文档编号：



**PICC数据仓库系统**

**Git使用指南**

**当前版本：V0.5**

**版本日期：2017年1月12日**

目录

[目录 2](#_Toc473990051)

[1. Git介绍 4](#_Toc473990052)

[1.1 Git介绍 4](#_Toc473990053)

[1.2 与集中型版本控制对比 4](#_Toc473990054)

[1.2.1 集中型： 4](#_Toc473990055)

[1.2.2 分布式： 5](#_Toc473990056)

[2. 安装部署 7](#_Toc473990057)

[2.1 SERVER安装 7](#_Toc473990058)

[2.2 CLIENT安装 7](#_Toc473990059)

[3. 用户管理 7](#_Toc473990060)

[4. 基本操作 8](#_Toc473990061)

[4.1 Git init - 初始化仓库 8](#_Toc473990062)

[4.2 Git status - 查看仓库状态 9](#_Toc473990063)

[4.3 Git add - 向暂存区中添加文件 10](#_Toc473990064)

[4.4 Git commit - 保存仓库的历史记录 10](#_Toc473990065)

[4.5 Git log - 查看提交日志 11](#_Toc473990066)

[4.6 Git diff - 查看更改前后的差别 12](#_Toc473990067)

[4.7 git push -u origin master，把本地仓库的提交推送到远程仓库 15](#_Toc473990068)

[5. 分支模块 15](#_Toc473990069)

[5.1 分支模型 15](#_Toc473990070)

[5.2 特性分支 16](#_Toc473990071)

[5.3 主干分支（保护分支） 17](#_Toc473990072)

[5.6.1 Master分支与develop分支的区别 17](#_Toc473990073)

[5.6.2 分支关系类似下图： 18](#_Toc473990074)

[5.4 发布分支 18](#_Toc473990075)

[5.5 修复分支 19](#_Toc473990076)

[5.6 分支操作 20](#_Toc473990077)

[5.6.1 分支的创建、合并、删除 20](#_Toc473990078)

[5.6.2 git merge - 合并分支 20](#_Toc473990079)

[5.6.3 git log –graph - 以图形形式查看分支 20](#_Toc473990080)

[6. 从远程仓库获取 21](#_Toc473990081)

[6.1 Git clone - 获取远程仓库 21](#_Toc473990082)

[6.1.1 获取远程仓库 21](#_Toc473990083)

[6.1.2 获取远程的feature-D分支 22](#_Toc473990084)

[6.1.3 向本地的feature-D分支提交更改 22](#_Toc473990085)

[6.1.4 推送feature-D分支 22](#_Toc473990086)

[6.2 Git pull - 获取最新的远程仓库分支 22](#_Toc473990087)

[7. 推送至远程仓库 23](#_Toc473990088)

[7.1 Git remote add - 设置（添加、关联）远程仓库 23](#_Toc473990089)

[7.2 Git push - 推送至远程仓库 24](#_Toc473990090)

[7.2.1 推送至master分支 24](#_Toc473990091)

[7.2.2 推送至master以外的分支 24](#_Toc473990092)

[8. 使用Git的开发流程 25](#_Toc473990093)

[8.1 以发布为中心的开发模式 25](#_Toc473990094)

[8.1.1 流程步骤详解： 25](#_Toc473990095)

[8.1.2 流程图A successful Git branching model： 26](#_Toc473990096)

[9. 模拟体验Git开发流程 26](#_Toc473990097)

[9.1 在特性分支feature中进行的工作步骤 26](#_Toc473990098)

[9.2 假设实现一个添加用户的功能 27](#_Toc473990099)

[9.2.1 创建分支 27](#_Toc473990100)

[9.2.2 在分支中进行作业 27](#_Toc473990101)

[9.2.3 Push到远程仓库（发送pull request） 27](#_Toc473990102)

[9.2.4 通过代码审查提高代码质量 28](#_Toc473990103)

[9.2.5 更新本地的develop分支 28](#_Toc473990104)

[9.2.6 在release分支中进行的工作 29](#_Toc473990105)

[9.3 更新到远程仓库 31](#_Toc473990106)

[9.3.1 Develop分支： 31](#_Toc473990107)

[9.3.2 master分支 31](#_Toc473990108)

[9.3.3 Tag标签 31](#_Toc473990109)

[9.4 在hotfix分支中进行的工作 31](#_Toc473990110)

[10. 权限管理 32](#_Toc473990111)

[11. 特殊操作 33](#_Toc473990112)

[11.1 Git reset - 回溯历史版本 33](#_Toc473990113)

[11.1.1 回溯到创建feature-A分之前 33](#_Toc473990114)

[11.1.2 创建fix-B分支：*git checkout –b fix-B* 34](#_Toc473990115)

[11.1.3 推进至feature-A分支合并后的状态 35](#_Toc473990116)

[11.2 消除冲突 37](#_Toc473990117)

[11.2.1 解决冲突 37](#_Toc473990118)

[11.2.2 提交解决冲突后文件： 38](#_Toc473990119)

[11.3 git commit –amend - 修改提交信息 38](#_Toc473990120)

[11.4 git rebase –i - 压缩历史 40](#_Toc473990121)

1. **Git介绍**
2. **Git介绍**

Git是一个开源的分布式版本控制系统，用以有效、高速的处理从很小到非常大的项目版本管理。

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。

Git具有以下特性：

* 直接记录快照，而非差异比较
* 完全分布式，近乎所有的操作都是在本地执行
* 时刻保持数据完整性（SHA-1哈希值）
* 分支功能简单实用
* 更少的仓库污染

1. **与集中型版本控制对比**
2. 集中型：

对于版本控制，大家比较熟悉的是CVS和SVN，这些都属于集中化的版本控制系统CVCS，它能有效的解决不同系统上的开发者协同工作，不同的人都是直接面向中心库进行操作：



集中型版本控制，工作区直接跟版本库打交道，为两区结构：



A：从版本库中检出内容到工作区。

B：从工作区提交内容到版本库。

1. 分布式：

Git是属于分布式版本控制系统DVCS，特点是所有的仓库都是完整的镜像，假如版本控制服务器发生了故障，可以用任何一个本地仓库进行恢复。



Git分布式为三区结构：工作区—暂存区—版本库，跟二区的不同在中间加了一个暂存区。



工作区直接跟暂存区打交道，暂存区跟版本库打交道：

A：从工作区中添加内容到暂存区。

B：将暂存区的内容提交到版本库。

C：从版本库中检出内容到工作区。

1. **安装部署**
2. SERVER安装

选定GitLab作为Git服务器，下载一键安装介质，进行程序安装及配置。

服务端IP：XX.XXX.XXX.XXX，Linux

1. CLIENT安装

使用Git前，每人需安装Git客户端，安装版本：（*SVN软件目录：svn\trunk\08\_资源域\02\_工具\Git*）

./SOFT/Git.for.windows/Git-2.5.1-64-bit.exe

./SOFT/Git.for.windows/TortoiseGit.exe

./SOFT/Git.for.windows/GUI/SourceTreeSetup\_1.6.23.exe（需.NET Framework 4.5）

1. **用户管理**

**gitlab网址：**

人员列表：

* *Gitlab端添加用户*
* SSHKEY认证配置
* 打开Git Bash命令行工具，执行命令ssh-keygen -t rsa -C Email-Addresss生成一个密钥对。
* 登录到GitLab，点击右上角你的用户头像，点击Edit Profile settings，点击SSH Keys，点击Add SSH Key，填写Title栏，复制用户目录下.ssh/id\_rsa.pub文件的内容到Key，点击Add Key。
* 配置本地操作用户

