数据类型

基本数据类型

```
byte/8
char/16
short/16
int/32
float/32
long/64
double/64
boolean/~
```

boolean 只有两个值：true、false，可以使用 1 bit 来存储，但是具体大小没有明确规定。JVM 会在编译时期将 boolean 类型的数据转换为 int(也即大小32bit)，使用 1 来表示 true，0 表示 false。JVM 支持 boolean 数组，但是是通过读写 byte 数组来实现的。

包装类型

```
Integer x = 2;     // 装箱
   int y = x;         // 拆箱
```

基本类型都有对应的包装类型，基本类型与其对应的包装类型之间的赋值使用自动装箱 与拆箱完成。 在装箱的时候自动调用包装类型的装箱方法，在拆箱时自动调用包装类型的拆箱方 法，如Integer，装箱时调用valueOf(int),拆箱时调用intValue()方法。

在通过valueOf方法创建Integer对象的时候，如果数值在[-128,127]之间，便返回指向IntegerCache.cache中已经存在的对象的引用；否则创建一个新的Integer对象。

```
public static Integer valueOf(int i) {
        if(i >= -128 && i <= IntegerCache.high)
            return IntegerCache.cache[i + 128];
        else
            return new Integer(i);
    }

private static class IntegerCache {
        static final int high;
        static final Integer cache[];

        static {
            final int low = -128;

            // high value may be configured by property
            int h = 127;
            if (integerCacheHighPropValue != null) {
                // Use Long.decode here to avoid invoking methods that
                // require Integer's autoboxing cache to be initialized
                int i = Long.decode(integerCacheHighPropValue).intValue();
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - -low);
            }
            high = h;

            cache = new Integer[(high - low) + 1];
            int j = low;
            for(int k = 0; k < cache.length; k++)
                cache[k] = new Integer(j++);
        }

        private IntegerCache() {}
    }


public class Main {
    public static void main(String[] args) {

        Integer i1 = 100;
        Integer i2 = 100;
        Integer i3 = 200;
        Integer i4 = 200;

        System.out.println(i1==i2);
        // 输出true
        System.out.println(i3==i4);
        // 输出false
    }
}


    public static Double valueOf(double d) {
        return new Double(d);
```

```
    }


public class Main {
    public static void main(String[] args) {

        Double i1 = 100.0;
        Double i2 = 100.0;
        Double i3 = 200.0;
        Double i4 = 200.0;

        System.out.println(i1==i2);
        // 输出false
        System.out.println(i3==i4);
        // 输出false
    }
}
```

Integer、Short、Byte、Character、Long这几个类的valueOf方法的实现是类似的。Double、Float的valueOf方法的实现是类似的。因为在某个范围内的整型数值的个数是有限的，而浮点数却不是。

Integer i = new Integer(10)和Integer i =10;这两种方式的区别。

- 第一种方式不会触发自动装箱的过程；而第二种方式会触发；
- 在执行效率和资源占用上的区别。第二种方式的执行效率和资源占用在一般性情况下要优于第一种情况（注意这并不是绝对的）。

当 "=="运算符的两个操作数都是 包装器类型的引用，则是比较指向的是否是同一个对象，而如果其中有一个操作数是表达式（即包含算术运算）则比较的是数值（即会触发自动拆箱的过程）。另外，对于包装器类型，equals方法并不会进行类型转换。

```
public class Main {
    public static void main(String[] args) {

        Integer a = 1;
        Integer b = 2;
        Integer c = 3;
        Integer d = 3;
        Integer e = 321;
        Integer f = 321;
        Long g = 3L;
        Long h = 2L;

        System.out.println(c==d);
        System.out.println(e==f);
        System.out.println(c==(a+b));
        System.out.println(c.equals(a+b));
        System.out.println(g==(a+b));
        System.out.println(g.equals(a+b));
        System.out.println(g.equals(a+h));
        // true
        // false
        // true
        // true
        // true
        // false
        // true
    }
}
```

缓存池

new Integer(123) 与 Integer.valueOf(123) 的区别在于：

- new Integer(123) 每次都会新建一个对象；
- Integer.valueOf(123) 会使用缓存池中的对象，多次调用会取得同一个对象的引用。

```
Integer x = new Integer(123);
Integer y = new Integer(123);
System.out.println(x == y);    // false
Integer z = Integer.valueOf(123);
Integer k = Integer.valueOf(123);
System.out.println(z == k);   // true
```

valueOf() 方法的实现比较简单，就是先判断值是否在缓存池中，如果在的话就直接返回缓存池的内容。

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

在 Java 8 中，Integer 缓存池的大小默认为 -128~127。

```
static final int low = -128;
static final int high;
```

```
static final Integer cache[];

static {
    // high value may be configured by property
    int h = 127;
    String integerCacheHighPropValue =
        sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
    if (integerCacheHighPropValue != null) {
        try {
            int i = parseInt(integerCacheHighPropValue);
            i = Math.max(i, 127);
            // Maximum array size is Integer.MAX_VALUE
            h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
        } catch( NumberFormatException nfe) {
            // If the property cannot be parsed into an int, ignore it.
        }
    }
    high = h;

    cache = new Integer[(high - low) + 1];
    int j = low;
    for(int k = 0; k < cache.length; k++)
        cache[k] = new Integer(j++);

    // range [-128, 127] must be interned (JLS7 5.1.7)
    assert IntegerCache.high >= 127;
}
```

编译器会在自动装箱过程调用 valueOf() 方法，因此多个值相同且值在缓存池范围内的 Integer 实例使用自动装箱来创建，那么就会引用相同的对象。

```
Integer m = 123;
Integer n = 123;
System.out.println(m == n); // true
```

基本类型对应的缓冲池如下：

- boolean values true and false
- all byte values
- short values between -128 and 127
- int values between -128 and 127
- char in the range \u0000 to \u007F

在使用这些基本类型对应的包装类型时，如果该数值范围在缓冲池范围内，就可以直接使用缓冲池中的对象。

在 jdk 1.8 所有的数值类缓冲池中，Integer 的缓冲池 IntegerCache 很特殊，这个缓冲池的下界是 - 128，上界默认是 127，但是这个上界是可调的，在启动 jvm 的时候，通过 -XX:AutoBoxCacheMax= 来指定这个缓冲池的大小，该选项在 JVM 初始化的时候会设定一个名为 java.lang.IntegerCache.high 系统属性，然后 IntegerCache 初始化的时候就会读取该系统属性来决定上界。

String

概览

String 被声明为 final，因此它不可被继承。在 Java 8 中，String 内部使用 char 数组存储数据。

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
}
```

在 Java 9 之后，String 类的实现改用 byte 数组存储字符串，同时使用 coder 来标识使用了哪种编码。

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final byte[] value;

    /** The identifier of the encoding used to encode the bytes in {@code value}. */
    private final byte coder;
}
```

value 数组被声明为 final，这意味着 value 数组初始化之后就不能再引用其它数组。并且 String 内部没有改变 value 数组的方法，因此可以保证 String 不可变。

