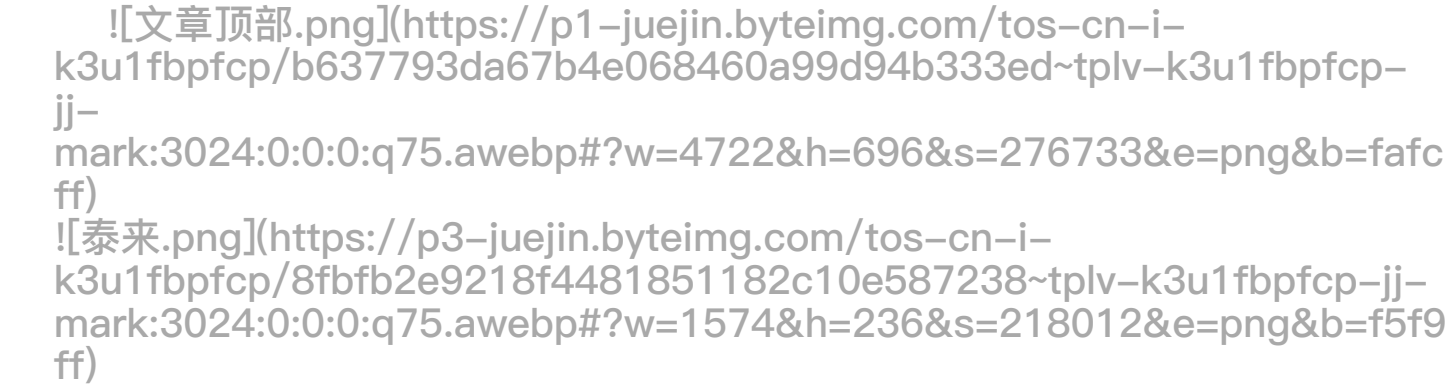
JVM-Sandbox 分享



JVM-Sandbox 分享

本文围绕 JVM-Sandbox 展开，介绍 JVM-Sandbox 架构，以及怎么开发一个自定义模块。主要内容如下：

* JVM-Sandbox 简介
* JVM-Sandbox 核心概念及架构
* JVM-Sandbox 设计相关技术
* JVM-Sandbox 使用示例
* JVM-Sandbox 使用场景

JVM-Sandbox 简介

我们先看一下经常使用的 Spring AOP，其基于代理，实现类功能增强，然后注入 Spring 容器中。Spring AOP 所面临的问题，对代码侵入性强，必须依赖 Spring 容器；目标代理方法在启动后，无法重新对一个已有方法进行 AOP 增强。

如果我们想在不被 Spring 容器托管的类或则运行的项目通过 AOP 做一些功能，要怎么实现?

Java 提供 JVM TI（Java Virtual Machine Tool Interface）和 Instrumentation API 相关能力，允许使用者对 JVM 的控制，实现复杂的能力

。本次分享的 JVM-Sandbox 就属于此类框架。

JVM-SANDBOX（沙箱）实现了一种在不重启、不侵入目标 JVM 应用的 AOP 解决方案。 JVM-SANDBOX 属于基于 Instrumentation 的动态编织类的 AOP 框架，通过精心构造了字节码增强逻辑，使得沙箱的模块能在不违反 `JDK` 约束情况下实现对目标应用方法的`无侵入`运行时 AOP 拦截。实现了一种在不重启、不侵入目标 JVM 应用的 AOP 解决方案。

