《31 讲带你搞懂 SkyWalking》

徐郡明　前搜狗资深技术专家、源码剖析系列畅销书作者

第09讲：SkyWalking Agent 启动流程剖析
领略微内核架构之美

Skywalking Agent 启动流程剖析

SkyWalking Agent 是通过 Java Agent 的方式随应用程序一起启动

然后通过 Byte Buddy 库动态插入埋点收集 Trace 信息

本课时：

- 深入研究 SkyWalking Agent 的架构、原理以及具体实现
- 深入分析 Tomcat、Dubbo、MySQL 等常用的插件

微内核架构

SkyWalking Agent 采用了微内核架构（Microkernel Architecture）

微内核架构也被称为插件化架构（Plug-in Architecture）

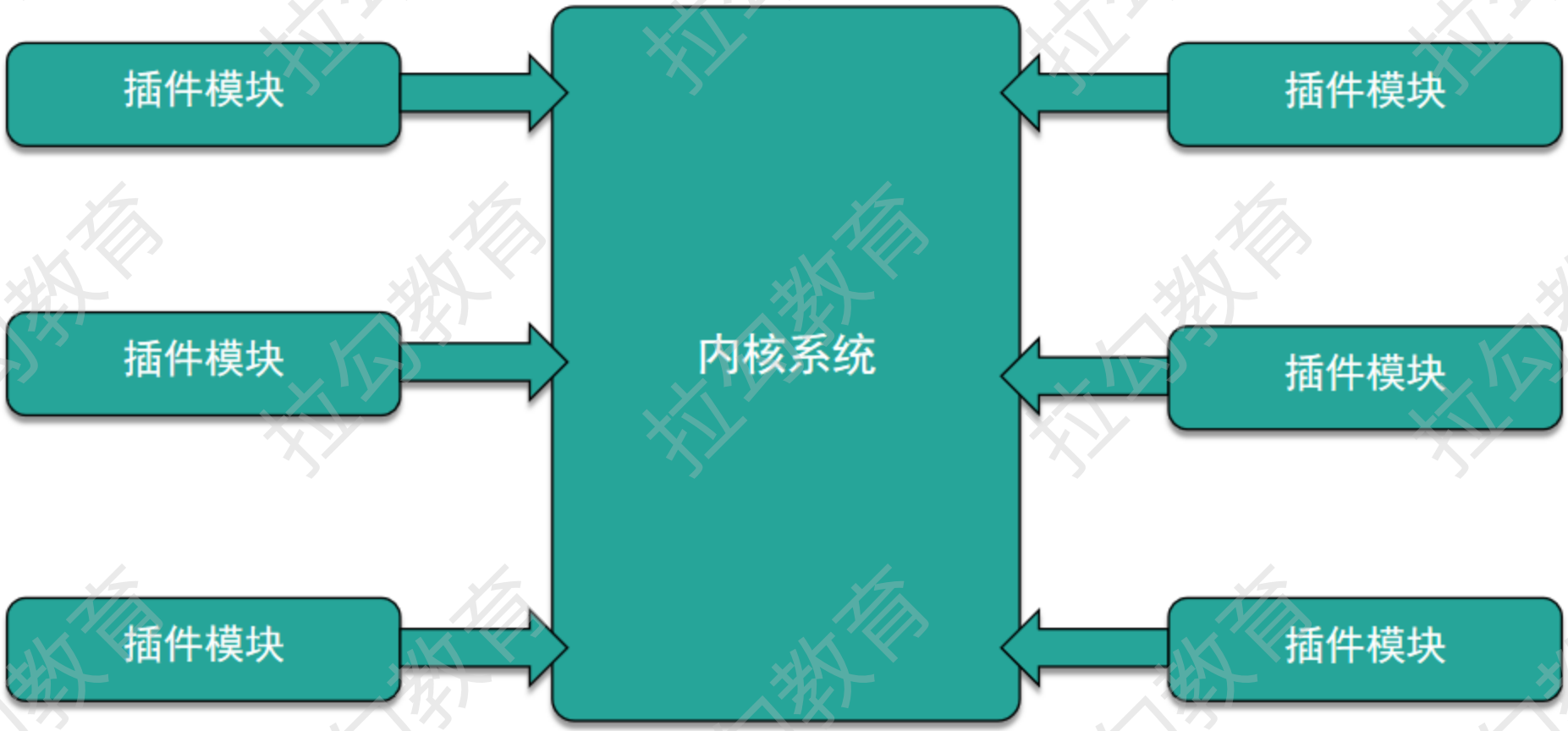
是一种面向功能进行拆分的可扩展性架构，在基于产品的应用中通常会使用微内核架构

例如：

IDEA、Eclipse 这类 IDE 开发工具，内核都是非常精简的

对 Maven、Gradle 等新功能的支持都是以插件的形式增加的

微内核架构



微内核架构

所有插件会由内核系统统一接入和管理：

- 内核系统必须知道要加载哪些插件

  一般会通过配置文件或是扫描 ClassPath 的方式（例如前文介绍的 SPI 技术）确定待加载的插件

- 内核系统还需要了解如何使用这些插件，微内核架构中需要定义一套插件的规范

  内核系统会按照统一的方式初始化、启动这些插件

- 内核需要提供一套规则，识别插件消息并能正确的在插件之间转发消息，成为插件消息的中转站

微内核架构

- 测试成本下降

  从软件工程的角度看，微内核架构将变化的部分和不变的部分拆分

  降低了测试的成本，符合设计模式中的开放封闭原则

- 稳定性

  由于每个插件模块相对独立，即使其中一个插件有问题，也可以保证内核系统以及其他插件的稳定性

- 可扩展性

  在增加新功能或接入新业务的时候，只需要新增相应插件模块即可

  在进行历史功能下线时，也只需删除相应插件模块即可

微内核架构

SkyWalking Agent 是微内核架构的一种落地方式

apm-agent-core 模块对应微内核架构中的内核系统

apm-sdk-plugin 模块中的各个子模块都是微内核架构中的插件模块

SkyWalking Agent 启动流程概述

在搭建 SkyWalking 源码环境的最后

