Dart 库概览

本章将介绍以下库的主要功能及使用方式， 所有Dart平台中都包含这些库：

dart:core
内置类型，集合和其他核心功能。 该库会被自动导入到所有的 Dart 程序。

dart:async
支持异步编程，包括Future和Stream等类。

dart:math
数学常数和函数，以及随机数生成器。

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

本章只是一个概述； 只涵盖了几个 dart:* 库， 不包括第三方库。 特定平台库 dart:io 和 dart:html 的介绍，详见 dart:io tour 和 dart:html tour 。

更多库信息可以在 [pub.dartlang.org][pub.dartlang.org] 和 Dart web developer library guide. 查找。 所有 dart:* 库的 API 文档可以在 Dart API reference 查找， 如果使用的是 Flutter 可以在 [Flutter API reference.][docs.flutter.io] 查找。

> **DartPad tip：** 可以通过将该页中的代码拷贝到 DartPad 中进行演示。

dart:core - 数字，集合，字符串等

dart:core 库 (API reference) 提供了一个少量但是重要的内置功能集合。 该库会被自动导入每个 Dart 程序。

控制台打印

顶级 `print()` 方法接受一个参数 任意对象） 并输出显示这个对象的字符串值(由 `toString()` 返回) 到控制台。

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

有关基本字符串和 `toString()` 的更多信息，参考 Strings in the language tour.

数字

dart:core 库定义了 num ， int 以及 double 类， 这些类拥有一定的工具方法来处理数字。

使用 int 和 double 的 `parse()` 方法将字符串转换为整型或双浮点型对象：

```
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，该方法可能会创建一个整型，否则为浮点型对象：

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

通过添加 `radix` 参数，指定整数的进制基数：

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

使用 `toString()` 方法将整型或双精度浮点类型转换为字符串类型。 使用 toStringAsFixed(). 指定小数点右边的位数， 使用 toStringAsPrecision(): 指定字符串中的有效数字的位数。

```dart
// 整型转换为字符串类型。
assert(42.toString() == '42');

// 双浮点型转换为字符串类型。
assert(123.456.toString() == '123.456');

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

// 指定有效数字的位数。
assert(123.456.toStringAsPrecision(2) == '1.2e+2');
assert(double.parse('1.2e+2') == 120.0);
```

更多详情， 参考 int， double， 和 num 的相关 API 文档。 也可参考 dart:math section。

字符和正则表达式

在 Dart 中一个字符串是一个固定不变的 UTF-16 编码单元序列。 语言概览中有更多关于 strings 的内容。 使用正则表达式 (RegExp 对象) 可以在字符串内搜索和替换部分字符串。

String 定义了例如 `split()`， `contains()`， `startsWith()`， `endsWith()` 等方法。

在字符串中搜索

可以在字符串内查找特定字符串的位置， 以及检查字符串是否以特定字符串作为开头或结尾。例如：

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

// 一个字符串是否以另一个字符串为开头?
assert('Never odd or even'.startsWith('Never'));

// 一个字符串是否以另一个字符串为结尾?
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

可以获取字符串中的单个字符，将其作为字符串或者整数。 确切地说，实际上获取的是单独的UTF-16编码单元; 诸如高音谱号符号 (‘\u{1D11E}’) 之类的高编号字符分别为两个编码单元。

你也可以获取字符串中的子字符串或者将一个字符串分割为子字符串列表：

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

// 使用字符串模式分割字符串。
var parts = 'structured web apps'.split(' ');
assert(parts.length == 3);
assert(parts[0] == 'structured');

// 通过下标获取 UTF-16 编码单元（编码单元作为字符串）。
assert('Never odd or even'[0] == 'N');

// 使用 split() 传入一个空字符串参数，
// 得到一个所有字符的 list 集合；
// 有助于字符迭代。
for (var char in 'hello'.split('')) {
  print(char);
}

// 获取一个字符串的所有 UTF-16 编码单元。
var codeUnitList =
    'Never odd or even'.codeUnits.toList();
assert(codeUnitList[0] == 78);
```

