java在线问题排查利器之Btrace&Greys

2017-10-11

本文主要介绍了两款java 排查问题的工具 Btrace 和 Greys。他们都比较适合对生成环境的问题进行排查，都属于“事后工具”，即服务已经上线了，无法再通过打印日志等方式埋点分析。这时可以使用这些工具，来跟踪代码执行耗时、堆栈情况等。

1. 背景说明

前段时间升级了urs新的远程cookie校验模式。功能上线后，发现涉及用户cookie 校验的接口，有时会报接口超时。通过日志埋点方式，确认了与urs提供的jar包内的新验证方法有关。通过反编译，看到相关方法执行过程中涉及参数校验、参数组装、远程访问校验、本地校验等步骤，究竟哪个步骤出了问题?

一种方式是让urs帮忙提供一个新的jar包，在关键步骤处加日志，记录执行时间，另一种方法，就是使用一些在线分析工具。显然第二种方式更方便快捷。本文主要介绍两款在线问题排的工具：**Btrace 和 Greys** 。

2. 工具简介

Btrace 和 Greys 都比较适合对生成环境的问题进行排查，都属于“事后工具”，即服务已经上线了，无法再通过打印日志等方式埋点分析。这时可以使用这些工具，来跟踪代码执行耗时、堆栈情况等。

原理

都是基于动态字节码修改技术(Hotswap)来实现运行时 java 程序的跟踪和替换。
利用了Java SE 6 新特性Instrumentation 。

使用场景

- 分析哪些方法慢，查询具体的故障点；

- 查看方法的参数、返回值；
- 查看对象属性等；

Btrace 和 Greys

Btrace 和 Greys 都属于工具，本文以实例，介绍如何使用，不再对其本身进行介绍，如果想进一步了解，可以直接去下面的链接：

类型	介绍
Btrace	https://github.com/btraceio/btrace/wiki
Greys介绍	https://yq.aliyun.com/articles/2390

3. 实例-使用工具排查问题

urs新提供了远程cookie的校验jar包，其中关键的方法为远程调用方法CookieDecoder.requestDecode(**)，我们主要对这个方法进行跟踪。

3.1 使用Btrace

由于我们的tomcat 在 appuser 用户下，为了有相应权限，我们的操作都在 appuser 用户下进行。
大体步骤分为：

- 下载解压 btrace 工具；
- 编写监控脚本；
- 设置jdk/btrace 环境变量，上传脚本，编译
- 获取tomcat 进程号
- 启动监控
- 查看详情

具体操作：

（1）下载btrace工具 btrace-bin-1.3.9.tgz 并解压缩

（2） 编写一个监控脚本（java代码 UrsInterfaceCalls.java），

即需要监控的具体类和方法

```
@BTrace                          // 备注1
public class UrsInterfaceCalls{

        /**
         * 备注2
         * 本代码用于监控 CookieDecoder 中 requestDecode 方法的执行时间，如果执行时间大于 500ms，则打印花费的时间和堆栈
         * @param duration
         */
        @OnMethod(
                        clazz="com.netease.urs.CookieDecoder",
                        method="requestDecode",
                        location=@Location(Kind.RETURN))    // 备注2
    public static void requestDecode( @Duration long duration ) {   // 备注3
                //备注4
                if(duration /1000000 > 500){
                        println("==CookieDecoder requestDecode spend: " + duration /1000000 + " ms");
                        jstack();
                }
    }

        /**
         * 本代码用于监控 CustomHttpComponent 中 execute 方法的执行时间，如果执行时间大于 500ms，则打印花费的时间和堆栈
         * @param duration
         */
        @OnMethod(
                        clazz="com.netease.urs.http.CustomHttpComponent",
                        method="execute",
                        location=@Location(Kind.RETURN))
    public static void execute( @Duration long duration ) {
                if(duration /1000000 > 500){
                        println("==ursCookieHttp doExecute spend: " + duration /1000000 + " ms");
                        jstack();
                }
    }
}
```

> 简要说明
>
> 本监控类，写了两个监控方法:
> 一个是监听CookieDecoder.requestDecode()的执行时间，如果大于500ms，则打印日志，并打印相关堆栈;
> 另一个监听CustomHttpComponent.execute()的执行时间。

备注1： 添加注释 @BTrace ， 代表本脚本将使用btrace相关功能;

备注2： 拦截方法定义 ， @OnMethod 可以指定 clazz 、 method、location。由此组成了在什么时机（location 决定）监控某个类/某些类（clazz 决定）下的某个方法/某些方法(method 决定)。

```
@OnMethod(clazz="com.netease.urs.CookieDecoder",method="requestDecode",location=@Location(Kind.RETURN))
```

