|  |
| --- |
| KCloudy |
| **SpringBoot In Linux** |
| [文档副标题] |

|  |
| --- |
| 田 长军  [日期] |

目录

**[1.](#_Toc8141440)****[部署前准备](#_Toc8141440)** [1](#_Toc8141440)

**[1.1](#_Toc8141441)****[安装Java](#_Toc8141441)** [1](#_Toc8141441)

**[1.2](#_Toc8141442)****[安装Maven](#_Toc8141442)** [1](#_Toc8141442)

**[1.2.1](#_Toc8141443)****[下载Maven安装包](#_Toc8141443)** [1](#_Toc8141443)

**[1.2.2](#_Toc8141444)****[解压Maven安装包](#_Toc8141444)** [2](#_Toc8141444)

**[1.2.3](#_Toc8141445)****[配置Maven路径](#_Toc8141445)** [2](#_Toc8141445)

**[1.2.4](#_Toc8141446)****[验证Maven是否配置成功](#_Toc8141446)** [2](#_Toc8141446)

**[1.2.5](#_Toc8141447)****[配置国内阿里云Maven镜像](#_Toc8141447)** [2](#_Toc8141447)

**[1.3](#_Toc8141448)****[安装Git](#_Toc8141448)** [3](#_Toc8141448)

**[1.4](#_Toc8141449)****[安装Docker](#_Toc8141449)** [3](#_Toc8141449)

**[1.4.1](#_Toc8141450)****[使用yum源安装](#_Toc8141450)** [3](#_Toc8141450)

**[1.4.2](#_Toc8141451)****[安装Docker Compose](#_Toc8141451)** [3](#_Toc8141451)

**[1.4.3](#_Toc8141452)****[安装Docker Machine](#_Toc8141452)** [3](#_Toc8141452)

**[1.4.4](#_Toc8141453)****[使用国内Docker镜像源](#_Toc8141453)** [4](#_Toc8141453)

**[1.4.5](#_Toc8141454)****[启动及测试](#_Toc8141454)** [4](#_Toc8141454)

**[1.5](#_Toc8141455)****[安装Jenkins](#_Toc8141455)** [5](#_Toc8141455)

**[1.5.1](#_Toc8141456)****[安装前准备](#_Toc8141456)** [5](#_Toc8141456)

**[1.5.2](#_Toc8141457)****[安装Jenkins](#_Toc8141457)** [6](#_Toc8141457)

**[1.5.3](#_Toc8141458)****[Jenkins相关配置信息](#_Toc8141458)** [6](#_Toc8141458)

**[1.5.4](#_Toc8141459)****[启动Jenkins及安装插件](#_Toc8141459)** [7](#_Toc8141459)

**[1.5.5](#_Toc8141460)****[配置Nginx的反向代理](#_Toc8141460)** [9](#_Toc8141460)

**[1.5.6](#_Toc8141461)****[配置Java JDK、Maven、Git](#_Toc8141461)** [11](#_Toc8141461)

**[2.](#_Toc8141462)****[SpringBoot](#_Toc8141462)** [11](#_Toc8141462)

**[2.1](#_Toc8141463)****[SpringBoot启动流程及自动配置](#_Toc8141463)** [11](#_Toc8141463)

**[2.1.1](#_Toc8141464)****[SpringBoot启动总体流程](#_Toc8141464)** [11](#_Toc8141464)

**[2.1.2](#_Toc8141465)****[SpringBoot应用程序入口](#_Toc8141465)** [12](#_Toc8141465)

**[2.1.3](#_Toc8141466)****[创建SpringApplication对象](#_Toc8141466)** [13](#_Toc8141466)

**[2.1.4](#_Toc8141467)****[启动SpringBoot应用](#_Toc8141467)** [13](#_Toc8141467)

**[2.1.5](#_Toc8141468)****[SpringBoot的自动配置](#_Toc8141468)** [17](#_Toc8141468)

**[2.2](#_Toc8141469)****[Spring MVC流程](#_Toc8141469)** [21](#_Toc8141469)

**[2.2.1](#_Toc8141470)****[流程图](#_Toc8141470)** [21](#_Toc8141470)

**[2.2.2](#_Toc8141471)****[其他](#_Toc8141471)** [21](#_Toc8141471)

**[2.3](#_Toc8141472)****[Web容器（默认使用Tomcat）](#_Toc8141472)** [21](#_Toc8141472)

**[2.3.1](#_Toc8141473)****[Jetty容器](#_Toc8141473)** [21](#_Toc8141473)

**[2.3.2](#_Toc8141474)****[undertow容器](#_Toc8141474)** [22](#_Toc8141474)

**[2.3.3](#_Toc8141475)****[其他](#_Toc8141475)** [23](#_Toc8141475)

**[2.4](#_Toc8141476)****[其他](#_Toc8141476)** [23](#_Toc8141476)

**[3.](#_Toc8141477)****[SpringBoot相关插件](#_Toc8141477)** [23](#_Toc8141477)

**[3.1](#_Toc8141478)****[Console支持彩色输出插件：ANSI Escape in Console](#_Toc8141478)** [23](#_Toc8141478)

[3.1.1 进入Eclipse Marketplace，搜索ANSI Escape in Console并插件 23](#_Toc8141479)

[3.1.2 设置application.properties 23](#_Toc8141480)

[3.1.3 配置日志（logback.xml） 23](#_Toc8141481)

[3.1.4 运行结果 26](#_Toc8141482)

**[3.2](#_Toc8141483)****[其他](#_Toc8141483)** [27](#_Toc8141483)

**[4.](#_Toc8141484)****[使用Docker部署SpringBoot Web项目](#_Toc8141484)** [27](#_Toc8141484)

**[4.1](#_Toc8141485)****[创建SpringBoot的Web项目](#_Toc8141485)** [27](#_Toc8141485)

**[4.2](#_Toc8141486)****[Web项目添加Docker支持](#_Toc8141486)** [28](#_Toc8141486)

**[4.2.1](#_Toc8141487)****[添加Docker插件](#_Toc8141487)** [28](#_Toc8141487)

**[4.2.2](#_Toc8141488)****[添加Dockerfile文件](#_Toc8141488)** [30](#_Toc8141488)

**[4.3](#_Toc8141489)****[将项目部署至Docker](#_Toc8141489)** [30](#_Toc8141489)

**[4.3.1](#_Toc8141490)****[检测环境](#_Toc8141490)** [30](#_Toc8141490)

**[4.3.2](#_Toc8141491)****[测试项目](#_Toc8141491)** [30](#_Toc8141491)

**[4.3.3](#_Toc8141492)****[生成Docker镜像](#_Toc8141492)** [31](#_Toc8141492)

**[4.3.4](#_Toc8141493)****[运行Docker镜像](#_Toc8141493)** [31](#_Toc8141493)

**[4.3.5](#_Toc8141494)****[测试Docker项目](#_Toc8141494)** [32](#_Toc8141494)

**[4.4](#_Toc8141495)****[相关问题](#_Toc8141495)** [32](#_Toc8141495)

**[4.4.1](#_Toc8141496)****[mvn docker:build fails with "{}->unix://localhost:80: Connection reset by peer](#_Toc8141496)** [32](#_Toc8141496)

**[4.4.2](#_Toc8141497)****[other](#_Toc8141497)** [32](#_Toc8141497)

**[4.5](#_Toc8141498)****[相关资料](#_Toc8141498)** [32](#_Toc8141498)

**[4.6](#_Toc8141499)****[其他](#_Toc8141499)** [32](#_Toc8141499)

**[5.](#_Toc8141500)****[其他](#_Toc8141500)** [32](#_Toc8141500)

**SpringBoot In Linux**

1. **部署前准备**
   1. **安装Java**

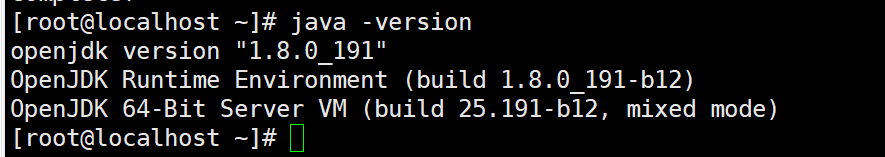
sudo yum -y install java-1.8.0-openjdk #jre

sudo yum -y install java-1.8.0-openjdk-devel #jdk

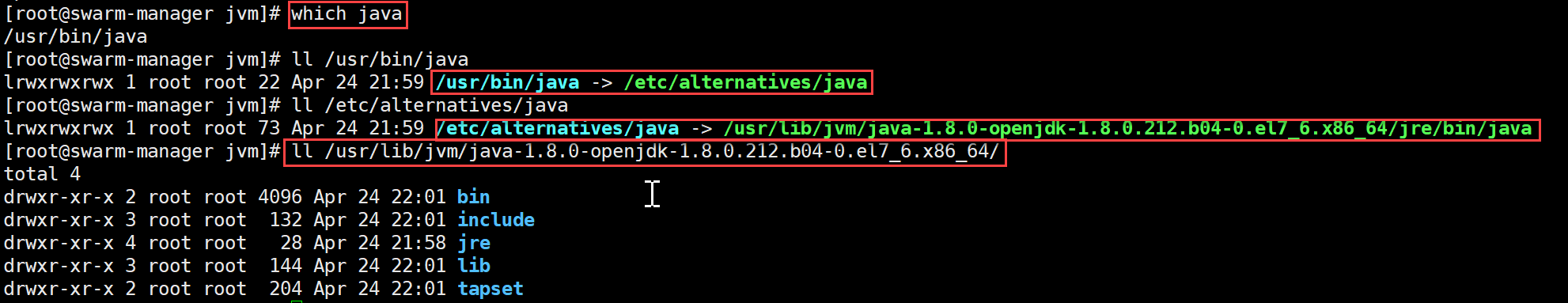
但对于centos有个问题，不可以安装成gcj(例如：gij(GNU libgcj) version 4.4.6 20110731(Red Hat 4.4.6-3)),导致[Jekins不工作](https://issues.jenkins-ci.org/browse/JENKINS-743" \t "_blank),如果查看java版本类似上例，需要卸载，安装其他版本

java -version #查看Java版本

yum remove java #卸载异常版本



which java #查看java执行路径，备注：java实际的路径为-/usr/lib/jvm



设置java路径：vi /etc/profile 在文件最后添加下列配置

export JAVA\_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.212.b04-0.el7\_6.x86\_64

export JRE\_HOME=$JAVA\_HOME/jre

export CLASSPATH=$JAVA\_HOME/lib:$JRE\_HOME/lib:$CLASSPATH

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

使配置生效： source /etc/profile

* 1. **安装Maven**
     1. **下载Maven安装包**

wget <https://mirrors.tuna.tsinghua.edu.cn/apache/maven/maven-3/3.6.2/binaries/apache-maven-3.6.2-bin.tar.gz>

* + 1. **解压Maven安装包**

tar -zxvf apache-maven-3.6.2-bin.tar.gz

mv apache-maven-3.6.2 /usr/local/maven3

* + 1. **配置Maven路径**

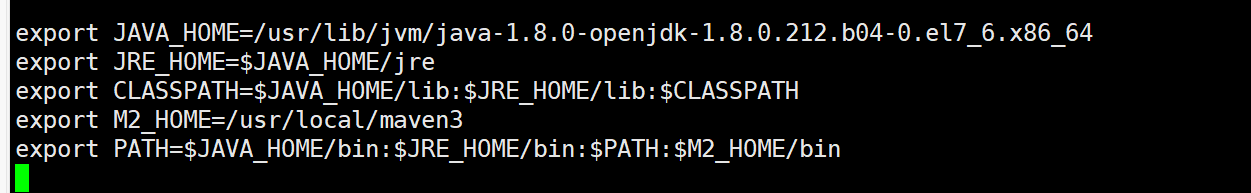
vi /etc/profile #最后添加以下内容

export M2\_HOME=/usr/local/maven3

export PATH=$PATH:$JAVA\_HOME/bin:$M2\_HOME/bin #保存退出后运行下面的命令使配置生效，或者重启服务器生效

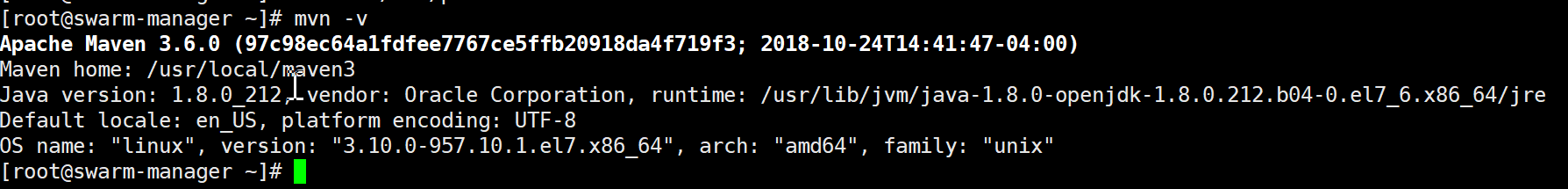
source /etc/profile

配置结果如下图：

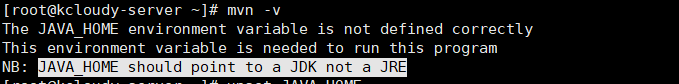


* + 1. **验证Maven是否配置成功**

mvn -v



* + 1. **使用mvn -v时，出现错误：JAVA\_HOME should point to a JDK not a JRE**



unset JAVA\_HOME #重置下JAVA\_HOME

* + 1. **配置国内阿里云Maven镜像**

vi /usr/local/maven3/conf/settings.xml

配置mirrors的子节点，添加如下mirror

<mirror>

<id>nexus-aliyun</id>

<mirrorOf>central</mirrorOf>

<name>Nexus aliyun</name>

<url>http://maven.aliyun.com/nexus/content/groups/public</url>

</mirror>



* 1. **安装Git**

sudo yum -y install git

git --version

* 1. **安装Docker**
     1. **使用yum源安装**

sudo wget -P /etc/yum.repos.d/ <http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo>

sudo yum install -y docker-ce-18.09.3-3.el7 docker-ce-cli-18.09.3 containerd.io-18.09.3

docker --version

* + 1. **安装Docker Compose**

curl -L https://github.com/docker/compose/releases/download/1.23.2/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

chmod +x /usr/local/bin/docker-compose

docker-compose --version

* + 1. **安装Docker Machine**

sudo wget -P /etc/yum.repos.d/ http://download.virtualbox.org/virtualbox/rpm/rhel/virtualbox.repo

sudo yum install -y VirtualBox-5.1 #安装VirtualBox

curl -L https://github.com/docker/machine/releases/download/v0.16.1/docker-machine-`uname -s`-`uname -m` >/tmp/docker-machine &&

chmod +x /tmp/docker-machine &&

sudo cp /tmp/docker-machine /usr/local/bin/docker-machine

docker-machine -version

* + 1. **使用国内Docker镜像源**

vi /etc/docker/daemon.json #修改为以下内容

{

"exec-opts": ["native.cgroupdriver=systemd"],

"registry-mirrors" : [

"http://ovfftd6p.mirror.aliyuncs.com",

"http://registry.docker-cn.com",

"http://docker.mirrors.ustc.edu.cn",

"http://hub-mirror.c.163.com"

],

"insecure-registries" : [

"registry.docker-cn.com",

"docker.mirrors.ustc.edu.cn"

],

"debug" : true,

"experimental" : true

}

* + 1. **启动及测试**
       1. **启动Docker**

service docker start

chkconfig docker on

#LCTT 译注：此处采用了旧式的 sysv 语法，如采用CentOS 7中支持的新式 systemd 语法，如下：

systemctl start docker.service #开机自启 Docker CE

systemctl enable docker.service

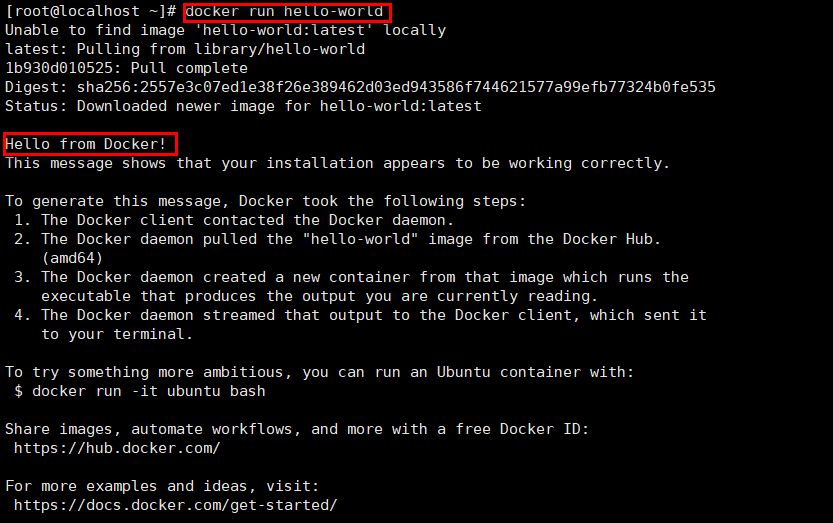
systemctl status docker #查看Docker状态

docker -v #查看docker版本



* + - 1. **测试Docker是否安装正确**

docker run hello-world



* 1. **安装Jenkins**
     1. **安装前准备**
        1. **防火墙设置**

#配置Jenkins端口8080

firewall-cmd --zone=public --add-port=8080/tcp --permanent

#开端口后必须重新加载

firewall-cmd --reload

* + - 1. **安装JQ**

yum install epel-release -y

yum install jq -y

jq --version

#或者

wget -o jq <https://github.com/stedolan/jq/releases/download/jq-1.6/jq-linux64>

chmod +x ./jq

cp jq /usr/bin

#验证

echo '{"first\_name":"John","full\_name":"John Doe","last\_name":"Doe"}' | jq .

