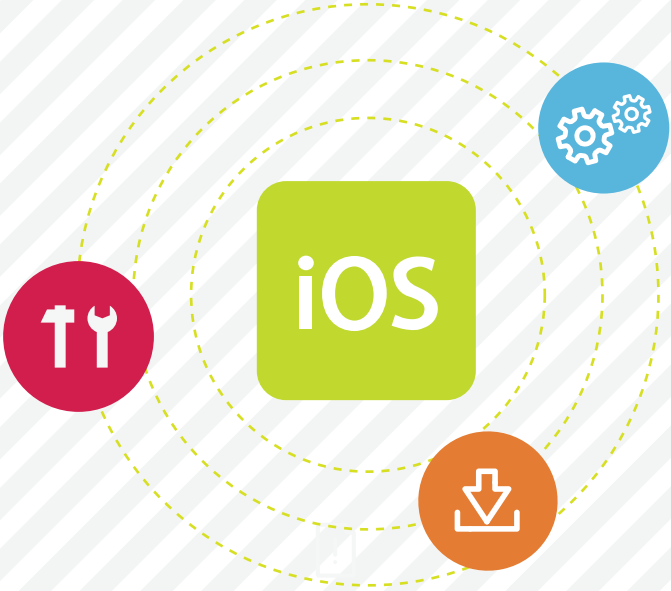
Ханг Во

Оптимизация производительности приложений для iOS



ДМК Пресс
издательство

УДК 004.451iOS
ББК 32.973.26-018.2
В61

Во Х.

В61 Оптимизация производительности приложений для iOS. – М.: ДМК Пресс, 2013. – 320 с.: ил.

ISBN 978-5-94074-856-4

Книга демонстрирует, как совершенствовать, увеличивать и оптимизировать производительность приложений для операционной системы iOS. Вы быстро научитесь создавать быстрые и отзывчивые приложения, пригодные для распространения на коммерческой основе. Эта книга охватывает множество общих и вместе с тем сложных проблем, возникающих при оптимизации производительности приложений для iPhone и iPad, и подробно описывает, как эффективно их решать. Она содержит массу практических знаний, приемов, советов и рекомендаций, которые помогут вам преуспеть в конкурентном мире разработки приложений для iOS.

Издание предназначено для программистов разной квалификации, разрабатывающих мобильные приложения под iOS.

УДК 004.451iOS
ББК 32.973.26-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-4302-3717-4 (анг.)
ISBN 978-5-94074-856-4 (рус.)

Copyright © 2011 by Khang Vo
© Оформление, перевод ДМК Пресс, 2013

Содержание

Об авторе ... 11

О техническом консультанте ... 12

Благодарности ... 13

Предисловие ... 14

Глава 1. Введение в оптимизацию производительности iOS ... 15

Новая эра смартфонов ... 15
Почему важна производительность ... 15
Для кого предназначена эта книга ... 16
Стиль автора ... 16
Что вам потребуется ... 17
Как пользоваться этой книгой ... 18
Обзор книги ... 18
Загружаемые примеры ... 20
Обратная связь ... 20

Глава 2. Оценка производительности с использованием эмулятора и действующего устройства ... 21

Эмулятор и действующее устройство ... 22
Память и производительность ... 23
Инструменты ... 23
Основные инструменты ... 24
Инструмент Allocations (Распределение памяти) ... 25
Унаследованный код ... 29
Инструменты оценки производительности ... 34
Выводы ... 55
Упражнения ... 55

Глава 3. Увеличение производительности UITableView 57
Предисловие к примерам 58
Повторный обзор инструментов 58
Первый пример 59
Второй пример 69
Чему учат данные примеры 73
Другие приемы 74
Сохранение значения высоты 74
Прозрачность 75
Отключение графических эффектов 76
Производительность редактирования/переупорядочивания 76
Выводы 77
Упражнения 79

Глава 4. Кеширование изображений и данных 80
Разница производительности обработки сети, файлов и памяти 80
Выявление узкого места 81
Введение в кеширование 83
Что такое кеширование? 83
Попадание в кеш 84
Промах кеша 84
Издержки извлечения 84
Издержки хранения 85
Недостоверность кеша 85
Политика вытеснения 86
Алгоритмы кеширования 87
Оценка кеша 94
Что следует кешировать 95
Где хранить изображения 95
Кеширование данных 100
Выводы 109
Упражнения 110

Глава 5. Оптимизация алгоритмов и структур данных 111
Первый пример 111
Теоретические вопросы тестирования производительности алгоритмов 114
Как измерить «большое O» 114
Сведения о реализации 116

