# 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