

C 博客

登录 | 注册

Q

≡


sunweizhong1024的专栏

目录视图

摘要视图

RSS 订阅

个人资料



sunweizhong1024

+ 加关注

发私信

访问：224709次

积分：2053

等级：BLOG 5

排名：第15386名

原创文章：18篇

转载：26篇

译文：0篇

评论：45条

文章搜索

文章分类

ALSA (6)

Input subsystem (0)

windows\_driver (3)

UEfi (1)

Android Sensor (2)

MTK\_Driver (4)

Display 相关驱动 (2)

android\_dispaly (4)

文章存档

2015年03月 (1)

2014年09月 (1)

2014年05月 (1)

2014年03月 (9)

2013年12月 (4)

展开

阅读排行

GIT 的使用方法详解

(46669)

repo使用

(24171)

repo用法详解

(21141)

LCD 驱动过程详解

(17972)

Uevent 上报event事件给

转 GIT 的使用方法详解

标签：git branch 服务器 工作 merge 文档

2012-10-10 10:53 46669人阅读 评论(2) 收藏 举报

快速回复

我要收藏

本文转载于：<http://blog.csdn.net/gemmem/article/details/7290125>

1. Git概念

1.1. Git库中由三部分组成

Git 仓库就是那个.git 目录，其中存放的是我们所提交的文档索引内容，Git 可基于文档索引内容对其所管理的文档进行内容追踪，从而实现文档的版本控制。.git目录位于工作目录下。

1) 工作目录：用户本地的目录；

2) Index（索引）：将工作目录下所有文件（包含子目录）生成快照，存放到一个临时的存储区域，Git 称该区域为索引。

3) 仓库：将索引通过commit命令提交至仓库中，每一次提交都意味着版本在进行一次更新。




图 1.1 Git 仓库建立流程

51CTO.com 技术博客 Blog

1.2. 使用Git时的初始化事项

1.2.1. Git初始化配置

1) 配置使用git仓库的人员姓名

git config --global user.name "Your Name Comes Here"

2) 配置使用git仓库的人员email

git config --global user.email you@yourdomain.example.com

1.2.2. Git文档忽略机制

工作目录中有一些文件是不希望接受Git 管理的，譬如程序编译时生成的中间文件等等。Git 提供了文档忽略机制，可以将工作目录中不希望接受Git 管理的文档信息写到同一目录下的.gitignore 文件中。

例如：工作目录下有个zh目录，如果不想把它加入到Git管理中，则执行：

echo "zh" &gt; .gitignore

git add .

有关gitignore 文件的诸多细节知识可阅读其使用手册：man gitignore

1.3. Git与Repo的比较

Git操作一般对应一个仓库，而Repo操作一般对应一个项目，即一个项目会由若干仓库组成。

例如，在操作整个Recket项目时使用Repo，而操作其中的某个仓库时使用Git。在包含隐藏目录.git的目录下执行git 操作。

2. Git help

Git help 获取git基本命令

MTK camera image sensor (13483)

MTK Ft6306 touch 驱动 (6604)

ALsa Control 从上层到驱动 (6419)

Power Management (6099)

Android sensor hal 详解 (5524)

评论排行

MTK camera image sensor (18)

LCD 驱动过程详解 (12)

Android sensor hal 详解 (3)

Mtk Ft6306 touch 驱动 (3)

GIT 的使用方法详解 (2)

MTK\_HDMI 驱动 (2)

input 子系统的详解讲解3 (2)

Power Management (2)

ALsa Control 从上层到驱动 (1)

android.mk 的使用 (0)

推荐文章

\* 而立之年——三线城市程序员的年终告白

\* Java集合框架中隐藏的设计套路

\* Python脚本下载今日头条视频(附加Android版本辅助下载器)

\* 人工智能的冷思考

\* React Native 实战系列教程之热更新原理分析与实现

最新评论

Mtk Ft6306 touch 驱动  
公司软测的女生好凶啊:要是加上硬件图就更好了

MTK camera image sensor driver  
Eliot\_shao: 请问image sensor 很多文件以kd开头, 是为什么? kd是什么简称呢?

MTK\_HDMI 驱动  
zenghaiqi: 问问:MTK的MHL驱动框架是和HDMI公用的吗?

