# 1. java性能调优及问题追踪--Btrace的使用

# [BTrace简介与使用说明](https://www.cnblogs.com/wei-zw/p/9502274.html)

       大家在开发过程中总会预告各种个样的BUG,不是什么时候都可以去debug,也不是什么时候都可以去在代码中增加需要的日志,这个时候该如何解决呢?这个时候就是BTrace的大展身手的时候时候了,下面通过对于BTrace的介绍,同时会有一些示例代码希望可以给大家带来一定的了解

## 简介

    Btrace (Byte Trace)是sun推出的一款java 动态、安全追踪工具，可以不停机的情况下监控线上情况，并且做到最少的侵入，占用最少的系统资源。BTrace应用较为广泛的原因应该是其安全性和无侵入性，其中热交互技术，使得我们无需启动Agent的情况下动态跟踪分析，其安全性不会导致对目标Java进程的任何破坏性影响，使得BTrace成为我们线上产品问题定位的利器。无侵入性无需我们对原有代码做任何修改，降低上线风险和测试成本，并且无需重启启动目标Java进程进行Agent加载即可动态分析和跟踪目标程序，可以说BTrace可以满足大部分的应用场景。

## 安装

### 下载BTrace

     BTrace已经迁移到GitHub, 最新到版本是[v1.3.11](https://github.com/btraceio/btrace/releases/tag/v1.3.11)  下载 https://github.com/btraceio/btrace/releases/download/v1.3.11/btrace-bin-1.3.11.zip 后解压到指定目录

### 配置BTRACE\_HOME

|  |  |
| --- | --- |
| 1 | vi .bash\_profile |
| 1  2  3  4  5  6 | JAVA\_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0\_45.jdk/Contents/Home  BTRACE\_HOME=/Users/david/Downloads/btrace-bin-1.3.11  PATH=$PATH:$BTRACE\_HOME/bin:$JAVA\_HOME/bin:$HOME/bin  export JAVA\_HOME  export BTRACE\_HOME  export PATH |

　配置完成后执行  source .bash\_profile

  在终端中输入btrace 可以看到如下内容

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | $ btrace  Usage: btrace <options> <pid> <btrace source or .class file> <btrace arguments>  where possible options include:    --version             Show the version    -v                    Run in verbose mode    -o <file>             The path to store the probe output (will disable showing the output in console)  -u                    Run in trusted mode    -d <path>             Dump the instrumented classes to the specified path    -pd <path>            The search path for the probe XML descriptors    -classpath <path>     Specify where to find user class files and annotation processors    -cp <path>            Specify where to find user class files and annotation processors    -I <path>             Specify where to find include files    -p <port>             Specify port to which the btrace agent listens for clients    -statsd <host[:port]> Specify the statsd server, if any |

## BTrace使用

1. jps命令查出需要监控的jvm pid
2. 编写BTrace跟踪程序
3. 执行：btrace <pid> BTrace跟踪程序

## 注意事项

     Btrace脚本就是一个普通的用@Btrace注解的Java类,其中包含一个或多个public static void修饰的方法,注意拦截方法必须是用public static  void 进行修饰的,如果不是静态方法则会抛出 instance methods are not allowed 这样的异常信息 如果不是public 则会提示btrace methods should be public ;如果有返回则提示信息为:btrace probe methods must return void

       为了保证对目标程序不造成影响,Btrace脚本对其可以执行的动作做了很多限制，如下：

1. 不能创建对象
2. 不能抛出或者捕获异常
3. 不能用synchronized关键字
4. 不能对目标程序中的instace或者static变量
5. 不能调用目标程序的instance或者static方法
6. 脚本的field、method都必须是static的
7. 脚本不能包括outer,inner,nested class
8. 脚本中不能有循环,不能继承任何类,任何接口与assert语句

## BTrace注解

   对于BTrace的学习离不开对BTrace注解的理解,BTrace注解可以分为类注解 @BTrace 方法注解如@OnMethod 参数注解如:@ProbeClassName 为了后面更好理解demo我们首先从参数注解开始介绍

### @ProbeClassName

       用于标记处理方法的参数,仅用户@OnMethod, 该参数的值就是被跟踪的类名称

### @ProbeMethodName

       用于表姐处理方法的参数,仅用户 @OnMethod,该参数值是被跟踪方法名称

### @Self

     当前截取方法的封闭实例参数

### @Return

    当前截取方法的的返回值, 只对 location=@Location(Kind.RETURN) 生效

### @Duration

    当前截取方法的执行时间

### @TargetInstance

 当前截取方法内部调用的实例

### @TargetMethodOrField

   当前截取方法内部被调用的方法名

方法注解重点介绍OnMethod这个也是我们在排查问题时重点使用的.

