

ChaosBlade

混沌工程

ChaosBlade

chaosblade-exec-jvm

Some Important

扩展插件的步骤

1. 在chaosblade-exec-plugin模块下新建子模块
2. 编写enhancer类
3. 编写PointCut类
4. 编写spec类
5. 编写ActionExecutor类
6. 自定义Plugin

chaosblade调用chaosblade-exec-jvm

JVM故障场景

plugin-jvm

delay

return

script

cpufullload

OutOfMemoryError

实现原理

根据不同区注入

CodeCacheFilling

实现原理

throwCustomException

FullGC

混沌工程

混沌工程，是一种提高技术架构弹性能力的复杂技术手段。Chaos工程经过实验可以确保系统的可用性。混沌工程旨在将故障扼杀在襁褓之中，也就是在故障造成中断之前将它们识别出来。通过主动制造故障，测试系统在各种压力下的行为，识别并修复故障问题，避免造成严重后果。

混沌工程师通过应用一些经验探索的原则，来学习观察系统是如何反应的。这就跟科学家做实验去学习物理定律一样，混沌工程师通过做实验去了解系统。


应用混沌工程能提升整个系统的弹性。通过设计并且进行混沌实验，我们可以了解到系统脆弱的一面，在还没出现对用户造成伤害之前，我们就能主动发现这些问题。

混沌工程和测试是有区别的。虽然混沌工程跟传统测试通常都会共用很多测试工具的，譬如都会使用错误注入工具，但混沌工程是通过实践对系统有更新的认知，而传统测试则是使用特定方式对某一块进行特定测试。譬如在传统测试里面，我们可以写一个断言，我们给定特定的条件，产生一个特定的输出，如果不满足断言条件，测试就出错了，这个其实是具有很明确的特性。但混沌工程是试验，而试验会有怎样的新信息生成，我们是不确定的。

在开始应用混沌工程之前，我们必须确保系统是弹性的，也就是当出现了系统错误我们的整个系统还能正常工作。如果不能确保，我们就需要先考虑提升整个系统的健壮性了，因为混沌工程主要是用来发现系统未知的脆弱一面的，如果我们知道应用混沌工程能导致显而易见的问题，那其实就没必要应用了。

ChaosBlade

chaosblade-exec-jvm



chaosblade-exec-jvm
ecutor for chaos experiments on Java
对 Java 应用实施混沌实验的
行器

10 Issues · 344 Stars · 170 Forks

chaosblade-exec-jvm/design.md at master · chaosblade-io/chaosblade-e...

Chaosblade executor for chaos experiments on Java applications (对 Java 应用实施混沌实验的 cha...

GitHub

Chaosblade-exec-jvm通过JavaAgent attach方式来实现类的transform注入故障，底层使用 [jvm-sandbox](#) 实现，通过插件的可拔插设计来扩展对不同java应用的支持，可以很方便的扩展插件。

chaosblade-exec-jvm集成了Jvm-SandBox，在make编译之后，作为Jvm-SandBox的一个module。chaosblade-exec-jvm基于Jvm-SandBox的事件监听机制，拓展了插件的设计。

此项目是 chaosblade 项目的一个底层混沌实验执行器，对运行在 JVM 平台上的应用或组件实施故障注入等混沌实验。

四个子模块：

- chaosblade-exec-bootstrap: jar 包启动器，执行入口。目前支持的 chaosblade-exec-bootstrap-jvmsandbox 是基于 jvm-sandbox 实现的，通过 java agent 机制动态加载 agent jar。
- chaosblade-exec-common: 是工程的通用模块，其他子模块都依赖于此模块，其包含故障注入模型的定义、类方法 AOP 定义等。
- chaosblade-exec-plugin: 支持混沌实验插件，比如 dubbo、mysql、servlet 等。jvm 插件是对随意指定的类和方法做混沌实验的插件。如果对插件进行扩展，也是在此模块下追加。具体的如何扩展插件，后续会有文章专门说明。
- chaosblade-exec-service: 包含两大功能：一是 chaosblade 通信协议解析，用来解析执行 chaosblade 下发的混沌实验指令；另一个是生成 jvm.spec.yaml 文件，此文件的内容定义了 JVM 相关的混沌实验场景，chaosblade 工具通过读取此文件，显示对 JVM 应用所支持的混沌实验场景。

Some Important

DirectlyInjectionAction

如果ModelSpec是PreCreateInjectionModelHandler类型，且ActionSpec的类型是DirectlyInjectionAction类型，将直接进行故障能力注入，比如JvmOom故障能力，ActionSpec的类型不是DirectlyInjectionAction类型，将加载插件。

