调试排错 - Java动态调试技术原理

简介

断点调试是我们最常使用的调试手段，它可以获取到方法执行过程中的变量信息，并可以观察到方法的执行路径。但断点调试会在断点位置停顿，使得整个应用停止响应。在线上停顿应用是致命的，动态调试技术给了我们创造新的调试模式的想象空间。本文将研究Java语言中的动态调试技术，首先概括Java动态调试所涉及的技术基础，接着介绍我们在Java动态调试领域的思考及实践，通过结合实际业务场景，设计并实现了一种具备动态性的断点调试工具Java-debug-tool，显著提高了故障排查效率。

JVMTI (JVM Tool Interface)是Java虚拟机对外提供的Native编程接口，通过JVMTI，外部进程可以获取到运行时JVM的诸多信息，比如线程、GC等。Agent是一个运行在目标JVM的特定程序，它的职责是负责从目标JVM中获取数据，然后将数据传递给外部进程。加载Agent的时机可以是目标JVM启动之时，也可以是在目标JVM运行时进行加载，而在目标JVM运行时进行Agent加载具备动态性，对于时机未知的Debug场景来说非常实用。下面将详细分析Java Agent技术的实现细节。

Agent的实现模式

JVMTI是一套Native接口，在Java SE 5之前，要实现一个Agent只能通过编写Native代码来实现。从Java SE 5开始，可以使用Java的Instrumentation接口(java.lang.instrument)来编写Agent。无论是通过Native的方式还是通过Java Instrumentation接口的方式来编写Agent，它们的工作都是借助JVMTI来进行完成，下面介绍通过Java Instrumentation接口编写Agent的方法。

通过*Java Instrumentation API*

- 实现Agent启动方法

Java Agent支持目标JVM启动时加载，也支持在目标JVM运行时加载，这两种不同的加载模式会使用不同的入口函数，如果需要在目标JVM启动的同时加载Agent，那么可以选择实现下面的方法：

```
[1] public static void premain(String agentArgs, Instrumentation inst);
[2] public static void premain(String agentArgs);
```

JVM将首先寻找[1]，如果没有发现[1]，再寻找[2]。如果希望在目标JVM运行时加载Agent，则需要实现下面的方法：

```
[1] public static void agentmain(String agentArgs, Instrumentation inst);
[2] public static void agentmain(String agentArgs);
```

这两组方法的第一个参数AgentArgs是随同 “– javaagent”一起传入的程序参数，如果这个字符串代表了多个参数，就需要自己解析这些参数。inst是Instrumentation类型的对象，是JVM自动传入的，我们可以拿这个参数进行类增强等操作。

- 指定Main-Class

Agent需要打包成一个jar包，在ManiFest属性中指定“Premain-Class”或者“Agent-Class”：

```
Premain-Class: class
Agent-Class: class
```

- 挂载到目标JVM

将编写的Agent打成jar包后，就可以挂载到目标JVM上去了。如果选择在目标JVM启动时加载Agent，则可以使用 “-javaagent:[=]“，具体的使用方法可以使用“Java -Help”来查看。如果想要在运行时挂载Agent到目标JVM，就需要做一些额外的开发了。

com.sun.tools.attach.VirtualMachine 这个类代表一个JVM抽象，可以通过这个类找到目标JVM，并且将Agent挂载到目标JVM上。下面是使用com.sun.tools.attach.VirtualMachine进行动态挂载Agent的一般实现：

```
    private void attachAgentToTargetJVM() throws Exception {
        List<VirtualMachineDescriptor> virtualMachineDescriptors = VirtualMachine.list();
        VirtualMachineDescriptor targetVM = null;
        for (VirtualMachineDescriptor descriptor : virtualMachineDescriptors) {
            if (descriptor.id().equals(configure.getPid())) {
                targetVM = descriptor;
                break;
            }
        }
        if (targetVM == null) {
            throw new IllegalArgumentException("could not find the target jvm by process id:" +
configure.getPid());
        }
        VirtualMachine virtualMachine = null;
        try {
            virtualMachine = VirtualMachine.attach(targetVM);
            virtualMachine.loadAgent("{agent}", "{params}");
        } catch (Exception e) {
            if (virtualMachine != null) {
                virtualMachine.detach();
            }
        }
    }
```

首先通过指定的进程ID找到目标JVM，然后通过Attach挂载到目标JVM上，执行加载Agent操作。VirtualMachine的Attach方法就是用来将Agent挂载到目标JVM上去的，而Detach则是将Agent从目标JVM卸载。关于Agent是如何挂载到目标JVM上的具体技术细节，将在下文中进行分析。

启动时加载Agent

参数解析

创建JVM时，JVM会进行参数解析，即解析那些用来配置JVM启动的参数，比如堆大小、GC等；本文主要关注解析的参数为-agentlib、 -agentpath、 -javaagent，这几个参数用来指定Agent，JVM会根据这几个参数加载Agent。下面来分析一下JVM是如何解析这几个参数的。

```
  // -agentlib and -agentpath
  if (match_option(option, "-agentlib:", &tail) ||
          (is_absolute_path = match_option(option, "-agentpath:", &tail))) {
```

```
      if(tail != NULL) {
        const char* pos = strchr(tail, '=');
        size_t len = (pos == NULL) ? strlen(tail) : pos - tail;
        char* name = strncpy(NEW_C_HEAP_ARRAY(char, len + 1, mtArguments), tail, len);
        name[len] = '\0';
        char *options = NULL;
        if(pos != NULL) {
          options = os::strdup_check_oom(pos + 1, mtArguments);
        }
#if !INCLUDE_JVMTI
        if (valid_jdwp_agent(name, is_absolute_path)) {
          jio_fprintf(defaultStream::error_stream(),
            "Debugging agents are not supported in this VM\n");
          return JNI_ERR;
        }
#endif // !INCLUDE_JVMTI
        add_init_agent(name, options, is_absolute_path);
      }
    // -javaagent
    } else if (match_option(option, "-javaagent:", &tail)) {
#if !INCLUDE_JVMTI
      jio_fprintf(defaultStream::error_stream(),
        "Instrumentation agents are not supported in this VM\n");
      return JNI_ERR;
#else
      if (tail != NULL) {
        size_t length = strlen(tail) + 1;
        char *options = NEW_C_HEAP_ARRAY(char, length, mtArguments);
        jio_snprintf(options, length, "%s", tail);
        add_init_agent("instrument", options, false);
        // java agents need module java.instrument
        if (!create_numbered_property("jdk.module.addmods", "java.instrument", addmods_count++))
{
          return JNI_ENOMEM;
        }
      }
#endif // !INCLUDE_JVMTI
    }
```

上面的代码片段截取自hotspot/src/share/vm/runtime/arguments.cpp中的 Arguments::parse_each_vm_init_arg(const JavaVMInitArgs* args, bool* patch_mod_javabase, Flag::Flags origin) 函数，该函数用来解析一个具体的JVM参数。这段代码的主要功能是解析出需要加载的Agent路径，然后调用add_init_agent函数进行解析结果的存储。下面先看一下add_init_agent函数的具体实现：

```
  // -agentlib and -agentpath arguments
  static AgentLibraryList _agentList;
  static void add_init_agent(const char* name, char* options, bool absolute_path)
    { _agentList.add(new AgentLibrary(name, options, absolute_path, NULL)); }
```

