代码托管

最佳实践

文档版本	01
发布日期	2024-06-18

华为云计算技术有限公司

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云计算技术有限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，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

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。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 Git On CodeArts Repo

1.1 概述

文档目的

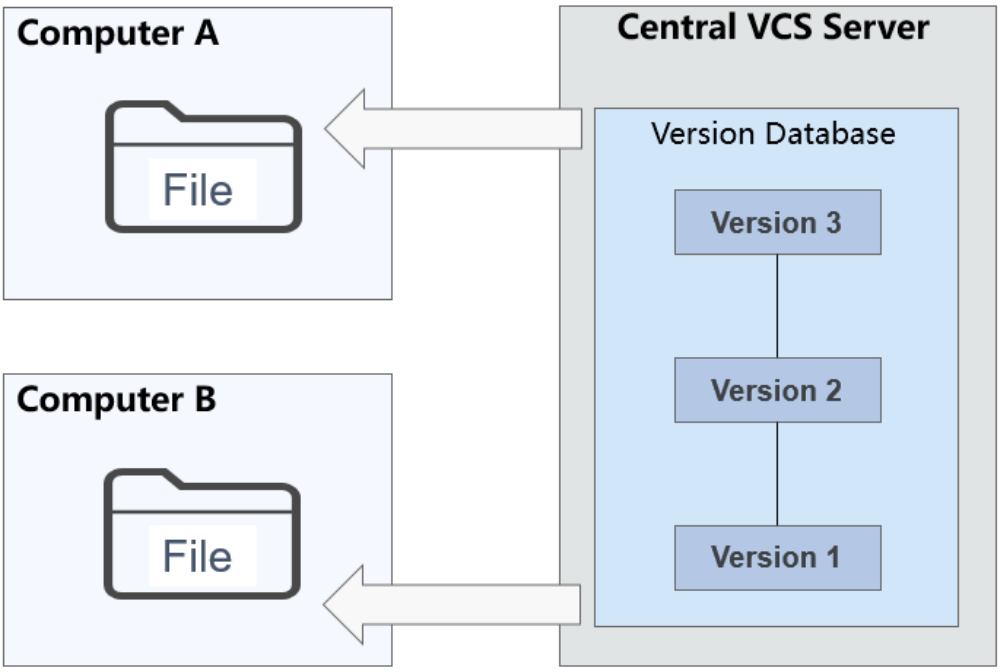
基于CodeArts实践所编写，用于帮助已经掌握或想要掌握Git的开发者，更好的应用Git，以及更好的将Git与CodeArts结合应用。

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，事实上您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为CVS、VSS、SVN、ClearCase。

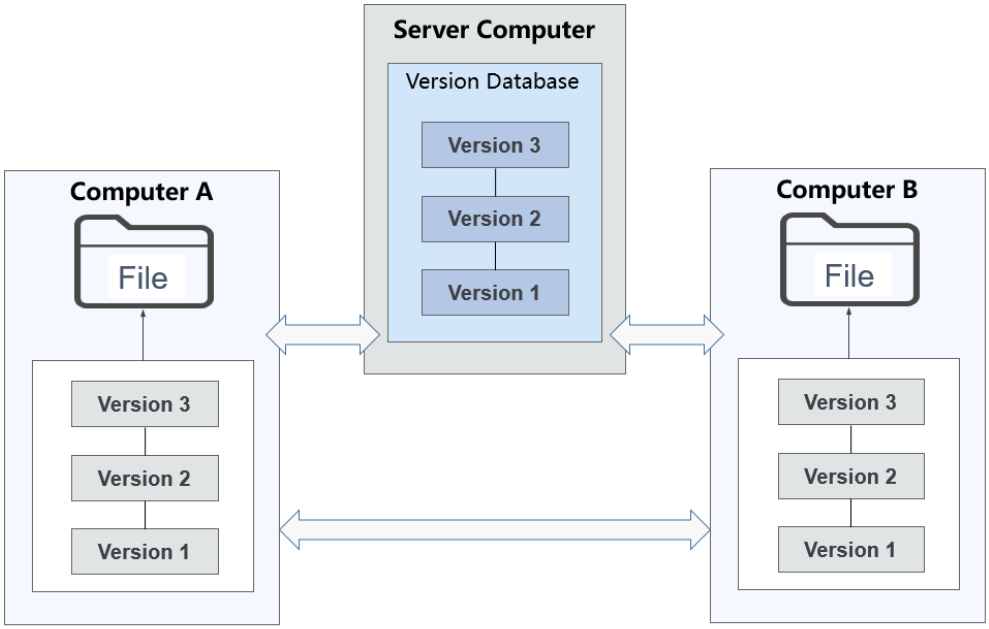
集中式版本控制系统的优点与缺点如下表所示。

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，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为Git、Mercurial、Bazaar、Bitkeeper。

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，创建和删除分支成本低。	• 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 CodeArts Repo 云端操作

准备工作

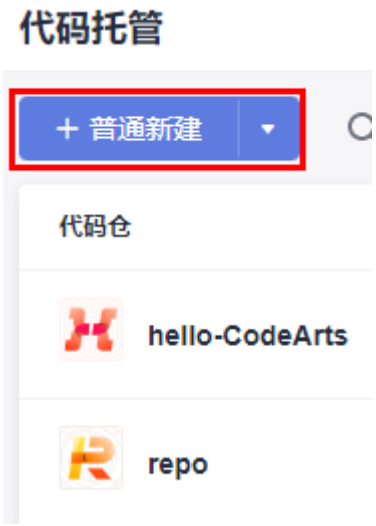
- 成为代码托管（CodeArts Repo）用户
- **已有Git客户端**
- **已创建好的项目**

云端仓库功能

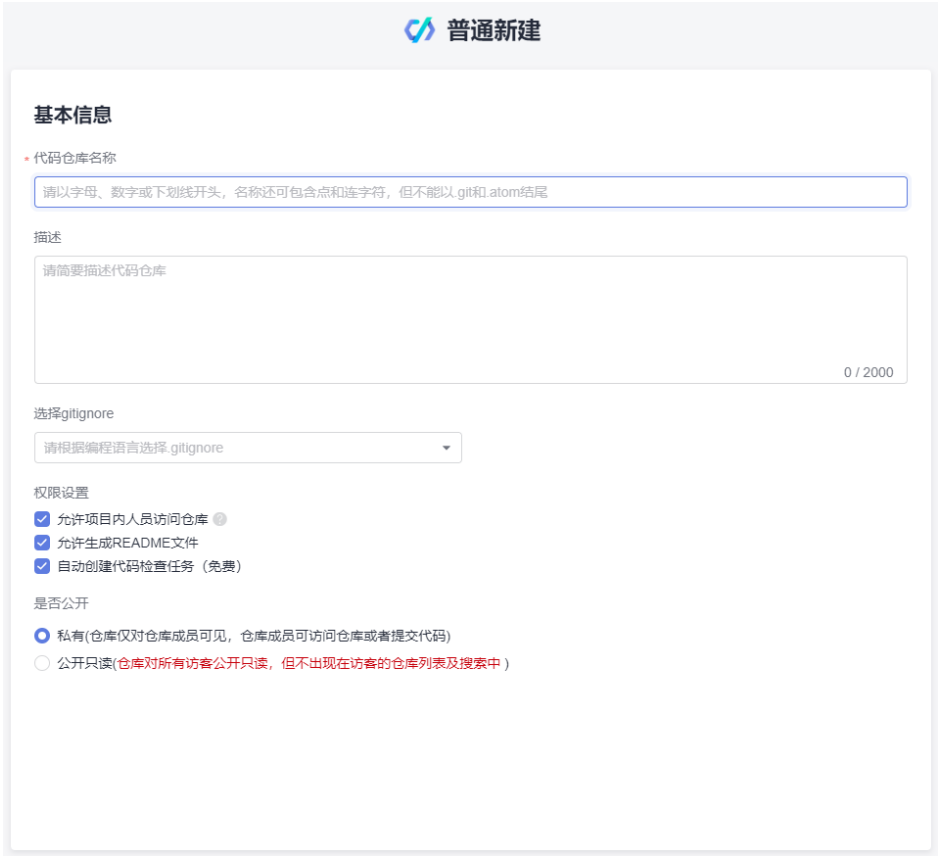
云端仓库功能支持新建仓库、仓库克隆、分支管理、标签管理、提交代码、拉取代码、推送代码、代码阅读、在线修改、仓库成员管理、密钥管理等，更多仓库功能介绍请参见**产品概述**。