echo '{"first\_name":"John","full\_name":"John Doe","last\_name":"Doe"}' | jq . full\_name

* + 1. **安装Jenkins**
       1. **拉取库的配置到本地**

sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/jenkins.repo

* + - 1. **导入公钥**

sudo rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key

* + - 1. **安装Jenkins**

＃`-y`参数：回答全部问题为是

sudo yum -y install jenkins

* + - 1. **更新Jenkins**

yum update jenkins

* + - 1. **卸载Jenkins**

rpm -e jenkins

删除其他相关文件

find / -iname jenkins | xargs -n 1000 rm -rf

* + 1. **Jenkins相关配置信息**
       - 1. 系统配置文件

cat /etc/sysconfig/jenkins | more

可以获得几个重要配置项目信息

JENKINS\_HOME="/var/lib/jenkins",存放jenkins 配置及工作文件

JENKINS\_PORT="8080",jenkins默认8080端口

* + - * 1. 配置文件夹

ls /var/lib/jenkins

有jobs、logs、plugins等文件夹及文件若干。  
这次主要看了看plugins文件夹，所有插件都在里面，如插件ssh-slaves,会有一个ssh-slaves文件夹及ssh-slaves.jpi。  
当某个插件未安装成功时，会有一个以.tmp结尾的文件

* + - * 1. 日志

/var/lib/jenkins/logs

/var/log/jenkins/jenkins.log,记录了插件安装等日志，失败信息原因等很清晰

* + - * 1. 其他
    1. **启动Jenkins及安装插件**
       1. **启动服务**

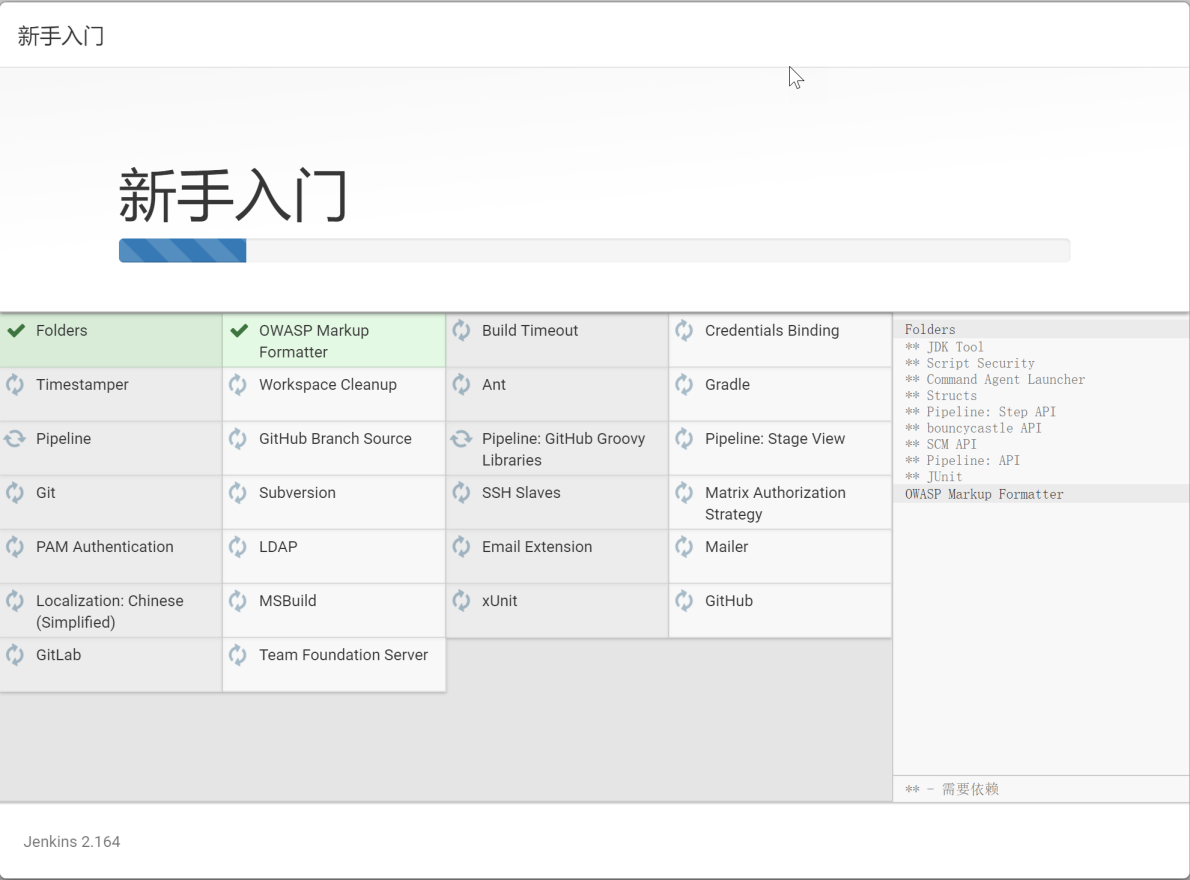
service jenkins start

* + - 1. **浏览器访问：http://192.168.56.99:8080**
      2. **获取管理员密码**

vi /var/lib/jenkins/secrets/initialAdminPassword

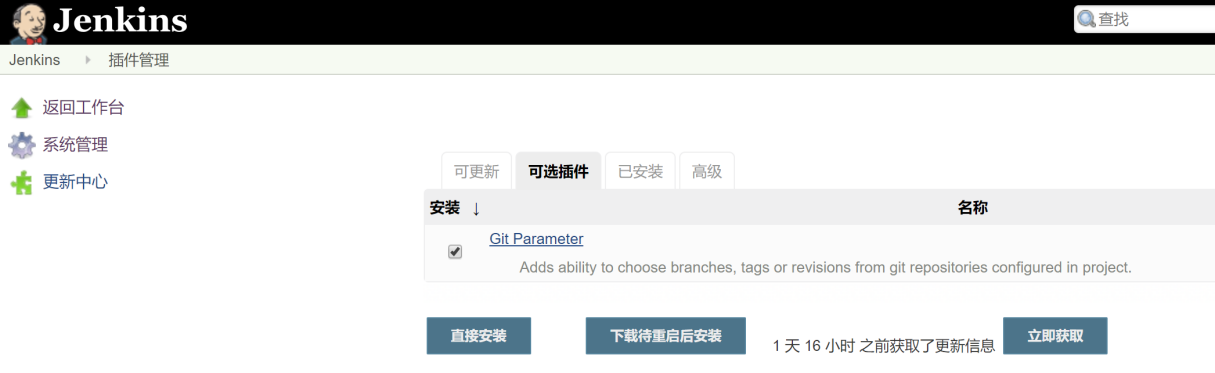


* + - 1. **安装插件**

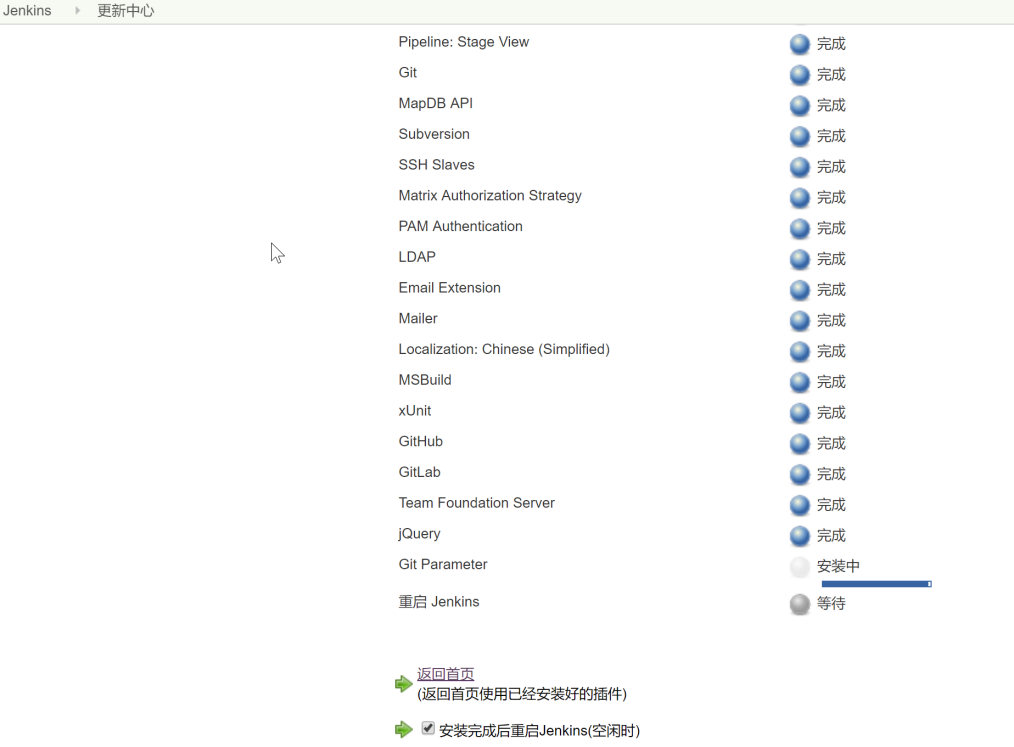


* + - 1. **安装Git Parameter插件**

1. 系统管理-->插件管理-->可选插件-->搜索git parameter；



1. 安装插件



* + 1. **配置Nginx的反向代理**
       1. **创建Nginx下Jenkins的配置文件**

vi /usr/local/nginx/conf/conf.d/jenkins.conf

* + - 1. **配置文件内容如下**

upstream jenkins {

keepalive 32; # keepalive connections

server 127.0.0.1:8080; # jenkins ip and port

}

server {

#注意这里，要把默认的那个default\_server去掉,因为我们在下面要单独配置域名访问，所以这里不要留default\_server，不然会报错。

listen 80;

#这里写你想设置的域名，可以写多个，与名之间用空格隔开

server\_name jenkins.domain.com;

# Load configuration files for the default server block.

location / {

proxy\_set\_header Host $host:$server\_port;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

# Fix the "It appears that your reverse proxy set up is broken" error.

proxy\_pass http://127.0.0.1:8080;

proxy\_read\_timeout 90;

proxy\_redirect http://127.0.0.1:8080 https://jenkins.domain.com;

# Required for new HTTP-based CLI

proxy\_http\_version 1.1;

proxy\_request\_buffering off;

# workaround for https://issues.jenkins-ci.org/browse/JENKINS-45651

add\_header 'X-SSH-Endpoint' 'jenkins. domain.com:50022' always;

}

access\_log /var/log/jenkins/access.log;

error\_log /var/log/jenkins/error.log;

error\_page 404 /404.html;

location = /404.html {

root /usr/local/nginx/html;

}

# redirect server error pages to the static page /50x.html

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root /usr/local/nginx/html;

}

}

* + - 1. **重启Nginx**

nginx -t

nginx -s reload

* + - 1. **配置Jenkins访问路径：**

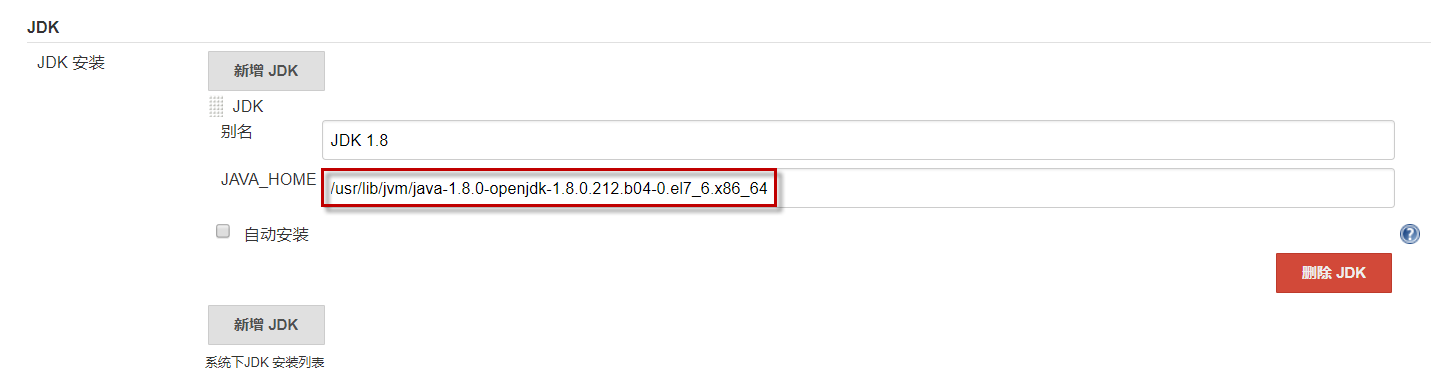
访问Jenkins：<http://jenkins.domain.com>

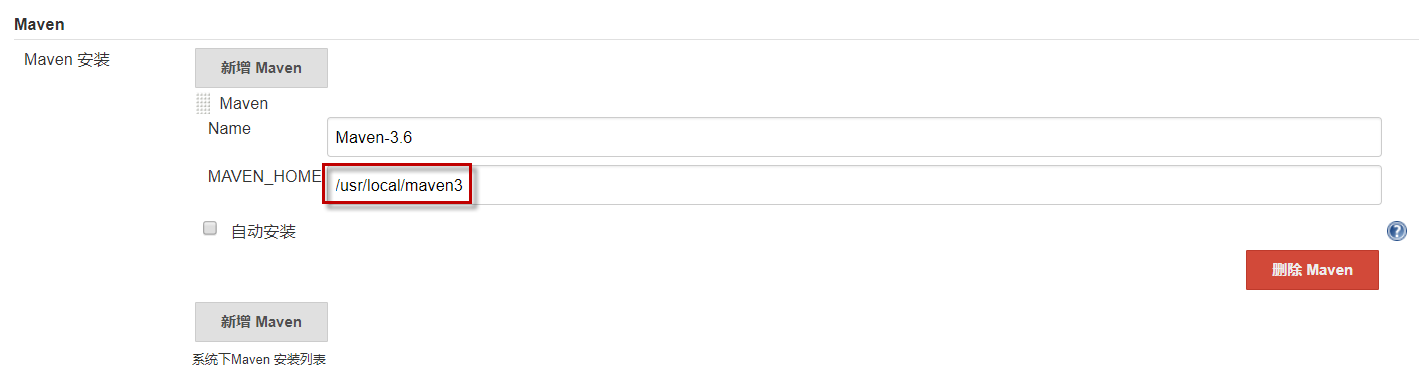
进入系统管理-->系统设置-->**Jenkins Location，设置URL为：**<http://jenkins.domain.com>



* + 1. **配置Java JDK、Maven、Git**

在“系统管理”—“系统全局工具配置”下设置JDK、Maven及Git



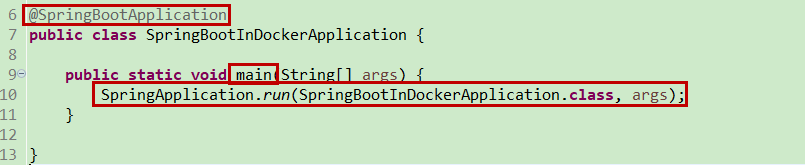


1. **SpringBoot**
   1. **SpringBoot启动流程及自动配置**
      1. **SpringBoot启动总体流程**



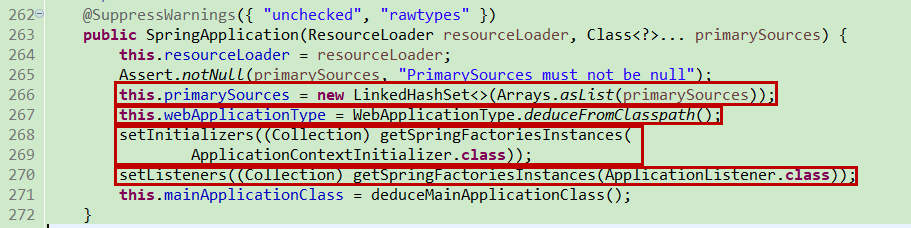
SpringBoot启动总体分为三个步骤：

* 第一部分进行SpringApplication的初始化模块，配置一些基本的环境变量、资源、构造器、监听器，
* 第二部分实现了应用具体的启动方案，包括启动流程的监听模块、加载配置环境模块及核心的创建上下文环境模块
* 第三部分是自动化配置模块，该模块作为springboot自动配置核心
  + 1. **SpringBoot应用程序入口**



