9

Работа со списками свойств

Списки свойств представляют собой способ сохранения структурированных данных. В списках свойств можно хранить стандартные классы коллекций (массивы и словари) том числе символьные строки, числа, даты и т.д.

Списки свойств подразумеваются как абстрактные. Они не связаны с конкретным языком программирования или представлением. Для упорядочения в последовательной форме (или так называемой сериализации) списков свойств в настоящее время определены три формата. Самым старым считается формат списков свойств в прикладном интерфейсе OpenStep. Это очень компактный, удобочитаемый формат.

К сожалению, в Mac OS X нельзя записывать данные в списки свойств старого формата. Их можно прочитать, но в спецификации не определено, как сохранять в них даты или другие элементы данных, и поэтому записать их в подобные списки не удастся. Формат OpenStep списков свойств был расширен в проекте GNUstep и позволяет сохранять в них все, что поддерживается в более новых форматах.

В Mac OS X компания Apple внедрила формат XML и двоичный формат списков свойств. Формат XML оказался невероятно многословным, но он обладает тем преимуществом, что может быть подвергнут синтаксическому анализу другими совместимыми с XML инструментальными средствами и встроен в другие XML-документы.

Двоичный формат списков свойств очень быстро подвергается синтаксическому анализу и довольно компактный. Но поскольку он двоичный, то неудобочитаем.

В некоторых библиотеках других языков программирования имеются средства для обработки списков свойств. Так, библиотека WINGS в диспетчере окон WindowMaker позволяет читать и записывать в них данные, как, впрочем, и библиотека, входящая в состав ОС NetBSD. В среде Core Foundation компании Apple и ее открытой библиотеке CFLite также можно пользоваться списками свойств, но из кода C.

Сохранение коллекций в списках свойств

```
 6  NSArray *array = [NSArray arrayWithObjects:
 7    @"array", @"containing", @"string",
 8    @"objects", nil];
 9  [array writeToFile: @"example.plist" atomically: NO];
10  NSMutableArray *cycle = [array mutableCopy];
11  [cycle addObject: cycle];
12  [cycle writeToFile: @"failure.plist" atomically: NO];
```

Пример кода из файла `writeplist.m`

Классы массивов и словарей реализуют метод `-writeToFile:atomically:`. Это позволяет очень просто вывести содержимое коллекции в файл, используя выбираемый по умолчанию формат списка свойств.

Подобные методы не делают ничего сверхъестественного. Они будут действовать только в том случае, если в коллекциях не содержатся данные ни одного их тех типов, которые нельзя сохранить в списке свойств.

Если передать упомянутому выше методу логическое значение `YES` в качестве второго параметра, он обеспечит постоянство и согласованность представления данных на диске. Этот метод сначала запишет список свойств во временный файл, а затем переименует этот файл по завершении записи.

При выполнении кода из примера, приведенного в начале этого раздела, будет сформирован файл example.plist, содержащий массив в качестве корневого элемента с четырьмя символьными строками в нем.

```
<?xml version="1.0" encoding="UTF8"?>
<!DOCTYPE plist PUBLIC "//Apple//DTD PLIST 1.0//
    EN" "http://www.apple.com/DTDs/PropertyList
    -1.0.dtd">
<plist version="1.0">
<array>
  <string>array</string>
  <string>containing</string>
  <string>string</string>
  <string>objects</string>
</array>
</plist>
```

Пример списка свойств из файла example.plist

В текущих версиях Mac OS X подобным способом списки свойств используются в формате XML. Как следует из приведенного выше примера, формат XML списков свойств оказывается слишком многословным. В старом формате OpenStep списков свойств массивы заключались в квадратные скобки, а символьные строки — в кавычки, и поэтому списки свойств получались компактными менее чем наполовину. Если же преобразовать список свойств из приведенного выше примера в двоичный формат, то размер его файла сократится с 294 до 82 байтов.

Для такого компактного списка свойств сокращение занимаемого им места на диске уже не имеет особого значения. Списки свойств в обоих упомянутых выше форматах, скорее всего, вместятся в одной единице размещения (так называемом кластере) на диске, и поэтому они займут одинаковое пространство на диске. А для синтаксического анализа списков свойств в обоих форматах не потребуется много времени.

