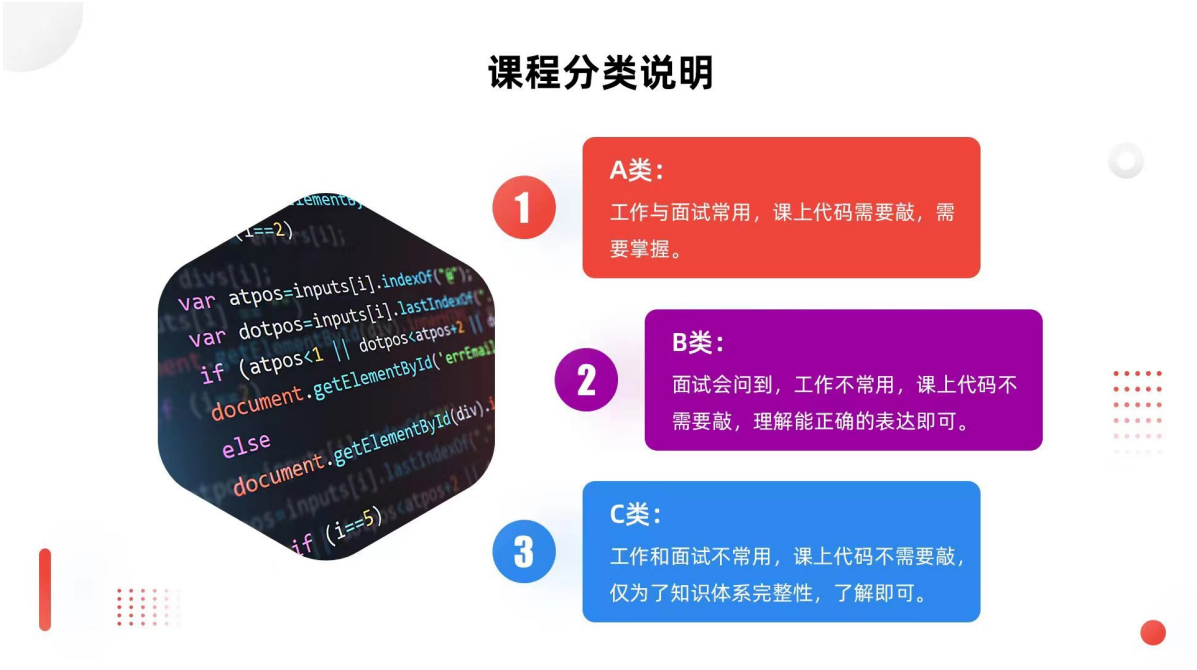
反射章节说明



学习计划说明

总学时：2时14分，分为1天学习。

第一天总学时：2时14分钟

序号	课程名称	时长	分类
1	反射介绍	05分24秒	A类
2	创建对象过程分析	07分49秒	A类
3	获取Class对象的三种方式-getClass()方法	09分52秒	A类
4	获取Class对象的三种方式-class静态属性	05分00秒	A类
5	获取Class对象的三种方式-forName()方法获取Class对象	06分18秒	A类
6	获取类的构造方法-方法说明与使用	15分11秒	A类
7	获取构造方法-通过构造方法创建对象	06分00秒	A类
8	获取成员变量-方法说明及使用	09分13秒	A类
9	获取成员变量-操作成员变量	09分58秒	A类
10	获取方法-方法说明及使用	10分23秒	A类
11	获取方法-调用方法	10分30秒	A类
12	获取类的其他信息	07分43秒	A类
13	反射应用案例	12分10秒	A类
14	反射机制效率	09分08秒	A类
15	setAccessible()方法	10分14秒	A类

实操说明

A类课程中的内容需要同学们跟着老师动手完成。

反射机制介绍

什么是反射

Java 反射机制是Java语言一个很重要的特性，它使得Java具有了“动态性”。在Java程序运行时，对于任意的一个类，我们能不能知道这个类有哪些属性和方法呢？对于任意的一个对象，我们又能不能调用它任意的方法？答案是肯定的！这种动态获取类的信息以及动态调用对象方法的功能就来自于Java 语言的反射（Reflection）机制。

反射的作用

简单来说两个作用，RTTI（运行时类型识别）和DC（动态创建）。

我们知道反射机制允许程序在运行时取得任何一个已知名称的class的内部信息，包括其modifiers(修饰符)，fields(属性)，methods(方法)等，并可于运行时改变fields内容或调用methods。那么我们便可以更灵活的编写代码，代码可以在运行时装配，无需在组件之间进行源代码链接，降低代码的耦合度；还有动态代理的实现等等；但是需要注意的是反射使用不当会造成很高的资源消耗！