每个SpringBoot程序都有一个主入口，也就是main方法，main里面调用SpringApplication.run()启动整个spring-boot程序，该方法所在类需要使用@SpringBootApplication注解，以及@ImportResource注解(if need)，@SpringBootApplication包括三个注解，功能如下：

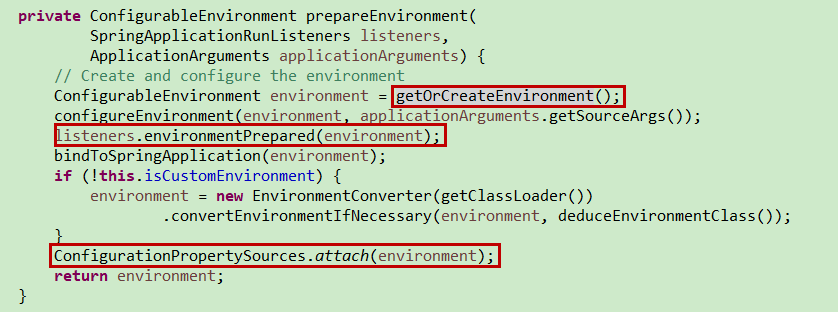
* @EnableAutoConfiguration：SpringBoot根据应用所声明的依赖来对Spring框架进行自动配置
* @SpringBootConfiguration(内部为@Configuration)：被标注的类等于在spring的XML配置文件中(applicationContext.xml)，装配所有bean事务，提供了一个spring的上下文环境
* @ComponentScan：组件扫描，可自动发现和装配Bean，默认扫描SpringApplication的run方法里的Booter.class所在的包路径下文件，所以最好将该启动类放到根包路径下
  + 1. **创建SpringApplication对象**

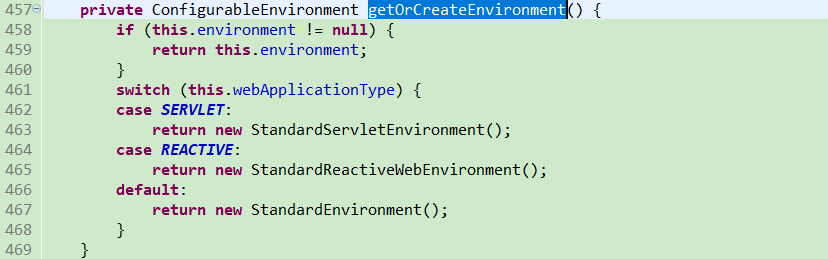


* + 1. **启动SpringBoot应用**

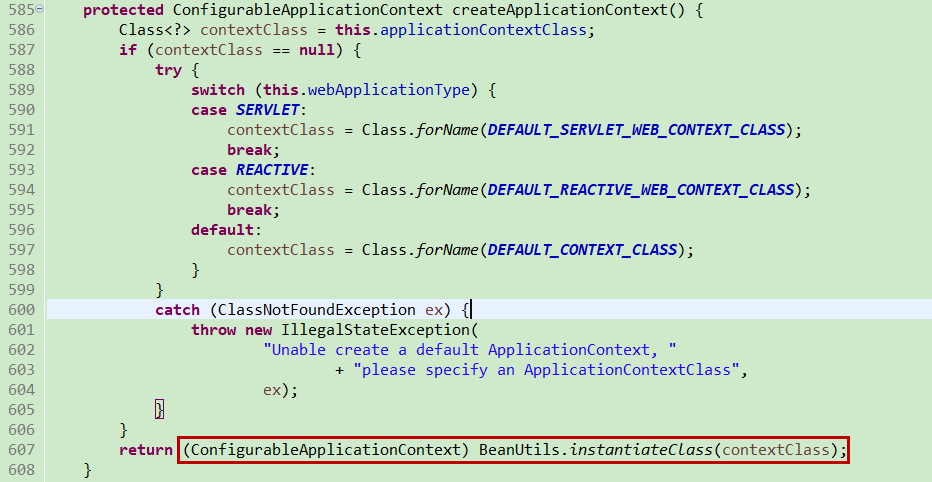


* 创建了应用的监听器SpringApplicationRunListeners并开始监听
* 创建SpringBoot配置环境ConfigurableEnvironment（如果是通过web容器发布，会加载StandardEnvironment，其最终也是继承了ConfigurableEnvironment)；配置环境(Environment)加入到监听器对象中(SpringApplicationRunListeners)；加载属性文件资源；





* 创建run方法的返回对象：ConfigurableApplicationContext(应用配置上下文)，我们可以看一下创建方法：

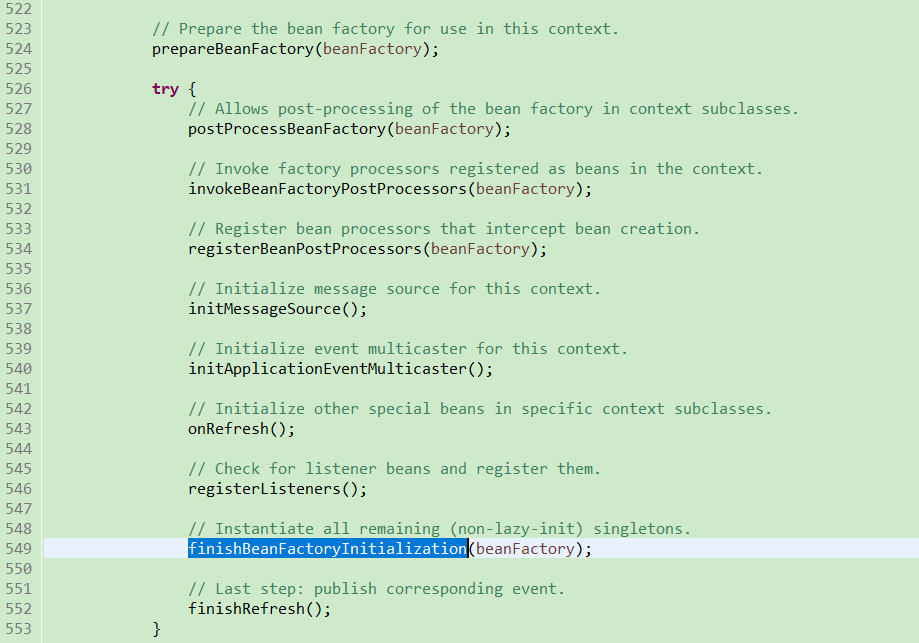


方法会先获取显式设置的应用上下文(applicationContextClass)，如果不存在，再加载默认的环境配置（通过是否是web environment判断），默认选择AnnotationConfigApplicationContext注解上下文（通过扫描所有注解类来加载bean），最后通过BeanUtils实例化上下文对象，并返回

* 回到run方法内，prepareContext方法将listeners、environment、applicationArguments、banner等重要组件与上下文对象关联



* 接下来的refreshContext(context)方法(调用基类： AbstractApplicationContext的初始化方法如下)将是实现spring-boot-starter-\*(mybatis、redis等)自动化配置的关键，包括spring.factories的加载，bean的实例化等核心工作



1. prepareBeanFactory(beanFactory)：设置beanFactory的classloader，BeanExpressionResolver，PropertyEditorRegistrar，ApplicationContextAwareProcessor和忽略xxxxAware，注册依赖，还有ApplicationListenerDetector

ApplicationContextAwareProcessor：只是将applicationContext传递给ApplicationContextAwareProcessor，方便后面的xxxAware调用

忽略xxxxAware：忽略这些Aware接口实现类中与接口set方法中入参类型相同的属性的的自动注入这样就保证了关键的类是由spring容器自己产生的而不是我们注入的，

自动注入不是指的@AutoWire 而是指的是beans的default-autowire="byType" 或在bean的autowire="byType" ，这样spring 回去ioc容器寻找类型相似的类型给其注入，如果实现了spring 的xxaware接口，就不会自动注入记载filterPropertyDescriptorsForDependencyCheck删除与入参类型相同的属性

注册依赖：即指定一些类自动注入的实例是spring指定的实例对象  
ApplicationListenerDetector：检测实现了ApplicationListener的实现类，因为有些实现类，无法通过getBeanNamesForType获取到

1. postProcessBeanFactory(beanFactory)：继续设置ignoreDependencyInterface（ServletContextAware）还有annotatedClasses，basePackages如果存在就设置
2. invokeBeanFactoryPostProcessors(beanFactory)：从beanFactoryPostProcessors获取BeanFactoryPostProcessor，然后先执行BeanDefinitionRegistryPostProcessor类型的postProcessBeanDefinitionRegistry，继续从beanFactory获取BeanDefinitionRegistryPostProcessor类型的bean然后执行postProcessBeanDefinitionRegistry，执行的过程按照PriorityOrdered，Ordered，普通的类型进行执行，然后优先执行registryProcessors的postProcessBeanFactory在执行regularPostProcessors的postProcessBeanFactory，再从BeanFactory获取PriorityOrdered，Ordered，普通的类型三种类型的BeanFactoryPostProcessor，并按照顺序执行。

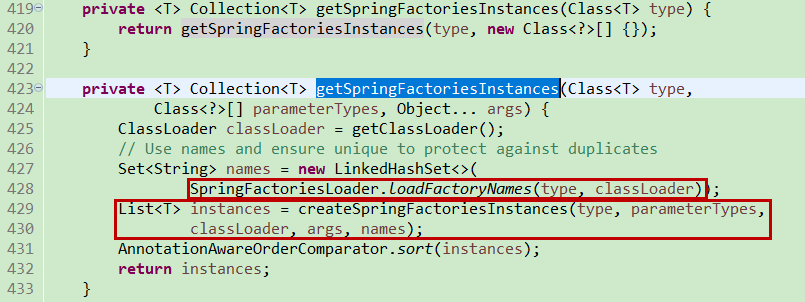
总结：从之前加入的beanFactoryPostProcessor先执行postProcessBeanDefinitionRegistry（假如是BeanDefinitionRegistryPostProcessor）然后在执行postProcessBeanFactory方法，然后从beanFactory获取BeanFactoryPostProcessor 然后执行postProcessBeanFactory，执行过程中都要按照PriorityOrdered，Ordered，普通的类型三种类型的顺序执行

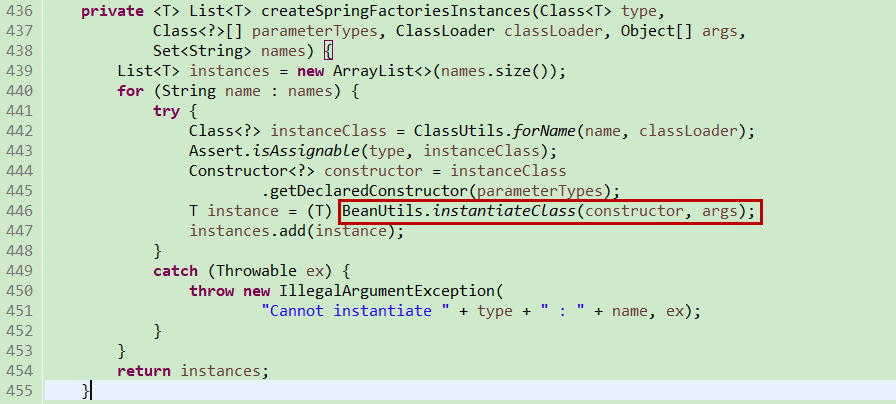
1. registerBeanPostProcessors：从beanFactory获取BeanPostProcessor分别按照PriorityOrdered，Ordered，普通的类型注册BeanPostProcessor
2. BeanPostProcessor和BeanFactoryPostProcessor:前者是对bean初始化前后进行设置，后者可以对beanFactory进行修改 或者，可以对beanDefinition进行修改或者增加或者初始化渴望提前初始化的bean
3. initMessageSource()：一般是我们用来初始化我们国际化文件的
4. initApplicationEventMulticaster():设置applicationEventMulticaster，spring发布各种事件就依靠他，这个和springboot发布事件使用相同的类
5. onRefresh()：初始化其他的子容器类中的bean，同时创建spring的内置tomcat，这在后期Springboot内嵌式tomcat中详细阐述
6. registerListeners()：添加用户设置applicationListeners，然后从beanFactory获取ApplicationListener，然后发布需要earlyApplicationEvents事件
7. finishBeanFactoryInitialization(beanFactory)：实例化非懒加载的剩余bean
8. finishRefresh：清理资源缓存，初始化lifecycle，调用lifecycle的onrefresh，发布ContextRefreshedEvent的事件,激活JMX,启动tomcat

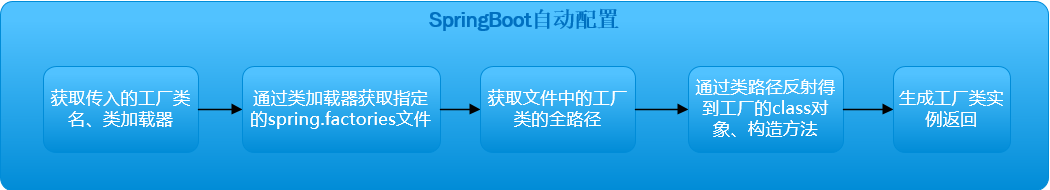
配置结束后，Springboot做了一些基本的收尾工作，返回了应用环境上下文。回顾整体流程，Springboot的启动，主要创建了配置环境(environment)、事件监听(listeners)、应用上下文(applicationContext)，并基于以上条件，在容器中开始实例化我们需要的Bean，至此，通过SpringBoot启动的程序已经构造完成。

* + 1. **SpringBoot的自动配置**
       1. SpringBoot启动时使用到的自动配置

之前的启动结构图中，我们注意到无论是应用初始化还是具体的执行过程，都调用了SpringBoot自动配置模块，调用方法入口如下图：

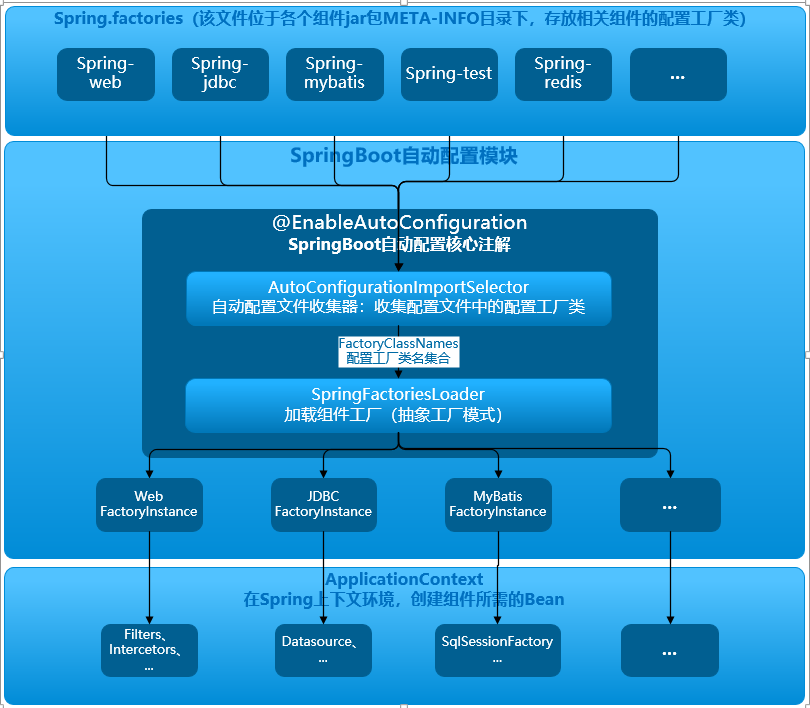






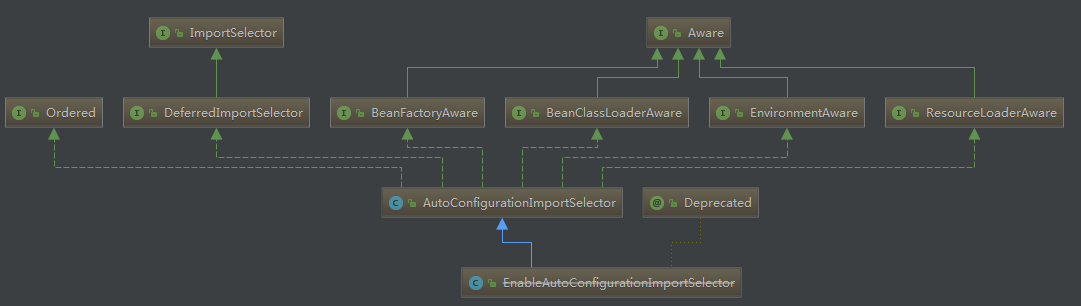
该配置模块的主要使用到了SpringFactoriesLoader，即Spring工厂加载器，该对象提供了loadFactoryNames方法，入参为factoryClass和classLoader，即需要传入上图中的工厂类名称和对应的类加载器，方法会根据指定的classLoader，加载该类加器搜索路径下的指定文件，即spring.factories文件，传入的工厂类为接口，而文件中对应的类则是接口的实现类，或最终作为实现类，所以文件中一般为如下图这种一对多的类名集合，获取到这些实现类的类名后，loadFactoryNames方法返回类名集合，方法调用方得到这些集合后，再通过反射获取这些类的类对象、构造方法，最终生成实例。

