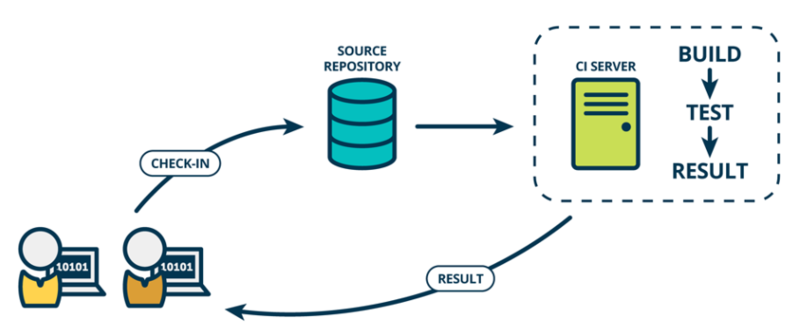
# 1、Jenkins概述与环境搭建

## 1.1 了解Jenkins

持续集成即 Continuous integration（CI） 是一种软件开发实践，即团队开发成员经常集成他们的工作，每次集成都通过自动化的构建（包括编译，发布，自动化测试）来验证，从而尽快地发现集成错误，让团队能够更高效的开发软件。



**持续集成要点**

* 统一的代码库(git)
* 统一的依赖包管理(nexus)
* 测试自动化
* 构建全自动化(maven)
* 部署自动化
* 可追踪的集成记录

jenkins 就是为了满足上述持续集成的要点而设计的一款工具，其主体框架采用JAVA开发，实质内部功能都是由各种插件实现，极大提高了系统的扩展性。其不仅可以满足JAVA系统的集成，也可以实现PHP等语言的集成发布。通过其pipeline 插件，用户可以随自己需要定制集成流程。

## 1.2 安装Jenkins

**下载：**jenkins 支持Docker、yum、msi 等安装，在这里推荐大家直接选择下载他对应的WAR包进行安装。<https://jenkins.io/download/>

**启动：**下载完成之后直接可通过 jar -jar 命令启动

java -jar jenkins.war --httpPort=8080

也可以将其放至到servlet容器（tomcat\jetty\jboss）中直接启动，无需过多的配置，一切插件化这是jenkins 比较优秀的设计。

**配置：**下载完成之后进入启动页(<http://127.0.0.1:8080/>) 会有一个 验证过程，验证码存储在 ${user\_home}\.jenkins\secrets\initialAdminPassword 中，接着就是进入安装插件页，选择默认即可，这个过程稍长。

## 1.3 常用插件下载

Jenkins需要maven与git环境，服务器需要先装好这两个工具

**插件下载**

**如果插件下载比较慢的话可以更换站点**

更换源 ->系统管理->管理插件->高级 ->升级站点

把：<http://updates.jenkins-ci.org/update-center.json>

换成：<http://mirror.esuni.jp/jenkins/updates/update-center.json>

镜像源查询：<http://mirrors.jenkins-ci.org/status.html>

基本插件列表

|  |  |
| --- | --- |
| **插件名称** | **插件描述** |
| **Maven Integration** | **maven 管理插件** |
| **Deploy to container** | **容器部署插件** |
| **Pipeline** | **管道集成插件** |
| **Email Extension** | **邮件通知插件** |
| **SSH** | **用于ssh 通信** |

