Annotation and Reflection

1.What is annotation:

It is not code, but interpret codes. It can be read by other programs (like compiler).

Annotations are read through reflection mechanisms.

它是以@的形式存在在代码中的,还可以添加一些参数.@SupperWarnings(values="unchecked")

它可以添加在package,class,method,field等上面,相当于给他们添加了额外的信息.他们有检查的功能。

2.什么是内置注解：

@Override lange package @Deprecated @SupressWarnings

3.meta-annotation

解释其它注解的注解： @Target @Retention @Documented @Inherited

@Target ---> 用于描述的注解的使用范围(Type(class),Method,Constructor,Parameter,Package)

@Retention---->什么级别保存该注解信息 （Source < Class <Runtime),framework 中统一用Runtime

@Documented---->说明该注解将被包含在javadoc 文档中

@Inherited---->说明子类可以继承父类的该注解

4.Annotation 的写法

```
public class Test03 {
    //if we don't assign a value, it will take default value
    @MyAnnotation2(name="Fred",age = 18)
    public void m1(){
    }

    @MyAnnotation3("Celine")
    public void m2(){

    }
}

@Target({ElementType.TYPE,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation2{
    //Annotation's parameter   Type + ParameterName +()
    String name() default "";
    int age() default 0;
    int id() default -1;// negative value means no such an id
    String[] schools() default {"Concordia","Mcgill"};

}
@interface MyAnnotation3{
    String value();
}
```

Reflection

1.静态语言与动态语言

动态语言：是一类在运行时可以改变其结构的语言：例如新的函数,对象,甚至代码可以被引进,已有函数可以被删除或者是其他结构上的变化。通俗点就是代码可以在运行时根据某些条件改变自身结构。

主要动态语言：Object-C, C#，JavaScript, PHP, Python等

静态语言：运行时结构不可变的语言就是静态语言。 例如 Java C C++

Java不是动态语言，但是可以称“准动态语言”。即java有一定的动态性，我们可以利用反射机制获得类似动态语言的性质。java的动态性让编程更加灵活。

2.反射

Reflection: 是Java语言的关键，反射允许程序在执行期间借助于Reflection API取得任何类的内部信息，

并能直接操作任意对象的内部属性以及方法。（private 类也可以）

正常方式：

引入需要的包类名称---->通过new实例化----取得实例化对象

反射方式：

实例化对象---->getClass()方法---->得到完整的包类名称

优点：

可以实现动态创建对象和编译，体现出很大的灵活性

缺点：

对性能有影响。使用反射基本是一种解释操作，我们告诉JVM,我们希望做什么并且它满足我们的需求。

这类操作总是慢于直接执行相同的操作

反射相关的API:

java.lang.class: 代表一个类

java.lang.reflect.method: 代表类的方法

java.lang.reflect.Field：代表类的成员变量

java.lang.reflect.Constructor：代表类的构造器

...

一个实例通过反射得到类的

```
public class Test02 {
    //get a class object through reflection
    public static void main(String[] args) throws ClassNotFoundException {
        Class<?> c1 = Class.forName("reflection.User");
        Class<?> c2 = Class.forName("reflection.User");
        Class<?> c3 = Class.forName("reflection.User");
        Class<?> c4 = Class.forName("reflection.User");

        System.out.println(c1);
        //一个类在内存中只有一个Class Object
        //一个类被加载后，整个类的结构就会封装到class object中
```

```
        System.out.println(c1.hashCode());
        System.out.println(c2.hashCode());
        System.out.println(c3.hashCode());
        System.out.println(c4.hashCode());

    }
}

//pojo entiy 实体类
class User{
    private String name;
    private int id;
    private int age;

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

    public String getName() {
        return name;
    }

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

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getAge() {
        return age;
    }

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

    @Override
    public String toString() {
        return "User{" +
                "name='" + name + '\'' +
                ", id=" + id +
                ", age=" + age +
                '}';
    }
}
```

public final Class getClass()

