目录

[1、Init{} 2](#_Toc25703)

[2、一、双冒号：： 2](#_Toc31341)

[！！ 3](#_Toc23878)

[返回值：Unit、Nothing、any 3](#_Toc24170)

[LaunchActivity::class.java 3](#_Toc21670)

[Kotlin 作用域函数let run apply also with 3](#_Toc24815)

[由参数函数的返回值决定 4](#_Toc6988)

["${a\_et.text}"串模版 4](#_Toc2555)

[Kotlin 类和构造参数的创建 4](#_Toc29255)

[Kotlin集合类型 6](#_Toc13128)

[Kotlin object 6](#_Toc32200)

[？ ：符号 6](#_Toc17461)

[@JvmOverloads 7](#_Toc21460)

[@JvmField 7](#_Toc15921)

[@JvmName、@JvmOverloads、@JvmStatic 7](#_Toc11679)

[const val 和val的区别 7](#_Toc9525)

[泛型实化 8](#_Toc14469)

[Operator 8](#_Toc27981)

[expect &actual 8](#_Toc11261)

[同步调用和异步调用的区别 8](#_Toc7498)

[协程———— 11](#_Toc13308)

[协程的优势特点： 11](#_Toc6153)

[协程的基础 11](#_Toc31652)

[协程的取消 13](#_Toc18683)

[超时 15](#_Toc16946)

[组合挂起函数 15](#_Toc5576)

[一、 默认顺序 15](#_Toc29828)

[二、使用 async 并发 15](#_Toc17995)

[三、惰性启动 async 16](#_Toc22450)

[四、异步风格的函数 17](#_Toc23869)

[五、使用 async 的结构化并发 19](#_Toc30092)

[协程上下文和调度器 21](#_Toc28475)

[一、调度器和线程 21](#_Toc16901)

[二、Unconfined vs confined dispatcher 22](#_Toc13079)

# 1、Init{}

是主构造函数的函数体

# 2、一、双冒号：：

1 [Kotlin](https://so.csdn.net/so/search?q=Kotlin&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/lv_fq/article/details/_blank) 中 双冒号操作符 表示把一个方法当做一个参数，传递到另一个方法中进行使用，通俗的来讲就是引用一个方法。

fun main(args: Array<String>) {

println(lock("param1", "param2", ::getResult))

}

/\*\*

\* @param str1 参数1

\* @param str2 参数2

\*/

fun getResult(str1: String, str2: String): String = "result is {$str1 , $str2}"

/\*\*

\* @param p1 参数1

\* @param p2 参数2

\* @param method 方法名称

\*/

fun lock(p1: String, p2: String, method: (str1: String, str2: String) -> String): String {

return method(p1, p2)

}

Lock函数第3个参数需要一个lambda表达式，所以在调用时把getResult当作函数传入，注意要确定参数个数、类型、返回值都和其形参一致。

2 如果我们需要调用其他 Class 中的某一个方法时：

var d = Test()

println(lock("param1", "param2", d::getResult)

先实例化类再：：方法

3 为了防止作用域混淆 ， :: 调用的函数如果是类的成员函数或者是扩展函数，必须使用限定符,比如this

注：main方法不是成员函数

# ！！

!! 加在变量之后，如果对象为空，则会报异常。等同于java。

# 返回值：Unit、Nothing、any

Unit

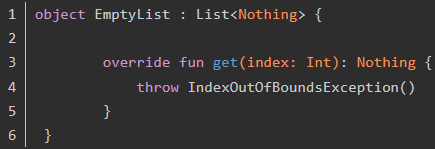
[Kotlin](https://so.csdn.net/so/search?q=Kotlin&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Jason_Lee155/article/details/_blank)也是面向表达的语言。Kotlin中Unit类型实现了与java中void一样的功能。当一个函数没有返回值的时候，我们用Unit来表示这个特征，而不是null，大多数时候我们不需要显示地返回Unit，或者声明一个函数的返回值是Unit，编译器会推断它。

Nothing

如果一个函数返回值是Nothing，那么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get永远不会反回值，这个时候可以用Nothing作为get函数的返回值。



