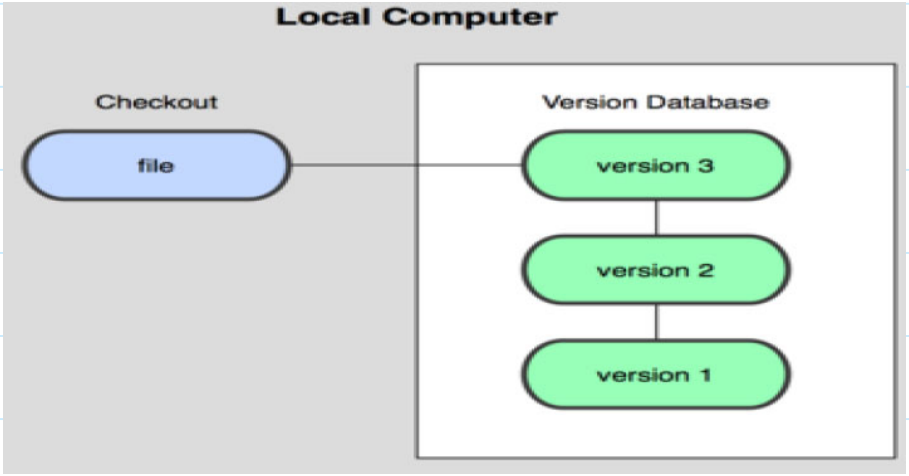
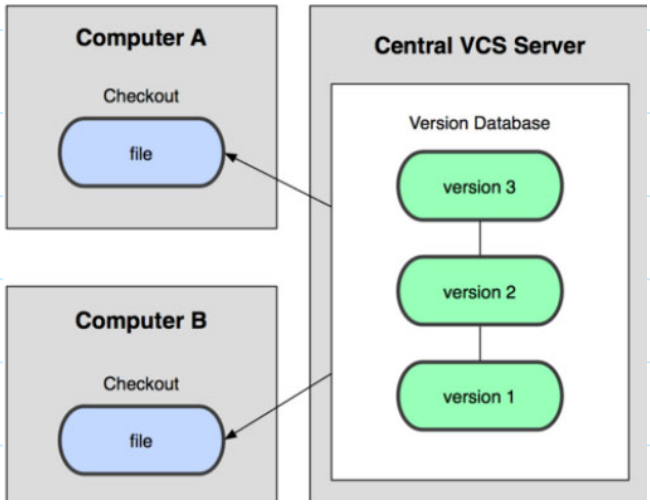
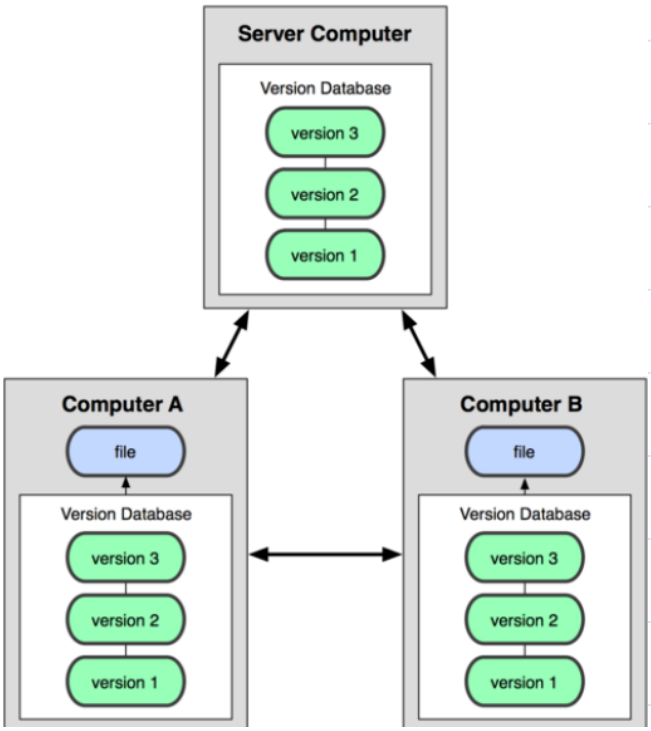
2-4-2 软件演化与配置管理

2019年4月13日　　23:09

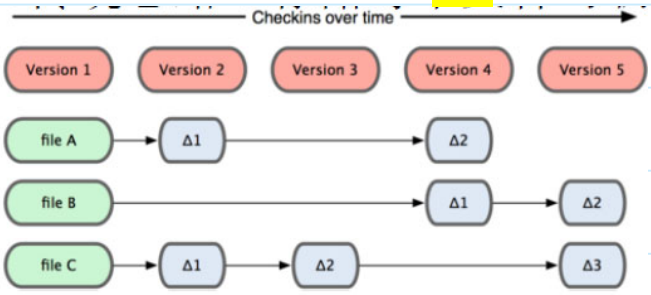
1 版本控制系统

- 本地版本控制系统(LocalVCS)
  - 采用简单的数据库或文件系统来记录本地文件的历次更新差异
  - 

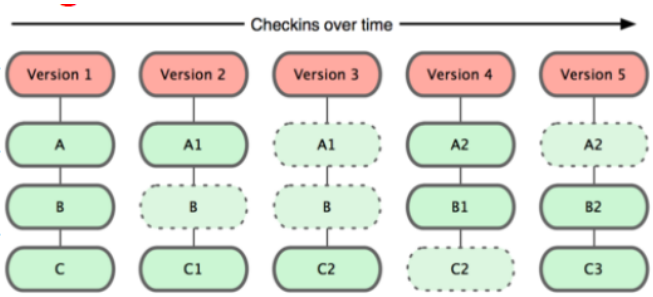
- 集中化的版本控制系统
  - CVS，Subversion(SVN)以及Perforce等，有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的开发者通过客户端连到这台服务器，取出最新的文件或者提交更新
  - 

- 分布式版本控制系统
  - 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。
  - 任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。
  - 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的基本思想

- 其他大多数SCM系统主要关心文件内容的具体差异(每次记录有哪些文件作了更新，以及都更新了哪些行的什么内容)；
- Git关心文件数据的整体是否发生变化，并不保存这些前后变化的差异数据
  - 把变化的文件作快照后，记录在一个微型的文件系统中。
  - 每次提交更新时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照的索引。
  - 为提高性能，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链接。
- 传统思路：存储每个文件与初始版本的差异
  - 

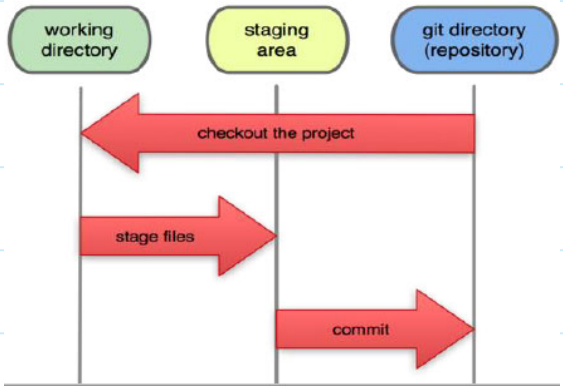
- git的思路：存储项目随时间改变的快照



- 在本地磁盘上保留项目的完整历史,以空间换时间
- 优点：高效

Git中文件的三种状态

- 对于任何一个文件，在Git内都只有三种状态：
  - 已提交(committed)：表示该文件已经被安全地保存在本地数据库中
  - 已修改(modified)：表示修改了某个文件，但还没有提交保存；
  - 已暂存(staged)：表示把已修改的文件放在下次提交时要保存的清单中
- Git管理项目的三个工作区域：
  - Git目录(仓库)
  - 工作目录
  - 暂存区域



Git的工作区

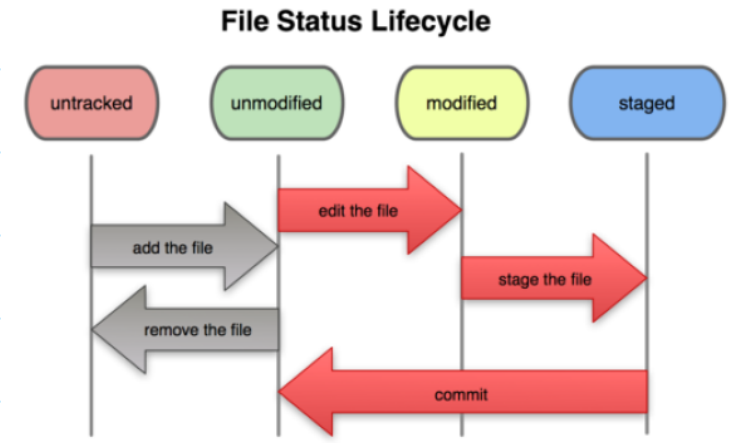
- 每个项目都有一个Git目录(Gitdirectory)，用来保存元数据和对象数据库。每次clone镜像仓库的时候，实际拷贝的就是这个目录里面的数据。
- 从Git目录中取出某个版本的所有文件和目录，用以开始后续工作，形成工作目录(workingdirectory)，接下来就可以在工作目录中对这些文件进行编辑。
- 所谓的暂存区域(stagingarea)只不过是个简单的索引文件，是一个包含文件索引的目录树，像是一个虚拟的工作区。在这个虚拟工作区的目录树中，记录了文件名、文件的状态信息（时间戳、文件长度等），文件的内容并不存储其中，而是保存在Git对象库中。暂存区域的设计可以实现更有效、灵活的控制要提交的文件对象。