AgentLibraryList是一个简单的链表结构，add_init_agent函数将解析好的、需要加载的Agent添加到这个链表中，等待后续的处理。

这里需要注意，解析-javaagent参数有一些特别之处，这个参数用来指定一个我们通过Java Instrumentation API来编写的Agent，Java Instrumentation API底层依赖的是JVMTI，对-JavaAgent的处理也说明了这一点，在调用add_init_agent函数时第一个参数是“instrument”，关于加载Agent这个问题在下一小节进行展开。到此，我们知道在启动JVM时指定的Agent已经被JVM解析完存放在了一个链表结构中。下面来分析一下JVM是如何加载这些Agent的。

执行加载操作

在创建JVM进程的函数中，解析完JVM参数之后，下面的这段代码和加载Agent相关：

```
  // Launch -agentlib/-agentpath and converted -Xrun agents
  if (Arguments::init_agents_at_startup()) {
    create_vm_init_agents();
  }
  static bool init_agents_at_startup() {
    return !_agentList.is_empty();
  }
```

当JVM判断出上一小节中解析出来的Agent不为空的时候，就要去调用函数create_vm_init_agents来加载Agent，下面来分析一下create_vm_init_agents函数是如何加载Agent的。

```
void Threads::create_vm_init_agents() {
  AgentLibrary* agent;
  for (agent = Arguments::agents(); agent != NULL; agent = agent->next()) {
    OnLoadEntry_t  on_load_entry = lookup_agent_on_load(agent);
    if (on_load_entry != NULL) {
      // Invoke the Agent_OnLoad function
      jint err = (*on_load_entry)(&main_vm, agent->options(), NULL);
    }
  }
}
```

create_vm_init_agents这个函数通过遍历Agent链表来逐个加载Agent。通过这段代码可以看出，首先通过lookup_agent_on_load来加载Agent并且找到Agent_OnLoad函数，这个函数是Agent的入口函数。如果没找到这个函数，则认为是加载了一个不合法的Agent，则什么也不做，否则调用这个函数，这样Agent的代码就开始执行起来了。对于使用Java Instrumentation API来编写Agent的方式来说，在解析阶段观察到在add_init_agent函数里面传递进去的是一个叫做”instrument”的字符串，其实这是一个动态链接库。在Linux里面，这个库叫做libinstrument.so，在BSD系统中叫做libinstrument.dylib，该动态链接库在{JAVA_HOME}/jre/lib/目录下。

*instrument*动态链接库

libinstrument用来支持使用Java Instrumentation API来编写Agent，在libinstrument中有一个非常重要的类称为：JPLISAgent(Java Programming Language Instrumentation Services Agent)，它的作用是初始化所有通过Java Instrumentation API编写的Agent，并且也承担着通过JVMTI实现Java Instrumentation中暴露API的责任。

我们已经知道，在JVM启动的时候，JVM会通过-javaagent参数加载Agent。最开始加载的是libinstrument动态链接库，然后在动态链接库里面找到JVMTI的入口方法：Agent_OnLoad。下面就来分析一下在libinstrument动态链接库中，Agent_OnLoad函数是怎么实现的。

```
JNIEXPORT jint JNICALL
DEF_Agent_OnLoad(JavaVM *vm, char *tail, void * reserved) {
    initerror = createNewJPLISAgent(vm, &agent);
    if ( initerror == JPLIS_INIT_ERROR_NONE ) {
        if (parseArgumentTail(tail, &jarfile, &options) != 0) {
            fprintf(stderr, "-javaagent: memory allocation failure.\n");
            return JNI_ERR;
        }
        attributes = readAttributes(jarfile);
        premainClass = getAttribute(attributes, "Premain-Class");
        /* Save the jarfile name */
```

```
        agent->mJarfile = jarfile;
        /*
         * Convert JAR attributes into agent capabilities
         */
        convertCapabilityAttributes(attributes, agent);
        /*
         * Track (record) the agent class name and options data
         */
        initerror = recordCommandLineData(agent, premainClass, options);
    }
    return result;
}
```

上述代码片段是经过精简的libinstrument中Agent_OnLoad实现的，大概的流程就是：先创建一个JPLISAgent，然后将ManiFest中设定的一些参数解析出来， 比如(Premain-Class)等。创建了JPLISAgent之后，调用initializeJPLISAgent对这个Agent进行初始化操作。跟进initializeJPLISAgent看一下是如何初始化的：

```
JPLISInitializationError initializeJPLISAgent(JPLISAgent *agent, JavaVM *vm, jvmtiEnv *jvmtienv)
{
    /* check what capabilities are available */
    checkCapabilities(agent);
    /* check phase - if live phase then we don't need the VMInit event */
    jvmtierror = (*jvmtienv)->GetPhase(jvmtienv, &phase);
    /* now turn on the VMInit event */
    if ( jvmtierror == JVMTI_ERROR_NONE ) {
        jvmtiEventCallbacks callbacks;
        memset(&callbacks, 0, sizeof(callbacks));
        callbacks.VMInit = &eventHandlerVMInit;
        jvmtierror = (*jvmtienv)->SetEventCallbacks(jvmtienv,&callbacks,sizeof(callbacks));
    }
    if ( jvmtierror == JVMTI_ERROR_NONE ) {
        jvmtierror = (*jvmtienv)-
>SetEventNotificationMode(jvmtienv,JVMTI_ENABLE,JVMTI_EVENT_VM_INIT,NULL);
    }
    return (jvmtierror == JVMTI_ERROR_NONE)? JPLIS_INIT_ERROR_NONE : JPLIS_INIT_ERROR_FAILURE;
}
```

这里，我们关注callbacks.VMInit = &eventHandlerVMInit;这行代码，这里设置了一个VMInit事件的回调函数，表示在JVM初始化的时候会回调eventHandlerVMInit函数。下面来看一下这个函数的实现细节，猜测就是在这里调用了Premain方法：

```
void JNICALL  eventHandlerVMInit( jvmtiEnv *jvmtienv,JNIEnv *jnienv,jthread thread) {
   // ...
   success = processJavaStart( environment->mAgent, jnienv);
  // ...
}
jboolean  processJavaStart(JPLISAgent *agent,JNIEnv *jnienv) {
    result = createInstrumentationImpl(jnienv, agent);
    /*
     *  Load the Java agent, and call the premain.
     */
    if ( result ) {
        result = startJavaAgent(agent, jnienv, agent->mAgentClassName, agent->mOptionsString,
agent->mPremainCaller);
    }
    return result;
}
jboolean startJavaAgent( JPLISAgent *agent,JNIEnv *jnienv,const char *classname,const char
*optionsString,jmethodID agentMainMethod) {
```

```
  // ...
  invokeJavaAgentMainMethod(jnienv,agent->mInstrumentationImpl,agentMainMethod,
classNameObject,optionsStringObject);
  // ...
}
```

看到这里，Instrument已经实例化，invokeJavaAgentMainMethod这个方法将我们的premain方法执行起来了。接着，我们就可以根据Instrument实例来做我们想要做的事情了。

运行时加载Agent

比起JVM启动时加载Agent，运行时加载Agent就比较有诱惑力了，因为运行时加载Agent的能力给我们提供了很强的动态性，我们可以在需要的时候加载Agent来进行一些工作。因为是动态的，我们可以按照需求来加载所需要的Agent，下面来分析一下动态加载Agent的相关技术细节。

