Globalize Your SwiftUI App: A Comprehensive Guide to Localization

fatbobman (东坡肘子) · Following

Published in Mobile App Circular · 16 min read · May 2

Photo by Alexandra Lawrence on Unsplash

When we use an English app, many people will first check if there is a corresponding Chinese version (the author uses Chinese, and this article is originally written for Chinese developers). It is evident that displaying the most familiar language text in the app is so important to users. For a considerable number of apps, if the text displayed in the UI can be localized, the localization work of the app is basically completed. In this article, we will explore how to achieve text localization in iOS development. The demo in this article is written in SwiftUI.

The principle of text localization

As a programmer, if you are asked to design a logic to localize original text for different languages, most people would consider using a dictionary (key-value pair) solution. Apple also takes the same approach, by creating multiple dictionaries for different languages, the system can easily find the localized text (value) corresponding to an original text (key). For example:

```
//en
"hello" = "Hello";
```

```
//zh
"hello" = "你好";
```

This set of methods is the main localization approach taken in this article for text localization.

During code compilation, the system marks the `text that can be localized`. When the app runs in a different language environment (such as French), the system will try to find the corresponding content from the French text key-value file for replacement. If it cannot find it, it will continue to search according to the language preference list. For some types such as `LocalizedStringKey`, the above action will be automatically completed, but for the most commonly used `String` in the code, this action needs to be explicitly completed in the code.

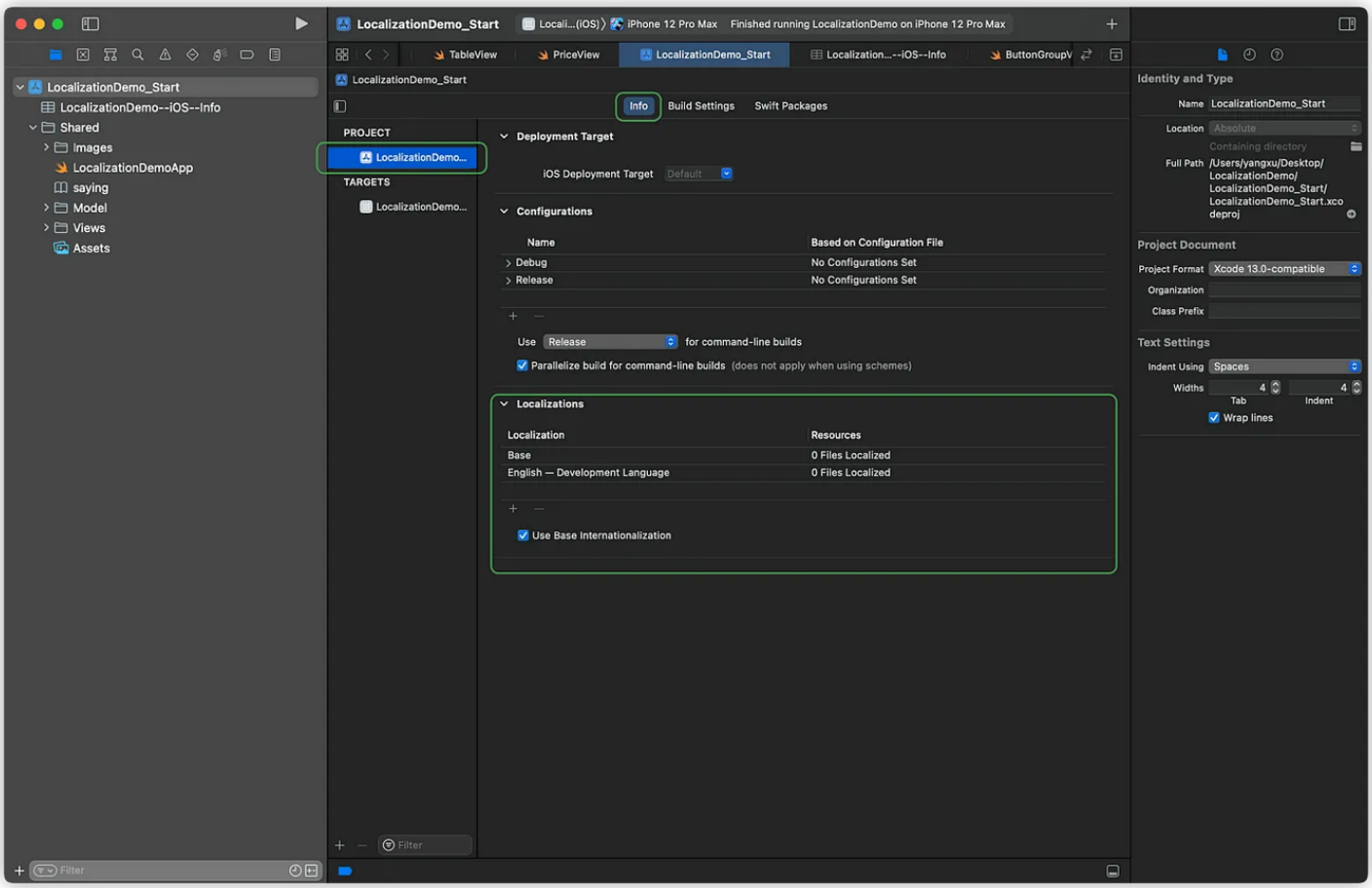
Fortunately, most of SwiftUI's controls (some of which currently have bugs) prioritize the use of the `LocalizedStringKey` constructor for text types, which greatly reduces the workload of manual processing for developers.

Adding Language

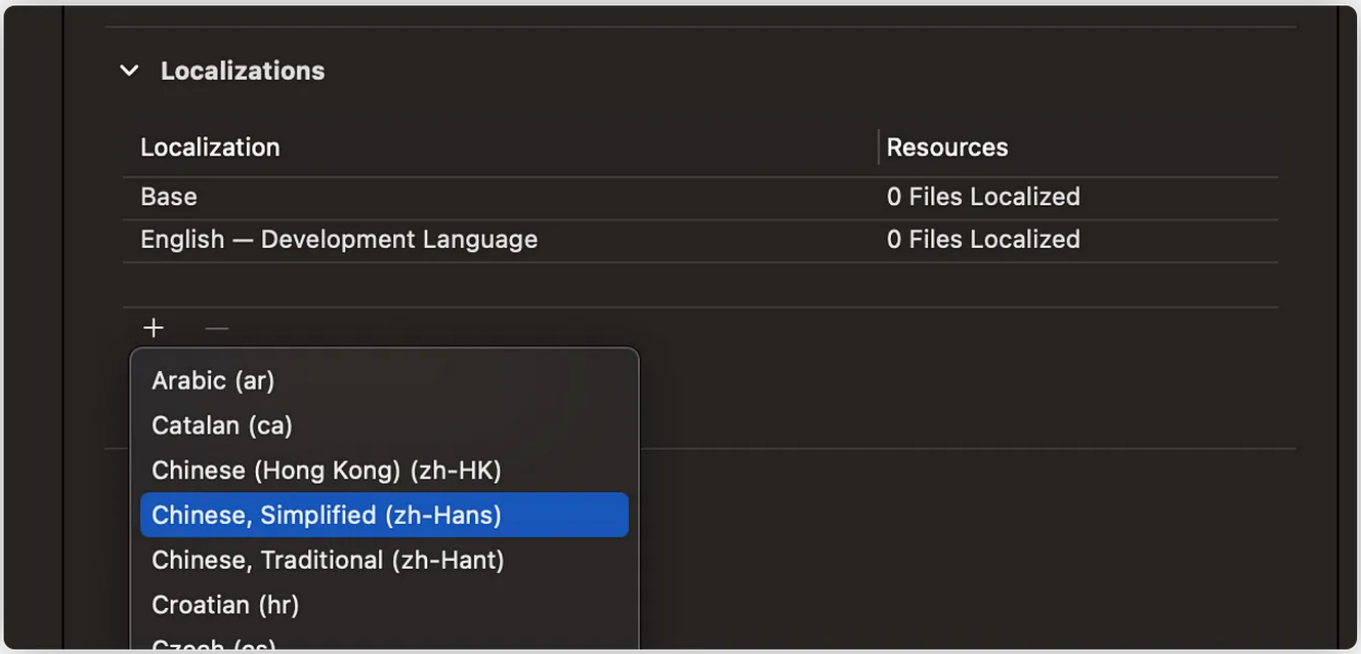
For contemporary programming languages and development environments, internationalization development capability is a must-have function. When we create a project in Xcode, by default, the app is developed only for its corresponding Development Language.

