Dart 简介

本文会通过示例向你展示 Dart 语言用法的简单的介绍。

想要深度学习 Dart 语言，请访问左侧栏 **Dart 开发语言** 的具体内容。

想要了解 Dart 的核心库，访问 核心库文档。你也可以查看 Dart 语言的速查表，来获得一些快速上手的指导。

Hello World

每个应用都有一个顶层的 `main()` 函数来作为运行入口。没有指定返回类型的方法的返回类型会推导为 `void`。你可以使用顶层函数 `print()` 来将一段文本输出显示到控制台：

```dart
void main() {
  print('Hello, World!');
}
```

你可以阅读 the `main()` function 了解更多 CLI 的可选参数。

变量

虽然 Dart 是 代码类型安全 的语言，你仍然可以用 `var` 来定义变量，而不用显式指定它们的类型。由于其支持类型推断，因此大多数变量的类型会由它们的初始化内容决定：

```dart
var name = 'Voyager I';
var year = 1977;
var antennaDiameter = 3.7;
var flybyObjects = ['Jupiter', 'Saturn', 'Uranus', 'Neptune'];
var image = {
  'tags': ['saturn'],
  'url': '//path/to/saturn.jpg'
};
```

你可以 阅读更多 Dart 中关于变量的内容，包括变量的默认值，`final` 和 `const` 关键字以及静态类型等。

流程控制语句

Dart 支持常用的流程控制语句：

```dart
if (year >= 2001) {
  print('21st century');
} else if (year >= 1901) {
  print('20th century');
}

for (final object in flybyObjects) {
  print(object);
}

for (int month = 1; month <= 12; month++) {
  print(month);
}

while (year < 2016) {
  year += 1;
}
```

你可以阅读更多 Dart 中关于控制流程语句的内容，包括 `break` 和 `continue` 关键字、 `switch` 语句和 `case` 子句 以及 `assert` 语句。

函数

我们建议 为每个函数的参数以及返回值都指定类型：

```dart
int fibonacci(int n) {
  if (n == 0 || n == 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

var result = fibonacci(20);
```

`=>` (**胖箭头**) 简写语法用于仅包含一条语句的函数。该语法在将匿名函数作为参数传递时非常有用：

```dart
flybyObjects.where((name) => name.contains('turn')).forEach(print);
```

上面的示例除了向你展示了匿名函数（上例中传入 `where()` 函数的参数即是一个匿名函数）外，还向你展示了将函数作为参数使用的方式：上面示例将顶层函数 `print()` 作为参数传给了 `forEach()` 函数。

你可以 阅读更多 Dart 中有关函数的内容，包括可选参数、默认参数值以及词法作用域。

注释

Dart 通常使用双斜杠 `//` 作为注释的开始。

```dart
// This is a normal, one-line comment.

/// This is a documentation comment, used to document libraries,
/// classes, and their members. Tools like IDEs and dartdoc treat
/// doc comments specially.

/* Comments like these are also supported. */
```

你可以 阅读更多 Dart 中有关注释的内容，包括文档工具的工作原理。

导入 (Import)

使用 `import` 关键字来访问在其它库中定义的 API。

```dart
// Importing core libraries
import 'dart:math';

// Importing libraries from external packages
import 'package:test/test.dart';

// Importing files
import 'path/to/my_other_file.dart';
```

你可以 阅读更多 Dart 中有关库和可见性的内容，包括库前缀、`show` 和 `hide` 关键字以及通过 `deferred` 关键字实现的懒加载。

类 (Class)

下面的示例中向你展示了一个包含三个属性、两个构造函数以及一个方法的类。其中一个属性不能直接赋值，因此它被定义为一个 getter 方法（而不是变量）。该方法使用字符串插值来打印字符串文字内变量的字符串。

```dart
class Spacecraft {
  String name;
  DateTime? launchDate;

  // Read-only non-final property
  int? get launchYear => launchDate?.year;

  // Constructor, with syntactic sugar for assignment to members.
  Spacecraft(this.name, this.launchDate) {
    // Initialization code goes here.
  }

  // Named constructor that forwards to the default one.
  Spacecraft.unlaunched(String name) : this(name, null);

  // Method.
  void describe() {
    print('Spacecraft: $name');
    // Type promotion doesn't work on getters.
    var launchDate = this.launchDate;
    if (launchDate != null) {
      int years = DateTime.now().difference(launchDate).inDays ~/ 365;
      print('Launched: $launchYear ($years years ago)');
    } else {
      print('Unlaunched');
    }
  }
}
```