* + - 1. SpringBoot自动配置关键组件关系图

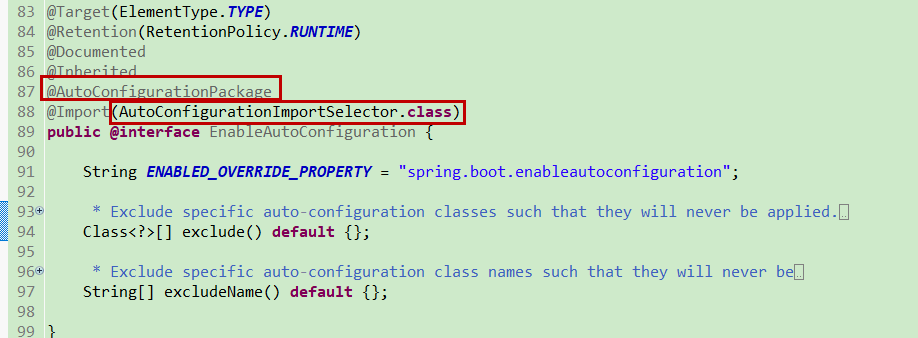


mybatis-spring-boot-starter、spring-boot-starter-web等组件的META-INF文件下均含有spring.factories文件，自动配置模块中，SpringFactoriesLoader收集到文件中的类全名并返回一个类全名的数组，返回的类全名通过反射被实例化，就形成了具体的工厂实例，工厂实例来生成组件具体需要的bean

* + - 1. @EnableAutoConfiguration注解

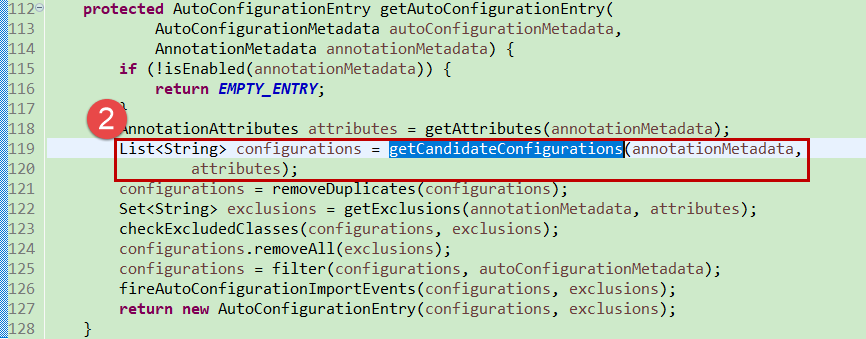


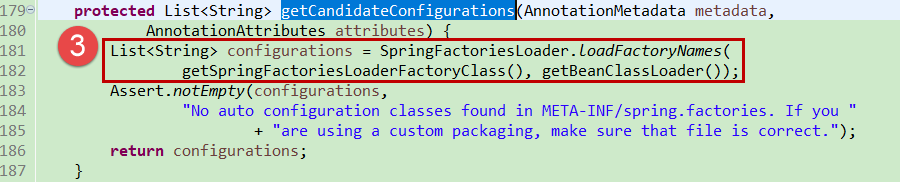
@EnableAutoConfiguration作为一个复合Annotation,其自身定义关键信息如下



其中，最关键的要属@Import(**AutoConfigurationImportSelector**.class)，借助**AutoConfigurationImportSelector**，@EnableAutoConfiguration可以帮助SpringBoot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器。



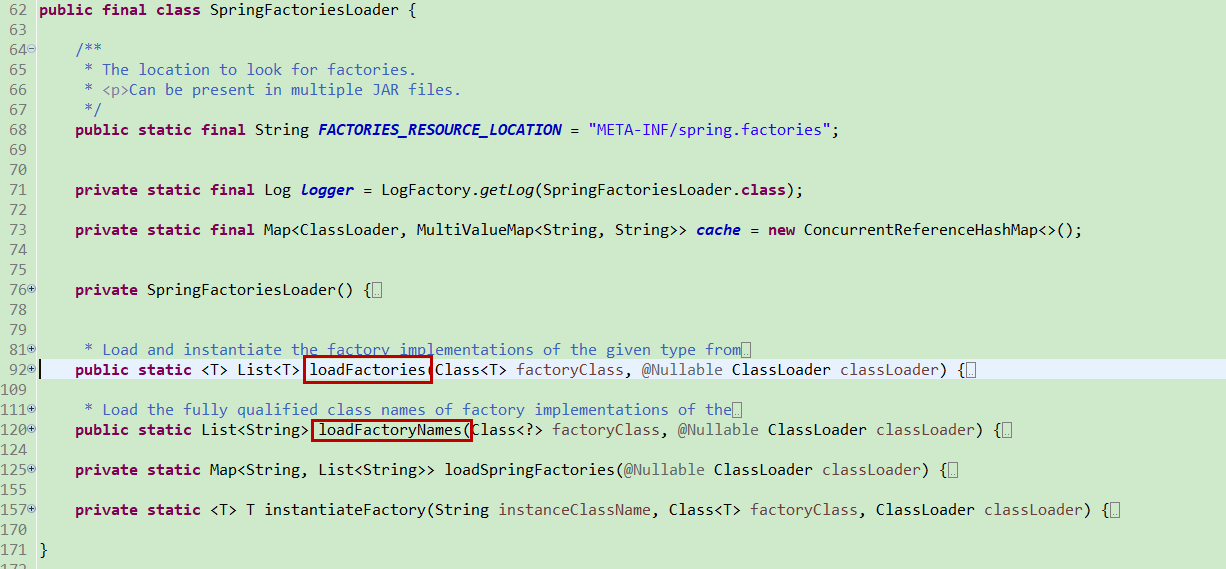




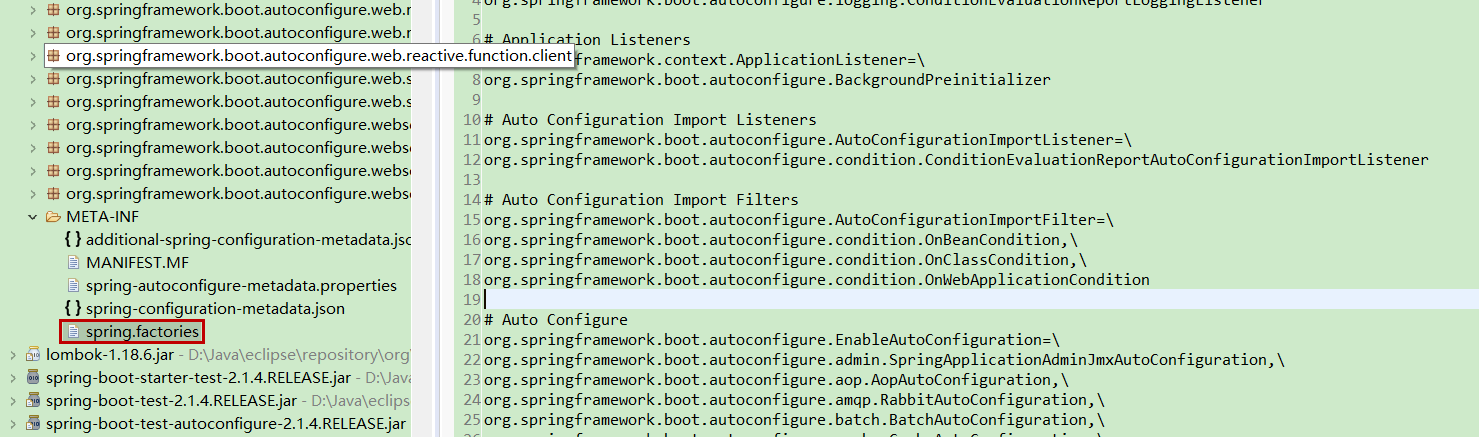
借助于Spring框架原有的一个工具类：SpringFactoriesLoader的支持，@EnableAutoConfiguration可以智能的自动配置功效才得以大功告成

* + - 1. SpringFactoriesLoader类加载器

SpringFactoriesLoader属于Spring框架私有的一种扩展方案，其主要功能就是从指定的配置文件META-INF/spring.factories加载配置



配合@EnableAutoConfiguration使用的话，它更多是提供一种配置查找的功能支持，即根据@EnableAutoConfiguration的完整类名org.springframework.boot.autoconfigure.EnableAutoConfiguration作为查找的Key,获取对应的一组@Configuration类



上图就是从SpringBoot的autoconfigure依赖包中的META-INF/spring.factories配置文件中摘录的一段内容，可以很好地说明问题。

所以，@EnableAutoConfiguration自动配置就变成了：从classpath中搜寻所有的META-INF/spring.factories配置文件，并将其中org.springframework.boot.autoconfigure.EnableutoConfiguration对应的配置项通过反射（Java Refletion）实例化为对应的标注了@Configuration的JavaConfig形式的IoC容器配置类，然后汇总为一个并加载到IoC容器。

* 1. **Spring JPA**
     1. **JPA中的@Query**
        1. **概述**

Spring Data提供了许多方法来定义我们可以执行的查询。其中之一是 @Query注释，在Spring Data JPA中使用@Query注释执行JPQL和Native SQL查询。

* + - 1. **选择查询**

为了定义要为Spring DataRepository方法执行的SQL，我们可以使用@Query注释对方法进行 注释-其value属性包含要执行的JPQL或SQL。

该@Query注释优先于@NamedQuery，将@Query定义放在Repository中而不是将@NamedQuery放在Model中的方法上是一种很好的方法。

* + - * 1. JPQL

默认情况下，查询定义使用JPQL。让我们看一个简单的Repository方法，该方法从数据库返回活动的User实体；

@Query("SELECT u FROM User u WHERE u.status = 1")

Collection<User> findAllActiveUsers();

* + - * 1. Native SQL

我们还可以使用NativeSQL定义查询。我们要做的就是将nativeQuery属性的值设置为true，并在注释的value属性中定义Native SQL查询：

@Query(

value = "SELECT \* FROM USERS u WHERE u.status = 1",

nativeQuery=true)

Collection<User> findAllActiveUsersNative();

* + - 1. **在查询中定义顺序**

我们可以将Sort类型的附加参数传递给具有@Query注解的Spring Data方法声明。它将转换为传递给数据库的ORDER BY子句。

* + - * 1. JPA提供的派生方法的排序

对于像findAll（Sort）这样的开箱即用的方法或通过解析方法签名生成的方法，我们只能使用对象属性来定义我们的排序：

userRepository.findAll(new Sort(Sort.Direction.ASC, "name"));

现在假设我们要按名称属性的长度进行排序：

userRepository.findAll(new Sort("LENGTH(name)"));

当我们执行上面的代码时，我们将收到一个异常：

org.springframework.data.mapping.PropertyReferenceException：未找到类型为User的属性LENGTH（name）！

* + - * 1. JPQL

当我们使用JPQL作为查询定义时，Spring Data可以毫无问题地处理排序，我们要做的就是添加类型为Sort的方法参数：

@Query(value = "SELECT u FROM User u")

List<User> findAllUsers(Sort sort);

我们可以调用此方法并传递一个Sort参数，该参数将按User对象的name属性对结果进行排序：

userRepository.findAllUsers(new Sort("name"));

并且由于我们使用了@Query批注，因此我们可以使用相同的方法按用户名称的长度来获取用户的排序列表：

userRepository.findAllUsers(JpaSort.unsafe("LENGTH(name)"));

使用JpaSort.unsafe（）创建Sort对象实例至关重要。

当我们使用：new Sort("LENGTH(name)"); 那么我们将收到与上述findAll（）方法完全相同的异常 。

当Spring Data发现使用@Query注释的方法的JpaSort.unsafe时，它仅将sort子句附加到查询中-跳过检查要排序的属性是否属于域模型。

* + - * 1. Native SQL

当@Query注释使用NativeSQL时，则无法定义Sort。如果这样做，我们将收到一个异常：

org.springframework.data.jpa.repository.query.InvalidJpaQueryMethodException：无法将Native SQL与动态排序和/或分页一起使用

如异常所示，Native SQL不支持该排序。该错误消息向我们暗示了分页也会导致异常。

* + - 1. **分页**

分页允许我们仅在Page中返回整个结果的子集。其优点是可以最大程度地减少从服务器发送到客户端的数据量。通过发送较小的数据，我们通常可以看到性能的提高。

* + - * 1. JPQL

在JPQL查询定义中使用分页很简单：

@Query(value = "SELECT u FROM User u ORDER BY id")

Page<User> findAllUsersWithPagination(Pageable pageable);

我们可以传递PageRequest 参数来获取数据页面。Native SQL也支持分页，但是需要一些额外的工作。

* + - * 1. Native SQL

我们可以通过声明其他属性countQuery来启用Native SQL的分页 功能 -这定义了要执行以对整个结果中的行数进行计数的SQL：

@Query(

value = "SELECT \* FROM Users ORDER BY id",

countQuery = "SELECT count(\*) FROM Users",

nativeQuery = true)

Page<User> findAllUsersWithPagination(Pageable pageable);

* + - 1. **索引查询参数**

我们可以通过两种方法将方法参数传递给查询。在本节中，我们将介绍索引参数。

* + - * 1. JPQL

对于JPQL中的索引参数，Spring Data会将方法参数按照在方法声明中出现的顺序传递给查询：

@Query("SELECT u FROM User u WHERE u.status = ?1")

User findUserByStatus(Integer status);

@Query("SELECT u FROM User u WHERE u.status = ?1 and u.name = ?2")

User findUserByStatusAndName(Integer status, String name);

对于上述查询，Status方法参数将被分配给查询参数?1，和Name方法参数将被分配给查询参数具有?2。

* + - * 1. Native SQL

Native SQL的索引参数与JPQL的工作方式完全相同：

@Query(

value = "SELECT \* FROM Users u WHERE u.status = ?1",

nativeQuery = true)

User findUserByStatusNative(Integer status);

在下一节中，我们将展示一种不同的方法-通过名称传递参数。

* + - 1. **@Param命名参数**

我们还可以使用命名参数将方法参数传递给查询。我们使用Repository层中的注解@Param定义它们。

用@Param注释的每个参数必须具有与相应的JPQL或SQL查询参数名称匹配的值的字符串。具有命名参数的查询更易于阅读，并且在需要重构查询的情况下更不会出错。

* + - * 1. JPQL

如上所述，我们在方法声明中使用@Param注解，以将JPQL中按名称定义的参数与方法声明中的参数进行匹配：

@Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")

User findUserByStatusAndNameNamedParams(

@Param("status") Integer status,

@Param("name") String name);

请注意，在上面的示例中，我们将SQL查询和方法参数定义为具有相同的名称，但是只要值字符串相同，则不是必需的：

@Query("SELECT u FROM User u WHERE u.status = :status and u.name = :name")

User findUserByUserStatusAndUserName(@Param("status") Integer userStatus,

@Param("name") String userName);

* + - * 1. Native SQL

对于Native SQL定义，与JPQL相比没有什么区别

@Query(value = "SELECT \* FROM Users u WHERE u.status = :status and u.name = :name",

nativeQuery = true)

User findUserByStatusAndNameNamedParamsNative(

@Param("status") Integer status, @Param("name") String name);

* + - * 1. 其他
      1. **Collection参数**

在JPQL或SQL查询中的where子句包含IN（或NOT IN）关键字的情况：

SELECT u FROM User u WHERE u.name IN :names

在这种情况下，我们可以定义一个以Collection 为参数的查询方法：

@Query(value = "SELECT u FROM User u WHERE u.name IN :names")

List<User> findUserByNameList(@Param("names") Collection<String> names);

由于参数是Collection，可以与List、HashSet等一起使用。

* + - 1. **使用@Modifying更新查询**
         1. JPQL

我们还可以通过将@ Modifying注释添加到Repository层中方法的上，使用@ Query注释来修改数据库的状态。

@Modifying

@Query("update User u set u.status = :status where u.name = :name")

int updateUserSetStatusForName(@Param("status") Integer status,

@Param("name") String name);

返回值定义查询的执行更新了多少行。索引参数和命名参数都可以在更新查询中使用。

* + - * 1. Native SQL

我们还可以使用Native SQL来修改数据库的状态-我们只需要添加@Modifying注解：

@Modifying

@Query(value = "update Users u set u.status = ? where u.name = ?",

nativeQuery = true)

int updateUserSetStatusForNameNative(Integer status, String name);

