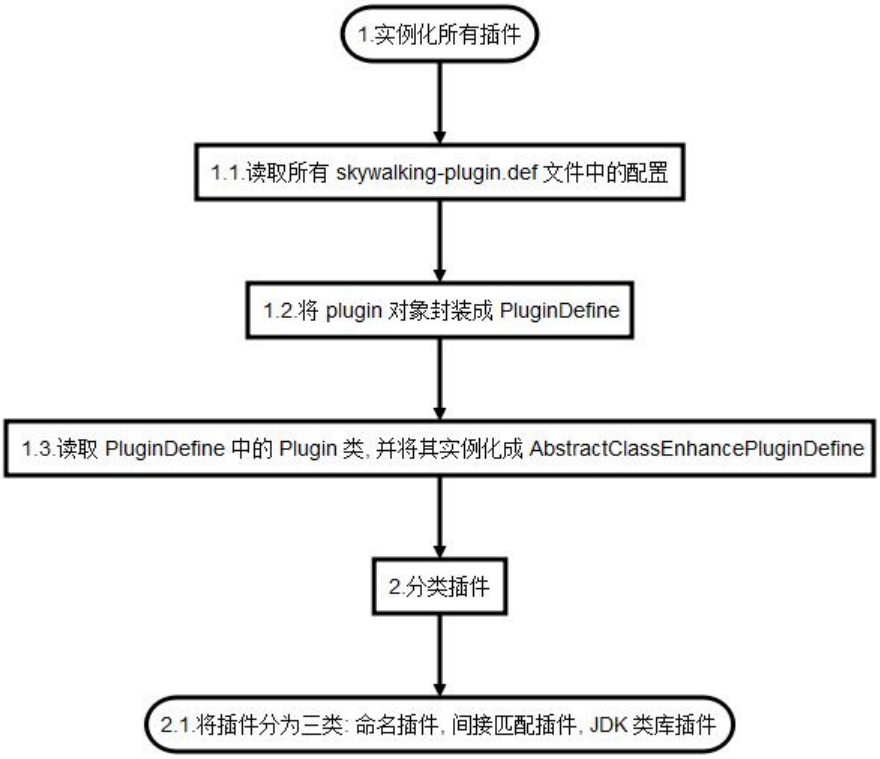
SkyWalking Java Agent 插件加载

作者：noelcliu
原文链接：https://ld246.com/article/1669956995032
来源网站：链滴

1. SkyWalking Java Agent 总览
2. SkyWalking Java Agent 配置文件加载
3. SkyWalking Java Agent 插件加载
4. SkyWalking Java Agent 插件详解
5. SkyWalking Java Agent 自定义插件
6. SkyWalking Java Agent 微内核剖析
7. Skywalking Java Agent 服务详解

项目源码注释地址：https://github.com/ChuckChen123/skywalking-java/tree/chuck_learn

回顾



上图是之前总览中插件加载的流程图，对于 SkyWalking Java Agent 的插件，我将分为两篇文章进讲解，本篇主要讲解 SkyWalking Java Agent 的插件加载过程，而另外一篇主要讲解 SkyWalking Ja a Agent 是如何设计的，以及如何自己编写一个插件，那么让我们开始吧！

SkyWalking Java Agent 插件的加载流程主要分为两个阶段：

- 实例化插件
- 分类插件

让我们看看源码是如何实现的吧！

实例化插件

我们还是从 SkyWalking Java Agent 的入口 premain 方法开始吧。在 premain 方法中，执行插件载相关的代码为

```
public static void premain(String agentArgs, Instrumentation instrumentation) throws PluginE
ception {

    ...

    // 2 加载插件
    pluginFinder = new PluginFinder(new PluginBootstrap().loadPlugins());

    ...

}
```

其中 PluginBootstrap 是负责从所有插件的配置文件中读取插件类，然后将插件加载到内存中并实例包装成 PluginDefine 对象，而 PluginFinder 则负责管理所有的 PluginDefine，它会将所有的 Plugi Define 分类，以便程序后续使用更加方便。

