# SkyWalking Java Agent 插件详解

作者：noelcliu
原文链接：https://ld246.com/article/1669960877529
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

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

上一篇我们了解了 SkyWalking Java Agent 是如何进行插件的加载的，这一篇我将深入讲解一下 Sk Walking Java Agent 的插件机制，从源码的角度来看看一个插件是如何工作的。

# 插件的作用

在讲解插件之前，我们需要先知道插件在 SkyWalking Java Agent 中的作用是什么；我们都知道 Sk Walking 是一个全链路的监控插件，使用了 ByteBuddy 框架来实现，也就是相当于它在我们的工程码的某一些方法的前后加入了它预设的逻辑，比如设置开始时间结束时间、设置标志位等等，然后通收集这些数据来达成监控的目的。那么它是怎么知道要在哪些方法前后插入逻辑呢、插入逻辑是定义哪里的呢，而且是全链路的，也就是在微服务中，它可以监控几乎所有的微服务框架，这些都是怎么现的呢。

这就是插件做的事情，SkyWalking Java Agent 根据不同的框架，不同的版本，定义了不同的 plugi ，然后依据 witnessClass 机制加载对应的插件，完成动态全链路配置的需求，通过插件不仅将不同框架和版本的逻辑分开，而且当我们需要自定义插件的时候也可以很方便的对 SkyWalking Java Agen 进行增强，不得不说，这个设计是很妙的。

# 插件的工作原理

我们可以先自己想想，如果要完成一个插件的设计，需要做什么事情。让我们先提出问题，然后带着题去看 SkyWalking Java Agent 是如何实现的：

1.如何基于 ByteBuddy 实现注入功能呢？

2.如何根据目标的特征找到对应的注入逻辑呢？

3.如何确定不同的代码版本注入不同的逻辑呢？

4.对于同一个类的同一个方法，有多段注入逻辑，我们又应该怎么处理呢？

# 架构设计

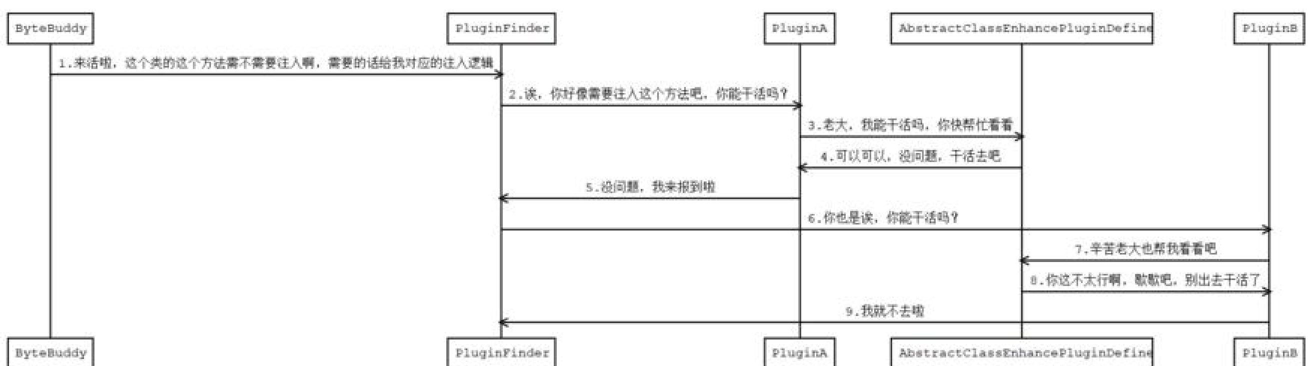
那么接下来就有请我们插件的三位顶层玩家：

- **ByteBuddy**: 负责具体的注入工作（什么设计不设计，最后的注入工作还不是得我来，麻溜的把注的东西给我，早注入完早下班~）
- **PluginFinder**: 负责所有的 Plugin 管理（来来来，所有的 Plugin 听我号令，你属于这类；你你你到那边去......）
- **AbstractClassEnhancePluginDefine**: Plugin 的顶层类，负责判断 Plugin 是否生效以及规范每个 Plugin（诶，你这个 Plugin 好像不太行啊，你，代它去！）

