

Git&GitLab

1 版本控制工具应该具备的功能

- 协同修改
 - 多人并行不悖的修改服务器端的同一个文件。
- 数据备份
 - 不仅保存目录和文件的当前状态，还能够保存每一个提交过的历史状态。
- 版本管理
 - 在保存每一个版本的文件信息的时候要做到不保存重复数据，以节约存储空间，提高运行效率。这方面SVN 采用的是增量式管理的方式，而Git 采取了文件系统快照的方式。
- 权限控制
 - 对团队中参与开发的人员进行权限控制。
 - 对团队外开发者贡献的代码进行审核——Git 独有。
- 历史记录
 - 查看修改人、修改时间、修改内容、日志信息。
 - 将本地文件恢复到某一个历史状态。
- 分支管理
 - 允许开发团队在工作过程中多条生产线同时推进任务，进一步提高效率。

2 版本控制简介

2.1 版本控制

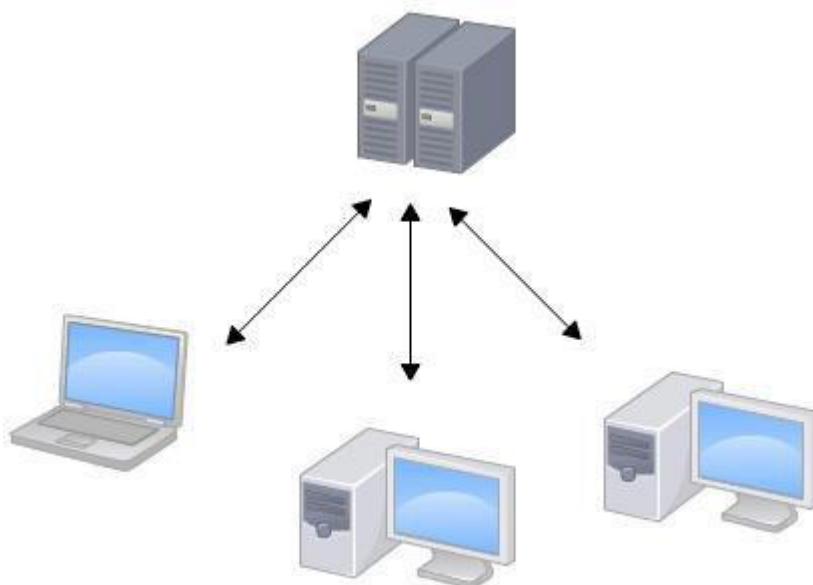
工程设计领域中使用版本控制管理工程蓝图的设计过程。在IT 开发过程中也可以使用版本控制思想管理代码的版本迭代。

2.2 版本控制工具

思想：版本控制 实现：版本控制工具

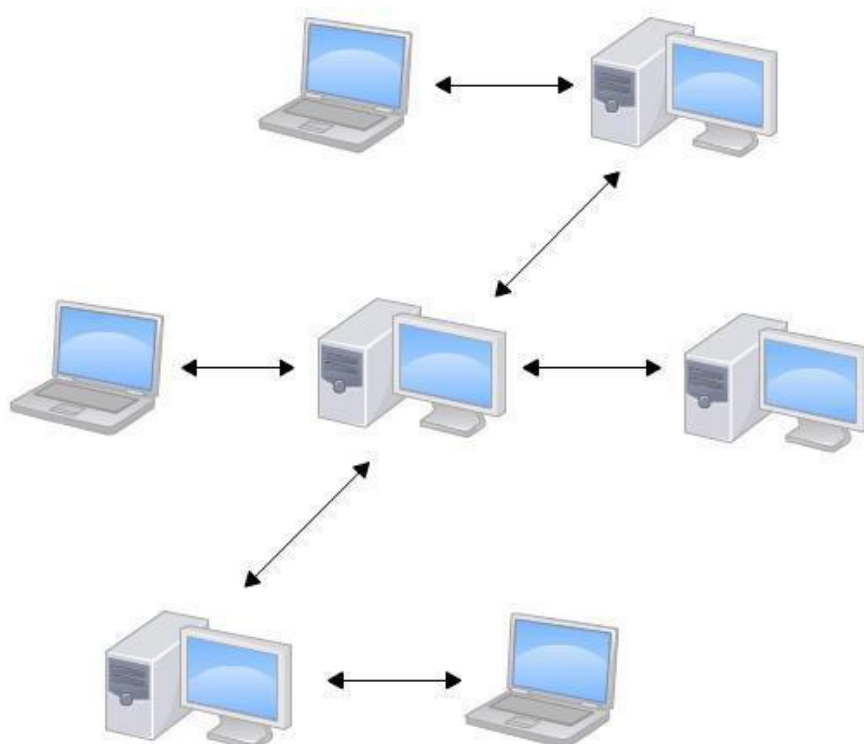
集中式版本控制工具：

CVS、SVN、VSS……



分布式版本控制工具：

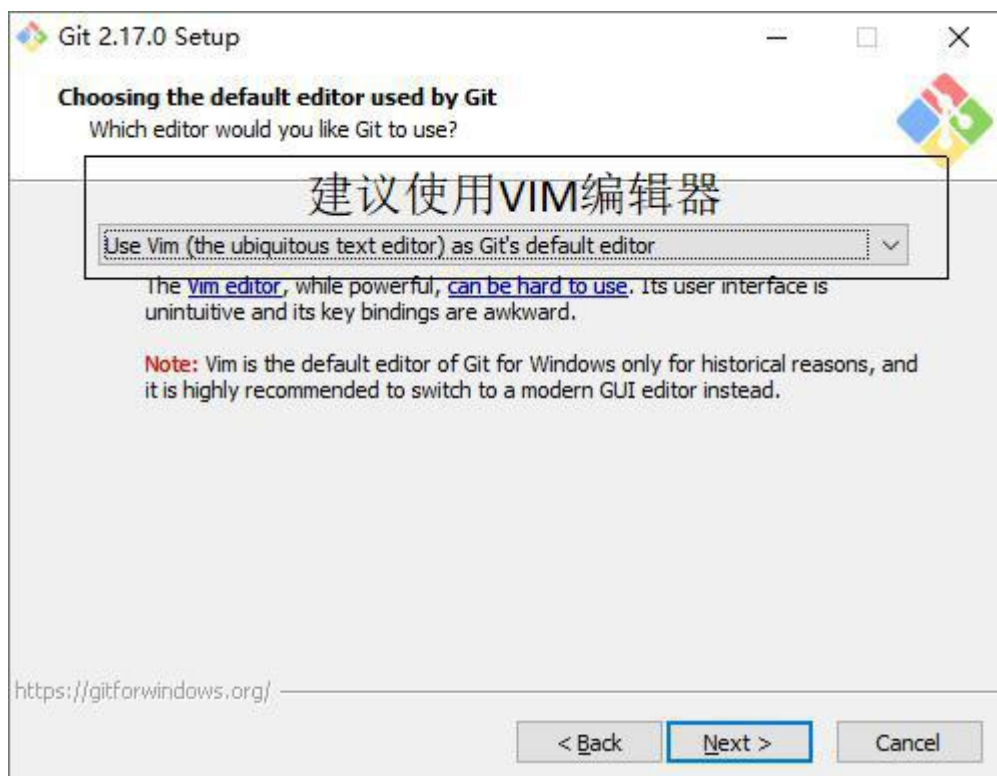
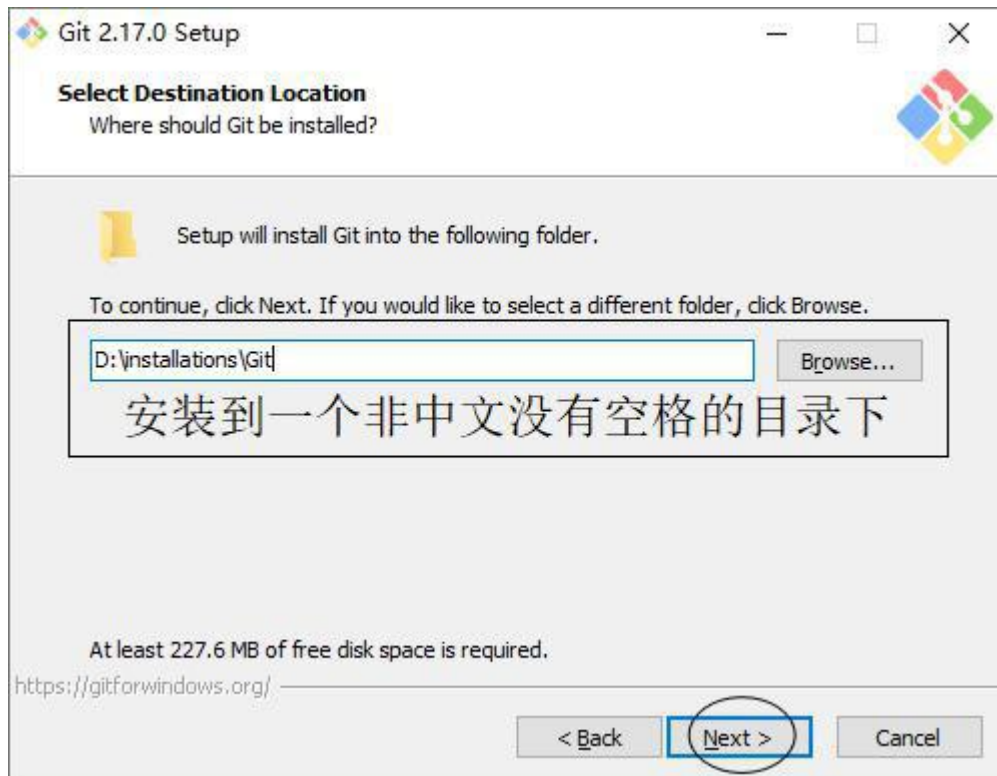
[Git](#)、Mercurial、Bazaar、Darcs……

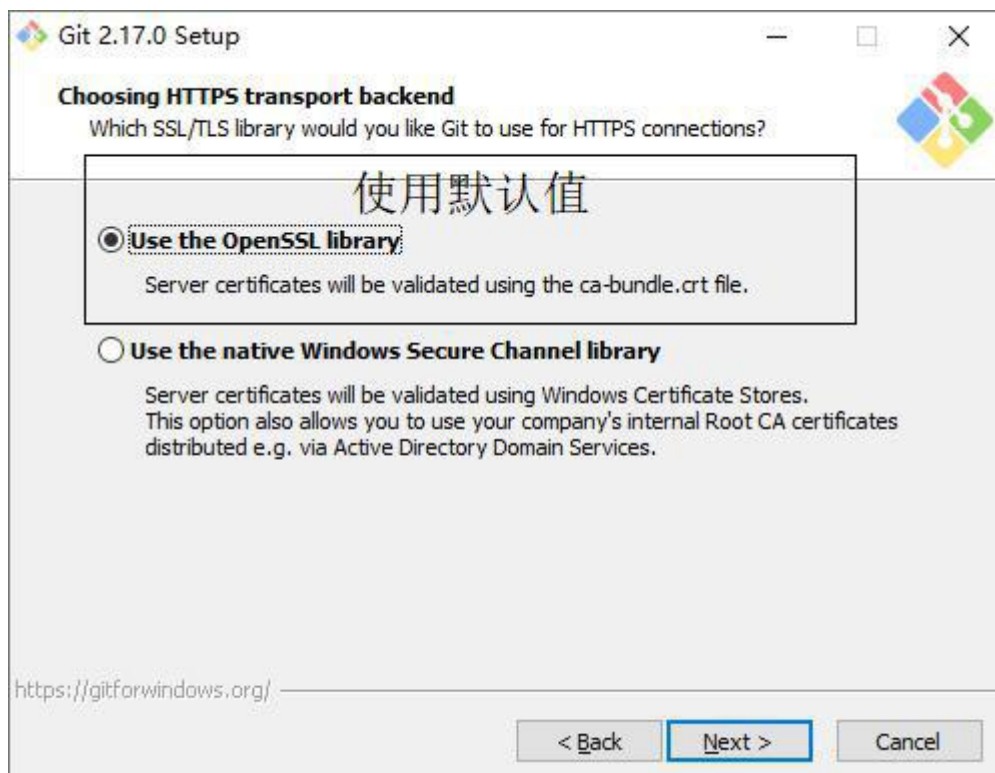
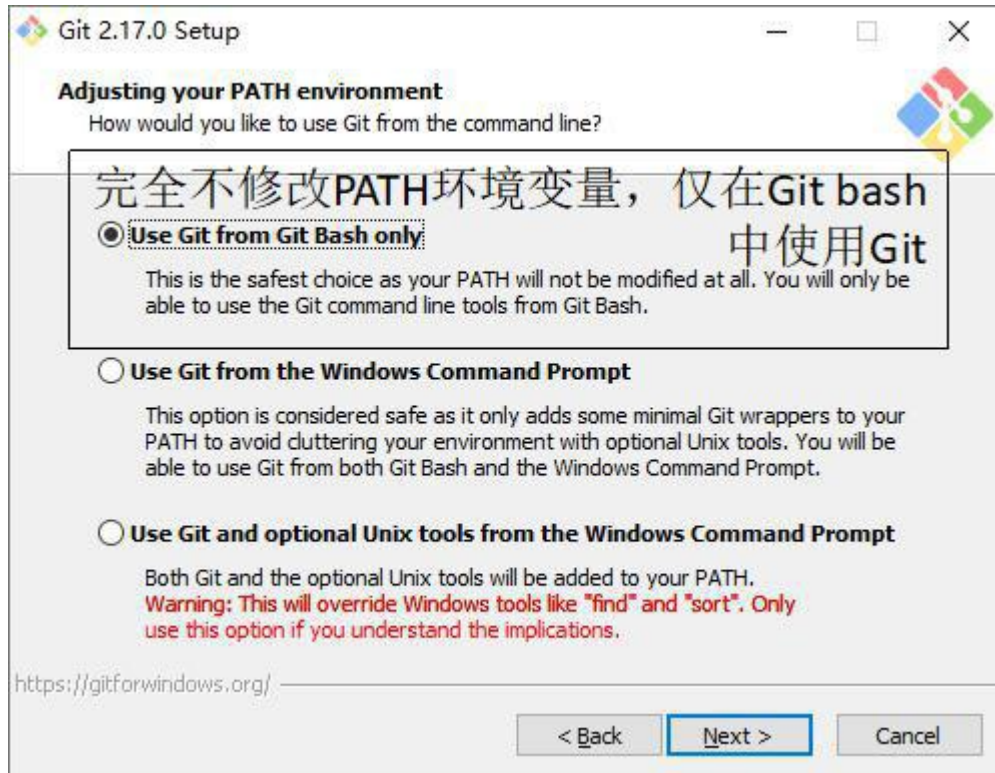


