git 用法简介

目录

插图目录

1 GIT 简介

1.1 概述

GIT --- The stupid content tracker, 傻瓜内容跟踪器。

GIT 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。Linus Torvalds 自嘲地取了这个名字“git”。在英式英语中指一个愚笨或者令人不快的人。

Torvalds 开始着手研发 GIT 是为了作为一种过渡方案来替代 BitKeeper，后者之前一直是 Linux 内核研发人员在全球使用的主要源代码工具。开放源码社区中的有些人觉得 BitKeeper 的许可证并不适合开放源码社区的工作，因此 Torvalds 决定着手研究许可证更为灵活的版本控制系统。

与常用的版本控制工具 CVS、Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持，使源代码的发布和交流极其方便。GIT 的速度很快，这对于诸如 Linux kernel 这样的大项目来说自然很重要。GIT 最为出色的是它的合并跟踪（merge tracing）能力。

- GIT 与 CVS/SVN 的区别

CVS/SVN

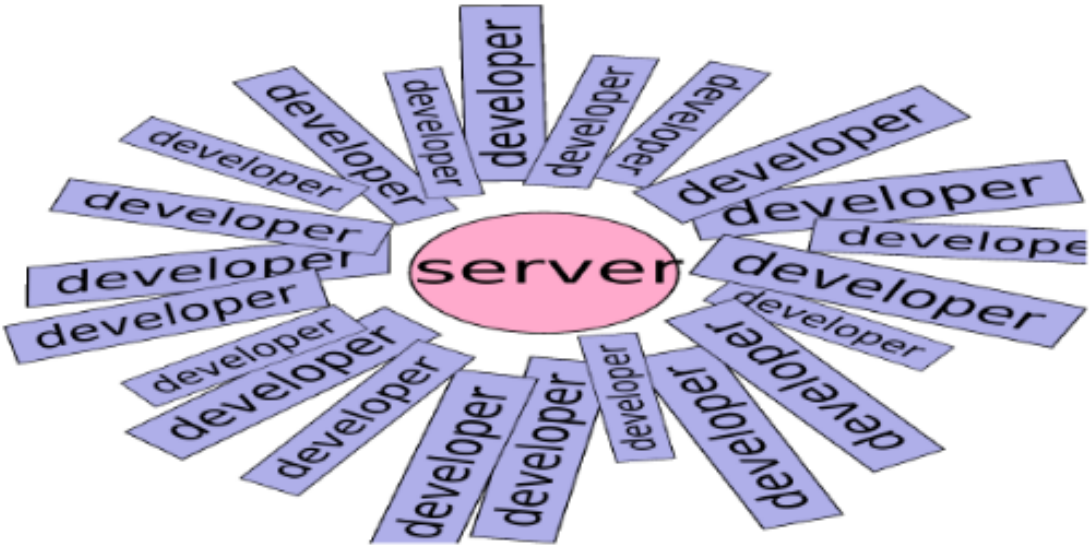

插图 1.1：CVS/SVN 系统结构

- 集中化的版本控制系统。基于服务器，每个人写完代码后都及时的提交到服务器上，进行合并。

- 额外的访问许可策略。如，对文档进行加锁，这样在某个时间只有一个研发人员对中央仓库具备写入权限。
- SCM 不是开发工具，而只是源代码存档的地方。
- 每个人的修改都影响其他所有的开发人员。
- 保存相邻版本之间的差别。

GIT

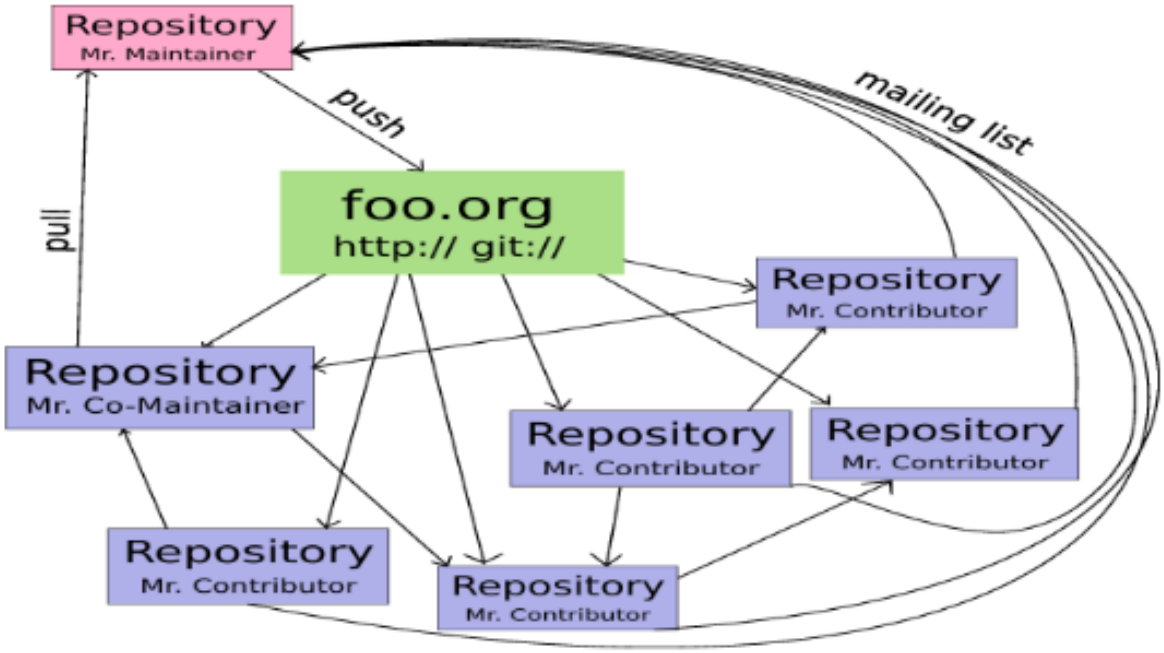

插图 1.2: GIT 系统结构

- 分布式的版本控制系统。每个开发者本地都有一套 git 库，每个人维护自己的版本（或者合并其他人的版本）。
- 所有者可以完全控制他自己的 git 库。
- 支持非线性开发。
- 公布 git 库很容易，可使用如 git://、ssh://、http://……
- 保存所有文件的快照。

1.2 基本概念

1.2.1 SHA

git 最初的设计并非是版本管理系统，而是一个基于内容的文件系统。所谓基于内容，即采用内容的 SHA-1 值作为存储的路径。SHA-1 值为 40 位 16 进制数的字符串，其中前 2 位作为目录名，后面 38 位作为文件名，git 系统可以通过已知的 SHA-1 值迅速定位到对象。