常见的应用场景

* 线上故障定位
* 线上系统流控
* 线上故障模拟
* 方法请求录制和结果回放
* 动态日志打印
* 安全信息监测和脱敏

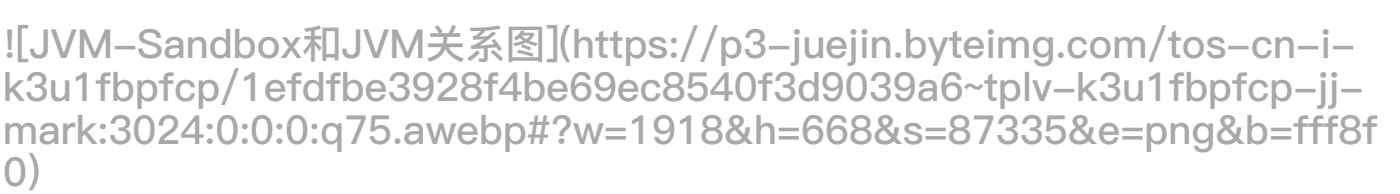
JVM-Sandbox 核心概念及架构

再使用 JVM-Sandbox 开发模块时，先了解 Sandbox 是怎么工作的

核心概念

Sandbox 面向模块研发者几个核心概念

* 模块 (Module): JVM-Sandbox 的功能单元，实现特定功能的代码集合；
* 事件 (Event): JVM 中发生的特定行为，例如方法调用、异常抛出等；
* 监听器 (Listener): 监听特定事件，并在事件发生时执行相应逻辑。



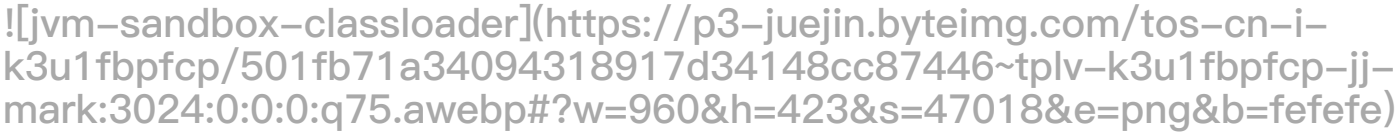
0. **目标 JVM:** 运行着需要监控或调试的目标应用程序 (App)。
1. **JVM-Sandbox Agent:** 通过 Java Agent 技术加载到目标 JVM 中，作为 JVM-Sandbox 的入口。

2. **Sandbox 核心:** 负责初始化 Sandbox 环境、加载模块、管理事件等核心功能。
3. **模块管理器:** 管理所有已加载的模块，包括加载、卸载、激活、禁用等操作。
4. **模块:** JVM-Sandbox 的功能单元，包含事件监听器和处理逻辑，例如监控模块、调试模块等。
5. **事件:** 目标应用程序运行过程中发生的特定行为，例如方法调用、异常抛出等。
6. **监听器:** 监听特定类型的事件，并在事件发生时执行相应的处理逻辑。

类隔离策略

在 Java 中对于任意一个类，都必须由加载它的类加载器和这个类本身一起共同确立其在 Java 虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话可以表达得更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个 Class 文件，被同一个 Java 虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

根据 JVM-Sandbox 官方提供的类隔离策略图，沙箱通过自定义的 `SandboxClassLoader` 破坏了双亲委派的约定，实现了和目标应用的类隔离。各模块之间类通过 `ModuleJarClassLoader` 实现了各自的独立，达到模块之间、模块和沙箱之间、模块和应用之间互不干扰。`sanbox-agent` 则由 `AppClassLoader` 进行加载。`sandbox-spy` 间谍类用 `BootstrapClassLoader` 进行加载，目的利用双亲委派加载模型，保证间谍类可以正确的被目标 JVM 加载，从而植入到目标 JVM 中，完成与业务代码的交互。

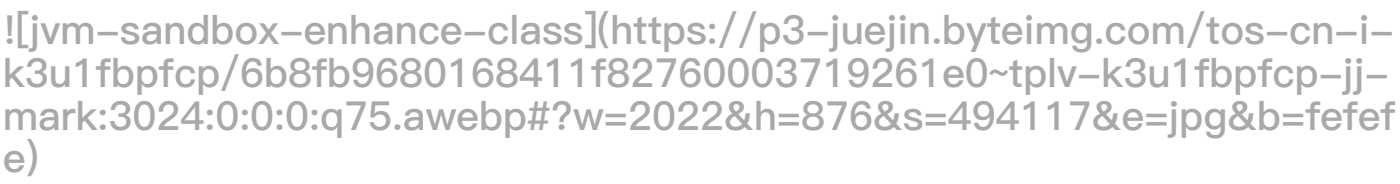


类增强策略

JVM-Sandbox 属于基于 Instrumentation 的动态编织类的 AOP 框架，**通过精心构造了字节码增强逻辑，使得沙箱的模块能在不违反 JDK 约束情况下实现对目标应用方法的`无侵入`运行时 AOP 拦截**。

