第 3 章　类型系统与可空类型

与 Java、C 和 C ++语言一样，Kotlin 语言也是“静态类型的编程语言”。通常，编程语言中的类型系统中定义了：

- 如何将数值和表达式归为不同的类型；
- 如何操作这些类型；
- 这些类型之间如何互相作用。

我们在编程语言中使用类型的目的是为了让编译器能够确定类型所关联的对象需要分配多少空间。

类型系统在各种语言之间有非常大的差异，主要的差异表现在编译时期的语法及运行时期的操作实现方式上。在每一种编程语言中，都有一个特定的类型系统。静态类型在编译时期就能可靠地发现类型错误，因此通常能增进最终程序的可靠性。然而，有多少的类型错误发生，以及有多少比例的错误能被静态类型所捕获，仍有争论。

本章简单介绍一下 Kotlin 的类型系统。

3.1　类 型 系 统

定型（typing，又称类型指派）的过程就是赋予一组比特以具体的意义。类型通常和存储器中的数值或对象（如变量）相联系。因为在计算机中，任何数值都是由一组简单的比特位组成的，硬件无法区分存储器地址、脚本、字符、整数及浮点数。类型可以告知程序和程序设计者，应该怎么对待那些比特位。

3.1.1　类型系统的作用

使用类型系统，编译器可以检查无意义的、无效的、类型不匹配等错误代码。这也正是强类型语言能够提供更多的代码安全性保障的原因之一。

另外，静态类型检查还可以提供有用的信息给编译器。与动态类型语言相比，由于有了类型的显式声明，静态类型的语言更加易读好懂。

有了类型，我们还可以更好地做抽象化、模块化的工作。这使得我们可以在较高抽象层次思考并解决问题。例如，Java 中的字符数组 char[]s={'a','b','c'} 和字符串类型 String str="abc"就是最简单、最典型的抽象封装实例。下面分别举例说明。

字符数组代码示例如下：

```
jshell> char[] s = {'a','b','c'}    //字符数组
s ==> char[3] { 'a', 'b', 'c' }
jshell> s[0]                        //访问第一个元素，注意下标是 0
$3 ==> 'a'

jshell> s[1]
$4 ==> 'b'

jshell> s[2]
$5 ==> 'c'
```

字符串代码示例如下：

```
jshell> String str = "abc"          //声明字符串
str ==> "abc"

jshell> str.toCharArray();          //String 类型转换成字符数组
$7 ==> char[3] { 'a', 'b', 'c' }
```