对每个文件来说，这个 SHA-1 值是独一无二的，git 系统可以通过比较 SHA-1 值来确定两个文件是否相同。另外，git 系统还可以通过比较记录的 SHA-1 值和文件现在的 SHA-1 值来检查该文件是否出错。

1.2.2 objects

在 git 版本管理系统中，最重要的几类 object 就是 blob、tree、commit，还有 tag。

- blob

即文件。注意只包含内容，没有名字，权限等属性（但包含大小）。

- tree

相当于文件夹。所包含的文件（blob）/文件夹（tree）的名字及其基本属性（比如权限、是否符号链接等）的列表。

- commit

表示修改历史。commit 对象可以视为类似矢量的概念，由父 commit（合并情形下可能不只一个）指向新的 tree 对象。

  - commit^n

    commit^n 指向该 commit 的第 n 个父 commit（合并情形下，commit 的父 commit 会不只一个。如果未指定 n，默认为第一个）。

  - commit~n

    commit~n 指向第 n 代祖先 commit，且每代都是指向第一个。

    如，rev~3 = rev^^^ = rev^1^1^1

commit 中包含以下信息：

  - tree　　文件夹的 SHA-1
  - parent(s)　父 commit 的 SHA-1
  - author　　作者
  - committer　提交人
  - Comment　　记录

如：

```
commit 635d18fb1b2d6c5ba1360af49da314d0dd3621bf
tree 0954a7830712592ea64329932a295ff32fc15f05
parent e8aba732f02004aa968f2d53cfda52f11192715a
author Li Zhi <lizhi@lizhi.(none)> 1249440029 +0800
committer Li Zhi <lizhi@lizhi.(none)> 1249440029 +0800

4th version

$ git ls-tree 0954a7830712592ea64329932a295ff32fc15f05
040000 tree 148922302de3ea7c41efb9597f6d64f36a30eb37    dir
$ git ls-tree 148922302de3ea7c41efb9597f6d64f36a30eb37
100644 blob 190a18037c64c43e6b11489df4bf0b9eb6d2c9bf    1.txt
100644 blob 190a18037c64c43e6b11489df4bf0b9eb6d2c9bf    2.txt
$ git show 190a18037c64c43e6b11489df4bf0b9eb6d2c9bf
this is a test file
```

- tag

  可以指向 blob、tree、commit 并包含签名，最常见的是指向 commit 的 PGP 签名的标签。

  tag 的主要作用是给某次 commit 起一个好记的名字，如：

  `git tag V3 635d18fb1b2d6c5ba1360af49da314d0dd3621bf`

  以后就可以用 V3 来代替复杂的 SHA-1 值 635d18fb1b2d6c5ba1360af49da314d0dd3621bf

  `git show V3` 即等同于 `git show 635d18fb1b2d6c5ba1360af49da314d0dd3621bf`

从本质上来讲，这几种 objects 并无二致，只是定义了不同的格式，用于不同的目的。blob、tree、commit 都是用其存储内容的 SHA-1 值命名的（不是简单的对整个文件取 SHA-1 值），tag 使用的是普通名字。

1.2.3 directory & index

CVS/SVN 中，代码库和工作拷贝都是分开的，工作拷贝的版本信息放在每一层目录中，这样不便于在工作拷贝中 grep 代码。git 将代码库和工作拷贝的版本信息统一放在工作拷贝的顶层目录下的.git/中（可以配置使用其它目录）。

```
.git/
    ├──branches
    ├──hooks
    ├──info
```

```
├──objects
│   ├──info
│   └──pack
├──refs
│   ├──heads
│   └──tags
└──remotes
```

objects 下面是存放 blob、tree、commit 对象的地方，取 SHA-1 的十六进制前两个字符做目录名，余下的做文件名。注意，一个对象的 SHA-1 值不是直接对这里面的文件计算 SHA-1 值得到的。git 使用 SHA-1 算法，通过对文件的内容或目录的结构计算数据的校验和，得到一个 SHA-1 值。

.git 下可以存在 config 文件，这个是一个库的配置文件，用户可以在自己的 HOME 目录下有.gitconfig 文件，两者格式是一样的，参考 git-repo-config(1)。

.git 下可能还存在一个 logs 目录，由于 git 的 fast forward 形式的合并做法，导致不能通过 git log 来获取一个分支的头曾经指向哪些 commit，这个 logs 目录下的文件就是用来记录这个信息的，要想使用它需要在.git/config 或者~/.gitconfig 里设置选项。这些 log 就称为 reflog，叫 ref 是因为分支名和标签都是 commit 的引用，都放在.git/refs 目录下。

.git 下还有 index 文件（对于刚创建的空库是没有的），这个文件就起到 CVS 的.cvs 目录或者 SVN 的.svn 目录的作用，它记录了工作拷贝在哪个 commit 上，以及工作拷贝中被 git 管理的文件的状态。在 SVN 中，.svn 里保存了一份代码，称为 BASE，index 里跟它作用类似，不过它只是记录了文件名和其 SHA-1 值、mtime、ctime、所在的设备号、权限位、uid、gid、size，真正的文件内容都在.git/objects 里。

index 跟 tree 很类似，除了包含一些信息用于跟踪工作拷贝中的文件状态，粗略的讲，使用 git 最常打交道的有三个 tree：HEAD 对应的 tree、index 对应的 tree、工作拷贝中的目录树。

git 和 svn 对于处理 index 和 BASE 版本的方式不大一样，SVN 是以工作拷贝为中心，BASE 在提交前是不会变的，git 则以 index 为中心，在提交前 index 是可以变的。例如，你在工作拷贝中修改了文件，用 git commit 提交时却告诉你没有修改，这是因为 git commit 是把 index 代表的状态提交到库里，只有你用 git-update-index your_file 将工作目录中的修改反映到 index 里，git commit 才会提交修改，不过文件被提交到库里的时机是在调用 git-update-index 时候，而非 git commit 的时候，git commit 只是依照 index 的状态创建一个 tree 放入.git/ojbects 里。也可以用 git commit -a，详见 git commit [-a]。

另外，index 也被称为 cache，这是早期的叫法。

1.2.4 tree-ish

我们可以使用多种方式来引用 tree。

- SHA-1 值的前几位

  用 SHA-1 引用对象时，一般取其前六个字符，只要能够唯一识别一个对象即可。

- branch、remote 或 tag 名称
- 指定日期，如 master@{yesterday}
- 指定版本号，如 master@{5}
- 指定为第几个父亲，如 master^2
- 指定为第几代的祖先，如 master~2
- 指定某个文件夹，如 master^{tree}
- 指定某个文件，如 master:/path/to/file
- 指定范围，如 7b593b5..51bea1

