



黑马程序员™  
www.itheima.com

传智播客旗下  
高端IT教育品牌

# Git

# 目录 Contents

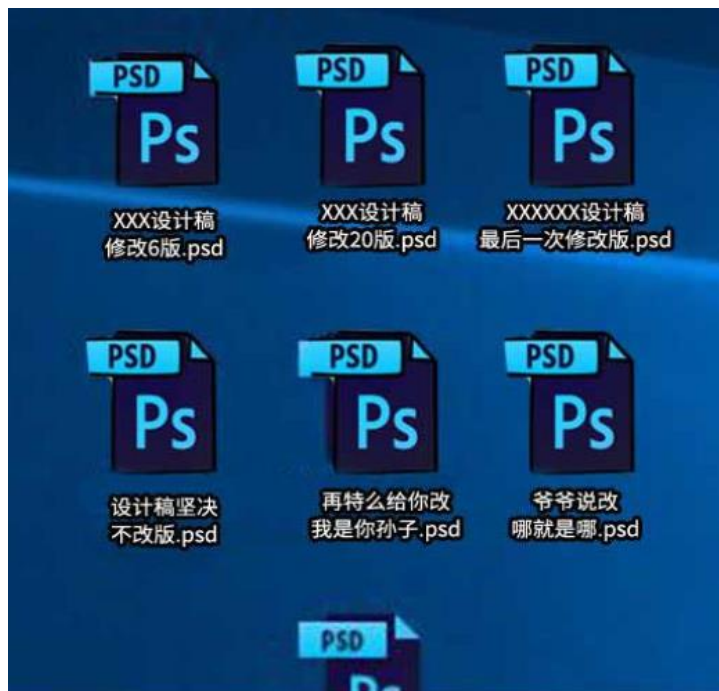
## ◆ 起步

### ◆ Git 基础

### ◆ Github

### ◆ Git 分支

## 1. 文件的版本



### 操作麻烦

每次都需要复制 → 粘贴 → 重命名

### 命名不规范

无法通过文件名知道具体做了哪些修改

### 容易丢失

如果硬盘故障或不小心删除，文件很容易丢失

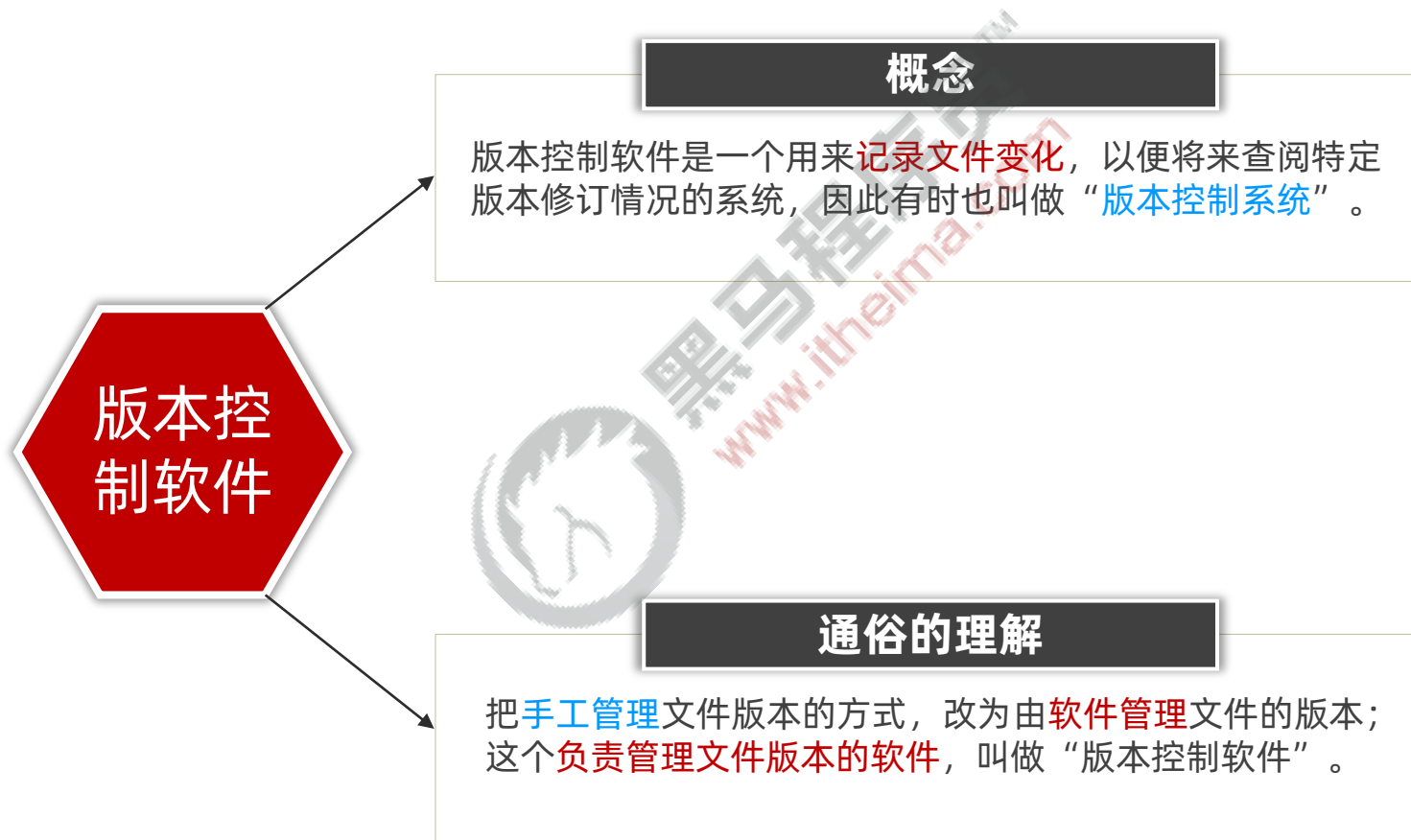
### 协作困难

需要手动合并每个人对项目文件的修改，合并时极易出错

① 人和动物的区别？

🔧 人会制造并使用工具

## 2. 版本控制软件



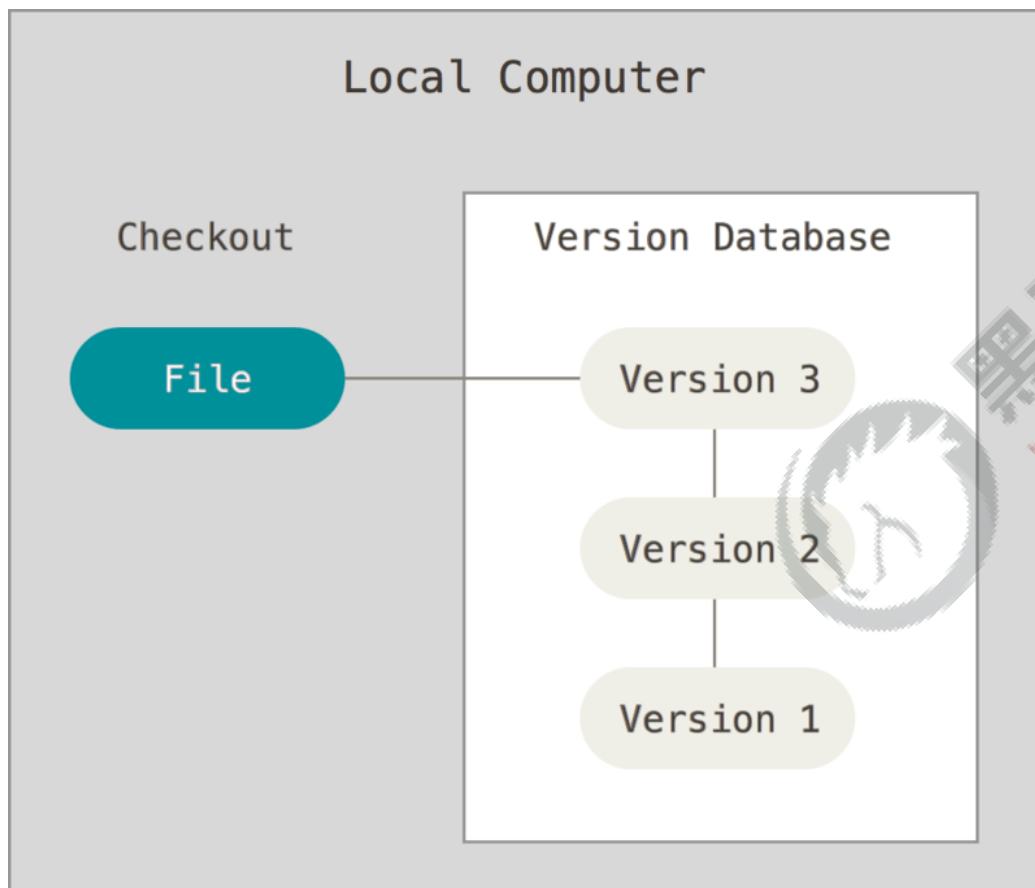
### 3. 使用版本控制软件的好处

操作简便	只需识记几组简单的终端命令，即可快速上手常见的版本控制软件
易于对比	基于版本控制软件提供的功能，能够方便地比较文件的变化细节，从而查找出导致问题的原因
易于回溯	可以将选定的文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态
不易丢失	在版本控制软件中，被用户误删除的文件，可以轻松的恢复回来
协作方便	基于版本控制软件提供的分支功能，可以轻松实现多人协作开发时的代码合并操作

