SA-JDI全称Serviceability Agent-Java Debug Interface，是java提供的用于调试java程序的接口。源代码存储于openjdk/hotspot/agent文件夹，编译的jar包存储于jdk/lib/sa-jdi.jar。

SA-JDI主要由JAVA类组成，但它包含少量本地代码用于从进程和核心文件读取二进制位数据。在Linux上，SA-JDI使用/proc文件系统和ptrace函数从进程读取数据，SA可以直接解析核心文件。

PTRACE\_ATTACH

SA工具的基类是sun.jvm.hotspot.tools.Tool，通过调用start方法启动（jdk版本1.7），该方法会new一个BugSpotAgent，并且调用attach方法附着到目标VM上面。

BugSpotAgent#attach(int)

/\*\* This attaches to a process running on the local machine. \*/

public synchronized void attach(int processID)

throws DebuggerException {

if (debugger != null) {

throw new DebuggerException("Already attached");

}

pid = processID;

startupMode = PROCESS\_MODE;

isServer = false;

go();

}

private void go() {

setupDebugger();

javaMode = setupVM();

}

setupDebuggerLinux

private void setupDebuggerLinux() {

////////////////////////////////////////

// 1. 设置虚拟机库文件名

////////////////////////////////////////

setupJVMLibNamesLinux();

////////////////////////////////////////

// 2. new LinuxDebuggerLocal

////////////////////////////////////////

if (cpu.equals("x86")) {

machDesc = new MachineDescriptionIntelX86();

} else if (cpu.equals("ia64")) {

machDesc = new MachineDescriptionIA64();

} else if (cpu.equals("amd64")) {

machDesc = new MachineDescriptionAMD64();

} else if (cpu.equals("sparc")) {

if (LinuxDebuggerLocal.getAddressSize()==8) {

machDesc = new MachineDescriptionSPARC64Bit();

} else {

machDesc = new MachineDescriptionSPARC32Bit();

}

} else {

try {

machDesc = (MachineDescription)

Class.forName("sun.jvm.hotspot.debugger.MachineDescription" +

cpu.toUpperCase()).newInstance();

} catch (Exception e) {

throw new DebuggerException("unsupported machine type");

}

}

debugger = new LinuxDebuggerLocal(machDesc, !isServer);

////////////////////////////////////////

// 3. 调用LinuxDebuggerLocal的attach方法

////////////////////////////////////////

attachDebugger();

}

LinuxDebugerLocal#attach(int)调用本地方法attach0

/\*\* From the Debugger interface via JVMDebugger \*/

public synchronized void attach(int processID) throws DebuggerException {

checkAttached();

threadList = new ArrayList();

loadObjectList = new ArrayList();

class AttachTask implements WorkerThreadTask {

int pid;

public void doit(LinuxDebuggerLocal debugger) {

debugger.attach0(pid);

debugger.attached = true;

debugger.isCore = false;

findABIVersion();

}

}

AttachTask task = new AttachTask();

task.pid = processID;

workerThread.execute(task);

}

private native void attach0(int pid)

throws DebuggerException;

attacth0方法（LinuxDebuggerLocal.c）

/\*

\* Class: sun\_jvm\_hotspot\_debugger\_linux\_LinuxDebuggerLocal

\* Method: attach0

\* Signature: (I)V

\*/

JNIEXPORT void JNICALL Java\_sun\_jvm\_hotspot\_debugger\_linux\_LinuxDebuggerLocal\_attach0\_\_I

(JNIEnv \*env, jobject this\_obj, jint jpid) {

struct ps\_prochandle\* ph;

if ( (ph = Pgrab(jpid)) == NULL) {

THROW\_NEW\_DEBUGGER\_EXCEPTION("Can't attach to the process");

}

(\*env)->SetLongField(env, this\_obj, p\_ps\_prochandle\_ID, (jlong)(intptr\_t)ph);

fillThreadsAndLoadObjects(env, this\_obj, ph);

