ИНФОРМАТИКА И ИНФОРМАЦИОННЫЕ ПРОЦЕССЫ

УДК 004.451.53 + 004.451.9 Обзорная статья
DOI: 10.35330/1991-6639-2024-26-1-29-38
EDN: DVIDUR

Методы решения задачи локализации iOS-приложений

А. Е. Науменко

ООО «Дирион»
346815, Россия, Ростовская область, Мясниковский район,
х. Красный Крым, ул. Юбилейная, 25а

Аннотация. Статья посвящена рассмотрению основных методов локализации приложения на любое количество языков. Рассмотрены методы локализации, доступные для XCode 14, описаны практические случаи их применения, а также даны примеры использования. Помимо этого, рассмотрены задачи, требующие большей гибкости, чем в случае использования стандартных подходов, и варианты их решения, в частности, методы динамической локализации, основанные как на использовании средств runtime, так и на создании собственных локализационных контейнеров.

Ключевые слова: iOS, мобильные приложения, интерфейс, Auto Layout, Swift, Interface Builder, Localization, Localizable, локализация

Поступила 26.01.2024, одобрена после рецензирования 31.01.2024, принята к публикации 09.02.2024

Для цитирования. Науменко А. Е. Методы решения задачи локализации iOS-приложений // Известия Кабардино-Балкарского научного центра РАН. 2024. Т. 26. № 1. С. 29–38. DOI: 10.35330/1991-6639-2024-26-1-29-38

MSC: 68-XX Review article

Methods for solving iOS applications localization task

A.E. Naumenko

“Dirion LLC”
346815, Russia, Rostov region, Myasnikovsky District,
Krasny Krym Farmstead, 25a Yubileynaya street

Abstract. The article is dedicated to exploring the main methods of localizing an application into any number of languages. It examines localization methods available for XCode 14, describes practical cases of their application, and provides examples of use. In addition to this, it looks at tasks that require more flexibility than standard approaches offer and explores solutions for such tasks, specifically, methods of dynamic localization, based both on the use of runtime features and on the creation of custom localization containers.

Keywords: iOS, mobile applications, interface, Auto Layout, Swift, Interface Builder, Localization, Localizable

Submitted 26.01.2024, approved after reviewing 31.01.2024, accepted for publication 09.02.2024

For citation. Naumenko A.E. Methods for solving iOS applications localization task. *News of the Kabardino-Balkarian Scientific Center of RAS.* 2024. Vol. 26. No. 1. Pp. 29–38. DOI: 10.35330/1991-6639-2024-26-1-29-38

Введение

Задача локализации приложения возникает перед разработчиками тогда, когда требуется поддержка более чем одного языка. Предпосылки для этого могут быть разными, самые частые из них – выход на международные рынки или поддержка иностранных пользователей (к примеру, туристов в приложениях такси или аэропортов) на отечественном рынке. В силу того, что задача возникает достаточно часто, в Apple созданы достаточно удобные способы ее решения [1]. В рамках данной статьи мы произведем разбор задач, которые решаются стандартными методами, а также рассмотрим нестандартные подходы для более гибких решений.

Локализация текста

Начнем с самой простой задачи, а именно с локализации текста. Задача состоит в том, чтобы программно в определенном месте приложения показывать разную текстовую информацию, причем зависеть она будет исключительно от выставленного в системе языка.

К примеру, у нас есть текст “Hello World!”. Мы хотим отобразить его на экране в метку, то есть компонент UILabel. Это сделать можно следующим образом:

```
label.text = NSLocalizedString("hello_world", comment: "Приветствие").
```

Мы используем стандартную функцию, которая принимает в себя параметром ключ “hello_world”, а не текст. Комментарий в параметре comment используется для того, чтобы для переводчиков был привязанный контекст к фрагменту текста, который им требуется перевести. Он используется при генерации таблиц, но это мы в данной статье рассматривать не будем.

