**Egit**

# Git版本

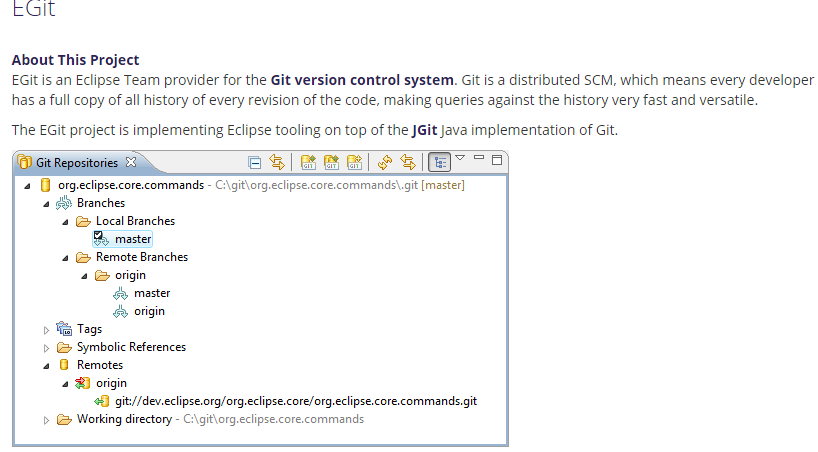


Git linux

Msysgit windwos

Egit java

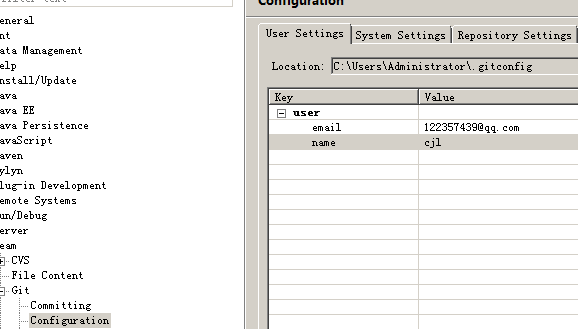
# Egit



官网教程：<http://wiki.eclipse.org/EGit/User_Guide>

官网书籍：<https://git-scm.com/book/en/v2>

## 初始配置

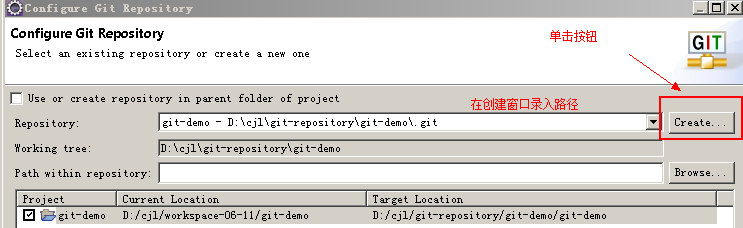


## 创建工程

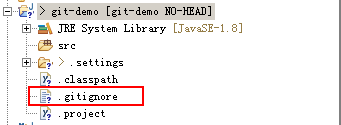
先创建一个普通的java工程

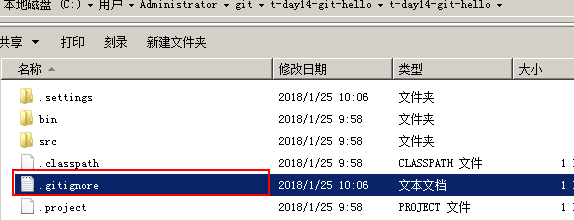


把工程交给egit去管理



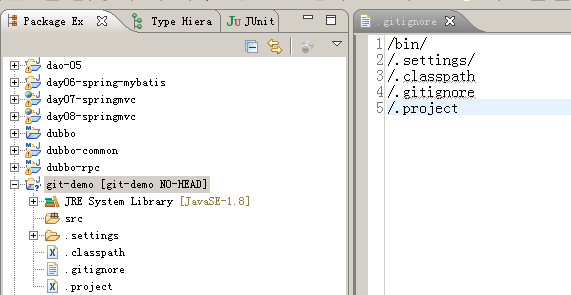
创建后如图





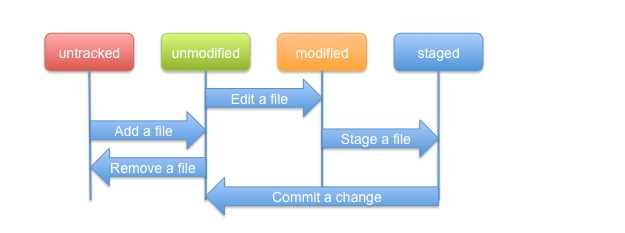
上面默认会创建一个git的忽略文件，这个文件的作用：

配置不需要被git版本库所管理的文件。

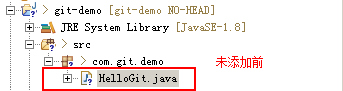


回顾下git原理

|  |
| --- |
| You start from a fresh checkout of a branch of a local repository. You want to do some changes and record snapshots of these changes in the repository whenever you reach a state you want to record.  Each file in the working directory can either be *tracked* or *untracked*:   * **Tracked** files are those which were in the last snapshot or files which have been newly staged into the *index*. They can be *unmodified*, *modified*, or *staged*. * **Untracked** files are all other files (they were not in the last snapshot and have not yet been added to the *index*).   When you first clone a repository, all files in the working directory will be *tracked* and *unmodified* since they have been freshly checked out and you haven't started editing them yet.  As you edit files, git will recognize they are *modified* with respect to the last commit. You *stage* the modified files into the index and then *commit* the staged changes. The cycle can then repeat.  This lifecycle is illustrated here: |