MTK\_HDMI 驱动  
攻城大狮: 博主, 你好, 请问MTK的MHL驱动框架是和HDMI公用的吗?

LCD 驱动过程详解  
JQ\_AK47: 楼主说做什么都一样, 都是linux让我感觉找到了不变的东西

LCD 驱动过程详解  
ly8713: @bboyiaoye:你研究明白了吗, 我也很纠结这个问题呢, 我是三星平台的

ALsa Control 从上层到驱动的详解  
Aderic: 我正在研究control,很好的一篇博文!

LCD 驱动过程详解  
DBOY: 分析的不错

LCD 驱动过程详解  
阿晔sir: 楼主, 请教一下, dsi\_set\_cmdq参数里的data\_array和data\_array怎么得...

Mtk Ft6306 touch 驱动  
yubing1818: 能把FT6306的驱动发我一份吗? 谢谢

（如果要知道某个特定命令的使用方法，例如：使用Git help clone，来获取git clone的使用方法）

3. Git本地操作基本命令

3.1. Git init

或者使用git init-db。

创建一个空的Git库。在当前目录中产生一个.git 的子目录。以后，所有的文件变化信息都会保存到这个目录下，而不像CVS那样，会在每个目录和子目录下都创建一个CVS目录。

在.git目录下有一个config文件，可以修改其中的配置信息。

3.2. Git add

将当前工作目录中更改或者新增的文件加入到Git的索引中，加入到Git的索引中就表示记入了版本历史中，这也是提交之前所需要执行的一步。

可以递归添加，即如果后面跟的是一个目录作为参数，则会递归添加整个目录中的所有子目录和文件。例如：

```
git add dir1 （添加dir1这个目录，目录下的所有文件都被加入）
Git add f1 f2 （添加f1，f2文件）
git add . （添加当前目录下的所有文件和子目录）
```

3.3. Git rm

从当前的工作目录中和索引中删除文件。

可以递归删除，即如果后面跟的是一个目录做为参数，则会递归删除整个目录中的所有子目录和文件。例如：

```
git rm -r * （进入某个目录中，执行此语句，会删除该目录下的所有文件和子目录）
git rm f1 （删除文件f1，包含本地目录和index中的此文件记录）
git rm --ached f1 (删除文件f1，不会删除本地目录文件，只删除index中的文件记录；将已经git add的文件remove到cache中,这样commit的时候不会提交这个文件, 适用于一下子添加了很多文件, 却又想排除其中个别几个文件的情况.)
```

3.4. Git commit

提交当前工作目录的修改内容。

直接调用git commit命令，会提示填写注释。通过如下方式在命令行就填写提交注释：git commit -m "Initial commit of gittutor reposistory"。注意，和CVS不同，git的提交注释必须不能为空，否则就会提交失败。

git commit还有一个 -a的参数，可以将那些没有通过git add标识的变化一并强行提交，但是不建议使用这种方式。

每一次提交，git就会为全局代码建立一个唯一的commit标识代码，用户可以通过git reset命令恢复到任意一次提交时的代码。

```
git commit --amend -m "message" （在一个commit id上不断修改提交的内容）
```

3.5. Git status

查看版本库的状态。可以得知哪些文件发生了变化，哪些文件还没有添加到git库中等等。建议每次commit前都要通过该命令确认库状态。

最常见的误操作是，修改了一个文件，没有调用git add通知git库该文件已经发生了变化就直接调用commit操作，从而导致该文件并没有真正的提交。这时如果开发者以为已经提交了该文件，就继续修改甚至删除这个文件，那么修改的内容就没有通过版本管理起来。如果每次在提交前，使用git status查看一下，就可以发现这种错误。因此，如果调用了git status命令，一定要格外注意那些提示为“Changed but not updated:”的文件。这些文件都是与上次commit相比发生了变化，但是却没有通过git add标识的文件。

3.6. Git log

查看历史日志，包含每次的版本变化。每次版本变化对应一个commit id。

```
Git log -1
-1的意思是只显示一个commit，如果想显示5个，就-5。不指定的话，git log会从该commit一直往后显示。
Git log --stat --summary （显示每次版本的详细变化）
```

