编译构建

# 常见问题

| | |
|---|---|
| 文档版本 | 01 |
| 发布日期 | 2024-08-07 |

华为技术有限公司

## 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。

# 安全声明

## 漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

# 目 录



# 1 通用构建问题

## 1.1 执行构建任务时，能否指定在某一台/一种配置的服务器上运行?

使用内置执行机时无法指定。

目前编译构建服务采取空闲服务器随机分配的方式，暂不支持指定特定机器执行构建任务。

可使用自定义执行实现，即，自定义资源池，且该资源池中只有一台执行机。

自定义执行机的指导可参考**新建CodeArts资源池**。

## 1.2 如何使用构建并发包

本节以用户当前使用X86/ARM 4U8G规格的执行机为例，且购买的套餐中默认单个构建任务并发执行数为5个。

- 当用户购买五个X86/ARM 4U8G并发包后，可以提升单个构建任务并发执行数到10个，且无需任何配置，在达到默认执行资源上限后，自动使用已购买的4U8G并发包资源。
- 当用户购买五个X86/ARM 8U16G并发包后，可以提升单个构建任务并发执行数到10个，因执行机规格不同，需要进行配置方可使用，使用场景及配置方法见下方说明：
  - 场景一：BuildFlow中共配置了10个子任务，默认单个构建任务并发执行数为5个，为了追求更高的执行效率，用户选择购买5个8U16G并发包，即使用5个默认执行资源，使用5个并发包资源。用户在使用并发资源的这5个任务的yaml文件中配置资源池即可。
  - 场景二：用户仅使用图形化构建，购买8U16G规格的并发包后，即可在**编辑构建任务**页面的“构建步骤 > 构建环境配置”中配置使用高规格执行机。
- 用户使用自定义执行机执行构建任务，单个构建任务并发执行数为10个，若想提升并发执行数，只需购买所需数量的自定义执行机并发包。

# 1.3 执行构建时找不到必须的项目文件

## 问题现象

使用Maven等工具构建时，通常会依赖特定的构建文件，如：pom.xml文件等。如果工具找不到相应的构建文件，则会失败并报“xxx工程找不到xxx文件”此类错误，常见的错误信息如下：

| 工具 | 构建文件 | 错误信息 |
| --- | --- | --- |
| Maven | pom.xml | The goal you specified requires a project to execute but there is no POM in this directory (). Please verify you invoked Maven from the correct directory |
| Ant | build.xml | Buildfile: build.xml does not exist! |
| NPM | package.json | npm ERR! enoent ENOENT: no such file or directory, open '/package.json' |
| Yarn | package.json | error Couldn't find a package.json file in "" |

## 原因分析

以上报错可能的原因有：

- 代码工程目录下没有相应的构建文件。
- 代码工程目录存在嵌套，执行构建命令时所在目录没有相应文件。

## 处理方法

- 检查项目中是否丢失构建工具需要的构建文件。
- 确认构建文件是否位于项目根目录（或是否在构建命令指定的构建文件路径），如有必要先执行“cd”命令进入子目录。

  示例：

  ```
  cd demo-root/demo
  mvn package -Dmaven.test.failure.ignore=true
  ```

# 1.4 上传软件包时找不到文件

## 问题现象

构建任务的“上传软件包到软件发布库”步骤中构建包路径填写错误时，在执行任务时会报错，日志中会出现如下错误信息：

```
[ERROR] [上传软件包到软件发布库:external_nexus_artifact_uploader] : 错误信息: DEV.CB.0220021,未找到文件,可能文件路径不对:**/target/bb.war。
```

## 原因分析

上传软件包到软件发布库的构建步骤，构建包路径配置错误，导致系统找不到对应的文件。如上配置的路径为“**/target/bb.war”，实际target目录下是不存在“bb.war”这个包的。

## 处理方法

- 确定target目录下有war包，只是名字可能不是“bb.war”。
  这种情况下修改构建包路径为“**/target/*.war”，正则匹配war包。
- 无法确定target目录下有哪些文件。
  在构建执行的步骤shell里最后增加“ls -al target”，再次执行构建，就会打印出target目录下的所有文件。找到需要的文件位置后，再重写构建包路径配置。

**说明**

- 如果构建结果不在workspace目录（构建命令在workspace目录或其子目录下执行），则在下一个Action中将丢失此构建包，因此需要提前拷贝构建包到workspace目录，如：mv /usr/bin/nginx ./。
- 相关构建步骤：**上传软件包到软件发布库**。

# 1.5 执行编译构建任务时提示：权限不足，无法获取信息

## 问题现象

执行编译构建任务失败，异常信息为：权限不足，无法获取信息。

## 原因分析

用户不知道自己的角色或者角色被修改时，导致执行编译构建的权限不足，无法操作该任务。

## 处理方法

**步骤1** 联系任务的管理员（任务创建者、项目创建者）配置任务的操作权限。

**步骤2** 进入任务的“权限管理”页面，开启对应操作权限。

**----结束**

# 1.6 通过流水线调用构建任务时，提示任务不存在

## 问题现象

执行流水线失败，流水线上挂载的构建任务报错，异常信息为：任务不存在。

## 原因分析

该报错构建任务被删除，导致流水线执行失败。

## 处理方法

**步骤1** 检查该任务是否被人为删除，且不可以从用户侧恢复。

**步骤2** 尝试重新配置构建任务和流水线。

**步骤3** 如果仍然未能解决，请联系技术支持工程师。

**----结束**

# 1.7 构建任务执行时被中止

## 问题现象

构建任务被中止，异常信息如下：


## 原因分析

编译构建单个构建任务单次构建最大时长限制为：1小时（非付费用户）/4小时（付费用户），构建时长如果超过了系统限定值，系统会强制中止任务执行。

# 1.8 Eclipse 普通 Java 项目上云

## 问题现象

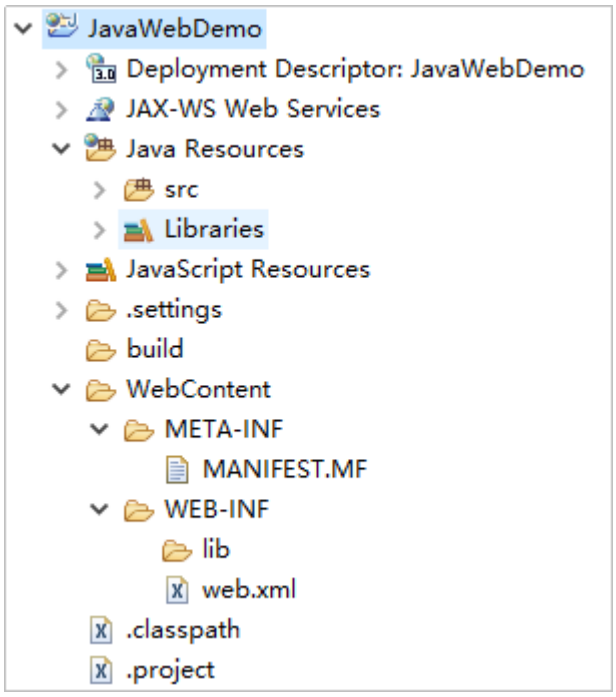
在Eclipse上开发的Java web项目无法在CodeArts Build上构建出包，需要转换项目。本文档将指导您如何将项目改造成Ant项目，在CodeArts Build上使用Ant工具进行构建出包。

## 处理方法

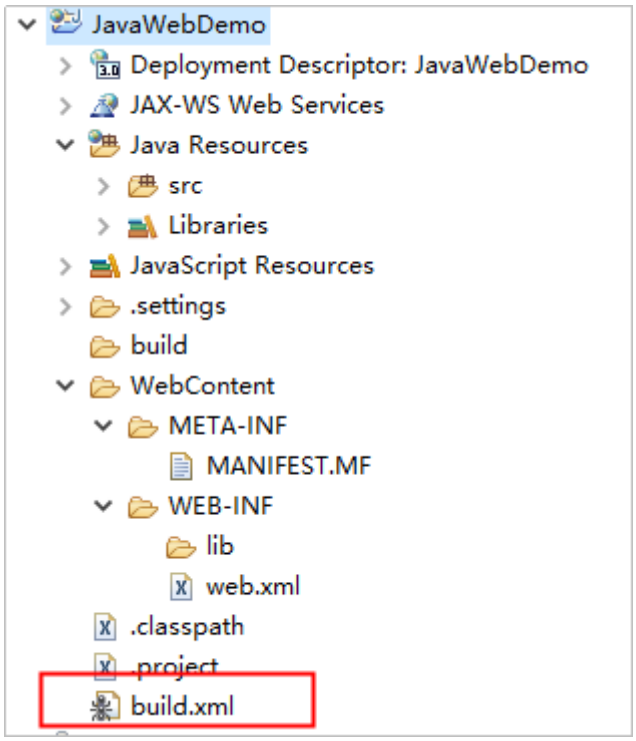
**步骤1** 在项目中创建一个build.xml文件。

下图为一个Eclipse创建出来的web项目的demo的目录结构。

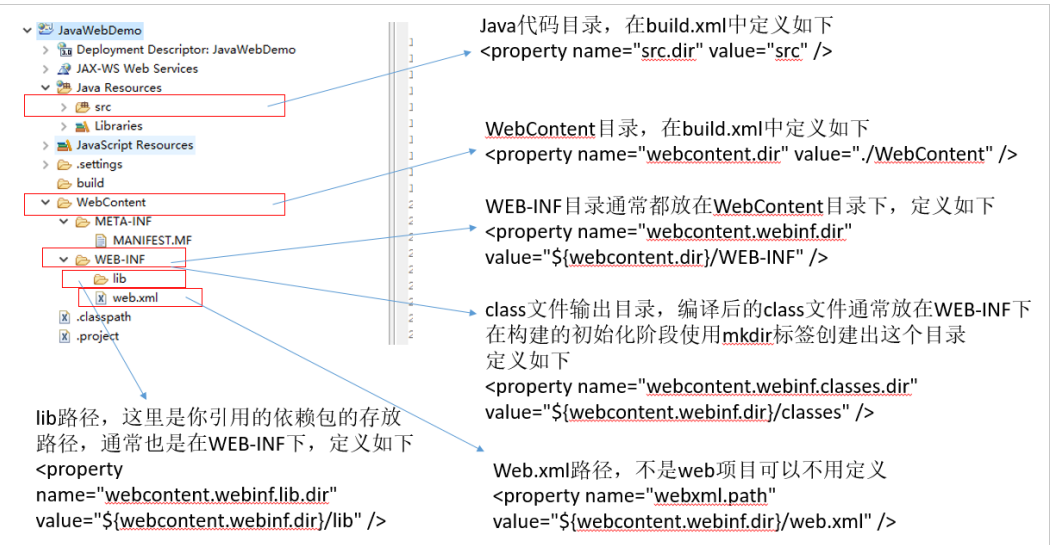



在根目录创建build.xml文件，目录结构变为下图。




配置文件build.xml内容说明如下：




下面是更加详细的说明，说明后有一个完整的build.xml例子，只要将例子中的各个属性修改成真正的项目对应的内容即可。

1. **定义属性部分**
   - 定义项目名称
     ```
     <property name="project.name" value="JavaWebTest" />
     ```
   - 定义包名：打包时生成的.war文件名，这里用到项目名的定义project.name，生成的war包名就是项目名.war。
     ```
     <property name="package.name" value="${project.name}.war" />
     ```
   - war包输出路径：上传软件包时以此路径+包名作为构建包路径,value值就是路径，这里可以自己定义。
     ```
     <property name="dist.war.dir" value="./targets" />
     ```
     **注意：如果“./targets”目录不存在，请在init步骤中创建。**
   - 源代码(*.java)路径，这里的value值指向项目java代码存放的路径。
     ```
     <property name="src.dir" value="src" />
     ```
   - 源码中WebContent目录路径，这里的value指向项目WebContent代码存放的路径。
     ```
     <property name="webcontent.dir" value="./WebContent" />
     ```
   - 源码中WEB-INF目录路径，这里的value指向项目WEB-INF代码存放的路径，通常都是在WebContent目录下，所以引用上面WebContent的路径。
     ```
     <property name="webcontent.webinf.dir" value="${webcontent.dir}/WEB-INF" />
     ```
   - class文件输出目录：通常编译后的class文件放在WEB-INF下。
     ```
     <property name="webcontent.webinf.classes.dir" value="${webcontent.webinf.dir}/classes" />
     ```
     **注意：如果classes目录不存在，请在init步骤中创建。**
   - 定义lib路径，这里是你引用的依赖包的存放路径，通常在WEB-INF下。
     ```
     <property name="webcontent.webinf.lib.dir" value="${webcontent.webinf.dir}/lib" />
     ```
     **注意：如果lib目录不存在，请在init步骤中创建。**
   - 定义java版本。
     ```
     <property name="source.version" value="1.8" />
     <property name="target.version" value="1.8" />
     ```
   - 定义web.xml路径，不是web项目可以不用定义。
     ```
     <property name="webxml.path" value="${webcontent.webinf.dir}/web.xml" />
     ```
2. **定义路径部分**
   - 定义 classpath 路径，如A引用B，A.java的编译在这里寻找B.class文件。

```
<path id="classpath">
   <!-- 项目的jar包 -->
   <fileset dir="${webcontent.webinf.lib.dir}">
      <include name="**/*.jar" />
   </fileset>

   <!-- 项目的classes文件 -->
   <pathelement location="${webcontent.webinf.classes.dir} " />

   <!--这里的需要用到的web服务器的包，可自行下载添加-->
   <!-- web 服务器的jar包 -->
   <!-- <fileset dir="${localWebServer.home}/lib">
      <include name="**/*.jar" />
   </fileset>     -->
</path>
```