2 GIT 库结构

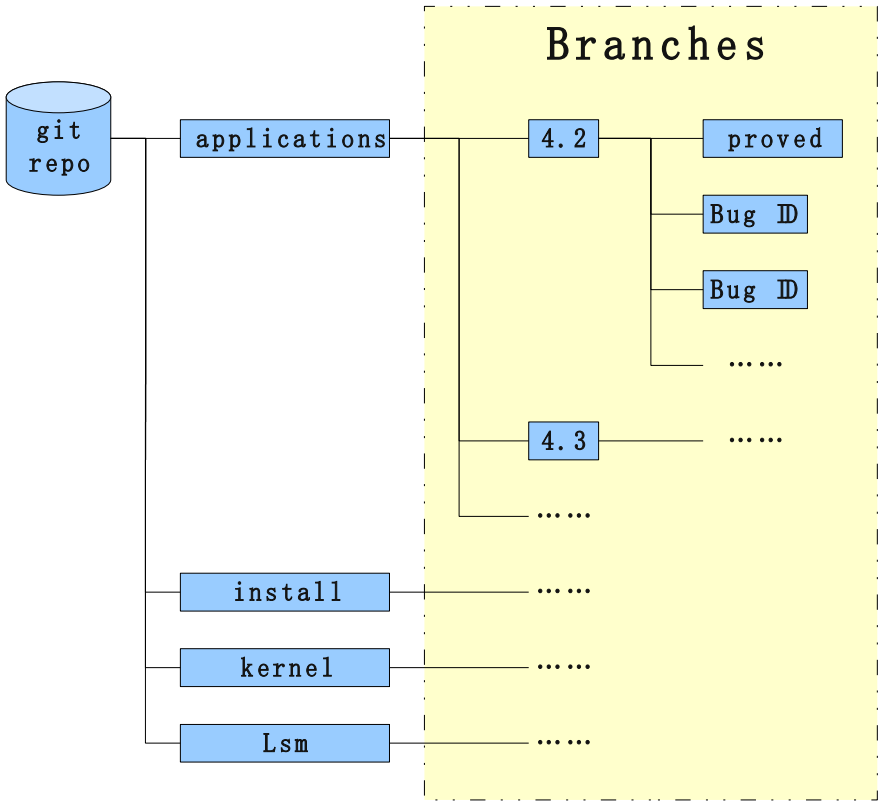

插图 2.1: 研发部 git 库结构

研发部 git 版本库结构如上图所示。

代码主要分为 4 个部分：

- applications　　所有应用程序软件包
- Install　　安装程序
- kernel　　内核
- lsm　　安全功能模块

Branches 分为 proved 和 bug：

- Bug ID

  代码所对应的 Bugzilla 上的 bug 号，用于存放对应 bug 或 feature 修改的代码。

- proved

  存放已通过评审的代码。

开发人员只能向 git 库对应的 Bug ID（包括 bug 和 feature）中提交代码，当 Bug ID 中的代码通过评审后，管理员就可以将其提交到 proved 中。集成人员从 proved 中取出代码进行编译、出盘。

完整的 branch 如：4.2/555

git 的使用者分为开发人员和管理员。开发人员主要是下载、修改、提交源码，管理员对 git 库及使用者进行管理。

3 工作流程

这一部分用来介绍开发人员要如何使用 git。

3.1 身份鉴别

把自己介绍给 git 系统，如，使用者姓名、email 地址。你可以使用全局设置，或者仅仅设置你已经克隆到本地的某个 git 目录。

- 全局设置：

```
git config --global user.name [yourname]
git config --global user.email [yourmail@linx-info.com]
```

- 目录设置：

```
cd <dir-to-your-repo>
git config user.name [yourname]
git config user.email [yourmail@linx-info.com]
```

注：

[yourname]为你的 git 用户名

[yourmail@linx-info.com]为你的公司邮箱地址

<dir-to-your-repo>为你克隆到本地的 git 库目录

3.2 克隆 git 库

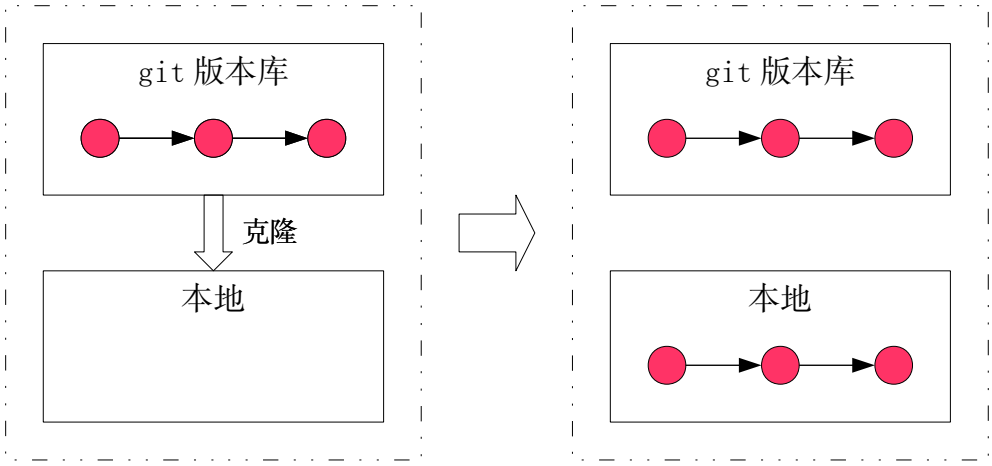

插图 3.1：克隆 git 库

你可以执行下面的命令将 git 库从服务器克隆到本地。

```
git clone [yourname]@[git-host]:[git-repo]
```

注：

[git-host]为 git 库所在服务器地址，目前为 rd-server（172.16.0.4）。

[git-repo]为你要克隆的 git 库目录，为：

- /home/git/applications.git
- /home/git/install.git
- /home/git/kernel.git
- /home/git/lsm.git

开发人员只要克隆自己工作所需的那个目录即可，如果涉及不止一个目录，则需要分别克隆。

然后，执行以下命令创建自己的工作分支。

```
git branch [branch-name] origin/[branch-name]
```

注：

[branch-name]为本地要创建的分支名称。

origin/[branch-name]为对应的服务器 git 库中的分支名称，请参考上一章。

例如：覃波要从 git 库克隆 applications 库并创建本地的某个工作分支

```
git clone bqin@rd-server:/home/git/applicaitons.git
cd applications
git branch 4.2/555 origin/4.2/555
```

如果是 git 服务器上还没有该 bug 的 branch，开发人员则需要自己创建一个新的 branch。如：

```
git branch 4.2/proved  origin/4.2/proved
git branch 4.2/545  4.2/proved
```