通过对象反射求出类的名称

Class 类的常用方法

static ClassforName(String name) 返回制定类名的class对象

Object newInstance() 调用缺省构造函数，返回class对象的实例

getName()返回此class对象所表示的实体（类，接口，数组类或void)的名称

Class getSuperClass() 返回当前Class对象的父类的Class 对象

Class[] getInterfaces() 获取当前class对象的接口

Constructor[] getConstructors() 返回一个包含某些Constructor对象的数组

Method　getMethod(String name,Class ... T) 返回一个Method对象，此对象的形参类型为paramType

Field[] getDeclaredFields() 返回Field对象的一个数组

获取Class类实例具体方法

(a) 此方法最为安全可靠，性能最高

Class clazz = Person.class;

(b)已知某个类的实例，调用该实例的getClass()方法获取Class对象

Class clazz = person.getClass();

(c)已知一个类的全类名，且该类在类路径下，可以通过Class 静态方法forName()获取，

肯能抛出ClassNotFoundException

Class clazz = Class.forName("demo01.Student")

```
public class Test03 {
    public static void main(String[] args) throws ClassNotFoundException {
        Person person = new Student();
        System.out.println("This is a "+ person.name);
        //Approach 1: get a Class object through Class
        Class<Person> c1 = Person.class;
        System.out.println(c1.hashCode());
        //Approach 2: get a class object through object
        Class c2 = person.getClass();
        System.out.println(c2.hashCode()+"student Class");
        //Approach 3: through forName
        Class c3 = Class.forName("reflection.Person");
        System.out.println(c3.hashCode());
        //Student hashcode
        Class c4 = Student.class;
        System.out.println(c4.hashCode()+"student class");
        //内置类型的包装类的Type属性
        Class c5 = Integer.TYPE;
        System.out.println(c5.hashCode());
        //Get a class's super Class
        Class c6 = c2.getSuperclass();
        System.out.println(c6);
    }
}
class Person{
    String name;
```

```
27
28      public Person(String name) {
29          this.name = name;
30      }
31      public Person(){}
32  }
33  class Student extends Person{
34      public Student(){
35          this.name = "Concordia Student";
36      }
37  }
38
39  class Teacher extends Person{
40      public Teacher(){
41          this.name = "Concordia Teacher";
42          }
43
44  }
```

哪些类有Class类型

```
1   //测试哪些类型有class
2   public class Test04 {
3       public static void main(String[] args) {
4           Class c1 = Object.class; //Class 类型
5           Class c2 = Comparable.class;//interface
6           Class c3 = String[].class;// one dimension array
7           Class c4 = int[][].class;// two dimensions array
8           Class c5 = Override.class;//annotation
9           Class c6 = ElementType.class;//enumeration type
10          Class c7 = Integer.class;//primitive
11          Class c8 = void.class;// void
12          Class c9 = Class.class;//class
13
14          System.out.println(c1);//ctr + d 复制
15          System.out.println(c2);
16          System.out.println(c3);
17          System.out.println(c4);
18          System.out.println(c5);
19          System.out.println(c6);
20          System.out.println(c7);
21          System.out.println(c8);
22          System.out.println(c9);
23          //只要元素类型与维度一样就是同一个class
24          int[] a =new int[10];
25          int[] b =new int[100];
26          System.out.println(a.getClass().hashCode());
27          System.out.println(b.getClass().hashCode());
28
29
30
31      }
32
33  }
```

Class的内存分析

加载：将class文件字节码（.class文件）内容加载到内存中，并将这些静态数据转化成方法区的运行时数据结构

，然后生成代表这个类的java.lang.Class对象

链接：将java类的二进制代码合并到JVM的运行状态之中的过程

验证：确保加载类信息符合JVM规范，没有安全方面问题

准备：正式为类变量（static）分配内存并设置类变量默认初始值的阶段，这些内存将在方法区中分配

解析：虚拟机常量池内的符号引用（常量名）替换为直接引用（地址）的过程

初始化：（JVM 工作)

执行类构造器()方法的过程。类构造器方法()是由编译期自动收集类中所有类变量

的赋值动作和静态代码块中的语句合并产生的。（类构造器是构造类信息的，不是构造类对象的构造器）

```
//demonstrate how a class is loaded
public class Test05 {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(A.m);
        /**
         * 1.加载到内存，会产生一个对应的class对象
         * 2.链接结束后，m=0 赋予了一个默认值
         * 3.初始化
         *         <clinit>(){
         *          System.out.println("A Class static codes are being initialized");
         *          m = 300;
         *          static int m = 100;
         *         }
         *         m = 100;
         * 4.new一个object时就会调用constructor,调用
         * System.out.println("A Class is being initialized through non-parameter constructor");
         */
    }


}

class A{
    static{
        System.out.println("A Class static codes are being initialized");
        m = 300;
    }
    static int m = 100;

    public A(){
        System.out.println("A Class is being initialized through non-parameter constructor");
    }
}
```

