JVM-Sandbox from Ali

随着软件部署规模的扩大，系统功能的细化，系统间耦合度和链路复杂度不断加强，若要继续保持现规模系统的稳定性，需要实现并完善监控体系、故障定位分析、流量录制回访、强弱依赖检测、故障演练等支撑工具平台。出于对服务器规模和业务稳定性的考量，这些配套工具平台要具备对目标应用具有无侵入、实时生效、动态可插拔的特点。要实现这些，会触及到底层技术-动态字节码框架，其中核心是面向切面编程技术（AOP），主要分为静态编织和动态编织两种：

1. 静态编织，发生在字节码生成时根据一定框架的规则提前将AOP字节码插入到目标类和方法中，实现AOP
2. 动态编织，允许在JVM运行过程中完成指定方法的AOP字节码增强。常见的动态编织方案大多采用重命名原有方法，再新建一个同签名的方法来做代理的工作模式来完成AOP的功能（常见方案如Cglib），但这种方式存在一些应用边界

* 侵入性，对代理的目标类需要侵入式改造，比如Spring容器中的Bean
* 固化性，目标代理方法在启动之后即固化，无法重新对一个已有方法进行AOP增强

JVM-Sandbox是一种基于JVM的非侵入式运行的AOP解决方案，可以实现以下功能：

1. 实现线上故障问题定位，支持线上链路监控排查
2. 动态添加日志，可以根据业务ID进行过滤
3. 系统内的异常模拟
4. 获取调用链路数据，用来识别场景、覆盖率统计等
5. 开发录制回放、故障模拟、动态日志、行链路获取等等工具

# JVM-Sandbox的使用

## 1.1 安装

1. 下载JVM Sandbox安装包

***wget http://ompc.oss-cn-hangzhou.aliyuncs.com/jvm-sandbox/release/sandbox-stable-bin.zip***

1. 下载完成并解压后，在sandbox目录下执行安装脚本~/sandbox/install-local.sh，并制定sandbox的安装目录，如若不指定，默认的安装目录是${home}/.opt