新建空仓库

1. 在目标项目下的代码托管服务中，单击“普通新建”按钮，如下图所示。



2. 填写仓库的基本信息，下图所示。



3. 单击“确定”按钮，完成仓库新建，跳转到仓库列表。

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。

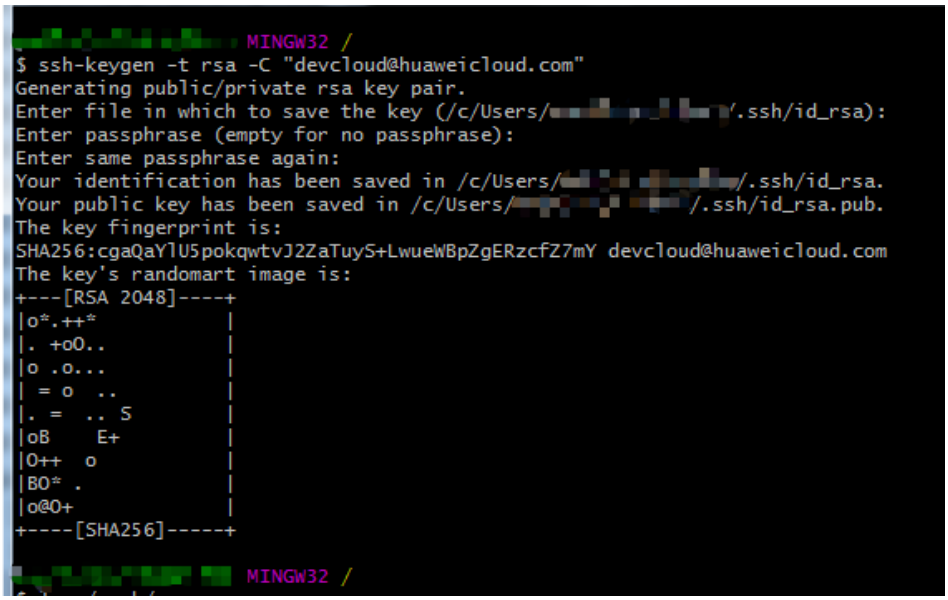
设置SSH密钥

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。

步骤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密钥对默认在“~/.ssh/id_rsa、~/.ssh/id_rsa.pub”位置，如下图所示。

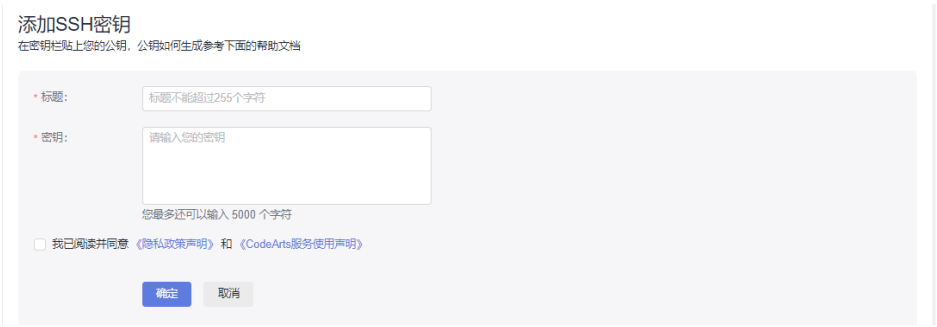


步骤2 添加SSH密钥到代码托管服务端：

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。

步骤3 复制上述的SSH密钥内容，登录您的代码托管服务仓库列表页，单击右上角昵称，单击“个人设置 > SSH密钥管理”，进入页面。

步骤4 在“SSH密钥管理”页面，单击“添加SSH密钥”，弹出“添加SSH密钥”页面，填写下图中信息，单击**“确定”**，页面会提示您操作成功。



您已经设置好了SSH密钥，您可以继续设置HTTPS密码。

----结束

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：

步骤1 登录您的代码托管服务仓库列表页，单击右上角昵称，单击“个人设置 > HTTPS密钥管理”，进入页面。

步骤2 单击“自行设置密码”，再单击“修改”进入“重设密码”页面。（如果您之前自主设置过HTTPS密码并正在使用，直接单击“修改”）

步骤3 填写新密码与邮箱验证码，勾选**“我已阅读并同意《隐私政策声明》和《CodeArts服务使用声明》”**，单击**“保存”**，页面会提示您操作成功。

----结束

1.3 Git 本地研发场景

背景介绍

在CodeArts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

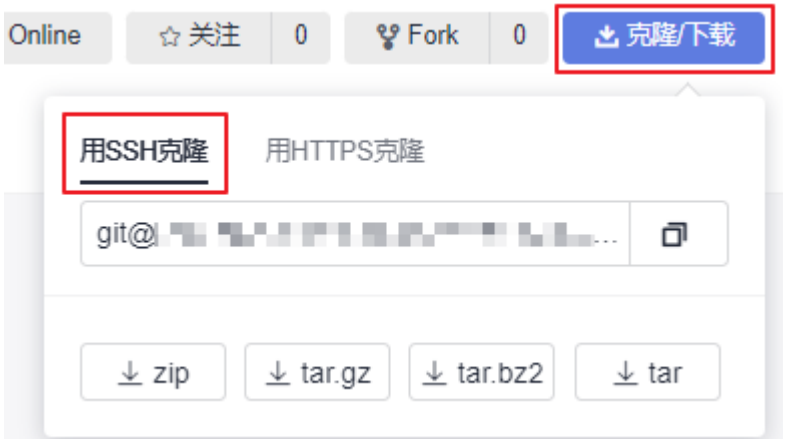
- Git代码传输支持SSH和HTTPS两种传输协议，本节基于SSH传输协议进行的操作。
- 如果想使用HTTPS方式，直接下载HTTPS密码，当克隆、推送代码时直接输入HTTPS用户名密码即可。
- 同一仓库SSH和HTTPS的地址不同。

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。
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/repo1/”下多了一个“.git”文件夹。
   ```
   git init
   ```
   b. 关联云端代码仓库。
   ```
   git remote add origin repoUrl
   ```
   仓库地址如下图进入仓库详情页，单击”**克隆/下载**“，所示单击红色方框处获取。

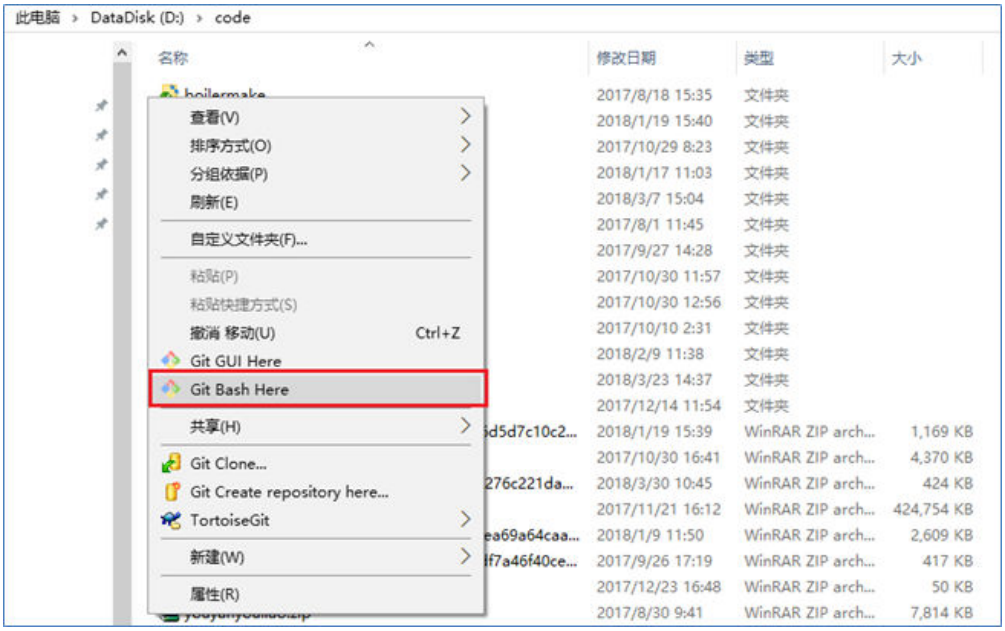
   

   c. 推送代码到云仓库。
   ```
   git  add .
   git  commit  -m "init project"
   git  branch --set-upstream-to=origin/master master
   git  pull  --rebase
   git  push
   ```