先根据 git 服务器的 proved 创建本地的 proved，再从本地 proved 创建对应某个 bug 的 branch。

3.3 本地修改

本地操作的 git 命令很多，详见 git 命令的使用。

基本操作如下：

1. 更新本地 git 库

执行命令 `git pull` 或 `git fectch; git merge` 将其它 git 版本库（如，git 服务器）上的代码修改合并到本地。

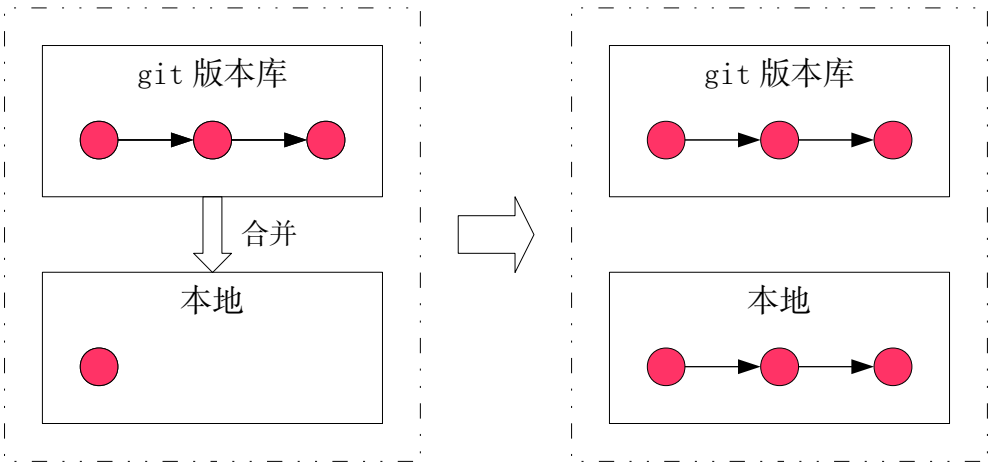

插图 3.2：合并（本地无修改）

或

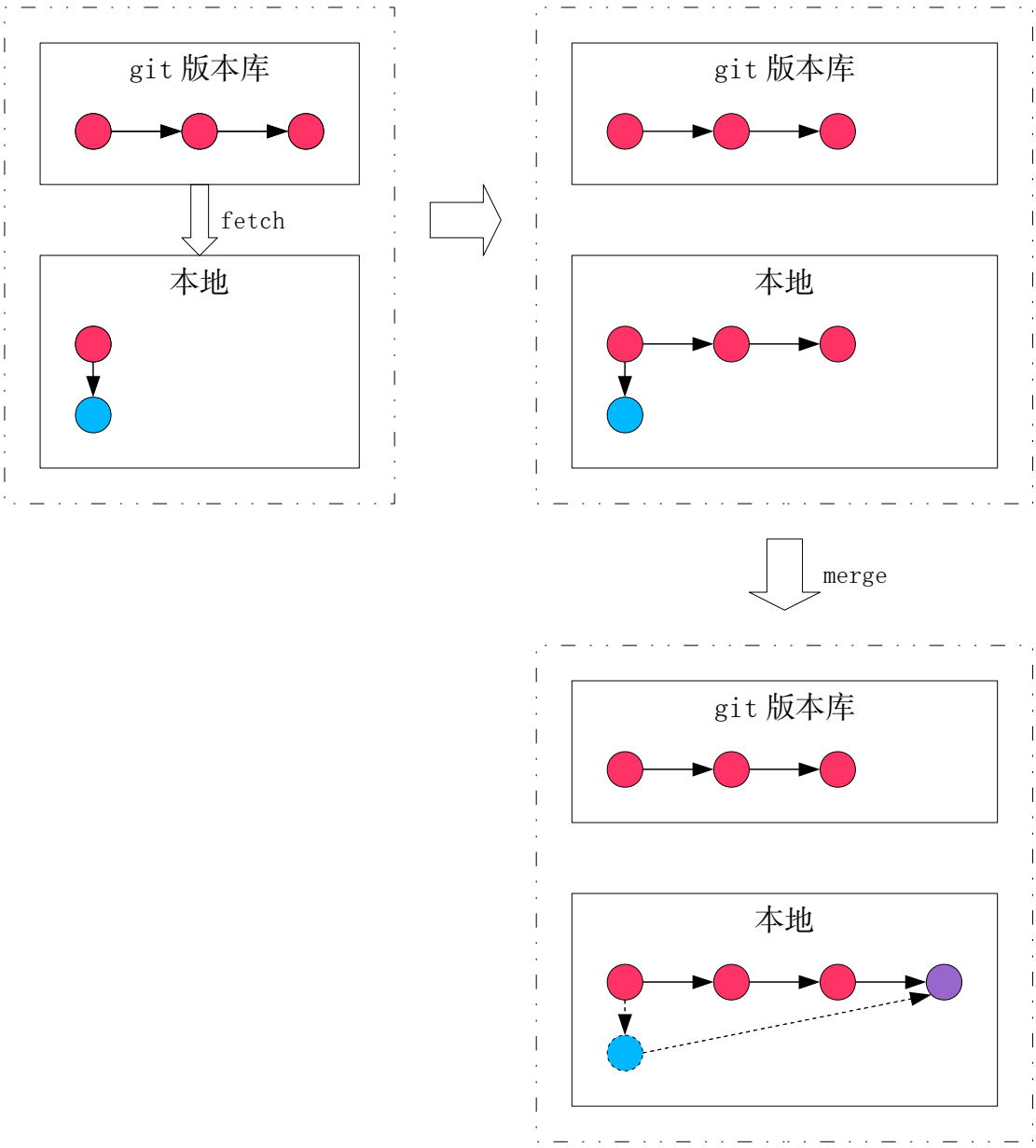

插图 3.3：合并（git 库有更新，且本地有未提交到 git 库的修改）

2. 修改代码文件
3. 提交修改的代码到本地 git 库

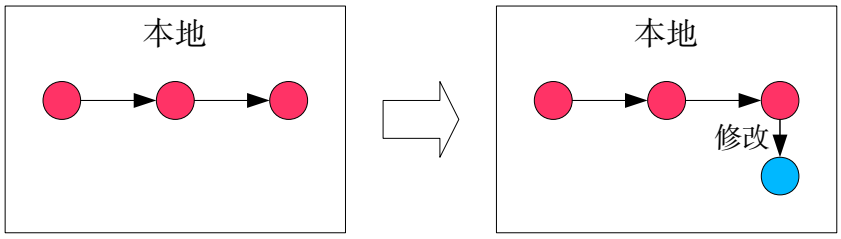

插图 3.4：本地提交

```
git add; git commit
```

或

```
git commit -a
```

注：

“-a”选项表示不需要将修改和删除的文件通过“git add”命令来加入索引。

提交时会打开系统的默认编辑器，你需要编辑一些提交说明，包括 bug ID 和简要的修改说明。bug ID 写在第一行，如“bug XXX”，“XXX”即该修改对应的 bugzilla 上的 bug 的 id。如果要提交的源码涉及多个 bug，可以写成“bug XXX XXX XXX……”或“bug XXX, XXX, XXX, ……”。

