**DART 學習筆記**

變數、常數、基本類型、運運算元、條件判斷以及類型轉換 2

迴圈運算式、List、Record、Set及Map的常用屬性和方法 7

List的高級用法 11

Records 記錄類型(元組) 表示和使用方法 14

Set：表示及使用 19

Map 的表示和使用方法 22

自訂義資料型態 26

forEach 的高級用法 29

switch/case 的高級用法 31

for loop 的高級用法 36

while loop 的高級用法 38

枚舉類型 40

函數 的高級用法 45

可變參數函示 50

匿名函數 54

回調函數 的高級用法 56

Function 的高級用法 60

filter **函數 62**

reduce 函數 64

fold 函數 66

where 函數 68

sort 函數 70

Future 的高級用法 80

異步程式設計 的用法 83

Duration 的高級用法 86

類別 9

類的高級用法 11

連結 python , c 程式 36

mixin 的高級用法 59

abstract 的高級用法 64

extends 的高級用法 70

factory 的高級用法 74

build 的高級用法 78

**變數、常數、基本類型、運運算元、條件判斷以及類型轉換**

**變數**

如前言所說，**Dart**和**Kotlin**一樣是強大的腳本類語言，可以不預先定義變數類型 ，自動會類型推倒，**Dart**中定義變數可以通過var關鍵字可以通過類型來申明變數：

var str = 'dart';

String str2 = 'this is dart';

int count = 123;

注意： var 後就不要寫類型 ， 寫了類型 不要var 兩者都寫 var a int = 5; 報錯

**常數**：final 和 const修飾符

* const修飾的值不變 要在定義變數的時候就得賦值；
* final可以開始不賦值 只能賦一次，而final不僅有const的編譯時常量的特性，最重要的它是運行時常量，並且final是惰性初始化，即在運行時第一次使用前才初始化。

final name = 'Max';

final String sex = '男';

const bar = 1000000;

const double atm = 1.01325 \* bar;

如果我們使用了阿里的代碼規範外掛插件，其實他會提示我們最好用const代替final。

#### Dart的命名規則

* 變數名稱必須由數位、字母、底線和美元符($)組成。
* 注意：識別字開頭不能是數字
* 識別字不能是保留字和關鍵字。
* 變數的名字是區分大小寫的如: age和Age是不同的變數。在實際的運用中,也建議,不要用一個單詞大小寫區分兩個變數。
* 識別字(變數名稱)一定要見名思意 ：變數名稱建議用名詞，方法名稱建議用動詞

#### Dart的入口方法

**Dart** 入口方法main有兩種定義

//表示main方法沒有返回值

void main(){

print('dart');

}

main(){

print('dart');

}

**基本類型**

#### 資料類型

**Dart**中常用的資料類型有以下的類型：

Numbers（數值）:

int

double

Strings（字串）

String

Booleans(布林)

bool

List（陣列）

在Dart中，陣列是清單物件，所以大多數人只是稱它們為列表

Maps（字典）

通常來說，Map 是一個鍵值對相關的物件。 鍵和值可以是任何類型的物件。每個鍵只出現一次，而一個值則可以出現多次

#### 數數值型別： int double

int整型：

int a=123;

a=45;

double既可以是整型，也可是浮點型：

double b=23.5;

b=24;

#### 字串類型

字串定義：

var str1='this is str1';

String str2='this is str2';

字串拼接：

print("$str1 $str2");

print(str1 + str2);

#### 布林類型

定義方式：

bool flag1=true;

var flag2=true;

判斷條件上和Kotlin使用無異。

#### List（陣列/集合）

不指定類型定義List：

var list1 = ["張三",20,true];

print(list1);

print(list1[2]);

這就有點顛覆我們以往的觀念了，一個list裡面還可以有不同的類型。

指定類型定義List：

var list2 = <String>["張三","李四"];

print(list2);

通過[]來定義Lsit：

通過[]創建的集合的容量可以變化：

var list = [];

list.add("小明");

list.add(24);

list.add(true);

print(list);

也可以指定List中的元素類型：

List<String> list = [];

又或者是：

List<String> list = List.empty(growable: true);

growable 為 false 是為 固定長度列表，為 true 是為 長度可變列表

通過List.filled創建的集合長度是固定：

var list1 = List.filled(2, "");

var list2 = List<String>.filled(2, "");

#### Map

Map的定義：

直接賦值方式：

var person={

"name":"小明",

"age":28,

"work":["程式師","Android開發"]

};

print(person["name"]);

print(person["age"]);

print(person["work"]);

通過Map分別賦值：

var map =new Map();

map["name"]="小明";

map["age"]=26;

map["work"]=["程式師","Android開發"];

print(map);

#### is 關鍵字來判斷類型

var str = 123;

if(str is String){

print('是string類型');

}else if(str is int){

print('int');

}else{

print('其他類型');

}

**運運算元**

#### 算術運算子

使用和符號上和Kotlin中的基本無異：

int a=13;

int b=5;

print(a+b); //加

print(a-b); //減

print(a\*b); //乘

print(a/b); //除

print(a%b); //其餘

print(a~/b); //取整

#### 關係運算子

關係運算子主要有：

== ！= > < >= <=

使用：

int a=5;

int b=3;

print(a==b); //判斷是否相等

print(a!=b); //判斷是否不等

print(a>b); //判斷是否大於

print(a<b); //判斷是否小於

print(a>=b); //判斷是否大於等於

print(a<=b); //判斷是否小於等於

#### 邏輯運算子

**! 取反：**

bool flag=false;

print(!flag); //取反

**&&並且:全部為true的話值為true 否則值為false：**

bool a=true;

bool b=true;

print(a && b);

**||或者：全為false的話值為false 否則值為true：**

bool a=false;

bool b=false;

print(a || b);

#### 設定運算子

**基礎設定運算子 =、??= ++ --**

int c=a+b; //從右向左

b??=23; 表示如果b為空的話把 23賦值給b

**++ --**

// ++ -- 表示自增 自減 1

//在賦值運算裡面 如果++ -- 寫在前面 這時候先運算 再賦值，如果++ --寫在後面 先賦值後運行運算

var a = 10;

var b = a--;

print(a); //9

print(b); //10

// var a=10;

// a++; //a=a+1;

// print(a);

**複合設定運算子 +=、-= 、\*= 、 /= 、%= 、~/=**

**+=**

var a=12;

a+=12; //a = a+12

print(a);

**-=**

a-=6; // a = a-6

\***=**

a\*=3; //a=a\*3;

**/=**

需要返回double類型

double a=12;

a/=12;

**%=**

double a=12;

a %= 12;

**~/=**

返回的是int整型

int a1 = 3;

int a2 = 2;

int a = a1 ~/= a2;

a = 1.

**條件判斷**

\*\*if else \*\*

bool flag=true;

if(flag){

print('true');

}else{

print('false');

}

**switch case**

var sex = "女";

switch (sex) {

case "男":

print('性別是男');

break;

case "女":

print('性別是女');

break;

default:

print('傳入參數錯誤');

break;

}

**三元運算子**

bool flag = false;

String str = flag?'我是true':'我是false';

print(str);

**??運算子**

var a;

var b= a ?? 10;

print(b); // a為空，則賦值為10

// var a=22;

// var b= a ?? 10;

//

// print(b); // 20

**類型轉換**

#### Number與String類型之間的轉換

* Number類型轉換成String類型toString()
* String類型轉成Number類型int.parse()

String轉Number

String str = '123';

var myNum = int.parse(str);

print(myNum is int);

// String str='123.1';

// var myNum=double.parse(str);

// print(myNum is double);

轉String:

var myNum=12;

var str=myNum.toString();

print(str is String);

#### 其他類型轉換成Boolean類型

isEmpty:判斷字串是否為空

var str = '';

if (str.isEmpty) {

print('str空');

} else {

print('str不為空');

}

isNaN：判斷值是否為非數字

var myNum = 0 / 0;

if (myNum.isNaN) {

print('NaN');

}

# 迴圈運算式、List、Record、Set及Map的常用屬性和方法

**迴圈運算式**

#### for迴圈

for (int i = 1; i<=100; i++) {

print(i);

}

也可以寫成：

for (var i = 1; i<=10; i++) {

print(i);

}

對於List的遍歷我們可以這樣做：

var list = <String>["張三","李四","王五"];

for (var element in list) {

print(element);

}

對於Map的反覆運算我們也可以使用for迴圈語句：

var person={

"name":"小明",

"age":28,

"work":["程式師","Android開發"]

};

person.forEach((key, value) {

print(value);

});

#### while語句

while有兩種語句格式：

while(運算式/迴圈條件){

}

do{

語句/循環體

}while(運算式/迴圈條件);

注意：

* 1、最後的分號不要忘記
* 2、迴圈條件中使用的變數需要經過初始化
* 3、循環體中，應有結束迴圈的條件，否則會造成閉環。

看下面代碼：

int i = 1;

while (i <= 10) {

print(i);

i++;

}

do...while()最大的區別就是不管條件成立與否都會至少執行一次：

var i = 2;

do{

print('執行代碼');

}while(i < 2);

#### break和continue語句

**break語句功能:**

* 在switch語句中使流程跳出switch結構。
* 在迴圈語句中使流程跳出當前迴圈,遇到break迴圈終止，後面代碼也不會執行 需要強調的是:
* 如果在迴圈中已經執行了break語句,就不會執行循環體中位於break後的語句。
* 在多層迴圈中,一個break語句只能向外跳出一層

break可以用在switch case中 也可以用在for迴圈和while迴圈中。

**continue語句的功能:**

只能在迴圈語句中使用，使本次迴圈結束，即跳過循環體中下面尚未執行的語句，接著進行下次的是否執行迴圈的判斷。

continue可以用在for迴圈以及while迴圈中，但是不建議用在while迴圈中，不小心容易閉環。

**break使用：**

//如果 i等於4的話跳出迴圈

for(var i=1;i<=10;i++){

if(i==4){

break; /\*跳出循環體\*/

}

print(i);

}

//break語句只能向外跳出一層

for(var i = 0;i < 5;i++){

for(var j = 0;j< 3;j++){

if(j == 1){

break;

}

}

}

while迴圈跳出：

//while迴圈 break跳出迴圈

var i = 1;

while(i< =10){

if(i == 4){

break;

}

print(i);

i++;

}

**continue使用：**

//如果i等於4的話跳過

for(var i=1;i<=5;i++){

if(i == 2){

continue; //跳過當前循環體 然後迴圈還會繼續執行

}

print(i);

}

**List常用屬性和方法**

**常用屬性：**

* length 長度
* reversed 翻轉
* isEmpty 是否為空
* isNotEmpty 是否不為空

**常用方法：**

* add 增加
* addAll 拼接陣列
* indexOf 查找 傳入具體值
* remove 刪除 傳入具體值
* removeAt 刪除 傳入索引值
* fillRange 修改
* insert(index,value) 指定位置插入
* insertAll(index,list) 指定位置插入List
* toList() 其他類型轉換成List
* join() List轉換成字串
* split() 字串轉化成List
* forEach
* map
* where
* any

一些常用屬性和方法使用舉例：

var list=['張三','李四','王五',"小明"];

print(list.length);

print(list.isEmpty);

print(list.isNotEmpty);

print(list.reversed); //對列表倒序排序

print(list.indexOf('李四')); //indexOf查找資料 查找不到返回-1 查找到返回索引值

list.remove('王五');

list.removeAt(2);

list.fillRange(1, 2,'a'); //修改 1是開始的位置 2二是結束的位置

print(list);

list.insert(1,'a');

print(list);

list.insertAll(1, ['a','b']); //插入多個

**Record常用屬性和方法**

**常用屬性：**

* keys: 返回一個包含所有欄位名稱的可反覆運算對象 (Iterable<Symbol>)。
* values: 返回一個包含所有欄位值的可反覆運算對象 (Iterable<Object?>)。
* entries: 返回一個包含所有欄位的可反覆運算對象 (Iterable<MapEntry<Symbol, Object?>>)。
* length: 返回記錄中欄位的數量。
* isEmpty: 檢查 Record 是否為空。
* forEach: 對 Record 中的每個鍵值對執行回呼函數。

示例:

* final person = (#name: 'Alice', #age: 30, #city: 'New York');
* print(person.keys); *// (Symbol("name"), Symbol("age"), Symbol("city"))*
* print(person.values); *// (Alice, 30, New York)*
* print(person.entries); *// (MapEntry(#name: Alice), MapEntry(#age: 30), MapEntry(#city: New York))*
* print(person.length); *// 3*

**常用方法：**

* get(Symbol fieldName): 返回指定欄位的值。如果欄位不存在,則返回 null。
* containsKey(Symbol fieldName): 檢查記錄是否包含指定的欄位名稱。
* toString(): 返回記錄的字串表示形式。
* hashCode: 返回記錄的雜湊碼。
* operator []: 可以使用方括號語法來訪問記錄的欄位值。
* []: 通過鍵獲取值。
* []=: 通過鍵設置值。
* containsValue: 檢查 Record 是否包含指定的值。
* toMap: 將 Record 轉換為 Map 對象。
* toString: 將 Record 轉換為字串表示形式。

示例:

* final person = (#name: 'Alice', #age: 30, #city: 'New York');
* print(person.get(#name)); *// Alice*
* print(person.containsKey(#age)); *// true*
* print(person['city']); *// New York*
* print(person); *// {#name: Alice, #age: 30, #city: New York}*

**操作方法：**

* operator +: 可以使用加號運算符將兩個記錄合併。
* operator -: 可以使用減號運算符從記錄中刪除指定的欄位。
* map: 將記錄中的每個欄位值映射到新的值。
* updateFields: 更新記錄中存在的欄位值。

示例:

* final person1 = (#name: 'Alice', #age: 30);
* final person2 = (#city: 'New York', #job: 'Engineer');
* final mergedPerson = person1 + person2; *// {#name: Alice, #age: 30, #city: New York, #job: Engineer}*
* final updatedPerson = mergedPerson.map((k, v) => MapEntry(k, v == 'Alice' ? 'Bob' : v)); *// {#name: Bob, #age: 30, #city: New York, #job: Engineer}*
* final removedPerson = mergedPerson - {#job}; *// {#name: Alice, #age: 30, #city: New York}*

### 示例

Record<String, dynamic> userProfile = {'name': 'Alice', 'age': 30, 'city': 'Taipei'};

// 獲取值

print(userProfile['name']); // Output: Alice

// 設置值

userProfile['city'] = 'New Taipei City';

// 檢查鍵是否存在

print(userProfile.containsKey('age')); // Output: true

// 檢查值是否存在

print(userProfile.containsValue('Alice')); // Output: true

// 遍歷鍵值對

userProfile.forEach((key, value) {

print('$key: $value');

});

// 轉換為 Map 對象

Map<String, dynamic> userProfileMap = userProfile.toMap();

// 轉換為字串表示形式

print(userProfile.toString()); // Output: {'name': 'Alice', 'age': 30, 'city': 'New Taipei City'}

**Set常用屬性和方法**

**Set**的最主要的功能就是去除數組重複內容，它是沒有順序且不能重複的集合，所以不能通過索引去獲取值。

var s = new Set();

s.add('A');

s.add('B');

s.add('B');

print(s); //{A, B}

當add相同內容時候無法添加進去的。

**Set**可以通過add方法添加一個List,並清除值相同的元素：

var list = ['香蕉','蘋果','西瓜','香蕉','蘋果','香蕉','蘋果'];

var s = new Set();

s.addAll(list);

print(s);

print(s.toList());

**Map的常用屬性和方法**

**Map**是無序的鍵值對，它的常用屬性主要有以下：

**常用屬性：**

* keys 獲取所有的key值
* values 獲取所有的value值
* isEmpty 是否為空
* isNotEmpty 是否不為空

**常用方法:**

* remove(key) 刪除指定key的資料
* addAll({...}) 合併映射 給映射內增加屬性
* containsValue 查看映射內的值 返回true/false
* forEach
* map
* where
* any
* every

**map轉換：**

List list = [1, 3, 4];

//map轉換，根據返回值返回新的元素清單

var newList = list.map((value) {

return value \* 2;

});

print(newList.toList());

**where：獲取符合條件的元素：**

List list = [1,3,4,5,7,8,9];

var newList = list.where((value){

return value > 5;

});

print(newList.toList());

**any：是否有符合條件的元素**

List list = [1, 3, 4, 5, 7, 8, 9];

//只要集合裡面有滿足條件的就返回true

var isContain = list.any((value) {

return value > 5;

});

print(isContain);

**every：需要每一個都滿足條件**

List myList=[1,3,4,5,7,8,9];

//每一個都滿足條件返回true 否則返回false

var flag = myList.every((value){

return value > 5;

});

print(flag);

**Set使用forEach遍歷：**

var s=new Set();

s.addAll([11,22,33]);

s.forEach((value) => print(value));

**Map使用forEach遍歷：**

Map person={

"name":"張三",

"age":28

};

person.forEach((key,value){

print("$key -> $value");

});

## Dart 中List的高級用法

Dart 中的 List 類型提供了豐富的高級用法，可用於更靈活地操作和管理清單中的資料。除了基本的添加、刪除和訪問元素之外，您還可以使用各種方法和技術來過濾、排序、轉換和組合清單資料。

## 1. 高效的列表操作

* **使用 Iterable 介面:** Iterable 介面提供了 forEach、map、where 等方法，可以更簡潔地遍歷和處理清單中的元素。例如：

List<int> numbers = [1, 2, 3, 4, 5];

// 使用 forEach 遍歷並列印每個元素

numbers.forEach((number) => print(number));

// 使用 map 將每個元素平方並返回新的清單

List<int> squares = numbers.map((number) => number \* number).toList();

print(squares); // 輸出: [1, 4, 9, 16, 25]

// 使用 where 過濾偶數並返回新的列表

List<int> evenNumbers = numbers.where((number) => number % 2 == 0).toList();

print(evenNumbers); // 輸出: [2, 4]

* **使用視圖列表:** 視圖列表（View List）提供了對現有列表的非破壞性視圖。您可以使用視圖列表來過濾、排序或截取原始列表的一部分，而無需創建新的列表副本。例如：

List<int> numbers = [1, 2, 3, 4, 5];

// 創建一個隻包含偶數的視圖列表

List<int> evenNumbersView = numbers.where((number) => number % 2 == 0).toList();

// 修改視圖清單中的元素

evenNumbersView[0] = 10;

// 原始列表也會受到影響

print(numbers); // 輸出: [1, 10, 3, 4, 5]

* **使用 spread operator:** 展開運運算元（Spread Operator）可以將多個可反覆運算物件合併成一個新的清單。例如：

List<int> numbers1 = [1, 2, 3];

List<int> numbers2 = [4, 5, 6];

// 將兩個列表合併成一個新的列表

List<int> allNumbers = [...numbers1, ...numbers2];

print(allNumbers); // 輸出: [1, 2, 3, 4, 5, 6]

## 2. 高階函數和函數運算式

* **使用高階函數:** 高階函數（Higher-Order Function）是指可以接收函數作為參數或返回值的函數。Dart 中提供了許多高階函數，例如 sort、any、reduce 等，可以用於更高級的列表操作。例如：

List<int> numbers = [5, 2, 4, 1, 3];

// 使用 sort 方法對列表排序

numbers.sort();

print(numbers); // 輸出: [1, 2, 3, 4, 5]

// 使用 any 方法檢查列表中是否存在大於 4 的元素

bool hasGreater = numbers.any((number) => number > 4);

print(hasGreater); // 輸出: true

// 使用 reduce 方法計算清單中所有元素的平方和

int sumOfSquares = numbers.reduce((sum, number) => sum + number \* number);

print(sumOfSquares); // 輸出: 55

* **使用函數運算式:** 函數運算式（Function Expression）允許您在代碼塊中定義匿名函數。這使得您可以在需要時動態創建和使用函數。例如：

List<int> numbers = [1, 2, 3, 4, 5];

// 使用函數運算式過濾偶數

List<int> evenNumbers = numbers.where((number) {

if (number % 2 == 0) return true;

return false;

}).toList();

print(evenNumbers); // 輸出: [2, 4]

## 3. 類型推斷和泛型

### 使用類型推斷

Dart 支援類型推斷，這意味著您可以省略清單元素的類型，Dart 編譯器會根據您提供的元素自動推斷類型。例如：

List<int> numbers = [1, 2, 3, 4, 5];

List<String> names = ["Alice", "Bob", "Charlie"];

// 編譯器會自動推斷 numbers 的元素類型為 int

// 編譯器會自動推斷 names 的元素類型為 String

類型推斷可以使您的代碼更加簡潔易讀，因為它可以減少您需要編寫的型別宣告。但是，在某些情況下，您可能需要顯式指定類型以提高代碼的可讀性和可維護性。

例如，如果您要創建一個包含不同類型元素的清單，則可以使用顯式型別宣告來確保代碼的可讀性：

List<Object> mixedList = [1, "Hello", true];

在某些情況下，Dart 編譯器無法正確推斷清單元素的類型。例如，如果您使用匿名函數來創建清單，則可能需要顯式指定類型：

List<int> numbers = [1, 2, 3].where((number) => number % 2 == 0).toList();

// 編譯器無法推斷匿名函數的參數類型

// 需要顯式指定類型

List<int> evenNumbers = [1, 2, 3].where((int number) => number % 2 == 0).toList();

* **使用泛型:** 泛型（Generic）允許您創建可用於不同類型資料的可重用清單類。例如：

class MyList<T> {

List<T> \_data = [];

void add(T item) {

\_data.add(item);

}

T operator [](int index) {

return \_data[index];

}

int get length => \_data.length;

}

// 創建一個 MyList<String> 物件

MyList<String> myStrings = MyList<String>();

myStrings.add("Hello");

myStrings.add("World");

print(myStrings[0]); // 輸出: Hello

print(myStrings.length); // 輸出: 2

// 創建一個 MyList<int> 物件

MyList<int> myNumbers = MyList<int>();

myNumbers.add(1);

myNumbers.add(2);

print(myNumbers[1]); // 輸出: 2

print(myNumbers.length); // 輸出: 2

## 4. 其他高級用法

* **使用不可變列表:** 不可變列表（Immutable List）是一種不可修改的列表。這使得它們非常適合在需要執行緒安全或不可變性的情況下使用。例如：

List<String> myNames = const ["Alice", "Bob", "Charlie"];

