Глава 10

Представление коллекции

В этой главе мы рассмотрим недавнее добавление в библиотеку UIKit, класс `UICollectionView`, и покажем, как он связан с хорошо известным классом `UITableView`, чем отличается от него и как с его помощью делать то, что невозможно при работе с классом `UITableView`.

Многие годы разработчики программ для системы iOS использовали компонент `UITableView` для создания самых разнообразных интерфейсов. Благодаря тому что класс `UITableView` предоставляет программисту возможности определять несколько типов ячеек, создавать их на лету и прокручивать в вертикальном направлении, он стал ключевым компонентом для тысяч приложений. На протяжении долгого времени компания Apple основывала свой подход к созданию интерфейсов прикладного программирования на классе табличного представления, постоянно добавляя новые, более эффективные возможности для его наполнения в новых выпусках системы iOS.

Однако во многих областях обработки данных табличное представление не является окончательным решением. Например, если необходимо представить данные из нескольких столбцов, приходится объединять все столбцы в каждой строке данных в одну ячейку. Кроме того, компонент `UITableView` не позволяет прокручивать свое содержимое в горизонтальном направлении. В целом б*о*льшая часть мощи класса `UITableView` является результатом определенного компромисса: разработчики не имеют контроля над общей схемой табличного представления. Программист может как угодно определить внешний вид любой отдельной ячейки по своему желанию, но в конце дня они просто нагромождаются одна над другой в одном большом списке прокрутки!

Разумеется, компания Apple также все это понимала. В системе iOS 6 был введен новый класс `UICollectionView`, который должен был устранить эти недостатки. Как и табличное представление, этот класс позволяет выводить на экран совокупность ячеек, заполненных данными, и работать с ними, располагая неиспользованные ячейки в очереди для использования в будущем. Однако, в отличие от табличного представления, класс `UICollectionView` не создает из этих ячеек вертикальный стек. На самом деле компонент `UICollectionView` вообще их не представляет на экране! Вместо этого, как вы вскоре убедитесь, он поручает эту работу вспомогательному классу.

Создание проекта DialogViewer

Для того чтобы продемонстрировать некоторые возможности класса `UICollection View`, мы будем использовать его для представления нескольких абзацев текста. Каждое слово будет помещаться в отдельной ячейке, и все ячейки каждого абзаца

будут объединяться в разделы. Каждый раздел также будет иметь свой заголовок. Все это, может быть, не слишком захватывающе, особенно если учесть, что библиотека `UIKit` уже содержит другие превосходные способы для представления текста, но их использование было бы непедагогичным, поскольку мы хотим объяснить вам, насколько гибким является новый класс. На практике вы вряд ли станете делать что-нибудь подобное тому, что изображено на рис. 10.1, с помощью табличного представления!

Для того чтобы решить поставленную задачу, необходимо определить несколько пользовательских классов и воспользоваться классом `UICollectionViewFlowLayout` (единственным вспомогательным классом для вывода данных на экран, включенных в библиотеку UIKit в настоящее время), а затем, как обычно, использовать контроллер представления для того, чтобы связать все это в одно целое. Итак, приступим!

Рис. 10.1. *Каждое слово находится в отдельной ячейке за исключением заголовков. Все они выводятся на экран с помощью компонента* UICollectionView *и не требуют от программиста явных геометрических вычислений*

Откройте в среде Xcode новый проект Single View Application, как мы делали уже много раз. Назовите свой проект `DialogViewer` и используйте стандартные настройки, применяемые повсеместно в этой книге (выберите пункт Swift в списке Language и пункт Universal в списке Devices).

