Тестирование производительности

Мы поговорим о таких показателях производительности как - загрузка ЦП, время отклика, затраты батареи, затраты ОП, утечки памяти.

На самом деле при тестировании производительности также вычисляется рабочее количество пользователей приложения, измеряется время выполнения различных операций, определяется производительность приложения и ее границы при различных степенях нагрузки, исследуется работу приложения при длительной нагрузке, проводится тестирование работоспособности приложения при нагрузках, превышающих пользовательские и тд.



### Для чего нужно тестировать производительность?

* Чтобы наше приложение было производительнее приложений конкурентов;
* Чтобы пользователям было удобно с ним взаимодействовать (приложение должно запускаться и работать быстро, не должно затрачивать слишком много батареи и ОЗУ, а также превышение количества пользователей/операций не должно критично влиять на работу системы).

Тестировать производительность на симуляторах/эмуляторах можно\*, но **не во всех случаях**. Лучший способ это делать на реальных девайсах, т.к симулятор будет использовать все вычислительные мощности компьютера, что в свою очередь может привести к необъективным результатам подобного теста.

\* на симуляторах можно потестить потребление ОЗУ.

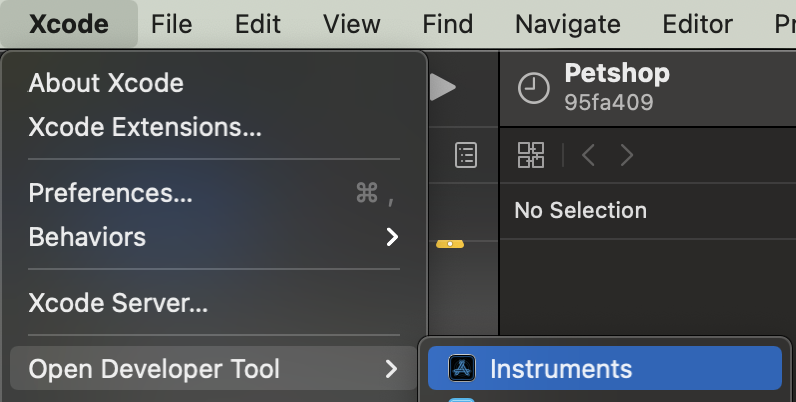
### Как выбрать девайсы для следующих тестов?

Как и обычно: если на менее производительных девайсах с меньшим объемом памяти тесты будут пройдены, то мы можем быть уверены, что наше МП будет вести себя корректно и на более производительных устройствах.

# Проверка производительности iOS-приложений

### Предварительные шаги:

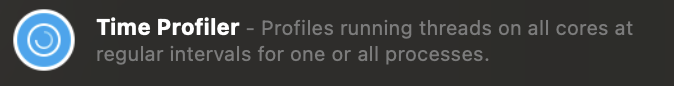
1. Подключить девайс к xcode
2. Собрать проект локально на девайс ([инструкция](https://docs.google.com/document/d/1yXSdlw_LryUBVGYwWSIsD8l-ogZx8Y-0SjAQZ5OdnsY/edit#))
3. Открыть панель инструментов Xcode - Developer tools - Instruments



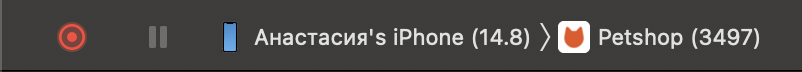
Для проверки разных показателей производительности следует создать разные наборы кейсов (влияющие как раз на этот параметр). Если мы занимаемся тестированием производительности на постоянной основе, то нужны чек-листы для каждого вида теста и обсуждение с разработчиками для определения важных кейсов и рисков. Если нам нужно локализовать проблему, то мы уже знаем потенциальное место (юз кейс).

## Загрузка ЦП

*Путь к инструменту: Developer tools - Instruments - Time Profiler*

**

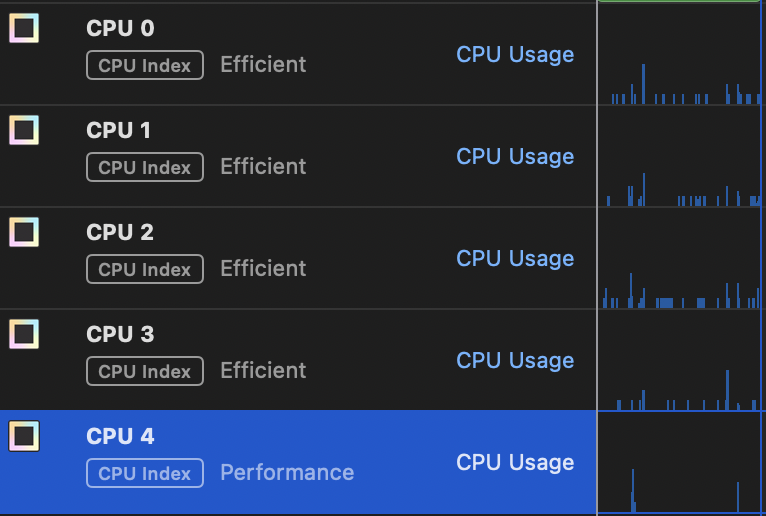
В верхней строке по клику на девайс выбираем наше приложение и кликаем на запись в верхнем левом углу.



Работаем с приложением (нужная часть приложения, нужные кейсы), после выполнения всех необходимых действий выключаем запись кнопкой на том же месте и смотрим на график.

