# 1.  JDBC

JDBC Java Database Connection

是一套api规范。

各个数据库厂商实现，提供自己的驱动包

mysql-connector-java-5.1.39-bin.jar

步骤：

1. 加载驱动

2. 获取连接

DriverManager.getConnection(url,username,password)

3. 获取PreparedStatement/Statement

4. 保存结果，ResultSet

核心类：

DriverManager

Connection

Statement

PreparedStatement

ResultSet

核心方法：

public static synchronized void registerDriver(java.sql.Driver driver)

一般不这么写，硬编码，耦合了，并且实际注册了2次驱动

DriverManager.registerDriver(new com.mysql.jdbc.Driver());

JDBC规定，在实现java.sql.Driver时，必须在静态部分完成Driver注册到DriverManager的动作。

mysql的实现：

public class Driver extends NonRegisteringDriver implements java.sql.Driver {

    static {

        try {

            java.sql.DriverManager.registerDriver(new Driver());

        } catch (SQLException E) {

            throw new RuntimeException("Can't register driver!");

        }

    }

}

所以，应该这么写：

Class.forName("com.mysql.jdbc.Driver");

JDBC规定url由3部分组成：

jdbc:${dbType}://${host}:${port}/#{dbName}

jdbc:mysql://localhost:3306/test

可以带上参数：

jdbc:mysql://localhost:3306/web08?useUnicode=true&characterEncoding=UTF8

规范的写法：

```java

@Test

public void query() {

    Connection conn = null;

    Statement stmt = null;

    ResultSet rs = null;

    try {

        conn = getConnection();

        stmt = conn.createStatement();

        String sql = "select \* from user";

        rs = stmt.executeQuery(sql);

        while(rs.next()) {

            String username = rs.getString(1);

            String password = rs.getString(2);

            System.out.println(username + ", " + password);

        }

    } catch(Exception e) {

        throw new RuntimeException(e);

    } finally {

        try {

            if(rs != null) rs.close();

            if(stmt != null) stmt.close();

            if(con != null) conn.close();

        } catch(SQLException e) {}

    }

}

```

注意关闭的顺序。

PreparedStatement

防止sql注入

性能更好

使用示例：

```java

String sql = “select \* from tab\_student where s\_number=?”;

PreparedStatement pstmt = conn.prepareStatement(sql);

pstmt.setString(1, “S\_1001”);

ResultSet rs = pstmt.executeQuery();

rs.close();

pstmt.clearParameters();

pstmt.setString(1, “S\_1002”);

rs = pstmt.executeQuery();

```

参考：

https://www.cnblogs.com/qlqwjy/p/8227665.html

# 2. 基础知识

Java注释。

1.单行注释 //

2.多行注释 /\* \*/。不可以嵌套

3.文档注释 /\*\* \*/。java特有。可以通过javadoc工具解析成html文档。不可以嵌套。 @version @author @since

Java源文件

Java源文件里可以有多个class类。

只能有一个类可以被public修饰，并且这个类名称要和源文件名称一致。不然，编译报错。

Java源文件名称和类名可以不一致，编译后的class文件名称和类名保持一致。

Java类的package包名可以和文件目录不一致（包名错误） 。编译不报错，但是字节码执行会报错，一般是找不到类。

# 泛型

“参数化类型”，类型作为一个参数进行定义。使用到的时候才传具体类型名称。

可以用在类、接口、方法中。分别被称为泛型类、泛型接口、泛型方法。

泛型只在编译阶段有效。编译完之后，泛型就被擦除了。

一个简单的例子：

```java

import java.util.ArrayList;

import java.util.List;

public class A2 {

    public static void main(String[] args) {

        List l1 = new ArrayList();

        l1.add(1);

        l1.add("aa");

        for (int i=0;i<l1.size();i++) {

            String s = (String)l1.get(i);

            System.out.println(s);

        }

    }

}

在编译的时候不报错。

List<Integer> l2 = new ArrayList();

l2.add(1);

//l2.add("aa"); // 编译时候就报错

List<String> l3 = new ArrayList();

l2.getClass().equals(l3.getClass()) // true

```

Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦除，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。

泛型信息不会进入到运行时阶段。

泛型类：

基本写法：

```java

public class Generic<T>{

    private T key;

    public Generic(T key) {

        this.key = key;

    }

    public T getKey(){

        return key;

    }

}

```

在实例化泛型类时，必须指定T的具体类型。

泛型的类型参数只能是类类型（包括自定义类），不能是简单类型。

传入的实参类型需与泛型的类型参数类型相同。

```java

Generic<Integer> g1 = new Generic<Integer>(11);

Generic<String> g2 = new Generic<String>("aaa");

```

泛型参数也可以不传入实参，此时泛型参数就没有任何限制，可以使用任何类型

```java

Generic g1 = new Generic(11);

Generic g2 = new Generic("bb");

Generic g3 = new Generic(false);

Generic g4 = new Generic(0.1);

```

不能对确切的泛型类型使用instanceof操作。

```java

if(g1 instanceof Generic<Number>){} // 编译报错

```

泛型接口：

泛型接口常被用在各种类的生产器中

基本定义：

```java

public interface Generator<T> {

    public T next();

}

```

当实现泛型接口的类，未传入泛型实参时，与泛型类的定义相同，在声明类的时候，需将泛型的声明也一起加到类中：

```java

class Test<T> implements Generator<T>{

    @Override

    public T next() {

        return null;

    }

}

// 如果不声明泛型，如：class FruitGenerator implements Generator<T>，编译器会报错："Unknown class"

```

当实现泛型接口的类，传入泛型实参时，所有使用泛型的地方都要替换成传入的实参类型：

```java

public class Test implements Generator<String> {

    private String[] fruits = new String[]{"Apple", "Banana", "Pear"};

    @Override

    public String next() {

        Random rand = new Random();

        return fruits[rand.nextInt(3)];

    }

}

```

泛型通配符

不同版本的泛型类实例是不兼容的。为了解决这个问题，引入通配符的概念。

```java

public void showKeyValue1(Generic<Number> obj){

    Log.d(obj.getKey());

}

Generic<Integer> g1 = new Generic<Integer>(123);

Generic<Number> g2 = new Generic<Number>(456);

showKeyValue(g2); // ok

showKeyValue(g1); // 编译报错，Generic<java.lang.Integer>

// cannot be applied to Generic<java.lang.Number>

修改：

public void showKeyValue1(Generic<?> obj){

    Log.d(obj.getKey());

}

```

类型通配符一般是使用？代替具体的类型实参。？是类型实参，而不是类型形参。可以把？看成所有类型的父类。是一种真实的类型。

泛型方法

泛型类，是在实例化类的时候指明泛型的具体类型。

泛型方法，是在调用方法的时候指明泛型的具体类型。

```java

public <T> T f1(Class<T> tClass)throws InstantiationException ,

  IllegalAccessException{

        T instance = tClass.newInstance();

        return instance;

}

```

<T>非常重要，可以理解为声明此方法为泛型方法。

只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。

<T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。

泛型方法基本用法：

```java

public class GenericTest {

    //这个类是个泛型类，在上面已经介绍过

    public class Generic<T>{

        private T key;

        public Generic(T key) {

            this.key = key;

        }

        //我想说的其实是这个，虽然在方法中使用了泛型，但是这并不是一个泛型方法。

        //这只是类中一个普通的成员方法，只不过他的返回值是在声明泛型类已经声明过的泛型。

        //所以在这个方法中才可以继续使用 T 这个泛型。

        public T getKey(){

            return key;

        }

        /\*\*

        \* 这个方法显然是有问题的，在编译器会给我们提示这样的错误信息"cannot reslove symbol E"

        \* 因为在类的声明中并未声明泛型E，所以在使用E做形参和返回值类型时，编译器会无法识别。

        public E setKey(E key){

            this.key = keu

        }

        \*/

    }

    /\*\*

    \* 这才是一个真正的泛型方法。

    \* 首先在public与返回值之间的<T>必不可少，这表明这是一个泛型方法，并且声明了一个泛型T

    \* 这个T可以出现在这个泛型方法的任意位置.

    \* 泛型的数量也可以为任意多个

    \*    如：public <T,K> K showKeyName(Generic<T> container){

    \*        ...

    \*        }

    \*/

    public <T> T showKeyName(Generic<T> container){

        System.out.println("container key :" + container.getKey());

        //当然这个例子举的不太合适，只是为了说明泛型方法的特性。

        T test = container.getKey();

        return test;

    }

    //这也不是一个泛型方法，这就是一个普通的方法，只是使用了Generic<Number>这个泛型类做形参而已。

    public void showKeyValue1(Generic<Number> obj){

        Log.d("泛型测试","key value is " + obj.getKey());

    }

    //这也不是一个泛型方法，这也是一个普通的方法，只不过使用了泛型通配符?

    //同时这也印证了泛型通配符章节所描述的，?是一种类型实参，可以看做为Number等所有类的父类

    public void showKeyValue2(Generic<?> obj){

        Log.d("泛型测试","key value is " + obj.getKey());

    }

    /\*\*

    \* 这个方法是有问题的，编译器会为我们提示错误信息："UnKnown class 'E' "

    \* 虽然我们声明了<T>,也表明了这是一个可以处理泛型的类型的泛型方法。

    \* 但是只声明了泛型类型T，并未声明泛型类型E，因此编译器并不知道该如何处理E这个类型。

    public <T> T showKeyName(Generic<E> container){

        ...

    }

    \*/

    /\*\*

    \* 这个方法也是有问题的，编译器会为我们提示错误信息："UnKnown class 'T' "

    \* 对于编译器来说T这个类型并未项目中声明过，因此编译也不知道该如何编译这个类。

    \* 所以这也不是一个正确的泛型方法声明。

    public void showkey(T genericObj){

    }

    \*/

    public static void main(String[] args) {

    }

}

```

泛型方法可以出现杂任何地方和任何场景中使用。但是有一种情况是非常特殊的，当泛型方法出现在泛型类中时：

```java

public class GenericFruit {

    static class Fruit{

        @Override

        public String toString() {

            return "fruit";

        }

    }

    static class Apple extends Fruit{

        @Override

        public String toString() {

            return "apple";

        }

    }

    static class Person{

        @Override

        public String toString() {

            return "Person";

        }

    }

    static class GenerateTest<T>{

        public void show\_1(T t){

            System.out.println(t.toString());

        }

        //在泛型类中声明了一个泛型方法，使用泛型E，这种泛型E可以为任意类型。可以类型与T相同，也可以不同。

        //由于泛型方法在声明的时候会声明泛型<E>，因此即使在泛型类中并未声明泛型，编译器也能够正确识别泛型方法中识别的泛型。

        public <E> void show\_3(E t){

            System.out.println(t.toString());

        }

        //在泛型类中声明了一个泛型方法，使用泛型T，注意这个T是一种全新的类型，可以与泛型类中声明的T不是同一种类型。

        public <T> void show\_2(T t){

            System.out.println(t.toString());

        }

    }

    public static void main(String[] args) {

        Apple apple = new Apple();

        Person person = new Person();

        GenerateTest<Fruit> generateTest = new GenerateTest<Fruit>();

        //apple是Fruit的子类，所以这里可以

        generateTest.show\_1(apple);

        //编译器会报错，因为泛型类型实参指定的是Fruit，而传入的实参类是Person

        //generateTest.show\_1(person);

        //使用这两个方法都可以成功

        generateTest.show\_2(apple);

        generateTest.show\_2(person);

        //使用这两个方法也都可以成功

        generateTest.show\_3(apple);

        generateTest.show\_3(person);

    }

}

```

泛型方法与可变参数

```java

public <T> void printMsg(T... args){

    for(T t : args){

        Log.d("泛型测试","t is " + t);

    }

}

printMsg("111",222,"aaaa","2323.4",55.55);

```

静态方法与泛型

静态方法无法访问类上定义的泛型；如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。

即：如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法。

```java

public class StaticGenerator<T> {

    /\*\*

     \* 如果在类中定义使用泛型的静态方法，需要添加额外的泛型声明（将这个方法定义成泛型方法）

     \* 即使静态方法要使用泛型类中已经声明过的泛型也不可以。

     \* 如：public static void show(T t){..},此时编译器会提示错误信息：

          "StaticGenerator cannot be refrenced from static context"

     \*/

    public static <T> void show(T t){

    }

}

```

泛型方法能使方法独立于类而产生变化，所以能用泛型方法，就要去用。

泛型上下边界

```java

public void showKeyValue1(Generic<? extends Number> obj){

    Log.d("泛型测试","key value is " + obj.getKey());

}

public void showKeyValue2(Generic<? super Number> obj){

    Log.d("泛型测试","key value is " + obj.getKey());

}

Generic<String> generic1 = new Generic<String>("11111");

Generic<Integer> generic2 = new Generic<Integer>(2222);

Generic<Float> generic3 = new Generic<Float>(2.4f);

Generic<Double> generic4 = new Generic<Double>(2.56);

//这一行代码编译器会提示错误，因为String类型并不是Number类型的子类

//showKeyValue1(generic1);

showKeyValue1(generic2);

showKeyValue1(generic3);

showKeyValue1(generic4);

public class Generic<T extends Number>{

    private T key;

    public Generic(T key) {

        this.key = key;

    }

    public T getKey(){

        return key;

    }

}

//这一行代码也会报错，因为String不是Number的子类

Generic<String> generic1 = new Generic<String>("11111");

