**DART 學習筆記**

Map 的表示和使用方法 2

自訂義資料型態 6

類別 9

類的高級用法 11

枚舉類型 16

Set：表示及使用 20

Function 的高级用法 24

List的高級用法 25

forEach 的高級用法 28

匿名函數 30

可變參數函示 32

連結 python , c 程式 36

filter **函數 37**

reduce 函數 39

fold 函數 41

where 函數 43

sort 函數 45

回調函數 的高级用法 55

mixin 的高級用法 59

abstract 的高級用法 64

Duration 的高級用法 68

extends 的高級用法 70

factory 的高級用法 74

build 的高級用法 78

異步程式設計 的用法 82

Future 的高級用法 85

switch/case 的高級用法 88

for loop 的高級用法 90

**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 中的類別（Class 1）：

**類別**（Class）是 Dart 中用於建立自訂義資料型態的基本結構。它定義了資料型態的屬性（欄位）和行為（方法）。您可以使用類別來建立具有相似特徵和行為的物件。

**建立類別**

要建立類別，請使用 class 關鍵字，後接類別名稱和大括號 {} 來定義其成員：

class Person {

// 類別成員（欄位、方法、建構子等）

}

**欄位**

欄位（Field）是與類別實例關聯的資料變數。它們在類別內部使用 var、dynamic、final 或 const 關鍵字聲明，後接欄位名稱和型別：

class Person {

String name;

int age;

}

在此範例中，name 和 age 是 Person 類別的欄位。String 表示 name 欄位的型別為字串，int 表示 age 欄位的型別為整數。

**方法**

方法（Method）是與類別關聯的功能，用於定義其行為。它們在類別內部使用 def 關鍵字聲明，後接方法名稱、參數、回傳型別和大括號 {} 來定義方法主體：

class Person {

String name;

int age;

void greet() {

print("Hello, my name is $name!");

}

}

在此處，greet() 是 Person 類別的方法，用於列印問候訊息。方法名稱後面是括號，其中包含方法的參數。在本例中，greet() 方法沒有參數。方法的回傳型別為 void，表示它不會回傳任何值。

**建構子**

建構子（Constructor）是特殊方法，當建立物件時，用於初始化物件的狀態。它們在類別內部使用 constructor 關鍵字聲明，後接類別名稱、參數和大括號 {} 來定義建構子主體：

class Person {

String name;

int age;

Person(this.name, this.age);

}

此建構子接收 name 和 age 作為參數，並使用建構子參數初始化 Person 物件的對應欄位。this 關鍵字用於將建構子參數與類別欄位關聯起來。

**建立物件**

要從類別建立物件，請使用 new 關鍵字，後接類別名稱和括號 () 來提供建構子參數：

Person john = Person("John Doe", 30);

這會建立一個名為 john 的 Person 物件，並使用建構子初始化其 name 和 age 欄位。

**使用類別**

定義類別後，您可以像使用任何其他資料型態一樣使用它：

Person john = Person("John Doe", 30);

john.greet(); // 列印 "Hello, my name is John Doe!"

List<Person> people = [john, Person("Jane Doe", 25)];

for (Person person in people) {

print(person.name);

} // 列印 "John Doe" 和 "Jane Doe"

此程式碼會建立一個 Person 物件、呼叫其 greet() 方法、建立一個 Person 物件的清單，然後反覆運算清單以列印每個人的姓名。

**類別的優點**

使用類別有以下優點：

* 提高程式碼的可讀性、可維護性和再使用性。
* 可以將具有相似特徵和行為的資料和功能組合在一起。
* 可以使用建構子初始化物件的狀態。
* 可以使用方法定義物件的行為。

**類別的局限性**

類別也有一些局限性：

* **不能在運行程式中建立類別**：類別必須在程式碼的靜態部分定義，也就是說，它們不能在運行程式中建立。
* **不能繼承多個類別**：Dart 中的類別只能繼承一個父類別。
* **不能使用抽象方法**：抽象方法只能在介面中定義。

**使用抽象類別**

要克服類別繼承的局限性，您可以使用抽象類別（Abstract Class）。抽象類別是一種特殊類別，其中可以定義抽象方法。抽象方法沒有實作，必須由其子類別實作。

abstract class 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);

}

在此範例中，Shape 是抽象類別，其中定義了 area() 和 perimeter() 抽象方法。Circle 和 Rectangle 類別繼承 Shape 類別，並實作其抽象方法。

**使用介面**

要克服類別繼承的另一個局限性，您可以使用介面（Interface）。介面是一種特殊類別，其中可以定義方法和屬性，但不能定義實作。類別可以使用介面來定義其契約，但它們不必繼承介面。

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 Square implements Shape {

double sideLength;

@override

double get area() => sideLength \* sideLength;

@override

double get perimeter() => 4 \* sideLength;

}

在此範例中，Shape 是介面，其中定義了 area() 和 perimeter() 方法。Circle 和 Square 類別實作 Shape 介面，並提供其方法的實作。

**結論**

類別是 Dart 中用於建立自訂義資料型態的基本結構。它們具有許多優點，但也有局限性。抽象類別和介面可用於克服類別繼承的局限性。

## Dart 中類的高級用法

除了基本的類定義和使用之外，Dart 中的類還支援一些高級用法，可以讓您更靈活地構建和組織您的代碼。

### 泛型類

泛型類（Generic Class）可以讓您定義帶有參數的類，這些參數可以是任何其他類型。這使得代碼更加靈活和可重用，因為它可以適應不同的資料類型。

例如，我們可以創建一個泛型類 Box 來存儲任意類型的值：

class Box<T> {

T value;

Box(this.value);

T get() => value;

void set(T newValue) {

value = newValue;

}

}

在這個例子中，T 是泛型參數，代表 Box 存儲的值的類型。我們可以創建一個 Box<String> 來存儲字串，或者創建一個 Box<int> 來存儲整數：

Box<String> stringBox = Box<String>("Hello, world!");

print(stringBox.get()); // 輸出 "Hello, world!"

