# 第一章 理解java

`Hello.java`

```
1   public class Hello{
2
3       public static void main(String[] args){
4           System.out.println("hello world");
5       }
6
7   }
```

## 1 问题

**以上面代码为例，展开以下问题的讨论：**

**问题一：**

Hello是我们自己定义的一个类，为什么写java代码的时候要写一个类？

因为在java中，类是组织代码的基本的单元，在类中，可以编写**方法**，在方法里面，才是真正需要JVM去执行的一行行的**代码**，我们平时所说的代码执行，就是这的这些**方法中**的**代码**。

java语言中规定，在大多数情况下，需要把执行的**代码**写在**方法**中，方法写到**类**中，所以在写代码的时候，一般都是先定义一个类，类中定义一个方法，然后把要**执行的代码**写到方法中。

那么这里所说的**类中**和**方法中**，具体的表现形式，就是一对花括号{}，如果这对花括号是属于类的，那么括号里面就表示**类中**，如果这对花括号属于**方法的**，那么括号里面就表示方法中。

例如：

```
1   public class Student{
2
3       //这里是就是类中
4
5   }
6   //--------------------
7   public class Student{
8
9       public void run(){
10          //这里就是方法中，当然这个方法整体也在类中
11      }
12
13  }
14
```

**问题二：**

Hello类中的方法，为什么名字要叫做main？

main方法是java中一个特殊的方法，它是作为java程序的唯一入口而存在的。

例如，一个项目中写了好几个百个类，每个类中的方法加起来一共又有上千个方法，那么当JVM加载了这么多个类和方法的时候，JVM应该从什么地方开始运行程序呢？

java中就使用了一个固定的方法来作为程序的入口，也就是无论写了多少个类，多少个方法，JVM一定是从这个固定的**程序入口方法**开始执行代码的，为了能够让JVM很好的识别这个入口方法，这个方法的编写形式就是**固定**的：

```
//这是java中的程序入口方法，一切代码的执行，都要从这里开始
public static void main(String[] args){

    //方法中，就可以编写一行行需要运行的代码了

}
```

所以，我们在Hello类中定义一个main方法的目的就是，让JVM运行Hello这个类的时候，可以直接从类中的程序入口方法，也就是main开始运行代码，又由于main方法是固定的写法，所以JVM很容易就识别出来，然后去运行它，main方法被运行了，那么main中的代码就会被一行行的顺序执行了。

> 思考，如果一个类中没有编写固定形式的main方法，那么使用java命令去运行这个类，会发生什么？为什么？在这情况下，如果还想使用这个Hello类，该怎么办？

1. 如果Hello中没有固定格式的main方法，使用 `java Hello` 运行的时候，会报错
2. 原因是因为JVM在Hello类中找不到指定的程序入口，也就是main方法
3. 这个情况下， 还要使用Hello这个的话，就需要在定义其他的一个类，例如Test类，在Test类定义程序入口main，并且在main方法中调用Hello类中的属性或其他方法，这时候就使用到了Hello类中的代码了

**问题三：**

`System.out.println("hello world")，这句代码具体是什么意思？`

System是JavaSE-API中所提供的一个类，

out是System类中的一个属性，

println是out中的一个方法，

所以，System.out是访问System类中的一个属性，System.out.println是调用out中的一个方法，
而println("hello world")方法的作用，是将字符串"hello world"输出到命令行或控制台中。

> 可以在API说明文档中，找到这些相关的说明

**问题四：**

`在Hello中，我们为什么可以使用System这个类？`

在自己的类中，想使用别人/自己提前写的另一个类，需要以下要求：
这里就以Hello类中使用System类的代码为例说明：

1. System这个类所在的.java文件已经编写完成，并且已经编译成.class文件
   **System.java文件在src.zip中，编译好的System.class文件在rt.jar中**
2. 这个.class文件所在位置，是JVM可以自动加载的指定路径，这样就可以保证JVM可以把这个.class文件中的字节码（也就是System这个类的代码），加载到JVM的内存中
   **JVM在启动的时候，会自动读取rt.jar中所需要的class文件，当然也包括System.class文件**
3. 在Hello中，使用**import**（导入）关键字，将要使用的System类，引入到Hello中，如果System类是在java.lang包中进行定义的，或者System类和Hello类是在同一个包下面定义的，那么import语句可以省去不写

> 当前Hello类中，是没有声明package（包）的，那么这时候Hello类就算是在**默认包**中，而System类，是在java.lang包下定义的，所以当前情况下， Hello类中不使用import导入，也可以直接使用System类。

**问题五：**

使用java Hello运行这个类的时候，JVM是通过什么找到Hello.class文件的？

通过CLASSPATH环境变量配置的路径，来查找的Hello.class文件的，如果配置的路径不对，那么运行Hello的时候就会报错，告诉我们找不到这个类。

> 注意，我们之前配置JDK的三个环境变量的时候，把CLASSPATH配置的路径就是点（.），表示当前路径。

**问题六：**

能否看到JVM去启动运行Hello类之前，确实从rt.jar中加载到了System这个类？

可以，只需要在运行Hello类的时候，加一个参数即可：

```
java -verbose Hello
```

verbose参数可以将JVM启动运行的时候加载的信息输出出来，由于内容太多，这里可以做一个输出重定向：

```
java -verbose Hello > a.txt
```

> 注意，这个操作在Windows里面也是一样的

通过文件中记录的输出内容，就可以看到JVM在运行Hello这个类之前，整个加载的过程和顺序。

注意，这里JVM其实就是给我们自己编写的类Hello，准备运行的环境。

rt.jar中rt就是runtime的缩写，表示运行时环境的意思。

**问题七：**

Hello这个类的名字和Hello.java的名字有什么必然关系么？

1. 如果文件中的类是public关键字修饰的，那么这个类的名字和java文件的名字就一定要一样
2. 如果文件中的类不是public关键字修饰的，那么这个类的名字和java文件的名字不一样也可以
3. 类名和java文件名字的首字母大写，这是编程规范，大家都是默认遵循这个要求，其实这个字母小写也没有任何问题