Nothing?唯一允许的值是null，可被用作任何可空类型的空引用。

Any

**Any类型就跟Java的object类型一样，是所有类的父类。**

# LaunchActivity::class.java

获取类的Kclass之后再获取类的class

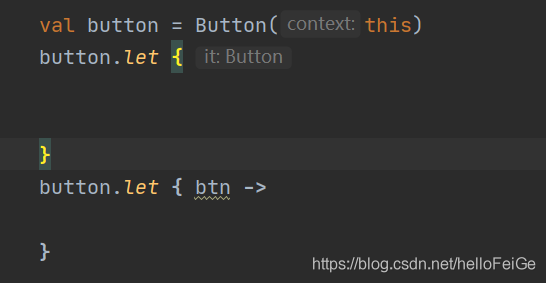
# Kotlin 作用域函数let run apply also with

当需要去定义一个变量在一个特定的作用域范围内时，就可以使用作用域函数



let把调用者T作为一个参数(默认名it可修改（见图）)传递到参数函数，返回值由参数函数的返回值决定

also把调用者T作为一个参数(默认名it可修改)传递到参数函数,返回值：this，也就是谁调用就返回谁



run是实例所在类的一个扩展函数（可用this表示调用者），无参数，返回值

由参数函数的返回值决定

apply是实例所在类的一个扩展函数（可用this表示调用者），无参数，返回值返回值：this，也就是谁调用就返回谁

# "${a\_et.text}"串模版

**串可以包含模板表达式，即：可计算的代码片段，其结果链接到串中。模板表达式以美元符号（$）开始，和简单的名字构成：**

**val i = 10**

**val s = "i = $i" // 计算结果是 "i = 10"**

或是在大括号中的任意表达式：

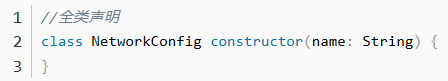
val s = "abc"

val str = "$s.length is ${s.length}" // 计算结果是 "abc.length is 3"

# Kotlin 类和构造参数的创建

kotlin中类是由class声明,kotlin的类声明有三部分, 类名,类头(指定类型参数,主构造函数),类体(花括号),类头和类体都是可选

kotlin可以拥有一个**主构造函数**和多个**次构造函数**,主构造函数是类头的一部分：它跟在类名（与可选的类型参数）后.

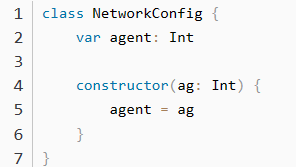


如果主构造函数没有任何注解和可见性修饰符,可以省略constructor

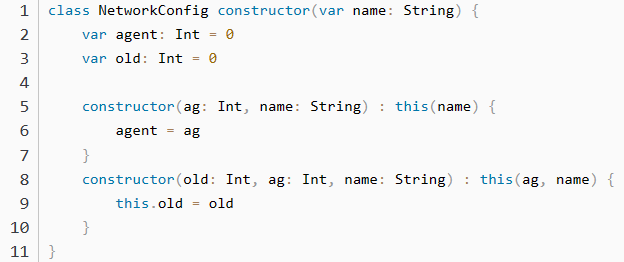


如果没有声明构造参数的类型,就是默认的val,不可修改,如果想要修改得声明var name String

可以声明由constructor的次构造函数:



如果有主构造函数,那么所有的次构造函数需要直接或者间接委托给主构造函数,  
委托到同一个类的另一个构造函数用 this 关键字即可:



* 第一执行的是主构造参数,
* 其次就是属性赋值,
* 第三就是init标识的类的属性初始化器,
* 第四就是次构造参数  
  同时因为对与this关键字的执行顺序也和java中的super一样,先执行this委托的函数

创建类



# Kotlin集合类型

Kotlin集合类型可分为可变和不可变集合类型。List、Set、Map为不可变集合类型，MutableList、MutableSet、MutableMap为可变集合类型

# Kotlin object

和companion object 区别

* Object（懒加载单例类）声明（一个类）是延迟加载的，只有当第一次被访问时才会初始化，所以被用来实现单例
* companion object静态函数（伴生对象）是当包含它的类被加载时就初始化了的，这一点和Java的static还是一样的

当使用了对象表达式和object生成一个匿名对象时，如果用private修饰则会生成一个新的对象，在外部就是可访问的。但如果不使用范围限定符修饰则生成的是匿名对象，在外部就是不可访问的。

# ？ ：符号

i == 0 ? "现在" : WeatherTimeUtils.get24Hour(data.getDate()),

i等于0返回现在，否则返回后面的表达式

# @JvmOverloads

在有默认参数值的方法中使用@JvmOverloads注解，则Kotlin就会暴露多个重载方法。

# @JvmField

使Kotlin编译器不再对该字段生成getter/setter并将其作为公开字段

# @JvmName、@JvmOverloads、@JvmStatic

## kotlin 实现静态的方式

在 kotin 语言中其实没有 java [static](https://so.csdn.net/so/search?q=static&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/fitaotao/article/details/_blank) 的这个概念，基本都是用一个静态对象来模拟 class 的静态属性和方法，目前有4种实现方式：