Git的基本工作流程

1. 在工作目录中修改某些文件。
2. 对修改后的文件进行快照，然后保存到暂存区域。
3. 提交更新，将保存在暂存区域的文件快照永久转储到Git目录中。

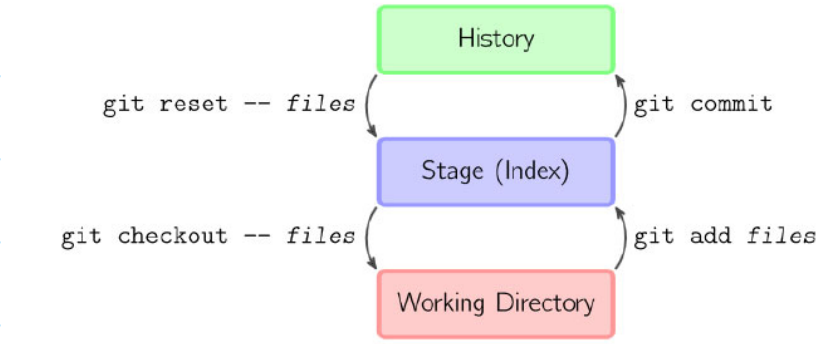
- 可以从文件所处的位置来判断状态：
  - 如果是Git目录中保存着的特定版本文件，就属于已提交状态；
  - 如果作了修改并已放入暂存区域，就属于已暂存状态；
  - 如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。

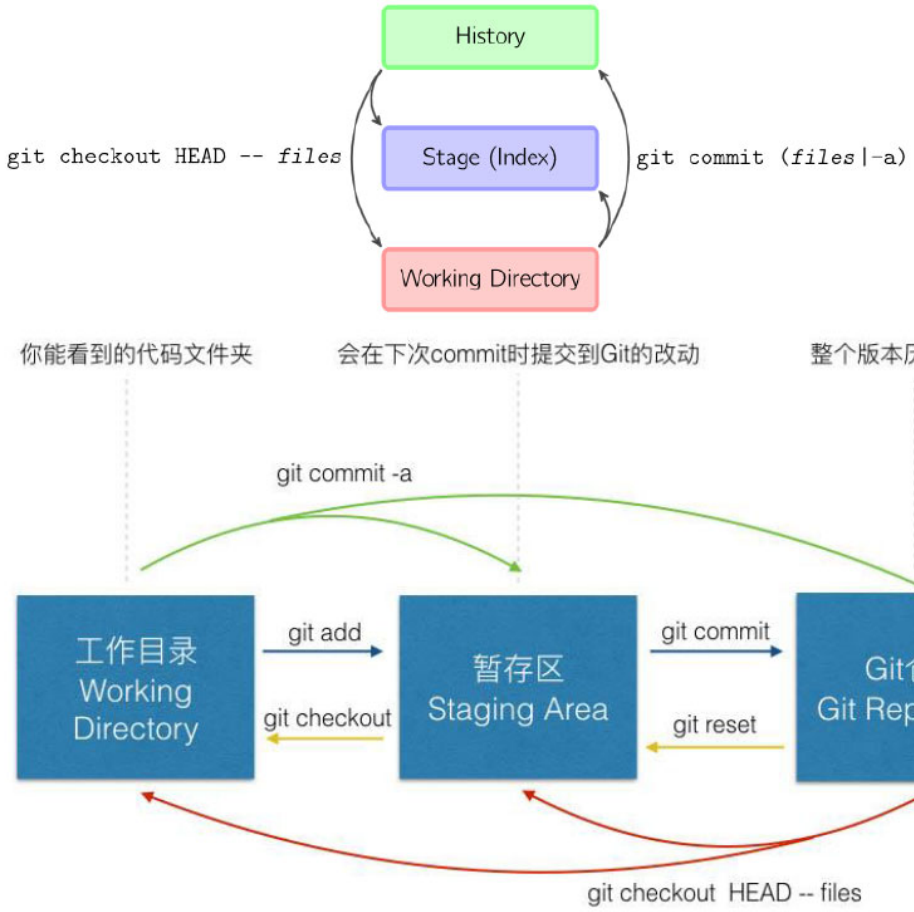
2 基本的Git命令

- 取得项目的Git仓库
  - 下载并在本机安装设置Git环境http://msysgit.github.io/
  - 在工作目录中初始化新仓库
  - 要对现有的某个项目开始用Git管理，只需到此项目所在的目录，执行git init命令，用git add命令告诉Git开始对这些文件进行跟踪，然后提交：
    - git add *.c
    - git add readme.txt
    - git commit -m'initial project version'
  - 从现有仓库克隆：复制服务器上项目的所有历史信息到本地
    - git clone [url]
- 记录每次更新到仓库
  - 在工作目录对某些文件作了修改之后，Git将这些文件标为“已修改”，可提交本次更新到仓库。
  - 逐步把这些修改过的文件放到暂存区域，直到最后一次性提交所有这些暂存起来的文件，如此重复
  - 

- 跟踪新文件、暂存已修改文件
  - 使用git add开始跟踪一个新文件（使某个文件纳入到git中管理）。
  - 一个修改过的且被跟踪的文件，处于暂存状态。
  - gita dd后面可以指明要跟踪的文件或目录路径。如果是目录的话，就说明要递归跟踪该目录下

的所有文件。
    - git add的潜台词：把目标文件快照放入暂存区域，也就是add file into staged area，同时之前未曾跟踪过的文件标记为需要跟踪。
    - 若对已跟踪的文件进行了修改，使用git add命令将其放入暂存区；
    - 运行了git add之后又对相应文件做了修改，要重新git add。
- 检查当前文件状态.要确定哪些文件当前处于什么状态，用git status命令：
    - #On branch master nothing to commit(working directory clean)当前没有任何跟踪着的文件，也没有任何文件在上次提交后更改过
    - #On branch master#Untracked files:...有未跟踪的文件，使用git add开始跟踪一个新文件
    - #On branch master#Changes to be committed:有处于已暂存状态的文件
- 查看已暂存和未暂存的更新
    - git status回答：当前做的哪些更新还没有暂存？有哪些更新已经暂存起来准备好了下次提交？
    - 如果要查看具体修改了什么地方，可以用git diff命令，使用文件补丁的格式显示具体添加和删除的行。
    - 要查看尚未暂存的文件更新了哪些部分，不加参数直接输入git diff：
        - 比较的是工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。
    - 若要查看已暂存起来的文件和上次提交时的快照之间的差异，可以用git diff --cached命令。
- 提交更新
    - 在使用git commit命令进行提交之前，要确认是否还有修改过的或新建的文件没有git add过，否则提交的时候不会记录这些还没暂存起来的变化。
        - 每次准备提交前，先用git status进行检查，然后再运行提交命令git commit。
    - 提交后返回结果：
        - 当前是在哪个分支(master)提交的
        - 本次提交的完整SHA1校验和是什么
        - 在本次提交中，有多少文件修订过、多少行添改和删改过。
    - 提交时记录的是放在暂存区域的快照，任何还未暂存的仍然保持已修改状态，可以在下次提交时纳入版本管理。每一次运行提交操作，都是对项目做一次快照，以后可以回到这个状态，或者进行比较。
- 跳过使用暂存区域、移除文件
    - Git提供了一个跳过使用暂存区域的方式，只要在提交的时候，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gitadd步骤；
    - 使用git rm命令从Git中移除某个文件，把它从已跟踪文件清单(暂存区域)中移除，并连带从工作目录中删除指定的文件。
- 小结

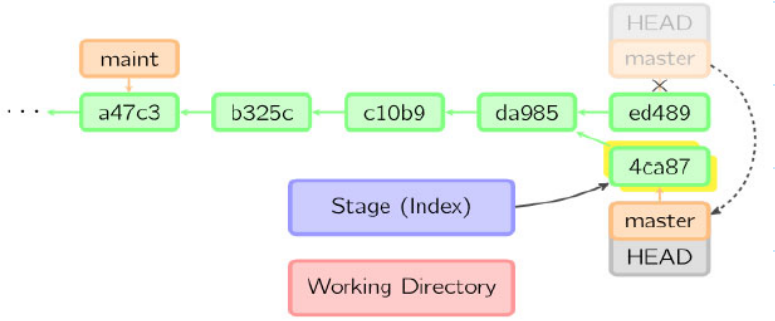
- 

- 

- 查看提交历史

  git log：一个强大的git查询工具，以各种方式查看git项目的提交历史，并进行各种统计分析。
  - 不加任何参数：会按提交时间列出所有的commit，最近的排在最上面，包含SHA1校验和、作者的名字和电子邮件地址、提交时间以及提交说明；
  - -p参数：显示每次提交的内容差异(针对每个变化了的文件，增加了哪些行、删除了哪些行)
  - --stat参数：查看每次commit的简要统计信息，列出所有被修改过的文件及总数量、每个被修改过的文件有多少行发生变化等；
  - --pretty=format:...按特定格式展示commit的结果(格式化输出)；
  - --graph：以图形方式展示commit的历史；
  - 其他各种参数
  - 非常强大的工具，类似于数据库中的SQL select语句
- 更改一次提交
  - 如果在一次commit之后发现遗漏了某些文件修改或者提交信息不正确，使用git commit --