沙箱通过在 `BootstrapClassLoader` 中埋藏的 `Spy` 类完成目标类和沙箱内核的通讯。

以下官方给的代码增强示例图：



事件

在沙箱的世界观中，任何一个 Java 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`三个环节事件分离，沙箱的模块可以完成很多类AOP的操作。

0. 可以感知和改变方法调用的入参
1. 可以感知和改变方法调用返回值和抛出的异常
2. 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

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`。沙箱的方法分量大类：`流程控制类`和`执行探测类`

流程控制类事件

在流程控制类事件的处理回调中，我们除了可以探测到一个方法执行的三个环节之外，还可以修改入参、返回值和抛出的异常。甚至你还可以控制流程将原本要正常返回的方法改变为抛出你指定的异常，或者原本要抛出异常的方法变为返回你指定的值。

* **BEFOR E事件**：执行方法体之前被调用
* **RETURN 事件**：执行方法体返回之前被调用
* **THROWS 事件**：执行方法体抛出异常之前被调用
* **IMMEDIATELY_RETURN**：立即调用：RETURN 事件
* **IMMEDIATELY_THROWS**：立即调用：THROWS 事件

严格意义上，`IMMEDIATELY_RETURN`和 `IMMEDIATELY_THROWS`不是方法触发的事件，他们是`流程控制`过程中内部触发。正常使用过程中不需要单独对这两个事件进行处理。

> 流程控制类事件的声明周期

```
                                  +--------+
                                  |        |
 +=========+  <return>            +=========+    | <return immediately>
 |         |  <return immediately> |         |    |
 | BEFORE |----------------------->| RETURN |<---+
 |         |                       |         |
 +=========+                       +=========+
     |                         |     ^
     |          <throws immediately> |     |
     |                         |     | <return immediately>
     |                         v     |
     |                        +=========+
     |                        |         |
     +------------------------->| THROWS |<---+
                  <throws>     |         |    |
```

```
        <throws immediately>      +========+    | <throws immediately>
                                  |        |
                                  +--------+
```

执行探测类事件

在执行探测类事件的处理回调中，我们可以感知到一个方法体内部运行的过程。在此类事件的回调中不允许改变方法执行的流程。

* **LINE事件**：方法中一行被执行之前，将会抛出行事件
* **CALL_BEFORE事件**：方法中调用另一个方法之前
* **CALL_RETURN事件**：方法中调用另一个方法之后，正常返回
* **CALL_THROWS事件**：方法中调用另一个方法之后，抛出异常

> 监听foo方法的BEFORE、RETURN、THROWS、LINE、CALL_BEFORE、CALL_RETURN、CALL_THROWS事件

```
void foo(){
  // BEFORE-EVENT
  try {

      /*
      * do something...
      */
      try{
          //LINE-EVENT
          //CALL_BEFORE-EVENT
        a();
        //CALL_RETURN-EVENT
      } catch (Throwable cause) {
        // CALL_THROWS-EVENT
    }
    //LiNE-EVENT
      // RETURN-EVENT
      return;

  } catch (Throwable cause) {
      // THROWS-EVENT
  }
```

}

```

### **模块生命周期**

模块生命周期类型有`模块加载`、`模块卸载`、`模块激活`、`模块冻结`、`模块加载完成`五个状态。

* **模块加载**：创建ClassLoader，完成模块的加载
* **模块卸载**：模块增强的类会重新load，去掉增强的字节码
* **模块激活**：模块被激活后，模块所增强的类将会被激活，所有`com.alibaba.jvm.sandbox.api.listener.EventListener` 将开始收到对应的事件
* **模块冻结**：模块被冻结后，模块所持有的所有`com.alibaba.jvm.sandbox.api.listener.EventListener` 将被静默，无法收到对应的事件。需要注意的是，模块冻结后虽然不再收到相关事件，但沙箱给对应类织入的增强代码仍然还在。
* **模块加载完成**：模块加载已经完成，这个状态是为了做日志处理，本身不会影响模块变更行为

模块可以通过实现`com.alibaba.jvm.sandbox.api.ModuleLifecycle`接口，对模块生命周期进行控制。

### 整体架构图

JVM-Sandbox 通过 HTTP 服务器（Jetty）接受用户命令对模块进行管理，通过 JVM TI 接口对目标方法增强。

在 JVM-Sandbox 项目中，`agent`、`core`、`spy` 是其核心主程序。

* `agent`：沙箱启动代理
* `core`：沙箱内核
* `spy`：沙箱间谍类（增强埋点）

官方整体架构图


