

学习报告

1.VIM:

vim 是 linux 平台上一个比较流行的编辑器，具有语法高亮、编译并运行程序、编辑文档等等功能。安装 vim 时，最好安装完整版本：在命令行下格式为：`sudo apt-get install vim-nox`，这样功能比较完整。

安装完 vim 后，我们需对 vim 进行配置，以保证 vim 的功能更加完整和强大。要配置 vim，主要就是对主目录下的隐藏文件 .vimrc 进行编辑，把我们所需要的功能加进去。如：`set nu` 是显示行数，`syntax on` 是语法高亮，等等；同时，我们还可以设置自动补存括号、引号等等，还能想 VC++ 那样按 F5 进行编译程序；

配置完 vim 后，我们就可以利用 vim 进行代码编写了。在命令行下输入：`vim 文件名`，然后在按下字母 i，界面左下方就会出现 INSERT 的提示，就可以对该文档进行编辑了；要对文档进行保存，则需先按下 ESC 键，再按下 `:`，然后输入 `w`（小写）；退出则为 `q`，保存并退出为 `wq`；不过，有时候我们会对系统文件进行修改，如配置 java 虚拟机时要对 `/etc/profile` 进行编辑，此时则需要提供 root 权限，所以我们在 vim 前要加上 `sudo`，完整格式为 `sudo vim 文件名`，否则无法保存编辑后的结果。

另外，vim 还有很多其他很强大的功能，比如：在一个界面上分屏同时修改多个文档，移动光标，删除、复制多行，查找等等。比如：如果我们要删除整行文字，则只需在命令行模式下把光

标移动到该行，然后按下 **dd** 即可。同时，**vim** 默认在修改完一个文件后会生成一个修改之前的备份文档，以便我们想要回滚到修改之前的状态；

2.JAVA:

JAVA 主要分为三部分：**J2SE**（桌面应用程序开发），**J2ME**（嵌入式软件开发、手机软件开发），**J2EE**（网络编程，**JSP** 等等）。

JAVA 作为一种跨平台的语言，可以运行在任何机器上，只需该机器上装有对应的 **JAVA** 虚拟机就可以了。**JAVA** 的跨平台性主要是因为是在编译时，**JAVA** 源程序会被编译成一种与机器无关的字节码格式文件（后缀为.class）；这种文件不是可执行文件，但是可以被 **JAVA** 虚拟机翻译成指令，从而被执行；

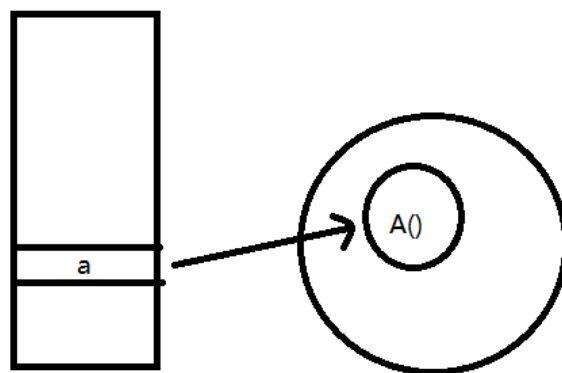
所以，如果我们要运行 **JAVA** 程序，只需安装好对应系统平台的 **JAVA** 虚拟机，并设置好环境配置就可以了。

```
32 #set the environment of JAVA
33 export JAVA_HOME=/usr/java
34 export JRE_HOME=/usr/java/jre
35 export CLASSPATH=.:$CLASSPATH:$JAVA_HOME/lib:$JRE_HOME/lib:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar:$JAVA_HOME/lib/junit-4.9.jar:$JAVA_HOME/lib/gridworld.jar
36 export PATH=$PATH:$JAVA_HOME/bin:$JRE_HOME/bin
```

JAVA 跟 **C++** 一样，也是一门面向对象的语言，不过 **JAVA** 的面向对象性更强，以至于所有 **JAVA** 变量和方法都是封装在类内部了。不过，从语法上来说，**JAVA** 和 **C++** 几乎一模一样。例如：在判断语句 **if** 和 **switch**，循环语句 **for** 等等一些基本的语法上，