AttachListener

Attach机制通过Attach Listener线程来进行相关事务的处理，下面来看一下Attach Listener线程是如何初始化的。

```
// Starts the Attach Listener thread
void AttachListener::init() {
  // 创建线程相关部分代码被去掉了
  const char thread_name[] = "Attach Listener";
  Handle string = java_lang_String::create_from_str(thread_name, THREAD);
  { MutexLocker mu(Threads_lock);
    JavaThread* listener_thread = new JavaThread(&attach_listener_thread_entry);
    // ...
  }
}
```

我们知道，一个线程启动之后都需要指定一个入口来执行代码，Attach Listener线程的入口是attach_listener_thread_entry，下面看一下这个函数的具体实现：

```
static void attach_listener_thread_entry(JavaThread* thread, TRAPS) {
  AttachListener::set_initialized();
  for (;;) {
      AttachOperation* op = AttachListener::dequeue();
      // find the function to dispatch too
      AttachOperationFunctionInfo* info = NULL;
      for (int i=0; funcs[i].name != NULL; i++) {
        const char* name = funcs[i].name;
        if (strcmp(op->name(), name) == 0) {
          info = &(funcs[i]); break;
        }}
       // dispatch to the function that implements this operation
        res = (info->func)(op, &st);
      //...
    }
}
```

整个函数执行逻辑，大概是这样的：

- 拉取一个需要执行的任务：AttachListener::dequeue。
- 查询匹配的命令处理函数。
- 执行匹配到的命令执行函数。

其中第二步里面存在一个命令函数表，整个表如下：

```
static AttachOperationFunctionInfo funcs[] = {
  { "agentProperties",  get_agent_properties },
  { "datadump",         data_dump },
  { "dumpheap",         dump_heap },
  { "load",             load_agent },
  { "properties",       get_system_properties },
  { "threaddump",       thread_dump },
  { "inspectheap",      heap_inspection },
  { "setflag",          set_flag },
  { "printflag",        print_flag },
  { "jcmd",             jcmd },
  { NULL,               NULL }
};
```

对于加载Agent来说，命令就是“load”。现在，我们知道了Attach Listener大概的工作模式，但是还是不太清楚任务从哪来，这个秘密就藏在AttachListener::dequeue这行代码里面，接下来我们来分析一下dequeue这个函数：

```
LinuxAttachOperation* LinuxAttachListener::dequeue() {
  for (;;) {
    // wait for client to connect
    struct sockaddr addr;
    socklen_t len = sizeof(addr);
    RESTARTABLE(::accept(listener(), &addr, &len), s);
    // get the credentials of the peer and check the effective uid/guid
    // - check with jeff on this.
    struct ucred cred_info;
    socklen_t optlen = sizeof(cred_info);
    if (::getsockopt(s, SOL_SOCKET, SO_PEERCRED, (void*)&cred_info, &optlen) == -1) {
      ::close(s);
      continue;
    }
    // peer credential look okay so we read the request
    LinuxAttachOperation* op = read_request(s);
    return op;
  }
}
```

这是Linux上的实现，不同的操作系统实现方式不太一样。上面的代码表面，Attach Listener在某个端口监听着，通过accept来接收一个连接，然后从这个连接里面将请求读取出来，然后将请求包装成一个AttachOperation类型的对象，之后就会从表里查询对应的处理函数，然后进行处理。

Attach Listener使用一种被称为“懒加载”的策略进行初始化，也就是说，JVM启动的时候Attach Listener并不一定会启动起来。下面我们来分析一下这种“懒加载”策略的具体实现方案。

```
  // Start Attach Listener if +StartAttachListener or it can't be started lazily
  if (!DisableAttachMechanism) {
    AttachListener::vm_start();
    if (StartAttachListener || AttachListener::init_at_startup()) {
      AttachListener::init();
    }
  }
// Attach Listener is started lazily except in the case when
// +ReduseSignalUsage is used
```

```
bool AttachListener::init_at_startup() {
  if (ReduceSignalUsage) {
    return true;
  } else {
    return false;
  }
}
```

上面的代码截取自create_vm函数，DisableAttachMechanism、StartAttachListener和ReduceSignalUsage这三个变量默认都是false，所以AttachListener::init();这行代码不会在create_vm的时候执行，而vm_start会执行。下面来看一下这个函数的实现细节：

```
void AttachListener::vm_start() {
  char fn[UNIX_PATH_MAX];
  struct stat64 st;
  int ret;
  int n = snprintf(fn, UNIX_PATH_MAX, "%s/.java_pid%d",
           os::get_temp_directory(), os::current_process_id());
  assert(n < (int)UNIX_PATH_MAX, "java_pid file name buffer overflow");
  RESTARTABLE(::stat64(fn, &st), ret);
  if (ret == 0) {
    ret = ::unlink(fn);
    if (ret == -1) {
      log_debug(attach)("Failed to remove stale attach pid file at %s", fn);
    }
  }
}
```

这是在Linux上的实现，是将/tmp/目录下的.java_pid{pid}文件删除，后面在创建Attach Listener线程的时候会创建出来这个文件。上面说到，AttachListener::init()这行代码不会在create_vm的时候执行，这行代码的实现已经在上文中分析了，就是创建Attach Listener线程，并监听其他JVM的命令请求。现在来分析一下这行代码是什么时候被调用的，也就是“懒加载”到底是怎么加载起来的。

```
  // Signal Dispatcher needs to be started before VMInit event is posted
  os::signal_init();
```

这是create_vm中的一段代码，看起来跟信号相关，其实Attach机制就是使用信号来实现“懒加载“的。下面我们来仔细地分析一下这个过程

```
void os::signal_init() {
  if (!ReduceSignalUsage) {
    // Setup JavaThread for processing signals
    EXCEPTION_MARK;
    Klass* k = SystemDictionary::resolve_or_fail(vmSymbols::java_lang_Thread(), true, CHECK);
    instanceKlassHandle klass (THREAD, k);
    instanceHandle thread_oop = klass->allocate_instance_handle(CHECK);
    const char thread_name[] = "Signal Dispatcher";
    Handle string = java_lang_String::create_from_str(thread_name, CHECK);
    // Initialize thread_oop to put it into the system threadGroup
    Handle thread_group (THREAD, Universe::system_thread_group());
    JavaValue result(T_VOID);
    JavaCalls::call_special(&result,
thread_oop,klass,vmSymbols::object_initializer_name(),vmSymbols::threadgroup_string_void_signatu
re(),
                            thread_group,string,CHECK);
    KlassHandle group(THREAD, SystemDictionary::ThreadGroup_klass());
```

```
 JavaCalls::call_special(&result,thread_group,group,vmSymbols::add_method_name(),vmSymbols::thre
ad_void_signature(),thread_oop,CHECK);
    os::signal_init_pd();
    { MutexLocker mu(Threads_lock);
      JavaThread* signal_thread = new JavaThread(&signal_thread_entry);
     // ...
    }
    // Handle ^BREAK
    os::signal(SIGBREAK, os::user_handler());
  }
}
```

JVM创建了一个新的进程来实现信号处理，这个线程叫“Signal Dispatcher”，一个线程创建之后需要有一个入口，“Signal Dispatcher”的入口是signal_thread_entry：

```
263
264         switch (sig) {
265           case SIGBREAK: {
266             // Check if the signal is a trigger to start the Attach Listener - in that
267             // case don't print stack traces.
268             if (!DisableAttachMechanism && AttachListener::is_init_trigger()) {
269               continue;
270             }
```

