

CodeArts Repo

最佳实践

文档版本 01
发布日期 2025-07-15



版权所有 © 华为云计算技术有限公司 2025。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HUAWEI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 CodeArts Repo 最佳实践汇总.....	1
2 企业如何进行仓库成员及权限管理.....	2
3 批量迁移 GitLab 内网仓库到 CodeArts Repo.....	6
4 如何批量将本地仓库导入 CodeArts Repo.....	15
5 代码托管安全配置概述.....	27
6 配置 HTTPS 密码实践.....	29
7 配置保护分支规则最佳实践.....	33
7.1 配置保护分支规则实践的概述.....	33
7.2 配置保护分支规则实施步骤.....	35

1 CodeArts Repo 最佳实践汇总

表 1-1 常用最佳实践

实践	描述
批量迁移GitLab内网仓库到CodeArts Repo	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。
如何批量将外部仓库导入CodeArts Repo	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。
代码托管安全配置概述	CodeArts Repo通过提供访问令牌、部署密钥、保护分支管理等能力，为代码资产安全保驾护航。
企业如何进行仓库成员及权限管理	CodeArts Repo涉及项目级、代码组和仓库级权限，该案例以一个企业项目开发为例，介绍如何为企业不同角色配置CodeArts Repo的权限。
配置HTTPS密码	当用户需要将代码推送到代码托管仓库或从代码托管仓库拉取代码时，代码托管仓库需要验证用户的身份与权限，HTTPS密码是对代码托管进行远程访问的一种身份验证方式，并且用户只需要设置一次HTTPS密码。
配置项目级保护分支规则	在代码协作开发过程中，配置保护分支规则可以确保代码质量、维护代码分支安全性和提升团队的协作效率，最终实现高效、安全且可持续的代码交付。

2 企业如何进行仓库成员及权限管理

背景介绍

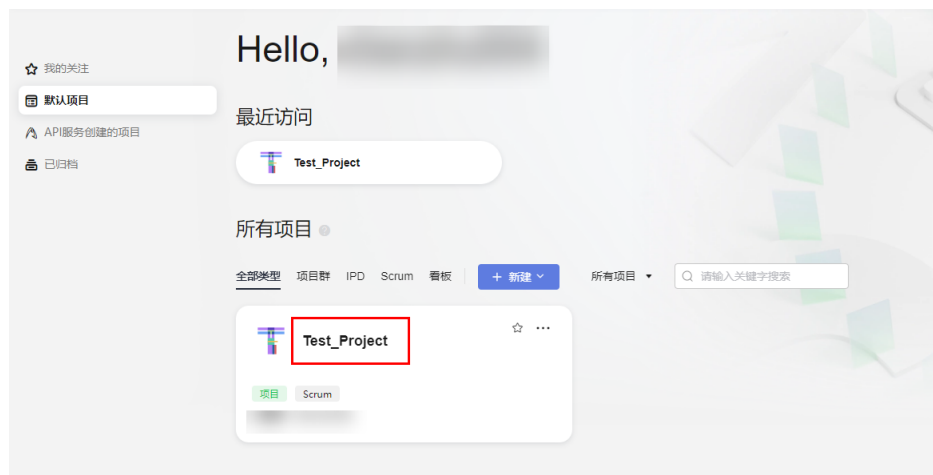
企业有30个成员参与软件产品线开发，其中有两个团队负责两个产品开发，分别对应CodeArts两个项目。其中“Test_Project”项目计划有20个团队成员，包括：1名项目经理，1名产品经理，1名测试经理，3名Committer，4名测试人员，10名开发人员。

创建项目

单击右上角头像，选择“租户设置 > 通用设置 > 设置项目创建者”，勾选“设置部分成员可以创建项目（拥有Tenant Administrator和CodeArts Administrator角色的用户默认能够创建项目，不受该设置的限制）”，在该页面给小明授予创建项目的权限。



小明得到项目创建的权限后，进入软件开发生产线首页，单击“新建”，创建了Scrum类型的项目“Test_Project”，小明默认拥有了“项目经理”权限。

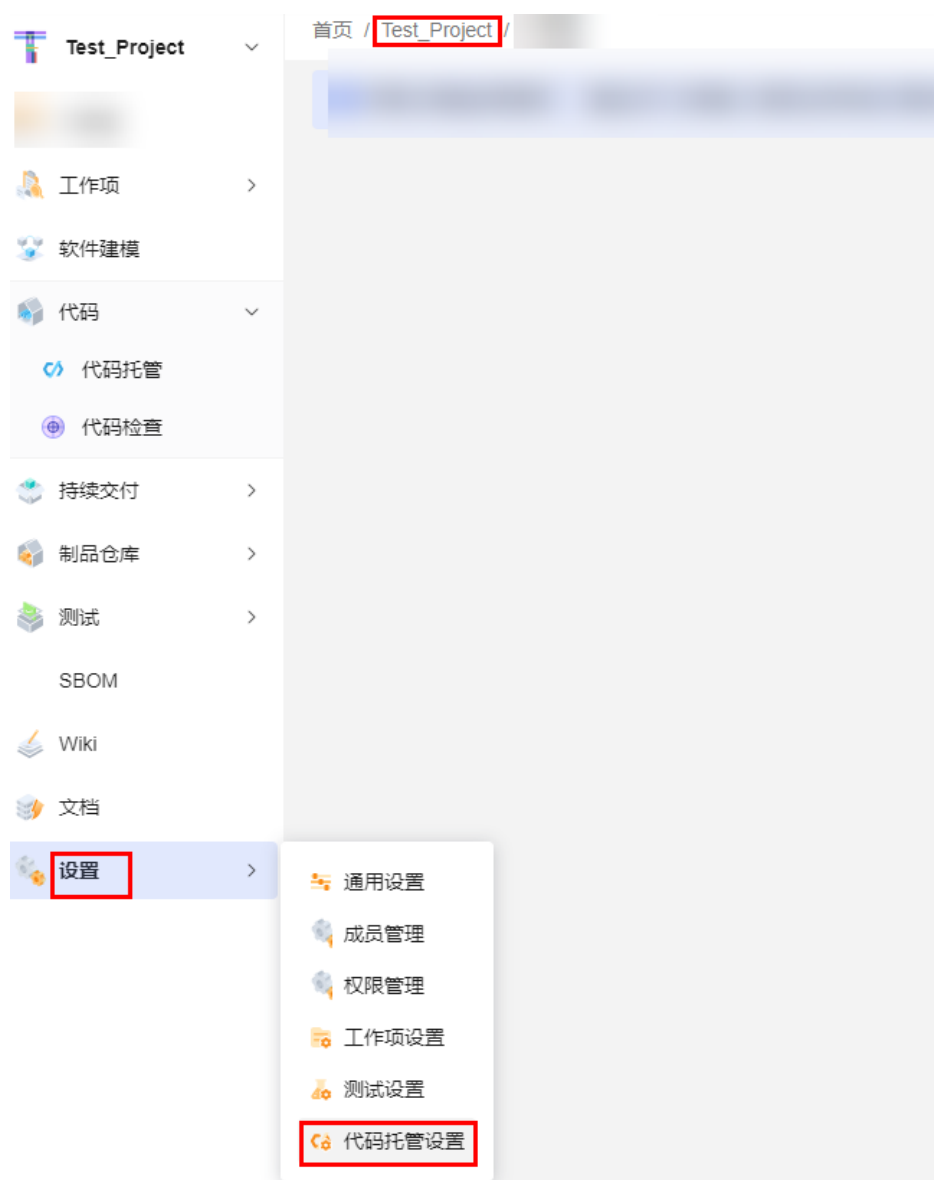


管理项目成员

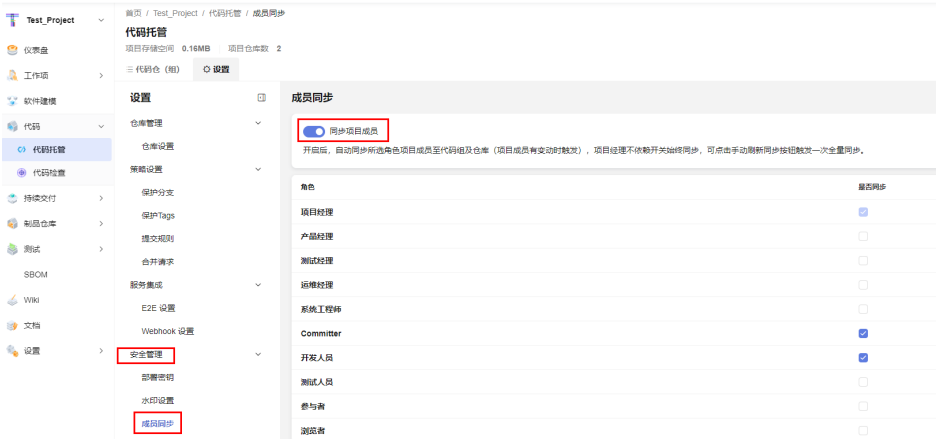
代码组及仓库成员来源必须是已添加到项目的项目成员，作为项目经理，小明需要提前配置项目成员并为他们设置相应的角色权限。

