Jenkins搭建与使用

[liangdong@smzdm.com](mailto:liangdong@smzdm.com)

[安装jenkins 2](#_Toc531870801)

[搭建master 2](#_Toc531870802)

[搭建slave 4](#_Toc531870803)

[添加slave到master 5](#_Toc531870804)

[配置实战 7](#_Toc531870805)

[构造nginx+php基础镜像 7](#_Toc531870806)

[nginx 7](#_Toc531870807)

[php 8](#_Toc531870808)

[测试 9](#_Toc531870809)

[上传镜像 10](#_Toc531870810)

[创建PHP测试项目 11](#_Toc531870811)

[build.sh 12](#_Toc531870812)

[deploy.sh 13](#_Toc531870813)

[构建镜像 13](#_Toc531870814)

[编写dockerfile 14](#_Toc531870815)

[测试构建 14](#_Toc531870816)

[推送镜像 16](#_Toc531870817)

[基于pipeline自动构建 17](#_Toc531870818)

[搭建骨架 17](#_Toc531870819)

[Stage 下载代码 21](#_Toc531870820)

[Stage 构建应用 23](#_Toc531870821)

[Stage 打包镜像 25](#_Toc531870822)

[Stage 上传镜像 27](#_Toc531870823)

# 安装jenkins

Jenkins搭建在物理机上，1个master做调度，N个slave执行任务。

我们只在master安装Jenkins，配置slave节点后，master会通过ssh连接到slave自动部署slave进程，这一点还是很方便的。

准备2台ubuntu服务器，下面开始搭建。

## 搭建master

参考：<https://pkg.jenkins.io/debian-stable/> <https://jenkins.io/zh/doc/book/installing/>

添加一个apt的东西：

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add –

编辑/etc/apt/sources.list文件，添加如下一行：

deb [https://pkg.jenkins.io/debian-stable binary/](https://pkg.jenkins.io/debian-stable%20binary/)

然后开始安装java和jenkins：

apt-get update

apt-get install openjdk-8-jdk

apt-get install jenkins

安装后jenkins就启动了，为了让它有足够的权限访问docker之类的，我们修改一下/etc/default/jenkins，把运行用户改成root：

JENKINS\_USER=root

JENKINS\_GROUP=root

然后重启systemctl restart jenkins

Jenkins的所有程序以及数据存放在目录/var/lib/jenkins下。

浏览器访问8080端口可以进入jenkins，默认登录账号：admin，密码查看：cat /var/lib/jenkins/secrets/initialAdminPassword。

根据引导安装推荐插件即可，期间可能因为GFW会导致部分插件安装失败，修复方法如下：



进入插件管理，替换一下插件仓库：



<https://mirrors.tuna.tsinghua.edu.cn/jenkins/updates/update-center.json>

## 搭建slave

进入slave机器，改一下root密码：

sudo su root

passwd

修改ssh允许root用户登录：

vim /etc/ssh/sshd\_config

修改如下的行：

PermitRootLogin yes

然后重启ssh：

systemctl restart ssh

**回到master机器**，生成rsa：

ssh-keygen

将pub拷贝到slave上，以便master免密登录slave：

ssh-copy-id [root@172.18.11.9](mailto:root@172.18.11.9) ，输入slave的密码，然后可以在slave上看到~/.ssh/authorized\_keys文件了，试一下ssh可以从master登到slave即可。

