

版本控制概述

2019年1月9日 15:10

1. 版本控制概念

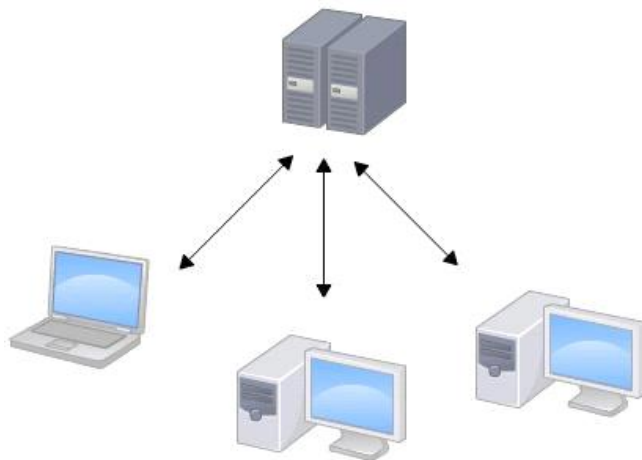
在代码开发过程中，往往需要对源码进行多次的修改操作，这样一来同一份代码就产生了多个版本，在开发过程中通常需要对这些多个版本代码进行管理，以便于在需要时进行代码回滚、多版本间比较、多人协作开发、代码分支、分支合并 等操作。

这样的需求大量的存在，而随着软件越来越复杂、代码越来越多、参与开发者越来越多，版本管理也变的越来越有难度，此时就需要专业的软件来对版本进行管理，这个过程就称之为版本控制，实现版本控制的软件就称之为版本控制软件。

2. 版本控制软件分类

a. 集中式版本控制

在集中式版本控制中，版本库是集中存放在中央服务器的，开发者在开发之前要先从中央服务器取得最新的版本，然后开始工作，工作完成后，再把自己的代码推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。



优点：

- 便于集中式的代码管理

- 便于进行权限控制

缺点：

- 需要联网才可以工作,而且项目庞大的情况下对带宽的要求比较高

- 中心服务器存在单节点故障风险

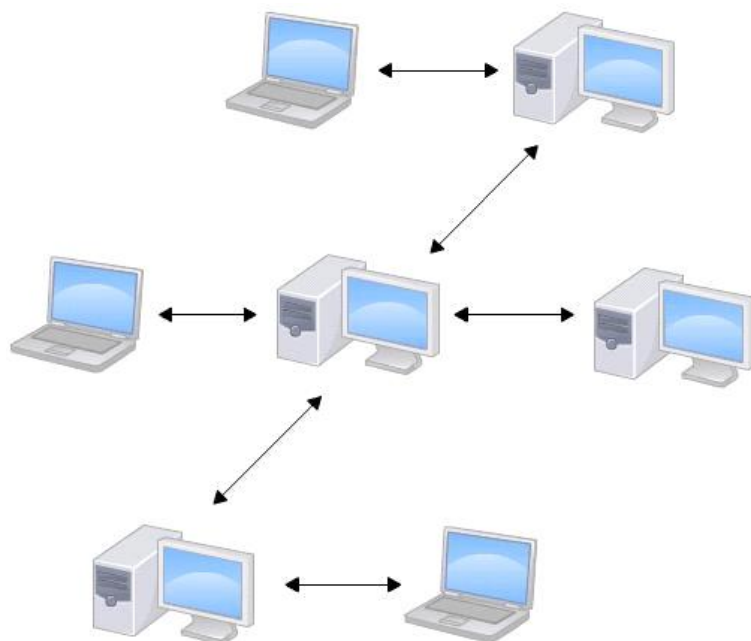
常见的集中式版本控制软件：

- CVS、SVN

b. 分布式版本控制

在分布式版本控制系统中，没有“中央服务器”的概念，每个人的电脑上都是一个完整的版本库。而在多人协同工作时，通过推送各自的修改，保证多人间的版本一致。

但其实，在实际开发中，很少真的在两个电脑间进行修改的推送，而是选择一台充当“中央服务器”，但这个服务器仅仅是为了使用便利，本质上和其他机器没有任何区别，即使宕机，整个分布式版本控制仍然可以工作。



优点：

不需要联网也可以工作

不存在单节点故障风险

缺点：

无法实现严格的权限控制

常见的分布式版本控制软件：

Git

3. git历史

Linus在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。

Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？

在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！

你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业

的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。

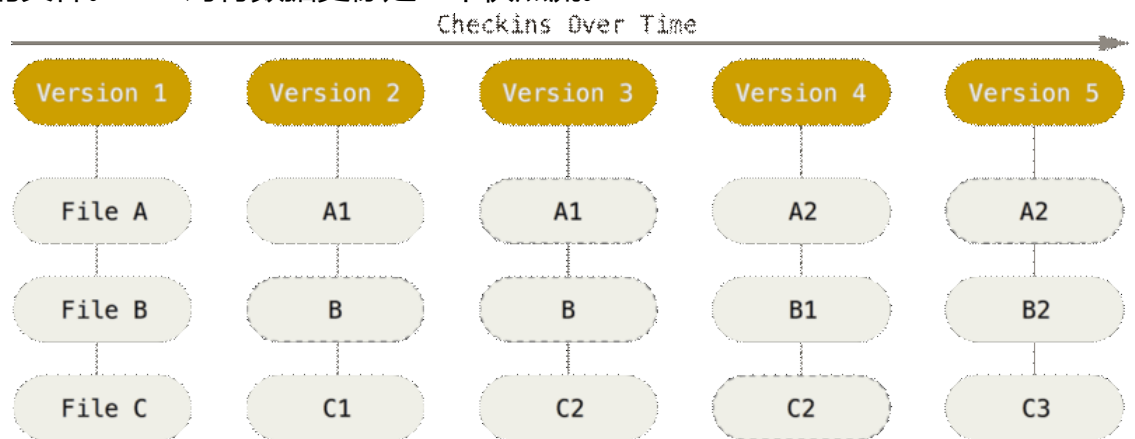
GIT基本概念

2019年1月9日 17:40

1. GIT原理

a. GIT基于版本快照工作

Git 更像是把数据看作是对小型文件系统的一组快照。每次你提交更新，或在 Git 中保存项目状态时，它主要对当时的全部文件制作一个快照并保存这个快照的索引。为了高效，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。Git 对待数据更像是一个快照流。



b. GIT保证数据完整性

Git 中所有数据在存储前都计算校验和，然后以校验和来引用。这个功能建构在 Git 底层，是构成 Git 哲学不可或缺的部分。若你在传送过程中丢失信息或损坏文件，Git 就能发现。

Git 用以计算校验和的机制叫做 SHA-1 散列 (hash, 哈希)。这是一个由 40 个十六进制字符 (0-9 和 a-f) 组成字符串，基于 Git 中文件的内容或目录结构计算出来。

SHA-1 哈希看起来是这样：

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

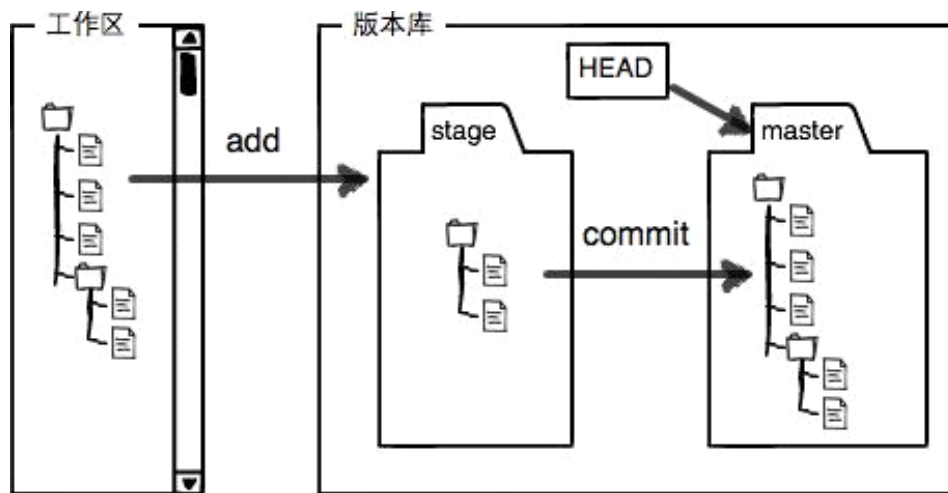
Git 中使用这种哈希值的情况很多，你将经常看到这种哈希值。实际上，Git 数据库中保存的信息都是以文件内容的哈希值来索引，而不是文件名。

c. GIT一般只添加数据,因此不用担心版本丢失

你执行的 Git 操作，几乎只往 Git 数据库中增加数据。很难让 Git 执行任何不可逆操作，或者让它以任何方式清除数据。同别的版本控制工具一样，未提交更新时有可能丢失或弄乱修改的内容；但是一旦你提交快照到 Git 中，就难以再丢失数据，特别是如果你定期的推送数据库到其它仓库的话。

这使得我们使用 Git 成为一个安心愉悦的过程，因为我们深知可以尽情做各种尝试，而没有把事情弄糟的危险。

2. 版本库(仓库) 工作区 暂存区 分支区



a. 版本库(仓库)

版本库又名仓库，英文名repository，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

b. 工作区

存放要管理的文件的位置

c. 暂存区

版本库中 包含暂存区 git add命令 或 git rm 命令加入的操作 被记录在暂存区内

d. 分支区

版本库中 包含分支区 是最终版本信息保存的位置 git commit命令将暂存区内记录的操作 提交到分支中

可以配置多个分支，如果不指定则默认为master分支，并有一个head指针指向master分支的最新位置

GIT的安装配置

2019年1月16日 9:53

1. 下载Git

Git最早只支持Linux平台，目前已经能够支持Linux、Unix、Windows、OS系统之上。

下载地址：

<https://git-scm.com/>

2. 安装Git

a. Linux上安装Git

解压Linux版源码包

依次执行

`./config`

`make`

`sudo make install`

b. Windows上安装Git

下载安装包

双击执行默认安装

双击 Git Bush 即可启动Git

3. 初始配置Git

因为Git是一款分布式的版本控制软件，多用户之间的互相通信需要确定身份，所以安装Git后需要先配置当前用户的名称和邮箱，才可以使用

```
$ git config --global user.name "Your Name"
$ git config --global user.email "email@example.com"
```

GIT本地版本管理

2019年1月10日 10:38

1. 创建版本库

通过cd命令进入目标文件夹 或 如果目标文件夹不存在可以通过mkdir命令创建文件夹

```
$cd e:
$pwd
$mkdir gitdemo01
$cd gitdemo01
```

通过git init 命令将该目录变为git所管理的仓库

```
$ git init
```

命令执行后，该目录下会多出一个.git的目录，这个目录是Git用来跟踪管理版本库的，请勿手动修改。

2. 向仓库提交新文件

在工作区中创建一个文本文件，并在文件中保存一些内容

```
test01.txt
Git is a version control system.
Git is free software.
```

查看版本库状态

```
git status
```

```
$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    test01.txt
nothing added to commit but untracked files present (use "git add" to track)
```

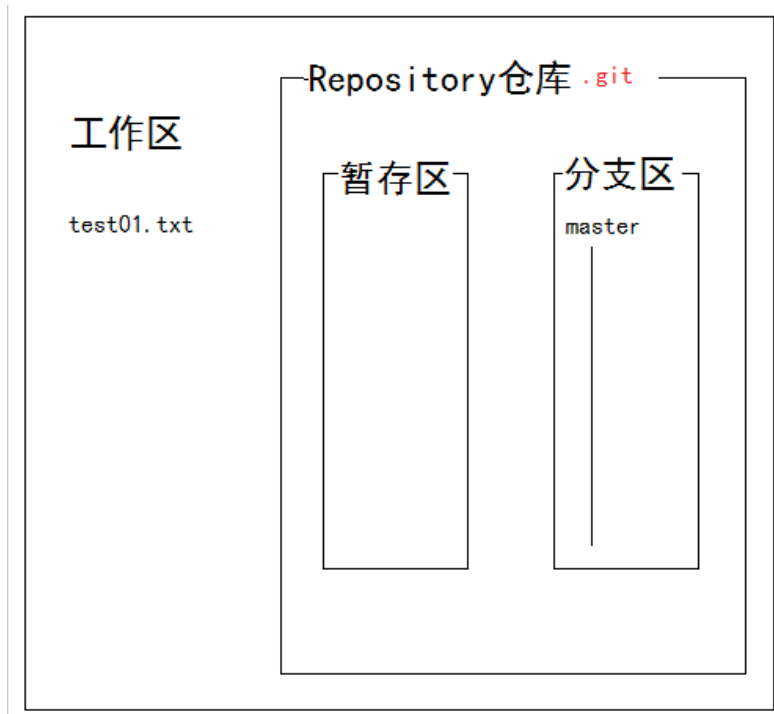
在master分支上

还没有提交

存在 [未追踪] 的文件：

test01.txt

没有东西可以提交，但有[未追踪]状态的文件，可以通过 "git add" 来追踪



增加文件到暂存区

```
git add test01.txt
```

查看版本库状态

```
git status
```

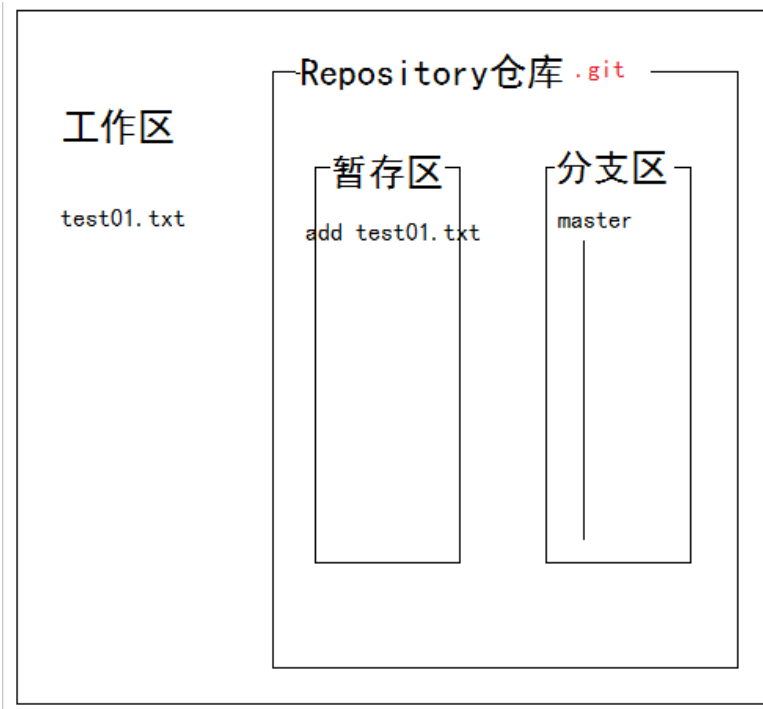
```
$ git status
On branch master
No commits yet
changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   test01.txt
```

在master分支上

有未提交的内容

等待提交的变更是：

新增了文件 test01.txt



提交版本到分支

```
git commit -m "add file test.txt"
```

```
$ git commit -m "add test01.txt"
[master (root-commit) 6bc0fb7] add test01.txt
1 file changed, 2 insertions(+)
create mode 100644 test01.txt
```

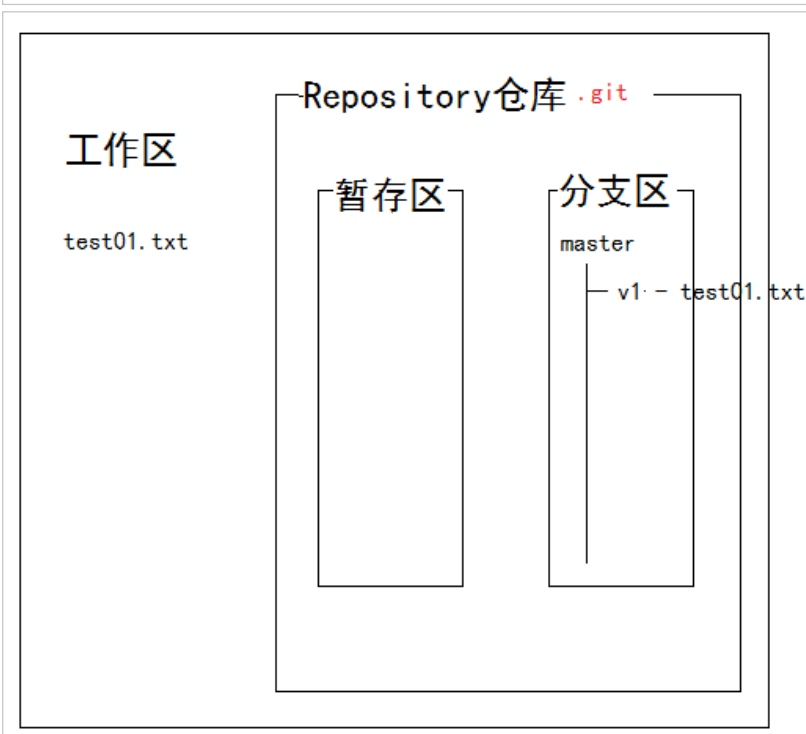
一个文件被修改，
包含两行内容

查看版本库状态

```
git status
```

```
$ git status
On branch master
nothing to commit, working tree clean
```

在master分支上，没有东西
要提交，工作树是干净的



** -m后面输入的是本次提交的说明，可以输入任意内容

** 本例中只经历了一个修改就提交了，其实完全可以 多个修改后一次提交

3. 提交修改

修改test01.txt

```
git is a version control system.  
git is free software.
```

查看版本库状态

```
git status
```

```
$ git status  
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be  
  committed)  
  (use "git checkout -- <file>..." to discard changes in working directory)  
  
modified:   test01.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")
```

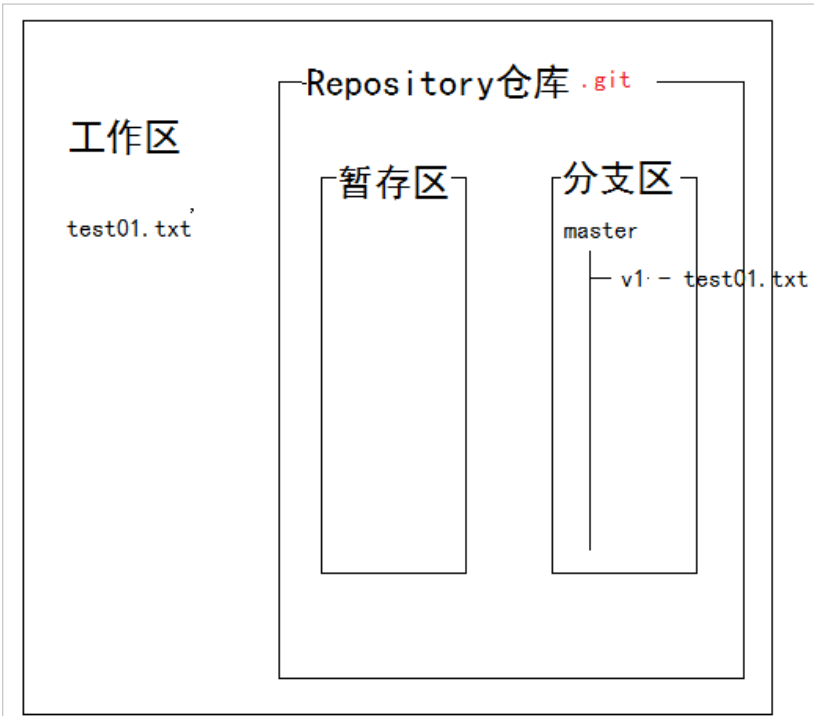
在master分支上

修改未被提交

可以通过"git add"命令将想要提交的修改加入暂存区

可以通过"git checkout"命令来丢弃工作区中的修改

没有任何修改被加入要提交



查看文件具体修改

```
git diff test01.txt
```

```
$ git diff test01.txt
diff --git a/test01.txt b/test01.txt
index d8036c1..ccd90e1 100644
--- a/test01.txt
+++ b/test01.txt
@@ -1,2 +1,2 @@
-Git is a version control system.
-Git is free software.
\ No newline at end of file
+git is a version control system.
+git is free software.
\ No newline at end of file
```

