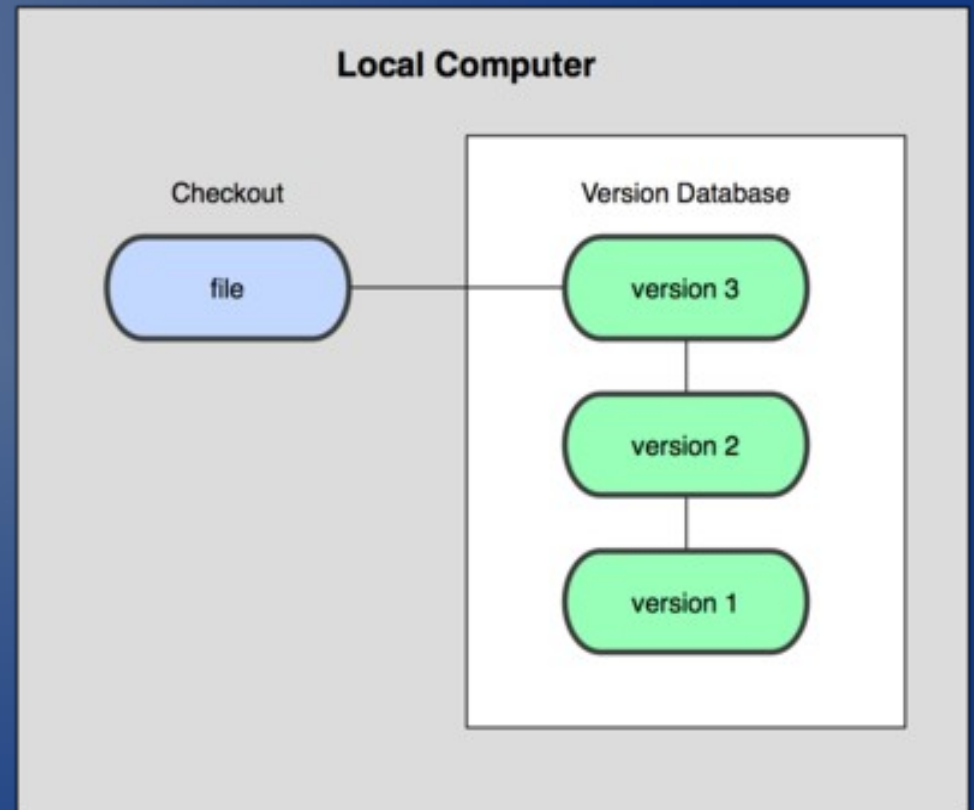


# Git—— 分布式版本控制系统

# 版本控制的发展历程

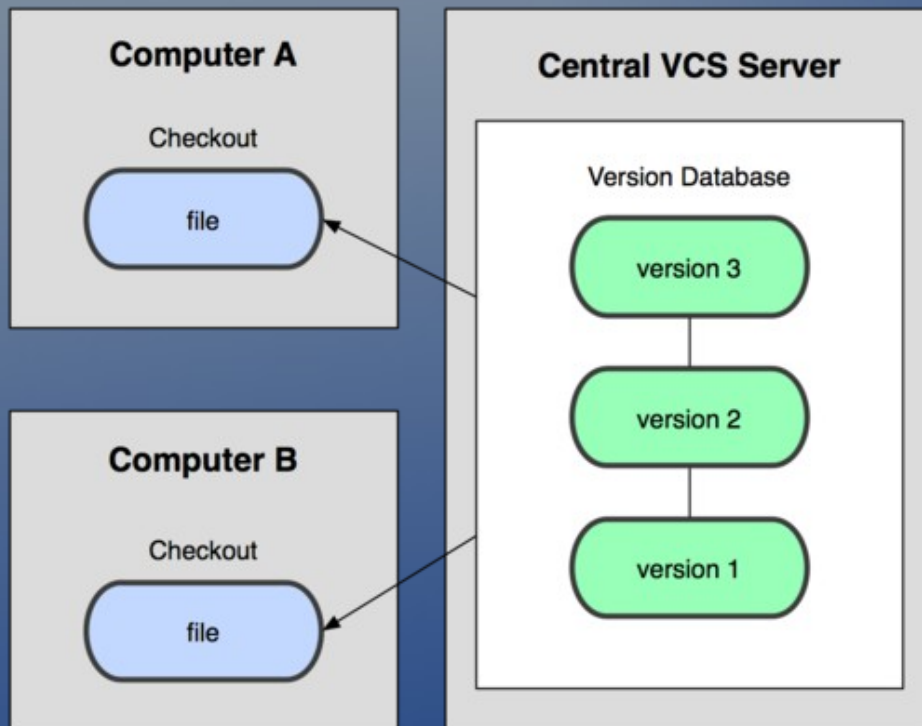
# 本地版本控制系统

- 最流行的 rcs
- 保存并管理文件补丁  
( patch )
- 文件补丁是一种特定格式的文本文件，记录着对应文件修订前后的内容变化。



# 集中化的版本控制系统

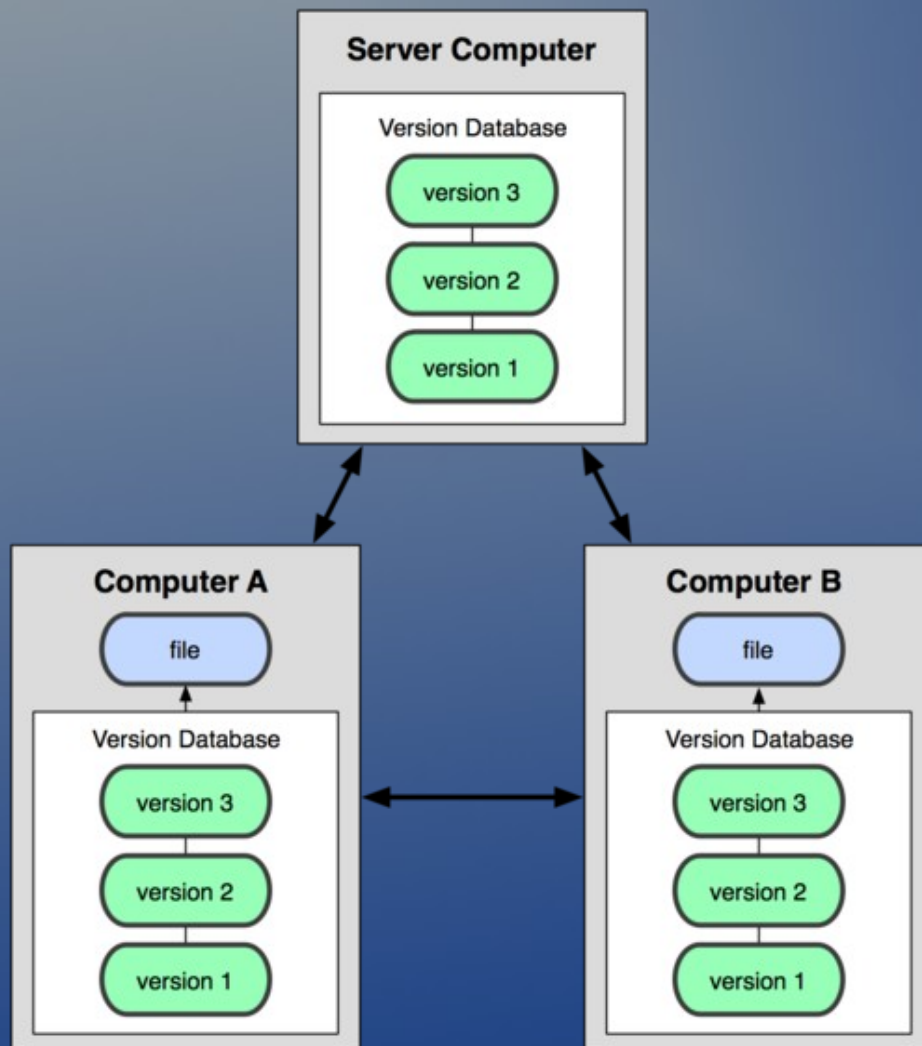
Centralized Version Control Systems



- CVS, Subversion
- 中央服务器的单点故障，整个项目的历史记录被保存在单一位置

# 分布式版本控制系统

Distributed Version Control System



- Git , Mercurial , Bazaar 还有 Darcs
- 客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来
- 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

# Git 特征

# Git 开发的目标

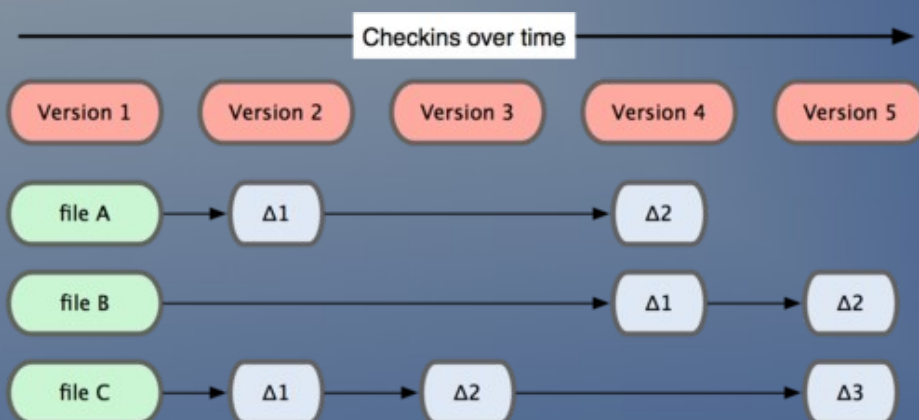
- 速度
- 简单的设计
- 对非线性开发模式的强力支持（允许上千个并行开发的分支）
- 完全分布式
- 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

# Git 基础要点

- 直接快照，而非比较差异
- 近乎所有操作都可本地执行（历史更新摘要）
- 时刻保持数据完整性（内容的校验和 checksum 计算，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，作为指纹字符串。该字符串由 40 个十六进制字符（0-9 及 a-f）组成）
- 多数操作仅添加数据
- 三种状态：已提交（committed），已修改（modified）和已暂存（staged）

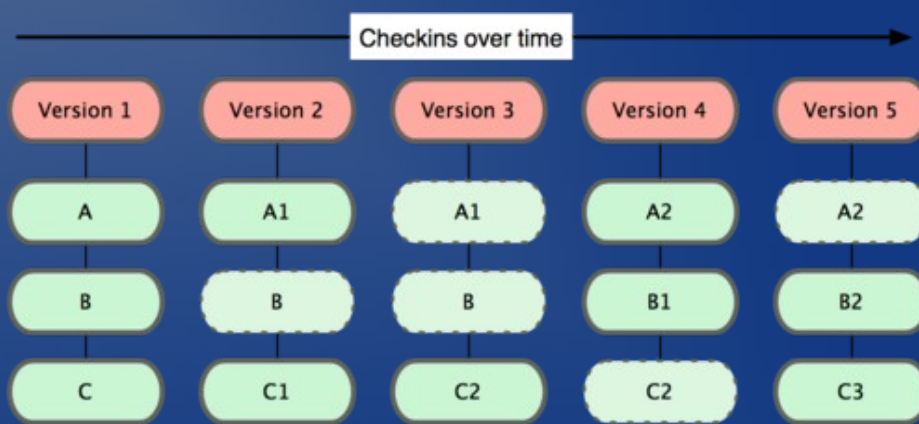


Git 更像是个小型的文件系统，但它同时还提供了许多以此为基础的超强工具，而不只是一个简单的 VCS

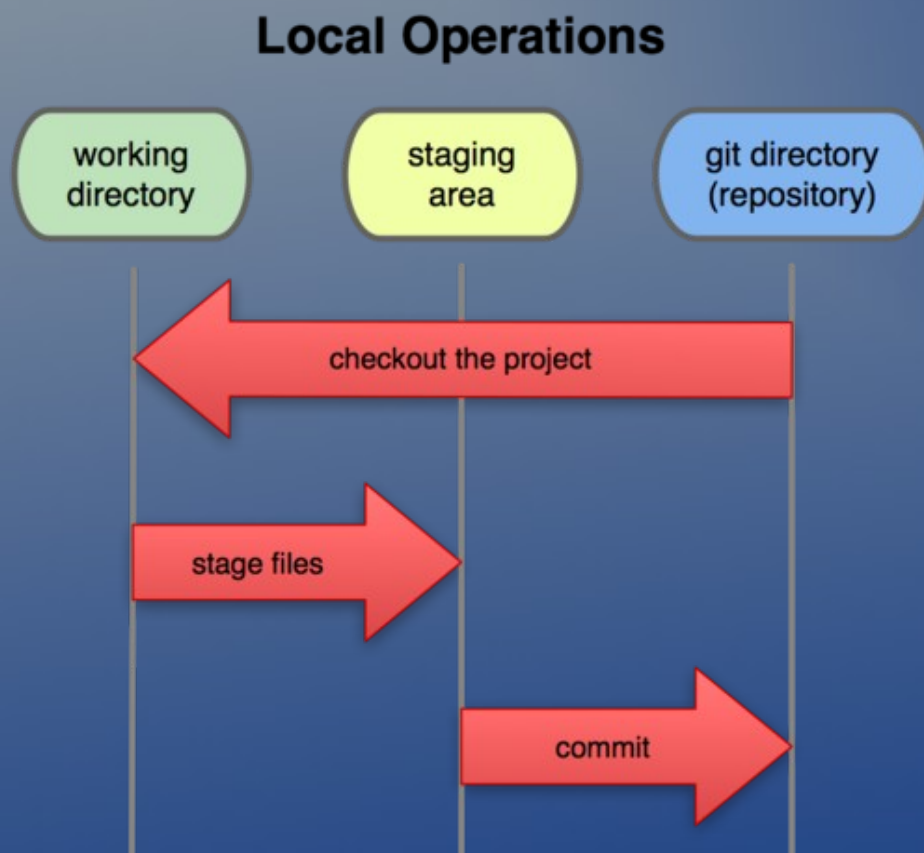


- 其他系统在每个版本中记录着各个文件的具体差异

- Git 保存每次更新时的文件快照
- 若文件没有变化，Git 不会再次保存，而只对上次保存的快照作一连接



# 对于任何一个文件，在 Git 内都只有三种状态



- 已提交：表示该文件已经被安全地保存在本地数据库中了；
- 已修改：表示修改了某个文件，但还没有提交保存；
- 已暂存：表示把已修改的文件放在下次提交时要保存的清单中。