### @OnMethod

#### 作用

       用于指定跟踪方法到目标类,目标方法和目标位置

#### 格式

     @OnMethod(clazz=<cname\_spec>[, method=<mname\_spec>]? [, type=<signature>]? [, location=<location\_spec>]?)

#### 参数说明

##### clazz  用于限定目标类

        cname\_spec = <class\_name> | +<class\_name> | /regex/  class\_name 是完全限定类名 +class\_name 完全限定类名称前加上“+”表示这个类的所有子类或实现,/regex/就是用户识别类名称的标准正则表达式

      method 用户限定目标方法, mname\_spec表示简单的方法名称,不包含签名和返回类型;

      type:用户限定目标方法的签名和返回类型 <return\_type> ((arg\_type(,arg\_type)\*)?  return\_type就是方法的返回类型,如void, java.lang.String; arg\_type就是参数类型

     location 用于限定目标方法的位置, 通过@Location注解进行指定

       @Location 属性有:

1. value 默认值为Kind.ENTRY 即参数的入口位置
2. where 限定探测位置 默认值为 Where.BEFORE 也可以设置为Where.AFTER
3. clazz
4. method
5. field
6. type
7. line

     其中 @Kind注解的值有

* Kind.ENTRY-被trace方法参数
* Kind.RETURN-被trace方法返回值
* Kind.THROW -抛异常
* Kind.ARRAY\_SET, Kind.ARRAY\_GET -数组索引
* Kind.CATCH -捕获异常
* Kind.FIELD\_SET -属性值
* Kind.LINE -行号
* Kind.NEW -类名
* Kind.ERROR -抛异常

    获取截取方法 类,方法,实例,返回值以及耗时信息,限定的类为Map接口的实现类,方法为put 位置为return 之前

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @BTrace  public class TracingScript {      @OnMethod(clazz="+java.util.Map",method="put",location=@Location(Kind.RETURN))    public static void testB(@ProbeClassName String pcm,@ProbeMethodName String pmn,@Self Object self,@Duration long duration,@Return  Object result){         println(strcat(strcat(strcat(strcat(pcm,"#"),pmn)," called in"),str(self)));          println(strcat(strcat("result is ",str(result)),strcat(" duration is ",str(duration))));              }    } |

　　输出的结果是:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | java.util.HashMap#put called in{name=javax.management.ObjectName$Property@338bf6b2, type=javax.management.ObjectName$Property@6f24f3ca}  result is null duration is 2488  java.util.HashMap#put called in{javax.management.ObjectName=java.lang.Object@74685cec}  result is null duration is 5692  java.util.HashMap#put called in{type=javax.management.ObjectName$Property@582e822b}  result is null duration is 2957  java.util.HashMap#put called in{name=javax.management.ObjectName$Property@10dc9047, type=javax.management.ObjectName$Property@582e822b}  result is null duration is 2683 |

　　获取截取方法内部调用实例以及方法

|  |  |
| --- | --- |
| 1  2  3  4  5 | @OnMethod(clazz="+java.util.Map",method="get",location=@Location(value=Kind.CALL, clazz="/.\*/", method="/.\*/"))       public static void testA(@ProbeClassName String pcm, @TargetInstance Object target, @TargetMethodOrField String field){             println(strcat("target is",strcat(strcat(str(target),"#"),field)));       } |

　　输出的结果是:

|  |  |
| --- | --- |
| 1  2 | target isjava.io.ObjectStreamClass$WeakClassKey@3952cb51#hashCode  target isjava.io.ObjectStreamClass$WeakClassKey@3952cb51#equals |