Рассмотрим теперь таблицы с переводом. Это простые текстовые файлы с расширением strings. В проекте по умолчанию для переводов используется таблица с именем Localizable.strings.

Изначально мы создаем один файл Localizable.strings и заносим в него все необходимые ключи. К примеру, создавая файл для данного проекта, мы внесем в него единственный ключ – "hello_world".

Вот так будет выглядеть наш файл:

```
1
2 "hello_world" = "Hello World!";
3
```

Рис. 1. *Скрин файла локализации*

Fig. 1. *The screen of a localization file*

Затем нам необходимо добавить в проект локализацию. Это делается в настройках проекта, секция Info, раздел Localizations. Добавив новый язык, мы сможем локализовать файл со строками. Просто выделив его, мы сможем в XCode в правой панели добавить один из доступных языков.

Сделав это и заглянув в проект, мы увидим, что файл локализации исчез со старого места. Однако появятся папки, каждая из которых будет иметь имя формата {lang_code}.lproj. Открыв эти папки, мы обнаружим в них новую копию файла Localizable.strings, а в проекте увидим этот файл в виде раскрытой группы:

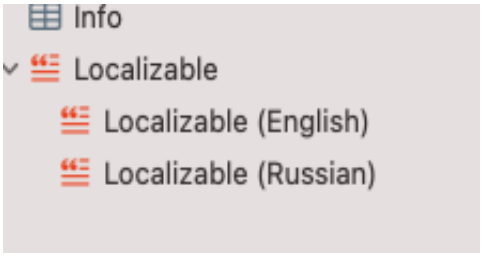

Рис. 2. *Группа файлов локализации*

Fig. 2. *The group of localization files*

Теперь мы можем в каждой копии этого файла вставлять текст соответствующей локализации файла. При смене языка приложение будет перезагружено, и код, который подставляет в метку локализованный текст, будет вызван заново. На этот раз он возьмет значения из файла, который соответствует выбранному пользователем языку, либо в случае отсутствия такового выберет язык по умолчанию. Обычно это английский язык.

Локализация XIB/Storyboards

В этом разделе мы коснемся встроенных средств локализации интерфейса, созданного в файлах XIB и Storyboards [2]. Несмотря на то, что задача локализации такого интерфейса легко может быть решена с помощью локализации текста, Apple предоставляет нам дополнительные инструменты локализации этих файлов.

Локализовать XIB или Storyboard можно двумя способами:

1. Локализовать текстовые значения элементов.
2. Создать копию файла для конкретного языка.

Рассмотрим оба способа. Создадим простую View с единственной меткой:

Рис. 3. *Тестовое представление с одной меткой*

Fig. 3. *The test view with a single label*

Теперь локализуем ее так же, как до этого поступали с файлом Localizable, а именно: нажмем на кнопку Localize в инспекторе объектов и выберем один из языков. Обратите внимание, что по умолчанию справа будет тип локализации "Localizable Strings", который указывает на то, что будут локализовываться только текстовые значения элементов:

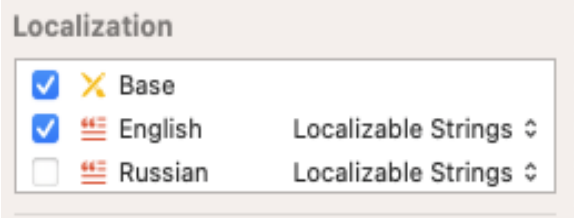

Рис. 4. *Выбор языков локализации*

Fig. 4. *Selecting languages of localization*

В навигаторе мы увидим, что XIB-файл превратился в группу. Появился файл с пометкой "Base", который содержит базовую версию файла, а также .strings файл для выбранной английской локализации. В нем мы увидим не очень читаемый текст:

```
/* Class = "UILabel"; text = "Base label"; ObjectID = "pPR-vp-xuM"; */
"pPR-vp-xuM.text" = "Base label".
```

Он будет означать, что для элемента с определенным ID, в нашем случае “pPR-vp-xuM”, будет локализовано свойство text. Если элементов будет больше, то мы увидим список их всех.