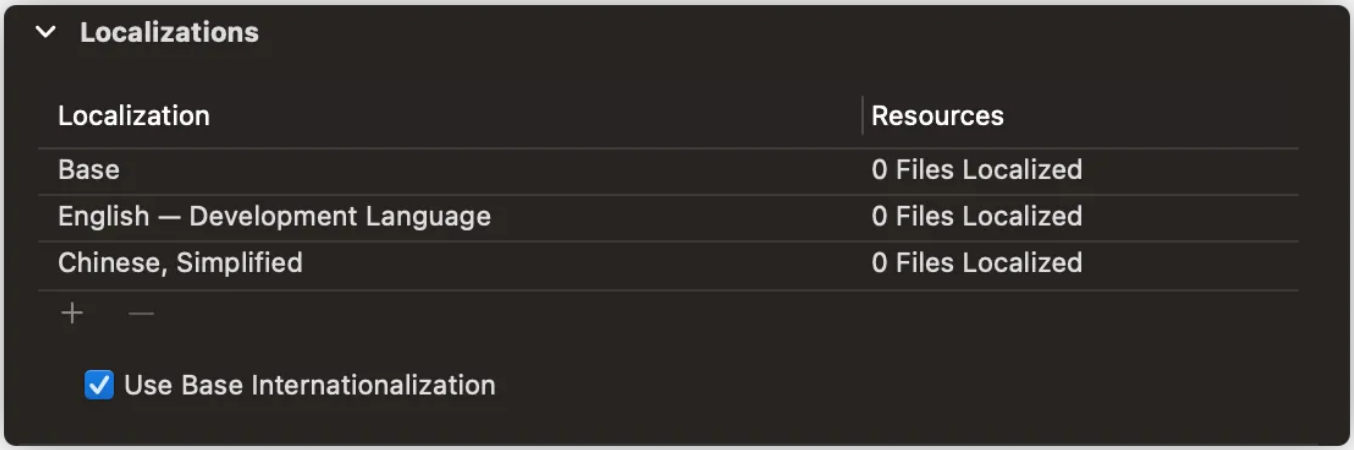
Therefore, we must first let the project know that we will perform localization operations on the project and select the corresponding language.

In the `Project Navigation`, click `PROJECT`, select `Info` to add languages in `Localizations`.



Click the + icon and choose the language that we are going to add.



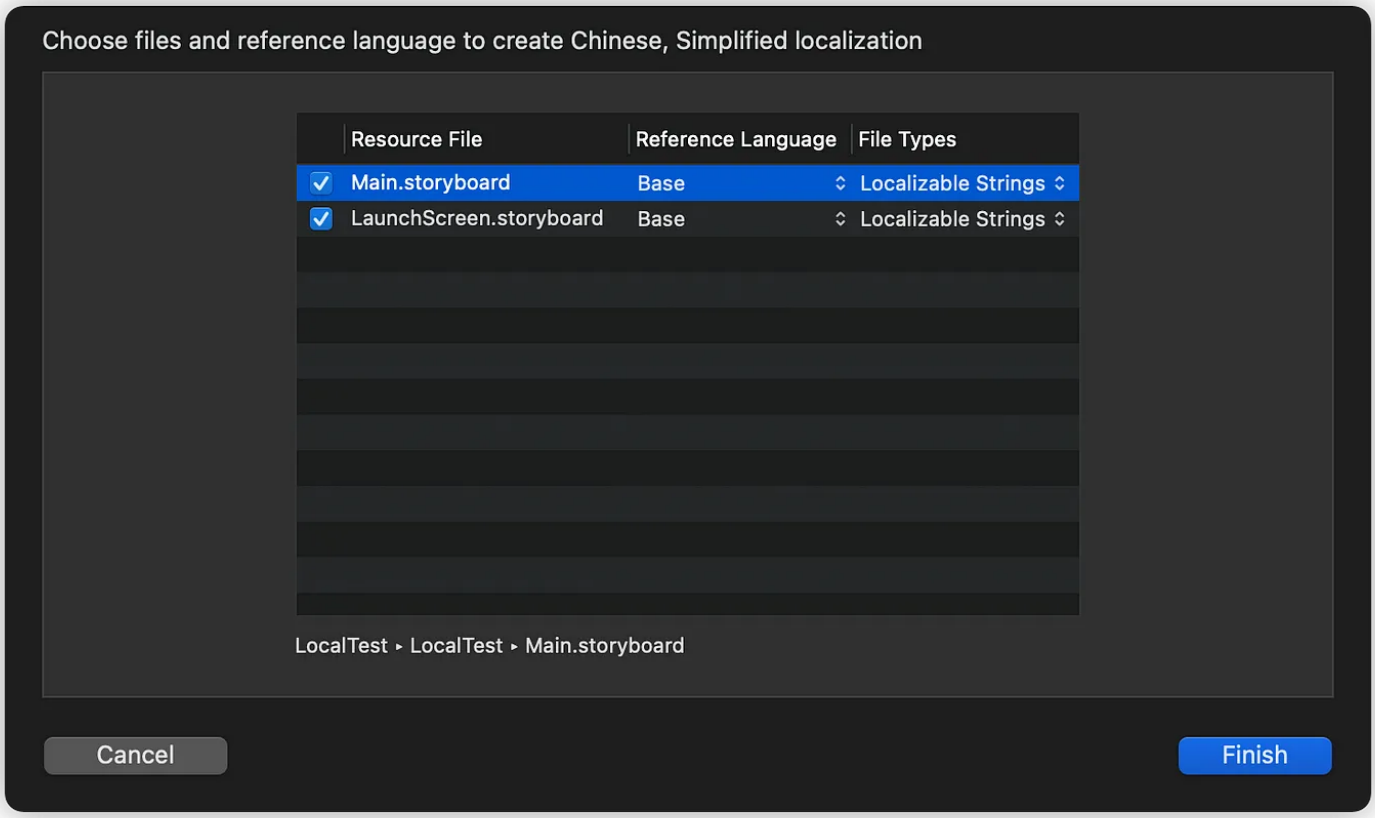


Here we are just informing the project that we may localize the languages listed. However, we still need to set it separately for how to localize and which files and resources to localize.

Enabling “Use Base Internationalization” will modify your project folder

structure in Xcode. Xib and storyboard files will be moved to the Base.lproj folder, while string elements will be extracted to the project's localization folder. This option is for developing with storyboards, and if you adopt SwiftUI, you don't need to worry about it.

For the UIKit framework, Xcode will let you choose the association method for the "storyboard". Since the "Demo Project" used in this article is based on SwiftUI architecture, there will be no such screen.



Create Localized String File

In Apple's development environment, the file type for the `string file` (text

key-value pair file) mentioned in our previous article is `.strings`. We can create multiple string files in an app, and some of them have special meanings with their names.

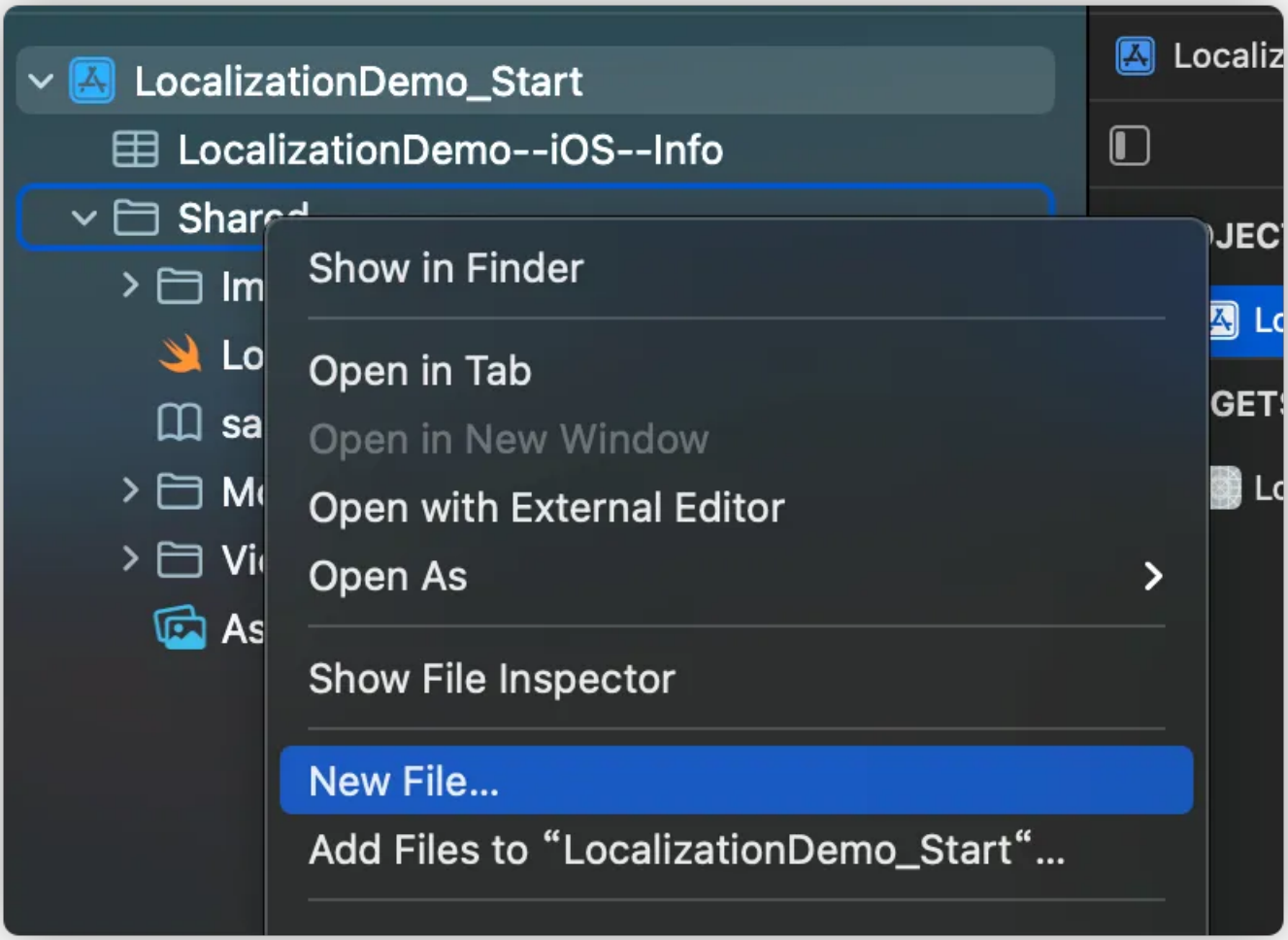
- Localizable.strings

The default string file for UI. If no string file name is specified, the app will retrieve the corresponding localized text content from Localizable.strings.