工作目录，暂存区域和 git 目录

# Git 目录结构

- |-- HEAD      # pointer to your current branch
- |-- config    # your configuration preferences
- |-- description # description of your project
- |-- hooks/    # pre/post action hooks
- |-- index     # index file(staging area)
- |-- logs/     # a history of where your branches have been
- |-- objects/   # your objects (commits, trees, blobs, tags)
- `-- refs/     # pointers to your branches

# Git 常用操作

# Git 的配置文件

- `/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。
- 当前项目的 `git` 目录中的配置文件（也就是工作目录中的 `.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` 里的配置会覆盖 `/etc/gitconfig` 中的同名变量。

# Git 配置范例

- `$ git config --global user.name "Wurui"`
- `$ git config --global user.email wurui@csdn.net`
- `$ git config --global core.editor vim`
- `$ git config --global merge.tool vimdiff`
- `$ git config --global alisa.co checkout`
- 列出配置信息：
- `$ git config --list`
- 列出某项配置信息
- `$ git config user.name`

# 获取 Git 帮助信息

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

# Git 常用基本命令

- 建立 git 版本库：进入项目目录，执行

```
$ git init
```

- 将文件加入版本库：

```
$ git add .
```

```
$ git add *.rb
```

```
$ git add README
```

- 提交到版本库

```
$ git commit -m 'initial project version'
```

- 从现有版本库 clone

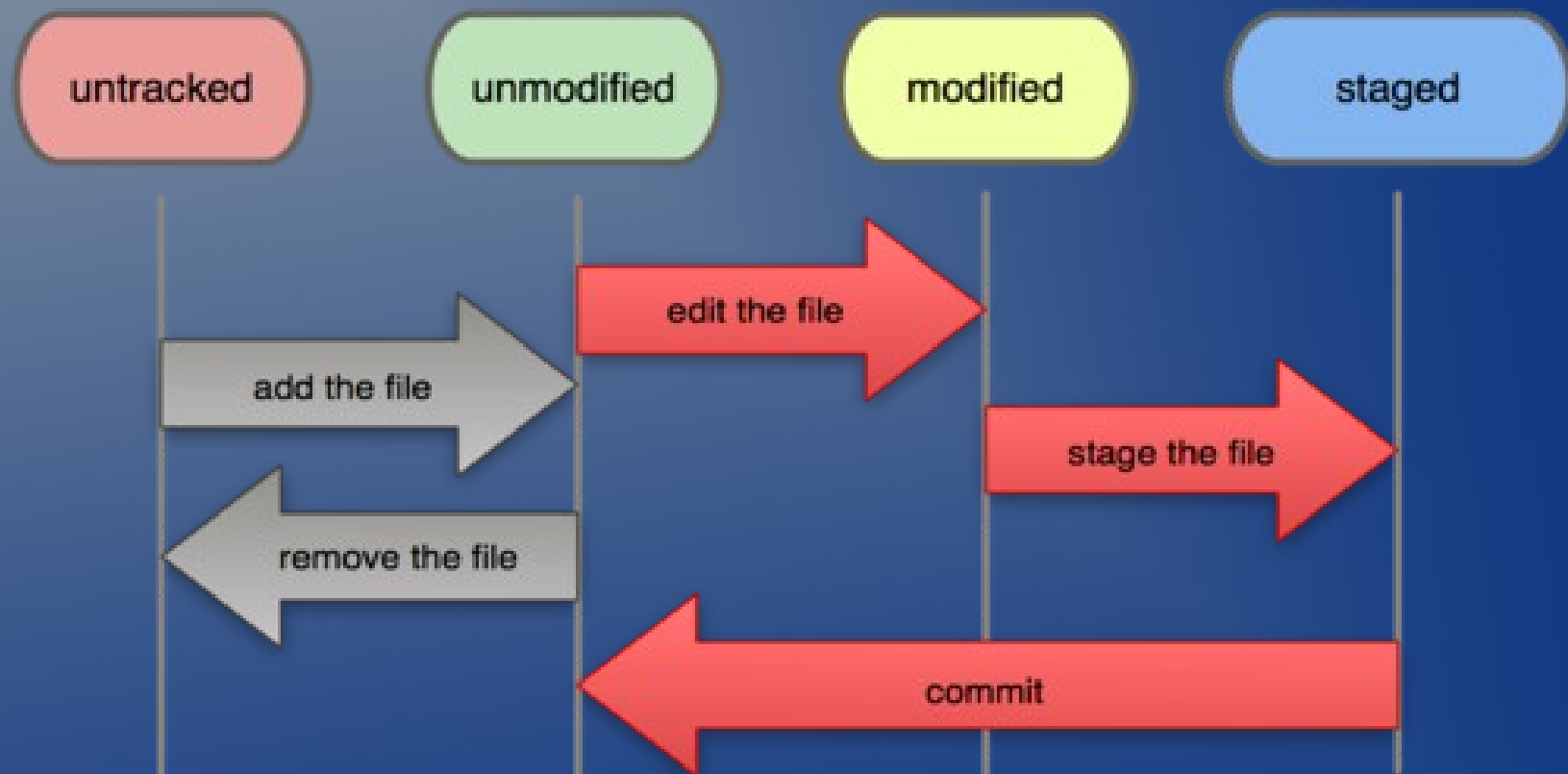
```
$ git clone git://github.com/inosin/study_rails3.git myrails3
```

本地项目目录，可选



# Git 中文件的状态变化周期

## File Status Lifecycle



# 查看 Git 中的文件状态

- \$ git status

```
File Edit View Terminal Help
[inosin@wurui study_git]$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   test-2
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   test-1
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       test-1~
[inosin@wurui study_git]$
```

# 忽略某些文件

- 项目目录下的文件 `.gitignore`
- 文件 `.gitignore` 的格式规范如下：

所有空行或者以注释符号 `#` 开头的行都会被 Git 忽略。

可以使用标准的 glob 模式匹配。\* 匹配模式最后跟反斜杠（/）说明要忽略的是目录。\* 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（!）取反。

```
db/*.sql
```

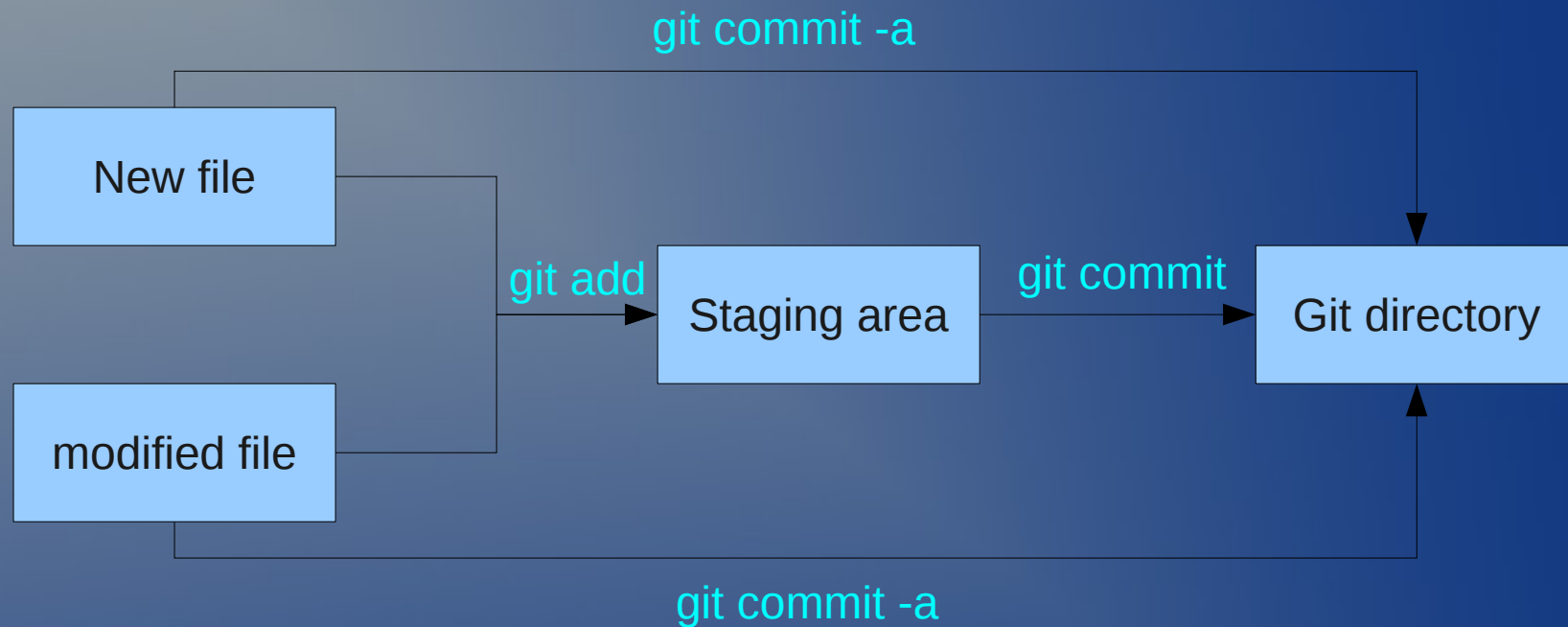
```
/log/*.log
```

```
tmp/*/
```

```
*~
```

```
.project
```

# 跳过使用暂存区域



给 `git commit` 加上 `-a` 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 `git add` 步骤

# 移除文件

- 删除本地和版本库文件

```
$ git rm testfile
```

- 只删除版本库中的文件

```
$ git rm --cache testfile
```

- 删除本目录下 ~ 结尾的文件

```
$ git rm *~
```

- 递归删除项目目录下所有 ~ 结尾的文件

```
$ git rm \*~
```

# 移动文件

- `$ git mv testfile testfile2`

等价于：

- `$ mv testfile testfile2`
- `$ git rm testfile`
- `$ git add testfile2`

# 查看提交历史

- `$ git log`
- `$ git log --p -2`
- `$ git log --stat`
- `$ git log --pretty=oneline`
- `$ git log --pretty=format:"%h - %an, %ar : %s"`

# log 常用的格式占位符写法及其代表的意义

- 选项 说明
- `%H` 提交对象 ( commit ) 的完整哈希字符串
- `%h` 提交对象的简短哈希字符串
- `%T` 树对象 ( tree ) 的完整哈希字符串
- `%t` 树对象的简短哈希字符串
- `%P` 父对象 ( parent ) 的完整哈希字符串
- `%p` 父对象的简短哈希字符串
- `%an` 作者 ( author ) 的名字
- `%ae` 作者的电子邮件地址
- `%ad` 作者修订日期 ( 可以用 `-date=` 选项定制格式 )
- `%ar` 作者修订日期, 按多久以前的方式显示
- `%cn` 提交者 ( committer ) 的名字
- `%ce` 提交者的电子邮件地址
- `%cd` 提交日期
- `%cr` 提交日期, 按多久以前的方式显示
- `%s` 提交说明



# log 一些其他常用的选项及其释义

- 选项 说明
- `-p` 按补丁格式显示每个更新之间的差异。
- `--stat` 显示每次更新的文件修改统计信息。
- `--shortstat` 只显示 `--stat` 中最后的行数修改添加移除统计。
- `--name-only` 仅在提交信息后显示已修改的文件清单。
- `--name-status` 显示新增、修改、删除的文件清单。
- `--abbrev-commit` 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。
- `--relative-date` 使用较短的相对时间显示（比如，“2 weeks ago”）。
- `--graph` 显示 ASCII 图形表示的分支合并历史。
- `--pretty` 使用其他格式显示历史提交信息。可用的选项包括 `oneline`，`short`，`full`，`fuller` 和 `format`（后跟指定格式）。