3. **定义构建过程**
   - 初始化步骤 （ init ），包含清空war包输出目录、创建 classes 路径等步骤。

| 属性 | 描述 |
| --- | --- |
| <delete>标签 | 删除动作，dir属性就是要删除的目录路径 |
| <mkdir>标签 | 创建目录动作，dir属性就是要创建的目录路径 |
| <echo>标签 | 打印动作，message属性就是要打印的内容 |

上面提到的 war包目录（ dist.war.dir ），编译存放class文件目录（ webcontent.webinf.classes.dir ），存放依赖包的lib目录（ webcontent.webinf.lib.dir ），如果原来项目中没有，就要在这一步创建。

```
<target name="init">
   <echo message="删除targets目录(war包输出目录)" />
   <delete dir="${dist.war.dir}" />
   <echo message="创建targets目录(war包输出目录)" />
   <mkdir dir="${dist.war.dir}" />
   <echo message="创建classes目录" />
   <!-- WebContent 下的 classes -->
   <mkdir dir="${webcontent.webinf.classes.dir}" />
</target>
```

- 编译 java 文件，使用<javac>标签将java文件编译到“dist.classes”下，编译步骤依赖于init步骤创建的classes目录。

| 属性 | 描述 |
| --- | --- |
| depends | depends="init"声明当前步骤需要在init步骤后使用 |
| srcdir | srcdir属性指定上面定义好的java代码的路径属性src.dir |
| destdir | destdir属性指定定义的存放编译完的class文件的classes目录 |
| source，target | 指定编译时使用的jdk版本 |

```
<target name="compile" depends="init">
   <echo message="使用指定的classpath编译源代码,输出到classes目录" />
   <!-- 这里指定jdk版本为1.8 -->
   <javac encoding="utf-8" listfiles="true" srcdir="${src.dir}"
      destdir="${webcontent.webinf.classes.dir}" debug="on" deprecation="false"
      optimize="true" failonerror="true" source="${source.version}" target="${target.version}">
```

```
        <classpath refid="classpath" />
    </javac>
</target>
```

- 将编译完的项目打成war包，首先使用<delete>标签清理掉原有的war包，再使用<war>标签打包。

| 属性 | 描述 |
| --- | --- |
| warfile | 该属性定义打出来的war包的包名，包含路径 |
| webxml | 指定web.xml的路径 |
| <fileset>子标签 | 指定webContent路径 |

```
<target name="war" depends="compile" description="将工程打成 war 包">
    <echo message="生成war包" />
    <delete file="${dist.war.dir}/${package.name}" />
    <war warfile="${dist.war.dir}/${package.name}" webxml="${webxml.path}">
        <fileset dir="${webcontent.dir}">
        </fileset>
    </war>
</target>
```

**说明**

如果是要打jar包，这里就不能用<war>标签，而是用<jar>标签，示例如下：

- jarfile属性和war包的属性类似，是存放打好的jar包存放的路径，需要在上面属性定义阶段定义，basedir属性是编译后的class的目录，就是上面的打war包时定义的webcontent.webinf.classes.dir属性。
- jar包就不需要webxml属性了。
- 如果要打的jar包是需要使用java -jar来执行的可执行jar包，则需要定义manifest，如果只是一个功能性的，被依赖的jar包就不需要了。

  Main-Class指定main函数所在的类。

```
<target name="jar" depends="compile" description="make jar file">
    <!--jar操作，jarfile指定jar包存放路径，basedir为编译后的class的目录-->
    <jar jarfile="${jarfilename}" basedir="${classes}">
        <!--为jar包指定manifest，当然，如果jar包不需要打成runnable的形式，manifest可以不要-->
        <manifest>
            <!--指定main-class-->
            <attribute name="Main-Class" value="demo.SayHello" />
        </manifest>
    </jar>
</target>
```

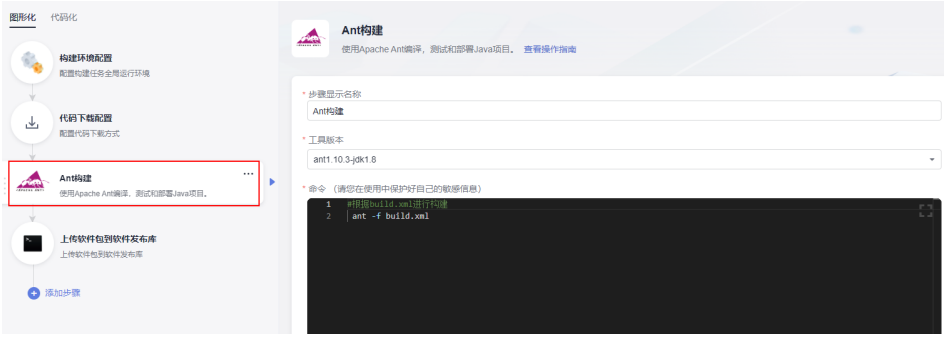
下面是完整的build.xml示例，供参考，通常只需要把属性定义阶段里面那些路径按照实际项目的路径填充好就可以。

```
<?xml version="1.0"?>
<project default="war" basedir=".">
    <echo message="pulling in property files" /> <property file="build.properties" />
    <property name="project.name" value="JavaWebDemo" />
    <property name="package.name" value="${project.name}.war" />
    <property name="dist.war.dir" value="./targets" />
    <property name="src.dir" value="src" />
    <property name="webcontent.dir" value="./WebContent" />
    <property name="webcontent.webinf.dir" value="${webcontent.dir}/WEB-INF" />
    <property name="webcontent.webinf.classes.dir" value="${webcontent.webinf.dir}/classes" />
    <property name="webcontent.webinf.lib.dir" value="${webcontent.webinf.dir}/lib" />
    <property name="source.version" value="1.8" />
    <property name="target.version" value="1.8" />
```

```
    <property name="webxml.path" value="${webcontent.webinf.dir}/web.xml" />

    <path id="classpath">
    <fileset dir="${webcontent.webinf.lib.dir}">
       <include name="**/*.jar" />
    </fileset>
    <pathelement location="${webcontent.webinf.classes.dir} " />
  </path>

    <target name="init">
       <echo message="删除targets目录(war包输出目录)" />
       <delete dir="${dist.war.dir}" />
       <echo message="创建targets目录(war包输出目录)" />
       <mkdir dir="${dist.war.dir}" />
       <echo message="创建classes目录" />
       <mkdir dir="${webcontent.webinf.classes.dir}" />
    </target>


  <target name="compile" depends="init">
     <echo message="使用指定的classpath编译源代码,输出到classes目录" />
     <javac encoding="utf-8" listfiles="true" srcdir="${src.dir}"
       destdir="${webcontent.webinf.classes.dir}" debug="on" deprecation="false"
       optimize="true" failonerror="true" source="${source.version}" target="${target.version}">
       <classpath refid="classpath" />
     </javac>
  </target>

    <target name="war" depends="compile" description="将工程打成 war 包">
       <echo message="生成war包" />
       <delete file="${dist.war.dir}/${package.name}" />
       <war warfile="${dist.war.dir}/${package.name}" webxml="${webxml.path}">
            <fileset dir="${webcontent.dir}">
            </fileset>
       </war>
    </target>
</project>
```

**步骤2** 将改好的build.xml提交到代码仓库，创建Ant类型的构建任务。




上传软件包到软件发布库中的构建包路径就可以按照上面build.xml说明的那样填写war包输出路径加上包名的格式。

**步骤3** 保存任务，执行构建，构建成功之后就可以在软件发布库看到编译打包好的war包。

**----结束**

# 1.9 对应的扩展点不存在

## 问题现象

构建任务执行失败，日志提示“对应的服务扩展点不存在”。

## 原因分析

服务扩展点数据丢失，构建任务如果关联了该服务扩展点，则执行时会报错。

## 处理方法

重新在服务扩展点页面新建服务扩展点，并将服务扩展点重新关联到构建任务中，以构建任务中的“通用Git”服务扩展点丢失为例。

1. 单击导航栏“设置 > 通用设置 > 服务扩展点管理”。
2. 新建通用Git服务扩展点。
3. 返回执行失败的构建任务，编辑该任务，在“源码选择”页签重新关联新建的通用Git服务扩展。
4. 重新执行构建任务。

# 1.10 多任务同时构建导致构建生成 jar 包内容缺失

## 问题现象

构建环境异常或不适当的构建方式可能会导致生成的jar包内容有缺失，但构建结果是成功，导致问题难以定位。

- 前置条件：A项目依赖B项目，同时构建并上传依赖A和依赖B（多人同时构建或流水线设置构建任务并行执行）
- 构建结果：构建任务B结果为成功，构建任务A结果为成功
- 问题描述：依赖B无异常，依赖A偶现内容缺失

## 原因分析

A依赖B且A、B项目同时构建时，可能出现B正在上传且未上传完时，A开始下载B依赖，导致A项目无法完整获取依赖B内容。

## 处理办法

**步骤1** 确定A项目所有依赖的自研项目B1、B2……Bn。

**步骤2** 排查相关流水线，确认是否有项目A与项目Bn并行构建。

**步骤3** 如果找到，修改流水线配置，将A、B项目构建方式改为串行。

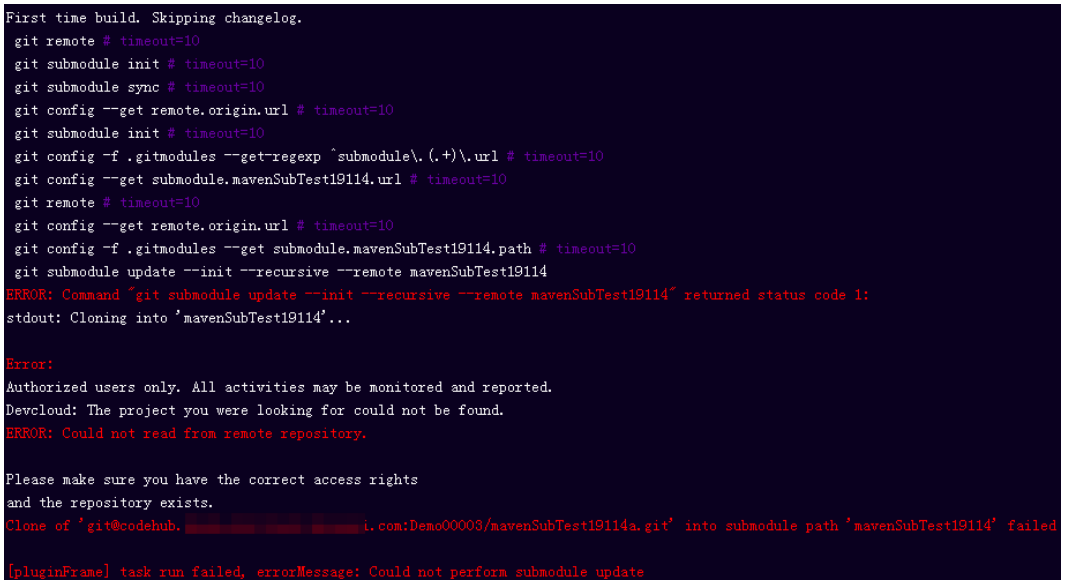
**步骤4** 如果没有，对比A、B项目构建历史，或与相关责任人确认构建时间，确认是否同时构建。

**----结束**

# 1.11 执行构建时拉取子模块代码出错

## 问题现象

执行构建任务时，报如下异常信息：




## 原因分析

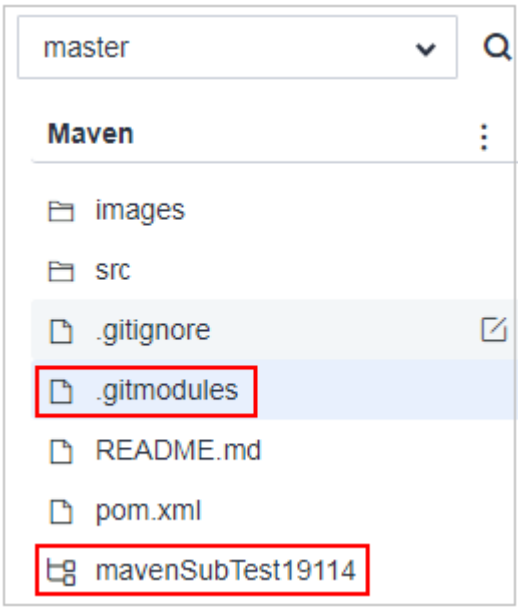
Git从CodeArts Repo拉取子模块时出现错误“Could not read from remote repository”，可能是没有权限或者“.gitmodules”文件配置错误。