* [companion object](https://blog.csdn.net/fitaotao/article/details/112763100?ops_request_misc=%7B%22request%5Fid%22%3A%22167807388216782425149658%22%2C%22scm%22%3A%2220140713.130102334..%22%7D&request_id=167807388216782425149658&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-112763100-null-null.142%5ev73%5epc_new_rank,201%5ev4%5eadd_ask,239%5ev2%5einsert_chatgpt&utm_term=kotlin%20%E9%9D%99%E6%80%81%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187) - 伴随对象，声明单例的方式（最常用）
* [@JvmField + @JvmStatic 注解](https://blog.csdn.net/fitaotao/article/details/112763100?ops_request_misc=%7B%22request%5Fid%22%3A%22167807388216782425149658%22%2C%22scm%22%3A%2220140713.130102334..%22%7D&request_id=167807388216782425149658&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-112763100-null-null.142%5ev73%5epc_new_rank,201%5ev4%5eadd_ask,239%5ev2%5einsert_chatgpt&utm_term=kotlin%20%E9%9D%99%E6%80%81%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187) - 使用注解标签声明 static 的部分
* [object 单例](https://blog.csdn.net/fitaotao/article/details/112763100?ops_request_misc=%7B%22request%5Fid%22%3A%22167807388216782425149658%22%2C%22scm%22%3A%2220140713.130102334..%22%7D&request_id=167807388216782425149658&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-112763100-null-null.142%5ev73%5epc_new_rank,201%5ev4%5eadd_ask,239%5ev2%5einsert_chatgpt&utm_term=kotlin%20%E9%9D%99%E6%80%81%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187) - 静态单例其实和 companion object 类似
* [const](https://blog.csdn.net/fitaotao/article/details/112763100?ops_request_misc=%7B%22request%5Fid%22%3A%22167807388216782425149658%22%2C%22scm%22%3A%2220140713.130102334..%22%7D&request_id=167807388216782425149658&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-112763100-null-null.142%5ev73%5epc_new_rank,201%5ev4%5eadd_ask,239%5ev2%5einsert_chatgpt&utm_term=kotlin%20%E9%9D%99%E6%80%81%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187) - 包内唯一性，脱离类的束缚，kotlin 的特性，在 java 中会编译生成一个 kotlin.kt 的文件专门对齐提供支持

# const val 和val的区别

const val相当于public final static

val相当于provite final static

前者可以直接调用，后者需要使用getter来调用

创建常量的时候尽量使用前者，避免频繁的函数调用。

# 泛型实化

两个前提：

1、使用inline修饰方法

2、使用reified修饰泛型

# Operator

重载运算符：比如+-\*/等，还见过getValue和setValue方法也用了的

# 委托by

private var age: Int by Delegates.observable(18) { property, oldValue, newValue ->

YYLogUtils.w("发生了回调 property:$property oldValue:$oldValue newValue：$newValue")

}

findViewById<Button>(R.id.btn\_load).click {

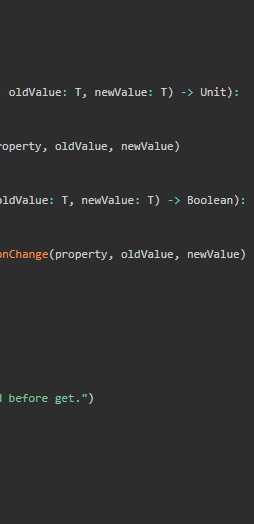
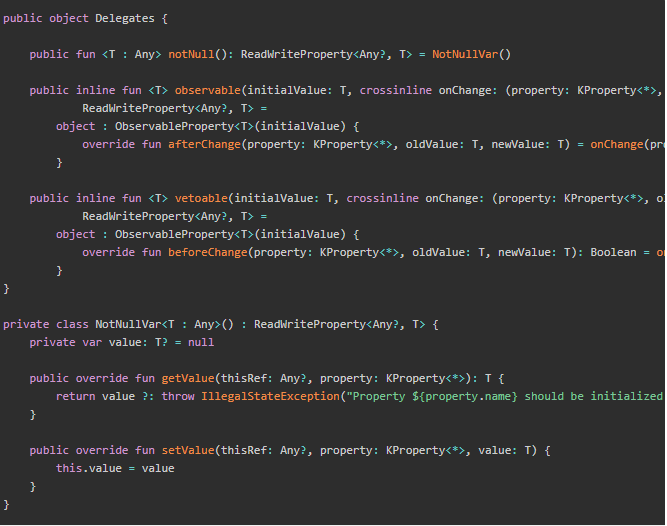
age = 25

YYLogUtils.w("name:$name age:$age")

}

我们通过 by Delegates 的方式就可以指定委托对象，这里我用的 Delegates.obsevable 它的作用是修改 age 的值之后会有回调的处理。

除了 Delegates.obsevable 它还有其他的用法。



notNull方法我们可以看到就是说这个对象不能为null，否则就会抛出异常。

observable方法主要用于监控属性值发生变更，类似于一个观察者。当属性值被修改后会往外部抛出一个变更的回调。

vetoable方法跟observable类似，都是用于监控属性值发生变更，当属性值被修改后会往外部抛出一个变更的回调。与observable不同的是这个回调会返回一个Boolean值，来决定此次属性值是否执行修改。

# 懒加载方式，by lazy 与 lateinit 的异同

## lateinit

见名知意，延时初始化的标记。lateinit var可以让我们声明一个变量并且不用马上初始化，在我们需要的时候进行手动初始化即可。

所以对应这一种情况我们会有一个是否初始化的判断

private lateinit var name: String

findViewById<Button>(R.id.btn\_load).click {

if (this::name.isInitialized) {

YYLogUtils.w("name:$name age:$age")