用户名和邮箱地址是本地git客户端的一个变量，不随git库而改变。每次commit都会用用户名和邮箱纪录。

*git config --global user.name "aaa"*

*git config --global user.email* [*aaa@picc.com.cn*](mailto:aaa@picc.com.cn)

* 查看用户名和邮箱地址：

*$ git config user.name*

*$ git config user.email*

* 修改用户名和邮箱地址：

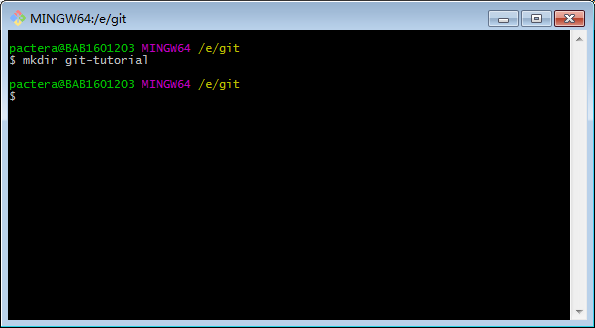
*$ git config --global user.name "username"*

*$ git config --global user.email "email"*

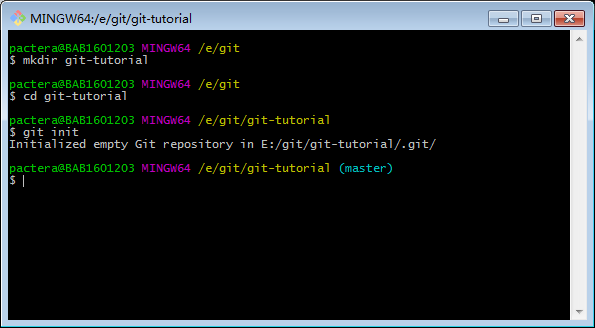
1. **基本操作**
2. **Git init - 初始化仓库**

要使用Git进行版本管理，先初始化仓库。Git是使用*git init --bare* 命令进行初始化，首先建立一个目录并初始化仓库。

创建仓库目录：*mkdir git-tutorial*



初始化仓库：*git init*

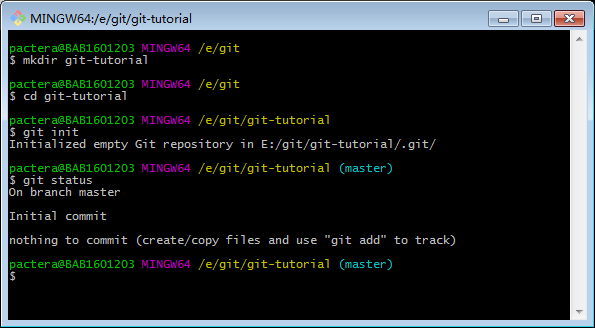


执行git init 命令的目录下就会产生.git目录，用来存储管理当前目录内容所需的仓库数据。

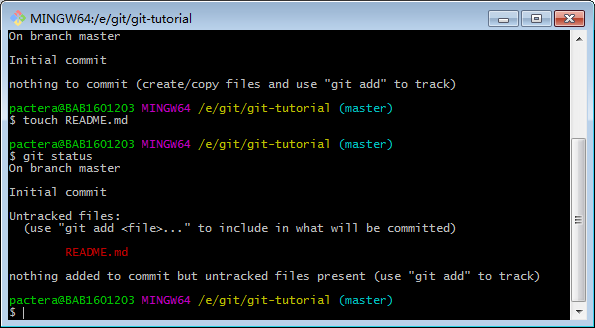
在Git中，称这个目录的内容为“附属于该仓库的工作树”，文件的编辑等操作，在工作树中进行。