### 4. 版本控制系统的分类



### 4.1 本地版本控制系统



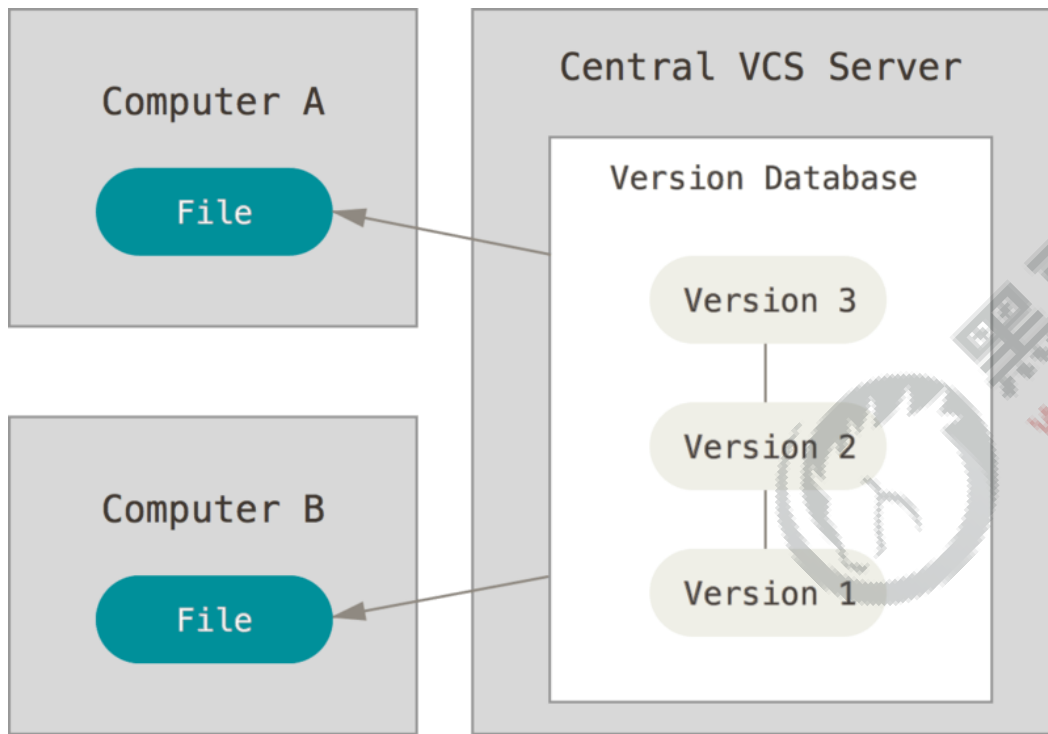
特点：


使用软件来记录文件的不同版本，提高了工作效率，降低了手动维护版本的出错率

缺点：

- ① 单机运行，不支持多人协作开发
- ② 版本数据库故障后，所有历史更新记录会丢失

### 4.2 集中化的版本控制系统



 典型代表：SVN

特点：基于服务器、客户端的运行模式

- ① 服务器保存文件的所有更新记录
- ② 客户端只保留最新的文件版本

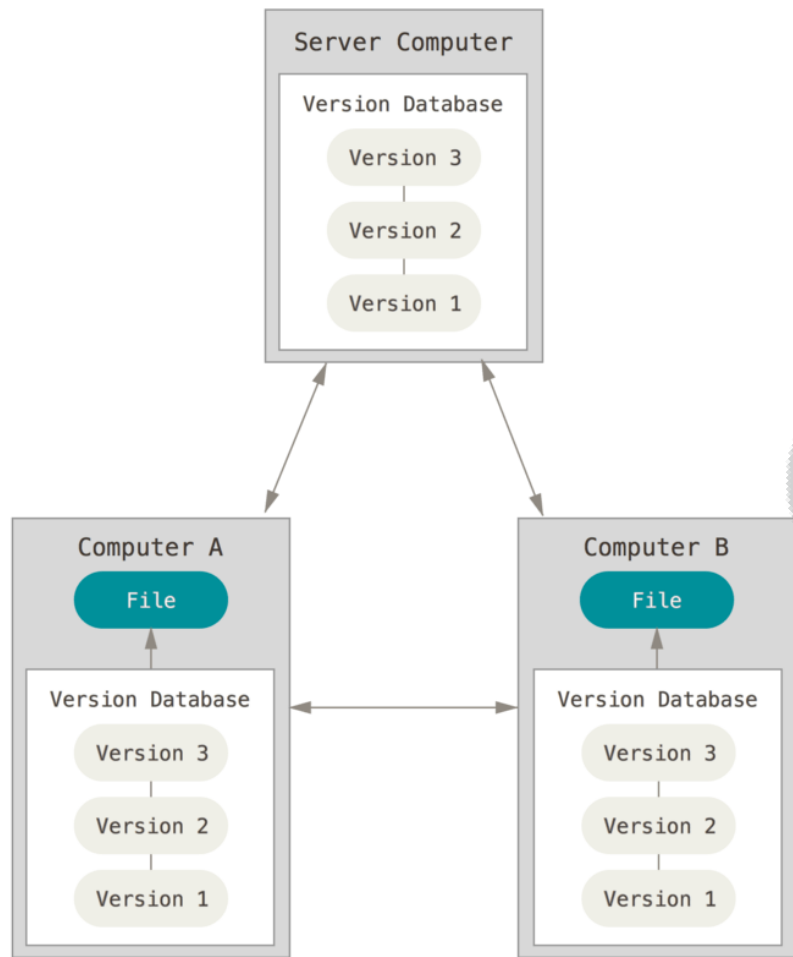
优点：联网运行，支持多人协作开发

缺点：

- ① 不支持离线提交版本更新
- ② 中心服务器崩溃后，所有人无法正常工作
- ③ 版本数据库故障后，所有历史更新记录会丢失



### 4.3 分布式版本控制系统



特点：基于服务器、客户端的运行模式

- 服务器保存文件的所有更新版本
- 客户端是服务器的完整备份，并不是只保留文件的最新版本

优点：

- ① 联网运行，支持多人协作开发
- ② 客户端断网后支持离线本地提交版本更新
- ③ 服务器故障或损坏后，可使用任何一个客户端的备份进行恢复

*i* 典型代表：Git

## 1. 什么是 Git

Git 是一个开源的分布式版本控制系统，是目前世界上最先进、最流行的版本控制系统。可以快速高效地处理从很小到非常大的项目版本管理。

特点：项目越大越复杂，协同开发者越多，越能体现出 Git 的高性能和高可用性！

### 2. Git 的特性

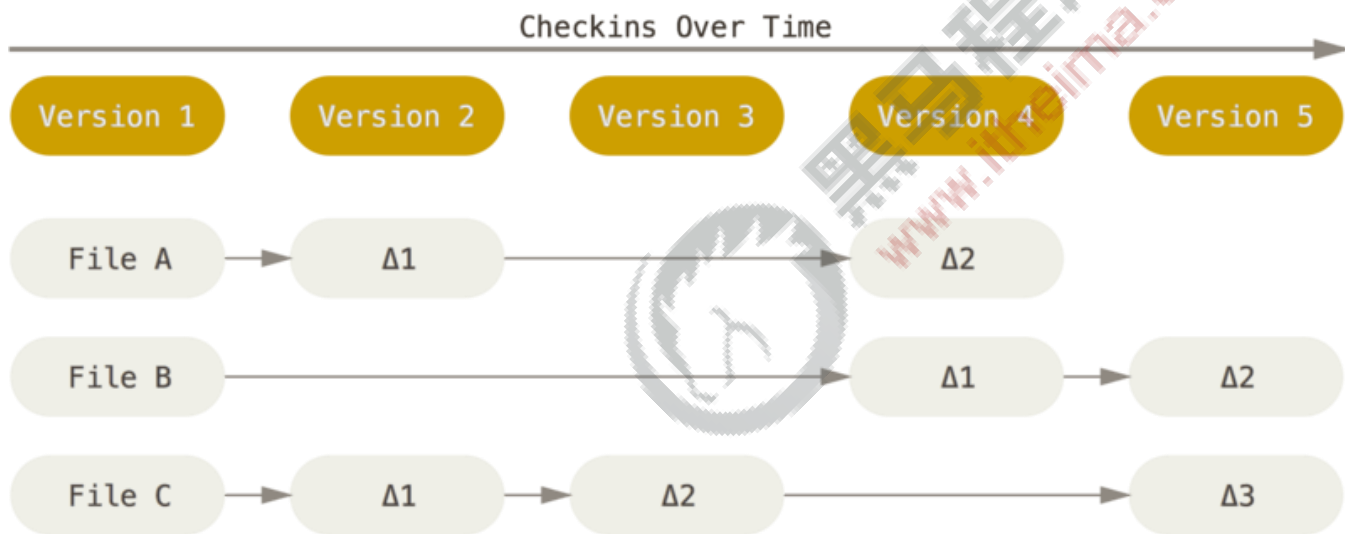
Git 之所以快速和高效，主要依赖于它的如下两个特性：

- ① 直接记录快照，而非差异比较
- ② 近乎所有操作都是本地执行



### 2.1 SVN 的差异比较

传统的版本控制系统（例如 SVN）是**基于差异**的版本控制，它们存储的是一组基本文件和**每个文件随时间逐步累积的差异**。



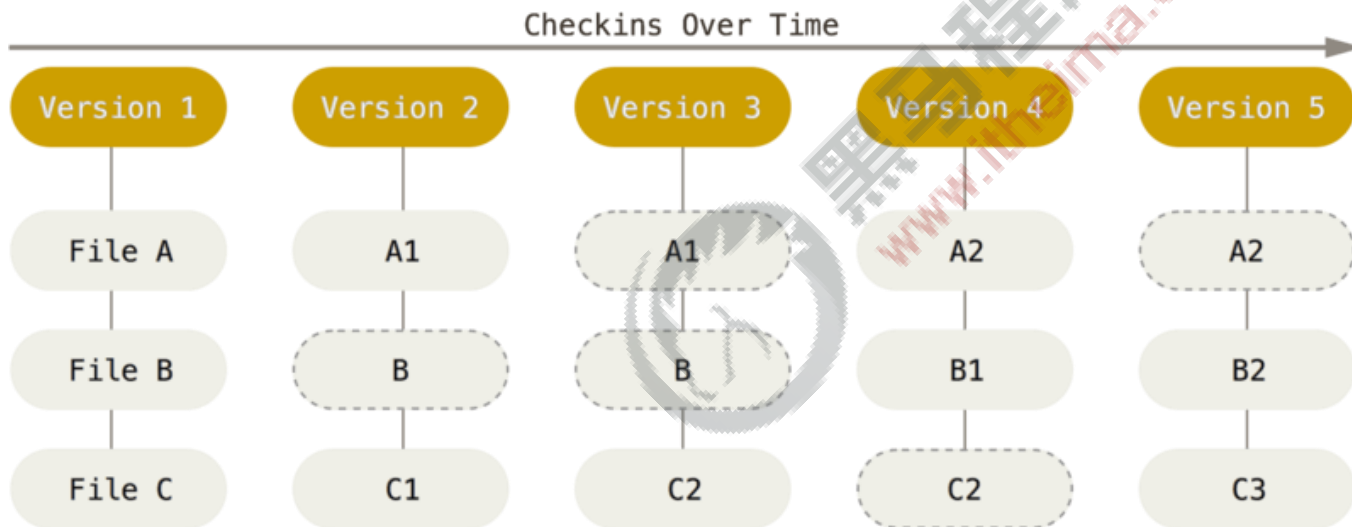
好处：节省磁盘空间

缺点：**耗时、效率低**

在每次切换版本的时候，都需要在基本文件的基础上，应用每个差异，从而生成目标版本对应的文件。

### 2.2 Git 的记录快照

**Git 快照**是在原有文件版本的基础上重新生成一份新的文件，**类似于备份**。为了效率，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。

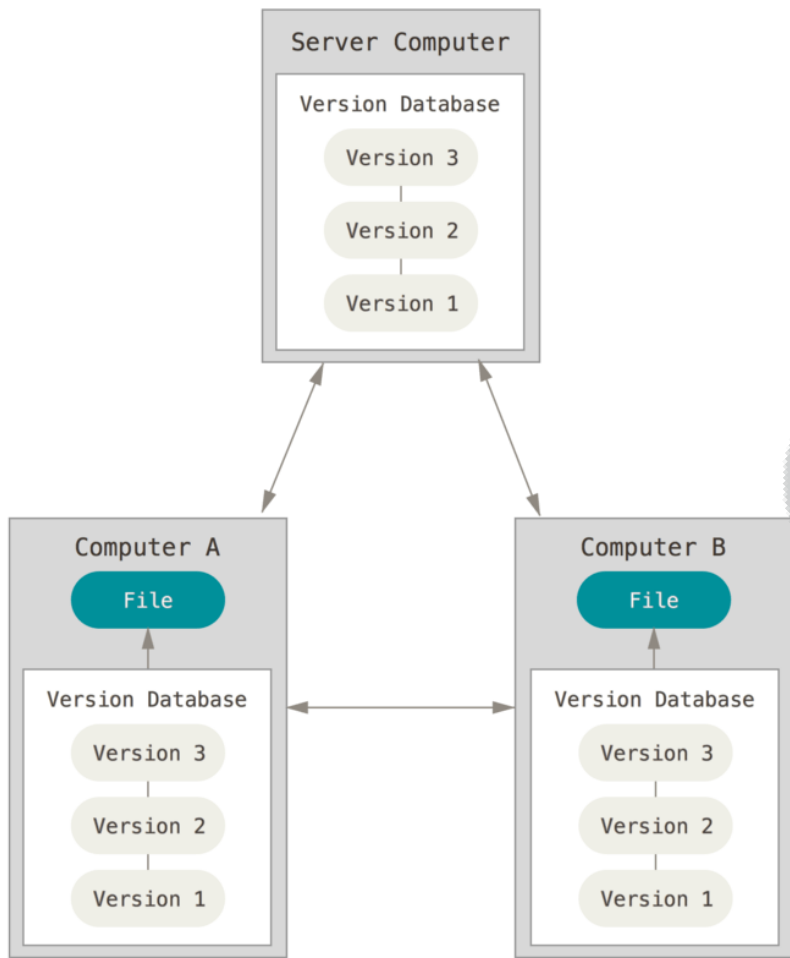


缺点：占用磁盘空间较大

优点：**版本切换时非常快**，因为每个版本都是完整的文件快照，切换版本时直接恢复目标版本的快照即可。

特点：**空间换时间**

### 2.3 近乎所有操作都是本地执行



在 Git 中的绝大多数操作都只需要访问本地文件和资源，一般不需要来自网络上其它计算机的信息。

特性：

- ① 断网后依旧可以在本地对项目进行版本管理
- ② 联网后，把本地修改的记录同步到云端服务器即可

### 3. Git 中的三个区域

使用 Git 管理的项目，拥有三个区域，分别是**工作区**、**暂存区**、**Git 仓库**。



**工作区**

处理工作的区域



**暂存区**

已完成的工作的**临时存放区域**，  
等待被提交



**Git 仓库**

最终的存放区域

### 4. Git 中的三种状态



#### 已修改

表示修改了文件，但还没将修改的结果放到暂存区

#### 已暂存

表示对已修改文件的当前版本做了标记，使之包含在下次提交的列表中

#### 已提交

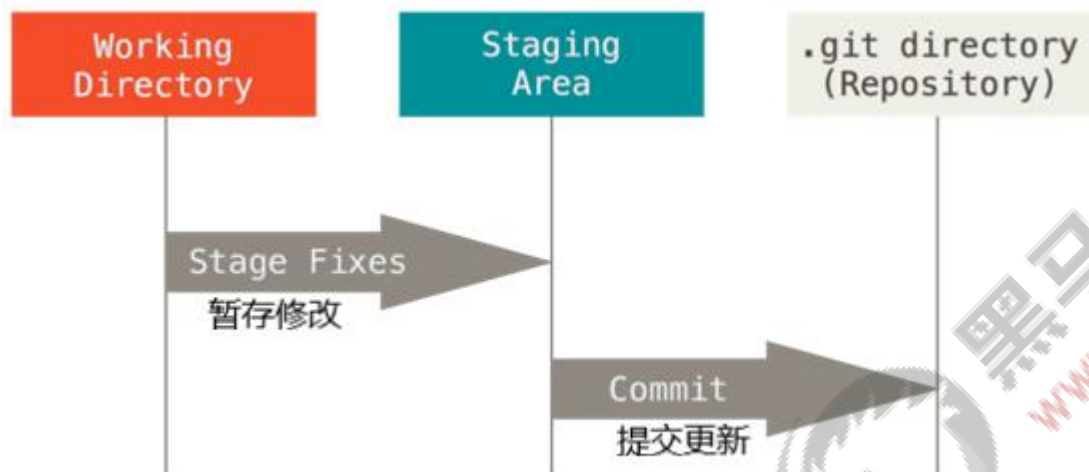
