Git Learning

Albert Zhang

Google Developers

GDG Tianjin

为啥要版本控制？

- 有历史可追溯。
  - 这下任何文件都有历史了，从创建开始的所有历史。
  - 再改东西，就不怕改不回来了。
  - 想知道昨天写的和今天有啥不同。
- 大家一起协作。
  - 这个东西到底是谁弄得，为啥？
  - 为什么我的东西没了，被别人覆盖了还是删了？
  - 需要同时做好多事情呀。

什么是版本控制系统

一个Version Control System（VCS）应该是这样滴：

1. 有一个仓库（Repository）来保存各种文件。
2. 记录什么时候，改了什么，以及为什么改。
3. 可以为不同场景分别开发，Branch（分支）。
4. 可以设置重要的里程碑，以便参考和整理。Tag（标签）。
5. 方便协作。

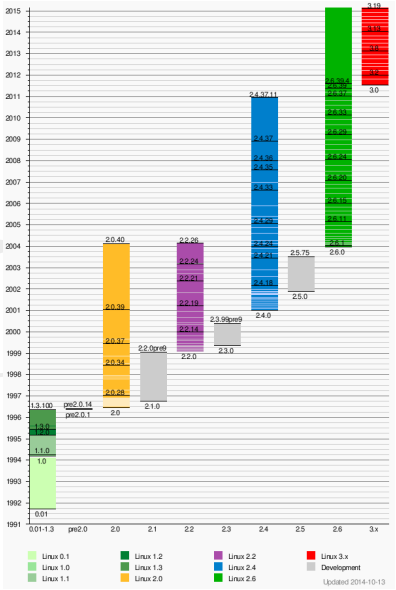
有哪些版本控制方法呢？

- 本地版本控制（Local VCS）
  - 在自己电脑上的粘帖/复制，如RCS（GNU Revision Control System）
  - 没法协同工作。
- 中心式版本控制系统（Centralized VCS）
  - 一台服务器存储所有版本记录，客户端去同步（更新、提交、删除）。
  - Subversion（SVN，现在的主流）、CVS……
  - 服务器挂了或者连不上怎么办？大家可以歇息了……
- 分布式版本控制系统（Distributed VCS）
  - 本地有完整的Repository，即使没有网络服务器崩了，也可以正常工作，有完整的历史备份。任何一个人都可以恢复完整的Repository。
  - Git、Mercurial（hg）.....
  - 有些学习成本。

git

Git是目前最热门而且开源的DVCS系统，事实上已经是目前主流的版本控制系统了。

Git是Linus Torvalds写的，一开始的目的是为了管理庞大复杂的Linux Kernel项目。



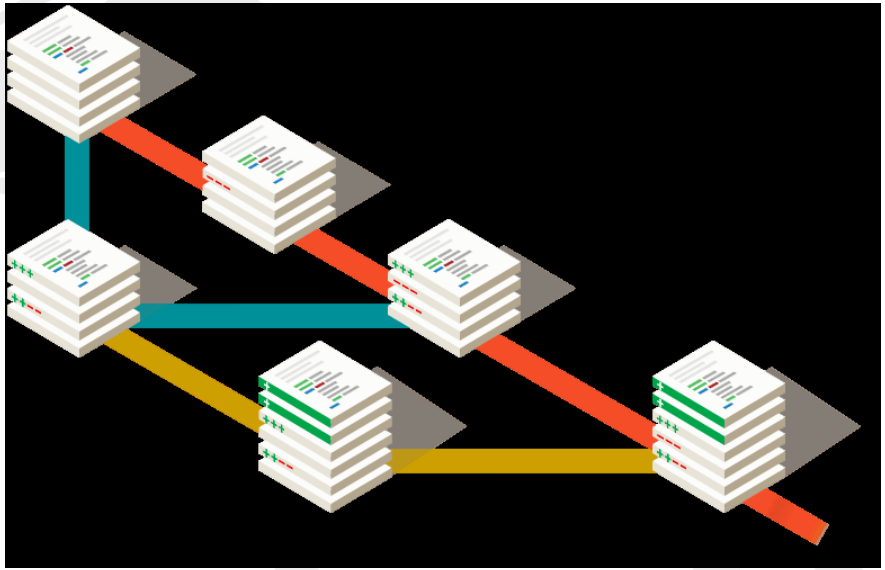
Companies & Projects Using Git

Google facebook Microsoft twitter LinkedIn NETFLIX PostgreSQL

android RAILS Qt GNOME eclipse

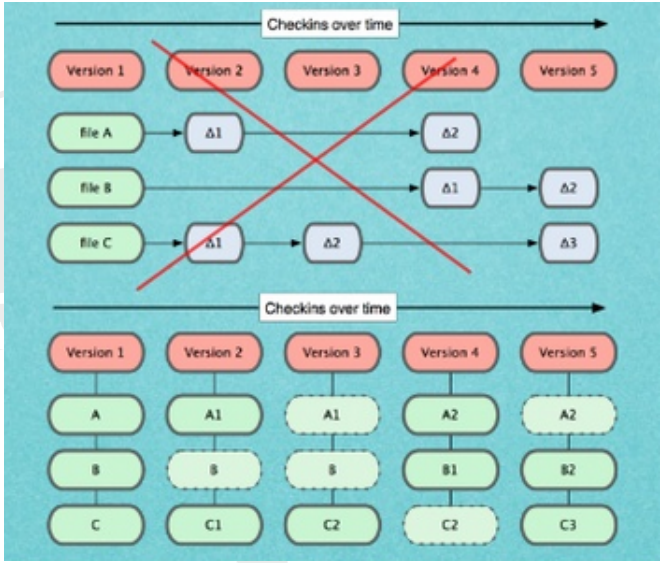
Git的设计目标是：

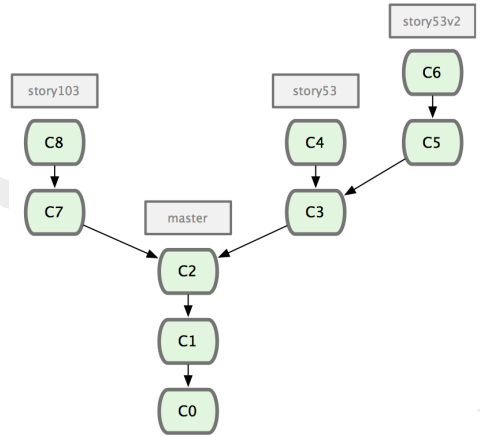
- 要求快，性能好
- 简单易用
- 支持非线性开发
- 去中心化，完全分散式的
- 能管理Linux Kernel这样的巨大项目



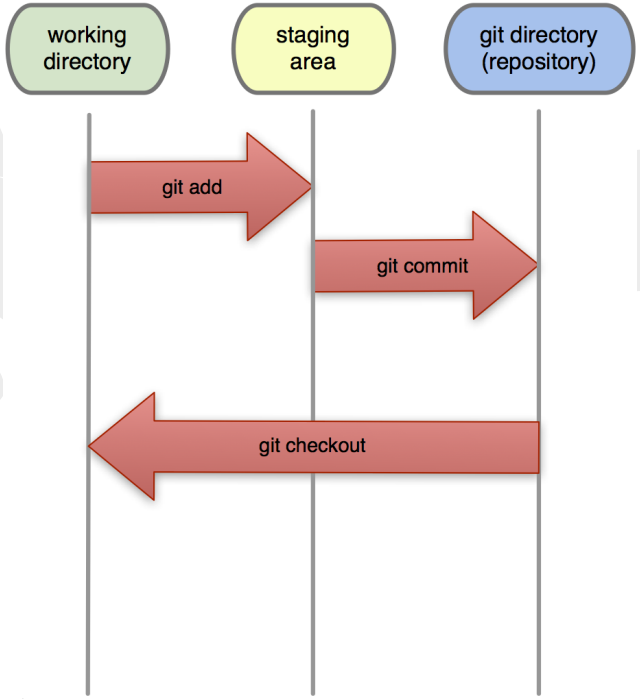
Git的基础知识：

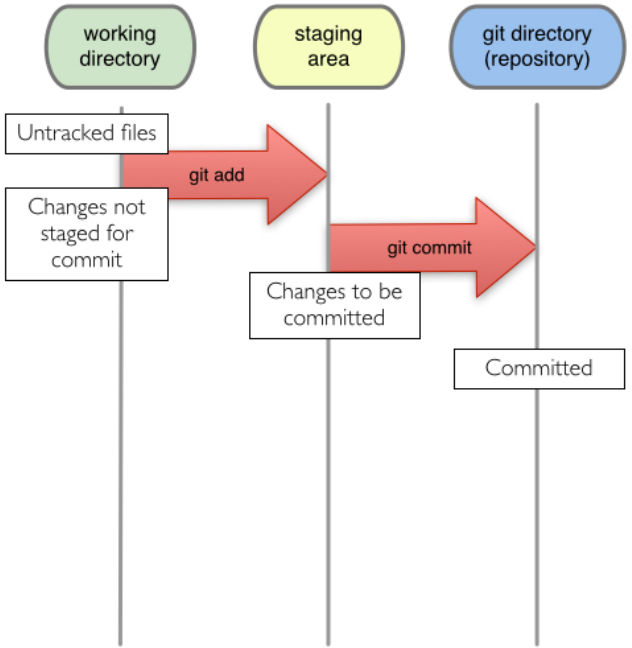
- 不同于传统的Delta方式：记录每次文件的变更，开分支需要建立副本。Git使用DAG（Directed Acyclic Graph）方式，利用有向无环图来记录元数据构建出的Snapshots（快照）。
- 几乎全部操作都在本机进行
  - 不只是一个完整的仓库，查看版本历史、提交更新这些操作都可以在本地完成。
  - 不需要网络也能单独工作。
- 根据文件内容来定位，Checksum（SHA1）是根据内容产生，跟文件名无关。
- 支持Mac / Linux / Windows





Git的文件状态：





1

Staging Area是git独有的设计，好处：Working Directory是我们的工作目录，可能很乱包含各种改动，而Staging Area设计让我们可以只提交想要的文件或部分修改，Staging Area也叫Index。不想被git追踪我们也可以放到每个项目的*.gitignore*文件里面。