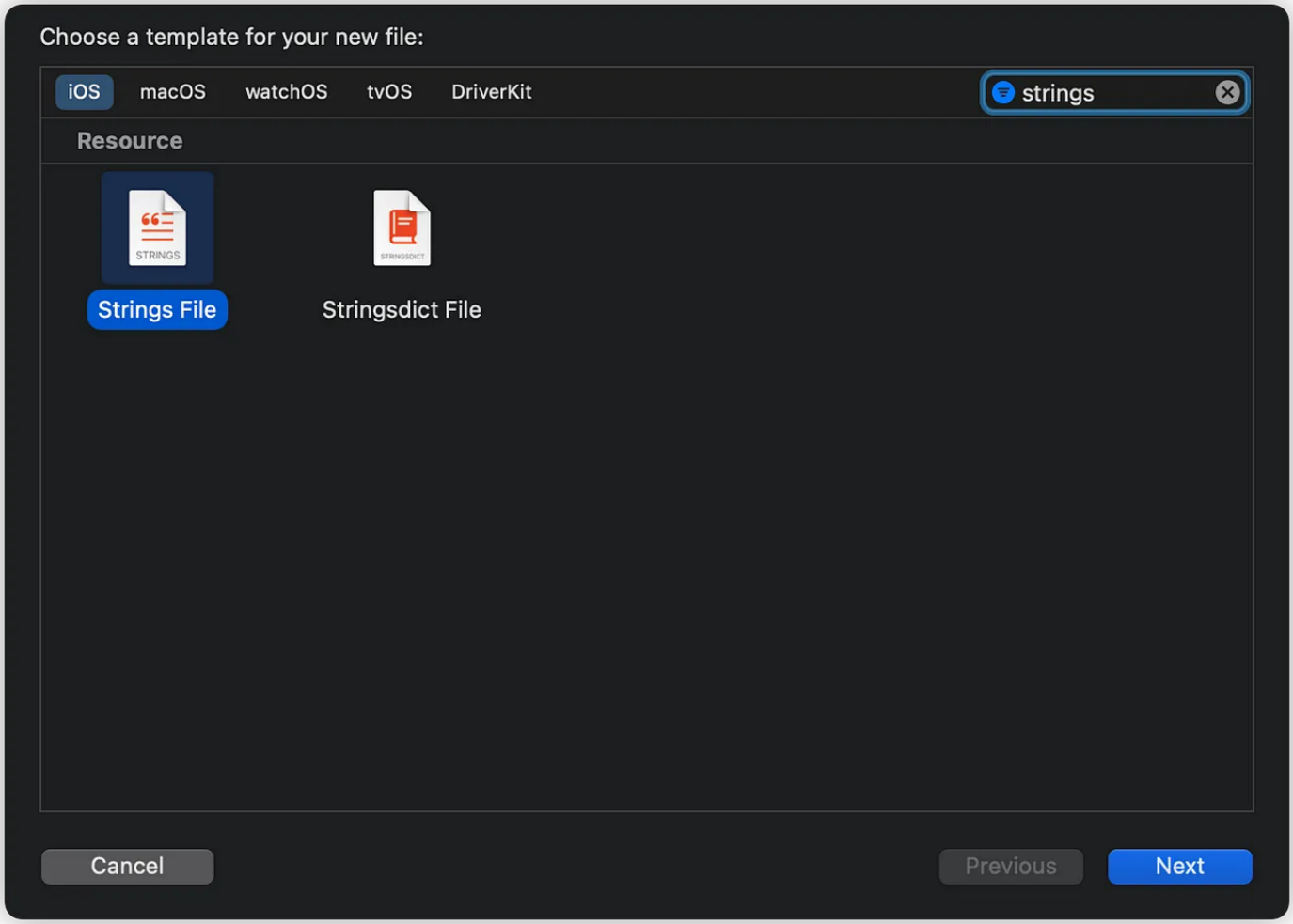
- InfoPlist.strings

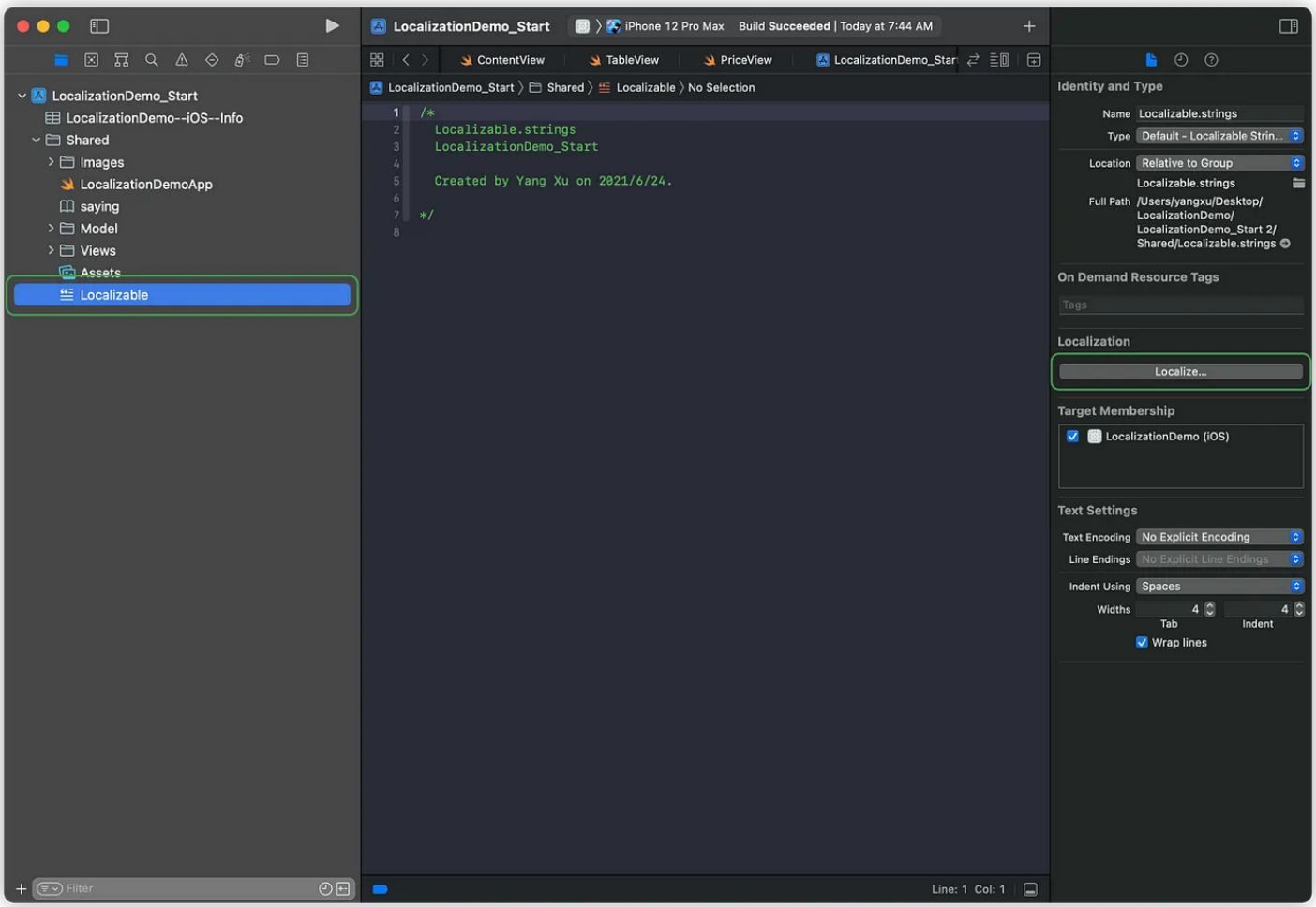
The string file corresponding to Info.plist. Usually used for localizing app names, permission warning prompts, and other content.

n the `Project Navigation`, we select `New File`.