> 所以，Hello这个类不一定在Hello.java中，但是成功后，Hello这个类一定在Hello.class中

## 2 pacakge

> 在java中，定义包的关键字是package

在程序中，要区分一些东西，一般会采用【命名空间】的设计方式，这是大多数语言都会采用的方式。

在java中，如果来区分俩个名字一样的类？例如，张三定义了一个类Hello，李四定义了一个类Hello，当把张三和李四的代码合并一起的时候，会出现俩个都叫Hello的类，那么这个时候该如果区分这个类？

可以使用package（包）来进行区分，例如张三定义的Hello这个类可以放在zhangsan这个名字的包下，李四定义的Hello这个类可以放在lisi这个包下，如下：

```
package zhangsan;
public class Hello{

}
```

```
package lisi;
public class Hello{

}
```

注意，在程序中，`package zhangsan;`对应的是一个名字叫zhangsan的文件夹，而`package lisi;`对应的是一个名字叫lisi的文件夹，这样俩个代码合并在一起，也完全可以区分开名字相同的俩个Hello类。

但是一般程序中，定义包的时候，不会直接用zhangsan、lisi这样的名字，而是都会遵从一些包的命名规则的：

1. package其实就是类的**命名空间**，用来唯一标识这个类的，避免和的类的名字重复
2. 一般情况，一个公司、组织、社团中所定义的包的名字，就是他们官网的域名（倒过来），因为域名一定是全球唯一的，不可能有俩个一样的域名。
   例如，http://commons.apache.org/ 这官网下的代码中的包，都是`package org.apache.commons;`开头的。
   例如，我们公司的代码中的包，都是以`package com.briup;`开头的
   例如，你个人写的代码，可以是以`package com.jim;`开头的，假设你的名字叫 jim
3. 类加上了包名，编译之后的效果
   这样的类，在编译之后，都必须要有和包名对应的文件夹。
   例如，`package com.briup.demo;`

> 这里是三个包，包和包之间用点（.）隔开，编译完之后，需要有对应的三个文件夹分别是com/briup/demo ,最后在demo目录中，才有编译生产的class文件

一个指定package的类，编译后该如何运行？

例如：

```
1  package com.briup.test;
2
3  public class Hello{
4
5      public static void main(String[] args){
6          System.out.println("hello world");
7      }
8
9  }
```

编译代码：

`javac Hello.java`

```
briup@briup:~/code/day01$ javac Hello.java
briup@briup:~/code/day01$ ls
Hello.class  Hello.java
briup@briup:~/code/day01$
```

编译通过，运行代码：

`java Hello`

```
briup@briup:~/code/day01$ java Hello
错误: 找不到或无法加载主类 Hello
briup@briup:~/code/day01$ echo $CLASSPATH
.
briup@briup:~/code/day01$
```

> 可以看到，运行报错，但是classpath的值也没有问题

**这个错误的原因是，Hello这个类是定义在指定的包中的，那么就需要在把生成的class文件存放到和包名相对于的文件夹中。**

新建文件夹，com/briup/test，并把生成的class文件存放进去

```
briup@briup:~/code/day01$ mkdir -p com/briup/test
briup@briup:~/code/day01$ mv Hello.class com/briup/test/
briup@briup:~/code/day01$ ls
com  Hello.java
```

```
briup@briup:~/code/day01$ ls -R com
com:
briup

com/briup:
test

com/briup/test:
Hello.class
briup@briup:~/code/day01$
```

这时候再运行这个Hello类：

`java Hello`

```
briup@briup:~/code/day01$ java Hello
错误: 找不到或无法加载主类 Hello
briup@briup:~/code/day01$
```

> 可以看到，这时候还是会报错

这是因为，一个类一旦指定的包，那么在运行它的时候，就一定要带上它的包名

加上包名后再次运行：

`java com.briup.test.Hello`

```
briup@briup:~/code/day01$ java Hello
错误: 找不到或无法加载主类 Hello
briup@briup:~/code/day01$ java com.briup.test.Hello
hello world
briup@briup:~/code/day01$
```

> 类的名字有俩种：
> 简单类名：就是直接一个类名，例如 Hello
> 全限定名：包名加类，例如 com.briup.test.Hello

难道每次编译好一个指定包的类，都需要手动去创建和包对应的目录么？

不需要，java中编码命令中有参数，可以直接帮我们在指定位置自动创建和包名对应的目录结构，并且把编译好的class文件自动存放到里面：

`javac -d . Hello.java`

```
briup@briup:~/code/day01$ ls
com  Hello.java
briup@briup:~/code/day01$ rm -rf com
briup@briup:~/code/day01$ ls
Hello.java
briup@briup:~/code/day01$ javac -d . Hello.java
briup@briup:~/code/day01$ ls
com  Hello.java
briup@briup:~/code/day01$
```

```
briup@briup:~/code/day01$ ls -R com
com:
briup

com/briup:
test

com/briup/test:
Hello.class
```

> 命令中，-d表示编译时自动生成和包名对应的目录结构，-d后面的点（.）表示就在当前目录中生成
> 注意，编译成功后，还会自动把编译好的class文件存到这个生成的目录中

运行这个编译好的带包的类：

`java com.briup.test.Hello`

```
briup@briup:~/code/day01$ java com.briup.test.Hello
hello world
briup@briup:~/code/day01$
```

也可以在指定的路径下，生成和包名对应的目录结构：

`javac -d bin Hello.java`

```
briup@briup:~/code/day01$ ls
com  Hello.java
briup@briup:~/code/day01$ rm -rf com
briup@briup:~/code/day01$ mkdir bin
briup@briup:~/code/day01$ javac -d bin Hello.java
briup@briup:~/code/day01$ ls
bin  Hello.java
briup@briup:~/code/day01$
```