在项目日志信息中，每条日志的首行（就是那一串字符）为版本更新提交所进行的命名，我们可以将该命名理解为项目版本号。项目版本号应该是唯一的，默认由Git自动生成，用以标示项目的某一次更新。如果我们将项目版本号用作git-show 命令的参数，即可查看该次项目版本的更新细节。例如：

1) Git log

```
commit dfb02e6e4f2f7b573337763e5c0013802e392818
Author: Li Yanrui <LiYanrui.m2@gmail.com>
Date: Wed Jul 9 16:32:25 2008 +0800
```

Git 使用指南文档项目初始化

```
commit 9a4a9ce37561bbb42d8187d7a851e228e26e1212
Author: Li Yanrui <LiYanrui.m2@gmail.com>
Date: Wed Jul 9 16:31:07 2008 +0800
```

添加 .gitignore 文件

```
commit 459640624390eb733fb2ad45bcb8731435931e60
Author: Li Yanrui <LiYanrui.m2@gmail.com>
Date: Wed Jul 9 16:28:50 2008 +0800
```

M2Doc 项目初始化  
lines 1-17/17 (END)

51CTO.com  
技术博客 Blog

## 2) Git show

```
$ git show dfb02e6e4f2f7b573337763e5c0013802e392818
```

除了使用完整的版本号查看项目版本更新细节之外, 还可以使用以下方式:

```
$ git show dfb02 # 一般只使用版本号的前几个字符即可
$ git show HEAD # 显示当前分支的最新版本的更新细节
```

每一个项目版本号通常都对应存在一个父版本号, 也就是项目的前一次版本状态。可使用如下命令查看当前项目版本的父版本更新细节:

```
$ git show HEAD~ # 查看 HEAD 的父版本更新细节
$ git show HEAD^^ # 查看 HEAD 的祖父版本更新细节
$ git show HEAD-4 # 查看 HEAD 的祖父之祖父的版本更新细节
```

你可以对项目版本号进行自定义, 然后就可以使用自定义的版本号查看对应的项目版本更新细节:

```
$ git tag v0.1 dfb02
$ git show
```

51CTO.com  
技术博客 Blog

实际上, 上述命令并非是真的进行版本号自定义, 只是制造了一个tag对象而已, 这在项目进行版本对外发布时比较有用。

## 3.7. Git merge

把服务器上下载下来的代码和本地代码合并。或者进行分支合并。

例如: 当前在master分支上, 若想将分支dev上的合并到master上, 则git merge dev

注意: git merge nov/eclair\_eocket (是将服务器git库的eclair\_eocket分支合并到本地分支上)

git rebase nov/eclair\_eocket (是将服务器git库的eclair\_eocket分支映射到本地的一个临时分支上, 然后将本地分支上的变化合并到这个临时分支, 然后再用这个临时分支初始化本地分支)

## 3.8. Git diff

把本地的代码和index中的代码进行比较, 或者是把index中的代码和本地仓库中的代码进行比较。

### 1) Git diff

比较工作目录和Index中的代码。

### 2) Git diff - - cached

比较index和本地仓库中的代码。

## 3.9. Git checkout

### 3.9.1. 切换到分支

#### 1) 创建一个新分支, 并切换到该分支上

Git checkout -b 新分支名

#### 2) 切换到某个已经建立的本地分支local\_branch

Git checkout local\_branch

(使用cat .git/HEAD后, 显示refs:refs/heads/ local\_branch)

#### 3) 切换到服务器上的某个分支remote\_branch

Git checkout remote\_branch

(远程分支remote\_branch可以通过 git branch -r 列出)

#### 4) 切换到某个commit id

Git checkout commit\_id

（使用`cat .git/HEAD`后，显示`commit_id`）

#### 5) 切换到某个tag

Git checkout tag

（使用`cat .git/HEAD`后，显示tag）

注意：除了1）和2）外，其余三种都只是切换到了一个临时的( no branch )状态（this head is detached），这时用 `git branch` 可以看到处于（no branch）上，`cat .git/HEAD` 看到指向相应的commit id。这个（no branch）只是临时存在的，并不是一个真正建立的branch。如果此时执行2），则这个（no branch）就自动消失了；如果执行1），则创建新分支 new branch，并把这个(no branch)挂到这个新分支上，此时`cat .git/refs/heads/new_branch` 可以看到已经指向了刚才那个commit id。

