JVM：从入门到入门（二）
jvm的启动

PCLT Lab

zhangxiang@nj.iscas.ac.cn

2021.8.11

- 以bishengJDK的源码作为学习模板。

- 源码地址：https://gitee.com/openeuler/bishengjdk-11/tree/risc-v
- 构建编译参考：https://github.com/openjdk-riscv/jdk11u/wiki/Configure-Guide
- GDB调试参考：https://github.com/openjdk-riscv/jdk11u/wiki/Debug-with-GDB-in-QEMU-user-mode

从main函数开始

- 我们选择slowdebug版本，是不含编译优化的。
- fastdebug版本是含有一些编译优化的。

- 利用gdb调试进入
- <path-to-qemu>/qemu/bin/qemu-riscv32 -L <path-to-riscv32>/riscv32/sysroot -g 33334 ./java -version &

- <path-to-riscv32>/riscv32/bin/riscv32-unknown-linux-gnu-gdb --args ./java -version

- target remote localhost:33334

```
Remote debugging using localhost:33334
Remote communication error.  Target disconnected.: Connection reset by peer.
(gdb) b main
Breakpoint 1 at 0x10846: file /home/zhangxiang/jdk/jdk11u/src/java.base/share/native/launcher/main.c, line 98.
(gdb)
```

```
#else /* JAVAW */
JNIEXPORT int
main(int argc, char **argv)
{
    int margc;
    char** margv;
    int jargc;
    char** jargv;
    const jboolean const_javaw = JNI_FALSE;
#endif /* JAVAW */
    {
        int i, main_jargc, extra_jargc;
```

```
    return JLI_Launch(margc, margv,
                      jargc, (const char**) jargv,
                      0, NULL,
                      VERSION_STRING,
                      DOT_VERSION,
                      (const_progname != NULL) ? const_progname : *margv,
                      (const_launcher != NULL) ? const_launcher : *margv,
                      jargc > 0,
                      const_cpwildcard, const_javaw, 0);
```

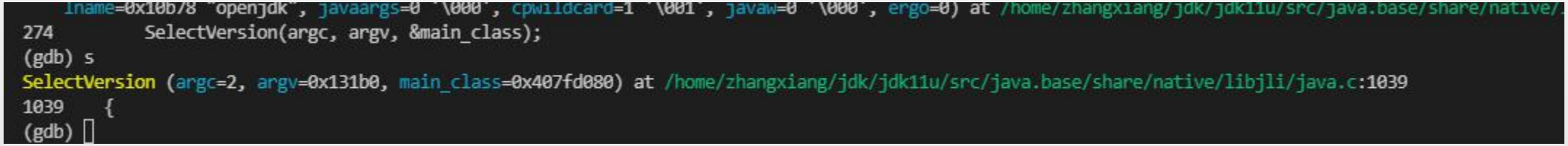
启动参数

```
(gdb) adv 274
JLI_Launch (argc=2, argv=0x131b0, jargc=0, jargv=0x12020 <const_jargs>, appclassc=0, appclassv=0x0, fullversion=0x10c08 "11.
0.9-internal+0-adhoc.zhangxiang.jdk11u", dotversion=0x10c04 "0.0", pname=0x10b68 "java",
    lname=0x10b78 "openjdk", javaargs=0 '\000', cpwildcard=1 '\001', javaw=0 '\000', ergo=0) at /home/zhangxiang/jdk/jdk11u/
src/java.base/share/native/libjli/java.c:274
```

主要流程

- 1、SelectVersion，从jar包中manifest文件或者命令行读取用户使用的JDK版本，判断当前版本是否合适。
- 2、CreateExecutionEnvironment，设置执行环境参数
- 3、LoadJavaVM，加载libjvm动态链接库
- 4、ParseArguments 解析命令行参数，如-version，-help等参数在该方法中解析的
- 5、JVMInit。

- 1)SelectVersion：选择jre的版本,这个函数实现的功能比较简单，就是选择正确的jre版本来作为即将运行java程序的版本。选择的方式，如果环境变量设置了_JAVA_VERSION_SET，那么代表已经选择了jre的版本，不再进行选择；否则，根据运行时给定的参数来搜索不同的目录选择，例如指定版本和限制了搜索目录等，也可能执行的是一个jar文件，所以需要解析manifest文件来获取相关信息，对应Manifest文件的数据结构如下，可以看到对应注释都是涉及到jre版本的相关信息。



```
 * Java launcher).
 */
typedef struct manifest_info {  /* Interesting fields from the Manifest */
    char        *manifest_version;      /* Manifest-Version string */
    char        *main_class;            /* Main-Class entry */
    char        *jre_version;           /* Appropriate J2SE release spec */
    char        jre_restrict_search;    /* Restricted JRE search */
    char        *splashscreen_image_file_name; /* splashscreen image file */
} manifest_info;

/*
```

```
argc--;
argv++;
while ((arg = *argv) != 0 && *arg == '-') {
    has_arg = IsOptionWithArgument(argc, argv);
    if (JLI_StrCCmp(arg, "-version:") == 0) {
        JLI_ReportErrorMessage(SPC_ERROR1);
    } else if (JLI_StrCmp(arg, "-jre-restrict-search") == 0) {
        JLI_ReportErrorMessage(SPC_ERROR2);
    } else if (JLI_StrCmp(arg, "-jre-no-restrict-search") == 0) {
        JLI_ReportErrorMessage(SPC_ERROR2);
    } else {
        if (JLI_StrCmp(arg, "-jar") == 0)
            jarflag = 1;
        if (IsWhiteSpaceOption(arg)) {
            if (has_arg) {
                argc--;
                argv++;
                arg = *argv;
            }
```