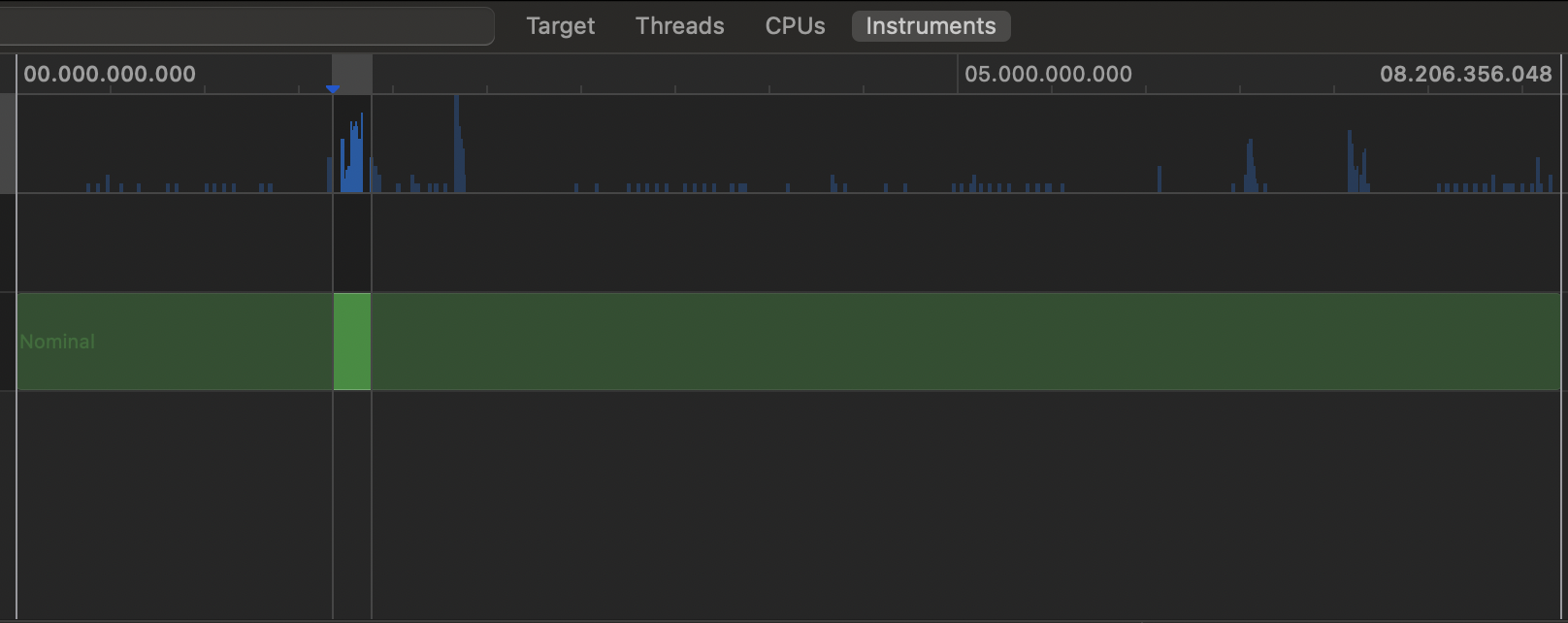
На что можно обратить внимание?

* должны быть загружены все ядра (у разных девайсов разное кол-во ядер),
* должна быть симметрия (иначе одно или несколько ядер простаивает) которая позволит в дальнейшем снизить нагрузку с помощью распределения задач и таким образом ускорить работу МП)

