进阶知识

泛型基础

jdk1.5 开始，可以使用任意大写字母来表示任意类型，java 中会自动根据实际情况进行类型推断。

通常约定的几种泛型如下：

泛型	用途	备注
?	不确定的类型	不会进行类型推断，可用 <? extends T>、<? super T> 限制范围
T	调用时的指定类型	
E	集合元素类型	
K	map 的 key 类型	
V	map 的 value 类型	

举例：

Java

```java
// 类上声明泛型（方法上存在泛型，类上可以没有泛型）
public class Application<K,V> {
    // 方法上声明泛型，返回值前面要使用泛型来指定
    public <K,V> Map<K,V> testMap(K key, V value){
        Map<K,V> map = new HashMap<>();
        map.put(key,value);
        return map;
    }
    // 此时方法返回值类型会根据入参类型推断
    public <T> T testItem(T value){
        return value;
    }
}
```

学过 java 的基本上都知道，使用 Object 也可以传递通用类型：

Java

```java
public Object testItem(Object value){ }
```

Object 类型的不足之处在于其值无法进行类型推断，必须要进行强制类型转换，并且只可传递继承了 Object 的类型。

注解与反射原理

注解

java 注解是一个 @interface 类，用于对代码标记，需要通过反射等手段获取注解并进行处理，否则注解没有任何作用。

定义注解：

Java

```java
@Target(ElementType.FIELD) // 注解可放置的位置
@Retention(RetentionPolicy.RUNTIME)
public @interface Column {
```

```
    String name() default "";       // 注解参数，默认值为空

    String type() default "string";

}
```

反射

Java 反射的用途在于运行时操作类及对象。

获取类对象：

Java

```
// 加载类并进行初始化：静态代码会执行
Class<?> clazz = Class.forName("com.demo.User");

// 加载类：静态代码不会执行
Class<?> clazz = ClassLoader.loadClass("com.demo.User");

// 通过对象实例获取类
Class<?> clazz = user.getClass();
```

类实例化：

Java

```
// 直接实例化：可以使用多种构造方法
User user = new User();

// 反射通过类对象实例化：实际调用无参构造
User user = clazz.newInstance();
```

类部分操作：

Java

```
class.getName():                    // 根据类获取类路径
class.getFields():                  // 获取类的public属性
class.getDeclaredFields():          // 获取类的全部属性
class.getDeclaredMethods():         // 获取类的全部方法
class.getDeclaredConstructors():    // 获取类的全部构造方法

class.getAnnotations();             // 获取注解：含继承的注解
class.getDeclaredAnnotations();     // 获取注解：不含继承的注解
clazz.isAnnotationPresent(Ann.class); // 判断是否存在注解

annotation.annotationType(): 获取注解类型
```

属性部分操作：

Java

```
field.isAnnotationPresent(MyHello.class): // 属性是否使用指定注解
field.getDeclaredAnnotations():           // 获取属性的全部注解
field.setAccessible(true):                // 设置私有属性访问权限
```

使用反射可以执行一个未知的方法（函数式接口也可实现类似操作）：

Java

```
1  // 传入对象和方法名，并执行
2  public static void consumer(Object methodObject, String methodName) throws Exception {
3      Method method = methodObject.getClass().getDeclaredMethod(methodName);
4      Object value =  method.invoke(methodObject);
5      System.out.println((String)value);
6  }
```

1.获取方法参数名

方式一：原生反射

Java

```
// JDK 1.8开始可以直接获取
public static List<String> getMethodParamName(Class<?> clazz,String methodName){
   List<String> listParam = new ArrayList<>();
   Method[] methods = clazz.getDeclaredMethods();
   for (Method method : methods) {
```

```
        if (method.getName().equals(methodName)){
            listParam = Arrays.stream(method.getParameters()).map(Parameter::getName).collect(Collectors.toList());
        }
    }
    return listParam;
}
```

方式二：字节码工具 javassist

XML

```
<dependency><!--字节码工具-->
    <groupId>org.javassist</groupId>
    <artifactId>javassist</artifactId>
    <version>3.23.1-GA</version>
</dependency>
```