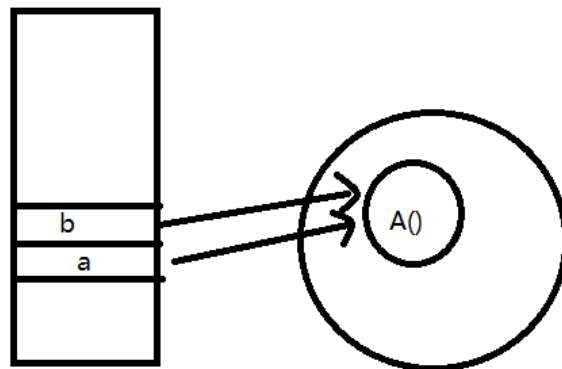
JAVA 和 C++都是一样；同时，这两者在变量的定义方式，代码注释方式，语法格式等方面都是一样。不过，JAVA 和 C++也有很多不同的地方。比如：在 C++中可以用来自 C 语言的指针，但 JAVA 中为了安全性而抛弃了指针。不过 JAVA 中的变量分为引用类型和值类型，其中引用类型就有点类似于指针，可以用多个变量名指向同一块内存区域；

JAVA 中的所有引用类型都是通过 new 出来的，存放在堆中。当 new 一个对象时，系统就会为该对象分配两块内存，一块在堆中，用来存放要存放的数据，另一块在栈中，用来存放指向堆中内存的地址；例如：A a = new A();，则 new 出来的 A 对象会保存在堆中，然后 a 是保存着 new 出来的对象在堆中的地址，a 存放于栈中。



所以，如果我们要将多个变量名指向堆中同一块内存的

话，只需直接将变量名赋值即可，如：`b = a;`



同时，**JAVA** 中还没有内存管理，所以 **JAVA** 中的类不像 **C++** 那样有析构函数。不过当一块内存不再有变量指向它时，这块内存就会被自动释放掉，这样可以避免内存泄漏等问题，保证程序的安全性。

由于 **JAVA** 中所有的变量和方法都是封装在类中，所以程序执行的入口也是放在类中。那么，当一份源代码声明和定义了多个类，我们如何决定哪个类是启动类呢？和 **C++** 一样，**JAVA** 程序的执行入口也是 `main` 函数；所以，要决定哪个类为启动类，我们只要将 `main` 函数放在那个类中就可以了。并且，启动类的前面最好加上 `public` 权限修饰符，同时建议启动类的类名要和该源程序的文件名相同，这样有利于程序的编译和执行；众所周知，在 **C++** 中，我们除了可以用 `true` 来表示真，用 `false` 来表示假，还可以用非零来表示

真，用零来表示假。但这种方法在 JAVA 中是禁止的。在 JAVA 中，我们只能用 `true` 来表示真，用 `false` 来表示假，而不能用其他的数据类型来表示真假；

既然 JAVA 都是对由象类构成的，那样必然也会像 C++那样有封装和继承；JAVA 中的封装修饰符也有 `public`、`private` 和 `protected`，还有一个 `default`。不过修饰的权限跟 C++有些不同。在 JAVA 中，`public`、`protected`、`default` 修饰的属性和方法都可以在类的外部访问，只有 `private` 的跟 C++一样；这些修饰符的真正作用在包与包之间访问时就会体现出来了。

在 JAVA 中，要继承一个类，则用以下语法格式：`class son extends father`；可是与 C++不同的一个地方是，在 C++中一个子类可以继承多个父类，而 JAVA 中一个类只能继承一个父类，这样就可以避免多重继承带来的混乱。同时，JAVA 中还有两个关键字：`abstract` 和 `final`；`abstract` 可以用来修饰一个方法或类；当一个类被 `abstract` 修饰时，表示该类是抽象类，不能用于生成对象。而 `final` 也可以用来修饰整个类或类中的部分成员；当一个类属性被 `final` 修饰时，则该属性相当于 C++中的常量，一旦赋值就不能被修改；`final` 修饰类方法，则该方法只能被子类继承，但不能被子类重写；当 `final` 修饰整个类时，则这个类就不能被其他类继承了，因为它是 `finally`（最后一个了）。

```

1
2 class Father
3 {
4     protected int i;
5 }
6
7 class Son extends Father
8 {
9     protected int j;
10 }

```

除了继承父类之外，JAVA 中还有一样东西是 C++ 中没有的，那就是 interface（接口）；

```

1 interface InTmp
2 {
3     public void sayHello();
4     int i = 10;
5 }