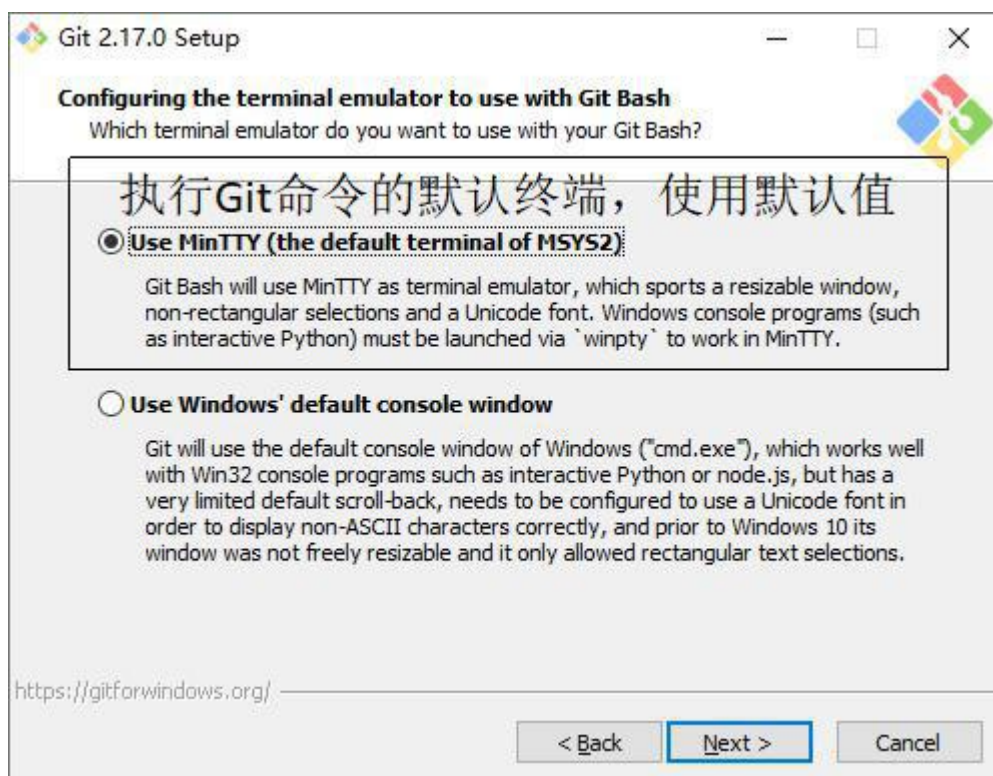
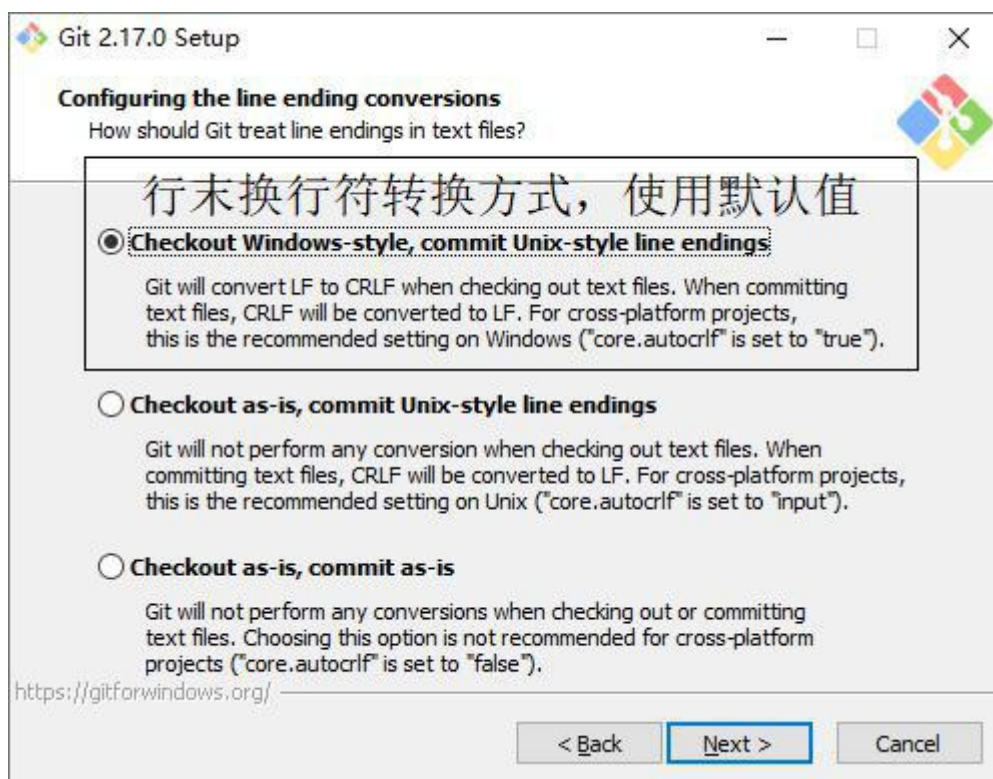
2.3 git的优缺点

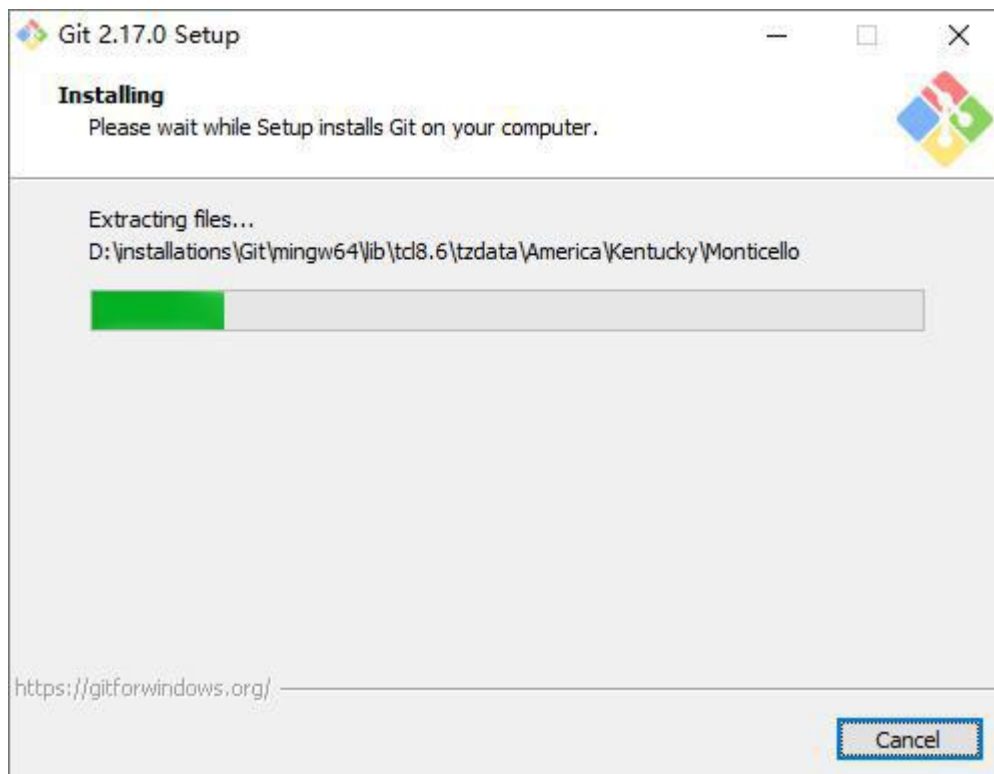
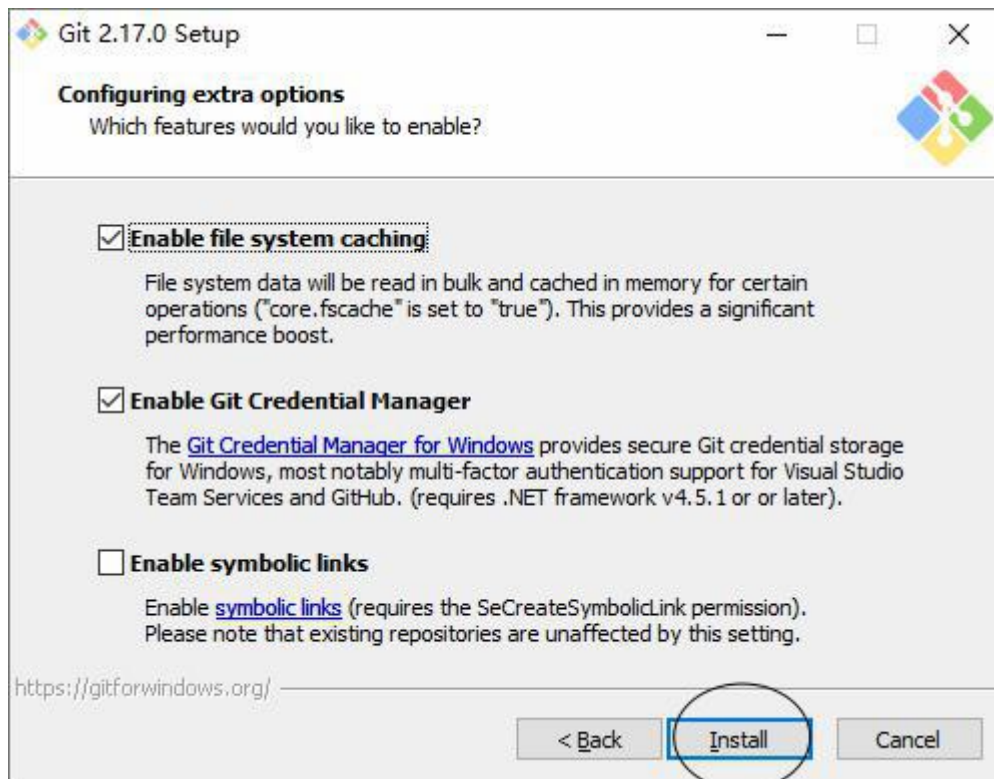
3 Git 简介