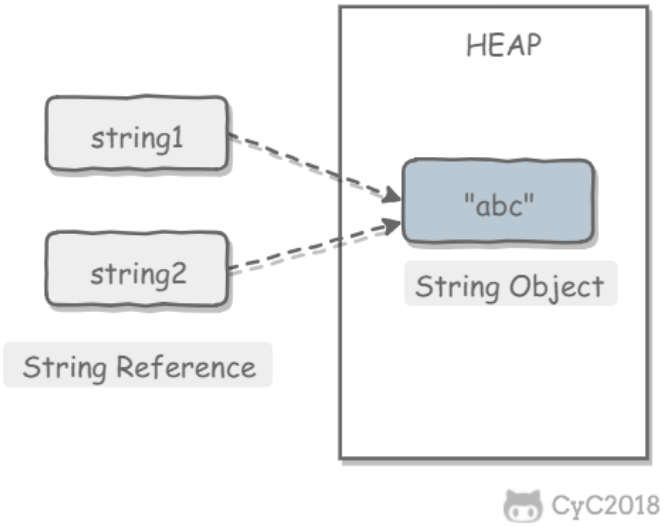
不可变的好处

1. 可以缓存 hash 值

   因为 String 的 hash 值经常被使用，例如 String 用做 HashMap 的 key。不可变的特性可以使得 hash 值也不可变，因此只需要进行一次计算。

2. String Pool 的需要

   如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用。只有 String 是不可变的，才可能使用 String Pool。



1. 安全性

   String 经常作为参数，String 不可变性可以保证参数不可变。例如在作为网络连接参数的情况下如果 String 是可变的，那么在网络连接过程中，String 被改变，改变 String 对象的那一方以为现在连接的是其它主机，而实际情况却不一定是。

2. 线程安全

   String 不可变性天生具备线程安全，可以在多个线程中安全地使用。

String, StringBuffer and StringBuilder

1. 可变性
   - String 不可变
   - StringBuffer 和 StringBuilder 可变
2. 线程安全
   - String 不可变，因此是线程安全的
   - StringBuilder 不是线程安全的
   - StringBuffer 是线程安全的，内部使用 synchronized 进行同步

String Pool

字符串常量池（String Pool）保存着所有字符串字面量（literal strings），这些字面量在编译时期就确定。不仅如此，还可以使用 String 的 intern() 方法在运行过程中将字符串添加到 String Pool 中。 当一个字符串调用 intern() 方法时，如果 String Pool 中已经存在一个字符串和该字符串值相等（使用 equals() 方法进行确定），那么就会返回 String Pool 中字符串的引用；否则，就会在 String Pool 中添加一个新的字符串，并返回这个新字符串的引用。

下面示例中，s1 和 s2 采用 new String() 的方式新建了两个不同字符串，而 s3 和 s4 是通过 s1.intern() 方法取得一个字符串引用。intern() 首先把 s1 引用的字符串放到 String Pool 中，然后返回这个字符串引用。因此 s3 和 s4 引用的是同一个字符串。

```
String s1 = new String("aaa");
String s2 = new String("aaa");
System.out.println(s1 == s2);           // false
String s3 = s1.intern();
String s4 = s1.intern();
System.out.println(s3 == s4);           // true
```

如果是采用 "bbb" 这种字面量的形式创建字符串，会自动地将字符串放入 String Pool 中。

```
String s5 = "bbb";
String s6 = "bbb";
System.out.println(s5 == s6);  // true
```

在 Java 7 之前，String Pool 被放在运行时常量池中，它属于永久代。而在 Java 7，String Pool 被移到堆中。这是因为永久代的空间有限，在大量使用字符串的场景下会导致 OutOfMemoryError 错误。

new String("abc")

使用这种方式一共会创建两个字符串对象（前提是 String Pool 中还没有 "abc" 字符串对象）。

- "abc" 属于字符串字面量，因此编译时期会在 String Pool 中创建一个字符串对象，指向这个 "abc" 字符串字面量；
- 而使用 new 的方式会在堆中创建一个字符串对象。

创建一个测试类，其 main 方法中使用这种方式来创建字符串对象。

```
public class NewStringTest {
    public static void main(String[] args) {
        String s = new String("abc");
    }
}
```

使用 javap -verbose 进行反编译，得到以下内容：

```
// ...
Constant pool:
// ...
   #2 = Class              #18            // java/lang/String
   #3 = String             #19            // abc
// ...
  #18 = Utf8               java/lang/String
  #19 = Utf8               abc
// ...

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
         0: new           #2                  // class java/lang/String
         3: dup
         4: ldc           #3                  // String abc
         6: invokespecial #4                  // Method java/lang/String."<init>":(Ljava/lang/String;)V
         9: astore_1
// ...
```

在 Constant Pool 中，#19 存储这字符串字面量 "abc"，#3 是 String Pool 的字符串对象，它指向 #19 这个字符串字面量。在 main 方法中，0: 行使用 new #2 在堆中创建一个字符串对象，并且使用 ldc #3 将 String Pool 中的字符串对象作为 String 构造函数的参数。

以下是 String 构造函数的源码，可以看到，在将一个字符串对象作为另一个字符串对象的构造函数参数时，并不会完全复制 value 数组内容，而是都会指向同一个 value 数组。

```
public String(String original) {
    this.value = original.value;
    this.hash = original.hash;
}
```

运算

参数传递

Java 的参数是以值传递的形式传入方法中，而不是引用传递。以下代码中 Dog dog 的 dog 是一个指针，存储的是对象的地址。在将一个参数传入一个方法时，本质上是将对象的地址以值的方式传递到形参中。因此在方法中使指针引用其它对象，那么这两个指针此时指向的是完全不同的对象，在一方改变其所指向对象的内容时对另一方没有影响。

```
public class Dog {

    String name;

    Dog(String name) {
        this.name = name;
    }

    String getName() {
        return this.name;
    }

    void setName(String name) {
        this.name = name;
    }

    String getObjectAddress() {
        return super.toString();
    }
}


public class PassByValueExample {
    public static void main(String[] args) {
        Dog dog = new Dog("A");
        System.out.println(dog.getObjectAddress()); // Dog@4554617c
```

```
        // 此处传递的时引用的值
        func(dog);
        System.out.println(dog.getObjectAddress()); // Dog@4554617c
        System.out.println(dog.getName());          // A
    }

    private static void func(Dog dog) {
        System.out.println(dog.getObjectAddress()); // Dog@4554617c
        dog = new Dog("B");
        System.out.println(dog.getObjectAddress()); // Dog@74a14482
        System.out.println(dog.getName());          // B
    }
}
```

如果在方法中改变对象的字段值会改变原对象该字段值，因为改变的是同一个地址指向的内容。

```
class PassByValueExample {
    public static void main(String[] args) {
        Dog dog = new Dog("A");
        func(dog);
        System.out.println(dog.getName());          // B
    }

    private static void func(Dog dog) {
        dog.setName("B");
    }
}
```

Stackoverflow问答:Java是传值还是传引用?

float 与 double

Java 不能隐式执行向下转型，因为这会使得精度降低。

1.1 字面量属于 double 类型，不能直接将 1.1 直接赋值给 float 变量，因为这是向下转型。

```
// float f = 1.1;
// 1.1f 字面量才是 float 类型。
float f = 1.1f;
```

隐式类型转换

因为字面量 1 是 int 类型，它比 short 类型精度要高，因此不能隐式地将 int 类型下转型为 short 类型。

```
short s1 = 1;
// s1 = s1 + 1;
```