}

}

lateinit var的作用相对较简单，其实就是让编译期在检查时不要因为属性变量未被初始化而报错。（注意一定要记得初始化哦！）

## by lazy

by lazy 委托延时处理，分为委托和延时

如果我们想实现延时初始化的关键就是 lazy 关键字，所以，lazy是如何工作的呢？ 让我们一起在Kotlin标准库参考中总结lazy()方法，如下所示：



lazy() 返回的是一个存储在lambda初始化器中的Lazy类型实例。

getter的第一次调用执行传递给lazy()的lambda并存储其结果。

后面再调用的话，getter调用只返回存储中的值。

简单地说，lazy创建一个实例，在第一次访问属性值时执行初始化，存储结果并返回存储的值。

## 总结

总的来说其实 lateinit 是延迟初始化， by lazy 是懒加载即初始化方式已确定，只是在使用的时候执行。

虽然两者都可以推迟属性初始化的时间，但是 lateinit var 只是让编译期忽略对属性未初始化的检查，后续在哪里以及何时初始化还需要开发者自己决定。而by lazy真正做到了声明的同时也指定了延迟初始化时的行为，在属性被第一次被使用的时候能自动初始化。

并且 lateinit 是可读写的，by lazy 是只读的。

其实大部分情况下都可以通用，只是 by lazy 一般用于非空只读属性，需要延迟加载情况，而 lateinit 一般用于非空可变属性，需要延迟加载情况。

# expect &actual

actual 关键字通常与 expect 关键字配合使用，用于定义多平台通用的接口和函数，从而允许在不同的平台上使用相同的 API

expect class PlatformSpecificClass {

fun platformSpecificMethod(): String}

actual class PlatformSpecificClass {

actual fun platformSpecificMethod(): String {

return "Android"

}}

expect 和 actual 关键字只能在 Kotlin 的多平台项目中使用，用于实现跨平台开发。在一个普通的 Kotlin 项目中，这两个关键字并没有实际意义

# 同步调用和异步调用的区别

同步调用指的是主线程需要等待运行的结果之后才能继续运行，异步调用指的是主线程在执行时不需要等待结果可以去做别的事，而是等到运行结束了直接获取结果。

异步调用常用的方法：

1. 回调（Callback）：使用回调函数来处理异步调用的结果。通过定义回调接口，在异步任务完成后，调用相应的回调方法来处理结果。这种方式需要预先定义回调接口，适用于简单的异步任务。
2. Future 和 Callable：使用 Future 和 Callable 接口来实现异步调用。通过提交一个 Callable 对象到 ExecutorService，返回一个 Future 对象来表示异步任务的结果。可以通过 Future 对象来获取任务的执行状态和结果。
3. RxJava：RxJava 是一个流式编程库，提供了丰富的操作符和功能，用于处理异步和事件驱动的编程。可以使用 RxJava 来处理网络请求、异步操作等复杂的业务逻辑。
4. 协程（Coroutine）：协程是一种轻量级的线程管理技术，可以简化异步编程的复杂性。通过使用协程，可以以顺序的方式编写异步代码，看起来像同步调用一样简洁易懂。

同步调用：

Callable<StudentInfo> callable = new Callable<StudentInfo>() {

@Override

public StudentInfo call() throws Exception {

try {

//模拟耗时操作

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

return new StudentInfo();

}

};

public StudentInfo getStuInfo(long stuId) {

//定义任务

FutureTask<StudentInfo> futureTask = new FutureTask<>(callable);

//开启线程，执行任务

new Thread(futureTask).start();

try {

//阻塞获取结果

StudentInfo studentInfo = futureTask.get();

return studentInfo;

} catch (ExecutionException e) {

e.printStackTrace();

} catch (InterruptedException e) {

e.printStackTrace();

}

return null;

}

### 异步调用与回调

//回调接口

public interface Callback {

void onCallback(StudentInfo studentInfo);

}

//异步调用

public void getStuInfoAsync(long stuId, Callback callback) {

new Thread(() -> {

try {

//模拟耗时操作

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

StudentInfo studentInfo = new StudentInfo();

if (callback != null) {

//回调给调用者

callback.onCallback(studentInfo);

}

}).start();

}

JavaStudent javaStudent = new JavaStudent();

javaStudent.getStuInfoAsync(999, new JavaStudent.Callback() {

@Override

public void onCallback(StudentInfo studentInfo) {

//异步调用，回调从子线程返回，需要切换到主线程更新UI

runOnUiThread(() -> {

Toast.makeText(TestJavaActivity.this, "学生姓名:" + studentInfo.getName(), Toast.LENGTH\_LONG).show();

});

}

});

异步调用的好处显而易见：

1、不用阻塞调用者，调用者可继续做其它事情。

2、线程没有被阻塞，相比同步调用效率更高。

缺点也是比较明显：

1、没有同步调用直观。

2、容易陷入多层回调，不利于阅读与调试。

3、从内到外的异常处理缺失传递性。

# 协程————

# 协程的优势特点：

**将异步编程同步化**

**例：**

Java 又收到需求变更了：

通过学生id，获取学生信息，通过学生信息，获取他的语文老师id，通过语文老师id，获取老师姓名，最后更新UI。

通过老师id，获取他所在的教研组信息，再通过教研组id获取教研组排名...

