Dart 库教程

翻译自 Dart 官网，查看原文。

当前版本：2.1.0 (stable)

完成度：100%

该教程为你展示了下面这些库的主要特性，这些库被包含在所有的 Dart 平台中。

dart:core

内置类型、集合和其他核心功能。该库在每一个 Dart 程序中被自动引入。

dart:async

支持异步编程，包括 Future 和 Stream 等类。

dart:math

数学常数和函数，外加随机数生成器。

dart:convert

用于在不同的数据表示之间进行转换的编码器和解码器，包括 JSON 和 UTF-8。

该篇文章只是一个概览；它只覆盖 dart:* 中的一小部分库而且不包含第三方库。平台相关的 dart:io 和 dart:html 库分别包含在 dart:io 教程 和 dart:html 教程 中。

其他查找库信息的地方是 pub.dartlang.org 和 Dart web 开发者库指引。 你可以在 Dart API 参考 中找到所有的 dart:* 库；而如果你在使用 Flutter，则是 Flutter API 参考。

> DartPad 小提示：你可以通过将其拷贝到 DartPad 来尝试本页的代码。

dart:core - 数值、集合、字符串和其他

库 dart:core（API 参考）提供了一组小而重要的内置功能。这个库在所有的 Dart 程序中被自动引入。

打印到控制台

顶级方法 **print()** 接受单个参数（任意类型）并将这个对象的字符串值（通过 **toString()** 返回）显示在控制台上。

```
print(anObject);
print('I drink $tea.');
```

要了解更多关于基本的字符串和 **toString()** 的信息，请参阅语言教程中的 String 章节。

数值

库 dart:core 定义了 num、int 和 double 类，它们拥有操作数值的一些基本方法。

你可以通过 int 和 double 的 **parse()** 方法将字符串分别转换为整数或浮点数：

```
assert(int.parse('42') == 42);
assert(int.parse('0x42') == 66);
assert(double.parse('0.50') == 0.5);
```

或者使用 num 的 parse() 方法，它在可能的情况下创建一个整数，否则创建一个浮点数：

```
assert(num.parse('42') is int);
assert(num.parse('0x42') is int);
assert(num.parse('0.50') is double);
```

要指定一个整数的进制，添加 **radix** 参数：

```
assert(int.parse('42', radix: 16) == 66);
```

使用 **toString()** 方法将一个 int 或者 double 装换成字符串。要指定小数点右边的位数，请使用 toStringAsFixed()。要指定字符串中的有效位数，请使用 toStringAsPrecision() ：

```
// 转换 int 为字符串
assert(42.toString() == '42');

// 转换 double 为字符串
assert(123.456.toString() == '123.456');

// 指定小数点后的位数
assert(123.456.toStringAsFixed(2) == '123.46');

// 指定数值的有效位数
assert(123.456.toStringAsPrecision(2) == '1.2e+2');
assert(double.parse('1.2e+2') == 120.0);
```

要了解更多信息，请参阅 int、double 和 num 的 API 文档。也请参阅 dart:math 章节。

字符串和正则表达式

字符串在 Dart 中表示 UTF-16 编码单元组成的一个不可变序列。语言教程中有关于 字符串 的更详细信息。你可以使用正则表达式（RegExp 对象）在字符串中查找和替换部分字符串。

字符串类定义了诸如 **split()**、**contains()**、**startsWith()**、**endsWith** 这样的以及其他更多的方法。

在字符串中查找

你可以在字符串中查找一个指定模式的位置，或者检查字符串是否以其开头或结尾。比如：

```
// 检查一个字符串是否包含另一个字符串
assert('Never odd or even'.contains('odd'));

// 检查一个字符串是否开始于另一个字符串
assert('Never odd or even'.startsWith('Never'));

// 检查一个字符串是否结束于另一个字符串
assert('Never odd or even'.endsWith('even'));

// 查找一个字符串在另一个字符串中的位置
assert('Never odd or even'.indexOf('odd') == 6);
```

从字符串中提取数据

你可以从一个字符串中获取独立的字符分别作为字符串或者整数。准确来说，你得到的其实是独立的 UTF-16 编码单元；像高音音符这样的高位字符 ('\u{1D11E}') 则为在一块的两个编码单元。

你也可以提取一个子字符串或者拆分一个字符串为一个子字符串列表：

```
// 抓取一个子字符串
assert('Never odd or even'.substring(6, 9) == 'odd');

// 使用一个字符串模式拆分字符串
var parts = 'structured web apps'.split(' ');
assert(parts.length == 3);
assert(parts[0] == 'structured');

// 使用索引获取 UTF-16 编码单元（作为字符串）
assert('Never odd or even'[0] == 'N');

// 使用空字符串作为 split() 的参数来获得
// 所有字符（作为字符串）的列表；便于迭代
for (var char in 'hello'.split('')) {
  print(char);
}

// 获取字符串中所有的 UTF-16 编码单元
var codeUnitList =
    'Never odd or even'.codeUnits.toList();
assert(codeUnitList[0] == 78);
```