但是使用 += 或者 ++ 运算符可以执行隐式类型转换。

```
s1 += 1;
// s1++;
```

上面的语句相当于将 s1 + 1 的计算结果进行了向下转型：

```
s1 = (short) (s1 + 1);
```

StackOverflow : Why don't Java's +=, -=, *=, /= compound assignment operators require casting?

switch

从 Java 7 开始，可以在 switch 条件判断语句中使用 String 对象。

```
String s = "a";
switch (s) {
    case "a":
        System.out.println("aaa");
        break;
    case "b":
        System.out.println("bbb");
        break;
}
```

switch 不支持 long，是因为 switch 的设计初衷是对那些只有少数的几个值进行等值判断，如果值过于复杂，那么还是用 if 比较合适。

```
// long x = 111;
// switch (x) { // Incompatible types. Found: 'long', required: 'char, byte, short, int, Character, Byte, Short, Integer, String, or an enum'
//     case 111:
//         System.out.println(111);
//         break;
//     case 222:
//         System.out.println(222);
//         break;
// }
```

StackOverflow : Why can't your switch statement data type be long, Java?

继承

访问权限

Java 中有三个访问权限修饰符：private、protected 以及 public，如果不加访问修饰符，表示包级可见。

可以对类或类中的成员（字段以及方法）加上访问修饰符。

- 类可见表示其它类可以用这个类创建实例对象(最外围的类默认为public，不能显示申明为其他类型)
- 成员可见表示其它类可以用这个类的实例对象访问到该成员

protected 用于修饰成员，表示在继承体系中成员对于子类可见，但是这个访问修饰符对于类没有意义。

设计良好的模块会隐藏所有的实现细节，把它的 API 与它的实现清晰地隔离开来。模块之间只通过它们的 API 进行通信，一个模块不需要知道其他模块的内部工作情况，这个概念被称为信息隐藏或封装。因此访问权限应当尽可能地使每个类或者成员不被外界访问。

如果子类的方法重写了父类的方法，那么子类中该方法的访问级别不允许低于父类的访问级别。这是为了确保可以使用父类实例的地方都可以使用子类实例，也就是确保满足里氏替换原则。

字段决不能是公有的，因为这么做的话就失去了对这个字段修改行为的控制，客户端可以对其随意修改。例如下面的例子中，AccessExample 拥有 id 公有字段，如果在某个时刻，我们想要使用 int 存储 id 字段，那么就需要修改所有的客户端代码。

```
public class AccessExample {
    public String id;
}
```

可以使用公有的 getter 和 setter 方法来替换公有字段，这样的话就可以控制对字段的修改行为。

```
public class AccessExample {

    private int id;

    public String getId() {
        return id + "";
    }

    public void setId(String id) {
        this.id = Integer.valueOf(id);
    }
}
```

但是也有例外，如果是包级私有的类或者私有的嵌套类，那么直接暴露成员不会有特别大的影响。

```
public class AccessWithInnerClassExample {

    private class InnerClass {
        int x;
    }

    private InnerClass innerClass;

    public AccessWithInnerClassExample() {
        innerClass = new InnerClass();
    }

    public int getValue() {
        return innerClass.x;  // 直接访问
    }
}
```

抽象类与接口

抽象类

抽象类和抽象方法都使用 abstract 关键字进行声明。如果一个类中包含抽象方法，那么这个类必须声明为抽象类。而抽象类中可以有非抽象方法

抽象类和普通类最大的区别是，抽象类不能被实例化，需要继承抽象类才能实例化其子类。

```
public abstract class AbstractClassExample {

    protected int x;
    private int y;

    public abstract void func1();

    public void func2() {
        System.out.println("func2");
    }
}


public class AbstractExtendClassExample extends AbstractClassExample {
    @Override
    public void func1() {
        System.out.println("func1");
    }
}
```

接口

接口是抽象类的延伸，在 Java 8 之前，它可以看成是一个完全抽象的类，也就是说它不能有任何的方法实现。

从 Java 8 开始，接口也可以拥有默认的方法实现，这是因为不支持默认方法的接口的维护成本太高了。在 Java 8 之前，如果一个接口想要添加新的方法，那么要修改所有实现了该接口的类。

- 接口的成员（字段 + 方法）默认都是 public 的，并且不允许定义为 private 或者 protected
- 接口的字段默认都是 static 和 final 的
- default函数，实现类可以不实现这个方法
- static函数，这允许API设计者在接口中定义像getInstance一样的静态工具方法，这样就能够使得API简洁而精练，子类可以不实现

```
public interface InterfaceExample {

    void func1();

    default void func2(){
        System.out.println("func2");
    }

    static void func3(){
        System.out.println("func2");
    }

    int x = 123;
    // int y;              // Variable 'y' might not have been initialized
    public int z = 0;      // Modifier 'public' is redundant for interface fields
    // private int k = 0;   // Modifier 'private' not allowed here
    // protected int l = 0; // Modifier 'protected' not allowed here
    // private void fun3(); // Modifier 'private' not allowed here
}
```

比较

- 从设计层面上看，抽象类提供了一种 IS-A 关系，那么就必须满足里式替换原则，即子类对象必须能够替换掉所有父类对象。而接口更像是一种 LIKE-A 关系，它只是提供一种方法实现契约，并不要求接口和实现接口的类具有 IS-A 关系。
- 从使用上来看，一个类可以实现多个接口，但是不能继承多个抽象类。
- 接口的字段只能是 static 和 final 类型的，而抽象类的字段没有这种限制。
- 接口的成员只能是 public 的，而抽象类的成员可以有多种访问权限。

使用选择

使用接口：

- 需要让不相关的类都实现一个方法，例如不相关的类都可以实现 Compareable 接口中的 compareTo() 方法
- 需要使用多重继承

使用抽象类

- 需要能控制继承来的成员的访问权限，而不是都为 public
- 需要继承非静态和非常量字段

在很多情况下，接口优先于抽象类。因为接口没有抽象类严格的类层次结构要求，可以灵活地为一个类添加行为。并且从 Java 8 开始，接口也可以有默认的方法实现，使得修改接口的成本也变的很低。

super

- 访问父类的构造函数：可以使用 super() 函数访问父类的构造函数，从而委托父类完成一些初始化的工作
- 访问父类的成员：如果子类重写了父类的某个方法，可以通过使用 super 关键字来引用父类的方法实现

```
public class SuperExample {

    protected int x;
    protected int y;

    public SuperExample(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void func() {
        System.out.println("SuperExample.func()");
    }
}


public class SuperExtendExample extends SuperExample {

    private int z;

    public SuperExtendExample(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }

    @Override
    public void func() {
        super.func();
        System.out.println("SuperExtendExample.func()");
    }
}


SuperExample e = new SuperExtendExample(1, 2, 3);
e.func();


SuperExample.func()
SuperExtendExample.func()
```

重写与重载

重写（Override）

存在于继承体系中，指子类实现了一个与父类在方法声明上完全相同的一个方法。

为了满足里式替换原则，重写有以下三个限制：

- 子类方法的访问权限必须大于等于父类方法
- 子类方法的返回类型必须是父类方法返回类型或为其子类型
- 子类方法抛出的异常类型必须是父类抛出异常类型或为其子类型