小明为方便统一管理项目成员，按照如下步骤把项目成员自动同步到项目下的代码组和代码仓。

步骤1 小明进入项目“Test_Project”，单击左侧导航栏“设置 > 代码托管设置”。



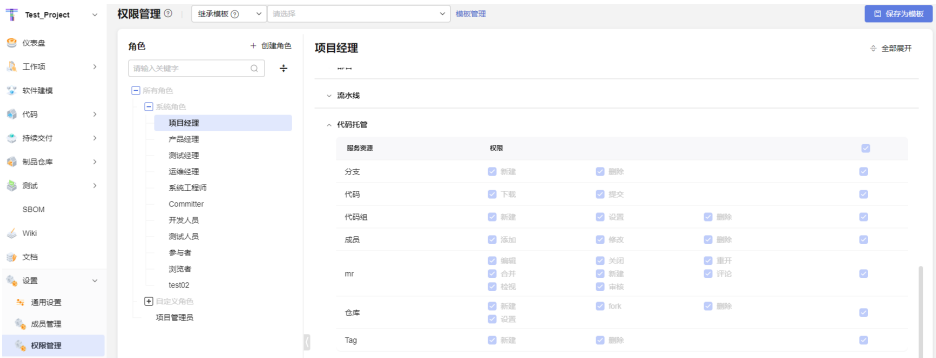
步骤2 小明继续选择“安全管理 > 成员同步”，打开“同步项目成员”开关，以此把团队成员添加进项目下的代码组和仓库中。



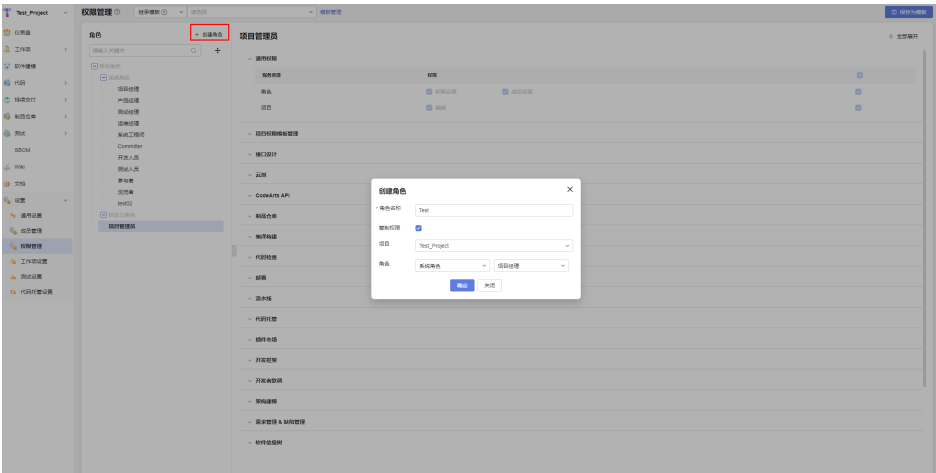
----结束

管理项目成员权限

管理员小明具有“设置”权限，为不同角色的用户配置权限。例如下图所示，小明作为项目经理，默认具有代码托管的所有权限，且支持取消任意权限，小明可以任意配置自己的权限。



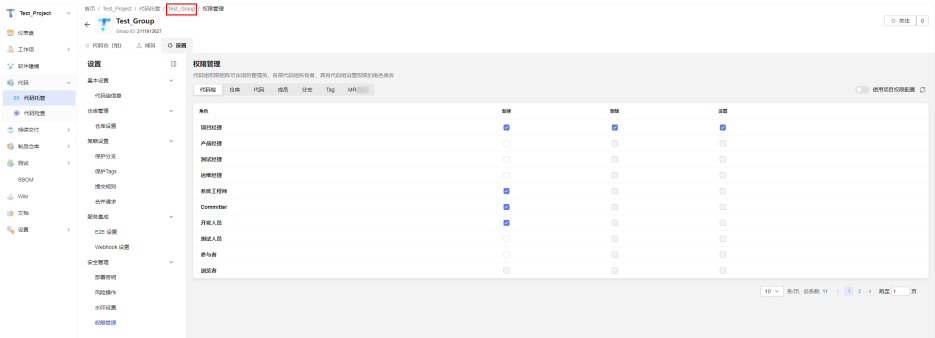
小明新增自定义角色“Test”，例如下图所示，单击“创建角色”，可复制其他项目下某角色的权限。



管理代码组成员权限

该项目下涉及多个代码仓，为方便管理，小明创建了代码组“Test_Group”，并进入代码组权限配置，考虑到代码安全，小明不继承“项目权限配置”，单独为代码组配置权限。

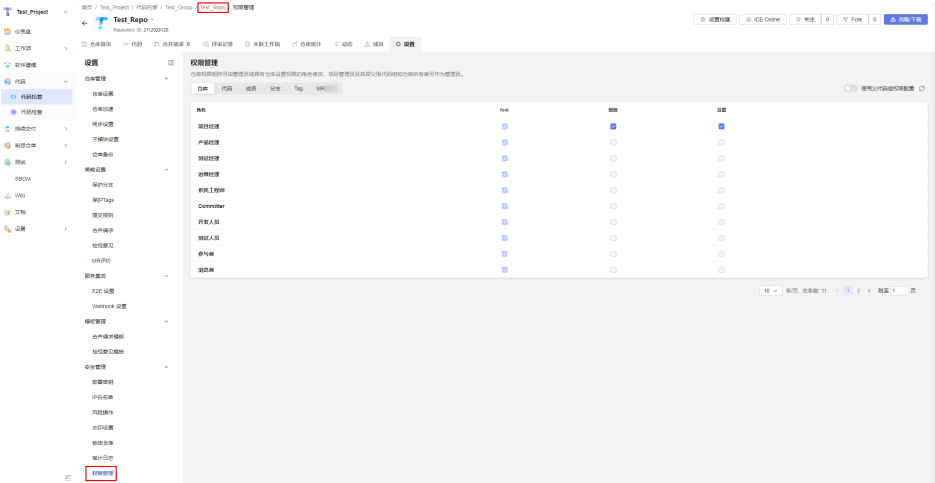
例如下图所示，☒ 表示具有权限，可以取消当前权限。☐ 表示没有权限，可添加该权限。☐ 表示没有权限，也不可以配置当前权限。☒ 表示有权限，且不可以取消。



管理代码仓库成员权限

由于开发人员负责不同的代码仓开发，小明未“使用父代码组权限配置”，而是进入要配置权限的代码仓，为不同开发配置权限。例如下图所示，小明进入代码仓“Test_Repo”为各个角色配置权限。

☒ 表示具有权限，可以取消当前权限。☐ 表示没有权限，可添加该权限。☐ 表示没有权限，也不可以配置当前权限。☒ 表示有权限，且不可以取消。



3 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ED25519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?simple=true",
  "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控制台](#)，单击，选择区域，单击“立即使用”。

步骤7 在CodeArts首页单击“新建项目”，选择“Scrum”。如果首页中显示“您还没有项目”，则单击“Scrum”。创建项目后请保存您的项目ID。

步骤8 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，如果没有提前在页面创建，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9 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