#### 3.9.2. 用已有分支初始化新分支

执行下面的命令，在切换到某个已经建立的local branch或者某个remote branch或者某个commit id 或者某个tag的同时，创建新分支new\_branch，并且挂到这个新分支上。

1）切换到某个已经建立的本地分支local\_branch，并且使用此分支初始化一个新分支new\_branch。

git checkout -b new\_branch local\_branch

2) 切换到某个远程分支remote\_branch，并且用此分支初始化一个新分支new\_branch。

Git checkout -b new\_branch remote\_branch

3) 切换到某个commit id，并建立新分支new\_branch

Git checkout -b new\_branch commit\_id

4) 切换到某个tag，并建立新分支new\_branch

Git checkout -b new\_branch tag

#### 3.9.3. 还原代码

例如“`git checkout app/model/user.rb`”就会将user.rb文件从上一个已提交的版本中更新回来，未提交的工作目录中的内容全部会被覆盖。

### 3.10. Git-ls-files

查看当前的git库中有那些文件。

### 3.11. Git mv

重命名一个文件、目录或者链接。

例如：`Git mv helloworld.c helloworld1.c`（把文件helloworld.c 重命名为 helloworld1.c）

### 3.12. Git branch

#### 3.12.1. 总述

在 git 版本库中创建分支的成本几乎为零，所以，不必吝啬多创建几个分支。当第一次执行`git init`时，系统就会创建一个名为“master”的分支。而其它分支则通过手工创建。

下面列举一些常见的分支策略：

创建一个属于自己的个人工作分支，以避免对主分支 master 造成太多的干扰，也方便与他人交流协作；

当进行高风险的工作时，创建一个试验性的分支；

合并别人的工作的时候，最好是创建一个临时的分支用来合并，合并完成后再“fetch”到自己的分支。

对分支进行增、删、查等操作。

注意：分支信息一般在`.git/refs/`目录下，其中heads目录下为本地分支，remotes为对应服务器上的分支，tags为标签。

#### 3.12.2. 查看分支

git branch 列出本地git库中的所有分支。在列出的分支中，若分支名前有\*，则表示此分支为当前分支。

git branch -r 列出服务器git库的所有分支。

（可以继续使用命令“`git checkout -b 本地分支名 服务器分支名`”来获取服务器上某个分支的代码文件）。

#### 3.12.3. 查看当前在哪个分支上

cat .git/HEAD

#### 3.12.4. 创建一个分支

##### 1) git branch 分支名

虽然创建了分支，但是不会将当前工作分支切换到新创建的分支上，因此，还需要命令“`git checkout 分支名`”来切换，

##### 2) git checkout -b 分支名

不但创建了分支，还将当前工作分支切换到了该分支上。

#### 3.12.5. 切换到某个分支：git checkout 分支名

切换到主分支：`git checkout master`

#### 3.12.6. 删除分支

`git branch -D` 分支名

注意：删除后，发生在该分支的所有变化都无法恢复。强制删除此分支。

### 3.12.7. 比较两个分支上的文件的区别

`git diff master` 分支名（比较主分支和另一个分支的区别）

### 3.12.8. 查看分支历史

`git-show-branch`（查看当前分支的提交注释及信息）

`git-show-branch -all`（查看所有分支的提交注释及信息）例如：

```
* [dev] d2
! [master] m2
--
* [dev] d2
* [dev^] d1
* [dev~2] d0
*+ [master] m2
```

在上述例子中，“--”之上的两行表示有两个分支`dev`和`master`，且`dev`分支上最后一次提交的日志是“d2”，`master`分支上最后一次提交的日志是“m2”。“--”之下的几行表示了分支演化的历史，其中 `dev`表示发生在`dev`分支上的最后一次提交，`dev^`表示发生在`dev`分支上的倒数第二次提交。`dev~2`表示发生在`dev`分支上的倒数第三次提交。