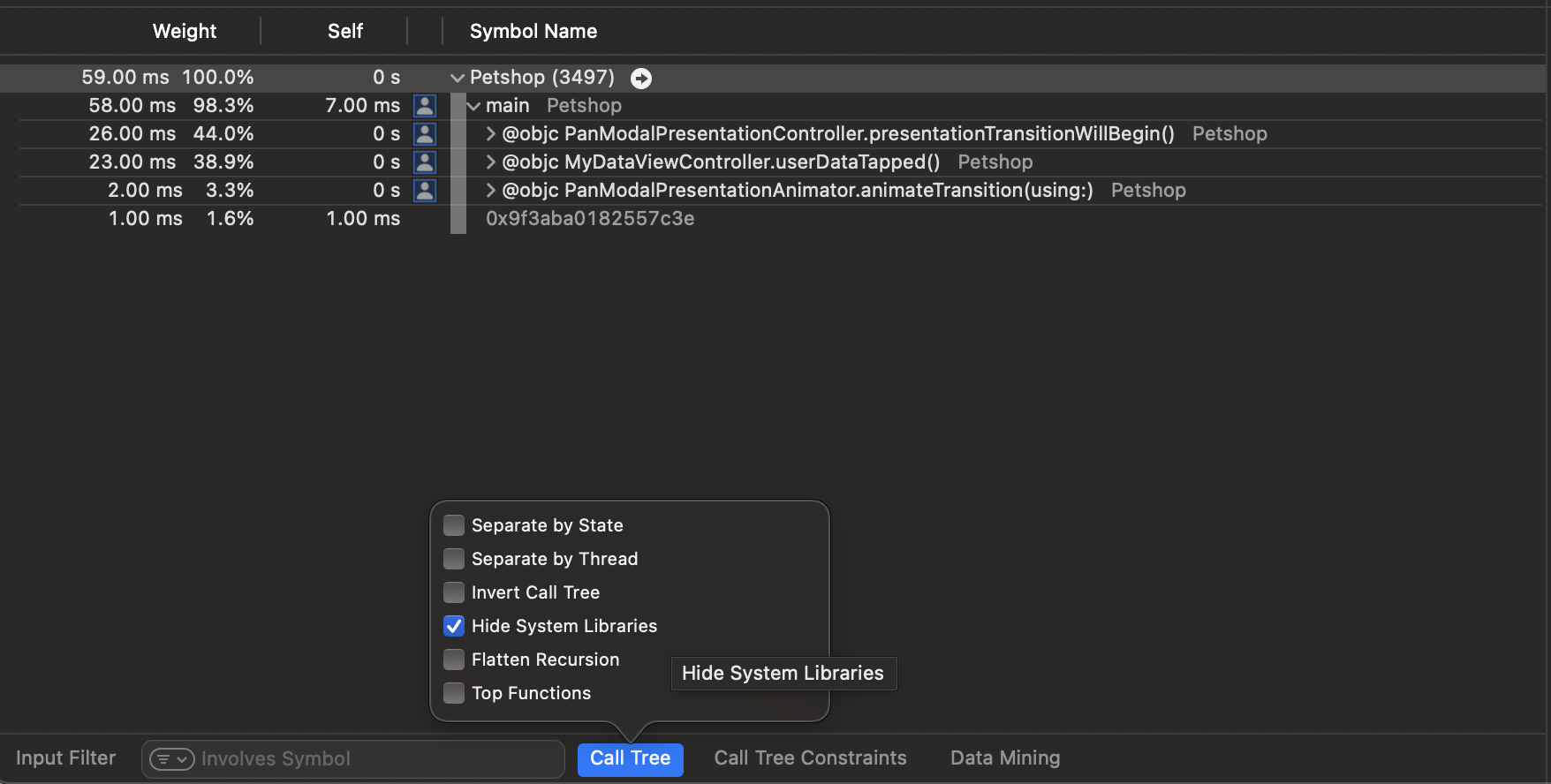


* можно раскрыть все методы и посмотреть какой метод вызывает проблему, какая часть кода затрачивает больше всего времени\*

\**Для этого после записи выбираем вкладку Instruments и выделяем нужный отрезок записи (тот, что грузил процессор).*



В нижней части отобразятся все методы, выполненные за этот промежуток времени. Для того, что бы увидеть только те, что написаны нашими разработчиками, необходимо нажать на кнопку Call Tree и там скрыть системные библиотеки.



Далее можно дважды кликнуть по затратному методу и заглянуть в код и ту его часть, что выполнялась дольше всего.

# Проверка производительности Android-приложения

## Предварительные шаги:

1. Подключить устройство к Android Studio по USB
2. На девайсе в настройках включить Параметры разработчика (Зайти в “О телефоне” ➝ много раз подряд тапнуть на Номер сборки)
3. В Параметрах разработчика включить Отладку по USB
4. Проверить, чтобы конфигурация USB находилась в режиме передачи файлов

## Обзор Profiler ЦП в Android Studio

Встроенный в Android Studio профилировщик позволяет отследить все главные метрики производительности приложения, такие как: память, процессор, использование сети и потребление энергии.