```

JAVA 可以定义一个接口，接口里面有些未实现的方法。然后，这些接口就可以被其他的类实现了，并且接口中的属性默认是 public、static、final 的，方法默认是 public、abstract 的：

```

2 interface InTmp
3 {
4     public void sayHello();
5 }
6
7 abstract class A implements InTmp
8 {
9     public void sayHello()
10    {
11        System.out.println("Hello World");
12    }
13 }

```

与继承不同的是，一个类可以实现多个接口；但如果一个类既要继承父类又要实现接口的话，继承要放在实现的前面：

```
2 interface InTmp
3 {
4     public void sayHello();
5 }
6 class A
7 {
8     public void sayGoodBye()
9     {
10         System.out.println("Goodbye!");
11     }
12 }
13
14 abstract class B extends A implements InTmp
15 {
16     public void sayHello()
17     {
18         System.out.println("Hello, World!");
19     }
20 }
```

public, protected, d
修饰符号的真正作用在
在 JAVA 中，
一个地方是，在 C++ 中
避免多重继承带来的混
一个类
修饰整个类或类中的部
就不能被修改；final
则这个类就不能被其他
除了继承父类之

JAVA 可以定义

JAVA 中也有多态。在 JAVA 中，我们可以将子类赋值给父类，但不能将父类赋值给子类。当子类赋值给父类时，父类只能调用与子类共同的方法和属性，而不能调用子类特有的方法和属性；如果父类要调用子类特有的方法，则可通过强制类型转化将父类转化成子类，再调用子类的方法：

```

2 class A
3 {
4     public void showA()
5     {
6         System.out.println("AAAAA");
7     }
8 }
9
10 class B extends A
11 {
12     public void showB()
13     {
14         System.out.println("BBBBB");
15     }
16 }
17
18 public class Tmp
19 {
20     public static void main()
21     {
22         A a = new A();
23         B b = new B();
24
25         a = b;
26         a.showB(); //Error
27
28         B b1 = (B)a; //强制类型转换
29         b1.showB(); //OK
30     }
31 }

```

与 C++不同的是，JAVA 可以在一个类的内部定义另一个类，也就是内部类：

```

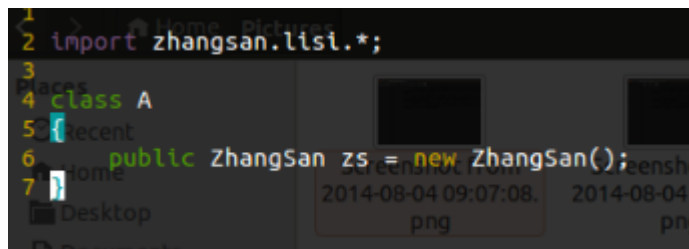
2 class A
3 {
4     String str = "Hello!";
5     class B
6     {
7         public void sayHello()
8         {
9             System.out.println(str);
10        }
11    }
12 }
13

```

内部类可以访问外部类的所有方法和属性（有点类似 C++ 的友元）

JAVA 还跟 C++ 有一个很大的不同之处，那就是 jar 包。jar 包就类似于 C++ 的 #include 后面库函数。我们可以将我们自己写的

源程序打包成一个 jar 包，然后就可以在其他地方 import 这个 jar 包，再调用里面的类和其方法了。

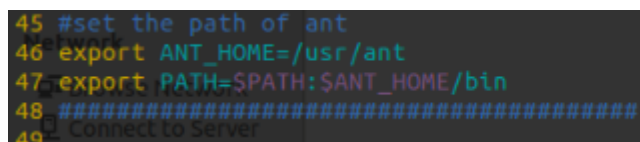


```
1
2 import zhangsan.list.*;
3
4 class A
5 {
6     public ZhangSan zs = new ZhangSan();
7 }
```

3.ANT:

Ant 作为一个构件工具，可以实现项目的自动构建和部署等功能。在这次实训中，我们要用 Ant 来编译和运行我们编写的 java 源程序，进行一些小的任务等等；

要安装 Ant，我们只需将 Ant 的安装包下载下来，解压，然后在配置好环境变量就可以使用了：



```
45 #set the path of ant
46 export ANT_HOME=/usr/ant
47 export PATH=$PATH:$ANT_HOME/bin
48 #####
49
```