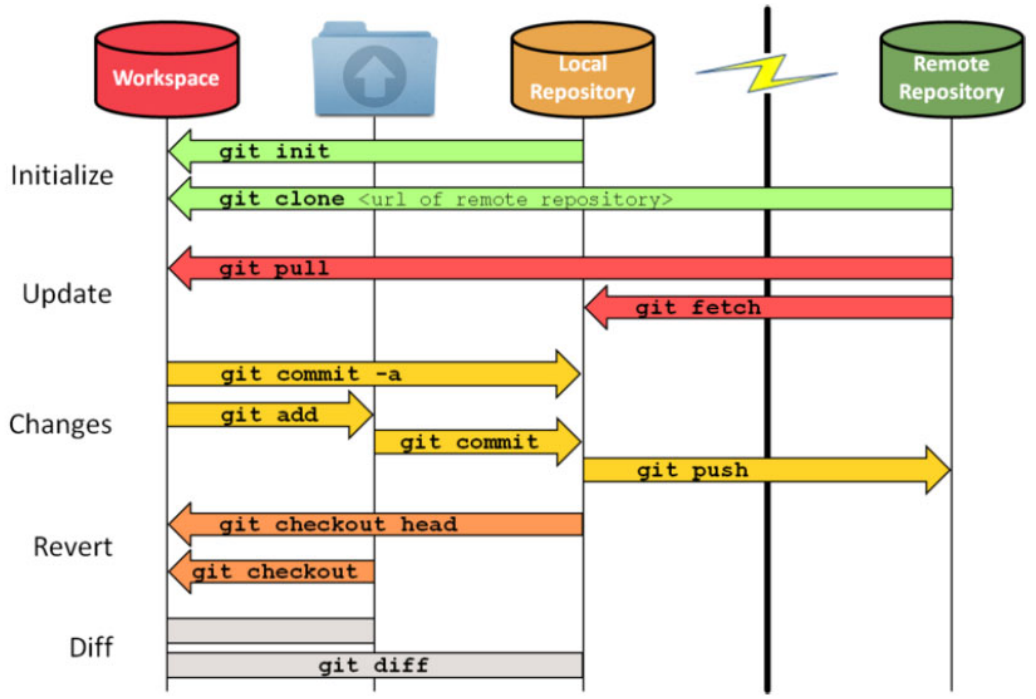
amend撤销上一次提交，形成新的提交；

- 本质：合并暂存区的修改和最近的一次commit的修改内容，用生成的新的commit替换掉原来的commit，而不会形成新的分支。
- 

- 更改一次提交另一种方法：
  i. git reset HEAD^：工作区不变，暂存区回退到上一次commit之前，上一次commit取消；
  ii. #重新修改
  iii. git add
  iv. git commit

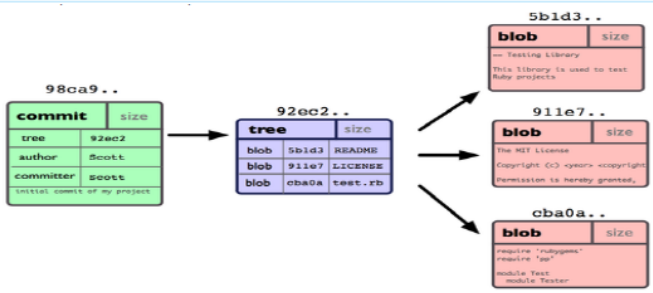
3 Git远程仓库指令

- 什么是“远程仓库”？
  - 远程仓库：托管在网络上的Git项目仓库；
  - 多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。
  - 管理远程仓库：添加远程库、移除废弃的远程库、管理不同的远程库分支、定义是否跟踪这些分支，等。
  - 从现有仓库克隆：复制服务器特定URL所指向的项目的所有历史信息到本地
    - git clone[url]
- 添加远程仓库：
  - git clone指令在克隆远程仓库内容到本地之后，自动生成了一个远程仓库配置(origin)。
  - git remote add<shortname><url>:添加一个新的远程Git仓库，同时指定一个缩写。
- 查看已配置的远程仓库：
  - git remote：获取当前配置的所有远程仓库；
  - git remote -v:显示当前配置的远程仓库及其读写操作权限(fetch,push)和URL地址
  - git remote show [remote-name]：查看某个远程仓库的详细信息；
- 移除/推送/拉取远程仓库
  - git remote rm pb：从本地移除远程仓库(不再关注其更新、不再对其有贡献)
  - git remote rename pb pb1：将远程仓库重命名；
  - git fetch：从远程仓库抓取数据到本地，获取本地仓库尚未拥有的全部更新

        - 如果本地仓库有了不同的修改，则需要手工将本地修改与远程仓库的修改合并起来(分支的合并gitmerge)。
    - git pull==git fetch+git merge：获取远程仓库的更新并与本地当前分支合并【慎用】；
    - git push [remote name][branch name]：将本地仓库中的数据推送到远程仓库
- 小结
    - 

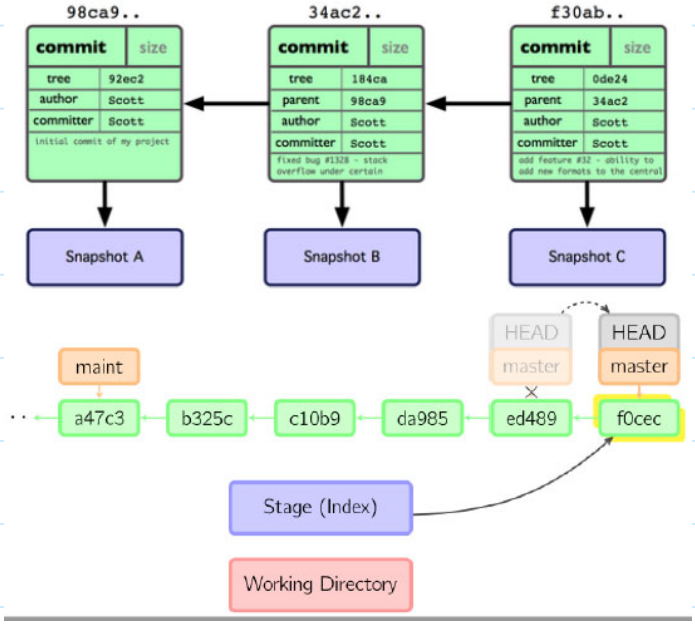
4 Git分支指令

单个提交(Commit)对象在仓库中的数据结构

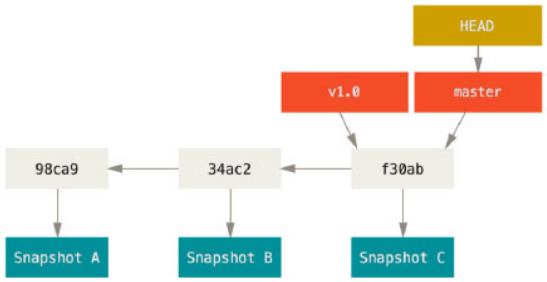
- 回顾：git每次提交不是保存文件的变化，而是保存文件的当前快照；
- 由三部分构成：
    - 多个表示待提交的文件快照内容的blob对象；
    - 一个记录着目录树内容及其中各个文件对应blob对象索引的tree对象；
    - 一个包含指向tree对象(根目录)的索引和其他提交信息元数据的commit对象。
    - 

多次提交之后

- 某些文件修改后再次提交，这次的commit对象会包含一个指向上次提交对象的指针。

- 

Git分支

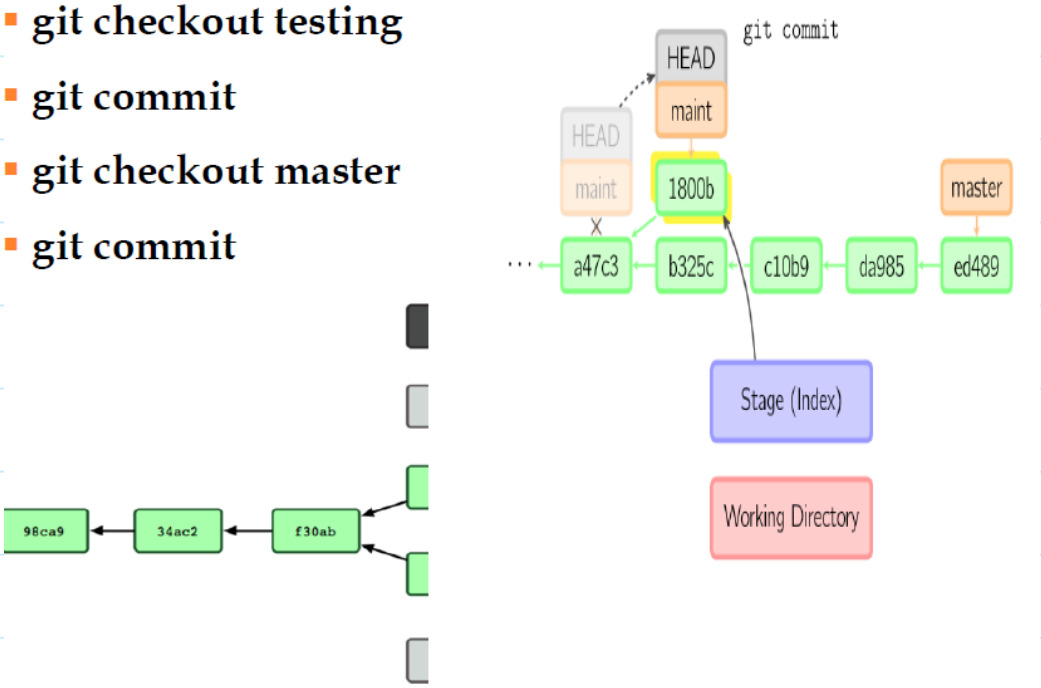
- 在git中提交时，会保存一个提交(commit)对象，该对象包含一个指向暂存内容快照的指针，包含本次提交的作者等相关附属信息，包含零个或多个指向该提交对象的父对象指针：
  - 首次提交：没有直接祖先；
  - 普通提交：有一个祖先；
  - 由两个或多个分支合并产生的提交：有多个祖先。
- Git中的分支本质上仅仅是个指向commit对象的可变指针。
  - 若干次提交后，master分支指向最后一次提交对象，它在每次提交的时候都会自动向前移动。
- Git的默认分支名：master
- 

