CodeArts Repo

Best Practices

Issue	01
Date	2025-07-22

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: https://www.huaweicloud.com/intl/en-us/

Contents

1 CodeArts Repo Best Practices

Table 1-1 Common best practices

Practice	Description
Migrating GitLab Intranet Repositories to CodeArts Repo in Batches	Currently, CodeArts Repo only allows for repository migrations between public networks. There is no quick solution for migrating repositories from a platform built on the customer's intranet to CodeArts Repo. Therefore, we provide a script for migrating repositories from an intranet platform to CodeArts Repo in batches.
Importing External Repositories to CodeArts Repo in Batches	Currently, CodeArts Repo only allows for repository migrations between public networks. There is no quick solution for migrating repositories from a platform built on the customer's intranet to CodeArts Repo. Therefore, we provide a script for migrating repositories from an intranet platform to CodeArts Repo in batches.
CodeArts Repo Security Configuration Overview	CodeArts Repo provides access tokens, deploy keys, and protected branches to safeguard your code assets.
Managing Repository Members and Permissions in an Enterprise	CodeArts Repo supports project-level, repository group-level, and repository-level permissions. This section uses project development as an example to describe how to configure CodeArts Repo permissions for different roles in an enterprise.
Configuring HTTPS Password	When you push code to or pull code from CodeArts Repo, the repository verifies your identity and permissions. HTTPS password is an identity authentication mode for remote access to CodeArts Repo. You only need to set this once.
Configuring Project-level Protected Branch Rules	Set up protected branch rules during collaborative coding to enhance code quality, safeguard branches, and boost teamwork for reliable, secure software delivery.

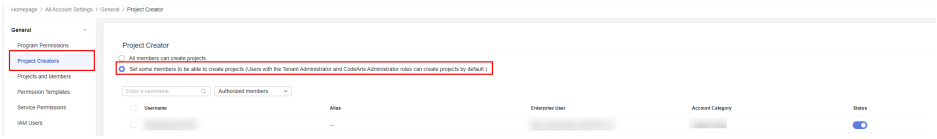
2 Managing Repository Members and Permissions in an Enterprise

Background

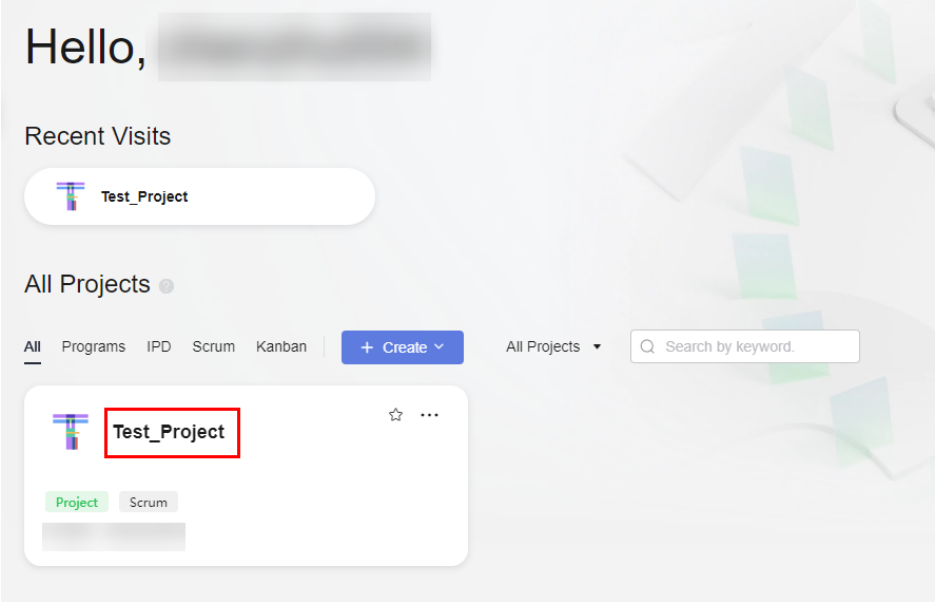
An enterprise has 30 members in two teams involved in software development. Each team develops a product and has a CodeArts project. **Test_Project** has 20 team members, including one project manager, one product manager, one test manager, three committers, four testers, and 10 developers.

Creating a Project

Click the profile picture in the upper right corner and choose **All Account Settings** > **General** > **Project Creators**. Select **Only specified members can create or maintain programs (By default, the organization system administrator can create projects.)** and grant the project creation permission to Sam.



Then, Sam goes to the CodeArts homepage, clicks **Create**, and creates a Scrum project named **Test_Project**. By default, Sam has project manager permissions.

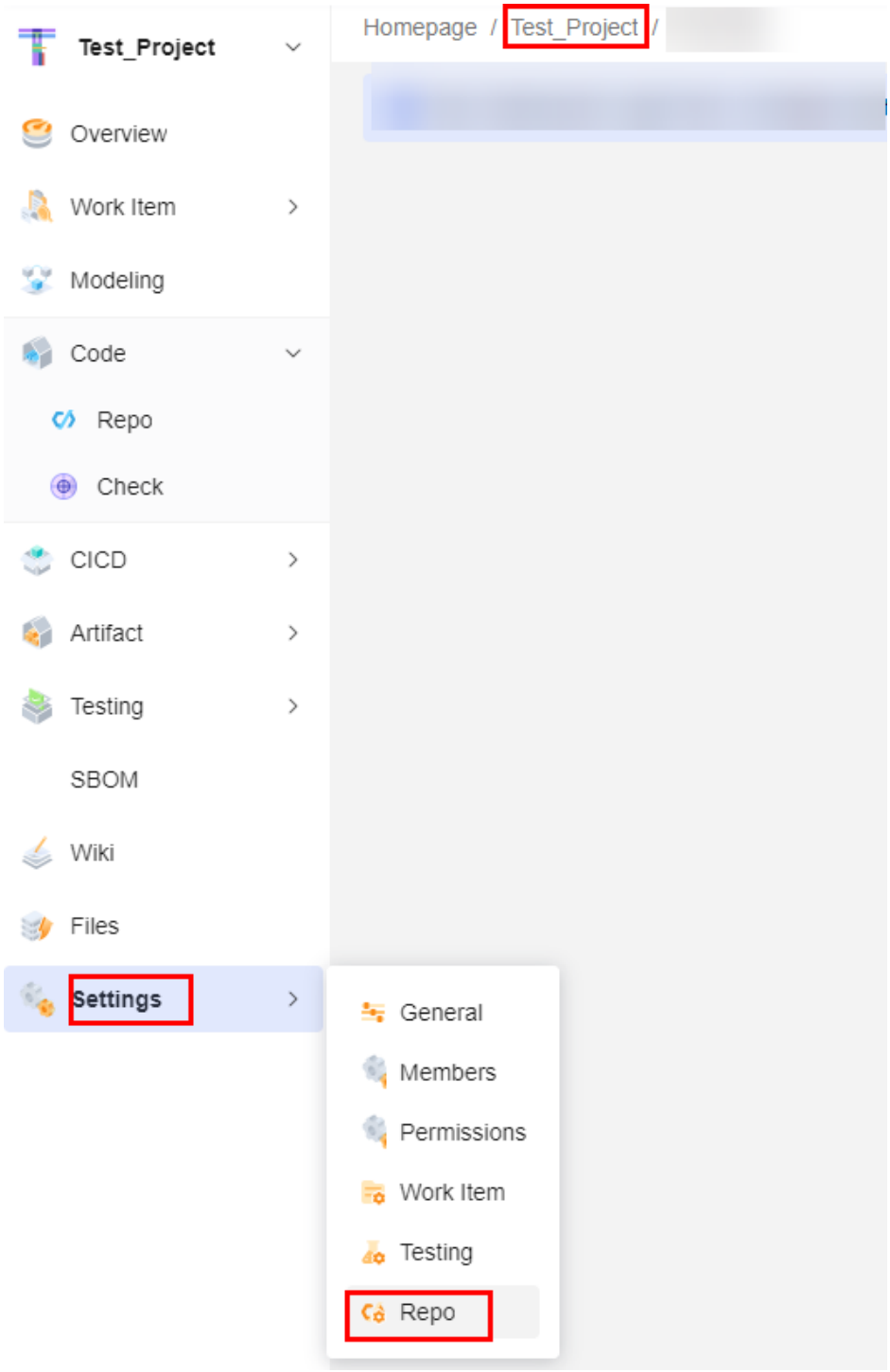


Managing Project Members

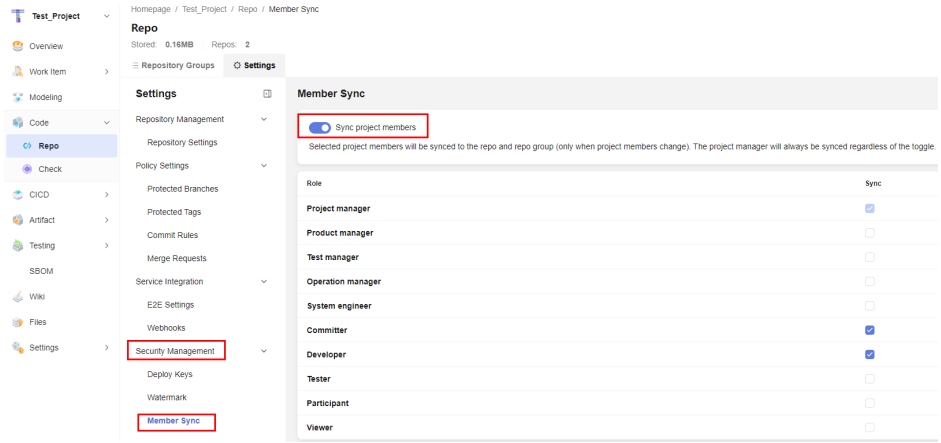
Repository groups and repository members must be project members added to the project. As the project manager, Sam configures project members and assigns them role permissions.