还有四位顶层接口：

- **ConstructorInterceptPoint**：表示构造方法注入点
- **InstanceMethodsInterceptPoint**：表示实例方法注入点
- **StaticMethodsInterceptPoint**：表示静态方法注入点
- **InstanceMethodsAroundInterceptor**：拦截器本器

他们之间的具体关系如下图（图比较小，需要放大网页才能看得清）：




记住上面的问题和流程图，接下来我们去源码中看看，各个角色是如何进行沟通和交互的。

# 源码剖析

还是一样，我们从流程的入口着手进行分析，先看看 ByteBuddy 是怎么处理的吧。

## 入口 —— ByteBuddy

ByteBuddy 的执行在 SkyWalkingAgent 中的这段代码：

```
// type 我们要通过插件增强的类
// transform 定义字节码更改逻辑
// lintener 打印一些日志
agentBuilder.type(pluginFinder.buildMatch())
        .transform(new Transformer(pluginFinder))
        .with(AgentBuilder.RedefinitionStrategy.RETRANSFORMATION)
        .with(new RedefinitionListener())
        .with(new Listener())
        .installOn(instrumentation);
```

通过 type() 传入了 PluginFinder 中所有 Plugin 关注的需要增强的类，通过 transform() 传入遇到配的类之后的处理逻辑。

### buildMatch

将在下一小节进行讲解。

### transformer

Transformer 是 SkyWalkingAgent 中的一个内部类，当 ByteBuddy 遇到 type 中传入的目标之后就会调用 Transformer 的 transform 方法，我们来看看 transform 方法：

```
@Override
public DynamicType.Builder<?> transform(final DynamicType.Builder<?> builder,
                                        final TypeDescription typeDescription,
                                        final ClassLoader classLoader,
                                        final JavaModule javaModule,
                                        final ProtectionDomain protectionDomain) {
    LoadedLibraryCollector.registerURLClassLoader(classLoader);
    // 拿到所有可以应用于当前被拦截的这个类的插件
    List<AbstractClassEnhancePluginDefine> pluginDefines = pluginFinder.find(typeDescriptio
);
    if (pluginDefines.size() > 0) {
        DynamicType.Builder<?> newBuilder = builder;
        EnhanceContext context = new EnhanceContext();
        for (AbstractClassEnhancePluginDefine define : pluginDefines) {
            DynamicType.Builder<?> possibleNewBuilder = define.define(
                    typeDescription, newBuilder, classLoader, context);
            if (possibleNewBuilder != null) {
                // 后一个插件是基于前一个插件已经修改了的字节码再次进行修改
                newBuilder = possibleNewBuilder;
            }
        }
        if (context.isEnhanced()) {
            LOGGER.debug("Finish the prepare stage for {}.", typeDescription.getName());
        }

        return newBuilder;
    }
```

```
    LOGGER.debug("Matched class {}, but ignore by finding mechanism.", typeDescription.getT
peName());
    return builder;
}
```

可以看到 transform 中先通过 pluginFinder 拿到了目标对应的 plugin（typeDescription 可以理解目标的唯一标识，里面记录了很多信息），然后会调用 plugin 的 define() 方法，其实这个方法是在bstractClassEnhancePluginDefine 中实现的，具体的逻辑我们将在 AbstractClassEnhancePluginDfine 讲解中看到它，这个方法的主要作用就是判断这个 plugin 是否生效，并且将 plugin 构建成 BytBuddy 所需要的 DynamicType.Builder 对象。

从代码中的 for 循环就可以看出来，最终的注入是链式的，还记得前面的问题吗，如果碰到多个 plugi 对一个目标进行加强应该怎么处理，这里就给出了答案，后面的加强会基于前面加强完成之后的代码续进行，最后构造出一个最终的 builder 返回给 ByteBuddy。