```

泛型的边界必须添加在泛型声明的地方，比如：

```java

//在泛型方法中添加上下边界限制的时候，必须在权限声明与返回值之间的<T>上添加上下边界，即在泛型声明的时候添加

//public <T> T showKeyName(Generic<T extends Number> container)，编译器会报错："Unexpected bound"

public <T extends Number> T showKeyName(Generic<T> container){

    System.out.println("container key :" + container.getKey());

    T test = container.getKey();

    return test;

}

```

泛型的上下边界添加，必须与泛型的声明在一起。

泛型数组

在java中，不能创建一个确切的泛型类型的数组。

List<String>[] ls = new ArrayList<String>[10]; // 编译报错， Generic Array Create

List<?>[] ls = new ArrayList<?>[10];  // ok

List<String>[] ls = new ArrayList[10]; // ok

```java

List<String>[] lsa = new List<String>[10]; // 假如这样可以用

Object o = lsa;

Object[] oa = (Object[]) o;

List<Integer> li = new ArrayList<Integer>();

li.add(new Integer(3));

oa[1] = li; // Unsound, but passes run time store check

String s = lsa[1].get(0); // Run-time error: ClassCastException.

上面这种情况，编译时完全没有问题，但是运行时就报错了。这和类型安全是不符合的。

```

数组的类型不可以是类型变量，除非是采用通配符的方式，因为对于通配符的方式，最后取出数据是要做显式的类型转换的。

```java

List<?>[] lsa = new List<?>[10]; // OK, array of unbounded wildcard type.

Object o = lsa;

Object[] oa = (Object[]) o;

List<Integer> li = new ArrayList<Integer>();

li.add(new Integer(3));

oa[1] = li; // Correct.

Integer i = (Integer) lsa[1].get(0); // OK

```

一句话，能用泛型就用泛型。

参考

https://www.cnblogs.com/coprince/p/8603492.html

# 注解

@Annotation

作用：

格式检查

减少配置

减少重复工作

大部分框架都使用注解来简化代码并提升编码效率。

javadoc中的@author、@version、@param、@return、@deprecated、@hide、@throws、@exception、@see是标记，不是注解

一个方法上可以拥有多个不同的注解

@Test是一种标记注解，起标记作用，运行时告诉测试框架该方法为测试方法。

```java

/\*\*

 \* Indicates that the annotated method is a test method

 \* Use only on parameterless static methods

\*/

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Test {

}

```

元注解，标记其他注解的注解。

几种元注解：

1. @Target

约束注解可以应用的地方。

当注解未指定Target值时，则此注解可以用于任何元素之上。

多个值使用{}包含并用逗号隔开

@Target(value={CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})

```java

public enum ElementType {

    /\*\*标明该注解可以用于类、接口（包括注解类型）或enum声明\*/

    TYPE,

    /\*\* 标明该注解可以用于字段(域)声明，包括enum实例 \*/

    FIELD,

    /\*\* 标明该注解可以用于方法声明 \*/

    METHOD,

    /\*\* 标明该注解可以用于参数声明 \*/

    PARAMETER,

    /\*\* 标明注解可以用于构造函数声明 \*/

    CONSTRUCTOR,

    /\*\* 标明注解可以用于局部变量声明 \*/

    LOCAL\_VARIABLE,

    /\*\* 标明注解可以用于注解声明(应用于另一个注解上)\*/

    ANNOTATION\_TYPE,

    /\*\* 标明注解可以用于包声明 \*/

    PACKAGE,

    /\*\*

     \* 标明注解可以用于类型参数声明（1.8新加入）

     \* @since 1.8

     \*/

    TYPE\_PARAMETER,

    /\*\*

     \* 类型使用声明（1.8新加入)

     \* @since 1.8

     \*/

    TYPE\_USE

}

```

2. @Retention

用来约束注解的生命周期

分别有三个值，源码级别（source），类文件级别（class）或者运行时级别（runtime）

SOURCE：注解将被编译器丢弃（该类型的注解信息只会保留在源码里，源码经过编译后，注解信息会被丢弃，不会保留在编译好的class文件里）

CLASS：注解在class文件中可用，但会被VM丢弃（该类型的注解信息会保留在源码里和class文件里，在执行的时候，不会加载到虚拟机中），请注意，当注解未定义Retention值时，默认值是CLASS，如Java内置注解，@Override、@Deprecated、@SuppressWarnning等

RUNTIME：注解信息将在运行期(JVM)也保留，因此可以通过反射机制读取注解的信息（源码、class文件和执行的时候都有注解的信息），如SpringMvc中的@Controller、@Autowired、@RequestMapping等。

3. @Documented

被修饰的注解会生成到javadoc中

```java

@Documented

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface DocumentA {

}

//没有使用@Documented

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface DocumentB {

}

//使用注解

@DocumentA

@DocumentB

public class DocumentDemo {

    public void A(){

    }

}

$ javadoc DocumentDemo.java DocumentA.java DocumentB.java

```

4. @Inherited

可以让注解被继承，但这并不是真的继承，只是通过使用@Inherited，可以让子类Class对象使用getAnnotations()获取父类被@Inherited修饰的注解

```java

@Inherited

@Documented

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface DocumentA {

}

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface DocumentB {

}

@DocumentA

class A{ }

class B extends A{ }

@DocumentB

class C{ }

class D extends C{ }

public class DocumentDemo {

    public static void main(String... args){

        A instanceA=new B();

        System.out.println("已使用的@Inherited注解:"+Arrays.toString(instanceA.getClass().getAnnotations()));

        C instanceC = new D();

        System.out.println("没有使用的@Inherited注解:"+Arrays.toString(instanceC.getClass().getAnnotations()));

    }

    /\*\*

     \* 运行结果:

     已使用的@Inherited注解:[@com.zejian.annotationdemo.DocumentA()]

     没有使用的@Inherited注解:[]

     \*/

}

```

注解元素及其数据类型

在自定义注解中，一般都会包含一些元素以表示某些值，方便处理器使用。

注解类中的方法只能用public或者默认这两个访问权修饰，不写public就是默认

```java

@Target(ElementType.TYPE)//只能应用于类上

@Retention(RetentionPolicy.RUNTIME)//保存到运行时

public @interface DBTable {

    String name() default "";

}

//在类上使用该注解

@DBTable(name = "MEMBER")

public class Member {

    //.......

}

```

注解支持的元素类型：

所有基本类型（int,float,boolean,byte,double,char,long,short）

String

Class

enum

Annotation

上述类型的数组

注意：

声明注解元素时可以使用基本类型但不允许使用任何包装类型。

注解也可以作为元素的类型，也就是嵌套注解。

```java

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

public class A13 {

}

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@interface Reference{

    boolean next() default false;

}

//枚举类型

enum Status {FIXED,NORMAL};

@interface AnnotationElementDemo {

    //声明枚举

    Status status() default Status.FIXED;

    //布尔类型

    boolean showSupport() default false;

    //String类型

    String name()default "";

    //class类型

    Class<?> testCase() default Void.class;

    //注解嵌套

    Reference reference() default @Reference(next=true);

    //数组类型

    long[] value();

}

```

编译器对默认值的限制

元素必须有确定的值。要么具有默认值，要么在使用注解时提供元素的值。

无论是在源代码中声明，还是在注解接口中定义默认值，都不能null作为值。

注解不支持继承

注解在编译后，编译器会自动继承java.lang.annotation.Annotation接口

```java

@Target(ElementType.TYPE)//只能应用于类上

@Retention(RetentionPolicy.RUNTIME)//保存到运行时

public @interface DBTable {

    String name() default "";

}

反编译

import java.lang.annotation.Annotation;

public interface DBTable extends Annotation {

    public abstract String name();

}

```

元素名称为value是可以缺省赋值

```java

//定义注解

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@interface IntegerVaule{

    int value() default 0;

    String name() default "";

}

//使用注解

public class QuicklyWay {

    //当只想给value赋值时,可以使用以下快捷方式

    @IntegerVaule(20)

    public int age;

    //当name也需要赋值时必须采用key=value的方式赋值

    @IntegerVaule(value = 10000, name = "MONEY")

    public int money;

}

```

Java内置注解，主要有3个

1. @Override：用于标明此方法覆盖了父类的方法

```java

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.SOURCE)

public @interface Override {

}

```

2. @Deprecated：用于标明已经过时的方法或类

```java

@Documented

@Retention(RetentionPolicy.RUNTIME)

@Target(value={CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})

public @interface Deprecated {

}

```

3. @SuppressWarnnings：用于有选择的关闭编译器对类、方法、成员变量、变量初始化的警告

```java

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE})

@Retention(RetentionPolicy.SOURCE)

public @interface SuppressWarnings {

    String[] value();

}

内部有一个String数组，主要接收值如下：

deprecation：使用了不赞成使用的类或方法时的警告；

unchecked：执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型;

fallthrough：当 Switch 程序块直接通往下一种情况而没有 Break 时的警告;

path：在类路径、源文件路径等中有不存在的路径时的警告;

serial：当在可序列化的类上缺少 serialVersionUID 定义时的警告;

finally：任何 finally 子句不能正常完成时的警告;

all：关于以上所有情况的警告。

```

使用：

```java

//注明该类已过时，不建议使用

@Deprecated

class A{

    public void A(){ }

    //注明该方法已过时，不建议使用

    @Deprecated()

    public void B(){ }

}

class B extends A{

    @Override //标明覆盖父类A的A方法

    public void A() {

        super.A();

    }

    //去掉检测警告

    @SuppressWarnings({"uncheck","deprecation"})

    public void C(){ }

    //去掉检测警告

    @SuppressWarnings("uncheck")

    public void D(){ }

}

```

反射只能获取到@Retention(RetentionPolicy.RUNTIME)的注解。

注解与反射机制

为了运行时能准确获取到注解的相关信息，Java在java.lang.reflect 反射包下新增了AnnotatedElement接口，它主要用于表示目前正在 VM 中运行的程序中已使用注解的元素，通过该接口提供的方法可以利用反射技术地读取注解的信息，如反射包的Constructor类、Field类、Method类、Package类和Class类都实现了AnnotatedElement接口。

Class：类的Class对象定义

Constructor：代表类的构造器定义

Field：代表类的成员变量定义

Method：代表类的方法定义

Package：代表类的包定义

AnnotatedElement中相关的API方法，以上5个类都实现以下的方法：

返回值                     方法名称

<A extends Annotation>      getAnnotation(Class<A> annotationClass) 该元素如果存在指定类型的注解，则返回这些注解，否则返回 null。

Annotation[]                getAnnotations()    返回此元素上存在的所有注解，包括从父类继承的

boolean                     isAnnotationPresent(Class<? extends Annotation> annotationClass)    如果指定类型的注解存在于此元素上，则返回 true，否则返回 false。

Annotation[]                getDeclaredAnnotations()    返回直接存在于此元素上的所有注解，注意，不包括父类的注解，调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响，没有则返回长度为0的数组

```java

import java.lang.annotation.Annotation;

import java.util.Arrays;

@DocumentA

class A{ }

//继承了A类

@DocumentB

public class DocumentDemo extends A{

    public static void main(String... args){

        Class<?> clazz = DocumentDemo.class;

        //根据指定注解类型获取该注解

        DocumentA documentA=clazz.getAnnotation(DocumentA.class);

        System.out.println("A:"+documentA);

        //获取该元素上的所有注解，包含从父类继承

        Annotation[] an= clazz.getAnnotations();

        System.out.println("an:"+ Arrays.toString(an));

        //获取该元素上的所有注解，但不包含继承！

        Annotation[] an2=clazz.getDeclaredAnnotations();

        System.out.println("an2:"+ Arrays.toString(an2));

        //判断注解DocumentA是否在该元素上

        boolean b=clazz.isAnnotationPresent(DocumentA.class);

        System.out.println("b:"+b);

        /\*\*

         \* 执行结果:

         A:@com.zejian.annotationdemo.DocumentA()

         an:[@com.zejian.annotationdemo.DocumentA(), @com.zejian.annotationdemo.DocumentB()]

         an2:@com.zejian.annotationdemo.DocumentB()

         b:true

         \*/

    }

}