显示了当前文件被修改的内容

增加文件到暂存区

```
git add test01.txt
```

查看状态

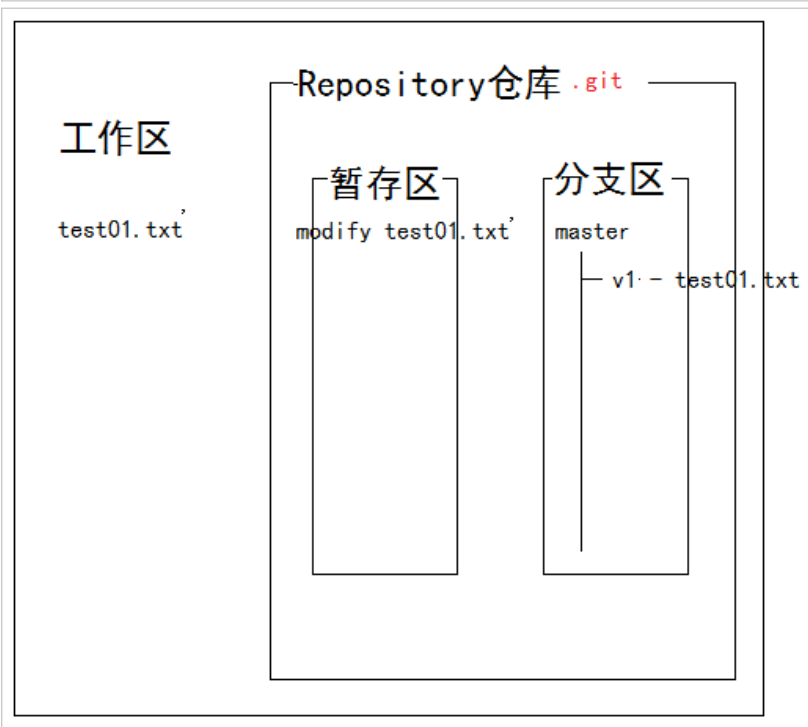
```
git status
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

在master分支上

修改可以被提交了

```
modified:   test01.txt
```



提交版本到分支

```
git commit -m "change Git to git"
```

```
$ git commit -m "modify test01.txt"
[master 8f1c5c7] modify test01.txt
1 file changed, 2 insertions(+), 2 deletions(-)
```

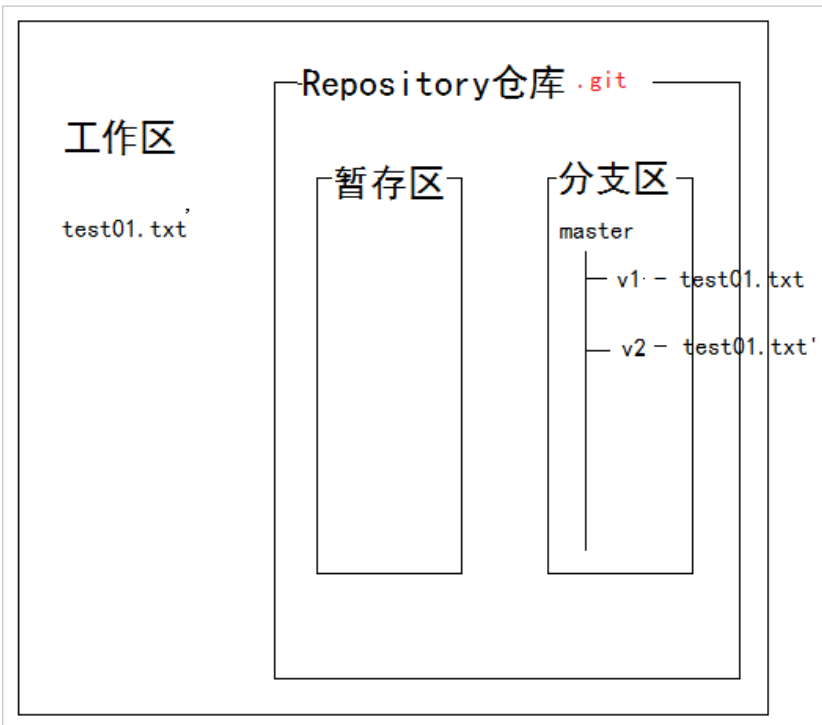
test01.txt被修改

再次查看状态

```
git status
```

```
$ git status  
On branch master  
nothing to commit, working tree clean
```

在master分支上，没有东西要提交，工作树是干净的



4. 查看历史版本

修改test01.txt

```
git is a version control system.  
git is free software.  
git is good.  
...
```

增加文件到暂存区

```
git add test.txt
```

提交版本到分支

```
git commit -m "...some msg..."
```

...重复以上步骤保存文件的多个版本...

略

查看历史版本

```
git log  
或  
git log --pretty=oneline
```

```

$ git log
commit 24f19f4aeac175986e35a53f90cc054622e13bd7 (HEAD -> master)
Author: park <piaoqian@tedu.cn>
Date: Wed Jan 16 11:40:18 2019 +0800

    modify test01.txt remove line

commit e11b91ffc3a1a60d4993d06c61a33c7e5ed410e6
Author: park <piaoqian@tedu.cn>
Date: Wed Jan 16 11:39:24 2019 +0800

    modify test01.txt add line

commit 8f1c5c79f8ed149ff35e499a4385baf3ac201058
Author: park <piaoqian@tedu.cn>
Date: Wed Jan 16 11:34:24 2019 +0800

    modify test01.txt

commit 6bc0fb7785bc93e420aba82247b48844beaf38ac
Author: park <piaoqian@tedu.cn>
Date: Wed Jan 16 11:07:26 2019 +0800

    add test01.txt

```

5. 回滚版本

- a. 基于当前版本回滚若干版本

```
git reset --hard HEAD^
```

在Git中，用HEAD表示当前版本，上一个版本就是HEAD[^]，上上一个版本就是HEAD^{^^}，当然往上100个版本写100个[^]比较容易数不过来，所以写成HEAD~100。

- b. 基于指定版本号回滚版本

```
git reset --hard 5527510751b4303390bb4f321bfa8b7f997cbfd0
```

```
git reset --hard 55275
```

这种方式的好处是可以任意跳转到指定版本,而不必参考当前版本位置。

- c. 查看历史命令及对应执行后的版本号：

```
git reflog
```

此命令将会列出所有执行过的导致git版本变化的命令及其对应的版本号，可以配合git reset方便回滚到任意版本

6. 撤销修改

- a. 在工作区中进行了修改，但尚未加入到暂存区：

撤销修改 -- 将工作区中的文件恢复到最近一次add 或 commit之前的状态

```
git checkout -- test.txt
```

- b. 在工作区中进行了修改，并已经增加文件到暂存区，但尚未提交到分支区：
撤销修改 -- 将暂存区中对这个文件的记录删除掉

```
git reset HEAD test.txt
```

撤销修改 -- 将工作区中的文件恢复到最近一次add 或 commit之前的状态

```
git checkout -- test.txt
```

- c. 在工作区中进行了修改，并且已经增加文件到暂存区，且已提交到分支：

可以通过之前所学的版本回退技术完成撤销操作。

但这仅仅是将版本进行了回滚，git并没有真正的忘记这次被回滚的提交，后续仍然可以再通过回滚操作回到这个版本上，这就是之前所说的git一旦提交就无法删除。

7. 删除文件

从工作区中删除文件

手动删除文件

或

命令删除文件 `rm test01.txt`

查看版本库状态

```
git status
```

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working c
ry)

    deleted:    test01.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

在master分支上

修改未被跟踪

可以通过git add / rm 跟踪修改
可以通过git checkout 丢弃修

增加删除文件操作到暂存区：

```
git rm test.txt
```

查看版本库状态

```
git status
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    test01.txt
```

在master分支上

有修改未被提

提交版本到分支

```
git commit
```

8. 细节:Git跟踪并管理的是修改,而非文件。

实验:

修改test.txt

```
test.txt
git is a version control system.
git is free software.
git is good
git is easy to use
```

增加文件到暂存区

```
git add test.txt
```

修改test.txt

```
test.txt
git is a version control system.
git is free software.
git is good
git is easy to use
git is very useful
```

提交版本到分支

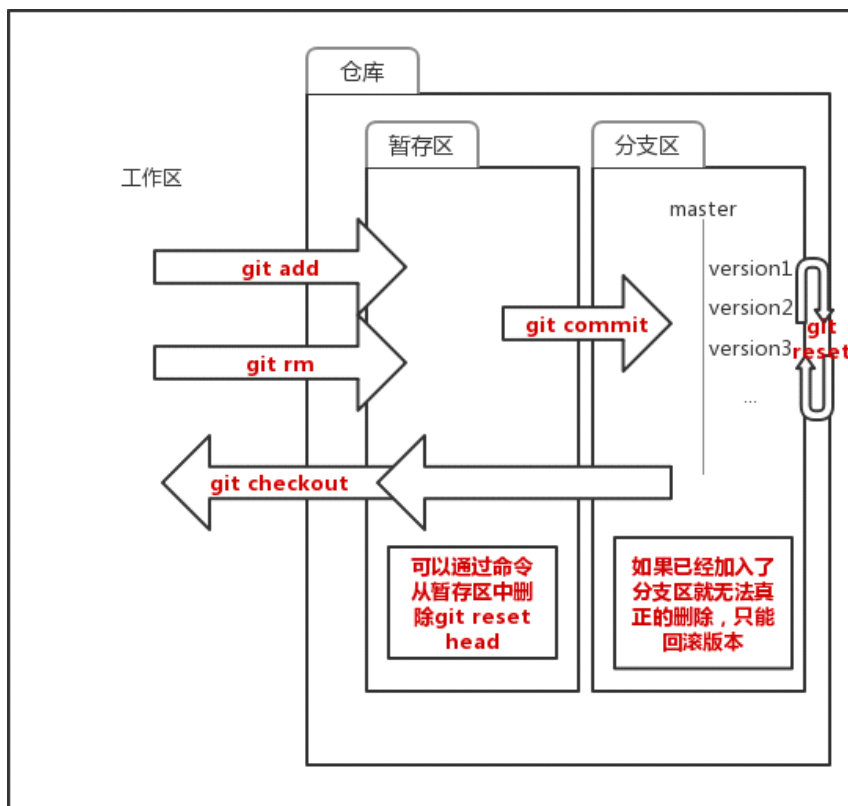
```
git commit -m "change test"
```

查看状态:

```
git status
```

```
发现git is very useful并没有被提交
```

9. 命令总结



10. 通过tag指定版本标签名

git中的版本都有版本编号，对代码的提交和回滚都是基于版本编号进行的，但是git中的版本编号是一串随机串，不好记忆，此时可以使用标签机制为某个提交增加标签，方便以后查找。打了标签之后所有需要用到版本编号的位置都可以用对应的标签替代。

在当前分支当前提交上打标签：

```
$git tag v1.0
```

查看所有标签,默认标签是打在最新提交的commit上的：

```
$git tag
```

如果想要打标签在某个指定历史commit上：

```
$ git tag v0.9 f52c633
```

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

```
$ git tag -a v0.1 -m "version 0.1 released" 1094adb
```

可以通过如下命令查看一个tag信息：

```
$ git show v0.1
```

如果标签打错了，也可以删除：

```
$ git tag -d v0.1
```

如果要推送某个标签到远程，使用命令 `git push origin <tagname>`：


```
$ git push origin v1.0
```

或者，一次性推送全部尚未推送到远程的本地标签：

```
$ git push origin --tags
```

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

```
$ git tag -d v0.9
```

然后，从远程删除。删除命令也是push，但是格式如下：

```
$ git push origin :refs/tags/v0.9
```

11. 指定git忽略指定内容

可以在仓库中配置.gitignore文件，在其中配置哪些文件是不需要git管理，则git在处理此仓库时，会自动自动忽略声明的内容。

.gitignore文件格式：

- 1) **空格**不匹配任意文件，可作为分隔符，可用反斜杠转义
- 2) 以“**#**”开头的行都会被 Git 忽略。即#开头的文件标识注释，可以使用反斜杠进行转义。
- 3) 可以使用标准的**glob**模式匹配。所谓的glob模式是指shell所使用的简化的正则表达式。
- 4) 以斜杠“**/**”开头表示目录；“**/**”结束的模式只匹配文件夹以及在该文件夹路径下的内容，但是不匹配该文件；“**/**”开始的模式匹配项目跟目录；如果一个模式不包含斜杠，则它匹配相对于当前 .gitignore 文件路径的内容，如果该模式不在 .gitignore 文件中，则相对于项目根目录。
- 5) 以星号“*****”通配多个字符，即匹配多个任意字符；使用两个星号“******”表示匹配任意中间目录，比如`a/**/z`可以匹配 a/z, a/b/z 或 a/b/c/z等。
- 6) 以问号“**?**”通配单个字符，即匹配一个任意字符；
- 7) 以方括号“**[]**”包含单个字符的匹配列表，即匹配任何一个列在方括号中的字符。比如[abc]表示要么匹配一个a，要么匹配一个b，要么匹配一个c；如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配。比如[0-9]表示匹配所有0到9的数字，[a-z]表示匹配任意的小写字母)。
- 8) 以叹号“**!**”表示不忽略(跟踪)匹配到的文件或目录，即要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号(!)取反。需要特别注意的是：**如果文件的父目录已经被前面的规则排除掉了，那么对这个文件**

用"!"规则是不起作用的。也就是说"!"开头的模式表示否定，该文件将会再次被包含，如果排除了该文件的父级目录，则使用"!"也不会再次被包含。可以使用反斜杠进行转义。

需要谨记：**git对于.ignore配置文件是按行从上到下进行规则匹配的，意味着如果前面的规则匹配的范围更大，则后面的规则将不会生效；**

示例：

```
#          表示此为注释,将被Git忽略
*.a        表示忽略所有 .a 结尾的文件
!lib.a     表示但lib.a除外
/TODO      表示仅仅忽略项目根目录下的 TODO 文件，不包括 subdir/TODO
build/     表示忽略 build/目录下的所有文件，过滤整个build文件夹；
doc/*.txt  表示会忽略doc/notes.txt但不包括 doc/server/arch.txt

bin/:      表示忽略当前路径下的bin文件夹，该文件夹下的所有内容都会被忽略，不忽略 bin 文件
/bin:      表示忽略根目录下的bin文件
/*.c:      表示忽略cat.c，不忽略 build/cat.c
debug/*.obj: 表示忽略debug/io.obj，不忽略 debug/common/io.obj和 tools/debug/io.obj
**/foo:    表示忽略/foo,a/foo,a/b/foo等
a/**/b:    表示忽略a/b, a/x/b,a/x/y/b等
!/bin/run.sh 表示不忽略bin目录下的run.sh文件
*.log:     表示忽略所有 .log 文件
config.php: 表示忽略当前路径的 config.php 文件

/mtk/      表示过滤整个文件夹
*.zip      表示过滤所有.zip文件
/mtk/do.c   表示过滤某个具体文件
```

被过滤掉的文件就不会出现在git仓库中（gitlab或github）了，当然本地库中还有，只是push的时候不会上传。

需要注意的是，gitignore还可以指定要将哪些文件添加到版本管理中，如下：

```
!*.*zip
!/mtk/one.txt
```

唯一的区别就是规则开头多了一个感叹号，Git会将满足这类规则的文件添加到版本管理中。为什么要有两种规则呢？

想象一个场景：假如我们只需要管理/mtk/目录中的one.txt文件，这个目录中的其他文件都不需要管理，那么.gitignore规则应写为：

```
/mtk/*  
!/mtk/one.txt
```

假设我们只有过滤规则，而没有添加规则，那么我们就需要把/mtk/目录下除了one.txt以外的所有文件都写出来！

注意上面的/mtk/*不能写为/mtk/，否则父目录被前面的规则排除掉了，one.txt文件虽然加了!过滤规则，也不会生效！

还有一些规则如下：

```
fd1/*
```

说明：忽略目录 fd1 下的全部内容；注意，不管是根目录下的 /fd1/ 目录，还是某个子目录 /child/fd1/ 目录，都会被忽略；

```
/fd1/*
```

说明：忽略根目录下的 /fd1/ 目录的全部内容；

```
/*
```

```
!.gitignore
```

```
!/fw/
```

```
/fw/*
```

```
!/fw/bin/
```

```
!/fw/sf/
```

说明：忽略全部内容，但是不忽略 .gitignore 文件、根目录下的 /fw/bin/ 和 /fw/sf/ 目录；注意要先对bin/的父目录使用!规则，使其不被排除。

IDEA本地版本控制

2019年12月14日 16:49



IDEA使用教
程之git...

GIT远程版本管理

2019年1月16日 15:13

1. 远程仓库概念

为了方便版本的交换，通常会使用一个中心服务器，24小时连续运行，提供版本控制服务

这就有两种做法：

- 自己搭建中心服务器

- 使用类似GitHub的代码托管网站

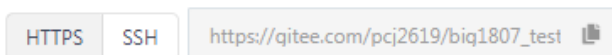
目前我们更多使用代码托管的方式进行开发工作

2. 远程仓库连接

**由于GitHub是境外的网站，连接不稳定，我们使用GitEE码云替代

注册GitEE账号，并配置仓库，会分配到仓库的地址

地址有两种形式，分别是HTTPS和SSH两种方式，用哪个都可以，区别是HTTPS需要输入用户名密码，而SSH可以实现免密登录



3. 将本地仓库关联到远程仓库

本地关联到远程仓库：

```
$ git remote add <服务器名称> <远程仓库地址>
$ git remote add origin https://gitee.com/pcj2619/big1807\_test01.git
```

4. 将本地仓库推送到远程仓库

把本地库的内容推送到远程：

```
$ git push -u <服务器名称> <本地分支名称>
$ git push -u origin master
```

**当使用HTTPS连接时，会弹出框要求输入用户名密码，输入码云的用户名和密码即可

**第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

```
$ git push -u origin master

Counting objects: 32, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (32/32), 2.62 KiB | 383.00 KiB/s, done.
Total 32 (delta 1), reused 0 (delta 0)
remote: Powered By Gitee.com
To https://gitee.com/pcj2619/big1807_test01.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

5. 从远程仓库克隆

将远程仓库克隆到本地

```
$ git clone <远程仓库地址> <本地目录名>
$ git clone https://gitee.com/pcj2619/big1807\_test01.git E:\gitdemo02
```

6. 拉取远程仓库中的更改到本地仓库

拉取远程仓库中的更改到本地仓库

```
$ git pull <远程仓库地址> <远程分支名>:<本地分支名>
$ git pull https://gitee.com/pcj2619/big1807\_test01.git master:master
```

7. 删除本地仓库到远程仓库关联

将本地仓库和远程仓库的关联关系删除

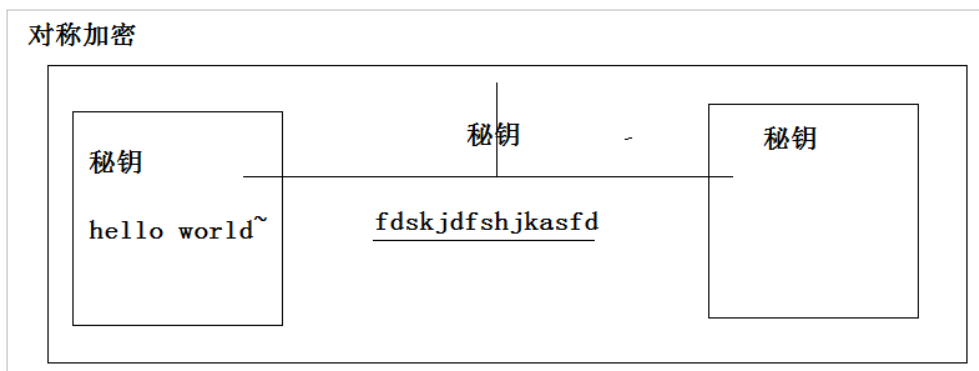
```
$git remote rm <服务器名称>
$git remote rm origin
```