如果想异步实现上述需求则需要回调嵌套回调再嵌套回调，很容易晕。但使用kotlin就不一样了，可以实现**将异步编程同步化。**

就是使用同步的方式做异步的事情。

具体的例子见如下：<https://juejin.cn/post/7108651566806073380#heading-7>

# 协程的基础

## coroutineScope

协程作用域

在它的上下文中通过 launch、async 等协程构造器（coroutine builder）来启动协程

## GlobalScope

全局作用域，该协程的生命周期只受整个应用程序的生命周期的限制，即只要整个应用程序还在运行中，只要协程的任务还未结束，该协程就可以一直运行。

它类似于守护线程，（当主线程消亡时，守护线程也将消亡）。

## runBlocking

直到 runBlocking 内部的所有协程执行完成后，之后的代码才会继续执行。runBlocking 代码块默认运行于其声明所在的线程，而 launch 代码块默认运行于线程池中

可以通过 runBlocking 将 main() 函数转为协程。

import kotlinx.coroutines.\*

fun main() = runBlocking { // this: CoroutineScope

launch { // launch a new coroutine in the scope of runBlocking

delay(1000L)

println("World!")

}

println("Hello,")

}

好处是因为外部协程在其作用域中启动的所有协程完成之前不会结束，所以不需要显示delay，它一定会运行完。且不需要害怕GlobalScope.launch一直运行导致占用内存。

runBlocking 并非一个挂起函数，但它可以用于挂起当前线程并执行挂起函数的协程。

## coroutineScope和runBlocking 的区别

runBlocking 方法会阻塞当前线程，而 coroutineScope 只是挂起并释放底层线程以供其它协程使用。所以runBlocking 是一个普通函数，而 coroutineScope 是一个挂起函数

## coroutineScope和runBlocking和GlobalScope的区别

coroutineScope和runBlocking会等待子线程执行完毕，而GlobalScope需要手动的使用async 和 await 或者其他手动的等待机制来管理和等待 GlobalScope 中的协程

## delay()

是一个挂起函数，只能由协程或者其它挂起函数进行调度，挂起函数不会阻塞线程，而是会将协程挂起，在特定的时候才再继续运行。当协程 A 调用 delay(1000L) 函数来指定延迟1秒后再运行时，协程 A 所在的线程只是会转而去执行协程 B，等到1秒后再把协程 A 加入到可调度队列里。线程并不会因为协程的延时而阻塞，但协程阻塞了。

## .join()

挂起当前协程等待调用的协程执行完毕再接下去执行

## Suspend fun

用于定义一个挂起函数、协程

## Async函数

在async函数中发起网络请求再调用await（）方法就可以实现只有在两个网络请求都成功响应后才会进一步执行程序。

另外由于只能在协程作用域中才能调用，所以写在coroutineScope内。

# 协程的取消

launch 函数的返回值 Job 对象就可用于取消正在运行的协程

job.cancel()

job.join()

import kotlinx.coroutines.\*

fun main() = runBlocking {

//sampleStart

val job = launch {

repeat(1000) { i ->

println("job: I'm sleeping $i ...")

delay(500L)

}

}

delay(1300L) // delay a bit

println("main: I'm tired of waiting!")

job.cancel() // cancels the job

job.join() // waits for job's completion

println("main: Now I can quit.")

//sampleEnd

}

因为 cancel() 函数调用后会马上返回而不是等待协程结束后再返回，为了确保协程执行结束后再执行后续代码，此时就需要调用 join() 方法来阻塞等待。

还有一个 Job 的扩展函数 cancelAndJoin ，它结合了 cancel 和 join 的调用。

public suspend fun Job.cancelAndJoin() {

cancel()

return join()

}

## 计算任务的取消

协程的取消操作是协作(cooperative)完成的，协程必须协作才能取消。kotlinx.coroutines 中的所有挂起函数都是可取消的，它们在运行时会检查协程是否被取消了，并在取消时抛出 CancellationException 。但是，如果协程正在执行计算任务，并且未检查是否已处于取消状态的话，则无法取消协程。

有两种方法可以使计算类型的代码可以被取消。第一种方法是定期调用一个挂起函数来检查取消操作，yieid() 函数是一个很好的选择。另一个方法是显示检查取消操作。使用isActive属性。isActive 是一个可通过 CoroutineScope 对象在协程内部使用的扩展属性

while (isActive) {

//计算任务

最后使用job.cancelAndJoin()