    # 调用接口创建仓库, 并记录仓库地址到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CodeArts Repo服务配置ssh key
    push_to_target(config, plans)
```

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

步骤10 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

4 如何批量将本地仓库导入 CodeArts Repo

背景介绍

CodeArts Repo现有导仓能力只支持从公网导入单个仓库，缺少客户本地代码仓往Repo迁移的快速方案，因此提供批量迁移本地代码仓到Repo的脚本。

前置准备

步骤1 进入[Python官网](#)下载并安装Python3。

步骤2 调试[获取IAM用户Token\(使用密码\)](#)接口，通过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。

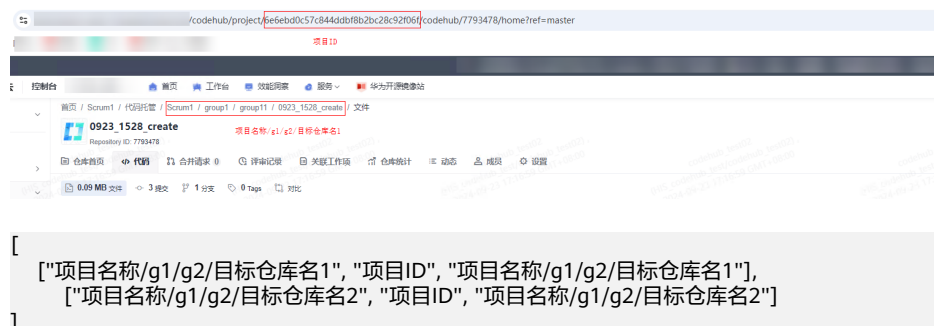
步骤3 用获取到的用户Token配置“config.json”文件。其中，repo_api_prefix是CodeArts Repo 的openAPI地址。“ ”

```
{
  "repo_api_prefix": "https://{open_api}",
  "x_auth_token": "用户Token"
}
```

步骤4 登录[CodeArts控制台](#)，单击，选择区域，单击“立即使用”。

步骤5 在CodeArts首页单击“新建项目”，选择“Scrum”。如果首页中显示“您还没有项目”，则单击“Scrum”。创建项目后请保存您的项目ID。

步骤6 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表示代码组路径，可以参考[说明](#)进行创建。图1展示了“项目ID”和“项目名称/g1/g2/目标仓库名1”在Repo界面的获取方式。



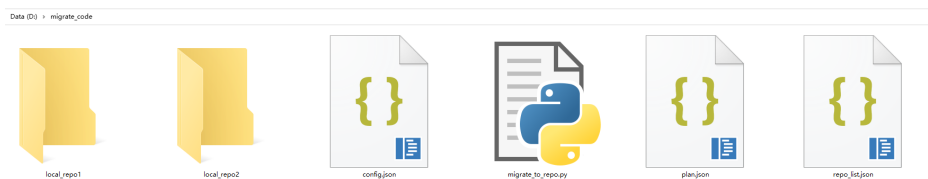
```
[
  ["项目名称/g1/g2/目标仓库名1", "项目ID", "项目名称/g1/g2/目标仓库名1"],
  ["项目名称/g1/g2/目标仓库名2", "项目ID", "项目名称/g1/g2/目标仓库名2"]
]
```

📖 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7 添加一个配置文件repo_list.json。

其中，“local_dir”表示把本地某个目录的代码文件传到目标仓库的路径，您上传的必须是一个完整的Git仓，并且需要与migrate_to_repo.py在同一级目录。如下图所示，local_repo1和local_repo2表示要上传的本地Git仓，即“local_dir”和“local_dir”的值分别为“local_repo1”和“local_repo2”。



如下代码示例中，g1/g2表示代码组路径，对应项目id可以参考[项目ID获取方式](#)获取，请根据实际情况填写。

```
[
  {
    "id": "对应项目的id",
    "namespace": "项目名称/g1/g2/目标仓库名1",
    "local_dir": "Git仓的本地路径1"
  },
  {
    "id": "对应项目的id",
    "namespace": "项目名称/g1/g2/目标仓库名2",
    "local_dir": "Git仓的本地路径2"
  }
]
```

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
import argparse
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_SOURCE_REPO_INFO_TXT = 'source_repos.txt'
FILE_TARGET_REPO_INFO = 'target_repos.json'
FILE_CONFIG = 'config.json'
FILE_PLAN = 'plan.json'
FILE_LOG = 'migrate.log'
```

```
FILE_REPO_LIST = 'repo_list.json'
X_AUTH_TOKEN = 'x-auth-token'
LOG_FORMAT = '%(asctime)s - %(pathname)s[line:%(lineno)d] - %(levelname)s: %(message)s'

class Logger(object):
    def __init__(self, filename):
        format_str = logging.Formatter(LOG_FORMAT)
        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
    except Exception as e:
        log.logger.info("request failed, e is %s", e)
        return STATUS_INTERNAL_SERVER_ERROR, "request failed"
    content = r.read().decode(CODE_UTF8)
    return STATUS_OK, content

def read_migrate_plan():
    log.logger.info('read_migrate_plan start')
    try:
        with open(FILE_PLAN, 'r') as f:
            migrate_plans = json.load(f)
    except Exception as e:
        log.logger.info("load plan.json, e is %s", e)
        return STATUS_INTERNAL_SERVER_ERROR, []
    plans = []
    for m_plan in migrate_plans:
        if len(m_plan) != 3:
            log.logger.error(
                "please check plan.json file"
            )
            return STATUS_INTERNAL_SERVER_ERROR, []
```

```
namespace = m_plan[2].split("/")
namespace_len = len(namespace)
if namespace_len < 1 or namespace_len > 4:
    log.logger.error("group level support 0 to 3")
    return STATUS_INTERNAL_SERVER_ERROR, []
plan = {
    "path_with_namespace": m_plan[0],
    "project_id": m_plan[1],
    "groups": namespace[0:namespace_len - 1],
    "repo_name": namespace[namespace_len - 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(source_host_url, private_token, protocol):
    log.logger.info('get repos by api start')
    headers = {'PRIVATE-TOKEN': private_token}
    url = 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 = {
                'id': repo['id'],
                'name': repo['name'],
                'path_with_namespace': namespace
            }
            if protocol == "ssh":
                repo_info["clone_url"] = repo["ssh_url_to_repo"]
            else:
                repo_info["clone_url"] = repo["http_url_to_repo"]
            data[namespace] = repo_info
        if len(repos) < per_page:
            break
        page = page + 1
    try:
        with open(FILE_SOURCE_REPO_INFO, 'w') as f:
            json.dump(data, f, indent=4)
    except Exception as e:
        log.logger.info("load source_repos.json, e is %s", e)
        return STATUS_INTERNAL_SERVER_ERROR
    log.logger.info('get_repos end with %s' % len(data))
    return STATUS_OK

def repo_info_from_file():
    log.logger.info('get repos by file start')
    data = {}
    try:
        with open(FILE_REPO_LIST, 'r') as f:
            repos = json.load(f)
    except Exception as e:
        log.logger.info("load repo_list.json, e is %s", e)
```

```
        return STATUS_INTERNAL_SERVER_ERROR
    for index, repo in enumerate(repos):
        if repo.get("id") is None:
            log.logger.error("line format not match id")
        if repo.get("namespace") is None:
            log.logger.error("line format not match namespace")
            return STATUS_INTERNAL_SERVER_ERROR
        if repo.get("local_dir") is None:
            log.logger.error("line format not match local_dir ")
            return STATUS_INTERNAL_SERVER_ERROR
        if not os.path.exists(repo.get("local_dir")):
            log.logger.warning("local dir %s non-existent" % repo.get("local_dir"))
            continue
        namespace = repo.get("namespace")
        repo_info = {
            'id': repo.get("id"),
            'name': namespace.split("/")[-1],
            'path_with_namespace': namespace,
            'clone_url': "",
            'local_dir': repo.get("local_dir")
        }
        data[namespace] = repo_info
    try:
        with open(FILE_SOURCE_REPO_INFO, 'w') as f:
            json.dump(data, f, indent=4)
    except Exception as e:
        log.logger.info("load source_repos.json, e is %s", e)
        return STATUS_INTERNAL_SERVER_ERROR
    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(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['clone_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)
        print(url_with_page, status, content)
        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(
            '%s skip: %s already exist' % (
                "create_group_and_repos",
                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():
    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 push_to_target_with_local():
    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]
        dir_name = repo["local_dir"]
        status = exec_cmd(
            "git config remote.origin.url",
            repo['codehub_sshUrl'],
            dir_name
        )
        if status != STATUS_OK:
            log.logger.error("%s git config failed" % dir_name)
            return
        status = exec_cmd("git push --all -f", "", dir_name)
        if status != STATUS_OK:
            log.logger.error("%s git push failed" % dir_name)
            return
    log.logger.info('push_repos end')

def get_args_from_command_line(args_list):
    # 解析命令行参数
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '-p',
        '--protocol',
        dest='protocol',
        default="SSH",
        choices=['SSH', 'HTTP', 'ssh', 'http'],
        required=False,
        help='protocol specified for clone or push'
    )
    parser.add_argument(
        '-m',
        '--mode',
        dest='mode',
        default="FILE",
        choices=['FILE', 'file'],
        required=False,
```

```
        help='import mode'
    )
    return parser.parse_args(args_list)

if __name__ == '__main__':
    if not os.path.exists(FILE_CONFIG):
        log.logger.info("config.json must be present")
        exit(1)
    if not os.path.exists(FILE_PLAN):
        log.logger.info("plan.json must be present")
        exit(1)

    # 获取映射, 仓库信息和namespace
    status, plans = read_migrate_plan()
    if status != STATUS_OK:
        log.logger.info("load plan.json failed")
        exit(1)

    # 加载配置文件
    try:
        with open(FILE_CONFIG, 'r') as f:
            config = json.load(f)
    except Exception as e:
        log.logger.info("load config.json, e is %s", e)
        exit(1)
    if config.get("repo_api_prefix") is None:
        log.logger.error("config.json not match repo_api_prefix")
        exit(1)
    if config.get("x_auth_token") is None:
        log.logger.error("config.json not match x_auth_token")
        exit(1)

    args = get_args_from_command_line(None)
    protocol = args.protocol
    mode = args.mode
    if mode.lower() == "api":
        log.logger.error("not allow mode is api")
        exit(1)
    if config.get("source_host_url") is None:
        log.logger.error("config.json not match source_host_url")
        exit(1)
    if config.get("private_token") is None:
        log.logger.error("config.json not match private_token")
        exit(1)
    if repo_info_from_source(
        config["source_host_url"],
        config["private_token"],
        protocol.lower()
    ) != STATUS_OK:
        exit(1)
    try:
        # clone仓库到本地
        status = clone_from_source(plans)
        if status != STATUS_OK:
            exit(1)
    except Exception as e:
        log.logger.info("clone_from_source fail, e is %s", e)
        exit(1)
    else:
        if repo_info_from_file() != STATUS_OK:
            exit(1)

    try:
        if create_group_and_repos(config, plans) != STATUS_OK:
            exit(1)
    except Exception as e:
        log.logger.info("create_group_and_repos fail, e is %s", e)
        exit(1)
```

```
try:
    if mode.lower() == "api":
        push_to_target()
    else:
        push_to_target_with_local()
except Exception as e:
    log.logger.info("push_to_target fail, e is %s", e)
    exit(1)
```