这段代码截取自signal_thread_entry函数，截取中的内容是和Attach机制信号处理相关的代码。这段代码的意思是，当接收到“SIGBREAK”信号，就执行接下来的代码，这个信号是需要Attach到JVM上的信号发出来，这个后面会再分析。我们先来看一句关键的代码：AttachListener::is_init_trigger()：

```
bool AttachListener::is_init_trigger() {
  if (init_at_startup() || is_initialized()) {
    return false;               // initialized at startup or already initialized
  }
  char fn[PATH_MAX+1];
  sprintf(fn, ".attach_pid%d", os::current_process_id());
  int ret;
  struct stat64 st;
  RESTARTABLE(::stat64(fn, &st), ret);
  if (ret == -1) {
    log_trace(attach)("Failed to find attach file: %s, trying alternate", fn);
    snprintf(fn, sizeof(fn), "%s/.attach_pid%d", os::get_temp_directory(),
os::current_process_id());
    RESTARTABLE(::stat64(fn, &st), ret);
  }
  if (ret == 0) {
    // simple check to avoid starting the attach mechanism when
    // a bogus user creates the file
    if (st.st_uid == geteuid()) {
      init();
      return true;
    }
  }
  return false;
}
```

首先检查了一下是否在JVM启动时启动了Attach Listener，或者是否已经启动过。如果没有，才继续执行，在/tmp目录下创建一个叫做.attach_pid%d的文件，然后执行AttachListener的init函数，这个函数就是用来创建Attach Listener线程的函数，上面已经提到多次并进行了分析。到此，我们知道Attach机制的奥秘所在，也就是Attach Listener线程的创建依靠Signal Dispatcher线程，Signal Dispatcher是用来处理信号的线程，当Signal Dispatcher线程接收到“SIGBREAK”信号之后，就会执行初始化Attach Listener的工作。

运行时加载*Agent*的实现

我们继续分析，到底是如何将一个Agent挂载到运行着的目标JVM上，在上文中提到了一段代码，用来进行运行时挂载Agent，可以参考上文中展示的关于“attachAgentToTargetJvm”方法的代码。这个方法里面的关键是调用VirtualMachine的attach方法进行Agent挂载的功能。下面我们就来分析一下VirtualMachine的attach方法具体是怎么实现的。

```
public static VirtualMachine attach(String var0) throws AttachNotSupportedException, IOException
{
    if (var0 == null) {
        throw new NullPointerException("id cannot be null");
    } else {
        List var1 = AttachProvider.providers();
        if (var1.size() == 0) {
            throw new AttachNotSupportedException("no providers installed");
        } else {
            AttachNotSupportedException var2 = null;
            Iterator var3 = var1.iterator();
            while(var3.hasNext()) {
                AttachProvider var4 = (AttachProvider)var3.next();
                try {
                    return var4.attachVirtualMachine(var0);
                } catch (AttachNotSupportedException var6) {
                    var2 = var6;
                }
            }
            throw var2;
        }
    }
}
```

这个方法通过attachVirtualMachine方法进行attach操作，在MacOS系统中，AttachProvider的实现类是BsdAttachProvider。我们来看一下BsdAttachProvider的attachVirtualMachine方法是如何实现的：

```
public VirtualMachine attachVirtualMachine(String var1) throws AttachNotSupportedException,
IOException {
    this.checkAttachPermission();
    this.testAttachable(var1);
    return new BsdVirtualMachine(this, var1);
}
BsdVirtualMachine(AttachProvider var1, String var2) throws AttachNotSupportedException,
IOException {
    int var3 = Integer.parseInt(var2);
    this.path = this.findSocketFile(var3);
    if (this.path == null) {
        File var4 = new File(tmpdir, ".attach_pid" + var3);
        createAttachFile(var4.getPath());
        try {
            sendQuitTo(var3);
            int var5 = 0;
            long var6 = 200L;
            int var8 = (int)(this.attachTimeout() / var6);
            do {
                try {
                    Thread.sleep(var6);
                } catch (InterruptedException var21) {
                    ;
                }
                this.path = this.findSocketFile(var3);
```

```
                ++var5;
            } while(var5 <= var8 && this.path == null);
        } finally {
            var4.delete();
        }
    }
    int var24 = socket();
    connect(var24, this.path);
}
private String findSocketFile(int var1) {
    String var2 = ".java_pid" + var1;
    File var3 = new File(tmpdir, var2);
    return var3.exists() ? var3.getPath() : null;
}
```

findSocketFile方法用来查询目标JVM上是否已经启动了Attach Listener，它通过检查”tmp/“目录下是否存在java_pid{pid}来进行实现。如果已经存在了，则说明Attach机制已经准备就绪，可以接受客户端的命令了，这个时候客户端就可以通过connect连接到目标JVM进行命令的发送，比如可以发送“load”命令来加载Agent。如果java_pid{pid}文件还不存在，则需要通过sendQuitTo方法向目标JVM发送一个“SIGBREAK”信号，让它初始化Attach Listener线程并准备接受客户端连接。可以看到，发送了信号之后客户端会循环等待java_pid{pid}这个文件，之后再通过connect连接到目标JVM上。

*load*命令的实现

下面来分析一下，“load”命令在JVM层面的实现：

```
static jint load_agent(AttachOperation* op, outputStream* out) {
  // get agent name and options
  const char* agent = op->arg(0);
  const char* absParam = op->arg(1);
  const char* options = op->arg(2);
  // If loading a java agent then need to ensure that the java.instrument module is loaded
  if (strcmp(agent, "instrument") == 0) {
    Thread* THREAD = Thread::current();
    ResourceMark rm(THREAD);
    HandleMark hm(THREAD);
    JavaValue result(T_OBJECT);
    Handle h_module_name = java_lang_String::create_from_str("java.instrument", THREAD);

 JavaCalls::call_static(&result,SystemDictionary::module_Modules_klass(),vmSymbols::loadModule_n
ame(),
                           vmSymbols::loadModule_signature(),h_module_name,THREAD);
  }
  return JvmtiExport::load_agent_library(agent, absParam, options, out);
}
```

这个函数先确保加载了java.instrument模块，之后真正执行Agent加载的函数是 load_agent_library ,这个函数的套路就是加载Agent动态链接库，如果是通过Java instrument API实现的Agent，则加载的是libinstrument动态链接库，然后通过libinstrument里面的代码实现运行agentmain方法的逻辑，这一部分内容和libinstrument实现premain方法运行的逻辑其实差不多，这里不再做分析。至此，我们对Java Agent技术已经有了一个全面而细致的了解。

动态字节码修改的限制

上文中已经详细分析了Agent技术的实现，我们使用Java Instrumentation API来完成动态类修改的功能，在Instrumentation接口中，通过addTransformer方法来增加一个类转换器，类转换器由类ClassFileTransformer接口实现。ClassFileTransformer接口中唯一的方法transform用于实现类转换，当类被加载的时候，就会调用transform方法，进行类转换。在运行时，我们可以通过Instrumentation的redefineClasses方法进行类重定义，在方法上有一段注释需要特别注意：

```
     * The redefinition may change method bodies, the constant pool and attributes.
     * The redefinition must not add, remove or rename fields or methods, change the
     * signatures of methods, or change inheritance.  These restrictions maybe be
     * lifted in future versions.  The class file bytes are not checked, verified and installed
     * until after the transformations have been applied, if the resultant bytes are in
     * error this method will throw an exception.
```