Но этого нельзя сказать о более крупных списках свойств. Например, в браузере Safari предыстория посещений веб-страниц сохраняется в двоичном списке свойств. На момент написания этой книги файл такого списка на моем компьютере занимал 15 Мбайт. Если преобразовать его в формат XML, размер его файла возрастет до 34 Мбайт. В связи с этим увеличиваются также издержки вычислительных ресурсов (времени ЦП и объема оперативной памяти) на синтаксический анализ крупных списков свойств.

К сожалению, в классах коллекций не поддерживается вывод коллекций в списки свойств ни в одном из форматов, кроме XML. Поэтому приходится обращаться к классу `NSPropertyListSerialization`, рассматриваемому далее в этой главе.

Все форматы списков свойств имеют иерархическую структуру. Они не поддерживают запись циклических структур данных. Второй массив в приведенном выше примере сохранения коллекции в списке свойств содержит указатель на самого себя. Поведение кода при записи такого массива в список свойств не определено. В Mac OS X, например, в файле списка свойств сохраняются только те элементы, которые находятся перед рекурсивным указателем, а все следующие за ним элементы просто игнорируются.

Чтение данных из списков свойств

```
 6  NSArray *array =
 7  [NSArray arrayWithContentsOfFile:
 8  @"example.plist"];
 9  NSData *data = [NSData dataWithContentsOfFile:
10    @"example.plist"];
11  NSMutableArray *mutable =
12    [NSPropertyListSerialization
13      propertyListFromData: data
14        mutabilityOption:
15    NSPropertyListMutableContainersAndLeaves
16                    format: NULL
```

```
17          errorDescription: NULL];
18  NSCAssert([mutable isKindOfClass:
19    [NSMutableArray class]],
20      @"Should have read a mutable array");
21  [[mutable objectAtIndex: 0]
22  appendString: @"suffix"];
```

Пример кода из файла `readplist.m`

Данные из списка свойств могут быть прочитаны двумя способами. Прежде всего, можно воспользоваться обратным вариантом рассмотренного выше метода сохранения данных в списке свойств. В обоих классах коллекций, поддерживающих сериализацию списков свойств, имеется метод инициализации `-initWithContentsOfFile:`, а также соответствующий конструктор.

На первый взгляд, реализовать такой способ совсем не трудно, но по ходу дела возникает ряд осложнений. И наиболее существенное из них состоит в том, что в списках свойств не сохраняются никакие сведения об изменчивости объектов. Например, объекты типа `NSString` и `NSMutableString` сохраняются совершенно одинаково.

При чтении этих объектов обратно из списка свойств возникает вопрос: какой же из них будет получен на самом деле? Так, если прочитать из списка свойств массив, отправив сообщение `+arrayWithContentsOfFile:` объекту типа `NSArray`, то в конечном итоге будет получен неизменяемый массив. А если отправить то же самое сообщение объекту типа `NSMutableArray`, то будет получен изменяемый массив.

Такой способ пригоден для чтения внешнего элемента из списка свойств, но что, если требуется прочитать объекты, находящиеся в массиве? В обоих случаях будут получены массивы, заполненные неизменяемыми объектами. Но ведь это может оказаться совсем не тем, что нужно.

Если требуется прочитать изменяемые объекты обратно из списка свойств, для этой цели следует воспользоваться

классом NSPropertyListSerialization. В этом классе организуется чтение и запись объектов в списки свойств и обеспечивается более скрупулезный контроль над этими процессами, чем в различных классах коллекций.

На заметку

В самой свежей документации на версию Objective-C от компании Apple можно обнаружить, что метод, вызываемый для объекта типа NSPropertyListSerialization в примере кода, приведенном в начале этого раздела, отмечен как не рекомендуемый в будущем к употреблению. Дело в том, что он был внедрен в то время, когда в методах версии Objective-C от компании Apple начали применяться параметры error:, но прежде, чем был внедрен класс NSError. К сожалению, метод, рекомендуемый в качестве замены, пока еще не обеспечивает аналогичные функциональные возможности.