// 嘗試修改不可變列表會報錯

// myNames[0] = "David";

* **使用擴展方法:** 擴展方法（Extension Method）允許您為現有類添加新方法。您可以使用擴展方法來為列表類添加自訂方法，例如過濾、排序或轉換清單資料。例如：

extension ListExtension<T> on List<T> {

List<T> whereEven() {

return where((element) => element % 2 == 0);

}

}

List<int> numbers = [1, 2, 3, 4, 5];

// 使用擴展方法過濾偶數

List<int> evenNumbers = numbers.whereEven();

print(evenNumbers); // 輸出: [2, 4]

以下是一些 List 的高級用法示例:

1. **List 展開運算符 (Spread Operator)**

展開運算符 ... 可用於將一個 List 插入到另一個 List 中,或者合併多個 List。

var list1 = [1, 2, 3];

var list2 = [4, 5];

var mergedList = [...list1, ...list2]; *// [1, 2, 3, 4, 5]*

1. **List 生成式 (List Comprehensions)**

List 生成式提供了一種簡潔的語法來創建新的 List。

var squares = [for (var i = 1; i <= 5; i++) i \* i]; *// [1, 4, 9, 16, 25]*

1. **高階方法**

List 提供了許多高階方法,如 map、reduce、where 等,可以更好地處理集合數據。

var numbers = [1, 2, 3, 4, 5];

var doubledNumbers = numbers.map((n) => n \* 2).toList(); *// [2, 4, 6, 8, 10]*

var evenNumbers = numbers.where((n) => n.isEven).toList(); *// [2, 4]*

var sum = numbers.reduce((a, b) => a + b); *// 15*

1. **List 切片 (Slicing)**

可以使用 sublist 方法從一個 List 中提取一個子列表。

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var slice1 = numbers.sublist(2, 5); *// [3, 4, 5]*

var slice2 = numbers.sublist(5); *// [6, 7, 8, 9, 10]*

1. **List 轉換**

List 可以轉換為其他集合類型,如 Set 或 Map。

var numbers = [1, 2, 3, 3, 4, 4, 5];

var numbersSet = numbers.toSet(); *// {1, 2, 3, 4, 5}*

var numbersMap = numbers.asMap(); *// {0: 1, 1: 2, 2: 3, 3: 3, 4: 4, 5: 4, 6: 5}*

1. **List 排序**

List 提供了內置的排序功能,可以按照自定義的規則對 List 進行排序。

var names = ['Alice', 'Bob', 'Charlie', 'David'];

names.sort(); *// ['Alice', 'Bob', 'Charlie', 'David']*

names.sort((a, b) => b.compareTo(a)); *// ['David', 'Charlie', 'Bob', 'Alice']*

這些只是 Dart 中 List 的一些高級用法示例。利用這些用法,您可以更加高效地處理集合數據,編寫更加簡潔和易讀的代碼。

**Records 記錄類型(元組) 表示和使用方法**

Records are an anonymous, immutable , aggregate type.記錄是一種匿名的、不可變的聚合類型。

#### 1. 記錄類型的聲明與訪問

透過()將若干個物件組合在一塊，作為一個新的聚合類型。定義時可以直接放入對象，也可以進行命名傳入：

var record = ('first', a: 2, b: true, 'last');

print(record.runtimeType);

--->[列印輸出]---

(String, String, {int a, bool b})

上面的record 物件由四個資料構成，透過runtimeType可以查看到其運行時類型，類型為各個非命名資料類型+ 各命名類型。

非命名類型資料可以透過$index進行存取：

print(record.$1);

print(record.$2);

--->[列印輸出]---

first

last

命名類型資料可以透過名稱進行存取：

print(record.a);

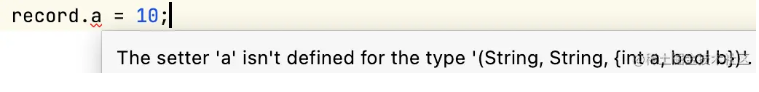
print(record.b);

--->[列印輸出]---

2

true

注意: 一個記錄物件的資料不允許被修改:



#### 2. 記錄類型聲明對象

一個Records 本質上也是一種類型，可以用該類型來宣告對象，例如現在透過(double,double,double) 的記錄類型表示三個座標，如下定義p0 和p1 物件：

void main() {

(double x, double y, double z) p0 = (1, 2, 3);

(double x, double y, double z) p1 = (1, 2, 6);

}

既然可以實例化為對象，那麼自然也可以將其作為參數類型傳入函數中，如下distance方法傳入兩個三維點，計算距離：

double distance(

(double x, double y, double z) p0,

(double x, double y, double z) p1,

) {

num result = pow(p0.$1 - p1.$1, 2) + pow(p0.$2 - p1.$2, 2) + pow(p0.$3 - p1.$3, 2);

return sqrt(result);

}

但記錄類型一旦顯示聲明，寫起來比較繁瑣；和函數類型類似，也可以透過typedef 來定義類型的別名。如下所示，定義Point3D作為別名，功能是等價的，但書寫和可讀性會更好一些：

typedef Point3D = (double, double, double);

void main() {

Point3D p0 = (1, 2, 3);

Point3D p1 = (1, 2, 6);

print(distance(p0, p1));

}

double distance(Point3D p0, Point3D p1) {

num result = pow(p0.$1 - p1.$1, 2) + pow(p0.$2 - p1.$2, 2) + pow(p0.$3 - p1.$3, 2);

return sqrt(result);

}

同理，記錄類型也可以作為傳回值，這樣可以解決一個函數傳回多值的問題。如下addTask方法可以計算1 ~ count 的累加值，傳回計算結果與耗時毫秒數：

({int result, int cost}) addTask2(int count) {

int start = DateTime.now().millisecondsSinceEpoch;

int sum = 0;

for (int i = 0; i < count; i++) {

sum += i;

}

int end = DateTime.now().millisecondsSinceEpoch;

return (

result: sum,

cost: end - start,

);

}

#### 3. 記錄類型物件的等值

記錄類型會根據欄位的結構自動定義hashCode 和== 方法。所以兩個記錄物件相等，就是其中的各個數值相等。但透過identical可以看出p0 和p1 仍是兩個對象，記憶體位址不同：

(double, double, double) p0 = (1, 2, 3);

(double, double, double) p1 = (1, 2, 3);

print(p0 == p1);

print(identical(p0, p1));

--->[列印輸出]---

true

false

如下圖所示，第二個資料是List<double>類型，兩個[2] 是兩個不同的對象，所以p2，p3 不相等：

(double, List<double>, double) p2 = (1, [2], 3);

(double, List<double>, double) p3 = (1, [2], 3);

print(p2==p3);

--->[列印輸出]---

false

在下面測試中， 列表使用相同對象，則p2，p3 相等：

List<double> li = [2];

(double, List<double>, double) p2 = (1, li, 3);

(double, List<double>, double) p3 = (1, li, 3);

print(p2==p3);

--->[列印輸出]---

true

#### 4. 記錄類型的價值

對於程式語言來說，Dart 的記錄類型也不是什麼新的東西，就是其他語言中的元組。如下所示，可以建立一個TaskResult類別來維護資料作為回傳值。但如果只是傳回一些臨時的數據，為此新建一個類別來維護數據就會顯得比較繁瑣，還要定義建構子。

class TaskResult{

final int result;

final int cost;

TaskResult(this.result, this.cost);

}

TaskResult addTask(int count) {

int start = DateTime.now().millisecondsSinceEpoch;

int sum = 0;

for (int i = 0; i < count; i++) {

sum += i;

}

int end = DateTime.now().millisecondsSinceEpoch;

return TaskResult(sum, end - start);

}

除此之外，一個函數傳回多個資料也可以使用Map 物件：

Map<String,dynamic> addTask(int count) {

int start = DateTime.now().millisecondsSinceEpoch;

int sum = 0;

for (int i = 0; i < count; i++) {

sum += i;

}

int end = DateTime.now().millisecondsSinceEpoch;

return {

'result' : sum,

'cost': end - start

};

}

但這種方式的弊端也很明顯，回傳和使用時都需要固定的字串作為key。如果key 寫錯了，程式碼在運行前也不會有任何錯誤，這樣很容易就會出現風險。當多人協作時，而且如果函數的書寫者和呼叫者不是一個人，那麼該使用什麼鍵得到什麼值就很難分辨。

Map<String,dynamic> task = addTask2(100000000);

print(task['result']);

print(task['cost']);

所以，相較於新建class 或透過Map 來維護多個數據，使用記錄類型更加方便快速精確。但話說回來，如果屬性資料量過多，使用記錄類型看起來會非常麻煩，也不能定義成員方法來操作、修改內部資料。所以它有自己的特色使用場景，例如暫時聚合多個資料來方便使用。

**Record 的高級用法**

Dart 中的 Record 是一種匿名的、不可變的聚合類型。它可以用來將多個值綁定在一起，就像結構體或字典一樣。但是，Record 與結構體和字典有一些關鍵的區別：

* Record 是匿名的，這意味著它沒有名稱。
* Record 是不可變的，這意味著一旦創建，就不能再更改其值。
* Record 的值是異構的，這意味著它可以包含不同類型的值。

這些特性使得 Record 非常適合用於創建臨時資料結構、傳遞參數和返回值，以及在需要類型安全性的情況下存儲資料。

### 高級用法示例

以下是一些 Dart 中 Record 的高級用法示例：

**1. 使用 Record 作為參數和返回值**

Record 可以作為函數的參數和返回值。這使得它們非常適合用於在函數之間傳遞資料。例如，以下代碼定義了一個函數，該函數接受一個 Record 作為參數並返回另一個 Record：

Record<String, int> createPerson(String name, int age) {

return {'name': name, 'age': age};

}

void main() {

var person = createPerson('Alice', 30);

print(person); // Output: {'name': 'Alice', 'age': 30}

}

**2. 使用 Record 作為臨時資料結構**

Record 可以用於創建臨時資料結構，例如在迴圈或條件塊中。這使得它們比結構體或字典更羽量級，並且可以避免不必要的記憶體分配。例如，以下代碼使用 Record 來存儲迴圈中每個元素的平方：

List<int> numbers = [1, 2, 3, 4, 5];

for (int number in numbers) {

var square = {'number': number, 'square': number \* number};

print(square);

}

**3. 使用 Record 來存儲類型安全的資料**

Record 可以用於存儲類型安全的資料，這可以説明防止執行階段錯誤。例如，以下代碼使用 Record 來存儲使用者設定檔：

Record<String, dynamic> getUserProfile(String userId) {

// Fetch user profile from database or API

return {'id': userId, 'name': 'Alice', 'age': 30};

}

void main() {

var profile = getUserProfile('user123');

print(profile['name']); // Output: Alice

print(profile['age']); // Output: 30

}

**4. 使用 Record 與泛型一起使用**

Record 可以與泛型一起使用來提高代碼的再使用性和類型安全性。例如，以下代碼定義了一個泛型函數，該函數接受一個 Record 作為參數並返回其值：

T getValue<T>(Record<String, T> record, String key) {

return record[key]!;

}

void main() {

var person = createPerson('Alice', 30);

var name = getValue(person, 'name');

var age = getValue(person, 'age');

print(name); // Output: Alice

print(age); // Output: 30

}

**5. 使用 Record 與模式匹配一起使用**

Record 可以與模式匹配一起使用來提取資料。例如，以下代碼使用模式匹配來提取 Record 中的值：

Record<String, dynamic> getUserProfile(String userId) {

// Fetch user profile from database or API

return {'id': userId, 'name': 'Alice', 'age': 30};

}

void main() {

var profile = getUserProfile('user123');

switch (profile) {

case Record<String, String>('id', userId, 'name', name, 'age', age):

print('User ID: $userId');

print('Name: $name');

print('Age: $age');

break;

}

}

使用模式匹配來解構Record並提取所需的欄位值。

void main() {

final person = (#name: 'Alice', #age: 30);

if (person is (#name: final String name, #age: final int age)) {

print('Name: $name, Age: $age'); *// Name: Alice, Age: 30*

}

}

**6.解構賦值**

你可以使用解構賦值來同時獲取Record中多個欄位的值。這可以使你的代碼更加簡潔和易讀。

void main() {

final person = (#name: 'Alice', #age: 30, #city: 'New York');

final (#name: name, #age: age, #city: city) = person;

print('Name: $name, Age: $age, City: $city'); *// Name: Alice, Age: 30, City: New York*

}

**7.嵌套Record**

Record可以包含其他Record作為欄位值,這使得它在表示複雜數據結構時特別有用。

void main() {

final address = (#street: '123 Main St.', #city: 'New York', #zip: '10001');

final person = (#name: 'Alice', #age: 30, #address: address);

print('Name: ${person.name}, Address: ${person.address.street}, ${person.address.city} ${person.address.zip}');

*// Name: Alice, Address: 123 Main St., New York 10001*

}

* 1. **Record操作**

Dart提供了一些方便的方法來操作Record,例如合併多個Record、更新欄位值等。

void main() {

final person1 = (#name: 'Alice', #age: 30);

final person2 = (#name: 'Bob', #city: 'London');

*// 合併Record*

final mergedPerson = {...person1, ...person2};

print(mergedPerson); *// {#name: Bob, #age: 30, #city: London}*

*// 更新欄位值*

final updatedPerson = {...person1, #age: 35};

print(updatedPerson); *// {#name: Alice, #age: 35}*

}

* 1. **Record作為數據類型**

Record可以用作函數參數或返回值的數據類型,這可以提高代碼的類型安全性和可讀性。

Record getPersonInfo(String name, int age) {

return (#name: name, #age: age);

}

void main() {

final person = getPersonInfo('Alice', 30);

print('Name: ${person.name}, Age: ${person.age}'); *// Name: Alice, Age: 30*

}

Record是Dart中一種強大的新數據結構,它提供了一種簡潔、安全和靈活的方式來處理數據。

## Dart 中的 Set：表示及使用(1)

Set（集合）是一種資料結構，用於儲存不重複的值。它與 List（列表）類似，但 Set 中的值是唯一的，並且順序是不確定的。

### 建立 Set

要建立 Set，請使用 Set 類別的建構子：

Set<String> mySet = Set<String>();

此程式碼會建立一個空 Set，其中包含字串型別的值。

您也可以使用字面量來建立 Set，其中包含要儲存的值：

Set<String> mySet = {"apple", "banana", "orange"};

此程式碼會建立一個 Set，其中包含 "apple"、"banana" 和 "orange" 字串。

### 檢查 Set 中的值

您可以使用 contains() 方法來檢查 Set 中是否包含特定值：

Set<String> mySet = {"apple", "banana", "orange"};

bool hasApple = mySet.contains("apple"); // true

bool hasGrape = mySet.contains("grape"); // false

### 新增值到 Set

您可以使用 add() 方法將值新增到 Set 中：

Set<String> mySet = {"apple", "banana", "orange"};

mySet.add("grape");

print(mySet); // {"apple", "banana", "orange", "grape"}

### 刪除值從 Set

您可以使用 remove() 方法從 Set 中刪除值：

Set<String> mySet = {"apple", "banana", "orange"};

mySet.remove("orange");

print(mySet); // {"apple", "banana"}

### 反覆運算 Set 中的值

您可以使用 forEach() 方法來反覆運算 Set 中的值：

Set<String> mySet = {"apple", "banana", "orange"};

mySet.forEach((value) {

print(value); // 列印 "apple"、"banana" 和 "orange"

});

### 合併 Set

您可以使用 union() 方法來合併兩個 Set：

Set<String> set1 = {"apple", "banana"};

Set<String> set2 = {"orange", "grape"};

Set<String> unionSet = set1.union(set2);

print(unionSet); // {"apple", "banana", "orange", "grape"}

### 交集 Set

您可以使用 intersection() 方法來查找兩個 Set 的交集：

Set<String> set1 = {"apple", "banana"};

Set<String> set2 = {"orange", "banana"};

Set<String> intersectionSet = set1.intersection(set2);

print(intersectionSet); // {"banana"}

### 差集 Set

您可以使用 difference() 方法來查找兩個 Set 的差集：

Set<String> set1 = {"apple", "banana"};

Set<String> set2 = {"orange", "banana"};

Set<String> differenceSet = set1.difference(set2);

print(differenceSet); // {"apple"}

### 結論

Set 是一種強大的資料結構，可用於存儲不重複的值。它可以用於各種任務，例如跟蹤已訪問的項目、刪除重複項或查找兩個資料集的共同點。

## Dart 中 Set 的高級用法

除了基本的 Set 表示和使用之外，Dart 中的 Set 還支持一些高級用法，可以讓您更靈活地操作和利用 Set。

### 視圖 Set

視圖 Set（View Set）是一種特殊的 Set，它基於另一個現有的 Set 創建，並提供對該基礎 Set 的部分或全部視圖。視圖 Set 不會存儲自己的資料，而是會引用基礎 Set 中的資料。

例如，我們可以創建一個視圖 Set，只包含偶數：

Set<int> numbers = {1, 2, 3, 4, 5, 6};

Set<int> evenNumbers = numbers.where((number) => number % 2 == 0);

print(evenNumbers); // 輸出 {2, 4, 6}

在這個例子中，evenNumbers 是一個視圖 Set，它基於 numbers Set 創建，並只包含偶數。對 evenNumbers 的任何修改都會反映到 numbers 中。

視圖 Set 還有其他一些用法，例如：

* 創建只包含特定範圍的值的視圖
* 創建只包含符合特定條件的值的視圖
* 創建兩個 Set 的交集、並集或差集的視圖

### 不可變 Set

不可變 Set（Immutable Set）是一種特殊的 Set，一旦創建後就不能再修改。這使得它們非常適合在需要執行緒安全或不可變性的情況下使用。

例如，我們可以創建一個不可變 Set：

Set<String> mySet = Set.from(["apple", "banana", "orange"]);

// mySet.add("grape"); // 會報錯

在這個例子中，mySet 是一個不可變 Set，一旦創建後就不能再添加或刪除元素。

### 類型推斷

Dart 支援類型推斷，這意味著您可以省略 Set 的元素類型，Dart 編譯器會根據您提供的元素自動推斷類型。

例如，我們可以創建一個 Set，其中包含字串：

Set mySet = {"apple", "banana", "orange"};

在這個例子中，編譯器會推斷 mySet 的元素類型為 String，因為您提供的元素都是字串。

### 總結

Dart 中的 Set 的高級用法提供了強大的功能，可以讓您更靈活地操作和利用 Set。通過視圖 Set、不可變 Set 和類型推斷，您可以創建更具表達力和可維護性的代碼。

## Dart 中的 Set：表示及使用(2)

在 Dart 中，Set 是一種無序且不重複的集合數據類型。以下是如何表示及使用 Set 的方式:

1. **創建 Set**

*// 使用字面量語法創建 Set*

var set1 = {'apple', 'banana', 'orange'};

*// 使用構造函數創建空 Set*

var set2 = <String>{};

*// 使用泛型語法創建 Set*

var set3 = <int>{1, 2, 3, 3, 2}; *// 重複的值會被忽略*

1. **存取 Set 元素**

print(set3.length); *// 輸出: 3*

print(set3.contains(2)); *// 輸出: true*

print(set3.first); *// 輸出: 1 (第一個元素)*

set1.add('grape'); *// 添加新元素*

set1.remove('banana'); *// 移除元素*

set3.addAll([4, 5]); *// 添加多個元素*

set3.removeAll([1, 3]); *// 移除多個元素*

1. **Set 操作**

var set5 = {2, 3, 4};

var union = set4.union(set5); *// 聯集 {1, 2, 3, 4}*

var intersection = set4.intersection(set5); *// 交集 {2}*

print(set4.difference(set5)); *// 補集 {1}*

print(set5.difference(set4)); *// 補集 {3, 4}*

1. **遍歷 Set**

for (var item in set1) {

print(item); *// 依次輸出 set1 中的元素*

}

set1.forEach(print); *// 使用 forEach 方法遍歷*

1. **從List創建Set**

var list = [1, 2, 3, 2, 4, 1];

var set = Set.from(list); *// 從 List 創建 Set, 去除重複項*

print(set); *// 輸出: {1, 2, 3, 4}*

Set 在 Dart 中非常有用,可用於存儲不重複的元素集合,比如標籤、id等。使用 Set 可以避免重複數據,並提供了聯集、交集和補集等常用操作。

需要注意的是,Set 是無序的,因此不能通過索引來訪問元素。如果需要訪問特定元素,可以使用 contains 方法檢查元素是否存在。

## Dart 中 Set 的高級用法

Dart 中的 Set 除了基本的添加、刪除和遍歷元素之外,還提供了一些高級用法,可以增強程式碼的靈活性和功能性。以下是一些 Set 的高級用法:

1. **Set 操作**

Set 提供了一些有用的操作,如聯集 (union)、交集 (intersection)、差集 (difference) 和補集 (difference)。這些操作可以方便地對多個 Set 執行集合運算。

var set1 = {1, 2, 3};

var set2 = {3, 4, 5};

var union = set1.union(set2); *// {1, 2, 3, 4, 5}*

var intersection = set1.intersection(set2); *// {3}*

var difference = set1.difference(set2); *// {1, 2}*

1. **Set 轉換為其他集合類型**

Set 可以轉換為 List 或 Map,反之亦然。這種轉換在某些情況下非常有用。

var set = {1, 2, 3};

var list = set.toList(); *// [1, 2, 3]*

var map = set.asMap(); *// {0: 1, 1: 2, 2: 3}*

var list2 = [4, 5, 5, 6];

var set2 = Set.from(list2); *// {4, 5, 6}*

1. **Set 作為函數參數**

Set 可以作為函數參數傳遞,用於去除重複項或執行集合操作。

void printDistinctElements(Set<int> elements) {

print(elements);