大小写转换

你可以很容易地转换一个字符串为它的大写或小写形式：

```
// 转换为大写
assert('structured web apps'.toUpperCase() ==
    'STRUCTURED WEB APPS');

// 转换为小写
assert('STRUCTURED WEB APPS'.toLowerCase() ==dart
    'structured web apps');
```

> 说明：这些方法并不适用于所有语言。比如，土耳其字母的 I 转换就是错误的。

修剪和空字符串

使用 **trim()** 移除前面和后面的所有空白。要检查一个字符串是否是空的（长度为0），使用 **isEmpty**。

```
// 修剪字符串
assert('  hello  '.trim() == 'hello');

// 检查字符串是否是空的
assert(''.isEmpty);

// 只有空白符的字符串不是空的
assert('  '.isNotEmpty);
```

替换字符串的部分内容

字符串是不可变的对象，意味着你可以创建它们但是不可以修改它们。如果你仔细观察 String API 索引，你会发现没有一个方法真正改变了一个字符串的状态。比如，方法 **replaceAll()** 不会修改原来的字符串：

```
var greetingTemplate = 'Hello, NAME!';
var greeting =
    greetingTemplate.replaceAll(RegExp('NAME'), 'Bob');

// greetingTemplate 没有改变
assert(greeting != greetingTemplate);
```

构建一个字符

要程序化的创建一个字符串，你可以使用 StringBuffer。一个 StringBuffer 不生成新的字符串除非 **toString()** 方法被调用。方法 **writeAll()** 有一个可选的第二个参数让你可以指定一个分隔符——在这里是一个空格。

```
var sb = StringBuffer();
sb
  ..write('Use a StringBuffer for ')
  ..writeAll(['efficient', 'string', 'creation'], ' ')
  ..write('.');

var fullString = sb.toString();

assert(fullString ==
    'Use a StringBuffer for efficient string creation.');
```

正则表达式

RegExp 类提供了与 JavaScript 中正则表达式相同的功能。使用正则表达式来高效地查找和匹配字符串中的模式。

```
// 这是一个表示一个或多个数字的正则表达式
var numbers = RegExp(r'\d+');

var allCharacters = 'llamas live fifteen to twenty years';
var someDigits = 'llamas live 15 to 20 years';

// contains() 可以使用正则表达式
assert(!allCharacters.contains(numbers));
assert(someDigits.contains(numbers));

// 替换所有匹配的部分为另一个字符串
var exedOut = someDigits.replaceAll(numbers, 'XX');
assert(exedOut == 'llamas live XX to XX years');
```

你也可以直接使用 RegExp 类。Match 类提供了对于正则表达式匹配结果的访问方法。

```
var numbers = RegExp(r'\d+');
var someDigits = 'llamas live 15 to 20 years';

// 检查正则表达式在字符串中是否有一个匹配
assert(numbers.hasMatch(someDigits));

// 循环遍历所有匹配结果
for (var match in numbers.allMatches(someDigits)) {
  print(match.group(0)); // 15，然后是 20
}
```

更多信息

参考 String API 索引 来获取方法的完整列表。另请参阅 StringBuffer、Pattern、RegExp 和 Match 的 API 索引。

集合

Dart 附带了核心的集合 API，包含了与列表、Set 和 Map 相关的类。

列表

就如语言教程所展示的，你可以使用字面量来创建和初始化 列表。除此以外，你还可以使用 List 的构造函数。List 类同时定义了一些往列表添加和从列表移除项目的方法。

```
// 使用一个集合的构造函数
var vegetables = List();

// 或者简单地使用字面量
var fruits = ['apples', 'oranges'];

// 往列表添加
fruits.add('kiwis');

// 往列表添加多个项目
fruits.addAll(['grapes', 'bananas']);
```

```
// 获取列表的长度
assert(fruits.length == 5);

// 移除单个项目
var appleIndex = fruits.indexOf('apples');
fruits.removeAt(appleIndex);
assert(fruits.length == 4);

// 从列表中移除所有项目
fruits.clear();
assert(fruits.length == 0);
```

使用 **indexOf()** 来获取一个对象在列表中的索引：

```
var fruits = ['apples', 'oranges'];

// 使用索引访问项目
assert(fruits[0] == 'apples');

// 在列表中查找一个项目
assert(fruits.indexOf('apples') == 0);
```

使用 **sort()** 方法对列表进行排序。你可以提供一个排序函数用于比较两个对象。该排序函数必须返回 < 0 的值来表示“更小”、0 表示相等、> 0 的值表示“更大”。下面的例子使用 **compareTo()**，该方法定义在 Compareable 类中并且被字符串所实现。

```
var fruits = ['bananas', 'apples', 'oranges'];

// 对列表进行排序
fruits.sort((a, b) => a.compareTo(b));
assert(fruits[0] == 'apples');
```

列表是参数化类型的，所以你可以指定一个列表应该包含的类型：

```
// 这个列表应该只包含字符串
var fruits = List<String>();

fruits.add('apples');
var fruit = fruits[0];
assert(fruit is String);
```

```
fruits.add(5); // 错误: 'int' 不可以赋值给 'String'
```

