- Git 独奏
 - ✗ 三个区域与对象模型
 - ✗ 集结、提交
 - ✗ 重置、签出
 - ✗ Git 分支

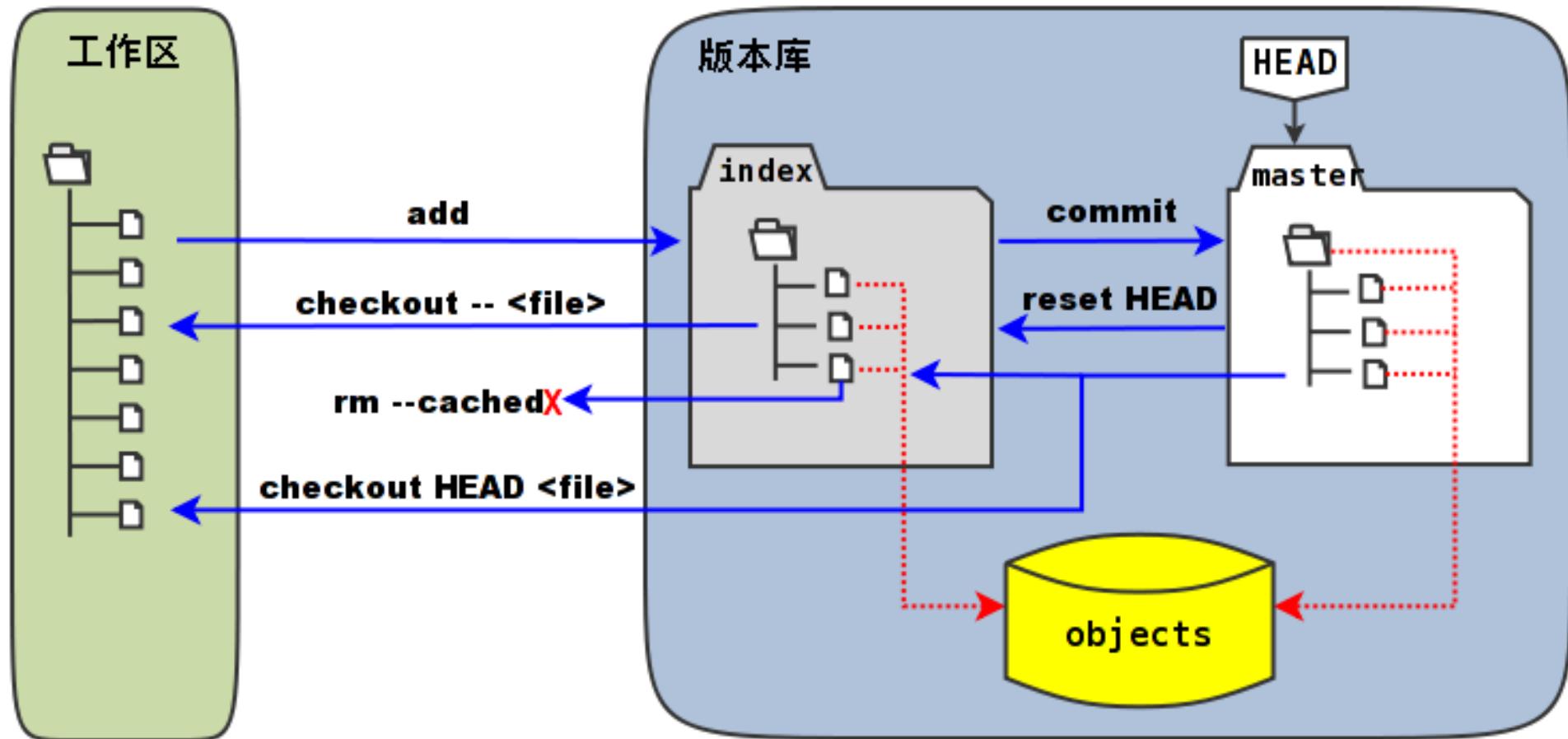
master 分支

- 当进行了多次提交时，.git 仓库中会产生如下的 Git 对象结构。



- 可以看出，必须跟踪最新的提交对象，我们通过在一个文件记录最新 commit 对象的索引来达到这个目的，默认情况下该文件为 .git/refs/heads/master，也就是通常所说的 master 分支文件。

Git 仓库的结构



几个术语

object name

The unique identifier of an [object](#). The object name is usually represented by a 40 character hexadecimal string. Also colloquially called [SHA-1](#).

ref

A name that begins with `refs/` (e.g. `refs/heads/master`) that points to an [object name](#) or another ref (the latter is called a [symbolic ref](#)). For convenience, a ref can sometimes be abbreviated when used as an argument to a Git command; see [gitrevisions\[7\]](#) for details. Refs are stored in the [repository](#).

The ref namespace is hierarchical. Different subhierarchies are used for different purposes (e.g. the `refs/heads/` hierarchy is used to represent local branches).

There are a few special-purpose refs that do not begin with `refs/`. The most notable example is `HEAD`.

几个术语

commit-ish (also committish)

A [commit object](#) or an [object](#) that can be recursively dereferenced to a commit object. The following are all commit-ishes: a commit object, a [tag object](#) that points to a commit object, a tag object that points to a tag object that points to a commit object, etc.

tree-ish (also treeish)

A [tree object](#) or an [object](#) that can be recursively dereferenced to a tree object. Dereferencing a [commit object](#) yields the tree object corresponding to the [revision's top directory](#). The following are all tree-ishes: a [commit-ish](#), a tree object, a [tag object](#) that points to a tree object, a tag object that points to a tag object that points to a tree object, etc.

HEAD

The current [branch](#). In more detail: Your [working tree](#) is normally derived from the state of the tree referred to by HEAD. HEAD is a reference to one of the [heads](#) in your repository, except when using a [detached HEAD](#), in which case it directly references an arbitrary commit.

几个术语

pathspec

Pattern used to limit paths in Git commands.

Pathsspecs are used on the command line of "git ls-files", "git ls-tree", "git add", "git grep", "git diff", "git checkout", and many other commands to limit the scope of operations to some subset of the tree or worktree. See the documentation of each command for whether paths are relative to the current directory or toplevel. The pathspec syntax is as follows:

- any path matches itself
- the pathspec up to the last slash represents a directory prefix. The scope of that pathspec is limited to that subtree.
- the rest of the pathspec is a pattern for the remainder of the pathname. Paths relative to the directory prefix will be matched against that pattern using fnmatch(3); in particular, ***** and **?** can match directory separators.

提交的重置

- Git 默认创建第一个开发分支是 master， master 分支实际上是一个文件 .git/refs/heads/master 表示，其内容就是该分支中最新提交 commit 对象的索引 ID。

```
root ➤ ~ ➤ temp ➤ hellogit ➤ cat .git/refs/heads/master  
a8bb41bf0629397921c842ed1155b985c45a3b48
```

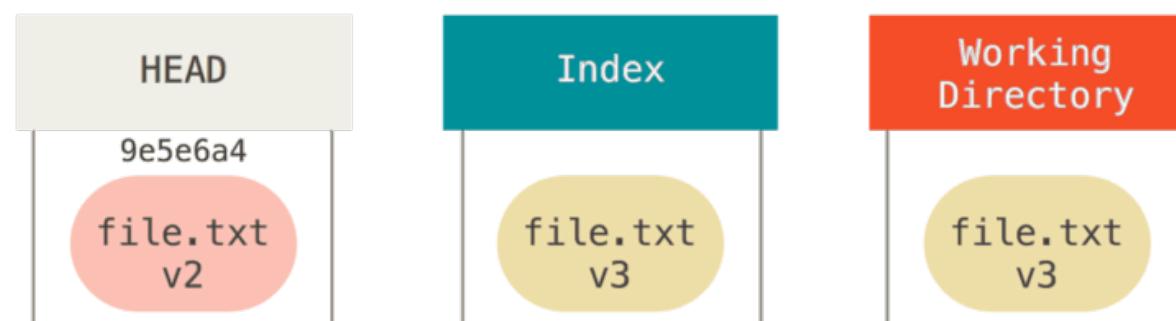
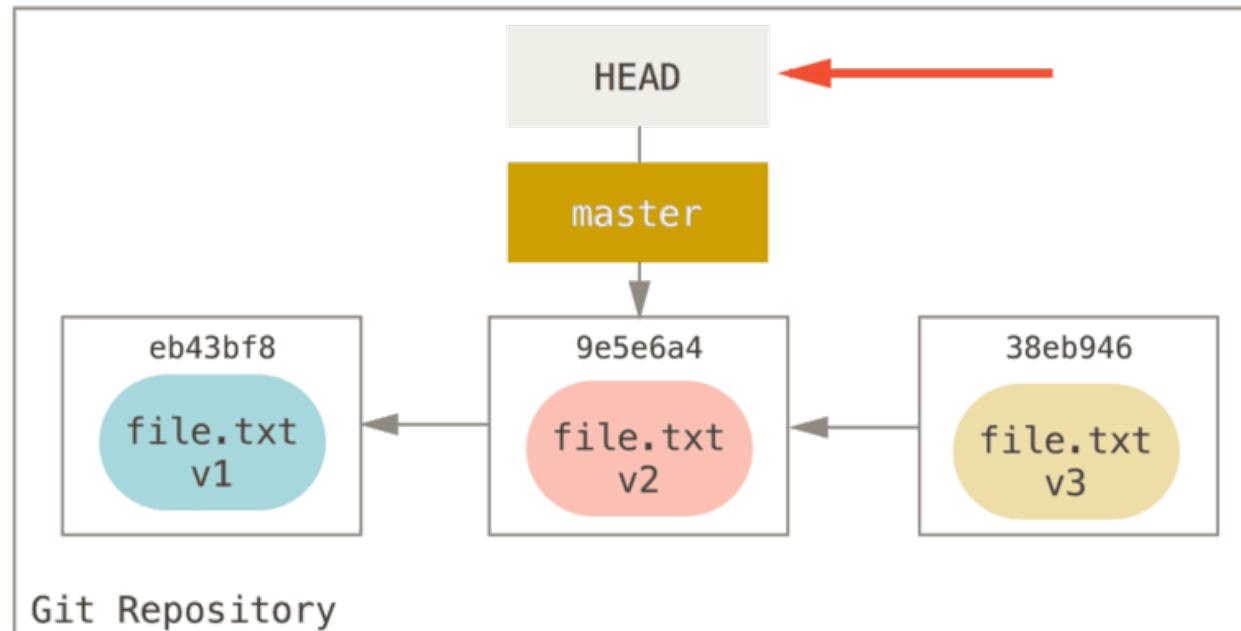
- 在进行新的 commit 的时候， master 文件的内容就会自动变化以跟踪最新的 commit 对象。
 - × 但是，分支文件 master 的一个重要的作用是可以修改它，让它指向以前的 commit 节点，从而实现该分支的提交的重置。这种行为是通过 git reset 重置命令来实现的。

git reset 的用法

```
git reset [-q] [<tree-ish>] [--] <pathspec>...
git reset [-q] [--pathspec-from-file=<file> [--pathspec-file-nul]] [<tree-ish>]
git reset (--patch | -p) [<tree-ish>] [--] [<pathspec>...]
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]
```

- 前 3 种形式用于重置集结区，不会重置分支文件 master，不会改变工作区，仅仅用于重置集结区中某个匹配 `<pathspec>` 的所有路径的项到这些项在 `<tree-ish>` 的状态。
 - × 例如： `git reset HEAD -- <pathspec>` 用最新提交的版本撤消之前执行 `git stage <pathspec>` 时进行的集结。“`include/foo.h`”
- 第 4 种是重置分支提交的用法，将分支指针（如 master）重新设置为提交对象链上的某个提交对象，并根据不同的选项，对集结区进行重置。
 - × 这种用法在单个分支的提交链上纵向操作分支游标（如 master），但是不会修改当前分支的头指针 HEAD。

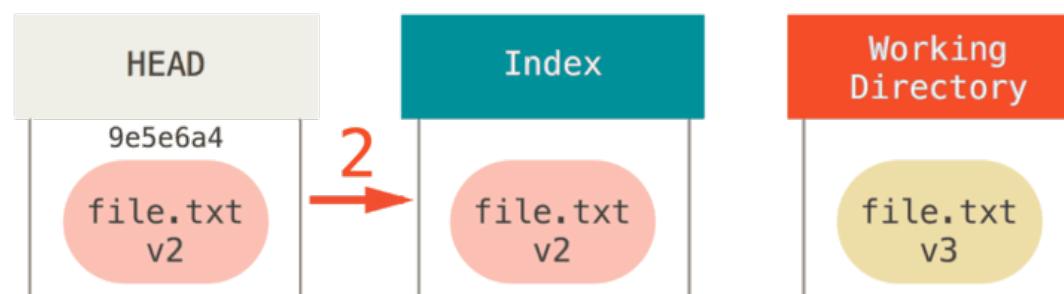
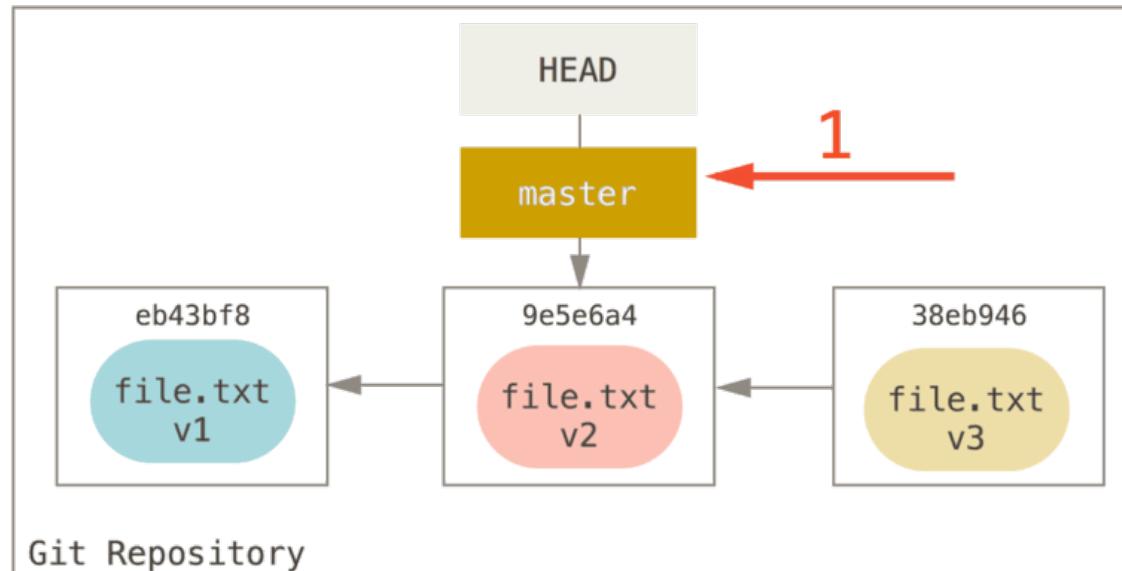
重置图解 1



`git reset --soft HEAD~`

本质上是撤销上一次的 git commit

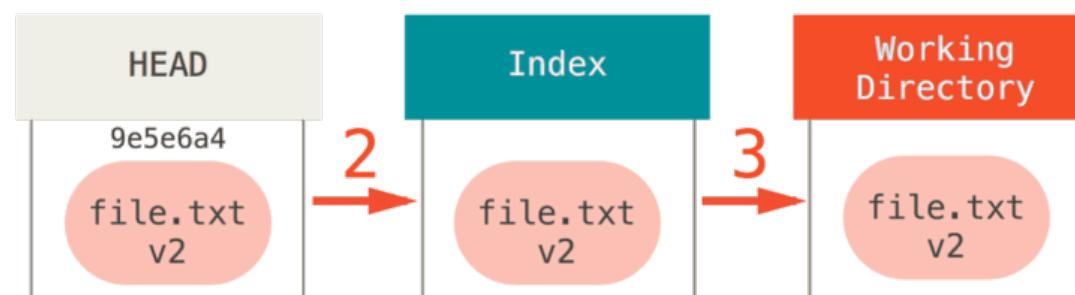
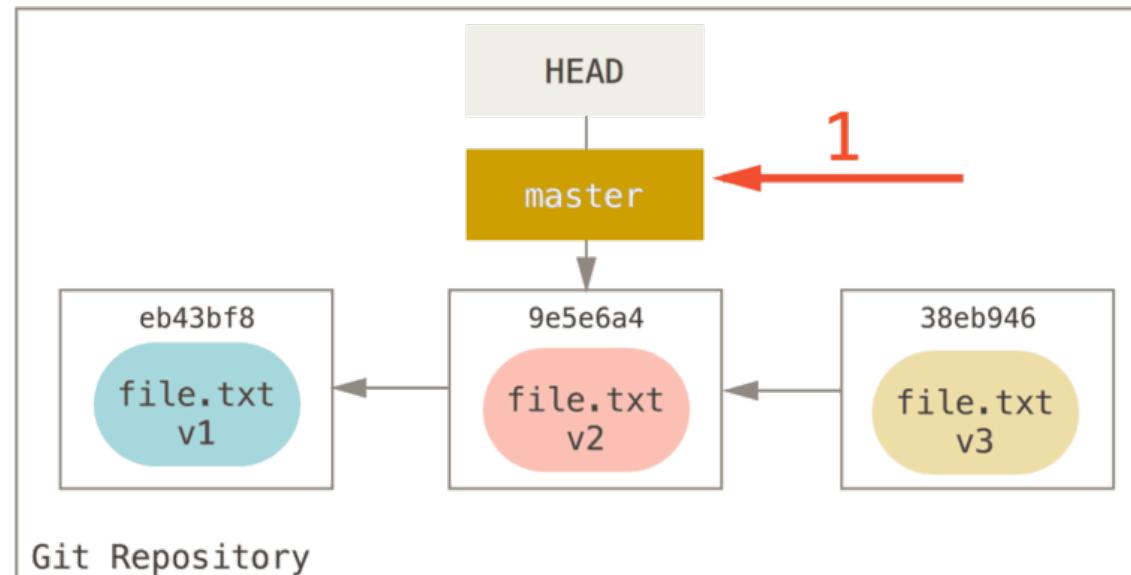
重置图解 2



`git reset [--mixed] HEAD~`

它依然会撤销一上次提交，但还会取消暂存所有的东西。

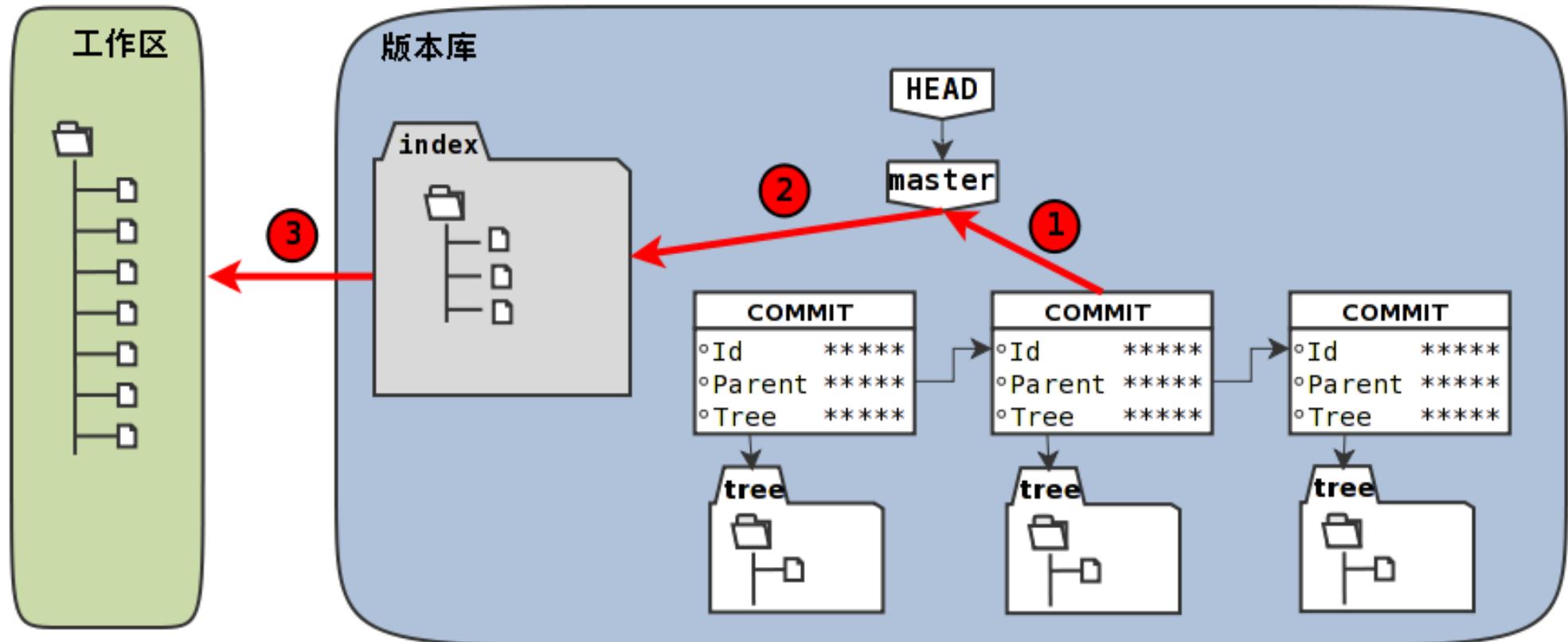
重置图解 3



`git reset --hard HEAD~`

撤销了最后的提交、`git add` 和 `git commit` 命令以及工作目录中的所有工作。`--hard` 标记是 `reset` 命令唯一的危险用法，它也是 Git 会真正地销毁数据的仅有的几个操作之一。

