Java的一些其他特性

泛型

- 作为一种静态类型语言，Java对参数需要固定类型，虽然增加了安全性，但却失去了灵活性
  - 当然可以通过重载来增加函数对与不同参数的支持，但是这样会增加很多冗余代码，因为很多时候我们对于不同类型的参数处理逻辑是一致的，这就是"参数多态"
- 通过引入"泛型"（generics）的概念，Java可以将类型也作为一种参数，从而实现参数多态

泛型的语法

注意这里T、E可以是其他符号，代表类型参数。

- 泛型类的定义：[modifiers] class className <T, E, ...> {...}
- 泛型接口的定义：[modifiers] interface interfaceName <T, E, ...> {}
- 泛型方法的定义：[modifiers] <T, E, ... > returnType *functionName* (parameterList) {...}

例子

形式类型参数

```
public class GenericsSample<T, E, F> {
    private T obj;
    protected E something;
    protected F otherthing;
    public T getObj(){
        return obj;
    }
    public void set(E something, F otherthing){
        this.something = something;
        this.otherthing = otherthing;
    }
    public void setObj(T obj){
        this.obj = obj;
    }
    public static void main(String[] args){
        GenericsSample<String, Integer, Double> t1 = new GenericsSample<>();
        GenericsSample<Integer, Long, List> t2 = new GenericsSample<Integer, Long, List>();
        t1.setObj("Tom");
        t2.set(12L, new ArrayList<Integer>(Arrays.asList(100, 200, 300)));
    }
}
```

这些形式类型和普通类型用法一样

实际类型参数，必须为对象类型

构造对象时类型可显式声明，也可缺省

例子

```
public interface GenericsInterface<E> {
    E getAge();
    void printAge(E e);
}


public class Sample_implments implements GenericsInterface<Integer> {
    public Integer getAge() {
        return 0;
    }
    public void printAge(Integer o) {
        System.out.println("年龄：" +o);
    }
}
```

例子

在返回类型前

```
public class GenericsMethod {
    public static <E> void display(E[] list){
        for(E e : list){
            System.out.print(e + " ");
        }
        System.out.println();
    }
    public <E, V> void display(E[] list, V[] list2){
        for(E e : list){
            System.out.print(e + " ");
        }
        System.out.println();

        for(V e : list2){
            System.out.print(e + " ");
        }
        System.out.println();
    }
    public static void main(String[] args){
        Integer[] num = {1, 2 ,3, 4, 5};
        String[] strs = {"赤", "橙","黄", "绿", "青", "蓝","紫"};
        GenericsMethod.display(num);
        GenericsMethod.display(strs);
        GenericsMethod app = new GenericsMethod();
        app.display(num, strs);
    }
}
```

泛型的类型可以做一些限制

- 语法： <T extends anyClass>

就可以可以限制 T 的实际参数必须是anyClass的子类，或者实现了anyClass接口的类， 其中anyClass无论是类或者接口，都必须用extends关键字。这里anyClass也被称为该形式类型的限定类型（ bounding type ）

```
class GeneralType <T extends Number> {
    T obj;
    public GeneralType (T obj){
        this.obj = obj;
    }
    public T getObj(){
        return obj;
    }
}
```

```
public class Test {
    public static void main(String[] args){
        GeneralType<Integer> num = new GeneralType<Integer>(5);
        System.out.println("给出的参数是： "+ num.getObj());
    }
}
```

```
public class Test {
    public static void main(String[] args){
        GeneralType<String> num = new GeneralType<String>("5");
        System.out.println("给出的参数是： "+ num.getObj());
    }
}
```

类型擦除（Erasure）

- 其实Java的泛型都是伪泛型，其为了能够后向兼容Java旧时代（Java 4.0及以前，还没有泛型时）代码，并不是运行时进行泛型的支持，而是通过编译器在编译阶段对类型进行擦除

```
class Foo<T> {
        T x;
        T someMethod(T y) { ... }
}

public Class Test{
  public static void main(String[] args){
      Foo<Integer> q = new Foo<Integer>();
      Integer r = q.someMethod(s);
  }
}
```

Actually

```
class Foo {
        Object x;
        Object someMethod(Object y) { ... }
}

public Class Test{
  public static void main(String[] args){
      Foo q = new Foo();
      Integer r = (Integer) q.someMethod( (Integer) s);
  }
}
```

