目标

- 了解版本控制软件的作用
- 了解版本控制系统的分类
- Git的特性
- 初始化 Git 仓库的命令
- 查看文件状态的命令
- 一次性将文件加入暂存区的命令
- 将暂存区的文件提交到 Git 仓库的命令

起步

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

版本控制软件(☆☆☆)

概念

版本控制软件是一个用来记录文件变化，以便将来查阅特定

版本修订情况的系统，因此有时也叫做“**版本控制系统**”

通俗的理解

把**手工管理**文件版本的方式，改为由**软件管理**文件的版本；

这个**负责管理文件版本的软件**，叫做“版本控制软件”

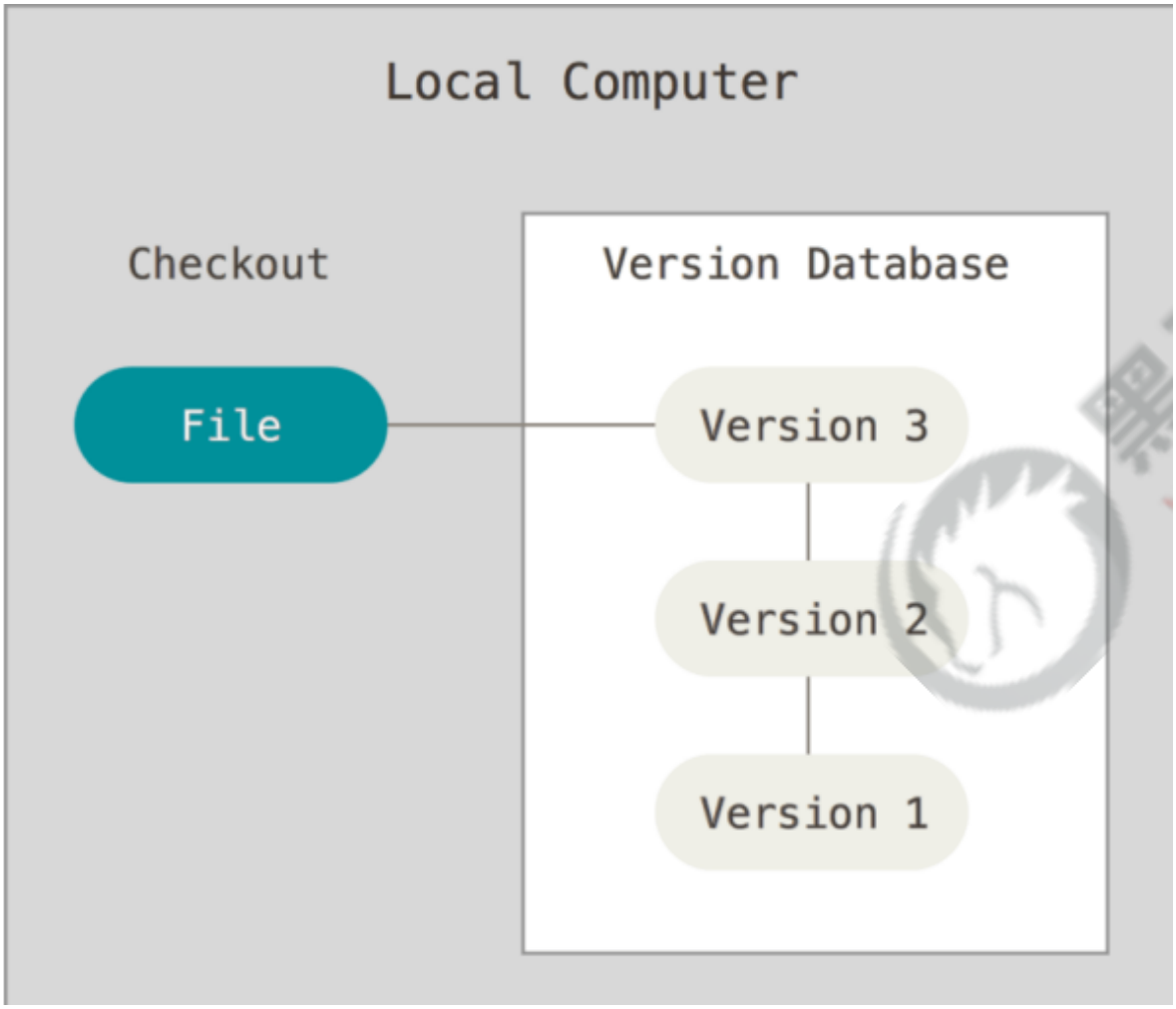
使用版本控制软件的好处

- **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

版本控制系统的分类(☆☆☆)