--hard 重置的过程



重置第四种用法解析

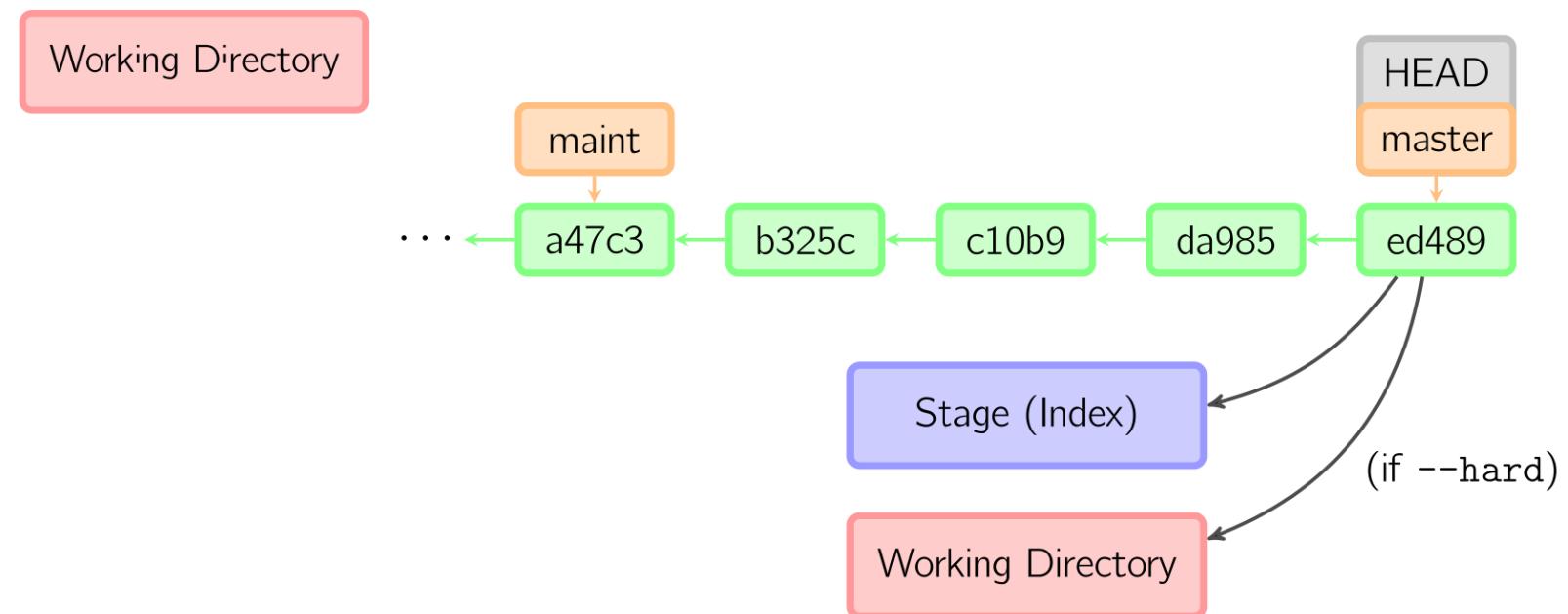
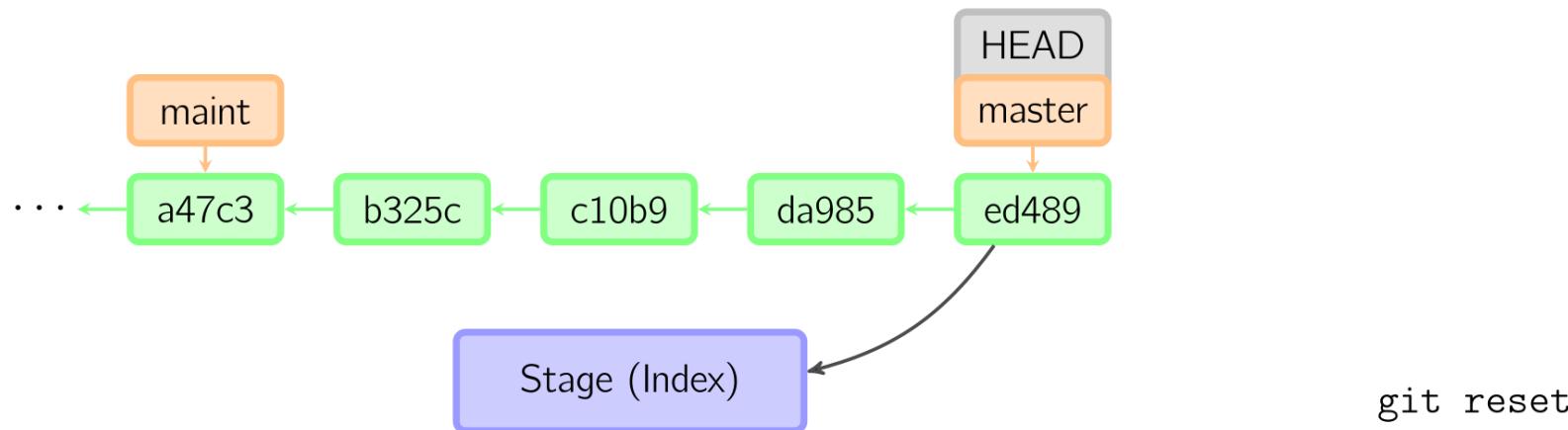
`git reset [--soft | --mixed | --hard] [<commit>]`

- 使用参数 `--hard`，如：`git reset --hard <commit>`，会执行上页图中的全部动作①、②、③，即：
 - × ① 覆盖分支文件的指向，使其指向新的 commit 对象；
 - × ② 覆盖集结区，使集结区恢复到新 commit 对象的数据对象版本
 - × ③ 覆盖工作区，使用新 commit 对象的快照版本签出到工作区。
- 使用参数 `--soft`，如：`git reset --soft <commit>`，只执行上页图中的动作①，不会修改集结区和工作区。
- 使用参数 `--mixed` 或不指定该参数（默认为 `--mixed`），如：`git reset <commit>`，只执行上页图中的动作①，②，不会修改工作区。

`<commit>` 为 `HEAD^` 时，三种重置方式的意义。

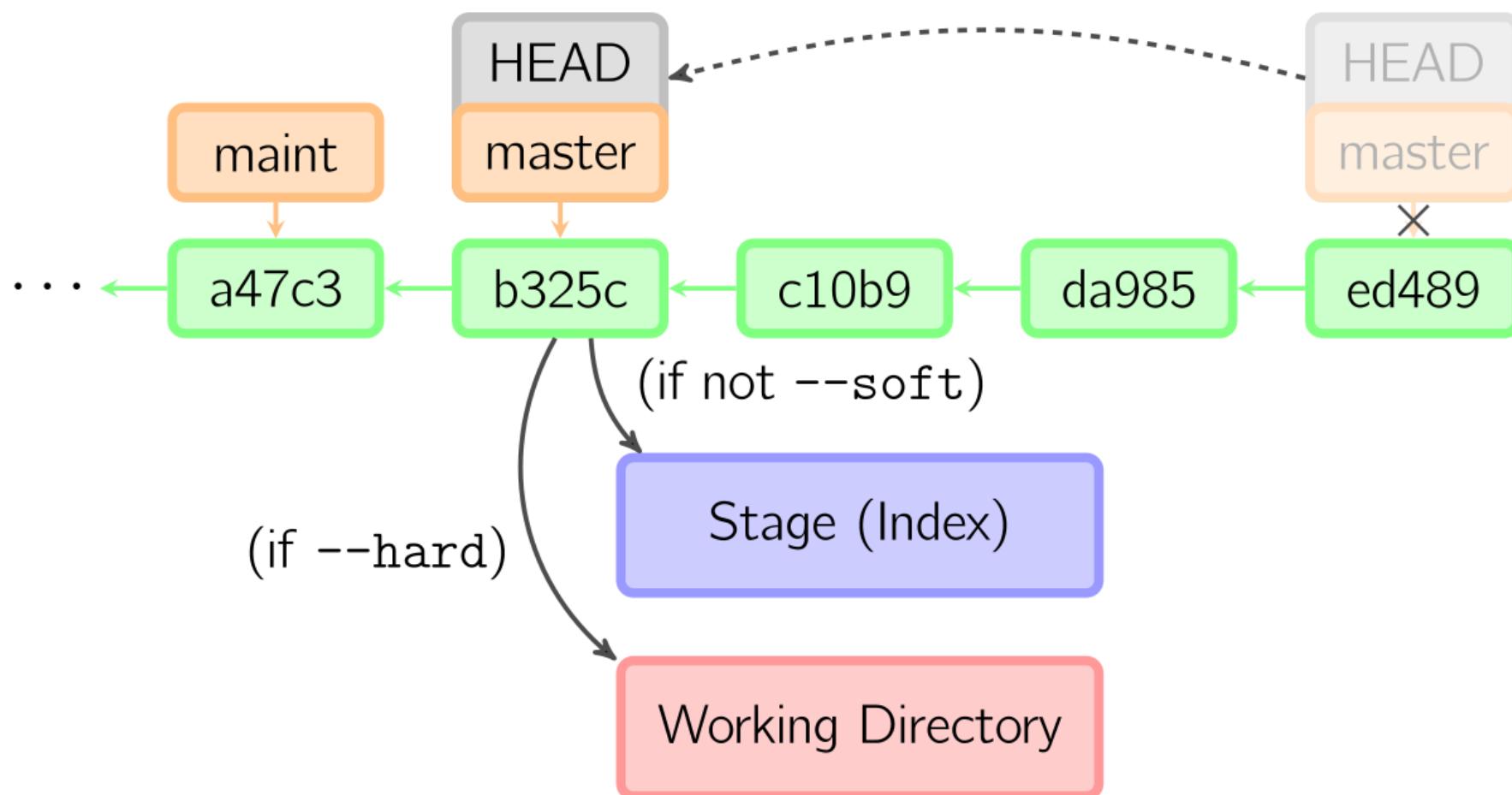
重置图解 4

git reset -- files



重置图解 5

git reset HEAD^{~3}



git checkout 简介

- git 的 **checkout** (签出 / 检出) 操作是恢复文件的某个快照版本到工作目录，**影响的是工作目录中的文件**，签出源可以 git 仓库或集结区。git 仓库中有可能有多个分支，每个分支上有自己的提交链。
 - × 在开发时，经常需要通过 git checkout 恢复某个或某些文件到它的某个版本，此时，不会修改 HEAD 头指针的指向，也就是说**当前分支仍然保持不变**。
 - + 需要在命令中指定要恢复的**文件路径参数**；
 - + git 仓库的多个**提交版本**，集结区的**集结版本**，形成了该路径所标示的文件的多个签出源
 - × 如果想**恢复(签出)**整个**工作目录**的某个版本快照到工作目录，则可以使用**不带文件路径参数**的 git checkout 命令来完成这个任务。此时，会修改 HEAD 头指针，也就是说会**切换当前分支**，然后将指定的目标快照恢复到集结区和工作目录。
 - + 目标如果是分支名称，此时实际上实现了**分支切换**
 - + 目标如果是某个提交 id，则此时得到一个“分离的头指针”，或者说是**切换到了某个匿名分支**。

git checkout 签出文件 / 切换分支

- git checkout——Switch **branches** or **restore** working tree files：
 - × 从仓库或集结区中恢复某个版本的文件到工作目录中，语法如下：

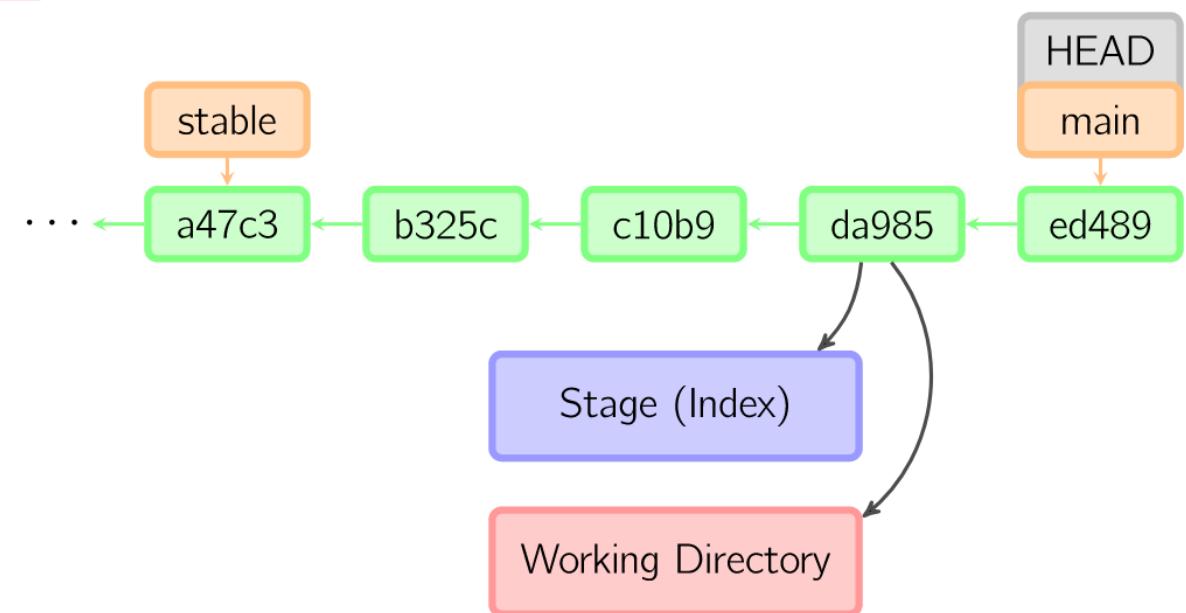
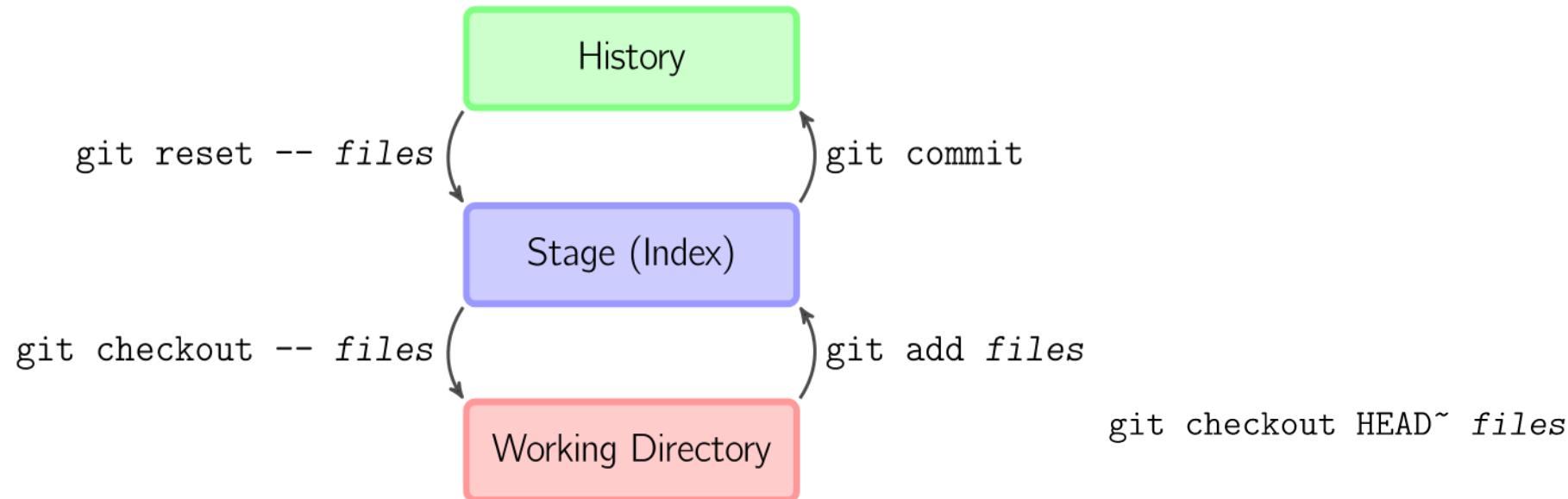
```
git checkout [<tree-ish>] [--] <pathspec>...
```

 - + <tree-ish>：作为签出源的 <commit> 或 <tree> 或 <tag> 对象；省略时，表示集结区。git checkout -- . // 表示使用集结区
 - + <pathspec> 表示想要签出的文件相对工作目录的路径。
 - × 切换到某个有名 / 匿名分支，并更新到集结区和工作目录。语法为：
 - + git checkout [-q] [-f] [-m] [<branch>] // 等价于 git switch [<branch>]
 - + git checkout [-q] [-f] [-m] [--detach] <commit>
// 等价于 git switch [<options>] --detach [<start-point>]

文件的签出场景

- 假设修改了文件 foo.c，进行了一次 stage，然后又修改了 foo.c。对于第二次修改后的 foo.c，如果想把 foo.c 恢复到：
 - ✗ 集结版本，使用 `git checkout -- foo.c` //-- 是分隔符，表示后面是文件路径，不是分支名字
 - ✗ 最新提交版本，使用 `git checkout HEAD -- foo.c` // 恢复 HEAD 中的 foo.c 版本来到集结区和工作目录
 - + 以交互的方式确定差异，使用 `git checkout -p HEAD -- foo.c`
 - ✗ 某个 <commit> 提交版本，使用 `git checkout <commit> -- foo.c` // 恢复 <commit> 中的 foo.c 版本来到集结区和工作目录
 - ✗ 某个 <tag> 对象版本，使用 `git checkout <tag> -- foo.c`