那样就能立马取消了，否则要等到条件结束（while (i<5) {

）才能取消。

**用 finally /use关闭资源**

可取消的挂起函数在取消时会抛出 CancellationException，可以用常用的方式来处理这种情况。例如，try {...} finally {...} 表达式和 kotlin 的 use 函数都可用于在取消协程时执行回收操作

file.bufferedReader().use { reader ->

val content = reader.readText()

println(content)

}  
use函数会自动释放资源

**在finally中调用挂起函数**

当需要在取消的协程中调用挂起函数时，可以使用 withContext 函数和 NonCancellable 上下文将相应的代码包装在 withContext(NonCancellable) {...} 代码块中

try {

repeat(1000) { i ->

println("job: I'm sleeping $i ...")

delay(500L)

}

} finally {

withContext(NonCancellable) {

println("job: I'm running finally")

delay(1000L)

println("job: And I've just delayed for 1 sec because I'm non-cancellable")

}

}

# **超时**

大多数情况下，我们会主动取消协程的原因是由于其执行时间已超出预估的最长时间。虽然我们可以手动跟踪对相应 Job 的引用，并在超时后取消 Job，但官方也提供了 withTimeout 函数来完成此类操作。

import kotlinx.coroutines.\*

fun main() = runBlocking {

//sampleStart

withTimeout(1300L) {

repeat(1000) { i ->

println("I'm sleeping $i ...")

delay(500L)

}

}

//sampleEnd

}

withTimeout 引发的 TimeoutCancellationException 是 CancellationException 的子类。之前我们从未在控制台上看过 CancellationException 这类异常的堆栈信息。这是因为对于一个已取消的协程来说，CancellationException 被认为是触发协程结束的正常原因。但是，在这个例子中，我们在主函数中使用了 withTimeout 函数，该函数会主动抛出 TimeoutCancellationException

你可以通过使用 try{...}catch（e:TimeoutCancellationException）{...} 代码块来对任何情况下的超时操作执行某些特定的附加操作，或者通过使用 withTimeoutOrNull 函数以便在超时时返回 null 而不是抛出异常

# 组合挂起函数

# 默认顺序

因为协程中的代码和常规代码一样，在默认情况下是顺序的执行的。

# 二、使用 async 并发

如果 doSomethingUsefulOne() 和 doSomethingUsefulTwo() 这两个函数之间没有依赖关系，并且我们希望通过同时执行这两个操作（并发）以便更快地得到答案，此时就需要用到 async 了

从概念上讲，async 就类似于 launch。async 启动一个单独的协程，这是一个与所有其它协程同时工作的轻量级协程。不同之处在于，launch 返回 Job 对象并且不携带任何运行结果值。而 async 返回一个轻量级非阻塞的 Deferred 对象，可用于在之后取出返回值，可以通过调用 Deferred 的 await() 方法来获取最终结果。此外，Deferred 也实现了 Job 接口，所以也可以根据需要来取消它

import kotlinx.coroutines.\*

import kotlin.system.\*

fun main() = runBlocking<Unit> {

//sampleStart

val time = measureTimeMillis {

val one = async { doSomethingUsefulOne() }

val two = async { doSomethingUsefulTwo() }

println("The answer is ${one.await() + two.await()}")

}

println("Completed in $time ms")

//sampleEnd

}

suspend fun doSomethingUsefulOne(): Int {

delay(1000L) // pretend we are doing something useful here

return 13

}

suspend fun doSomethingUsefulTwo(): Int {

delay(1000L) // pretend we are doing something useful here, too

return 29

}

运行耗时几乎是减半了，因为这两个协程是同时运行，总的耗时时间可以说是取决于耗时最长的任务。需要注意，协程的并发总是显式的

# 三、惰性启动 async

可选的，可以将 async 的 start 参数设置为 CoroutineStart.lazy 使其变为懒加载模式。在这种模式下，只有在主动调用 Deferred 的 await() 或者 start() 方法时才会启动协程。运行以下示例：

import kotlinx.coroutines.\*

import kotlin.system.\*

fun main() = runBlocking<Unit> {

//sampleStart

val time = measureTimeMillis {

val one = async(start = CoroutineStart.LAZY) { doSomethingUsefulOne() }

val two = async(start = CoroutineStart.LAZY) { doSomethingUsefulTwo() }

// some computation

one.start() // start the first one

two.start() // start the second one

println("The answer is ${one.await() + two.await()}")

}

println("Completed in $time ms")

//sampleEnd

}

suspend fun doSomethingUsefulOne(): Int {

delay(1000L) // pretend we are doing something useful here

return 13

}

suspend fun doSomethingUsefulTwo(): Int {

delay(1000L) // pretend we are doing something useful here, too

return 29

}

以上定义了两个协程，但没有像前面的例子那样直接执行，而是将控制权交给了开发者，由开发者通过调用 start() 函数来确切地开始执行。首先启动了协程 one，然后启动了协程 two，然后再等待协程运行结束