### 3.12.9. 查看当前分支的操作记录

`git whatchanged`

### 3.12.10. 合并分支

法一：

`git merge` “注释” 合并的目标分支 合并的来源分支

如果合并有冲突，`git`会有提示。

例如：`git checkout master`（切换到`master`分支）

`git merge HEAD dev~2`（合并`master`分支和`dev~2`分支）或者：`git merge master dev~2`

法二：

`git pull` 合并的目标分支 合并的来源分支

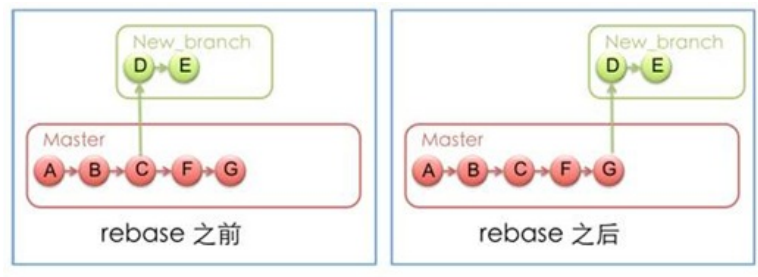
例如：`git checkout master`（切换到`master`分支）

`git pull . dev~2`（合并当前分支和`dev~2`分支）

## 3.13. Git rebase

一般在将服务器最新内容合并到本地时使用，例如：在版本C时从服务器上获取内容到本地，修改了本地内容，此时想把本地修改的内容提交到服务器上；但发现服务器上的版本已经变为G了，此时就需要先执行Git rebase，将服务器上的最新版本合并到本地。例如：

用下面两幅图解释会比较清楚一些，`rebase`命令执行后，实际上是将分支点从C移到了G，这样分支也就具有了从C到G的功能。



## 3.14. Git reset

库的逆转与恢复除了用来进行一些废弃的研发代码的重置外，还有一个重要的作用。比如我们从远程clone了一个代码库，在本地开发后，准备提交回远程。但是本地代码库在开发时，有功能性的commit，也有出于备份目的的commit等等。总之，commit的日志中有大量无用log，我们并不想把这些log在提交回远程时也提交到库中。因此，就要用到git reset。

`git reset`的概念比较复杂。它的命令形式：`git reset [--mixed | --soft | --hard] [<commit-ish>]`

命令的选项：

`--mixed` 这个是默认的选项。如`git reset [--mixed] dev^`(`dev^`的定义可以参见2.6.5)。它的作用仅是重置分支状态到`dev^`，但是却不改变任何工作文件的内容。即，从`dev^`到`dev1`的所有文件变化都保留了，但是`dev1`到`dev1`之间的所有commit日志都被清除了，而且，发生变化的文件内容也没有通过`git add`标识，如果您要重新commit，还需要对变化的文件做一次`git add`。这样，commit后，就得到了一份非常干净的提交记录。（回退了index和仓库中的内容）



--soft相当于做了git reset --mixed，后，又对变化的文件做了git add。如果用了该选项，就可以直接commit了。（回退了仓库中的内容）

--hard这个命令就会导致所有信息的回退，包括文件内容。一般只有在重置废弃代码时，才用它。执行后，文件内容也无法恢复回来了。（回退了工作目录、index和仓库中的内容）

例如：

切换到使用的分支上：

git reset HEAD^ 回退第一个记录

git reset HEAD~2 回退第二个记录

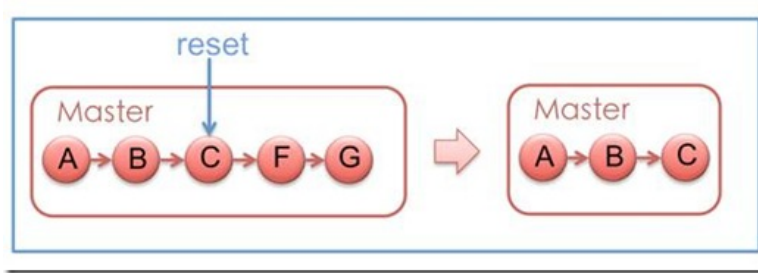
如果想把工作目录下的文件也回退，则使用git reset - - hard HEAD^回退第一个记录