1. **Git status - 查看仓库状态**

git status 显示Git仓库的状态，十分常用，请务必牢记。



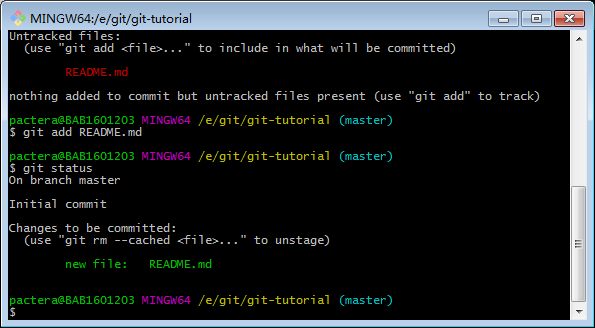
Touch README.md，查看status变化



1. **Git add - 向暂存区中添加文件**

如果只是用Git仓库的工作树创建了文件，文件并不会被记录入Git仓库的版本管理对象当中。

想让文件成为Git仓库的管理对象，就需要用git add 将其加入暂存区（Stage或者Index）中，暂存区是提交之前的一个临时区域。

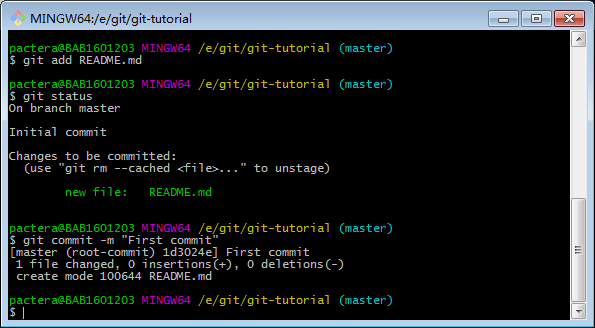


查看状态，显示为Changes to be committed。

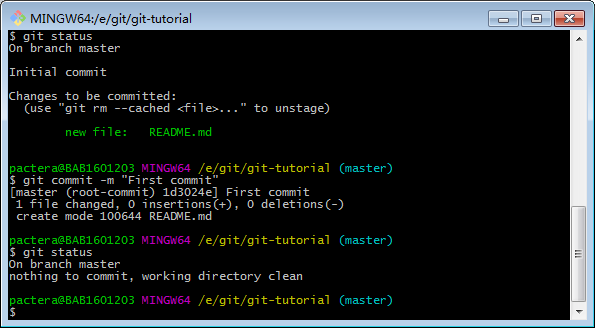
1. **Git commit - 保存仓库的历史记录**

Git commit将当前暂存区中的文件实际保存到仓库的历史记录中，通过这些记录，就可以在工作树中复原文件。

*git commit -m "注释"*，把暂存区内容提交到本地仓库

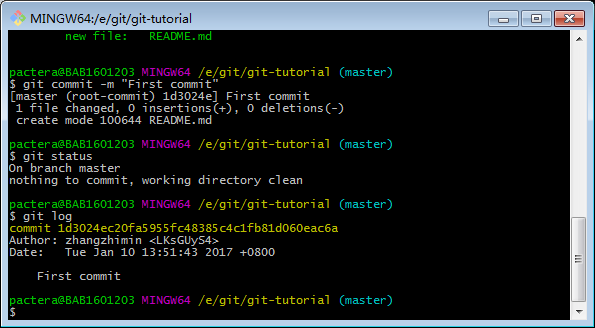


查看当前提交后的状态：

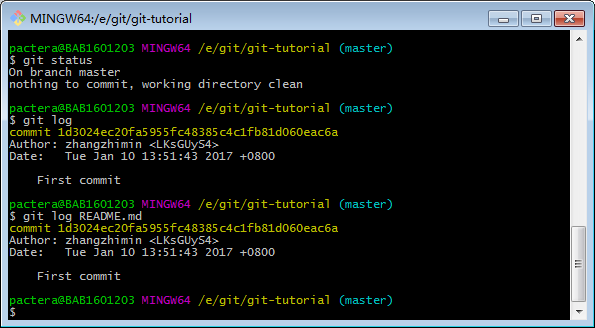


工作树显示完成提交的最新状态，结果显示没有更改。

1. **Git log - 查看提交日志**

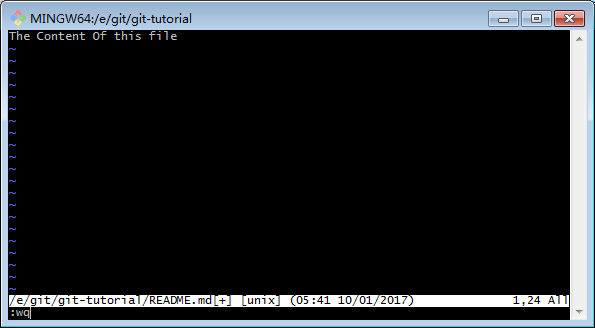


或指定查看具体目录、文件的日志：

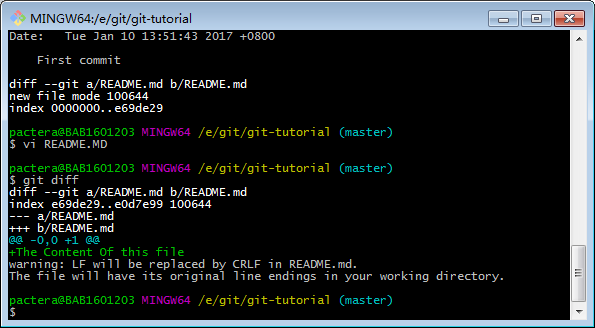


1. **Git diff - 查看更改前后的差别**

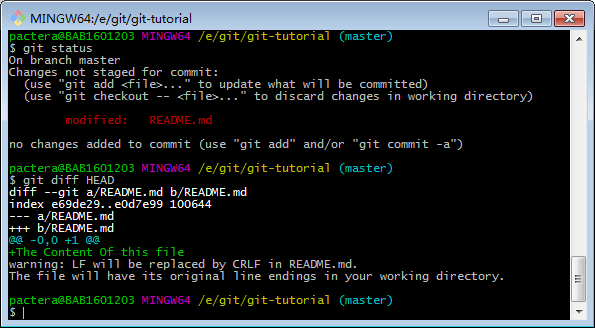
首先在刚才的文件中添加内容：



查看变化：



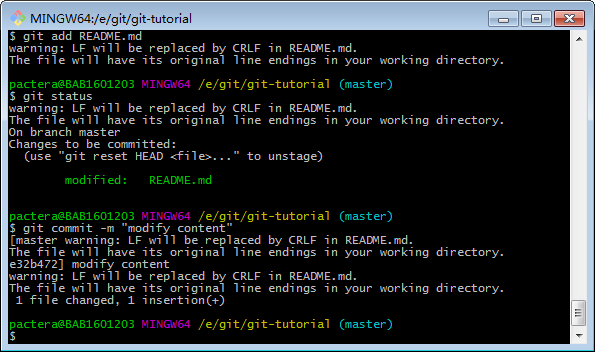
查看**本次提交和上次提交的差别*git diff HEAD*：**



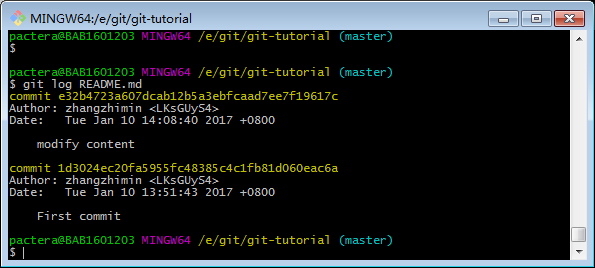
提交修改：

*Git add README.md*

*Git commit –m “modify content”*



查看过程日志：



1. **git push -u origin master，把本地仓库的提交推送到远程仓库**
2. **Git常用命令参考速查表**

****

1. **分支模块**

在进行多个并行作业时，会用到分支，往往存在多个最新代码状态。

1. **分支模型**

**整体分支流程图:**



1. **特性分支**

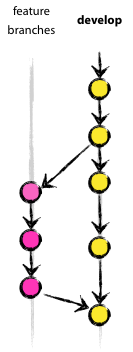
Git与其他集中型版本管理系统不同，创建分支时不需要连接中央仓库，所以能够相对轻松的创建分支。当今当部分工作流程中都用到特性（Topic）分支。

**特性分支定义：**是集中实现单一特征（主题），除此之外不进行任何作业的分支，即便在开发中遇到BUG，也需要再创建新的分支，在新分支中进行BUG修改。

从develop分支创建，用于特性开发，完成后要合并回develop分支。   
操作过程：

* git checkout -b newfeature develop，从develop分支创建newfeature特性分支
* git checkout develop，开发完成后，需要合并回develop分支，先切换到develop分支
* git merge --no-ff newfeature，合并回develop分支，必须加--no-ff参数
* git branch -d newfeature，删除特性分支
* git push origin develop，把合并后的develop分支推送到远程仓库

分支关系类似下图:



1. **主干分支（保护分支）**

主干分支是上面所说的特性分支的原点，同时也是合并的终点，通常开发中会用master分支作为主干分支，主干分支中没有开发到一半的代码，可随时发布及供他人查看。

有时需要同标签Tag等创建版本信息，同时管理多个版本发布，拥有多个版本发布时，主干分支也有多个。

* master 主分支，稳定代码，为生产环境做准备的
* develop 开发分支，为开发服务
  1. **Master分支与develop分支的区别**
* **Master分支：**