注意，如果只是在 println 中调用了 await() 而不首先调用 start() ,这将形成顺序行为，因为 await() 会启动协程并等待其完成，这不是 lazy 模式的预期结果。async(start=CoroutineStart.LAZY) 的用例是标准标准库中的 lazy 函数的替代品，用于在值的计算涉及挂起函数的情况下

# 四、异步风格的函数

我们可以定义异步风格的函数，使用带有显式 GlobalScope 引用的异步协程生成器来调用 doSomethingUsefulOne 和 doSomethingUsefulTwo 函数。用 “…Async” 后缀来命名这些函数，以此来强调它们用来启动异步计算，并且需要通过其返回的延迟值来获取结果

// The result type of somethingUsefulOneAsync is Deferred<Int>

fun somethingUsefulOneAsync() = GlobalScope.async {

doSomethingUsefulOne()

}

// The result type of somethingUsefulTwoAsync is Deferred<Int>

fun somethingUsefulTwoAsync() = GlobalScope.async {

doSomethingUsefulTwo()

}

注意，这些 xxxAsync 函数不是挂起函数，它们可以从任何地方调用。但是，调用这些函数意味着是要用异步形式来执行操作

以下示例展示了它们在协程之外的使用：

import kotlinx.coroutines.\*

import kotlin.system.\*

//sampleStart

// note that we don't have `runBlocking` to the right of `main` in this example

fun main() {

val time = measureTimeMillis {

// we can initiate async actions outside of a coroutine

val one = somethingUsefulOneAsync()

val two = somethingUsefulTwoAsync()

// but waiting for a result must involve either suspending or blocking.

// here we use `runBlocking { ... }` to block the main thread while waiting for the result

runBlocking {

println("The answer is ${one.await() + two.await()}")

}

}

println("Completed in $time ms")

}

//sampleEnd

fun somethingUsefulOneAsync() = GlobalScope.async {

doSomethingUsefulOne()

}

fun somethingUsefulTwoAsync() = GlobalScope.async {

doSomethingUsefulTwo()

}

suspend fun doSomethingUsefulOne(): Int {

delay(1000L) // pretend we are doing something useful here

return 13

}

suspend fun doSomethingUsefulTwo(): Int {

delay(1000L) // pretend we are doing something useful here, too

return 29

}

这里展示的带有异步函数的编程样式仅供说明，因为它是其它编程语言中的流行样式。强烈建议不要将此样式与 kotlin 协程一起使用，因为如下所述

想象一下，如果在 val one = somethingUsefulOneAsync() 和 one.await() 这两行代码之间存在逻辑错误，导致程序抛出异常，正在执行的操作也被中止，此时会发生什么情况？通常，全局的错误处理者可以捕获此异常，为开发人员记录并报告错误，但是程序可以继续执行其它操作。但是这里 somethingUsefulOneAsync() 函数仍然还在后台运行（因为其协程作用域是 GlobalScope），即使其启动者已经被中止了。这个问题不会在结构化并发中出现，如下一节所示

# 五、使用 async 的结构化并发

让我们以 Concurrent using async 章节为例，提取一个同时执行 doSomethingUsefulOne() 和 doSomethingUsefulTwo() 并返回其结果之和的函数。因为 async 函数被定义为 CoroutineScope 上的一个扩展函数，所以我们需要将它放在 CoroutineScope 中，这就是 coroutineScope 函数提供的功能：

suspend fun concurrentSum(): Int = coroutineScope {

val one = async { doSomethingUsefulOne() }

val two = async { doSomethingUsefulTwo() }

one.await() + two.await()

}

这样，如果 concurrentSum() 函数发生错误并引发异常，则在其作用域中启动的所有协程都将被取消

import kotlinx.coroutines.\*

import kotlin.system.\*

fun main() = runBlocking<Unit> {

//sampleStart

val time = measureTimeMillis {

println("The answer is ${concurrentSum()}")

}

println("Completed in $time ms")

//sampleEnd

}

suspend fun concurrentSum(): Int = coroutineScope {

val one = async { doSomethingUsefulOne() }

val two = async { doSomethingUsefulTwo() }

one.await() + two.await()

}

suspend fun doSomethingUsefulOne(): Int {

delay(1000L) // pretend we are doing something useful here

return 13

}

suspend fun doSomethingUsefulTwo(): Int {

delay(1000L) // pretend we are doing something useful here, too

return 29

}

从 main 函数的输出结果来看，两个操作仍然是同时执行的

**我的理解是：他们同时进行，但是会等待执行完毕之后再继续获取结果，所以时间是最迟结束的那个。**

The answer is 42

Completed in 1017 ms

取消操作始终通过协程的层次结构来进行传播

fun main() = runBlocking<Unit> {

try {

failedConcurrentSum()

} catch(e: ArithmeticException) {

println("Computation failed with ArithmeticException")