参见 List API 索引 获取列表的完整方法列表。

Set

Dart 中的 set 是独特元素的无序集合。因为 set 是无序的，你不可以使用索引（为值）获取项目。

```
var ingredients = Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);
assert(ingredients.length == 3);

// 添加重复项目是无效的
ingredients.add('gold');
assert(ingredients.length == 3);

// 从 set 中移除项目
ingredients.remove('gold');
assert(ingredients.length == 2);
```

使用 **contains()** 和 **containsAll()** 来检查一个或多个对象是否在一个 set 中：

```
var ingredients = Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);

// 检查一个项目是否在 set 中
assert(ingredients.contains('titanium'));

// 检查所有项目是否都在 set 中
assert(ingredients.containsAll(['titanium', 'xenon']));
```

交集是一个所有项目同时在两个其他 set 中的 set。

```
var ingredients = Set();
ingredients.addAll(['gold', 'titanium', 'xenon']);

// 创建两个 set 的交集
var nobleGases = Set.from(['xenon', 'argon']);
var intersection = ingredients.intersection(nobleGases);
assert(intersection.length == 1);
assert(intersection.contains('xenon'));
```

参见 Set API 索引 获取 set 的完整方法列表。

Maps

一个 map，通常被称作“字典”或者“映射”，是键值对的无序集合。Map 关联一个键到一些值为了便于取回。不像 JavaScript，Dart 对象不是 map。

你可以使用字面量声明一个 map，或者使用一个传统的构造函数：

```
// Map 经常使用字符串作为键
var hawaiianBeaches = {
  'Oahu': ['Waikiki', 'Kailua', 'Waimanalo'],
  'Big Island': ['Wailea Bay', 'Pololu Beach'],
  'Kauai': ['Hanalei', 'Poipu']
};

// Map 可以由一个构造函数构造
var searchTerms = Map();

// Map 是参数类型的，你可以指定
// 哪些类型可以作为键和值
var nobleGases = Map<int, String>();
```

使用中括号语法来添加、获取和设置 map 的项目。使用 **remove()** 方法从 map 中移除一个键和它对应的值。

```
var nobleGases = {54: 'xenon'};

// 使用一个键取回一个值
assert(nobleGases[54] == 'xenon');

// 检查 map 是否包含一个键
assert(nobleGases.containsKey(54));

// 移除一个键和它对应的值
nobleGases.remove(54);
assert(!nobleGases.containsKey(54));
```

你可以从一个 map 中取回所有的值或所有的键：

```
var hawaiianBeaches = {
  'Oahu': ['Waikiki', 'Kailua', 'Waimanalo'],
  'Big Island': ['Wailea Bay', 'Pololu Beach'],
  'Kauai': ['Hanalei', 'Poipu']
};

// 获取所有的键作为一个无序集合
//  (一个 Iterable)
var keys = hawaiianBeaches.keys;

assert(keys.length == 3);
assert(Set.from(keys).contains('Oahu'));

// 获取所有的值作为一个无序集合
// (一个列表组成的 Iterable)
var values = hawaiianBeaches.values;
assert(values.length == 3);
assert(values.any((v) => v.contains('Waikiki')));
```

要检查一个 map 是否包含一个键，使用 **containsKey()**。因为 map 的值可以为空，你不能简单地依赖获取并判断值是否为空来决定一个键是否存在。

```
var hawaiianBeaches = {
  'Oahu': ['Waikiki', 'Kailua', 'Waimanalo'],
  'Big Island': ['Wailea Bay', 'Pololu Beach'],
  'Kauai': ['Hanalei', 'Poipu']
};

assert(hawaiianBeaches.containsKey('Oahu'));
assert(!hawaiianBeaches.containsKey('Florida'));
```

当你仅想当一个键不存在于一个 map 中时才指定一个值到该键，使用 **putIfAbsent()** 方法。你必须指定一个方法返回该值。

```
var teamAssignments = {};
teamAssignments.putIfAbsent(
    'Catcher', () => pickToughestKid());
assert(teamAssignments['Catcher'] != null);
```

参见 Map API 索引 获取 map 的完整方法列表。

通用集合方法

列表、Set 和 Map 共享了一些在许多集合类型中都可找到的通用方法。一些通用方法定义在 Iterable 类中，该类被列表和 Set 所实现。

> 说明：尽管 Map 没有实现 Iterable，你仍可以使用 **keys** 和 **values** 属性从 map 中获取 Iterable。

使用 **isEmpty** 或 **isNotEmpty** 来检查一个列表、set 或 map 是否拥有项目：

```
var coffees = [];
var teas = ['green', 'black', 'chamomile', 'earl grey'];
assert(coffees.isEmpty);
assert(teas.isNotEmpty);
```

要应用一个函数到一个列表、set 或 map 的每一个元素上，你可以使用 **forEach()**：

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

teas.forEach((tea) => print('I drink $tea'));
```

当你在一个 map 上调用 **forEach()** 时，你的函数必须接受两个参数（键和值）。

