### L1.IDEA部署Kotlin环境

## IDEA环境部署

### 1.1 下载IDEA编译器下载最新版的就可以了

<https://www.jetbrains.com/idea/>

### 1.2 安装IDEA插件

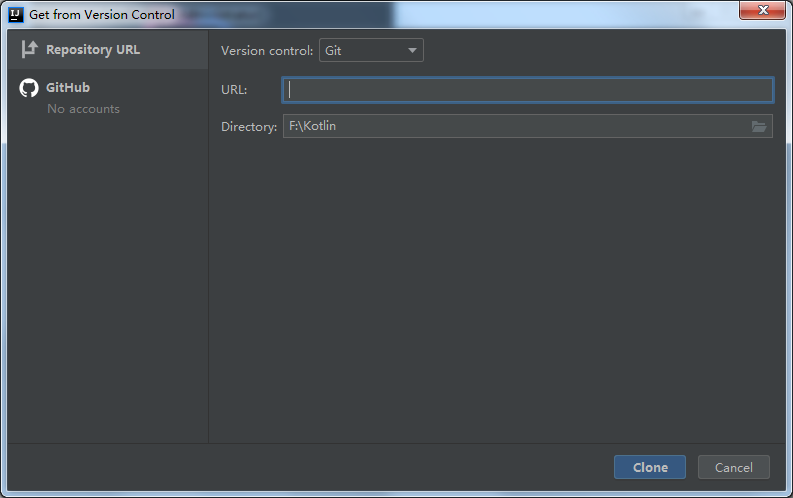
<https://zhile.io/2020/11/18/jetbrains-eval-reset-da33a93d.html>