我们尝试 Debug 了一下 SkyWalking Agent 的源码

其入口是 apm-agent 模块中 SkyWalkingAgent 类的 premain() 方法

SkyWalking Agent 启动流程概述

1. 初始化配置信息

   该步骤中会加载 agent.config 配置文件

   其中会检测 Java Agent 参数以及环境变量是否覆盖了相应配置项
2. 查找并解析 skywalking-plugin.def 插件文件
3. AgentClassLoader 加载插件
4. PluginFinder 对插件进行分类管理
5. 使用 Byte Buddy 库创建 AgentBuilder

   根据已加载的插件动态增强目标类，插入埋点逻辑
6. 使用 JDK SPI 加载并启动 BootService 服务
7. 添加一个 JVM 钩子，在 JVM 退出时关闭所有 BootService 服务

SkyWalking Agent 启动流程概述

拉勾教育

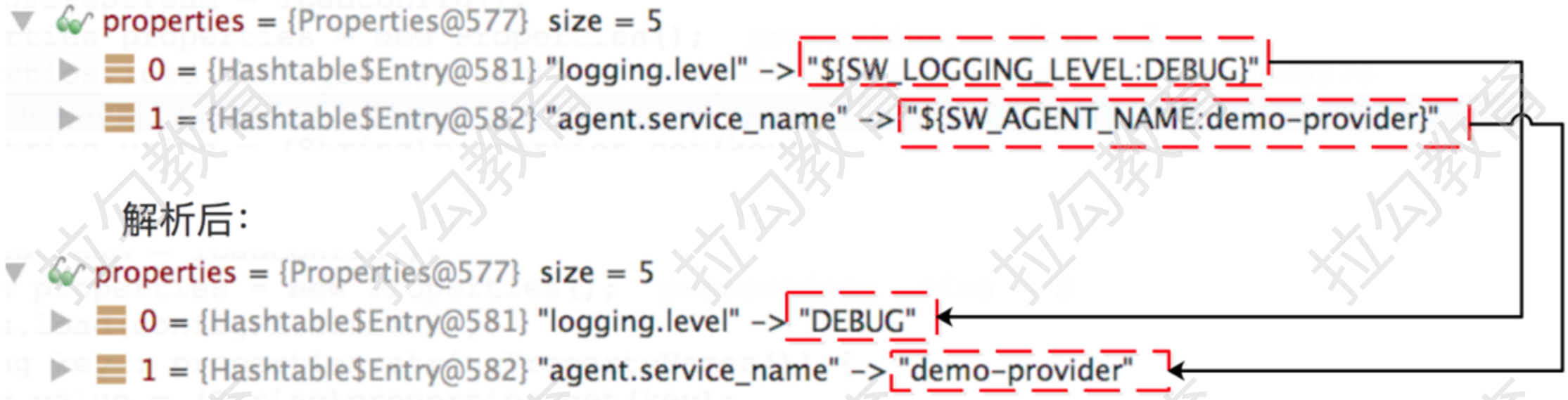
```
public static void premain(String agentArgs,
    Instrumentation instrumentation) throws PluginException {
  // 步骤1、初始化配置信息
  SnifferConfigInitializer.initialize(agentArgs);
  // 步骤2~4、查找并解析skywalking-plugin.def插件文件；
  // AgentClassLoader加载插件类并进行实例化；PluginFinder提供插件匹配的功能
  final PluginFinder pluginFinder = new PluginFinder(
    new PluginBootstrap().loadPlugins());
  // 步骤5、使用 Byte Buddy 库创建 AgentBuilder
  final ByteBuddy byteBuddy = new ByteBuddy()
    .with(TypeValidation.of(Config.Agent.IS_OPEN_DEBUGGING_CLASS));
  new AgentBuilder.Default(byteBuddy)...installOn(instrumentation);
  // 这里省略创建 AgentBuilder的具体代码，后面展开详细说
```

