# 第三课：WEB入口与拦截与Service拦截

**课程概要：**

1. Service 层拦截
2. WEB 入口拦截
3. Agent 采集器整合

## 一、Service 层拦截

---

### 1. 确定采集目标

采集目标即找出哪些是需要监控的方法 。这里采取的办法是通过参数配置来实现采集的目标。通过通配符正则匹配的方式配置需要采集类。方法采集范围是 当前类所有的public方法并且是非抽像，非静态，非本地（native）方法。

为降低使用成本，这里的配置采用简单通配符的方式进行匹配，规则如下：

- * 表示多个任意字符
- ? 表示单个任意字符
- & 用于分割多个匹配语句

示例：

```
#匹配Server或Service的所有类
com.tuling.server.*Server&com.tuling.server.*Service
```

### 2.实现数据采集并打印至日志

流程如下：

1. 编写参数解析方法
2. 编写监控起始方法
3. 编写监控结束方法
4. 基于javassist 实现插桩

## 二、WEB 入口拦截

---

### 1、采集要求

1. 数据需求
   a. 采集url、参数、cookie、hand、执行时间、异常信息
2. 可用性需求
   a. 项目无关：不限制使用的项目
   b. 架构无关：无论采用spring mvc 或structs 或servlet 都应该支持
   c. 容器无关：无论采用Tomcat或jetty 、spring boot 都应该支持

## 2、埋点目标

需要将点埋在哪里 才能满足上述两大需求呢？可选有以下三方案：

| 方案 | 优点 | 缺点 |
| --- | --- | --- |
| 应用层Control类 | 简单，风险因素低 | 判别成本高，有局限性，只能根据HttpServlet 子类或@RequestMapping进行识别。 |
| DispatcherServlet.doDispatch | 简单，适应性强 | 1、只能针对spring mvc 项目 2、spring boot 项目不支持 |
| HttpServlet.service | 适应性强，与应用层和框架无关 | 1、不同的容器ClassPath不一样，存在兼容性问题。 2、存在风险，几乎所有请求都会经过此方法 3、业务异常无法捕获 |

明显前面两个无法满足项目无关与架构无关两个需求，在此不在叙述。关于Servlet 它是j2ee的标准，任何框架与容器都要基于此实现WEB服务，选用它可在一定程度上满足上述需求。（如果项目采用 netty 或java se 内置 非标准servlet 则无法采集）

## 3、实现 servlet 采集

实现过程如下：

1. 添加servlet-api依赖
2. 编写buildWebMonitor() 生成插桩字，已 HttpServlet#service 方法
3. 编写begin 与end 方法
4. 编写 WebTraceInfo 实体类，用于存放 http 数据
5. 启动jetty 测试服务
6. 基于Tomcat容器进行测试

### 添加 servlet-api 依赖

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
```

### 编写buildWebMonitor() 生成插桩字，已 HttpServlet#service 方法

```
private static byte[] buildWebMonitor(ClassLoader loader, String name)
throws Exception {
        ClassPool pool = new ClassPool();
        pool.insertClassPath(new LoaderClassPath(loader));
        CtClass ctClass = pool.get(name);
        CtMethod ctMethod = ctClass.getDeclaredMethod("service",
                pool.get(new String[]
{"javax.servlet.http.HttpServletRequest",
                        "javax.servlet.http.HttpServletResponse"}));
        CtMethod copyMethod = CtNewMethod.copy(ctMethod, ctClass, new
ClassMap());
        ctMethod.setName("service$agent");
        copyMethod.setBody(" {\n" +
                "       Object traceBean=
com.tuling.agent.WebAgent.begin($args);\n" +
                "            try {\n" +
                "                service$agent($$);\n" +
                "            } finally {\n" +
                "
com.tuling.agent.WebAgent.end(traceBean);\n" +
                "            }\n" +
                "        }");
        ctClass.addMethod(copyMethod);
        return ctClass.toBytecode();
    }
```

**编写begin 与end 方法 用于实际采集数据**