}

printDistinctElements({1, 2, 3, 2, 4}); *// {1, 2, 3, 4}*

1. **使用 Set 進行集合操作**

由於 Set 不允許重複元素,因此可以使用它來去除重複項或執行其他集合操作。

var list = [1, 2, 3, 3, 4, 4, 5];

var distinctList = list.toSet().toList(); *// [1, 2, 3, 4, 5]*

1. **使用 Set 作為快取**

Set 可以用作快取,因為它不允許重複元素,並且提供了快速的查找操作。

Set<String> cache = {};

bool isCached(String key) {

return cache.contains(key);

}

void cacheData(String key, String value) {

cache.add(key);

*// 執行一些操作以存儲 (key, value) 對*

}

Set 在 Dart 中提供了強大的功能,可以用於去重、集合運算和快取等場景。利用 Set 的高級用法,可以使程式碼更加簡潔、高效和靈活。

**Map 的表示和使用方法(1)**

在 Dart 中,Map是一種鍵值對的資料結構,可以用來存儲和檢索值。以下是如何表示和使用 Map 的方式:

1. **創建 Map**

*// 使用字面量語法創建 Map*

var map1 = {'name': 'John', 'age': 30};

*// 使用構造函數創建空 Map*

var map2 = Map();

*// 使用泛型語法創建 Map*

var map3 = <String, int>{'id': 123, 'score': 90};

1. **存取 Map 值**

print(map1['name']); *// 輸出: John*

print(map3['id']); *// 輸出: 123*

map1['city'] = 'New York'; *// 添加新鍵值對*

print(map1); *// 輸出: {name: John, age: 30, city: New York}*

1. **Map 操作**

print(map1.length); *// 取得 Map 長度, 輸出: 3*

print(map1.isEmpty); *// 檢查 Map 是否為空, 輸出: false*

print(map1.containsKey('name')); *// 檢查是否包含某個鍵, 輸出: true*

map1.remove('age'); *// 移除一個鍵值對*

map1.clear(); *// 清空整個 Map*

*// 遍歷 Map*

map3.forEach((key, value) {

print('$key: $value');

});

1. **Map 合併**

var map4 = {'a': 1, 'b': 2};

var map5 = {'c': 3, 'd': 4};

var merged = {...map4, ...map5}; *// 合併兩個 Map*

print(merged); *// 輸出: {a: 1, b: 2, c: 3, d: 4}*

1. **Map 作為函數參數**

void printMap(Map<String, dynamic> map) {

map.forEach((key, value) {

print('$key: $value');

});

}

printMap({'x': 10, 'y': 20}); *// 將 Map 作為參數傳遞*

Map 在 Dart 中非常有用,可用於存儲各種類型的鍵值對資料,如對象屬性、查詢參數等。使用 Map 可以方便地組織和訪問資料,使代碼更加清晰和可維護。

**Map 的表示和使用方法(2)**

在 Dart 中，Map 是一種基本資料結構，用於表示鍵值對的集合。每個鍵必須在該映射中唯一，但值可以重複。Map 是無序集合，這意味著添加元素到映射中的順序不會保留。

**創建 Map**

有兩種主要方法可以在 Dart 中創建 Map：

1. **使用 Map 字面量：**

Map<String, String> fruits = {

"apple": "red",

"banana": "yellow",

"orange": "orange",

};

此方法直接定義了一個帶有大括號括起來的鍵值對的映射。鍵類型指定為 String，值類型指定為 String。在此示例中，fruits 映射將鍵 "apple" 與值 "red" 相關聯，鍵 "banana" 與值 "yellow" 相關聯，鍵 "orange" 與值 "orange" 相關聯。

1. **使用 Map 構造函數：**

Map<String, String> fruits = Map<String, String>();

fruits["apple"] = "red";

fruits["banana"] = "yellow";

fruits["orange"] = "orange";

此方法涉及使用 Map 構造函數創建空映射，然後使用 [] 運算子添加鍵值對。此處，fruits 映射最初為空，然後使用 [] 運算子逐個添加元素。

**訪問元素**

要檢索與 Map 中特定鍵關聯的值，請直接使用鍵。例如，要從 fruits 映射中獲取鍵 "apple" 的值：

String appleColor = fruits["apple"];

// appleColor 將為 "red"

如果指定的鍵不存在，您可以獲取空值或提供預設值：

String grapeColor = fruits["grape"]; // grapeColor 將為 null

String grapeColorOrDefault = fruits["grape"] ?? "unknown"; // grapeColorOrDefault 將為 "unknown"

在第一行中，從 fruits 映射中訪問鍵 "grape" 會導致 null 值，因為鍵 "grape" 不存在於映射中。在第二行中，使用空合併運算子 (??) 來提供預設值 "unknown"，如果找不到鍵 "grape"。

**添加元素**

要將新元素添加到 Map 中，請使用 [] 運算子並將其值分配給所需的鍵：

fruits["grape"] = "purple";

此行將新的鍵值對添加到 fruits 映射中。鍵 "grape" 與值 "purple" 相關聯。

**刪除元素**

要從 Map 中刪除元素，請使用 remove() 方法並傳遞要刪除的元素的鍵：

fruits.remove("grape");

此行從 fruits 映射中刪除與鍵 "grape" 關聯的鍵值對。

**遍歷 Map**

要遍歷 Map 中的鍵值對，請使用 for 迴圈：

for (var entry in fruits.entries) {

print("${entry.key}: ${entry.value}");

}

此迴圈反覆運算 fruits 映射中的每個鍵值對。entry 變數表示每個對，其 key 和 value 屬性提供對相應鍵和值的訪問。迴圈以 "鍵: 值" 格式列印每個對。

**檢查鍵是否存在**

要驗證鍵是否存在於 Map 中，請使用 containsKey() 方法：

bool hasGrape = fruits.containsKey("grape"); // hasGrape 將為 false

此行檢查鍵 "grape" 是否存在於 fruits 映射中。hasGrape 變數將設置為 false，因為鍵 "grape" 已被先前刪除。

**獲取 Map 的大小**

要確定 Map 的大小，請使用 length 屬性：

int fruitCount = fruits.length; // fruitCount 將為 2

此行檢索 fruits 映射中的鍵值對數。fruitCount 變數將設置為 2，因為映射現在包含兩個對。

**Map 的高級用法**

除了基本的創建、訪問、添加和刪除元素等操作之外，Map 在 Dart 中還提供了許多高級功能，可用於更複雜的場景。以下是一些高級用法的示例：

**1. 使用 Map.fromIterable 從可反覆運算對象創建 Map:**

List<String> keys = ["name", "age", "occupation"];

List<String> values = ["Alice", "30", "Software Engineer"];

Map<String, String> person = Map.fromIterable(keys, value: (key) => values[keys.indexOf(key)]);

此代碼將創建以下 Map：

{

"name": "Alice",

"age": "30",

"occupation": "Software Engineer"

}

Map.fromIterable 方法可用於將可反覆運算物件（例如清單）轉換為 Map。第一個參數是鍵列表，第二個參數是值生成器函數。值生成器函數將為每個鍵生成相應的值。

**2. 使用 Map.fromKeyMap 從鍵映射創建 Map:**

Map<String, dynamic> keyMap = {

"name": (String name) => name.toUpperCase(),

"age": (int age) => age + 1,

"occupation": (String occupation) => occupation.replaceAll(' ', '\_'),

};

Map<String, dynamic> person = Map.fromKeyMap(keyMap, values: ["Alice", 30, "Software Engineer"]);

此代碼將創建以下 Map：

{

"NAME": "ALICE",

"AGE": 31,

"OCCUPATION\_SOFTWARE\_ENGINEER": "Software\_Engineer"

}

Map.fromKeyMap 方法可用於將鍵映射（其中鍵與值生成器函數相關聯）轉換為 Map。值生成器函數將為每個鍵生成相應的值。

**3. 使用 spread 運算符合並 Map:**

Map<String, String> person1 = {

"name": "Alice",

"age": "30"

};

Map<String, String> person2 = {

"occupation": "Software Engineer",

"city": "Taipei"

};

Map<String, String> combinedPerson = {...person1, ...person2};

此代碼將創建以下 Map：

{

"name": "Alice",

"age": "30",

"occupation": "Software Engineer",

"city": "Taipei"

}

spread 運算子可用於將多個 Map 合併為一個 Map。

**4. 使用 forEach 方法遍歷 Map 並執行操作:**

Map<String, String> person = {

"name": "Alice",

"age": "30",

"occupation": "Software Engineer"

};

person.forEach((key, value) {

print("Key: $key, Value: $value");

});

此代碼將列印以下輸出：

Key: name, Value: Alice

Key: age, Value: 30

Key: occupation, Value: Software Engineer

forEach 方法可用於遍歷 Map 中的鍵值對並對每個對執行操作。

**5. 使用 map 方法轉換 Map 的值:**

Map<String, String> person = {

"name": "Alice",

"age": "30",

"occupation": "Software Engineer"

};

Map<String, String> upperCasePerson = person.map((key, value) => MapEntry(key, value.toUpperCase())).toMap();

此代碼將創建以下 Map：

{

"NAME": "ALICE",

"AGE": "30",

"OCCUPATION": "SOFTWARE ENGINEER"

}

map 方法可用於轉換 Map 中的值。它接受一個值轉換器函數作為參數，該函數將為每個鍵生成新的值。

**6. 使用 where 方法過濾 Map:**

Map<String, String> person = {

"name": "Alice",

"age": "30",

"occupation": "Software Engineer"

};

Map<String, String> filteredPerson = person.where((key, value) => key.length == 4);

此代碼將創建以下 Map：

{

"name": "Alice"

}

where 方法可用於過濾 Map 中的鍵值對。它接受一個謂詞函數作為參數，該函數將為每個對返回 true 或 false。只有鍵值對返回 true 的才會包含在結果 Map 中。

**7. 使用 reduce 方法將 Map 的值折疊為單個值:**

Map<String, int> salaries = {

"Alice": 50000,

"Bob": 60000,

"Charlie": 70000

};

int totalSalary = salaries.reduce((sum, value) => sum + value, 0);

此代碼將計算 salaries 映射中所有值的總和，結果為 180000。

reduce 方法可用於將 Map 中的值折疊為單個值。它接受一個累加器函數和初始值作為參數。累加器函數將為每個鍵值對和累加器值調用，累加器值將更新為函數的返回值。最終的累加器值將作為結果返回。

**8. 使用 any 方法檢查 Map 中是否存在滿足條件的元素:**

Map<String, String> person = {

"name": "Alice",

"age": "30",

"occupation": "Software Engineer"

};

bool hasAdult = person.any((key, value) => value == "30");

此代碼將檢查 person 映射中是否存在值為 "30" 的元素。hasAdult 變數將設置為 true。

any 方法可用於檢查 Map 中是否存在滿足條件的元素。它接受一個謂詞函數作為參數，該函數將為每個鍵值對返回 true 或 false。如果任何鍵值對返回 true，則方法將返回 true；否則，將返回 false。

**9. 使用 sortKeys 方法按鍵對 Map 進行排序:**

Map<String, String> person = {

"occupation": "Software Engineer",

"name": "Alice",

"age": "30"

};

Map<String, String> sortedPerson = Map.fromIterable(person.keys.toList()..sort(), value: (key) => person[key]);

此代碼將按鍵對 person 映射進行昇冪排序。sortedPerson 映射將包含按字母順序排列的鍵。

**10. 使用 removeWhere 方法刪除 Map 中滿足條件的元素:**

Map<String, String> person = {

"name": "Alice",

"age": "30",

"occupation": "Software Engineer"

};

person.removeWhere((key, value) => value == "30");

此代碼將從 person 映射中刪除所有值為 "30" 的元素。

removeWhere 方法可用於刪除 Map 中滿足條件的元素。它接受一個謂詞函數作為參數，該函數將為每個鍵值對返回 true 或 false。如果任何鍵值對返回 true，則該對將被刪除。

這些只是 Dart 中 Map 高級用法的一些示例。Map 提供了許多其他功能，可用於各種複雜任務

**自訂義資料型態**

在 Dart 中,有幾種方式可以自訂義資料型態:

1. **Class**
   * 使用 class 關鍵字定義一個類別
   * 可以包含屬性(properties)和方法(methods)
   * 支持繼承、介面實作和泛型等面向物件的功能

範例:

class Person {

String name;

int age;

Person(this.name, this.age);

void greeting() {

print('Hello, my name is $name and I am $age years old.');

}

}

1. **Enum**
   * 使用 enum 關鍵字定義一個枚舉型別
   * 可以定義一組有限的常數值
   * 適用於代表一組固定選項的場景

範例:

enum Color { red, green, blue }

1. **Typedef**
   * 使用 typedef 為現有的資料型態定義一個別名
   * 通常用於簡化較複雜的型別定義

範例:

typedef IntList = List<int>;

IntList numberList = [1, 2, 3];

1. **Extension**
   * 使用 extension 為現有的類別添加額外的功能和方法
   * 不會修改原始類別的定義

範例:

extension NumberParsing on String {

int? toIntOrNull() {

return int.tryParse(this);

}

}

1. **Mixin**
   * 使用 mixin 為類別添加可重用的功能
   * 通過繼承和組合的方式實現代碼重用

範例:

mixin CanFly {

void fly() {

print('I can fly!');

}

}

class Bird with CanFly {}

6.interface

* + 介面與類別類似，但僅定義類別必須實作的方法和屬性。它們不提供任何實作細節。要建立介面，請使用 interface 關鍵字，後接介面名稱和大括號 {} 來定義其成員：通過繼承和組合的方式實現代碼重用
  + 介面定義了兩個方法，area() 和 perimeter()，任何實作 Shape 介面的類別都必須提供。

範例:

interface Shape { double get area(); double get perimeter(); }

### 實作介面

要實作介面，類別使用 implements 關鍵字，後接介面名稱：

class Circle implements Shape {

double radius;

@override

double get area() => PI \* radius \* radius;

@override

double get perimeter() => 2 \* PI \* radius;

}

**使用介面**

您可以使用介面來定義不同類別可以實作的通用行為。例如，您可以建立一個 Shape 介面，定義 area() 和 perimeter() 方法，然後建立 Circle、Rectangle 和 Triangle 等類別來實作此介面。這讓您可以以一致的方式處理各種形狀，而無需擔心每個類別的具體實作細節。

interface Shape {

double get area();

double get perimeter();

}

class Circle implements Shape {

double radius;

@override

double get area() => PI \* radius \* radius;

@override

double get perimeter() => 2 \* PI \* radius;

}

class Rectangle implements Shape {

double width;

double height;

@override

double get area() => width \* height;

@override

double get perimeter() => 2 \* (width + height);

}

class Triangle implements Shape {

double base;

double height;

double sideA;

double sideB;

@override

double get area() => 0.5 \* base \* height;

@override

double get perimeter() => base + sideA + sideB;

}

void calculateAreaAndPerimeter(List<Shape> shapes) {

for (Shape shape in shapes) {

print("Area: ${shape.area}");

print("Perimeter: ${shape.perimeter}");

}

}

List<Shape> shapes = [

Circle(5.0),

Rectangle(10.0, 5.0),

Triangle(7.0, 5.0, 6.0, 8.0),

];

calculateAreaAndPerimeter(shapes);

此程式碼會建立 Circle、Rectangle 和 Triangle 類別，這些類別都實作 Shape 介面。然後，它會建立一個 shapes 清單，其中包含這些類別的實例。最後，它會呼叫 calculateAreaAndPerimeter() 函數，該函數會迭代清單並計算每個形狀的面積和周長。

自訂義資料型態和介面是 Dart 中強大的工具，可讓您以更結構化和可重用的方式組織程式碼。它們有助於提高程式碼的可讀性、可維護性和可測試性。

這些方式都可以用於自訂義資料型態,選擇哪種方式取決於您的具體需求和場景。通常建議優先考慮使用 class,因為它是面向物件編程的核心,可以充分利用 Dart 的面向物件特性。

## Dart 中 forEach 的高級用法

forEach 方法是 Dart 中一種常用的方法，用於遍歷清單或集合中的每個元素。除了基本的遍歷元素之外，forEach 還支援一些高級用法，可以讓您更靈活地操作和處理遍歷中的元素。

### 使用 forEach 過濾元素

您可以使用 where 方法來過濾 forEach 遍歷中的元素。where 方法會返回一個新的清單或集合，其中只包含符合條件的元素。

例如，我們可以創建一個 List，並使用 forEach 遍歷其中的偶數：

List<int> numbers = [1, 2, 3, 4, 5, 6];

numbers.forEach((number) {

if (number % 2 == 0) {

print(number); // 輸出 2, 4, 6

}

});

您也可以使用箭頭函數和三元運算子來簡化代碼：

numbers.forEach((number) => number % 2 == 0 ? print(number) : null);

### 使用 forEach 修改元素

您可以直接在 forEach 遍歷中修改元素。例如，我們可以將 numbers 列表中的所有偶數轉換為奇數：

numbers.forEach((number) {

if (number % 2 == 0) {

number += 1;

}

});

print(numbers); // 輸出 [1, 3, 5, 5, 7, 7]

### 使用 forEach 累加值

您可以使用 forEach 遍歷中的累加器來累加值。例如，我們可以計算 numbers 清單中所有元素的平方和：

int sum = 0;

numbers.forEach((number) {

sum += number \* number;

});

print(sum); // 輸出 122

### 使用 forEach 提前終止遍歷

您可以使用 break 關鍵字來提前終止 forEach 遍歷。例如，我們可以找到 numbers 列表中第一個偶數並立即終止遍歷：

for (int number in numbers) {

if (number % 2 == 0) {

print(number); // 輸出 2

break;

}

}

Dart 中的 forEach 是一種反覆運算器,它可以用於遍歷集合類型的元素,如 List、Set 和 Map。除了基本使用方式外,forEach 還提供了一些高級用法,可以讓您的代碼更加簡潔和強大。以下是一些 forEach 的高級用法示例:

1. **中斷遍歷**

有時我們需要在滿足某些條件時中斷遍歷,這可以通過引入一個控制變數來實現。

List<int> numbers = [1, 2, 3, 4, 5];

bool shouldBreak = false;

numbers.forEach((number) {

if (number == 3) {

shouldBreak = true;

return;

}

print(number);

});

在這個示例中,我們使用了一個布爾變數 shouldBreak 來控制是否中斷遍歷。當遇到數字 3 時,我們將 shouldBreak 設置為 true 並返回,有效地中斷了遍歷。

1. **異步遍歷**

在處理異步操作時,我們可以使用 forEach 來遍歷集合中的元素,並對每個元素執行異步操作。

List<String> urls = ['https://example.com', 'https://google.com', 'https://dart.dev'];

urls.forEach((url) async {

try {

var response = await http.get(Uri.parse(url));

print('${url}: ${response.statusCode}');

} catch (e) {

print('Error: $e');

}

});

在這個示例中,我們使用了 async 和 await 關鍵字來處理異步 HTTP 請求。forEach 反覆運算器會等待每個請求完成後再繼續下一個請求。

1. **映射和過濾**

forEach 可以與其他集合操作結合使用,如 map 和 where。

List<int> numbers = [1, 2, 3, 4, 5];

numbers.where((number) => number % 2 == 0).map((number) => number \* 2).forEach(print);

*// 輸出: 4, 8*

在這個示例中,我們首先使用 where 過濾出偶數,然後使用 map 將每個偶數乘以 2,最後使用 forEach 列印結果。

1. **使用索引**

在某些情況下,我們需要訪問集合元素的索引。forEach 提供了一種方便的方式來獲取索引。

List<String> words = ['apple', 'banana', 'cherry'];

words.asMap().forEach((index, word) {

print('$index: $word');

});

*// 輸出: 0: apple, 1: banana, 2: cherry*

在這個示例中,我們使用 asMap 將 List 轉換為 Map,然後使用 forEach 反覆運算 Map 的鍵值對,鍵就是元素的索引。

### 總結

forEach 方法是 Dart 中一種強大的工具，可用於遍歷清單或集合中的每個元素。通過高級用法，您可以過濾、修改、累加值和提前終止遍歷，從而更靈活地操作和處理遍歷中的元素。

**switch/case 的高級用法**

1. **使用空 case 捕獲所有其他情況**

在 switch 語句的最後,可以使用一個空的 case 來捕獲所有其他未明確定義的情況。這樣可以確保代碼涵蓋了所有可能的情況,避免了意外錯誤。

String getOperationName(int operation) {

switch (operation) {

case 1:

return 'Addition';

case 2:

return 'Subtraction';

case 3:

return 'Multiplication';

case 4:

return 'Division';

default:

return 'Unknown operation';

}

}

1. **在 case 子句中使用範圍**

Dart 允許在 case 子句中使用範圍,這對於處理一組連續的值非常有用。

String getGradeDescription(int score) {

switch (score) {

case 90:

case 91:

case 92:

case 93:

case 94:

case 95:

case 96:

case 97:

case 98:

case 99:

case 100:

return 'Excellent';

case 80 ... 89:

return 'Good';

case 70 ... 79:

return 'Average';

case 60 ... 69:

return 'Poor';

default:

return 'Fail';

}

}

1. **在 case 子句中使用 when 子句**

在 Dart 中,可以在 case 子句中使用 when 子句來添加額外的條件檢查。這使得 switch 語句更加強大和靈活。

void printDayPeriod(int hour) {

switch (hour) {

case 5:

case 6:

case 7:

case 8:

case 9:

case 10:

case 11:

print('Morning');

break;

case 12:

case 13:

case 14:

case 15:

case 16:

print('Afternoon');

break;

case 17:

case 18:

case 19:

case 20:

case 21:

case 22:

print('Evening');

break;

case 23:

case 24:

case 1:

case 2:

case 3:

case 4:

when (hour < 6) {

print('Night');

} else {

print('Late Night');

}

break;

default:

print('Invalid hour');

