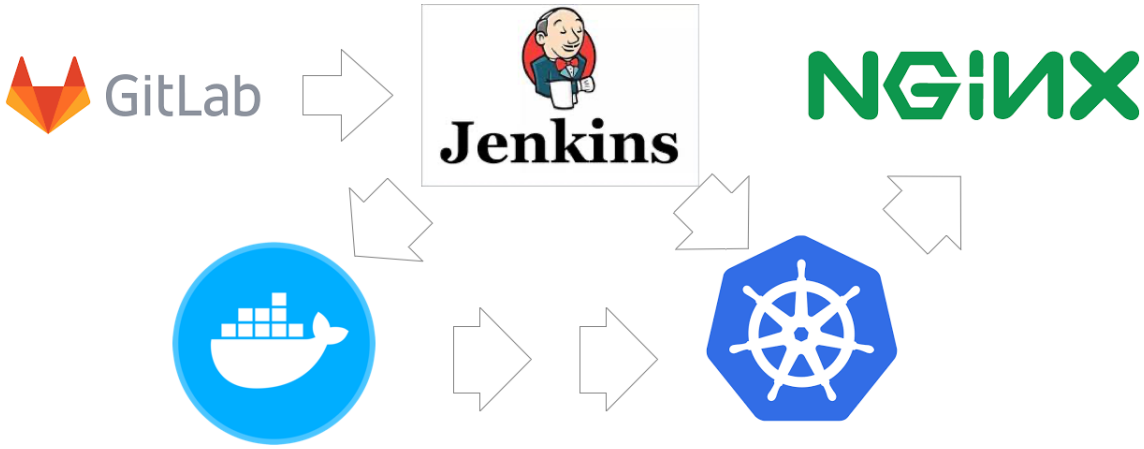
讲师(老司机)

容器虚拟化技术和自动化部署



skywalking源码解析

初始化项目

本地调试必须先从源码编译Skywalking，有两种方式，一种是从GitHub拉取代码，一种是从Apache Skywalking的release页面下载代码。区别在于GitHub上面的代码是使用git module管理的，拉取下来需要执行一系列操作，最主要的是没有科学上网的话，速度比较慢。Release页面下载的是已经把依赖关系全部整理好的代码，整个源码包不到3MB，还有很多国内镜像地址，所以下载非常快。两种我都使用过，我的建议是：如果你想看历史提交记录或者想持续跟上游版本的话，就选用从GitHub拉取代码的方式；如果你想方便或者从GitHub clone超级慢的话，建议直接从Release处下载。不管哪种，编译以及导入IDEA或Eclipse官方文档写的都比较详细，我就不做翻译了，基本都是命令操作。源码编译成功以后（务必保证编译成功），就可以准备进行调试了。

官网教程

```
https://github.com/apache/skywalking/blob/master/docs/en/guides/How-to-build.md
```

下载地址

github

```
https://github.com/apache/skywalking
```

官网

```
http://skywalking.apache.org/zh/downloads/

在页面中找到->所有历史版本->归档库->对应版本->下载(apache-skywalking-apm-6.6.0-src.tgz)
```

官网安装skywalking步骤

通过官网下载skywalking对应版本。

```
1.解压缩apache-skywalking-apm-6.6.0-src.tgz

2.导入idea开发工具

3.下载maven依赖

4.所有项目官方都已加载完毕。编译工程
mvn clean package -Dmaven.test.skip=true
```

git安装skywalking步骤

注意事项

无论是那种安装方式，都有一些jar国内无法下载。需要修改maven的settings文件，临时注释掉aliyun的私服镜像仓库地址。

clone方式

```
克隆skyswalking源码：
git clone https://github.com/apache/skywalking.git
cd skywalking/

初始化子模块
git submodule init
更新子模块
git submodule update

mvn clean package -Dmaven.test.skip=true
```

zip包方式

```
在github官网选择skywalking对应的tag分支。下载zip包。
```

javaagent

首先说一下javaagent是什么。javaagent是一种能够在不影响正常编译的情况下，修改字节码的技术。JavaAgent 是JDK 1.5 以后引入的，也可以叫做Java代理。JavaAgent 是运行在 main方法之前的拦截器，它内定的方法名叫 premain ，也就是说先执行 premain 方法然后再执行 main 方法。

javaagent是java命令的一个参数。参数 javaagent 可以用于指定一个 jar 包，并且对该 java 包有2个要求：

1. 这个 jar 包的MANIFEST.MF 文件必须指定 Premain-Class 项。
2. Premain-Class 指定的那个类必须实现 premain（）方法。