}

}

suspend fun failedConcurrentSum(): Int = coroutineScope {

val one = async<Int> {

try {

delay(Long.MAX\_VALUE) // Emulates very long computation

42

} finally {

println("First child was cancelled")

}

}

val two = async<Int> {

println("Second child throws an exception")

throw ArithmeticException()

}

one.await() + two.await()

}

需要注意协程 one 和正在等待的父级是如何在协程 two 失败时取消的

Second child throws an exception

First child was cancelled

Computation failed with ArithmeticException

# 协程上下文和调度器

协程总是在由 Kotlin 标准库中定义的 CoroutineContext 表示的某个上下文中执行

协程上下文包含多种子元素。主要的元素是协程作业（Job，我们之前见过），以及它的调度器（Dispatche，本节将介绍）

# 一、调度器和线程

协程上下文（coroutine context）包含一个协程调度器（参阅 CoroutineDispatcher），协程调度器 用于确定执行协程的目标载体，即运行于哪个线程，包含一个还是多个线程。协程调度器可以将协程的执行操作限制在特定线程上，也可以将其分派到线程池中，或者让它无限制地运行

所有协程构造器（如 launch 和 async）都接受一个可选参数，即 CoroutineContext ，该参数可用于显式指定要创建的协程和其它上下文元素所要使用的调度器

import kotlinx.coroutines.\*

fun main() = runBlocking<Unit> {

//sampleStart

launch { // context of the parent, main runBlocking coroutine

println("main runBlocking : I'm working in thread ${Thread.currentThread().name}")

}

launch(Dispatchers.Unconfined) { // not confined -- will work with main thread

println("Unconfined : I'm working in thread ${Thread.currentThread().name}")

}

launch(Dispatchers.Default) { // will get dispatched to DefaultDispatcher

println("Default : I'm working in thread ${Thread.currentThread().name}")

}

launch(newSingleThreadContext("MyOwnThread")) { // will get its own new thread

println("newSingleThreadContext: I'm working in thread ${Thread.currentThread().name}")

}

//sampleEnd

}

Unconfined : I'm working in thread main

Default : I'm working in thread DefaultDispatcher-worker-1

newSingleThreadContext: I'm working in thread MyOwnThread

main runBlocking : I'm working in thread main

当 launch {...} 在不带参数的情况下使用时，它从外部的协程作用域继承上下文和调度器。在本例中，它继承于在主线程中中运行的 runBlocking 协程的上下文

Dispatchers.Unconfined 是一个特殊的调度器，看起来似乎也在主线程中运行，但实际上它是一种不同的机制，稍后将进行解释

在 GlobalScope 中启动协程时默认使用的调度器是 Dispatchers.default，并使用共享的后台线程池，因此 launch(Dispatchers.default){...} 与 GlobalScope.launch{...} 是使用相同的调度器

newSingleThreadContext 用于为协程专门创建一个新的线程来运行。专用线程是非常昂贵的资源。在实际的应用程序中，它必须在不再需要时使用 close 函数释放掉，或者存储在顶级变量中以此实现在整个应用程序中重用

# 二、Unconfined vs confined dispatcher

Dispatchers.Unconfined 调度器在调用者线程中启动一个协程，但它仅仅只是运行到第一个挂起点。在挂起之后，它将恢复线程中的协程，该协程完全由调用的挂起函数决定。Unconfined 调度器适用于既不消耗CPU时间和不更新任何受限于特定线程的共享数据（如UI）的协程

另一方面，调度器是默认继承于外部的协程作用域的。尤其是 runBlocking 启动的协程的调度器只能是调用者所在的线程，因此继承 runBlocking 的结果是在此线程上的调度执行操作是可预测的 FIFO

import kotlinx.coroutines.\*

fun main() = runBlocking<Unit> {

//sampleStart

launch(Dispatchers.Unconfined) { // not confined -- will work with main thread

println("Unconfined : I'm working in thread ${Thread.currentThread().name}")

delay(500)

println("Unconfined : After delay in thread ${Thread.currentThread().name}")

}

launch { // context of the parent, main runBlocking coroutine

println("main runBlocking: I'm working in thread ${Thread.currentThread().name}")

delay(1000)

println("main runBlocking: After delay in thread ${Thread.currentThread().name}")

}

//sampleEnd

}

Unconfined : I'm working in thread main

main runBlocking: I'm working in thread main

Unconfined : After delay in thread kotlinx.coroutines.DefaultExecutor

main runBlocking: After delay in thread main

因此，从 runBlocking{...} 继承了上下文的协程继续在主线程中执行，而调度器是 unconfined 的协程，在 delay 函数之后的代码则默认运行于 delay 函数所使用的运行线程

unconfined 调度器是一种高级机制，可以在某些极端情况下提供帮助而不需要调度协程以便稍后执行或产生不希望的副作用， 因为某些操作必须立即在协程中执行。 非受限调度器不应该在一般的代码中使用