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。



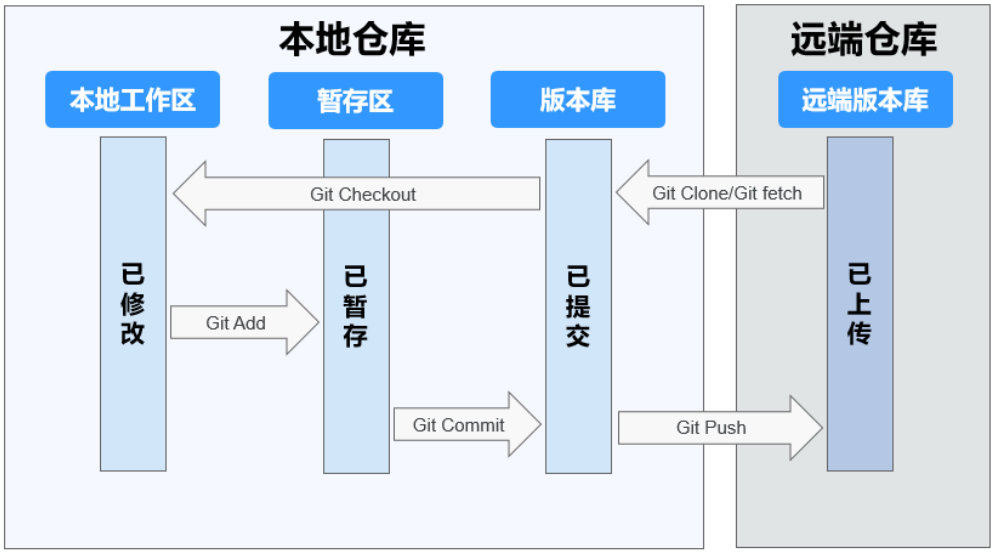
2. 克隆仓库，URL地址如下图所示单击红色方框处获取。

```
git clone repoUrl  //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

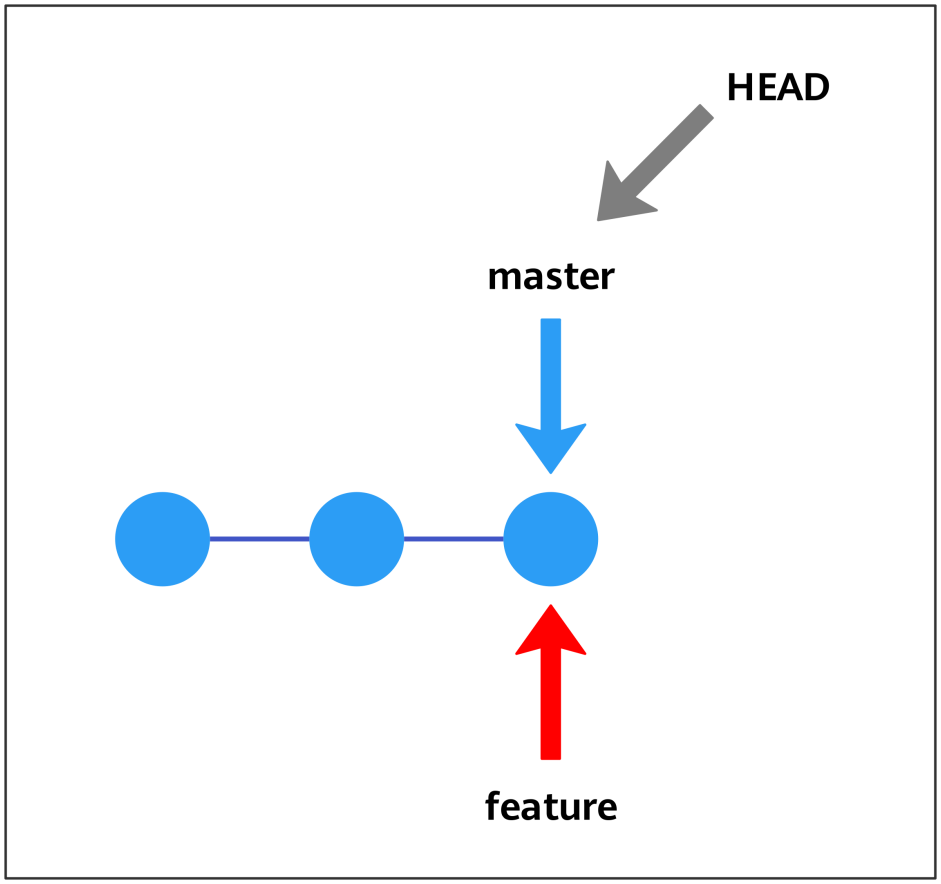
- 新建分支

  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。

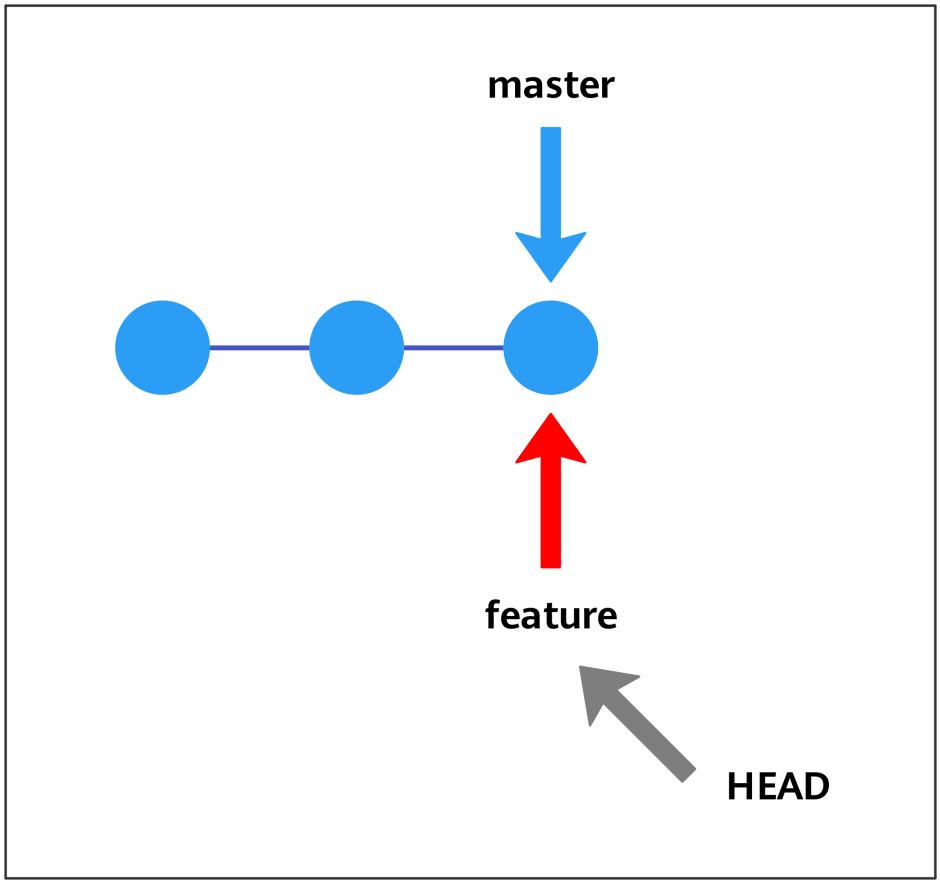
  ```
  #git branch branchName commitID
  ```

  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。



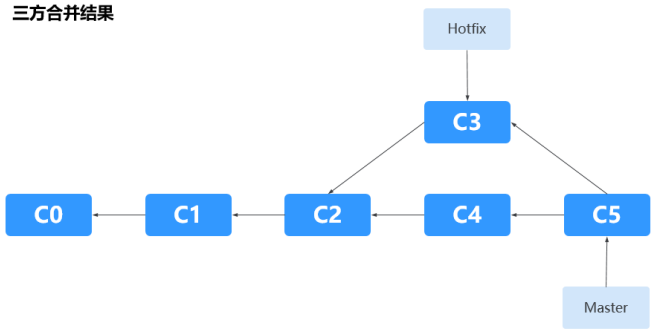
- 分支合并

  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：

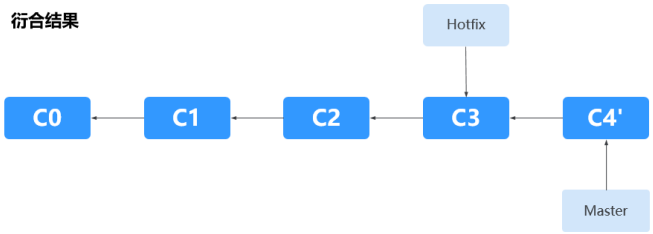
  a. 三方包括hotfix新增节点C3，master新增节点C4，以及两者的共同祖先节点C2。这种合并操作简单，但新增合并节点C5，形成了环形，版本记录可读性差，如下图所示。