## 1.4 Jenkins集成过程中的问题点

### 4.1 linux可以执行的脚本在Jenkins中不能执行



更多解决方案请参照：  
<https://blog.csdn.net/u011781521/article/details/80210985>

# 2、Jenkins可持续化集成

## 2.1 持续化集成完成的目标

持续集成是指软件个人研发的部分向软件整体部分交付，频繁进行集成以便更快地发现其中的错误。

需要达成的目标：

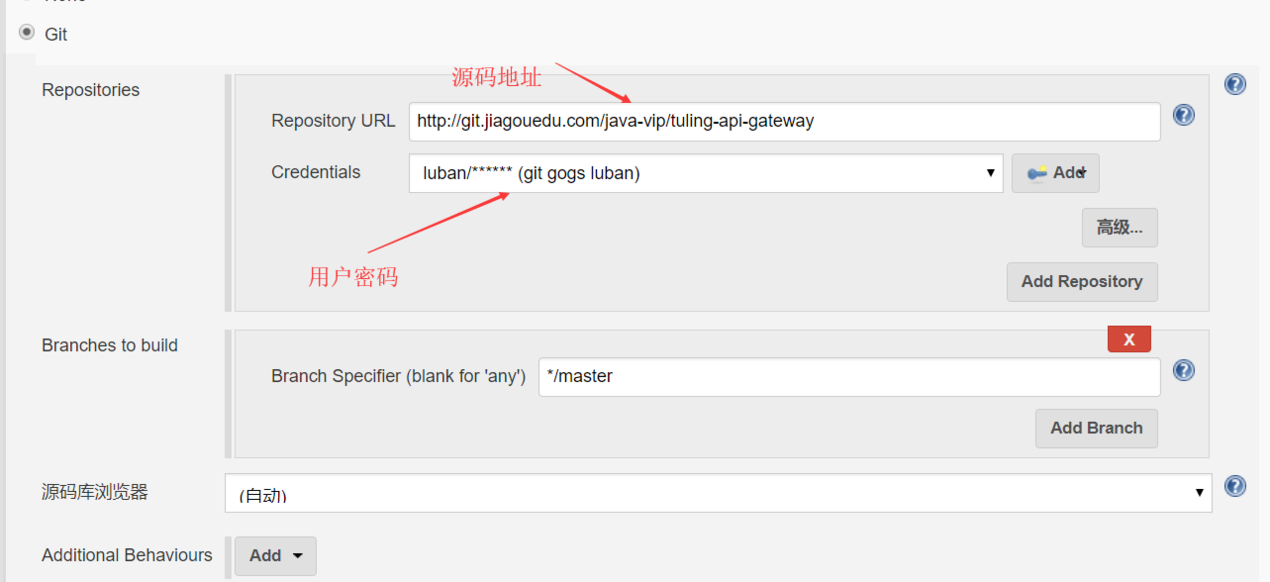
1. 自动基于分支构建项目
2. 构建好的项目自动部署至Tomcat容器（可选）
3. 构建好的项目自动上传至Nexus 私服存档
4. 保存构建历史记录，并可以下载历史记录

## 2.2 持续化集成的配置

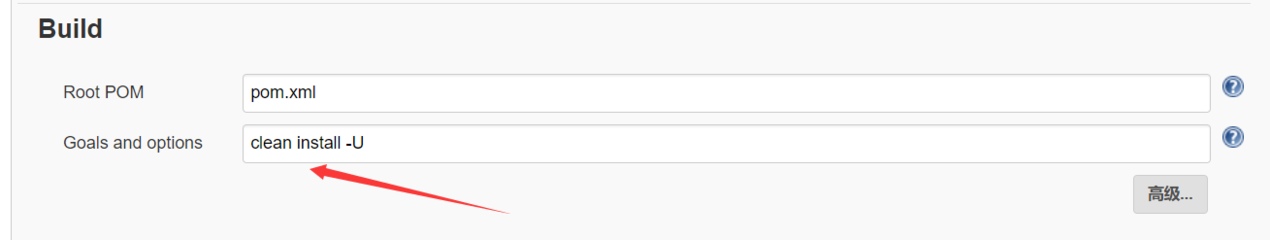
1、新建maven job



2、配置checkout源码



3、编写maven构建命令



4、自动部署至tomcat配置

添加构建后操作：Deploy war/ear to container 项目



自动部署的前提条件：

1. 需要下载 Deploy to container 插件
2. 设置Tomcat manager 用户和密码，以下配置加入至 Tomcat conf/tomcat-users.xml 中
3. tomcat webapp 中必须保留 manager 项目，在user-setting中配置

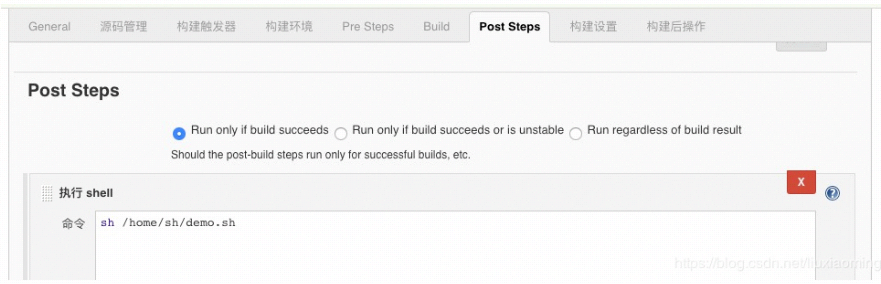
<role rolename="admin-gui"/>

<role rolename="manager-gui"/>

<role rolename="manager-script"/>

<user username="manager" password="manager" roles="manager-gui,manager-script"/>

<user username="admin" password="admin" roles="admin-gui,manager-gui"/>



Jar包部署方式，可以执行shell命令，也可以直接执行本地shell脚本

脚本内容

#!/bin/sh

#Copy jar to target path

BUILD\_ID=DONTKILLME #后台执行，必须添加，不然会不执行【nohup java -jar demo.jar --#server.port=8082 >demo.log 2>&1 &】

pid=$(ps -ef|grep demo.jar|grep -v grep | awk '{print $2}')