**回到slave机器**，创建slave的工作目录：

mkdir -p /var/lib/jenkins/

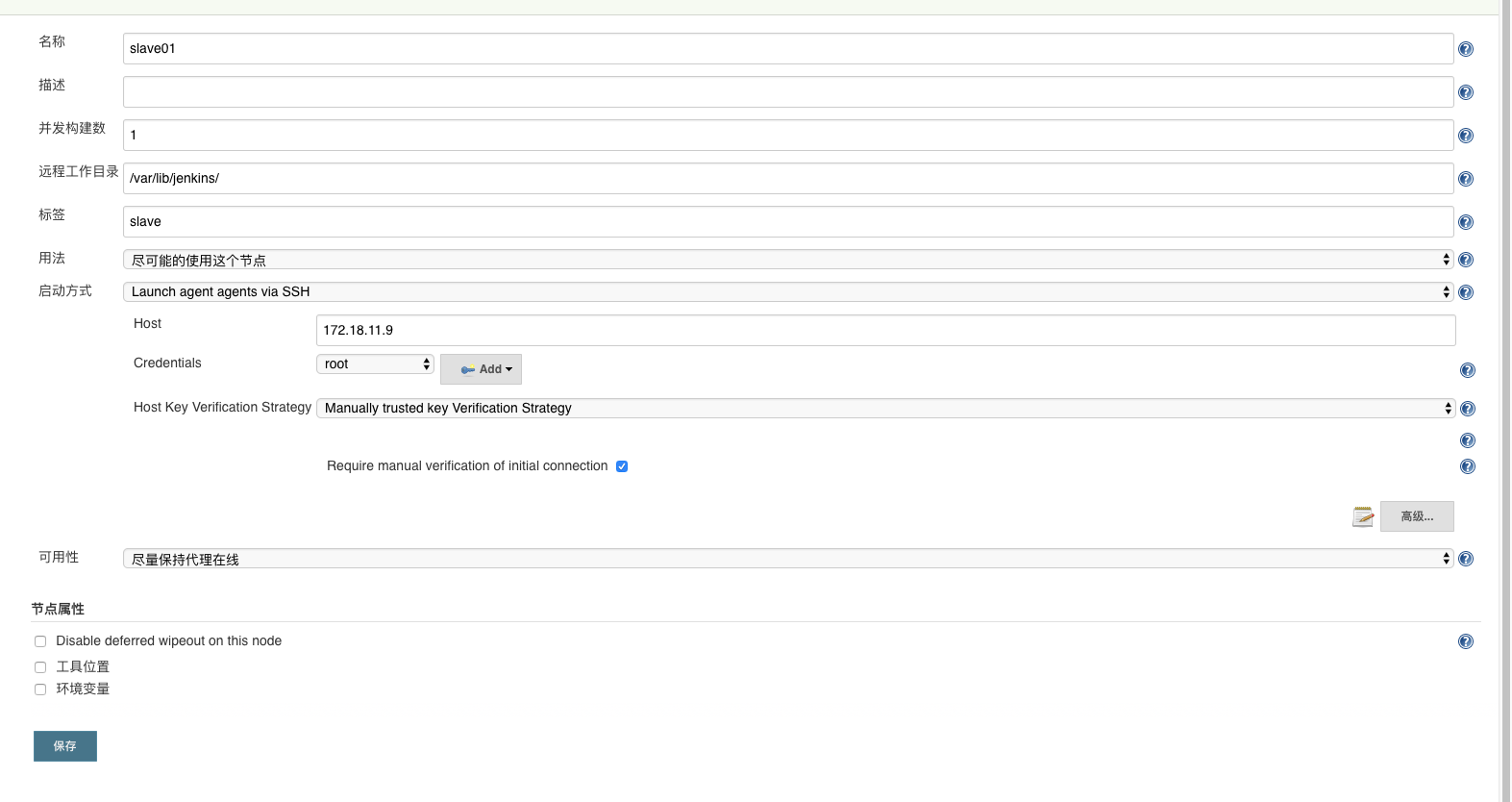
安装java环境：

apt-get update

apt-get install openjdk-8-jdk

## 添加slave到master

回到master，进入系统设置->节点管理，添加一个从节点：

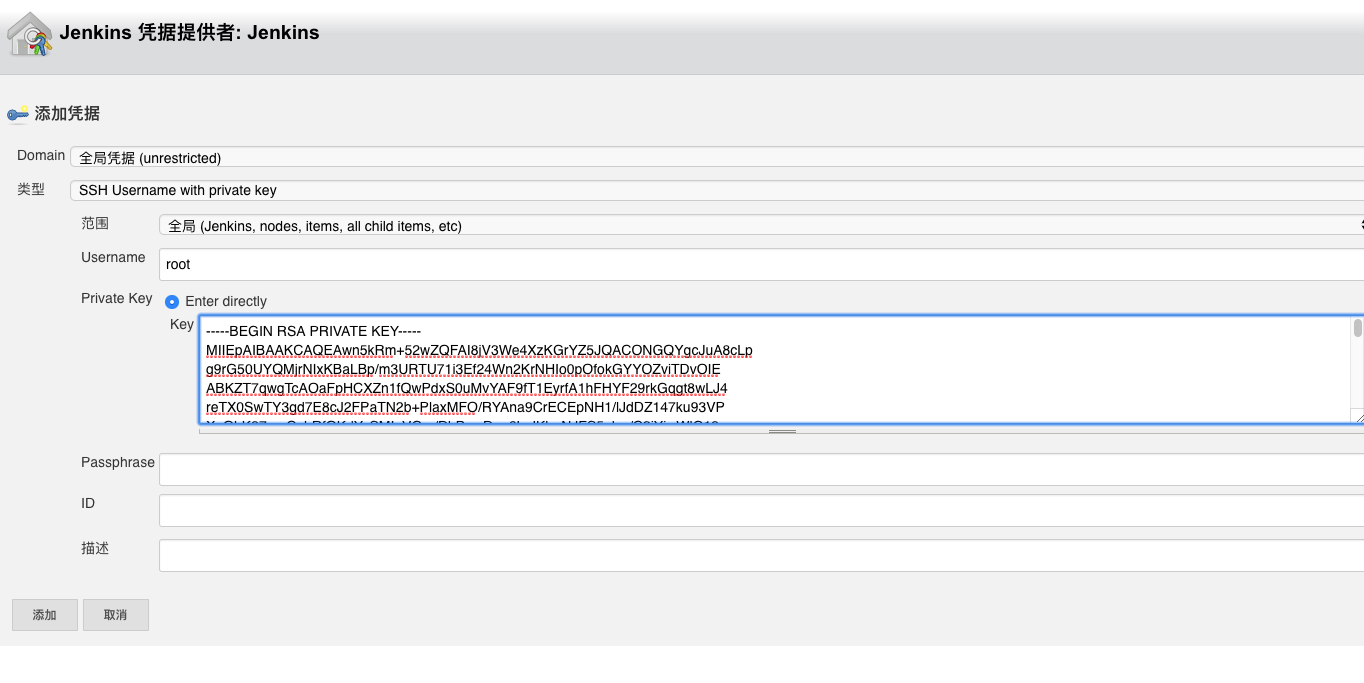


并发构建数用于控制同时执行的任务数量，slave就1个cpu所以就写了1，也可以多写。

工作目录/var/lib/jenkins，刚才我建的。

可以给节点打上label，这样后续可以为特定任务指定执行节点，我这里打一个slave标签，后续我们让任务都在slave上执行，保证master资源充足。

启动方式配置slave机器的IP，点击add添加ssh配置（如下图），这样master就可以通过ssh连到slave上部署slave进程了。



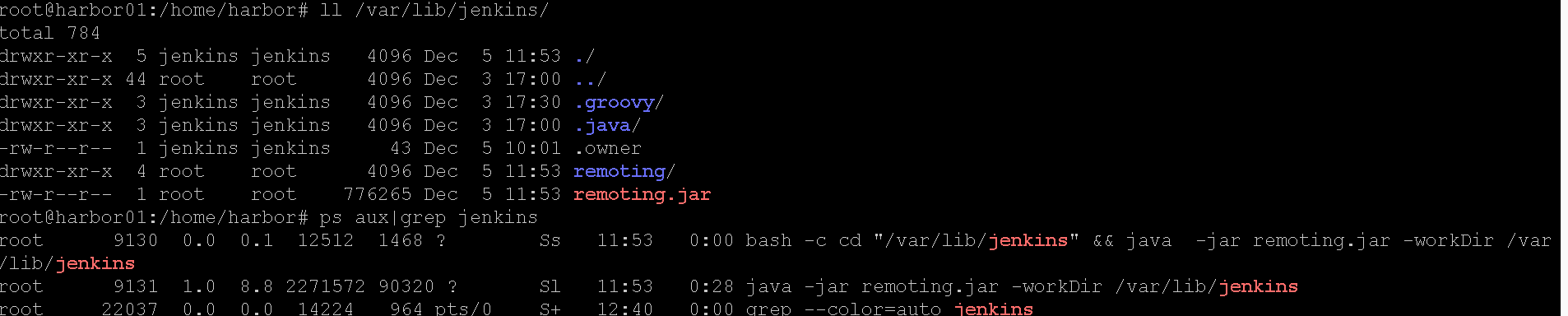
帐号是slave的root，私钥是master的~/.ssh/id\_rsa，因为我们配置了slave信任master，所以只要master用私钥加密ssh请求就可以证明自己的身份了，很常规的ssh免登要求。

Host Key Verification Strategy照选即可，即首次ssh连接slave需要我们人工确认一次。

一切正常可以看到slave节点：



进入slave机器，查看/var/lib/jenkins目录有数据出现：



Slave进程也启动了。

# 配置实战

Jenkins核心就是用pipeline功能配置CI/CD流程，其基本骨架教程：<https://jenkins.io/doc/book/pipeline/> 。

在具体pipeline的时候会有很具体的目标，需要用到jenkins提供的各种具体指令，可以打开：<http://IP:8080/pipeline-syntax/>，可视化的选择与生成指令。

片段生成器和[Declarative Directive Generator](http://172.18.10.236:8080/directive-generator)都很有用。

下面我制定一个小目标：

1. 准备一个centos的base镜像，里面提前安装了标配的nginx和php环境，把镜像存到harbor中。
2. 每个接入项目在根目录下放一个build.sh脚本，在pipeline中会调用给应用一个自定义构建的机会，要求build.sh产出一个deploy.tar.gz文件，它将被打入到镜像中。
3. 在deploy.tar.gz中，要求放一个deploy.sh脚本，镜像启动后会先执行deploy.sh给应用部署的机会（比如把代码放到webroot下，把nginx server配置文件放到指定目录），然后镜像会令nginx重新加载配置，容器就算启动完成了。

## 构造nginx+php基础镜像

### nginx

拉一个centos:6的镜像，进去把nginx和php都装一下，然后再保存这个镜像到harbor作为base镜像即可。

下载centos（版本7的systemctl有点问题，我们用6就行）：

docker pull centos:6

后台运行centos6：

docker run -d centos:6 sh -c 'while true; do sleep1; done'

找到容器ID：

docker ps

连接到容器里（写dockerfile太麻烦，直接进去装程序好了）：

docker exec -it c600aeab916f /bin/bash

安装vim：

yum install vim

安装nginx（参考：）

vim  /etc/yum.repos.d/nginx.repo

添加

[nginx]

name=nginx repo

baseurl=http://nginx.org/packages/centos/6/x86\_64/

gpgcheck=0

enabled=1

然后yum install nginx完成安装。

删除nginx自带的80端口服务：/etc/nginx/conf.d/default.conf，然后拉起执行/etc/init.d/nginx start启动nginx。

我们现在可以配置/etc/nginx/nginx.conf改公共基础配置，后续我们的PHP应用只需要把自己的配置放到conf.d目录即可。

现在把这个容器保存到镜像中持久化下来：

docker commit de04b78f32e8 php-base:v1

### php

回到容器中：docker exec -it c600aeab916f /bin/bash

现在安装php（参考<https://www.tecmint.com/install-php-7-in-centos-6/>，随便用yum一安吧，以后生产环境还是自己编译PHP）：

yum install <https://dl.fedoraproject.org/pub/epel/epel-release-latest-6.noarch.rpm>

yum install <http://rpms.remirepo.net/enterprise/remi-release-6.rpm>

yum install yum-utils

yum-config-manager --enable remi-php71

yum install php  php-bcmath php-cli php-common  php-devel php-fpm    php-gd php-imap  php-ldap php-mbstring php-mcrypt php-mysqlnd   php-odbc   php-pdo   php-pear  php-pecl-igbinary  php-xml php-xmlrpc php-opcache php-intl php-pecl-memcache php-pecl-redis

然后启动php：

/etc/init.d/php-fpm start

然后再保存一下镜像：

docker commit de04b78f32e8 php-base:v2

### 测试

创建目录：

mkdir -p /data/webroot/phpsrc/test

编辑/data/webroot/phpsrc/test/index.php:

<?php

echo 'hello world';

编辑/etc/nginx/conf.d/test.conf：

server {

listen 809;

server\_name test.smzdm.com;

root /data/webroot/phpsrc/test;

charset utf-8;

location / {

index index.php;

if (!-e $request\_filename){

rewrite ^(.\*)$ /index.php last;

break;

}

}

location ~ \.php$ {

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

fastcgi\_connect\_timeout 300;

fastcgi\_send\_timeout 300;

fastcgi\_read\_timeout 300;

fastcgi\_buffer\_size 128k;

fastcgi\_buffers 32 32k;

include fastcgi\_params;

port\_in\_redirect off;

fastcgi\_pass 127.0.0.1:9000;

}

}

然后热加载nginx配置：

/etc/init.d/nginx reload

访问测试：

curl localhost:809

打印了：

hello world

说明nginx和php可以按照预期工作，我们只需要把PHP代码和nginx配置放到对应位置，就可以工作了。

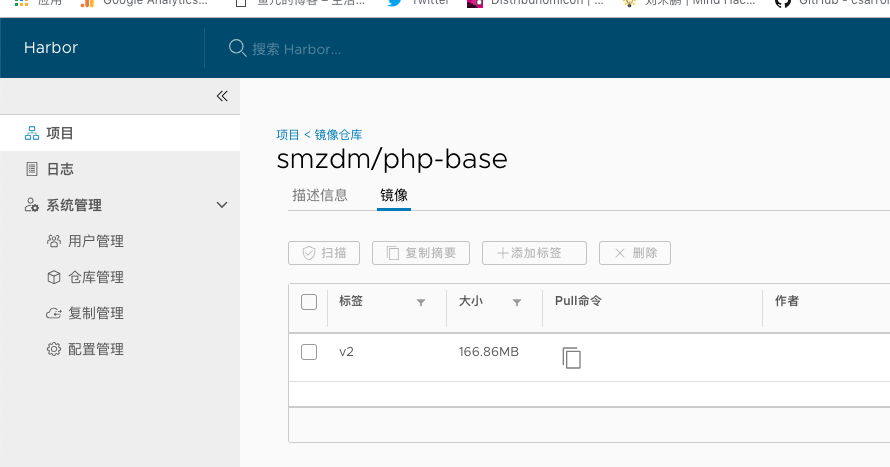
### 上传镜像

我自己搭了harbor私有仓库，所以我把镜像改改名推上去（没有Harbor的直接推到docker hub就行）：

docker tag php-base:v2 harbor.smzdm.com/smzdm/php-base:v2

docker push harbor.smzdm.com/smzdm/php-base:v2

这样harbor仓库中就有项目了：



我们可以把本地的镜像删了，拉一个下来试试：

docker rmi harbor.smzdm.com/smzdm/php-base:v2

docker pull harbor.smzdm.com/smzdm/php-base:v2

重新运行一个容器

docker run -d harbor.smzdm.com/smzdm/php-base:v2 sh -c 'while true; do sleep 1; done'

连接上去：

docker exec -it e9eb132e446cf37679220da9be35a40ad8420be57b582328a3b7664ba745f02c /bin/bash

启动nginx和php：

/etc/init.d/nginx start

/etc/init.d/php-fpm start

访问测试：

curl localhost:809

## 创建PHP测试项目

现在申请一个github项目，里面放一个index.php文件，然后写好build.sh和deploy.sh脚本。

我的PHP项目地址：

<https://github.com/owenliang/jenkins_test_project>

在里面放一个index.php，是我们唯一的项目代码：

<?php

echo 'hello jenkins!';