Теперь добавим новую локализацию, на прошлом скрине у нас осталась не выбранной русская. И после добавления изменим тип локализации с "Localizable Strings" на "Interface Builder Cocoa Touch XIB". В этом случае (что можно проверить в том числе и на диске в самой папке с проектом) будет создана копия изначального XIB-файла. Теперь для русской локализации мы можем создавать не только новые значения элементов, но и совершенно другой интерфейс.

Надо отметить, что этот тип локализации не очень часто используется. Дело в том, что при использовании Localizable Strings в XIB и Storyboard, не очень удобно иметь дело с ID элементов. А при добавлении элементов приходится либо перегенерировать файлы, либо добавлять их вручную. Что касается создания копии XIB-файла для определенных языков, то в этом случае совершенно очевидна проблема внесения изменений в интерфейс – придется вносить изменения во все копии файлов.

Локализация параметризованного текста

При работе с множеством языков рано или поздно сталкиваешься с достаточно неприятной задачей, которая заключается в обработке множественного числа. Дело в том, что в разных языках для этого свои правила обработки множественного числа. Разберем простой пример: выведем на русском и английском языке простую фразу "У меня есть N яблок". Рассмотрим варианты на английском в зависимости от числа N:

"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 5 apples"
"I have 51 apples".

Мы видим, что будет только три варианта форматирования: "I have no apples", "I have N apple" и "I have N apples". А теперь рассмотрим то же самое на русском:

"У меня нет яблок"
"У меня есть 1 яблоко"
"У меня есть 2 яблока"
"У меня есть 5 яблок"
"У меня есть 51 яблоко".

Как видим, теперь у нас 4 варианта формата. И если мы будем использовать обычную локализацию текста с помощью файла Localizable.strings, то нам придется вручную реализовывать обработку каждого языка. К счастью, Apple предлагает нам достаточно мощный инструмент для этой задачи. Для его использования необходимо создать файл

Localizable.stringsdict, в котором мы сможем прописать отдельно все варианты для каждого языка.

Рассмотрим пример основной части файла для английской локализации:

```
<dict>
    <key>apples_have</key>
    <dict>
        <key>NSStringLocalizedFormatKey</key>
        <string>I have %#@apples@</string>
        <key>apples</key>
        <dict>
            <key>NSStringFormatSpecTypeKey</key>
            <string>NSStringPluralRuleType</string>
            <key>NSStringFormatValueTypeKey</key>
            <string>d</string>
            <key>zero</key>
            <string>no apples</string>
            <key>one</key>
            <string>%d apple</string>
            <key>other</key>
            <string>%d apples</string>
        </dict>
    </dict>
</dict>
```

Посмотреть все возможные значения свойств можно в документации. Отметим основные моменты:

1. По ключу NSStringLocalizedFormatKey мы задаем полный формат фразы. В ней параметром задаем подстроку, для которой применяется локализация с учетом множественного числа. Затем мы описываем эти параметры, определяя словарь с описанием для каждого, используя сам параметр как ключ.

2. NSStringFormatValueTypeKey – задает тип параметра. Мы выбрали d, что означает целые числа.

3. В опциях zero, one и others мы задаем различные форматы, которые полностью охватывают все возможные значения для английского языка.

