## Btrace

### BTrace Restrictions

To guarantee that the tracing actions are "read-only" (i.e., the trace actions don't change the state of the program traced) and bounded (i.e., trace actions terminate in bounded time), a BTrace program is allowed to do only a restricted set of actions. In particular, a BTrace class

* can **not** create new objects.
* can **not** create new arrays.
* can **not** throw exceptions.
* can **not** catch exceptions.
* can **not** make arbitrary instance or static method calls - only the public static methods of com.sun.btrace.BTraceUtils class or methods declared in the same program may be called from a BTrace program.
* **(pre 1.2)** can **not** have instance fields and methods. Only static public void returning methods are allowed for a BTrace class. And all fields have to be static.
* can **not** assign to static or instance fields of target program's classes and objects. But, BTrace class can assign to it's own static fields ("trace state" can be mutated).
* can **not** have outer, inner, nested or local classes.
* can **not** have synchronized blocks or synchronized methods.
* can **not** have loops (for, while, do..while)
* can **not** extend arbitrary class (super class has to be java.lang.Object)
* can **not** implement interfaces.
* can **not** contains assert statements.
* can **not** use class literals.

These restrictions could be circumvented by running BTrace in unsafe mode. Both the tracing script and the engine must be set up to require unsafe mode. The script must be annotated by @BTrace(unsafe = true) annotation and the engine must be started in unsafe mode.

### 从入门到熟练小工的手册

**作者介绍**

江南白衣的Java后端开发，唯品会基础架构，微服务实践的日常。

**01**

**前言**

**\_\_\_\_\_**

BTrace是神器，每一个需要每天解决线上问题，但完全不用BTrace的Java工程师，都是可疑的 -- 凯尔文. 萧

BTrace的最大好处，是可以通过自己编写的脚本，获取应用的一切调用信息。而不需要不断地修改代码，加入System.out.println()， 然后重启，然后重启，然后重启应用！！！

同时，特别严格的约束，保证自己的消耗特别小，只要定义脚本时不作大死，直接在生产环境打开也没影响。

在网上搜索BTrace出来的文章都有点旧了，而且不够详细，于是决定，重新写一份，这个版本顺便把代码改成可C&P的。

**02**

**概述**

**\_\_\_\_\_**

2.1 快速开始

BTrace搬家了!! 已经从 Sun 搬到了

http://github.com/btraceio/btrace，当前版本已是1.38。

在Release页面里下载最新Zip版，解压就能用，UserGuide和Samples也在里面。

先抄一个UserGuide里的例子：

import com.sun.btrace.annotations.\*;  
  
import static com.sun.btrace.BTraceUtils.\*;  
  
  
@BTrace  
  
public class HelloWorld {  
  
    @OnMethod(clazz="java.lang.Thread", method="start")  
  
    public static void onThreadStart() {  
  
        println("thread start!");  
  
    }  
  
}  
}

然后ps找出要监控的java应用的pid，*./btrace $pid HelloWorld.java* 就跑起来了。

是不是很简单？？基本上不用任何BTrace的知识，都能猜出HelloWorld会干啥。通过JVM Attach API，BTrace把自己绑进了被监控的进程，按HelloWorld.java里的定义，进行AOP式的代码植入。

最开心就是这里，如果还想监控其他内容，直接修改HelloWorld.java，再执行一次btrace就可以了，不需要重启应用!! 重启应用!!

2.2 典型的场景

1. 服务慢，能找出慢在哪一步，哪个函数里么？

2. 谁调用了System.gc()，调用栈为何？

3. 谁构造了一个超大的ArrayList?

4. 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？

2.3 一些重要事情

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：比如只能调用BTraceUtils 里的一系列方法和脚本里定义的static方法，不允许其他调用任何类的任何方法。 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。

当然，可以用-u 运行在unsafe mode来规避限制，但不推荐。

在以前的例子里，甚至还不能字符串相加，必须用strcat:

println(strcat(strcat(probeClass, "."), probeMethod));

好在新版里已经可以写回：

println(probeClass + '.' + probeMethod);

另外，BTrace植入过的代码，会一直在，直到应用重启为止。所以即使BTrace退出了，业务函数每次执行时都会多出一次BTrace是否Attach状态的判断。

2.4 其他命令行选项

2.4.1 定义classpath

如果在HelloWorld.java里使用了JDK外的其他类，比如Netty的:

./btrace -cp .:netty-all-4.0.41.Final.jar $pid HelloWorld.java

但上面定义的classpath只在编译脚本时使用，而脚本里需要显式使用非JDK类的机会其实很少(后面真正用到的时候会提起)。

而在运行时，因为已经绑到目标应用的JVM里，用的是目标JVM的classpath。

2.4.2 结果输出到文件

./btrace -o mylog $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。

所以最好直接用转向了事：

./btrace $pid HelloWorld.java > mylog

2.4.3.预编译脚本

虽然btrace可以实时编译Java源文件，但如果你的脚本是要给运维同学执行的，线上运行时才发现写错了就尴尬了。此时可以用**btracec**预编译一下：

./btracec HelloWorld.java

接下来，开始一步步讲解脚本的编写。

**03**

**拦截方法定义**

**\_\_\_\_\_**

3.1 精准定位

就是HelloWorld的例子，精确定义要监控的类名与方法名。

3.2 正则表达式定位

批量定义需要监控的类与方法。正则表达式需要写在两个 "/" 中间。

下例监控javax.swing下的所有类的所有方法....会非常慢，实际使用时范围还是要窄些。

@OnMethod(clazz="/javax\.swing\..\*/", method="/.\*/")  
  
public static void swingMethods( @ProbeClassName String probeClass, @ProbeMethodName String probeMethod) {  
  
    print("entered " + probeClass + "."  + probeMethod);  
  
}

通过在拦截函数的参数定义里注入@ProbeClassName和 @ProbeMethodName参数，告诉脚本实际匹配到的类和方法名。

另一个例子，监控Statement的executeUpdate(), executeQuery() 和 executeBatch() 三个方法：

method = "/execute($|Update|Query|Batch)/"

3.3 按接口，父类，Annotation定位

要匹配所有Filter类，在接口或父类的名称前面，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，前面加个@就行

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

3.4 其他

构造函数的名字是*"<init>"*

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

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5.1）。

**04**

**拦截时机**

**\_\_\_\_\_**

可以为同一个函数的不同Location，分别定义多个拦截函数。

4.1 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)。

但如果你想获得函数的返回结果或执行时间，则必须把拦截点定在返回时。

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。

4.2 Kind.Error, Kind.Throw和 Kind.Catch

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

在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)

4.3 Kind.Call与Kind.Line

下例定义监控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。

静态函数中，instance的值为空。如果想获得执行时间，必须把Where定义成AFTER。

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 = "/.\*/"))

下例代码监控是否到达了Socket类的第363行。

@OnMethod(clazz = "java.net.ServerSocket", location = @Location(value = Kind.LINE, line = 363))  
  
public static void onBind4(int line) {  
  
    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。

**05**

**打印this，参数与返回值**

**\_\_\_\_\_**

5.1 定义参数注入

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。

**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。

**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。

**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。

5.2 打印

再次强调，为了保证性能不受影响，BTrace不允许调用任何实例方法。  
比如不能调用getter方法（怕在getter里有复杂的计算），只能通过直接反射来读取属性名。

又比如，除了JDK类，其他类toString时只会打印其类名＋System.IdentityHashCode。  
println, printArray，都按上面的规律进行，所以只能打打基本类型。

如果想打印一个Object的所有属性，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，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));  
  
}

5.3.TLS，拦截函数们之间间的通信机制

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);  
  
}

**06**

**典型场景**

**\_\_\_\_\_**

6.1 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。

6.2 谁调用了这个函数

比如，谁调用了System.gc() ?

@OnMethod(clazz = "java.lang.System", method = "gc")  
  
public static void onSystemGC() {  
  
    println("entered System.gc()");  
  
    jstack();  
  
}

6.3 捕捉异常，或进入了某个特定代码行时，this对象及参数的值

按之前的提示，自己组合一下即可。

6.4 打印函数的调用的统计信息

如果你已经看到了这里，那基本也不用我再啰嗦了，自己看Samples里的

Histogram.java 和 HistoOnEvent.java

可以用AtomicInteger构造计数器，然后定时(@OnTimer)，或根据事件(@OnEvent)输出结果，ctrl+c后选择发送事件。

**07**

**小结**

**\_\_\_\_\_**

恭喜你最后看到这里，成为了一个及格的BTrace小工，从此线上Java地界里的事情，没有瞒得过你的了。

### 从如何在生产环境使用Btrace进行调试

#### 背景

记得前几天有人问我：在生产环境中可能经常遇到各种问题，你们一般是如何进行调试的？ 很惭愧，没有经验。因为平时碰不到生产环境的服务器，定位问题需要各种数据，所以大多数问题的解决方式都是在本地打断点进行调试，或者在测试环境利用输出日志进行调试，这种方式简单粗暴，但过程比较繁琐，需要各种重新发布，重启应用，还不能保证一次就找到问题的根源。直到最近才了解到Btrace这个工具，对于这样一个神器，我觉得有必要记录一篇，让更多的人知道。

#### Btrace

BTrace是sun公司推出的一款Java 动态、安全追踪（监控）工具，可以在不用重启的情况下监控系统运行情况，方便的获取程序运行时的数据信息，如方法参数、返回值、全局变量和堆栈信息等，并且做到最少的侵入，占用最少的系统资源。

项目地址：[Btrace](https://link.jianshu.com?t=http%3A%2F%2Fgithub.com%2Fbtraceio%2Fbtrace)  
用户指南：[UserGuide](https://link.jianshu.com?t=https%3A%2F%2Fkenai.com%2Fprojects%2Fbtrace%2Fpages%2FUserGuide)

由于Btrace会把脚本逻辑直接侵入到运行的代码中，所以在使用上做很多限制：  
1、不能创建对象  
2、不能使用数组  
3、不能抛出或捕获异常  
4、不能使用循环  
5、不能使用synchronized关键字  
6、属性和方法必须使用static修饰

根据官方声明，不恰当的使用BTrace可能导致JVM崩溃，如在BTrace脚本使用错误的class文件，所以在上生产环境之前，务必在本地充分的验证脚本的正确性。

#### Btrace可以做什么？

1、接口性能变慢，分析每个方法的耗时情况；  
2、当在Map中插入大量数据，分析其扩容情况；  
3、分析哪个方法调用了System.gc()，调用栈如何；  
4、执行某个方法抛出异常时，分析运行时参数；  
5、....

