Flutter2 空安全适配指南

AlanGe

一、走进空安全（空安全最小必备知识）

从 Flutter 2 开始，Flutter 便在配置中默认启用了空安全，通过将空检查合并到类型系统中，可以在开发过程中捕获这些错误，从而防止再生产环境导致的崩溃。

什么是空安全

时至今日，空安全已经是一个屡见不鲜的话题，目前像主流的编程语言 Kotlin、Swift、Rust 等都对空安全有自己的支持。Dart 从 2.12 版本开始支持了空安全，通过空安全开发人员可以有效避免 null 错误崩溃。空安全性可以说是 Dart 语言的重要补充，它通过区分可空类型和非可空类型进一步增强了类型系统。

引入空安全的好处

- 可以将原本运行时的空值引用错误将变为编辑时的分析错误；
- 增强程序的健壮性，有效避免由 Null 而导致的崩溃；

- 跟随Dart和Flutter的发展趋势，为程序的后续迭代不留坑；

空安全最小必备知识

- 空安全的原则
- 引入空安全前后Dart类型系统的变化
- 可空（?）类型的使用
- 延迟初始化（late）的使用
- 空值断言操作符（!）的使用

空安全的原则

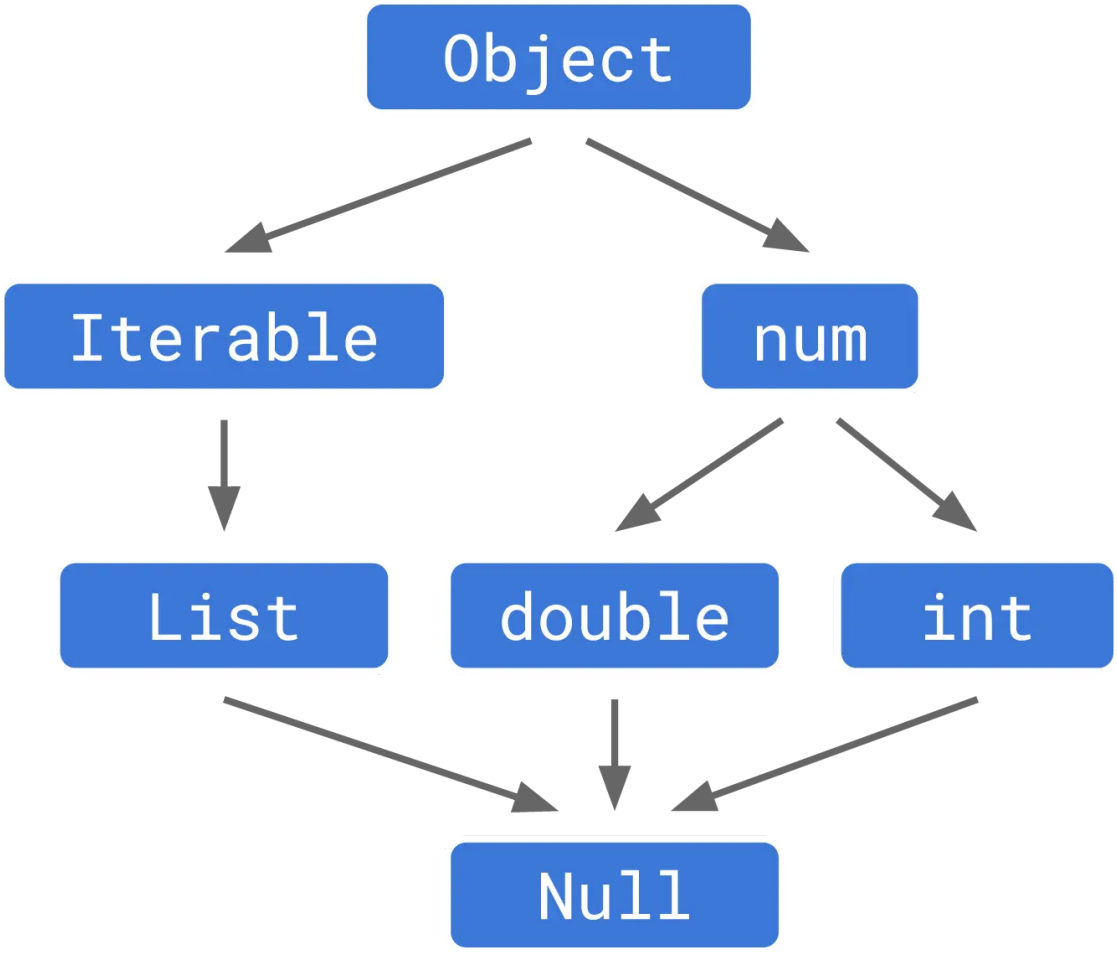
Dart 的空安全支持基于以下三条核心原则：

- 默认不可空：除非您将变量显式声明为可空，否则它一定是非空的类型；
- 渐进迁移：您可以自由地选择何时进行迁移，多少代码会进行迁移；
- 完全可靠：Dart 的空安全是非常可靠的，意味着编译期间包含了很多优化，
    - 如果类型系统推断出某个变量不为空，那么它 永远 不为空。当您将整个项目和其依赖完全迁移至空安全后，您会享有健全性带来的所有优势——更少的 BUG、更小的二进制文件以及更快的执行速

度。

引入空安全前后Dart类型系统的变化

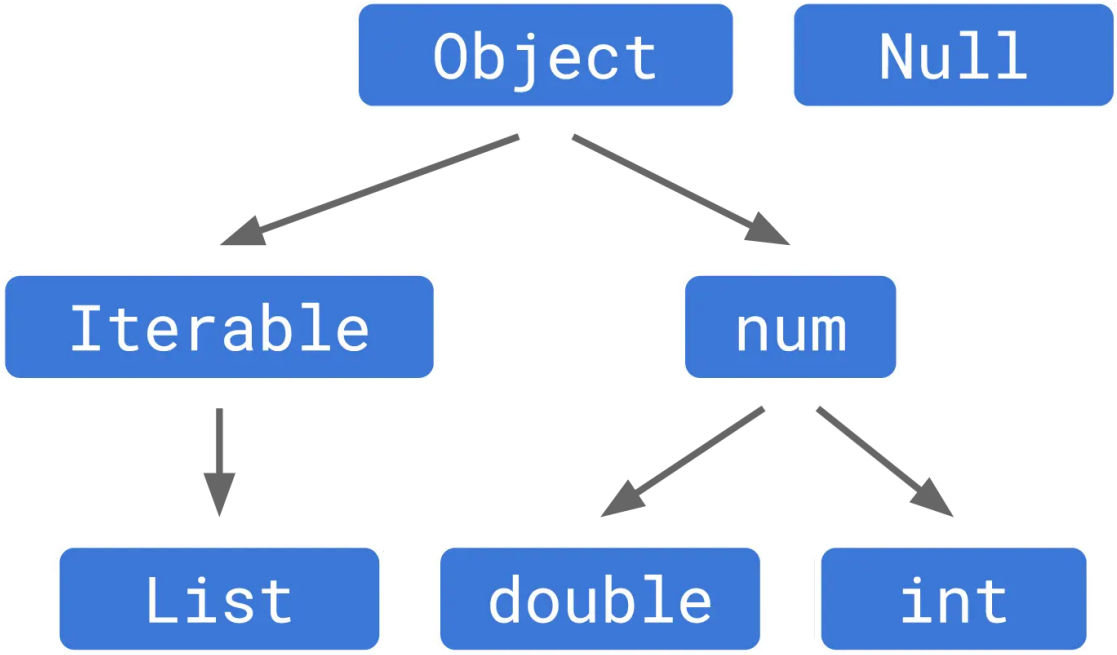
在引入空安全前Dart的类型系统是这样的：



image

这意味着在之前，所有的类型都可以为Null，也就是Nul类型被看作是所有类型的子类。

在引入空安全之后：



image

可以看出，最大的变化是将Null类型独立出来了，**这意味着Null不在是其它类型的子类型，所以对于一个非Null类型的变量传递一个Null值时会报类型转换错误。**