8. 扩展：对称加密VS非对称加密

a. 对称机密

只有一个密钥，同时作为加密密钥和解密密钥使用

缺点是密钥传输过程中无法保证完全安全。



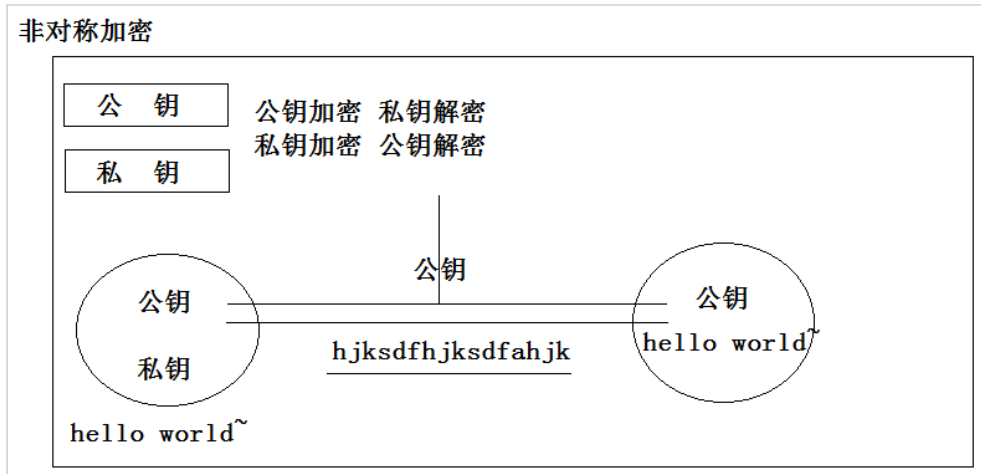
b. 非对称加密

有一对秘钥，称为公钥和私钥

公钥加密只有私钥能解，私钥加密只有公钥能解

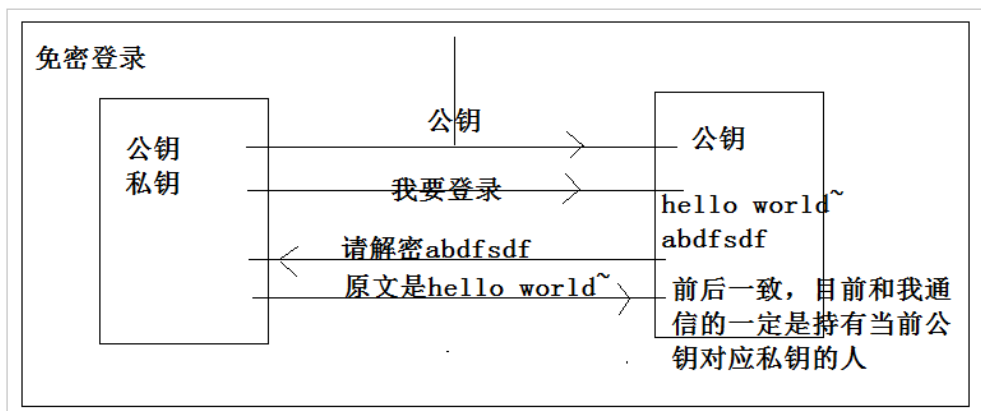
可以在网络中只传输公钥进行加密，再接收端通过私钥进行解密

这种方式解密用的私钥没有在网络中传输过，更加安全。



非对称加密在许多场景下有所应用，包括免密登录、数字签名等。

免密登录原理:



9. 使用SSH连接

使用SSH连接相对于HTTPS连接的优势在于,是通过非对称加密保护的,更加安全，且可以实现免密登录.推荐大家使用SSH连接.

a. 客户端生成公钥私钥

```
$ ssh-keygen -t rsa -C "youremail@example.com"
```

此命令会在用户主目录下生成.ssh目录，里面有id_rsa和id_rsa.pub两个文件，其中id_rsa是私钥，不能泄露出去，id_rsa.pub是公钥，可以放心的传播。

b. 在码云中配置公钥

在码云的 [配置] - [SSH公钥] 中 将本机中生成的公钥内容粘贴配置进去。之后就可以使用ssh连接免密访问码云了。

IDEA远程版本控制

2019年12月14日 16:49



IDEA 插件
- 码云 ...

GIT的协作开发 - 分支管理

2019年1月18日 9:26

1. 分支概念

所谓的分支就是从当前的开发线中分割出的一条新的开发线。

利用新开出的开发线在不影响原来开发线的过程中，开发并提交代码，等到新的开发线开发完成，可以一次性加入原开发线，即进行分支合并操作。

分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

2. 分支的使用 - 基本使用

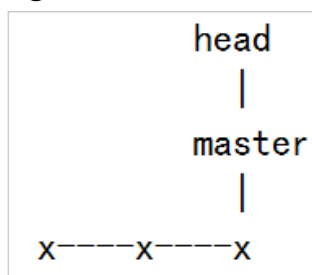
a. 查看分支

```
查看分支  
$ git branch
```

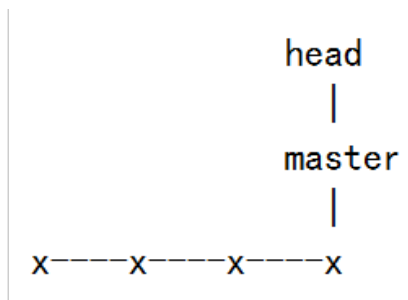
b. HEAD指针

在git中存在一个名为HEAD的指针,这个指针指向的位置就是当前操作位，这个指针指向哪里，之后的指令操作的就是哪里。

默认git中只有一个主分支master，head就指向master



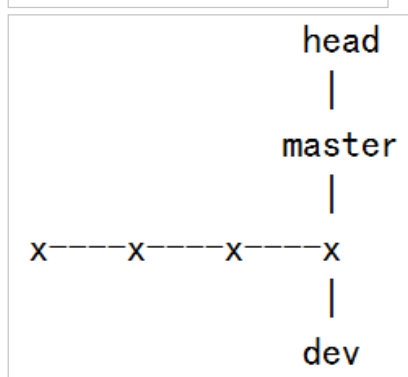
每次向master分支提交时，master和head都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长



c. 创建新的分支

当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交位

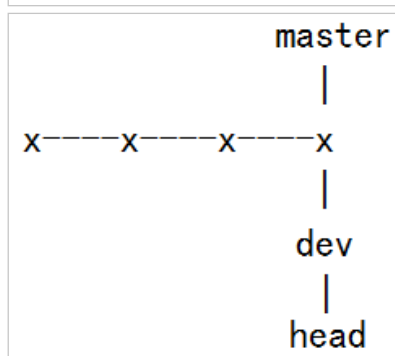
```
$ git branch <分支名>
$ git branch dev
```



d. 切换分支

可以通过命令切换分支，切换分支的本质就是将head指针指向了目标分支，由于只是切换一个指针指向，所以切换分支的操作效率非常的高

```
$ git checkout <分支名>
$ git checkout dev
```



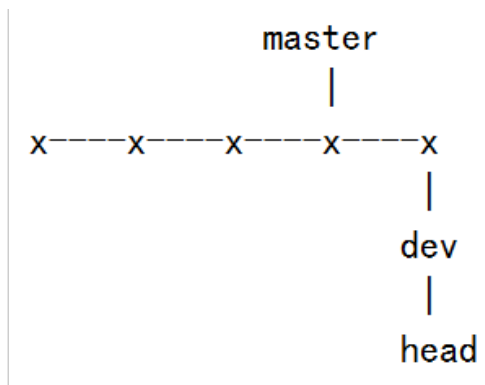
e. 快速创建和切换分支

也可以通过如下命令创建分支并直接切换到该分支，等价于如上的两个命令

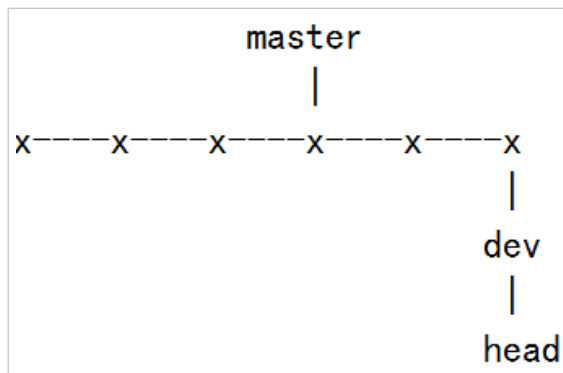
```
$ git checkout -b <分支名>
$ git checkout -b dev
```

f. 提交新版本到分支中

此时新的版本会加入到head指定的分支中，即dev分支



此过程可能重复多次，在dev分支中完成开发



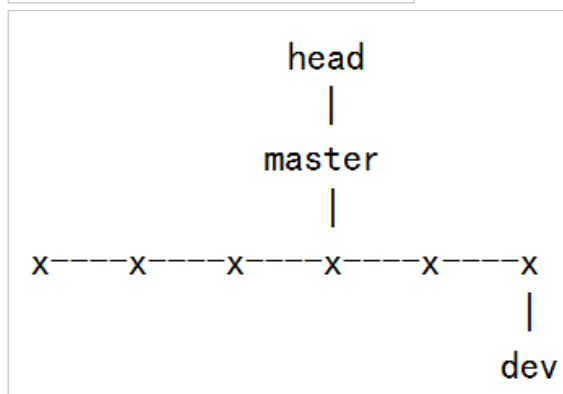
g. 将dev分支合并到master分支中

想要将dev分支合并到master分支，只要将head重新指向master分支，并命令head指向的master分支合并到dev的指针位即可 - 本质上就是将master的指针指向dev的指针,最后删除dev分支。

切换回主分支

```
$git checkout <分支名>
```

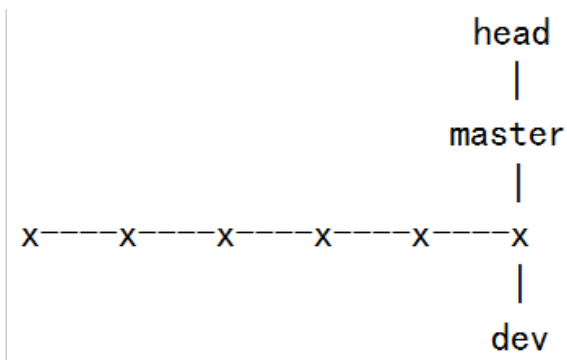
```
$git checkout master
```



合并分支

```
$git merge <要被合并的分支名>
```

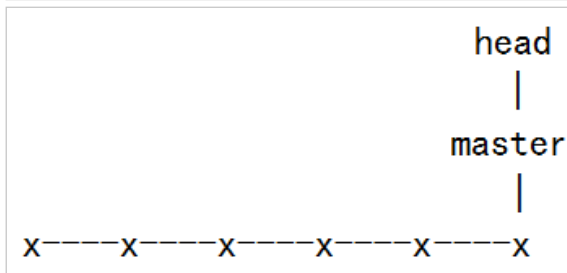
```
$git merge dev
```



删除dev分支

```
$ git branch -d <要删除的分支名称>
```

```
$ git branch -d dev
```



查看分支

```
$ git branch
```

**在合并分支时可以增加一个--no-ff参数，此参数将会在合并分支时禁用Fast forward模式，效果上回自动在merge时增加一个提交，方便后续管理

```
$git merge --no-ff -m "som commit msg.." dev2
```

3. 分支的使用 - 冲突处理

a. 冲突的产生

当多个分支之间的进行合并时，如果多个分支对同一个文件进行过不同处理，则在合并时GIT不知道要保留哪个处理结果，就会产生冲突。

在master分支上创建demo01.txt

```
demo01.txt
```

```
aaa
```

在master分支上提交demo01.txt

```
$git add demo01.txt
```

```
$git commit -m "add demo01.txt"
```

创建新的分支dev并切分支换到dev

```
$ git checkout -b dev
```

修改文件demo01.txt,内容为bbb

demo01.txt

bbb

在dev分支上提交:

```
$ git add demo01.txt
```

```
$ git commit -m "modify demo01.txt to bbb"
```

切换到master分支:

```
$ git checkout master
```

修改demo01.txt,内容为ccc:

demo01.txt

ccc

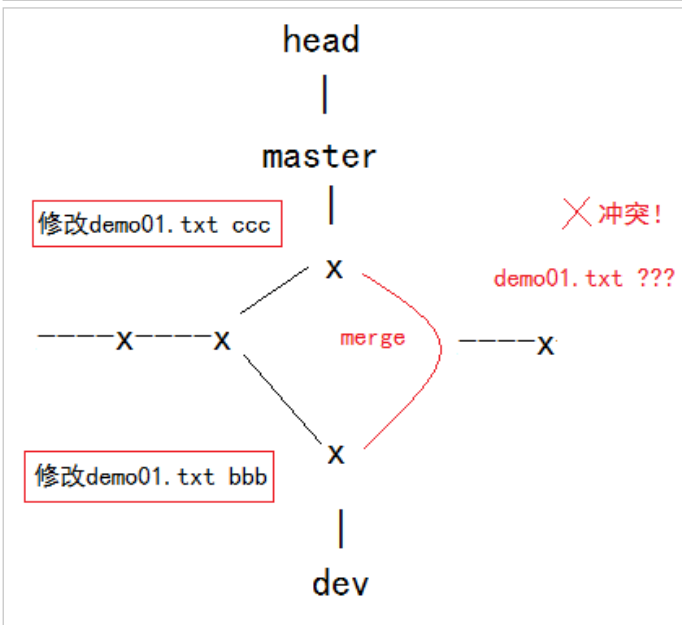
在master分支上提交:

```
$ git add demo01.txt
```

```
$ git commit -m "modify demo01.txt to ccc"
```

尝试通过merge操作在master分支上合并dev分支,由于master分支和dev分支对demo01.txt文件的状态不一致,造成合并时冲突

```
$ git merge dev
```



```
$ git merge dev
Auto-merging demo01.txt
CONFLICT (content): Merge conflict in demo01.txt
Automatic merge failed; fix conflicts and then commit the result.
```

自动合并demo01.txt时产生了冲突,要求合并冲突后再提交

```

$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   demo01.txt

no changes added to commit (use "git add" and/or
"git commit -a")

```

b. 解决冲突

打开冲突的文件

```

<<<<<< HEAD
ccc
=====
bbb
>>>>>> dev

```

找到冲突部分并手工进行修正

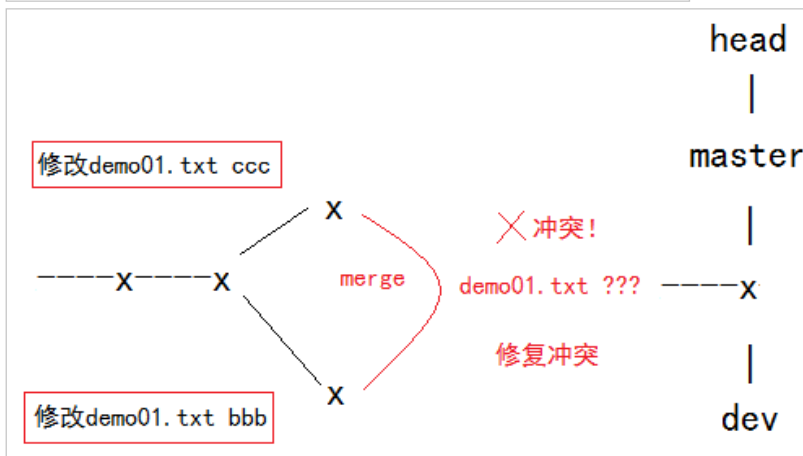
```
ccbb|
```

再正常提交一次

```

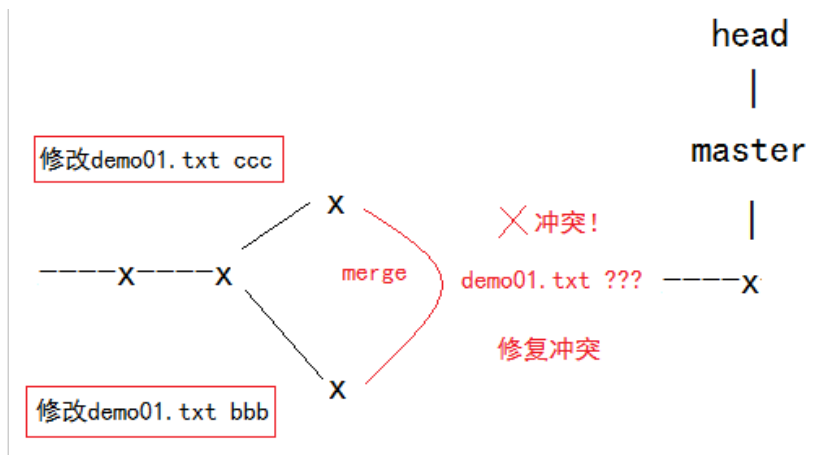
$git add demo01.txt
$git commit -m "demo01.txt fix conflict"

```



删除dev分支

```
$ git branch -d dev
```



4. 暂存分支

当某一分支工作到一半，突然需要切换到其他分支时，可以通过stash机制将当前分支状态暂存起来，再在需要的时候恢复。

新建并提交demo01.txt

```
demo01.txt
```

```
aaba
```

创建并切换到dev分支

```
$ git checkout -b dev
```

创建文件demo02.txt:

```
demo02.txt
```

```
xxx
```

将文件加入暂存区：

```
$ git add demo02.txt
```

临时需要切换到其他分支，暂存当前分支：

```
$ git stash
```

查看状态：

```
$ git status
```

切换到debug分支

```
$ git checkout master
```

```
$ git checkout -b debug
```

调试bug,修改文件：

```
demo01.txt
```

```
aaba --> aaaa
```

提交文件：

```
$ git add demo01.txt
```

```
$ git commit -m "debug fix demo01.txt"
```