```

mark:3024:0:0:0:q75.awebp#?w=1061&h=375&s=84004&e=png&a=1&b=f9f8f8)

* 模块控制管理：负责管理 JVM-Sandbox 自身模块以及使用者自定义模块，例如模块的加载，激活，冻结，卸载。
* 事件监听处理：用户自定义模块实现 Event 接口对增强的事件进行自定义处理，等待事件分发处理器的触发。
* 沙箱事件分发处理器：对目标方法增强后会对目标方法追加三个环节，分别为方法调用前 `BEFORE`、调用后 `RETURN`、调用抛异常 `THROWS`、当代码执行到这三个环节时则会由分发器分配到对应的事件监听执行器中执行。
* 代码编织框架：通过 ASM 框架依托于 JVM TI 对目标方法进行字节码修改，从而完成代码增强的能力。
* 检索过滤器：当用户对目标方法创建增强事件时，沙箱会对目标 JVM 中的目标类和方法进行匹配以及过滤。匹配到用户设定目标类和方法进行增强，过滤掉 Sandbox 内部的类以及 JVM 认为不可修改的类。
* 加载类检索: 获取到需要增强的类集合依赖检索过滤器模块。
* HTTP 服务器：通过 HTTP 协议与客户端进行通信（sandbox.sh 即可理解为客户端）本质是通过 `curl` 命令下发指令到沙箱的 HTTP 服务器。例如模块的加载，激活，冻结，卸载等指令。

相关底层技术

Sandbox 中 SPI

在沙箱中，自定义模块的加载利用的 Java SPI 机制，所以模块需要符合 `Java SPI` 规范要求。

* 必须拥有 `public` 的无参构造函数；
* 必须实现 `com.alibaba.jvm.sandbox.api.Module` 接口；
* 必须要在约定的 `resources/META-INF/services` 目录下创建 `com.alibaba.jvm.sandbox.api.Module` 为名的文件，文件内容则为自定义 `Module` 的完整类路径。

其中 Java SPI 文件注册可以通过在实现类上加 `org.kohsuke.MetaInfServices` 注解实现。编译时自动在 `resources/META-INF/services` 目录创建接口文件，并写入实现类路径。注解依赖

```
```

```
<dependency>
    <groupId>org.kohsuke.metainf-services</groupId>
    <artifactId>metainf-services</artifactId>
    <version>1.7</version>
    <scope>compile</scope>
</dependency>
```

JVM TI

JVM TI (Java Virtual Machine Tool Interface) 是 **Java 虚拟机所提供的 native 编程接口**，是 JVM 提供的一组用于构建性能分析、调试、监控和其他工具的编程接口。它允许开发者深入 JVM 内部，访问运行时数据，并控制 JVM 的行为，从而实现强大的工具功能。

JVM TI 基于事件驱动和回调机制。工具通过注册回调函数来响应 JVM 中的各种事件。主要包含以下组件:

* **JVMTI 环境 (Environment):** 代表 JVM TI 的上下文，提供访问 JVM TI 函数的接口。
* **事件 (Event):** JVM 中发生的特定行为，例如类加载、方法调用、异常抛出等。
* **事件回调函数 (Event Callback Function):** 用户定义的函数，用于处理特定类型的事件。
* **能力 (Capabilities):** 一组控制 JVM TI 行为的标志，例如是否允许修改字节码、是否启用对象分配跟踪等。

当 JVM 加载类文件时会触发类文件加载钩子事件 ClassFileLoadHook，从而触发 Instrumentation 类库中的 ClassFileTransformer (字节码转换器)的 `transform` 方法，在 `transfrom` 方法中可以对字节码进行转换。