`ls -R bin`

```
bin:
com

bin/com:
briup

bin/com/briup:
test

bin/com/briup/test:
Hello.class
```

运行这个类：

`java com.briup.test.hello`

```
briup@briup:~/code/day01$ ls
bin  Hello.java
briup@briup:~/code/day01$ java com.briup.test.Hello
错误: 找不到或无法加载主类 com.briup.test.Hello
briup@briup:~/code/day01$ echo $CLASSPATH
.
briup@briup:~/code/day01$
```

> 可以看出，这时候又报错了

这个错误原因是因为，CLASSPATH中的路径配置的不对，因为我们要运行的Hello类所对应的Hello.class文件，并不在当前目录下，而是./bin中，所以这时候我们可以临时指定一下路径，或者配置CLASSPATH变量

运行当前命令时，临时指定：

`java -cp ./bin com.briup.test.Hello`

```
briup@briup:~/code/day01$ ls
bin  Hello.java
briup@briup:~/code/day01$ java -cp ./bin com.briup.test.Hel
lo
hello world
briup@briup:~/code/day01$ java -cp bin com.briup.test.Hello

hello world
briup@briup:~/code/day01$
```

> -cp参数是-classpath的意思，表示执行当前java命令的时候，临时指定classpath一次，只生效这一次
> 另外，./bin 路径中，可以把./去掉，直接写成bin，因为这样默认就可以表示当前路径下的bin目录

也可以用之前的方式配置CLASSPATH的值：

```
briup@briup:~/code/day01$ java com.briup.test.Hello
错误: 找不到或无法加载主类 com.briup.test.Hello
briup@briup:~/code/day01$ CLASSPATH=$CLASSPATH:bin
briup@briup:~/code/day01$ echo $CLASSPATH
.:bin
briup@briup:~/code/day01$ java com.briup.test.Hello
hello world
briup@briup:~/code/day01$
```

运行一个类的时候，JVM加载这个类的规则是什么？

1. 如果运行的Hello类，**没有指定包**，Hello类一定对应的是Hello.class （ 固定要求 ）
   那么当运行 `java Hello` 的时候，JVM会从CLASSPATH中指定的路径中查找，是否有Hello.class这个文件，如果有那么就加载到内存，然后运行，如果没有那么就报错。
   - 这个情况下，CLASSPATH中就要配置Hello.class文件所在的路径
2. 如果运行的Hello类，**指定了包**，例如是package com.briup.test; Hello类一定对应的是com/briup/test/Hello.class（固定要求）
   那么当运行 `java com.briup.test.Hello` 的时候，JVM会从CLASSPATH中指定的路径中查找，是否有com/briup/test/Hello.class这个文件，如果有那么就加载到内存，然后运行，如果没有那么就报错。
   注意，这个时候JVM从CLASSPATH的路径中，会先找com这个文件夹，然后依次找下去。**因为如果有包存在的时候，这个包就是这个类不可分割的一部分。**

- 这个情况下，CLASSPATH中就要配置com文件夹所在的路径

3. 如果运行的Hello类，被打包到一个jar中，比如是me.jar
   那么当运行 `java Hello` 的时候，JVM会从CLASSPATH中指定的路径中查找，是否有me.jar，如果有那么就从me.jar中将Hello.class载到内存，然后运行，如果没有那么就报错。（这是Hello**没指定包**的情况）
   那么当运行 `java com.briup.test.Hello` 的时候，JVM会从CLASSPATH中指定的路径中查找，

是否有me.jar，如果有那么就从me.jar中将com/briup/test/Hello.class加载到内存，然后运行，如果没有那么就报错。（这是Hello**指定包**的情况）

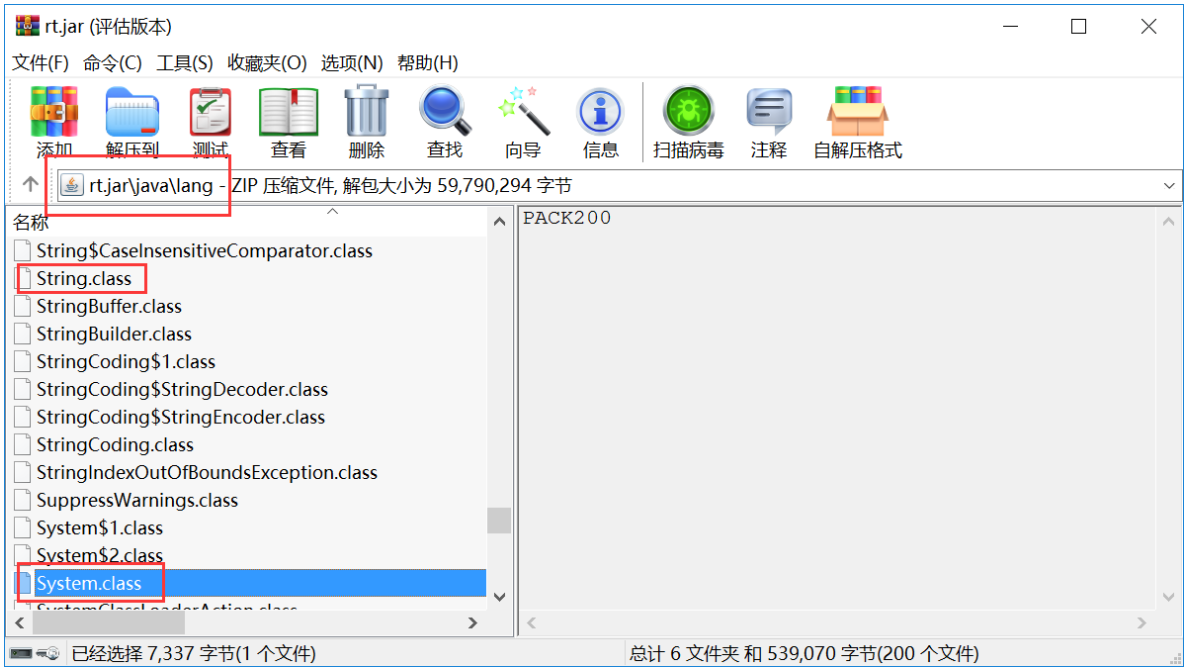
> 注意，这时候，是要把jar文件的路径和jar文件的名字，都配置到CLASSPATH中