首字母大小写转换

可以轻松的对字符串的首字母大小写进行转换：

```dart
// 转换为首字母大写。
assert('structured web apps'.toUpperCase() ==
    'STRUCTURED WEB APPS');

// 转换为首字母小写。
assert('STRUCTURED WEB APPS'.toLowerCase() ==
    'structured web apps');
```

> **提示：** 这些方法不是在所有语言上都有效的。例如， 土耳其字母表的无点 I 转换是不正确的。

Trimming 和空字符串

使用 `trim()` 移除首尾空格。 使用 `isEmpty` 检查一个字符串是否为空（长度为0）。

```dart
// Trim a string.
assert('  hello  '.trim() == 'hello');

// 检查字符串是否为空。
assert(''.isEmpty);

// 空格字符串不是空字符串。
assert('  '.isNotEmpty);
```

替换部分字符串

字符串是不可变的对象，也就是说字符串可以创建但是不能被修改。 如果仔细阅读了 String API docs, 你会注意到，没有一个方法实际的改变了字符串的状态。 例如，方法 `replaceAll()` 返回一个新字符串， 并没有改变原始字符串：

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

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

构建一个字符串

要以代码方式生成字符串，可以使用 StringBuffer 。在调用 `toString()` 之前， StringBuffer 不会生成新字符串对象。 `writeAll()` 的第二个参数为可选参数，用来指定分隔符， 本例中使用空格作为分隔符。

```
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类提供与JavaScript正则表达式相同的功能。 使用正则表达式可以对字符串进行高效搜索和模式匹配。

```
// 下面正则表达式用于匹配一个或多个数字。
var numbers = RegExp(r'\d+');

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

// contains() 能够使用正则表达式。
assert(!allCharacters.contains(numbers));
assert(someDigits.contains(numbers));

// 替换所有匹配对象为另一个字符串。
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 类提供对正则表达式匹配对象的访问。

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

// 检查正则表达式是否在字符串中匹配到对象。
assert(numbers.hasMatch(someDigits));

// 迭代所有匹配对象
for (var match in numbers.allMatches(someDigits)) {
  print(match.group(0)); // 15, then 20
}
```

更多信息

有关完整的方法列表， 请参考 String API docs。另请参考 StringBuffer、 Pattern、 RegExp、 和 Match 的 API 文档。

集合

Dart 附带了核心集合 API ， 其中包括 list ， set 和 map 类。

Lists

如语言概览中介绍，lists 可以通过字面量来创建和初始化。 另外，也可以使用 List 的构造函数。 List 类还定义了若干方法，用于向列表添加或删除项目。

```dart
// 使用 List 构造函数。
var vegetables = List();

// 或者仅使用一个 list 字面量。
var fruits = ['apples', 'oranges'];

// 添加一个元素到 list 对象。
fruits.add('kiwis');

// 添加多个元素到 list 对象。
fruits.addAll(['grapes', 'bananas']);

// 获取 list 长度。
assert(fruits.length == 5);

// 移除一个元素到 list 对象。
var appleIndex = fruits.indexOf('apples');
fruits.removeAt(appleIndex);
assert(fruits.length == 4);

// 移除多个元素到 list 对象。
fruits.clear();
assert(fruits.length == 0);
```

使用 `indexOf()` 方法查找一个对象在 list 中的下标值。

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

// 使用下标访问 list 中的元素
assert(fruits[0] == 'apples');

// 查找一个元素在 list 中的下标。
assert(fruits.indexOf('apples') == 0);
```

使用 `sort()` 方法排序一个 list 。你可以提供一个排序函数用于比较两个对象。 比较函数在 *小于* 时返回 \ <0， *相等* 时返回 0， *bigger* 时返回 > 0 。下面示例中使用 `compareTo()` 函数， 该函数在 Comparable 中定义， 并被 String 类实现。

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

// 排序一个 list 。
fruits.sort((a, b) => a.compareTo(b));
assert(fruits[0] == 'apples');
```

list 是参数化类型， 因此可以指定 list 应该包含的元素类型：

```dart
// 这个 list 只能包含字符串类型。
var fruits = List<String>();

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

// 产生静态分析警告，num 不是字符串类型。
fruits.add(5); // BAD: Throws exception in checked mode.
```

✗ static analysis: error/warning