«Большое O» известных алгоритмов ... 117
Практическая оценка ... 118
Структуры данных и алгоритмы ... 119
Структуры данных Cocoa Touch ... 120
Прочие структуры данных ... 132
Двоичное дерево ... 147
Граф ... 152
Прочие алгоритмы и подходы к решению проблем ... 159
Рекурсия ... 160
Парсер SAX/DOM для xml-парсинга ... 161
Выводы ... 163
Упражнения ... 164

Глава 6. Ускорение доступа к данным с использованием приемов многопоточного программирования ... 166
Что такое потоки и многопоточность ... 166
Терминология ... 168
Первый пример ... 169
Преимущества многопоточности ... 172
Создание многопоточных приложений ... 174
Создание потока ... 174
Настройка потока ... 182
Начало работы потока ... 183
Опасности, связанные с потоками ... 187
Синхронизация потоков ... 205
Альтернативы потокам ... 207
Инструмент Thread States (Состояние потоков) ... 211
Выводы ... 212
Упражнения ... 212

Глава 7. Оптимизация использования памяти ... 213
Краткий обзор ... 214
Старая политика владения объектом ... 214
Автоматическое освобождение объектов ... 214
Пул автоматически освобождаемых объектов ... 215
Автоматический подсчет ссылок ... 217
Как настроить проект в XCode ... 217
Политика автоматического подсчета ссылок ... 218
Новые модификаторы для автоматического подсчета ссылок ... 219
Свойства объектов ... 220
Дополнительные вопросы ... 221

Удерживающие циклические ссылки ... 221
Слабые ссылки ... 222
UIViewController ... 222
Процесс загрузки представления ... 223
Процесс выгрузки представления ... 224
Отображение и сокрытие представлений ... 225
Копирование объектов ... 226
Поверхностное и полное копирование ... 227
Реализация полного копирования ... 227
Интегрирование метода копирования в объект ... 229
Дополнительные возможности пула автоматически освобождаемых объектов ... 230
Пул автоматически освобождаемых объектов и стек ... 230
Пулы автоматически освобождаемых объектов и потоки ... 231
Влияние пулов автоматически освобождаемых объектов на производительность ... 231
Инструменты ... 232
Статический анализатор ... 232
Инструмент Leaks (Утечки памяти) ... 233
Объекты-зомби ... 233
Инструмент Allocations (Распределение памяти) ... 234
Уровни предупреждений о нехватке памяти ... 235
Выводы ... 235
Упражнения ... 235

Глава 8. Использование приемов многопоточного программирования и эффективного управления памятью в многозадачных приложениях ... 236

Что представляет собой многозадачность в iPhone ... 236
Жизненный цикл многозадачности ... 237
Преимущества и издержки многозадачности ... 244
Фоновые службы ... 245
Служба воспроизведения звука ... 245
Отображение экрана-заставки ... 247
Служба определения географического местоположения ... 247
Локальное уведомление ... 251
IP-телефония (VOIP) ... 252
Выполнение в фоновом режиме ... 252
На что обратить внимание при работе в фоновом режиме ... 255
Уведомления о системных изменениях ... 257
Совместимость с разными версиями iOS ... 258

Выводы 259
Упражнения 259

Глава 9. Улучшение производительности с использованием C/C++ 260
Преимущества и недостатки 261
Основы программирования на C и C++ 262
Программирование на C 262
Программирование на C++ 275
Практический пример 282
SQLite 282
Интеграция C++ в приложение 284
Выводы 284
Упражнения 285

Глава 10. Сравнение проблем производительности устройств на базе Android и Windows Phone 286
Общий обзор 287
Тестирование в эмуляторе и на действующих устройствах 288
Эмулятор и действующие устройства 288
Оценка производительности 289
Производительность ListView 291
Android 292
Windows Phone 295
Кеширование данных 296
Android 297
Windows Phone 299
Структуры данных и алгоритмы 301
Многопоточность 302
Android 303
Windows Phone 306
Управление памятью 307
Andriod 308
Windows Phone 309
Многозадачность 311
Android 311
Поддержка программирования на C/C++ 312
Выводы 313
Упражнения 314

Предметный указатель 315

Глава 2. Оценка производительности с использованием эмулятора и действующего устройства

Данная глава посвящена следующим вопросам:

- различия между эмулятором и действующим устройством;
- влияние управления памятью на производительность приложения;
- инструменты и способы оценки производительности приложения, включая следующее:
  - базовые инструменты измерения производительности и расходования памяти;
  - улучшенные инструменты исследования различных аспектов управления памятью, таких как утечки или неоптимальное использование памяти;
  - улучшенные инструменты исследования различных аспектов, влияющих на производительность, таких как аккумулятор, необходимость загрузки файлов и отображение информации;
  - разделение программы на более мелкие части с целью упростить выявление узких мест в производительности.