***unzip sandbox-stable-bin.zip***

***cd sandbox***

***./install-local.sh***

1. 成功安装会有以下输出，安装指定目录之后，沙箱脚本的工作目录将为指定的安装目录

***VERSION=0.1.0.6***

***PATH=/home/hdfs/.opt/sandbox***

***install sandbox successful.***

沙箱在启动过程中需要定位到配置文件和几个核心Jar包，如果直接运行，则沙箱的工作目录为./sandbox所在的目录。

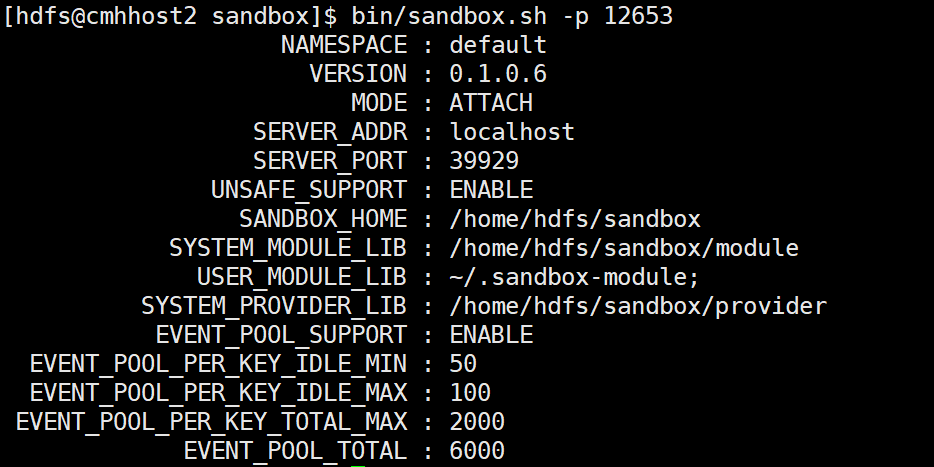
## 1.2 启动

沙箱的启动方式有两种：ATTACH和AGENT

1. ATTTCH方法启动，即插即用的启动模式，可以在不重启目标JVM的情况下完成沙箱的植入。原理和GREYS、BTrace类似，利用了JVM的Attach机制实现。命令如下：

***bin/sandbox.sh -p 12653***

如果输出：



则说明启动成功，沙箱已经顺利植入到目标JVM中，并完成底层端口的打开和所有模块的加载。

1. AGENT方式启动

使用ATTACH方式加载，可能会引起目标JVM的卡顿或者停顿（GC），所有在沙箱工作的方式可以采用AGENT方式启动，在JVM启动参数中增加上：

***-javaagent: /home/hdfs/.opt/sandbox/lib/lib/sandbox-agent.jar***

这样沙箱将会伴随着JVM启动而主动启动并加载对应的沙箱模块。

1. 查看已经加载的模块

***$ bin/sandbox.sh -p 12653 -l***

***module-mgr ACTIVE LOADED 0 0 0.0.1 luanjia@taobao.com***

***control ACTIVE LOADED 0 0 0.0.1 luanjia@taobao.com***

***info ACTIVE LOADED 0 0 0.0.3 luanjia@taobao.com***

4）sandbox.sh命令说明

|  |  |
| --- | --- |
| 选项 | 作用 |
| -h | 输出帮助信息 |
| -p | 指定目标JVM进程号，不用关心底层绑定的沙箱HTTP端口 |
| -v | 输出加载到目标JVM中的沙箱版本信息 |
| -l | 列出目标JVM沙箱中已经加载的模块 |
| -F | 强制刷新用户模块，在强制刷新的时候，首先卸载当前所有已经被加载的模块（释放所有模块开启的资源），然后再重新对用户模块进行加载 |
| -f | 刷新用户模块，仅对发生变化的文件进行重新加载操作 |
| -R | 沙箱模块重置，刷新所有的模块，包括用户模块和系统模块 |
| -u | 卸载指定模块，支持通配符表达式，不会区分系统模块和用户模块，所有模块都可以通过这个参数完成卸载，但是不要卸载module-mgr，否则将失去模块管理功能 |
| -a | 激活模块，激活后才收到沙箱事件 |
| -A | 冻结模块，将感知不到任何沙箱事件，但对象的代码插桩还在 |
| -m | 查看模块详细信息，模块名需要精确匹配，不支持通配符 |
| -I | 指定Sandbox IP Addr，默认使用0.0.0.0 |
| -P | 指定Sandbox的网络端口，默认使用随机端口 |
| -C | 仅仅是连接Server |
| -S | 关闭Server |
| -n | 指定jvm-sandbox的命名空间，默认default |
| -d | 执行module中的请求 |

-d命令的使用示例：

***$ bin/sandbox.sh -p 10525 -d 'info/event-pool'***

***BEFORE : 0 / 0***

***RETURN : 0 / 0***

***THROWS : 0 / 0***

***LINE : 0 / 0***

***CALL\_BEFORE : 0 / 0***

***CALL\_RETURN : 0 / 0***

***CALL\_THROWS : 0 / 0***

***IMMEDIATELY\_RETURN : 0 / 0***

***IMMEDIATELY\_THROWS : 0 / 0***

## 1.3 目录结构

sandbox的应用目录结构如下所示：

*./sandbox/*

*+--bin/*

*| +--sandbox.sh //sandbox的客户端脚本，用户启动和管理沙箱*

*|+--cfg/ //配置文件存放目录，里面存放了沙箱的所有的配置文件*

*| +--sandbox-logback.xml //日志框架：Logback，运行进行个性化的日志配置*

*| +--sandbox.properties //配置信息，配置文件只会在沙箱容器启动的时候加载以此*

*| `--version //沙箱容器的版本号*

*|+--lib/ //沙箱主程序的库包目录，存放的是沙箱工程的主程序，不能随意的删除、改变或者移动*

*| +--sandbox-agent.jar //沙箱启动代理*

*| +--sandbox-core.jar //沙箱内核*

*| `--sandbox-spy.jar //沙箱间谍库，用户提供插桩埋点的间谍类*

*|+--provider/*

*|. `--sandbox-mgr-provider.jar*

*|`--module/ //沙箱的系统模块目录，由配置项system\_module进行定义，比如沙箱模块管理功能的module-mgr模块，未来模块运行包括质量监控模块、安全校验模块*

*| `--sandbox-mgr-module.jar*

*|+${HOME}/.sandbox-module //沙箱用户模块目录，由配置项user\_module进行定义，一般存放用户自研的模块。*

*|+${HOME}/.sandbox.token，这个文件将完成目标JVM进程和沙箱客户端进程一些信息交互*

*|+${HOME}/logs，日志文件目录，可以通过修改sandbox-logback.xml文件进行修改日志输出配置*

## 1.4 配置文件详解

1) sandbox.properties配置详解

|  |  |
| --- | --- |
| 选项 | 作用 |
| system\_module | 系统模块本地路径 |
| user\_module | 用户模块本地路径，多值通配符表达式，多个路径可以通过”;”分割 |
| provider | 容器服务提供本地路径 |
| server.ip | 容器HTTP服务IP |
| server.port | 容器HTTP服务端口 |
| unsafe.enable | 是否增强JDK自带类，默认false |
| event.pool.enable | 是否启用事件对象池，启动后，容器占用一定的持久化内存 |
| event.pool.max.total | 整个事件池最大容量 |
| event.pool.min.idle.per.event | 事件池每个事件最小空闲容量 |
| event.pool.max.idle.per.event | 事件池每个事件最大空闲容量 |
| event.pool.max.total.per.event | 获取事件池每个事件最大容量 |

# JVM-Sandbox的原理

JVM-Sandbox基于AOP技术实现故障演练、强弱依赖检测、录制回放、精准回归等功能，这些功能的本质是完成java方法的环绕管控和运行时行链路的获取。在解决上面四个功能时，还要具备：