Рассмотрим теперь то же самое для русского языка:

```
<dict>
    <key>apples_have</key>
    <dict>
        <key>NSStringLocalizedFormatKey</key>
        <string>У меня %#@apples@</string>
        <key>apples</key>
        <dict>
            <key>NSStringFormatSpecTypeKey</key>
            <string>NSStringPluralRuleType</string>
```

```
                    <key>NSStringFormatValueTypeKey</key>
                    <string>d</string>
                    <key>zero</key>
                    <string>нет яблок</string>
                    <key>one</key>
                    <string>%d яблоко</string>
                    <key>other</key>
                    <string>%d яблок</string>
                    <key>few</key>
                    <string>%d яблока</string>
                </dict>
            </dict>
        </dict>
```

Заметим, что мы добавили опцию **few**. Для русской локализации она будет срабатывать для чисел, заканчивающихся на 2, 3 и 4, за исключением случая, когда перед ними 1. Теперь приведем пример кода, использующего локализацию множественных чисел:

```
let applesFormat = NSLocalizedString("apples_have", comment: "")
label.text = String(format: applesFormat, 23)
```

И запустим код, посмотрев результат на View, созданной ранее в прошлой главе:

Рис. 5. *Результат запуска приложения с локализацией множественного числа*

Fig. 5. *The result of launching the app with plural localization*

То есть мы получаем сначала формат основной строки, а затем в него подставляем параметр. Значение параметра определит нужную форму. Эта магия достигается тем, что строка applesFormat имеет под капотом знание о том, что она получена из файла stringsdict и должна быть должным образом параметризована.

Локализация интерфейса

В этом разделе коснемся темы локализации интерфейса, а точнее локализации *ориентации* интерфейса. Далеко не все с этим сталкиваются, но знать об этом полезно. Речь идет о смене ориентации интерфейса для языков, в которых направление текста идет не слева направо, как привыкли англо- и русскоязычные пользователи, а справа налево. Пример такого языка – арабский. Если переключить iPhone на арабский язык, то можно увидеть не только смену текста, но и смену ориентации всего интерфейса. Кнопка "Назад" будет не в левом верхнем углу, а в правом верхнем. Анимация перехода на новый экран будет не справа налево, а слева направо.

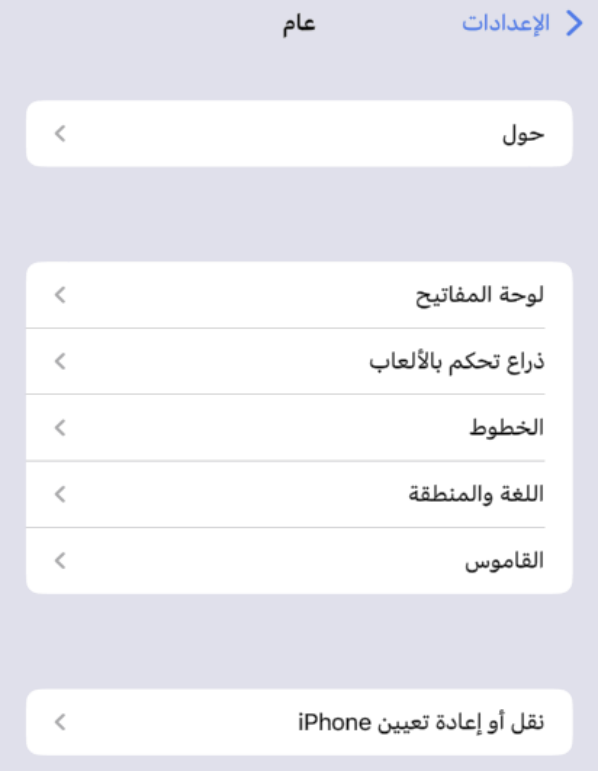

Рис. 6. *Основные настройки в арабской локализации*

Fig. 6. *General settings in Arabic localization*

Стоит отметить, что большую часть работы UIKit делает автоматически. Все анимации и UINavigationController автоматически подстраиваются под выбранный язык, и для арабского они меняют привычное для нас направление. Но вот что необходимо помнить при локализации на несколько языков с разными ориентациями:

1. Использование констрейнтов [3] Left и Right не изменит ориентацию. При использовании их все элементы привязываются конкретно к левому или к правому краю. Что и логично – сами понятия "лево" и "право" не меняются в зависимости от языка.
2. Использование констрейнтов Leading и Trailing изменит ориентацию. В английском Leading будет эквивалентно Left, а в арабском оно же будет эквивалентно Right.
3. Существует программный способ изменения ориентации у View, а именно: установка значения у представления свойства semanticContentAttribute. Оно может принимать два значения – forceRightToLeft и forceLeftToRight. Это может быть полезно при изменении языка и из настроек телефона, и из настроек приложения нестандартными способами.