}

Pgarb方法（ps\_proc.c）

// attach to the process. One and only one exposed stuff

struct ps\_prochandle\* Pgrab(pid\_t pid) {

struct ps\_prochandle\* ph = NULL;

thread\_info\* thr = NULL;

if ( (ph = (struct ps\_prochandle\*) calloc(1, sizeof(struct ps\_prochandle))) == NULL) {

print\_debug("can't allocate memory for ps\_prochandle\n");

return NULL;

}

////////////////////////////////////////

// 1. attach到目标进程

////////////////////////////////////////

if (ptrace\_attach(pid) != true) {

free(ph);

return NULL;

}

////////////////////////////////////////

// 2. 填充ps\_prochandle

////////////////////////////////////////

// initialize ps\_prochandle

ph->pid = pid;

// initialize vtable

ph->ops = &process\_ops;

////////////////////////////////////////

// 读取目标进程的库文件信息和符号表

////////////////////////////////////////

// read library info and symbol tables, must do this before attaching threads,

// as the symbols in the pthread library will be used to figure out

// the list of threads within the same process.

read\_lib\_info(ph);

// read thread info

read\_thread\_info(ph, add\_new\_thread);

// attach to the threads

thr = ph->threads;

while (thr) {

// don't attach to the main thread again

if (ph->pid != thr->lwp\_id && ptrace\_attach(thr->lwp\_id) != true) {

// even if one attach fails, we get return NULL

Prelease(ph);

return NULL;

}

thr = thr->next;

}

return ph;

}

ptrace方法（sys/ptrace.h）

// attach to a process/thread specified by "pid"

static bool ptrace\_attach(pid\_t pid) {

if (ptrace(PTRACE\_ATTACH, pid, NULL, NULL) < 0) {

print\_debug("ptrace(PTRACE\_ATTACH, ..) failed for %d\n", pid);

return false;

} else {

return ptrace\_waitpid(pid);

}

}

setupVM

setupDebugger结束之后，需要[setupVM](http://hg.openjdk.java.net/jdk7u/jdk7u/hotspot/file/2cd3690f644c/agent/src/share/classes/sun/jvm/hotspot/bugspot/BugSpotAgent.java#l554)，

private boolean setupVM() {

// We need to instantiate a HotSpotTypeDataBase on both the client

// and server machine. On the server it is only currently used to

// configure the Java primitive type sizes (which we should

// consider making constant). On the client it is used to

// configure the VM.

////////////////////////////////////////

// 1. 构建HotSpotTypeDataBase

////////////////////////////////////////

try {

if (os.equals("solaris")) {

db = new HotSpotTypeDataBase(machDesc, new HotSpotSolarisVtblAccess(debugger, jvmLibNames),

debugger, jvmLibNames);

} else if (os.equals("win32")) {

db = new HotSpotTypeDataBase(machDesc, new Win32VtblAccess(debugger, jvmLibNames),

debugger, jvmLibNames);

} else if (os.equals("linux")) {

db = new HotSpotTypeDataBase(machDesc, new LinuxVtblAccess(debugger, jvmLibNames),

debugger, jvmLibNames);

} else if (os.equals("bsd")) {

db = new HotSpotTypeDataBase(machDesc, new BsdVtblAccess(debugger, jvmLibNames),

debugger, jvmLibNames);

} else {

throw new DebuggerException("OS \"" + os + "\" not yet supported (no VtblAccess implemented yet)");

}

}

catch (NoSuchSymbolException e) {

e.printStackTrace();

return false;

}

////////////////////////////////////////

// 2. 设置原生类型大小，从目标VM获取

////////////////////////////////////////

if (startupMode != REMOTE\_MODE) {

// Configure the debugger with the primitive type sizes just obtained from the VM

debugger.configureJavaPrimitiveTypeSizes(db.getJBooleanType().getSize(),

db.getJByteType().getSize(),

db.getJCharType().getSize(),

db.getJDoubleType().getSize(),

db.getJFloatType().getSize(),

db.getJIntType().getSize(),

db.getJLongType().getSize(),

db.getJShortType().getSize());