使用 @Override 注解，可以让编译器帮忙检查是否满足上面的三个限制条件。

下面的示例中，SubClass 为 SuperClass 的子类，SubClass 重写了 SuperClass 的 func() 方法。其中：

- 子类方法访问权限为 public，大于父类的 protected
- 子类的返回类型为 ArrayList，是父类返回类型 List 的子类
- 子类抛出的异常类型为 Exception，是父类抛出异常 Throwable 的子类
- 子类重写方法使用 @Override 注解，从而让编译器自动检查是否满足限制条件

```
class SuperClass {
    protected List<Integer> func() throws Throwable {
        return new ArrayList<>();
    }
}

class SubClass extends SuperClass {
    @Override
    public ArrayList<Integer> func() throws Exception {
        return new ArrayList<>();
    }
}
```

在调用一个方法时，先从本类中查找看是否有对应的方法，如果没有查找到再到父类中查看，看是否有继承来的方法。否则就要对参数进行转型，转成父类之后看是否有对应的方法。总的来说，方法调用的优先级为：

- this.func(this)
- super.func(this)
- this.func(super)
- super.func(super)

```
/*
    A
```

```
    |
    B
    |
    C
    |
    D
 */


class A {

    public void show(A obj) {
        System.out.println("A.show(A)");
    }

    public void show(C obj) {
        System.out.println("A.show(C)");
    }
}

class B extends A {

    @Override
    public void show(A obj) {
        System.out.println("B.show(A)");
    }
}

class C extends B {
}

class D extends C {
}


public static void main(String[] args) {

    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();

    // 在 A 中存在 show(A obj)，直接调用
    a.show(a); // A.show(A)
    // 在 A 中不存在 show(B obj)，将 B 转型成其父类 A
    a.show(b); // A.show(A)
    // 在 B 中存在从 A 继承来的 show(C obj)，直接调用
    b.show(c); // A.show(C)
    // 在 B 中不存在 show(D obj)，但是存在从 A 继承来的 show(C obj)，将 D 转型成其父类 C
    b.show(d); // A.show(C)

    // 引用的还是 B 对象，所以 ba 和 b 的调用结果一样
    A ba = new B();
    ba.show(c); // A.show(C)
    ba.show(d); // A.show(C)
}
```

重载（Overload）

存在于同一个类中，指一个方法与已经存在的方法名称上相同，但是参数类型、个数、顺序至少有一个不同。

应该注意的是，返回值不同，其它都相同不算是重载。

Object通用方法

equals()

1. 等价关系

   - 自反性

     ```
     x.equals(x); // true
     ```

   - 对称性

     ```
     x.equals(y) == y.equals(x); // true
     ```

   - 传递性

     ```
     if (x.equals(y) && y.equals(z))
     x.equals(z); // true;
     ```

   - 一致性

     ```
     // 多次调用 equals() 方法结果不变
     x.equals(y) == x.equals(y); // true
     ```

   - 与 null 的比较

```
// 对任何不是 null 的对象 x 调用 x.equals(null) 结果都为 false
x.equals(null); // false;
```

2. 等价与相等

- 对于基本类型，== 判断两个值是否相等，基本类型没有 equals() 方法
- 对于引用类型，== 判断两个变量是否引用同一个对象，而 equals() 判断引用的对象是否等价。

```
Integer x = new Integer(1);
Integer y = new Integer(1);
System.out.println(x.equals(y)); // true
System.out.println(x == y);      // false
```

1. 实现

- 检查是否为同一个对象的引用，如果是直接返回 true；
- 检查是否是同一个类型，如果不是，直接返回 false；
- 将 Object 对象进行转型；
- 判断每个关键域是否相等。

```
// jdk实现
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

```
public class EqualExample {

    private int x;
    private int y;
    private int z;

    public EqualExample(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        EqualExample that = (EqualExample) o;

        if (x != that.x) return false;
        if (y != that.y) return false;
        return z == that.z;
    }
}
```

hashCode()

hashCode() 返回散列值，而 equals() 是用来判断两个对象是否等价。等价的两个对象散列值一定相同，但是散列值相同的两个对象不一定等价。

在覆盖 equals() 方法时应当总是覆盖 hashCode() 方法，保证等价的两个对象散列值也相等。

下面的代码中，新建了两个等价的对象，并将它们添加到 HashSet 中。我们希望将这两个对象当成一样的，只在集合中添加一个对象，但是因为 EqualExample 没有实现 hashCode() 方法，因此这两个对象的散列值是不同的，最终导致集合添加了两个等价的对象。

```
EqualExample e1 = new EqualExample(1, 1, 1);
EqualExample e2 = new EqualExample(1, 1, 1);
System.out.println(e1.equals(e2)); // true
HashSet<EqualExample> set = new HashSet<>();
set.add(e1);
set.add(e2);
System.out.println(set.size());   // 2
```

理想的散列函数应当具有均匀性，即不相等的对象应当均匀分布到所有可能的散列值上。这就要求了散列函数要把所有域的值都考虑进来。可以将每个域都当成 R 进制的某一位，然后组成一个 R 进制的整数。R 一般取 31，因为它是一个奇素数，如果是偶数的话，当出现乘法溢出，信息就会丢失，因为与 2 相乘相当于向左移一位。

一个数与 31 相乘可以转换成移位和减法：31*x == (x<<5)-x，编译器会自动进行这个优化。

```
    // jdk实现
public int hashCode() {
     // Default to 0
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;

            for (int i = 0; i < value.length; i++) {
                h = 31 * h + val[i];
            }
            hash = h;
        }
        return h;
    }
```

```
@Override
public int hashCode() {
    int result = 17;
    result = 31 * result + x;
    result = 31 * result + y;
    result = 31 * result + z;
    return result;
}
```

toString()

默认返回 ToStringExample@4554617c 这种形式，其中 @ 后面的数值为散列码的无符号十六进制表示。

```
public class ToStringExample {

    private int number;

    public ToStringExample(int number) {
        this.number = number;
    }
}
```

```
ToStringExample example = new ToStringExample(123);
System.out.println(example.toString());
// ToStringExample@4554617c
```

```
// jdk实现
 public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
```

clone()

1. cloneable

clone() 是 Object 的 protected 方法，它不是 public，一个类不显式去重写 clone()，其它类就不能直接去调用该类实例的 clone() 方法。

```
public class CloneExample {
    private int a;
    private int b;
}
```

```
CloneExample e1 = new CloneExample();
// CloneExample e2 = e1.clone(); // 'clone()' has protected access in 'java.lang.Object'
```

重写 clone() 得到以下实现：

```
public class CloneExample {
    private int a;
    private int b;

    @Override
    public CloneExample clone() throws CloneNotSupportedException {
        return (CloneExample)super.clone();
    }
}
```

```
CloneExample e1 = new CloneExample();
try {
    CloneExample e2 = e1.clone();
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
```

```
}
// java.lang.CloneNotSupportedException: CloneExample
```

以上抛出了 CloneNotSupportedException，这是因为 CloneExample 没有实现 Cloneable 接口。

应该注意的是，clone() 方法并不是 Cloneable 接口的方法，而是 Object 的一个 protected 方法。Cloneable 接口只是规定，如果一个类没有实现 Cloneable 接口又调用了 clone() 方法，就会抛出 CloneNotSupportedException。

