代码托管

最佳实践

文档版本	06
发布日期	2020-02-25

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 Git on CodeHub.....1
1.1 **概述**..... 1
1.2 **CodeHub 云端操作**..... 4
1.3 **Git 本地研发场景**..... 7
1.4 **Git 工作流**..... 12
1.4.1 **Git 工作流概述**..... 12
1.4.2 **集中式工作流**..... 12
1.4.3 **功能分支工作流**..... 13
1.4.4 **Gitflow 工作流**..... 14
1.4.5 **Forking 工作流**..... 15

1 Git on CodeHub

1.1 概述

文档目的

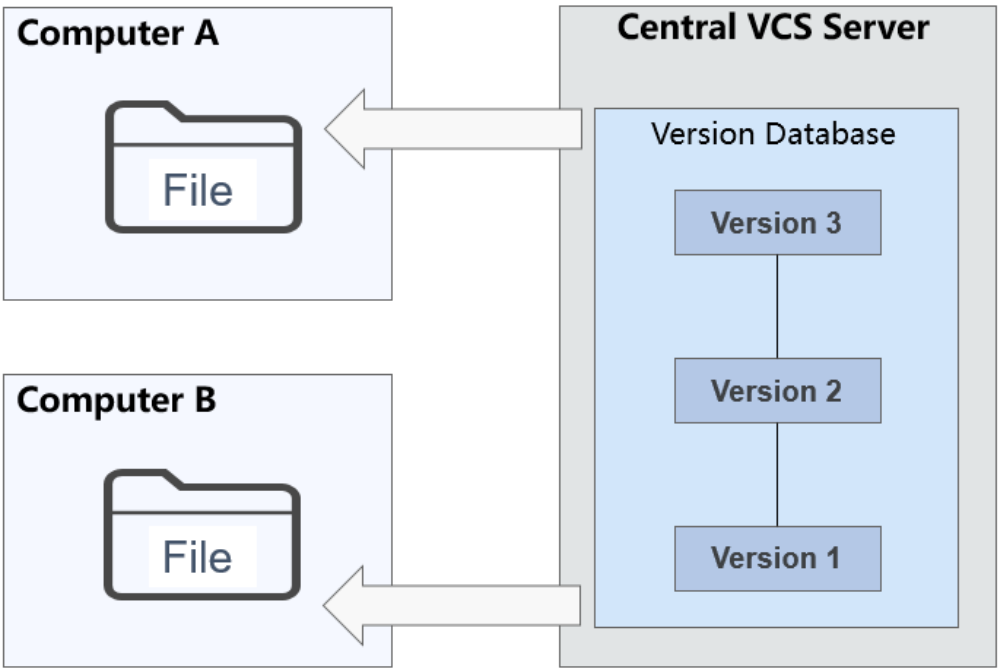
基于DevCloud实践所编写，用于帮助已经掌握或想要掌握Git的开发者，更好的应用Git，以及更好的将Git与DevCloud结合应用。

Git 概述

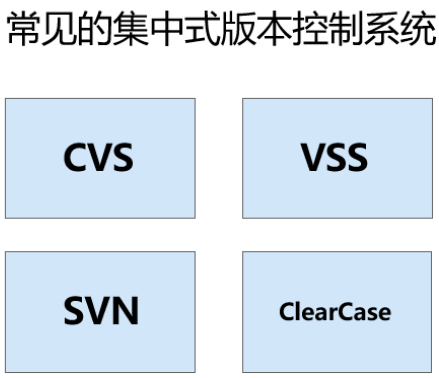
从狭义上来说，版本控制系统是软件项目开发过程中管理代码所有修订版本的软件，能够存储、追踪文件的修改历史，记录多个版本的开发和维护，事实上我们可以将任何对项目有帮助的文档交付版本控制系统进行管理。版本控制系统（Version Control Systems）主要分为两类，集中式和分布式。

集中式版本控制系统

集中式版本控制系统的特点是只有一台中央服务器，存放着所有研发数据，而其它客户端机器上保存的是中央服务器最新版本的文件快照，不包括项目文件的变更历史。所以，每个相关人员工作开始前，都需要从这台中央服务器同步最新版本，才能开始工作，如下图所示。