}

////////////////////////////////////////

// 3. 构建目标VM的本机表示

////////////////////////////////////////

if (!isServer) {

// Do not initialize the VM on the server (unnecessary, since it's

// instantiated on the client)

VM.initialize(db, debugger);

}

try {

jvmdi = new ServiceabilityAgentJVMDIModule(debugger, saLibNames);

if (jvmdi.canAttach()) {

jvmdi.attach();

jvmdi.setCommandTimeout(6000);

debugPrintln("Attached to Serviceability Agent's JVMDI module.");

// Jog VM to suspended point with JVMDI module

resume();

suspendJava();

suspend();

debugPrintln("Suspended all Java threads.");

} else {

debugPrintln("Could not locate SA's JVMDI module; skipping attachment");

jvmdi = null;

}

} catch (Exception e) {

e.printStackTrace();

jvmdi = null;

}

return true;

}

HotSpotTypeDataBase

sun.jvm.hotspot. HotSpotTypeDataBase用于存储目标VM上面的类型信息。

public HotSpotTypeDataBase(MachineDescription machDesc,

VtblAccess vtblAccess,

Debugger symbolLookup,

String[] jvmLibNames) throws NoSuchSymbolException {

super(machDesc, vtblAccess);

this.symbolLookup = symbolLookup;

this.jvmLibNames = jvmLibNames;

readVMTypes();

initializePrimitiveTypes();

readVMStructs();

readVMIntConstants();

readVMLongConstants();

readExternalDefinitions();

}

readXXX方法从HotSpotVM中读取信息。

readVMStructs

private void readVMStructs() {

////////////////////////////////////////

// VMStructEntry结构体各个成员变量的offset

////////////////////////////////////////

// Get the variables we need in order to traverse the VMStructEntry[]

long structEntryTypeNameOffset;

long structEntryFieldNameOffset;

long structEntryTypeStringOffset;

long structEntryIsStaticOffset;

long structEntryOffsetOffset;

long structEntryAddressOffset;

long structEntryArrayStride;

structEntryTypeNameOffset = getLongValueFromProcess("gHotSpotVMStructEntryTypeNameOffset");

structEntryFieldNameOffset = getLongValueFromProcess("gHotSpotVMStructEntryFieldNameOffset");

structEntryTypeStringOffset = getLongValueFromProcess("gHotSpotVMStructEntryTypeStringOffset");

structEntryIsStaticOffset = getLongValueFromProcess("gHotSpotVMStructEntryIsStaticOffset");

structEntryOffsetOffset = getLongValueFromProcess("gHotSpotVMStructEntryOffsetOffset");

structEntryAddressOffset = getLongValueFromProcess("gHotSpotVMStructEntryAddressOffset");

structEntryArrayStride = getLongValueFromProcess("gHotSpotVMStructEntryArrayStride");

////////////////////////////////////////

// 通过符号表查找目标VM中gHotSpotVMStructs变量的地址

////////////////////////////////////////

// Fetch the address of the VMStructEntry\*

Address entryAddr = lookupInProcess("gHotSpotVMStructs");

// Dereference this once to get the pointer to the first VMStructEntry

entryAddr = entryAddr.getAddressAt(0);

if (entryAddr == null) {

throw new RuntimeException("gHotSpotVMStructs was not initialized properly in the remote process; can not continue");

}

// Start iterating down it until we find an entry with no name

Address fieldNameAddr = null;

String typeName = null;

String fieldName = null;

String typeString = null;

boolean isStatic = false;

long offset = 0;

Address staticFieldAddr = null;

long size = 0;

long index = 0;

String opaqueName = "<opaque>";

lookupOrCreateClass(opaqueName, false, false, false);

do {

////////////////////////////////////////

// 读取VMStructEntry各个成员变量的值

////////////////////////////////////////

// Fetch the field name first

fieldNameAddr = entryAddr.getAddressAt(structEntryFieldNameOffset);

if (fieldNameAddr != null) {

fieldName = CStringUtilities.getString(fieldNameAddr);

// Now the rest of the names. Keep in mind that the type name

// may be NULL, indicating that the type is opaque.

