BTrace 简明使用手册

文档信息

文件状态： ［√］草稿 ［ ］正式发布 ［ ］正在修改	当前版本：	0.1
	作　　者：	张伟杰
	完成日期：	

修订记录

编号	修订内容简述	修订日期	版本	修订人
1	初稿	2015/12/25	0.1	张伟杰

Index

一、BTrace 基础概念

1.1 BTrace 是什么

BTrace 是一款 JVM 监控工具，它允许用户使用 java 语言编写 trace 代码，并将这些代码动态注入到 JVM 的指定位置，以执行灵活和强大的信息收集、分析、统计工作。

1.2 BTrace 的优势

通过字节码注入，用户可以在不重启程序的情况下，实时获取程序的运行状态，极大的方便了 java 应用的测试和调试。

1.3 怎么启动 BTrace

1. 安装 BTrace，下载 tar.gz 包，在服务器上解压即可运行，最新版本为 1.2.5.1。
2. 确认需要监控的 java 应用，获取进程 PID。
3. 使用 java 语言写一个 BTrace 脚本，如 Demo.java。
4. 进入$BTRACE_HOME/bin 目录
   执行 ./btrace [目标进程的 PID] Demo.java
   Tips
   如果被监控的目标 jvm 是运行在 openjdk 下，启动 BTrace 脚本会报错：
   Connection refused
   为了能顺利执行脚本，被监控的目标 JVM 需要使用 orace 的 jdk 运行，同样，我们在运行 btrace 前，需要 export JAVA_HOME=[oracle 版本的 jdk 路径，经测试 1.6、1.7 均可，最好可目标 JVM 使用的 jdk 版本一致]。

1.4 编写第一个 BTrace 脚本

1.4.1 准备一个被监控的 java 进程

要演示 BTrace 的注入效果，必须先有一个被监控的 java 进程。我们编写一个很简单的 java 程序作为被监控的目标，后续的演示中，都将使用这个程序。

```
package com.netease.qa.btrace;
import java.io.IOException;
import java.util.Random;

public class Demo1 {
```

```
        private String name;

        public Demo1(String name) {
                super();
                this.name = name;
        }

        public String toString(){
                return this.name;
        }

        public int add(int a, int b){
                Test test = new Test();
                int result = 0;
                try {
                        result = test.add(a, b);
                } catch (IOException e) {
                        e.printStackTrace();
                }
                return result;
        }

        public static void main(String [] args){
                Random random = new Random();
                Demo1 demo = new Demo1("this is a Demo1 instace");
                while(true){
                        int a = random.nextInt(100);
                        int b = random.nextInt(100);
                        int c = demo.add(a, b);
                        System.out.println("a:" + a);
                        System.out.println("b:" + b);
                        System.out.println("a+b:" + c);
                        try {
                                Thread.sleep(2000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}

class Test{
        public int add(int a, int b) throws IOException{
```

```
                    if(a>50 && a<80)
                            throw new IOException("this is a exception!");
                    return a + b;
            }
}
```

编写好这个 java 类文件后，拷贝至服务器，执行

```
javac com.netease.qa.btrace.Demo1.java
```

```
java com.netease.qa.btrace.Demo1
```

启动程序 main 函数，程序持续不间断运行，每隔 2000ms 执行一次加法操作。如下所示

```
a:26
b:67
a+b:93
java.io.IOException: this is a exception!
    at com.netease.qa.btrace.Test.add(Demo1.java:53)
    at com.netease.qa.btrace.Demo1.add(Demo1.java:23)
    at com.netease.qa.btrace.Demo1.main(Demo1.java:36)
a:77
b:61
a+b:0
java.io.IOException: this is a exception!
    at com.netease.qa.btrace.Test.add(Demo1.java:53)
    at com.netease.qa.btrace.Demo1.add(Demo1.java:23)
    at com.netease.qa.btrace.Demo1.main(Demo1.java:36)
```

使用 jps 获取可进程 pid。自此，被监控对象已经准备到位。

1.4.2 编写 BTrace 脚本

下面是一个最简单的 BTrace 脚本：

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BtraceUtils.*;
@BTrace
public class Simple {
```