----结束

配置访问 CodeArts Repo 的 SSH 公钥

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ED25519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公钥到剪贴板。请根据您的操作系统，选择相应的执行命令，将SSH公钥复制到您的剪贴板。

- **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

开始批量迁移

步骤1 执行如下命令，查看脚本参数。

```
python migrate_to_repo.py -h

usage: migrate_to_repo.py [-h] [-p {SSH,HTTP,ssh,http}]
                        [-m {API,FILE,api,file}]

optional arguments:
  -h, --help            show this help message and exit
  -p {SSH,HTTP,ssh,http}, --protocol {SSH,HTTP,ssh,http}
                        protocol specified for clone or push
  -m {API,FILE,api,file}, --mode {API,FILE,api,file}
                        import mode

# 参数说明
# -p 协议，默认是SSH协议，可选为SSH/ssh/HTTP/http
```

----结束

5 代码托管安全配置概述

代码安全

代码托管服务（CodeArts Repo）通过提供访问令牌、部署密钥、保护分支管理等能力，为代码资产安全保驾护航。

表 5-1 代码安全

安全配置	说明	配置建议	参考资料
访问令牌	CodeArts Repo为每个用户提供生成访问令牌功能，令牌仅在生成时显示一次，并且可以设置有效期，默认有效期1个月，最长有效期为1年。	建议在授权个人仓库访问权限给第三方时，创建访问令牌并设置有效期，避免暴露账号密码。	设置访问令牌
部署密钥	CodeArts Repo支持为代码仓库添加部署密钥，通过部署密钥访问代码仓库时只有只读权限。	建议在构建等对代码仓库只读场景下，使用部署密钥克隆代码仓，尽可能避免密钥泄露等影响代码仓安全。	配置部署密钥
保护分支管理	在代码仓库中可以设置保护分支，防止此分支被修改或删除。	建议将主干分支设置为保护分支，仅能通过合并请求的方式将代码合入主干分支，且禁止任何人对保护分支做强制推送。	配置保护分支

安全配置	说明	配置建议	参考资料
可见范围管理	CodeArts Repo支持对代码仓库设置以下可见范围。 - 私有（仓库仅对仓库成员可见，仓库成员可读写和访问仓库） - 公开 - 项目内成员只读 - 租户内成员只读 - 所有访客只读	建议结合业务需要，在新建仓库时选择相应的可见范围，或为已创建的代码仓库调整可见范围。 管理员也可以根据需要设置是否可以创建可见范围为“公开”的代码仓库。	- 设置仓库可见范围请参考新建仓库。 - 管理员设置是否创建可见范围为“公开”的代码仓库请调整仓库公开性。
提交规则管理	CodeArts Repo支持为代码仓库设置提交规则，用户可以选择服务提供的提交规则，也可以自定义提交规则。	建议结合业务需要，为每个仓库设置提交规则，防止代码被随意修改。	配置提交规则

6 配置 HTTPS 密码实践

背景介绍

如下图所示，项目的研发小明拥有了新的子账号，需要在此账号下开发项目“Test_Project”的“Test_Repo”代码仓。



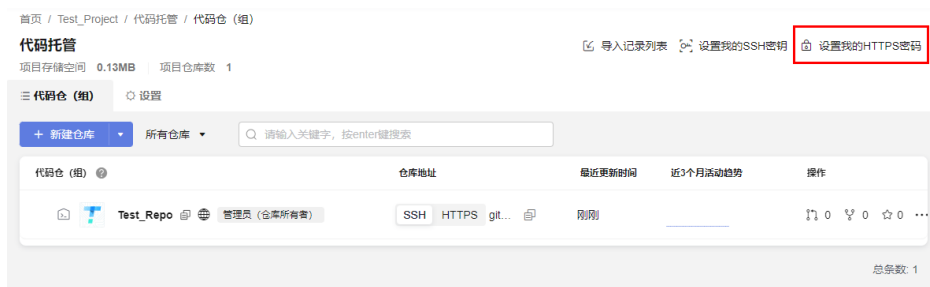
小明将代码推送到代码托管仓库或从代码托管仓库拉取代码时，代码托管仓库需要验证用户的身份与权限，HTTPS密码是对代码托管进行远程访问的一种身份验证方式，本案例会介绍小明如何初次配置HTTPS密码和修改HTTPS密码，并通过Windows从Repo克隆代码文件、把代码文件上传到Repo。

前置准备

开发需要在本地配置好Git客户端，安装的版本号需要大于2.30版本，并配置好用户名和邮箱，具体配置方式可参考[Git客户端安装与配置](#)。

首次配置 HTTPS 密码

步骤1 小明首次[登录代码托管首页](#)，单击页面右上角“设置我的HTTPS密码”。



步骤2 默认“账号”是小明的“租户名/IAM用户名”，“密码”是小明账号的登录密码。为了保障信息安全，小明选择“自行设置密码”，重置并保存新密码“RepoNewPassword”。



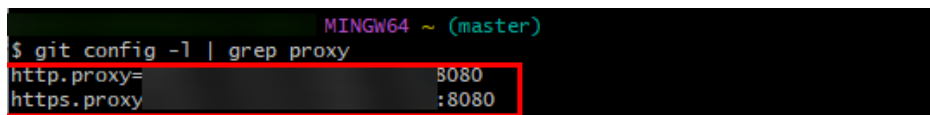
步骤3 右上角弹框提示小明成功自行设置密码，并自动跳转到“自行设置密码”页面，小明复制并在本地保存了“用户名”和“密码”。



步骤4 小明本地电脑配置过代理，于是小明执行如下命令查看已经配置的Git代理。

```
git config -l | grep proxy
```

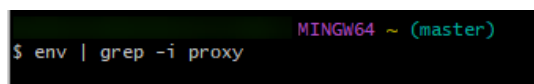
回显表示小明配置了Git的HTTP和HTTPS代理。



步骤5 小明继续执行如下命令查看环境变量的代理。

```
env | grep -i proxy
```

回显为空，表示小明没有配置环境变量的代理。



步骤6 小明继续执行如下命令，全局取消Git的HTTP和HTTPS代理。

```
git config --global --unset http.proxy
git config --global --unset https.proxy
```