```
hawaiianBeaches.forEach((k, v) {
  print('I want to visit $k and swim at $v');
  // I want to visit Oahu and swim at
  // [Waikiki, Kailua, Waimanalo], 等等
});
```

Iterable 提供了 **map()** 方法，它以单个对象给你所有的结果：

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

var loudTeas = teas.map((tea) => tea.toUpperCase());
loudTeas.forEach(print);
```

> 说明：由 **map()** 返回的对象是一个“懒求值”的 Iterable：直到你请求返回的对象时你的函数才会被调用。

要强制你的方法在每一个项目上被立即调用，使用 **map().toList()** 或 **map().toSet()**：

```
var loudTeas =
    teas.map((tea) => tea.toUpperCase()).toList();
```

使用 Iterable 的 **where()** 方法来获取所有符合某个条件的元素。使用 Iterable 的 **any()** 方法和 **every()** 方法来检查是否有一些元素或所有元素符合某个条件。

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

// Chamomile (黄春菊) 是不含咖啡因的
bool isDecaffeinated(String teaName) =>
    teaName == 'chamomile';

// 使用 where() 来查找只有
// 从提供的函数返回 true 的项目
var decaffeinatedTeas =
    teas.where((tea) => isDecaffeinated(tea));
// 或者 teas.where(isDecaffeinated)

// 使用 any() 来检查是否至少有一个
// 集合中的项目满足条件
assert(teas.any(isDecaffeinated));

// 使用 every() 来检查是否集合中
// 所有的项目都满足条件
assert(!teas.every(isDecaffeinated));
```

要获取完整的方法列表，请参见 Iterable API 索引，以及 列表、Set 和 Map 中的这些方法。

URI

Uri 类 提供编码和解码字符串用来作为 URI（就是你可能知道的 URL）的函数。这些函数处理 URI 中的特殊字符，比如 **&** 和 **=**。Uri 类也包含解析和获取 URI 的各个部件的方法——host、port、schema 等等。

编码和解码完全限定的 URI

要编码和解码 URI 中除去那些有特殊意义（比如 **/**、**:**、**&**、**#**）之外的字符，使用 **encodeFull()** 和 **decodeFull()** 方法。这些方法对于编码和解码完全限定的 URI 是非常好的，它们会留下 URI 中完整的特殊字符。

```
var uri = 'http://example.org/api?foo=some message';

var encoded = Uri.encodeFull(uri);
assert(encoded ==
    'http://example.org/api?foo=some%20message');

var decoded = Uri.decodeFull(encoded);
assert(uri == decoded);
```

注意只有 **some** 和 **message** 之间的空格被编码了。

编码和解码 URI 部件

要编码和解码一个字符串在 URI 中有特殊意义的所有字符，包括（但不限于）**/**、**&** 和 **:**，使用 **encodeComponent()** 和 **decodeComponent** 方法。

```
var uri = 'http://example.org/api?foo=some message';

var encoded = Uri.encodeComponent(uri);
assert(encoded ==
    'http%3A%2F%2Fexample.org%2Fapi%3Ffoo%3Dsome%20message');

var decoded = Uri.decodeComponent(encoded);
assert(uri == decoded);
```

解析 URI

如果你有一个 Uri 对象或者一个 URI 字符串，你可以使用 Uri 的属性获取它的部件比如 **path**。要从一个字符串创建一个 Uri，使用静态方法 **parse()**：

```
var uri =
    Uri.parse('http://example.org:8080/foo/bar#frag');

assert(uri.scheme == 'http');
assert(uri.host == 'example.org');
assert(uri.path == '/foo/bar');
assert(uri.fragment == 'frag');
assert(uri.origin == 'http://example.org:8080');
```

参见 Uri API 索引 了解所有你可以获取的部件。

构建 URI

你可以使用 **Uri()** 构造函数从独立的部件构建 URL：

```
var uri = Uri(
    scheme: 'http',
    host: 'example.org',
    path: '/foo/bar',
    fragment: 'frag');
assert(
    uri.toString() == 'http://example.org/foo/bar#frag');
```

日期和时间

一个 DateTime 对象代表一个时间点。时区可以是 UTC 或者本地时区。

你可以使用多个构造函数创建 DateTime 对象：

```
// 获取现在的日期和时间
var now = DateTime.now();

// 使用本地时区创建一个新的 DateTime 对象
var y2k = DateTime(2000); // 2000年1月1日

// 指定月和日
y2k = DateTime(2000, 1, 2); // 2000年1月2日

// 指定日期为 UTC 时间
y2k = DateTime.utc(2000); // 2000.01.01, UTC

// 使用 Unix 时间戳指定一个日期和时间
y2k = DateTime.fromMillisecondsSinceEpoch(946684800000,
    isUtc: true);