	DirectlyInjectionAction	Not DirectlyInjectionAction
PreCreateInjectionModelHandler (ModelSpec)	直接进行故障能力注入	加载插件
PreDestroyInjectionModelHandler (ModelSpec)	停止故障能力注入	卸载插件

故障能力注入

故障能力注入的方式，最终都是调用ActionExecutor执行故障能力。

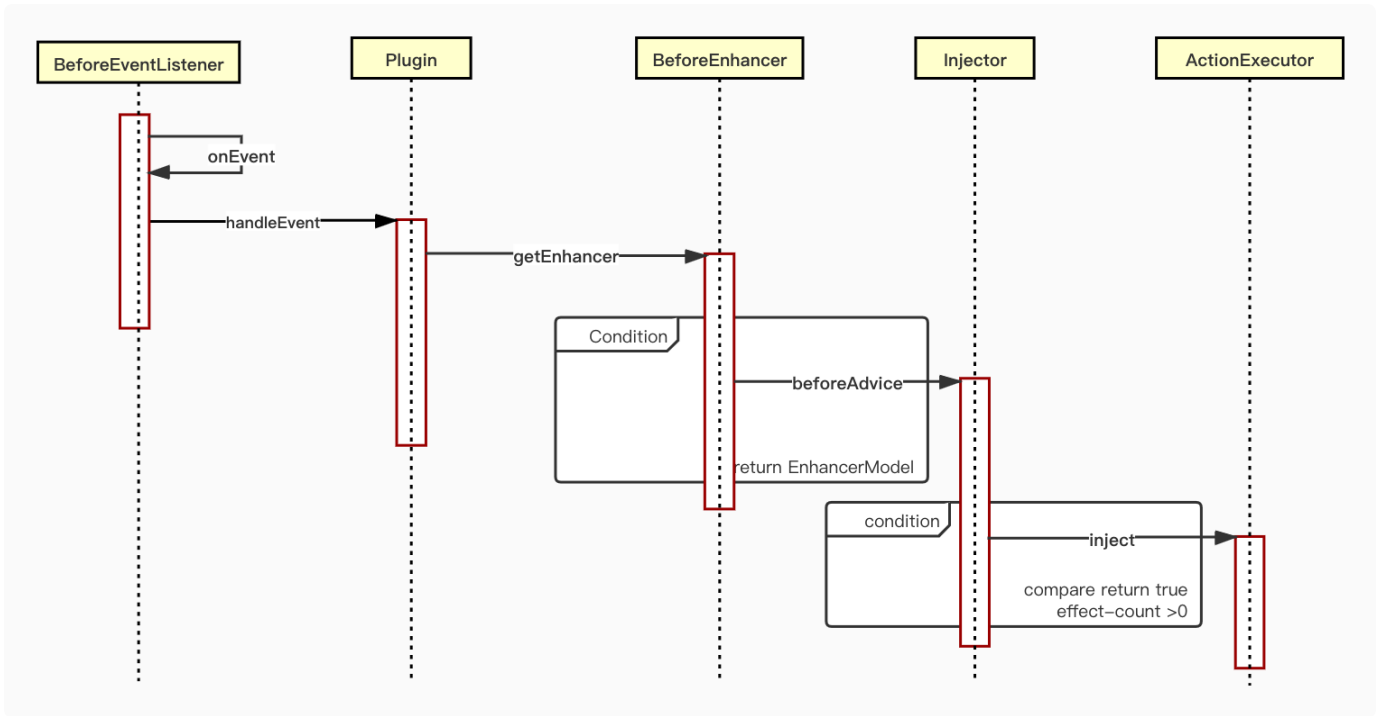
- 通过Inject注入。
- DirectlyInjectionAction直接注入，直接注入不经过Inject类调用阶段，如果jvm oom等。

DirectlyInjectionAction直接注入不经过Enhancer参数包装匹配直接到故障触发ActionExecutor执行阶段，如果是Inject注入此时因为StatusManager已经注册了实验，当事件再次出发后

ManagerFactory.getStatusManager().expExists(targetName)的判断不会被中断，继续往下走，到了自定义的Enhancer，在自定义的Enhancer里面可以拿到原方法的参数、类型等，甚至可以反射调原类型的其他方法，这样做风险较大，一般在这里往往是取一些成员变量或者get方法等，用户后续参数匹配。

