代码托管(CodeArts Repo)

最佳实践

文档版本 01

发布日期 2024-10-14

华为云计算技术有限公司

商标声明


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

注意

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

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

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 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。

2 批量迁移 GitLab 内网仓库到 CodeArts Repo

背景介绍

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

配置访问 CodeArts Repo 的 SSH 公钥

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

步骤1 运行Git Bash，先检查本地是否已生成过SSH密钥。

如果选择RSA算法，请在Git Bash中执行如下命令：

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

如果选择ED255219算法，请在Git Bash中执行如下命令：

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

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

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

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

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

如果选择ED25519算法，在Git Bash中生成密钥的命令如下：

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

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

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

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

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

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

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

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

----结束

批量迁移 GitLab 内网仓库到 CodeArts Repo

步骤1 进入**Python官网**下载并安装Python3。

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

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

步骤4 调试接口，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?page=1&per_page=5",
   "private_token": "GitLab上获取的private_token",
   "repo_api_prefix": "https://${open_api}",
   "x_auth_token": "用户Token"
}
```

步骤6 登录CodeArts首页创建项目并保存您的项目ID。

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

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

说明

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

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

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

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

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


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


log = Logger(FILE_LOG)


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

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


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


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

    repo = repos[namespace]
    return STATUS_OK, repo


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

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

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

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


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


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


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

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


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

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

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


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

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

    return STATUS_OK, ssh_url


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

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

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

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


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


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

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

    return status, ssh_url


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

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

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

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

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


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

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

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


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

    # 调用接口创建仓库，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9 执行如下命令，启动脚本并完成代码仓的批量迁移。

```
python migrate_to_repo.py
```

----结束

3 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。

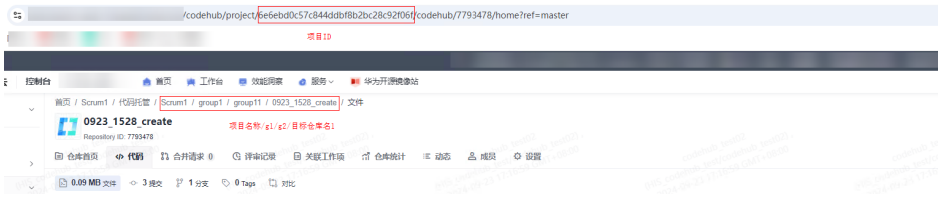
步骤2 调试接口，参考**获取IAM用户Token（使用密码）**通过账号的用户密码获取用户Token。参数的填写方法，您可以在接口的调试界面，单击右侧“请求示例”，填写好参数后，单击“调试”，将获取到的用户Token复制并保存到本地。

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**创建项目并保存您的项目ID。

步骤5 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。



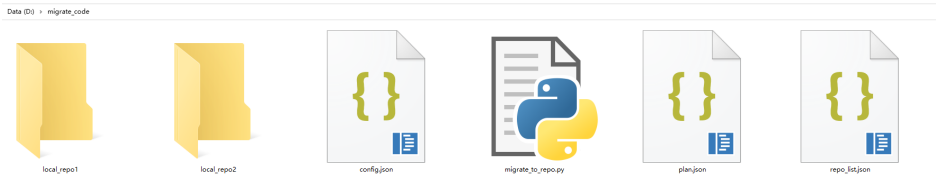

```
[
   ["项目名称/g1/g2/目标仓库名1", "项目ID", "项目名称/g1/g2/目标仓库名1"],
      ["项目名称/g1/g2/目标仓库名2", "项目ID", "项目名称/g1/g2/目标仓库名2"]
]
```

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6 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7 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ED255219算法，请在Git Bash中执行如下命令：

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

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

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

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

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

如果选择ED25519算法，在Git Bash中生成密钥的命令如下：

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

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

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

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

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

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

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

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

----结束

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