Box<int> intBox = Box<int>(10);

print(intBox.get()); // 輸出 10

泛型類還可以用於定義泛型方法和構造函數。

### 混合類

混合類（Mixin Class）可以讓您將多個類的功能組合到一個類中。這對於共用代碼和實現多重繼承很有用。

例如，我們可以創建一個 Printable 混合類，為任何類添加列印功能：

mixin Printable {

void print() {

print("This object is printable.");

}

}

然後，我們可以將 Printable 混入另一個類：

class Person implements Printable {

String name;

int age;

Person(this.name, this.age);

@override

void print() {

super.print(); // 調用父類的方法

print("Name: $name");

print("Age: $age");

}

}

現在，Person 類具有 Printable 混合類中的 print() 方法：

Person john = Person("John Doe", 30);

john.print();

### 工廠構造函數

工廠構造函數（Factory Constructor）是一種特殊類型的構造函數，可以用來創建更複雜的類實例。工廠構造函數可以返回任何類型的物件，而不僅僅是當前類的實例。

例如，我們可以創建一個工廠構造函數來創建 Person 實例，並根據提供的年齡自動生成姓名：

class Person {

String name;

int age;

Person(this.name, this.age);

factory Person.fromAge(int age) {

if (age < 0) {

throw ArgumentError("Age cannot be negative");

}

return Person(generateName(age), age);

}

static String generateName(int age) {

if (age < 18) {

return "Child";

} else if (age < 65) {

return "Adult";

} else {

return "Senior";

}

}

}

現在，我們可以使用工廠構造函數來創建 Person 實例：

Person john = Person.fromAge(30);

print(john.name); // 輸出 "Adult"

### 私有成員

私有成員（Private Member）只能在定義它們的類內部訪問。這有助於封裝類內部實現細節，並防止意外修改。

要定義私有成員，請在成員名稱前面加上 \_ 底線：

class Person {

String \_name; // 私有欄位

int age;

Person(this.\_name, this.age);

String get name => \_name; // 公共 getter 方法

void setName(String newName) {

\_name = newName;

}

}

在上面的例子中，\_name 是私有欄位，只能在 Person 類內部訪問。但是，我們可以通過公共 name getter 和 setName setter 方法來訪問和修改私有欄位。

### 靜態成員

靜態成員（Static Member）與特定的類實例無關，而是與整個類相關聯。靜態成員可以用作類級別的常量、方法或屬性。

要定義靜態成員，請在成員名稱前面加上 static 關鍵字：

class Person {

static const int maxAge = 120; // 靜態常量

static void printGreeting() {

print("Hello from Person class!");

**靜態方法**

靜態方法可以用作類級別的函數，與特定的類實例無關。它們通常用於提供與類相關的實用功能，例如工廠構造函數或輔助方法。

例如，我們可以創建一個靜態方法來計算兩個數的平方和：

class MathUtils {

static int squareSum(int a, int b) {

return a \* a + b \* b;

}

}

int result = MathUtils.squareSum(10, 5);

print(result); // 輸出 125

**靜態屬性**

靜態屬性可以用作類級別的變數，與特定的類實例無關。它們通常用於存儲與類相關的全域資料，例如常量或配置值。

例如，我們可以創建一個靜態屬性來存儲應用程式的版本號：

class AppConfig {

static const String version = "1.0.0";

}

print(AppConfig.version); // 輸出 "1.0.0"

**擴展類**

擴展類（Extension Class）可以讓您為現有類添加新的功能，而無需修改其原始代碼。這對於向協力廠商類添加功能或修復已知問題很有用。

要創建一個擴展類，請使用 extension 關鍵字，後接擴展類的名稱和要擴展的類名：

extension StringExtensions on String {

String toUppercaseFirstLetter() {

if (isEmpty) return this;

return this[0].toUpperCase() + this.substring(1);

}

}

String message = "hello, world!";

print(message.toUppercaseFirstLetter()); // 輸出 "Hello, world!"

在這個例子中，StringExtensions 是擴展類，它為 String 類添加了一個新的 toUppercaseFirstLetter() 方法。

**抽象類別和介面**

**抽象類別**

抽象類別是一種特殊類型的類，其中可以定義抽象方法。抽象方法沒有實作，必須由其子類實作。抽象類別用於定義通用的介面或行為，而無需提供具體的實現細節。

例如，我們可以創建一個抽象類別 Shape 來定義形狀的基本特徵：

abstract class Shape {

double get area();

double get perimeter();

}

抽象類別 Shape 定義了 area() 和 perimeter() 抽象方法，子類必須實作這些方法才能提供具體的形狀面積和周長計算。

**介面**

介面與抽象類別類似，但它們只能定義方法和屬性，而不能定義實作。介面用於定義契約或規範，類可以實現這些契約或規範。

例如，我們可以創建一個介面 Drawable 來定義可繪製物件的規範：

interface Drawable {

void draw();

}

介面 Drawable 定義了 draw() 方法，任何實現此介面的類都必須提供 draw() 方法的實作。

**總結**

Dart 中類的高級用法提供了強大的功能，可以讓您更靈活地構建和組織您的代碼。通過泛型類、混合類、工廠構造函數、私有成員、靜態成員、擴展類、抽象類別和介面，您可以創建更具表達力和可維護性的代碼。

## Dart 中的類別（Class 2）：

在 Dart 中,class是用來定義物件的藍圖或範本。透過class可以創建具有特定屬性和方法的物件實例。以下是如何表示及使用class的方式:

1. **定義class**

class Rectangle {

double width;

double height;

Rectangle(this.width, this.height);

double get area => width \* height;

}

在這個例子中,我們定義了一個名為Rectangle的類別,它包含兩個實例變數width和height,以及一個建構子和一個area的getter方法。

1. **創建物件實例**

var rect1 = Rectangle(3.0, 4.0);

var rect2 = Rectangle(5.0, 6.0);

print(rect1.area); *// 輸出: 12.0*

print(rect2.area); *// 輸出: 30.0*

我們可以使用class的建構子來創建物件實例,並透過實例存取其屬性和方法。

1. **類別方法和getter/setter**

class Circle {

double \_radius;

Circle(this.\_radius);

double get area => Math.pi \* \_radius \* \_radius;

set radius(double value) {

if (value < 0) {

throw ArgumentError('Radius cannot be negative');

}

\_radius = value;

}

static double circumference(double radius) {

return 2 \* Math.pi \* radius;

}

}

在上面這個例子中,我們定義了一個Circle類別,它包含一個私有實例變數\_radius,以及一個area的getter、一個radius的setter和一個靜態方法circumference。

1. **繼承和多態**

class Shape {

double area() {

throw UnimplementedError('Area not implemented');

}

}

class Rectangle extends Shape {

double width, height;

Rectangle(this.width, this.height);

@override

double area() => width \* height;

}

class Circle extends Shape {

double \_radius;

Circle(this.\_radius);

@override

double area() => Math.pi \* \_radius \* \_radius;

}

這個例子展示了如何使用繼承和多態。我們定義了一個抽象基類Shape,然後讓Rectangle和Circle繼承自它。每個子類都必須實作area()方法,但具體實作細節可以因類別而有所不同。

總的來說,在 Dart 中使用class可以實現物件導向程式設計,讓您更容易組織和維護程式碼。透過適當地設計類別、屬性和方法,您可以創建出清晰、模組化和可重用的程式碼。