从 index 或某个提交签出文件

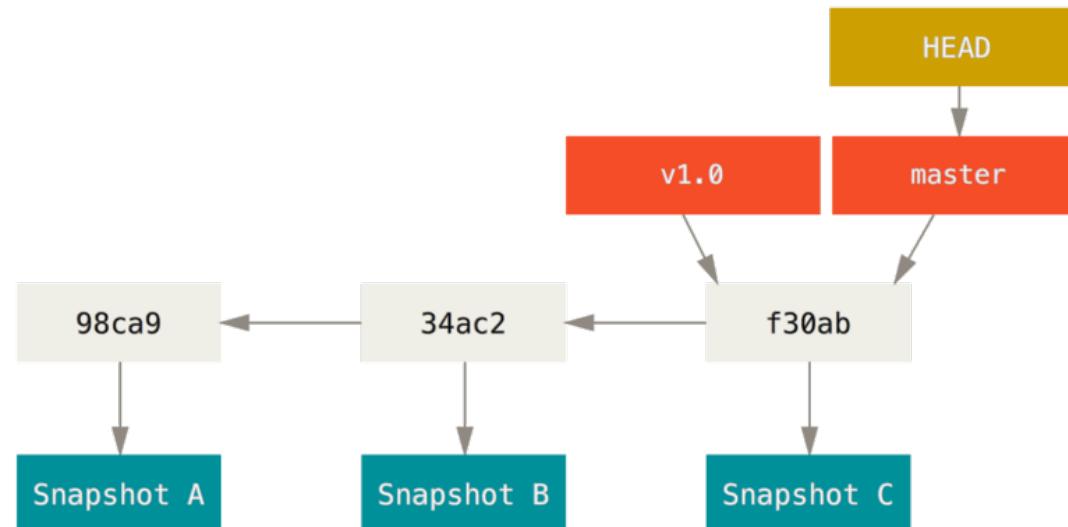


- 版本控制系统
- Git 独奏
- Git 和声

- Git 独奏
 - ✗ 三个区域与对象模型
 - ✗ 集结、提交
 - ✗ 重置、签出
 - ✗ Git 分支

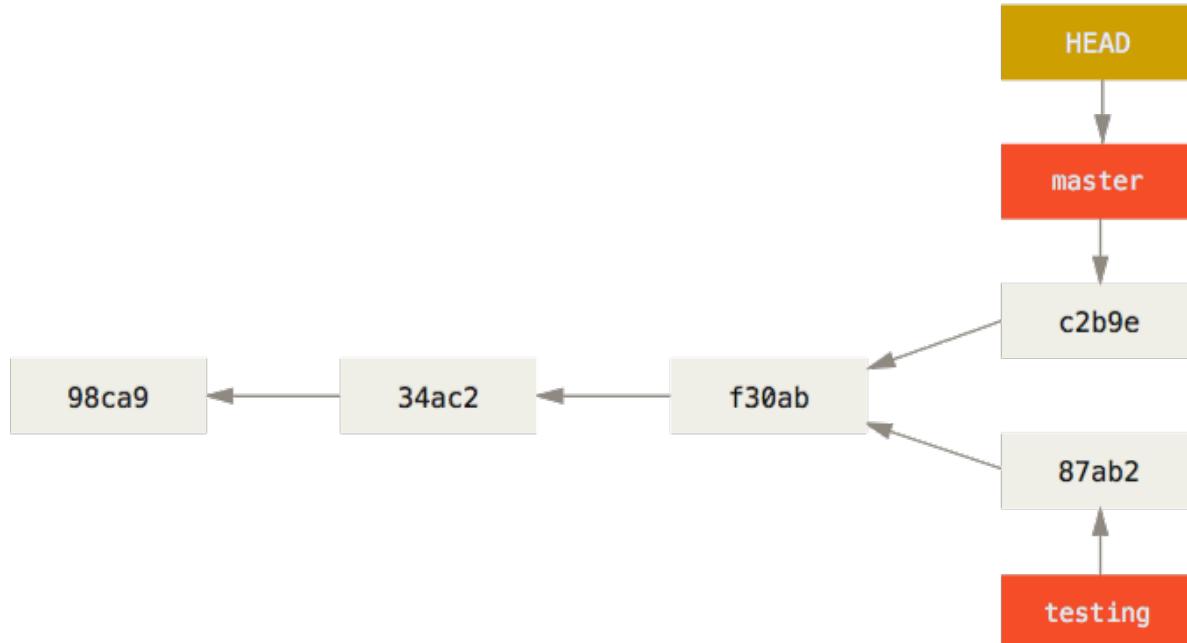
git 分支

- git 中的分支，其实是指向某个提交序列中最后一个 commit 对象的“可变指针”。这种指针实际上是一个文本文件，使用**分支名字**进行命名，其内容是最后一个 commit 对象的指纹字符串，一般存储在 .git/refs/heads/ 目录中，也称为命名引用 named reference。
- git 的默认分支为 master 主分支。在若干次提交后，版本库中就有了一个指向最后一个 commit 对象的 master 分支，它在每次提交的时候都会自动向前移动，相当于一个分支演进的“游标”。

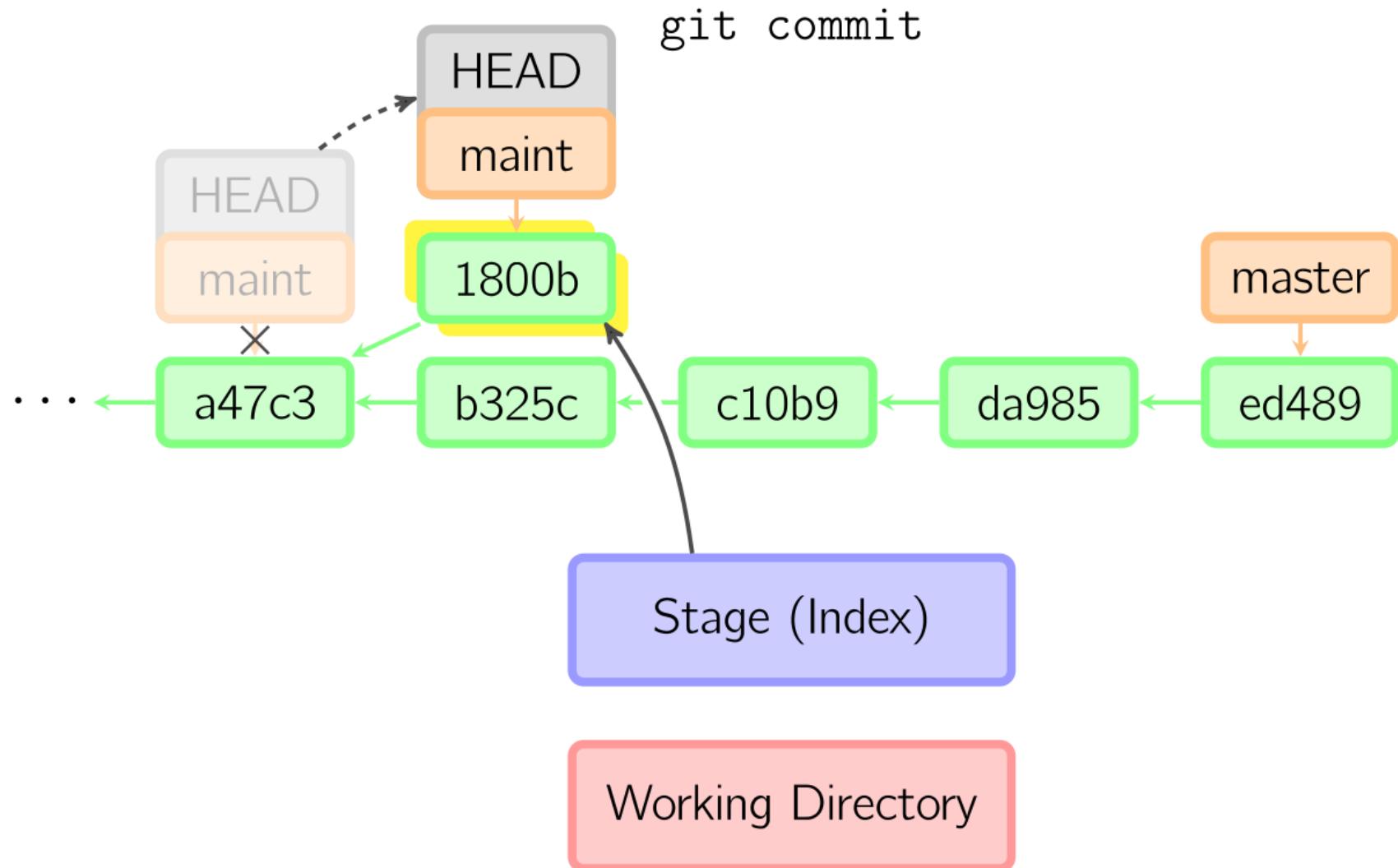


HEAD 与当前分支

- 仓库经常有多个分支，这意味着在 `.git/refs/heads/` 目录下会有多个对应的分支文件。但是，在任何时刻，开发者只能在一个分支上工作，这个分支称为当前分支。
- 为了指示当前分支是当前众多分支的哪一个，git 在 `.git` 目录下使用了一个名为 `HEAD` 的文件，用于保存当前分支的分支文件的路径；在 git 命令中，也使用字符串 `HEAD` 来表示当前分支。



maint 分支的提交

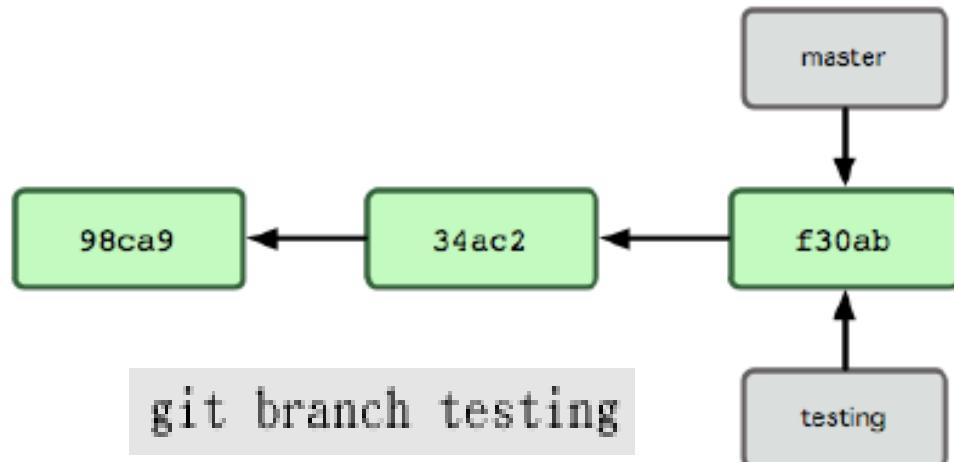


分支简介

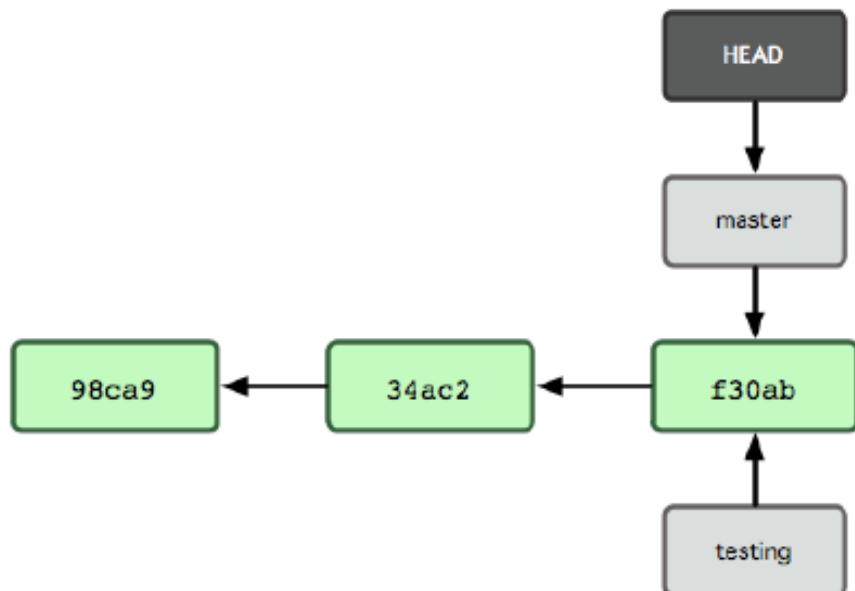
- 几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。在很多版本控制系统中，这是个昂贵的过程，常常需要创建一个源代码目录的完整副本，对大型项目来说会花费很长时间。
- Git 的分支模型是 Git 的“必杀技”，其特别之处在于：
 - Git 分支操作是难以置信的轻量级，它的新建操作几乎可以在瞬间完成，并且在不同分支间切换起来也差不多一样快。
 - 和许多其他版本控制系统不同，Git 鼓励在工作流程中频繁使用分支与合并，哪怕一天之内进行许多次都没有关系。

理解分支的概念并熟练运用后，你才会意识到为什么 Git 是一个如此强大而独特的工具，并从此真正改变你的开发方式。

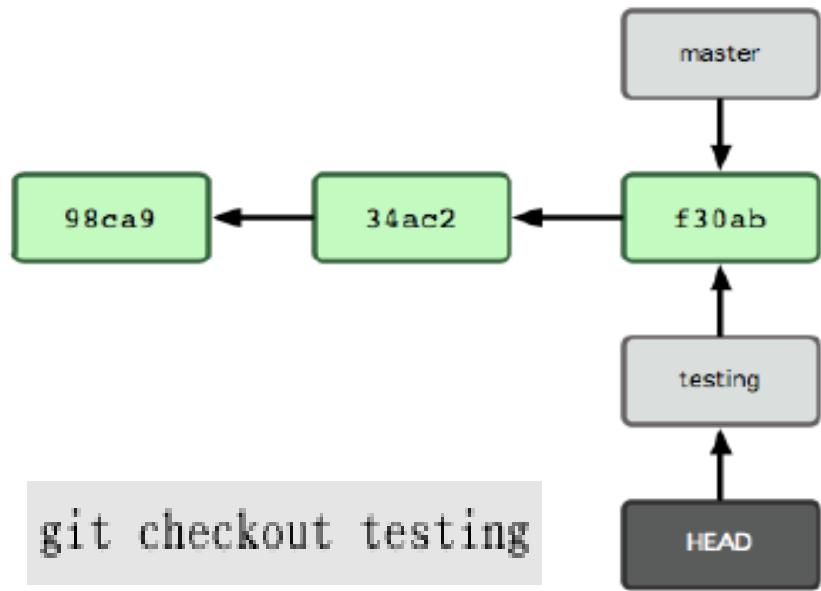
创建并切换分支



Creating a new branch pointer



- 为了同时进行多个开发任务，可以创建多个分支。不同的开发者分别在不同分支上开发，最后在合适的时候再将各个分支上的开发成果合并。
- 有时，创建分支也可能意味着项目真正的 fork



HEAD 在你转换分支时指向新的分支

使用 git checkout 签出分支

git-checkout 可以签出某个分支到工作树中，用法如下：

git checkout [<branch>]

git checkout [-b] <new_branch> <start-point>

- 第 1 种用法，会修改 HEAD 指向 <branch> 分支文件，使得当前分支变成了 <branch> 分支，然后签出 <branch> 分支的最新 <commit> 对象的版本到集结区和工作区中。
 - × 但对工作树的局部修改会被保存，以便可以被提交到 <brach>。
- 第 2 种用法，用于创建并切换到新的分支 <new_branch>，新的分支从 <start_point> 指向的 commit 对象开始创建。
- git checkout 这种用法的等价命令为 **git switch**

git switch

```
git switch [<options>] [--no-guess] <branch>
git switch [<options>] --detach [<start-point>]
git switch [<options>] (-c|-C) <new-branch> [<start-point>]
git switch [<options>] --orphan <new-branch>
```

DESCRIPTION

Switch to a specified branch. The working tree and the index are updated to match the branch. All new commits will be added to the tip of this branch.

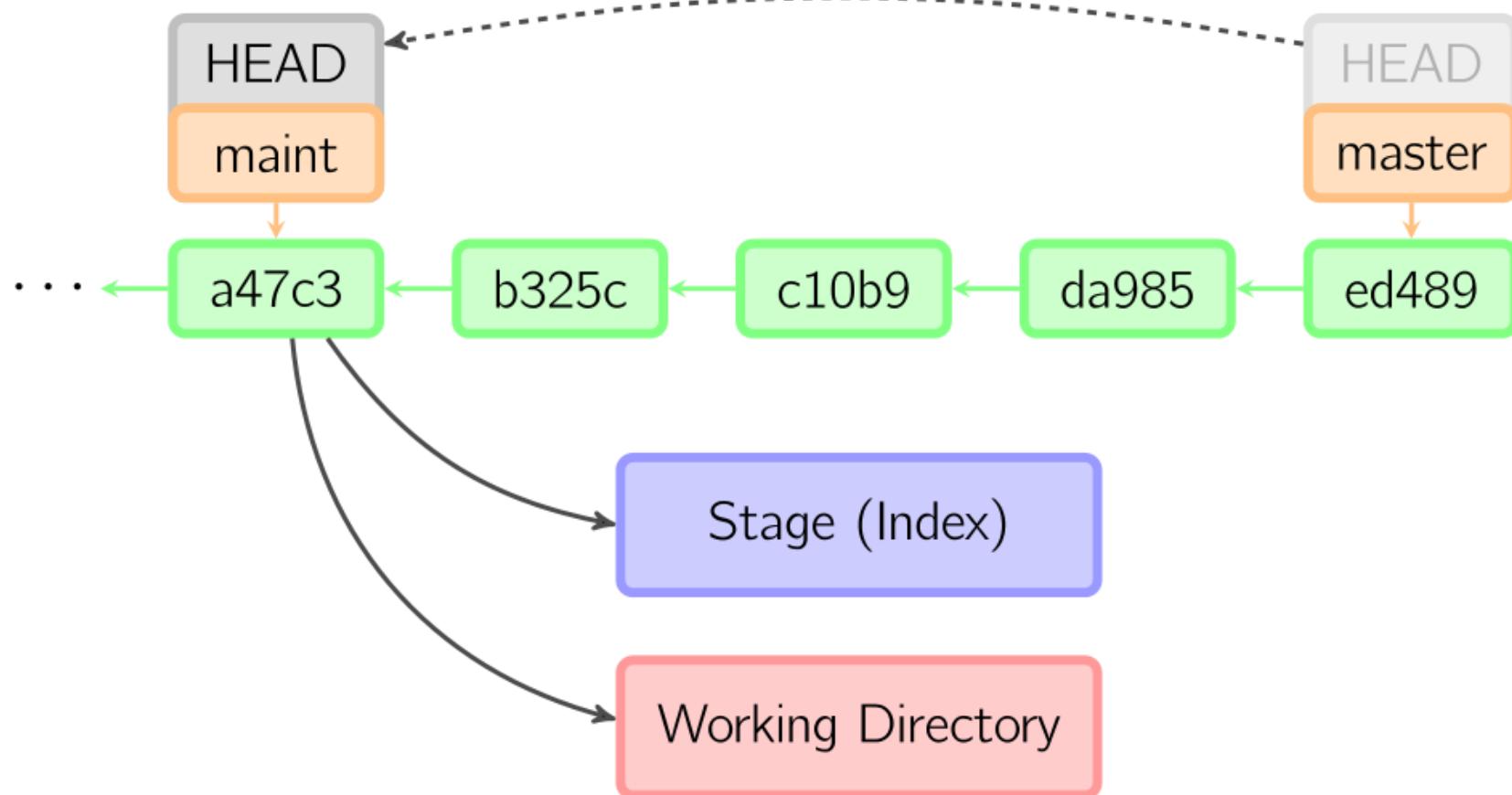
Optionally a new branch could be created with either `-c`, `-C`, automatically from a remote branch of same name (see `--guess`), or detach the working tree from any branch with `--detach`, along with switching.

Switching branches does not require a clean index and working tree (i.e. no differences compared to `HEAD`). The operation is aborted however if the operation leads to loss of local changes, unless told otherwise with `--discard-changes` or `--merge`.