实例化插件步骤如下：

1. 初始化 AgentClassLoader
2. 读取所有 plugin jar 文件里的 skywalking-plugin.def 配置文件，并将其实例化为 PluginDefine
3. 使用 AgentClassLoader 实例化 PluginDefine value 中的类（也就是插件的具体实现类）
4. 动态注入插件，通过 Java SPI 机制，扫描所有实现了 InstrumentationLoader 接口的类，并执行其 load 方法，将结果加入到插件列表中

初始化 AgentClassLoader

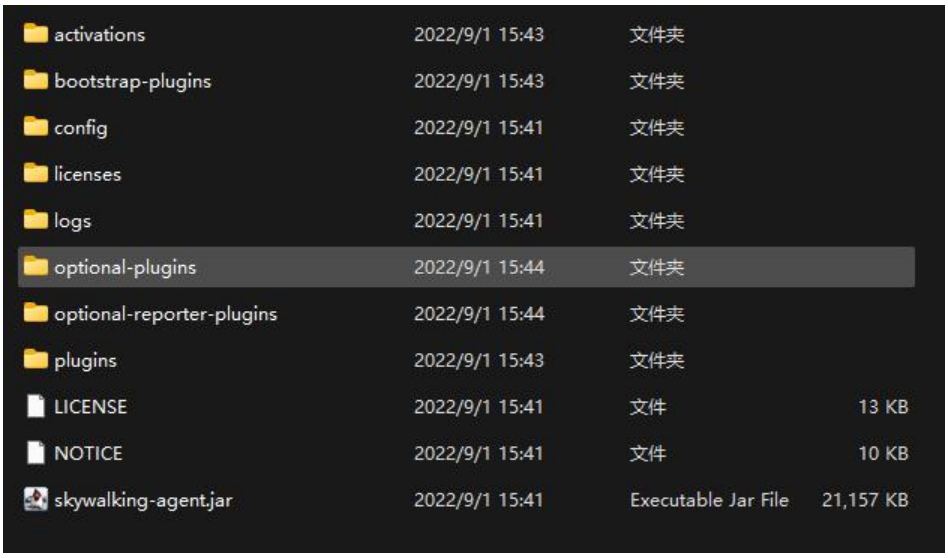
所有的加载实现都在 PluginBootstrap 类的 loadPlugins() 方法中，初始化 AgentClassLoader 的代如下：

```
public List<AbstractClassEnhancePluginDefine> loadPlugins() throws AgentPackageNotFoun
Exception {

        // 初始化一个 ClassLoader 用来加载 plugin
        AgentClassLoader.initDefaultLoader();

        ...

    }

public class AgentClassLoader extends ClassLoader {

    // 初始化 AgentClassLoader
    public static void initDefaultLoader() throws AgentPackageNotFoundException {
        if (DEFAULT_LOADER == null) {
            synchronized (AgentClassLoader.class) {
                if (DEFAULT_LOADER == null) {
                    DEFAULT_LOADER = new AgentClassLoader(PluginBootstrap.class.getClassLoader
));
                }
            }
        }
    }

    // 构造方法
    public AgentClassLoader(ClassLoader parent) throws AgentPackageNotFoundException {
        super(parent);
        File agentDictionary = AgentPackagePath.getPath();
        classpath = new LinkedList<>();
        // 将需要加载的 jar folder 加入到 classpath 中
        Config.Plugin.MOUNT.forEach(mountFolder -> classpath.add(new File(agentDictionary,
ountFolder)));
    }
}
```

可以看到 AgentClassLoader 实例化的代码是很简短的，就只是使用了双重检查的方式实现了一个单的 AgentClassLoader，AgentClassLoader 的父 ClassLoader 是加载 PluginBootstrap 类的 ClassLader，那么为什么 SkyWalking 要单独定义一个 ClassLoader 来实现功能，而不用加载 PluginBoostap 类的 ClassLoader呢？这可能跟 AgentClassLoader 所需要实现的功能有关系。

在我们使用 SkyWalking Java Agent 的时候，并不仅仅需要 skywalking-agent.jar 这一个 jar 包就以了，我们下载解压 skywalking-agent 包之后可以看到，里面并不只包含 skywalking-agent.jar，有许多其他的文件夹，这里面就包含所需要加载的 plugin，负责加载这些 plugin 的就是咱们的 Agen ClassLoader。同时 AgentClassLoader 应该还有另外一个任务，就是将 plugin 的类与程序的类隔开。