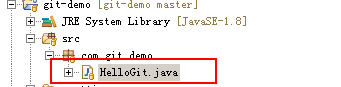
未添加状态：



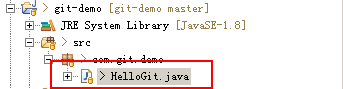
添加后状态：



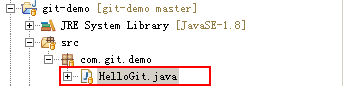
提交后状态：



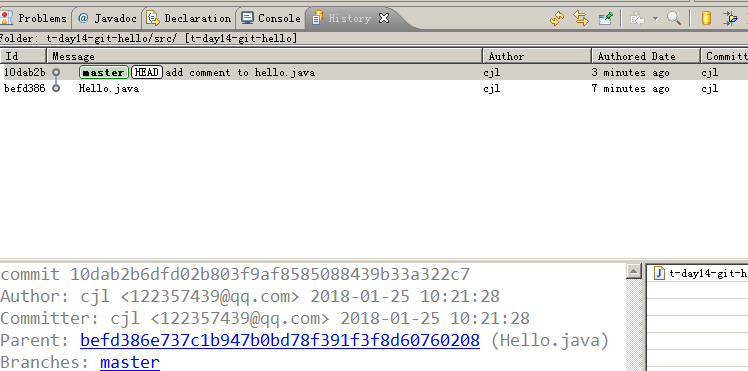
修改已提交文件



你要再次提交已修改文件

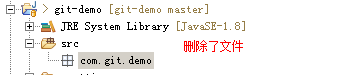


## 查看版本库

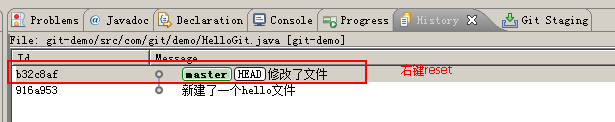


## 删除

如果只是在本地删除，恢复文件简单方法就是使用“revert”就可以，无污染，对版本库没有影响！！图像用不了！！



用“reset”可以恢复删除文件

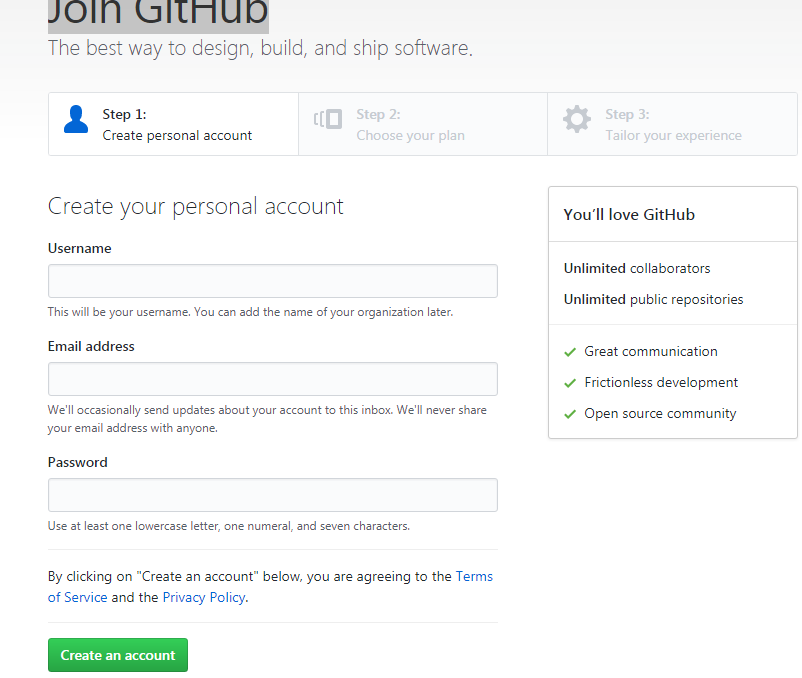


# 远程版本库

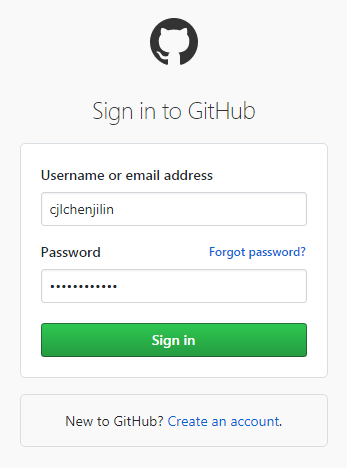
## 免费的

**Github，2018,6月被微软收购！！！**

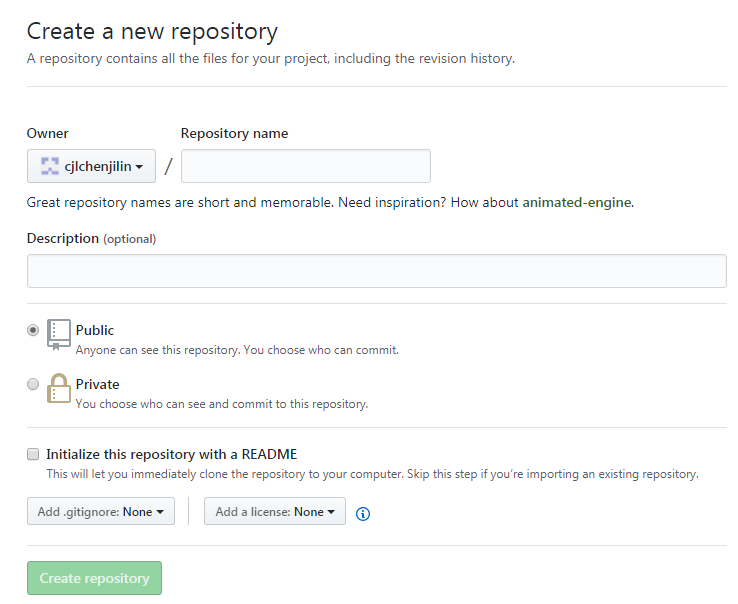
**注册界面**



登录界面



创建版本库



**oschina**

<https://gitee.com/>

用法和github类似！！

## 收费的

## 自己公司的

自己动手搭建可参考网上博客教程。。。

或者官网参考资料！！

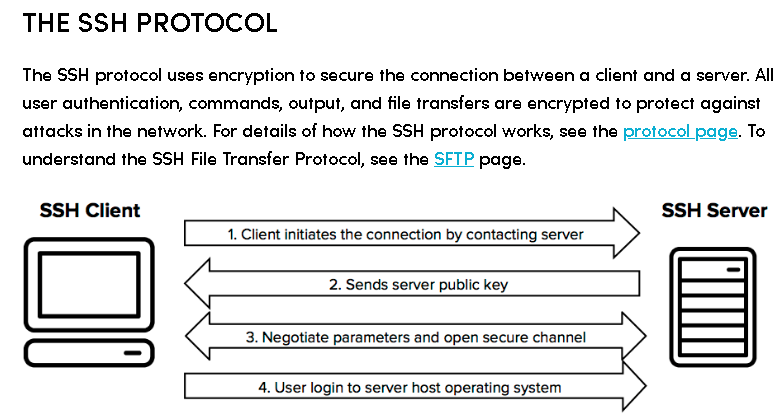
# Ssh

|  |