```
public static Object begin(Object[] args) {
        WebTraceInfo trace = new WebTraceInfo();
        HttpServletRequest request = (HttpServletRequest) args[0];
        trace.setParams(request.getParameterMap());
        trace.setCookie(request.getCookies());
        trace.setUrl(request.getRequestURI());
        trace.setBegin(System.currentTimeMillis());
        return trace;
}
public static void end(Object webTraceInfo) {
        WebTraceInfo trace = (WebTraceInfo) webTraceInfo;
        trace.setUseTime(System.currentTimeMillis() - trace.getBegin());
        System.out.println(trace);
}
```

实际测试过程中遇到的问题：找不到javassist 类异常。原因是WEB容器启动时找不到javassist.jar ，解决办法：设置 <Boot-Class-Path> 参数，措定依赖包路径，或者直接在打包的时候将javassist 包一起打进去，通过shade插件即可实现：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
```

```
    <artifactId>maven-shade-plugin</artifactId>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <transformers>
                    <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTr
ansformer">
                    </transformer>
                </transformers>
            </configuration>
        </execution>
    </executions>
</plugin>
```

**启动jetty WEB服务进行测试**

添加jetty 依赖

```
<dependency>
    <groupId>org.eclipse.jetty.aggregate</groupId>
    <artifactId>jetty-all</artifactId>
    <version>9.2.11.v20150529</version>
    <scope>test</scope>
</dependency>
```

jetty 启动方法

```
 public static void main(String[] args) {
        // junit web test
        try {
            Server server = new Server(8008);//设置端口号
            WebAppContext context = new WebAppContext();
            context.setContextPath("/");//访问路径

context.setResourceBase(WebAgentTest.class.getResource("/webapp/").getPath
());//路径

context.setDescriptor(WebAgentTest.class.getResource("/webapp/WEB-
INF/web.xml").getPath());//读取web.xml文件
            server.setHandler(context);
            server.start();
            System.out.println("启动成功：端口号：8008");
            server.join();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

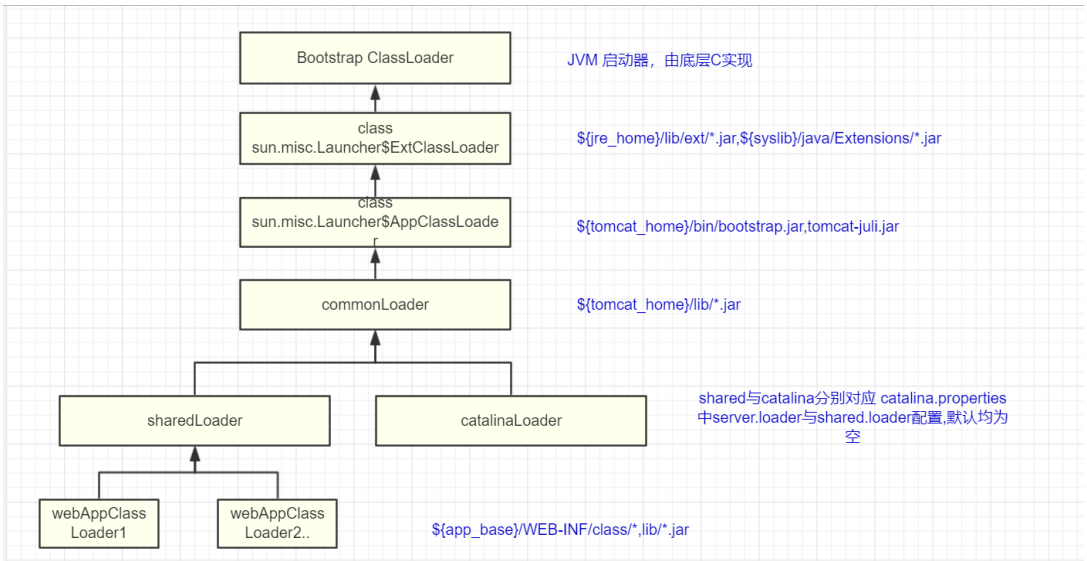
添加jvm 参数 并执行启动方法：