## 3 jar

> java中有jar命令，可以将一个或多个class文件，打包到一个指定的jar文件中（xxx.jar）

例如，jre中的rt.jar，就是将src.zip中的java源代码编译成class文件后，又将这些class文件打包到rt.jar中的。

如果电脑中安装了解压缩功能，也可以直接查看这个jar中的内容，解压后再查看：

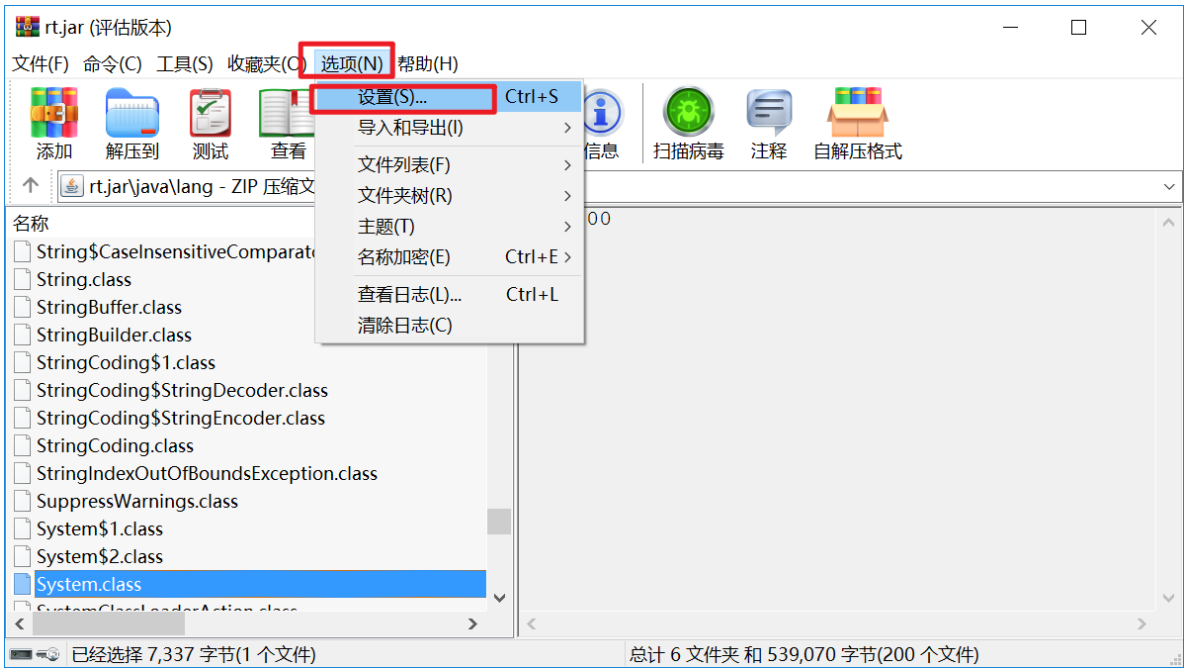