你可以 阅读更多 关于字符串的内容，包括字符串插值、表达式以及 `toString()` 方法。

你可以像下面这样使用 `Spacecraft` 类：

```dart
var voyager = Spacecraft('Voyager I', DateTime(1977, 9, 5));
voyager.describe();

var voyager3 = Spacecraft.unlaunched('Voyager III');
voyager3.describe();
```

你可以 阅读更多 Dart 中有关类的内容，包括初始化列表、可选的 `new` 和 `const` 关键字、重定向构造函数、由 `factory` 关键字定义的工厂构造函数以及 Getter 和 Setter 方法等等。

枚举类型 (Enum)

枚举类型的取值范围是一组预定义的值或实例。

下面这个简单的枚举示例定义了一组行星类别：

```dart
enum PlanetType { terrestrial, gas, ice }
```

下面是一个增强型枚举的示例，定义了一组行星类的常量实例，即太阳系的行星：

```dart
/// Enum that enumerates the different planets in our solar system
/// and some of their properties.
enum Planet {
  mercury(planetType: PlanetType.terrestrial, moons: 0, hasRings: false),
  venus(planetType: PlanetType.terrestrial, moons: 0, hasRings: false),
  // ···
  uranus(planetType: PlanetType.ice, moons: 27, hasRings: true),
  neptune(planetType: PlanetType.ice, moons: 14, hasRings: true);

  /// A constant generating constructor
  const Planet(
      {required this.planetType, required this.moons, required this.hasRings});

  /// All instance variables are final
  final PlanetType planetType;
  final int moons;
  final bool hasRings;

  /// Enhanced enums support getters and other methods
  bool get isGiant =>
      planetType == PlanetType.gas || planetType == PlanetType.ice;
}
```

你可以这样使用 `Planet` 枚举：

```dart
final yourPlanet = Planet.earth;

if (!yourPlanet.isGiant) {
  print('Your planet is not a "giant planet".');
}
```

你可以 阅读更多 Dart 中有关枚举的内容，包括增强型枚举的限制条件、枚举默认包含的属性、如何获取枚举值的名称以及在 `switch` 语句中使用枚举等等。

扩展类（继承）

Dart 支持单继承。

```dart
class Orbiter extends Spacecraft {
  double altitude;

  Orbiter(super.name, DateTime super.launchDate, this.altitude);
}
```

你可以 阅读更多 Dart 中有关类继承的内容，比如可选的 `@override` 注解等等。

Mixins

Mixin 是一种在多个类层次结构中重用代码的方法。下面的是声明一个 Mixin 的做法：

```dart
mixin Piloted {
  int astronauts = 1;

  void describeCrew() {
    print('Number of astronauts: $astronauts');
  }
}
```

现在你只需使用 Mixin 的方式继承这个类就可将该类中的功能添加给其它类。

```dart
class PilotedCraft extends Spacecraft with Piloted {
  // ···
}
```

自此，`PilotedCraft` 类中就包含了 `astronauts` 字段以及 `describeCrew()` 方法。

你可以 阅读更多 关于 Mixin 的内容。

接口和抽象类

所有的类都隐式定义成了一个接口。因此，任意类都可以作为接口被实现。

```dart
class MockSpaceship implements Spacecraft {
  // ···
}
```

你可以阅读更多关于 隐式接口 或者 `interface` 关键词 的内容。

你可以创建一个被任意具体类扩展（或实现）的抽象类。抽象类可以包含抽象方法（不含方法体的方法）。

```dart
abstract class Describable {
  void describe();

  void describeWithEmphasis() {
    print('=========');
    describe();
    print('=========');
  }
}
```

任意一个扩展了 `Describable` 的类都拥有 `describeWithEmphasis()` 方法，这个方法又会去调用实现类中实现的 `describe()` 方法。

你可以 阅读更多 关于抽象类和抽象方法的内容。

异步

使用 `async` 和 `await` 关键字可以让你避免回调地狱 (Callback Hell) 并使你的代码更具可读性。

```dart
const oneSecond = Duration(seconds: 1);
// ···
Future<void> printWithDelay(String message) async {
  await Future.delayed(oneSecond);
  print(message);
}
```

上面的方法相当于：

```dart
Future<void> printWithDelay(String message) {
  return Future.delayed(oneSecond).then((_) {
    print(message);
  });
}
```