* + - * 1. 插入对象

要执行插入操作，我们必须同时应用@Modifying并使用Native SQL，因为INSERT不是JPA接口的一部分：

@Modifying

@Query(

value =

"insert into Users (name, age, email, status) values (:name, :age, :email, :status)",

nativeQuery = true)

void insertUser(@Param("name") String name, @Param("age") Integer age,

@Param("status") Integer status, @Param("email") String email);

* + - 1. **动态查询**

通常，我们会遇到根据条件或数据集（仅在运行时才知道值）构建SQL语句的需求。而且，在那种情况下，我们不能只使用静态查询。

* + - * 1. 动态查询的例子

例如，我们根据运行时所选择得电子邮件列表，筛选出相应的用户：

SELECT u FROM User u WHERE u.email LIKE '%email1%'

or u.email LIKE '%email2%'

...

or u.email LIKE '%emailn%'

由于集合是动态构造的，因此在编译时我们无法知道要添加多少个LIKE子句。在这种情况下，我们不能仅使用@Query注释，因为我们不能提供静态SQL语句。相反，通过实现定制的复合存储库，我们可以扩展基本的JpaRepository功能，并提供自己的逻辑来构建动态查询。让我们看看如何执行此操作。

* + - * 1. 定制Repository和JPA Criteria API

Spring提供了一种通过使用自定义片段接口来扩展JpaRepository的方法。然后，我们可以将它们链接在一起以创建一个自定义Repository。

首先创建一个自定义的Repository：

public interface UserRepositoryCustom {

List<User> findUserByEmails(Set<String> emails);

}

实现类型：

public class UserRepositoryCustomImpl implements UserRepositoryCustom {

@PersistenceContext

private EntityManager entityManager;

@Override

public List<User> findUserByEmails(Set<String> emails) {

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<User> query = cb.createQuery(User.class);

Root<User> user = query.from(User.class);

Path<String> emailPath = user.get("email");

List<Predicate> predicates = new ArrayList<>();

for (String email : emails) {

predicates.add(cb.like(emailPath, email));

}

query.select(user)

.where(cb.or(predicates.toArray(new Predicate[predicates.size()])));

return entityManager.createQuery(query)

.getResultList();

}

}

如上所示，我们利用了JPA Criteria API来构建动态查询。

另外，我们需要确保在类名中包含Impl后缀。Spring将搜索UserRepositoryCustom实现作为UserRepositoryCustomImpl。由于实现类不是存储库，因此Spring依靠此机制来查找实现。

* + - * 1. 扩展现有JpaRepository

public interface UserRepository extends JpaRepository<User, Integer>, UserRepositoryCustom {

// query methods from section 2 - section 7

}

* + - * 1. 使用自定义Repository

Set<String> emails = new HashSet<>();

// filling the set with any number of items

userRepository.findUserByEmails(emails);

* + 1. **多表级联查询的5种方法**
       - 1. 映射关系上调用方法
         2. 使用JPQL中的Join Fetch

使用JPQL中的Join Fetch，在同一查询中加载选定的实体和关系，这种方法适用于需要加载的关联对象少，过多的话将导致代码维护困难，且在JPQL语句中使用联接查询时，需要进行语句分析，从而导致产生大量查询，造成性能问题。

Query query = this.EntityManager.createQuery("SELECT d FROM MenuNode d JOIN FETCH d.ChildNodes e WHERE d.Id = :id");

query.setParameter("id", id);

return (T) query.getSingleResult();

* + - * 1. 使用Hibernate的Criteria API中的fetch方法

这种方法和JPQL中的Fetch加载一致，Hibernate使用一个查询从数据库中检索实体和关系，并且每种关联组合都需要特定的代码

CriteriaBuilder builder = this.EntityManager.getCriteriaBuilder();

CriteriaQuery<T> query = builder.createQuery(clazz);

Root<T> root = query.from(clazz);

root.fetch("ChildNodes", JoinType.INNER);

query.select(root);

query.where(builder.equal(root.get("Id"), id));

return (T)em.createQuery(query).getSingleResult();

* + - * 1. 使用JPA中的注解@NamedEntityGraph

@NamedEntityGraph是JPA 2.1的新功能。它可用于定义应从数据库中查询的实体图。实体图的定义是通过注释完成的，并且与查询无关。使用方法如下：

数据模型Model中使用@NamedEntityGraph标注需要加载的子对象

@NamedEntityGraph(name = "TreeNode.Graph", attributeNodes = {@NamedAttributeNode("ChildNodes")})

//@NamedEntityGraphs({

// @NamedEntityGraph(name = "Graph.TreeNode.ChildNodes",

// attributeNodes = {@NamedAttributeNode("items")})

//})

public abstract class TreeNode<T> extends Entity implements Serializable {

@Id

private int Id;

@ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)

@JoinColumn(name = "parentid")

private T parentNode;

@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy = "parentNode")

private Set<T> ChildNodes = new HashSet<T>(0);

}

对应的Repository接口方法下标注使用上面定义的Graph

@EntityGraph(value = "TreeNode.Graph", type = EntityGraph.EntityGraphType.FETCH)

T getTreeNodeWithNestChildById(Class<T> clazz, int id);

* + - * 1. 使用动态EntityGraph

EntityGraph<T> graph = this.EntityManager.createEntityGraph(clazz);

Subgraph<?> employeesGraph = graph.addSubgraph("ChildNodes");

Map<String, Object> props = new HashMap<>();

props.put("javax.persistence.loadgraph", graph);

return em.find(clazz, id, props);

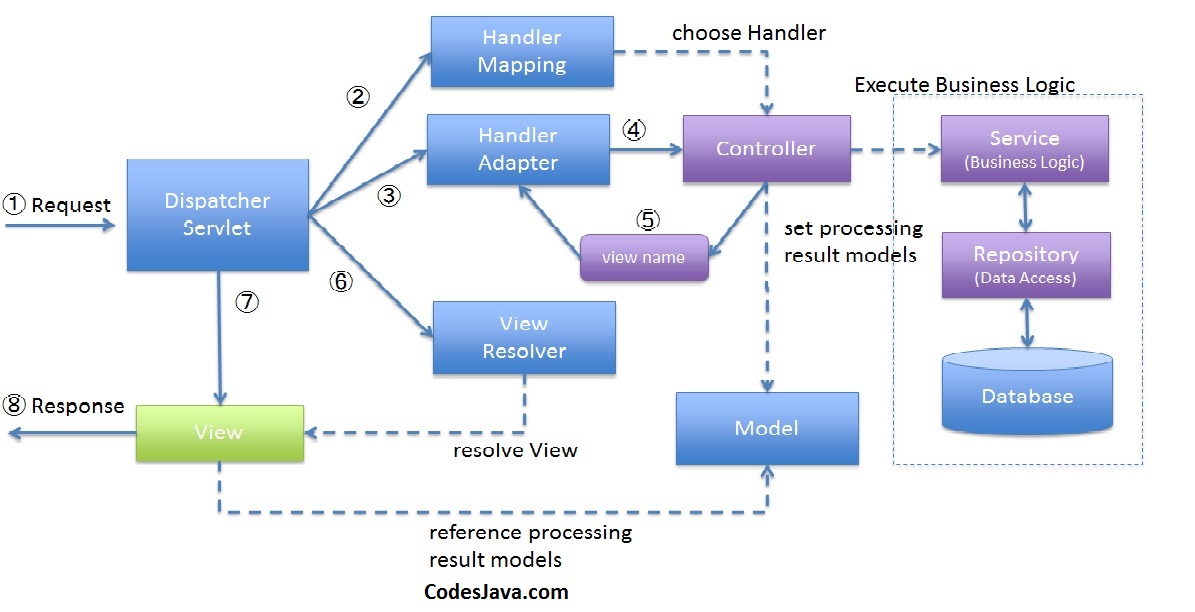
* + - * 1. 使用JPA中的Specifications API
    1. **常用关键字**

|  |  |  |
| --- | --- | --- |
| **Keyword** | **Sample** | **JPQL snippet** |
| **And** | findByNickNameAndName | … where x.NickName = ?1 and x.Name = ?2 |
| **Or** | findByNickNameOrName | … where x.NickName = ?1 or x.Name = ?2 |
| **Between** | findByStartDateBetween | … where x.startDate between 1? and ?2 |
| **LessThan** | findByAgeLessThan | … where x.age < ?1 |
| **GreaterThan** | findByAgeGreaterThan | … where x.age > ?1 |
| **After** | findByStartDateAfter | … where x.startDate > ?1 |
| **Before** | findByStartDateBefore | … where x.startDate < ?1 |
| **IsNull** | findByAgeIsNull | … where x.age is null |
| **IsNotNull,NotNull** | findByAge(Is)NotNull | … where x.age not null |
| **Like** | findByNameLike | … where x.Name like ?1 |
| **NotLike** | findByNameNotLike | … where x.Name not like ?1 |
| **StartingWith** | findByNameStartingWith | … where x.Name like ?1 (parameter bound with appended %) |
| **EndingWith** | findByNameEndingWith | … where x.Name like ?1 (parameter bound with prepended %) |
| **Containing** | findByNameContaining | … where x.Name like ?1 (parameter bound wrapped in %) |
| **OrderBy** | findByAgeOrderByNickNameDesc | … where x.age = ?1 order by x.NickName desc |
| **Not** | findByNickNameNot | … where x.NickName <> ?1 |
| **In** | findByAgeIn(Collection<Age> ages) | … where x.age in ?1 |
| **NotIn** | findByAgeNotIn(Collection<Age> age) | … where x.age not in ?1 |
| **True** | findByActiveTrue() | … where x.active = true |
| **False** | findByActiveFalse() | … where x.active = false |

* + 1. **相关资料**

官网文档：<https://docs.spring.io/spring-data/data-jpa/docs/2.1.5.RELEASE/reference/html/>

* + 1. **其他**
  1. **Spring MVC流程**
     1. **流程图**



* + 1. **其他**
  1. **Web容器（默认使用Tomcat）**
     1. **Jetty容器**
* 修改pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<!-- 排除默认使用tomcat容器 -->

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<!-- 使用Jetty容器 -->

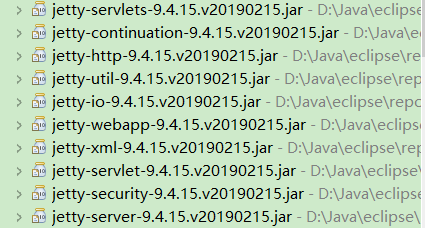
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

* 相关的jar包



* + 1. **undertow容器**

修改pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<!-- 排除默认使用tomcat容器 -->

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<!-- undertow不支持jsp -->

<!-- 使用undertow容器 -->

<dependency>

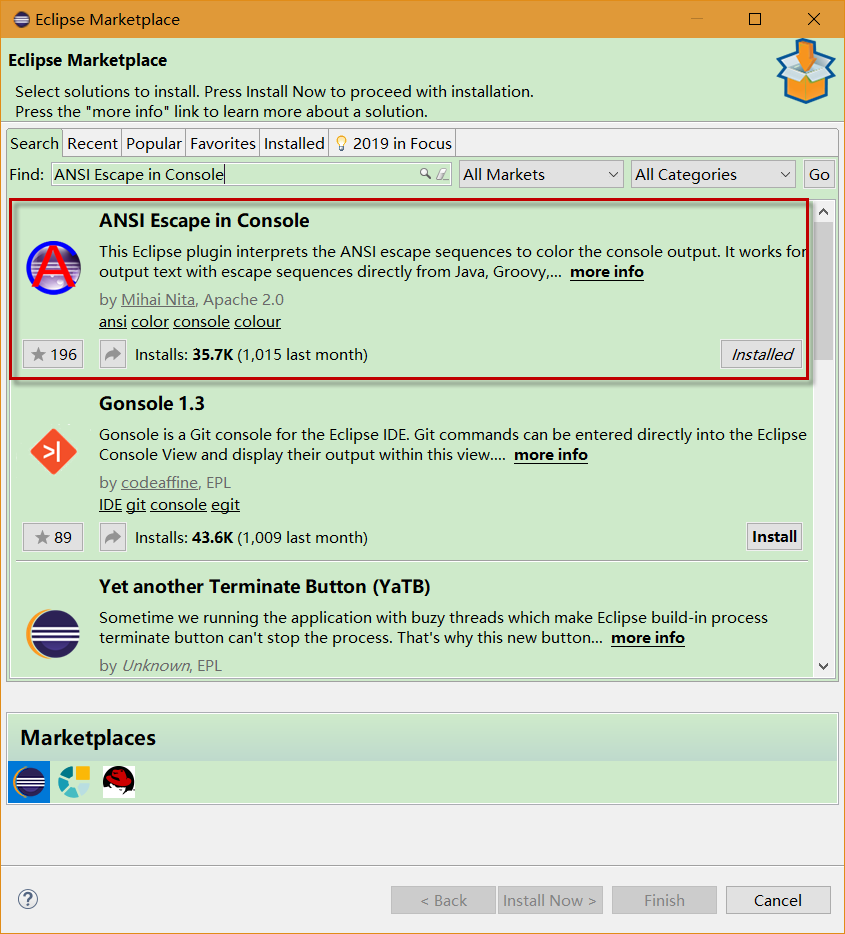
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-undertow</artifactId>

</dependency>

* + 1. **其他**
  1. **其他**

1. **SpringBoot相关插件**
   1. **Console支持彩色输出插件：ANSI Escape in Console**
      1. 进入Eclipse Marketplace，搜索ANSI Escape in Console并插件



* + 1. 设置application.properties

#Console支持彩色输出，需要安装插件：ANSI Escape in Console

spring.output.ansi.enabled=DETECT

* + 1. 配置日志（logback.xml）

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

<configuration scan="true" scanPeriod="3 seconds">

<property name="LOG\_HOME" value="D://logs" />

<!-- 彩色日志 -->

<!-- 彩色日志依赖的渲染类 -->

<conversionRule conversionWord="clr" converterClass="org.springframework.boot.logging.logback.ColorConverter" />

<conversionRule conversionWord="wex" converterClass="org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter" />

<conversionRule conversionWord="wEx" converterClass="org.springframework.boot.logging.logback.ExtendedWhitespaceThrowableProxyConverter" />

<!-- 彩色日志格式 -->

<property name="CONSOLE\_LOG\_PATTERN" value="${CONSOLE\_LOG\_PATTERN:-%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}}" />

<!--设置日志输出为控制台-->

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>${CONSOLE\_LOG\_PATTERN}</pattern>

<charset>utf8</charset>

</encoder>

</appender>

<!-- 不带彩色的日志在控制台输出时候的设置 -->

<!-- <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">

格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>

</encoder>

</appender> -->

<!-- 按照每天生成日志文件 -->

<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">

<!-- 日志记录器的滚动策略，按日期，按大小记录 -->

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<!--日志文件输出的文件名-->

<FileNamePattern>${LOG\_HOME}/MultiTenant-%d{yyyy-MM-dd}.log</FileNamePattern>

<!--日志文件保留天数-->

<MaxHistory>30</MaxHistory>

</rollingPolicy>

<!-- 追加方式记录日志 -->

<append>true</append>

<!-- 日志文件的格式 -->

<encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">

<!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符-->

<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>

<charset>utf-8</charset>

</encoder>

<!--日志文件最大的大小-->

<triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">

<MaxFileSize>10MB</MaxFileSize>

</triggeringPolicy>

</appender>

<!-- 日志输出级别 -->

<root>

<level value="INFO"/>

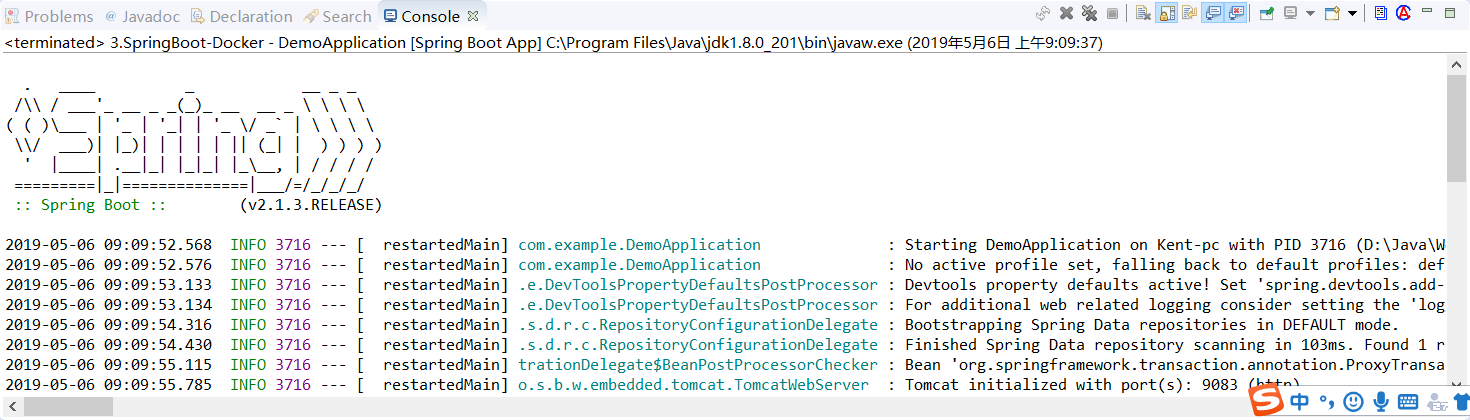
<appender-ref ref="STDOUT"/>

<appender-ref ref="FILE"/>

</root>

</configuration>

* + 1. 运行结果



* 1. **lombok插件：实体类getter/setter自动生成**
     1. **在Eclipse中配置lombok插件**
* 从lombok的官方网址：<http://projectlombok.org/，下载lombok.jar>包；
* 拷贝lombok.jar至eclipse安装目录下；
* 修改eclipse启动配置文件：eclipse.ini，添加如下配置