To manage project members, Sam syncs project members to repository groups and repositories in the project.

Step 1 He goes to the **Test_Project** project and chooses **Settings** > **Repo** from the navigation pane.



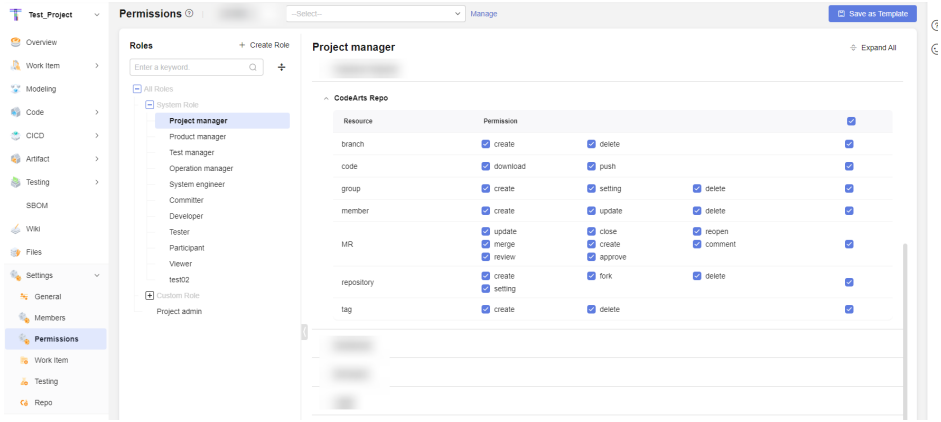
Step 2 Sam chooses **Security Management** > **Member Sync** and enables **Sync project members** to add team members to repository groups and repositories in the project.



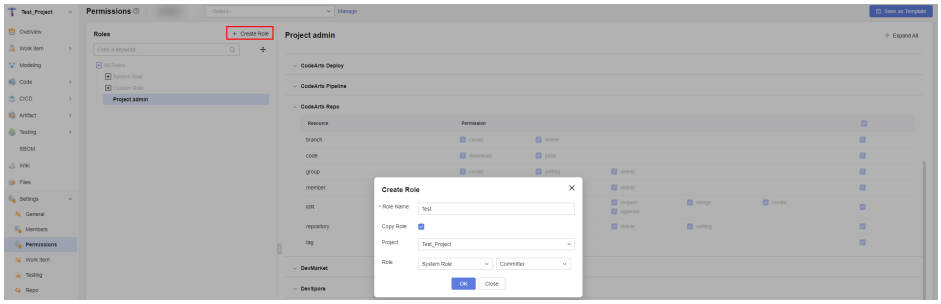
----End

Managing Project Members' Permissions

Administrator Sam has the permission to configure permissions for users of different roles. The following picture shows that Sam is a project manager and has all permissions of CodeArts Repo by default. He can cancel any permission and configure his own permissions.



The following picture shows that Sam adds a custom role **Test**. Click **Create Role** to copy the permissions of a role in another project.

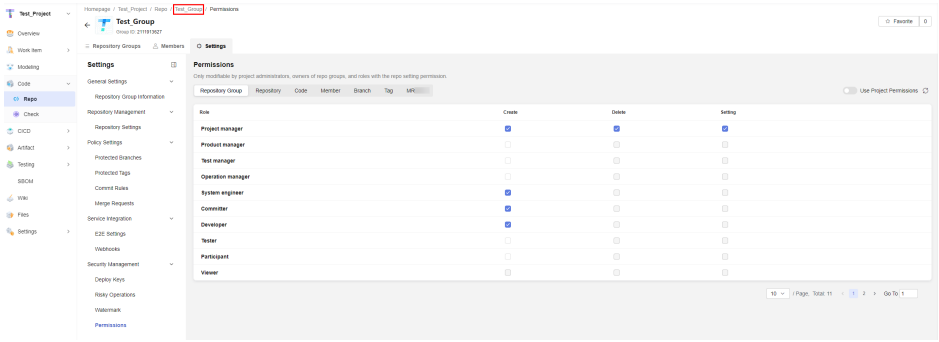


Managing Repository Group Members' Permissions

Considering there are many other repositories under this project, Sam creates a repository group **Test_Group** and configures its permissions for easier

management. Sam does not inherit the project permissions and configures another set of permissions for this repository group.

In the following figure,  indicates that the permission is granted, and you can cancel the permission. indicates that the permission has not been granted and can be granted. indicates that the permission has not been granted and cannot be configured. indicates that the permission has been granted and cannot be canceled.

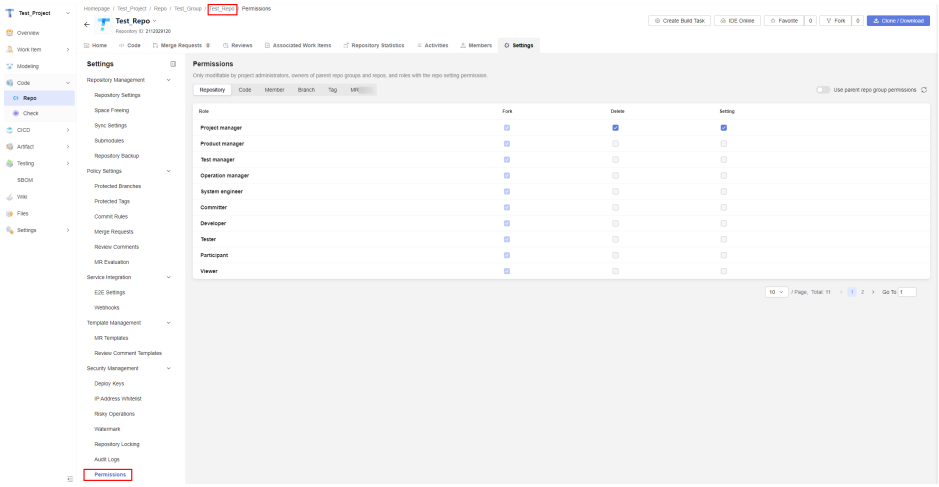


Managing Repository Member Permissions

Sam enters the target repository to configure permissions for developers and does not inherit the parent repository group's permissions because developers are working on different repositories. The following picture shows what permissions Sam configures for each role in **Test_Repo**.

 indicates that the permission has been granted and can be canceled. indicates that the permission has not been granted and can be granted. indicates that the permission has not been granted and cannot be configured.

 indicates that the permission has been granted and cannot be canceled.



3 Migrating GitLab Intranet Repositories to CodeArts Repo in Batches

Background

Currently, CodeArts Repo only allows for repository migrations between public networks. There is no quick solution for migrating repositories from a platform built on the customer's intranet to CodeArts Repo. Therefore, we provide a script for migrating repositories from an intranet platform to CodeArts Repo in batches.

Configuring the SSH Public Key for Accessing CodeArts Repo

To migrate GitLab code repositories to CodeArts Repo in batches, you must install the Git Bash client and configure the locally generated SSH public key to CodeArts Repo. The procedure is as follows:

Step 1 Run Git Bash to check whether an SSH key has been generated locally.

If you select the RSA algorithm, run the following command in Git Bash:

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

If you select the ED255219 algorithm, run the following command in Git Bash:

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

- If **No such file or directory** is displayed, no SSH key has been generated on your computer. Go to **Step 2**.
- If a character string starting with **ssh-rsa** or **ssh-ed25519** is returned, an SSH key has been generated on your computer. If you want to use the generated key, go to **Step 3**. If you want to generate a new key, go to **Step 2**.

Step 2 Generate an SSH key. If you select the RSA algorithm, run the following command to generate a key in Git Bash:

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

In the preceding command, **-t rsa** indicates that an RSA key is generated, **-b 4096** indicates the key length (which is more secure), and **-C your_email@example.com** indicates that comments are added to the generated public key file to help identify the purpose of the key pair.

If you select the ED25519 algorithm, run the following command to generate a key in Git Bash:


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

In the preceding command, **-t ed25519** indicates that an ED25519 key is generated, **-b 521** indicates the key length (which is more secure), and **-C your_email@example.com** indicates that comments are added to the generated public key file to help identify the purpose of the key pair.

Press **Enter**. The key is stored in **~/.ssh/id_rsa** or **~/.ssh/id_ed25519** by default, the corresponding public key file is **~/.ssh/id_rsa.pub** or **~/.ssh/id_ed25519.pub**.

Step 3 Copy the SSH public key to the clipboard. Run the corresponding command based on your operating system to copy the SSH public key to your clipboard.

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

Step 4 Log in to Repo and go to the code repository list page. Click the alias in the upper right corner and choose **This Account Settings** > **Repo** > **SSH Keys**. The **SSH Keys** page is displayed.