```
    @OnMethod(
        clazz="com.netease.qa.btrace.Demo1",
        method="add",
        location=@Location(value=Kind.ENTRY)
    )
    public static void onMethodCall0(@ProbeClassName String pcn, @ProbeMethodName String pmn) {
            println("================ Method add is call ==================");
            println(Strings.strcat("class name: ", pcn));
            println(Strings.strcat("method name: ", pmn));
    }
}
```

可以看到，BTrace 脚本看上去就是一个 java class，语法和标准 java 代码比较类似。除了标准 java class 常见的 import 声明，class 声明，method 定义外，还有一些特殊的标签（@标注）。其中，

- @BTrace 是类注解，表示 Simple 这个 java class 是一个 BTrace 脚本
- @OnMethod 是方法注解，表示 onMethodCall0()这个 java method 是一个 BTrace 方法，也叫 **action method。**
- @ProbeClassName、 @ProbeMethodName：分别是加在参数 pcn、pmn 之前的参数注解，表示这些参数有特殊的意义。
- @Location、Kind.Entry： 这些是使用在@OnMethod 方法注解内部的其他参数
- Strings.strcat()方法、println()方法， 这个是在 com.sun.btrace.BTraceUtils 包中定义的静态方法。BTrace 的 com.sun.btrace.BTraceUtils.*包下面定义了很多常见的工具方法，如字符串操作等。

上述的 btrace 脚本是怎样工作的？

红色的代码，指定了 btrace 监控的位置、时机。

其中 clazz=... method=... 这两个属性，指定了这个 BTrace 方法的注入位置，这个注入的位置叫 **probe point**， 具体来讲，add()方法叫 **probed method**， com.netease.qa.btrace.Demo1 类叫 **probed class**。 也就是说， Btrace 脚本中的方法 onMethodCall0()会注入在目标 JVM 的 com.netease.qa.btrace.Demo1.add()调用处。注入操作是通过修改 add()方法的字节码实现的，详见附录参考资料。

一旦注入成功，action method 会在被监控应用执行到 probe point 的时候被触发执行。

但是，这里具体是 add()方法调用前，还是调用完成时呢？ 这个由@Location 注解指定，这里的 value=Kind.ENTRY，指定了 onMethodCall0 方法会在 com.netease.qa.btrace.Demo1.add()调用结束后执行。

蓝色的代码，指定了被注入到 JVM 中的代码（action method），具体执行了哪些行为(action)。示例中的行为非常简单，就是打印一些基本的信息。

其中@com.sun.btrace.annotations.ProbeClassName 参数注解，能获取 probed 类的全限定名，@com.sun.btrace.annotations.ProbeMethodName 参数注解，能获取 probed 方法的方法名。

下面启动 btrace 脚本，实际的执行结果如下：

```
hzzhangweijie@app-50:~/btrace$ ./bin/btrace 15082 Simple.java
================ Method add is call ===================
class name: com.netease.qa.btrace.Demo1
method name: add
================ Method add is call ===================
class name: com.netease.qa.btrace.Demo1
method name: add
================ Method add is call ===================
class name: com.netease.qa.btrace.Demo1
method name: add
```

通过上面的例子可以看到，每当目标进程执行到 com.netease.qa.btrace.Demo1.add()方法时，就会 onMethodCall0()方法，打印出一些信息。

二、简明语法教程和示例

2.1 BTrace 中的语法限制

通过上面的例子，可以看到 BTrace 脚本看上去是一个标准 java 类，加上一些特殊的注解。实际上，在这个 java 类中，存在一些语法的限制。

- Action method 必须是 public static void。
- 出于安全限制，在一个 BTrace 脚本中不允许以下操作：

- 创建新对象、数组
- throw 异常、catch 异常
- 调用对象的方法、调用 static 方法，只允许调用 com.sun.btrace.BTraceUtils.*包下面定义好的 static 方法、或者同一个 BTrace 脚本里面定义的方法。
- 拥有成员变量、方法。只能拥有 public static void 方法 (1.2 版本以前)
- 给目标 jvm 的成员变量、static 变量赋值。只能给 btrace 类自己的 static 变量赋值
- 使用循环
- 继承类和实现接口
- 使用 assert 断言
- 使用 class literal

2.2 开启 unsafe 模式

上述的限制都是出于安全方面的考虑，最大程度上避免代码注入操作影响被监控的应用的正常运行。有时候，我们需要调用 jdk 里面的各种方法，或者调用目标应用内部的方法，以方便在监控脚本里实现更复杂的逻辑，由于这些限制的存在，就会报错，如调用其他方法会报：

method calls are not allowed - only calls to BTraceUtils are allowed

这时候，可以考虑使用 unsafe 模式。

开启 unsafe 模式需要执行两步操作：

1. $BTRACE_HOME/bin/btrace 启动脚本中，修改 JVM 启动参数，设置

   -Dcom.sun.btrace.unsafe=true

2. BTrace 脚本中，修改类注解 @BTrace，添加(unsafe=true)，如下所示：

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BtraceUtils.*;
@Btrace(unsafe=true)
public class Simple {
    ...
```

开启 unsafe 模式后，上述安全限值全部失效。

Tips

针对一个被监控的 jvm，需要第一次执行 btrace 的时候，即第一次 attach 的时候就设置好 unsafe 模式，否则修改无效，除非重启目标 JVM，重新注入。

2.3 BTrace 注解概览

BTrace 支持的注解，按照作用级别和位置，共分为 4 类，其中

BTrace 支持的类注解：

```
@DTrace
@DTraceRef
@BTrace
```

BTrace 支持的方法注解：

```
OnMethod
OnTimer
OnError
OnExit
OnEvent
OnLowMemory
OnProbe
```

BTrace 支持的参数注解：

```
@Self
@Return
@ProbeClassName
@ProbeMethodName
@Duration
@TargetInstance
@TargetMethodOrField
```