git reset - - hard HEAD~2 回退第二个记录

还可以使用如下方法：

将当前的工作目录完全回滚到指定的版本号，假设如下图，我们有A-G五次提交的版本，其中C的版本号是bbaf6fb5060b4875b18ff9ff637ce118256d6f20，我们执行了'git reset

bbaf6fb5060b4875b18ff9ff637ce118256d6f20'那么结果就只剩下了A-C三个提交的版本



### 3.15. Git revert

还原某次对版本的修改，例如：git revert commit\_id（其中commit\_id为commit代码时生成的一个唯一表示的字符串）

例如：（3.6中）git revert dfb02e6e4f2f7b573337763e5c0013802e392818（执行此操作，则还原上一次commit的操作）

### 3.16. Git config

利用这个命令可以新增、更改Git的各种设置，例如“git config branch.master.remote origin”就将master的远程版本库设置为别名叫做origin版本库。

### 3.17. Git show

显示对象的不同类型。

### 3.18. Git tag

创建、列出、删除或者验证一个标签对象（使用GPG签名的）。

可以将某个具体的版本打上一个标签，这样就不需要记忆复杂的版本号哈希值字符串了，例如你可以使用“git tag revert\_version bbaf6fb5060b4875b18ff9ff637ce118256d6f20”来标记这个被你还原的版本，那么以后你想查看该版本时，就可以使用 revert\_version标签名，而不是哈希值了。

## 4. Git服务器操作命令（与服务器交互）

### 4.1. Git clone

取出服务器的仓库的代码到本地建立的目录中（与服务器交互）

通过git clone获取远端git库后，.git/config中的开发者信息不会被一起clone过来。仍然需要为本地库的.git/config文件添加开发者信息。此外，开发者还需要自己添加 .gitignore文件。

通过git clone获取的远端git库，只包含了远端git库的当前工作分支。如果想获取其它分支信息，需要使用“git branch -r”来查看，如果需要将远程的其它分支代码也获取过来，可以使用命令“git checkout -b 本地分支名 远程分支名”，其中，远程分支名为“git branch -r”所列出的分支名，一般是诸如“origin/分支名”的样子。如果本地分支名已经存在，则不需要“-b”参数。

例如：

Lyr 的 M2GE 工作树为 /work/m2ge, Tzc 可通过以下命令获得与 Lyr 同样的工作树:

```
$ cd work
$ git clone lyr@192.168.0.7:~/work/m2ge m2ge
```

git-clone 可利用各种网络协议访问远端机器中的 Git 仓库,从中导出完整的工作树到本地。在上述示例中,Tzc 通过 SSH 协议访问了 Lyr 机器上的 lyr 账户的 M2GE 仓库并进行导出,从而在当前目录下建立了 m2ge 工作树。若上述命令中未指定本地工作树名,那么 git-clone 会在 Tzc 当前所在目录中建立与 Lyr 的 M2GE 工作树同名的工作树,所以上述命令指定 Tzc 的工作树名为 m2ge 显得有些多余。

注意,git-clone 命令只要碰到类似以下格式的远端仓库地址,它就会认为该地址是符合 SSH 协议的。

账户@IP:工作树路径

一个阶段之后,二人均可将所做的工作不断地提交到各自的 Git 仓库中,直至他们觉得有必要将各自所做的工作合并起来之后再行新的开发阶段。由于 Lyr 作为主要开发者,二人的工作在他的机器上进行合并是比较自然的。当然在 Tzc 机器上合并也未尝不可,因为 Git 是不分主次仓库的,同一项目的

51CTO.com  
技术博客 Blog

#### 4.2. Git pull

从服务器的仓库中获取代码,和本地代码合并。(与服务器交互,从服务器上下载最新代码,等同于:Git fetch + Git merge)

从其它的版本库(既可以是远程的也可以是本地的)将代码更新到本地,例如:“git pull origin master”就是将origin 这个版本库的代码更新到本地的master主分支。

git pull可以从任意一个git库获取某个分支的内容。用法如下:

git pull username@ipaddr:远端repository名远端分支名 本地分支名。这条命令将从远端git库的远端分支名获取到本地git库的一个本地分支中。其中,如果不写本地分支名,则默认pull到本地当前分支。