SkyWalking Agent 启动流程概述


```
        new PluginBootstrap().loadPlugins();
        // 步骤5、使用 Byte Buddy 库创建 AgentBuilder
        final ByteBuddy byteBuddy = new ByteBuddy()
                .with(TypeValidation.of(Config.Agent.IS_OPEN_DEBUGGING_CLASS));
        new AgentBuilder.Default(byteBuddy)...installOn(instrumentation);
        // 这里省略创建 AgentBuilder的具体代码，后面展开详细说
        // 步骤6、使用 JDK SPI加载的方式并启动 BootService 服务
        ServiceManager.INSTANCE.boot();
        // 步骤7、添加一个JVM钩子
        Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
            public void run() { ServiceManager.INSTANCE.shutdown(); }
        }, "skywalking service shutdown thread"));
    }
```

初始化配置

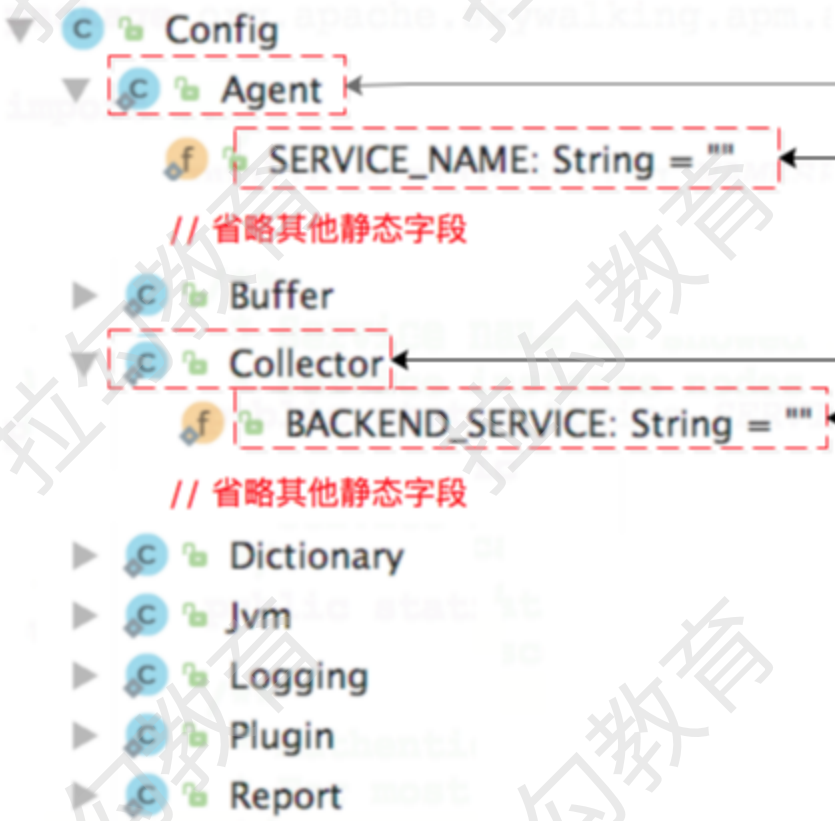
解析前：



解析后：

初始化配置

Config 类中的静态字段：



agent.config 配置文件：

```
# The service name in UI
agent.service_name=${SW_AGENT_NAME:demo-provider}

# Backend service addresses.
collector.backend_service=${SW_AGENT_COLLECTOR_BA
```