代码：需要编译后才能获取

Java

```
// 所有版本jdk均可用
public static List<String> getClassMethodParamNames(String className, String methodName){
    List<String> list = new ArrayList<>();
    ClassPool classPool = ClassPool.getDefault();
    try {
        CtClass clazz = classPool.get(className);
        CtMethod method = clazz.getDeclaredMethod(methodName);
        MethodInfo info = method.getMethodInfo();
        //如果为接口类
        if (clazz.isInterface()) {
            //编译一次后生效
            MethodParametersAttribute paras = (MethodParametersAttribute)info.getAttribute(MethodParametersAttribute.tag);
            for (int i = 0; i <method.getParameterTypes().length; i++) {
                list.add(paras.getConstPool().getUtf8Info(ByteArray.readU16bit(paras.get(),i * 4 + 1)));
            }
        }else{
            int pos=Modifier.isStatic(method.getModifiers()) ? 0 : 1;// 非静态的成员函数的第一个参数是this
            LocalVariableAttribute attr = (LocalVariableAttribute)info.getCodeAttribute().getAttribute(LocalVariableAttribute.tag);
            for (int i = 0;i < method.getParameterTypes().length; i++){
                list.add(attr.variableName(i+pos));
            }
        }
        return list;
    } catch (NotFoundException e) {
        throw new NullPointerException("无法加载指定方法!");
    }
}
```

2.接口类实例化

使用 cglib 工具包

XML

```
<dependency><!--cglib-->
    <groupId>net.sourceforge.cglib</groupId>
    <artifactId>com.springsource.net.sf.cglib</artifactId>
    <version>2.1.3</version>
</dependency>
```

生成对象

Java

```
// 接口类clazz，可以使用此方法实例化dao层接口，并在intercept方法：设置参数、执行sql、映射结果
private static Object interfaceDaoToBean(Class<?> clazz){

    Class<?>[] classes = { clazz };
    Enhancer enhancer = new Enhancer();//获取接口实现类对象
    enhancer.setInterfaces(classes);
    enhancer.setCallback(new DaoProxy());
    return enhancer.create();

}
```

Stream 流

Stream是1.8的特性，可以将元素集合看成流在管道中流动，并在途中进行筛选， 排序，聚合等操作。

注意这里的流是util包下的，不是io流：

```java
// Stream
import java.util.stream.Stream;
// IO Stream
import java.io.InputStream;
```

流有并行流与串行流，它们的用法一样：

- 串行流：线程安全，速度慢，通常用stream( ) 构建。
- 并行流：线程不安全，速度块，通常用 parallelStream( )构建.

生成流

创建流很简单，基本上都是集合.stream( )，下面是一些stream流的构建方法：

```java
// 构建空流
Stream<Integer> stream1 = Stream.empty();

// 数组转stream
int[] ints = {1,2,3,4,5,6,7,8,9};
IntStream stream = Arrays.stream(ints);// 串行流
IntStream stream = Arrays.parallelStream(ints); // 并行流

// List转stream
List<Integer> list = new ArrayList<>();
Stream<Integer> stream = list.stream();
```

Collection接口提供了stream( )、parallelStream( )方法来转为Stream流，因此Collection及其子类都可以使用此方法转换为流。IDEA可使用Ctrl+Alt+B 查看此接口有哪些实现类。

操作流

返回流：对流进行操作后会关闭旧流，返回一个新的流：

```java
// 过滤元素,去大于5的
IntStream stream1 = stream.filter(i -> i > 5);

// 修改元素，所有元素值+1
IntStream stream2 = stream.map(i -> i + 1);

// 截取元素，取前5个
IntStream stream3 = stream.limit(5);

// 元素排序，IntStream流
IntStream stream4 = stream.sorted();

// 元素去重
IntStream stream5 = stream.distinct();

// 操作流数据，比如更改map流中的属性
Stream<Map<String, Object>> stream = stream().peek();
```

peek( )举例

```java
// 将map中的部分元素value替换
List<Map<String, Object>> listResult = listMap.stream().peek(map-> {
    map.putIfAbsent("key1", 0);
    map.putIfAbsent("key2", 0);
    map.putIfAbsent("key3", 0);
}).collect(Collectors.toList());
```

结束流

使用下面命令后会自动关闭流，关闭后再操作会出现异常：

```java
// 打印元素
stream.forEach(System.out::println);

// 统计元素数量,注意流会关闭
long num = stream.count();

// 流转数组
Object[] objs = stream.toArray();

// 流转为list，注意转换前要为list才行
List<Integer> list = stream.collect(Collectors.toList());

// 流转数值收集器，可以获得最大值、求和等，数值流
IntSummaryStatistics sums = stream.summaryStatistics();

// 拼接字符串流
String str = stream.collect(Collectors.joining(","));

// 元素值求和
int value = result.stream().mapToInt(map -> MapUtil.getInt(map, "value")).sum();

// 获取对象value最大值
int value = result.stream().mapToInt(map -> MapUtil.getInt(map, "value")).max();

// 获取value值最大的对象元素
Map<String, Object> maxMap = result.stream().max(Comparator.comparing(map -> MapUtil.getInt(map, "value"))).get();
```