```

运行时注解处理器

通过一个简单的ORM例子模拟Hibernate的原理。

```java

package com.wh.p1;

import java.lang.annotation.Annotation;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

import java.lang.reflect.Field;

import java.util.ArrayList;

import java.util.List;

/\*\*

 \* 表注解

 \*/

@Target(ElementType.TYPE) // 只能应用于类上

@Retention(RetentionPolicy.RUNTIME) // 保存到运行时

@interface DBTable {

    String name() default "";

}

/\*\*

 \* 注解Integer类型的字段

 \*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@interface SQLInteger {

    String name() default ""; // 字段对应数据库表列名

    Constraints constraint() default @Constraints; // 嵌套注解

}

/\*\*

 \* 注解String类型的字段

 \*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@interface SQLString {

    String name() default ""; // 字段对应数据库表列名

    int value() default 0; // 列类型分配的长度，如varchar(30)的30

    Constraints constraint() default @Constraints; // 嵌套注解

}

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@interface Constraints {

    boolean primaryKey() default false; // 判断是否作为主键约束

    boolean allowNull() default false; // 判断是否允许为null

    boolean unique() default false; // 判断是否唯一

}

/\*\*

 \* 运行时注解处理器，构造表创建语句

 \*/

class TableCreator {

    public static String createTableSql(String className) throws ClassNotFoundException {

        Class<?> cls = Class.forName(className);

        DBTable dbTable = cls.getAnnotation(DBTable.class);

        // 如果没有表注解，直接返回

        if(dbTable == null) {

            System.out.println(

                    "No DBTable annotations in class " + className);

            return null;

        }

        String tableName = dbTable.name();

        // If the name is empty, use the Class name:

        if(tableName.length() < 1)

            tableName = cls.getName().toUpperCase();

        List<String> columnDefs = new ArrayList<String>();

        // 通过Class类API获取到所有成员字段

        for (Field field : cls.getDeclaredFields()) {

            String columnName = null;

            // 获取字段上的注解

            Annotation[] anns = field.getDeclaredAnnotations();

            if(anns.length < 1)

                continue; // Not a db table column

            // 判断注解类型

            if(anns[0] instanceof SQLInteger) {

                SQLInteger sInt = (SQLInteger) anns[0];

                // 获取字段对应列名称，如果没有就是使用字段名称替代

                if (sInt.name().length() < 1)

                    columnName = field.getName().toUpperCase();

                else

                    columnName = sInt.name();

                // 构建语句

                columnDefs.add(columnName + " INT" +

                        getConstraints(sInt.constraint()));

            }

            // 判断String类型

            if(anns[0] instanceof SQLString) {

                SQLString sString = (SQLString) anns[0];

                // Use field name if name not specified.

                if (sString.name().length() < 1)

                    columnName = field.getName().toUpperCase();

                else

                    columnName = sString.name();

                columnDefs.add(columnName + " VARCHAR(" +

                        sString.value() + ")" +

                        getConstraints(sString.constraint()));

            }

        }

        // 数据库表构建语句

        StringBuilder createCommand = new StringBuilder(

                "CREATE TABLE " + tableName + "(");

        for(String columnDef : columnDefs)

            createCommand.append("\n    " + columnDef + ",");

        // Remove trailing comma

        String tableCreate = createCommand.substring(

                0, createCommand.length() - 1) + ");";

        return tableCreate;

    }

    /\*\*

     \* 判断该字段是否有其他约束

     \*/

    static String getConstraints(Constraints con) {

        String constraints = "";

        if(!con.allowNull())

            constraints += " NOT NULL";

        if(con.primaryKey())

            constraints += " PRIMARY KEY";

        if(con.unique())

            constraints += " UNIQUE";

        return constraints;

    }

}

/\*\*

 \* 数据库表Member对应实例类bean

 \*/

@DBTable(name = "MEMBER")

class Member {

    //主键ID

    @SQLString(name = "ID", value = 50, constraint = @Constraints(primaryKey = true))

    private String id;

    @SQLString(name = "NAME" , value = 30)

    private String name;

    @SQLInteger(name = "AGE")

    private int age;

    @SQLString(name = "DESCRIPTION" ,value = 150 , constraint = @Constraints(allowNull = true))

    private String description;

}

/\*\*

 \* Hibernate的原理

 \*/

public class A14 {

    public static void main(String[] args) throws ClassNotFoundException {

        String[] arg={"com.wh.p1.Member"};

        for(String className : arg) {

            System.out.println("Table Creation SQL for " +

                    className + " is :\n" + TableCreator.createTableSql(className));

        }

    }

    /\*\*

     \* Table Creation SQL for com.wh.p1.Member is :

     \* CREATE TABLE MEMBER(

     \*     ID VARCHAR(50) NOT NULL PRIMARY KEY,

     \*     NAME VARCHAR(30) NOT NULL,

     \*     AGE INT NOT NULL,

     \*     DESCRIPTION VARCHAR(150));

     \*/

}

```

Java8新增注解

1. 元注解@Repeatable

表示在同一个位置重复相同的注解。在没有该注解前，一般是无法在同一个类型上使用相同的注解的

```java

//Java8前无法这样使用

@FilterPath("/web/update")

@FilterPath("/web/add")

public class A {}

如果是想实现类似的功能，我们需要在定义@FilterPath注解时定义一个数组元素接收多个值如下：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface FilterPath {

    String [] value();

}

//使用

@FilterPath({"/update","/add"})

public class A { }

//使用Java8新增@Repeatable原注解

@Target({ElementType.TYPE,ElementType.FIELD,ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Repeatable(FilterPaths.class)//参数指明接收的注解class

public @interface FilterPath {

    String  value();

}

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@interface FilterPaths {

    FilterPath[] value();

}

//使用案例

@FilterPath("/web/update")

@FilterPath("/web/add")

@FilterPath("/web/delete")

class AA{ }

```

为了处理新增注解，Java8还在AnnotatedElement接口新增了getDeclaredAnnotationsByType() 和 getAnnotationsByType()两个方法并在接口给出了默认实现。在指定@Repeatable的注解时，可以通过这两个方法获取到注解相关信息。

但旧版API中的getDeclaredAnnotation()和getAnnotation()是不对@Repeatable注解的处理的(除非该注解没有在同一个声明上重复出现)。

getDeclaredAnnotationsByType方法获取到的注解不包括父类，其实当 getAnnotationsByType()方法调用时，其内部先执行了getDeclaredAnnotationsByType方法，只有当前类不存在指定注解时，getAnnotationsByType()才会继续从其父类寻找，如果@FilterPath和@FilterPaths没有使用了@Inherited的话，仍然无法获取。

```java

@Target({ElementType.TYPE,ElementType.FIELD,ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Repeatable(FilterPaths.class)

public @interface FilterPath {

    String  value();

}

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@interface FilterPaths {

    FilterPath[] value();

}

@FilterPath("/web/list")

class CC { }

//使用案例

@FilterPath("/web/update")

@FilterPath("/web/add")

@FilterPath("/web/delete")

class AA extends CC{

    public static void main(String[] args) {

        Class<?> clazz = AA.class;

        //通过getAnnotationsByType方法获取所有重复注解

        FilterPath[] annotationsByType = clazz.getAnnotationsByType(FilterPath.class);

        FilterPath[] annotationsByType2 = clazz.getDeclaredAnnotationsByType(FilterPath.class);

        if (annotationsByType != null) {

            for (FilterPath filter : annotationsByType) {

                System.out.println("1:"+filter.value());

            }

        }

        System.out.println("-----------------");

        if (annotationsByType2 != null) {

            for (FilterPath filter : annotationsByType2) {

                System.out.println("2:"+filter.value());

            }

        }

        System.out.println("使用getAnnotation的结果:"+clazz.getAnnotation(FilterPath.class));

        /\*\*

         \* 执行结果(当前类拥有该注解FilterPath,则不会从CC父类寻找)

         1:/web/update

         1:/web/add

         1:/web/delete

         -----------------

         2:/web/update

         2:/web/add

         2:/web/delete

         使用getAnnotation的结果:null

         \*/

    }

}

```

另外一种情况：

注意定义@FilterPath和@FilterPath时必须指明@Inherited，getAnnotationsByType方法否则依旧无法从父类获取@FilterPath注解

```java

@Target({ElementType.TYPE,ElementType.FIELD,ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Inherited //添加可继承元注解

@Repeatable(FilterPaths.class)

public @interface FilterPath {

    String  value();

}

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Inherited //添加可继承元注解

@interface FilterPaths {

    FilterPath[] value();

}

@FilterPath("/web/list")

@FilterPath("/web/getList")

class CC { }

//AA上不使用@FilterPath注解,getAnnotationsByType将会从父类查询

class AA extends CC{

    public static void main(String[] args) {

        Class<?> clazz = AA.class;

        //通过getAnnotationsByType方法获取所有重复注解

        FilterPath[] annotationsByType = clazz.getAnnotationsByType(FilterPath.class);

        FilterPath[] annotationsByType2 = clazz.getDeclaredAnnotationsByType(FilterPath.class);

        if (annotationsByType != null) {

            for (FilterPath filter : annotationsByType) {

                System.out.println("1:"+filter.value());

            }

        }

        System.out.println("-----------------");

        if (annotationsByType2 != null) {

            for (FilterPath filter : annotationsByType2) {

                System.out.println("2:"+filter.value());

            }

        }

        System.out.println("使用getAnnotation的结果:"+clazz.getAnnotation(FilterPath.class));

        /\*\*

         \* 执行结果(当前类没有@FilterPath,getAnnotationsByType方法从CC父类寻找)

         1:/web/list

         1:/web/getList

         -----------------

         使用getAnnotation的结果:null

         \*/

    }

}

```

看下源码：

```java

package java.lang.reflect;

public interface AnnotatedElement {

    //接口默认实现方法

    default <T extends Annotation> T[] getAnnotationsByType(Class<T> annotationClass) {

        //先调用getDeclaredAnnotationsByType方法

        T[] result = getDeclaredAnnotationsByType(annotationClass);

        //判断当前类获取到的注解数组是否为0

        if (result.length == 0 && this instanceof Class &&

        //判断定义注解上是否使用了@Inherited元注解

        AnnotationType.getInstance(annotationClass).isInherited()) { // Inheritable

                //从父类获取

            Class<?> superClass = ((Class<?>) this).getSuperclass();

        if (superClass != null) {

            result = superClass.getAnnotationsByType(annotationClass);

            }

        }

        return result;

    }

}

```

新增的两种ElementType

在Java8中ElementType新增两个枚举成员，TYPE\_PARAMETER 和 TYPE\_USE。

在Java8前注解只能标注在一个声明(如字段、类、方法)上，Java8后，新增的TYPE\_PARAMETER可以用于标注类型参数，而TYPE\_USE则可以用于标注任意类型(不包括class)。

```java

//TYPE\_PARAMETER 标注在类型参数上

class D<@Parameter T> { }

//TYPE\_USE则可以用于标注任意类型(不包括class)

//用于父类或者接口

class Image implements @Rectangular Shape { }

//用于构造函数

new @Path String("/usr/bin")

//用于强制转换和instanceof检查,注意这些注解中用于外部工具，它们不会对类型转换或者instanceof的检查行为带来任何影响。

String path=(@Path String)input;

if(input instanceof @Path String)

//用于指定异常

public Person read() throws @Localized IOException.

//用于通配符绑定

List<@ReadOnly ? extends Person>

List<? extends @ReadOnly Person>

@NotNull String.class //非法，不能标注class

import java.lang.@NotNull String //非法，不能标注import

```

TYPE\_USE，类型注解用来支持在Java的程序中做强类型检查，配合第三方插件工具（如Checker Framework），可以在编译期检测出runtime error（如UnsupportedOperationException、NullPointerException异常），避免异常延续到运行期才发现，从而提高代码质量，这就是类型注解的主要作用。

Java 8新增加了两个注解的元素类型ElementType.TYPE\_USE 和ElementType.TYPE\_PARAMETER ，通过它们，我们可以把注解应用到各种新场合中。

# 枚举

类类型。

关键字是enum，与class关键字类似。

枚举表示的类型其取值是必须有限的。

基本使用：

```java

public class EnumDemo {

    public static void main(String[] args){

        //直接引用

        Day day =Day.MONDAY;

    }

}

enum Day {

    MONDAY, TUESDAY, WEDNESDAY,

    THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

```

枚举实现原理

编译器会自动帮助我们生成一个与枚举相关的类。这个类继承了Java API中的java.lang.Enum类。

```

$ ls

EnumDemo.java

$ javac EnumDemo.java

$ ls

Day.class  EnumDemo.class  EnumDemo.java

```

```

反编译

$ javap Day.class

final class Day extends Enum

{

    //编译器为我们添加的静态的values()方法

    public static Day[] values()

    {

        return (Day[])$VALUES.clone();

    }

    //编译器为我们添加的静态的valueOf()方法，注意间接调用了Enum也类的valueOf方法

    public static Day valueOf(String s)

    {

        return (Day)Enum.valueOf(com/zejian/enumdemo/Day, s);

    }

    //私有构造函数

    private Day(String s, int i)

    {

        super(s, i);

    }

     //前面定义的7种枚举实例

    public static final Day MONDAY;

    public static final Day TUESDAY;

    public static final Day WEDNESDAY;

    public static final Day THURSDAY;

    public static final Day FRIDAY;

    public static final Day SATURDAY;

    public static final Day SUNDAY;

    private static final Day $VALUES[];

    static

    {

        //实例化枚举实例

        MONDAY = new Day("MONDAY", 0);

        TUESDAY = new Day("TUESDAY", 1);

        WEDNESDAY = new Day("WEDNESDAY", 2);

        THURSDAY = new Day("THURSDAY", 3);

        FRIDAY = new Day("FRIDAY", 4);

        SATURDAY = new Day("SATURDAY", 5);

        SUNDAY = new Day("SUNDAY", 6);

        $VALUES = (new Day[] {

            MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

        });

    }

}