## Dart 中類的高級用法

Dart 中的 class 除了基本的用法之外,還提供了一些高級用法,可以讓您的程式碼更加靈活和強大。以下是一些 class 的高級用法:

1. **泛型類別**

泛型類別可以使用參數化類型,以提高程式碼的靈活性和重用性。

class Stack<T> {

final List<T> \_items = [];

void push(T item) => \_items.add(item);

T pop() => \_items.removeLast();

}

void main() {

var stackOfNumbers = Stack<int>();

stackOfNumbers.push(1);

stackOfNumbers.push(2);

print(stackOfNumbers.pop()); *// 輸出: 2*

}

1. **mixin**

mixin 可以為類別添加額外的功能,而不需要繼承。它提供了一種代碼重用的機制。

mixin Walkable {

void walk() {

print('Walking...');

}

}

class Person with Walkable {

*// 其他代碼...*

}

void main() {

var person = Person();

person.walk(); *// 輸出: Walking...*

}

1. **介面**

介面定義了一個類別應該實作的方法和屬性,可以確保類別符合特定的規範。

abstract class Flyable {

void fly();

}

class Bird implements Flyable {

@override

void fly() {

print('Bird is flying');

}

}

1. **工廠建構子**

工廠建構子可以返回一個現有的實例或創建新的實例,從而實現更加靈活的物件創建方式。

class Logger {

static final Logger \_instance = Logger.\_internal();

Logger.\_internal();

factory Logger() {

return \_instance;

}

void log(String message) {

print(message);

}

}

1. **擴展方法**

使用 extension 關鍵字,您可以為現有的類別添加新的方法,而無需修改原始類別的代碼。

extension NumberExtension on int {

bool isEven() {

return this % 2 == 0;

}

}

void main() {

print(4.isEven()); *// 輸出: true*

print(5.isEven()); *// 輸出: false*

}

這些只是 Dart 中 class 的一些高級用法。透過利用這些功能,您可以創建更加靈活、模組化和可重用的程式碼。不過,在使用這些高級功能時,也需要注意可讀性和可維護性,以確保代碼的質量。

## 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。

## 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。

**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。

## 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。

**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。**

**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;

}

**連結 python , c 程式**

Dart 可以連接 Python 和 C 程式。Dart 提供了多種機制來實現與其他語言的交互操作，包括：

* **平臺通道（Platform Channels）**：平臺通道是一種用於與原生平台（例如 Android、iOS 或桌面）代碼通信的機制。您可以使用平臺通道來調用 Python 或 C 代碼，並將結果傳遞回 Dart 代碼。
* **WebAssembly**：WebAssembly 是一種低級位元組碼格式，可以在 Web 流覽器中運行。您可以使用 WebAssembly 來載入和執行 Python 或 C 代碼，並與 Dart 代碼進行交互。
* **FFI（Foreign Function Interface）**：FFI 是一種用於在 Dart 代碼中調用 C 代碼的庫。您可以使用 FFI 來直接調用 C 函數，並訪問 C 資料結構。

以下是一些具體的示例：

* **使用平臺通道連接 Python**：您可以使用 flutter\_python 外掛程式來在 Dart 代碼中調用 Python 代碼。該外掛程式提供了用於啟動 Python 解譯器、執行 Python 代碼和獲取結果的 API。
* **使用 WebAssembly 連接 C**：您可以使用 dart:ffi 庫來載入和執行 WebAssembly 模組。該庫提供了用於載入模組、調用模組中的函數和訪問模組中資料的 API。
* **使用 FFI 連接 C**：您可以使用 dart:ffi 庫來直接調用 C 函數。該庫提供了用於定義 C 函數簽名、分配和釋放 C 記憶體以及調用 C 函數的 API。

與 Python 交互操作:

Dart 可以通過 FFI (Foreign Function Interface) 來調用 Python 的原生代碼。FFI 允許 Dart 直接調用其他語言編寫的函數。有一些協力廠商庫可以幫助實現這一點,例如:

* dart:ffi: Dart 官方提供的 FFI 庫。
* py-ffi: 一個用於在 Dart 中調用 Python 的庫。

與 C 程式交互操作:

Dart 可以通過以下幾種方式與 C 程式進行交互操作:

1. **FFI (Foreign Function Interface)**

與 Python 交互操作類似,Dart 也可以通過 FFI 直接調用 C 程式庫中的函數。dart:ffi 庫就是官方提供的 FFI 解決方案。

1. **Dart Native Extensions**

Native Extensions 允許開發人員使用 C/C++ 編寫 Dart 擴展,以提供對系統級功能的訪問。這可以用於實現無法在純 Dart 代碼中實現的功能,例如訪問特定硬體或系統 API。

1. **Dart 與 C 程式庫綁定**

一些流行的 C 程式庫已經有了與 Dart 的綁定庫,可以直接在 Dart 代碼中使用。比如 curl、OpenSSL 等。

無論是與 Python 還是 C 進行交互操作,主要思路都是利用 FFI 機制,允許 Dart 直接調用其他語言編寫的函數和庫。不過,實現細節可能因語言和具體庫而有所不同。在進行交互操作時,需要注意內存管理、數據類型轉換等問題,以確保應用程式的正確性和穩定性。

以下是一些可能有用的資源：

* **Flutter 平臺通道文檔**：<https://docs.flutter.dev/platform-integration/platform-channels>
* **WebAssembly 官方網站**：<https://webassembly.org/>
* **Dart FFI 庫文檔**：<https://pub.dev/packages/ffi>
* **flutter\_python 外掛程式**：<https://pub.dev/>

**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。

**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。

**mixin 的高級用法**

Dart 中的 mixin 是一種非常強大的功能,它允許您在多個類層次結構中重用代碼。除了基本的用法之外,mixin 還有一些高級用法,可以讓您的代碼更加靈活和可擴展。以下是一些常見的高級用法:

1. **在 mixin 中使用構造函數**

雖然 mixin 本身不能有自己的構造函數,但您可以在 mixin 中定義一個帶有常量構造函數的抽象類,然後在混入該 mixin 的類中實現該構造函數。

mixin Initializable {

Initializable() {

print('Initializing...');

init();

}

void init();

}

class MyClass with Initializable {

MyClass() {

print('MyClass constructor');

}

@override

void init() {

print('Initialized');

}

}

void main() {

MyClass(); *// 輸出: Initializing... MyClass constructor Initialized*

}

1. **使用 mixin 實現多重繼承**

由於 Dart 不支持傳統的多重繼承,但您可以使用 mixin 來組合多個類的行為。這使得您可以從不同的源中獲取所需的功能,而不會產生鑽石問題。

mixin Walkable {

void walk() => print('Walking...');