# 撤消操作

- 修改最后一次提交

```
$ git commit --amend
```

- 取消已经暂存的文件

```
$ git reset HEAD <file>...
```

- 取消对文件的修改

```
$ git checkout -- <file>...
```

# 远程版本库使用

- 查看当前的远程库

`$ git remote` 只显示远程版本库名称

`$ git remote -v` 显示远程版本库名称和地址

- 添加远程仓库

`$ git remote add [shortname] [url]`

`$ git remote add origin git@192.168.4.99:inosin/study_git.git`

- 修改远程仓库地址

- `$ git remote set-url origin git@192.168.4.98:inosin/study_git.git`

- 从远程仓库抓取数据（只抓取，不合并）

`$ git fetch [remote-name]`

`$ git fetch origin`

- 从远程仓库抓取并合并分支

```
$ git pull [remote-name]
```

- 推送数据到远程仓库

```
$ git push [remote-name] [branch-name]
```

- 查看远程仓库信息

```
$ git remote show [remote-name]
```

- 远程仓库的重命名

```
$ git remote rename [remote-name]
```

- 远程仓库的删除

```
$ git remote rm [remote-name]
```

# 打标签

- 查看已有的标签

```
$ git tag
```

```
$ git tag -l "v1.1.*"
```

- 新建标签

```
$ git tag v1.1.1 (简易标签)
```

```
$ git tag -a v1.1.1 -m 'my version 1.1.1' (带注释的标签)
```

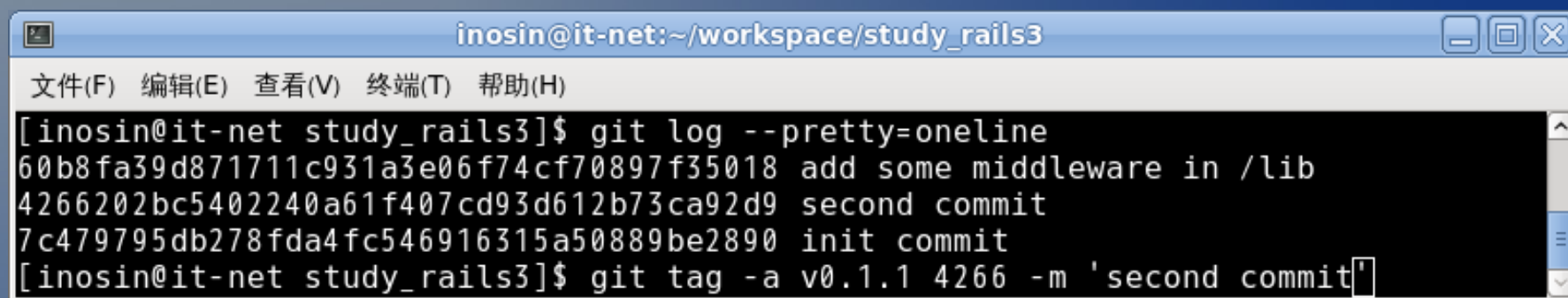
- 查看某标签内容

```
$ git show v1.1.1
```

- 给之前提交的版本打标签

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

```
$ git tag -a v0.1.1 [commit object hash key]
```

A terminal window titled 'inosin@it-net:~/workspace/study\_rails3'. The window has a menu bar with '文件(F)', '编辑(E)', '查看(V)', '终端(T)', and '帮助(H)'. The terminal content shows the following commands and output:

```
[inosin@it-net study_rails3]$ git log --pretty=oneline
60b8fa39d871711c931a3e06f74cf70897f35018 add some middleware in /lib
4266202bc5402240a61f407cd93d612b73ca92d9 second commit
7c479795db278fda4fc546916315a50889be2890 init commit
[inosin@it-net study_rails3]$ git tag -a v0.1.1 4266 -m 'second commit'
```

- 分享标签

默认情况下，git push 并不会把标签传送到远端服务器上，只有通过显式命令才能分享标签到远端仓库。其命令格式如同推送分支，运行 `git push origin [tagname]` 即可

```
$ git push origin v0.1.1    (推送某个标签)
```

```
$ git push origin --tags    (推送所有标签)
```

# 一些小技巧

- 自动完成：`~/.git-completion.bash`

```
$ git co<tab> <tab>
```

```
co commit config
```

```
$ git log --p<tab><tab>
```

```
--parents      --patience      --pickaxe-regex
```

```
--patch-with-stat --pickaxe-all  --pretty=
```

- Yum 安装 git :

加入 “`source /etc/bash_completion.d/git`” 到 `~/.bashrc`

- Git 命令别名

```
$ git config --global alias.co checkout
```

```
$ git config --global alias.br branch
```

```
$ git config --global alias.ci commit
```

```
$ git config --global alias.st status
```

```
$ git config --global alias.unstage 'reset HEAD --'
```

```
$ git config --global alias.last 'log -1 HEAD'
```

- 设置外部命令

```
$ git config --global alias.visual '!gitk'
```

- 运行别名

```
$ git <alias name>
```



- 添加颜色

所有的 color.\* 选项请参见 git config 的文档

```
$ git config color.branch auto
```

```
$ git config color.diff auto
```

```
$ git config color.interactive auto
```

```
$ git config color.status auto
```

- 或者你可以通过 color.ui 选项把颜色全部打开：

```
$ git config color.ui true
```

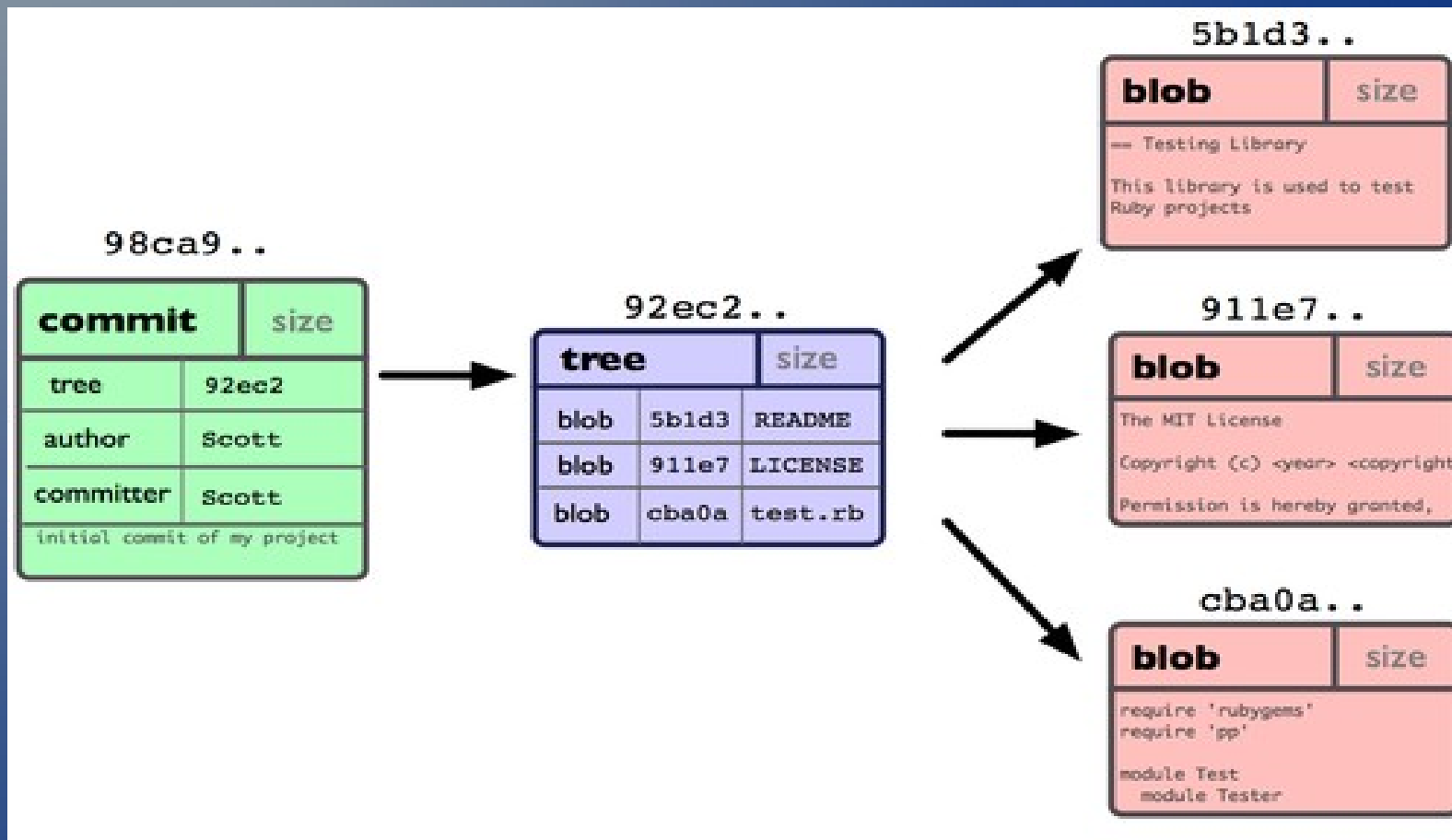
# Git 分支 Branch

# Git 是如何储存数据

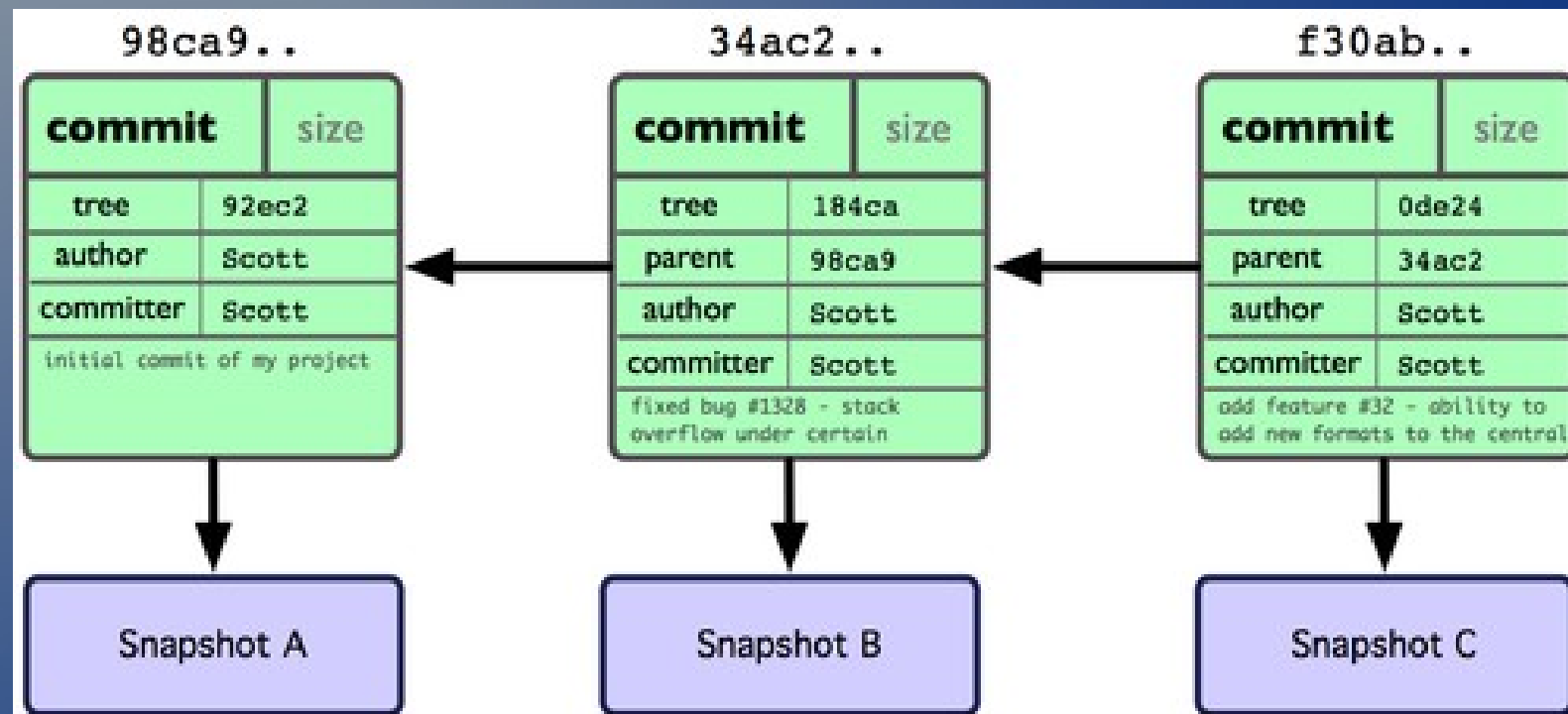
- Git 保存的不是文件差异或者变化量，而只是一系列文件快照。
- 在 Git 中提交时，会保存三类对象：
  - 一个提交（commit）对象：包含指向 tree 对象的索引和其他提交信息元数据
  - 树（tree）对象：记录着目录树内容及其中各个文件对应 blob 对象索引
  - blob 对象：保存文件快照内容

```
$ git add README test.rb LICENSE2
```

```
$ git commit -m 'initial commit of my project'
```