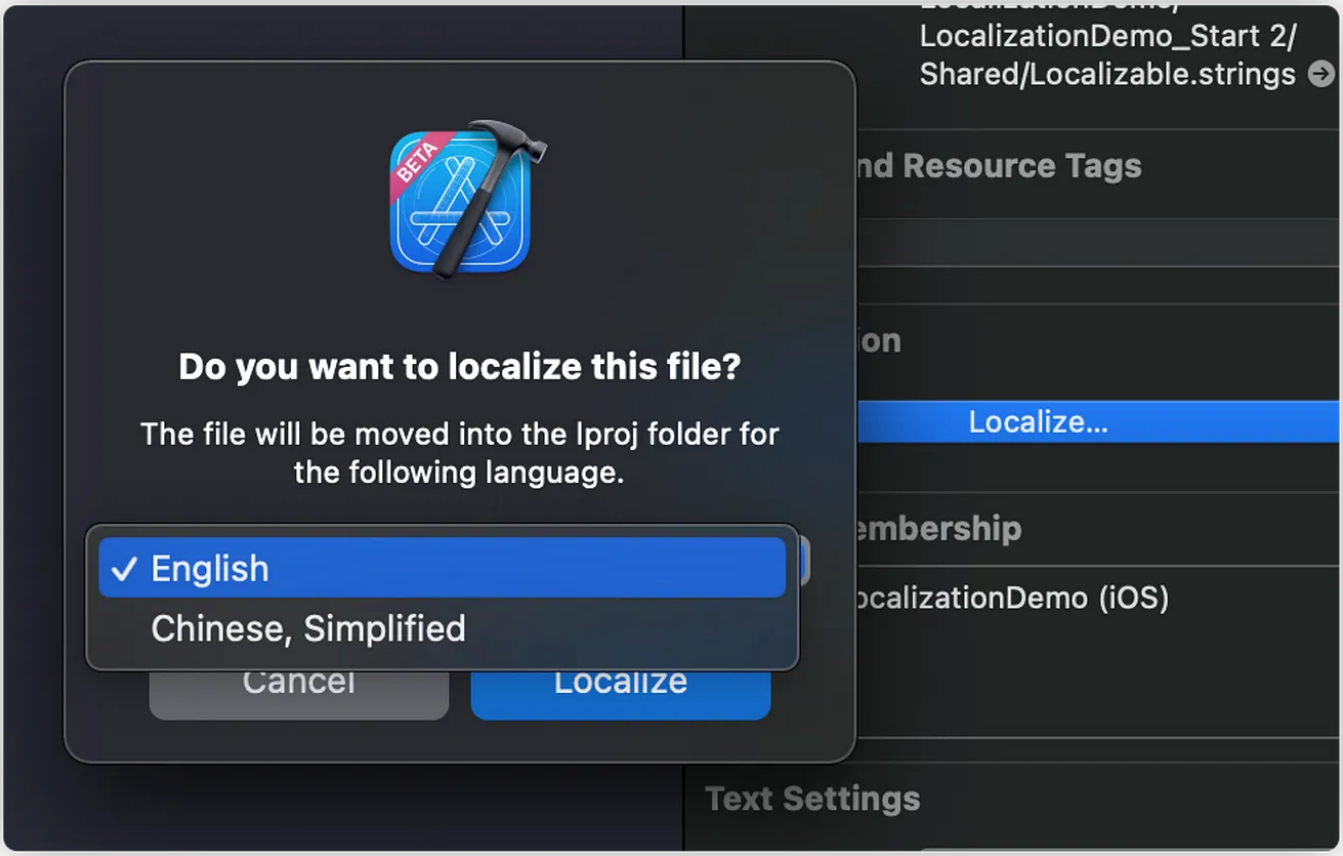


Choose the file type `Strings File` and name it Localizable.strings.

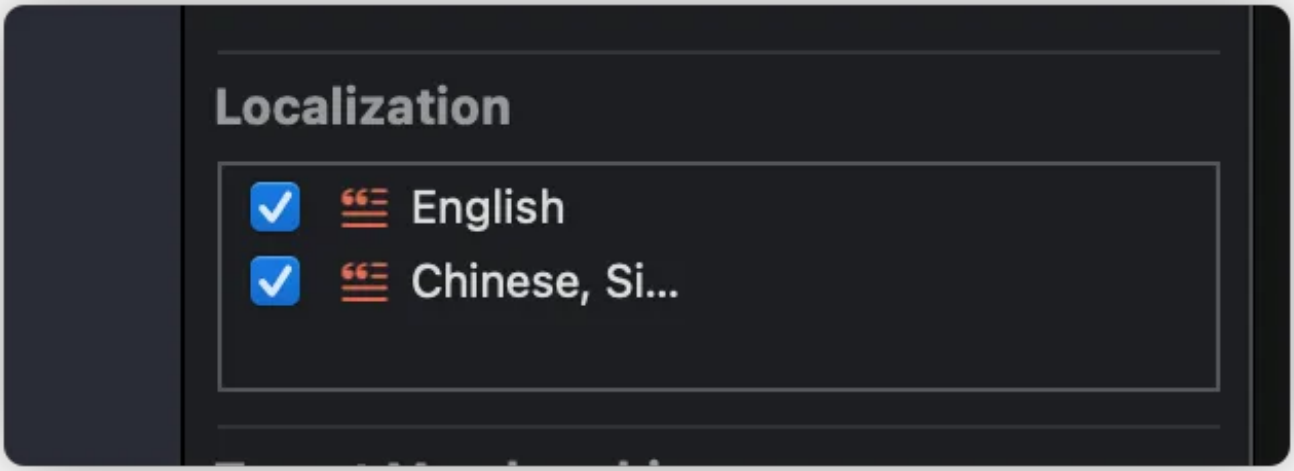

Choose a template for your new file:
iOS
macOS
watchOS
tvOS
DriverKit
strings
Resource
STRINGS
Strings File
STRINGSDICT
Stringsdict File
Cancel
Previous
Next



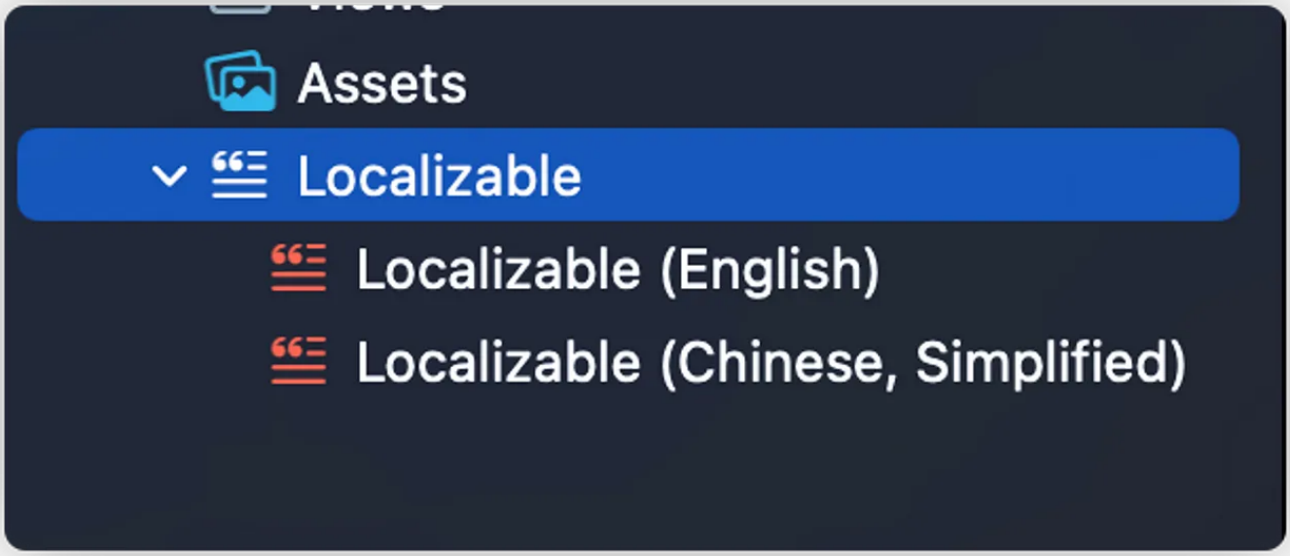
At this point, the `Localizable.strings` file has not been localized yet. Currently, there is only one file in your project where you define text key-value pairs, which will only apply to the project's `development language`. By clicking the `Localize...` button on the right, we can choose to generate the corresponding language files for `Localizable.strings` (the language list is based on the languages added in the project settings).



After checking both languages on the right side,

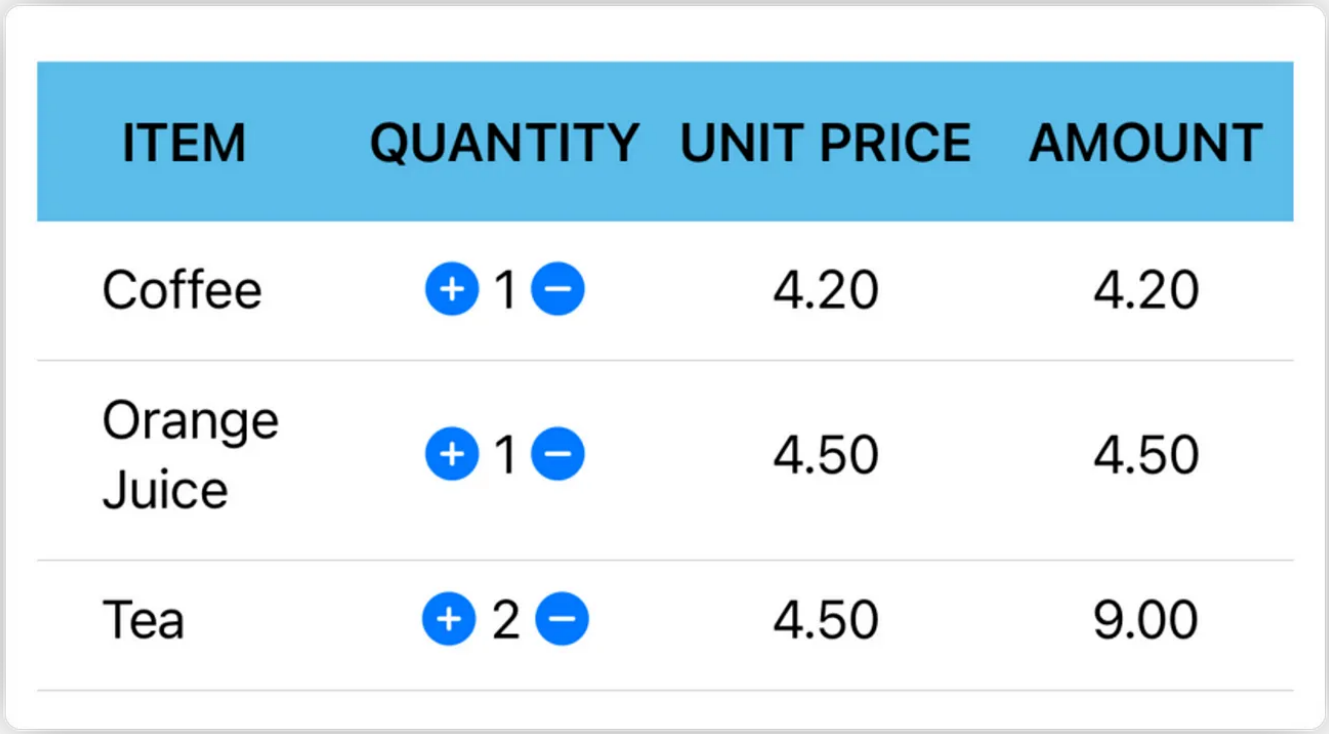


Localizable.strings in the left Project Navigation will be in the following status:



Currently, the `English` and `Chinese` files are empty. We can now create the corresponding text key-value pairs here.

