Окончание табл. 7

Протокол	Описание
`SignedNumberType`	Предназначен для тех типов данных, которые допускают вычитание, отрицание или инициализацию с нуля
`Streamable`	Предназначен для тех типов данных, которые могут быть направлены в поток вывода (например, данные типа `String`, `Character` или `UnicodeScalar`)
`Strideable`	Предназначен для типов данных, допускающих представление непрерывной последовательности значений, которые могут быть смещены и отсчитаны

Если вас интересует реализация этих протоколов или исследование любых других языковых средств Swift, откройте игровую площадку или любой исходный файл Swift и найдите (или просто наберите) имя **`println`** глобальной функции. Далее нажмите комбинацию клавиш <Command+Option> и щелкните кнопкой мыши на имени этой функции. В итоге откроется помощник редактора с эквивалентом заголовочного файла, где определены и задокументированы многие языковые средства Swift. Все это нужно сделать, чтобы выяснить, какие именно языковые средства должны быть введены в ваш собственный класс, чтобы он соответствовал каждому протоколу.

Управление памятью

Как и в языке Objective-C, в языке Swift применяется *подсчет ссылок* в качестве основного механизма для отслеживания моментов, когда динамически выделяемая оперативная память больше не используется и может быть освобождения для других ресурсов.

В течение многих лет (да и теперь) в языке Objective-C по-прежнему применяется ручной подсчет ссылок, но для этого от разработчика требуется прилежание. И хотя этим приемом можно овладеть, начинающим программировать на Objective-C трудно сразу уяснить, когда следует, а когда не

следует применять некоторые методы, связанные с подсчетом ссылок.

В результате непродолжительного экспериментирования со сборщиком "мусора" в Objective-C для автоматического управления памятью, в 2011 году компания Apple объявила о разработке механизма ARC (Automatic Reference Counting — автоматический подсчет ссылок). В этом механизме применяется тот же самый подход, что и при ручном подсчете ссылок, но только более строгим и детерминированным образом.

Принцип действия подсчета ссылок

В основу механизма подсчета ссылок положен довольно простой принцип. Каждый объект, т.е. каждый экземпляр класса, имеет встроенное свойство для *подсчета ссылок*, в котором устанавливается значение **1**, когда получается экземпляр объекта.

Всякий раз, когда во фрагменте кода требуется выразить интерес к объекту или овладеть им, т.е. когда создается указатель на объект, необходимо инкрементировать счетчик ссылок. А когда во фрагменте кода завершается обработка объекта и он больше не представляет никакого интереса, т.е. когда указатель на объект больше не нужен, необходимо декрементировать счетчик ссылок. В языке Objective-C интерес к объекту выражается посредством вызова метода `retain()`, а последующая потеря интереса к объекту — посредством вызова метода `release()` для данного объекта. Когда же счетчик ссылок на объект декрементируется до нуля, это означает, что текущие ссылки на объект отсутствуют и объект может быть уничтожен, а выделенная для него оперативная память — освобождена.

Если получение экземпляра и освобождение объекта осуществляется в одной функции, овладеть механизмом подсчета ссылок совсем не трудно. Сложности подсчета ссылок проявляются лишь после разрыва соединения между операциями

получения экземпляра и его освобождения программно или во время выполнения. На данной стадии начинающий программист может относительно легко допустить ошибку, не освободив объект, на который он создал ссылку, что приведет к утечке памяти, или же освободив объект, которым он фактически не владеет, что может привести к аварийному сбою.

Механизм ARC управляет всем процессом подсчета ссылок, автоматически определяя места для ввода вызова на сохранение или освобождение объектов. Следовательно, этот механизм освобождает от обязанности делать это.

Циклы сохранения ссылок и строгие ссылки

Один из главных недостатков подхода к управлению памятью путем подсчета ссылок заключается в *циклах сохранения ссылок*. В простейшем виде эти циклы происходят, когда два объекта содержат *строгие ссылки* друг на друга. В качестве примера рассмотрим следующий фрагмент кода, где в переменной `a` сохраняется вновь созданный экземпляр класса `A`:

```
class A { }
var a = A()
```

Строгие ссылки создаются также по умолчанию, когда объекты сохраняют ссылки друг на друга. В следующем примере код сокращен до минимума, но подобная ситуация возникает во многих местах, где применяются сложные взаимосвязанные структуры данных:

```
class A
{
  var otherObject: B?
}

class B
{
  var otherObject: A?
}

var a = A() // число ссылок на новый экземпляр класса A
```

```
            // устанавливается равным 1
var b = B() // число ссылок на новый экземпляр класса B
            // устанавливается равным 1
a.otherObject = b
// число ссылок на экземпляр класса B инкрементируется
// до 2
b.otherObject = a
// число ссылок на экземпляр класса A инкрементируется
// до 2
```

В приведенном выше примере кода демонстрируются два экземпляра, a и b, которые также ссылаются друг на друга. После выполнения данного фрагмента кода число ссылок на каждый из этих экземпляров увеличивается до **2**.