Для улучшения производительности необходимо провести оценочное тестирование, чтобы увидеть, в чем заключается проблема. Чтобы провести тестирование с пользой, необходимо понимать причины, ведущие к замедленной работе программы или сегмента кода.

На начальном этапе придется принять два фундаментальных решения: что выбрать – эмулятор или действующее устройство и как сохранить баланс между эффективным использованием памяти и высокой производительностью программного кода.

Для начала необходимо понять разницу между эмулятором и действующим устройством.

Эмулятор и действующее устройство

Главная причина низкой производительности приложений для iPhone состоит в том, что они выполняются в среде с ограниченными ресурсами и невысокой скоростью обработки. *Эмулятор среды iPhone* работает гораздо быстрее настоящего устройства. В сущности, скорость работы *эмулятора* зависит от скорости работы компьютера, на котором он выполняется. В результате очень большим и неприятным сюрпризом может явиться тот факт, что программа действительно быстро выполняется в *эмуляторе* и гораздо медленнее – на *действующем устройстве*. Я встречал множество людей, которые винили сеть в плохой производительности приложения. Безусловно, это иногда верно. Однако в большинстве случаев производительность приложения может сильно страдать именно из-за особенностей реализации кода, а не из-за сетевых проблем. Поэтому тщательное тестирование и оценка производительности приложения основными инструментами в стандартных средах позволят получить полное представление о скорости работы приложения и его пользовательского интерфейса.

Чтобы продемонстрировать значительную разницу между эмулятором и действующим устройством, я протестировал программу в эмуляторе iPhone и на действующем устройстве iPhone. Результаты впечатляют:

- в эмуляторе iPhone вычисления занимают 0,5 сек;
- на действующем устройстве iPhone те же вычисления занимают 7 сек.

Программа была несложная: я сделал простой тест с двумя массивами по тысяче элементов в каждом. Программа проходит в цикле по обоим массивам, чтобы найти одно и то же число, и выводит слово «привет». В действительности вам может и не понадобиться обработка тысячи элементов в массиве, или вы можете не использовать обработку массивов для поиска одинаковых чисел. Однако суть не в этом. Я выбрал именно эти действия, чтобы продемонстрировать, что действующее устройство iPhone намного медленнее эмулятора iPhone.

Отсюда следует вывод, к которому я буду еще не раз возвращаться в этой книге: приложение необходимо тестировать как в эмуляторе, так и на действующем устройстве. Так почему же не тестировать только на действующем устройстве? Потому, что эмулятор обладает следующими значительными преимуществами:

- тестирование в эмуляторе занимает меньше времени, что означает для разработчиков меньше временных потерь;
- эмулятор отлично подходит для выявления утечек памяти и проблем ее распределения.

Память и производительность

Память и производительность представляют собой разные вещи. Под памятью обычно понимается *оперативная память* (Random Access Memory – RAM) и, соответственно, сколько места вы использовали и сколько осталось свободным. Производительность обозначает, насколько быстро приложение выполняет определенную операцию.

Память может оказывать большое влияние на производительность. Когда устройство обладает большим объемом оперативной памяти, можно предварительно загрузить и кешировать в ней больше данных. Оперативная память обладает более высокой скоростью доступа, по сравнению с файловым и сетевым хранилищами. Предварительно загрузив и кешировав больше данных в оперативной памяти, во многих случаях можно значительно увеличить скорость программ. Например, для игрового приложения, требующего загрузки большого количества изображений, большой объем памяти является необходимым условием, потому что позволяет заранее загрузить картинки и отображать их по мере необходимости. Загрузка из оперативной памяти в 10 раз быстрее, чем загрузка из файловой системы.

Однако более эффективное использование памяти не всегда означает лучшую производительность. Некоторые приложения не требуют много памяти, поэтому, оптимизируя использование памяти, можно столкнуться с тем, что производительность уже не повышается. Обратная ситуация не лучше: приложение может занять всю память, чтобы добиться лучшей производительности, а затем ему памяти просто не хватит.

Таким образом, необходимо всегда тщательно оценивать как память, так и скорость выполнения, чтобы убедиться в наличии строгого баланса между использованием памяти и производительностью во время выполнения.