常见的集中式版本控制系统如下图所示。



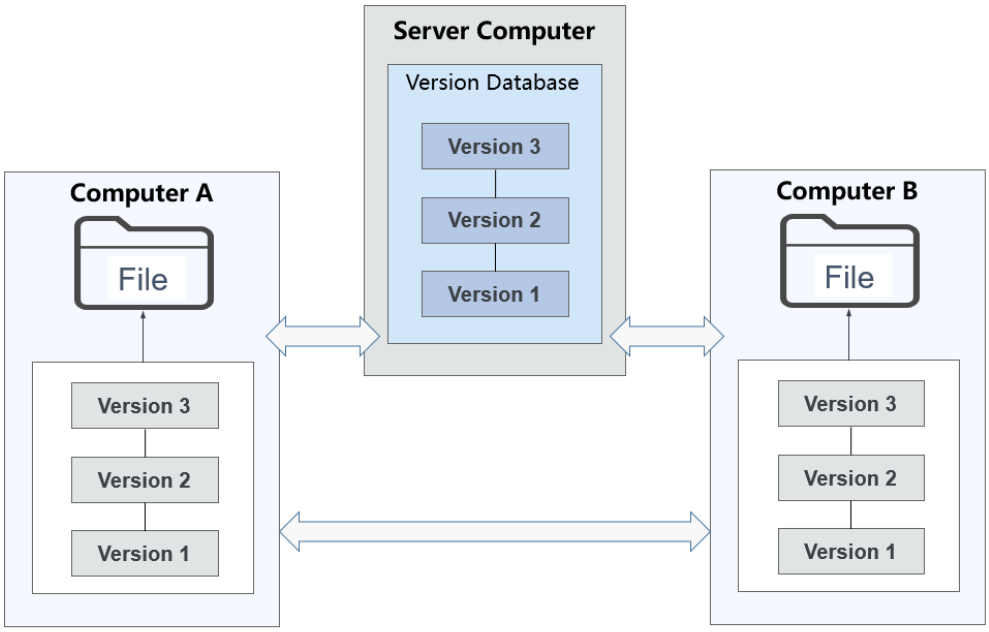
集中式版本控制系统的优点与缺点如下表所示。

表 1-1 集中式版本控制系统描述

优点	缺点
• 操作简单，使用没有难度，可轻松上手。 • 文件夹级权限控制，权限控制粒度小。 • 对客户端配置要求不高，无需存储全套代码。	• 网络环境要求高，相关人员必须联网才能工作。 • 中央服务器的单点故障影响全局，如果服务器宕机，所有人都无法工作。 • 中央服务器在没有备份的情况下，磁盘一旦被损坏，将丢失所有数据。

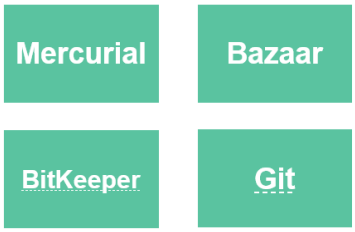
分布式版本控制系统

分布式版本控制系统的特点是每个客户端都是代码仓库的完整镜像，包括项目文件的变更历史。所有数据分布的存储在每个客户端，不存在中央服务器。可能有人会问，我们公司使用Git分布式存储工具，也有“中央服务器”啊？其实，这个所谓的“中央服务器”仅仅是用来方便管理多人协作，任何一台客户端都可以胜任它的工作，它和所有客户端没有本质区别，如下图所示。



常见的分布式版本控制系统如下图所示。

常见的分布式版本控制系统



分布式版本控制系统的优点与缺点如下表所示。

表 1-2 分布式版本控制系统描述

优点	缺点
• 版本库本地化，版本库的完整克隆，包括标签、分支、版本记录等。 • 支持离线提交，适合跨地域协同开发。 • 分支切换快速高效，创建和销毁分支廉价。	• 学习成本高，不容易上手。 • 只能针对整个仓库创建分支，无法根据目录建立层次性的分支。

1.2 CodeHub 云端操作

准备工作

成为代码托管（CodeHub）用户

Git客户端

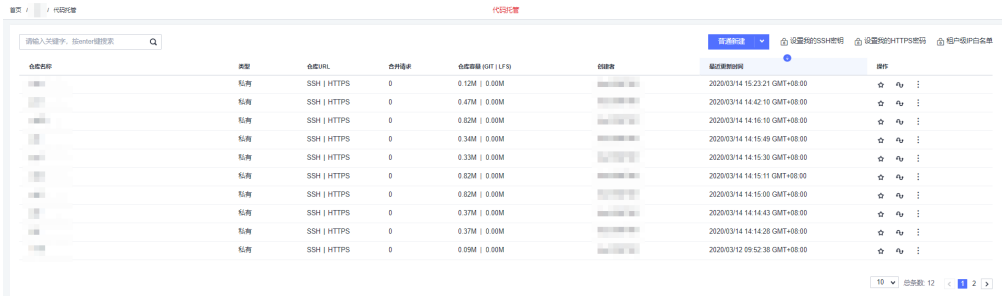
已创建好的项目

云端仓库功能

云端仓库功能支持新建仓库、仓库克隆、分支管理、标签管理、提交代码、拉取代码、推送代码、代码阅读、在线修改、仓库成员管理、密钥管理等，更多仓库功能介绍请参见**代码托管产品介绍** 。

新建空仓库

1. 在DevCloud代码托管服务中，单击上方“普通新建”按钮，如下图所示。



2. 配置新仓库的详细信息，下图所示。

3. 保存仓库信息后，新建成功的仓库如下图所示。

设置 SSH 密钥/HTTPS 密码

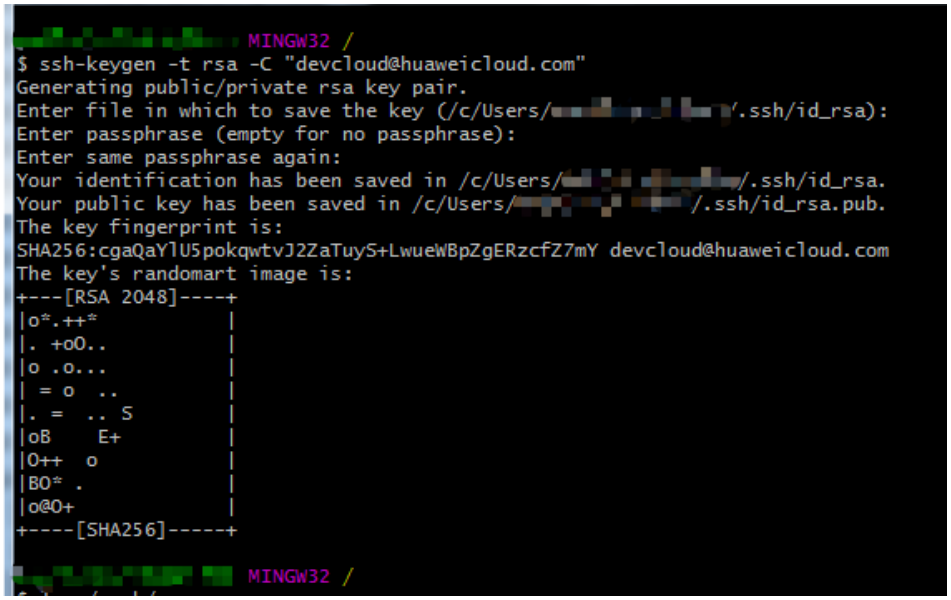
后续需要在本地客户端进行代码仓库的克隆/推送，SSH密钥和HTTPS密码是客户端和服务端交互的凭证，需要先对它们进行设置。