Address addr = entryAddr.getAddressAt(structEntryTypeNameOffset);

if (addr == null) {

throw new RuntimeException("gHotSpotVMStructs unexpectedly had a NULL type name at index " + index);

}

typeName = CStringUtilities.getString(addr);

addr = entryAddr.getAddressAt(structEntryTypeStringOffset);

if (addr == null) {

typeString = opaqueName;

} else {

typeString = CStringUtilities.getString(addr);

}

isStatic = !(entryAddr.getCIntegerAt(structEntryIsStaticOffset, C\_INT32\_SIZE, false) == 0);

if (isStatic) {

staticFieldAddr = entryAddr.getAddressAt(structEntryAddressOffset);

offset = 0;

} else {

offset = entryAddr.getCIntegerAt(structEntryOffsetOffset, C\_INT64\_SIZE, true);

staticFieldAddr = null;

}

// The containing Type must already be in the database -- no exceptions

BasicType containingType = lookupOrFail(typeName);

// The field's Type must already be in the database -- no exceptions

BasicType fieldType = (BasicType)lookupType(typeString);

////////////////////////////////////////

// 根据VMStructEntry创建对应的Field

////////////////////////////////////////

// Create field by type

createField(containingType, fieldName, fieldType,

isStatic, offset, staticFieldAddr);

}

++index;

////////////////////////////////////////

// 下一个VMStructEntry

////////////////////////////////////////

entryAddr = entryAddr.addOffsetTo(structEntryArrayStride);

} while (fieldNameAddr != null);

}

lookupInProcess

// Fetch the address of the VMStructEntry\*

Address entryAddr = lookupInProcess("gHotSpotVMStructs");

private Address lookupInProcess(String symbol) throws NoSuchSymbolException {

// FIXME: abstract away the loadobject name

for (int i = 0; i < jvmLibNames.length; i++) {

////////////////////////////////////////

// 委托给Debugger来查找

////////////////////////////////////////

Address addr = symbolLookup.lookup(jvmLibNames[i], symbol);

if (addr != null) {

return addr;

}

}

String errStr = "(";

for (int i = 0; i < jvmLibNames.length; i++) {

errStr += jvmLibNames[i];

if (i < jvmLibNames.length - 1) {

errStr += ", ";

}

}

errStr += ")";

throw new NoSuchSymbolException(symbol,

"Could not find symbol \"" + symbol +

"\" in any of the known library names " +

errStr);

}

lookup方法

/\*\* From the SymbolLookup interface via Debugger and JVMDebugger \*/

public synchronized Address lookup(String objectName, String symbol) {

requireAttach();

if (!attached) {

return null;

}

if (isCore) {

long addr = lookupByName0(objectName, symbol);

return (addr == 0)? null : new LinuxAddress(this, handleGCC32ABI(addr, symbol));

} else {

class LookupByNameTask implements WorkerThreadTask {

String objectName, symbol;

Address result;

public void doit(LinuxDebuggerLocal debugger) {

long addr = debugger.lookupByName0(objectName, symbol);

result = (addr == 0 ? null : new LinuxAddress(debugger, handleGCC32ABI(addr, symbol)));

}

}

LookupByNameTask task = new LookupByNameTask();

task.objectName = objectName;

task.symbol = symbol;

workerThread.execute(task);

return task.result;

}

}

lookupByName0方法

/\*

\* Class: sun\_jvm\_hotspot\_debugger\_linux\_LinuxDebuggerLocal

\* Method: lookupByName0

\* Signature: (Ljava/lang/String;Ljava/lang/String;)J

\*/

JNIEXPORT jlong JNICALL Java\_sun\_jvm\_hotspot\_debugger\_linux\_LinuxDebuggerLocal\_lookupByName0