Инструменты

Инструменты делятся на следующие три основные категории:

- основные, помимо инструментов среды разработки Xcode;

- инструменты оценки правильности и эффективности использования памяти;
- инструменты оценки производительности, позволяющие определить, насколько быстро выполняется каждая часть программы, и выявить слабые места.

Основные инструменты

В этой части в качестве основного метода измерения будет обсуждаться метод регистрации времени выполнения блоков кода.

Регистрация времени выполнения

Одним из самых простых методов является регистрация временной разницы между началом и окончанием выполнения блока кода. Обычно такая регистрация осуществляется посредством функции `NSLog`. Используя ее, разработчики могут выяснить, насколько быстро выполняется каждая строка или блок кода.

Например, при выполнении следующего фрагмента кода

```
NSDate *date1 = [NSDate date];
for (int i = 0; i < 1000; i++) {
  // Здесь выполняются вычисления
}
NSDate *date2 = [NSDate date];
NSLog (@"time: %f", [date2 timeIntervalSinceDate:date1]);
```

были получены такие результаты:

```
time: 0.0123 (измеряется в секундах)
```

Преимущества:

- простой и понятный способ измерения производительности;
- возможность проверки производительности отдельных строк и блоков кода.

Недостатки:

- отсутствие возможности оценить производительность пользовательского интерфейса (то есть время отображения, зависящее от действий пользователя);
- бессмысленная оптимизация (можно потратить много времени на какой-то определенный блок кода, увеличив скорость его работы совсем незначительно);

- в эмуляторе приложение обычно выполняется быстро. При такой скорости функция `NSLog` не поможет выявить разницу в производительности выполняющегося приложения. С другой стороны, так как `NSLog` на действующем устройстве работает медленно, она поможет выявить такую разницу.

Применение:

- когда необходимо быстро провести измерения без долгих приготовлений;
- когда необходимо быстро получить результат;
- когда необходимо изолировать небольшой блок кода, чтобы проверить предполагаемую производительность.

Инструменты оценки использования памяти

В том, что касается памяти, главной задачей является ее наиболее эффективное использование. Существуют также менее значительные проблемы с унаследованным кодом, такие как утечки и чрезмерное расходование памяти. В новых проектах предпочтение следует отдавать новому механизму *автоматического подсчета ссылок* (*ARC – Automatic Reference Counting*). Некоторые старые проекты можно попробовать преобразовать с помощью утилиты, входящей в комплект среды разработки Xcode.

Однако преобразованию поддается не каждый проект. Политика управления памятью и еще ряд проблем не позволят этого сделать. Попытки придерживаться новой политики управления могут создать еще большие проблемы. Поэтому здесь в основном рассматриваются инструменты, позволяющие выяснить распределение памяти для объектов, и более кратко инструменты, выявляющие утечки и чрезмерное расходование памяти.

> **Примечание**: все инструменты оценки использования памяти, которые представлены здесь (а здесь представлены все целевые инструменты Apple), могут использоваться в эмуляторе. Он хорош тем, что действительно быстр в исполнении и быстро загружает приложения. Однако я настоятельно рекомендую тестировать свои приложения также и на действующем устройстве, потому что эмулятор и действующее устройство не всегда одно и то же. Они имеют разные строение и архитектуру.

Инструмент Allocations (Распределение памяти)

Инструмент **Allocations** (Распределение памяти) помогает узнать, как много памяти выделяется для объектов. Он позволяет выяснить,

не слишком ли много объектов хранится в памяти и какие из них пока не могут быть освобождены, потому что продолжают использоваться.

Распределение памяти

Выберите пункт меню **Product** (Продукт) ⇒ **Profile** (Профилировать), а затем в открывшемся окне выберите ярлык **Allocations** (Распределение памяти) (как показано на рис. 2.1).