Master为时常保持可发布状态的分支，不允许开发者直接对master分支的代码进行修改和提交。

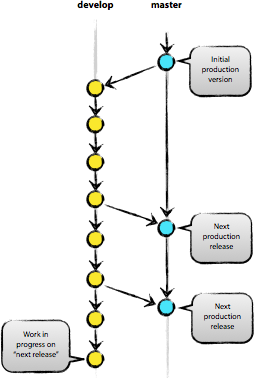
其他分支的开发工作进展到可以发布的程度后，将会与master分支进行合并，而且这一合并只在发布成品是进行，发布时会附加包含版本编号的Git标签（Tag）。

* **Develop分支**

Develop分支是开发过程中的代码中心分支，与master分支一样，这个分支也不允许开发者直接进行修改和提交。

开发者要以develop分支为起点新建feature分支，在feature分支中进行新功能的开发或者代码的修改。也就是说，develop分支维持着开发过程中的最新源代码，以便程序员创建feature分支进行自己的工作。

* 1. **分支关系类似下图：**

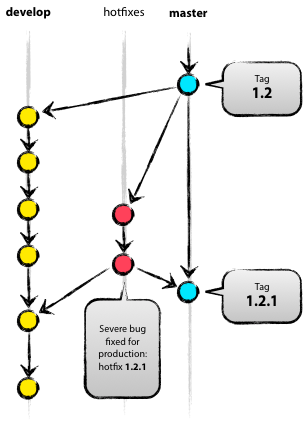


1. **发布分支**

* 从develop分支创建，用于预发布版本，允许小bug修复，完成后要合并回develop和master。
* 操作过程：
* git checkout -b release-1.2 develop，创建一个发布分支
* git checkout master，切换到master分支，准备合并
* git merge --no-ff release-1.2，把release-1.2分支合并到master分支
* git tag 1.2，从master分支打一个标签
* git checkout develop，切换到develop分支，准备合并
* git merge --no-ff release-1.2，把release-1.2分支合并到develop分支
* git branch -d release-1.2，删除这个发布分支

1. **修复分支**

* 从master分支创建，用于生产环境上的Bug修复，完成后要合并回develop和master。
* 操作过程：
* git checkout -b hotfix-1.2.1 master，从master分支创建一个Bug修复分支
* git checkout master，切换到master分支，准备合并
* git merge --no-ff hotfix-1.2.1，合并到master分支
* git tag 1.2.1，为master分支创建一个标签
* git checkout develop，切换到develop分支，准备合并
* git merge --no-ff hotfix-1.2.1，合并到develop分支
* git branch -d hotfix-1.2.1，删除hotfix-1.2.1分支
* 分支关系类似下图:



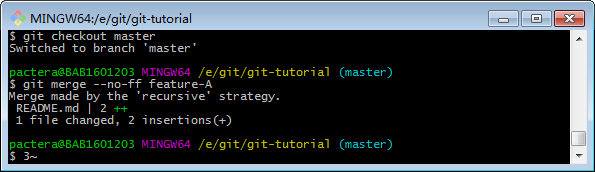
1. **分支操作**
   1. **分支的创建、合并、删除**

* git branch，显示所有分支
* git branch b1，从当前分支创建一个叫b1的分支
* git checkout b1，切换到b1分支
* git checkout -b b1，相当于以上两条命令的组合
* git checkout master，切换到master主分支
* git merge b1，把b1分支的代码合并到master上
* git branch -d b1，删除b1分支，不能在被删除分支上执行
  1. **git merge - 合并分支**

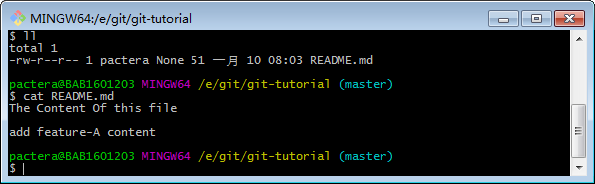
接下来假设feature-A已经实现完毕，想要将开发的内容合并到master分支中，首先切换到master分支中：

***git checkout master***

***git merge –no-ff feature-A***



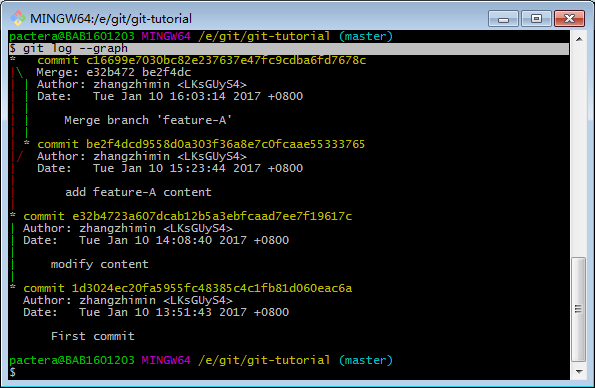
上面进行了切换及合并，下面查看一下合并后的内容，看是否有了feature-A修改提交的内容：



查看文件中已经包含了feature-A修改提交的内容。

* 1. **git log –graph - 以图形形式查看分支**

用git log –graph命令进行查看的话，能很清楚看到特性分支提交的内容已被合并，除此之外，特性分支的创建以及合并都清楚明了。



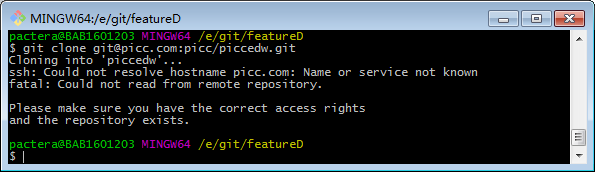
1. **从远程仓库获取**

假设当前时间又有个另一位新同事参与进来共同开发，需要执行从远程仓库获取内容的相关操作。

1. Git clone - 获取远程仓库
2. 获取远程仓库

将gitlab上的仓库clone到本地：

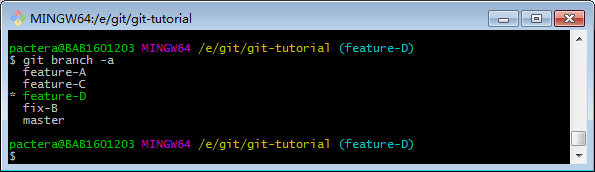
*git clone git@IP:AAA/AAA.git*



*（因远程仓库地址不可用，所以提示不能解析远程仓库地址。）*

执行git clone命令后，默认处于master分支下，同时系统会自动将origin设置成远程仓库的标识符。即当前本地仓库的master分支与gitlab端远程仓库（origin）的master分支在内容上是完全相同的。

查看分支相关信息：



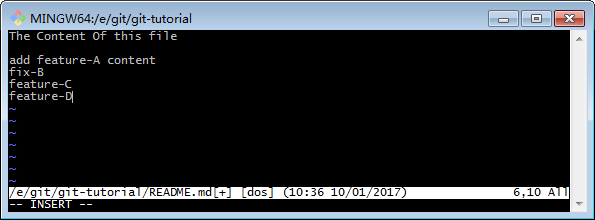
使用git branch –a 命令 查看，添加-a参数可以同时显示本地仓库和远程仓库的分支信息。