-javaagent:lombok.jar

-Xbootclasspath/a:lombok.jar

* 重新启动Eclipse
  + 1. **在pom.xml中添加lombok相关引用**

<dependencies>

<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<scope>provided</scope>

</dependency>

</dependencies>

* + 1. **常用注解**

@Data：注解在类上；提供类所有属性的 getting 和 setting 方法，此外还提供了equals、canEqual、hashCode、toString 方法

@Setter：注解在属性上；为属性提供 setting 方法

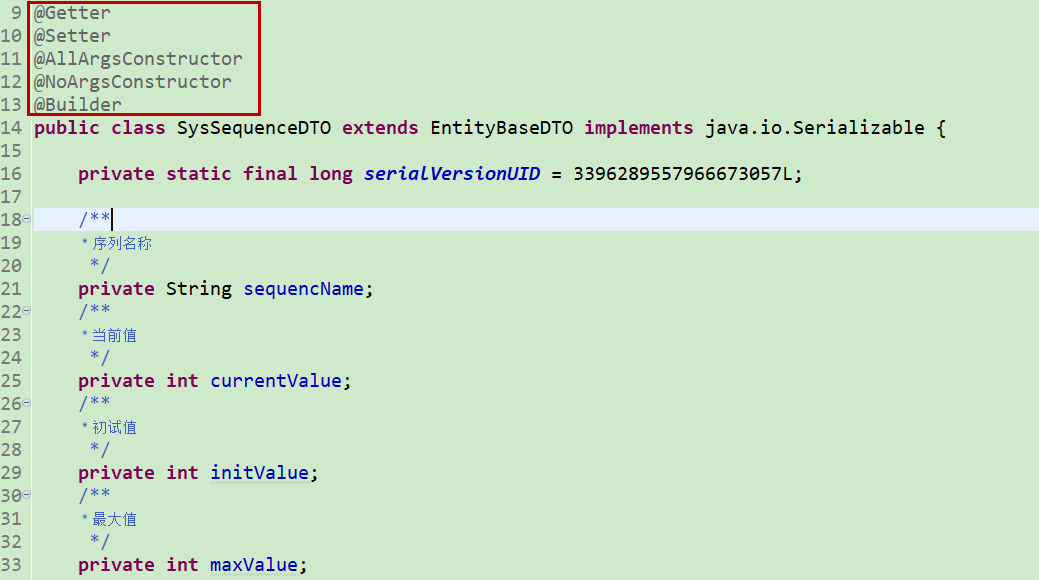
@Getter：注解在属性上；为属性提供 getting 方法

@Log4j ：注解在类上；为类提供一个 属性名为log 的 log4j 日志对象

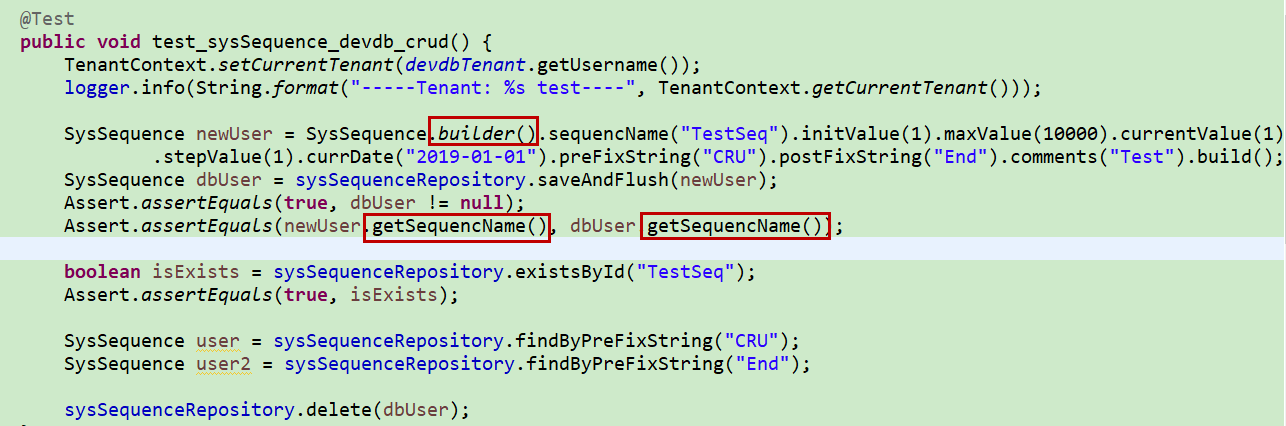
@NoArgsConstructor：注解在类上；为类提供一个无参的构造方法

@AllArgsConstructor：注解在类上；为类提供一个全参的构造方法

* + 1. **在实体类中添加相关注解**



* + 1. **测试代码**



* + 1. **相关资料**

官网：<http://mapstruct.org/documentation/installation/>

* 1. **MapStruct插件：对象映射**
     1. **在pom.xml中添加MapStruct相关引用**

<dependencies>

<dependency>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct</artifactId>

<version>1.3.0.Final</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.5.1</version> <!-- or newer version -->

<configuration>

<source>1.8</source> <!-- depending on your project -->

<target>1.8</target> <!-- depending on your project -->

<annotationProcessorPaths>

<path>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct-processor</artifactId>

<version>1.3.0.Final</version>

</path>

<!-- other annotation processors -->

</annotationProcessorPaths>

</configuration>

</plugin>

</plugins>

</build>

* + 1. **Entity类**

public class Target {

private Long testing;

public Long getTesting() {

return testing;

}

public void setTesting( Long testing ) {

this.testing = testing;

}

}

* + 1. **DTO类**

@Data

public class Source {

private String test;

}

* + 1. **Mapper类**

@Mapper

public interface SourceTargetMapper {

SourceTargetMapper MAPPER = Mappers.getMapper( SourceTargetMapper.class );

@Mapping( source = "test", target = "testing" )

Target toTarget( Source s );

}

* + 1. **单元测试**

public class SourceTargetMapperTest {

@Test

public void testMapping() {

Source s = new Source();

s.setTest( "5" );

Target t = SourceTargetMapper.MAPPER.toTarget( s );

assertEquals( 5, (long) t.getTesting() );

}

}

* + 1. **相关资料**

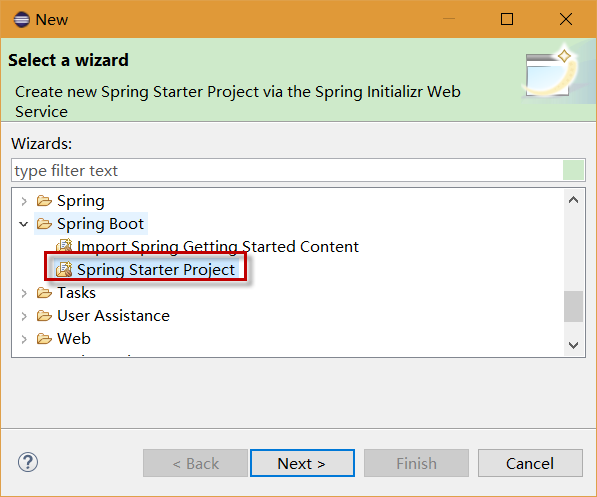
官网：<http://mapstruct.org/documentation/installation/>

Git实例代码：<https://github.com/mapstruct/mapstruct-examples>

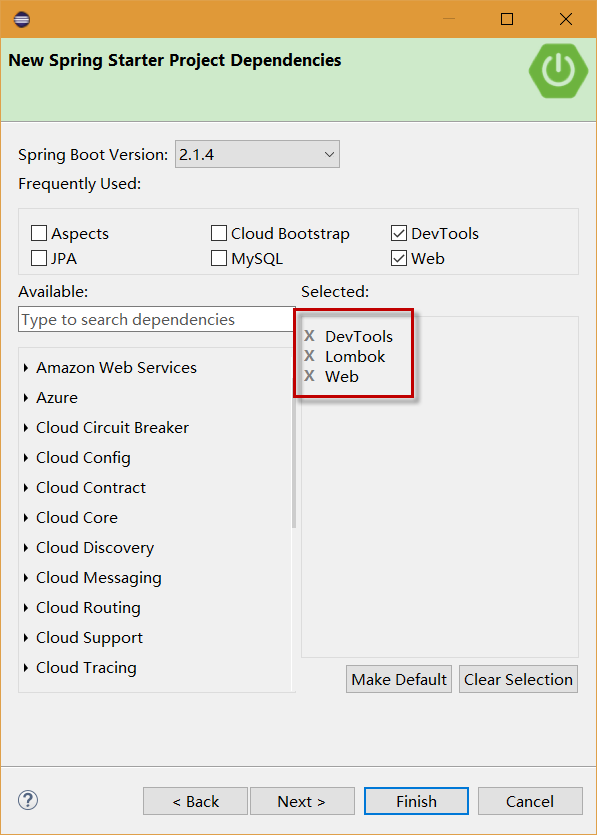
* 1. **其他**

1. **使用Docker部署SpringBoot Web项目**
   1. **创建SpringBoot的Web项目**

* 创建SpringBoot项目



* 选择项目依赖项



* 创建HelloController，添加index方法，返回字符串：Hello Docker！

@RestController

public class HelloController {

@RequestMapping("/")

public String index() {

return "Hello Docker!";

}

}

* 设置application.properties

#设置应用名

spring.application.name=Bootstrap Spring Boot

#设置启动端口

server.port=9090

#热部署生效

spring.devtools.restart.enabled: true

* 1. **Web项目添加Docker支持**
     1. **添加Docker插件**
* pom.xml文件添加相关属性

<properties>

<java.version>1.8</java.version>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<docker.image.prefix>springboot</docker.image.prefix>

<maven.build.timestamp.format>yyyyMMddHHmmss</maven.build.timestamp.format>

</properties>

* pom.xml中添加Docker插件

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

<!-- Docker maven plugin -->

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>1.0.0</version>

<configuration>

<!-- 镜像名称 -->

<imageName>${docker.image.prefix}/${project.artifactId}</imageName>

<!-- 镜像标签 -->

<imageTags>

<!--可以指定多个标签 建议弄成自增序列或者时间戳类型，用于区分版本 -->

<imageTag>${maven.build.timestamp}</imageTag>

</imageTags>

<baseImage>java</baseImage>

<entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>

<!-- dockerfile文件路径 -->

<dockerDirectory>src/main/docker</dockerDirectory>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