这里面提到，我们不可以增加、删除或者重命名字段和方法，改变方法的签名或者类的继承关系。认识到这一点很重要，当我们通过ASM获取到增强的字节码之后，如果增强后的字节码没有遵守这些规则，那么调用redefineClasses方法来进行类的重定义就会失败。那redefineClasses方法具体是怎么实现类的重定义的呢？它对运行时的JVM会造成什么样的影响呢？下面来分析redefineClasses的实现细节。

重定义类字节码的实现细节

上文中我们提到，libinstrument动态链接库中，JPLISAgent不仅实现了Agent入口代码执行的路由，而且还是Java代码与JVMTI之间的一道桥梁。我们在Java代码中调用Java Instrumentation API的redefineClasses，其实会调用libinstrument中的相关代码，我们来分析一下这条路径。

```
public void redefineClasses(ClassDefinition... var1) throws ClassNotFoundException {
    if (!this.isRedefineClassesSupported()) {
        throw new UnsupportedOperationException("redefineClasses is not supported in this
environment");
    } else if (var1 == null) {
        throw new NullPointerException("null passed as 'definitions' in redefineClasses");
    } else {
        for(int var2 = 0; var2 < var1.length; ++var2) {
            if (var1[var2] == null) {
                throw new NullPointerException("element of 'definitions' is null in
redefineClasses");
            }
        }
        if (var1.length != 0) {
            this.redefineClasses0(this.mNativeAgent, var1);
        }
    }
}
private native void redefineClasses0(long var1, ClassDefinition[] var3) throws
ClassNotFoundException;
```

这是InstrumentationImpl中的redefineClasses实现，该方法的具体实现依赖一个Native方法redefineClasses()，我们可以在libinstrument中找到这个Native方法的实现：

```
JNIEXPORT void JNICALL Java_sun_instrument_InstrumentationImpl_redefineClasses0
  (JNIEnv * jnienv, jobject implThis, jlong agent, jobjectArray classDefinitions) {
    redefineClasses(jnienv, (JPLISAgent*)(intptr_t)agent, classDefinitions);
}
```

redefineClasses这个函数的实现比较复杂，代码很长。下面是一段关键的代码片段：

```
    if (!errorOccurred) {
        jvmtiError  errorCode = JVMTI_ERROR_NONE;
        errorCode = (*jvmtienv)->RedefineClasses(jvmtienv, numDefs, classDefs);
        if (errorCode == JVMTI_ERROR_WRONG_PHASE) {
            /* insulate caller from the wrong phase error */
            errorCode = JVMTI_ERROR_NONE;
        } else {
            errorOccurred = (errorCode != JVMTI_ERROR_NONE);
            if ( errorOccurred ) {
                createAndThrowThrowableFromJVMTIErrorCode(jnienv, errorCode);
            }
        }
    }
```

可以看到，其实是调用了JVMTI的RetransformClasses函数来完成类的重定义细节。

```
// class_count - pre-checked to be greater than or equal to 0
// class_definitions - pre-checked for NULL
jvmtiError JvmtiEnv::RedefineClasses(jint class_count, const jvmtiClassDefinition*
class_definitions) {
//TODO: add locking
  VM_RedefineClasses op(class_count, class_definitions, jvmti_class_load_kind_redefine);
  VMThread::execute(&op);
  return (op.check_error());
} /* end RedefineClasses */
```

重定义类的请求会被JVM包装成一个VM_RedefineClasses类型的VM_Operation，VM_Operation是JVM内部的一些操作的基类，包括GC操作等。VM_Operation由VMThread来执行，新的VM_Operation操作会被添加到VMThread的运行队列中去，VMThread会不断从队列里面拉取VM_Operation并调用其doit等函数执行具体的操作。VM_RedefineClasses函数的流程较为复杂，下面是VM_RedefineClasses的大致流程：

- 加载新的字节码，合并常量池，并且对新的字节码进行校验工作

```
  // Load the caller's new class definition(s) into _scratch_classes.
  // Constant pool merging work is done here as needed. Also calls
  // compare_and_normalize_class_versions() to verify the class
  // definition(s).
  jvmtiError load_new_class_versions(TRAPS);
```

- 清除方法上的断点

```
  // Remove all breakpoints in methods of this class
  JvmtiBreakpoints& jvmti_breakpoints = JvmtiCurrentBreakpoints::get_jvmti_breakpoints();
  jvmti_breakpoints.clearall_in_class_at_safepoint(the_class());
```

- JIT逆优化

```
  // Deoptimize all compiled code that depends on this class
  flush_dependent_code(the_class, THREAD);
```

- 进行字节码替换工作，需要进行更新类itable/vtable等操作
- 进行类重定义通知

```
  SystemDictionary::notice_modification();
```

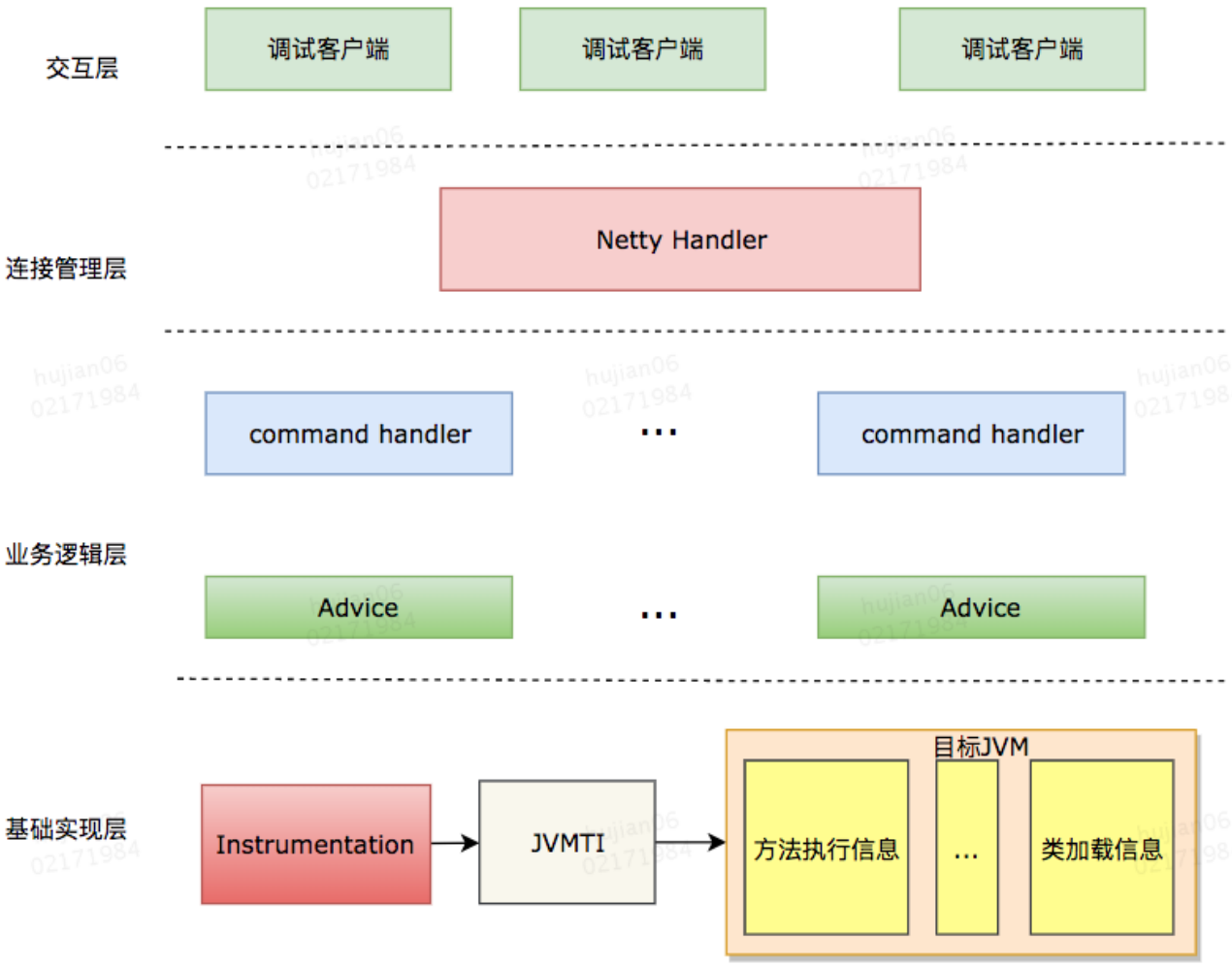
VM_RedefineClasses实现比较复杂的，详细实现可以参考 RedefineClasses (opens new window)的实现。