## 处理办法

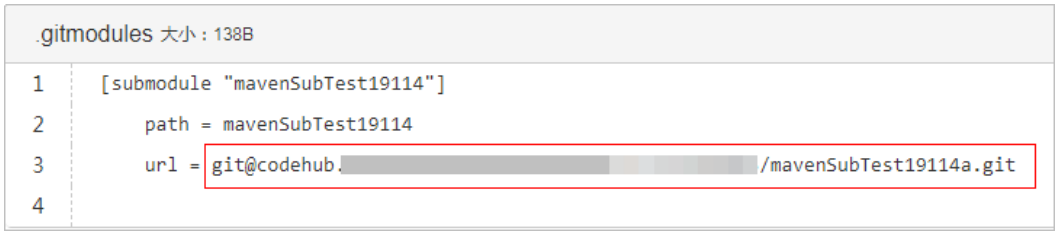
**步骤1** 打开主代码仓库，选择“设置 > 子模块设置”，部署秘钥没有同步，单击同步按钮，之后再尝试编译构建。

**步骤2** 如果步骤1已同步，很可能是主仓库的“.gitmodules”文件配置出错，先检查存在“.gitmodules”文件且子模块是“mavenSubTest19114”。




**步骤3** 打开“.gitmodules”文件，修改成正确的子模块配置“mavenSubTest19114a.git”。




**步骤4** 重新修改好“.gitmodules”，再尝试构建。

**----结束**

# 1.12 执行构建时拉取子模组失败，找不到子模组的修订版本

## 问题现象

异常信息如下：

```
[2019-07-02 08:29:23.179] ERROR: Command "git submodule update --init --recursive --remote asae-feign" returned status code 1:
[2019-07-02 08:29:23.179] stdout: Cloning into 'asae-feign'...
[2019-07-02 08:29:23.179]
[2019-07-02 08:29:23.179] Error: ERROR: Needed a single revision
[2019-07-02 08:29:23.179] Unable to find current origin/develop revision in submodule path 'asae-feign'
[2019-07-02 08:29:23.179]
[2019-07-02 08:29:23.202] [INTERNAL] : [pluginFrame] step run failed, errorMessage: Could not perform submodule update
[2019-07-02 08:29:23.250] [INFO] [代码检出] : StagePostExecution started
[2019-07-02 08:29:23.251] [INFO] [代码检出] : StagePostExecution finished
```

## 原因分析

原先检出的目录有问题，本例的目录为“asae-feign”，这个问题属于Git本身的bug。

### 处理办法

在代码仓库删除该目录，然后重新执行**git submodule update**，然后重新执行构建任务。

## 1.13 执行构建时未拉取子模块

### 问题现象

构建拉取Repo代码时，存在“.gitmodules”文件且确认配置正确，但是没有去拉取子模块。

### 原因分析

此问题一般为没开启子模块自动更新。

### 处理办法

编辑构建任务，选择“代码下载配置”构建步骤，将“子模块（submodules）自动更新”开关打开。

# 2 Maven 构建

## 2.1 执行 Maven 构建时，提示未开通私有依赖仓

### 问题现象

异常信息为：**may be you have not init release repository**。

### 原因分析

没有开通私有依赖仓。

### 处理方法

单击CodeArts首页“服务 > 私有依赖库”，选择开通，待开通完成之后重新执行构建即可。

## 2.2 执行 Maven 构建时，提示 license 信息检查不通过

### 问题现象

异常信息如下：

```
[ERROR] Failed to execute goal org.apache.rat:apache-rat-plugin:0.12:check (rat-check) on project maven: Too many files with unapproved license: 7 See RAT report in: /xxx/slave1/workspace/job_4f1501a3-542c-4f3d-a2bb-8fdbd4d76678_1534924094266/target/rat.txt -&gt; [Help 1]
```

### 原因分析

文件License信息检查不通过。

### 处理方法

在mvn命令中添加参数：

```
apache-rat:check -Drat.numUnapprovedLicenses=600
```

# 2.3 使用 maven deploy 命令上传包失败

## 问题现象

通过执行Maven构建任务上传依赖到私有依赖库时，执行任务时日志报如下异常信息：

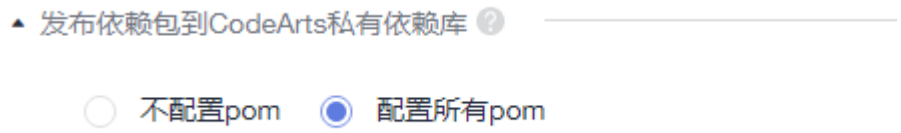
```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy (default-deploy) on project javaMavenDemo: Deployment failed: repository element was not specified in the POM inside distributionManagement element or in -DaltDeploymentRepository=id::layout::url parameter -&gt; [Help 1]
```

## 原因分析

“pom.xml”文件没有正确配置“distributionManagement”。

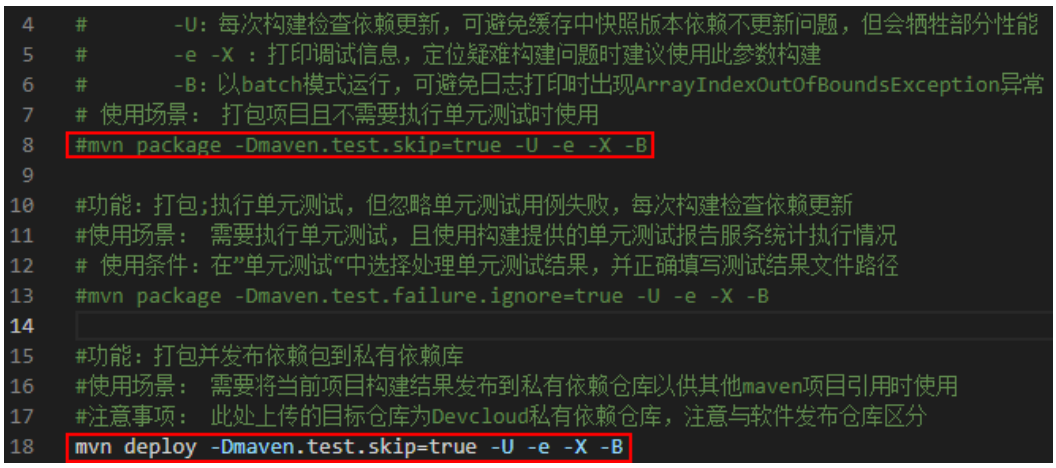
## 处理方法

**步骤1** 配置“Maven构建”构建步骤，展开“发布依赖包到CodeArts私有依赖库”，选择“配置所有pom”。




- 不配置pom：表示无需发布私有依赖包到CodeArts私有依赖库。
- 配置所有pom：表示在项目下所有“pom.xml”文件增加deploy配置，使用**mvn deploy**命令将构建出的依赖包上传到私有依赖仓库。

**步骤2** 在命令窗口，使用“#”注释掉第8行的默认命令，并删除第18行命令前的“#”。




**步骤3** 配置完成后执行构建任务。执行成功后即可将依赖包发布到私有依赖库。

**----结束**

# 2.4 执行 Maven 构建时，提示找不到 pom 文件

## 问题现象

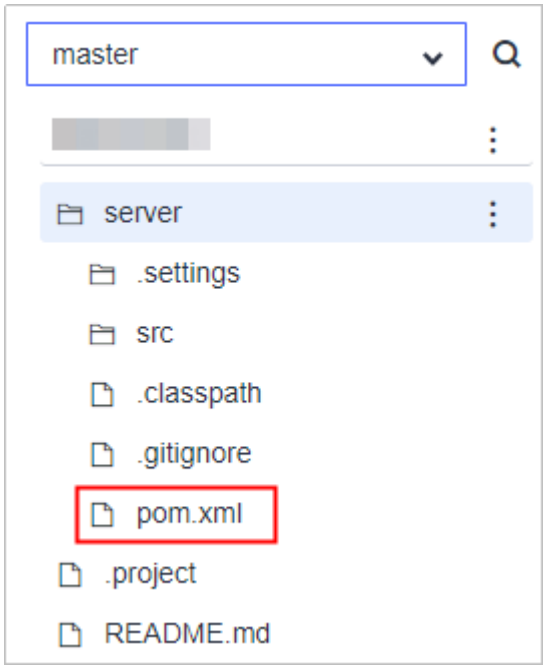
异常信息如下：

```
[ERROR] The goal you specified requires a project to execute but there is no POM in this directory (/xxx/slavespace/slave3/workspace/job_4a1d5be4-b273-4ac8-8d5d-2ee583e71832_1544498089095). Please verify you invoked Maven from the correct directory.
```

## 原因分析

异常信息显示找不到POM文件。系统默认的编译命令是找源码根目录下的POM文件，这个错误就是说源码根目录下不存在POM文件。

例如：下图中源码根目录下显然不存在POM文件的，进入server目录下才发现POM文件。




## 处理方法

这种情况下，需要修改系统默认构建命令。以上面的源码结构为例，解决办法两种，两者选其一即可：

- 先执行**cd server**进入server目录，然后执行**mvn**编译命令。
- 在maven编译命令后增加**-f ./server/pom.xml**，指定pom文件的路径。


# 2.5 执行 Maven 构建时，提示找不到 package/symbol

## 问题现象

执行Maven构建任务时，日志报异常信息提示找不到package或symbol，例如：

```
com/xxx/xxx/configserver/encryptor/xxx.java:[11,40] package com.sun.jersey.api.client.config does not exist
```

## 原因分析

分析日志可知，项目中引用了“com.sun.jersey.api.client.config”包下面的内容，但构建时无法从项目中以及所有解析出的依赖包中找到此包导致。导致此结果的原因一般有两大类：

- 代码问题：代码中包引用不正确，此类问题较易排查，如有遇到可优先排查代码。
- 环境/组件问题：依赖包损坏或不一致，此类问题常表现为本地可编译而云端构建失败；此章节主要为此类问题提供一些可能的解决方案。其中可能的环境/组件问题有：
  - **依赖包冲突**
  - **依赖范围错误**
  - **使用GAV模式上传依赖包**
  - **依赖包损坏**
  - 其他

## 依赖包冲突

部分场景下，因为操作失误或一些第三方依赖被动引入，项目中可能同时存在同一依赖的多个版本。同时引入不同版本可能会导致实际使用的版本与预期不符，进而导致找不到指定的包。处理此类问题操作步骤：

**步骤1** 确认使用的依赖包版本，有两种方式可参考。

1. 参考Maven定义的两个依赖调解原则：

- 第一原则：路径最近者优先， 如：A->B->C->X(1.0)、A->D->X(2.0)，则最终引用X版本为2.0。
- 第二原则：第一声明者优先， 两个引入路径长度相同时，先引入的版本为最终版本。

2. 使用dependency插件，在构建任务执行**mvn dependency:tree**查看。

**步骤2** 如果确定构建使用的依赖与预期不符，导致构建失败，在pom最外层引入需要的依赖并重试：

```
<dependencies>
   <dependency>
      <groupId></groupId>
      <artifactId></artifactId>
      <version></version>
   </dependency>
</dependencies>
```

**----结束**

## 依赖范围错误

使用Maven管理依赖时，Maven坐标中scope属性定义了依赖的有效范围。错误地指定依赖范围会导致该依赖在compile时无效，如果此时在项目主代码中使用了此依赖中的包，则会导致编译错误。处理步骤：

**步骤1** 使用**mvn dependency:tree**查看项目使用的依赖及依赖范围。

**步骤2** 对比依赖范围以及项目中使用依赖的位置。

若在主代码中使用了依赖中的包，且要求依赖在编译时有效，则依赖的范围需要为以下之一：

- compile
- provided
- system：系统依赖范围必须通过systemPath指定依赖文件位置，且依赖文件必须存在于指定目录。

**----结束**

## 使用 GAV 模式上传依赖包

- 在私有依赖仓库上传自研依赖包A时，如果选择GAV模式，只需要上传jar文件，系统会自动生成对应的pom文件；但是，此pom文件中只包含此依赖本身的坐标定义，原来的**<dependencies>**节点则会全部丢失。
- 假设当前构建项目D，使用了项目A构建的依赖A，依赖A引入了第三方依赖B（D > A > B），此时，在构建D项目时，因为以上原因，Maven解析依赖A时，无法识别引入的依赖B，最终导致项目D中找不到依赖B的内容，遇到此场景时，可尝试按以下步骤排查：

**步骤1** 查看项目D的依赖树，确定缺失的内容是否由项目A的pom文件引入，如果是则进入下一步，否则请尝试其他解决方案。

**步骤2** 从私有依赖仓库下载依赖A的pom文件，与项目A中pom对比，如果线上pom缺失了B依赖的引入，进入下一步，否则请尝试其他解决方案。

**步骤3** 更新依赖A的版本号并重新上传，此处提供两种解决方案：

- 使用编译构建服务构建项目A，使用deploy命令将依赖A上传到Maven私有仓库（推荐：可集成于流水线中实现自动化）。
- 在Maven私有仓库重新上传依赖A，此时选择POM模式，分别上传jar文件和pom文件。

**步骤4** 更新完成后，尝试重新构建项目D。

**----结束**

### 依赖包损坏

依赖包损坏可能会导致依赖包中某些文件缺失，此时构建可以找到相应依赖包，但无法找到其中的class文件或者package，导致此类问题。此场景下可按包类型分不同方式处理：