然后编写一个nginx配置test.conf，后续要放到/etc/nginx/conf.d/test.conf：

server {

listen 809;

server\_name test.smzdm.com;

root /data/webroot/phpsrc/test;

charset utf-8;

location / {

index index.php;

if (!-e $request\_filename){

rewrite ^(.\*)$ /index.php last;

break;

}

}

location ~ \.php$ {

fastcgi\_index index.php;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

fastcgi\_connect\_timeout 300;

fastcgi\_send\_timeout 300;

fastcgi\_read\_timeout 300;

fastcgi\_buffer\_size 128k;

fastcgi\_buffers 32 32k;

include fastcgi\_params;

port\_in\_redirect off;

fastcgi\_pass 127.0.0.1:9000;

}

}

### build.sh

编写build.sh编译脚本：

#!/bin/bash

# 该PHP项目最终要部署到/data/webroot/phpsrc/test下面

rm -rf deployment

# 代码

mkdir -p deployment/code

cp index.php deployment/code

cp test.conf deployment/

# 部署脚本

cp deploy.sh deployment/

# 打包目录

cd deployment

tar czvf ../deploy.tar.gz \*

cd -

# 删除临时目录

rm -rf deployment

Jenkins pipeline会先checkout项目代码，然后执行其中的build.sh来自定义构建项目，build.sh产出一个结果叫做deploy.tar.gz，由pipeline拿来继续做docker镜像的构建。

### deploy.sh

编写deploy.sh部署脚本：

#!/bin/bash

# 代码部署到test目录

cp -r code/\* /data/webroot/phpsrc/test

# 部署nginx配置

cp test.conf /etc/nginx/conf.d/

Jenkins pipeline在build.sh之后，会基于dockerfile构建镜像，即首先解压deploy.tar.gz到之前制作的base镜像某个目录下，并指定容器启动命令为：先执行deploy.sh，再拉起php-fpm，最后拉起nginx。

至此，我们的PHP测试项目就完成了，包含了index.php，build.sh，

deploy.sh，把它们上传到github。

接下来要干嘛呢？就是编写dockerfile，把base镜像和deploy.tar.gz打包到一起，生成后续用来上线用的docker镜像文件。

## 构建镜像

假设jenkins pipeline执行了build.sh得到了deploy.tar.gz，接下来要把部署文件打包到base image里，这样还不足够，我们还需要指定容器ENTRY命令，也就是容器启动后执行deploy.sh，以及拉起php和nginx。

为什么不在build镜像阶段执行deploy.sh呢？因为deploy.sh在我的定义中是一个运行时的回调，很有可能它还要启动点什么别的东西，所以需要放在容器的启动命令中执行。

所以，我们先写dockerfile并做好测试，只要dockerfile好用，我们后面要做的无非就是在jenkins去执行dockerfile完成构建而已。

### 编写dockerfile

我们新建一个github项目，用来存放jenkins构建项目过程中要用到的文件： <https://github.com/owenliang/jenkins_tools.git> ，现在我们要保存的就是php类型项目的通用dockerfile文件：

# PHP项目 - 通用dockerfile

# 包含PHP和NGINX环境的基础镜像

FROM harbor.smzdm.com/smzdm/php-base:v2

# 接下来的构建工作目录，是容器内的

WORKDIR /root/deploy

# 把build.sh产生的部署包解压到WORKDIR目录

ADD ["deploy.tar.gz", "./"]

# 容器启动命令

ENTRYPOINT sh deploy.sh && /etc/init.d/php-fpm start && /etc/init.d/nginx start && while true; do sleep 1; done

构建时会把deploy.tar.gz通过ADD命令，解压到镜像的/root/deploy目录。

启动容器时，会执行/root/deploy下的deploy.sh，并且拉起php-fpm和Nginx，最后while sleep保证容器不退出，除了deploy.sh的命令可以放在一个脚本例，一起ADD到容器里比较好。

### 测试构建

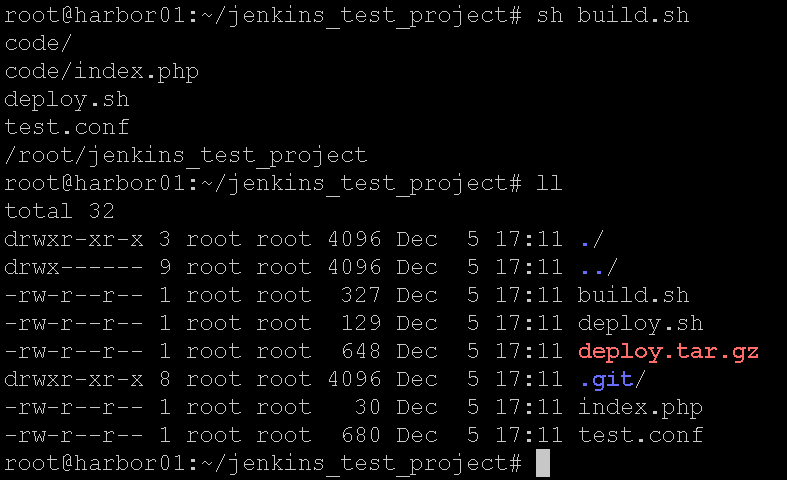
我们可以直接拿着这个dockerfile，以及build.sh出来的deploy.tar.gz做个实验：

在jenkins机器上，把2个github项目都拖下来：

git clone <https://github.com/owenliang/jenkins_test_project.git>

git clone <https://github.com/owenliang/jenkins_tools.git>

进入test\_project做build.sh，生成deploy.tar.gz：

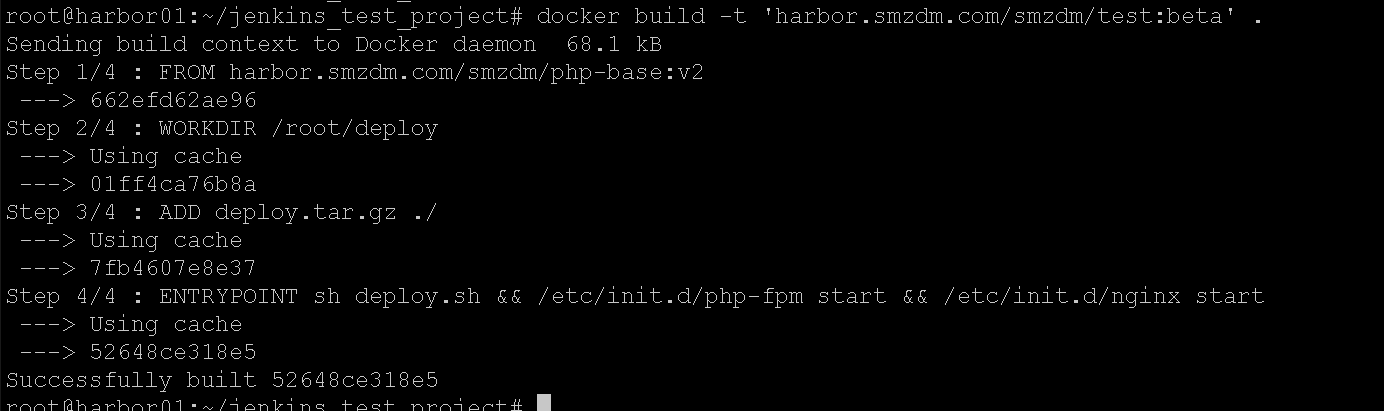


把php通用的dockerfile拷贝到当前目录下：

cp ../jenkins\_tools/dockerfile-php ./Dockerfile

然后build构建（-t指定生成镜像的名字，在jenkins中应该使用”项目名:构建版本”这样的结构; 后面的.是build的宿主机上下文，例如COPY命令会从这个目录寻找文件）：

docker build -t 'harbor.smzdm.com/smzdm/test:beta' .



可以查看镜像：

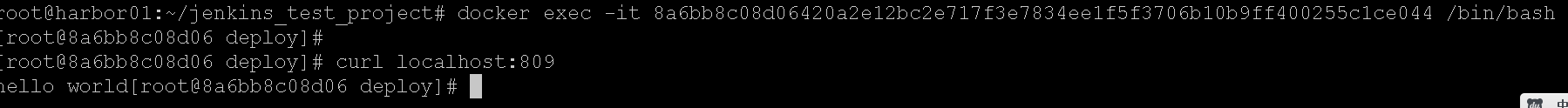


启动容器：

root@harbor01:~/jenkins\_test\_project# docker run -d harbor.smzdm.com/smzdm/test:beta

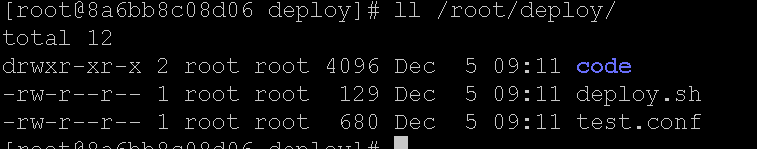
8a6bb8c08d06420a2e12bc2e717f3e7834ee1f5f3706b10b9ff400255c1ce044

然后连接上去：

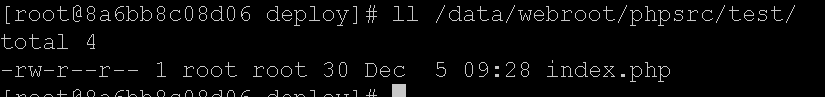


请求成功！

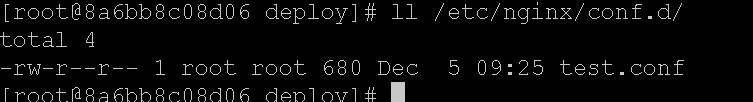
下面是容器内的WORKDIR工作目录，可见代码被解压到镜像里了：



Webroot下部署了test项目：

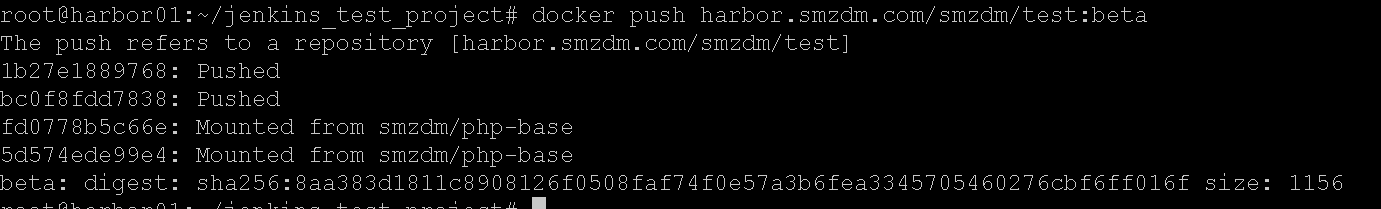


Nginx配置：



### 推送镜像

Jenkins构建完镜像，下一步肯定是把image传到harbor仓库里，等待上线使用了，这一步通过docker push就可以搞定。

未来可以在发布系统中，调用harbor的API获取镜像列表，从而发起上线，这是后话。

推送瞬间完成，为什么这么快呢？ 因为docker是层叠文件系统，我们这个镜像只是多了一层代码，所以只需要把新增的layer上传harbor，所以很快，也很节约仓库的磁盘。