Организуя чтение объектов из списка свойств средствами этого класса, можно передать в качестве параметра режим изменчивости объектов. Для этого имеются три возможности. По умолчанию выбирается режим, делающий все объекты неизменяемыми. В примере кода, приведенном в начале этого раздела, выбран совершенно другой режим, делающий все объекты изменяемыми. На это указывает суффикс, присоединяемый к одной из символьных строк в массиве.

Третий режим представляет собой нечто среднее между двумя предыдущими. В этом режиме контейнерные объекты становятся изменяемыми, а остальные объекты (символьные строки, числа, даты и т.д.) — неизменяемыми. Этот режим оказывается удобным в том случае, если требуется видоизменить древовидную структуру списка свойств, но не элементы на ее вершинах, т.е. листьях. Если объекты читаются из списка свойств подобным способом, то класс сериализации этого списка сообщает его формат.

Преобразование форматов списков свойств

```
 7  NSString *file =
 8    [NSString stringWithUTF8String: argv[1]];
 9  NSData *data = [NSData dataWithContentsOfFile:
10    file];
11  NSPropertyListFormat fmt;
12  id plist = [NSPropertyListSerialization
13    propertyListWithData: data
14                 options: 0
15                   format: &fmt
16                     error: NULL];
17  if (fmt == NSPropertyListBinaryFormat_v1_0)
18  {
19    return 0;
20  }
21  data = [NSPropertyListSerialization
22    dataWithPropertyList: plist
23                  format: NSPropertyListBinaryFormat_v1_0
24                 options: 0
25                   error: NULL];
26
27 [data writeToFile: file atomically: NO];
```

Пример кода из файла: `makebinaryplist.m`

В Mac OS X нельзя записывать данные в списки свойств формата OpenStep, но все-таки можно воспользоваться двумя другими форматами. Как правило, для сохранения закрытых данных следует пользоваться двоичным форматом, а для сохранения всех остальных данных, которые пользователю, возможно, потребуется отредактировать другими инструментальными средствами, — форматом XML.

Впрочем, эти правила не являются раз и навсегда установленными. Назначение формата списка свойств состоит в том, чтобы преобразовать его в другое представление без потерь. Поэтому и рекомендуется пользоваться преимущественно двоичным форматом. Ведь он более компактный, обеспечивает ускоренный синтаксический анализ, а также возможность

для пользователя преобразовать список свойств в более удобный формат, если его потребуется отредактировать.

Для выбора формата списка свойств при его записи в файл служит класс `NSPropertyListSerialization`. Этот класс любезно уведомляет также о текущем формате списка свойств при его чтении.

На заметку

Методы, применяемые в примере кода, приведенном в начале этого раздела, были внедрены в Mac OS X 10.6. Если вы пользуетесь более старой версией Mac OS X, то метод чтения из списка свойств можете заменить рассмотренным в предыдущем разделе, а метод записи в список свойств — соответствующим методом из раздела, предшествовавшего предыдущему.

Нередко списки свойств приходится сохранять в двоичном формате ради повышения эффективности. Если же пользователям прикладной программы потребуется загрузить файлы списков свойств в текстовый редактор для последующего редактирования, они могут воспользоваться утилитой `plutil`, чтобы преобразовать списки свойств в формат XML. Но они могут, конечно, забыть преобразовать их обратно в двоичный формат, и поэтому в следующий раз прикладная программа прочитает их медленно.

Если же видоизменение списка свойств не предполагается при каждом запуске прикладной программы на выполнение, то в этот момент целесообразно сначала проверить, является ли его формат по-прежнему двоичным, а затем перезаписать его в этом формате, если он окажется в другом формате.