各类语言项目的.gitignore可以参考：https://github.com/github/gitignore

Git基本操作(以下操作完全不需要任何服 务器):

- 初始化全局配置

```
git config --global user.name "your_name"
git config --global user.email "your@email.com"
git config --list //列出git在该处找到的所有设置
```

- 建立仓库(Repository)

```
git init // 只会在根目录下创建 .git 文件夹
git status // 当前状态
```

- 第一次提交(commit)

```
git add somefile.txt //添加单个文件
git add *.txt //添加所有 txt 文件
git add . //添加所有文件, 包括子目录, 但不包括空目录
git commit -m "add all txt files"
git commit -C head -a --amend //不会产生新的提交历史记录, 复用HEAD留言, 增补提交, 而不增加提交记录
git log //历史记录
git checkout head readme.txt todo.txt //撤销readme.txt和todo.txt的改动
git checkout head *.txt //撤销所有txt文件的改动
git checkout head . //撤销所有文件的改动
```

Git的Style（Git风格建议）——提交（Commit）

- 每次提交尽量只包含近似的功能或修正，不要混合多个 feature或bug fixed。
- 同样，同一个feature或bug fixed（包含unit test）不要分多次提交，尽量用一次提交。
- 多利用squash将单个feature或bug fixed的多个分散提交合并成一个
- 在保证逻辑独立的前提下，尽快尽早提交，这样便于发现错误和追溯。
- 提交尽量保持逻辑顺序和依赖顺序。
- 提交前必须经过测试，杜绝有问题的半成品提交。
- 提交必须写提交摘要（comments），摘要要求：
  - 首行必须用明确简单的一行描述本次提交的概要内容（对于使用Jira等平台的需要根据需要 带上对应的标识）。尽量在50－60字内。
  - 第二行后写上本次提交的 详细描述。可多行段落，每一行尽量在 80字内，便于git log的显示。

（以上内容在 https://github.com/agis-/git-style-guide 基础上进行了一定调整）

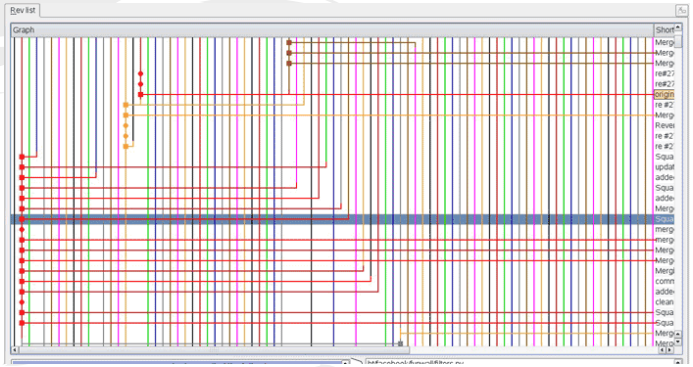
Git的Style（Git风格建议）——分支（Branch）

- 分支命名基本建议：
  - 简短清晰、描述性文字、单词间以破折号连接。杜绝模糊命名；
  - 也可以使用外部标识符（如Jira、GitHub的issue编号）来命名；
  - 多人开发同一个feature时候，分支可以带上开发者标识。
- 合并分支前使用git merge --rebase来让你的分支和主分支保持同步。
- 合并时候不能篡改历史：尤其是master分支或者有特殊含义的分支。
- 合并后除非有特殊原因，建议删除上游分支。
- 在合并分支之前，尽量保持自己的提交历史简单清晰，多数场景尽量不要使用快进。

（以上内容在 https://github.com/agis-/git-style-guide 基础上进行了一定调整）

Git分支(Branch):

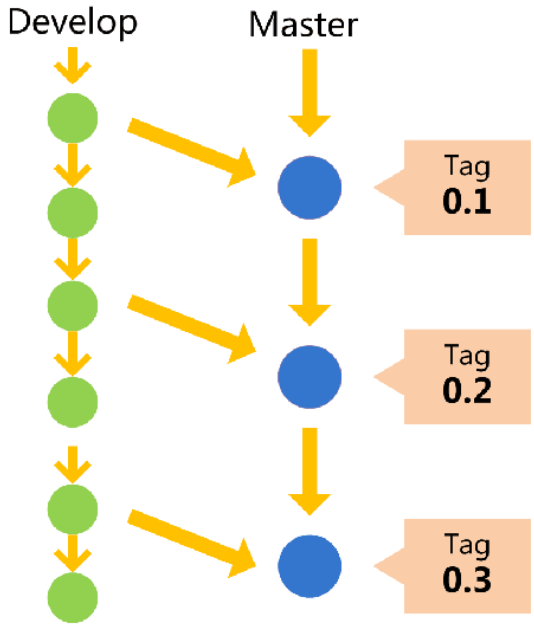
- Git的设计初衷是为了管理复杂大项目，所以Git优点之一就是分支和合并的非常方便。
- 多数传统VCS的分支操作是生成一份现有代码的物理拷贝(如SVN)，而Git只生成一个指向当前版本(快照)的指针，因此成本很低，非常快捷。
- 太方便了也会产生问题，太多分支、太多版本，完全失控。
- 什么时候需要开分支呢？
  - 开发新功能
  - 重构
  - Bugs
  - 实验性的项目.....

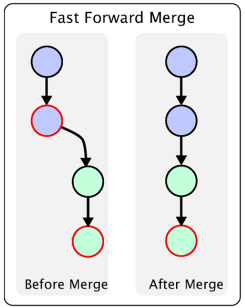


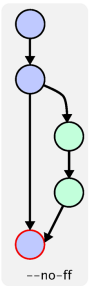
- 因此基于项目管理和团队合作，我们需要一个分支管理策略来进行管理。来保证版本库的主线简洁、历史清晰，分支井井有条，各司其职。
  - GitHub flow
  - Git-flow
  - 自己的flow

Git分支管理策略(类似Git-flow分支策略，仅抛砖)：

- 主分支(Master分支)
  - 有且仅有一个
  - 所有提供给客户的版本(tag)都应该从主分支发布
  - Git主分支默认为Master，在初始化(init)项目时候，自动创建并默认在主分支进行开发。
- 开发分支(暂定名字Develop)
  - Master分支只用来发布重要版本，日常开 发在另一个分支上完成。
  - 这个分支可以用来支持 Nightly Build，持续集成等。
  - 如果需要发布就需要到Master分支上，对Develop分支进行合并，然后再打一个tag。为了保证历史清晰，Merge操作不允许使用Git默认的“快进模式”，必须带上--no-ff参数。
- 临时性分支
  - 用于一些特殊用途的版本开 发，常用有以下：
    - 功能分支(feature)
    - 预发布分支(release)
    - bug修改分支(fixbug)
  - 临时分支使用完后，一般都 应该删除，保证代码库的整洁。

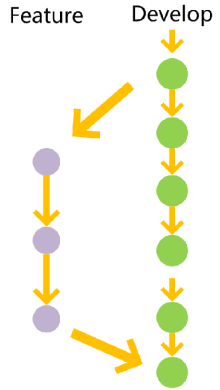


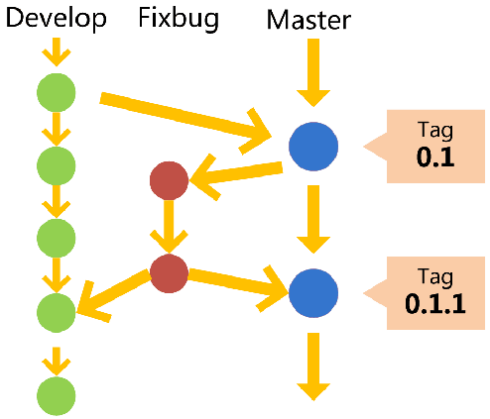




临时性分支举例：

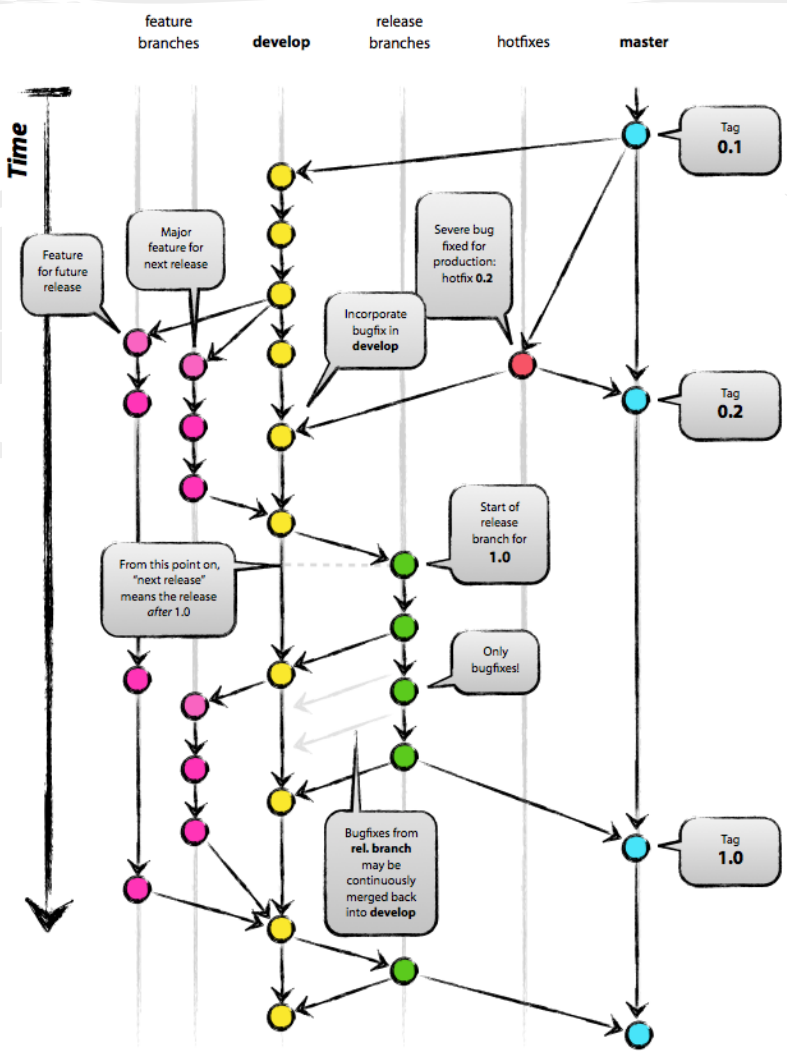
- 功能分支（Feature分支）
  - 开发特定功能时候从Develop分支上fork出来的。
  - 分支命名建议以feature-*作为前缀，比如feature-usermanager。
  - 开发完成后需要merge回Develop分支，需要附加参数--no-ff。然后删除分支。
- 预发布分支（Release分支）
  - 正式发布前（也就是从Develop正式合并到Master之前），我们可能需要产品或集成测试，所以需要一个预发布的分支。
  - 预发布分支也是从Develop分支分出来，但测试结束后，必须将分支上的提交合并进Develop和Master分支。
  - 预发布分支命名可以采用release-*形式，*可以为版本号或者特殊项目/产品代码等。
  - 合并时候原则也是需要加参数--no-ff，完成后删除分支。
- bug修改分支（Fixbug分支）
  - 发布上线后，可能会发现bug，这时候需要对bug进行修改，这时候就需要从Master分支上面分出一个Fixbug分支。
  - 分支命名可以采用fixbug-*形式
  - bug修改完成后，需要将提交合并进Master和Develop分支，merge必须带参数--no-ff。
  - 合并后删除分支。