创建/切换/删除分支

- git branch列出当前所有分支；
- git branch (name)在当前commit对象上新建一个分支指针；
- git使用一个叫做HEAD的特别指针来获知你当前在哪个分支上工作。
- 要切换到其他分支，可以执行git checkout命令。
- 创建一个新的分支并立即切换过去：git branch -b
- 删除一个分支：git branch -d
- 使用git log -decorate查看当前各个分支所指的commit对象

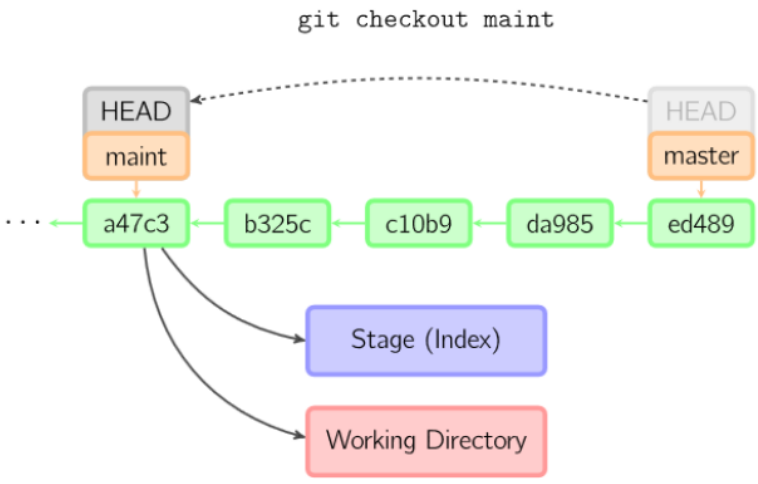
在新的分支上提交

- git checkout testing
- git commit
- git checkout master
- git commit



切换分支的本质

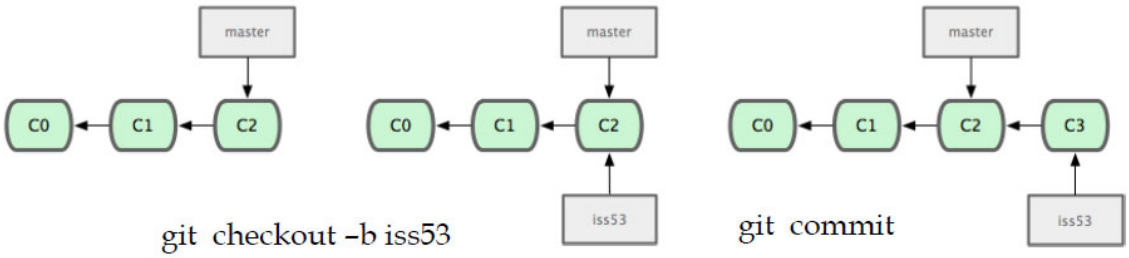
- HEAD标识会移动到那个分支，暂存区域和工作目录中的内容会和HEAD对应的提交节点一致。
- 新提交节点中的所有文件都会被复制到暂存区域和工作目录中；只存在于老的提交节点中的文件会被删除。

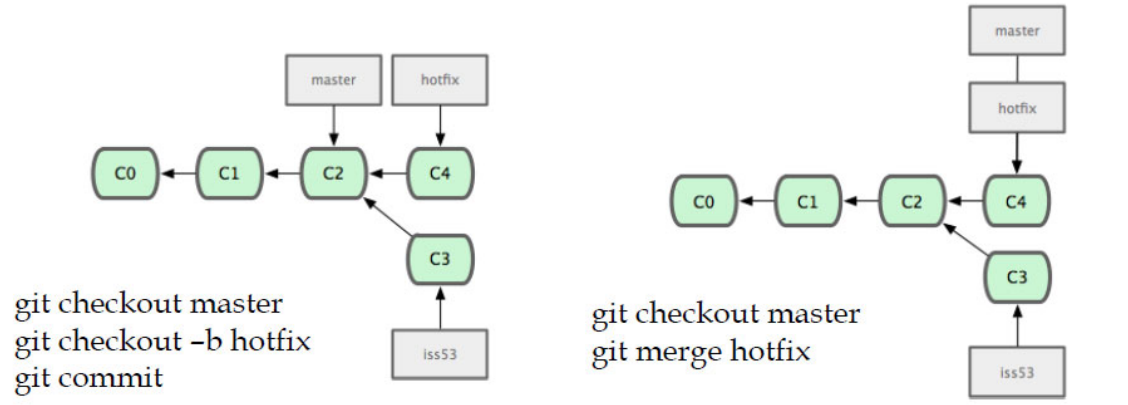


Git的分支机制的优越性

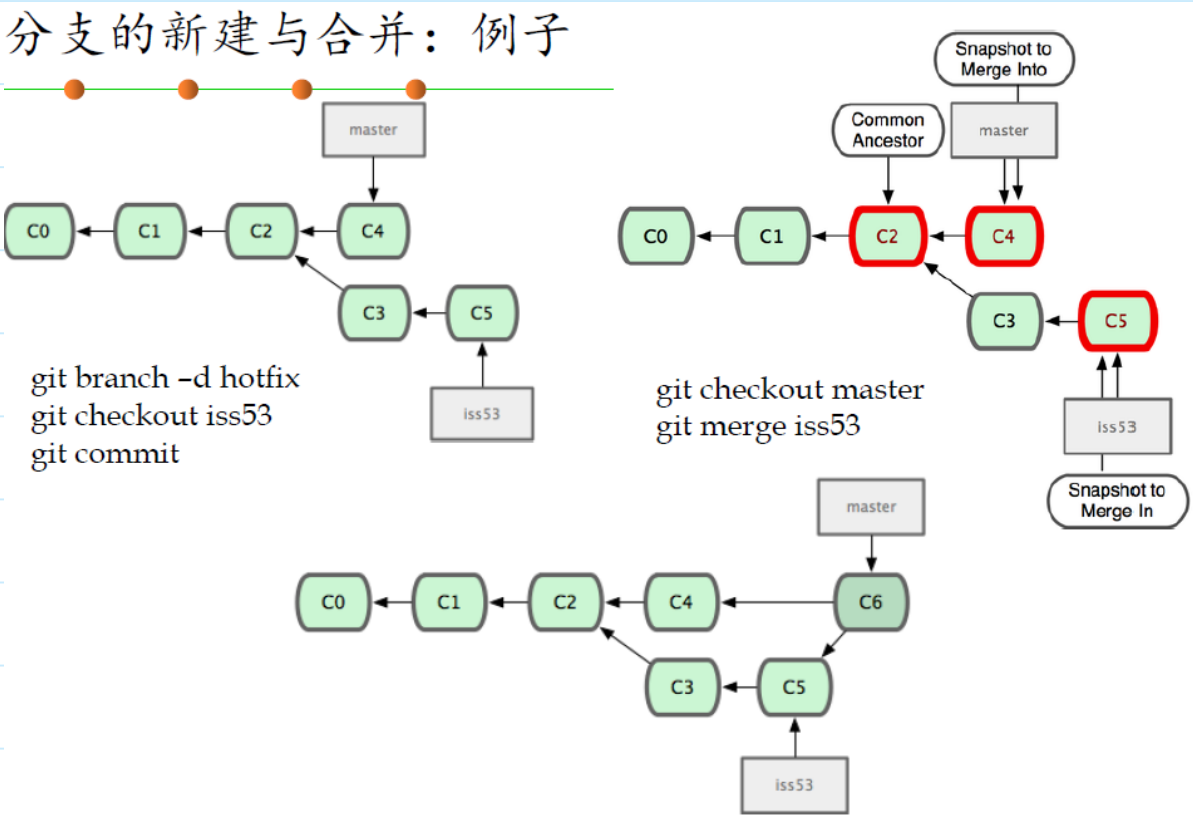
- Git中的分支实际上仅是一个包含所指对象校验和(40个字符长度SHA-1字串)的文件，非常方便；
- 传统的版本控制系统则采用将文件备份到目录的方式。
- Git的分支实现与项目复杂度无关，可以在几毫秒的时间内完成分支的创建和切换。
- 因为每次提交时都记录了祖先信息(即parent对象)，将来要合并分支时，寻找恰当的合并基础(即共同祖先)，实现起来非常容易。
- git merge (name)：将该分支合并到当前分支