切换到master分支

```
$ git checkout master
```

合并debug分支

```
$ git merge "merge debug" debug
```

切换回dev分支：

```
$ git checkout dev
```

```
$ git status
```

发现工作区是干净的，需要恢复暂存分支：

```
$ git stash list
```

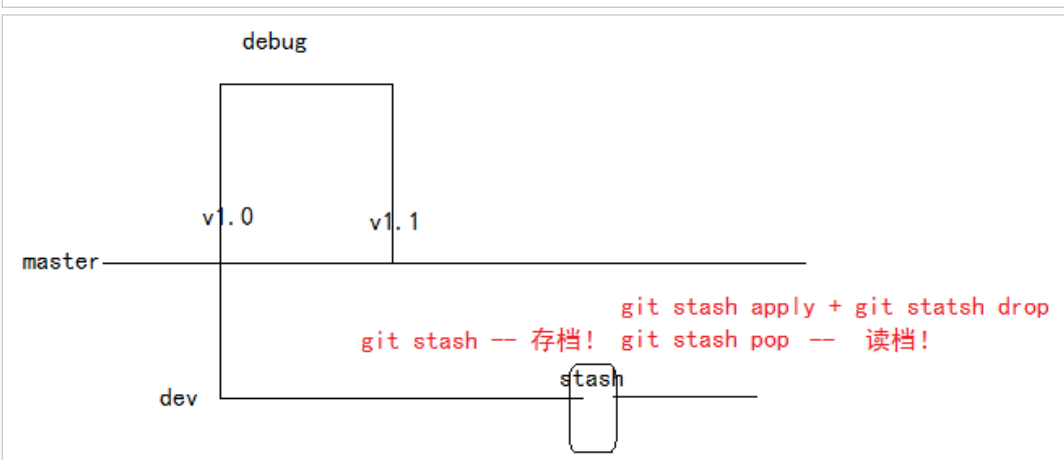
恢复到之前暂存的状态

有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来手动删除暂存状态

另一种方式是用git stash pop，恢复的同时把stash内容同时删除

```
$ git stash pop
```



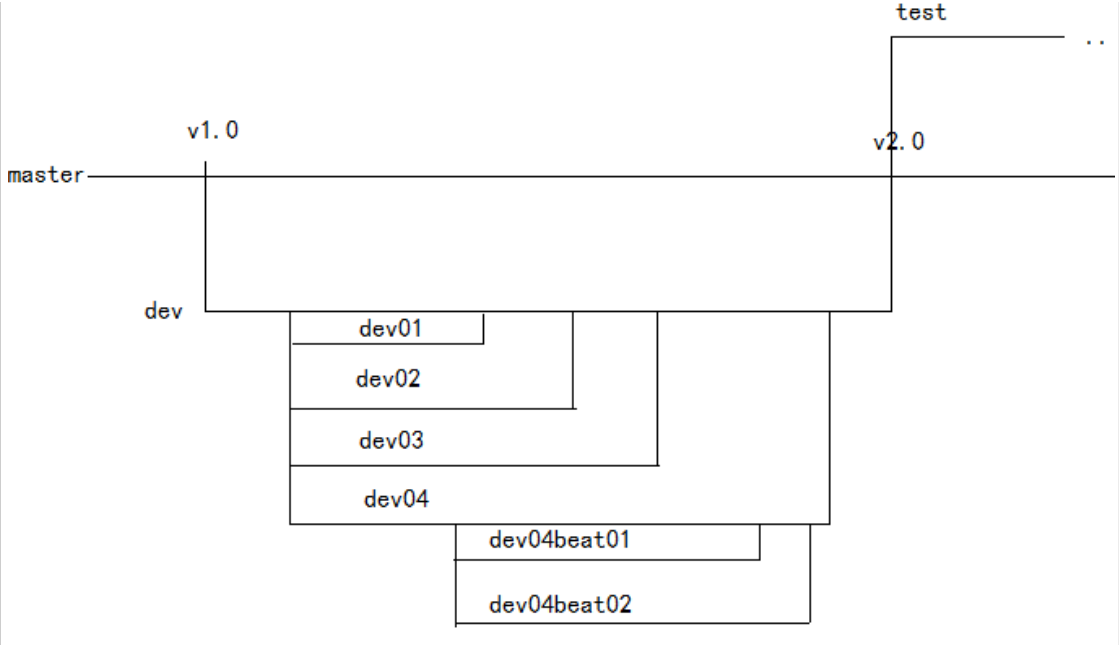
5. 企业级开发中分支使用策略

master分支应该是非常稳定的，也就是仅用来发布新版本

干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本

在企业级开发中，不同小组同时开发不同功能模块也是非常常见的，此时可以开辟分支之下更多个子分支进行并行开发。

灵活的利用分支机制，可以实现高效的并行开发下的版本管理。



IDEA分支管理

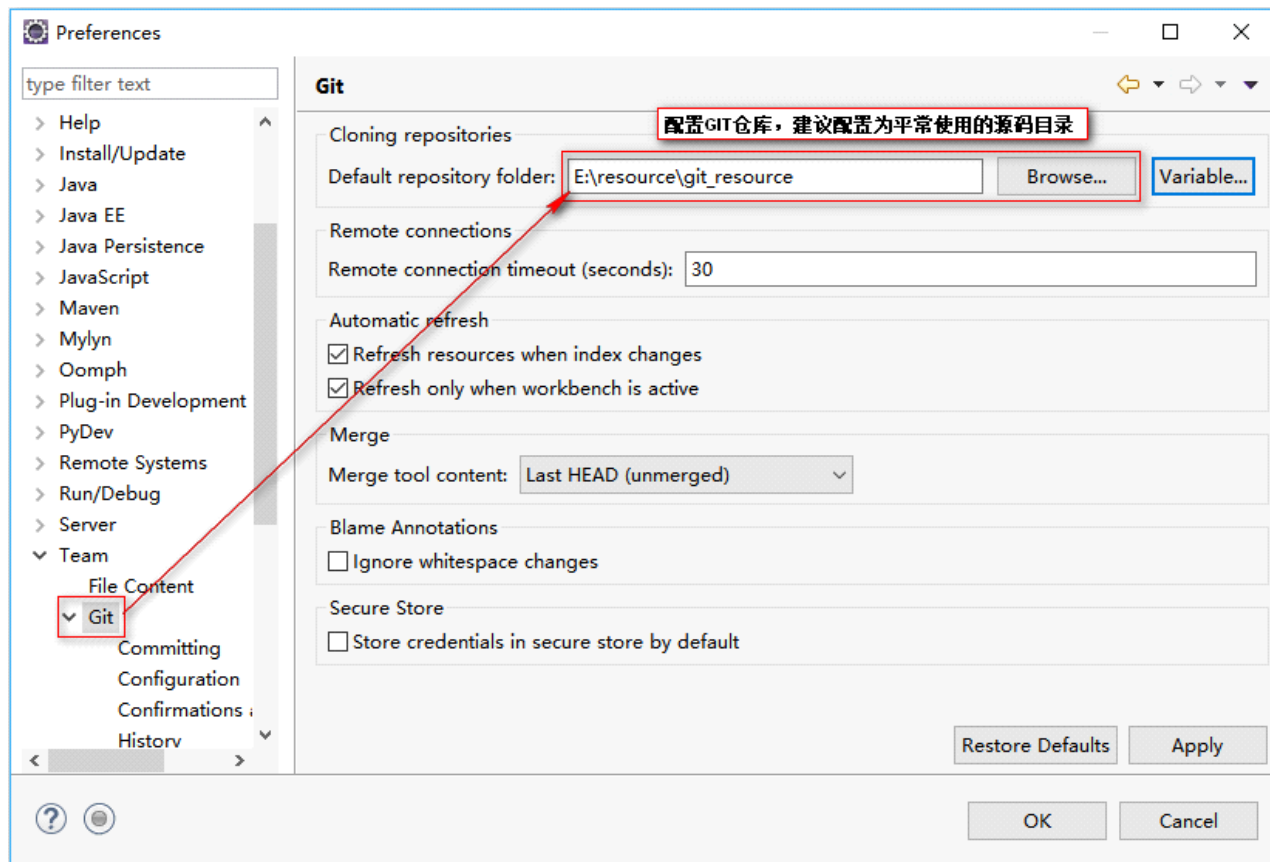
2019年12月14日 16:51

*EGIT - Eclipse下的GIT插件

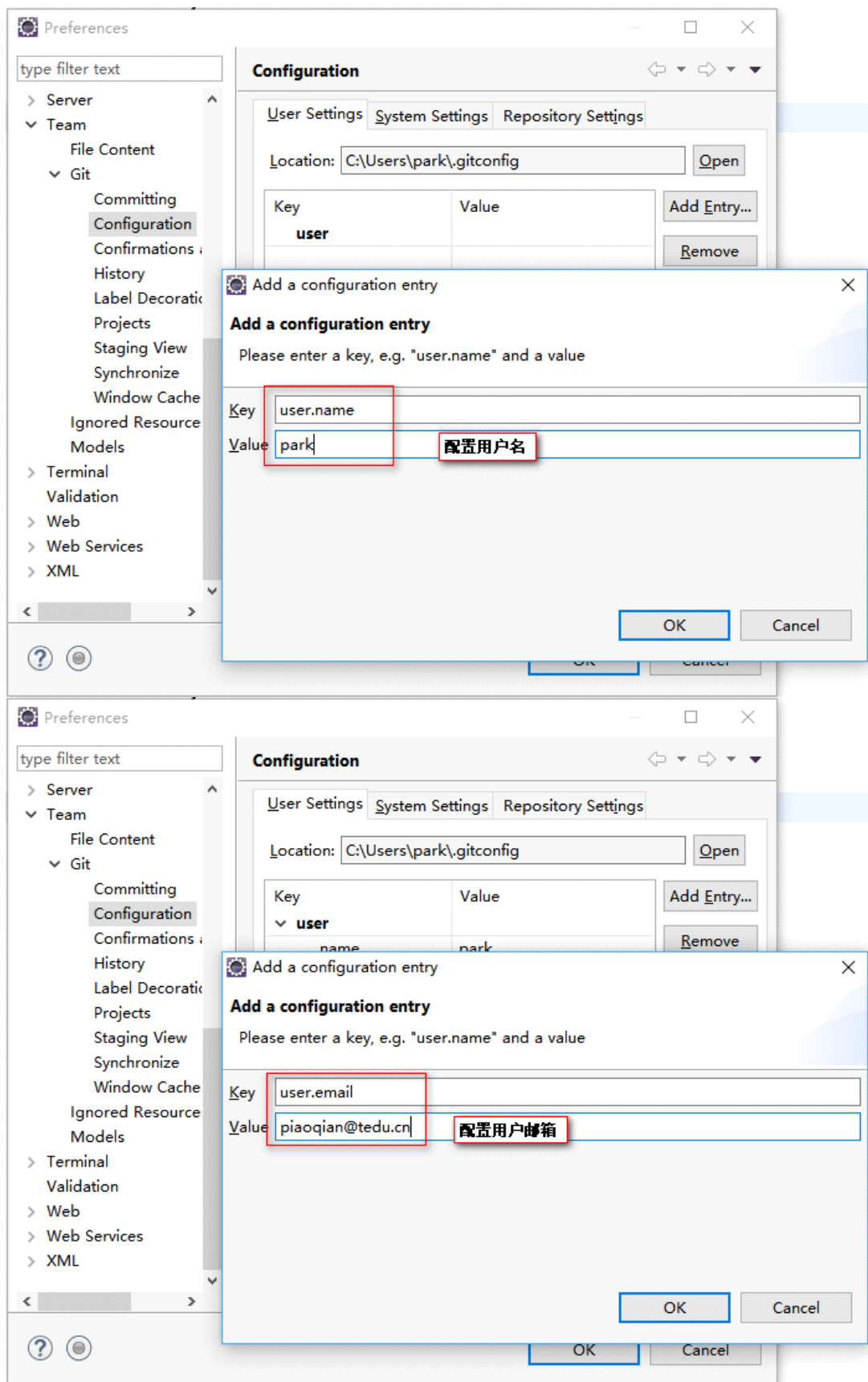
2019年1月9日 16:21

1. EGit-基本配置

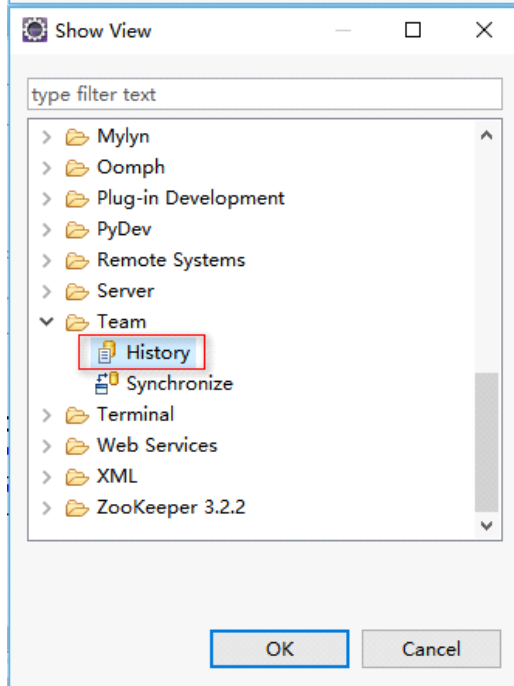
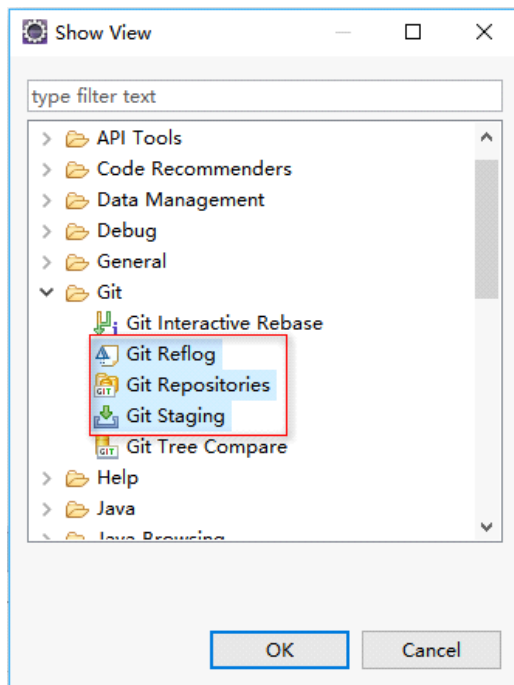
a. 配置GIT默认仓库



b. 配置GIT用户信息



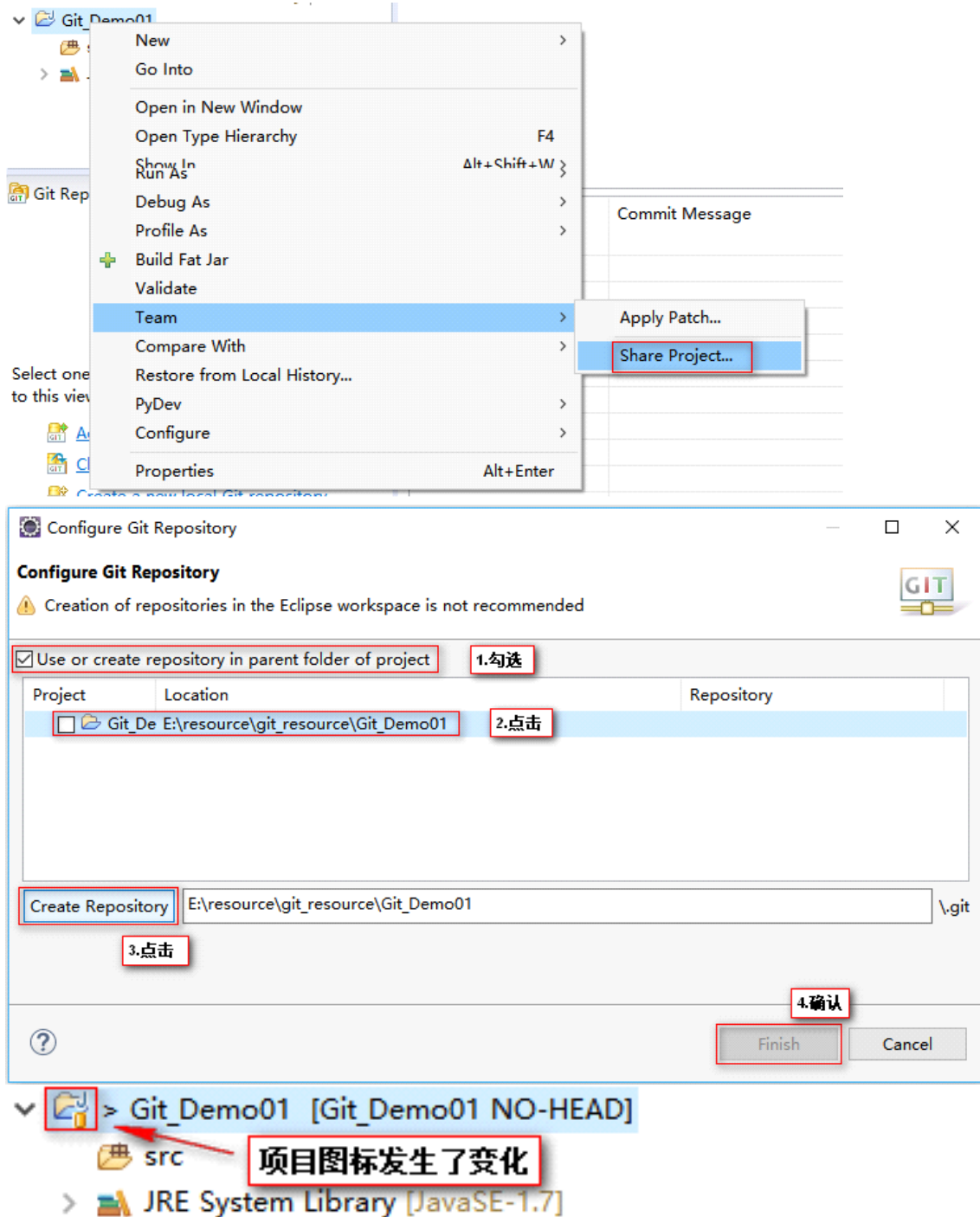
c. 打开EGit相关视图



*EGIT - 本地版本控制


2019年1月10日 17:05

1. 将项目转换为仓库



2. EGIT图标

a. 已跟踪[tracked]

 tracked.txt

文件已被仓库记录。

b. 未跟踪[untracked]



untracked.txt

仓库未跟踪，通常是新建的文件，要接入版本管理可以通过“Add to Index”或直接“Commit”操作。

c. 忽略[ignored]



ignored.txt

仓库认为该文件不存在（如bin目录，不需要关注）。通过右键Team => Ignore 添加忽略文件

d. 已修改[dirty]



> dirty.txt

修改“已跟踪[tracked]”的文件，未添加到暂存区Index（未“Add to Index”或“Commit”）的文件，标志与本地库不一致。

e. 已暂存[staged]



staged.txt

修改“已跟踪[tracked]”的文件，并添加到暂存区Index（即执行“Add to Index”）；

f. 已部分暂存[partially-staged]



> partially-staged.txt

修改“已跟踪[tracked]”的文件，部分修改已添加暂存区Index，部分未添加。相当于：已跟踪的文件修改，Add to Index，Commit前又修改了文件。

g. 已添加[added]



added.txt

untracked 状态的文件，通过“Add to Index”被仓库已知，但是没有“Commit”，“Commit”后可变为“已跟踪[tracked]”状态。

h. 已删除[removed]



removed.txt

从工作区中删除文件，文件会消失，也就没有图标出现，下一次提交时被删除。Team => Untrack可以触发本图标，在“Commit”对话框中可以看到图标。

i. 冲突[conflict]



> conflict.txt

进行Merge合并操作会引起冲突，需要人工解决并添加到索引区修改状态。

j. 假设有效[assume-valid]