3.4 提交修改到服务器

执行 git push 命令将修改后的代码从本地 git 库提交到 git 服务器。

如，执行 git push origin 4.2/555:4.2/555，将 bug 555 的修改提交到服务器。其中，“:”前为本地的分支名，“:”后的为 git 服务器中的分支名。如果 git 服务器上原来没有该 branch，这同时也将在 git 服务器上创建这个 branch。

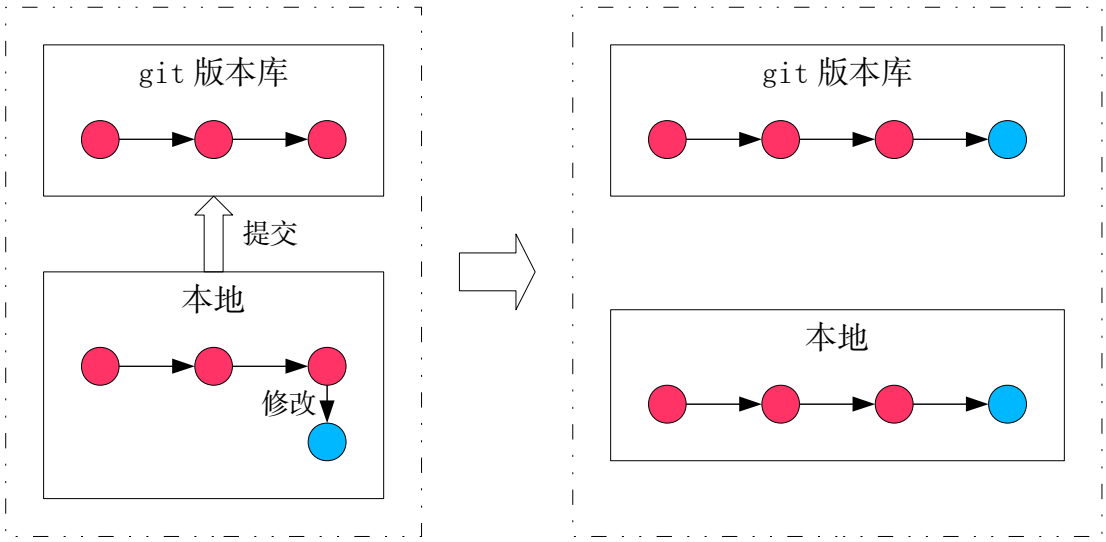

插图 3.5：从本地提交到远程 git 库

如果提交失败，并且错误信息为“non fast forward”，即本地的修改基础不是最新的 git 版本库，git 库在你 pull 和 push 的过程中又有过更新，这时需要执行 git pull 或 git fetch; git merge 将最新的 git 库合并到本地，然后再提交。