Practical 1: Localize bill table column names



In this section, we attempt to provide corresponding Chinese localization texts for ITEM, QUANTITY, UNIT PRICE, and AMOUNT.

Following the key-value pair declaration rules, we add the following content to the `Localizable.Strings (Chinese)` file:

```
"ITEM" = "种类";
"QUANTITY" = "数量";
"UNIT PRICE" = "单价";
"AMOUNT" = "合计";
```

Open `TableView` and add localization environment configuration in the

preview.

```
TableView()
            .environmentObject(Order.sampleOrder)
            .previewLayout(.sizeThatFits)
            .environment(\.locale,Locale(identifier: "zh"))
```

What changes do we see in the Preview area at this point? **Nothing has changed!**

The reason is that the `key` we set in the `string file` is incorrect. The `ITEM` we see in the app presentation corresponds to the following code in the `TableView`:

```
HStack{
            Text("Item")
                .frame(maxWidth:.infinity)
            Text("Quantity")
                .frame(maxWidth:.infinity)
            Text("Unit Price")
                .frame(maxWidth:.infinity)
            Text("Amount")
                .frame(maxWidth:.infinity)
        }
        .foregroundStyle(.primary)
        .textCase(.uppercase) //转换成大写
```

In the `Text` file, `Item` is used as the key for lookup, but we defined it as

`ITEM`, so the corresponding value was not found. Note: keys in the string file are case-sensitive.

Modify the `chinese` file as follows:

```
"Item" = "种类";
"Quantity" = "数量";
"Unit Price" = "单价";
"Amount" = "合计";
```

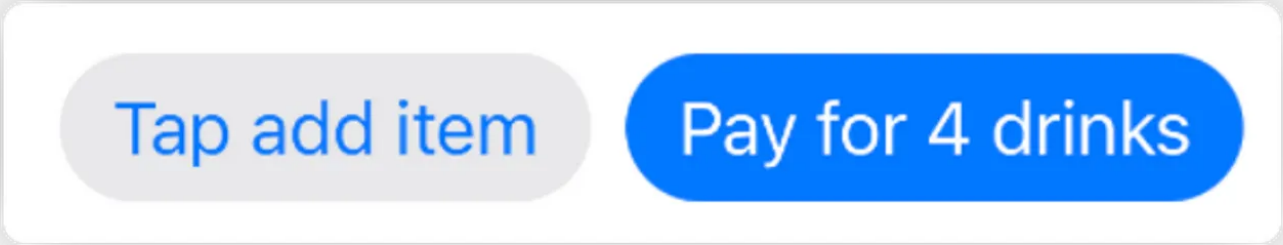
At this point, in the preview window, we can see the results of the localization:

Congratulations, you have already mastered most of the content of text localization.

- I don't know if you noticed that the current `English` file is empty, and we have only set corresponding localized text for four contents in the `Chinese` file. Therefore, the app will display the original text we set in the code for all contents that we have not set.

> When defining in the string file, it is easy to make two mistakes: 1. inputting Chinese punctuation incorrectly, 2. forgetting the semicolon at the end.

Practice 2: Localize the Payment Button



In this section, we will attempt to translate the text in "Pay for 4 drinks" into Chinese.

The button is defined in the `ButtonGroupView` as follows:

```
Button {
      showPayResult.toggle()
    } label: {
      Text("Pay for \(order.totalQuantity) drinks")
    }
```

How to set the corresponding key in the Localizable.strings file for “Pay for \(order.totalQuantity) drinks”?

For localized strings that use string interpolation, we need to use string format specifiers. Apple’s official documentationprovides a detailed comparison of usage instructions for us.

In the code, “order.totalQuantity” corresponds to an “Int” (on a 64-bit system, “Int” in Swift corresponds to “Int64”), so we need to use “%lld” in the key-value pair to replace it. Define it in the “Chinese” file as follows:

```
"Pay for %lld drinks" = "为%lld 杯饮品付款";
```

Thus, we get the desired result. When you try to add or reduce the quantity of the drinks, the quantity in the text will change accordingly.

> Choose the correct format specifier for your interpolation, for example, if the above example is set to `%d`, the system will recognize it as another key and will not be able to complete the conversion.

Practice 3: Localizing the App's Program Name

In Xcode projects, we usually configure some specific system parameters in the `Info.plist` file, such as `Bundle identifier`, `Bundle name`, etc. If we need to localize some of these configurations, we can use the `InfoPlist.strings` file mentioned earlier.

Using the same steps as creating the `Localizable.strings` file, we create a string file named `InfoPlist.strings` (don't forget to localize the created file and ensure that both Chinese and English are selected).

Add the following content to the `Chinese` and `English` files respectively in `InfoPlist.strings`:

```
//chinese
"CFBundleDisplayName" = "肥嘟嘟酒吧";
//english
"CFBundleDisplayName" = "FatbobBar";
```

At this point, if you install the app on a simulator or a real device, the app's name will be displayed in the corresponding text for different languages.

> In the current version of Xcode, you can modify the values in the Target's Info section without directly setting the Info.plist.

> Localized configurations do not necessarily have to appear in the info or Info.plist. As long as we set the localized key-value pairs in InfoPlist.strings, the app will prioritize that setting. Usually, besides the app name CFBundleDisplayName, we localize other items in InfoPlist.strings, such as CFBundleName, CFBundleShortVersionString, NSHumanReadableCopyright, and various system permission request descriptions like NSAppleMusicUsageDescription and NSCameraUsageDescription. For more information about the Info.plist parameters, please refer to the official documentation.

Practice 4: Localizing Beverage Names

Add the following content to the `Localizable(Chinese)` string file:

```
"Orange Juice" = "橙汁";
"Tea" = "茶";
"Coffee" = "咖啡";
"Coke" = "快乐水";
"Sprite" = "透心凉";
```

Please refer to `Model/Drink.swift` for the definition of beverages.

> To preview the app, set the local environment variable, change the language of the simulator to Chinese, or change the app language in the Scheme to Chinese.

When running the app, we did not get the expected result. The name of the beverage did **not become Chinese**. By checking `Drink.swift`, we can find the reason: for a clearly defined `String` type, Text does not treat it as a `LocalizedStringKey`.

Previously in `ItemRowView`, we displayed the beverage name using the following code:

```
Text(item.drink.name)
          .padding(.leading,20)
          .frame(maxWidth:.infinity,alignment: .leading)
```

And the definition of the drink's name in `Drink` is as follows:

```
struct Drink:Identifiable,Hashable,Comparable{
    let id = UUID()
    let name:String //String 类型
    let price:Double
    let calories:Double
```

So the easiest way is to modify the code of `ItemRowView`.

```
Text(LocalizedStringKey(item.drink.name))
         .padding(.leading,20)
```

```
            .frame(maxWidth:.infinity,alignment: .leading)
```

> In some cases, we can only obtain String type data, and we may often need to do similar conversions.

After running again, you will see that the names of the drinks in the table have been changed to the correct Chinese display.

Similarly, modify the code in `ItemListView`:

```
// Change
Button(drink.name)
// To
Button(LocalizedStringKey(drink.name))
```

The display of the drink addition list is now working properly:

After the modification, the Chinese names of the drinks can be displayed correctly.

> The above method is a good solution in most cases, but it is not suitable for generating key-value pairs for localization that rely entirely on

> Export Localizations....

In order to sort the localized text more accurately, we can further modify the comparison function of `Drink`:

```
//change
lhs.name < rhs.name
//to
NSLocalizedString(lhs.name,comment: "") < NSLocalizedString(rhs.name,comment: "
```

> NSLocalizedString can obtain the corresponding text value by the given text key.

Translate the text within `InfoView`:

```
var list:String {
        order.list.map(\.drink.name).joined(separator: " ")
}
```

To:

```
order.list.map{NSLocalizedString($0.drink.name, comment: "")}.joined(separator:
```

> Can’t we just define the name of Drink as a LocalizedStringKey type

> directly? Because LocalizedStringKey does not support the Identifiable, Hashable, and Comparable protocols, and Apple has not provided any method to convert LocalizedStringKey to String. Therefore, if we want to define name as LocalizedStringKey type, we need to use some special methods (which require using Mirror, which will not be discussed in this article).