}

mixin Flyable {

void fly() => print('Flying...');

}

class Superhero extends Object with Walkable, Flyable {}

void main() {

Superhero hero = Superhero();

hero.walk(); *// 輸出: Walking...*

hero.fly(); *// 輸出: Flying...*

}

1. **在 mixin 中使用靜態成員**

mixin 可以定義靜態成員,這些靜態成員可以被混入該 mixin 的任何類所使用。這為您提供了一種共享公用方法和常量的便捷方式。

mixin Utils {

static double pi = 3.14159;

static double calculateArea(double radius) {

return pi \* radius \* radius;

}

}

class Circle with Utils {

double radius;

Circle(this.radius);

double get area => calculateArea(radius);

}

void main() {

Circle c = Circle(5);

print(c.area); *// 輸出: 78.53975*

print(Utils.pi); *// 輸出: 3.14159*

}

1. **使用 mixin 實現介面**

雖然 Dart 有專門的介面語法,但您也可以使用抽象類和 mixin 來實現類似的功能。這可以讓您的代碼更加靈活,因為 mixin 可以包含實現細節,而不僅僅是定義方法簽名。

abstract class Printable {

void printData();

}

mixin PrintableMixin implements Printable {

String data;

@override

void printData() {

print(data);

}

}

class MyClass with PrintableMixin {

MyClass(this.data);

}

void main() {

MyClass obj = MyClass('Hello, World!');

obj.printData(); *// 輸出: Hello, World!*

}

1. **使用 mixin 實現裝飾器模式**

通過將 mixin 與抽象類相結合,您可以輕鬆地實現裝飾器模式。這種模式允許您動態地為對象添加新的行為,而無需修改原始類。

abstract class Component {

void operation();

}

class ConcreteComponent extends Component {

@override

void operation() {

print('Concrete operation');

}

}

mixin Decorator implements Component {

late Component component;

@override

void operation() {

component.operation();

addedBehavior();

}

void addedBehavior();

}

class ConcreteDecoratorA extends ConcreteComponent with Decorator {

@override

void addedBehavior() {

print('Added behavior from Decorator A');

}

}

class ConcreteDecoratorB extends ConcreteComponent with Decorator {

@override

void addedBehavior() {

print('Added behavior from Decorator B');

}

}

void main() {

Component c1 = ConcreteComponent();

c1.operation(); *// 輸出: Concrete operation*

Component c2 = ConcreteDecoratorA();

c2.operation(); *// 輸出: Concrete operation Added behavior from Decorator A*

Component c3 = ConcreteDecoratorB();

c3.operation(); *// 輸出: Concrete operation Added behavior from Decorator B*

}

以下是一些 Dart 中 Mixin 的高級用法：

**1. 有狀態 Mixin**

Mixin 不僅可以包含方法，還可以包含狀態。這意味著您可以使用 Mixin 來創建可共享的狀態屬性和方法。例如，以下 Mixin 提供了一個 counter 屬性和一個 increment 方法：

mixin CounterMixin {

int counter = 0;

void increment() {

counter++;

}

}

您可以將此 Mixin 引入任何類中，以獲得 counter 屬性和 increment 方法。例如：

class Person with CounterMixin {

String name = 'John Doe';

void printInfo() {

print('Name: $name');

print('Counter: $counter');

}

}

void main() {

var person = Person();

person.increment();

person.printInfo(); // Output: Name: John Doe, Counter: 1

}

**2. 多個 Mixin 的組合**

一個類可以引入多個 Mixin，以組合不同的功能。例如，以下 Mixin 提供了一個 greet 方法和一個 age 屬性：

Dart

mixin GreeterMixin {

String greet() {

return 'Hello!';

}

}

mixin AgeMixin {

int age = 30;

}

您可以將這兩個 Mixin 引入任何類中，以獲得 greet 方法和 age 屬性。例如：

Dart

class Student with GreeterMixin, AgeMixin {

String name = 'Alice';

void printInfo() {

print('Name: $name');

print('Age: $age');

print('Greeting: ${greet()}');

}

}

void main() {

var student = Student();

student.printInfo(); // Output: Name: Alice, Age: 30, Greeting: Hello!

}

**3. 解決命名衝突**

如果 Mixin 之間存在命名衝突，您可以使用 as 關鍵字來為 Mixin 的成員重命名。例如，以下兩個 Mixin 都定義了一個 id 屬性：

mixin IDMixin {

int id = 10;

}

mixin OtherIDMixin {

int id = 20;

}

您可以使用以下代碼將這兩個 Mixin 引入同一個類中，並避免命名衝突：

class User with IDMixin as userMixin, OtherIDMixin as otherMixin {

String name = 'Bob';

void printInfo() {

print('Name: $name');

print('User ID: ${userMixin.id}');

print('Other ID: ${otherMixin.id}');

}

}

void main() {

var user = User();

user.printInfo(); // Output: Name: Bob, User ID: 10, Other ID: 20

}

在上面的代碼中，userMixin 和 otherMixin 是為 IDMixin 和 OtherIDMixin 分別定義的別名。這使得您可以訪問這兩個 Mixin 的成員而不會發生衝突。

**4. Mixin 的條件組合**

您可以使用條件來控制 Mixin 的組合方式。例如，以下代碼僅在 debug 模式下將 LoggerMixin 引入 App 類：

class App {

void run() {

if (const bool.fromEnvironment('dart.vm.debug')) {

with LoggerMixin;

}

// App logic here

}

}

mixin LoggerMixin {

void log(String message) {

print(message);

}

}

在上面的代碼中，LoggerMixin 僅在 debug 模式下引入 App 類。這意味著只有在 debug 模式下，App 類才能訪問 log 方法。

**5. Mixin 的擴展**

Mixin 可以用於擴展現有類，為其添加新的功能或行為。這可以用於以下幾種情況：

* 為現有類添加實用程序方法或屬性。
* 將現有類的行為分組到 Mixin 中，以便在多個類之間重用。
* 創建自定義 Mixin 來滿足特定需求。

### 示例

以下是一個示例，說明如何使用 Mixin 來擴展 String 類：

mixin StringUtils {

String toUpper() {

return this.toUpperCase();

}

String toLower() {

return this.toLowerCase();

}

String trim() {

return this.trim();

}

}

extension StringExtension on String {

String toUpper() {

return this.toUpperCase();

}

String toLower() {

return this.toLowerCase();

}

String trim() {

return this.trim();

}

}

void main() {

String str = 'Hello, World!';

print(str.toUpper()); // Output: HELLO, WORLD!

print(str.toLower()); // Output: hello, world!

print(str.trim()); // Output: Hello, World!