## 基于pipeline自动构建

我们使用Jenkins一定要考虑如何N个项目如何统一配置的问题，否则一个项目一个pipeline配置，一旦要调整构建流程，每个项目的pipeline文件都得改一遍，显然是扯淡。

也就是说，我们整个团队应该只维护一份Pipeline的配置文件，所有同类型的项目构建流程都是一样的。

不同项目可以通过params参数化的方式对公共pipeline流程进行干预控制，从而实现一套配置N个项目的效果。

美团有一些说明：<https://tech.meituan.com/erp_cd_jenkins_pipeline.html>

### 搭建骨架

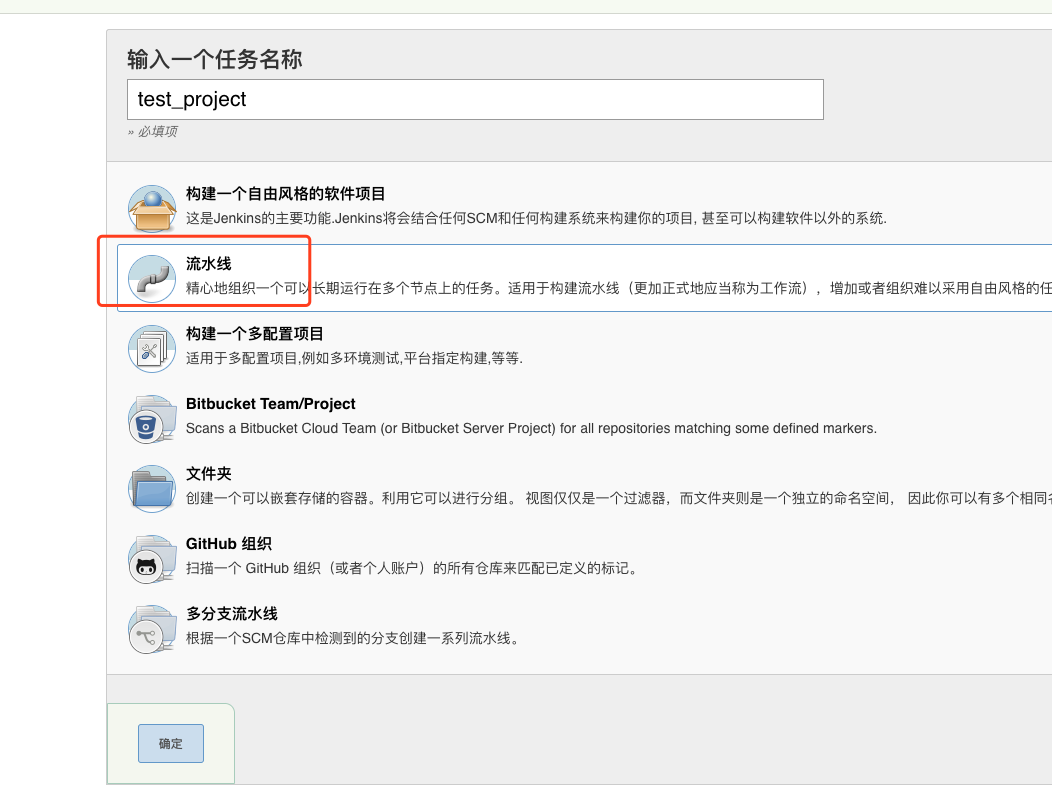
我之前有一个<https://github.com/owenliang/jenkins_tools> 项目，它除了存放dockerfile用于构建镜像之外，还应该存放我们的公共pipeline配置，也就是一个Jenkinsfile。

我们后续接入jenkins的所有项目，都会使用这个jenkinks\_tools项目中的Jenkinsfile来运行pipeline流程。不同项目要做的事情，就是配置自己的参数来指导pipeline流程。

创建任务

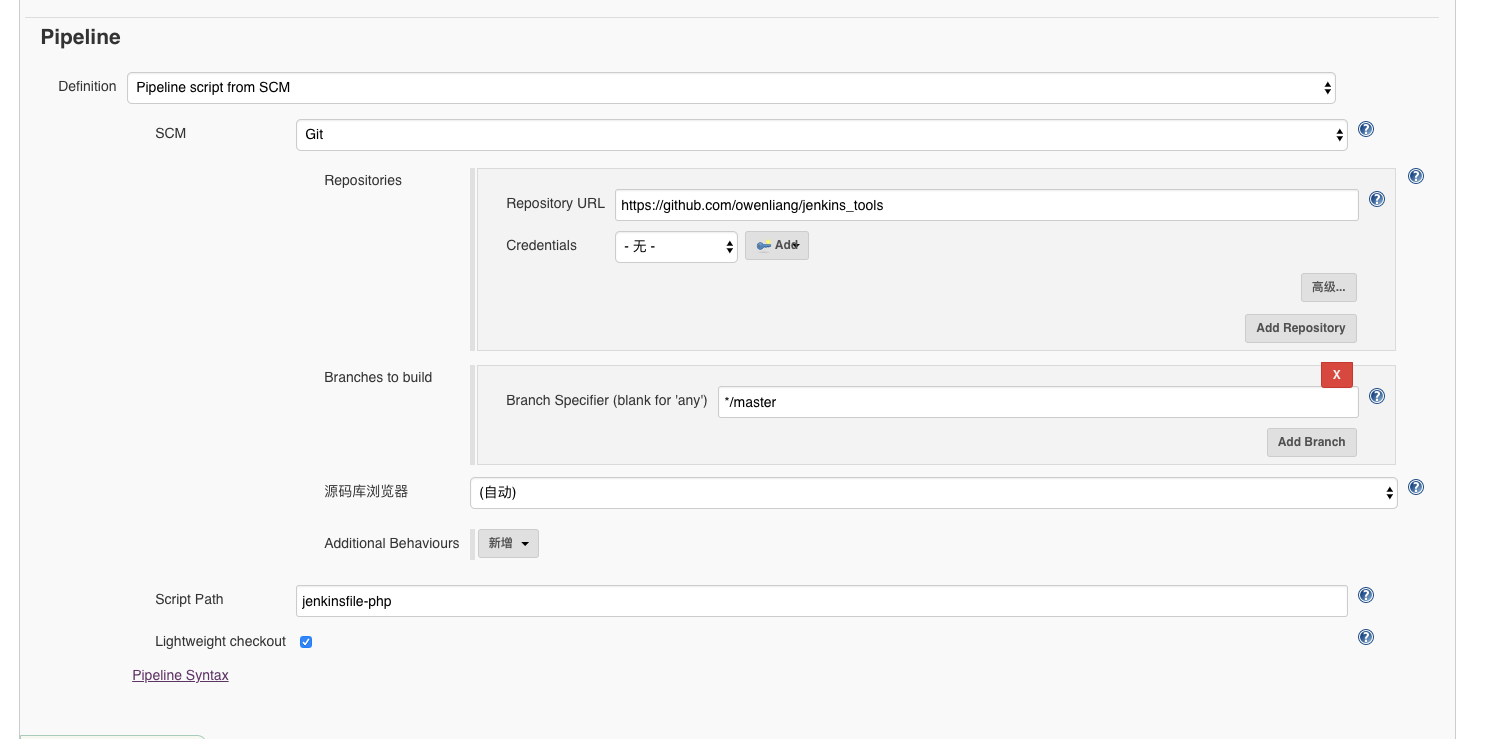


我们要构建的项目是<https://github.com/owenliang/jenkins_test_project>，所以我起名叫做：test\_project吧：



Jenkins支持指定Jenkinsfile从哪里加载，触发时jenkins会首先帮我们把jenkinsfile下载回来，而jenkinsfile里面就定义了通用的pipeline流程。

我现在假设已经写好了jenkinsfile-php放在了<https://github.com/owenliang/jenkins_tools> 项目里，那么就这样配置一下：



Credentials是可以配置下载git代码的账号与密码，我这里github是开放的，所以就不配置了。

Branches指定了构建哪个分支，指定master即可，我们所有的jenkinsfile都放在jenkins\_tools的Master里面。

保存一下上述配置，然后我们编写一个没有任何功能的jenkinsfile-php上传到jenkins\_tools项目下：

pipeline {

agent any

stages {

stage("下载代码") {

steps {

echo "下载代码"

}

}

stage("构建应用") {

steps {

echo "构建应用"

}

}

stage("打包镜像") {

steps {

echo "打包镜像"

}

}

stage("推送镜像") {

steps {

echo "推送镜像"