样例分支管理策略汇总(类Git-flow):

- 主要分支
  - master:永远处于产品发布状态,即与线上产品一致。
  - develop:最新的开发状态。
- 临时性性分支
  - feature:开发新功能,从develop分支出来,完成后需要merge回develop。
  - release:准备要发布的版本,用来测试修改bugs。从develop分支出来,完成后merge回master和develop。
  - fixbug:立即修复的线上(产品环境)bug,从master分支出来,修改完成后merge回master和develop



Review一下Git分支策略

- 分支策略主要流派有：
  - Github flow：没有release分支，pull request，比较适合频繁发布。如web项目。
  - Git-flow：分支追踪清晰，有点复 杂，不太适合 频繁发布。如桌面、app项目等。
- 结合到项目管理方法和流程上：
  - Scrum项目（迭代开 发的）比较适合有release分支的Git-flow。
  - 采用看板模式的，比 较适合快速响应的Github flow。
  - Git分支策略必 须考虑项目全生命周期，除了开 发还有测试（CI、集成测试）、发布。
  - 根据Feature或者Case（User） Story建立分支，分支 时效越短越好，最 长控制在一个迭代周期（最 长2周）。
  - master有变化要及时合并进分支，避免最后 积累很多造成大量冲突，影响合并。所以建 议定期合并（如每天一次或每半天一次）
  - 避免同时开大量的feature分支，给CI、合并增加很大 难度，而且也会 导致大量半成品。
  - 切忌超大branch，当merge回master时候会是一场噩梦。
  - Github的pull request是非常好的Code Review Policy。

Review一下Git分支策略(续)

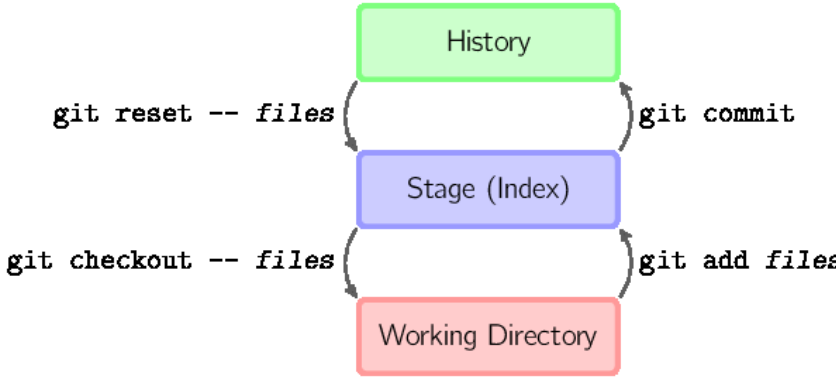
- 分支注意事项:
  - 切换工作目录到某一分支时候, 如果有文件在staging area或者modified状态, 无法切换分支。可以先commit(记住千万别push出去, 这样可以随时reset回来), 事后通过squash整理提交。
  - 也可以使用stash命令来临时存储:*git statsh (apply / clear)*。

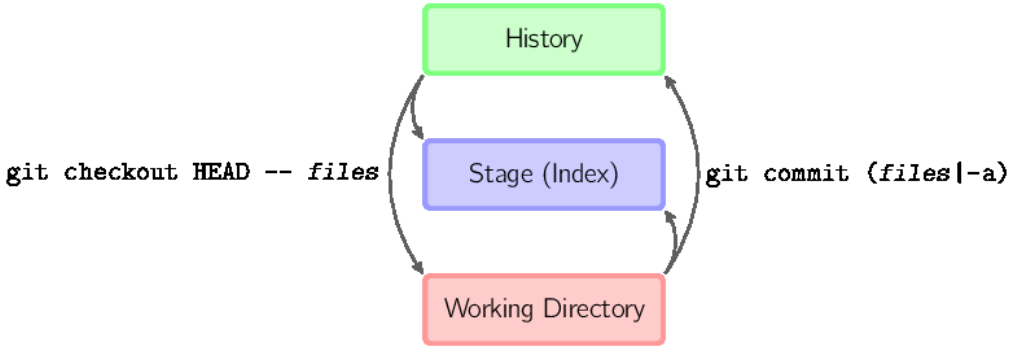
- 参考资料:
  - *A successful Git branching model*
  - *git-flow*
  - *https://www.atlassian.com/git/tutorials/comparing-workflows/*
  - *https://guides.github.com/introduction/flow/index.html*
  - *https://source.android.com/source/developing.html*
  - *http://continuousdelivery.com/2011/05/make-large-scale-changes-incrementally-with-branch-by-abstraction/*
  - *http://martinfowler.com/bliki/FeatureToggle.html*
  - *https://sandofsky.com/blog/git-workflow.html*
  - *https://about.gitlab.com/2014/09/29/gitlab-flow/*

Git合并操作：

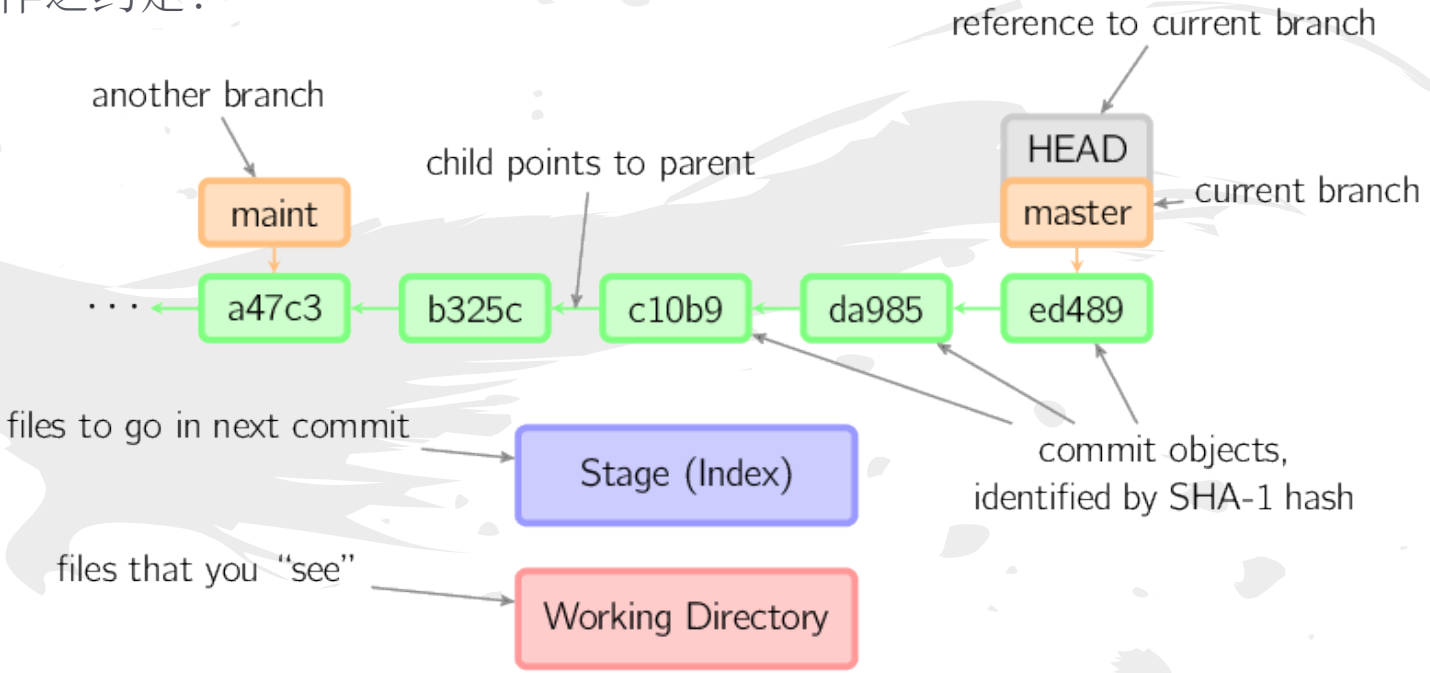
因为Git基本围绕分支来进行(本地分支或后面提到的 remote分支)，所以我们会经常进行合并操作，将多个分支提交合并到当前分支。同 时我们也可以根据项目实际需要来决定合并是否需要改 变实际的提交历史。