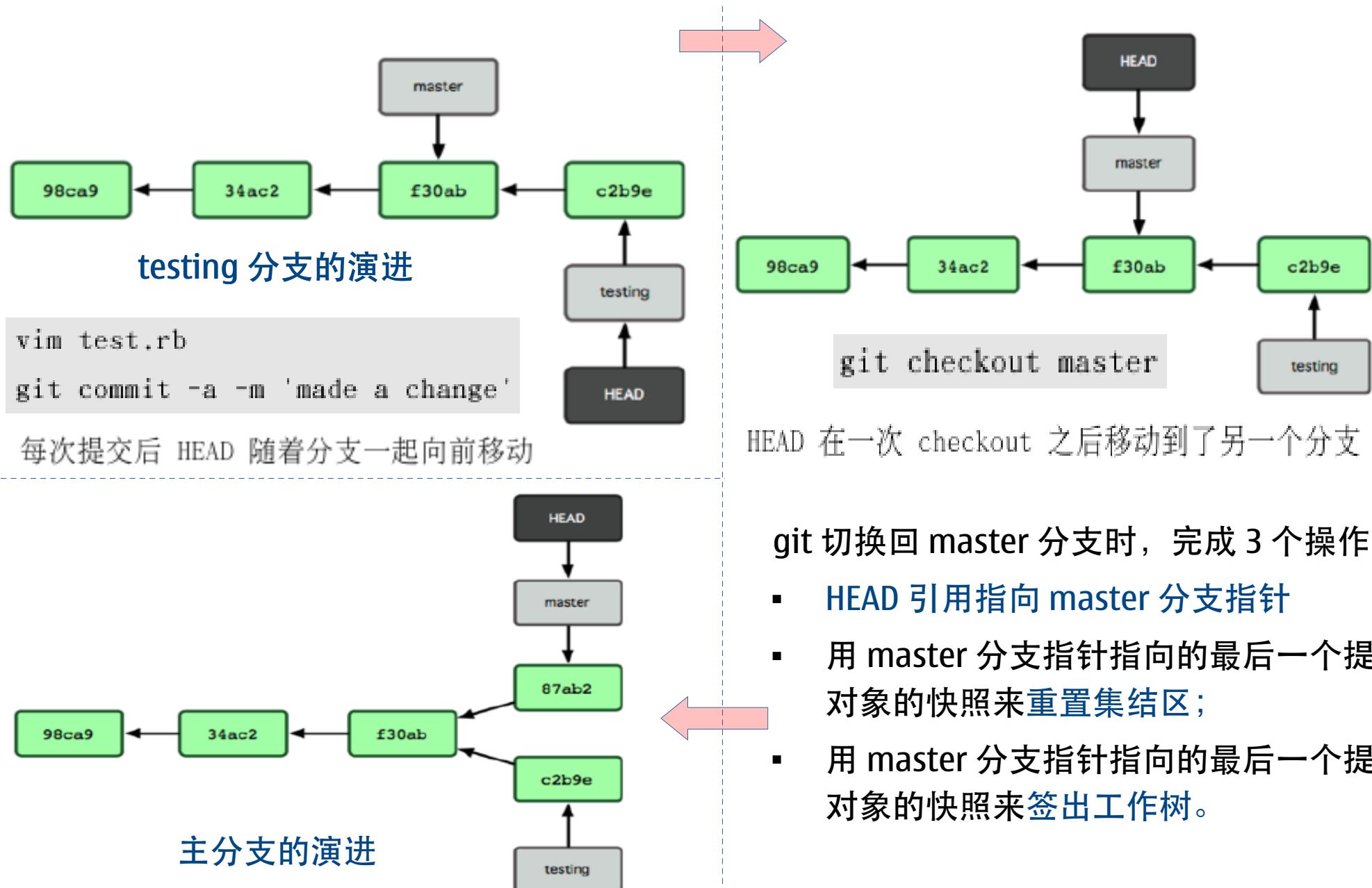
THIS COMMAND IS EXPERIMENTAL. THE BEHAVIOR MAY CHANGE.

git switch 切换分支

git switch maint



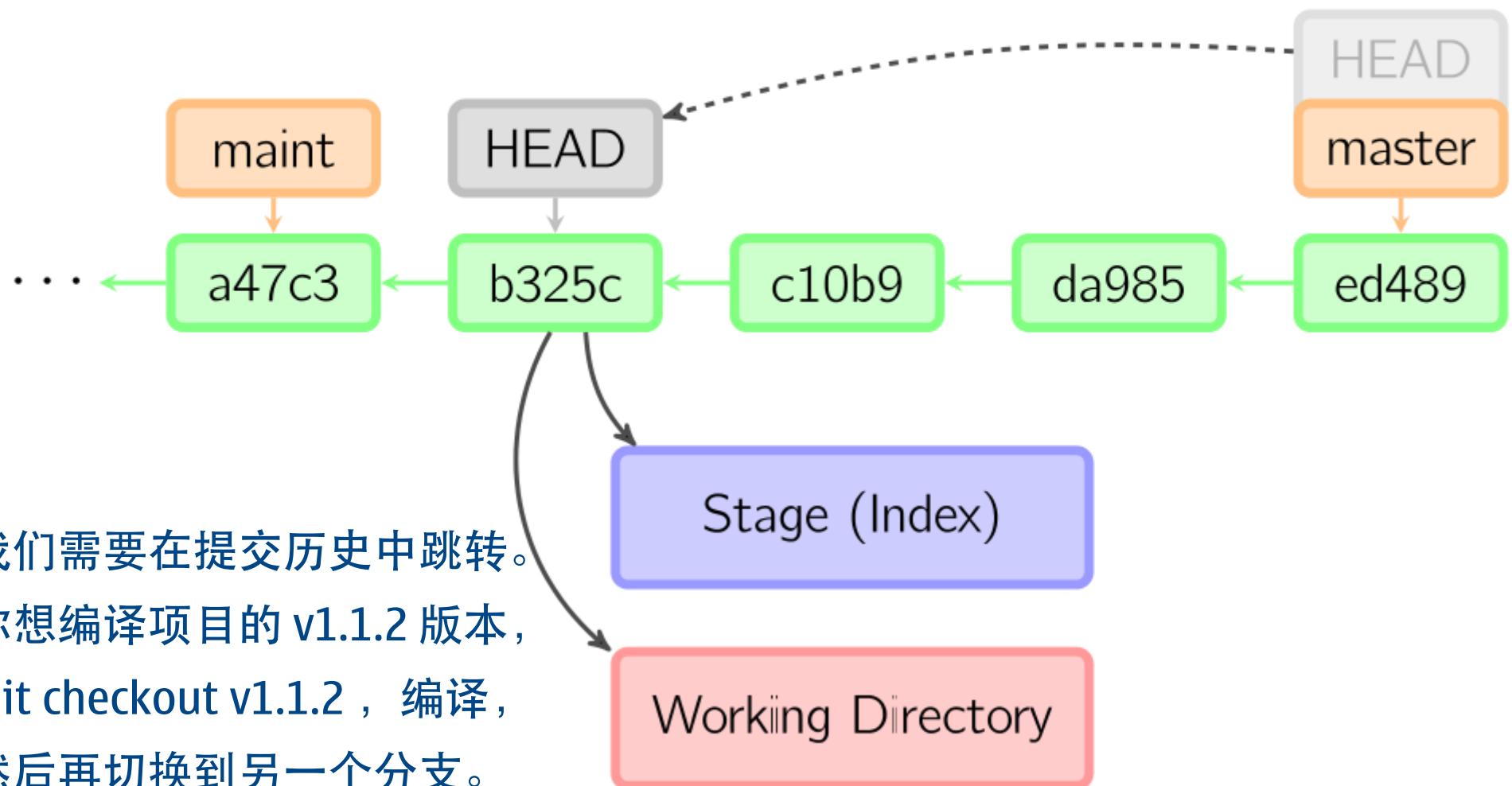
切换与签出



切换到匿名分支 1/3

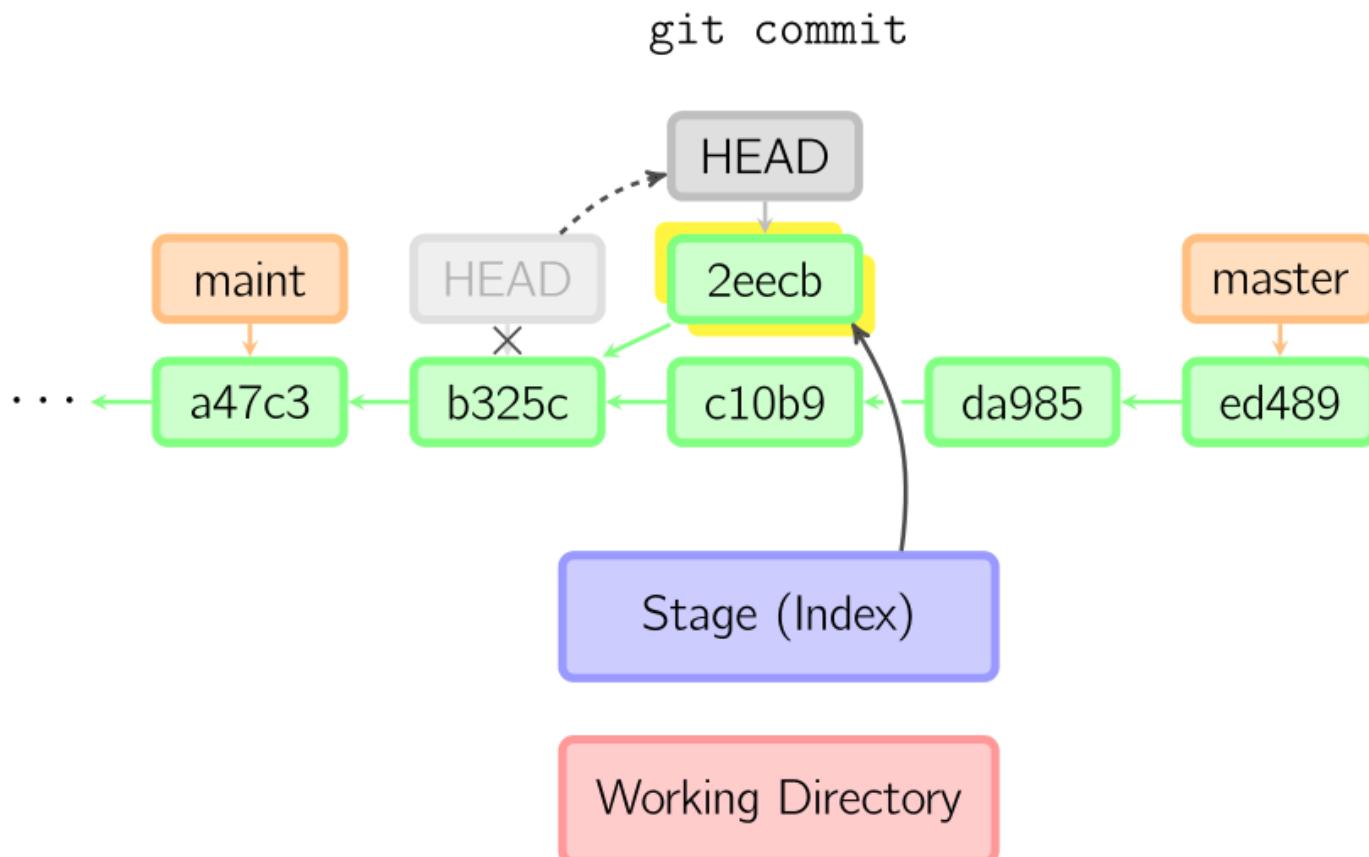
`git switch --detached b325c // 等价于 git checkout b324c/master~3`