3.1.2 Java 类型系统

Java 类型系统可以简单用下面的图 3-1 来表示。

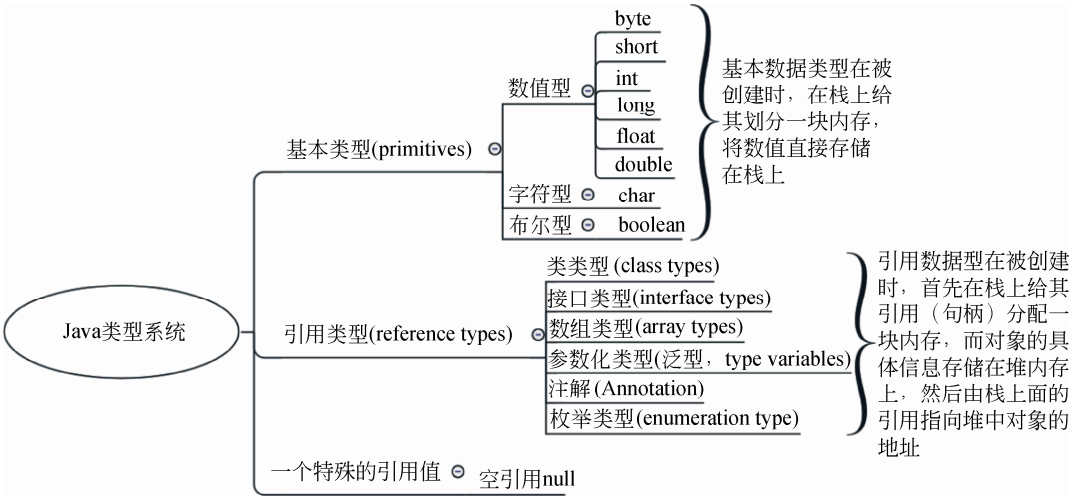

图 3-1　Java 类型系统

关于 Java 中的 null，有很多比较“坑”的地方。例如：

```
int i = null;       //type mismatch : cannot convert from null to int
short s = null;     //type mismatch : cannot convert from null to short
byte b = null:      //type mismatch : cannot convert from null to byte
double d = null;    //type mismatch : cannot convert from null to double
Integer io = null;  //编译
int j = io;         //编译 ok，但运行时报 NullPointerException
```

基本数据类型与引用数据类型在创建时，内存存储方式区别如下：

- 基本数据类型在被创建时，在栈上给其划分一块内存，将数值直接存储在栈上（性能高）；
- 引用数据类型在被创建时，首先在栈上给其引用（句柄）分配一块内存，而对象的具体信息存储在堆内存上，然后由栈上面的引用指向堆中对象的地址。

3.1.3　Kotlin 的类型系统

Java 是一个近乎“纯洁”的面向对象编程语言，但是为了编程方便还是引入了基本数据类型。为了能够将这些基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类型就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。Java 为每个原始类型提供了相应的包装类型。

- 原始类型：boolean，char，byte，short，int，long，float，double。
- 相应的包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double。

Kotlin 中去掉了原始类型，只有包装类型，编译器在编译代码的时候，会自动优化性能，把对应的包装类型拆箱为原始类型。

Kotlin 系统类型分为可空类型和不可空类型。Kotlin 中引入了可空类型，把有可能为 null 的值单独用可空类型来表示。这样就在可空引用与不可空引用之间划分出一条明确的、显式的“界线”。

Kotlin 类型层次结构如图 3-2 所示。

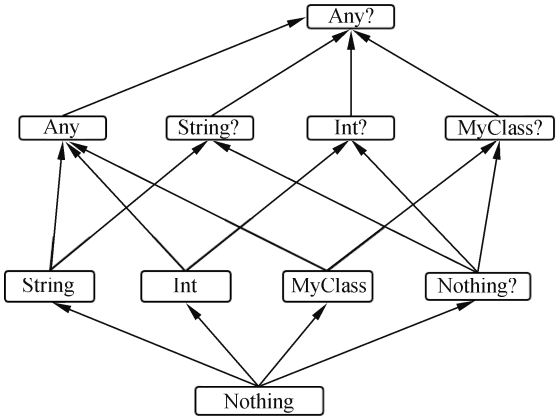

图 3-2　Kotlin 类型层次结构

通过这样显式地使用可空类型，并在编译期作类型检查，大大降低了出现空指针异常的概率。

对于 Kotlin 中的数字类型而言，不可空类型与 Java 中原始的数字类型对应，如表 3-1 所示。

Kotlin 中对应的可空数字类型就相当于 Java 中的装箱数字类型，如表 3-2 所示。

表 3-1　Kotlin中的数字类型与Java中原始的数字类型

Kotlin	Java
Int	int
Long	long
Float	float
Double	double

表 3-2　Kotlin中的可空数字类型与Java中的装箱数字类型

Kotlin	Java
Int?	Integer
Long?	Long
Float?	Float
Double?	Double

在 Java 中，从基本数字类型到引用数字类型的转换过程就是典型的装箱操作，例如 int 转为 Integer。倒过来，从 Integer 转为 int 就是拆箱操作。同理，在 Kotlin 中非空数字类型 Int 到可空数字类型 Int?需要进行装箱操作。同时，非空的 Int 类型会被编译器自动拆箱成基本数据类型 int，存储的时候也会存到栈空间。例如下面的代码，当为 Int 类型的时候，a===b 返回的是 true；而当为 Int?的时候，a===b 返回的是 false。

```
>>> val a: Int = 1000
>>> val b:Int = 1000
>>> a===b       //引用相等
true
>>> a==b        //值相等
true
```