- 第三方依赖包：直接联系技术支持处理。
- 自研（手动上传到Maven私有仓库）的依赖包，按如下步骤排查：
  a. 从Maven私有依赖仓库下载依赖包。
  b. 解压缩并查看依赖包内容是否正常。
  c. 若依赖包内容异常，再分两种情况排查：
     - 如果是第三方提供的包手动上传到maven私有依赖仓库，确认包文件无误并尝试重新上传（注意同时上传pom与jar文件）。
     - 如果是自己构建（本地/云端构建）的依赖包，且已确认代码无误，则检查是否是**多任务同时构建导致构建生成jar包内容缺失**。

## 2.6 使用 exec-maven-plugin 插件实现 Maven 和 npm 混合编译

### 问题现象

Maven项目里包含前端代码，需要npm构建，而系统提供的Maven镜像不包含npm构建环境。

### 处理办法

Maven插件exec-maven-plugin实现混合编译，首先配置插件，其次配置npm环境，最后执行构建。

**步骤1** pom文件配置。

每条npm命令都是<executions>标签中的一个<execution>，不建议配置代理和私有的npm镜像仓，而是使用华为开源镜像站，其配置如下：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <executions>
        <execution>
          <id>exec-npm-config</id>
          <phase>prepare-package</phase>
```

```
                <goals>
                    <goal>exec</goal>
                </goals>
                <configuration>
                    <executable>npm</executable>
                    <arguments>
                        <argument>config</argument>
                        <argument>set</argument>
                        <argument>registry</argument>
                        <argument>https://mirrors.xxcloud.com/repository/npm/</argument>
                    </arguments>
                    <!-- <workingDirectory>${basedir}</workingDirectory>-->
                </configuration>
            </execution>

            <execution>
                <id>exec-npm-config-4</id>
                <phase>prepare-package</phase>
                <goals>
                    <goal>exec</goal>
                </goals>
                <configuration>
                    <executable>npm</executable>
                    <arguments>
                        <argument>config</argument>
                        <argument>set</argument>
                        <argument>sass_binary_site</argument>
                        <argument>https://repo.huaweicloud.com/node-sass</argument>
                    </arguments>
                    <!-- <workingDirectory>${basedir}</workingDirectory>-->
                </configuration>
            </execution>

            <execution>
                <id>exec-npm-install</id>
                <phase>prepare-package</phase>
                <goals>
                    <goal>exec</goal>
                </goals>
                <configuration>
                    <executable>npm</executable>
                    <arguments>
                        <argument>install</argument>
                    </arguments>
                    <workingDirectory>${basedir}</workingDirectory>
                </configuration>
            </execution>

            <execution>
                <id>exec-npm-run-build</id>
                <phase>prepare-package</phase>
                <goals>
                    <goal>exec</goal>
                </goals>
                <configuration>
                    <executable>npm</executable>
                    <arguments>
                        <argument>run</argument>
                        <argument>build</argument>
                    </arguments>
                    <workingDirectory>${basedir}</workingDirectory>
                </configuration>
            </execution>
        </executions>
    </plugin>
  </plugins>
</build>
```

**步骤2** 新建Maven构建任务。

**步骤3** 在新建的Maven构建任务里增加以下npm的安装及环境配置命令：

# 创建文件目录

```
mkdir ./node
```

# 使用curl命令下载Node.js软件包

```
curl -kv https://mirrors.xxxcloud.com/nodejs/v10.15.3/node-v10.15.3-linux-x64.tar.gz  -o ./node/node-v10.15.3-linux-x64.tar.gz
```

# 使用tar命令解压

```
tar -zxvf ./node/node-v10.15.3-linux-x64.tar.gz -C ./node
```

# 配置环境变量

```
export NODEJS_HOME="${WORKSPACE}/node/node-v10.15.3-linux-x64"
export PATH="${NODEJS_HOME}/bin:${PATH}"
```


Node.js包（如本例中的node-v10.15.3）的下载地址请访问**华为开源镜像站**查找并复制链接地址。

**步骤4** 保存后，执行构建验证。

**----结束**

# 2.7 执行 Maven 构建时，多个子项目和父项目之间引用报错

## 问题现象

Maven构建任务，pom文件存在多个子项目和父项目之间的引用，在执行任务时，日志报如下异常信息：

```
[ERROR] Project 'xxx.xxx:xxx1:1.0-SNAPSHOT' is duplicated in the reactor @
[2022-03-02 14:02:52.656] [ERROR] Project 'xxx.xxx:xxx2:1.0-SNAPSHOT' is duplicated in the reactor -> [Help 1]
[2022-03-02 14:02:52.656] [ERROR]
[2022-03-02 14:02:52.656] [ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[2022-03-02 14:02:52.656] [ERROR] Re-run Maven using the -X switch to enable full debug logging.
```

## 原因分析

在Maven中，parent模块组织好childA和childB，叫做"聚合"。多个模块联合编译实现起来很简单，按照以下方式即可：

1. 在parent的pom文件里加入以下内容：

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.demo</groupId>
<artifactId>parent</artifactId>
<version>1.0</version>
<modules>
```

```
  <module>childA</module>
  <module>childB</module>
</modules>
```

2. 在childA和childB的pom文件中添加相应的标签来标记父模块。
   - childA：
     ```
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.demo</groupId>
     <artifactId>childA</artifactId>
     <version>1.0</version>
     <parent>
       <groupId>com.demo</groupId>
       <artifactId>parent</artifactId>
       <version>1.0</version>
     </parent>
     ```
   - childB：
     ```
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.demo</groupId>
     <artifactId>childB</artifactId>
     <version>1.0</version>
     <parent>
       <groupId>com.demo</groupId>
       <artifactId>parent</artifactId>
       <version>1.0</version>
     </parent>
     ```

在上述的配置形式中指定了一个父项目，下面有两个同级的子项目A和B，如果A项目的pom文件中把B项目当做自己的子项目来引用或者把parent项目作为子项目就会引起冲突，构建时就是出现上面的报错。

## 处理办法

检查项目的pom的引用情况 ，如果要B项目作为A的子项目，则需要从parent的pom中把B项目的引用去掉，把B项目的父标签指向A项目，如下：

- parent项目：
  ```
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.demo</groupId>
  <artifactId>parent</artifactId>
  <version>1.0</version>
  <modules>
    <module>childA</module>
  </modules>
  ```
- A项目：
  ```
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.demo</groupId>
  <artifactId>childA</artifactId>
  <version>1.0</version>
  <parent>
    <groupId>com.demo</groupId>
    <artifactId>parent</artifactId>
    <version>1.0</version>
  </parent>
  <modules>
    <module>childB</module>
  </modules>
  ```
- B项目：
  ```
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.demo</groupId>
  <artifactId>childA</artifactId>
  <version>1.0</version>
  <parent>
    <groupId>com.demo</groupId>
    <artifactId>childA</artifactId>
  ```

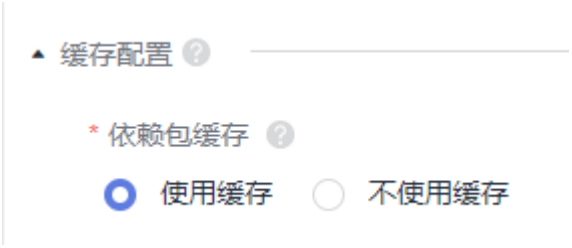
```
  <version>1.0</version>
</parent>
```

# 2.8 如何配置及清理 Maven 构建缓存

编译构建提供了构建缓存功能，构建时可将依赖缓存于用户私有存储空间，下次构建时直接使用，无需重复下载，可极大提高构建效率。

## 构建缓存配置

新建编译构建任务时，默认选择使用缓存加速构建，用户可以在配置“Maven构建”步骤时选择是否使用缓存。




## 缓存清理步骤

由于网络抖动、并发构建或其他极端情况，可能出现缓存内容异常导致构建异常，本章节介绍异常缓存的清理过程。

**须知**

执行缓存清理操作前，请务必仔细阅读以下缓存清理风险以及注意事项：

- 由于缓存目录为同租户共享，频繁清理缓存会概率性导致同租户用户构建异常（常表现为“xxx文件不存在”），故只可在缓存异常时清理，任务执行成功后务必再次编辑任务，删除清理命令，并且在执行清理缓存操作的同时，不要执行其他的使用缓存的编译构建任务。
- 清理缓存时需要使用精确的文件路径，如：清理XXX厂商demo 1.0.0版本，请使用命令rm -rf /path/com/*xxx*/demo/1.0.0。尽量避免删除目录层级过高，导致下次构建缓慢或因网络问题导致依赖异常。
- 出于安全考虑，缓存清理命令只可在“Maven构建”步骤执行，在其他步骤执行此命令会导致“目录不存在”或”清理无效”等报错。

**步骤1** 单击构建任务列表操作列，进入“编译构建编辑”页面。

**步骤2** 选择“构建步骤 > Maven构建”，找到命令行**mvn xxxx**。

**步骤3** 在命令行“mvn xxx”前输入缓存清理命令，单击“保存”。

缓存清理命令格式为：rm -rf /repository/local/maven/{groupId}/{artifactId}/{version}，需填入的参数分别对应依赖包坐标中的groupId、artifactId、version，其中，groupId中的点会被自动分割为层级目录。

若依赖包如下：

```
<dependency>
<groupId>com.xxx.xxx</groupId>
```

```
<artifactId>demo</artifactId>
<version>1.0.9-SNAPSHOT</version>
</dependency>
```

则清理该依赖包所需命令为：rm -rf /repository/local/maven/com/xxx/xxx/demo/1.0.9-SNAPSHOT。


**步骤4** 重新执行构建任务，执行成功后按照上面步骤再次编辑任务，移除清理缓存命令。

**----结束**

# 2.9 如何查找 Maven 构建中正确的构建包路径

**步骤1** 新建Maven构建任务，在“Maven构建”构建步骤后增加“上传软件包到软件发布库”构建步骤。

**步骤2** 配置构建包路径，填写任意路径并保存。

**步骤3** 执行构建任务，在日志中找到BUILD SUCCESS信息。

往上几行找到形如“/target/****.war”的信息，即为准确的构建包路径。

```
[2020/05/07 16:17:57.927] snapshots: [enabled => true, update => always]
[2020/05/07 16:17:57.927]  releases: [enabled => false, update => daily]
[2020/05/07 16:17:57.927] ,       id: central
[2020/05/07 16:17:57.927]       url: https://repo.maven.apache.org/maven2
[2020/05/07 16:17:57.927]    layout: default
[2020/05/07 16:17:57.927] snapshots: [enabled => false, update => daily]
[2020/05/07 16:17:57.927]  releases: [enabled => true, update => never]
[2020/05/07 16:17:57.927] ]
[2020/05/07 16:17:57.927] [INFO] No tests to run.
[2020/05/07 16:17:57.927] [INFO]
[2020/05/07 16:17:57.927] [INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ javaMavenDemo ---
[2020/05/07 16:17:59.048] [INFO] Building jar: ***/target/javaMavenDemo-1.0.jar
[2020/05/07 16:17:59.048] [INFO] ------------------------------------------------------------------------
[2020/05/07 16:17:59.048] [INFO] BUILD SUCCESS
[2020/05/07 16:17:59.048] [INFO] ------------------------------------------------------------------------
```

**----结束**

# 2.10 如何使用 jib-maven-plugin 插件构建 Maven 工程制作镜像

## 问题现象

由于CodeArts官方提供的maven镜像中没有docker环境，所以，使用docker-maven-plugin插件构建的项目通过CodeArts的编译构建制作镜像时会报错，比如：

```
INFO: I/O exception (java.io.IOException) caught when processing request to {}->unix://localhost:80: No such file or directory
```

本文将指导用户在没有docker的maven环境使用jib-maven-plugin插件制作出带有镜像的tar文件。

## 处理方法

1. 修改需要制作镜像的项目代码。

   找到pom文件并引入插件，内容如下：

```
<!--使用jib插件-->
        <plugin>
          <groupId>com.google.cloud.tools</groupId>
          <artifactId>jib-maven-plugin</artifactId>
          <version>1.3.0</version>
          <configuration>
            <!--from节点用来设置镜像的基础镜像，相当于Docerkfile中的FROM关键字-->
            <from>
              <!—建议使用swr公开镜像，下载速度快，更稳定 -->
              <image>swr.example.myxxcloud.com/xxxx/xxx:xxxxx</image>
            </from>
            <to>
              <!--镜像名称和tag，使用了mvn内置变量${project.version}，表示当前工程的version-->
              <image> hellojib:${project.version}</image>
            </to>
            <!--容器相关的属性-->
            <container>
              <!--jvm内存参数-->
              <jvmFlags>
```

```
                    <jvmFlag>-Xms4g</jvmFlag>
                    <jvmFlag>-Xmx4g</jvmFlag>
                </jvmFlags>
                <!--要暴露的端口-->
                <ports>
                    <port>8080</port>
                </ports>
            </container>
        </configuration>
    </plugin>
```

- From标签：设置基础镜像，相当于dockerfile中的FROM关键字，这里推荐使用SWR中的镜像，构建时下载速度快并且稳定。
- To标签：设置制作出来的镜像的镜像名称和tag。
- Container标签：设置容器的相关属性，jvm内存参数，端口等。
- mainClass标签：设置项目启动的主程序，也就是Spring Boot的Application类。