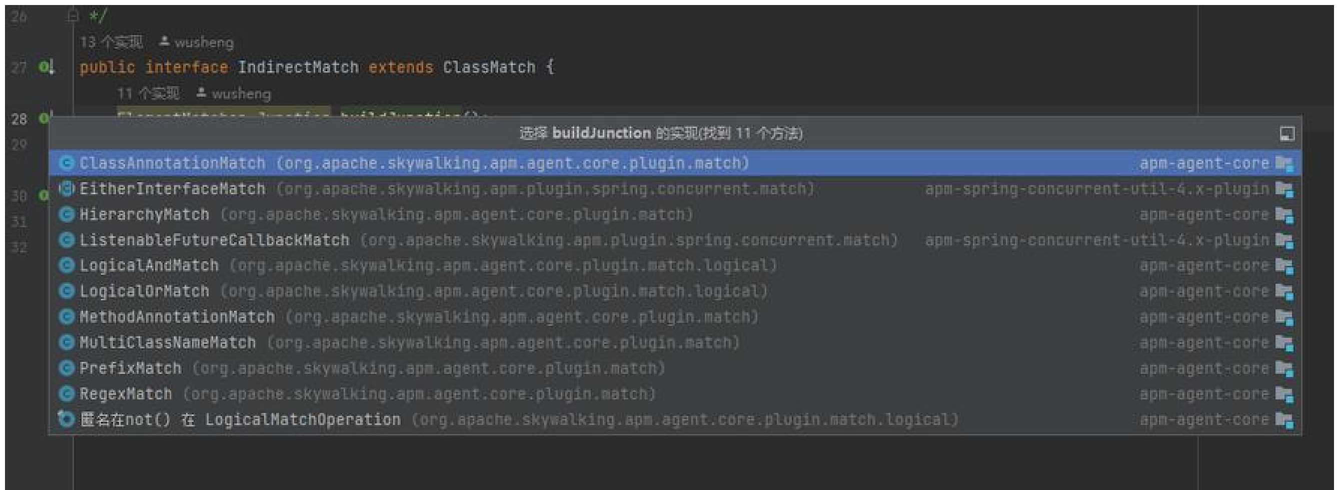
## Plugin 管理 —— PluginFinder

上一篇我们学习 PluginFinder 的时候说到，它负责了 Plugin 的分类，将 Plugin 分为了三类以便后处理，这里就是用它来后续处理的地方，现在它又多了两个职责：1.负责找到所有 plugin 的目标 match；2.负责按照给定的 typeDescription 找到它对应的 plugin 列表；让我们看看这两个方法是如何做的。

### buildMatch

```
// 把所有需要增强的类构造为ElementMatcher
public ElementMatcher<? super TypeDescription> buildMatch() {
    ElementMatcher.Junction judge = new AbstractJunction<NamedElement>() {
        @Override
        public boolean matches(NamedElement target) {
            return nameMatchDefine.containsKey(target.getActualName());
        }
    };
    judge = judge.and(not(isInterface()));
    for (AbstractClassEnhancePluginDefine define : signatureMatchDefine) {
        ClassMatch match = define.enhanceClass();
        if (match instanceof IndirectMatch) {
            judge = judge.or(((IndirectMatch) match).buildJunction());
        }
    }
    return new ProtectiveShieldMatcher(judge);
}
```

buildMatch 主要是构建 ByteBuddy 的 ElementMatcher.Junction，根据 PluginFinder 之前存储分类，对 Plugin 进行了不同的处理，nameMatchDfine 是直接根据名称进行匹配，直接通过 contain进行判断，signatureMatchDefine 是通过间接的方式来进行匹配，如下图，可以通过这些 match进行匹配。




### find

```
// typeDescription 是对一个类型的完整描述，其中包含了这个类的全类名
public List<AbstractClassEnhancePluginDefine> find(TypeDescription typeDescription) {
    List<AbstractClassEnhancePluginDefine> matchedPlugins = new LinkedList<AbstractClass
nhancePluginDefine>();
    String typeName = typeDescription.getTypeName();
    if (nameMatchDefine.containsKey(typeName)) {
        matchedPlugins.addAll(nameMatchDefine.get(typeName));
    }

    for (AbstractClassEnhancePluginDefine pluginDefine : signatureMatchDefine) {
        IndirectMatch match = (IndirectMatch) pluginDefine.enhanceClass();
        if (match.isMatch(typeDescription)) {
            matchedPlugins.add(pluginDefine);
        }
    }

    return matchedPlugins;
}
```

find 和 buildMatch 的逻辑是类似的，只不过 buildMatch 是交给 ByteBuddy 进行处理，而 find 是 luginFinder 自己寻找 plugin 中满足条件的 plugin，所以逻辑可以参照 buildMatch 的讲解。