```
-javaagent:tuling-agent-1.0-SNAPSHOT.jar
```

**在Tomcat 启动下会遇到的问题：**


**问题原因分析：**

如下图所示Tomcat 容器类装载器是多层次，agnet.jar 在Launcher$AppClassLoader 这一层装载，而servlet-api.jar 在commonLoader 这一层，即Launcher的子装载器。基于ClassLoader 访问机制，子Loader 可以访问父Loader中类，但父却不能访问子。而我们在Agent 却引用了 HttpServletRequest 类，该类在commonLoader 中装载 所以是找不到的。




**解决办法：**

1. 将agent.jar 强行注入到commonLoader中
2. 将servlet-api 加入agnet 依赖路径中，让servlet-api.jar 在Launcher$AppClassLoader 中装载
3. 不直接调用servlet-api对像，而是通过反谢去访问HttpServletRequest 对象的值。

3种方法均可以实现，简单起见 直接采用第三2中，将servlet-api 一起打进agent 包中。

## 三、Agent 采集器整合

---

**知识点：**

1. 整合需求分析

2. TraceSession 会话机制设计
3. 编写实现TraceSession

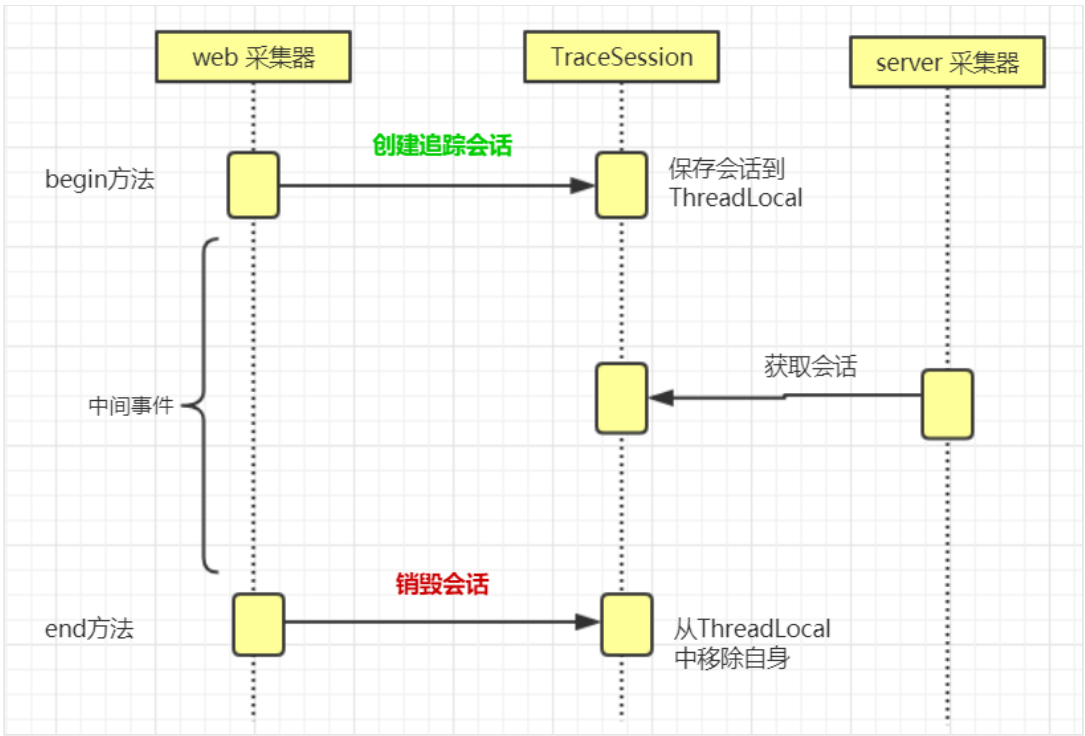
## 1、整合需求分析

前面我们已经实现了 Service 与 http 的执行执行参数采集，但二者的数据是相互独立的，并没有整合在一起，接下来我就一起 基于调用链中的TraceId 与EventId 整合在一起。整合后的采集器需要满足以下需求：

1. 将Http 请求所发送的事件基于TraceId进行串联
2. 事件之间要有层级与先后顺序
3. 多线程的情况下不影响采集结果

## 2、TraceSession 会话机制设计

为达到上述目标，这里需要设一个采集会话的概念，会话用于存储当前请求的trace 信息，包括traceId ,当前eventId 。流程如下图所示:

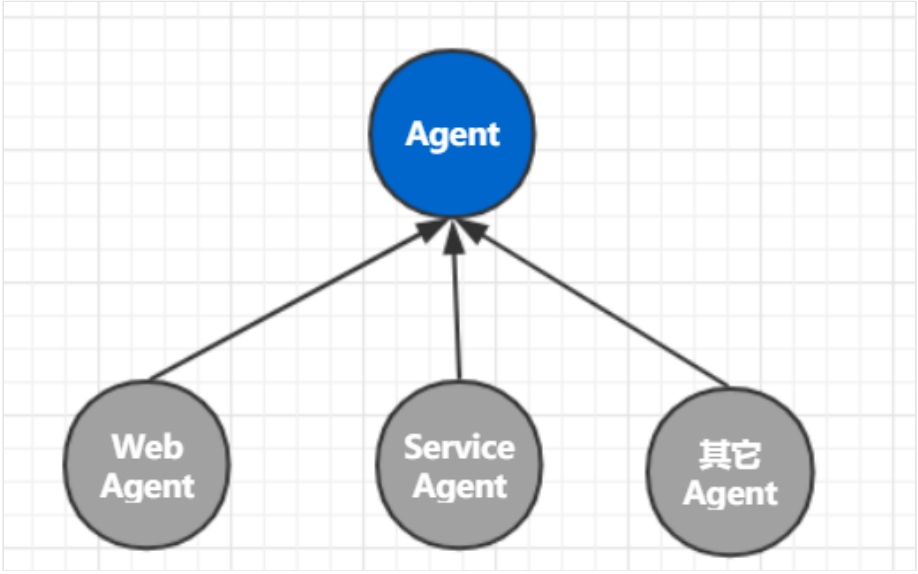



流程说明：

1. 由web采集器生成TraceID并开启会话
2. 其它采集器在这中间去获取会话信息，并生成EventID保存
3. web采集器关闭会话事件，采集过程结束

**关于Agent整合**

之 前是两个Agent 两个入口方法，显然是不行的，必须将其整合在一个Agent当中，由一个入口方法进行代理初始操作。整合之后如下图：




实现如下代码实示：

```
public static void premain(String args, Instrumentation instrumentation) {
    WebAgent.premain(args, instrumentation);
    ServiceAgent.premain(args, instrumentation);
}
```

### 3、编写实现TraceSession

ThreadLocal 关键代码

```
 public class TraceSession {
    private static ThreadLocal<TraceSession> session = new ThreadLocal<>
();
    private String traceId;
    private int currentEventId;
    private String parentId;
    //开启会话
    public TraceSession(String traceId, String parentId) {
        this.traceId = traceId;
        this.parentId = parentId;
        session.set(this);
        currentEventId = 0;
    }

    public int getNextEventId() {
        return ++currentEventId;
    }
    // 获取会话
    public static TraceSession getCurrentSession() {
        return session.get();
    }
    // 关闭会话
    public static void close() {
        session.remove();
    }
```

WEB开启会话：

```
  String traceId = UUID.randomUUID().toString().replaceAll("-", "");
        TraceSession session = new TraceSession(traceId, "0");
        trace.setTraceId(traceId);
        trace.setEventId(session.getParentId() + "." +
session.getNextEventId());
```

WEB关闭会话:

```
public static void end(Object webTraceInfo) {
    TraceSession.close();
}
```