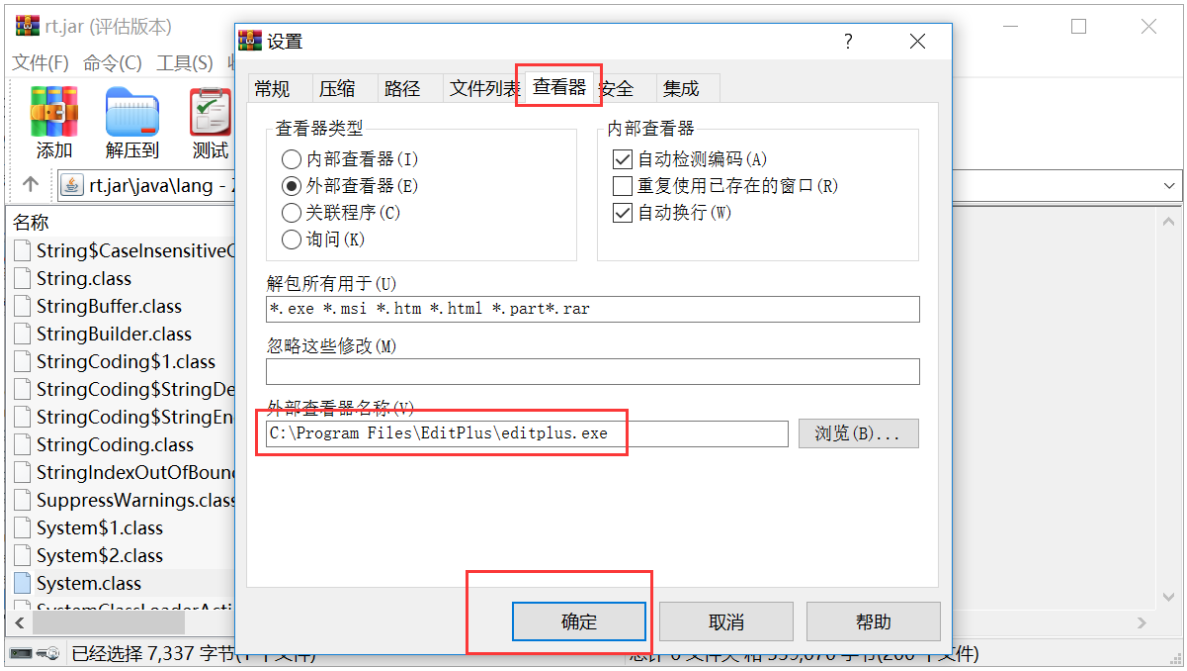

> 注意，Ubuntu中，双击jar包，也可以打开

---

**额外了解内容，start**

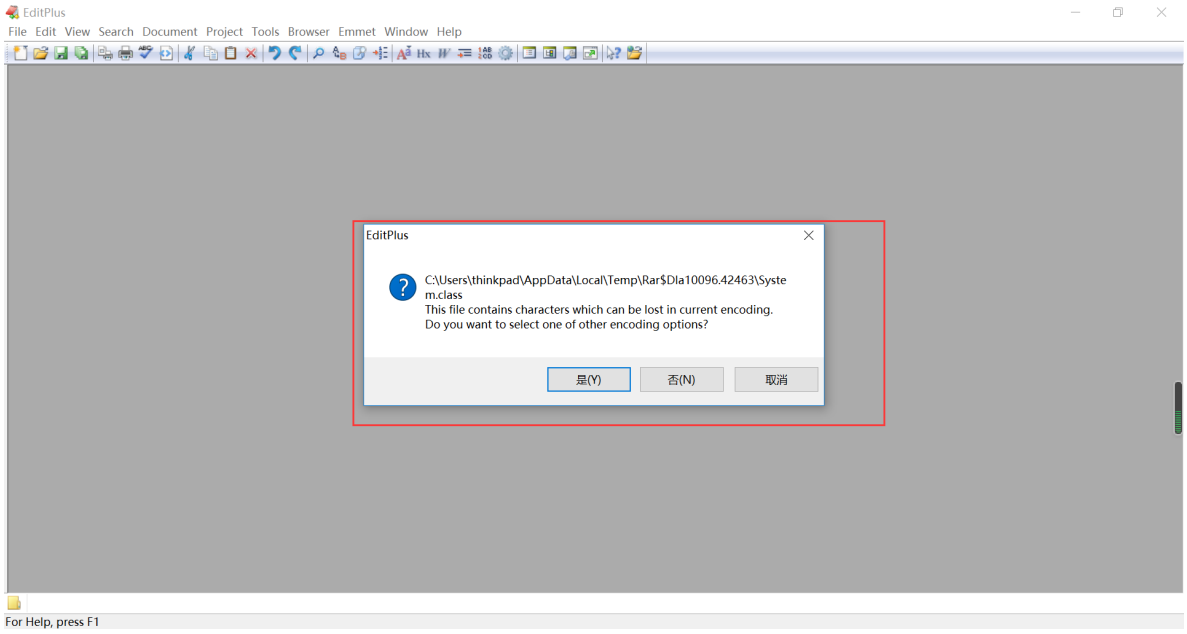
设置了解压缩工具的查看器，并关联系统中安装的编辑器，可以使用指定的编辑软件压缩包的指定文件，而不需要把这个文件真的解压出来：