```

枚举的常见方法

Enum抽象类常见方法

Enum是所有Java语言枚举类型的公共基本类（注意Enum是抽象类），以下是它的常见方法：

int compareTo(E o)                  比较此枚举与指定对象的顺序

boolean equals(Object other)        当指定对象等于此枚举常量时，返回 true。

Class<?> getDeclaringClass()        返回与此枚举常量的枚举类型相对应的 Class 对象

String name()                       返回此枚举常量的名称，在其枚举声明中对其进行声明

int ordinal()                       返回枚举常量的序数（它在枚举声明中的位置，其中初始常量序数为零）

String toString()                   返回枚举常量的名称，它包含在声明中

static<T extends Enum<T>> T static valueOf(Class<T> enumType, String name)  返回带指定名称的指定枚举类型的枚举常量。

ordinal()方法，该方法获取的是枚举变量在枚举类中声明的顺序，下标从0开始。

如果MONDAY的声明位置发生变化，那么ordinal方法获取到的值也随之变化，注意在大多数情况下我们都不应该首先使用该方法，这个方法是为实现EnumSet和EnumMap准备的。

compareTo(E o)方法则是比较枚举的大小，注意其内部实现是根据每个枚举的ordinal值大小进行比较的。

name()方法与toString()几乎是等同的，都是输出变量的字符串形式。

valueOf(Class<T> enumType, String name)方法则是根据枚举类的Class对象和枚举名称获取枚举常量，注意该方法是静态的。

使用：

```java

enum Day {

    MONDAY, TUESDAY, WEDNESDAY,

    THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

public class EnumDemo {

    public static void main(String[] args){

        //创建枚举数组

        Day[] days=new Day[]{Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY,

                Day.THURSDAY, Day.FRIDAY, Day.SATURDAY, Day.SUNDAY};

        for (int i = 0; i <days.length ; i++) {

            System.out.println("day["+i+"].ordinal():"+days[i].ordinal());

        }

        System.out.println("-------------------------------------");

        //通过compareTo方法比较,实际上其内部是通过ordinal()值比较的

        System.out.println("days[0].compareTo(days[1]):"+days[0].compareTo(days[1]));

        System.out.println("days[0].compareTo(days[1]):"+days[0].compareTo(days[2]));

        //获取该枚举对象的Class对象引用,当然也可以通过getClass方法

        Class<?> clazz = days[0].getDeclaringClass();

        System.out.println("clazz:"+clazz);

        System.out.println("-------------------------------------");

        //name()

        System.out.println("days[0].name():"+days[0].name());

        System.out.println("days[1].name():"+days[1].name());

        System.out.println("days[2].name():"+days[2].name());

        System.out.println("days[3].name():"+days[3].name());

        System.out.println("-------------------------------------");

        System.out.println("days[0].toString():"+days[0].toString());

        System.out.println("days[1].toString():"+days[1].toString());

        System.out.println("days[2].toString():"+days[2].toString());

        System.out.println("days[3].toString():"+days[3].toString());

        System.out.println("-------------------------------------");

        Day d=Enum.valueOf(Day.class,days[0].name());

        Day d2=Day.valueOf(Day.class,days[0].name());

        System.out.println("d:"+d);

        System.out.println("d2:"+d2);

    }

 /\*\*

 执行结果:

   day[0].ordinal():0

   day[1].ordinal():1

   day[2].ordinal():2

   day[3].ordinal():3

   day[4].ordinal():4

   day[5].ordinal():5

   day[6].ordinal():6

   -------------------------------------

   days[0].compareTo(days[1]):-1

   days[0].compareTo(days[1]):-2

   clazz:class com.zejian.enumdemo.Day

   -------------------------------------

   days[0].name():MONDAY

   days[1].name():TUESDAY

   days[2].name():WEDNESDAY

   days[3].name():THURSDAY

   -------------------------------------

   days[0].toString():MONDAY

   days[1].toString():TUESDAY

   days[2].toString():WEDNESDAY

   days[3].toString():THURSDAY

   -------------------------------------

   d:MONDAY

   d2:MONDAY

   \*/

}

```

注意：

Enum类内部会有一个构造函数，该构造函数只能有编译器调用，我们是无法手动操作的。

部分源码：

```java

package java.lang;

public abstract class Enum<E extends Enum<E>>

        implements Comparable<E>, Serializable {

    private final String name; //枚举字符串名称

    public final String name() {

        return name;

    }

    private final int ordinal;//枚举顺序值

    public final int ordinal() {

        return ordinal;

    }

    //枚举的构造方法，只能由编译器调用

    protected Enum(String name, int ordinal) {

        this.name = name;

        this.ordinal = ordinal;

    }

    public String toString() {

        return name;

    }

    public final boolean equals(Object other) {

        return this==other;

    }

    //比较的是ordinal值

    public final int compareTo(E o) {

        Enum<?> other = (Enum<?>)o;

        Enum<E> self = this;

        if (self.getClass() != other.getClass() && // optimization

            self.getDeclaringClass() != other.getDeclaringClass())

            throw new ClassCastException();

        return self.ordinal - other.ordinal;//根据ordinal值比较大小

    }

    @SuppressWarnings("unchecked")

    public final Class<E> getDeclaringClass() {

        //获取class对象引用，getClass()是Object的方法

        Class<?> clazz = getClass();

        //获取父类Class对象引用

        Class<?> zuper = clazz.getSuperclass();

        return (zuper == Enum.class) ? (Class<E>)clazz : (Class<E>)zuper;

    }

    public static <T extends Enum<T>> T valueOf(Class<T> enumType,

                                                String name) {

        //enumType.enumConstantDirectory()获取到的是一个map集合，key值就是name值，value则是枚举变量值

        //enumConstantDirectory是class对象内部的方法，根据class对象获取一个map集合的值

        T result = enumType.enumConstantDirectory().get(name);

        if (result != null)

            return result;

        if (name == null)

            throw new NullPointerException("Name is null");

        throw new IllegalArgumentException(

            "No enum constant " + enumType.getCanonicalName() + "." + name);

    }

    //.....省略其他没用的方法

}

```

编译后生成的Values()与ValueOf()

values()和valueOf(String name)是编译器生成的static方法

```java

Day[] days2 = Day.values();

System.out.println("day2:"+Arrays.toString(days2));

Day day = Day.valueOf("MONDAY");

System.out.println("day:"+day);

/\*\*

 输出结果:

 day2:[MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]

 day:MONDAY

 \*/

// 正常使用

Day[] ds=Day.values();

// 向上转型Enum

Enum e = Day.MONDAY;

// 无法调用,没有此方法

//e.values();

```

枚举与Class对象

虽然Enum类没有values()，但是通过Enum.class对象可以获取到所有枚举对象。

Class对象中存在如下方法：

T[] getEnumConstants()  返回该枚举类型的所有元素，如果Class对象不是枚举类型，则返回null。

boolean isEnum()        当且仅当该类声明为源代码中的枚举时返回 true

```java

// 获取class对象引用

Class<?> clasz = e.getDeclaringClass();

if(clasz.isEnum()) {

    Day[] dsz = (Day[]) clasz.getEnumConstants();

    System.out.println("dsz:"+Arrays.toString(dsz));

}

/\*\*

   输出结果:

   dsz:[MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]

 \*/

```

枚举进阶

实际上枚举类除了不能使用继承外，可以把enum类当成常规类。可以添加变量和方法。

向enum类添加方法与自定义构造函数

```java

public enum Day2 {

    MONDAY("星期一"),

    TUESDAY("星期二"),

    WEDNESDAY("星期三"),

    THURSDAY("星期四"),

    FRIDAY("星期五"),

    SATURDAY("星期六"),

    SUNDAY("星期日");//记住要用分号结束

    private String desc;//中文描述

    /\*\*

     \* 私有构造,防止被外部调用

     \* @param desc

     \*/

    private Day2(String desc){

        this.desc=desc;

    }

    /\*\*

     \* 定义方法,返回描述,跟常规类的定义没区别

     \* @return

     \*/

    public String getDesc(){

        return desc;

    }

    public static void main(String[] args){

        for (Day2 day:Day2.values()) {

            System.out.println("name:"+day.name()+

                    ",desc:"+day.getDesc());

        }

    }

    /\*\*

     输出结果:

     name:MONDAY,desc:星期一

     name:TUESDAY,desc:星期二

     name:WEDNESDAY,desc:星期三

     name:THURSDAY,desc:星期四

     name:FRIDAY,desc:星期五

     name:SATURDAY,desc:星期六

     name:SUNDAY,desc:星期日

     \*/

}

```

注意：

不能在枚举常量定义之前定义任何方法，否则编译报错。

即使定义了构造方法，仍然不能调用构造方法，枚举类的构造方法只能由编译器调用。

覆盖Enum类方法

Enum中的定义的方法只有toString方法没有使用final修饰，因此只能覆盖toString方法

```java

public enum Day2 {

    MONDAY("星期一"),

    TUESDAY("星期二"),

    WEDNESDAY("星期三"),

    THURSDAY("星期四"),

    FRIDAY("星期五"),

    SATURDAY("星期六"),

    SUNDAY("星期日");//记住要用分号结束

    private String desc;//中文描述

    /\*\*

     \* 私有构造,防止被外部调用

     \* @param desc

     \*/

    private Day2(String desc){

        this.desc=desc;

    }

    /\*\*

     \* 覆盖

     \* @return

     \*/

    @Override

    public String toString() {

        return desc;

    }

    public static void main(String[] args){

        for (Day2 day:Day2.values()) {

            System.out.println("name:"+day.name()+

                    ",desc:"+day.toString());

        }

    }

    /\*\*

     输出结果:

     name:MONDAY,desc:星期一

     name:TUESDAY,desc:星期二

     name:WEDNESDAY,desc:星期三

     name:THURSDAY,desc:星期四

     name:FRIDAY,desc:星期五

     name:SATURDAY,desc:星期六

     name:SUNDAY,desc:星期日

     \*/

}

```

枚举类中定义抽象方法

可以在枚举类中定义抽象方法，然后每个枚举常量可以提供不同的实现。abstract关键字对于枚举类来说并不是必须。

```java

public enum EnumDemo3 {

    FIRST {

        @Override

        public String getInfo() {

            return "FIRST TIME";

        }

    },

    SECOND {

        @Override

        public String getInfo() {

            return "SECOND TIME";

        }

    };

    /\*\*

     \* 定义抽象方法

     \* @return

     \*/

    public abstract String getInfo();

    //测试

    public static void main(String[] args) {

        System.out.println("F:"+EnumDemo3.FIRST.getInfo());

        System.out.println("S:"+EnumDemo3.SECOND.getInfo());

        /\*\*

         输出结果:

         F:FIRST TIME

         S:SECOND TIME

         \*/

    }

}

```

注意：

枚举类型的实例终究不能作为类型传递使用。

```java

// 无法通过编译，EnumDemo3.FIRST是个实例对象

public void text(EnumDemo3.FIRST instance){}

```

枚举类与接口

可以实现多个接口。

```java

interface food{

    void eat();

}

interface sport{

    void run();

}

public enum EnumDemo2 implements food ,sport{

    FOOD,

    SPORT;

    @Override

    public void eat() {

        System.out.println("eat.....");

    }

    @Override

    public void run() {

        System.out.println("run.....");

    }

}

```

```java

public enum Meal{

    APPETIZER(Food.Appetizer.class),

    MAINCOURSE(Food.MainCourse.class),

    DESSERT(Food.Dessert.class),

    COFFEE(Food.Coffee.class);

    private Food[] values;

    private Meal(Class<? extends Food> kind) {

      // 通过class对象获取枚举实例

      values = kind.getEnumConstants();

    }

    public interface Food {

      enum Appetizer implements Food {

        SALAD, SOUP, SPRING\_ROLLS;

      }

      enum MainCourse implements Food {

        LASAGNE, BURRITO, PAD\_THAI,

        LENTILS, HUMMOUS, VINDALOO;

      }

      enum Dessert implements Food {

        TIRAMISU, GELATO, BLACK\_FOREST\_CAKE,

        FRUIT, CREME\_CARAMEL;

      }

      enum Coffee implements Food {

        BLACK\_COFFEE, DECAF\_COFFEE, ESPRESSO,

        LATTE, CAPPUCCINO, TEA, HERB\_TEA;

      }

    }

}

```

枚举与switch

switch支持int，char，1.5之后支持enum，1.7之后支持String。

```java

enum Color {GREEN,RED,BLUE}

public class EnumDemo4 {

    public static void printName(Color color){

        switch (color){

            case BLUE: // 无需使用Color进行引用

                System.out.println("蓝色");

                break;

            case RED:

                System.out.println("红色");

                break;

            case GREEN:

                System.out.println("绿色");

                break;

        }

    }

    public static void main(String[] args){

        printName(Color.BLUE);

        printName(Color.RED);

        printName(Color.GREEN);

        //蓝色

        //红色

        //绿色

    }

}

// switch条件进行结合使用时，无需使用Color引用。