提示：在使用了空安全的Flutter或Dart项目中你会经常看到`?.`、`!`、`late`的大量应用，那么他们分别是什么又改如何使用呢？请看下文的分析

可空（?）类型的使用

我们可以通过将`?`跟在类型的后面来表示它后面的变量或参数可接受Null：

```
class CommonModel {
```

```
  String? firstName; //可空的成员变量
  int getNameLen(String? lastName /*可空的参数*/) {
    int firstLen = firstName?.length ?? 0;
    int lastLen = lastName?.length ?? 0;
    return firstLen + lastLen;
  }
}
```

对于可空的变量或参数在使用的时候需要通过 Dart 的避空运算符`?.`来进行访问，否则会抛出编译错误。

当程序启用空安全后，类的成员变量默认是不可空的，所以对于一个非空的成员变量需要指定其初始化方式：

```
class CommonModel {
  List names=[];//定义时初始化
  final List colors;//在构造方法中初始化
  late List urls;//延时初始化
  CommonModel(this.colors);
  ...
```

延迟初始化（late）的使用

对于无法在定义时进行初始化，并且又想避免使用`?.`，那么延迟初始化可以帮到你。通过`late`修饰的变量，可以让开发者选择初始化的时机，并且在使用这个变量时可以不用`?.`。

```
  late List urls;//延时初始化
  setUrls(List urls){
```

```
    this.urls=urls;
  }
  int getUrlLen(){
    return urls.length;
  }
```

延时初始化虽然能为我们编码带来一定便利，但如果使用不当会带来空异常的问题，**所以在使用的时候一定保证赋值和访问的顺序，切莫颠倒**。

延迟初始化（late）使用范式

在 Flutter 中 State 的 `initState` 方法中初始化的一些变量是比较适合使用 late 来进行延时初始化的，因为在 Widget 生命周期中 `initState` 方法是最先执行的，所以它里面初始化的变量通过 `late` 修饰后既能保障使用时的便利，又能防止空异常，下面就以 Flutter从入门到进阶-语音搜索模块 为例来看下具体的用法：

```
class _SpeakPageState extends State<SpeakPage>
    with SingleTickerProviderStateMixin {
  String speakTips = '长按说话';
  String speakResult = '';
  late Animation<double> animation;
  late AnimationController controller;

  @override
  void initState() {
    controller = AnimationController(
```

```
        super.initState();
        vsync: this, duration: Duration(milliseconds:
1000));
    animation = CurvedAnimation(parent: controller, curve:
Curves.easeIn)
      ..addStatusListener((status) {
        if (status == AnimationStatus.completed) {
          controller.reverse();
        } else if (status == AnimationStatus.dismissed) {
          controller.forward();
        }
      });
  }
  ...
```

空值断言操作符（!）的使用

当我们排除变量或参数的可空的可能后，可以通过!来告诉编译器这个可空的变量或参数不可空，这对我们进行方法传参或将可空参数传递给一个不可空的入参时特别有用：

```
  Widget get _listView {
    return ListView(
      children: <Widget>[
        _banner,
        Padding(
          padding: EdgeInsets.fromLTRB(7, 4, 7, 4),
          child: LocalNav(localNavList: localNavList),
        ),
        if (gridNavModel != null)
          Padding(
```

```
            padding: EdgeInsets.fromLTRB(7, 0, 7, 4),
            child: GridNav(gridNavModel: gridNavModel!)),
        Padding(
            padding: EdgeInsets.fromLTRB(7, 0, 7, 4),
            child: SubNav(subNavList: subNavList)),
        if (salesBoxModel != null)
          Padding(
            padding: EdgeInsets.fromLTRB(7, 0, 7, 4),
            child: SalesBox(salesBox: salesBoxModel!)),
      ],
    );
  }
```

上述代码是Flutter从入门到进阶-首页模块根据gridNavModel与salesBoxModel模块数据是否为空时动态创建的列表，在确保变量不为空的情况下使用了空值断言操作符`!`。

除此之外，`!`还有一个常见的用处：

```
bool isEmptyList(Object object) {
  if (object is! List) return false;
  return object.isEmpty;
}
```

用在这里表示取反，上述代码等价于：

```
bool isEmptyList(Object object) {
```

```
  if (!(object is List)) return false;
  return object.isEmpty;
}
```

二、Flutter如何做空安全适配

Step 1: 启用空安全

Flutter 2默认启用了空安全，所以通过Flutter 2创建的项目是已经开启了空安全的检查的，另外，小伙伴也可以可以通过下面命令来查看你的Flutter SDK版本：

```
flutter doctor
```