设置SSH密钥

SSH密钥是使用SSH协议和代码托管服务端交互的凭证，如果您使用windows下的Git Bash客户端并在其中已经生成，此步骤可以略过。

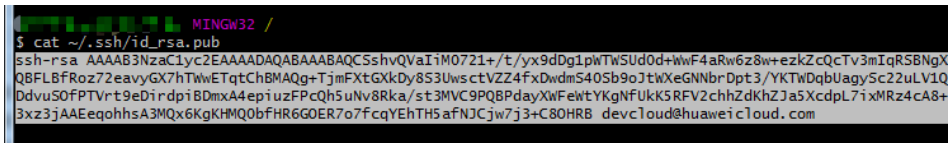
1. 打开Git客户端（Git Bash或linux的命令行窗口），输入以下命令行：
   ```
   ssh-keygen -t rsa -C "<您的邮箱>"
   ```


然后输入3个回车（Enter键）即可，生成的SSH秘钥对默认在“~/.ssh/id_rsa、~/.ssh/id_rsa.pub”位置，如下图所示。

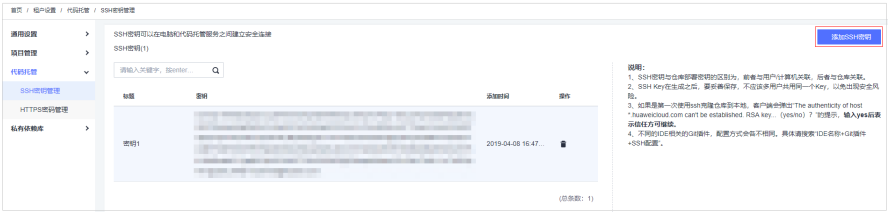


2. 添加SSH密钥到代码托管服务端：
   a. 打开Git客户端（Git Bash或linux的命令行窗口），将SSH密钥“~/.ssh/id_rsa.pub”的内容打印出来，如下图所示。

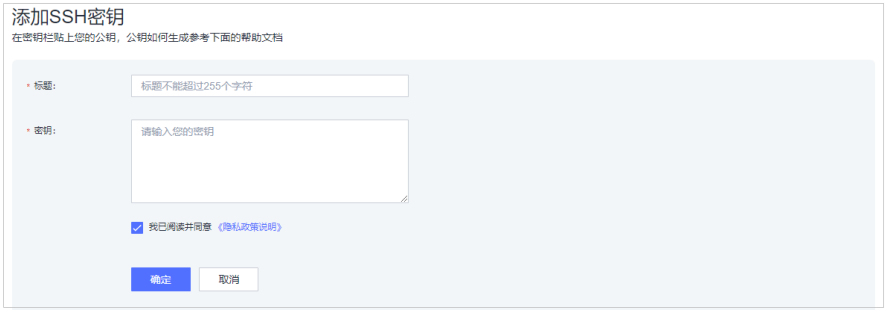
```
cat ~/.ssh/id_rsa.pub
```



   b. 复制上述的SSH密钥内容，去到代码托管首页，单击“设置我的SSH密钥”按钮去到SSH密钥管理页面，单击“添加SSH密钥”进行添加。
      i. 进入SSH密钥管理页面，如下图所示。



      ii. 单击“添加SSH密钥”按钮进行添加，粘贴上述复制的SSH密钥内容、填写标题，单击“确定”即可，如下图所示。



您已经设置好了SSH密钥，您可以继续设置HTTPS密码。

设置HTTPS密码

HTTPS密码是使用HTTPS协议和代码托管服务端交互的凭证，设置步骤如下：

1. 进入代码托管首页，单击“设置我的HTTPS密钥”，显示“HTTPS密钥管理”页面。
   a. 如果您是第一次进行设置，则输入2次HTTPS密码保存即可。
   b. 如果您不是第一次进行设置，单击“修改”，需要输入邮箱验证码，重新设置新密码“保存”即可。