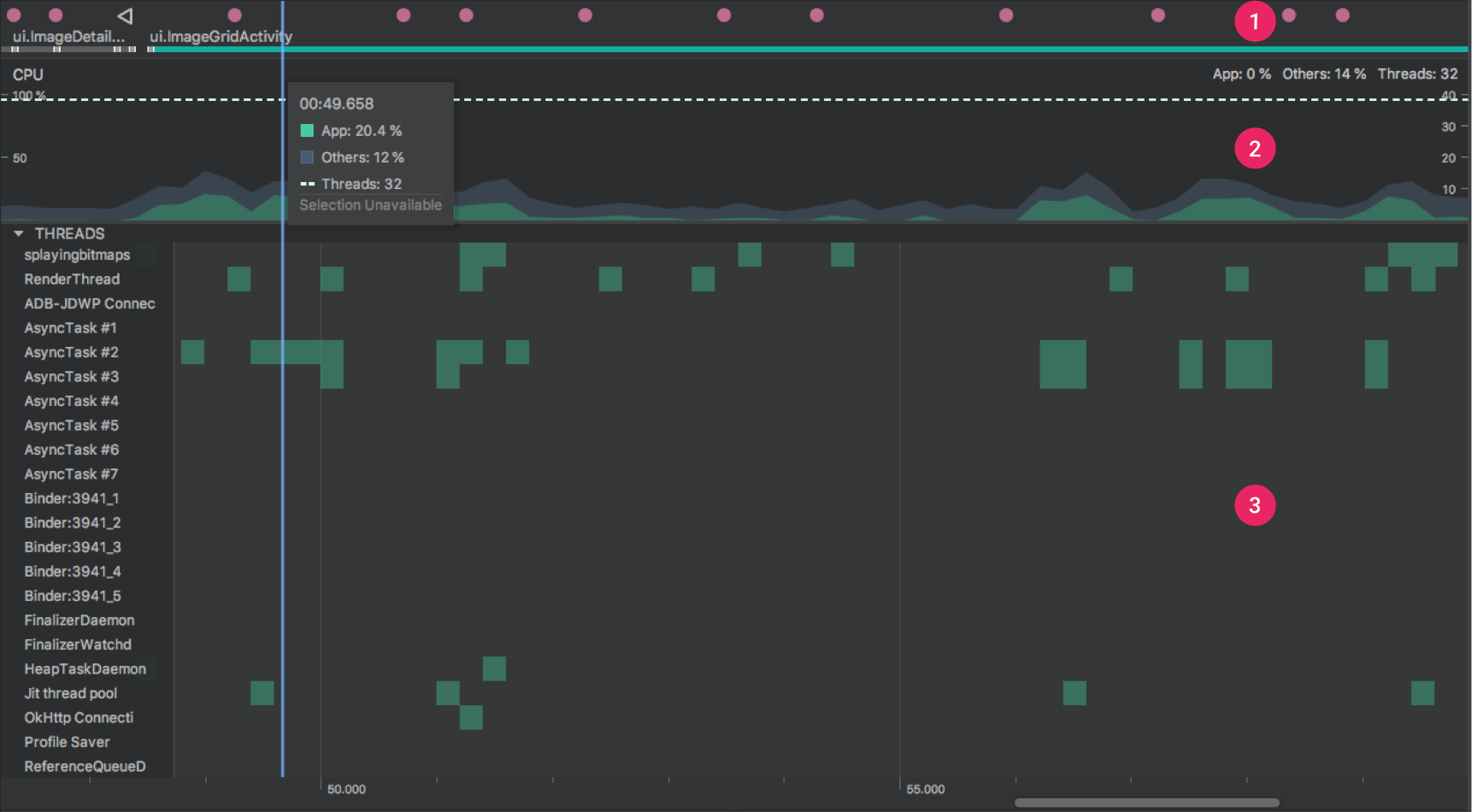
### Мониторинг использования процессора

Если приложение подвисает при переходе между экранами или же просто долго отвечает при нажатии на кнопку, то вероятнее всего процессор слишком сильно загружен в данный момент.

Чтобы понять, что именно тормозит выполнение программы достаточно открыть Profiler ЦП (выполните следующие действия):

1. Нажмите **View > Tool Windows > Profiler**  или « **Profiler**  » на панели инструментов.
2. При появлении запроса в диалоговом окне «**Select Deployment Target**» выберите ваше устройство.
3. Щелкните в любом месте временной шкалы **ЦП** , чтобы открыть Profiler ЦП.

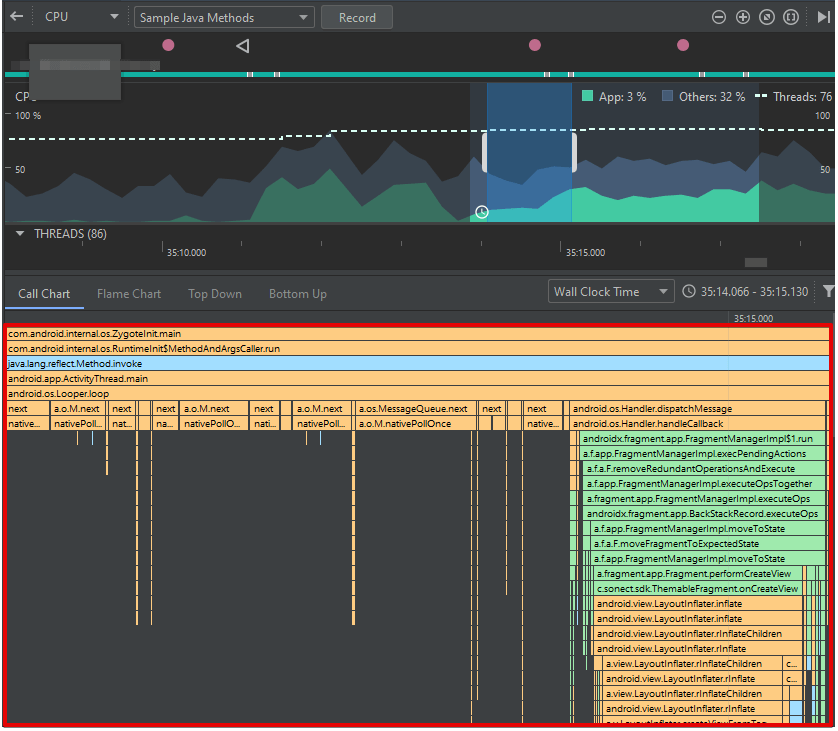
При открытии CPU Profiler - сразу же начинает отображаться использование ЦП вашим приложением и активность потока. Пример показан на изображении ниже:



*Временные шкалы в CPU Profiler*

Для того чтобы узнать какие методы выполнялись и за сколько времени, необходимо сделать следующие действия:

* Нажать кнопку Record
* Выполнить действия, которые вызывают зависание программы
* Остановить запись



Таким образом сможете найти проблемное место приложения.

### Когда рекомендуется использовать инструменты мониторинга производительности на Android и iOS?

* высоконагруженное сложное МП (например, перед релизом, чтобы удостовериться, что новые фичи верно распределяют нагрузку и тд)
* в случае жалобы на долгую операцию (выполняем описанную операцию в профайлере, смотрим на загрузку процессора во время долгой операции, дальше можно перейти к процессам, отсортированным по проценту использования цп и методам в коде, чтобы показать разработчику)

# Утечки памяти

### Как утечки памяти могут отразиться на приложении?

В приложении многие объекты имеют ограниченное время жизни. По его прошествии они уничтожаются сборщиком мусора. Но если объект доступен по цепочке ссылок, он не будет уничтожен. Приложение выделяет память, но не освобождает её, это происходит, пока не исчерпается лимит и не случится краш.

Первая проблема, с которой мы можем столкнуться связано с тем, что приложение крашится с ошибкой **OutOfMemoryError**. Именно эта ошибка говорит о том, что у приложения закончилась выделенная ему память.

Иногда все не так очевидно. Был пример в виде стабильного и отлаженного приложение музыкального плеера. На тестирование поступил билд с парой новых экранов. Сначала приложение вело себя как обычно, но спустя минут десять начинало тормозить. Иногда зависало или закрывалось без алертов.