试试javap -verbose看看

类型擦除（Erasure）

- 如果类型是受限的，则会替换为其限定类型

```
class Foo<T extends Number> {
        T x;
        T someMethod(T y) { ... }
}

public Class Test{
  public static void main(String[] args){
      Foo<Integer> q = new Foo<Integer>();
      Integer r = q.someMethod(s);
  }
}
```

Actually

```
class Foo {
        Number x;
        Number someMethod(Number y) { ... }
}

public Class Test{
  public static void main(String[] args){
      Foo q = new Foo();
      Integer r = (Integer) q.someMethod( (Integer) s);
  }
}
```

一些注意点

- 泛型的实际参数类型不能是原始类型（因为都会被擦除为Object或其限定类）
- instanceof 判断不了泛型，比如： C<String> a = ... ; a instanceof C<Integer> ; //Error
- 不能用泛型创建对象，即 T a = new T(); （因为本质上是new Object()！而不是想要的类对象）
- 不能用泛型创建数组对象，即 T[] a = new T[size]；（与上述原因一致）
- 不能声明静态的泛类型的变量，如：

```
public class Singleton<T>
{
    public static T singleInstance; // ERROR
}
```

思考擦除后发生什么？

泛型的类型通配符（Wildcard）

- 之前一个泛型对象名只能引用同一种泛型对象，如 GeneralType<String> a 只能指向GeneralType<String>的对象。
- 如果要使用同一个泛型对象名去引用不同的泛型对象，就需要使用通配符 "?" 创建泛型类对象。
  - 但要求不同泛型对象的类型实参必须是某个类或者其子类，或实现某个接口

```
泛型类名 <? extends T> x = null;
```

例子

```
class GeneralType <T>{
    T obj;
    public void setObj (T obj){
        this.obj = obj;
    }
    public T getObj(){
        return obj;
    }
}
```

只能是Number的子类，如果没有extends，那么默认extends Objects

```
GeneralType <? extends Number> x = null;
x = new GeneralType <Long> ();
x = new GeneralType <Integer> ();
Number a = x.getObj(); //Correct
x.setObj(Integer.valueOf(1)); //Error
```

对于JVM而言，其无法确定x的对象的实际参数类型到底指向Number哪个子类，

如果实际是Integer类型，此时写入Long修改会出错（Integer和Long不是父子关系！）

但是读取是不影响的（都以Number类型的方式读取）

泛型的类型通配符

- 除了可以利用extends限定实际类型参数是某个类型的子类外（设置上限），还可以用super 限定其是某个类的父类（设置下限）
- 语法： 泛型类型 <? super anyclass> x = null;

例子

```
class GeneralType <T>{
    T obj;
    public void setObj (T obj){
        this.obj = obj;
    }
    public T getObj(){
        return obj;
    }
}
```

```
GeneralType <? super Integer> x = null;
x = new GeneralType <Object> ();
x = new GeneralType <Number> ();
Number a  = x.getObj(); //Error
x.setObj(Integer.valueOf(1)); //Correct
```

只能是Integer的父类

编译器并不知道到底是Integer的哪个父类

（注：这里可以强制转化为Object类使得编译通过）

可以写入Integer，因为其赋给任何其父类都是合法的

其他特性

对象的一些特殊函数

- 对于对象类而言
  - String toString()，默认返回 class名称 + @ + hashCode 的十六进制字符串，重写可以定制一个对象的具体打印结果（System.out.println()）
  - boolean equals()，判断两个对象是否相等，默认两个对象是否是同一个对象，但一般将其重写为连个对象的内容是否相等（自己定义"内容相等"）。比如字符串类就重写了这个函数，只要字符串包含的内容相等，equals()就返回 True
  - int hashCode()，返回对象的散列值（hash code），当重写了equals()之后，必须也重写该函数，使得equals()为真的两个对象，hashCode()的值也必须一致 （hashCode是hashMap的核心）。

Lambda表达式

lambda表达式不可对外面变量作修改！

```
parameter -> expression
(parameter1, parameter2, ...) -> { code block }
```

例子：

```
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3,5,7});
int factor = 2;
primes.forEach(element -> { factor++; });
```

error

也可以将其赋给一个变量：

```
Consumer<Integer> a =  (n) -> { System.out.println(n); }
numbers.forEach(a);
```

Lambda表达式

- Java内置的存储lambda函数的变量类型：
  - Consumer<T> 接受一个输入，不返回值
  - Supplier<T> 不接受输入，返回一个值
  - Function<T, R> 接受一个输入，返回一个输出
  - BiFunction<T, U, R>接受两个输入，返回一个输出