You can also click **Set SSH Keys** in the upper right corner of the code repository list page. The **SSH Keys** page is displayed.

Step 5 In **Key Name**, enter a name for your new key. Paste the SSH public key copied in **Step 3** to **Key** and click **OK**. The message "The key has been set successfully. Click Return immediately, automatically jump after 3s without operation" is displayed, indicating that the key is set successfully.

----End

Migrating GitLab Intranet Repositories to CodeArts Repo in Batches

Step 1 Go to **Python official website** to download and install Python3.

Step 2 Log in to GitLab and obtain private_token. In **User settings**, choose **Access Tokens** > **Add new token**.

Step 3 You need to generate an SSH public key locally and configure it in GitLab and CodeArts Repo. For details about how to configure it in CodeArts Repo, see **Configuring the SSH Public Key for Accessing CodeArts Repo**.

Step 4 Call the API for **obtaining a user token (using a password)**. Use the password of your account to obtain a user token. Click the request example button on the right of the API debugging page, set parameters, click the debug button, and copy and save the obtained user token to the local host.

Step 5 Use the obtained user token to configure the **config.json** file. **source_host_url** indicates the GitLab API address on your intranet, and **repo_api_prefix** indicates the open API address of CodeArts Repo.

```
{
    "source_host_url": "http://{source_host}/api/v4/projects?simple=true",
    "private_token": "private_token obtained from GitLab",
    "repo_api_prefix": "https://${open_api}",
    "x_auth_token": " User Token"
}
```


Step 6 Log in to the **CodeArts console**, click , select a region, and click **Access Service**.

Step 7 On the CodeArts homepage, click **Create Project**, and select **Scrum**. If there is no project on the homepage, click **Select** on the **Scrum** card. After creating a project, save the project ID.

Step 8 Use the obtained project ID to configure the **plan.json** file. The following example shows the migration configuration of two code repositories. You can configure the file as required. **g1/g2/g3** indicates the repository group path. If the path is not pre-created, it will be automatically generated according to the configuration.

```
[
   ["path_with_namespace", "Project ID", "g1/g2/g3/Target repository name 1"],
      ["path_with_namespace", "Project ID", "g1/g2/g3/Target repository name 2"]
]
```

NOTE

- To create a repository group, go to the CodeArts Repo homepage, click the drop-down list box next to **New Repository**, and select **New Repository Group**.
- Repository name: Start with a letter, digit, or underscore (_), and use letters, digits, hyphens (-), underscores (_), and periods (.). Do not end with .git, .atom, or periods (.).

Step 9 On the local Python console, create a **migrate_to_repo.py** file.

```
#!/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

# Skip creating a repository with the same name.
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


# Specify a repository group to create a repository.
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 "repository or repository group with the same name" 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"]

    # Check after the creation.
    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']:
            # For scenarios with no repository group.
            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):
    # Obtain the repository group list of a project.
    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"]:
        # Check the target repository group. If the target repository group exists, check the next layer.
        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
        # If the file does not exist, create one and update it.
        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
    # Obtain the list of self-built GitLab repositories and export the result to the FILE_SOURCE_REPO_INFO
file.
    if repo_info_from_source(config) != STATUS_OK:
```



```
        return
    # Clone the repository to your local host.
    status = clone_from_source(config, plans)
    if status != STATUS_OK:
        return

    # Call the CodeArts API to create a repository and record its address in the FILE_SOURCE_REPO_INFO file.
    if create_group_and_repos(config, plans) != STATUS_OK:
        return

    # Push code using SSH. Configure the SSH key in CodeArts Repo first.
    push_to_target(config, plans)


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

Step 10 Run the following command to start the script and migrate code repositories in batches:

```
python migrate_to_repo.py
```

----End

4 Importing Local Repositories to CodeArts Repo in Batches

Background

Currently, CodeArts Repo only supports a single repo import from the public network. There is no quick solution for migrating local repositories to CodeArts Repo. Therefore, we provide a script for migrating local repositories to CodeArts Repo in batches.

Preparations

Step 1 Go to **Python official website** to download and install Python3.

Step 2 Call the API for **obtaining a user token (using a password)**. Use the password of your account to obtain a user token. Click the request example button on the right of the API debugging page, set parameters, click the debug button, and copy and save the obtained user token to the local host.

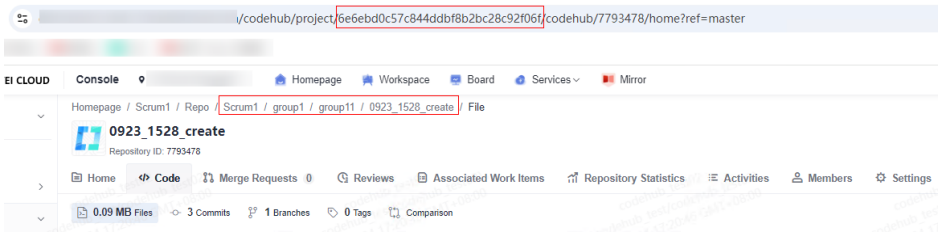
Step 3 Use the obtained user token to configure the **config.json** file. **repo_api_prefix** indicates the open API address of CodeArts Repo.

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

Step 4 Log in to the **CodeArts console**, click , select a region, and click **Access Service**.

Step 5 On the CodeArts homepage, click **Create Project**, and select **Scrum**. If there is no project on the homepage, click **Select** on the **Scrum** card. After creating a project, save the project ID.

Step 6 Configure the **plan.json** file using the obtained project ID. The following example shows the migration configurations of the two code repositories. You can configure them as required. In the following figure, g1/g2 indicates the repo group path. For details about how to create a path, see **NOTE**. This figure shows how to obtain the project ID and *project name/g1/g2/target repo name 1* on the CodeArts Repo page.



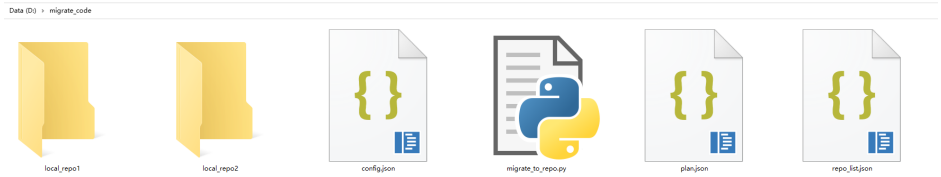

```
[
   ["Project name/g1/g2/Target repo name 1," "Project ID," "Project name/g1/g2/Target repo name 1"],
      ["Project name/g1/g2/Target repo name 2," "Project ID," "Project name/g1/g2/Target repo name 2"]
]
```

NOTE

- To create a repository group, go to the CodeArts Repo homepage, click the drop-down list box next to **New Repository**, and select **New Repository Group**.
- Repository name: Start with a letter, digit, or underscore (_), and use letters, digits, hyphens (-), underscores (_), and periods (.). Do not end with .git, .atom, or periods (.).

Step 7 Add a configuration file named **repo_list.json**.

In this file, **local_dir** indicates the path of the target repository to which the code file in a local directory is uploaded. You must upload a complete Git repository and it must be in the same directory as **migrate_to_repo.py**. As shown in the following figure, **local_repo1** and **local_repo2** indicate the local Git repositories to be uploaded. That is, the values of **local_dir** and **local_dir** are **local_repo1** and **local_repo2**, respectively.



In the following code example, **g1/g2** indicates the repo group path. For details about how to obtain the project ID, see **Obtaining a Project ID**.

```
[
   {
      "id": "Project ID",
      "namespace":"Project name/g1/g2/target repo name 1"
      "local_dir": "Local path 1 of Git repository"
   },
   {
      "id": "Project ID",
      "namespace":"Project name/g1/g2/target repo name 2"
      "local_dir": "Local path 2 of Git repository"
   }
]
```

Step 8 On the local Python console, create a **migrate_to_repo.py** file.