实时效果反馈

1.如下对Java反射描述错误的是？

A 反射可以使代码在运行时装配；

B 反射可以降低代码的耦合度；

C 通过反射可以实现动态代理；

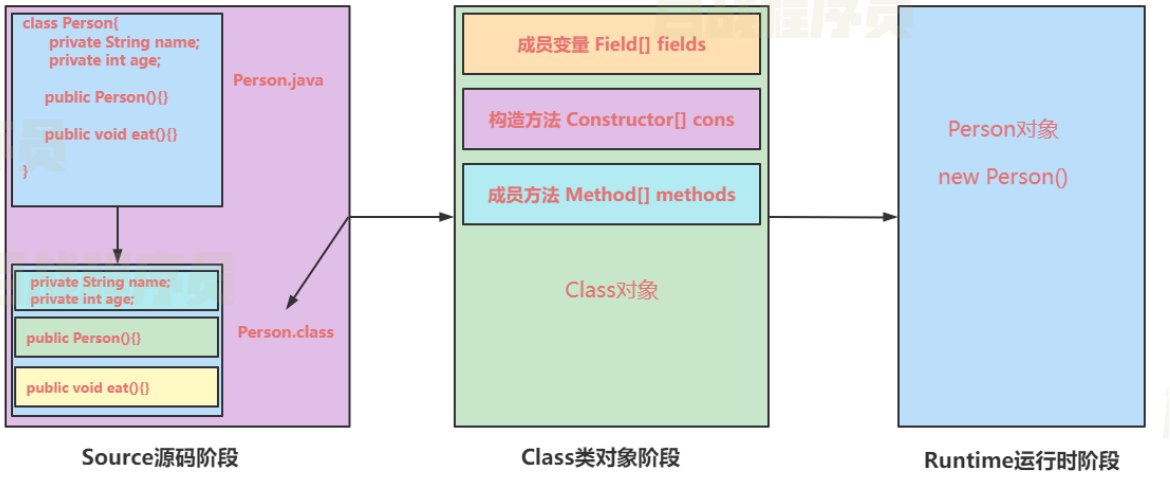
D 反射不会造成很高的资源消耗；

答案

1=>D

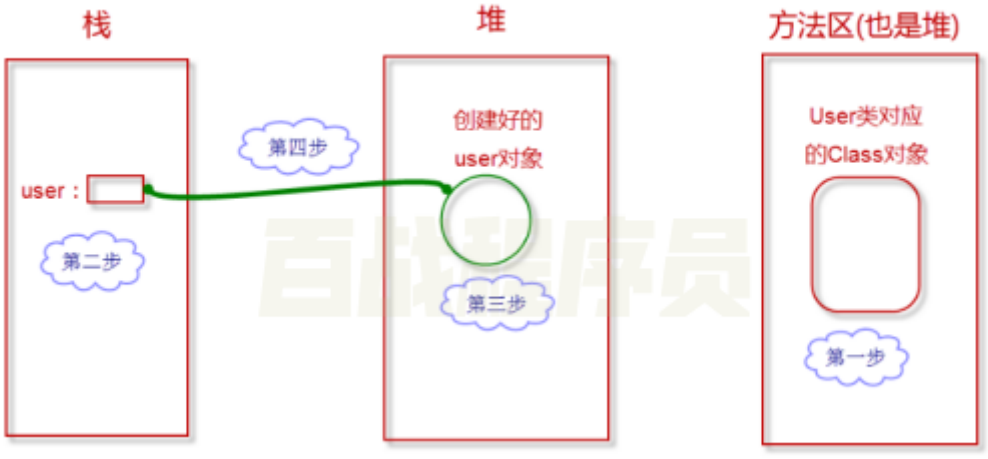
创建对象过程

Java创建对象的三个阶段



创建对象时内存结构

```
1  Users user = new Users();
2
```



实际上，我们在加载任何一个类时都会在方法区中建立“这个类对应的Class对象”，由于“Class对象”包含了这个类的整个结构信息，所以我们可以通过这个“Class对象”来操作这个类。

我们要使用一个类，首先要加载类；加载完类之后，在堆内存中，就产生了一个 Class 类型的对象（一个类只有一个 Class 对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象知道类的结构。这个对象就像一面镜子，透过这个镜子可以看到类的结构，所以，我们形象的称之为：反射。 因此，“Class对象”是反射机制的核心。