### @OnTimer

   用于指定跟踪操作定时执行。value用于指定时间间隔

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import com.sun.btrace.annotations.\*;    import static com.sun.btrace.BTraceUtils.\*;    import java.util.Map;  import java.util.concurrent.atomic.AtomicInteger;      @BTrace  public class TracingScript {      private static Map<String, AtomicInteger> histo = Collections.newHashMap();        @OnMethod(clazz = "+java.util.Map", method = "put", location = @Location(value = Kind.RETURN, clazz = "/.\*/", method = "/.\*/"))      public static void testB(@ProbeClassName String pcm, @ProbeMethodName String pmn, @Self Object self, @Duration long duration, @Return Object result) {            String cn = Reflective.name(classOf(self));          AtomicInteger ai = Collections.get(histo, cn);          if (ai == null) {              ai = Atomic.newAtomicInteger(1);              Collections.put(histo, cn, ai);          } else {              Atomic.incrementAndGet(ai);          }          }        @OnTimer(1000)      public static void print() {          if (Collections.size(histo) != 0) {              printNumberMap("Component Histogram", histo);          }      }  } |

　　输出的结果是:

|  |  |
| --- | --- |
| 1  2 | \* Component Histogram \*  java.util.HashMap = 130 |

### @OnError

   当trace代码抛异常或者错误时，该注解的方法会被执行.如果同一个trace脚本中其他方法抛异常,该注解方法也会被执行。

### @OnEvent

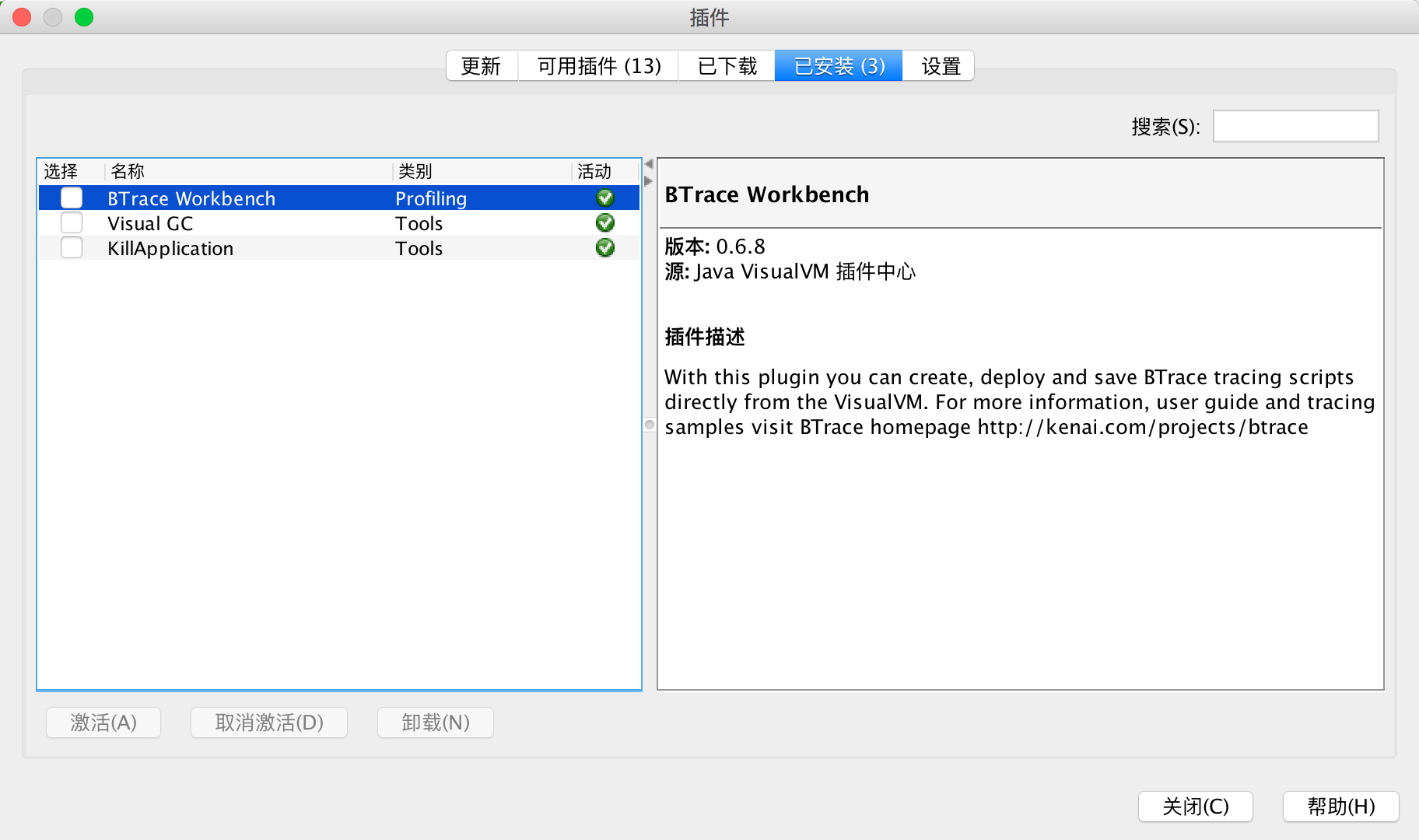
  通过btrace client事件进行触发,在jvisualvm 通过发送事件 就可以触发如下代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @OnEvent      public static void printEvent(){          if(Collections.size(histo)!=0){                  printNumberMap("Component Histogram", histo);              }       } |