}

}

在這個例子中,我們使用了 when 子句來區分淩晨時間和深夜時間。

1. **使用 continue 語句繼續執行下一個 case**

在 Dart 中,您可以使用 continue 語句繼續執行下一個 case 子句,而不是中斷整個 switch 語句。這在某些情況下非常有用。

void printNumberProperties(int number) {

switch (number) {

case 0:

print('Zero');

continue case 2;

case 1:

print('Odd');

continue case 3;

case 2:

print('Even');

break;

case 3:

print('Odd');

break;

default:

print('Number is $number');

}

}

在這個例子中,如果輸入的數字是 0,它將列印 "Zero" 和 "Even"。如果輸入的數字是 1,它將列印 "Odd"。

### 5. 使用多個 case 條件

switch/case 語句可以同時使用多個 case 條件來匹配運算式的值。例如，以下代碼使用多個 case 條件來根據字母將學生分配到不同的班級：

String getStudentClass(String letterGrade) {

switch (letterGrade) {

case 'A':

return 'Excellent';

case 'B':

return 'Good';

case 'C':

return 'Average';

case 'D':

return 'Below Average';

case 'F':

return 'Failing';

default:

return 'Invalid grade';

}

}

### 6. 使用範圍匹配

switch/case 語句可以使用範圍匹配來匹配運算式的值。例如，以下代碼使用範圍匹配來根據分數將學生分配到不同的等級：

String getStudentGrade(int score) {

switch (score) {

case 90..100:

return 'A';

case 80..89:

return 'B';

case 70..79:

return 'C';

case 60..69:

return 'D';

default:

return 'F';

}

}

### 7. 使用 fallthrough

fallthrough 關鍵字可用於指示下一個 case 條件塊繼續執行。例如，以下代碼使用 fallthrough 來為多個 case 條件塊執行相同的代碼：

void printMessage(String message) {

switch (message) {

case 'Hello':

case 'Hi':

print('Greetings!');

fallthrough; // Continue to the next case

case 'Goodbye':

print('Farewell!');

}

}

### 8. 使用 yield 返回值

yield 關鍵字可用於在 switch/case 語句中返回多個值。例如，以下代碼使用 yield 來生成一個斐波那契數列：

Iterable<int> fibonacci() sync\* {

int a = 0, b = 1;

while (true) {

yield a;

int temp = a;

a = b;

b = temp + a;

}

}

### 9. 使用 switch 運算式

switch 運算式可用於將 switch/case 語句的結果存儲在變數中。例如，以下代碼使用 switch 運算式來獲取學生成績的字母等級：

String getStudentGradeLetter(int score) {

var letterGrade = '';

switch (score) {

case 90..100:

letterGrade = 'A';

break;

case 80..89:

letterGrade = 'B';

break;

// ... other cases

}

return letterGrade;

}

**Dart 3.0 中 switch/case 的高級用法**

新功能:

1. **支援模式匹配**

Dart 3.0 引入了模式匹配功能,它可以在 case 語句中使用。這使得 switch 語句更加靈活,可以匹配更複雜的模式。

void printObject(Object obj) {

switch (obj) {

case int i:

print('Integer: $i');

break;

case String s:

print('String: $s');

break;

case [int, int] pair:

print('Pair: ${pair[0]}, ${pair[1]}');

break;

default:

print('Unknown object: $obj');

}

}

在這個例子中,我們使用模式匹配來匹配不同類型的對象,包括整數、字串和整數對。這種方式比傳統的類型檢查更加簡潔和易讀。

以下代碼使用模式匹配來根據清單中的元素類型列印消息：

void printListMessage(List<dynamic> list) {

switch (list) {

case List<int>:

print('List of integers');

break;

case List<String>:

print('List of strings');

break;

default:

print('List of unknown type');

}

}

1. **支持 switch 表達式**

Dart 3.0 還引入了 switch 表達式,它可以將整個 switch 語句視為一個表達式,並返回相應的值。這使得編寫簡潔的單行代碼變得更加容易。

String getGradeDescription(int score) => switch (score) {

90 <= score && score <= 100 => 'Excellent',

80 <= score && score < 90 => 'Good',

70 <= score && score < 80 => 'Average',

60 <= score && score < 70 => 'Poor',

\_ => 'Fail', *// \_ 匹配所有其他情況*

};

在這個例子中,我們使用了 switch 表達式來返回相應的成績描述。case 語句使用範圍匹配,\_ 用於匹配所有其他情況。

1. **支持 case null**

在 Dart 3.0 中,你可以使用 case null 來匹配 null 值。這對於處理可空類型非常有用。

String? nullableString = null;

String result = switch (nullableString) {

'hello' => 'Hello, world!',

null => 'Null value',

\_ => 'Other value: $nullableString',

};

print(result); *// 輸出: Null value*

### 4. 解構模式

解構模式允許您將模式匹配的結果分解為多個變數。這對於從複雜的資料結構中提取值非常有用。

例如，以下代碼使用解構模式從點物件中提取座標：

class Point {

int x;

int y;

Point(this.x, this.y);

}

void printPointCoordinates(Point point) {

switch (point) {

case Point(x, y):

print('Coordinates: ($x, $y)');

break;

}

}

### 5. 嵌套模式

嵌套模式允許您在單個 switch/case 語句中嵌套多個模式匹配運算式。這對於處理複雜的資料結構或多級條件邏輯非常有用。

例如，以下代碼使用嵌套模式來根據清單中的元素類型和值列印消息：

void printListMessageWithValues(List<dynamic> list) {

switch (list) {

case List<int> value:

print('List of integers: $value');

break;

case List<String> value:

print('List of strings: $value');

break;

default:

print('List of unknown type');

}

}

### 6. 關聯類型

關聯類型允許您將模式匹配與類型別名相結合，以提高代碼的可讀性和可維護性。

例如，以下代碼使用關聯類型來定義模式匹配運算式，用於匹配 Point 物件：

typedef PointPattern = Pattern<Point>;

void printPointCoordinatesWithPattern(Point point) {

switch (point) {

case PointPattern(x, y):

print('Coordinates: ($x, $y)');

break;

}

}

### 7. 類型推斷

Dart 3.0 的 switch/case 語句支援類型推斷，這意味著編譯器可以自動推斷模式匹配運算式的類型。這使得代碼更加簡潔易讀。

例如，以下代碼使用類型推斷來匹配 Point 物件：

void printPointCoordinatesWithTypeInference(Point point) {

switch (point) {

case Pattern<Point>(x, y):

print('Coordinates: ($x, $y)');

break;

}

}

**for loop 的高級用法**

Dart 中的 for 循環除了基本的用法外,還有一些高級用法,可以讓你的代碼更加簡潔、可讀和高效。以下是一些常見的高級用法:

1. **For-in 循環**

Dart 提供了 for-in 循環,它可以方便地遍歷可反覆運算對象,如清單(List)、集合(Set)和映射(Map)。

List<int> numbers = [1, 2, 3, 4, 5];

for (int number in numbers) {

print(number);

}

Map<String, int> ages = {'Alice': 25, 'Bob': 32, 'Charlie': 28};

for (MapEntry<String, int> entry in ages.entries) {

print('${entry.key}: ${entry.value}');

}

1. **For 循環中的解構**

Dart 支援在 for 循環中對可反覆運算對象進行解構,這可以讓你更方便地訪問元素的屬性或值。

List<Map<String, dynamic>> persons = [

{'name': 'Alice', 'age': 25},

{'name': 'Bob', 'age': 32},

{'name': 'Charlie', 'age': 28},

];

for (var person in persons) {

print('Name: ${person['name']}, Age: ${person['age']}');

}

*// 使用解構*

for (var {'name': name, 'age': age} in persons) {

print('Name: $name, Age: $age');

}

1. **For 循環中的條件**

你可以在 for 循環中添加條件語句,只遍歷滿足特定條件的元素。

List<int> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

*// 只遍歷偶數*

for (int number in numbers) {

if (number % 2 == 0) {

print(number);

}

}

*// 使用 where() 方法*

for (int number in numbers.where((n) => n % 2 == 0)) {

print(number);

}

1. **For 循環的索引訪問**

在某些情況下,你可能需要同時訪問元素及其索引。Dart 提供了一種簡潔的方式來實現這一點。

List<String> names = ['Alice', 'Bob', 'Charlie', 'David'];

*// 使用索引訪問*

for (int i = 0; i < names.length; i++) {

print('$i: ${names[i]}');

}

*// 使用 entries 方法*

for (var entry in names.asMap().entries) {

print('${entry.key}: ${entry.value}');

}

1. **For 循環的控制流**

在 for 循環中,你可以使用 break 和 continue 語句來控制循環的流程。

List<int> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

for (int number in numbers) {

if (number == 5) {

break; *// 跳出循環*

}

print(number);

}

for (int number in numbers) {

if (number % 2 == 0) {

continue; *// 跳過當前反覆運算*

}

print(number);

}

**while loop 的高級用法**

常見的高級用法:

1. **中斷循環**

在某些情況下,你可能需要根據特定條件提前中斷 while 循環。Dart 提供了 break 語句來實現這一點。

int number = 0;

bool isContinue = true;

while (isContinue) {

number++;

if (number > 10) {

isContinue = false;

break;

}

print(number);

}

1. **跳過當前迭代**

有時你可能需要跳過當前迭代,直接進入下一次迭代。Dart 提供了 continue 語句來實現這一點。

int number = 0;

while (number < 10) {

number++;

if (number % 2 == 0) {

continue;

}

print(number);

}

1. **標記循環**

如果你有嵌套的 while 循環,可以使用標記來指定要中斷或跳過哪一層循環。

outerLoop:

while (true) {

print('Outer loop');

int i = 0;

while (true) {

print('Inner loop $i');

if (i == 2) {

break outerLoop; *// 跳出外層循環*

}

i++;

}

}

1. **無限循環**

在某些情況下,你可能需要一個無限循環,直到滿足特定條件時才中斷。while 循環適合這種用法。

int number = 0;

String input = '';

while (true) {

print('Enter a number (or "q" to quit):');

input = stdin.readLineSync()!;

if (input == 'q') {

break;

}

number += int.parse(input);

print('Current total: $number');

}

1. **實現計時器或動畫**

通過結合 while 循環和 Future.delayed() 方法,你可以實現一個簡單的計時器或動畫。

import 'dart:async';

void main() {

int remainingSeconds = 10;

Timer? timer;

void startTimer() {

timer = Timer.periodic(Duration(seconds: 1), (\_) {

if (remainingSeconds == 0) {

timer?.cancel();

print('Time is up!');

} else {

print('$remainingSeconds seconds remaining');

remainingSeconds--;

}

});

}

startTimer();

}

## Dart 中的枚舉類型（Enum 1）

在 Dart 中,enum(枚舉)用於定義一組有限的命名常量。枚舉為編碼提供了更好的可讀性和類型安全性。以下是如何表示和使用 enum 的方式:

1. **定義枚舉**

enum Color { red, green, blue }

在這個例子中,我們定義了一個名為 Color 的枚舉,它有三個值分別是 red、green 和 blue。

1. **獲取和比較枚舉值**

var color = Color.red;

print(color); *// 輸出: Color.red*

if (color == Color.red) {

print('The color is red');

}

switch (color) {

case Color.red:

print('Red');

break;

case Color.green:

print('Green');

break;

case Color.blue:

print('Blue');

break;

}

我們可以直接使用枚舉值,也可以在條件語句和 switch 語句中使用枚舉進行比較和判斷。

1. **獲取枚舉索引和值**

print(Color.values); *// 輸出: [Color.red, Color.green, Color.blue]*

print(Color.red.index); *// 輸出: 0*

print(Color.blue.index); *// 輸出: 2*

每個枚舉值都有一個從 0 開始的索引值。我們可以通過 values 屬性獲取所有枚舉值的列表,也可以通過 index 屬性獲取枚舉值的索引。

1. **給枚舉值賦值**

enum Size { small = 10, medium = 20, large = 30 }

print(Size.small.index); *// 輸出: 10*

print(Size.medium.index); *// 輸出: 20*

print(Size.large.index); *// 輸出: 30*

在定義枚舉時,我們可以給每個枚舉值賦予一個特定的值,而不僅僅是使用默認的索引值。

1. **使用枚舉作為switch語句的case值**

void printSize(Size size) {

switch (size) {

case Size.small:

print('Small');

break;

case Size.medium:

print('Medium');

break;

case Size.large:

print('Large');

break;

}

}

printSize(Size.medium); *// 輸出: Medium*

枚舉非常適合用於 switch 語句的 case 值,因為它們是固定的常量值,可以提高代碼的可讀性和可維護性。

enum 的高級用法

Dart 中的 enum 不僅可以用於定義一組有限的命名常量,它還提供了一些高級用法,可以讓您在代碼中更好地利用枚舉。以下是一些 enum 的高級用法:

1. **枚舉類別擴展**

您可以為枚舉類別添加方法、getter 和其他實例變量,從而擴展枚舉的功能。

enum ColorType {

red(0xFFFF0000),

green(0xFF00FF00),

blue(0xFF0000FF);

const ColorType(this.value);

final int value;

int get rgb => value;

}

void main() {

print(ColorType.red.rgb); *// 輸出: 16711680*

}

在這個示例中,我們為 ColorType 枚舉添加了一個整數值 value,並定義了一個 rgb getter 來獲取該值。這樣,我們就可以通過枚舉值直接訪問 RGB 值。

1. **使用 extension 為枚舉添加功能**

您可以使用 Dart 的 extension 功能為枚舉添加更多方法和屬性。

enum Status { pending, approved, rejected }

extension StatusExtension on Status {

String get description {

switch (this) {

case Status.pending:

return 'Waiting for approval';

case Status.approved:

return 'Approved';

case Status.rejected:

return 'Rejected';

}

}

}

void main() {

print(Status.pending.description); *// 輸出: Waiting for approval*

print(Status.approved.description); *// 輸出: Approved*

print(Status.rejected.description); *// 輸出: Rejected*

}

在這個示例中,我們使用了 extension 為 Status 枚舉添加了一個 description getter,它根據枚舉值返回相應的描述字串。這樣做可以讓我們的代碼更加清晰易讀。

1. **枚舉映射**

您可以使用枚舉作為映射(Map)的鍵或值,從而實現更靈活的數據結構。

enum DayOfWeek { monday, tuesday, wednesday, thursday, friday, saturday, sunday }

Map<DayOfWeek, String> dayNames = {

DayOfWeek.monday: 'Monday',

DayOfWeek.tuesday: 'Tuesday',

*// ... 其他日期*

};

void main() {

print(dayNames[DayOfWeek.monday]); *// 輸出: Monday*

}

在這個示例中,我們使用了一個 Map,將 DayOfWeek 枚舉作為鍵,將日期名稱作為值,從而實現了一種更加清晰和易於維護的數據結構。

## Dart 中的枚舉類型（Enum 2）

枚舉類型（Enum）是一種用於表示一組固定值的特殊類。枚舉的每個值稱為枚舉成員。枚舉成員通常用大寫字母表示，以提高可讀性。

### 創建枚舉

要創建枚舉，請使用 enum 關鍵字，然後指定枚舉名稱和枚舉成員列表。枚舉成員之間用逗號分隔。

enum Color {

red,

green,

blue,

purple,

}

此代碼創建名為 Color 的枚舉，其中包含四個枚舉成員：red、green、blue 和 purple。

### 訪問枚舉成員

可以使用點號語法訪問枚舉成員。例如，要訪問 Color 枚舉的 red 成員，可以使用以下代碼：

Color myColor = Color.red;

此代碼將 Color.red 枚舉成員的值分配給 myColor 變數。

### 枚舉成員的值

每個枚舉成員都有一個與之關聯的整數值。預設情況下，第一個枚舉成員的值為 0，後續成員的值遞增 1。但是，您可以顯式指定每個成員的值。

enum Status {

pending = 0,

active = 1,

completed = 2,

}

在此示例中，pending 成員的值為 0，active 成員的值為 1，completed 成員的值為 2。

### 使用枚舉進行switch操作

可以使用 switch 語句根據枚舉成員的值執行不同的代碼塊。

enum Color {

red,

green,

blue,

purple,

}

Color myColor = Color.blue;

switch (myColor) {

case Color.red:

print("The color is red");

break;

case Color.green:

print("The color is green");

break;

case Color.blue:

print("The color is blue");

break;

case Color.purple:

print("The color is purple");

break;

}

此代碼將列印 "The color is blue"，因為 myColor 變數的值是 Color.blue。

### 枚舉的優點

使用枚舉有以下優點：

* 提高代碼的可讀性和可維護性。
* 防止使用無效值。
* 可以更容易地使用switch語句。

### 枚舉的局限性

枚舉也有一些局限性：

* 枚舉成員的值是固定的，不能在運行時更改。
* 枚舉的成員數量有限。

總的來說，枚舉是一種有用的工具，可用於提高 Dart 代碼的可讀性、可維護性和安全性。

### 枚舉的附加功能

Dart 中的枚舉提供了一些額外的功能，例如：

* **關聯值：** 每個枚舉成員都可以關聯一個值。例如：

enum Shape {

circle(double radius),

rectangle(double width, double height),

square(double sideLength),

}

* **方法：** 枚舉可以定義自己的方法。例如：

enum Shape {

circle(double radius) {

@override

double get area => PI \* radius \* radius;

},

rectangle(double width, double height) {

@override

double get area => width \* height;

},

square(double sideLength) {

@override

double get area => sideLength \* sideLength;

},

}

## Dart 中枚舉的高級用法

枚舉（enum）是 Dart 中一種用於表示一組固定值的特殊類。除了基本用法之外，枚舉在 Dart 中還提供了一些高級功能，可用於更複雜的操作。以下是一些高級用法的示例：

**1. 使用 const 關鍵字創建枚舉：**

const enum Color {

red,

green,

blue,

purple,

}

使用 const 關鍵字可以讓枚舉成為常量枚舉。這意味著枚舉成員的值在運行時不可更改。

**2. 使用 values 屬性獲取枚舉的所有成員：**

enum Color {

red,

green,

blue,

purple,

}

List<Color> allColors = Color.values;

此代碼將 Color 枚舉的所有成員存儲在 allColors 清單中。

**3. 使用 index 屬性獲取枚舉成員的索引：**

enum Color {

red,

green,

blue,

purple,

}

int redIndex = Color.red.index; // redIndex 將為 0

此代碼將 Color.red 枚舉成員的索引存儲在 redIndex 變數中。索引從 0 開始，每個後續成員的索引遞增 1。

**4. 使用 map 方法轉換枚舉成員：**

enum Color {

red,

green,

blue,

purple,

}

Map<Color, String> colorNames = Color.values.map((color) => MapEntry(color, color.toString())).toMap();

此代碼將 Color 枚舉的所有成員轉換為鍵值對的映射，其中鍵是枚舉成員，值是枚舉成員的字串表示形式。

**5. 使用 switch 語句與 when 子句：**

enum Status {

pending,

active,

completed,

}

Status myStatus = Status.active;

switch (myStatus) {

case Status.pending when myStatus.index == 0:

print("The status is pending and it's the first one");

break;

case Status.active:

print("The status is active");

break;

case Status.completed:

print("The status is completed");

break;

}

此代碼將列印 "The status is active"，因為 myStatus 變數的值是 Status.active。when 子句允許您在 switch 語句中執行額外的檢查。

**6. 使用枚舉擴展類：**

enum Shape {

circle,

rectangle,

square,

}

extension ShapeExtension on Shape {

double get area {

switch (this) {

case Shape.circle:

return PI \* radius \* radius;

case Shape.rectangle:

return width \* height;

case Shape.square:

return sideLength \* sideLength;

}

}

}

此代碼將 area 方法添加到 Shape 枚舉中。該方法將計算每個形狀的面積。

**7. 使用枚舉作為泛型類型：**

enum SortOrder {

ascending,

descending,

}

class Sorter<T> {

void sort(List<T> items, SortOrder order) {

// ... sorting logic based on order

}

}

此代碼創建了一個名為 Sorter 的泛型類，該類可以對不同類型的列表進行排序。SortOrder 枚舉用於指定排序順序。

這些只是 Dart 中枚舉高級用法的一些示例。枚舉提供了許多其他功能，可用於各種複雜任務。

**函數 的高級用法**

方法定義、箭頭函數、函數相互調用、匿名、自執行方法及閉包

方法定義

dart自訂方法的基本格式：

返回類型 方法名稱（參數1，參數2,...）{

方法體

return 返回值 / 或無返回值;

}

定義方法的的幾個例子：

void printInfo(){

print('我是一個自訂方法');

}

int getNum(){

var count = 123;

return count;

}

String printUserInfo(){

return 'this is str';

}

List getList(){

return ['111','2222','333'];

}

**Dart**沒有public 、private等關鍵字，\_ 下橫向直接代表 private。

#### 方法的作用域

void main(){

void outFun(){

innerFun(){

print('aaa');

}

innerFun();

}

// innerFun(); 錯誤寫法

outFun(); //調用方法

}

#### 方法傳參

**一般定義：**

String getUserInfo(String username, int age) {

//形參

return "姓名:$username -> 年齡:$age";

}

print(printUserInfo('小明', 23)); //實參

**Dart中可以定義一個帶可選參數的方法 ，可選參數需要指定類型預設值：**

void main() {

String printUserInfo(String username, [int age = 0]) { //age格式表示可選

//形參

if (age != 0) {

return "姓名:$username -> 年齡:$age";

}

return "姓名:$username -> 年齡不詳";

}

print(printUserInfo('小明', 28)); //實參

//可選就可以不傳了

print(printUserInfo('李四'));

}

**定義一個帶默認參數的方法：**

String getUserInfo(String username,[String sex='男',int age=0]){ //形參

if(age!=0){

return "姓名:$username -> 性別:$sex -> 年齡:$age";

}

return "姓名:$username -> 性別:$sex -> 年齡不詳";

}

print(getUserInfo('張三'));

print(getUserInfo('李四','男'));

print(getUserInfo('李梅梅','女',25));

**定義一個具名引數的方法，定義具名引數需要指定類型預設值：**

具名引數的好處是在使用時可以不用按順序賦值，看下面代碼：

String getUserInfo(String username, {int age = 0, String sex = '男'}) {//形參

if (age != 0) {

return "姓名:$username -> 性別:$sex -> 年齡:$age";