1. **获取远程的feature-D分支**

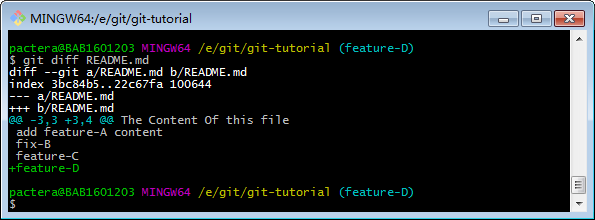
*git checkout –b feature-D origin/feature-D*

1. **向本地的feature-D分支提交更改**

假设现在开发者是另一名新开发者，要做一个提交，在README.md中添加一行文字feature-D：



查看变化：

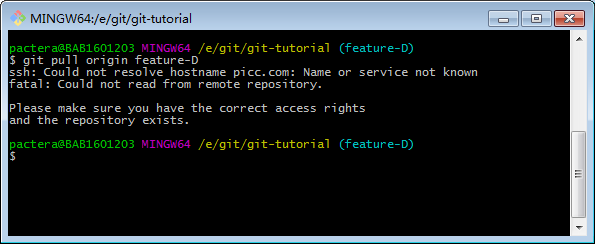


1. **推送feature-D分支**

*git push origin feature-D*

1. **Git pull - 获取最新的远程仓库分支**

*git pull*



*（因远程仓库地址不可用，所以提示不能解析远程仓库地址。）*

当多名开发者在同一个分支中进行作业时，为减少冲突情况发生，建议更频繁的进行push和pull操作。

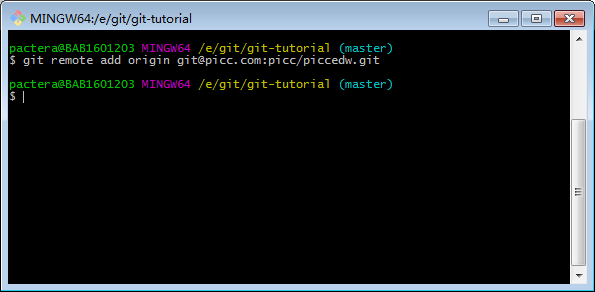
1. **推送至远程仓库**

远程仓库，是与我们本地仓库相对独立的另一个仓库，目前我们使用gitlab为远程仓库服务器，首先在gitlab上创建一个仓库，并将其设置为本地仓库的远程仓库。

1. **Git remote add - 设置（添加、关联）远程仓库**

用git remote add 命令将远程仓库设置成本地仓库的远程仓库。

*git remote add origin git@AAA:AAA/AAA.git*



执行命令之后，git会自动将git@AAA:AAA/AAA.git远程仓库的名称设置为origin（标识符）

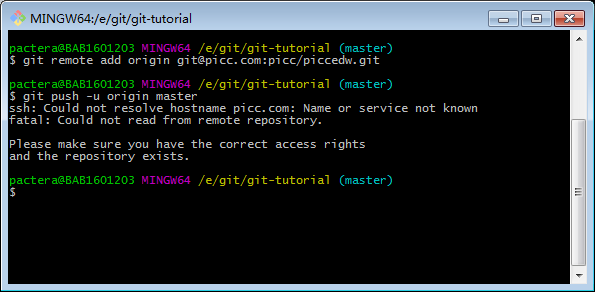
可以通过以下两种方式修改配置的远程仓库地址：

* + 修改命令   
    *git remote origin set-*[*url*](http://www.baidu.com/s?wd=url&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLm1NWPW6kuHuBPHPbuWI90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbdrHfdnWnYn16LnWTvP1R3n0)[*URL*](http://www.baidu.com/s?wd=URL&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLm1NWPW6kuHuBPHPbuWI90ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHbdrHfdnWnYn16LnWTvP1R3n0)
  + 先删后加   
    *git remote rm origin   
    git remote add origin git@AAA:AAA/AAA.git*
  + 直接修改config文件

1. **Git push - 推送至远程仓库**
2. **推送至master分支**

如果要将当前分支下本地仓库中的内容推送给远程仓库，需要用到git push 命令，现在以master分支为基础进行操作：

*git push -u origin master:develop*



*（因远程仓库地址不可用，所以提示不能解析远程仓库地址。）*

执行git push后，当前分支的内容就会被推送给远程仓库origin的master分支，-u参数可以在推送的同时，将origin仓库的master分支设置为本地仓库当前分支的upstream（上游），添加了这个参数，将来运行git pull命令从远程仓库获取内容时，本地仓库的这个分支就可以直接从origin的master分支获取内容，省去了另外添加参数的麻烦。