  ```
  #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。

2 将仓库迁移至代码托管平台

本实践将展示如何将您自有的**本地仓库**或**云端仓库**迁移至代码托管服务。

应用场景

随着**“代码上云”**浪潮的推动，仓库的自主迁移趋于常态化，代码托管针对准备进行仓库迁移的用户提供完整的操作指导，帮助您将仓库迁移至代码托管服务。

实现原理

代码托管服务（CodeArts Repo）结合仓库的存储方式提供以下迁移方案：

- **HTTP在线导入**

  通过HTTP协议直接将您的远程仓库导入到代码托管中，全程线上操作，但导入仓库的时长会受到网络条件及仓库容量的影响。

  说明

  仓库容量相对较大的云端仓库推荐使用**Git客户端推送**的方式进行迁移。

- **Git客户端推送**

  通过使用Git客户端将本地仓库中的代码文件推送至代码托管服务。

  - 将项目文件存放在本地计算机的用户，建议先将本地项目文件初始化成Git仓库，再使用Git客户端进行迁移。
  - 新建仓库对于仓库容量相对较大的云端仓库，建议先将云端仓库克隆或下载到本地，再使用Git客户端进行迁移。

前提条件

- 已有可用项目，如果没有，请先新建项目。
- 已有可用仓库，如果没有，请先新建仓库。
- 在仓库迁移的过程中，要保持网络稳定畅通。
- 被迁移仓库的容量不能超过2GB，否则新建的代码托管仓库会被冻结，不可使用。

HTTP 在线导入

步骤1 进入软件开发生产线首页，单击目标项目名称，进入项目。

步骤2 单击菜单**“代码 > 代码托管”**，进入代码托管服务。

步骤3 在代码托管仓库列表页，单击**“普通新建”**旁的图标，在下拉列表中单击**“导入外部仓库”**。

步骤4 在**“填写外部仓库信息”**页面，根据实际情况填写以下参数。

表 2-1 外部仓库参数说明

参数项	是否必填	说明
源仓库路径	是	填写源仓库的仓库地址，以（http://）或（https://）开头，以（.git）结尾。
源仓库访问权限	是	• **不需要用户名/密码**：如果源仓库是开源仓库（公开仓），请勾选该选项。 • **需要用户名/密码**：如果源仓库是私有仓库，请勾选该选项，并填写HTTPS克隆代码时的用户名及密码。

步骤5 勾选**“我已阅读并同意《隐私政策声明》和《CodeArts服务使用声明》”**，单击**“下一步”**。

步骤6 在**“创建仓库”**页面，根据实际情况填写以下参数。

表 2-2 “同步仓库设置”参数填写说明

字段名称	是否必填	备注说明
分支设置	是	可选择同步源仓库的**默认分支**或**全部分支**。
增加定时同步	否	同步分为手动同步和定时同步，同步分支配置后不可更改。如果勾选**“增加定时同步”**功能： • 每天自动从源仓库导入仓库的默认分支。 • 仓库将成为只读镜像仓库，不能写入，并且只同步当前创建仓库的默认分支对应的第三方仓库的分支。

步骤7 单击**“确定”**，完成仓库导入，跳转到仓库列表页。

----结束

Git 客户端推送（以 Git Bash 为例）

说明

在进行Git Bash客户端推送之前请确保已在代码托管服务中**配置SSH密钥或HTTPS密码**。

步骤1 进入目标代码托管服务。

步骤2 将本地仓库初始化为Git仓库，用于与代码托管仓库进行关联。

在您的仓库中打开Git Bash客户端，执行以下命令：

```
git init
```

初始化成功如下图，此时当前文件夹已经是本地Git仓库了。

步骤3 将本地仓库与代码托管仓库进行绑定。

1. 进入代码托管仓库，获取仓库地址。
2. 在本地使用remote命令，将本地仓库与代码托管仓库进行绑定。

   ```
   git remote add 仓库别名 仓库地址
   ```

   示例为：

   ```
   git remote add origin git@*****/java-remote.git  #复制使用时注意换成您自己的仓库地址
   ```

   说明

   - 一般用origin作为仓库别名，因为当您从远程仓库clone到本地时，默认产生的别名就是origin，您也可以使用任意别名。
   - 如果提示仓库名重复，更换一个即可。
   - 无回显即为绑定成功。

步骤4 将代码托管仓库master分支拉取到本地仓库。

此步骤主要是避免冲突。

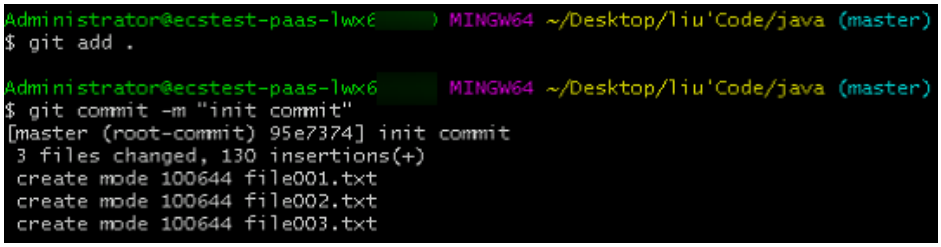
```
git fetch origin master  #复制使用时 注意是否需要将origin替换为您仓库的别名
```

步骤5 将本地代码文件提交到master分支。

依次执行：

```
git add .
git commit -m "您的提交备注"
```

下图为成功的执行。



步骤6 将本地master分支与代码托管master分支进行绑定。

```
git branch --set-upstream-to=origin/master master  #复制使用时 注意是否需要将origin替换为您仓库的别名
```


成功执行如下图所示，提示您已经将合并后的仓库放在工作区与版本库。

步骤7 合并代码托管仓库与本地仓库的文件，并存储在本地。

```
git pull --rebase origin master   #复制使用时 注意是否需要将origin替换为您仓库的别名
```

成功执行如下图所示，提示您已经将合并后的仓库放在工作区与版本库。

步骤8 将本地仓库推送覆盖代码托管仓库。

因为之前已经进行了绑定，直接push即可。

```
git push
```

成功后，再直接拉取pull，验证代码托管仓库与本地仓库版本相同，如下图。

----结束

3 代码检视操作实践

说明

“检视意见设置”与“检视意见模板设置”仅支持“专业版套餐及以上”用户。

背景信息

代码检视是指开发人员在完成代码设计、编写、调试后指派给其它人员对代码展开阅读、审视的过程。代码检视的目的是发现代码中的格式、逻辑、语法等规范性和正确性问题，从而保证代码的质量。由于在代码检视阶段发现代码问题的成本是最低的，因此严格认真的执行代码检视对于提升代码质量是十分必要的。为了能使您更高效、快捷地进行代码检视，您可根据自己的实际需求在代码托管服务添加检视意见设置模板。

操作前提

- 已有可用项目，如果没有，请先**新建项目**。

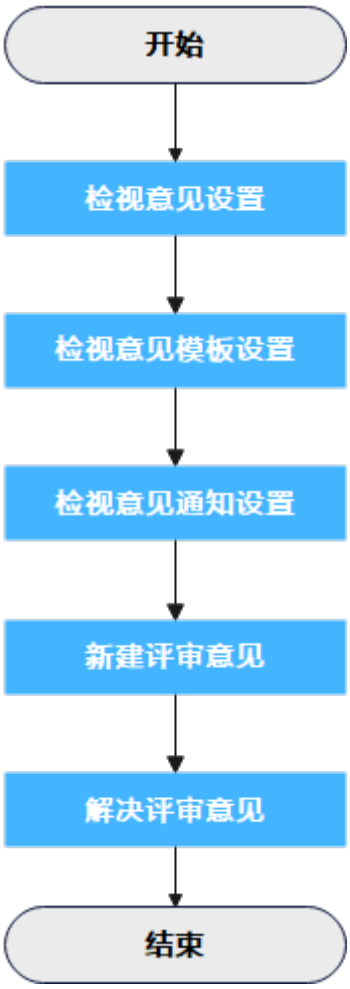
  说明

  - 如果您购买CodeArts整体服务的套餐，则需要在需求管理服务中进行**新建项目**。
  - 如果您购买CodeArts Repo单服务的套餐，则需要在新建仓库的同时进行新建项目，此时创建的项目为Scrum项目或IPD项目。
- 已有可用仓库，如果没有，请先**新建仓库**。