(JNIEnv \*env, jobject this\_obj, jstring objectName, jstring symbolName) {

const char \*objectName\_cstr, \*symbolName\_cstr;

jlong addr;

jboolean isCopy;

////////////////////////////////////////

// 获取之前attach后保存下来的ps\_prochandle

////////////////////////////////////////

struct ps\_prochandle\* ph = get\_proc\_handle(env, this\_obj);

objectName\_cstr = NULL;

if (objectName != NULL) {

objectName\_cstr = (\*env)->GetStringUTFChars(env, objectName, &isCopy);

CHECK\_EXCEPTION\_(0);

}

symbolName\_cstr = (\*env)->GetStringUTFChars(env, symbolName, &isCopy);

CHECK\_EXCEPTION\_(0);

////////////////////////////////////////

// 调用lookup\_symbol方法

////////////////////////////////////////

addr = (jlong) lookup\_symbol(ph, objectName\_cstr, symbolName\_cstr);

if (objectName\_cstr != NULL) {

(\*env)->ReleaseStringUTFChars(env, objectName, objectName\_cstr);

}

(\*env)->ReleaseStringUTFChars(env, symbolName, symbolName\_cstr);

return addr;

}

[lookup\_symbol方法](http://hg.openjdk.java.net/jdk7u/jdk7u/hotspot/file/2cd3690f644c/agent/src/os/linux/libproc_impl.c#l206)

// lookup for a specific symbol

uintptr\_t lookup\_symbol(struct ps\_prochandle\* ph, const char\* object\_name,

const char\* sym\_name) {

////////////////////////////////////////

// 传进来的库文件名被忽略了

////////////////////////////////////////

// ignore object\_name. search in all libraries

// FIXME: what should we do with object\_name?? The library names are obtained

// by parsing /proc/<pid>/maps, which may not be the same as object\_name.

// What we need is a utility to map object\_name to real file name, something

// dlopen() does by looking at LD\_LIBRARY\_PATH and /etc/ld.so.cache. For

// now, we just ignore object\_name and do a global search for the symbol.

////////////////////////////////////////

// 在之前保存下来的符号表中查找

////////////////////////////////////////

lib\_info\* lib = ph->libs;

while (lib) {

if (lib->symtab) {

uintptr\_t res = search\_symbol(lib->symtab, lib->base, sym\_name, NULL);

if (res) return res;

}

lib = lib->next;

}

print\_debug("lookup failed for symbol '%s' in obj '%s'\n",

sym\_name, object\_name);

return (uintptr\_t) NULL;

}

SA中有两种方式来获取HotSpotVM里面的变量地址，一种是通过符号表，另一种是通过VMStructEntry这种VM提供的元信息（也就是通过&运算获取的地址）。

PTRACE\_PEEKDATA

LinuxAddress#getCIntegerAt方法

public long getCIntegerAt(long offset, long numBytes, boolean isUnsigned)

throws UnalignedAddressException, UnmappedAddressException {

return debugger.readCInteger(addr + offset, numBytes, isUnsigned);

}

LinuxDebuggerLocal

public long readCInteger(long address, long numBytes, boolean isUnsigned)

throws UnmappedAddressException, UnalignedAddressException {

// Only slightly relaxed semantics -- this is a hack, but is

// necessary on x86 where it seems the compiler is

// putting some global 64-bit data on 32-bit boundaries

if (numBytes == 8) {

utils.checkAlignment(address, 4);

} else {

utils.checkAlignment(address, numBytes);

}

byte[] data = readBytes(address, numBytes);

return utils.dataToCInteger(data, isUnsigned);

}

DebuggerBase#readBytes -> LinuxDebuggerLocal#readBytesFromProcess ->LinuxDebuggerLocal#readBytesFromProcess0

JNIEXPORT jbyteArray JNICALL Java\_sun\_jvm\_hotspot\_debugger\_linux\_LinuxDebuggerLocal\_readBytesFromProcess0