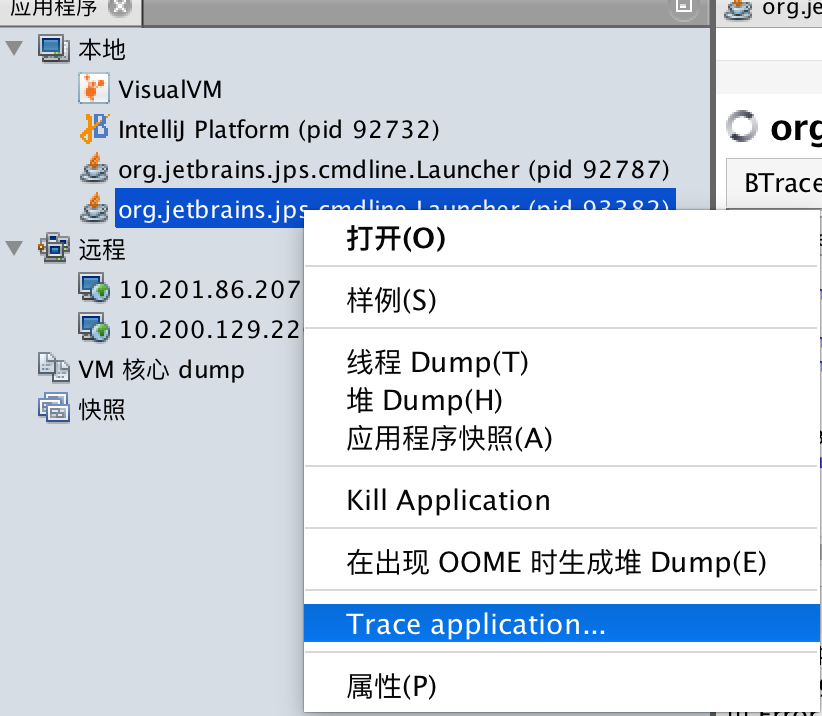
|  |  |
| --- | --- |
| 1  2 | \* Component Histogram \*  java.util.HashMap = 131 |

## 在jvisualvm中使用BTrace

   首先选择 工具->插件 安装BTrace插件



选择需要跟踪的工程,



## BTraceUtils方法介绍

* 获取当前线程 BTraceUtils.currentThread()
* 获取当前线程名称 BTraceUtis.Threads.name(BTraceUtils.currentThread())
* 打印当前线程jstack BTraceUtils.Threads.jstack()
* 获取当前时间 timestamp()
* 获取当前时间 毫秒值  millis()
* 创建StringBuilder newStringBuilder()
* 追加字符串append()

|  |  |
| --- | --- |
| 1  2  3  4 | Appendable builder=Strings.newStringBuilder();  Strings.append(builder,"当前时间：");  Strings.append(builder,timestamp());  println(str(builder)); |

* 字符串拼接 strcat("str1","str2")  concat("str1","str2")
* 字符串比较strcmp("str1","str2")
* 截取字符串 substr(“str",0,1)
* 获取字符串长度  strlen("str")
* 字符串是否匹配指定的正则表达式 matches(regex, input)
* 将对象转换为字符串 str()

## 1 使用场景

1. 服务慢，能找出慢在哪一步，哪个函数里么？
2. 谁构造了一个超大的ArrayList?
3. 什么样的入参或对象属性，导致抛出了这个异常？或进入了这个处理分支？

## 2 限制

为了避免Btrace脚本的消耗过大影响真正业务，所以定义了一系列不允许的事情：比如不允许调用任何类的任何方法，只能调用BTraceUtils 里的一系列方法和脚本里定义的static方法。 比如不允许创建对象，比如不允许For 循环等等，更多规定看User Guide