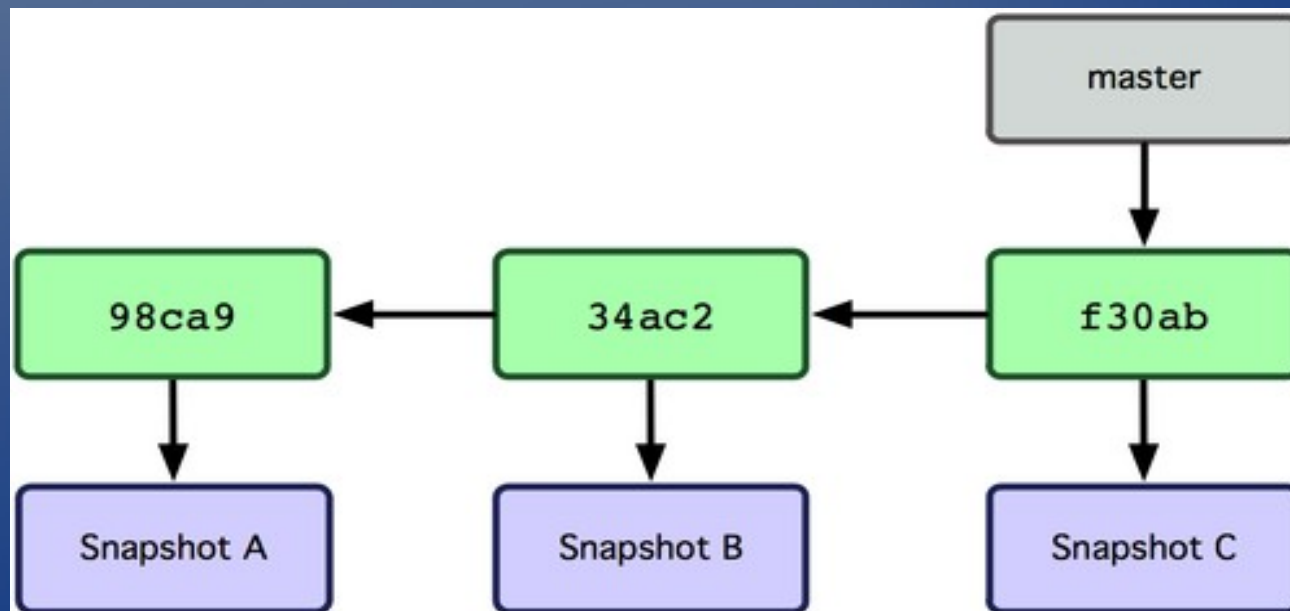


- 三次提交示意图



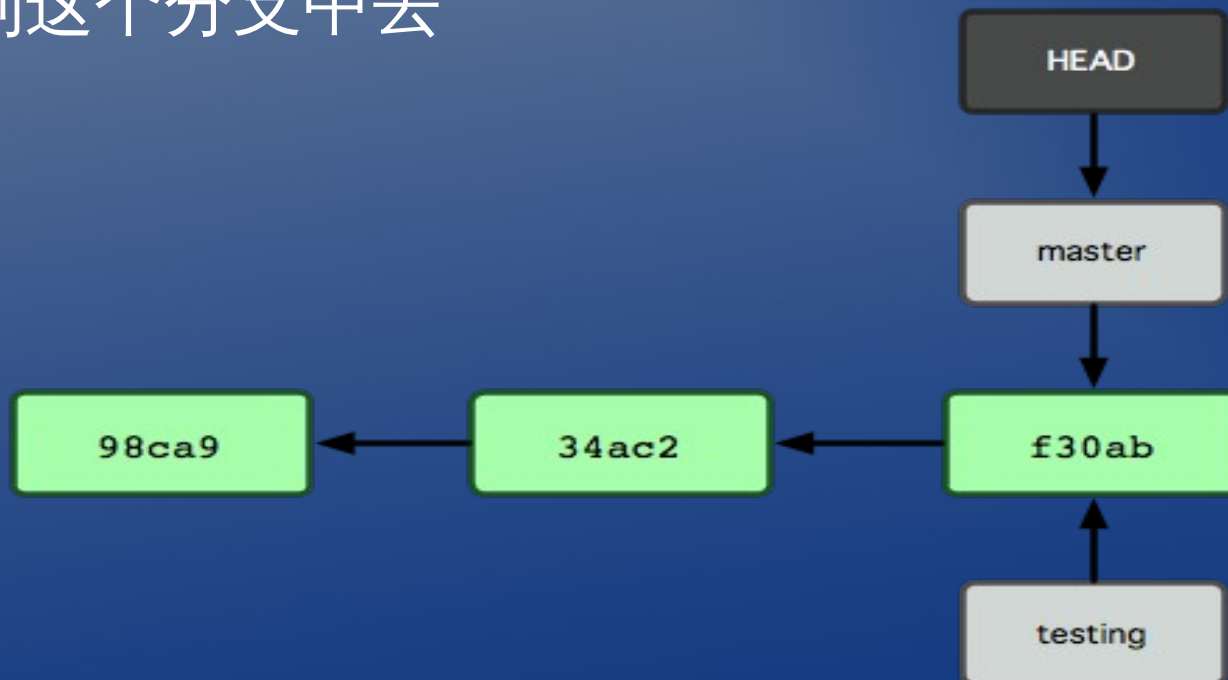
# Git 中的分支是指针

- Git 中的分支，其实本质上仅仅是个指向 commit 对象的可变指针
- Git 会使用 master 作为分支的默认名字
- 在每次提交的时候都会自动向前移动



# 创建一个新的分支

- `$ git branch testing`
- HEAD 特别指针：是一个指向你正在工作中的本地分支的指针
- `git branch` 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去



# 查看和切换分支

- 查看分支

`$ git branch`

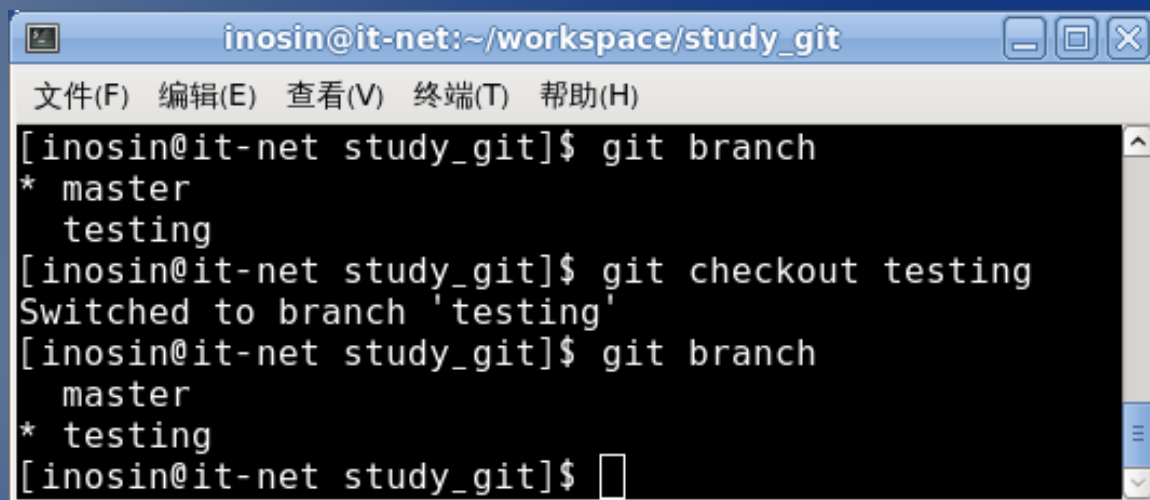
- 切换分支

`$ git checkout testing`

- 创建并切换到新的分支

`$ git checkout -b testing`

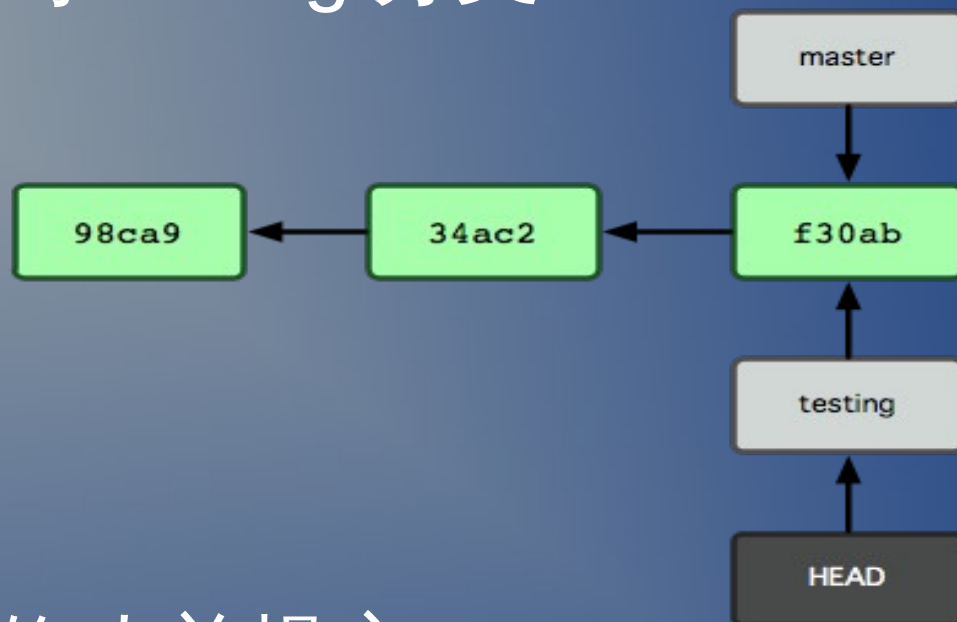
- 分支名前标记 \* 表示现在所在的分支



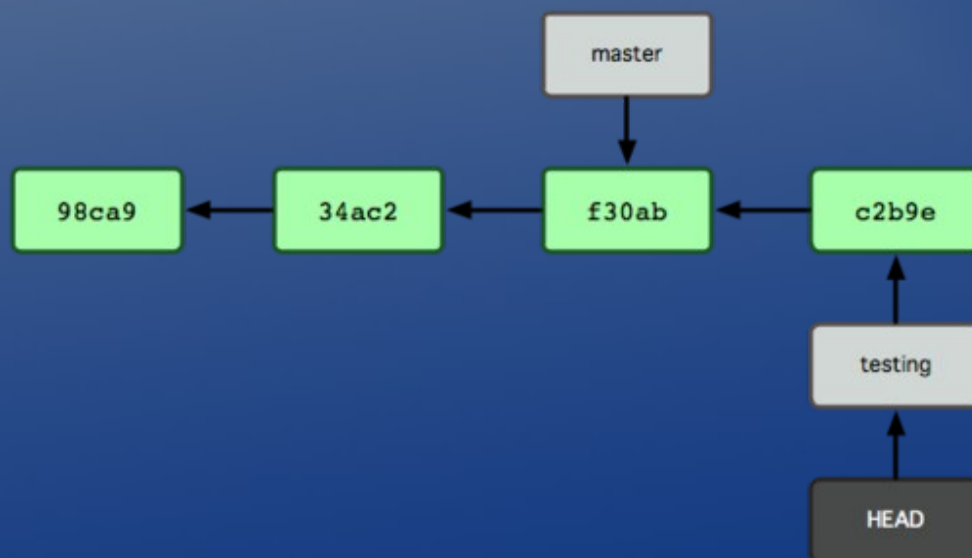
```
inosin@it-net:~/workspace/study_git
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
[inosin@it-net study_git]$ git branch
* master
  testing
[inosin@it-net study_git]$ git checkout testing
Switched to branch 'testing'
[inosin@it-net study_git]$ git branch
  master
* testing
[inosin@it-net study_git]$
```



