项目介绍

先贴上项目的开源地址：

https://github.com/TangBean/OnlineExecutor

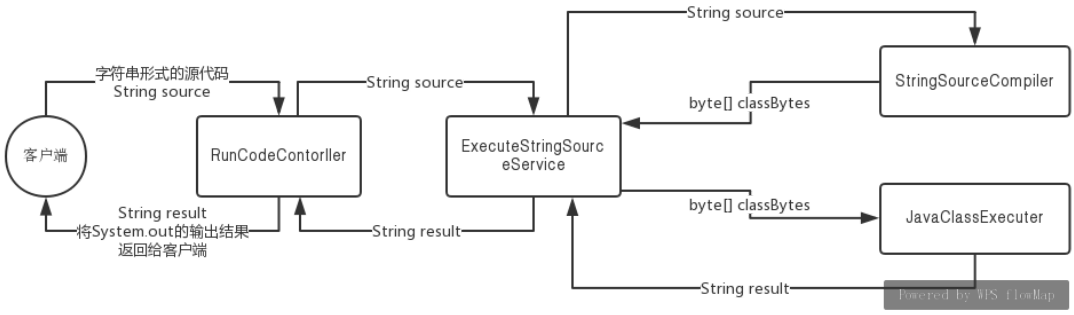
牛客上看到的一个很有意思的开源项目，里面涉及了很多Java的知识点，遂决定好好完成这个项目，一方面是复习，另一方面是把它能够写上简历。

大致的功能是：

- 在网页提交自己的代码可以在后台运行。
- 后台运行结束后，返回结果或者返回编译错误或者运行时异常。
- 可以导入库来使用一些类。
- 可以获取用户输入，但输入方式仅限于Scanner。

我在本次大作业的分工是后台功能的实现，但是为了测试正确性，我用springboot快速搭建了一个网页，通过网页运行来验证，但是springboot搭建过程不是本文章的重点，因此跳过介绍。

工作流程



模块介绍

SourceCompiler

进行用户代码的编译，输入为String字符串，输出为编译结果byte[]数组。

一般而言我们编译java代码都会用javac命令来编译生成class文件，这个class是与平台无关的字节码文件，之后jvm再解释执行字节码文件，因此我们最直接的思路就是：将用户代码写成.java文件，在编译得到.class文件，但这么做有两个坏处：

- IO耗时
- 生成文件污染服务器

因此这里用到的是java的动态编译的API：JavaCompiler，用到的方法为：

```
JavaCompiler.CompilationTask getTask(Writer out,
                                     JavaFileManager fileManager,
                                     DiagnosticListener<? super JavaFileObject>
diagnosticListener,
                                     Iterable<String> options,
                                     Iterable<String> classes,
                                     Iterable<? extends JavaFileObject>
compilationUnits)
```

其中，有两个参数需要注意：JavaFileManager和JavaFileObject

JavaFileObject用于封装用户代码和编译好的字节码，它需要提供的方法有：

- getCharContent，给compiler提供给源码的字符序列。
- openOutputStream，提供编译好的字节码的字节流。

JavaFileManager主要就是管理JavaFileObject，他要给compiler提供一个JavaFileObject来接编译好的字节码：

- getJavaFileForOutput,它会给compiler一个JavaFileObject来封装编译结果，同时方法重写中，我将这个JavaFileObject放进hashmap中，后续通过类名取得这个JavaFileObject来获取编译结果。

这个编译流程还是挺显而易见的，但是这里感觉可能会出现bug，就是我考虑了并发情况下，这个HashMap需要使用ConcurrentHashMap，但是我这里key-value中，key用的是客户定义的类的名字，有可能出现这样一种情况：

用户类同名，一个用户编译完成将结果存进ConcurrentHashMap后，还未去取，同时另外一个用户这时候也提交编译，这样前一个用户的编译结果会被覆盖，后面可能会出现一个用户运行的是别的用户的代码的情况。

所以考虑用synchronize进行代码同步，但这样的效率会降低，因为将这个代码块进行同步的话，同步粒度太大的，并行度降低。

HackSystem和HackScanner

获取了我们的编译结果之后，需要有几件事来处理：

- 用户的输入和输出如果调用System.out和System.in的话，会发生竞态，因为标准输入输出设备是整个虚拟机全局共享的资源，出现互相抢夺资源的情况。
- 存在一些方法可能比较危险，比如System.exit(),System.console(),System.setOut()等等方法对于服务器来说是危险的。

对于第一个问题，我们可以使用ThreadLocal来解决，通过构建ThreadLocal和ThreadLocal来给每个线程构建自己栈封闭的输入输出流。

对于第二个问题，可以自己实现自己的一个System来实现，因为java/lang/System是被final修饰的，不能被继承，因此我们自己实现的HackSystem要将所有方法都自己写一遍，大概的内容就是：