本地版本控制系统

单机运行，使维护文件版本的操作工具化



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

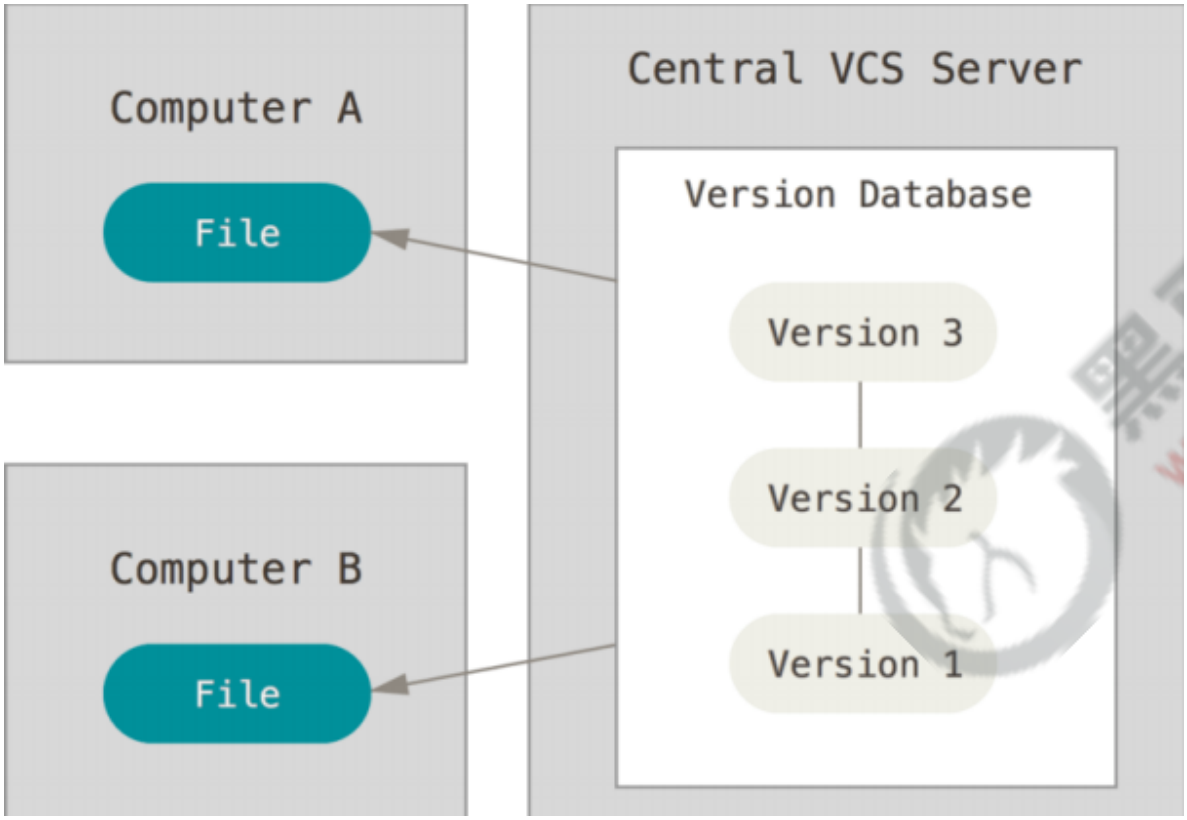
缺点：

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

集中化的版本控制系统

联网运行，支持多人协作开发；**性能差、用户体验不好**



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：

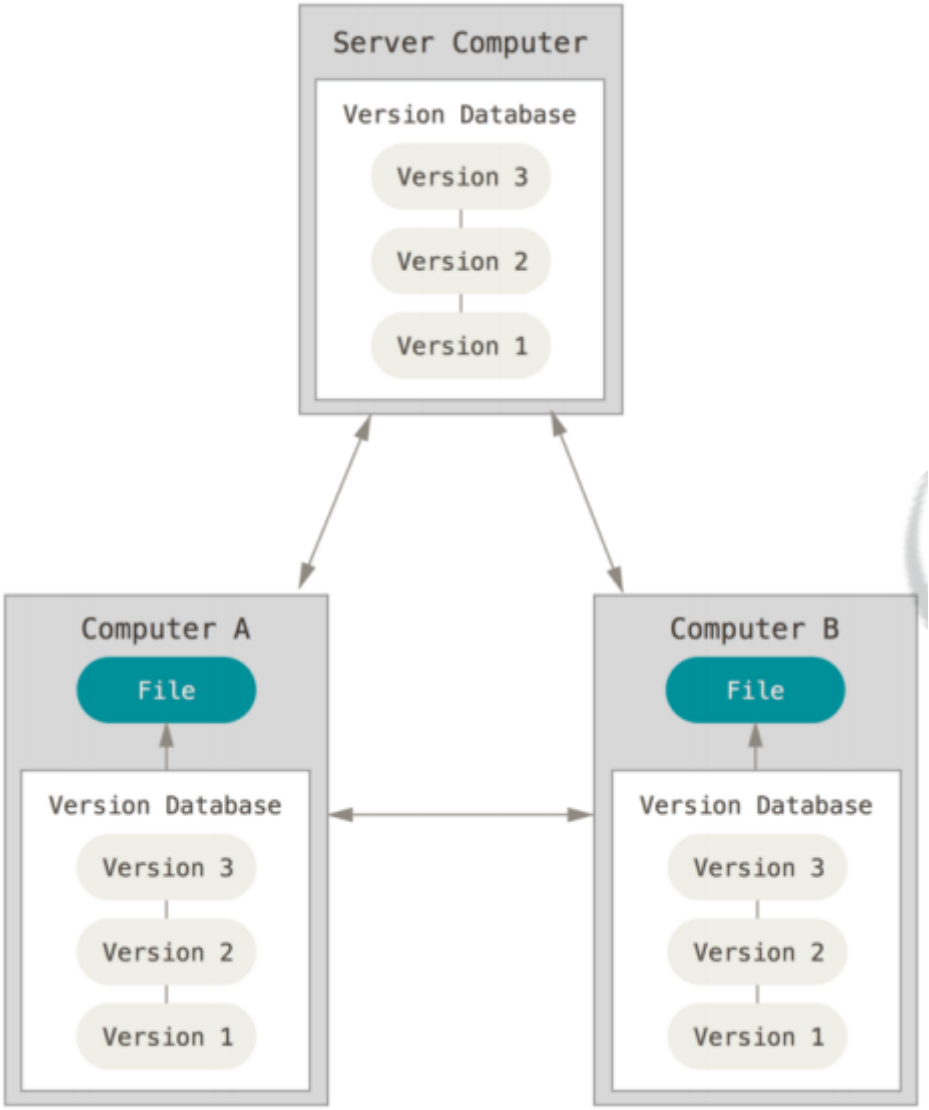
① 不支持离线提交版本更新

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

分布式版本控制系统

联网运行，支持多人协作开发；性能优秀、用户体验好



典型代表： `Git`

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版本

② **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

Git基础概念

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**！

Git 的特性(☆☆☆)