(JNIEnv \*env, jobject this\_obj, jlong addr, jlong numBytes) {

jboolean isCopy;

jbyteArray array;

jbyte \*bufPtr;

ps\_err\_e err;

array = (\*env)->NewByteArray(env, numBytes);

CHECK\_EXCEPTION\_(0);

bufPtr = (\*env)->GetByteArrayElements(env, array, &isCopy);

CHECK\_EXCEPTION\_(0);

err = ps\_pdread(get\_proc\_handle(env, this\_obj), (psaddr\_t) (uintptr\_t)addr, bufPtr, numBytes);

(\*env)->ReleaseByteArrayElements(env, array, bufPtr, 0);

return (err == PS\_OK)? array : 0;

}

ps\_pdread

// read "size" bytes info "buf" from address "addr"

ps\_err\_e ps\_pdread(struct ps\_prochandle \*ph, psaddr\_t addr,

void \*buf, size\_t size) {

return ph->ops->p\_pread(ph, (uintptr\_t) addr, buf, size)? PS\_OK: PS\_ERR;

}

static ps\_prochandle\_ops process\_ops = {

.release= process\_cleanup,

.p\_pread= process\_read\_data,

.p\_pwrite= process\_write\_data,

.get\_lwp\_regs= process\_get\_lwp\_regs

};

process\_read\_data方法

// read "size" bytes of data from "addr" within the target process.

// unlike the standard ptrace() function, process\_read\_data() can handle

// unaligned address - alignment check, if required, should be done

// before calling process\_read\_data.

static bool process\_read\_data(struct ps\_prochandle\* ph, uintptr\_t addr, char \*buf, size\_t size) {

long rslt;

size\_t i, words;

uintptr\_t end\_addr = addr + size;

uintptr\_t aligned\_addr = align(addr, sizeof(long));

if (aligned\_addr != addr) {

char \*ptr = (char \*)&rslt;

errno = 0;

rslt = ptrace(PTRACE\_PEEKDATA, ph->pid, aligned\_addr, 0);

if (errno) {

print\_debug("ptrace(PTRACE\_PEEKDATA, ..) failed for %d bytes @ %lx\n", size, addr);

return false;

}

for (; aligned\_addr != addr; aligned\_addr++, ptr++);

for (; ((intptr\_t)aligned\_addr % sizeof(long)) && aligned\_addr < end\_addr;

aligned\_addr++)

\*(buf++) = \*(ptr++);

}

words = (end\_addr - aligned\_addr) / sizeof(long);

// assert((intptr\_t)aligned\_addr % sizeof(long) == 0);

for (i = 0; i < words; i++) {

errno = 0;

rslt = ptrace(PTRACE\_PEEKDATA, ph->pid, aligned\_addr, 0);

if (errno) {

print\_debug("ptrace(PTRACE\_PEEKDATA, ..) failed for %d bytes @ %lx\n", size, addr);

return false;

}

\*(long \*)buf = rslt;

buf += sizeof(long);

aligned\_addr += sizeof(long);

}

if (aligned\_addr != end\_addr) {

char \*ptr = (char \*)&rslt;

errno = 0;

rslt = ptrace(PTRACE\_PEEKDATA, ph->pid, aligned\_addr, 0);

if (errno) {

print\_debug("ptrace(PTRACE\_PEEKDATA, ..) failed for %d bytes @ %lx\n", size, addr);

return false;

}

for (; aligned\_addr != end\_addr; aligned\_addr++)

\*(buf++) = \*(ptr++);

}

return true;

}

通过ptrace方法传入PTRACE\_PEEKDATA，pid，地址参数来读取目标VM中的数据。

1.通过/proc/[pid]/maps读取ELF文件，保存符号表；

2.通过符号表读取HotSpotVM中localHotSpotVMStructs，localHotSpotVMTypes等变量的地址；

3.使用ptrace读取上述变量的值；

4.这两个变量值包含了SA需要用到的HotSpotVM中的数据的元信息（类型信息，字段offset，地址等）；

5.有了这些元信息就可以使用ptrace读取目标VM上这些数据的值。