操作流程

在开始代码检视之前，需要管理员先做如下管理设置，再进行代码检视，具体流程如下：

图 3-1 代码检视操作过程



本文详细描述了检视意见设置、检视评论模板设置、检视意见通知设置、新建评审意见和解决评审意见整个操作的完整过程。大致分为以下步骤，您可根据熟悉程度选择阅读：

- **检视意见设置**
- **检视意见模板设置**
- **检视意见通知设置**
- **新建评审意见**
- **解决评审意见**

检视意见设置

检视意见设置用于规范检视意见及配置**检视意见模板设置**，此设置只针对被设置的仓库生效，只有仓库管理员和仓库所有者能看到这个页面且有设置权限。

步骤1 进入软件开发生产线首页，单击目标项目名称，进入项目。

步骤2 单击菜单“**代码 > 代码托管**”，进入代码托管服务。

步骤3 进入仓库详情，单击“**设置 > 策略设置 > 检视意见**”，进入检视意见设置页面。

步骤4 根据需要选择是否勾选“**启用检视意见分类与模块**”启用检视意见。

步骤5 配置检视意见分类。

- 启用系统预置检视意见分类
  勾选“**启用系统预置检视意见分类**”，可直接使用系统预置检视意见分类。
- 自定义分类
  支持自定义检视意见分类，输入“**类型名称**”如代码规范问题，按Enter键保存。

  说明

  名称最多200个字符，最多支持新建20个。

步骤6 在“**检视意见模块设置**”下输入框输入“**类型名称**”。

说明

名称最多200个字符，最多支持新建20个。

步骤7 根据您的需要勾选“**启用新建/编辑检视意见时必填字段校验**”。

步骤8 单击“**提交**”，保存设置。

----结束

检视意见模板设置

检视评论模板位于仓库详情中的“**设置 > 模板管理 > 检视意见模板**”。您可新建、编辑和删除检视意见模板，并根据自身习惯定制检视意见模板信息，包括：严重程度、指派给、意见分类、意见模板和描述。当仓库成员进行检视评论时，您可选择一个检视意见模板，模板内容将会自动应用到合并请求或待检视代码文件上。此设置只针对被设置的仓库生效。只有仓库管理员和仓库所有者能看到这个页面且有设置权限。

您可根据表3-1新建检视意见模板。

表 3-1 字段说明

字段	说明
模板名称	必填项，新建模板的名称。如代码检视意见模板。
设置为默认模板	非必填项，勾选后，进行检视评论时默认应用此模板。
严重程度	非必填项，根据问题的严重程度可分为“致命”、“严重”、“一般”、“建议”四种类型。如“严重程度”为“一般”。

字段	说明
指派给	非必填项。 • 指派给为“**空**”。 – MR中添加检视意见时，默认指派给MR创建者。 – 文件或Commit中添加检视意见时，默认不指派。 • 指派给为“**MR创建者/Commit作者**”。 – MR中添加检视意见时，默认指派给MR创建者。 – 文件或Commit中添加检视意见时，默认指派给Commit作者。 • 指派给为“**具体人员**”。 – MR中添加检视意见时，默认指派给具体人员。 – 文件或Commit中添加检视意见时，默认指派给具体人员。 如指派给为“MR创建者”。
意见分类	非必填项，默认禁用，需“启用检视意见分类与模块”并配置检视意见分类才可设置，详情请参见**检视意见**。
意见模块	非必填项，默认禁用，需“启用检视意见分类与模块”后配置检视意见模块才可设置，详情请参考**检视意见**。
描述	非必填项，填写模板的描述信息，支持描述信息预览。如代码格式问题。

检视意见通知设置

通知设置位于仓库详情的“**设置 > 基本设置 > 通知设置**”。

图 3-2 通知设置页面

- 新建评审意见：可手动设置给MR创建人发送邮件通知。
- 解决评审意见：可手动设置给MR创建人发送邮件通知。

新建评审意见

您可以在“**代码**”页签的“**文件**”和“**提交**”子菜单下，也可以在“**合并请求**”中“**文件变更**”子菜单下给待检视文件添加检视意见。

您可以在“合并请求”详情页“评审意见”下给待检视的合并请求添加评论，也可以在“代码”页签下的“提交”子菜单下给提交文件添加评论。

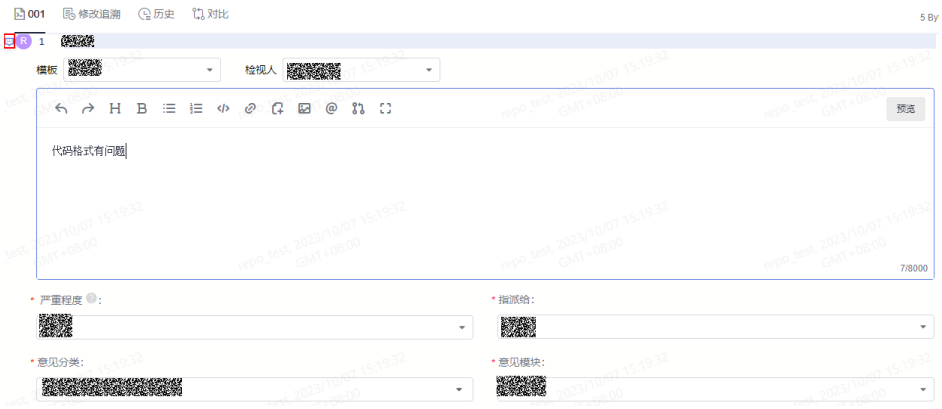
其中在“**代码**”页签的“**文件**”和“**提交**”子菜单下添加的检视意见和评论可以在“**评审记录**”页签下“**源自Commit的评审记录**”中查看。

在“**合并请求**”中“**文件变更**”子菜单下给待检视文件添加检视意见和在“**合并请求**”详情页“**评审意见**”中添加的评论可以在“**评审记录**”页签下“**源自合并请求的评审记录**”中查看。

- 在“**代码**”页签下“**文件**”子菜单，新建检视意见。

  进入“**文件**”子菜单，单击“待评审文件”，单击代码行  图标，在“文本框”输入评审意见，选择“**严重程度**”和“**指派给**”，如“**严重程度**”为“一般”，“**指派给**”为“MR创建者”，在下拉框选择“**意见分类**”和“**意见模块**”，单击“**确定**”完成检视意见添加。

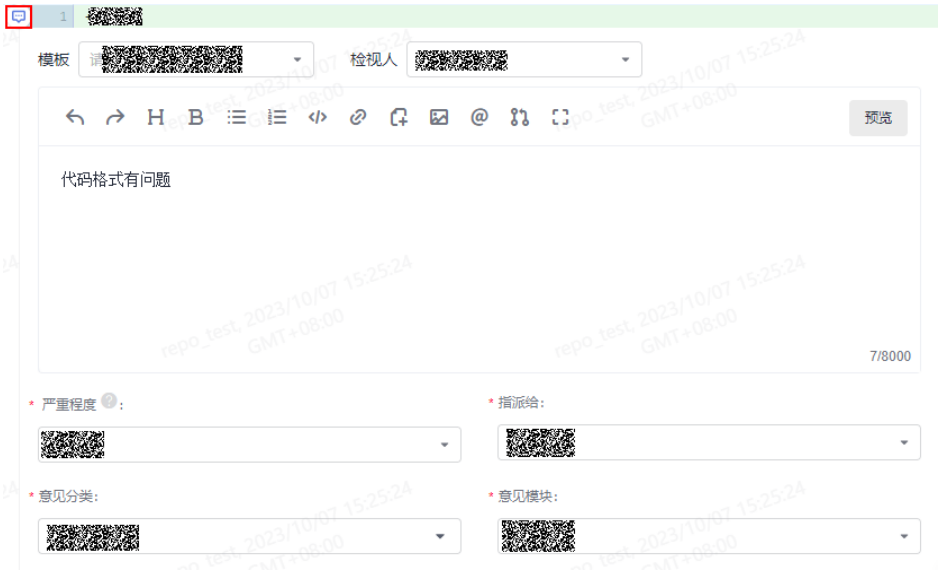
  图 3-3 新建检视意见

- 在“**代码**”页签下“**提交**”子菜单，新建检视意见。

  进入“**提交**”子菜单，单击“**提交记录**”下“待检视的文件”，单击代码行  图标，在“文本框”输入评审意见，选择“**严重程度**”和“**指派给**”，如“**严重程度**”为“一般”，“**指派给**”为“MR创建者”，在下拉框选择“**意见分类**”和“**意见模块**”，单击“**确定**”完成检视意见添加。