Исправление класса контроллера представления

От делегата приложения ничего особенного не требуется, поэтому сразу переходите к файлу `ViewController.swift` и внесите в него простое изменение, заменив суперкласс на `UICollectionView`:

```
import UIKit

~~class ViewController: UIViewController {~~
class ViewController: UICollectionViewController {
```

Затем откройте файл `Main.storyboard`. Нам нужно настроить контроллер представления, чтобы он соответствовал тому, что мы только что указали в файле `ViewController.swift`. Выберите элемент View Controller в окне Document Outline и удалите его, оставив раскадровку пустой. Найдите в библиотеке объектов элемент

Collection View Controller и перетащите его в область редактирования. Выберите пиктограмму только что перетащенного элемента View Controller и с помощью инспектора идентичности замените его класс на ViewController. В окне инспектора атрибутов убедитесь, что флажок Is Initial View Controller установлен. Далее выберите в окне Document Outline элемент Collection View и с помощью инспектора атрибутов измените его фон на белый. Наконец вы увидите, что объект Collection View в окне Document Outline имеет дочерний объект, который называется Collection View Cell. Это ячейка-прототип, которую можно использовать для разработки макета ваших реальных ячеек в программе Interface Builder. Мы не собираемся делать это в данной главе, так что выделите эту ячейку и удалите ее.

Определение пользовательских ячеек

Теперь определим несколько классов для ячеек. Как показано на рис. 10.1, мы выводим на экран два основных вида ячеек: "обычную", которая содержит слово, и "необычную", которая используется как заголовок. Любая ячейка, которую вы будете использовать совместно с классом UICollectionView, должна быть подклассом системного класса UICollectionViewCell, обеспечивающего основные функциональные возможности, аналогичные возможностям класса UITableViewCell. Эта функциональная возможность включает backgroundView, contentView и т.д. Поскольку две наши ячейки имеют сходные функциональные возможности, мы сделаем одну из них подклассом другой и будем замещать в подклассе некоторые методы.

Начнем с создания нового класса Cocoa Touch в среде Xcode. Назовите новый класс ContentCell и сделайте его подклассом класса UICollectionViewCell. Выберите исходный файл нового класса и добавьте в него объявления трех свойств и заготовку метода класса.

```
class ContentCell: UICollectionViewCell {
    var label: UILabel!
    var text: String!
    var maxWidth: CGFloat!

    class func sizeForContentString(s: String,
            forMaxWidth maxWidth: CGFloat) -> CGSize {
    return CGSizeZero
    }
}
```

Свойство label будет указывать на объект UILabel, используемый для вывода метки на экран. Свойство text будет сообщать ячейке, что именно в ней должно отображаться, свойство maxWidth — управлять максимальной шириной ячейки, а метод sizeForContentString(_,forMaxWidth:), который мы вскоре реализуем, будет использоваться для запроса размера ячейки, чтобы она могла уместить заданную строку на экране. Это удобно при создании и настройке экземпляров классов, описывающих наши ячейки.

Добавьте перекрытия методов `init(frame:)` и `init(coder:)` в `UIView`, как показано ниже.

```
override init(frame: CGRect) {
    super.init(frame: frame)
    label = UILabel(frame: self.contentView.bounds)
    label.opaque = false
    label.backgroundColor =
        UIColor(red: 0.8, green: 0.9, blue: 1.0, alpha: 1.0)
    label.textColor = UIColor.blackColor()
    label.textAlignment = .Center
    label.font = self.dynamicType.defaultFont()
    contentView.addSubview(label)
}

required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}
```

Это довольно простой код. Он создает метку, задает ее свойства для отображения на экране и добавляет ее в объект `contentView`. Единственная хитрость состоит в том, что этот код определяет шрифт для метки с помощью метода `defaultFont()`. Идея заключается в том, что этот класс должен определять, каким шрифтом отображать содержимое, позволяя своим подклассам объявлять собственные шрифты, перекрывая метод `defaultFont()`. Обратите внимание, как он вызывается:

```
label.font = self.dynamicType.defaultFont()
```

Метод `defaultFont()` является методом типа класса `ContentCell`. Для того чтобы вызвать его, как правило, используется имя класса:

```
ContentCell.defaultFont()
```

В данном случае это не сработает, если вызов производится от имени подкласса `ContentCell` (такого, как класс `HeaderCell`, который мы вскоре создадим). На самом деле мы хотим вызвать замещенный метод `defaultFont()` из подкласса. Чтобы сделать это, нам нужна ссылка на объект подкласса. Это именно то, что дает нам выражение `self.dynamicType`. Если это выражение выполняется из экземпляра класса `ContentCell`, оно превращается в объект типа `ContentCell`, и мы будем вызывать метод `defaultFont()` этого класса; но в подклассе `HeaderCell` оно превращается в объект `HeaderCell`, так что будет вызываться метод `defaultFont()` класса `HeaderCell`, а это именно то, что нам нужно.