Add Position Indexes for Localized Placeholders

When declaring localized strings, placeholders of the same type may have different word orders in different languages. For example, consider the following date and location:

```
// English
Go to the hospital on May 3
// Chinese
五月三日去医院
```

By adding position indexes to the placeholders, it is easier to adjust the word order in different language versions of the Localizable.strings file. For example:

```
// Localizable.strings - en
"GO %1$@ ON %2$@" = "Go to %1$@ on %2$@";
"HOSPITAL" = "the hospital";

// Localizable.strings - zh
```

```
"GO %1$@ ON %2$@" = "%2$@去%1$@";
"HOSPITAL" = "医院";
```

Currently, we can only add interpolated content based on positional index using the `String.localizedStringWithFormat` method:

```
var string:String{
    let formatString = NSLocalizedString("GO %1$@ ON %2$@", comment: "")
    let location = String(localized: "HOSPITAL", comment: "")
    return String.localizedStringWithFormat(
        formatString,
        location,
        Date.now.formatted(.dateTime.month().day())
    )
}

Text(string)
```

> This method cannot dynamically view changes by modifying the environment value in preview (but it can be done correctly in simulator or real device).

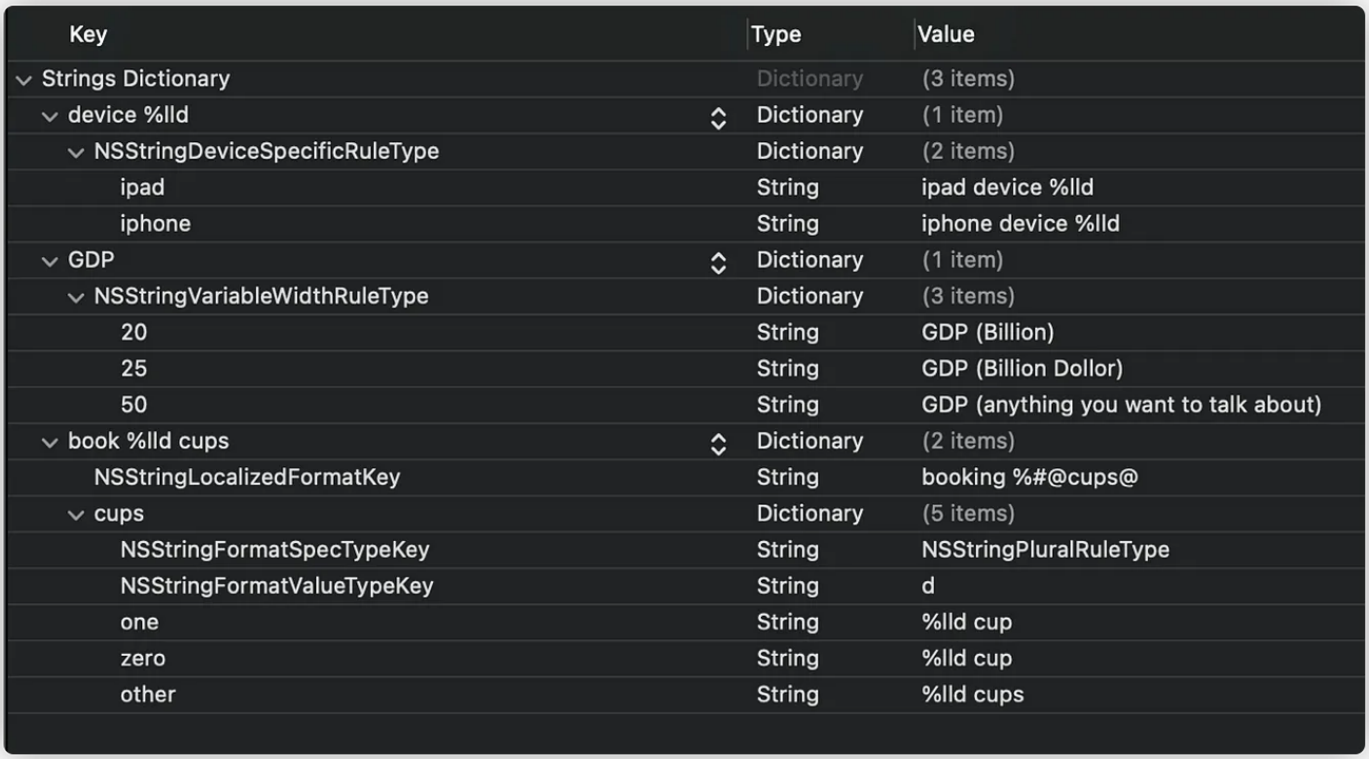
```
if cups <= 1 {
  cupstring = "cup"
}
else {
  cupstring == "cups"
}
```

However, this approach is not conducive to code maintenance and is inflexible for certain languages with complex plural rules, such as Russian, Arabic, etc.

To address how to define plural rules for different languages, Apple provides another solution outside of `.strings` called the `.stringdict` string dictionary file.

It is a property list file with the extension `.stringsdict`, and its operations and editing are identical to other property lists (such as Info.plist).

`.stringsdict` was originally introduced to address pluralization issues, but over the years it has added features such as displaying different text for different values (usually used for changes in screen size) and displaying corresponding text for specific platforms (iPhone, iPad, Mac, tvOS).



Key	Type	Value
Strings Dictionary	Dictionary	(3 items)
device %lld	Dictionary	(1 item)
NSStringDeviceSpecificRuleType	Dictionary	(2 items)
ipad	String	ipad device %lld
iphone	String	iphone device %lld
GDP	Dictionary	(1 item)
NSStringVariableWidthRuleType	Dictionary	(3 items)
20	String	GDP (Billion)
25	String	GDP (Billion Dollor)
50	String	GDP (anything you want to talk about)
book %lld cups	Dictionary	(2 items)
NSStringLocalizedFormatKey	String	booking %#@cups@
cups	Dictionary	(5 items)
NSStringFormatSpecTypeKey	String	NSStringPluralRuleType
NSStringFormatValueTypeKey	String	d
one	String	%lld cup
zero	String	%lld cup
other	String	%lld cups

In the above figure, we have established plural rules using `NSStringLocalizedFormatKey`, variable width rules using `NSStringVariableWidthRuleType`, and device-specific content rules using `NSStringDeviceSpecificRuleType`.

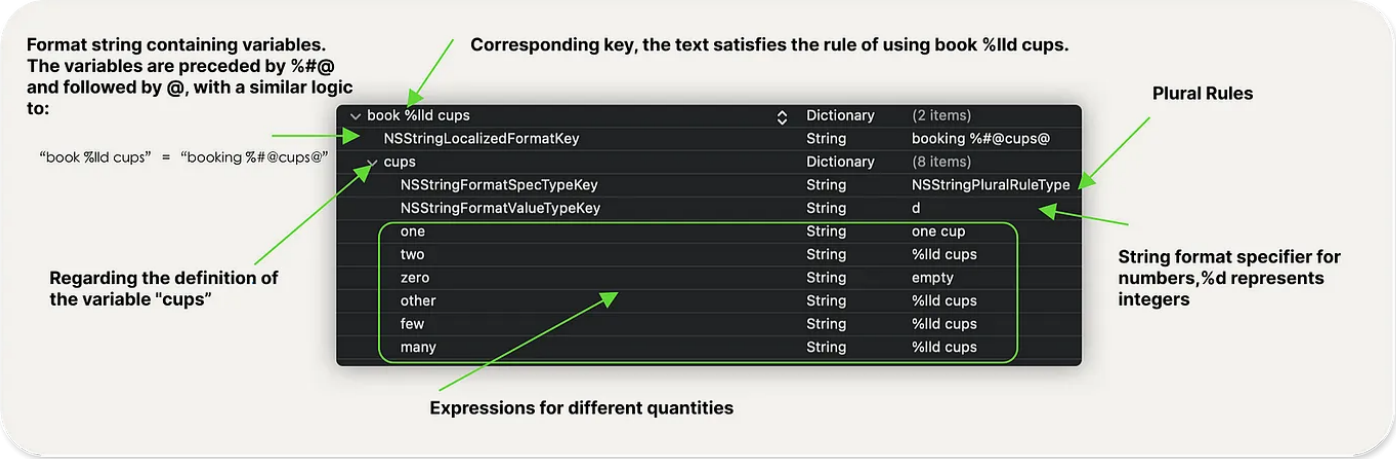
The root node of `.stringdict` is `Strings Dictionary`, and all our rules need to be established under it. We need to create a `Dictionary` for each rule. In the figure above, the three rules correspond to the `keys` of `device %lld`, `GDP`, and `book %lld cups`. When the program encounters text content that satisfies these three `keys`, it will use the corresponding rule to generate the correct localized content.

Therefore, although it may appear slightly different from `.strings`, the

underlying logic is consistent.

- We can establish any number of rules within it.
- The default filename for the corresponding string dictionary file is `Localizable.stringsdict`.
- `.stringdict` has a higher execution priority than `.strings`. For example, if we define `GDP` in both files, only the content corresponding to `.stringdict` will be used.

Establishing plural rules



- The meaning of quantity categories depends on the language, and not all languages have the same categories.

For example, English only uses the `one` and `other` categories to represent plural forms. Arabic has different plural forms for `zero`, `one`, `two`, `few`, `many`, and `other` categories. Although Russian also uses the `many` category,

the rules for numbers in the `many` category are different from the rules in Arabic.