- 对于危险的方法直接抛出异常。
- 将输入输出流out、in、err由原来的PrintStream变为我们的HackPrintStream，InputStream变为我们的HackInputStream，HackPrintStream和HackInputStream里面的属性通过Threadlocal来封装线程独有的流。

HackInputStream和HackPrintStream

和上面分析一致，HackInputStream有自己的threadLocal属性：

```
private final static ThreadLocal<InputStream> holdInputStream=new ThreadLocal<>
();
```

HackInputStream继承InputStream需要重写read方法，但是在本次项目中，该方法没有实际作用，因此只用简单重写即可：

```
@Override
    public int read() throws IOException {
        return 0;
    }
```

另外需要实现自己的set和get方法：

```
   public InputStream get(){
        return holdInputStream.get();
    }

    public void set(String systemIn)
    {
        holdInputStream.set(new ByteArrayInputStream(systemIn.getBytes()));
    }
```

同时HackPrintStream也是一样：

```
    private ThreadLocal<ByteArrayOutputStream> out=new ThreadLocal<>();;
    private ThreadLocal<Boolean> trouble=new ThreadLocal<>();;
```

它的其他方法都要重新写一遍，原因在于之前直接在out上进行操作，现在需要out.get()在进行后续操作。

有了HackPrintStream和HackInputStream后，继续对HackScanner和HackSystem进行修改。

- HackScanner的构造函数由Scanner写为HackScanner，并且在其中一个构造函数上需要注意：

  ```
      public HackScanner(InputStream source) {
          // 判断一下输入的 InputStream 是不是 HackInputStream，
          // 如果是就调用一下它的 get 方法，把当前线程的输入流从 ThreadLocal 中取出来传
  入
          this(new InputStreamReader(
                          (source instanceof HackInputStream) ?
  ((HackInputStream) source).get() : source),
                  WHITESPACE_PATTERN);
      }
  ```

  这个构造方法很重要，因为我们获取用户输入的流程是这样的：

  - 用户传入字符串input。
  - 通过HackSystem.in.set（）设置input到线程独有的HackInputStream中。

    - 原本调用Scanner的方式为Scanner sc=new Scanner（System.in），经过字节码修改会变为（字节码修改这部分内容下面会讲到）HackScanner sc=new HackScanner（HackSystem.in），因此该构造器就是为这条代码做准备，能够将用户传入了我们的Scanner之中。
- 其他方法都可以照着写，除了返回类型从java.util.Scanner -> com.su.HackScanner。

另外HackSystem中要做的修改为：

- 标准输入输出替换成HackInputStream、HackOutputStream

```
    public final static InputStream in=new HackInputStream();
    public final static PrintStream out = new HackPrintStream();
    public final static PrintStream err=out;
```

- 将不安全的方法直接抛出异常。
- 将可以保留的方法比如arraycopy这些直接调用System原本的方法。
- 提供获取输出的接口，因为我们的输出结果还在流中，最后需要通过这个接口获取程序运行的输出。

```
public static String getBufferString()
    {
        return out.toString();
    }

    public static void closeBuffer()
    {
        ((HackInputStream)in).close();
        out.close();
    }
```

ClassModifiler和ByteUtils

上面System和Scannner的问题解决了以后，我们需要考虑这样一个问题，怎么把客户端对System的调用改为对HackSystem的调用呢？

最开始的想法就是，对用户代码的字符串进行修改，但是这种做法不优雅很粗糙，而且反复的字符串的替换，很繁琐。所以我们的做法是，在编译完成的.class文件中，将对System的符号引用改写为对HackSystem。

为了完成这一步，需要了解.class的文件结构，具体可参考

(7条消息) Class文件中的常量池详解（上） XINJing的专栏-CSDN博客

(7条消息) Class文件中的常量池详解（下） XINJing的专栏-CSDN博客

其中我们需要具备的知识就是.class的前8个字节中，前4个字节为魔数，用于验证class文件，后4个字节为.class的版本号，第九第十个字节为常量池项的数目的计数值，而需要注意的是，这个计数值是从1开始而不是从0开始。

从第十一个字节开始才为我们的常量池项，常量池项包括：

- 字面量：字符串、final常量值、基本类型。
- 符号引用：以符号描述引用的目标。

一共有14种常量池项类型，通过表中字段tag来进行标识，而我们需要关注的是：

- CONSTANT_Class_info
- CONSTANT_Utf8_info

CONSTANT_Class_info 的存储结构为：

```
... [ tag=7 ] [ name_index ] ...
... [  1位  ] [     2位    ] ...
```