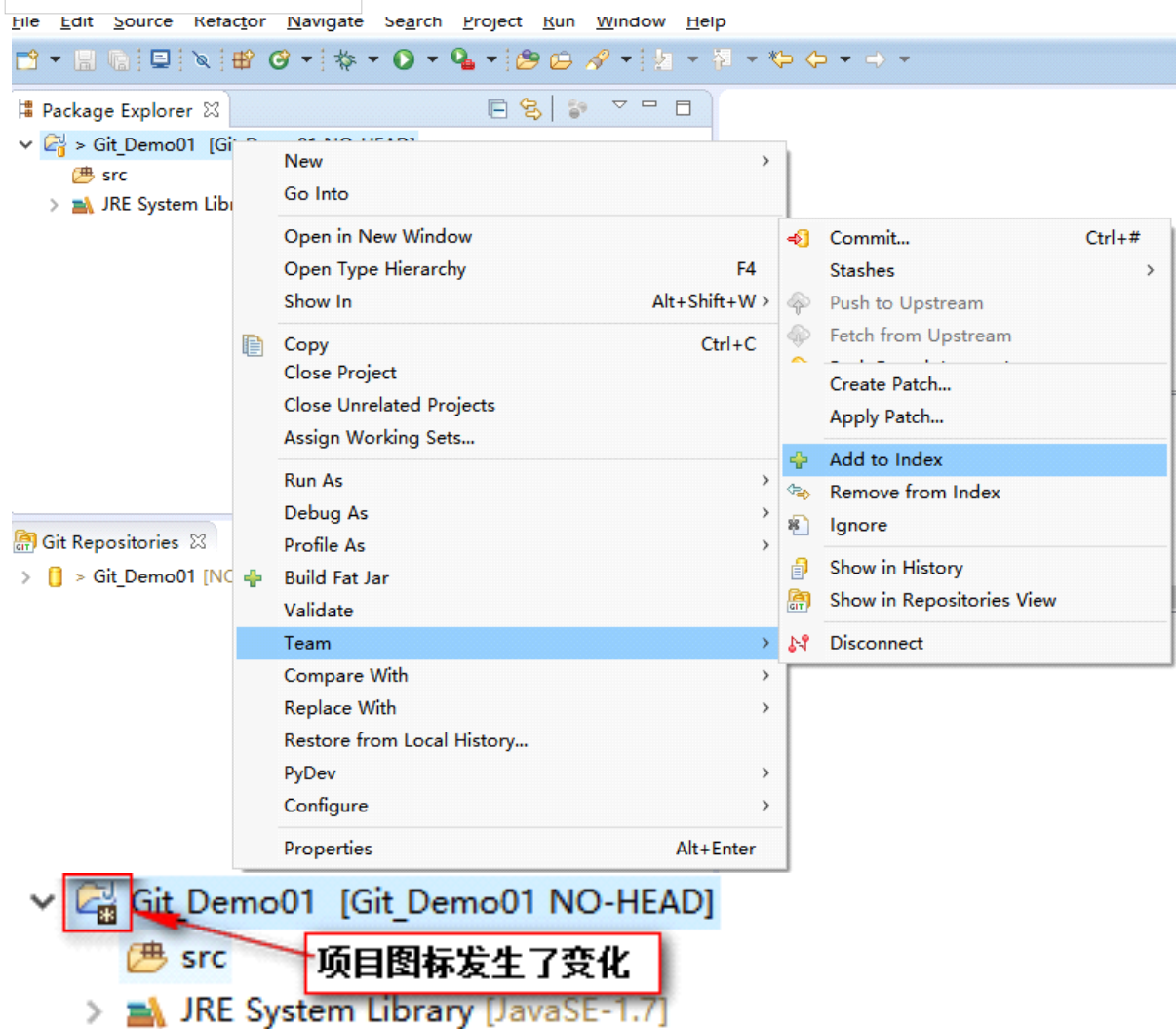


assume-valid.txt

一些修改未被Git检查。右键Team => Assume unchanged可产生该状态。

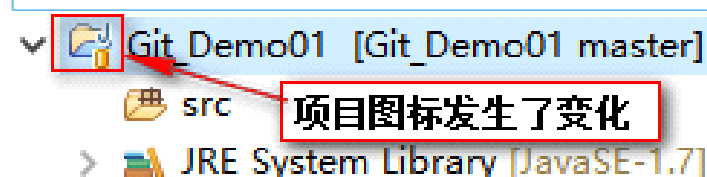
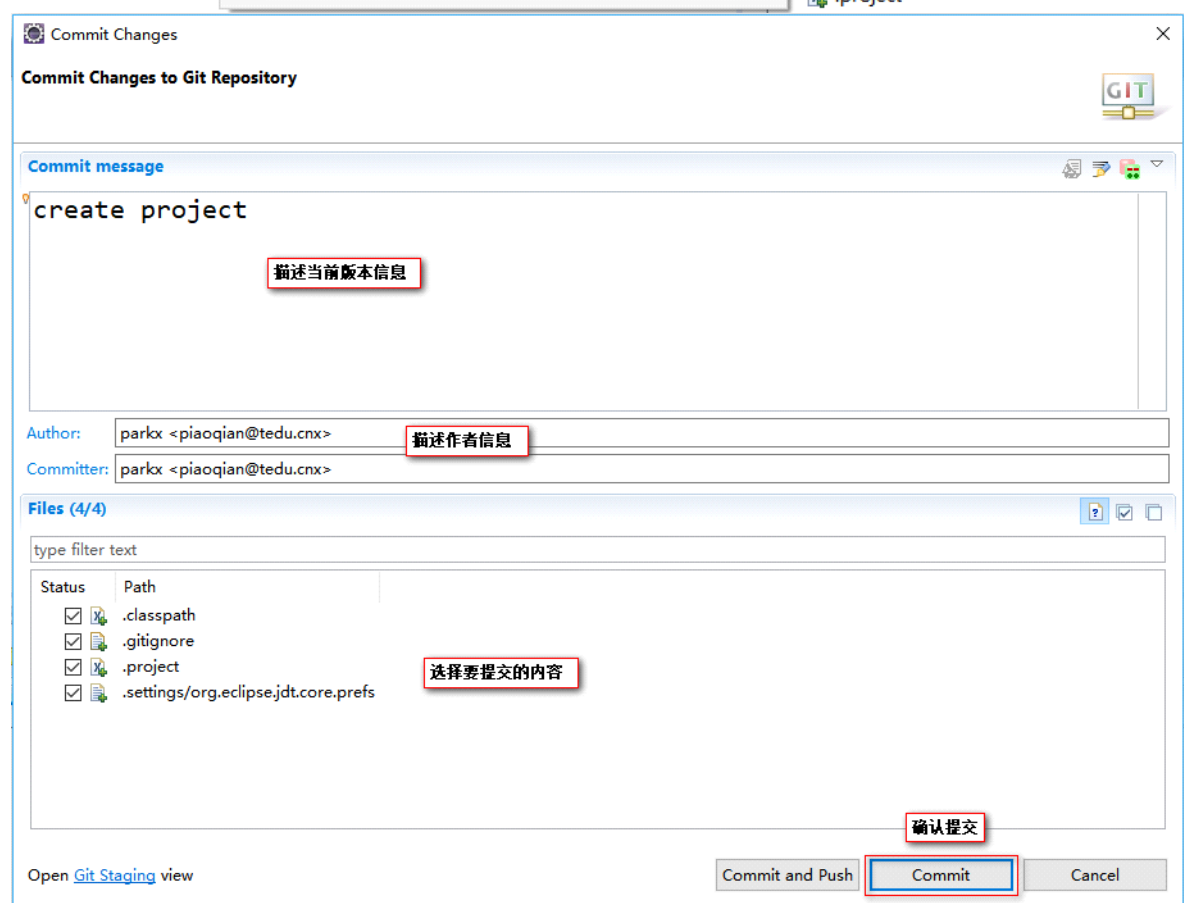
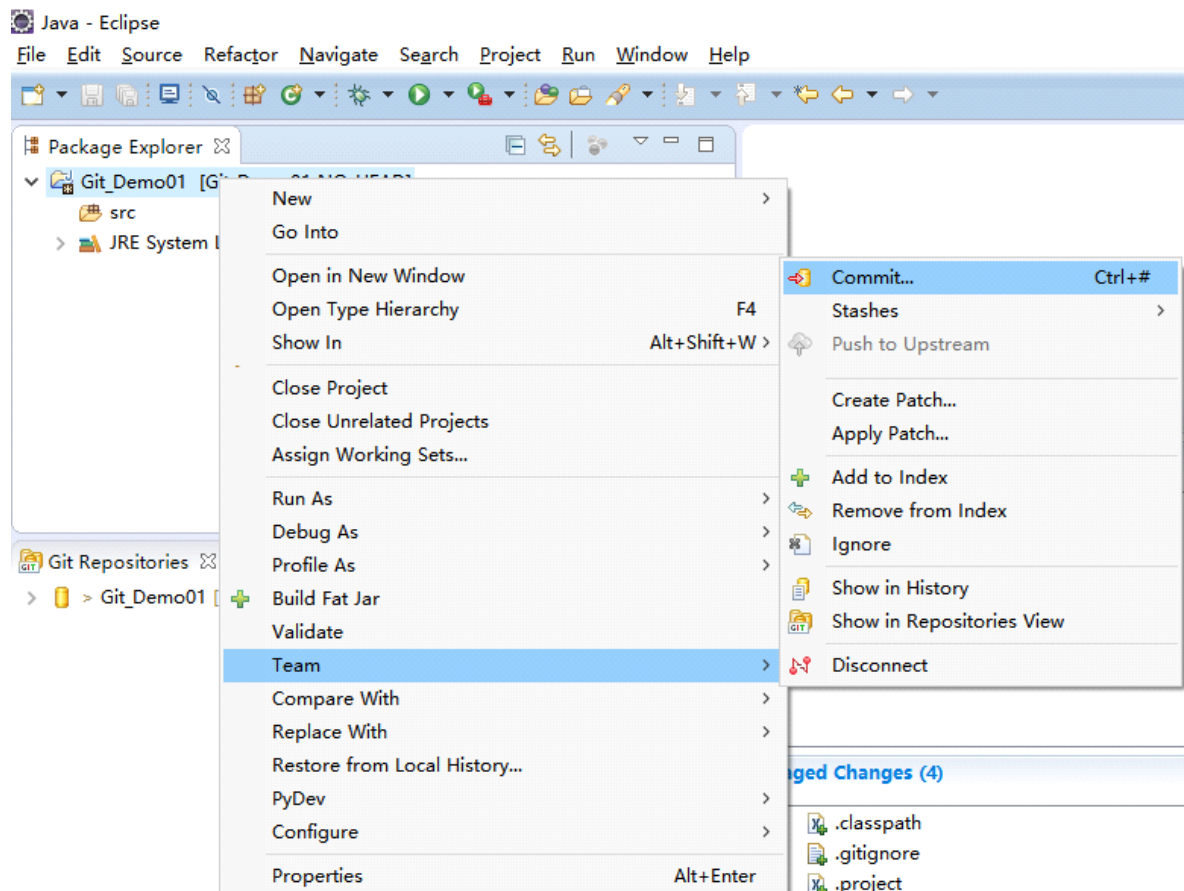
3. 增加到缓存区

Team->Add To Index



4. 提交到分支区

Team->Commit



5. 一步提交版本(增加到暂存区+提交到分支区)

Team->Commit

在进行提交操作时，EGIT允许直接勾选 [未跟踪] 状态的文件，此时，EGit会自动将该类型文件Add To Index 后再 Commit

6. 查看版本信息

Team->Show In History

The screenshot shows the Eclipse IDE interface with the 'Git Reflog' tab selected. The top panel displays a table of commit history for 'Project: Git_Demo01 [Git_Demo01]'. The table has columns: Id, Message, Author, Authored Date, Committer, and Committed Date. The first commit is '4dde9ef' with message 'create project', authored by 'parkx' on '2019-01-10 13:22:18'. Below the table, the commit details are shown, including the commit hash, author, committer, and branches. The bottom panel shows the 'Git Demo01' project with a search bar and a table of commit history.

Id	Message	Author	Authored Date	Committer	Committed Date
4dde9ef	create project	parkx	2 minutes ago	parkx	2 minutes ago

commit 4dde9efbc48f428b63cc3d21b15b0f65a19013
Author: parkx <piaoqian@tedu.cn> 2019-01-10 13:22:18
Committer: parkx <piaoqian@tedu.cn> 2019-01-10 13:22:18
Branches: master

create project

Commit	Commit Message	Date	Reflog Message
4dde9ef	create project	2019-01-10 13:22:18	commit (initial): create project

7. 增加文件到本地仓库

Team->Add To Index

Team->Commit

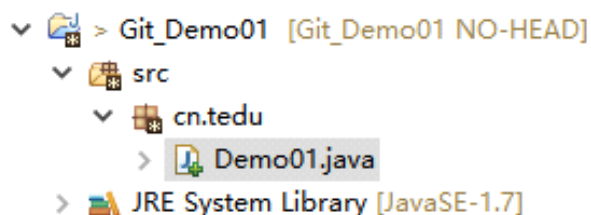
Team->Commit

创建文件

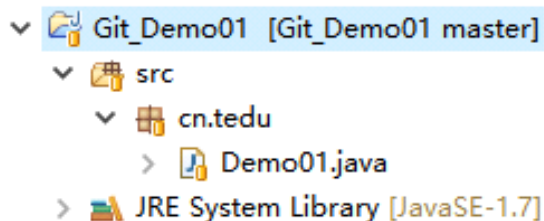
The screenshot shows the Eclipse IDE interface with the 'Git_Demo01 [Git_Demo01 NO-HEAD]' project selected. The project structure is shown in the left pane, with the 'src' folder expanded, showing the 'cn.tedu' package and the 'Demo01.java' file. The code editor shows the content of 'Demo01.java'.

```
1 package cn.tedu;  
2  
3 public class Demo01 {  
4     public static void main(String[] args) {  
5         System.out.println("aaa");  
6     }  
7 }
```

增加到暂存区 [Team->Add To Index]



提交到分支区 [Team->Commit]



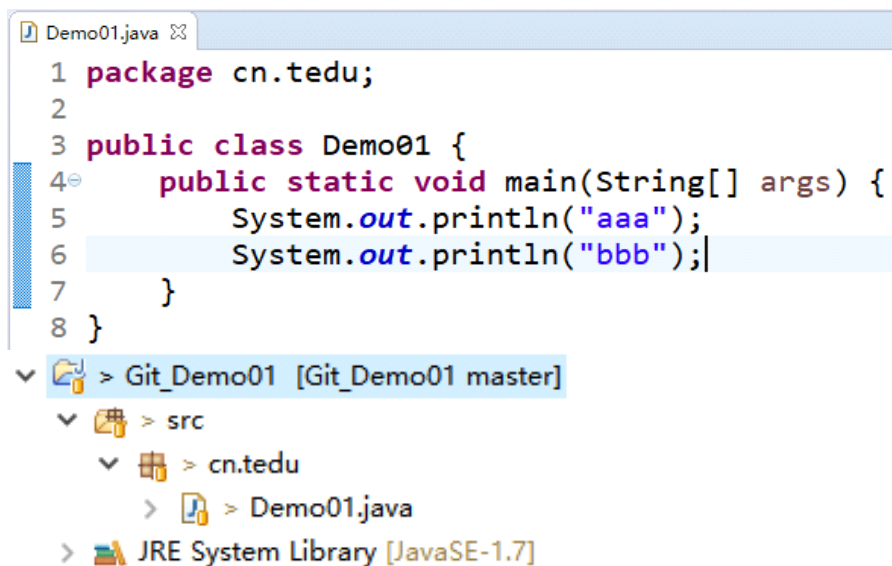
8. 提交修改到本地仓库

Team->Add To Index

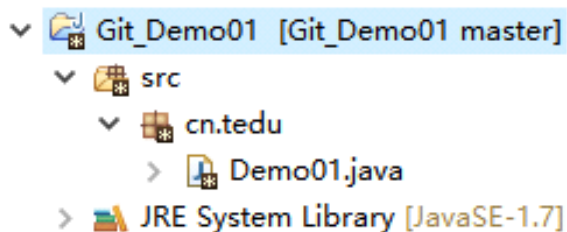
Team->Commit

Team->Commit

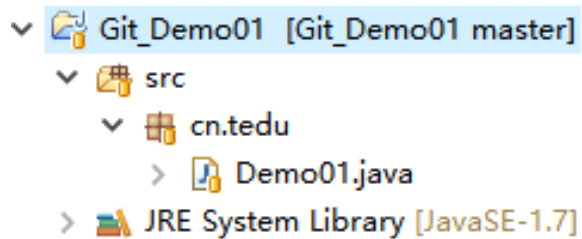
修改文件



增加到暂存区 [Team->Add To Index]



提交到分支区 [Team->Commit]

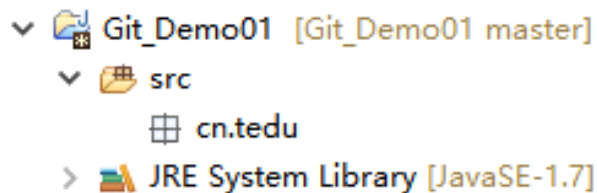


9. 删除文件从本地仓库

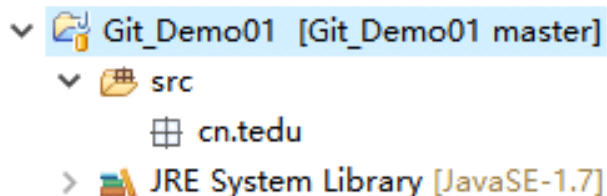
删除文件

Team->Commit

直接删除文件



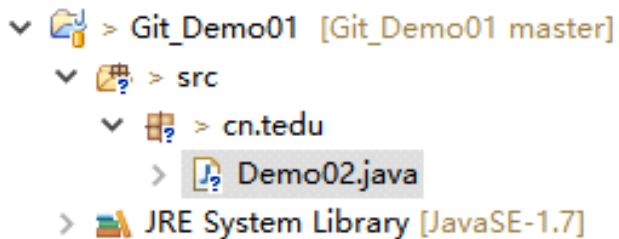
提交到分支区 [Team->Commit]



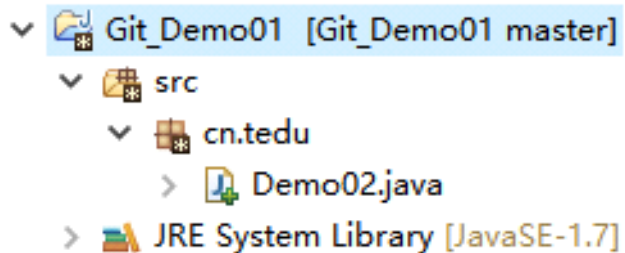
10. 从暂存区中撤销

Team->Remove From Index

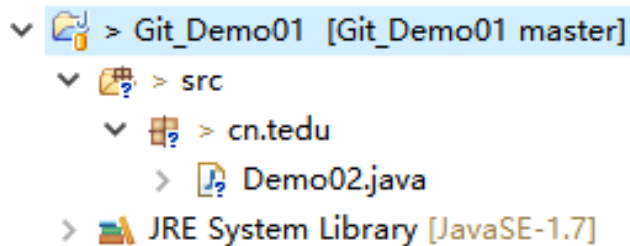
修改工作区



增加到暂存区 [Team->Add To Index]



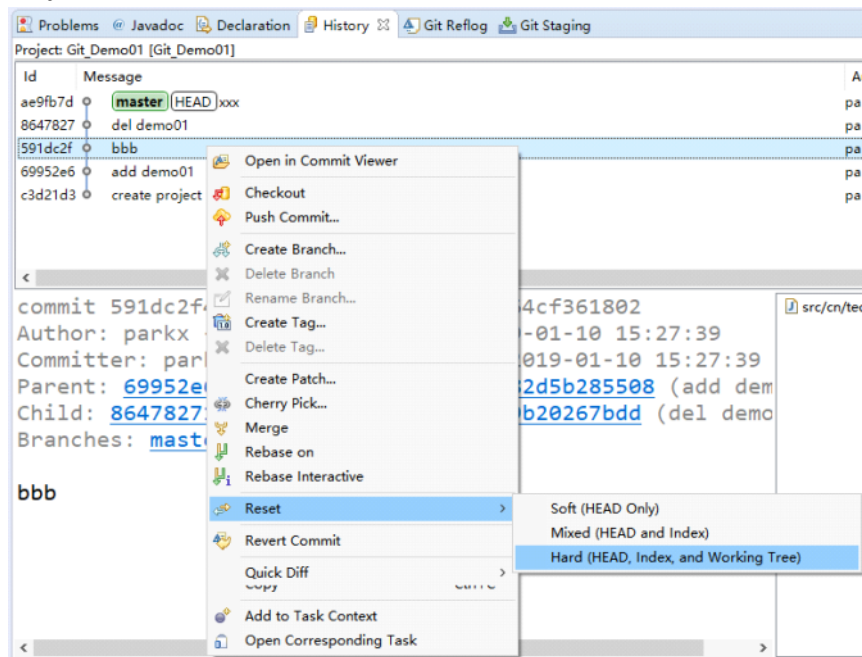
从暂存区中撤销 [Team->Remove From Index]



11. 基于分支树回滚版本

[History窗口中]Reset->Soft、Mixed、Hard

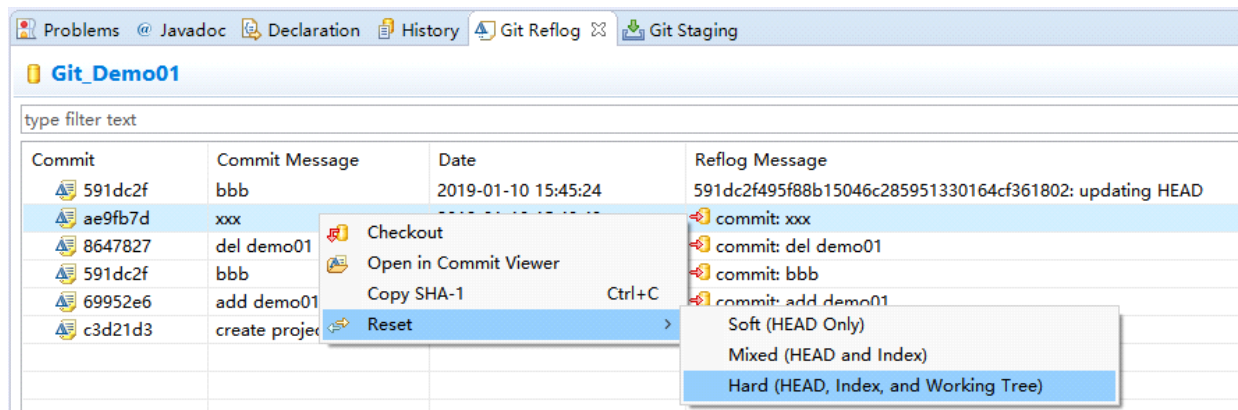
History窗口中选择目标版本并回滚 [Reset->Soft、Mixed、Hard]