1. **推送至master以外的分支**

除了master分支之外，远程仓库也可以创建其他分支，例如：

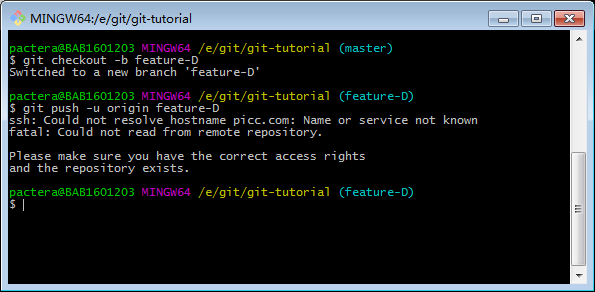
本地仓库创建feature-D分支，并将它已同名形式push至远程仓库：

***git checkout –b feature-D***

***git push –u origin feature-D:***默认push到远端的同名feature-D分支

如果推送到其他分支，通过以下命令执行：

***git push origin 本地分支名:远程分支名***



*（因远程仓库地址不可用，所以提示不能解析远程仓库地址。）*

执行后远程仓库就可以查看到feature-D分支了。

1. **使用Git的开发流程**

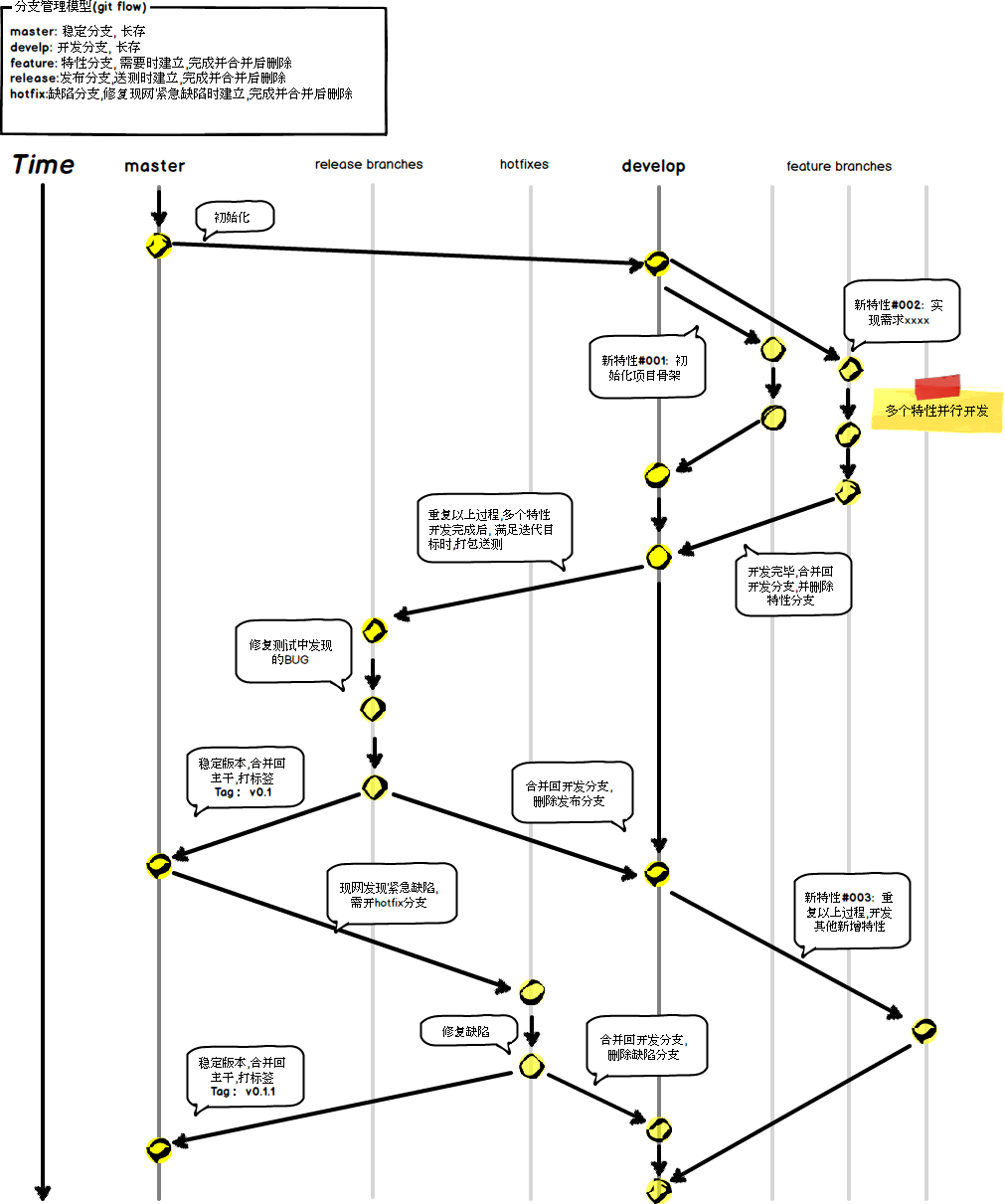


1. **以发布为中心的开发模式**

在这个开发流程中，每个分支都显示出代码的当前状态。流程中设置了负责管理软件发布Release的发布管理员。适用于以发布为中心的软件开发。

从软件开发者的角度观察这一开发流程时会发现，该流程用分支名表示标准软件开发中开发状态的迁移。

1. **流程步骤详解：**
2. 从开发版的分支（develop）创建工作分支（feature branches），进行功能的实现或修正；
3. 工作分支（feature branches）的修改结束后，与开发版的分支（develop）进行合并；
4. 重复上述1和2，不断实现功能直至可以发布；
5. 创建用于发布的分支（release branches），处理发布的各项工作；
6. 发布工作完成后与master分支合并，打上版本标签（Tag）进行发布；
7. 如果发布的程序出现BUG，以打了标签的版本为基础进行修正（hotfixes）。
8. **流程图A successful Git branching model：**

******

整个流程看上去比较好理解，这一流程最大的亮点在于考虑了紧急的BUG应对措施。

在这个流程中，程序员必须理解自己正在进行的修改会对哪些分支产生影响。一个分支的工作结束后，有时需要与多个目标分支合并。这些是该流程中最为复杂的部分，需要谨慎处理。

1. **模拟体验Git开发流程**

接下来以真实实践体验Git 开发流程：

1. **在特性分支feature中进行的工作步骤**

feature分支以develop分支为起点，是开发者直接更改代码发送提交的分支，开发以下述流程进行：

1. 从develop分支创建feature分支；
2. 在feature分支中实现目标功能；
3. 通过GitLab向develop分支发送pull request；
4. 接受其他开发者审查后，将pull request合并至develop分支。

与develop分支合并后，已经完成工作的feature分支就失去了作用，可以适当的时间删除。

1. **假设实现一个添加用户的功能**
2. **创建分支**

因为develop分支是feature分支的起点，所以首先将develop分支新为最新状态，然后创建一个feature分支实现添加用户功能，分支名称定为add-user：

*git pull*

*git branch feature\_name*



1. **在分支中进行作业**



1. **Push到远程仓库（发送pull request）**

首先将feature\_name分支push到GitLab端远程仓库：

*git push origin feature\_name*

如果是与其他开发者共同开发同一个feature分支，那么远程仓库的add-user分支可能已经被更新，所以要养成在push之前先获取pull最新develop分支的习惯，确保上述两点之后再进行push。

1. **通过代码审查提高代码质量**
2. 由其他开发者进行代码审查，在pull request中提供反馈



1. 修正代码以反映反馈内容（在本地feature/add-user分支中）；
2. 将feature/add分支push到远程仓库（自动添加至之前的pull request）；
3. 重复前三步；
4. 确认pull request没有问题后，由其他开发者将其合并至develop分支。

**监测的几个要点：**

1. 没有测试or测试未通过；
2. 违反编码规则；
3. 代码品质过低（命名不明确，方法冗长等）；
4. 还有重构的余地；
5. 由重复部分。
6. **更新本地的develop分支**

发送的pull request在GitLab端与develop合并后，为让其反映到本地的develop分支中，需要进行以下操作：

A：切换到develop分支

B：执行git pull（fetch & merge）

每当需要通过develop分支创建feature等分支时，记得一定要先执行上述操作，保证develop分支处于最新状态。

1. **在release分支中进行的工作**

接下来给程序分配一个版本号进行发布，今后对这个版本的程序只做BUG修复，不再进行其他支持

*git checkout develop*

*git pull*

*git release start ‘1.0.0’*

release/1.0.0分支已经成功创建：



**发布与合并至master分支：**

发布前的修正全部处理完后，结束这一分支

*git release finish ’1.0.0’*



合并后的master分支加入一个与版本号相同编号的标签：

*git merge branch ‘release/1.0.0’*

*git release 1.0.0*



将release分支的状态合并至develop分支：

*git merge branch ‘release/1.0.0’ into develop*



1. **更新到远程仓库**

到目前为止，已经修改了多个分支，现在将各分支利用push操作更新到远程仓库：

1. Develop分支：

*git push origin develop*

1. master分支

*git checkout master*

*git push origin master*

1. Tag标签

*git push –tags*

1. **在hotfix分支中进行的工作**

Hotfix分支并不是预期中计划出现的分支，它是一个紧急应对措施，只有当前发布的版本中出现BUG或漏洞，而且严重程度要求必须立刻处理，无法等到下一个版本发布时，hotfix分支才会被创建。

所以， hotfix分支是以发布版本的标签或master分支为起点，借助hotfix分支，可以不影响develop分支正常开发的情况下，由其他开发者处理修正工作。



