Git

分布式版本控制管理器

1.git的安装和配置

1.1 下载

下载地址：Git - Downloads

备注：

- Git GUI：Git提供的图像界面工具
- Git Bash：Git提供的命令行工具

1.2 基本配置

1. 打开Git Bash
2. 设置用户信息

- ```
  git config --global user.name "Caaat"
  ```
- ```
  git config --global user.email "Caaat@gmail.com"
  ```

查看配置信息

- ```
  git config --global user.name
  ```
- ```
  git config --global user.email
  ```

2.git基本操作指令

2.1 git init（初始化）

初始化git仓库

1. 选择一个目录，打开命令行窗口（需要配置好环境变量）或者Git bash窗口
2. 执行命令`git init`
3. 创建成功会在此目录下生成一个隐藏的`.git`目录

命令形式：

```
git init
```

2.2 git add / git commit（提交）

一次提交就是一个版本

1. `git add .`：将**工作区**内容提交到**暂存区**
2. `git commit -m "注释"`：将**暂存区**内容提交到**本地仓库**的**当前分支**

工作区 -> git add -> 暂存区 -> git commit -> 本地仓库

命令形式：

```
git add 文件名
```

```
git commit -m "注释"
```

2.3 git log（查看修改日志）

提交日志

命令形式：

```
git log [option]
```

- options
  - --all
  - --pretty
  - --graph
  - --abbrev-commit

2.4 git status（查看修改状态）

查看修改状态

命令形式：

```
git status
```

2.5 git reset（版本回退）

版本切换

命令形式：

```
git reset --hard commitID
```

- commitID
  - 可以使用`git log`查看
- 如何查看已经删除的记录
  - ```
    git reflog
    ```
    ```

- 所有操作日志

3.忽略列表.gitignore

3.1 创建git忽略列表

- 在与`.git`同级目录下创建文件`.gitignore`
- 修改`.gitignore`文件内容，添加要忽略的目标

3.2 忽略列表

写法	含义
`# 注释`	注释说明，不会影响忽略规则
`*.log`	忽略所有 `.log` 后缀的文件
`*.tmp`	忽略所有 `.tmp` 后缀的文件
`build/`	忽略 `build` 文件夹及其下所有内容
`!important.txt`	不忽略 `important.txt`，即使之前的规则匹配
`/config/config.yaml`	只忽略项目根目录下的这个文件，不影响子目录
`**/temp`	忽略所有路径中名为 `temp` 的目录
`debug/*`	忽略 `debug` 文件夹下的所有内容，但不忽略子文件夹中的内容
`debug/**`	忽略 `debug` 文件夹下所有内容（包括子文件夹内容）

4.Git分支

几乎所有的版本控制系统都以某种形式支持分支。使用分支意味着你可以把你的工作从开发主线上分离开来进行重大的bug修改、开发新的功能，以免影响主线开发

HEAD:使用`git log`时，`HEAD->`指向哪个分支，当前就是哪个分支

4.1 git branch（查看本地分支/创建新分支）

查看本地分支/创建新分支

命令形式：

```
git branch [新分支名]
```

4.2 git checkout (切换分支)

切换分支

命令形式：

```
git checkout 分支名
```

也可直接创建并切换到一个当前不存在的分支:

```
git checkout -b 分支名
```

4.3 git merge (合并分支)

将另一个分支的内容**合并**到你当前所在的分支上。

一般情况下都是将其他分支合并到`master`分支上

1. 切换到master分支
2. 使用`git merge`指令

命令形式:

```
git merge 分支名称
```

4.4 删除分支

不能删除当前分支，只能删除其他分支

命令形式:

- 删除分支时，需要做各种检查

```
git branch -d b1
```

- 不做任何检查，强制删除

```
git branch -D b1
```

4.5 解决冲突

当两个分支上对文件的修改可能存在冲突，比如同时修改了一个文件的同一行，`git merge`时会提示冲突地方，需要手动解决冲突

如:

```
<<<<<<<< HEAD
当前分支代码
=========
合并分支上的代码
>>>>>>>> dev
```

解决冲突:

1. 处理文件中冲突的地方
2. 将解决完冲突的文件加入暂存区
3. 提交到仓库（commit）

5.远程仓库

5.1 常用的远程仓库

- github：面向开源及私有软件项目托管平台，只支持Git作为唯一的版本库格式进行管理
- gitee：即码云，国内的一个代码托管平台
- gitlab：用于仓库管理系统的开源项目，使用Git作为代码管理工具，并在此基础上搭建起来的web服务，一般用于企业、学校等内部网络搭建git私服

5.2 创建远程仓库

1. 选择一个远程仓库平台，注册登录
2. 创建远程仓库
3. 选择推送方式（Http/ssh）

5.3 配置SSH公钥

- 生成SSH公钥
  - ```
    ssh-keygen -t rsa
    ```
  - 不断回车，如果公钥已经存在，则会自动覆盖
- 远程仓库设置账户公钥
  - 获取公钥
    - ```
      cat ~/.ssh/id_rsa.pub
      ```
  - 远程仓库的设置中添加SSH公钥
  - 验证是否配置成功
    - ```
      ssh-T git@github.com
      ```

5.4 项目绑定远程仓库

命令形式：

```
git remote add <远程仓库默认别名> <远程仓库URL>
```

远程仓库默认别名通常使用`origin`，`origin`是你给远程仓库起的名字，通常用作默认别名，方便后续的推送/拉取操作。

也可查看远程仓库有哪些

命令形式：

```
git remote
```

如果设置错了远程仓库，可以删除再添加

命令形式：
```

```
git remote remove origin
```

5.5 推送

将本地仓库的内容推送到远程仓库

命令形式：

```
git push [-f] [-set-upstream] [远程仓库别名 [本地仓库分支名][:远程仓库分支名]]
```

- 如果远程分支名和本地分支名相同，则可以只写本地分支名
  - ```
    git push origin master
    ```
- `[-f]` 强制覆盖
- `[--set-upstream]` 推送到远程的同时并建立起和远程分支的关联关系
- 如果**当前分支已经和远程分支关联**，则可以省略分支名和远程名
  - ```
    git push
    ```
  - 将master分支推送到已经关联的远程分支

注意：**从Git2.0开始，`[-u]` 是 `[--set-upstream]` 的简写**

> ### 查看当前本地分支的 upstream 设置：
>
> ```
> git branch -vv
> ```
>
> 输出示例：
>
> ```
> * feature-1 1234abc [origin/feature-1] work in progress
> ```
>
> 表示当前分支已绑定到 `origin/feature-1`

6.从远程仓库获取到本地

6.1 clone (克隆)

`git clone` 是用来 **从远程仓库复制一个完整项目** 到你本地的命令。并且自动设置好了远程仓库（通常叫 `origin`）

它不仅复制代码，还包含了整个 Git 的历史记录（提交记录、分支、标签等）。

命令形式：

```
git clone <远程仓库地址> [本地目录名]
```

- 不加本地目录名就**默认使用远程项目名**作为文件夹名

6.2 抓取

远程分支和本地的分支一样，我们可以进行merge操作，只需要先把远程仓库的更新都下载到本地，再进行操作

1.抓取：将仓库里的更新都抓取到本地，不会进行合并

命令形式：

```
git fetch [远程仓库别名] [分支名]
```

- 不会改变当前工作区
- 如果不指定远程名称和分支名，则会抓取所有分支

2.比较区别：

`git fetch`之后会抓取远程仓库的最新提交，但是不会改变当前工作区，可以使用`git diff`查看差别后再决定是否合并等下一步操作

命令形式：

命令	作用
`git diff`	比较 **工作区** 和 **暂存区**（你改了但还没 `add` 的）
`git diff --cached` 或 `git diff --staged`	比较 **暂存区** 和 **HEAD（上次提交）**
`git diff HEAD`	比较 **工作区** 和 **HEAD（上次提交）**（=全部改动）
`git diff 分支1 分支2`	比较两个分支的差异
`git diff origin/main`	比较本地分支与远程主分支的差异（前提是已 `git fetch`）

3.合并分支：

查看差别满意后合并到当前分支

命令：