先review一下git的基本操作：





Git合并操作之约定：



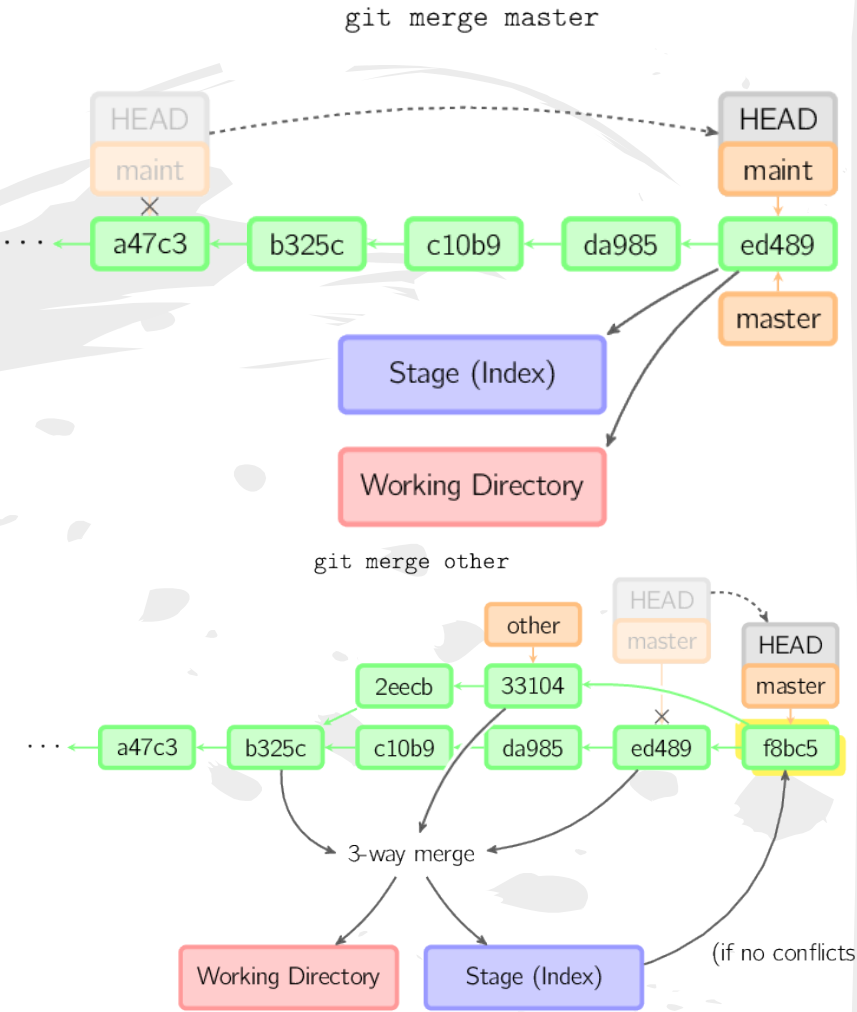
绿色的5位字符表示提交的ID，分别指向父节点。分支用橘色显示，分别指向特定的提交。当前分支由附在其上的*HEAD*标识。这张图片里显示最后5次提交，*ed489*是最新提交。*master*分支指向此次提交，另一个*maint*分支指向祖父提交节点。

Git主要有四种合并方式：merge；merge squash；rebase；cherry-pick。

图片引用自：http://github.com/MarkLodato/visual-git-guide

Git合并操作之merge：

- merge 命令把不同分支合并起来。合并前，索引必须和当前提交相同。如果另一个分支是当前提交的祖父节点，那么合并命令将什么也不做。另一种情况是如果当前提交是另一个分支的祖父节点，就导致快进（*fast-forward*，默认方式）合并。指向只是简单的移动，并生成一个新的提交，如右上图。
- 这种合并会导致提交历史不能清晰表达，一般来说我们使用*--no-ff*参数，这样我们能保留并合并的分支提交历史，并加上一个merge commit，从图线上看会有两条父级线。
- 三方合并（Three-way merge），如右下图：默认把当前提交(*ed489*)和另一个提交(*33104*)以及他们的共同祖父节点(*b325c*)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。
  - Two-way merge：只在2个分支进行合并，如SVN的默认合并。
  - Three-way merge：除了合并2个分支，还要加上两个分支的共同祖先，这样能大大减少人工处理冲突的情况。
- Git可以一次合并多个分支，如：*git merge A B C*

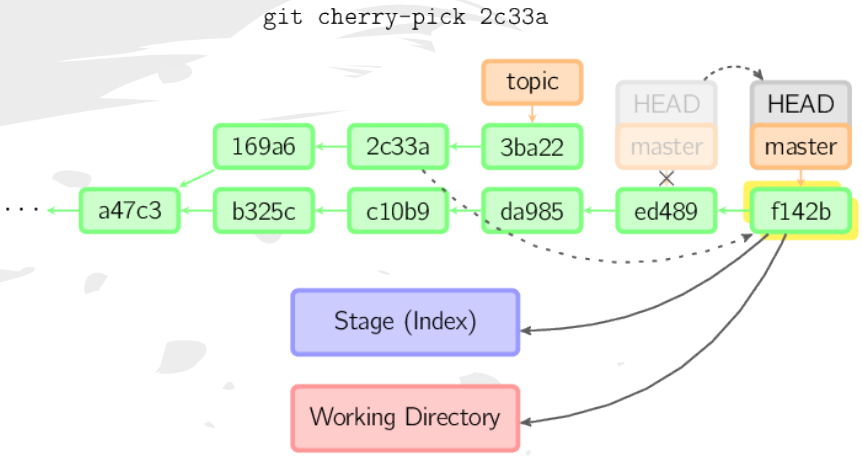


Git合并操作之 *git merge --squash XXX*

- --squash的含义：本地文件内容与不使用此参数的合并结果相同，但是不提交不移动HEAD，需要一条额外的commit命令。
- 通俗来说，其效果相当于将待合并分支的上的多个commit合并成一个，放在当前分支上，待合并分支的commit历史没有拿过来。也就是压缩成只有一个merge-commit，且不会有被合并的log。等同于SVN的默认merge。
- 判断是否采用--squash的标准：待合并分支的历史是否有意义。
  - 如果分支上提交非常随意，甚至有改动就提交这种，那就一定使用--squsn来进行合并。版本历史记录应该是代码的发展历史，而不简单是开发人员在编码时候的具体活动。
  - 如果分支上每次提交都有独立存在意义，尤其是能正确通过编译甚至通过测试的，应该采用默认merge方式保留历史。
- Git的合并策略
  - recursice：当共同祖先不止一个commit，则递归式的进行三方合并。
  - resovle，直接挑选一个共同祖先进行三方合并。
  - octopus，合并超过2个以上的分支。
  - ours，只用自己分支的cmmits。
  - subtree，合成一个子目录。
- 更多：*http://git-scm.com/docs/git-merge*

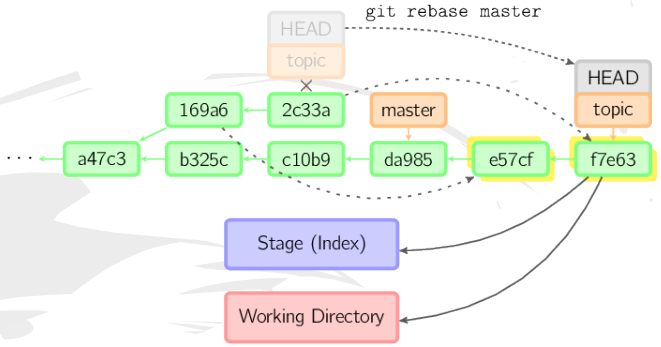
Git合并操作之cherry-pick：

- cherry-pick命令"复制"一个提交节点并在当前分支做一次完全一样的新提交。
- 简单说就是对已经存在的commit进行再次提交。
- cherry-pick完成后，将会产生一个新提交，新提交的哈希值和原有不同，但标识名一样。
- 例如：我们有个稳定分支v1.0，另外还有一个开发分支v2.0，我们不能直接合并两个，这样会导致Stable版本的混乱，但我们有希望在v1.0中增加一个v2.0的新功能，这里就可以使用cherry-pick了。
- 使用*-x*参数，可以在提交的log中加注来自原始哪个commit的哈希。这个参数只有在没有冲突时候才会生效。如果私有项目历史无意义建议不要用此参数。例如：*"(cherry picked from commit ...)"*
- 更多：*http://git-scm.com/docs/git-cherry-pick*

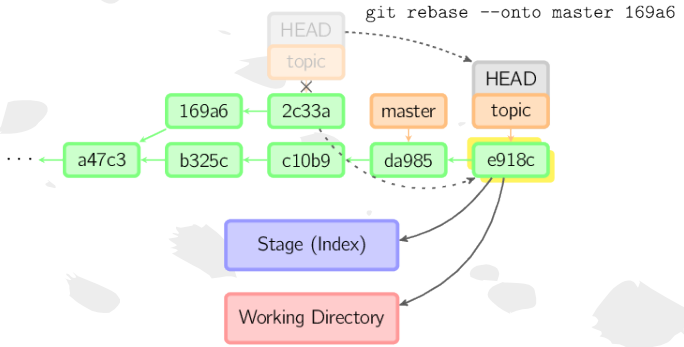


Git合并操作之rebase（衍合）：

- 一般合并操作是把两个父分支合并进行一次提交，提交历史是非线性的。但衍合命令执行的操作：在当前分支上重演另一个分支的历史，提交历史是线性的。**本质上，是线性化的自动cherry-pick**。
- git rebase -i 可以开启互动模式，更方便完成一些复杂操作，比如丢弃、重排、修改、合并提交。
- 注意：衍合（rebase）命令仅适用于本地分支，也就是还没有push出去的分支。如果已经执行push分享出去commits记录，千万不要再rebase后push。
- 在rebase过程中，会遇到冲突，这时候，Git会停止rebase让人工去解决冲突。解决完冲突后，需要使用git add去更新这些内容的index，然后，无需执行git commit，只要执行：*git rebase --continue* 。这样Git会继续apply余下的补丁。
- 在任何时候，都可以用*git rebase --abort* 来终止rebase操作，并且分支会恢复到rebase开始前状态。
- 在实际中我们在feature或其他分支开发完成后需要合并会主develop分支时候需要采用rebase+merge。rebase原因是feature提交会很凌乱，可能包含一些typo，甚至有些commits想合并或拆开。这就需要多次执行rebase，merge提交记得*--no-ff*，rebase之后feature分支（如有远程也有一起）可以删除不要push出去。*如果对图线有洁癖这也是不二的做法 :)*
- rebase时候如果怕有问题，可以开个临时分支来存档，再继续rebase直到ok为止。
- 更多：*http://git-scm.com/docs/git-rebase*