</resource>

</resources>

</configuration>

</plugin>

<!-- Docker maven plugin -->

</plugins>

</build>

* + 1. **添加Dockerfile文件**

在项目路径（src/main/docker）下添加dockerfile文件，文件内容如下：

FROM openjdk:8-jdk-alpine

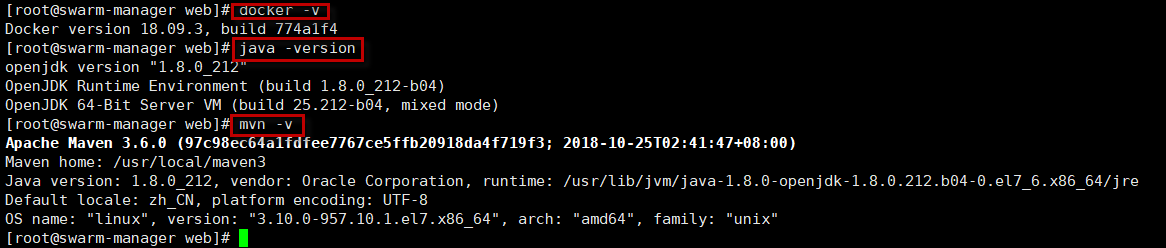
VOLUME /tmp

ADD SpringBootInDocker-1.0.0.1-SNAPSHOT.jar app.jar

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

* 1. **将项目部署至Docker**
     1. **检测环境**

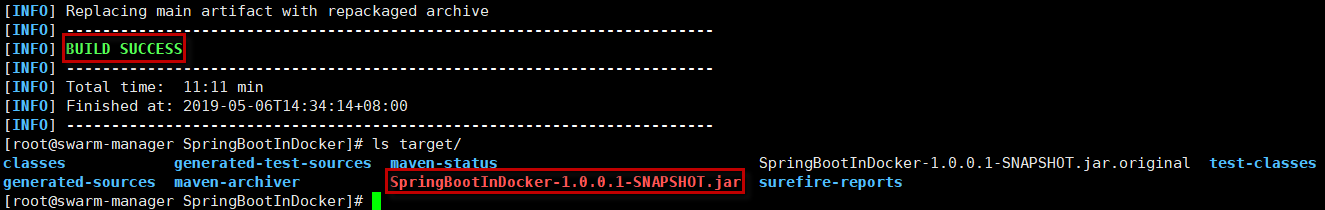
查看Linux系统是否安装Docker、Java、Maven等环境



* + 1. **测试项目**
* 打包Java项目

mvn pacakge

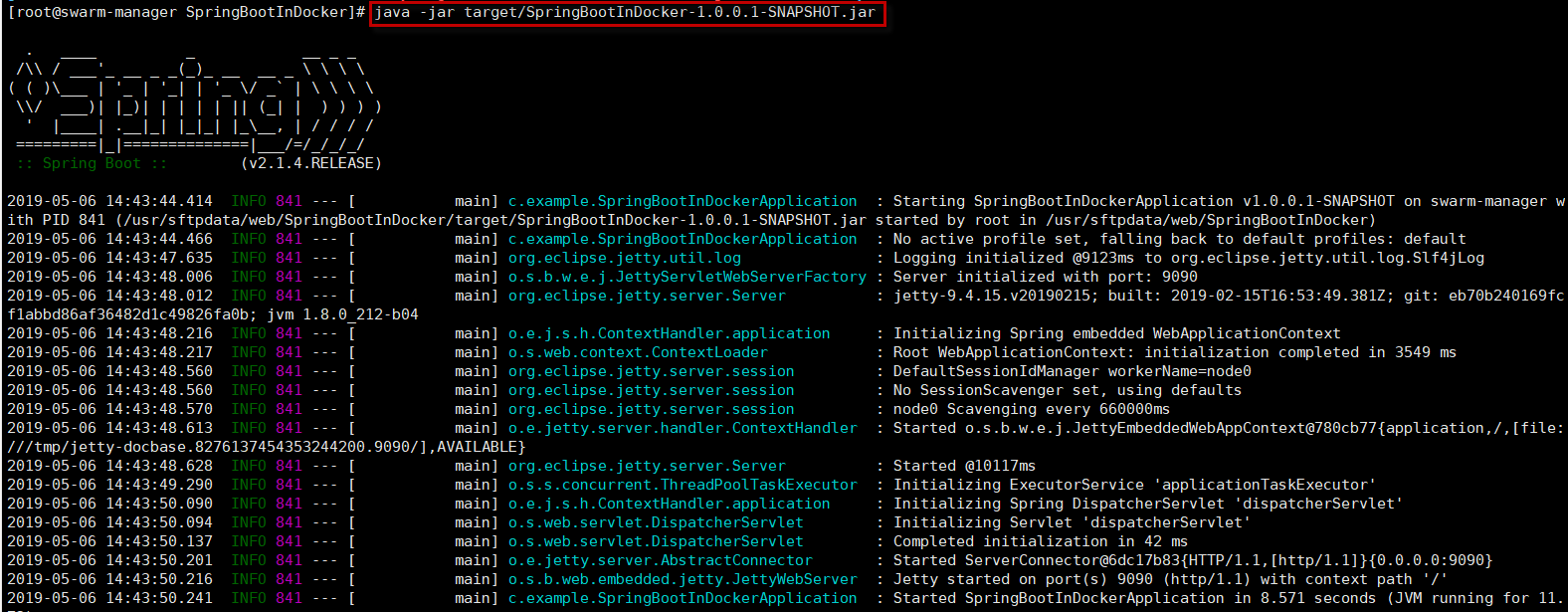
打包结果如下：



* 测试项目是否正常运行

java -jar target/SpringBootInDocker-1.0.0.1-SNAPSHOT.jar

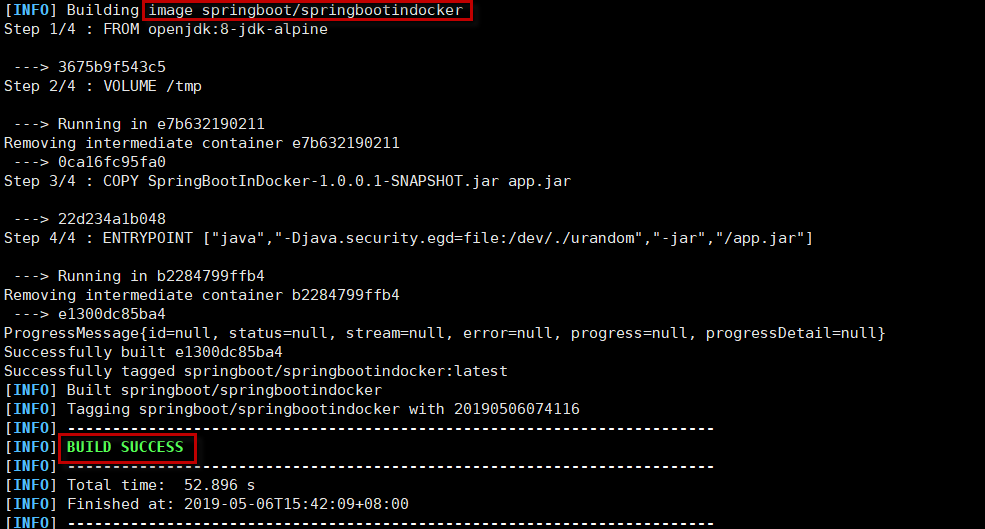
运行结果如下：



* 访问项目：curl http://192.168.56.99:9090
  + 1. **生成Docker镜像**

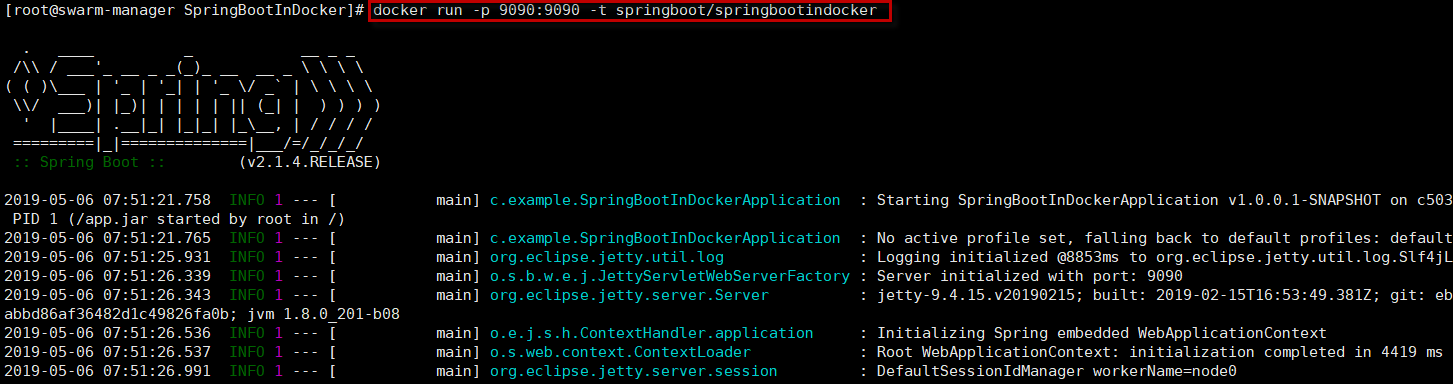
mvn package docker:build

查看结果如下：



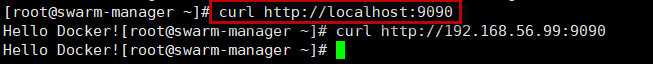
* + 1. **运行Docker镜像**

docker run -p 9090:9090 -t springboot/springbootindocker



* + 1. **测试Docker项目**

curl <http://192.168.56.99:9090>

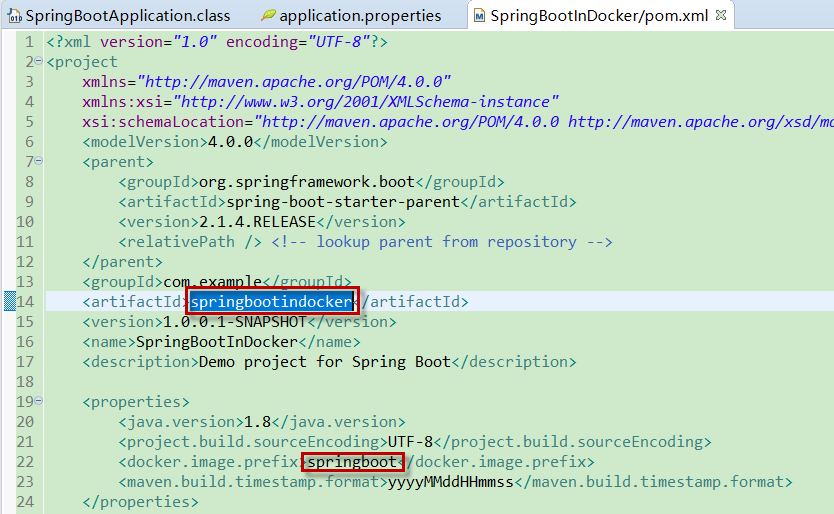


* 1. **相关问题**
     1. **mvn docker:build fails with "{}->unix://localhost:80: Connection reset by peer**

相关问题：<https://github.com/spotify/docker-maven-plugin/issues/357>

原因：docker镜像名称不支持大写字母

解决方案：pom.xml文件中的imageName修改成小写字母，即project.artifactId改为小写



* + 1. **other**
  1. **相关资料**
* 官方资料：**<https://spring.io/guides/gs/spring-boot-docker/>**
  1. **其他**

1. **使用Nexus搭建maven/docker私服**
   1. **准备前工作**

在Linux系统下先安装好JDK、Maven

* 1. **下载安装Nexus**
     1. **下载Nexus**

wget https://sonatype-download.global.ssl.fastly.net/repository/repositoryManager/3/nexus-3.18.1-01-unix.tar.gz

* + 1. **解压、重命名**

tar -zxvf nexus-3.18.1-01-unix.tar.gz //解压

mv nexus-3.18.1-01 nexus //重命名

* + 1. **配置Nexus**

如果Linux硬件配置比较低的话，建议修改为合适的大小，否则会出现运行崩溃的现象

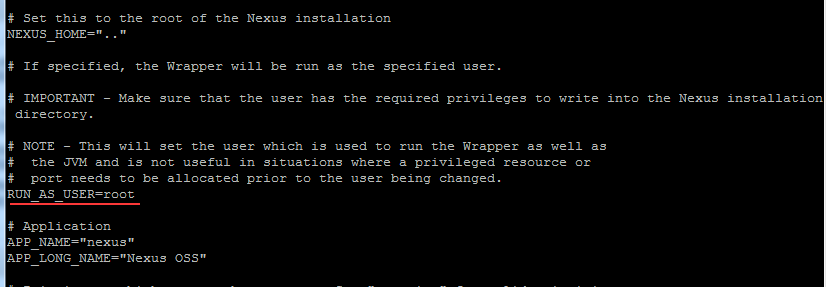
vi nexus/bin/nexus.vmoptions //虚拟机选项配置文件

* + 1. **启动Nexus**

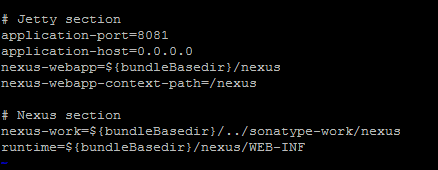
启动 Nexus（默认端口是8081），Nexus 常用的一些命令包括：/usr/local/nexus/nexus/bin/nexus {start|stop|run|run-redirect|status|restart|force-reload}，下面我们启动Nexus：

默认情况下，不建议以root用户运行Nexus，可以修改bin/nexus中的配置跳过警告

vi bin/nexus #（修改RUN\_AS\_USER=root）

 重新启动Nexus：./bin/nexus start

注：Nexus默认端口8081，如果想修改端口。修改/conf/nexus.properties文件



* + 1. **修改nexus3配置文件（nexus.vmoptions）**

vi bin/nexus.vmoptions

#调整Nexus内存参数，以防占用内存太大

-Xms256M

-XmX512M

-XX:MaxDirectMemerySize=512M

* + 1. **其他**
  1. **使用Docker安装Nexus**
     1. **查找镜像，并选取使用次数较多的镜像进行拉取**

docker search nexus

docker pull sonatype/nexus3

mkdir /opt/nexus-data

chmod 777 /opt/nexus-data

* + 1. **启动镜像**

docker run -d --name nexus3 \

--restart=always \

-m 512m \

-p 8081:8081 \

-p 8082:8082 \

-p 8083:8083 \

-p 8084:8084 \

-p 8085:8085 \

-v /opt/nexus-data:/nexus-data \

-e MIN\_HEAP=256m -e MAX\_HEAP=512m \

-e JAVA\_OPTS=’-Xms256m’ -e JAVA\_OPTS=’-Xmx512m’ -e JAVA\_OPTS=’-XX:MaxDirectMemorySize=512m’ \

sonatype/nexus3

* + 1. **查看容器日志**

docker logs nexus3

docker exec -it nexus3 /bin/bash #进入docker容器内部

* + 1. **相关资料**

Nexus官方：<https://hub.docker.com/r/sonatype/nexus/>

* + 1. **相关问题**
       1. **启动nexus时，没有权限操作宿主机文件夹**

相关日志：

Warning: Cannot open log file: ../sonatype-work/nexus3/log/jvm.log

Warning: Forcing option -XX:LogFile=/tmp/jvm.log

java.io.FileNotFoundException: ../sonatype-work/nexus3/tmp/i4j\_ZTDnGON8hezynsMX2ZCYAVDtQog=.lock (No such file or directory)

at java.io.RandomAccessFile.open0(Native Method)

at java.io.RandomAccessFile.open(RandomAccessFile.java:316)

at java.io.RandomAccessFile.<init>(RandomAccessFile.java:243)

at com.install4j.runtime.launcher.util.SingleInstance.check(SingleInstance.java:72)

at com.install4j.runtime.launcher.util.SingleInstance.checkForCurrentLauncher(SingleInstance.java:31)

at com.install4j.runtime.launcher.UnixLauncher.checkSingleInstance(UnixLauncher.java:88)

at com.install4j.runtime.launcher.UnixLauncher.main(UnixLauncher.java:67)

Unable to update instance pid: Unable to create directory /nexus-data/instances

/nexus-data/log/karaf.log (No such file or directory)

Unable to update instance pid: Unable to create directory /nexus-data/instances

mkdir: cannot create directory '../sonatype-work/nexus3/log': Permission denied

mkdir: cannot create directory '../sonatype-work/nexus3/tmp': Permission denied

OpenJDK 64-Bit Server VM warning: Cannot open file ../sonatype-work/nexus3/log/jvm.log due to No such file or directory

