第1篇-关于运行时，开篇说的简单些

Original 马智 深入剖析Java虚拟机HotSpot 2021-12-03 16:44

收录于合集

#Java\ 1 #虚拟机 10 #hotspot 10 #运行时 9 #main() 3

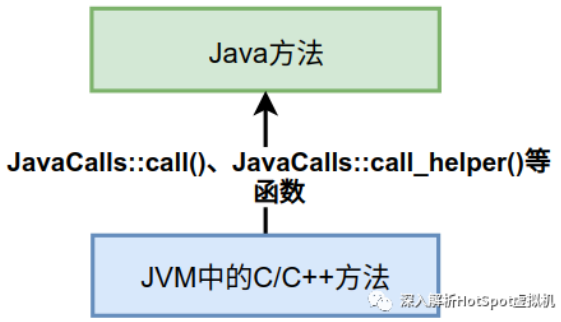
深入剖析Java虚拟机HotSpot

对HotSpot VM进行深度源码剖析，如果要系统的学习相关内容，推荐作者的《深入剖析Ja...

84篇原创内容

公众号

开讲Java运行时，这一篇讲一些简单的内容。我们写的主类中的main()方法是如何被Java虚拟机调用到的？在Java类中的一些方法会被由C/C++编写的HotSpot虚拟机的C/C++函数调用，不过由于Java方法与C/C++函数的调用约定不同，所以并不能直接调用，需要JavaCalls::call()这个函数辅助调用。（我把由C/C++编写的叫函数，把Java编写的叫方法，后续也会延用这样的叫法）如下图所示。



从C/C++方法中调用的一些Java方法主要有：

（1）Java主类中的main()方法；

（2）Java主类装载时，调用JavaCalls::call()函数执行checkAndLoadMain()方法；

（3）类的初始化过程中，调用JavaCalls::call()函数执行的Java类初始化方法<clinit>，可以查看JavaCalls::call_default_constructor()函数，有对<clinit>方法的调用逻辑；

（4）我们先省略main方法的执行流程（其实main方法的执行也是先启动一个JavaMain线程，套路都是一样的），单看某个JavaThread的启动过程。JavaThread的启动最终都要通过一个native方法java.lang.Thread#start0()完成的，这个方法经过解释器的native_entry入口，调用到了JVM_StartThread()函数。其中的static void thread_entry(JavaThread* thread, TRAPS)函数中会调用JavaCalls::call_virtual()函数。JavaThread最终会通过JavaCalls::call_virtual()函数来调用字节码中的run()方法；

（5）在SystemDictionary::load_instance_class()这个能体现双亲委派的函数中，如果类加载器对象不为空，则会调用这个类加载器的loadClass()函数（通过call_virtual()函数来调用）来加载类。

当然还会有其它方法，这里就不一一列举了。通过JavaCalls::call()、JavaCalls::call_helper()等函数调用Java方法，这些函数定义在JavaCalls类中，这个类的定义如下：

源代码位置：openjdk/hotspot/src/share/vm/runtime/javaCalls.hpp

```
class JavaCalls: AllStatic {
  static void call_helper(JavaValue* result, methodHandle* method, JavaCallArguments* args, TRAPS);
 public:

  static void call_default_constructor(JavaThread* thread, methodHandle method, Handle receiver, TRAPS);

  // 使用如下函数调用Java中一些特殊的方法，如类初始化方法<clinit>等
  // receiver表示方法的接收者，如A.main()调用中，A就是方法的接收者
  static void call_special(JavaValue* result, KlassHandle klass, Symbol* name,Symbol* signature, JavaCallArgum
  static void call_special(JavaValue* result, Handle receiver, KlassHandle klass,Symbol* name, Symbol* signatu
  static void call_special(JavaValue* result, Handle receiver, KlassHandle klass,Symbol* name, Symbol* signatu
  static void call_special(JavaValue* result, Handle receiver, KlassHandle klass,Symbol* name, Symbol* signatu

  // 使用如下函数调用动态分派的一些方法
  static void call_virtual(JavaValue* result, KlassHandle spec_klass, Symbol* name,Symbol* signature, JavaCall
  static void call_virtual(JavaValue* result, Handle receiver, KlassHandle spec_klass,Symbol* name, Symbol* si
  static void call_virtual(JavaValue* result, Handle receiver, KlassHandle spec_klass,Symbol* name, Symbol* si
  static void call_virtual(JavaValue* result, Handle receiver, KlassHandle spec_klass,Symbol* name, Symbol* si

  // 使用如下函数调用Java静态方法
  static void call_static(JavaValue* result, KlassHandle klass,Symbol* name, Symbol* signature, JavaCallArgume
   static void call_static(JavaValue* result, KlassHandle klass,Symbol* name, Symbol* signature, TRAPS);
  static void call_static(JavaValue* result, KlassHandle klass,Symbol* name, Symbol* signature, Handle arg1, T
  static void call_static(JavaValue* result, KlassHandle klass,Symbol* name, Symbol* signature, Handle arg1, H

  // 更低一层的接口，如上的一些函数可能会最终调用到如下这个函数
  static void call(JavaValue* result, methodHandle method, JavaCallArguments* args, TRAPS);
};
```