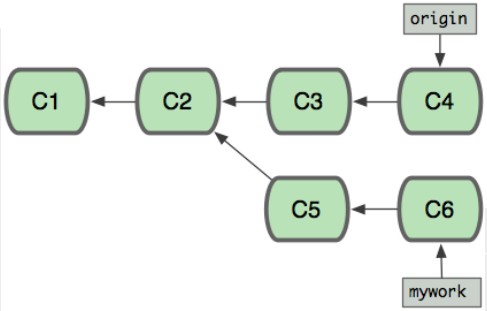


上面的命令都在topic分支中进行，而不是master分支。在master分支上重演，并把分支指向新的节点。注意旧提交没有被引用将会被回收。



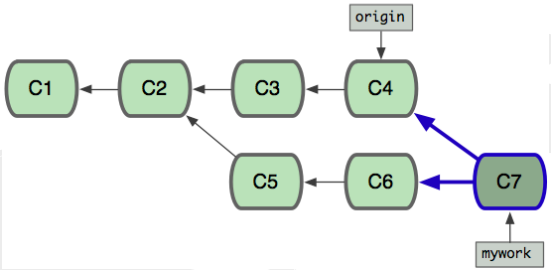
如果要限制回滚范围，使用*--onto*参数。上面的命令在master分支上重演当前分支从169a6以来的最近提交，即2c33a。

Git合并操作之rebase与merge-commit对比：

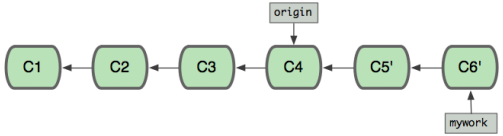


原始结构

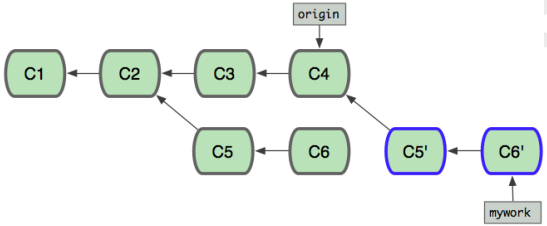
git merge



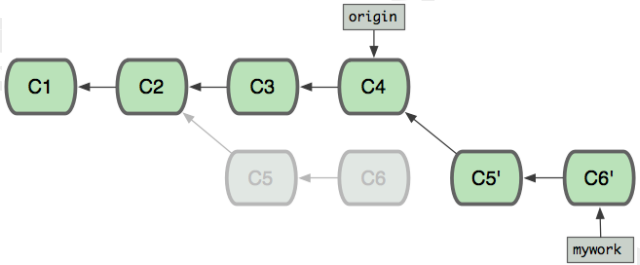
git rebase



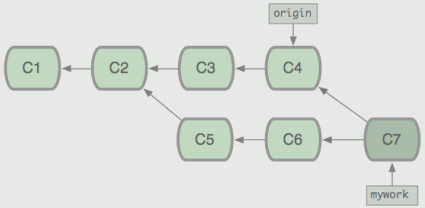
git rebase



执行了merge commit之后



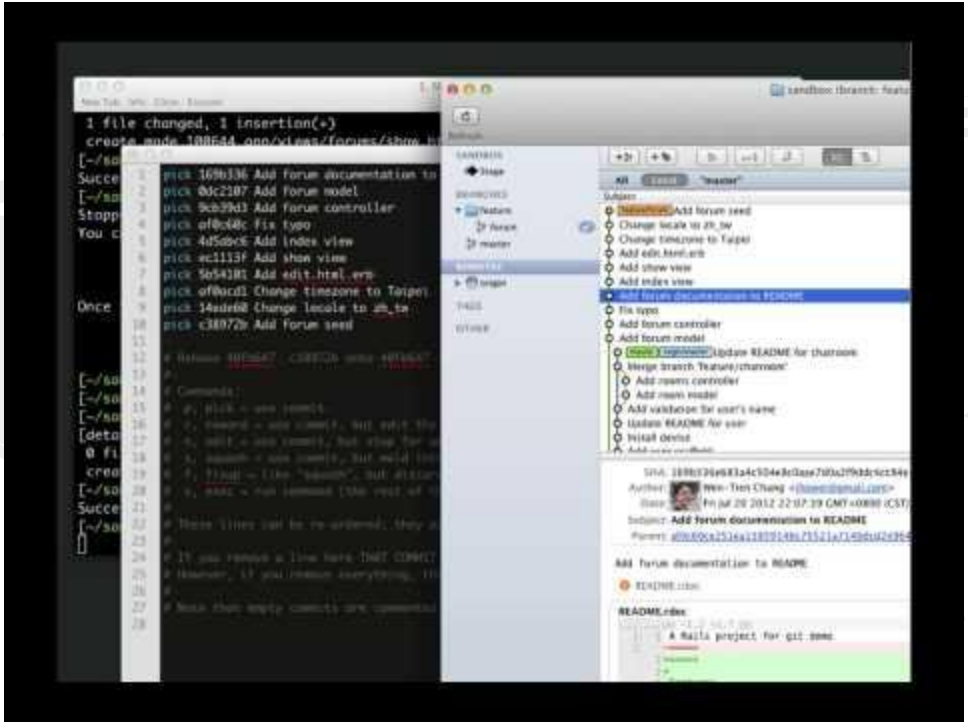
git merge




```
git checkout mywork
git rebase origin
```

这些命令会把你的mywork分支里的每个提交取消，并且把它们临时 保存为补丁(patch)(这些补丁放到".git/rebase"目录中),然后把mywork分支更新到最新的origin分支，最后把保存的这些补丁应用到mywork分支上。mywork分支更新完成后，分支会将指向这些新创建的提交，而那些老的提交会被丢弃。如果允许垃圾收集命令(git gc)，这些被丢弃的提交就会被删除。

一个Git rebase+merge的合并操作示范



Git的Reflog：

- git reflog是对reflog进行管理的命令。
- reflog是git用来记录引用变化的一种机制，比如记录分支的变化或者HEAD引用变化。当reflog命令不指定引用时候，默认列出HEAD的reflog。
- git reflog可以帮我们找到丢失的commit，比如在某个detached HEAD（即不在任何分支只是在某个历史commit的节点上）的时候进行了一次提交，然后切换分支后想把刚才的东西合并进来，这个时候会突然发现刚才那次提交找不到了，这时就可以通过HEAD@{1}引用刚才的体积，通过git reflog中找到对应commit的SHA1值，然后进行merge。

```
2ab4043 (HEAD, refs/heads/master, refs/heads/a) HEAD@{0}: checkout: moving from master to a
2ab4043 (HEAD, refs/heads/master, refs/heads/a) HEAD@{1}: commit (merge): Merge branch 'mybranch'
bf98582 HEAD@{2}: rebase: aborting
bf98582 HEAD@{3}: checkout: moving from 7e9938d9f5b9f7835359ca87da8a329781ed74b6 to master
7e9938d (refs/remotes/origin/mybranch, refs/heads/mybranch) HEAD@{4}: checkout: moving from master to 7
bf98582 HEAD@{5}: reset: moving to bf985821e12129ea3dc9d4150792b8dae798773c
be93372 (refs/remotes/origin/master) HEAD@{6}: commit: add merge detail to read me
3eeedca HEAD@{7}: commit (merge): Merge branch 'mybranch'
bf98582 HEAD@{8}: reset: moving to bf985821e12129ea3dc9d4150792b8dae798773c
00fa1f0 HEAD@{9}: commit (merge): Merge branch 'mybranch'
bf98582 HEAD@{10}: reset: moving to bf985821e12129ea3dc9d4150792b8dae798773c
663feb9 HEAD@{11}: commit: merge mybranch
bf98582 HEAD@{12}: reset: moving to bf985821e12129ea3dc9d4150792b8dae798773c
aa73004 HEAD@{13}: commit: merge issue
bf98582 HEAD@{14}: commit: modify readme file
cd2eddb HEAD@{15}: checkout: moving from mybranch to master
7e9938d (refs/remotes/origin/mybranch, refs/heads/mybranch) HEAD@{16}: commit: modiy readme add issueF
cd2eddb HEAD@{17}: checkout: moving from master to mybranch
cd2eddb HEAD@{18}: commit: prepare files
b39fed8 HEAD@{19}: commit (initial): first commit
```

上图，HEAD@{0}代表HEAD当前的值，HEAD@{2}代表HEAD两次变化之前的值。这里输出了HEAD的所有变化历史，每条记录包含了变化所对应的git操作，如commit、checkout、rebase、merge等，以及变化的详细内容。

- git会将变化记录到HEAD对应的reflog文件中，其路径为.git/logs/HEAD，分支的reflog文件都放在.git/logs/refs目录下的子目录中。文件都是纯文本文件，如下图。
- 对于dangling对象（悬空）：一个对象无法从任何其它对象或引用访问到。但垃圾回收机制无法清理掉，可能是reflog entry的old或new SHA1有引用。这时候就需要git reflog delete ref@{specifier}。但删掉一个entry会导致reflog出现gap，这时候需要加上 --rewrite 选项，这样再删除掉entry后会修改后面被删除后面的entry的old sha1为前一条entry的new值。