Мы еще не создали метод `defaultFont()`, так что давайте займемся этим.

```
class func defaultFont() -> UIFont {
    return UIFont.preferredFontForTextStyle(UIFontTextStyleBody)
}
```

Все довольно просто. Здесь, чтобы получить предпочтительный пользовательский шрифт для основного текста, используется метод `preferredFontForTextStyle()` класса `UIFont`. Пользователь может использовать приложение Settings, чтобы изме-

нить размер данного шрифта. С помощью этого метода (вместо жесткого кодирования размера шрифта) мы делаем наши приложения более удобными для пользователей.

Для того чтобы закончить этот класс, реализуем метод, заготовку которого добавили ранее и который вычисляет подходящий размер ячейки:

```
class func sizeForContentString(s: String,
            forMaxWidth maxWidth: CGFloat) -> CGSize {
    let maxSize = CGSizeMake(maxWidth, 1000)
    let opts = NSStringDrawingOptions.UsesLineFragmentOrigin

    let style = NSMutableParagraphStyle()
    style.lineBreakMode = NSLineBreakMode.ByCharWrapping
    let attributes = [ NSFontAttributeName: self.defaultFont(),
            NSParagraphStyleAttributeName: style]

    let string = s as NSString
    let rect = string.boundingRectWithSize(maxSize, options: opts,
            attributes: attributes, context: nil)

    return rect.size
}
```

Этот метод делает много полезного, так что его стоит рассмотреть подробнее. Сначала мы объявляем максимальный размер так, что не допускаются слова, которые могут быть шире, чем значение аргумента `maxWidth`, которое получается из ширины `UICollectionView`. Мы также создаем стиль абзаца, который позволяет переносить слишком большие строки текста на следующие строки абзаца. Мы также создаем словарь атрибутов, который содержит определенный нами для данного класса шрифт по умолчанию и стиль абзаца, который мы только что создали. Наконец, используем функции класса `NSString`, предоставляемого библиотекой UIKit, который позволяет нам рассчитать размеры строки. Мы передаем максимальный размер и другие настроенные нами параметры и атрибуты и получаем размер ячейки.

Осталось правильно задать свойство `text`. Вместо обычной практики, когда мы используем неявную переменную экземпляра, определим методы, которые будут извлекать и задавать значение на основе объекта класса `UILabel`, созданного ранее, используя в качестве хранилища для отображаемого значения объект класса `UILabel`. Благодаря этому мы можем также использовать set-метод для повторного вычисления геометрических размеров ячейки при изменении текста. Заменим определение свойства `text` в файле `ContentCell.swift` следующим кодом:

```
var label: UILabel!
var text: String! {
    get {
        return label.text
    }
    set(newText) {
        label.text = newText
        var newLabelFrame = label.frame
        var newContentFrame = contentView.frame
```

```
        let textSize = self.dynamicType.sizeForContentString(newText,
                    forMaxWidth: maxWidth)
        newLabelFrame.size = textSize
        newContentFrame.size = textSize
        label.frame = newLabelFrame
        contentView.frame = newContentFrame
    }
}
var maxWidth: CGFloat!
```

В get-методе нет ничего особенного, но set-метод выполняет некоторую дополнительную работу. В основном он модифицирует рамку для метки и представления содержимого, ориентируясь на размеры, требуемые для вывода текущей строки.

