

# java反序列化基础知识总结

## java的类加载过程

1. 首先是调用 `public Class<?> loadClass(String name)` 方法，通过public方法调用保护方法

`protected Class<?> loadClass(String name, boolean resolve)`

```
350 public Class<?> loadClass(String name) throws ClassNotFoundException {
351     return loadClass(name, resolve: false);
352 }
353
```

Loads the class with the specified binary name. The default implementation of this method searches for classes in the following order:

1. Invoke `findLoadedClass(String)` to check if the class has already been loaded.
2. Invoke the `loadClass` method on the parent class loader. If the parent is null the class loader built-in to the virtual machine is used, instead.

2. 在 `protected loadClass` 方法中，第400行会调用一个 `findLoadedClass` 方法判断当前类是否已经加载。如果类已经加载，直接返回当前类的类对象。

```
395 protected Class<?> loadClass(String name, boolean resolve)
396     throws ClassNotFoundException
397 {
398     synchronized (getClassLoadingLock(name)) {
399         // First, check if the class has already been loaded
400         Class<?> c = findLoadedClass(name);
401         if (c == null) {
402             long t0 = System.nanoTime();
403             try {
404                 if (parent != null) {
405                     c = parent.loadClass(name, resolve: false);
406                 } else {
407                     c = findBootstrapClassOrNull(name);
408                 }
409             } catch (ClassNotFoundException e) {
410                 // ClassNotFoundException thrown if class not found
411                 // from the non-null parent class loader
412             }
413
414             if (c == null) {
415                 // If still not found, then invoke findClass in order

```

3. 如果创建当前 `ClassLoader` 时传入了父类加载器 (`new ClassLoader (父类加载器)`) 就使用父类加载器加载 `TestHelloWorld` 类，否则使用 JVM 的 `Bootstrap ClassLoader` 加载。

```
403 try {
404     if (parent != null) {
405         c = parent.loadClass(name, resolve: false);
406     } else {
407         c = findBootstrapClassOrNull(name);
408     }
409 } catch (ClassNotFoundException e) {
410     // ClassNotFoundException thrown if class not found
411     // from the non-null parent class loader
412 }
413
```

4. 如果通过类加载器没有办法加载类，则会通过 `findClass` 方法尝试加载类。

```
414         if (c == null) {
415             // If still not found, then invoke findClass in order
416             // to find the class
417             long t1 = System.nanoTime();
418             c = findClass(name);
419
420             // this is the defining class loader; record the stats
421             sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
422             sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
423             sun.misc.PerfCounter.getFindClasses().increment();
424         }
425     }
426     if (resolve) {
427         resolveClass(c);
428     }
429     return c;
430 }
```

5. 如果当前的 `ClassLoader` 没有重写 `findClass` 方法，则会直接返回类不存在。跟进 `findClass` 方法进行查看。如果当前类重写了 `findClass` 方法并通过传入的类名找到了对应的类字节码，那么应该调用 `defineClass` 方法去 JVM 中注册该类。

6. 如果调用 `loadClass` 的时候传入的 `resolve` 参数为 `true`，那么还需要调用 `resolveClass` 方法链接类，默认为 `false`。

7. 返回一个 JVM 加载后的 `java.lang.Class` 类对象

8. 通过重写 `ClassLoader#findClass` 方法实现自定义类的加载。

```
package ClassLoader_;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.lang.reflect.Method;
import java.util.Arrays;

public class selfClassload_ extends ClassLoader{
    public static String className="ClassLoader_.HelloWorld";
    public static byte[] fileByte;
    public static void main(String[] args) throws Exception {
        FileInputStream fis=new
        FileInputStream("D:\\Java\\project\\study\\serializeSummary\\src\\HelloWorld.class");
        fileByte=fileToByte(fis);
        System.out.println(Arrays.toString(fileByte));
        selfClassload_ classload = new selfClassload_();
        Class<?> aclass = classload.loadClass(className);
        Object helloworld = aclass.newInstance();
        Method hello = helloworld.getClass().getMethod("hello");
        System.out.println((String)hello.invoke(helloworld));
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        if (name.equals(className)){
            return defineClass(className,fileByte,0,fileByte.length);
        }
        return super.findClass(name);
    }

    public static byte[] fileToByte(FileInputStream fis) throws Exception{ //
        将.class文件转成二进制编码
        byte[] buffer=null;
```