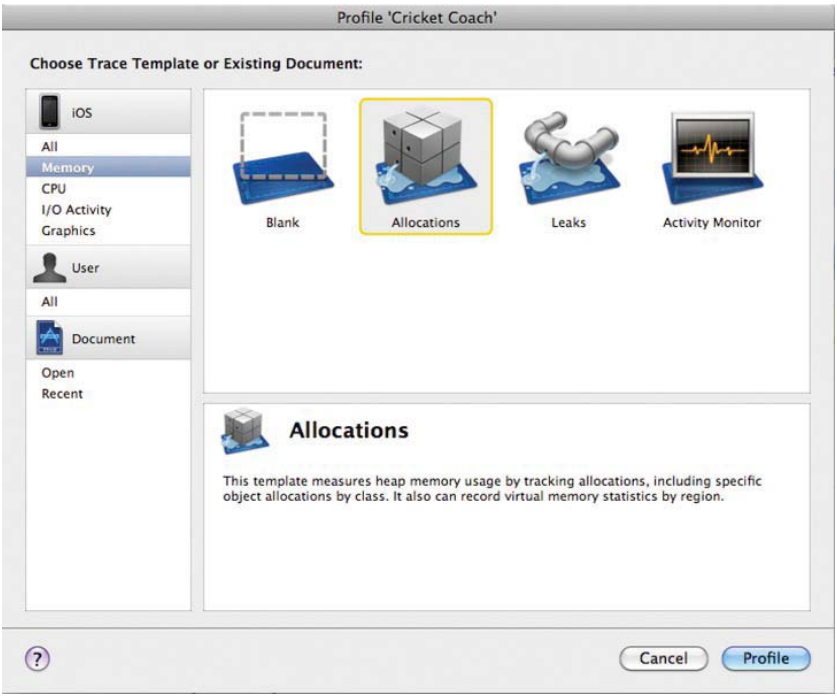


Рис. 2.1. Выбор ярлыка инструмента отображения распределения памяти

После выбора инструмента откроется его основная панель, на которой предоставлена вся необходимая информация, как показано на рис. 2.2.

Панель инструмента **Allocations** (Распределение памяти) (рис. 2.2) по умолчанию отображает объекты **Created & still living** (Созданные и существующие в настоящий момент), что позволяет

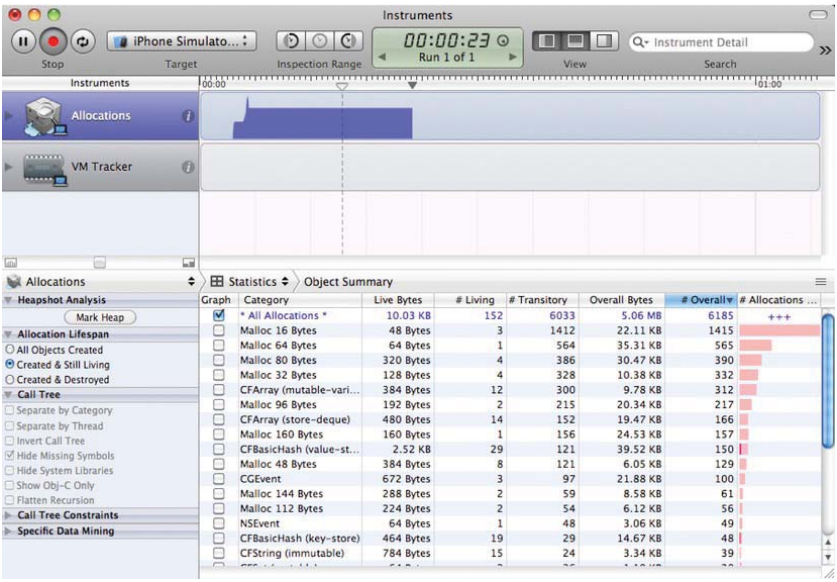


Рис. 2.2. Основная панель инструмента отображения распределения памяти

увидеть объекты, которые все еще находятся в памяти, и объекты, занимающие большую ее часть. Этот инструмент следует использовать, когда начинает появляться большое количество предупреждений от среды iOS, как, например, «Received memory warning. Level = 1» (Получено предупреждение о нехватке памяти. Уровень = 1).

Таблица с подробной информацией содержит сведения о том, какие строки кода и какие классы ответственны за создание и удержание объекта. Используя эту информацию, можно легко найти способ более эффективного использования памяти. Это хороший инструмент для исследования алгоритмов и методов кеширования (подробнее о кешировании рассказывается в главе 4).

Рисунки 2.3 и 2.4 более детально отображают, какие объекты продолжают существовать и потребляют большую часть памяти приложения. На рис. 2.3 изображена таблица с подробной информацией об объектах, созданных и продолжающих существовать внутри приложения.

На рис. 2.4 видно, какие методы вызывались для создания этих объектов.