```dart
fruits.add(5); // Error: 'int' can't be assigned to 'String'
```

全部的方法介绍， 请参考 List API docs 。

Sets

在 Dart 中，set 是一个无序的，元素唯一的集合。 因为一个 set 是无序的，所以无法通过下标（位置）获取 set 中的元素。

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

// 添加一个重复的元素是无效的。
ingredients.add('gold');
assert(ingredients.length == 3);

// 从 set 中移除一个元素。
ingredients.remove('gold');
assert(ingredients.length == 2);
```

使用 `contains()` 和 `containsAll()` 来检查一个或多个元素是否在 set 中。

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

// 检查一个元素是否在该 set 中。
assert(ingredients.contains('titanium'));

// 检查多个元素是否在该 set 中。
assert(ingredients.containsAll(['titanium', 'xenon']));
```

交集是另外两个 set 中的公共元素组成的 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'));
```

全部的方法介绍， 请参考 Set API docs 。

Maps

map 是一个无序的 key-value （键值对）集合，就是大家熟知的 *dictionary* 或者 *hash*。map 将 kay 与 value 关联，以便于检索。和 JavaScript 不同，Dart 对象不是 map 。

声明 map 可以使用简洁的字面量语法，也可以使用传统构造函数：

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

// map 可以通过构造函数构建。
var searchTerms = Map();

// map 是参数化类型；
// 可以指定一个 map 中 key 和 value 的类型。
var nobleGases = Map<int, String>();
```

通过大括号语法可以为 map 添加，获取，设置元素。 使用 `remove()` 方法从 map 中移除键值对。

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

// 使用 key 检索 value 。
assert(nobleGases[54] == 'xenon');

// 检查 map 是否包含 key 。
assert(nobleGases.containsKey(54));

// 移除一个 key 及其 value。
nobleGases.remove(54);
assert(!nobleGases.containsKey(54));
```

可以从一个 map 中检索出所有的 key 或所有的 value：

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

// 获取的所有的 key 是一个无序集合
// (可迭代 list 对象)。
var keys = hawaiianBeaches.keys;

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

// 获取的所有的 value 是一个无序集合
// (可迭代 list 对象).
var values = hawaiianBeaches.values;
assert(values.length == 3);
assert(values.any((v) => v.contains('Waikiki')));
```

使用 `containsKey()` 方法检查一个 map 中是否包含某个key 。因为 map 中的 value 可能会是 null， 所有通过 key 获取 value，并通过判断 value 是否为 null 来判断 key 是否存在是不可靠的。

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

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

如果当且仅当该 key 不存在于 map 中，且要为这个 key 赋值， 可使用`putIfAbsent ()` 方法。 该方法需要一个方法返回这个 value 。

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

全部的方法介绍， 请参考 Map API docs 。

公共集合方法

List, Set, 和 Map 共享许多集合中的常用功能。 其中一些常见功能由 Iterable 类定义， 这些函数由 List 和 Set 实现。

> **提示：** 虽然Map没有实现 Iterable， 但可以使用 Map `keys` 和 `values` 属性从中获取 Iterable 对象。

使用 `isEmpty` 和 `isNotEmpty` 方法可以检查 list， set 或 map 对象中是否包含元素：

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

使用 `forEach()` 可以让 list， set 或 map 对象中的每个元素都使用一个方法。

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

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

当在 map 对象上调用 `forEach() 方法时，函数必须带两个参数（key 和 value）：

```
hawaiianBeaches.forEach((k, v) {
  print('I want to visit $k and swim at $v');
  // 我想去瓦胡岛并且在
  // [Waikiki, Kailua, Waimanalo]游泳， 等等。
});
```

Iterable 提供 `map()` 方法，这个方法将所有结果返回到一个对象中。

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

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

> **提示：** `map()` 方法返回的对象是一个 *懒求值（lazily evaluated）*对象： 只有当访问对象里面的元素时，函数才会被调用。

使用 `map().toList()` 或 `map().toSet()`， 可以强制在每个项目上立即调用函数。

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

使用 Iterable 的 `where()` 方法可以获取所有匹配条件的元素。 使用 Iterable 的 `any()` 和 `every()` 方法可以检查部分或者所有元素是否匹配某个条件。

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

// 洋甘菊不含咖啡因。
bool isDecaffeinated(String teaName) =>
    teaName == 'chamomile';

// 使用 where() 来查找元素，
// 这些元素在给定的函数中返回 true 。
var decaffeinatedTeas =
    teas.where((tea) => isDecaffeinated(tea));
// 或者 teas.where(isDecaffeinated)

// 使用 any() 来检查集合中是否至少有一个元素满足条件。
assert(teas.any(isDecaffeinated));

// 使用 every() 来检查集合中是否所有元素满足条件。
assert(!teas.every(isDecaffeinated));
```