需要注意的是,git pull也可以用来合并分支。和git merge的作用相同。因此,如果你的本地分支已经有内容,则git pull会合并这些文件,如果有冲突会报警。

例如:

为实现与 Tzc 的工作合并,Lyr 执行了以下操作:

```
$ cd ~/work/m2ge
$ git pull tzc@192.168.0.5:~/work/m2ge
```

git-pull 命令可将属于同一项目的远端仓库与同样属于同一项目的本地仓库进行合并,它包含了两个操作:从远端仓库中取出更新版本,然后合并到本地仓库。上述命令可在 Lyr 的 m2ge 仓库中完成对 Tzc 机器上的 myge 仓库的合并。

如果 Lyr 与 Tzc 是对了不同的文件进行了改动,那么可以不费周折地完成仓库合并。但是倘若二人对一些相同的文件进行了改动,那么在合并时必然会遭遇合并冲突的问题,此时手动修改发生合并冲突的文件,然后将结果提交到本地仓库。由,

现在假设 Lyr 与 Tzc 在各自的工作树中对同一份文件 foo.txt 进行了修改,而 foo.txt 原内容如下:

```
one
two
three
```

51CTO.com  
技术博客 Blog

Lyr 对 foo.txt 进行了如下改动,并将该改动提交到本地仓库。

```
ONE
two
three
```

Tzc 对 foo.txt 进行了以下改动,也将该改动提交到本地仓库。

```
one
two
THREE
```

51CTO.com  
技术博客 Blog

当 Lyr 在合并 Tzc 的 Git 仓库时, Git 会自动合并二人对 foo.txt 的修改:

```
ONE
two
THREE
```

现在 Lyr 工作树中的 foo.txt 文件即包含了 Lyr 的改动, 也包含了 Tzc 的改动, 而且合并结果自动作为新版本提交到 Lyr 的仓库中。

观察上述合并冲突示例, 可以看出, 虽然 Lyr 与 Tzc 是对同一份文件进行了修改, 但是他们的修改并未重叠。现在假设二人对 foo.txt 的同一行做出了修改, 那么在仓库合并时会发生什么, 应当如何处理呢?

现在假定上述示例中, Tzc 对 foo.txt 的修改如下:

```
one ONE
two
three
```

这样, 二人对 foo.txt 的同一行进行了不同的修改, 当合并时, Git 会给出以下反馈信息:

```
Auto-merged foo.txt
CONFLICT (content): Merge conflict in foo
Automatic merge failed; fix conflicts and then commit the result.
```

51CTO.com  
技术博客 Blog

上述信息之意是: 尝试合并 foo.txt 文件的改动发生了冲突, 自动合并失败, 请用户手动修复冲突然后将结果提交到仓库中。Lyr 看到上述信息, 就打开了合并后的 foo.txt, 他看到了以下内容:

```
<<<<<< HEAD:foo
ONE
=====
one ONE
>>>>>> 1116d3270764d91a25532a753a47b8b0e1b6f1b8:foo
two
three
```

以一串 < 开头的字符串表示 Lyr 的当前项目版本对 foo.txt 的修改结果, 而以一串 > 开头的字符串表示 Tzc 的当前项目版本对 foo.txt 的修改结果。中间用了一串 = 号将二人修改结果隔开。一旦理解了版本冲突的表示格式, Lyr 就很容易地根据现实情况将合并冲突问题解决掉, 他认为 Tzc 的改动是不符合项目需求的, 并且按

照项目的实际需求进行了手工合并。最后, Lyr 将合并处理结果提交到了仓库中, 成了重叠冲突的合并问题的解决。

51CTO.com  
技术博客 Blog

#### 4.3. Git push

将本地 commit 的代码更新到远程版本库中, 例如 “git push origin” 就会将本地的代码更新到名为 origin 的远程版本库中。

git push 和 git pull 正好想反, 是将本地某个分支的内容提交到远端某个分支上。用法: git push username@ipaddr:远端repository名 本地分支名 远端分支名。这条命令将本地 git 库的一个本地分支 push 到远端 git 库的远端分支名中。