}

return "姓名:$username -> 性別:$sex -> 年齡保密";

}

print(getUserInfo('張三',sex: '男',age: 20));

**定義一個把方法當做參數的方法：**

其實就是方法可以當做參數來用，這點和Kotlin也是一樣的：

//方法1 隨便列印一下

fun1() {

print('fun1');

}

//方法2 參數是一個方法

fun2(fun) {

fun();

}

//調用fun2這個方法 把fun1這個方法當做參數傳入

fun2(fun1());

箭頭函數、函數相互調用

箭頭函數

使用forEach來遍歷List,其一般格式如下：

List list = ['a', 'b', 'c'];

list.forEach((value) {

print(value);

});

而箭頭函數就是可以簡寫這種格式：

list.forEach((value) => print(value));

箭頭後面指向的就是方法的返回值，這裡要注意的是：

箭頭函數內只能寫一條語句，並且語句後面沒有分號(;)

對於之前map轉換的例子也可以使用箭頭方法來簡化一下：

List list = [1, 3, 6, 8, 9];

var newList = list.map((value) {

if (value > 3) {

return value \* 2;

}

return value;

});

這裡就是修改List裡面的資料，讓陣列中大於3的值乘以2。那用箭頭函數簡化後可以寫成：

var newList = list.map((value) => value > 3 ? value\*2 : value);

一句代碼完成，非常有意思。

#### 函數相互調用

// 定義一個方法來判斷一個數是否是偶數

bool isEvenNumber(int n) {

if (n % 2 == 0) {

return true;

}

return false;

}

// 定義一個方法列印1-n以內的所有偶數

prinEvenNumber(int n) {

for (var i = 1; i <= n; i++) {

if (isEvenNumber(i)) {

print(i);

}

}

}

prinEvenNumber(10);

匿名、自執行方法及方法遞迴

匿名var printNum = (){

print(12);

};

printNum();

這裡很明顯跟Kotlin中的特性基本是一樣的。帶參數的匿名方法：

var printNum = (int n) {

print(n + 2);

};

printNum(3);

自執行方法

自執行方法顧名思義就是不需要調用，會自動去執行的，這是因為自執行函數的定義和調用合為了一體。當我們創建了一個匿名函數，並執行了它，由於外部無法引用的它的內部變數，所以在執行完就會很快被釋放，而且這種做法不會污染到全域物件。看如下代碼：

((int n) {

print("這是一個自執行方法 + $n");

})(666);

}

#### 方法遞迴

方法的遞迴無非就是在條件滿足的條件下繼續在方法內調用自己本身，看以下代碼：

var sum = 0;

void fn(int n) {

sum += n;

if (n == 0) {

return;

}

fn(n - 1);

}

fn(100);

print(sum);

實現的是1加到100。

閉包

閉包是一個前端的概念，用戶端開發早期使用Java可以說是不支援閉包，或是不完整的閉包，但Kotlin是可以支援閉包的操作。

閉包的意思就是**函數嵌套函數, 內建函式會調用外部函數的變數或參數, 變數或參數不會被系統回收(不會釋放記憶體)**。所以閉包解決的兩個問題是：

* 變數常駐記憶體
* 變數不污染全域

閉包的一般寫法是：

* **函數嵌套函數，並return 裡面的函數，這樣就形成了閉包**。

閉包的寫法：

Function func() {

var a = 1; /\*不會污染全域 常駐記憶體\*/

return () {

a++;

print(a);

};

}

這裡return匿名方法後，a的值就可以常駐記憶體了：

var mFun = func();

mFun();

mFun();

mFun();

列印：2、3、4。

**函數 的高級用法**

常見的高級用法:

1. **匿名函數**

Dart 支持匿名函數,也稱為 lambda 表達式。這種函數沒有名稱,通常用作回調或作為其他函數的參數。

List<int> numbers = [5, 2, 4, 1, 3];

numbers.sort((a, b) => a - b);

print(numbers); // Output: [1, 2, 3, 4, 5]

var list = [1, 2, 3, 4, 5];

var doubledList = list.map((num) => num \* 2).toList();

print(doubledList); *// [2, 4, 6, 8, 10]*

1. **箭頭語法**

Dart 提供了一種簡潔的箭頭語法來定義簡單的函數。如果函數體只有一個表達式,可以省略 return 關鍵字和花括號。

List<int> numbers = [5, 2, 4, 1, 3];

numbers.sort((a, b) => a - b);

print(numbers); // Output: [1, 2, 3, 4, 5]

int add(int a, int b) => a + b;

print(add(2, 3)); *// 5*

1. **可選參數**

Dart 支援可選的位置參數、命名參數和默認參數值。這可以讓你的函數更加靈活和易用。

void greetPerson({String name = 'Guest', int age}) {

print('Hello, $name! You are ${age ?? 0} years old.');

}

您可以使用以下方式調用此函數：

greetPerson(name: 'Alice', age: 30); // 具名引數

greetPerson(name: 'Bob'); // 省略可選參數

void greet(String name, {String? greeting, int age = 30}) {

greeting ??= 'Hello';

print('$greeting, $name. You are $age years old.');

}

greet('Alice'); *// Hello, Alice. You are 30 years old.*

greet('Bob', greeting: 'Hi', age: 25); *// Hi, Bob. You are 25 years old.*

1. **函數作為參數**

在 Dart 中,函數是一等公民,可以像其他對象一樣被傳遞和操作。你可以將一個函數作為另一個函數的參數。

int applyOperation(int a, int b, int Function(int, int) operation) {

return operation(a, b);

}

int add(int a, int b) => a + b;

int subtract(int a, int b) => a - b;

print(applyOperation(2, 3, add)); *// 5*

print(applyOperation(5, 3, subtract)); *// 2*

1. **高階函數**

Dart 提供了許多內置的高階函數,如 map、where、reduce 等,可以方便地對集合進行操作。你也可以自己定義高階函數。

以下代碼定義了一個高階函數，用於對清單中的每個元素應用函數：

void applyFunction<T>(List<T> list, Function<T, T> f) {

for (T item in list) {

list[list.indexOf(item)] = f(item);

}

}

List<int> numbers = [1, 2, 3, 4, 5];

applyFunction(numbers, (x) => x \* 2);

print(numbers); // Output: [2, 4, 6, 8, 10]

List<int> applyOperation(int Function(int) operation, Iterable<int> numbers) {

return numbers.map(operation).toList();

}

int double(int x) => x \* 2;

int square(int x) => x \* x;

var numbers = [1, 2, 3, 4, 5];

print(applyOperation(double, numbers)); *// [2, 4, 6, 8, 10]*

print(applyOperation(square, numbers)); *// [1, 4, 9, 16, 25]*

1. **函數作用域:閉包(Closure)**

Dart 中的函數可以嵌套定義,內部函數可以訪問外部函數的變量和參數。這種特性稱為閉包(Closure)。

閉包是一種可以訪問其外部作用域中的變數的函數，即使該作用域已經完成執行。閉包通常用於創建具有記憶功能的函數。

例如，以下代碼使用閉包來創建計數器函數：

int counter() {

int count = 0;

return () {

count++;

return count;

};

}

var c = counter();

print(c()); // Output: 1

print(c()); // Output: 2

print(c()); // Output: 3

Function counter() {

int count = 0;

return () {

count++;

return count;

};

}

var increment = counter();

print(increment()); *// 1*

print(increment()); *// 2*

print(increment()); *// 3*

**7. 遞迴函數**

遞迴函數是一種在函數內部調用自身的功能。遞迴函數通常用於解決具有自相似結構的問題。

例如，以下代碼使用遞迴函數來計算斐波那契數列：

int fibonacci(int n) {

if (n == 0 || n == 1) {

return n;

} else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

print(fibonacci(10)); // Output: 55

**可變參數函示(1)**

在 Dart 中,可以使用可變參數來定義一個函數,該函數可以接受不確定數量的參數。有以下幾種方式來表示可變參數:

1. **位置可變參數**

使用方括號 [] 來表示位置可變參數。這種參數可以接受任意數量的參數值,並將它們存儲為一個列表。

void printNumbers(String label, [List<int>? numbers]) {

print(label);

if (numbers != null) {

numbers.forEach(print);

}

}

printNumbers('Numbers: '); *// 輸出: Numbers:*

printNumbers('Numbers: ', [1, 2, 3]); *// 輸出: Numbers: 1 2 3*

1. **命名可變參數**

使用大括號 {} 來表示命名可變參數。這種參數可以接受任意數量的命名參數,並將它們存儲為一個映射(Map)。

void printInfo({String? name, int? age}) {

print('Name: $name');

print('Age: $age');

}

printInfo(name: 'John', age: 30); *// 輸出: Name: John Age: 30*

printInfo(age: 25); *// 輸出: Name: null Age: 25*

1. **同時使用位置和命名可變參數**

可以在同一個函數定義中同時使用位置和命名可變參數。但是位置可變參數必須在命名可變參數之前。

void printData(String label, [List<int>? numbers, {bool? isDescending}]) {

print(label);

if (numbers != null) {

if (isDescending == true) {

numbers.sort((a, b) => b.compareTo(a));

}

numbers.forEach(print);

}

}

printData('Numbers: ', [3, 1, 2]); *// 輸出: Numbers: 3 1 2*

printData('Numbers: ', [3, 1, 2], isDescending: true); *// 輸出: Numbers: 3 2 1*

在上述示例中，printData 函數接受一個標籤字符串、一個可選的整數列表和一個可選的命名參數 isDescending。如果提供了 isDescending 且為 true，則對列表進行降序排序。

使用可變參數可以增強函數的靈活性,允許根據不同的情況傳遞不同數量的參數。但是,在使用可變參數時,也要注意檢查傳入的參數是否為 null,以避免發生錯誤。

## Dart 中可變參數的高級用法

Dart 中的可變參數是一個非常強大的特性,它使函數可以接受任意數量的參數。除了基本的使用方式之外,可變參數還有一些高級用法,可以幫助您編寫更加靈活和強大的代碼。以下是一些可變參數的高級用法:

1. **使用可變參數進行函數柯裡化**

函數柯裡化是一種將多參數函數轉換為一系列單參數函數的技術。在 Dart 中,可以利用可變參數和匿名函數來實現函數柯裡化。

Function sum = (int a, [int b = 0, int c = 0]) => a + b + c;

var curriedSum = ([int a = 0, int b = 0, int c = 0]) {

return sum(a, b, c);

};

print(curriedSum(1, 2, 3)); *// 輸出: 6*

var add2 = curriedSum(2);

print(add2(3, 4)); *// 輸出: 9*

1. **使用可變參數實現函數重載**

在 Dart 中,可以使用可變參數來模擬函數重載的行為。

int add(int a, int b) => a + b;

int add(int a, [int b = 0, int c = 0, int d = 0]) {

return a + b + c + d;

}

print(add(1, 2)); *// 輸出: 3*

print(add(1, 2, 3, 4)); *// 輸出: 10*

1. **使用可變參數進行異常處理**

可變參數可以用於異常處理,例如將異常作為參數傳遞給一個自定義的異常處理函數。

void handleException(Function function, [List<Object>? args]) {

try {

if (args != null) {

function(\*args);

} else {

function();

}

} catch (e) {

print('Exception caught: $e');

}

}

void myFunction(int a, int b) {

if (a == 0 || b == 0) {

throw ArgumentError('Arguments cannot be zero');

}

print(a / b);

}

handleException(myFunction, [4, 2]); *// 輸出: 2.0*

handleException(myFunction, [4, 0]); *// 輸出: Exception caught: Invalid argument(s): Arguments cannot be zero*

1. **使用可變參數實現動態參數**

可變參數可以用於實現動態參數的功能,允許函數在運行時接受不確定數量和類型的參數。

dynamic callFunction(String name, [dynamic args]) {

switch (name) {

case 'sum':

if (args is List<int>) {

return args.reduce((a, b) => a + b);

}

break;

case 'multiply':

if (args is List<int>) {

return args.reduce((a, b) => a \* b);

}

break;

*// 其他函數...*

}

throw ArgumentError('Invalid function name or arguments');

}

print(callFunction('sum', [1, 2, 3, 4, 5])); *// 輸出: 15*

print(callFunction('multiply', [2, 3, 4])); *// 輸出: 24*

可變參數的高級用法可以幫助您編寫更加靈活、模組化和可擴展的代碼。

**可變參數函示(2)**

在 Dart 中，可變參數函數是指可以接收數量可變的參數的函數。可變參數函數使用三個點 ... 來表示。可變參數函數只能有一個，並且必須是參數列表中的最後一個參數。

**可變參數函數的語法**

可變參數函數的語法如下：

void functionName(parameter1, parameter2, ..., parameterN, [...restParameters]) {

// 函數體

}

其中：

* parameter1, parameter2 等是必需參數
* ...restParameters 是可變參數
* 函數體 是函數的程式碼

**可變參數函數的使用**

可變參數函數可以用於以下情況：

* 處理數量可變的輸入
* 創建可接受不同數量參數的通用函數

**可變參數函數的用法示例**

以下是一些可變參數函數的用法示例：

**處理數量可變的輸入**

void printNumbers(num number1, [num? number2, num? number3]) {

print(number1);

if (number2 != null) {

print(number2);

}

if (number3 != null) {

print(number3);

}

}

printNumbers(1); // Output: 1

printNumbers(1, 2); // Output: 1 2

printNumbers(1, 2, 3); // Output: 1 2 3

在這個示例中，函數 printNumbers 接受一個必需參數 number1 和兩個可變參數 number2 和 number3。函數 printNumbers 首先打印 number1，然後檢查 number2 和 number3 是否為空。如果非空，則打印它們。

**創建可接受不同數量參數的通用函數**

List<num> sumNumbers(List<num> numbers) {

return numbers.map((num number) => number).toList();

}

List<num> numbers1 = [1, 2, 3];

List<num> numbers2 = [4, 5, 6, 7];

List<num> sum1 = sumNumbers(numbers1);

List<num> sum2 = sumNumbers(numbers2);

print(sum1); // Output: [1, 2, 3]

print(sum2); // Output: [4, 5, 6, 7]

在這個示例中，函數 sumNumbers 接受一個列表參數 numbers。函數 sumNumbers 將 numbers 列表中的每個數字都映射到它自己，然後返回結果列表。函數 sumNumbers 可以用於對不同長度的數字列表進行求和。

**可變參數函數的優點**

可變參數函數具有以下優點：

* 可以使程式碼更靈活，因為它們可以處理數量可變的輸入。
* 可以使程式碼更簡潔，因為無需為每個可能的輸入數量創建單獨的函數。
* 可以提高程式碼的可重用性，因為可變參數函數可用於多種情況。

**可變參數函數的局限性**

可變參數函數也有一些局限性，例如：

* 可變參數函數可能會使程式碼更難閱讀，尤其是當可變參數函數很長或很複雜時。
* 可變參數函數可能不如具有固定數量參數的函數那麼高效。

**總體而言，可變參數函數是一種強大的工具，可用於使 Dart 程式碼更靈活、簡潔和可重用。但是，在使用可變參數函數時，也應該考慮其局限性。**

## Dart 中可變參數的高級用法

可變參數（Variadic Parameter）是 Dart 中的一種特殊參數，它允許函數接收數量不定的參數。這使得函數更加靈活，因為它可以適應不同的參數數量。

### 基本用法

在 Dart 中，可變參數可以使用三個點 ... 來表示。可變參數必須位於必選參數和可選參數之後。

例如，我們可以創建一個函數來計算任意數量數位的平方和：

double sumSquares(double a, ...double numbers) {

double sum = a \* a;

for (double number in numbers) {

sum += number \* number;

}

return sum;

}

double result = sumSquares(1, 2, 3, 4, 5);

print(result); // 輸出 55

在這個例子中，sumSquares() 函數接收一個必選參數 a 和任意數量的可變參數 numbers。函數將 a 和 numbers 中所有數字的平方相加，並返回結果。

### 使用可變參數的最佳實踐

可變參數可以使您的代碼更加靈活，但也要注意以下最佳實踐：

* **避免過度使用可變參數**：過度使用可變參數會使您的代碼難以閱讀和理解。如果您發現自己經常使用可變參數，請考慮使用更具描述性的函數或參數清單。
* **使用具名引數**：具名引數可以使您的代碼更易於閱讀，尤其是當您有多個可變參數時。例如，您可以使用以下函數來計算任意數量數位的平均值：

double average(double a, {required List<double> numbers}) {

double sum = a;

for (double number in numbers) {

sum += number;

}

return sum / (1 + numbers.length);

}

* **使用默認參數**：默認參數可以使您的代碼更具可讀性和易用性。例如，您可以使用以下函數來計算任意數量數位的平方和，並提供可選的預設值：

double sumSquares(double a, [double defaultValue = 0]) {

double sum = a \* a;

for (double number in numbers) {

sum += number \* number;

}

return sum;

}

**匿名函數(1)**

在 Dart 中,匿名函數是一種沒有名稱的函數,通常用作一次性使用或作為參數傳遞給其他函數。它們非常靈活,可以簡化代碼並提高可讀性。以下是一些示例說明匿名函數的用法:

1. **簡單匿名函數**

var greet = () {

print('Hello!');

};

greet(); *// 輸出: Hello!*

在這個例子中,我們定義了一個簡單的匿名函數並將其賦值給變量 greet。然後我們可以通過調用 greet() 來執行該匿名函數。

1. **帶參數的匿名函數**

var multiply = (int a, int b) {

return a \* b;

};

print(multiply(3, 4)); *// 輸出: 12*

這裡我們定義了一個接受兩個整數參數並返回它們乘積的匿名函數。

1. **簡潔語法**

如果匿名函數只有一個語句,可以使用簡潔語法:

var add = (int a, int b) => a + b;

print(add(2, 3)); *// 輸出: 5*

使用 => 語法可以更加簡潔地定義匿名函數。

1. **作為參數傳遞**

匿名函數通常作為參數傳遞給其他函數:

void calculate(int a, int b, Function operation) {

print(operation(a, b));

}

calculate(4, 5, (a, b) => a + b); *// 輸出: 9*

calculate(4, 5, (a, b) => a \* b); *// 輸出: 20*

在這個例子中,我們定義了一個 calculate 函數,它接受兩個整數和一個 Function 類型的參數。我們可以將匿名函數作為最後一個參數傳遞給 calculate,以執行不同的操作。

總的來說,匿名函數非常靈活,可以簡化代碼並提高可讀性。它們常用於作為高階函數的參數或作為一次性使用的函數。適當使用匿名函數可以使您的 Dart 代碼更加簡潔和富有表現力。

**匿名函數(2)**

在 Dart 中，匿名函數是一種沒有名稱的函數。它可以用於簡化程式碼並提高可讀性。匿名函數通常用於以下情況：

* 將簡單的函數分配給變數
* 作為參數傳遞給其他函數
* 作為回傳值

**匿名函數的語法**

匿名函數的語法如下：

(parameter1, parameter2, ...) => expression;

其中：

* parameter1, parameter2 等是函數的参数
* expression 是函數的函數體

**匿名函數的用法**

以下是一些匿名函數的用法示例：

**將匿名函數分配給變數**

var square = (num x) => x \* x;

print(square(5)); // Output: 25

在這個示例中，匿名函數 (num x) => x \* x 被分配給變數 square。然後，變數 square 就像一個普通函數一樣使用，用於計算數字的平方。

**將匿名函數作為參數傳遞給其他函數**

void forEach(List<num> numbers, void Function(num) action) {

for (num number in numbers) {

action(number);

}

}

List<num> numbers = [1, 2, 3, 4, 5];

forEach(numbers, (num number) => print(number \* 2));

在這個示例中，匿名函數 (num number) => print(number \* 2) 作為參數傳遞給函數 forEach。函數 forEach 遍歷數組 numbers，並將每個數字傳遞給匿名函數。匿名函數對每個數字進行平方並打印結果。

**將匿名函數作為回傳值**

num max(num a, num b) {

return a > b ? a : b;

}

num compare(num a, num b) {

return max(a, b) - min(a, b);

}

print(compare(10, 20)); // Output: 10

在這個示例中，匿名函數 (num a, num b) => max(a, b) - min(a, b) 作為函數 compare 的回傳值。函數 compare 計算兩個數字之間的差，使用匿名函數來計算兩個數字的最大值和最小值。

**匿名函數的優點**

匿名函數具有以下優點：

* 可以簡化程式碼，使程式碼更易於閱讀和理解。
* 可以提高程式碼的可重用性，因為匿名函數可以很容易地傳遞給其他函數或存儲在變數中。
* 可以增強程式碼的靈活性，因為匿名函數可以根據需要進行自定義。

**匿名函數的局限性**

匿名函數也有一些局限性，例如：

* 匿名函數通常比命名函數更難調試，因為它們沒有名稱。
* 匿名函數可能會使程式碼更難閱讀，尤其是當匿名函數很長或很複雜時。
* 匿名函數可能不如命名函數那麼高效，尤其是當匿名函數被頻繁調用時。

**總體而言，匿名函數是一種強大的工具，可用於簡化 Dart 程式碼並提高其可讀性。但是，在使用匿名函數時，也應該考慮其局限性。**

**回調函數 的高级用法**

回調函數是一種在完成特定任務或操作後通知其他部分程式碼的設計模式。它是一種將控制流程從一個部分程式碼轉移到另一部分程式碼的有效方法。在 Dart 中，回調函數通常用於非同步操作，例如加載數據或處理用戶輸入。

以下是 Dart 中回調函數的一些高級用法：

**1. 使用回調函數實現鏈式調用**

鏈式調用是一種將多個方法調用連接在一起的技術，使代碼更具可讀性和可維護性。在 Dart 中，可以使用回調函數來實現鏈式調用。例如，以下代碼演示了如何使用回調函數來創建一個鏈式調用，用於加載數據並更新 UI：

void loadData(Function callback) {

// 模擬加載數據

Future.delayed(Duration(seconds: 2), () {

String data = '{"name": "John Doe", "age": 30}';

callback(data);