表示文件已经安全地保存在本地的 Git 仓库中

注意：

- 工作区的文件被修改了，但还没有放到暂存区，就是**已修改**状态。
- 如果文件已修改并放入暂存区，就属于**已暂存**状态。
- 如果 Git 仓库中**保存着特定版本**的文件，就属于**已提交**状态。



## 5. 基本的 Git 工作流程



基本的 Git 工作流程如下：

- ① 在工作区中修改文件
- ② 将你想要下次提交的更改进行暂存
- ③ 提交更新，找到暂存区的文件，将快照永久性存储到 Git 仓库

# 目录 Contents

◆ 起步

◆ Git 基础

◆ Github

◆ Git 分支

## 1. 在 Windows 中下载并安装 Git

在开始使用 Git 管理项目的版本之前，需要将它安装到计算机上。可以使用浏览器访问如下的网址，根据自己的操作系统，选择下载对应的 Git 安装包：

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



## 2. 配置用户信息

安装完 Git 之后，要做的第一件事就是设置自己的用户名和邮件地址。因为通过 Git 对项目进行版本管理的时候，Git 需要使用这些基本信息，来记录是谁对项目进行了操作：

```
1 git config --global user.name "itheima"
2 git config --global user.email "itheima@itcast.cn"
```

❗ 注意：如果使用了 `--global` 选项，那么该命令只需要运行一次，即可永久生效。

## 2. Git 的全局配置文件

通过 `git config --global user.name` 和 `git config --global user.email` 配置的用户名和邮箱地址，会被写入到 `C:/Users/用户名文件夹/.gitconfig` 文件中。这个文件是 Git 的**全局配置文件**，**配置一次即可永久生效**。

可以使用记事本打开此文件，从而查看自己曾经对 Git 做了哪些全局性的配置。



```
*.gitconfig - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
[user]
    name = itheima
    email = itheima@itcast.cn
第 3 行, 第 27 列 100% Unix (LF) UTF-8
```

## 4. 检查配置信息

除了使用记事本查看全局的配置信息之外，还可以运行如下的终端命令，快速的查看 Git 的全局配置信息：

```
1 # 查看所有的全局配置项
2 git config --list --global
3 # 查看指定的全局配置项
4 git config user.name
5 git config user.email
```

## 5. 获取帮助信息

可以使用 `git help <verb>` 命令，无需联网即可在浏览器中打开帮助手册，例如：

```
1 # 要想打开 git config 命令的帮助手册
2 git help config
```

如果不想查看完整的手册，那么可以用 `-h` 选项获得更简明的“help”输出：

```
1 # 想要获取 git config 命令的快速参考
2 git config -h
```

## 1. 获取 Git 仓库的两种方式

- ① 将尚未进行版本控制的本地目录**转换**为 Git 仓库
- ② 从其它服务器**克隆**一个已存在的 Git 仓库

以上两种方式都能够在自己的电脑上得到一个可用的 Git 仓库





## 2. 在现有目录中初始化仓库

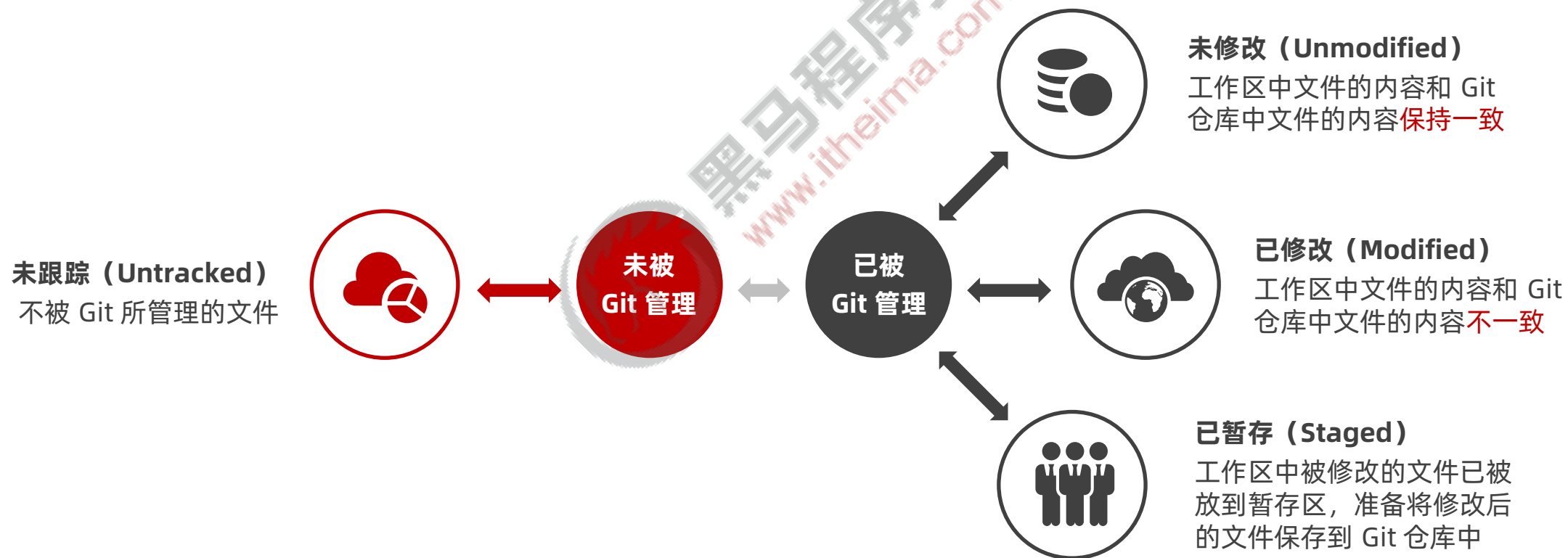
如果自己有一个尚未进行版本控制的项目目录，想要用 Git 来控制它，需要执行如下两个步骤：