#copy jar 到启动目录

cp -r /var/lib/jenkins/workspace/demo/demo/target/demo.jar /home/jar/demo

# 关闭已经启动的jar进程

function stop(){

if [ -n "$pid" ]

then

echo "pid进程 :$pid"

kill -9 $pid

else

echo "进程没有启动"

fi

}

stop

sleep 5s

#发布jar服务

function start(){

cd /home/jar/demo

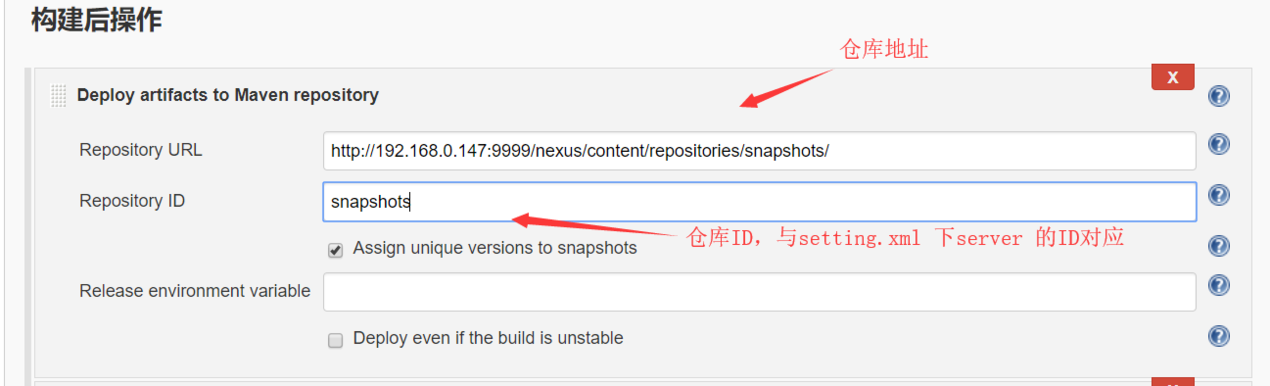
nohup java -jar demo.jar --server.port=8082 >demo.log 2>&1 &

}

start

**存档配置**

构建后操作添加 Deploy war/ear to a container 项目



配置setting.xml 用于获取上传至nexus 的权限

vim ~/.m2/settings.xml

# 添加一个 server id 与 存档配置当中的repository id 相对应。

 <server>

          <id>nexusReleases</id>

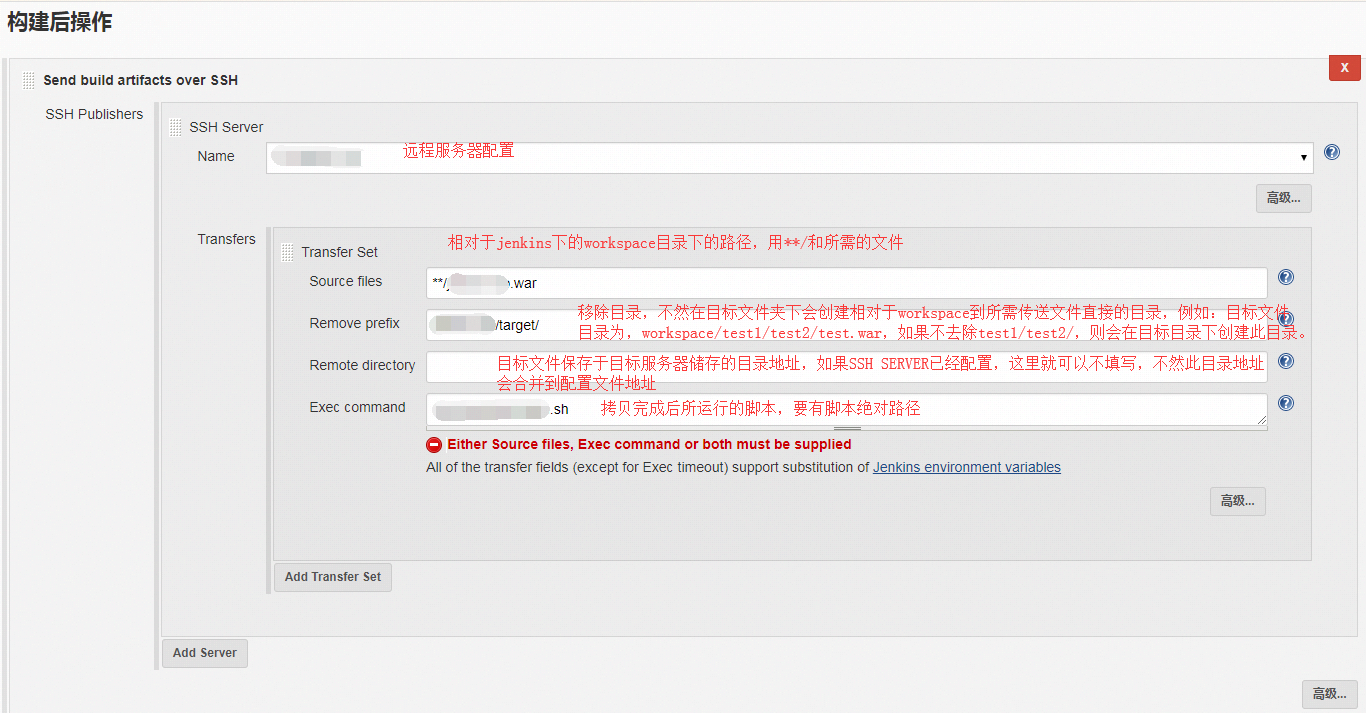
          <username>deployment</username>