### 1.3 获取教程项目

完成以上部署后获取教程项目：

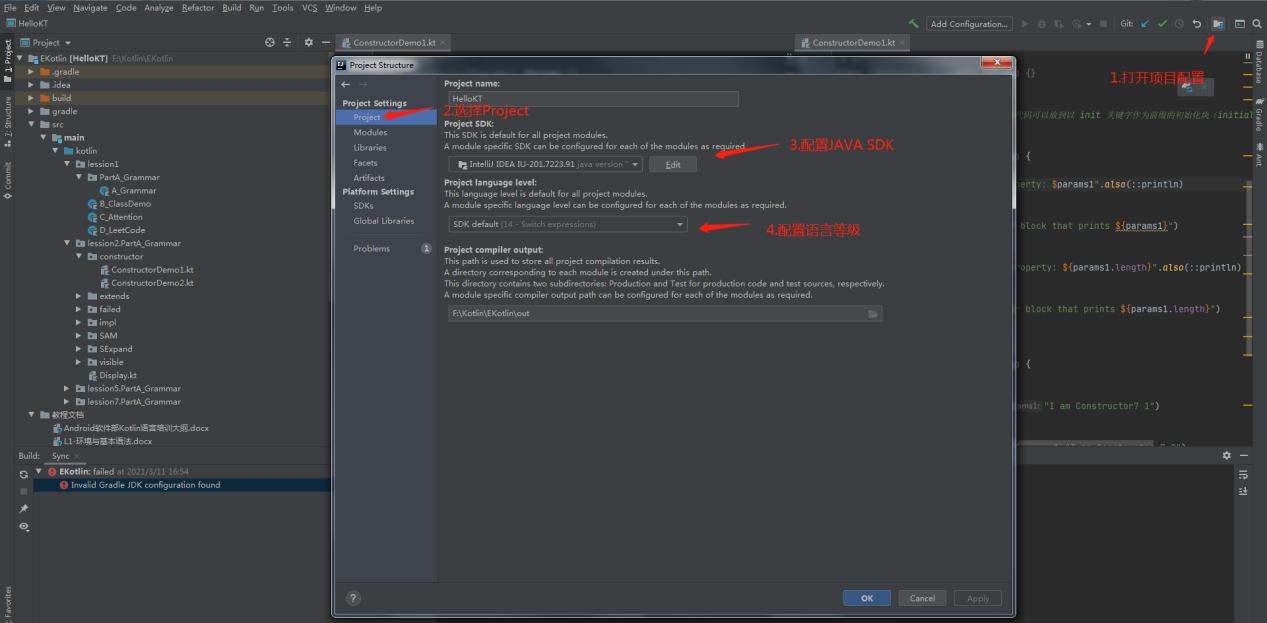


输入: <https://github.com/JustGank/EKotlin.git>



正常Build获取即可。

### 1.4 获取完成配置项目



配置完成后项目不报错，代表环境部署成功。

此时就具备了Kotlin语法的实验环境。

## 二、基本语法介绍

此部分作为Kotlin的大体语法展示单元，主要是展示Kotlin在常规编程中的使用方式。此部分讲解时需要体现特性和重点，部分细节可以放到后面单元专项中进行讲解。

### 2.1 包的定义与导入

包的声明应处于源文件顶部：

package my.demo

import kotlin.text.\*

此部分和JAVA类似，不用特殊关注

### 2.2 程序的入口点

Kotlin 应用程序的入口点是 main 函数。

fun main() {

println("Hello world!")

}

每一个文件中都可以有 fun main()的存在。这样可以很方便的让我们在每个文件内部完成展示和调试。

### 2.3函数

带有两个 Int 参数、返回 Int 的函数：

fun sum1(a: Int, b: Int): Int {

return a + b

}

Kotlin是先声明变量名，再声明变量类型。

将表达式作为函数体、返回值类型自动推断的函数：

fun sum2(a: Int, b: Int) = a + b

函数返回无意义的值,此处可以理解为Void,Unit也可以不写：

fun printSum(a: Int, b: Int): Unit {

println("sum of $a and $b is ${a + b}")

}

### 2.4变量

定义只读局部变量使用关键字 val 定义。**只能为其赋值一次。**

val a: Int = 1 // 立即赋值

val b = 2 // 自动推断出 `Int` 类型

val c: Int // 如果没有初始值类型不能省略

c = 3 // 明确赋值

可重新赋值的变量使用 var 关键字：

var x = 5 // 自动推断出 `Int` 类型

x += 1

顶层变量：

val PI = 3.14

var x = 0

fun incrementX() {

x += 1

}

### 2.5 注释

与大多数现代语言一样，Kotlin 支持单行（或行末）与多行（块）注释。

// 这是一个行注释

/\* 这是一个多行的

块注释。 \*/

Kotlin 中的块注释可以嵌套。

/\* 注释从这里开始

/\* 包含嵌套的注释 \*/

并且在这里结束。 \*/

### 2.6 字符串模板

此处就是$引用，直接引用变量时使用$即可，使用变量的某个变量时可以使用 {} 将要输入的内容包起来。

var a = 1

// 模板中的简单名称：

val s1 = "a is $a"

a = 2

// 模板中的任意表达式：

val s2 = "${s1.replace("is", "was")}, but now is $a"

### 2.7 条件表达式

下面两种写法等价：

fun maxOf(a: Int, b: Int): Int {

if (a > b) {

return a

} else {

return b

}

}

fun maxOf(a: Int, b: Int) = if (a > b) a else b（简单逻辑更推荐）

### 2.8 空值与 null 检测

**当某个变量的值可以为 null 的时候，必须在声明处的类型后添加 “？” 来标识该引用可为空。**

如果 str 的内容不是数字返回 null：

fun parseInt(str: String): Int? {}

使用返回可空值的函数:

fun printProduct(arg1: String, arg2: String) {

val x = parseInt(arg1)

val y = parseInt(arg2)

// 直接使用 `x \* y` 会导致编译错误，因为它们可能为 null

if (x != null && y != null) {

// 在空检测后，x 与 y 会自动转换为非空值（non-nullable）

println(x \* y)

}

else {

println("'$arg1' or '$arg2' is not a number")

}

if (x == null) {

println("Wrong number format in arg1: '$arg1'")

return

}

if (y == null) {

println("Wrong number format in arg2: '$arg2'")

return

}

// 在空检测后，x 与 y 会自动转换为非空值

println(x \* y)

}

### 2.9 类型检测与自动类型转换

is 运算符检测一个表达式是否某类型的一个实例。 如果一个不可变的局部变量或属性已经判断出为某类型，那么检测后的分支中可以直接当作该类型使用，无需显式转换：

fun getStringLength(obj: Any): Int? {

if (obj is String) {

// `obj` 在该条件分支内自动转换成 `String`

return obj.length

}

// 在离开类型检测分支后，`obj` 仍然是 `Any` 类型

return null

}

写法1：

fun getStringLength(obj: Any): Int? {

if (obj !is String) return null

// `obj` 在这一分支自动转换为 `String`

return obj.length

}

写法2：

fun getStringLength(obj: Any): Int? {

// `obj` 在 `&&` 右边自动转换成 `String` 类型

if (obj is String && obj.length > 0) {

return obj.length

}

return null

}

### 2.10 for 循环

val items = listOf("apple", "banana", "kiwifruit")

for (item in items) {

println(item)