2. 创建构建任务并执行。

   a. 添加两个构建步骤：Maven构建和上传软件包到软件发布库，并将Maven构建默认命令修改为：

   ```
   mvn compile jib:buildTar -Dmaven.test.skip=true -U -e -X –B
   ```

   **说明**

   jib构建工具主要包含了四个强大的功能，由于编译构建是在没有docker环境的情况下构建，所以使用build命令和dockerBuild命令并不能制作出镜像，只能使用buildTar命令制作出一个包含镜像的tar文件。

   - build提供了创建镜像并推送到远程仓库功能。
   - buildTar提供创建一个包含镜像的tar文件功能。
   - dockerBuild提供创建docker镜像到本地功能。
   - exportDockerContext提供创建dockerfile功能。

   构建成功后，日志显示如下信息：

   ```
   35  [2021/01/04 15:51:33.154] [INFO]
   36  [2021/01/04 15:51:33.154] [INFO] --- jib-maven-plugin:1.3.0:buildTar (default-cli) @ javaMavenDemo ---
   37  [2021/01/04 15:51:36.709] [INFO]
   38  [2021/01/04 15:51:36.709] [INFO] Containerizing application to file at '***//target/jib-image.tar'...
   39  [2021/01/04 15:51:36.709] [INFO] Getting base image**********
   40  [2021/01/04 15:51:36.709] [INFO] Building classes layer...
   41  [2021/01/04 15:51:37.528] [INFO] The base image requires auth. Trying again for**********
   42  [2021/01/04 15:51:37.529] [INFO] Retrieving registry credentials for swr.cn-north-5.myhuaweicloud.com...
   43  [2021/01/04 15:51:55.378] [INFO]
   44  [2021/01/04 15:51:55.378] [INFO] Container entrypoint set to [java, -Xms4g, -Xmx4g, -cp, /app/resources:/app/classes:/app/libs/*, HelloWorld]
   45  [2021/01/04 15:51:55.378] [INFO] Building image to tar file...
   46  [2021/01/04 15:51:58.647] [INFO]
   47  [2021/01/04 15:51:58.647] [INFO] Built image tarball at ***//target/jib-image.tar
   48  [2021/01/04 15:51:58.647] [INFO]
   49  [2021/01/04 15:51:58.647] [INFO] ------------------------------------------------------------
   50  [2021/01/04 15:51:58.647] [INFO] BUILD SUCCESS
   51  [2021/01/04 15:51:58.647] [INFO] ------------------------------------------------------------
   52  [2021/01/04 15:51:58.648] [INFO] Total time: 30.526 s
   53  [2021/01/04 15:51:58.648] [INFO] Finished at: 2021-01-04T07:51:57Z
   ```

   b. 在java工程的target目录下，可以看到生成了名为jib-image.tar的文件，同时任务会通过上传软件到发布库步骤上传到发布库。

3. 使用tar镜像。

   通过执行脚本或下载命令从发布库中将tar文件下载到要部署应用的服务器上，执行docker load命令将tar文件的镜像加载到本地镜像仓库，再使用docker run等命令启动镜像即可。

# 2.11 使用 Maven 构建时，代码更新后构建出来的包还是旧的

## 问题现象

本地提交了代码到远程仓库，并且确认远程仓库代码已经更新，但是构建后打出来的包，解压并反编译后发现还是旧的代码。

## 原因分析

这种问题一般是用户不小心将本地编译后的文件（“target”目录文件）上传到远程仓库，同时打包前没有执行clean操作导致。

## 处理方法

- 方法一：删除远程仓库的“target”目录。
- 方法二：打包命令增加“clean”参数，如：原先打包命令为：mvn package -Dmaven.test.skip=true -U -e -X -B，增加“clean”参数后如下：

```
mvn clean package -Dmaven.test.skip=true -U -e -X -B
```

# 2.12 使用 Maven 构建时，Maven 组件下载缓慢

## 问题现象

使用Maven构建时，Maven组件下载缓慢。

## 原因分析

Maven构建步骤默认生成的settings配置文件中，Maven镜像仓库地址为公网地址，导致部分Maven组件依赖下载时会优先从公网下载，公网访问超时，然后再从私有库下载，从而导致Maven组件下载缓慢。

## 处理方法

**方法一：（推荐）**

1. **访问服务首页**。
2. 参考**自定义settings.xml文件**，将修改后的settings.xml文件上传至代码仓根目录。
3. 在构建任务列表页，单击对应的构建任务名称。
4. 单击“编辑”按钮，在“Maven构建”步骤的命令窗口中，**--settings settings.xml**，即可使用已添加的“settings.xml”文件执行Maven构建。

```
# 使用场景： 打包项目且不需要执行单元测试时使用
mvn package -Dmaven.test.skip=true -U -e -X -B --settings settings.xml
```

**方法二：**

1. **访问服务首页**。
2. 在构建任务列表页，单击对应的构建任务名称。
3. 单击“编辑”按钮，在“Maven构建”步骤前，添加“下载文件管理的文件”步骤。
4. 单击“上传”，上传自定义的“settings.xml”文件，其他参数保持默认即可。自定义settings.xml文件的方法可参考**自定义settings.xml文件**。
5. 在“Maven构建”的命令窗口中，**--settings settings.xml**，即可使用已添加的“settings.xml”文件执行Maven构建。

```
# 使用场景： 打包项目且不需要执行单元测试时使用
mvn package -Dmaven.test.skip=true -U -e -X -B --settings settings.xml
```

## 自定义 settings.xml 文件

**步骤1** 在“Maven构建”的命令窗口执行**cat /home/build/.m2/settings.xml**命令，任务执行完成后，会在构建日志中展示settings.xml文件的内容。

**步骤2** 参考构建日志中的settings.xml的信息自定义新的settings.xml文件。

**----结束**

# 3 Android 构建

## 3.1 使用 Android 构建时，项目配置的 Jcenter()不稳定

### 问题现象

执行过构建任务日志报错信息如下：

```
Caused by: org.gradle.internal.resource.transport.http.HttpErrorStatusCodeException: Could not GET 'https://jcenter.bintray.com/org/apache/commons/commons-compress/1.8.1/commons-compress-1.8.1.pom'. Received status code 504 from server: Gateway Time-out
```

### 原因分析

- 网络异常无法连接依赖镜像仓。
- 依赖镜像仓异常。

### 处理方法

建议配置开源镜像站，稳定、快速，配置方法如下：

进入构建任务依赖的代码仓库，在“build.gradle”文件中添加如下代码，即可配置开源镜像仓。

```
allprojects {
repositories {
maven {
url 'https://repo.xxcloud.com/repository/maven/'
}
jcenter()
}
}
```

# 3.2 执行 Android 构建时，lint 检查出错终止任务执行

## 问题现象


## 处理方法

可以在命令行中的gradle命令后加上-xlint参数，跳过lint检查。如：

```
/bin/bash ./gradlew assembleDebug -Dorg.gradle.daemon=false -d --stacktrace -xlint
```

或

```
gradle assembleDebug -Dorg.gradle.daemon=false -d --stacktrace --init-script /root/.gradle/init.gradle -xlint
```

# 3.3 执行 Android 构建时，无法下载 com.android.tools.build:gradle:3.0.1 依赖

## 问题现象

错误信息如下：

```
Could not find com.android.tools.build:gradle:3.0.1
```

## 处理方法

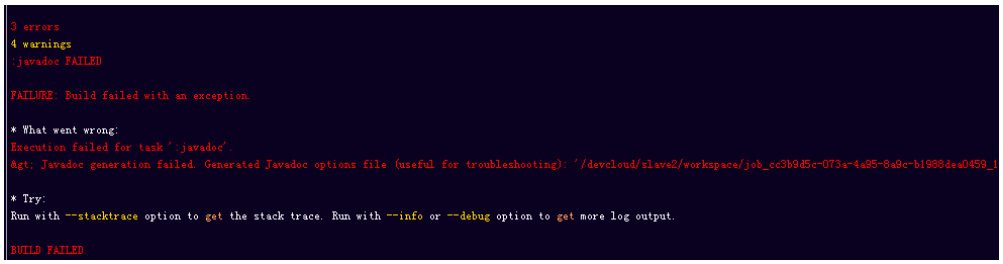
根据日志提示，对“app”目录下的“build.gradle”文件添加google()仓库，进行如下修正：

```
allprojects {
repositories {
google()
jcenter()
}
}
```

# 3.4 执行 Android 构建时，报错提示 Javadoc generation failed

## 问题现象

Gradle在构建过程中执行了**javadoc**检查，可能会报“Javadoc generation failed”的错误：




## 处理方法

避免**javadoc**的检查，在项目根目录下的Gradle下面就要添加如下配置：

```
allprojects {
repositories {
jcenter()
}
tasks.withType(Javadoc) {
options.addStringOption('Xdoclint:none', '-quiet')
options.addStringOption('encoding', 'UTF-8')
}
}
```

# 3.5 执行 Android 构建时，报错提示 Could not find method google()

## 问题现象

Gradle插件升级到Gradle Plugin Build Tool 3.0版本后，对应的Gradle需要升级到4.1版本。如果编译插件找不到对应的Gradle 4.1版本，就会报如下错误：

```
Could not find method google() for arguments [] on repository container
```

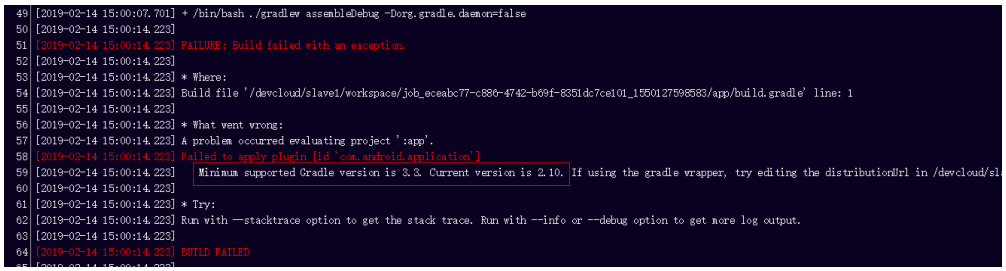
## 处理方法

构建工具Gradle选择4.1以上版本即可。

# 3.6 执行 Android 构建时，报错提示 Gradle 版本过低

## 问题现象

执行Android构建后出现如下所示提示，需要Gradle最低版本是3.3，当前是2.10。




## 原因分析

编译环境的Gradle版本较低不满足编译要求。

## 处理方法

- 如果是Gradle构建，则选择符合条件的Gradle版本。
- 如果是Gradlew构建，则修改“gradle/wrapper/gradle-wrapper.properties”文件，修改“gradle-*.*-all.zip”的版本。

gradle-wrapper.properties 大小：231B

```
#Mon Dec 28 10:00:20 PST 2015
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-2.10-all.zip

```

# 3.7 执行 Android 构建时，Android APK 签名失败

## 问题现象

Android构建时报签名错误：

错误信息类似于 “Execution failed for task ':app:validateSigningDebug” 或者 “Execution failed for task ':app:validateSigningRelease” 。

## 处理方法

在Android构建过程中推荐使用“Android APK签名”构建步骤完成APK签名，编译构建提供了Android APK签名构建步骤，配置方法如下：

1. 在“Android构建”步骤后添加“Android APK签名”步骤。


参数说明如下：

| 参数 | 说明 |
| --- | --- |
| 需要签名的APK路径 | Android构建后生成要签名的.apk文件位置，支持正则表达式，如：可以使用build/bin/*.apk匹配构建出来的APK包。 |
| Keystore文件 | 用于签名的Keystore文件，单击下拉列表，展示**文件管理**已经上传的Keystore文件，请根据需要选择。 |
| keystore password | 密钥文件密码。 |
| 别名（Alias） | 密钥别名。 |
| key password | 密钥密码。 |
| apksigner命令行 | 用户自定义签名参数，默认“--verbose”显示签名详细。 |

2. 验证签名是否成功。

   配置完成后执行构建任务，当显示任务执行成功后，查看构建日志，若“Android APK签名”那段日志中显示“result: Signed”即为签名成功。

# 4 Gradle 构建

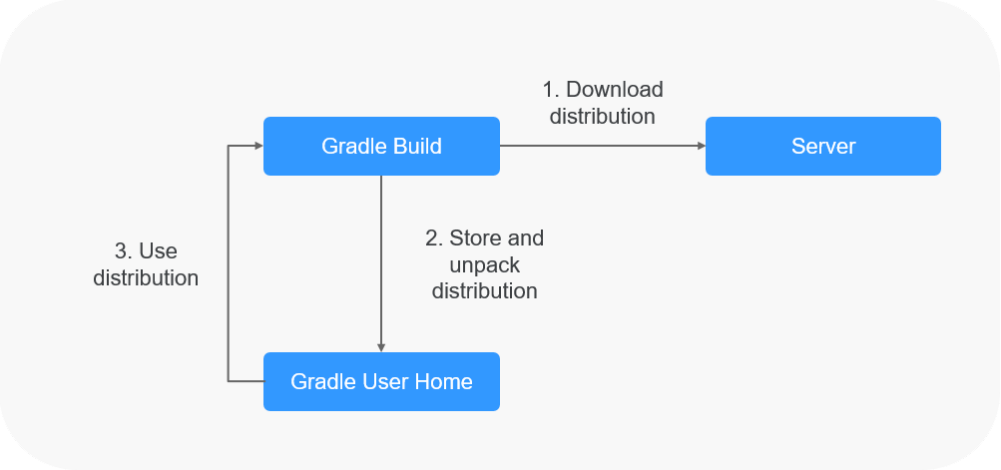
## 4.1 找不到指定版本的 Gradle 工具

### 问题现象

编辑Action时找不到想要的Gradle工具版本。

### 原因分析

- 如果您需要编译的工程依赖的Gradle版本不在列表中，可以使用gradlew(gradle wrapper)封装Gradle命令。
- Gradlew封装了Gradle命令，将首先安装指定版本的Gradle，再执行构建命令。
- Gradle官方建议所有Gradle项目中都创建Wrapper文件，方便没有安装Gradle的用户使用。