Когда часть кода, где получаются экземпляры этих двух классов, выходит из области своего действия, локальные переменные a и b также выходят из области своего действия и удаляются. В частности, когда удаляется переменная a, число ссылок на тот экземпляр, на который она ссылается, декрементируется на **1**. Но поскольку оно не достигло нуля, то сам экземпляр не удаляется. Аналогично, когда удаляется переменная b, число ссылок на тот экземпляр, на который она ссылается, также декрементируется на **1**. Но и в этом случае экземпляр не удаляется, поскольку это число не достигло нуля.

В итоге получаются два экземпляра (один — класса A, другой — класса B), ссылающиеся друг на друга, а следовательно, они поддерживают друг друга в активном состоянии. А поскольку число ссылок на оба экземпляра вообще не достигает нуля, то ни один из объектов не может быть удален и оба попрежнему занимают память.

Подобная ситуация называется *утечкой памяти*. И если она повторяется во время выполнения программы, то объем памяти, выделяемой программе, будет постоянно сокращаться, что может отрицательно сказаться на производительности или в конечном итоге привести к аварийному завершению программы в операционной системе, если она ограничит объем памяти, требующийся данной программе.

Если опытные программисты еще могут принять во внимание циклы сохранения ссылок, управляя памятью вручную, то механизм ARC не в состоянии предотвратить их без некоторой помощи со стороны программиста. По существу, если механизм ARC служит для автоматического управления памятью, ему требуются дополнительные сведения о характере ссылок на другие объекты, чтобы разделить их на категории *слабых* и *ничейных* ссылок.

Слабые ссылки

Чтобы предотвратить циклы сохранения ссылок, можно, в частности, заменить одну из строгих ссылок на слабую. Для этого достаточно предварить ключевым словом `weak` объявление переменной с помощью ключевого слова `var`, как демонстрируется в следующем примере кода:

```
class A
{
  var otherObject: B?
}

class B
{
  weak var otherObject: A?
}
```

Слабые ссылки оказывают следующие воздействия.

- Нет никакой уверенности в том, что ссылающийся элемент "владеет" тем экземпляром, на который он ссылается, и ему приходится мириться с тем, что экземпляр может исчезнуть. На практике это означает, что когда устанавливается слабая ссылка, число ссылок на объект не инкрементируется.
- Когда освобождается экземпляр, на который делается слабая ссылка, механизм ARC устанавливает в ней пустое значение (`nil`). Следовательно, слабые ссылки должны объявляться как переменные, а не как константы.

Итак, установив слабую ссылку, рассмотрим следующий пример кода, где демонстрируется изменение числа ссылок на объекты по мере выполнения кода:

```
var a = A() // число ссылок на новый экземпляр класса A
            // устанавливается равным 1
var b = B() // число ссылок на новый экземпляр класса B
            // устанавливается равным 1
a.otherObject = b
// число ссылок на экземпляр класса B инкрементируется
// до 2
b.otherObject = a
// число ссылок на экземпляр класса A остается равным 1
```

Ссылка на экземпляр класса `A`, которая удерживается экземпляром класса `B`, является слабой. Поэтому число ссылок на экземпляр класса `A` остается равным **1**.

Как и прежде, когда локальные переменные `a` и `b` выходят из своей области действия и удаляются, число ссылок на те экземпляры, на которые они ссылаются, декрементируется. Это означает, что число ссылок на экземпляр класса `A` сокращается до нуля, а число ссылок на экземпляр класса `B` — до **1**.

Теперь число ссылок на экземпляр класса `A` достигло нуля, и хранить его в памяти больше не требуется. Поэтому начинается процесс восстановления и освобождения памяти от этого экземпляра. Во время освобождения памяти происходит следующее.