}

在上面的示例中，StringUtils Mixin 為 String 類添加了 toUpper(), toLower() 和 trim() 方法。然後，StringExtension 類擴展了 String 類，並提供了與 StringUtils Mixin 相同的方法。

您可以根據需要創建自定義 Mixin 來滿足您的特定需求。Mixin 是一種強大的工具，可讓您重用代碼並提高 Dart 中類的可維護性。

以下是一些使用 Mixin 擴展現有類的最佳實踐：

* 將相關功能組合到單個 Mixin 中。
* 為 Mixin 使用有意義的命名。
* 使用 as 關鍵字來解決命名衝突。
* 在可能的情況下使用條件來控制 Mixin 的組合方式。

**abstract 的高級用法**

Dart 中的 abstract 關鍵字用於定義抽象類和抽象方法,它們不能被直接實例化或調用,而是需要在子類中實現。除了基本用法之外,abstract 還有一些高級用法,可以幫助您編寫更乾淨、更靈活的代碼。以下是一些常見的高級用法:

1. **抽象類工廠方法**

你可以在抽象類中定義一個工廠方法,該方法負責創建具體子類的實例。這樣做有助於隱藏實現細節,並提供了更大的靈活性。

abstract class Vehicle {

factory Vehicle(String type) {

switch (type) {

case 'car':

return Car();

case 'truck':

return Truck();

default:

throw ArgumentError('Invalid vehicle type');

}

}

void drive();

}

class Car implements Vehicle {

@override

void drive() {

print('Driving a car');

}

}

class Truck implements Vehicle {

@override

void drive() {

print('Driving a truck');

}

}

void main() {

final car = Vehicle('car');

car.drive(); *// 輸出: Driving a car*

final truck = Vehicle('truck');

truck.drive(); *// 輸出: Driving a truck*

}

1. **抽象類方法**

你可以在抽象類中定義抽象方法,而不需要提供實現。這些方法必須在子類中實現,這樣可以確保子類遵循所需的協定。

abstract class Shape {

double get area;

double get perimeter;

}

class Circle implements Shape {

final double radius;

Circle(this.radius);

@override

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

@override

double get perimeter => 2 \* pi \* radius;

}

1. **抽象類混合 (Mixins)**

在 Dart 中, abstract 類可以作為 mixin 來使用,提供了代碼重用的能力。mixin 允許你在多個類層次結構中共享行為和功能。

abstract class Walkable {

void walk() {

print('Walking...');

}

}

abstract class Flyable {

void fly() {

print('Flying...');

}

}

class Bird extends Object with Walkable, Flyable {}

void main() {

final bird = Bird();

bird.walk(); *// 輸出: Walking...*

bird.fly(); *// 輸出: Flying...*

}

1. **多態和介面**

抽象類可以與接口一起使用,利用多態來實現更靈活的設計。你可以將對象視為抽象類型,從而編寫更通用的代碼。

abstract class Shape {

double get area;

double get perimeter;

}

class Circle implements Shape {

final double radius;

Circle(this.radius);

@override

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

@override

double get perimeter => 2 \* pi \* radius;

}

void printShapeInfo(Shape shape) {

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

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

}

void main() {

final circle = Circle(5);

printShapeInfo(circle);

}

1. **抽象類測試**

使用抽象類可以更輕鬆地進行單元測試。你可以創建抽象類的模擬實現,並在測試中使用它們來驗證代碼的正確性。

abstract class DataSource {

Future<List<String>> fetchData();

}

class MockDataSource implements DataSource {

@override

Future<List<String>> fetchData() async {

return ['mock', 'data'];

}

}

*// 在測試中使用 MockDataSource*

void main() {

final dataSource = MockDataSource();

*// 對 dataSource 進行測試...*

}

### 高級用法

以下是一些 Dart 中 abstract 的高級用法：

**1. 創建抽象類**

抽象類可以用於定義共性行為或接口。例如，以下代碼定義了一個抽象類 Animal，該類具有 eat(), sleep() 和 makeSound() 三個抽象方法：

abstract class Animal {

void eat();

void sleep();

void makeSound();

}

在上面的代碼中，Animal 是一個抽象類，它具有 eat(), sleep() 和 makeSound() 三個抽象方法。這些方法沒有具體的實現，它們必須在子類中實現。

**2. 實現抽象方法**

抽象方法必須在子類中實現。例如，以下代碼定義了 Dog 和 Cat 兩個子類，它們分別實現了 Animal 類的抽象方法：

class Dog extends Animal {

@override

void eat() {

print('Dog is eating');

}

@override

void sleep() {

print('Dog is sleeping');

}

@override

void makeSound() {

print('Woof!');

}

}

class Cat extends Animal {

@override

void eat() {

print('Cat is eating');

}

@override

void sleep() {

print('Cat is sleeping');

}

@override

void makeSound() {

print('Meow!');

}

}

在上面的代碼中，Dog 和 Cat 兩個子類分別實現了 Animal 類的 eat(), sleep() 和 makeSound() 抽象方法。

**3. 使用抽象類來定義接口**

抽象類可以用於定義接口，從而強制子類實現特定的行為。例如，以下代碼定義了一個抽象類 Drawable，該類具有 draw() 抽象方法：

abstract class Drawable {

void draw();

}

在上面的代碼中，Drawable 是一個抽象類，它具有 draw() 抽象方法。這個抽象方法強制任何實現 Drawable 接口的類都必須實現 draw() 方法。

**4. 使用抽象類來實現多重繼承**

Dart 不支持多重繼承，但可以使用抽象類來實現多重繼承的效果。例如，以下代碼定義了 Speaker 和 Walker 兩個抽象類，它們都具有 speak() 和 walk() 方法：

abstract class Speaker {

void speak();

}

abstract class Walker {

void walk();

}

然後，您可以創建一個子類，該子類同時繼承 Speaker 和 Walker 兩個抽象類：

class Person implements Speaker, Walker {

@override

void speak() {

print('Person is speaking');

}

@override

void walk() {

print('Person is walking');

}

}

在上面的代碼中，Person 類同時繼承了 Speaker 和 Walker 兩個抽象類，從而實現了多重繼承的效果。

abstract 是一種強大的工具，可讓您在 Dart 中創建更抽象和可重用的代碼。通過了解 abstract 的高級用法，您可以創建更複雜和靈活的應用程序。

以下是一些使用 abstract 的最佳實踐：

* 將抽象類用於定義共性行為或接口。
* 使用抽象方法來強制子類實現特定的行為。
* 使用抽象類來實現多重繼承的效果。
* 避免過度使用抽象類，因為這會使您的代碼變得難以理解和維護。