```
#!/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


# Skip creating a repository with the same name.
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


# Specify a repository group to create a repository.
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 "repository or repository group with the same name" 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"]
    # Check after the creation.
    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']:
            # For scenarios with no repository group.
            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):
    # Obtain the repository group list of a project.
    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"]:
        # Check the target repository group. If the target repository group exists, check the next layer.
        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
        # If the file does not exist, create one and update it.
        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):
    # Parse CLI parameters.
```



```
    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)

    # Obtain the mapping, repo information, and namespace.
    status, plans = read_migrate_plan()
    if status != STATUS_OK:
        log.logger.info("load plan.json failed")
        exit(1)

    # Load the configuration file.
    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 the repository to your local host.
        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)
```

----End

Configuring the SSH Public Key for Accessing CodeArts Repo

Step 1 Run Git Bash to check whether an SSH key has been generated locally.

If you select the RSA algorithm, run the following command in Git Bash:

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

If you select the ED255219 algorithm, run the following command in Git Bash:

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

- If **No such file or directory** is displayed, no SSH key has been generated on your computer. Go to **step 2**.
- If a character string starting with **ssh-rsa** or **ssh-ed25519** is returned, an SSH key has already been generated on your computer. If you want to use this key, go to **step 3**. If you want to generate a new key, go to **step 2**.

Step 2 Generate an SSH key. If you select the RSA algorithm, run the following command to generate a key in Git Bash:

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

In the preceding command, **-t rsa** indicates that an RSA key is generated, **-b 4096** indicates the key length (which is more secure), and **-C your_email@example.com** indicates that comments are added to the generated public key file to help identify the purpose of the key pair.

If you select the ED25519 algorithm, run the following command to generate a key in Git Bash:

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

In the preceding command, **-t ed25519** indicates that an ED25519 key is generated, **-b 521** indicates the key length (which is more secure), and **-C your_email@example.com** indicates that comments are added to the generated public key file to help identify the purpose of the key pair.

Press **Enter**. The key is stored in **~/.ssh/id_rsa** or **~/.ssh/id_ed25519** by default, the corresponding public key file is **~/.ssh/id_rsa.pub** or **~/.ssh/id_ed25519.pub**.

Step 3 Copy the SSH public key to the clipboard. Run the corresponding command based on your operating system to copy the SSH public key to your clipboard.

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

Step 4 Log in to Repo and go to the code repository list page. Click the alias in the upper right corner and choose **This Account Settings** > **Repo** > **SSH Keys**. The **SSH Keys** page is displayed.

You can also click **Set SSH Keys** in the upper right corner of the code repository list page. The **SSH Keys** page is displayed.

Step 5 In **Key Name**, enter a name for your new key. Paste the SSH public key copied in **Step 3** to **Key** and click **OK**. The message "The key has been set successfully. Click Return immediately, automatically jump after 3s without operation" is displayed, indicating that the key is set successfully.

----End

Starting Migration in Batch

Step 1 Run the following commands to view the script parameters:

```
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

# Parameter description
# -p: Protocol. SSH by default. SSH, ssh, HTTP, and http are also supported.
```

----End

5 CodeArts Repo Security Configuration Overview

Code Security

CodeArts Repo provides access tokens, deploy keys, and protected branches to safeguard your code assets.

Table 5-1 Code security

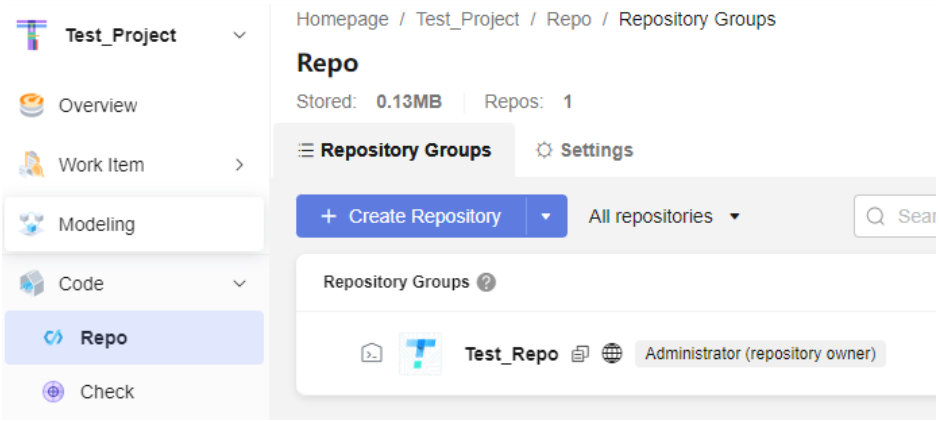
Security Configuration	Description	Suggestion	Reference
Access tokens	CodeArts Repo allows each user to generate access tokens. Tokens are displayed only when generated. You can set the validity period (max. 1 year) of a token. By default, a token is valid for 1 month.	When granting repo access to a third party, create an access token with a specific validity period. Access tokens prevent account and password disclosure.	**Configuring an Access Token**
Deploy keys	CodeArts Repo allows you to add deploy keys for each code repo. Users only have read permissions when accessing a repo using a deploy key.	In code repo reading scenarios, such as builds, use a deploy key to clone a repo to improve code repo security.	**Configuring a Deploy Key for a Repository**

Security Configuration	Description	Suggestion	Reference
Protected branches	You can set branch protection rules in a code repo to prevent branches from being modified or mis-deleted.	Set a protection rule for the master branch so that code can only be merged into it via merge requests. Only authorized roles can push code to protected branches.	**Configuring Protected Branch Rules**
Visibility	CodeArts Repo allows you to set the following visibility options for code repos: • Private (A repo can only be read, written, and accessed by its members.) • Public – Read-only for project members – Read-only for tenant members – Read-only for all visitors	Set the visibility when creating a repo or adjust the visibility for an existing repo to scale to your needs. The administrator can determine whether to allow members to create "Public" code repos.	• For details about how to set the visibility of a repo, see **Creating a Custom Repository**. • To set whether to allow members to create "Public" repos, see **Adjusting Repository Visibility**.
Commit rules	CodeArts Repo control code commits using specific rules. You can use the preconfigured commit rules or create new ones.	Set commit rules for each repo to prevent your code from being modified without permission.	**Configuring Commit Rules**

6 Configuring HTTPS Password

Background

As shown in the following figure, developer Sam uses a new IAM account to develop the **Test_Repo** repository of the **Test_Project**.



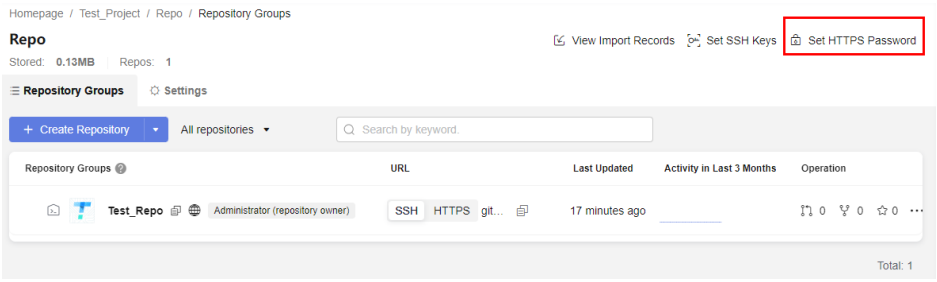
When Sam pushes code to or pulls code from CodeArts Repo, CodeArts Repo needs to verify his identity and permissions. HTTPS password is an authentication mode for remote access to CodeArts Repo. This case describes how Sam configures the HTTPS password for the first time, changes the HTTPS password, clones code files from CodeArts Repo on Windows, and uploads code files to CodeArts Repo.

Preparations

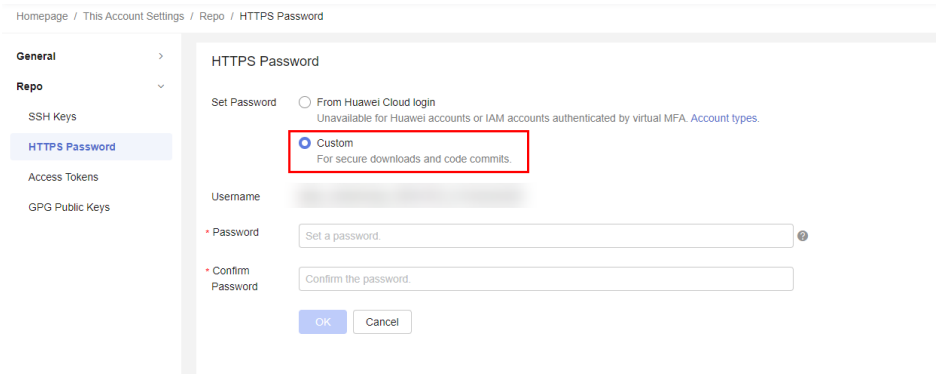
Developers need to configure the Git client (later than 2.30) on a local PC and configure the username and email address. For details, see **Installing and Configuring Git Client**.

Configuring an HTTPS Password for the First Time

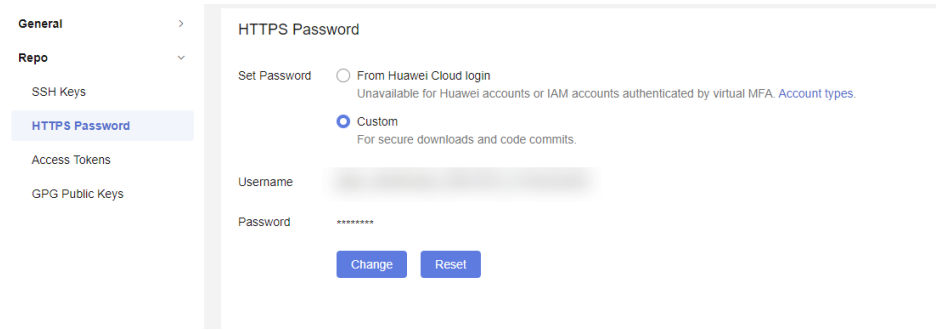
Step 1 Sam **logs in to the CodeArts Repo homepage** for the first time and clicks **Set HTTPS Password** in the upper right corner of the page.



Step 2 By default, Sam's tenant name or IAM username is the account name, and the password is the login password of Sam's account. To ensure information security, Sam selects **Custom**, resets and saves the new password **RepoNewPassword**.



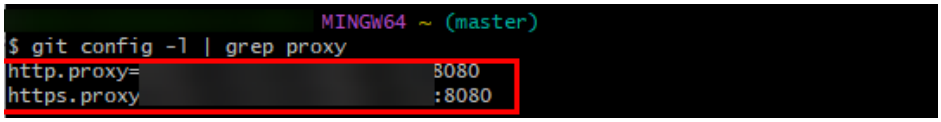
Step 3 A dialog box is displayed in the upper right corner, indicating that Sam successfully sets the custom password. The custom password page is displayed. Sam copies and saves the username and password on the local PC.



Step 4 Sam has already configured a proxy on the local PC previously and now he runs the following command to view the configured Git proxy:

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

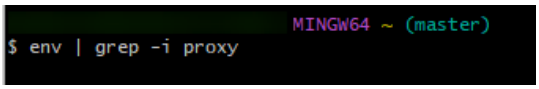
The command output indicates that the HTTP and HTTPS proxies of Git have been configured.



Step 5 Sam runs the following command to check the proxy of the environment:

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

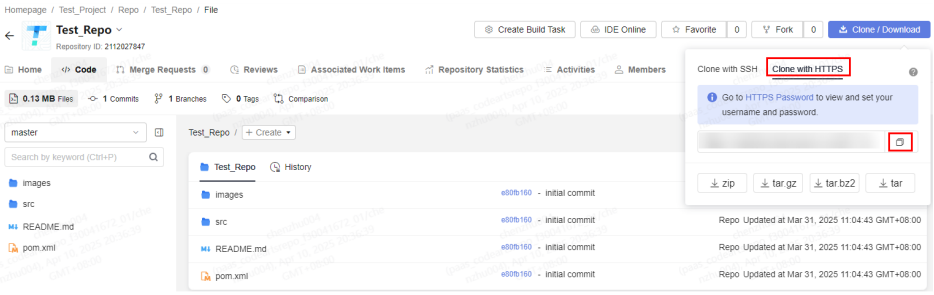
If the command output is empty, no proxy is configured for the environment.



Step 6 Sam runs the following command to cancel the global HTTP and HTTPS proxies of Git:

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

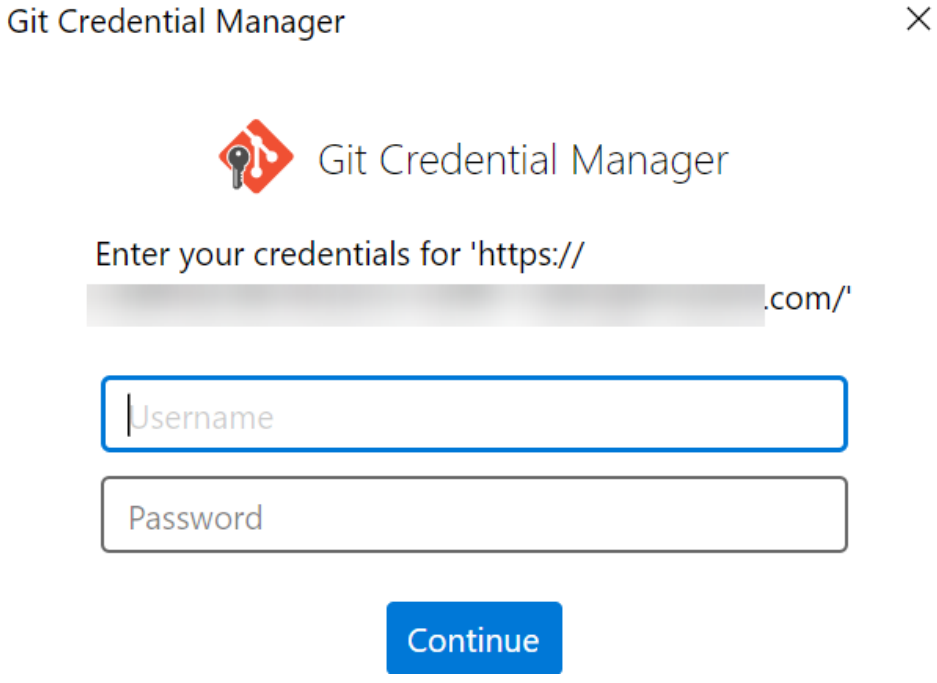
Step 7 Sam copies the HTTPS address and clones the repository to the local host for development.



Step 8 Sam runs the following command to check whether the configured HTTPS password is valid. **https//*example*.git** is the address of the repository to be cloned.

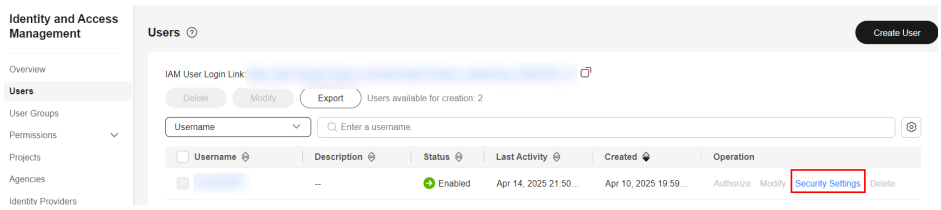
```
git clone https://example.git
```

Step 9 As shown in the following figure, Sam enters the account name and password copied and saved in **Step 3**.

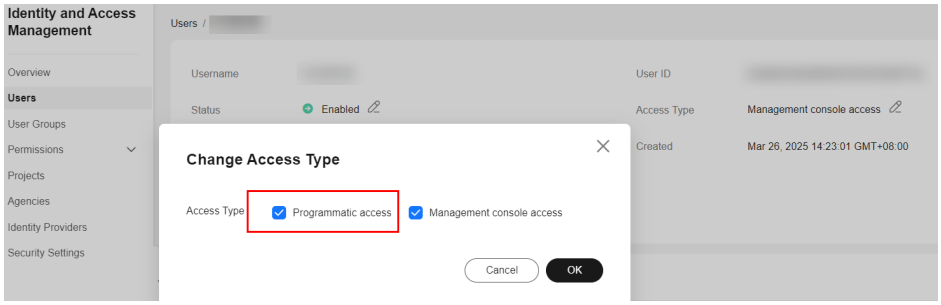


Step 10 After Sam enters the account and password, the repository fails to be cloned, and the error message "**remote: <CH.00905401> HTTP Basic: Access denied. Request-id is sQIEbze60W. fatal: Authentication failed for 'https://example.com/Test_Repo.git/**" is displayed, indicating that the authentication fails.

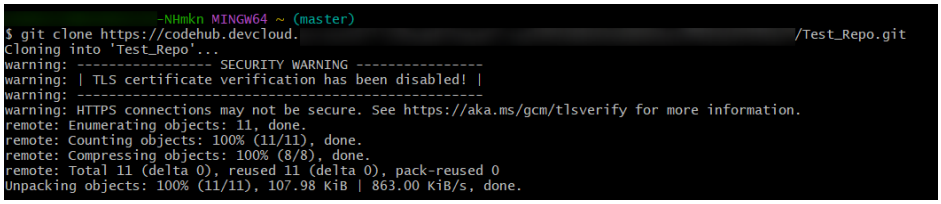
Step 11 Sam contacts the tenant administrator to add permissions for Sam. After logging in to the console, the administrator clicks the profile image in the upper right corner and chooses **Identity and Access Management**, as shown in the following figure.