});

}

void updateUI(String data) {

// 更新 UI 界面

print('更新 UI 界面：$data');

}

void main() {

loadData((data) => updateUI(data));

}

**2. 使用回調函數處理錯誤**

回調函數可用於處理錯誤和其他異常情況。例如，以下代碼演示了如何使用回調函數來處理加載數據時發生的錯誤：

void loadData(Function callback, Function onError) {

// 模擬加載數據

Future.delayed(Duration(seconds: 2), () {

if (Random().nextBool()) {

String data = '{"name": "John Doe", "age": 30}';

callback(data);

} else {

onError('加載數據失敗');

}

});

}

void updateUI(String data) {

// 更新 UI 界面

print('更新 UI 界面：$data');

}

void handleError(String error) {

// 處理錯誤

print('錯誤：$error');

}

void main() {

loadData(updateUI, handleError);

}

**3. 使用回調函數實現自定義事件**

回調函數可用於實現自定義事件。例如，以下代碼演示了如何使用回調函數來創建一個自定義事件，用於表示按鈕被按下：

class Button {

Function? onPress;

void press() {

if (onPress != null) {

onPress!();

}

}

}

void handleButtonPress() {

// 處理按鈕按下事件

print('按鈕被按下');

}

void main() {

Button button = Button();

button.onPress = handleButtonPress;

button.press();

}

**4. 使用回調函數實現插件系統**

回調函數可用於實現插件系統。例如，以下代碼演示了如何使用回調函數來創建一個插件系統，用於擴展應用程式的功能：

class Plugin {

Function? onInit;

Function? onRun;

void init() {

if (onInit != null) {

onInit!();

}

}

void run() {

if (onRun != null) {

onRun!();

}

}

}

class App {

List<Plugin> plugins = [];

void loadPlugins() {

// 載入插件

plugins.add(Plugin()..onInit = () => print('插件初始化'));

plugins.add(Plugin()..onRun = () => print('插件運行'));

}

void run() {

// 運行應用程式

for (Plugin plugin in plugins) {

plugin.init();

plugin.run();

}

}

}

void main() {

App app = App();

app.loadPlugins();

app.run();

}

常見的高級用法:

1. **使用匿名函數作為參數**
2. *// 定義 Surface 類別*
3. class Surface {
4. *// 定義回調函數 onTouch*
5. Function(double, double)? onTouch;
6. *// 定義 touch 方法*
7. void touch(double x, double y) {
8. if (onTouch != null) {
9. onTouch!(x, y);
10. }
11. }
12. }
13. *// main 函數*
14. void main() {
15. *// 創建 Surface 實例*
16. Surface surface = Surface();
17. *// 傳入匿名函數作為 onTouch 回調*
18. surface.onTouch = (double x, double y) {
19. print('x: $x, y: $y');
20. };
21. *// 調用 touch 方法*
22. surface.touch(202.3, 134.0);
23. }

將匿名函數作為參數傳遞給回調函數。這種方式可以使您的代碼更加簡潔,避免了定義單獨的函數。

1. **使用 Lambda 函數**

Dart 支持 Lambda 函數,這是一種簡化的匿名函數語法。Lambda 函數可以使回調函數更加簡潔,例如:

myFunction.doSomething((x, y) => print('$x, $y'));

1. **使用閉包**

閉包是一種特殊的匿名函數,它可以訪問其作用域之外的變數。這使得回調函數可以記住和修改外部變數的狀態,從而實現更複雜的功能。

void makeAdder(int addBy) {

final adder = (int value) => value + addBy;

return adder;

}

void main() {

final add5 = makeAdder(5);

final add10 = makeAdder(10);

print(add5(3)); *// 輸出: 8*

print(add10(3)); *// 輸出: 13*

}

1. **回調鏈**

在某些情況下,您可能需要在一個操作完成後執行另一個操作。這可以通過將一個回調函數作為參數傳遞給另一個回調函數來實現,稱為回調鏈。

void operation1(Function callback) {

print('Operation 1 completed');

callback();

}

void operation2(Function callback) {

print('Operation 2 completed');

callback();

}

void operation3() {

print('Operation 3 completed');

}

void main() {

operation1(() {

operation2(() {

operation3();

});

});

}

1. **使用 Future 和 async/await**

在處理異步操作時,回調函數可以與 Future 和 async/await 語法相結合,使代碼更加清晰易讀。

Future<String> fetchData() async {

*// 模擬一個異步操作*

await Future.delayed(Duration(seconds: 2));

return 'Data fetched';

}

void main() async {

final data = await fetchData();

print(data); *// 輸出: "Data fetched"*

}

這些只是 Dart 中回調函數的一些高級用法。通過巧妙地使用匿名函數、Lambda 函數、閉包、回調鏈和異步編程,您可以編寫出更加強大和靈活的代碼。

**Dart 中 Function 的高级用法**

Dart 中的 Function 是一等公民,這意味著函數可以作為參數傳遞給其他函數,也可以作為返回值返回。這種特性使得函數在 Dart 中具有了很多高級用法。以下是一些 Function 的高級用法:

1. **高階函數**

高階函數是指將一個函數作為參數傳遞給另一個函數,或者返回一個新的函數。這種編程風格可以提高代碼的抽象層次和可重用性。

Function applyOperation(Function operation, int a, int b) {

return operation(a, b);

}

int add(int a, int b) => a + b;

int subtract(int a, int b) => a - b;

print(applyOperation(add, 3, 2)); *// 輸出: 5*

print(applyOperation(subtract, 3, 2)); *// 輸出: 1*

在這個示例中,applyOperation 是一個高階函數,它接受一個函數作為參數,並對其進行調用。我們可以將不同的函數作為參數傳遞給 applyOperation。

1. **閉包**

閉包是指一個函數可以記住並訪問它被創建時所在的作用域。這個特性使得函數可以"攜帶"一些數據,從而實現更加靈活的功能。

Function makeCounter() {

int count = 0;

Function increment = () {

count++;

return count;

};

return increment;

}

var counter = makeCounter();

print(counter()); *// 輸出: 1*

print(counter()); *// 輸出: 2*

print(counter()); *// 輸出: 3*

在這個示例中,makeCounter 函數返回一個閉包 increment。每次調用 increment 時,它都記住了上一次調用時 count 的值,並對其進行遞增。

Function(int x) makeAdder(int addBy) {

return (int y) => print(addBy + y) ;

}

void main() {

final a = makeAdder(2);

a(5);

print(a(5));

1. **箭頭函數**

Dart 支持使用箭頭語法 (=>) 定義簡潔的單行函數,這有助於提高代碼的可讀性。

var square = (int n) => n \* n;

print(square(3)); *// 輸出: 9*

1. **匿名函數**

匿名函數是指沒有名字的函數,它們通常作為臨時函數使用,可以作為參數傳遞給其他函數或者作為回調函數。

List<int> numbers = [1, 2, 3, 4, 5];

numbers.forEach((n) => print(n \* n));

*// 輸出: 1, 4, 9, 16, 25*

在這個示例中,我們使用了一個匿名函數作為 forEach 的回調函數,對每個元素進行平方運算並打印結果。

1. **函數式編程**

Dart 支持函數式編程的概念和技術,如高階函數、閉包和不可變數據。這種編程風格可以提高代碼的可讀性、可測試性和可維護性。

var numbers = [1, 2, 3, 4, 5];

var doubledNumbers = numbers.map((n) => n \* 2).toList();

print(doubledNumbers); *// 輸出: [2, 4, 6, 8, 10]*

在這個示例中,我們使用了 map 高階函數和匿名函數來實現對列表中每個元素的映射操作,生成一個新的列表。

總之,Dart 中的 Function 提供了許多強大的高級用法,如高階函數、閉包、箭頭函數和匿名函數等。

**Dart 中的 filter 函數**

filter 函數是 Dart 中一個功能強大的函數，可用於根據指定條件從列表中選擇元素。它通常與 where 子句結合使用來定義複雜的篩選條件。以下是一些探索 Dart 中高級 filter 用法的技巧：

**1. 連接 filter 操作**

您可以連接多個 filter 操作來將多個篩選條件應用於列表。例如，考慮一個包含產品及其價格和類別的列表：

List<Product> products = [

Product('Laptop', 1200, 'Electronics'),

Product('Book', 20, 'Books'),

Product('Phone', 600, 'Electronics'),

Product('Shirt', 50, 'Clothing'),

];

要篩選價格低於 500 美元的產品並屬於“電子產品”類別，您可以連接 filter 操作：

List<Product> filteredProducts = products

.filter((product) => product.price < 500)

.filter((product) => product.category == 'Electronics');

print(filteredProducts); // 輸出： [Product(Laptop, 1200, Electronics)]

**2. 使用 where 子句進行複雜篩選**

where 子句提供了一種更具表現力的方式來定義篩選條件。它允許您使用邏輯運算符、比較運算符和函數調用來創建複雜的條件。例如，要篩選價格在 200 美元到 500 美元之間或屬於“電子產品”或“書籍”類別的產品，您可以使用 where 子句：

List<Product> filteredProducts = products.where((product) {

return product.price >= 200 && product.price <= 500 ||

product.category == 'Electronics' ||

product.category == 'Books';

});

print(filteredProducts); // 輸出： [Product(Laptop, 1200, Electronics), Product(Book, 20, Books), Product(Phone, 600, Electronics)]

**3. 使用 null 值進行篩選**

您可以在 filter 或 where 子句中使用 null 檢查來處理 null 值。例如，要篩選價格非 null 的產品，您可以使用：

List<Product> filteredProducts = products.where((product) => product.price != null);

print(filteredProducts); // 輸出： [Product(Laptop, 1200, Electronics), Product(Phone, 600, Electronics)]

**4. 使用自定義函數進行篩選**

您可以將自定義函數傳遞給 filter 或 where 來應用更複雜的篩選邏輯。例如，要篩選名稱以字母“P”開頭的產品，您可以定義自定義函數並使用它：

bool startsWithP(String name) => name.startsWith('P');

List<Product> filteredProducts = products.where((product) => startsWithP(product.name));

print(filteredProducts); // 輸出： [Product(Phone, 600, Electronics)]

**5. 使用 map 和 expand 進行篩選**

您可以將 filter 與 map 和 expand 結合使用來變換和篩選數據結構。例如，要篩選產品名稱並將其轉換為大寫，您可以使用：

List<String> filteredNames = products

.map((product) => product.name)

.filter((name) => name.length > 5) // 篩選長度大於 5 個字元的姓名

.map((name) => name.toUpperCase()) // 將姓名轉換為大寫

.toList();

print(filteredNames); // 輸出： [LAPTOP, PHONE]

這些高級技術展示了 Dart 中 filter 函數的多功能性，使您能夠執行複雜的數據篩選和操作任務。

**filter 高級用法**

使用 where 方法和匿名函數

where 方法可以接受一個匿名函數作為參數,該函數返回一個布林值,用於決定元素是否保留。這種寫法比使用 filter 更加簡潔:

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var evenNumbers = numbers.where((n) => n.isEven);

print(evenNumbers); // (2, 4, 6, 8, 10)

使用級聯運算符

Dart 支援級聯運算符 (..)，可以將多個操作串接在一起。這樣可以使代碼更加簡潔和易讀:

var people = [

{'name': 'Alice', 'age': 25},

{'name': 'Bob', 'age': 30},

{'name': 'Charlie', 'age': 35},

{'name': 'David', 'age': 40},

];

var youngPeople = people

.where((person) => person['age']! < 30)

.map((person) => person['name'])

.toList();

print(youngPeople); // ['Alice']

使用 takeWhile 和 skipWhile

takeWhile 和 skipWhile 方法可以根據條件來獲取或跳過集合中的元素,直到條件不再滿足為止。例如:

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var lessThanFive = numbers.takeWhile((n) => n < 5); // [1, 2, 3, 4]

var moreThanFive = numbers.skipWhile((n) => n < 5); // [5, 6, 7, 8, 9, 10]

使用 any 和 every

any 和 every 方法可以檢查集合中是否至少有一個元素滿足條件(any)或者所有元素都滿足條件(every)。這在做一些驗證或檢查時非常有用:

var numbers = [1, 3, 5, 7, 9];

var hasEven = numbers.any((n) => n.isEven); // false

var allOdd = numbers.every((n) => n.isOdd); // true

使用 fold 和 reduce

fold 和 reduce 方法可以對集合進行歸納操作,將集合中的所有元素組合成一個值。fold 允許指定初始值,而 reduce 則使用集合的第一個元素作為初始值:

var numbers = [1, 2, 3, 4, 5];

var sum = numbers.fold(0, (prev, n) => prev + n); // 15

var product = numbers.reduce((prev, n) => prev \* n); // 120

總的來說,Dart 的 filter 以及相關的集合操作方法非常強大和靈活,可以讓我們以更加簡潔和高效的方式對集合進行各種操作和轉換,是 Dart 編程中不可或缺的一部分。

**Dart 中的 reduce 函數**

Dart 中的 reduce 函數是一種強大的工具，可用於將迭代器的元素聚合或組合成一個值。它依次對迭代器的元素進行操作，應用提供的函數來累積結果。reduce 函數對於執行計算、總結數據或轉換集合特別有用。

**reduce 函數的語法和用法**

reduce 函數在 dart:core 庫中可用，可用於任何迭代器對象，例如列表、集合或流。其基本語法如下：

Iterable.reduce(initialValue, combine)

* initialValue: 這是可選參數，指定累加器的初始值。如果未提供，則迭代器的第一個元素將用作初始值。
* combine: 這是必需參數，代表要應用於迭代器中每個元素對的函數。該函數接受兩個參數：
  + accumulator: 累加器的當前值，初始化為 initialValue 或第一個元素（如果未提供 initialValue）。
  + currentElement: 正在從迭代器中處理的當前元素。

combine 函數應根據當前的 accumulator 和 currentElement 返回累加器的更新值。reduce 函數將繼續將 combine 函數應用於連續的元素對，直到處理整個迭代器，最後返回累加器的最終值。

**reduce 函數的示例用法**

考慮一個場景，您想計算列表中所有元素的總和：

List<int> numbers = [1, 2, 3, 4, 5];

int sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue);

print(sum); // 輸出：15

在此示例中，reduce 函數用於 numbers 列表，將初始值 0 傳遞給 combine 函數。combine 函數簡單地將 currentValue 添加到 accumulator 中，從而有效地累積所有元素的總和。累加器的最終值為 15，返回為結果。

**reduce 函數的高級應用**

reduce 函數可用於超越簡單計算的更複雜任務。它可以用於轉換集合、總結數據甚至實現自定義算法。以下是一些高級用法示例：

1. **計數元素的出現次數：**

要計數列表中每個元素的出現次數，您可以將 reduce 函數與映射一起使用：

List<String> words = ['hello', 'world', 'hello', 'again'];

Map<String, int> wordCounts = words.reduce((counts, word) {

counts[word] ??= 0;

counts[word]++;

return counts;

}, {});

print(wordCounts); // 輸出：{'hello': 2, 'world': 1, 'again': 1}

1. **創建自定義數據結構：**

您可以使用 reduce 函數從迭代器中構建自定義數據結構。例如，創建嵌套樹結構：

List<Map<String, dynamic>> data = [

{'name': 'Alice', 'age': 30},

{'name': 'Bob', 'age': 25},

{'name': 'Charlie', 'age': 40},

];

Map<String, List<Map<String, dynamic>>> groupedData = data.reduce((groups, user) {

groups[user['name']] ??= [];

groups[user['name']].add(user);

return groups;

}, {});

print(groupedData); // 輸出：{'Alice': [{'name': 'Alice', 'age': 30}], 'Bob': [{'name': 'Bob', 'age': 25}], 'Charlie': [{'name': 'Charlie', 'age': 40}]}

1. **實現自定義算法：**

reduce 函數可用於實現自定義算法，例如在迭代器中查找最大或最小值：

List<int> numbers = [10, 20, 3, 45, -5];

int maximum = numbers.reduce((max, current

share

more\_vert

**reduce 高級用法**

使用初始值

reduce 方法預設使用集合的第一個元素作為初始值,但我們也可以指定一個自定義的初始值,這在某些情況下非常有用。例如:

var numbers = [2, 4, 6, 8];

var sum = numbers.reduce((value, element) => value + element, 1); // 輸出 21

在這個例子中,我們將初始值設為 1,然後對集合中的每個元素執行相加操作,最終得到的結果是 21。

組合多個操作

我們可以將 reduce 與其他集合操作相組合,實現更複雜的功能。例如,下面的代碼展示了如何計算一個字串列表中所有字串的總長度:

var strings = ['apple', 'banana', 'cherry'];

var totalLength = strings.map((str) => str.length).reduce((value, length) => value + length);

print(totalLength); // 輸出 15

處理嵌套集合

reduce 方法也可以用於處理嵌套集合,例如將一個二維列表展平為一個一維列表:

var matrix = [[1, 2, 3], [4, 5], [6, 7, 8, 9]];

var flattened = matrix.reduce((acc, row) => acc..addAll(row), []);

print(flattened); // 輸出 [1, 2, 3, 4, 5, 6, 7, 8, 9]

實現自定義邏輯

我們可以在 reduce 的回調函數中實現任何自定義的邏輯,使其更加靈活和強大。例如,下面的代碼展示了如何實現一個簡單的字符串模板替換功能:

var template = 'Hello, {name}! You are {age} years old.';

var values = {'name': 'Alice', 'age': 25};

var result = template.split(RegExp(r'\{(.\*?)\}')).reduce((acc, part) {

var key = part.contains('}') ? part.substring(0, part.indexOf('}')) : null;

var value = key != null ? values[key] : null;

return acc + (value ?? part);

}, '');

print(result); // 輸出 'Hello, Alice! You are 25 years old.'

**Dart 中的 fold 函數**

在 Dart 中，fold 方法是一個高階函數，它可以用於對集合中的元素進行累加操作。與 reduce 方法類似，fold 方法接受一個初始值和一個將前一個值與當前元素組合成新值的函數。不過，fold 方法需要顯式地提供初始值，而 reduce 方法則使用集合的第一個元素作為初始值。

使用 fold 方法的基本語法如下:

collection.fold(initialValue, (previousValue, element) => newValue)

其中：

collection 是要進行折疊操作的集合。

initialValue 是折疊操作的初始值。

(previousValue, element) => newValue 是一個函數，用於將前一個值與當前元素組合成一個新值。

下面是一個使用 fold 方法求和的示例：

var numbers = [1, 2, 3, 4, 5];

var sum = numbers.fold(0, (prev, current) => prev + current);

print(sum); // 輸出: 15

在這個例子中，我們使用 fold 將 numbers 列表中的所有元素相加。初始值為 0，並使用匿名函數 (prev, current) => prev + current 將前一個值 prev 與當前元素 current 相加，得到新的值。

fold 方法也可以用於更複雜的操作，例如將一個嵌套的列表展平為一個一維列表：

var nestedList = [[1, 2], [3, 4, 5], [6]];

var flattenedList = nestedList.fold<List<int>>(

[], (result, element) => [...result, ...element]);

print(flattenedList); // 輸出: [1, 2, 3, 4, 5, 6]

在這個例子中，我們首先將初始值設置為一個空列表 []。然後，使用展開運算符 ... 將前一個結果 result 和當前子列表 element 合併成一個新的列表。最終，我們得到了一個包含所有元素的展平後的列表 [1, 2, 3, 4, 5, 6]。

**fold 高級用法**

使用 foldRight 從右側開始折疊

foldRight 方法類似於 fold,但是它從右側開始對集合進行折疊。對於一些特殊情況,如處理運算優先級等,使用 foldRight 可能會更加方便。

var numbers = [1, 2, 3, 4, 5];

var result = numbers.foldRight(0, (element, value) => value \* 10 + element);

print(result); // 輸出: 54321

在這個例子中,我們從右側開始折疊,將每個數字與前一個值相乘後加上自己,實現了將數字顛倒的效果。

使用泛型類型參數

由於 fold 的返回值類型取決於初始值和組合函數的返回類型,因此我們可以通過指定泛型類型參數來限制返回類型,從而提高代碼的類型安全性。

var numbers = [1, 2, 3, 4, 5];

var sum = numbers.fold<int>(0, (prev, current) => prev + current);

print(sum); // 輸出: 15

在這個例子中,我們將 fold 的泛型類型參數設置為 int,確保了返回值的類型為 int。

實現自定義邏輯

fold 方法可以實現各種自定義的邏輯,甚至可以用於實現其他集合操作。例如,我們可以使用 fold 來實現 map 操作:

var numbers = [1, 2, 3, 4, 5];

var doubledNumbers = numbers.fold<List<int>>([], (result, element) => [...result, element \* 2]);

print(doubledNumbers); // 輸出: [2, 4, 6, 8, 10]

在這個例子中,我們使用 fold 將每個元素乘以 2,並將結果添加到一個新的列表中。初始值為空列表 [],組合函數則使用展開運算符 ... 將當前元素添加到結果列表中。

處理異常情況

通過合理設計初始值和組合函數,我們可以使用 fold 來處理異常情況,如空集合或含有 null 值的集合。

var numbers = [1, 2, null, 3, 4];

var sum = numbers.fold<int?>(0, (prev, current) => current == null ? prev : prev + current);

print(sum); // 輸出: 10

在這個例子中,我們使用了一個可空的整數類型 int? 作為泛型類型參數。如果當前元素為 null,則返回前一個值;否則返回兩個值的和。這樣可以避免由於 null 值而導致的 NullPointerException。

**Dart 中的 where 函數**

在 Dart 中，where 是集合的一個實例方法，它用於過濾集合中滿足指定條件的元素。where 方法接受一個函數作為參數，該函數應該返回一個布林值,用於判斷當前元素是否應該被保留。

where 方法的基本語法如下:

collection.where((element) => condition)

其中:

collection 是要進行過濾的集合。

(element) => condition 是一個函數,它接受當前元素作為參數,並返回一個布林值,用於判斷該元素是否應該被保留。

下面是一個使用 where 方法過濾偶數的示例:

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var evenNumbers = numbers.where((number) => number.isEven);

print(evenNumbers); // 輸出: (2, 4, 6, 8, 10)

在這個例子中，我們使用 where 方法過濾 numbers 列表中的偶數。(number) => number.isEven 是一個匿名函數,它檢查當前數字 number 是否為偶數,並返回一個布林值。where 方法會保留那些使函數返回 true 的元素。

where 方法還可以與其他集合操作相結合,以實現更複雜的過濾和轉換操作。例如,我們可以先使用 where 過濾出偶數,然後使用 map 將它們乘以 2:

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var doubledEvenNumbers = numbers.where((number) => number.isEven).map((number) => number \* 2);

print(doubledEvenNumbers); // 輸出: (4, 8, 12, 16, 20)

在這個例子中,我們先使用 where 過濾出偶數,然後使用 map 將這些偶數乘以 2。最終,我們得到了一個包含所有偶數的兩倍值的新集合。

where 方法還支持使用更複雜的條件表達式,例如使用邏輯運算符 && 和 || 組合多個條件。這樣可以實現更精細的過濾控制。

var people = [

{'name': 'Alice', 'age': 25, 'gender': 'female'},

{'name': 'Bob', 'age': 30, 'gender': 'male'},

{'name': 'Charlie', 'age': 35, 'gender': 'male'},

{'name': 'David', 'age': 40, 'gender': 'male'},

];

var youngFemales = people.where((person) => person['age']! < 30 && person['gender'] == 'female');

print(youngFemales); // 輸出: [{'name': 'Alice', 'age': 25, 'gender': 'female'}]

在這個例子中,我們使用 where 方法過濾出年齡小於 30 歲且性別為女性的人。

**where 高級用法**

在 Dart 中,where 方法除了基本的過濾功能外,還有一些高級用法值得探討。

與其他集合操作組合使用

where 可以與其他集合操作組合使用,實現更複雜的數據處理邏輯。例如,我們可以將 where 與 map 和 toList 等方法結合,對集合進行過濾和轉換:

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var squaresOfEvenNumbers = numbers

.where((n) => n.isEven)

.map((n) => n \* n)

.toList();

print(squaresOfEvenNumbers); // 輸出: [4, 16, 36, 64, 100]

在這個例子中,我們首先使用 where 過濾出偶數,然後使用 map 對這些偶數進行平方運算,最後使用 toList 將結果轉換為列表。

使用級聯運算符

Dart 支持級聯運算符 (..),可以將多個集合操作串接在一起,使代碼更加簡潔。我們可以將 where 與級聯運算符結合使用:

var people = [

{'name': 'Alice', 'age': 25, 'gender': 'female'},

{'name': 'Bob', 'age': 30, 'gender': 'male'},

{'name': 'Charlie', 'age': 35, 'gender': 'male'},

{'name': 'David', 'age': 40, 'gender': 'male'},

];

var youngFemaleNames = people

..where((person) => person['age']! < 30 && person['gender'] == 'female')

.map((person) => person['name'])

.toList();

print(youngFemaleNames); // 輸出: ['Alice']

在這個例子中,我們使用級聯運算符將 where、map 和 toList 串接在一起,過濾出年齡小於 30 歲且性別為女性的人,並獲取他們的名字列表。

使用高階函數

where 接受一個函數作為參數,因此我們可以傳遞一個高階函數來實現更複雜的過濾邏輯。例如,我們可以使用 any 和 every 等高階函數來檢查集合中的元素是否滿足某些條件:

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var hasEvenNumber = numbers.any((n) => n.isEven); // true

var allEvenNumbers = numbers.every((n) => n.isEven); // false

在這個例子中,我們使用 any 檢查集合中是否至少有一個偶數,使用 every 檢查集合中的所有數字是否都是偶數。

處理空集合和 null 值

在實際開發中,我們經常需要處理空集合和包含 null 值的集合。where 方法可以與 ?? 運算符和 isNotEmpty 方法結合使用,以便更優雅地處理這些情況:

var numbers = [1, 2, null, 4, 5, null];

var nonNullEvenNumbers = numbers

.where((n) => n != null && n.isEven)

.toList();

print(nonNullEvenNumbers); // 輸出: [2, 4]

var emptyList = [];

var result = emptyList.where((n) => n.isEven) ?? [];

print(result); // 輸出: []

在第一個例子中,我們使用 where 過濾出非 null 且為偶數的元素。在第二個例子中,我們使用 ?? 運算符來處理空集合,避免對空集合進行操作時引發異常。

**Dart 中的 sort 函數**

在 Dart 中，sort 是一個集合方法，用於對集合中的元素進行排序。它提供了多種排序方式供選擇。

最簡單的用法是直接調用 sort 方法，它會根據元素的自然排序規則對集合進行排序。對於數字和字串類型的集合,sort 方法會依照它們的大小順序進行排序。例如:

var numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5];

numbers.sort(); // 對數字列表進行排序

print(numbers); // 輸出: [1, 1, 2, 3, 4, 5, 5, 6, 9]

var names = ['Alice', 'Bob', 'Charlie', 'David'];

names.sort(); // 對字串清單進行排序

print(names); // 輸出: ['Alice', 'Bob', 'Charlie', 'David']

但是,如果集合中的元素是自定義對像或者更複雜的數據類型,那麼我們需要提供一個比較函數來定義排序規則。比較函數必須返回一個整數值,表示兩個元素的相對大小。

var people = [

{'name': 'Alice', 'age': 25},

{'name': 'Bob', 'age': 30},

{'name': 'Charlie', 'age': 35},

{'name': 'David', 'age': 40},

];

// 按照年齡昇冪排序

people.sort((a, b) => a['age']!.compareTo(b['age']!));

print(people); // 輸出: [{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30}, {'name': 'Charlie', 'age': 35}, {'name': 'David', 'age': 40}]

// 按照年齡降冪排序

people.sort((a, b) => b['age']!.compareTo(a['age']!));

print(people); // 輸出: [{'name': 'David', 'age': 40}, {'name': 'Charlie', 'age': 35}, {'name': 'Bob', 'age': 30}, {'name': 'Alice', 'age': 25}]

在這個例子中,我們首先定義了一個包含人員資訊的清單 people。然後,我們使用 sort 方法並提供一個比較函數來定義排序規則。比較函數 (a, b) => a['age']!.compareTo(b['age']!) 會比較兩個人員對象的年齡,並根據比較結果返回一個整數值。如果返回值為負數,則表示 a 應該排在 b 之前;如果返回值為正數,則表示 b 應該排在 a 之前;如果返回值為 0,則表示 a 和 b 的順序不變。

除了上述用法,sort 方法還支持一些可選參數,例如 start 和 end 參數,用於指定需要排序的元素範圍。另外,還有一個 sortBy 方法,可以讓我們根據集合中元素的某個屬性進行排序。

var people = [

{'name': 'Alice', 'age': 25},

{'name': 'Bob', 'age': 30},

{'name': 'Charlie', 'age': 35},

{'name': 'David', 'age': 40},

];

// 按照名字長度昇冪排序

people.sortBy((person) => person['name']!.length);

print(people); // 輸出: [{'name': 'Bob', 'age': 30}, {'name': 'Alice', 'age': 25}, {'name': 'David', 'age': 40}, {'name': 'Charlie', 'age': 35}]

在這個例子中,我們使用 sortBy 方法根據人員名字的長度對列表進行排序。(person) => person['name']!.length 是一個函數,它返回人員名字的長度,並被用作排序依據。

**sort 高級用法**

使用 Comparable 介面

如果集合中的元素實現了 Comparable 介面,那麼我們就可以直接使用 sort 方法進行排序,而不需要提供比較函數。Comparable 介面定義了 compareTo 方法,用於比較兩個對像的大小。

class Person implements Comparable<Person> {

final String name;

final int age;

Person(this.name, this.age);

@override

int compareTo(Person other) {

return age.compareTo(other.age);

}

}

var people = [

Person('Alice', 25),

Person('Bob', 30),

Person('Charlie', 35),

Person('David', 40),

];

people.sort(); // 直接使用 sort 方法進行排序

print(people); // 輸出: [Person('Alice', 25), Person('Bob', 30), Person('Charlie', 35), Person('David', 40)]

在這個例子中,我們定義了一個 Person 類,它實現了 Comparable 介面。在 compareTo 方法中,我們比較了兩個 Person 對象的年齡。由於 Person 類已經實現了比較邏輯,因此我們可以直接在 people 列表上調用 sort 方法,而不需要再提供額外的比較函數。

使用 Comparator 類

如果我們需要使用多種不同的排序規則,那麼可以考慮使用 Comparator 類。Comparator 類提供了一種靈活的方式來定義比較邏輯,並且可以被重用。

import 'dart:math' as math;

class Person {

final String name;

final int age;

Person(this.name, this.age);

}

var people = [

Person('Alice', 25),

Person('Bob', 30),

Person('Charlie', 35),

Person('David', 40),

];

// 按年齡昇冪排序

people.sort(Comparator<Person>((p1, p2) => p1.age.compareTo(p2.age)));

print(people); // 輸出: [Person('Alice', 25), Person('Bob', 30), Person('Charlie', 35), Person('David', 40)]

// 按姓名長度降冪排序

people.sort(Comparator<Person>((p1, p2) => -p1.name.length.compareTo(p2.name.length)));

print(people); // 輸出: [Person('Charlie', 35), Person('David', 40), Person('Alice', 25), Person('Bob', 30)]

// 按照年齡和姓名的長度進行複合排序

people.sort(Comparator<Person>((p1, p2) {

var ageComparison = p1.age.compareTo(p2.age);

if (ageComparison != 0) return ageComparison;

return p1.name.length.compareTo(p2.name.length);

}));

print(people); // 輸出: [Person('Alice', 25), Person('Bob', 30), Person('Charlie', 35), Person('David', 40)]

在這個例子中,我們首先定義了一個 Person 類,包含名字和年齡屬性。然後,我們創建了一個 people 列表,包含了多個 Person 對象。

接下來,我們使用了 Comparator 類來定義不同的排序規則。首先,我們按照年齡昇冪對 people 列表進行排序。然後,我們按照姓名長度降冪對列表進行排序。最後,我們定義了一個複合排序規則,首先按照年齡昇冪排序,如果年齡相同,則按照姓名長度昇冪排序。

使用 Comparator 類的好處是,我們可以將比較邏輯封裝在一個單獨的對象中,並且可以輕鬆地重用或修改這些比較邏輯。這樣可以使我們的代碼更加靈活和易於維護。

使用 sortedBy 和 sortedByDescending 方法

除了 sort 方法,Dart 還提供了 sortedBy 和 sortedByDescending 方法,它們可以根據指定的鍵函數對集合進行排序,並返回一個新的排序後的集合,而不會修改原始集合。

var numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5];

// 昇冪排序

var sortedNumbers = numbers.sortedBy((num) => num);

print(sortedNumbers); // 輸出: [1, 1, 2, 3, 4, 5, 5, 6, 9]

// 降冪排序

var reversedNumbers = numbers.sortedByDescending((num) => num);

print(reversedNumbers); // 輸出: [9, 6, 5, 5, 4, 3, 2, 1, 1]

在這個例子中,我們首先定義了一個數字列表 numbers。然後,我們使用 sortedBy 方法根據數字的大小對列表進行昇冪排序,並返回一個新的排序後的列表 sortedNumbers。接著,我們使用 sortedByDescending 方法根據數字的大小對列表進行降冪排序,並返回一個新的排序後的列表 reversedNumbers。

使用 sortedBy 和 sortedByDescending 方法的好處是,它們不會修改原始集合,而是返回一個新的排序後的集合。這在某些情況下可能更加方便和安全。

其他用於排序的內建函數或方法

除了已經討論過的 sort、sortedBy 和 sortedByDescending 方法之外,Dart 還提供了一些其他內建的函數和方法,可以用於排序操作。

List.sort、List.sorted 和 List.sortedByComparing

這些方法是 List 類中的靜態方法,可以直接調用並傳入相應的參數。

List.sort(list, [compare]) 方法會就地修改原始列表,根據提供的比較器對元素進行排序。

List.sorted(list, [compare]) 方法不會修改原始列表,而是返回一個新的排序後的列表。

List.sortedByComparing(list, {valueComparison, valueDescendingComparison}) 方法類似於 List.sorted方法,但允許使用更複雜的比較規則進行排序。

var numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5];

// 使用 List.sort 方法對原始列表進行就地排序

List.sort(numbers);

print(numbers); // 輸出: [1, 1, 2, 3, 4, 5, 5, 6, 9]

// 使用 List.sorted 方法獲取一個新的排序後的列表

var sortedNumbers = List.sorted(numbers);

print(sortedNumbers); // 輸出: [1, 1, 2, 3, 4, 5, 5, 6, 9]

// 使用 List.sortedByComparing 方法進行更複雜的排序

var people = [

{'name': 'Alice', 'age': 25},

{'name': 'Bob', 'age': 30},

{'name': 'Charlie', 'age': 35},

{'name': 'David', 'age': 40},

];

var sortedPeople = List.sortedByComparing(

people,

valueComparison: (person1, person2) => person1['age']!.compareTo(person2['age']!),

valueDescendingComparison: (person1, person2) => person2['age']!.compareTo(person1['age']!),

);

print(sortedPeople); // 輸出: [{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 30}, {'name': 'Charlie', 'age': 35}, {'name': 'David', 'age': 40}]

fold 和 reduce 方法

雖然 fold 和 reduce 方法主要用於對集合進行歸約操作,但它們也可以用於實現特定的排序演算法,例如合併排序和計數排序。

// 使用 fold 方法實現合併排序

List<int> mergeSort(List<int> list) {

if (list.length <= 1) return list;

var middle = list.length ~/ 2;

var left = mergeSort(list.sublist(0, middle));

var right = mergeSort(list.sublist(middle));

return left.fold<List<int>>(

right,

(result, element) {

if (result.isEmpty) return [element];

if (element < result.first) {

return [element, ...result];

} else {

return [result.first, ...result.skip(1).fold([], (r, e) => [...r, e])];

}

},

);

}

var numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5];

var sortedNumbers = mergeSort(numbers);

print(sortedNumbers); // 輸出: [1, 1, 2, 3, 4, 5, 5, 6, 9]

在這個例子中,我們使用 fold 方法實現了一個合併排序演算法。該演算法首先將清單一分為二,然後對左右兩半分別進行排序,最後將排序後的左右兩半合併為一個新的排序列表。

Iterable 和 Iterator

Iterable 和 Iterator 類提供了遍歷集合的功能,我們可以利用這些功能實現特定的排序演算法。

// 使用 Iterable 和 Iterator 實現選擇排序

List<T> selectionSort<T extends Comparable<T>>(Iterable<T> iterable) {

final iterator = iterable.iterator;

final sortedList = <T>[];

while (iterator.moveNext()) {

T minValue = iterator.current;

final minIterator = iterator;

while (iterator.moveNext()) {

if (iterator.current.compareTo(minValue) < 0) {

minValue = iterator.current;

}

}

sortedList.add(minValue);

minIterator.current = iterator.current;

}

return sortedList;

}

var numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5];

var sortedNumbers = selectionSort(numbers);

print(sortedNumbers); // 輸出: [1, 1, 2, 3, 4, 5, 5, 6, 9]

在這個例子中,我們使用 Iterable 和 Iterator 實現了一個選擇排序演算法。演算法遍歷元素,每次找到最小的元素並將其添加到排序後的列表中。

其他排序演算法在Dart中實現

Dart 作為一種現代化的編程語言,可以實現各種經典和先進的排序演算法。以下是一些常見的排序演算法及其在 Dart 中的實現示例:

**冒泡排序 (Bubble Sort)**

List<int> bubbleSort(List<int> list) {

for (int i = 0; i < list.length - 1; i++) {

bool swapped = false;

for (int j = 0; j < list.length - i - 1; j++) {

if (list[j] > list[j + 1]) {

int temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break;

}

return list;

}

**插入排序 (Insertion Sort)**

List<int> insertionSort(List<int> list) {

for (int i = 1; i < list.length; i++) {

int key = list[i];

int j = i - 1;

while (j >= 0 && list[j] > key) {

list[j + 1] = list[j];

j--;

}

list[j + 1] = key;

}

return list;

}

**選擇排序 (Selection Sort)**

List<int> selectionSort(List<int> list) {

for (int i = 0; i < list.length - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < list.length; j++) {

if (list[j] < list[minIndex]) {

minIndex = j;

}

}

if (minIndex != i) {

int temp = list[i];

list[i] = list[minIndex];

list[minIndex] = temp;

}

}

return list;

}

**快速排序 (Quick Sort)**

List<int> quickSort(List<int> list) {

if (list.length <= 1) return list;

int pivotIndex = list.length ~/ 2;

int pivot = list[pivotIndex];

list.removeAt(pivotIndex);

List<int> left = list.where((num) => num < pivot).toList();

List<int> right = list.where((num) => num >= pivot).toList();

return [...quickSort(left), pivot, ...quickSort(right)];

}

**合併排序 (Merge Sort)**

List<int> mergeSort(List<int> list) {

if (list.length <= 1) return list;

int middle = list.length ~/ 2;

List<int> left = mergeSort(list.sublist(0, middle));

List<int> right = mergeSort(list.sublist(middle));

return merge(left, right);

}

List<int> merge(List<int> left, List<int> right) {

List<int> result = [];

int leftIndex = 0, rightIndex = 0;

while (leftIndex < left.length && rightIndex < right.length) {

if (left[leftIndex] < right[rightIndex]) {

result.add(left[leftIndex]);

leftIndex++;

} else {

result.add(right[rightIndex]);

rightIndex++;

}

}

result.addAll(left.skip(leftIndex));

result.addAll(right.skip(rightIndex));

return result;

}

**計數排序 (Counting Sort)**

List<int> countingSort(List<int> list) {

int max = list.reduce(math.max);

List<int> counts = List.filled(max + 1, 0);

for (int num in list) {

counts[num]++;

}

List<int> result = [];

for (int i = 0; i < counts.length; i++) {

result.addAll(List.filled(counts[i], i));

}

return result;

}

**基數排序 (Radix Sort)**

List<int> radixSort(List<int> list) {

int max = list.reduce(math.max);

int digits = max.toString().length;

for (int digit = 0; digit < digits; digit++) {

list = countingSort(

list,

(num) => (num ~/ math.pow(10, digit)) % 10,

);

}

return list;

}

List<int> countingSort(List<int> list, int Function(int) digitExtractor) {

List<int> counts = List.filled(10, 0);

List<int> result = List.filled(list.length, 0);

for (int num in list) {

counts[digitExtractor(num)]++;

}

for (int i = 1; i < 10; i++) {

counts[i] += counts[i - 1];

}

for (int i = list.length - 1; i >= 0; i--) {

int digit = digitExtractor(list[i]);

counts[digit]--;

result[counts[digit]] = list[i];

}

return result;

}

這些只是一些常見的排序演算法示例,還有許多其他的排序演算法可以在 Dart 中實現,例如堆排序、桶排序、Shell 排序等。不同的排序演算法具有不同的時間和空間複雜度,適用於不同的場景。根據具體的需求和數據特點選擇合適的排序演算法非常重要。

**高效的排序演算法**

是的,除了上面提到的一些經典排序演算法之外,Dart 也可以實現一些更高效的先進排序演算法。以下是一些值得關注的高效排序演算法:

TimSort

TimSort 是一種自適應的混合排序演算法,它結合了合併排序和插入排序的優勢。TimSort 在大多數情況下的表現都很出色,它已經被廣泛應用於 Python、Java、Android 和 GNU 等編程語言和系統中。以下是 TimSort 在 Dart 中的實現:

List<int> timSort(List<int> list) {

const minMerge = 32;

const maxMerge = 64;

int minGallop = minMerge;

Range startTimSort(int lo, int hi) {

rangeCheck(list, lo, hi);

int nRemaining = hi - lo;

if (nRemaining < 2) return new Range(lo, hi);

// 如果陣列已經幾乎排好序,使用插入排序

if (nRemaining < minMerge) {

int initRunLen = countRunAndMakeAscending(lo, hi);

binarySort(lo, hi, lo + initRunLen);

return new Range(lo + initRunLen, hi);

}

// 使用時間線排序

int minRun = minRunLength(nRemaining);

for (int i = lo; i < hi; i += minRun) {

int runHi = minRunLength(nRemaining) + i;

if (runHi > hi) runHi = hi;

// 使用插入排序對小數組進行排序

int lastRun = countRunAndMakeAscending(i, runHi);

if (lastRun < runHi - i) {

binarySort(i, runHi, lastRun);

reverseRange(i, runHi);

}

}

// 現在讓我們合併所有的排序子數組

mergeCollapse(lo, hi, minMerge);

return new Range(lo, hi);

}

startTimSort(0, list.length);

return list;

}

Introsort

Introsort 是一種混合排序演算法,它結合了快速排序和堆排序的優勢。在最壞情況下,Introsort 會退化為堆排序,從而避免了快速排序在處理已經排序或反向排序的數據時的性能下降。Introsort 已被應用於 C++ 標準庫中。以下是 Introsort 在 Dart 中的實現:

List<int> introSort(List<int> list, [int maxDepth = floor(2 \* log(list.length))]) {

\_introSort(list, 0, list.length, maxDepth);

return list;

}

void \_introSort(List<int> list, int start, int end, int maxDepth) {

if (end - start <= 1) return;

if (maxDepth == 0) {

heapSort(list, start, end);

return;

}

int pivot = \_partition(list, start, end);

\_introSort(list, start, pivot, maxDepth - 1);

\_introSort(list, pivot + 1, end, maxDepth - 1);

}

int \_partition(List<int> list, int start, int end) {

int pivot = start + Random().nextInt(end - start);

int pivotValue = list[pivot];

swap(list, pivot, end - 1);

int storeIndex = start;

for (int i = start; i < end - 1; i++) {

if (list[i] < pivotValue) {

swap(list, storeIndex, i);

storeIndex++;