实时效果反馈

1.如下对Class对象描述错误的是？

A Class对象包含了这个类的整个结构信息；

B 通过Class对象可以获取类的相关信息；

C 一个类可以有多个 Class 对象；

D Class对象是反射机制的核心；

答案

1=>C

反射的具体实现

获取Class对象的三种方式

- 通过getClass()方法;
- 通过.class 静态属性;
- 通过Class类中的静态方法forName();

创建Users类

```
public class Users {
  private String username;
  private int userage;


  public String getUsername() {
    return username;
   }


  public void setUsername(String username) {
    this.username = username;
   }

```

```
15
16    public int getUserage() {
17      return userage;
18     }
19
20
21    public void setUserage(int userage) {
22      this.userage = userage;
23     }
24  }
25
26
27
```

通过getClass()方法获取Class对象

```
1   /*
2   * 通过getClass()方法获取该类的Class对象
3   getClass()为Object类下的非静态方法，在使用时需要先实例化对象
4    */
5   public class GetClass1 {
6     public static void main(String[] args) {
7       Users users = new Users();
8       Users users1 = new Users();
9       Class clazz = users.getClass();
10      System.out.println(clazz);
11      System.out.println(clazz.getName());
12      System.out.println(users.getClass() == users1.getClass());
13     }
14  }
15
```

通过forName()获取Class对象

```java
/**
 * 通过Class.forName("class Name")获取Class对象
 */
public class GetClass3 {
  public static void main(String[] args)throws Exception {
    Class clazz = Class.forName("com.bjsxt.Users");
    Class clazz2 = Class.forName("com.bjsxt.Users");
    System.out.println(clazz);
    System.out.println(clazz.getName());
    System.out.println(clazz == clazz2);
   }
}

```

通过.class 静态属性获取Class对象

```java
/**
 * .class静态属性获取Class对象
 */
public class GetClass2 {
  public static void main(String[] args) {
    Class clazz = Users.class;
    Class clazz2 = Users.class;
    System.out.println(clazz);
    System.out.println(clazz.getName());
    System.out.println(clazz == clazz2);
   }
}

```

获取类的构造方法

方法介绍

方法名	描述
getDeclaredConstructors()	返回 Constructor 对象的一个数组，这些对象反映此 Class 对象表示的类声明的所有构造方法。
getConstructors()	返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共（public）构造方法。
getConstructor(Class<?>... parameterTypes)	返回一个 Constructor 对象，它反映此 Class 对象所表示的类的指定公共（public）构造方法。
getDeclaredConstructor(Class<?>... parameterTypes)	返回一个 Constructor 对象，该对象反映此 Class 对象所表示的类或接口的指定构造方法。

方法使用

修改Users类

```
public class Users {
  private String username;
  private int userage;
  public Users(){
   }

  public Users(String username,int userage){
    this.username= username;
    this.userage=userage;
   }

  public Users(String username){
    this.username= username;
   }

  private Users(int userage){
    this.userage = userage;
   }


  public String getUsername() {
    return username;
   }


  public void setUsername(String username) {
```

```
    this.username = username;
   }


  public int getUserage() {
    return userage;
   }


  public void setUserage(int userage) {
    this.userage = userage;
   }
}

```

获取构造方法

```
public class GetConstructor {
  public static void main(String[] args)throws Exception {
    Class clazz = Users.class;
    Constructor[] arr = clazz.getDeclaredConstructors();
    for(Constructor c:arr){
      System.out.println(c);
     }
    System.out.println("----------------------");

    Constructor[] arr1 = clazz.getConstructors();
    for(Constructor c:arr1){
      System.out.println(c);
     }
    System.out.println("------------------------");

    Constructor c =  clazz.getDeclaredConstructor(int.class);
    System.out.println(c);
    System.out.println("------------------------");

    Constructor c1 =  clazz.getConstructor(null);
    System.out.println(c1);
   }
}

```

通过构造方法创建对象　**newInstance()**

```java
/**
**通过反射实例化对象
*/
public class GetConstructor2 {

    public static void main(String[] args) throws Exception{
        //创建类对象
        Class<Users> usersClass = Users.class;
        //通过类对象获取指定的构造方法对象
        Constructor<Users> constructor =
usersClass.getConstructor(String.class, int.class);
        //通过指定的构造方法对象创建对象
        Users users = constructor.newInstance("郭家旗", 20);

        System.out.println(users);
    }
}
```