分析类的初始化

```java
//测试类什么时候会初始化
public class Test06 {
    static{
        System.out.println("Main class is being loaded");
    }

    public static void main(String[] args) throws ClassNotFoundException {
        //1.主动引用
       // Son son = new Son();
        //2.反射也会产生主动引用
        //Class.forName("reflection.Son");
        //3.不会产生类的引用的方法
        //3.1通过子类调用父类的static变量或者方法
       // System.out.println(Son.b);
        //3.2 定义数组
        Son[] arr = new Son[5];
        //3.3 子类常量池的数据
        System.out.println(Son.M);
    }
}
class Father{
    static int b = 2;
    static{
        System.out.println("Father class is being loaded");
    }
}

class Son extends Father{
    static{
        System.out.println("Son class is being loaded");
        m = 300;
    }
    static int m = 100;

    static final int M = 1;
}
```

类加载器的作用

类加载的作用：将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后在堆中生成一个代表这个类的java.lang.Class对象,作为方法区中类数据的访问入口

类缓存：

标准的javaSE类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间

不过JVM垃圾回收机制可以回收这些class对象。

类加载器有哪些类型：

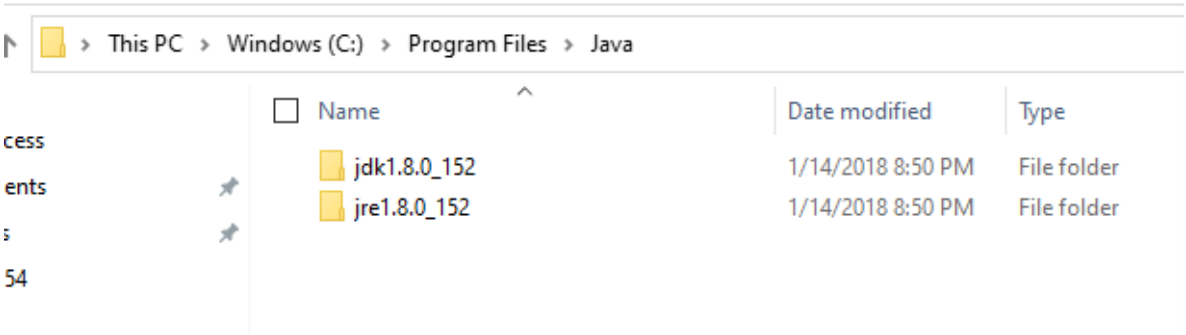
（1）引导类加载器： C++编写，JVM自带的类加载器,负责java平台核心库。用来装载核心类库，该加载器无法直　　接获得.（rt.jar） ------>(Bootstrap classloader)

(2) 扩展类加载器：负责 jre/lib/ext目录下的jar包加载到我们的工作库（ext.jar)------>(Extension classloader)