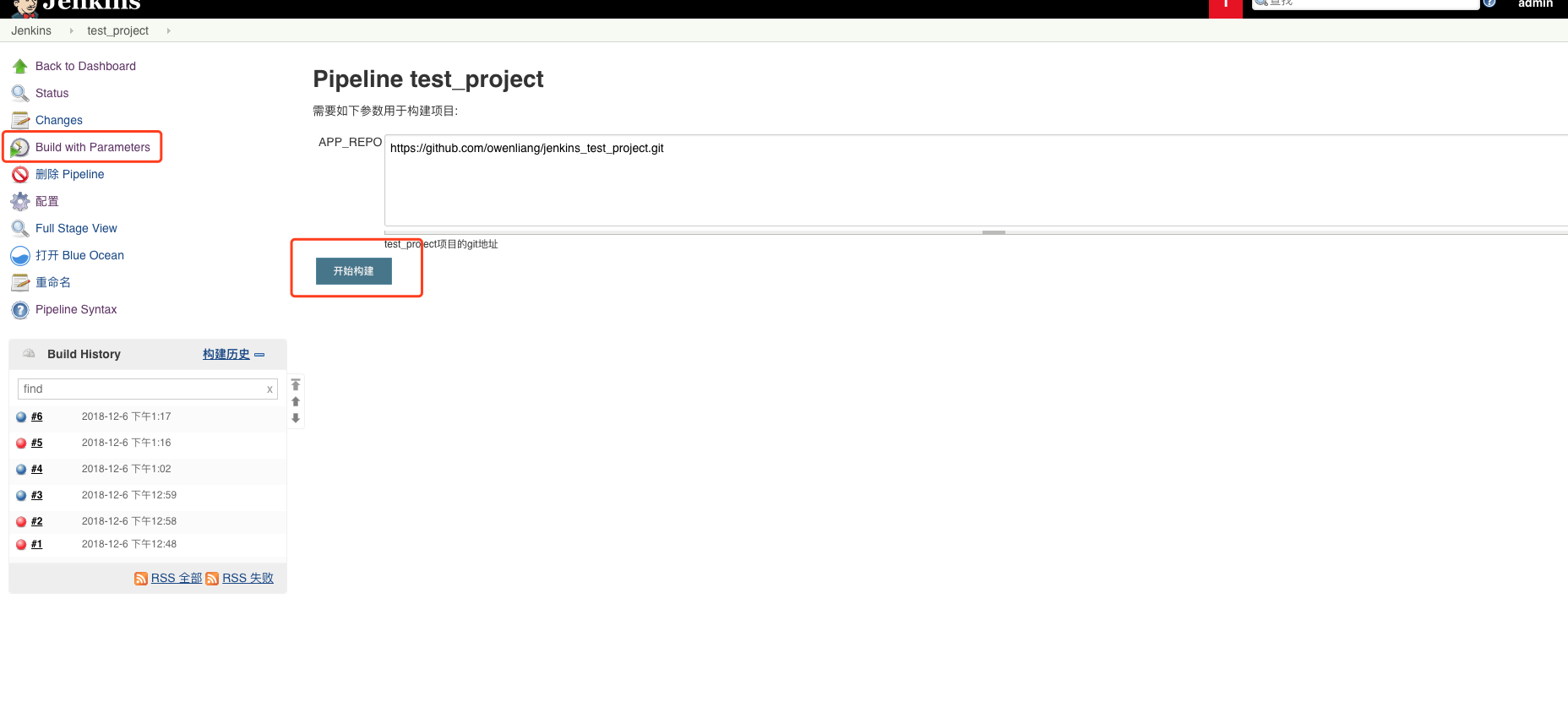
}

}

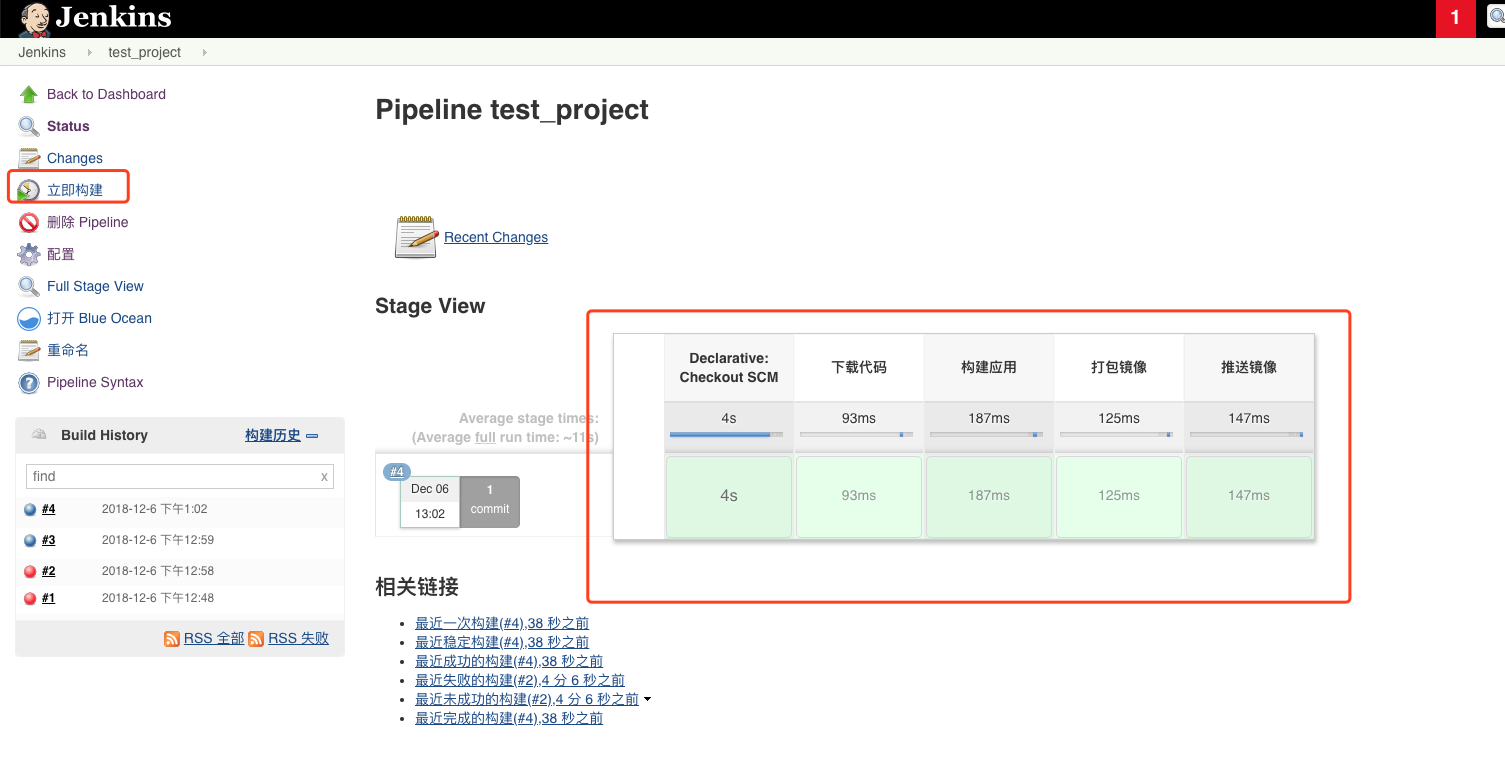
}

}

然后触发一下test\_project的构建流程：



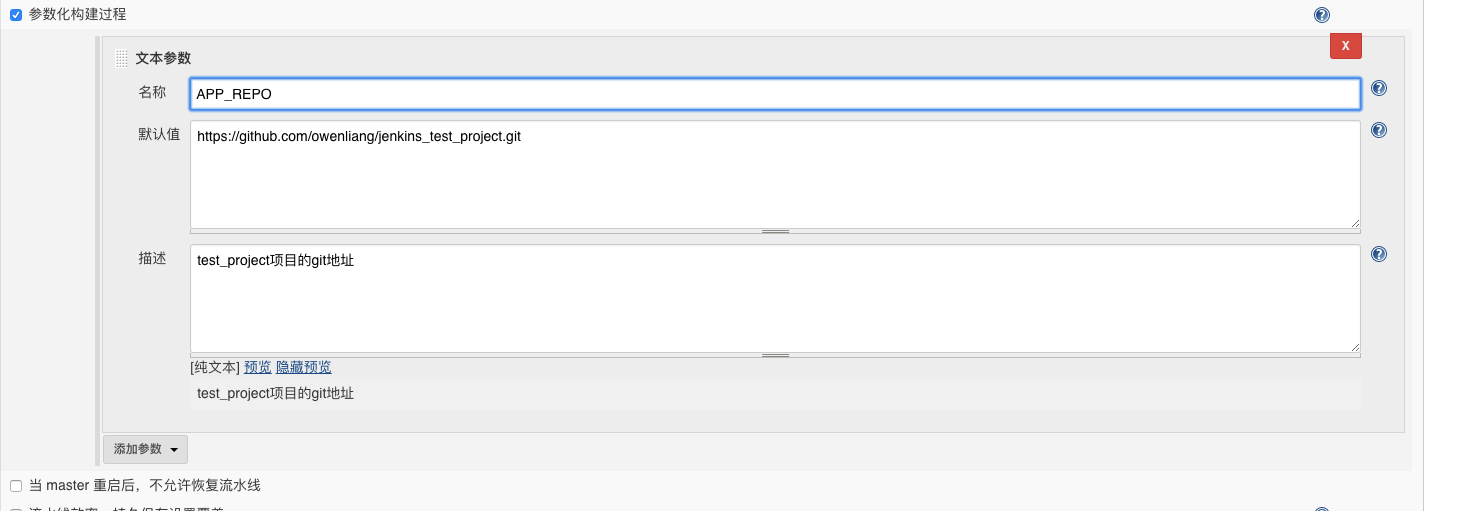
看到结果成功：



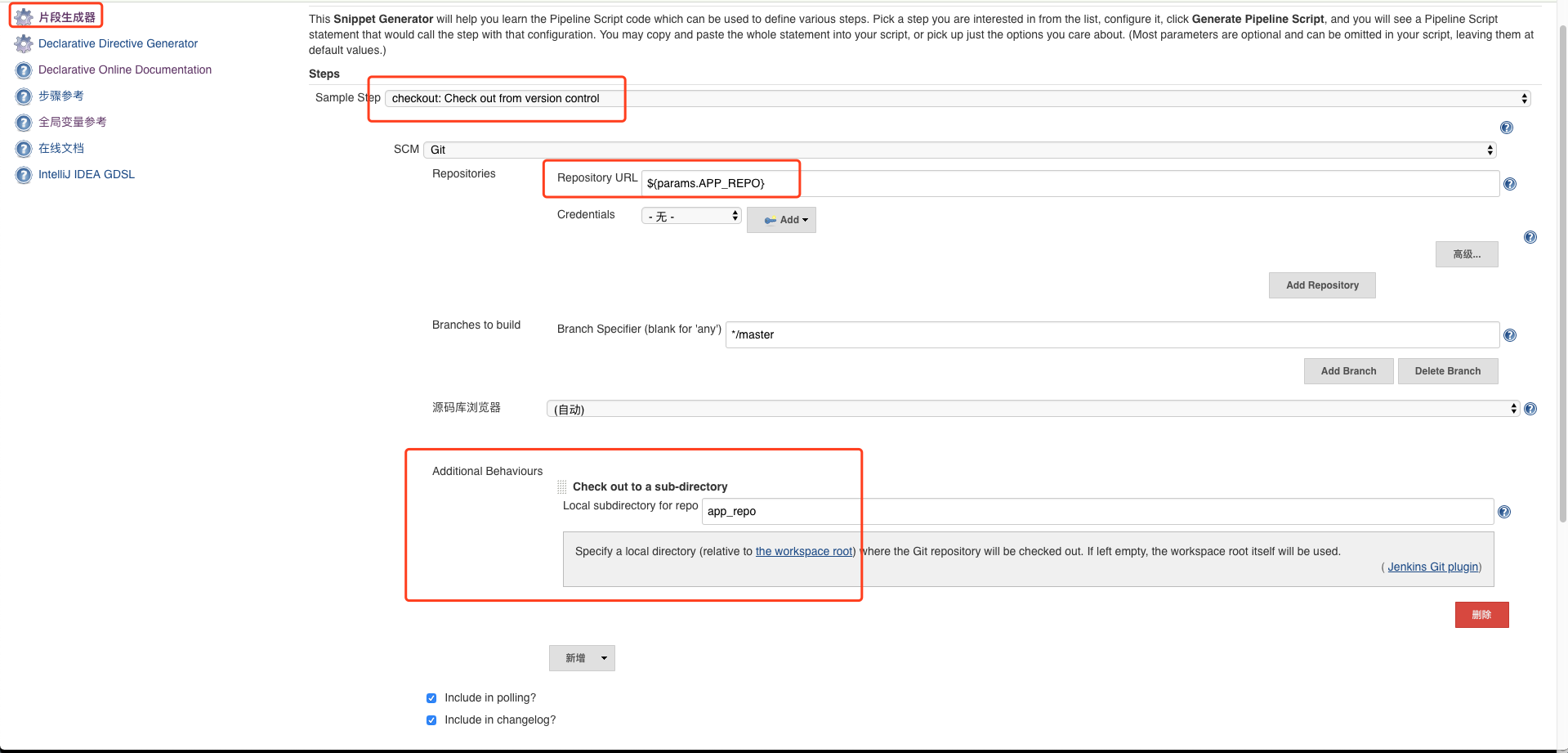
Pipeline骨架就搭建好了，接下来就是填充每个stage的细节工作了。

### Stage 下载代码

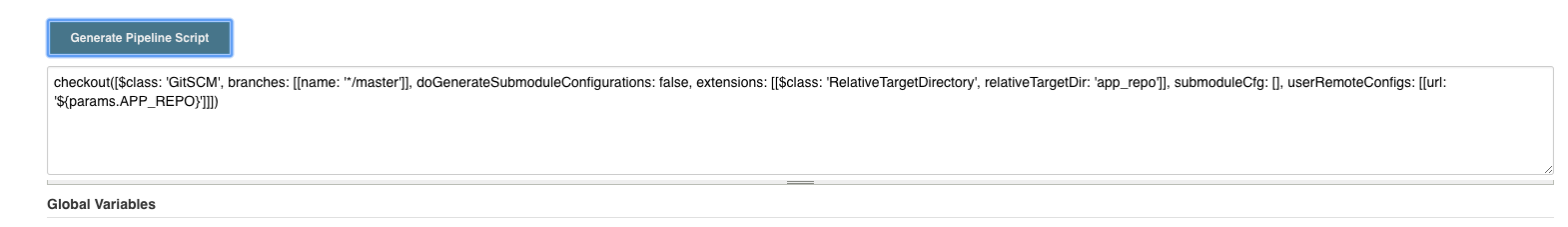
编辑test\_project的配置，添加一个params记录该项目的源码下载路径：



我们要让pipeline在stage(“下载代码”)阶段，使用git命令把params.APP\_REPO下载回来，所以打开jenkins的：<http://172.18.10.236:8080/job/test_project/pipeline-syntax/> ，点击片段生成器，点击checkout指令，填入必要信息，点击生成：