Применение формата JSON

```
22  NSData *jsonData =
23    [json dataUsingEncoding: NSUTF8StringEncoding];
24  NSError *e;
25  id object =
```

```
26  [NSJSONSerialization JSONObjectWithData: jsonData
27                   options: 0
28                     error: &e];
29  NSCAssert(nil == e, @"Failed to parse JSON");
30  NSData *data =
31    [NSJSONSerialization dataWithJSONObject: object
32                   options: NSJSONWritingPrettyPrinted
33                     error: &e];
34  NSCAssert(nil == e, @"Failed to export JSON");
```

Пример кода из файла `json.m`

За последние несколько лет широкое распространение получил еще один формат, очень похожий на старый формат OpenStep списков свойств. Это формат *JSON* (JavaScript Object Notation — Представление объектов JavaScript). Он является подмножеством упорядоченных в двоичной форме (сериализованных) объектов JavaScript, хранящих данные, но не код.

В последние версии среды Cocoa включен класс, смоделированный на основе класса `NSPropertyListSerialization` для загрузки и сохранения данных в формате JSON. Такому способу сохранения и загрузки данных присуще больше ограничений, чем кодировке списков свойств. В частности, числа или символьные строки можно хранить только в листьях древовидной структуры, а все коллекции должны быть массивами или словарями.

Как правило, пользоваться форматом JSON для приложений, функционирующих только в среде Objective-C, не приходится. Этот формат оказывается удобным, главным образом, для обеспечения стыкуемости программных средств. На многих платформах имеются библиотеки и языки программирования, способные читать и записывать данные в формате JSON, а для большинства веб-приложений он является собственным форматом. Если данные записываются в формате JSON, то в коде JavaScript веб-страницы на стороне клиента нетрудно преобразовать в нужную форму. Справедливо и обратное: можно сформировать данные

в формате JSON на веб-странице и затем воспользоваться ими в коде на Objective-C.

В примере кода, приведенном в начале этого раздела, показано, каким образом осуществляется преобразование данных в формат JSON, и наоборот. В соответствии со стандартом JSON данные должны быть представлены в некоторой кодировке уникода. Класс сериализации способен автоматически распознать эту кодировку, поскольку два первых символа в потоке данных формата JSON всегда представлены в коде ASCII. Это дает возможность обнаруживать многобайтовые кодировки, размещая на месте первых четырех байтов пустые байты NULL, если метка порядка следования байтов отсутствует.

В рассматриваемом здесь примере осуществляется структурная распечатка программы в формате JSON, что полезно, главным образом, для отладки. Выводимый на печать исходный код программы автоматически форматируется с отступами для удобства его чтения. Для этого требуется немного больше времени ЦП и объема оперативной памяти, и поэтому передавать данные в таком формате по сети или применять их в другой программе нецелесообразно, но имеет смысл для правки данных вручную.

Сохранение пользовательских настроек по умолчанию

```
 6  NSUserDefaults *def =
 7    [NSUserDefaults standardUserDefaults];
 8  id persistentString =
 9    [def stringForKey: @"example"];
10  NSLog(@"Old value: %@", persistentString);
11  if (argc > 1)
12  {
13    NSString *new =
14      [NSString stringWithUTF8String:
15        argv[1]];
```

```
16    [def setObject: new
17            forKey: @"example"];
18    [def synchronize];
19  }
```

Пример кода из файла `defaults.m`

Система пользовательских настроек по умолчанию считается одним из самых крупных потребителей списков свойств. Если заглянуть в каталог `~/Library/Preferences`, то в нем можно обнаружить немало файлов со списками свойств. В этих файлах хранятся глобальные параметры настройки системы текущего пользователя. Соответствующий глобальный каталог располагается в каталоге `/Library/Preferences`.

Для каждого приложения выделяется своя *область настроек по умолчанию*, которая, как правило, обозначается строкой в обратном порядке по отношению к форме записи доменных имен DNS (например, `com.apple.TextEdit`). Доступ к данным из этой области может быть осуществлен средствами класса `NSUserDefaults`.

Система пользовательских настроек по умолчанию может читать данные из самых разных источников. К числу самых полезных источников относится командная строка. Пользовательские настройки можно указать парами "ключ–значение" в качестве аргументов командной строки любого приложения для Mac OS X, что очень удобно для целей отладки.