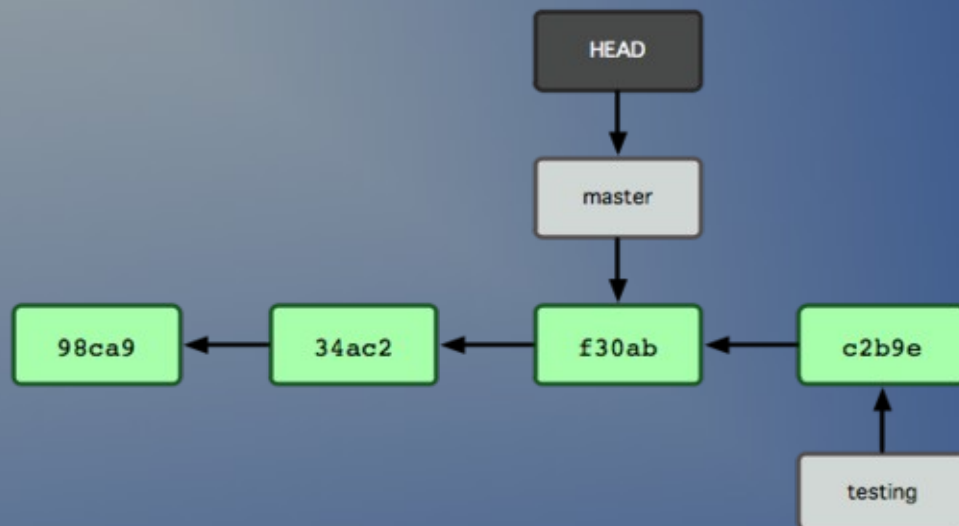
- 切换到 testing 分支



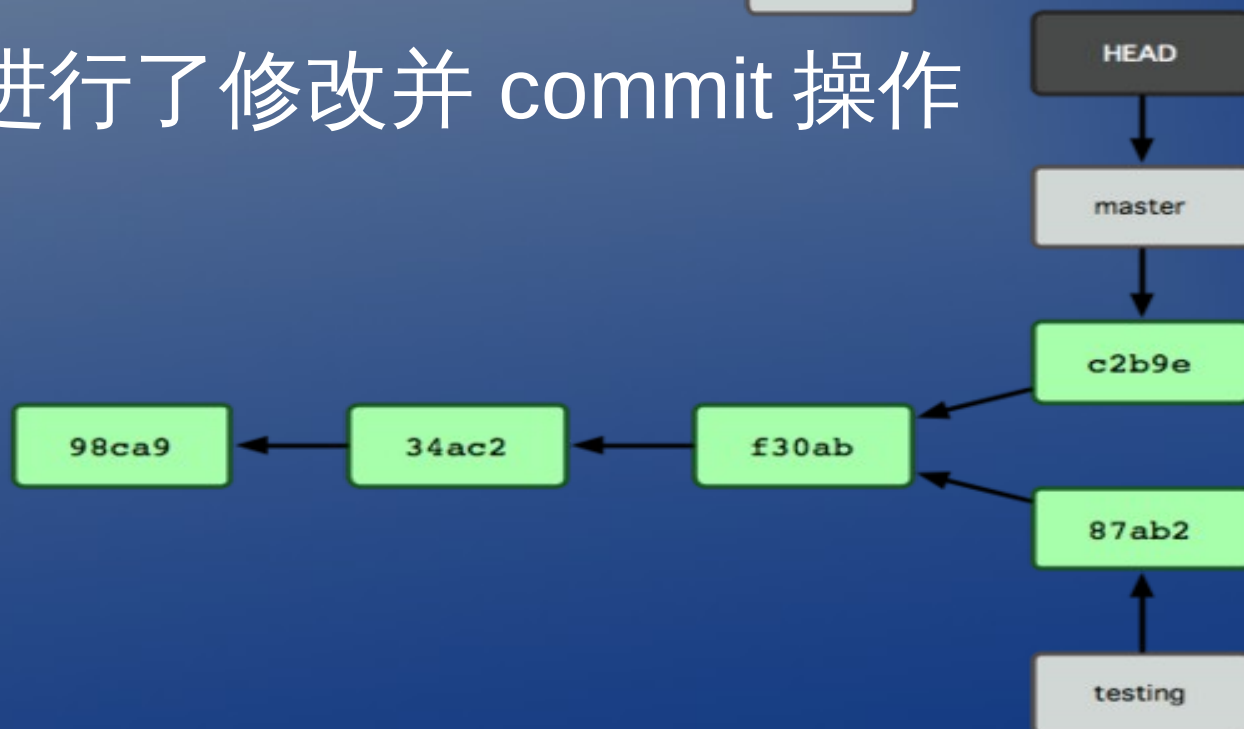
- 进行修改并提交



- 返回到 master 分支



- 进行了修改并 commit 操作

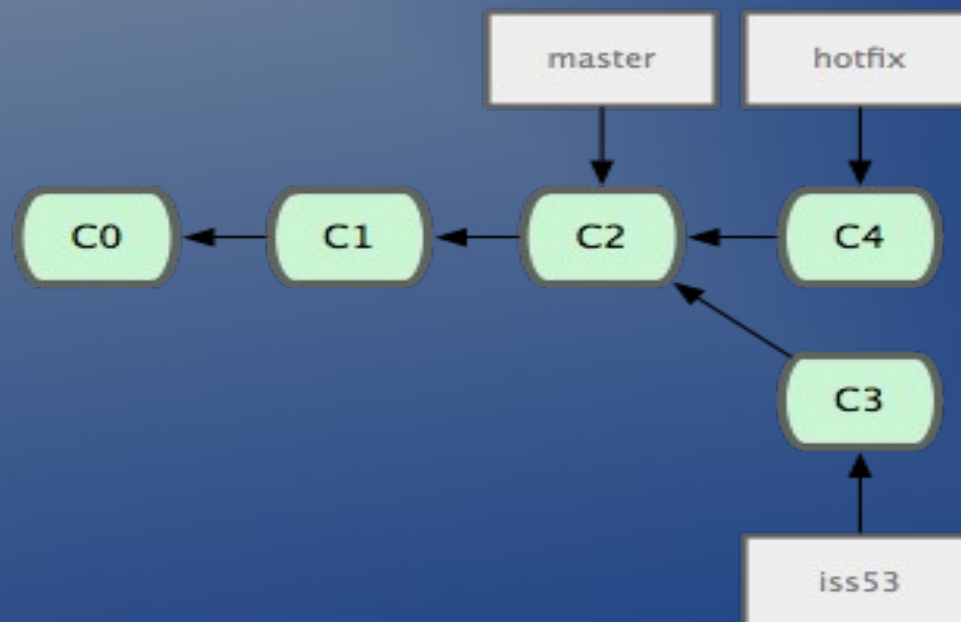


# 分支总结

- Git 中的分支实际上仅是一个包含所指对象校验和（40 个字符长度 SHA-1 字符串）的文件，所以创建和销毁一个分支就变得非常廉价，说白了，新建一个分支就是向一个文件写入 41 个字节（外加一个换行符）那么简单，当然也就很快了。
- Git 的实现与项目复杂度无关，它永远可以在几毫秒的时间内完成分支的创建和切换
- 因为每次提交时都记录了祖先信息（即 parent 对象），所以以后要合并分支时，寻找恰当的合并基础（即共同祖先）的工作其实已经完成了一大半，实现起来非常容易。

# 合并分支

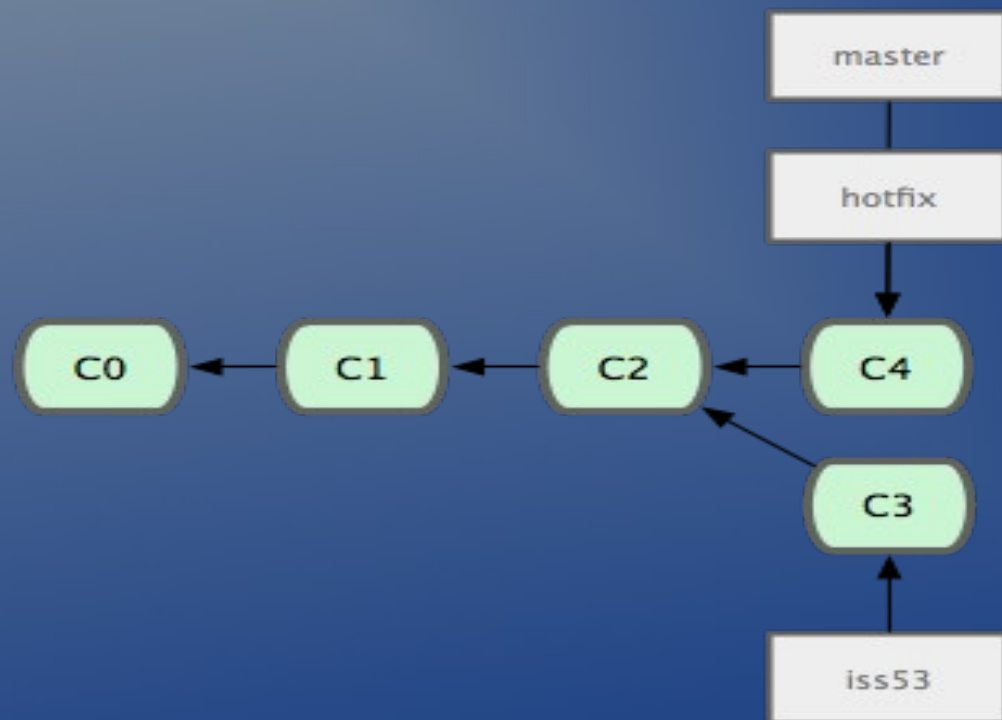
- master、hotfix、iss53 三个分支合并前的状态



- 将 hotfix 分支合并到 master 分支

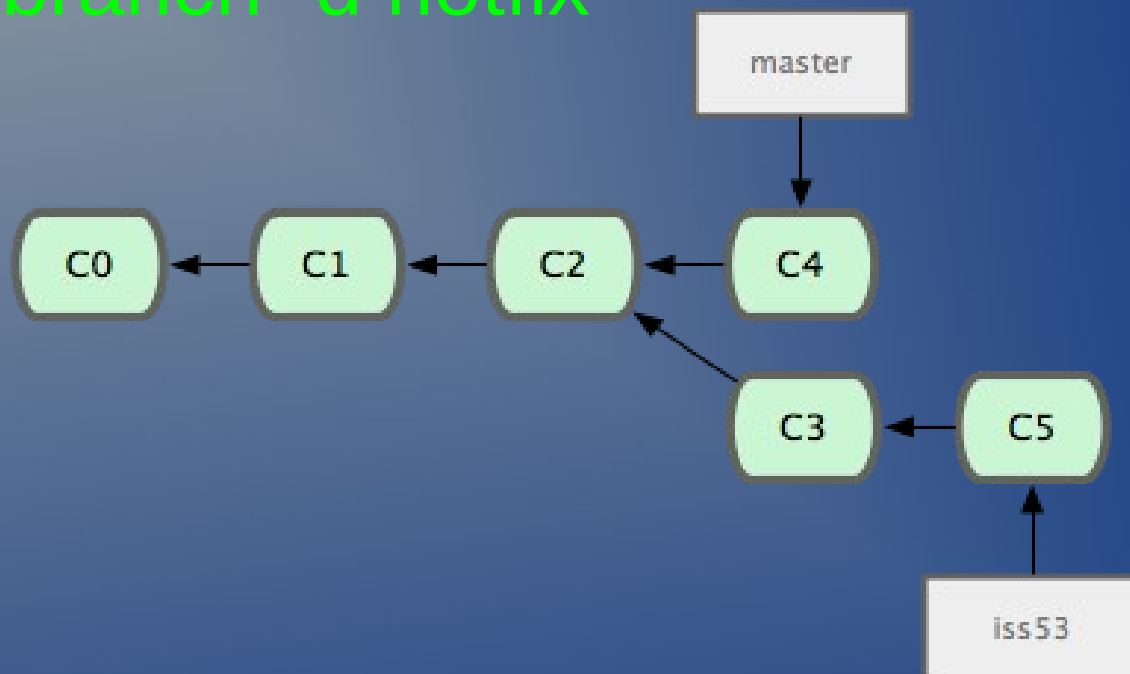
\$ git checkout master

\$ git merge hotfix



- 合并后 hotfix 和 master 都指向同一位置了，hotfix 已经没用了，我们删除它

`$ git branch -d hotfix`

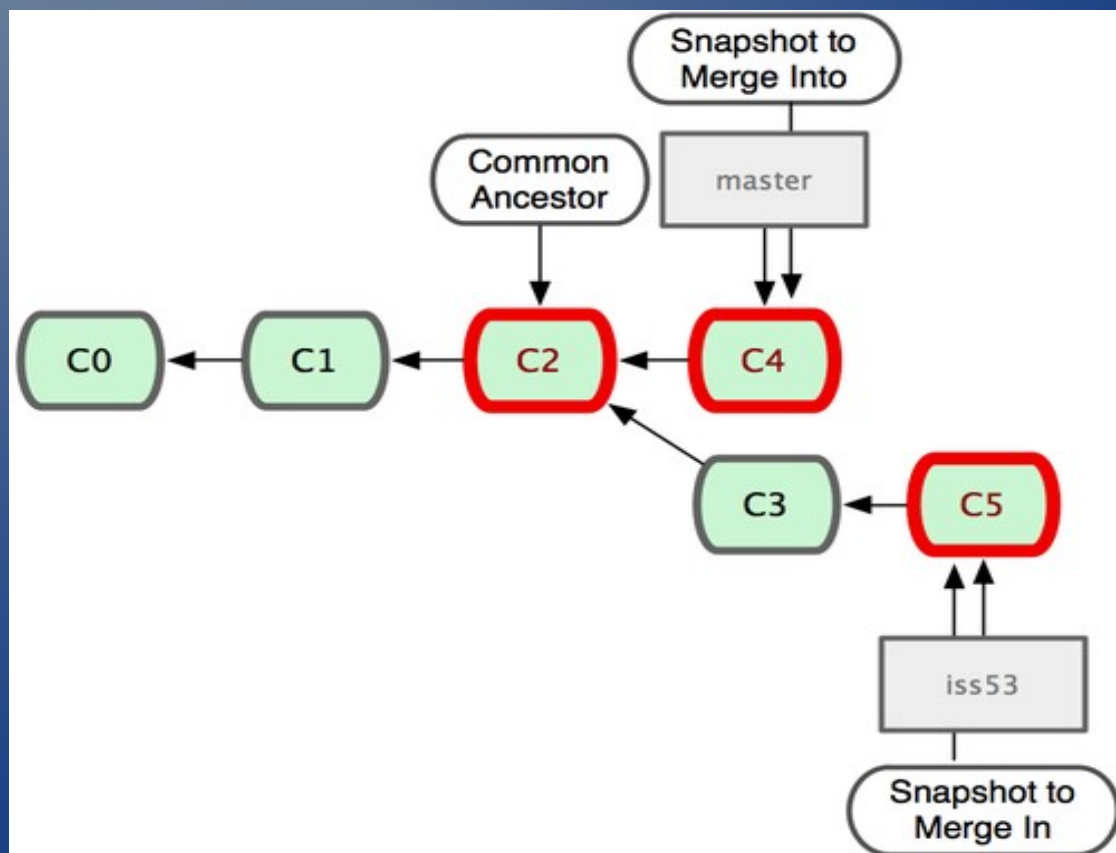


如果需要将之前 hotfix 的内容并入到 issue53 分支中，可以用 `git merge master` 把 master 分支合并到 iss53，或者等完成后，再将 iss53 分支中的更新并入 master

