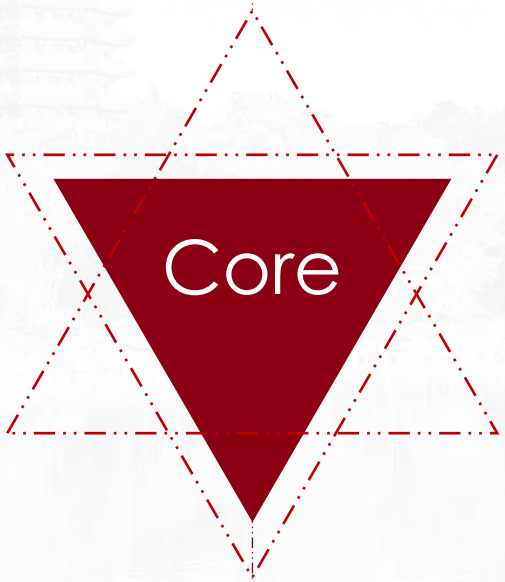
OnlineJudge

JavaProgrammingTeamwork

盛禹 何易峰 胡晓峰



- 主框架技术　SpringBoot
- 前端技术　Bootstrap
- 权限框架　SpringSecurity
- 持久层框架　SpringDataJPA
- 模版引擎　Thymeleaf

- Java 动态编译
- Java 类加载器
- Java 类的热替换
- Java 反射
- ThreadLocal 线程安全

登录 OJ

127.0.0.1:8080

前往注册页

Now Recording

登录

Username

Password

登录

01 *Authority Manage*
用户权限管理

权限管理

Security加密策略

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
   return new BCryptPasswordEncoder();
}
```

id	password	username
3	$2a$10$nbrhut5nSgNv/ENxr8bc9OMHHj5gc08rUCzNrHbPH8oAXOHGwMDW6	heyifeng

权限管理

Security用户权限管理核心

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeRequests()
            .antMatchers("/**", "/registration","/ide","/run","/question").permitAll()
            .anyRequest().authenticated()
            .and()
        .formLogin()
            .loginPage("/login")
            .defaultSuccessUrl("/question")
            .permitAll()
            .and()
        .logout()
            .permitAll();
}
```

02 *Dynamic compilation*
动态编译

动态编译

JDK 1.6 实现了动态编译

无动态编译

将 .java 文件翻译成二进制的字节码
将字节码存储在 .class 文件中
通过 ClassLoader 加载 .class 文件进内存获得对象

动态编译

Java 动态编译技术，跳过磁盘中 IO 操作生成两文件的过程
直接在内存中将源代码字符串编译为字节码的字节数组

动态编译

实现编译器

```
// 获取编译器对象
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
// 执行编译
Boolean result = compiler.getTask(null, manager, collector, options, null, Arrays.asList(javaFileObject)).call();
```

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

参数列表	含义
out	编译器一个额外的输出，为 null 的话就是 System.err
fileManager	文件管理器
diagnosticListener	诊断信息收集器
options	编译器的配置
classes	需要被 annotation processing 处理的类的类名
compilationUnits	需要被编译的单元，即 JavaFileObject

03 *Java Reflect* 反射机制

反射机制

类的加载与运行

通过该字节码数组将目标类的 main 方法运行起来，该过程可以分为以下 2 步：

- 类的加载：通过类加载器将字节码加载为 Class 对象；
- 类的运行：通过反射调用 Class 对象的 main 方法。

反射机制

重写类加载器

系统加载目标类的时候 使用唯一的应用程序类加载器

使用系统的类加载器加载会产生如下的问题：
如果通过该类加载器加载了目标字节码，
当客户端对源码进行了修改，
再次提交运行时，
应用程序类加载器会认为这个类已经加载过了，
不会再次加载它

```
package com.shengyu.oj.execute;

public class HotSwapClassLoader extends ClassLoader {

    public HotSwapClassLoader() {
        super(HotSwapClassLoader.class.getClassLoader());
    }

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

反射机制

动态获取信息以及动态调用对象方法

JAVA反射机制是在运行状态中：
对任意一个类，获得其所有属性和方法
对于任意一个对象，能调用其任意方法和属性

见右边案例

```
public class Phone {
    private int price;
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }

    public static void main(String[] args) throws Exception{
        //正常的调用
        Phone phone = new Phone();
        phone.setPrice(5000);
        System.out.println("Phone Price:" + phone.getPrice());
        //使用反射调用
        Class clz = Class.forName("com.xxp.api.Phone");
        Method setPriceMethod = clz.getMethod("setPrice", int.class);
        Constructor phoneConstructor = clz.getConstructor();
        Object phoneObj = phoneConstructor.newInstance();
        setPriceMethod.invoke(phoneObj, 6000);
        Method getPriceMethod = clz.getMethod("getPrice");
        System.out.println("Phone Price:" + getPriceMethod.invoke(phoneObj));
    }
}
```

反射机制

OJ中反射运行main方法

将类加载进虚拟机之后
可以通过反射机制来运行该类的 main 方法

```
// 通过反射调用Class对象的main方法
try {
    Method mainMethod = clazz.getMethod("main", new Class[]
{ String[].class });
    mainMethod.invoke(null, new String[] { null });
} catch (NoSuchMethodException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
} catch (InvocationTargetException e) {
    e.getCause().printStackTrace(HackSystem.err);
}
```

04 *System Safety*
系统安全与多线程

系统安全与多线程

三个风险问题

- 允许客户端程序随便调用 System 的方法还存在着安全隐患
- 新的系统调用类存在多线程安全问题
- 标准输出设备是整个虚拟机进程全局共享的资源，在多线程的情况下可能会将其它线程的结果也收集到一起

风险问题实现思路：

1.把要执行的类对 System 的符号引用替换为重写的System类的符号引用

2.重写标准输入输出流

3.使用ThreadLocal来解决线程安全问题

系统安全与多线程

重写System类

仿造 System 的写法，对 out 和 err 两个字段的实际类型进行修改，修改为我们自己写的 HackPrintStream 对象：

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

新加两个方法，用来获取当前线程的输出流中的内容和关闭当前线程的输出流：

```
public static String getBufferString() { return out.toString(); }
public static void closeBuffer() { ((HackInputStream) in).close();out.close(); }
```

比较危险的方法，设置禁止客户端调用，客户端一旦调用类这些方法就会抛出异常：

```
public static void exit(int status) { throw new SecurityException(“Use hazardous method: System.exit().”); }
```

系统安全与多线程

重写输入输出类

新的输入输出类继承 PrintStream 类并重写 PrintStream 的所有公有方法

使用 ThreadLocal 来实现线程的封闭

```
private ThreadLocal<ByteArrayOutputStream> out;
private ThreadLocal<Boolean> trouble;
```

重写父类 PrintStream 中所有对流进行操作的方法，比如下列：

```
public void write(byte buf[], int off, int len) {
    try {
        ensureOpen();
        out.get().write(buf, off, len); // out.get()才是当前线程的OutputStream
    }
    catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    }
    catch (IOException x) {
        trouble.set(true);
    }
}
```

请大家批评指正！

Java Programming Teamwork

第22组