AgentClassLoader

SkyWalking Agent 加载插件时

使用自定义的 ClassLoader —— AgentClassLoader

目的是不在应用的 Classpath 中引入 SkyWalking 的插件 jar 包

这样就可以让应用无依赖、无感知的插件

AgentClassLoader

```
private static void tryRegisterAsParallelCapable() {
    Method[] methods = ClassLoader.class.getDeclaredMethods();
    for (int i = 0; i < methods.length; i++) {
        Method method = methods[i];
        String methodName = method.getName();
        // 查找ClassLoader 中的 registerAsParallelCapable()静态方
法
        if
("registerAsParallelCapable".equalsIgnoreCase(methodName))
        {
            method.setAccessible(true);
            method.invoke(null); // 调用registerAsParallelCapable()
方法
            return;
        }
    }
```

AgentClassLoader

Java 7 之后提供了两种加锁模式：

- 串行模式下，锁的对象是还是 ClassLoader 本身，和 Java 6 里面的行为一样
- 另外一种就是调用 registerAsParallelCapable() 方法之后，开启的并行加载模式

  在并行模式下加载类时，会按照 classname 去获取锁

AgentClassLoader

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException{
    // getClassLoadingLock() 方法会返回加锁的对象
    synchronized (getClassLoadingLock(name)) {
        ... ... // 加载指定类，具体加载细节不展开介绍
    }
}
```

AgentClassLoader

```
protected Object getClassLoadingLock(String className) {
    Object lock = this;
    if (parallelLockMap != null) { //检测是否开启了并行加载功能
      Object newLock = new Object();
      // 若开启了并行加载，则一个className对应一把锁；否则还是只
      // 对当前ClassLoader进行加锁
      lock = parallelLockMap.putIfAbsent(className, newLock);
      if (lock == null) {
        lock = newLock;
      }
    }
    return lock;
  }
```

AgentClassLoader


```
private List<File> classpath;

public AgentClassLoader(ClassLoader parent) {
    super(parent); // 双亲委派机制
    // 获取 skywalking-agent.jar所在的目录
    File agentDictionary = AgentPackagePath.getPath();
    classpath = new LinkedList<File>();
    // 初始化 classpathé 集合，指向了skywalking-agent.jar包同目录的两个
目录
    classpath.add(new File(agentDictionary, "plugins"));
    classpath.add(new File(agentDictionary, "activations"));
}
```

AgentClassLoader

```
// 在下面的getAllJars()方法中会扫描全部jar文件，并缓存到
// allJars字段(List<Jar>类型)中，后续再次扫描时会重用该缓存
private List<Jar> allJars;