Ant 的源程序写在 build.xml 文件中。在这个文件中，有一个 project，然后 project 下可以定义多个我们要进行的小任务<target>，每个任务以不同的任务名来区分。在不同任务之间，我们可以设置任务之间的关系，例如某些任务的执行需要另一些任务的执行结果等等，这要用到任务之间的依赖关系 depends。

```

1 <?xml version="1.0"?>
2 <project name="Hello">
3   <target name="A">
4     <echo message="AAAA" />
5   </target>
6   <target name="B" depends="A">
7     <echo message="BBBB" />
8   </target>
9
10 </project>
11
12

```

每一个 project 都要有一个启动任务，即 default，当我们在命令行下执行 ant 时，就会自动从启动任务入手，再根据任务间的依赖关系来依次执行任务。如果没有指定启动任务，则需在 ant 后面来执行要运行哪个任务，否则系统不知道该首先执行哪个任务。

```

hadoop@Master:~/Code$ ant B
Buildfile: /home/hadoop/Code/build.xml

A:
    [echo] AAAA

B:
    [echo] BBBB

BUILD SUCCESSFUL
Total time: 0 seconds

```

同时，在 project 中我们还可以指定 basedir，不过当我们要表示主目录时，不能用～，而必须用/home/..这样来表示，否则会报错。

```
hadoop@Master:~/Code$ ant hello
Buildfile: /home/hadoop/Code/build.xml

BUILD FAILED
Basedir /home/hadoop/Code/~/Downloads does not exist.

Total time: 0 seconds
hadoop@Master:~/Code$ vim build.xml
hadoop@Master:~/Code$ ant hello
Buildfile: /home/hadoop/Code/build.xml

hello:
    [echo] /home/hadoop/Downloads

BUILD SUCCESSFUL
Total time: 0 seconds
hadoop@Master:~/Code$
```

我们还可以用 ant 来编译和运行我们的 java 程序；

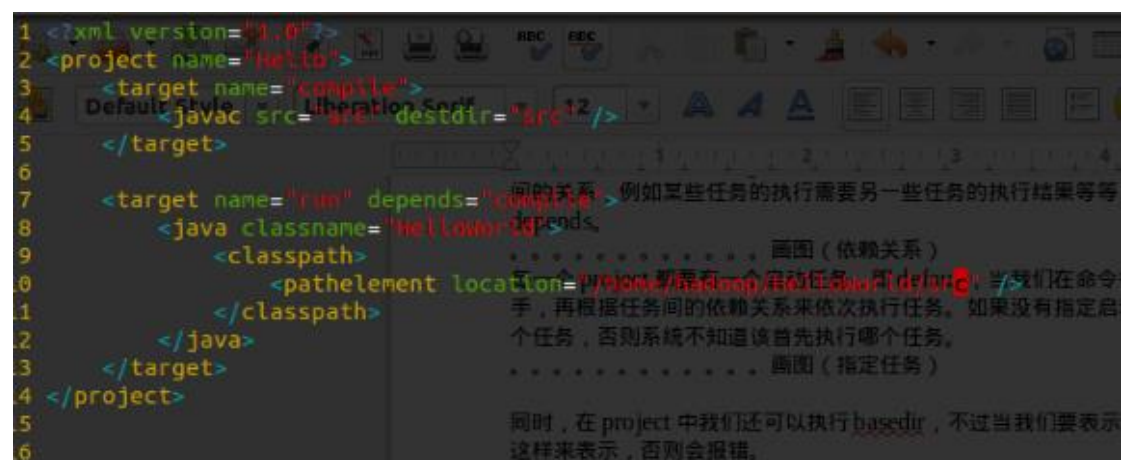
要编译 java 程序，我们需用到 javac，同时要指定我们的源码所在路径，和编译生成的 class 文件存放路径。

```
1 <?xml version="1.0"?>
2 <project name="Hello">
3   <target name="compile">
4     <javac src="src" destdir="src"/>
5   </target>
6
7 </project>
```

要运行 java 程序，则要用到 java，并指定启动类名，和启动类所在路径就可以了。

```
1 <?xml version="1.0"?>
2 <project name="Hello">
3   <target name="compile">
4     <javac src="src" destdir="src"/>
5   </target>
6
7   <target name="run" depends="compile">
8     <java classname="HelloWorld">
9       <classpath>
10        <pathelement path="src"/>
11      </classpath>
12    </java>
13  </target>
14 </project>
```

需要注意的是，这里 `pathement` 中的 `path` 路径是指相对路径；如果要用绝对路径则可以用 `location`：



4.Junit:

Junit 主要是用来对我们编写的代码进行部分测试，并将测试结果返回给我们。这样，我们程序员就可以根据返回的测试结果来判断我们的程序（或某个函数）是否运行正确了，而避免了我们要对整个程序进行测试的麻烦。

要使用 Junit，首先我们要将 Junit 的 jar 包下载下来，就可以用了。

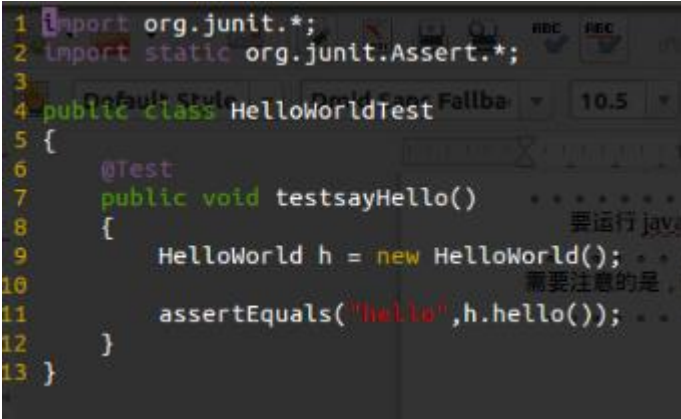
当然，如果我们要在命令行下利用 Junit 进行测试，我们还要将 Junit 的 jar 包导入：



要对某个类（假设为 A）中的方法进行测试，我们首先要写一个测试类，然后在测试类中生成类 A，这样我们就可以在测试类内

部对 A 中某些方法进行测试了。

要写测试类，我们要将 junit 中的各种 jar 包装载进来，并且每个测试方法前要加上 `@Test`，以告诉编译器这是个测试方法。同时测试方法名还要以 `test` 开头；



```
1 import org.junit.*;
2 import static org.junit.Assert.*;
3
4 public class HelloWorldTest
5 {
6     @Test
7     public void testsayHello()
8     {
9         HelloWorld h = new HelloWorld();
10        assertEquals("hello", h.hello());
11    }
12 }
13 }
```

The image shows a code editor with a dark theme. It displays a Java test class named `HelloWorldTest`. The code includes imports for `org.junit.*` and `org.junit.Assert.*`. The test class contains a single test method `testsayHello()` annotated with `@Test`. Inside the method, a `HelloWorld` object is instantiated, and `assertEquals` is used to verify that the `hello()` method returns the string "hello". The code is numbered from 1 to 13. On the right side of the editor, there are some UI elements like a search bar and a dropdown menu.

测试方法的主要工作原理是调用要测试类 A 中的带返回值的某个方法，然后再用 `assertEquals` 来判断方法返回值与期望值是否相同；如果不同，则说明该方法肯定错误了。

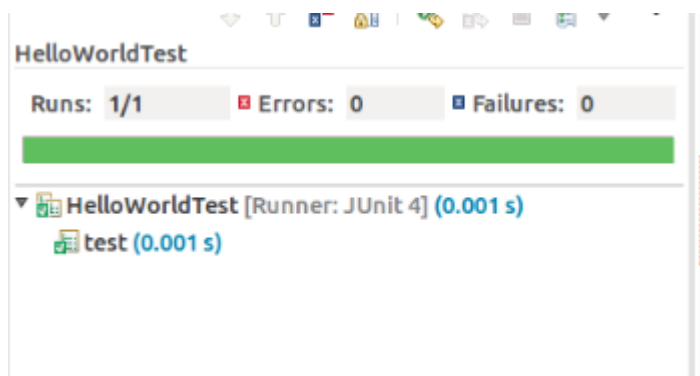
同时，为了使得测试前有一个统一的环境和测试后返回有个统一出口，我们可以用 `@Before` 和 `@After`，在一个方法前加上 `@Before` 表示每次测试之前都会调用这个方法，而 `@After` 则表示每次测试之后都会调用这个方法；

```

1 import org.junit.*;
2 import static org.junit.Assert.*;
3
4 public class HelloWorldTest
5 {
6     @Before
7     public void setUp()
8     {
9         System.out.println("Hello World!");
10    }
11
12    @After
13    public void leave()
14    {
15        System.out.println("Goodbye!");
16    }
17
18    @Test
19    public void testSayHello()
20    {
21        HelloWorld h = new HelloWorld();
22        assertEquals("hello", h.hello());
23    }
24 }