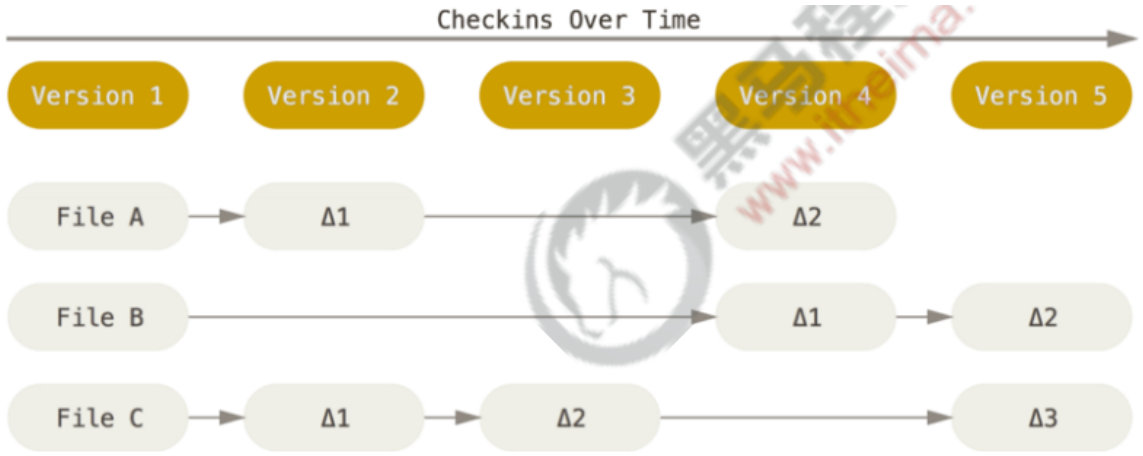
Git 之所以快速和高效，主要依赖于它的如下两个特性：

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

`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

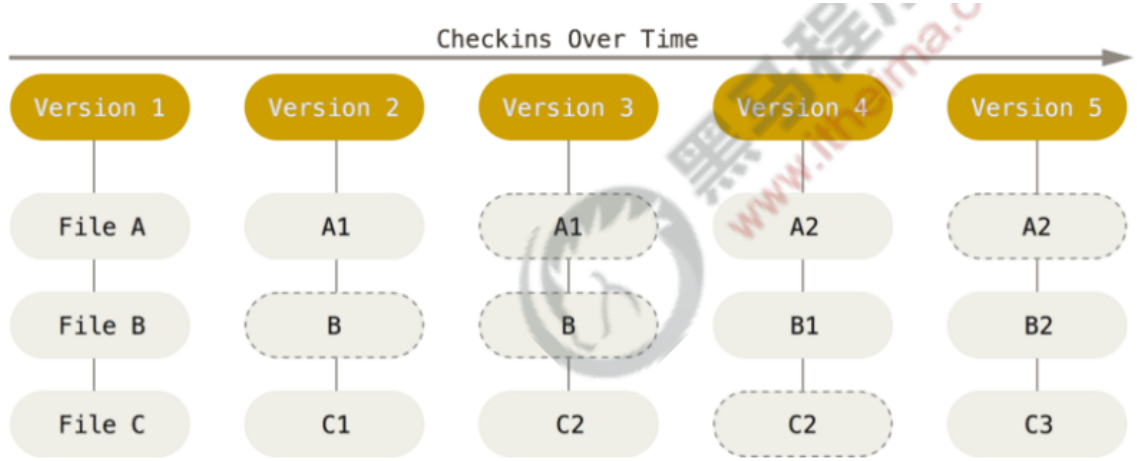
缺点：耗时、效率低

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

`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

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

`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

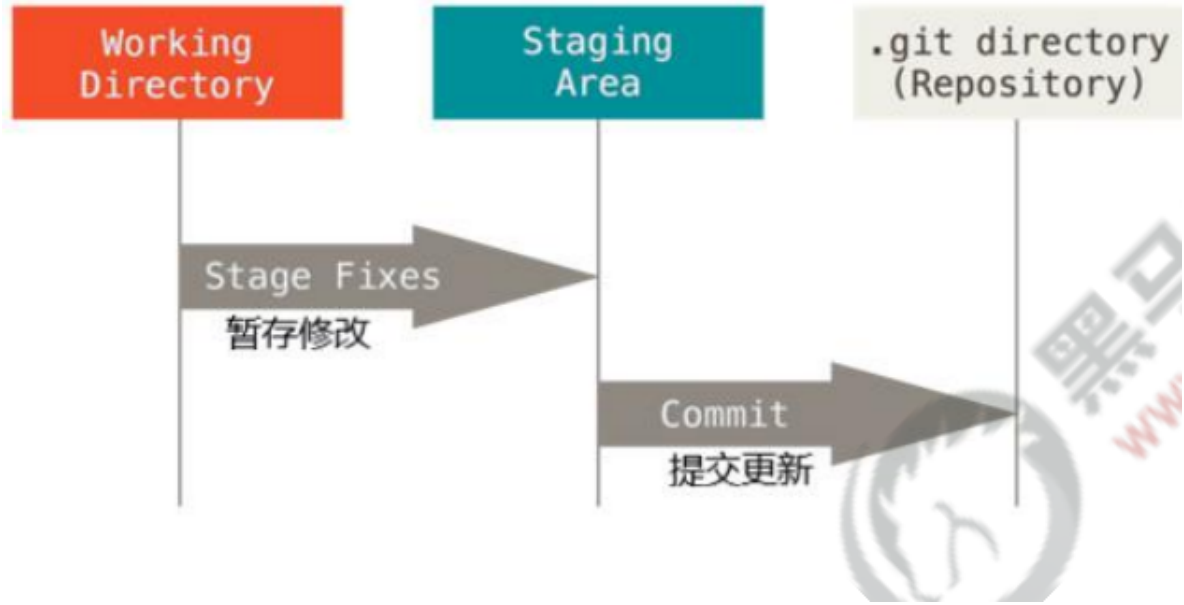
`Git` 中的三种状态

- **已修改** `modified`
  - 表示修改了文件，但还没将修改的结果放到**暂存区**
- **已暂存** `staged`
  - 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** `committed`
  - 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。