分支的新建与合并：例子



- 

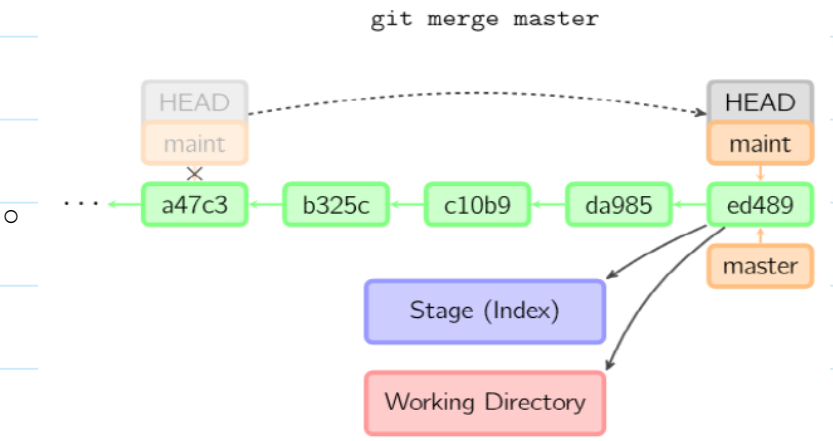
分支的新建与合并：例子

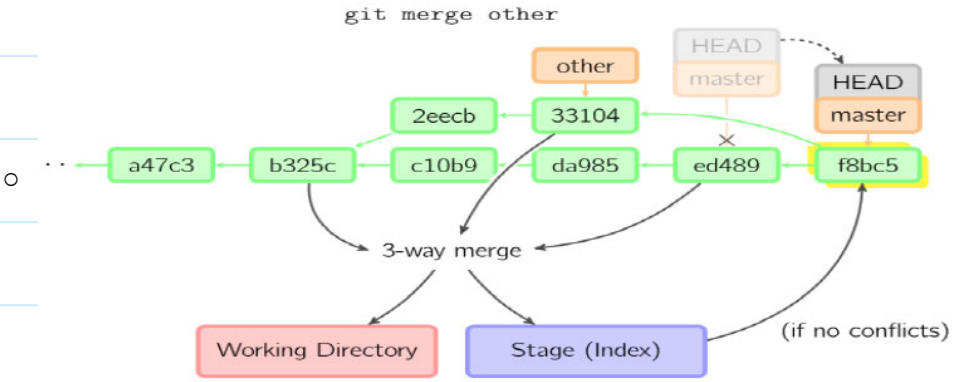
- 

分支的新建与合并：有冲突时如何?

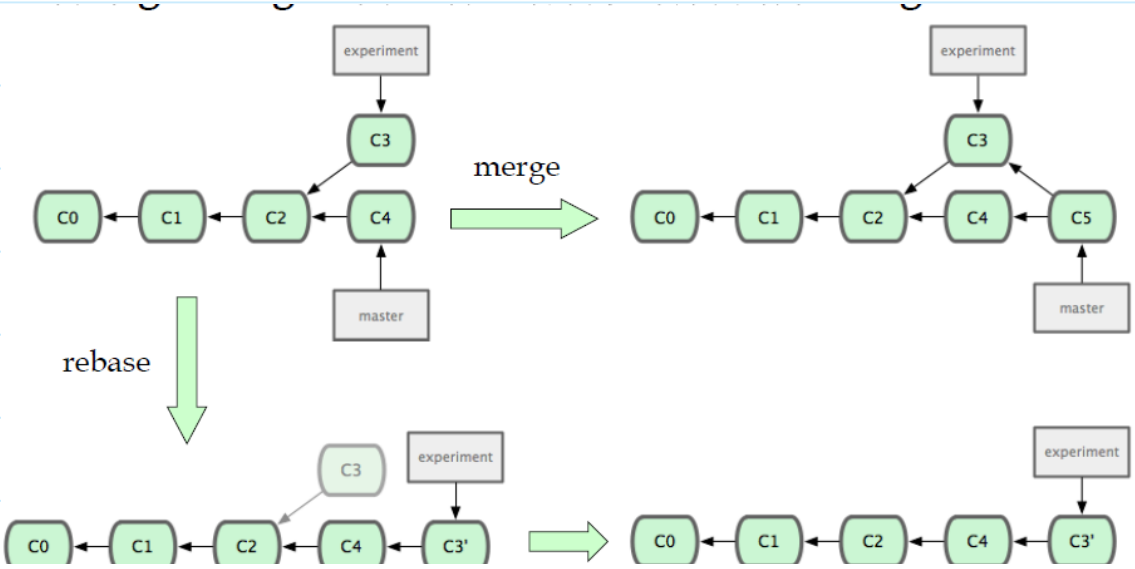
- 如果在不同的分支中都修改了同一个文件的同一部分，Git无法干净地把两者合到一起，需要依赖于人的裁决。
- 使用git status查看任何包含未解决冲突的文件，在有冲突的文件里加入标准的冲突解决标记，可以通过它们来手工定位并解决这些冲突。

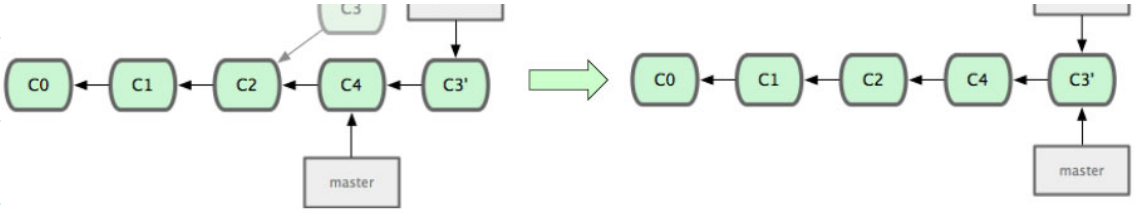
分支合并的本质（1）

- 如果被合并的分支是当前commit对象的祖父节点，那么合并命令将什么也不做。
- 如果当前commit是被合并分支的祖父节点，就导致fastforward合并。指向只是简单的移动，并生成一个新的提交。
  - 

- 否则，默认把当前commit对象和被合并的分支，以及他们的共同祖父commit节点进行一次三方合并。
- 结果是先保存当前目录和索引，然后和父节点33104一起做一次新提交
  - 

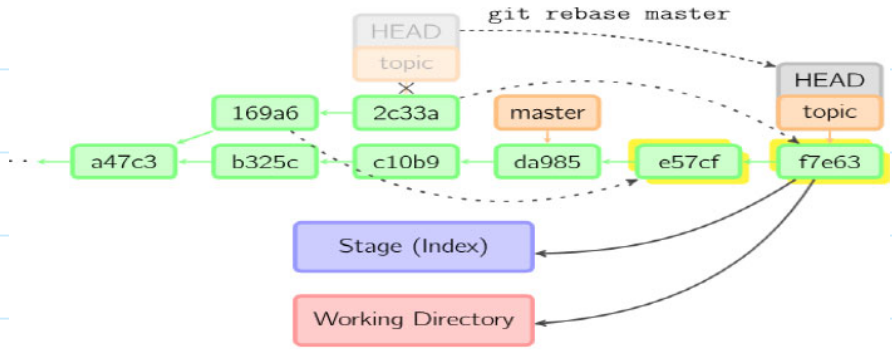
分支的衍合/变基(rebase)