这里创建了一个没有名字的匿名分支，指向 `b325c`。



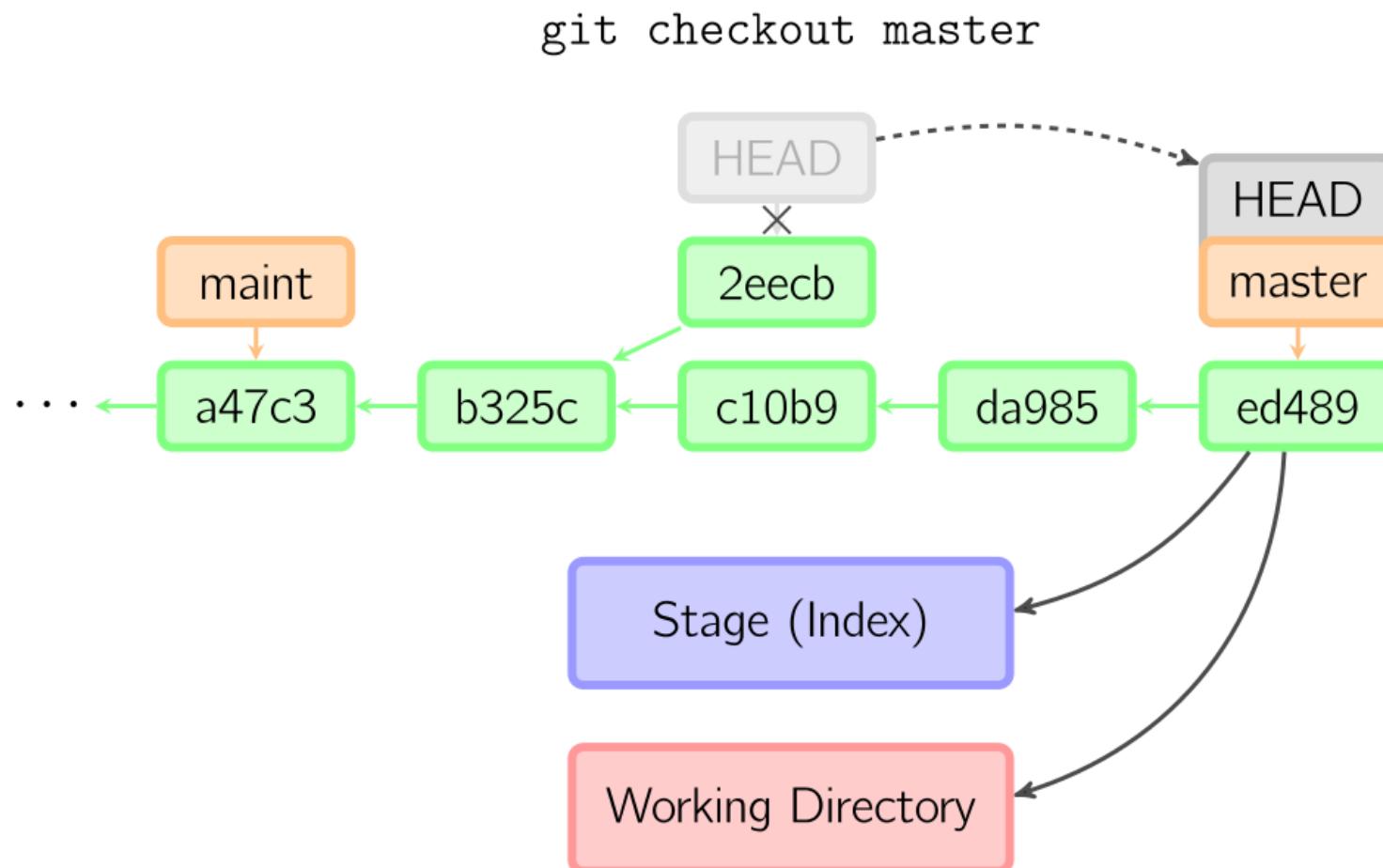
切换到匿名分支 2/3

- 对于匿名分支，其 commit 工作与普通情况一样，只是没有命名的分支文件被更新，由 HEAD 指向最新的提交节点。下面表示匿名分支做了 1 次提交。



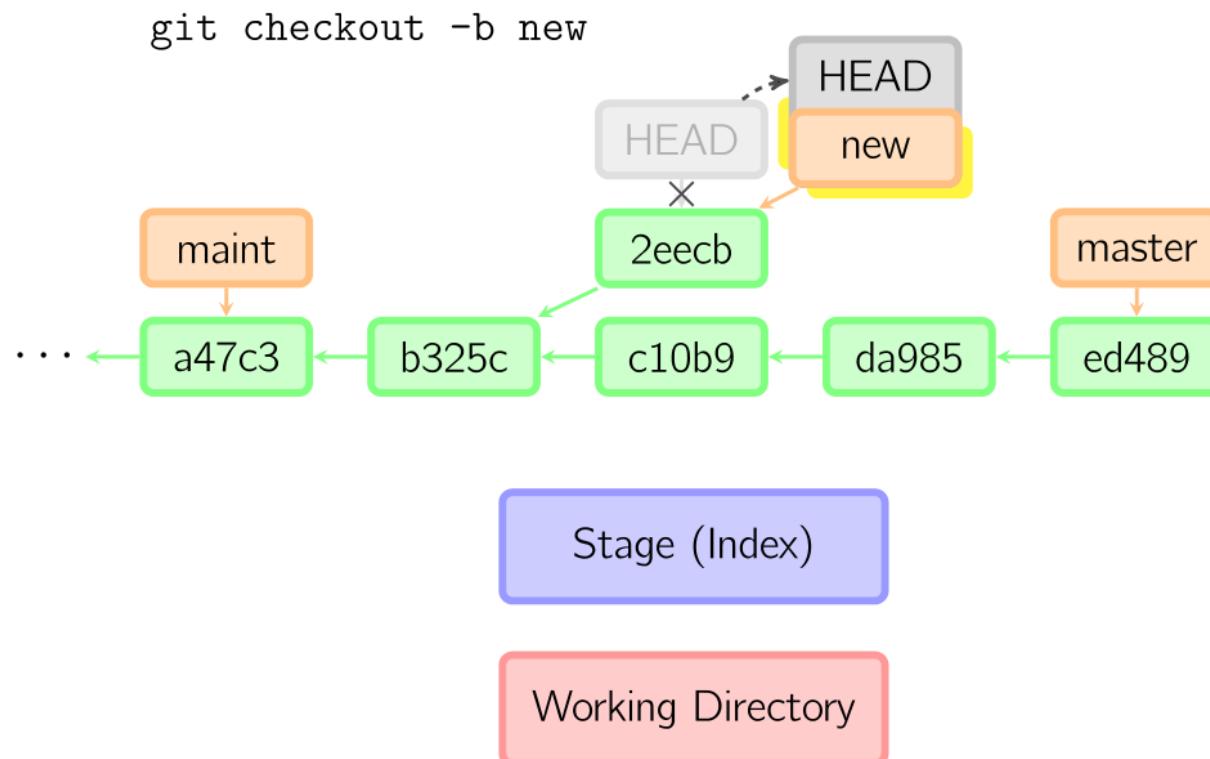
切换到匿名分支 3/3

- 匿名分支做 1 次提交后，切换到 master，此时匿名分支无法通常引用找到，就被丢掉了。

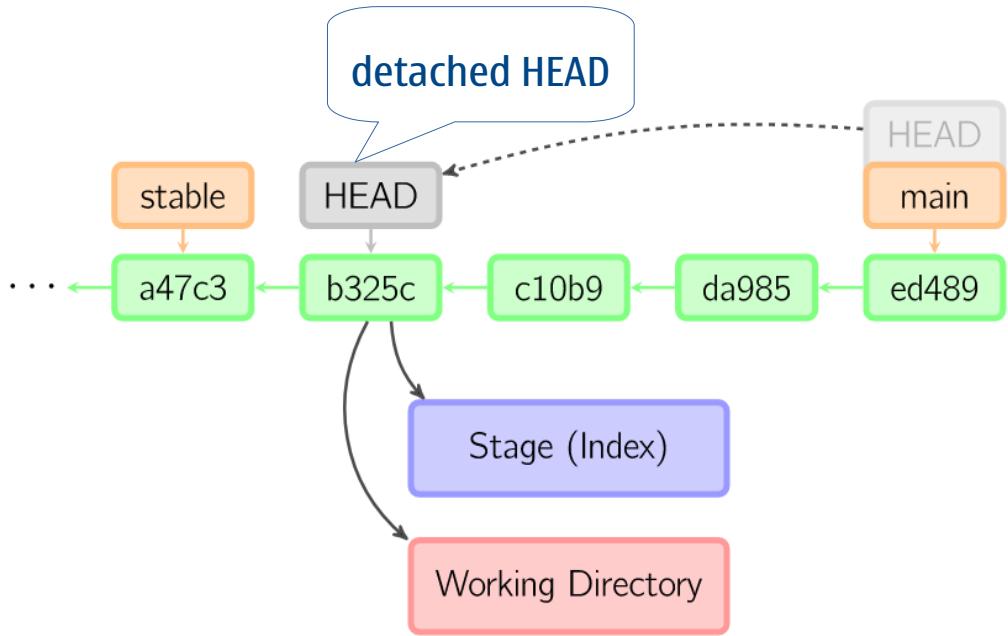


切换到匿名分支 3' / 3

- 如果想保存匿名分支做的提交，可以使用下面的命令给分支命名，相当于 git branch new

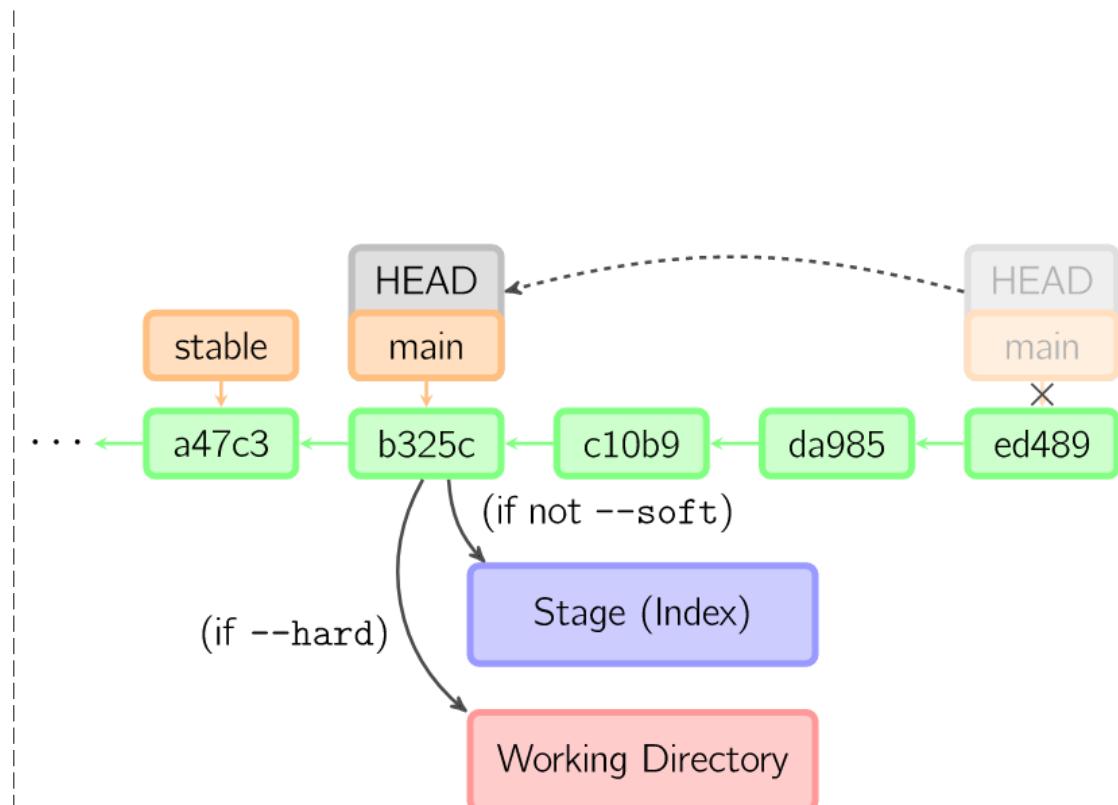


git switch --detached <commit> 和 git reset --hard 对比



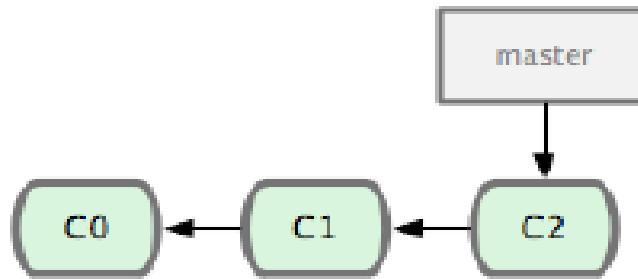
git switch --detached b325c

匿名分支的 HEAD 称为 **detached HEAD**，表示此时的 HEAD 和与有名分支的名字的暂时分离

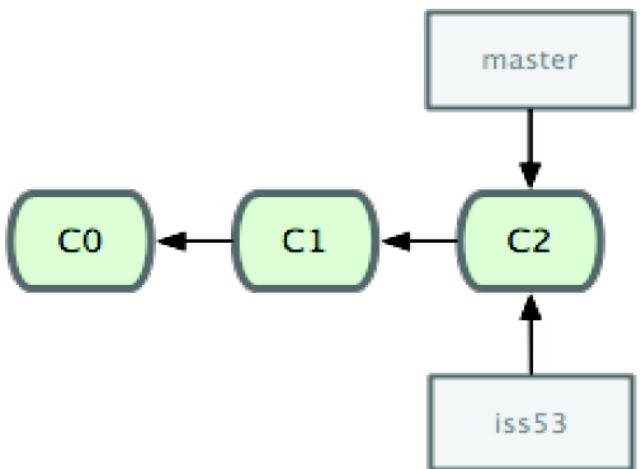


git reset b325c

git 分支管理示例



假设你正在项目中愉快地工作，并且已经提交了几次更新



创建了一个新的分支指针

`$ git switch -c iss53`

等价于

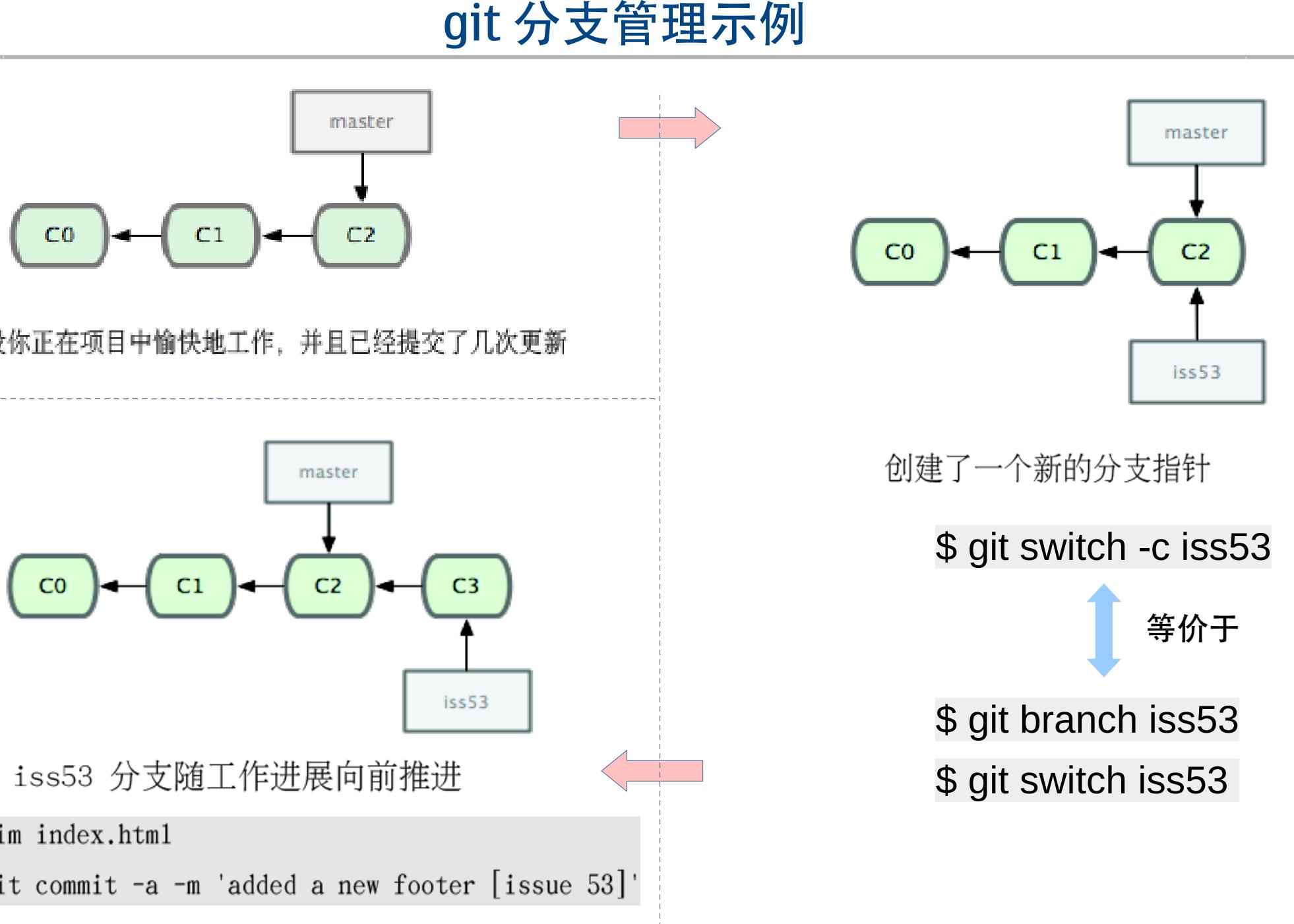
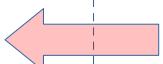
`$ git branch iss53`

`$ git switch iss53`

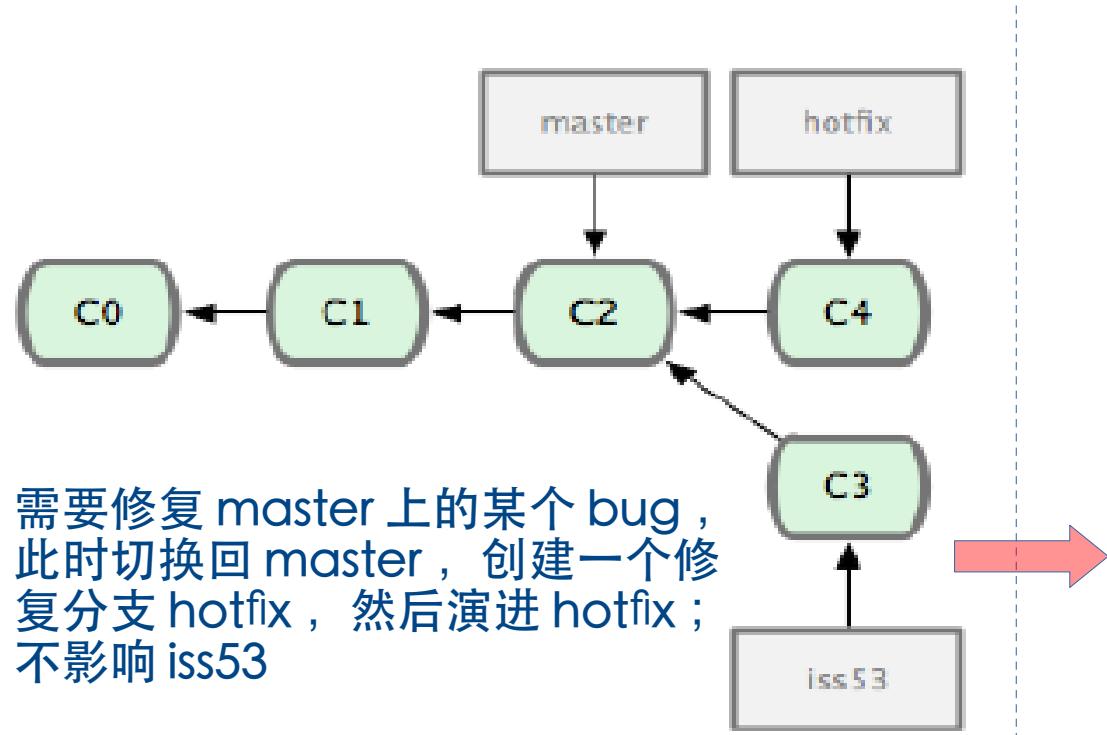
iss53 分支随工作进展向前推进

```

$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
  
```



git 分支管理示例 (续 1)

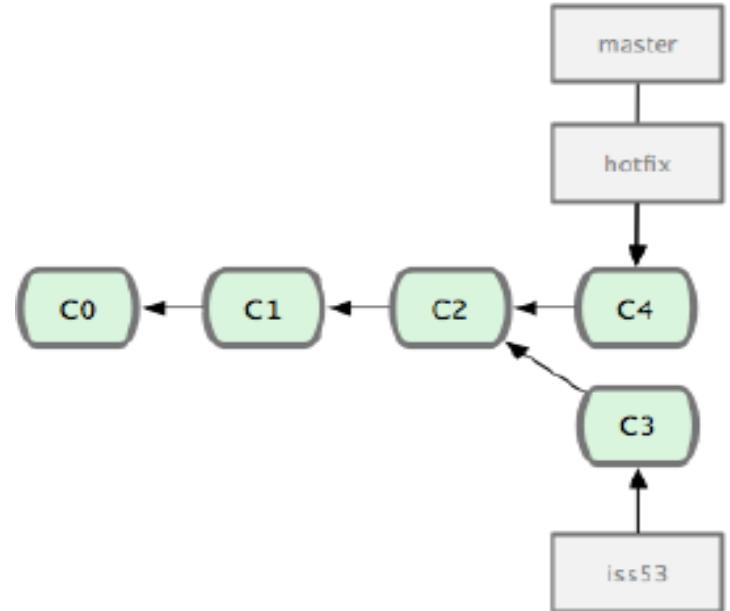


```
$ git switch master
```

```
$ git switch -c hotfix
```

```
$ vim index.html
```

```
$ git commit -a -m 'fixed the broken email address'
```



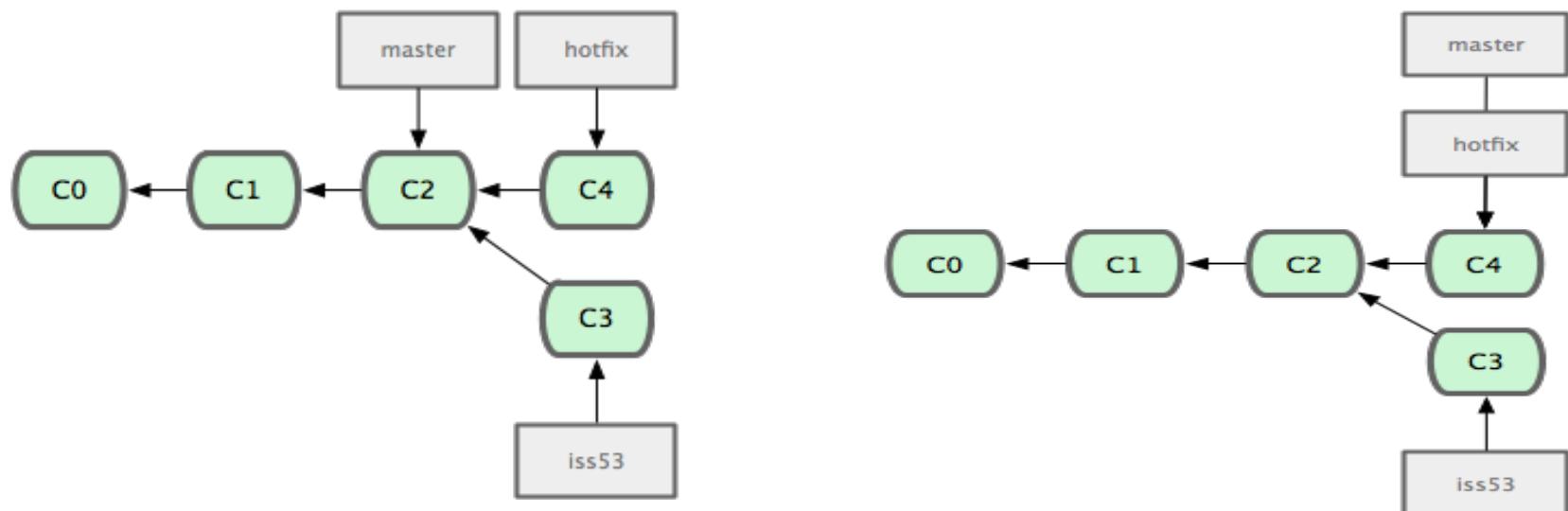
```
$ git switch master
```

```
$ git merge hotfix
```

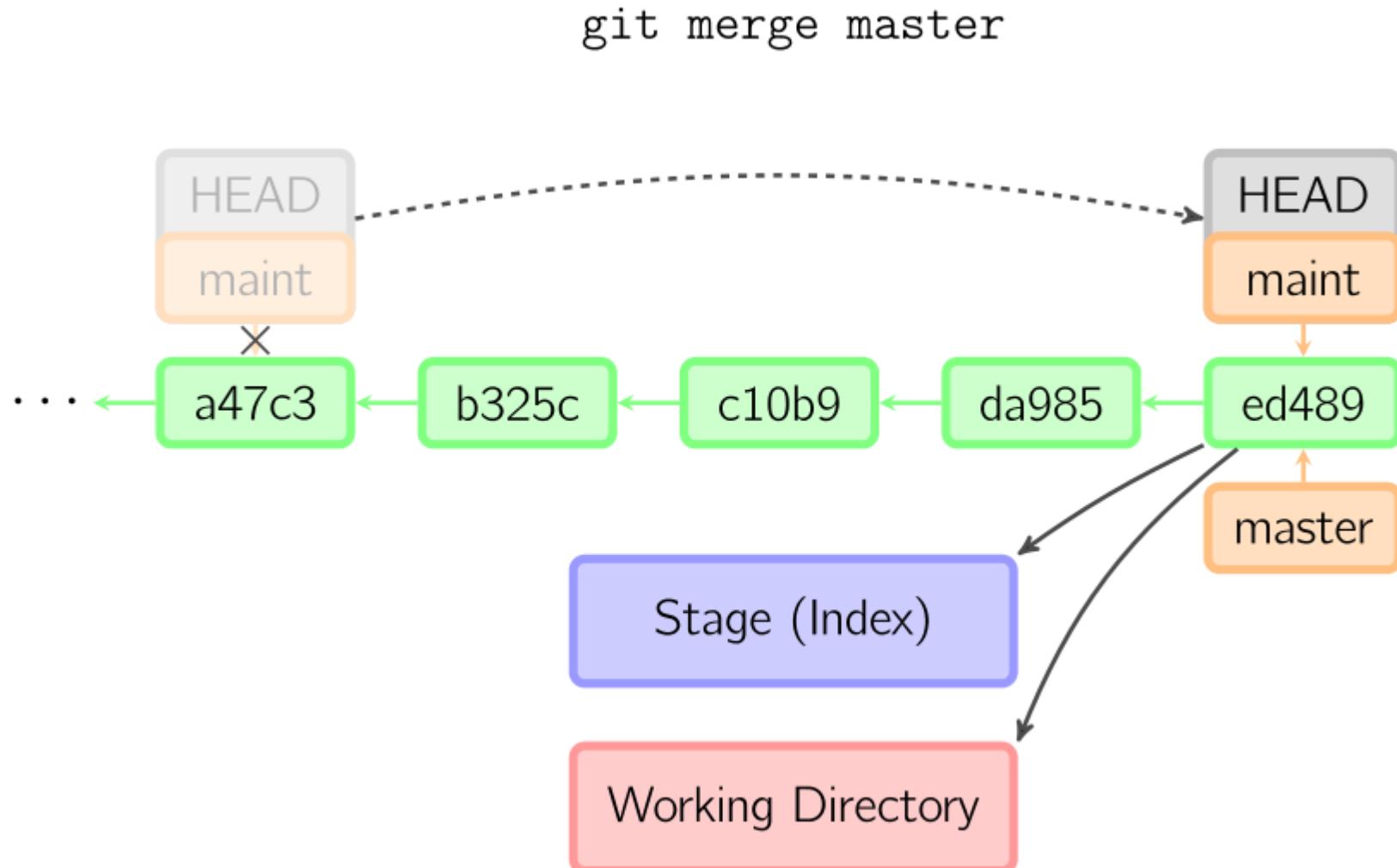
合并之后, master 分支和 hotfix 分支指向同一位置

快进合并

- fast-forward 是一种特殊的 merge 类型。master 分支在 hotfix 分支演进之后没有任何提交，即 master 分支真包含于 hotfix 分支，这时在 master 分支上合并 hotfix 分支，git 并不会创建一个新的合并提交对象 (merge commit) 以表达两个分支的合并，而是直接把 master 的内容修改为与 hotfix 一致，这就是 fast forward 合并。



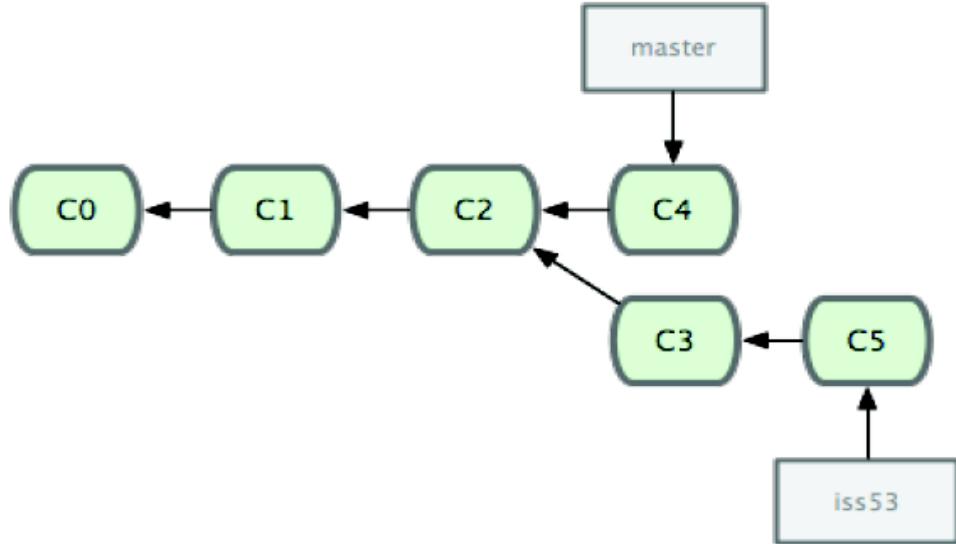
快进合并图解



git 分支管理示例 (续 3)