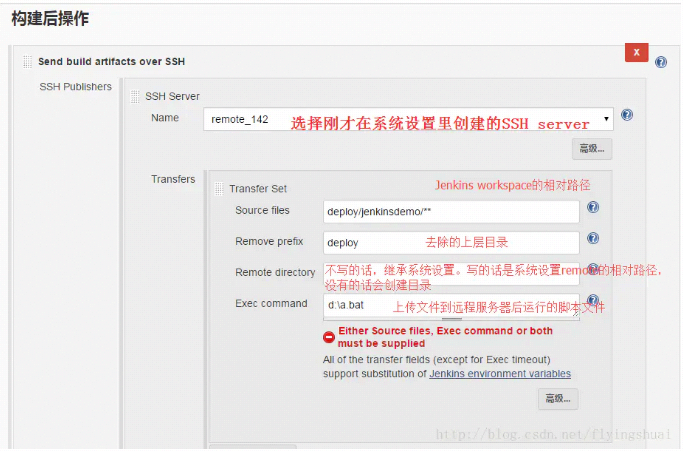
          <password>111111</password>

</server>

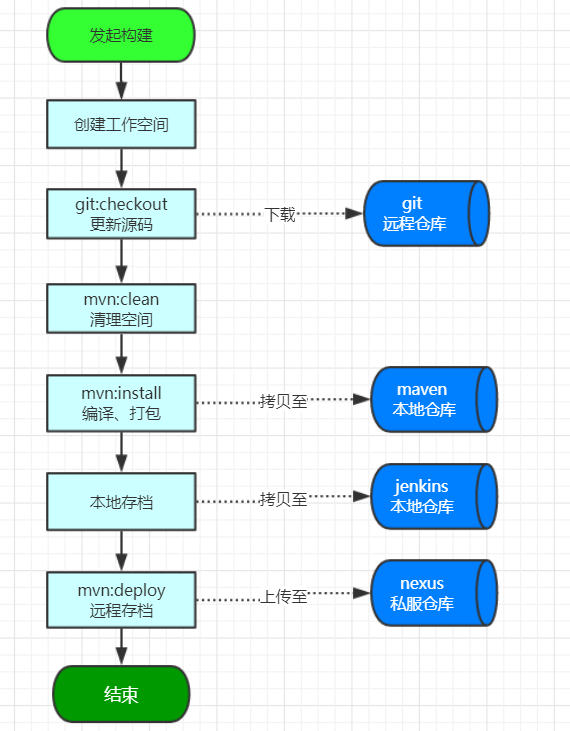
### 远程部署项目的方式：

在进行远程部署操作前，先要确保客户机能ssh 登录到远程机器。如果不知道SSH怎么登陆，请参考[http://blog.csdn.net/flyingshuai/article/details/72897692](https://link.jianshu.com?t=http%3A%2F%2Fblog.csdn.net%2Fflyingshuai%2Farticle%2Fdetails%2F72897692)





## 2.3 集成原理



# 3、Jenkins pipeline核心应用

## 3.1 pipeline概要

前面我们演示的 使用maven 来进行自动化构建，其流程分别是：构建环境准备==》源码下载 ==》构建 ==》存档 ==》部署。这是一种固化的构建流程，如果你们的需求是多个项目需要进行依赖构建这种复杂的构建场景 时该怎么办？jenkins pipeline 可以做到这一点。

Jenkins从根本上讲是一种支持多种自动化模式的自动化引擎。Pipeline在Jenkins上添加了一套强大的自动化工具，支持从简单的连续集成到全面的连续输送Pipeline的用例。用户可以基于他实现更为复杂的建模场景。

## 3.2 pipeline的基础语法

以下就是一个非常简单的 pipeline 脚本：

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'make'

}

}

stage('Test'){

steps {

sh 'make check'

junit 'reports/\*\*/\*.xml'

}

}

stage('Deploy') {

steps {

sh 'make publish'

}

}

}

}

agent  表示Jenkins应该为Pipeline的这一部分分配一个执行者和工作区。

stage  描述了这条Pipeline的一个阶段。

steps  描述了要在其中运行的步骤 stage

sh  执行给定的shell命令

junit 是由JUnit插件提供的 用于聚合测试报告的Pipeline步骤。

## 3.3 pipeline 演示

1. Jenkins 2.x或更高版本
2. Pipeline插件

可以通过以下任一方式创建基本Pipeline：

1. 直接在Jenkins网页界面中输入脚本。
2. 通过创建一个Jenkinsfile可以检入项目的源代码管理库。

用任一方法定义Pipeline的语法是一样的，从项目源码中检入jenkinsfile 文件会更方便一些。

Pipeline脚本可以通过在Jenkins页面生成，选择相应的指令及操作即可生成脚本，cope到文件中即可，例如此部分脚本([$class: 'GitSCM', branches: [[name: '\*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'gogs\_luban', url: 'http://git.jiagouedu.com/java-vip/tuling-api-gateway']]])

基于脚本构建 pipeline

pipeline {

agent any

stages {

stage('Checkout') {

steps {

echo 'Checkout'

stage('Build') {

steps {

echo 'Building'