那么，**如何手动开启和关闭空区安全的?**

```
environment:
  sdk: ">=2.12.0 <3.0.0" //sdk >=2.12.0表示开启空安全检查
```

提示：一旦项目开启了空安全检查，那么你的代码包括项目所依赖的三方插件必须是要支持空安全的否则是无法正常编译的。

如果想关闭空安全检查，可以将SDK的支持范围调整到

`2.12.0`以下即可，如：

```
environment:
  sdk: ">=2.7.0 <3.0.0"
```

Step 2: 进行空安全适配

开启空安全之后，然后运行下项目你会看到很多的报错，然后定位到报错的文件，通过本章所讲技能进行适配。首先需要对文件进行分类：

- 自定义 Widget（包含你所创建的 Flutter 页面）如：
  - Flutter 高级进阶实战 仿哔哩哔哩 APP 中的：登录、注册、首页、收藏、排行、详情等页面，以及`video_card.dart`、`login_effect.dart`、`hi_tab.dart`、`hi_banner.dart`等自定义 widget。
  - Flutter 从入门到进阶 实战携程网 App 中的：搜索、旅拍、我的、首页等页面，以及`grid_nav.dart`、`loading_container.dart`、`local_nav.dart`、`sales_box.dart`、`search_bar.dart`、`sub_nav.dart`等自定义 widget。
- 数据模型（Model）如：
  - Flutter 高级进阶实战 仿哔哩哔哩 APP 中的：`home_mo.dart`、`notice_mo.dart`、`profile_mo.dart`、

`ranking_mo.dart`、`video_detail_mo.dart`、`video_model.dart`等model。

- Flutter从入门到进阶 实战携程网App中的：`common_model.dart`、`config_model.dart`、`grid_nav_model.dart`、`home_model.dart`、`sales_box_model.dart`、`seach_model.dart`、`travel_model.dart`、`travel_tab_model.dart`等model。

- 单例如：
  - Flutter高级进阶实战 仿哔哩哔哩APP中的：`hi_navigator.dart`、`hi_cache.dart`、`hi_net.dart`等。

然后结合着后面对应小节的教程进行适配即可。

三、自定义Widget的空安全适配技巧

自定义Widget的空安全适配分两种情况：

- Widget的空安全适配
- State的空安全适配

Widget的空安全适配

对于自定的Widget无论是页面的某控件还是整个页面，通常都会为Widget定义一些属性。在进行空安全适配时要对属性进行一下分类：

- 可空的属性：通过`?`进行修饰
- 不可空的属性：在构造函数中设置默认值或者通过`required`进行修饰

```
class WebView extends StatefulWidget {
  String? url;
  final String? statusBarColor;
  final String? title;
  final bool? hideAppBar;
  final bool backForbid;

  WebView(
      {this.url,
      this.statusBarColor,
      this.title,
      this.hideAppBar,
      this.backForbid = false})
      ...
```

提示：如果构造方法中使用了`@required`那么需要改成`required`。

State的空安全适配

State的空安全适配主要是根据它的成员变量是否可空进行分

类：

- 可空的变量：通过`?`进行修饰
- 不可空的变量：可采用以下两种方式进行适配
  - 定义时初始化
  - 使用`late`修饰为延时变量

四、数据模型（Model）空安全适配技巧

数据模型（Model）空安全适配主要以下两种情况：

- 含有命令构造函数的模型
- 含有命名工厂构造函数的模型

含有命令构造函数的模型

含有命令构造函数的模型的空安全适配技巧：

适配前：

```
///旅拍页模型
class TravelItemModel {
  int totalCount;
  List<TravelItem> resultList;
  TravelItemModel.fromJson(Map<String, dynamic> json) {
    totalCount = json['totalCount'];
```

```
    if (json['resultList'] != null) {
      resultList = new List<TravelItem>();
      json['resultList'].forEach((v) {
        resultList.add(new TravelItem.fromJson(v));
      });
    }
  }
  Map<String, dynamic> toJson() {
    final Map<String, dynamic> data = new Map<String,
dynamic>();
    data['totalCount'] = this.totalCount;
    if (this.resultList != null) {
      data['resultList'] = this.resultList.map((v) =>
v.toJson()).toList();
    }
    return data;
  }
}
```