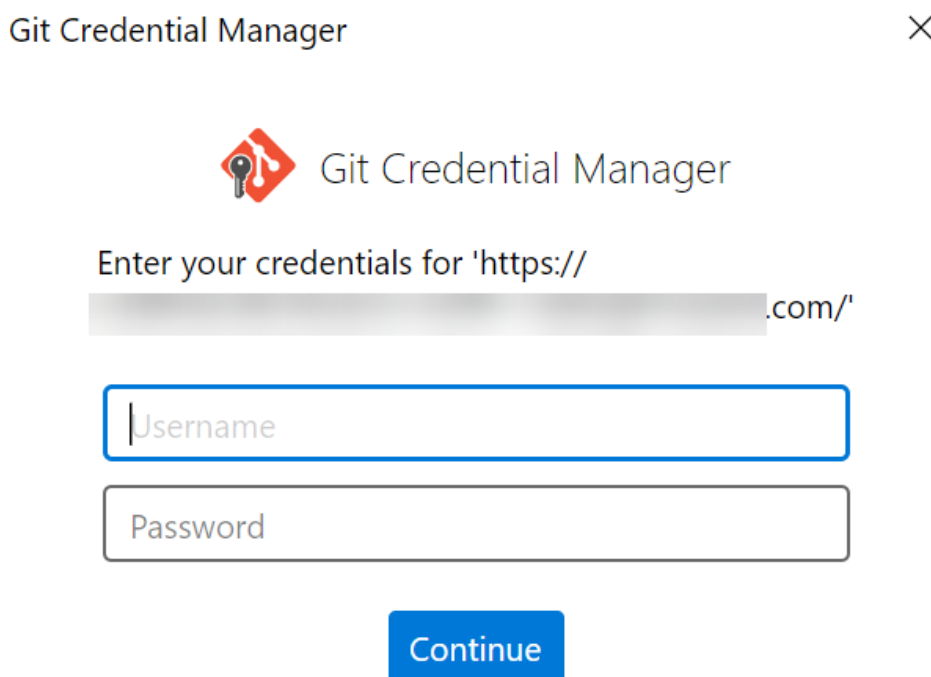
步骤7 小明复制HTTPS地址，克隆代码仓到本地进行开发。



步骤8 小明执行如下命令，验证设置的HTTPS密码是否生效，其中https://example.git为小明要克隆的代码仓地址。

```
git clone https://example.git
```

步骤9 回显如下图所示，小明输入**步骤3**复制保存的账号和密码。



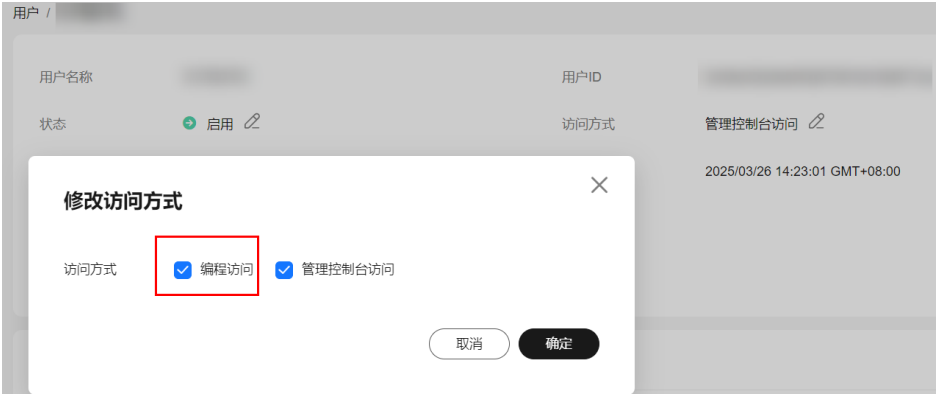
步骤10 小明输入账号密码后，克隆仓库失败，报错“remote: <CH.00905401> HTTP Basic: Access denied. Request-id is sqIEbze60W. fatal: Authentication failed for 'https://example.com/Test_Repo.git/'”，鉴权失败。

```
~Hmkn MINGW64 ~ (master)
$ git clone https://codehub.devcloud.com/Test_Repo.git
Cloning into 'Test_Repo'...
warning: ----- SECURITY WARNING -----
warning: | TLS certificate verification has been disabled! |
warning: -----
warning: HTTPS connections may not be secure. See https://aka.ms/ocm/tlsverify for more information.
remote: <CH.00905401> HTTP Basic: Access denied. Request-id is sqIEbze60W.
fatal: Authentication failed for 'https://codehub.devcloud.com/Test_Repo.git/'
```

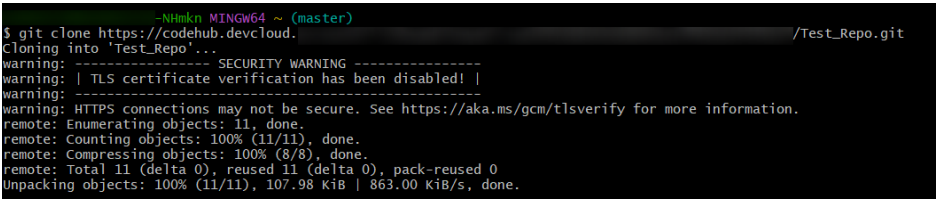
步骤11 小明联系租户管理员为小明增加权限。管理员登录后，单击右上角头像，选择“子账号管理”，如下图所示，进入控制台页面。



步骤12 单击“用户”，管理员找到小明账号，单击“操作”列的“安全设置”，为小明勾选了“编程访问”权限。管理员加完权限后，通知小明重新克隆代码仓。



步骤13 小明本地重新执行Git克隆命令，小明成功克隆了代码仓到本地，如下图所示。



----结束

7 配置保护分支规则最佳实践

7.1 配置保护分支规则实践的概述

在代码协作开发过程中，配置保护分支规则可以确保代码质量、维护代码分支安全性和提升团队的协作效率，最终实现高效、安全且可持续的代码交付。

本章节作为最佳实践的整体介绍，建议包含以下内容：

- [应用场景](#)
- [方案优势](#)
- [约束与限制](#)

应用场景

在团队协作开发过程中，没有保护分支规则可能会导致代码混乱，甚至会影响业务的交付效率，例如：

- 未经评审的代码直接合入主线分支，可能引入严重缺陷，导致线上故障频发，增加运维成本。
- 关键分支被强制推送覆盖，造成历史记录丢失，问题难以追溯，影响故障排查与合规审计。
- CI/CD流程未强制触发，低质量代码进入生产环境，可能会引发版本回滚或用户投诉。
- 多人协作缺乏规范，随意合并分支，导致代码冲突频发，影响版本的迭代进度。

通过在Repo配置保护分支规则，可针对性解决这些痛点：

- 核心分支（如master）：禁止任何角色直接推送代码到核心分支，确保只有合规代码才能上线，从而降低生产风险。
- 权限管控：限制关键分支的合并权限，可以避免合入不满足要求的代码，同时满足企业安全合规要求。

方案优势

通过在CodeArts Repo为关键代码分支设置保护分支规则，团队可以实现以下核心优势：

- 防止直接推送代码。通过合并请求提交变更，可以避免开发人员直接推送代码到保护分支，以此减少为评审代码引入的风险。
- 保障代码质量。通过指定角色成员合入合并请求，提升代码可维护性。
- 权限与合规控制。通过指定角色人员具有代码的合并权限，可以满足企业分级管控的需求，方便追踪代码的变更来源。
- 减少冲突与提升协作效率。通过禁止向保护分支直接推送代码，可以减少团队成员因不规范操作导致代码冲突，同时标准化流程可加速新成员上手。