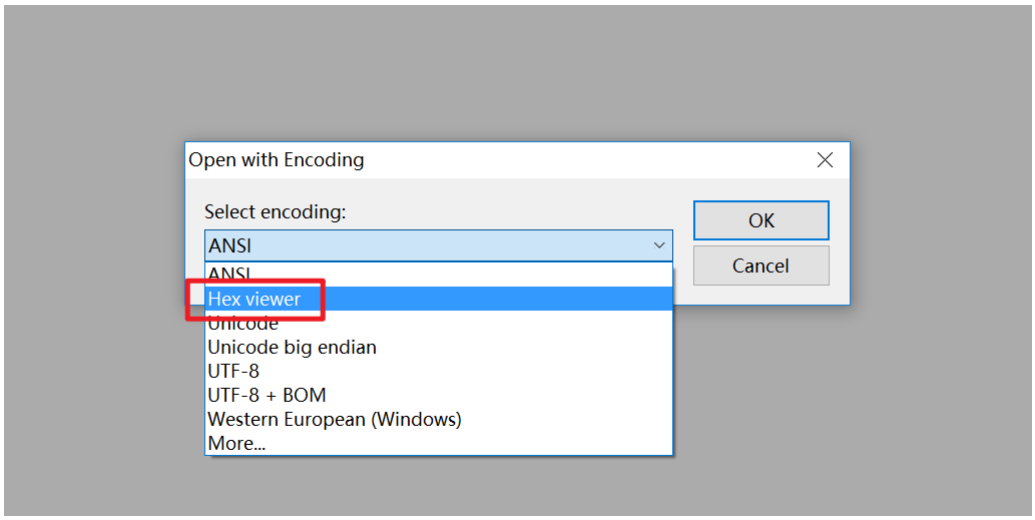



设置完成后，双击指定文件，即可使用系统安装的指定编辑器打开压缩包中的文件

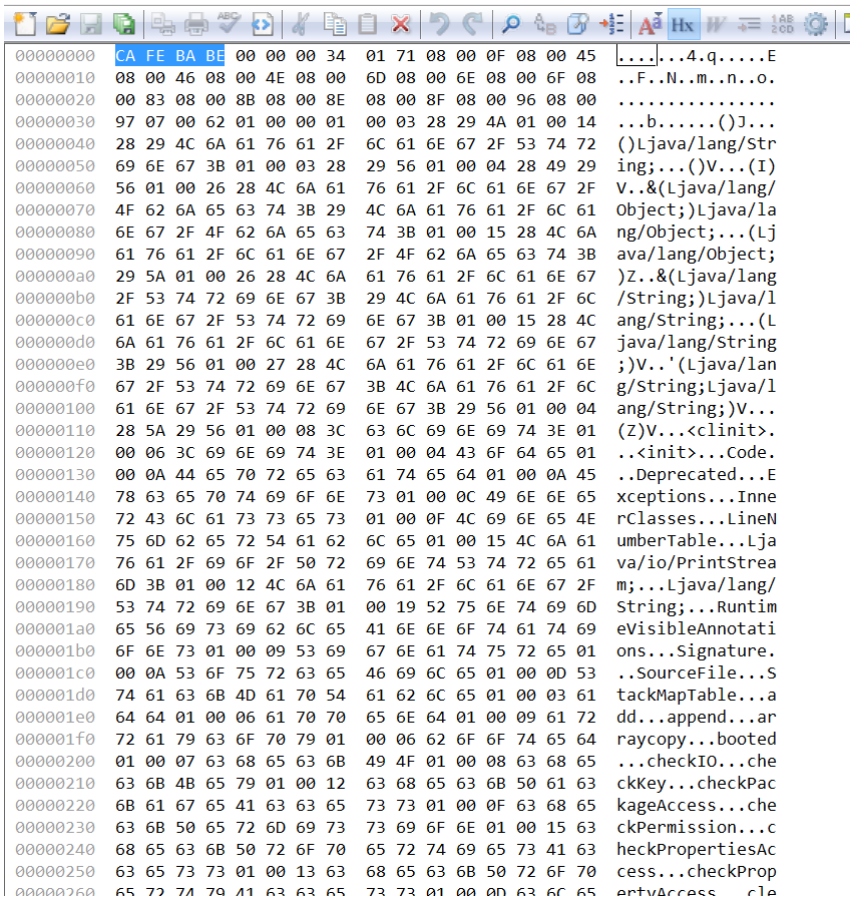
双击System.class文件后，使用editplus打开它，显示让我们显示编码，默认编码打不开，因为class文件是字节码文件，里面是01010的二进制代码：



选择16进制查看器：就是把二进制转为16进制然后显示出来




这个就是System.class文件的16进制表示形式：




> 其中，前面的CA FE BA BE表示当前class是java语言编译而成，这叫**魔数**，java代码编译成的class文件中，最前面一定是这个值。
>
> 其中，00 00 00 34，这个表示当前class文件是那个版本JDK编译出来的。16进制的34 等于 10进制的52 ，52代表的JDK1.8版本，51就是JDK1.7，依次类推

也可以使用javap -verbose Hello.class 命令其查看对应的JDK版本

```
briup@briup:~/code/day01$ javap -verbose bin/com/briup/test
/Hello.class
Classfile /home/briup/code/day01/bin/com/briup/test/Hello.c
lass
  Last modified 2020-7-21; size 430 bytes
  MD5 checksum 9febbaa7e2d7f5db6cc97b04b931031f
  Compiled from "Hello.java"
public class com.briup.test.Hello
  minor version: 0
  major version: 52
```

**额外了解内容，end**

---

如何将自己的class文件进成打包？

1. 把当前目录中的Hello.class打到hello.jar这个jar包中
   jar -cvf hello.jar Hello.class
2. 把当前目录下的Hello.class 以及 World.class打到hello.jar这个jar包中
   jar -cvf hello.jar Hello.class Word.class
3. 把当前目录下的所有的class打到hello.jar这个jar包中
   jar -cvf hello.jar *.class
4. 把当前目录下的bin文件夹里面的所有文件打到这个jar包中，同时【包含】bin目录本身
   jar -cvf hello.jar bin
5. 把当前目录下的bin文件夹里面的所有文件打到这个jar包中，但是【不包含】bin目录本身
   jar -cvf hello.jar -C bin .
   -C bin表示切换到bin目录下执行这个命令，注意bin后的那个点（.）