```
git merge origin/main
```

6.3 拉取

从远程仓库抓取代码并合并到当前分支

命令形式：

```
git pull [远程仓库别名] [分支名]
```

- 等于fetch+merge

6.3 git fetch/git pull 的区别

操作	命令	会更新本地工作区吗?	会更新本地分支吗?	场景
抓取	`git fetch`	❌ 不会	✅ 更新远程分支引用（如 `origin/main`）	想先看看远程变化但不合并
拉取	`git pull`	✅ 会	✅ 更新远程分支引用，并**自动合并到当前分支**	想把远程代码同步到当前分支

7.远程解决冲突

7.1 远程仓库冲突

当你推送（`git push`）时，如果远程仓库有别人提交的新内容，而你本地没有更新这些内容，就会被 Git 拒绝推送，提示如下：

```
! [rejected]        main -> main (fetch first)
error: failed to push some refs
hint: Updates were rejected because the remote contains work that you do
not have locally.
```

为什么会有冲突?

Git 的原则是：**你不能把旧的东西覆盖掉别人的新提交。**你必须先“把远程最新的改动拿回来”，再整合你自己的代码。

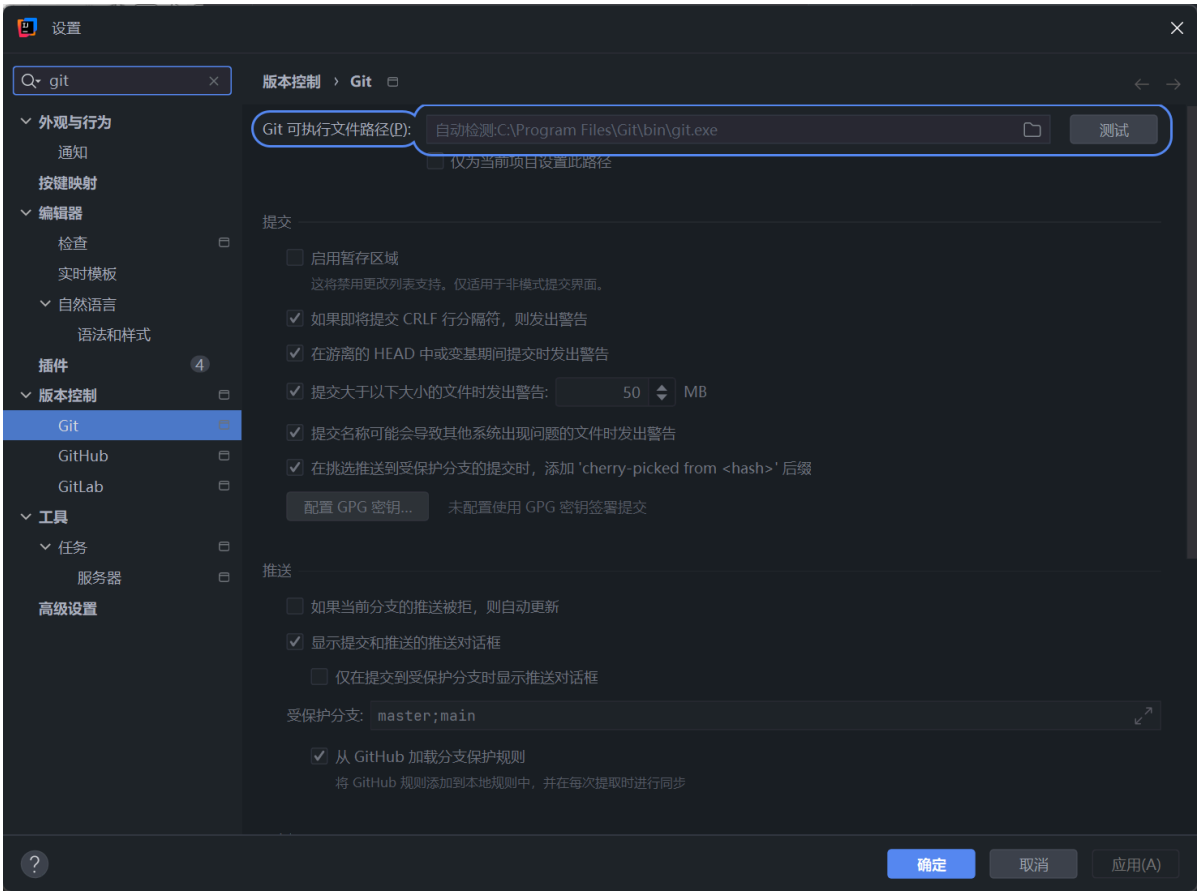
7.2 处理方法

1. 先拉取远程分支最新内容（如果没有冲突，它会自动合并，可继续推送）
2. 如果出现冲突，表示和别人**同时改了同一个文件的同一位置**
3. 这时手动打开冲突文件，手动解决冲突，和本地解决冲突一样，然后再推送

场景	会不会冲突	如何处理
多人改不同文件	❌ 不会冲突	自动合并
多人改相同位置	✅ 会冲突	`pull` → 手动解决 → `commit` → `push`
本地使用 `rebase`	⚠️ 有可能	可能需要 `--force`，协商使用
远程强推后再 `push`	✅ 会冲突	`rebase` 或重拉分支

8.idea配置git

8.1 设置git



8.2 idea操作远程仓库

1. 项目创建远程仓库
2. idea图形化初始化本地仓库

3. idea图形化实现git操作

9.常见多人协作开发形式-Git Feature分支模型

9.1 基本流程（推荐给中小型项目）