* 动态可插拔，即实现埋点方式的统一的API
* 无侵入性，即解决JVM类隔离的问题

下面是实现以上功能提供的模型和技术介绍。

## 2.1 沙箱容器

JVM-Sandbox由Java编写完成，基于JVMTI技术规范，为观察和改变代码运行结果提供了即插即用模块接口的容器，提供核心功能：

1. 使用埋点技术提供统一的API，来实现无需重启的AOP解决方案
2. 使用容器完成JVM类隔离，来解决侵入性问题
3. 提供模块管理机制，来完成各种模块的管理

沙箱容器采用了传统的C/S架构模式，通讯协议考虑到可能要部署到内外，需要NAT网关转发，所以采用了HTTP协议，对应实现如下：

* HTTP-CLIENT，sandbox.sh
* HTTP-SERVER，沙箱容器（jetty8实现）

在对每个目标JVM的进程启动沙箱容器时，容器会主动向目标JVM植入一个Agent，一般通过热部署的方式（基于JDK Instrumentation-API实现的HotSwap技术），容器加载的多个Module中共享这个Agent。JVM-Sandbox的整体架构如下：



Sandbox核心功能组件构成：

* 代码编织组件，负责完成预设代码的重写和生效
* 事件处理分发组件，负责完成事件的分发和方法流控控制的执行
* 模块管理组件，负责控制和管理沙箱的各个模块
* 沙箱底层提供了一个HTTP-SERVER(Jetty)，通过HTTP协议完成sandbox.sh和沙箱的控制交互，同事也给各个模块提供了基于HttpServlet和WebSocket规范的API，各模块可以复用沙箱完成各自模块的控制与交互。

## 2.2 核心事件模型

在JVM Sandbox中，Java的方法的调用都可以分解为BEFORE,RETURN和THROWS三个环节，由此在三个环节上引申出对应环节的事件探测和流程控制机制：

*// BEFORE*

*try {*

*/\**

*\* do something...*

*\*/*

*// RETURN*

*return;*

*} catch (Throwable cause) {*

*// THROWS*

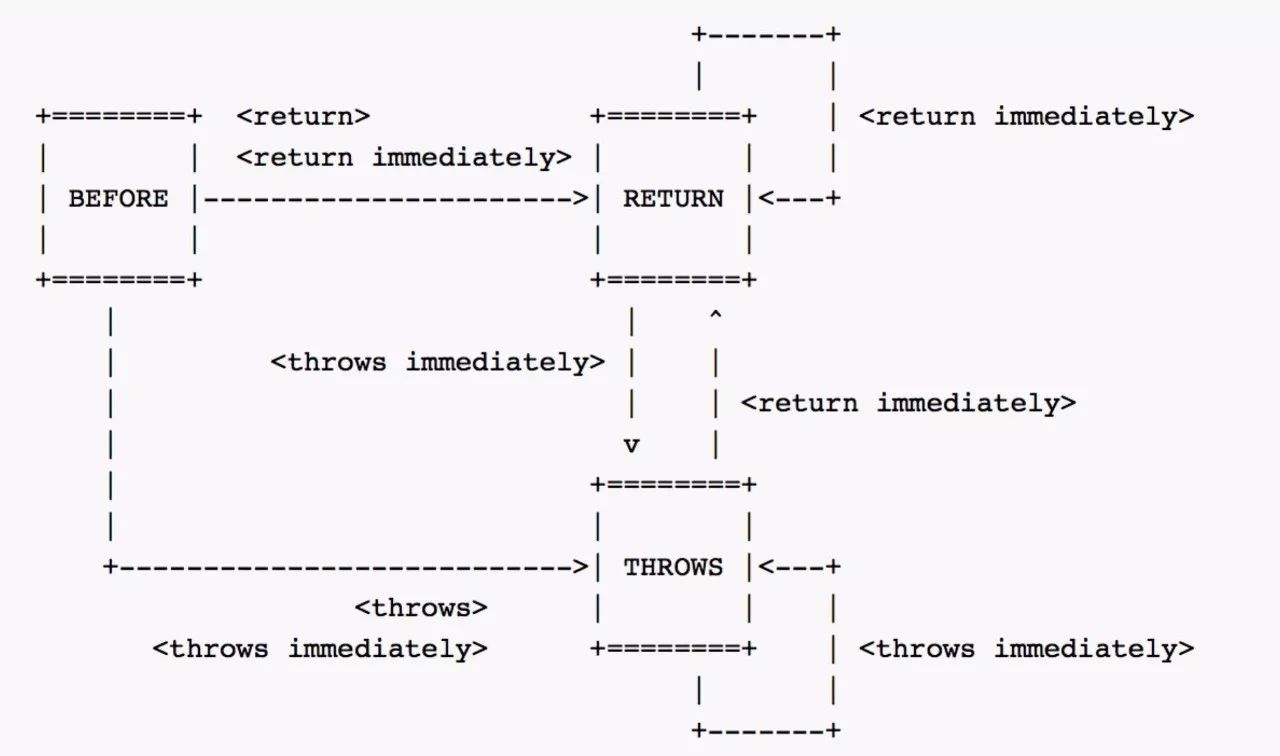
*}*

基于BEFORE，RETURN和THROWS三个环节事件，可以完成很多类AOP的操作：

1. 可以感知和改变方法调用的入参
2. 可以感知和改变方法调用的返回值和抛出的异常
3. 可以改变方法执行的流程

* 在方法体执行之前直接返回自定义结果对象，原有方法代码将不会被执行
* 在方法体返回之前重新构造新的结果对象，甚至可以改变为抛出异常
* 在方法体抛出异常之后重新抛出新的异常，甚至可以改变为正常返回