### Зачем тестировщику знать, как локализовывать утечки?

Если сказать разработчику, что приложение само свернулось, тормозит или зависает, он скорее всего мало чем сможет помочь. С большей долей вероятности вас отправят локализовывать проблему и искать точные шаги ее воспроизведения. Разработчик надеется, что ваша инициатива померкнет и вы больше к нему с подобными глупостями приходить не станете. Но так как вы ответственный борец за качественную и стабильную работу приложения, просто так это не оставите. Надо локализовать проблему и обосновать разработчику её важность и критичность.

### Можно поделить мобильные приложения на два типа:

#### Пользователь находится в приложении менее 10 минут

Примером могут послужить мобильные приложения для покупки билетов, мобильный банк, органайзер. Пользователь заходит в приложение для достижения конкретной цели.

В этом случае нет острой необходимости тщательно следить за утечками. Мелкие не успеют серьезно отразиться на работе приложения, при этом крупные легко найти (пример - экран пересоздается при каждом заходе на него)

1. Пользователь находится в приложении более 10 минут

Это история в большей части связана с музыкальными плеерами и соц. сетями, которые пользователь будет испытывать на прочность часами. Наравне с тестированием вы должны следить за утечками памяти и стабильной работой приложения. Даже небольшая утечка при сессии в 40 минут может отразиться на работе.

# Подходы к локализации утечек

## Локализация вручную:

Если повезет, сразу найдете последовательность, которая порождает утечки. Локализовывать утечки вручную крайне трудоемко. Вам придется повторять однотипные действия десятки раз и смотреть, повлияло ли это на работу вашего приложения. Потратив неприлично большое количество времени, выявить закономерность. Например: после 20 переключений между табами приложение работало некорректно. Затем и вовсе крашилось. На разных девайсах количество переключений может варьироваться.

Когда пытаетесь локализовать утечки только руками, нет гарантий, что утечка реально прикрыта. Возможно, она просто не воспроизводится по вашим шагам.

*Совет: при локализации утечек в первую очередь проверьте вложенность экранов. Чем глубже, тем лучше. Надо учесть всё: пуши, объекты с картинками, анимации, списки, карты с отражающимися на них элементами, вертикальную и альбомную ориентацию.*

В связи с тем, что поиск утечек вручную достаточно трудоемкий и не всегда эффективный процесс, рекомендуется использовать специальные для этого инструменты.

## Локализация с помощью специальных инструментов:

### Поиск утечек с помощью Android-приложений.

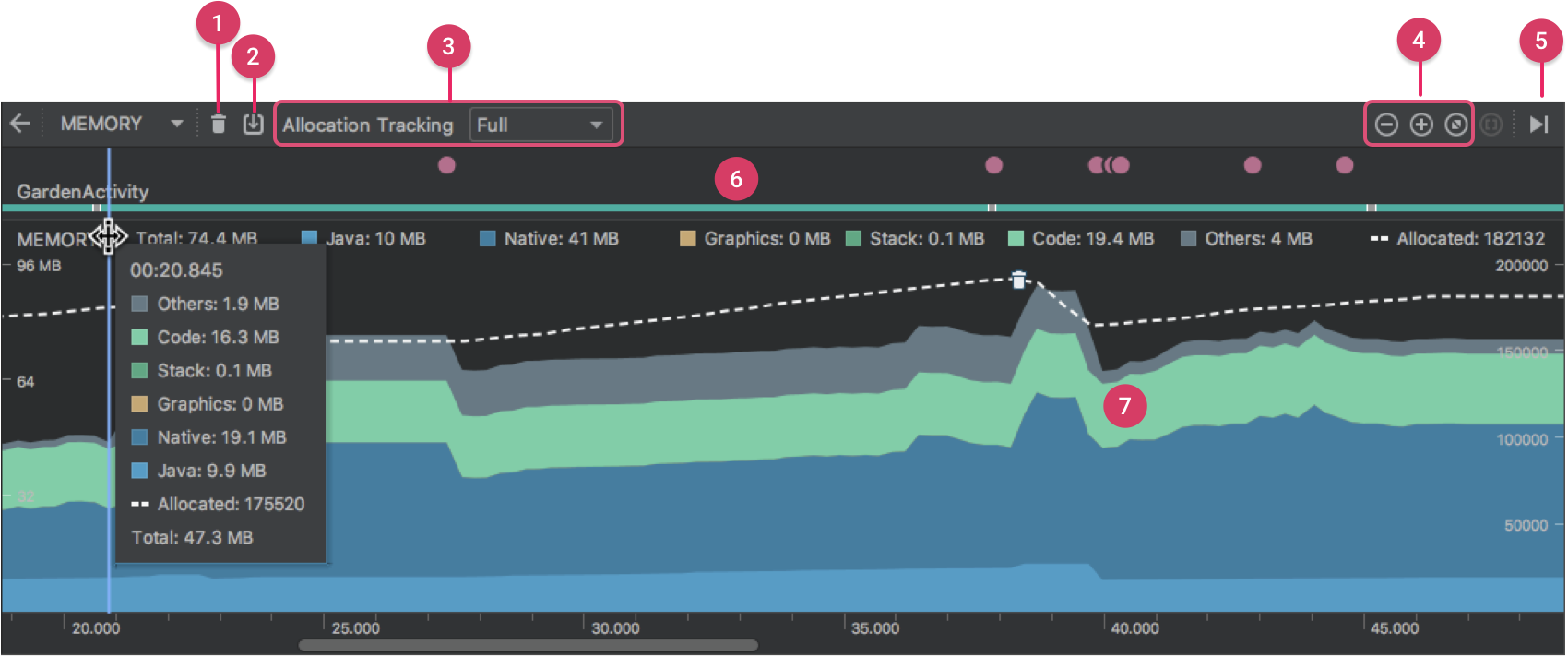
В Android Studio Profiler можно заметить, что иногда, при активном использовании приложения график Memory неустанно растет. Это может означать утечку памяти в вашей программе. Android Studio позволяет посмотреть какие объекты создавались при работе программы. Таким образом, повторяя действия, которые приводят к утечке памяти можно узнать, какой тип объектов влияет на производительность.