- ① 在项目目录中，通过鼠标右键打开 “Git Bash”
- ② 执行 `git init` 命令将当前的目录转化为 Git 仓库

`git init` 命令会创建一个名为 `.git` 的隐藏目录，**这个 `.git` 目录就是当前项目的 Git 仓库**，里面包含了**初始的必要文件**，这些文件是 Git 仓库的**必要组成部分**。

## 3. 工作区中文件的 4 种状态

工作区中的每一个文件可能有 4 种状态，这四种状态共分为两大类，如图所示：



🚀 Git 操作的终极结果：让工作区中的文件都处于“未修改”的状态。



## 4. 检查文件的状态

可以使用 `git status` 命令查看文件处于什么状态，例如：

```
C:\Windows\System32\cmd.exe

E:\code>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    index.html

nothing added to commit but untracked files present (use "git add" to track)
```

在状态报告中可以看到新建的 `index.html` 文件出现在 `Untracked files`（未跟踪的文件）下面。

未跟踪的文件意味着 `Git` 在之前的快照（提交）中没有这些文件；`Git` 不会自动将之纳入跟踪范围，除非明确地告诉它“我需要使用 `Git` 跟踪管理该文件”。



## 5. 以精简的方式显示文件状态

使用 `git status` 输出的状态报告很详细，但有些繁琐。如果希望以精简的方式显示文件的状态，可以使用如下两条完全等价的命令，其中 `-s` 是 `--short` 的简写形式：

```
1 # 以精简的方式显示文件状态
2 git status -s
3 git status --short
```

未跟踪文件前面有红色的 `??` 标记，例如：

```
C:\> 选择C:\Windows\System32\cmd.exe

E:\code>git status -s
?? index.html
```

## 6. 跟踪新文件

使用命令 `git add` 开始跟踪一个文件。所以，要跟踪 `index.html` 文件，运行如下的命令即可：

```
1 git add index.html
```

此时再运行 `git status` 命令，会看到 `index.html` 文件在 `Changes to be committed` 这行的下面，说明 **已被跟踪**，并处于暂存状态：

```
C:\Windows\System32\cmd.exe

E:\code>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   index.html
```

```
C:\Windows\System32\cmd.exe

E:\code>git status -s
A  index.html
```

以精简的方式显示文件的状态：

新添加到暂存区中的文件前面有绿色的 **A** 标记



## 7. 提交更新

现在暂存区中有一个 index.html 文件等待被提交到 Git 仓库中进行保存。可以执行 `git commit` 命令进行提交, 其中 `-m` 选项后面是本次的提交消息, 用来对提交的内容做进一步的描述:

```
1 git commit -m "新建了index.html文件"
```

提交成功之后, 会显示如下的信息:

```
C:\Windows\System32\cmd.exe

E:\code>git commit -m "新建了index.html文件"
[master (root-commit) 270b1f3] 新建了index.html文件
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 index.html
```

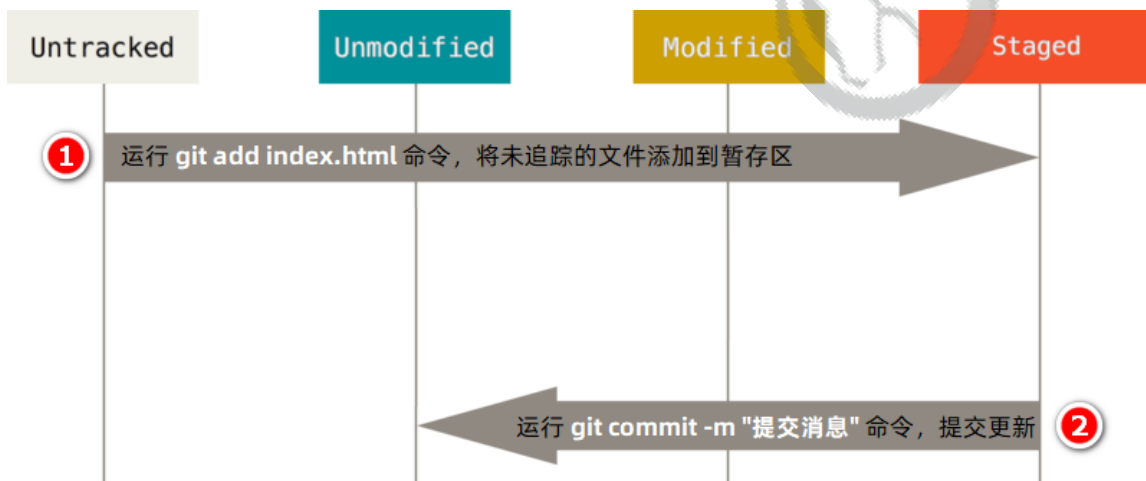
## 7. 提交更新

提交成功之后，再次检查文件的状态，得到提示如下：

```
C:\Windows\System32\cmd.exe

E:\code>git status
On branch master
nothing to commit, working tree clean
```

证明工作区中所有的文件都处于“未修改”的状态，没有任何文件需要被提交。





## 8. 对已提交的文件进行修改

目前，index.html 文件已经被 Git 跟踪，并且工作区和 Git 仓库中的 index.html 文件内容保持一致。当我们修改了工作区中 index.html 的内容之后，再次运行 `git status` 和 `git status -s` 命令，会看到如下的内容：

```
C:\Windows\System32\cmd.exe

E:\code>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:   index.html

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

E:\code>git status -s
 M index.html
```

文件 index.html 出现在 `Changes not staged for commit` 这行下面，说明已跟踪文件的内容发生了变化，但还没有放到暂存区。

❗ 注意：修改过的、没有放入暂存区的文件前面有红色的 M 标记。





## 9. 暂存已修改的文件

目前，工作区中的 index.html 文件已被修改，如果要暂存这次修改，需要再次运行 `git add` 命令，这个命令是个多功能的命令，主要有如下 3 个功效：

- ① 可以用它开始跟踪新文件
- ② 把已跟踪的、且已修改的文件放到暂存区
- ③ 把有冲突的文件标记为已解决状态

```
C:\Windows\System32\cmd.exe

E:\code>git add index.html 把已修改的文件放入暂存区

E:\code>git status 查看详细的文件状态报告
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   index.html

E:\code>git status -s 查看精简的文件状态报告，
M index.html 绿色的 M 表示文件已修改且已放入暂存区
```