}

}

stage('test'){

steps{

echo 'test'

}

}

}

}

基于Jenkinsfile 构建

pipeline {

agent any

stages {

stage('checkout') {

steps {

echo 'checkout'

checkout([$class: 'GitSCM', branches: [[name: '\*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: 'gogs\_luban', url: 'http://git.jiagouedu.com/java-vip/tuling-api-gateway']]])

}

}

stage('build'){

steps {

echo 'build'

sh 'mvn clean install'

}

}

stage('save') {

steps {

echo 'save'

archiveArtifacts 'target/\*.war'

}

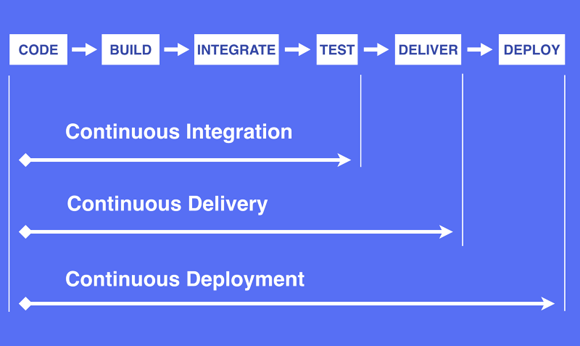
}

}

}

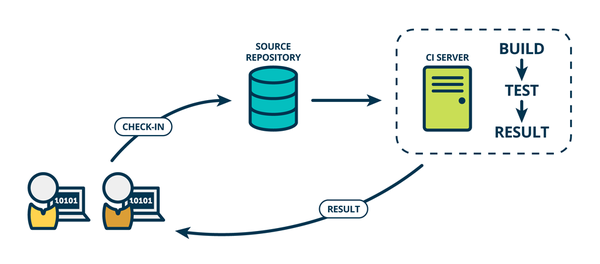
# 4、项目的可持续化集成

## 4.1 概念



**持续集成（continuous** **INTEGRATE）：**

持续集成是指软件个人研发的部分向软件整体部分交付，频繁进行集成以便更快地发现其中的错误。

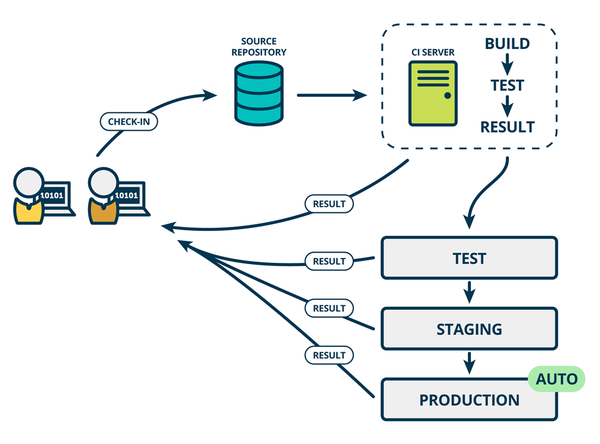


**持续交付（continuous** **DELIVER）**

持续交付在持续集成的基础上，将集成后的代码部署到更贴近真实运行环境的,预演境。

**持续部署 (continuous deployment )**

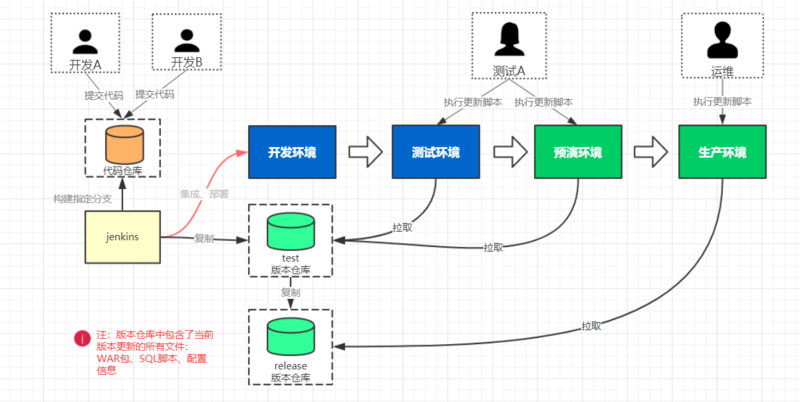
持续部署则是在持续交付的基础上，把部署到生产环境的过程自动化。



## 4.2 可持续化集成设计

### 1、整体流程设计

开发环境使用自动推送，在Jenkins打包完成后执行shell推送jar或war然后自动部署，测试环境到生产环境使用自动获取，通过shell脚本拉取jar或war，存储到svn类似的服务器上（使用svn的原因是nexus不能存储脚本跟配置文件），部署时在svn服务器上下载文件及配置