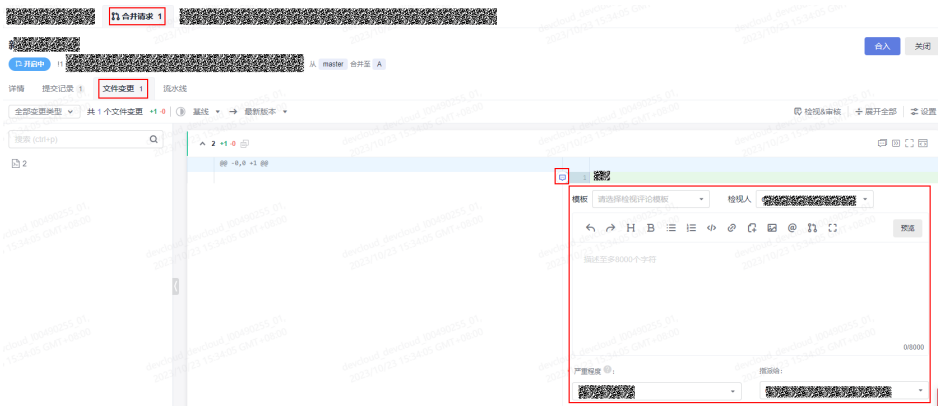
图 3-4 新建检视意见



- 在“**合并请求**”页签中“**文件变更**”子菜单下，新建检视意见。

  进入“**文件变更**”子菜单，单击代码行图标，在“文本框”输入评审意见，选择“**严重程度**”和“**指派给**”，如“**严重程度**”为“一般”，“**指派给**”为“MR创建者”，在下拉框选择“**意见分类**”和“**意见模块**”，单击“**确定**”完成检视意见添加。

图 3-5 新建检视意见



- 在“**合并请求**”页签中合并请求详情页，新建评论。

  进入“**合并请求**”页签，单击“待检视的合并请求”，进入合并请求详情页。单击“**评审记录**”，在评审记录下输入评审意见，单击“**确定**”完成评论的添加。
- 在“**提交**”页面中，单击某个提交，切换“**评论**”界面，即可新建评论。

图 3-6 新建评论

解决评审意见

您可在“**评审记录**”页签，查看“**源自合并请求的评审记录**”和“**源自Commit的评审记录**”。

- 根据“**源自Commit的评审记录**”评审意见修改代码文件后，通知committer审核，审核通过后把状态修改为  。
- 根据“**源自合并请求的评审记录**”评审意见修改合并请求中代码文件后，通知committer审核，审核通过后把状态修改为  。

4 HE2E DevOps 实践——管理代码

背景介绍

本文以“DevOps全流程示例项目”为例，介绍如何在项目中进行代码开发。

开展实践前，需要完成**创建项目**。

本样例项目中采用分支来进行代码的开发。首先由开发人员Chris在代码仓库中创建分支，并进行代码开发；然后开发人员Chris在代码仓库中提交分支合并请求，项目经理Maggie评审通过后合并分支至主干。

创建分支

步骤1 将master分支设置为受保护分支（本文档中由项目经理Maggie操作）。

1. 进入“凤凰商城”项目，单击导航“代码 > 代码托管”，找到代码仓库“phoenix-sample”。
2. 单击仓库名称进入代码仓库，选择“设置”页签。在导航中单击“策略设置 > 保护分支”。
3. 单击“新建保护分支”，参照下表在弹框中完成配置，单击“确定”。

表 4-1 新建保护分支配置

配置项	配置建议
选择需要保护的分支	选择“master”。
能推送	保持默认配置。
能合并	保持默认配置。
成员	保持默认配置。

说明

如果页面中已存在保护分支“master”，可单击，根据需要修改保护分支配置。

步骤2 创建功能分支（本文档中由开发者Chris操作）。

1. 进入“凤凰商城”项目，在代码托管页面中找到仓库“phoenix-sample”。
2. 单击仓库名称进入代码仓库，在“代码”页签中单击“分支”。
3. 单击“新建分支”，参照表4-2输入分支信息，单击“确定”。

表 4-2 新建分支

配置项	配置建议
基于	选择“master”。
分支名称	输入“Feature-Store”。
关联工作项	选择“作为用户可以查询所有门店网络”。

----结束

修改、提交代码

步骤1 单击导航“工作 > 需求管理”，选择“迭代”页签。

在迭代4中找到Task“前端展示 - 添加门店网络菜单”，将Task的状态修改为“进行中”。

步骤2 单击导航“代码 > 代码托管”，找到仓库“phoenix-sample”。

步骤3 单击仓库名称进入代码仓库，选择“代码”页签。

步骤4 单击文件列表上方的“master”，在下拉列表中选择分支“Feature-Store”。

步骤5 在文件列表中找到“vote/templates/store-network.html”并打开。

步骤6 单击，根据Story添加门店地址，并在页面底部文本框中输入备注信息“添加门店列表”，单击“确定”。

```
<ul>
   <li>A分店：E机场1号航站楼出发层靠右直行123米右侧</li>
   <li>B分店：F区G路456号</li>
   <li>C分店：H区J街789号</li>
   <li>D分店：K区L大道K大楼西侧</li>
</ul>
```

步骤7 以同样方法打开并编辑“/vote/templates/index.html”。

在179行添加菜单“门店网络”，输入提交信息“fix #xxxxxx 前端展示 - 添加门店网络菜单”，单击“确定”。

其中“#xxxxxx”是Task“前端展示 - 添加门店网络菜单”的编号，在工作项列表中获取，实践中修改为实际Task的编号。

```
<li class="nav-item"> <a href="store-network" class="nav-link">门店网络</a> </li>
```

步骤8 单击导航“工作 > 需求管理”，选择“迭代”页签。

在迭代4中找到Task“前端展示 - 添加门店网络菜单”。

- 单击Task名称，在详情页中可看到状态自动变为“已解决”。
- 选择“关联”页签，在“代码提交记录”下可看到一条记录，记录的描述与上一步中输入的提交信息相同。

图 4-1 代码提交记录

----结束

检视代码、合并分支

步骤1 开发人员发起合并请求。

开发人员Chris完成代码开发，确认无误后，即可发起合并请求，将功能分支合并到master中。

1. 进入代码仓库，选择“合并请求”页签，单击“新建合并请求”。
2. 源分支选择“Feature-Store”，目标分支选择“master”，单击“下一步”。
3. 参照表4-3编辑合并请求详情。

表 4-3 合并请求配置

配置项	配置建议
标题	输入“添加门店网络列表”。
合并人	单击，在弹框中勾选“Maggie”，单击“确定”。
审核人	单击，在弹框中勾选“Maggie”，单击“确定”。

4. 单击“新建合并请求”完成合并请求的创建。

步骤2 项目经理评审并完成代码合入。

本文档中，合并请求的评审人与合并人均是项目经理Maggie。因此Maggie可评审合并请求内容，并在评审通过后完成分支合入。

1. 进入代码仓库后，选择“合并请求”页签，可找到由开发人员Chris创建的合并请求。
2. 单击该请求，查看合并请求详情。
3. 可在页面中留下评审意见。单击审核门禁中“通过”完成审核。
4. 单击“合入”，将分支合入“master”。

说明

如果发起分支合并请求时勾选了“合并后删除源分支”，分支“Feature-Store”将在分支合并完成后被删除。

----结束

5 批量迁移 GitLab 内网仓库到 CodeArts Repo

背景介绍

CodeArts Repo现有迁仓能力只支持公网之间迁移，缺少客户内网自建代码托管平台往Repo迁移的快速方案，因此提供批量迁移内网代码托管平台仓库到Repo的脚本。

配置访问 CodeArts Repo 的 SSH 公钥

在进行批量迁移GitLab的代码仓到CodeArts Repo前，您需要安装Git Bash客户端，并且把本地生成的SSH公钥配置到CodeArts Repo，具体操作步骤如下：

步骤1 运行Git Bash，先检查本地是否已生成过SSH密钥。

如果选择RSA算法，请在Git Bash中执行如下命令：

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

如果选择ED255219算法，请在Git Bash中执行如下命令：

```
cat ~/.ssh/id_ed25519.pub
```

- 如果提示“No such file or directory”，说明您这台计算机没生成过SSH密钥，请继续执行**步骤2**。
- 如果返回以ssh-rsa或ssh-ed25519开头的字符串，说明您这台计算机已经生成过SSH密钥，如果想使用已经生成的密钥请直接跳到**步骤3**，如果想重新生成密钥，请从**步骤2**向下执行。

步骤2 生成SSH密钥。如果选择RSA算法，在Git Bash中生成密钥的命令如下：

```
ssh-keygen -t rsa -b 4096 -C your_email@example.com
```

其中，-t rsa表示生成的是RSA类型密钥，-b 4096是密钥长度（该长度的RSA密钥更具安全性），-C your_email@example.com表示在生成的公钥文件中添加注释，方便识别这个密钥对的用途。

如果选择ED25519算法，在Git Bash中生成密钥的命令如下：

```
ssh-keygen -t ed25519 -b 521 -C your_email@example.com
```

其中，-t ed25519表示生成的是ED25519类型密钥，-b 521是密钥长度（该长度的ED25519密钥更具安全性），-C your_email@example.com表示在生成的公钥文件中添加注释，方便识别这个密钥对的用途。