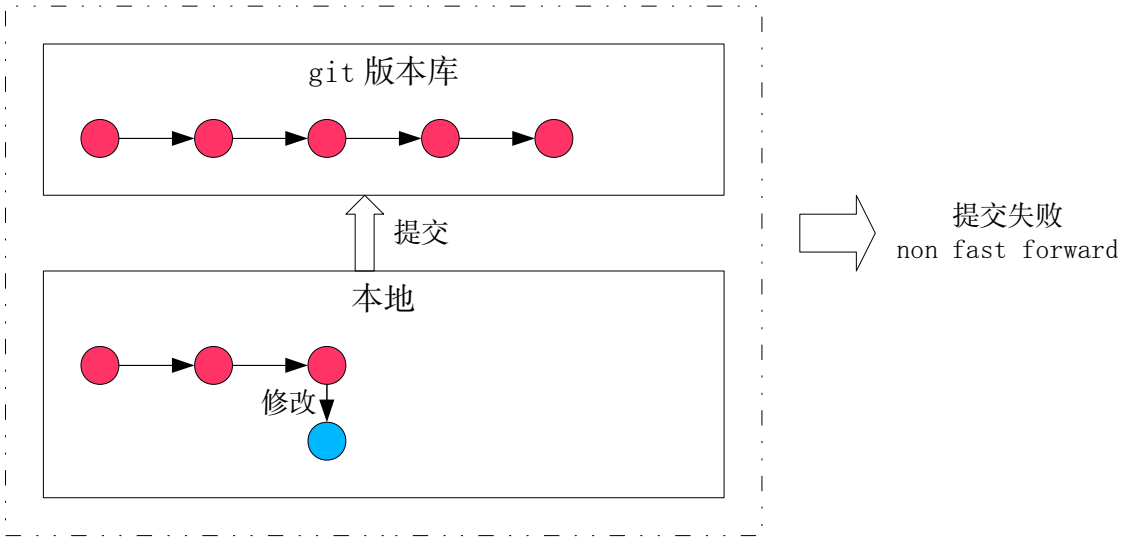

插图 3.6：提交失败

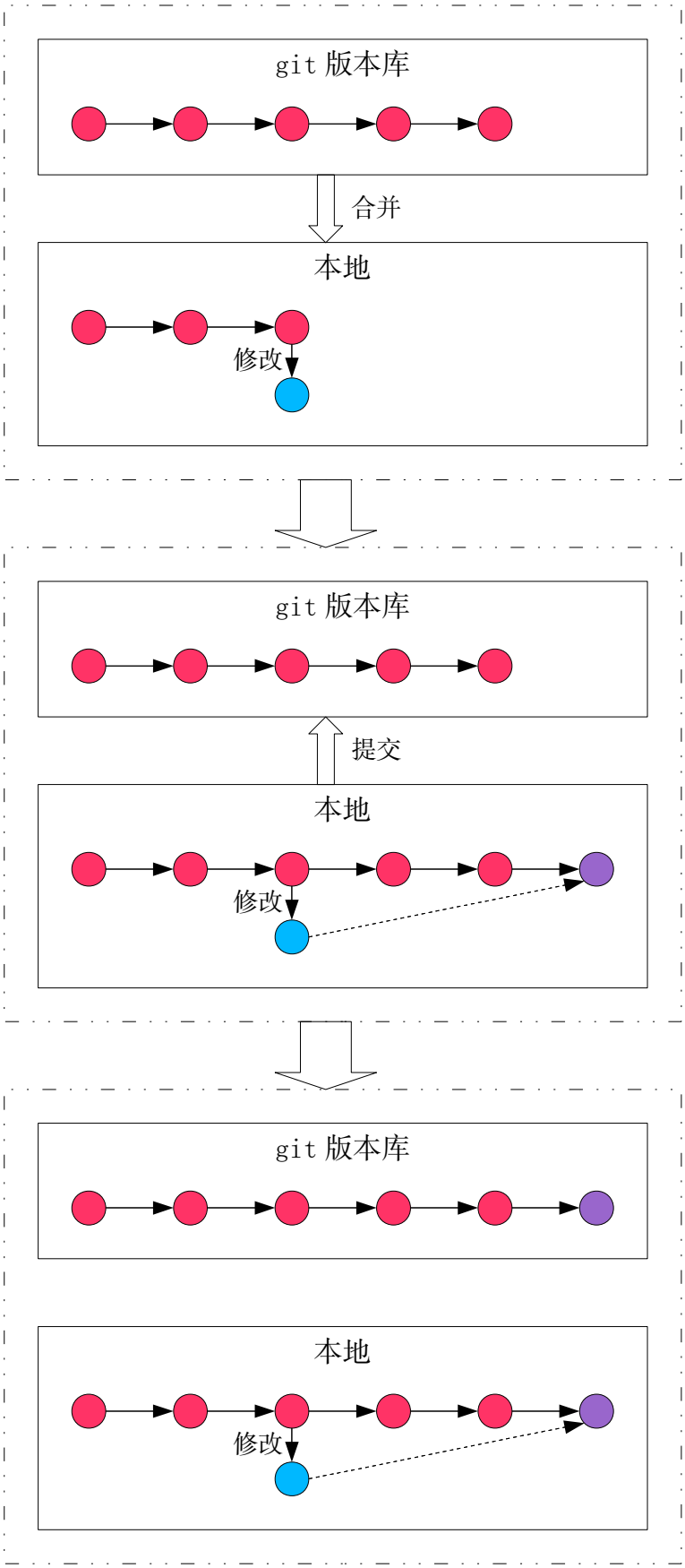

插图 3.7：合并后提交

3.5 提交 patch 到 bugzilla

将代码上传到 git 服务器后，还需要向 bugzilla 上相应的 bug 提交一个 attachment。

由于 git2bugzilla 工具要在 git 服务器系统上使用，需要先使用 git-tools 帐户远程登录到服务器上，再执行命令：

git2bugzilla <git-repo> <start-commit> <end-commit> <bugzilla-username> <bugzilla-userpassword> <bug-ID>

或执行命令

ssh git-tools@rd-server '/home/git-tools/bin/git2bugzilla <git-repo> <start-commit> <end-commit> <bugzilla-username> <bugzilla-userpassword> <bug-ID>'

再输入 git-tools 的口令即可。

其中：

- <git-repo> git 库的路径
- <start-commit> 修改代码的起始点
- <end-commit> 修改代码的结束点
- <bugzilla-username> 在 bugzilla 上的用户名
- <bugzilla-userpassword> 在 bugzilla 上的口令
- <bug-ID> 该 patch 对应的 bug 号

如，覃波提交 bug545 的 patch 到 bugzilla：

ssh git-tools@rd-server '/home/git-tools/bin/git2bugzilla /home/git/applications.git 73c10bcf42727ed4bca141c3471900bd2b2b46a246eb362cc7e6d563f93801ddd4649381b8e bqin@linx-info.com rocky 545'

提示：起始点和结束点的 commit 可以使用任意有效的方式来指定，详见 tree-ish。

4 git 命令的使用

```
usage: git [--version] [--exec-path[=GIT_EXEC_PATH]] [-p|--paginate] [--bare] [--git-
dir=GIT_DIR] [--help] COMMAND [ARGS]

    The most commonly used git commands are:
    add           Add file contents to the changeset to be committed next
    apply         Apply a patch on a git index file and a working tree
    archive       Create an archive of files from a named tree
    bisect        Find the change that introduced a bug by binary search
    branch        List, create, or delete branches
    checkout      Checkout and switch to a branch
    cherry-pick   Apply the change introduced by an existing commit
    clone         Clone a repository into a new directory
    commit        Record changes to the repository
    diff          Show changes between commits, commit and working tree, etc
    fetch         Download objects and refs from another repository
    grep          Print lines matching a pattern
    init          Create an empty git repository or reinitialize an existing one
    log           Show commit logs
    merge         Join two or more development histories together
    mv            Move or rename a file, a directory, or a symlink
    prune         Prune all unreachable objects from the object database
    pull          Fetch from and merge with another repository or a local branch
    push          Update remote refs along with associated objects
    rebase        Forward-port local commits to the updated upstream head
    reset         Reset current HEAD to the specified state
    revert        Revert an existing commit
    rm            Remove files from the working tree and from the index
    show          Show various types of objects
    show-branch   Show branches and their commits
    status        Show the working tree status
    tag           Create, list, delete or verify a tag object signed with GPG
```

git 的命令很多，下面只列举一些常用的命令进行说明，并提供一些使用上的小技巧。

4.1 常用命令

4.1.1 git add

将文件加入到版本库文件索引当中。需要注意的是，这样只是刷新了 git 的跟踪信息，

而并没有提交到 git 的内容跟踪范畴之内，还需要执行 git commit 命令来提交。

4.1.2 git commit [-a]

提交文件到 git 的内容跟踪范畴。“-a”选项表示不需要将修改和删除的文件通过“git add”命令来加入索引。

注：与其它版本控制系统不同，git commit 是在本地提交，还需要使用 git push 来上传到 git 服务器。

4.1.3 git diff [--cached]

查看当前工作（索引）和最后提交的代码（最新版本库）的差别。差别将以 patch 方式表示出来。

4.1.4 git status

查看工作树的状态以及 git 库的修改记录。显示提示信息如：版本库中加入了文件，并且 git 提示我们提交这些文件。

4.1.5 git log [-p]

显示提交日志。“-p”选项表示同时以 patch 方式显示每次提交的代码修改。

4.1.6 git reset

将当前的工作目录完全回滚到指定的版本。“完全”的意思是会删除指定版本之后的数据。

如图 4.1 所示，master 上有 A-G 五次提交的版本。

其中 C 的版本号是 bbaf6fb5060b4875b18ff9ff637ce118256d6f20，

执行 git reset bbaf6fb5060b4875b18ff9ff637ce118256d6f20 后，

结果就只剩下了 A-C 三个提交的版本。

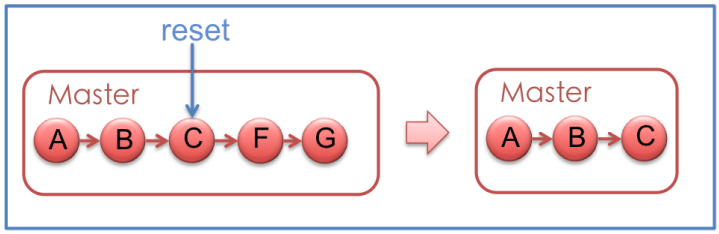

插图 4.1: git reset

4.1.7 git rebase

命令格式：git rebase [--onto <newbase>] <upstream> [<branch>]