获取类的成员变量

方法介绍

方法名	描述
getFields()	返回Field类型的一个数组,其中包含 Field对象的所有公共(public)字段。
getDeclaredFields()	返回Field类型的一个数组,其中包含 Field对象的所有字段。
getField(String fieldName)	返回一个公共成员的Field指定对象。
getDeclaredField(String fieldName)	返回一个 Field指定对象。

方法使用

修改Users类

```java
public class Users {
  private String username;
  public int userage;
  public Users(){
```

```
    }
   public Users(String username,int userage){
     this.username= username;
     this.userage=userage;
    }
   public Users(String username){
     this.username= username;
    }
   private Users(int userage){
     this.userage = userage;
    }


   public String getUsername() {
     return username;
    }


   public void setUsername(String username) {
     this.username = username;
    }


   public int getUserage() {
     return userage;
    }


   public void setUserage(int userage) {
     this.userage = userage;
    }
}


```

获取成员变量

```
public class GetField {

    public static void main(String[] args) throws Exception{
        Class<Users> usersClass = Users.class;
        Field[] fields = usersClass.getFields();
        for(Field field:fields){
            System.out.println(field);
        }
        System.out.println("====================");
```

```
        Field[] declaredFields = usersClass.getDeclaredFields();
        for(Field field:declaredFields){
            System.out.println(field);
        }
        System.out.println("=================");
        Field userName = usersClass.getDeclaredField("userName");
        System.out.println(userName);
        System.out.println("=================");
        Field userAge = usersClass.getField("userAge");
        System.out.println(userAge);
    }
}
```

操作成员变量 先实例化对象

```
public class GetField2 {
    public static void main(String[] args)throws Exception {
        //获取Users类的类对象
        Class<Users> usersClass = Users.class;
        //获取类的成员变量
        Field userAge = usersClass.getField("userAge");
        //通过构造方法实例化users对象
        Users users = usersClass.getConstructor(null).newInstance();
        //给指定成员变量赋值
        userAge.set(users,20);
        System.out.println(userAge.get(users));
    }
}

```

获取类的方法

方法介绍

方法名	描述
getMethods()	返回一个Method类型的数组,其中包含 所有公共(public)方法。包含父类中的（public）方法！！！！！
getDeclaredMethods()	返回一个Method类型的数组,其中包含 所有方法。
getMethod(String name, Class<?>... parameterTypes)	返回一个公共的Method方法对象。
getDeclaredMethod(String name, Class<?>... parameterTypes)	返回一个方法Method对象

方法使用

修改Users类

```
public class Users {
    private String userName;
    public int userAge;

    private Users(String userName){
        this.userName = userName;
    }
    public Users(String userName, int userAge){
        this.userName = userName;
        this.userAge = userAge;
    }

    public Users(){

    }

    public Users(int userAge){
        this.userAge = userAge;
    }


    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

```

```
    public int getUserAge() {
        return userAge;
    }

    public void setUserAge(int userAge) {
        this.userAge = userAge;
    }


    private void sing(){
        System.out.println("郭家旗爱唱歌");
    }


    @Override
    public String toString() {
        return "Users{" +
                "userName='" + userName + '\'' +
                ", userAge=" + userAge +
                '}';
    }
}

```

获取方法

```
public class GetMethod {

    public static void main(String[] args) throws Exception{
        Class<Users> usersClass = Users.class;
        Method[] classMethods = usersClass.getMethods();
        for(Method method : classMethods){
            System.out.println(method);
            System.out.println(method.getName());
        }
        System.out.println("----------------");
        Method[] declaredMethods = usersClass.getDeclaredMethods();
        for(Method method : declaredMethods){
            System.out.println(method);
            System.out.println(method.getName());
        }
        System.out.println("=================");
        Method setUserAge = usersClass.getMethod("setUserAge",
int.class);
        System.out.println(setUserAge.getName());
```

```
        System.out.println(setUserAge);
        System.out.println("===============");
        Method sing = usersClass.getDeclaredMethod("sing");
        System.out.println(sing);
        System.out.println(sing.getName());
    }
}

```

调用方法 **invoke( )**

```
public class GetMethod2 {

    public static void main(String[] args)throws Exception {
        //实例化类对象
        Class<Users> usersClass = Users.class;
        Method setUserName = usersClass.getMethod("setUserName",
String.class);
        //实例化对象
        Users users = usersClass.getConstructor(null).newInstance();
        //通过setUserName赋值
        setUserName.invoke(users,"郭家旗");
        //通过getUserName获取值
        Method getUserName = usersClass.getMethod("getUserName");
        Object userName = getUserName.invoke(users);
        System.out.println(userName);

    }
}