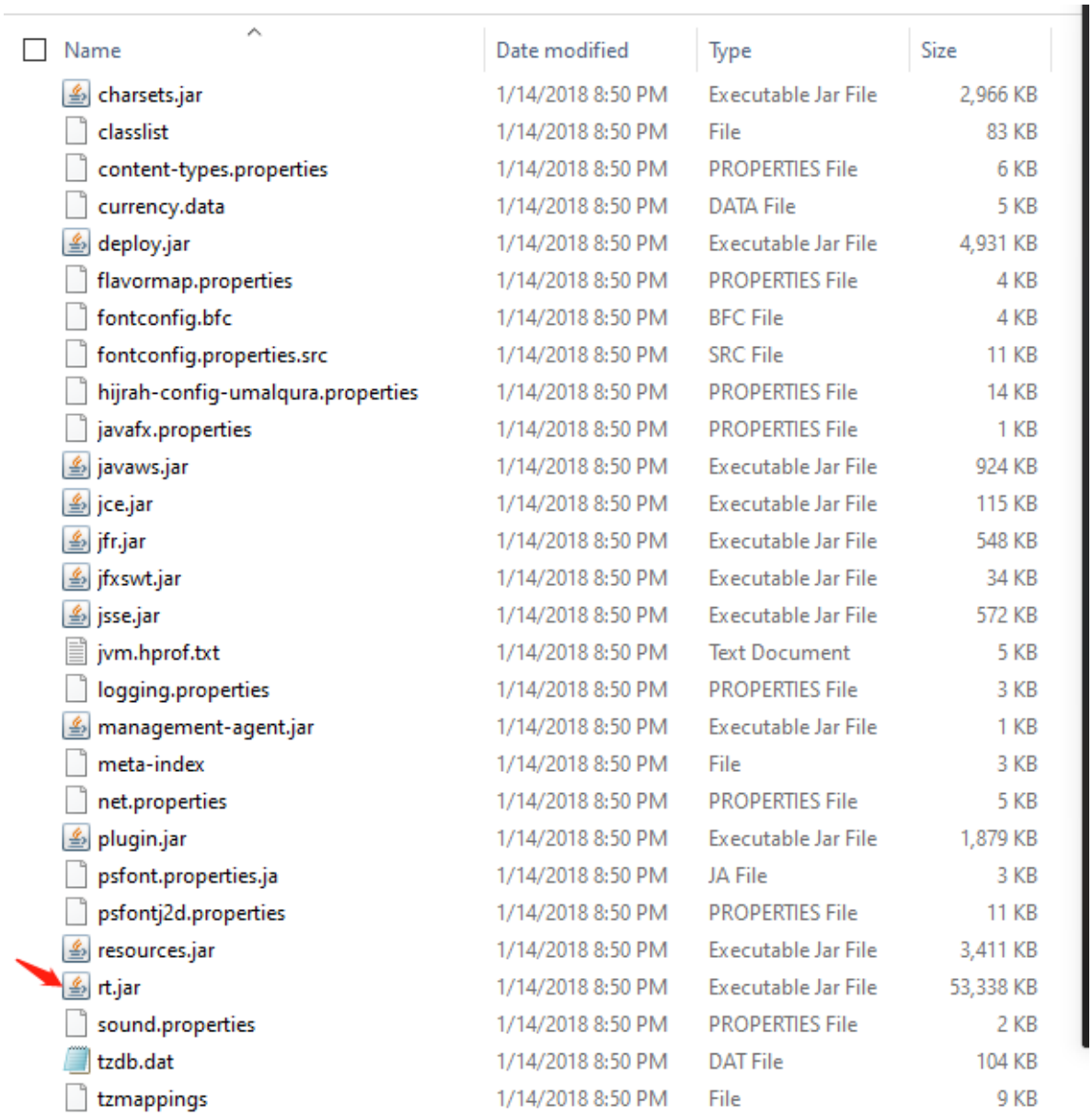
（3）系统类加载器：负责java-classpath或-D java.class.path所指的目录下的类与jar包装入工作。是最常用的类 加载器。------->(System classloader)

JRE 解析：

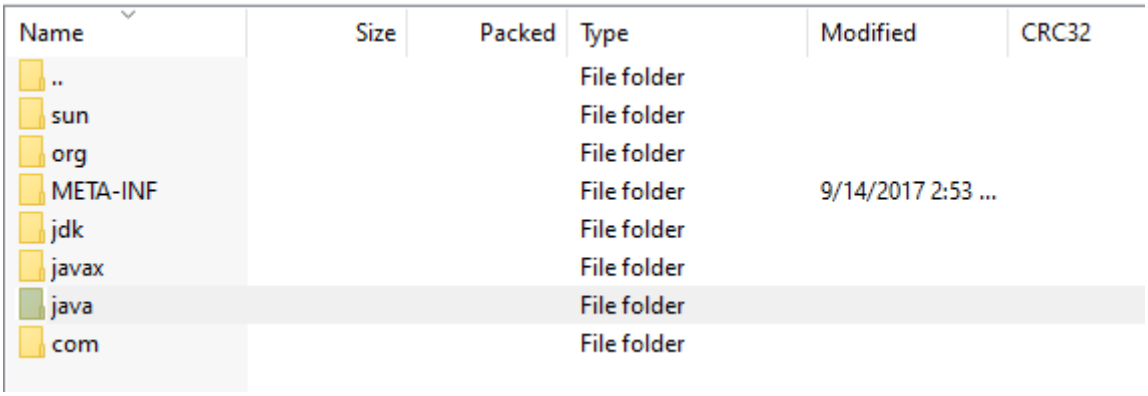
（1）找到jre jdk包



（2）打开lib里的rt.jar(open with winRAR)---->(Bootstrap classloader)