protected Class<?> findClass(String name) {
    List<Jar> allJars = getAllJars(); // 扫描过程比较简单，不在
展开介绍
    String path = name.replace('.', '/').concat(".class");
    for (Jar jar : allJars) { // 扫描所有jar包，查找类文件
        JarEntry entry = jar.jarFile.getJarEntry(path);
        if (entry != null) {
            URL classFileUrl = new URL("jar:file:" +
                jar.sourceFile.getAbsolutePath() + "!/" + path);
            byte[] data = ...;//省略读取“.class”文件的逻辑
            // 加载类文件内容，创建相应的Class对象
            return defineClass(name, data, 0, data.length);
        }
    } // 类查找失败，直接抛出异常
    throw new ClassNotFoundException("Can't find " + name);
}
```

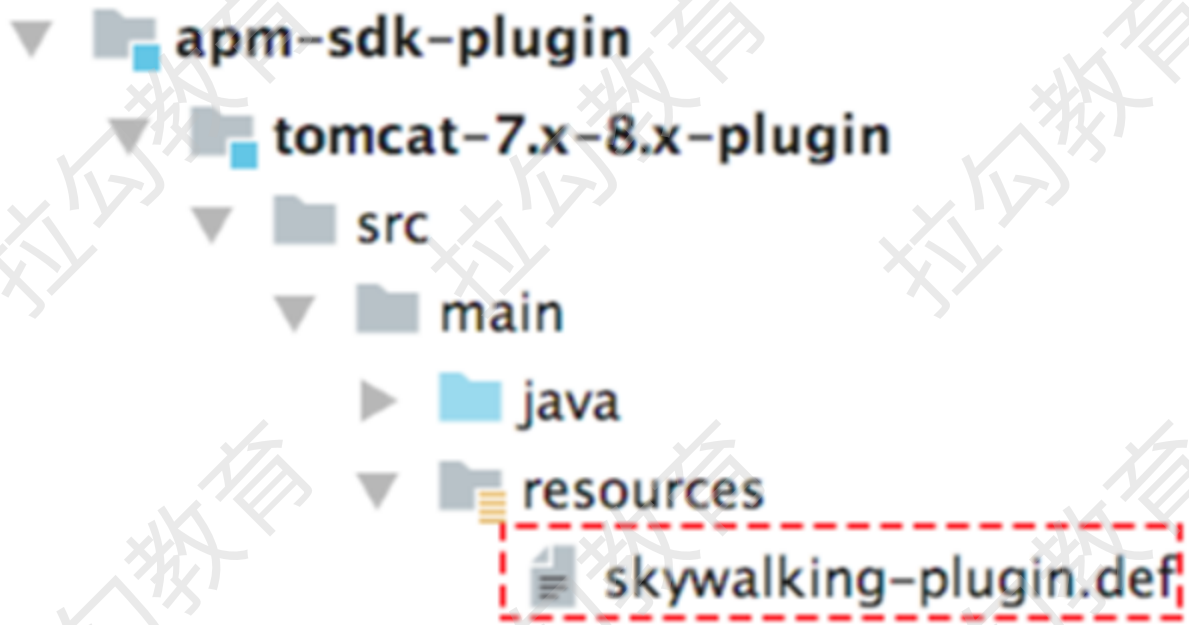
AgentClassLoader

```
private static AgentClassLoader DEFAULT_LOADER;
```

注意 AgentClassLoader 并不是单例

解析插件定义

每个 Agent 插件中都会定义一个 skywalking-plugin.def 文件



解析插件定义


```
tomcat-7.x/8.x=org.apache.skywalking.apm.plugin.tomcat78x.define
.TomcatInstrumentation

tomcat-7.x/8.x=org.apache.skywalking.apm.plugin.tomcat78x.define
.ApplicationDispatcherInstrumentation
```

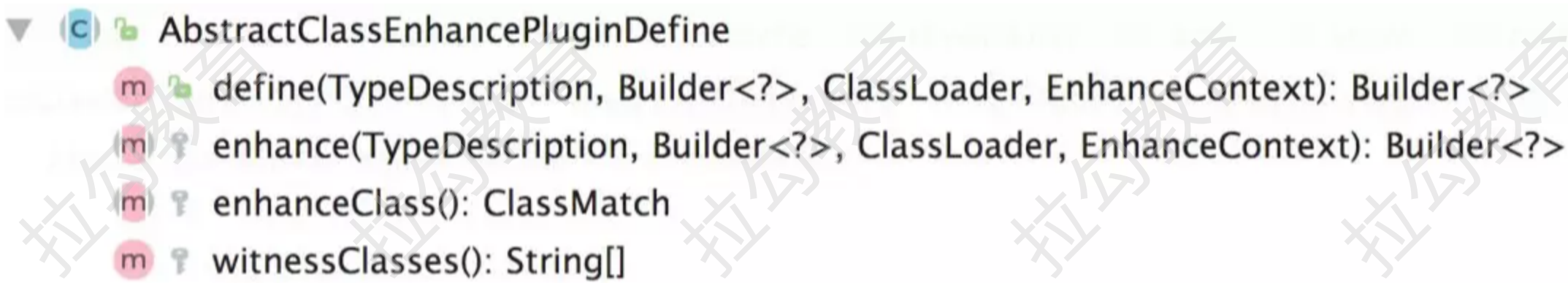
AbstractClassEnhancePluginDefine