- 除了git merge，另一种进行分支合并的方法：git rebase。
- 



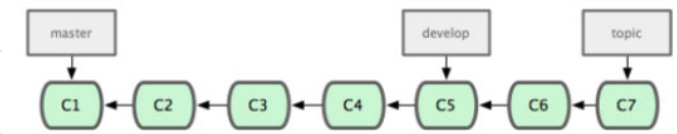
Merge和Rebase的区别：

- merge把两个父分支合并进行一次提交，提交历史不是线性的。
- rebase在当前分支上重演另一个分支的历史，提交历史是线性的——保持提交历史的整洁

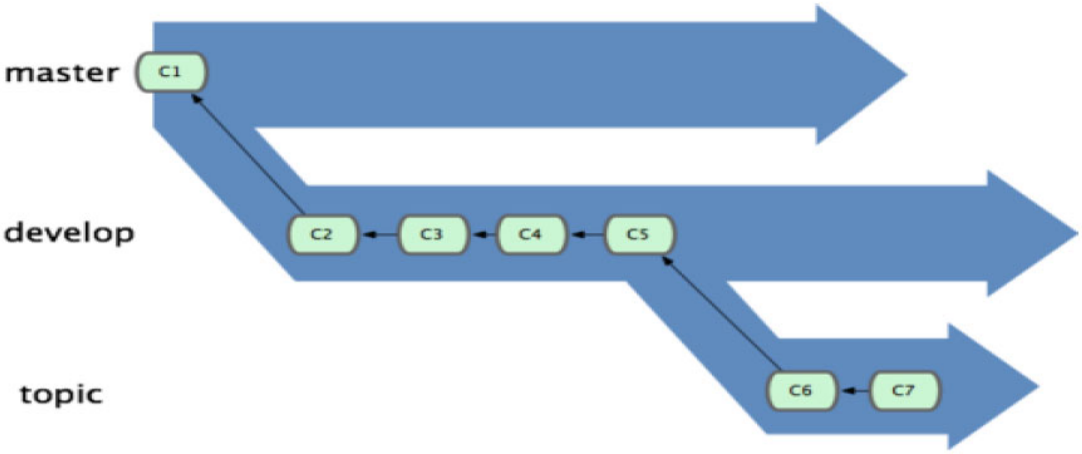


5 分支的价值

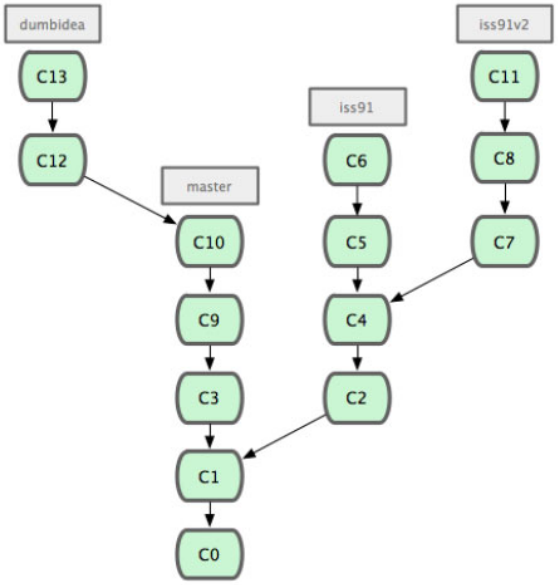
利用分支进行开发的工作流程

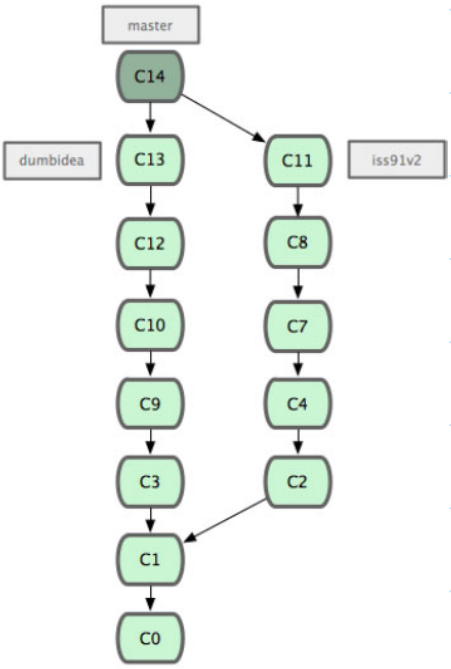
- 长期分支：可以同时拥有多个开放的分支，每个分支用于完成特定的任务，随着开发的推进，可以随时把某个特性分支并到其他分支中。
  - 仅在master分支中保留完全稳定的代码，即已经发布或即将发布的代码。
  - 同时还有一个名为develop或next的平行分支，专门用于后续的开发，或仅用于稳定性测试，一旦进入某种稳定状态，便可以把它合并到master里。
  - 这样，在确保这些已完成的特性分支能够通过所有测试并且不会引入更多错误之后，就可以并到主干分支中，等待下一次的发布。
  - 随着提交对象不断右移的指针。稳定分支的指针总是在提交历史中落后一大截，而前沿分支总是比较靠前。
    - 

  - 这么做的目的是拥有不同层次的稳定性：当这些分支进入到更稳定的水平时，再把它们合并到更高层分支中去。



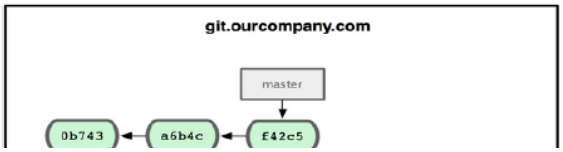
- 特性分支(Topic):是指一个短期的，用来实现单一特性或与其相关工作的分支。
  - 创建特性分支，在提交了若干更新后，把它们合并到主干分支，然后删除，从而支持迅速且完全的进行语境切换。
  - 因为开发工作分散在不同的流水线里，每个分支里的改变都和它的目标特性相关，浏览代码之类的事情因而变得更简单了。
  - 可以把作出的改变保持在特性分支中几分钟，几天甚至几个月，等它们成熟以后再合并，而不用在乎它们建立的顺序或者进度。
  - 特性分支的例子

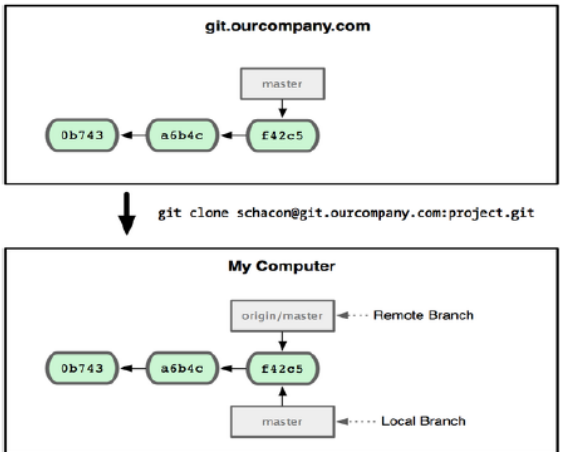




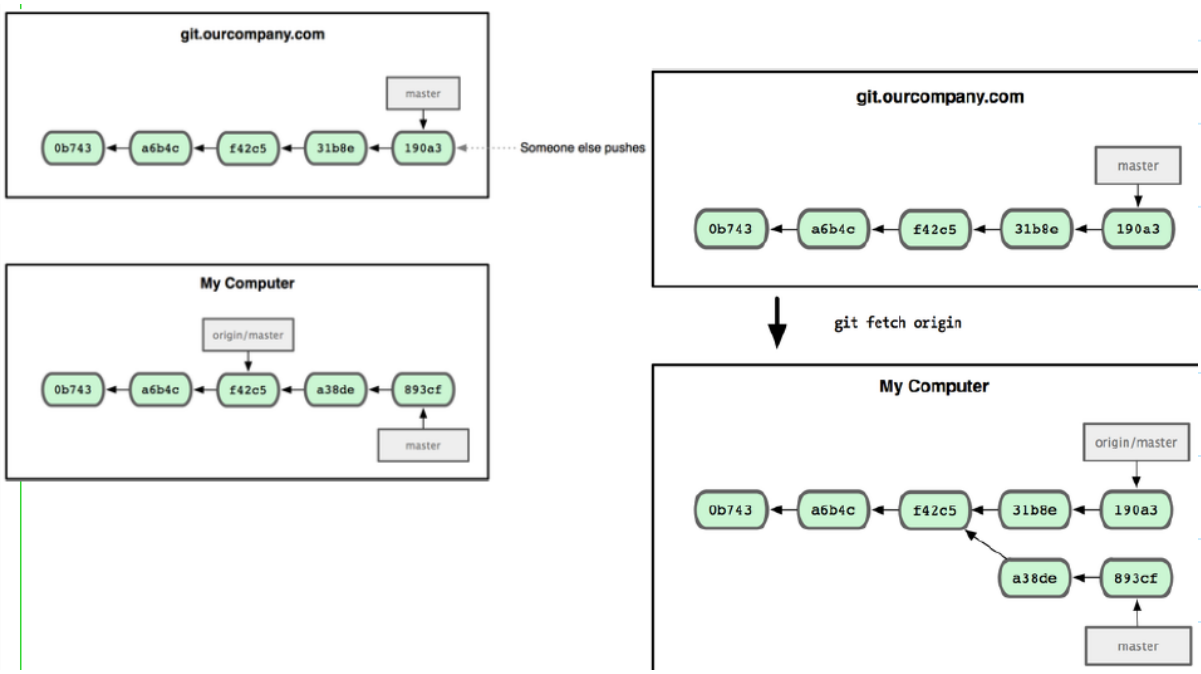
6 远程分支(remotebranch)

- 远程分支对远程仓库中的分支的索引，类似于书签，提醒上次连接远程仓库时上面各分支的位置。
- 使用“(远程仓库名)/(分支名)”表示。
- 远程分支只能看，不能修改

