

KOTLIN 快速参考

空安全、协程、数据类、函数式编程要点

基础

Hello World

```
fun main() {
    println("Hello, World!")
}
```

变量

```
val name = "Kotlin" // immutable (prefer)
var count = 0 // mutable
val pi: Double = 3.14159 // explicit type
const val MAX = 100 // compile-time constant
```

基本类型

Int, Long 32 位 / 64 位有符号整数
Double, Float 64 位 / 32 位浮点数
Boolean true / false
Char 单个 Unicode 字符
String 不可变文本, 支持字符串模板
Unit 等同于 `void` (单一值)
Nothing 函数永不返回 (如抛出异常)

字符串模板

```
val name = "World"
println("Hello, $name!")
println("Length: ${name.length}")
val raw = """line 1
           |line 2"""
println(raw)
```

函数

函数声明

```
fun add(a: Int, b: Int): Int {
    return a + b
}
fun add(a: Int, b: Int) = a + b // single expression
```

默认参数与具名参数

```
fun greet(name: String, greeting: String = "Hello") {
    println("$greeting, $name!")
}
greet("Alice") // Hello, Alice!
greet("Bob", greeting = "Hi") // Hi, Bob!
```

高阶函数

```
fun operate(a: Int, b: Int, op: (Int, Int) -> Int): Int {
    return op(a, b)
}
val sum = operate(3, 4) { a, b -> a + b }
```

可变参数

```
fun sum(vararg nums: Int): Int = nums.sum()
sum(1, 2, 3)
val arr = intArrayOf(1, 2, 3)
sum(*arr) // spread operator
```

类

类定义

```
class Person(val name: String, var age: Int) {
    fun greet() = "Hi, I'm $name"
}
val p = Person("Alice", 30)
println(p.name)
```

继承

```
open class Shape(val sides: Int) { open fun area(): Double = 0.0 }
class Circle(val r: Double) : Shape(0) {
    override fun area() = Math.PI * r * r
}
```

可见性修饰符

public 任何地方可见 (默认)
private 仅在类/文件内可见
protected 类及其子类可见
internal 仅同一模块内可见

抽象类与接口

```
interface Drawable { fun draw() }
abstract class Widget : Drawable { abstract val label: String }
class Button(override val label: String) : Widget() {
    override fun draw() = println("Drawing $label")
}
```

空安全

可空类型

```
var name: String? = null // nullable
val len = name?.length // safe call: null
val len2 = name?.length ?: 0 // Elvis operator: 0
val len3 = name!!.length // assert non-null (throws)
```

安全操作符

?. 安全调用—接收者为 null 时返回 null
?: Elvis—null 时使用默认值
!! 非空断言 (为 null 时抛出异常)
?..let { } 仅在非 null 时执行代码块
as? 安全转型—失败时返回 null

智能转型

```
if (obj is String) println(obj.length) // auto-cast
when (obj) {
    is Int -> println(obj + 1)
    is String -> println(obj.uppercase())
}
```

集合

创建集合

```
val list = listOf(1, 2, 3) // immutable
val mList = mutableListOf(1, 2, 3) // mutable
val map = mapOf("a" to 1, "b" to 2)
val set = setOf("x", "y", "z")
```

集合操作

```
val nums = listOf(1, 2, 3, 4, 5)
nums.filter { it > 2 } // [3, 4, 5]
nums.map { it * 2 } // [2, 4, 6, 8, 10]
nums.firstOrNull { it > 3 } // 4
nums.sumOf { it } // 15
```

常用操作

.filter { } 保留满足谓词的元素
.map { } 转换每个元素
.flatMap { } 映射并展平
.groupBy { } 按键分组为 Map
.sortedBy { } 按选择器排序
.associate { } 转换为 Map (键值对)
.any { } / **.all { }** 检查是否有任何/全部元素满足谓词
.fold (init) { } 带初始值的归约

协程

基础协程

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    launch { delay(1000); println("World") }
    println("Hello")
}
```

Async / Await

```
val deferred = async { fetchData() }
val result = deferred.await()
// parallel: launch multiple async, await all
val (a, b) = awaitAll(async { fetchA() }, async { fetchB() })
```

协程构建器

launch { } 即发即忘协程 (返回 Job)
async { } 返回带结果的 Deferred<T>
runBlocking { } 桥接阻塞与挂起代码
withContext (dispatcher) 切换协程上下文
coroutineScope { } 结构化并发作用域

调度器

Dispatchers.Default CPU 密集型任务 (线程池)
Dispatchers.IO 阻塞 I/O 操作
Dispatchers.Main 主线程/UI 线程 (Android, Swing)
Dispatchers.Unconfined 在调用线程启动, 在任意线程恢复

扩展

扩展函数

```
fun String.isPalindrome(): Boolean {
    return this == this.reversed()
}
println("racecar".isPalindrome()) // true
```

扩展属性

```
val String.wordCount: Int
get() = this.split("\\s+").toRegex().size
println("hello world".wordCount) // 2
```

运算符重载

```
data class Vec(val x: Double, val y: Double) {
    operator fun plus(other: Vec) = Vec(x + other.x, y + other.y)
}
val v = Vec(1.0, 2.0) + Vec(3.0, 4.0) // Vec(4.0, 6.0)
```

数据类

数据类

```
data class User(val name: String, val age: Int)
val u1 = User("Alice", 30)
val u2 = u1.copy(age = 31) // non-destructive copy
val (name, age) = u1 // destructuring
```

自动生成的成员

equals() 基于属性的结构相等
hashCode() 与 `equals()` 一致
toString() `User(name=Alice, age=30)`
copy() 创建修改后的副本
componentN() 解构支持

枚举类

```
enum class Direction { NORTH, SOUTH, EAST, WEST }
val dir = Direction.NORTH
when (dir) { Direction.NORTH -> "up"; else -> "other" }
```

密封类

密封类层级

```
sealed class Result<out T> {
    data class Success<T>(val data: T) : Result<T>()
    data class Error(val message: String) : Result<Nothing>()
    data object Loading : Result<Nothing>()
}
```

穷举 When

```
fun handle(result: Result<String>): String = when (result) {
    is Result.Success -> result.data
    is Result.Error -> "Error: ${result.message}"
    is Result.Loading -> "Loading..."
} // no else needed - compiler checks exhaustiveness
```

密封类 vs 枚举

Sealed class 子类可持有不同状态
Sealed interface 允许多重继承
Enum class 固定的单例实例集合
data object 带 `toString()` 重写的单例

作用域函数

作用域函数对比

let 上下文为 `it`, 返回 lambda 结果
run 上下文为 `this`, 返回 lambda 结果
with (obj) 上下文为 `this`, 返回 lambda 结果
apply 上下文为 `this`, 返回上下文对象
also 上下文为 `it`, 返回上下文对象

let 与 apply

```
val name: String? = "Alice"
name?.let { println("Name is $it") }
val person = Person("Bob", 25).apply {
    age = 26 // configure object
}
```

run 与 with

```
val result = "Hello".run { uppercase() + " WORLD" }
val info = with(person) { "$name is $age years old" }
```

also

```
val numbers = mutableListOf(1, 2, 3)
.also { println("Original: $it") }
// also is useful for side effects (logging, validation)
```