```
text复制编辑主分支：main（或 master）
每个人：从 main 拉自己的 feature 分支开发
开发完后：合并到 main，再推送远程
```

9.2 开发流程（每人）

1. 从主分支创建开发分支

```
git checkout main
git pull origin main
git checkout -b feature/xxx
```

2. 自己开发功能

```
...修改代码...
git add .
git commit -m "开发xxx功能"
```

3. 提交前拉取主分支防冲突

```
git fetch origin
git merge origin/main   # 或 git pull origin main
# 若有冲突，解决并提交
```

4. 合并回主分支

```
git checkout main
git merge feature/xxx
```

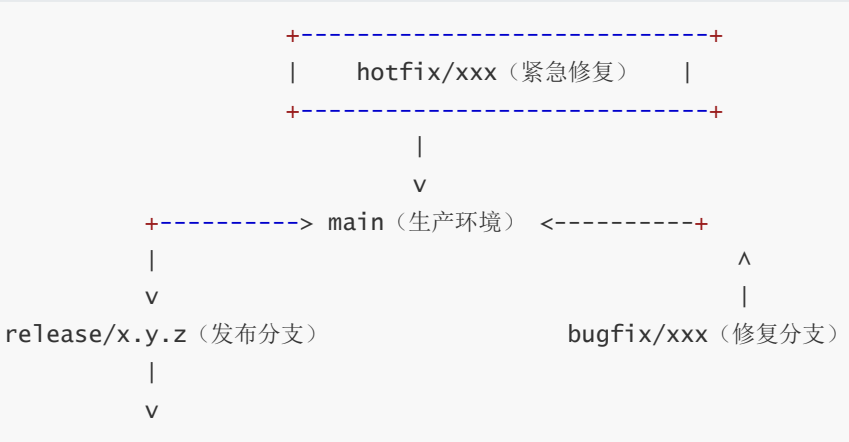
5. 推送主分支

```
git push origin main
```

9.3 常见协作策略对比

模式	简介	适用人群
🟢 GitHub Flow	只有主分支 + feature 分支 + Pull Request	个人项目、小团队
🔵 Git Flow	有主分支、开发分支、功能分支、发布分支	中大型团队，发布流程清晰
🔴 Trunk-based	所有人提交到主分支 + 自动测试部署	DevOps 团队、高级自动化场景

9.4 标准git分支结构