12. 基于操作历史回滚版本

[GitRef窗口中]Reset->Soft、Mixed、Hard

GitRef窗口中选择某个历史指令并回滚到该指令执行过后的状态 [Reset->Soft、Mixed、Hard]

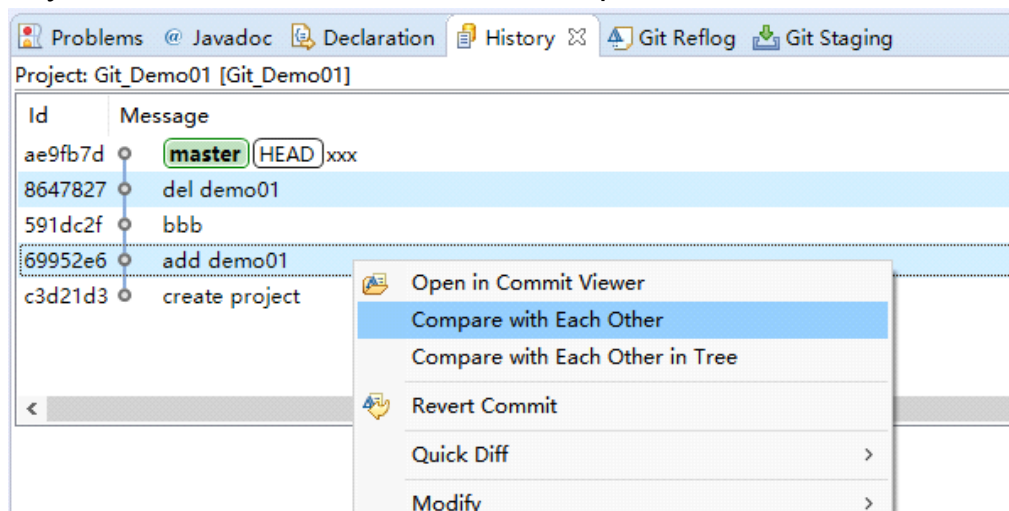


13. 文件版本比较

a. 两个历史版本之间的比较

[History窗口中，选择两个版本]->Compare With Each Other

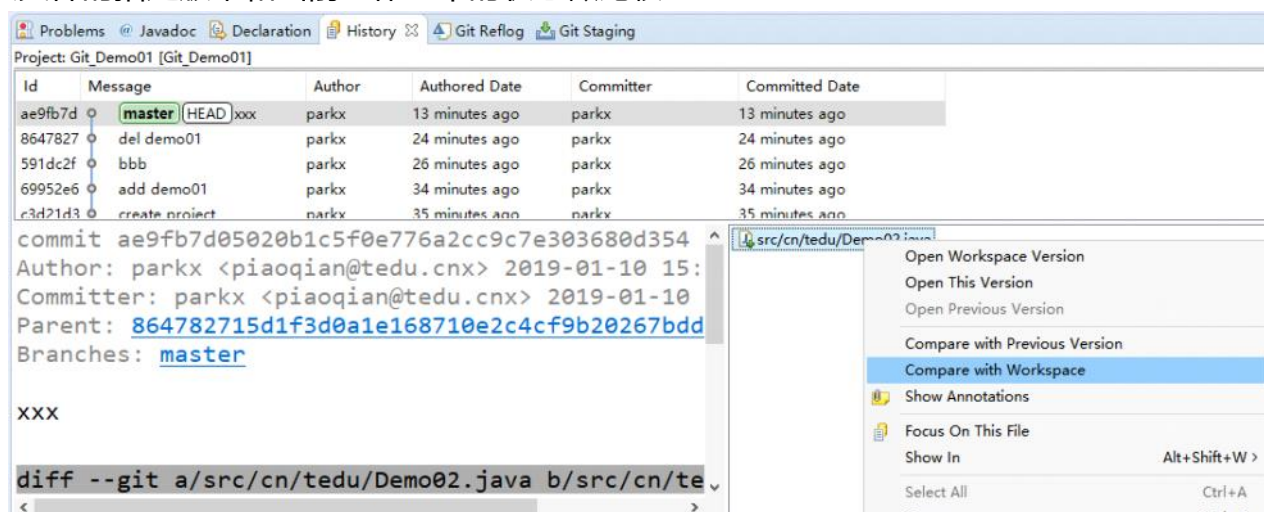
History窗口中，选择两个版本，并通过[Compare With Each Other]进行比较



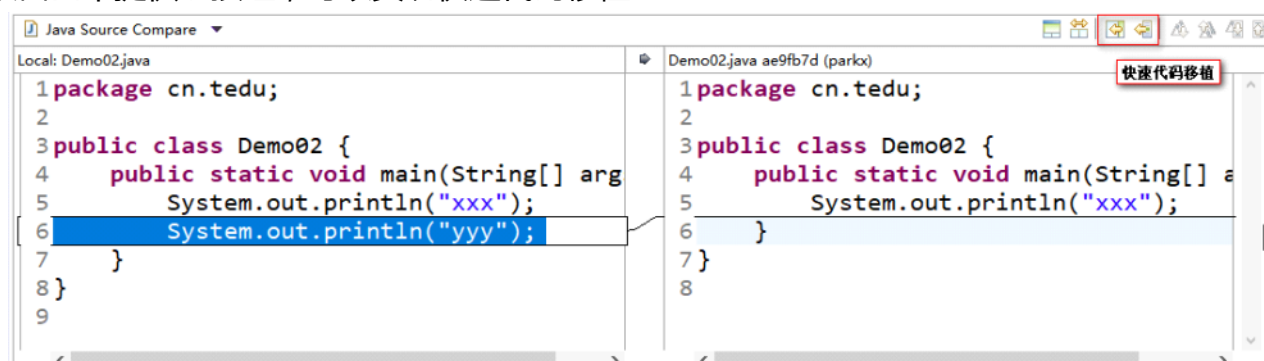
b. 指定历史版本和当前工作区中文件的比较

[History窗口中，选择某一个版本，选择其中一个文件]->Compare With WorkSpace

History窗口中，选择某一个版本，选择其中一个文件，并通过[Compare With WorkSpace]将该文件的指定版本和当前工作区中的状态做比较



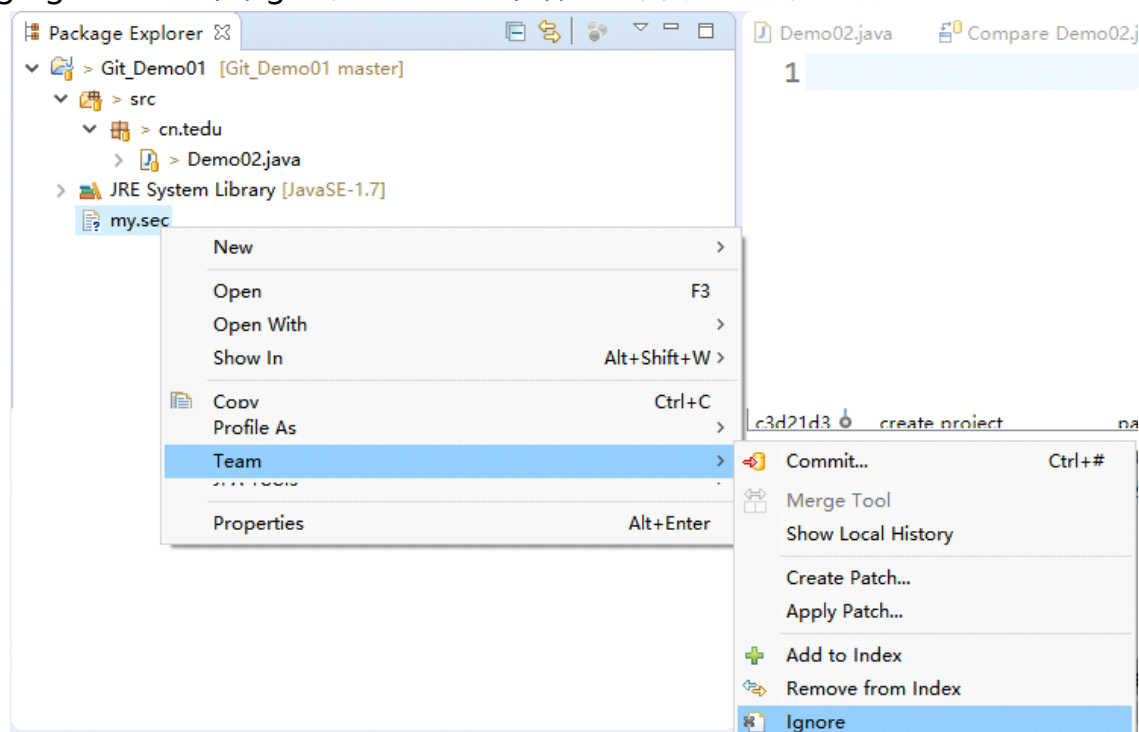
比较窗口中提供了按钮，可以实现快速代码移植



14. 忽略文件

Team->Ignore

可以通过[Team->Ignore]命令Git忽略指定文件，此命令本质上会将该文件位置信息写入.gitignore文件中，git会根据此文件中指定的规则忽略相应文件



因此也可以直接修改.gitignore指定忽略文件规则

此电脑 > 工作 (E:) > resource > git_resource > Git_Demo01				
名称	修改日期	类型	大小	
.git	2019/1/10 15:47	文件夹		
.settings	2019/1/10 15:18	文件夹		
bin	2019/1/10 15:19	文件夹		
src	2019/1/10 15:47	文件夹		
.classpath	2019/1/10 15:18	CLASSPATH 文件	1 KB	
.gitignore	2019/1/10 15:18	文本文档	1 KB	
.project	2019/1/10 15:18	PROJECT 文件	1 KB	
my.sec	2019/1/10 16:01	SEC 文件	0 KB	

.gitignore文件格式：

- 1) **空格**不匹配任意文件，可作为分隔符，可用反斜杠转义
- 2) 以“**#**”开头的行都会被 Git 忽略。即#开头的文件标识注释，可以使用反斜杠进行转义。
- 3) 可以使用标准的**glob**模式匹配。所谓的glob模式是指shell所使用的简化了的正则表达式。
- 4) 以斜杠“**/**”开头表示目录；“**/**”结束的模式只匹配文件夹以及在该文件夹路径下的内容，但是不匹配该文件；“**/**”开始的模式匹配项目跟目录；如果一个模式不包含斜杠，则它匹配相对于当前 .gitignore 文件路径的内容，如果该模式不在 .gitignore 文件中，则相对于项目根目录。
- 5) 以星号“*****”通配多个字符，即匹配多个任意字符；使用两个星号“******”表示匹配任意中间目录，比如`a/**/z`可以匹配 a/z, a/b/z 或 a/b/c/z等。

6) 以问号"?"通配单个字符，即匹配一个任意字符；

7) 以方括号"[]"包含单个字符的匹配列表，即匹配任何一个列在方括号中的字符。比如[abc]表示要么匹配一个a，要么匹配一个b，要么匹配一个c；如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配。比如[0-9]表示匹配所有0到9的数字，[a-z]表示匹配任意的小写字母）。

8) 以叹号"!"表示不忽略(跟踪)匹配到的文件或目录，即要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号(!)取反。需要特别注意的是：**如果文件的父目录已经被前面的规则排除掉了，那么对这个文件用"!"规则是不起作用的**。也就是说"!"开头的模式表示否定，该文件将会再次被包含，如果排除了该文件的父级目录，则使用"!"也不会再次被包含。可以使用反斜杠进行转义。

需要谨记：**git对于.ignore配置文件是按行从上到下进行规则匹配的，意味着如果前面的规则匹配的范围更大，则后面的规则将不会生效；**

*EGIT - 远程版本控制

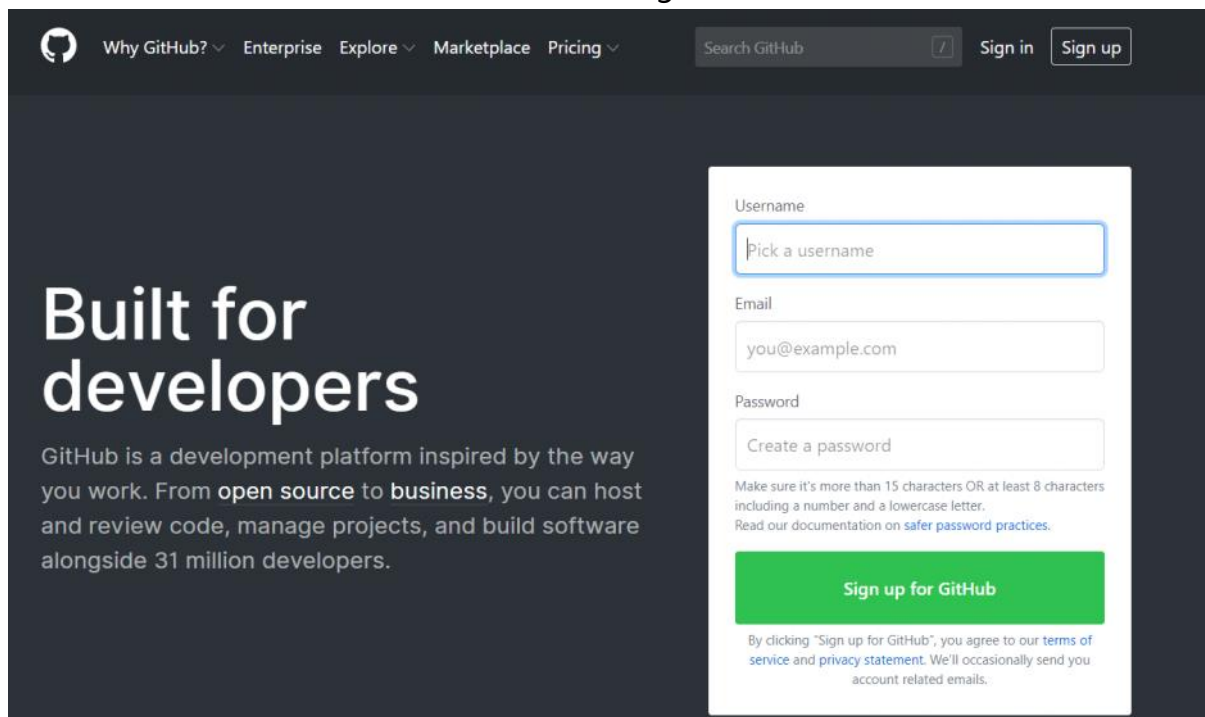
2019年1月10日 17:18

1. 代码托管站点

基于Git这一优秀的分布式版本控制工具，市面上出现了大量基于Git的代码托管网站，开发者可以选择使用代码托管网站作为分布式版本控制中的“中央服务器”，从而非常便利的实现代码托管、分布式开发、项目管理等功能，省去了自己搭设中心服务器的麻烦。

目前，许多开源项目也基于代码托管网站进行管理，随着越来越多的人参与和使用，Git代码托管网站早已不再只是基本的版本管理工具，而成为了一个完整的开发者社区，在开源领域有着重要的作用。

目前有很多代码托管网站，其中最知名的是GitHub，托管着大量的私有或开源的项目，其他国外较为知名的代码托管网站包括GitLab、BitBucket等，但这些代码托管网站通常架设在国外，国内访问并不稳定，而国内也有相关产品，例如 码云、coding等，在国内也有较为广泛的应用。



2. GitEE码云

码云(gitee.com)是开源中国推出的代码托管平台,支持 Git 和 SVN,提供免费的私有仓库托管。目前已成为国内最大的代码托管系统，有超过 300 万的开发者选择码云。

a. 注册码云

访问码云官网<https://gitee.com/>，注册并登陆



稳定 · 高效 · 安全

码云 Gitee - 云端软件开发协作平台

帮助个人、团队、企业轻松实现 Git/SVN 代码托管、协作开发

加入码云

免费开通企业版

我已经有GitHub了, 为什么要用码云? 如何一键导入 GitHub 仓库?

b. 在码云上创建仓库



c. 配置仓库基本信息

新建仓库

仓库名称

仓库名称

归属

pcj2619

路径

https://gitee.com/pcj2619/

仓库路径，如果不手动指定会采用仓库名自动生成路径

仓库介绍 非必填

用简短的语言来描述一下吧

是否开源

☒ 公开

☐ 私有

是否开源，如果选择公开则任何人都可以访问此项目，如果选择私有，则只有具有授权者才可以访问

选择语言

请选择语言

添加 .gitignore

请选择 .gitignore 模板

添加开源许可证 ^①

请选择开源许可证

☒ 使用Readme文件初始化这个仓库

☐ 使用Issue模板文件初始化这个仓库 ^①

☐ 使用Pull Request模板文件初始化这个仓库 ^①

 导入已有仓库

创建

d. 获取项目地址

快速设置— 如果你知道该怎么操作，直接使用下面的地址

HTTPS

SSH

https://gitee.com/pcj2619/git_demo01.git

GITEE提供了HTTPS和SSH两种访问方式，点击可以复制对应地址

我们强烈建议所有的git仓库都有一个 README， LICENSE， .gitignore 文件

Git入门？查看 [帮助](#)，[Visual Studio](#) / [TortoiseGit](#) / [Eclipse](#) / [Xcode](#) 下如何连接本站，[如何导入仓库](#)

3. EGit推送代码到远程仓库 - HTTPS方式

a. 提交本地代码

Team->Commit

b. 推送本地仓库代码到远程仓库

Team->Remote->Push

Push to Another Repository

Destination Git Repository

Enter the location of the destination repository.

Location

URI: 远程仓库地址，HTTPS格式

Host:

Repository path:

Connection

Protocol: 使用的协议，选择https

Port:

Authentication

User: 远程仓库用户名密码

Password:

☒ Store in Secure Store

?

< Back

Next >

Finish

Cancel

Push to: https://gitee.com/pcj2619/git_demo01.git

Push Ref Specifications

Select refs to push.

Add create/update specification

Source ref: Destination ref: 选择要提交哪个分支,点击 +Add Spec

+ Add Spec

Add delete ref specification

Remote ref to delete:

Add Spec

Add predefined specification

Add Configured Push Specs

Add All Branches Spec

Add All Tags Spec

Specifications for push

Mode	Source Ref	Destination Ref	Force Update	Remove
+ Update	refs/heads/master	refs/heads/master	<input checked="" type="checkbox"/>	<input type="checkbox"/>

要提交的分支列表，如果是第一次提交，勾选ForceUpdate

Force Update All Specs

Remove All Specs

?

