新特性概述

> ### 本节目标
>
> 1. 枚举
> 2. 注解
> 3. 接口定义加强
> 4. Lambda表达式
> 5. 方法引用
> 6. 内建函数式接口

1.枚举

1.1 问题引出

要认识枚举，首先我们回顾一下多例设计模式特点：构造方法私有化，类内部需要提供若干个实例化对象，后面通过static方法返回。

范例：定义一个描述颜色基色的多例类。

```
package www.bit.java.testdemo;

class Color {
    private String title ;
    public static final int RED_FLAG = 1 ;
    public static final int GREEN_FLAG = 2 ;
    public static final int BLUE_FLAG = 3 ;
    private static final Color RED = new Color("RED") ;
    private static final Color GREEN = new Color("GREEN") ;
    private static final Color BLUE = new Color("BLUE") ;
    private Color(String title) {
        this.title = title ;
    }
    public static Color getInstance(int ch){
        switch (ch) {
        case RED_FLAG :
            return RED ;
        case GREEN_FLAG :
            return GREEN ;
        case BLUE_FLAG :
            return BLUE ;
        default:
            return null ;
        }
    }
```

```
    @Override
    public String toString() {
        return this.title ;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(Color.getInstance(Color.BLUE_FLAG));
    }
}
```

以上做法是在JDK1.5以前的做法，这样做的目的是限制本类实例化对象的产生个数。但是从JDK1.5开始有了枚举，上述代码可以使用枚举来取代。

范例：基于枚举开发

```
package www.bit.java.testdemo;

enum Color {
    RED,GREEN,BLUE
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(Color.BLUE);
    }
}
```

实际上枚举就是一种高级的多例设计模式。

1.2 Enum类

虽然JDK1.5提供了enum关键字，但是enum并不是一种新的结构，相反，它只是对一种类型的包装：使用enum关键字定义的枚举类本质上就相当于一个Class定义的类，继承了java.lang.Enum父类。

在Enum类里面有以下方法：

1.构造方法

```
protected Enum(String name, int ordinal)
```

2.取得枚举名字：

```
public final String name()
```

3.取得枚举序号：

```
public final int ordinal()
```

范例：观察上面方法的使用

```
enum Color {
    RED,GREEN,BLUE
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(Color.BLUE.ordinal() + "=" +Color.BLUE.name());
    }
}
```

通过以上代码我们发现，所有的枚举类默认继承于Enum类

在枚举操作中还有一个方法可以取得所有的枚举数据：values()返回的是一个枚举的对象数组。

范例：取得所有枚举数据

```
enum Color {
    RED,GREEN,BLUE
}
public class TestDemo {
    public static void main(String[] args) {
        for (Color temp : Color.values()) {
            System.out.println(temp.ordinal()+" = " +temp.name());
        }
    }
}
```

面试：请解释enum与Enum区别：

> - enum是一个关键字，使用enum定义的枚举类本质上就相对于一个类继承了Enum这个抽象类而已。

1.3 定义结构

虽然枚举等同于多例设计，但是多例设计是在一个类中产生的，所以该类中可以定义更多的属性或方法。所以在枚举设计的时候考虑到这些因素，提出了更为全面的设计方案：可以在枚举中定义属性、方法、实现接口。

范例：在枚举中定义更多的结构

```
package www.bit.java.testdemo;

enum Color {
    RED("红色"),GREEN("绿色"),BLUE("蓝色")  ; // 如果定义有很多内容，枚举对象必须写在第一行
```

```
    private String title ;
    private Color(String title) { // 构造方法私有化
        this.title = title ;
    }
    @Override
    public String toString() {
        return this.title ;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(Color.BLUE);
    }
}
```

枚举还可以实现接口，这样枚举中的每一个对象都变成了接口对象。

范例：枚举实现接口

```
package www.bit.java.testdemo;
interface IColor {
        public String getColor() ;
}
enum Color implements IColor{
        RED("红色"),GREEN("绿色"),BLUE("蓝色")  ; // 如果定义有很多内容，枚举对象必须
写在第一行
        private String title ;
        private Color(String title) { // 构造方法私有化
                this.title = title ;
        }
        @Override
        public String toString() {
                return this.title ;
        }
        @Override
        public String getColor() {
                return this.title ;
        }
}
public class TestDemo {
        public static void main(String[] args) {
                IColor iColor = Color.BLUE ;
                System.out.println(iColor.getColor());
        }
}
```