Instrumentation

Instrumentation 是 JVM 提供的可以在运行时动态修改已加载类的基础库，**Instrumentation 的底层实现依赖于 JVM TI。**

Java Agent 通过 Java Instrumentation API 实现，其核心机制是对字节码进行操作。Instrumentation API 提供了一组接口，允许开发者在类加载之前或之后修改类的字节码。Instrumentation 实例只能通过 Agent 的 `premain` 或者 `agentmian` 方法的参数中获取。

Java Agent 关键组件和步骤

0. **Premain 方法**：

* 在 JVM 启动时执行，用于初始化代理。
* 方法签名：`public static void premain(String agentArgs, Instrumentation inst)`
1. **Agentmain 方法**：

* 在 JVM 运行时执行，用于动态附加代理。
* 方法签名：`public static void agentmain(String agentArgs, Instrumentation inst)`
2. **Instrumentation 接口**：

* 提供操作字节码的功能，例如添加 ClassFileTransformer、重新定义类等。
3. **ClassFileTransformer 接口**：

* 用于实现字节码转换逻辑。

综上加载 Agent 常见方式

0. 在 JVM 启动时执行。通过 JVM 启动参数 `–javaagent` 加载 Agent。这种方式会伴随着 JVM 一起启动，利用 Agent 提供的 `premain` 方法。

优点

* **完全控制**：在 JVM 启动时加载，可以确保在应用程序代码执行之前完成 Agent 的初始化。
* **性能影响小**：由于在应用启动时加载，避免了运行时附加带来的开销。如果 Agent 启动需要加载大量的类，随着 JVM 启动时直接加载不会导致 JVM 在运行时卡顿或者 CPU 抖动，缺点是不够灵活。**缺点**

* **需要重启 JVM**：如果需要更换或更新 Agent，需要重启 JVM。
1. 在 JVM 运行时执行，用于动态附加代理。利用 Attach API 在 JVM 运行时不需要重启的情况下即可完成挂载，使用 Agent 提供 `agentmain` 方法，即插即用的模式。Attach API 提供了一种附加到 Java 虚拟机的机制，使用此 API

附加到目标虚拟机并将其工具代理加载到该虚拟机中，本质上就是提供了和目标 JVM 通讯的能力。

优点

* **无需重启**：可以在应用运行时动态附加 Agent，不需要重启 JVM。
* **灵活性高**：可以根据需要随时附加或移除 Agent。**缺点**

* **性能影响**：在运行时动态附加可能会带来一些性能开销，特别是在高并发场景下。如果 Agent 启动需要加载大量的类可能会导致目标 JVM 出现卡顿，CPU 抖动等情况。
* **复杂性**：实现和管理 Attach 逻辑相对复杂。

在 Instrumentation 接口中提供了多个 API 用来管理和操作字节码

* `addTransformer`：注册字节码转换器,当注册一个字节码转换器后，所有的类加载都会经过字节码转换器进行处理。
* `retransformClasses`：重新对 JVM 已加载的类进行字节码转换
* `removeTransformer`：删除已注册的字节码转换器，删除后新加载的类不会再经过字节码转换器处理，但是已经“增强”过的类还是会继续保留

ClassFileTransformer

ClassFileTransformer（字节码转换器）是一个接口。

在 `transform` 可以返回转换后的字节码 `byte[]`，可以通过 `Instrumentation#addTransformer` 方法将实现的字节码转换器进行注册，一旦注册后字节码转换器就会在合适的时机被触发。

0. 新加载类的时候，例如 `ClassLoader.defineClass`
1. 重新定义类的时候，例如`Instrumentation.redefineClasses`
2. 对类重新转换的时候，例如 `Instrumentation.retransformClasses`

模块声明启动模式

沙箱有两种启动方式：`Attach` 和 `Agent`。

在模块中使用 `@Information` 声明 `mode`，自动模块启用方式。

* `Attach`：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；
* `Agent`：随着 `JVM` 启动而主动启动并加载对应的沙箱模块。

此方式需要在 `JVM` 启动参数中增加参数。

如 Sandbox 安装位置在 `/Users/xxx/DevEnv/sandbox`，需要在 `JVM` 启动参数中增加参数

```
-javaagent:/Users/xxx/DevEnv/sandbox/lib/sandbox-agent.jar
```