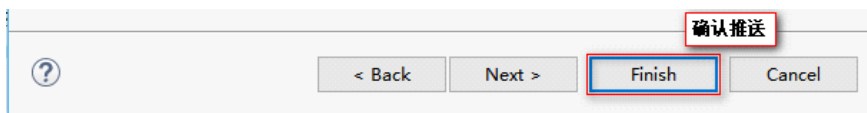
< Back

Next >

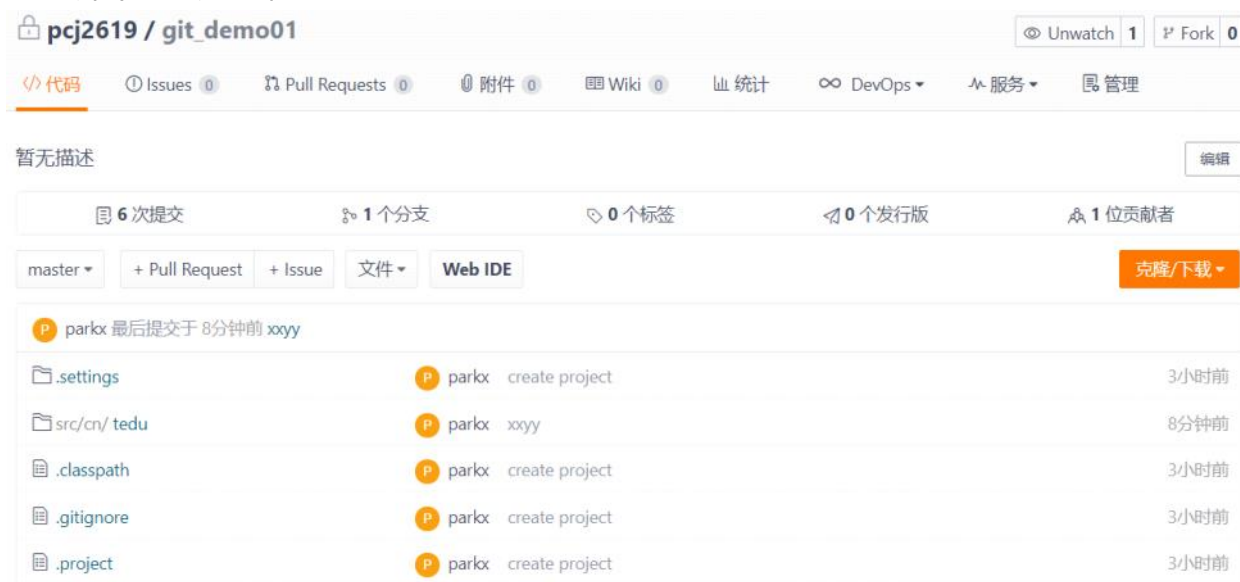
Finish

Cancel

确认推送



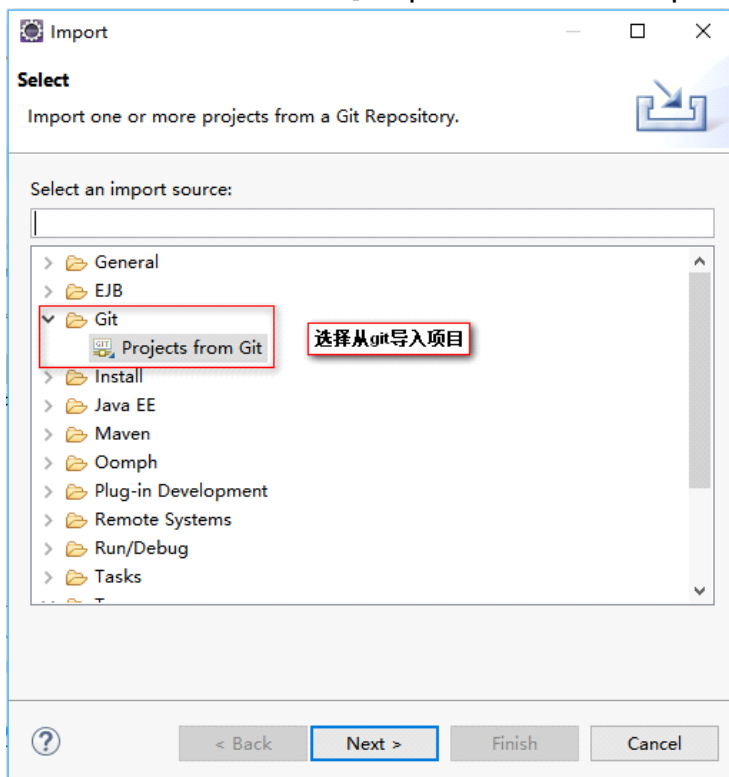
c. 在远程仓库中检查是否推送成功

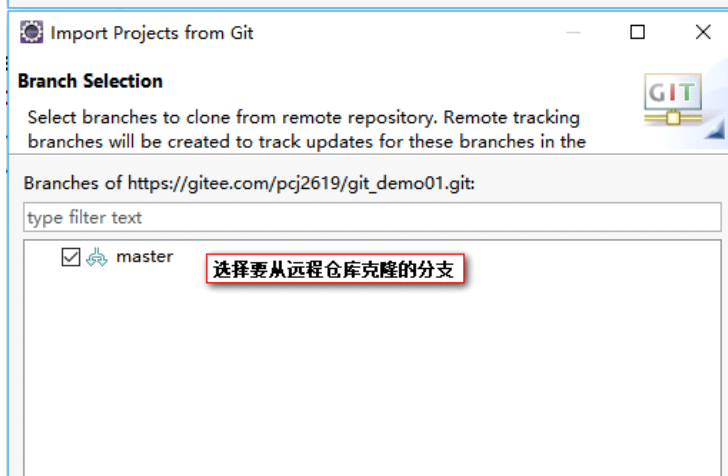
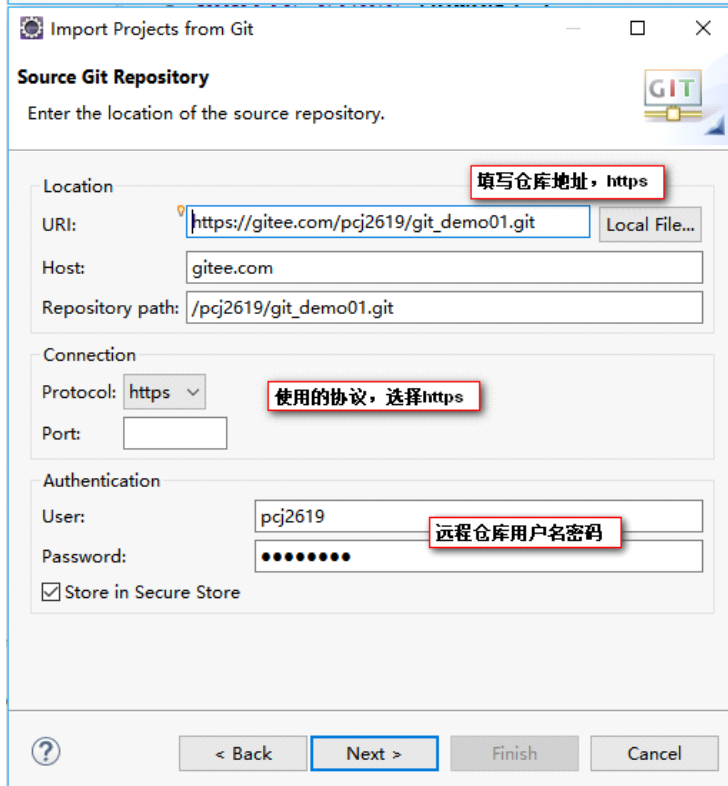
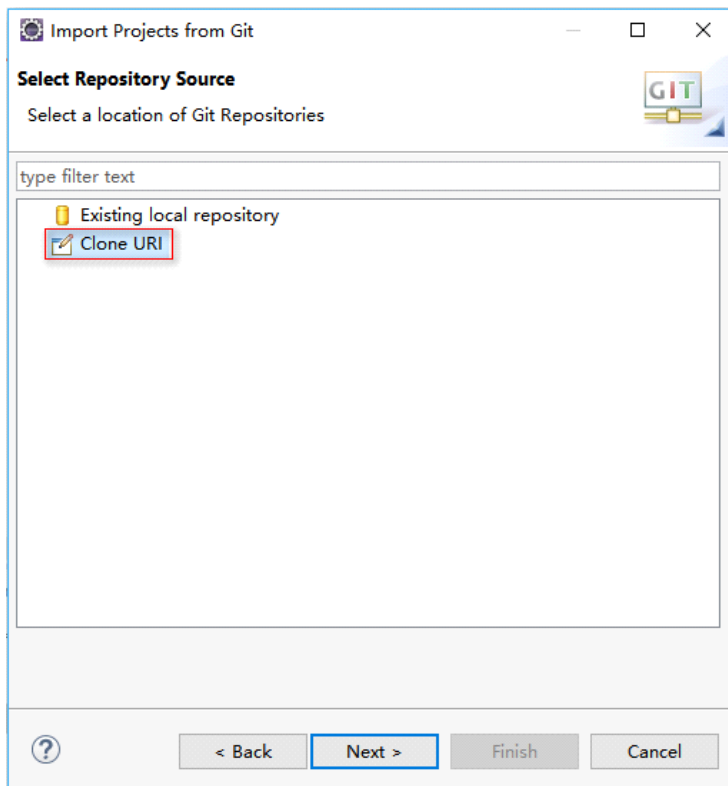


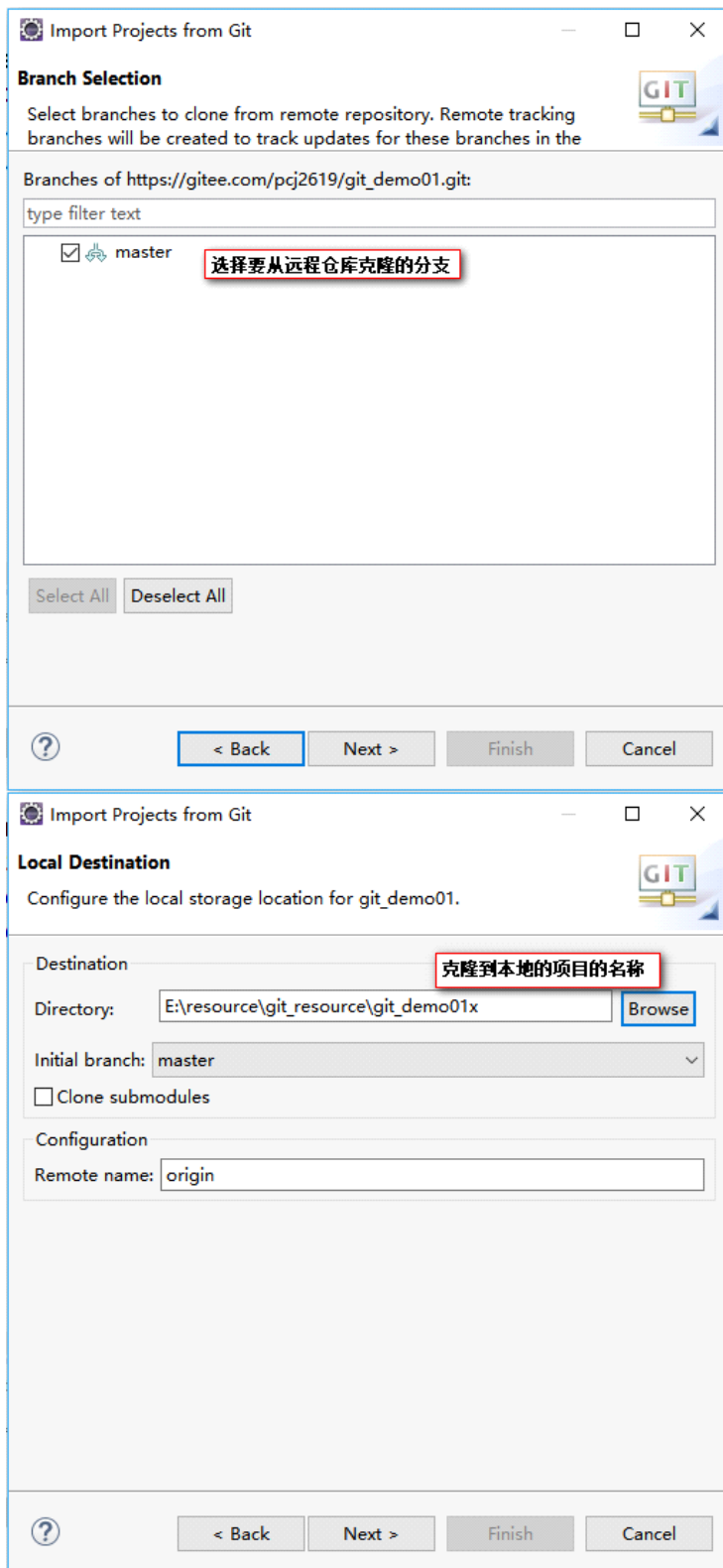
4. EGit从远程仓库克隆代码到本地仓库 - HTTPS方式

[Eclipse菜单 - File - Import]

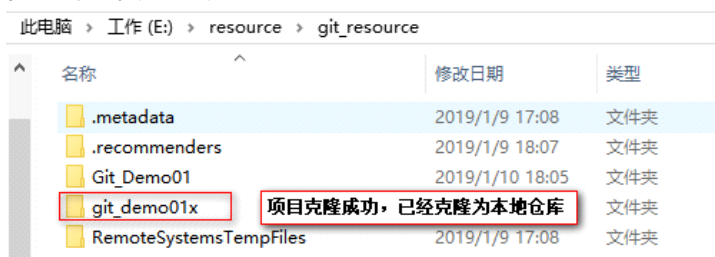
从远程仓库中克隆项目，选择[Eclipse菜单 - File - Import]



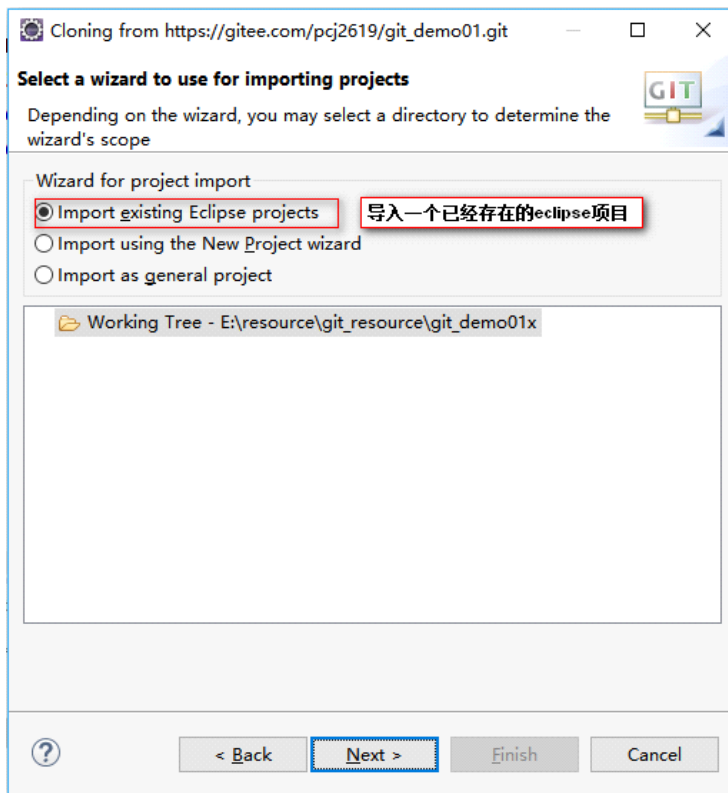




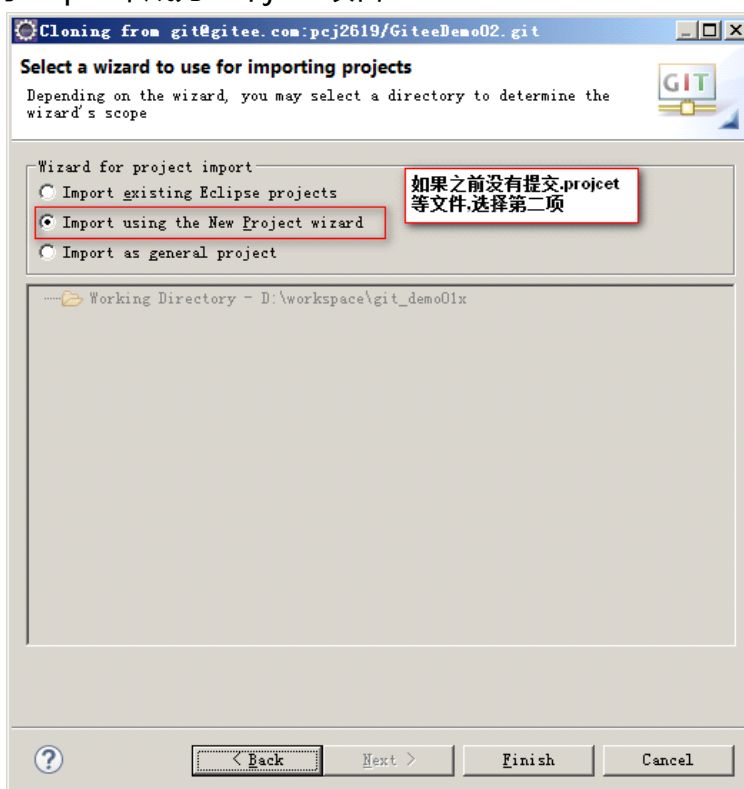
点击下一步，克隆项目成功

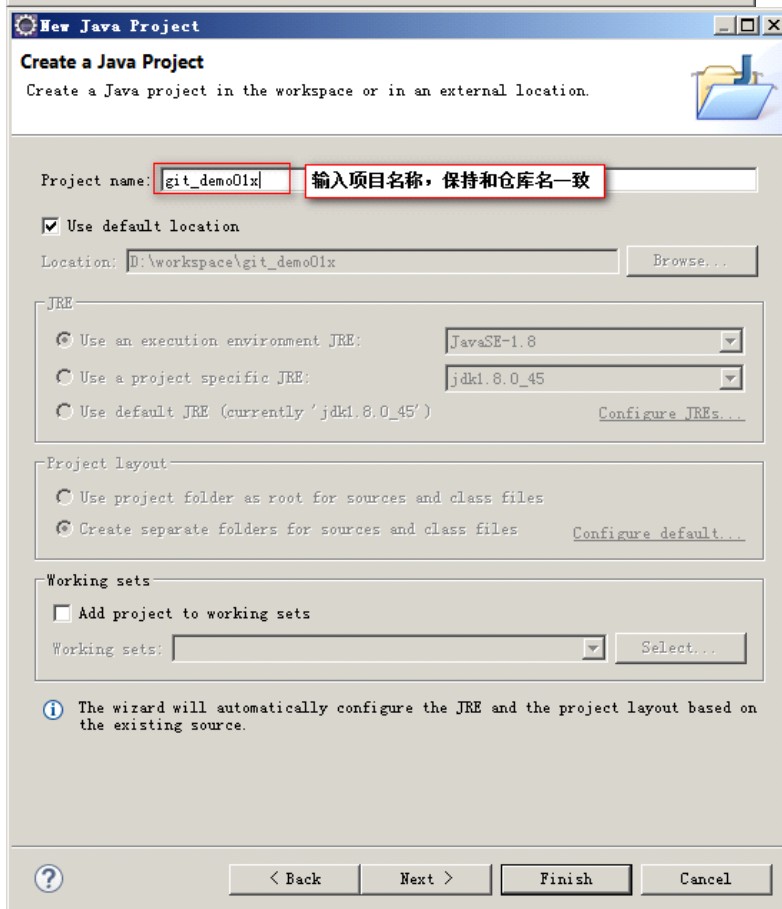
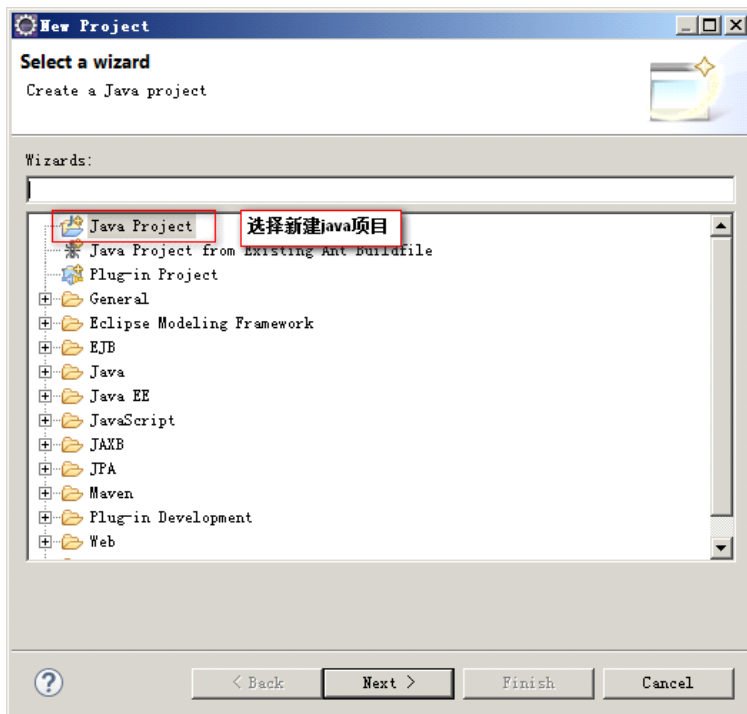