有关方法的完整列表，请参考 Iterable API docs, 以及 List, Set, and Map.

URI

在使用 URI（可能你会称它为 *URLs*） 时，Uri 类 提供对字符串的编解码操作。 这些函数用来处理 URI 特有的字符，例如 `&` 和 `=` 。Uri 类还可以解析和处理 URI—host， port， scheme等组件。

编码和解码完整合法的URI

使用 `encodeFull()` 和 `decodeFull()` 方法， 对 URI 中除了特殊字符（例如 `/`， `:`， `&`， `#`）以外的字符进行编解码， 这些方法非常适合编解码完整合法的 URI，并保留 URI 中的特殊字符。

```
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);
```

注意上面代码只编码了 `some` 和 `message` 之间的空格。

编码和解码 URI 组件

使用 `encodeComponent()` 和 `decodeComponent()` 方法， 对 URI 中具有特殊含义的所有字符串字符，特殊字符包括（但不限于）`/`， `&`, 和 `:`。

```
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);
```

注意上面代码编码了所有的字符。例如 `/` 被编码为 `%2F`。

解析 URI

使用 Uri 对象的字段（例如 `path`）， 来获取一个 Uri 对象或者 URI 字符串的一部分。 使用 `parse()` 静态方法，可以使用字符串创建 Uri 对象。

```
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');
```

有关 URI 组件的更多内容，参考 Uri API docs 。

构建 URI

使用 `Uri()` 构造函数，可以将各组件部分构建成 URI 。

```
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

DateTime 对象代表某个时刻，时区可以是 UTC 或者 本地时区。

DateTime 对象可以通过若干构造函数创建：

```
// 获取当前时刻。
var now = DateTime.now();

// 更具本地时区创建 DateTime 对象。
var y2k = DateTime(2000); // January 1, 2000

// 指定年月日。
y2k = DateTime(2000, 1, 2); // January 2, 2000

// 将日期指定为 UTC 时区。
y2k = DateTime.utc(2000); // 1/1/2000, UTC

// 指定自Unix纪元以来，以毫秒为单位的日期和时间。
y2k = DateTime.fromMillisecondsSinceEpoch(946684800000,
    isUtc: true);

// 解析ISO 8601日期。
y2k = DateTime.parse('2000-01-01T00:00:00Z');
```

日期中 `millisecondsSinceEpoch` 属性返回自 “Unix纪元（January 1, 1970, UTC）“以来的毫秒数：

```
// 1/1/2000, UTC
var y2k = DateTime.utc(2000);
assert(y2k.millisecondsSinceEpoch == 946684800000);

// 1/1/1970, UTC
var unixEpoch = DateTime.utc(1970);
assert(unixEpoch.millisecondsSinceEpoch == 0);
```

使用 Duration 类来计算两个日期的查，也可以对时刻进行前移和后移操作：

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

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

// 减少30天。
var december2000 =
    y2001.subtract(const Duration(days: 30));
assert(december2000.year == 2000);
assert(december2000.month == 12);

// 计算两个时刻之间的查，
// 返回 Duration 对象。
var duration = y2001.difference(y2k);
assert(duration.inDays == 366); // y2k was a leap year.
```

> **警告：** 由于时钟转换（例如，夏令时）的原因， 使用 Duration 对 DateTime 按天移动可能会有问题。 如果要按照天数来位移时间，请使用 UTC 日期。