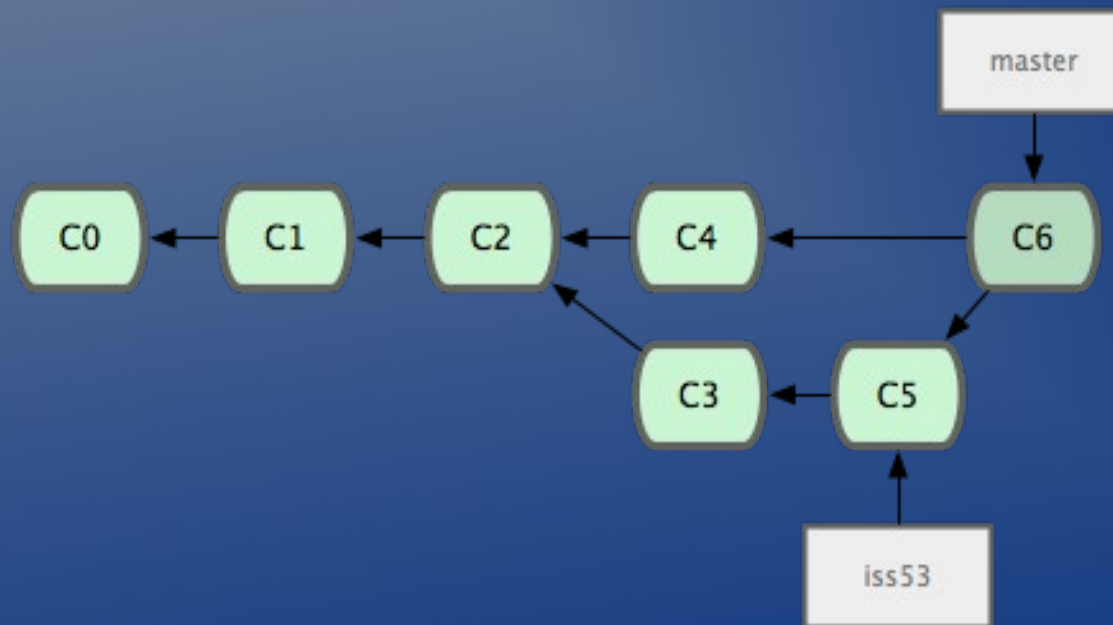
- 将 issue53 合并到 master 中

`$ git merge iss53`

Git 为分支合并自动识别出最佳的同源合并点。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



- 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新的快照，并自动创建一个指向它的 commit（C6）。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





# 冲突的合并

- 如果你修改了两个待合并分支里同一个文件的同一部分，Git 就无法干净地把两者合到一起

```
[inosin@it-net study_git]$ git merge iss53
Auto-merging confilct_test
CONFLICT (add/add): Merge conflict in confilct_test
Automatic merge failed; fix conflicts and then commit the result.
[inosin@it-net study_git]$ git status
# On branch master
# Changes to be committed:
#
#       new file:   test4
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both added:    confilct_test
#
```

需要手动去解决“confilct\_test”这个文件中的冲突  
然后运行

```
$ git add confilct_test
```

```
$ git commit -m 'resovle confilct'
```

# 常用分支管理

- 查看分支

```
$ git branch
```

- 查看分支和最后一次 commit 信息

```
$ git branch -v
```

- 查看已合并的分支

```
$ git branch --merged
```

- 查看未合并的分支

```
$ git branch --no-merged
```

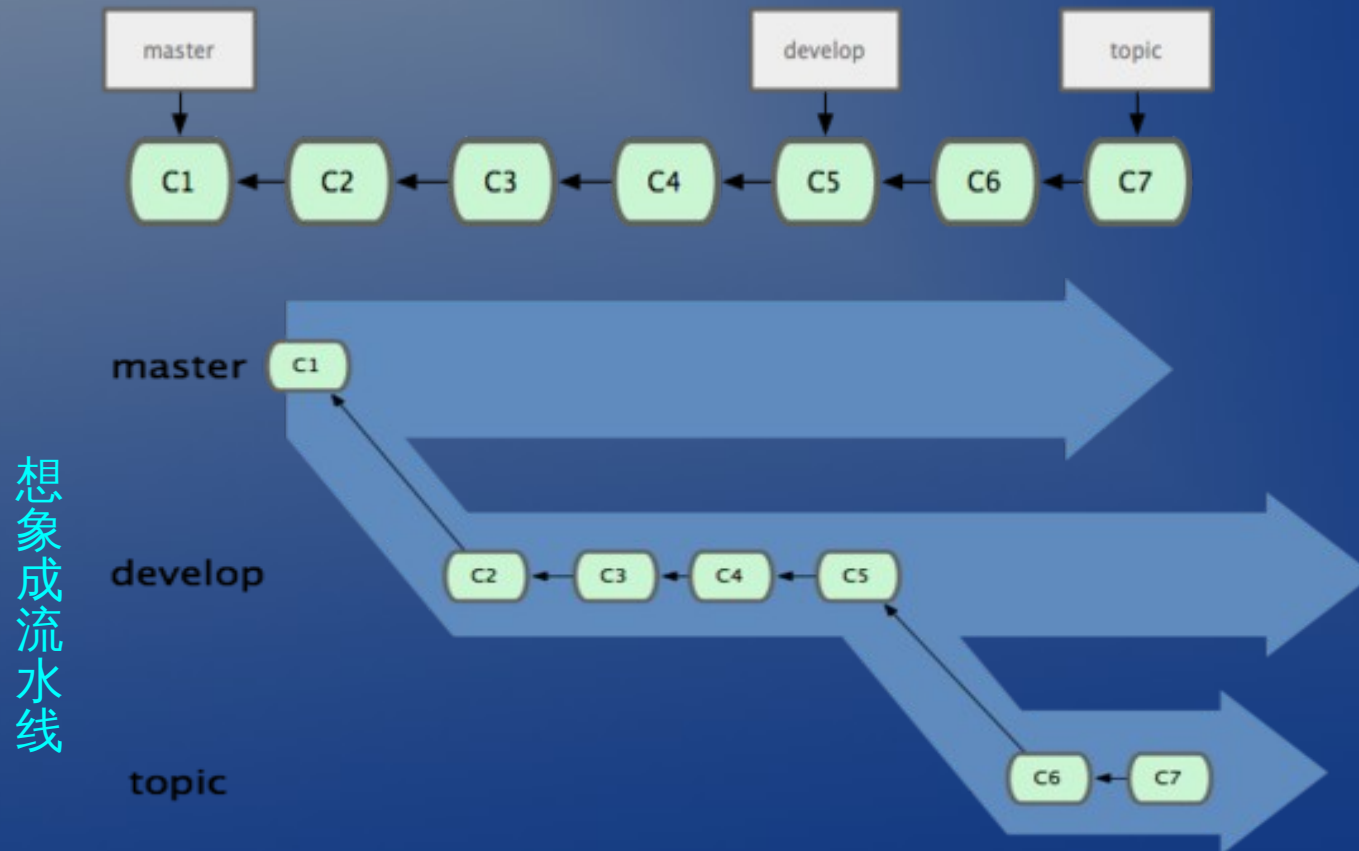
- 删除分支

```
$ git branch -d <branchname>
```

# 分支式工作流程

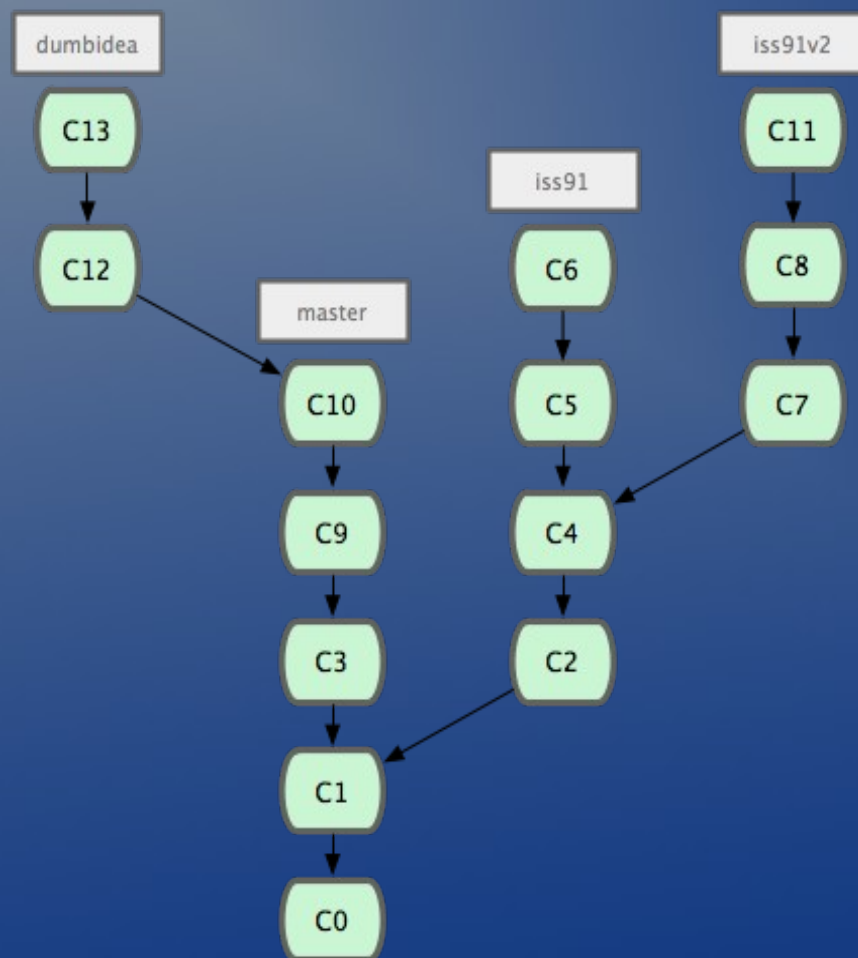
- 长期分支：

在 master 分支中保留完全稳定的代码，一个名为 develop 或 next 的平行分支，专门用于后续的开发，或仅用于稳定性测试 ...

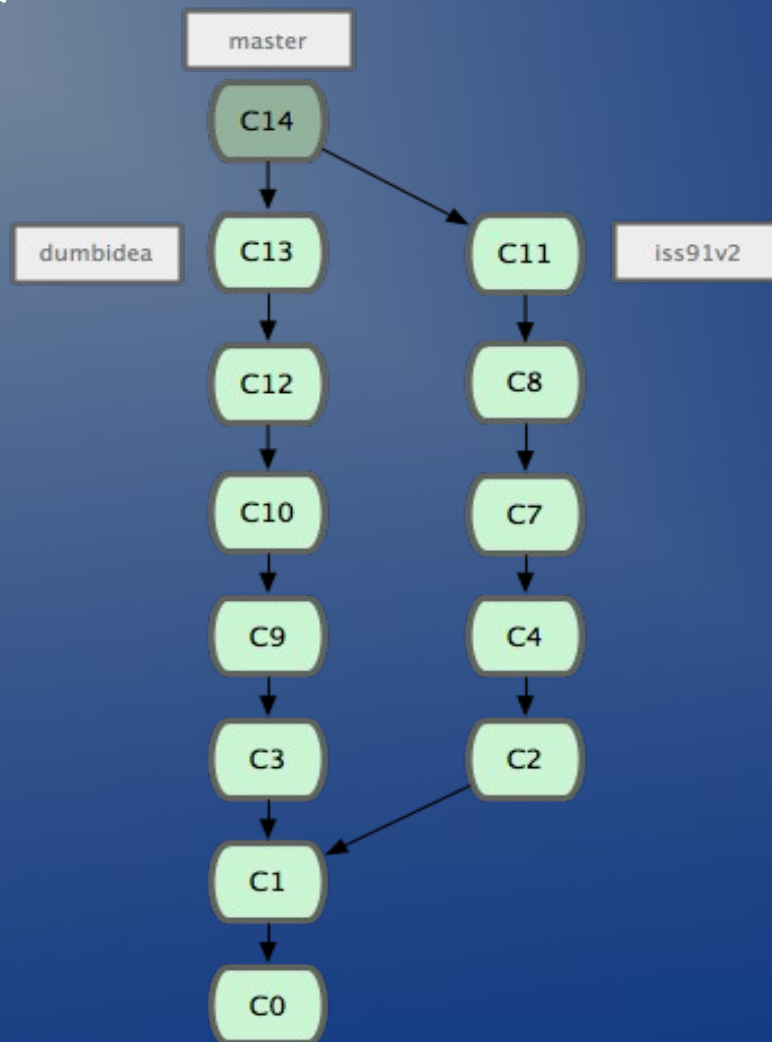


## 特性分支

起先我们在 master 工作到 C1，然后开始一个新分支 iss91 尝试修复 91 号缺陷，提交到 C6 的时候，又冒出一个新的解决问题的想法，于是从之前 C4 的地方又分出一个分支 iss91v2，干到 C8 的时候，又回到主干中提交了 C9 和 C10，再回到 iss91v2 继续工作，提交 C11，接着，又冒出个不太确定的想法，从 master 的最新提交 C10 处开了个新的分支 dumbidea 做些试验。