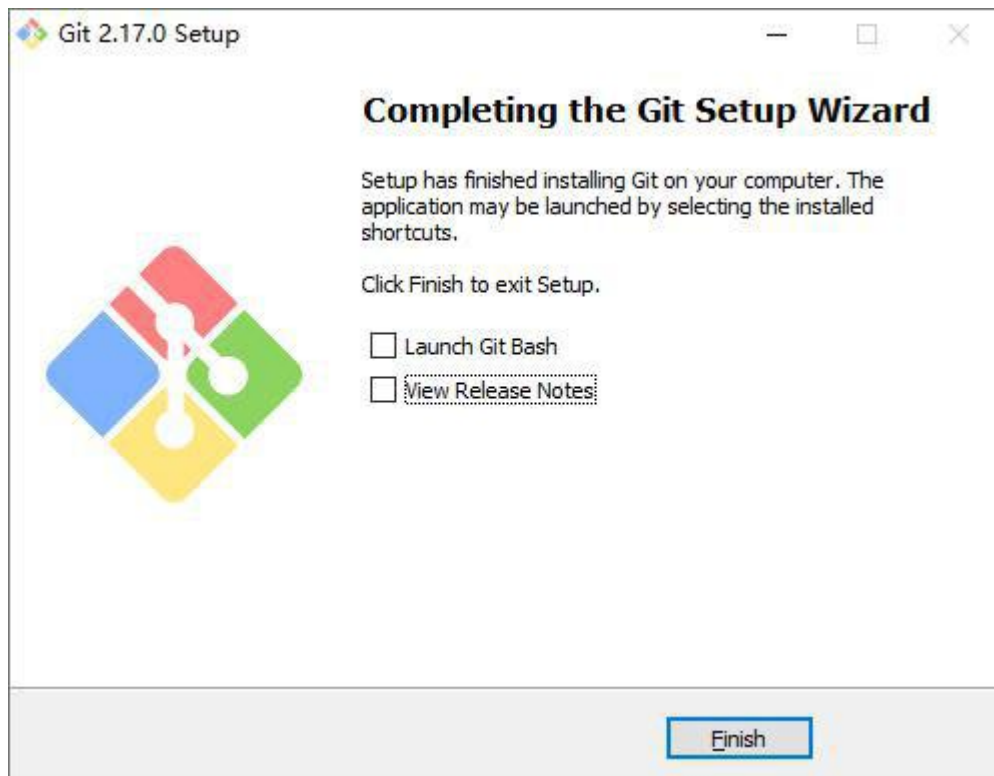
3.1 Git 安装



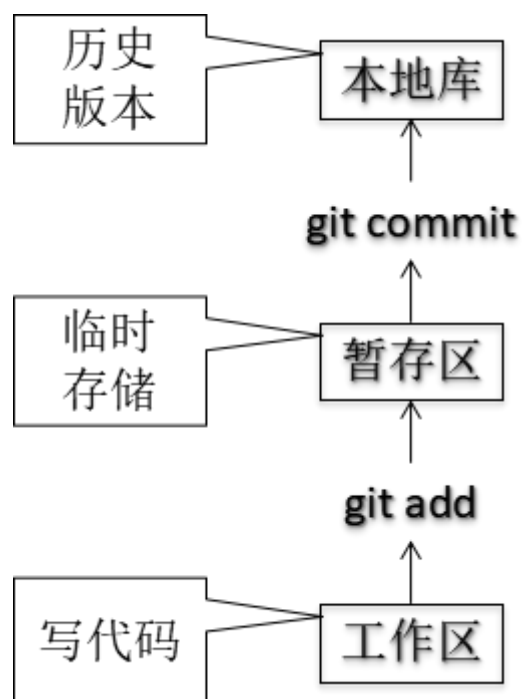








3.5 Git 结构



3.6 Git 和代码托管中心

代码托管中心的任务：维护远程库

局域网环境下

GitLab 服务器

外网环境下

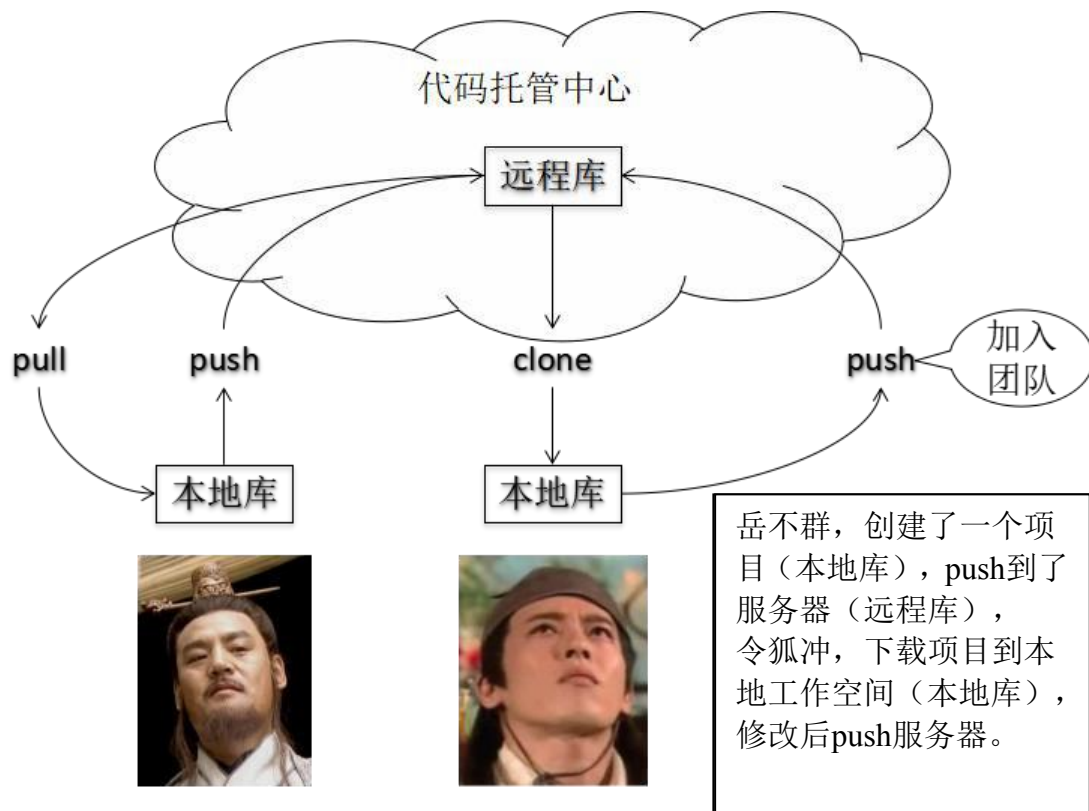
GitHub

码云

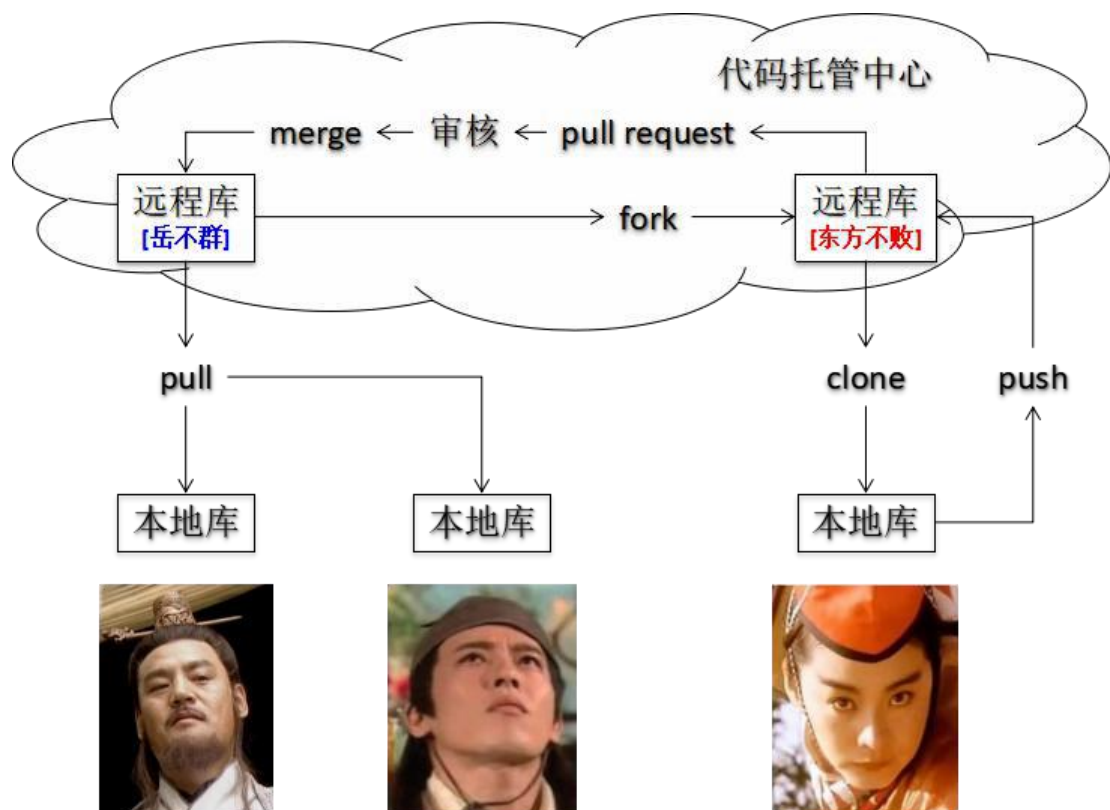
3.7 本地库和远程库

- **Fetch** (获取) , 从远程代码库更新数据到本地代码库。 注意 : Fetch 只是将代码更新到本地代码库 , 你需要检出 (check out) 或与当前工作分支合并 (merge) 才能在你的工作目录中看到代码的改变。
- **Pull** (拉取) , 从远程代码库更新数据到本地代码库 , 并与当前工作分支合并 , 等同于 **Fetch + Merge**。
- **Push** (推送) , 将本地代码库中已提交 (commit) 的数据推送到指定的 **remote** , 没有 commit 的数据 , 不会push
- **HEAD** , 指向你正在工作中的本地分支的指针
- **Master 分支** : 主分支 , 所有提供给用户使用的正式版本 , 都在这个主分支上发布。
- **Tags (标签)** : 用来记录重要的版本历史 , 例如里程碑版本
- **Origin** : 默认的 remote 的名称
- **Git clone** (克隆版本库) : 从服务端将项目的版本库克隆下来
- **Git init** (在本地初始化版本库) : 在本地创建版本库的时候使用
- **pull request** 拉取请求 , 请求项目的拥有人使用你的代码
- **remote** 远程库的操作命令

3.7.1 团队内部协作



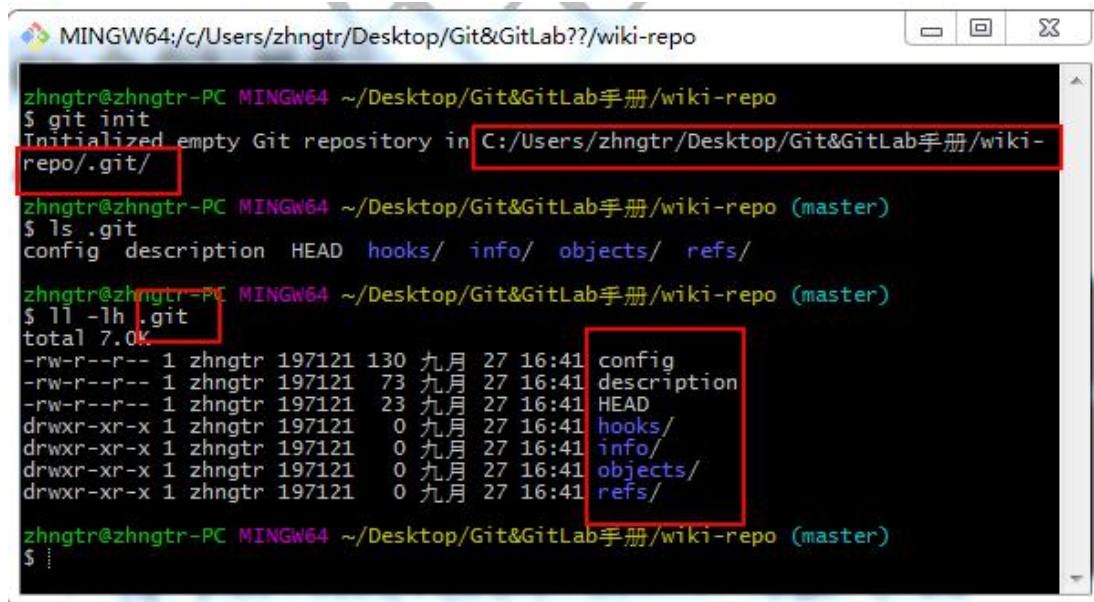
3.7.1 团队外部协作



4 Git 命令行操作

4.1 本地库初始化

命令 : git init



```
MINGW64:/c/Users/zhngtr/Desktop/Git&GitLab??/wiki-repo

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo
$ git init
Initialized empty Git repository in C:/Users/zhngtr/Desktop/Git&GitLab手册/wiki-repo/.git/

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$ ls .git
config description HEAD hooks/ info/ objects/ refs/

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$ ll -lh .git
total 7.0K
-rw-r--r-- 1 zhngtr 197121 130 九月 27 16:41 config
-rw-r--r-- 1 zhngtr 197121 73 九月 27 16:41 description
-rw-r--r-- 1 zhngtr 197121 23 九月 27 16:41 HEAD
drwxr-xr-x 1 zhngtr 197121 0 九月 27 16:41 hooks/
drwxr-xr-x 1 zhngtr 197121 0 九月 27 16:41 info/
drwxr-xr-x 1 zhngtr 197121 0 九月 27 16:41 objects/
drwxr-xr-x 1 zhngtr 197121 0 九月 27 16:41 refs/

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$
```

在一个空文件夹中运行，init 命令 会生成一个 .git的文件夹，把这个文件夹初始化为git的仓库，.git文件夹中存放着仓库的配置信息。

4.2 设置签名

注：没有签名是能够进行操作的。

➤ 形式

用户名：tom

Email 地址：goodMorning@atguigu.com

➤ 作用：区分不同开发人员的身份

➤ 辨析：这里设置的签名和登录远程库(代码托管中心)的账号、密码没有任何关系。

➤ 命令

项目级别/仓库级别：仅在当前本地库范围内有效

git **config** user.name namedemo

git **config** user.email namedemo_pro@gitlab.com

信息保存位置：./.git/config 文件

```

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$ git config user.name namedemo

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$ git config user.email namedemo_pro@gitlab.com

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    logallrefupdates = true
    symlinks = false
    ignorecase = true
[user]
    name = namedemo
    email = namedemo_pro@gitlab.com

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$

```

系统用户级别：登录当前操作系统的用户范围

git config --global user.name namedemo

git config --global user.email namedemo_pro@gitlab.com

信息保存位置：~/.gitconfig 文件

```

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$ git config --global user.name namedemo

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$ cat ~/.gitconfig
[filter "lfs"]
    process = git-lfs filter-process
    clean = git-lfs clean -- %f
    smudge = git-lfs smudge -- %f
    required = true
[user]
    name = namedemo
[user]
    email = namedemo_pro@gitlab.com
[gui]
    recentrepo = C:/Users/zhngtr/IdeaProjects/gitIdeaRespository
[http]
[https]
[https]
    proxy = http://10.5.3.9:80
[http]
    proxy = http://10.5.3.9:80

zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册/wiki-repo (master)
$

```

级别优先级

- ◆ 就近原则：项目级别优先于系统用户级别，二者都有时采用项目级别的签名
- ◆ 如果只有系统用户级别的签名，就以系统用户级别的签名为准
- ◆ 二者都没有不允许

4.3 基本操作

常用命令：参考 Git Cheat Sheet

4.3.1 状态查看

```
git status
```

查看工作区、暂存区状态

4.3.2 添加

```
git add [file name]
```

将工作区的“新建/修改”添加到暂存区

4.3.3 提交

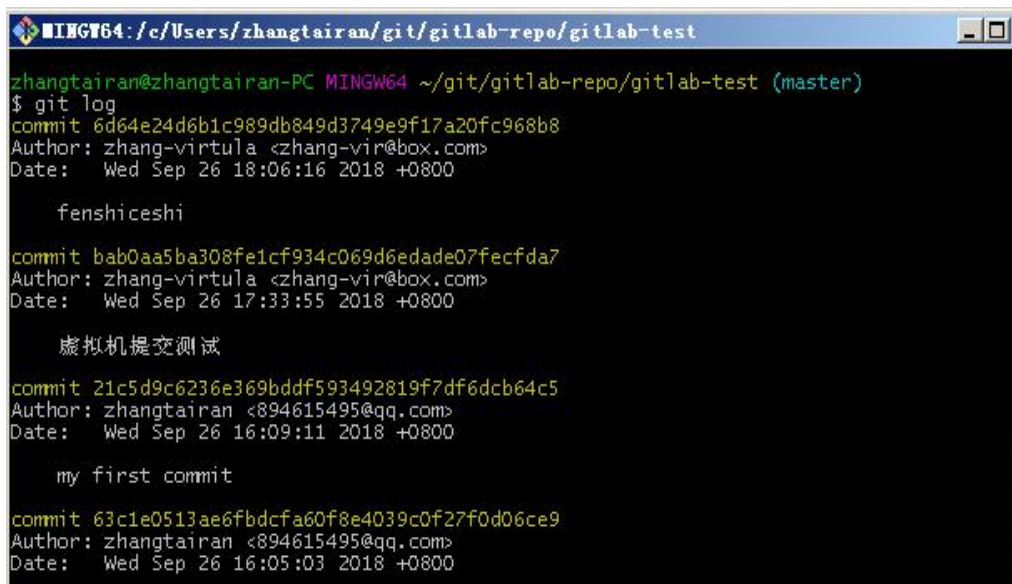
```
git commit -m "commit message" [file name]
```