约束与限制

如果您要为项目下的所有代码仓配置保护分支规则，您的系统角色需要是项目经理或者项目管理员，确认方式如下：

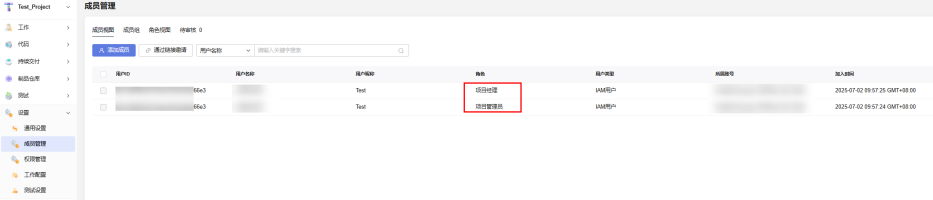
步骤1 如下图所示，进入要配置项目“Test_Project”的代码托管页面。

图 7-1 项目“Test_Project”的代码托管页面



步骤2 单击左侧导航栏“设置 > 成员管理”。如下图所示，当前用户是项目经理，也是项目管理员，所以当前用户具备配置项目保护分支规则的权限。

图 7-2 查看成员视图页面



----结束

7.2 配置保护分支规则实施步骤

步骤一：用户自检是否具备操作权限

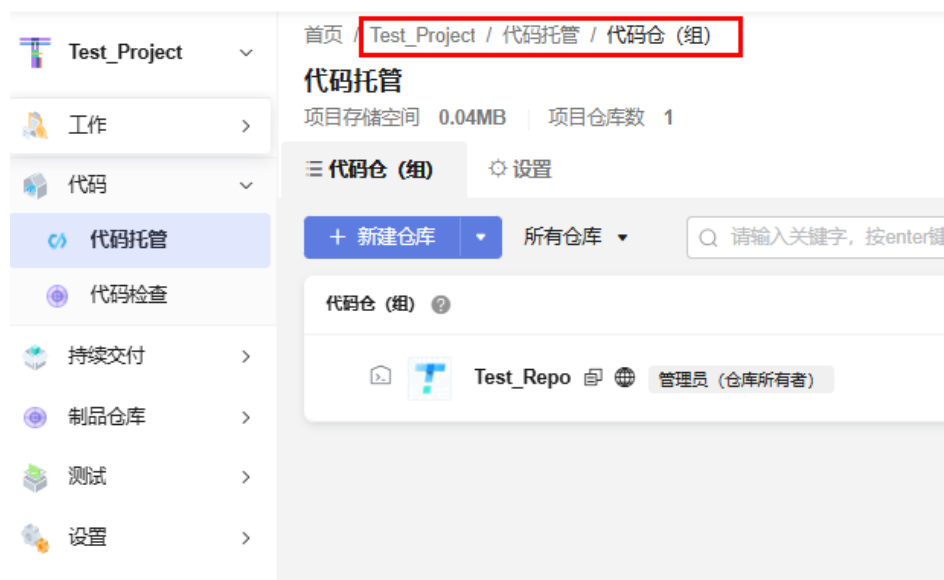
如果用户要配置项目级的代码分支规则，需要确认是否为项目经理或者项目管理员，请参考[约束与限制](#)确认。

如果用户不是项目经理或者项目管理员，可联系该项目的项目经理或者项目管理员配置保护分支规则。

步骤二：配置项目级的保护分支规则

步骤1 如下图所示，项目经理或者项目管理员，进入要配置项目“Test_Project”的代码托管页面。

图 7-3 项目“Test_Project”的代码托管页面



步骤2 在上方导航栏单击“设置 > 策略设置 > 保护分支”，单击“新建保护分支”。

图 7-4 进入“新建保护分支”页面

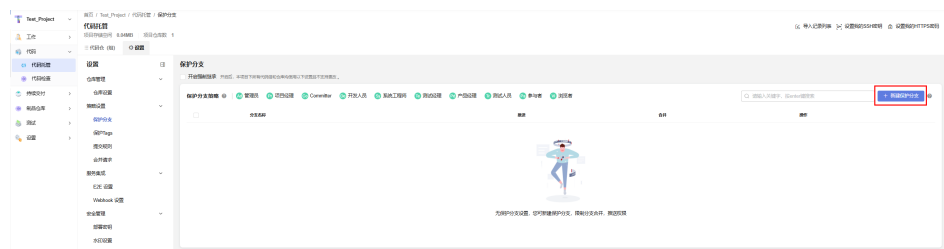
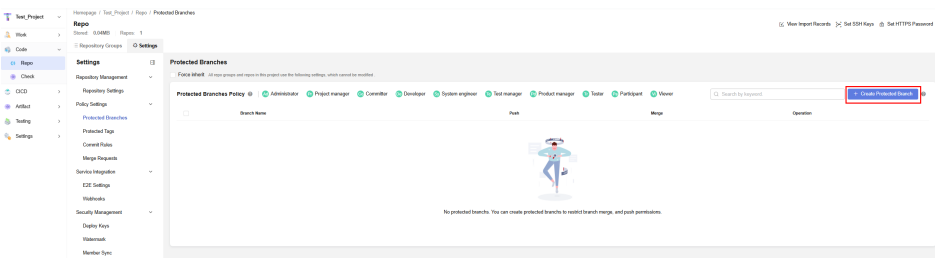
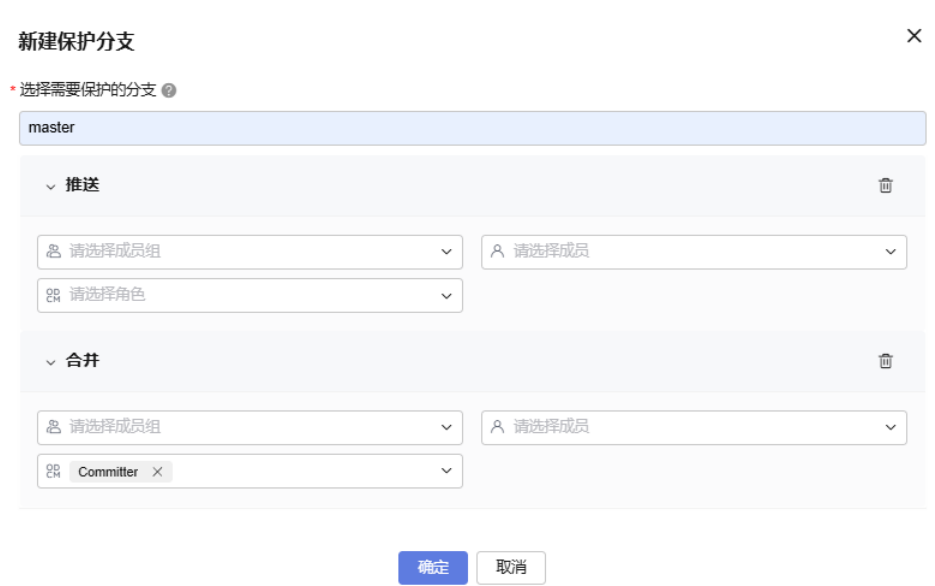