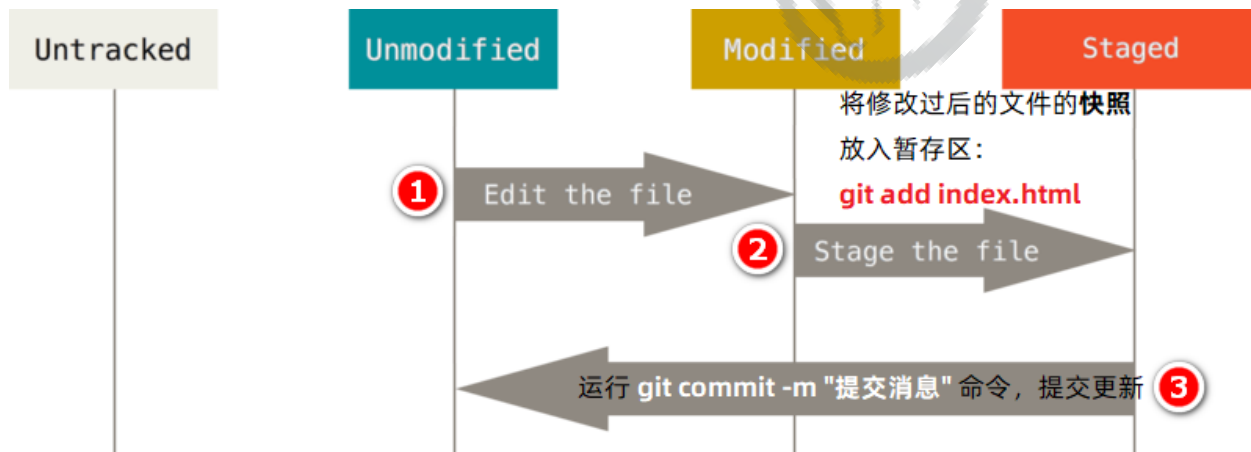
## 10. 提交已暂存的文件

再次运行 `git commit -m "提交消息"` 命令，即可将暂存区中记录的 index.html 的快照，提交到 Git 仓库中进行保存：

```
C:\Windows\System32\cmd.exe

E:\code>git commit -m "初始化了index.html中的内容" 将暂存区中的文件提交到 Git 仓库
[master 554647a] 初始化了index.html中的内容
1 file changed, 14 insertions(+)

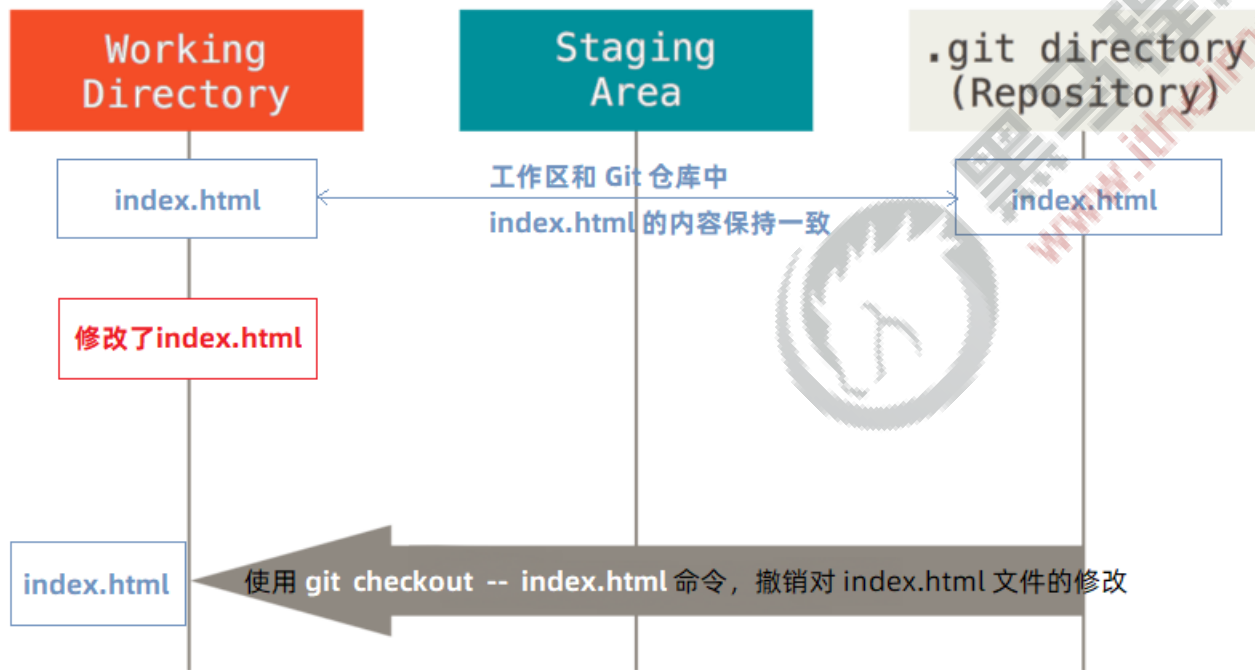
E:\code>git status 检查工作区中文件的状态
On branch master
nothing to commit, working tree clean
```



## 11. 撤销对文件的修改

撤销对文件的修改指的是：把对工作区中对应文件的修改，**还原**成 Git 仓库中所保存的版本。

操作的结果：所有的修改会丢失，且无法恢复！**危险性比较高，请慎重操作！**



**i** 撤销操作的本质：用 Git 仓库中保存的文件，覆盖工作区中指定的文件。

## 12. 向暂存区中一次性添加多个文件

如果需要被暂存的文件个数比较多，可以使用如下的命令，一次性将所有的新增和修改过的文件加入暂存区：

```
1 git add .
```

① 今后在项目开发中，会经常使用这个命令，将新增和修改过后的文件加入暂存区。

## 13. 取消暂存的文件

如果需要从暂存区中移除对应的文件，可以使用如下的命令：

```
1 git reset HEAD 要移除的文件名称
```



## 14. 跳过使用暂存区域

Git 标准的工作流程是工作区 → 暂存区 → Git 仓库，但有时候这么做略显繁琐，此时可以跳过暂存区，直接将工作区中的修改提交到 Git 仓库，这时候 Git 工作的流程简化为了工作区 → Git 仓库。

Git 提供了一个跳过使用暂存区域的方式，只要在提交的时候，给 `git commit` 加上 `-a` 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 `git add` 步骤：

```
1 git commit -a -m "描述消息"
```



## 15. 移除文件

从 Git 仓库中移除文件的方式有两种：

- ① 从 Git 仓库和工作区中同时移除对应的文件
- ② 只从 Git 仓库中移除指定的文件，但保留工作区中对应的文件

```
1 # 从 Git 仓库和工作区中同时移除 index.js 文件
2 git rm -f index.js
3 # 只从 Git 仓库中移除 index.css，但保留工作区中的 index.css 文件
4 git rm --cached index.css
```



## 16. 忽略文件

一般我们总会有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。在这种情况下，我们可以创建一个名为 `.gitignore` 的配置文件，列出要忽略的文件的匹配模式。

文件 `.gitignore` 的格式规范如下：

- ① 以 `#` 开头的是注释
- ② 以 `/` 结尾的是目录
- ③ 以 `/` 开头防止递归
- ④ 以 `!` 开头表示取反
- ⑤ 可以使用 `glob 模式` 进行文件和文件夹的匹配（glob 指简化了的正则表达式）



## 17. glob 模式

所谓的 **glob 模式**是指简化了的正则表达式：

- ① **星号 \*** 匹配零个或多个任意字符
- ② **[abc]** 匹配任何一个列在方括号中的字符（此案例匹配一个 a 或匹配一个 b 或匹配一个 c）
- ③ **问号 ?** 只匹配一个任意字符
- ④ 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）
- ⑤ **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）



## 18. .gitignore 文件的例子

```
1 # 忽略所有的 .a 文件
2 *.a
3
4 # 但跟踪所有的 lib.a, 即便你在前面忽略了 .a 文件
5 !lib.a
6
7 # 只忽略当前目录下的 TODO 文件, 而不忽略 subdir/TODO
8 /TODO
9
10 # 忽略任何目录下名为 build 的文件夹
11 build/
12
13 # 忽略 doc/notes.txt, 但不忽略 doc/server/arch.txt
14 doc/*.txt
15
16 # 忽略 doc/ 目录及其所有子目录下的 .pdf 文件
17 doc/**/*.*pdf
```

## 19. 查看提交历史

如果希望回顾项目的提交历史，可以使用 `git log` 这个简单且有效的命令。

```
1 # 按时间先后顺序列出所有的提交历史，最近的提交排在最上面
2 git log
3
4 # 只展示最新的两条提交历史，数字可以按需进行填写
5 git log -2
6
7 # 在一行上展示最近两条提交历史的信息
8 git log -2 --pretty=oneline
9
10 # 在一行上展示最近两条提交历史的信息，并自定义输出的格式
11 # %h 提交的简写哈希值    %an作者名字    %ar作者修订日期，按多久以前的方式显示    %s提交说明
12 git log -2 --pretty=format:@"%h | %an | %ar | %s"
```

## 20. 回退到指定的版本

```
1 # 在一行上展示所有的提交历史
2 git log --pretty=oneline
3
4 # 使用 git reset --hard 命令, 根据指定的提交 ID 回退到指定版本
5 git reset --hard <CommitID>
6
7 # 在旧版本中使用 git reflog --pretty=oneline 命令, 查看命令操作的历史
8 git reflog --pretty=oneline
9
10 # 再次根据最新的提交 ID, 跳转到最新的版本
11 git reset --hard <CommitID>
```

## 21. 小结

### ① 初始化 Git 仓库的命令

- `git init`

### ② 查看文件状态的命令

- `git status` 或 `git status -s`

### ③ 一次性将文件加入暂存区的命令

- `git add .`

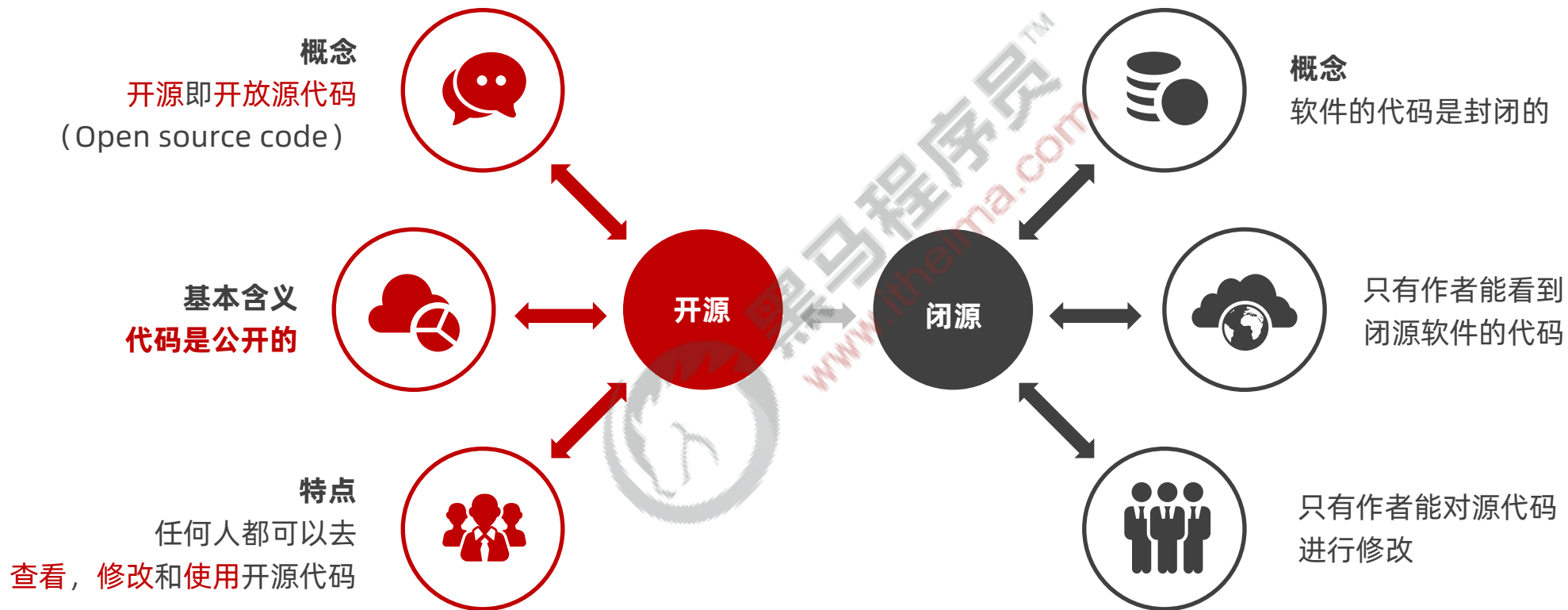
### ④ 将暂存区的文件提交到 Git 仓库的命令

- `git commit -m "提交消息"`

# 目录 Contents

- ◆ 起步
- ◆ Git 基础
- ◆ Github
- ◆ Git 分支

## 1. 什么是开源



### 通俗的理解

开源是指不仅提供程序还提供程序的源代码

闭源是只提供程序, 不提供源代码

## 2. 什么是**开源许可协议**

开源并不意味着完全没有限制，为了**限制使用者的使用范围**和**保护作者的权利**，每个开源项目都应该遵守**开源许可协议**（ Open Source License ）。