将暂存区的内容提交到本地库

4.3.4 查看历史记录

```
git log
```

显示从最近到最远的提交日志



```
MINGW64: /c/Users/zhangtairan/git/gitlab-repo/gitlab-test

zhangtairan@zhangtairan-PC MINGW64 ~/git/gitlab-repo/gitlab-test (master)
$ git log
commit 6d64e24d6b1c989db849d3749e9f17a20fc968b8
Author: zhang-virtula <zhang-vir@box.com>
Date:   Wed Sep 26 18:06:16 2018 +0800

    fenshiceshi

commit bab0aa5ba308fe1cf934c069d6edade07fecfda7
Author: zhang-virtula <zhang-vir@box.com>
Date:   Wed Sep 26 17:33:55 2018 +0800

    虚拟机提交测试

commit 21c5d9c6236e369bddf593492819f7df6dcb64c5
Author: zhangtairan <894615495@qq.com>
Date:   Wed Sep 26 16:09:11 2018 +0800

    my first commit

commit 63c1e0513ae6fbdcfa60f8e4039c0f27f0d06ce9
Author: zhangtairan <894615495@qq.com>
Date:   Wed Sep 26 16:05:03 2018 +0800
```


多屏显示控制方式:

空格向下翻页

b 向上翻页

q 退出

git log --pretty=oneline

```
zhangtairan@zhangtairan-PC MINGW64 ~/git/gitlab-repo/gitlab-test (master)
$ git log --pretty=oneline
6d64e24d6b1c989db849d3749e9f17a20fc968b8 fenshiceshi
bab0aa5ba308fe1cf934c069d6edade07fecfda7 虚拟机提交测试
21c5d9c6236e369bddf593492819f7df6dcb64c5 my first commit
63c1e0513ae6fbdcfa60f8e4039c0f27f0d06ce9 xinzeng
```

git reflog (查看所有的版本)

```
zhangtairan@zhangtairan-PC MINGW64 ~/git/gitlab-repo/gitlab-test (master)
$ git reflog
6d64e24 HEAD@{0}: commit: fenshiceshi
bab0aa5 HEAD@{1}: commit: 虚拟机提交测试
21c5d9c HEAD@{2}: clone: from http://10.5.96.13/gitlab/root/gitlab-test.git
```

```
zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册 (master)
$ git reflog
3dfdb47 HEAD@{0}: commit: 修改后添加 oneline
29e0a90 HEAD@{1}: commit: readme push
141b0aa HEAD@{2}: commit (initial): 初始化提交,文档文件
```

推荐使用匿名功能, 简化查询日志的操作:

```
git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<an>%Creset' --abbrev-commit"
```

然后可以使用 git lg 查询带颜色标注的日志信息

```
zhangtairan@zhangtairan-PC MINGW64 ~/git/gitlab-repo/gitlab-test (master)
$ git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<an>%Creset' --abbrev-commit"

zhangtairan@zhangtairan-PC MINGW64 ~/git/gitlab-repo/gitlab-test (master)
$ git lg
* efd6183 - (HEAD -> master) test reflog (4 hours ago) <zhang-virtula>
* 6d64e24 - fenshiceshi (2 days ago) <zhang-virtula>
* bab0aa5 - 虚拟机提交测试 (2 days ago) <zhang-virtula>
* 21c5d9c - my first commit (2 days ago) <zhangtairan>
* 63c1e05 - xinzeng (2 days ago) <zhangtairan>
```

4.3.5 前进后退 (切换版本)

版本信息


```
zhngtr@zhngtr-PC MINGW64 ~/Desktop/Git&GitLab手册 (master)
$ git reflog
d940d4b HEAD@{0}: commit: 提交一些问题
3dfdb47 HEAD@{1}: commit: 修改后添加 oneline
29e0a90 HEAD@{2}: commit: readme push
141b0aa HEAD@{3}: commit (initial): 初始化提交,文档文件
```

- 基于索引值操作[推荐]
 - `git reset --hard [局部索引值]`
 - `git reset --hard 141b0aa`
- 使用^符号：只能后退
 - `git reset --hard HEAD^`
 - 注：一个^表示后退一步，n 个表示后退 n 步
- 使用~符号：只能后退
 - `git reset --hard HEAD~n`
 - 注：表示后退 n 步

reset 命令的三个参数对比

- soft 参数
 - 仅仅在本地库移动 HEAD 指针
- mixed 参数
 - 在本地库移动 HEAD 指针
 - 重置暂存区
- hard 参数
 - 在本地库移动 HEAD 指针
 - 重置暂存区
 - 重置工作区

4.3.6 删除文件并找回

前提：删除前，文件存在时的状态提交到了本地库。

操作：`git reset --hard [指针位置]`

删除操作已经提交到本地库：指针位置指向历史记录

删除操作尚未提交到本地库：指针位置使用 HEAD

4.3.6.1 怎样删除文件

本地工作区删除文件后，按照正常的操作流程，`add commit push`后，就会删除文件

4.3.7 比较文件差异

`git diff [文件名]`

将工作区中的文件和暂存区进行比较

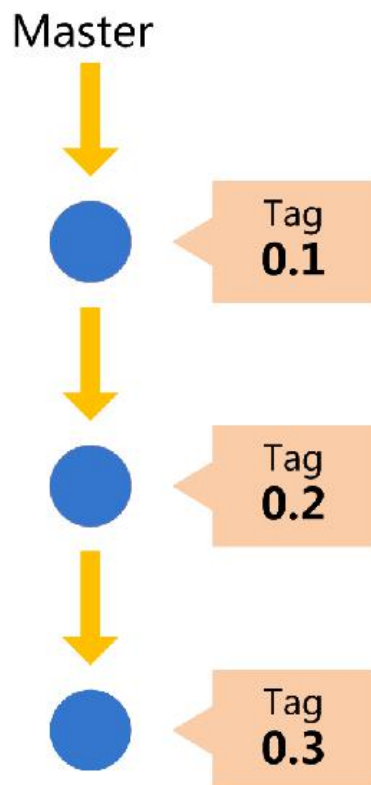
`git diff` [本地库中历史版本] [文件名]
将工作区中的文件和本地库历史记录比较
不带文件名比较多个文件

4. 4分支管理（重点）

4.4.1 主分支master

一、主分支Master

首先，代码库应该有一个、且仅有一个主分支。所有提供给用户使用的正式版本，都在这个主分支上发布。

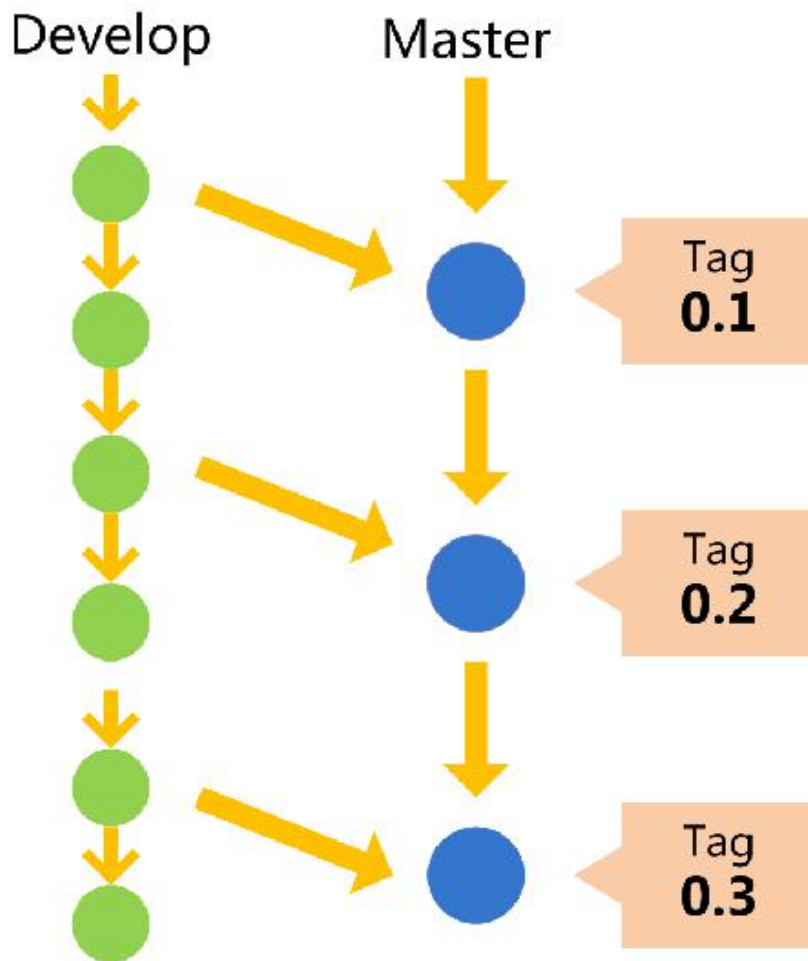


Git主分支的名字，默认叫做Master。它是自动建立的，版本库初始化以后，默认就是在主分支在进行开发。

4.4.2 开发分支Develop

二、开发分支Develop

主分支只用来发布重大版本，日常开发应该在另一条分支上完成。我们把开发用的分支，叫做Develop。



这个分支可以用来生成代码的最新隔夜版本（nightly）。如果想正式对外发布，就在Master分支上，对Develop分支进行"合并"（merge）。

Git创建Develop分支的命令：

```
git checkout -b develop master
```

将Develop分支发布到Master分支的命令：

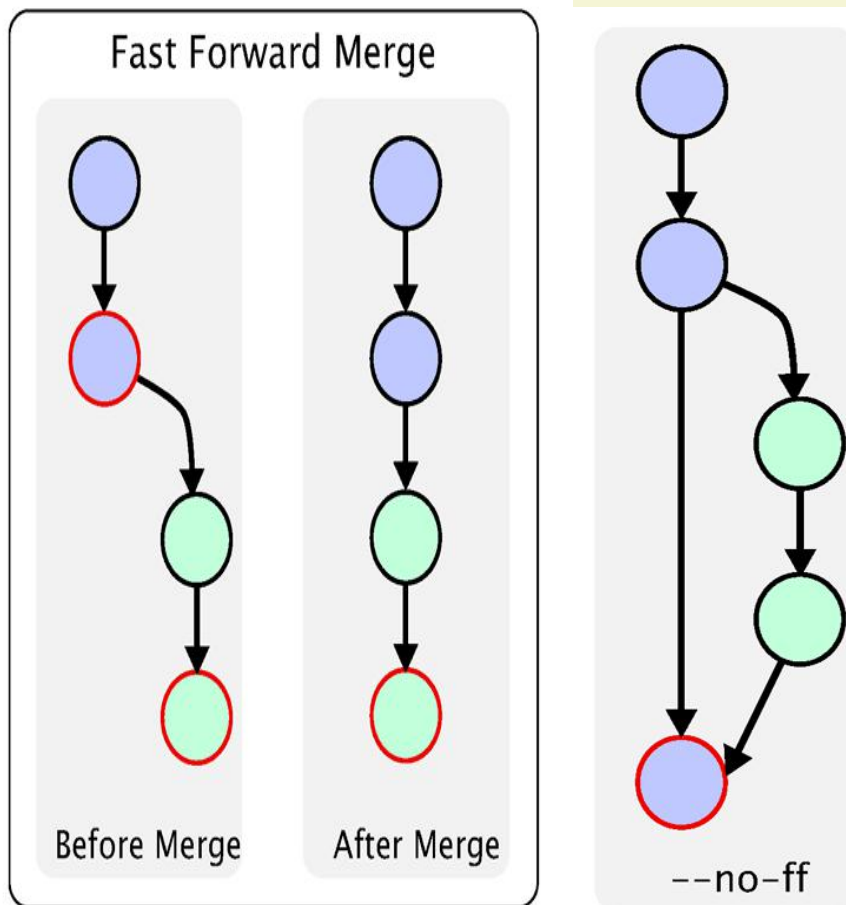
```
# 切换到Master分支
```

```
git checkout master
```

```
# 对Develop分支进行合并
```

```
git merge --no-ff develop
```

这里稍微解释一下,上一条命令的--no-ff参数是什么意思。默认情况下, Git执行"快进式合并" (fast-forward merge), 会直接将Master分支指向Develop分支。



使用--no-ff参数后, 会执行正常合并, 在Master分支上生成一个新节点。为了保证版本演进的清晰, 我们希望采用这种做法。关于合并的更多解释, 请参考Benjamin Sandofsky的[《Understanding the Git Workflow》](#)。