// 解析一个 ISO 8601 格式的日期
y2k = DateTime.parse('2000-01-01T00:00:00Z');
```

一个日期的 **millisecondsSinceEpoch** 属性返回距“Unix 纪元”——UTC 时间1970年1月1日 ——的毫秒数：

```
// 2000.01.01, UTC
var y2k = DateTime.utc(2000);
assert(y2k.millisecondsSinceEpoch == 946684800000);

// 1970.01.01, UTC
var unixEpoch = DateTime.utc(1970);
assert(unixEpoch.millisecondsSinceEpoch == 0);
```

使用 Duration 类来计算两个日期的差异，并向前或向后偏移日期：

```
var y2k = DateTime.utc(2000);

// 增加一年
var y2001 = y2k.add(Duration(days: 366));
assert(y2001.year == 2001);
```

```dart
// 减去30天
var december2000 = y2001.subtract(Duration(days: 30));
assert(december2000.year == 2000);
assert(december2000.month == 12);

// 计算两个日期之间的差异
// 返回一个 Duration 对象
var duration = y2001.difference(y2k);
assert(duration.inDays == 366); // y2k 是一个闰年
```

> 警告：使用 Duration 来偏移一个 DateTime 的天数可能会出问题，因为时间转换（比如夏令时）。如果你一定要偏移天数，可以使用 UTC 日期。

要获取完整的方法列表，请参见 DateTime 和 Duration 的 API 索引。

实用工具类

核心库包含各种各样的实用工具类，可以用来排序、映射值和迭代。

比较对象

实现 Comparable 接口来表明一个对象可以与其他对象进行比较，通常用来排序。方法 **compareTo()** 返回 < 0 的值来表示"更小"、0 表示相等、> 0 的值表示"更大"。

```dart
class Line implements Comparable<Line> {
  final int length;
  const Line(this.length);

  @override
  int compareTo(Line other) => length - other.length;
}

void main() {
  var short = const Line(1);
  var long = const Line(100);
  assert(short.compareTo(long) < 0);
}
```

实现 map 的键

Dart 中的每一个对象都提供了一个整数类型的 hash 码，因此可以用来作为一个 map 的键。然而，你可以重载 **hashCode** 的 getter 来生成一个自定义的 hash 码。如果你这样做，你必须同时重载 **==** 操作符。相等的对象（使用 ==）必须拥有相同的 hash 码。一个 hash 码不一定要是唯一的，但是应该合理地分发。

```dart
class Person {
  final String firstName, lastName;

  Person(this.firstName, this.lastName);

  // 使用 Effective Java 第11章中的策略重载 hash 码
  @override
  int get hashCode {
```

```
    int result = 17;
    result = 37 * result + firstName.hashCode;
    result = 37 * result + lastName.hashCode;
    return result;
  }

  // 如果你重载 hash 码，你通常要重载 == 运算符
  @override
  bool operator ==(dynamic other) {
    if (other is! Person) return false;
    Person person = other;
    return (person.firstName == firstName &&
        person.lastName == lastName);
  }
}

void main() {
  var p1 = Person('Bob', 'Smith');
  var p2 = Person('Bob', 'Smith');
  var p3 = 'not a person';
  assert(p1.hashCode == p2.hashCode);
  assert(p1 == p2);
  assert(p1 != p3);
}
```

迭代

Iterable 类 和 Iterator 类 提供了 for-in 循环。当你想要创建一个提供 for-in 循环的迭代器类时，继承（如果可以）或者实现 Iterable。实现 Iterator 来定义真正的迭代能力。

```
class Process {
  // 代表一个处理...
}

class ProcessIterator implements Iterator<Process> {
  @override
  Process get current => ...
  @override
  bool moveNext() => ...
}

// 一个虚拟的类，它继承了 Iterable 的子类，使你可以
// 从所有的处理中迭代
class Processes extends IterableBase<Process> {
  @override
  final Iterator<Process> iterator = ProcessIterator();
}

void main() {
  // Iterable 对象可以使用 for-in
  for (var process in Processes()) {
    // 对 process 做一些处理
  }
```

```
}
```

异常

Dart 核心库定义了许多通用的异常和错误。异常是指你计划要面对和处理的情况。错误是指你不期望或计划的情况。

几个最常见的错误为：

NoSuchMethodError

当接收对象（可能是 null）未实现一个方法时抛出。

ArgumentError

当方法遇到一个不期望的参数时可以抛出。

抛出一个应用级别的异常是表明错误发生了的一个常用的方法。你可以通过实现 Exception 接口自定义一个异常:

```
class FooException implements Exception {
  final String msg;

  const FooException([this.msg]);

  @override
  String toString() => msg ?? 'FooException';
}
```

要了解更多信息，请参阅语言教程中的 异常 和 Exception API 索引。

dart:async - 异步编程

异步编程经常使用回调函数，但是 Dart 提供了其他的选项：Future 和 Stream 对象。一个 Future 就像对将来某个时刻提供结果的承诺。Stream 是一种获取一系列值的方法，例如事件。Future、Stream 还有其他更多内容包含在 dart:async 库中 (API 索引)。

> 提示：你并不总是需要直接使用 Future 或 Stream API。Dart 语言支持使用诸如 **async** 和 **await** 这样的关键字进行异步编码。请参阅语言教程中的 异步支持 了解详情。

库 dart:async 在 web 应用和命令行应用中均可试用。要使用它，引入 dart:async：