```
public class CloneExample implements Cloneable {
    private int a;
    private int b;

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

1. 拷贝对象和原始对象的引用类型引用同一个对象。

```
public class ShallowCloneExample implements Cloneable {

    private int[] arr;

    public ShallowCloneExample() {
        arr = new int[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
    }

    public void set(int index, int value) {
        arr[index] = value;
    }

    public int get(int index) {
        return arr[index];
    }

    @Override
    protected ShallowCloneExample clone() throws CloneNotSupportedException {
        return (ShallowCloneExample) super.clone();
    }
}
```

```
ShallowCloneExample e1 = new ShallowCloneExample();
ShallowCloneExample e2 = null;
try {
    e2 = e1.clone();
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
e1.set(2, 222);
System.out.println(e2.get(2)); // 222
```

1. 深拷贝

拷贝对象和原始对象的引用类型引用不同对象。

```
public class DeepCloneExample implements Cloneable {

    private int[] arr;

    public DeepCloneExample() {
        arr = new int[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
    }

    public void set(int index, int value) {
        arr[index] = value;
    }

    public int get(int index) {
        return arr[index];
    }

    @Override
    protected DeepCloneExample clone() throws CloneNotSupportedException {
        DeepCloneExample result = (DeepCloneExample) super.clone();
        result.arr = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
            result.arr[i] = arr[i];
        }
        return result;
    }
}
```

```
DeepCloneExample e1 = new DeepCloneExample();
DeepCloneExample e2 = null;
try {
    e2 = e1.clone();
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
e1.set(2, 222);
System.out.println(e2.get(2)); // 2
```

1. clone() 的替代方案

使用 clone() 方法来拷贝一个对象即复杂又有风险，它会抛出异常，并且还需要类型转换。Effective Java 书上讲到，最好不要去使用 clone()，可以使用拷贝构造函数或者拷贝工厂来拷贝一个对象。其他方法有通过序列化对象到流中再反序列化或者先转json串在解析成对象

```
public class CloneConstructorExample {

    private int[] arr;

    public CloneConstructorExample() {
        arr = new int[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
    }

    public CloneConstructorExample(CloneConstructorExample original) {
        arr = new int[original.arr.length];
        for (int i = 0; i < original.arr.length; i++) {
            arr[i] = original.arr[i];
        }
    }

    public void set(int index, int value) {
        arr[index] = value;
    }

    public int get(int index) {
        return arr[index];
    }
}
```

```
CloneConstructorExample e1 = new CloneConstructorExample();
CloneConstructorExample e2 = new CloneConstructorExample(e1);
e1.set(2, 222);
System.out.println(e2.get(2)); // 2
```

关键字

final

1. 数据

声明数据为常量，可以是编译时常量，也可以是在运行时被初始化后不能被改变的常量

- 对于基本类型，final 使数值不变
- 对于引用类型，final 使引用不变，也就不能引用其它对象，但是被引用的对象本身是可以修改的

```
final int x = 1;
// x = 2;  // cannot assign value to final variable 'x'
final A y = new A();
y.a = 1;
```

1. 方法

声明方法不能被子类重写

private 方法隐式地被指定为 final，如果在子类中定义的方法和基类中的一个 private 方法签名相同，此时子类的方法不是重写基类方法，而是在子类中定义了一个新的方法。

1. 类

声明类不允许被继承

static

1. 静态变量

- 静态变量：又称为类变量，也就是说这个变量属于类的，类所有的实例都共享静态变量，可以直接通过类名来访问它。静态变量在内存中只存在一份

- 实例变量：每创建一个实例就会产生一个实例变量，它与该实例同生共死。

```
public class A {

    private int x;         // 实例变量
    private static int y;  // 静态变量

    public static void main(String[] args) {
        // int x = A.x;  // Non-static field 'x' cannot be referenced from a static context
        A a = new A();
        int x = a.x;
        int y = A.y;
    }
}
```

1. 静态方法

静态方法在类加载的时候就存在了，它不依赖于任何实例。所以静态方法必须有实现，也就是说它不能是抽象方法。

```
public abstract class A {
    public static void func1(){
    }
    // public abstract static void func2();  // Illegal combination of modifiers: 'abstract' and 'static'
}
```

只能访问所属类的静态字段和静态方法，方法中不能有 this 和 super 关键字。

```
public class A {

    private static int x;
    private int y;

    public static void func1(){
        int a = x;
        // int b = y;  // Non-static field 'y' cannot be referenced from a static context
        // int b = this.y;     // 'A.this' cannot be referenced from a static context
    }
}
```

1. 静态语句块

静态语句块在类初始化时运行一次

```
public class A {
    static {
        System.out.println("123");
    }

    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
    }
}
// 123
```

1. 静态内部类

非静态内部类依赖于外部类的实例，而静态内部类不需要。

```
public class OuterClass {

    class InnerClass {
    }

    static class StaticInnerClass {
    }

    public static void main(String[] args) {
        // InnerClass innerClass = new InnerClass(); // 'OuterClass.this' cannot be referenced from a static context
        OuterClass outerClass = new OuterClass();
        InnerClass innerClass = outerClass.new InnerClass();
        StaticInnerClass staticInnerClass = new StaticInnerClass();
    }
}
```

静态内部类不能访问外部类的非静态的变量和方法。

1. 静态导包

在使用静态变量和方法时不用再指明 ClassName，从而简化代码，但可读性大大降低。

```
import static com.xxx.ClassName.*
```

1. 初始化顺序

静态变量和静态语句块优先于实例变量和普通语句块，静态变量和静态语句块的初始化顺序取决于它们在代码中的顺序。

```
public static String staticField = "静态变量";

static {
    System.out.println("静态语句块");
}

public String field = "实例变量";

{
    System.out.println("普通语句块");
}
```

最后才是构造函数的初始化。

```
public InitialOrderTest() {
    System.out.println("构造函数");
}
```

存在继承的情况下，初始化顺序为：

- 父类（静态变量、静态语句块）
- 子类（静态变量、静态语句块）
- 父类（实例变量、普通语句块）
- 父类（构造函数）
- 子类（实例变量、普通语句块）
- 子类（构造函数）

反射

每个类都有一个 Class 对象，包含了与类有关的信息。当编译一个新类时，会产生一个同名的 .class 文件，该文件内容保存着 Class 对象。

类加载相当于 Class 对象的加载，类在第一次使用时才动态加载到 JVM 中。也可以使用 Class.forName("com.mysql.jdbc.Driver") 这种方式来控制类的加载，该方法会返回一个 Class 对象。

获得Class对象的三种方法

- Class.forName(driver);
- obj.class;
- obj.getClass;

反射可以提供运行时的类信息，并且这个类可以在运行时才加载进来，甚至在编译时期该类的 .class 不存在也可以加载进来。

Class 和 java.lang.reflect 一起对反射提供了支持，java.lang.reflect 类库主要包含了以下三个类：

- Field ： 可以使用 get() 和 set() 方法读取和修改 Field 对象关联的字段
- Method ： 可以使用 invoke() 方法调用与 Method 对象关联的方法
- Constructor ： 可以用 Constructor 创建新的对象