参考 DateTime 和 Duration API 文档了解全部方法列表。

工具类

核心库包含各种工具类，可用于排序，映射值以及迭代。

比较对象

如果实现了 Comparable 接口， 也就是说可以将该对象与另一个对象进行比较， 通常用于排序。 `compareTo()` 方法在 *小于* 时返回 < 0，在 *相等* 时返回 0， 在 *大于* 时返回 > 0。

```
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);
}
```

Implementing map keys

在 Dart 中每个对象会默认提供一个整数的哈希值， 因此在 map 中可以作为 key 来使用， 重写 `hashCode` 的 getter 方法来生成自定义哈希值。 如果重写 `hashCode` 的 getter 方法，那么可能还需要重写 `==` 运算符。 相等的（通过 `==` ）对象必须拥有相同的哈希值。 哈希值并不要求是唯一的， 但是应该具有良好的分布形态。。

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

  Person(this.firstName, this.lastName);

  // 重写 hashCode，实现策略源于  Effective Java，
  // 第11章。
  @override
  int get hashCode {
    int result = 17;
    result = 37 * result + firstName.hashCode;
    result = 37 * result + lastName.hashCode;
    return result;
  }

  // 如果重写了 hashCode，通常应该从新实现 == 操作符。
  @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][] 和 Iterator 类支持 for-in 循环。 当创建一个类的时候，继承或者实现 Iterable，可以为该类提供用于 for-in 循环的 Iterators。实现 Iterator 来定义实际的遍历操作。

```dart
class Process {
  // Represents a process...
}

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

// A mythical class that lets you iterate through all
// processes. Extends a subclass of [Iterable].
class Processes extends IterableBase<Process> {
  @override
  final Iterator<Process> iterator = ProcessIterator();
}

void main() {
  // Iterable objects can be used with for-in.
  for (var process in Processes()) {
    // Do something with the process.
  }
}
```

异常

Dart 核心库定义了很多公共的异常和错误类。 异常通常是一些可以预见和预知的情况。 错误是无法预见或者预防的情况。

两个最常见的错误：

NoSuchMethodError
当方法的接受对象（可能为null）没有实现该方法时抛出。

ArgumentError
当方法在接受到一个不合法参数时抛出。

通常通过抛出一个应用特定的异常，来表示应用发生了错误。 通过实现 Exception 接口来自定义异常。

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

  const FooException([this.msg]);

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

更多内容，参考 Exceptions 以及 Exception API 文档。

dart:async - 异步编程

异步编程通常使用回调方法来实现，但是 Dart 提供了其他方案：Future 和 Stream 对象。 Future 类似与 JavaScript 中的 Promise ， 代表在将来某个时刻会返回一个结果。 Stream 类可以用来获取一系列的值，比如，一些列事件。 Future， Stream，以及更多内容，参考 dart:async library (API reference)。

> **提示：** 你并不总是需要直接使用 Future 或 Stream 的 API。Dart 语言支持使用关键字（例如，`async` 和 `await` ）来实现异步编程。更多详情，参考语言概览中 Asynchrony support。

dart:async 库可以工作在 web 应用及 command-line 应用。 通过 import dart:async 来使用。

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

> **版本提示：** 从 Dart 2.1 开始，使用 Future 和 Stream 不需要导入 dart:async， 因为 dart:core 库 export 了这些类。

Future

在 Dart 库中随处可见 Future 对象，通常异步函数返回的对象就是一个 Future。当一个 future *完成执行后*，future 中的值就已经可以使用了。

使用 await

在直接使用 Future API 前，首先应该考虑 `await` 来替代。 代码中使用 `await` 表达式会比直接使用 Future API 更容易理解。

阅读思考下面代码。 代码使用 Future 的 `then()` 方法在同一行执行了三个异步函数， 要等待上一个执行完成，再执行下一个任务之。

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

通过 await 表达式实现等价的代码， 看起来非常像同步代码：

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

async 函数能够捕获来自 Future 的异常。 例如：

```
var entryPoint = await findEntryPoint();
try {
  var exitCode = await runExecutable(entryPoint, args);
  await flushThenExit(exitCode);
} catch (e) {
  // Handle the error...
}
```

> **重要：** async 函数 返回 Future 对象。 如果你不希望你的函数返回一个 future 对象， 可以使用其他方案。 例如，你可以在你的方法中调用一个 async 方法。