3 个参数同时使用时，会把从 upstream 到 branch 的改动移动到 newbase 上去。

- <newbase>指定要移动到的 commit 点；
- <upstream>指定要移动的 commit 点；
- <branch>指定要进行 rebase 操作的分支，未指定时默认为当前分支；

未使用[--onto <newbase>]参数时，<upstream>指定要移动到的 commit 点。

如图 4.2 所示，master 上有 A-G 五次提交的版本，new_branch 是 master 的 C 上的分支。

在 new_branch 下执行 git rebase master 或 git rebase master new_branch

结果 new_branch 的分支点移动到 master 的最后的版本 G 了。

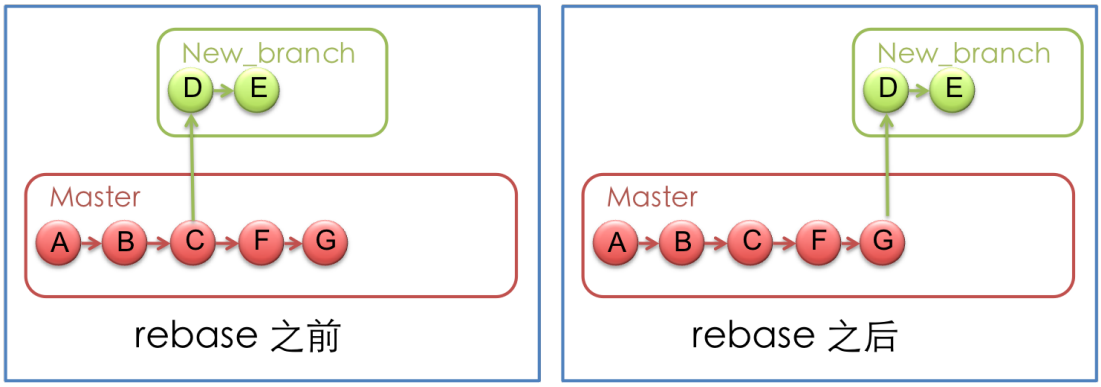

插图 4.2: git rebase

注：<branch>、<newbase>、<upstream>可以使用任意有效的方式来指定，详见 tree-ish。

4.1.8 git revert

还原一个版本的修改，必须提供一个具体的 git 版本号。

如，git revert bbaf6fb5060b4875b18ff9ff637ce118256d6f20

git revert 与 git reset 的区别：

- git reset 是还原到指定的版本上，这将扔掉指定版本之后的版本。
- git revert 是提交一个新的版本将需要 revert 的版本的内容再反向修改回去，版本会递增，不影响之前提交的内容。

4.1.9 git remote

git remote 可用来创建一个远程仓库的链接。

如，

```
git remote add rd-server git://172.16.0.4/git/applications.git
```

增加一个远程服务器端，地址为 git://172.16.0.4/git/applications.git，设置其名称为 rd-server，以后提交代码的时候只需要使用 rd-server 别名即可。

```
git fetch rd-server
```

从 rd-server 的远程仓库提取代码。

```
git merge git/applications
```

将 rd-server 上的修改合并到本地的主分支（git/applications）里。

4.1.10 git branch

分支管理命令，可以创建、删除分支，以及查看分支信息。

- git-branch [new_branch]

  在当前的工作目录下创建一个新的分支 new_branch。

- git-branch -D [new_branch]

  强制删除版本库中的名为 new_branch 的分支。

- git-branch

  显示分支列表并且标识当前所在分支。

4.1.11 git checkout

git 的 checkout 有两个作用：导出分支内容、切换分支。

- 将当前所在的分支的最新版本内容导出来（如果你在修改后还没有提交的话，那种这种导出将直接覆盖你的修改）。
- 从当前分支切换到指定的分支上。例如 git checkout new_branch 就会切换到 new_branch 的分支上去。

4.1.12 git fetch

从另一个 reposoitory 下载 objects 和 refs。

命令格式：git fetch <options> <repository> <refspec>...

<repository>表示远端的仓库路径。

<refspec>的标准格式应该为<src>:<dst>，<src>表示源的分支，如果<dst>不为空，则表示本地的分支；如果为空，则使用当前分支。

如：git fetch origin 4.2/555:4.2/555

从服务器的工作目录的 4.2/555 分支下载 objects 和 refs 到本地的 4.2/555 分支中。

4.1.13 git merge

将两个或两个以上的分支进行合并。

git merge branchname 用于将 branchname 分支合并到当前分支中。(如果合并发生冲突，需要自己解决冲突)

当 merge 命令自身无法解决冲突的时候，它会将工作树置于一种特殊的状态，并且给用户提供冲突信息，以期用户可以自己解决这些问题。而未发生冲突的代码已经被 git merge 记录在索引里了。如果这个时候使用 git diff，显示出来的只是发生冲突的代码信息。

在解决冲突之前，发生冲突的文件会一直在索引中被标记出来。这个时候，如果使用 git commit 提交的话，git 会提示：filename.txt needs merge。

在发生冲突的时候，如果你使用 git status 命令，那么会显示出发生冲突的具体信息。

4.1.14 git pull

git-pull 的作用就是从一个 repository 取出内容并合并到另一个 repository 中。

命令格式与 git fetch 相同。

git pull 是 git fetch 和 git merge 命令的一个组合。

如： git pull origin 4.2/555

从服务器的工作目录的 4.2/555 分支中取出内容并合并到当前分支中。

4.1.15 git push

将本地内容上传到另一个 reposoitory。

4.2 Tips

4.2.1 ..和...的区别

git log -p HEAD..FETCH_HEAD

显示 FETCH_HEAD 可获得，但 HEAD 不能获得的更改。

```
git log -p HEAD...FETCH_HEAD
```

显示除了 HEAD 和 FETCH_HEADHEAD 都可获得之外的，HEAD 或 FETCH_HEAD 可获得的更改。即，FETCH_HEAD 可获得但 HEAD 不能获得的更改，以及 HEAD 可获得但 FETCH_HEAD 不能获得的更改。

如：1--2--3--4--5

```
            \
             3'
              \
               3"
5..3"      3--3'--3"
5...3"     3--4--5
            \
             3'
              \
               3"
```