Graph	Category	Live Bytes	# Living	# Transitory	Overall Bytes	# Overall	# Allocations (Net / Overall)
☑	* All Allocations *	27.77 KB	565	55386	17.06 MB	55951	
☐	CFRunLoopTimer	11.72 KB	125	250	35.16 KB	375	
☐	CFBasicHash (value-st...	4.50 KB	135	1146	109.23 KB	1281	
☐	CFSet (mutable)	3.91 KB	125	266	12.22 KB	391	
☐	CFBasicHash (count-s...	2.36 KB	8	57	16.86 KB	65	
☐	__NSCFDate	1.94 KB	124	247	5.80 KB	371	
☐	CGEvent	672 Bytes	3	291	64.31 KB	294	
☐	Malloc 144 Bytes	576 Bytes	4	924	130.50 KB	928	
☐	Malloc 48 Bytes	480 Bytes	10	1353	63.89 KB	1363	
☐	Malloc 80 Bytes	320 Bytes	4	4787	374.30 KB	4791	

Рис. 2.3. Подробная информация о распределении памяти

Category	Timestamp	Live	Size	Responsible Library	Responsible Caller
Malloc 80 Bytes	00:22.186.397	•	80	CoreGraphics	CGClipStackCreateMutable
Malloc 80 Bytes	00:25.680.592	•	80	CoreGraphics	CGClipStackCreateMutable
Malloc 80 Bytes	00:26.680.849	•	80	AppKit	-[NSViewHierarchyLock lock...
Malloc 80 Bytes	00:26.680.980	•	80	CoreGraphics	CGClipStackCreateMutable

Рис. 2.4. Детали распределения памяти

Преимущества:

- наличие точной и подробной информации о времени и условиях, в которых приложение потребляет больше всего памяти;
- позволяет получить достаточно полное представление о жизненном цикле объекта по отношению к жизненному циклу приложения.

Недостатки:

- результаты зависят от того, как выполняется приложение. Требуется тщательная подготовка комплексного набора тестов, чтобы охватить как можно больше вероятных ситуаций;
- возможны временные и трудовые затраты на создание набора тестов, который поможет разработчикам выявить место и время, когда приложение потребляет больше всего памяти;
- необходимо выполнять тестирование на действующем устройстве, чтобы можно было получать предупреждения о нехватке памяти. Эмулятор практически никогда не выдает таких предупреждений. Проблема заключается в том, что компьютер будет иметь оперативную память 2–4 Гб, тогда как действующее устройство – гораздо меньше.

Применение:

- это один из первых инструментов, к которым следует обратиться, если при работе с приложением появляются предупреждения о нехватке памяти.

Унаследованный код

Текущая версия утилиты преобразования проекта с ручного управления памятью на использование нового механизма автоматического подсчета ссылок может не сработать. Она может запросить исправления слишком многих участков текущего программного кода, чтобы убедиться, что проект можно преобразовать в проект с автоматическим подсчетом ссылок. Может случиться так, что не получится преобразовать библиотеку с открытым исходным кодом, а трогать ее вам не захочется. Поэтому для начала хорошо бы ознакомиться с основами ручного управления памятью.

Утечки памяти

Утечки памяти случаются при создании в памяти все новых и новых объектов без их своевременного удаления. Эти объекты будут продолжать оставаться в памяти в течение всего времени выполнения приложения. В результате приложению может не хватить памяти или, того хуже, iOS постарается его закрыть.

Статический анализатор

Этот простой *инструмент* предназначен для выявления утечек памяти. Как показано на рис. 2.5 и 2.6, данный инструмент позволяет определить, какая строка или блок кода **могут** являться причиной утечки.

Как показано на рис. 2.5, необходимо выбрать пункт меню **Product** (Продукт) ⇒ **Analyze** (Анализировать) или нажать комбинацию клавиш **Command Shift+B**.

Как видно на рис. 2.6, объект `str`, созданный в строке 19, нигде не освобождается. В данном случае статический анализатор выдал верное предупреждение.

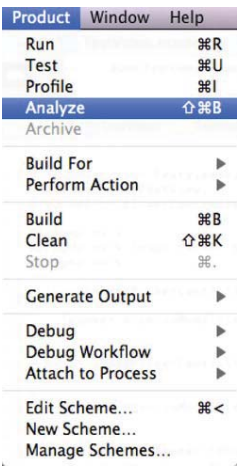


Рис. 2.5. Выбрать **Product** ⇒ **Analyze**



Рис. 2.6. Статический анализатор отмечает потенциальную утечку при распределении памяти для объекта `str` в строке 19

Преимущества:

- ❑ быстрый и полный обзор мест, где возможны утечки памяти;
- ❑ проверка выполняется действительно быстро: статический анализатор просто собирает и просматривает исходный код, не запуская программу;
- ❑ не требует никаких усилий от разработчика, достаточно выбрать пункты меню **Product** (Продукт) и **Analyze** (Анализировать).