1.3 Git 本地研发场景

背景介绍

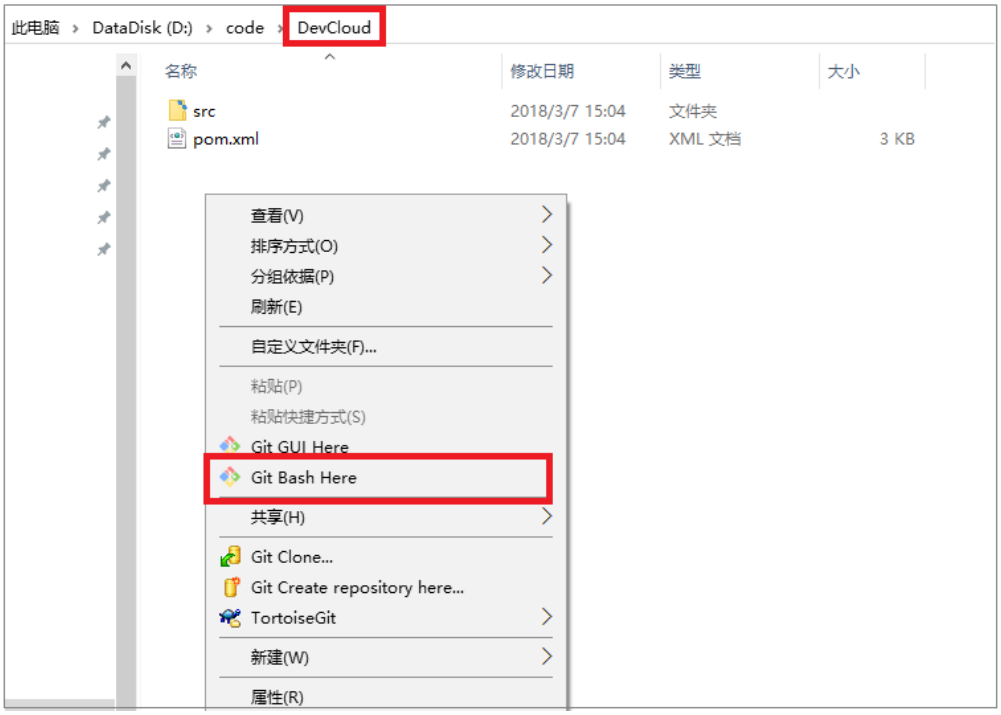
在DevCloud云端创建一个README文件的空仓库，然后架构师或者项目负责人需要把本地框架代码推送到这个空仓库，最后，其他开发人员将云端架构代码克隆到本地，进行增量应用开发。

说明

- Git代码传输支持SSH和HTTPS两种传输协议，本节基于SSH传输协议进行的操作。
- 如果想使用HTTPS方式，直接下载HTTPS密码，当克隆、推送代码时直接输入HTTPS用户名密码即可。
- 同一仓库SSH和HTTPS的地址不同。

推送架构代码

1. 打开本地框架代码，确保根目录名（DevCloud）与云端创建的代码仓库名一致，在根目录下右键打开**Git bash**终端，如下图所示。



2. 推送本地代码到云端。

   在当前**Git Bash**终端依次输入如下命令：

   a. 初始化本地代码仓库，执行该命令后，在“D:/code/DevCloud/”下多了一个“.git”文件夹。

   ```
   $ git init
   ```

   b. 关联云端代码仓库。

   ```
   $ git remote add origin CodeHubUrl
   ```

   仓库地址“*CodeHubUrl*”如下图所示单击红色方框处获取。

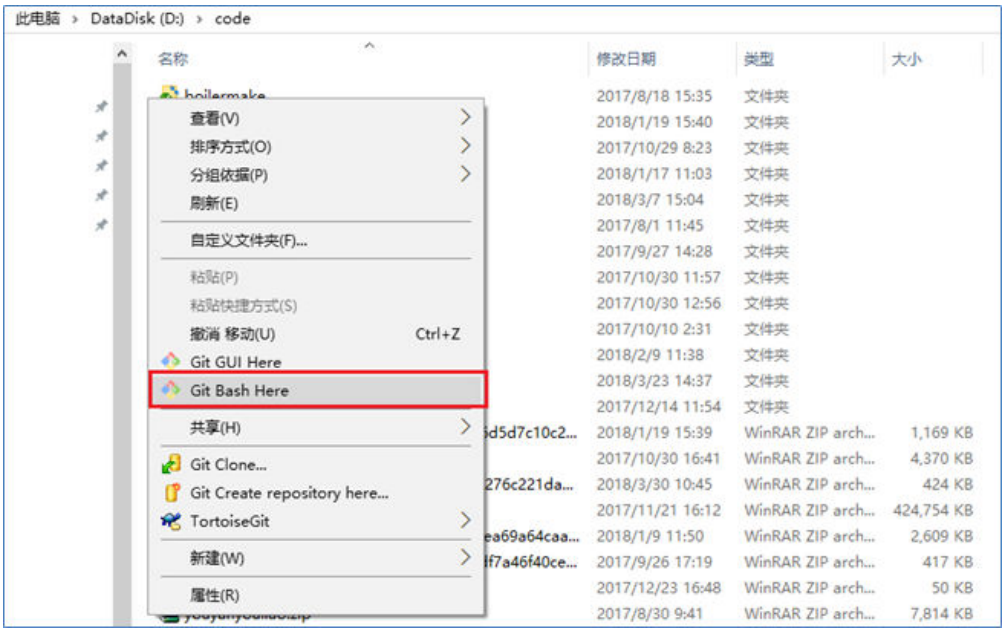
   c. 推送代码到云仓库。

   ```
   $ git  add .
   $ git  commit  -m "init project"
   $ git  branch --set-upstream-to=origin/master master
   $ git  pull  --rebase
   $ git  push
   ```