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

`Git`基础

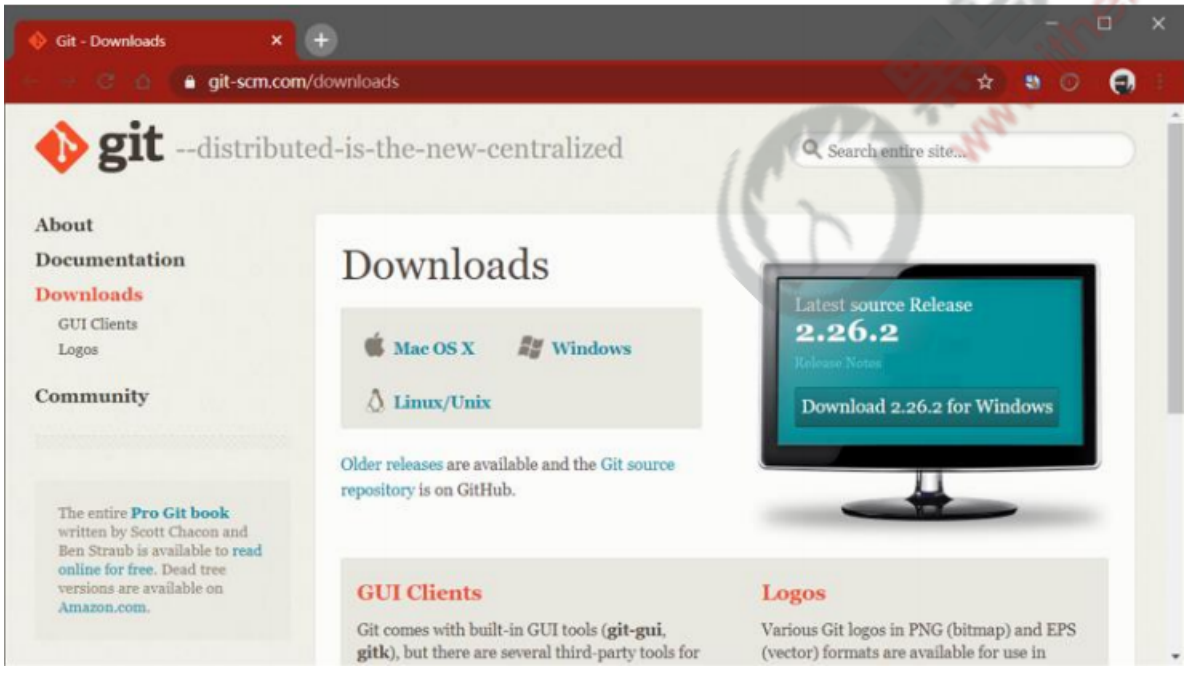
安装并配置 Git

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



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：

```
git config --global user.name "itheima"
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。

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。

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：

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

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：

```
# 打开 git config 命令的帮助手册
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：

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

Git 的基本操作

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

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：

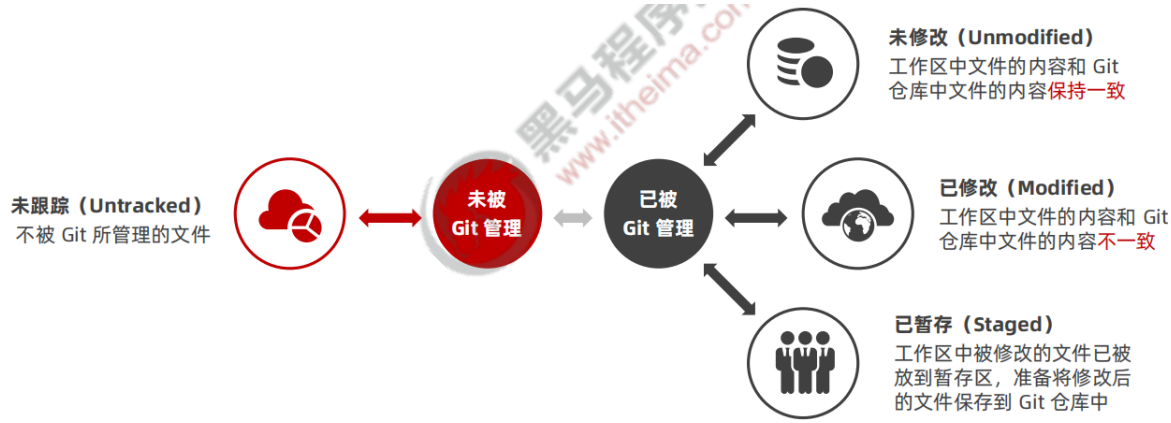
① 在项目目录中，通过鼠标右键打开“Git Bash”