扩展插件的步骤

一个插件包括Plugin、PointCut、Enhancer、ModelSpec、ActionSpec、ActionExecutor



1. 在chaosblade-exec-plugin模块下新建子模块

如chaosblade-exec-plugin-servlet

2. 编写enhancer类

jvm-sandbox 提供了 EventListener 接口给自定义 module 去实现事件的自定义动作（这里是故障注入），通过 onEvent 方法触发事件，chaosblade-exec-jvm 定义了 BeforeEnhancer 和 AfterEnhancer 两种类，分别对应 jvm-sandbox 的 before 和 return 事件来完成方法执行前注入故障和方法返回后注入故障。

由于不同的故障注入操作对应不同的类、方法、参数、动作等，chaosblade-exec-jvm 将这些信息抽象成了一个 EnhancerModel 作为操作媒介，并将故障注入操作实现为一个静态方法 Injector.inject(enhancerModel)，不同的故障注入类型自定义 enhancer 去继承

BeforeEnhancer、AfterEnhancer 并返回 EnhancerModel，再去执行 inject。例如 ServletEnhancer，获取 ContextPath、RequestURI、Method 等，将获取到的参数放到 MatcherModel，返回 EnhancerModel，Inject 阶段会与输入的参数做比对。

故障注入操作基本都是响应 before 事件，所以基本都是继承 BeforeEnhancer 去实现方法调用前的 AOP 增强操作，有一个特殊的是 jvm 故障注入类型如 delay、return 等操作是可以定义方法执行完成返回前注入故障的，所以特殊地存在一个 MethodEnhancer 同时实现了 beforeAdvice 和 afterAdvice 支持部分 jvm 场景的故障注入操作。

获取ContextPath、RequestURI、Method等，将获取到的参数放到MatcherModel，返回 EnhancerModel，Inject阶段会与输入的参数做比对。

```

1 public class ServletEnhancer extends BeforeEnhancer {
2
3     @Override
4     public EnhancerModel doBeforeAdvice(ClassLoader classLoader,
5                                         String className,
6                                         Object object,
7                                         Method method,
8                                         Object[] methodArguments
9                                         ,String targetName) throws Excepti
10     on {
11         Object request = methodArguments[0];
12         // 执行被增强类的方法，获取一些需要的值
13         String queryString = ReflectUtil.invokeMethod(request, "getQuerySt
14         ring", new Object[] {}, false);
15         String contextPath = ReflectUtil.invokeMethod(request, "getContext
16         Path", new Object[] {}, false);
17         String requestURI = ReflectUtil.invokeMethod(request, "getRequestU
18         RI", new Object[] {}, false);
19         String requestMethod = ReflectUtil.invokeMethod(request, "getMetho
20         d", new Object[] {}, false);
21
22         String requestPath = StringUtils.isBlank(contextPath) ? requestURI
23         : requestURI.replaceFirst(contextPath, "");
24
25         MatcherModel matcherModel = new MatcherModel();
26         matcherModel.add(ServletConstant.QUERY_STRING_KEY, queryString);
27         matcherModel.add(ServletConstant.METHOD_KEY, requestMethod);
28         matcherModel.add(ServletConstant.REQUEST_PATH_KEY, requestPath);
29
30         return new EnhancerModel(classLoader, matcherModel);
31     }
32 }

```

不同的通知可继承不同的类

- beforeAdvice: 继承com.alibaba.chaosblade.exec.common.aop.BeforeEnhancer
- afterAdvice: 继承com.alibaba.chaosblade.exec.common.aop.AfterEnhancer

3. 编写PointCut类

定义故障注入操作执行的AOP 切点，可以拦截类和方法。SandboxModule onActive() 事件触发 Plugin 加载后，SandboxEnhancerFactory 创建 filter，filter 内部通过PointCut 的 ClassMatcher 和 MethodMatcher 过滤。