克隆代码

开发人员在本地准备克隆云端架构代码。

1. 在准备把代码克隆到的目标文件夹下，右键打开**Git bash**终端，如下图所示。



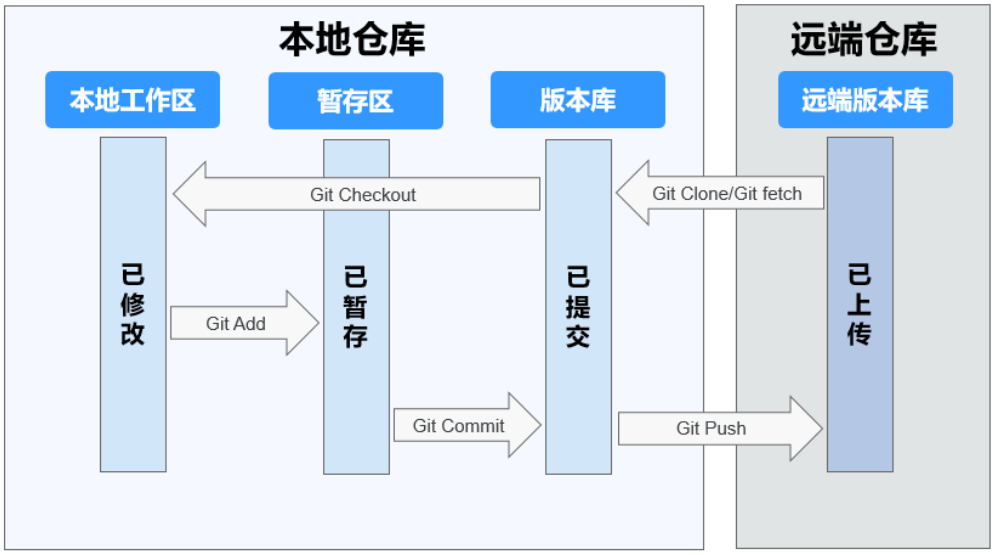
2. 克隆仓库，URL地址如下图所示单击红色方框处获取。

```
$ git clone CodeHubUrl  //将代码从远端仓库clone到本地
```

代码提交

一次修改被成功提交到远端仓库会历经四个阶段：“1本地工作区 > 2缓存区 > 3版本库 > 4远端版本库”。

通过执行相应的Git命令，文件在这四个区域跳转，并呈现不同的状态，如下图所示。



主要涉及如下三步操作：

1. **#git add/rm filename** *//将新增、修改或者删除的文件增加到暂存区*
2. **#git commit –m “commit message”** *//将已暂存的文件提交到本地仓库*
3. **#git push** *//将本地代码仓库修改推送到远端仓库*

分支操作

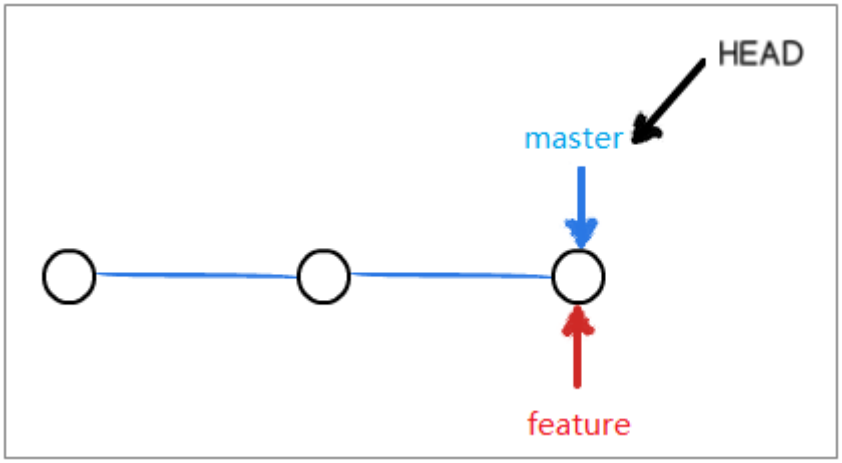
- 新建分支

  Git新建分支的本质就是创建一个指向最后一次提交的可变指针，所以，Git分支的创建不是复制版本库的内容，仅仅是新建了一个指针，它以40个字符长度SHA-1字串形式保存在文件中。

  #git branch *branchName commitID*

  基于commitID即某一个全球版本号拉出新分支，如果没有commitID则基于当前分支的HEAD拉出新分支。

  例如，新建feature分支，执行的命令为**git branch feature**，如下图所示。

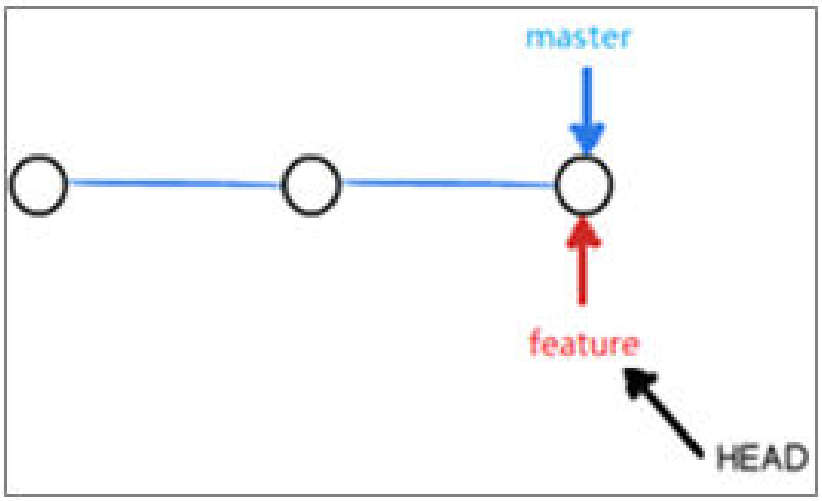
  

- 切换分支

  命令如下。

  #git checkout *branchName*

  例如，切换到feature分支，执行的命令为**git checkout feature**，如下图所示。

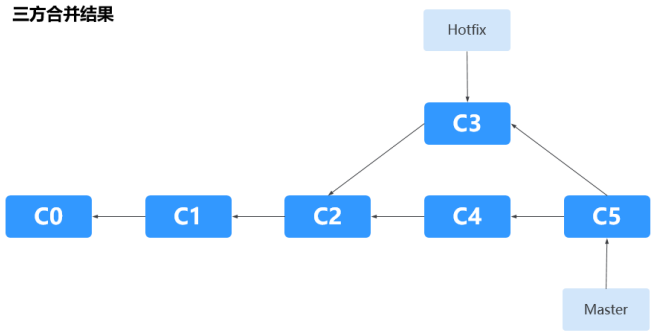
- 分支合并

  无论哪种工作流都会涉及到分支合并（把一个分支中的修改整合到当前分支），主要有两种方法：三方合并（merge）和衍合（rebase）。通过对同一种场景进行不同操作体会两种合并方法的区别。