上面返回的都是 true，因为 a、b 它们都是以原始类型存储的，类似于 Java 中的基本数字类型。

```
>>> val a:Int? = 1000
>>> val b:Int? = 1000
>>> a==b
true
>>> a===b //可空类型 Int?等价于 Java 中的 Integer 包装类型，这里 a、b 引用不相等
false
```

可以看出，当 a、b 都为可空类型时，a 与 b 的引用是不等的。“等于”号的简单说明如表 3-3 所示。

表 3-3　Kotlin中的“等于”号说明

“等于”符号	功 能 说 明
=	赋值，在逻辑运算时也有效
==	等于运算，比较的是值，而不是引用
===	完全等于运算，不仅比较值，而且还比较引用，只有两者一致才为真

另外，Java 中的数组也是一个较为特殊的类型。这个类型是 T[]，这个方括号让我们觉得不太“优雅”。Kotlin 中摒弃了这个数组类型声明的语法。Kotlin 简单直接地使用 Array 类型代表数组类型。这个 Array 中定义了 get、set 算子函数，同时有一个 size 属性代表数

组的长度，还有一个返回数组元素的迭代子 Iterator 的函数 iterator()。完整的定义如下：

```
public class Array<T> {
    public inline constructor(size: Int, init: (Int) -> T)
    public operator fun get(index: Int): T
    public operator fun set(index: Int, value: T): Unit
    public val size: Int
    public operator fun iterator(): Iterator<T>
}
```

其中，构造函数我们可以这么用：

```
>>> val square = Array(5, { i -> i * i }) //构造 5 个元素的数组，元素初始值是 i*i
>>> square.forEach(::println)
0
1
4
9
16
```

在编程过程中常用的是 boolean[]、char[]、byte[]、short[]、int[]、long[]、float[]、double[]；Kotlin 直接使用了 8 个新的类型来对应这样的编程场景：

```
BooleanArray
ByteArray
CharArray
DoubleArray
FloatArray
IntArray
LongArray
ShortArray
```

3.2　可 空 类 型

或许 Java 和 Android 开发者早已厌倦了空指针异常（Null Pointer Exception）。因为其在运行时总会在某个意想不到的地方忽然出现，让开发者感到措手不及。

那么为何开发者不能在编译时就提前发现这类空指针异常，并大量修复这些问题呢？现代编程语言正是这么做的。Kotlin 自然也不例外。在 Java 8 中，我们可以使用 Optional 类型来表达可空的类型。

```
package com.easy.kotlin;

import java.util.Optional;
import static java.lang.System.out;

public class Java8OptionalDemo {

    public static void main(String[] args) {
        out.println(strLength(Optional.of("abc")));
        out.println(strLength(Optional.ofNullable(null)));
    }

    static Integer strLength(Optional<String> s) {
        return s.orElse("").length();
```

```
                    //Optional 类型中的 orElse 方法，等价于 Elvis 表达式的逻辑
    }
}
```

运行程序，输出如下：

```
3
0
```

但是这样的代码依然不是那么“优雅”。

针对这方面，Groovy 提供了一种安全的属性/方法访问操作符“?.”：

```
user?.getUsername()?.toUpperCase(); //安全调用符 ?.
```

Swift 也有类似的语法，只作用在 Optional 的类型上。

Kotlin 中使用了 Groovy 里面的安全调用符，并简化了 Optional 类型的使用，直接通过在类型 T 后面加个“?”，就表达了 Optional 的意义。