JVM-Sandbox 使用示例

具体使用方式参考[官方文档](http://cxyroad.com/”https://github.com/alibaba/jvm-sandbox”)

一个简单的示例

在开发中我们有时需要系统进行故障演练，在系统制定。通过一个简单的故障演练 `Demo` 快速了解 `JVM-Sandbox` 使用。

首先新建一个单独的工程，在 `Maven` 依赖中引用 `JVM-Sandbox`，官方推荐独立工程使用 `parent` 方式。

```
<parent>
    <groupId>com.alibaba.jvm.sandbox</groupId>
    <artifactId>sandbox-module-starter</artifactId>
    <version>1.4.0</version>
```

</parent>

```

新建一个类作为一个 `JVM-Sandbox` 模块

```
@MetaInfServices(Module.class)
@Information(id = "demo-breakdown", version = "0.0.1")
public class DemoBreakdownModule implements Module,
ModuleLifecycle {

    private static final Logger logger =
LoggerFactory.getLogger(DemoBreakdownModule.class);

    @Resource
    private ModuleEventWatcher moduleEventWatcher;

    @Override
    public void onLoad() throws Throwable {
        logger.info("DemoBreakdownModule onLoad");
    }

    @Override
    public void onUnload() throws Throwable {
        logger.info("DemoBreakdownModule onUnload");
    }

    @Override
    public void onActive() throws Throwable {
        logger.info("DemoBreakdownModule onActive");
    }

    @Override
    public void onFrozen() throws Throwable {
        logger.info("DemoBreakdownModule onFrozen");
    }

    @Override
    public void loadCompleted() {
        logger.info("DemoBreakdownModule loadCompleted");
    }

    /**
     * 自定义用户命令
     * @param param
```
```

```
     * @param writer
     */
    @Command("breakdown")
    public void breakdown(final Map<String, String> param, final
PrintWriter writer) {
        final Printer printer = new ConcurrentLinkedQueuePrinter(writer);

        // -------- 参数解析 --------
        final String cnPattern = param.get("class");
        final String mnPattern = param.get("method");

        // 过滤器
        Filter exceptionFilter = new Filter() {
            @Override
            public boolean doClassFilter(int access,
                    String javaClassName,
                    String superClassTypeJavaClassName,
                    String[] interfaceTypeJavaClassNameArray,
                    String[] annotationTypeJavaClassNameArray) {
                return Objects.equals(javaClassName, cnPattern);
            }

            @Override
            public boolean doMethodFilter(int access,
                    String javaMethodName,
                    String[] parameterTypeJavaClassNameArray,
                    String[] throwsTypeJavaClassNameArray,
                    String[] annotationTypeJavaClassNameArray) {
                return Objects.equals(javaMethodName, mnPattern);
            }
        };

        // 模块事件监听
        int watchId = moduleEventWatcher.watch(exceptionFilter, event ->
{
            switch (event.type) {
                case BEFORE:
                    ProcessController.throwsImmediately(new
RuntimeException("breakdown it by demo-break"));
                    break;
                case RETURN:
                    printer.println(String.format(
                            "return on [%s#%s]",
                            cnPattern,
                            mnPattern
                    ));
                    break;
            }
```

```
        }, new ProgressPrinter(printer), Type.BEFORE, Type.RETURN);

        // -------- 等待结束 --------
        try {
            printer.println(String.format(
                    "exception on [%s#%s] exception: .\nPress CTRL_C abort
it!",
                    cnPattern,
                    mnPattern
            ));
            printer.waitingForBroken();
        } finally {
            moduleEventWatcher.delete(watchId);
        }
    }

}
```

此模块代码实现比较简单。通过模块生命周期接口，在模块生命周期阶段打印日志信息，了解模块生命周期阶段。

`public void breakdown(final Map<String, String> param, final PrintWriter writer)` 方法通过用户自定义命令的方式，在方法入口处拦截方法直接跑出异常。

其中实现的核心 `com.alibaba.jvm.sandbox.api.resource.ModuleEventWatcher`，通过事件监听器，监听需要处理事件，实现 AOP 能力。