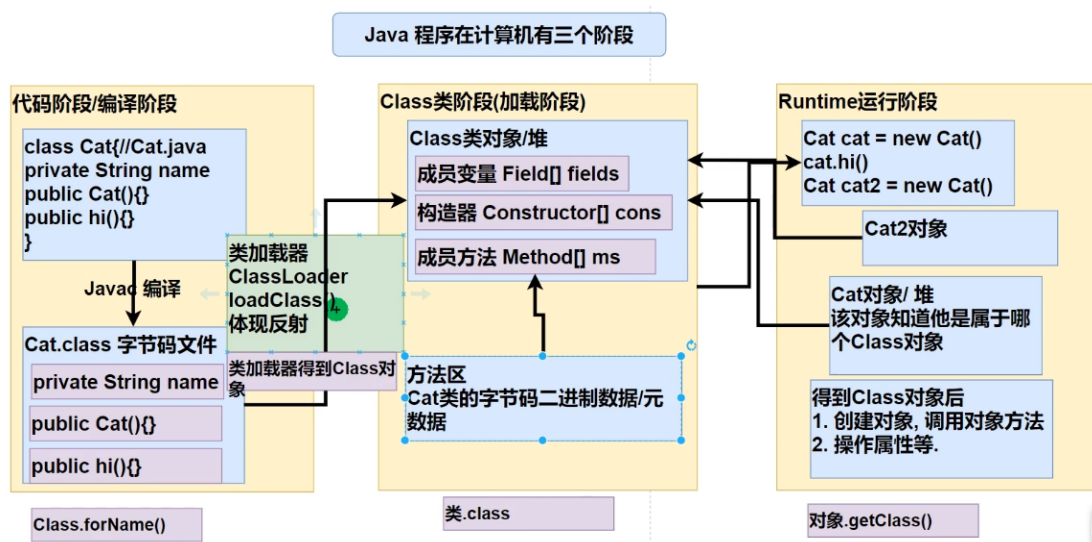
```

        ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
        byte[] b=new byte[1024];
        int n;
        while ((n=fis.read(b))!=-1){
            byteArrayOutputStream.write(b,0,n);
        }
        fis.close();
        byteArrayOutputStream.close();
        buffer=byteArrayOutputStream.toByteArray();
        return buffer;
    }
}

```

## Class类

1. 上面讲到通过 `loadClass` 方法加载之后得到的是一个 `java.lang.Class` 类对象，关于这个 `Class` 类对象需要简单说明一下。在 `java` 中存在一个 `Class` 类，作用是当类加载完成之后将类的各种属性，方法进行封装成单独的对象。



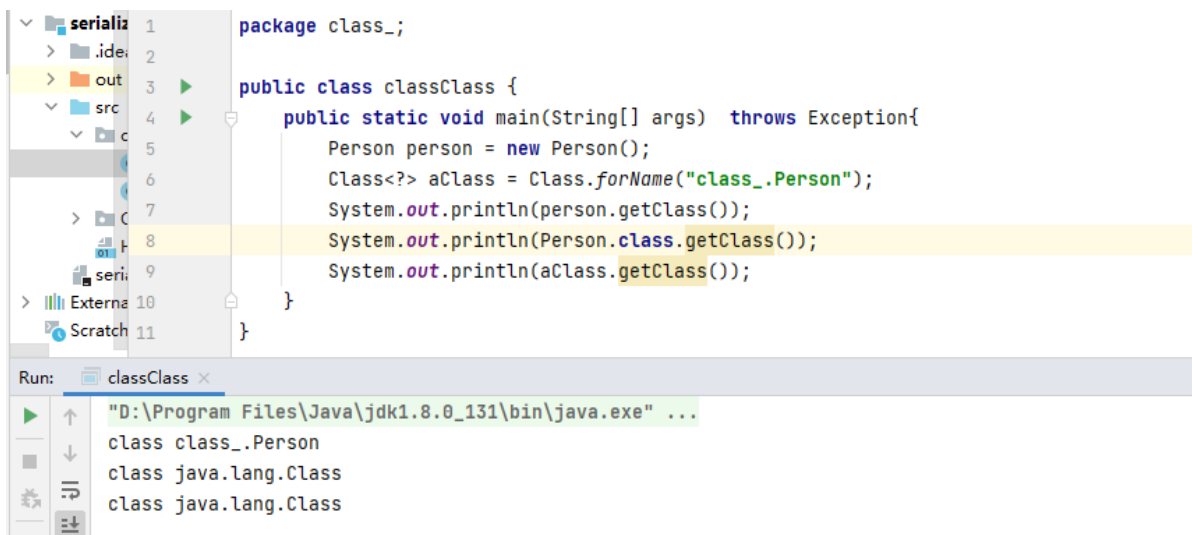
2. Source阶段 `Person.class` 表示 `Person.java` 的字节码文件，在Class类对象阶段，通过实例化一个 `Class` 类生成一个 `Class` 类对象用于描述 `Person.class` 字节码当中的内容，然后runtime阶段创建对象，也是通过 `Class` 类对象进行创建的。
3. 一个小案例解释一下 `Class` 类对象。

```

package class_;

public class classClass {
    public static void main(String[] args) throws Exception{
        Person person = new Person();
        Class<?> aClass = Class.forName("class_.Person");
        System.out.println(person.getClass());
        System.out.println(Person.class.getClass());
        System.out.println(aClass.getClass());
    }
}