适配之前首先要和服务端协商好，模型中那些字段可空，那些字段是一定会下发的。对于这个案例假如：totalCount字段是一定会下发的，resultList字段是不能保证一定会下发，那么我们可以这样来适配：

适配后：

```
///旅拍页模型
class TravelItemModel {
  late int totalCount;
  List<TravelItem>? resultList;

  //命名构造方法
  TravelItemModel.fromJson(Map<String, dynamic> json) {
```

```
    totalCount = json['totalCount'];
    if (json['resultList'] != null) {
      resultList = new List<TravelItem>.empty(growable:
true);
      json['resultList'].forEach((v) {
        resultList!.add(new TravelItem.fromJson(v));
      });
    }
  }

  Map<String, dynamic> toJson() {
    final Map<String, dynamic> data = new Map<String,
dynamic>();
    data['totalCount'] = this.totalCount;
    data['resultList'] = this.resultList!.map((v) =>
v.toJson()).toList();
    return data;
  }
}
```

- 对于一定会下发的字段我们通过`late`来修饰为延迟初始化的字段以方便访问
- 对于不能保证一定会下发的字段，我们通过`?`将其修饰为可空的变量

含有命名工厂构造函数的模型

命名工厂构造函的数据模型也是比较常见的数据模型之一，公共数据模型为例来分享含有命名工厂构造函的数据模型的空安全适配技巧：

适配前：

```
class CommonModel {
  final String icon;
  final String title;
  final String url;
  final String statusBarColor;
  final bool hideAppBar;

  CommonModel(
      {this.icon, this.title, this.url,
this.statusBarColor, this.hideAppBar});

  factory CommonModel.fromJson(Map<String, dynamic> json) {
    return CommonModel(
      icon: json['icon'],
      title: json['title'],
      url: json['url'],
      statusBarColor: json['statusBarColor'],
      hideAppBar: json['hideAppBar']
    );
  }
}
```

含有命名工厂构造函数的模型通常需要有自己的构造函数，构造函数通常采用可选参数，所以在进行适配时首先要明确哪些字段一定不为空，哪些字段可空，确认好之后就可以进行下面适配了：

适配后：

```
class CommonModel {
```

```
  final String? icon;
  final String? title;
  final String url;
  final String? statusBarColor;
  final bool? hideAppBar;

  CommonModel(
      {this.icon,
      this.title,
      required this.url,
      this.statusBarColor,
      this.hideAppBar});
  //命名工厂构造函数必须要有返回值，类似static 函数无法访问成员变量和
方法
  factory CommonModel.fromJson(Map<String, dynamic> json) {
    return CommonModel(
      icon: json['icon'],
      title: json['title'],
      url: json['url'],
      statusBarColor: json['statusBarColor'],
      hideAppBar: json['hideAppBar']
    );
  }
}
```

- 对于可空的字段通过?进行修饰
- 对于不可空的字段，需要在构造方法中在对应的字段前面添加 `required` 修饰符来表示这个参数是必传参数

五、单例空安全适配技巧

单例是Flutter开发中使用最广的一种设计模式，那么单例该

如何适配空安全呢?

接下来就以[Flutter高级进阶实战中缓存模块]单例的空安全适配技巧

适配前:

```
///缓存管理类
class HiCache {
  SharedPreferences prefs;
  static HiCache _instance;
  HiCache._() {
    init();
  }
  HiCache._pre(SharedPreferences prefs) {
    this.prefs = prefs;
  }
  static Future<HiCache> preInit() async {
    if (_instance == null) {
      var prefs = await SharedPreferences.getInstance();
      _instance = HiCache._pre(prefs);
    }
    return _instance;
  }

  static HiCache getInstance() {
    if (_instance == null) {
      _instance = HiCache._();
    }
    return _instance;
  }
```

```
  void init() async {
    if (prefs == null) {
      prefs = await SharedPreferences.getInstance();
    }
  }
  setString(String key, String value) {
    prefs.setString(key, value);
  }

  setDouble(String key, double value) {
    prefs.setDouble(key, value);
  }

  setInt(String key, int value) {
    prefs.setInt(key, value);
  }

  setBool(String key, bool value) {
    prefs.setBool(key, value);
  }

  setStringList(String key, List<String> value) {
    prefs.setStringList(key, value);
  }

  T get<T>(String key) {
    return prefs?.get(key) ?? null;
  }
}
```