BTrace 支持的成员变量注解

```
@Export
@Property
@TLS
```

此外，BTrace 脚本中还包含一些其他参数，全都是配合方法注解@OnMethod 使用：

```
Kind.ENTRY, Kind.RETURN
Kind.THROW
Kind.ARRAY_SET, Kind.ARRAY_GET
Kind.CATCH
```

```
Kind.FIELD_SET

Kind.LINE

Kind.NEW

Kind.ERROR
```

2.4 BTrace 类注解

类注解有三个：

```
@BTrace

@DTrace

@DTraceRef
```

其中@BTrace 之前已经介绍过，是标注这个 java 类是一个 BTrace 脚本，而且这个注解是必须的。

@DTrace、@DTraceRef 注解和 DTrace 相关， DTrace 用在 Solaris 系统上（DTrace is a dynamic, safe tracing system for Solaris programs - both kernel and user land programs.），就不用介绍了。

2.5 BTrace 方法注解

2.5.1 OnMethod

OnMethod 是最常用的方法注解。当为一个 action method 添加@OnMethod 注解后，当目标代码执行到注解中定义的位置时， action method 也会被触发执行。

必选属性：

clazz： class 的全限定名，可以使用正则表达式

可选属性：

method： method 的名称，可以使用正则表达式

localtion: 详细注入位置说明

```
    Kind.ENTRY, Kind.RETURN- the probed method arguments

    Kind.THROW - the thrown exception

    Kind.ARRAY_SET, Kind.ARRAY_GET - the array index

    Kind.CATCH - the caught exception

    Kind.FIELD_SET - the field value

    Kind.LINE - the line number

    Kind.NEW - the class name
```

```
        Kind.ERROR - the thrown exception
    type
    follow
```

示例 1：clazz 和 method 使用正则表达式

clazz 和 method 都是 key-value 对的格式，其中 value 部分可以使用正则表达式的写法，符合正则匹配的 class、mehod 中的方法，一旦被执行，都能触发 action method。 可以参考 BTrace sample 目录下的实例程序 MultiClass。这个示例中，被监控应用中，只要有 java.io.XXXInputXXX 类的 readXXX() 方法被调用，就会触发 action method，打印出 probed class 的全限定名。

```
package com.sun.btrace.samples;
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
@BTrace public class MultiClass {
    @OnMethod(
        clazz="/java\.io\..Input./",
        method="/read.*/"
    )
    public static void onread(@ProbeClassName String pcn) {
        println(Strings.strcat("read on ", pcn));
    }
}
```

location 参数中，也可以会出现 clazz 和 method，这里同样也可以使用正则。

可以参考 BTrace sample 目录下的实例程序 AllCalls2，摘要如下：

```
location=@Location(value=Kind.CALL, clazz="/./", method="/./"))
```

Tips：

正则表达式前后需要用正斜杠标志。

示例 2:clazz 使用+号

如果一个 class 继承了某个类，或者实现某个接口，即使不知道 class 名字，也能通过+号进行匹配。可以参考 BTrace sample 目录下的示例程序：Classload，任何继承 java.lang.ClassLoader 的类中的 defineClass 方法被调用后，都会触发 action method。

```
package com.sun.btrace.samples;
```

```
import static com.sun.btrace.BTraceUtils.*;
import com.sun.btrace.annotations.*;
@BTrace public class Classload {
  @OnMethod(
    clazz="+java.lang.ClassLoader",
    method="defineClass",
    location=@Location(Kind.RETURN)
  )
  public static void defineclass(@Return Class cl) {
     println(Strings.strcat("loaded ", Reflective.name(cl)));
     Threads.jstack();
     println("==========================");
  }
}
```

Tips：

com.sun.btrace.BTraceUtils.*包下的工具方法 Threads.jstack()，可以打印出被监控程序运行到 probe point 时的 JVM 堆栈。

示例 3：类和方法本身使用@标签—— 方便应用于 web 类产品

如果应用采用如 Spring MVC 架构实现，那么很多类和方法会带上各种 Spring 的标签，如@Resource、@Controller 等。clazz 和 method 可以直接匹配这种类和方法，而不需要知道具体的类名和方法名。参考 BTrace sample 目录下的示例程序 WebServiceTracker。每当有包含@javax.jws.WebService 类标注、@javax.jws.WebMethod 方法标注的方法被执行，就会触发 action method。

```
package com.sun.btrace.samples;
import static com.sun.btrace.BTraceUtils.*;
import com.sun.btrace.annotations.*;
@BTrace public class WebServiceTracker {
  @OnMethod(
    clazz="@javax.jws.WebService",
    method="@javax.jws.WebMethod"
  )
```

```
  public static void onWebserviceEntry(@ProbeClassName String pcn, @ProbeMethodName
 String pmn) {
      print("entering webservice ");
      println(Strings.strcat(Strings.strcat(pcn, "."), pmn));
  }
}
```