每一个reflog的entry都包含变化前commit的SHA1以及变化后的commit的SHA1，第一次commit对应变化的old SHA1值为全0，每一个entry还包含了用户名、email、变化时间戳以及变化的具体内容。

```
0000000000000000000000000000000000000000 b39fed82cd3225eb524f6f0184c0ba49a4f6952c 卢克 <kejinlu@gmail.com> 1366871718 +0800    commit (initial): first commit
b39fed82cd3225eb524f6f0184c0ba49a4f6952c cd2eddb41a632f68b0655366d5ca99f4701bb9b4 卢克 <kejinlu@gmail.com> 1366871885 +0800    commit: prepare files
cd2eddb41a632f68b0655366d5ca99f4701bb9b4 cd2eddb41a632f68b0655366d5ca99f4701bb9b4 卢克 <kejinlu@gmail.com> 1366871950 +0800    checkout: moving from master to mybranch
cd2eddb41a632f68b0655366d5ca99f4701bb9b4 7e9938d9f5b9f7835359ca87da8a329781ed74b6 卢克 <kejinlu@gmail.com> 1366872039 +0800    commit: modiy readme add issueFile
7e9938d9f5b9f7835359ca87da8a329781ed74b6 cd2eddb41a632f68b0655366d5ca99f4701bb9b4 卢克 <kejinlu@gmail.com> 1366872046 +0800    checkout: moving from mybranch to master
cd2eddb41a632f68b0655366d5ca99f4701bb9b4 bf985821e12129ea3dc9d4150792b8dae798773c 卢克 <kejinlu@gmail.com> 1366872107 +0800    commit: modify readme file
bf985821e12129ea3dc9d4150792b8dae798773c aa7300408b6865b105d196c1acf60dc83ffccef1 卢克 <kejinlu@gmail.com> 1366872272 +0800    commit: merge issue
aa7300408b6865b105d196c1acf60dc83ffccef1 bf985821e12129ea3dc9d4150792b8dae798773c 卢克 <kejinlu@gmail.com> 1366872623 +0800    reset: moving to bf985821e12129ea3dc9d4150792b8dae798773c
bf985821e12129ea3dc9d4150792b8dae798773c 663feb94a406a8d3600f2d24bf4dafc565a5f9da 卢克 <kejinlu@gmail.com> 1366872760 +0800    commit: merge mybranch
663feb94a406a8d3600f2d24bf4dafc565a5f9da bf985821e12129ea3dc9d4150792b8dae798773c 卢克 <kejinlu@gmail.com> 1366872810 +0800    reset: moving to bf985821e12129ea3dc9d4150792b8dae798773c
bf985821e12129ea3dc9d4150792b8dae798773c 00fa1f0727bca42305b2a7c5dc53f44c33a17a96 卢克 <kejinlu@gmail.com> 1366872883 +0800    commit (merge): Merge branch 'mybranch'
00fa1f0727bca42305b2a7c5dc53f44c33a17a96 bf985821e12129ea3dc9d4150792b8dae798773c 卢克 <kejinlu@gmail.com> 1366873467 +0800    reset: moving to bf985821e12129ea3dc9d4150792b8dae798773c
bf985821e12129ea3dc9d4150792b8dae798773c 3eeedca7ea0b821088faba84b3157493eae4e13d 卢克 <kejinlu@gmail.com> 1366873579 +0800    commit (merge): Merge branch 'mybranch'
3eeedca7ea0b821088faba84b3157493eae4e13d be933721a15b18605aaf9d4d9f5c5eff3281b9b4 卢克 <kejinlu@gmail.com> 1366874038 +0800    commit: add merge detail to read me
```

查找问题的利器:Git Bisect(二分查找)

- 如果代码出现问题,但不知道哪里出现问题,并且自上次已知的正确提交之后已经有很多个提交了,这个时候可以用bisect来帮助查找,该命令会对提交历史进行二分查找来帮助尽快找到是哪一个提交引入了问题。
- 首先执行git bisect start来启动,接着执行git bisect bad来告诉系统当前所在的提交是否有问题。然后告诉bisect已知的最后一次正常提交是哪次,使用 git bisect good [good_commit],如下左图:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

- 续上,git发现正确与当前错误之间有6个提交,于是git检出中间的提交,我们可以测试看看这个提交下是否有问题,用git bisect good/bad来告诉git,git会继续寻找(good继续向后找,bad向前找),直到git拥有的信息可以明确引入问题的提交是哪里,它会告知第一个错误提交的SHA1值并显示一些提交说明,以及哪些文件在那次体检中修改过,这样我们就可以找到引入bug的根源。如上右图。
- 最后执行git bisect reset重置HEAD指针回到开始的位置。

查找问题的利器:Git Blame(文件标注)

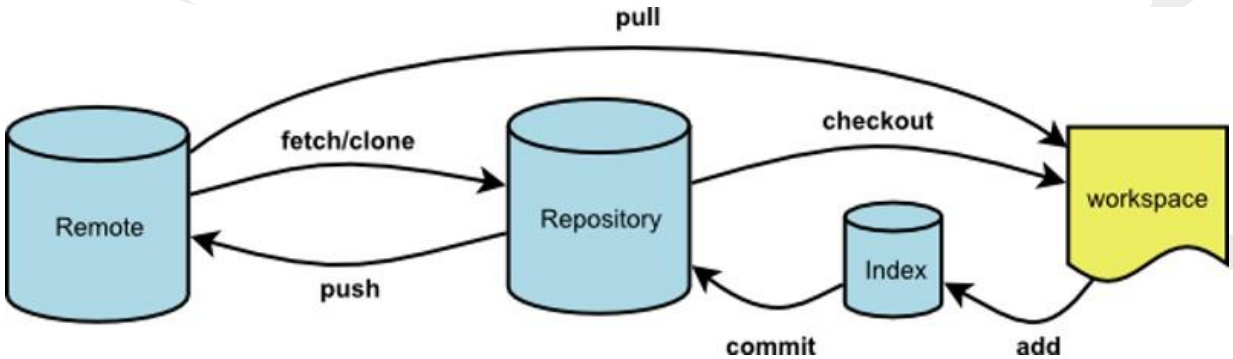
- 如果要检查文件的每个部分是谁修改的,我们只要执行git blame [filename],就可以得到整个文件的每一行的详细修改信息:包括SHA1、日期和作者,如下左图。当然也可以用-L参数在命令中指定开始和结束行,如见下中图。如果带上-C参数,git会分析正在标注的文件,并且尝试找出文件中从别的地方复制过来的代码片段的原始出处,如下右图。
- 这个命令可以帮忙我们追踪代码中的一个bug,并且知道什么时候为何会引入。这在文件被修改了(reverted)或者编译失败了时候可以大显身手!

```
$ git blame sha1_file.c
...
0fcfd160 (Linus Torvalds  2005-04-18 13:04:43 -0700    8)  */
0fcfd160 (Linus Torvalds  2005-04-18 13:04:43 -0700    9) #include "cache.h"
1f688557 (Junio C Hamano  2005-06-27 03:35:33 -0700   10) #include "delta.h"
a733cb60 (Linus Torvalds  2005-06-28 14:21:02 -0700   11) #include "pack.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   12) #include "blob.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   13) #include "commit.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   14) #include "tag.h"
8e440259 (Peter Eriksen   2006-04-02 14:44:09 +0200   15) #include "tree.h"
f35a6d3b (Linus Torvalds  2007-04-09 21:20:29 -0700   16) #include "refs.h"
70f5d5d3 (Nicolas Pitre   2008-02-28 00:25:19 -0500   17) #include "pack-revindex.h"6
...
```

```
$>git blame -L 160,+10 sha1_file.c
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       160)}
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       161)
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700       162)/*
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700       163) * NOTE! This returns a st
790296fd (Jim Meyering   2008-01-03 15:18:07 +0100       164) * careful about using it.
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700       165) * filename.
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       166) *
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       167) * Also note that this ret
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700       168) * SHA1 file can happen fr
d19938ab (Junio C Hamano 2005-05-09 17:57:56 -0700       169) * DB_ENVIRONMENT environm
```

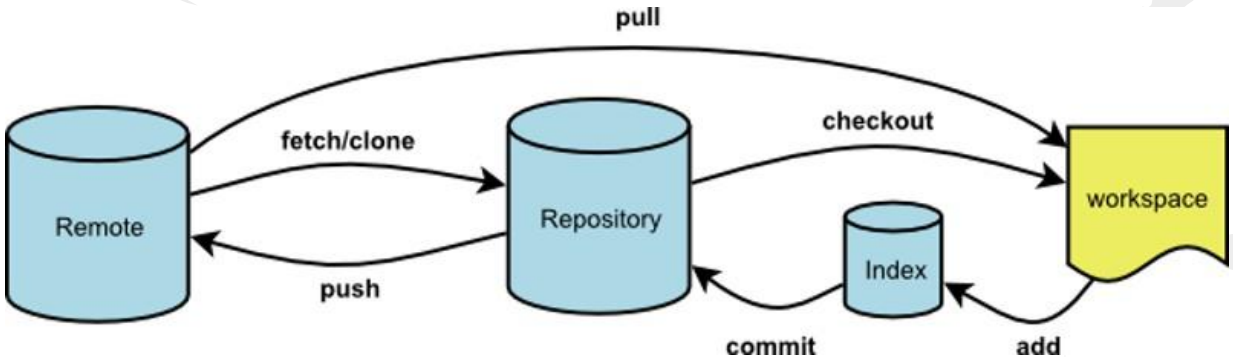
Git远程管理之团队开发协同操作基本命令

- git clone从远程服务器下载资源库，Git的远程服务器有以下几种：
  - Git自己的协议，下载最快：*git clone git://github.com/gdg/sandbox.git/*
  - SSH，安全性最佳：*git clone git@github.com:gdgtianjin/sandbox.git*
  - HTTP/HTTPS，速度最差但能避免被防火 墙阻挡：*git clone https://gdg@github.com/gdgtianjin/sandbox.git*
  - 本地文件协议（file\ftp(s)\rsync...）：*git clone file://path/to/repo.git*
- git remote可以列出所有 远程主机，-v可以查看主机地址。默 认clone时候git都会建立一个origin的远程主机名。可以通 过add/rm/rename等参数来管理 远程主机。
- git merge和git rebase也可以在本地上合并 远程分支。