| --- |
| SSH 为 [Secure Shell](https://baike.baidu.com/item/Secure%20Shell) 的缩写，由 IETF 的网络小组（Network Working Group）所制定；SSH 为建立在应用层基础上的安全协议。SSH 是目前较可靠，专为[远程登录](https://baike.baidu.com/item/%E8%BF%9C%E7%A8%8B%E7%99%BB%E5%BD%95/1071998)会话和其他网络服务提供安全性的协议。利用 SSH 协议可以有效防止远程管理过程中的信息泄露问题。SSH最初是UNIX系统上的一个程序，后来又迅速扩展到其他操作平台。SSH在正确使用时可弥补网络中的漏洞。SSH客户端适用于多种平台。几乎所有UNIX平台—包括[HP-UX](https://baike.baidu.com/item/HP-UX)、[Linux](https://baike.baidu.com/item/Linux" \t "_blank)、[AIX](https://baike.baidu.com/item/AIX" \t "_blank)、[Solaris](https://baike.baidu.com/item/Solaris/3517" \t "_blank)、[Digital](https://baike.baidu.com/item/Digital" \t "_blank) [UNIX](https://baike.baidu.com/item/UNIX)、[Irix](https://baike.baidu.com/item/Irix" \t "_blank)，以及其他平台，都可运行SSH。 |