Это все, что требуется для базового класса ячейки. Теперь создадим класс ячейки для заголовка. Создайте в среде Xcode еще один новый класс на языке Cocoa Touch, назовите его `HeaderCell` и сделайте его подклассом класса `ContentCell`. Заголовочный файл трогать не надо, поэтому перейдите к файлу `HeaderCell.swift` и внесите в него несколько изменений. В этом классе мы заместим несколько методов класса `ContentCell` для изменения внешнего представления ячейки, с тем чтобы оно отличалось от обычного.

```
override init(frame: CGRect) {
    super.init(frame: frame)
    label.backgroundColor = UIColor(red: 0.9, green: 0.9,
            blue: 0.8, alpha: 1.0)
    label.textColor = UIColor.blackColor()
}

required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}

override class func defaultFont() -> UIFont {
    return UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline)
}
```

Теперь ячейка, содержащая заголовок, будет выглядеть иначе, потому что у нее свой цвет и шрифт.

Настройка контроллера представления

Перенесем теперь свое внимание на контроллер представления. Выберите файл `ViewController.swift`, и начните с объявления массива для содержимого, которое мы собираемся выводить.

```
class ViewController: UICollectionViewController {
private var sections: [[String: String]]!
```

Затем мы будем использовать метод viewDidLoad() для создания этих данных. Массив sections будет содержать список словарей, каждый из которых имеет два ключа: header и content. Значения, ассоциированные с этими ключами, будут использованы для определения содержимого экрана. Реальное содержимое адаптировано из хорошо известной игры.

```
override func viewDidLoad() {
    super.viewDidLoad()
    sections = [
    ["header": "First Witch",
     "content" : "Hey, when will the three of us meet up later?"],
    ["header" : "Second Witch",
     "content" : "When everything's straightened out."],
    ["header" : "Third Witch",
     "content" : "That'll be just before sunset."],
    ["header" : "First Witch",
     "content" : "Where?"],
    ["header" : "Second Witch",
     "content" : "The dirt patch."],
    ["header" : "Third Witch",
     "content" : "I guess we'll see Mac there."]
    ]
}
```

Аналогично классу UITableView, класс UICollectionView позволяет зарегистрировать класс повторно используемой ячейки, используя идентификатор. Это позволяет в дальнейшем вызвать метод извлечения из очереди, в которой будет находиться ячейка, и, если ячейка недоступна, представление коллекции создаст ее автоматически. Похоже на класс UITableView! Добавьте следующую строку в конец метода viewDidLoad(), чтобы реализовать эту возможность:

```
collectionView!.registerClass(ContentCell.self,
            forCellWithReuseIdentifier: "CONTENT")
```

Мы внесем еще только одно изменение в viewDidLoad(). Так как это приложение не имеет панели навигации, основное представление будет перекрываться со строкой состояния. Для того чтобы предотвратить это явление, добавьте следующие строки в конец viewDidLoad():

```
var contentInset = collectionView!.contentInset
contentInset.top = 20
collectionView!.contentInset = contentInset
```

Указанной конфигурации метода viewDidLoad() пока достаточно. Прежде чем код станет заполнять представление коллекции, необходимо написать небольшой вспомогательный метод. Все наше содержимое экрана содержится в длинных строках, а мы собираемся выводить слова одно за другим, помещая их в отдельные ячейки. Следовательно, нам нужен внутренний метод для разделения строк на слова. Этот метод будет получать номер раздела, извлекать соответствующие строки из данных указанного раздела и разделять их на слова.

```
func wordsInSection(section: Int) -> [String] {
    let content = sections[section]["content"]
    let spaces = NSCharacterSet.whitespaceAndNewlineCharacterSet()
    let words = content?.componentsSeparatedByCharactersInSet(spaces)
    return words!
}
```

Предоставление содержимого ячеек

Пришло время для группы методов, которые будут заполнять представление коллекции реальными данными. Следующие три метода очень похожи на свои аналоги из класса `UITableView`. Для начала нам нужен метод, сообщающий коллекции, сколько разделов надо вывести на экран.

```
override func numberOfSectionsInCollectionView(
            collectionView: UICollectionView) -> Int {
    return sections.count
}
```

За ним следует метод, сообщающий коллекции, сколько элементов содержится в каждом разделе. Эту задачу решает метод `wordsInSection:`, определенный ранее.

```
override func collectionView(collectionView: UICollectionView,
            numberOfItemsInSection section: Int) -> Int {
    let words = wordsInSection(section)
    return words.count
}
```