```
    develop（开发主分支）
      |   \
      |   feature/xxx（功能开发分支）
      |
      v
  test（测试分支）
```

开发中一般有如下分支使用的原则与流程：

- `master` **（生产）分支**

  线上分支、主分支，中小规模项目作为线上运行的应用对应的分支；

- `develop` **（开发）分支**

  从master创建的分支，一般作为开发部门的主要开发分支，如果没有其他并行开发不同期上线要求，都可以再次版本进行开发，阶段开发完成后，需要是合并到master分支，准备上线

- feature/xxxx分支

  从develop创建的分支，一般是同期并行开发，但不同期上线时创建的分支，分支上的研发任务完成后合并到develop分支

- hotfix/xxxx分支

  从master派生的分支，一般作为线上bug修复使用，修复完成后需要合并到master、test、develop分支

- 还有一些其他分支，如test分支（用于测试）、pre分支（预上线分支）等

各分支角色说明：

分支类型	命名规范	作用
`main` 或 `master`	`main`	生产部署用，**只放发布过的稳定代码**
`develop`	`develop`	开发主干，所有功能都从这里分出、合回来
功能分支	`feature/xxx`	新功能开发，来自 `develop`，合回 `develop`
修复分支	`fix/xxx` 或 `bugfix/xxx`	修 Bug，来自 `develop`，合回 `develop`
测试分支	`test/xxx`	用于集成测试或专项测试（可选）
发布分支	`release/1.0.0`	准备上线前生成，合并到 `main` 和 `develop`
热修复分支	`hotfix/xxx`	线上故障紧急修复，直接从 `main` 拉，修完合到 `main` 和 `develop`

9.5 各类分支使用流程

🌱 1. 开发新功能流程（feature）

```
bash复制编辑# 1. 从 develop 分支创建功能分支
git checkout develop
git pull origin develop
git checkout -b feature/login
```

```
# 2. 编码开发
...写代码...
git add .
git commit -m "feat: 完成登录功能"

# 3. 合并回 develop
git checkout develop
git pull origin develop
git merge feature/login
git push origin develop
```

🐞 2. 修复 Bug（fix/bugfix）

```
bash复制编辑# 从 develop 创建修复分支
git checkout develop
git checkout -b fix/login-bug

# 修改代码
git add .
git commit -m "fix: 修复登录页跳转问题"

# 合并回 develop
git checkout develop
git merge fix/login-bug
git push origin develop
```

🧪 3. 测试流程（test）

```
bash复制编辑# （可选）从 develop 创建测试分支
git checkout develop
git checkout -b test/integration
git push origin test/integration

# 提交给测试人员，测试后如有问题，再 fix/bugfix 修复
```

🚀 4. 发布版本（release）

```
bash复制编辑# 从 develop 创建发布分支
git checkout develop
git checkout -b release/1.0.0

# 执行打包、预发布、改版本号等工作
git add .
git commit -m "chore: release 1.0.0"

# 合并回 main 和 develop
git checkout main
git merge release/1.0.0
git push origin main

git checkout develop
git merge release/1.0.0
git push origin develop
```

```
# 删除发布分支
git branch -d release/1.0.0
```

🔥 5. 热修复（hotfix）

```
bash复制编辑# 线上出 bug，从 main 分支拉热修复分支
git checkout main
git pull origin main
git checkout -b hotfix/fix-crash

# 修复 bug 并提交
git add .
git commit -m "hotfix: 修复崩溃问题"

# 合并回 main（马上部署）
git checkout main
git merge hotfix/fix-crash
git push origin main

# 合并回 develop（保持一致）
git checkout develop
git merge hotfix/fix-crash
git push origin develop
```

分支类型	来源	合并目标	是否保留	用于什么
`feature/*`	`develop`	`develop`	合并后可删	每个功能一个分支
`fix/*`	`develop`	`develop`	合并后可删	非线上 bug 修复
`release/*`	`develop`	`main` 和 `develop`	合并后删	版本发布前准备
`hotfix/*`	`main`	`main` 和 `develop`	合并后删	线上紧急修复
`test/*`	`develop`	自定义	可选	集成/专项测试用