Lambda表达式

- 也可以自定义能够存储Lambda表达式的变量
  - 前提是该变量的类型是只有一个方法的接口。Lambda 表达式应该具有与该方法相同数量的参数和相同的返回类型。

```
interface Lambda {
    int display(int number);
}
Lambda lambda = number -> {
    return number * number;
};

System.out.println(lambda.display(20));
```

Lambda表达式

- 柯里化
  - Java也可以将Lambda表达式柯里化

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> curried = a -> b -> c ->(a + b +c);

// 调用柯里化函数
int result = curried.apply(1).apply(2).apply(3);
System.out.println("Result: " + result);
```

流（Stream）

- Java流是一种对集合进行链状流式的操作，但是操作不会改变原来的数据（函数式编程思想）

例子：

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());
```

流（Stream）

map

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
List<Integer> squaresList = numbers.stream().map( i -> i*i).collect(Collectors.toList());
squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

filter

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
long count = strings.stream().filter(string -> string.isEmpty()).count();
```

sorted

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
List<Integer> sortedList = numbers.stream().sorted().collect(Collectors.toList());
List<Integer> sortedList = numbers.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
numbers.stream().sorted((a, b) -> {return a.compareTo(b);}).forEach( n ->  System.out.println(n) );
```

reduce

```
int reduced = IntStream.range(1, 4).reduce((a, b) -> a + b).getAsInt();
```

流（Stream）

Iterate

```
Stream<Integer> stream = Stream.iterate(0, (x) -> x + 2).limit(6);
stream.forEach(System.out::println);
```

Match

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
boolean allMatch = list.stream().allMatch(e -> e > 3); //false
boolean anyMatch = list.stream().anyMatch(e -> e > 3);  //true
boolean noneMatch = list.stream().noneMatch(e -> e > 10); //true
```

Min, Max

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
Integer max = list.stream().max((a, b) -> a.compareTo(b)).get(); //5
Integer min = list.stream().min(Integer::compareTo).get(); //1
```

并行流

- 串行流（Sequential Stream）：默认情况下，流是串行的，即所有操作在单线程中依次执行。
- 并行流（Parallel Stream）：通过内部实现的 ForkJoin 框架，将流分成多个子任务，并在多个线程上并行执行。

并行流

- 并行流的写法和串型流类似，函数式的写法使的我们不需要关心太多并发的细节（当然，底层实现中这些细节还在）

```
import java.util.Arrays;
import java.util.List;
public class ParallelStreamExample {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        // 使用并行流计算平方和
        int sumOfSquares = list.parallelStream().map(i -> i * i).sum();
        System.out.println("Sum of squares: " + sumOfSquares);
    }
}
```

反射（Reflection）

- 反射是一种在运行时可以检视自身程序和操纵程序内部属性的一种语言特性。
  - 比如对于Java而言，反射可以使其运行时动态的加载类并获取类的详细信息，从而可以操作类和对象的属性和方法

例子

```
import java.lang.reflect.*;

    public class DumpMethods {
        public static void main(String args[])
        {
            try {
                Class c = Class.forName("java.util.Stack");
                Method m[] = c.getDeclaredMethods();
                for (int i = 0; i < m.length; i++)
                System.out.println(m[i].toString());
            }
            catch (Throwable e) {
                System.err.println(e);
            }
        }
    }
```

Output:

```
public boolean java.util.Stack.empty()
public synchronized java.lang.Object java.util.Stack.peek()
public synchronized int java.util.Stack.search(java.lang.Object)
public java.lang.Object java.util.Stack.push(java.lang.Object)
public synchronized java.lang.Object java.util.Stack.pop()
```