- В слабой ссылке на экземпляр класса `A`, удерживаемой в экземпляре класса `B`, устанавливается пустое значение (`nil`).
- В освобождаемом из памяти экземпляре класса `A` удерживается строгая ссылка на экземпляр класса `B`, и поэтому в ходе данного процесса экземпляру класса `B` посылается сообщение об освобождении памяти, а число его ссылок на экземпляр класса `A` декрементируется до нуля.

В данный момент экземпляр класса `A` удаляется из памяти, где остается только экземпляр класса `B` с нулевым числом

ссылок. Поэтому начинается процесс восстановления и освобождения памяти от этого экземпляра.

В отношении слабых ссылок необходимо иметь в виду следующее.

- Слабые ссылки следует применять в том случае, если в прикладном коде и модели данных допускается наличие пустых ссылок во время выполнения программы.
- Слабые ссылки следует всегда определять как необязательные.

Ничейные ссылки

Ничейная ссылка подобна слабой ссылке в том отношении, что нет никакой уверенности, что ссылающийся элемент "владеет" тем экземпляром, на который он ссылается. И когда устанавливается ничейная ссылка, число ссылок на экземпляр не инкрементируется.

А главное отличие ничейной ссылки от слабой состоит в том, что как только ничейная ссылка будет установлена, она должна всегда иметь конкретное значение, тогда как для слабой ссылки иногда допускается пустое значение (`nil`). Чтобы определить ничейную ссылку, ее объявление с помощью ключевого слова `var` достаточно предварить ключевым словом `unowned`, как показано ниже.

```
class B
{
  unowned var otherObject: A
}
```

В отношении ничейных ссылок необходимо иметь в виду следующее.

- Ничейные ссылки следует применять в том случае, если в прикладном коде и модели данных предполагается, что созданная однажды ссылка будет существовать всегда, оставаясь достоверной, — по крайней мере, до тех

пор, пока ссылающийся элемент не выйдет из своей области действия или не будет удален.

- Ничейные ссылки следует всегда определять как не являющиеся необязательными.
- Если попытаться получить доступ к экземпляру, который был освобожден из памяти по ничейной ссылке, программа на Swift завершится аварийно с выдачей ошибки времени выполнения.

Циклы сохранения ссылок и замыкания

Подобно классам, замыкания фактически относятся к ссылочным типам. Так, если присвоить замыкание свойству экземпляра и захватить в нем этот экземпляр, сделав ссылку на его свойство или вызвав для него метод, в конечном итоге образуется цикл сохранения ссылок между данным экземпляром и замыканием.

В качестве выхода из этого положения в замыкании можно воспользоваться слабой или ничейной ссылкой на захватываемый экземпляр или метод. Но, как пояснялось ранее, синтаксис для определения этих ссылок разный. Вместо этого ссылки можно указать в виде *списка захвата* в определении замыкания.

Список захвата указывается в определении замыкания непосредственно перед списком параметров или перед ключевым словом `in`, если у замыкания отсутствуют параметры. Список захвата состоит из одного или нескольких разделяемых запятыми типов ссылок (`unowned` или `weak`), после которых следует имя свойства или метода, на которые делается ссылка, как показано в следующей общей форме:

```
{
  [ТипСсылки свойство_или_метод [, ...] ]
  (параметры) -> возвращаемый_тип in
    операторы
}
```

Например, определение замыкания, сохраняемого в свойстве `aClosure` и обращающегося по ссылке `self`, будет выглядеть следующим образом:

```
var aClosure: (параметры) -> возвращаемый_тип =
{
  [unowned self]
  (параметры) -> возвращаемый_тип in
    операторы
}
```

Правила, по которым следует применять конкретный тип ссылки, остаются такими же, как и для ссылок одних экземпляров на другие. В частности, слабую и определяемую как необязательную ссылку типа `weak` следует применять в том случае, если ссылка может достоверно стать пустой (`nil`). А ничейную ссылку типа `unowned` следует применять в том случае, если замыкание и экземпляр, который оно захватывает, ссылаются друг на друга, а ссылка остается достоверной до тех пор, пока оба объекта не освобождаются из памяти.

Обобщения

Обобщения как языковые средства предоставляют программирующим на Swift возможность писать обобщенный код для обработки данных любого типа. Аналогичные средства имеются в языках C++ (в виде *шаблонов*) и C# (в виде *обобщений*). Часть стандартной библиотеки Swift реализуется в виде обобщений. Например, типы `Array` и `Dictionary` относятся к обобщенным коллекциям, в которых можно хранить данные любого типа. Обобщенный код можно писать на Swift самыми разными способами, включая обобщенные функции, типы и протоколы.