在下面的例子中，拦截了spring的FrameworkServlet、webx的WebxFrameworkFilter、及父类为HttpServletBean或HttpServlet的子类。拦截了doGet、doPost、doDelete、doPut、doFilter方法。

```
1 // 继承com.alibaba.chaosblade.exec.common.aop.PointCut
2 public class ServletPointCut implements PointCut {
3
4     // 拦截的类名
5     public static final String SPRING_FRAMEWORK_SERVLET = "org.springframework.web.servlet.FrameworkServlet";
6     public static final String ALIBABA_WEBX_FRAMEWORK_FILTER = "com.alibaba.citrus.webx.servlet.WebxFrameworkFilter";
7     public static final String SPRING_HTTP_SERVLET_BEAN = "org.springframework.web.servlet.HttpServletBean";
8     public static final String HTTP_SERVLET = "javax.servlet.http.HttpServlet";
9
10    public static Set<String> enhanceMethodSet = new HashSet<String>();
11    public static Set<String> enhanceMethodFilterSet = new HashSet<String>();
12
13    // 拦截的方法名
14    static {
15        enhanceMethodSet.add("doGet");
16        enhanceMethodSet.add("doPost");
17        enhanceMethodSet.add("doDelete");
18        enhanceMethodSet.add("doPut");
19        enhanceMethodFilterSet.add("doFilter");
20    }
21
22    // 类匹配
23    @Override
24    public ClassMatcher getClassMatcher() {
25        OrClassMatcher orClassMatcher = new OrClassMatcher();
26        orClassMatcher.or(new NameClassMatcher(SPRING_FRAMEWORK_SERVLET)).
27    or(
28        new NameClassMatcher(ALIBABA_WEBX_FRAMEWORK_FILTER)).or(
29        new SuperClassMatcher(SPRING_HTTP_SERVLET_BEAN, HTTP_SERVLET))
30    ;
31        return orClassMatcher;
32    }
33
34    // 类方法匹配
35    @Override
36    public MethodMatcher getMethodMatcher() {
37        AndMethodMatcher andMethodMatcher = new AndMethodMatcher();
38        OrMethodMatcher orMethodMatcher = new OrMethodMatcher();
39        orMethodMatcher.or(new ManyNameMethodMatcher(enhanceMethodSet)).or(
40        new ManyNameMethodMatcher
```



```

38         (enhanceMethodFilterSet));
39         andMethodMatcher.and(orMethodMatcher).and(new ParameterMethodMatch
40 er(1, ParameterMethodMatcher.GREAT_THAN));
41         return andMethodMatcher;
42     }

```

4. 编写spec类

例如命令[./blade create servlet delay --time=3000]对于命令而言主要分为phases、target、action、flag，phases相对插件而言不需要很强的灵活性，因此由chaosblade-exec-service模块管理，对于自定义插件只需要扩展ModelSpec(也就是target，实验靶点，发生实验的组件，如容器、应用框架<Dubbo、Redis、Zookeeper>等)、action(实验模拟的具体场景)、flag。

command

```

      phases  ModelSpec ActionSpec FlagSpec
      ┌───┐  ┌───┐  ┌───┐  ┌───┐
./blade create servlet delay --time=3000

```

ModelSpec

ModelSpec的getTarget()方法对于命令中target部分的名称，如servlet、dubbo等，createNewMatcherSpecs()方法添加ModelSpec下的FlagSpec。

createNewMatcherSpecs()包含很多flagSpec，那么ModelSpec支持的命令如下：

```
./blade create servlet --method=post --requestpath=/index --contextpath=/shop
```

```
1 public class ServletModelSpec extends FrameworkModelSpec {
2
3     public ServletModelSpec() {
4         addModifyHttpAction();
5         addActionExample();
6     }
7
8     private void addModifyHttpAction() {
9         ModifyHttpCodeActionSpec modifyHttpCodeActionSpec = new ModifyHttp
10 CodeActionSpec();
11         modifyHttpCodeActionSpec.addMatcherDesc(new ServletQueryStringMatc
12 herSpec());
13         modifyHttpCodeActionSpec.addMatcherDesc(new ServletQueryStringRege
14 xPatternMatcherSpec());
15         modifyHttpCodeActionSpec.addMatcherDesc(new ServletMethodMatcherSp
16 ec());
17         modifyHttpCodeActionSpec.addMatcherDesc(new ServletRequestPathMatc
18 herSpec());
19         modifyHttpCodeActionSpec.addMatcherDesc(new ServletRequestPathRege
20 xPatternMatcherSpec());
21         modifyHttpCodeActionSpec.setLongDesc("Servlet return custom statu
22 s code(4xx,5xx)");
23         modifyHttpCodeActionSpec.setExample("# Request to http://localhos
24 t:8080/dubbodemo/servlet/path?name=bob return 404\n" +
25         "blade c servlet mc --requestpath /dubbodemo/servlet/path
26 --code=404\n\n");
27         // 添加actionSpec
28         addActionSpec(modifyHttpCodeActionSpec);
29     }
30
31     private void addActionExample() {
32         List<ActionSpec> actions = getActions();
33         for (ActionSpec action : actions) {
34             if (action instanceof DelayActionSpec) {
35                 action.setLongDesc("Servlet delay experiment, support serv
36 let springMVC webX");
37                 action.setExample("# Request to http://localhost:8080/dubb
38 odemo/servlet/path?name=bob delays 3s, effect two requests\n" +
39                 "blade c servlet delay --time 3000 --requestpath /
40 dubbodemo/servlet/path --effect-count 2\n\n" +
41                 "# The request parameter is name=family, the dela
42 y is 2 seconds, the delay time floats up and down for 1 second, the impac
43 t range is 50% of the request, and the debug log is turned on to troublesh
44 oot the problem\n" +
```

```

31         "blade c servlet delay --time 2000 --offset 1000 -
32 -querystring name=family --effect-percent 50 --debug");
33     } else if (action instanceof ThrowCustomExceptionActionSpec) {
34         action.setLongDesc("Servlet throw custom exception experim
35 ent, support servlet springMVC webX");
36         action.setExample("# Request to http://localhost:8080/dubb
37 odemo/hello throws custom exception, effect three requests\n" +
38         "blade c servlet throwCustomException --exception
39 org.springframework.beans.BeansException --exception-message mock-beans-ex
40 ception --requestpath /hello --effect-count 3");
41     }
42 }
43
44 // modelSpec的名字
45 @Override
46 public String getTarget() {
47     return "servlet";
48 }
49
50 @Override
51 public String getShortDesc() {
52     return "java servlet experiment";
53 }
54
55 @Override
56 public String getLongDesc() {
57     return "Java servlet experiment, support path, query string, conte
58 xt path and request method matcher";
59 }
60
61 @Override
62 public String getExample() {
63     return "servlet --requestpath /hello --method post";
64 }
65
66 // 添加ModelSpec下的FlagSpec
67 @Override
68 protected List<MatcherSpec> createNewMatcherSpecs() {
69     ArrayList<MatcherSpec> matcherSpecs = new ArrayList<MatcherSpec>()
70 ;
71     matcherSpecs.add(new ServletContextPathMatcherSpec()); // 对应flags
72 pec的--contextpath
73     matcherSpecs.add(new ServletQueryStringMatcherSpec());
74     matcherSpecs.add(new ServletMethodMatcherSpec()); // 对应flagspec
75 的--method
76     matcherSpecs.add(new ServletRequestPathMatcherSpec()); // 对应flags
77 pec的--requestpath