BEFORE、RETURN和THROWS三个环节事件的正常流转和干预流转如下图所示：



在Sandbox中InvokeEvent的类型如下所示：



AOP的作用是在这些点植入控制代码，当程序运行到植入点时（即事件发生），通过调用事件监听器EventListenerHandlers，调用用户自定义行为。

为了方便控制切入的代码，Sandbox提供active、frozen等功能，可以方便打开和关闭对应的功能，简单来说：

1. 用户自定义事件会分配一个唯一id，即listenerid
2. 当自定义的Module激活时，把它放入全局的globalEventListenerMap中，冻结就是移除
3. AOP植入的代码里会把listenerid也植入进入，在Event的处理流程中，如果没有找到对应的Listener，响应功能就不会执行。

***执行流程如下图所示：***

## 2.3 模块及模块的管理

JVM-Sandbox通过Module(模块)来实现对目标应用无侵入的监控、故障定位分析、流量录制回访、故障依赖等功能。

### 2.3.1 沙箱中的模块

沙箱中加载的模块分为两种：

1. 系统模块，进行沙箱通用的管理，比如沙箱模块管理功能的module-mgr模块，以后可能会支持的质量监控模块、安全校验模块等。系统模块不受刷新、强制刷新功能的影响，只有容器重置能让沙箱重新加载系统模块目录下的所有模块，定义在./sandbox/module。
2. 用户模块，由配置项user\_module进行定义，用于存放用户自研模块，这些模块可以动态热插拔，刷新或强制刷新可以完成重新加载。

目前Sandbox中自带的Module，如下所示：



### 2.3.2 模块的状态和生命周期

对于沙箱模块共有四种状态：

* 加载，模块被沙箱正确加载，沙箱将会被允许模块正常申请HTTP、WEBSOCKET等资源。
* 卸载，沙箱不再使用该模块，对分配给这些模块的资源进行回收，包括模块已经侦听事件的类都将会移除侦听插桩。
* 冻结，模块加载成功后默认的是冻结状态，之前侦听的所有沙箱事件都将被屏蔽，冻结的模块不会退回事件侦听的代码插桩，只有delete、watching或者模块被卸载的时候插桩代码才会被清理
* 激活，模块需要进行激活，激活后才能监听到沙箱事件。

模块的加载和卸载的生命周期被定义在ModuleLifecycle中，在类中的体现如下：



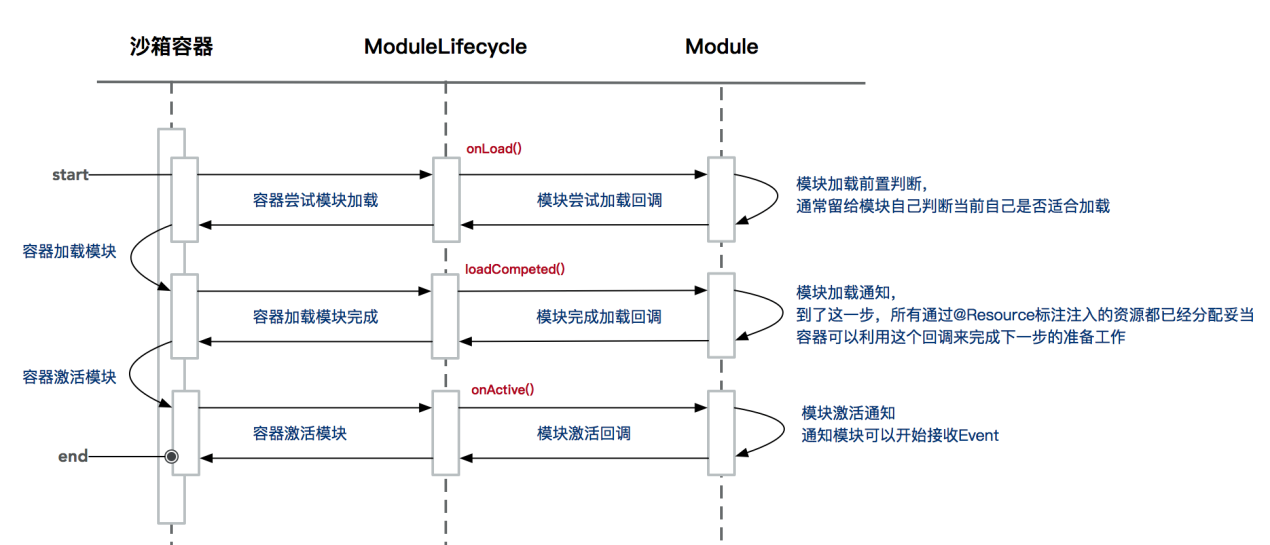
模块的生命周期，如下图所示：



一个模块要发挥作用，要经过以下步骤：

1. 加载，将模块的JAR包加载到容器中，并完成类初始化和依赖注入的工作，JAR包中可以申明多个模块，模块需要符合Java SPI规范。
2. 对观察类进行插桩操作
3. 激活模块，使得模块能接受Event
4. 当模块没有作用时，冻结模块，该模块将无法接收到观察类的Event
5. 卸载，对观察类进行反向插桩，消除本模块对类的影响。模块类从容器中移除，若当前模块是JAR文件中的最后一个模块，则将整个ModuleClassLoader进行销毁。