4.4.3 临时分支

三、临时性分支

前面讲到版本库的两条主要分支：**Master**和**Develop**。前者用于正式发布，后者用于日常开发。其实，常设分支只需要这两条就够了，不需要其他了。

但是，除了常设分支以外，还有一些临时性分支，用于应对一些特定目的的版本开发。临时性分支主要有三种：

- * 功能（**feature**）分支

- * 预发布（**release**）分支

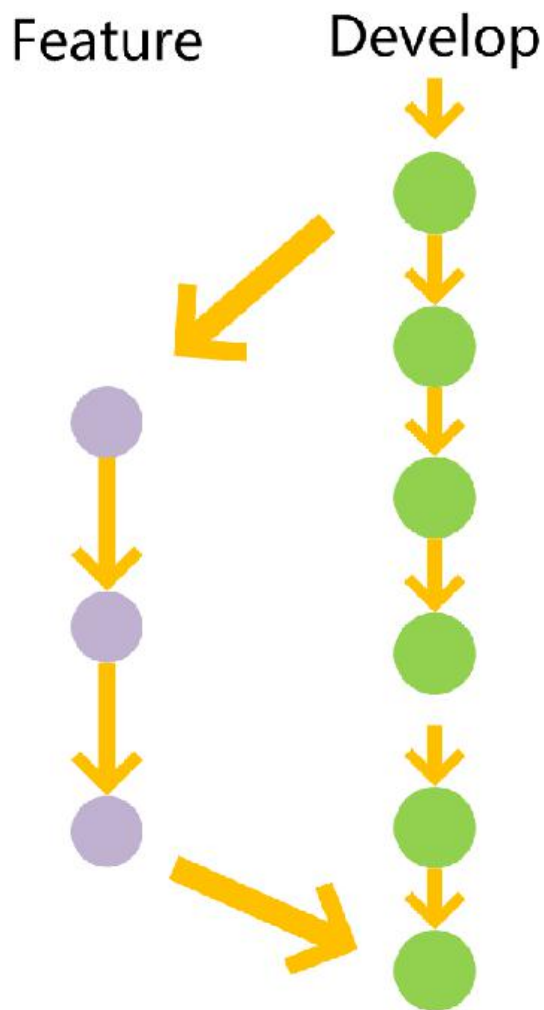
- * 修补**bug**（**fixbug**）分支

这三种分支都属于临时性需要，使用完以后，应该删除，使得代码库的常设分支始终只有**Master**和**Develop**。

四、 功能分支

接下来，一个个来看这三种"临时性分支"。

第一种是功能分支，它是为了开发某种特定功能，从**Develop**分支上面分出来的。开发完成后，要再并入**Develop**。



功能分支的名字，可以采用feature-*的形式命名。

创建一个功能分支：

```
git checkout -b feature-x develop
```

开发完成后，将功能分支合并到develop分支：

```
git checkout develop
```

```
git merge --no-ff feature-x
```

删除feature分支：

```
git branch -d feature-x
```


4.4.4 预发布分支--(准生产分支)

五、预发布分支

第二种是预发布分支，它是指发布正式版本之前（即合并到Master分支之前），我们可能需要有一个预发布的版本进行测试。

预发布分支是从Develop分支上面分出来的，预发布结束后，必须合并进Develop和Master分支。它的命名，可以采用release-*的形式。

创建一个预发布分支：

```
git checkout -b release-1.2 develop
```

确认没有问题后，合并到master分支：

```
git checkout master
```

```
git merge --no-ff release-1.2
```

```
# 对合并生成的新节点，做一个标签
```

```
git tag -a 1.2
```

再合并到develop分支：

```
git checkout develop
```

```
git merge --no-ff release-1.2
```

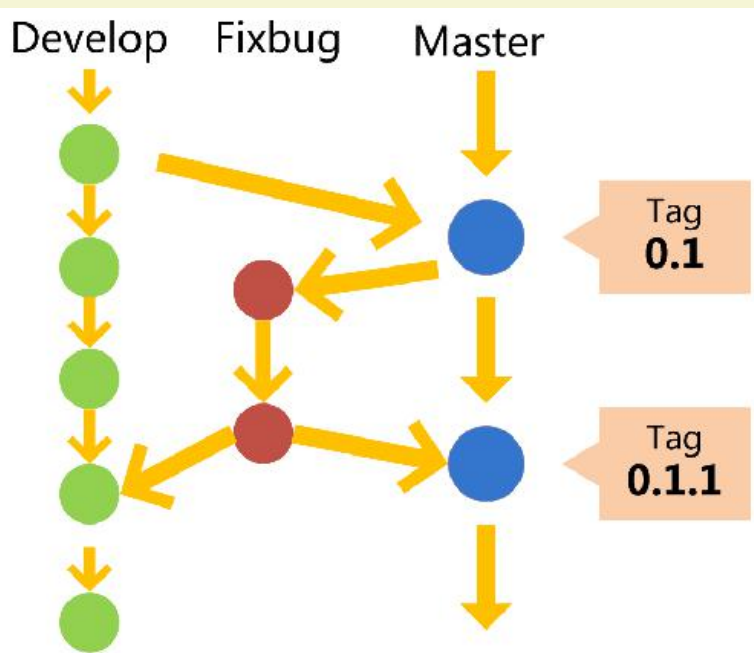
最后，删除预发布分支：

```
git branch -d release-1.2
```

4.4.5 修改bug分支--(准生产分支)

最后一种是**修补bug分支**。软件正式发布以后，难免会出现bug。这时就需要创建一个分支，进行bug修补。

修补bug分支是从Master分支上面分出来的。修补结束以后，再合并进Master和Develop分支。它的命名，可以采用fixbug-*的形式。



创建一个修补bug分支：

```
git checkout -b fixbug-0.1 master
```

修补结束后，合并到master分支：

```
git checkout master
```

```
git merge --no-ff fixbug-0.1
```

```
git tag -a 0.1.1
```

再合并到develop分支：

```
git checkout develop
```

```
git merge --no-ff fixbug-0.1
```

最后，删除"修补bug分支":

```
git branch -d fixbug-0.1
```

4.5 分支管理流程（重点）

4.5.1 官方文档

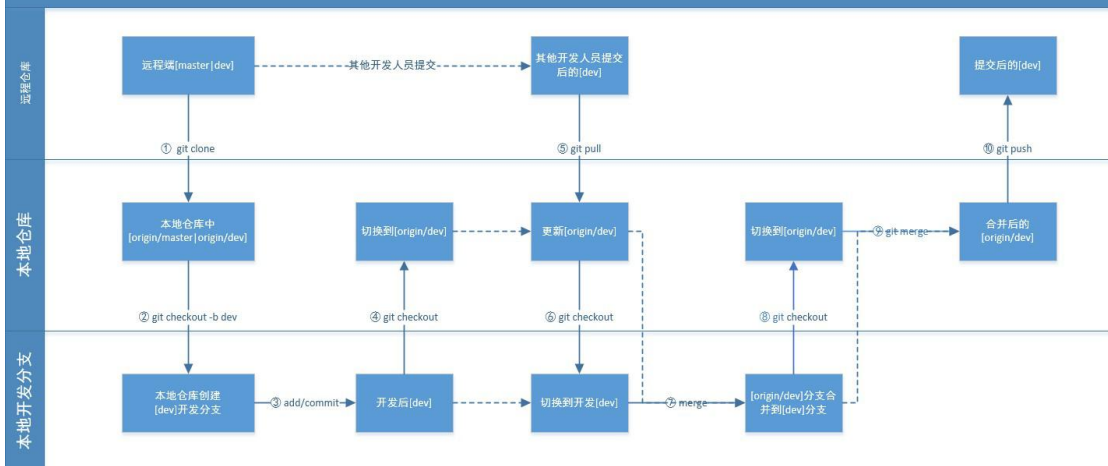
<http://git.oschina.net/progit/3-Git-%E5%88%86%E6%94%AF.html>

4.5.2 整理的操作步骤

开发人员git操作步骤：

- 1.git clone 把远程dev上的代码克隆到本地（origin/dev）
- 2.git checkout -b dev 在本地创建一个dev分支，在这个分支上修改代码
- 3.dev分支上add和commit代码
- 4.切换到origin/dev分支上
- 5.git pull 把远程dev分支上的代码更新下来。因为在修改代码期间，或许有人提交了代码，需要把别人提交的代码更新下来，我们提交时才能保证不会覆盖掉别人的代码。
- 6.切换到dev分支
- 7.把origin/dev分支合并到dev分支，如果有冲突解决冲突。解决完冲突需要add和commit提交代码
- 8.切换到origin/dev分支上
- 9.把dev分支合并到origin/dev分支上
- 10.git push origin 把origin/dev分支提交到远程dev上

Git开发操作流程



5 eclipse中插件使用(egit)

5.0 参考

教程

<https://blog.csdn.net/mengxiangxingdong/article/details/78926336>

官方文档

<http://git.oschina.net/progit/3-Git-%E5%88%86%E6%94%AF.html>

5.1 配置

插件安装省略,和一般插件安装方法相同,且eclipse自带git插件

Eclipse中的位置

Window->Preferences->Team->Git->Configuration

5.1.1 配置git的签名(名字全拼和邮箱或向管理人员索要)