Последний метод возвращает отдельную ячейку, предназначенную для хранения одного слова. Этот метод использует метод `wordsInSection()`. Он применяет метод извлечения из очереди к объекту класса `UICollectionView`, аналогично классу `UITableView`. Поскольку мы зарегистрировали класс ячейки для идентификатора, то знаем, что метод извлечения из очереди всегда возвращает экземпляр.

```
override func collectionView(collectionView: UICollectionView,
            cellForItemAtIndexPath indexPath: NSIndexPath)
                -> UICollectionViewCell {
    let words = wordsInSection(indexPath.section)
    let cell = collectionView.dequeueReusableCellWithReuseIdentifier(
            "CONTENT", forIndexPath: indexPath) as ContentCell
    cell.maxWidth = collectionView.bounds.size.width
    cell.text = words[indexPath.row]
    return cell
}
```

Зная, как работает класс `UITableView`, вы можете предположить, что в этот момент приложение уже способно что-то делать. Скомпилируйте и запустите приложение, и вы увидите, что это не так (рис. 10.2).

Мы видим отдельные слова, а не поток слов. Все ячейки имеют одинаковые размеры, и все они прижаты друг к другу. Причина заключается в том, что нам нужно предпринять действия для дополнительной ответственности делегатов.

Рис. 10.2. *Это выглядит не слишком работоспособным…*

Создание потока

До сих пор мы работали с классом `UICollectionView`, но, как уже указывалось, этот класс имеет ассистента, который на самом деле реализует макет. Класс `UICollectionViewFlowLayout`, по умолчанию выполняющий работу по созданию макета для класса `UICollectionView`, имеет несколько методов делегата, с помощью которого он выдает нам информацию. Один из этих методов реализуем прямо сейчас. Объект макета применяет этот метод к каждой ячейке, чтобы определить ее требуемый размер. Здесь мы снова используем метод `wordsInSection()` для получения доступа к требуемому слову, а затем метод, определенный ранее в классе `ContentCell` и определяющий требуемый размер ячейки.

При инициализации `UICollectionViewController` он становится делегатом своего класса `UICollectionView`. `UICollectionViewFlowLayout` представления коллекции будет рассматривать контроллер представления как свой собственный делегат, если он объявит о его соответствии протоколу `UICollectionViewDelegateFlowLayout`. Первое, что нам нужно сделать, — это изменить объявление нашего контроллера представления в файле `ViewController.swift` так, чтобы он объявил о соответствии этому протоколу:

```
class ViewController: UICollectionViewController,
                      UICollectionViewDelegateFlowLayout {
```

Все методы протокола `UICollectionViewDelegateFlowLayout` являются необязательными, и нам нужно реализовать только один из них. Добавьте следующий метод в файл `ViewController.swift`:

```
func collectionView(collectionView: UICollectionView,
            layout collectionViewLayout: UICollectionViewLayout,
            sizeForItemAtIndexPath indexPath: NSIndexPath) -> CGSize {
    let words = wordsInSection(indexPath.section)
    let size = ContentCell.sizeForContentString(words[indexPath.row],
            forMaxWidth: collectionView!.bounds.size.width)
    return size
}
```

Скомпилируйте и запустите приложение снова, и вы увидите, что оно стало много лучше (рис. 10.3).

Теперь ячейки образуют поток, и текст стал более удобочитаемым, причем каждый раздел немного смещен вниз. И все же разделы слишком прижаты друг к другу. Это не слишком красиво. Исправим это, добавив немного параметров конфигурации. Добавьте следующие строки в конец метода `viewDidLoad()`:

```
let layout = collectionView!.collectionViewLayout
let flow = layout as UICollectionViewFlowLayout
flow.sectionInset = UIEdgeInsetsMake(10, 20, 30, 20)
```

Здесь мы извлекаем объект макета из представления коллекции. Сначала присваиваем его временной переменной, тип которой выводится как указатель на объект класса `UICollectionViewLayout`. Мы делаем это в основном для того, чтобы подчеркнуть, что только этот объект знает о существовании обобщенного класса макета, но в действительности он использует экземпляр класса `UICollectionFlowLayout`, который является подклассом класса `UICollectionViewLayout`. Зная истинный тип объекта макета, мы можем использовать операторы приведения типа для присвоения его другой переменной корректного типа и для доступа к методам, которые содержатся только в этом подклассе, — в данном случае нам нужен set-метод для свойства `sectionInset`.