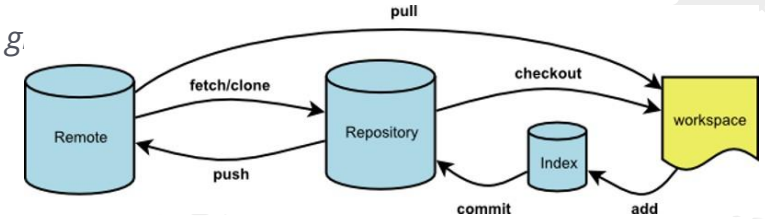
Git远程管理之团队开发协同操作命令：git pull

- git pull从远程更新：先git fetch远程的分支，然后与本地分支做 merge，产生一个merge commit。
  - 命令格式：git pull <远程主机名> <远程分支名>:<本地分支名>
  - 在某些场合，Git会自动在本地分支与远程分支之间，建立一种追踪关系（tracking）。如在git clone的时候，所有本地分支默认与远程主机的同名分支，建立追踪关系，也就是 说，本地的master分支自动"追踪"origin/master分支，如果当前分支与远程分支存在追踪关系，git pull就可以省略远程分支名。手动建立追踪可以：git branch --set-upstream master origin/next
  - 如果合并需要采用rebase模式，可以使用--rebase选项。可以避免杂乱的历史，尤其是多人合作开发尤其适合，可以让历史变得整洁。



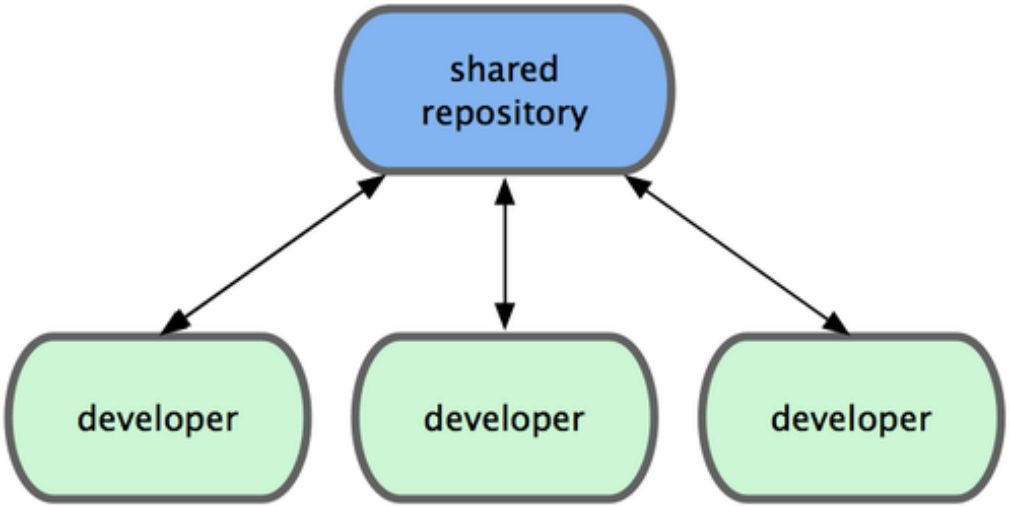
Git远程管理之团队开发协同操作命令：git push

- git push将commit推送出去，命令格式：git push <远程主机名> <本地分支名>:<远程分支名>
  - *git push origin master*，将本地master分支与远程master分支进行快进合并，一般如果出现[rejected]错误，表示需要先从远程pull再push。
  - 省略本地分支名，表示 删除远程分支，因为这等同于推送一个空分支到 远程，如：
    git push origin :master = git push origin --delete master
  - 如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。
  - 如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用git push。
  - 将本地的所有分支都推送到 远程主机，这时需要使用--all选项。如：git push --all origin，如果远程分支不存在会自 动创建。
  - 如果远程主机的版本比本地版本更新，推送 时Git会报错，要求先在本地做git pull合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用 --force选项。如：git push --force origin ，该命令使用--force选项，结果导致远程主机上更新的版本被覆盖。除非你很确定要 这样做，否则应该尽量避免使用--force选项。
  - git push不会推送标签（tag），除非使用--tags选项。如：*g*



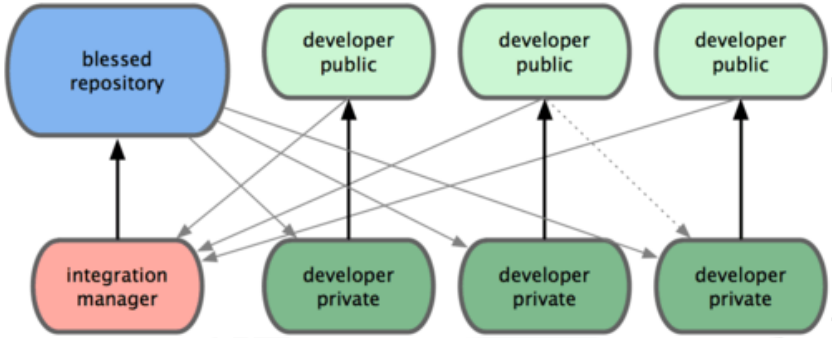
分布式Git:集中式工作流

- 很像传统的SVN的工作方式，这样用Git完全支持。
- 如果两人都修改代码，甚至有冲突，Git不会覆盖他人代码，它直接拒绝第二个人的提交。告知第二个提交者所作的修 订无法通过快进来合并，必须先拉取最新数据下来，手工解决冲突合并后，才能继续推送新的提交。



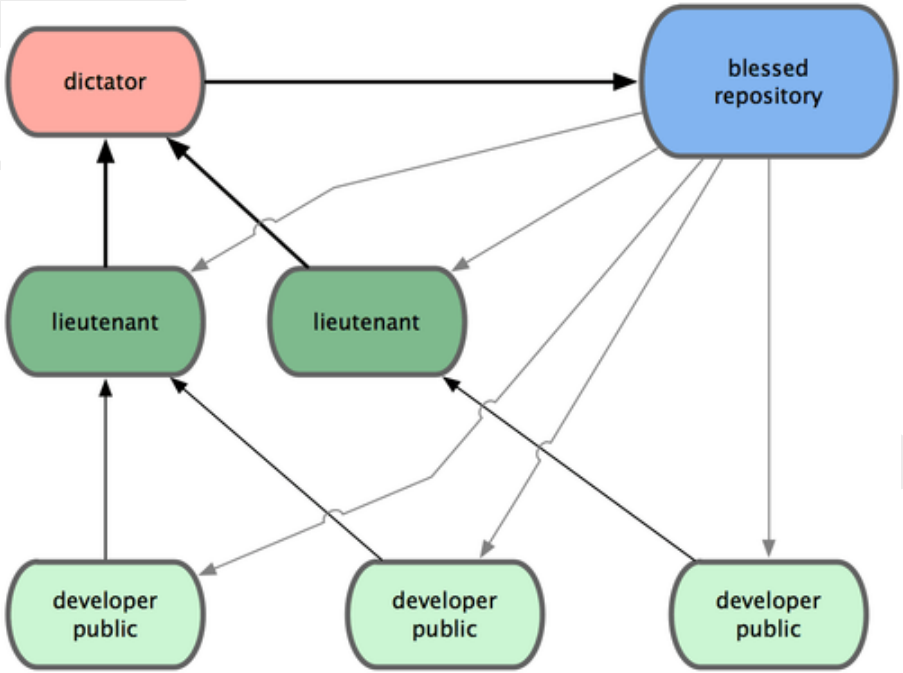
分布式Git:集成管理员工作流

- Github上的典型工作流。
- 流程:项目维护者可以推送数据到blessed repository;贡献者克隆此仓库，编写新代码;贡献者推送数据到自己的公共developer public;贡献者给维护者发送邮件，请求拉取自己的最新修订;维护者在自己本地的integration manger仓库中，将贡献者的仓库加为远程仓库，合并更新并做测试;维护者将合并后的更新推送到主仓库 blessed repository。
- 这么做最主要的优点在于，你可以按照自己的节奏继续工作，而不必等待维护者处理你提交的更新;而维护者也可以按照自己的节奏，任何时候都可以过来处理接纳你的贡献。



分布式Git：司令官与副官工作流

- 集成管理员式工作流的变体。
- 一般超大型的项目才会用到这样的工作方式，如Linux 内核项目（数百名开发者）。
- 各个集成管理员分别负责集成项目中的特定部分，所以称为副官（lieutenant）。
- 总管理员负责整个项目，维护blessed库。



Git团队协作

- 提交几个重点提示：
  - 不要在更新中提交多余的白字符。在提交之前，先运行 git diff --check，会把可能的多余白字符修正列出来。
  - 每次提交限定于完成一次逻辑功能。并且可能的话，适当地分解为多次小更新，以便每次小型提交都更易于理解。
  - 如果针对两个问题改动的是同一个文件，可以试试看 git add --patch 的方式将部分内容置入暂存区域。
  - 提交务必记得写comments。