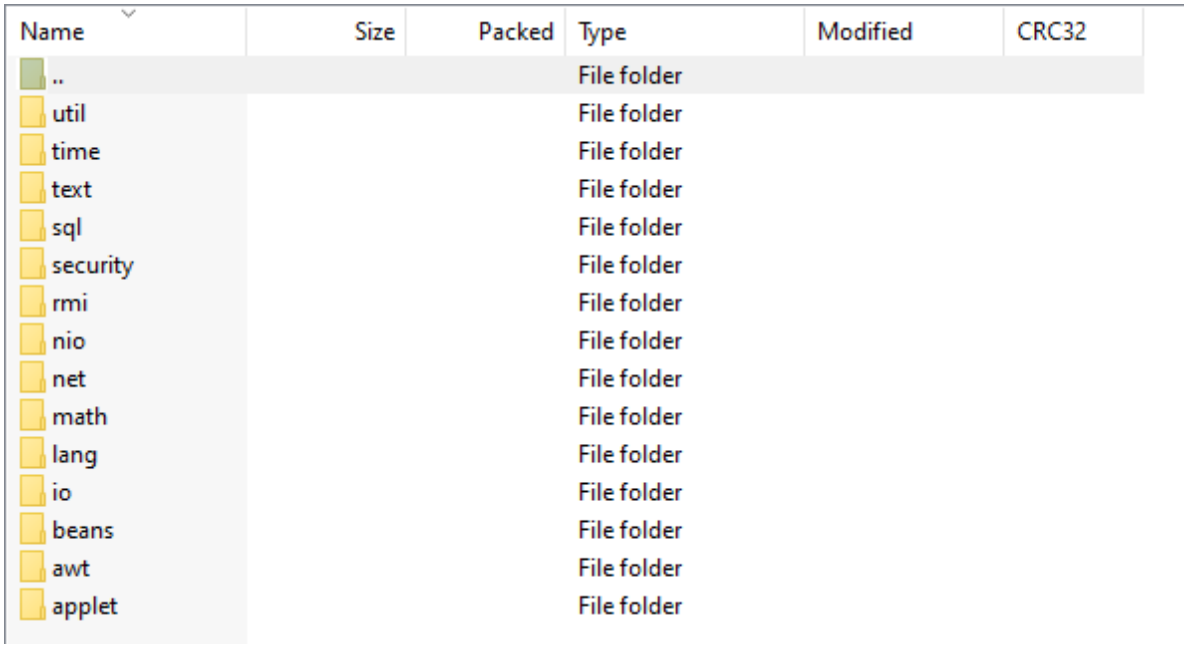


(3)打开java文件夹



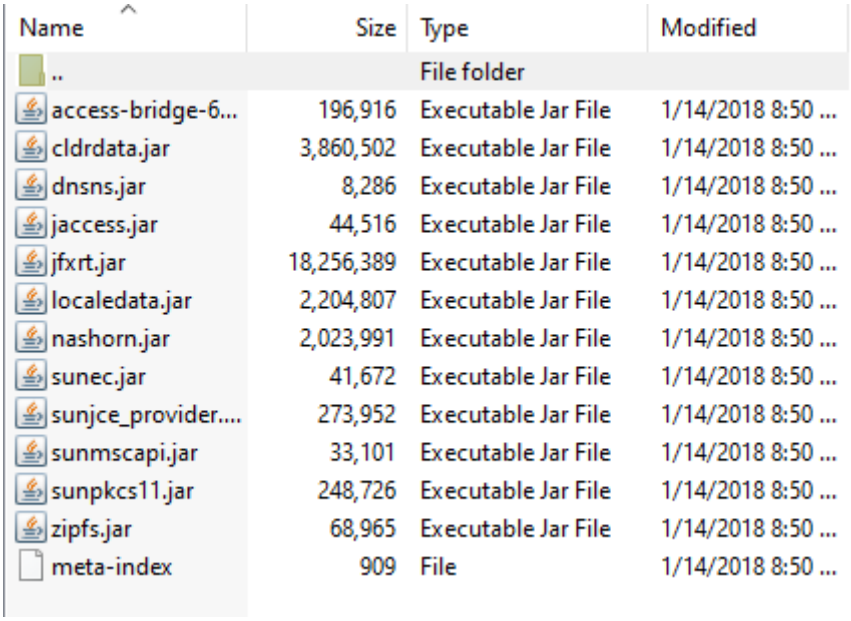
Name	Size	Packed	Type	Modified	CRC32
..			File folder		
sun			File folder		
org			File folder		
META-INF			File folder	9/14/2017 2:53 ...	
jdk			File folder		
javax			File folder		
java			File folder		
com			File folder		