```

除了在命令行下测试外，我们还可以利用 Eclipse 这个软件来进行测试，这样会比较简单，直接运行就可以了：



5.Sonar

Sonar 是一个用于提升代码质量的平台；

当我们书写一段代码时，我们要注意到很多方面，比如：代码风格，注释风格，注释的多少，代码是否冗余等等。而这些往往是

我们程序员比较难以发现和改正的，因为这与我们平时写代码的习惯有关。但是，这些方面又会对我们的代码质量产生非常大的影响。这时，我们就可以通过 Sonar 来检测我们的代码，以帮助我们提升代码的质量。

要安装 sonar，首先我们要下载 sonar 和 sonar-runner 的压缩包，然后解压到某一路径下。解压完后，我们再设置下环境变量就可以了。

```
50 #set the path of sonar
51 export SONAR_HOME=/usr/local/sonar-3.7.4/bin/linux-x86-64
52 export SONAR_RUNNER_HOME=/usr/local/sonar-runner-2.4
53 export PATH=$PATH:$SONAR_RUNNER_HOME/bin
54 #####
```

安装好后，启动 sonar，我们就可以用 sonar 来检测我们的代码了。

```
hadoop@Master:~$ cd $SONAR_HOME
hadoop@Master:/usr/local/sonar-3.7.4/bin/linux-x86-64$ ./sonar.sh restart
Stopping sonar...
Stopped sonar.
Starting sonar...
Started sonar.
```

首先，我们要创建一个配置文件 sonar-project.properties，通过这个配置文件，我们来指定要检测的代码等等。

```
1 #required metadata
2 #projectKey是项目的唯一标识，不能重复
3 #对于每一个part，我们只需修改projectKey, projectName, projectBaseDir这三项
4
5 sonar.projectKey=HelloWorld
6 sonar.projectName=HelloWorld
7 sonar.projectVersion=1.0
8 sonar.sourceEncoding=UTF-8
9 sonar.modules=java-module
10
11 #java-module
12 java-module.sonar.projectName=Java Module
13 java-module.sonar.language=java
14
15 #.表示projectBaseDir指定的目录
16 java-module.sonar.sources=
17 java-module.sonar.projectBaseDir=
18
```

配置好 sonar-project.properties 后，我们就在命令行下输入

sonar-runner 来检测代码，

```
14:59:31.702 INFO - Sensor VersionEventsSensor done: 4 ms
14:59:31.703 INFO - Sensor CpdSensor...
14:59:31.703 INFO - SonarEngine is used
14:59:31.703 INFO - Sensor CpdSensor done: 0 ms
14:59:31.703 INFO - Sensor JaCoCoSensor... 画图 (启动 sonar)
14:59:31.703 INFO - Project coverage is set to 0% since there is no directories with classes. 我们指定要检测的代码
14:59:31.703 INFO - Sensor JaCoCoSensor done: 0 ms
14:59:31.858 INFO - Execute decorators... 画图 (sonar-project)
14:59:31.991 INFO - Store results in database
14:59:32.095 INFO - ANALYSIS SUCCESSFUL, you can browse http://localhost:9000/dashboard/index/HelloWorld
14:59:32.244 INFO - Executing post-job class org.sonar.plugins.dbcleaner.ProjectPurgePostJob
14:59:32.264 INFO - -> Keep one snapshot per day between 2014-07-07 and 2014-08-03
14:59:32.266 INFO - -> Delete snapshot: 2014-08-03T22:31:39+0800 [6]
14:59:32.405 INFO - -> Keep one snapshot per week between 2013-08-05 and 2014-07-07
14:59:32.405 INFO - -> Keep one snapshot per month between 2009-08-10 and 2013-08-05
14:59:32.405 INFO - -> Delete data prior to: 2009-08-10
14:59:32.426 INFO - -> Clean HelloWorld [id=1]
14:59:32.430 INFO - -> Clean Java Module [id=2]
14:59:32.441 INFO - Executing post-job class org.sonar.plugins.core.issue.notification.SendIssueNotificationsPostJob
14:59:32.447 INFO - Executing post-job class org.sonar.plugins.core.batch.IndexProjectPostJob
INFO: -----
INFO: EXECUTION SUCCESS
INFO: -----
Total time: 11.720s
Final Memory: 11M/163M
INFO: -----
```

然后在浏览器中输入 <http://localhost:9000>，来查看检测结果。