图 7-5 进入“新建保护分支”页面



步骤3 如下图所示，配置master为保护分支，任何人不能直接推送代码到master分支，仅Committer可合入代码到master分支。

图 7-6 配置保护分支规则



步骤4 如下图所示，勾选“开启强制继承”，表示该项目下的所有代码仓均使用该保护分支规则。

图 7-7 开启强制继承



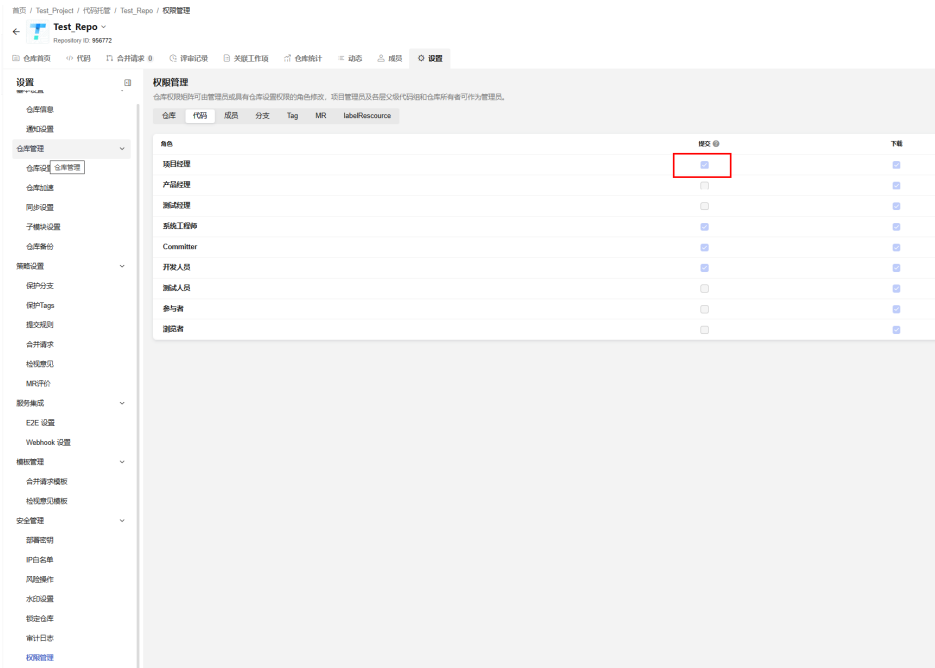
----结束

步骤三：项目经理推送文件到 master 分支失败演示

步骤1 任何代码仓成员推送代码文件到仓库之前，需要确认自己是否具备代码的“提交”权限。

如下图所示，项目经理进入要推送的代码仓“Test Repo”，单击上方导航栏“设置 > 安全管理 > 权限管理”，单击“代码”。

图 7-8 项目经理确认是否具备代码的“推送”权限



步骤2 项目经理确认具备“推送”权限后，如下图所示，单击右上角“克隆/下载 > 用HTTPS克隆”，单击  复制代码仓的HTTPS地址。

图 7-9 复制 HTTPS 地址



步骤3 如下图所示，执行Clone命令，克隆代码仓“Test_Repo”到本地。

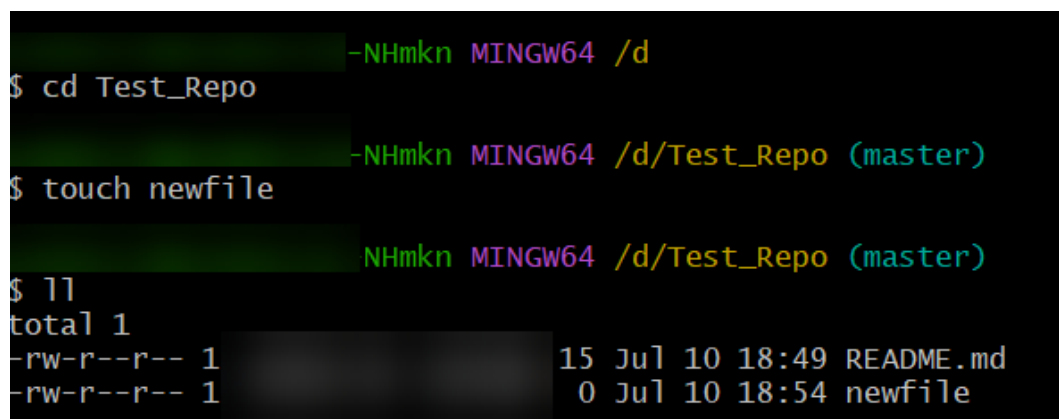
图 7-10 克隆代码仓“Test_Repo”到本地



步骤4 用户依次执行如下命令，进入代码仓Test_Repo路径下，创建新文件newfile，检查newfile文件是否执行成功。

```
cd Test_Repo
touch newfile
ll
```

图 7-11 新建 newfile 并检验是否成功新建文件



A terminal window showing the execution of commands to create a new file and list the directory contents. The prompt is MINGW64 /d. The user enters 'cd Test_Repo', then 'touch newfile', and finally 'll'. The output shows the file 'newfile' has been created with permissions '-rw-r--r--' and size '0'.

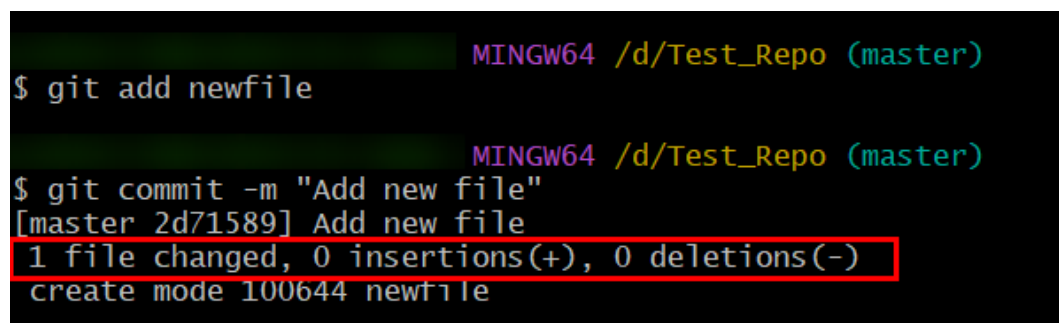
```
MINGW64 /d
$ cd Test_Repo
MINGW64 /d/Test_Repo (master)
$ touch newfile
MINGW64 /d/Test_Repo (master)
$ ll
total 1
-rw-r--r-- 1 15 Jul 10 18:49 README.md
-rw-r--r-- 1 0 Jul 10 18:54 newfile
```

步骤5 用户依次执行如下命令，执行git add命令，将新文件newtext添加到暂存区；执行git commit命令，表示将这个文件的更改提交到Git的历史中，并说明这次提交是“添加了一个新文件”。

```
git add newfile
git commit -m "Add new file"
```

如下图所示，出现红框所示的回显，表示此次提交成功。

图 7-12 提交文件 newfile 到 Git 的历史中



A terminal window showing the execution of git add and git commit commands. The prompt is MINGW64 /d/Test_Repo (master). The user enters 'git add newfile', then 'git commit -m "Add new file"'. The output shows the commit message '[master 2d71589] Add new file' and a summary of changes: '1 file changed, 0 insertions(+), 0 deletions(-)'. The summary line is highlighted with a red box.

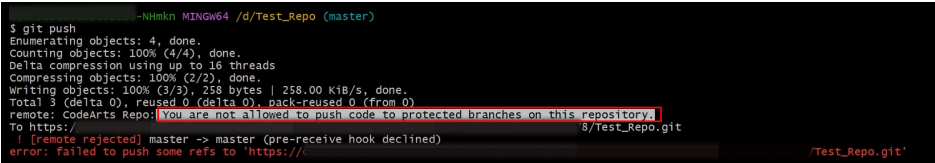
```
MINGW64 /d/Test_Repo (master)
$ git add newfile
MINGW64 /d/Test_Repo (master)
$ git commit -m "Add new file"
[master 2d71589] Add new file
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 newfile
```