黑马程序员™  
www.itheima.com



## 3. 常见的 5 种开源许可协议

- ① BSD (Berkeley Software Distribution)
- ② Apache Licence 2.0
- ③ **GPL** (GNU General Public License)
  - 具有传染性的一种开源协议，不允许修改后和衍生的代码做为闭源的商业软件发布和销售
  - 使用 GPL 的最著名的软件项目是：Linux
- ④ LGPL (GNU Lesser General Public License)
- ⑤ **MIT** (Massachusetts Institute of Technology, MIT)
  - 是目前限制最少的协议，唯一的条件：在修改后的代码或者发行包中，必须包含原作者的许可信息
  - 使用 MIT 的软件项目有：jquery、Node.js

关于更多开源许可协议的介绍，可以参考博客 <https://www.runoob.com/w3cnote/open-source-license.html>

## 4. 为什么要**拥抱开源**

开源的**核心思想**是“**我为人人，人人为我**”，人们越来越喜欢开源大致是出于以下 3 个原因：

- ① 开源给使用者更多的控制权
- ② 开源让学习变得容易
- ③ 开源才有真正的安全

开源是软件开发领域的大趋势，**拥抱开源就像站在了巨人的肩膀上**，不用自己重复造轮子，让开发越来越容易。

## 5. 开源项目托管平台

专门用于免费存放开源项目源代码的网站，叫做**开源项目托管平台**。目前世界上比较出名的开源项目托管平台主要有以下 3 个：

- Github（全球最牛的开源项目托管平台，没有之一）
- Gitlab（对代码私有性支持较好，因此企业用户较多）
- Gitee（又叫做**码云**，是国产的开源项目托管平台。访问速度快、纯中文界面、使用友好）

注意：以上 3 个开源项目托管平台，只能托管以 Git 管理的项目源代码，因此，它们的名字都以 Git 开头。

## 6. 什么是 Github

Github 是全球最大的**开源项目托管平台**。因为只支持 Git 作为唯一的版本控制工具，故名 GitHub。

在 Github 中，你可以：

- ① 关注自己喜欢的开源项目，为其点赞打 call
- ② 为自己喜欢的开源项目做贡献（Pull Request）
- ③ 和开源项目的作者讨论 Bug 和提需求（Issues）
- ④ 把喜欢的项目复制一份作为自己的项目进行修改（Fork）
- ⑤ 创建属于自己的开源项目
- ⑥ etc...

So, **Github  $\neq$  Git**

## 1. 注册 Github 账号的流程

- ① 访问 Github 的官网首页 <https://github.com/>
- ② 点击 “Sign up” 按钮跳转到注册页面
- ③ 填写可用的用户名、邮箱、密码
- ④ 通过点击箭头的形式，将验证图片摆正
- ⑤ 点击 “Create account” 按钮注册新用户
- ⑥ 登录到第三步填写的邮箱中，点击激活链接，完成注册

Username \*  
teacher-liu ✓

Email address \*  
liu@itcast.cn ✓

Password \*  
.....  
Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter.  
[Learn more.](#)

Email preferences  
☐ Send me occasional product updates, announcements, and offers.

Verify your account

当图像正向显示，  
请点击“完成”。

完成

?

🔊

Create account

## 2. 激活 Github 账号



Please verify your email address

Before you can contribute on GitHub, we need you to verify your email address.

An email containing verification instructions was sent to liu@itcast.cn.

[Resend verification email](#)

[Change your email settings](#)



Almost done, @teacher-liu! To complete your GitHub sign up, we just need to verify your email address: [liu@itcast.cn](#).

[Verify email address](#)

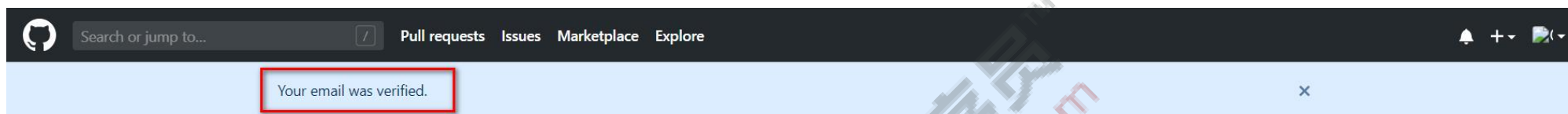
Once verified, you can start using all of GitHub's features to explore, build, and share projects.

Button not working? Paste the following link into your browser: [https://github.com/users/teacher-liu/emails/103277464/confirm\\_verification\\_token/d2caf3b0](https://github.com/users/teacher-liu/emails/103277464/confirm_verification_token/d2caf3b0)

You're receiving this email because you recently created a new GitHub account or added a new email address. If this wasn't you, please ignore this email.


[Email preferences](#) · [Terms](#) · [Privacy](#) · [Sign into GitHub](#)

## 3. 完成注册




### What do you want to do first?

Every developer needs to configure their environment, so let's get your GitHub experience optimized for you.




**Start a new project**  
Start a new repository or bring over an existing repository to keep contributing to it.

Create a repository



**Collaborate with your team**  
Improve the way your team works together and get access to more features with an organization.

Create an organization



**Learn how to use GitHub**  
Get started with an "Introduction to GitHub" course in our Learning Lab.

Start Learning

[Skip this for now >](#)

## 1. 新建空白远程仓库

The screenshot shows the GitHub interface for creating a new repository. The page title is "Create a new repository". Below the title, there is a description: "A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)".

The form fields are as follows:

- Owner:** A dropdown menu showing "teacher-liu".
- Repository name:** A text input field containing "project\_02".
- Description (optional):** A text input field containing "这是一个测试仓库".
- Visibility:** Two radio buttons: "Public" (selected) and "Private".
- Initialize this repository with a README:** A checkbox that is currently unchecked.
- Add .gitignore:** A dropdown menu showing "None".
- Add a license:** A dropdown menu showing "None".
- Create repository:** A green button at the bottom.

Numbered annotations (1-5) are placed on the page:

1. The "+" button in the top right corner of the GitHub header.
2. The "New repository" option in the dropdown menu that appears after clicking the "+" button.
3. The "Repository name" input field.
4. The "Description (optional)" input field.
5. The "Create repository" button.



## 2. 新建空白远程仓库成功

The screenshot shows the GitHub interface for a repository named 'project\_02' by user 'teacher-liu'. The repository has 1 watch, 0 stars, and 0 forks. The 'Code' tab is selected, showing the 'Quick setup' section. This section provides instructions for cloning the repository using HTTPS or SSH, and for creating a new repository on the command line. The command line instructions are as follows:

```
echo "# project_02" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/teacher-liu/project_02.git
git push -u origin master
```

Below this, there is a section for pushing an existing repository from the command line with the following instructions:

```
git remote add origin https://github.com/teacher-liu/project_02.git
git push -u origin master
```

The bottom of the screenshot shows the beginning of a section for importing code from another repository.

## 3. 远程仓库的两种访问方式

Github 上的远程仓库，有两种访问方式，分别是 **HTTPS** 和 **SSH**。它们的区别是：

- ① **HTTPS**：零配置；但是每次访问仓库时，需要重复输入 Github 的账号和密码才能访问成功
- ② **SSH**：需要进行额外的配置；但是配置成功后，每次访问仓库时，不需重复输入 Github 的账号和密码


注意：在实际开发中，推荐使用 **SSH** 的方式访问远程仓库。



# Github - 远程仓库的使用

## 4. 基于 HTTPS 将本地仓库上传到 Github

Quick setup — if you've done this kind of thing before

 Set up in Desktop

or

HTTPS

SSH

`https://github.com/teacher-liu/project_02.git`



Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

### ① 本地没有现成的 Git 仓库

...or create a new repository on the command line

```
echo "# project 02" >> README.md
git init
git add README.md
git commit -m "first commit"
```

1. 使用终端命令创建 README.md 文档，并写入初始内容为 # project 02

2. 初始化本地 Git 仓库，并将文件的修改提交到本地的 Git 仓库中

```
git remote add origin https://github.com/teacher-liu/project_02.git
git push -u origin master
```

3. 将本地仓库和远程仓库进行关联，并把远程仓库命名为 origin

4. 将本地仓库中的内容推送到远程的 origin 仓库中

### ② 本地有现成的 Git 仓库

...or push an existing repository from the command line

```
git remote add origin https://github.com/teacher-liu/project_02.git
git push -u origin master
```

1. 将本地仓库和远程仓库进行关联，并把远程仓库命名为 origin

2. 将本地仓库中的内容推送到远程的 origin 仓库中

## 5. SSH key

SSH key 的**作用**：实现本地仓库和 Github 之间**免登录**的**加密数据传输**。

SSH key 的**好处**：免登录身份认证、数据加密传输。

SSH key 由**两部分组成**，分别是：

- ① id\_rsa（私钥文件，存放于客户端的电脑中即可）
- ② id\_rsa.pub（公钥文件，需要配置到 Github 中）

## 6. 生成 SSH key

- ① 打开 Git Bash
- ② 粘贴如下的命令，并将 your\_email@example.com 替换为注册 Github 账号时填写的邮箱：
  - `ssh-keygen -t rsa -b 4096 -C "your_email@example.com"`
- ③ 连续敲击 3 次回车，即可在 C:\Users\用户名文件夹\.ssh 目录中生成 id\_rsa 和 id\_rsa.pub 两个文件

## 7. 配置 SSH key

- ① 使用记事本打开 `id_rsa.pub` 文件，复制里面的文本内容
- ② 在浏览器中登录 Github，[点击头像](#) -> [Settings](#) -> [SSH and GPG Keys](#) -> [New SSH key](#)
- ③ 将 `id_rsa.pub` 文件中的内容，[粘贴到 Key 对应的文本框中](#)
- ④ 在 Title 文本框中任意填写一个名称，来标识这个 Key 从何而来



## 8. 检测 Github 的 SSH key 是否配置成功

打开 Git Bash, 输入如下的命令并回车执行:

```
1 ssh -T git@github.com
```

上述的命令执行成功后, 可能会看到如下的提示消息:

```
1 The authenticity of host 'github.com (IP ADDRESS)' can't be established.  
2 RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.  
3 Are you sure you want to continue connecting (yes/no)?
```