### Gradle Wrapper 使用教程

**步骤1** 在本地环境中，进入代码根目录，执行gradle wrapper命令。命令执行完毕后，可以发现代码仓库中新增了以下文件：

- gradlew (Unix Shell 脚本)

- gradlew.bat (Windows批处理文件)
- gradle/wrapper/gradle-wrapper.jar (Wrapper JAR文件)
- gradle/wrapper/gradle-wrapper.properties (Wrapper属性文件)

**步骤2** 提交代码到代码仓库。

**步骤3** 修改构建任务中命令行里的语句，将gradle替换成./gradlew，如将gradle build替换为./gradlew build。

**----结束**

# 5 Msbuild 构建

## 5.1 执行 Msbuild 构建时，找不到程序集（**.dll）

### 问题现象

构建告警：

- warning MSB3245: 未能解析此引用。
- 未能找到程序集“Microsoft.Office.Interop.Word, Version=15.0.0.0, Culture=neutral, PublicKeyToken=71e9bce111e9429c, processorArchitecture=MSIL”。请检查磁盘上是否存在该程序集。
- 如果您的代码需要此引用，则可能出现编译错误。

### 原因分析

- 由提示信息可知，解决方案中引入了程序集“Microsoft.Office.Interop.Word”，但构建时环境中不具备此程序集，导致编译告警，如果项目代码中使用了此引用，甚至会直接导致失败。
- 通常情况下，此类程序集默认安装于本地系统，无需指定程序集位置，VS构建时会从默认配置的几个程序集路径查找，可以构建成功；而云端构建环境对应目录无此程序集，进而导致本地与云端构建不一致。
- 为解决此类场景，Msbuild集成了NuGet，可以在构建时从远程仓库下载对应程序集，此时只需于项目中指定“packages.config”，并于其中声明依赖的程序集即可。
- 特殊情况下，项目引用的程序集可能无法在远程仓库找到，此时需要手工保存程序集至代码仓库中，并显示指定程序集路径。

### 处理方法 1：使用 NuGet 管理依赖（VS 版）

如果在本地使用VS构建，遇到此问题时，可以直接使用“VS > 管理NuGet程序包”功能，操作步骤如下：

**步骤1** 使用VS打开项目，选中“解决方案>管理NuGet程序包”。

**步骤2** 在“浏览”页签中搜索需要的程序集。

**步骤3** 选择需要的程序集，单击“安装”，并在弹出的提示框单击“确认”。

**步骤4** 提交更改后的代码，再次构建即可解决此问题。

**----结束**

### 处理方法 2：使用 NuGet 管理依赖（手工修改）

如果本地没有VS集成工具，可手工修改解决方案文件实现用NuGet管理依赖，操作步骤如下：

**步骤1** 修改“csproj”文件中的依赖项，增加<HintPath>项。

**步骤2** 在csproj文件同级目录新增packages.config文件，如已存在新增依赖信息即可。

**步骤3** 在csproj文件中引入packages.config文件，如已有配置跳过即可。

**步骤4** 提交修改后的代码，重新构建即可解决此问题。

**----结束**

### 处理方法 3：从项目中引入程序集

部分场景下，需要从引入项目中的程序集，操作步骤如下：

**步骤1** 拷贝已有程序集到项目下，一般在根目录建立packages文件夹存放程序集。

**步骤2** 在csproj文件中引入程序集，并指定程序集地址。

**步骤3** 提交更改后的代码、程序集，重新构建即可解决问题。

**----结束**

## 5.2 执行 Msbuild 构建时，提示 Object、namespace 未定义

### 问题现象

构建：“**object、**namespace未定义”。

### 原因分析

由于解决方案中存在多个csproj文件，生成时使用默认构建命令指定OutputPath=../buildResult/Release/bin会造成所有的csproj生成时在一个文件夹中，造成obj文件等发生链接冲突。

### 处理方法

在构建命令中删除“OutputPath=../buildResult/Release/bin”选项。

# 5.3 执行 Msbuild 构建时，报错提示当前路径下存在多个解决方案/不存在项目文件

## 问题现象

异常信息如下：

```
This folder contains more than one solution file.
The folder 'XXX' does not contain an msbuild solution or packages.config file to restore.
```

## 原因分析

由于当前路径下存在多个sln文件或者不存在项目定义文件，造成构建失败。

## 处理方法

- 若当前路径存在多个sln文件请指定需要构建的sln文件，如**msbuild demo.sln**。
- 若当前路径不存在sln文件，请用cd命令定位到sln所在路径，如**cd src**。

# 5.4 执行 Msbuild 构建时，项目指定了.NET SDK XXX 版本

## 问题现象

构建失败，异常信息为：项目指定了dotnet sdk XXX，在XXX路径下不能找到对应的SDK工具集，请确定是否安装了对应版本的SDK。

## 原因分析

由于项目指定了特定版本的.NET Core SDK，不使用镜像预装的兼容的.NET 2.1.402版本SDK，造成没有指定版本的SDK，无法进行构建。

## 处理方法

**步骤1** 下载对应版本的SDK（例：2.0.0，其他版本请直接替换命令中的2.0.0）。

```
powershell -Command Invoke-WebRequest -UseBasicParsing https://dotnetcli.blob.core.windows.net/
dotnet/Sdk/2.0.0/dotnet-sdk-2.0.0-win-x64.zip
-OutFile dotnet2.0.0.zip;
```

**步骤2** 解压到当前路径下。

```
powershell -Command Expand-Archive dotnet2.0.0.zip;
```

**步骤3** 复制sdk目录到“${Env:ProgramFiles}\dotnet\sdk”下。

```
powershell -Command Copy-Item -Recurse dotnet2.0.0\sdk\2.0.0 ${Env:ProgramFiles}\dotnet\sdk ;
```

**----结束**

# 5.5 执行 Msbuild 构建时，找不到**文件

## 问题现象

异常信息为“找不到**文件”，但是核实后项目已存在该文件。

## 原因分析

项目中文件（文件夹）名带有空格，造成Msbuild编译时从空格处截断文件路径，报找不到文件异常，导致构建失败。

## 处理方法

去除目录中的空格以及对应引用路径中的空格，可保证编译过程中不出现空格导致的构建失败问题。

# 5.6 执行 Msbuild 构建时，编译过程出现的 file path too long 问题

## 问题现象

成功拉取代码以后，构建过程中， 提示无法拷贝***文件，文件路径长度超过260字符。

## 原因分析

Windows系统中，文件全路径的最大长度限制为260字符，超过此长度会导致Msbuild构建失败，项目中引用了路径过长的文件，导致Msbuild执行**copy**命令时，无法拷贝路径过长的文件，导致构建失败。

## 处理方法

修改文件路径长度至系统要求大小即可。

**说明**

- 项目文件全路径长度实际为项目下文件相对路径长度与编译构建服务默认路径长度之和。
- 编译构建服务默认路径长度为45字符。

  因此，在使用Msbuild构建的过程中，您的项目文件路径需满足：项目下文件相对路径（以代码仓库为根目录）长度不可大于215字符。
- 一些特殊场景（如构建时指定输出目录为“Output/release”)下，可能会额外占用路径长度。

  建议您的项目下文件相对路径（以代码仓库为根目录）长度保持在200个字符以下，原则上尽可能短最好。

# 5.7 执行 Msbuild 构建时，找不到 AxImp.exe

## 问题现象

构建“microsoft.common.currentversion.targets”找不到“AxImp.exe”，需要安装SDK。

## 原因分析

- 系统中安装了4.7.2版本的的SDK，项目中没有特殊指定SDK时，可以兼容4.0以上版本。
- 项目中若特殊指定了某SDK版本，Msbuild构建会去对应版本路径下查找，导致找不到SDK错误。

## 处理方法

- 尽可能避免强制指定SDK版本。
- 如确属业务需要无法更改，请联系客服。

# 6 Npm 构建

## 6.1 执行 Npm 构建时，报错提示 JavaScript heap out of memory

### 问题现象

执行Npm构建任务时，日志报如下异常信息：

```
FATAL ERROR：CALL_AND_RETRY_LAST Allocation failed - JavaScript heap out of memory.
```

### 原因分析

Nodejs运行时使用内存是有大小限制的，64位系统约为1.4GB，32位系统约为0.7GB，该次构建内存使用超出了默认大小。

### 处理方法

**方法一：**升级nodejs版本。

**方法二：**启动Node时设置“--max_old_space_size”或“--max_new_space_size”参数来调整内存大小的使用限制。

```
node --max_old_space_size=1700 test.js // 单位为MB 修改老生代内存限制
node --max_new_space_size=1024 test.js // 单位为KB 修改新生代内存限制
```

针对前端三大框架的解决方法如下：

| 框架类型 | 解决方法 |
| --- | --- |
| Vue | 只需要修改“package.json”文件中“build”属性值，在命令中加入带参数的node命令即可，例如：<br>`"build": "node --max_old_space_size=4096  ./node_modules/vite/bin/vite.js build"`<br>或<br>`"build": "node --max_old_space_size=4096 ./node_modules/@vue/cli-service/bin/vue-cli-service.js build"` |

| 框架类型 | 解决方法 |
| --- | --- |
| React | 举例说明“package.json”里面“scripts”字段的内容如下：<br>"scripts": {<br>"start": "react-scripts start",<br>"build": "react-scripts build",<br>"test": "react-scripts test --env=jsdom",<br>"eject": "react-scripts eject"<br>}<br>运行npm run build的时候跑的实际代码是react-scripts build，项目根目录下“node_modules”文件夹，找到.bin目录并打开它找到“react-scripts”文件，打开这个文件，把--max_old_space_size=4096这行代码写在#!/usr/bin/env node后面：<br>#!/usr/bin/env node --max_old_space_size=4096 |
| Angular | 举例说明“package.json”里面“scripts”字段的内容如下：<br>"scripts": {<br>"ng": "ng",<br>"start": "ng serve",<br>"build": "ng build",<br>"test": "ng test",<br>"lint": "ng lint",<br>"e2e": "ng e2e"<br>}<br>这里的ng命令也和React一样，在项目根目录“node_modules”文件夹下的.bin目录里面存在名为ng的文件，修改该文件的首行：<br>#!/usr/bin/env  node --max_old_space_size=4096 |

# 6.2 执行 Npm 构建时，报错提示 Unexpected end of JSON ...

## 问题现象

执行npm install时，提示异常信息如下：


## 原因分析

解析文件中的json字符串失败，有可能从镜像仓下载的文件不完整。

## 处理方法

**步骤1** 修改Npm镜像仓，在Npm构建步骤里，添加如下命令：

```
npm config set registry https://repo.xxcloud.com/repository/npm/
```

或

```
npm config set registry https://registry.npm.taobao.org
```

**步骤2** 重新执行构建任务。

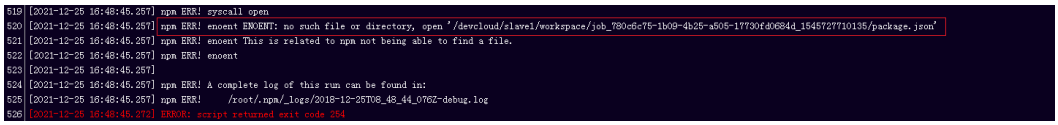
**----结束**

# 6.3 执行 Npm 构建时，报错提示 enoent ENOENT: no such file or directory

## 问题现象

异常信息如下：




## 原因分析

项目缺少关键文件。

上图中520行的错误日志，“npm ERR! enoent ENOENT: no such file or directory, open '/xxx/slave1/workspace/job_780c6c75-1b09-4b25-a505-17730fd0684d_1545727710135/package.json'”，表示缺少“package.json”文件。

## 处理方法

补充错误日志中提示缺失的文件。比如缺少“package.json”文件的情况，需要在代码根目录下增加“package.json”文件。

# 6.4 执行 Npm 构建时，报错提示 Module not found: Error: Can't resolve ...

## 问题现象

执行Npm构建任务时，日志报如下异常信息：


## 原因分析

找不到需要的文件。

上图中6068行的错误日志，“Module not found: Error: Can't resolve './App.Vue' in '/xxx/slave1/workspace/job_d5d70df6-9b64-4faa-ba67-93c06d4a1972_1545727944134/src'”，在“src”文件夹下找不到“./App.Vue”文件。可能原因如下：