Java-debug-tool

Java-debug-tool是一个使用Java Instrument API来实现的动态调试工具，它通过在目标JVM上启动一个TcpServer来和调试客户端通信。调试客户端通过命令行来发送调试命令给TcpServer，TcpServer中有专门用来处理命令的handler，handler处理完命令之后会将结果发送回客户端，客户端通过处理将调试结果展示出来。下面将详细介绍Java-debug-tool的整体设计和实现。

*Java-debug-tool*整体架构

Java-debug-tool包括一个Java Agent和一个用于处理调试命令的核心API，核心API通过一个自定义的类加载器加载进来，以保证目标JVM的类不会被污染。整体上Java-debug-tool的设计是一个Client-Server的架构，命令客户端需要完整的完成一个命令之后才能继续执行下一个调试命令。Java-debug-tool支持多人同时进行调试，下面是整体架构图：



下面对每一层做简单介绍：

- 交互层：负责将程序员的输入转换成调试交互协议，并且将调试信息呈现出来。
- 连接管理层：负责管理客户端连接，从连接中读调试协议数据并解码，对调试结果编码并将其写到连接中去；同时将那些超时未活动的连接关闭。
- 业务逻辑层：实现调试命令处理，包括命令分发、数据收集、数据处理等过程。
- 基础实现层：Java-debug-tool实现的底层依赖，通过Java Instrumentation提供的API进行类查找、类重定义等能力，Java Instrumentation底层依赖JVMTI来完成具体的功能。

在Agent被挂载到目标JVM上之后，Java-debug-tool会安排一个Spy在目标JVM内活动，这个Spy负责将目标JVM内部的相关调试数据转移到命令处理模块，命令处理模块会处理这些数据，然后给客户端返回调试结果。命令处理模块会增强目标类的字节码来达到数据获取的目的，多个客户端可以共享一份增强过的字节码，无需重复增强。下面从Java-debug-tool的字节码增强方案、命令设计与实现等角度详细说明。

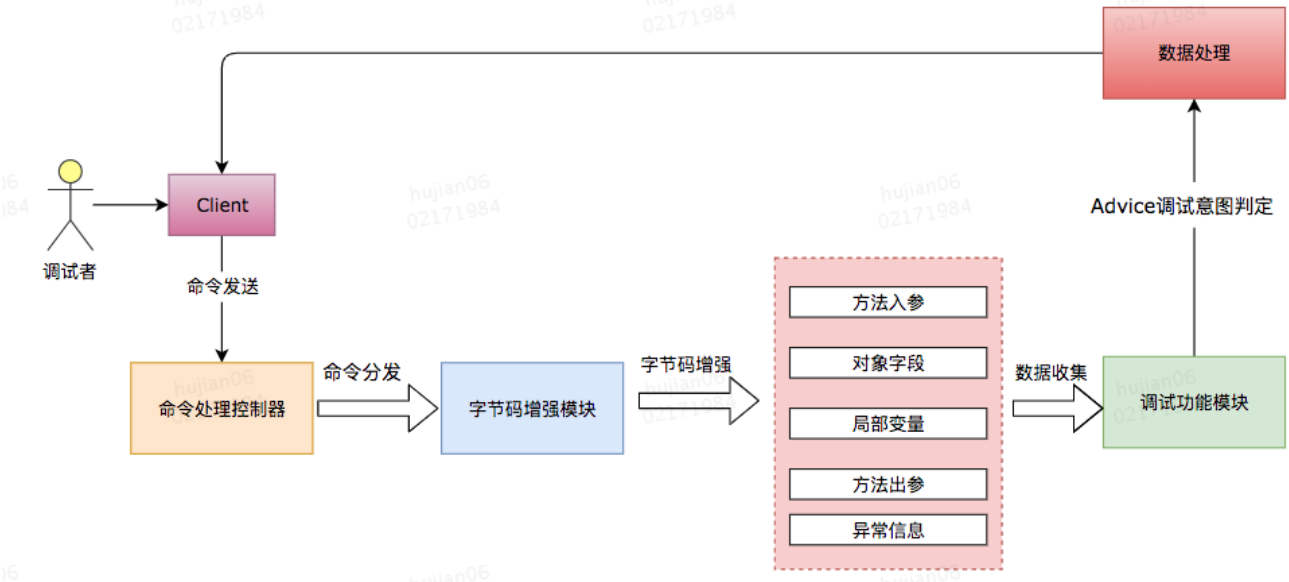
*Java-debug-tool*的字节码增强方案

Java-debug-tool使用字节码增强来获取到方法运行时的信息，比如方法入参、出参等，可以在不同的字节码位置进行增强，这种行为可以称为“插桩”，每个“桩”用于获取数据并将他转储出去。Java-debug-tool具备强大的插桩能力，不同的桩负责获取不同类别的数据，下面是Java-debug-tool目前所支持的“桩”：

- 方法进入点：用于获取方法入参信息。
- Fields获取点1：在方法执行前获取到对象的字段信息。
- 变量存储点：获取局部变量信息。
- Fields获取点2：在方法退出前获取到对象的字段信息。
- 方法退出点：用于获取方法返回值。
- 抛出异常点：用于获取方法抛出的异常信息。
- 通过上面这些代码桩，Java-debug-tool可以收集到丰富的方法执行信息，经过处理可以返回更加可视化的调试结果。

字节码增强

Java-debug-tool在实现上使用了ASM工具来进行字节码增强，并且每个插桩点都可以进行配置，如果不想要什么信息，则没必要进行对应的插桩操作。这种可配置的设计是非常有必要的，因为有时候我们仅仅是想要知道方法的入参和出参，但Java-debug-tool却给我们返回了所有的调试信息，这样我们就得在众多的输出中找到我们所关注的内容。如果可以进行配置，则除了入参点和出参点外其他的桩都不插，那么就可以快速看到我们想要的调试数据，这种设计的本质是为了让调试者更加专注。下面是Java-debug-tool的字节码增强工作方式：