- 一个简单的小型团队协作示范

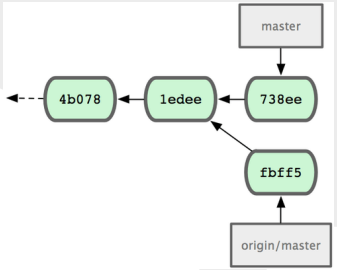


图1

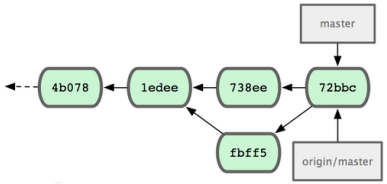


图2

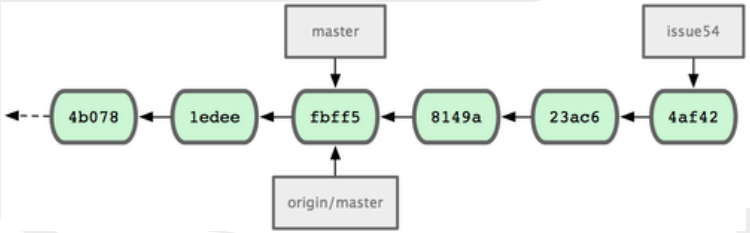


图3

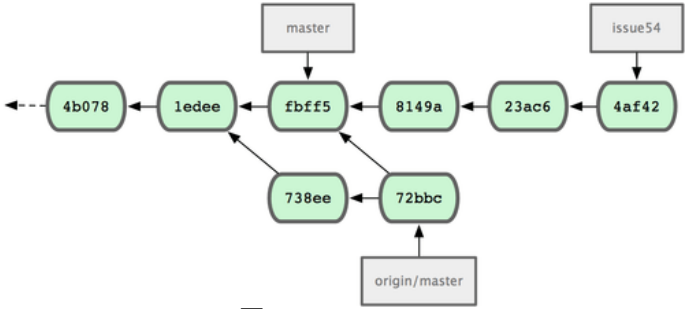


图4

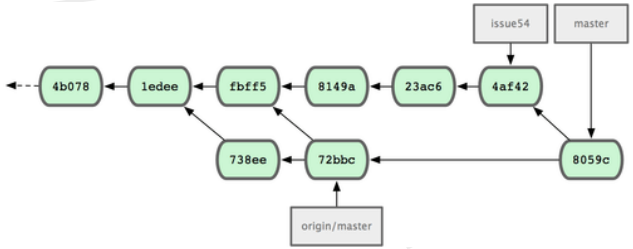


图5

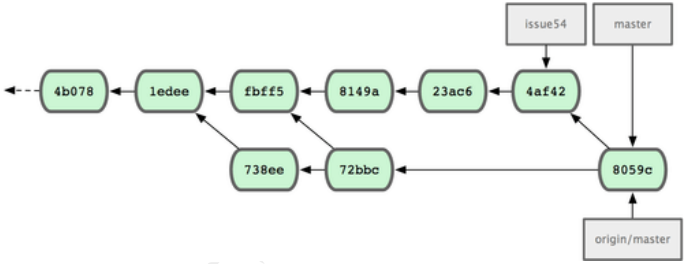
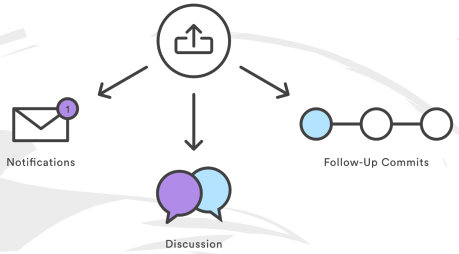
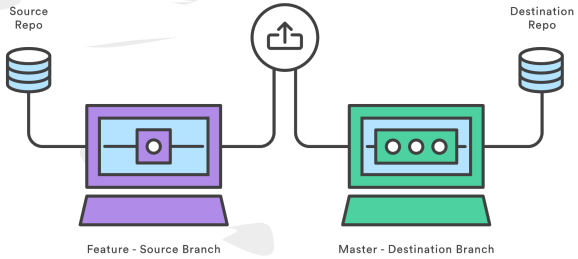


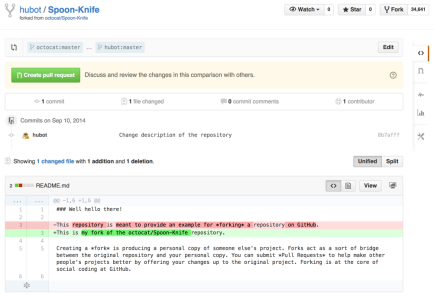
图6

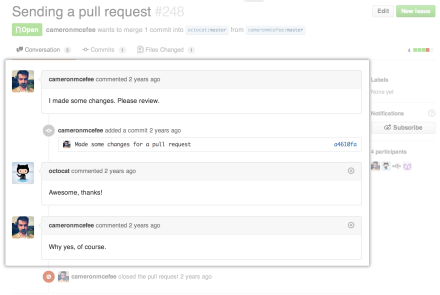
杀手级功能Pull Request（后面用PR代表）

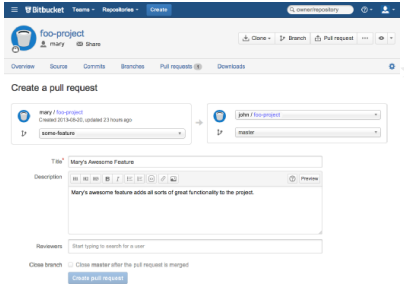
- Pull Request是GitHub、Bitbucket等的杀手功能，多人协作极其威力强大，是现在很多开源项目的主要协作方式。
- 其实git自身也提供了非常粗浅的git request-pull命令，可以将change通过email发给审核人员，非常麻烦也不直观。GitHub们给完善优化，完成了神一般的进化。
- 不管有没有写入权限都可以提交一个PR，请求项目合并你的分支。
- **搭配团队的Code Review，让Code Review更容易更方便更透明，**比起传统SVN，简直甩出去好几条街了。
- 更多阅读
  - https://www.atlassian.com/git/tutorials/making-a-pull-request/
  - https://help.github.com/articles/using-pull-requests/
  - http://doc.gitlab.com/ce/api/merge_requests.html
  - http://git-scm.com/docs/git-request-pull

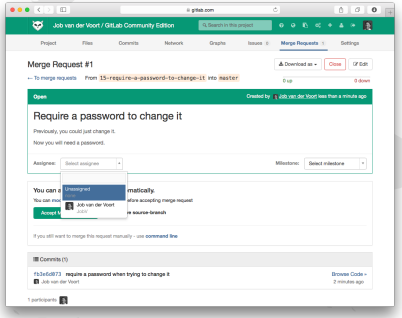






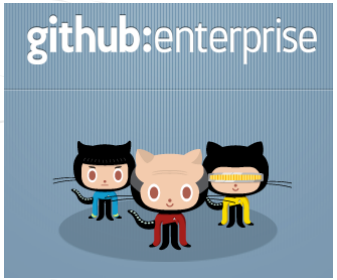






Git托管服务

- GitHub
  - 代言人：Octocat（= Octopus + Cat）
  - 公开的项目无限免费，私有项目收费，最低 7usd/mo
  - 提供收费私服：GitHub Enterpris



- Atlassian Bitbucket Atlassian
  - 私有项目无限免费（5个user），公开项目和多加用户收费。
  - 提供收费私服： Stash Starter License 10 users 10 $
- GitCafé
  - 国产，完全免费
  - 目前在内测私服企业版，近期推出，对于创业团队据说价格很低廉。
- GitLab
  - 免费，可以建公开也可以私有 项目。
  - 提供免费的社区版私服，也提供付 费的企业版私服。

GitHub Tips

- 大量的资源：https://help.github.com/
- 基于Git的轻型wiki系统：https://github.com/gollum/gollum
- 写Blog：Git + GitHub + Markdown + Jekyll
- 免费的项目主页服务：https://pages.github.com/
- 阅读跟踪开源代码的好工具：https://sourcegraph.com/
- 基于GitHub的演讲PPT服务：https://speakerdeck.com/
- 不比较空白字符：在任意 diff页面的URL后加上?w=1，可以去掉那些只是空白字符的改 动，使你能更 专注于代码改动
- 在 diff 或文件的 URL 后面加上 ?ts=4 ，这样当显示 tab 字符的长度时就会是 4 个空格的长度，不再是默认的 8 个空格。
- Gists 方便我们管理代码片段，不必使用功能 齐全的仓库。
- Git.io是GitHub提供的短网址服 务。
- 配合Travis CI进行持续集成和构建。
- GitHub提供了专用的客户端，仅适用于GitHub。
- 大家一起去探索更多的有趣有用的 东西.....

摘录：https://github.com/tiimgreen/github-cheat-sheet
http://www.yangzhiping.com/tech/github.html

Git Tips

- 可以使用alias，减少命令字数，可以编辑.gitconfig里面的[alias]项目。如右边列出一些比较常用的alias，最后一行是一个组合命令。
- 建立没有父的独立branch，典型应用就是GitHub Pages。建立毫不相干的分支主要用来保存其它信息，就像目录一样，不需要被合并到Master。可以执行右侧示例操作。
- 个性化的bash提示，可以参考RGBA的文章或者zsh扩展。或者自己写一个.bash_profile，如右下图，源码见gist地址。
- 小工具：
  - Git的命令行自动补全功能，需要先安装bash-completion。
  - git svn，git和svn一起用。
  - windows和mac上的GUI神器：SourceTree

Atlassian SourceTree

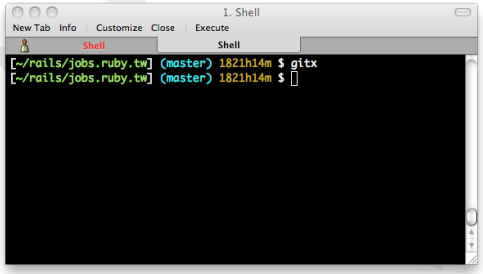
- 个性化设置 git config

```
[
编辑~/.gitconfig

[alias]
  co = checkout
  ci = commit
  st = status
  br = branch -v
  rt = reset --hard
  unstage= reset HEAD
  uncommit = reset --soft HEAD^
  l = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short
  amend = commit --amend
  who = shortlog -n -s --no-merges
  g = grep -n --color -E
  cp = cherry-pick -x
  nb = checkout -b
  type = cat-file -t
  dump = cat-file -p
  addall = !sh -c 'git add . $$ git add -u'
```

```
[
建立一个孤儿分支的操作示例(方法不唯一)

git checkout --orphan newbranch
git rm -rf . #删除所有文件
..... #各种操作, 增加了新文件
git add .
git commit -m 'create new orphan branch!'
```



https://gist.github.com/ihower/897951#file-gistfile1-txt

了解更多

- Git官网：http://git-scm.com/
- Introductory Ref & Tutorial：http://gitref.org/
- Pro Git
- Git Community Book
- http://try.github.com/
- http://think-like-a-git.net/
- Git Ready（使用操作tips）：http://gitready.com/
- Git Immersion：http://gitimmersion.com/
- Git Magic：http://www-cs-students.stanford.edu/~blynn/gitmagic/
- http://documentup.com/skwp/git-workflows-book
- http://www.liaoxuefeng.com/wiki/001373951630592960dd18361248578c67b8067c8c017b000
- https://ihower.tw/git
- Learning Git Branch（互动教学）：http://pcottle.github.io/learnGitBranching/
- 提供15分钟的线上入门体验Try Git：https://www.codeschool.com/courses/try-git
- 大量图示引用Scott Chacon的公开的OminiGraffle：https://github.com/schacon/git-presentations

Google Developers

Just Git It!

谢谢 :)

GDG Tianjin