1.4 枚举应用

枚举的最大特点是只有指定的几个对象可以使用。

例如：定义一个表示性别的枚举类，只能有两个对象。

```
package www.bit.java.testdemo;
enum Sex {
        MALE("男"),FEMALE("女") ;
        private String title ;
        private Sex(String title) {
                this.title = title ;
        }
        @Override
        public String toString() {
                return this.title ;
        }
}
class Person {
        private String name ;
        private int age ;
        private Sex sex ;
        public Person(String name, int age, Sex sex) {
                super();
                this.name = name;
                this.age = age;
                this.sex = sex;
        }
        @Override
        public String toString() {
                return "Person [name=" + name + ", age=" + age + ", sex=" +
sex + "]";
        }

}
public class TestDemo {
        public static void main(String[] args) {
                Person per = new Person("张三", 20, Sex.MALE) ;
                System.out.println(per);
        }
}
```

另外需要注意的是，枚举本身还支持switch判断。

范例：switch使用枚举

```
package www.bit.java.testdemo;
enum Sex {
        MALE,FEMALE
}
```

```
public class TestDemo {
        public static void main(String[] args) {
                switch(Sex.MALE) {
                case MALE :
                        System.out.println("男人");
                        break ;
                case FEMALE :
                        System.out.println("女人");
                        break ;
                }
        }
}
```