将模块打包 `mvn clean package`，将打包好的 `jar` 放入 `sandbox` 安装目录 `./sandbox/sandbox-module/` 下。

启动 `sandbox` ， `./sandbox.sh -p <PID>` 其中 `PID` 为绑定目标 Java 进程 ID。具体命令可参考[官方文档](http://cxyroad.com/"https://github.com/alibaba/jvm-sandbox/wiki/USER-INSTALL-and-CONFIG"):

最终效果

执行命令：

`./sandbox.sh -p 6521 -d 'demo-breakdown/breakdown?class=com.springboot.demo02.web.service.UserService&method=combinePrefix'`

方法执行结果：

```
java.lang.RuntimeException: breakdown it by demo-break
  at com.col.jvm.demo.module.DemoBreakdownModule.lambda$breakdown$0(DemoBreakdownModule.java:97) ~[na:na]
  at com.alibaba.jvm.sandbox.core.enhance.weaver.EventListenerHandler.handleEvent(EventListenerHandler.java:116) ~[na:na]
  at com.alibaba.jvm.sandbox.core.enhance.weaver.EventListenerHandler.handleOnBefore(EventListenerHandler.java:350) ~[na:na]
  at java.com.alibaba.jvm.sandbox.spy.Spy.spyMethodOnBefore(Spy.java:164) ~[na:na]
  at com.springboot.demo02.web.service.UserService.combinePrefix(UserService.java) ~[classes/:na]
```

模块自定义命令

`-d`：模块自定义命令

自定义命令使用方式：

* `./sandbox.sh -p <PId> -d <MODULE-ID>/<COMMAND-NAME>[?<PARAM1=VALUE1>[&PARAM2=VALUE2]]`

自定义命令实现方式：

在模块中可以通过对方法标记 `@Command` 注释，让 `sandbox.sh` 可以将自定义命令传递给被标记的方法。

```
@MetaInfServices(Module.class)
@Information(id = "demo-exception", version = "0.0.1")
public class DemoExceptionModule extends ParamSupported
implements Module {
    /**
     * 自定义用户命令
     * @param param
     * @param writer
     */
    @Command("exception")
    public void breakdown(final Map<String, String> param, final
PrintWriter writer) {
        final Printer printer = new ConcurrentLinkedQueuePrinter(writer);
        // -------- 参数解析 --------
        final String cnPattern = getParameter(param, "class");
        final String mnPattern = getParameter(param, "method");
        // ......
    }
}
```

此时对应过来的`-d`命令参数为：`-d 'demo-exception/exception?class=<CLASS>&method=<METHOD>'` 即可指定到这个方法。

JVM-Sandbox 应用模块目录

沙箱拥有两个加载模块的目录，用途各自不一。

* **sandbox/module/**

沙箱系统模块目录，由配置项 `system_module` 进行定义。用于存放沙箱通用的管理模块，比如用于沙箱模块管理功能的module-mgr模块，未来的模块运行质量监控模块、安全校验模块也都将存放在此处，跟随沙箱的发布而分发。

系统模块不受**刷新(-f)** 、**强制刷新(-F)** 功能的影响，只有**容器重置

(-S)** 能让沙箱重新加载系统模块目录下的所有模块。
* **sandbox/sandbox-module/**

沙箱用户模块目录，由配置项`user_module`进行定义。一般用于存放用户自研的模块。自研的模块经常要面临频繁的版本升级工作，当需要进行模块动态热插拔替换的时候，可以通过**刷新(-f)** 或**强制刷新(-F)** 来完成重新加载。

用户将自定模块打包后的 `jar` 文件，放入 sandbox 安装路径下 `sandbox-module` 目录。

使用场景
---

0. **性能监控与调优**

* **方法级性能监控**：监控方法的执行时间，帮助识别性能瓶颈。
* **资源使用监控**：监控内存、CPU 等资源的使用情况，优化资源分配。
1. **故障诊断**