输入 yes 之后, 如果能看到类似于下面的提示消息, 证明 SSH key 已经配置成功了:


```
1 Hi username! You've successfully authenticated, but GitHub does not  
2 provide shell access.
```



# Github - 远程仓库的使用

## 9. 基于 SSH 将本地仓库上传到 Github

Quick setup — if you've <sup>1</sup> done this kind of thing before

 Set up in Desktop

or

HTTPS

SSH

git@github.com:teacher-liu/project\_03.git



Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# project_03" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:teacher-liu/project_03.git
git push -u origin master
```



将本地现成的仓库推送到 Github

...or push an existing repository from the command line

2

```
git remote add origin git@github.com:teacher-liu/project_03.git
git push -u origin master
```

1. 将本地仓库和远程仓库进行关联, 并把远程仓库命名为 origin

2. 将本地仓库中的内容推送到远程的 origin 仓库中



## 10. 将远程仓库克隆到本地

打开 Git Bash，输入如下的命令并回车执行：

```
1 git clone 远程仓库的地址
```

# 目录 Contents

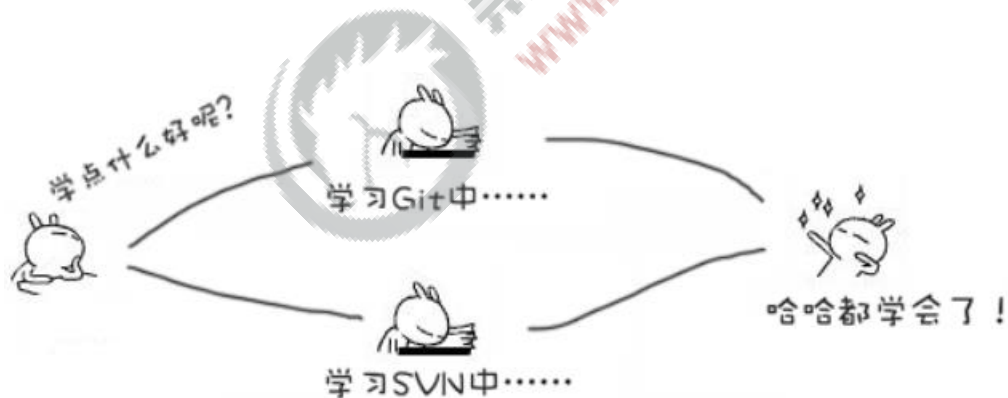
- ◆ 起步
- ◆ Git 基础
- ◆ Github
- ◆ Git 分支

## 1. 分支的概念

分支就是科幻电影里面的**平行宇宙**，当你正在电脑前努力学习Git的时候，另一个你正在另一个平行宇宙里努力学习SVN。

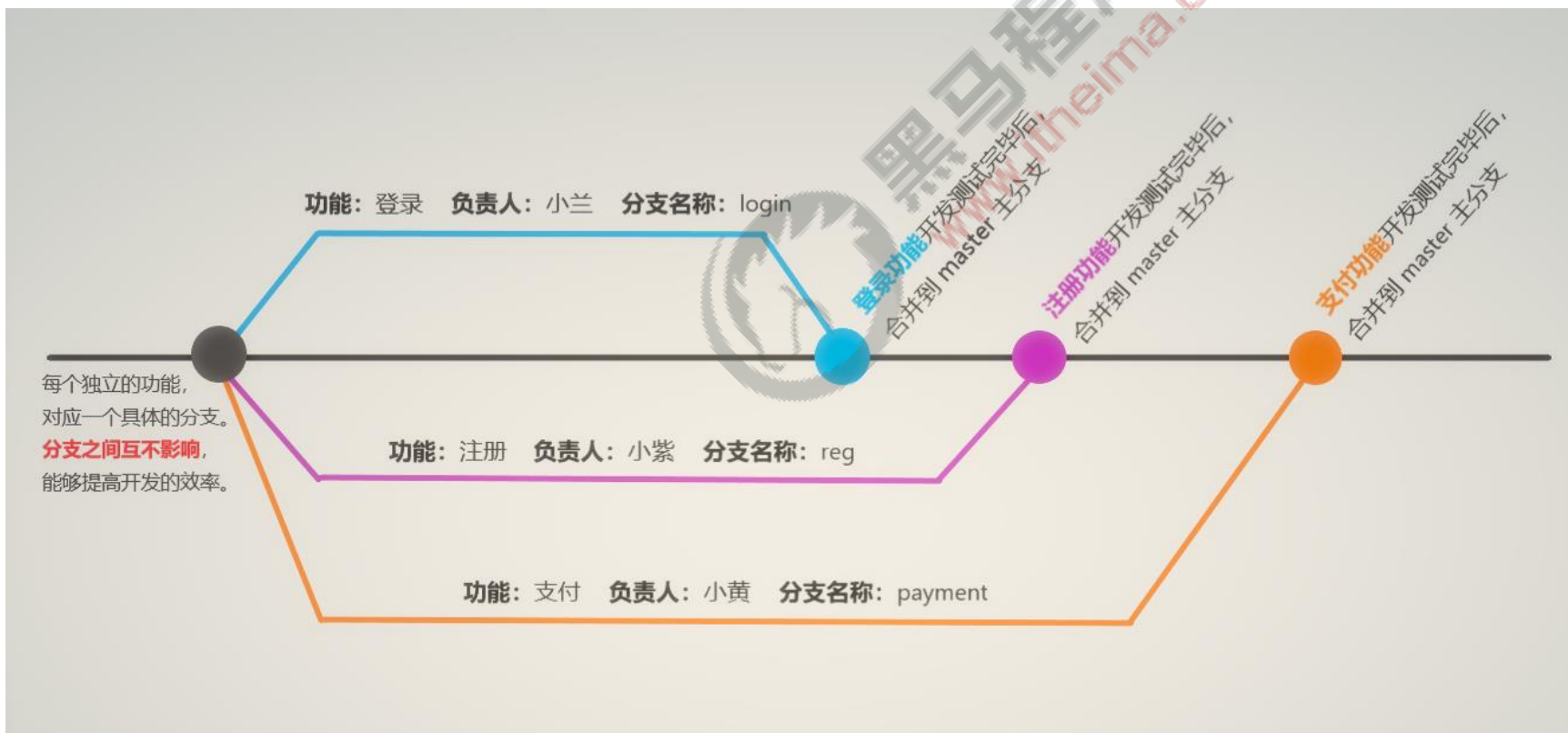
如果两个平行宇宙互不干扰，那对现在的你也没啥影响。

不过，**在某个时间点，两个平行宇宙合并了**，结果，你既学会了Git又学会了SVN！



## 2. 分支在实际开发中的作用

在进行多人协作开发的时候，为了防止互相干扰，提高协同开发的体验，建议每个开发者都基于分支进行项目功能的开发，例如：



## 3. master 主分支

在初始化本地 Git 仓库的时候，Git 默认已经帮我们创建了一个名字叫做 **master** 的分支。通常我们把这个 master 分支叫做**主分支**。



在实际工作中，master 主分支的作用是：**用来保存和记录整个项目已完成的功能代码。**

因此，不允许程序员直接在 master 分支上修改代码，因为这样做的风险太高，容易导致整个项目崩溃。

## 4. 功能分支

由于程序员不能直接在 master 分支上进行功能的开发，所以就有了**功能分支**的概念。

**功能分支**指的是**专门用来开发新功能的分支**，它是临时从 master 主分支上分叉出来的，当新功能开发且测试完毕后，最终需要合并到 master 主分支上，如图所示：



## 5. 查看分支列表

使用如下的命令，可以查看当前 Git 仓库中所有的分支列表：

```
1 git branch
```

运行的结果如下所示：

```
C:\Windows\System32\cmd.exe  
  
E:\code2>git branch  
* master  
  
E:\code2>
```

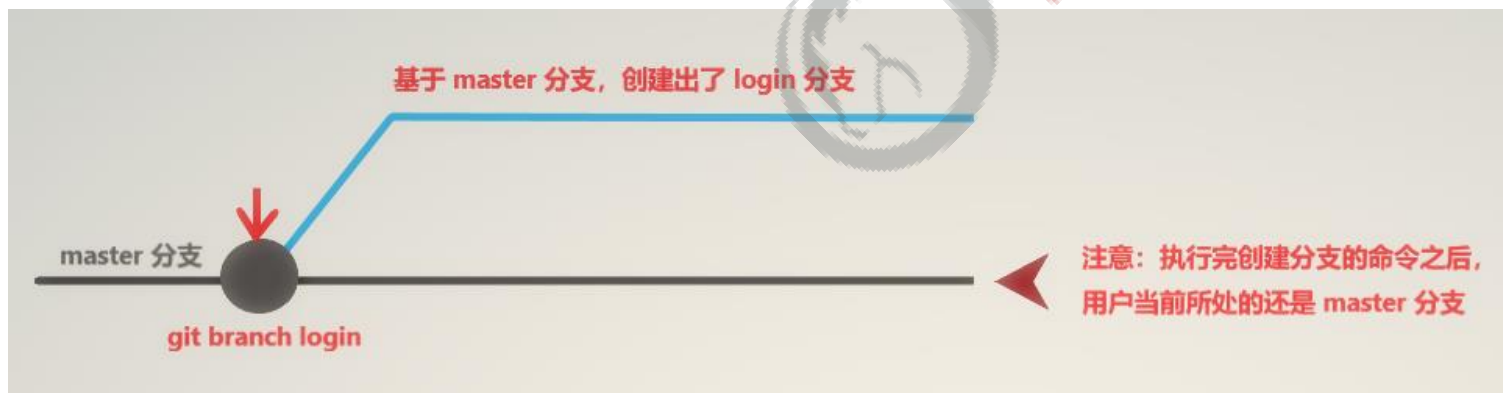
❗ 注意：分支名字前面的 \* 号表示当前所处的分支。

## 6. 创建新分支

使用如下的命令，可以基于当前分支，创建一个新的分支，此时，新分支中的代码和当前分支完全一样：