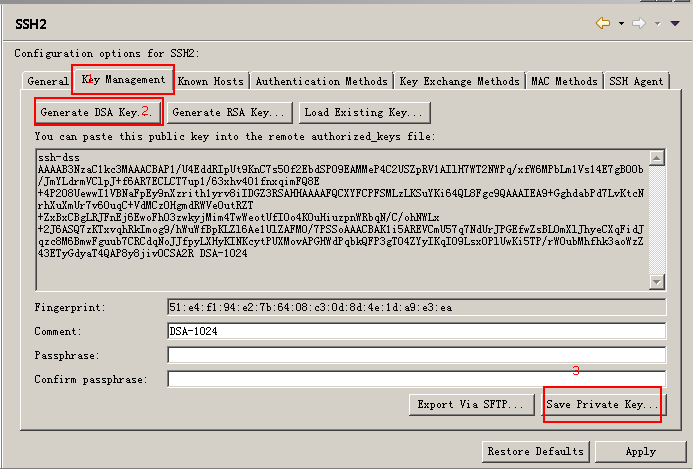
验证

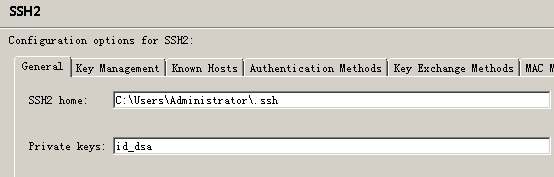
|  |
| --- |
| 从客户端来看，SSH提供两种级别的安全验证。  **第一种级别（基于口令的安全验证）**  只要你知道自己帐号和口令，就可以登录到远程主机。所有传输的数据都会被加密，但是不能保证你正在连接的服务器就是你想连接的[服务器](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E5%99%A8)。可能会有别的服务器在冒充真正的服务器，也就是受到“中间人”这种方式的攻击。  **第二种级别（基于密匙的安全验证）**  需要依靠[密匙](https://baike.baidu.com/item/%E5%AF%86%E5%8C%99)，也就是你必须为自己创建一对密匙，并把公用密匙放在需要访问的服务器上。如果你要连接到SSH服务器上，客户端软件就会向服务器发出请求，请求用你的密匙进行安全验证。服务器收到请求之后，先在该服务器上你的主目录下寻找你的公用密匙，然后把它和你发送过来的公用密匙进行比较。如果两个密匙一致，服务器就用公用密匙加密“质询”（challenge）并把它发送给客户端软件。客户端软件收到“质询”之后就可以用你的私人密匙解密再把它发送给服务器。  用这种方式，你必须知道自己密匙的[口令](https://baike.baidu.com/item/%E5%8F%A3%E4%BB%A4" \t "_blank)。但是，与第一种级别相比，第二种级别不需要在网络上传送口令。  第二种级别不仅加密所有传送的数据，而且“中间人”这种攻击方式也是不可能的（因为他没有你的私人密匙）。但是整个登录的过程可能需要10秒[2]  。 |

官网：<https://www.ssh.com/ssh/>

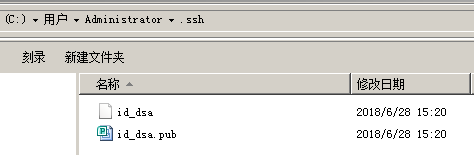


配置ssh(eclipse)





