flutter 规范

#flutter 规范

##说明

使用背景
* 鉴于使用 flutter 框架开发 APP 开发统一化
* 开发需要统一的架构及规范
* 故在此进行架构及开发规范总结，后续继续补充

分支命名及使用规范
* 分支命名规范
    - 自己开发分支命名统一为 username ，如：zhangsan
    - 分支两条主线为 Master 分支和 feature 分支
    - feature 分支对应年月日（例如： feature/20220308）
    - Master 作为发布分支，feature 作为开发测试分支、自己开发分支从 feature checkout 出去，发布即 merge to master
* 分支合并规范
    - 从最新的 feature 分支 checkout 出自己的开发分支
    - 在自己开发开发分支开发完成后，先去 feature 分支 pull 最新代码，
    - 将 feature 分支最新代码 merge 到自己分支，确保无冲突
    - 再切回 feature 分支 merge 自己开发分支代码，确保无冲突，且能正常运行

commit 提交规范
* $git cz

* 用于说明 commit 的类别，只允许使用下面 7 个标识。

  - feat：新功能（feature）

  - fix：修补 bug

- docs：文档（documentation）

- style： 格式（不影响代码运行的变动）

- refactor：重构（即不是新增功能，也不是修改 bug 的代码变动）

- test：增加测试

- chore：构建过程或辅助工具的变动

代码规范

* 文件命名规范
    - 文件命名使用下划线命名法，如：hello_world
    - 请使用英文进行命名，不允许使用拼音。命名要求具有可读性，尽量避免使用缩写与数字
    - 未完待续

* 代码编码规范
    - 文件编码统一使用 UTF-8 编码；
    - 前端编码采用首字母小写驼峰法. Widget Class 必须采用首字母大写驼峰法.

文件目录结构(以 LIb 文件说明)

- lib
    - main.dart 入口文件
    - blocs 如果采用 BLoC 开发模式的话存放在此目录
    - config 基础配置目录
    - enums 枚举类存放目录
    - generated 数据模型解析中间层
    - models 存放模型，不应该加入逻辑层
    - net 网络基础层
    - observer 生命周期监测层
    - page 展示界面
    - plugs 本地 plugin 非远程加载
    - routers 路由处理
    - style 公共样式处理层

- utils 工具类
- viewmodel viewModel 层
- widgets 页面封装层

```javascript
├── main.dart //入口文件
├── blocs //采用 BLoC 或 provider 开发模式存放的目录
|   ├── bloc_provider.dart
|   └── xxx.dart
├── config //基础配置目录
|   └── xxx.dart
├── enums //枚举类存放目录
|   └── xxx.dart
├── generated //数据模型解析中间层
|   └── xxx.dart
├── models //本地存放模型，不应该加入逻辑层
|   └── xx.dart
├── net //网络基础层
|   └── xx.dart
├── observer //生命周期监测层
|   └── xx.dart
├── plugs //本地 plugin 非远程加载
|   └── xx.dart
├── style //公共样式处理层
|   └── xx.dart
├── routers //路由处理
|   └── routers.dart
├── utils //工具类
|   └── xxx.dart
├── viewmodel //viewModel 层
|   └── xxx.dart
├── page //app 展示界面
|   ├── home
|   |   ├── home.dart
|   |   └── xx.dart
|   └── xx.dart
└── widgets
```

```
    └── ... //下面详细说明
```

代码风格

标识符三种类型

大驼峰
类、枚举、typedef 和类型参数

```
  class SliderMenu { ... }

  class HttpRequest { ... }

  typedef Predicate = bool Function<T>(T value);
```

包括用于元数据注释的类

```
  class Foo {
    const Foo([arg]);
  }

  @Foo(anArg)
  class A { ... }

  @Foo()
  class B { ... }
```

使用小写加下划线来命名库和源文件

```
  library peg_parser.source_scanner;
```

```
  import 'file_system.dart';
  import 'slider_menu.dart';
```

不推荐如下写法：

```
  library pegparser.SourceScanner;

  import 'file-system.dart';
  import 'SliderMenu.dart';
```

使用小写加下划线来命名导入前缀

```
  import 'dart:math' as math;
  import 'package:angular_components/angular_components'
      as angular_components;
  import 'package:js/js.dart' as js;
```

不推荐如下写法：

```
  import 'dart:math' as Math;
  import 'package:angular_components/angular_components'
      as angularComponents;
  import 'package:js/js.dart' as JS;
```