**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：

Dart

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。

**extends 的高級用法**

Dart 中的 extends 關鍵字用於實現單一繼承,允許子類從父類繼承方法、屬性和其他成員。除了基本用法外，extends 還有一些高級用法,可以幫助你編寫更加靈活和優雅的代碼。以下是一些常見的高級用法:

1. **工廠構造函數**

在子類中,你可以定義一個工廠構造函數,該構造函數可以返回父類或其他子類的實例。這在需要根據某些條件動態選擇要實例化的類型時非常有用。

abstract class Shape {

factory Shape(String type) {

switch (type) {

case 'circle':

return Circle(2.0);

case 'square':

return Square(3.0);

default:

throw ArgumentError('Invalid shape type');

}

}

double get area;

}

class Circle extends Shape {

final double radius;

Circle(this.radius);

@override

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

}

class Square extends Shape {

final double side;

Square(this.side);

@override

double get area => side \* side;

}

void main() {

final circle = Shape('circle');

final square = Shape('square');

print(circle.area); *// 12.566370614359172*

print(square.area); *// 9.0*

}

1. **重寫操作符**

在 Dart 中,你可以重寫操作符,如 +、-、[] 等,以實現自定義的運算行為。在子類中重寫操作符可以為你的類型添加新的功能和行為。

class Vector2D {

final double x, y;

Vector2D(this.x, this.y);

Vector2D operator +(Vector2D other) {

return Vector2D(x + other.x, y + other.y);

}

}

void main() {

final v1 = Vector2D(1, 2);

final v2 = Vector2D(3, 4);

final sum = v1 + v2;

print(sum.x); *// 4.0*

print(sum.y); *// 6.0*

}

1. **調用父類成員**

在子類中,你可以使用 super 關鍵字來調用父類的方法或構造函數。這在你需要在子類中擴展或修改父類行為時非常有用。

class Person {

String name;

Person(this.name);

void introduce() {

print('Hello, my name is $name');

}

}

class Student extends Person {

String major;

Student(String name, this.major) : super(name);

@override

void introduce() {

super.introduce();

print('I am studying $major');

}

}

void main() {

final student = Student('Alice', 'Computer Science');

student.introduce();

*// Output:*

*// Hello, my name is Alice*

*// I am studying Computer Science*

}

1. **mixins**

在 Dart 中,你可以使用 with 關鍵字將 mixins 添加到一個類中。這可以讓你在不同的類層次結構中重用代碼,實現更好的模塊化和可重用性。

mixin Walkable {

void walk() {

print('Walking...');

}

}

mixin Flyable {

void fly() {

print('Flying...');

}

}

class Animal {}

class Bird extends Animal with Walkable, Flyable {}

void main() {

final bird = Bird();

bird.walk(); *// Walking...*

bird.fly(); *// Flying...*

}

1. **泛型類**

在 Dart 中,你可以定義泛型類,並通過 extends 關鍵字實現泛型類的繼承。這可以提高代碼的靈活性和可重用性。

class Stack<T> {

final List<T> \_items = [];

void push(T item) {

\_items.add(item);

}

T pop() {

return \_items.removeLast();

}

}

class StackWithMax<T extends num> extends Stack<T> {

T get max {

if (\_items.isEmpty) {

throw StateError('Stack is empty');

}

return \_items.reduce(math.max);

}

}

void main() {

final stack = StackWithMax<int>();

stack.push(1);

stack.push(3);

stack.push(2);

print(stack.max); *// 3*

}

這些只是 extends 關鍵字的一些高級用法。通過有效地利用繼承、工廠構造函數、操作符重載、mixins 和泛型,你可以編寫出更加乾淨、靈活和可重用的 Dart 代碼。

### 高級用法

以下是一些 Dart 中 extends 的高級用法：

**1. 單繼承**

Dart 支持單繼承，這意味著一個類只能從一個父類繼承。例如，以下代碼定義了一個 Animal 類和一個 Dog 類，其中 Dog 類繼承了 Animal 類：

class Animal {

String name;

void eat() {

print('Animal is eating');

}

void sleep() {

print('Animal is sleeping');

}

}

class Dog extends Animal {

String breed;

void bark() {

print('Woof!');

}

}

在上面的代碼中，Dog 類繼承了 Animal 類的 name, eat() 和 sleep() 屬性和方法。此外，Dog 類還具有自己的 breed 屬性和 bark() 方法。

**2. 方法覆蓋**

子類可以覆蓋父類的方法，以提供自己的實現。例如，以下代碼覆蓋了 Animal 類的 eat() 方法：

class Dog extends Animal {

@override

void eat() {

print('Dog is eating bones');

}

}

在上面的代碼中，Dog 類覆蓋了 Animal 類的 eat() 方法，以提供自己的實現。覆蓋的方法將取代父類的方法。

**3. 抽象類**

抽象類不能被直接創建實例，它們只能被用作子類的基類。抽象類可以包含抽象方法，這些方法必須在子類中實現。例如，以下代碼定義了一個抽象類 Animal，該類具有 eat() 和 sleep() 兩個抽象方法：

Dart

abstract class Animal {

void eat();

void sleep();

}

在上面的代碼中，Animal 是一個抽象類，它具有 eat() 和 sleep() 兩個抽象方法。這些方法沒有具體的實現，它們必須在子類中實現。

**4. 多重繼承**

Dart 不支持多重繼承，但可以使用抽象類來模擬多重繼承。例如，以下代碼定義了 Speaker 和 Walker 兩個抽象類，它們都具有 speak() 和 walk() 方法：

Dart

abstract class Speaker {

void speak();

}

abstract class Walker {

void walk();

}

然後，您可以創建一個子類，該子類同時繼承 Speaker 和 Walker 兩個抽象類：

class Person implements Speaker, Walker {

@override

void speak() {

print('Person is speaking');

}

@override

void walk() {

print('Person is walking');

}

}

在上面的代碼中，Person 類同時繼承了 Speaker 和 Walker 兩個抽象類，從而實現了多重繼承的效果。

### 注意事項

* 在使用 extends 時，請注意單繼承的限制。
* 在覆蓋方法時，請確保子類的方法與父類的方法具有兼容的簽名。
* 抽象類不能被直接創建實例，它們只能被用作子類的基類。
* 雖然 Dart 不支持多重繼承，但可以使用抽象類來模擬多重繼承。

**factory 的高級用法**

Dart 中的 factory 關鍵字用於定義工廠構造函數,它們可以返回一個現有對象或創建一個新對象。除了基本用法外,factory 還有一些高級用法,可以讓你的代碼更加靈活和高效。以下是一些常見的高級用法:

1. **單例模式**

單例模式是一種常見的設計模式,它確保一個類只有一個實例。你可以使用 factory 構造函數來實現單例模式。

class Singleton {

static final Singleton \_instance = Singleton.\_internal();

factory Singleton() {

return \_instance;

}

Singleton.\_internal();

*// 其他成員...*

}

void main() {

final singleton1 = Singleton();

final singleton2 = Singleton();

print(identical(singleton1, singleton2)); *// true*

}

1. **複雜對象初始化**

在某些情況下,創建對象可能需要複雜的初始化邏輯。你可以使用 factory 構造函數來封裝這些邏輯,使你的代碼更加簡潔和易於維護。

class Point {

final double x, y;

const Point(this.x, this.y);

factory Point.fromPolar(double radius, double angle) {

final x = radius \* cos(angle);

final y = radius \* sin(angle);

return Point(x, y);

}

}

void main() {

final point = Point.fromPolar(2.0, pi / 4);

print(point.x); *// 1.4142135623730951*

print(point.y); *// 1.4142135623730951*

}

1. **返回子類型對象**

factory 構造函數可以返回當前類或子類的實例,這在需要根據一些條件動態選擇要創建的對象類型時非常有用。

abstract class Shape {

factory Shape(String type) {

switch (type) {

case 'circle':

return Circle(2.0);

case 'square':

return Square(3.0);

default:

throw ArgumentError('Invalid shape type');

}

}

double get area;

}

class Circle implements Shape {

final double radius;

Circle(this.radius);

@override

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

}

class Square implements Shape {

final double side;

Square(this.side);

@override

double get area => side \* side;

}

void main() {

final circle = Shape('circle');

final square = Shape('square');

print(circle.area); *// 12.566370614359172*

print(square.area); *// 9.0*

}

1. **對象池和緩存**

factory 構造函數可以用於實現對象池或緩存機制,以提高對象創建和獲取的效率。

class ExpensiveObject {

static final \_cache = <String, ExpensiveObject>{};

factory ExpensiveObject(String key) {

if (\_cache.containsKey(key)) {

return \_cache[key]!;

} else {

final obj = ExpensiveObject.\_internal(key);

\_cache[key] = obj;

return obj;

}

}

ExpensiveObject.\_internal(this.key);

final String key;

*// 其他成員...*

}

1. **延遲實例化**

在某些情況下,你可能希望延遲實例化對象,直到它們真正被需要時才創建。factory 構造函數可以用於實現這種延遲實例化策略。

class LazyObject {

static LazyObject? \_instance;

factory LazyObject() {

\_instance ??= LazyObject.\_internal();

return \_instance!;

}

LazyObject.\_internal() {

print('LazyObject initialized');

}

*// 其他成員...*

}

void main() {

print('Before instantiation');

final obj1 = LazyObject();

print('After first instantiation');

final obj2 = LazyObject();

print('After second instantiation');

}

這些只是 factory 關鍵字的一些高級用法。通過有效地利用工廠模式和延遲實例化,你可以編寫出更加高效、靈活和可維護的 Dart 代碼。

一些 Dart 中 factory 的高級用法：

**1. 從緩存中返回實例**

工廠構造函數可以用於從緩存中返回現有實例。例如，以下代碼定義了一個 Logger 類，該類使用工廠構造函數從緩存中返回單例實例：

class Logger {

String name;

bool mute = false;

static final Map<String, Logger> \_cache = <String, Logger>{};

factory Logger(String name) {

if (\_cache.containsKey(name)) {

return \_cache[name]!;

} else {

var logger = Logger.\_internal(name);

\_cache[name] = logger;

return logger;

}

}

Logger.\_internal(this.name);

}

在上面的代碼中，Logger 類的工廠構造函數首先檢查緩存中是否包含指定名稱的 Logger 實例。如果存在，則直接從緩存中返回該實例。否則，創建一個新的 Logger 實例並將其添加到緩存中，然後返回該實例。

**2. 進行輸入驗證**

工廠構造函數可以用於進行輸入驗證。例如，以下代碼定義了一個 Point 類，該類的工廠構造函數用於驗證 x 和 y 參數的值：

class Point {

int x;

int y;

factory Point(int x, int y) {

if (x < 0 || y < 0) {

throw ArgumentError('Coordinates must be non-negative');

}

return Point.\_internal(x, y);

}

Point.\_internal(this.x, this.y);

}

在上面的代碼中，Point 類的工廠構造函數首先檢查 x 和 y 參數的值是否小於 0。如果小於 0，則拋出 ArgumentError 異常。否則，創建一個新的 Point 實例並返回該實例。

**3. 返回子類的實例**

工廠構造函數可以用於返回子類的實例。例如，以下代碼定義了一個 Shape 類和兩個子類 Circle 和 Rectangle。Shape 類的工廠構造函數根據 type 參數的值返回 Circle 或 Rectangle 的實例：

abstract class Shape {

factory Shape(String type) {

switch (type) {

case 'circle':

return Circle();

case 'rectangle':

return Rectangle();

default:

throw ArgumentError('Invalid shape type');

}

}

}

class Circle extends Shape {

// ...

}

class Rectangle extends Shape {

// ...

}

在上面的代碼中，Shape 類的工廠構造函數根據 type 參數的值返回 Circle 或 Rectangle 的實例。如果 type 參數的值不是 'circle' 或 'rectangle'，則拋出 ArgumentError 異常。

### 注意事項

* 工廠構造函數不能使用 this 關鍵字。
* 工廠構造函數不能使用 super 關鍵字調用父類的構造函數。
* 工廠構造函數可以有多個參數。

#### 來源

1. [github.com/briand787b/learning\_dart](https://github.com/briand787b/learning_dart)
2. [github.com/songdongliang/flutter\_app](https://github.com/songdongliang/flutter_app)

**build 的高級用法**

Dart 中的 build 方法通常與框架和庫密切相關,尤其是在構建 UI 時發揮著重要作用。

Dart 中的 build 函數是 Flutter 框架中一個重要的函數，用於構建小部件。它接受一個 BuildContext 對象作為參數，並返回一個小部件。build 函數是構建 Flutter 應用程序的核心，它可以通過各種方式使用來創建複雜的用戶界面。

1. **在 Flutter 中構建 Widget 樹**

在 Flutter 中,build 方法是用於構建 Widget 樹的核心方法。它返回一個 Widget 作為應用程序 UI 的根節點。Flutter 框架在需要重新渲染 UI 時會調用此方法。

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

title: 'My App',

home: Scaffold(

appBar: AppBar(

title: Text('Home'),

),

body: Center(

child: Text('Hello, World!'),

),

),

);