例子

```
import java.lang.reflect.*;
    public class TestInvoke {
        public int add(int a, int b) {
            return a + b;
        }
        public static void main(String args[])
        {
            try {
                Class cls = Class.forName("TestInvoke"); //完整的包名.ClassName
                Class partypes[] = new Class[2];
                partypes[0] = Integer.TYPE;
                partypes[1] = Integer.TYPE;
                Method meth = cls.getMethod("add", partypes);
                TestInvoke methobj = new TestInvoke();
                Object arglist[] = new Object[2];
                arglist[0] = Integer.valueof(37);
                arglist[1] = Integer.valueof(47);
                Object retobj  = meth.invoke(methobj, arglist);
                Integer retval = (Integer)retobj;
                System.out.println(retval.intValue());
            }
            catch (Throwable e) {
                System.err.println(e);
            }
        }
    }
```

```
import java.lang.reflect.*;

    public class TestConstruct {
        public TestConstruct(){
        }
        public TestConstruct(int a, int b){
            System.out.println(
              "a = " + a + " b = " + b);
        }
        public static void main(String args[])
        {
            try {
                Class cls = Class.forName("TestConstruct");
                Class partypes[] = new Class[2];
                 partypes[0] = Integer.TYPE;
                 partypes[1] = Integer.TYPE;
                 Constructor ct = cls.getConstructor(partypes);
                 Object arglist[] = new Object[2];
                 arglist[0] = new Integer(37);
                 arglist[1] = new Integer(47);
                 Object retobj = ct.newInstance(arglist);
            }
            catch (Throwable e) {
                System.err.println(e);
            }
        }
    }
```

反射的好处

- 反射是一种"动态"特性，其使的我们无需在编译阶段知道类的信息，也可以在运行时加载类、调用这个类的方法、甚至修改和访问类的字段
- 这是实现Java很多框架和库的基础
  - 反射是许多框架和库（如 Spring、Hibernate、JUnit）的基础
- 一个例子：你如何测试大家写的代码？
  - 没有反射。你需要提前知道大家写的函数，然后"显式"的调用他们，但是有了反射就轻松多了

利用反射实现一个简易测试框架*

注解

- 注解（Annotation）是放在Java源码的类、方法、字段、参数前的一种特殊"注释"
- 与普通注释直接被编译器忽视不同，注解则可以被编译器打包进入class文件，因此，注解是一种用作标注的"元数据"。
- Java标注可以通过反射来获取标注的内容

一些内部注解

- @Override
  - 标记方法重写父类方法，编译器要对重写方法进行检查
- @SuppressWarnings
  - 抑制编译器警告
- @Deprecated
  - 标记过时的方法、类等，编译器发出警告
- @Retention
  - 元注解，可以用来自定义其他注解

利用反射实现一个简易测试框架

- 定义两个注解：@Test 和 @BeforeEach。@Test 用于标记测试方法，@BeforeEach用于标记在每个测试方法执行前运行的方法。

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
}

@Retention(RetentionPolicy.RUNTIME)
public @interface BeforeEach {
}
```

利用反射实现一个简易测试框架

- 创建待测类，用之前自定义的注解来标记你想要测的类

```
public class MyTests {
    @BeforeEach
    public void setup() {
        System.out.println("Setup before each test");
    }
    @Test
    public void test1() {
        System.out.println("Running test1");
    }
    @Test
    public void test2() {
        System.out.println("Running test2");
    }
    public void notATest() {
        System.out.println("This is not a test method");
    }
}
```

利用反射实现一个简易测试框架

- 利用反射，我们可以动态地扫描找测试类，然后找到这些标记了 `@Test` 和 `@BeforeEach` 的方法，然后执行他们

```
Class<?> testClass = Class.forName("MyTests");
Object testInstance = testClass.getDeclaredConstructor().newInstance();
Method[] methods = testClass.getDeclaredMethods();
Method beforeEachMethod = null;

// 找到 @BeforeEach 方法
for (Method method : methods) {
    if (method.isAnnotationPresent(BeforeEach.class)) {
        beforeEachMethod = method;
        break;
    }
}
```

```
// 运行标记了 @Test 的方法
for (Method method : methods) {
    if (method.isAnnotationPresent(Test.class)) {
        if (beforeEachMethod != null) {
            beforeEachMethod.invoke(testInstance);
        }
        method.invoke(testInstance);
    }
}
```

一些常见的有用的Java库

一些常见的有用的Java库

	描述	常用的类
java.lang	语言包（默认引入）	Object、String、Math、System、Exception、Class、Thread、Throwable
java.io	输入输出流的文件包	OutputStream、InputStream、PrintWriter、File、FileInputStream、FileOutputStream、BufferedReader、BufferedWriter
java.util	实用工具包	Date、Calendar、List、Map、Set、Stack、Random、Currency、Locale
java.net	网络包	URL、Socket、ServerSocket、HttpCookie
java.sql	数据库处理包	Connection、Statement、PreparedStatement、ResultSet
java.text	文本处理包	Format、DateFormat、NumberFormat

此外还有一些常用的第三方Java库：Junit、Log4j、JavaFx、Weka、Hadoop、SpringBoot、Android...

Any questions ?