## Plugin 的 Leader —— AbstractClassEnhancePluginDefine

AbstractClassEnhancePluginDefine 作为 plugin 的顶层类，它承受了太多，它不仅要确定所有的 pl gin 是否可以生效，还需要定义 plugin 的具体的模板方法，以便继承的 plugin 可以很方便的实现功 ；所以为了更方便的进行管理，它引入了两个小助手：

- **WitnessFinder**：负责确认 plugin 是否生效
- **ClassEnhancePluginDefine**：负责实现不同位置注入（构造方法注入、实例方法注入和静态方法 入）的公共逻辑

### define

我们还是先从 define() 方法的具体逻辑入手，来看看 AbstractClassEnhancePluginDefine 是如何为

原文链接：SkyWalking Java Agent 插件详解

lugin 排忧解难的。

```
public DynamicType.Builder<?> define(TypeDescription typeDescription, DynamicType.Builde
<?> builder,
    ClassLoader classLoader, EnhanceContext context) throws PluginException {
    String interceptorDefineClassName = this.getClass().getName();
    // 1.拿到被拦截到这个类的全类名用来打印log
    String transformClassName = typeDescription.getTypeName();
    if (StringUtil.isEmpty(transformClassName)) {
        LOGGER.warn("classname of being intercepted is not defined by {}.", interceptorDefineCl
ssName);
        return null;
    }

    LOGGER.debug("prepare to enhance class {} by {}.", transformClassName, interceptorDefin
ClassName);
    WitnessFinder finder = WitnessFinder.INSTANCE;
    /**
     * find witness classes for enhance class
     * 2.判断 plugin 是否生效
     */
    String[] witnessClasses = witnessClasses();
    if (witnessClasses != null) {
        for (String witnessClass : witnessClasses) {
            if (!finder.exist(witnessClass, classLoader)) {
                LOGGER.warn("enhance class {} by plugin {} is not activated. Witness class {} does no
 exist.", transformClassName, interceptorDefineClassName, witnessClass);
                return null;
            }
        }
    }
    List<WitnessMethod> witnessMethods = witnessMethods();
    if (!CollectionUtil.isEmpty(witnessMethods)) {
        for (WitnessMethod witnessMethod : witnessMethods) {
            if (!finder.exist(witnessMethod, classLoader)) {
                LOGGER.warn("enhance class {} by plugin {} is not activated. Witness method {} does
not exist.", transformClassName, interceptorDefineClassName, witnessMethod);
                return null;
            }
        }
    }

    /**
     * find origin class source code for interceptor
     * 3.执行拦截器注入，并返回DynamicType.Builder
     */
    DynamicType.Builder<?> newClassBuilder = this.enhance(typeDescription, builder, classLo
der, context);

    context.initializationStageCompleted();
    LOGGER.debug("enhance class {} by {} completely.", transformClassName, interceptorDefin
ClassName);

    return newClassBuilder;
```

```
}

protected DynamicType.Builder<?> enhance(TypeDescription typeDescription, DynamicType.
uilder<?> newClassBuilder,
                                         ClassLoader classLoader, EnhanceContext context) throws PluginE
ception {
    newClassBuilder = this.enhanceClass(typeDescription, newClassBuilder, classLoader);

    newClassBuilder = this.enhanceInstance(typeDescription, newClassBuilder, classLoader, con
ext);

    return newClassBuilder;
}

protected abstract DynamicType.Builder<?> enhanceInstance(TypeDescription typeDescripti
n,
                                                  DynamicType.Builder<?> newClassBuilder, ClassLoader class
oader,
                                                  EnhanceContext context) throws PluginException;

protected abstract DynamicType.Builder<?> enhanceClass(TypeDescription typeDescription,
ynamicType.Builder<?> newClassBuilder,
                                               ClassLoader classLoader) throws PluginException;
```

define() 方法还是蛮长的，但是逻辑却很清晰，就做了三件事：1.打印 log；2.判断 plugin 是否生效
3.如果 plugin 生效那么执行注入逻辑，并返回 DynamicType.Builder。