意思是监控CookieDecoder.requestDecode() ， 在执行结束后

（location=@Location(Kind.RETURN) 执行相关操作。

备注3： @Duration 代表方法执行时间，纳秒。

备注4： 只打印 耗时超过500ms的信息及堆栈，防止记录打印大多。

> 方法注解说明
>
> - @OnMethod:指定使用当前注解的方法应该在什么情况下触发：
>   - claszz属性指定要匹配的类的全限定类名,可以用正则表达式;
>   - method属性指定要匹配的方法名称,可以用正则表达式;
>   - type属性void(java.lang.String)可以用于匹配:public void funcName(String param) 中的方法入参；
>   - location属性用@Location来表明,匹配了clazz,method情况,在方法执行的何时去执行脚本(前,后,异常,行,某个方法调用)
> - @OnTimer:指定一个定时任务
> - @OnExit:当脚本运行Sys.exit(code)时触发
> - @OnError:当脚本运行抛出异常时触发
> - @OnEvent:脚本运行时Ctrl+C可以发送事件
> - @OnLowMemory:让你指定一个阀值,内存低于阀值触发
> - @OnProbe:可以用一个xml文件来描述你想在什么时候触发该方法
>
> 方法参数注解说明
>
> - @Self:目标对象本身
> - @Retrun:目标程序方法返回值(Kind.RETURN)
> - @ProbeClassName:目标类名
> - @ProbeMethodName:目标方法名
> - @targetInstance:@Location指定的clazz,method的目标(Kind.CALL)
> - @targetMethodOrField:@Location指定的clazz,method的目标的方法或字段(Kind.CALL)
> - @Duration:目标方法执行时间,单位是纳秒,需要与 Kind.RETURN 或者 Kind.ERROR 一起使用

（3）设置jdk/btrace 环境变量

```
1  export JAVA_HOME=/home/jdk1.8.0
```

```
export JRE_HOME=/home/jdk1.8.0/jre
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
export PATH=$JAVA_HOME/bin:$PATH

export BTRACE_HOME=/home/appuser/bTrace
export PATH=$BTRACE_HOME/bin:$PATH
```

上传监控脚本 UrsInterfaceCalls.java 到服务器；

> 可以用 btracec 进行预编译，以保证代码无误

（4）获取tomcat的执行进程号 ps aux |grep “/fa.163.com”

（5）进入UrsInterfaceCalls.java所在目录，启动btrace监控，监听指定进程号19504（即jvm的进程号）

```
sh btrace -p 2021 19054  UrsInterfaceCalls.java
```

> -p 2021： 指定一个端口号，防止多个执行导致端口冲突；
> 19054： 要监听的进程号
> UrsInterfaceCalls.java： 监听脚本

（6）查看结果
如果方法执行超过500ms，会打印日志，同时打印堆栈；

```

==CookieDecoder requestDecode spend: 525 ms
com.netease.urs.CookieDecoder.requestDecode(CookieDecoder.java:64)
com.netease.urs.ntescode.validate_cookie_online(ntescode.java:49)
com.netease.common.util.CookieUtil.getUserInfoFromUrsRemoteCookie(CookieUtil.java:317)
com.netease.lottery.service.util.CookieUtilServiceImpl.getUserInfoFromUrsRemoteCookie(CookieUtilServiceImpl.jav
com.netease.lottery.service.util.CookieUtilServiceImpl$$FastClassByCGLIB$$1bd66cf1.invoke(<generated>)
org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)

.......

org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.run(NioEndpoint.java:1708)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
java.lang.Thread.run(Thread.java:745)
```

如果要定位具体耗时，需要对各个关键方法，都添加监控脚本。

3.2 使用Greys

使用greys，无需编写 脚步，它是命令交互式的，直接输入命令指定监控的

类、方法。
但是每次只能监控一个方法，不能像 Btrace，可以同时监控多个方法。

使用过程大体步骤：

- 下载解压 Greys工具，安装；
- 设置jdk 环境变量
- 获取tomcat 进程号
- 启动监控
- 查看详情

具体步骤：
（1）下载最新版本的Greys、解压后，执行安装命令

```
sh ./install-local.sh
```

（2）设置环境变量

```
export JAVA_HOME=/home/jdk1.8.0
export JRE_HOME=/home/jdk1.8.0/jre
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
export PATH=$JAVA_HOME/bin:$PATH
```