JavaAgent 实际应用

1. 可以在加载java文件之前做拦截把字节码做修改
2. 获取所有已经被加载过的类
3. 获取某个对象的大小
4. 将某个jar加入到bootstrapclasspath里作为高优先级被bootstrapClassloader加载
5. 将某个jar加入到classpath里供AppClassload去加载
6. 设置某些native方法的前缀，主要在查找native方法的时候做规则匹配

agent案例一

agentdemo1项目

普通maven工程。该项目主要编写premain方法。

```
package com.lagou.agent;

import java.lang.instrument.Instrumentation;

public class PremainTest1 {

    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println("hello agent demo1");
        System.out.println("agentArgs =====> " + agentArgs);
    }
}
```

MANIFREST.MF 手工编写

不推荐使用这种方式。比较麻烦，简单了解一下即可。 在 resources 目录下新建目录：META-INF，在该目录下新建文件MANIFREST.MF 。文件内容如下：

```
Manifest-Version: 1.0
Premain-Class: com.lagou.agent.PremainTest1
Agent-Class: com.lagou.agent.PremainTest1
Can-Redefine-Classes: true
Can-Retransform-Classes: true
Build-Jdk-Spec: 1.8
Created-By: Maven Jar Plugin 3.2.0
```

这里如果你不去手动指定的话，直接打包，默认会在打包的文件中生成一个MANIFREST.MF文件 。 默认的文件中包含当前的一些版本信息，当前工程的启动类。我们需要增加参数做更多的事情。

> Premain-Class： 包含 premain 方法的类（类的全路径名）
>
> Agent-Class： 包含 agentmain 方法的类（类的全路径名）
>
> Boot-Class-Path： 设置引导类加载器搜索的路径列表。查找类的特定于平台的机制失败后，引导类加载器会搜索这些路径。按列出的顺序搜索路径。列表中的路径由一个或多个空格分开。路径使用分层 URI 的路径组件语法。如果该路径以斜杠字符（“/”）开头，则为绝对路径，否则为相对路径。相对路径根据代理 JAR 文件的绝对路径解析。忽略格式不正确的路径和不存在的路径。如果代理是在 VM 启动之后某一时刻启动的，则忽略不表示 JAR 文件的路径。（可选）
>
> Can-Redefine-Classes： true表示能重定义此代理所需的类，默认值为 false（可选）
>
> Can-Retransform-Classes： true 表示能重转换此代理所需的类，默认值为 false （可选）
>
> Can-Set-Native-Method-Prefix： true表示能设置此代理所需的本机方法前缀，默认值为 false（可选）

MANIFREST.MF 插件一

介绍几种主流的可执行jar包打包插件。

使用maven-jar-plugin插件进行打包。

```
<build>
        <finalName>agentdemo1</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <archive>
                        <!--自动添加META-INF/MANIFEST.MF -->
                        <manifest>
                            <addClasspath>true</addClasspath>
                        </manifest>
                        <manifestEntries>
                            <!--permain方法所在类的完全限定名-->
                            <Premain-
Class>com.lagou.agent.PremainTest1</Premain-Class>
                            <Agent-
Class>com.lagou.agent.PremainTest1</Agent-Class>
                            <Can-Redefine-Classes>true</Can-Redefine-
Classes>
                            <Can-Retransform-Classes>true</Can-Retransform-
Classes>
                        </manifestEntries>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

MANIFREST.MF 插件二

使用maven-assembly-plugin插件进行打包。