反射的优点：

- 可扩展性 ： 应用程序可以利用全限定名创建可扩展对象的实例，来使用来自外部的用户自定义类。
- 类浏览器和可视化开发环境 ： 一个类浏览器需要可以枚举类的成员。可视化开发环境（如 IDE）可以从利用反射中可用的类型信息中受益，以帮助程序员编写正确的代码。
- 调试器和测试工具 ： 调试器需要能够检查一个类里的私有成员。测试工具可以利用反射来自动地调用类里定义的可被发现的 API 定义，以确保一组测试中有较高的代码覆盖率。

反射的缺点：

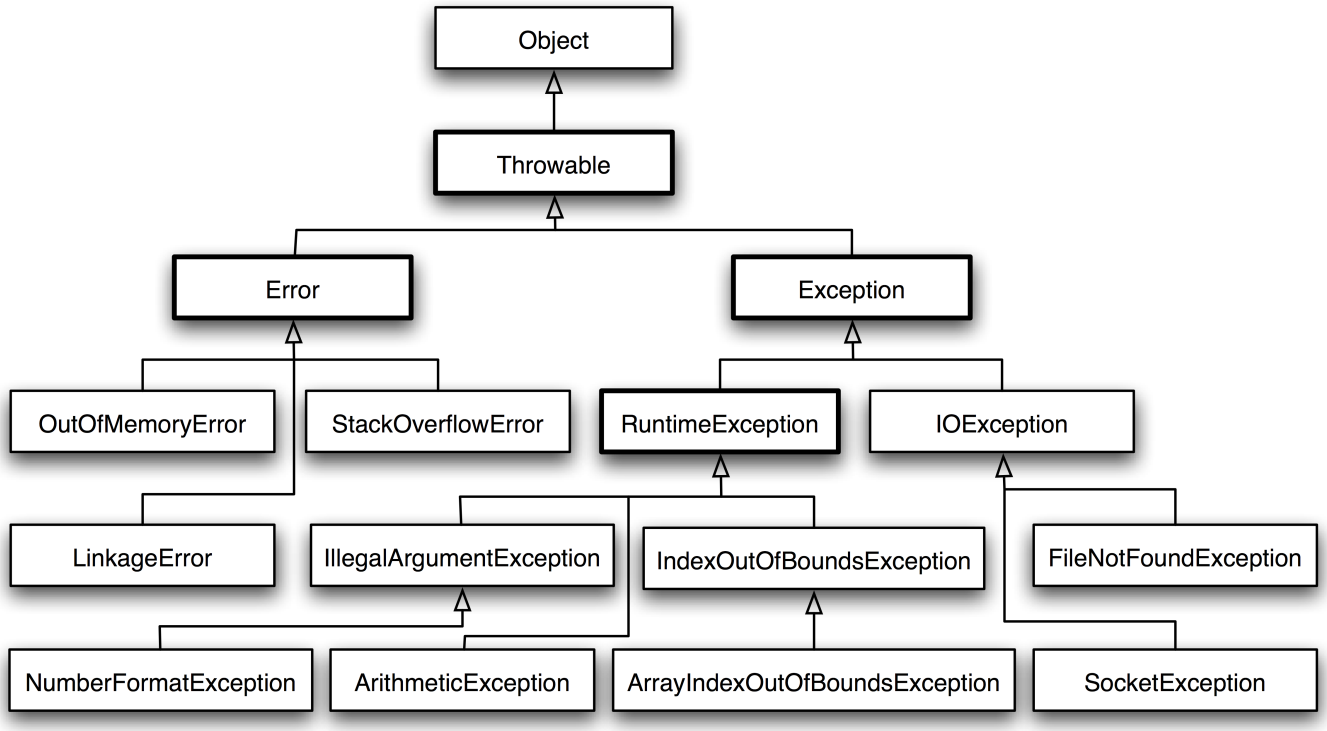
尽管反射非常强大，但也不能滥用。如果一个功能可以不用反射完成，那么最好就不用。在我们使用反射技术时，下面几条内容应该牢记于心。

- 性能开销 ： 反射涉及了动态类型的解析，所以 JVM 无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多。我们应该避免在经常被执行的代码或对性能要求很高的程序中使用反射。
- 安全限制 ： 使用反射技术要求程序必须在一个没有安全限制的环境中运行。如果一个程序必须在有安全限制的环境中运行，如 Applet，那么这就是个问题了。
- 内部暴露 ： 由于反射允许代码执行一些在正常情况下不被允许的操作（比如访问私有的属性和方法），所以使用反射可能会导致意料之外的副作用，这可能导致代码功能失调并破坏可移植性。反射代码破坏了抽象性，因此当平台发生改变的时候，代码的行为就有可能也随着变化。

异常

Throwable 可以用来表示任何可以作为异常抛出的类，分为两种： Error 和 Exception。其中 Error 用来表示 JVM 无法处理的错误，Exception 分为两种：

- 受检异常： 需要用 try...catch... 语句捕获并进行处理，并且可以从异常中恢复；
- 非受检异常： 是程序运行时错误，例如除 0 会引发 Arithmetic Exception，此时程序崩溃并且无法恢复。



自定义异常

使用Java内置的异常类可以描述在编程时出现的大部分异常情况。除此之外，用户还可以自定义异常。用户自定义异常类，只需继承Exception类即可。

在程序中使用自定义异常类，大体可分为以下几个步骤:

- 创建自定义异常类。
- 在方法中通过throw关键字抛出异常对象。
- 如果在当前抛出异常的方法中处理异常，可以使用try-catch语句捕获并处理；否则在方法的声明处通过throws关键字指明要抛出给方法调用者的异常，继续进行下一步操作。
- 在出现异常方法的调用者中捕获并处理异常。

```
class MyException extends Exception {
    private int detail;
    MyException(int a){
        detail = a;
    }
    public String toString(){
        return "MyException ["+ detail + "]";
    }
}
public class TestMyException{
    static void compute(int a) throws MyException{
        System.out.println("Called compute(" + a + ")");
        if(a > 10){
            throw new MyException(a);
        }
        System.out.println("Normal exit!");
    }
    public static void main(String [] args){
        try{
            compute(1);
            compute(20);
        }catch(MyException me){
            System.out.println("Caught " + me);
        }
    }
}
```

泛型

概述

什么是泛型？为什么要使用泛型？

> 泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。 泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

作用示例

```
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        List arrayList = new ArrayList();
        arrayList.add("aaaa");
        arrayList.add(100);

        for (int i = 0; i < arrayList.size(); i++) {
            String item = (String) arrayList.get(i);
            System.out.println("泛型测试item = " + item);
        }
    }
}

// 泛型测试item = aaaa
// Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
//    at Main.main(Main.java:15)
```

ArrayList可以存放任意类型，例子中添加了一个String类型，添加了一个Integer类型，再使用时都以String的方式使用，因此程序崩溃了。为了解决类似这样的问题（在编译阶段就可以解决），泛型应运而生。

我们将第一行声明初始化list的代码更改一下，编译器会在编译阶段就能够帮我们发现类似这样的问题。

```
List<String> arrayList = new ArrayList<String>();
...
//arrayList.add(100); 在编译阶段，编译器就会报错
```