- 2)CreateExecutionEnvironment，这个函数主要创建执行的一些环境，这个环境主要是指jvm的环境，例如需要确定数据模型，是32位还是64位以及jvm本身的一些配置在jvm.cfg文件中读取和解析。

```
(gdb) s
CreateExecutionEnvironment (pargc=0x407fd06c, pargv=0x407fd068, jrepath=0x407fe0ac "", so_jrepath=4096, jvmpath=0x407fd0ac "", so_jvmpath=4096, jvmcfg=0x407ff0ac "", so_jvmcfg=
    at /home/zhangxiang/jdk/jdk11u/src/java.base/unix/native/libjli/java_md_solinux.c:299
299                             char jvmcfg[],  jint so_jvmcfg) {
(gdb)
```

```
/* Values for vmdesc.flag */
enum vmdesc_flag {
    VM_UNKNOWN = -1,
    VM_KNOWN,
    VM_ALIASED_TO,
    VM_WARN,
    VM_ERROR,
    VM_IF_SERVER_CLASS,
    VM_IGNORE
};
```

```
};

struct vmdesc {
    char *name;
    int flag;
    char *alias;
    char *server_class;
};
```

```
#ifdef SETENV_REQUIRED
    jboolean mustsetenv = JNI_FALSE;
    char *runpath = NULL; /* existing effective LD_LIBRARY_PATH setting */
    char* new_runpath = NULL; /* desired new LD_LIBRARY_PATH string */
    char* newpath = NULL; /* path on new LD_LIBRARY_PATH */
    char* lastslash = NULL;
    char** newenvp = NULL; /* current environment */
    size_t new_runpath_size;
#endif  /* SETENV_REQUIRED */

    /* Compute/set the name of the executable */
    SetExecname(*pargv);

    /* Check to see if the jvmpath exists */
    /* Find out where the JRE is that we will be using. */
    if (!GetJREPath(jrepath, so_jrepath, JNI_FALSE)) {
        JLI_ReportErrorMessage(JRE_ERROR1);
        exit(2);
    }
    JLI_Snprintf(jvmcfg, so_jvmcfg, "%s%slib%sjvm.cfg",
            jrepath, FILESEP, FILESEP);
    /* Find the specified JVM type */
    if (ReadKnownVMs(jvmcfg, JNI_FALSE) < 1) {
        JLI_ReportErrorMessage(CFG_ERROR7);
        exit(1);
    }

    jvmpath[0] = '\0';
    jvmtype = CheckJvmType(pargc, pargv, JNI_FALSE);
    if (JLI_StrCmp(jvmtype, "ERROR") == 0) {
        JLI_ReportErrorMessage(CFG_ERROR9);
        exit(4);
    }

    if (!GetJVMPath(jrepath, jvmtype, jvmpath, so_jvmpath)) {
        JLI_ReportErrorMessage(CFG_ERROR8, jvmtype, jvmpath);
        exit(4);
    }
    /*
     * we seem to have everything we need, so without further ado
     * we return back, otherwise proceed to set the environment.
     */
#ifdef SETENV_REQUIRED
```

从函数主体内容看，主要涉及到找的jre是否是需要的jre（通过jrepath），找到对应的jvm的类型，
这个函数主要是确定jvm的信息并且做一个初始化相关信息，为后面的jvm执行做准备。

```
(gdb) s
LoadJavaVM (jvmpath=0x407fd0ac "/home/zhangxiang/jdk/jdk11u/build/linux-riscv32-normal-core-slowdebug/jdk/lib/server/libjvm.so", ifn=0x407fd0a0)
    at /home/zhangxiang/jdk/jdk11u/src/java.base/unix/native/libjli/java_md_solinux.c:546
546     {
```