- 对应文件夹下，没有所需文件。
- 文件路径大小写配置有误。图中代码配置的是“'./App.Vue'”，实际文件名是“'./App.vue'”，导致找不到所需文件。因为Windows系统不区分大小写，而Linux系统区分，所以可能本地能构建成功，在编译构建服务上却构建失败。

### 处理方法

**步骤1** 在代码项目中的相应文件夹下，补充错误日志中提示缺失的文件。

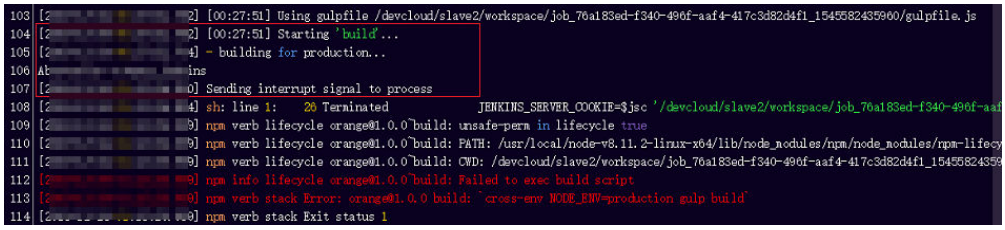
**步骤2** 修改出错的代码中配置的文件路径。

**----结束**

## 6.5 执行 Npm 构建失败，但不显示错误日志

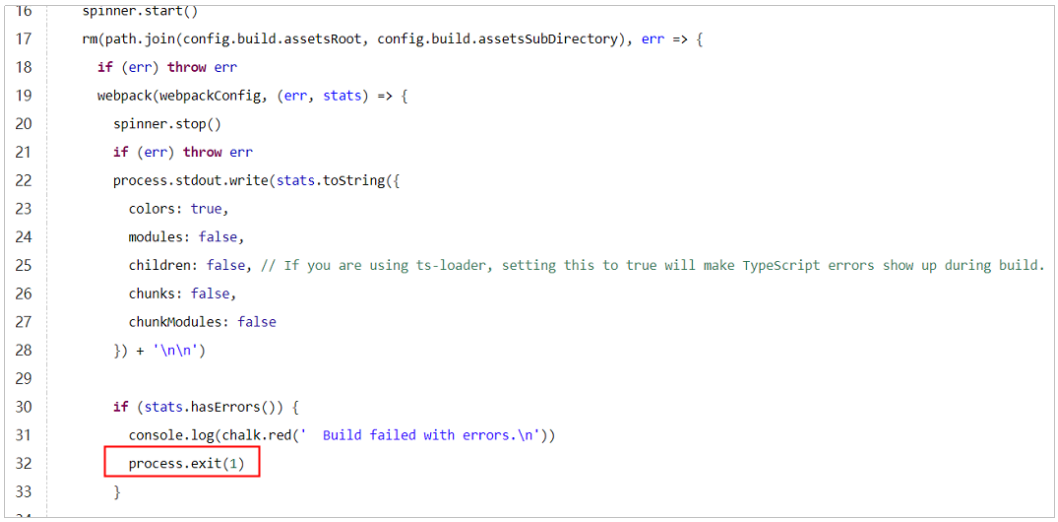
### 问题现象

Npm构建失败，但不显示错误日志，异常信息如下：




### 原因分析

在构建脚本中，设置了出现错误时，直接退出构建。




### 处理方法

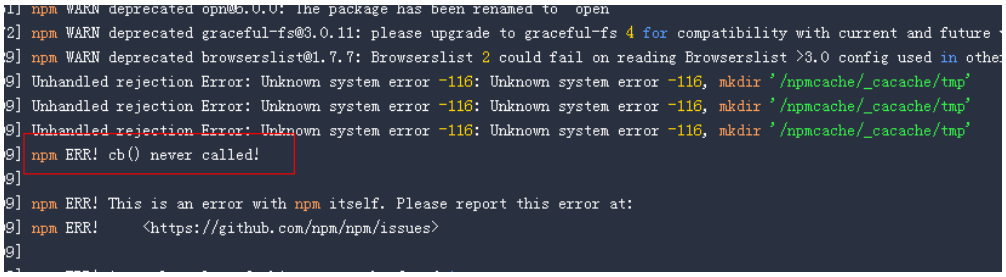
检查构建脚本中对错误情况的处理，删除“process.exit(1)”等可能导致构建出错时直接退出的情况。

## 6.6 执行 Npm 构建时，报错提示 npm cb() never called

### 问题现象

执行Npm构建任务时，日志报如下异常信息：

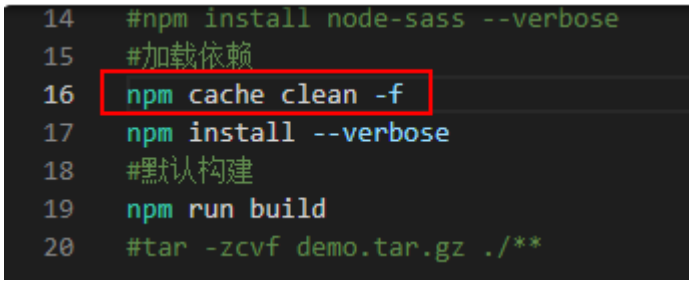



## 原因分析

NPM缓存发生异常，需要清理缓存。

## 处理方法

编辑任务，在命令行**npm install**命令之前添加命令**npm cache clean -f**，然后保存任务重新执行。




# 6.7 执行 Npm 构建时，报错提示 gyp ERR! stack Error: EACCES: permission denied

## 问题现象

执行Npm构建任务时，日志报如下异常信息：

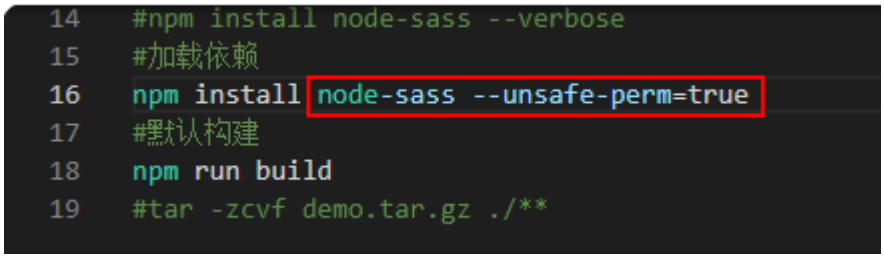
```
gyp ERR! stack Error: EACCES: permission denied, mkdir '******/
job_1451ba57-0c35-4daa-99c2-21425404f61c_1564043318112/saas_shop/node_modules/node-sass/.node-gyp'
```

## 原因分析

当前目录没有读写权限。

## 处理办法

编辑任务，在命令行**npm install**命令之后添加**node-sass --unsafe-perm=true**，保存任务重新执行。




# 6.8 执行 Npm 构建时，报错提示 eslint: error 'CLODOP' is not defined

## 问题现象

执行Npm构建任务时，日志报如下异常信息：

```
Module Error (from ./node_modules/@vue/cli-plugin-eslint/node_modules/eslint-loader/index.js):

***//public/LodopFuncs.js
  79:25  error  'getCLodop' is not defined  no-undef
  80:27  error  Empty block statement       no-empty
  89:21  error  'CLODOP' is not defined     no-undef
```

## 原因分析

如上异常报LodopFuncs.js文件中函数未声明is not defined，可先排查文件；文件正常则可能是不符合eslint规范导致报错。

## 处理方法

1. 检查LodopFuncs.js文件中getCLodop函数是否已定义。
2. 如果文件正常，可以在eslint检查不通过的文件头部添加如下命令行忽略eslint的检查。
   ```
   /* eslint-disable */
   ```

# 6.9 执行 Npm 构建时，报错提示 node-sass 下载失败

## 问题现象

执行Npm构建任务时，日志报如下异常信息：

```
Downloading binary from https://github.com/sass/node-sass/releases/download/v4.14.1/linux-x64-72_binding.node
Cannot download "https://github.com/sass/node-sass/releases/download/v4.14.1/linux-x64-72_binding.node":

read ECONNRESET

...

npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! node-sass@4.14.1 postinstall: `node scripts/build.js`
npm ERR! Exit status 1
npm ERR!
```

```
npm ERR! Failed at the node-sass@4.14.1 postinstall script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.
```

## 原因分析

node-sass的镜像源需要单独设置，如果没有设置，npm默认会去github下载。从软件开发生产线到github的网络不太稳定，容易下载失败。

## 处理方法

在默认命令npm install之前先加上如下命令，选择使用华为云的mirror源，重新执行构建即可。

```
npm config set sass_binary_site https://repo.xxcloud.com/node-sass/
```

# 6.10 执行 Npm 构建时，报错提示 error: could not write config file

## 问题现象

执行Npm构建任务时，日志报异常信息：error: could not write config file / npmcache/_cacache/tmp/git-clone-b0ba91a1/.git/config: Disk quota exceeded

## 原因分析

NPM缓存空间已满，需要清理缓存。

## 处理方法

1. 进入编译构建服务首页。
2. 选择对应的构建任务，单击任务所在行的 ，单击“编辑”。
3. 在“构建步骤”页面编辑“NPM构建”。
4. 在命令行**npm install**命令之前添加命令**npm cache clean -f**，然后保存任务重新执行。


# 6.11 Npm 构建耗时且安装依赖缓慢

## 原因分析

默认的镜像仓地址因网络原因可能导致网络下行效率低。

## 处理方法

**步骤1** 进入编译构建服务首页。

**步骤2** 选择对应的构建任务，单击任务所在行的 *** ，单击“编辑”。

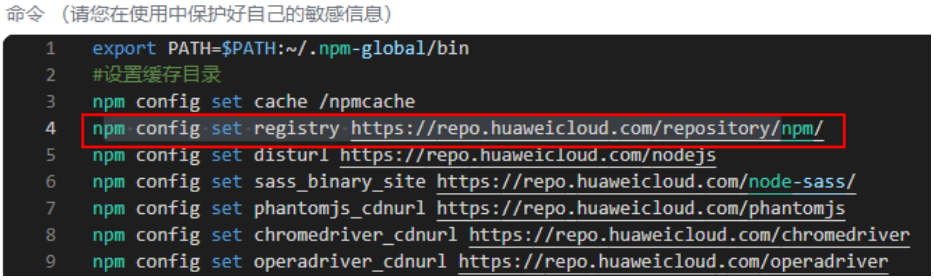
**步骤3** 在“构建步骤”页面编辑“NPM构建”。

**步骤4** 在NPM构建步骤里，添加如下命令，修改Npm镜像仓地址：

```
npm config set registry https://repo.xxcloud.com/repository/npm/
```

或

```
npm config set registry https://registry.npm.taobao.org
```



**步骤5** 单击“保存并执行”，重新执行构建任务。

**----结束**

# 6.12 执行 Npm 构建时，报错提示找不到依赖版本

## 问题现象

npm找不到依赖版本，请确认依赖版本是否存在，提示异常信息如下：


## 原因分析

npm找不到依赖版本。

## 处理方法

1. 检查package.json/package-lock.json中配置的依赖版本是否正确，若不正确，需要修改。
2. 检查所使用的镜像源站上是否存在该版本的依赖。

# 7 镜像问题

## 7.1 使用 Dockerfile 制作镜像失败

使用步骤“制作镜像并推送到SWR”或“执行Docker命令”制作镜像时，docker build阶段可能会制作镜像失败，可参考各场景对应解决方案处理：

- **COPY或者ADD指令找不到文件**
- **制作镜像时拉取基础镜像失败**
- **执行命令失败**
- **拉取DockerHub镜像超时或失败**

### COPY 或者 ADD 指令找不到文件

**问题现象**

构建任务中有“制作镜像并推送SWR”或“执行Docker命令”构建步骤，执行任务时日志报如下异常信息：

```
ADD failed: stat /var/lib/docker/tmp/docker-builder154037010/temp: no such file or directory
[ERROR][制作镜像并推送到SWR仓库]：错误信息：DEV.CB.0210043，制作Docker镜像失败。
COPY failed: stat /var/lib/docker/tmp/docker-builder076130522/test.txt: no such file or directory
```

**原因分析**

ADD指令的源文件为“./temp”，而当前工作目录下没有temp文件。

**处理方法**

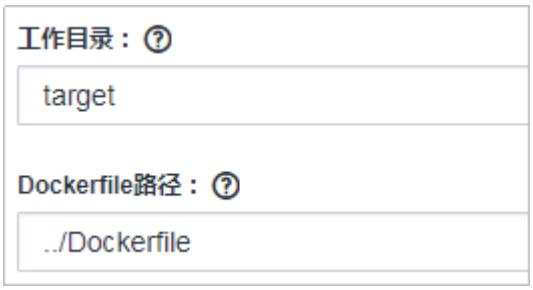
假设当前目录的结构如下：

```
+ target
  - temp
- Dockerfile
```

target目录下有temp文件，而Dockerfile文件和target同级。

- 方法一：将ADD指令的源文件改为“./target/temp”。
- 方法二：target目录作为工作目录，将“制作镜像并推送到SWR仓库”构建步骤的工作目录改为“target”，Dockerfile路径改为“../Dockerfile”。