```
import 'dart:async'
```

> 版本说明：在 Dart 2.1 中，你不需要引入 dart:async 来使用 Future 和 Stream API，因为 dart:core 导出了这些类。

Future

Future 对象经常以异步方法返回值的方式出现在 Dart 库中。当一个 future “完成”时，它的值随时可用。

使用 await

在你直接使用 Future API 之前，请考虑使用 **await** 来代替。使用 **await** 的代码会比使用 Future API 的代码更容易理解。

考虑下面的函数。它使用 Future 的 **then()** 方法连续执行三个异步函数，每个函数执行前等待前面的函数执行完成。

```
runUsingFuture() {
  // ...
  findEntryPoint().then((entryPoint) {
    return runExecutable(entryPoint, args);
  }).then(flushThenExit);
}
```

而使用 await 表达式的等效代码看起来更像是同步代码：

```
runUsingAsyncAwait() async {
  // ...
  var entryPoint = await findEntryPoint();
  var exitCode = await runExecutable(entryPoint, args);
  await flushThenExit(exitCode);
}
```

一个异步函数可以从 Future 中捕获异常，例如：

```
var entryPoint = await findEntryPoint();
try {
  var exitCode = await runExecutable(entryPoint, args);
  await flushThenExit(exitCode);
} catch (e) {
  // 处理错误...
}
```

> 重要：异步函数返回 Future。如果你不想你的函数返回一个 future，那么请使用其他方案。例如，你可以在你的函数中调用一个异步函数。

要了解更多关于使用 **await** 和相关的 Dart 语言特性，请参阅 异步支持。

基本用法

你可以使用 **when()** 来调度那些在 future 完成后执行的代码。例如，**HttpRequest.getString()** 返回一个 Future，因为 HTTP 请求可能花上一段时间。使用 **then()** 来让你在 Future 完成并且承诺的字符串返回值可用时运行代码：

```
HttpRequest.getString(url).then((String result) {
  print(result);
});
```

使用 **catchError()** 来处理一个 Future 对象可能会抛出的任意错误或异常。

```
HttpRequest.getString(url).then((String result) {
  print(result);
}).catchError((e) {
  // 处理或忽略错误
});
```

这个 **then().catchError()** 模式就是异步版的 **try-catch**。

> 重要：确保在 **then()** 的返回结果上调用 **catchError()**——而不是原本 Future 的返回结果上。否则，**catchError()** 只能捕获从原本 Future 的计算过程中抛出的错误，而不是从通过 **then()** 注册的处理器中抛出的错误。

链接多个异步方法

方法 **then()** 返回一个 Future，提供一个有用的方法来以指定顺序运行多个异步函数。如果通过 **then()** 注册的回调返回一个 Future，**then()** 返回一个相同的 Future。如果这个回调返回其他任意类型的值，**then()** 以该值作为完成结果创建一个新的 Future。

```
Future result = costlyQuery(url);
result
    .then((value) => expensiveWork(value))
    .then((_) => lengthyComputation())
    .then((_) => print('Done!'))
    .catchError((exception) {
  /* 处理异常... */
});
```

在前面的例子中，这些方法以下面的顺序执行：

1. costlyQuery()
2. expensiveWork()
3. lengthyComputation()

下面是使用 await 的等效代码：

```
try {
  final value = await costlyQuery(url);
  await expensiveWork(value);
  await lengthyComputation();
  print('Done!');
} catch (e) {
  /* 处理异常... */
}
```

等待多个 future

有时你的算法需要调用多个异步函数然后等待它们全部完成后才能继续。使用 Future.wait() 静态方法来管理多个 Future 并等待他们完成：

```
Future deleteLotsOfFiles() async =>  ...
Future copyLotsOfFiles() async =>  ...
Future checksumLotsOfOtherFiles() async =>  ...

await Future.wait([
  deleteLotsOfFiles(),
  copyLotsOfFiles(),
  checksumLotsOfOtherFiles(),
]);
print('Done with all the long steps!');
```

Stream

Stream 对象出现在整个 Dart API 中，表示数据序列。例如，按钮点击等 HTML 事件通过 stream 来传递。你也可以读取一个文件作为 stream。

使用异步 for 循环

有时你可以使用一个异步 for 循环 (**await for**) 来代替 Stream API。

考虑下面的函数。它使用 Stream 的 **listen()** 方法来订阅一个列表中的文件，传递一个函数字面量来搜索每一个文件或目录。