2.5.2 OnTimer

OnTimer 是一个很有用的注解，在 Action method 上添加了 OnTimer 注解后，action method 会被定时执行（单位毫秒），常用于定时打印一些统计信息。

2.5.3 OnExit

OnExit 也是一个常用的注解，在 Action method 上添加了 OnExit 注解后，一旦 BTrace 脚本中显式调用了 exit()方法时， action method 就会执行。一般会在 action method 内部实现一些结果统计、资源清理之类的工作。

示例 4 OnTimer & OnExit:

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
@BTrace
public class OnTimerAndExit {
    private static volatile int i;

    @OnTimer(1000)
    public static void onTimer(){
        println("=====on second====");
        i++;
        if(i == 5){
            Sys.exit(0);
        }
```

```
    }

    @OnExit
    public static void onExit(int code){
        println("=====Btrace exit!====");
        println(Strings.strcat("exit code: ", str(code)));
    }
}
```

示例程序中，onTimer 方法每隔 1000ms 执行一次，每次将类静态变量 i 加 1，之后显式调用 exit()方法。 此时 onExit 方法被执行，并且能拿到程序的退出码。程序输出如下

```
hzzhangweijie@app-50:~/btrace$ ./bin/btrace 29571 OnTimerAndExit.java
=====one second====
=====one second====
=====one second====
=====one second====
=====one second====
=====Btrace exit!====
exit code: 0
```

2.5.4 OnEvent

OnEvent 注解支持用户自定义一些事件标记，BTrace 脚本运行中，如果用户键入 ctrl+c，就可以进入交互模式，输入相应的事件名称，可触发事件执行。

示例 5 OnEvent

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
@BTrace
public class OnEventTest {
    @OnEvent
    public static void onUnnameEvent(){
```

```
        println("=====receive a unname event====");
    }

    @OnEvent("abc")
    public static void onEvent(){
        println("=====receive event abc!====");
    }
}
```

示例中定义了两个事件，其中一个未命名，另一个命名为“abc”。启动 btrace 脚本后，键入 ctrl+c，进入交互模式，用户可以选择 2，向 btrace 发送一个未命名的事件，btrace 脚本捕获到事件后，触发 onUnnameEvent， 类似用户可以选择 3，发送带名字的事件，以触发相应的 action method。如果输入的事件名称在脚本中未定义，则无效。程序输出如下：

```
hzzhangweijie@app-50:~/btrace$ ./bin/btrace 29571 OnEventTest.java
^CPlease enter your option:
    1. exit
    2. send an event
    3. send a named event
2
=====receive a unname event====

^CPlease enter your option:
    1. exit
    2. send an event
    3. send a named event
3
Please enter the event name: abc
=====receive event abc!====
```

Btrace 官方示例上的使用场景：将一些统计信息打印在包含 OnEvent 注解的 action 方法中，通过发起事件，随时查阅这些统计信息，具体参考 sample 目录下的示例程序 HistoOnEvent。

2.5.5 OnError

当 BTrace 脚本出错时，触发该 action method。不常用

2.5.6 OnLowMemory

通过 OnLowMemory 注解，用户可以在 BTrace 脚本中实现简单的内存监控。参考 sample 目录下的示例程序 HistoOnEvent。pool 参数指定了目标内存区域是老年代，阈值是 6000000 字节。如果老年代的使用量超过该阈值，就会触发 onLowMem 方法。

示例 6 OnLowMemory

```
package com.sun.btrace.samples;
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
import java.lang.management.MemoryUsage;

@BTrace
public class MemAlerter {
    @OnLowMemory(
       pool = "Tenured Gen",
       threshold=6000000
    )
    public static void onLowMem(MemoryUsage mu) {
        println(mu);
    }
}
```

2.6 BTrace 参数注解

根据 2.5 节的描述，可以看出方法注解主要是定义了应该何时、何地执行 action method。 而本节介绍的参数注解，主要定义了通过执行 action method，用户能获得什么样的信息。

2.6.1 Self

获取 probed 类在调用 probed 方法时，调用者对象的实例，相当于 java 中的 this。示例程序见 2.6.3。

2.6.2 ProbeClassName

获取 probed 类的全限定名。示例程序见 2.6.3。

2.6.3 ProbeMethodName

获取 probed 方法的方法名

示例 7 ProbeClassName & ProbeMethodName

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
@BTrace
public class Simple {
    @OnMethod(
        clazz="com.netease.qa.btrace.Demo1",
        method="add",
        location=@Location(value=Kind.ENTRY)
    )
    public static void onMethodCall0(@Self Object self, @ProbeClassName String pcn, @ProbeMethodName String pmn) {
            println("================ Method add is call ==================");
            println(Strings.strcat("self: ", str(self)));
            println(Strings.strcat("class name: ", pcn));
            println(Strings.strcat("method name: ", pmn));
    }
}
```

示例程序共添加了@Self、 @ProbeClassName、@ProbeMethodName 三个参数注解，其中@ProbeClassName、@ProbeMethodName 标注的参数，分别是 probed 方法和 probed 类的名字，@Self 则对应 probed

类的具体实例，通过这三个参数，能很方便的监控这个哪些类（哪个实例）的哪些方法被调用了。

程序输出如下：