## 制作镜像时拉取基础镜像失败

使用Dockerfile制作镜像时，如果指定的基础镜像参数有误，会导致镜像拉取失败，主包括以下两个场景：

- 指定的镜像不存在或无权限

  **错误日志**

  ```
  pull access denied for java1, repository does not exist or may require 'docker login'
  ```

  **分析处理**

  镜像仓库中找不到指定的镜像或当前用户对该镜像没有pull权限时，会出现该错误。

  此例的Dockerfile中，*FROM java1:8ull-jdk-alpine*命令指定的镜像“java1”无法在镜像仓库中找到，故出现此错误，请核对并修正镜像名后重试即可。

- 指定的镜像标签不存在

  **错误日志**

  ```
  manifest for java:8ull-jdk-alpine not found
  ```

  **分析处理**

  镜像仓库中存在指定镜像，但不存在镜像的对应版本/标签时会出现“manifest not found”错误。此例的Dockerfile中，*FROM java:8ull-jdk-alpine*命令指定了镜像“java:8ull-jdk-alpine”，镜像仓库中存在“java”镜像，但没有对应的版本/标签“8ull-jdk-alpine”，故出现此错误，请核对并修正镜像版本后重试即可。

## 执行命令失败

**问题现象**

使用Dockerfile制作镜像时，在执行docker build阶段报如下错：

```
exec user process caused "exec format error"
```

**原因分析**

此问题出现的原因一般有两个：

- 制作镜像的基础镜像和执行机不匹配，如：镜像为arm的，但是执行机是x86的。
- Dockerfile文件内容从其他地方复制过来时出现问题。

**处理方法**

1. 先确认镜像和执行机是否匹配，如果镜像是x86的镜像，就只能用x86的执行机。
2. 重新执行构建，查看是否成功，如果不成功，手动输入Dockerfile后再重新执行。

## 拉取 DockerHub 镜像超时或失败

### 问题现象

- 现象一

  错误日志如下：

  ```
  Error response from daemon: Get https://registry.docker-cn.com/v2/: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)
  ```

- 现象二

  错误日志如下：

  ```
  toomanyrequests: You have reached your pull rate limit. You may increase the limit by authenticating and upgrading: https://www.docker.com/increase-rate-limit
  ```

### 原因分析

- 现象一：dockerHub网络不稳定导致拉取镜像超时或失败。
- 现象二：dockerHub的频率限制导致拉取镜像失败。

### 处理方法

- 方法一：将dockerHub的镜像迁移到SWR上，再拉取镜像
  a. 将需要使用的dockerHub镜像下载到本地。
  b. 登录**容器镜像服务**，在总览页面，单击右上角“上传镜像”或快速入门区域的“上传自有镜像”。
  c. 在上传页面选择组织和需要上传的镜像。
  d. 还可以通过客户端上传，选择“我的镜像”，单击右上角的“客户端上传”，根据弹出的页面提示进行操作。
  e. 镜像上传成功后，需要将镜像设置成公开，在“我的镜像”页面，找到刚刚上传的镜像，单击镜像名称，然后单击右上方“编辑”，在编辑页面中将镜像设置成公开。
  f. 最后替换dockerfile中FROM的基础镜像地址并重新执行构建任务，镜像地址格式一般为swr.cn-south-1.myxxcloud.com/{组织名称}/{镜像名称}:{版本名称}，具体内容可以从下载指令中截取。
- 方法二：通过配置SWR镜像加速器地址解决

  **说明**

  该方法只适用于现象一：dockerHub网络不稳定导致拉取镜像超时或失败。

  a. 登录**华为云SWR**。
  b. 选择“镜像资源 > 镜像中心 > 镜像加速器”，复制加速器地址。
  c. 替换dockerfile中FROM的基础镜像地址后重新执行构建任务。如：原先基础镜像地址为*library/node:8.16-slim*，加速器地址为*https://a786190f76fb41679546b24d8d08d8b8.mirror.swr.myxxcloud.com*，则将镜像地址替换为：

  ```
  a786190f76fb41679546b24d8d08d8b8.mirror.swr.myxxcloud.com/library/node:8.16-slim
  ```

# 7.2 推送镜像到 SWR 失败

使用步骤“制作镜像并推送到SWR”或“执行Docker命令”时，因参数错误、环境问题等，可能会出现推送镜像失败，可参考各场景对应解决方案处理。

- **推送镜像提示无权限**（denied: you do not have the permission）
- **推送镜像提示组织数达到上限**（denied: The number of namespaces exceeds the upper limit）
- **推送镜像提示未登录**（denied: You may not login yet）
- **推送镜像提示认证失败**（denied: Authenticate Error）
- **推送镜像提示组织名非法**（invalid reference format）
- **推送镜像提示本地镜像不存在**（An image does not exist locally with the tag: ***）
- **推送镜像提示非法摘要**（digest invalid: Invalid digest）

## 推送镜像提示无权限

### 错误日志

上传镜像到SWR仓库，提示如下错误：

```
denied: you do not have the permission
```

```
[ERROR] : [pluginFrame] step run failed, errorMessage: DEV.CB.0210044, Docker push failed
```

### 分析处理

此错误表示当前用户对目标组织没有权限，请逐步排查以下可能的原因：

1. 编辑构建任务，单击“制作镜像并推送到SWR仓库”构建步骤，查看组织名。
2. 登录容器镜像服务，在组织管理里查看组织是否存在。
   - 组织不存在，**创建组织**即可（组织数不可超过上限）。
   - 组织存在，但当前用户对该组织没有编辑权限，推送镜像时仍然会出现此错误，管理员可参考**授权管理**选择性为当前用户授权。
   - 组织存在，且用户对该组织有编辑权限，那么请进入统一身份认证服务，检查该用户是不是在只读权限的用户组里，如果是，请移除该用户。

## 推送镜像提示组织数达到上限

### 错误日志

```
denied: The number of namespaces exceeds the upper limit
```

```
[ERROR] : [pluginFrame] step run failed, errorMessage: DEV.CB.0210044, Docker push failed
```

### 分析处理

推送镜像时，如果使用未被任何用户占用的全新组织名，SWR服务会尝试为当前租户创建该组织；由于SWR服务限制了每个租户可创建的组织数，如果超过此限制，则会出现该错误。

出现此错误时，使用管理员账号（或任何有SWR组织管理权限的账号）前往**组织管理**并切换至对应region，查看已有的组织列表，选择使用已有组织或删除无用组织即可。

## 推送镜像提示未登录

### 错误日志

```
denied: You may not login yet
 [ERROR] : [pluginFrame] step run failed, errorMessage: fail to execute docker command
```

**分析处理**

此类错误发生的原因一般有如下两种：

- push操作前未使用“docker login”命令登录，此时添加对应登录命令即可。
- 执行了登录命令，但是登录命令中SWR地址错误，导致执行没报错但实际登录未生效，需要核对登录命令是否正确。

## 推送镜像提示认证失败

**错误日志**

```
Error response from daemon: Get https://swr.example.example.com/v2/: denied: Authenticate Error
 [ERROR] : [pluginFrame] step run failed, errorMessage: fail to execute docker command.
```

**分析处理**

此类错误一般为SWR登录命令中账号/密码填写错误或临时登录账号信息已过期导致，**获取有效登录指令**重试即可。

## 推送镜像提示组织名非法

**错误日志**

```
invalid reference format
 [ERROR] : [pluginFrame] step run failed, errorMessage: fail to execute docker command.
```

**分析处理**

SWR服务对“组织”命名有相应格式要求，推送镜像时，如果使用的组织名不满足其格式要求，则会出现此错误。

出现此错误时，请前往**组织管理**并切换至对应region，核对填写的组织名是否正确；如果属于新建组织，请尝试按规范手动创建组织后再试。

## 推送镜像提示本地镜像不存在

**错误日志**

```
[2022-03-05 17:01:05.816] An image does not exist locally with the tag:
swr.example.example.com/demo/faqdemo1
 [ERROR] : [pluginFrame] step run failed, errorMessage: fail to execute docker command.
```

**分析处理**

此类错误一般为镜像制作失败或push命令中镜像名、标签等信息填写错误，导致push命令中期望的镜像与build/tag命令中实际生成的镜像不一致，需要检查镜像制作过程或push参数是否正确。

此例中镜像docker push swr.example.example.com/demo/faqdemo1:v1.1 中faqdemo1填写错误，build参数中指定的镜像名为faqdemo，修正push参数后再试即可。

## 推送镜像提示非法摘要

**错误日志**

```
digest invalid: Invalid digest
```

**分析处理**

此问题一般为SWR网络不稳定导致，重试几次即可。

# 7.3 执行构建任务时，拉取镜像失败

## 问题现象

执行构建任务时，日志报如下异常信息：

```
ERROR: docker pull image failed, dockerImage
```

## 原因分析

镜像拉取失败的原因可能有以下几种：

- 网络异常导致拉取超时。
- 拉取的镜像不存在。
- 拉取的镜像为私有镜像。

## 处理方法

- 网络异常导致，可以通过以下方法处理：
  a. 重试确认是否能解决，如果拉取失败的镜像是dockerHub镜像且重试无法解决，可参考**拉取dockerHub镜像超时或失败**。
  b. 如频繁出现或重试仍然失败请联系客服。
- 镜像不存在：请确保镜像已经上传至镜像仓，且镜像名称、镜像版本正确。
- 镜像为私有镜像：请将镜像设置为公开，或者先执行 *docker login* 鉴权通过后再执行 *docker pull* 操作。

# 7.4 使用 SWR 公共镜像时拉取镜像无权限

## 问题现象

执行构建任务时，日志报如下异常信息：

```
Get https://swr.example.example.com/v2/codeciexample-test/demo/manifests/v1.1: denied: You may not login yet
```

## 原因分析

构建任务中有“使用SWR公共镜像”构建步骤时，由于构建所调用的Docker镜像没有设置权限为公开导致报错。

## 处理办法

进入容器镜像服务，找到构建过程所使用到的镜像，编辑镜像将镜像的类型设置为“公开”，具体操作如下：

1. 登录容器镜像服务。

2. 在左侧导航单击“我的镜像”，然后单击镜像名称进入镜像详情页面，然后单击右上角“编辑”。
3. 在编辑框中，将“类型”设置为“公开”。


# 7.5 镜像仓库登录异常

## 问题现象

异常信息如下：

```
Error response from daemon: login attempt to https://hub.docker.com/v2/ failed with status: 404 Not Found
```

## 原因分析

镜像仓库地址填写有误，编译构建不支持自定义https请求的镜像仓库。

## 处理方法

镜像仓库地址保持系统提供的默认值。

# 7.6 如何推送镜像到其他租户

## 问题现象

制作镜像并推送到SWR仓库时，提示错误信息“DEV.CB.0210043”，并提示制作Docker镜像失败。

## 处理方法

1. 进入编译构建服务首页。
2. 选择对应的构建任务，单击任务所在行的 ··· ，单击“编辑”。

3. 在“构建步骤”页面编辑“制作镜像并推送到SWR仓库”。
4. 单击“管理IAM账号”。


5. 单击“新建扩展服务点”，选择“IAM账户”。


6. 在弹出的窗口中填写参数信息。


Access Key Id和Secret Access Key获取方式如下：

a. 单击页面左上角“控制台”。
b. 单击页面右上角账号名称，选择“我的凭证”。
c. 单击“访问密钥”。
d. 单击“新增访问密钥”，填写相关描述，单击“确定”。


e. 在弹出的窗口中单击“立即下载”，可将密钥信息下载到本地。


7. 步骤4中的IAM账号选择步骤6中新建的服务扩展点。

# 7.7 构建时拉取 dockerhub 镜像超时/次数限制

## 问题现象

执行构建任务时，拉取dockerhub镜像超时/次数限制，日志报如下异常信息：

```
Error response from daemon: Get https://registry.docker-cn.com/v2/: net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)
```

或

```
toomanyrequests: You have reached your pull rate limit. You may increase the limit by authenticating and upgrading: https://www.docker.com/increase-rate-limit
```

## 原因分析

可能是因为dockerhub镜像源的网络不稳定并且存在频率限制，容易导致拉取超时或失败。可以将dockerhub镜像源的镜像迁移到SWR上，再拉取镜像。

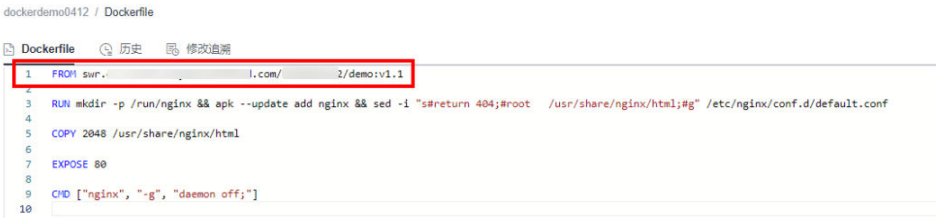
## 处理方法

**步骤1** 下载dockerhub镜像源的镜像到本地。

**步骤2** 参考**页面上传镜像**页面，上传镜像到SWR。

**步骤3** 在镜像详情页面中，单击对应镜像版本“下载指令”列的复制图标，复制镜像下载指令。


**步骤4** 修改代码仓中Dockerfile文件，将文件中镜像地址修改为**步骤3**中拷贝的地址。




**----结束**