```

枚举与单例模式

在实际应用中，线程池、缓存、日志对象、对话框对象常被设计成单例，总之，选择单例模式就是为了避免对象不一致状态。

```java

/\*\*

 \* 饿汉式（基于classloder机制避免了多线程的同步问题）

 \* 无法做到延迟创建对象，事实上如果该单例类涉及资源较多，创建比较耗时间时，我们更希望它可以尽可能地延迟加载，从而减小初始化的负载

 \*/

public class SingletonHungry {

    private static SingletonHungry instance = new SingletonHungry();

    private SingletonHungry() {

    }

    public static SingletonHungry getInstance() {

        return instance;

    }

}

/\*\*

 \* 懒汉式单例模式（适合多线程安全）

 \* 遗憾的是，由于synchronized的存在，效率很低

 \*/

public class SingletonLazy {

    private static volatile SingletonLazy instance;

    private SingletonLazy() {

    }

    public static synchronized SingletonLazy getInstance() {

        if (instance == null) {

            instance = new SingletonLazy();

        }

        return instance;

    }

}

// 改进

// 双重检查锁

public class Singleton {

    private static volatile Singleton singleton = null;

    private Singleton() {}

    public static Singleton getSingleton() {

        if(singleton == null) {

            synchronized (Singleton.class) {

                if(singleton == null) {

                    singleton = new Singleton();

                }

            }

        }

        return singleton;

    }

}

volatile关键字，该关键字有两层语义。

第一层语义是可见性，可见性是指在一个线程中对该变量的修改会马上由工作内存（Work Memory）写回主内存（Main Memory），所以其它线程会马上读取到已修改的值，关于工作内存和主内存可简单理解为高速缓存（直接与CPU打交道）和主存（日常所说的内存条），注意工作内存是线程独享的，主存是线程共享的。

volatile的第二层语义是禁止指令重排序优化。

volatile的禁止指令重排序优化功能在Java 1.5后才得以实现，因此1.5前的版本仍然是不安全的，即使使用了volatile关键字。

/\*\*

 \* 静态内部类

 \*/

public class SingletonInner {

    private static class Holder {

        private static SingletonInner singleton = new SingletonInner();

    }

    private SingletonInner(){}

    public static SingletonInner getSingleton(){

        return Holder.singleton;

    }

}

由于静态内部类只会被加载一次，所以这种写法也是线程安全的

上述几种写法在序列化的时候单例都会被破坏：

1. 序列化可能会破坏单例模式，比较每次反序列化一个序列化的对象实例时都会创建一个新的实例

//测试例子(四种写解决方式雷同)

public class Singleton implements java.io.Serializable {

   public static Singleton INSTANCE = new Singleton();

   protected Singleton() {

   }

   //反序列时直接返回当前INSTANCE

   private Object readResolve() {

            return INSTANCE;

      }

}

2. 使用反射强行调用私有构造器，解决方式可以修改构造器，让它在创建第二个实例的时候抛异常

public static Singleton INSTANCE = new Singleton();

private static volatile  boolean  flag = true;

private Singleton(){

    if(flag){

    flag = false;

    }else{

        throw new RuntimeException("The instance  already exists ！");

    }

}

一劳永逸的办法实现单例：

/\*\*

 \* 枚举单例

 \*/

public enum  SingletonEnum {

    INSTANCE;

    private String name;

    public String getName(){

        return name;

    }

    public void setName(String name){

        this.name = name;

    }

}

枚举序列化是由jvm保证的，每一个枚举类型和定义的枚举变量在JVM中都是唯一的，在枚举类型的序列化和反序列化上，Java做了特殊的规定：在序列化时Java仅仅是将枚举对象的name属性输出到结果中，反序列化的时候则是通过java.lang.Enum的valueOf方法来根据名字查找枚举对象。

同时，编译器是不允许任何对这种序列化机制的定制，并禁用了writeObject、readObject、readObjectNoData、writeReplace和readResolve等方法，从而保证了枚举实例的唯一性。

下面试图通过反射获取构造器并创建枚举：

public static void main(String[] args) throws IllegalAccessException, InvocationTargetException, InstantiationException, NoSuchMethodException {

    //获取枚举类的构造函数(前面的源码已分析过)

     Constructor<SingletonEnum> constructor=SingletonEnum.class.getDeclaredConstructor(String.class,int.class);

     constructor.setAccessible(true);

     //创建枚举

     SingletonEnum singleton=constructor.newInstance("otherInstance",9);

}

public T newInstance(Object ... initargs)

throws InstantiationException, IllegalAccessException,

       IllegalArgumentException, InvocationTargetException

{

    if (!override) {

        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {

            Class<?> caller = Reflection.getCallerClass();

            checkAccess(caller, clazz, null, modifiers);

        }

    }

    //这里判断Modifier.ENUM是不是枚举修饰符，如果是就抛异常

    if ((clazz.getModifiers() & Modifier.ENUM) != 0)

        throw new IllegalArgumentException("Cannot reflectively create enum objects");

    ConstructorAccessor ca = constructorAccessor;   // read volatile

    if (ca == null) {

        ca = acquireConstructorAccessor();

    }

    @SuppressWarnings("unchecked")

    T inst = (T) ca.newInstance(initargs);

    return inst;

}

```

EnumMap

EnumMap要求其Key必须为Enum类型。

EnumMap的key值不能为null。

操作与一般的Map差不多。

```java

构造函数：

//创建一个具有指定键类型的空枚举映射。

EnumMap(Class<K> keyType)

//创建一个其键类型与指定枚举映射相同的枚举映射，最初包含相同的映射关系（如果有的话）。

EnumMap(EnumMap<K,? extends V> m)

//创建一个枚举映射，从指定映射对其初始化。

EnumMap(Map<K,? extends V> m)

//使用第一种构造

Map<Color,Integer> enumMap=new EnumMap<>(Color.class);

//使用第二种构造

Map<Color,Integer> enumMap2=new EnumMap<>(enumMap);

//使用第三种构造

Map<Color,Integer> hashMap = new HashMap<>();

hashMap.put(Color.GREEN, 2);

hashMap.put(Color.BLUE, 3);

Map<Color, Integer> enumMap = new EnumMap<>(hashMap);

```

注意与HashMap的主要不同在于构造方法需要传递类型参数和EnumMap保证Key顺序与枚举中的顺序一致，记住Key不能为null。

EnumSet

EnumSet是与枚举类型一起使用的专用 Set 集合，EnumSet中所有元素都必须是枚举类型。

EnumSet不允许使用 null 元素。试图插入 null 元素将抛出 NullPointerException。

EnumSet不是线程安全的。

创建EnumSet并不能使用new关键字，因为它是个抽象类，而应该使用其提供的静态工厂方法，EnumSet的静态工厂方法比较多，如下：

```java

创建一个具有指定元素类型的空EnumSet。

EnumSet<E>  noneOf(Class<E> elementType)

//创建一个指定元素类型并包含所有枚举值的EnumSet

<E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)

// 创建一个包括枚举值中指定范围元素的EnumSet

<E extends Enum<E>> EnumSet<E> range(E from, E to)

// 初始集合包括指定集合的补集

<E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)

// 创建一个包括参数中所有元素的EnumSet

<E extends Enum<E>> EnumSet<E> of(E e)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)

<E extends Enum<E>> EnumSet<E> of(E first, E... rest)

//创建一个包含参数容器中的所有元素的EnumSet

<E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)

<E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)

import java.util.ArrayList;

import java.util.EnumSet;

import java.util.List;

/\*\*

 \* Created by wuzejian on 2017/5/12.

 \*

 \*/

enum Color {

    GREEN , RED , BLUE , BLACK , YELLOW

}

public class EnumSetDemo {

    public static void main(String[] args){

        //空集合

        EnumSet<Color> enumSet= EnumSet.noneOf(Color.class);

        System.out.println("添加前："+enumSet.toString());

        enumSet.add(Color.GREEN);

        enumSet.add(Color.RED);

        enumSet.add(Color.BLACK);

        enumSet.add(Color.BLUE);

        enumSet.add(Color.YELLOW);

        System.out.println("添加后："+enumSet.toString());

        System.out.println("-----------------------------------");

        //使用allOf创建包含所有枚举类型的enumSet，其内部根据Class对象初始化了所有枚举实例

        EnumSet<Color> enumSet1= EnumSet.allOf(Color.class);

        System.out.println("allOf直接填充："+enumSet1.toString());

        System.out.println("-----------------------------------");

        //初始集合包括枚举值中指定范围的元素

        EnumSet<Color> enumSet2= EnumSet.range(Color.BLACK,Color.YELLOW);

        System.out.println("指定初始化范围："+enumSet2.toString());

        System.out.println("-----------------------------------");

        //指定补集，也就是从全部枚举类型中去除参数集合中的元素，如下去掉上述enumSet2的元素

        EnumSet<Color> enumSet3= EnumSet.complementOf(enumSet2);

        System.out.println("指定补集："+enumSet3.toString());

        System.out.println("-----------------------------------");

        //初始化时直接指定元素

        EnumSet<Color> enumSet4= EnumSet.of(Color.BLACK);

        System.out.println("指定Color.BLACK元素："+enumSet4.toString());

        EnumSet<Color> enumSet5= EnumSet.of(Color.BLACK,Color.GREEN);

        System.out.println("指定Color.BLACK和Color.GREEN元素："+enumSet5.toString());

        System.out.println("-----------------------------------");

        //复制enumSet5容器的数据作为初始化数据

        EnumSet<Color> enumSet6= EnumSet.copyOf(enumSet5);

        System.out.println("enumSet6："+enumSet6.toString());

        System.out.println("-----------------------------------");

        List<Color> list = new ArrayList<Color>();

        list.add(Color.BLACK);

        list.add(Color.BLACK);//重复元素

        list.add(Color.RED);

        list.add(Color.BLUE);

        System.out.println("list:"+list.toString());

        //使用copyOf(Collection<E> c)

        EnumSet enumSet7=EnumSet.copyOf(list);

        System.out.println("enumSet7:"+enumSet7.toString());

        /\*\*

         输出结果：

         添加前：[]

         添加后：[GREEN, RED, BLUE, BLACK, YELLOW]

         -----------------------------------

         allOf直接填充：[GREEN, RED, BLUE, BLACK, YELLOW]

         -----------------------------------

         指定初始化范围：[BLACK, YELLOW]

         -----------------------------------

         指定补集：[GREEN, RED, BLUE]

         -----------------------------------

         指定Color.BLACK元素：[BLACK]

         指定Color.BLACK和Color.GREEN元素：[GREEN, BLACK]

         -----------------------------------

         enumSet6：[GREEN, BLACK]

         -----------------------------------

         list:[BLACK, BLACK, RED, BLUE]

         enumSet7:[RED, BLUE, BLACK]

         \*/

    }

}

```

EnumSet的实现是通过位向量运算。非常高效。

参考：

https://blog.csdn.net/javazejian/article/details/71333103

# 异常

编译期异常

一般的异常处理是在运行期间出现的异常进行处理。

JVM运行时接收到异常默认处理是打印堆栈。

简单示例：

```java

try {

    int a = 1 / 0;

} catch (ArithmeticException ex) {

    System.out.println("error");

    ex.printStackTrace();

}

class MyException extends RuntimeException {

    public MyException() {

        super();

    }

    public MyException(String message) {

        super(message);

    }

}

class SomeException extends Exception {

    SomeException(String msg) {

        super(msg);

    }

}

public class B1 {

    public static void main(String[] args) {

        try {

            f1();

        } catch (SomeException ex) {

            System.out.println(ex.getMessage()); // error

        }

    }

    static void f1() throws SomeException {

        throw new SomeException("error");

    }

}

```

获取到异常一定要进行处理，可以继续抛出向下传递或者进行异常的处理。

异常的分类

1. Error：程序中无法处理的错误，表示运行应用程序中出现了严重的错误。此类错误一般表示代码运行时JVM出现问题。常见的

OutOfMemoryError extends VirtualMachineError

StackOverflowError extends VirtualMachineError

    VirtualMachineError extends Error

NoClassDefFoundError extends LinkageError

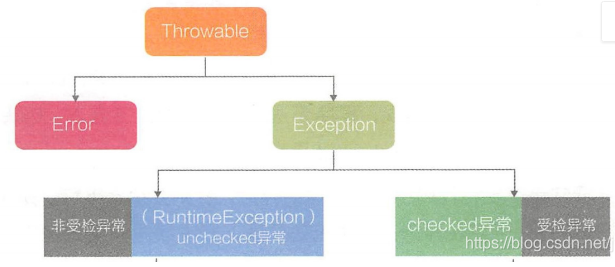
    LinkageError extends Error

当此类错误发生时，应用不应该去处理此类错误。

2. Exception：所有异常的父类。程序本身可以捕获并且可以处理的异常。

3. RuntimeException：运行时异常(非受检异常)，RuntimeException类和它的子类表示JVM在运行期间可能出现的错误。编译器不会检查此类异常，并且不要求处理异常，比如用空值对象的引用（NullPointerException）、数组下标越界（ArrayIndexOutBoundException）。此类异常属于不可查异常，一般是由程序逻辑错误引起的，在程序中可以选择捕获处理，也可以不处理。

4. 非运行时异常(受检异常)：Exception中除RuntimeException和它的子类之外的异常。编译器会检查此类异常，如果程序中出现此类异常，比如说IOException，必须对该异常进行处理，要么使用try-catch捕获，要么使用throws语句抛出，否则编译不通过。



异常的处理

1. throw

用在方法内，用来抛出一个异常对象，将这个异常对象传递到调用者处，并结束当前方法的执行。

```java

void f1() throws 异常类名 {

    throw new 异常类名(参数);

}

int div(int a, int b) {

    if(b==0)

        throw new ArithmeticException("异常信息：除数不能为0");

    return a/b;

}

```

2. throws

运用于方法声明之上，用于表示当前方法不处理异常，而是提醒该方法的调用者来处理异常。

```java

修饰符 返回值类型 方法名（参数） throws 异常类名1，异常类名2 ... { }

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.InputStream;

public class DemoThrows {

    public static void main(String[] args) throws FileNotFoundException{

        readFile();

    }

    public static void readFile() throws FileNotFoundException {

        InputStream is = new FileInputStream("E:/iodemo/ch01.txt");

    }

    void f2() {

        try {

            readFile();

        } catch (FileNotFoundException e) {

            System.out.println("系统找不到指定的路径");

        }

        System.out.println("后续代码");

    }

}