- 3）LoadJavaVM：动态加载jvm.so这个共享库，并把jvm.so中的相关函数导出并且初始化，例如JNI_CreateJavaVM函数。后期启动真正的java虚拟就是通过这里面加载的函数。

```
    }
#endif
        JLI_ReportErrorMessage(DLL_ERROR1, __LINE__);
        JLI_ReportErrorMessage(DLL_ERROR2, jvmpath, dlerror());
        return JNI_FALSE;
    }

    ifn->CreateJavaVM = (CreateJavaVM_t)
        dlsym(libjvm, "JNI_CreateJavaVM");
    if (ifn->CreateJavaVM == NULL) {
        JLI_ReportErrorMessage(DLL_ERROR2, jvmpath, dlerror());
        return JNI_FALSE;
    }

    ifn->GetDefaultJavaVMInitArgs = (GetDefaultJavaVMInitArgs_t)
        dlsym(libjvm, "JNI_GetDefaultJavaVMInitArgs");
    if (ifn->GetDefaultJavaVMInitArgs == NULL) {
        JLI_ReportErrorMessage(DLL_ERROR2, jvmpath, dlerror());
        return JNI_FALSE;
    }

    ifn->GetCreatedJavaVMs = (GetCreatedJavaVMs_t)
        dlsym(libjvm, "JNI_GetCreatedJavaVMs");
    if (ifn->GetCreatedJavaVMs == NULL) {
        JLI_ReportErrorMessage(DLL_ERROR2, jvmpath, dlerror());
        return JNI_FALSE;
    }

    return JNI_TRUE;
}
```

- 4）是命令行参数解析

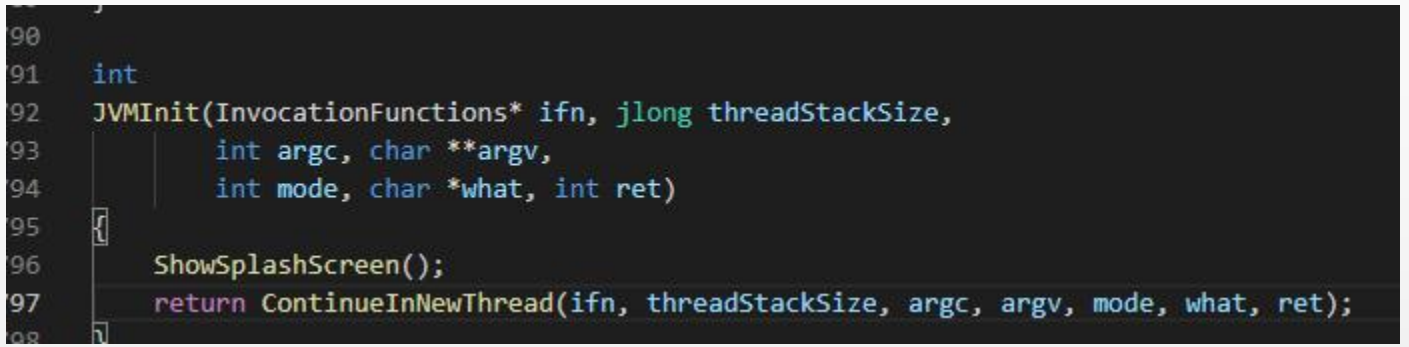
```
            REPORT_ERROR (has_arg, ARG_ERROR4, arg);

        } else if (!has_arg && (IsModuleOption(arg) || IsLongFormModuleOpti
            REPORT_ERROR (has_arg, ARG_ERROR6, arg);
/*
 * The following cases will cause the argument parsing to stop
 */
        } else if (JLI_StrCmp(arg, "-help") == 0 ||
                   JLI_StrCmp(arg, "-h") == 0 ||
                   JLI_StrCmp(arg, "-?") == 0) {
            printUsage = JNI_TRUE;
            return JNI_TRUE;
        } else if (JLI_StrCmp(arg, "--help") == 0) {
            printUsage = JNI_TRUE;
            printTo = USE_STDOUT;
            return JNI_TRUE;
        } else if (JLI_StrCmp(arg, "-version") == 0) {
            printVersion = JNI_TRUE;
            return JNI_TRUE;
        } else if (JLI_StrCmp(arg, "--version") == 0) {
            printVersion = JNI_TRUE;
            printTo = USE_STDOUT;
            return JNI_TRUE;
        } else if (JLI_StrCmp(arg, "-showversion") == 0) {
            showVersion = JNI_TRUE;
        } else if (JLI_StrCmp(arg, "--show-version") == 0) {
            showVersion = JNI_TRUE;
            printTo = USE_STDOUT;
        } else if (JLI_StrCmp(arg, "--dry-run") == 0) {
            dryRun = JNI_TRUE;
        } else if (JLI_StrCmp(arg, "-X") == 0) {
```

输出　终端　端口

可以看到对“-help”、“-h”以及“-version”的比较

- JVMInit：这是启动流程最后执行的一个函数，如果这个函数返回了那么这个java启动就结束了，所有这个函数最终会以某种形式进行执行下去



- JVMInit->ContinueInNewThread->ContinueInNewThread0->(可能是新线程的入口函数进行执行，新线程创建失败就在原来的线程继续支持这个函数)JavaMain->InitializeJVM(初始化jvm，这个函数调用jvm.so里面导出的CreateJavaVM函数创建jvm了，JNI_CreateJavaVM这个函数很复杂)->LoadMainClass（这个函数就是找到我们真正java程序的入口类，就是我们开发应用程序带有main函数的类）->GetApplicationClass->后面就是调用环境类的工具获得main函数并且传递参数调用main函数，查找main和调用main函数都是使用类似java里面支持的反射实现的。

- 参考来源：https://zhuanlan.zhihu.com/p/379257556

谢谢！