上面 Java 8 的例子用 Kotlin 来写就显得更加简单、“优雅”了：

```
package com.easy.kotlin

fun main(args: Array<String>) {
    println(strLength(null))
    println(strLength("abc"))
}

fun strLength(s: String?): Int {
    return s?.length ?: 0          //?.是安全调用符，?:是 Elvis 操作符
}
```

其中，我们使用 String?同样表达了 Optional 的意思，相比之下，哪种方式更简单？答案一目了然。

还有 Java 8 的 Optional 提供的 orElse：

```
s.orElse("").length();
```

其在 Kotlin 中是最最常见的 Elvis 运算符了：

```
s?.length ?: 0
```

相比之下，我们还有什么理由继续用 Java 8 的 Optional 呢？

3.3 安全操作符

扔掉 Java 中的一堆 null 的防御式样板代码吧！当我们使用 Java 开发的时候，我们的代码大多是防御性的。如果我们不想遇到 NullPointerException，就需要在使用它之前不停地判断它是否为 null。

Kotlin 正如很多现代编程语言一样是空安全的。因为我们需要通过一个可空类型符号“T?”来明确地指定一个对象类型 T 是否能为空。

我们可以像这样去写：

```
>>> val str: String = null //编译不通过：不可空 String 类型的 str 禁止赋值 null
error: null can not be a value of a non-null type String
val str: String = null
               ^
```

可以看到，这里不能通过编译，因为 String 类型不能是 null。

一个可以赋值为 null 的 String 类型的正确写法是：String?，代码如下：

```
>>> var nullableStr: String? = null
        //可空类型 String?，nullableStr 可能会导致空指针异常
>>> nullableStrnull
```

我们再来看一下 Kotlin 中关于 null 的一些有趣的运算。null 与 null 是相等的：

```
>>> null==null
true
>>> null!=null
false
```

null 这个值比较特殊，null 不是 Any 类型，例如：

```
>>> null is Any
false
```

但是，null 是 Any?类型，例如：

```
>>> null is Any?
true
```

我们再来看看 null 对应的类型是什么：

```
>>> var a=null
>>> a
null
>>> a=1
error: the integer literal does not conform to the expected type Nothing?
a=1
  ^
```

从报错信息中可以看出，null 的类型是 Nothing?。关于 Nothing?的内容，将会在后面介绍。

3.3.1　安全调用符“?”

我们不能直接使用可空的 nullableStr 来调用其属性或者方法，例如下面的代码直接报错：

```
>>> nullableStr.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a
nullable receiver of type String?
nullableStr.length
          ^
```

上面的代码无法编译，nullableStr 可能是 null。我们需要使用安全调用符“?.”来调用：

```
>>> var nullableStr: String? = null
```

```
>>> nullableStr?.length
null
>>> null ableStr = "abc"
>>> nullableStr?.length //安全调用符“?.”
3
```

只有在 nullableStr!=null 时才会去调用其 length 属性。

3.3.2 非空断言“!!”

Kotlin 中提供了断言操作符“!!”，使得可空类型对象可以调用成员方法或者属性（但遇见 null，就会导致空指针异常），代码示例如下：

```
>>> nullableStr = null
>>> nullableStr!!.length           //抛出空指针异常
kotlin.KotlinNullPointerException
```

3.3.3 Elvis 运算符“?:”

使用 Elvis 操作符“?:”来给定一个在 null 情况下的替代值：

```
>>> nullableStr
null
>>> var s= nullableStr?:"NULL" //当 s 是 null 的时候，返回"NULL"字符串
>>> s
NULL
```