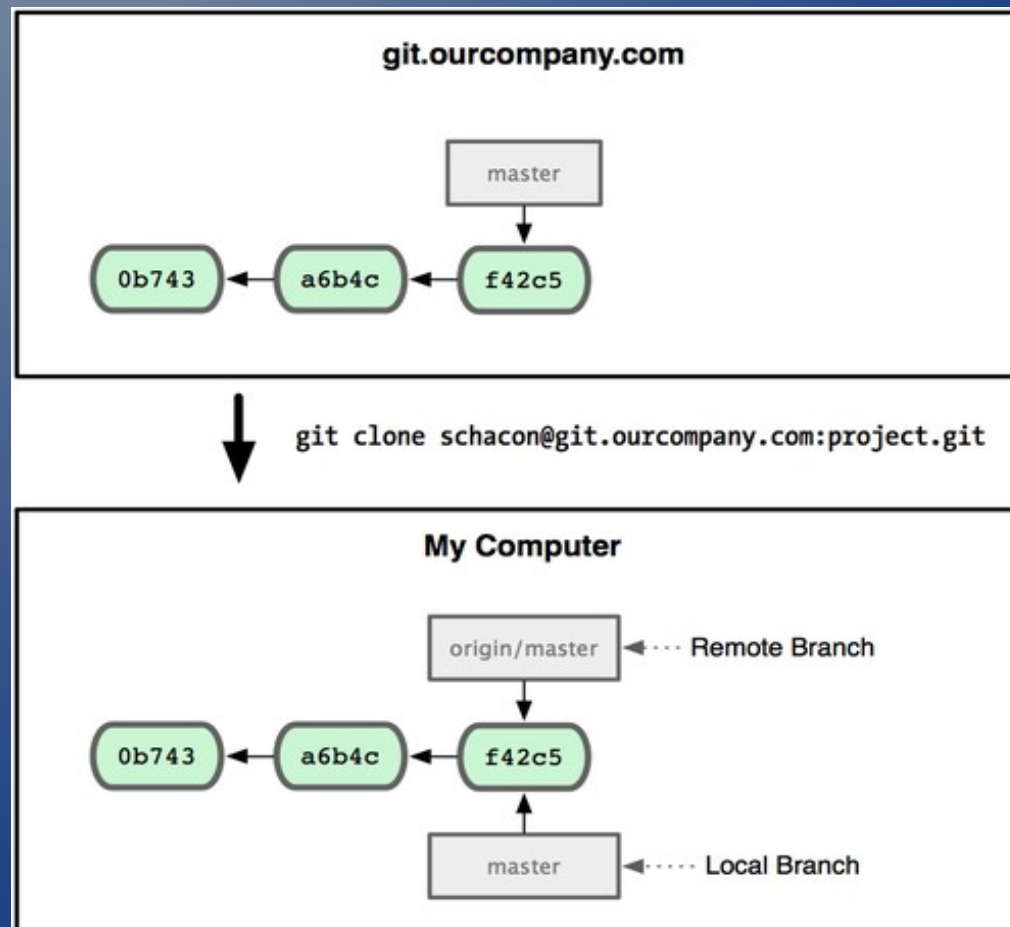
现在，假定两件事情：我们最终决定使用第二个解决方案，即 iss91v2 中的办法；另外，我们把 dumbidea 分支拿给同事们看了以后，发现它竟然是个天才之作。所以接下来，我们抛弃原来的 iss91 分支（即丢弃 C5 和 C6），直接在主干中并入另外两个分支。最终的提交历史将变成：



- 请务必牢记这些分支全部都是本地分支，这一点很重要。当你在使用分支及合并的时候，一切都是在你自己的 Git 仓库中进行的——完全不涉及与服务器的交互。

# 远程分支

- 用 clone 命令得到的就是一个指向远程 origin/master 的本地分支 master



- 将本地分支推送到远程

```
$ git push < 远程仓库名 > <[ 本地分支名 :] 远程分支名 >
```

```
$ git push origin inosinfix
```

```
$ git push origin inosinfix:teamfix (本地分支名 :inosinfix , 远程分支名 : teamfix )
```

- 查看远程分支名

```
$ git branch -r
```

- 查看远程分支名和最后一次提交信息

```
$ git branch -rv
```

可以看到远程分支多了一个

origin/master

origin/inosinfix



- 其他人想抓取这个分支与你共同工作

```
$ git checkout -b teamfix origin/inosinfix
```

- 当你做了些修改并想提交到远程时，远程又有了新的更新，这时 push 会失败，这是可以有两种方式解决：

```
$ git fetch 抓取新的内容
```

```
$ git merge origin/inosinfix 进行手工合并
```

- 或者

```
$ git pull origin inosinfix 抓取最新内容并自动合并
```

- 这时再 push 就没有问题了

# 值得注意的是

- 在 fetch 操作抓来新的远程分支之后，你仍然无法在本地编辑该远程仓库。换句话说，在远程仓库中，别人提交了一个新的分支，你用 fetch 命令不会有一个新的本地分支，有的只是一个你无法移动的远程分支的指针。如果想对这个分支做点什么，必须建立一个本地分支与之相对应
- 建一个与远程分支同名的本地分支：  
`$ git checkout --track origin/inosinfix`
- 建一个与远程分支不同名的本地分支：  
`$ git checkout -b teamfix origin/inosinfix`

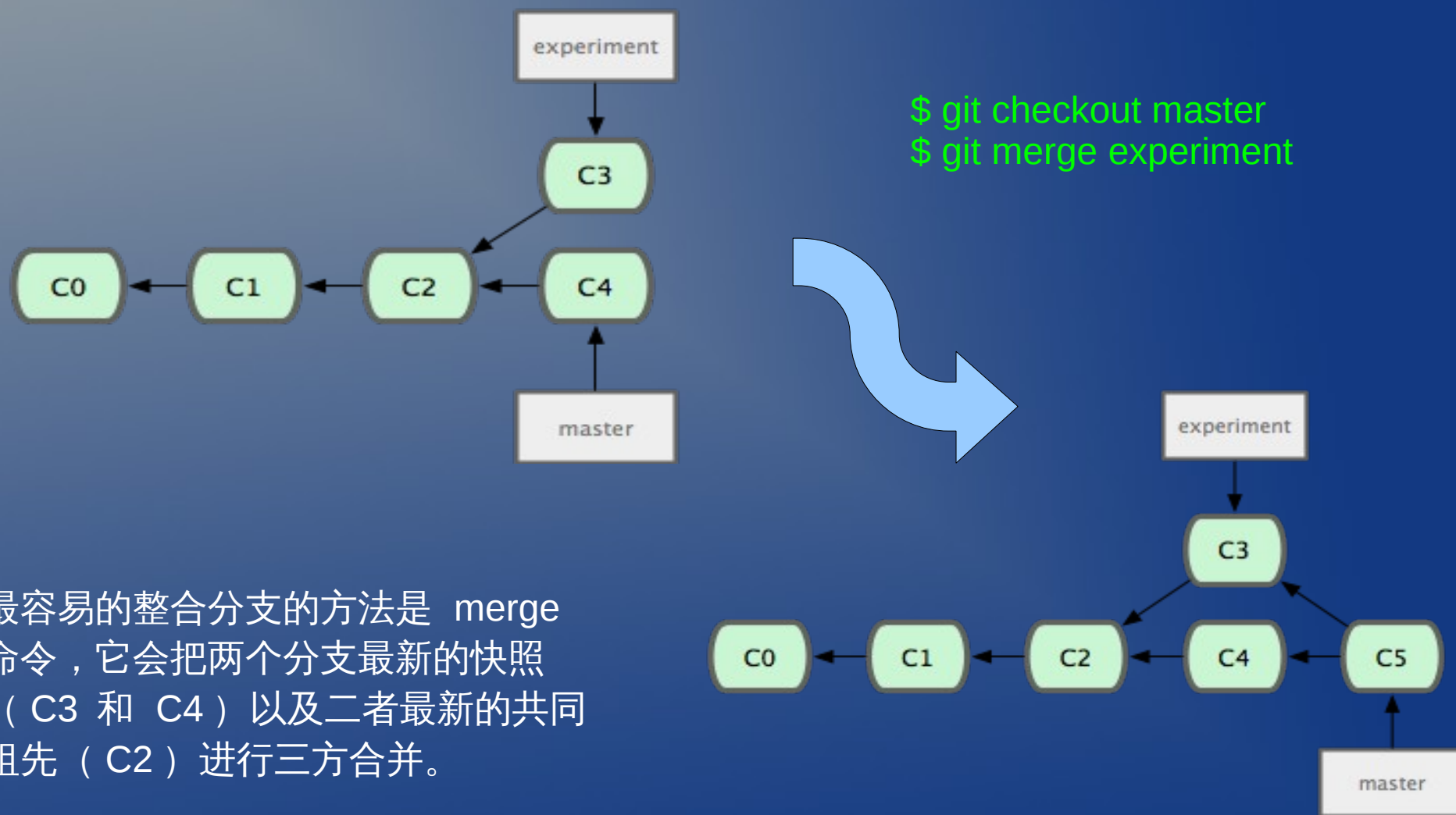
- 删除远程分支

```
$ git push < 远程名 > :< 分支名 >
```

```
$ git push origin :inofix
```

# 把一个分支整合到另一个分支的办法 1

## Merge



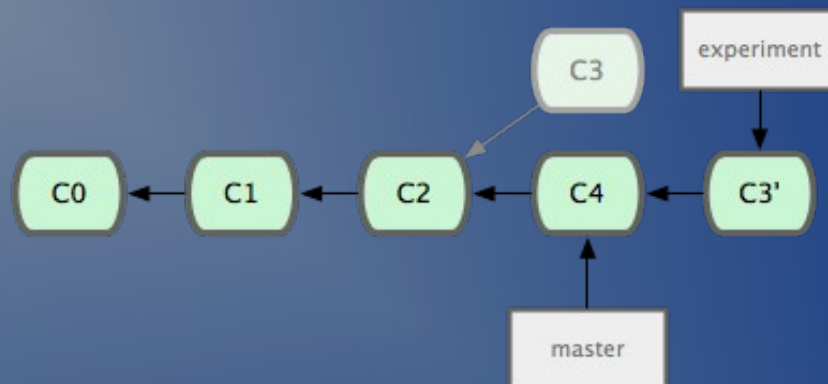
最容易的整合分支的方法是 merge 命令，它会把两个分支最新的快照（C3 和 C4）以及二者最新的共同祖先（C2）进行三方合并。