*git fetch origin*

*git flow hotfix start ‘1.0.1’ ‘1.0.0’*

*git push origin hotfix/1.0.1*

1. **权限管理**

* **GitLab**有五种身份权限，分别是：
* Owner 项目所有者，拥有所有的操作权限
* Master 项目的管理者，除更改、删除项目元信息外其它操作均可
* Developer 项目的开发人员，做一些开发工作，对受保护内容无权限
* Reporter 项目的报告者，只有项目的读权限，可以创建代码片断
* Guest 项目的游客，只能提交问题和评论内容

为项目添加成员时可指定成员的身份权限。

* **关于权限：**
* Guest 可以提供给测试部，用于提交问题；
* Reporter 可以clone（下载）工程，但不能提交；
* Developer可以clone工程，但是push提交时，只能提交到非保护的分支，然后从网站上请求合并到主分支；
* Master 可以clone工程，可以提交所有分支。负责处理Developer的合并请求，并在Review之后合并到主分支或者修改或者拒绝。
* **保护主分支：**

此为关键

必须将主分支设置为protected，此时只有project的所有者或者Master可以提交以及合并到此分支。未保护的分支Developer也可以提交

注：创建完工程后，必须上传过文件后才可设置

* **命名规则**
* 每次提交必须写明注释，如果是修复Bug，请加上Bug号
* 创建特性分支，名称要以f-开头，加上特性名
* 创建发布分支，名称要以r-开头，加上预发布版本号
* 创建Bug修复分支，名称要以b-开头，加上Bug号
* 创建标签，名称要以t-开头，加上发布版本号
* 合并分支时必须使用--no-ff参数，以保留合并历史轨迹

1. **特殊操作**
2. **Git reset - 回溯历史版本**

git 另一个特性便是可以灵活操作历史版本，借助分散仓库的优势，可以在不影响其他仓库的前提下对历史版本进行操作。

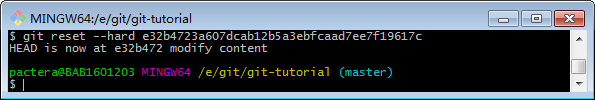
为了回溯历史版本，再创建一个特性分支fix-B：



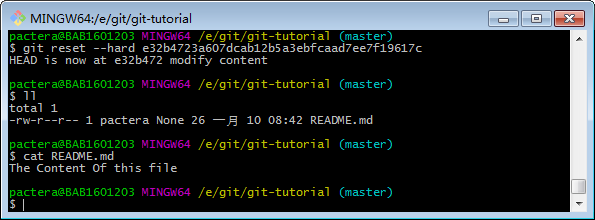
1. **回溯到创建feature-A分之前**

要让仓库的HEAD、暂存区、当前工作树回溯到指定状态，需要用到git reset –hard命令，只要提供目标时间点的哈希值：

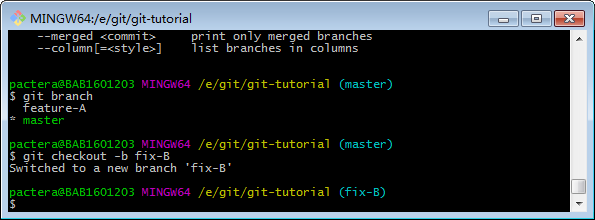
*git reset --hard e32b4723a607dcab12b5a3ebfcaad7ee7f19617c*



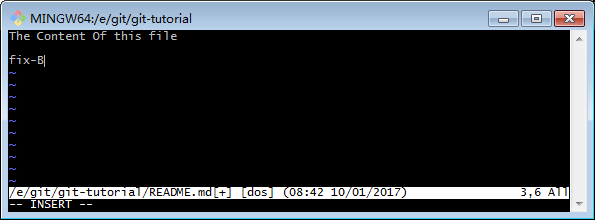
现在已经成功回溯到特性分支feature-A修改之前的状态，README.md文件内容也恢复到了当时的状态。



1. **创建fix-B分支：*git checkout –b fix-B***



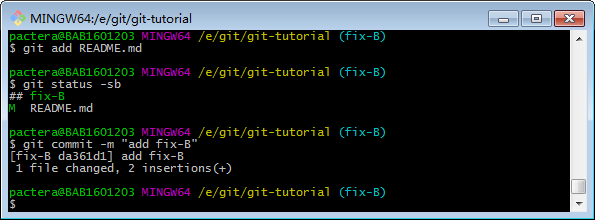
**作为fix-B主题的作业内容，修改README.md ，添加fix-B内容：**



**提交：**

*git add README.md*

*git commit –m “add fix-B”*



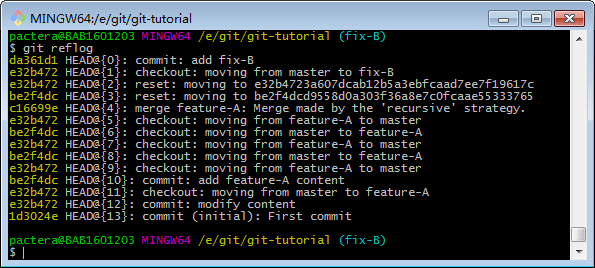
**Fix-B分支的下一步目标：**



1. **推进至feature-A分支合并后的状态**

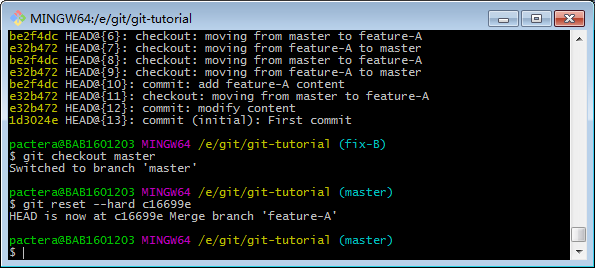
首先恢复到feature-A分支合并后的状态，暂称这一操作为“推进历史”。

git reflog，查看当前仓库执行过的操作的日志：



只要不进行git的GC，就可以通过日志随意调取近期的历史状态，就像给时间机器指定一个时间点，即便开发者错误执行了Git操作，基本也都可以利用git reflog命令恢复到原先的状态。

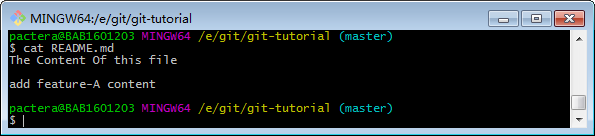
现在将HEAD、暂存区、工作树恢复到feature-A特性分支合并后**c16699e**时间点的状态：