  场景：master分支新增了C4节点， hotfix分支新增了C3节点，现将hotfix分支合并到master分支：

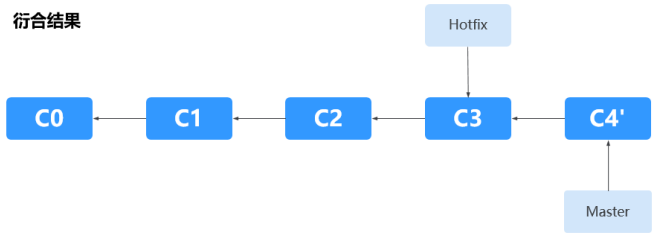
  a. 三方包括hotfix新增节点C3，master新增节点C4，以及两者的共同祖先节点C2。这种合并操作简单，但新增合并节点C5，形成了环形，版本记录可读性差，如下图所示。

  ```
  #git checkout master
  #git merge hotfix
  ```

  b. 衍合先将master分支新增节点C4以补丁形式保存在.git/rebase目录中，然后同步hotfix分支最新代码，再应用补丁C4’，如下图所示。

  ```
  #git checkout master
  #git rebase hotfix
  ```

- 冲突解决

  a. 场景一：两个合并分支修改了同一行代码

  ```
  $ cat doc/README.txt
  User1 hacked.
  <<<<<<< HEAD
  Hello, user2. #当前分支修改方法
  =======
  Hello, user1. #源分支修改方法
  >>>>>>> a123390b8936882bd53033a582ab540850b6b5fb
  User2 hacked.
  User2 hacked again.
  ```


解决方法：

i. 分析哪种修改方法正确，手动合并。

ii. 提交修改。

b. 场景二：文件被重命名为不同的名字

解决方法：

i. 确认哪个名字是正确的，删除错误的。

ii. 提交修改。

1.4 Git 工作流

1.4.1 Git 工作流概述

什么是Git工作流？你可以理解为代码管理的分支策略，它不仅仅是版本管理范畴，更服务于项目流程管理和团队协同开发。所以，有必要制定适合自己研发场景的工作流。

下面介绍四种工作流的工作方式、优缺点，以及使用中的一些注意事项。

- 集中式工作流
- 功能分支工作流
- Gitflow工作流（Devcloud推荐）
- Forking工作流

研发团队可以根据实际研发场景制定合理的工作流，能有效提高项目管理水平和团队协同开发能力， 并通过CodeHub平台，高效、安全的管理代码资产，将更多的精力集中在业务开发上，实现持续集成、持续交付和快速迭代的目标。

1.4.2 集中式工作流

集中式工作流适合5人左右小开发团队，或是刚从SVN工具转型为Git的团队，它只有一个默认的maste分支（相当于svn的trunk主分支），所有人的修改都是在master分支上进行的。但是，这种工作流无法充分发挥git优势和多人协同，不推荐使用。

工作方式

开发人员将master分支从中央仓库克隆到本地，修改完成后再推送回中央仓库master分支。

优点

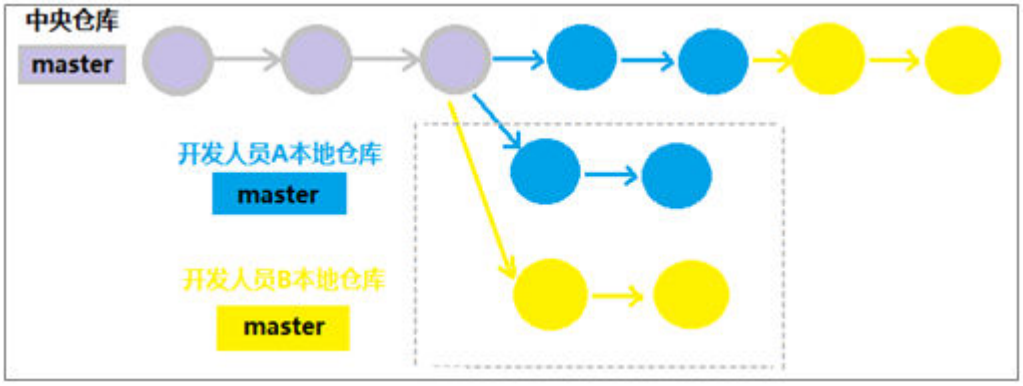
不涉及分支交互操作。

缺点

- 不适合人员较多的团队，当人员10+时，解决开发人员之间的代码冲突会耗费很多时间。
- master分支提交频繁。
- master分支不稳定，不利于集成测试。

Tips：如何尽量避免产生冲突和不合理的提交历史？

开发人员在开发一个新功能之前，一定要在本地同步中央仓库最新代码，使自己的工作基于最新的代码之上；开发完成后，在提交新功能到中央仓库前，需要先fetch中央库的新增提交，并rebase自己的提交。这样做的目的是，把自己的修改加到中央仓别人已经提交的修改之上，使最终的提交记录是一个完美的线性历史，而不是环形，工作流举例如下图所示。



1. 开发人员A和开发人员B同时在某个时间拉取了中央仓库的代码。
2. 开发人员A先完成了自己的工作，并提交到中央仓库。
3. 开发人员B需要在本地执行**git pull –rebase**中央仓库的新提交，这时开发人员B的本地仓库就包含了开发人员A修改的内容，并在A的基础上增加了自己的修改。
4. 开发人员B将代码推送到中央仓库。