**Memory Profiler** — это компонент [Android Profiler](https://developer.android.com/studio/preview/features/android-profiler), который помогает выявлять утечки памяти и нехватку памяти, которые могут привести к лагам, зависаниям и даже сбоям приложений. Он показывает график использования памяти вашим приложением в реальном времени и позволяет записывать дамп, принудительно собирать мусор отслеживать выделение памяти.

#### **Обзор Memory Profiler**

* открыть Profiler ЦП
* кликнуть в любом месте временной шкалы MEMORY, чтобы открыть Memory Profiler.

Когда вы впервые откроете профилировщик памяти, вы увидите подробную временную шкалу использования памяти вашим приложением и получите доступ к инструментам для принудительной сборки мусора и записи выделения памяти.



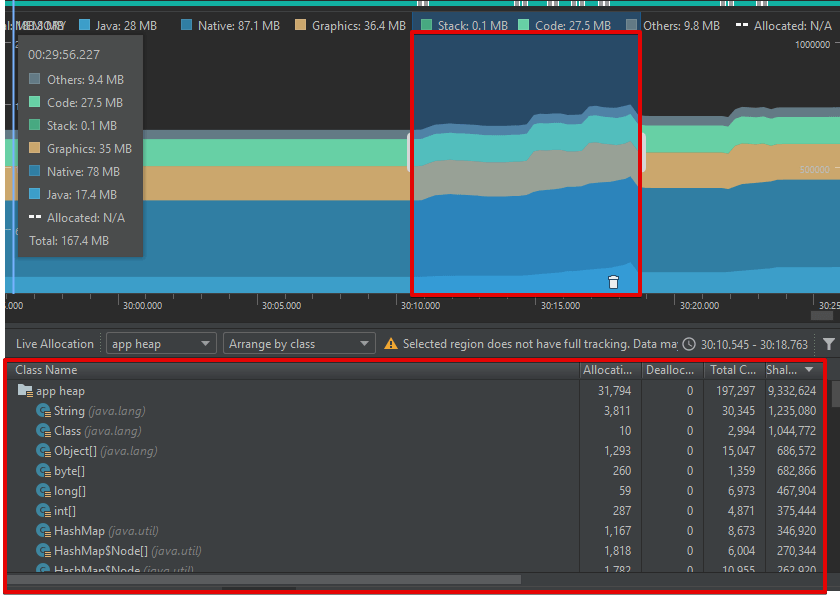
*Профилировщик памяти*

Как показано на рисунке, представление по умолчанию для профилировщика памяти включает следующее:

1. Кнопка для запуска события сборки мусора.
2. Кнопка для захвата HeapDump .
3. Выпадающее меню, чтобы указать, как часто профилировщик фиксирует выделение памяти.
4. Кнопки для увеличения/уменьшения масштаба временной шкалы.
5. Кнопка для перехода к данным в оперативной памяти.
6. Временная шкала событий, которая показывает состояния активности, события пользовательского ввода и события поворота экрана.
7. Временная шкала использования памяти, которая включает в себя следующее:
   * Сложенный график того, сколько памяти используется каждой категорией памяти, как показано осью Y слева и цветовой клавишей вверху.
   * Пунктирная линия указывает количество выделенных объектов, как указано по оси Y справа.
   * Значок для каждого события сборки мусора.

#### **Утечки памяти в Android < 8.0**

В устройствах под управлением Android версии меньше 8.0 нужно запустить allocation tracker, выполнить действия, которые приводят к утечке и остановить трекер. Вам будет показано какие объекты успели создастся.



#### **Утечки памяти в Android >= 8.0**

Все тоже самое, что описано выше, при этом Allocation Tracker включен всегда. Просто выбираете участок на графике, который нужно исследовать и получаете объекты, созданные в этот промежуток.

Если вы только приступили к поиску утечек, весьма полезно в совокупности с Memory Monitor запустить **Monkey**. Программа прекрасно подходит для стрессового тестирования. Она рандомно перемещается по приложению и генерирует пользовательские события: тап, клик, жест.

#### **Основное использование Monkey**

Вы можете запустить Monkey с помощью командной строки на своем компьютере или из скрипта.