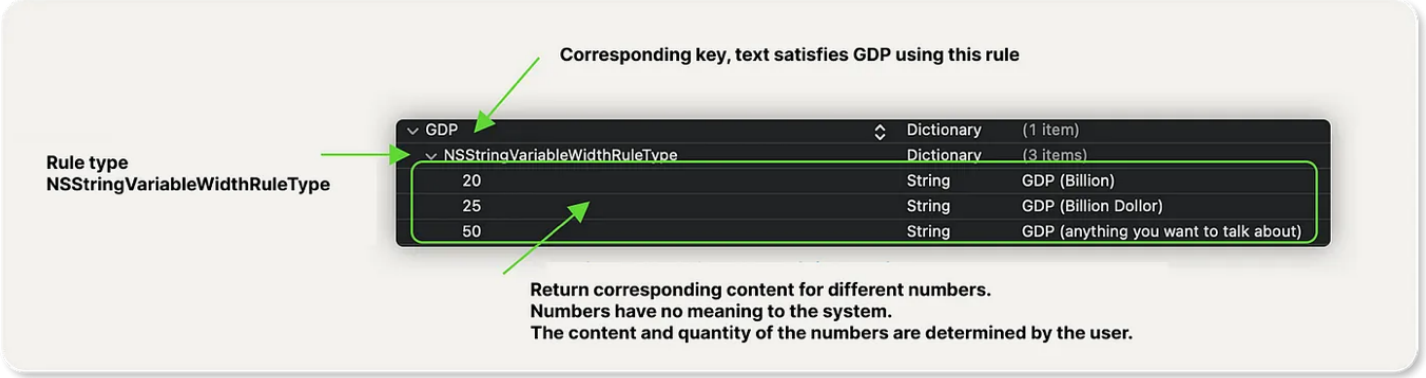
- Except for `other`, all categories are optional.

However, if you do not provide rules for all specific language categories, your text may be grammatically incorrect. Conversely, if you provide rules for categories that are not used in the language, they will be ignored and the `other` format string will be used.

- Using the `NSStringFormatValueTypeKey` format specifier in `zero`, `one`, `two`, `few`, `many`, and `other` format strings is optional. For example, in the above definition, when the number is 1, the returned string is "one cup", and it is not necessary to include the corresponding `%lld`.

> Please refer to the official UNICODE documentation for how to define plural rules in different languages.
>
> For example, English only uses the `one` and `other` categories to indicate plural forms. Arabic has different plural forms for the categories `zero`, `one`, `two`, `few`, `many`, and `other`. Although Russian also uses the `many` category, the rules for the `many` category in numbers are different from the Arabic rules.

Variable width rules



Unlike plural and device rules, the system will not automatically adapt the return value. Users need to explicitly annotate when defining localized text, for example:

```
let gdp = (NSLocalizedString("GDP",comment: "") as NSString).variantFittingPres
Text(gdp) // GDP(Billon Dollor)
let gdp = (NSLocalizedString("GDP",comment: "") as NSString).variantFittingPres
Text(gdp) // GDP(anything you want to talk about)
```

When there are no exact matching numbers, the closest content will be returned.

I feel that its use case is not irreplaceable. After all, there is a higher level of involvement in the code.

Specific Device Rules

Currently supported device types include: appletv, apple watch, ipad, iphone, ipod, mac.

Users do not need to intervene in the code, as the system will return corresponding content based on their hardware device.

Practical 5: Redefining the Payment Button

Use plural rules to improve the payment button.

The code for the payment button is located in `ButtonView`:

```
Button {
     showPayResult.toggle()
   } label: {
      Text("Pay for \(order.totalQuantity) drinks")
  }
```

We need to set up `Pay for \(order.totalQuantity) drinks`.

First, create a `Localizable.stringsdict` file.

For English, we need to set cases for zero, one, and other. In English, we set them up as follows:

English, only need to set zero and other

Adjust the quantity of the order, and the button will return the corresponding localized text according to the different languages and order quantities.

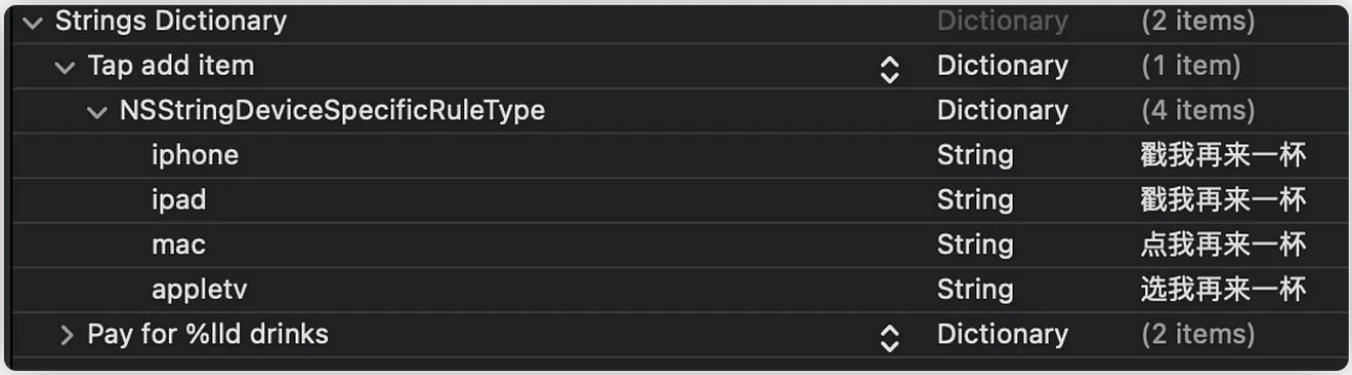
In Practical 2, we set the key-value pair for “Pay for %lld drinks” in Localizable.strings, but because .stringdict has a higher priority, the system will prioritize using the NSStringPluralRuleType rule.

Practice 6: Tap Me or Click Me

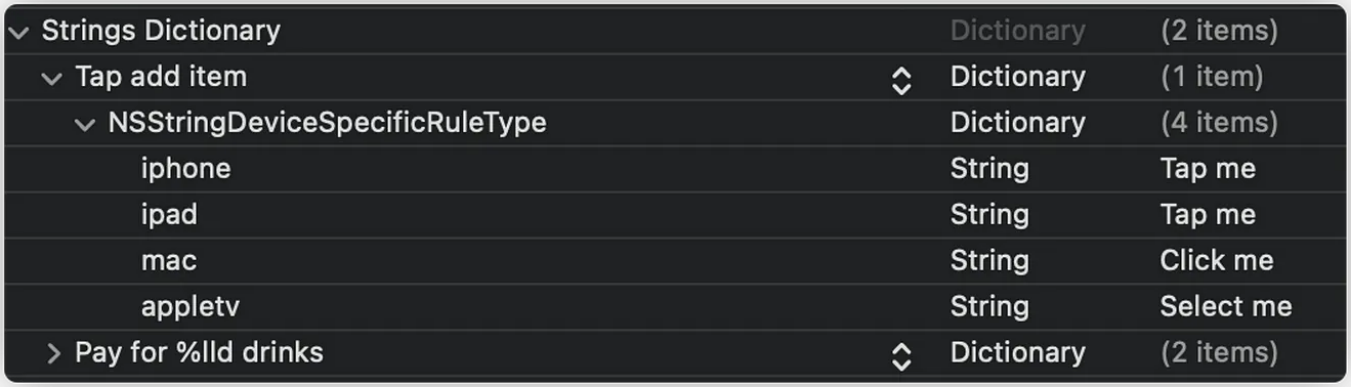
Display different content on the add drink button based on the device.

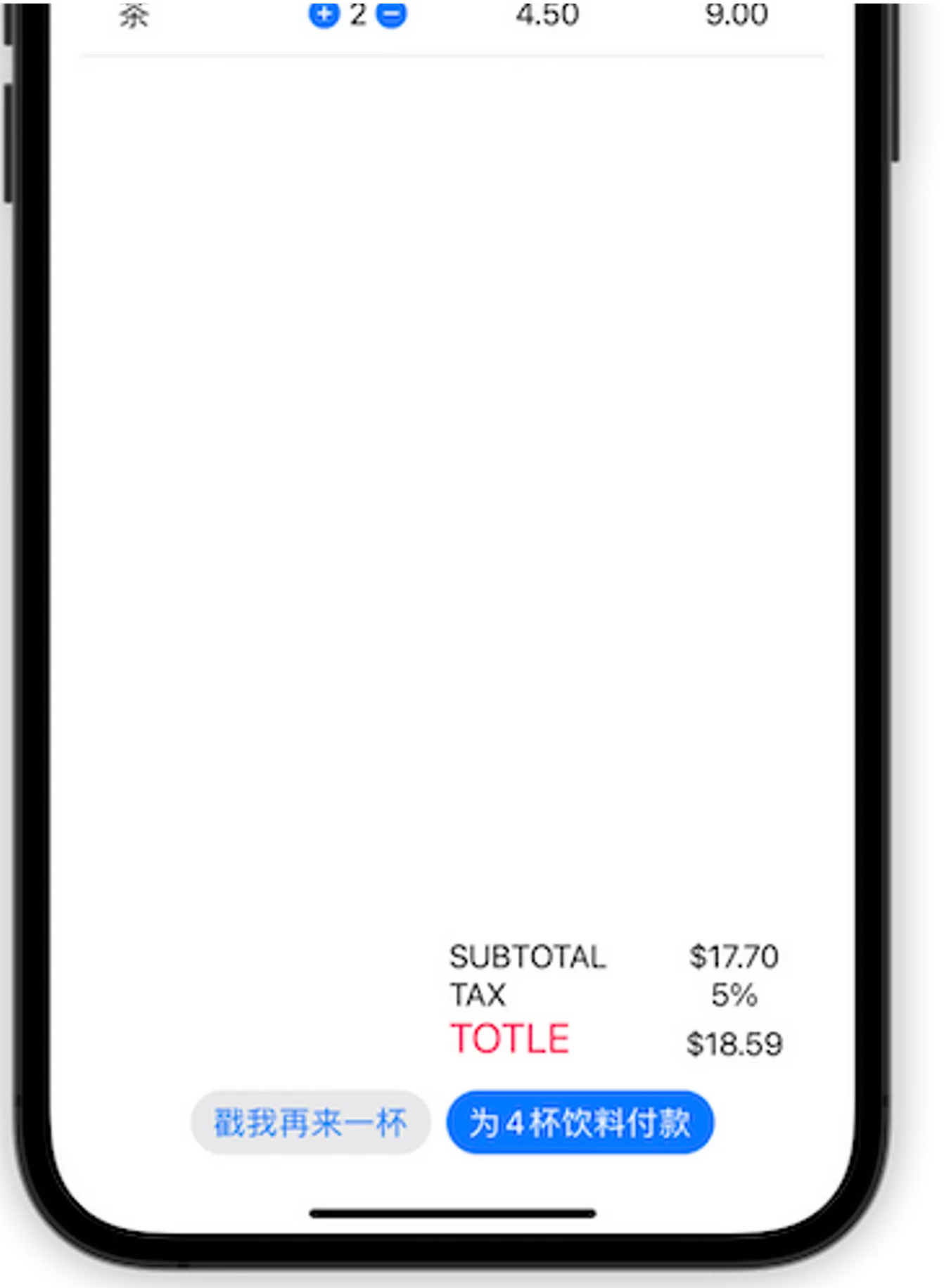
For example, we can display `tap` on iPhone and iPad, `select` on Apple TV, and `click` on Mac.