使用小驼峰法命名其他标识符

```
  var item;
```

```
  HttpRequest httpRequest;

  void align(bool clearItems) {
    // ...
  }
```

优先使用小驼峰法作为常量命名

```
  const pi = 3.14;
  const defaultTimeout = 1000;
  final urlScheme = RegExp('^([a-z]+):');

  class Dice {
    static final numberGenerator = Random();
  }
```

不推荐如下写法：

```
  const PI = 3.14;
  const DefaultTimeout = 1000;
  final URL_SCHEME = RegExp('^([a-z]+):');

  class Dice {
    static final NUMBER_GENERATOR = Random();
  }
```

不使用前缀字母

因为 Dart 可以告诉您声明的类型、范围、可变性和其他属性，所以没有理由将这些属性编码为标识符名称。

```
```

```
  defaultTimeout
```

不推荐如下写法：

```
  kDefaultTimeout
```

排序

为了使你的文件前言保持整洁，我们有规定的命令，指示应该出现在其中。每个“部分”应该用空行分隔。

在其他引入之前引入所需的 dart 库

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

  import 'package:bar/bar.dart';
  import 'package:foo/foo.dart';
```

在相对引入之前先引入在包中的库

```
  import 'package:bar/bar.dart';
  import 'package:foo/foo.dart';

  import 'util.dart';
```

第三方包的导入先于其他包

```
  import 'package:bar/bar.dart';
```

```
  import 'package:foo/foo.dart';

  import 'package:my_package/util.dart';
```

在所有导入之后，在单独的部分中指定导出

```
  import 'src/error.dart';
  import 'src/foo_bar.dart';

  export 'src/error.dart';
```

不推荐如下写法：

```
  import 'src/error.dart';
  export 'src/error.dart';
  import 'src/foo_bar.dart';
```

所有流控制结构，请使用大括号

这样做可以避免悬浮的 else 问题

```
  if (isWeekDay) {
    print('Bike to work!');
  } else {
    print('Go dancing or read a book!');
  }
```

例外

一个 if 语句没有 else 子句，其中整个 if 语句和 then 主体都适合一行。在这种情况下，如果

你喜欢的话，你可以去掉大括号

```
  if (arg == null) return defaultValue;
```

如果流程体超出了一行需要分划请使用大括号：

```
  if (overflowChars != other.overflowChars) {
    return overflowChars < other.overflowChars;
  }
```

不推荐如下写法：

```
  if (overflowChars != other.overflowChars)
    return overflowChars < other.overflowChars;
```

注释

要像句子一样格式化

除非是区分大小写的标识符，否则第一个单词要大写。以句号结尾(或“!”或“?”)。对于所有的注释都是如此：doc 注释、内联内容，甚至 TODOs。即使是一个句子片段。

```
  greet(name) {
    // Assume we have a valid name.
    print('Hi, $name!');
  }
```

不推荐如下写法：

```
```

```
  greet(name) {
    /* Assume we have a valid name. */
    print('Hi, $name!');
  }
```

可以使用块注释(/…/)临时注释掉一段代码，但是所有其他注释都应该使用//

Doc 注释
使用///文档注释来记录成员和类型。

使用 doc 注释而不是常规注释，可以让 dartdoc 找到并生成文档。

```
  /// The number of characters in this chunk when unsplit.
  int get length => ...
```

> 由于历史原因，达特茅斯学院支持道格评论的两种语法:///(“C#风格”)和/**...*/(“JavaDoc 风格”)。我们更喜欢/// 因为它更紧凑。/** 和 */在多行文档注释中添加两个无内容的行。在某些情况下，///语法也更容易阅读，例如文档注释包含使用* 标记列表项的项目符号列表。

考虑为私有 api 编写文档注释

Doc 注释并不仅仅针对库的公共 API 的外部使用者。它们还有助于理解从库的其他部分调用的私有成员

用一句话总结开始 doc 注释

以简短的、以用户为中心的描述开始你的文档注释，以句号结尾。

```
/// Deletes the file at [path] from the file system.
void delete(String path) {
  ...
}
```

```

不推荐如下写法：

```
  /// Depending on the state of the file system and the user's permissions,
  /// certain operations may or may not be possible. If there is no file at
  /// [path] or it can't be accessed, this function throws either [IOError]
  /// or [PermissionError], respectively. Otherwise, this deletes the file.
  void delete(String path) {
    ...
  }
```