如图所示，当调试者发出调试命令之后，Java-debug-tool会识别命令并判断是否需要进行字节码增强，如果命令需要增强字节码，则判断当前类+当前方法是否已经被增强过。上文已经提到，字节码替换是有一定损耗的，这种具有损耗的操作发生的次数越少越好，所以字节码替换操作会被记录起来，后续命令直接使用即可，不需要重复进行字节码增强，字节码增强还涉及多个调试客户端的协同工作问题，当一个客户端增强了一个类的字节码之后，这个客户端就锁定了该字节码，其他客户端变成只读，无法对该类进行字节码增强，只有当持有锁的客户端主动释放锁或者断开连接之后，其他客户端才能继续增强该类的字节码。

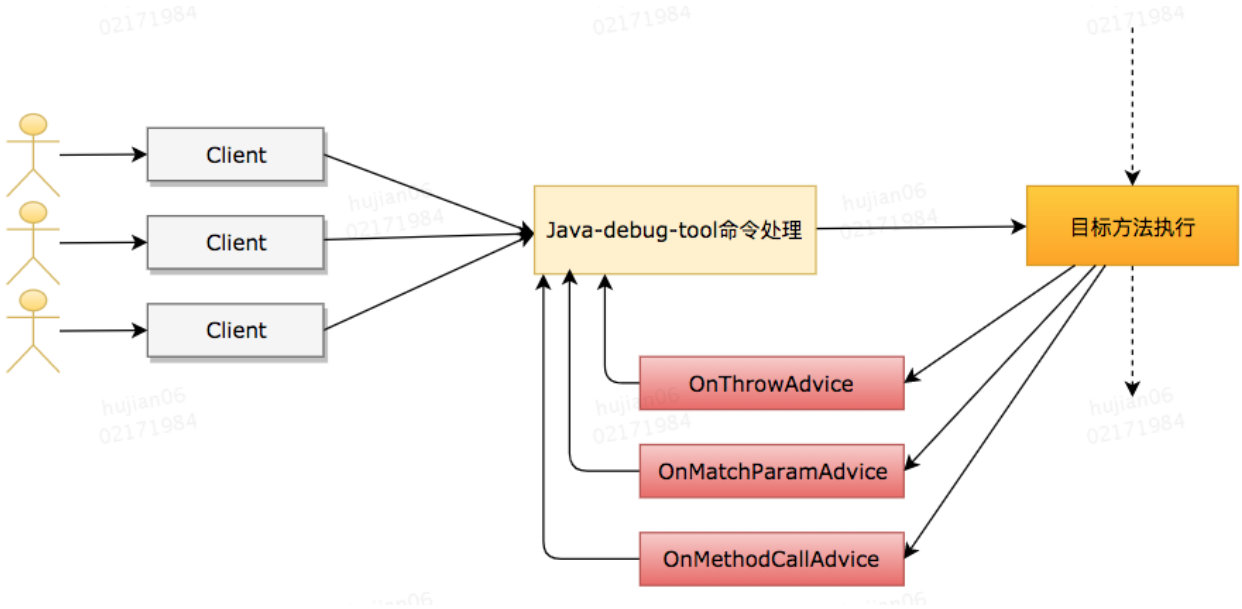
字节码增强模块收到字节码增强请求之后，会判断每个增强点是否需要插桩，这个判断的根据就是上文提到的插桩配置，之后字节码增强模块会生成新的字节码，Java-debug-tool将执行字节码替换操作，之后就可以进行调试数据收集了。

经过字节码增强之后，原来的方法中会插入收集运行时数据的代码，这些代码在方法被调用的时候执行，获取到诸如方法入参、局部变量等信息，这些信息将传递给数据收集装置进行处理。数据收集的工作通过Advice完成，每个客户端同一时间只能注册一个Advice到Java-debug-tool调试模块上，多个客户端可以同时注册自己的Advice到调试模块上。Advice负责收集数据并进行判断，如果当前数据符合调试命令的要求，Java-debug-tool就会卸载这个Advice，Advice的数据就会被转移到Java-debug-tool的命令结果处理模块进行处理，并将结果发送到客户端。

Advice的工作方式

Advice是调试数据收集器，不同的调试策略会对应不同的Advice。Advice是工作在目标JVM的线程内部的，它需要轻量级和高效，意味着Advice不能做太过于复杂的事情，它的核心接口“match”用来判断本次收集到的调试数据是否满足调试需求。如果满足，那么Java-debug-tool就会将其卸载，否则会继续让他收集调试数据，这种“加载Advice” -> “卸载Advice”的工作模式具备很好的灵活性。

关于Advice，需要说明的另外一点就是线程安全，因为它加载之后会运行在目标JVM的线程中，目标JVM的方法极有可能是多线程访问的，这也就是说，Advice需要有能力处理多个线程同时访问方法的能力，如果Advice处理不当，则可能会收集到杂乱无章的调试数据。下面的图片展示了Advice和Java-debug-tool调试分析模块、目标方法执行以及调试客户端等模块的关系。

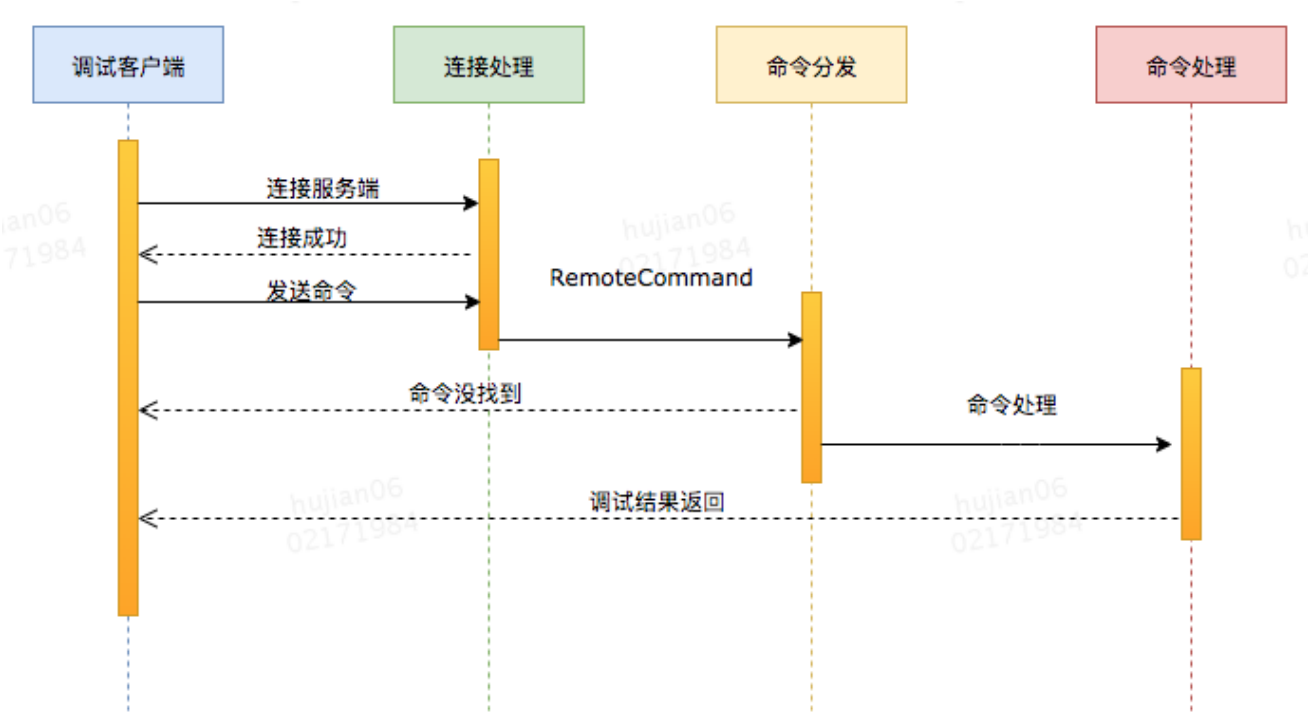


Advice的首次挂载由Java-debug-tool的命令处理器完成，当一次调试数据收集完成之后，调试数据处理模块会自动卸载Advice，然后进行判断，如果调试数据符合Advice的策略，则直接将数据交由数据处理模块进行处理，否则会清空调试数据，并再次将Advice挂载到目标方法上去，等待下一次调试数据。非首次挂载由调试数据处理模块进行，它借助Advice按需取数据，如果不符合需求，则继续挂载Advice来获取数据，否则对调试数据进行处理并返回给客户端。