#### Btrace第一个例子

package com.metty.rpc.common;

import java.util.Random;

/\*\*

\* Created by j\_zhan on 2016/11/28.

\*/

public class BtraceCase {

public static Random random = new Random();

public int size;

public static void main(String[] args) throws Exception {

new BtraceCase().run();

}

public void run() throws Exception {

while (true) {

add(random.nextInt(10), random.nextInt(10));

}

}

public int add(int a, int b) throws Exception {

Thread.sleep(random.nextInt(10) \* 100);

return a + b;

}

}

执行add方法时，对传入参数、返回值以及执行耗时进行分析，btrace脚本：

通过jps命令获取pid为8454  
执行btrace 8454 Debug.java实现对运行代码的监控，输出结果如下：

可以发现，Btrace可以获取每次执行add方法时的数据，当然Btrace能做的远远不止这些，比如获取当前jvm堆使用情况、当前线程的执行栈等等。

#### 参数说明

##### @OnMethod

Btrace使用@OnMethod注解定义需要分析的方法入口

在@OnMethod注解中，需要指定class、method以及location等，**class**表明需要监控的类，**method**表明需要监控的方法，指定方式如下：  
1、使用全限定名：clazz="com.metty.rpc.common.BtraceCase", method="add"  
2、使用正则表达式：clazz="/javax\\.swing\\..\*/", method="/.\*/"  
3、使用接口：clazz="+com.ctrip.demo.Filter", method="doFilter"  
4、使用注解：clazz="@javax.jws.WebService", method=""@javax.jws.WebMethod"  
5、如果需要分析构造方法，需要指定method="<init>"

##### @Location

定义Btrace对方法的拦截位置，通过@Location注解指定，默认为Kind.ENTRY  
1、**Kind.ENTRY**：在进入方法时，调用Btrace脚本  
2、**Kind.RETURN**：方法执行完时，调用Btrace脚本，只有把拦截位置定义为Kind.RETURN，才能获取方法的返回结果@Return和执行时间@Duration

3、**Kind.CALL**：分析方法中调用其它方法的执行情况，比如在execute方法中，想获取add方法的执行耗时，必须把where设置成Where.AFTER

4、**Kind.LINE**：通过设置line，可以监控代码是否执行到指定的位置

5、**Kind.ERROR, Kind.THROW, Kind.CATCH**  
用于对某些异常情况的跟踪，包括异常抛出，异常被捕获，异常未捕获被抛出方法之外

#### 如何使用Btrace定位问题

1、找出所有耗时超过1ms的过滤器Filter

由于@Dutation返回的时间是纳秒级别，需要进行转换，如果定位一个Filter性能变慢，接着使用@Location(Kind.CALL)进行更细粒度的分析。

2、分析哪个方法调用了System.gc()，调用栈如何？

通过查看调用栈，可以很清楚的发现哪个类哪个方法调用了System.gc()

3、统计方法的调用次数，且每隔1分钟打印调用次数

Btrace的@OnTimer注解可以实现定时执行脚本中的一个方法

4、方法执行时，查看对象的实例属性值

通过反射机制，可以很方法的得到当前实例的属性值

#### 总结

Btrace能做的事情太多，但使用之前切记检查脚本的可行性，一旦Btrace脚本侵入到系统中，只有通过重启才能恢复。

我是占小狼  
坐标魔都，白天是上班族，晚上是知识的分享者  
如果读完觉得有收获的话，欢迎点赞加关注

作者：占小狼  
链接：https://www.jianshu.com/p/dbb3a8b5c92f  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

### BTrace: Wiki: UserGuide — Project Kenai

原文链接： [kenai.com](https://kenai.com/projects/btrace/pages/UserGuide)

ver. 1.2 (20101020)

**BTrace** is a safe, dynamic tracing tool for Java. BTrace works by dynamically (bytecode) instrumenting classes of a running Java program. BTrace inserts tracing actions into the classes of a running Java program and hotswaps the traced program classes.

#### BTrace Terminology

**Probe Point**

"location" or "event" at which a set of tracing statements are executed. Probe point is "place" or "event" of interest where we want to execute some tracing statements.

**Trace Actions or Actions**

Trace statements that are executed whenever a probe "fires".

**Action Methods**

BTrace trace statements that are executed when a probe fires are defined inside a static method a class. Such methods are called "action" methods.