解决方案：修改宿主文件夹权限

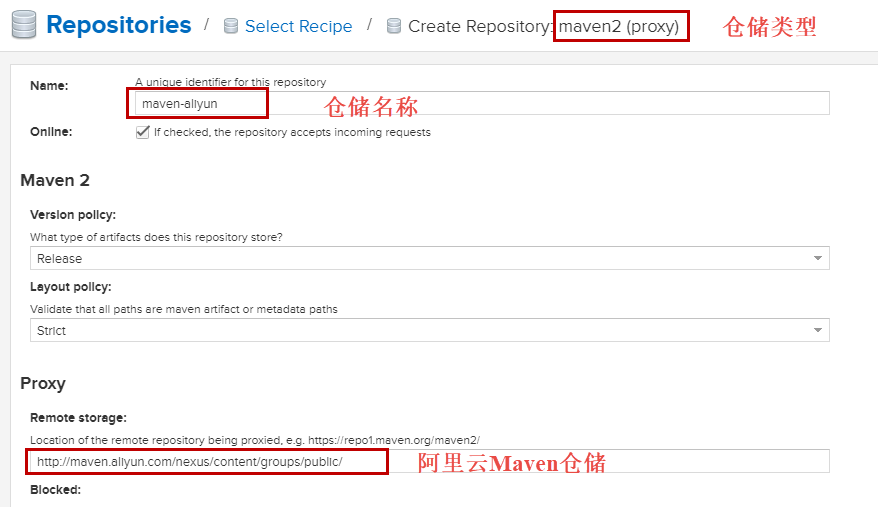
chmod 777 /opt/nexus-data

* + - 1. **设置端口8082，拒绝访问**

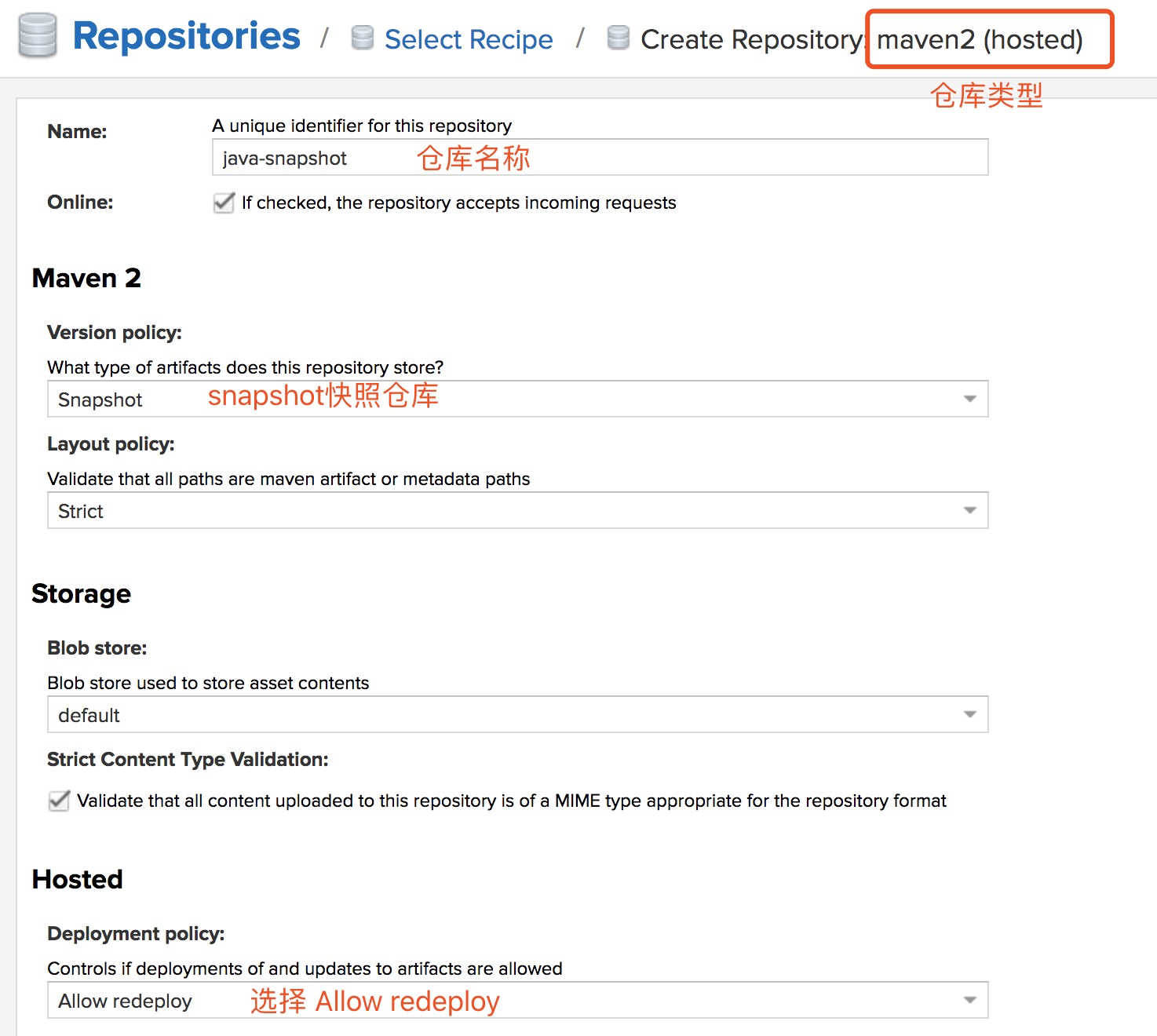
解决方案：修改nexus配置文件 nexus.properties

application-port=8082 #修改成自己想要的端口

* 1. **配置Maven私有仓**
     1. **配置Proxy仓，将默认仓设置为阿里云中央仓**

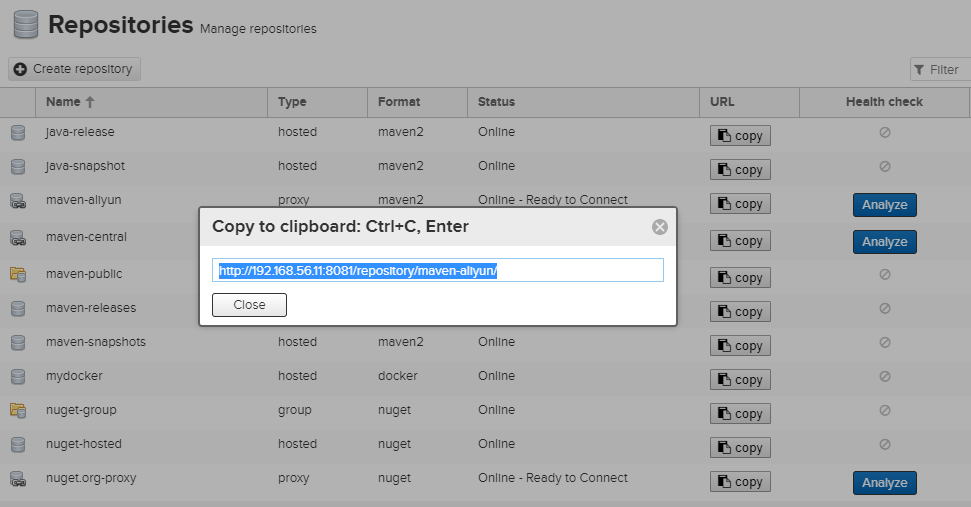


* + 1. **配置hosted仓，包括两类**





* + 1. **其他**
  1. **Maven私有仓的使用**
     1. **查找Maven私有仓地址**



* + 1. **修改本地Maven的全局setting.xml，指向Maven proxy私有仓**

vi /usr/local/maven3/conf/settings.xml

添加如下配置：

<mirror>

<id>local</id>

<mirrorOf>\*</mirrorOf>

<name>local Maven</name>

<url>http://192.168.56.11:8081/repository/maven-aliyun</url>

</mirror>

* + 1. **设置本地Maven使用maven hosted私有仓进行发布**

<server>

<id>java-release</id>

<username>username</username>

<password>userpassword</password>

</server>

<server>

<id>java-snapshot</id>

<username>username</username>

<password>userpassword</password>

</server>

* + 1. **修改项目的pom.xml，指向Maven hosted私有仓**

<distributionManagement>

<repository>

<id>java-release</id>

<name>Local Nexus Release Repository</name>

<url>http://192.168.56.11:8081/repository/java-release/</url>

</repository>

<snapshotRepository>

<id>java-snapshot</id>

<name>Local Nexus Snapshot Repository</name>

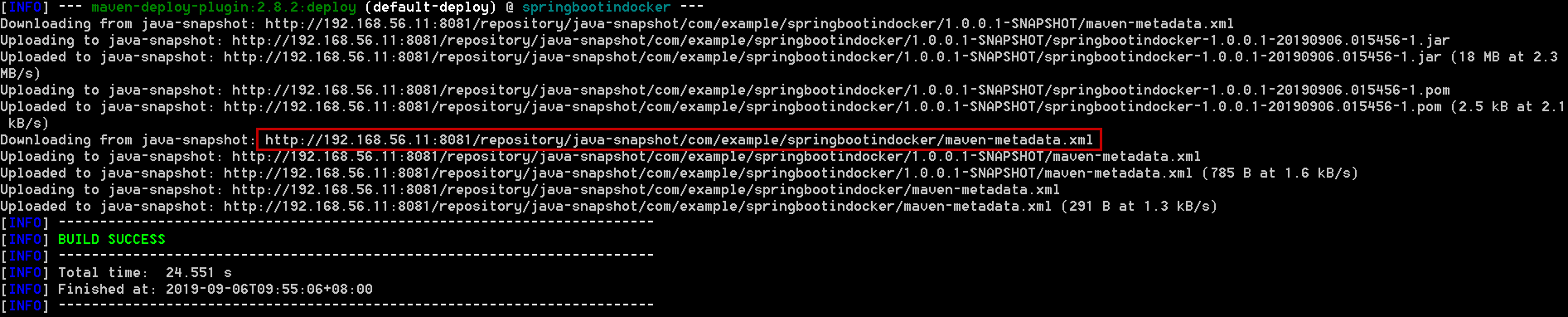
<url>http://192.168.56.11:8081/repository/java-snapshot/</url>

</snapshotRepository>

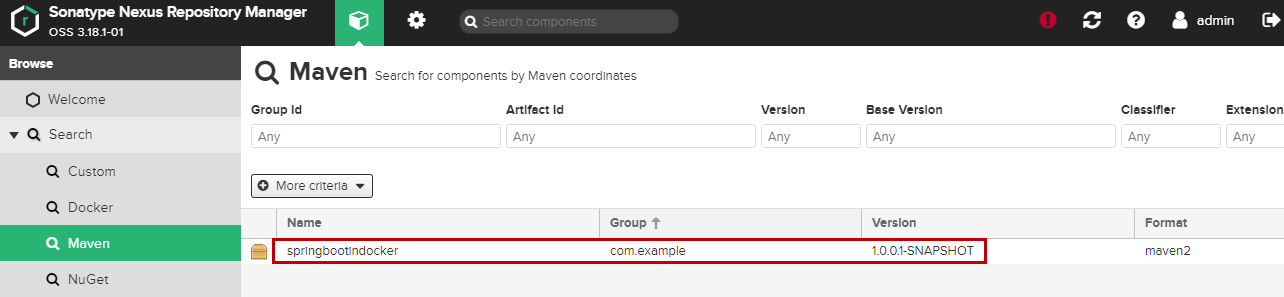
</distributionManagement>

备注：Id需要和setting.xml中设置的Server相关的Id保持一致，以及username、password需要能够访问私有仓的权限

* + 1. **执行项目执行mvn deploy命令，发布至Maven hosted私有仓**



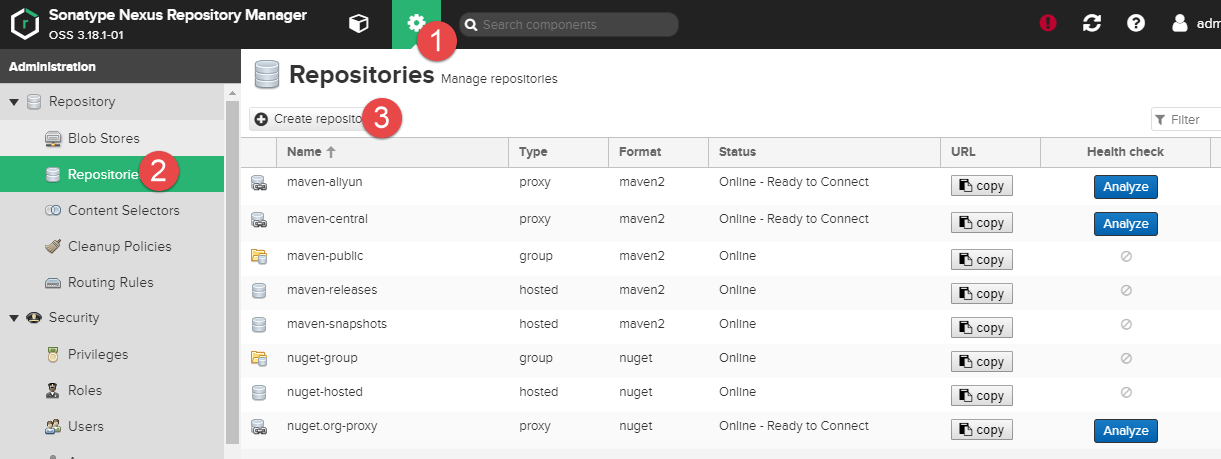
* + 1. **查看发布的jar包**



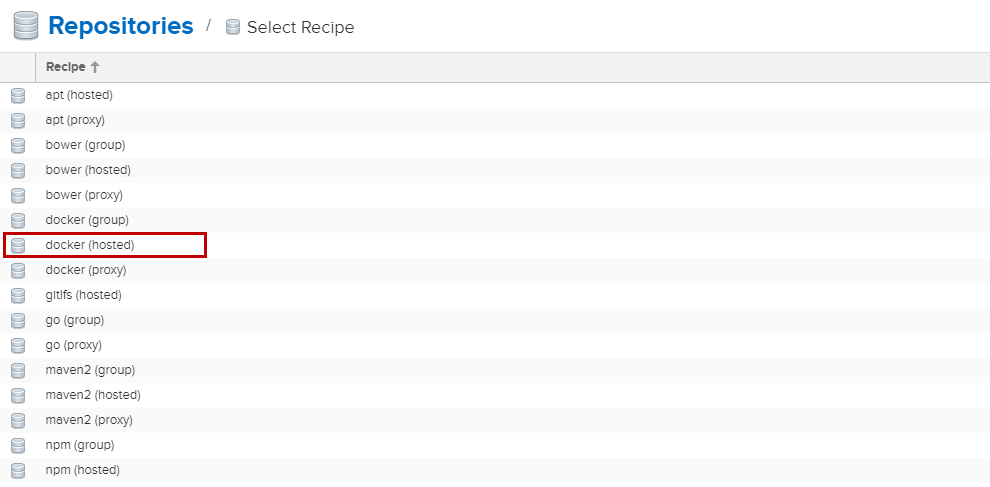
* + 1. **其他**
  1. **配置Docekr私有仓**
     1. **确保正常启动后 使用浏览器访问<http://服务器ip:8081>**

点击右上角登录 账号密码：admin/admin123

登录后点击设置界面 选择Repositories，点击Create repository，如图所示

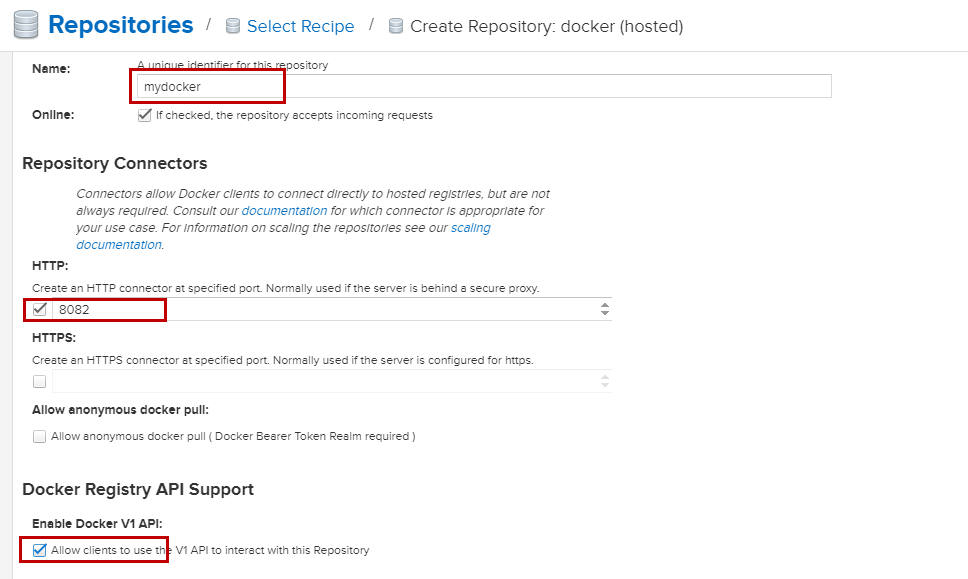


* + 1. **选择仓库类型，这里选择hosted类型**



* + 1. **配置仓库**

该仓库指定一个唯一的名称、HTTP的端口、允许交互的API等



* + 1. **连接仓库**

其他机器需要连接仓库才能进行push、pull等操作，连接仓库前需要进行配置

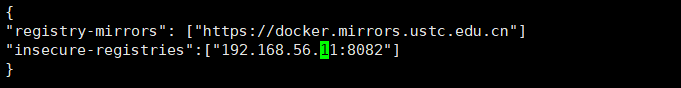
vi /etc/docker/daemon.json #设置如下：

{

"registry-mirrors": ["https://docker.mirrors.ustc.edu.cn"],

"insecure-registries": ["192.168.56.11:8082"]

}

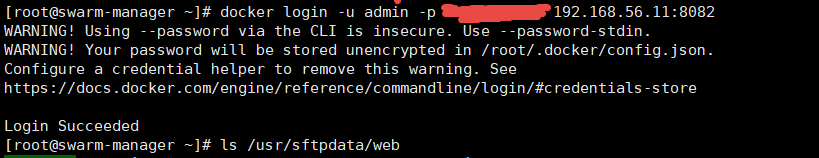
}

systemctl daemon-reload

systemctl restart docker

* + 1. **使用仓库**

docker login -u admin -p admin123 192.168.56.11:8082 #注意这里的端口是配置仓库时选择的端口号



docker tag nginx:latest 192.168.56.11:8082/kc.web.sso:1.0.34-test

docker push 192.168.56.11:8082/kc.web.sso:1.0.34-test #上传镜像

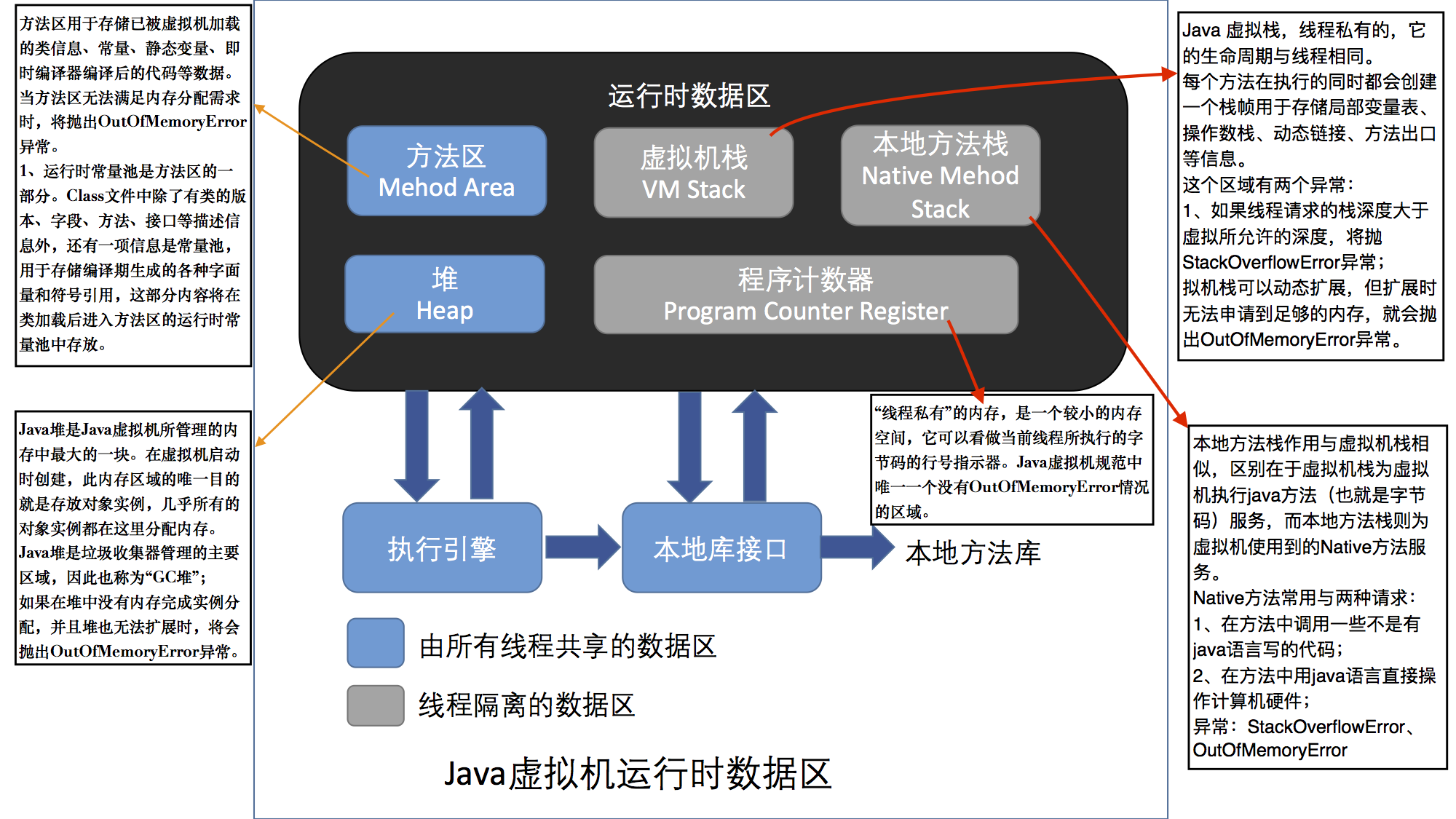
docker pull 192.168.56.11:8082/kc.web.sso:1.0.34-test #拉取镜像

* + 1. **参考资料**

<https://help.sonatype.com/repomanager3>

* 1. **其他**

1. **Java内存区域**



1. **其他**