1.4.3 功能分支工作流

通过新建几个功能分支，增加开发者的交流和协作，它的理念是所有的功能开发都应该在master分支外的一个独立分支进行，这种方式隔离了开发者的工作空间不被互相干扰，保证了master分支的稳定性。

工作方式

开发人员每次在开始新功能开发前，需要在master分支上拉取一个新分支，并起个有描述性的名字，比如**video-output**或**issue-#1061**，这样可以让分支用途明确。功能分支不但存在开发人员本地仓库，也应该推送到中央仓库，这样就可以在代码不合入master分支的情况下与其他开发人员分享代码。

优点

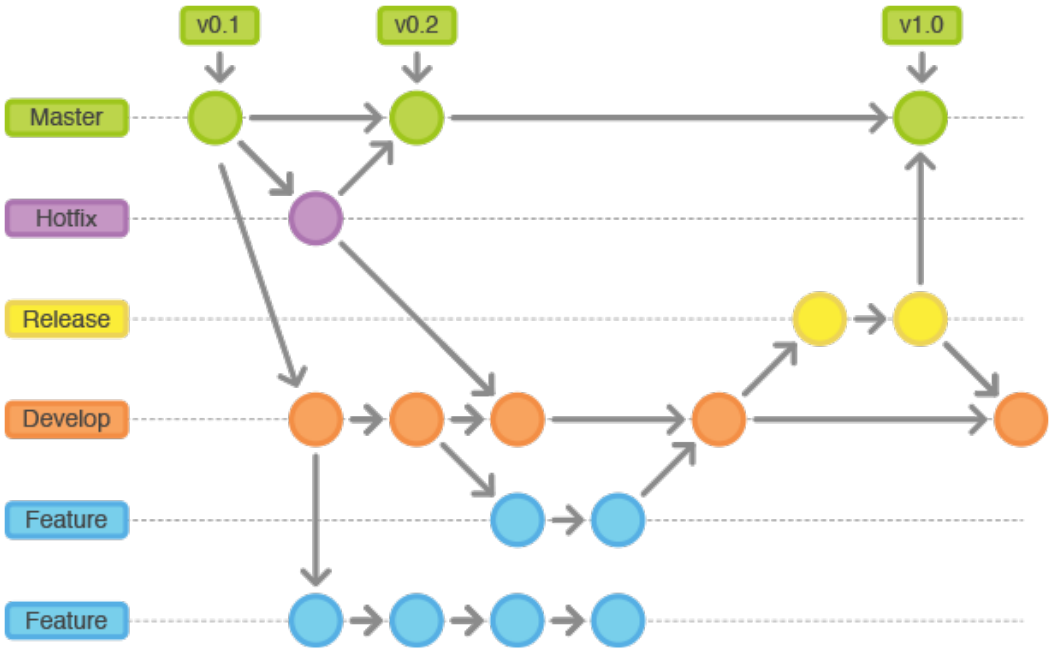
- 分支合并前可以使用**pull request**进行**code review**。
- 降低了master分支的提交频率。

缺点

只有一个master分支作为集成，仍然不是很稳定，不适合大型开发。

1.4.4 Gitflow 工作流

Gitflow一般用于管理大型项目，它为不同的分支分配一个很明确的工作角色，并定义分支之间什么时候进行交互，如Gitflow工作流如下图所示。



工作方式

- **master分支：**

  生产分支，最稳定的版本，一直是ready to deploy状态。不接受开发人员直接commit，只接受从其他分支merge操作。在很多企业中，这个分支被默认开启分支保护，只有维护者可以操作。

- **hotfix分支：**

  从master分支拉取的临时修复分支，用于解决一线紧急bug。bug解决后需要合入master分支并打上新的版本号，这个修改也需要同时合入develop分支。

- **develop分支：**

  从master分支拉取的开发分支，用于功能集成。包含所有要发布到下一个Release的代码用于开发集成、系统测试。

- **release分支：**

  临近既定的发布日，就从develop分支上拉取一个release分支，任何不在当前分支中的新功能都推到下个发布中。release分支用于发布，所以从当前时间点之后新的功能不能再加到这个分支上，这个分支只做Bug修复、文档生成和其它面向发布的任务。当对外发布的工作都完成了，release分支合并到master分支并分配一个版本号打好Tag；另外，这些从release分支新做的修改要反向合并回develop分支。

- **feature分支：**

  开发者使用的特性分支，父分支是develop分支，当新功能完成时，合入develop分支。新功能提交从不直接与master分支交互。

开发人员提交新功能的两种途径：

- 团队有专人review审核新功能
  a. 开发人员将feature分支推送到代码托管平台（中央仓库）。
  b. 发起一个从feature分支合并到develop分支的合并请求，并指给review专员。

     说明

     DevCloud代码托管中支持“合并请求”功能，直接选择源和目的分支，仓库管理员（项目经理、创建仓库的开发人员、被给予仓库管理权限的开发人员）有权限接受此合并请求。

  c. review专员审核。如果通过，将feature分支的新功能合并到develop分支，并删除feature分支；如果未通过，拒绝该请求并注明拒绝原因。

- 开发人员自审核新功能
  a. 开发人员在本地仓库将feature分支合并到develop分支，并删除feature分支。
  b. 将本地develop分支的修改推送到软件开发云代码托管平台（中央仓库）。