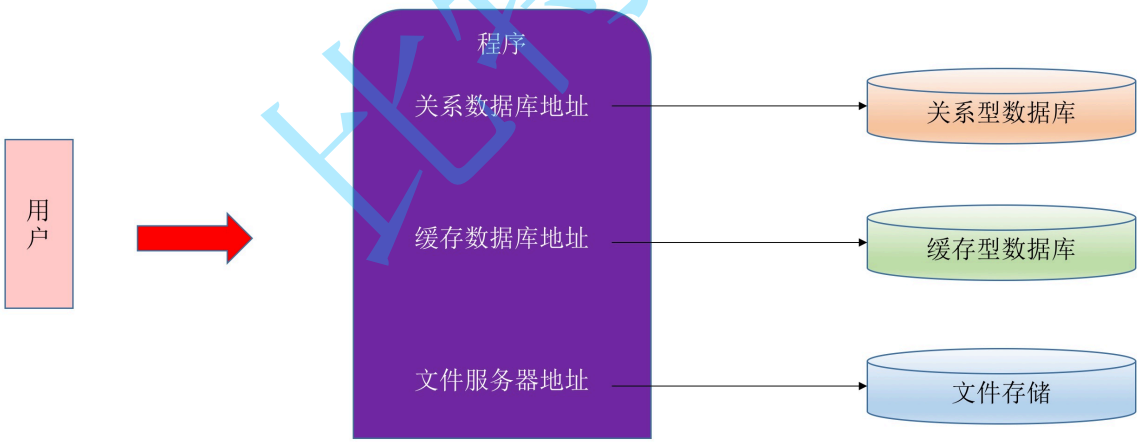
2.注解（Annotation）

2.1 问题引出

Annotation可以说是JDK发展的重要技术，从现在的开发来讲，Annotation的使用已经变得非常常见。

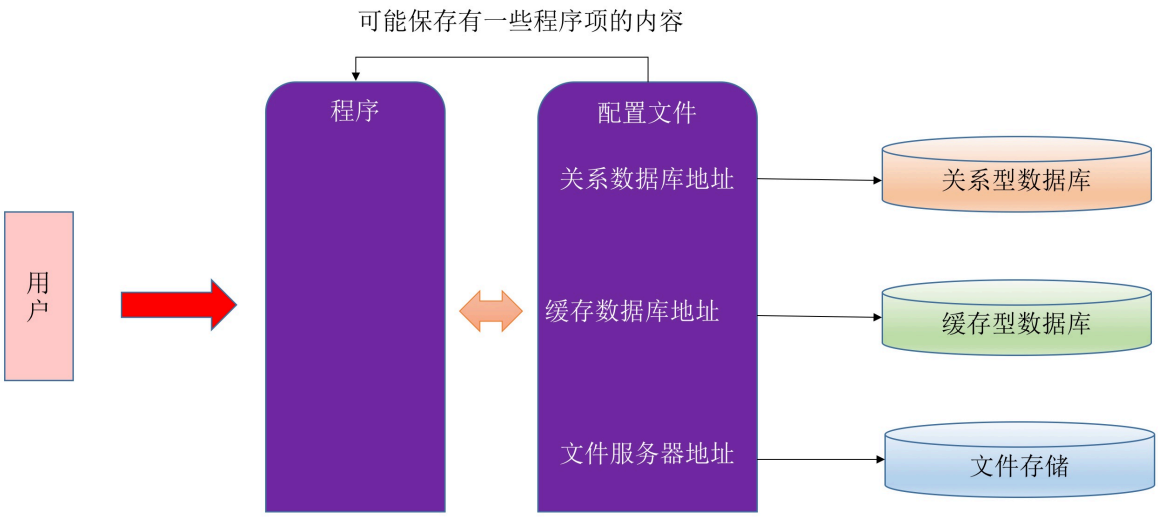
要想理解Annotation的作用，对于软件项目的开发往往会经历以下过程：

> - 在进行软件项目的开发过程之中，会将所有使用到的第三方信息或者程序有关的操作都写在程序里：



如果现在服务器地址改变了，意味着需要你更改程序源代码。而这个工程就相当庞大了。

> - 使用一个配置文件，程序运行的时候要通过配置文件读取相关的配置操作。



如果此时要想更改一些配置，那么只需要更改配置文件即可，也就是可以在不修改源代码的前提下实现项目的变更。

使用配置文件之后，虽然代码的维护方便了，但是在开发里不方便。另外，对于非专业人士，很难去修改，并且一个项目的配置文件可能会非常多。于是后来JDK提供了一个新的做法，将配置写回到程序里，但是与传统程序作了区分，这样就形成了注解的概念。

但是需要注意的是，并不是说写了注解以后就可以不使用配置文件了，只是现在使用的配置文件少了。

本次我们先来开JDK提供的三个内置注解：@Override、@Deprecated、@SuppressWarnings

2.2 准确覆写

`@Override`

方法覆写：发生在继承关系之中，子类定义了与父类的方法名称相同、参数列表相同、返回值类型相同称为方法的覆写，被覆写的方法不能够拥有比父类更为严格的访问控制权限。

范例：观察问题

```
package www.bit.java.testdemo;
class Person {
   // 由于输入错误导致覆写失败
        public String tostring() { // 现在希望对toString()方法覆写
                return "小比特们" ;
        }
}
public class TestDemo {
        public static void main(String[] args) {
                System.out.println(new Person());
        }
}
```

这个时候不叫覆写，属于自己定义了一个扩展的方法，最为重要的是，这个问题在程序编译的时候根本无法显示出来。为了保证覆写方法的严格，可以使用一个注解(@Override)来检测：如果该方法确定成功覆写，则不会有语法错误；如果没有成功覆写，则认为是语法错误。

范例：使用@Override

```
package www.bit.java.testdemo;
class Person {
        @Override // 追加了此注解后将明确的表示该方法是一个覆写的方法，如果覆写错误会出现语法错误
        public String tostring() {
                return "小比特们" ;
        }
}
public class TestDemo {
        public static void main(String[] args) {
                System.out.println(new Person());
        }
}
```

当你覆写的方法正确的时候，就表示没有任何问题。

```
package www.bit.java.testdemo;
class Person {
        @Override
        public String toString() {
                return "小比特们" ;
        }
}
public class TestDemo {
        public static void main(String[] args) {
                System.out.println(new Person());
        }
}
```

在Eclipse中，只要是你覆写的方法基本都会自动添加@Override注解。

2.3 声明过期

`@Deprecated`

如果现在你有一个程序类，从项目1.0版本到一直到99.0版本一直都在使用着，但是从100.0版本后发现该程序类可能会产生问题，那么这个时候你能直接删除这个类换一个新的吗?

绝对不能，因为其他旧版本还在使用这个类，并且这个类在旧版本中没有问题。这个时候就希望在进行新版本扩展的时候不再去使用这个不建议的类，所以加一个过期的注解(@Deprecated)。

范例：观察过期操作

```
package www.bit.java.testdemo;
class Person {
        @Deprecated // 表示该方法已经不建议使用了，但是即便你使用了也不会出错
        public Person() {}
        public Person(String name) {}
        @Deprecated
        public void fun() {}
}
public class TestDemo {
        public static void main(String[] args) {
                Person person = new Person() ; // 明确的标记出过期
                person.fun();
                person = new Person("") ;
        }
}
```