```
void main(List<String> arguments) {
  // ...
  FileSystemEntity.isDirectory(searchPath).then((isDir) {
    if (isDir) {
      final startingDir = Directory(searchPath);
      startingDir
          .list(
              recursive: argResults[recursive],
              followLinks: argResults[followLinks])
          .listen((entity) {
        if (entity is File) {
          searchFile(entity, searchTerms);
        }
      });
    } else {
      searchFile(File(searchPath), searchTerms);
    }
  });
}
```

使用 await 表达式，并包含一个异步 for 循环 (**await for**) 的等效代码看起来更像同步代码：

```
Future main(List<String> arguments) async {
  // ...
  if (await FileSystemEntity.isDirectory(searchPath)) {
    final startingDir = Directory(searchPath);
    await for (var entity in startingDir.list(
        recursive: argResults[recursive],
        followLinks: argResults[followLinks])) {
```

```
      if (entity is File) {
        searchFile(entity, searchTerms);
      }
    }
  } else {
    searchFile(File(searchPath), searchTerms);
  }
}
```

> 重要：在使用 **await for** 之前，确保代码清晰并且你真的想要等待所有 stream 的结果。例如，你通常**不**应该在 DOM 事件监器听上使用 **await for**，因为 DOM 会不停的发送事件。如果你使用 **await for** 先后注册两个 DOM 事件监听器，第二种事件从不会被处理。

要了解更多关于 **await** 和相关 Dart 语言特性的用法，请参阅 异步支持。

监听 stream 数据

要在每个值到来时获取它，要么使用 **await for**，要么使用 **listen()** 方法监听 stream：

```
// 使用 ID 获取一个按钮并添加事件处理
querySelector('#submitInfo').onClick.listen((e) {
  // 当按钮被点击时，该代码被执行
  submitData();
});
```

在这个例子中，**onClick** 属性是 "submitInfo" 按钮提供的一个 Stream 对象。

如果你只关心一个事件，可以使用类似 **first**、**last** 或者 **single** 这样的属性获得它。要在处理事件前测试它，使用类似 **firstWhere()**、**lastWhere()** 或者 **singleWhere()** 的方法。

如果你在意一个事件的子集，你可以使用类似 **skip()**、**skipWhile()**、**take()** 、**takeWhile()** 以及 **where()** 这样的方法。

转换 stream 数据

经常地，你需要在使用 stream 的数据前改变它的格式。使用 **transform()** 方法来生成一个不同类型的 stream 数据：

```
var lines = inputStream
    .transform(utf8.decoder)
    .transform(LineSplitter());
```

这个例子使用了两个转换器。第一个使用了 utf8.decoder 来转换整数的 stream 为字符串的 stream。然后它使用了一个 LineSplitter 来转换字符串的 stream 为按行分割的 stream。这个转换器来自 dart:convert 库（参见 dart:convert 章节）。

处理错误和完成

你如何指定错误和完成处理代码取决于你使用的是异步 for 循环 (**await for**) 还是 Stream API。

如果你使用了一个异步 for 循环，那么使用 try-catch 来处理错误。异步 for 循环之后的代码将在 stream 关闭后执行。

```
Future readFileAwaitFor() async {
  var config = File('config.txt');
  Stream<List<int>> inputStream = config.openRead();

  var lines = inputStream
      .transform(utf8.decoder)
      .transform(LineSplitter());
  try {
    await for (var line in lines) {
      print('Got ${line.length} characters from stream');
    }
    print('file is now closed');
  } catch (e) {
    print(e);
  }
}
```

如果你使用 Stream API，那么通过注册一个 **onError** 监听器来处理错误。使用 **onDone()** 监听器在 stream 关闭后执行代码。

```
var config = File('config.txt');
Stream<List<int>> inputStream = config.openRead();

inputStream
    .transform(utf8.decoder)
    .transform(LineSplitter())
    .listen((String line) {
  print('Got ${line.length} characters from stream');
}, onDone: () {
  print('file is now closed');
}, onError: (e) {
  print(e);
});
```

更多信息

要了解更多 Future 和 Stream 在命令行应用中的用法，请参阅 dart:io 教程。也请参阅以下文章和教程：

- Asynchronous Programming: Futures
- Futures and Error Handling
- The Event Loop and Dart
- Asynchronous Programming: Streams
- Creating Streams in Dart

dart:math - 数学和随机

库 dart:math（API 参考）提供类似正弦、余弦、最大值和最小值这些通用的功能，还有像 *Pi* 和 *e* 这样的常数。Math 库中的大部分功能都以顶级函数的方式被实现。

要在你的应用中使用这个库，导入 dart:math.