特性

泛型只在编译阶段有效。看下面的代码：

```
import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        List<String> stringArrayList = new ArrayList<>();
        List<Integer> integerArrayList = new ArrayList<>();

        Class classStringArrayList = stringArrayList.getClass();
        Class classIntegerArrayList = integerArrayList.getClass();

        if (classStringArrayList.equals(classIntegerArrayList)) {
            System.out.println("泛型测试类型相同");
        }
    }
}

// 泛型测试类型相同
```

通过上面的例子可以证明，在编译之后程序会采取去泛型化的措施。也就是说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出，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。也就是说，泛型信息不会进入到运行时阶段。

对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

泛型类

泛型类型用于类的定义中，被称为泛型类。通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Set、Map。

泛型类的最基本写法:

```
class 类名称 <泛型标识：可以随便写任意标识号，标识指定的泛型的类型>{
  private 泛型标识 /*（成员变量类型）*/ var;
  .....

  }
}
```

一个普通的泛型类

```
public class Generic<T>{

    private T key;

    public Generic(T key) {
        this.key = key;
    }

    public T getKey(){
        return key;
    }
}
```

```
Generic<Integer> genericInteger = new Generic<Integer>(123456);
```

```
Generic<String> genericString = new Generic<String>("key_vlaue");
System.out.println("泛型测试key is " + genericInteger.getKey());
        System.out.println("泛型测试key is " + genericString.getKey());

// 泛型测试: key is 123456
// 泛型测试: key is key_vlaue
```

定义的泛型类，就一定要传入泛型类型实参么？并不是这样，在使用泛型的时候如果传入泛型实参，则会根据传入的泛型实参做相应的限制，此时泛型才会起到本应起到的限制作用。如果不传入泛型类型实参的话，在泛型类中使用泛型的方法或成员变量定义的类型可以为任何的类型。

注意：

- 泛型的类型参数只能是类类型，不能是简单类型。
- 不能对确切的泛型类型使用instanceof操作。如下面的操作是非法的，编译时会出错。

```
// if(ex_num instanceof Generic<Number>){   }
// if(ex_num instanceof Generic){   }
// 都无法通过编译
```

泛型接口

泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中，可以看一个例子：

```
public interface Generator<T> {
    public T next();
}
```

当实现泛型接口的类，未传入泛型实参时：

```
/**
 * 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
 * 即：class FruitGenerator<T> implements Generator<T>
 * 如果不声明泛型，如：class FruitGenerator implements Generator<T>，编译器会报错："Unknown class"
 */
class FruitGenerator<T> implements Generator<T>{
    @Override
    public T next() {
        return null;
    }
}
```

当实现泛型接口的类，传入泛型实参时：

```
/**
 * 传入泛型实参时：
 * 定义一个生产器实现这个接口,虽然我们只创建了一个泛型接口Generator<T>
 * 但是我们可以为T传入无数个实参，形成无数种类型的Generator接口。
 * 在实现类实现泛型接口时，如已将泛型类型传入实参类型，则所有使用泛型的地方都要替换成传入的实参类型
 * 即：Generator<T>，public T next();中的的T都要替换成传入的String类型。
 */
public class FruitGenerator 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

我们知道Ingeter是Number的一个子类，同时Generic与Generic实际上是相同的一种基本类型。那么问题来了，在使用Generic作为形参的方法中，能否使用Generic的实例传入呢？在逻辑上类似于Generic和Generic是否可以看成具有父子关系的泛型类型呢？

```
public static void showKeyValue(Generic<Number> obj) {
        System.out.println("泛型测试key value is " + obj.getKey());
    }


Generic<Integer> gInteger = new Generic<Integer>(123);
        Generic<Number> gNumber = new Generic<Number>(456);

        showKeyValue(gNumber);

        // showKeyValue(gInteger);
        // 无法通过编译
```

将方法改成：

```
 public static void showKeyValue(Generic<?> obj) {
        System.out.println("泛型测试key value is " + obj.getKey());
    }
```

类型通配符一般是使用？代替具体的类型实参，注意了，此处'？'是类型实参，而不是类型形参 。再直白点的意思就是，此处的？和Number、String、Integer一样都是一种实际的类型，可以把？看成所有类型的父类。是一种真实的类型。

可以解决当具体类型不确定的时候，这个通配符就是？；当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用？通配符来表未知类型。

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 。

```
/**
 * 泛型方法的基本介绍
 * @param tClass 传入的泛型实参
 * @return T 返回值为T类型
 * 说明：
 *     1）public 与 返回值中间<T>非常重要，可以理解为声明此方法为泛型方法。
 *     2）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。
 *     3）<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。
 *     4）与泛型类的定义一样，此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型。
 */
public <T> T genericMethod(Class<T> tClass)throws InstantiationException ,
  IllegalAccessException{
        T instance = tClass.newInstance();
        return instance;
}


public class GenericTest {

    public class Generic<T>{
        private T key;

        public Generic(T key) {
            this.key = key;
        }



        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());

        T test = container.getKey();
        return test;
```

```
    }

    public void showKeyValue1(Generic<Number> obj){
        System.out.println("泛型测试key value is " + obj.getKey());
    }



    public void showKeyValue2(Generic<?> obj){
        System.out.println("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) {


    }
}
```

泛型讲解

注解

概述

实际上Java注解与普通修饰符(public、static、void等)的使用方式并没有多大区别。

```
public class AnnotationDemo {
    //@Test注解修饰方法A
    @Test
    public static void A(){
        System.out.println("Test.....");
    }

    //一个方法上可以拥有多个不同的注解
    @Deprecated
    @SuppressWarnings("uncheck")
    public static void B(){

    }
}
```

通过在方法上使用@Test注解后，在运行该方法时，测试框架会自动识别该方法并单独调用，@Test实际上是一种标记注解，起标记作用，运行时告诉测试框架该方法为测试方法。而对于@Deprecated和@SuppressWarnings(“uncheck”)，则是Java本身内置的注解，在代码中，可以经常看见它们，但这并不是一件好事，毕竟当方法或是类上面有@Deprecated注解时，说明该方法或是类都已经过期不建议再用，@SuppressWarnings 则表示忽略指定警告，比如@SuppressWarnings(“uncheck”)，这就是注解的最简单的使用方式，那么下面我们就来看看注解定义的基本语法

基本语法

声明注解和元注解

上例中注解的声明

```
//声明Test注解
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {

}
```

我们使用了@interface声明了Test注解，并使用@Target注解传入ElementType.METHOD参数来标明@Test只能用于方法上，@Retention(RetentionPolicy.RUNTIME)则用来表示该注解生存期是运行时，从代码上看注解的定义很像接口的定义，确实如此，毕竟在编译后也会生成Test.class文件。对于@Target和@Retention是由Java提供的元注解，所谓元注解就是标记其他注解的注解

- @Target 用来约束注解可以应用的地方（如方法、类或字段），其中ElementType是枚举类型，其定义如下，也代表可能的取值范围

```
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
}
```

请注意，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，如下：

```
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
```

- @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），其含有如下：
  - 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。

注解元素及其数据类型

通过上述对@Test注解的定义，我们了解了注解定义的过程，由于@Test内部没有定义其他元素，所以@Test也称为标记注解（marker annotation），但在自定义注解中，一般都会包含一些元素以表示某些值，方便处理器使用，这点在下面的例子将会看到：

