BTrace User's Guide 用户指南

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往"正在运行的Java程序的类"中插入追踪行为，并热交换追踪的类。

BTrace Terminology 术语

Probe Point 探测点 "位置"或"事件"，一系列追踪语句在这里被执行。

"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 追踪行为或行为（追踪语句）

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

Action Methods 行为方法

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 程序结构

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

```
public static void
```

注解，用于指定追踪的程序"位置"("探测点")

methods that are annotated with BTrace annotations. 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 约束

追踪行为是只读和有界的

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) 一个简单的BTrace程序

```
// 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   告知这是一个BTrace程序
@BTrace
class HelloWorld {

    // @OnMethod annotation tells where to probe.   告知在哪里探测
    // In this example, we are interested in entry   在本例中，对Thread.start()方法的入口感兴趣
    // 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 (<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
public 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"
    )
    public static void func() {
        // println is defined in BTraceUtils
        // you can only call the static methods of BTraceUtils
        println("about to start a thread!");
    }
}
```

The above BTrace program can be run against a running Java process. This program will print "about to start a thread!" at the BTrace client whenever the target program is about to start a thread by Thread.start() method. There are other interesting probe points possible. For example, we can insert trace action at return from a method, exception return from a method, a field get or set within method(s), object/array creation, line number(s), throwing an exception. Please refer to the @OnMethod and other annotations for details.

Steps to run BTrace 运行BTrace步骤

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:

```
btrace <pid> <btrace-script>
```

1. 找出目标Java进程的ID
2. 编写BTrace程序
3. 通过以下命令行运行btrace工具

BTrace Command Line 命令行

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 预编译的BTrace脚本

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].

在编译时验证你的BTrace程序，避免在运行时验证错误

Starting an application with BTrace agent 开启BTrace代理应用

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 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-class-file-name>.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 支持的参数

- **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 注解

Method Annotations 方法注解

可用于指定目标类、目标方法和方法中的"位置"

- @com.sun.btrace.annotations.OnMethod 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 and Classload.java. 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. 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** 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. 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.
- @com.sun.btrace.annotations.OnTimer 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
- @com.sun.btrace.annotations.OnError 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 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.
- @com.sun.btrace.annotations.OnEvent 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
- @com.sun.btrace.annotations.OnLowMemory annotation can be used to trace memory threshold exceed event. See sample MemAlerter.java
- @com.sun.btrace.annotations.OnProbe 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 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 参数注解

标记一个参数来持有本实例(this)

- @com.sun.btrace.annotations.Self annotation can be used to mark an argument to hold **this** (or **self**) value. Refer to the samples AWTEventTracer.java or AllCalls1.java

标记一个参数来持有被追踪的方法的返回值

- @com.sun.btrace.annotations.Return annotation can be used to mark an argument to hold the return value. Refer to the sample Classload.java
- @com.sun.btrace.annotations.ProbeClassName (since 1.1) annotation can be used to mark an argument to hold the probed class name value. Refer to the sample AllMethods.java
- @com.sun.btrace.annotations.ProbeMethodName (since 1.1) annotation can be used to mark an argument to hold the probed method name. Refer to the sample WebServiceTracker.java
  - since 1.2 it accepts boolean parameter **fqn** to get a fully qualified probed method name
- @com.sun.btrace.annotations.Duration 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) annotation can be used to mark an argument to hold the called instance value. Refer to the sample AllCalls2.java
- @com.sun.btrace.annotations.TargetMethodOrField (since 1.1) can be used to mark an argument to hold the called method name. Refer to the samples AllCalls1.java or AllCalls2.java
  - since 1.2 it accepts boolean parameter **fqn** to get a fully qualified target method name

Unannotated arguments 未注解的参数

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 探测的方法参数
- **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 字段注解

- @com.sun.btrace.annotations.Export 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
- @com.sun.btrace.annotations.Property 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 and HistogramBean.java.
- @com.sun.btrace.annotations.TLS 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 and WebServiceTracker.java