#### “doc 注释”的第一句话分隔成自己的段落

在第一个句子之后添加一个空行，把它分成自己的段落

```
  /// Deletes the file at [path].
  ///
  /// Throws an [IOError] if the file could not be found. Throws a
  /// [PermissionError] if the file is present but could not be deleted.
  void delete(String path) {
    ...
  }
```

使用参考

库的引用

导入 lib 下文件库，统一指定包名，避免过多的```../../```
```
package:flutter_packages/
```
```

字符串的使用

使用相邻字符串连接字符串文字

如果有两个字符串字面值(不是值，而是实际引用的字面值)，则不需要使用+连接它们。就像在 C 和 c++中，简单地把它们放在一起就能做到。这是创建一个长字符串很好的方法但是不适用于单独一行。

```
raiseAlarm(
    'ERROR: Parts of the spaceship are on fire. Other '
    'parts are overrun by martians. Unclear which are which.');
```

不推荐如下写法:

```
raiseAlarm('ERROR: Parts of the spaceship are on fire. Other ' +
    'parts are overrun by martians. Unclear which are which.');
```

优先使用模板字符串

```
'Hello, $name! You are ${year - birth} years old.';
```

在不需要的时候，避免使用花括号

```
  'Hi, $name!'
  "Wear your wildest $decade's outfit."
```

不推荐如下写法：

```
  'Hello, ' + name + '! You are ' + (year - birth).toString() + ' y...';
```

不推荐如下写法：

```
  'Hi, ${name}!'
  "Wear your wildest ${decade}'s outfit."
```

集合

尽可能使用集合字面量

如果要创建一个不可增长的列表，或者其他一些自定义集合类型，那么无论如何，都要使用构造函数。

```
  var points = [];
  var addresses = {};
  var lines = <Lines>[];
```

不推荐如下写法：

```
  var points = List();
  var addresses = Map();
```

不要使用.length 查看集合是否为空

```
if (lunchBox.isEmpty) return 'so hungry...';
if (words.isNotEmpty) return words.join(' ');
```

不推荐如下写法：

```
  if (lunchBox.length == 0) return 'so hungry...';
  if (!words.isEmpty) return words.join(' ');
```

考虑使用高阶方法转换序列

如果有一个集合，并且希望从中生成一个新的修改后的集合，那么使用.map()、.where()和Iterable 上的其他方便的方法通常更短，也更具有声明性

```
  var aquaticNames = animals
      .where((animal) => animal.isAquatic)
      .map((animal) => animal.name);
```

避免使用带有函数字面量的 Iterable.forEach()

在 Dart 中，如果你想遍历一个序列，惯用的方法是使用循环。

```
for (var person in people) {
  ...
}
```

不推荐如下写法：

```
  people.forEach((person) {
    ...
  });
```

不要使用 List.from()，除非打算更改结果的类型

给定一个迭代，有两种明显的方法可以生成包含相同元素的新列表

```
var copy1 = iterable.toList();
var copy2 = List.from(iterable);
```

明显的区别是第一个比较短。重要的区别是第一个保留了原始对象的类型参数

```
// Creates a List<int>:
var iterable = [1, 2, 3];

// Prints "List<int>":
print(iterable.toList().runtimeType);
```

```
// Creates a List<int>:
var iterable = [1, 2, 3];

// Prints "List<dynamic>":
print(List.from(iterable).runtimeType);
```

参数的使用

使用=将命名参数与其默认值分割开

由于遗留原因，Dart 均允许“:”和“=”作为指定参数的默认值分隔符。为了与可选的位置参数保持一致，使用“=”。

```
  void insert(Object item, {int at = 0}) { ... }
```

不推荐如下写法：

```
```

```
  void insert(Object item, {int at: 0}) { ... }
```

不要使用显式默认值 null

如果参数是可选的，但没有给它一个默认值，则语言隐式地使用 null 作为默认值，因此不需要编写它

```
void error([String message]) {
  stderr.write(message ?? '\n');
}
```

不推荐如下写法:

```
void error([String message = null]) {
  stderr.write(message ?? '\n');
}
```

变量

不要显式地将变量初始化为空
在 Dart 中，未显式初始化的变量或字段自动被初始化为 null。不要多余赋值 null

```
  int _nextId;

  class LazyId {
    int _id;

    int get id {
      if (_nextId == null) _nextId = 0;
      if (_id == null) _id = _nextId++;
```