(4)java所有的核心包



Name	Size	Packed	Type	Modified	CRC32
..			File folder		
util			File folder		
time			File folder		
text			File folder		
sql			File folder		
security			File folder		
rmi			File folder		
nio			File folder		
net			File folder		
math			File folder		
lang			File folder		
io			File folder		
beans			File folder		
awt			File folder		
applet			File folder		

（5）ext.jar (Extension classLoader)



Name	Size	Type	Modified
..		File folder	
access-bridge-6...	196,916	Executable Jar File	1/14/2018 8:50 ...
cldrdata.jar	3,860,502	Executable Jar File	1/14/2018 8:50 ...
dnsns.jar	8,286	Executable Jar File	1/14/2018 8:50 ...
jaccess.jar	44,516	Executable Jar File	1/14/2018 8:50 ...
jfxrt.jar	18,256,389	Executable Jar File	1/14/2018 8:50 ...
localedata.jar	2,204,807	Executable Jar File	1/14/2018 8:50 ...
nashorn.jar	2,023,991	Executable Jar File	1/14/2018 8:50 ...
sunec.jar	41,672	Executable Jar File	1/14/2018 8:50 ...
sunjce_provider....	273,952	Executable Jar File	1/14/2018 8:50 ...
sunmscapi.jar	33,101	Executable Jar File	1/14/2018 8:50 ...
sunpkcs11.jar	248,726	Executable Jar File	1/14/2018 8:50 ...
zipfs.jar	68,965	Executable Jar File	1/14/2018 8:50 ...
meta-index	909	File	1/14/2018 8:50 ...

(6)类加载器的获得

```
1  //获得类加载器
2  public class Test07 {
3      public static void main(String[] args) throws ClassNotFoundException {
4          //获得system的类加载器------>system classloader
5          ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
```

```
        System.out.println(systemClassLoader);
        //获得system父类的类加载器----->Extension classloader
        ClassLoader parent = systemClassLoader.getParent();
        System.out.println(parent);
        //Bootstrap classloader(c++)
        ClassLoader parent1 = parent.getParent();
        System.out.println(parent1);


        //测试当前类是哪个类加载器加载的

            Class<?> aClass = Class.forName("reflection.Test07");
        ClassLoader classLoader = aClass.getClassLoader();
        System.out.println(classLoader);
      //测试jdk里面的内置类的加载器

 System.out.println(Class.forName("java.lang.Object").getClassLoader());

        //如何获得系统类加载器可加载的路径
        System.out.println(System.getProperty("java.class.path"));

        /*
          双亲委派机制(Parent delegation mechanism)：java.lang.String--->先找system classloader--->extension classloader--->
          bootstrap classloader. 若是在extension Or bootstrap exists, does not work
         */
        /*
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\charsets.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\deploy.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\access-bridge-64.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\cldrdata.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\dnsns.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\jaccess.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\jfxrt.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\localedata.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\nashorn.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\sunec.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\sunjce_provider.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\sunmscapi.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\sunpkcs11.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\ext\zipfs.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\javaws.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\jce.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\jfr.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\jfxswt.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\jsse.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\management-agent.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\plugin.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\resources.jar;
        C:\Program Files\Java\jdk1.8.0_152\jre\lib\rt.jar;
        E:\Java\work-place\design-pattern\Annotation and Reflection\target\classes;
        E:\Java\IntelliJ IDEA\IntelliJ IDEA Educational Edition 2019.3.3\lib\idea_rt.jar

         */
```

```
58
59      }
60
61  }
```