3.4 特 殊 类 型

本节我们介绍 Kotlin 中的特殊类型：Unit、Nothing、Any 及其对应的可空类型 Unit?、Nothing?、Any?。

3.4.1 Unit 类型

Kotlin 也是面向表达式的语言。在 Kotlin 中所有控制流语句都是表达式（除了变量赋值、异常等）。

Kotlin 中的 Unit 类型实现了与 Java 中的 void 一样的功能。总的来说，这个 Unit 类型并没有什么特别之处。它的定义如下：

```
package kotlin
public object Unit {                          //Unit 类型是一个 object 对象类型
    override fun toString() ="kotlin.Unit"  //toString()函数返回值
}
```

不同的是，当一个函数没有返回值的时候，我们用 Unit 来表示这个特征，而不是 null。大多数时候，我们并不需要显式地返回 Unit，或者声明一个函数的返回类型为 Unit。

编译器会推断出它。代码示例如下：

```
>>> fun unitExample(){println("Hello,Unit")}
>>> val helloUnit = unitExample()
Hello,Unit
>>> helloUnit              //函数的返回类型是 Unit
kotlin.Unit
>>> println(helloUnit)
kotlin.Unit
>>> helloUnit is Unit      //判断是否 Unit 类型
true
```

可以看出，变量 helloUnit 的类型是 kotlin.Unit。下面几种写法是等价的：

```
@RunWith(JUnit4::class)
class UnitDemoTest {
    @Test fun testUnitDemo() {
        val ur1 = unitReturn1()
        println(ur1) //kotlin.Unit
        val ur2 = unitReturn2()
        println(ur2) //kotlin.Unit
        val ur3 = unitReturn3()
        println(ur3) //kotlin.Unit
    }

    fun unitReturn1() {             //空函数体，返回类型是 Unit

    }

    fun unitReturn2() {             //显式 return
        return Unit
    }

    fun unitReturn3(): Unit {       //显式声明返回类型 Unit
    }
}
```

跟其他类型一样，Kotlin.Unit 父类型是 Any。如果是一个可空的 Unit?，那么父类型是 Any?。Unit 类型层次结构如图 3-3 所示。

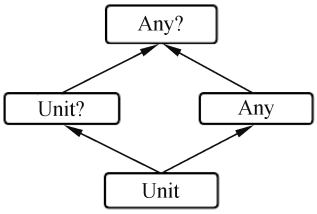

图 3-3　Unit 类型层次结构

3.4.2　Nothing 与 Nothing?类型

在 Java 中，void 不能是变量的类型，也不能作为值打印输出。但是在 Java 中有个包装类 Void 是 void 的自动装箱类型。如果你想让一个方法的返回类型永远是 null 的话，可以把返回类型置为这个大写的 Void 类型。

代码示例如下：

```
public Void voidDemo() {        //声明方法的返回类型是 Void
    System.out.println("Hello,Void");
    return null;                //这个返回类型 Void 的方法只能返回 null 值
}
```

测试代码如下：

```
@RunWith(JUnit4.class)
public class VoidDemoTest {
    @Test
    public void testVoid() {
        VoidDemo voidDemo = new VoidDemo();
        Void v = voidDemo.voidDemo();    //输出：Hello,Void
        System.out.println(v);           //输出：null
    }
}
```

这个 Void 对应 Kotlin 中的 Nothing?，其唯一可被访问的返回值也是 null。

如 3.13 节中的 Kotlin 类型层次结构图（图 3-2）所示，在 Kotlin 类型层次结构的最底层就是 Nothing 类型，Nothing 类型层次结构如图 3-4 所示。

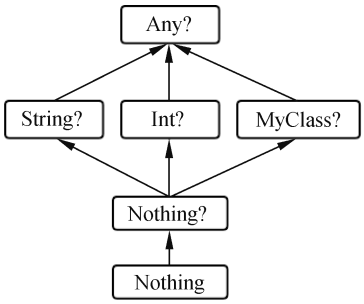

图 3-4　Nothing 类型层次结构

Nothing 类的定义如下：

```
public class Nothing private constructor()
                    //Nothing 的构造函数是 private 的，外界无法创建 Nothing 对象
```