```
import 'dart:math';
```

三角

Math 库提供了基本的三角函数：

```
// 余弦
assert(cos(pi) == -1.0);

// 正弦
var degrees = 30;
var radians = degrees * (pi / 180);
// 弧度现在是 0.52359.
var sinOf30degrees = sin(radians);
// sin 30° = 0.5
assert((sinOf30degrees - 0.5).abs() < 0.01);
```

> 说明：这些函数使用弧度，而不是度数！

最大和最小

Math 库提供了 **max()** 和 **min()** 函数：

```
assert(max(1, 1000) == 1000);
assert(min(1, -1000) == -1000);
```

数学常数

你可以在 Math 库中找到你最喜欢的——*pi*、*e* 和 其他更多的常数。

```
// 查看 Math 库来找到更多的常量
print(e); // 2.718281828459045
print(pi); // 3.141592653589793
print(sqrt2); // 1.4142135623730951
```

随机数

使用 Random 类来生成随机数。你可以选用带 seed 的 Random 构造函数。

```
var random = Random();
random.nextDouble(); // 介于 0.0 和 1.0: [0, 1)
random.nextInt(10); // 介于 0 和 9.
```

你甚至可以生成随机的布尔值：

```
var random = Random();
random.nextBool(); // true 或 false
```

更多信息

参见 Math API 参考 获取完整的方法列表。另请参阅 num、int 和 double 的API 参考。

dart:convert - 解码和编码 JSON、UTF-8 和其他

库 dart:convert（API 参考）包含 JSON 和 UTF-8 的转换器，当然也支持创建其他的转换器。JSON 是表示结构化对象和集合的一种简单的文本格式。UTF-8 是一种可表示 Unicode 字符集中所有字符的通用的可变宽度编码。

库 dart:convert 在 web 应用和命令行应用中均可使用。要使用它，导入 dart:convert。

```
import 'dart:convert';
```

解码和编码 JSON

在 Dart 中，使用 **jsonEncode()** 来解码一个 JSON 编码的字符串：

```
// 说明：在 JSON 字符串中，确保使用双引号 (")，
// 而不是单引号 (')。
// 这个字符串是 JSON，而不是 Dart。
var jsonString = '''
  [
    {"score": 40},
    {"score": 80}
  ]
''';

var scores = jsonDecode(jsonString);
assert(scores is List);

var firstScore = scores[0];
assert(firstScore is Map);
assert(firstScore['score'] == 40);
```

使用 **jsonEncode()** 编码一个受支持的对象为一个 JSON 字符串：

```
var scores = [
  {'score': 40},
  {'score': 80},
  {'score': 100, 'overtime': true, 'special_guest': null}
];

var jsonText = jsonEncode(scores);
assert(jsonText ==
    '[{"score":40},{"score":80},'
    '{"score":100,"overtime":true,'
    '"special_guest":null}]');
```

只有类型为 int、double、String、bool、null、List 或 Map（只包含字符串的 key）是可以直接编码为 JSON 的。List 和 Map 对象被递归地编码。

你有两个选项来编码那些不被直接支持的对象。第一个是调用包含第二个参数的 **encode()**：一个函数返回一个可被直接编码的对象。你的第二个选项是忽略第二个参数，这样编码器会调用对象的 **toJson()** 方法。

解码和编码 UTF-8 字符

在 Dart 中，使用 **utf8.decode()** 来解码 UTF8 编码的字节为一个 Dart 字符串：

```
List<int> utf8Bytes = [
  0xc3, 0x8e, 0xc3, 0xb1, 0xc5, 0xa3, 0xc3, 0xa9,
  0x72, 0xc3, 0xb1, 0xc3, 0xa5, 0xc5, 0xa3, 0xc3,
  0xae, 0xc3, 0xb6, 0xc3, 0xb1, 0xc3, 0xa5, 0xc4,
  0xbc, 0xc3, 0xae, 0xc5, 0xbe, 0xc3, 0xa5, 0xc5,
  0xa3, 0xc3, 0xae, 0xe1, 0xbb, 0x9d, 0xc3, 0xb1
];

var funnyWord = utf8.decode(utf8Bytes);

assert(funnyWord == 'Îñţérñåţîöñåļîžåţîờñ');
```

要转换一个 UTF-8 字符流为一个 Dart 字符串，指定 **utf8.decoder** 到 Stream 的 **transform()** 方法：

```
var lines = inputStream
    .transform(utf8.decoder)
    .transform(LineSplitter());
try {
  await for (var line in lines) {
    print('Got ${line.length} characters from stream');
  }
  print('file is now closed');
} catch (e) {
  print(e);
}
```

使用 **utf8.encode()** 来编码一个 Dart 字符串为一个 UTF-8 编码的字节列表：

```
List<int> encoded = utf8.encode('Îñţérñåţîöñåļîžåţîờñ');

assert(encoded.length == utf8Bytes.length);
for (int i = 0; i < encoded.length; i++) {
  assert(encoded[i] == utf8Bytes[i]);
}
```

其他功能

库 dart:convert 也包含 ASCII 和 ISO-8859-1 (Latin1) 的转换器。要了解详情，请参阅 dart:convert 库的 API 参考。

总结

该页为你介绍了最常用的 Dart 内置库。然而，它没有覆盖所有的内置库。其他你可能想了解的包括 dart:collection 和 dart:typed_data，还有像 Dart web 开发库 和 Flutter 库 这样的平台特定库。

你仍然可以使用 pub 工具获取更多库。库 collection、crypto、http、intl 和 test 仅仅是你可以使用 pub 安装的库的一个样本。

要了解更多关于 Dart 语言的内容，请参阅 语言教程。