Основной синтаксис:

$ adb shell monkey [options] <event-count>

Если параметры не указаны, Monkey запустится в тихом (не подробном) режиме и будет отправлять события любым (и всем) пакетам, установленным на вашей цели. На примере ниже представлена типичная команда, которая запустит ваше приложение и отправит ему 500 псевдослучайных событий:

$ adb shell monkey -p your.package.name -v 500

### Поиск утечек памяти в iOS-приложении:

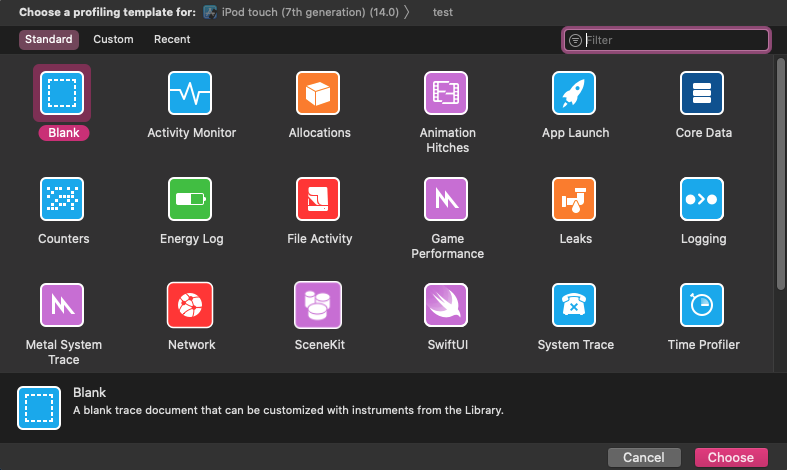
#### **Существует два вида утечек:**

1. **True memory leaks** (действительные утечки памяти), где на объект больше ничто не ссылается, но он по-прежнему занимает место - это означает, что память не может быть использована повторно. Даже в Swift и ARC, наиболее распространенный вид утечки памяти это retain cycle (сохраненный цикл) или strong reference cycle (цикл сильных ссылок). Этот цикл образуется, когда два объекта удерживают сильные ссылки друг на друга, так что каждый объект сохраняет другого от того, чтобы стать освобожденным. Это означает, что их память никогда не освободится!
2. **Unbounded memory growth** (Неограниченный рост памяти), где память по-прежнему распределяется и не освобождается. Если это продолжается вечно, то в какой-то момент система памяти будет заполнена, и у вас возникнет проблема с памятью. В iOS это означает, что приложение будет убито системой.

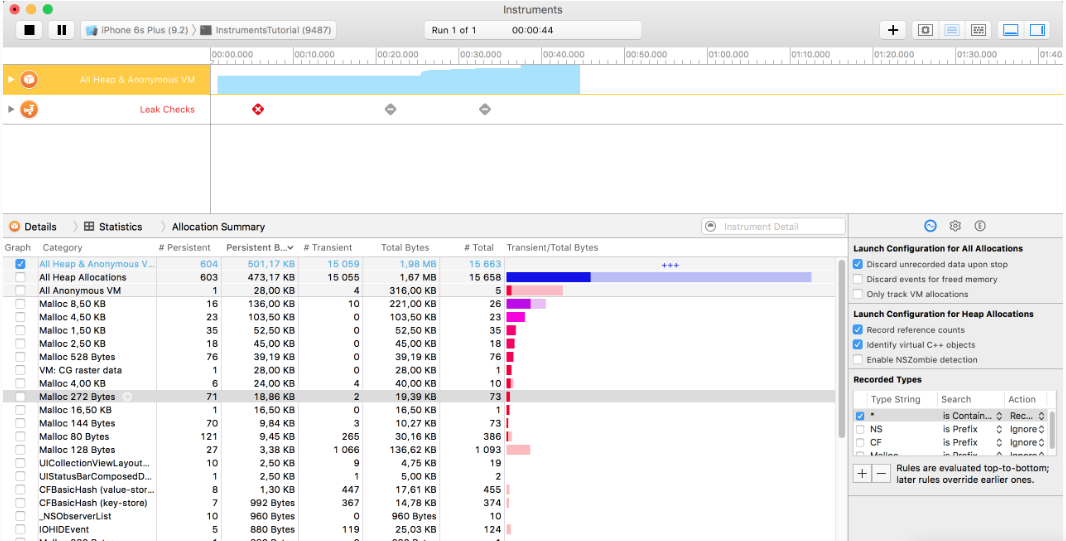
#### **Allocations**

Первый инструмент, который мы рассмотрим, это Allocations. Он даст вам подробную информацию обо всех объектах, которые были созданы, и о памяти, которую они заняли. Также данный инструмент показывает количество ссылок на каждый объект.

Начнём профилирование нашего приложения с новым профилем. Выберите пункт меню Product - Profile. Откроется Instruments. Затем выберите профиль Allocations и нажмите кнопку Choose.



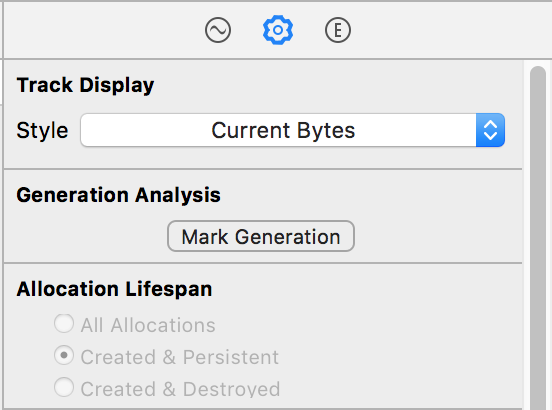
Вы увидите следующее окно:



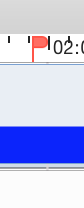
С работающим в приложении инструментом Allocations, создайте пять различных поисковых запросов, но пока не углубляйтесь в результаты. Убедитесь, что поиски дали результаты. Дайте приложению "догрузиться", подождав несколько секунд.

Вы должны были заметить, что диаграмма в треке Allocations начала расти. Это говорит о том, что начала распределяться память. Этот график вам говорит о том, что вы, вероятно, нашли утечку тип "неограниченного роста памяти".