}

有索引的遍历：

val items = listOf("apple", "banana", "kiwifruit")

for (index in items.indices) {

println("item at $index is ${items[index]}")

}

### 2.11 while 循环

val items = listOf("apple", "banana", "kiwifruit")

var index = 0

while (index < items.size) {

println("item at $index is ${items[index]}")

index++

}

### 2.12 when 表达式

fun describe(obj: Any): String =

when (obj) {

1 -> "One"

"Hello" -> "Greeting"

is Long -> "Long"

!is String -> "Not a string"

else -> "Unknown"

}

### 2.13 使用区间（range）

使用 in 运算符来检测某个数字是否在指定区间内，区间是包含收尾元素的！：

val x = 10

val y = 9

if (x in 1..y+1) {

println("fits in range")

}

检测某个数字是否在指定区间外:

val list = listOf("a", "b", "c")

if (-1 !in 0..list.lastIndex) {

println("-1 is out of range")

}

if (list.size !in list.indices) {

println("list size is out of valid list indices range, too")

}

区间迭代:

for (x in 1..5) {

print(x)

}

或数列迭代：

//代表从1开始 每次+2 一直到9

for (x in 1..10 step 2) {

print(x)

}

输出：13579

//在9-0这个逆序区间 每次减3

for (x in 9 downTo 0 step 3) {

print(x)

}

输出：9630

### 2.14 集合

对集合进行迭代:

for (item in items) {

println(item)

}

使用 in 运算符来判断集合内是否包含某实例：

//这是代替 if-else if-else if 的一种语法糖

val items = setOf("apple", "banana", "kiwifruit")

when {

"orange" in items -> println("juicy")

"apple" in items -> println("apple is fine too")

}

使用 lambda 表达式来过滤（filter）与映射（map）集合：

val fruits = listOf("banana", "avocado", "apple", "kiwifruit")

fruits.filter { it.startsWith("a") }

.sortedBy { it }

.map { it.toUpperCase() }

.forEach { println(it) }

### 2.15 创建基本类及其实例

fun main() {

val rectangle = Rectangle(5.0, 2.0)

val triangle = Triangle(3.0, 4.0, 5.0)

println("Area of rectangle is ${rectangle.calculateArea()}, its perimeter is ${rectangle.perimeter}")

println("Area of triangle is ${triangle.calculateArea()}, its perimeter is ${triangle.perimeter}")

}

//抽象父类 “多边形” 直接声明构造方法

abstract class Shape(val sides: List<Double>) {

val perimeter: Double get() = sides.sum()

abstract fun calculateArea(): Double

}

interface RectangleProperties {

val isSquare: Boolean

}

//声明矩形类

class Rectangle(

var height: Double,

var length: Double

) : Shape(listOf(height, length, height, length)), RectangleProperties {

//重写接口属性

override val isSquare: Boolean get() = length == height

//重写面积计算

override fun calculateArea(): Double = height \* length

}

class Triangle(

var sideA: Double,

var sideB: Double,

var sideC: Double

) : Shape(listOf(sideA, sideB, sideC)) {

override fun calculateArea(): Double {

val s = perimeter / 2

return Math.sqrt(s \* (s - sideA) \* (s - sideB) \* (s - sideC))

}

}

## 三、习惯用法

### 3.1 创建 DTOs（POJOs/POCOs）

POJOs有时候也称作Plain Ordinary Java Objects，表示一个数据集合。

POJO只是一个普通的，已删除限制的Java Bean。 Java Bean必须满足以下要求：

1. 默认无参数构造函数

2. 对于值foo为可变属性，遵循get()方法和set()方法的Bean协议；如果值foo是不可变的，则不使用setFoo。

3.必须实现java的序列化(java.io.Serializable)