#### BTrace Program Structure

A BTrace program is a plain Java class that has one or more

**public** **static** **void**

methods that are annotated with [BTrace annotations](https://zcfy.cc/original/btrace-wiki-userguide-mdash-project-kenai-952.html?t=unclaimed#btrace_anno). The annotations are used to specify traced program "locations" (also known as "probe points"). The tracing actions are specified inside the static method bodies. These static methods are referred as "action" methods.

#### BTrace Restrictions

To guarantee that the tracing actions are "read-only" (i.e., the trace actions don't change the state of the program traced) and bounded (i.e., trace actions terminate in bounded time), a BTrace program is allowed to do only a restricted set of actions. In particular, a BTrace class

* can **not** create new objects.
* can **not** create new arrays.
* can **not** throw exceptions.
* can **not** catch exceptions.
* can **not** make arbitrary instance or static method calls - only the public static methods of com.sun.btrace.BTraceUtils class or methods declared in the same program may be called from a BTrace program.
* **(pre 1.2)** can **not** have instance fields and methods. Only static public void returning methods are allowed for a BTrace class. And all fields have to be static.
* can **not** assign to static or instance fields of target program's classes and objects. But, BTrace class can assign to it's own static fields ("trace state" can be mutated).
* can **not** have outer, inner, nested or local classes.
* can **not** have synchronized blocks or synchronized methods.
* can **not** have loops (for, while, do..while)
* can **not** extend arbitrary class (super class has to be java.lang.Object)
* can **not** implement interfaces.
* can **not** contains assert statements.
* can **not** use class literals.

These restrictions could be circumvented by running BTrace in unsafe mode. Both the tracing script and the engine must be set up to require unsafe mode. The script must be annotated by @BTrace(unsafe = true) annotation and the engine must be started in unsafe mode.

#### A simple BTrace program (1.2)

// import all BTrace annotations

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

// import statics from BTraceUtils class

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

// @BTrace annotation tells that this is a BTrace program

@BTrace

**class** **HelloWorld** {

// @OnMethod annotation tells where to probe.

// In this example, we are interested in entry

// into the Thread.start() method.

@OnMethod(

clazz="java.lang.Thread",

method="start"

)

**void** **func**() {

sharedMethod(msg);

}

**void** **sharedMethod**(String msg) {

// println is defined in BTraceUtils

println(msg);

}

}

#### A simple BTrace program ( Steps to run BTrace

1. Find the process id of the target Java process that you want to trace. You can use jps tool to find the pid.
2. Write a BTrace program - you may want to start modifying one of the samples.
3. Run btrace tool by the following command line:
4. **btrace** <pid> <**btrace-script>**

#### BTrace Command Line

BTrace is run using the command line tool btrace as shown below:

btrace [-I <include-path>] [-p <port>] [-cp <classpath>] <pid> <btrace-script> [<args>]

where

* include-path is a set of include directories that are searched for header files. BTrace includes a simple preprocess with support for #define, #include and conditional compilation. It is **not** like a complete C/C++ preprocessor - but a useful subset. See the sample "ThreadBean.java". If -I is not specified, BTrace skips the preprocessor invocation step.
* port is the port in which BTrace agent listens. This is optional argument.
* classpath is set of directories, jar files where BTrace searches for classes during compilation. Default is ".".
* pid is the process id of the traced Java program
* btrace-script is the trace program. If it is a ".java", then it is compiled before submission. Or else, it is assumed to be pre-compiled [i.e., it has to be a .class] and submitted.

##### optional

* port is the server socket port at which BTrace agent listens for clients. Default is 2020.
* path is the classpath used for compiling BTrace program. Default is ".".
* args is command line arguments passed to BTrace program. BTrace program can access these using the built-in functions "$" and "$length".

#### Pre-compiling BTrace scripts

It is possible to precompile BTrace program using btracec script. btracec is a javac-like program that takes a BTrace program and produces a .class file.

btracec [-I <include-path>] [-**cp** <classpath>] [-d <directory>] <one-or-more-BTrace-.java-**files**>

where

* include-path is a set of include directories that are searched for header files. BTrace includes a simple preprocess with support for #define, #include and conditional compilation. It is **not** like a complete C/C++ preprocessor - but a useful subset. See the sample "ThreadBean.java". If -I is not specified, BTrace skips the preprocessor invocation step.
* classpath is the classpath used for compiling BTrace program(s). Default is "."
* directory is the output directory where compiled .class files are stored. Default is ".".

This script uses BTrace compiler class - rather than regular javac and therefore will validate your BTrace program at compile time [so that you can avoid BTrace verify error at runtime].