```

3. try...catch...finally

注意：

在一个try catch语句块中，父类异常的捕获语句不可以写在子类异常捕获语句的上面。

捕获到异常一定要进行处理，而不是什么都不做，这会隐藏异常。

如果想把程序写得特别健壮，使用try catch去捕获异常并处理掉捕获后的异常是必不可少的做法。见 java-effect.docx

Exception类提供的几种方法

```java

package java.lang;

public class Exception extends Throwable {

    void printStackTrace();

    String getMessage();

}

```

注意：

当子类重写父类的带有throws声明的方法时，throws声明的异常必须在父类异常的可控范围内，用于处理父类的throws方法的异常处理器，必须也适用于子类的这个带throws方法。

比如：

父类方法throws的是2个异常，子类就不能throws 3个以上的异常。父类throws IOException，子类就必须throws IOException或者IOException的子类。

```java

class A {

    public void f1() throws IOException {

        throw new IOException("111");

    }

    public static void main(String[] args) {

        A[] arr = new A[2];

        arr[0] = new A();

        arr[1] = new B();

        for (A a : arr) {

            try {

                a.f1();

            } catch (IOException ex) {

                System.out.println(ex.getMessage());

            }

        }

    }

    static class B extends A {

//            @Override

//    public void f1() throws FileNotFoundException {

//                throw new FileNotFoundException("bb");

//            }; // allow

        @Override

        public void f1() throws NullPointerException, IOError {

            throw new NullPointerException("22");

        }; // not allow

//        @Override

//        public void f1() {

//            System.out.println("bb");

//        }; // allow

    }

}

```

自定义异常

1. 继承java.lang.Exception

2. 方法中throw new MyException

3. 方法声明throws MyException

```java

class MyException extends Exception {

    private int code; // 状态码

    private String msg; // 异常信息

    MyException(String msg, int code) {

        super(msg);

        this.code = code;

    }

    public int code() {

        return code;

    }

}

public class B1 {

    public static void main(String[] args) {

        try {

            f1();

        } catch (MyException ex) {

            System.out.println(ex.getMessage()); // error

        }

    }

    static void f1() throws MyException {

        throw new MyException("error", 100);

    }

}

```

注意：

按照国际惯例，自定义的异常应该总是包含如下的构造函数：

一个无参构造函数

一个带有String参数的构造函数，并传递给父类的构造函数

一个带有String参数和Throwable参数，并都传递给父类构造函数

一个带有Throwable 参数的构造函数，并传递给父类的构造函数

可以参考IOException实现：

```java

public class IOException extends Exception {

    static final long serialVersionUID = 7818375828146090155L;

    public IOException() {

        super();

    }

    public IOException(String message) {

        super(message);

    }

    public IOException(String message, Throwable cause) {

        super(message, cause);

    }

    public IOException(Throwable cause) {

        super(cause);

    }

}

```

异常的链化

假设B模块完成自己的逻辑需要调用A模块的方法，如果A模块发生异常，则B也将不能完成而发生异常，但是B在抛出异常时，会将A的异常信息掩盖掉，这将使得异常的根源信息丢失。异常的链化可以将多个模块的异常串联起来，使得异常信息不会丢失。

```java

package java.lang;

public class Throwable implements Serializable {

    private Throwable cause = this;

    public Throwable();

    public Throwable(String message);

    public Throwable(Throwable cause);

    public Throwable(String message, Throwable cause);

}

public static void main(String[] args) {

    int result;

    try {

        result = add();

        System.out.println("结果:"+result);

    } catch (Exception e){

        e.printStackTrace();

    }

}

private static List<Integer> getInputNumbers() {

    List<Integer> nums = new ArrayList<>();

    Scanner scan = new Scanner(System.in);

    try {

        int num1 = scan.nextInt();

        int num2 = scan.nextInt();

        nums.add(new Integer(num1));

        nums.add(new Integer(num2));

    }catch(InputMismatchException immExp){

        throw immExp;

    }finally {

        scan.close();

    }

    return nums;

}

private static int add() throws Exception {

    int result;

    try {

        List<Integer> nums = getInputNumbers();

        result = nums.get(0) + nums.get(1);

    }catch(InputMismatchException immExp){

        throw new Exception("计算失败", immExp);

    }

    return result;

}

```

注意：

Java中的异常是线程独立的，线程的问题应该由线程自己来解决，而不要委托到外部，也不会直接影响到其它线程的执行。

在try块中即便有return，break，continue等改变执行流的语句，finally也会执行。

finally中的return 会覆盖 try 或者catch中的返回值。

```java

public static void main(String[] args) {

    int result;

    result  =  foo();

    System.out.println(result); // 2

    result = bar();

    System.out.println(result); // 2

}

@SuppressWarnings("finally")

public static int foo() {

    trz{

        int a = 5 / 0;

    } catch (Exception e){

        return 1;

    } finally{

        return 2;

    }

}

@SuppressWarnings("finally")

public static int bar() {

    try {

        return 1;

    }finally {

        return 2;

    }

}

```

finally中的return会抑制（消灭）前面try或者catch块中的异常

```java

class TestException {

    public static void main(String[] args) {

        int result;

        try {

            result = foo();

            System.out.println(result);           // 输出100

        } catch (Exception e) {

            System.out.println(e.getMessage());    // 没有捕获到异常

        }

        try {

            result  = bar();

            System.out.println(result);           // 输出100

        } catch (Exception e) {

            System.out.println(e.getMessage());    // 没有捕获到异常

        }

    }

    // catch中的异常被抑制

    @SuppressWarnings("finally")

    public static int foo() throws Exception {

        try {

            int a = 5/0;

            return 1;

        } catch(ArithmeticException amExp) {

            throw new Exception("我将被忽略，因为下面的finally中使用了return");

        } finally {

            return 100;

        }

    }

    //try中的异常被抑制

    @SuppressWarnings("finally")

    public static int bar() throws Exception {

        try {

            int a = 5/0;

            return 1;

        } finally {

            return 100;

        }

    }

}

```

finally中的异常会覆盖（消灭）前面try或者catch中的异常

```java

class TestException {

    public static void main(String[] args) {

        int result;

        try {

            result = foo();

        } catch (Exception e) {

            System.out.println(e.getMessage());    // 输出：我是finaly中的Exception

        }

        try{

            result  = bar();

        } catch (Exception e) {

            System.out.println(e.getMessage());    // 输出：我是finaly中的Exception

        }

    }

    // catch中的异常被抑制

    @SuppressWarnings("finally")

    public static int foo() throws Exception {

        try {

            int a = 5/0;

            return 1;

        } catch(ArithmeticException amExp) {

            throw new Exception("我将被忽略，因为下面的finally中抛出了新的异常");

        } finally {

            throw new Exception("我是finaly中的Exception");

        }

    }

    // try中的异常被抑制

    @SuppressWarnings("finally")

    public static int bar() throws Exception {

        try {

            int a = 5/0;

            return 1;

        } finally {

            throw new Exception("我是finaly中的Exception");

        }

    }

}

```

建议：

不要在fianlly中使用return

不要在finally中抛出异常

减轻finally的任务，不要在finally中做一些其它的事情，finally块仅仅用来释放资源是最合适的

将尽量将所有的return写在函数的最后面，而不是try ... catch ... finally中

# 7. Class对象

RTTI概念：

RTTI（Run-Time Type Identification）运行时类型识别

作用是在运行时识别一个对象的类型和类的信息。

分两种：

一种是类的类型信息在编译期就已经确定

另一种是通过反射机制，允许在运行时发现和使用类型的信息

在Java中用来表示运行时类型信息的对应类就是Class类。

java.lang.Class

```java

package java.lang;

public final class Class<T> implements java.io.Serializable,GenericDeclaration,Type, AnnotatedElement {

    private static final int ANNOTATION= 0x00002000;

    private static final int ENUM      = 0x00004000;

    private static final int SYNTHETIC = 0x00001000;

    private static native void registerNatives();

    static {

        registerNatives();

    }

    /\*

     \* Private constructor. Only the Java Virtual Machine creates Class objects.（私有构造，只能由JVM创建该类）

     \* This constructor is not used and prevents the default constructor being

     \* generated.

     \*/

    private Class(ClassLoader loader) {

        // Initialize final field for classLoader.  The initialization value of non-null

        // prevents future JIT optimizations from assuming this final field is null.

        classLoader = loader;

    }

}

```

每个类都有一个Class对象。

编写并编译一个类就会产生这个类对应的Class对象，并且这个对象保存在类同名的.class文件中（编译后的字节码文件保存的就是Class对象）。

当new一个对象或者引用静态成员变量时，Java虚拟机(JVM)中的类加载器子系统会将对应Class对象加载到JVM中，然后JVM再根据这个类型信息相关的Class对象创建我们需要实例对象或者提供静态变量的引用值。

注意，手动编写的每个class类，无论创建多少个实例对象，在JVM中都只有一个Class对象，即在内存中每个类有且只有一个相对应的Class对象。

Class类只存私有构造函数，因此对应Class对象只能有JVM创建和加载。

Class类的对象作用是运行时提供或获得某个对象的类型信息，这点对于反射技术很重要。

Class对象的加载

Class对象是由JVM加载的。

加载的时机是，当对类第一次使用时才动态加载到JVM中。

当创建第一个对类的静态成员引用时，就会加载类的字节码文件。注意，使用new创建新实例对象也会被当作对类的静态成员的引用(构造函数也是类的静态方法)。

类是按需加载的，使用到类时，类加载器首先会检查这个类的Class对象是否已加载，如果没有，默认的类加载器就会先根据类名查找.class文件。找到之后，必须接受相关验证，以确保其没有被破坏或者包含不良Java代码（这是java的安全机制检测），没问题后字节码文件保存的Class对象就被加载到内存中，可以被JVM使用来初始化对象。

测试类加载时机

```java

public class B2 {

    public static void print(Object obj) {

        System.out.println(obj);

    }

    public static void main(String[] args) {

        print("inside main");

        new Candy();

        print("After creating Candy");

        try {

            Class.forName("com.zejian.Gum");

        } catch(ClassNotFoundException e) {

            print("Couldn't find Gum");

        }

        print("After Class.forName(\"com.zejian.Gum\")");

        new Cookie();

        print("After creating Cookie");

    }

}

class Candy {

    static {System.out.println("Loading Candy");}

}

class Gum {

    static {System.out.println("Loading Gum");}

}

class Cookie {

    static {System.out.println("Loading Cookie");}

}

```

Class对象的获取方式

获取Class对象引用的方式3种：

1. 通过继承自Object类的getClass方法

2. Class类的静态方法forName

3. 字面常量的方式.class

获取一个类的Class对象引用：

```java

java.lang.Class

public static Class<?> forName(String className)

                throws ClassNotFoundException {}

public static Class<?> forName(String name, boolean initialize,

                    ClassLoader loader)

throws ClassNotFoundException {}

java.lang.Object

public final native Class<?> getClass();

```

```java

public static void main(String[] args) {

    try{

        // 通过Class.forName获取Gum类的Class对象

        Class clazz=Class.forName("com.zejian.Gum");

        System.out.println("forName=clazz:"+clazz.getName());

    }catch (ClassNotFoundException e) { // 运行时检查

        e.printStackTrace();

    }

    // 通过实例对象获取Gum的Class对象

    Gum gum = new Gum();

    Class clazz2=gum.getClass();

    System.out.println("new=clazz2:"+clazz2.getName());

}

```

Class字面常量

Class clazz = Gum.class;

这种方式简单，高效，更安全。

通过字面量的方法获取Class对象的引用不会自动初始化该类。

类，接口，数组，基本数据类型都可以用

基本数据类型的包装类型有个Type，指向基本数据类型的Class对象。

```java

Class c = B2.class;

int a = 1;

Class<Boolean> cc = Boolean.TYPE;

System.out.println(cc); // boolean

//        char.class = Character.TYPE;

//        byte.class = Byte.TYPE;

//        short.class = Short.TYPE;

//        int.class = Integer.TYPE;

//        long.class = Long.TYPE;

//        float.class = Float.TYPE;

//        double.class = Double.TYPE;

//        void.class = Void.TYPE;

```

```java

import java.util.\*;

class Initable {

  // 编译期静态常量

  static final int staticFinal = 47;

  // 非编期静态常量

  static final int staticFinal2 =

    ClassInitialization.rand.nextInt(1000);

  static {

    System.out.println("Initializing Initable");

  }

}

class Initable2 {

  // 静态成员变量

  static int staticNonFinal = 147;

  static {

    System.out.println("Initializing Initable2");

  }

}

class Initable3 {

  // 静态成员变量

  static int staticNonFinal = 74;

  static {

    System.out.println("Initializing Initable3");

  }

}

public class ClassInitialization {

  public static Random rand = new Random(47);

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

    // 字面常量获取方式获取Class对象

    Class initable = Initable.class;

    System.out.println("After creating Initable ref");

    // 不触发类初始化

    System.out.println(Initable.staticFinal);

    // 会触发类初始化

    System.out.println(Initable.staticFinal2);

    // 会触发类初始化

    System.out.println(Initable2.staticNonFinal);

    // forName方法获取Class对象

    Class initable3 = Class.forName("Initable3");

    System.out.println("After creating Initable3 ref");

    System.out.println(Initable3.staticNonFinal);

  }

}

After creating Initable ref

47

Initializing Initable

258

Initializing Initable2

147

Initializing Initable3

After creating Initable3 ref

74