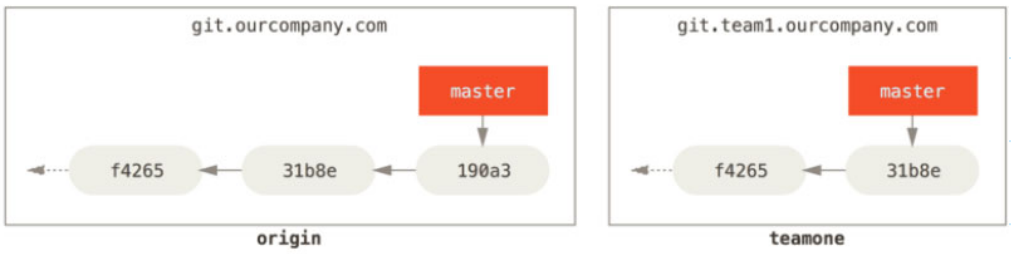




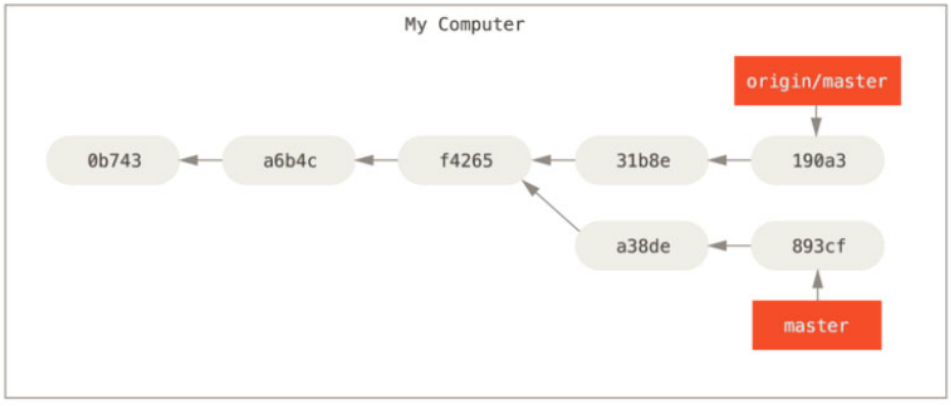
- 远程分支和本地分支的差异

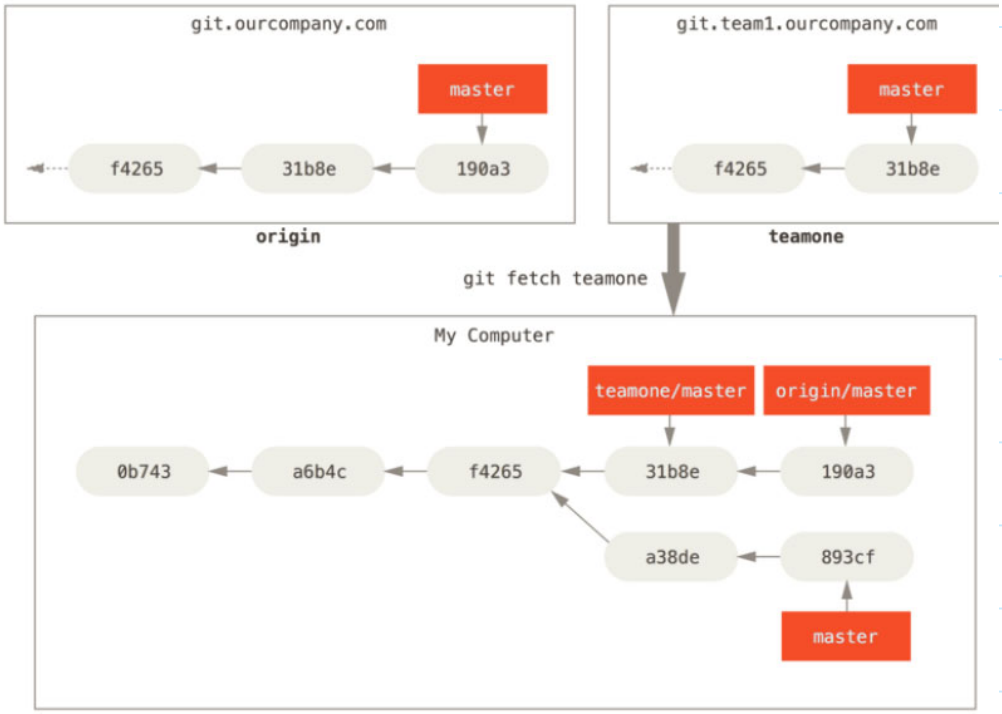


- 更复杂的远程分支：多个远程服务器



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





本地分支<->远程分支(fetch&push)

- Push：推送本地分支至远程
  - 将本地分支推送至有写入权限的远程仓库上。
  - 不能自动同步，由本地开发者自行决定要分享哪些新工作给其他人；
  - git push [remote] [branch]将本地的当前分支推送到remote服务器的branch分支上。
    - 但是，如果该远程分支已经被其他人做过更新，则本次推送无法生效。
    - 解决办法：将其他人的更新fetch到本地，在本地合并之后，再重新push。
- Fetch：将远程分支同步到本地
  - git fetch [remote] [branch]
  - git merge remote/branch：将远程分支的变化合并到本地当前分支

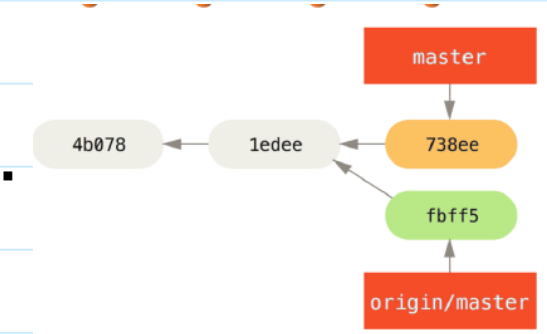
跟踪远程分支

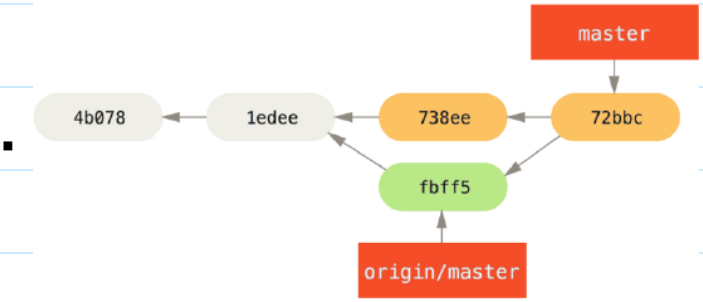
- 跟踪分支是与远程分支有直接关系的本地分支。
  - git checkout --track origin/serverfix：创建一个叫做serverfix的本地分支，并跟踪origin远程仓库上的serverfix分支；
  - git checkout -b sf origin/serverfix：创建一个名为sf的本地分支，并跟踪origin远程仓库上的serverfix分支；
  - git branch -u origin/serverfix：设置本地的当前分支跟踪origin远程仓库上的serverfix分支。
  - 使用git branch的-vv参数列出各本地分支是否跟踪远程分支、进度如何(领先、落后)；
- 如果在一个跟踪分支上输入git pull，git能自动地识别去哪个服务器上抓取、合并到哪个分支。
  - 等价于git fetch+git merge

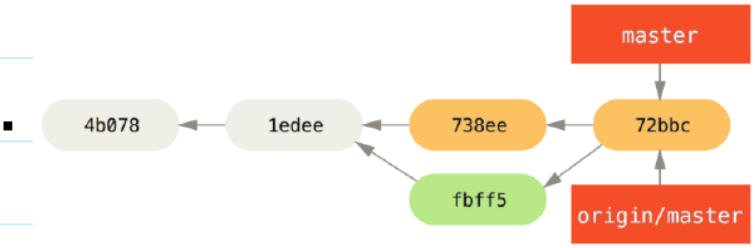
删除远程分支

git push origin --delete serverfix从origin远程仓库上删除serverfix的远程分支

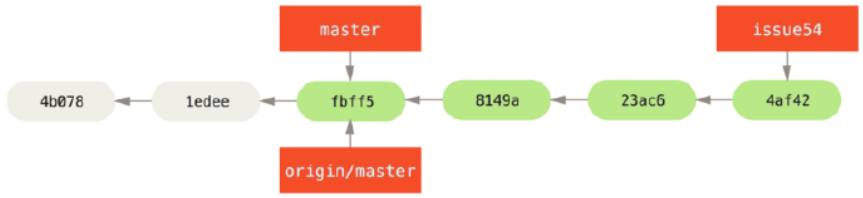
7 使用git进行协同开发的实例

- 某个二人团队John+Jessica
  - John从git服务器上克隆了当前的项目代码，在本地进行了修改和本地提交；
  - So does Jessica；
  - Jessica将其改动推送至服务器上；
  - John也尝试着推送至服务器--rejected(why?)
  - How?
  - John抓取服务器上的最新改动至本地，并与自己的修改进行合并；
  - John再次推送到服务器。
- John的提交历史
  - John的分叉历史
    - 

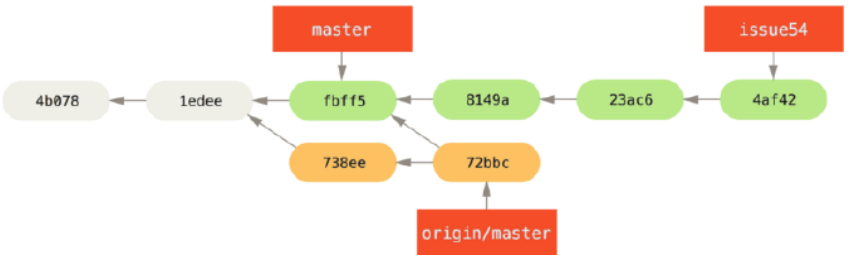
  - 合并了`origin/master`之后John的仓库。
    - 

  - 推送到`origin`服务器后John的历史。
    - 

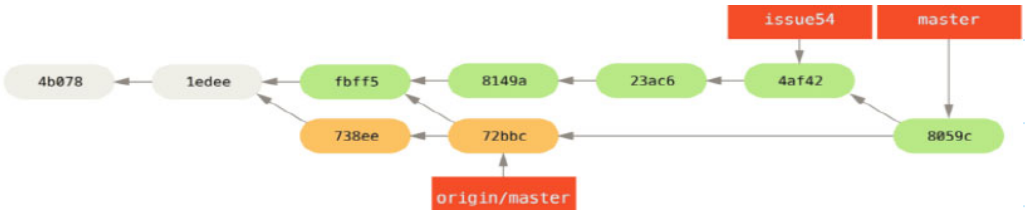
- Jessica的工作
  - 在此期间，Jessica在一个特性分支上工作，她创建了一个称作issue54的特性分支并且在那个分支上做了三次提交。