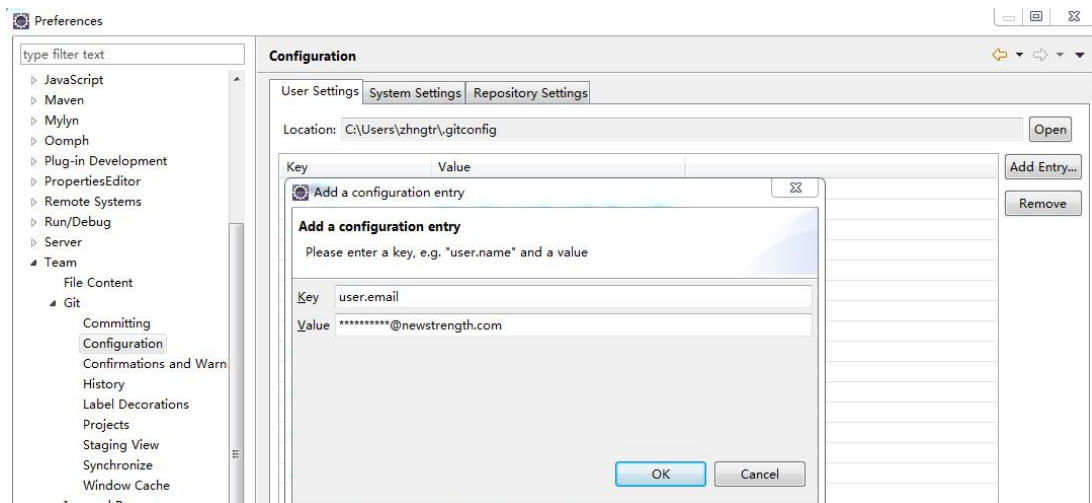
点击Add Entry, 在弹出框里面输入key和value的值

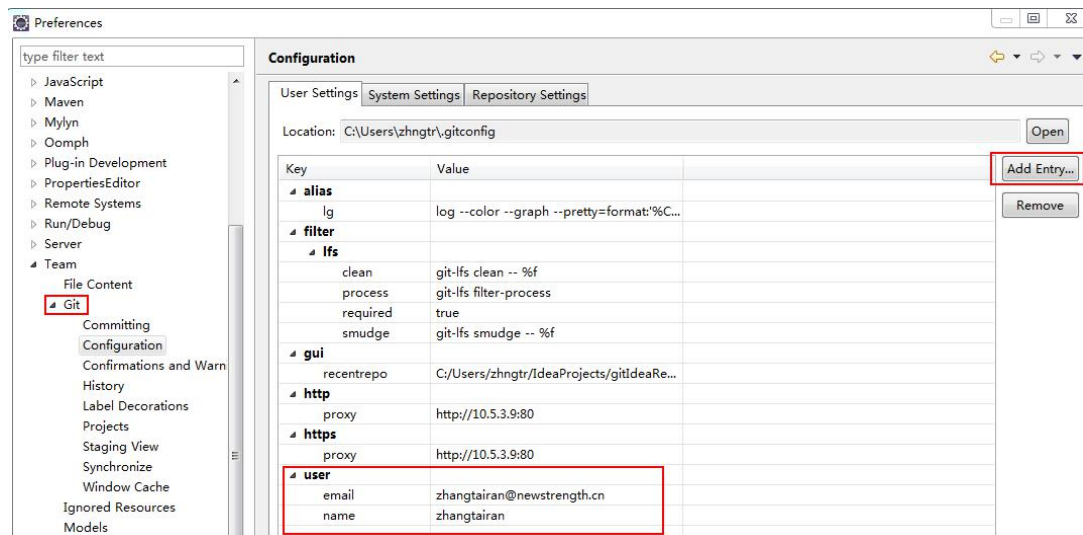
名字填写 key: user.name

value:你的名字。

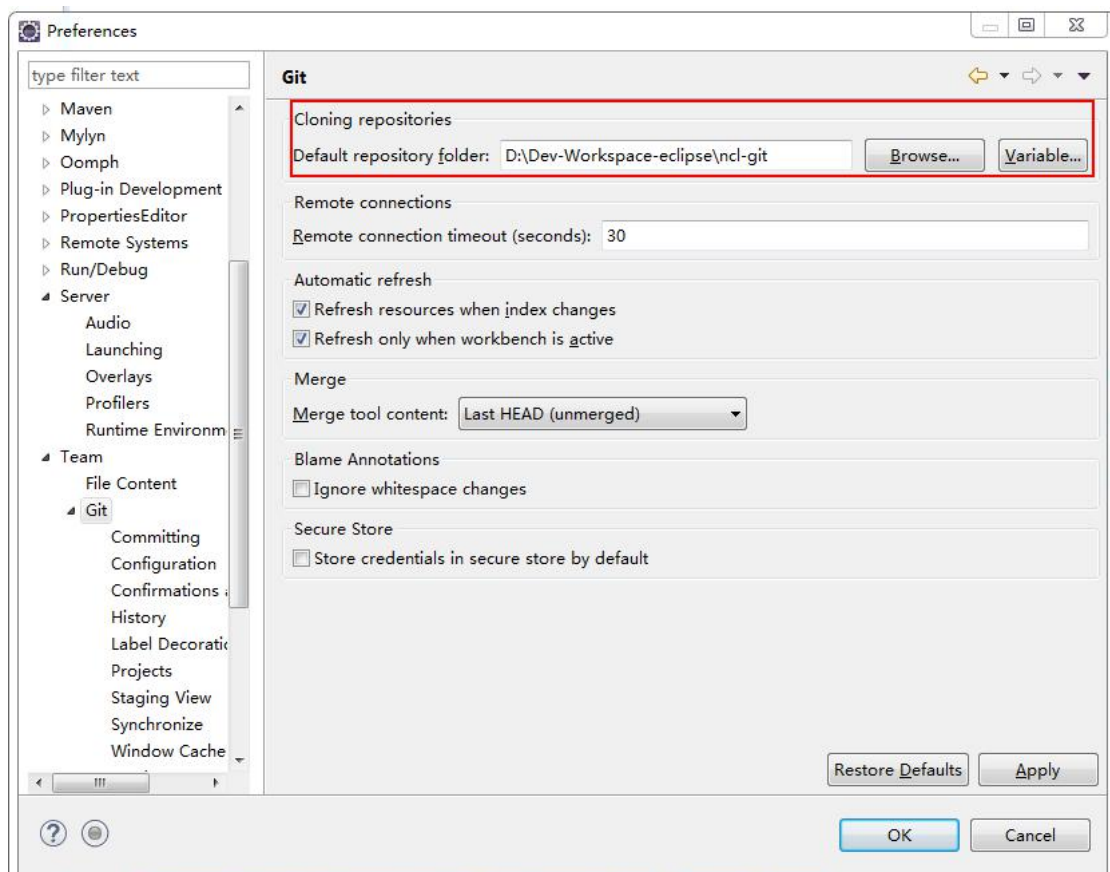
邮箱填写 key: user.email

value:你的邮箱账号



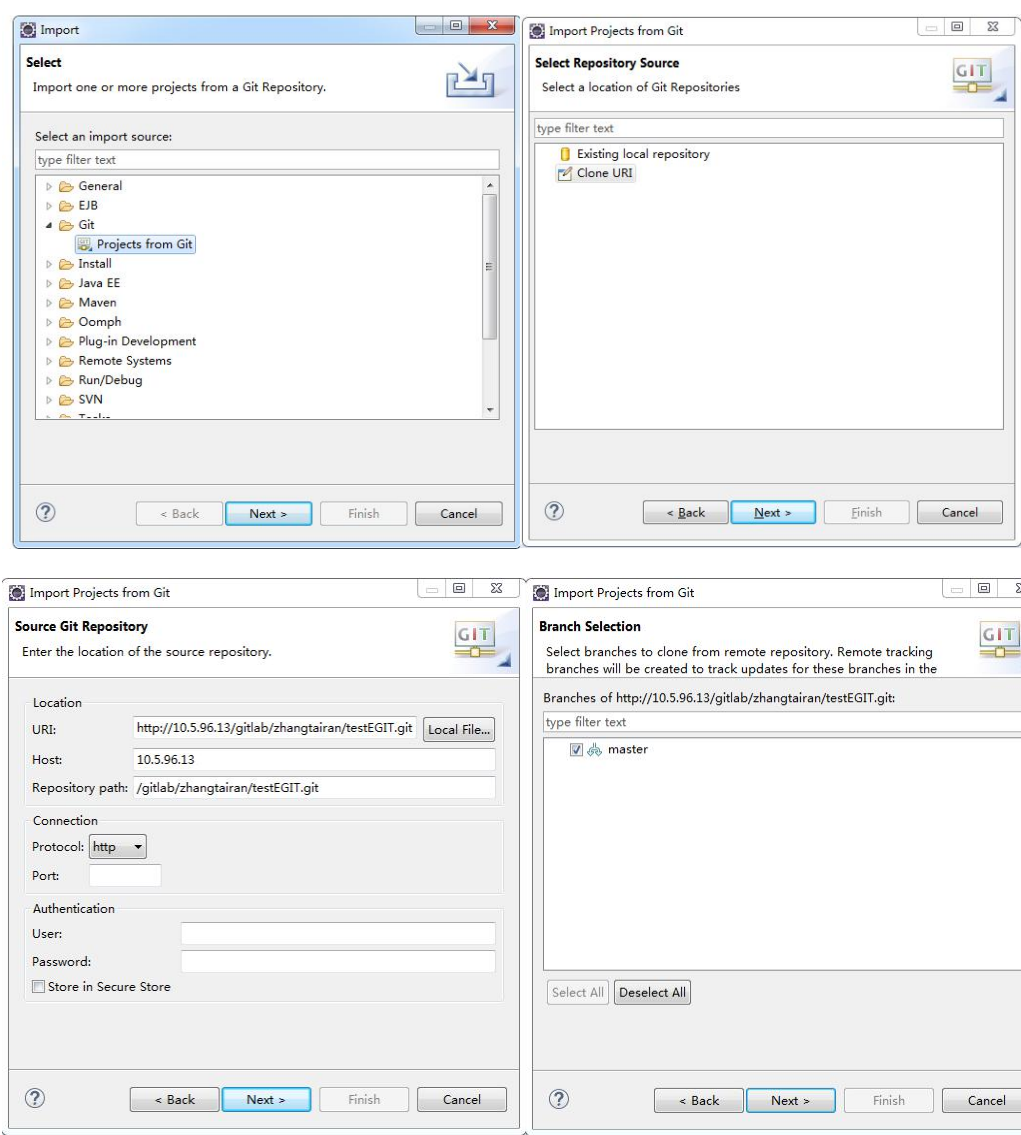
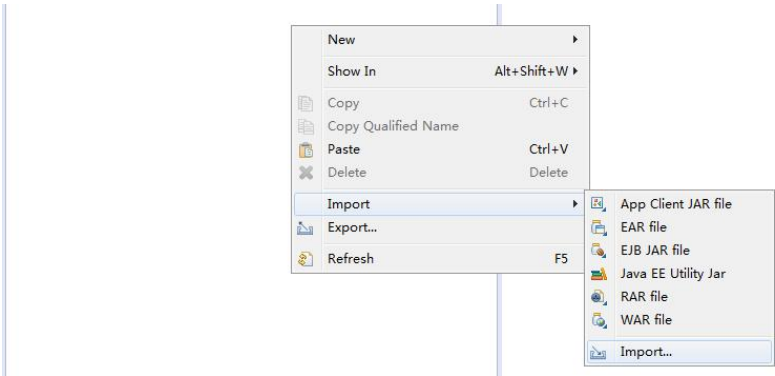


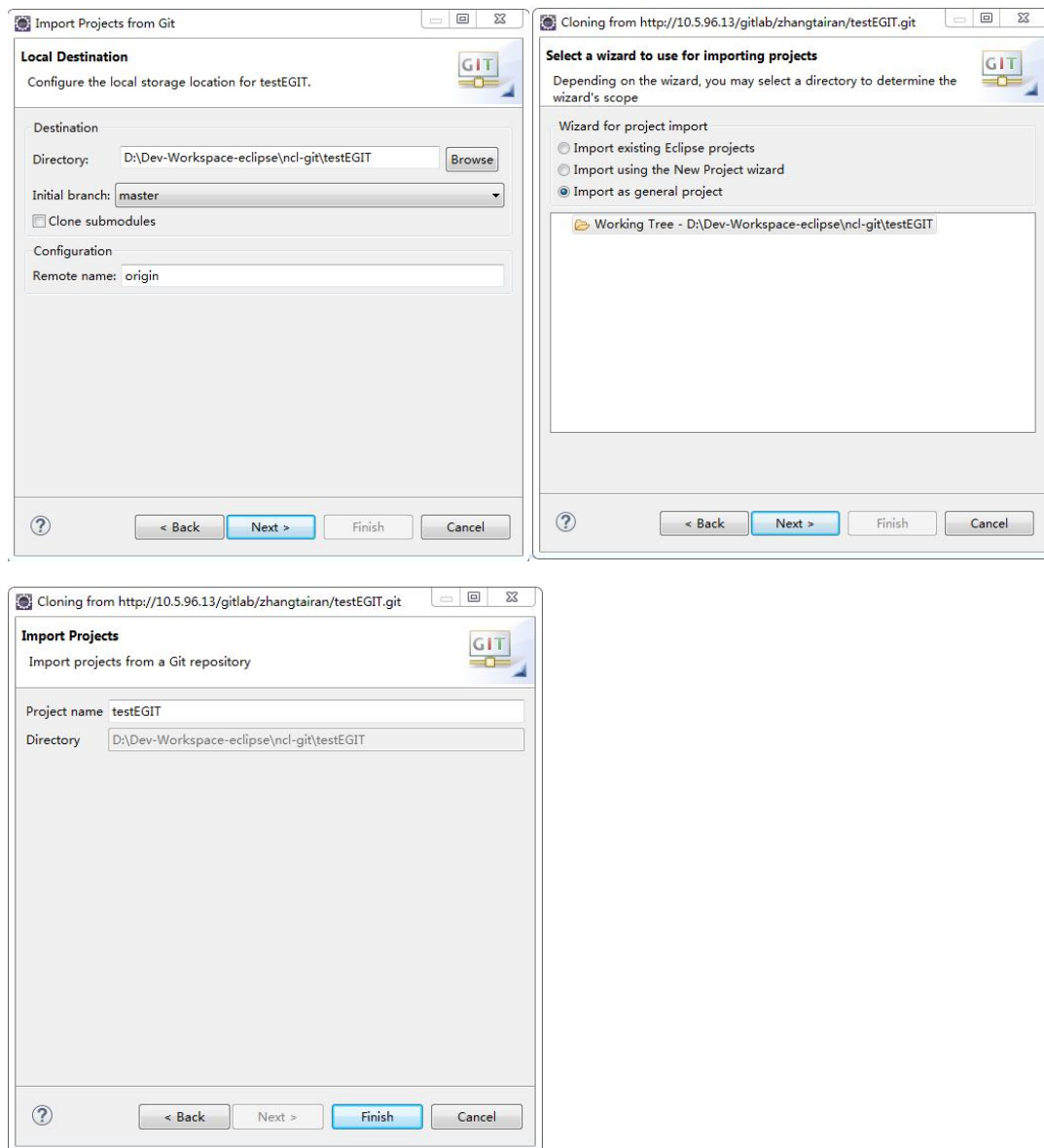
5.1.2 配置git的仓库地址(推荐选到工作空间)



5.2 从服务器clone项目

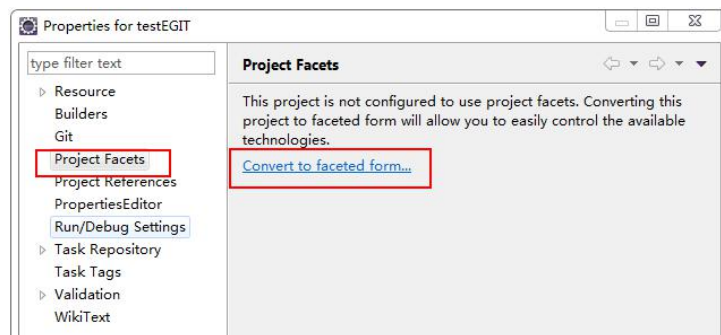
5.2.1 导入项目





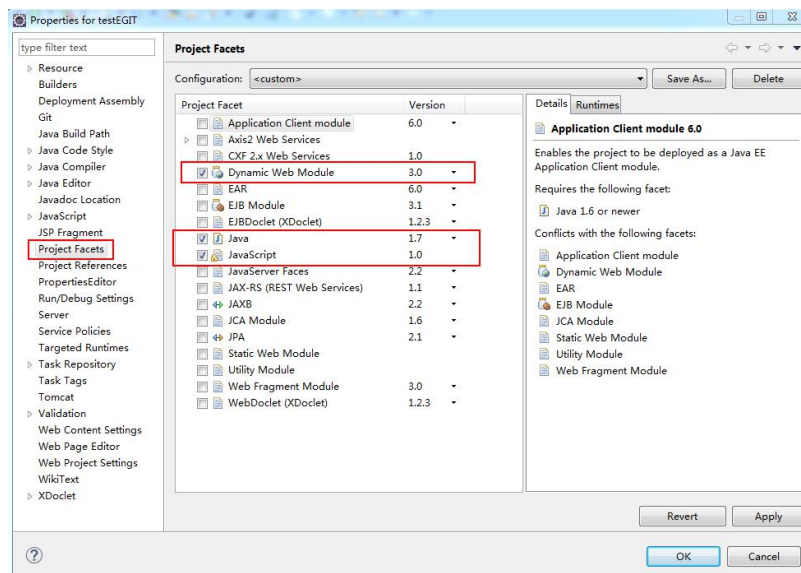
5.2.2 配置项目特性

项目上右键properties, 配置项目的特性(是否为web项目,是否java项目,是否javascript)



在Eclipse中, 新建的 Java Project 都有一个默认的 java facet, 那么

Eclipse 就只提供 JavaSE 项目支持,当你需要将该项目升级为 java web 项目时,可以为项目添加 Dynamic Web Module (就是一个支持Web的facet) **fe和RBAC是2.4版本**,这样 eclipse 就会将项目结构调整为带有 WebContent or WebRoot 目录的标准结构且添加 deployment descriptor (web.xml) 并调整默认的 classpath,同时,如果你要用到 javascript,可以对应地添加 javascript facet,这样 eclipse 就会为项目添加 javascript 相关的支持(构建、校验、提示等等),如果你的项目用到了 hibernate,则可以添加 jpa facet 来让 eclipse 提供对应的功能支持等等。



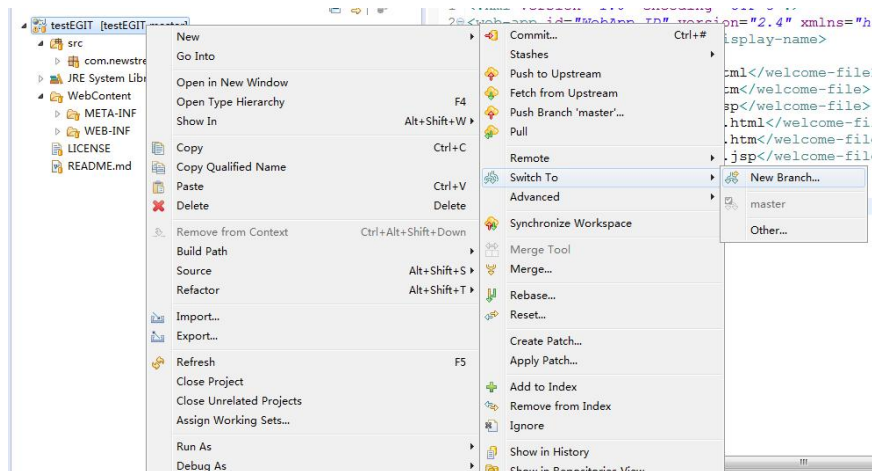
5.3 使用EGIT (结合4.5说明)