```
hzzhangweijie@app-50:~/btrace$ ./bin/btrace 29571 Simple.java
================ Method add is call ===================
self: com.netease.qa.btrace.Demo1@20c1f10e
class name: com.netease.qa.btrace.Demo1
method name: add
================ Method add is call ===================
self: com.netease.qa.btrace.Demo1@20c1f10e
class name: com.netease.qa.btrace.Demo1
method name: add
```

2.6.4 TargetInstance

获取 probed 方法调用的子方法的对象实例。必须和 Kind.CALL 同时使用。示例程序见 2.6.5。

2.6.5 TargetMethodOrField

获取 probed 方法调用的子方法的方法名。必须和 Kind.CALL 同时使用。@TargetInstance 和@TargetMethodOrField 注解可以获取方法调用的关系信息。

示例 8 TargetInstance & TargetMethodOrField

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
@BTrace
public class Simple {
    @OnMethod(
        clazz="/com.netease.qa.btrace./",
        method="/./",
        location=@Location(value=Kind.CALL, clazz="/./", method="/./")
    )
```

```
    public static void onMethodCall1(@Self Object self, @ProbeClassName String pcn, @ProbeMethodName String pmn, @TargetInstance Object instance, @TargetMethodOrField String method) {
            println("================ Method add is call ==================");
            println(Strings.strcat("self: ", str(self)));
            println(Strings.strcat("class name: ", pcn));
            println(Strings.strcat("method name: ", pmn));
            println(Strings.strcat("Target instance: ", str(instance)));
            println(Strings.strcat("Target Method: ", method));
    }
}
```

示例程序中，OnMethod 注解中没有指定具体的类和方法名，通过 TargetInstance、TargetMethodOrField 注解，可以在 Demo1 实例的 add()方法调用 Test 实例的 add()方法时，触发 action method，打印调用者的实例名、方法名，com/netease/qa/btrace/Demo1、add，已经被调用者的实例名、方法名，com.netease.qa.btrace.Test@3a6ac461、add。

程序输出如下：

```
hzzhangweijie@app-50:~/btrace$ ./bin/btrace 29571 Simple.java
================ Method add is call ==================
self: com.netease.qa.btrace.Demo1@20c1f10e
class name: com/netease/qa/btrace/Demo1
method name: add
Target instance: com.netease.qa.btrace.Test@3a6ac461
Target Method: add
```

2.6.6 Return

获取 probed 方法的返回值，必须和 Kind.RETURN 或者 Kind.ERROR 配合使用。实例程序见 2.6.7

2.6.7 Duration

获取 probed 方法的调用时间(单位为纳秒)，必须和 Kind.RETURN 或者 Kind.ERROR 配合使用

示例 9 Return & Duration

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
@BTrace
public class Simple {
    @OnMethod(
        clazz="/com.netease.qa.btrace.*/",
        method="add",
        location=@Location(value=Kind.RETURN)
    )
    public static void onMethodCall3(@Self Object self, @ProbeClassName String pcn, @
ProbeMethodName String pmn, int a, int b, @Duration long time, @Return int c) {
            println("================ Method add is call ==================");
            println(Strings.strcat("self: ", str(self)));
            println(Strings.strcat("class name: ", pcn));
            println(Strings.strcat("method name: ", pmn));
            println(Strings.strcat("input a=: ", str(a)));
            println(Strings.strcat(input b=: ", str(b)));
            println(Strings.strcat("return c=: ", str(c)));
            println(Strings.strcat("time used: ", str(time)));
    }
}
```

示例程序中，可以通过 `int a, int b` 这样的形式，获取 probed 方法的具体传入参数， 通过`@Return` 注解过的参数 c，拿到 probed 方法的返回值。此处要注意参数的类型。同时，通过`@Duration` 注解过的参数 time，拿到 probed 方法的执行耗时。

程序输出如下：

```
hzzhangweijie@app-50:~/btrace$ ./bin/btrace 29571 Simple.java
================ Method add is call ==================
self: com.netease.qa.btrace.Test@5d5bdc50
class name: com.netease.qa.btrace.Test
method name: add
input a=: 7
```

```
input b=: 73

return c=: 80

time used: 569

================ Method add is call ==================

self: com.netease.qa.btrace.Demo1@20c1f10e

class name: com.netease.qa.btrace.Demo1

method name: add

input a=: 7

input b=: 73

return c=: 80

time used: 500581
```

通过输出可以看到， com.netease.qa.btrace.Demo1、 com.netease.qa.btrace.Test 两个实例的 add 方法调用成功后，都能触发 action method，因此 main 函数一次循环，会产生两次输出，且两次输出中的 probed class 名字和实例名不同。同时可以看到，由于两者的调用关系，com.netease.qa.btrace.Test.add()方法先返回，耗时很短只有 0.569ms，而 com.netease.qa.btrace.Demo1.add()方法后后返回，耗时中包含了子函数的调用过程，明显更长。

2.6.8 AnyType

2.6.7 的例子可以看到，在 action method 的参数列表中，可以通过 int a, int b 这样的形式，获取 probed 方法的具体传入参数。但是有些时候，我们无法知道哪些 probed 方法会被匹配，因此更加无法知道这些方法的传入参数有几个，分别是哪些类型。 这时候，可以使用 AnyType 注解。

AnyType 注解没有@标记，但是仍可以认为是一种参数注解。通过它可以一次性获取 probed 方法的整个参数列表，而事先不用知道参数的数量和类型。

示例 10 AnyType