POJO不强制执行这些操作。顾名思义：在JDK下编译的对象可以被认为是一个普通的Java对象。没有应用服务器，没有基类，没有需要使用的接口。

**简答来说就是使用data class声明数据Bean。**

**data class Customer(val name: String, val email: String)**

**声明后，编译器会为我们重写很多方法：**

* 所有属性的 getters （对于 var 定义的还有 setters）
* equals()
* hashCode()
* toString()
* copy()
* 所有属性的 component1()、 component2()……等等

### 3.2 函数的默认参数

fun foo(a: Int = 0, b: String = "") { …… }

使用带默认参数的方法，有默认参数的参数，其值是选写的。

### 3.3 过滤 list

val positives = list.filter { x -> x > 0 }

或者可以更短:

val positives = list.filter { it > 0 }

### 3.4 检测元素是否存在于集合中

In在返回true，!in不在返回true

if ("john@example.com" in emailsList) { …… }

if ("jane@example.com" !in emailsList) { …… }

### 3.5字符串内插(同2.6)

println("Name $name")

### 3.6 类型判断

when (x) {

is Foo //-> ……

is Bar //-> ……

else //-> ……

}

### 3.7 遍历 map/pair型list

for ((k, v) in map) {

println("$k -> $v")

}

### 3.8 使用区间

for (i in 1..100) { …… }  **// 闭区间：包含 100**

for (i in 1 until 100) { …… } // **半开区间：不包含 100**

for (x in 2..10 step 2) { …… }

for (x in 10 downTo 1) { …… }

if (x in 1..10) { …… }

### 3.9 只读 list(只读代表声明后不能编辑)

val list = listOf("a", "b", "c")

### 3.10 只读 map(to 是 Pair 声明的一个中缀拓展函数)

val map = mapOf("a" to 1, "b" to 2, "c" to 3)

### 3.11 访问  map（mutable代表可读写 ）

fun visitMap() {  
 val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)  
 map["key"] = 123  
 println(map["key"])  
}

### 3.13 延迟属性(使用时初始化变量值)

val p: String by lazy {

println(“Init p”)

“Hello!”

}

延迟属性Lazy 与 lateinit 区别

以下是lateinit var和by lazy { ... }委托属性之间的显著差异：

lazy { ... }代表只能用于val属性，而lateinit只能用于var，因为它不能编译到final字段，因此不能保证不变性;

lateinit var具有存储值的后备字段(backing field)，而by lazy { ... }创建一个委托对象，其中存储一次计算的值，将对代理实例的引用存储在类对象中，并为与委托实例一起使用的属性生成getter。

除了val之外，lateinit不能用于可空属性和Java原语类型(这是因为null用于未初始化的值);所以如果你需要在类中存在的支持字段，请使用lateinit; lateinit var可以从对象被看到的任何地方被初始化。从一个框架代码的内部，多个初始化方案是可能的单一类的不同对象。 by lazy { ... }反过来又定义了属性的唯一初始化器，只能通过覆盖子类中的属性进行更改。如果您希望以预先未知的方式从外部初始化属性，请使用lateinit。

另外，还有一个方法没有提到Delegates.notNull()，它适用于non-null属性的延迟初始化，包括Java原始类型的属性。

延迟属性Lazy 与 lateinit 使用总结

lateinit用于外部初始化：当需要外部资料通过调用方法初始化您的值时。 例如通过调用：

private lateinit var value: MyClass

fun init(externalProperties: Any) {

value = somethingThatDependsOn(externalProperties)}

而lazy当它只使用对象内部的依赖关系时。 嗯。静态变量的初始化挺适合这种方式的。

### 3.14 拓展函数

通过：类.拓展方法 声明

fun String.spaceToCamelCase() { …… }

### 3.15 创建单例（伴生对象）

object Resource {

val name = "Name"

}

里面的属性和方法都可以使用 类.的形式直接使用。

### 3.16 If not null 缩写

**当files为空时println会输出null**

val files = File("Test").listFiles()

println(files?.size)

// ！！断言会空针异常  
println(files!!.size)

### 3.17 If not null and else 缩写

？：类似于 三目原运算，当files为空的时候会输出empty

val files = File("Test").listFiles()

println(files?.size ?: "empty")

### 3.18 if null 执行一个语句

val values = ……

val email = values["email"] ?: throw IllegalStateException("Email is missing!")