类型转换

可以使用 map 进行流类型的转换，但要注意集合要使用包装类：

```java
Stream<Integer> stream1 = Arrays.stream(new Integer[]{1,2,3,4,5,6,7,8,9});
// 数值流转字符串流，注意int[] 构建的流无法转换
Stream<String> stream2 = stream1.map(String::valueOf);
// 字符串流转数值流
Stream<Integer> stream3 = stream2.map(Integer::valueOf);
```

日期操作：

```java
SimpleDateFormat format= new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
Date date = format.parse("2021-08-17 12:57:20"); // 字符串转日期
String dateStr = dateFormat.format(new Date());  // 日期转字符串
```

日历对象：Calendar

```java
// 创建日期对象：默认时间为当前日期
Calendar calendar = Calendar.getInstance();

// 修改年：修改年或月时，日不会变，因此需要注意2月31出现
calendar.set(Calendar.YEAR,2021);
// 修改日历时间
calendar.setTime(date);
// 日历日期增加6天
calendar.add(Calendar.DATE, 6);

// 获取周几：1为周日
int day = calendar.get(Calendar.DAY_OF_WEEK); // DAY_OF_YEAR年、DAY_OF_MONTH月、DAY_OF_WEEK周
// 获取日期
Date time = calendar.getTime();

```

函数式编程

函数式接口

在java中可以使用函数式接口将一个方法作为另一个方法的参数进行传递并执行。函数式接口通常与lombda表达式搭配使用。

```java
// runnable虽然是线程接口，但本身也可作为函数式接口使用，并且没有参数
Runnable runnable = () ->{ };

// 无返回值函数式接口：泛型为参数a类型。
Consumer<String> consumer = (a) -> { };

// callable接口可以作为无参带返回值的函数式接口
Callable<String> callable = () -> {
    return "hello";
};

// 有返回值函数式接口：泛型1为参数a类型，泛型2为响应类型。
Function<String,String> function = (a) -> {
    return "hello";
};

```

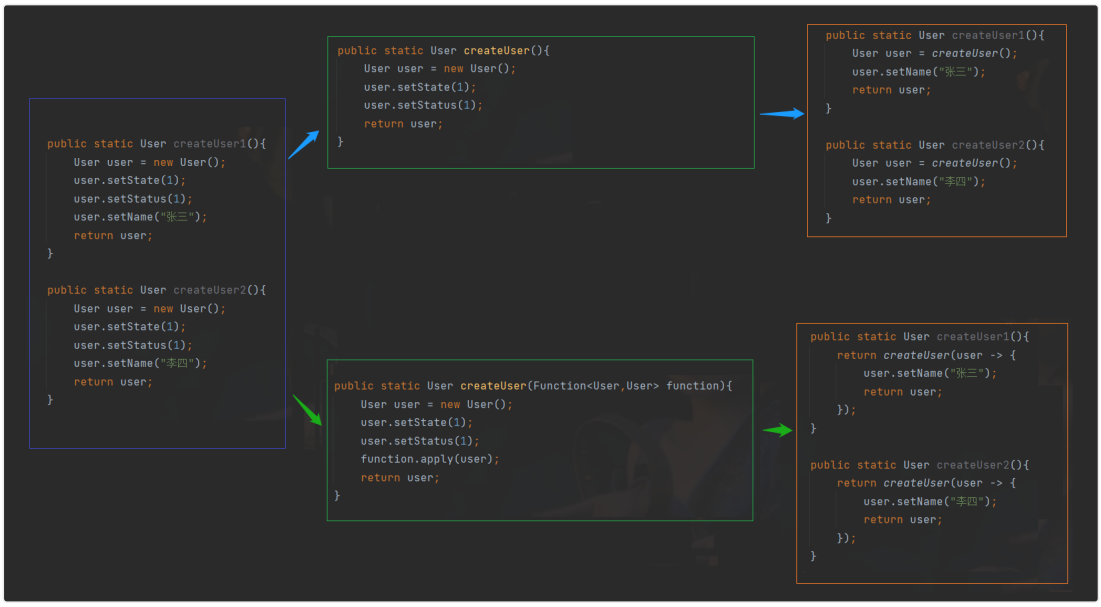
只有一个接口方法的接口类均可作为函数式接口，因此我们也可以自定义。

```java
@FunctionalInterface // 注释用于错误检查，可以省略
public interface MyFunction<T> {

    void run(T t);

}
```

使用函数式接口

```java
// 直接调用函数式接口内部的方法即可执行对应的lombda表达式
public static void run(Consumer<String> consumer,Function<String,String> function){
    consumer.accept("hello");
    String hello = function.apply("hello");
}
```