优点

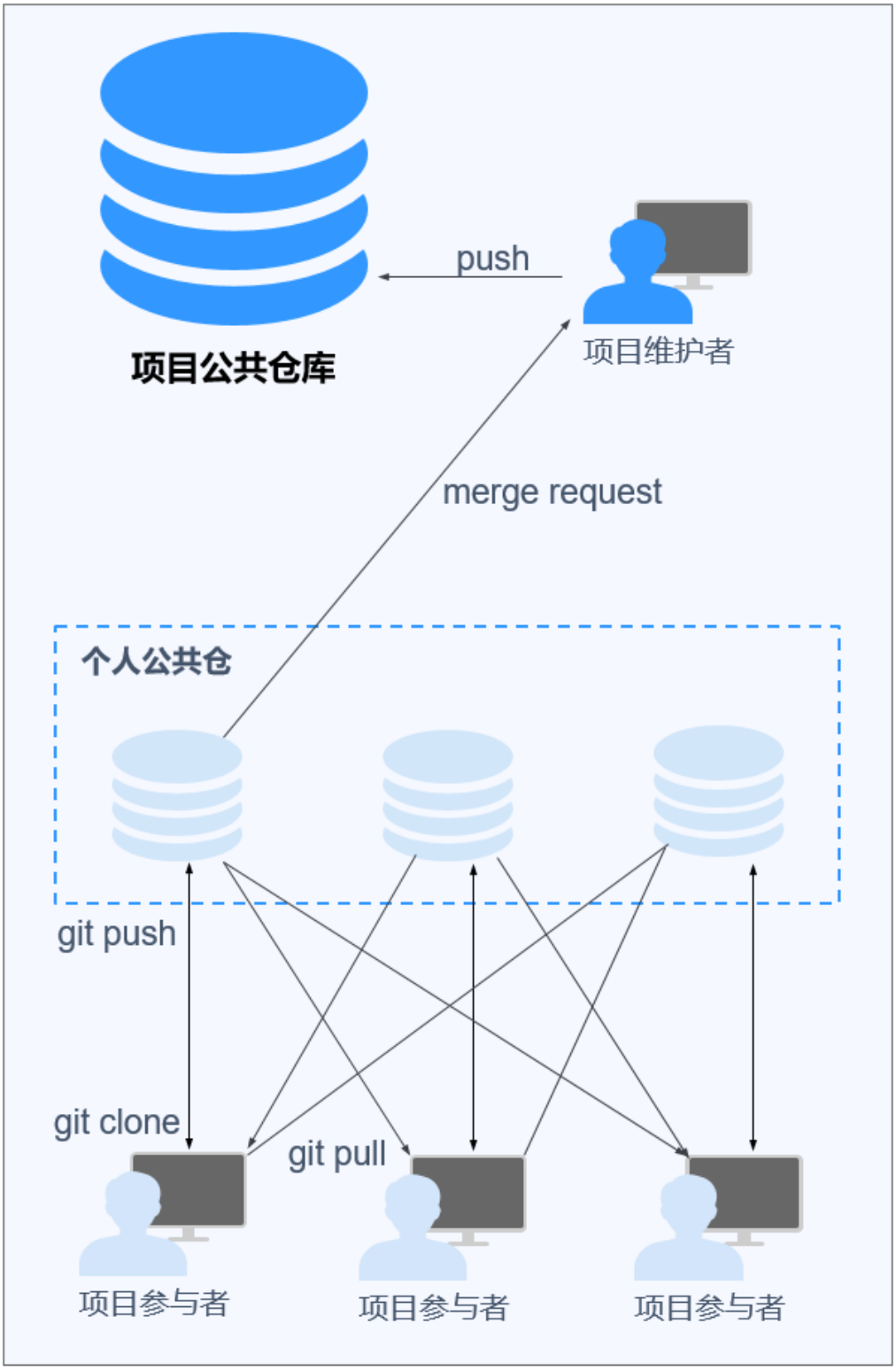
- 使用一个用于发布准备的专门分支（release分支），使得一个团队可以在完善当前的发布版本的同时，可以在develop分支并行继续开发下个版本的功能。这也打造了可视化的发布阶段，团队成员都可以在仓库网状结构中可以看到发布状态。
- 使用紧急修复分支（hotfix分支）让团队可以处理紧急问题的同时而不打断其它工作或是等待下一个发布再合入hotfix修改。我们可以把hotfix分支想成是一个直接在master分支上处理的临时发布。
- 大型项目人员协作频繁，流程较多，合理的多角色分支帮助研发有条不紊进行。
- 更符合devops理念。

缺点

- 学习成本较高。
- 如果团队不遵守使用约定，带来的影响更大。

1.4.5 Forking 工作流

Forking工作流区别于前三种工作流的最大特点是每个开发人员都有一个从公共仓库fork出来的属于自己的公共仓。Forking工作流适合外包、众包以及众创和开源场景。接包方的开发人员从项目公共仓fork自己的公共仓库进行操作，并不需要被项目公共仓直接授权，Forking工作流如下图所示。



工作方式

1. 将“项目公共仓”fork出一个“个人公共仓”。

2. 将“个人公共仓”clone到“本地仓库”。
3. 操作“本地仓库”，修改完成后提交到“个人公共仓”。
4. 为“个人公共仓”提交一个pull request给项目维护者，申请代码合入“项目公共仓”。
5. 项目维护者在本地review、验证本地提交，审核通过后push进入“项目公共仓”。

说明

如果开发人员A的代码未被审核通过合入“公共仓库”，而此代码对开发人员B有借鉴作用，开发人员B可以直接从开发人员A的“个人公共仓”拉取代码。

优点

- 开发人员之间若需要代码协作，可以直接从其他人的“个人公共仓”拉取，无需等到代码提交到项目公共仓。
- “项目公共仓”无需为每个代码贡献者授权。
- 项目维护者通过审核pull request成为代码安全的重要防线。
- 仓库分支的选择可以根据项目实际情况综合使用前三种工作流。

缺点

提交开发人员代码到最终版本库的周期较长，步骤繁琐。