Step 12 After clicking **User**, the administrator locates Sam's account and clicks **Security Settings** in the **Operation** column to select **Programmatic access** for Sam. After the administrator adds the permission, Sam will be notified to clone the repository again.



Step 13 Sam runs the Git clone command again on the local host and the repository is successfully cloned to the local host, as shown in the following figure.



----End

7 Best Practices for Configuring Protected Branch Rules

7.1 Overview

Set up protected branch rules during collaborative coding to enhance code quality, safeguard branches, and boost teamwork for reliable, secure code delivery.

This section provides an overview of the best practices including:

- **Application Scenarios**
- **Advantages**
- **Constraints**

Application Scenarios

Developing code without protected branch rules may interrupt your development pace and even delivery.

- Merging unreviewed code into main branches may cause defects and further incur online faults and higher O&M costs.
- Key branches are forcibly pushed and overwritten, causing historical records to be lost. As a result, problems are difficult to trace, affecting fault locating and compliance audit.
- CI/CD process is not strictly followed. As a result, bad code enters the production environment, which may cause version rollback or user complaints.
- Unstandardized collaboration and random branch merge cause frequent code conflicts and disrupted version iteration.

Protected branch rules can help you solve the following pain points:

- Core branches (such as master): No role is allowed to push code to core branches. Only compliant code can be brought online, reducing production risks.
- Permission control: Only authorized members can merge key branches to block unqualified code and maintain enterprise security standards.

Advantages

Setting protected branch rules for key code branches in CodeArts Repo, your team will enjoy the following core advantages:

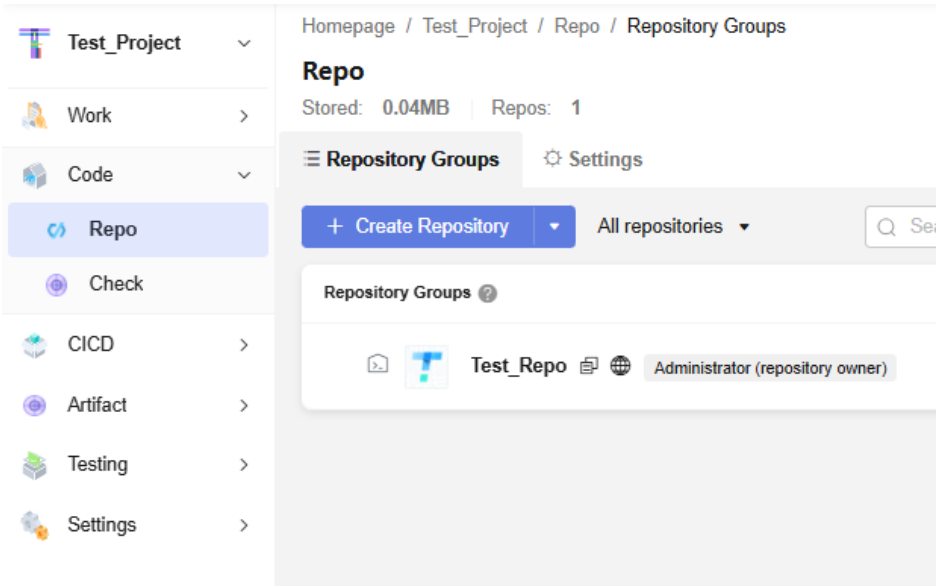
- Prevent direct code push. Submitting changes via merge requests can shield developers from directly pushing code to the protected branches, facilitating code review.
- Ensure code quality. Merging merge requests by authorized members improves code maintainability.
- Control permissions and compliance. Allowing authorized members to merge code strengthens enterprises' hierarchical management and helps trace code change sources.
- Reduce conflicts and improve collaboration efficiency. Prohibiting direct pushes to protected branches minimizes code conflicts caused by members' unstandardized operations and helps new members quickly get started by standardizing the push process.

Constraints

Only a project manager or project administrator can configure protected branch rules for all repositories in a project. Refer to the following steps:

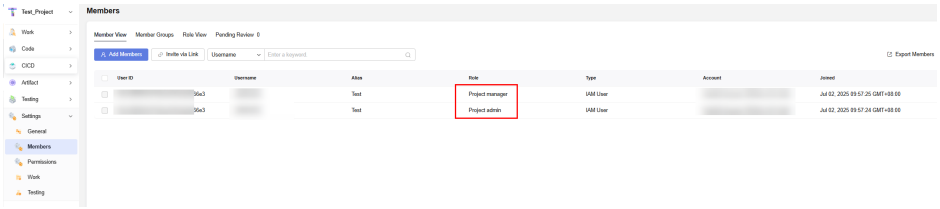
Step 1 As shown in the following figure, go to the **Repo** page of **Test_Project** to be configured.

Figure 7-1 Repo page of **Test_Project**



Step 2 In the left navigation pane, choose **Settings > Members**. As shown in the following figure, the current user is a project manager and project administrator. Therefore, the current user has the permission to configure protected branch rules for the project.

Figure 7-2 Member view page



----End

7.2 Steps

Step 1: Check Whether You Have the Operation Permissions

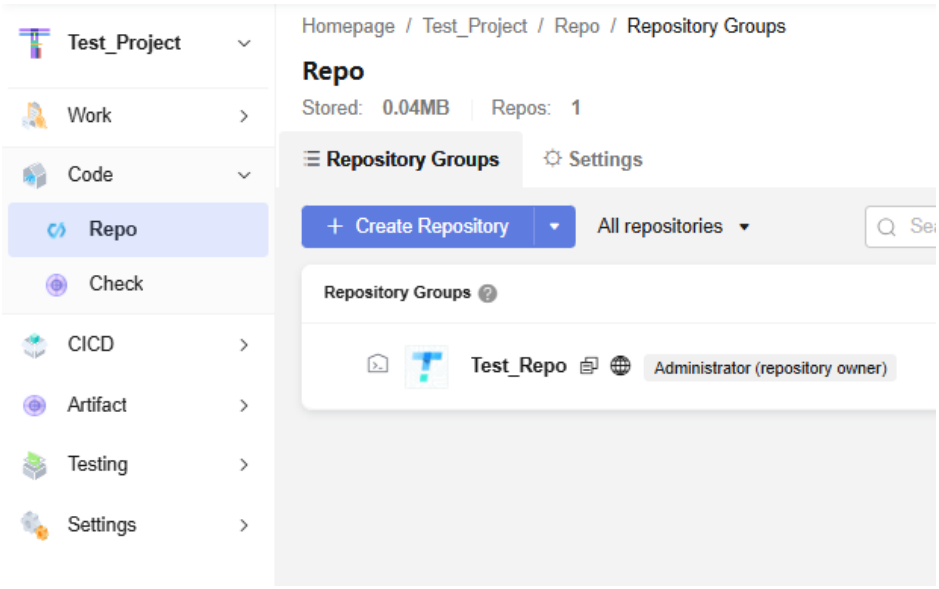
If you want to configure project-level branch rules, check whether you are the project manager or project administrator by referring to **Constraints**.

If you are not the project manager or project administrator, contact the project manager or project administrator to configure protected branch rules.

Step 2: Configure Project-level Protected Branch Rules

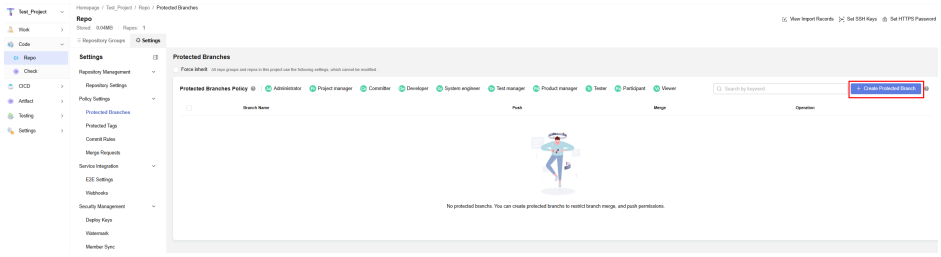
Step 1 As shown in the following figure, access the **Repo** page of **Test_Project** to be configured as a project manager or project administrator.

Figure 7-3 Repo page of **Test_Project**



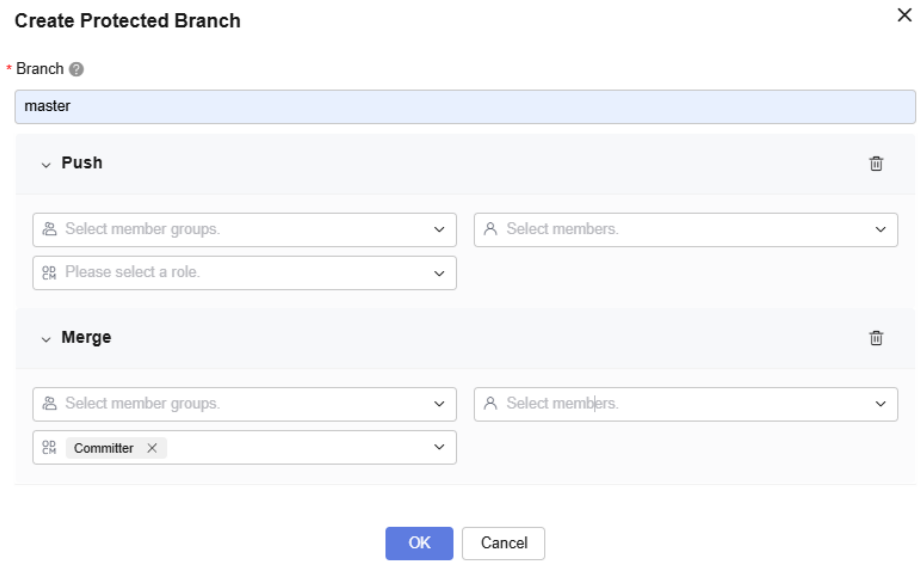
Step 2 On the top navigation bar, choose **Settings** > **Policy Settings** > **Protected Branches** and click **Create Protected Branch**.

Figure 7-4 Going to the page for creating a protected branch



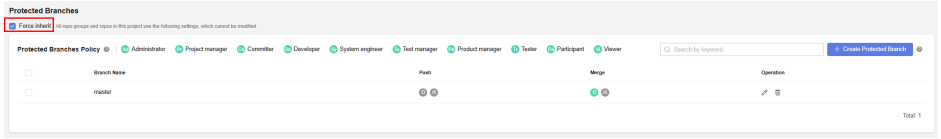
Step 3 As shown in the following figure, configure the master branch as a protected branch. No one but committers can push code to the master branch.

Figure 7-5 Configuring protected branch rules



Step 4 As shown in the following figure, select **Force inherit**, indicating that all repositories in the project will use the protected branch rule.

Figure 7-6 Force inherit



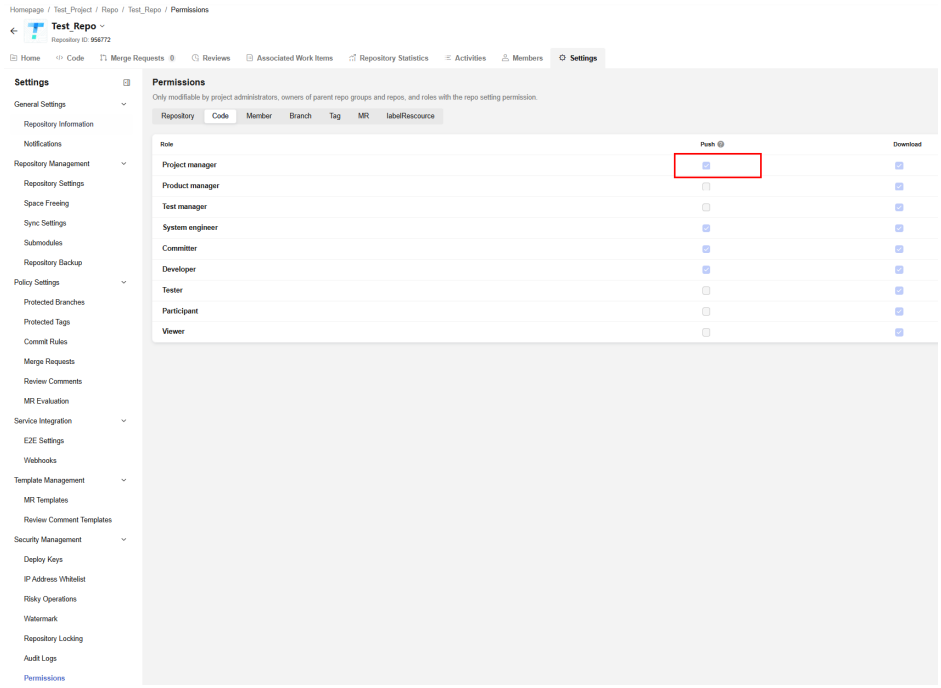
----End

Step 3: Demo on How a Project Manager Failed to Push Files to the Master Branch

Step 1 As a project manager, check whether you have the commit permission on the code. All repository members need to do this before pushing a code file to the repository.

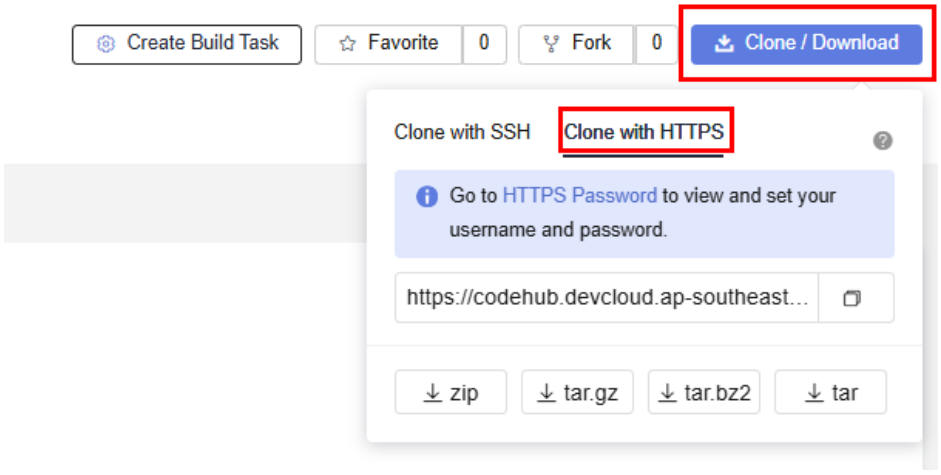
As shown in the following figure, access **Test_Repo** to be pushed, choose **Settings** > **Security Management** > **Permissions** on the top navigation bar, and click **Code**.

Figure 7-7 Checking whether you have the push permission on the code



Step 2 After confirming that you have the push permission, click **Clone/Download** > **Clone with HTTPS** in the upper right corner, and click to copy the HTTPS address of the repository.

Figure 7-8 Copying the HTTPS address



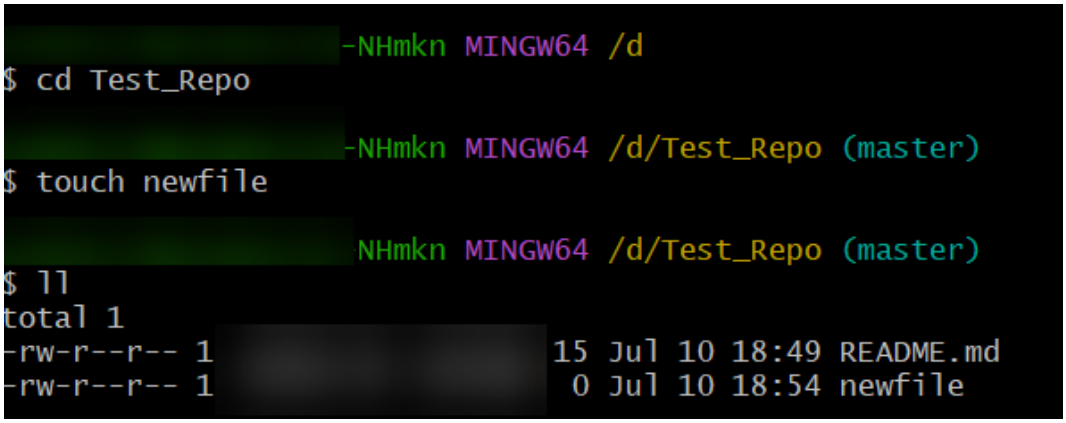
Step 3 Run the clone command to clone **Test_Repo** to the local host, as shown in the following figure.

Figure 7-9 Cloning "Test_Repo" to the local PC

Step 4 Run the following commands in sequence to go to the repository directory of **Test_Repo**, create the **newfile** file, and check whether it is successfully executed.