本地默认目录有一对



服务器端添加公匙



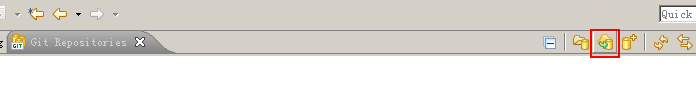


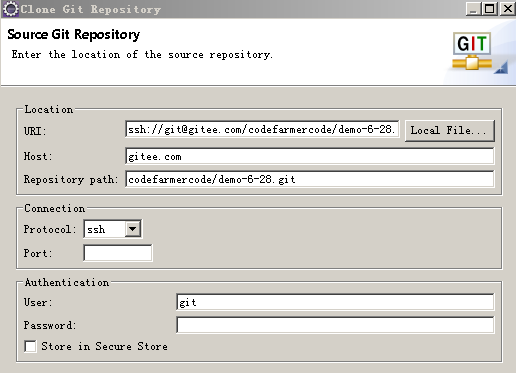
# 本地库和远程库的协同

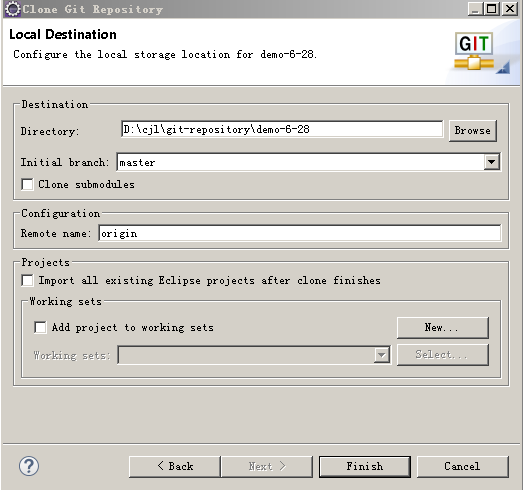
请先在服务器端创建一个工程



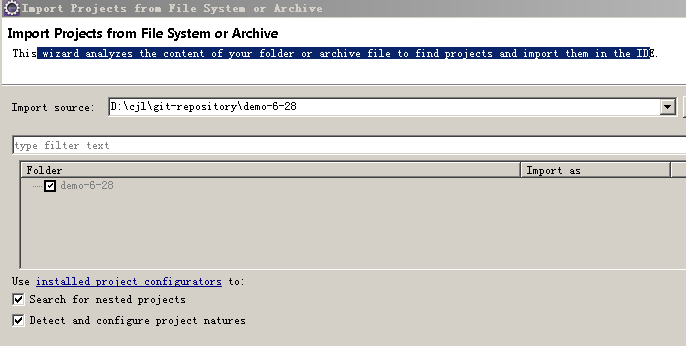
在本地克隆远程服务器库

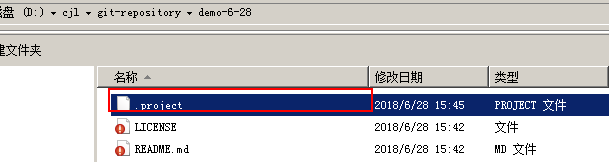


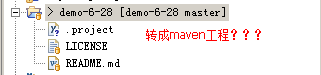




导入版本库到eclipse工程

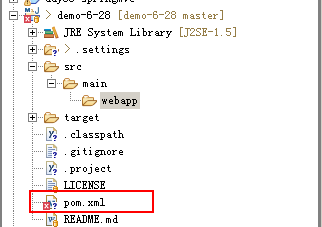




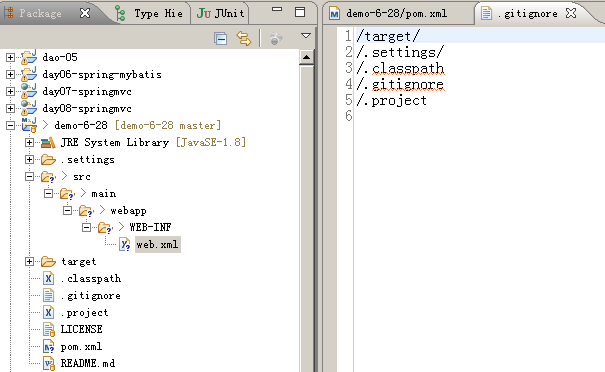


菜单“configuration”选择

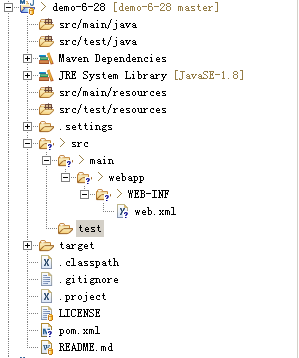
转换maven



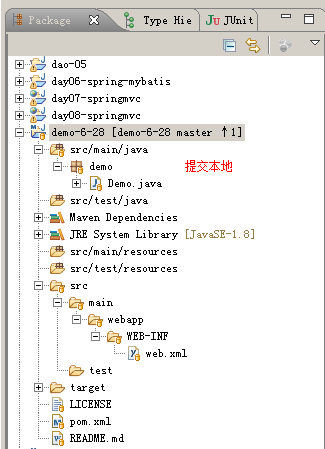
添加源代码



手动创建maven项目目录结构

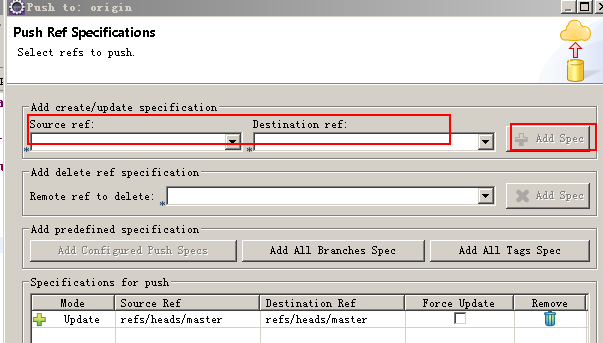


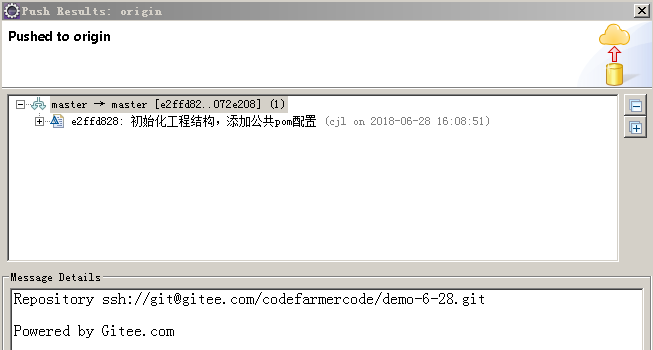
提交本地版本库



同步到远程服务器Push

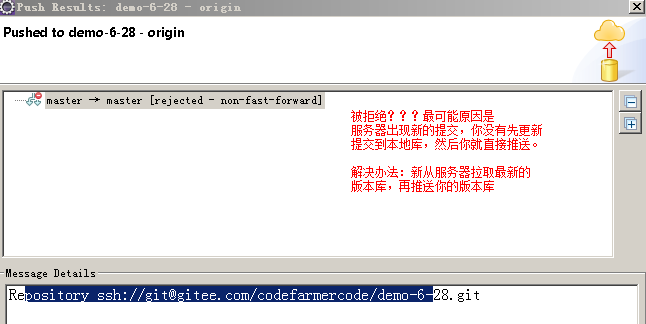


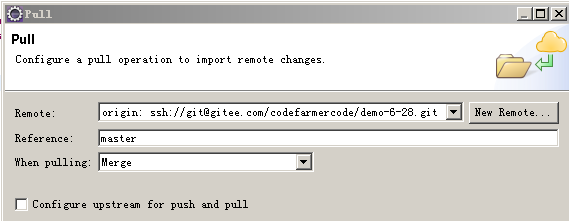


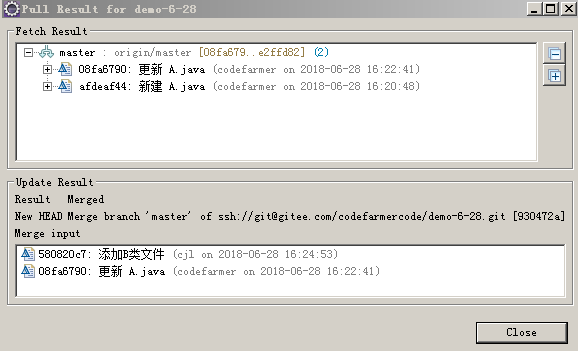


查看服务器端

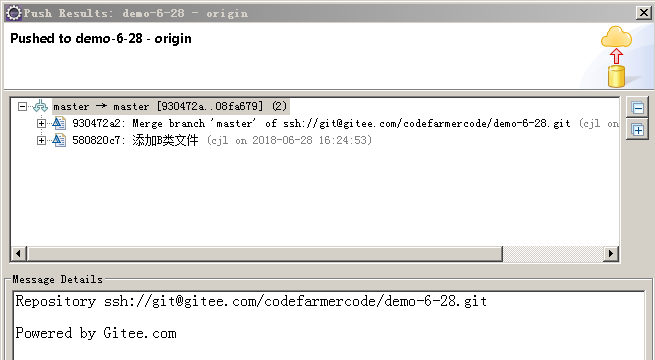


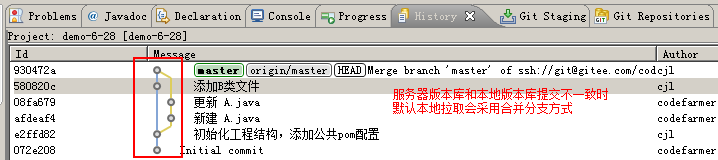