```

```

69         return matcherSpecs;
70     }
71 }

```

具体实现方式如下：

- 实现com.alibaba.chaosblade.exec.common.model.ModelSpec
- 继承BaseModelSpec，实现了对CreateHandler阶段的输入参数的校验
- 继承FrameworkModelSpec包含DelayActionSpec、ThrowCustomExceptionActionSpec，默认实现了不同target的延迟侵入和异常侵入。

ActionSpec

例如DelayActionSpec，支持参数 --time=xx --offset=xx

```
./blade create servlet --method=post delay --time=3000 ----offset=10
```

ActionSpec的getName()方法对应命令中action部分的名称，如delay、throwCustomException等，ActionSpec由ModelSpec的addActionSpec()方法添加，可以有以下方式实现：

- 实现com.alibaba.chaosblade.exec.common.model.action.ActionSpec
- 继承BaseActionSpec，实现了对CreateHandler阶段的输入参数的校验

```
1 public class DelayActionSpec extends BaseActionSpec {
2
3     private static TimeFlagSpec timeFlag = new TimeFlagSpec();
4     private static TimeOffsetFlagSpec offsetFlag = new TimeOffsetFlagSpec(
5 );
6
7     public DelayActionSpec() {
8         //添加 actionExecutor
9         super(new DefaultDelayExecutor(timeFlag, offsetFlag));
10    }
11
12    @Override
13    public String getName() {
14        return "delay";
15    }
16
17    @Override
18    public String[] getAliases() {
19        return new String[0];
20    }
21
22    @Override
23    public String getShortDesc() {
24        return "delay time";
25    }
26
27    @Override
28    public String getLongDesc() {
29        return "delay time...";
30    }
31
32    @Override
33    public List<FlagSpec> getActionFlags() {
34        return Arrays.asList(timeFlag, offsetFlag);
35    }
36
37    @Override
38    public PredicateResult predicate(ActionModel actionModel) {
39        if (StringUtil.isBlank(actionModel.getFlag(timeFlag.getName()))){
40            return PredicateResult.fail("less time argument");
41        }
42        return PredicateResult.success();
43    }
44 }
```

FlagSpec

上面ModelSpec中createNewMatcherSpecs中添加的ServletContextPathMatcherSpec、ServletQueryStringMatcherSpec、ServletMethodMatcherSpec、ServletRequestPathMatcherSpec等类都是FlagSpec类。

例如TimeOffsetFlagSpec，支持--offset=10 的参数

```
Java | 复制代码

1 public class TimeOffsetFlagSpec implements FlagSpec {
2     @Override
3     public String getName() {
4         return "offset";
5     }
6
7     @Override
8     public String getDesc() {
9         return "delay offset for the time";
10    }
11
12    @Override
13    public boolean noArgs() {
14        return false;
15    }
16
17    @Override
18    public boolean required() {
19        return false;
20    }
21 }
```