```
import com.sun.btrace.annotations.*;

import static com.sun.btrace.BTraceUtils.*;

import com.sun.btrace.AnyType;

@BTrace

public class Simple {

    @OnMethod(

        clazz="/com.netease./",

        method="/./",
```

```
        location=@Location(value=Kind.RETURN)
    )
    public static void onMethodCall4(AnyType[] args, @Self Object self, @ProbeClassName String pcn, @ProbeMethodName String pmn) {
            println("================ Method add is call ==================");
            println(Strings.strcat("self: ", str(self)));
            println(Strings.strcat("class name: ", pcn));
            println(Strings.strcat("method name: ", pmn));
            printArray(args);
    }
}
```

示例程序输出如下：

```
================ Method add is call ==================
self: com.netease.qa.btrace.Test@57a7ddcf
class name: com.netease.qa.btrace.Test
method name:
[]
================ Method add is call ==================
self: com.netease.qa.btrace.Test@57a7ddcf
class name: com.netease.qa.btrace.Test
method name: add
[42, 27, ]
================ Method add is call ==================
self: com.netease.qa.btrace.Demo1@20c1f10e
class name: com.netease.qa.btrace.Demo1
method name: add
[42, 27, ]
```

main 方法每次循环，会有触发三次 action method，根据输出结果可以很清楚的看到 probed 方法之间执行的先后关系：com.netease.qa.btrace.Demo1.add 调用 com.netease.qa.btrace.Test.add，在这之前，还要先调用 com.netease.qa.btrace.Test 类的构造函数。在输出结果中，还能看到这些 probed 方法的入参值。

Tips

1、AnyType[] args 需要写在 action method 方法参数列表的 最前面。

2、需要 import com.sun.btrace.AnyType;

3、BTrace 定义了 printArray 方法，可以很方便的打印数组中的每个元素值。

2.7 BTrace 成员变量注解

2.7.1 Export

2.7.2 Property

2.7.3 TLS

标记脚本中的某个成员变量为 thread local。使用场景不多见，某些场合需要保证是同一个线程触发了多个 action method。

示例 11 TLS

```
package com.sun.btrace.samples;
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
@BTrace public class OnThrow {
    @TLS static Throwable currentException;
    @OnMethod(
        clazz="java.lang.Throwable",
        method=""
    )
    public static void onthrow(@Self Throwable self) {
        currentException = self;
    }

    @OnMethod(
        clazz="java.lang.Throwable",
        method="<init>",
```

```
        location=@Location(Kind.RETURN)
    )
    public static void onthrowreturn() {
        if (currentException != null) {
            Threads.jstack(currentException);
            println("====================");
            currentException = null;
        }
    }
}
```

在上述示例代码中，每当目标 JVM 中有 new Exception 操作，都会触发 onthrow()这个 action method，为 thread local 类型变量 currentException 赋值。 同时，当本次 new Exception 完成后，会触发 onthrowreturn()这个 action method，获取并打印这个线程自己的 currentException。

由于 JVM 内部普遍存在多线程操作，可能同时有多个线程在进行 Exception 对象初始化，如果 currentException 变量不是 thread local 类型，则可能出现混乱。

示例程序输出如下：

```
hzzhangweijie@app-50:~/btrace$ ./bin/btrace 29571 samples/OnThrow.java
java.io.IOException: this is a exception!
    com.netease.qa.btrace.Test.add(Demo1.java:53)
    com.netease.qa.btrace.Demo1.add(Demo1.java:23)
    com.netease.qa.btrace.Demo1.main(Demo1.java:36)
====================
java.io.IOException: this is a exception!
    com.netease.qa.btrace.Test.add(Demo1.java:53)
    com.netease.qa.btrace.Demo1.add(Demo1.java:23)
    com.netease.qa.btrace.Demo1.main(Demo1.java:36)
====================
```

2.8 其他参数

介绍@OnMehod 方法注解时，location=@Location()括号内出现了很多种 Kind.XXX 类型的参数。这批参数都是配合@OnMehod、location 使用的，可以视作是对 Location 的补充说明。下面介绍几种：

2.8.1 kind.THROW， kind.ERROR

probed 方法 throw 出了 exception 时，触发 action method。示例程序见 2.8.3.

2.8.2 Kind.ENTRY， Kind.RETURN

probed 方法刚开始调用，或者调用完成时，触发 action method。

上文提到过，使用@Return 参数注解时，必须配合 Kind.RETURN 使用。

2.8.3 Kind.CATCH

probed 方法内，程序逻辑进入到 try-catch 的 catch 块时，触发 action method。

示例 12 kind.THROW，kind.CATCH

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
import com.sun.btrace.AnyType;

@BTrace
public class Simple {
    @OnMethod(
        clazz="/com.netease./",
        method="/./",
        location=@Location(value=Kind.THROW)
    )
    public static void onMethodCall5(@ProbeClassName String pcn, @ProbeMethodName St
ring pmn) {
            println("================ throw an exception ==================");
            println(Strings.strcat("class name: ", pcn));
            println(Strings.strcat("method name: ", pmn));
    }
```

```
    @OnMethod(
        clazz="/com.netease.*/",
        method="/.*/",
        location=@Location(value=Kind.CATCH)
    )
    public static void onMethodCall6(@ProbeClassName String pcn, @ProbeMethodName St
ring pmn) {
        println("================ catch an exception ==================");
        println(Strings.strcat("class name: ", pcn));
        println(Strings.strcat("method name: ", pmn));
    }
}
```