```
// 插件名称，以 tomcat-7.x-8.x-plugin 插件第一行为例，就是tomcat-7.x/8.x
private String name;
// 插件类，对应上例中的org.apache.skywalking.apm.plugin.tomcat78x.define
// .TomcatInstrumentation
private String defineClass;
```

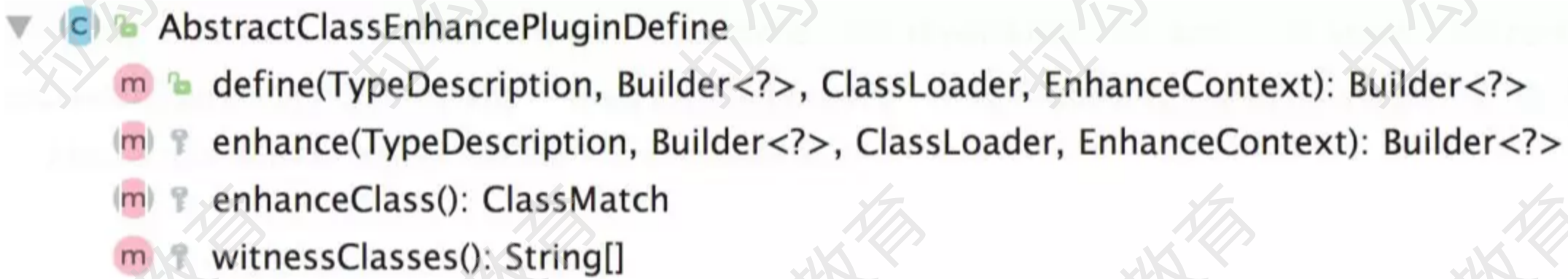
AbstractClassEnhancePluginDefine