Еще раз скомпилируйте и выполните приложение, и вы увидите, что ячейки равномернее заполняют пространство (рис. 10.4).

Представления заголовка

Осталось вывести на экран объекты заголовков. Класс `UITableView` имеет представления для заголовков и сносок. Их можно было бы использовать для вывода каждого раздела. Класс `UICollectionView` использует эту концепцию в несколько более обобщенном виде, обеспечивая б*о*льшую гибкость макета. Наряду с системой доступа к обычным ячейкам из делегатов существует параллельная система для доступа к дополнительным представлениям, которые можно использовать для заголовков, сносок и других элементов. Добавьте следующий код в конец метода `viewDidLoad()`, чтобы представление коллекции знало о существовании класса ячеек для заголовка:

```
collectionView!.registerClass(HeaderCell.self,
        forSupplementaryViewOfKind: UICollectionElementKindSectionHeader,
        withReuseIdentifier: "HEADER")
```

Как видим, в данном случае мы не только указываем класс ячейки и ее идентификатор, но и вид. Идея заключается в том, что разные макеты могут определять разные виды дополнительных представлений и запрашивать эти представления у делегатов. Класс `UICollectionFlowLayout` будет запрашивать раздел заголовка для каждого раздела представления коллекции.

Рис. 10.3. *Теперь абзацы выделяются*

Рис. 10.4. *Стало куда свободнее*

```
override func collectionView(collectionView: UICollectionView,
                viewForSupplementaryElementOfKind kind: String,
                atIndexPath indexPath: NSIndexPath)
                    -> UICollectionReusableView {
    if (kind == UICollectionElementKindSectionHeader) {
        let cell =
            collectionView.dequeueReusableSupplementaryViewOfKind(
                    kind, withReuseIdentifier: "HEADER",
                    forIndexPath: indexPath) as HeaderCell
        cell.maxWidth = collectionView.bounds.size.width
        cell.text = sections[indexPath.section]["header"]
        return cell
    }
    abort()
}
```

Обратите внимание на вызов `abort()` в конце данного метода. Эта функция заставляет приложение немедленно прекратить работу. Это не та вещь, к которой следует часто прибегать в рабочем коде. Здесь мы ожидаем только вызова для создания ячейки заголовка и ничего не можем сделать, если будет запрошен другой вид

ячеек, — мы даже не можем вернуть nil, потому что возвращаемый тип метода это не допускает. Если этот метод вызван для создания другой разновидности заголовка, то это серьезная ошибка нашей программы или ошибка в библиотеке UIKit.

Скомпилируйте и выполните приложение, и вы увидите... постойте! А где же заголовки? Оказывается, класс `UICollectionFlowLayout` не выделяет для них места, если ему точно не указать их размеры. Вернемся к методу `viewDidLoad()` и добавим в него следующую строку:

```
flow.headerReferenceSize = CGSizeMake(100, 25)
```

Снова скомпилируйте и выполните приложение, и теперь все заголовки окажутся на своих местах, как показано на рис. 10.1 и 10.5.

Рис. 10.5. *Окончательный вид приложения DialogViewer*

В этой главе мы только слегка коснулись класса `UICollectionView` и того, что может быть достигнуто с помощью класса по умолчанию `UICollectionFlowLayout`. С его помощью путем определения собственных классов макетов можно получить еще более красивые приложения, но это тема для другой книги.

Теперь, когда вы знакомы со всеми основными компонентами большой картины, пришло время взглянуть на создание приложений master-detail наподобие приложения iOS Mail. Переворачивайте страницу и приступайте к главе 11.