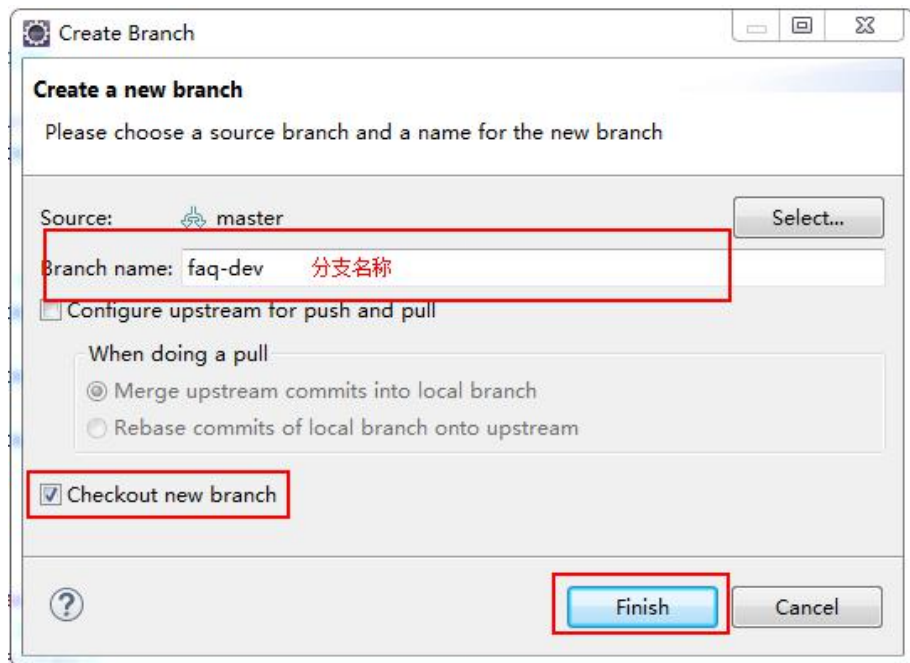
开发人员git操作步骤:

1.git clone 把远程 dev 上的代码克隆到本地 (origin/dev---->>dev)

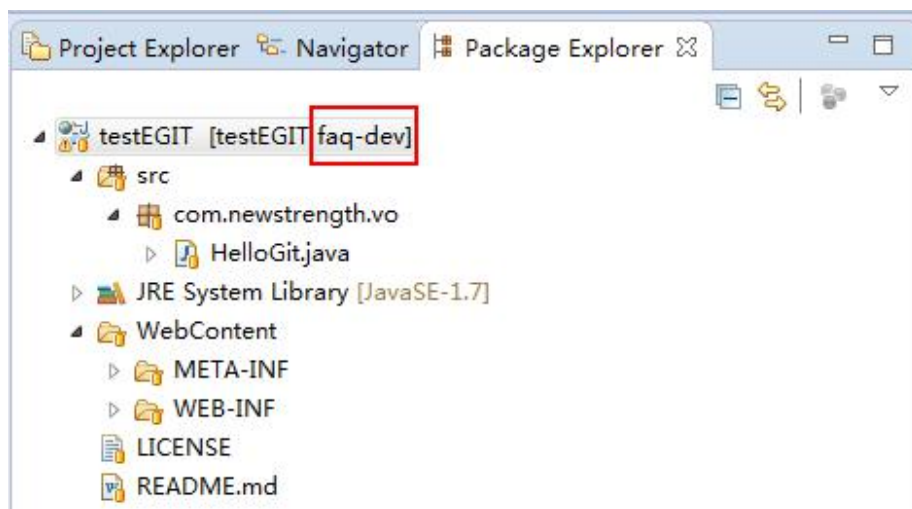
clone参考上方的5.2

2.git checkout -b faq-dev在本地创建一个xxx-dev分支,在这个分支上修改或开发功能

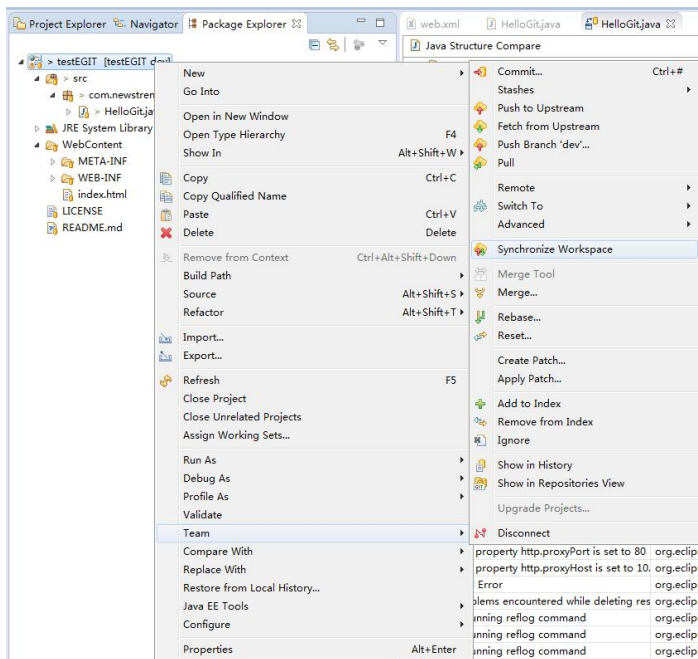




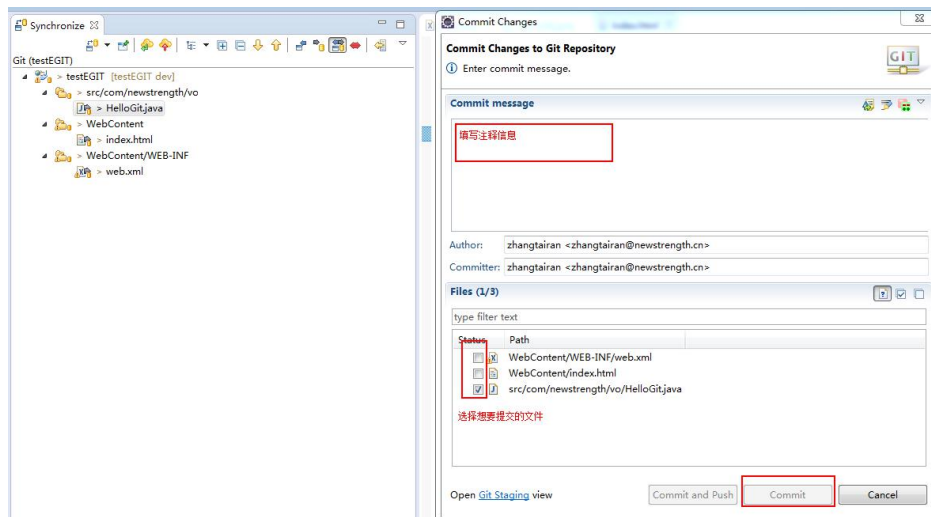
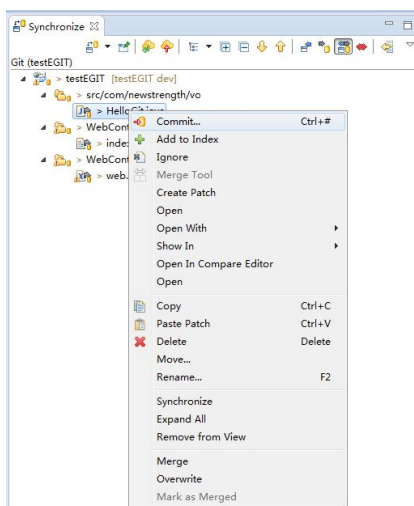
显示切换分支的名称



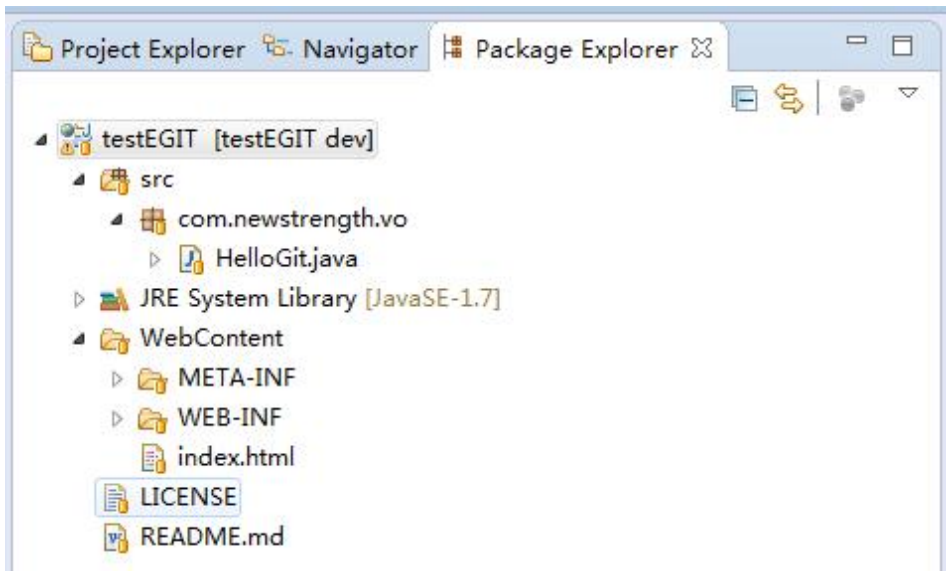
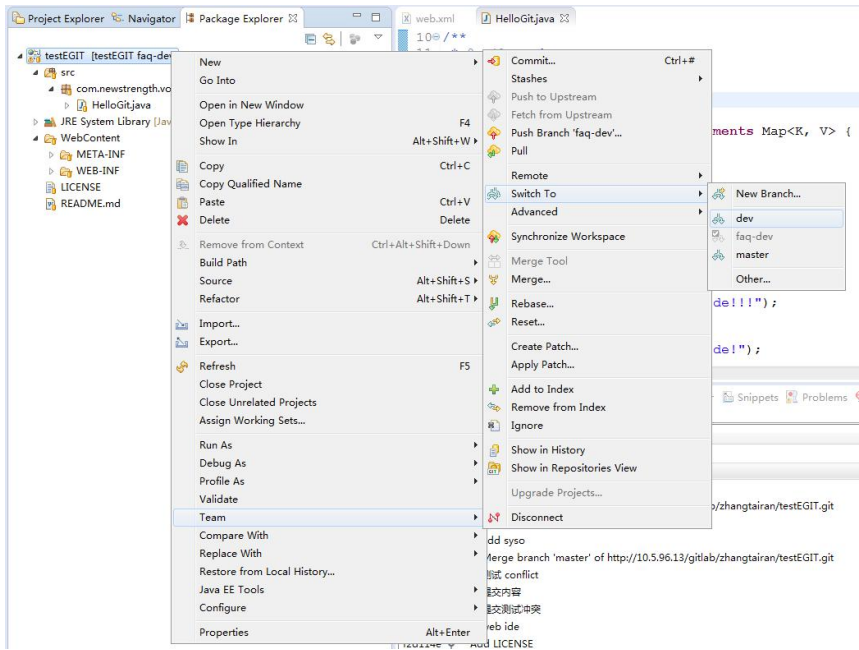
3.faq-dev分支上add和commit代码



对比界面查询有没有冲突,然后提交

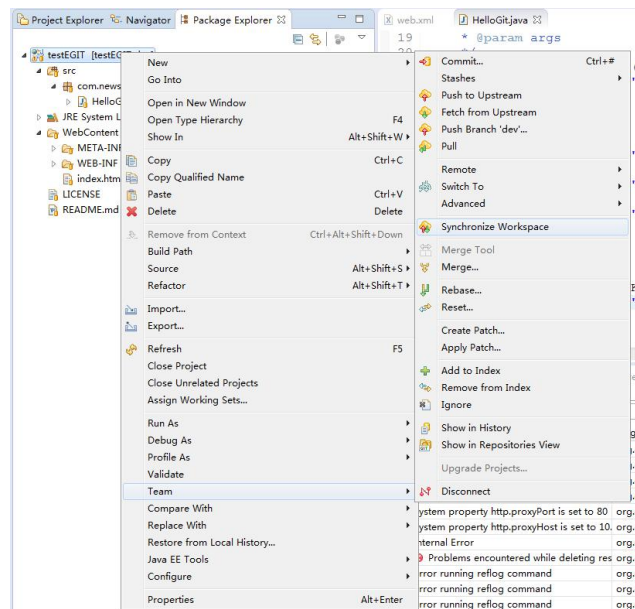


4.切换到origin/dev(dev)分支上



5.git pull 把远程dev分支上的代码更新下来。因为在修改代码期间，或许有人提交了代码，需要把别人提交的代码更新下来，我们提交时才能保证不会覆盖掉别人的代码。