### 3.19 在可能会空的集合中取第一元素

val emails = …… // 可能会是空集合 emails 需要是集合类型

val mainEmail = emails.firstOrNull() ?: ""

### 3.20 if not null 执行代码

val value = ……

value?.let {

…… // 代码会执行到此处, 假如data不为null

}

### 3.21 映射可空值（如果非空的话）

val value = ……

val mapped = value?.let { transformValue(it) } ?: defaultValue

// 如果该值或其转换结果为空，那么返回 defaultValue。

### 3.22返回 when 表达式

fun transform(color: String): Int {

return when (color) {

"Red" -> 0

"Green" -> 1

"Blue" -> 2

else -> throw IllegalArgumentException("Invalid color param value")

}

}

### 3.23 “try/catch”表达式

fun test() {

val result = try {

count()

} catch (e: ArithmeticException) {

throw IllegalStateException(e)

}

}

### 3.24 “if”表达式

fun foo(param: Int) {

val result = if (param == 1) {

"one"

} else if (param == 2) {

"two"

} else {

"three"

}

}

### 3.25 返回类型为 Unit 的方法的 Builder 风格用法

Fill()函数 是简单的将集合内的所有元素替换成指定元素

fun arrayOfMinusOnes(size: Int): IntArray {

//生成一个大小为Size的集合 让里面的元素都是-1

return IntArray(size).apply { fill(-1) }

}

### 3.26 单表达式函数

fun theAnswer() = 42

等价于

fun theAnswer(): Int {

return 42

}

单表达式函数与其它惯用法一起使用能简化代码，例如和 when 表达式一起使用：

fun transform(color: String): Int = when (color) {

"Red" -> 0

"Green" -> 1

"Blue" -> 2

else -> throw IllegalArgumentException("Invalid color param value")

}

### 3.27 对一个对象实例调用多个方法 （with）

没有实现Builder的类，可以使用类于Builder的写法，让程序更加简洁。

class Turtle {

fun penDown()

fun penUp()

fun turn(degrees: Double)

fun forward(pixels: Double)

}

val myTurtle = Turtle()

with(myTurtle) { // 画一个 100 像素的正方形

penDown()

for (i in 1..4) {

forward(100.0)

turn(90.0)

}

penUp()

}

### 3.28 配置对象的属性（apply）

val myRectangle = Rectangle().apply { //直接对参数赋值

length = 4

breadth = 5

color = 0xFAFAFA

}

这对于配置未出现在对象构造函数中的属性非常有用。

### 3.29 对于需要泛型信息的泛型函数的适宜形式

inline fun <reified T: Any> Gson.fromJson(json: JsonElement): T = this.fromJson(json, T::class.java)

Reified：这个是为了满足inline特性而设计的语法糖，因为给函数使用内联之后，编译器会用其函数体来替换掉函数调用，而如果该函数里面有泛型就可能会出现编译器不懂该泛型的问题，所以引入reified，使该泛型被智能替换成对应的类型

### 3.30 使用可空布尔

val b: Boolean? = ……

if (b == true) {

……

} else {

// `b` 是 false 或者 null

}

### 3.31 交换两个变量

Also代表做这个操作的同时，也做另外一个操作。

var a = 1

var b = 2

a = b.also { b = a }

### 3.32 TODO()：将代码标记为不完整

Kotlin 的标准库有一个 TODO() 函数，该函数总是抛出一个 NotImplementedError。 其返回类型为 Nothing，因此无论预期类型是什么都可以使用它。 还有一个接受原因参数的重载：

fun calcTaxes(): BigDecimal = TODO("Waiting for feedback from accounting")

IntelliJ IDEA 的 kotlin 插件理解 TODO() 的语言，并且会自动在 TODO 工具窗口中添加代码指示。

## 四、基本类型

### 4.1 数字

Kotlin 提供了一组表示数字的内置类型。 对于整数，有四种不同大小的类型，因此值的范围也不同。

| 类型 | 大小（比特数） | 最小值 | 最大值 |
| --- | --- | --- | --- |
| Byte | 8 | -128 | 127 |
| Short | 16 | -32768 | 32767 |
| Int | 32 | -2,147,483,648 (-231) | 2,147,483,647 (231 - 1) |
| Long | 64 | -9,223,372,036,854,775,808 (-263) | 9,223,372,036,854,775,807 (263 - 1) |