```
for (PluginDefine pluginDefine : pluginClassList) {
    // 注意，这里使用类加载器是默认的AgentClassLoader实例
    AbstractClassEnhancePluginDefine plugin =
        (AbstractClassEnhancePluginDefine)
            Class.forName(pluginDefine.getDefineClass(), true,
                AgentClassLoader.getDefault()).newInstance();
    plugins.add(plugin); // 记录AbstractClassEnhancePluginDefine
对象
}
```

AbstractClassEnhancePluginDefine



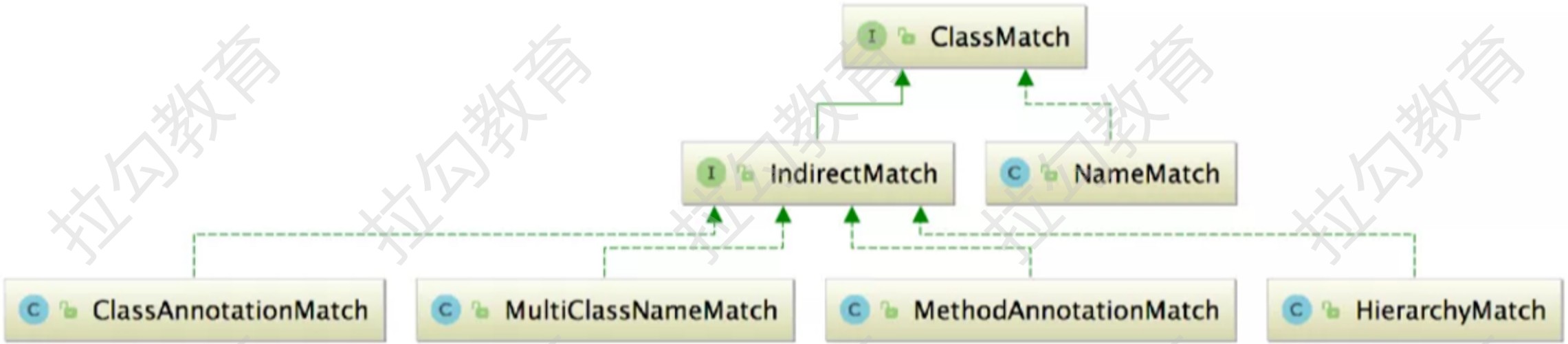
AbstractClassEnhancePluginDefine



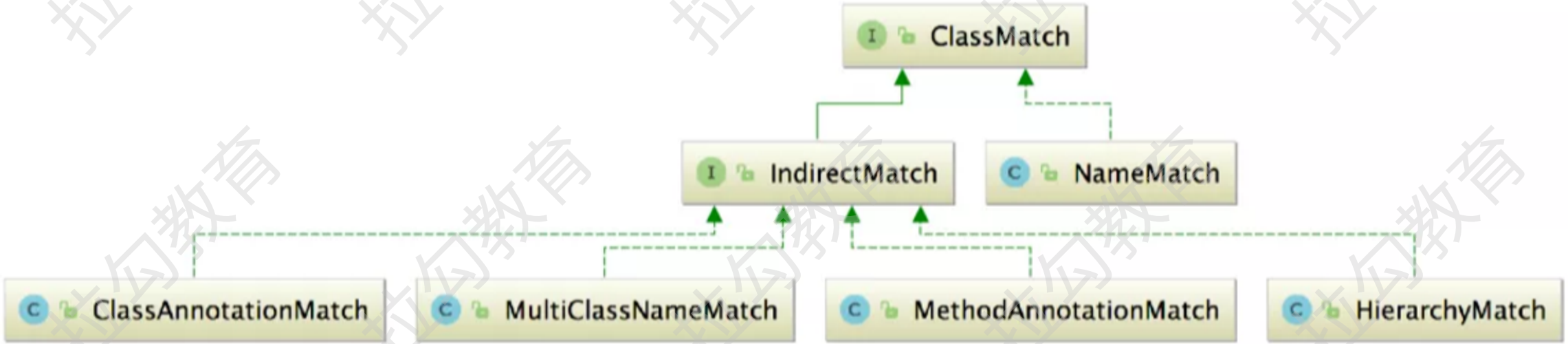
- enhanceClass() 方法：返回的 ClassMatch，用于匹配当前插件要增强的目标类
- define() 方法：插件类增强逻辑的入口，底层会调用下面的 enhance() 方法和 witnessClass() 方法
- enhance() 方法：真正执行增强逻辑的地方
- witnessClass() 方法：一个开源组件可能有多个版本，插件会通过该方法识别组件的不同版本

防止对不兼容的版本进行增强

AbstractClassEnhancePluginDefine



AbstractClassEnhancePluginDefine

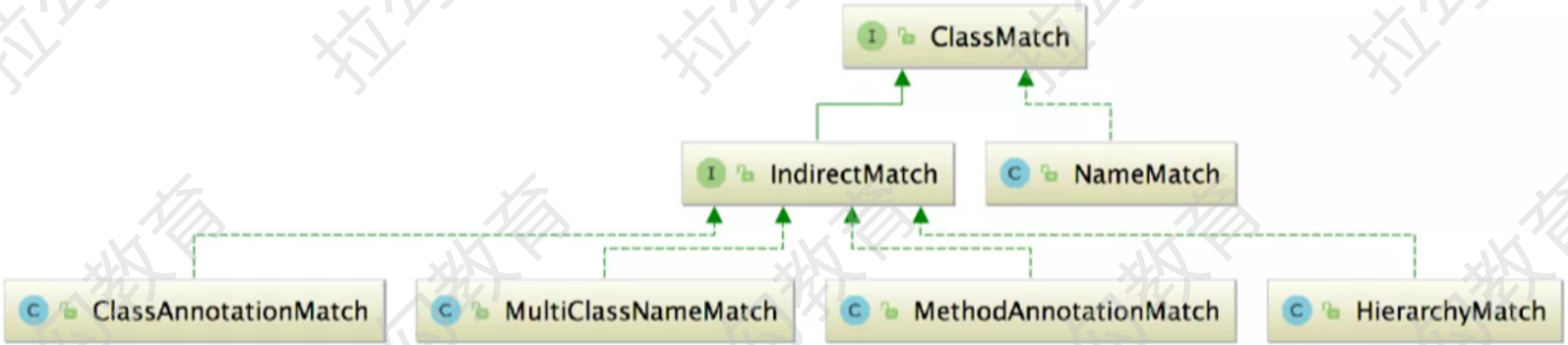


- NameMatch：根据其 className 字段（String 类型）匹配目标类的名称
- IndirectMatch：子接口中定义了两个方法

AbstractClassEnhancePluginDefine