```

获取类的其他信息

```
public class GetClassInfo {

    public static void main(String[] args) {
        Class<Users> usersClass = Users.class;

        //获取包名

```

```
        Package usersClassPackage = usersClass.getPackage();
        System.out.println(usersClassPackage);
        System.out.println(usersClassPackage.getName());

        //获取类名
        String usersClassName = usersClass.getName();
        System.out.println(usersClassName);

        //获取超类
        Class<? super Users> superclass = usersClass.getSuperclass();
        System.out.println(superclass.getName());

        //获取所有接口
        Class<?>[] classInterfaces = usersClass.getInterfaces();
        for(Class interfaces:classInterfaces){
            System.out.println(interfaces.getName());
        }
    }
}

```

反射应用案例

需求：根据给定的方法名顺序来决定方法的执行顺序。

```
class Reflect {
    public void method1(){
        System.out.println("method1---------");
    }
    public void method2(){
        System.out.println("method2---------");
    }
    public void method3(){
        System.out.println("method3---------");
    }
}


public class ReflectDemo {
    public static void main(String[] args) throws Exception{
        Reflect reflect = new Reflect();
        if(args≠null&&args.length>0){
            Class<? extends Reflect> reflectClass =
reflect.getClass();
```

```
            Method[] classMethods = reflectClass.getMethods();
            for(String s:args){
                for(Method method:classMethods){
                    if(s.equalsIgnoreCase(method.getName())){
                        method.invoke(reflect);
                        break;
                    }
                }
            }
        }else {
            reflect.method1();
            reflect.method2();
            reflect.method3();
        }
    }
}

```

反射机制的效率

由于Java反射是要解析字节码，将内存中的对象进行解析，包括了一些动态类型，而JVM无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多！

接下来我们做个简单的测试来直接感受一下反射的效率。

反射机制的效率测试

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

        Class<?> aClass = Class.forName("com.itbaizhan.Users");
        Object o = aClass.getConstructor(null).newInstance();
        Method setUsername =
aClass.getMethod("setUserName",String.class);
        long reflectStart = System.currentTimeMillis();
        for(int i =0;i<100000000;i++){
            setUsername.invoke(o,"郭家旗");
        }
        long reflectEnd = System.currentTimeMillis();

        long start = System.currentTimeMillis();
        Users u =new Users();
```

```
        for(int i=0;i<100000000;i++){
            u.setUserName("郭家旗");
        }
        long end = System.currentTimeMillis();

        System.out.println("反射执行时间: "+(reflectEnd-reflectStart));
        System.out.println("普通方式执行时间: "+(end-start));
    }
}
反射执行时间: 169
普通方式执行时间: 0
```

setAccessible方法

setAccessible()方法：

setAccessible是启用和禁用访问安全检查的开关。值为 true 则指示反射的对象在使用时应该取消 Java 语言访问检查。值为 false 则指示反射的对象应该实施 Java 语言访问检查;默认值为false。

由于JDK的安全检查耗时较多.所以通过setAccessible(true)的方式关闭安全检查就可以达到提升反射速度的目的。

```
public class Test2 {
    public static void main(String[] args) throws Exception{
        Class<Users> usersClass = Users.class;
        Users users = usersClass.getConstructor(null).newInstance();
        Field userName = usersClass.getDeclaredField("userName");
        //忽略安全检查
        userName.setAccessible(true);
        userName.set(users,"郭家旗");
        Object o = userName.get(users);
        System.out.println(o);

        Method sing = usersClass.getDeclaredMethod("sing");
        //忽略安全检查
        sing.setAccessible(true);
        sing.invoke(users);

    }
}
```

本章总结

- Java 反射机制是Java语言一个很重要的特性，它使得Java具有了“动态性”。
- 反射机制的优点：
  - 更灵活。
  - 更开放。
- 反射机制的缺点：
  - 降低程序执行的效率。
  - 增加代码维护的困难。
- 获取Class类的对象的三种方式：
  - 运用getClass()。（非静态方法，需要先实例化）
  - 运用.class 语法。
  - 运用Class.forName()（最常被使用）。
- 反射机制的常见操作
  - 动态加载类、动态获取类的信息（属性、方法、构造器）。
  - 动态构造对象。
  - 动态调用类和对象的任意方法。
  - 动态调用和处理属性。
  - 获取泛型信息。
  - 处理注解。