#### Svn的安装

SVN服务安装

#yum 安装

yum install subversion

#查看svnserver版本

svnversion --version

创建SVN版本目录

mkdir -p /data/svn/repository

svnadmin create /data/svn/repository

SVN配置

配置目录：  /data/svn/repository/conf

authz :目录权限设置

vim authz

#表示添加一个admin帐号，密码为 admin123

admin=admin123

passwd：用户与密码设置

vim passwd

#表示admin 拥有所有目录的读写权限

[/]

admin=rw

svnserve.conf ：svn服务

vim svnserve.conf

anon-access = read

auth-access = write

password-db = passwd

authz-db = authz

realm = /data/svn/repository

启动svn 服务

svnserve -d -r /var/svn/svnrepos

基于客户端访问svn

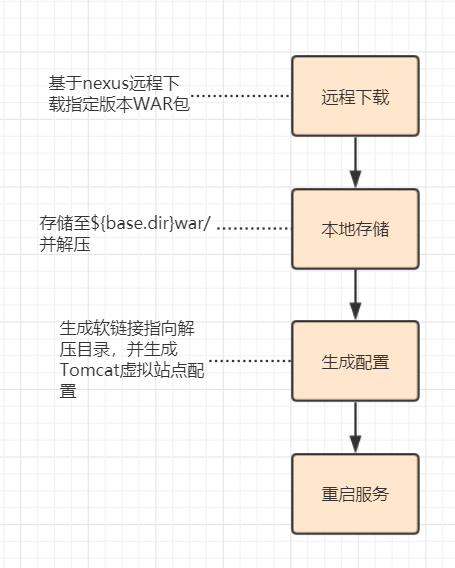
默认端口：3690

地址：svn://192.168.17.200:3690/

### 2、更新机制

更新机制是指项目如何进行实质的更新，图中我们充计了两种方式：一种是自动推送，另外一种是手动拉取。前者用于开发环境、后者可以用于所有环境。

手动拉取



上述流程由自己编写的shell脚本deploy.sh实现

#!/bin/bash -e

cd "`dirname $0`"

. ./pom.sh

#1. download war, ready env

echo "deploy time: $work\_time"

mkdir -p war/

war=war/$pom\_a-$pom\_v.war

download\_path="$nexus\_redirect?r=$pom\_r&g=$pom\_g&a=$pom\_a&v=$pom\_v&e=war"

wget  $download\_path -O $war

deploy\_war() {

        target\_d=war/${pom\_a}-${pom\_v}-$work\_time

        target\_dir=`pwd`/$target\_d

        if [ ! -f "$war" ]; then

                echo "war not exist: $war"

                exit 1

        fi

        unzip -q $war -d $target\_dir

        cp -r app-conf/\* $target\_dir/WEB-INF/classes/

        rm -rf appwar

        ln -sf $target\_d appwar

        ./tomcat.sh stop

        target\_ln=`pwd`/appwar

        echo '<?xml version="1.0" encoding="UTF-8" ?>

<Context docBase="'$target\_ln'" allowLinking="true">

</Context>' > conf/Catalina/localhost/ROOT.xml

        ./tomcat.sh start

}

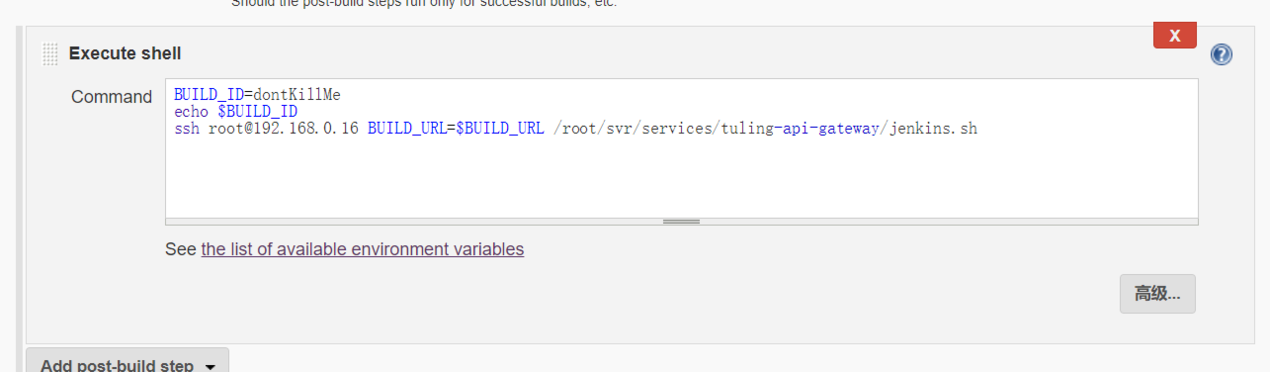
deploy\_war

注：在上述deploy.sh 脚本中还用到了 pom.sh 与tomcat.sh 其作用后续在做说明。

在测试环境、预演环境还有生产环境的时候 ，我们是不会直接从nexus 中下载的，而是特定的版本库。这时只需要修改 $download\_path 参数即可。

自动推送

自动推送方式采用在jenkins 构建完成之后，执行远程sh 脚本 用于下载本次构建WAR包，在自动部署重启。基jenkins 配置如下：



注：前提条件是 jenkins 服务要在业务服务器上做免密登录。（将Jenkins服务器公钥配置于业务服务器上）

此处只能用 Execute shell 而不用能用：Execute shell script on remote host using ssh ，因为BUILD\_ID=dontKillMe 不会被修改 ，而是当作参数来传递