这种过期的处理操作往往出现在一些平台支持的工具上，例如：JDK就是一个平台，所以在JDK中有很多方法都推荐用户不再使用了。

2.4 压制警告

`@SuppressWarnings`

当调用了某些操作可能产生问题的时候就会出现警告进行，但是警告信息并不是错，这个时候又不想总提示警告，这个时候可以使用压制警告。

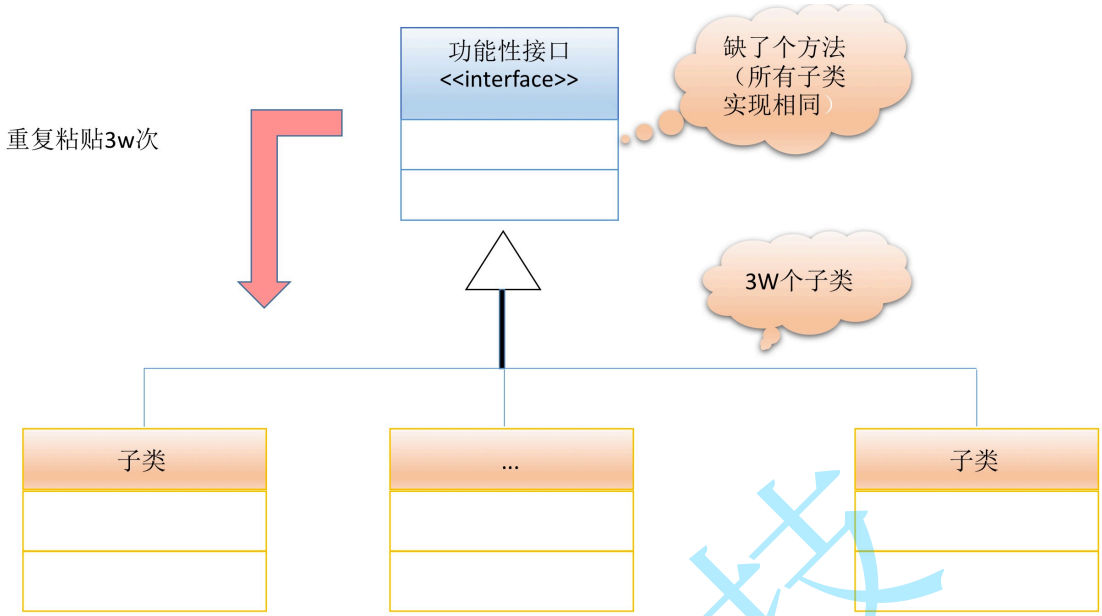
范例：观察压制警告

```
package www.bit.java.testdemo;
class Person <T> {
        @Deprecated // 表示该方法已经不建议使用了，但是即便你使用了也不会出错
        public Person() {}
        public Person(String name) {}
        @Deprecated
        public void fun() {}
}
public class TestDemo {
        @SuppressWarnings({ "rawtypes", "unused" })
        public static void main(String[] args) {
                Person person = new Person() ;
        }
}
```

以上三种Annotation是JDK默认支持的程序类中使用的，以后会接触到一些功能性的Annotation。Java还支持我们自定义注解，这个我们后续讲到JavaEE部分会详细介绍。

3.接口定义加强

从JDK1.8开始，支持的新特性非常多，并且打破了很多原有的设计方案，其中具有最大变化的就是接口。但是千万要记住一点：现阶段你们自己写的接口请遵循之前的原则。



造成此种现象的核心问题在于：接口只是一个方法的声明，而没有具体的方法实现，所以随着时间的推移，如果真的出现了以上的问题，那么该接口就将无法继续使用。

从JDK1.8开始，为了解决这样的问题，专门提供了两类新的结构：

- **可以使用default来定义普通方法，需要通过对象调用**
- **可以使用static来定义静态方法，通过接口名就可以调用**

范例：定义普通方法

```
package www.bit.java.testdemo;
interface IMessage {
        public default void fun() { // 追加了普通方法,有方法体了
                System.out.println("Hello IMessage");
        }
        public void print() ;
}
class MessageImpl implements IMessage {

        @Override
        public void print() {
                System.out.println("Hello MessageImpl");
        }

}
public class TestDemo {
        public static void main(String[] args) {
                IMessage message = new MessageImpl() ;
                message.print();
```

```
                message.fun();
        }
}
```

范例：定义static方法

```
package www.bit.java.testdemo;
interface IMessage {
        public default void fun() { // 追加了普通方法,有方法体了
                System.out.println("Hello IMessage");
        }
        // 可以直接由接口名称直接调用
        public static IMessage getInstance() {
                return new MessageImpl() ;
        }
        public void print() ;

}
class MessageImpl implements IMessage {

        @Override
        public void print() {
                System.out.println("Hello MessageImpl");
        }

}
public class TestDemo {
        public static void main(String[] args) {
                IMessage message = IMessage.getInstance() ;
                System.out.println(message);
                message.print();
                message.fun();
        }
}
```