适配后：

```
class HiCache {
  SharedPreferences? prefs;
  static HiCache? _instance;
```

```
HiCache._() {
  init();
}
HiCache._pre(SharedPreferences prefs) {
  this.prefs = prefs;
}
static Future<HiCache> preInit() async {
  if (_instance == null) {
    var prefs = await SharedPreferences.getInstance();
    _instance = HiCache._pre(prefs);
  }
  return _instance!;
}

static HiCache getInstance() {
  if (_instance == null) {
    _instance = HiCache._();
  }
  return _instance!;
}

void init() async {
  if (prefs == null) {
    prefs = await SharedPreferences.getInstance();
  }
}

setString(String key, String value) {
  prefs?.setString(key, value);
}

setDouble(String key, double value) {
  prefs?.setDouble(key, value);
}
```

```
  setInt(String key, int value) {
    prefs?.setInt(key, value);
  }

  setBool(String key, bool value) {
    prefs?.setBool(key, value);
  }

  setStringList(String key, List<String> value) {
    prefs?.setStringList(key, value);
  }

  remove(String key) {
    prefs?.remove(key);
  }

  T? get<T>(String key) {
    var result = prefs?.get(key);
    if (result != null) {
      return result as T;
    }
    return null;
  }
}
```

核心适配的地方主要有两点：

- 因为是懒汉模式的单例，所以单例 instance 设置成可空
- getInstance 中因为会有 null 时创建单例，所以返回 instance 时将其转换成非空

六、插件的空安全适配问题

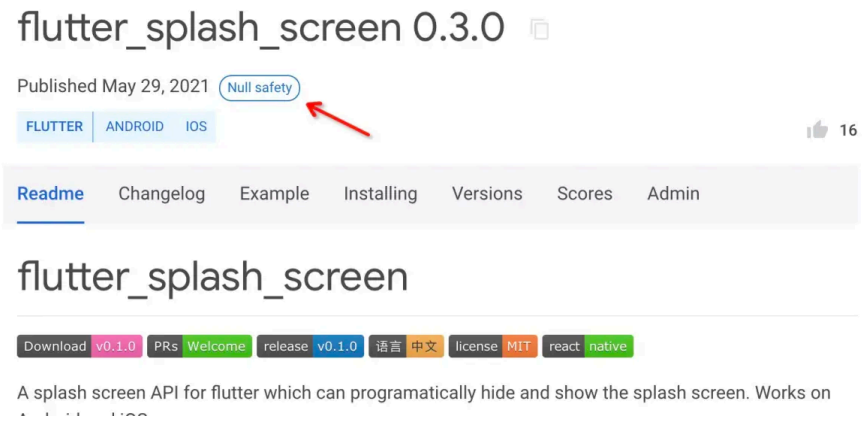
三方插件的空安全适配问题

目前在Dart的官方插件平台上的主流插件都陆续进行了空安全支持，如果你的项目开启了空安全那么所有使用的插件也必须是要支持空安全的，否则会导致无法编译：

```
Xcode's output:
↳
    Error: Cannot run with sound null safety, because the
following dependencies
    don't support null safety:

     - package:flutter_splash_screen
```

遇到这个问题后可以到Dart的官方插件平台查看这个flutter_splash_screen插件是否有支持了空安全的版本。如果插件支持了空安全插件平台会为其打上空安全的标：



image

如果你所使用的某个插件还不支持空安全，而且你又必须要使用这个插件，那么可以通过上文所讲的方式来关闭空安全检查。

我的插件该如何适配空安全?

通过Flutter进阶拓展：开发包和插件开发的学习，有不少小伙伴已经开发并发布了一些插件，那么该如何为你所开发的插件适配空安全呢?

回顾我对一些插件的适配的整个过程来讲，可以分为三个关键步骤：

- 开启空安全
- 代码适配：进行编译，对编译的报错进行空安全适配
- 发布：将适配后的代码发布到插件市场