${BUILD\_URL} 是一个隐示参数，表示本次构建的URL。

jenkins.sh 是放置在应用目录下的一个脚本：

#!/bin/bash -e

cd "`dirname $0`"

. ./pom.sh

#1. download war, ready env

echo "deploy time: $work\_time"

mkdir -p war/

# 配置下载存放目录

war=war/$pom\_a-$pom\_v.war

# 基于远程传过来的 BUILD\_URL下载本次构建

wget "${BUILD\_URL}${pom\_g}\$${pom\_a}/artifact/$pom\_g/$pom\_a/$pom\_v/$pom\_a-$pom\_v.war" -O $war

# 执行部署函数

deploy\_war

**配置文件更新：**

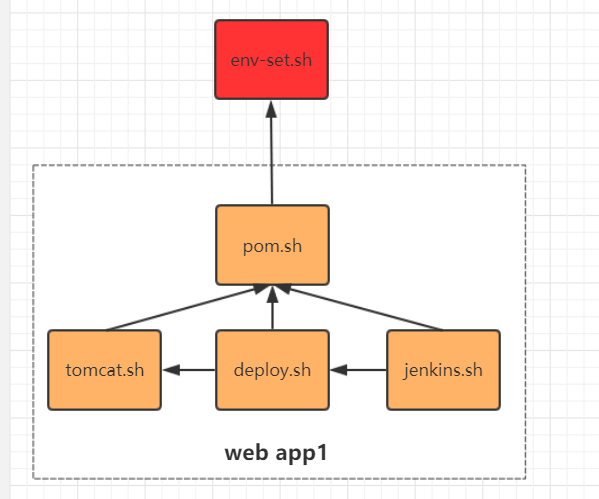
我们在跌代的过程当中总共 要经过四个环境，每个环境的配置信息是不一样的，如何在更新项目的时候把对应的配置文件也更新了呢? 有在有一种做法是 采用 Disconf 之类的配置系统来管理各个环境 的配置，但这里我们采用的是一个简单些的方案: 把当前环境 的配置文件放到 app-conf 目录下，等更新脚本的时候会一同覆盖原来的配置文件。

**版本回滚：**

之前在部署的时候是通过软链接的形式指向指定程序目录，而且原历史版本不会删除，回滚的时候只要把原软软链接 指定历史程序目录即可。而且配置文件也会一同回滚。

**更新脚本说明：**

前面说过deploy.sh 部署脚本中还用到了 pom.sh 与tomcat.sh 等脚本，这些脚本的作用是什么呢？整体逻辑如下依赖关系如下:

 说明：

* env-set.sh : 设置jvm 、Tomcat 等环境参数
* pom.sh: 设置当前项目的 groupid 、artifact、version 信息
* tomcat.sh：启动关闭Tomcat
* deploy.sh : 下载并部署项目
* jenkins.sh: 用于被jenkins 远程触发下载指定更新版本

注：以上脚本分别存储至 env-set.sh 与 Tomcat-deomo.zip 附件中,也可在相关课件目录中去获取

**Jenkins.sh**

#!/bin/bash -e

#进入当前shell脚本的存放目录

cd "`dirname $0`"

. ./pom.sh

#BUILD\_URL=http://10.200.51.105:9000/jenkins/job/bubugao-goods/269/

#pom\_v=0.5.0

#1. download war, ready env

echo "deploy time: $work\_time"

mkdir -p war/

war=war/$pom\_a-$pom\_v.war

wget "${BUILD\_URL}${pom\_g}\$${pom\_a}/artifact/$pom\_g/$pom\_a/$pom\_v/$pom\_a-$pom\_v.war" -O $war

#使用pom中定义的deploy\_war函数

deploy\_war

**deploy.sh**

#!/bin/bash -e

cd "`dirname $0`"

. ./pom.sh

#1. download war, ready env

echo "deploy time: $work\_time"

mkdir -p war/

war=war/$pom\_a-$pom\_v.war

download\_path="$nexus\_redirect?r=$pom\_r&g=$pom\_g&a=$pom\_a&v=$pom\_v&e=war"

wget $download\_path -O $war

#使用pom中定义的deploy\_war函数

deploy\_war

**pom.sh**

#!/bin/bash -e

. ../bin/env-set.sh

pom\_g=tuling.common

pom\_a=tuling-api-gateway

pom\_v=0.0.1-SNAPSHOT

pom\_r=snapshots

#自定义函数

deploy\_war() {

target\_d=war/${pom\_a}-${pom\_v}-$work\_time

#`pwd`返回当前所在地址

target\_dir=`pwd`/$target\_d

#-f 文件表达式 如果文件为常规文件则为真

if [ ! -f "$war" ]; then

echo "war not exist: $war"