当多个开发人员同时修改一个代码文件时，这种查看方式比较有效。使用“..”可以查看自从分叉之后某人又做了什么修改，或使用“...”查看大家各自做了些什么。

4.2.2 查看日志

我们可以使用几个不同的命令来查看日志。

- git log

显示 commit 日志。

- git show

显示更详细的信息，包括了提交时间、修改内容、git diff 信息等等。

- git grep

搜索。与 shell 中的 grep 命令相比，功能更强大，可以指定搜索某一个特定 tree-ish 的文件内容。

4.2.3 ignore files

git 将根据在版本库根目录中的.gitignore 文件中列出的文件名，忽略对那些文件的跟踪，文件和目录名可以用正则表达式来表示。这样，当使用 git add .命令和 git commit

-a 命令时，这些文件就不会被提交到 git 了。

也可以把.ignore 文件放到工作目录下的任意 tree 目录里，然后只需要使用 git add 命令把这个.ignore 文件加到 git 控制下即可。

例如，.gitignore 文件内容如下：

```
*#*
.footprint.diff
.footprint.orig
.md5sum.diff
.md5sum.orig
*.pyc
*.o
*.so.*
```

4.2.4 删除或修改之前的 commit

- 删除

我们可以使用 git rebase 命令来删除 commit。

如：master 的版本跟踪如 E---F---G---H---I---J

执行命令 `git rebase --onto master~5 master~3 master`

master 的版本跟踪将变成 E---H---I---J

这样就删除了 F 和 G 的版本。

当 F 和 G 有缺陷，或不打算将其放在 master 中时，这种方式很有用。

- 修改

例如：master 的版本跟踪如 E---F---G---H，需要修改 F，步骤如下：

1. 新建一个 branch 用于进行修改操作，并从 master 取出 F 及之前的版本到新建的 branch：

   ```
   git checkout -f -b tmp HEAD~2
   ```

   tmp 的版本跟踪为：E---F

2. 在 tmp 上，回退到上一个 commit，然后对文件进行修改再提交

```
git reset HEAD^
```

tmp 的版本跟踪变为：E

```
vim foo.c
```

```
git commit -a -c ORIG_HEAD
```

修改并提交文件。其中 ORIG_HEAD 指向 F，“-c”选项使得新的 commit 使用 F 的修改记录和作者信息（包括时间），并可以进一步编辑提交信息。

tmp 的版本跟踪变为：E---F'

3. 用 tmp 修改结果 F'替换 master 上的 F

```
git rebase --onto tmp master~2 master
```

将 master 上的 F 之后的版本 rebase 到 tmp 的 F'上。

master 的版本跟踪变为：E---F'---G---H

4. 删除 branch tmp

```
git branch -D tmp
```

4.2.5 放弃修改

在未提交到 git 服务器之前，我们可以使用以下命令来放弃工作目录中的修改。

- git checkout

  放弃当前修改。

  对特定文件使用不带其它参数的 git checkout 命令可以将文件恢复到 index 中的状态，如果你想恢复的特定的版本，可是使用类似：git checkout HEAD file 这样的操作，将文件恢复到 HEAD tree 即最近一次提交的状态。

- git reset

  将某分支的版本恢复到以前的某个状态。

- git revert

  放弃一个修改。

  即提交一个新的版本将需要 revert 的版本的内容再反向修改回去，版本会递增，不影响之前提交的内容。

- git commit --amend

  修改最后一次 commit 的内容而不会递增版本。

4.3 git 图形工具

我们还可以使用 gitk 等图形化的 git 工具来管理 git 库。

gitk 是一个查看主干/分支情况的工具，它主要用于观察整个项目的分支状况，使用 gitk 命令就会出现一个图形化界面供你查看。

另外，还可以使用 git gui 命令打开一个 git 图形化管理界面。

5 GIT 库的管理

这一部分用来介绍管理员要如何管理 git。

5.1 添加组和用户

- 管理员可执行以下命令来添加一个 git 组：

  ```
  groupadd git-developers
  ```

- 再使用以下命令将开发人员用户添加到该 git 组：

  ```
  useradd -m -g git-developers [developer-name]
  ```

- 编辑/etc/passwd，将开发人员用户的默认 shell 改为/usr/bin/git-shell

  如：

  ```
  bqin:x:2003:1000::/home/bqin:/usr/bin/git-shell
  hbzhao:x:2004:1000::/home/hbzhao:/usr/bin/git-shell
  xzliu:x:2005:1000::/home/xzliu:/usr/bin/git-shell
  jyang:x:2006:1000::/home/jyang:/usr/bin/git-shell
  zhi:x:2007:1000::/home/zhi:/usr/bin/git-shell
  ```

5.2 创建 git 库

从本地克隆文件到 git 库中，如：

```
git clone --bare /mnt/resources/proved/git/secure/install install.git
```

git 库一般分为两部分，一个是.git 仓库，存放源文件以及用于版本控制的索引文件等，另一个是工作树，存放用于修改操作的源码文件。使用“--bare”选项让服务器的 git 版本库中仅创建.git 仓库，而不创建工作树。当开发人员克隆 git 库时，可根据.git 仓库的源文件和修改记录来生成自己的工作树。

5.3 设置 git 库权限

- 创建 $GIT_DIR/hooks/update 用来对开发人员提交源码时的操作进行一些限制：
  - 查看提交源码的开发人员本地的 git 库，本地的修改基础不是最新的 git 版本库的不允许提交，详见 3.4 提交修改到服务器。
  - 根据$GIT_DIR/info/allowed-user 和$GIT_DIR/info/allowed-group 文件中的设

置对 git 使用者的操作进行限制，如：开发人员仅可对 Bug ID 中的文件进行更新，只有管理员可以更新 proved 中的文件。

除此之外，还可以设置：没有说明 bug 号的源码不允许提交。

- 创建 $GIT_DIR/info/allowed-group，设置用户组对 git 库的使用权限，如：

  refs/heads/[[:digit:]]\.[[:digit:]]/.* git-developers

  refs/heads/[[:digit:]]\.[[:digit:]]/proved$ git-management

  git-developers 组可以修改 Bug ID 中的文件，git-management 组可以修改 proved 中的文件，未定义的其他组不能修改。

  注：管理员还可以在 $GIT_DIR/info/allowed-user 文件中设置单个用户对 git 库的使用权限，格式与 allowed-group 文件相同。目前 git 库仅通过设置组权限进行管理。

- 设置用户对 git 库的目录的访问权限，如：

  setfacl -R -m “g:git-developers:rwx” install.git

  setfacl -dR -m “g:git-developers:rwx” install.git

  注：

  “-R”选项表示该目录中的所有子目录和文件都设置一样的 ACL。

  “-d”选项表示之后向该目录中添加子目录和文件也设置一样的 ACL。

5.4 通知开发人员

经过上述步骤设置好 git 库后，将信息通知相关开发人员。