```
<plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <appendAssemblyId>false</appendAssemblyId>
                    <descriptorRefs>
                        <descriptorRef>jar-with-
dependencies</descriptorRef>
                    </descriptorRefs>
                    <archive>
                        <!--自动添加META-INF/MANIFEST.MF -->
                        <manifest>
                            <addClasspath>true</addClasspath>
                        </manifest>
                        <manifestEntries>
                            <Premain-
Class>com.lagou.agent.PremainTest1</Premain-Class>
                            <Agent-
Class>com.lagou.agent.PremainTest1</Agent-Class>
                            <Can-Redefine-Classes>true</Can-Redefine-
Classes>
                            <Can-Retransform-Classes>true</Can-Retransform-
Classes>
                        </manifestEntries>
                    </archive>
                </configuration>
                <!--
                其中<phase>package</phase>、<goal>single</goal>
                即表示在执行package打包时，执行assembly:single
                 -->
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
```

注意事项

```
选择插件二方式打包时。如果java.exe进程被占用，使用mvn clean package命令或报错。可以直接在idea插件中assembly中执行assembly:single。或者执行mvn package assembly:single命令。
```

MANIFREST.MF 插件三

使用maven-shade-plugin插件进行打包。

```
<plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

 implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTr
ansformer">
                                    <manifestEntries>
                                        <Premain-
Class>com.lagou.agent.PremainTest1</Premain-Class>
                                    </manifestEntries>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
```

打包命令

```
mvn clean package
```

编译工程

选择插件一方式。将agentdemo1项目打包。

```
mvn clean package
```

agenttest1项目

普通maven工程。该项目主要编写main方法。

编写main方法

```
package com.lagou.test1;

public class AgentTest1 {
    public static void main(String[] args) {
        System.out.println("这里是agent第一个main方法测试。");
    }
}
```

测试main方法

```
测试main方法是否正常运行。
```

idea配置

必须先运行一次main，然后点击编辑AgentTest1启动类->edit configurations -> VM options

测试agent一

不带参数测试

```
-javaagent:D:\ideaworkspaces\202001test\agentdemo1\target\agentdemo1.jar

运行main方法测试：
```

测试agent二

带参数测试

```
-
javaagent:D:\ideaworkspaces\202001test\agentdemo1\target\agentdemo1.jar=lagou

运行main方法测试：
```

permain方法

重点就在 premain 方法，也就是我们今天的标题。从字面上理解，就是运行在 main 函数之前的的类。当Java 虚拟机启动时，在执行 main 函数之前，JVM 会先运行 -javaagent 所指定 jar 包内 Premain-Class 这个类的 premain 方法，其中，该方法可以签名如下：

1.public static void premain(String agentArgs, Instrumentation inst)
2.public static void premain(String agentArgs)

JVM 会优先加载 1 签名的方法，加载成功忽略 2，如果1 没有，加载 2 方法。

这个逻辑在sun.instrument.InstrumentationImpl 类中：

> InstrumentationImpl 类源码loadClassAndStartAgent方法的217行左右。

```
private void loadClassAndStartAgent(String var1, String var2, String
var3) throws Throwable {
```

```
    ClassLoader var4 = ClassLoader.getSystemClassLoader();
    Class var5 = var4.loadClass(var1);
    Method var6 = null;
    NoSuchMethodException var7 = null;
    boolean var8 = false;

    try {
        var6 = var5.getDeclaredMethod(var2, String.class,
Instrumentation.class);
        var8 = true;
    } catch (NoSuchMethodException var13) {
        var7 = var13;
    }

    if (var6 == null) {
        try {
            var6 = var5.getDeclaredMethod(var2, String.class);
        } catch (NoSuchMethodException var12) {
        }
    }
```

```
package com.lagou.agent;

import java.lang.instrument.Instrumentation;

public class PremainTest1 {
/*
    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println("hello agent demo1");
        System.out.println("agentArgs =====> " + agentArgs);
    }
    */

    public static void premain(String agentArgs) {
        System.out.println("hello agent demo2");
        System.out.println("agentArgs =====> " + agentArgs);
    }
}
```

agent案例二

agentdemo1项目

```
package com.lagou.agent;