#exit 0正常结束 1 异常结束

exit 1

fi

unzip -q $war -d $target\_dir

cp -r app-conf/\* $target\_dir/WEB-INF/classes/

rm -rf appwar

#创建软链接

ln -sf $target\_d appwar

./tomcat.sh stop

target\_ln=`pwd`/appwar

echo '<?xml version="1.0" encoding="UTF-8" ?>

<Context docBase="'$target\_ln'" allowLinking="true">

</Context>' > conf/Catalina/localhost/ROOT.xml

./tomcat.sh start

}

**Tomcat.sh**

#!/bin/bash

if [ "`whoami`" != "root" ];then

echo "Error: You must be apps to run this command."

exit 1

fi

cd "`dirname $0`"

. ../bin/env-set.sh

. ./pom.sh

export CATALINA\_BASE="`pwd`"

tomcat "$1" "${pom\_a}-${pom\_v}"

**env-set.sh**

#!/bin/bash -e

#当前时间 export将此变量变为此脚本与其子脚本的环境变量，当父脚本执行完毕后，该变量消失

export now\_time=$(date +%Y-%m-%d\_%H-%M-%S)

#整个期间的开始时间

[ -z "$work\_time" ] && export work\_time=$(date +%Y-%m-%d\_%H-%M-%S)

#-z用来检查变量是否定义，原理为未定义的变量值为空，即字符串为空则返回true

if [ -z "$g\_env\_set" ]; then

unset JRE\_HOME JAVA\_HOME CLASSPATH

export nexus\_redirect="http://192.168.0.15:7777/nexus/service/local/artifact/maven/redirect"

export data\_home="/root/svr"

export server\_home="$data\_home/services"

export JAVA\_HOME="$data\_home/jdk"

export CATALINA\_HOME="$data\_home/apache-tomcat"

export CATALINA\_BASE="$CATALINA\_HOME"

export PATH=$JAVA\_HOME/bin:$PATH

export g\_env\_set=true

fi

#能够输入2个grep的字符串用来查找

proc\_find() {

if [ "$#" = 2 ]; then

ps -eo pid,cmd|grep -v 'cmd\|grep'|grep "$1"|grep "$2"|sed 's/^ \*\(.\*\) \*$/\1/'

else

ps -eo pid,cmd|grep -v 'cmd\|grep'|grep "$1"|sed 's/^ \*\(.\*\) \*$/\1/'

fi

}

#能够输入2个grep的字符串用来查找

proc\_pid() {

proc\_find "$@" | cut -d ' ' -f1

}

#$1: 要重试的次数，超过此次数后使用kill -9

#$2: 能够输入2个grep的字符串用来查找

proc\_kill() {

grep\_args="${@:2}"

for ((i=1;i<=$1;++i))

do

pids=(`proc\_pid "$grep\_args"`)

[ ${#pids[@]} = 0 ] && return 0

echo "n=$i kill ${pids[@]}"

kill ${pids[@]}

sleep 1s

done

pids=(`proc\_pid "$grep\_args"`)

[ ${#pids[@]} = 0 ] || kill -9 ${pids[@]}

}

tomcat\_start0() {

[ "$JAVA\_OPTS" = "" ] && export JAVA\_OPTS="-server -XX:MaxPermSize=128m -Xms512m -Xmx512m -javaagent:/root/svr/services/cbt-agent/cbt-agent.lib"

[ "`echo "$JAVA\_OPTS"|grep '\-Djava.security.egd='`" = "" ] && export JAVA\_OPTS="$JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom"

#防止X11导致awt在X11关闭后异常

export JAVA\_OPTS="-Djava.awt.headless=true $JAVA\_OPTS -$grep\_server"

echo "START $grep\_server: JAVA\_OPTS=$JAVA\_OPTS"

"$CATALINA\_HOME"/bin/startup.sh

}

tomcat\_stop0() {

proc\_kill 3 "$grep\_server"

}

tomcat\_status0() {

tmp=`proc\_find "$grep\_server"`

if [ -z "$tmp" ]; then

echo "$grep\_server stopped."

else

echo "$grep\_server running: $tmp"

fi

}

tomcat() {

if [ "$1" = "" ]; then

printf 'Usage: %s {start|stop|restart|status} ????\n' "$prog"

exit 1

fi

if [ "$2" = "" ]; then

echo 'java -Dmykey=${mykey} miss: mykey'

exit 1

fi

if [ "$CATALINA\_BASE" = "" ]; then

echo 'miss: $CATALINA\_BASE='

exit 1

fi

cd "$CATALINA\_BASE"

export grep\_server="Dmykey=${2}x"

case "$1" in

start)

tmp=`proc\_find "$grep\_server"`

if [ -z "$tmp" ]; then

tomcat\_start0

else

echo "$grep\_server already running: $tmp"

fi

sleep 1s

tomcat\_status0

;;

stop)

tomcat\_stop0

tomcat\_status0

;;

status)

tomcat\_status0

;;

restart)

tomcat\_stop0

tomcat\_start0

sleep 1s

tomcat\_status0

;;

\*)

printf 'Usage: %s {start|stop|restart|status} ????\n' "$prog"

exit 1

;;

esac

}