Недостатки:

- ❑ неточность. Иногда тест выдает неверное предупреждение или пропускает места утечек.

Применение:

- ❑ первым используется для выявления утечек памяти, потому что является быстрым и легким инструментом.

Инструмент Leaks (Утечки памяти)

Данный *инструмент* точнее, поскольку выявляет утечки во время выполнения приложения и имеет возможность удостовериться, что контроль над объектом действительно потерян. В случае утраты контроля над объектом пользователь получит сообщение. В процессе использования инструмента необходимо запускать различные функции приложения, а он выделит те места, где обнаружатся утечки.

При работе с инструментом необходимо найти места в программном коде, где на горизонтальной полосе он поставит вертикальную черту. Высота черты показывает, сколько памяти приложение потеряло в данный момент (см. рис. 2.7).



Рис. 2.7. Объем потерь во время выполнения

В таблице с детальной информацией можно увидеть список всех утечек, обнаруженных во время выполнения приложения. Отсортировав по имени библиотеки и отыскав название приложения (в данном случае `LeaksViewController`), обнаруживаем два потерянных объекта. Сразу видно, что в классе `RootViewController` были потеряны два изображения.

На рис. 2.9 видна небольшая стрелка в столбце **Address** (Адрес). Если щелкнуть на ней, инструмент **Leaks** (Утечки памяти) выполнит переход в то место программы, где произошла утечка (см. рис. 2.9).

Leaked Object	#	Address	Size	Responsible Library	Responsible Frame
UIImage		0x8026020	16 Bytes	LeaksViewController	-[RootViewController
UIImage		0x4b2a250	16 Bytes	LeaksViewController	-[RootViewController
Malloc 9.00 KB		0x5026a00	9.00 KB	ImageIO	initImageJPEG