更多关于 `await` 的使用及相关的 Dart 语言特征，参考 Asynchrony support。

基本用法

当 future 执行完成后，`then()` 中的代码会被执行。

`then()` 中的代码会在 future 完成后被执行。 例如， `HttpRequest.getString()` 返回一个 future 对象，因为 HTTP 请求可能需要一段时间。 当 Future 完成并且保证字符串值有效后，使用 `then()` 来执行你需要的代码：

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

使用 `catchError()` 来处理一些 Future 对象可能抛出的错误或者异常。

```dart
HttpRequest.getString(url).then((String result) {
  print(result);
}).catchError((e) {
  // Handle or ignore the error.
});
```

`then().catchError()` 组合是 `try-catch` 的异步版本。

> **重要：** 确保调用 `catchError()` 方式在 `then()` 的结果上，而不是在原来的 Future 对象上调用。 否则的话，`catchError()` 就只能处理原来 Future 对象抛出的异常， 而无法处理 `then()` 代码里面的异常。

链式异步编程

`then()` 方法返回一个 Future 对象， 这样就提供了一个非常好的方式让多个异步方法按顺序依次执行。 如果用 `then()` 注册的回调返回一个 Future ， 那么 `then()` 返回一个等价的 Future 。如果回调返回任何其他类型的值， 那么 `then()` 会创建一个以该值完成的新 Future 。

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

在上面的示例中，方法按下面顺序执行：

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

这是使用 await 编写的等效代码：

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

等待多个 Future

有时代码逻辑需要调用多个异步函数， 并等待它们全部完成后再继续执行。 使用 Future.wait() 静态方法管理多个 Future 以及等待它们完成：

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

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

Stream

在 Dart API 中 Stream 对象随处可见，Stream 用来表示一些列数据。 例如，HTML 中的按钮点击就是通过 stream 传递的。 同样也可以将文件作为数据流来读取。

异步循环

有时，可以使用异步 for 循环 `await for` ，来替代 Stream API 。

思考下面示例函数。 它使用 Stream 的 `listen()` 方法来订阅文件列表， 传入一个搜索文件或目录的函数。

```
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);
    }
  });
}
```

下面是使用 await 表达式和异步 for 循环 (`await for`) 实现的等价的代码， 看起来更像是同步代码：

```
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` 前，确认这样能保持代码清晰， 并希望获取所有 stream 的结果。 例如，你通常并 **不** 会使用 `await for` 来监听 DOM 事件， 因为 DOM 会发送无尽的流事件。 如果在同一行使用 `await for` 注册两个 DOM 事件， 那么第二个事件永远不会被处理。

有关 `await` 的使用及 Dart 语言的相关信息，参考 Asynchrony support。

监听流数据（stream data）

使用 `await for` 或者使用 `listen()` 方法监听 stream， 来获取每个到达的数据流值：

```
// 通过 ID 获取 button 并添加事件处理函数。
querySelector('#submitInfo').onClick.listen((e) {
  // 当 button 被点击是，该代码会执行。
  submitData();
});
```

下面示例中，ID 为 “submitInfo” button 提供的 `onClick` 属性是一个 Stream 对象。

如果只关心其中一个事件，可以使用，例如，`first`， `last`，或 `single` 属性来获取。 要在处理时间前对事件进行测试，可以使用，例如 `firstWhere()`， `lastWhere()`， 或 `singleWhere()` 方法。

如果只关心事件中的一个子集，可以使用，例如，`skip()`， `skipWhile()`，`take()`，`takeWhile()`， 和 `where()`。

传递流数据（stream data）

常常，在使用流数据前需要改变数据的格式。 使用 `transform()` 方法生成具有不同类型数据的流：

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

上面例子中使用了两个 transformer 。第一个使用 utf8.decoder 将整型流转换为字符串流。 接着，使用了 LineSplitter 将字符串流转换为多行字符串流。 这些 transformer 来自 dart:convert 库 （参考dart:convert section）。

处理错误和完成

处理错误和完成代码方式， 取决于使用的是 异步 for 循环（`await for`）还是 Stream API 。

如果使用的是异步 for 循环， 那么通过 try-catch 来处理错误。 代码位于异步 for 循环之后， 会在 stream 被关闭后执行。