Система пользовательских настроек по умолчанию доступна в виде единого словаря с несколькими служебными методами. В общем, объект пользовательских настроек по умолчанию можно себе представить как постоянный словарь.

На заметку

Систему пользовательских настроек по умолчанию можно сравнить с реестром Windows, но с учетом нескольких важных отличий. В обоих поддерживается несколько уровней ключей (пользовательских, систем-

ных и сетевых), но все они скрыты от программиста в системе пользовательских настроек по умолчанию. И хотя оба имеют форму древовидной структуры, реестр представляет собой единую базу данных, тогда как система пользовательских настроек по умолчанию хранит данные в отдельных файлах для каждой области. Благодаря этому упрощается процесс внесения изменений в пользовательские настройки по умолчанию инструментальными средствами независимых производителей. А кроме того, система пользовательских настроек по умолчанию масштабируется лучше, поскольку она не требует загрузки больше двух файлов в оперативную память одновременно. К числу недостатков такой системы относится то обстоятельство, что операции с настройками по умолчанию не носят характер транзакций. Внесение изменений в одной и той же области одновременно из двух программ не определено.

В отличие от словаря, у объекта пользовательских настроек по умолчанию имеется несколько служебных методов для доступа к необъектным типам данных и контроля объектных типов. В примере кода, приведенном в начале этого раздела, этому объекту отправляется сообщение `-stringForKey:`. При этом в оболочку заключается метод `-objectForKey:`, но гарантируется, что возвращаемое значение окажется строковым.

Имеются и другие служебные методы, в том числе `-floatForKey:` и `-setFloat:ForKey:`, принимающие примитивные типы C и заключающие их в оболочку экземпляров класса `NSNumber` перед сохранением в системе пользовательских настроек по умолчанию.

Код из примера, приведенного в начале этого раздела, завершается отправкой сообщения `-synchronize` объекту настроек по умолчанию, но зачастую в этом нет особой необходимости. Ведь объект настроек по умолчанию периодически синхронизируется с хранилищем на диске. Но в данном примере цикл исполнения не применяется, и поэтому таймеры не действуют, а выполнение кода завершается сразу же после внесения изменений в настройки по умолчанию. Следовательно, внесенные изменения не будут вообще зафиксированы на диске без отправки этого сообщения.

Если выполнить код из рассматриваемого здесь примера, то можно обнаружить, что начальное значение равно `nil`, поскольку в настройках по умолчанию пока еще ничего не хранится. Значение, передаваемое короткой программе в качестве аргумента, сохраняется в настройках по умолчанию и выбирается автоматически при последующем ее запуске на выполнение.

Это положение можно продемонстрировать и средствами класса `NSArgumentDomain`. Для настроек по умолчанию имеется несколько источников, загружаемых по порядку и перезаписываемых предыдущие. Последними и наиболее высокоприоритетными настройками по умолчанию являются аргументы. Так, если указать аргумент `-example` в командной строке, то в настройках по умолчанию будет задан ключ `@"example"` для любого значения, передаваемого в качестве аргумента.

Но это не заменяет постоянное значение, а лишь переопределяет его при данном конкретном запуске программы на выполнение. В этом можно убедиться, если выполнить программу еще раз, но без указания упомянутого выше аргумента в командной строке. В итоге будет использовано старое значение, установленное два вызова программы назад. Оно было зафиксировано на диске при третьем запуске программы на выполнение, но не использовалось при четвертом ее запуске, поскольку аргумент был заменен. А при пятом запуске без аргумента `-example` это значение появится снова. Отдельные области настроек по умолчанию могут быть просмотрены средствами класса `NSUserDefaults`, хотя потребность в этом возникает редко.

```
1  $ ./a.out persistent
2  a.out[72216:903] Old value: (nil)
3  $ ./a.out new
4  a.out[72216:903] Old value: persistent
5  $ ./a.out example arg store
6  a.out[72362:903] Old value: arg
```

```
$ ./a.out example arg
a.out[72363:903] Old value: arg
$ ./a.out new
a.out[72363:903] Old value: store
$ plutil convertxml1 \
~/Library/Preferences/a.out.plist
$ cat ~/Library/Preferences/a.out.plist
<?xml version="1.0" encoding="UTF8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//
     EN" "http://www.apple.com/DTDs/PropertyList
     -1.0.dtd">
<plist version="1.0">
<dict>
  <key>example</key>
  <string>new</string>
</dict>
</plist>
```