这个 Nothing()不能被实例化：

```
>>> Nothing() is Any    //不能实例化，因为默认的主构造函数是私有的
error: cannot access '<init>': it is private in 'Nothing'
Nothing() is Any
^
```

从上面的代码示例中可以看出 Nothing()不可被访问。如果一个函数的返回值是 Nothing，就意味着这个函数永远不会有返回值。

但是我们可以使用 Nothing 来表达一个从来不存在的返回值。例如 EmptyList 中的 get()函数

```
internal object EmptyList : List<Nothing>, Serializable, RandomAccess {
    override fun get(index: Int): Nothing = throw IndexOutOfBoundsException
    ("Empty list doesn't contain element at index $index.")
                                            //get 函数的返回类型是 Nothing
```

```
    }
}
```

一个空的 List 调用 get()函数，直接抛出了 IndexOutOfBoundsException，这个时候可以使用 Nothing 作为这个 get()函数的返回类型，因为它永远不会返回某个值，而是直接抛出了异常。

再例如 Kotlin 标准库里面的 exitProcess()函数：

```
@file:kotlin.jvm.JvmName("ProcessKt")
@file:kotlin.jvm.JvmVersion
package kotlin.system
@kotlin.internal.InlineOnly
public inline fun exitProcess(status: Int): Nothing {
                                    //exitProcess 函数的返回类型是 Nothing
    System.exit(status)
    throw RuntimeException("System.exit returned normally, while it was
    supposed to halt JVM.")
}
```

注意：Unit 与 Nothing 之间的区别是，Unit 类型表达式计算结果的返回类型是 Unit; Nothing 类型的表达式计算结果是永远不会返回的（与 Java 中的 void 相同）。

Nothing?可以只包含一个值 null。代码示例如下：

```
>>> var nul:Nothing?=null
>>> nul = 1          //Nothing?除了 null 值之外，不能赋其他值
error: the integer literal does not conform to the expected type Nothing?
nul = 1
      ^
>>> nul = true
error: the boolean literal does not conform to the expected type Nothing?
nul = true
      ^
>>> nul = null       //Nothing? 只能赋值为 null 值
>>> nul
null
```

从上面的代码示例中可以看出：Nothing?唯一允许的值是 null，可被用作任何可空类型的空引用。

3.4.3　Any 与 Any?类型

就像 Any 是在非空类型层次结构的根一样，Any?是可空类型层次的根。Any?是 Any 的超集，Any?是 Kotlin 类型层次结构的最顶端。Any 类型层次结构如图 3-5 所示。

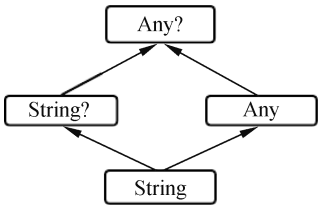

图 3-5　Any 类型层次结构

代码示例：

```
>>> 1 is Any         //Int 类型的 1 是 Any 类型
true
>>> 1 is Any?        //Int 类型的 1 是 Any?类型
true
>>> null is Any     //null 不是 Any 类型
false
>>> null is Any?    //null 是 Any?类型
true
>>> Any() is Any?   //Any()是 Any?类型
true
```

3.5 类型检测与类型转换

Kotlin 在运行时通过使用 is 操作符或其否定形式!is 来检查对象是否符合给定类型。在一般情况下，不需要在 Kotlin 中使用显式转换操作符，因为编译器跟踪不可变值的 is 检查，并在需要时自动插入（安全的）转换。下面分别具体介绍。

3.5.1 is 运算符

is 运算符可以检查对象 A 是否与特定的类型 X 兼容（此对象 A 是 X 类型或者派生于 X 类型），还可以用来检查一个对象（变量）是否属于某数据类型（如 Int、String、Boolean 等）。C#里面也有 is 运算符。

is 运算符类似 Java 中的 instanceof：

```
jshell> "abc" instanceof String    //Java 中的 instanceof 操作符
$10 ==> true
```