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**

工作区中文件的 4 种状态

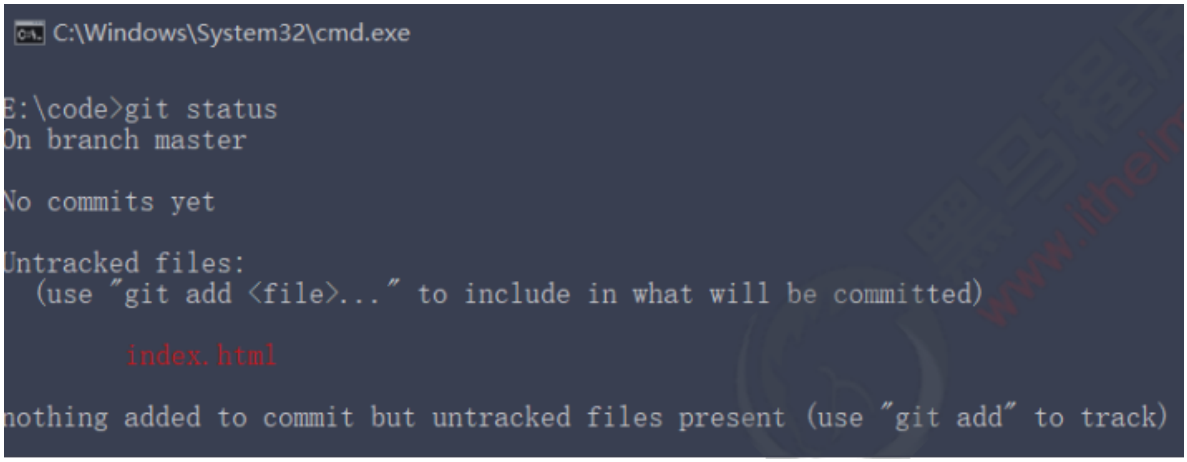
工作区中的每一个文件可能有 4 种状态，这四种状态共分为两大类，如图所示：



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。

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



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”。

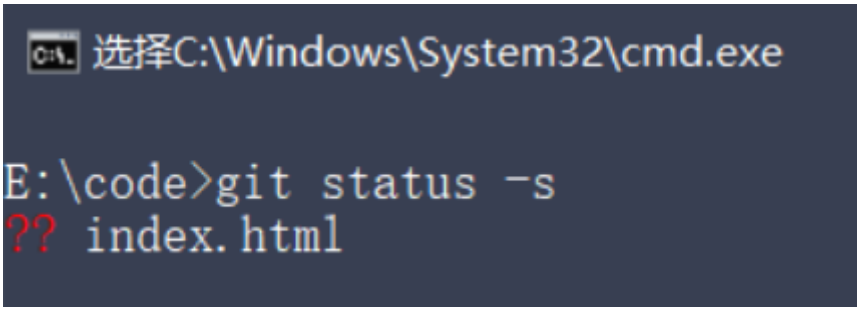
以精简的方式显示文件状态

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：

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

未跟踪文件前面有红色的 **??** 标记，例如：

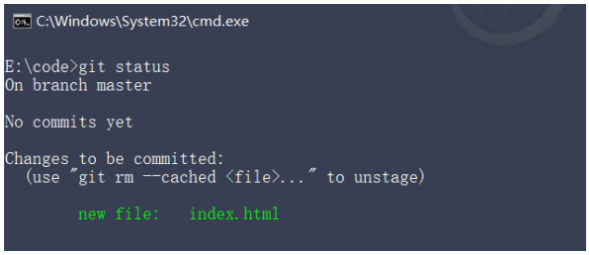


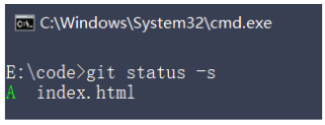
跟踪新文件(☆☆☆)

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：

```
git add index.html
# 如果文件过多，你项跟踪目录下所有文件
git add *.*
```

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：





以精简的方式显示文件的状态：

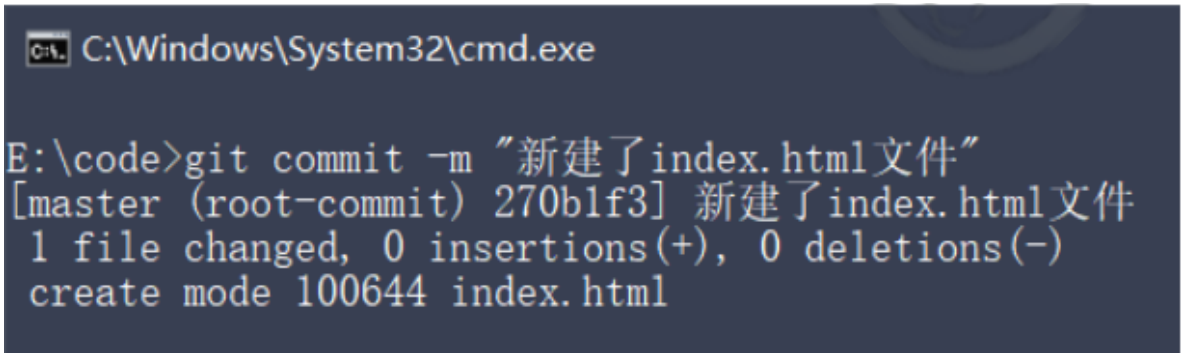
新添加到暂存区中的文件前面有绿色的 A 标记

提交更新(☆☆☆)

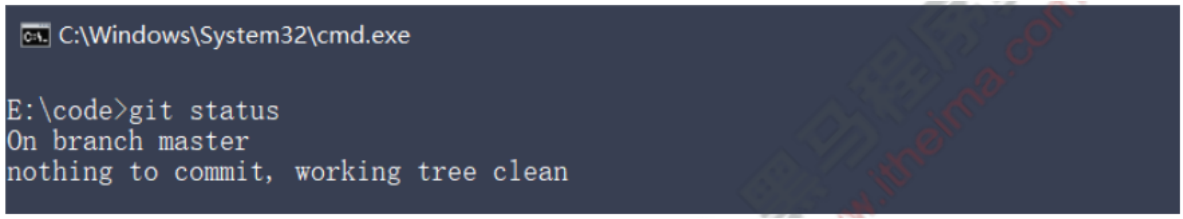
现在暂存区中有一个 `index.html` 文件等待被提交到 `Git` 仓库中进行保存。可以执行 `git commit` 命令进行提交,其中 `-m` 选项后面是本次的提交消息，用来对提交的内容做进一步的描述：