应用示例：可以通过函数式接口优化重复代码。



通过上图可以看出，函数式接口也可以进行代码复用。

- 普通方法倾向于从不同逻辑中抽出相同代码块，
- 函数式接口倾向于在相同逻辑中插入不同代码块。

线程池与多线程

多线程即可以在同一时间内同时处理多个任务。其本质上是由处理器在多个任务之间快速来回切换处理实现的，因此并不是绝对的同步处理。

当多个线程操作同一块资源的时候，可能会导致这块资源的数据发生错乱，因此在使用共享的资源区时需要加锁处理。

线程有7种状态：**新建、就绪、运行、等待、阻塞、超时等待、停止**

创建线程的3种方式

1.基础 Thread 类

创建线程类

```Java
public class ComThread extends Thread{

    @Override
    public void run() {
        System.out.println("线程执行内容：启动了一个线程...");
    }

}
```

启动线程

```Java
// 启动两个同样的线程
new ComThread().start();
new ComThread().start();
```

分析

Thread 实际上是 Runnabe 接口的实现类，并且自身使用本地方法 start 启动线程，使用它的子类创建线程扩展性不强，因为 java 不支持多重继承。

2.实现 Runnabe 接口

创建线程类

```Java
public class ComRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("线程执行方法：启动了一个线程...");
    }
}
```

启动线程

```Java
new Thread(new ComRunnable()).start();
new Thread(new ComRunnable()).start();
```

分析

实际上 Thread 类通过构造方法传入了一个 Runnable 类型的参数并作为属性，并在自身的 run 方法中调用了此属性的 run 方法，这是一个典型的代理模式实现。

使用 Runnable 接口可以使线程类拓展性较强，不过 Runnable 执行的线程方法不允许返回内容。

3.实现 Callable<>接口

创建线程类：

```Java
public class RunCallable implements Callable {
    @Override
    public Object call() throws Exception {
        System.out.println("启动了一个线程,并返回0...");
        return 0;
    }
}
```

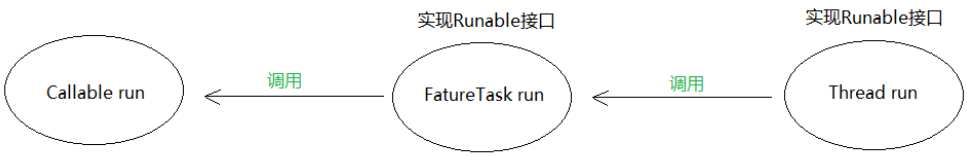
启动线程

```Java
FutureTask<String> task = new FutureTask<String>(new ComCallable());
// 启动线程
```

```
new Thread(task).start();
// 获取返回值
task.get();
```

分析

将Callable放在FutureTask类中，FutureTask是Runnable的子类，因此Thread的run方法调用了FutureTask的run方法，在FutureTask的run方法中调用了。



线程池

1.线程池构建

线程池可以帮助我们操作多个线程，从而避免线程使用不当造成的资源浪费。线程池管理线程，线程池内部使用Queue队列存放线程。

线程执行器

Java

```
// 执行器可以帮助我们创建一些已有的线程池实现
Executor executor = Executors.newSingleThreadExecutor();

// 执行线程，使用execute()方法执行线程类，下面使用lombda表达式简化语法，因为runnable可作为函数式接口
executor.execute(()-> System.out.println("执行了一个线程"));
```

自定义线程池

Java

```
// 最多2-5个线程，空闲存活20s 放在TaskQueue队列中
Executor executor = new ThreadPoolExecutor(2,5,20, TimeUnit.SECONDS,new TaskQueue());
```

说明：线程池无非就是线程的管理器，通过Executor 类可以创建应用于多种场景的线程池。

2.线程池分类

jdk提供了5种线程池：

Java

```
1  // 单任务线程池：一次只执行一个线程
2  Executor executor = Executors.newSingleThreadExecutor();
3
4  // 队列线程池：超出的线程在队列中等待
5  Executor executor = Executors.newSingleThreadExecutor();
6
7  // 可缓存线程池：根据任务数动态回收（空闲60秒）、创建线程
8  Executor executor = Executors.newCachedThreadPool();
9
10 // 定时任务线程池：定时、周期执行
11 Executor executor = Executors.newScheduledThreadPool();
12
13 // java8新出：线程处理完处理其它任务
14 Executor executor = Executors.newWorkStealingPool();
```