```
@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 "";
}
```

上述定义一个名为DBTable的注解，该用于主要用于数据库表与Bean类的映射，与前面Test注解不同的是，我们声明一个String类型的name元素，其默认值为空字符，但是必须注意到对应任何元素的声明应采用方法的声明方式，同时可选择使用default提供默认值，@DBTable使用方式如下：

```
//在类上使用该注解
@DBTable(name = "MEMBER")
public class Member {
    //.......
}
```

关于注解支持的元素数据类型除了上述的String，还支持如下数据类型:

- 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

倘若使用了其他数据类型，编译器将会丢出一个编译错误，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，同时还应该注意到注解也可以作为元素的类型，也就是嵌套注解，下面的代码演示了上述类型的使用过程：

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

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface Reference {
    boolean next() default false;
}

public @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();

    //枚举类型
    enum Status {FIXED, NORMAL}
}
```

编译器对默认值的限制

编译器对元素的默认值有些过分挑剔。首先，元素不能有不确定的值。也就是说，元素必须要么具有默认值，要么在使用注解时提供元素的值。其次，对于非基本类型的元素，无论是在源代码中声明，还是在注解接口中定义默认值，都不能以null作为值，这就是限制，没有什么利用可言，但造成一个元素的存在或缺失状态，因为每个注解的声明中，所有的元素都存在，并且都具有相应的值，为了绕开这个限制，只能定义一些特殊的值，例如空字符串或负数，表示某个元素不存在。

注解不支持继承

注解是不支持继承的，因此不能使用关键字extends来继承某个@interface，但注解在编译后，编译器会自动继承java.lang.annotation.Annotation接口，这里我们反编译前面定义的DBTable注解

```
import java.lang.annotation.Annotation;
//反编译后的代码
public interface DBTable extends Annotation
{
    public abstract String name();
}
```

虽然反编译后发现DBTable注解继承了Annotation接口，请记住，即使Java的接口可以实现多继承，但定义注解时依然无法使用extends关键字继承@interface。

快捷方式

所谓的快捷方式就是注解中定义了名为value的元素，并且在使用该注解时，如果该元素是唯一需要赋值的一个元素，那么此时无需使用key=value的语法，而只需在括号内给出value元素所需的值即可。这可以应用于任何合法类型的元素，记住，这限制了元素名必须为value，简单案例如下

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

//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内置注解及其它元注解

- @Override：用于标明此方法覆盖了父类的方法，源码如下

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

- @Deprecated：用于标明已经过时的方法或类，源码如下

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
public @interface Deprecated {
}
```

- @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，其实现源码如下：

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

其内部有一个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。

@Documented和@Inherited，下面分别介绍：

- @Documented 被修饰的注解会生成到javadoc中

```
import java.lang.annotation.*;

@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() {
    }
}
```

使用javadoc命令生成文档：

```
javadoc DocumentDemo.java DocumentA.java DocumentB.java
```

可以发现使用@Documented元注解定义的注解(@DocumentA)将会生成到javadoc中,而@DocumentB则没有在doc文档中出现，这就是元注解@Documented的作用。

- @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，如下：

```java
import java.lang.annotation.*;

@Inherited
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DocumentA {
}


import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DocumentB {
}


import java.util.Arrays;


@DocumentA
class A {
}

class B extends A {
}

@DocumentB
class C {
}

class D extends C {
}

//测试
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:[@DocumentA()]
     没有使用的@Inherited注解:[]
     */
}
```

注解与反射

前面经过反编译后，我们知道Java所有注解都继承了Annotation接口，也就是说　Java使用Annotation接口代表注解元素，该接口是所有Annotation类型的父接口。同时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，它简要含义如下

- 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

返回值|方法名称|说明 -|:-:|-: \|getAnnotation(Class 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 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:@DocumentA()
         an:[@DocumentA(), @DocumentB()]
         an2:[@DocumentB()]
         b:true
         */
    }
}
```

运行时注解处理器

了解完注解与反射的相关API后，现在通过一个实例（该例子是博主改编自《Tinking in Java》）来演示利用运行时注解来组装数据库SQL的构建语句的过程

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

@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 "";
}
```

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

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
    //该字段对应数据库表列名
    String name() default "";
    //嵌套注解
    Constraints constraint() default @Constraints;
}
```

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

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {

    //对应数据库表的列名
    String name() default "";

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

    Constraints constraint() default @Constraints;
}
```

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

@Target(ElementType.FIELD)//只能应用在字段上
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
    //判断是否作为主键约束
    boolean primaryKey() default false;
    //判断是否允许为null
    boolean allowNull() default false;
    //判断是否唯一
    boolean unique() default false;
}
```

```
@DBTable(name = "MEMBER")
public 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;//个人描述

    //省略set get.....
}
```

上述定义4个注解，分别是@DBTable(用于类上)、@Constraints(用于字段上)、 @SQLString(用于字段上)、@SQLString(用于字段上)并在Member类中使用这些注解，这些注解的作用的是用于帮助注解处理器生成创建数据库表MEMBER的构建语句，在这里有点需要注意的是，我们使用了嵌套注解@Constraints，该注解主要用于判断字段是否为null或者字段是否唯一。必须清楚认识到上述提供的注解生命周期必须为@Retention(RetentionPolicy.RUNTIME)，即运行时，这样才可以使用反射机制获取其信息。有了上述注解和使用，剩余的就是编写上述的注解处理器了，前面我们聊了很多注解，其处理器要么是Java自身已提供、要么是框架已提供的，我们自己都没有涉及到注解处理器的编写，但上述定义处理SQL的注解，其处理器必须由我们自己编写了，如下

```
import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.List;

public class TableCreator {

    public static String createTableSql(String className) throws ClassNotFoundException {
        Class<?> cl = Class.forName(className);
        DBTable dbTable = cl.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 = cl.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 : cl.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
     *
     * @param con
     * @return
     */
    private 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;
    }

    public static void main(String[] args) throws Exception {
        String[] arg = {"Member"};
        for (String className : arg) {
            System.out.println("Table Creation SQL for " +
                    className + " is :\n" + createTableSql(className));
        }

        /**
         * 输出结果:
         Table Creation SQL for 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中注解增强

元注解@Repeateable

元注解@Repeatable是JDK1.8新加入的，它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

```
//Java8前无法这样使用
@FilterPath("/web/update")
@FilterPath("/web/add")
public class A {}
```

Java8前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 { }
```

但在Java8新增了@Repeatable注解后就可以采用如下的方式定义并使用了

```
import java.lang.annotation.*;

//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
@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 {
}
```

我们可以简单理解为通过使用@Repeatable后，将使用@FilterPaths注解作为接收同一个类型上重复注解的容器，而每个@FilterPath则负责保存指定的路径串。为了处理上述的新增注解，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。但请注意，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)。注意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，但请注意如果@FilterPath和@FilterPaths没有使用了@Inherited的话，仍然无法获取。下面通过代码来演示：

```
import java.lang.annotation.*;

//使用Java8新增@Repeatable原注解
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(FilterPaths.class)
@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
         */
    }
}
```