### 2.3.3 模块加载流程



https://github.com/alibaba/jvm-sandbox/wiki/USER-GUIDE

https://github.com/alibaba/jvm-sandbox/wiki/FOR-EXAMPLE

## 2.4 模块隔离与通讯

### 2.4.1 模块隔离

1）沙箱通过自定义的SandboxClassLoader破坏了双亲委派的约定，实现和观察应用的类隔离，所有不用担心加载沙箱会引起应用的类污染、冲突。

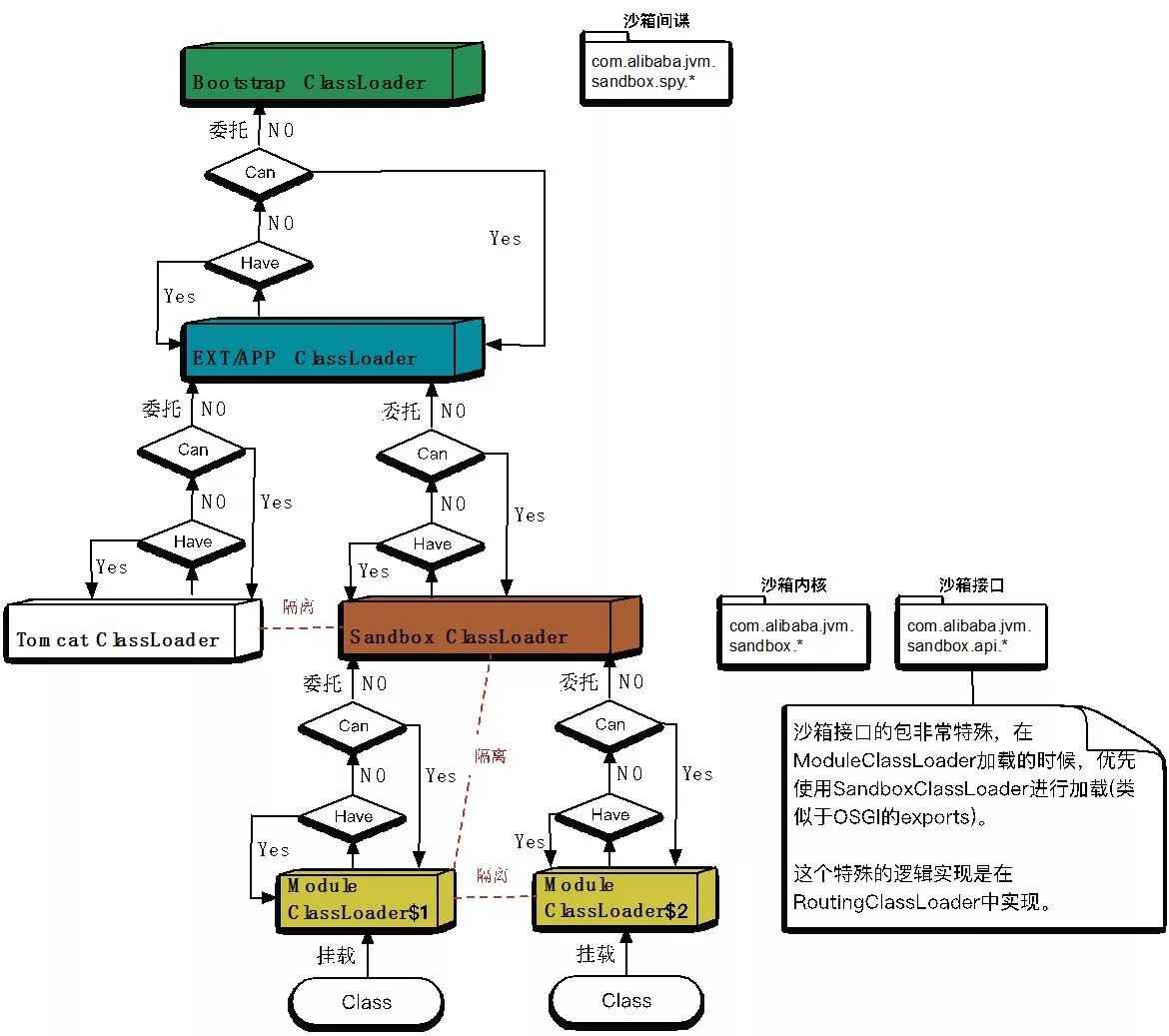
//详细源码分析

2）沙箱各模块之间通过ModuleClassLoader实现各自的独立，达到模块之间、模块和沙箱之间、模块和应用之间互不干扰。

//详细源码分析

### 2.4.2 模块通讯

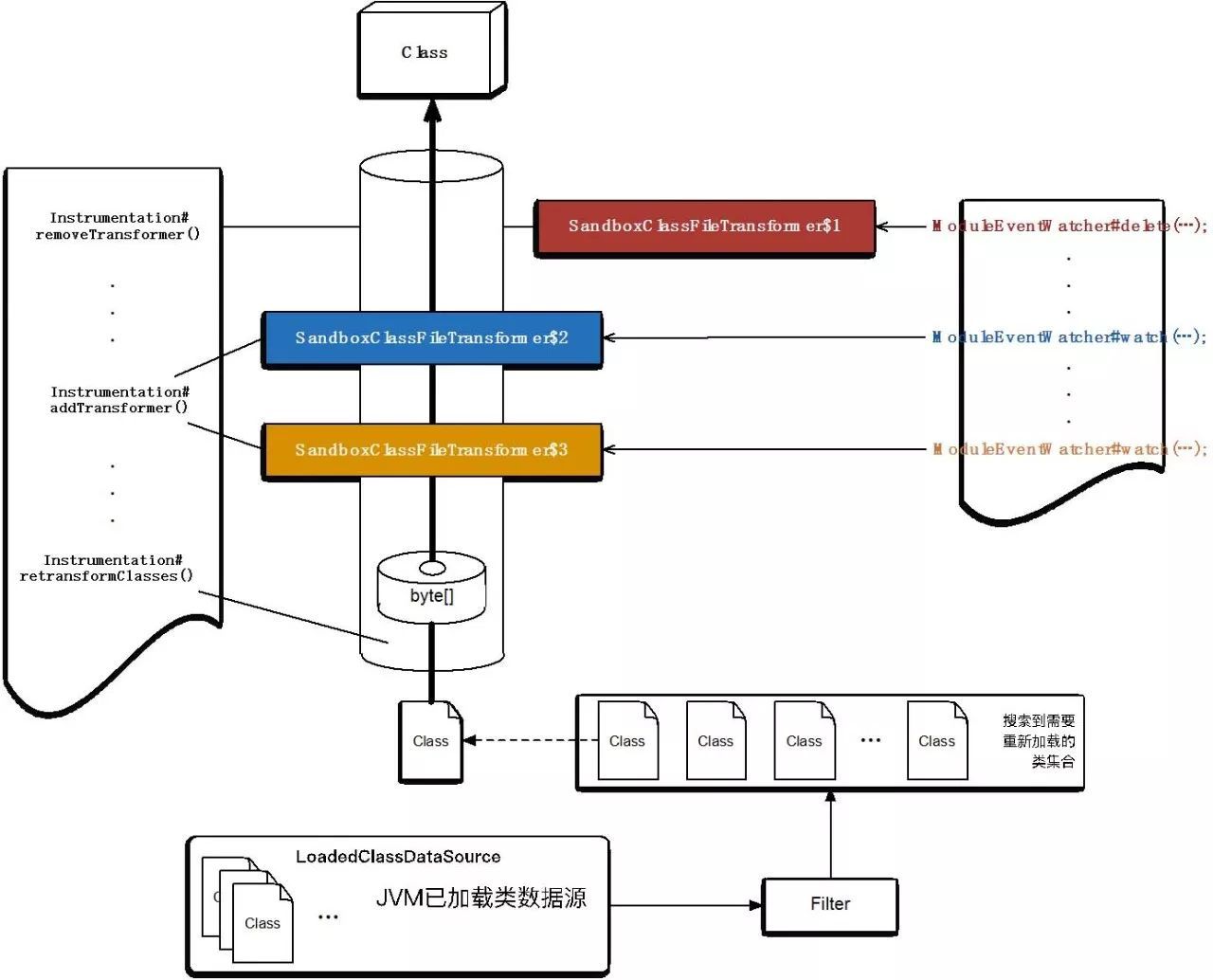
通过向Bootstrap ClassLoader中注入Spy类，完成观察的应用与JVM-Sandbox的通讯，ClassLoader的加载如下图所示：