```



```
1 package class_;
```

```
2
```

```
3 public class classClass {
```

```
4     public static void main(String[] args) throws Exception{
```

```
5         Person person = new Person();
```

```
6         Class<?> aClass = Class.forName("class_.Person");
```

```
7         System.out.println(person.getClass());
```

```
8         System.out.println(Person.class.getClass());
```

```
9         System.out.println(aClass.getClass());
```

```
10     }
```

```
11 }
```

Run: classClass x

```
"D:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
```

```
class class_.Person
```

```
class java.lang.Class
```

```
class java.lang.Class
```

## 解释

1. person对象 -->类型Person类
2. aclass对象 -->类型Class类（Class类的一个对象）
3. 在加载类之后，在堆中就产生一个Class类型的对象（一个类只有一个Class对象），这个对象包含了被加载的类的完整结构信息。通过这个对象得到类的结构，这个Class类型的对象就像一面镜子，可以看到类的结构，所以形象的称之为反射。

## 关于class类的几个点：

1. Class也是类，因此也继承Object类（类图）
2. Class类对象不是new出来的，而是系统创建的
3. 对于某个类的Class类对象，在内存中只有一份，因此类只加载一次。
4. 每个类的实例都会记得自己是由哪个Class实例生成的
5. 通过Class可以完整的得到一个类的完整结构，通过一系列的API
6. Class对象是存放在堆当中的
7. 类的字节码二进制数据是放在方法区的，有的地方称之为类的元数据（包括方法代码，变量名，方法名，访问权限等等）

# java反射

## 概念：

将类的各个组成部分封装为其他对象，这就是反射机制

Java反射操作的是java.lang.Class对象。

在加载类之后，在堆中就产生一个Class类型的对象（一个类只有一个Class对象），这个对象包含了被加载的类的完整结构信息。通过这个对象得到类的结构，这个Class类型的对象就像一面镜子，可以看到类的结构，通过对Class类提供的一系列API就可以对某个对象进行操作。所以形象的称之为反射。

```
person.java
```

```
package reflect1;
```

```
public class Person {
```

```
    private String name="张三";
```

```
    public int age=10;
```

```
    private int bge=20;
```

```
    protected int cge=30;
```

```
    int dge=40;
```

```
    public Person(){
```

```
    }
```

```

public Person(String name, int age) {
    this.name = name;
    this.age = age;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

@Override
public String toString() {
    return "Person{" +
        "name='" + name + '\'' +
        ", age=" + age +
        ", bge=" + bge +
        ", cge=" + cge +
        ", dge=" + dge +
        '}';
}

public void eat(){
    System.out.println("eating。。。");
}

public void say(String content){
    System.out.println(content);
}

}

```

## 1. 反射获取 class 类对象

```

package reflect1;

public class Demo1 {

    public static void main(String[] args) throws Exception {
        //获取class类对象的方式
        //字节码阶段，此时字节码还未进入内存当中
        //Class.forName("全类名")，将字节码文件加载进入内存，返回Class对象
        Class cls1=Class.forName("reflect1.Person");
        System.out.println(cls1);

        //字节码文件已经加载进入内存
        //通过类名获取。
        //类名.class
    }
}

```

```

        Class cls2=Person.class;//
        System.out.println(cls2);
        Class runtime2=Runtime.class;
        System.out.println(runtime2);
        Runtime runtime3=Runtime.getRuntime();
        System.out.println(runtime3.getClass());
        //runtime阶段
        //对象.getClass() 在Object类中定义，全部对象都继承了这个方法
        Person p=new Person();
        Class cls3=p.getClass();
        System.out.println(cls3);
    }
}
/*
类加载器加载class文件进入内存
类加载器对应java的ClassLoader对象
在内存中通过Class类来描述.class字节码文件
Class类用来描述字节码的内容
将其余类对象的变量封装为Field对象
构造方法封装为Constructor对象
成员方法封装为Method对象

同一个字节码文件在一次程序的运行过程中，只会被加载一次。
*/