整体来讲，接口感觉更像抽象类了，但是比抽象类更强大的在于：接口的子类依然可以实现多继承的关系，而抽象类依然保持单继承。

因为时间一长，许多的支持就会出现问题（量大的问题），这个时候为了解决这种扩充的问题，才追加了此类支持。但是此操作不属于标准设计，属于挽救设计。

4.Lambda表达式

Lambda是JDK1.8推出的重要新特性。很多开发语言都开始支持函数式编程，其中最具备代表性的就是`haskell`。

函数式编程和面向对象编程可以理解为两大开发阵营。很多人认为面向对象的概念过于完整，结构操作不明确。

范例：传统面向对象开发

```
package www.bit.java.testdemo;
interface IMessage {
        public void print() ; // 这是一个接口，接口中的抽象方法必须由子类覆写。
}

public class TestDemo {
        public static void main(String[] args) {
                IMessage message = new IMessage() { // 匿名内部类
                        @Override
                        public void print() { // 必须编写完整语法
                                System.out.println("Hello World");
                        }
                };
                message.print();
        }
}
```

对于此类操作有了更简化实现，如果采用函数式编程，则代码如下：

范例：函数式编程

```
package www.bit.java.testdemo;
interface IMessage {
        public void print() ; // 这是一个接口，接口中的抽象方法必须由子类覆写。
}

public class TestDemo {
        public static void main(String[] args) {
                // 函数式编程的使用，目的还是输出一句话
                IMessage message = () -> System.out.println("Hello World");
                message.print();
        }
}
```

面向对象语法最大的局限：结构必须非常完整。

要想使用函数式编程有一个前提：接口必须只有一个抽象方法，如果有两个抽象方法，则无法使用函数式编程。如果现在某个接口就是为了函数式编程而生的，最好在定义时就让其只能够定义一个抽象方法，所以有了一个新的注解： `@FunctionalInterface`

范例：使用`@FunctionalInterface`注解

```
package www.bit.java.testdemo;
@FunctionalInterface // 是一个函数式编程接口，只允许有一个抽象方法
interface IMessage {
        public void print() ; // 这是一个接口，接口中的抽象方法必须由子类覆写。
}

public class TestDemo {
        public static void main(String[] args) {
                // 函数式编程的使用，目的还是输出一句话
                IMessage message = () -> System.out.println("Hello World");
                message.print();
        }
}
```

语法如下:

```
(参数) -> 单行语句 ;
```

这个时候方法本身只包含有一行语句，直接写单行语句即可；如果有多行语句，就需要使用"{}"

语法如下:

```
(参数) -> {} ;
```

范例：多行语句

```
package www.bit.java.testdemo;
@FunctionalInterface // 是一个函数式编程接口，只允许有一个方法
interface IMessage {
```

```
        public void print() ; // 这是一个接口，接口中的抽象方法必须由子类覆写。
}

public class TestDemo {
        public static void main(String[] args) {
                // 函数式编程的使用，目的还是输出一句话
                IMessage message = () -> {
                        System.out.println("Hello World");
                        System.out.println("Hello World111");
                        System.out.println("Hello World11111");
                } ;
                message.print();
        }
}
```

如果现在你的表达式里只有一行进行数据的返回，那么直接使用语句即可，可以不使用return。

范例：直接进行计算

```
package www.bit.java.testdemo;
@FunctionalInterface
interface IMath {
        public int add(int x,int y) ;
}

public class TestDemo {
        public static void main(String[] args) {
                // 函数式编程的使用，目的还是输出一句话
                IMath msg = (p1,p2) -> p1+p2 ; // 只有一行返回
                System.out.println(msg.add(10, 20));
        }
}
```


行业之内已经有了一家公司专门基于java做了一套自己的函数式编程语言：Scala。（数据分析）

5.方法引用

从最初开始，只要是进行引用都是针对于引用类型完成的，也就是只有数组、类、接口具备引用操作。但是JDK1.8开始追加了方法引用的概念。实际上引用的本质就是别名。所以方法的引用也是别名的使用。而方法引用的类型有四种形式：