```

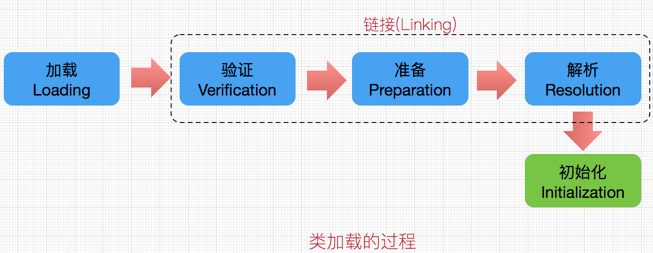
其中实例类的getClass方法和Class类的静态方法forName都将会触发类的初始化阶段，而字面常量获取Class对象的方式则不会触发初始化。

类加载的过程：

1. 加载：类加载过程的一个阶段：通过一个类的完全限定查找此类字节码文件，并利用字节码文件创建一个Class对象

2. 链接：验证字节码的安全性和完整性，准备阶段正式为静态域分配存储空间，注意此时只是分配静态成员变量的存储空间，不包含实例成员变量，如果必要的话，解析这个类创建的对其他类的所有引用。

1. 初始化：类加载最后阶段，若该类具有超类，则对其进行初始化，执行静态初始化器和静态初始化成员变量。



获取字面常量的Class引用时，触发的应该是加载阶段，因为在这个阶段Class对象已创建完成，获取其引用并不困难，而无需触发类的最后阶段初始化。

获取编译期静态常量也不会触发初始化。这是因为在编译阶段通过常量传播优化的方式将常量存储到了一个称为NotInitialization类的常量池中，在以后对常量的引用实际都转化为对NotInitialization类对自身常量池的引用，所以在编译期后，对编译期常量的引用都将在NotInitialization类的常量池获取，这也就是引用编译期静态常量不会触发Initable类初始化的重要原因。

初始化是类加载的最后一个阶段，也就是说完成这个阶段后类也就加载到内存中(Class对象在加载阶段已被创建)，此时可以对类进行各种必要的操作了（如new对象，调用静态成员等），注意在这个阶段，才真正开始执行类中定义的Java程序代码或者字节码。

关于类加载的初始化阶段，在虚拟机规范严格规定了有且只有5种场景必须对类进行初始化：

1. 使用new关键字实例化对象时、读取或者设置一个类的静态字段(不包含编译期常量)以及调用静态方法的时候，必须触发类加载的初始化过程(类加载过程最终阶段)。

2. 使用反射包(java.lang.reflect)的方法对类进行反射调用时，如果类还没有被初始化，则需先进行初始化，这点对反射很重要。

3. 当初始化一个类的时候，如果其父类还没进行初始化则需先触发其父类的初始化。

4. 当Java虚拟机启动时，用户需要指定一个要执行的主类(包含main方法的类)，虚拟机会先初始化这个主类

5. 当使用JDK 1.7 的动态语言支持时，如果一个java.lang.invoke.MethodHandle 实例最后解析结果为REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄对应类没有初始化时，必须触发其初始化

Class泛型

泛型提供了编译期的类型安全保证

```java

// 没有泛型

Class intClass = int.class;

// 带泛型的Class对象

Class<Integer> integerClass = int.class;

integerClass = Integer.class;

intClass= double.class; // 没有泛型的约束,可以随意赋值

// integerClass = double.class // 编译期错误,无法编译通过

// Class<Number> numberClass=Integer.class; // 编译期错误,无法编译通过

Class<?> intClass1 = int.class;

intClass1 = double.class;

Class<? extends Number> clazz = Integer.class;

clazz = double.class;

clazz = Number.class;

```

类型转换

所有类型转换都是在运行时进行正确性检查的。

```java

interface Animal {}

class Dog implements Animal {}

Animal animal= new Dog();

Dog dog = (Dog) animal;

// 这两句等同于Dog dog = (Dog) animal;

Class<Dog> dogType = Dog.class;

Dog dog = dogType.cast(animal)

java.lang.Class

public T cast(Object obj) {

    if (obj != null && !isInstance(obj))

        throw new ClassCastException(cannotCastMsg(obj));

    return (T) obj;

}

```

instanceof关键字与isInstance()

```java

public void cast2(Object obj) {

    // instanceof关键字

    if(obj instanceof Animal){

        Animal animal= (Animal) obj;

    }

    // isInstance方法

    if(Animal.class.isInstance(obj)){

        Animal animal= (Animal) obj;

    }

}

java.lang.Class

public native boolean isInstance(Object obj);

```

两者是等价的

验证isInstance方法与instanceof等价性：

```java

class A {}

class B extends A {}

public class C {

  static void test(Object x) {

    print("Testing x of type " + x.getClass());

    print("x instanceof A " + (x instanceof A));

    print("x instanceof B "+ (x instanceof B));

    print("A.isInstance(x) "+ A.class.isInstance(x));

    print("B.isInstance(x) " +

      B.class.isInstance(x));

    print("x.getClass() == A.class " +

      (x.getClass() == A.class));

    print("x.getClass() == B.class " +

      (x.getClass() == B.class));

    print("x.getClass().equals(A.class)) "+

      (x.getClass().equals(A.class)));

    print("x.getClass().equals(B.class)) " +

      (x.getClass().equals(B.class)));

  }

  public static void main(String[] args) {

    test(new A());

    test(new B());

  }

}

Testing x of type class com.zejian.A

x instanceof A true

x instanceof B false //父类不一定是子类的某个类型

A.isInstance(x) true

B.isInstance(x) false

x.getClass() == A.class true

x.getClass() == B.class false

x.getClass().equals(A.class)) true

x.getClass().equals(B.class)) false

---------------------------------------------

Testing x of type class com.zejian.B

x instanceof A true

x instanceof B true

A.isInstance(x) true

B.isInstance(x) true

x.getClass() == A.class false

x.getClass() == B.class true

x.getClass().equals(A.class)) false

x.getClass().equals(B.class)) true

```

# 8. 反射

在运行期间动态获取类的信息和调用类的方法

java.lang.Class类与java.lang.reflect类库一起对反射技术进行了全力的支持。

常用的类主要有：

Constructor类表示的是Class对象所表示的类的构造方法，利用它可以在运行时动态创建对象

Field表示Class对象所表示的类的成员变量，通过它可以在运行时动态修改成员变量的属性值(包含private)

Method表示Class对象所表示的类的成员方法，通过它可以动态调用对象的方法(包含private)

Constructor类及其用法

java.lang.Class常用方法：

```java

static Class<?> forName(String className) 返回与带有给定字符串名的类或接口相关联的Class对象

Constructor<T> getConstructor(Class<?>... parameterTypes) 返回指定参数类型、具有public访问权限的构造函数对象

Constructor<?>[] getConstructors() 返回所有具有public访问权限的构造函数的Constructor对象数组

Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes) 返回指定参数类型、所有声明的（包括private）构造函数对象

Constructor<?>[] getDeclaredConstructor() 返回所有声明的（包括private）构造函数对象

T newInstance() 创建此Class对象所表示的类的一个新实例。

```

java.lang.reflect.Constructor常用方法：

```java

Class<T> getDeclaringClass() 返回 Class 对象，该对象表示声明由此 Constructor 对象表示的构造方法的类,其实就是返回真实类型（不包含参数）

Type[] getGenericParameterTypes() 按照声明顺序返回一组 Type 对象，返回的就是 Constructor对象构造函数的形参类型。

String getName() 以字符串形式返回此构造方法的名称。

Class<?>[] getParameterTypes() 按照声明顺序返回一组 Class 对象，即返回Constructor 对象所表示构造方法的形参类型

T newInstance(Object... initargs) 使用此 Constructor对象表示的构造函数来创建新实例

String toGenericString() 返回描述此 Constructor 的字符串，其中包括类型参数。

```

```java

import java.io.Serializable;

import java.lang.reflect.Constructor;

public class ReflectDemo implements Serializable {

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

        Class<?> clazz = null;

        // 获取Class对象的引用

        clazz = Class.forName("reflect.User");

        // 第一种方法，实例化默认构造方法，User必须无参构造函数,否则将抛异常

        User user = (User) clazz.newInstance();

        user.setAge(20);

        user.setName("Rollen");

        System.out.println(user);

        // 获取带String参数的public构造函数

        Constructor cs1 = clazz.getConstructor(String.class);

        // 创建User

        User user1= (User) cs1.newInstance("xiaolong");

        user1.setAge(22);

        System.out.println("user1:"+user1.toString());

        // 取得指定带int和String参数构造函数,该方法是私有构造private

        Constructor cs2 = clazz.getDeclaredConstructor(int.class,String.class);

        // 由于是private必须设置可访问

        cs2.setAccessible(true);

        // 创建user对象

        User user2= (User) cs2.newInstance(25,"lidakang");

        System.out.println("user2:"+user2.toString());

        // 获取所有构造包含private

        Constructor<?> cons[] = clazz.getDeclaredConstructors();

        // 查看每个构造方法需要的参数

        for (int i = 0; i < cons.length; i++) {

            // 获取构造函数参数类型

            Class<?> clazzs[] = cons[i].getParameterTypes();

            System.out.println("构造函数["+i+"]:"+cons[i].toString() );

            System.out.print("参数类型["+i+"]:(");

            for (int j = 0; j < clazzs.length; j++) {

                if (j == clazzs.length - 1)

                    System.out.print(clazzs[j].getName());

                else

                    System.out.print(clazzs[j].getName() + ",");

            }

            System.out.println(")");

        }

    }

}

class User {

    private int age;

    private String name;

    public User() {

        super();

    }

    public User(String name) {

        super();

        this.name = name;

    }

    private User(int age, String name) {

        super();

        this.age = age;

        this.name = name;

    }

}

User [age=20, name=Rollen]

--------------------------------------------

user1:User [age=22, name=xiaolong]

--------------------------------------------

user2:User [age=25, name=lidakang]

--------------------------------------------

构造函数[0]:private reflect.User(int,java.lang.String)

参数类型[0]:(int,java.lang.String)

构造函数[1]:public reflect.User(java.lang.String)

参数类型[1]:(java.lang.String)

构造函数[2]:public reflect.User()

参数类型[2]:()

```

```java

Constructor cs3=clazz.getDeclaredConstructor(int.class,String.class);

System.out.println("-----getDeclaringClass-----");

Class uclazz=cs3.getDeclaringClass();

// Constructor对象表示的构造方法的类

System.out.println("构造方法的类:"+uclazz.getName());

System.out.println("-----getGenericParameterTypes-----");

// 对象表示此 Constructor 对象所表示的方法的形参类型

Type[] tps=cs3.getGenericParameterTypes();

for (Type tp:tps) {

    System.out.println("参数名称tp:"+tp);

}

System.out.println("-----getParameterTypes-----");

// 获取构造函数参数类型

Class<?> clazzs[] = cs3.getParameterTypes();

for (Class claz:clazzs) {

    System.out.println("参数名称:"+claz.getName());

}

System.out.println("-----getName-----");

// 以字符串形式返回此构造方法的名称

System.out.println("getName:"+cs3.getName());

System.out.println("-----getoGenericString-----");

// 返回描述此 Constructor 的字符串，其中包括类型参数。

System.out.println("getoGenericString():"+cs3.toGenericString());

/\*\*

 输出结果:

 -----getDeclaringClass-----

 构造方法的类:reflect.User

 -----getGenericParameterTypes-----

 参数名称tp:int

 参数名称tp:class java.lang.String

 -----getParameterTypes-----

 参数名称:int

 参数名称:java.lang.String

 -----getName-----

 getName:reflect.User

 -----getoGenericString-----

 getoGenericString():private reflect.User(int,java.lang.String)

 \*/

```

Type是所有类型的公共接口。包括原始类型、参数化类型、数组类型、类型变量和基本类型。

getGenericParameterTypes()与getParameterTypes()都是获取构成函数的参数类型，前者返回的是Type类型，后者返回的是Class类型。

Class实现了Type接口，Type表示全部类型而每个Class对象表示一个具体类型的实例。一般情况下两者表示的类型基本一样，只不过Type的表示范围比Class大。

Type还有其他子类：

1. TypeVariable：表示类型参数，可以有上界，比如：T extends Number

2. ParameterizedType：表示参数化的类型，有原始类型和具体的类型参数，比如：List<String>

3. WildcardType：表示通配符类型，比如：?, ? extends Number, ? super Integer

利用好Class类和Constructor类，可以在运行时动态创建任意对象。

Field类及其用法

获取类或者接口的成员字段。

java.lang.Class的主要方法：

```java

Field getDeclaredField(String name) 获取指定name名称的(包含private修饰的)字段，不包括继承的字段

Field[] getDeclaredField() 获取Class对象所表示的类或接口的所有(包含private修饰的)字段,不包括继承的字段

Field getField(String name) 获取指定name名称、具有public修饰的字段，包含继承字段

Field[] getField() 获取修饰符为public的字段，包含继承字段