#### Starting an application with BTrace agent

So far, we saw how to trace a running Java program. It is also possible to start an application with BTrace agent in it. If you want to start tracing the application from the very "beginning", you may want to start the app with BTrace agent and specify trace scripts along with it [i.e., BTrace agent is attach-on-demand loadable as well as pre-loadable agent] You can use the following command to start an app and specify BTrace script files. But, you need to [precompile your BTrace scripts](https://zcfy.cc/original/btrace-wiki-userguide-mdash-project-kenai-952.html?t=unclaimed#precompile) for this kind of usage.

java -javaagent:btrace-agent.jar=script=<pre-compiled-btrace-script1>[,<pre-compiled-btrace-script1>]\* <MainClass> <AppArguments>

When starting the application this way, the trace output goes to a file named .btrace in the current directory. Also, you can avoid starting server for other remote BTrace clients by specifying noServer=true as an argument to the BTrace agent. There is a convenient script called **btracer** to do the above:

btracer <pre-compiled-btrace.**class**> <application-main-**class**> <application-**args**>

##### Supported Arguments

* **bootClassPath** - boot classpath to be used
* **systemClassPath** - system classpath to be used
* **debug** - turns on verbose debug messages (true/false)
* **unsafe** - do not check for btrace restrictions violations (true/false)
* **dumpClasses** - dump the transformed bytecode to files (true/false)
* **dumpDir** - specifies the folder where the transformed classes will be dumped to
* **stdout** - redirect the btrace output to stdout instead of writing it to an arbitrary file (true/false)
* **probeDescPath** - the path to search for probe descriptor XMLs
* **script** - the path to a script to be run at the agent startup
* **scriptdir** - the path to a directory containing scripts to be run at the agent startup
* **scriptOutputFile** - the path to a file the btrace agent will store its output

##### Important system properties

* **btrace.agentname** - use to distinguish the outputs of various btrace agents running on the same machine

#### BTrace Annotations

##### Method Annotations

* [@com.sun.btrace.annotations.OnMethod](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/OnMethod.html) annotation can be used to specify target class(es), target method(s) and "location(s)" within the method(s). An action method annotated by this annotation is called when the matching method(s) reaches specified the location. In OnMethod annotation, traced class name is specified by "clazz" property and traced method is specified by "method" property. "clazz" may be a fully qualified class name (like **java.awt.Component** or a regular expression specified within two forward slashes. Refer to the samples [NewComponent.java](https://kenai.com/projects/btrace/sources/hg/content/samples/NewComponent.java) and [Classload.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Classload.java?rev=317). The regular expression can match zero or more classes in which case all matching classes are instrumented. For example **/java\.awt\..+/** matches all classes in **java.awt package**. Also, method name can be a regular expression as well - matching zero or more methods. Refer to the sample [MultiClass.java](https://kenai.com/projects/btrace/sources/hg/content/samples/MultiClass.java). There is another way to abstractly specify traced class(es) and method(s). Traced classes and methods may be specified by annotation. For example, if the "clazz" attribute is specified as [\*\*@javax.jws.WebService](mailto:**@javax.jws.WebService)**BTrace will instrument all classes that are annotated by the WebService annotation. Similarly, method level annotations may be used to specify methods abstractly. Refer to the sample**[**WebServiceTracker.java**](https://kenai.com/projects/btrace/sources/hg/content/samples/WebServiceTracker.java)**. It is also possible to combine regular expressions with annotations - like**@/com\.acme\..+/**matches any class that is annotated by any annotation that matches the given regular expression. It is possible to match multiple classes by specifying super type. i.e., match all classes that are subtypes of a given super type.**+java.lang.Runnable**matches all classes implementing**java.lang.Runnable\*\* interface. Refer to the sample [SubtypeTracer.java](https://kenai.com/projects/btrace/sources/hg/content/samples/SubtypeTracer.java).
* [@com.sun.btrace.annotations.OnTimer](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/OnTimer.html) annotation can be used to specify tracing actions that have to run periodically once every N milliseconds. Time period is specified as long "value" property of this annotation. Refer to the sample [Histogram.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Histogram.java)
* [@com.sun.btrace.annotations.OnError](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/OnError.html) annotation can be used to specify actions that are run whenever any exception is thrown by tracing actions of some other probe. BTrace method annotated by this annotation is called when any exception is thrown by any of the other BTrace action methods in the same BTrace class.
* [@com.sun.btrace.annotations.OnExit](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/OnExit.html) annotation can be used to specify actions that are run when BTrace code calls "exit(int)" built-in function to finish the tracing "session". Refer to the sample [ProbeExit.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ProbeExit.java).
* [@com.sun.btrace.annotations.OnEvent](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/OnEvent.html) annotation is used to associate tracing methods with "external" events send by BTrace client. BTrace methods annotated by this annotation are called when BTrace client sends an "event". Client may send an event based on some form of user request to send (like pressing Ctrl-C or a GUI menu). String value may used as the name of the event. This way certain tracing actions can be executed whenever an external event "fires". As of now, the command line BTrace client sends "events" whenever use presses Ctrl-C (SIGINT). On SIGINT, a console menu is shown to send an event or exit the client [which is the default for SIGINT]. Refer to the sample [HistoOnEvent.java](https://kenai.com/projects/btrace/sources/hg/content/samples/HistoOnEvent.java)
* [@com.sun.btrace.annotations.OnLowMemory](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/OnLowMemory.html) annotation can be used to trace memory threshold exceed event. See sample [MemAlerter.java](https://kenai.com/projects/btrace/sources/hg/content/samples/MemAlerter.java)
* [@com.sun.btrace.annotations.OnProbe](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/OnProbe.html) annotation can be used to specify to avoid using implementation internal classes in BTrace scripts. @OnProbe probe specifications are mapped to one or more @OnMethod specifications by the BTrace VM agent. Currently, this mapping is done using a XML probe descriptor file [accessed by the BTrace agent]. Refer to the sample [SocketTracker1.java](https://kenai.com/projects/btrace/sources/hg/content/samples/SocketTracer1.java) and associated probe descriptor file java.net.socket.xml. When running this sample, this xml file needs to be copied in the directory where the target JVM runs (or fix probeDescPath option in btracer.bat to point to whereever the .xml file is).