需要格外注意的是, git push 好像不会自动合并文件。因此, 如果 git push 时, 发生了冲突, 就会被后 push 的文件内容强行覆盖, 而且没有什么提示。这在合作开发时是很危险的事情。

例如:



前文在讲述二人协同模式时,强调了 Lyr 与 Tzc 的主次关系,这种关系似乎对于三人以至更多人的协同也有效。现在我们再引入两位故事主角 Lxc 与 Zhu 来说明此问题。假设 Lxc 与 Zhu 仿照 Tzc 那样从 Lyr 那里 git-clone 了一份项目仓库,进行了一番卓有成效的版本更新。最后,Lyr 需要一一取回其他三人的仓库,然后再一一合并方能完成一个协同周期,这些工作逐渐让 Lyr 汗流浹背,疲于应付。因此 Lyr 希望其他三人能够分担一下项目版本合并问题的处理工作。

Git 提供了 git-pull 的对偶命令,即 git-push。顾名思义,git-pull 命令负责从远端仓库取回版本更新,而 git-push 可将本地版本更新推送到远端仓库中。

利用 git-pull 与 git-push 命令,那么在一个协同周期之内,除了 Lyr 之外,其余三人的项目开发流程大致如下:

```
$ git clone lyr@192.168.0.7:~/work/m2ge
... 项目开发 ...
$ git add 改动的文件
$ git commit
$ git pull
... 解决版本合并问题 ...
$ git push
```

在一个协同周期内,Lyr 对 M2GE 仓库的管理工作相当于管理一份他个人项目一般,因为 M2GE 库是位于他的机器上,他是不需要 git-pull 与 git-push 的。

这样,一个基于 Git 较为简单的三人以至更多人的协同工作模式被实现了,这是我们在尚未熟悉 Git 应用之时比较稳妥的协同方案。下一节将介绍 M2GE 库仓库的建立以及多人协同开发过程的具体实现。



4.4. Git fetch

从服务器的仓库中下载代码。(与服务器交互,从服务器上下载最新代码)

相当于从远程获取最新版本到本地,不会自动merge,比Git pull更安全些。

使用此方法来获取服务器上的更新。

例如:如果使用git checkout nov/eclair\_rocket (nov/eclair\_rocket为服务器上的分支名),则是获取上次使用git fetch命令时从服务器上下载的代码;如果先使用 git fetch, 再使用git checkout nov/eclair\_rocket,则是先从服务器上获取最新的更新信息,然后从服务器上下载最新的代码

顶

3

踩

0

- 上一篇
- repo用法详解
- 下一篇
- Android sensor hal 详解

参考知识库



.NET知识库  
2921 关注 | 815 收录



Git知识库  
5580 关注 | 612 收录

猜你在找

- 【国内首套H3C V7交换机实战视频课程-7】ACL和QoS配置
- Windows Server 2012 DHCP Server 管理
- Oracle高级管理
- Oracle初级管理
- Oracle11g服务器、客户端的安装和plsql developer
- Xutil
- 前端面试题2
- 前端面试题大集合
- Web 前端知识点总结

查看评论

2楼 忘世麒麟 2013-07-08 16:13发表

这个挺好的。



1楼 oldbout 2013-07-02 15:40发表



[java]

01. 第三方

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

- 全部主题
- Hadoop
- AWS
- 移动游戏
- Java
- Android
- iOS
- Swift
- 智能硬件
- Docker
- OpenStack
- VPN
- Spark
- ERP
- IE10
- Eclipse
- CRM
- JavaScript
- 数据库
- Ubuntu
- NFC
- WAP
- jQuery
- BI
- HTML5
- Spring
- Apache
- .NET
- API
- HTML
- SDK
- IIS
- Fedora
- XML
- LBS
- Unity
- Splashtop
- UML
- components
- Windows Mobile
- Rails
- QEMU
- KDE
- Cassandra
- CloudStack
- FTC
- coremail
- OPhone
- CouchBase
- 云计算
- iOS6
- Rackspace
- Web App
- SpringSide
- Maemo
- Compuware
- 大数据
- aptch
- Perl
- Tornado
- Ruby
- Hibernate
- ThinkPHP
- HBase
- Pure
- Solr
- Angular
- Cloud Foundry
- Redis
- Scala
- Django
- Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved