本章目录

让App支持多语言

如果我们的应用要支持多种语言，那么我们需要“国际化”它。这意味着我们在开发时需要为应用程序支持的每种语言环境设置“本地化”的一些值，如文本和布局。Flutter提供一些widget和类以帮助实现国际化，而Flutter的库本身也是国际化的。

接下来我们以MaterialApp类为入口的应用来说明如何支持国际化。

> 大多数应用程序都是通过MaterialApp为入口，但根据低级别的WidgetsApp类为入口编写的应用程序也可以使用相同的类和逻辑进行国际化。MaterialApp实际上也是WidgetsApp的一个包装。

支持国际化

默认情况下，Flutter仅提供美国英语本地化。要添加对其他语言的支持，应用程序必须指定其他MaterialApp属性，并包含一个名为“flutter_localizations”的包。 截至2018年5月，该包支持24种语言。要使用flutter_localizations包，首先需要添加依赖到`pubspec.yaml`文件中：

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
```

接下来，下载flutter_localizations库，然后指定MaterialApp的`localizationsDelegates`和`supportedLocales`：

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

new MaterialApp(
 localizationsDelegates: [
   // 本地化的代理类
   GlobalMaterialLocalizations.delegate,
   GlobalWidgetsLocalizations.delegate,
 ],
 supportedLocales: [
    const Locale('en', 'US'), // 美国英语
    const Locale('zh', 'CN'), // 中文简体
    //其它Locales
  ],
  // ...
)
```

基于WidgetsApp的应用程序类似，只是不需要`GlobalMaterialLocalizations.delegate`。

`localizationsDelegates`列表中的元素是生成本地化值集合的工厂。`GlobalMaterialLocalizations.delegate`为Material 组件库提供的本地化的字符串和其他值，它可以使Material Widget支持多语言。
`GlobalWidgetsLocalizations.delegate`定义widget默认的文本方向，从左到右或从右到左，这是因为有些语言的阅读习惯并不是从左到右，比如如阿拉伯语就是从右向左的。

获取当前区域Locale

`Locale`类是用来标识用户的语言环境的，它包括语言和国家两个标志如：

```
const Locale('zh', 'CN') // 中文简体
```

我们始终可以通过以下方式来获取应用的当前区域Locale：

```
Locale myLocale = Localizations.localeOf(context);
```

`Localizations` Widget一般位于Widget树中其它业务组件的顶部，它的作用是定义区域Locale以及设置子树依赖的本地化资源。 如果系统的语言环境发生变化，WidgetsApp将创建一个新的Localizations Widget并重建它，这样子树中通过 `Localizations.localeOf(context)` 获取的Locale就会更新。

监听系统语言切换

当我们更改系统语言设置时，APP中的Localizations widget会重新构建，Localizations.localeOf(context)` 获取的Locale就会更新，最终界面会重新build达到切换语言的效果。但是这个过程是隐式完成的，我们并没有主动去监听系统语言切换，但是有时我们需要在系统语言发生改变时做一些事，这时我们就需要监听locale改变事件。

我们可以通过localeResolutionCallback或localeListResolutionCallback回调来监听locale改变的事件，我们先看看localeResolutionCallback的回调函数签名：

```
Locale Function(Locale locale, Iterable<Locale> supportedLocales)
```

- 参数`locale`的值为当前的当前的系统语言设置，当应用启动时或用户动态改变系统语言设置时此locale即为系统的当前locale。当开发者手动指定APP的locale时，那么此locale参数代表开发者指定的locale，此时将忽略系统locale如：

  ```
  MaterialApp(
   ...
   locale: const Locale('en', 'US'), //手动指定locale
   ...
  )
  ```

  上面的例子中手动指定了应用locale为美国英语，指定后即使设备当前语言是中文简体，应用中的locale也依然是美国英语。

  如果`locale`为null，则表示Flutter未能获取到设备的Locale信息，所以我们在使用`locale`之前一定要先判空。

- `supportedLocales`为当前应用支持的locale列表，是开发者在MaterialApp中通过`supportedLocales`属性注册的。

- 返回值是一个Locale，此Locale为Flutter APP最终的Locale。通常在不支持的语言区域时返回一个默认的Locale。

localeListResolutionCallback和localeResolutionCallback唯一的不同就在第一个参数类型，前者接收的是一个Locale列表，而后者接收的是单个Locale。

```
Locale Function(List<Locale> locales, Iterable<Locale>
supportedLocales)
```

在新版的Android系统中，用户可以设置一个语言列表，这样一来，支持多语言的应用就会得到这个列表，应用通常的处理方式就是按照列表的顺序依次尝试加载相应的Locale，如果某一种语言加载成功则会停止。下面是Android中设置语言列表的截图：

在Flutter中，应该优先使用localeListResolutionCallback，当然你不必担心Android系统的差异性，如果在低版本的Android系统中，Flutter会自动处理这种情况，这时Locale列表只会包含一项。

Localization widget

Localizations widget用于加载和查找包含本地化值的对象。应用程序通过 `Localizations.of(context,type)` 来引用这些对象。 如果设备的Locale区域设置发生更改，则Localizations widget会自动加载新区域的Locale值，然后重新构建使用了它们的widget。 发生这种情况是因为Localizations内部使用了InheritedWidget ，当build函数引用了InheritedWidget时，会创建对InheritedWidget的隐式依赖关系。当InheritedWidget发生更改时，即Localizations widget的Locale设置发生更改时，将重建其依赖的上下文。

本地化值由Localizations的 LocalizationsDelegates 列表加载 。 **每个委托必须定义一个异步load() 方法**，以生成封装了一系列本地化值的对象。通常这些对象为每个本地化值定义一个方法。

在大型应用程序中，不同模块或Package可能会与自己的本地化值捆绑在一起。 这就是为什么要用Localizations 管理对象表的原因。 要检索由LocalizationsDelegate `load` 方法之一产生的对象，可以指定一个BuildContext和对象的类型。例如，Material 组件库的本地化字符串由MaterialLocalizations类定义， 此类的实例由MaterialApp类提供的LocalizationDelegate创建， 它们可以通过 `Localizations.of` 被获取到：

```
Localizations.of<MaterialLocalizations>(context,
MaterialLocalizations);
```

这个特殊的 `Localizations.of()` 表达式会经常使用，所以MaterialLocalizations类提供了一个便捷方法：

```
static MaterialLocalizations of(BuildContext context) {
  return Localizations.of<MaterialLocalizations>(context,
MaterialLocalizations);
}

// 可以直接调用便捷方法
tooltip: MaterialLocalizations.of(context).backButtonTooltip,
```

使用打包好的LocalizationsDelegates

为了尽可能小而且简单，flutter软件包中仅提供美国英语值的MaterialLocalizations和WidgetsLocalizations接口的实现。 这些实现类分别称为DefaultMaterialLocalizations和DefaultWidgetsLocalizations。flutter_localizations Package包含GlobalMaterialLocalizations和GlobalWidgetsLocalizations的本地化接口的多语言实现， 国际化的应用程序必须按照本节开头说明的那样为这些类指定本地化Delegate。

上述的GlobalMaterialLocalizations和GlobalWidgetsLocalizations只是Material组件库的本地化实现，如果我们要让自己的布局支持多语言，那么就需要实现在即的Localizations，我们将在下一节介绍其具体的实现方式。

实现Localizations

前面讲了Material组件库如何支持国际化，本节我们将介绍一下我们自己的UI中如何支持多语言。根据上节所述，我们需要实现两个类：一个Delegate类一个Localizations类，下面我们通过一个实例说明。

实现Localizations类

我们已经知道Localizations类中主要实现提供了本地化值，如文本：

```
//Locale资源类
class DemoLocalizations {
  DemoLocalizations(this.isZh);
  //是否为中文
  bool isZh = false;
  //为了使用方便，我们定义一个静态方法
  static DemoLocalizations of(BuildContext context) {
    return Localizations.of<DemoLocalizations>(context,
DemoLocalizations);
  }
  //Locale相关值，title为应用标题
  String get title {
    return isZh ? "Flutter应用" : "Flutter APP";
  }
  //... 其它的值
}
```

DemoLocalizations中会根据当前的语言来返回不同的文本，如`title`，我们可以将所有需要支持多语言的文本都在此类中定义。DemoLocalizations的实例将会在Delegate类的`load`方法中创建。

实现Delegate类

Delegate类的职责是在Locale改变时加载新的Locale资源，所以它有一个`load`方法，Delegate类需要继承自LocalizationsDelegate类，实现相应的接口，示例如下：

```
//Locale代理类
class DemoLocalizationsDelegate extends
LocalizationsDelegate<DemoLocalizations> {
  const DemoLocalizationsDelegate();

  //是否支持某个Local
  @override
  bool isSupported(Locale locale) => ['en',
'zh'].contains(locale.languageCode);

  // Flutter会调用此类加载相应的Locale资源类
  @override
  Future<DemoLocalizations> load(Locale locale) {
    print("xxxx$locale");
    return SynchronousFuture<DemoLocalizations>(
        DemoLocalizations(locale.languageCode == "zh")
    );
  }

  @override
  bool shouldReload(DemoLocalizationsDelegate old) => false;
}
```

`shouldReload`的返回值决定当Localizations Widget重新build时，是否调用`load`方法重新加载Locale资源。一般情况下，Locale资源只应该在Locale切换时加载一次，不需要每次在Localizations 重新build时都加载，所以返回`false`即可。可能有些人会担心返回`false`的话在APP启动后用户再改变系统语言时`load`方法将不会被调用，所以Locale资源将不会被加载。事实上，每当Locale改变时Flutter都会再调用`load`方法加载新的Locale，无论`shouldReload`返回`true`还是`false`。

最后一步：添加多语言支持

和上一节中介绍的相同，我们现在需要先注册DemoLocalizationsDelegate类，然后再通过`DemoLocalizations.of(context)`来动态获取当前Locale文本。

只需要在MaterialApp或WidgetsApp的`localizationsDelegates`列表中添加我们的Delegate实例即可完成注册：

```
localizationsDelegates: [
 // 本地化的代理类
 GlobalMaterialLocalizations.delegate,
 GlobalWidgetsLocalizations.delegate,
 // 注册我们的Delegate
 DemoLocalizationsDelegate()
],
```

接下来我们可以在Widget中使用Locale值：

```
return Scaffold(
  appBar: AppBar(
    //使用Locale title
    title: Text(DemoLocalizations.of(context).title),
  ),
  ... //省略无关代码
 )
```

这样，当在美国英语和中文简体之间切换系统语言时，APP的标题将会分别为“Flutter APP”和“Flutter应用”。

总结

本节我们通过一个简单的示例说明了Flutter应用国际化的基本过程及原理。但是上面的实例还有一个严重的不足就是我们需要在DemoLocalizations类中获取`title`时手动的判断当前语言Locale，然后返回合适的文本。试想一下，当我们要支持的语言不是两种而是8种甚至20几种时，如果为每个文本属性都要分别去判断到底是哪种Locale从而获取相应语言的文本将会是一件非常复杂的事。还有，通常情况下翻译人员并不是开发人员，能不能像i18n或i10n标准那样可以将翻译单独保存为一个arb文件交由翻译人员去翻译，翻译好之后开发人员再通过工具将arb文件转为代码。答案是肯定的！我们将在下一节介绍如何通过Dart intl包来实现这些。

使用Intl包

使用Intl包我们不仅可以非常轻松的实现国际化，而且也可以将字符串文本分离成单独的文件，方便开发人员和翻译人员分工协作。为了使用Intl包我们需要添加两个依赖：

```
dependencies:
  #...省略无关项
  intl: ^0.15.7
dev_dependencies:
   #...省略无关项
  intl_translation: ^0.17.2
```

intl_translation 包主要包含了一些工具，它在开发阶段主要主要的作用是从代码中提取要国际化的字符串到单独的arb文件和根据arb文件生成对应语言的dart代码，而intl包主要是引用和加载intl_translation生成后的dart代码。下面我们将一步步来说明如何使用：

第一步：创建必要目录

首先，在项目根目录下创建一个i10n-arb目录，该目录保存我们接下来通过intl_translation命令生成的arb文件。一个简单的arb文件内容如下：

```
{
  "@@last_modified": "2018-12-10T15:46:20.897228",
  "@@locale":"zh_CH",
  "title": "Flutter应用",
  "@title": {
    "description": "Title for the Demo application",
    "type": "text",
    "placeholders": {}
  }
}
```

我们根据"@@locale"字段可以看出这个arb对应的是中文简体的翻译，里面的`title`字段对应的正是我们应用标题的中文简体翻译。`@title`字段是对`title`的一些描述信息。

接下来，我们在lib目录下创建一个i10n的目录，该目录用于保存从arb文件生成的dart代码文件。

第二步：实现Localizations和Delegate类

和上一节中的步骤类似，我们仍然要实现Localizations和Delegate类，不同的是，现在我们在实现时要使用intl包的一些方法（有些是动态生成的）。

下面我们在`lib/i10n`目录下新建一个“localization_intl.dart”的文件，文件内容如下：

```
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'messages_all.dart'; //1

class DemoLocalizations {
  static Future<DemoLocalizations> load(Locale locale) {
    final String name = locale.countryCode.isEmpty ?
locale.languageCode : locale.toString();
    final String localeName = Intl.canonicalizedLocale(name);
    //2
    return initializeMessages(localeName).then((b) {
      Intl.defaultLocale = localeName;
      return new DemoLocalizations();
    });
  }

  static DemoLocalizations of(BuildContext context) {
    return Localizations.of<DemoLocalizations>(context,
DemoLocalizations);
  }

  String get title {
    return Intl.message(
      'Flutter APP',
      name: 'title',
      desc: 'Title for the Demo application',
    );
  }
}

//Locale代理类
class DemoLocalizationsDelegate extends
LocalizationsDelegate<DemoLocalizations> {
  const DemoLocalizationsDelegate();
```

```
  //是否支持某个Local
  @override
  bool isSupported(Locale locale) => ['en',
'zh'].contains(locale.languageCode);

  // Flutter会调用此类加载相应的Locale资源类
  @override
  Future<DemoLocalizations> load(Locale locale) {
    //3
    return  DemoLocalizations.load(locale);
  }

  // 当Localizations Widget重新build时，是否调用load重新加载Locale资源.
  @override
  bool shouldReload(DemoLocalizationsDelegate old) => false;
}
```

注意：

- 注释1的"messages_all.dart"文件是通过intl_translation工具从arb文件生成的代码，所以在第一次运行生成命令之前，此文件不存在。注释2处的`initializeMessages()`方法和"messages_all.dart"文件一样，是同时生成的。
- 注释3处和上一节示例代码不同，这里我们直接调用`DemoLocalizations.load()`即可。

第三部：添加需要国际化的属性

现在我们可以在DemoLocalizations类中添加需要国际化的属性或方法，如上面示例代码中的`title`属性，这时我们就要用到Intl库提供的一些方法，这些方法可以帮我们轻松实现不同语言的一些语法特性，如复数语境，举个例子，比如我们有一个电子邮件列表页，我们需要在顶部显示未读邮件的数量，在未读数量不同事，我们展示的文本可能会不同：

未读邮件数	提示语
0	There are no emails left
1	There is 1 email left
n(n>1)	There are n emails left

我们可以通过`Intl.plural(...)`来实现：

```
remainingEmailsMessage(int howMany) => Intl.plural(howMany,
    zero: 'There are no emails left',
    one: 'There is $howMany email left',
    other: 'There are $howMany emails left',
    name: "remainingEmailsMessage",
    args: [howMany],
    desc: "How many emails remain after archiving.",
    examples: const {'howMany': 42, 'userName': 'Fred'});
```

可以看到通过`Intl.plural`方法可以在`howMany`值不同时输出不同的提示信息。

Intl包还有一些其他的方法，读者可以自行查看其文档，本书不在赘述。

第四步：生成arb文件

现在我们可以通intl_translation包的工具来提取代码中的字符串到一个arb文件，运行如下命名：

```
flutter pub pub run intl_translation:extract_to_arb --output-
dir=i10n-arb \ lib/i10n/localization_intl.dart
```

运行此命令后，会将我们之前通过Intl API标识的属性和字符串提取到“i10n-arb/intl_messages.arb”文件中，我们看看其内容：

```
{

  "@@last_modified": "2018-12-10T17:37:28.505088",
  "title": "Flutter APP",
  "@title": {
    "description": "Title for the Demo application",
    "type": "text",
    "placeholders": {}
  },
  "remainingEmailsMessage": "{howMany,plural, =0{There are no
emails left}=1{There is {howMany} email left}other{There are
{howMany} emails left}}",
  "@remainingEmailsMessage": {
    "description": "How many emails remain after archiving.",
    "type": "text",
    "placeholders": {
      "howMany": {
        "example": 42
      }
    }
  }
}
```

这个是默认的Locale资源文件，如果我们现在要支持中文简体，只需要在该文件同级目录创建一个"intl_zh_CN.arb"文件，然后将"intl_messages.arb"的内容拷贝到"intl_zh_CN.arb"文件，接下来将英文翻译为中文即可，翻译后的"intl_zh_CN.arb"文件内容如下：

```
{
  "@@last_modified": "2018-12-10T15:46:20.897228",
  "@@locale":"zh_CH",
  "title": "Flutter应用",
  "@title": {
    "description": "Title for the Demo application",
    "type": "text",
    "placeholders": {}
  },
  "remainingEmailsMessage": "{howMany,plural, =0{没有未读邮件}=1{有{howMany}封未读邮件}other{有{howMany}封未读邮件}}",
  "@remainingEmailsMessage": {
    "description": "How many emails remain after archiving.",
    "type": "text",
    "placeholders": {
      "howMany": {
        "example": 42
      }
    }
  }
}
```

我们必须要翻译`title`和`remainingEmailsMessage`字段，`description`是该字段的说明，通常给翻译人员看，代码中不会用到。

有两点需要说明：

1. 如果某个特定的arb中缺失某个属性，那么应用将会加载默认的arb文件(intl_messages.arb)中的相应属性，这是Intl的托底策略。
2. 每次运行提取命令时，intl_messages.arb都会根据代码重新生成，但其他arb文件不会，所以当要添加新的字段或方法时，其他arb文件是增量的，不用担心会覆盖。
3. arb文件是标准的，其格式规范可以自行了解。通常会将arb文件交给翻译人员，当他们完成翻译后，我们再通过下面的步骤根据arb文件生成最终的dart代码。

第五步：生成dart代码

最后一步就是根据arb生成dart文件：

```
flutter pub pub run intl_translation:generate_from_arb --output-
dir=lib/i10n --no-use-deferred-loading
lib/i10n/localization_intl.dart i10n-arb/intl_*.arb
```

这句命令在首次运行时会在"lib/i10n"目录下生成多个文件，对应多种Locale，这些代码便是最终要使用的dart代码。

总结

至此，我们将使用Intl包对APP进行国际化的流程介绍完了，我们可以发现，其中第一步和第二步只在第一次需要，而我们开发时的主要的工作都是在第三步。由于最后两步在第三步完成后每次也都需要，所以我们可以将最后两步放在一个shell脚本里，当我们完成第三步或完成arb文件翻译后只需要分别执行该脚本即可。我们在根目录下创建一个intl.sh的脚本，内容为：

```
flutter pub pub run intl_translation:extract_to_arb --output-
dir=i10n-arb lib/i10n/localization_intl.dart
flutter pub pub run intl_translation:generate_from_arb --output-
dir=lib/i10n --no-use-deferred-loading
lib/i10n/localization_intl.dart i10n-arb/intl_*.arb
```

然后授予执行权限：

```
chmod +x intl.sh
```

执行intl.sh

```
./intl.sh
```