```

```java

public class ReflectField {

    public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException {

        Class<?> clazz = Class.forName("reflect.Student");

        // 获取指定字段名称的Field类,注意字段修饰符必须为public而且存在该字段,

        // 否则抛NoSuchFieldException

        Field field = clazz.getField("age");

        System.out.println("field:"+field);

        // 获取所有修饰符为public的字段,包含父类字段,注意修饰符为public才会获取

        Field fields[] = clazz.getFields();

        for (Field f:fields) {

            System.out.println("f:"+f.getDeclaringClass());

        }

        System.out.println("================getDeclaredFields====================");

        // 获取当前类所字段(包含private字段),注意不包含父类的字段

        Field fields2[] = clazz.getDeclaredFields();

        for (Field f:fields2) {

            System.out.println("f2:"+f.getDeclaringClass());

        }

        // 获取指定字段名称的Field类,可以是任意修饰符的自动,注意不包含父类的字段

        Field field2 = clazz.getDeclaredField("desc");

        System.out.println("field2:"+field2);

    }

    /\*\*

      输出结果:

     field:public int reflect.Person.age

     f:public java.lang.String reflect.Student.desc

     f:public int reflect.Person.age

     f:public java.lang.String reflect.Person.name

     ================getDeclaredFields====================

     f2:public java.lang.String reflect.Student.desc

     f2:private int reflect.Student.score

     field2:public java.lang.String reflect.Student.desc

     \*/

}

class Person {

    public int age;

    public String name;

}

class Student extends Person {

    public String desc;

    private int score;

}

```

如果不获取父类的字段，使用Class类的getDeclaredField()/getDeclaredFields()，要获取父类的字段，使用Class类的getField()/getFields()，但是也只能获取到public修饰的的字段，无法获取父类的私有字段。

通过Field类本身的方法对指定类属性赋值：

```java

// 获取Class对象引用

Class<?> clazz = Class.forName("reflect.Student");

Student st= (Student) clazz.newInstance();

// 获取父类public字段并赋值

Field ageField = clazz.getField("age");

ageField.set(st,18);

Field nameField = clazz.getField("name");

nameField.set(st,"Lily");

// 只获取当前类的字段,不获取父类的字段

Field descField = clazz.getDeclaredField("desc");

descField.set(st,"I am student");

Field scoreField = clazz.getDeclaredField("score");

// 设置可访问，score是private的

scoreField.setAccessible(true);

scoreField.set(st,88);

System.out.println(st.toString());

// 输出结果：Student{age=18, name='Lily ,desc='I am student', score=88}

// 获取字段值

System.out.println(scoreField.get(st));

// 88

```

java.lang.reflect.Field的主要方法：

```java

void set(Object obj, Object value) 将指定对象变量上此 Field 对象表示的字段设置为指定的新值。

Object get(Object obj) 返回指定对象上此 Field 表示的字段的值

Class<?> getType() 返回一个 Class 对象，它标识了此Field 对象所表示字段的声明类型。

boolean isEnumConstant() 如果此字段表示枚举类型的元素则返回 true；否则返回 false

String toGenericString() 返回一个描述此 Field（包括其一般类型）的字符串

String getName() 返回此 Field 对象表示的字段的名称

Class<?> getDeclaringClass() 返回表示类或接口的 Class 对象，该类或接口声明由此 Field 对象表示的字段

void setAccessible(boolean flag) 将此对象的 accessible 标志设置为指示的布尔值,即设置其可访问性

```

在设置值的方法上，Field类还提供了专门针对基本数据类型的方法，如setInt()/getInt()、setBoolean()/getBoolean、setChar()/getChar()等方法。

注意：

被final关键字修饰的Field字段是安全的，在运行时可以接收任何修改，但最终其实际值是不会发生改变的。

Method类及其用法

提供类或者接口的方法信息

java.lang.Class的主要方法：

```java

Method getDeclaredMethod(String name, Class<?>... parameterTypes) 返回一个指定参数的Method对象，该对象反映此 Class 对象所表示的类或接口的指定已声明方法。

Method[] getDeclaredMethod() 返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。

Method getMethod(String name, Class<?>... parameterTypes) 返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法。

Method[] getMethods() 返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 member 方法。

```

```java

import java.lang.reflect.Method;

public class ReflectMethod  {

    public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException {

        Class clazz = Class.forName("reflect.Circle");

        // 根据参数获取public的Method,包含继承自父类的方法

        Method method = clazz.getMethod("draw",int.class,String.class);

        System.out.println("method:"+method);

        // 获取所有public的方法:

        Method[] methods =clazz.getMethods();

        for (Method m:methods){

            System.out.println("m::"+m);

        }

        System.out.println("=========================================");

        // 获取当前类的方法包含private,该方法无法获取继承自父类的method

        Method method1 = clazz.getDeclaredMethod("drawCircle");

        System.out.println("method1::"+method1);

        // 获取当前类的所有方法包含private,该方法无法获取继承自父类的method

        Method[] methods1=clazz.getDeclaredMethods();

        for (Method m:methods1){

            System.out.println("m1::"+m);

        }

    }

/\*\*

     输出结果:

     method:public void reflect.Shape.draw(int,java.lang.String)

     m::public int reflect.Circle.getAllCount()

     m::public void reflect.Shape.draw()

     m::public void reflect.Shape.draw(int,java.lang.String)

     m::public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

     m::public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

     m::public final void java.lang.Object.wait() throws java.lang.InterruptedException

     m::public boolean java.lang.Object.equals(java.lang.Object)

     m::public java.lang.String java.lang.Object.toString()

     m::public native int java.lang.Object.hashCode()

     m::public final native java.lang.Class java.lang.Object.getClass()

     m::public final native void java.lang.Object.notify()

     m::public final native void java.lang.Object.notifyAll()

     =========================================

     method1::private void reflect.Circle.drawCircle()

     m1::public int reflect.Circle.getAllCount()

     m1::private void reflect.Circle.drawCircle()

     \*/

}

class Shape {

    public void draw() {

        System.out.println("draw");

    }

    public void draw(int count , String name) {

        System.out.println("draw "+ name +",count="+count);

    }

}

class Circle extends Shape {

    private void drawCircle() {

        System.out.println("drawCircle");

    }

    public int getAllCount() {

        return 100;

    }

}

```

通过getMethods()获取Method对象时，会把父类的方法也获取到，如上的输出结果，把Object类的方法都打印出来了。而getDeclaredMethod()/getDeclaredMethods()只能获取当前类的方法。

通过Method对象调用指定类的方法：

```java

Class clazz = Class.forName("reflect.Circle");

// 创建对象

Circle circle = (Circle) clazz.newInstance();

// 获取指定参数的方法对象Method

Method method = clazz.getMethod("draw",int.class,String.class);

// 通过Method对象的invoke(Object obj,Object... args)方法调用

method.invoke(circle,15,"圈圈");

// 对私有无参方法的操作

Method method1 = clazz.getDeclaredMethod("drawCircle");

// 修改私有方法的访问标识

method1.setAccessible(true);

method1.invoke(circle);

// 对有返回值得方法操作

Method method2 =clazz.getDeclaredMethod("getAllCount");

Integer count = (Integer) method2.invoke(circle);

System.out.println("count:"+count);

/\*\*

    输出结果:

    draw 圈圈,count=15

    drawCircle

    count:100

\*/

```

java.lang.reflect.Method的主要方法：

```java

Object invoke(Object obj, Object... args) 对带有指定参数的指定对象调用由此 Method 对象表示的底层方法。

Class<?> getReturnType() 返回一个 Class 对象，该对象描述了此 Method 对象所表示的方法的正式返回类型,即方法的返回类型

Type getGenericReturnType() 返回表示由此 Method 对象所表示方法的正式返回类型的 Type 对象，也是方法的返回类型。

Class<?>[] getParameterTypes() 按照声明顺序返回 Class 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型。即返回方法的参数类型组成的数组

Type[] getGenericParameterTypes() 按照声明顺序返回 Type 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型的，也是返回方法的参数类型

String getName() 以 String 形式返回此 Method 对象表示的方法名称，即返回方法的名称

boolean isVarArgs() 判断方法是否带可变参数，如果将此方法声明为带有可变数量的参数，则返回 true；否则，返回 false。

String toGenericString() 返回描述此 Method 的字符串，包括类型参数。

```

getReturnType()/getGenericReturnType()都是获取Method对象表示的方法的返回类型，只不过前者返回的Class类型后者返回的Type。

getParameterTypes/getGenericParameterTypes也是同样的道理，都是获取Method对象所表示的方法的参数类型，其他方法与前面的Field和Constructor是类似的。

反射包中的Array类

提供了动态创建和访问 Java 数组的方法。

java.lang.Class的主要方法：

```java

Class<?> getComponentType() 返回表示数组元素类型的 Class，即数组的类型

boolean isArray() 判定此 Class 对象是否表示一个数组类。

```

java.lang.reflect.Array中的常用静态方法如下：

```java

static Object set(Object array, int index) 返回指定数组对象中索引组件的值。

static int getLength(Object array) 以 int 形式返回指定数组对象的长度

static object newInstance(Class<?> componentType, int... dimensions) 创建一个具有指定类型和维度的新数组。

static Object newInstance(Class<?> componentType, int length) 创建一个具有指定的组件类型和长度的新数组。

static void set(Object array, int index, Object value) 将指定数组对象中索引组件的值设置为指定的新值。

```

代码示例：

```java

import java.lang.reflect.Array;

public class ReflectArray {

    public static void main(String[] args) throws ClassNotFoundException {

        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // 获取数组类型的Class 即int.class

        Class<?> clazz = array.getClass().getComponentType();

        // 创建一个具有指定的组件类型和长度的新数组。

        // 第一个参数:数组的类型,第二个参数:数组的长度

        Object newArr = Array.newInstance(clazz, 15);

        // 获取原数组的长度

        int co = Array.getLength(array);

        // 赋值原数组到新数组

        System.arraycopy(array, 0, newArr, 0, co);

        for (int i:(int[]) newArr) {

            System.out.print(i+",");

        }

        // 创建了一个长度为10 的字符串数组，

        // 接着把索引位置为6 的元素设为"hello world!"，然后再读取索引位置为6 的元素的值

        Class clazz2 = Class.forName("java.lang.String");

        // 创建一个长度为10的字符串数组，在Java中数组也可以作为Object对象

        Object array2 = Array.newInstance(clazz2, 10);

        // 把字符串数组对象的索引位置为6的元素设置为"hello"

        Array.set(array2, 6, "hello world!");

        // 获得字符串数组对象的索引位置为5的元素的值

        String str = (String)Array.get(array2, 6);

        System.out.println();

        System.out.println(str);//hello

    }

    /\*\*

     输出结果:

     1,2,3,4,5,6,7,8,9,0,0,0,0,0,0,

     hello world!

     \*/

}

```

除了上的set/get外Array还专门为8种基本数据类型提供特有的方法，如setInt/getInt、setBoolean/getBoolean等。

除了上述动态修改数组长度或者动态创建数组或动态获取值或设置值外，可以利用泛型动态创建泛型数组如下：

```java

/\*\*

  \* 接收一个泛型数组，然后创建一个长度与接收的数组长度一样的泛型数组，

  \* 并把接收的数组的元素复制到新创建的数组中，

  \* 最后找出新数组中的最小元素，并打印出来

  \* @param a

  \* @param <T>

  \*/

  public  <T extends Comparable<T>> void min(T[] a) {

    // 通过反射创建相同类型的数组

    T[] b = (T[]) Array.newInstance(a.getClass().getComponentType(), a.length);

    for (int i = 0; i < a.length; i++) {

        b[i] = a[i];

    }

    T min = null;

    boolean flag = true;

    for (int i = 0; i < b.length; i++) {

        if (flag) {

            min = b[i];

            flag = false;

        }

        if (b[i].compareTo(min) < 0) {

            min = b[i];

        }

    }

    System.out.println(min);

}

```

无法直接创建泛型数组，有了Array的动态创建数组的方式这个问题也就迎刃而解了。

T[] a = new T[]; // 无效语句，编译不通

List<String>[] a = new ArrayList<String>[1]; // 无效语句，编译不通

其它方法：

```java

/\*\*

  \*    修饰符、父类、实现的接口、注解相关

  \*/

// 获取修饰符，返回值可通过Modifier类进行解读

public native int getModifiers();

// 获取父类，如果为Object，父类为null

public native Class<? super T> getSuperclass();

// 对于类，为自己声明实现的所有接口，对于接口，为直接扩展的接口，不包括通过父类间接继承来的

public native Class<?>[] getInterfaces();

// 自己声明的注解

public Annotation[] getDeclaredAnnotations();

// 所有的注解，包括继承得到的

public Annotation[] getAnnotations();

// 获取或检查指定类型的注解，包括继承得到的

public <A extends Annotation> A getAnnotation(Class<A> annotationClass);

public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass);

/\*\*

  \*   内部类相关

  \*/

// 获取所有的public的内部类和接口，包括从父类继承得到的

public Class<?>[] getClasses();

// 获取自己声明的所有的内部类和接口

public Class<?>[] getDeclaredClasses();

// 如果当前Class为内部类，获取声明该类的最外部的Class对象

public Class<?> getDeclaringClass();

// 如果当前Class为内部类，获取直接包含该类的类

public Class<?> getEnclosingClass();

// 如果当前Class为本地类或匿名内部类，返回包含它的方法

public Method getEnclosingMethod();

/\*\*

  \*    Class对象类型判断相关

  \*/

// 是否是数组

public native boolean isArray();

// 是否是基本类型

public native boolean isPrimitive();

// 是否是接口

public native boolean isInterface();

// 是否是枚举

public boolean isEnum();

// 是否是注解

public boolean isAnnotation();

// 是否是匿名内部类

public boolean isAnonymousClass();

// 是否是成员类

public boolean isMemberClass();

// 是否是本地类

public boolean isLocalClass();

```