```
      return _id;
    }
  }
```

不推荐如下写法：

```
  int _nextId = null;

  class LazyId {
    int _id = null;

    int get id {
      if (_nextId == null) _nextId = 0;
      if (_id == null) _id = _nextId++;

      return _id;
    }
  }
```

避免储存你能计算的东西

在设计类时，您通常希望将多个视图公开到相同的底层状态。通常你会看到在构造函数中计算所有视图的代码，然后存储它们:

应该避免的写法：

```
  class Circle {
    num radius;
    num area;
    num circumference;

    Circle(num radius)
        : radius = radius,
```

```
        area = pi * radius * radius,
        circumference = pi * 2.0 * radius;
  }
```

如上代码问题：

- 浪费内存
- 缓存的问题是无效——如何知道何时缓存过期需要重新计算？

推荐的写法如下：

```
  class Circle {
    num radius;

    Circle(this.radius);

    num get area => pi * radius * radius;
    num get circumference => pi * 2.0 * radius;
  }
```

类成员

不要把不必要地将字段包装在 getter 和 setter 中

不推荐如下写法：

```
  class Box {
    var _contents;
    get contents => _contents;
    set contents(value) {
      _contents = value;
    }
  }
```

```

#### 优先使用 final 字段来创建只读属性

尤其对于 ```StatelessWidget```

#### 在不需要的时候不要用 this

不推荐如下写法：

```
  class Box {
    var value;

    void clear() {
      this.update(null);
    }

    void update(value) {
      this.value = value;
    }
  }
```

推荐如下写法：

```
  class Box {
    var value;

    void clear() {
      update(null);
    }

    void update(value) {
      this.value = value;
    }
```
```

```
  }
```

构造函数

尽可能使用初始化的形式

不推荐如下写法：

```
  class Point {
    num x, y;
    Point(num x, num y) {
      this.x = x;
      this.y = y;
    }
  }
```

推荐如下写法：

```
class Point {
  num x, y;
  Point(this.x, this.y);
}
```

不要使用 new

Dart2 使 new 关键字可选

推荐写法：

```
  Widget build(BuildContext context) {
    return Row(
```

```
        children: [
          RaisedButton(
            child: Text('Increment'),
          ),
          Text('Click!'),
        ],
      );
  }
```

不推荐如下写法：

```
  Widget build(BuildContext context) {
    return new Row(
      children: [
        new RaisedButton(
          child: new Text('Increment'),
        ),
        new Text('Click!'),
      ],
    );
  }
```

异步

优先使用 async/await 代替原始的 futures

async/await 语法提高了可读性，允许你在异步代码中使用所有 Dart 控制流结构。

```
  Future<int> countActivePlayers(String teamName) async {
    try {
      var team = await downloadTeam(teamName);
      if (team == null) return 0;
```

```
      var players = await team.roster;
      return players.where((player) => player.isActive).length;
    } catch (e) {
      log.error(e);
      return 0;
    }
  }
```

当异步没有任何用处时，不要使用它

如果可以在不改变函数行为的情况下省略异步，那么就这样做。、

```
  Future afterTwoThings(Future first, Future second) {
    return Future.wait([first, second]);
  }
```

不推荐写法：

```
  Future afterTwoThings(Future first, Future second) async {
    return Future.wait([first, second]);
  }
```

空安全
如果一个对象不确定是否初始化尽量不要用 late 用?
例如
```
User? user;

//接口获取处理：
```

```
user = User.jsonFrom(json);

//调用
String? name = user?.name;
Text(user.name!);
```

对于 json 数据解析，尽量不要直接强转

错误示例
```
userFromJson( Map<String, dynamic> json){
    User user = User();
    user.list = []..addAll((map['list'] as List ?? []).map((o) => ListModel.fromMap(o)));
    user.name = json['name'];
}
```
正确用法
```
userFromJson( Map<String, dynamic> json){
    User user = User();
    user.list = json['list'] == null ? [] []..addAll((json['list'] as List).map((o) =>
ListModel.fromMap(o)));
    user.name = json['name'];
}
```

late 用法 在调用的时候才加载，不调用不加载
```
late User user = User();
```

late 尽量少用 如果没有确定初始化就调用的地方尽量别用
例如：

```
late ExampleModel model = ExampleModel();

ExampleModel? model;
```

网络请求 json 数据解析 尽量用 ？
例如
```
setMyName(name = json["name"]);

setMyName(String? name){
}
```
bool 类型的属性 在做判断的时间 尽量 做空校验
例如：
```
if(item.sysGroupLabelFlag ?? false) {}
```
var 尽量少用 如果用 var 下边一定要有类型判断
例如：
```
var list = json["list"];
if(!(list is List)) return;

```