FlagSpec的getName()方法对应命令中flag部分的名称，如--time、--offset等

- 实现com.alibaba.chaosblade.exec.common.model.FlagSpec，由ActionSpec的getFlagSpec方法添加
- 继承com.alibaba.chaosblade.exec.common.model.matcher.MatcherSpec，由ActionSpec的addActionSpec添加，CreateHandler阶段会做参数校验
- 继承com.alibaba.chaosblade.exec.common.model.matcher.BasePredicateMatcherSpec

5. 编写ActionExecutor类

ActionExecutor执行器作为BaseActionSpec的构造参数，ActionExecutor可以自定义一些增强业务的操作，如修改方法的参数、篡改方法的返回值等。

```
Java | 复制代码

1 public interface ActionExecutor {
2
3     /**
4      * Run executor
5      *
6      * @param enhancerModel
7      * @throws Exception
8      */
9     void run(EnhancerModel enhancerModel) throws Exception;
10 }
```

实现ActionExecutor的接口，EnhancerModel里面可以拿到命令行输入的参数以及原始方法的参数，类型，返回值、异常，做一些增强业务操作。

```
Java | 复制代码

1 // 延迟多少毫秒
2 Long time = Long.valueOf(enhancerModel.getActionFlag("time"));
3 TimeUnit.MILLISECONDS.sleep(time);
```

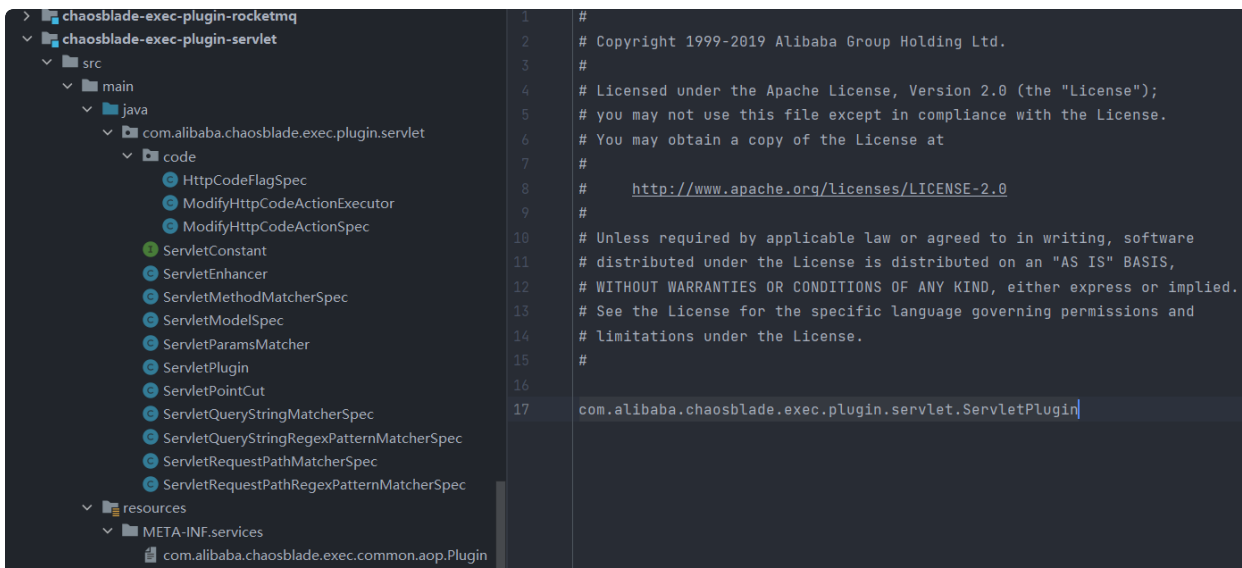
6. 自定义Plugin

继承com.alibaba.chaosblade.exec.common.aop.Plugin，自定义target名称，添加Enhancer、PointCut、ModelSpec即可，实现类需要全路径名复制到：resources/META-INF/services/com.alibaba.chaosblade.exec.common.aop.Plugin 挂载Agent，模块激活后plugin自动加载。

```

1 public class ServletPlugin implements Plugin {
2
3     @Override
4     public String getName() {
5         return "servlet";
6     }
7
8     @Override
9     public ModelSpec getModelSpec() {
10        return new ServletModelSpec();
11    }
12
13    @Override
14    public PointCut getPointCut() {
15        return new ServletPointCut();
16    }
17
18    @Override
19    public Enhancer getEnhancer() {
20        return new ServletEnhancer();
21    }
22 }