其中，tag 是标志位，用来区分常量类型的，tag = 7 就表示接下来的这个表是一个 CONSTANT_Class_info，name_index 是一个索引值，指向常量池中的一个 CONSTANT_Utf8_info 类型的常量所在的索引值，CONSTANT_Utf8_info 类型常量一般被用来描述类的全限定名、方法名和字段名。它的存储结构如下：

```
... [ tag=1 ] [ 当前常量的长度 len ] [ 常量的符号引用的字符串值 ] ...
... [  1位  ] [        2位        ] [          len位          ] ...
```

在本项目中，我们需要修改的就是值为 `java/lang/System` 的 CONSTANT_Utf8_info 的常量，因为在类加载的解析阶段中，虚拟机会将常量池中的"符号引用"替换为"直接引用"，而 `java/lang/System` 就是用来寻找其方法的直接引用的关键所在，我们只要将 `java/lang/System` 修改为我们的类的全限定名，就可以在运行时将通过 `System.xxx` 运行的方法偷偷的替换为我们的方法。

而ClassModifiler的功能就是进行这一项工作，它的代码逻辑就是遍历每个常量池项，判断该项是否为CONSTANT_Utf8_info，不是的话跳过继续进行下一项，是的话判断该Utf8项是否表示 `java/lang/System` ，是的话获取com/su/util/HackSystem的字节数组表示，用该字节数组替换掉原字节数组的内容，完成后返回。

为了更好地完成这项工作，另外写了ByteUtils类来帮我们进行工作，ByteUtils就是对于字节数组进行一些操作，其中包括：byte2Int、int2Byte、byte2String、string2Byte、byteReplace等方法。

HotswapClassLoader

这里自定义类加载器有两个原因考虑：

- 首先如果我们用系统提供的应用程序类加载器去加载的话，，会有这样的坏处：

  我们客户端对源码修改后，反复提交，应用程序类加载器看类名没有改变，而且这个类已经加载过了，他就不会再对该类进行加载，而是将之前的类信息作为本次创建类的入口，这样使得客户端提交的新代码不能被执行。

  所以我们每次客户提交新代码时，都会创建一个新的自定义类加载器去进行加载，所以即使是同名的类，在不同加载器的加载下，这两个类被认为是不同的类。
- 另外，这里要考虑一个因素就是关于方法区中类的卸载，因为要知道我们客户端中的类，经过加载后，它的类信息是存放在服务端JVM的方法区，所以当用户的代码执行完后，这个类信息其实已经无用了，而方法区类信息的卸载有这么几个要求：
  - 不存在类实例
  - 不存在类的反射调用
  - 类加载器被卸载

前两个条件好满足，但是第三个条件，如果我们用应用程序类加载器去加载我们的用户类的话，而且这个应用程序类加载器是个全局属性一直不被回收的话，那么我们用户的类的信息就一直不能被卸载。所以我们实现一个自定义类加载器来进行加载，在用户类执行完毕后，将加载器引用置为null，以此来进行类的卸载。

```
public class HotSwapClassLoader extends ClassLoader{
    public HotSwapClassLoader(){
        super(HotSwapClassLoader.class.getClassLoader());
    }

    public Class loadByte(byte[] classBytes)
    {
        return defineClass(null,classBytes,0,classBytes.length);
    }
}
```

线程池设置

这里线程池有几个设置：

- 核心线程数和最大线程数都设置为3（或者更大），这么设置是因为我们的程序是一个CPU密集型任务，一般设置线程数为核数+1，避免频繁的线程切换导致吞吐量的降低，另外额外的一个线程是为了当有线程缺页或者阻塞时，能够更好的利用CPU资源。

  另外，保证并发度只有3，因为因为服务器的配置比较差，而且有在运行另一个项目。担心服务端方法区放不下这么多类信息导致OOM。
- 任务队列ArrayBlockQueue长度设置为5，阿里巴巴手册要求不能使用LinkBlockQueue，因为太多的任务会导致OOM。
- 超时回收时间设置为0s，其实这个参数没有用，因为我的核心线程数和最大线程数都为3，不会进行线程回收。
- 拒绝策略，用的是默认拒绝策略：放弃任务，抛出异常。

另外，我们用Callable+Future的方式来获取线程的执行结果，并且Future.get可以设置线程的执行时间，防止客户端的恶意死循环代码一直占用服务端资源。

工作流程：

- 输入用户代码和用户字符串。
- HackSystem.in.set来把用户的输入设置进threadlocal的输入流中。
- 使用SourceCompiler对用户代码进行编译。
- ClassModifiler对.class的字节码来进行替换（System和Scanner）。
- HotSwapClassLoader进行类的加载。
- 通过Class进行反射获取main方法。
- invoke执行main方法。
- 通过HackSystem.out来获取程序输出结果。
- 返回结果。