（3）查询 jvm的进程号，进入greys安装目录，启动Greys

```
./greys.sh 10437
```

> 10437 为 jvm的进程号

（4）启动后，就可以通过交互式命令方式，对指定的类、方法进行分析。

使用help 可以看到各种命令。

（5）使用 trace命令 跟踪指定类、方法的执行时间、参数、返回值情况；
使用 help trace，查询使用方式

例如：跟踪CookieDecoder类中 requestDecode()方法耗时超过500ms 的方法执

行情况：

```
trace  -n 2  com.netease.urs.CookieDecoder  requestDecode    '#cost>500'
```

> -n 2： 代表只打印2次就退出（防止刷屏，影响性能）；
> com.netease.urs.CookieDecoder： 监听的类名
> requestDecode :监听的方法名
> ‘#cost>500’: 打印条件为 耗时超过 500ms

执行后，会显示：

```
ga?>trace  -n 2  com.netease.urs.CookieDecoder  requestDecode    '#cost>10'
Press Ctrl+D to abort.
Affect(class-cnt:1 , method-cnt:2) cost in 262 ms.
```

代表动态修改了一个类，对两个方法（例如方法重载）进行监控，修改花费262毫秒。

如果出现满足条件的情况，则我们会看到打印结果：

```
`---+Tracing for : thread_name="http-nio-8003-exec-8" thread_id=0x7a;is_daemon=true;priority=5;
    `---+[5283,5283ms]com.netease.urs.CookieDecoder:requestDecode()
        +---[1,0ms]java.lang.System:nanoTime()
        +---[2,1ms]org.apache.http.client.methods.HttpPost:<init>(@39)
        +---[2,0ms]java.lang.StringBuffer:<init>(@41)
        +---[2,0ms]java.lang.StringBuffer:append(@42)
        +---[2,0ms]java.net.URLEncoder:encode(@43)
        +---[2,0ms]java.lang.StringBuffer:append(@43)
        +---[2,0ms]java.lang.StringBuffer:append(@44)
        +---[2,0ms]java.lang.StringBuffer:append(@45)
        +---[2,0ms]java.lang.StringBuffer:append(@46)
        +---[2,0ms]java.lang.StringBuffer:append(@47)
        +---[2,0ms]java.lang.StringBuffer:append(@48)
        +---[2,0ms]java.lang.Integer:<init>(@49)
        +---[2,0ms]java.lang.Integer:<init>(@49)
        +---[2,0ms]java.lang.reflect.Method:invoke(@49)
        +---[2,0ms]java.lang.StringBuffer:append(@49)
        +---[2,0ms]java.lang.StringBuffer:toString(@50)
        +---[2,0ms]org.apache.http.entity.StringEntity:<init>(@50)
        +---[2,0ms]org.apache.http.entity.StringEntity:setContentType(@51)
        +---[2,0ms]org.apache.http.client.methods.HttpPost:setEntity(@52)
        +---[2,0ms]org.apache.http.client.methods.HttpPost:getParams(@53)
        +---[2,0ms]org.apache.http.params.HttpParams:setIntParameter(@55)
        +---[2,0ms]org.apache.http.params.HttpParams:setIntParameter(@58)
        +---[5282,5280ms]com.netease.urs.http.CustomHttpComponent:execute(@60)
        +---[5283,0ms]org.apache.http.HttpResponse:getEntity(@61)
        +---[5283,0ms]org.apache.http.util.EntityUtils:toString(@62)
        +---[5283,0ms]com.netease.urs.util.LogUtil:debug(@63)
        +---[5283,0ms]org.apache.http.client.methods.HttpPost:releaseConnection(@71)
        +---[5283,0ms]java.lang.System:nanoTime(@64)
        `---[5283,0ms]com.netease.urs.CookieDecoder:$btrace$com$netease$fa$trace$UrsInterfaceCalls$2$requestDec
```

可以看到，主要耗时在

```
1  +---[5282,5280ms]com.netease.urs.http.CustomHttpComponent:execute(@60)
```

只要一层一层跟踪下去，就可以最终定位问题。

（6）退出前可以使用 reset 恢复增强类（即被动态修改的代码）

（7）最后，使用shutdown 关闭greys 并退出

4.总结说明

（1） 相比两个工具，btrace 需要手写脚步，每次更新都要重新上传再执行，而greys 支持命令式交互，无需手写脚本；

（2）btrace 脚步中，可以写多个监听类和方法，但是greys 命令同时只能输入一个。（但是greys 可以支持多个用户操作，所如果想同时监控多个方法，只能开多窗口）

（3）btrace 要确保监控脚本的正确性，使用前最好预编译，防止动态增强后影响在线功能；

（4）监控时，设置合适的条件，例如在 greys实例中，花费时间大于 N ms 才输出，且只打印2个。

（5）greys 中只能显示1层的方法调用情况，无法直接跟踪到最底层；只能自己一层一层往下跟进。