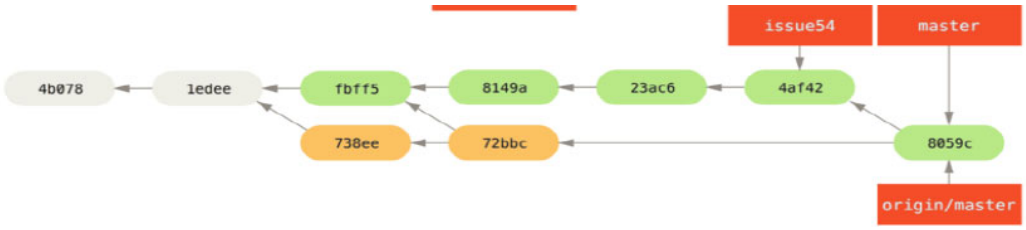
- Jessica想要获取John的最新工作，从服务器获取之；



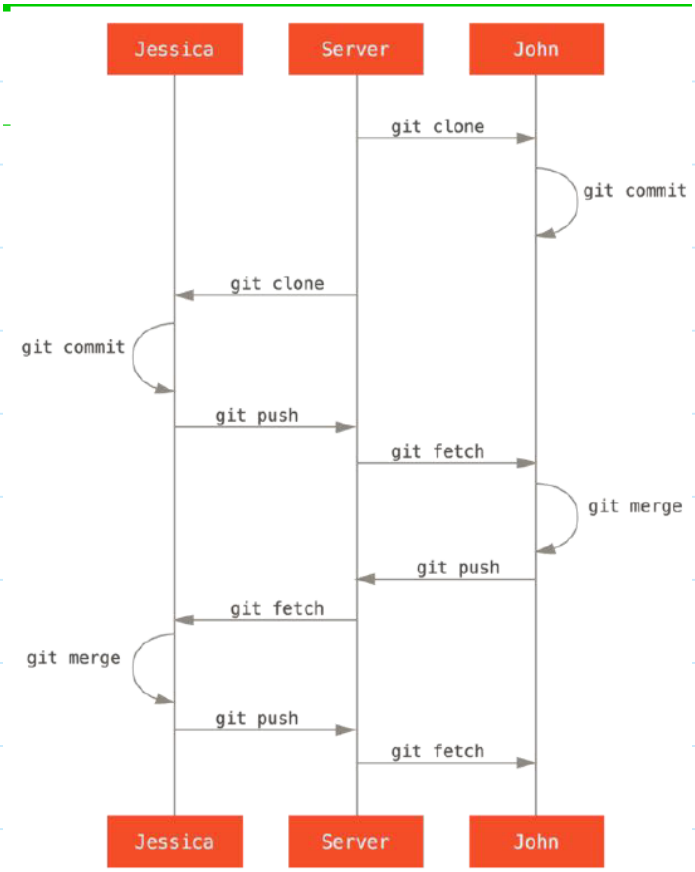
- Jessica认为她的特性分支已经准备好了，可以合并到主分支。
- 她的master之后有两个变化：自己的issue54、John所做的推送。
- 先合并issue54，再合并origin/master；
- 然后推送至服务器。
- 合并了John的改动后Jessica的历史。



- 推送所有的改动回服务器后Jessica的历史。

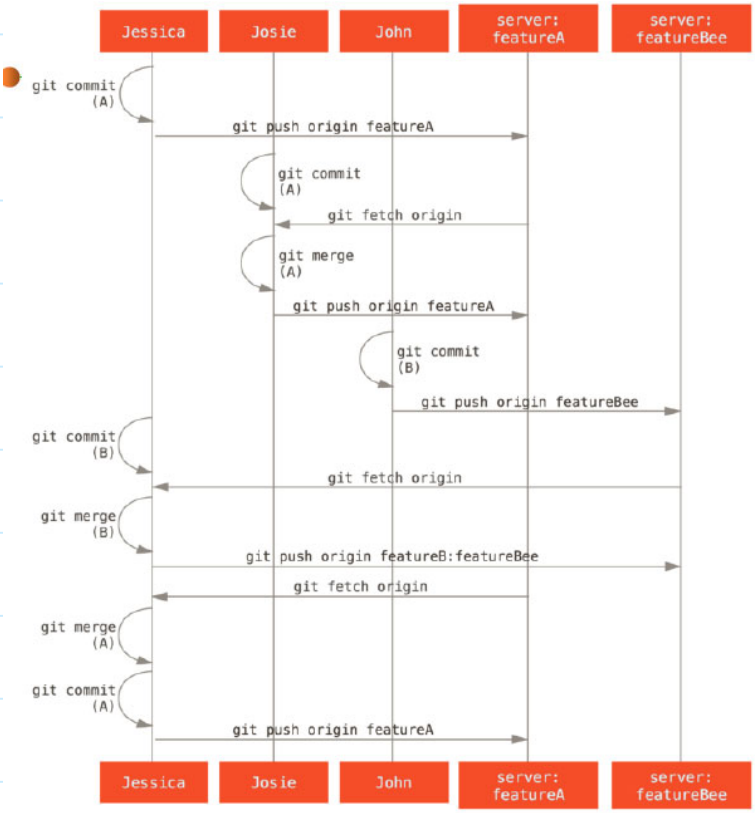


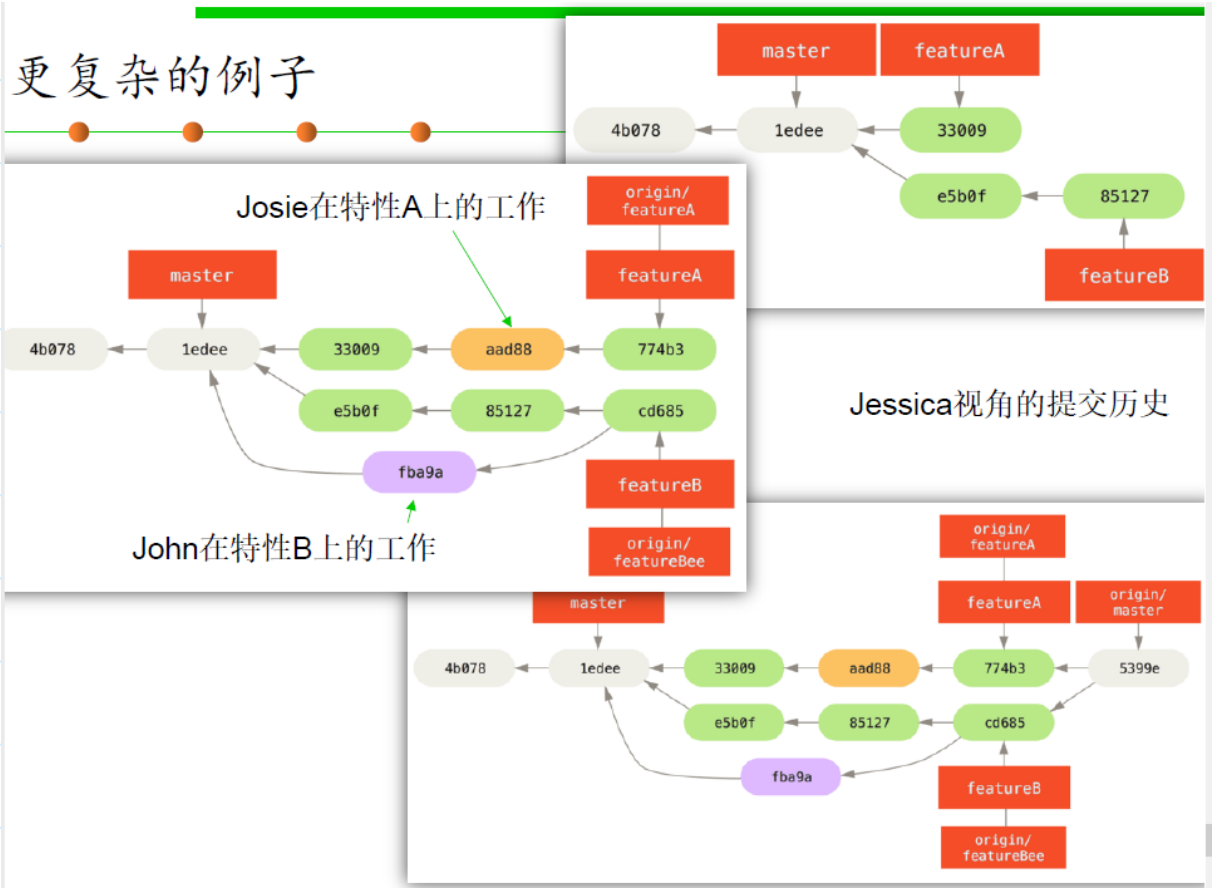
- 小结--一个简单的多人Git工作流程的通常事件顺序



更复杂的例子

- Jessica与Josie在特性A上协作开发，Jessica与John在特性B上协作开发——即：Jessica在两个特性上工作，并且平行地与两个不同的开发者协作；
- 两个Git服务器，分别用于特性A和特性B的协作开发。
- 这种管理团队工作流程的基本顺序。



- 

8 持续集成

- 持续集成:敏捷开发的一项重要实践。
  - MartinFowler：团队开发成员经常集成他们的工作，每个成员每天至少集成一次，每天可能会发生多次集成。每次集成都通过自动化的构建(包括编译，发布，自动化测试)来验证，从而尽快地发现集成错误，大大减少集成的问题，让团队能够更快的开发内聚的软件。
- 价值：
  - 减少风险：不是等到最后再做集成测试，而是每天都做；
  - 减少重复过程：通过自动化来实现；
  - 任何时间、任何地点生成可部署的软件；
  - 增强项目的可见性；
  - 建立团队对开发产品的信心；
- 特点
  - 所有的开发人员需要在本地机器上做本地构建，然后再提交到版本控制库中，从而确保他们的变更不会导致持续集成失败。
  - 开发人员每天至少向版本控制库中提交一次代码。
  - 开发人员每天至少需要从版本控制库中更新一次代码到本地机器
  - 需要有专门的集成服务器来执行集成构建，每天要执行多次构建
  - 每次构建都要100%通过。

- 每次构建都可以生成可发布的产品。
- 修复失败的构建是优先级最高的事情