例如，把当前目录下的bin文件夹里面的所有文件打到这个jar包中，但是【不包含】bin目录本身

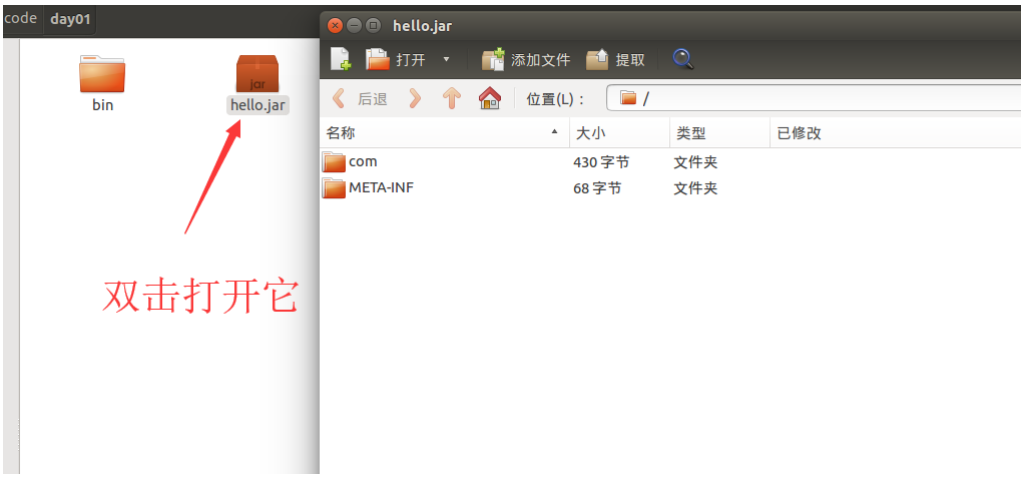
`jar -cvf hello.jar -C bin .`

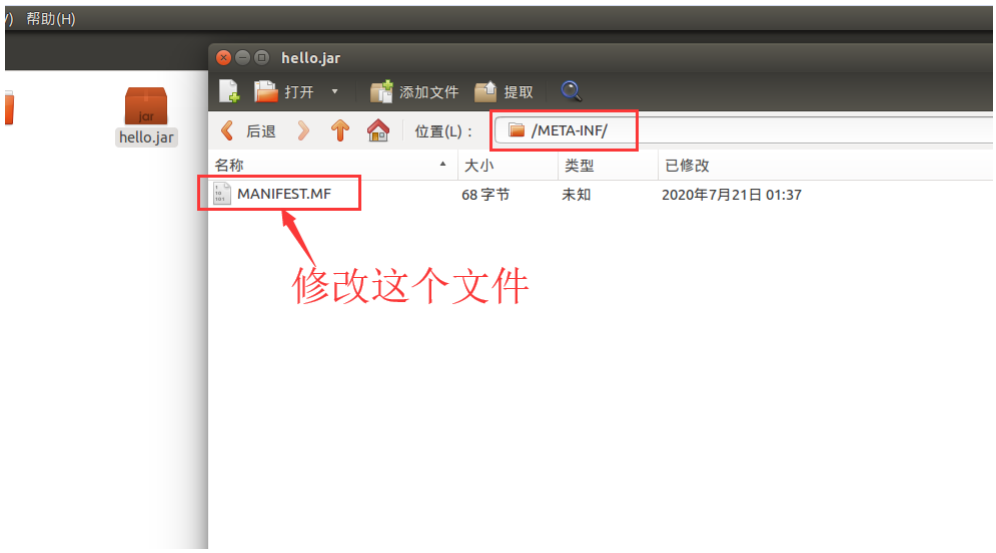
```
briup@briup:~/code/day01$ jar -cvf hello.jar -C bin .
已添加清单
正在添加: com/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: com/briup/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: com/briup/test/(输入 = 0) (输出 = 0)(存储了 0%)
正在添加: com/briup/test/Hello.class(输入 = 430) (输出 = 29
8)(压缩了 30%)
briup@briup:~/code/day01$
```

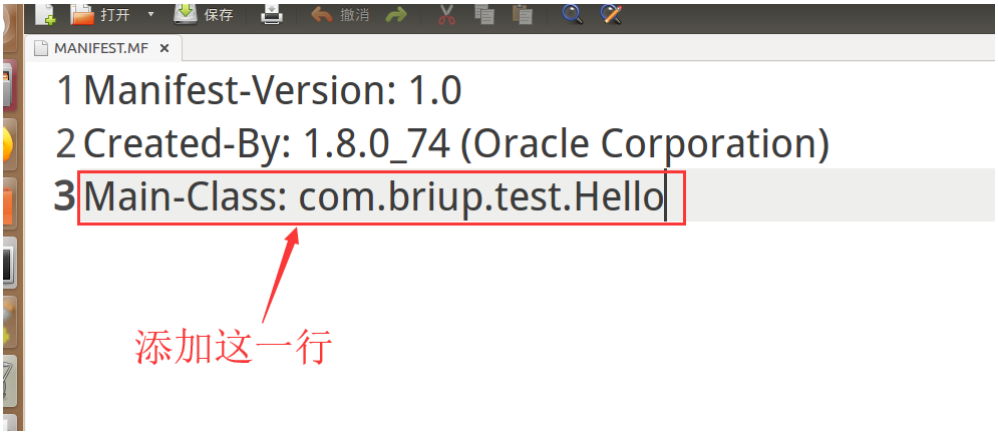
这时候这个jar是一个普通的jar，不能直接java -jar的方式进行运行：

```
briup@briup:~/code/day01$ java -jar hello.jar
hello.jar中没有主清单属性
briup@briup:~/code/day01$
```

原因是因为没有在jar中指定哪一个类中有程序的入口，需要修改jar中的配置：




> Main-Class: com.briup.Hello
> 注意，冒号（:）后面有一个空格，这个是必须的，没有的话报错。

报错这个修改过的文件（ctrl+s），Ubuntu中的解压缩工具会提示我们文件已经修改，是否更新，点击确定就可以了，然后就可以命令窗口中，再运行：

`java -jar hello.jar`

```
briup@briup:~/code/day01$ java -jar hello.jar
hello world
briup@briup:~/code/day01$
```

> 这就是一个可执行的jar包了，因为我们配置了程序的入口类，JVM可以根据这个信息直接运行这个类

同时，我们也可以运行Hello这类，并把这个类所在的jar包位置配置到CLASSPATH中，以便让JVM可以从这个jar包中找Hello.class文件，然后加载运行。

`java com.briup.test.Hello`

```
briup@briup:~/code/day01$ ls
bin  hello.jar  Hello.java
briup@briup:~/code/day01$ rm -rf bin
briup@briup:~/code/day01$ java com.briup.test.Hello
错误: 找不到或无法加载主类 com.briup.test.Hello
briup@briup:~/code/day01$ echo $CLASSPATH
.:bin
briup@briup:~/code/day01$ CLASSPATH=$CLASSPATH:hello.jar
briup@briup:~/code/day01$ echo $CLASSPATH
.:bin:hello.jar
briup@briup:~/code/day01$ java com.briup.test.Hello
hello world
briup@briup:~/code/day01$
```

> 删除bin目录的目的，是为了排除干扰，这样以来，Hello.class文件就只存在于hello.jar包中了

> java com.briup.test.Hello，想要执行，就必须让JVM从CLASSPATH中，找到hello.jar，然后从jar包加载com/briup/test/Hello.class这个文件，所以需要把hello.jar加入到CLASSPATH中

把hello.jar移动到当前目录中的test子目录，然后在做这个操作：

```
briup@briup:~/code/day01$ ls
hello.jar  Hello.java
briup@briup:~/code/day01$ mkdir test
briup@briup:~/code/day01$ mv hello.jar test
briup@briup:~/code/day01$ ls
Hello.java  test
briup@briup:~/code/day01$ java com.briup.test.Hello
错误: 找不到或无法加载主类 com.briup.test.Hello
briup@briup:~/code/day01$ echo $CLASSPATH
.:bin:hello.jar
briup@briup:~/code/day01$
```