删除修复分支 hotfix

```
$ git branch -d hotfix
```



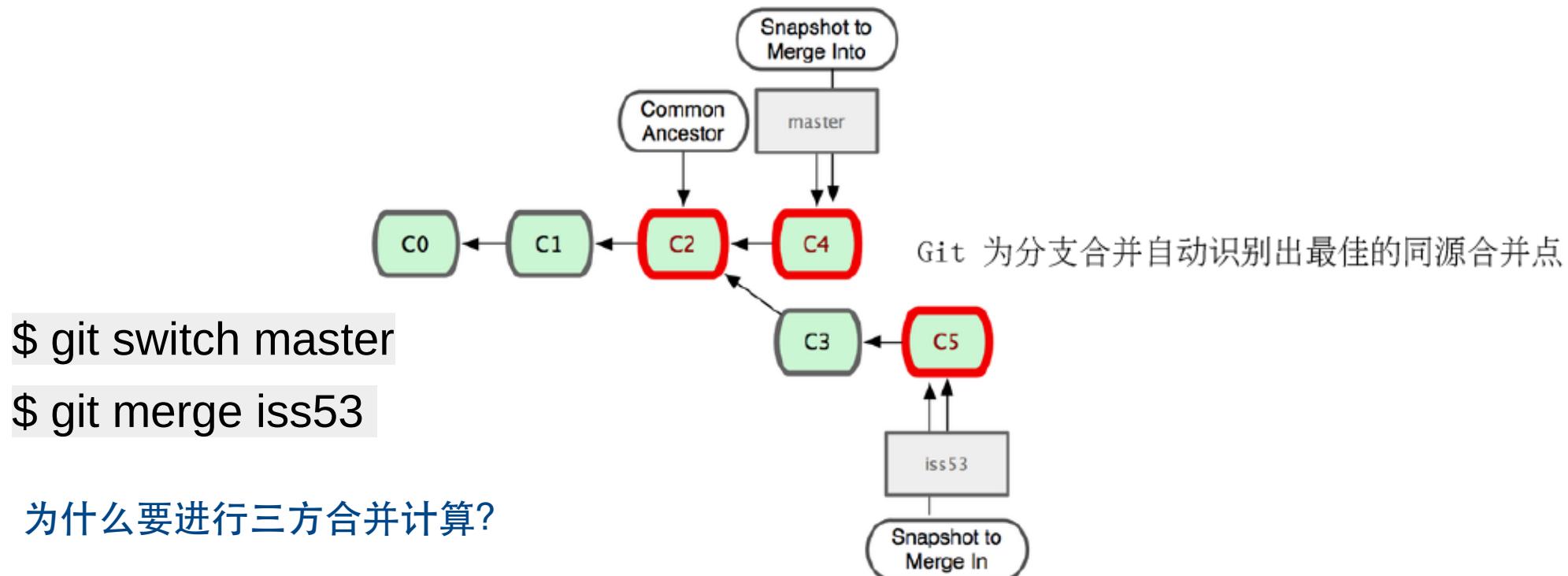
iss53 分支可以不受影响继续推进

```
$ git switch iss53
```

```
$ vim index.html  
$ git commit -a -m 'finished the new footer [issue 53]'
```

分支 iss53 上的开发工作完成后，接下来需要把 iss53 合并到 master 分支上

git 分支管理示例 (续 4)



请注意，这次合并的实现，并不同于之前 hotfix 的并入方式。这一次，你的开发历史是从更早的地方开始分叉的。由于当前 master 分支所指向的 commit (C4) 并非想要并入分支 (iss53) 的直接祖先，Git 不得不进行一些处理。就此例而言，Git 会用两个分支的末端 (C4 和 C5) 和它们的共同祖先 (C2) 进行一次简单的三方合并计算。图 3.16 标出了 Git 在用于合并的三个更新快照：

Git 没有简单地把分支指针右移，而是对三方合并的结果作一个新的快照，并自动创建一个指向它的 commit (C6)（见图 3.17）。我们把这个特殊的 commit 称作合并提交（merge commit），因为它的祖先不止一个。

值得一提的是 Git 可以自己裁决哪个共同祖先才是最佳合并基础；这和 CVS 或 Subversion (1.5 以后的版本) 不同，它们需要开发者手工指定合并基础。所以此特性让 Git 的合并操作比其他系统都要简单不少。

三方合并计算：自动合并成功

假设有一个文件 main.c，在不同的提交版本，经历了如下的变化过程。
由于三方合并计算，才会得到最后自动合并的结果：

```
#include <stdlib.h>
int main(void)
{
    printf("hello");
    return 1;
}
```

master c1

```
#include <stdlib.h>
int main(void)
{
    printf("hello");
    return 0;
}
```

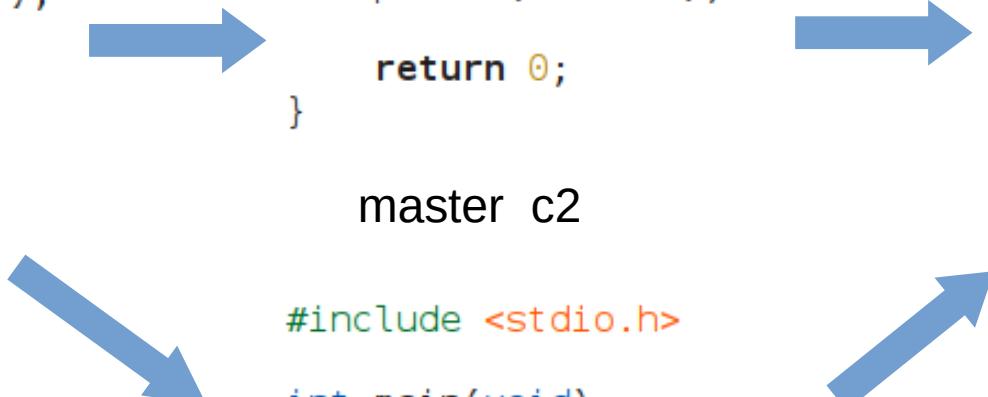
master c2

```
#include <stdio.h>
int main(void)
{
    printf("hello");
    return 0;
}
```

master merge dev

```
#include <stdio.h>
int main(void)
{
    printf("hello");
    return 1;
}
```

dev c1



三方合并计算：自动合并失败

如果项目分支时基于的提交不包含 main.c 文件，并且在 master 分支和 dev 分支分离后，分别增加了 main.c 文件。那么在 master 合并 dev 时，这两个 main.c 文件将会自动合并失败，并报告第 1 行和 return 语句行有冲突。

master c1 :
haven't main.c

```
#include <stdlib.h>

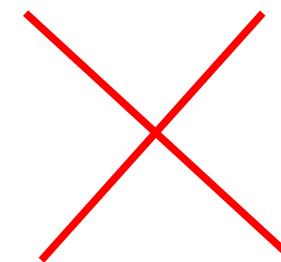
int main(void)
{
    printf("hello");
    return 0;
}
```

master c2

```
#include <stdio.h>

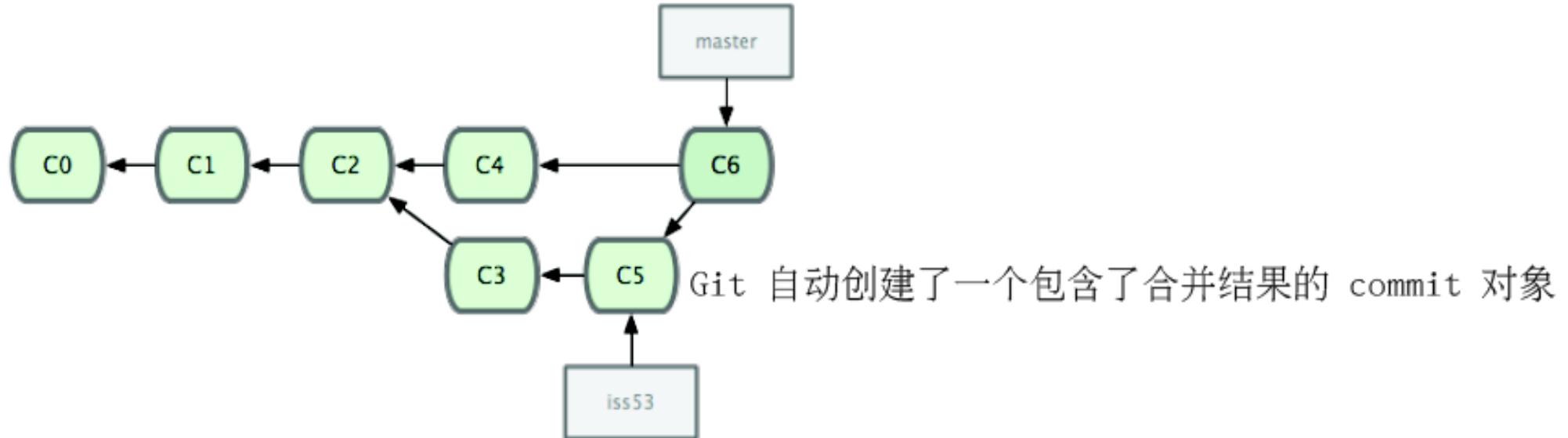
int main(void)
{
    printf("hello");
    return 1;
}
```

dev c1

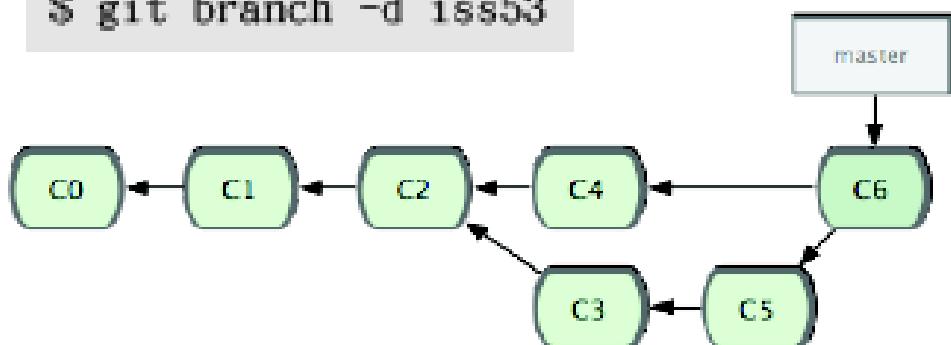


master merge dev

git 分支管理示例 (续 5)



```
$ git branch -d iss53
```



冲突合并

如果三方合并时发生冲突，需要进行冲突合并

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

```
[master*]$ git status
index.html: needs merge
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# unmerged:    index.html
#
```

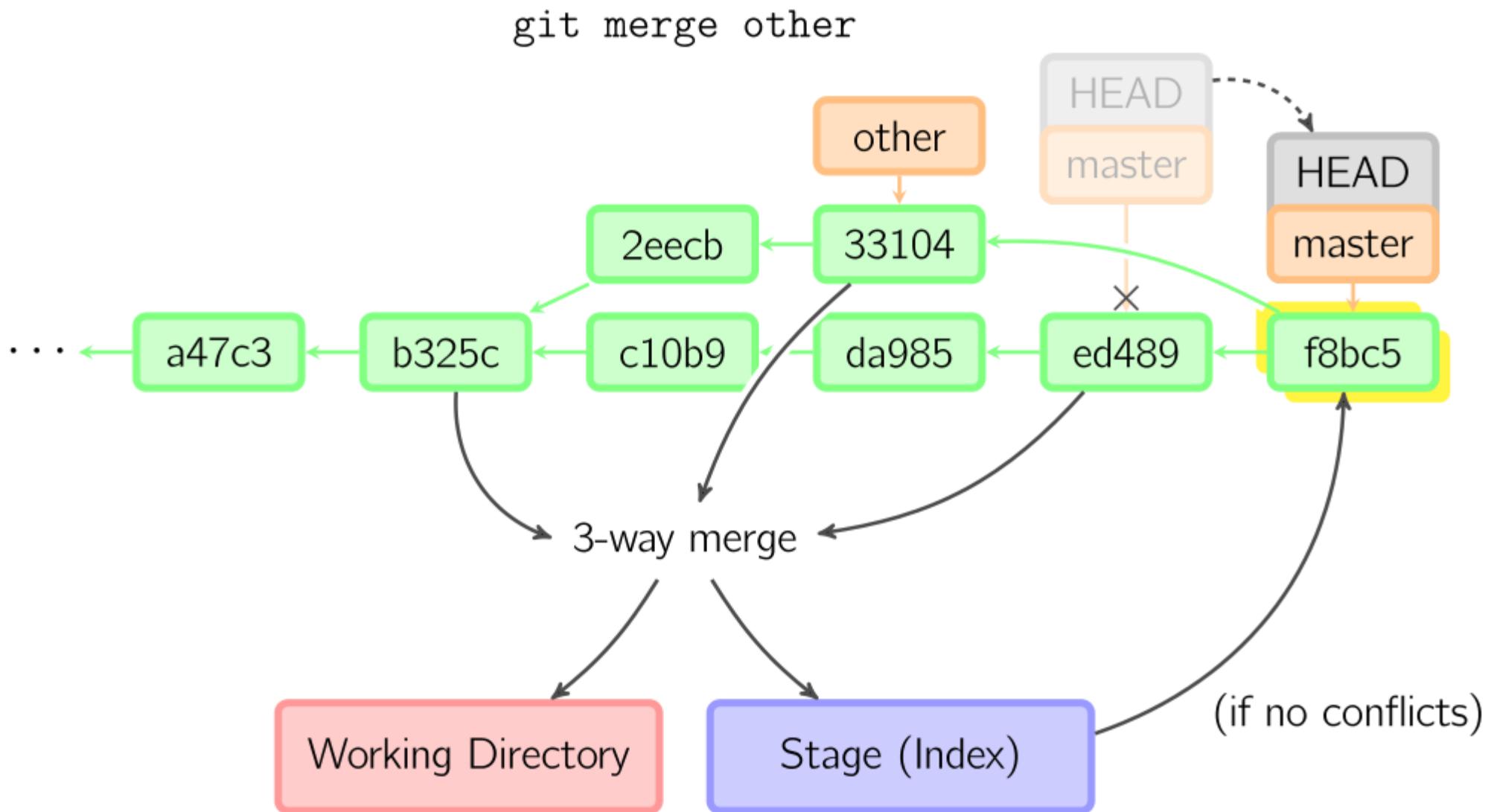
```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>> iss53:index.html
```



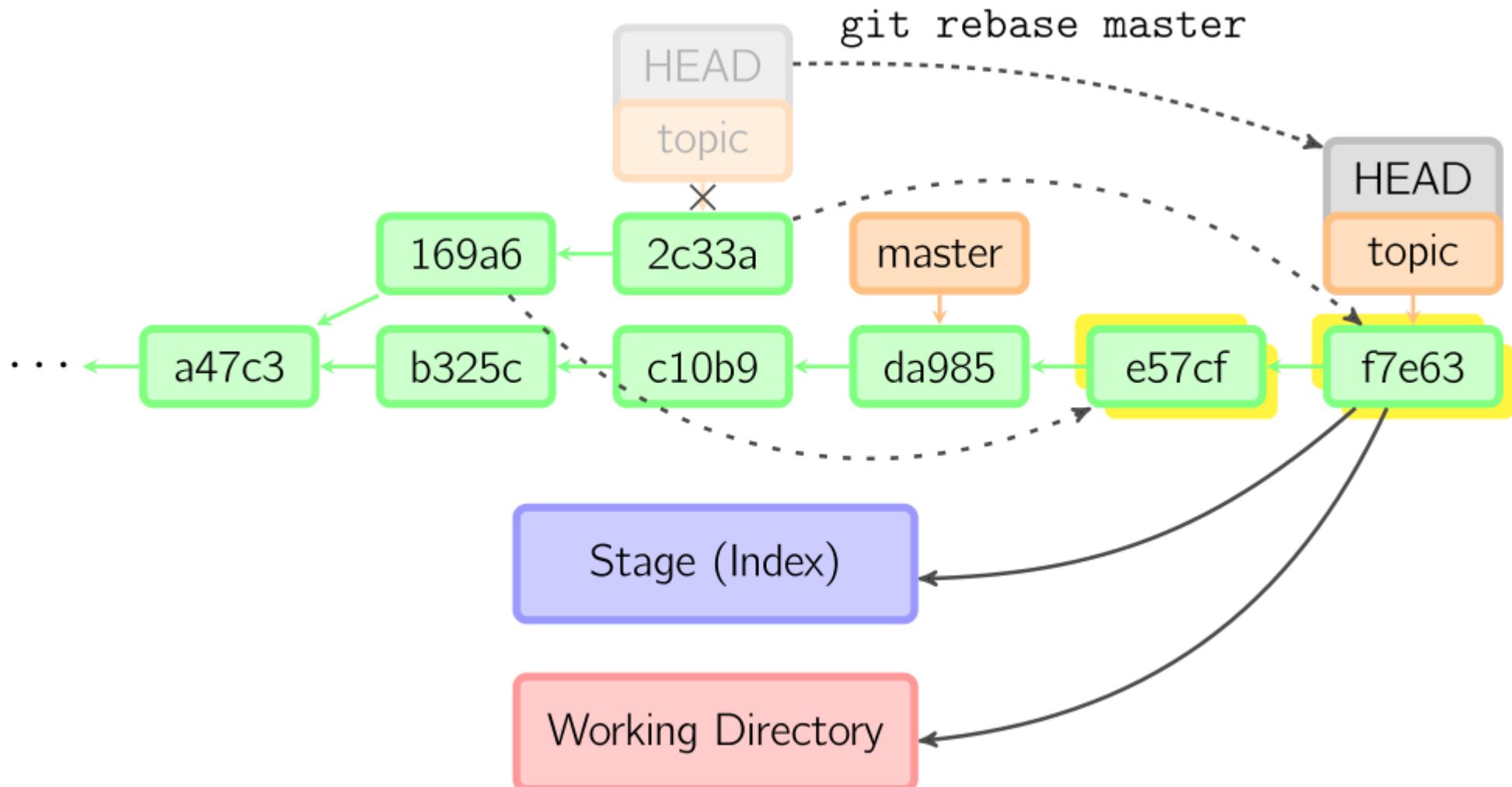
```
<div id="footer">
  please contact us at email.support@github.com
</div>
```