```

## 2. 反射获取成员变量

```

package reflect1;

import java.lang.reflect.Field;

public class Demo2 {
    public static void main(String[] args){
        /*
        获取全部成员变量
        */
        try {
            Class personClass = Person.class;
            Field[] fields=personClass.getFields(); //获取所有public修饰的成员变量，
            其余类型都不可以

            for (Field field:fields){
                System.out.println(field); //public int reflect1.Person.age
            }
            Field ageField=personClass.getField("age"); //同样只能获取public成员变
            量

            System.out.println(ageField);
            //操作成员变量
            //Person p=new Person();
            Object p=personClass.getConstructor().newInstance();
            Object result=ageField.get(p); //get方法需要传递一个对象作为参数
            System.out.println(result);
            //设置值
            ageField.set(p,100);
            System.out.println(ageField.get(p)); //再次取值变成100
            System.out.println("=====");

```

```

        Field[] fields1=personClass.getDeclaredFields(); //可以打印全部的成员变量，不管修饰符
        for (Field field : fields1){
            System.out.println(field);
        }
        Field bgeField=personClass.getDeclaredField("bge");
        bgeField.setAccessible(true);//(暴力反射)直接取值会爆出异常。想要直接取值，需要忽略权限修饰符的安全检查
        Object bge=bgeField.get(p); //直接取值会爆出异常。想要直接取值，需要忽略权限修饰符的安全检查
        System.out.println(bge);
        bgeField.set(p,200);
        System.out.println(bgeField.get(p));
    }catch (Exception e){
        System.out.println("=====");
        System.out.println(e);
    }
}
}
}

```

### 3. 反射创建对象

```

package reflect1;

import java.lang.reflect.Constructor;

public class Demo3 {
    public static void main(String[] args) {
        //获取构造方法
        try{
            Class personClass=Person.class;
            Constructor
            constructor=personClass.getConstructor(String.class,int.class);
            System.out.println(constructor);

            //创建对象
            Object zhangsan=constructor.newInstance("张三",20);
            System.out.println(zhangsan.toString());
            System.out.println("=====");
            Constructor constructor2=personClass.getConstructor();
            Object lisi=constructor2.newInstance();
            ///lisi.setName("李四");
            System.out.println(lisi.toString());
        }catch (Exception e){
            System.out.println(e);
        }
    }
}
}

```

### 5. 反射获取成员方法并执行

```

package reflect1;

```

```

import java.lang.reflect.Method;

public class Demo4 {
    //获取成员方法
    public static void main(String[] args) {
        Class personClasee = Person.class;
        //获取public成员方法
        try {
            Method[] personMethod = personClasee.getMethods();
            for (Method method: personMethod) {
                System.out.println(method);
                /*
                public java.lang.String reflect1.Person.toString()
                public java.lang.String reflect1.Person.getName()
                public void reflect1.Person.setName(java.lang.String)
                public void reflect1.Person.say(java.lang.String)
                public void reflect1.Person.eat()
                public void reflect1.Person.setAge(int)
                public int reflect1.Person.getAge()
                public final void java.lang.Object.wait() throws
java.lang.InterruptedException
                public final void java.lang.Object.wait(long,int) throws
java.lang.InterruptedException
                public final native void java.lang.Object.wait(long) throws
java.lang.InterruptedException
                public boolean java.lang.Object.equals(java.lang.Object)
                public native int java.lang.Object.hashCode()
                public final native java.lang.Class java.lang.Object.getClass()
                public final native void java.lang.Object.notify()
                public final native void java.lang.Object.notifyAll()
                */
                //获取方法名称
                System.out.println(method.getName());
            }
        } catch (Exception e){
            System.out.println(e);
        }
        System.out.println("=====");
        //获取指定方法{
        try {
            Person p=new Person();
            //System.out.println(personClasee.getClass());
            Method methodEat = personClasee.getMethod("eat"); //空参方法1
            methodEat.invoke(p);

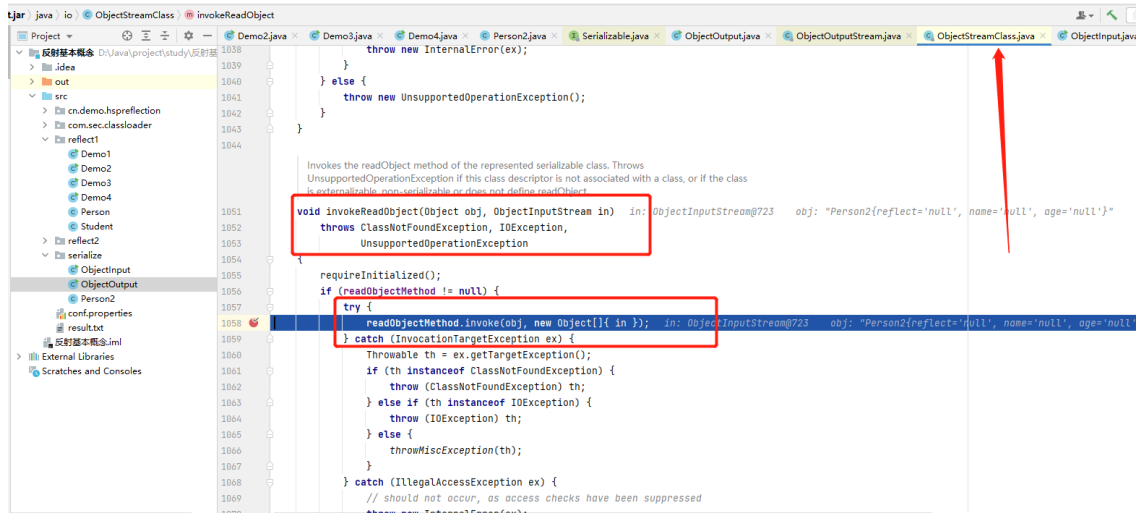
            Method methodSay=personClasee.getMethod("say", String.class);
            methodSay.invoke(p,"张三");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```



# 序列化与反序列化

- 关于重写 `readObject` 方法, 以及重写 `readObject` 方法之后, 反序列化的时候是如何调用 `readObject` 的, 可以通过 debug 的方式进行跟踪, 最后发现在 `ObjectStreamClass.java` 中存在一个反射调用。



- 简单的反序列化实例。

Person类:

```
package serialize;
```

```
import java.io.IOException;
```

```
import java.io.Serializable;
```

```
/*
```

```
需要实现serializable接口
```

```
*/
```

```
public class Person2 implements Serializable {
```

```
    public transient String reflect;
```

```
    private String name;
```

```
    public String age;
```

```
    //private static final long serialVersionUID=-5702540850087186263L;
```

//SerialVersionUID 序列化版本号的作用是用来区分我们所编写的类的版本, 用于判断反序列化时类的版本是否一直, 如果不一致会出现版本不一致异常。

```
    private void readObject(java.io.ObjectInputStream in)
```

```
        throws IOException, ClassNotFoundException{
```

```
        //in.defaultReadObject();
```

```
        System.out.println("111111");
```

```
    }
```

```
@Override
```

```
public String toString() {
```

```
    return "Person2{" +
```

```
        "reflect='" + reflect + '\'' +
```

```
        ", name='" + name + '\'' +
```

```
        ", age='" + age + '\'' +
```

```
        '}';
```

```
}
```

```
public Person2(String reflect, String name, String age) {
```

```
    this.reflect = reflect;
```

```
    this.name = name;
```

```

        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }
}

```

ObjectOutput类:

```
package serialize;
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
```

```
/*
```

```
java的序列化流，ObjectOutputStream (OutputStream out)
```

```
1、创建objoutput对象
```

```
2、使用writeobj方法将对象写入文件
```

```
*/
```

```
public class ObjectOutput {
    public static void main(String[] args) throws IOException {
        ObjectOutputStream oos=new ObjectOutputStream(new
FileOutputStream("src\\result.txt"));
        oos.writeObject(new Person2("test","张三","20"));
        //java.io.NotSerializableException: serialize.Person  Person类为实现序列化
接口
        oos.close();
    }
}

```

ObjectInput类:

```
package serialize;
```

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
```

```
/*
```

```
在序列化当中，因为静态方法是优先于对象加载进入内存的，所以成员变量是不能被序列化的
```

```
*/
```

```
public class ObjectInput {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {

```

```
        ObjectInputStream objInput=new ObjectInputStream(new
FileInputStream("src\\result.txt"));
        Object obj=objInput.readObject();
        //obj.getName();
        System.out.println(obj);
        Person2 p=(Person2) obj;
        System.out.println(p.toString());
    }

}
```

- 反序列的几个关键知识点.

1. 读写顺序一致
2. 实现 Serializable 接口
3. static 和 transient 关键字修饰的属性不被反序列化
4. 内部属性的类型也需要实现 Serializable 接口
5. 具有继承性,父类可以序列化那么子类同样可以