可以看出，JavaCalls::call()函数为虚拟机调用Java方法提供了便利。如上的函数都是自解释的，对应各自的invoke* 指令，Java虚拟机有invokestatic、invokedynamic、invokestatic、invokespecial、invokevirtual几种方法调用指令。这些call_static()、call_virtual()函数内部调用了call()函数。这一节我们先不介绍各个方法的具体实现。下一篇将详细介绍。

我们选一个重要的main()方法来查看具体的调用逻辑。如下基本照搬R大的内容，不过我略做了一些修改，如下：

假设我们的Java主类的类名为JavaMainClass，下面为了区分java launcher里C/C++的main()与Java层程序里的main()，把后者写作JavaMainClass.main()方法。
从刚进入C/C++的main()函数开始：

启动并调用HotSpot虚拟机的main()函数的线程：

```
main()
-> //... 做一些参数检查
-> //... 开启新线程作为main线程，让它从JavaMain()函数开始执行；该线程等待main线程执行结束
```

在如上线程中会启动另外一个线程执行JavaMain()函数，如下：

```
JavaMain()
-> //... 找到指定的JVM
-> //... 加载并初始化JVM
-> //... 根据Main-Class指定的类名加载JavaMainClass
-> //... 在JavaMainClass类里找到名为"main"的方法，签名为"([Ljava/lang/String;)V"，修饰符是public的静态方法
-> (*env)->CallStaticVoidMethod(env, mainClass, mainID, mainArgs); // 通过JNI调用JavaMainClass.main()方法
```

以上步骤都还在java launcher的控制下；当控制权转移到JavaMainClass.main()方法之后就没java launcher什么事了，等JavaMainClass.main()方法返回之后java launcher才接手过来清理和关闭JVM。

下面看一下调用Java主类main()函数时会经过的主要方法及执行的主要逻辑，如下：

```
// HotSpot VM里对JNI的CallStaticVoidMethod的实现。留意要传给Java方法的参数
// 以C的可变长度参数传入，这个函数将其收集打包为JNI_ArgumentPusherVaArg对象
-> jni_CallStaticVoidMethod()

    // 这里进一步将要传给Java的参数转换为JavaCallArguments对象传下去
    -> jni_invoke_static()

        // 真正底层实现的开始。这个方法只是层皮，把JavaCalls::call_helper()
        // 用os::os_exception_wrapper()包装起来，目的是设置HotSpot VM的C++层面的异常处理
        -> JavaCalls::call()

            -> JavaCalls::call_helper()
                -> //... 检查目标方法是否为空方法，是的话直接返回
                -> //... 检查目标方法是否“首次执行前就必须被编译”，是的话调用JIT编译器去编译目标方法
                -> //... 获取目标方法的解释模式入口from_interpreted_entry，下面将其称为entry_point
                -> //... 确保Java栈溢出检查机制正确启动
                -> //... 创建一个JavaCallWrapper，用于管理JNIHandleBlock的分配与释放，
                    // 以及在调用Java方法前后保存和恢复Java的frame pointer/stack pointer
```

```
//... StubRoutines::call_stub()返回一个指向call stub的函数指针，
// 紧接着调用这个call stub，传入前面获取的entry_point和要传给Java方法的参数等信息
-> StubRoutines::call_stub()(...)
   // call stub是在VM初始化时生成的。对应的代码在
   // StubGenerator::generate_call_stub()函数中
   -> //... 把相关寄存器的状态调整到解释器所需的状态
   -> //... 把要传给Java方法的参数从JavaCallArguments对象解包展开到解释模
      // 式calling convention所要求的位置
   -> //... 跳转到前面传入的entry_point，也就是目标方法的from_interpreted_entry


      -> //... 在-Xcomp模式下，实际跳入的是i2c adapter stub，将解释模式calling convention
         // 传入的参数挪到编译模式calling convention所要求的位置
            -> //... 跳转到目标方法被JIT编译后的代码里，也就是跳到 nmethod 的 VEP 所指向的位置
               -> //... 正式开始执行目标方法被JIT编译好的代码 <- 这里就是"main()方法的真正入口"
```

后面3个步骤是在编译执行的模式下，不过后续我们从解释执行开始研究，所以需要为虚拟机配置-Xint选项，有了这个选项后，Java主类的main()方法就会解释执行了。

在调用Java主类main()方法的过程中，我们看到了虚拟机是通过JavaCalls::call()函数来间接调用main()方法的，下一篇我们研究一下具体的调用逻辑。