输入生成密钥的命令后，直接回车，密钥会默认存储到~/.ssh/id_rsa或者~/.ssh/id_ed25519路径下，对应的公钥文件为~/.ssh/id_rsa.pub或者~/.ssh/id_ed25519.pub。

步骤3 复制SSH公钥到剪切板。请根据您的操作系统，选择相应的执行命令，将SSH公钥复制到您的剪切板。

- **Windows:**
  ```
  clip < ~/.ssh/id_rsa.pub
  ```
- **Mac:**
  ```
  pbcopy < ~/.ssh/id_rsa.pub
  ```
- **Linux (xclip required):**
  ```
  xclip -sel clip < ~/.ssh/id_rsa.pub
  ```

步骤4 登录并进入Repo的代码仓库列表页，单击右上角昵称，选择“个人设置” > “代码托管” > “SSH密钥”，进入配置SSH密钥页面。

也可以在Repo的代码仓库列表页，单击右上角“设置我的SSH密钥”，进入配置SSH密钥页面。

步骤5 在**“标题”**中为您的新密钥起一个名称，将您在**步骤3**中复制的SSH公钥粘贴进**“密钥”**中，单击确定后，弹出页面“密钥已设置成功，单击 立即返回，无操作3S后自动跳转”，表示密钥设置成功。

----结束

批量迁移 GitLab 内网仓库到 CodeArts Repo

步骤1 进入**Python官网**下载并安装Python3。

步骤2 登录GitLab并获取private_token，在“用户设置”里，选择“访问令牌” > “添加新令牌”。

步骤3 您需要在本地生成SSH公钥并配置到GitLab和CodeArts Repo，其中配置到CodeArts Repo可参考**配置访问CodeArts Repo的SSH公钥**。

步骤4 调试**获取IAM用户Token(使用密码)**接口，通过华为云账号的用户密码获取用户Token。参数的填写方法，您可以在接口的调试界面，单击右侧“请求示例”，填写好参数后，单击“调试”，将获取到的用户Token复制并保存到本地。

步骤5 用获取到的用户Token配置“config.json”文件。其中，source_host_url是您内网的GitLab的接口地址，repo_api_prefix是CodeArts Repo 的openAPI地址。

```
{
   "source_host_url": "http://{source_host}/api/v4/projects?page=1&per_page=5",
   "private_token": "GitLab上获取的private_token",
   "repo_api_prefix": "https://${open_api}",
   "x_auth_token": "用户Token"
}
```

步骤6 登录CodeArts首页创建项目并保存您的项目ID。

步骤7 用获取的项目ID配置“plan.json”文件，如下的示例表示两个代码仓的迁移配置，您可以根据需要进行配置。此处的g1/g2/g3表示代码组路径，如果没有提前设置，根据该配置会自动生成。

```
[
   ["path_with_namespace", "项目ID", "g1/g2/g3/目标仓库名1"],
      ["path_with_namespace", "项目ID", "g1/g2/g3/目标仓库名2"]
]
```


说明

- 代码组的创建请进入CodeArts Repo首页，单击“新建仓库”旁的下拉框，选择“新建代码组”。
- 代码仓库的名字需要以大小写字母、数字、下划线开头，可包含大小写字母、数字、中划线、下划线、英文句点，但不能以.git、.atom或.结尾。

步骤8 在本地Python控制台，创建migrate_to_repo.py文件。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import json
import logging
import os
import subprocess
import time
import urllib.parse
import urllib.request
from logging import handlers

# 存在同名仓库时是否跳过
SKIP_SAME_NAME_REPO = True

STATUS_OK = 200
STATUS_CREATED = 201
STATUS_INTERNAL_SERVER_ERROR = 500
STATUS_NOT_FOUND = 404
HTTP_METHOD_POST = "POST"
CODE_UTF8 = 'utf-8'
FILE_SOURCE_REPO_INFO = 'source_repos.json'
FILE_TARGET_REPO_INFO = 'target_repos.json'
FILE_CONFIG = 'config.json'
FILE_PLAN = 'plan.json'
FILE_LOG = 'migrate.log'
X_AUTH_TOKEN = 'x-auth-token'


class Logger(object):
    def __init__(self, filename):
        format_str = logging.Formatter('%(asctime)s - %(pathname)s[line:%(lineno)d] - %(levelname)s: %
(message)s')
        self.logger = logging.getLogger(filename)
        self.logger.setLevel(logging.INFO)
        sh = logging.StreamHandler()
        sh.setFormatter(format_str)
        th = handlers.TimedRotatingFileHandler(filename=filename, when='D', backupCount=3,
encoding=CODE_UTF8)
        th.setFormatter(format_str)
        self.logger.addHandler(sh)
        self.logger.addHandler(th)


log = Logger(FILE_LOG)


def make_request(url, data={}, headers={}, method='GET'):
    headers["Content-Type"] = 'application/json'
    headers['Accept-Charset'] = CODE_UTF8
    params = json.dumps(data)
    params = bytes(params, 'utf8')
    try:
        import ssl
        ssl._create_default_https_context = ssl._create_unverified_context
        request = urllib.request.Request(url, data=params, headers=headers, method=method)
        r = urllib.request.urlopen(request)
        if r.status != STATUS_OK and r.status != STATUS_CREATED:
            log.logger.error('request error: ' + str(r.status))
            return r.status, ""
    except urllib.request.HTTPError as e:
```



```
        log.logger.error('request with code: ' + str(e.code))
        msg = str(e.read().decode(CODE_UTF8))
        log.logger.error('request error: ' + msg)
        return STATUS_INTERNAL_SERVER_ERROR, msg
    content = r.read().decode(CODE_UTF8)
    return STATUS_OK, content


def read_migrate_plan():
    log.logger.info('read_migrate_plan start')
    with open(FILE_PLAN, 'r') as f:
        migrate_plans = json.load(f)
    plans = []
    for m_plan in migrate_plans:
        if len(m_plan) != 3:
            log.logger.error("line format not match \"source_path_with_namespace\",\"project_id
\",\"target_namespace\"")
            return STATUS_INTERNAL_SERVER_ERROR, []
        namespace = m_plan[2].split("/")
        if len(namespace) < 1 or len(namespace) > 4:
            log.logger.error("group level support 0 to 3")
            return STATUS_INTERNAL_SERVER_ERROR, []
        l = len(namespace)
        plan = {
            "path_with_namespace": m_plan[0],
            "project_id": m_plan[1],
            "groups": namespace[0:l - 1],
            "repo_name": namespace[l - 1]
        }
        plans.append(plan)
    return STATUS_OK, plans


def get_repo_by_plan(namespace, repos):
    if namespace not in repos:
        log.logger.info("%s not found in gitlab, skip" % namespace)
        return STATUS_NOT_FOUND, {}

    repo = repos[namespace]
    return STATUS_OK, repo


def repo_info_from_source(config):
    if os.path.exists(FILE_SOURCE_REPO_INFO):
        log.logger.info('get_repos skip: %s already exist' % FILE_SOURCE_REPO_INFO)
        return STATUS_OK

    log.logger.info('get_repos start')
    headers = {'PRIVATE-TOKEN': config['private_token']}
    url = config['source_host_url']
    per_page = 100
    page = 1
    data = {}

    while True:
        url_with_page = "%s&page=%s&per_page=%s" % (url, page, per_page)
        status, content = make_request(url_with_page, headers=headers)
        if status != STATUS_OK:
            return status
        repos = json.loads(content)
        for repo in repos:
            namespace = repo['path_with_namespace']
            repo_info = {'name': repo['name'], 'id': repo['id'], 'path_with_namespace': namespace,
                         'ssh_url': repo['ssh_url_to_repo']}
            data[namespace] = repo_info
        if len(repos) < per_page:
            break
        page = page + 1
```