```
cd Test_Repo
touch newfile
ll
```

Figure 7-10 Creating newfile and checking whether it is successfully created

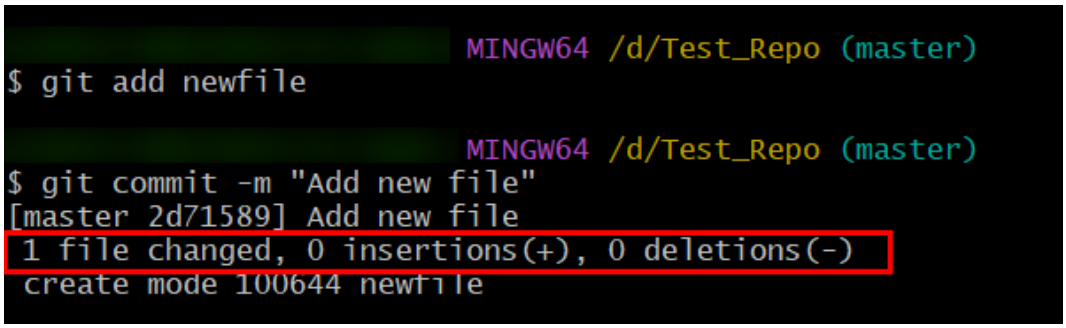


Step 5 Run the following commands in sequence: Run the **git add** command to add the new file **newtext** to the staging area. Run the **git commit** command to submit the change of the file to the Git history and make a note that the commit adds a new file.

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

As shown in the following figure, if the command output in the red box is displayed, the commit is successful.

Figure 7-11 Committing the newfile file to the Git history

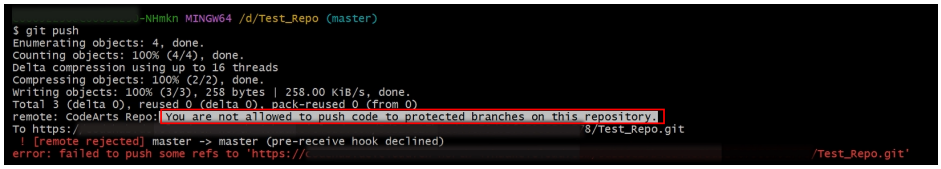


Step 6 Run the **git push** command to push the latest commit on the local main branch to the main branch of the remote repository.