AgentClassLoader 获取需要加载 jar folder 的代码在构造方法中，它会从 Config 类中读取 Plugin. OUNT，将所有的 folder 放入到类变量 classpath 中。而 AgentClassLoader 中的其他方法会在实化 PluginDefine 中用到，主要用来解析所有的 jar 包，并返回对应路径，具体实现可以自行查看源。



activations	2022/9/1 15:43	文件夹	
bootstrap-plugins	2022/9/1 15:43	文件夹	
config	2022/9/1 15:41	文件夹	
licenses	2022/9/1 15:41	文件夹	
logs	2022/9/1 15:41	文件夹	
optional-plugins	2022/9/1 15:44	文件夹	
optional-reporter-plugins	2022/9/1 15:44	文件夹	
plugins	2022/9/1 15:43	文件夹	
LICENSE	2022/9/1 15:41	文件	13 KB
NOTICE	2022/9/1 15:41	文件	10 KB
skywalking-agent.jar	2022/9/1 15:41	Executable Jar File	21,157 KB

实例化 PluginDefine

实例化 PluginDefine 主要源码如下：

```
public List<AbstractClassEnhancePluginDefine> loadPlugins() throws AgentPackageNotFoun
Exception {

    ...

    // 1 使用 AgentClassLoader 来获取所有配置文件路径
    PluginResourcesResolver resolver = new PluginResourcesResolver();
    List<URL> resources = resolver.getResources();

    ...

    // 2 将文件转换为 PluginDefine
    PluginCfg.INSTANCE.load(pluginUrl.openStream());

    ...

    List<PluginDefine> pluginClassList = PluginCfg.INSTANCE.getPluginClassList();

    ...

  }

public class PluginResourcesResolver {
```

```
    // 1.1调用 AgentClassLoader findResources，获取所有 plugin 的 skywalking-plugin.def 配置
件
    public List<URL> getResources() {
        List<URL> cfgUrlPaths = new ArrayList<URL>();
        Enumeration<URL> urls;
        try {
            urls = AgentClassLoader.getDefault().getResources("skywalking-plugin.def");

            while (urls.hasMoreElements()) {
                URL pluginUrl = urls.nextElement();
                cfgUrlPaths.add(pluginUrl);
                LOGGER.info("find skywalking plugin define in {}", pluginUrl);
            }

            return cfgUrlPaths;
        } catch (IOException e) {
            LOGGER.error("read resources failure.", e);
        }
        return null;
    }
}

public class AgentClassLoader extends ClassLoader {

    // 1.2读取配置文件
    protected Enumeration<URL> findResources(String name) throws IOException {
        List<URL> allResources = new LinkedList<>();
        List<Jar> allJars = getAllJars();
        for (Jar jar : allJars) {
            JarEntry entry = jar.jarFile.getJarEntry(name);
            if (entry != null) {
                allResources.add(new URL("jar:file:" + jar.sourceFile.getAbsolutePath() + "!/" + name
);
            }
        }

        final Iterator<URL> iterator = allResources.iterator();
        return new Enumeration<URL>() {
            @Override
            public boolean hasMoreElements() {
                return iterator.hasNext();
            }

            @Override
            public URL nextElement() {
                return iterator.next();
            }
        };
    }
}
```

这一步主要就是读取所有 plugin jar包中的 skywalking-plugin.def 的配置文件，并将其封装成为 PluinDefine，其实 skywalking-plugin.def 就是一个 key value配置文件，用 = 隔开（也就是 key = va

ue），所以 PluginDefine 的内容也比较简单。可以看到上面源码选段中的 1.2，基本上就是拼接 URL 的操作，就是在 jar 包中找到 resource 文件下面的 skywalking-plugin.def 文件，最终交给 PluginCg 去处理文件流，处理文件流的细节这里就不作过多的讲解了，下面是一个 rocketMQ 的配置文件内：

```
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

rocketMQ-4.x=org.apache.skywalking.apm.plugin.rocketMQ.v4.define.ConsumeMessageConcurrentlyInstrumentation
rocketMQ-4.x=org.apache.skywalking.apm.plugin.rocketMQ.v4.define.ConsumeMessageOrderlyInstrumentation
rocketMQ-4.x=org.apache.skywalking.apm.plugin.rocketMQ.v4.define.MQClientAPIImplInstrumentation
rocketMQ-4.x=org.apache.skywalking.apm.plugin.rocketMQ.v4.define.SendCallbackInstrumentation
```