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

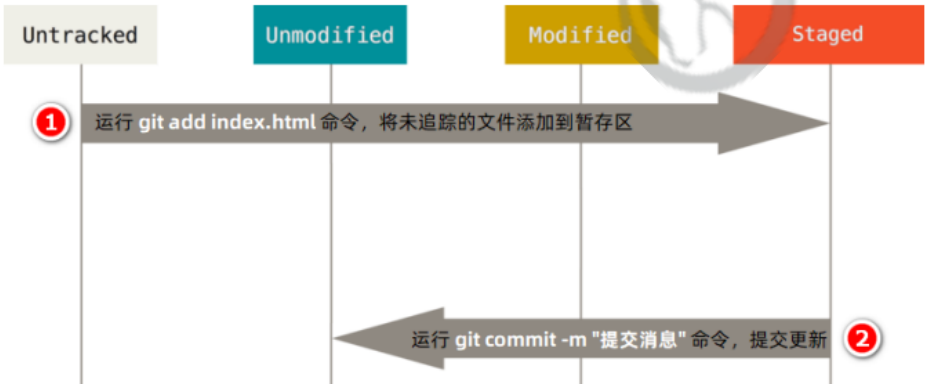
提交成功之后，会显示如下的信息：



提交成功之后，再次检查文件的状态，得到提示如下：

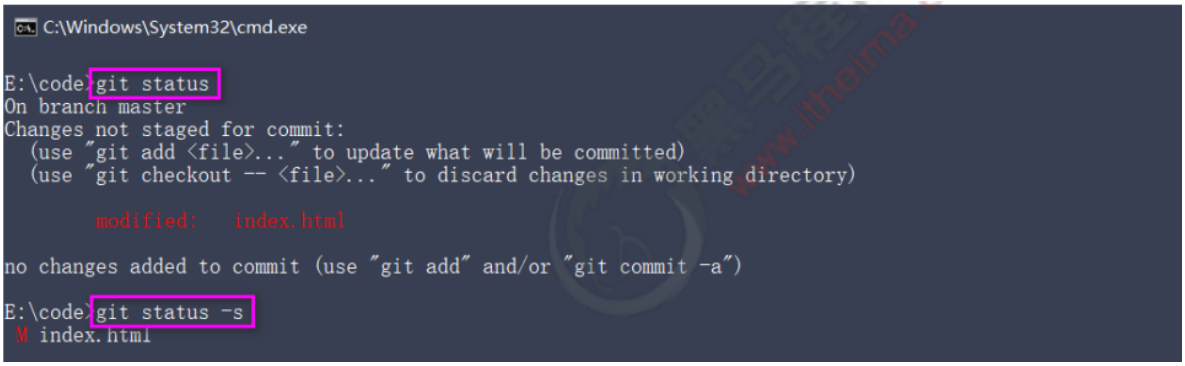


证明工作区中所有的文件都处于“未修改”的状态，没有任何文件需要被提交。



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：



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。

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：

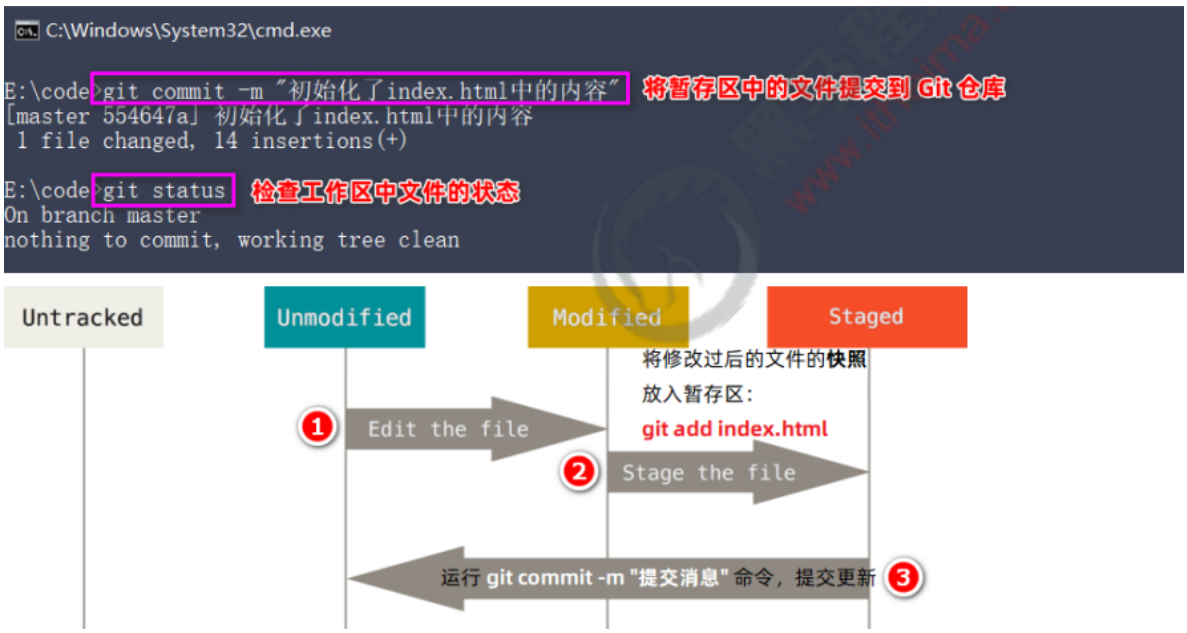
① 可以用它**开始跟踪新文件**

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

提交已暂存的文件

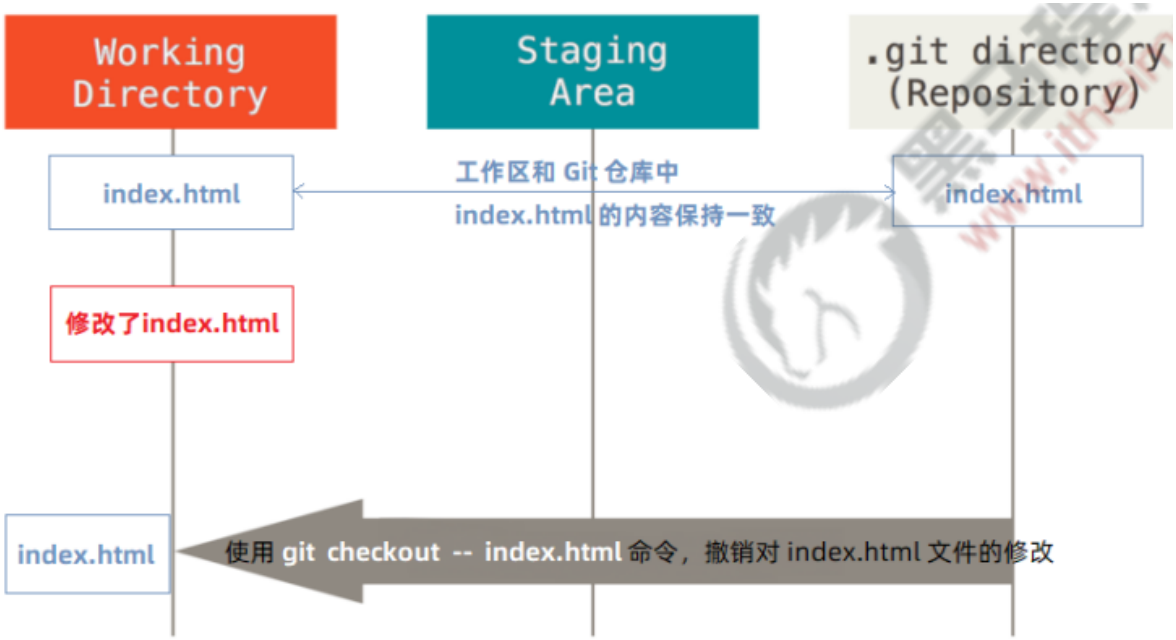
再次运行 `git commit -m "提交消息"` 命令，即可将暂存区中记录的 `index.html` 的快照，提交到 `Git` 仓库中进行保存：



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！**



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。

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：

```
git add .
```

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

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：

```
git reset HEAD 要移出的文件名称
```

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：

```
git commit -a -m "日志信息"
```

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