Динамическая локализация

Перейдем теперь к не совсем стандартной задаче, а именно – к динамическим значениям текстовых блоков для ключей локализации. Задача эта может возникнуть, например, при желании разработчиков хранить локализацию на серверной части и подгружать во время работы приложения. Рассмотрим несколько вариантов решения этой задачи.

Создание нового Bundle

Наверное, самый простой способ этой задачи будет заключаться в создании отдельного объекта Bundle [4]. Когда мы не передаем в функцию локализации какой-либо bundle, у нас

по умолчанию используется Main bundle, который работает с ресурсами нашего приложения. Но ресурсы приложения мы не можем изменять, а значит, нам нужно создать хранилище в папке Documents.

Продемонстрируем простое программное создание собственного bundle и локализационного файла внутри него:

```
let docURL = getDocumentsDirectory()
    let dataPath = docURL.appendingPathComponent("MyLocalizationFolder")

    if !FileManager.default.fileExists(atPath: dataPath.path) {
        do {
            try FileManager.default.createDirectory(
                    atPath: dataPath.path,
                    withIntermediateDirectories: true, attributes: nil)
        } catch {
            print(error.localizedDescription)
        }
    }

    let fileLoc = dataPath.appendingPathComponent("Localizable.strings")
    let content = "\"hello_world\" = \"Новый бандл!!!\";"

    try? content.write(to: fileLoc, atomically: true, encoding: String.Encoding.utf8)

let myBundle = Bundle(url: dataPath)!

    label.text = NSLocalizedString("hello_world", bundle: myBundle, comment: "")
```

В этом примере мы создали новую папку MyLocalizationFolder, в ней создали файл Localizable.strings с локализационным контентом. Затем создали bundle, который смотрит в эту новую папку, и передали его при вызове в функцию NSLocalizedString.

Заметим, что в этом примере мы использовали не статические файлы из ресурсов приложения, а динамический контент из папки приложения. Это дает нам полный контроль над их содержимым. Главное, что нужно помнить, – обязательно следить за форматом файлов, так как генерировать их придется самим. Хотя эту задачу вполне можно переложить в этом случае на серверную часть.

Подмена селекторов

Рассмотрим более экзотический способ работы с локализацией. Заключается он в переопределении функции localizedString, которая вызывается в классе Bundle у его экземпляра в момент, когда мы вызываем функцию NSLocalizedString.

Воспользуемся методом, который называется свизлинг селекторов (Swizzle Method, Swizzling [5]). Заключается он в том, что мы в runtime, то есть во время исполнения приложения, подменяем у класса Bundle реализацию метода. Для этого необходимо написать свою реализацию. Реализуем это в виде расширения класса Bundle:

```
extension Bundle {
    static func swizzleLocalization() {
        let orginalSelector = #selector(localizedString(forKey:value:table:))
```

```
            let orginalMethod = class_getInstanceMethod(self, orginalSelector)!
            let mySelector = #selector(myLocaLizedString(forKey:value:table:))
            let myMethod = class_getInstanceMethod(self, mySelector)!

            method_exchangeImplementations(orginalMethod, myMethod)
        }

        @objc private func myLocaLizedString(forKey key: String,
                                             value: String?,
                                             table: String?) -> String {
            let oldLocResult = myLocaLizedString(forKey: key,
                                             value: value,
                                             table: table)
            // Do or return something
            // ...
            return oldLocResult
        }
    }
```

И теперь при старте приложения вызовем: Bundle.swizzleLocalization().