运行一次 git status 来确认所有冲突都已解决

没有冲突的合并图解



使用 rebase 进行合并图解 1

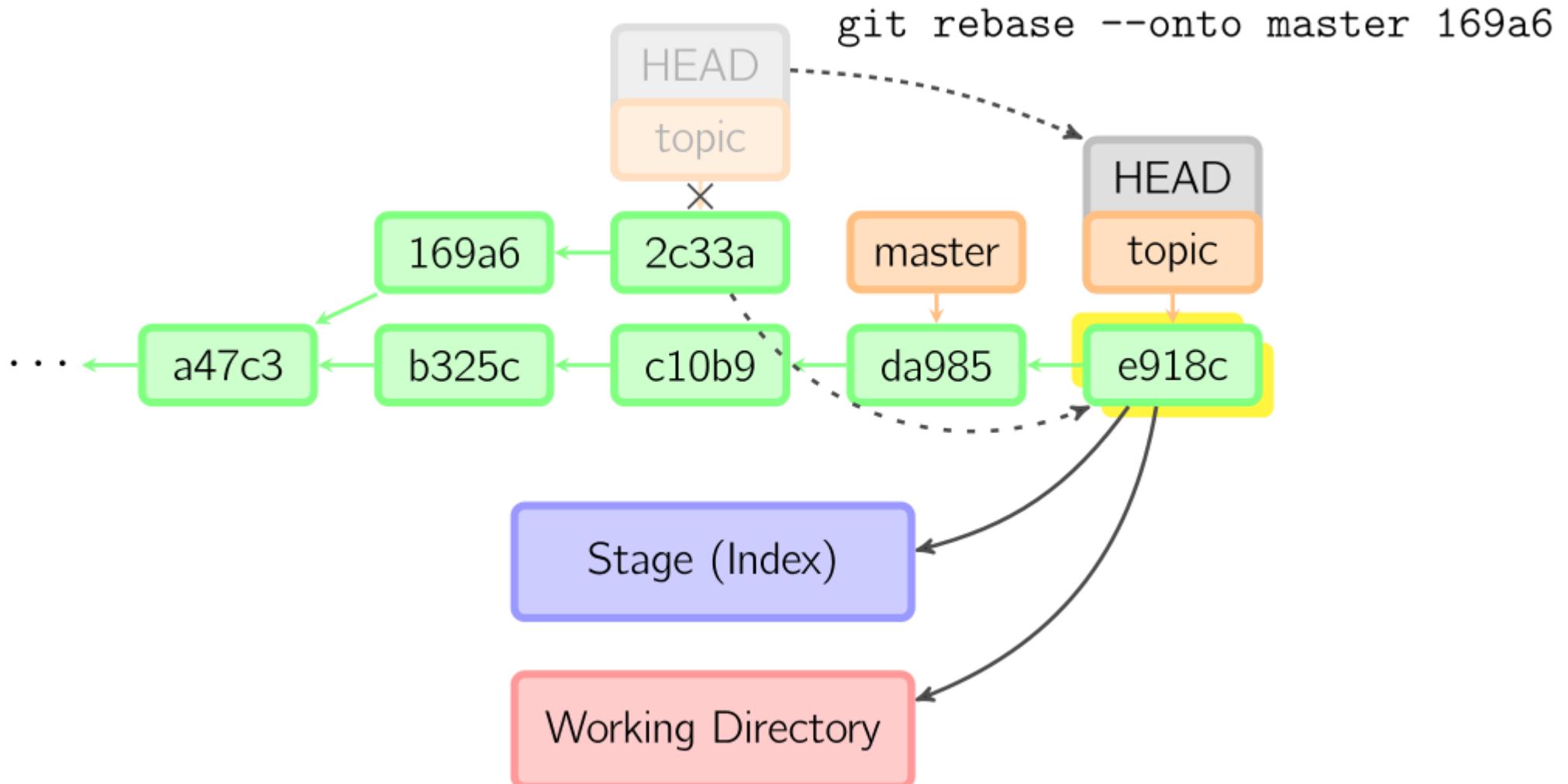


使用 rebase 进行合并图解 1 解释

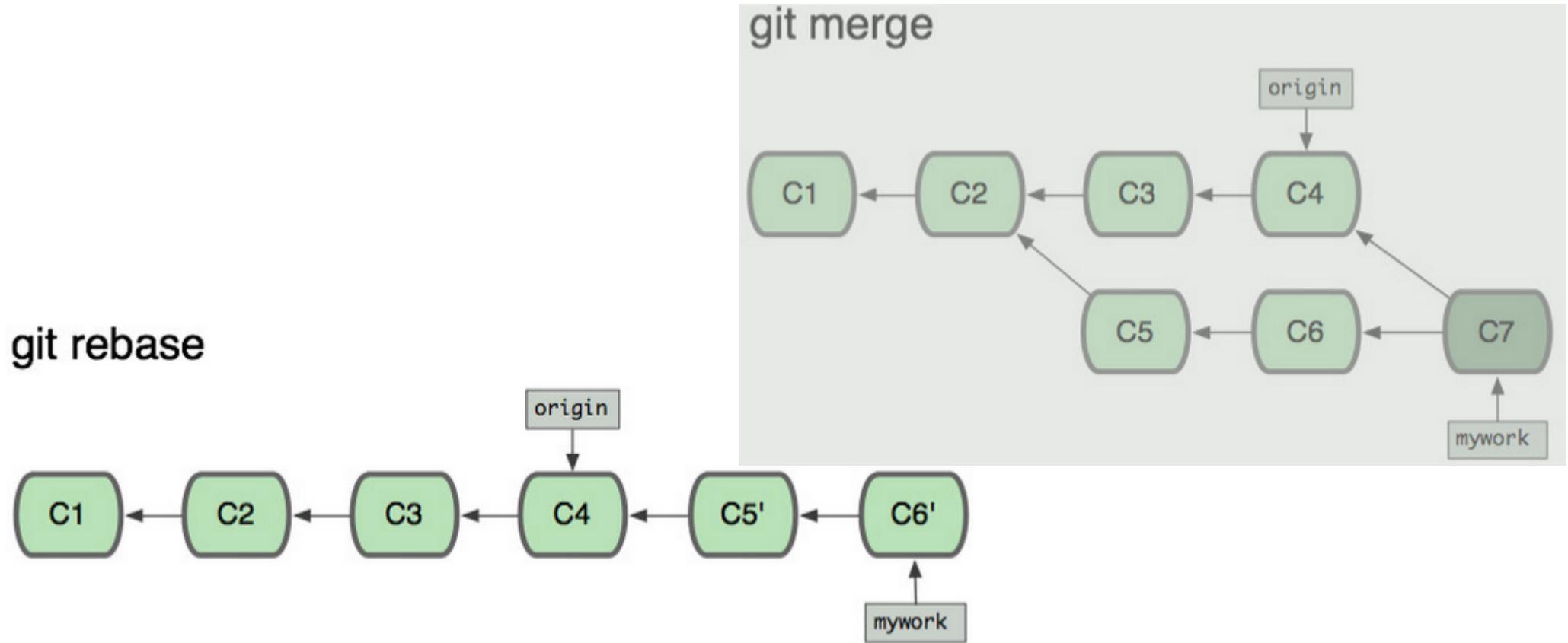
- 在 rebase 的过程中，也许会出现冲突 (conflict)。
在这种情况下，Git 会停止 rebase 并会让你去解决冲突；在解决完冲突后，用 "git-stage" 命令去更新这些内容的索引 (index)，然后，你无需执行 git-commit，只要执行：\$ git rebase --continue 这样 git 会继续应用 (apply) 余下的补丁。
- 在任何时候，你可以用 --abort 参数来终止 rebase 的行动，并且 "mywork" 分支会回到 rebase 开始前的状态。

```
$ git rebase --abort
```

使用 rebase 进行合并图解 2



合并分支时，使用 git rebase 和 git merge 的区别



当我们使用 **Git log** 来参看 **commit** 时，其 **commit** 的顺序也有所不同。

假设 **C3** 提交于 **9:00AM**, **C5** 提交于 **10:00AM**, **C4** 提交于 **11:00AM**, **C6** 提交于 **12:00AM**,

对于使用 **git merge** 来合并所看到的 **commit** 的顺序（从新到旧）是：**C7 ,C6,C4,C5,C3,C2,C1**

对于使用 **git rebase** 来合并所看到的 **commit** 的顺序（从新到旧）是：**C7 ,C6',C5',C4,C3,C2,C1**

因为 **C6'** 提交只是 **C6** 提交的克隆，**C5'** 提交只是 **C5** 提交的克隆，

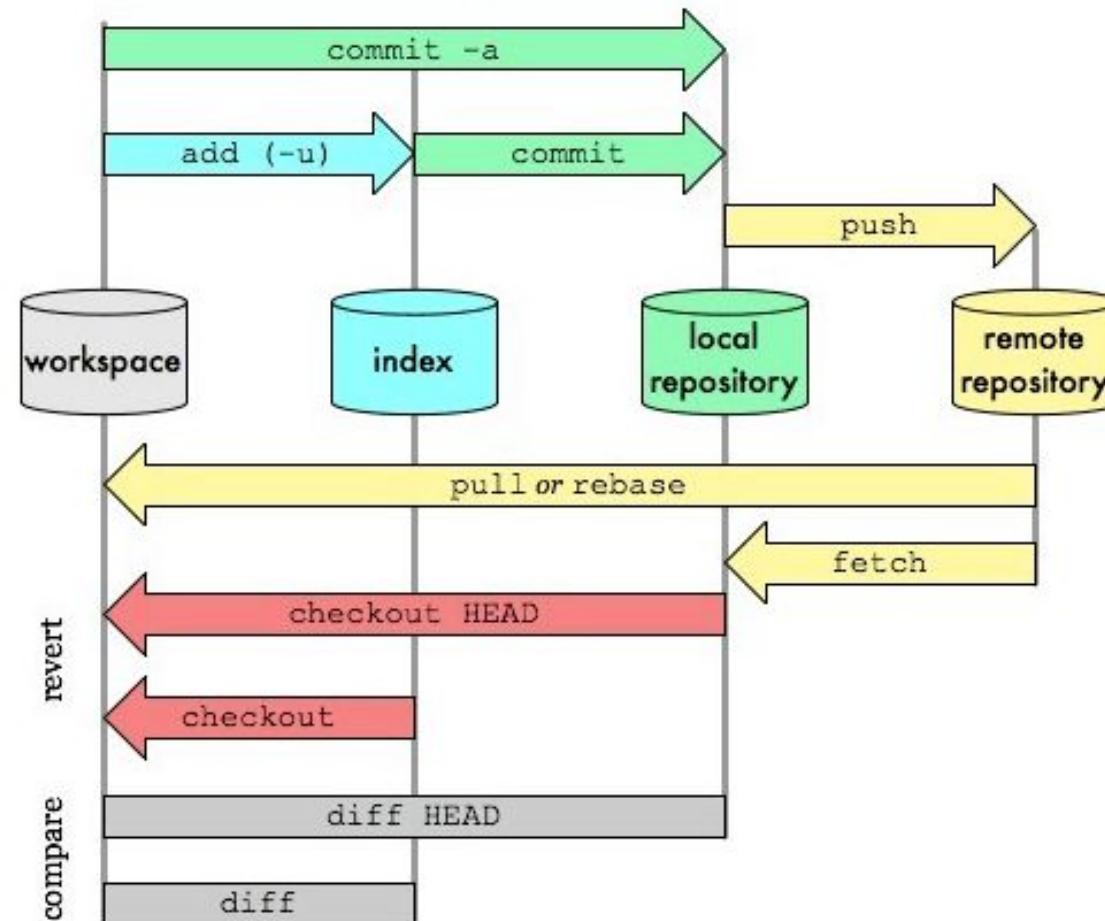
从用户的角度看使用 **git rebase** 来合并后所看到的 **commit** 的顺序（从新到旧）是：**C7 ,C6,C5,C4,C3,C2,C1**

Git 之路

- 版本控制系统
- Git 独奏
- Git 和声

多人协作

- Git 除了单独的版本控制之外，还为项目的多人协作开发提供了很好的支持。此时，Git 作为 DVCS，是开源项目最佳的协作工具。



- 版本控制系统
- Git 独奏
- Git 和声

- Git 和声
 - ✗ Git 远程仓库

概述

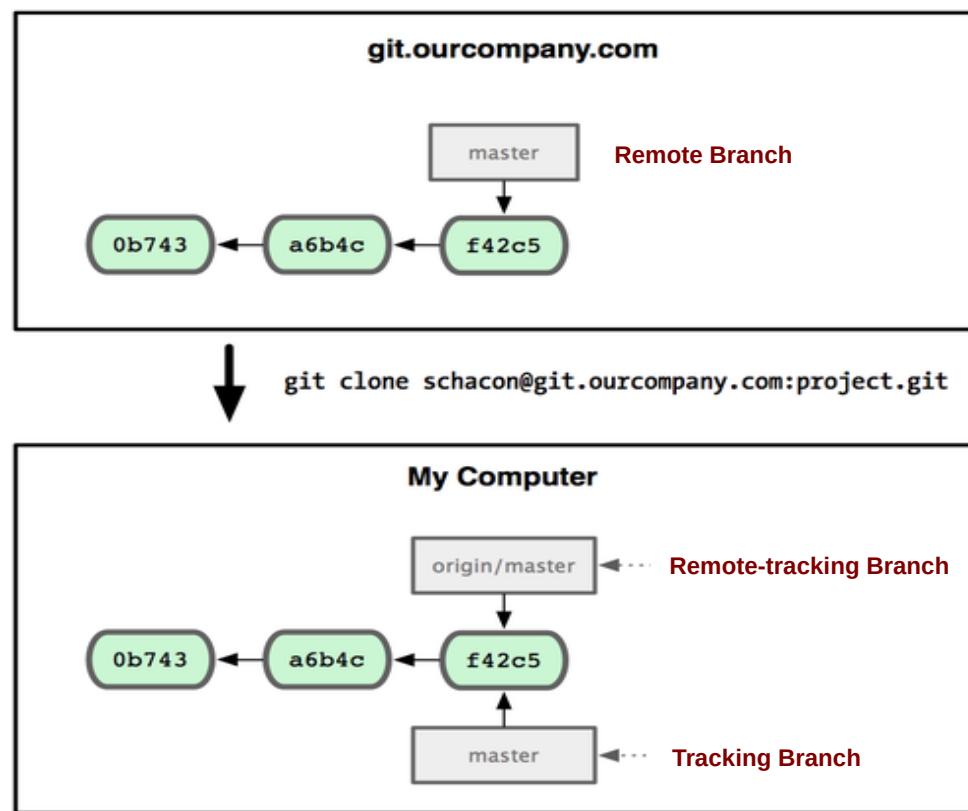
- 要参与任何一个 git 项目的协作，必须要了解该如何管理远程仓库。
远程仓库是指托管在网络服务器上的项目仓库，可能会有好多个，其中有些你只能读，另外有些可以写。同他人协作开发某个项目时，需要管理这些远程仓库，以便推送或拉取数据，分享各自的工作进展。
 - ✗ 一个远程仓库通常只是一个裸仓库 (**bare repository**)——即一个没有当前工作目录的仓库。因为该仓库仅仅作为合作媒介，不需要从磁盘检出快照；存放的只有 Git 的数据。简单的说，裸仓库就是你工作目录内的 .git 目录，不包含其他数据。
- 管理远程仓库的工作，包括添加远程库，移除废弃的远程库，管理各个远程分支，定义是否跟踪这些分支，等等，都需要使用相应的传输协议来进行本地与服务端的通信。

Git 的传输协议

- **本地协议**: 其中的远程版本库就是同一主机上的另一个目录。这常常见于团队每一个成员都对一个共享的文件系统(例如一个挂载的 NFS)拥有访问权, 或者多人共用同一台电脑的情况。
- **SSH 协议**: 架设服务器最常用的传输协议。SSH 协议也是一个验证授权的网络协议; 并且, 因为其普遍性, 架设和使用都很容易。
- **智能 HTTP 协议**: 是最流行的使用 Git 的方式, 它既支持像 git:// 协议一样设置匿名服务, 也可以像 SSH 一样提供传输时的授权和加密。而且只用一个 URL 就可以都做到, 省去了为不同的需求设置不同的 URL。如果你要推送到一个需要授权的服务器上(一般都需要), 服务器会提示你输入用户名和密码。从服务器获取数据时也一样。
- **Git 协议**: Git 使用的网络传输协议里最快的。

深入理解远程库的克隆

- 假设远程仓库有一个默认远程分支 master。如果将其克隆为本地库：
 - git 会自动将远程仓库命名为 origin，并拉下所有的数据，同时创建一个 master 远程分支的指针，即远程跟踪分支 origin/master。
 - 创建一个默认远程分支的本地副本，命名为同名的跟踪分支 master



远程分支、远程跟踪分支

- 前面的 "origin" 是一个远程仓库。该仓库的所有分支被称为远程分支。
- 远程跟踪分支是本地仓库中指向远程仓库中远程分支的指针，因此名字使用了"远程仓库名 / 远程分支名"的形式。
 - × 在克隆时，通常都会在本地仓库中为每个远程分支创建对应的远程跟踪分支，并在远程抓取或推送 (git fetch/git pull 和 git push) 时更新。
 - × 远程跟踪分支就像书签，提醒你在上次网络通信时，相应远程分支位于远程仓库的何处
 - + 这些远程跟踪分支是指向远程分支的远程引用，而不是本地的引用，因此，本地的分支操作没办法使它们移动。
 - + 这些远程引用，在每次与远程仓库进行网络通信时，Git 会移动它们，以确保它们准确地表示了远程仓库的状态。

跟踪分支 (tracking branch)

- 跟踪分支：在签出远程跟踪分支时自动创建的本地分支。

跟踪分支是一种和某个远程分支有直接联系的本地分支。在跟踪分支里输入 `git push`，Git 会自行推断应该向哪个服务器的那个远程分支推送数据。同样，在这些分支里运行 `git pull` 会获取所有远程分支索引，并把它们的数据都合并到本地跟踪分支中来。

- 前面的 `master` 就是基于 `origin/master` 创建的跟踪分支，此时相应的服务器上的 `master` 远程分支是该跟踪分支的上游分支。然而，如果想创建其它远程分支的各个跟踪分支。可以使用命令：

```
git switch -c [branch] [remoteName]/[branch]
```

例如：`git switch -c tracking-dev origin/dev`

- 甚至，在切换分支命令中，如果你试图切换的分支名不存在，并且精确匹配一个远程仓库的一个远程分支名字，Git 会为你自动创建一个跟踪分支：

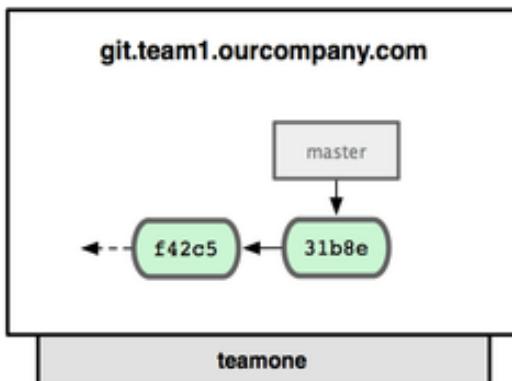
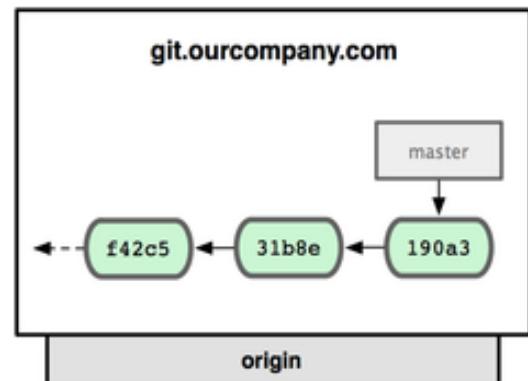
```
git switch dev
```

添加远程仓库

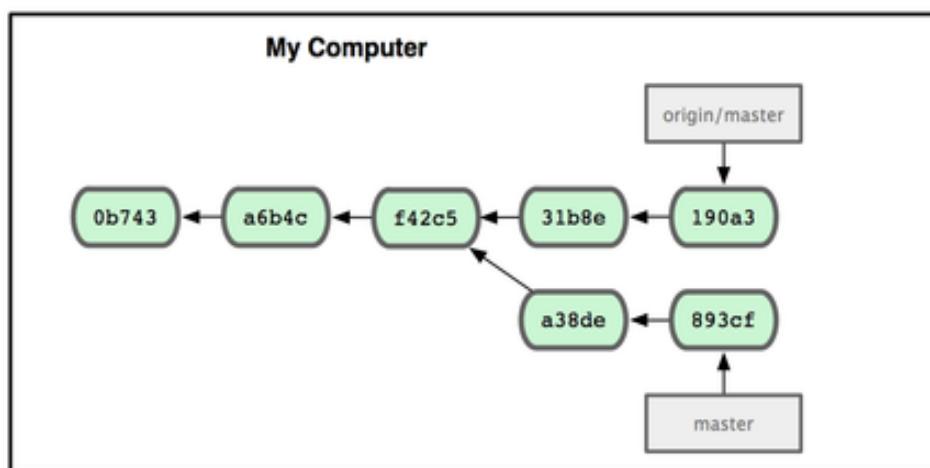
- `git remote [-v]` # 查看当前配置有哪些远程仓库 -v 显示克隆 url
- `git remote add [alias_name] [url]` # 新添加一个远程仓库并取别名
 - ✗ `$ git remote add git_dev https://github.com/git/git.git`

添加远程服务器

- 为了演示具有多个远程服务器和这些远程项目的各个远程分支看起来是什么样子，我们假设你还有另一个仅供你的敏捷小组开发使用的内部服务器 `git.team1.ourcompany.com`。



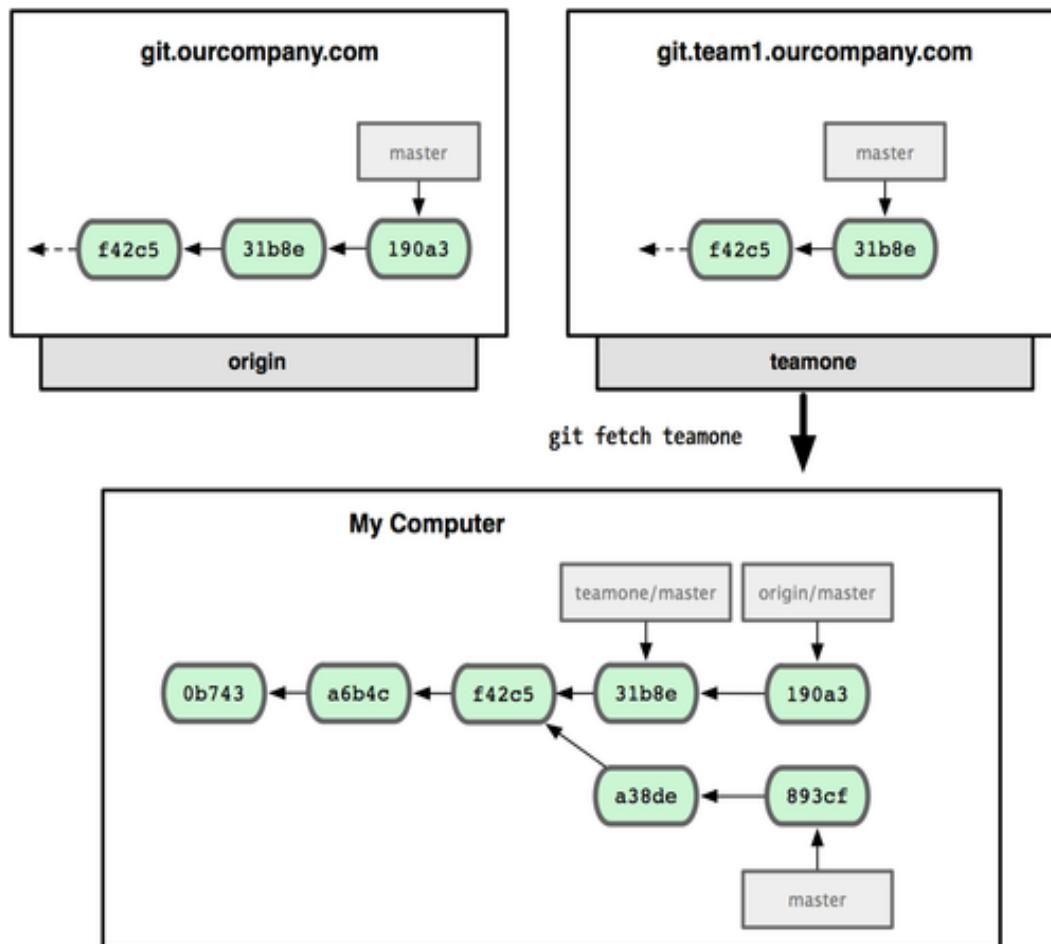
```
git remote add teamone git://git.team1.ourcompany.com
```