```
git push
```

As shown in the following figure, if the information in the red box is displayed, the file fails to be pushed to the master branch because it is a protected branch. Even if you have the permission to push code, you cannot directly push code files to the repository.

Figure 7-12 git push failure



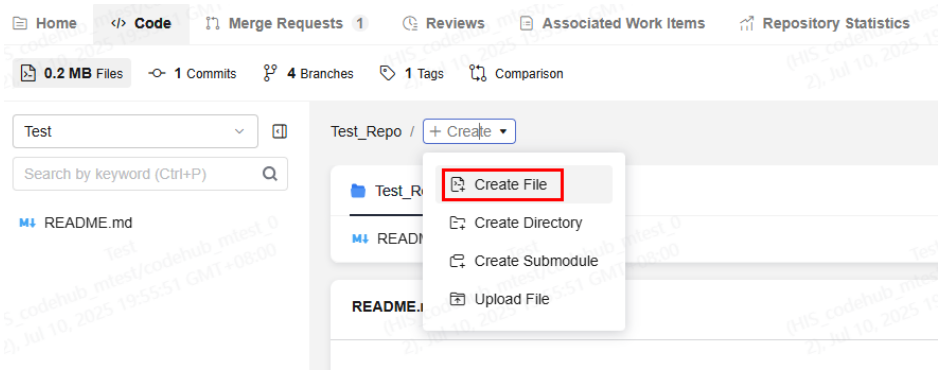
----End

Step 4: Demo on How a Committer Merges the Code to the Master Branch

Step 1 As shown in the following figure, the project manager goes to the **Code** tab page of **Test_Repo**, selects the master branch, clicks **Create Branch**, enters **Test** for the branch name, and clicks **OK**.

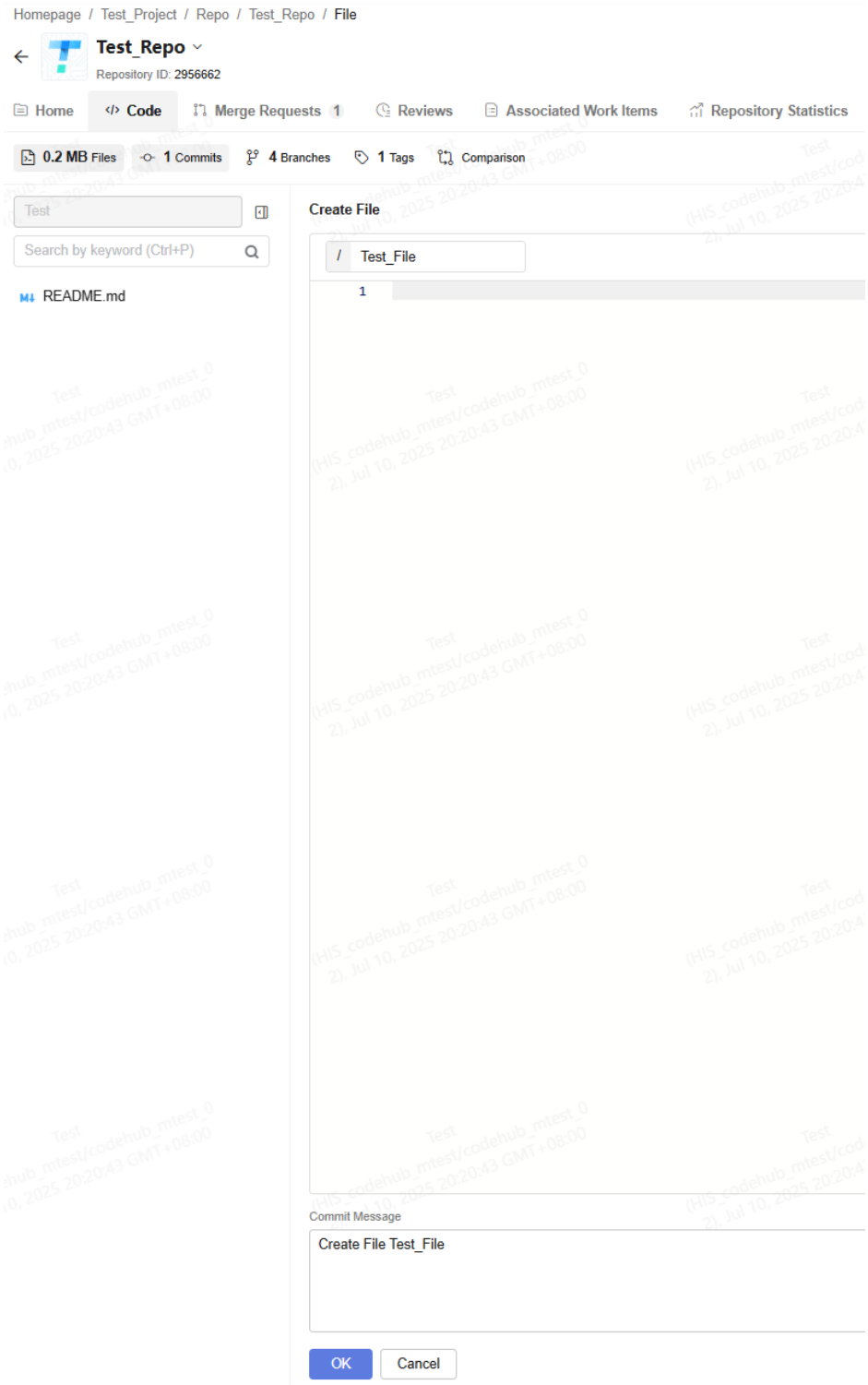
Step 2 As shown in the following figure, click **Create File** for the **Test** branch.

Figure 7-13 Clicking **Create File**



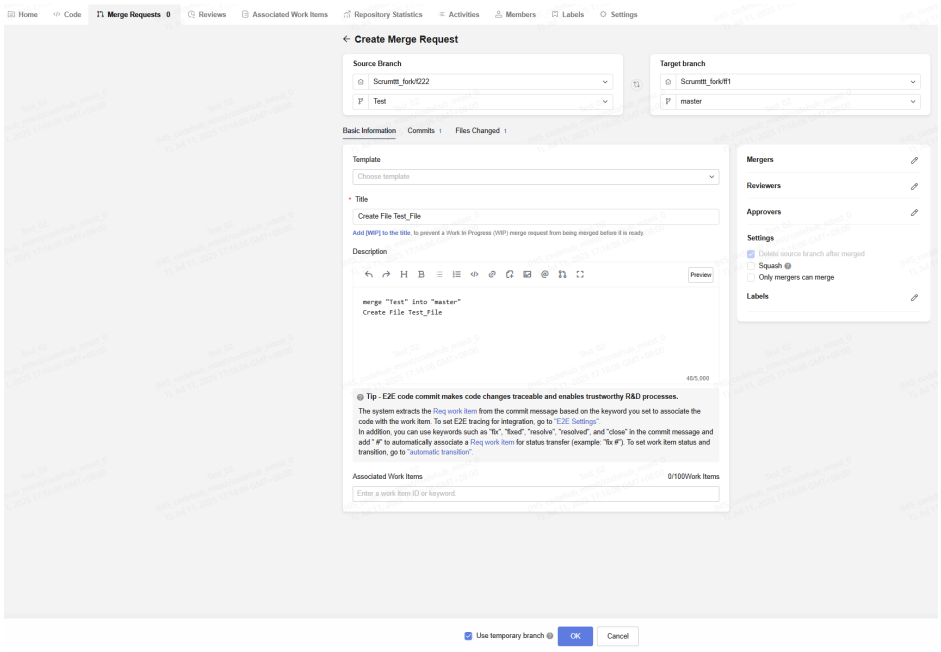
Step 3 As shown in the following figure, enter **Test_File** for the file name and click **OK**.

Figure 7-14 Creating the **Test_File** file



Step 4 Click **Merge Requests** in the top navigation bar and click **Create MR**. Enter **Title**, select **Use temporary branch**, and click **OK**.

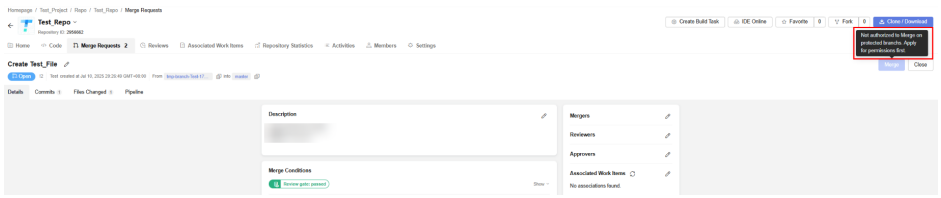
Figure 7-15 Creating a merge request



Step 5 On the **Merge Requests** tab page, the **Merge** button in the upper right corner is unavailable. When you click **Merge**, the error message shown in the following figure is displayed.

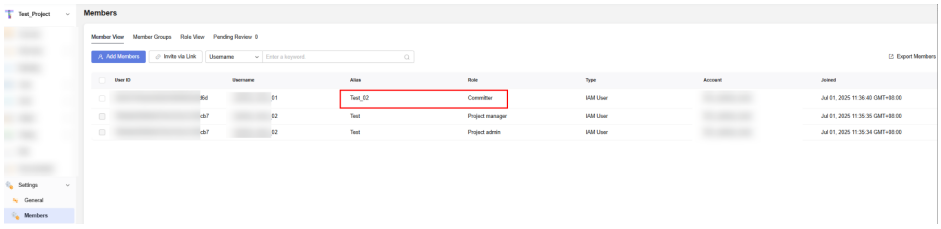
It is because the master branch is a protected branch. Only the committer role can merge code.

Figure 7-16 Project manager failing to merge a merge request



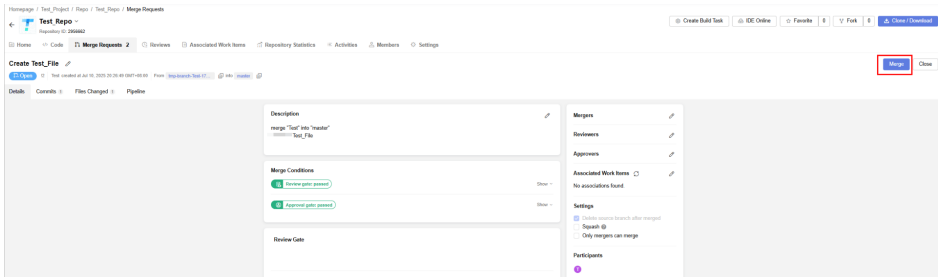
Step 6 As shown in the following figure, **Test_02** is a committer. The project manager **Test** contacts **Test_02** to merge the merge request.

Figure 7-17 Confirming the committer role



Step 7 As shown in the following figure, **Test_02** goes to the **Merge Requests** page and merges the merge request of the project manager **Test**.

Figure 7-18 Committer merging the merge request



----End