##### Argument Annotations

* [@com.sun.btrace.annotations.Self](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/Self.html) annotation can be used to mark an argument to hold **this** (or **self**) value. Refer to the samples [AWTEventTracer.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AWTEventTracer.java) or [AllCalls1.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AllCalls1.java?rev=404)
* [@com.sun.btrace.annotations.Return](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/Return.html) annotation can be used to mark an argument to hold the return value. Refer to the sample [Classload.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Classload.java)
* [@com.sun.btrace.annotations.ProbeClassName (since 1.1)](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/ProbeClassName.html) annotation can be used to mark an argument to hold the probed class name value. Refer to the sample [AllMethods.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AllMethods.java?rev=404)
* [@com.sun.btrace.annotations.ProbeMethodName (since 1.1)](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/ProbeMethodName.html) annotation can be used to mark an argument to hold the probed method name. Refer to the sample [WebServiceTracker.java](https://kenai.com/projects/btrace/sources/hg/content/samples/WebServiceTracker.java)
* since 1.2 it accepts boolean parameter **fqn** to get a fully qualified probed method name
* [@com.sun.btrace.annotations.Duration](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/Duration.html) annotation can be used to mark an argument to hold the duration of the method call in nanoseconds. The argument must be a long. Use with **Kind.RETURN** or **Kind.ERROR** locations.
* [@com.sun.btrace.annotations.TargetInstance (since 1.1)](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/TargetInstance.htm) annotation can be used to mark an argument to hold the called instance value. Refer to the sample [AllCalls2.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AllCalls2.java?rev=404)
* [@com.sun.btrace.annotations.TargetMethodOrField (since 1.1)](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/TargetMethodOrField.html) can be used to mark an argument to hold the called method name. Refer to the samples [AllCalls1.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AllCalls1.java?rev=404) or [AllCalls2.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AllCalls2.java?rev=404)
* since 1.2 it accepts boolean parameter **fqn** to get a fully qualified target method name

###### Unannotated arguments

The unannotated BTrace probe method arguments are used for the signature matching and therefore they must appear in the order they are defined in the traced method. However, they can be interleaved with any number of annotated arguments. If an argument of type AnyType[] is used it will "eat" all the rest of the arguments in they order. The exact meaning of the unannotated arguments depends on the **Location** used:

* **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

##### Field Annotations

* [@com.sun.btrace.annotations.Export](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/Export.htm) annotation can be used with BTrace fields (static fields) to specify that the field has to be mapped to a jvmstat counter. Using this, a BTrace program can expose tracing counters to external jvmstat clients (such as jstat). Refer to the sample [ThreadCounter.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ThreadCounter.java)
* [@com.sun.btrace.annotations.Property](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/Property.html) annotation can be used to flag a specific (static) field as a MBean attribute. If a BTrace class has atleast one static field with @Property attribute, then a MBean is created and registered with platform MBean server. JMX clients such as VisualVM, jconsole can be used to view such BTrace MBeans. After attaching BTrace to the target program, you can attach VisualVM or jconsole to the same program and view the newly created BTrace MBean. With VisualVM and jconsole, you can use MBeans tab to view the BTrace domain and check out it's attributes. Refer to the samples [ThreadCounterBean.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ThreadCountBean.java) and [HistogramBean.java](https://kenai.com/projects/btrace/sources/hg/content/samples/HistogramBean.java).
* [@com.sun.btrace.annotations.TLS](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/TLS.html) annotation can be used with BTrace fields (static fields) to specify that the field is a thread local field. Please, be aware that you can not access such marked fields in other than **@OnMethod** handlers. Each Java thread gets a separate "copy" of the field. In order for this to work correctly the field type must be either **immutable** (eg. primitives) or **cloneable** (implements Cloneable interface and overrides clone() method).These thread local fields may be used by BTrace programs to identify whether we reached multiple probe actions from the same thread or not. Refer to the samples [OnThrow.java](https://kenai.com/projects/btrace/sources/hg/content/samples/OnThrow.java) and [WebServiceTracker.java](https://kenai.com/projects/btrace/sources/hg/content/samples/WebServiceTracker.java)