Это подменит реализации методов у двух сигнатур – старой и новой. Обратите внимание на вызов внутри функции myLocaLizedString самой себя. При чтении кода это очень похоже на рекурсивный вызов, хотя это не так. Давайте разберем этот момент. Рассмотрим сначала отдельно сигнатуру метода и его реализацию. До подмены селекторов картина была такой:

Таблица 1. Соответствие сигнатур и тел методов в runtime до свизлинга

Table 1. Correspondence of method signatures and bodies in runtime prior to Swizzling

Сигнатура **localizedString**	Тело **localizedString**
Сигнатура *myLocaLizedString*	Тело *myLocaLizedString*

После подмены селекторов, то есть вызова метода swizzleLocalization, картина теперь такая:

Таблица 2. Соответствие сигнатур и тел методов в runtime после свизлинга

Table 2. Correspondence of method signatures and bodies in runtime after Swizzling

Сигнатура **localizedString**	Тело *myLocaLizedString*
Сигнатура *myLocaLizedString*	Тело **localizedString**

Соответственно, когда мы вызываем в теле метода myLocaLizedString метод с сигнатурой myLocaLizedString, на самом деле вызывается тело метода localizedString. Что это нам дает в итоге?

Теперь при вызове NSLocalizedString мы перехватываем контроль, не теряя при этом возможности возвращать значение, которое было бы без этих манипуляций. В методе

myLocaLizedString мы можем написать любую логику, получать значение из любого удобного нам хранилища. Делать это лучше для ключей без множественной локализации, чтобы не писать собственную логику ее обработки.

Несмотря на излишнюю сложность и подключение runtime, мы можем достаточно легко поменять логику локализации в большом проекте, в котором все построено на стандартных средствах локализации, а нам нужно сделать логику динамического обновления значений локализации или сделать ее изменение в приложении без его перезагрузки.

Заключение

Мы рассмотрели основные методы локализации компонентов приложения для iOS, доступные для XCode 14. Также рассмотрели решение задач, требующих немного большей гибкости, и привели примеры их решения. Приведенные выше техники применимы как к локализации текстов, так и к локализации любых других ресурсов приложения (картинок, шрифтов и т. п.). Проведенная работа предлагает использовать данный обзор методов локализации и вариантов решения задач, выходящих за рамки стандартного функционала, как отправную точку в разработке более сложных решений.

REFERENCES

1. Localization. URL: https://developer.apple.com/documentation/xcode/localization (дата обращения: 04.12.2023).

2. Customizing the behavior of segue-based presentations. URL: https://developer.apple.com/documentation/uikit/resource_management/customizing_the_behavior_of_segue-based_presentations (дата обращения: 04.12.2023).

3. *Романков С. В.* Технология *auto layout* на платформе IOS // Точная наука. 2022. Выпуск 137.

Romankov S.V. Auto layout technology on the IOS platform. *Tochnaya nauka* [Exact science]. 2022. No. 137. (In Russian)

4. Placing Content in a Bundle. URL: https://developer.apple.com/documentation/bundleresources/placing_content_in_a_bundle (дата обращения: 04.12.2023).

5. Method Swizzling in iOS Development. URL: https://www.innominds.com/blog/method-swizzling-in-ios-development (дата обращения: 04.12.2023).

Информация об авторе

Науменко Александр Евгеньевич, руководитель отдела разработки мобильных приложений в ООО «Дирион»;

346815, Ростовская область, Мясниковский район, х. Красный Крым, ул. Юбилейная, 25а;
naumenko10@yandex.ru, ORCID: https://orcid.org/0009-0000-0264-7949

Information about the author

Aleksander E. Naumenko, Head of Mobile Application Development Department at “Dirion LLC”;
346815, Russia, Rostov region, Myasnikovsky District, Krasny Krym Farmstead, 25a Yubileynaya street;
naumenko10@yandex.ru, ORCID: https://orcid.org/0009-0000-0264-7949