* **实时异常捕获**：捕获运行时异常并分析异常堆栈，快速定位问题。
* **日志分析**：实时收集和分析日志数据，帮助诊断问题。
2. **安全审计**

* **敏感操作监控**：监控敏感操作（如数据库访问、文件读写）以确保安全性。
* **行为审计**：记录和分析用户行为，检测异常活动。
3. **开发调试**

* **动态调试**：在不停止应用的情况下，插入调试代码，查看应用的内部状态。
* **代码热替换**：在运行时修改和替换代码，进行快速测试和验证。
4. **系统分析**

* **依赖分析**：分析类和方法之间的调用关系，了解系统的依赖结构。
* **调用链追踪**：追踪请求的完整调用链，识别复杂问题的根本原因。

优点

0. **实时性**

* 能够实时监控和诊断运行中的应用，不需要重启或停止服务。
1. **灵活性**

* 插桩点和监控逻辑可以动态配置和修改，适应不同的需求和场景。
2. **低侵入性**

* 通过字节码操作实现插桩，对原有代码和业务逻辑的侵入性较低。
3. **丰富的功能**

* 提供丰富的监控和诊断功能，覆盖性能监控、故障诊断、安全审计等多个方面。

缺点

0. **性能开销**

* 虽然 JVM-Sandbox 设计上尽量降低对目标 JVM 的性能影响，但插桩和事件处理仍然会带来一定的性能开销，特别是在高并发或性能敏感的应用中。
1. **复杂性**

* 由于涉及字节码操作和动态插桩，JVM-Sandbox 的使用和配置相对复杂，需要一定的学习和理解成本。
2. **兼容性问题**

* 可能会与某些 JVM 特性或其他字节码操作工具（如 APM 工具、性能分析工具）产生兼容性问题。
3. **调试难度**

* 由于插桩代码和业务代码是动态结合的，调试插桩逻辑和排查问题可能会比较困难。

总结

JVM-Sandbox 通过动态插桩和事件监听，提供了强大的实时监控和诊断能力，适用于性能监控、故障诊断、安全审计和开发调试等多个场景。尽管其性能开销和复杂性是使用中的主要挑战，但其带来的实时性和灵活性使其成为 Java 应用监控和诊断的有力工具。

参考资料

* [github.com/alibaba/jvm...](http://cxyroad.com/”https://github.com/alibaba/jvm-sandbox/wiki”)
* juejin.cn/post/727593...

推荐阅读

Kubernetes Informer基本原理

JDK17 与 JDK11 特性差异浅谈

业务分析师眼中的数据中台

政采云大数据权限系统设计和实现

JDK11 与 JDK8 特性差异浅谈

招贤纳士

政采云技术团队（Zero），Base 杭州，一个富有激情和技术匠心精神的成长型团队。规模 500 人左右，在日常业务开发之外，还分别在云原生、区块链、人工智能、低代码平台、中间件、大数据、物料体系、工程平台、性能体验、可视化等领域进行技术探索和实践，推动并落地了一系列的内部技术产品，持续探索技术的新边界。此外，团队还纷纷投身社区建设，目前已经是 google flutter、scikit-learn、Apache Dubbo、Apache Rocketmq、Apache Pulsar、CNCF Dapr、Apache DolphinScheduler、alibaba Seata 等众多优秀开源社区的贡献者。

如果你想改变一直被事折腾，希望开始折腾事；如果你想改变一直被告诫需要多些想法，却无从破局；如果你想改变你有能力去做成那个结果，却不需要你；如果你想改变你想做成的事需要一个团队去支撑，但没你带人的位置；如果你想改变本来悟性不错，但总是有那一层窗户纸的模糊……如果你相信相信的力量，相信平凡人能成就非凡事，相信能遇到更好的自己。如果你希望参与到随着业务腾飞的过程，亲手推动一个有着深入的业务理解、完善的技术体系、技术创造价值、影响力外溢的技术团队的成长过程，我觉得我们该聊聊。任何时间，等着你写点什么，发给 [zcy-tc@cai-inc.com](http://cxyroad.com/”mailto:zcy-tc@cai-inc.com”)

公众号

文章同步发布，政采云技术团队公众号，欢迎 ![文章顶部.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/aaafc13f1d1e4145a3d6e94d0987e2c3~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=4722&h=696&s=276733&e=png&b=fafcff)

原文链接: https://juejin.cn/post/7381412483040624659