JVM-Sandbox会将事件分发给各个Module，完成JVM-Sandbox与Module之间的通讯。

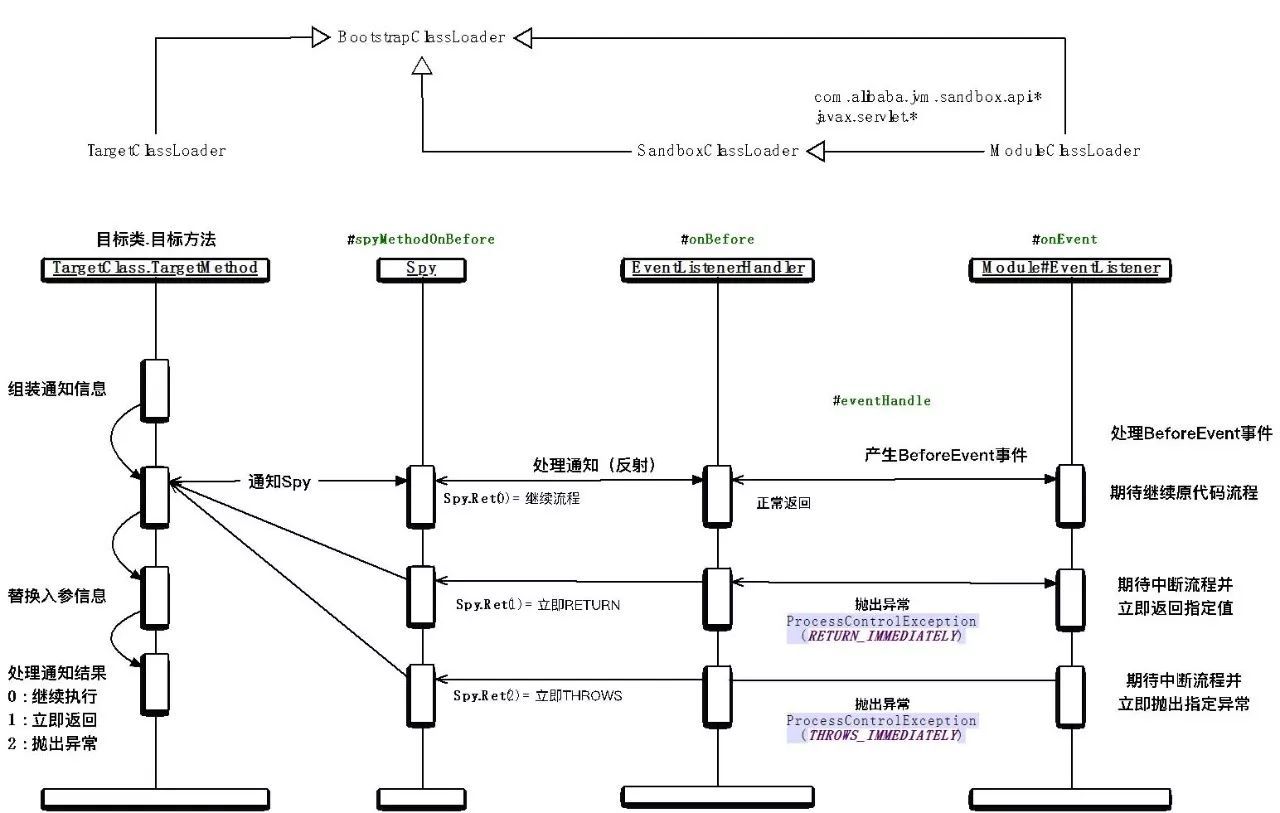
## 2.5 代码编织

通过transform方法形变原生字节码，如下图所示：



插入Spy类到字节码中，Spy方法中反射调用JVM-Sandbox的方法，以BeforeEvent为例，进行代码编织展示：





# JVM-Sandbox Module的开发

在该部分，定义一个JVM-Sandbox Module，定义一个抽象类的钟Clock，期望每隔一定时间进行报时，但是实现的钟为BrokenClock，在报时时抛出异常。通过Module实现正常的输出时间：