在 Kotlin 中，我们可以在运行时通过使用 is 运算符或其否定形式!is，来检查对象是否符合给定类型：

```
>>> "abc" is String
true
>>> "abc" !is String
false

>>> null is Any
false
>>> null is Any?
true
```

代码示例如下：

```
@RunWith(JUnit4::class)
class ISTest {
    @Test fun testIS() {
        val foo = Foo()
        val goo = Goo()
        println(foo is Foo)  //true
```

```
        println(goo is Foo)  //子类 is 父类=true
        println(foo is Goo)  //父类 is 子类=false
        println(goo is Goo)  //true
    }
}

open class Foo
class Goo : Foo()
```

3.5.2　类型自动转换

在 Java 代码中，当我们使用 str instanceof String 来判断其值为 true 的时候，我们想使用 str 变量，还需要显式地强制转换类型：

```
@RunWith(org.junit.runners.JUnit4.class)
public class TypeSystemDemo {
    @org.junit.Test
    public void testVoid() {
        Object str = "abc";
        if (str instanceof String) {
            int len = ((String)str).length();  //显式地强制转换类型为 String
            println(str + " is instanceof String");
            println("Length: " + len);
        } else {
            println(str + " is not instanceof String");
        }
        boolean is = "1" instanceof String;
        println(is);
    }
    void println(Object obj) {
        System.out.println(obj);
    }
}
```

而大多数情况下不需要在 Kotlin 中使用显式转换操作符，因为编译器会跟踪不可变值的 is 检查，并在需要时自动插入（安全的）转换：

```
@Test fun testIS() {
    val len = strlen("abc")
    println(len) //3
    val lens = strlen(1)
    println(lens) //1
}

fun strlen(ani: Any): Int {
    if (ani is String) { //ani 变量的类型如果是 String 类型，编译器会存储它的类型
        return ani.length
                    //这里的 ani 类型已经是 String，可以直接作为 String 类型使用
    } else if (ani is Number) {
        return ani.toString().length
    } else if (ani is Char) {
        return 1
    } else if (ani is Boolean) {
        return 1
    }
    print("Not A String")
    return -1
}
```

3.5.3 as 运算符

as 运算符用于执行引用类型的显式类型转换。如果要转换的类型与指定的类型兼容，转换就会成功进行；如果类型不兼容，使用 as?运算符就会返回 null。

代码示例如下：

```
>>> open class Foo      //父类 Foo
>>> class Goo:Foo()     //子类 Goo
>>> val foo = Foo()
>>> val goo = Goo()
>>> foo as Goo          //父类型不能强制转换为子类型
java.lang.ClassCastException: Line69$Foo cannot be cast to Line71$Goo
>>> foo as? Goo
null
>>> goo as Foo          //子类型可以转换为父类型
Line71$Goo@73dce0e6
```

可以看出，在 Kotlin 中，父类是禁止转换为子类型的。

按照 Liskov 替换原则，父类转换为子类是对 OOP 的严重违反，因为子类除了包含父类所有的方法和属性之外，还可以自定义成员方法与属性，而父类则未必具有和子类同样的成员，所以这种转换是不允许的。

3.6 本 章 小 结

Kotlin 通过引入可空类型，在编译时就大量“清扫了”空指针异常。同时，Kotlin 中还引入了安全调用符“?.”及 Elvis 操作符“?:”，使得我们的代码写起来更加简洁。

Kotlin 的类型系统比 Java 更加简单、一致，Java 中的原始类型与数组类型在 Kotlin 中都统一表现为引用类型。

Kotlin 中还引入了 Unit、Nothing 等特殊类型，使得没有返回值的函数与永远不会返回的函数有了更加规范和一致的签名。

我们可以使用 is 操作符来判断对象实例的类型，使用 as 操作符进行类型的转换。