# 把一个分支整合到另一个分支的办法 2

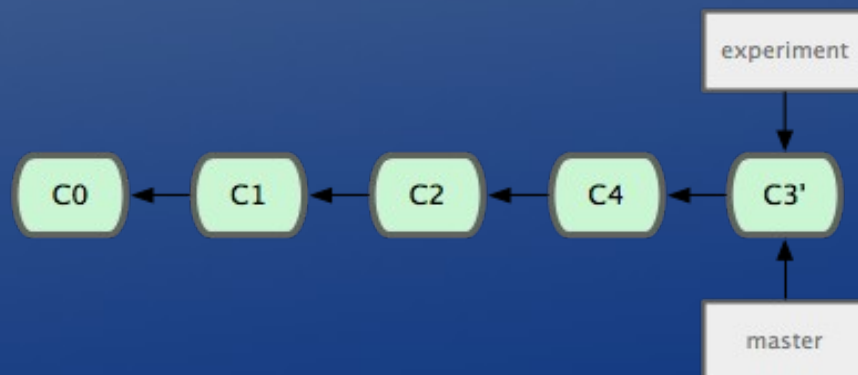
## Rebase

- 把在 C3 里产生的变化补丁重新在 C4 的基础上打一遍

`$ git checkout experiment`  
`$ git rebase master`



- 回到 master 分支然后进行一次快进合并



- Rebase 的原理

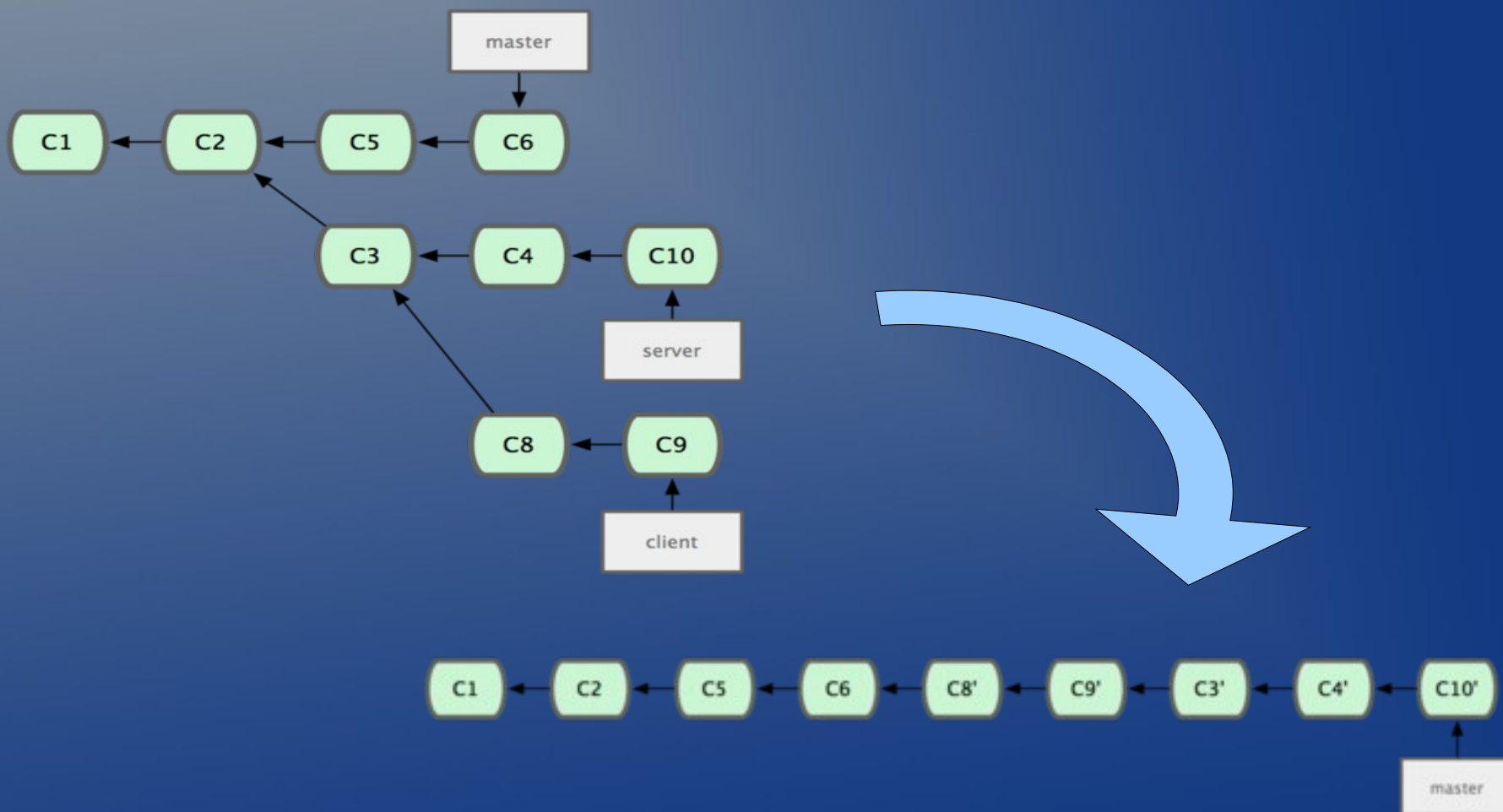
回到两个分支（你所在的分支和你想要衍合进去的分支）的共同祖先，提取你所在分支每次提交时产生的差异（diff），把这些差异分别保存到临时文件里，然后从当前分支转换到你想要衍合入的分支，依序施用每一个差异补丁文件

- merge 和 rebase 的区别

合并结果中最后一次提交所指向的快照，无论是通过一次衍合还是一次三方合并，都是同样的快照内容，只是提交的历史不同罢了。衍合按照每行改变发生的次序重演发生的改变，而合并是把最终结果合在一起。

# Rebase 带来得好处

- 通过 rebase 可以将复杂的提交历史更清晰



# Rebase 的风险

- 永远不要衍合那些已经推送到公共仓库的更新。  
在衍合的时候，实际上抛弃了一些现存的 commit 而创造了一些类似但不同的新 commit。如果你把 commit 推送到某处然后其他人下载并在其基础上工作，然后你用 git rebase 重写了这些 commit 再推送一次，你的合作者就不得不重新合并他们的工作，这样当你再次从他们那里获取内容的时候事情就会变得一团糟。
- 如果把衍合当成一种在推送之前清理提交历史的手段，而且仅仅衍合那些永远不会公开的 commit，



# 服务器上的 **Git**

# 协议

- 本地协议

`$ git clone /opt/git/project.git`

`$ git clone file:///opt/git/project.git`

- SSH 协议

`$ git clone ssh:user@server:project.git`

`$ git clone user@server:project.git`

- Git 协议

- HTTP/S 协议

# 服务器上的部署

- 建立一个 git 用户，shell 设定为 git-shell  
`$ useradd -s /usr/bin/git-shell git`
- 把开发者的 SSH 公钥添加到这个用户的 `authorized_keys` 文件中
- 建立空仓库  
`$ git --bare init`

# Git 管理的一些工具

- 网页界面 GitWeb
- 权限管理器 Gitis
- ...

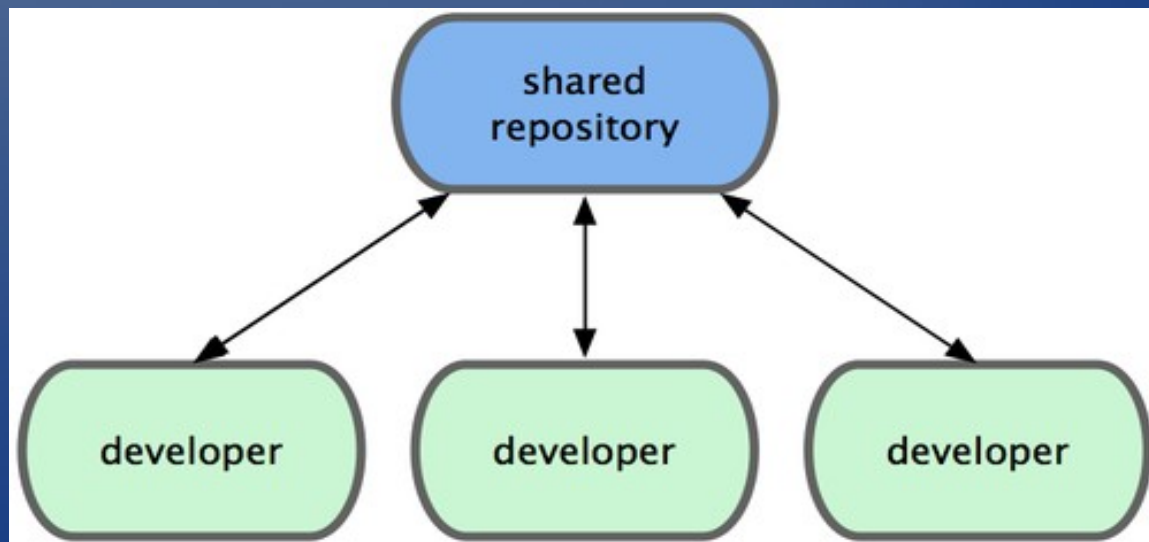
# 分布式 **Git**

# 分布式工作流程

- 集中式系统上，每个开发者就像是连接在集线器上的节点，彼此的工作方式大体相像。
- 在 Git 网络中，每个开发者同时扮演着节点和集线器的角色，这就是说，每一个开发者都可以将自己的代码贡献到另外一个开发者的仓库中，或者建立自己的公共仓库，让其他开发者基于自己的工作开始，为自己的仓库贡献代码

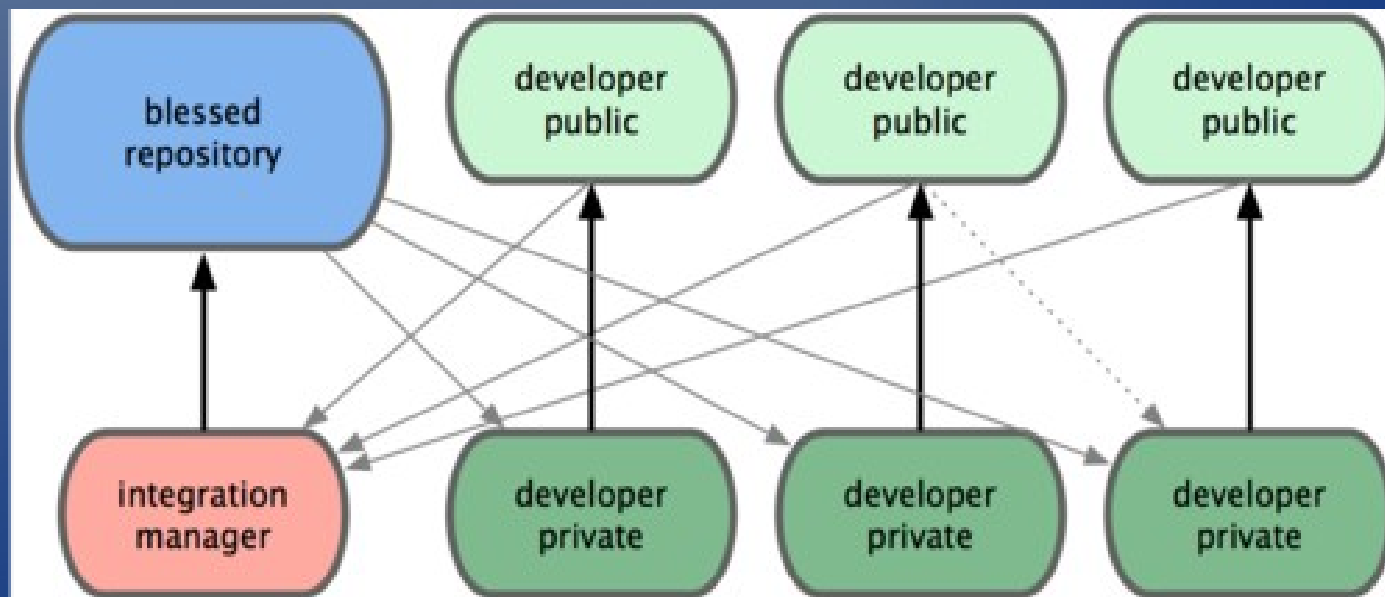
# 传统的集中式工作流

- 只需要配置好一台中心服务器
- 每个开发者都有推送数据的权限
- 如果提交代码时有冲突，Git 根本就不会让用户覆盖他人代码，它直接驳回第二个人的提交操作。



# 集成管理员 workflow

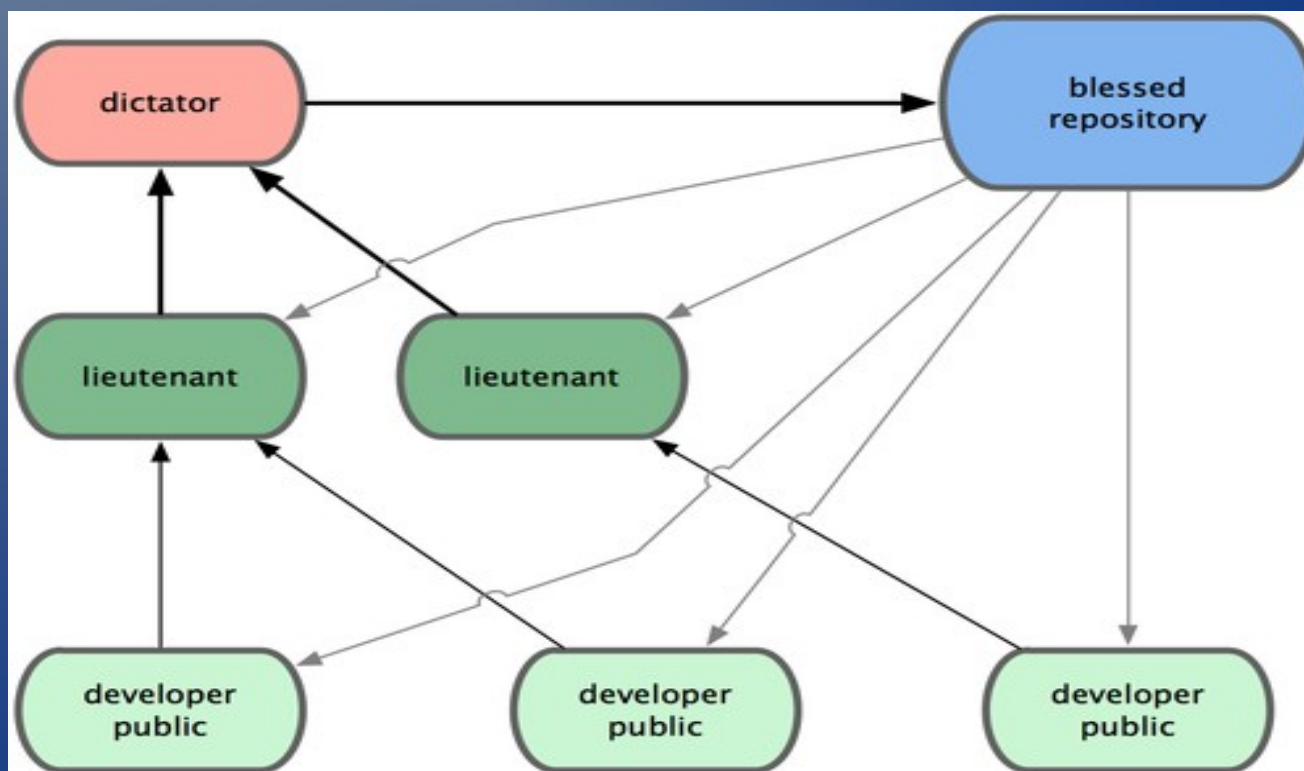
- 由于 Git 允许使用多个远程仓库，开发者便可以建立自己的公共仓库，往里面写数据并共享给他人，而同时又可以从别人的仓库中提取他们的更新过来。





# 司令官与副官 workflow

- 这种 workflow 并不常用，只有当项目极为庞杂，或者需要多级别管理时，才会体现出优势。利用这种方式，项目总负责人（即司令官）可以把大量分散的集成工作委托给不同的小组负责人分别处理，最后再统筹起来，如此各人的职责清晰明确，也不易出错（此乃分而治之）。



**Let's Git**