你可以通过 `git remote add` 命令将该服务器作为一个新的远端引用加入到你项目中，并将该远程服务器命名为 `teamone`，作为整个 URL 的缩写。

创建新的远程跟踪分支

- 现在可以用 `git fetch teamone` 来获取远程服务器 `teamone` 上有，但你还没有的数据了



但在这个例子中，由于当前该服务器上的内容是你 `origin` 服务器上的子集，Git 不会下载任何数据，而只是简单地创建一个名为 `teamone/master` 的远程跟踪分支，指向 `teamone` 服务器上 `master` 分支所在的提交对象 `31b8e`

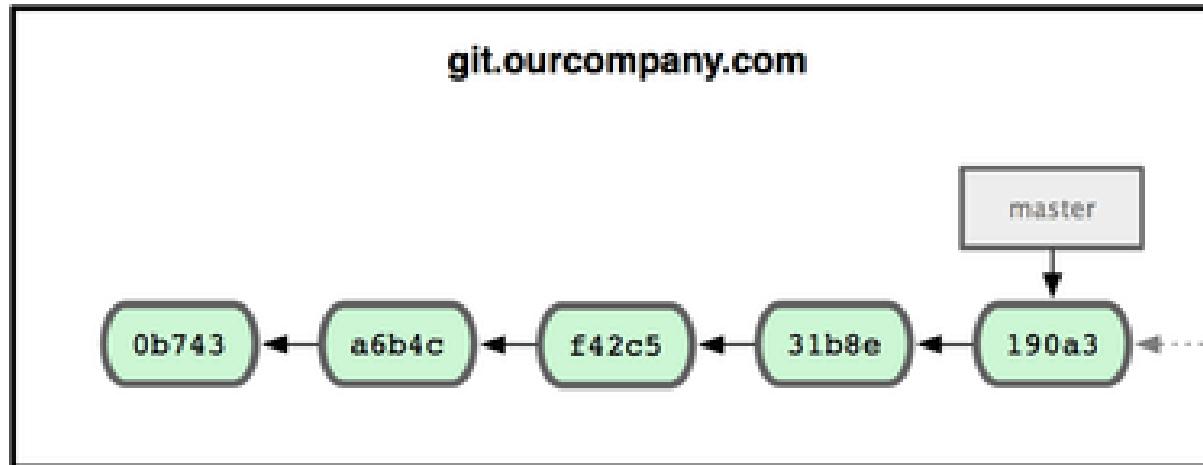
git fetch 用法

- `git fetch [<repository> [<refspec>...]]` // `<refspec>` = `[+]<src>[:<dst>]`

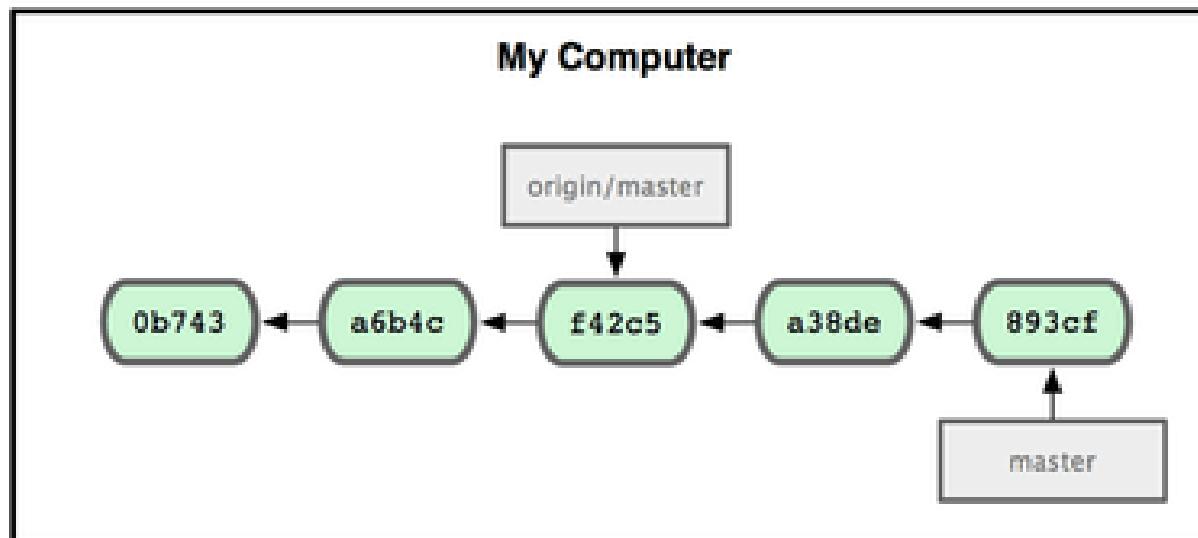
该命令的行为与 `<refspec>` 相关，主要两种用法：

- 如果没有指定参数 `<refspec>`，则从另一个仓库 `<repository>` 下载所有远程分支的对象和引用（若没有指定 `<repository>`，则默认为 `origin` 远程仓库），然后创建或更新本地的所有远程跟踪分支 `<repository>/*`。
 - + 参见上一页的用法： `git fetch teamone`
- 如果指定了参数 `<refspec>`，则从远程仓库 `<repository>` 的远程分支 `<src>` 获取对象和引用，然后更新对应的远程跟踪分支 `<repository>/<dst>`，并在本地创建新的或者快进合并已有的跟踪分支 `<dst>`
 - + 如果 `<dst>` 省略，则 `<dst>` 默认等于 `<src>`
 - + + 表示强制性的获取，等价于 `--force`

远程分支与本地分支的分别演进

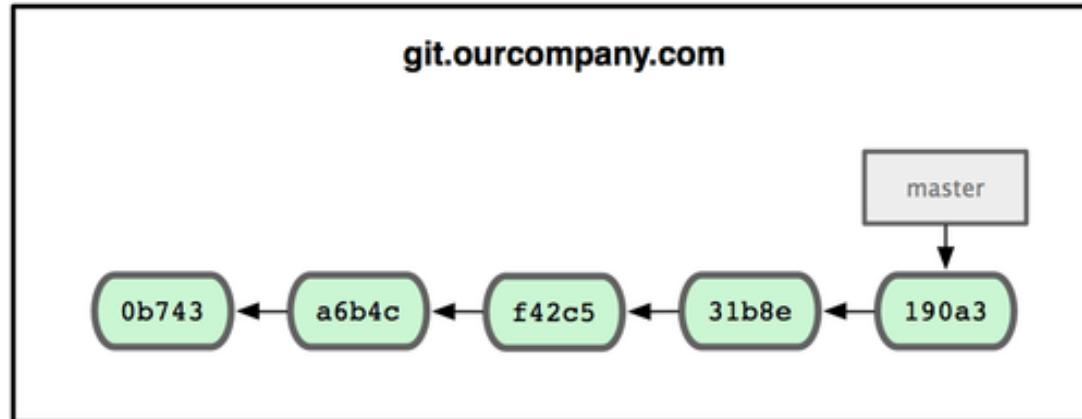


只要你不和服务器通讯，你的 origin/master 指针仍然保持原位不会移动

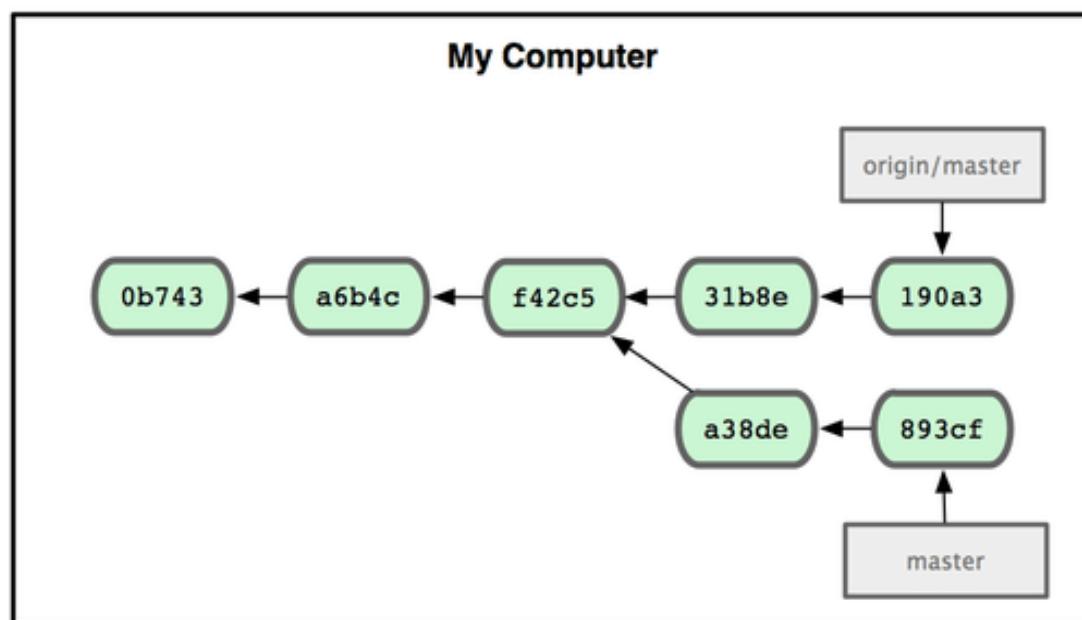


如果你在本地 master 分支做了些改动，与此同时，其他人向 git.ourcompany.com 推送了他们的更新，那么服务器上的 master 分支就会向前推进，而于此同时，你在本地的提交历史正朝向不同方向发展。

Update remote reference



git fetch origin



可以运行 `git fetch origin` 来同步远程服务器上的数据到本地。该命令首先找到 `origin` 是哪个服务器（本例为 `git.ourcompany.com`），从上面获取你尚未拥有的数据，更新你本地的数据库，然后把 `origin/master` 的指针移到它最新的位置上

必须将服务器的更新抓取到本地，然后合并后再推送

`$git switch master`

`$git fetch [origin]`

`$git merge origin/master`

`$git push [origin master:master]`

指定 <refspec> 的 fetch 命令示例

- Using refsspecs explicitly:

```
git fetch origin +seen:seen maint:tmp
```

This updates (or creates, as necessary) branches `seen` and `tmp` in the local repository by fetching from the branches (respectively) `seen` and `maint` from the remote repository.

The `seen` branch will be updated even if it does not fast-forward, because it is prefixed with a plus sign; `tmp` will not be.

- 注意：该命令执行时，当前分支必须是 `seen`、`tmp` 之外的分支，否则会执行失败。

推送本地分支

- 要想和其他人分享某个本地分支，你需要把它推送到一个你拥有写权限的远程仓库。
 - 你创建的本地分支不会因为你的写入操作而被自动同步到你引入的远程服务器上，你需要明确地执行推送分支的操作。
 - 换句话说，对于无意分享的分支，你尽管保留为私人分支好了，而只推送那些协同工作要用到的特性分支。
- 推送本地分支到远程仓库的命令是 `git push`，其基本语法为：

`git push [<repository> [<refspec>...]] // <refspec> = [+]<src>[:<dst>]`

- 该命令将某个本地的跟踪分支 `src` 推送到远程仓库 `<repository>` 的某个远程分支 `dst`。如果 `dst` 省略，则更新与本地同名的远程分支
- 如果没有指定 `<repository>`，则默认为 `origin` 远程仓库。
- `+` 表示强制性的推送，哪怕 `<dst>` 的更新不是一个 `fast forward` // `Danger`

推送本地分支、删除远程分支

- 把本地的 testing 分支推送到远程仓库 origin 的 master 远程分支上：
 - ✗ `git push origin testing:master`
注意：最好 testing 是 master 的快进合并
- 可以使用 push 来删除远程仓库的远程分支
 - ✗ `git push origin --delete serverfix` // 等价 `git push origin :serverfix`
这样删除了远程仓库的远程分支 serverfix。
- 不带参数的 git push 会默认推送当前分支（称为 simple 方式）；但可以修改配置，使得默认推送所有有对应的远程分支的本地跟踪分支（称为 matching 方式）：
 - ✗ `git config --global push.default matching` // 修改默认方式为 matching

远程协作示例：创建远程分支

- 如果远程仓库没有远程分支 serverfix，则可以把 serverfix 本地分支推送到服务器上可以创建同名的远程分支
 - × `git push origin serverfix`

这里其实走了一点捷径。Git 自动把 serverfix 分支名扩展为 `refs/heads/serverfix:refs/heads/serverfix`，意为“取出我在本地的 serverfix 分支，推送到远程仓库的 serverfix 分支中去”。

- 如果运行 `git push origin serverfix:serverfix` 也可以实现相同的效果，它的意思是“上传我本地的 serverfix 分支到远程仓库中去，仍旧称它为 serverfix 分支”。
- 若想把远程分支叫作 awesomebranch，还可以使用：
`git push origin serverfix:awesomebranch`

远程协作示例：协作者获取新创建的远程分支

- 接下来，当你的协作者再次从服务器上获取数据时，他们将得到一个新的远程跟踪分支 `origin/serverfix`，并指向服务器上的远程分支 `serverfix`：

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

- 注意，在 `fetch` 操作下载好新的远程分支之后，你仍然无法在本地编辑该远程仓库中的分支。换句话说，在本例中，你不会有一个新的 `serverfix` 分支，有的只是一个你无法移动的 `origin/serverfix` 指针

远程协作示例：协作者创建新远程分支的跟踪分支

- 由于远程分支的内容已被本地仓库的远程跟踪分支标识，如果想要一份自己的 serverfix 来开发，可以在远程跟踪分支的基础上创建出一个跟踪分支：

```
git switch -c serverfix origin/serverfix
```

这会切换到新建的 serverfix 本地跟踪分支，这使得我们可以基于远程跟踪分支 origin/serverfix 继续向前开发。

- 如果要把远程分支的内容合并到当前 master 分支，则可以运行：

```
git merge origin/serverfix
```

git pull 命令主要用法

- `git pull [<repository> [<refspec>...]]` // `<refspec> = [+]<src>[:<dst>]`
- 使用远程仓库 `<repository>` 的 `<src>` 远程分支更新并合并到本地的跟踪分支 `<dst>`。
 - × 如果当前分支落后于远程分支，那么默认情况下，该命令将快进当前分支以匹配远程分支。
 - × 如果当前分支和远程分支已经分离，用户需要指定如何协调合并这些分离的分支（用 `--rebase` 或 `--no-rebase`）。
- 更准确地说，`git pull` 是使用给定的参数运行 `git fetch`，然后根据配置选项或命令行标志，调用 `git rebase` 或 `git merge` 来协调合并分离的分支。

git pull 命令示例 (续)

- git pull

该命令通常等价于 git pull origin , 相当于 git fetch 后跟 git merge FETCH-HEAD 两条命令的结果。即

- ✗ git pull == git pull origin *:origin/*
== git fetch origin *:origin/* && git merge origin/*

这里 git fetch 默认使用的 refspec 与克隆命令中的 refspec 相同，即：

+refs/heads/*:refs/remotes/origin/*

这一步执行的操作：

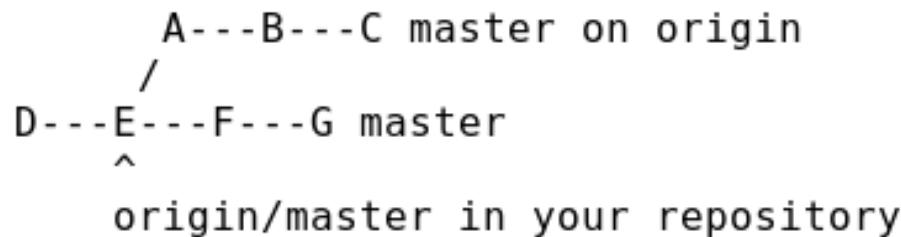
- + 创建或更新所有远程分支对应的远程跟踪分支；
 - + 设定当前分支的 **FETCH_HEAD** 为远程跟踪分支。
- 建议使用 fetch 和 merge 命令，因为 git fetch 之后，你还可以选择不 merge ; 同时 git pull 有时会让人迷惑。

特殊的引用

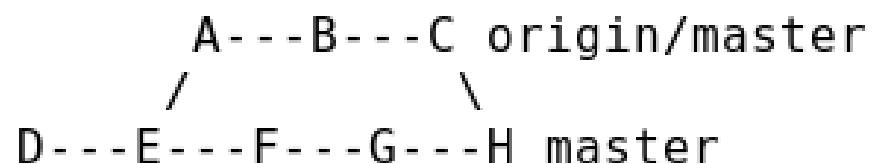
- There are a few special refs, which are upper-case by default:
 - ✗ **HEAD**: this is a pointer to the current branch, i.e. it points to something like "refs/heads/master", usually.
 - ✗ **ORIG_HEAD**: when pulling or merging, ORIG_HEAD refers to the previous revision.
 - ✗ **FETCH_HEAD**: when pulling or fetching, the fetched ref is stored in **FETCH_HEAD**. Note: if you use the convenient "git pull" without argument, chances are that **FETCH_HEAD** contains more than one ref, and it is not really useful.

git pull 示例

- 假设存在以下提交历史，当前分支为 master:



- 执行 git pull 时，git 会在远程跟踪分支 origin/master 获取并重演来自 master 远程分支的变化（从它与本地 master 跟踪分支分叉之后），一直到远程分支 master 的当前提交 C；然后在已经分离了的跟踪分支 master 中创建一个新的 commit 对象 H 中记录合并结果，并标记两个父 commit 对象的名字和用户描述变化的日志信息



- origin/master 在这里是 FETCH_HEAD 指向的远程跟踪分支； master 是 origin/master 的跟踪分支，将会与 origin/master 合并。

- 版本控制系统
- Git 独奏
- Git 和声

- Git 和声
 - ✗ Git 远程仓库
 - ✗ 使用 Github 协作开发

github 的配置

- github 有两种名字：帐号是注册的名字，作为远程推送时的用户名；还可以设置开发者名字。下图的 stardust 是开发者名字，open-src 是帐号名字。
- 在使用 `git push` 推送本地分支到远程仓库时，需要输入用户名和密码，其中用户名就是 github 的帐号，密码是在 github 上配置的个人访问令牌 (PAT)
 - × PAT 在 `settings` 中设置，在作为密码输入时不会回显，只需粘贴一次，然后直接回车就行了，不要以为没有输入成功！
 - × 输入一次 PAT 之后，可以使用下面的命令来缓存 PAT，后续使用就不用输入密码了：

```
git config --global credential.helper cache
```



stardust
open-src

github 的访问和中文文档

- Github 的访问，可以修改 hosts 文件，直接指定其域名的 ip 地址。
 - × 通过网站 <https://websites.ipaddress.com> 可以查询常用的域名的 ip 地址。
 - × 下面是某次查询得到的 hosts 文件中的域名和 ip 地址设置，由于不同的网络，访问的 ip 地址可能不一样，所以在访问缓慢时，需要重新查询 ip 地址。

140.82.112.4 github.com

199.232.69.194 github.global.ssl.fastly.net

- Github 目前正在创建中文帮助文档，下面是网址：
 - × <https://docs.github.com/cn/get-started/quickstart/set-up-git>