##### Class Annotations

* [@com.sun.btrace.annotations.DTrace](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/DTrace.html) annotation can be used to associate a simple one-liner D-script (inlined in BTrace Java class) with the BTrace program. Refer to the sample [DTraceInline.java](https://kenai.com/projects/btrace/sources/hg/content/samples/DTraceInline.java?rev=317).
* [@com.sun.btrace.annotations.DTraceRef](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/DTraceRef.html) annotation can be used to associate a D-script (stored in a separate file) with the BTrace program. Refer to the sample [DTraceRefDemo.java](https://kenai.com/projects/btrace/sources/hg/content/samples/DTraceRefDemo.java?rev=317).
* [@com.sun.btrace.annotations.BTrace](http://btrace.kenai.com/javadoc/1.2/com/sun/btrace/annotations/BTrace.html) annotation must be used to designate a given Java class as a BTrace program. This annotation is enforced by the BTrace compiler as well as by the BTrace agent.

#### DTrace Integration

Solaris DTrace is a dynamic, safe tracing system for Solaris programs - both kernel and user land programs. Because of the obvious parallels b/w DTrace and BTrace, it is natural to expect integration b/w BTrace and DTrace. There are two ways in which BTrace is integrated with DTrace.

* BTrace program can raise a DTrace probe - by calling dtraceProbe -- see BTraceUtils javadoc referred above. For this feature to work, you need to be running on **Solaris 10 or beyond**. For other platforms (Solaris 9 or below or any other OS), dtraceProbe() will be a no-op.
* BTrace program can associate a D-script with it-- by @DTrace annotation (if the D-script is a simple one liner) or by @DTraceRef if the D-script is longer and hence stored outside of the BTrace program. See DTrace integration samples in the BTrace samples section below. This feature works using the . For this DTrace feature to work (o.e., being able to run associated D-script), you need to be running on **Solaris 11 build 35 or beyond**. You may want to check whether you have **/usr/share/lib/java/dtrace.jar** on your machine or not. @DTrace and @DTraceRef annotations are ignored on other platforms (Solaris 10 or below or any other OS).

#### BTrace Samples

[BTrace samples](https://kenai.com/projects/btrace/sources/hg/content/samples)

One liners about samples:

* [AWTEventTracer.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AWTEventTracer.java?rev=267) - demonstrates tracing of AWT events by instrumenting EventQueue.dispatchEvent() method. Can filter events by instanceof check. This sample filters and prints only focus events.
* [AllLines.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AllLines.java?rev=404) - demonstrates line number based BTrace probes. It is possible to probe into any class (or classes) and line number(s). When the specified line number(s) of specified class(es) is reached, the BTrace probe fires and the corresponding action is executed.
* [AllSync.java](https://kenai.com/projects/btrace/sources/hg/content/samples/AllSync.java?rev=404) - demonstrates tracing of a synchronized block entry/exit.
* [ArgArray.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ArgArray.java?rev=267) - prints all input arguments in every readXXX method of every class in java.io package. Demonstrates argument array access and multiple class/method matching in probe specifications.
* [Classload.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Classload.java?rev=267) - prints stack trace on every successful classload (defineClass returns) by any userdefined class loader.
* [CommandArg.java](https://kenai.com/projects/btrace/sources/hg/content/samples/CommandArg.java?rev=404) - demonstrates BTrace command line argument access.
* [Deadlock.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Deadlock.java?rev=317) - demonstrates @OnTimer probe and deadlock() built-in function.
* [DTraceInline.java](https://kenai.com/projects/btrace/sources/hg/content/samples/DTraceInline.java?rev=267) - demonstrates @DTrace annotation to associate a simple one-line D-script with the BTrace program.
* [DTraceDemoRef.java](https://kenai.com/projects/btrace/sources/hg/content/samples/DTraceDemoRef.java?rev=267) - demonstrates @DTraceRef annotation to associate a D-script file with the BTrace program. This sample associates classload.dwith itself.
* [FileTracker.java](https://kenai.com/projects/btrace/sources/hg/content/samples/FileTracker.java?rev=404) - prints file open for read/write by probing into File{Input/Output}Stream constructor entry points.
* [FinalizeTracker.java](https://kenai.com/projects/btrace/sources/hg/content/samples/FinalizeTracker.java?rev=404) - demonstrates that we can print all fields of an object and access (private) fields (read-only) as well. This is useful in debugging / troubleshooting scenarios. This sample prints info on close() /finalize() methods of java.io.FileInputStream class.
* [Histogram.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Histogram.java?rev=404) - demonstrates usage of BTrace maps for collecting histogram (of javax.swing.JComponent objects created by an app - histogram by subclass name and count).
* [HistogramBean.java](https://kenai.com/projects/btrace/sources/hg/content/samples/HistogramBean.java?rev=404) - demonstrates JMX integration. This sample exposes a Map as MBean attribute using @Property annotation.
* [HistoOnEvent.java](https://kenai.com/projects/btrace/sources/hg/content/samples/HistoOnEvent.java?rev=404) - demonstrates getting trace output based on client side event. After starting the script by BTrace client, press Ctrl-C to get menu to send event or exit. On sending event, histogram is printed. This way client can "pull" trace output whenever needed rather than BTrace agent pushing the trace output always or based on timer.
* [JdbcQueries.java](https://kenai.com/projects/btrace/sources/hg/content/samples/JdbcQueries.java?rev=408) - demonstrates BTrace aggregation facility. This facility is similar to DTrace aggregation facility.
* [JInfo.java](https://kenai.com/projects/btrace/sources/hg/content/samples/JInfo.java?rev=404) - demonstrates printVmArguments(), printProperties() and printEnv() built-in functions.
* [JMap.java](https://kenai.com/projects/btrace/sources/hg/content/samples/JMap.java?rev=404) - demonstrates dumpHeap() built-in function to dump (hprof binary format) heap dump of the target application.
* [JStack.java](https://kenai.com/projects/btrace/sources/hg/content/samples/JStack.java?rev=317) - demonstrates jstackAll() built-in function to print stack traces of all the threads.
* [LogTracer.java](https://kenai.com/projects/btrace/sources/hg/content/samples/LogTracer.java?rev=317) - demonstrates trapping into an instance method (Logger.log) and printing private field value (of LogRecord object) by field() and objectValue() built-in functions.
* [MemAlerter.java](https://kenai.com/projects/btrace/sources/hg/content/samples/MemAlerter.java?rev=267) - demonstrates tracing low memory event by @OnLowMememory annotation.
* [Memory.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Memory.java?rev=404) - prints memory stat once every 4 seconds by a timer probe. Demonstrates memory stat built-in functions.
* [MultiClass.java](https://kenai.com/projects/btrace/sources/hg/content/samples/MultiClass.java?rev=317) - demonstrates inserting trace code into multiple methods of multiple classes using regular expressions for clazz and method fields of @OnMethod annotation.
* [NewComponent.java](https://kenai.com/projects/btrace/sources/hg/content/samples/NewComponent.java?rev=317) - tracks every java.awt.Component creation and increments a counter and prints the counter based on a timer.
* [OnThrow.java](https://kenai.com/projects/btrace/sources/hg/content/samples/OnThrow.java?rev=317) - prints exception stack trace every time any exception instance is created. In most scenarios, exceptions are thrown immediately after creation. So, it we get stack trace of throw points.
* [ProbeExit.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ProbeExit.java?rev=317) - demonstrates @OnExit probe and exit(int) built-in function.
* [Profiling.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Profiling.java?rev=404) - demonstrates the usage of the profiling support
* [Sizeof.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Sizeof.java?rev=404) - demonstrates "sizeof" built-in function that can be used to get (approx.) size of a given Java object. It is possible to get size-wise histogram etc. using this built-in.
* [SocketTracker.java](https://kenai.com/projects/btrace/sources/hg/content/samples/SocketTracker.java?rev=404) - prints every server socker creation/bind and client socket accepts.
* [SocketTracker1.java](https://kenai.com/projects/btrace/sources/hg/content/samples/SocketTracker1.java?rev=404) - similar to SocketTracker.java sample, except that this sample uses @OnProbe probes to avoid using Sun specific socket channel implementation classes in BTrace program. Instead @OnProbe probes are mapped to @OnMethod probes by BTrace agent.
* [SysProp.java](https://kenai.com/projects/btrace/sources/hg/content/samples/SysProp.java?rev=317) - demonstrates that it is okay to probe into System classes (like java.lang.System) and call BTrace built-in functions in the action method.
* [SubtypeTracer.java](https://kenai.com/projects/btrace/sources/hg/content/samples/SubtypeTracer.java?rev=267) - demonstrates that it is possible to match all subtypes of a given supertype.
* [ThreadCounter.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ThreadCounter.java?rev=317) - demonstrates use of jvmstat counters from BTrace programs.
* [ThreadCounterBean.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ThreadCounterBean.java?rev=404) - demonstrates exposing the BTrace program as a JMX bean with one attribute (using @Property annotation).
* [ThreadBean.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ThreadBean.java?rev=267) - demonstrates the use of preprocessor of BTrace [and JMX integratio].
* [ThreadStart.java](https://kenai.com/projects/btrace/sources/hg/content/samples/ThreadStart.java?rev=317) - demonstrates raising DTrace probes from BTrace programs. See also jthread.d - the associated D-script. This sample raises a DTrace USDT probe whenever the traced program enters java.lang.Thread.start() method. The BTrace program passes JavaThread's name to DTrace.
* [Timers.java](https://kenai.com/projects/btrace/sources/hg/content/samples/Timers.java?rev=404) - demonstrates multiple timer probes (@OnTimer) in a BTrace program.
* [URLTracker.java](https://kenai.com/projects/btrace/sources/hg/content/samples/URLTracker.java?rev=404) - prints URL every time URL.openConnection returns successfully. This program uses jurls.d D-script as well (to show histogram of URLs opened via DTrace).
* [WebServiceTracker.java](https://kenai.com/projects/btrace/sources/hg/content/samples/WebServiceTracker.java?rev=404) - demonstrates tracing classes and methods by specifying class and method level annotations rather than class and method names.