**所有以未超出 Int 最大值的整型值初始化的变量都会推断为 Int 类型**。如果初始值超过了其最大值，那么推断为 Long 类型。 如需显式指定 Long 型值，请在该值后追加 L 后缀。

val one = 1 // Int

val threeBillion = 3000000000 // Long

val oneLong = 1L // Long

val oneByte: Byte = 1

对于浮点数，Kotlin 提供了 Float 与 Double 类型。 根据 [IEEE 754 标准](https://zh.wikipedia.org/wiki/IEEE_754)， 两种浮点类型的十进制位数（即可以存储多少位十进制数）不同。 Float 反映了 IEEE 754 单精度，而 Double 提供了双精度。

| 类型 | 大小（比特数） | 有效数字比特数 | 指数比特数 | 十进制位数 |
| --- | --- | --- | --- | --- |
| Float | 32 | 24 | 8 | 6-7 |
| Double | 64 | 53 | 11 | 15-16 |

对于以小数初始化的变量，编译器会推断为 Double 类型。 如需将一个值显式指定为 Float 类型，请添加 f 或 F 后缀。 如果这样的值包含多于 6～7 位十进制数，那么会将其舍入。

val pi = 3.14 // Double

val e = 2.7182818284 // Double

val eFloat = 2.7182818284f // Float，实际值为 2.7182817

请注意，与一些其他语言不同，Kotlin 中的数字没有隐式拓宽转换。 **例如，具有 Double 参数的函数只能对 Double 值调用，而不能对 Float、 Int 或者其他数字值调用**。

fun printDouble(d: Double) { print(d) }

val i = 1

val d = 1.1

val f = 1.1f

printDouble(d)

// printDouble(i) // 错误：类型不匹配

// printDouble(f) // 错误：类型不匹配

### 4.2 字面常量

数值常量字面值有以下几种:

十进制: 123

Long 类型用大写 L 标记: 123L

十六进制: 0x0F

二进制: 0b00001011

**注意: 不支持八进制**

Kotlin 同样支持浮点数的常规表示方法:

默认 double：123.5、123.5e10

Float 用 f 或者 F 标记: 123.5f

### 4.2 数字字面值中的下划线（自 1.1 起）

你可以使用下划线使数字常量更易读：

val oneMillion = 1\_000\_000

val creditCardNumber = 1234\_5678\_9012\_3456L

val socialSecurityNumber = 999\_99\_9999L

val hexBytes = 0xFF\_EC\_DE\_5E

val bytes = 0b11010010\_01101001\_10010100\_10010010

### 4.3 表示方式

在 Java 平台数字是物理存储为 JVM 的原生类型，除非我们需要一个可空的引用（如 Int?）或泛型。 后者情况下会把数字装箱。

注意数字装箱不一定保留同一性:

val a: Int = 100

val boxedA: Int? = a

val anotherBoxedA: Int? = a

val b: Int = 10000

val boxedB: Int? = b

val anotherBoxedB: Int? = b

println(boxedA === anotherBoxedA) // true

println(boxedB === anotherBoxedB) // false

另一方面，它保留了相等性:

val a: Int = 10000

val boxedA: Int? = a

val anotherBoxedA: Int? = a

println(boxedA == anotherBoxedA) // 输出“true”

### 4.4 显式转换

由于不同的表示方式，较小类型并不是较大类型的子类型。 如果它们是的话，就会出现下述问题：

// 假想的代码，实际上并不能编译：

val a: Int? = 1 // 一个装箱的 Int (java.lang.Integer)

val b: Long? = a // 隐式转换产生一个装箱的 Long (java.lang.Long)

print(b == a) // 惊！这将输出“false”鉴于 Long 的 equals() 会检测另一个是否也为 Long

所以相等性会在所有地方悄无声息地失去，更别说同一性了。

因此较小的类型不能隐式转换为较大的类型。 这意味着在不进行显式转换的情况下我们不能把 Byte 型值赋给一个 Int 变量。

val b: Byte = 1 // OK, 字面值是静态检测的

val i: Int = b // 错误

我们可以显式转换来拓宽数字

**val i: Int = b.toInt() // OK：显式拓宽**

print(i)

**每个数字类型支持如下的转换:**

toByte(): Byte

toShort(): Short

toInt(): Int

toLong(): Long

toFloat(): Float

toDouble(): Double

toChar(): Char

缺乏隐式类型转换很少会引起注意，因为类型会从上下文推断出来，而算术运算会有重载做适当转换，例如：

val l = 1L + 3 // Long + Int => Long