可以看到判断 plugin 是否生效的逻辑是交给 WitnessFinder 来完成的，而具体的注入逻辑则是调用两个抽象方法 enhanceInstance 和 enhanceClass，这两个方法其实是在 ClassEnhancePluginDefin 中进行实现的，那么让我们继续看看它们俩是如何完成任务的吧。

## plugin 的开关 —— witness

还记得前面我们提过的问题吗，SkyWalking Java Agent 是如何保证不同的目标框架版本来注入不同逻辑的吗，这里就是解答的地方；SkyWalking Java Agent 用了一个名叫 witness 机制来达成这个的。

witness 具体逻辑其实是根据框架不同的特点作为它的唯一标识（比如这个框架只有这个版本有某个或者某个方法），根据定义不同的标识类和标识方法就可以确认运行 SkyWalking Java Agent 的程中的框架版本是多少，然后就可以注入对应的逻辑了。

```
if (witnessClasses != null) {
    for (String witnessClass : witnessClasses) {
        if (!finder.exist(witnessClass, classLoader)) {
            LOGGER.warn("enhance class {} by plugin {} is not activated. Witness class {} does not
xist.", transformClassName, interceptorDefineClassName, witnessClass);
                return null;
        }
    }
}
List<WitnessMethod> witnessMethods = witnessMethods();
if (!CollectionUtil.isEmpty(witnessMethods)) {
    for (WitnessMethod witnessMethod : witnessMethods) {
        if (!finder.exist(witnessMethod, classLoader)) {
```

```
            LOGGER.warn("enhance class {} by plugin {} is not activated. Witness method {} does n
t exist.", transformClassName, interceptorDefineClassName, witnessMethod);
            return null;
        }
    }
}
```

define() 方法中的这段代码就是执行 witness 判断，可以看到所有的条件必须都满足，才会进行下一的处理，如果有一个条件不满足，就会直接返回 null。

### 脏活累活全我来 —— ClassEnhancePluginDefine

如果 plugin 满足条件可以注入之后，就来到 ClassEnhancePluginDefine 执行注入的环节，这里就贴 ClassEnhancePluginDefine 的代码了，具体可以自行去看 ClassEnhancePluginDefine 或 ClassEhancePluginDefineV2 的源代码；这里就只讲解一下这两个方法的作用，因为方法的实现大部分都调用 ByteBuddy 的方法，所以这里只描述一下方法的逻辑。

enhanceClass() 负责处理静态方法的注入，主要的逻辑是将 plugin 对象中的 StaticMethodsIntercetV2Point[] 数组拿出来进行遍历，StaticMethodsInterceptV2Point 中存入了方法的 matcher（用匹配对应方法）、拦截器和是否需要覆盖参数这个配置项，然后调用 ByteBuddy 的方法进行注入。

enhanceInstance() 负责处理构造方法和实例方法，和 enhanceClass() 方法类似，只不过存储构造法的类是 ConstructorInterceptPoint，存储实例方法的类是 InstanceMethodsInterceptV2Point，续的判断逻辑与 enhanceClass() 是一致的。

# 总结

Plugin 的具体设计和实现已经介绍完毕了，大家可以看看之前我们提出的问题有没有得到解决呢；最可以结合后面的源码讲解结合之前的流程图看看是否是这个样子的，这样有助于记忆，然后对于更细的代码逻辑，可以自行去查看 ClassEnhancePluginDefine、Inter系列类（InstMethodsInterV2、InsMethodsInterV2WithOverrideArgs、StaticMethodsInterV2 和 StaticMethodsInterV2WithOverrdeArgs，它们是 byte-buddy 和 sky-walking plugin 的桥梁）和 SkyWalking Java Agent 定义的套 Matcher 逻辑。