> 可以看到，这时候又报错了，原因是因为CLASSPATH中配置的不对，hello.jar不在当前目录了，然后在当前目录的子目录test中

设置CLASSPATH，再去执行

```
briup@briup:~/code/day01$ ls
Hello.java  test
briup@briup:~/code/day01$ echo $CLASSPATH
.:bin:hello.jar
briup@briup:~/code/day01$ CLASSPATH=$CLASSPATH:test/hello.j
ar
briup@briup:~/code/day01$ echo $CLASSPATH
.:bin:hello.jar:test/hello.jar
briup@briup:~/code/day01$ java com.briup.test.Hello
hello world
briup@briup:~/code/day01$
```

> 设置CLASSPATH设置正常，运行成功。
> 这里其实可以直接设置CLASSPATH=test/hello.jar ，因为当前例子中 .:bin:hello.jar 这三个值都用不到

# 4 类加载

> java中的类，想要运行就必须把类对应的class文件加载到内存，JVM中真正负责加载class文件内容的是**类加载器**

在java中，负责把class文件加载到内存的是类加载器（ClassLoader）

JavaSE-API中，有这么一个类：`java.lang.ClassLoader`，它就表示JVM中的类加载器。

JVM启动后，默认会有几种类加载器：

- 启动类加载器 bootstrapClassLoader，非java语言实现
  作用：加载指定路径中jar里面的class文件
  路径1：C:\Program Files\Java\jdk1.8.0_74\jre\lib\
  路径2：C:\Program Files\Java\jdk1.8.0_74\jre\classes\ （ 如果有这个目录的话 ）
  例如：rt.jar
- 扩展类加载器 ExtClassLoader，java语言实现，是ClassLoader类型的对象

  作用：加载指定路径中jar里面的class文件（ **只能是jar中存在的class** ）
  路径：C:\Program Files\Java\jdk1.8.0_74\jre\lib\ext\
  例如：ext中默认存在的jar，或者用户放到ext目录下的jar包
- 应用类加载器 AppClassLoader，java语言实现，是ClassLoader类型的对象

作用：加载指定路径中class文件或者jar里面的class文件
路径：CLASSPATH中配置路径，这个是用户自己配置的
例如：.:bin:hello.jar

> 我们最常使用的就是应用类加载器，因为它可以通过CLASSPATH中的路径，去加载程序员自己编写并编译的class文件到内存中。

> 我们也可以把自己最常用的jar包，放到ext目录中，让扩展类加载器去自动加载这个jar中的class文件到内存中，这样我们的代码就可以直接使用到这个jar中的类了

> 但是其实，大多数情况下，即使我们需要用到其他jar中的代码，也一般会把jar所在的路径配置到CLASSPATH中，让应用类加载器进行加载，这样会更加方便统一管理项目中使用的所有jar

> 关于启动类加载器，它不是java语言编写的，我们一般也不要去动它的路径或者jar，它是负责在JVM启动的时候，把JRE环境中最重要的一些library加载到内存，一旦出问题，JVM就无法正常运行。

思考：我们之前运行的命令 java com.briup.test.Hello，这里的Hello类默认是被哪个类加器加载的？

思考：是否可以改为让扩展类加载器去加载这个Hello类？

---

**额外了解内容，start**

在代码程序中，我们也可以看到这些ClassLoader：

```
1  package com.briup.test;
2  import java.util.Properties;
3
4  //在java中，使用这个变量来代表启动类加载器加载的路径：sun.boot.class.path
5  //在java中，使用这个变量来代表扩展类加载器加载的路径：java.ext.dirs
6  //在java中，使用这个变量来代表应用类加载器加载的路径：java.class.path
7  public class ClassLoaderTest{
8      //main方法中的代码，是查看java运行中可以拿到哪些环境变量和对应的值
9      public static void main(String[] args){
10         Properties p = System.getProperties();
11         p.forEach((k,v)->System.out.println(k+"\t"+v));
12     }
13 }
```

```
1  package com.briup.test;
2  import java.util.Properties;
3
4  public class ClassLoaderTest{
```

```
    public static void main(String[] args){
        ClassLoader appClassLoader = ClassLoader.getSystemClassLoader();
        System.out.println(appClassLoader);
        //sun.misc.Launcher$AppClassLoader@2a139a55

        ClassLoader extClassLoader = appClassLoader.getParent();
        System.out.println(extClassLoader);
        //sun.misc.Launcher$ExtClassLoader@7852e922

        ClassLoader bootClassLoader = extClassLoader.getParent();
        System.out.println(bootClassLoader);
        //null
        //注意：启动类加载在java中无法表示，因为它不是java语言实现的。所有这里输出null
    }
}
```

**额外了解内容，end**

---

# 5 双亲委托机制

作为了解的内容

> 多个类加载器之间共同协作，然后把需要使用或运行的类加载到内存去执行，它们直接共同合作的方式就是双亲委托机制。

例如：java com.briup.test.Hello 命令

- 现在要加载Hello.class文件中的类
- 首先加载任务就交给了AppClassLoader
- 然后AppClassLoader把这个任务委托给自己的父加载器，也就是ExtClassLoader
- 然后ExtClassLoader把这个任务委托给自己的父加载器，也就是bootstrapClassLoader
- 然后bootstrapClassLoader就尝试去加载Hello这个类，但是在指定路径下并没有找到
- 然后任务又交回给了ExtClassLoader，ExtClassLoader尝试加载Hello这个类，但是在指定路径中没找到
- 然后任务又交回给了AppClassLoader
- 最后AppClassLoader从CLASSPATH中指定的路径里面找到并加载了这个Hello类，完成类加载的过程

思考：JavaSE-API中已经提供了一个类java.lang.System的类，如果我们也编写一个类加做System，同时指定它在java.lang包下面，那么这个时候我们是否能使用自己编写的java.lang.System类来代替JavaSE-API中的java.lang.System类？