自动生成了这段指令：



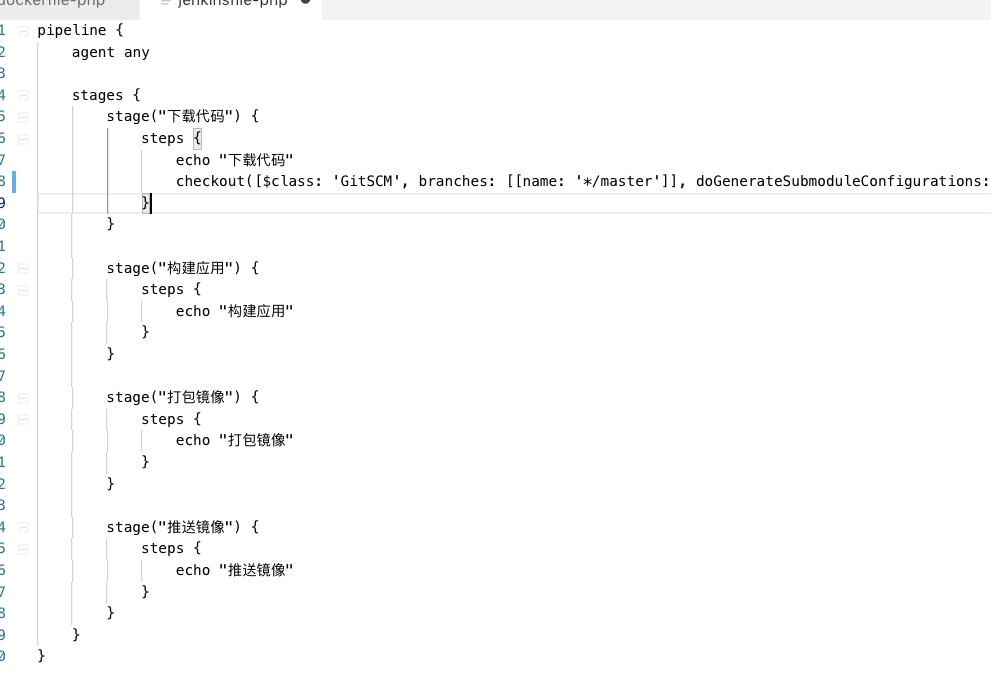
checkout([$class: 'GitSCM', branches: [[name: '\*/master']], doGenerateSubmoduleConfigurations: false, extensions: [[$class: 'RelativeTargetDirectory', relativeTargetDir: 'app\_repo']], submoduleCfg: [], userRemoteConfigs: [[url: '${params.APP\_REPO}']]])

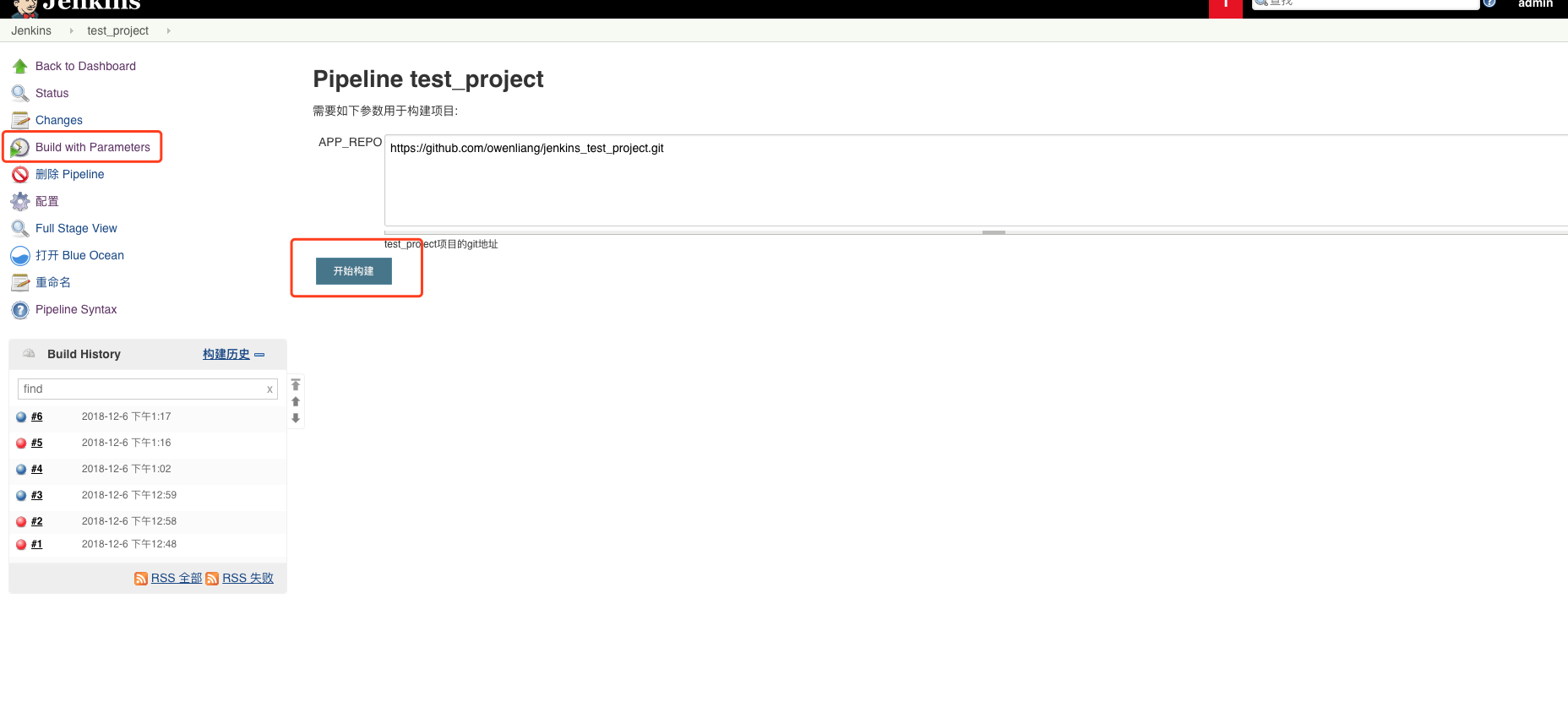
图形化生成非常方便，这段指令放在pipeline step里就可以完成代码下载，并且代码被下载到子目录app\_repo中。