通过反射获得的东西

```
1   public class Test08 {
2       public static void main(String[] args) throws Exception {
3           Class c1 = Class.forName("reflection.User");
4
5           User user  = new User();
6           Class c2 = user.getClass();
7
8           //obtain a class's name
9           System.out.println(c1.getName()); //package name + class name
10          System.out.println(c1.getSimpleName());// class name
11          System.out.println(c2.getName());
12          System.out.println("-------------------------------------");
13          //obtain class's public instance variables
14          Field[] fields = c1.getFields();// only get public instance
    variables
15          for(Field field: fields){
16              System.out.println(field + "-------->I belong to public<-------
    ---");
17          }
18          //obtain class's all instance variables
19          fields = c1.getDeclaredFields();
20          for(Field field:fields){
21              System.out.println(field);
22          }
23          //get specific instance variable
24          Field name = c1.getDeclaredField("name");// for all instance
    variables
25          System.out.println(name);
26
27          //obtain class's methods
28          Method[] methods = c1.getMethods();//obtain this class and its
    parent class's public methods
29          for(Method method:methods){
30              System.out.println(method+"--------getMethods----------");
31          }
32          methods = c1.getDeclaredMethods();//obtain this class's all(private
    protected public) methods
33
34          for(Method method:methods){
35
     System.out.println(method+"***********getDeclaredMethods**********");
36          }
37
38          //obtain the specific public method
39          Method getName = c1.getMethod("getName", null);//parameterType, not
    return type
40          Method setName = c1.getMethod("setName", String.class);// in case
    of overload methods
41          System.out.println("getName is---> "+getName+" $$$$$$$ "+"setName
    is----> "+setName);
```

```
42
43          System.out.println("");
44          System.out.println("");
45          //obtain specific constructor
46          Constructor[] constructors = c1.getConstructors();
47          Constructor[] declaredConstructors = c1.getDeclaredConstructors();
48          for (Constructor constructor : constructors) {
49
50              System.out.println(constructor+" Come from getConstructors");
51          }
52          System.out.println();
53          for (Constructor declaredConstructor : declaredConstructors) {
54              System.out.println(declaredConstructor+" From
    getDeclaredConstructors");
55          }
56
57          //obtain the specific constructor
58          Constructor constructor = c1.getConstructor(String.class,
    int.class, int.class);
59          System.out.println(constructor);
60      }
61
62
63  }
```

通过反射来创建对象

```
1   //通过反射动态的创建对象
2   public class Test09 {
3       public static void main(String[] args) throws Exception {
4           //获得class对象
5           Class c1 = Class.forName("reflection.User");
6
7           // 构造对象 Approach1
8           Object user = c1.newInstance();//调用了类的non-parameter constructor
9           System.out.println(user);
10
11          //通过构造器创建对象 Approach2
12          // Constructor declaredConstructor =
    c1.getDeclaredConstructor(String.class, int.class, int.class);
13          //  User user2 =(User) declaredConstructor.newInstance("Fred",
    40067259, 36);
14          //System.out.println(user2);
15
16          //通过反射调用methods Approach3
17          User user3 = (User) c1.newInstance();
18          //通过反射获得方法
19          Method setName = c1.getDeclaredMethod("setName", String.class);
20
21          // an object + value
22          setName.invoke(user3,"Taylor");
23          System.out.println(user3.getName());
24
25          //operate instance variables through reflection
26          User user4 = (User) c1.newInstance();
27          Field name = c1.getDeclaredField("name");
28
```

```
29
30          //we can't operate private variables directly
31          name.setAccessible(true); //shut down the access authentication
(security inspection)
32          name.set(user4,"Emma Watson");
33          System.out.println(user4.getName());
34
35
36      }
37  }
```