}

}

swap(list, storeIndex, end - 1);

return storeIndex;

}

void swap(List<int> list, int a, int b) {

int temp = list[a];

list[a] = list[b];

list[b] = temp;

}

Smooth Sort

Smooth Sort 是一種基於二叉樹的排序演算法,它具有非常好的最壞情況時間複雜度和內存使用情況。Smooth Sort 在處理大型數據集時表現非常出色。以下是 Smooth Sort 在 Dart 中的實現:

List<int> smoothSort(List<int> list) {

int n = list.length;

// 創建一個 n 個節點的二叉樹

List<Node> nodes = List.generate(n, (i) => Node(list[i], i));

// 構建初始二叉樹

Node root = buildTree(nodes);

// 遍歷二叉樹,收集排序元素

int index = 0;

root.traverseInOrder((node) {

list[index++] = node.value;

});

return list;

}

Node buildTree(List<Node> nodes) {

if (nodes.isEmpty) return null;

List<Node> children = [];

for (int i = 0, m = nodes.length; i < m - m ~/ 2; i++) {

children.add(nodes[i]);

}

Node root = Node(

nodes.last.value,

nodes.last.index,

buildTree(children),

buildTree(nodes.skip(nodes.length ~/ 2).take(m ~/ 2).toList()),

);

for (Node child in root.children) {

child.parent = root;

}

return root;

}

class Node {

int value;

int index;

Node parent;

List<Node> children;

Node(this.value, this.index, [this.left, Node right]) {

children = [];

if (left != null) children.add(left);

if (right != null) children.add(right);

}

Node get left => children.length > 0 ? children[0] : null;

Node get right => children.length > 1 ? children[1] : null;

void traverseInOrder(void Function(Node node) action) {

left?.traverseInOrder(action);

action(this);

right?.traverseInOrder(action);

}

}

以上是三種高效的先進排序演算法在 Dart 中的實現示例。這些演算法在處理大型數據集或特殊情況下具有更好的性能表現。不過,它們的實現也相對較為複雜,需要更多的代碼和數據結構支持。

**多維陣列排序使用**

在 Dart 中對多維陣列進行排序時,需要根據具體的排序需求定義適當的比較函數。Dart 提供了 sort() 方法和 sortedBy() 方法來對列表進行排序,這兩個方法都接受一個可選的 Comparator 參數,用於定義排序的規則。

以下是一些示例:

按照第一個元素排序二維陣列

void main() {

List<List<int>> matrix = [

[3, 1],

[1, 2],

[2, 3],

[4, 0],

];

matrix.sort((a, b) => a[0].compareTo(b[0]));

print(matrix); // [[1, 2], [2, 3], [3, 1], [4, 0]]

}

按照第二個元素排序二維陣列

void main() {

List<List<int>> matrix = [

[3, 1],

[1, 2],

[2, 3],

[4, 0],

];

matrix.sort((a, b) => a[1].compareTo(b[1]));

print(matrix); // [[4, 0], [3, 1], [1, 2], [2, 3]]

}

按照多個元素的組合排序二維陣列

void main() {

List<List<int>> matrix = [

[3, 1],

[1, 2],

[2, 3],

[4, 0],

[3, 2],

];

matrix.sortedBy((List<int> row) {

// 首先按第一個元素排序，如果第一個元素相同，則按第二個元素排序

return [row[0], row[1]];

});

print(matrix); // [[1, 2], [2, 3], [3, 1], [3, 2], [4, 0]]

}

在上面的例子中，我們使用 sortedBy() 方法並傳遞一個匿名函數作為比較器。該匿名函數接受一個二維陣列的行作為參數，並返回一個列表作為排序鍵。在這個例子中，我們首先按第一個元素進行排序，如果第一個元素相同，則按第二個元素進行排序。

按照物件屬性排序二維陣列

void main() {

List<Person> people = [

Person(3, 25),

Person(2, 30),

Person(1, 35),

Person(4, 20),

];

people.sort((a, b) => a.age.compareTo(b.age));

print(people); // [Person(4, 20), Person(3, 25), Person(2, 30), Person(1, 35)]

}

class Person {

final int id;

final int age;

Person(this.id, this.age);

@override

String toString() {

return 'Person($id, $age)';

}

}

在這個例子中，我們定義了一個 Person 類，並按照 age 屬性對 Person 物件清單進行排序。

**Future 的高級用法**

Dart 中的 Future 提供了一種強大的機制來處理異步操作。除了基本的異步編程功能外,Future 還有一些高級用法,可以幫助您編寫更加健壯、高效和可擴展的異步代碼。

1. **Future 鏈和組合**

您可以將多個 Future 組合在一起,形成一個 Future 鏈。這種方式可以減少嵌套層級,使代碼更加簡潔易讀。

Future<String> fetchData() async {

final data1 = await loadData1();

final data2 = await loadData2(data1);

final data3 = await loadData3(data2);

return data3;

}

您還可以使用 Future.wait 和 Future.forEach 等方法來協調多個 Future 的執行和處理。

1. **錯誤處理**

在處理異步操作時,適當的錯誤處理非常重要。Future 提供了多種方法來處理錯誤,如 catchError、whenComplete 和 then。

Future<void> fetchData() async {

try {

final data = await loadData();

processData(data);

} catch (e) {

handleError(e);

} finally {

cleanup();

}

}

1. **Completer**

在某些情況下,您需要手動創建和完成一個 Future。Completer 類可以幫助您實現這一點。

Future<String> fetchData() {

final completer = Completer<String>();

*// 執行一些異步操作...*

*// 然後完成 Future*

completer.complete('Data fetched');

return completer.future;

}

1. **隔離和錯誤捕獲**

Dart 提供了 runZonedGuarded 等機制,可以幫助您隔離和捕獲未處理的錯誤,防止它們傳播到其他代碼路徑。

runZonedGuarded(() async {

*// 執行一些異步操作*

await fetchData();

}, (error, stackTrace) {

*// 處理未捕獲的錯誤*

handleError(error, stackTrace);

});

1. **延遲和超時**

您可以使用 Future.delayed 來引入延遲,或者使用 Future.timeout 來設置異步操作的超時時間。

Future<void> delayedOperation() async {

await Future.delayed(Duration(seconds: 3));

print('Delayed operation completed');

}

Future<String> fetchDataWithTimeout() async {

try {

return await loadData().timeout(Duration(seconds: 10));

} catch (e) {

return 'Timeout: Data could not be fetched';

}

}

Future 是一種表示非同步操作結果的數據類型。它可以用於處理耗時操作，例如網絡請求或文件 I/O。

Future 的高級用法包括：

* **錯誤處理**

Future 可以使用 catchError() 方法來處理錯誤。如果 Future 操作失敗，catchError() 回調函數將會被調用，並傳遞錯誤信息。

Future<String> fetchData() async {

try {

final response = await http.get('https://example.com/data.json');

if (response.statusCode == 200) {

return response.body;

} else {

throw Exception('Failed to fetch data: ${response.statusCode}');

}

} catch (error) {

return 'Error: $error';

}

}

fetchData().then((data) => print(data)).catchError((error) => print(error));

* **組合 Futures**

Future 可以使用 then() 和 whenComplete() 方法來組合。then() 方法用於在 Future 完成後執行操作，whenComplete() 方法用於在 Future 完成或失敗後執行操作。

Future<int> future1 = Future.delayed(Duration(seconds: 1), () => 1);

Future<int> future2 = Future.delayed(Duration(seconds: 2), () => 2);

future1.then((value) => print('future1: $value')).then((\_) {

return future2.then((value) => print('future2: $value'));

});

future1.whenComplete(() => print('future1 complete')).then((\_) {

return future2.whenComplete(() => print('future2 complete'));

});

* **流處理**

Future 可以使用 asStream() 方法轉換為 Stream。Stream 是一種表示一系列數據值的數據類型。

Future<List<int>> future = Future.value([1, 2, 3]);

future.asStream().listen((data) => print(data));

* **自定義 Future**

可以使用 Future.microtask() 和 Future.delayed() 方法來創建自定義 Future。Future.microtask() 方法創建一個立即完成的 Future，Future.delayed() 方法創建一個在指定延遲時間後完成的 Future。

Future<int> future1 = Future.microtask(() => 1);

Future<int> future2 = Future.delayed(Duration(seconds: 1), () => 2);

print(future1); // 立即打印 1

print(future2); // 1 秒後打印 2

以下是一些 Future 的高級用法示例：

* **實現並行操作**

可以使用 Future 來實現並行操作。例如，以下代碼使用 Future 並行發送兩個網絡請求：

Future<String> fetchData1() async {

final response = await http.get('https://example.com/data1.json');

if (response.statusCode == 200) {

return response.body;

} else {

throw Exception('Failed to fetch data1: ${response.statusCode}');

}

}

Future<String> fetchData2() async {

final response = await http.get('https://example.com/data2.json');

if (response.statusCode == 200) {

return response.body;

} else {

throw Exception('Failed to fetch data2: ${response.statusCode}');

}

}

Future.wait([fetchData1(), fetchData2()]).then((data) {

print('data1: ${data[0]}');

print('data2: ${data[1]}');

});

* **實現流水線操作**

可以使用 Future 來實現流水線操作。例如，以下代碼使用 Future 將圖像從磁盤讀取、縮放和保存到文件：

Future<Image> loadImage(String path) async {

final bytes = await File(path).readAsBytes();

return decodeImageFromList(bytes);

}

Future<Image> resizeImage(Image image) async {

final resizedImage = await image.resize(100, 100);

return resizedImage;

}

Future<void> saveImage(Image image, String path) async {

final pngBytes = await image.toByteData(format: ImageByteFormat.png);

await File(path).writeAsBytes(pngBytes.buffer.asUint8List

void main() async {

final image = await loadImage('path/to/image.jpg');

final resizedImage = await resizeImage(image);

await saveImage(resizedImage, 'path/to/saved\_image.png');

}

在上述代碼中，loadImage() 函數使用 Future 來讀取圖像文件。resizeImage() 函數使用 Future 來縮放圖像。saveImage() 函數使用 Future 來保存圖像文件。

這些函數使用 await 關鍵字來等待 Future 完成。await 關鍵字會阻塞當前的函數，直到 Future 完成。

通過使用 Future，可以將這些操作組合成一個流水線。圖像將從磁盤讀取，然後縮放，最後保存到文件。

以下是一些 Future 的高級用法的好處：

* **提高代碼的可讀性和可維護性**

Future 可以使代碼更易於閱讀和維護。通過將非同步操作包裝在 Future 中，可以使代碼更直觀地理解。

* **簡化錯誤處理**

Future 提供了統一的錯誤處理機制。可以使用 catchError() 方法來處理 Future 操作失敗的情況。

* **提高性能**

Future 可以提高性能。通過使用 Future，可以在多個 CPU 或核心上並行執行操作。

以下是一些 Future 的高級用法需要注意的事項：

* **避免嵌套 Future 過多**

嵌套 Future 過多會使代碼難以閱讀和理解。如果需要嵌套多個 Future，可以考慮使用 async 和 await 關鍵字來簡化代碼。

* **注意錯誤處理**

應始終在使用 Future 時處理錯誤。可以使用 catchError() 方法來處理 Future 操作失敗的情況。

* **避免洩漏 Future**

Future 可能會洩漏，如果 Future 未被使用，則可能導致資源洩漏。應始終確保 Future 已被使用或取消。

總體而言，Future 是一種強大的工具，可用於在 Dart 中實現非同步編程。

**異步程式設計 的用法**

Dart 中的異步程式設計允許您在不阻塞主線程的情況下執行耗時操作。這對於構建響應式和用戶友好的應用程序非常重要。

* **使用 Future 和 async / await**

Future 是一個表示值的最終值的對象。async 和 await 關鍵字用於簡化異步程式碼的編寫。

以下是一個使用 Future 和 async / await 的簡單示例：

Future<String> fetchData() async {

final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/todos/1'));

if (response.statusCode == 200) {

return jsonDecode(response.body)['title'];

} else {

throw Exception('Failed to fetch data');

}

}

void main() async {

final title = await fetchData();

print(title);

}

* **使用 Stream**

Stream 是一個表示一系列值的對象。它通常用於處理來自網絡或傳感器的實時數據。

以下是一個使用 Stream 的簡單示例：

Stream<String> counterStream() async\* {

for (int i = 0; i < 10; i++) {

yield 'Count: $i';

await Future.delayed(const Duration(seconds: 1));

}

}

void main() async {

final subscription = counterStream().listen((event) {

print(event);

});

await Future.delayed(const Duration(seconds: 15));

subscription.cancel();

}

* **使用 async / await for 錯誤處理**

async / await 也可以用於處理錯誤。您可以使用 try / catch 塊來捕獲錯誤。

以下是一個使用 async / await 進行錯誤處理的簡單示例：

Future<String> fetchData() async {

try {

final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/todos/1'));

if (response.statusCode == 200) {

return jsonDecode(response.body)['title'];

} else {

throw Exception('Failed to fetch data');

}

} catch (error) {

return 'Error: $error';

}

}

void main() async {

try {

final title = await fetchData();

print(title);

} catch (error) {

print('Error: $error');

}

}

* **使用 Future.wait 和 Future.microtasks**

Future.wait 函數允許您等待多個 Future 同時完成。Future.microtasks 函數允許您等待所有微任務完成。

以下是一個使用 Future.wait 的簡單示例：

Future<List<String>> fetchMultipleData() async {

final futures = [

fetchData(),

fetchData(),

fetchData(),

];

final results = await Future.wait(futures);

return results;

}

void main() async {

final results = await fetchMultipleData();

print(results);

}

常見的高級用法:

1. **Future 鏈**

您可以將多個 Future 組合在一起,形成一個 Future 鏈。這種方式可以減少嵌套層級,使代碼更加簡潔易讀。

Future<String> fetchData() async {

final data1 = await loadData1();

final data2 = await loadData2(data1);

final data3 = await loadData3(data2);

return data3;

}

Future<String> loadData1() {

*// ...*

}

Future<String> loadData2(String data1) {

*// ...*

}

Future<String> loadData3(String data2) {

*// ...*

}

1. **Future.wait 和 Future.forEach**

Future.wait 用於同時等待多個 Future 完成,而 Future.forEach 則用於對一組 Future 執行迭代操作。這些方法可以幫助您更好地管理和協調多個異步操作。

Future<void> processData() async {

final futures = [loadData1(), loadData2(), loadData3()];

final results = await Future.wait(futures);

print(results); *// [data1, data2, data3]*

await Future.forEach(results, (data) {

*// 對每個數據進行處理*

processData(data);

});

}

1. **Stream 管道**

Dart 中的 Stream 提供了一種強大的方式來處理異步數據流。您可以使用各種轉換器和操作符來建立複雜的異步管道,對數據進行過濾、映射、合併等操作。

final stream = dataSource

.where((data) => data.isValid)

.map((data) => data.transform())

.transform(ThrottleStreamTransformer(Duration(seconds: 1)))

.handleError((error) => print('Error: $error'))

.listen((data) => processData(data));

1. **異步生成器**

異步生成器允許您以懶惰的方式生成異步序列,每次只在需要時計算下一個值。這種方式可以避免不必要的計算,提高效率。

Stream<int> countStream(int max) async\* {

for (int i = 0; i < max; i++) {

await Future.delayed(Duration(seconds: 1));

yield i;

}

}

void main() async {

await for (int value in countStream(5)) {

print(value); *// 每隔一秒輸出 0, 1, 2, 3, 4*

}

}

1. **隔離和錯誤處理**

在處理異步操作時,適當的隔離和錯誤處理非常重要。Dart 提供了多種機制來實現這一點,例如 runZonedGuarded、catchError 和 whenComplete。

Future<void> fetchData() async {

try {

final data = await loadData();

processData(data);

} catch (e) {

handleError(e);

} finally {

cleanup();

}

}

**Duration 的高級用法**

Dart 中的 Duration 類是用於表示一段時間間隔的類型。除了基本的時間計算和格式化功能外,Duration 還有一些高級用法,可以幫助您更有效地處理時間相關的任務。以下是一些常見的高級用法:

1. **使用 inMilliseconds 和 inMicroseconds 進行精確計時**

如果您需要進行高精度的計時操作,可以使用 inMilliseconds 和 inMicroseconds 屬性來獲取毫秒級和微秒級的時間值。這在測量短暫事件或進行基準測試時特別有用。

final startTime = DateTime.now().microsecondsSinceEpoch;

*// 執行某些操作...*

final endTime = DateTime.now().microsecondsSinceEpoch;

final duration = Duration(microseconds: endTime - startTime);

print('Operation took ${duration.inMicroseconds} microseconds');

1. **使用 Duration 創建延遲**

在處理異步操作時,您可以使用 Duration 來創建延遲,模擬某些緩慢的操作或等待某些條件。這在測試和模擬環境中非常有用。

Future<void> slowOperation() async {

await Future.delayed(Duration(seconds: 3));

print('Slow operation completed');

}

void main() async {

print('Starting slow operation...');

await slowOperation();

print('Finished');

}

1. **使用 Duration 控制計時器和動畫**

Duration 可以用於控制計時器和動畫的速度和持續時間。例如,在創建動畫時,您可以使用 Duration 來設置每個動畫幀的持續時間。

import 'dart:async';

void main() {

final controller = AnimationController(

duration: Duration(seconds: 2),

vsync: *// ...*

);

final animation = Tween<double>(begin: 0.0, end: 1.0).animate(controller);

Timer.periodic(Duration(milliseconds: 16), (\_) {

controller.forward(from: animation.value);

});

}

1. **使用 operator+、operator- 和 operator\* 進行時間計算**

Duration 類重載了一些運算符,如 +、- 和 \*。這使得您可以更方便地執行時間計算,例如將一個 Duration 與另一個相加、相減或進行乘法運算。

final duration1 = Duration(hours: 2, minutes: 30);

final duration2 = Duration(minutes: 45);

final totalDuration = duration1 + duration2;

print('Total duration: $totalDuration'); *// 輸出: Total duration: 3:15:00.000000*

final halfDuration = duration1 \* 0.5;

print('Half duration: $halfDuration'); *// 輸出: Half duration: 1:15:00.000000*

1. **使用 toString() 和自定義模式進行格式化**

您可以使用 toString() 方法將 Duration 格式化為人類可讀的字符串表示形式。此外,您還可以通過傳遞自定義模式來控制輸出格式。

final duration = Duration(hours: 2, minutes: 30, seconds: 15);

print(duration.toString()); *// 輸出: 2:30:15.000000*

print(duration.toString().split('.')[0]); *// 輸出: 2:30:15*

*// 自定義模式*

print(duration.toString(

'h "hour(s)", m "minute(s)", s "second(s)"')); *// 輸出: 2 hour(s), 30 minute(s), 15 second(s)*

Duration 類用於表示時間段，它可以表示秒、毫秒、微秒、納秒等時間單位。Duration 類具有以下屬性：

* inDays: 表示時間段的整數天數。
* inHours: 表示時間段的整數小時數。
* inMinutes: 表示時間段的整數分鐘數。
* inSeconds: 表示時間段的整數秒數。
* inMilliseconds: 表示時間段的整數毫秒數。
* inMicroseconds: 表示時間段的整數微秒數。
* inNanoseconds: 表示時間段的整數納秒數。

Duration 類具有以下方法：

* compareTo(Duration other): 比較兩個 Duration 對象的大小。
* hashCode: 返回 Duration 對象的哈希碼。
* toString(): 返回 Duration 對象的字符串表示形式。
* add(Duration other): 將另一個 Duration 對象添加到當前 Duration 對象中。
* subtract(Duration other): 從當前 Duration 對象中減去另一個 Duration 對象。
* multiply(double multiplier): 將當前 Duration 對象乘以一個數字。
* divide(double divisor): 將當前 Duration 對象除以一個數字。
* floor()：取 Duration 對象的向下取整部分。
* ceil()：取 Duration 對象的向上取整部分。
* truncate()：截斷 Duration 對象的小數部分。
* round()：四捨五入 Duration 對象的小數部分。

### 高級用法

以下是一些 Dart 中 Duration 的高級用法：

**1. 使用 Duration 來表示時間間隔**

Duration 類可以用於表示時間間隔，例如兩個時間戳之間的時間差。例如，以下代碼計算兩個時間戳之間的時間差：

DateTime startTime = DateTime(2023, 12, 31, 23, 59, 59);

DateTime endTime = DateTime(2024, 1, 1, 0, 0, 0);

Duration duration = endTime.difference(startTime);

print(duration.inSeconds); // Output: 1

在上面的代碼中，duration 变量表示 startTime 和 endTime 之間的時間差，即 1 秒。

**2. 使用 Duration 來設置延時**

Duration 類可以用於設置延時，例如在异步操作完成之前等待一段時間。例如，以下代碼使用 Future.delayed() 方法來設置 1 秒的延時：

Future<void> delayedFunction() async {

await Future.delayed(Duration(seconds: 1));

print('Delayed function executed');

}

void main() {

delayedFunction();

}

在上面的代碼中，delayedFunction() 函數在 1 秒後執行。

**3. 使用 Duration 來控制動畫**

Duration 類可以用於控制動畫，例如動畫的速度和持續時間。例如，以下代碼使用 Animation<double> 類來創建一個動畫，該動畫在 2 秒內將值從 0 變化到 1：

AnimationController controller = AnimationController(

vsync: TickerProviderStateMixin.tickerProvider(null),

duration: Duration(seconds: 2),

);

Animation<double> animation = Tween<double>(begin: 0, end: 1).animate(controller);

controller.forward();

animation.addListener(() {

print(animation.value);

});

在上面的代碼中，animation 变量表示一個動畫，該動畫在 2 秒內將值從 0 變化到 1。

**4. 使用 Duration 來表示時區**

Duration 類可以用於表示時區，例如 UTC 和 PST 之間的時差。例如，以下代碼創建一個表示 UTC 時區的 Duration 對象：

Duration utcOffset = Duration(hours: 0);

在上面的代碼中，utcOffset 变量表示 UTC 時區的 Duration 對象。