```
1 git branch 分支名称
```

图示如下：





## 7. 切换分支

使用如下的命令，可以切换到指定的分支上进行开发：

```
1 git checkout login
```

图示如下：

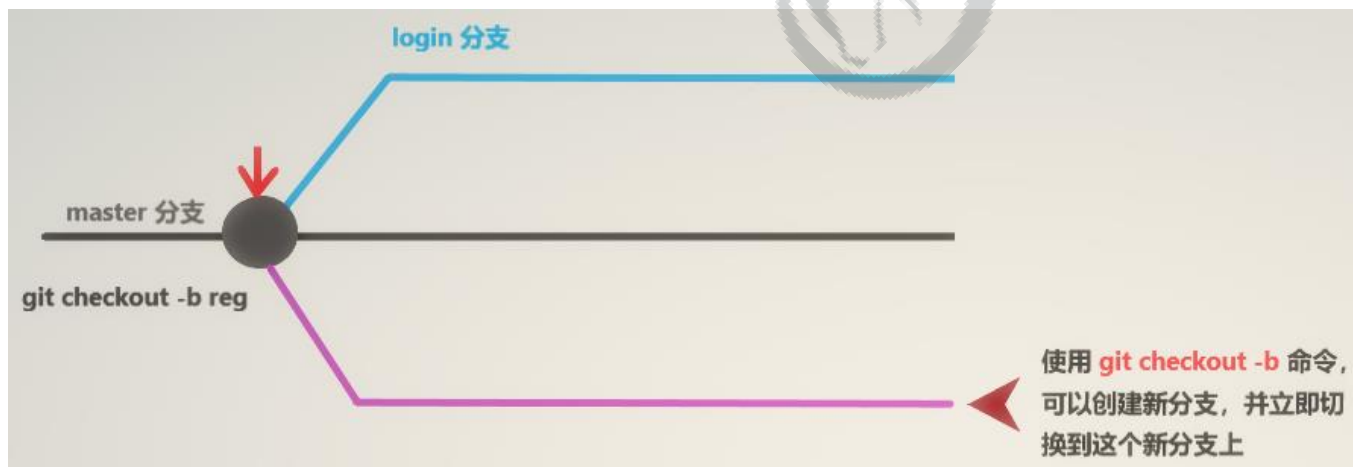


## 8. 分支的快速创建和切换

使用如下的命令，可以创建指定名称的新分支，并立即切换到新分支上：

```
1 # -b 表示创建一个新分支
2 # checkout 表示切换到刚才新建的分支上
3 git checkout -b 分支名称
```

图示如下：



注意：

"git checkout -b 分支名称" 是下面两条命令的简写形式：

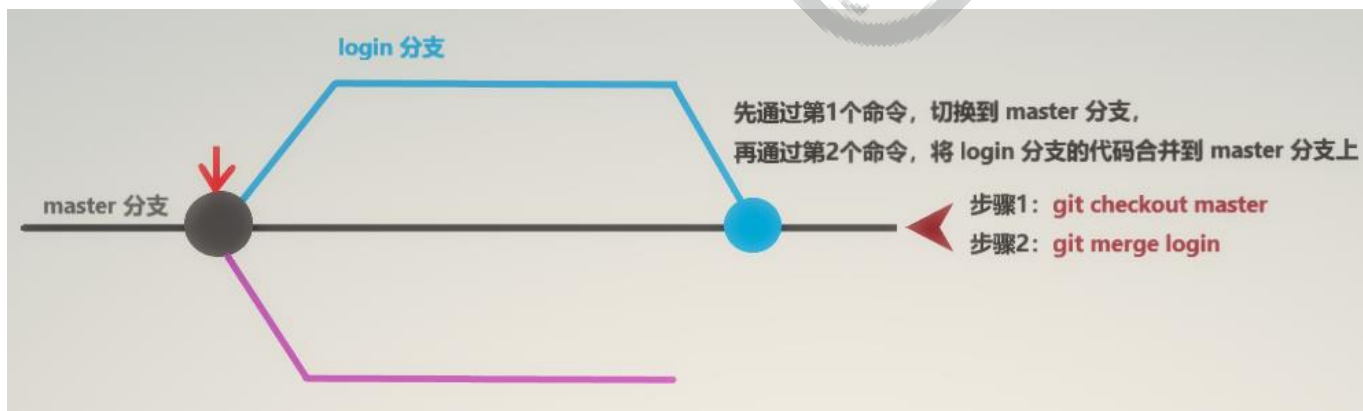
- ① git branch 分支名称
- ② git checkout 分支名称

## 9. 合并分支

功能分支的代码开发测试完毕之后，可以使用如下的命令，将完成后的代码合并到 master 主分支上：

```
1 # 1. 切换到 master 分支
2 git checkout master
3 # 2. 在 master 分支上运行 git merge 命令，将 login 分支的代码合并到 master 分支
4 git merge login
```

图示如下：



合并分支时的注意点：

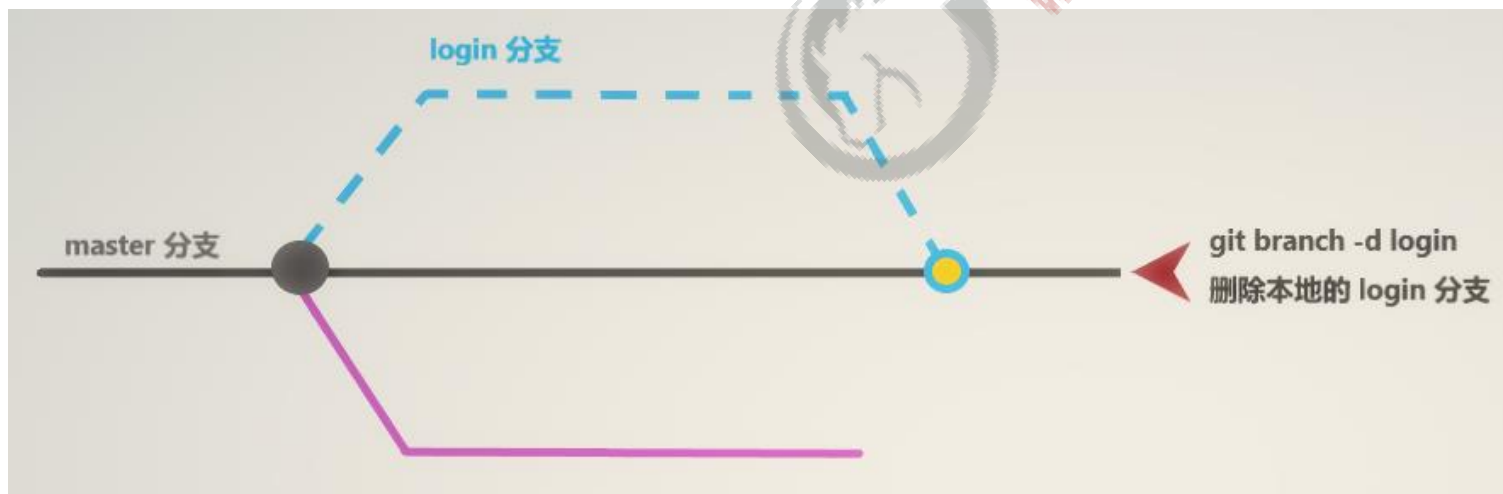
假设要把 C 分支的代码合并到 A 分支，  
则必须**先切换到 A 分支上**，**再运行 git merge 命令**，来合并 C 分支！

## 9. 删除分支

当把功能分支的代码合并到 master 主分支上以后，就可以使用如下的命令，删除对应的功能分支：

```
1 git branch -d 分支名称
```

图示如下：





## 10. 遇到冲突时的分支合并

如果在两个不同的分支中，对同一个文件进行了不同的修改，Git 就没法干净的合并它们。此时，我们需要打开这些包含冲突的文件然后**手动解决冲突**。

```
1 # 假设：在把 reg 分支合并到 master 分支期间，代码发生了冲突
2 git checkout master
3 git merge reg
4
5 # 打开包含冲突的文件，手动解决冲突之后，再执行如下的命令
6 git add .
7 git commit -m "解决了分支合并冲突的问题"
```

## 1. 将本地分支推送到远程仓库

如果是**第一次**将本地分支推送到远程仓库，需要运行如下的命令：

```
1 # -u 表示把本地分支和远程分支进行关联，只在第一次推送的时候需要带 -u 参数
2 git push -u 远程仓库的别名 本地分支名称:远程分支名称
3
4 # 实际案例:
5 git push -u origin payment:pay
6
7 # 如果希望远程分支的名称和本地分支名称保持一致，可以对命令进行简化:
8 git push -u origin payment
```

注意：第一次推送分支需要带 **-u 参数**，此后可以直接使用 **git push** 推送代码到远程分支。

## 2. 查看远程仓库中所有的分支列表

通过如下的命令，可以查看远程仓库中，所有的分支列表的信息：

```
1 git remote show 远程仓库名称
```

## 3. 跟踪分支

跟踪分支指的是：从远程仓库中，把远程分支下载到本地仓库中。需要运行的命令如下：

```
1 # 从远程仓库中，把对应的远程分支下载到本地仓库，保持本地分支和远程分支名称相同
2 git checkout 远程分支的名称
3 # 示例：
4 git checkout pay
5
6 # 从远程仓库中，把对应的远程分支下载到本地仓库，并把下载的本地分支进行重命名
7 git checkout -b 本地分支名称 远程仓库名称/远程分支名称
8 # 示例：
9 git checkout -b payment origin/pay
```



### 4. 拉取远程分支的最新的代码

可以使用如下的命令，把远程分支最新的代码下载到本地对应的分支中：

```
1 # 从远程仓库，拉取当前分支最新的代码，保持当前分支的代码和远程分支代码一致
2 git pull
```

## 5. 删除远程分支

可以使用如下的命令，删除远程仓库中指定的分支：

```
1 # 删除远程仓库中，指定名称的远程分支
2 git push 远程仓库名称 --delete 远程分支名称
3 # 示例：
4 git push origin --delete pay
```



## 总结

### ① 能够掌握 Git 中基本命令的使用

- git init
- git add .
- git commit -m "提交消息"
- git status 和 git status -s

### ② 能够使用 Github 创建和维护远程仓库

- 能够配置 Github 的 SSH 访问
- 能够将本地仓库上传到 Github

### ③ 能够掌握 Git 分支的基本使用

- git checkout -b 新分支名称
- git push -u origin 新分支名称
- git checkout 分支名称
- git branch



黑马程序员

[www.itheima.com](http://www.itheima.com)

传智播客旗下高端IT教育品牌