性能分析

```
1   //分析性能问题
2   public class Test10 {
3       public static void main(String[] args) throws Exception {
4
5           test01();
6           test02();
7           test03();
8       }
9
10      public static void test01(){
11          User user = new User();
12
13          long startTime = System.currentTimeMillis();
14
15          for (int i = 0; i < 1000000000; i++) {
16              user.getName();
17          }
18          long endTime = System.currentTimeMillis();
19
20          System.out.println("Execution time for one billion times using
regular approach: "+(endTime-startTime)+"ms");
21      }
22
23      public static void test02() throws Exception {
24          User user = new User();
25          long startTime = System.currentTimeMillis();
26          Class c1 = Class.forName("reflection.User");
27          Method getName = c1.getDeclaredMethod("getName", null);
28         // getName.setAccessible(true);
29          for (int i = 0; i < 1000000000; i++) {
30              getName.invoke(user,null);
31          }
32          long endTime = System.currentTimeMillis();
33
34          System.out.println("Execution time for one billion times using
reflection approach:----> "+(endTime-startTime)+"ms");
35
36      }
37
38      public static void test03() throws Exception {
39          User user = new User();
40          long startTime = System.currentTimeMillis();
41          Class c1 = Class.forName("reflection.User");
42          Method getName = c1.getDeclaredMethod("getName", null);
```

```
        getName.setAccessible(true);//shut down security inspection
        for (int i = 0; i < 1000000000; i++) {
            getName.invoke(user,null);
        }
        long endTime = System.currentTimeMillis();

        System.out.println("Reflection approach with no inspection:########> "+(endTime-startTime)+"ms");

    }
}
```

通过反射来操作泛型

```
//通过反射来操作泛型
public class Test11 {
    public void test01(Map<String,User> map, List<User>list){

    }

    public Map<String,User> test02(){
        System.out.println("test02");
        return null;
    }

    public static void main(String[] args) throws NoSuchMethodException {
        Method method = Test11.class.getDeclaredMethod("test01", Map.class, List.class);

        Type[] genericParameterTypes = method.getGenericParameterTypes();
        for (Type genericParameterType : genericParameterTypes) {
            System.out.println(genericParameterType);
            if(genericParameterType instanceof ParameterizedType){
                Type[] actualTypeArguments = ((ParameterizedType) genericParameterType).getActualTypeArguments();
                for (Type actualTypeArgument : actualTypeArguments) {
                    System.out.println("------"+actualTypeArgument+"------");
                }
            }
        }

        Method method2 = Test11.class.getDeclaredMethod("test02", null);
        Type genericReturnType = method2.getGenericReturnType();

        if(genericReturnType instanceof ParameterizedType){
            Type[] actualTypeArguments = ((ParameterizedType) genericReturnType).getActualTypeArguments();
            for (Type actualTypeArgument : actualTypeArguments) {
                System.out.println("******"+actualTypeArgument+"*******");
            }
        }

    }
}
```

通过反射操作注释

```
//Operate annotation through reflection
public class Test12 {
    public static void main(String[] args) throws ClassNotFoundException,
NoSuchFieldException {
        Class c1 = Class.forName("reflection.Student2");
        //通过反射得到注解
        Annotation[] annotations = c1.getAnnotations();
        for (Annotation annotation : annotations) {
            System.out.println(annotation);
        }
        //获得注解value的值
        TableFred tableFred = (TableFred)c1.getAnnotation(TableFred.class);

        String value = tableFred.value();
        System.out.println(value);

        //获得类制定的注解
        Field f = c1.getDeclaredField("name");
        FieldFred annotation = f.getAnnotation(FieldFred.class);
        System.out.println(annotation.columnName());
        System.out.println(annotation.type());
        System.out.println(annotation.length());


    }
}
@TableFred("db_student")
class Student2{
    @FieldFred(columnName = "db_id",type="int",length = 10)
    private int id;
    @FieldFred(columnName = "db_age",type="int",length = 3)
    private int age;
    @FieldFred(columnName = "db_name",type="varchar",length = 30)
    private String name;

    public Student2() {
    }

    public Student2(int id, int age, String name) {
        this.id = id;
        this.age = age;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getAge() {
        return age;
    }
```

```
55
56      public void setAge(int age) {
57          this.age = age;
58      }
59
60      public String getName() {
61          return name;
62      }
63
64      public void setName(String name) {
65          this.name = name;
66      }
67
68      @Override
69      public String toString() {
70          return "Student2{" +
71                  "id=" + id +
72                  ", age=" + age +
73                  ", name='" + name + '\'' +
74                  '}';
75      }
76  }
77
78  //类名的注解
79  @Target(ElementType.TYPE)//class
80  @Retention(RetentionPolicy.RUNTIME)// all
81  @interface TableFred{
82      String value();
83  }
84
85  //Field's annotation
86  @Target(ElementType.FIELD)
87  @Retention(RetentionPolicy.RUNTIME)
88  @interface FieldFred{
89      String columnName();
90      String type();
91      int length();
92  }
```