```
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);
  }
}
```

如果使用的是 Stream API， 那么通过注册 `onError` 监听来处理错误。 代码位于注册的 `onDone` 中， 会在 stream 被关闭后执行。

```
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);
});
```

更多内容

更多在 command-line 应用中使用 Future 和 Stream 的实例，参考 dart:io tour. 也可以参考下列文章和教程：

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

dart:math - 数学和随机数

dart:math 库（API reference） 提供通用的功能，例如，正弦和余弦， 最大值和最小值，以及数学常数，例如 *pi* 和 *e*。大多数在 Math 库中的功能是作为顶级函数实现的。

通过 import dart:math 来引入使用该库。 下面的实例中使用 `math` 前缀，来说明顶级函数及常量源于 Math 库。

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

三角函数

Math 库提供基本的三角函数：

```
// Cosine
assert(cos(pi) == -1.0);

// Sine
var degrees = 30;
var radians = degrees * (pi / 180);
// radians is now 0.52359.
var sinOf30degrees = sin(radians);
// sin 30° = 0.5
assert((sinOf30degrees - 0.5).abs() < 0.01);
```

> **提示：** 这些函数参数单位是弧度，不是角度！

最大值和最小值

Math 库提供 `max()` 和 `min()` 方法：

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

数学常数

在 Math 库中可以找到你需要的数学常熟，例如，*pi*， *e* 等等：

```
// See the Math library for additional constants.
print(e); // 2.718281828459045
print(pi); // 3.141592653589793
print(sqrt2); // 1.4142135623730951
```

随机数

使用 Random 类产生随机数。 可以为 Random 构造函数提供一个可选的种子参数。

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

也可以产生随机布尔值序列：

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

更多内容

完整方法列表参考 Math API docs。在 API 文档中参考 num, int, 和 double。

dart:convert - 编解码JSON， UTF-8等

dart:convert 库 （API reference）提供 JSON 和 UTF-8 转换器， 以及创建其他转换器。 JSON 是一种用于表示结构化对象和集合的简单文本格式。 UTF-8 是一种常见的可变宽度编码，可以表示Unicode字符集中的每个字符。

dart:convert 库可以在 web 及 命令行应用中使用。 使用时，通过 import dart:convert 引入。

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

编解码JSON

使用 `jsonDecode()` 解码 JSON 编码的字符串为 Dart 对象：

```
// 提示：在 JSON 字符串中，必须使用双引号（"），
// 而不是单引号（'）。
// 下面是 JSON 字符串，非 Dart 字符串。
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 Dart 对象为 JSON 格式的字符串：

```
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 int， double， String， bool, null, List, 或者 Map 类型对象可以直接编码成 JSON。List 和 Map 对象进行递归编码。

不能直接编码的对象有两种方式对其编码。 第一种方式是调用 `encode()` 时赋值第二个参数， 这个参数是一个函数， 该函数返回一个能够直接编码的对象 第二种方式是省略第二个参数，着这种情况下编码器调用对象的 `toJson()` 方法。 更多示例及 JSON 包相关链接，参考 JSON Support 。

编解码 UTF-8 字符

使用 `utf8.decode()` 解码 UTF8 编码的字符创为 Dart 字符创：

```dart
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');
```

将 UTF-8 字符串流转换为 Dart 字符串，为 Stream 的 `transform()` 方法上指定 `utf8.decoder`：

```dart
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()` 将 Dart 字符串编码为一个 UTF8 编码的字节流：

```dart
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

dart:convert 库同样包含 ASCII 和 ISO-8859-1 (Latin1) 转换器。 更多详情，参考 API docs for the dart:convert library。

总结

本页向您介绍了 Dart 内置库中最常用的功能。 但是，并没有涵盖所有内置库。 您可能想要查看的其他内容包括 dart:collection 和 dart:typed_data, ， 以及特定于平台的库，如 Dart web development libraries 和 Flutter libraries. 。

您可以使用 pub tool 工具获得更多库。 collection, crypto, http, intl, 以及 test 以上只是简单的列举了一些可以通过 pub 安装的库。

要了解有关 Dart 语言的更多信息，请参考 language tour。