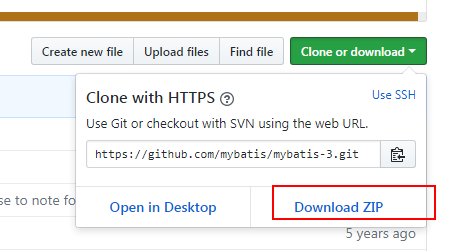
再次push



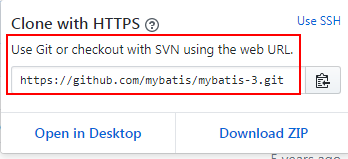


# 获取开源项目源码

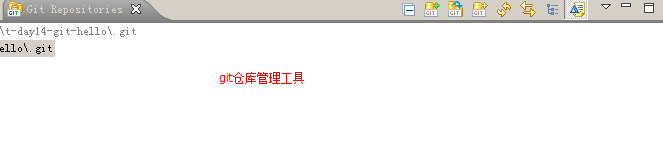
## 下载\*.zip文件

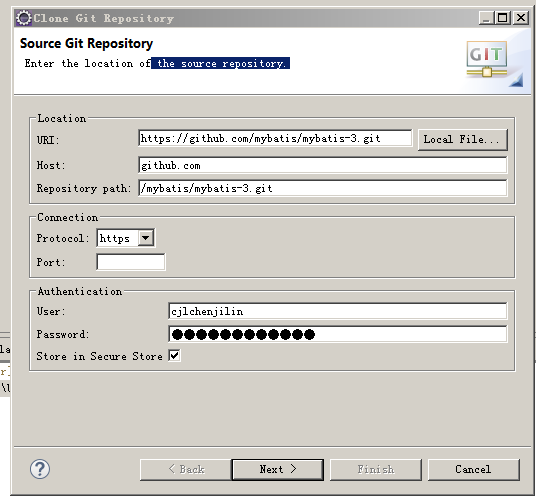


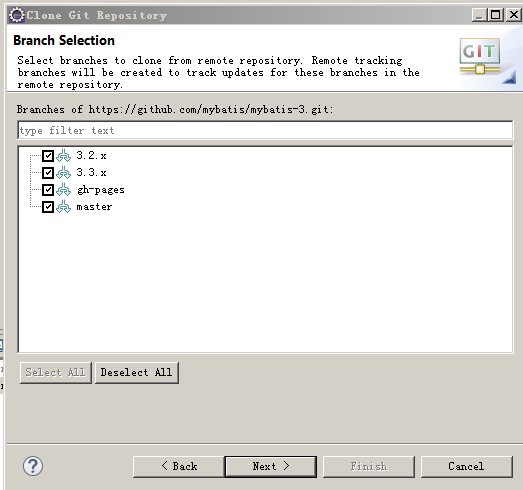
## 克隆的方式

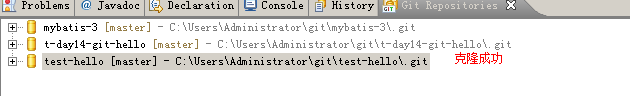


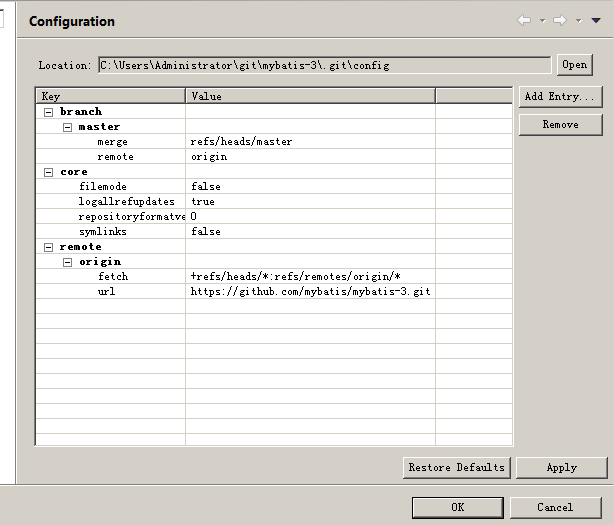
Egit的仓库管理界面：

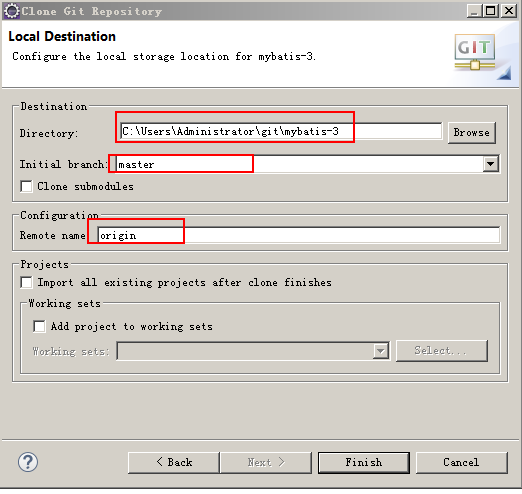












**《补充内容》**

# Git概念

## 仓库

The Repository or Object Database stores all objects which make up the history of the project. All objects in this database are identified through a secure 20 byte [**SHA-1 hash**](http://en.wikipedia.org/wiki/SHA-1) of the object content.

Git has four object types :

* A **Blob object** stores file content.
* A **Tree object** stores the directory structure and contains **Blob objects** and other **Tree objects** together with their file system names and modes.
* A **Commit object** represents a snapshot of the directory structure at the time of the commit and has a link to its predecessor **Commit object** which form an acyclic graph of the repository revisions forming the repository history.
* A **Tag object** is a symbolic named link to another repository object which contains the object's name and type. Optionally, it also contains information about who created the tag and other signing information.

The object database is stored in the .git/objects directory. Objects are either stored as loose objects or in a single-file packed format for efficient storage and transport.

## Index

The **Git Index** is a binary file stored in the .git/index directory containing a sorted list of file names, file modes, and file meta data used to efficiently detect file modifications. It also contains the SHA-1 object names of blob objects.