1. 引用静态方法：类名称::static 方法名称；
2. 引用某个对象的方法：实例化对象 :: 普通方法；
3. 引用某个特定类的方法： 类名称 :: 普通方法；
4. 引用构造方法： 类名称 :: new 。

注意：方法引用一般结合函数式编程使用

5.1 引用静态方法

String类的valueOf()方法

```
package www.bit.java.testdemo;
@FunctionalInterface // 是一个函数式编程接口，只允许有一个方法
interface IUtil<P,R> {
        public R switchPara(P p) ;
}

public class TestDemo {
        public static void main(String[] args) {
                IUtil<Integer,String> iu = String :: valueOf ; //进行方法引用
                String str = iu.switchPara(1000) ; // 相当于调用了
String.valueOf(1000)
                System.out.println(str.length());
        }
}
```

就相当于为方法起了个别名。

5.2 引用对象方法

String中的toUpperCase()方法为对象方法

```
package www.bit.java.testdemo;
@FunctionalInterface // 是一个函数式编程接口，只允许有一个方法
interface IUtil<R> {
        public R switchPara() ;
}

public class TestDemo {
        public static void main(String[] args) {
                IUtil<String> iu = "hello" :: toUpperCase ; // 进行方法引用
                System.out.println(iu.switchPara()); // 转换的就是这个"hello"
        }
}
```

5.3 引用类中普通方法

String有一个compareTo方法，此方法为普通方法

```
package www.bit.java.testdemo;
@FunctionalInterface
interface IUtil<R,P> {
        public R compare(P p1,P p2) ;
}
public class TestDemo {
        public static void main(String[] args) {
                IUtil<Integer,String> iu = String :: compareTo ;
                System.out.println(iu.compare("刘", "霍"));
        }
}
```

5.4 引用构造方法

```
package www.bit.java.testdemo;

class Person {
        private String name ;
        private int age ;
        public Person(String name, int age) {
                super();
                this.name = name;
                this.age = age;
        }
        @Override
        public String toString() {
                return "Person [name=" + name + ", age=" + age + "]";
        }

}

@FunctionalInterface
interface IUtil<R,PN,PA> {
        public R createPerson(PN p1,PA p2) ;
}
public class TestDemo {
        public static void main(String[] args) {
                IUtil<Person,String,Integer> iu = Person :: new;
                System.out.println(iu.createPerson("yuisama", 25)); // 相当于调
用Person类的构造方法
        }
}
```

这些都属于函数式编程的补充，但不作为我们现阶段的考虑。

6.内建函数式接口

Lamdba的核心在于：函数式接口。而函数式接口的核心：只有一个抽象方法。

`java.util.function`实际上函数式编程分为以下四种接口：

1. 功能型函数式接口：public interface Function<T, R> R apply(T t);
2. 供给型函数式接口： public interface Supplier T get();
3. 消费型函数式接口：public interface Consumer void accept(T t);
4. 断言型接口：public interface Predicate boolean test(T t);

6.1 功能型接口

```
package www.bit.java.testdemo;

import java.util.function.Function;

public class TestDemo {
        public static void main(String[] args) {
                Function<Integer, String> fun = String :: valueOf ;
                System.out.println(fun.apply(1000));
        }
}
```

功能型指的是你输入一个数据，而后将数据处理后进行输出。

实际上所有函数式接口里面都会有一些小的扩展，如果现在确定操作的数据是int，则可以使用IntFunction接口。

范例：使用IntFunction

```
package www.bit.java.testdemo;

import java.util.function.IntFunction;

public class TestDemo {
        public static void main(String[] args) {
                IntFunction<String> fun = String :: valueOf ;
                System.out.println(fun.apply(1000));
        }
}
```

6.2 供给型接口

```
package www.bit.java.testdemo;

import java.util.function.Supplier;

public class TestDemo {

        public static void main(String[] args) {
                Supplier<String> sup = "hello"::toUpperCase ;
                System.out.println(sup.get());
        }
}
```

6.3 消费型接口

```
package www.bit.java.testdemo;

import java.util.function.Consumer;

public class TestDemo {
        public static void main(String[] args) {
                Consumer<String> cons = System.out :: println ;
                cons.accept("嘿嘿嘿");
        }
}
```

6.4 断言型接口

```
package www.bit.java.testdemo;

import java.util.function.Predicate;

public class TestDemo {
        public static void main(String[] args) {
                Predicate<String> pre = "##123shdbs" :: startsWith ;
                System.out.println(pre.test("##"));
        }
}
```

如果要进行复杂的Lamdba运算，就需要利用这类函数式接口进行运算。