```



chaosblade调用chaosblade-exec-jvm

chaosblade-exec-jvm 的工程模块和打包后的文件包括 jvm.spec.yaml 和 chaosblade-java-agent-0.0.1-SNAPSHOT.jar 两个文件。第一个文件是 chaosblade 感知关于 jvm 应用所支持的混沌实验场景有哪些，第二个文件的作用是 chaosblade 通过此文件执行 jvm 应用所支持的混沌实验场景。

下载 chaosblade RELEASE 包，解压后，会看到两个文件夹，一个文件：

```

1  | bin
2  |   | jvm.spec.yaml
3  |   | tools.jar
4  | blade
5  | lib
6  | sandbox
7  | module
8  | chaosblade-java-agent-0.0.1.jar
```

- blade 文件是 chaosblade 工具的执行文件
- bin 目录是 chaosblade 工具所支持的混沌实验场景目录。此处的 jvm.spec.yaml 文件就是 chaosblade-exec-jvm 编译后生成的文件。
- lib 目录是第三方的依赖组件，目前只依赖了 jvm-sandbox，其 module 下的 chaosblade-java-agent-0.0.1.jar 文件是 chaosblade-exec-jvm 编译后生成 agent jar 包。

chaosblade 执行 JVM 应用混沌实验的流程是：

- blade 工具执行时，先读取 bin 目录下的 jvm.spec.yaml 文件，将场景注册到混沌实验场景列表中。
- 执行 blade prepare jvm --process <PROCESS NAME> 命令，会内部调用 jvm-sandbox flush 指令，加载其 module 下的 chaosblade-java-agent-0.0.1.jar agent 文件，修改混沌实验组件所匹配到的类。
- 执行 blade create dubbo xxx 等组件的混沌实验，会内部通过 HTTP 协议调用已加载的 agent，agent 会缓存下发的参数，然后匹配混沌实验组件所获取到的数据是否一致。如果一致，则调用对应的场景执行器，比如 DelayExecutor，ThrowExceptionExecutor，如果不一致，则不处理。
- 执行 blade destroy <UID> 命令，销毁所创建的实验，即 agent 会删除之前缓存的实验。
- 执行 blade revoke <UID> 命令，会内部调用 jvm-sandbox shutdown 指令，卸载所加载的 module，还原之前所修改的类，关闭 jvm-sandbox 自身，清除 jvm-sandbox 和 chaosblade-java-agent-0.0.1.jar 所加载的类。

以上是一次完整的关于 JVM 应用的混沌实验实施流程。

JVM故障场景

plugin-jvm



blade create jvm – chaosblade–help–zh–CN

chaosblade–help–zh–CN

jvm 本身相关场景，以及可以指定类，方法注入延迟、返回值、异常故障场景，也可以编写 groovy 和 java 脚本来实现复杂的场景。目前支持的场景如下：

delay

指定类方法调用延迟

return

指定类方法的返回值，仅支持基本类型、null 和 String 类型的返回值。

script

编写 java 或者 groovy 脚本实现复杂的故障场景，比如篡改参数、修改返回值、抛自定义异常等

cpufullload

指定 java 进程 CPU 满载，可以简写为 blade c jvm cfl

OutOfMemoryError

内存溢出场景，命令可以简写为：blade c jvm oom

实现原理

根据不同区注入

java.lang.OutOfMemoryError: Java heap space

创建 Heap的话分为Young, Old, 这块区域的oom是最好重现, 只需要不断的创建对象就可以, 如果内存使用达到了 Xmx或者Xmn所规定的大小, 并且gc回收不了,就会触发oom错误。

检查 • 可以通过 jmap -heap pid 来查看当前堆占用情况是否到了100% • 可以通过jstat -gcutil pid 来查看是否发生了gc,因为会一直创建新的对象, 所以会频繁触发gc操作

恢复 当演练终止后, 会停止产生新的对象, 但此时不一定heap就恢复了, 因为恢复需要触发gc才可以进行回收,当然也可以通过手动调用 System.gc()来强行触发gc, 但是如果你的启动参数里面有 -XX:+DisableExplicitGC 那么这个命令就无法生效了。

注意 触发OOM的时候可能会导致进程被操作系统所kill, 这个原因是因为你的Xmx设置的不合理, 比如操作系统内存只有3G, 但是你Xmx会设置了3G甚至更多, 那么就会因为系统内存不足, 而被os kill掉进程, 所以这里务必要注意Xmx大小