3.线程池案例

1.使用缓存线程池批量处理list所有元素

Java

```
1  public class ThreadHandler {
```

```
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
        handListByExecutor(list,3);
    }

    /**
     * 多线程批量处理
     */
    public static void handListByExecutor(List<Integer> list, int maxThNum){
        if (ObjectUtil.isEmpty(list)){
            return;
        }
        // 使用线程池处理，保存数据
        Executor executor = Executors.newCachedThreadPool();
        // 前9个线程分别处理的数量、最后一个线程处理的数量
        int oneThNum = list.size() / (maxThNum - 1);
        int endThNum = list.size() % (maxThNum - 1);
        for (int i = 0; i < (maxThNum - 1); i++) {
            // 每个线程处理的起始位置
            int start = i * oneThNum;
            executor.execute(()-> handlerIntervalElement(list,start,oneThNum));
        }
        executor.execute(()-> handlerIntervalElement(list,oneThNum * (maxThNum -1),endThNum));
    }

    /**
     * 单个线程的执行内容,分区间处理
     */
    public static void handlerIntervalElement(List<Integer> list,int start,int num){
        for (int i = 0; i < num; i++) {
            int index = start + i;
            System.out.println(list.get(index));
        }
    }

}
```

4.多线程事务

使用多线程插入数据到数据库中，如果失败，则回滚事务。

独立事务

Java

```
// 只有报错的线程回滚，其余不回滚

public class ThreadHandler {
    Executor executor = Executors.newCachedThreadPool();
    executor.execute(()-> handlerIntervalElement(list,start,oneThNum));
    executor.execute(()-> handlerIntervalElement(list,start,oneThNum));
}
```

整体事务

Java

```

```

测试

–

Java

```

```

使用案例

异步调用

方法的执行过程一般是从前到后顺序，后面的代码需要等待前面的代码执行后才会执行。因此会出现当某一段代码执行时间过长，但这段代码的执行结果并不会对方法结果产生影响时，我们就需要"跳过"这段代码而执行后续代码，且不影响这段代码的执行。

多线程方式

启动一个新的线程执行耗时较长的代码：

```Java
public void come(){

  // 前置代码
  Executors.newSingleThreadExecutor().execute(()->{
      // 耗时的代码
  });
  // 后续代码
}
```

获取方法调用者

当一个方法被调用，我们可以通过这个方法去获取调用这个方法的方法等信息。

我们可以利用这个方式为目标方法添加更详细的日志信息。

```Java
package com.jhy.getStackTraceDemo;

import java.util.Map;

public class DemoTest {

    public static void main(String[] args) {
        DemoTest demo = new DemoTest();
        demo.testOne();
        System.out.println("++++++++++++++++++++++++++++++++");
        demo.testTwo();
        System.out.println("++++++++++++++++++++++++++++++++");
        demo.testThree();
    }

    /**
     * 借助getAllStackTraces
     */
    public void testOne() {
        StackTraceElement[] stackTraceElement = Thread.getAllStackTraces().get(Thread.currentThread());
        for (int i = 0; i < stackTraceElement.length; i++) {
            System.out.println(Thread.currentThread().getName() + "::::::" + stackTraceElement[i].getClassName() + "."
                    + stackTraceElement[i].getMethodName());
        }
    }

    /**
     * 借助currentThread()
     */
    public void testTwo() {
        StackTraceElement[] stackTraceElement = Thread.currentThread().getStackTrace();
        for (int i = 0; i < stackTraceElement.length; i++) {
            System.out.println(Thread.currentThread().getName() + "::::::" + stackTraceElement[i].getClassName() + "."
                    + stackTraceElement[i].getMethodName());
        }
    }

    /**
     * 借助Exception
     */
    public void testThree() {
        StackTraceElement[] stackTraceElement=new Exception().getStackTrace();
        for (int i = 0; i < stackTraceElement.length; i++) {
            System.out.println(Thread.currentThread().getName() + "::::::" + stackTraceElement[i].getClassName() + "."+ stackTraceElement[i]
        }
    }

}
```

结果：实际可以获取更多信息，此处仅用类名举例。

```ABAP
main::::::java.lang.Thread.dumpThreads
main::::::java.lang.Thread.getAllStackTraces
main::::::com.jhy.getStackTraceDemo.DemoTest.testOne
main::::::com.jhy.getStackTraceDemo.DemoTest.main
++++++++++++++++++++++++++++++++
```

```
main::::::java.lang.Thread.getStackTrace
main::::::com.jhy.getStackTraceDemo.DemoTest.testTwo
main::::::com.jhy.getStackTraceDemo.DemoTest.main
++++++++++++++++++++++++++++++++
main::::::com.jhy.getStackTraceDemo.DemoTest.testThree
main::::::com.jhy.getStackTraceDemo.DemoTest.main
```