如果之前提交时，提交过.project和.setting等文件，选择第一项可以进一步导入该仓库中的代码到Eclipse中成为一个java项目

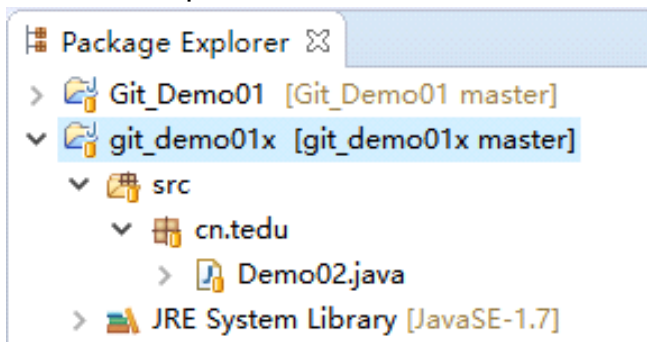


如果之前提交时，没有提交过.project和.setting等文件，选择第二项，依次填写信息即可将项目导入到Eclipse中成为一个java项目





导入成功，在Eclipse下使用该项目



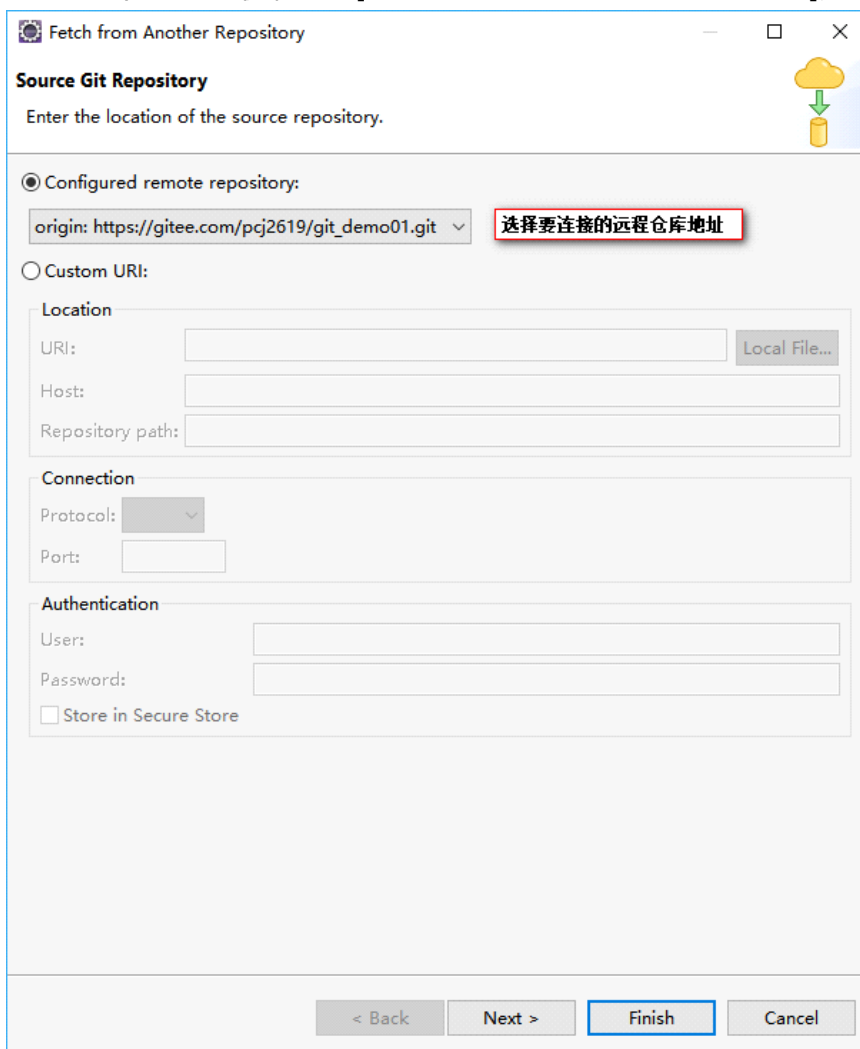
5. EGit从远程仓库拉取代码 - HTTPS方式

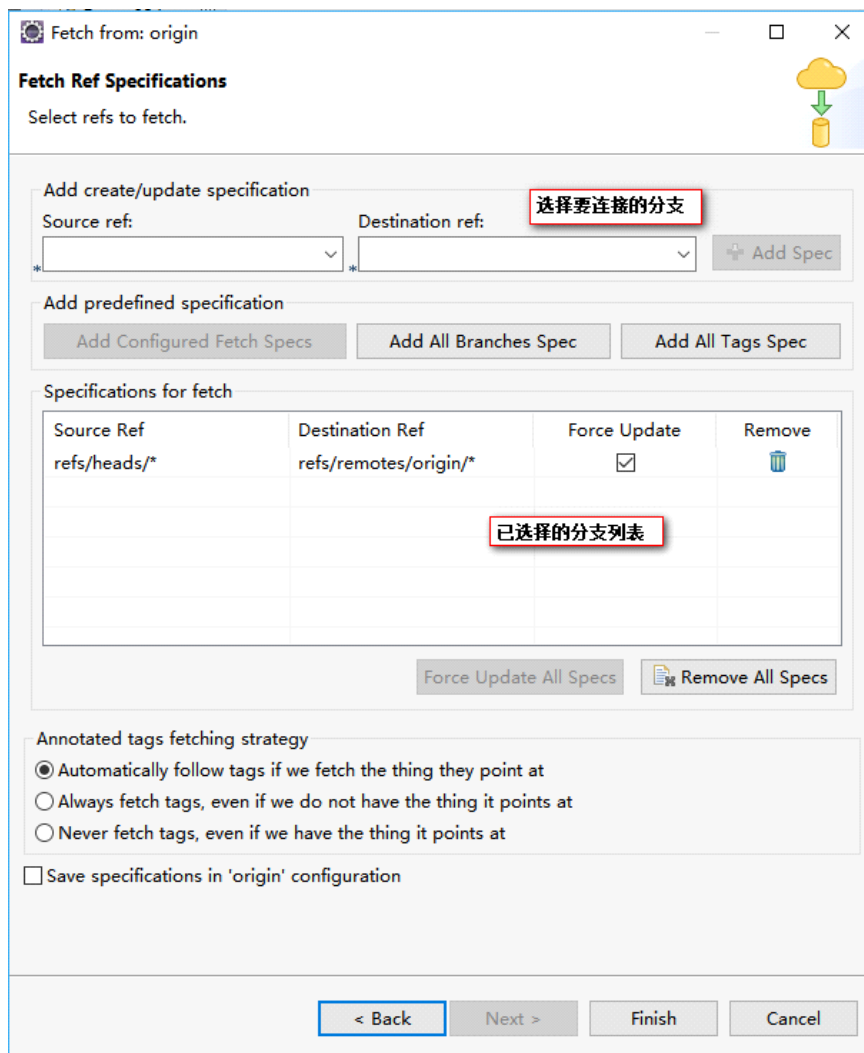
Team->Remote->Fetch From..

当远程仓库中有高于本地版本的更新内容时，可以在本地仓库通过拉取操作获取该版本内容。
为了方便测试，在远程仓库中，创建一个新文件

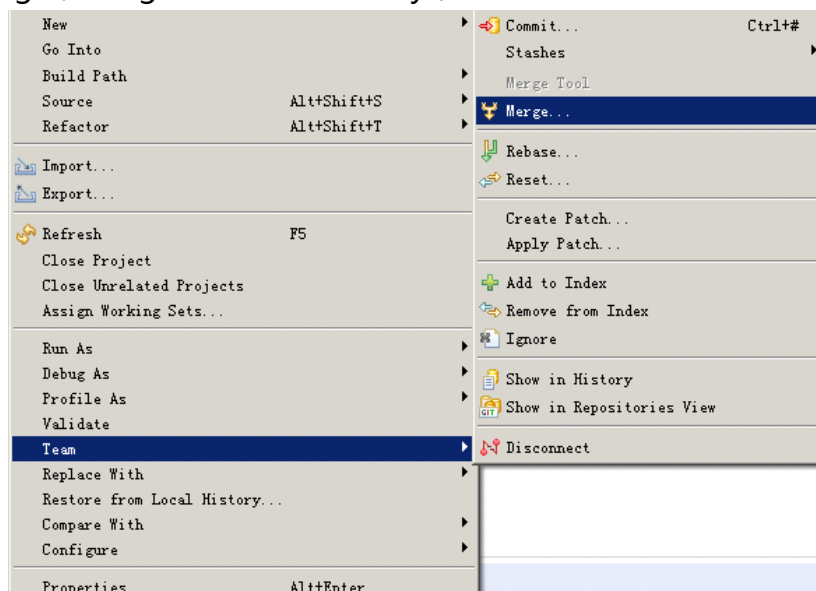


尝试从EGIT中进行拉取，通过[Team->Remote->Fetch From..]开始拉取

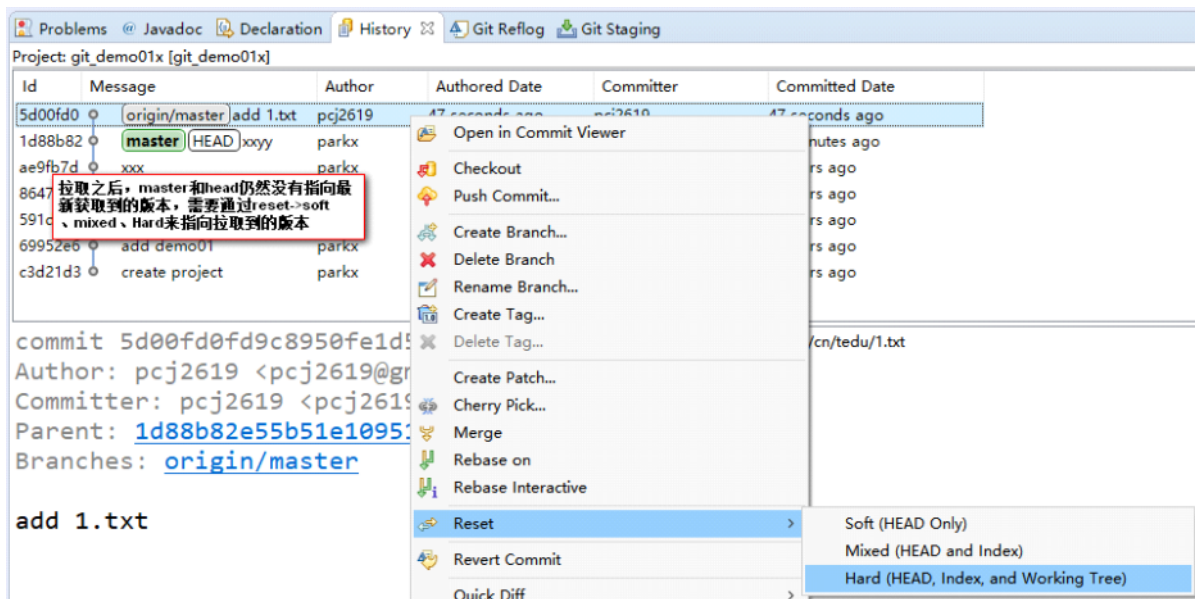




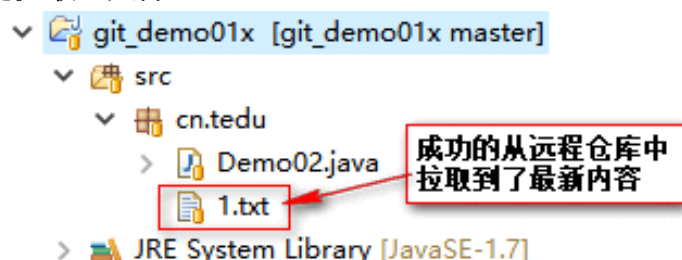
拉取之后在History里可以看到从服务器中拉取到的版本，如果没有，则可以在项目上进行team->merge，merge之后就会在history中显示



但master和HEAD仍未指向该版本，通过[Rest->Soft、Mixed、Hard]来进行指向



成功拉取了文件



6. 使用SSH方式连接远程仓库

a. 非对称加密简介

非对称加密算法是一种密钥的保密方法。

非对称加密算法需要两个密钥：[公开密钥](#)（publickey）和私有密钥（privatekey）。公开密钥与私有密钥是一对，如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能解密。因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。非对称加密算法实现机密信息交换的基本过程是：甲方生成一对[密钥](#)并将其中的一把作为公用密钥向其它方公开；得到该公用密钥的乙方使用该密钥对机密信息进行加密后再发送给甲方；甲方再用自己保存的另一把专用密钥对加密后的信息进行解密。

另一方面，甲方可以使用乙方的公钥对机密信息进行签名后再发送给乙方；甲方再用自己的私钥对乙方发送回来的数据进行验签。

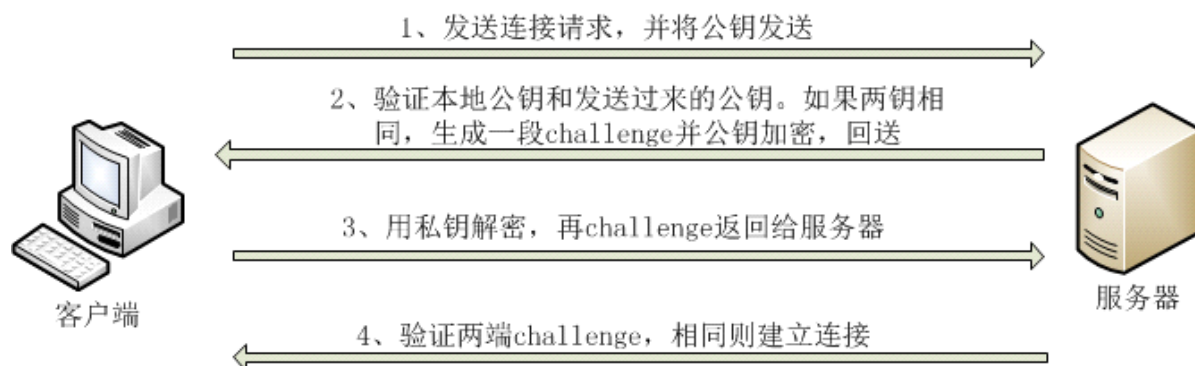
甲方只能用其专用密钥解密由其公用密钥加密后的任何信息。非对称加密算法的保密性比较好，它消除了最终用户交换密钥的需要。

非对称密码体制的特点：算法强度复杂、安全性依赖于算法与密钥但是由于其算法复杂，而使得加密解密速度没有对称加密解密的速度快。对称密码体制中只有一种密钥，并且是非公开的，如果要解密就得让对方知道密钥。所以保证其安全性就是保证密钥的安全，而非对称密钥体制有两种密钥，其中一个是公开的，这样就可以不需要像对称密码那样传输对方的密钥了。这样安全性就大了很多。

非对称加密主要算法有：[RSA](#)、[Elgamal](#)、背包算法、Rabin、D-H、[ECC](#)（椭圆曲线加密算

法)等

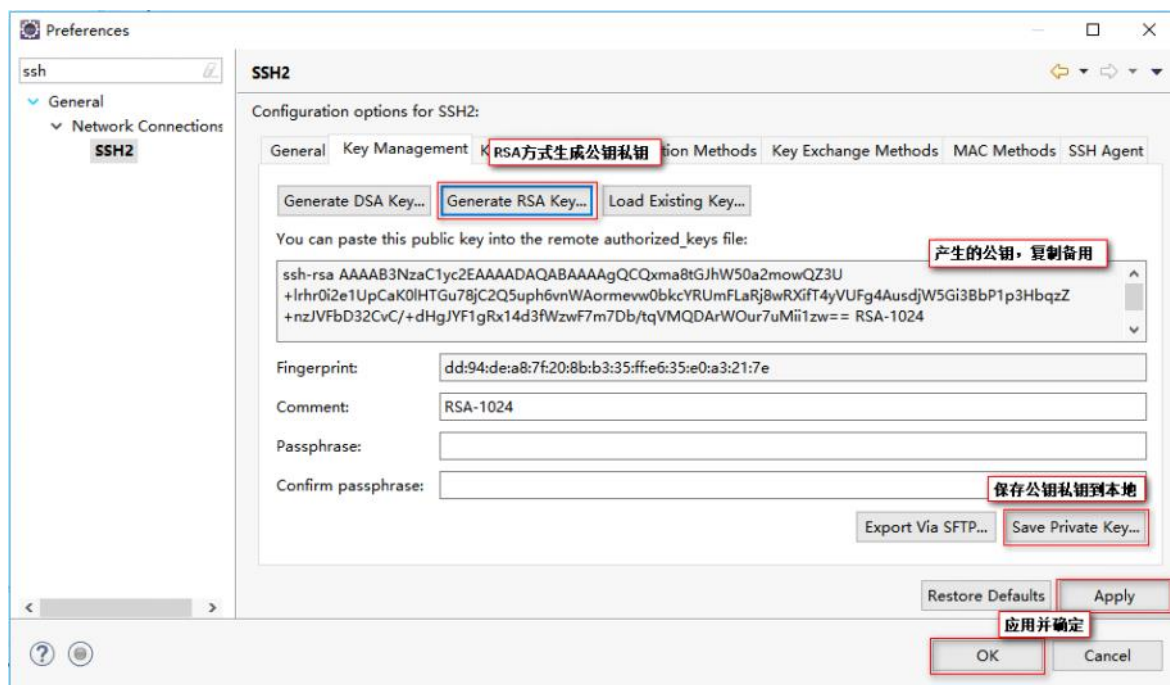
b. SSH免密登录原理



c. SSH方式免密连接远程仓库

以上的案例中均是使用HTTPS方式连接的远程仓库，但通常远程仓库通常都提供SSH方式的连接，且SSH方式以非对称加密为原理可以实现免密登录，从而可以避免频繁的输入用户名密码，是一种更加安全、便捷的连接方式，目前应用的非常广泛，推荐使用。

i. 本地生成公钥私钥对



ii. 在远程仓库中配置公钥

G 码云 开源软件 企业版 特选 高校版 博客 我的码云

搜索项目、代码片段

个人主页 设置 代码片段 帮助 退出

pcj2619 加入时间 7月前

SSH公钥

使用SSH公钥可以让你在你的电脑和码云通讯的时候使用安全连接（Git的Remote要使用SSH地址）

您当前的SSH公钥数: 0

你还没有添加任何SSH公钥

添加公钥

标题

park_001 为公钥起一个名字方便管理

公钥

把你的公钥粘贴到这里，查看 怎样生成公钥

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQgQCQxma8tGjhW50a2mowQZ3U+lrhr0i2e1UpCaK0lHTGu78jC2Q5uph6vnWAormeww0bkcyRUmFLaRj8wRXifT4yVUFg4AusdjW5Gi3BbP1p3HbqzZ+nz/VFbD32CvC/+dHgJYF1gRx14d3fWzwF7m7Db/tqVMQDArWOu7uMii1zw== RSA-1024
```

粘贴公钥

确定

基本设置

- 个人资料
- 修改密码
- 个性域名
- 通知设置

安全设置

- SSH公钥

数据管理

- 升级为组织
- 升级为企业版
- 私有仓库成员
- 代码风格
- 第三方应用
- 私人令牌

iii. 之后就可以通过SSH地址来免密连接远程仓库了

Push to Another Repository

Destination Git Repository

Enter the location of the destination repository.

☐ Configured remote repository:

origin: `https://gitee.com/pcj2619/git_demo01.git`

☒ Custom URI:

Location

URI: `gitee.com:pcj2619/git_demo01.git` Local File...

Host: `gitee.com`

Repository path: `pcj2619/git_demo01.git`

Connection

Protocol: `ssh`

Port: `22`

Authentication

User: `pcj2619`

Password: `ssh-keygen`

☐ Store in Secure Store

通过ssh方式连接，不需要输入用户名密码

远程库地址，SSH格式

< Back Next > Finish Cancel