其中需要改一个地方就是${params.APP\_REPO}这个参数，得改用双引号，否则pipeline不会替换为真实的值：

checkout([$class: 'GitSCM', branches: [[name: '\*/master']], doGenerateSubmoduleConfigurations: false, extensions: [[$class: 'RelativeTargetDirectory', relativeTargetDir: 'app\_repo']], submoduleCfg: [], userRemoteConfigs: [[url: "${params.APP\_REPO}"]]])

我们把它贴到pipeline中，把pipeline上传github：



然后重新执行构建：

观察pipeline的控制台输出：

可见，pipeline在app\_repo目录下拉取了github上的jenkins\_test\_project项目代码，放在了构建目录下（workspace/test\_project/app\_repo）。

### Stage 构建应用

很简单，我们给stage(“构建应用”)增加相关step：

pipeline {

agent any

stages {

stage("下载代码") {

steps {

echo "下载代码"

checkout([$class: 'GitSCM', branches: [[name: '\*/master']], doGenerateSubmoduleConfigurations: false, extensions: [[$class: 'RelativeTargetDirectory', relativeTargetDir: 'app\_repo']], submoduleCfg: [], userRemoteConfigs: [[url: "${params.APP\_REPO}"]]])

}

}

stage("构建应用") {

steps {

echo "构建应用"

sh "cd app\_repo && sh build.sh"

sh "cp app\_repo/deploy.tar.gz ./"

}

}

stage("打包镜像") {

steps {

echo "打包镜像"

}

}

stage("推送镜像") {

steps {

echo "推送镜像"

}

}

}

}

第一步进入app\_repo目录执行build.sh，第二步把产出的deploy.tar.gz拷贝到构建根目录下。

每一个step，jenkins都会重置当前工作目录为构建根目录，这个路径关系我们要搞清楚。



上述就是构建后的效果，jenkinsfile-php就是当前pipeline的配置，app\_repo是项目源码，deploy.tar.gz是app\_repo里build.sh出来的，dockerfile-php是我们接下来要用来打镜像的。

### Stage 打包镜像

继续修改pipeline：

pipeline {

agent any

stages {

stage("下载代码") {

steps {

echo "下载代码"

checkout([$class: 'GitSCM', branches: [[name: '\*/master']], doGenerateSubmoduleConfigurations: false, extensions: [[$class: 'RelativeTargetDirectory', relativeTargetDir: 'app\_repo']], submoduleCfg: [], userRemoteConfigs: [[url: "${params.APP\_REPO}"]]])

}

}

stage("构建应用") {

steps {

echo "构建应用"

sh "cd app\_repo && sh build.sh"

sh "cp app\_repo/deploy.tar.gz ./"

}

}

stage("打包镜像") {

steps {

echo "打包镜像"

sh "cp dockerfile-php Dockerfile"

sh "docker build -t 'harbor.smzdm.com/smzdm/${currentBuild.projectName}:${currentBuild.id}-${currentBuild.startTimeInMillis}' ."

}

}

stage("推送镜像") {

steps {

echo "推送镜像"

}

}

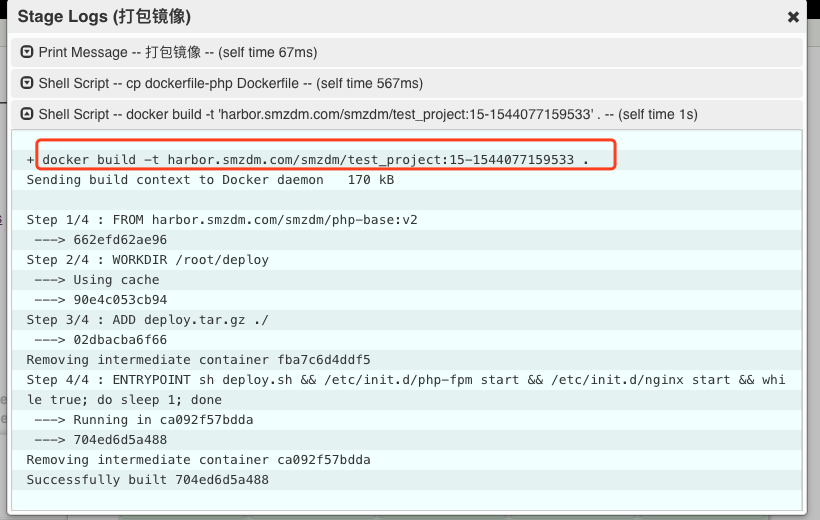
}

}

在构建根目录下准备好对应的Dockerfile，然后build镜像，我们之前写的dockerfile会把deploy.tar.gz打包到镜像里。

-t指定生成的镜像名字，镜像最终要推送到私有仓库harbor.smzdm.com，属于smzdm项目，镜像名是currentBuild.projectName，也就是jenkins配置的pipeline项目名：test\_project。

currentBuild .id是每次构建任务的唯一标识，currentBuild .startTimeInMills是构建开始的时间，这些都是jenkins内置变量，详细信息参考： <http://172.18.10.236:8080/pipeline-syntax/globals#currentBuild>



Docker build会根据dockerfile去私有仓库拉取镜像，我们可以提前在jenkins部署的机器上做好docker login登录，也可以在steps里配置一个docker login登录步骤，我在最后推送镜像的部分做一个演示。

### Stage 上传镜像

下面先演示一下如何通过命令行登录docker私有仓库。

正常来说，我们想访问docker仓库需要登录：

docker login harbor.smzdm.com -u admin -p Harbor12345

就可以登录到私有仓库，并且密码会被保存下来，后续就不用登录了：

cat ~/.docker/config.json

{

"auths": {

"harbor.smzdm.com": {

"auth": "YWRtaW46SGFyYm9yMTIzNDU="

}

}

如果我们没有在机器上登录过docker仓库，那么可以把上述登录命令写在pipeline的流程里，就像这样：

stage("推送镜像") {

steps {

echo "推送镜像"

sh "docker login harbor.smzdm.com -u admin -p Harbor12345"

sh "docker push 'harbor.smzdm.com/smzdm/${currentBuild.projectName}:${currentBuild.id}-${currentBuild.startTimeInMillis}' "

}

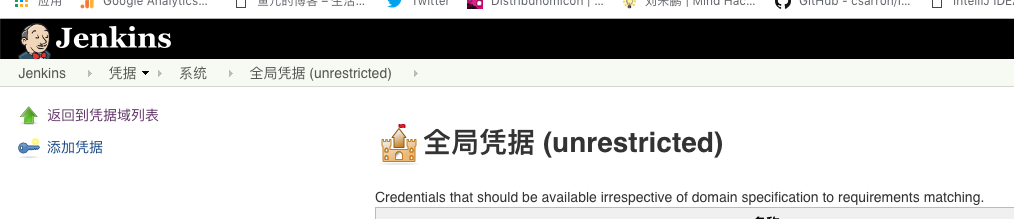
}

我们的pipeline配置是上传在github的，我肯定不希望密码如此暴露给公众，怎么办呢？

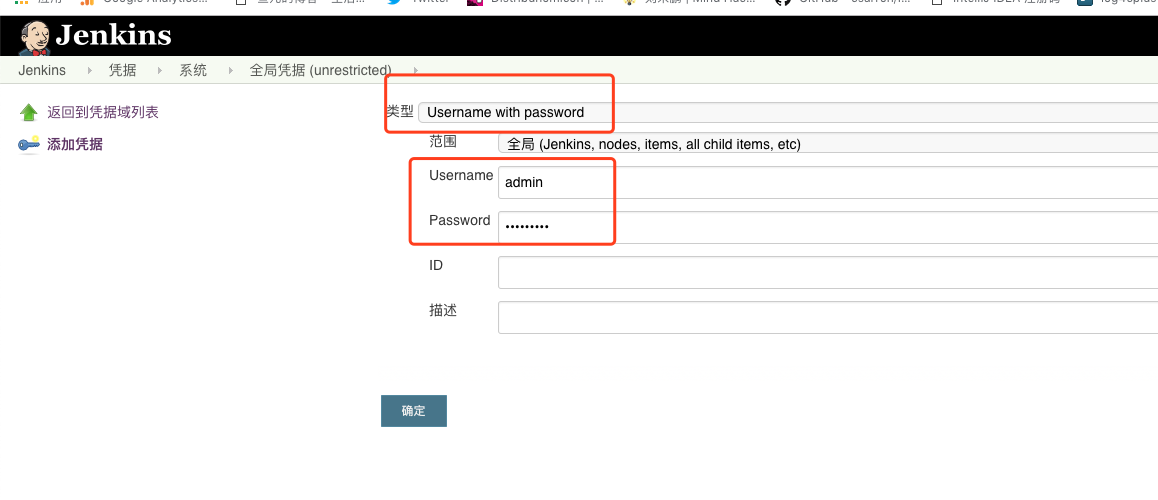
可以把帐号密码保存在jenkins里，jenkins可以通过env环境变量的形式把帐号密码注入到pipeline中使用。