## 3.1 定义功能类

Clock及BrokenClock定义如下：

*public abstract class Clock {*

*abstract void checkState();*

*private final java.text.SimpleDateFormat clockDateFormat =*

*new java.text.SimpleDateFormat("yyyy-MM-dd HH:mm:ss");*

*final String formatDate(Date date) {*

*return clockDateFormat.format(date);*

*}*

*final Date nowDate() {*

*return new Date();*

*}*

*final String report() {*

*checkState();*

*return formatDate(nowDate());*

*}*

*abstract void delay() throws InterruptedException;*

*final void loopReport() throws InterruptedException {*

*while(true) {*

*try {*

*System.out.println(report());*

*} catch(Throwable cause) {*

*cause.printStackTrace();*

*}*

*delay();*

*}*

*}*

*}*

BrokenClock实现如下：

*public class BrokenClock extends Clock {*

*@Override*

*void checkState() {*

*throw new IllegalStateException();*

*}*

*@Override*

*void delay() throws InterruptedException {*

*Thread.sleep(10000L);*

*}*

*public static void main(String[] args) {*

*BrokenClock clock = new BrokenClock();*

*try {*

*clock.loopReport();*

*} catch (InterruptedException e) {*

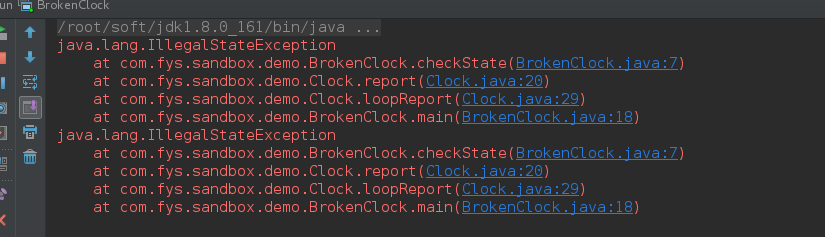
*e.printStackTrace();*

*}*

*}*

*}*

运行时，抛出异常：



## 3.2 通过Module修复损坏的钟

分析BrokenClock，问题出现在checkState方法时抛出了一个异常，而且delay时间比较长延迟了10秒钟。

### 3.2.1 创建Java工程Clock-tinker

在maven pom.xml中添加Sandbox 库的依赖

*<dependencies>*

*<dependency>*

*<groupId>com.alibaba.jvm.sandbox</groupId>*

*<artifactId>sandbox-api</artifactId>*

*<version>1.0.14</version>*

*</dependency>*

*<dependency>*

*<groupId>com.alibaba.jvm.sandbox</groupId>*

*<artifactId>sandbox-common-api</artifactId>*

*<version>1.0.14</version>*

*</dependency>*

*<dependency>*

*<groupId>javax.servlet</groupId>*

*<artifactId>javax.servlet-api</artifactId>*

*<version>3.0.1</version>*

*<scope>provided</scope>*

*</dependency>*

*</dependencies>*

编写Module类BrokenClockTinkerModule：

*@Information(id = "broken-clock-tinker")*

*public class BrokenClockTinkerModule implements Module {*

*@Resource*

*private ModuleEventWatcher moduleEventWatcher;*

*@Http("/repairCheckState")*

*public void repairCheckState() {*

*moduleEventWatcher.watch(*

*new NameRegexFilter("com\\.fys\\.sandbox\\.demo\\.BrokenClock",*

*"checkState"),*

*new EventListener() {*

*public void onEvent(Event event) throws Throwable {*

*ProcessControlException.throwReturnImmediately(null);*

*}*

*},*

*Event.Type.THROWS*

*);*

*}*

*@Http("/repairDelay")*

*public void repairDelay() {*

*moduleEventWatcher.watch(*

*new NameRegexFilter("com\\.fys\\.sandbox\\.demo\\.BrokenClock",*

*"delay"),*

*new EventListener() {*

*public void onEvent(Event event) throws Throwable {*

*Thread.sleep(1000L);*

*System.out.println("Call Before invoked");*

*ProcessControlException.throwReturnImmediately(null);*

*}*

*},*

*Event.Type.CALL\_BEFORE*

*);*

*}*

*}*

### 3.2.2 根据SPI规范注册

1）创建*META-INF/services/com.alibaba.jvm.sandbox.api.Module*，

2）文件中添加模块的类路径：*com.fys.sandbox.module.BrokenClockTinkerModule*

### 3.2.3 编译部署clock-tinker模块

在pom中添加编译配置

*<build>*

*<plugins>*

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

*<artifactId>maven-assembly-plugin</artifactId>*

*<executions>*

*<execution>*

*<goals>*

*<goal>attached</goal>*

*</goals>*

*<phase>package</phase>*

*<configuration>*

*<descriptorRefs>*

*<descriptorRef>jar-with-dependencies</descriptorRef>*

*</descriptorRefs>*

*</configuration>*

*</execution>*

*</executions>*

*</plugin>*

*</plugins>*

*</build>*

运行命令打包：

*mvn clean package*

将打包好的包复制到用户模块目录.sanbox-module中

*cp target/clock-tinker-1.0 -jar-with-dependencies.jar ~/.sandbox-module/*

### 3.2.4启动Sandbox

*# bin/sandbox.sh -p 20925*

*# bin/sandbox.sh -p 20925 -l*

*broken-clock-tinker ACTIVE LOADED 0 0 UNKNOW\_VERSION UNKNOW\_AUTHOR*

*module-mgr ACTIVE LOADED 0 0 0.0.1*

*control ACTIVE LOADED 0 0 0.0.1*

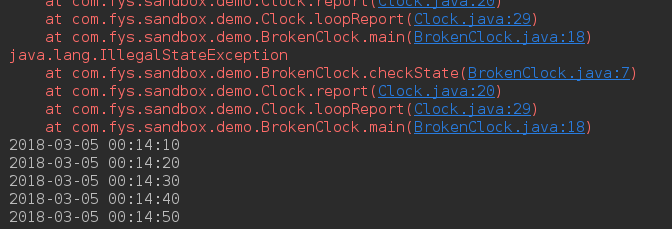
*info ACTIVE LOADED 0 0 0.0.3*

模块添加成功

### 3.2.5 激活修复checkState方法

*# bin/sandbox.sh -p 20925 -d 'broken-clock-tinker/repairCheckState'*

执行后，程序输出如下：



修复成功

## 3.3 代码分析，核心方法

*@Http("/repairCheckState")*

*public void repairCheckState() {*

*moduleEventWatcher.watch(*

*new NameRegexFilter("com\\.fys\\.sandbox\\.demo\\.BrokenClock",*

*"checkState"), //配置要识别的类*

*new EventListener() {*

*public void onEvent(Event event) throws Throwable {*

*//遇到监听事件的处理方式，直接返回null*

*ProcessControlException.throwReturnImmediately(null);*

*}*

*},*

*Event.Type.THROWS //调用事件，监听throws事件*

*);*

*}*

https://www.jianshu.com/p/9d76aa950317

https://github.com/alibaba/jvm-sandbox/blob/master/README.md