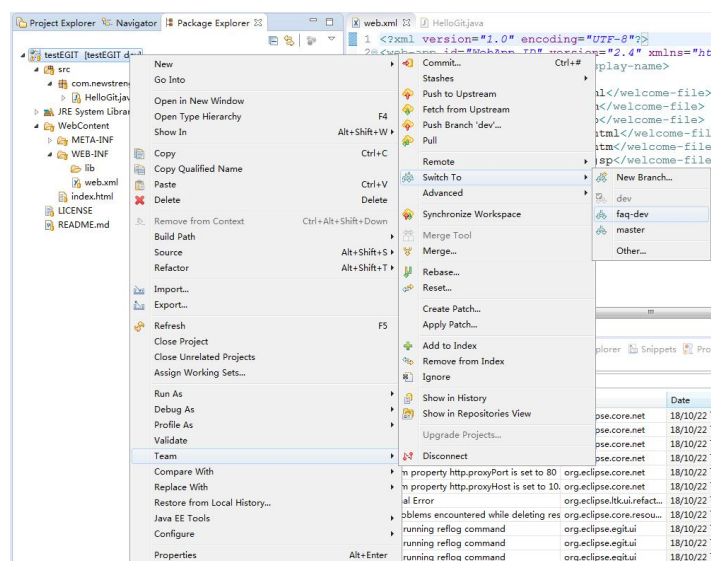
5.1 对比和服务器中代码的区别



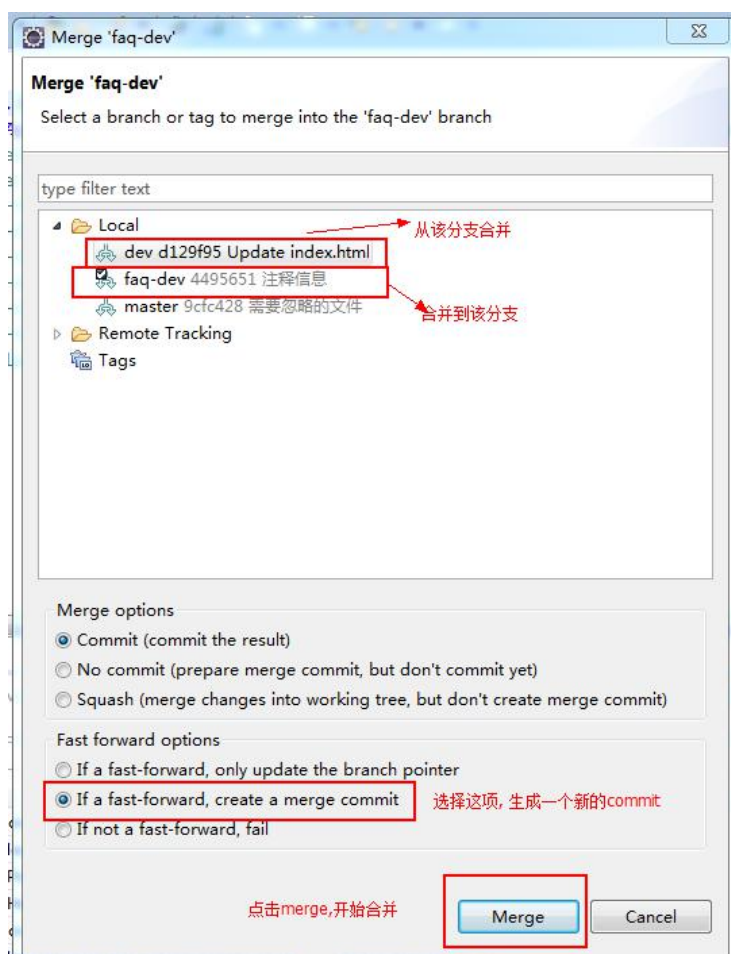
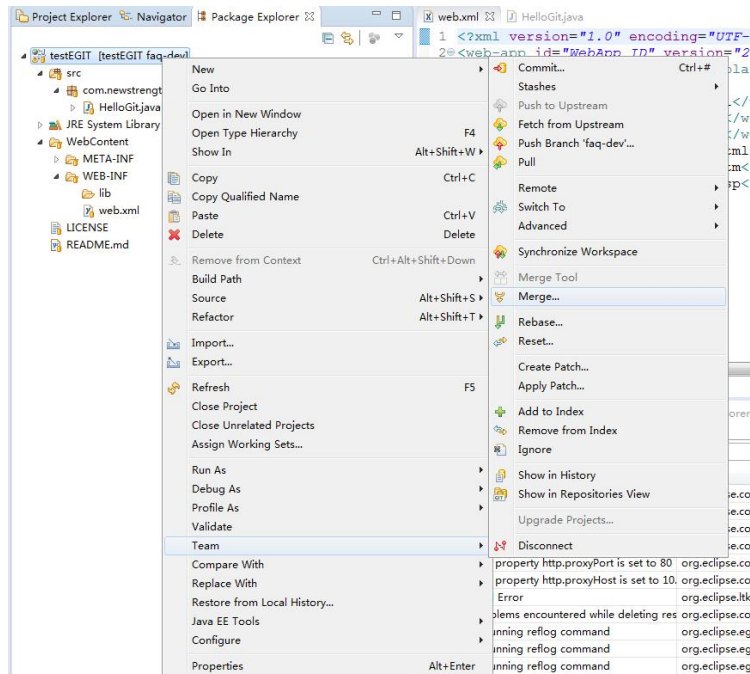
对比界面和 svn中的类似



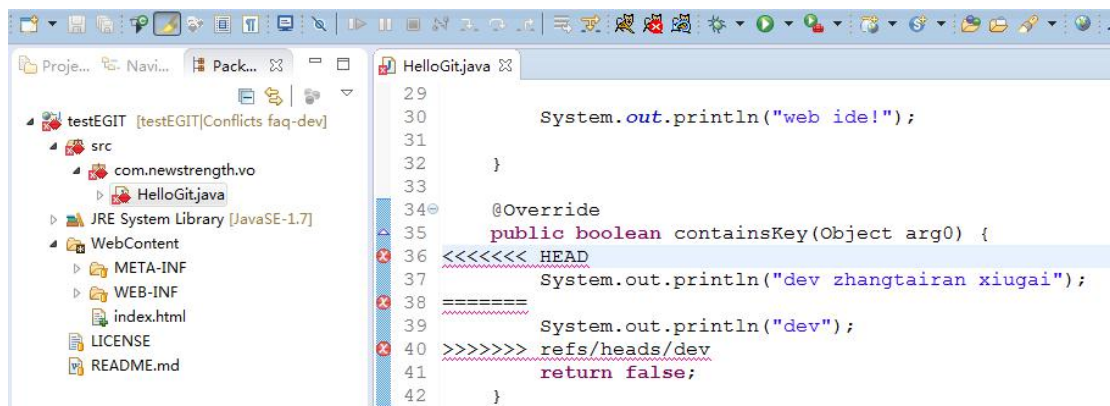
6.切换到faq-dev分支



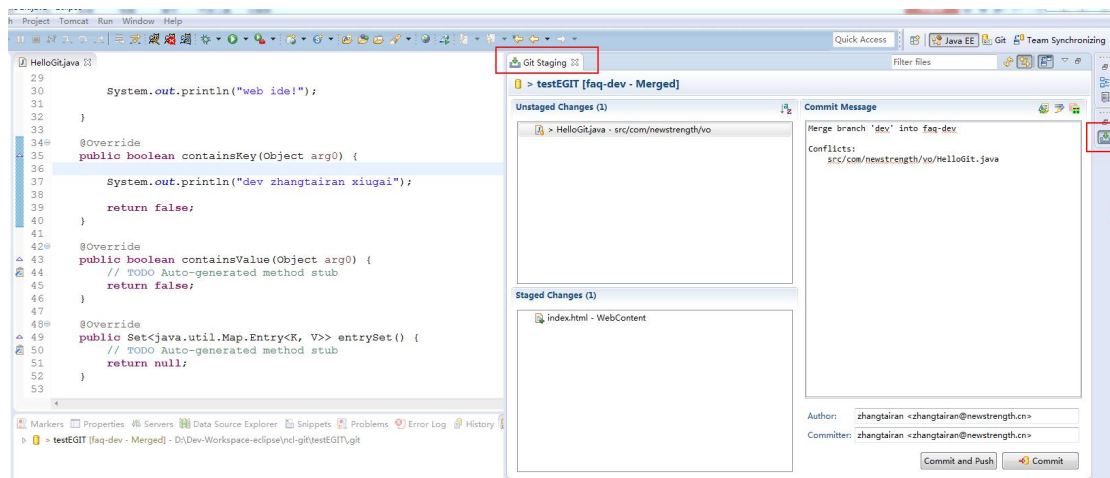
7.把origin/dev分支合并到dev分支，如果有冲突解决冲突。解决完冲突需要add和commit提交代码



若存在冲突, 需要处理冲突, “=====” 上方为本地修改, 下方为 远程内容



处理完冲突后, 点击 Git Staging, 把冲突文件添加到staged区然后编写注释, 提交文件

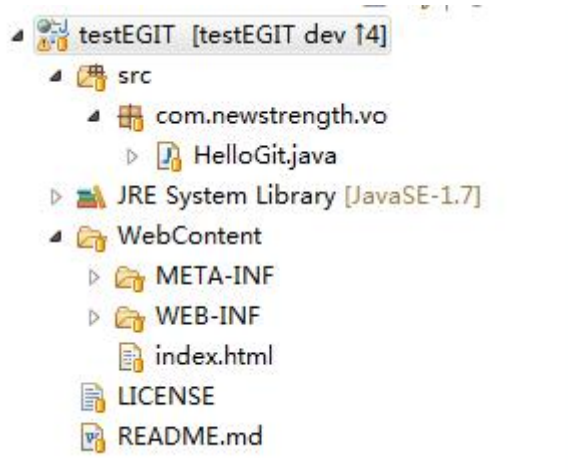


8. 切换到origin/dev (dev)分支上

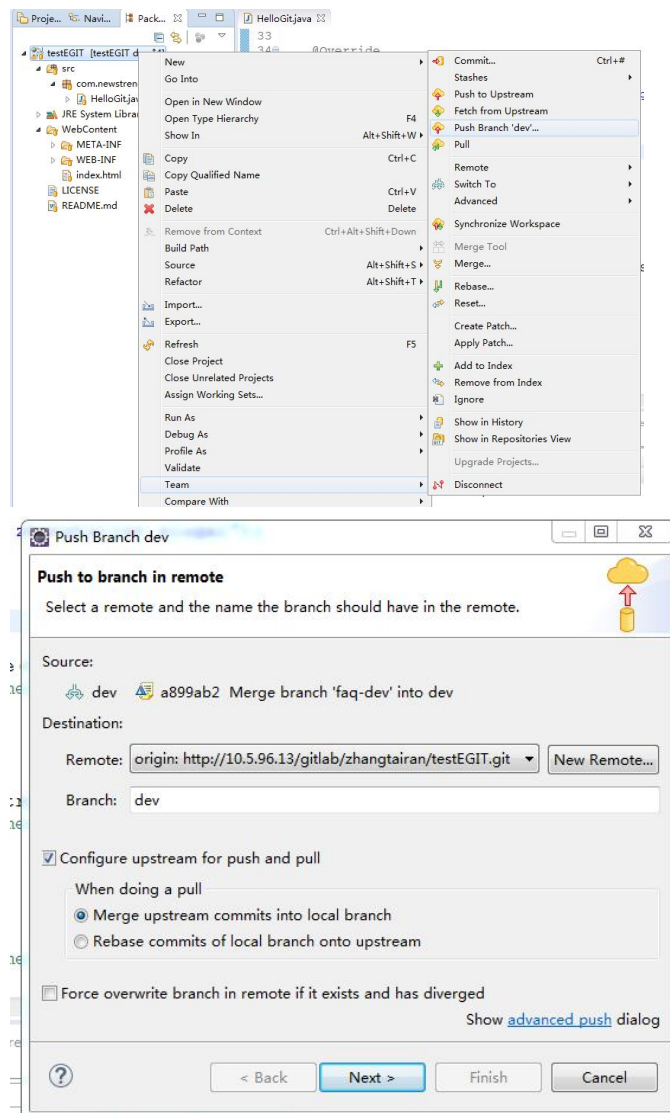


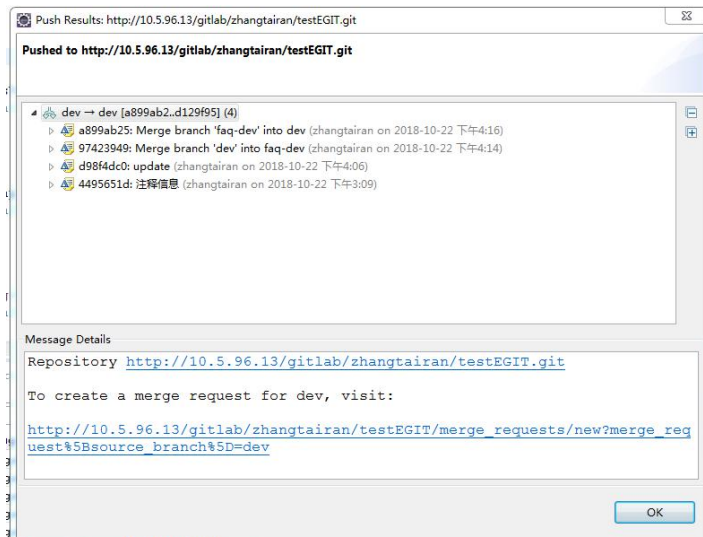
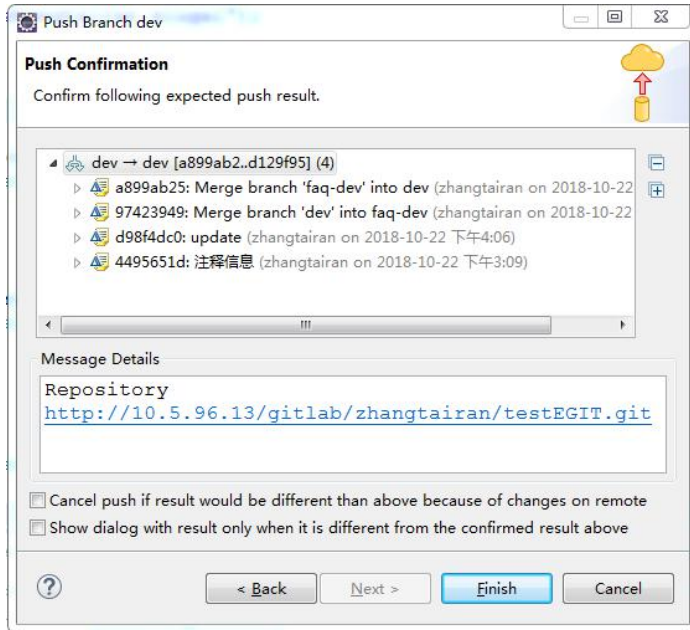
9.把dev分支合并到origin/dev分支上

操作步骤同 7



10.git push origin 把origin/dev分支提交到远程dev上





6 总结

本文档，已经包含了基本的开发操作。可以进行正常的开发操作。如果了解更过，可以查看下方提供的资料

6.1 Pro Git (中文版)

<http://git.oschina.net/progit/>

6.2 廖雪峰Git教程

<https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000>

6.3 阮一峰 Git分支管理策略

<http://www.ruanyifeng.com/blog/2012/07/git.html>

6.4 阮一峰 Git远程操作详解

http://www.ruanyifeng.com/blog/2014/06/git_remote.html

6.5 【Git系列】 git rebase详解

<https://www.cnblogs.com/pinefantasy/articles/6287147.html>