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.lang.instrument.Instrumentation;
import java.security.ProtectionDomain;

public class PremainTest1 {
```

```
9
10      public static void premain(String agentArgs, Instrumentation inst) {
11          System.out.println("hello agent demo1");
12          System.out.println("agentArgs =====> " + agentArgs);
13          inst.addTransformer(new ClassFileTransformer() {
14              @Override
15              public byte[] transform(ClassLoader loader, String className,
    Class<?> classBeingRedefined,
16                                      ProtectionDomain protectionDomain,
    byte[] classfileBuffer)
17                      throws IllegalClassFormatException {
18                  System.out.println("premain load Class     :" + className);
19                  return classfileBuffer;
20              }
21          }, true);
22      }
23
24
25      public static void premain(String agentArgs) {
26          System.out.println("hello agent demo2");
27          System.out.println("agentArgs =====> " + agentArgs);
28      }
29  }
```

JVMTI（Java Virtual Machine Tool Interface）是一套本地编程接口集合，它提供了一套"代理"程序机制，可以支持第三方工具程序以代理的方式连接和访问 JVM，并利用 JVMTI 提供的丰富的编程接口，完成很多跟 JVM 相关的功能。关于 JVMTI 的详细信息，请参考 Java SE 6 文档当中的介绍。

java.lang.instrument 包的实现，也就是基于这种机制的：在 Instrumentation 的实现当中，存在一个 JVMTI 的代理程序，通过调用 JVMTI 当中 Java 类相关的函数来完成 Java 类的动态操作。

Instrumentation 的最大作用，就是类定义动态改变和操作。在 Java SE 5 及其后续版本当中，开发者可以在一个普通 Java 程序（带有 main 函数的 Java 类）运行时，通过 `- javaagent` 参数指定一个特定的 jar 文件（包含 Instrumentation 代理）来启动 Instrumentation 的代理程序。

和类加载器比较

类加载器也可以实现运行时修改代码。但是对代码的侵入性很高。使用 java agent 能让修改字节码这个动作化于无形，对业务透明，减少侵入性。

agent的缺点

需要设置参数javaagent

javassist使用

目的

使用javaagent和javassist实现了把特定包下面的class的所有类的所有有body的非native的方法前面添加一句 ***System.out.println("hello im agent :" + ctMethod.getName());***

javassist简介

Java 字节码以二进制的形式存储在 .class 文件中，每一个 .class 文件包含一个 Java 类或接口。Javassist 就是一个用来 处理 Java 字节码的类库。它可以在一个已经编译好的类中添加新的方法，或者是修改已有的方法，并且不需要对字节码方面有深入的了解。同时也可以去生成一个新的类对象，通过完全手动的方式。

增加依赖

agentdemo1项目的pom文件增加javassist依赖包

```
    <dependencies>
        <dependency>
            <groupId>org.javassist</groupId>
            <artifactId>javassist</artifactId>
            <version>3.27.0-GA</version>
        </dependency>
    </dependencies>
```

FirstAgent类

FirstAgent类实现ClassFileTransformer接口。重写transform方法。实现 ClassFileTransformer 这个接口的目的就是在class被装载到JVM之前将class字节码转换掉，从而达到动态注入代码的目的。其中代码的修改使用到了javassist。

FirstAgent类实现了把特定包下面的class的所有类的所有有body的非native的方法前面添加一句 ***System.out.println("hello im agent :" + ctMethod.getName());***

```
package com.lagou.instrumentationimpl;

import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;
import javassist.bytecode.CodeAttribute;

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;

public class FirstAgent implements ClassFileTransformer {

    //对特定包下的类修改字节码
    public final String injectedClassName = "com.lagou.test";

    @Override
    public byte[] transform(ClassLoader loader, String className,
                            Class<?> classBeingRedefined,
                            ProtectionDomain protectionDomain,
```