Class Annotations 类型注解

- @com.sun.btrace.annotations.DTrace 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.
- @com.sun.btrace.annotations.DTraceRef annotation can be used to associate a D-script (stored in a separate file) with the BTrace program. Refer to the sample DTraceRefDemo.java.
- @com.sun.btrace.annotations.BTrace 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 示例

BTrace samples

One liners about samples:

- AWTEventTracer.java - 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 - 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 - demonstrates tracing of a synchronized block entry/exit. 追踪"同步块"进入/退出
- ArgArray.java - 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. 多个"类/方法"匹配
- Classload.java - prints stack trace on every successful classload (defineClass returns) by any userdefined class loader. 打印"栈踪迹"信息
- CommandArg.java - demonstrates BTrace command line argument access. "命令行参数"访问
- Deadlock.java - demonstrates @OnTimer probe and deadlock() built-in function. "死锁"探测
- DTraceInline.java - demonstrates @DTrace annotation to associate a simple one-line D-script with the BTrace program.
- DTraceDemoRef.java - demonstrates @DTraceRef annotation to associate a D-script file with the BTrace program. This sample associates classload.dwith itself.
- FileTracker.java - prints file open for read/write by probing into File{Input/Output}Stream constructor entry points. 在调试/故障排除中非常有用！
- FinalizeTracker.java - 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. 打印"对象的所有状态"
- Histogram.java - 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 - demonstrates JMX integration. This sample exposes a Map as MBean attribute using @Property annotation.
- HistoOnEvent.java - demonstrates getting trace output based on client side event. After starting the script by BTrace client, press Ctrl-C to get menu to senc 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 - demonstrates BTrace aggregation facility. This facility is similar to DTrace aggregation facility.
- JInfo.java - demonstrates printVmArguments(), printProperties() and printEnv() built-in functions.
- JMap.java - demonstrates dumpHeap() built-in function to dump (hprof binary format) heap dump of the target application. 转储目标应用的"堆信息"
- JStack.java - demonstrates jstackAll() built-in function to print stack traces of all the threads. 打印所有线程的"调用栈信息"
- LogTracer.java - demonstrates trapping into an instance method (Logger.log) and printing private field value (of LogRecord object) by field() and

捕捉到"一个实例方法"并打印"私有字段属性值"

objectValue() built-in functions.

- MemAlerter.java - demonstrates tracing low memory event by @OnLowMememory annotation.
- Memory.java - prints memory stat once every 4 seconds by a timer probe. Demonstrates memory stat built-in functions.
- MultiClass.java - demonstrates inserting trace code into multiple methods of multiple classes using regular expressions for clazz and method fields of @OnMethod annotation. **使用"正则表达式"来插入追踪代码到"多个类的多个方法"中**
- NewComponent.java - tracks every java.awt.Component creation and increments a counter and prints the counter based on a timer.
- OnThrow.java - 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. **打印任何异常类型被创建时的"异常栈踪迹信息"**
- ProbeExit.java - demonstrates @OnExit probe and exit(int) built-in function.
- Profiling.java - demonstrates the usage of the profiling support **"分析用法"支持**
- Sizeof.java - 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 - prints every server socker creation/bind and client socket accepts. **打印每个"服务器套接字"创建/绑定和"客户端套接字"接收信息**
- SocketTracker1.java - 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 - 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 - demonstrates that it is possible to match all subtypes of a given supertype. **匹配"给定超类的所有子类"** **调用"BTrace内建函数"**
- ThreadCounter.java - demonstrates use of jvmstat counters from BTrace programs. **使用"jvmstat计数器"**
- ThreadCounterBean.java - demonstrates exposing the BTrace program as a JMX bean with one attribute (using @Property annotation).
- ThreadBean.java - demonstrates the use of preprocessor of BTrace [and JMX integratio]. **"预处理器"的使用**
- ThreadStart.java - 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 - demonstrates multiple timer probes (@OnTimer) in a BTrace program. **多个"定时器"探测**
- URLTracker.java - 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 - demonstrates tracing classes and methods by specifying class and method level annotations rather than class and method names.

通过指定"类和方法层的注解"来追踪"类和方法"