It has the following important properties:

* The index contains all information necessary to generate a single uniquely defined tree object. E.g. a commit operation generates this tree, stores it in the object database and associates it with the commit.
* The index enables fast comparison of the tree it defines with the current working directory. This is achieved by storing additional meta data about the involved files in the index data.
* The index can efficiently store information about merge conflicts between the trees involved in the merge so that for each pathname there is enough information about the involved trees to enable a three-way merge.

## Branch

A branch in Git is a named reference to a commit. There are two types of branches, namely "Local" and "Remote Tracking" branches which serve different purposes.

### Local Branches

Whenever a change to a (local) Repository is committed, a new commit object is created. Without any other means, it would be very difficult to keep track of the changes in the Repository, in particular when other commits are added to the Repository, for example due to an update from the remote Repository or when checking out another commit.

A local branch helps with this task by providing a (local) name by which the "current" commit can be found. When changes are committed to the local repository, the branch is automatically updated to point to the newly created commit.

In addition, it is possible to add a so-called upstream configuration to a local branch which can be helpful when synchronizing with a remote repository.

### Remote Tracking Branches

Remote tracking branches are created automatically when cloning and fetching from remote repositories. A remote tracking branch in the local repository always corresponds to a (local) branch in the remote repository. The name of such a branch follows certain conventions.

The remote tracking branch points to the same commit as the corresponding branch in the remote repository (at the time of the clone/fetch).

Remote tracking branches can be used for automated creation of upstream configuration for local branches.

## Working Directory

The working directory is the directory used to modify files for the next commit. By default it is located one level above the .git directory. Making a new commit typically involves the following steps:

* Check out the branch the new commit shall be based on. This changes the working directory so that it reflects the *HEAD* revision of the branch.
* Do modifications in the working directory.
* Tell git about these modifications (add modified files). This transfers the modified file contents into the object database and prepares the tree to be committed in the index.
* Commit the tree prepared in the index into the object database.
* The result is a new commit object and the *HEAD* of the current branch moves to the new commit.

## Recording Changes in the Repository

You start from a fresh checkout of a branch of a local repository. You want to do some changes and record snapshots of these changes in the repository whenever you reach a state you want to record.

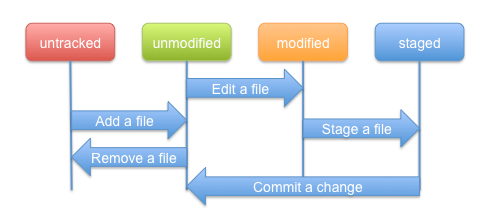
Each file in the working directory can either be *tracked* or *untracked*:

* **Tracked** files are those which were in the last snapshot or files which have been newly staged into the *index*. They can be *unmodified*, *modified*, or *staged*.
* **Untracked** files are all other files (they were not in the last snapshot and have not yet been added to the *index*).

When you first clone a repository, all files in the working directory will be *tracked* and *unmodified* since they have been freshly checked out and you haven't started editing them yet.

As you edit files, git will recognize they are *modified* with respect to the last commit. You *stage* the modified files into the index and then *commit* the staged changes. The cycle can then repeat.

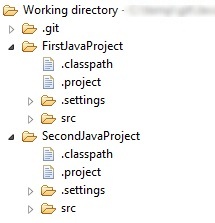
**This lifecycle** is illustrated here:



## 练习

* Don't create the Repository within the Eclipse workspace.
  + Be careful when cloning or creating a Repository.
  + Make sure to use the Git Sharing Wizard correctly.
* Don't create a Repository with an Eclipse project as root.
  + Make sure to use the Git Sharing Wizard correctly.

Both mistakes will happen when you use the Git Sharing Wizard from an Eclipse project that you have created manually in your workspace without taking precautions (the wizard has been fixed in the latest version).



### Creating a new empty Git Repository

You can create a project first and share it afterwards. The Share Project Wizard supports creation of Git repositories (see [**Adding a project to version control**](http://wiki.eclipse.org/EGit/User_Guide/Sharing#Adding_a_project_to_version_control)).

You can also create a new empty Git Repository from the Git Repositories View (see [**Creating a Repository**](http://wiki.eclipse.org/EGit/User_Guide#Creating_a_Repository)).

### Creating a Git Repository for multiple Projects

You may create multiple projects under a common directory and then create a common repository for all projects in one go:

* create the Eclipse projects (e.g. a, b, c) under a common directory (e.g. /repos/examples/)
* select all projects (a, b, c), and in the contextual menu click **Team > Share Project > Git**
* press **Next**
* select all projects (a, b, c)
* the wizard automatically moves up the default repository location to the parent folder /repos/examples/ since multiple projects have been selected
* click **Create Repository** and **Finish**

参考文档：<http://wiki.eclipse.org/EGit/User_Guide>