```
    with open(FILE_SOURCE_REPO_INFO, 'w') as f:
        json.dump(data, f, indent=4)
    log.logger.info('get_repos end with %s' % len(data))
    return STATUS_OK


def get_repo_dir(repo):
    return "repo_%s" % repo['id']


def exec_cmd(cmd, ssh_url, dir_name):
    log.logger.info("will exec %s %s" % (cmd, ssh_url))
    pr = subprocess.Popen(cmd + " " + ssh_url, cwd=dir_name, shell=True, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
    (out, error) = pr.communicate()
    log.logger.info("stdout of %s is:%s" % (cmd, str(out)))
    log.logger.info("stderr of %s is:%s" % (cmd, str(error)))
    if "Error" in str(error) or "err" in str(error) or "failed" in str(error):
        log.logger.error("%s failed" % cmd)
        return STATUS_INTERNAL_SERVER_ERROR
    return STATUS_OK


def clone_from_source(config, plans):
    log.logger.info('clone_repos start')
    with open(FILE_SOURCE_REPO_INFO, 'r') as f:
        repos = json.load(f)
    for plan in plans:
        status, repo = get_repo_by_plan(plan["path_with_namespace"], repos)
        if status == STATUS_NOT_FOUND:
            return status

        name = repo["name"]
        dir_name = get_repo_dir(repo)
        folder = os.path.exists(dir_name)
        if folder:
            log.logger.info("skip clone " + name)
            continue
        os.makedirs(dir_name)
        status = exec_cmd("git clone --mirror", repo['ssh_url'], dir_name)
        if status != STATUS_OK:
            return status
    log.logger.info('clone_repos end')
    return STATUS_OK


def get_groups(config, project_id):
    log.logger.info('get_groups start')
    headers = {X_AUTH_TOKEN: config['x_auth_token']}
    api_prefix = config['repo_api_prefix']
    limit = 100
    offset = 0
    data = {}
    while True:
        url_with_page = "%s/v4/%s/manageable-groups?offset=%s&limit=%s" % (api_prefix, project_id, offset,
limit)
        status, content = make_request(url_with_page, headers=headers)
        if status != STATUS_OK:
            return status, dict()
        rows = json.loads(content)
        for row in rows:
            full_name = row['full_name']
            data[full_name] = row
        if len(rows) < limit:
            break
        offset = offset + len(rows)
    log.logger.info('get_groups end with %s' % len(data))
    return STATUS_OK, data
```

```
def create_group(config, project_id, name, parent, has_parent):
    log.logger.info('create_group start')
    headers = {X_AUTH_TOKEN: config['x_auth_token']}
    api_prefix = config['repo_api_prefix']
    data = {
        'name': name,
        'visibility': 'private',
        'description': ''
    }
    if has_parent:
        data['parent_id'] = parent['id']

    url = "%s/v4/%s/groups" % (api_prefix, project_id)
    status, content = make_request(url, data=data, headers=headers, method='POST')
    if status != STATUS_OK:
        log.logger.error('create_group error: %s', str(status))
        return status
    return STATUS_OK


# 指定代码组创建仓库
def create_repo(config, project_id, name, parent, has_parent):
    log.logger.info('create_repo start')
    headers = {X_AUTH_TOKEN: config['x_auth_token']}
    api_prefix = config['repo_api_prefix']
    data = {
        'name': name,
        'project_uuid': project_id,
        'enable_readme': 0
    }
    if has_parent:
        data['group_id'] = parent['id']
    url = "%s/v1/repositories" % api_prefix
    status, content = make_request(url, data=data, headers=headers, method='POST')
    if "同名仓库或代码组" in content:
        log.logger.info("repo %s already exist. %s" % (name, content))
        log.logger.info("skip same name repo %s: %s" % (name, SKIP_SAME_NAME_REPO))
        return check_repo_conflict(config, project_id, parent, name)
    elif status != STATUS_OK:
        log.logger.error('create_repo error: %s', str(status))
        return status, ""
    response = json.loads(content)
    repo_uuid = response["result"]["repository_uuid"]

    # 创建后检查
    for retry in range(1, 4):
        status, ssh_url = get_repo_detail(config, repo_uuid)
        if status != STATUS_OK:
            if retry == 3:
                return status, ""
            time.sleep(retry * 2)
            continue
        break

    return STATUS_OK, ssh_url


def check_repo_conflict(config, project_id, group, name):
    if not SKIP_SAME_NAME_REPO:
        return STATUS_INTERNAL_SERVER_ERROR, ""

    log.logger.info('check_repo_conflict start')
    headers = {X_AUTH_TOKEN: config['x_auth_token']}
    api_prefix = config['repo_api_prefix']
    url_with_page = "%s/v2/projects/%s/repositories?search=%s" % (api_prefix, project_id, name)
    status, content = make_request(url_with_page, headers=headers)
    if status != STATUS_OK:
        return status, ""
```

```
    rows = json.loads(content)
    for row in rows["result"]["repositories"]:
        if "full_name" in group and "group_name" in row:
            g = group["full_name"].replace(" ", "")
            if row["group_name"].endswith(g):
                return STATUS_OK, row["ssh_url"]
        elif "full_name" not in group and name == row['repository_name']:
            # 没有代码组的场景
            return STATUS_OK, row["ssh_url"]

    log.logger.info('check_repo_conflict end, failed to find: %s' % name)
    return STATUS_INTERNAL_SERVER_ERROR, ""


def get_repo_detail(config, repo_uuid):
    log.logger.info('get_repo_detail start')
    headers = {X_AUTH_TOKEN: config['x_auth_token']}
    api_prefix = config['repo_api_prefix']
    url_with_page = "%s/v2/repositories/%s" % (api_prefix, repo_uuid)
    status, content = make_request(url_with_page, headers=headers)
    if status != STATUS_OK:
        return status, ""
    rows = json.loads(content)
    log.logger.info('get_repo_detail end')
    return STATUS_OK, rows["result"]["ssh_url"]


def process_plan(config, plan):
    # 获取项目下的组织列表
    project_id = plan["project_id"]
    status, group_dict = get_groups(config, project_id)
    if status != STATUS_OK:
        return status, ""
    group = ""
    last_group = {}
    has_group = False
    for g in plan["groups"]:
        # 检查目标代码组，如果存在则检查下一层
        if group == "":
            group = " %s" % g
        else:
            group = "%s / %s" % (group, g)
        if group in group_dict:
            last_group = group_dict[group]
            has_group = True
            continue
        # 不存在则创建，并更新
        status = create_group(config, project_id, g, last_group, has_group)
        if status != STATUS_OK:
            return status, ""
        status, group_dict = get_groups(config, project_id)
        if status != STATUS_OK:
            return status, ""
        last_group = group_dict[group]
        has_group = True

    status, ssh_url = create_repo(config, project_id, plan["repo_name"], last_group, has_group)
    if status != STATUS_OK:
        return status, ""

    return status, ssh_url


def create_group_and_repos(config, plans):
    if os.path.exists(FILE_TARGET_REPO_INFO):
        log.logger.info('create_group_and_repos skip: %s already exist' % FILE_TARGET_REPO_INFO)
        return STATUS_OK

    log.logger.info('create_group_and_repos start')
```

```
    with open(FILE_SOURCE_REPO_INFO, 'r') as f:
        repos = json.load(f)
        target_repo_info = {}
    for plan in plans:
        status, ssh_url = process_plan(config, plan)
        if status != STATUS_OK:
            return status

        status, repo = get_repo_by_plan(plan["path_with_namespace"], repos)
        if status == STATUS_NOT_FOUND:
            return
        repo['codehub_sshUrl'] = ssh_url
        target_repo_info[repo['path_with_namespace']] = repo

    with open(FILE_TARGET_REPO_INFO, 'w') as f:
        json.dump(target_repo_info, f, indent=4)
    log.logger.info('create_group_and_repos end')
    return STATUS_OK


def push_to_target(config, plans):
    log.logger.info('push_repos start')
    with open(FILE_TARGET_REPO_INFO, 'r') as f:
        repos = json.load(f)
    for r in repos:
        repo = repos[r]
        name = repo["name"]
        dir_name = get_repo_dir(repo)

        status = exec_cmd("git config remote.origin.url", repo['codehub_sshUrl'], dir_name + "/" + name +
".git")
        if status != STATUS_OK:
            log.logger.error("%s git config failed" % name)
            return

        status = exec_cmd("git push --mirror -f", "", dir_name + "/" + name + ".git")
        if status != STATUS_OK:
            log.logger.error("%s git push failed" % name)
            return
    log.logger.info('push_repos end')


def main():
    with open(FILE_CONFIG, 'r') as f:
        config = json.load(f)
    # read plan
    status, plans = read_migrate_plan()
    if status != STATUS_OK:
        return
    # 获取自建gitlab仓库列表，结果输出到FILE_SOURCE_REPO_INFO文件中
    if repo_info_from_source(config) != STATUS_OK:
        return
    # clone仓库到本地
    status = clone_from_source(config, plans)
    if status != STATUS_OK:
        return

    # 调用devcloud接口创建仓库，并记录devcloud仓库地址到FILE_SOURCE_REPO_INFO中
    if create_group_and_repos(config, plans) != STATUS_OK:
        return

    # 推送时使用ssh方式推送，请提前在codehub服务配置ssh key
    push_to_target(config, plans)


if __name__ == '__main__':
    main()
```

步骤9 执行如下命令，启动脚本并完成代码仓的批量迁移。


```
python migrate_to_repo.py
```

----结束