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

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）

- **星号 *** 匹配**零个或多个任意字符**
- `[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）
- **问号 ?** 只匹配**一个任意字符**
- **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）
- 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）

`.gitignore` 文件的例子

```
# 忽略所有的 .a 文件
*.a

# 但跟踪所有的 lib.a，即便你在前面忽略了 .a 文件
!lib.a

# 只忽略当前目录下的 TODO 文件，而不忽略 subdir/TODO
/TODO

# 忽略任何目录下名为 build 的文件夹
build/

# 忽略 doc/notes.txt，但不忽略 doc/server/arch.txt
doc/*.txt

# 忽略 doc/ 目录及其所有子目录下的 .pdf 文件
doc/**/*.pdf
```

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

```
# 按时间先后顺序列出所有的提交历史，最近的提交在最上面
git log

# 只展示最新的两条提交历史，数字可以按需进行填写
git log -2

# 在一行上展示最近两条提交历史的信息
git log -2 --pretty=oneline

# 在一行上展示最近两条提交历史信息，并自定义输出的格式
# &h 提交的简写哈希值  %an 作者名字  %ar 作者修订日志  %s 提交说明
git log -2 --pretty=format:"%h | %an | %ar | %s"
```

回退到指定的版本

```
# 在一行上展示所有的提交历史
git log --pretty=oneline

# 使用 git reset --hard 命令，根据指定的提交 ID 回退到指定版本
git reset --hard <CommitID>

# 在旧版本中使用 git reflog --pretty=oneline 命令，查看命令操作的历史
git reflog --pretty=onelone

# 再次根据最新的提交 ID，跳转到最新的版本
git reset --hard <CommitID>
```