```
                            byte[] classfileBuffer)
            throws IllegalClassFormatException {

        className = className.replace("/", ".");
        if (className.startsWith(injectedClassName)) {
            CtClass ctclass = null;
            try {
                // 使用全称,用于取得字节码类，使用javassist技术实现
                ctclass = ClassPool.getDefault().get(className);
                CtMethod[] ctmethods = ctclass.getMethods();
                for (CtMethod ctMethod : ctmethods) {
                    CodeAttribute ca =
ctMethod.getMethodInfo2().getCodeAttribute();
                    if (ca == null) {
                        continue;
                    }
                    if (!ctMethod.isEmpty()) {
                        ctMethod.insertBefore("System.out.println(\"hello
Im agent : " + ctMethod.getName() + "\");");
                    }
                }
                return ctclass.toBytecode();
            } catch (Exception e) {
                System.out.println(e.getMessage());
                e.printStackTrace();
            }
        }
        return null;
    }
}
```

PremainTest1类

```
package com.lagou.agent;

import com.lagou.instrumentationimpl.FirstAgent;

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.lang.instrument.Instrumentation;
import java.security.ProtectionDomain;

public class PremainTest1 {

    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println("hello agent demo1");
        System.out.println("agentArgs =====> " + agentArgs);

        inst.addTransformer(new FirstAgent());
    }


    public static void premain(String agentArgs) {
```

```
        System.out.println("hello agent demo2");
        System.out.println("agentArgs =====> " + agentArgs);
    }
}
```

更新打包插件

使用maven-jar-plugin插件进行打包。无法将第三方依赖的javassist包进行打包。如果需要将依赖包打入jar中。需要额外引入maven-dependency-plugin插件。为了简单，使用maven-shade-plugin进行打包。

修改agentdemo1的pom文件。将打包插件更换为maven-shade-plugin。

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer
 implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                    <manifestEntries>
                                        <Premain-Class>com.lagou.agent.PremainTest1</Premain-Class>
                                    </manifestEntries>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
```

打包

```
mvn clean package
```

agenttest1项目

修改AgentTest1类

```
package com.lagou.test1;

```

```java
public class AgentTest1 {
    public static void main(String[] args) {
        System.out.println("这里是agent第一个main方法测试。");
        testAgent();

        testAgent1("one", "two");
    }

    public static void testAgent() {
        System.out.println("test agent say hello");
    }

    public static void testAgent1(String one, String two) {
        System.out.println("test agent say hello" + one + two);
    }
}
```

测试

```
测试结果如下：

hello agent demo1
agentArgs =====> lagou
hello Im agent : main
这里是agent第一个main方法测试。
hello Im agent : testAgent
test agent say hello
hello Im agent : testAgent1
test agent say helloonetwo

Process finished with exit code 0
```

独立测试

脱离idea开发工具，独立使用java -javaagent命令测试。

通过 -javaagent 参数来指定我们的Java代理包，值得一说的是 -javaagent 这个参数的个数是不限的，如果指定了多个，则会按指定的先后执行，执行完各个 agent 后，才会执行主程序的 main 方法。

特别提醒：如果你把 -javaagent 放在 -jar 后面，则不会生效。也就是说，放在主程序后面的 agent 是无效的。

class方式

agentdemo1项目打jar包

agenttest1项目没有打包。进入 ..\agenttest1\target\classes目录中，执行如下命令

```
java -javaagent:e:\javaagent\agentdemo1.jar com.lagou.test1.AgentTest1
```

jar包方式

agentdemo1和agenttest1两个项目全部打jar包。

```
    <build>
        <finalName>agenttest1</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <archive>
                        <!--自动添加META-INF/MANIFEST.MF -->
                        <manifest>
                            <addClasspath>true</addClasspath>

 <mainClass>com.lagou.test1.AgentTest1</mainClass>
                        </manifest>

                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

```
agentdemo1项目打包：将agentdemo1.jar复制到d:\javaagent目录中
mvn clean package

agenttest1项目打包：将agenttest1.jar复制到d:\javaagent目录中


使用java -javaagent命令，不带参数
java -javaagent:d:\javaagent\agentdemo1.jar -jar agenttest1.jar

使用java -javaagent命令，带参数
java -javaagent:d:\javaagent\agentdemo1.jar -jar agenttest1.jar
```