Рис. 2.8. Список утечек памяти в программе

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSString *avatarFile = [NSString stringWithFormat:@"a0"];
    NSString *avatarName = [[NSBundle mainBundle] pathForResource:avatarFile ofType:@"jpeg"];
    UIImage *image1 = [[UIImage alloc] initWithContentsOfFile:avatarName];
    NSLog(@"image: %@", image1);
    image1 = [[UIImage alloc] initWithContentsOfFile:avatarName];
    NSLog(@"image: %@", image1);
```

Рис. 2.9. Строки кода, вызвавшие утечку

Теперь можно исследовать строку кода, где произошла утечка. Обычно этот инструмент точно указывает место, где случилась утечка, так что ее довольно легко исправить.

Преимущества:

- точность и детальность.

Недостатки:

- результаты зависят от того, как выполняется приложение. Необходимо подготовить хороший набор тестов, чтобы охватить как можно больше случаев;
- низкая скорость, потому что приходится запускать несколько раз, чтобы посмотреть, как программа работает в разных случаях.

Применение:

- данный инструмент необходимо использовать после статического анализатора. Он выявит все остальные места, которые пропустил статический анализатор.

Чрезмерное расходование памяти

На первый взгляд, *чрезмерное расходование памяти* никак не связано с производительностью. Однако аварийное завершение программы – гораздо худшая ситуация, чем низкая производительность, поскольку уничтожает то, чего достиг пользователь на этот момент. Поэтому необходимо научиться эффективнее использовать память.

Объекты-зомби

Выберите **Product** ⇒ **Profile** ⇒ **Allocations** (Продукт ⇒ Профилировать ⇒ Распределение памяти).

Вы увидите уже знакомый инструмент **Allocations** (Распределение памяти). Проблема заключается в том, что он не способен

выявлять объекты-зомби или как-то иначе помочь в решении этой проблемы, поэтому необходимо его остановить и перенастроить на работу с объектами-зомби. Иными словами, в случае сбоя этот инструмент укажет, где именно он произошел.

На этом этапе данные в нижней части рис. 2.10 не представляют интереса. Наша цель – настроить инструмент **Allocations** (Распределение памяти) на выявление объектов-зомби. Необходимые настройки изображены на рис. 2.11.

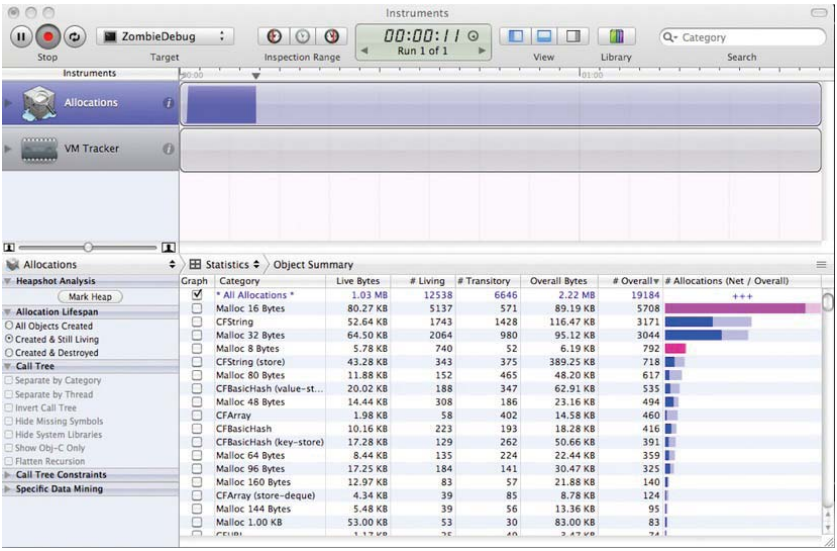


Рис. 2.10. Вид инструмента **Allocations**

После настройки инструмента его необходимо запустить заново. Делать это нужно в эмуляторе iPhone или на действующем устройстве.

Далее нужно просто попытаться воспризвести ошибку, выполняя различные функции приложения до его отказа. Тогда в инструменте **Allocations** (Распределение памяти) появится сообщение, как показано на рис. 2.12.

Если щелкнуть на стрелке, чтобы перейти к просмотру подробной информации, в нижней части появится список операций с памятью для указанного объекта, такие как выделение памяти (malloc), автоматическое освобождение (autorelease), удержание (retain) и осво-

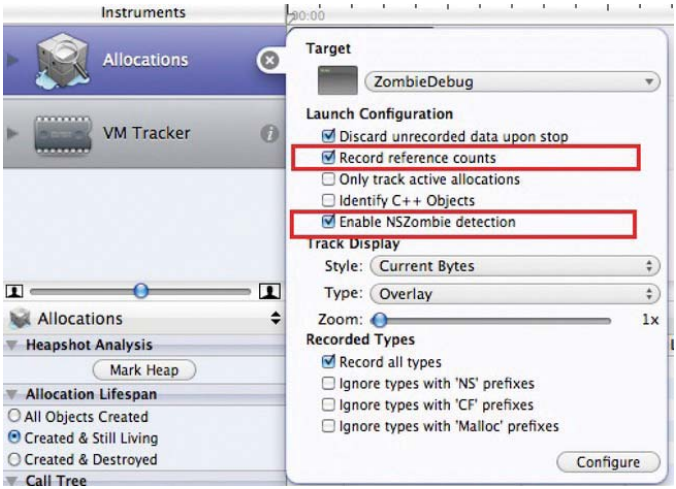


Рис. 2.11. Основные параметры настройки инструмента **Allocations** (Распределение памяти)

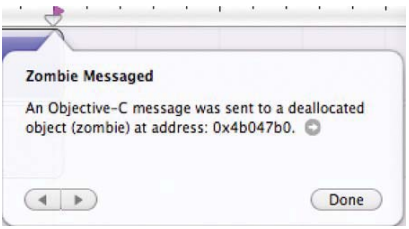


Рис. 2.12. Сообщение об аварийном завершении, вызванном появлением объекта-зомби

бождение (release), как показано на рис. 2.13. Исследуемый проект называется `ZombieDebug`, поэтому в таблице нас интересуют только вторая, третья и четвертая строки.

Statistics › Object Summary › History: 0x4b047b0

#	Category	Event Type	RefCt	Timestamp	Address	Size	Responsible Li...	Responsible Caller
0	CFNumber	Malloc	1	00:14.433	0x4b047b0	16	Foundation	-[NSPlaceholderNumber ...
1	CFNumber	Autorelease		00:14.433	0x4b047b0	0	ZombieDebug	-[ZombieDebugViewCon...
2	CFNumber	CFRetain	2	00:14.433	0x4b047b0	0	ZombieDebug	-[ZombieDebugViewCon...
3	CFNumber	CFRelease	1	00:14.433	0x4b047b0	0	ZombieDebug	-[ZombieDebugViewCon...
4	CFNumber	CFRelease	0	00:14.433	0x4b047b0	0	Foundation	-[NSAutoreleasePool rel...
5	CFNumber	Zombie	-1	00:14.977	0x4b047b0	0	Foundation	-[NSCFString appendFor...

Рис. 2.13. Список операций с памятью, таких как выделение памяти, автоматическое освобождение, удержание и освобождение