}

}

1. **在 Flutter 中使用條件渲染**

build 方法中的條件邏輯可以用於根據應用程序狀態動態渲染不同的 Widget。這可以避免創建不必要的 Widget,從而提高性能。

class MyWidget extends StatefulWidget {

@override

\_MyWidgetState createState() => \_MyWidgetState();

}

class \_MyWidgetState extends State<MyWidget> {

bool \_isLoading = true;

@override

void initState() {

super.initState();

\_loadData();

}

Future<void> \_loadData() async {

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

setState(() {

\_isLoading = false;

});

}

@override

Widget build(BuildContext context) {

if (\_isLoading) {

return CircularProgressIndicator();

} else {

return Text('Data loaded');

}

}

}

1. **在 Angular Dart 中渲染組件**

在 Angular Dart 中,build 方法用於渲染組件的視圖。它返回一個 ComponentFactory 對象,該對象表示要渲染的組件。

@Component(

selector: 'my-component',

template: '<h1>{{title}}</h1>',

)

class MyComponent implements OnInit {

String title = 'Hello, World!';

@override

ngOnInit() {

*// 初始化組件...*

}

@override

ComponentFactory<dynamic> build() {

return ComponentFactory(

'my-component',

view: () => [

Text('$title'),

],

);

}

}

1. **在服務器端渲染中使用 build**

當在服務器端渲染應用程序時,build 方法可以用於生成初始 HTML 內容。這可以提高初始渲染性能,並為搜索引擎優化 (SEO) 做好準備。

import 'package:angular/angular.dart';

@Component(

selector: 'my-component',

template: '<h1>{{title}}</h1>',

)

class MyComponent {

String title = 'Hello, World!';

@override

ComponentFactory<dynamic> build() {

return ComponentFactory(

'my-component',

view: () => [

Text('$title'),

],

);

}

}

void main() async {

final renderer = await createRenderer();

final html = renderer.renderComponent(MyComponent());

print(html); *// 輸出渲染後的 HTML*

}

1. **在自定義庫和框架中使用 build**

如果你正在構建自己的庫或框架,你可以使用 build 方法作為構建對象或組件的入口點。這樣做可以提供更好的擴展性和可定制性。

abstract class Widget {

Widget build(BuildContext context);

}

class Text extends Widget {

final String data;

Text(this.data);

@override

Widget build(BuildContext context) {

return TextWidget(data);

}

}

class TextWidget extends Widget {

final String data;

TextWidget(this.data);

@override

Widget build(BuildContext context) {

*// 渲染文本 Widget*

return ...;

}

}

這些只是 build 方法的一些高級用法。

以下是一些 Dart 中 build 函數的高級用法：

* **使用條件邏輯構建小部件**

build 函數可以用於根據條件邏輯構建不同的子小部件。例如，以下代碼將根據 isDarkMode 變量的值構建一個明亮或黑暗的主題小部件：

Widget build(BuildContext context) {

final isDarkMode = Theme.of(context).brightness == Brightness.dark;

return Scaffold(

appBar: AppBar(

title: Text('My App'),

backgroundColor: isDarkMode ? Colors.black : Colors.white,

),

body: Center(

child: Text(

'Hello, world!',

style: TextStyle(

color: isDarkMode ? Colors.white : Colors.black,

),

),

),

);

}

* **使用列表構建小部件**

build 函數可以用於使用 ListView 或 GridView 等小部件構建列表或網格。例如，以下代碼將構建一個顯示字符串列表的 ListView：

Dart

Widget build(BuildContext context) {

final items = ['Item 1', 'Item 2', 'Item 3'];

return ListView.builder(

itemCount: items.length,

itemBuilder: (context, index) {

final item = items[index];

return ListTile(

title: Text(item),

);

},

);

}

* **使用自定義小部件構建小部件**

build 函數可以用於使用自定義小部件構建小部件。這可以讓您創建可重複使用的 UI 組件，從而簡化您的代碼並提高其可維護性。例如，以下代碼將創建一個自定義的 Button 小部件：

Dart

class MyButton extends StatelessWidget {

final String text;

final VoidCallback onPressed;

const MyButton({Key? key, required this.text, required this.onPressed}) : super(key: key);

@override

Widget build(BuildContext context) {

return ElevatedButton(

onPressed: onPressed,

child: Text(text),

);

}

}

* **使用動畫構建小部件**

build 函數可以用於使用 Animation 或 AnimationController 對象構建動畫小部件。這可以讓您創建具有動態效果的 UI。例如，以下代碼將創建一個旋轉動畫小部件：

class RotatingWidget extends StatefulWidget {

const RotatingWidget({Key? key}) : super(key: key);

@override

\_RotatingWidgetState createState() => \_RotatingWidgetState();

}

class \_RotatingWidgetState extends State<RotatingWidget>

with SingleTickerProviderStateMixin {

late AnimationController \_controller;

@override

void initState() {

super.initState();

\_controller = AnimationController(

vsync: this,

duration: const Duration(seconds: 1),

)..repeat();

}

@override

void dispose() {

\_controller.dispose();

super.dispose();

}

@override

Widget build(BuildContext context) {

return AnimatedRotation(

turns: \_controller.value,

duration: const Duration(seconds: 1),

child: Container(

width: 100,

height: 100,

color: Colors.red,

),

);

}

}

這些只是一些 Dart 中 build 函數的高級用法。build 函數是一個功能強大的工具，可讓您創建複雜的用戶界面。通過了解其功能，您可以創建美觀且響應的 Flutter 應用程序。

#### 來源

1. [github.com/IzuUni/mafdss\_mobile\_app](https://github.com/IzuUni/mafdss_mobile_app)
2. [stackoverflow.com/questions/65117669/how-do-you-update-a-futurebuilder-when-the-sqlite-db-it-gets-data-from-gets-new](https://stackoverflow.com/questions/65117669/how-do-you-update-a-futurebuilder-when-the-sqlite-db-it-gets-data-from-gets-new)
3. [github.com/Mukul-Dagar/project\_f](https://github.com/Mukul-Dagar/project_f)
4. [stackoverflow.com/questions/69807636/flutter-run-code-before-or-after-execution-of-function-that-was-passed-as-a](https://stackoverflow.com/questions/69807636/flutter-run-code-before-or-after-execution-of-function-that-was-passed-as-a)
5. [stackoverflow.com/questions/66260860/how-to-create-a-flutter-stateful-widget-that-rotate-its-child](https://stackoverflow.com/questions/66260860/how-to-create-a-flutter-stateful-widget-that-rotate-its-child)

**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：

Dart

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();

}

}

**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。

Dart

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。

Dart

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。

**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);

}