*Java-debug-tool*的命令设计与实现

命令执行

上文已经完整的描述了Java-debug-tool的设计以及核心技术方案，本小节将详细介绍Java-debug-tool的命令设计与实现。首先需要将一个调试命令的执行流程描述清楚，下面是一张用来表示命令请求处理流程的图片：



上图简单的描述了Java-debug-tool的命令处理方式，客户端连接到服务端之后，会进行一些协议解析、协议认证、协议填充等工作，之后将进行命令分发。服务端如果发现客户端的命令不合法，则会立即返回错误信息，否则再进行命令处理。命令处理属于典型的三段式处理，前置命令处理、命令处理以及后置命令处理，同时会对命令处理过程中的异常信息进行捕获处理，三段式处理的好处是命令处理被拆成了多个阶段，多个阶段负责不同的职责。前置命令处理用来做一些命令权限控制的工作，并填充一些类似命令处理开始时间戳等信息，命令处理就是通过字节码增强，挂载Advice进行数据收集，再经过数据处理来产生命令结果的过程，后置处理则用来处理一些连接关闭、字节码解锁等事项。

Java-debug-tool允许客户端设置一个命令执行超时时间，超过这个时间则认为命令没有结果，如果客户端没有设置自己的超时时间，就使用默认的超时时间进行超时控制。Java-debug-tool通过设计了两阶段的超时检测机制来实现命令执行超时功能：首先，第一阶段超时触发，则Java-debug-tool会友好的警告命令处理模块处理时间已经超时，需要立即停止命令执行，这允许命令自己做一些现场清理工作，当然需要命令执行线程自己感知到这种超时警告；当第二阶段超时触发，则Java-debug-tool认为命令必须结束执行，会强行打断命令执行线程。超时机制的目的是为了不让命令执行太长时间，命令如果长时间没有收集到调试数据，则应该停止执行，并思考是否调试了一个错误的方法。当然，超时机制还可以定期清理那些因为未知原因断开连接的客户端持有的调试资源，比如字节码锁。

获取方法执行视图

Java-debug-tool通过下面的信息来向调试者呈现出一次方法执行的视图：

- 正在调试的方法信息。
- 方法调用堆栈。
- 调试耗时，包括对目标JVM造成的STW时间。
- 方法入参，包括入参的类型及参数值。
- 方法的执行路径。
- 代码执行耗时。
- 局部变量信息。
- 方法返回结果。
- 方法抛出的异常。
- 对象字段值快照。

下图展示了Java-debug-tool获取到正在运行的方法的执行视图的信息。

```
127.0.0.1:11234>mt -c R -m call -t watch -i #p0+#p1<10 -timeout 10 -s fd

-----------------------------------------------------------------------------------------------
命令                        : mt
命令执行Round               : 3
客户端ID                    : 10000
客户端类型                  : client:1
协议版本                    : version:1
命令耗时                    : 427 (ms)
STW时间                     : 0 (ms)
-----------------------------------------------------------------------------------------------
[R.call] by invoking:Thread[Thread-0,5,main]
with params
[
[0] @class:java.lang.Integer -> 2,
[1] @class:java.lang.Integer -> 4,
[2] @class:C -> {"@class":"C","a":0}
]
[1 ms] (34) [sa = 1]
[0 ms] (35)
[0 ms] (36) [ii = 0]
[0 ms] (37) [ij = 0]
[0 ms] (38) [jk = 1]
[0 ms] (39) [f = 1.0]
[0 ms] (40) [d = 0.123]
[0 ms] (41) [name = hello2,4]
[0 ms] (42) [list = [5]]
[0 ms] (43)
[0 ms] (47)
[0 ms] (50)
[0 ms] (51)
[0 ms] (53)
[0 ms] (57)
return value:[1]  at line:57 with cost:4 ms

Before Invoking Method
ClassField:
     |_ hello[world]@java.lang.String
ClassField:
     |_ input[11]@int

Before Exiting Method
ClassField:
     |_ hello[world]@java.lang.String
ClassField:
     |_ input[6]@int

-----------------------------------------------------------------------------------------------
```

*Java-debug-tool*与同类产品对比分析

Java-debug-tool的同类产品主要是greys，其他类似的工具大部分都是基于greys进行的二次开发，所以直接选择greys来和Java-debug-tool进行对比。

对比项	greys	Java-debug-tool	说明
多调试客户端支持	支持	支持	都是用观察者模式来支持多调试客户端同时调试；
方法增强	入参/返回/抛出异常	入参/执行路径/局部变量/出参/抛出异常/对象字段	相比greys，Java-debug-tool有更多的插桩选择，意味着可以获取到更丰富的调试信息
工具使用难度	适中，但是命令较多，需要配和多个命令进行调试	适中，调试集中在很少几个命令上	因为是基于shell的交互模式，两种工具都存在一定的使用难度
类隔离	自定义类加载器加载工具代码与目标JVM的类进行隔离	使用自定义类加载器	这样对目标JVM不会产生类污染
Agent加载模式	运行时加载	运行时加载	这也符合“动态调试”的场景需求
交互模式	命令行	命令行	greys的调试服务端是一个使用NIO实现的tcpServer，Java-debug-tool使用Netty实现调试服务端
命令数量	10+	5+	greys提供了大量的命令，而Java-debug-tool则专注于方法级别的链路追踪，没有提供太多的命令
表达式支持	支持，使用OGNL表达式；	支持，使用Spring表达式	Java-debug-tool的表达式仅在方法追踪时有用；
方法追踪调试信息获取	指定具体的调试信息，比如方法退出前；tt命令可以记录方法的执行信息；	执行mt命令，默认即可获取到包括方法入参、执行路径、执行耗时（包括单行耗时）、方法出参、抛出的异常等丰富的信息；	Java-debug-tool的mt命令和单步调试更类似，区别就是Java-debug-tool没有断点；
对象打印	如果类没有覆盖toString方法则不能很好的获取到对象的信息；	当检测到类没有覆盖toString方法之后，对象会被打印成json；	
命令特点	命令多，每个命令各司其职	命令较少，mt命令相当于聚合了多个greys的命令，调试信息较为丰富；	
网络连接管理	一个客户端连接idle一段时间会被关闭	一个客户端连接idle一段时间会被关闭，可配置服务端是否在所有连接都关闭的情况下关闭服务；	
语言	Java	Java	