实例化和动态注入 Plugin

实例化和动态注入 Plugin 源码如下：

```
public class PluginBootstrap {

    public List<AbstractClassEnhancePluginDefine> loadPlugins() throws AgentPackageNotFo
ndException {

        ...

        List<AbstractClassEnhancePluginDefine> plugins = new ArrayList<>();
        for (PluginDefine pluginDefine : pluginClassList) {
            try {
                LOGGER.debug("loading plugin class {}.", pluginDefine.getDefineClass());
                // 读取配置文件中的 value，并实例化为 AbstractClassEnhancePluginDefine
                AbstractClassEnhancePluginDefine plugin = (AbstractClassEnhancePluginDefine) Cl
ss
                        .forName(pluginDefine.getDefineClass(), true, AgentClassLoader
                        .getDefault()).newInstance();
                plugins.add(plugin);
            } catch (Throwable t) {
                LOGGER.error(t, "load plugin [{}] failure.", pluginDefine.getDefineClass());
            }
        }

        // 动态加载所有实现了 InstrumentationLoader 的插件
        plugins.addAll(DynamicPluginLoader.INSTANCE.load(AgentClassLoader.getDefault()));
```

```
    }
}
```

这一步就是将上一步实例化的 PluginDefine 的 value 依次用 AgentClassLoader 进行实例化，并存入 plugins，AbstractClassEnhancePluginDefine 是一个比较重要的类，它是所有 plugin 的父类，为 plugin 定义了具体的架构和逻辑；处理完 PluginDefine 之后，就会对实现了 InstrumentationLoader 的 loader 进行处理，这里用到了 Java 的 SPI 机制，会将所有 InstrumentationLoader 的实现类获到并且调用其 load() 方法来获取 plugin，并且加入到 plugins 中。

分类插件

PluginBootstrap 加载实例化完 Plugin 之后，会将所有的 plugin 交给 PluginFinder 进行管理，PluginFinder 会将所有的 plugin 分类，存储到不同的 list 中，并提供对应的方法获取这些 plugin，源码下：

```
public class PluginFinder {

    public PluginFinder(List<AbstractClassEnhancePluginDefine> plugins) {

        // 为插件做分类，分别放到nameMatchDefine、SignatureMatchDefine和bootstrapClassMatchDefine
        for (AbstractClassEnhancePluginDefine plugin : plugins) {
            ClassMatch match = plugin.enhanceClass();

            if (match == null) {
                continue;
            }

            if (match instanceof NameMatch) {
                NameMatch nameMatch = (NameMatch) match;
                LinkedList<AbstractClassEnhancePluginDefine> pluginDefines = nameMatchDefine.get(nameMatch.getClassName());
                if (pluginDefines == null) {
                    pluginDefines = new LinkedList<AbstractClassEnhancePluginDefine>();
                    nameMatchDefine.put(nameMatch.getClassName(), pluginDefines);
                }
                pluginDefines.add(plugin);
            } else {
                signatureMatchDefine.add(plugin);
            }

            if (plugin.isBootstrapInstrumentation()) {
                bootstrapClassMatchDefine.add(plugin);
            }
        }
    }
}
```

源码中遍历了 plugins，并调用 plugin 的 enhanceClass() 方法，根据返回的 ClassMatch 将 plugin 分为 nameMatchDefine、signatureMatchDefine 和 bootstrapClassMatchDefine；关于分类的用以及 AbstractClassEnhancePluginDefine 中的方法将会在 Plugin 详解中进行深入解析。

总结

至此 Plugin 的加载流程就分析完毕了，后续将会对 Plugin 的具体使用以及设计进行剖析，如果有问或者建议，欢迎大家留言，谢谢。