**恢复后状态：**



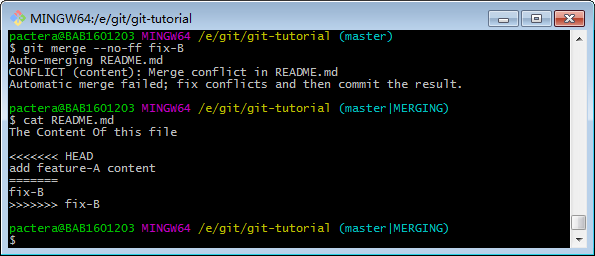
**当前README.md内容：**



1. **消除冲突**

现在首先对fix-B分支内容进行合并：

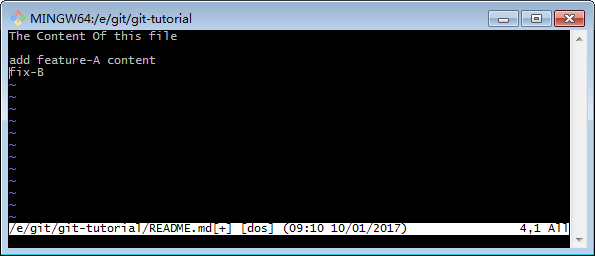
*git merge –no-ff fix-B*



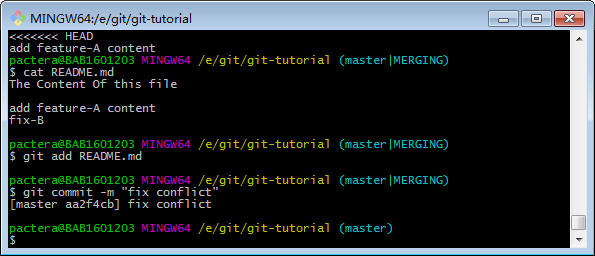
1. **解决冲突**

现在系统显示README.md文件发生了冲突（Conflict），feature-A与fix-B更改的内容发生了冲突。

修正文件，解决冲突：



1. **提交解决冲突后文件：**

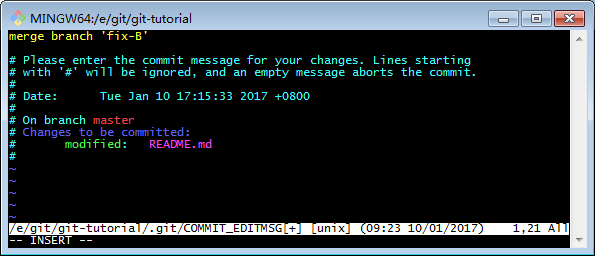


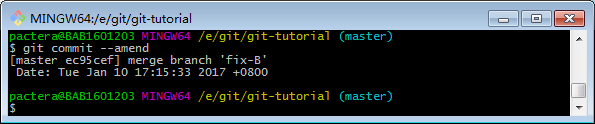
1. **git commit –amend - 修改提交信息**

上一次提交记录信息为“fix conflict”，解决冲突的提交，其实他是fix-B分支的合并，解决合并时发生的冲突只是过程之一，这样标记并不好，于是修改这条提交信息：

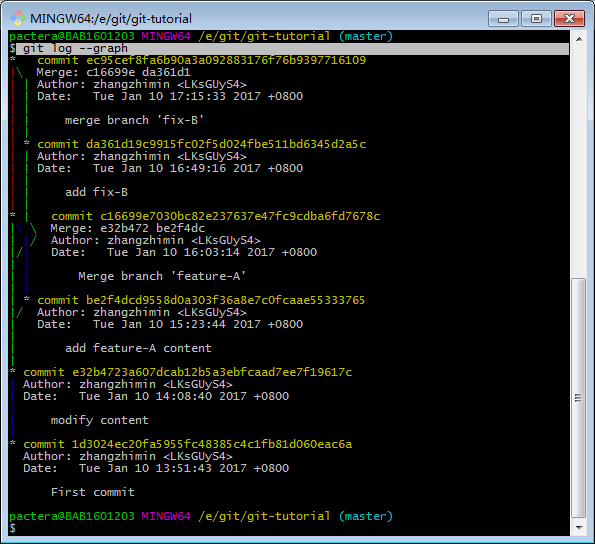
*git commit –amend*

启动编辑器，修改fix Conflict内容为merge branch ‘fix-B’





**通过g*it log –graph*查看提交日志：**



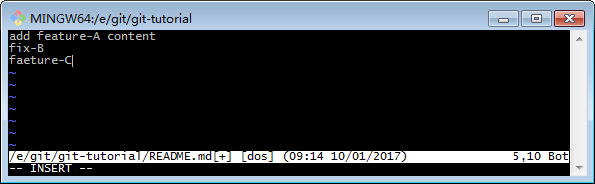
显示提交日志信息已经被修改。

1. **git rebase –i - 压缩历史**

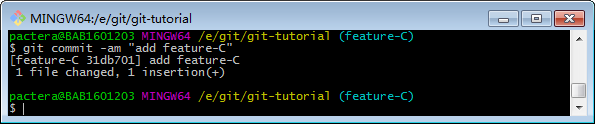
在合并特性分支之前，如果发现已提交的内容中有些许拼写错误等，不妨提交一个修改，然后将这个修改包含到前一个提交当中，压缩成一个历史版本，这是个会经常用到的技巧。

1. **创建feature-C分支**

创建feature-C分支，在README.md文件中添加内容，在内容中留下拼写错误，然后修改：



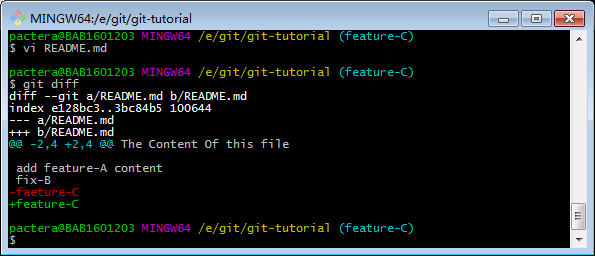
**提交：*git commit –am “add feature-C”***



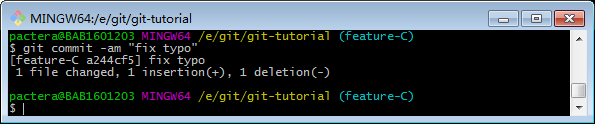
1. **修正拼写错误**

进行README.md文件内容修正，将“feature-C”修正为“feature-C”：

查看修改前后差别：*git diff*



然后进行提交：*git commit –am “fix typo”*



其实在历史记录中不希望看到这类提交，因为健全的历史记录并不需要它们，下面对这类历史进行更改。

1. **更改历史**

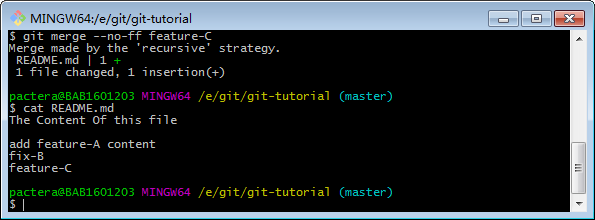
将“fix typo”修正内容与之前一次的提交合并，在历史记录中合并为一次完美的提交，需要用到*git rebase*命令：

git rebase –i HEAD~2

将fix typo左侧的pick部分删除，改为fixup。

这样一来，fix typo 就从历史中被抹去，也就相当于Add feature-C中从来没有出现过拼写错误。

1. 合并至master



Master分支整合了feature-C分支。