示例程序输出如下：

```
================ throw an exception ==================
class name: com.netease.qa.btrace.Test
method name: add
================ catch an exception ==================
class name: com/netease/qa/btrace/Demo1
method name: add
```

每当 com.netease.qa.btrace.Test.add()抛出异常时，onMethodCall5 方法触发执行，

同时 com/netease/qa/btrace/Demo1.add()会进行异常捕获，onMethodCall6 方法触发执行。

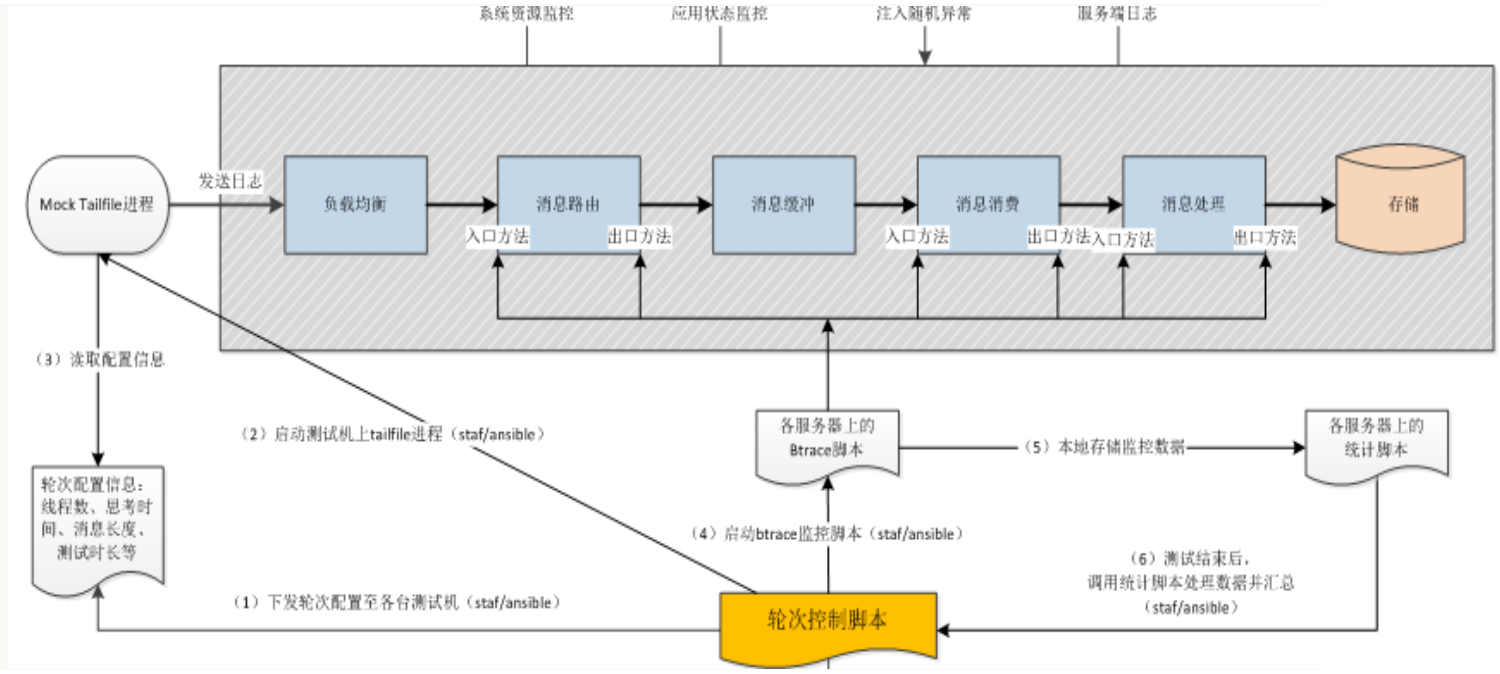
2.8.4 Kind.LINE

probed 方法内，程序逻辑执行到具体某行时，触发 action method。

三、应用实例

3.1 产品背景介绍

Datastream产品是一个消息传输服务，消息从源头到存储介质，全流程包含多个独立的java类型的节点，节点使用内部消息协议进行通信，传输数据。并且，节点之间通信全部为异步方式。如下图所示：