步骤6 用户继续执行git push命令，将本地main分支上最新的提交推送到远程仓库的main分支。

```
git push
```

如下图所示，出现红框部分，表示推送文件到master分支失败，原因是master为保护分支，即使用户拥有代码的推送权限，也不能直接推送代码文件到代码仓。

图 7-13 git push 失败

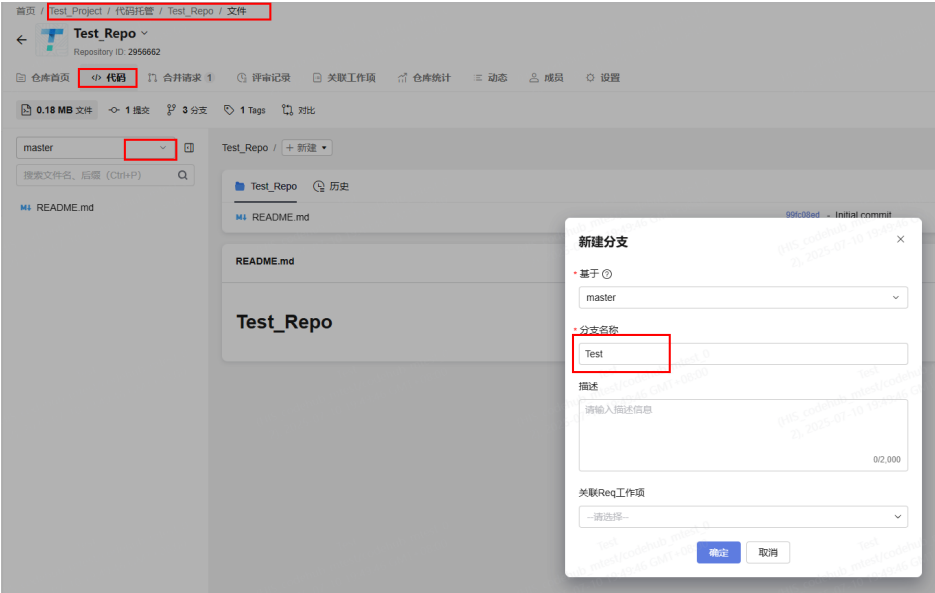


----结束

步骤四：演示 Committer 合入代码到 master 分支

步骤1 如下图所示，项目经理进入代码仓“Test Repo”的“代码”页签，下拉master分支，单击“新建分支”，填写分支名称为“Test”，单击“确定”。

图 7-14 新建 Test 分支



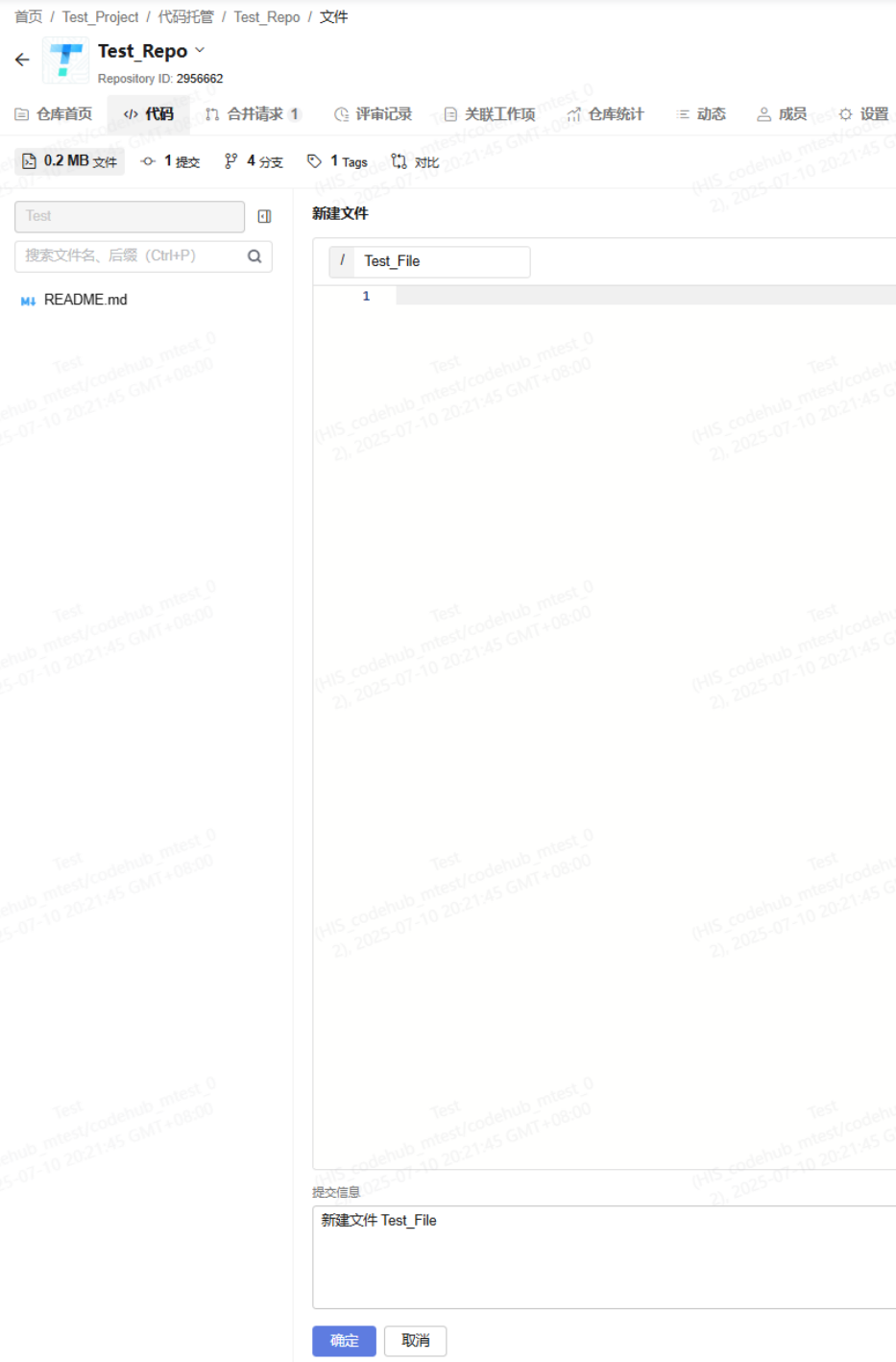
步骤2 如下图所示，在Test分支下，单击“新建文件”。

图 7-15 单击“新建文件”



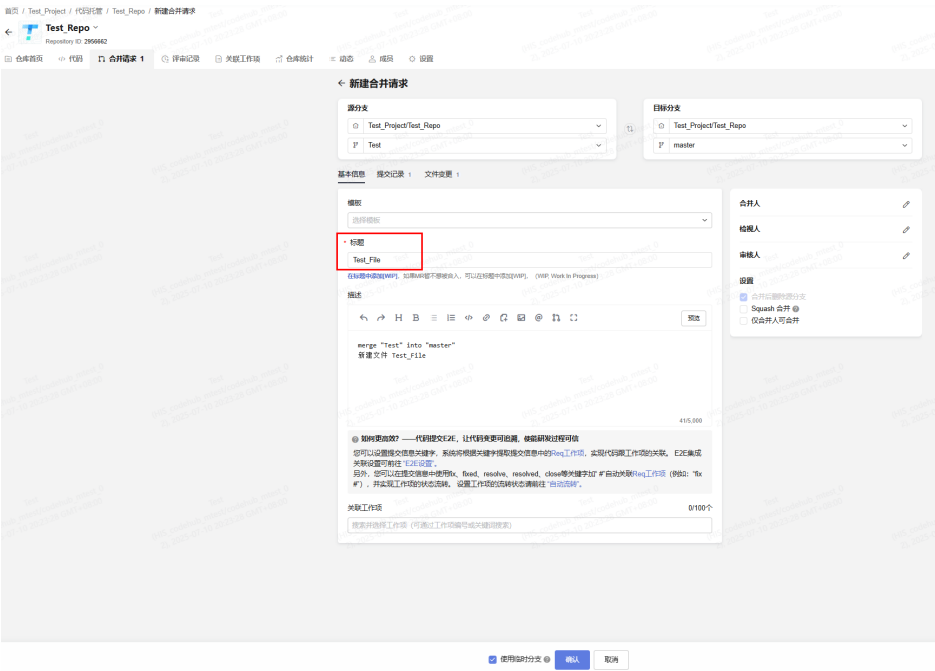
步骤3 如下图所示，填写文件名称为Test_File，单击“确定”。

图 7-16 新建文件 Test_File



步骤4 单击上方导航栏的“合并请求”，单击“新建合并请求”。填写“标题”，勾选“使用临时分支”，单击“确认”。

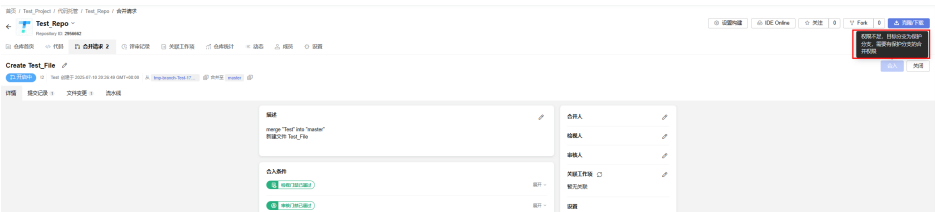
图 7-17 新建合并请求



步骤5 项目经理在合并请求页签，右上角“合入”按钮置灰，单击“合入”按钮，出现如下图所示的报错。

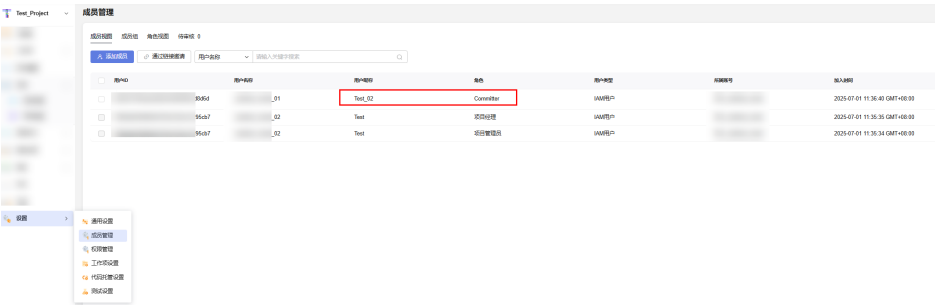
原因是master为保护分支，仅Committer角色可合入代码。

图 7-18 项目经理合入合并请求失败



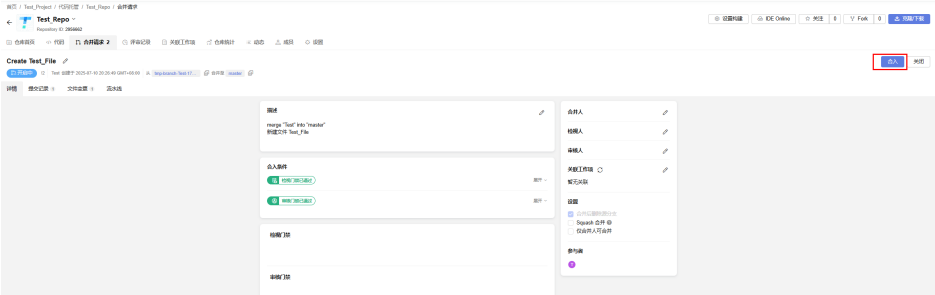
步骤6 如下图所示，Test_02为Committer角色，项目经理Test联系Test_02合入该合并请求。

图 7-19 确认 Committer 角色



步骤7 如下图所示，Test_02进入合并请求页面，合入了项目经理Test的合并请求。

图 7-20 Committer 合入合并请求



----结束