## 3 下载

<http://github.com/btraceio/btrace>，在Release页面里下载最新版本，解压就可以使用

## 4 基本命令

./btrace $pid HelloWorld.java

./btrace -o mylog.log $pid HelloWorld.java

很坑新人的参数，首先，这个mylog会生成在应用的启动目录，而不是btrace的启动目录。其次，执行过一次-o之后，再执行btrace不加-o 也不会再输出回console，直到应用重启为止。

./btrace $pid HelloWorld.java > mylog.log

预编译脚本

./btracec HelloWorld.java

## 5 拦截方法定义

### 5.1 精准定位

@OnMethod(clazz="cn.fraudmetrix.forseti.api.module.screen.RiskService", method="execute")

public static void execute() {

println("ttt");

}

### 5.2 正则

//下例监控javax.swing下的所有类的所有方法....可能会非常慢，建议范围还是窄些

@OnMethod(clazz="/javax\\.swing\\..\*/", method="/.\*/")

public static void swingMethods( @ProbeClassName String probeClass, @ProbeMethodName String probeMethod) {

print("entered " + probeClass + "." + probeMethod);

}

### 5.3 按接口，父类，Annotation定位

比如我想匹配所有的Filter类，在接口或基类的名称前面，加个+ 就行

@OnMethod(clazz="+com.vip.demo.Filter", method="doFilter")

也可以按类或方法上的annotaiton匹配，前面加上@就行

@OnMethod(clazz="@javax.jws.WebService", method="@javax.jws.WebMethod")

### 5.4 其他

1. 构造函数的名字是 <init>

@OnMethod(clazz="java.net.ServerSocket", method="<init>")

2. 静态内部类的写法，是在类与内部类之间加上"$"

@OnMethod(clazz="com.vip.MyServer$MyInnerClass", method="hello")

3. 如果有多个同名的函数，想区分开来，可以在拦截函数上定义不同的参数列表（见4.1）。

## 6 拦截时机

### 6.1 Kind.Entry与Kind.Return

@OnMethod( clazz="java.net.ServerSocket", method="bind" )

不写Location，默认就是刚进入函数的时候(Kind.ENTRY)。

OnMethod(clazz = "java.net.ServerSocket", method = "getLocalPort", location = @Location(Kind.RETURN))

public static void onGetPort(@Return int port, @Duration long duration)

duration的单位是纳秒，要除以 1,000,000 才是毫秒。

### 6.2 Kind.Error, Kind.Throw和 Kind.Catch

异常抛出(Throw)，异常被捕获(Catch)，异常没被捕获被抛出函数之外(Error)，主要用于对某些异常情况的跟踪。

在拦截函数的参数定义里注入一个Throwable的参数，代表异常。

@OnMethod(clazz = "java.net.ServerSocket", method = "bind", location = @Location(Kind.ERROR))

public static void onBind(Throwable exception, @Duration long duration)

### 6.3 Kind.Call与Kind.Line

* 监控bind()函数里调用的所有其他函数：

@OnMethod(clazz = "java.net.ServerSocket", method = "bind", location = @Location(value = Kind.CALL, clazz = "/.\*/", method = "/.\*/", where = Where.AFTER))

public static void onBind(@Self Object self, @TargetInstance Object instance, @TargetMethodOrField String method, @Duration long duration)

所调用的类及方法名所注入到@TargetInstance与 @TargetMethodOrField中。

注意这里，一定不要像下面这样大范围的匹配，否则这性能是神仙也没法救了

@OnMethod(clazz = "/javax\\.swing\\..\*/", method = "/.\*/", location = @Location(value = Kind.CALL, clazz = "/.\*/", method = "/.\*/"))

* 1
* 监控代码是否到达了Socket类的第363行

@OnMethod(clazz = "java.net.ServerSocket", location = @Location(value = Kind.LINE, line = 363))

public static void onBind4() {

println("socket bind reach line:363");

}

line还可以为-1，然后每行都会打印出来，加参数int line 获得的当前行数。此时会显示函数里完整的执行路径，但肯定又非常慢。

## 7 打印this，参数 与 返回值

import com.sun.btrace.AnyType;

@OnMethod(clazz = "java.io.File", method = "createTempFile", location = @Location(value = Kind.RETURN))

public static void o(@Self Object self, String prefix, String suffix, @Return AnyType result)