3.2 BTrace 应用场合分析

由于节点之间全部使用异步通信方式，在性能测试过程中，无法通过同步压测的方式进行测试和指标统计。因此，只能通过其他方式来收集性能指标。需求如下：

1、获取每个节点接收、发送消息的吞吐量。

2、获取一条消息从产生，到传输至某个节点时，消息的延迟时间。

这是一个极为适合 BTrace 发挥作用的场合。经过分析，可以通过@OnMethod 注解，获取关键方法的调优频率，由此计算接收消息和发送消息的 TPS。

同样的，配合@OnMethod 注解，在 action method 中调用产品自身解析工具，解析消息体，可以计算延迟时间。

由于这里需要调用被监控应用自身的方法，所以需要开启 Unsafe 模式。

Tips

哨兵系统的 java method 监控也是用字节码注入的方式进行，因此和 BTrace 存在冲突，如果哨兵系统开启了某些包、类或者方法的监控，BTrace 脚本就无法顺利注入了（这是因为哨兵系统利用 java instrument 修改字节码，方法名和类名都会发生变化）。

3.3 BTrace 脚本实例

计算某个节点消息接收 TPS 和消息平均延迟时间的脚本如下：

示例 12 DataStream 应用实例

```
import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;
import java.util.concurrent.atomic.AtomicLong;
import com.cloudera.flume.core.Event;
import com.netease.datastream.common.entity.LogMessage;
import com.netease.datastream.common.exception.DataStreamException;
import com.netease.datastream.sinkcommon.tag.TagConfigServiceImp;

@BTrace(unsafe=true)
public class AgentTps {
    private static Long count = 0l;// Atomic.newAtomicLong(0);
    private static Long time = 0l;//Atomic.newAtomicLong(0);

    @OnMethod(
        clazz="com.netease.datastream.flumeExtra.datastreamSink.collector.TagAllocat
ionSink",
        method="append"
    )
    public static void onCount0(@ProbeClassName String pcn, @ProbeMethodName String
pmn, Event e) throws DataStreamException {
        println(str(TagConfigServiceImp.getInstance().getTagName(e)));
        count = count +1;
        time = time + (Time.millis()/1000-parseLong(new String(e.getAttrs().get("_ds_
time_stamp")))/1000);
    }

    @OnTimer(1000)
    public static void onTimer() {
        println(str(Time.millis()/1000) + " " + str(count)+" "+ str(time));
```

```
        count =0l;
        time =0l;
    }
}
```

在@OnMethod 注解的 action method 中，统计了 probed method 的调用次数。同时，脚本开启了 unsafe 模式后调用了应用自身的大量方法，解析消息体计算并累加延迟。

在@OnTimer 注解的 action method 中，每个 1 秒打印一次方法调用频率（即为 TPS），已经统计间隔内的平均延迟时间。同时对 static 变量进行归零。

四、常见错误

4.1 常见错误

Port 2020 unavailable

btrace 报错：Port 2020 unavailable

btrace 会在目标 java 进程中开启一个远程 socketserver，默认端口是 2020，如果这台服务器有多个 JVM 实例需要监控，在第一个 JVM 上添加 Btrace 监控后，2020 端口会被占，对其他进程进行监控会报错。

正确的做法是：./btrace -p【另一个端口】[PID] Demo.java。

There is already a BTrace server active on port 2020!

btrace 报错： There is already a BTrace server active on port 2020!

一旦 btrace 脚本 attach 到一个 java 进程之后，即使关闭 btrace 程序，attach 仍然有效，即目标 java 进程上的端口 2020 仍开启着。后续对这个 java 进程进行监控，必须使用-p 指定为同一个端口才可以。

Connection refused

btrace 报错：Connection refused

见 1.3 小节 Tips。

method calls are not allowed - only calls to BTraceUtils are allowed

btrace 报错：method calls are not allowed - only calls to BTraceUtils are allowed

见 2.2 小节

4.2 Q&A

能否监控第三方 jar 包中的代码？

可以。启动 btrace 的时候，在-cp 参数中添加这些 jar 的路径。

BTrace 对被监控 java 应用有多大影响？
对性能有一定影响，取决于 action method 的执行次数、执行内容，执行越频繁，执行逻辑越耗时，影响越大。
此外，一旦对一个 java 应用进行过 BTrace 监控，即使停止监控，字节码注入效果仍然存在。因此，对线上环境进行操作时，不要开启 unsafe 模式。

附录

参考资料：

BTrace 官方手册：https://kenai.com/projects/btrace/pages/UserGuide

BTrace java doc：https://btrace.kenai.com/javadoc/1.2/index.html

BTrace 实现机制介绍：http://www.iteye.com/topic/1005918