我们先把帐号密码配置到jenkins中：

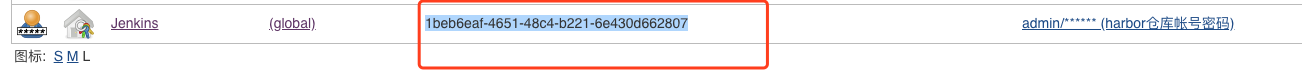
依次点击 凭据 -》 系统 => 全局凭据：



点击左侧添加凭据：



把harbor仓库的账号密码配置一下，得到凭据的唯一ID：



我们可以在pipeline中引入密钥，机制是通过environment环境变量以及一个方法叫做credentials：

<https://jenkins.io/doc/book/pipeline/syntax/#environment>

所以最终jenkinsfile如下：

pipeline {

agent any

stages {

stage("下载代码") {

steps {

echo "下载代码"

checkout([$class: 'GitSCM', branches: [[name: '\*/master']], doGenerateSubmoduleConfigurations: false, extensions: [[$class: 'RelativeTargetDirectory', relativeTargetDir: 'app\_repo']], submoduleCfg: [], userRemoteConfigs: [[url: "${params.APP\_REPO}"]]])

}

}

stage("构建应用") {

steps {

echo "构建应用"

sh "cd app\_repo && sh build.sh"

sh "cp app\_repo/deploy.tar.gz ./"

}

}

stage("打包镜像") {

environment {

HARBOR\_ACCOUNT = credentials('1beb6eaf-4651-48c4-b221-6e430d662807')

}

steps {

echo "打包镜像"

sh "cp dockerfile-php Dockerfile"

sh "docker login harbor.smzdm.com -u ${HARBOR\_ACCOUNT\_USR} -p ${HARBOR\_ACCOUNT\_PSW}"

sh "docker build -t 'harbor.smzdm.com/smzdm/${currentBuild.projectName}:${currentBuild.id}-${currentBuild.startTimeInMillis}' ."

}

}

stage("推送镜像") {

steps {

echo "推送镜像"

sh "docker push 'harbor.smzdm.com/smzdm/${currentBuild.projectName}:${currentBuild.id}-${currentBuild.startTimeInMillis}' "

sh "docker rmi 'harbor.smzdm.com/smzdm/${currentBuild.projectName}:${currentBuild.id}-${currentBuild.startTimeInMillis}' "

}

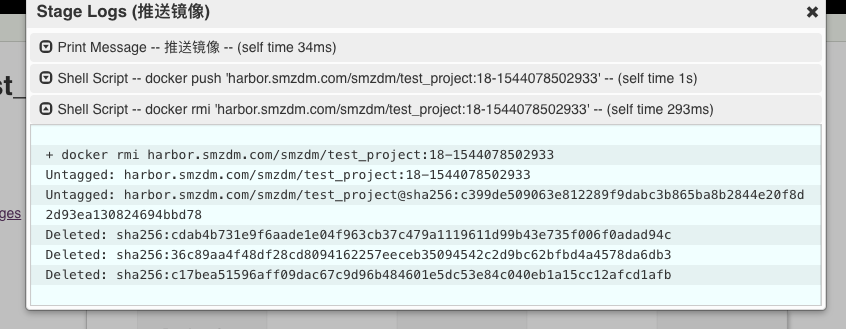
}

}

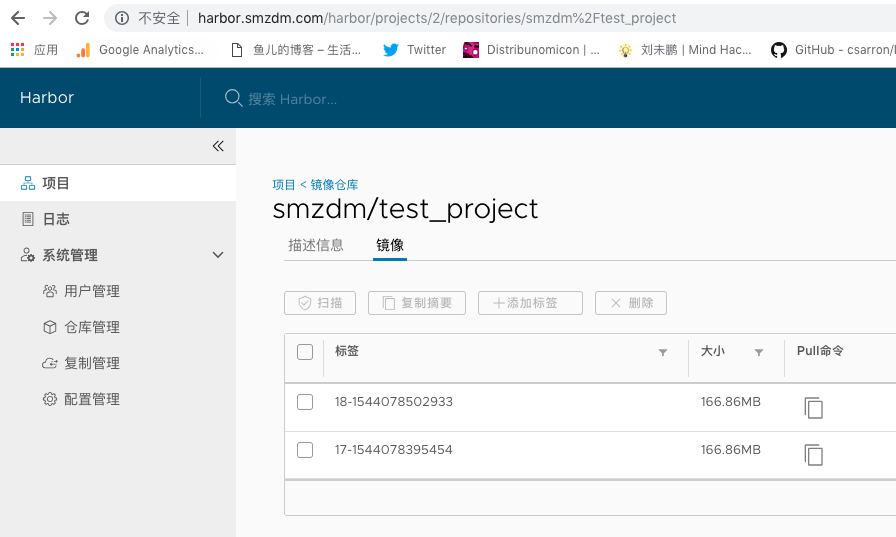
}

引入密钥的变量加上\_USR和\_PSW后缀就是帐号和密码了，在镜像推送到harbor后我还rmi把本机构建的镜像删除了，免得占用jenkins的磁盘。





harbor中的镜像如下：



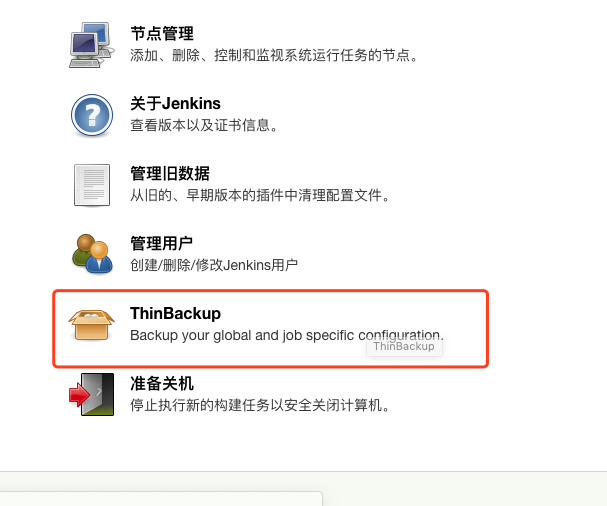
这样就没问题了。

# 其他

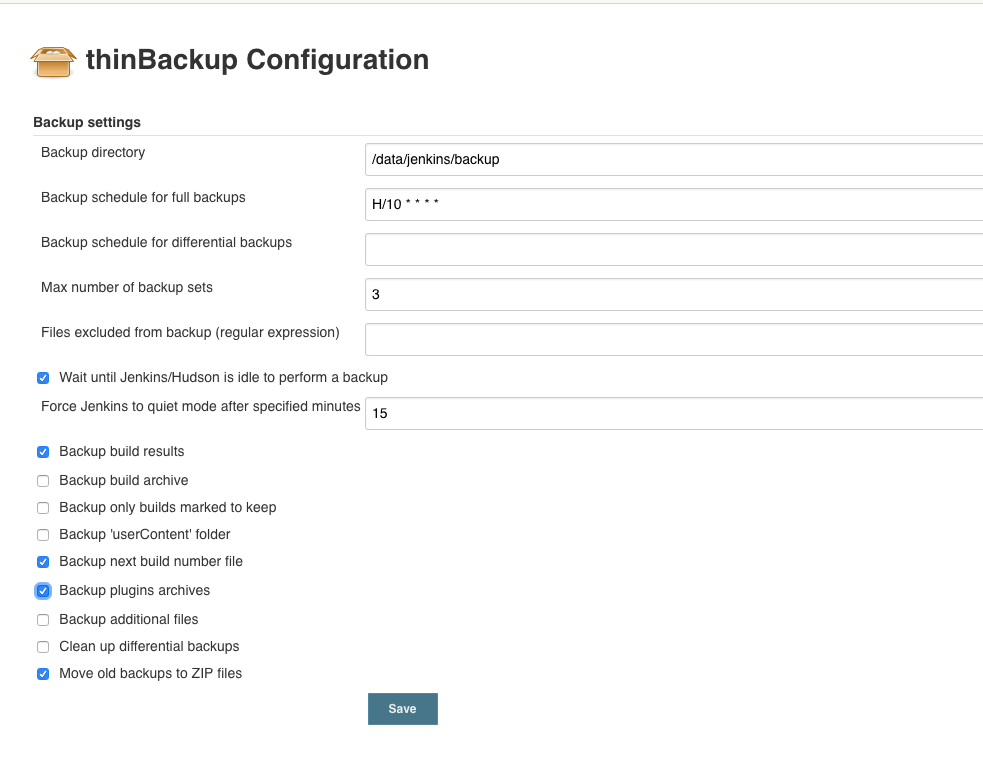
## 高可用

Jenkins master主要就是各种配置不希望丢失，所以安装一个ThinBackup插件（安装报错就取消掉插件镜像加速），做定时的全量配置备份即可。

安装后从系统管理进入：



然后配置定时全量备份：



每10分钟一次，如果jenkinks繁忙则可以延长至15分钟一次。

jenkins数据保存到目录：/data/jenkins/backup ，最多保存3份全量历史，备份会压缩为zip。

全局构建的id号也最好备份一下，丢了怪可惜的。

然后插件也备份一下，方便恢复！

老的备份会被zip压缩，节约空间！

记得把目录创建出来！

定期的把这个目录下的zip文件，拷贝走，备份一下就行！

Jenkins配置本身不是一个常修改的东西，问题不大。

## Shell脚本与jenkins

可以把大部分pipeline逻辑写在shell脚本里封装起来，但是测试shell脚本只能访问到pipeline中的env和params，其他的参数想传进来除了给shell脚本传参外，可以使用withEnv{}指令，通过env把更多的变量带到shell里访问。

另外，shell脚本肯定也想往jenkins传递变量，我觉得简单做法就是临时写到一个文件里，后续步骤source命令引进来就好了，不需要依赖额外的插件。