如果想打印它们，首先按顺序定义用@Self 注释的this， 完整的参数列表，以及用@Return 注释的返回值。

需要打印哪个就定义哪个，不需要的就不要定义。但定义一定要按顺序，比如参数列表不能跑到返回值的后面。

### 7.1 Self

如果是静态函数， self为空。   
前面提到，如果上述使用了非JDK的类，命令行里要指定classpath。不过，如前所述，因为BTrace里不允许调用类的方法，所以定义具体类很多时候也没意思，所以self定义为Object就够了。

### 7.2 参数

参数数列表要么不要定义，要定义就要定义完整，否则BTrace无法处理不同参数的同名函数。   
如果有些参数你实在不想引入非JDK类，又不会造成同名函数不可区分，可以用AnyType来定义（不能用Object）。   
如果拦截点用正则表达式中匹配了多个函数，函数之间的参数个数不一样，你又还是想把参数打印出来时，可以用AnyType[] args来定义。   
但不知道是不是当前版本的bug，AnyType[] args 不能和 location＝Kind.RETURN 同用，否则会进入一种奇怪的静默状态，只要有一个函数定义错了，整个Btrace就什么都打印不出来。

### 7.3 结果

同理，结果也可以用AnyType来定义，特别是用正则表达式匹配多个函数的时候，连void都可以表示。

### 7.4 打印

如果想打印一个Object的属性，用printFields()来反射。   
如果只想反射某个属性，参照下面打印Port属性的写法。从性能考虑，应把field用静态变量缓存起来。

如果只想反射某个属性，参照下面打印Port属性的写法。从性能考虑，应把field用静态变量缓存起来。

注意JDK类与非JDK类的区别：

import java.lang.reflect.Field;

//JDK的类这样写就行

private static Field fdFiled = field("java.io,FileInputStream", "fd");

//非JDK的类，要给出ClassLoader，否则ClassNotFound

private static Field portField = field(classForName("com.vip.demo.MyObject", contextClassLoader()), "port");

public static void onChannelRead(@Self Object self) {

println("port:" + getInt(portField, self));

}

## 8 TLS，拦截函数间的通信机制

如果要多个拦截函数之间要通信，可以使用@TLS定义 ThreadLocal的变量来共享

@TLS

private static int port = -1;

@OnMethod(clazz = "java.net.ServerSocket", method = "<init>")

public static void onServerSocket(int p){

port = p;

}

@OnMethod(clazz = "java.net.ServerSocket", method = "bind")

public static void onBind(){

println("server socket at " + port);

}

## 9 案例

### 9.1 打印慢调用

下例打印所有用时超过1毫秒的filter。

@OnMethod(clazz = "+com.vip.demo.Filter", method = "doFilter", location = @Location(Kind.RETURN))

public static void onDoFilter2(@ProbeClassName String pcn, @Duration long duration) {

if (duration > 1000000) {

println(pcn + ",duration:" + (duration / 100000));

}

}

最好能抽取了打印耗时的函数，减少代码重复度。

定位到某一个Filter慢了之后，可以直接用Location(Kind.CALL)，进一步找出它里面的哪一步慢了。

### 9.2 谁调用了这个函数

@OnMethod(clazz = "java.lang.System", method = "gc")

public static void onSystemGC() {

println("entered System.gc()");

jstack();

}

### 9.3 其他

按照 sample目录下的demo去学习

## 样例代码

package samples;

import com.sun.btrace.BTraceUtils;

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

import com.sun.btrace.annotations.Export;

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

@BTrace

public class OnlineDebug {

@OnExit//当成程序退出时，执行一些命令

public static void onexit(int code) {

println("BTrace program exits! with code: " + code);

}

@Export //可以用来统计调用次数

public static long counter;

@OnMethod(clazz="com.test.BtraceTest", method="add",

location=@Location(value=Kind.RETURN))

public static void m(@Self Object self, int a,int b,@Return int result,@Duration long time) {

BTraceUtils.println("参数： a: "+a+" b: "+b);

BTraceUtils.println("花费时间： "+time\*1.0/1000+"ms");

jstack();//打印堆栈信息

counter++;

}

@OnEvent("1")//通过事件触发，打印当前的程序调用次数

public static void setL1() {

BTraceUtils.println("executor count： "+counter);

}

//监控程序是否走到第22行代码

@OnMethod(clazz = "com.test.BtraceTest", location = @Location(value = Kind.LINE, line = 22))

public static void onBind() {

println("执行到第22行");

}

//每隔指定时间打印一下调用次数

@OnTimer(5000)

public static void run(){

BTraceUtils.println("executor count： "+counter);

}

}