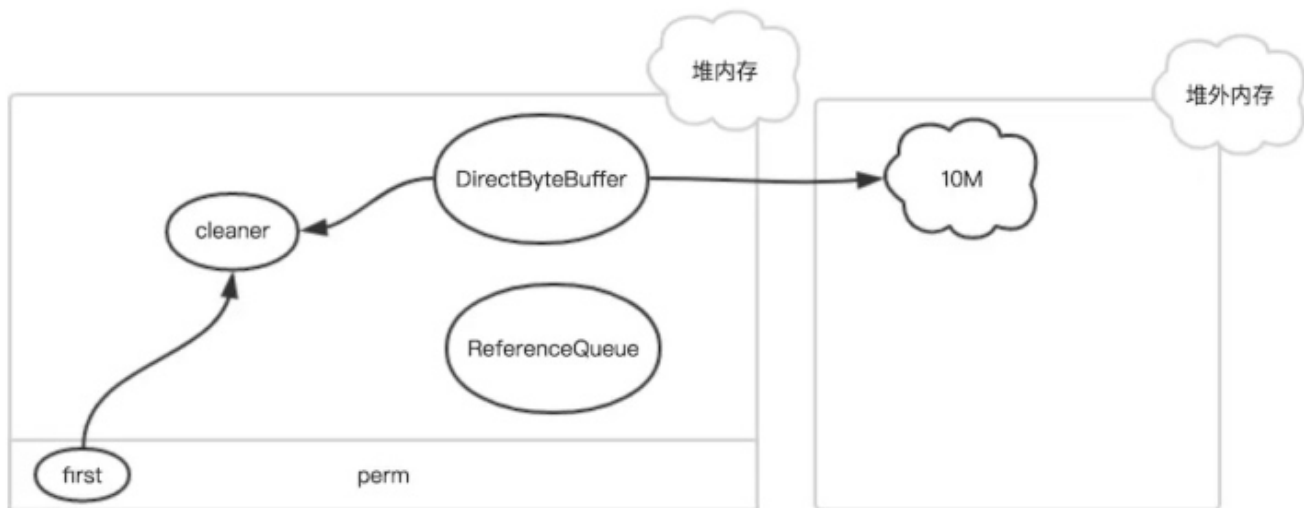
java.lang.OutOfMemoryError: Metaspace

创建 Metaspace可以通过不断的加载类对象来创建, 当大小超过了 -XX:MaxMetaspaceSize 并且无法进行gc回收就会抛出 oom错误了

检查 • 可以通过jstat -gcutil pid 来查看 M区的使用情况以及gc的次数

恢复 类对象的回收条件在jvm里面比较苛刻, 需要满足很多条件, 就算满足了条件, 触发gc了也不一定回收,只要有下面任何一个条件就无法被回收. • objects of that class are still reachable. • the Class object representing the class is still reachable • the ClassLoader that loaded the class is still reachable • other classes loaded by the ClassLoader are still reachable 因此最好的办法就是重启应用.

java.lang.OutOfMemoryError: Direct buffer memoryDirectBuffer



创建: 堆外内存可以直接通过ByteBuffer.allocateDirect 来产生,并且会一直消耗系统内存.

检查：因为堆外内存不属于堆里面，所以你通过jmap命令很难发现，但是可以通过 `jstat -gcutil pid` 来查看，如果频发出发了fullgc，但是e,O,M区都没发生变化，那就是进行堆外内存回收 • 可以通过`free -m` 查看内存使用情况

注意 同样，如果没有设置最大堆外内存大小，同样会因为OS的memory耗尽而导致进程被杀，所以需要配置比如下面的参数: `-XX:MaxDirectMemorySize=100M`

CodeCacheFilling

CodeCache主要用于存放native code，其中主要是JIT编译后的代码。被JIT编译的一般都是“热代码”，简单说就是调用频率比较高的代码，JIT编译后，代码的执行效率会变高，CodeCache满会导致JVM关闭JIT编译且不可再开启，那么CodeCache满会引起系统运行效率降低，导致系统最大负载下降，当系统流量较大时，可表现为RT（1个请求完成的时间）增高、QPS（1秒内完成的请求数量）下降等。命令可以简写为：`blade c jvm ccf`

实现原理

由于CodeCache主要存放JIT编译的结果，所以填充CodeCache分为两步，第一步是生成用于触发JIT编译的class，方式是通过动态编译生成大量的class；第二步是编译后生成的class进行实例化和频繁调用（“加热”），直到触发JIT编译后进入CodeCache区。通过这样方式不停的填充CodeCache，直到JIT编译关闭

throwCustomException

指定类方法抛自定义异常，命令可以简写为 `blade c jvm tce`

FullGC