Результат выполнения кода из файла `defaults.m`

При разработке инструментального средства, работающего в режиме командной строки, настройки по умолчанию оказываются очень удобным средством для синтаксического анализа аргументов командной строки. Исходные значения аргументов командной строки можно хранить в системе пользовательских настроек по умолчанию, предоставив пользователю возможность переопределять их автоматически, не прибегая к синтаксическому анализу кода.

Сохранение произвольных объектов в пользовательских настройках по умолчанию

```
NSUserDefaults *def =
  [NSUserDefaults standardUserDefaults];
NSData *serialized = [def objectForKey: @"color"];

id favoriteColor =
[NSKeyedUnarchiver unarchiveObjectWithData:
```

```
12    serialized];
13
14  if (nil == favoriteColor)
15  {
16    favoriteColor = [NSColor blackColor];
17    serialized = [NSKeyedArchiver
18      archivedDataWithRootObject: favoriteColor];
19    [def setObject: serialized
20            forKey: @"color"];
21    [def synchronize];
22  }
```

Пример кода из файла: `colordefault.m`

Очень часто в настройках по умолчанию требуется сохранить нечто, отличающееся от данных основных типов, поддерживаемых в среде Foundation. Характерным тому примером служат цвета. Если предоставить пользователям возможность настраивать каким-то образом интерфейс прикладной программы, то, как правило, придется сохранять выбранные ими цвета в промежутках между последовательными вызовами программы. И сделать это, на первый взгляд, совсем не трудно: достаточно сохранить объект типа `NSColor` в настройках по умолчанию.

В самом деле, это было бы нетрудно, если бы в списках свойств и расширяющих их пользовательских настройках по умолчанию поддерживались объекты типа `NSColor`. К сожалению, такая поддержка отсутствует. Но в то же время в пользовательских настройках по умолчанию можно хранить экземпляры класса `NSData`. И если иметь в своем распоряжении какой-нибудь механизм преобразования объекта в данные, то этот объект можно сохранить в пользовательских настройках по умолчанию.

Для этой цели, к счастью, имеется общий механизм, поддерживаемый большинством наиболее распространенных объектов. Любой объект, реализующий протокол

NSCoding, может быть подвергнут сериализации и десериализации средствами класса NSCoder.

В отличие от списков свойств, механизм, реализуемый в классе NSCoder, поддерживает произвольные объекты и циклы сохранения. К сожалению, для этого требуется реализовать в объекте два метода из протокола NSCoding, что подходит не для всех, но все же для большинства объектов. Но благодаря этому нетрудно организовать сохранение объектов в настройках по умолчанию.

Если операцию сохранения цветов приходится выполнять часто, то нетрудно создать *категорию*, вводящую в класс NSUserDefaults методы -colorForKey: и -setColor:forKey:. Эти методы могут принимать экземпляр класса NSColor в качестве своего параметра, подвергать его сериализации с помощью кодера и затем восстанавливать в исходном виде.

В качестве более общего решения можно ввести метод -setEncodedObject:forKey:, принимающий в качестве своего аргумента указатель id<NSCoding> и сохраняющий его с помощью кода из примера, приведенного в начале этого раздела.

Для сохранения в настройках по умолчанию объекта, не поддерживающего протокол NSCoding, имеются две возможности. Во-первых, можно ввести в него поддержку протокола NSCoding. С этой целью придется реализовать методы -encodeWithCoder: и -initWithCoder: в отдельной категории.

И во-вторых, можно выполнить преобразование другого рода. У большинства объектов имеются методы -stringValue и -initWithString:. С их помощью можно сохранить объекты в настройках по умолчанию, предварительно преобразовав их в символьные строки, а при чтении из настроек по умолчанию — создав новые объекты из этих символьных строк.