Add the following in Chinese:



Add in `English`




茶
2
4.50
9.00
SUBTOTAL
$17.70
TAX
5%
TOTLE
$18.59
戳我再来一杯
为4杯饮料付款

Formatter

It's not enough to just localize display labels. In applications, there are many aspects that require localization, such as numbers, dates, currencies, units of measurement, and names.

Apple has invested heavily in providing developers with a complete solution — Formatter.

In 2021, Apple further upgraded Formatter, not only improving the convenience of calling under Swift, but also introducing the FormatStyle protocol suitable for use under Swift.

> *For more information on FormatStyle, please read Apple's New Formatter API: Comparison of Old and New and How to Customize.*

Practical 7: Date, Currency, Percentage

Date

```
Text(order.date,style: .date) //displays year, month, day
Text(order.date.formatted(.dateTime.weekday())) //displays weekday
```

In the demo, we have two ways to localize the display of dates.

- Text itself supports formatting dates, but this way has limited customization.
- Use the new FormatStyle to define output content in a chain:

`order.date.formatted(.dateTime.weekday())` will only display which day of the week

Currency

- Creating NumberFormatter

```
private func currencyFormatter() -> NumberFormatter {
          let formatter = NumberFormatter()
          formatter.numberStyle = .currency
          formatter.maximumFractionDigits = 2
          if locale.identifier != "zh_CN" {
              formatter.locale = Locale(identifier: "en-us")
          }
          return formatter
      }
```

In the demo, only the prices of two currencies are provided. If the system's region setting is not mainland China, the currency will be set to US dollars.

- Applying Formatter in Text

```
Text(NSNumber(value: item.amount),formatter:currencyFormatter() )
```

Since Formatter can only be used for NSObject in Text, Double needs to be converted to NSNumber.

Of course, we can also directly use FormatStyle to format currency display.

Percentage

```
Text(order.tax.formatted(.percent))
```

Practical 8: Units of Measurement, Sequences

Calories

Use MeasureMent to define units of energy. A MeasureMent object represents a quantity and a unit of measurement. The Measurement type provides a programming interface for converting measurement values to different units, and for calculating the sum or difference between two measurement values.

```
init(name: String, price: Double, calories: Double) {
```

```
        self.name = String.localizedStringWithFormat(NSLocalizedString(name, co
        self.price = price
        self.calories = Measurement<UnitEnergy>(value:calories,unit: .calories)
    }
```

Measurement objects can also perform data calculations:

```
var totalCalories:Measurement<UnitEnergy>{
        items.keys.map{ drink in
            drink.calories * Double(items[drink] ?? 0)
        }.reduce(Measurement<UnitEnergy>(value: 0, unit: .calories), +)
    }
```

Create a Formatter to describe the MeasureMent.

```
var measureFormatter:MeasurementFormatter{
        let formatter = MeasurementFormatter()
        formatter.unitStyle = .medium
        return formatter
    }
```

Displaying in SwiftUI

```
Text(order.totalCalories,formatter: measureFormatter)
```

Sequence

Creating hyphenation rules that align with different language conventions (punctuation, and/or, etc.).

```
var list:String {
        order.list.map{NSLocalizedString($0.drink.name, comment: "")}.formatted
    }
```

Other

Use tabname to specify a specific name for string files

You can create multiple string files, and when the filename is not Localizable, you need to specify the file name, such as `Other.strings`.

```
Text("Item",tableName: "Other")
```

`tableName` also applies to `.stringdict`.

Specify string files from other bundles

If your app uses another bundle that contains multilingual resources, you can specify to use string files from that bundle.

```
import MultiLanguPackage // ML
Text("some text",bundle:ML.self)
```

For a Package containing multiple language resources, you can use the following code to specify the Bundle:

```
Text("some text",bundle:Self.self)
```

Markdown Symbol Support

Apple announced at WWDC 2021 that some markdown symbols can be used directly in Text. For example:

```
Text("**Hello** *\(year)*")
```

We can also use markdown symbols in string files.

```
"**Hello** *%lld*" = "**你好** *%lld*";
```

Additionally, the newly added `AttributedString` type can bring more creativity to text.

> For more information about AttributedString, please read AttributedString — Making Text More Beautiful Than Ever

Conclusion

So, if you're trying to get your app seen by more users, localizing it can definitely help with that. And the good news is, it's not even that hard to do in SwiftUI. So why not just go ahead and get it done right now?

DONATE WITH PAYPAL

I hope this article can be helpful to you. You are also welcome to communicate with me through Twitter, Discord channel, or the message board of my blog.

Swiftui Swift IOS App Development IOS Development Mobile App Development

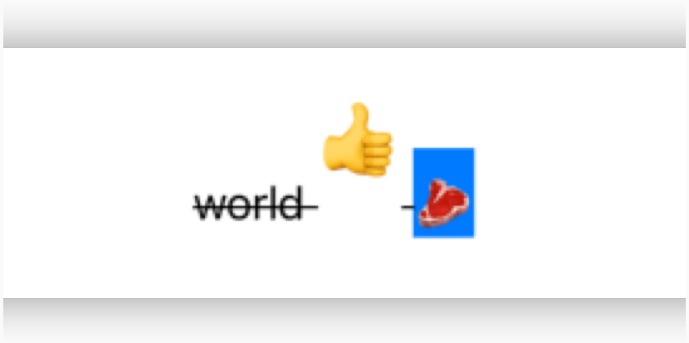
Written by fatbobman (东坡肘子)

Following

109 Followers · Writer for Mobile App Circular

As a developer enthusiast, I enjoy fitness, wellness, and drinking tea. My energy is mainly focused on taking care of my furry pets.

More from fatbobman (东坡肘子) and Mobile App Circular



fatbobman (东坡肘子)

AttributedString—Making Text More Beautiful Than Ever

At WWDC 2021, Apple introduced a long-awaited feature for developers—...

15 min read · Apr 7

29 1

Berke Kurnaz in Mobile App Circular

How did I develop 18 apps in just 5 months?

Hello everyone 😊 I hope everything is perfect for you. Today, I want to write this...

5 min read · Mar 7

449 14

Abdullah Mujahid in Mobile App Circular

The Road Less Traveled: Exploring the Advanced Features of Django...

Django is a robust web framework that allows developers to build complex web...

fatbobman (东坡肘子) in Better Programming

Alignment in SwiftUI: Everything You Need To Know

This article blends the Layout protocol with SwiftUI's "alignment" to aid readers in...

5 min read · Apr 16

126 1

15 min read · May 2

77

See all from fatbobman (东坡肘子)

See all from Mobile App Circular

Recommended from Medium

Garrett Barker

SwiftUI View Life Cycle

In this article, we will discuss the SwiftUI view life cycle and how it works.

4 min read · 4 days ago

85 2

Imad Ali Mohammad

10 Swift Coding Tips—ONE Liner

Swift is an incredibly powerful and expressive programming language that...

3 min read · Apr 27

73

Lists

Staff Picks
300 stories · 62 saves

Stories to Help You Level-Up at Work
19 stories · 19 saves

Self-Improvement 101
20 stories · 44 saves

Productivity 101
20 stories · 42 saves

fatbobman (东坡肘子)

Core Data with CloudKit: Syncing Local Database to iCloud Private...

In this article, we have explored how to synchronize a local database with an iCloud...

11 min read · Mar 17

3

Hemal Asanka

Making API calls with iOS Combine Future Publisher

What Is the Combine Framework?

8 min read · May 5

12

Mobile@Exxeta

SwiftGen—How to neatly get rid of magic strings in iOS projects

It's worth a try

8 min read · May 8

21

Umut SERIFLER

Drag&Drop in SwiftUI

Moving content from one part of an app to another, or from one app to another is...

5 min read · Jan 20

44