```
// Junction是Byte Buddy中的类，可以通过and、or等操作串联多个ElementMatcher
// 进行匹配
ElementMatcher.Junction buildJunction();
// 用于检测传入的类型是否匹配该Match
boolean isMatch(TypeDescription typeDescription);
```

AbstractClassEnhancePluginDefine



- MultiClassNameMatch：其中会指定一个 matchClassNames 集合，该集合内的类即为目标类
- ClassAnnotationMatch：根据标注在类上的注解匹配目标类
- MethodAnnotationMatch：根据标注在方法上的注解匹配目标类
- HierarchyMatch：根据父类或是接口匹配目标类

PluginFinder

PluginFinder

是 AbstractClassEnhancePluginDefine 查找器

可根据给定的类查找用于增强的 AbstractClassEnhancePluginDefine 集合

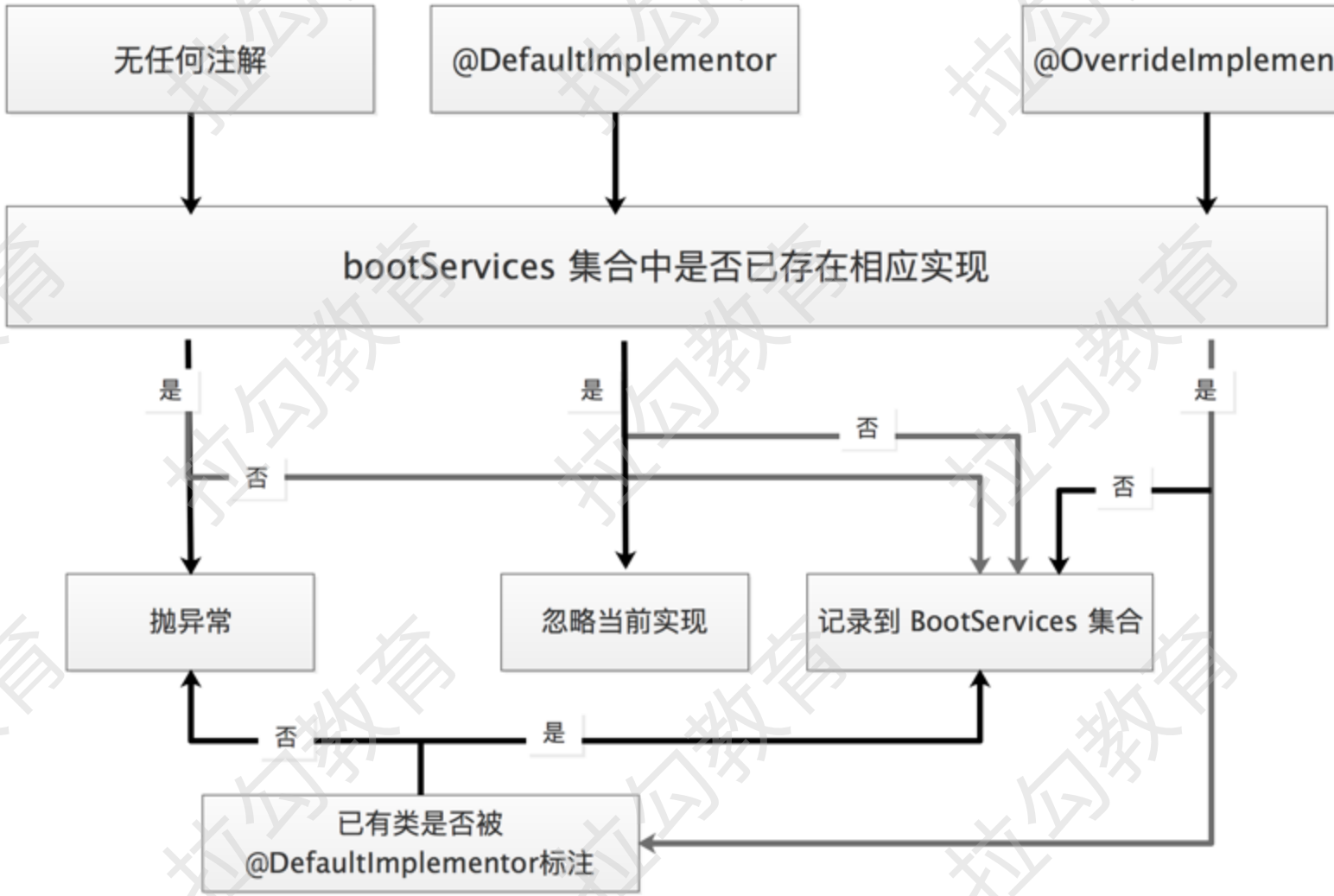
加载 BootService

@DefaultImplementor 和 @OverrideImplementor 注解进行处理：

- @DefaultImplementor 注解用于标识 BootService 接口的默认实现
- @OverrideImplementor 注解用于覆盖默认 BootService 实现

  通过其 value 字段指定要覆盖的默认实现

加载 BootService



总结

- 重点介绍了 SkyWalking Agent 启动核心流程的实现
- 深入分析了 Skywalking Agent 配置信息的初始化、插件加载原理

  AgentBuilder 如何与插件类配合增强目标类、BootService 的加载流程

Next：第10讲《深入剖析 Agent 插件原理，无侵入性埋点》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U