如下一个示例所示，`async` 和 `await` 关键字有助于使异步代码变得易于阅读。

```dart
Future<void> createDescriptions(Iterable<String> objects) async {
  for (final object in objects) {
    try {
      var file = File('$object.txt');
      if (await file.exists()) {
        var modified = await file.lastModified();
        print(
            'File for $object already exists. It was modified on $modified.');
        continue;
      }
      await file.create();
      await file.writeAsString('Start describing $object in this file.');
    } on IOException catch (e) {
      print('Cannot create description for $object: $e');
    }
  }
}
```

你也可以使用 `async*` 关键字，其可以为你提供一个可读性更好的方式去生成 Stream。

```dart
Stream<String> report(Spacecraft craft, Iterable<String> objects) async* {
  for (final object in objects) {
    await Future.delayed(oneSecond);
    yield '${craft.name} flies by $object';
  }
}
```

你可以 阅读更多 关于异步支持的内容，包括异步函数、`Future`、`Stream` 以及异步循环（`await for`）。

异常

使用 `throw` 关键字抛出一个异常：

```dart
if (astronauts == 0) {
  throw StateError('No astronauts.');
}
```

使用 `try` 语句配合 `on` 或 `catch`（两者也可同时使用）关键字来捕获一个异常:

```dart
Future<void> describeFlybyObjects(List<String> flybyObjects) async {
  try {
    for (final object in flybyObjects) {
      var description = await File('$object.txt').readAsString();
      print(description);
    }
  } on IOException catch (e) {
    print('Could not describe object: $e');
  } finally {
    flybyObjects.clear();
  }
}
```

注意上述代码是异步的；同步代码以及异步函数中得代码都可以使用 `try` 捕获异常。

你可以 阅读更多 关于异常的内容，包括栈追踪、`rethrow` 关键字以及 Error 和 Exception 之间的区别。

Important concepts

As you continue to learn about the Dart language, keep these facts and concepts in mind:

- Everything you can place in a variable is an *object*, and every object is an instance of a *class*. Even numbers, functions, and `null` are objects. With the exception of `null` (if you enable sound null safety), all objects inherit from the `Object` class.

  > 版本提示
  >
  > Null safety was introduced in Dart 2.12. Using null safety requires a language version of at least 2.12.

- Although Dart is strongly typed, type annotations are optional because Dart can infer types. In `var number = 101`, `number` is inferred to be of type `int`.
- If you enable null safety, variables can't contain `null` unless you say they can. You can make a variable nullable by putting a question mark (`?`) at the end of its type. For example, a variable of type `int?` might be an integer, or it might be `null`. If you *know* that an expression never evaluates to `null` but Dart disagrees, you can add `!` to assert that it isn't null (and to throw an exception if it is). An example: `int x = nullableButNotNullInt!`
- When you want to explicitly say that any type is allowed, use the type `Object?` (if you've enabled null safety), `Object`, or—if you must defer type checking until runtime—the special type `dynamic`.
- Dart supports generic types, like `List<int>` (a list of integers) or `List<Object>` (a list of objects of any type).
- Dart supports top-level functions (such as `main()`), as well as functions tied to a class or object (*static* and *instance methods*, respectively). You can also create functions within functions (*nested* or *local functions*).
- Similarly, Dart supports top-level *variables*, as well as variables tied to a class or object (static and instance variables). Instance variables are sometimes known as *fields* or *properties*.
- Unlike Java, Dart doesn't have the keywords `public`, `protected`, and `private`. If an identifier starts with an underscore (_), it's private to its library. For details, see Libraries and imports.

- *Identifiers* can start with a letter or underscore (`_`), followed by any combination of those characters plus digits.
- Dart has both *expressions* (which have runtime values) and *statements* (which don't). For example, the conditional expression `condition ? expr1 : expr2` has a value of `expr1` or `expr2`. Compare that to an if-else statement, which has no value. A statement often contains one or more expressions, but an expression can't directly contain a statement.
- Dart tools can report two kinds of problems: *warnings* and *errors*. Warnings are just indications that your code might not work, but they don't prevent your program from executing. Errors can be either compile-time or run-time. A compile-time error prevents the code from executing at all; a run-time error results in an exception being raised while the code executes.

其他资源

核心库文档 中会有更多的文档和代码示例。你也可以查阅 Dart API 文档，里面也常常会有示例代码。本网站的代码风格遵循 Dart 代码风格指南。