Для подробного анализа - "Generation analysis" ("генерационный анализ"), необходимо нажать кнопку под названием Mark Generation. Вы найдете кнопку в верхней части Display Settings:



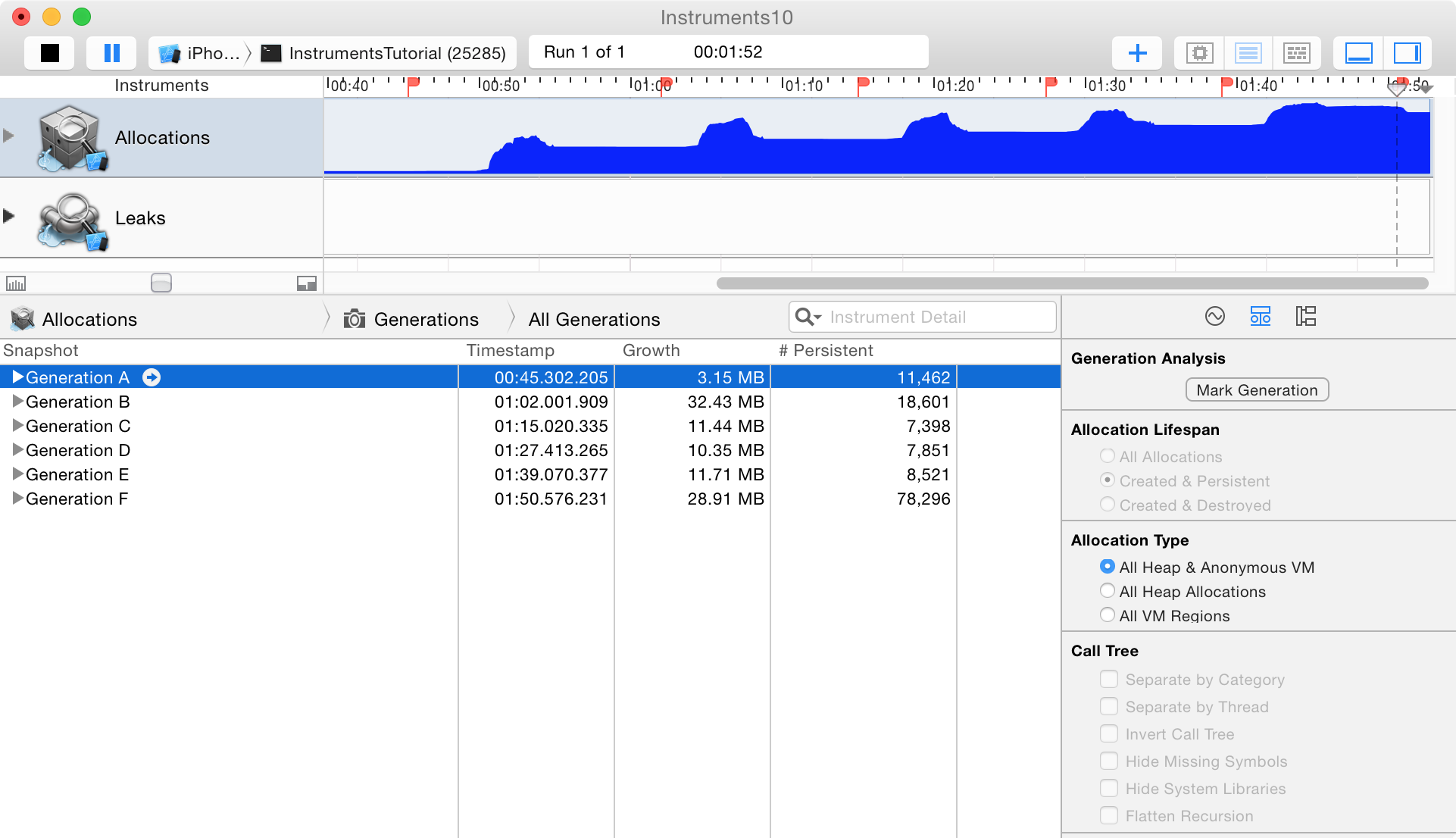
Нажмите ее и вы увидите как на треке появится красный флажок:



Целью генерационного анализа является выполнение действия несколько раз подряд для того, чтобы можно было увидеть, начинает ли потребление памяти постоянно и неограниченно расти.

Зайдите в поиск, подождите несколько секунд для того, чтобы все изображения загрузились, а затем вернитесь на главную страницу. Снова нажмите Mark Generation. Повторите эти действия несколько раз для различных поисковых запросов.

После того, как вы несколько раз запустите поиск, Инструменты будут выглядеть следующим образом:



Здесь вы должны почувствовать неладное. Обратите внимание, что строчка, выделенная синим, идет вверх с каждым поисковым запросом. Это, безусловно, не очень хорошо. Но подождите, что же с "предупреждениями о проблемах с памятью"? "Предупреждения о проблемах с памятью" являются способом iOS сообщить приложению о том, что памяти не хватает или она ограничена, и вы должны ее почистить.

Вполне возможно, что этот рост связан не только с вашим приложением, но также это может быть что-то в глубинах UIKit, что держит память. Дайте приложению шанс очистить память, прежде чем искать причину.

Сымитируйте предупреждение о проблемах с памятью, выбрав Instrument\Simulate Memory Warning на панели Инструментов (или Hardware\Simulate Memory Warning на панели симулятора). Вы заметите, что использование памяти ослабевает совсем немного или возможно вообще остается без изменений. На это не стоит обращать внимание, искать неограниченный рост памяти надо где-то еще.

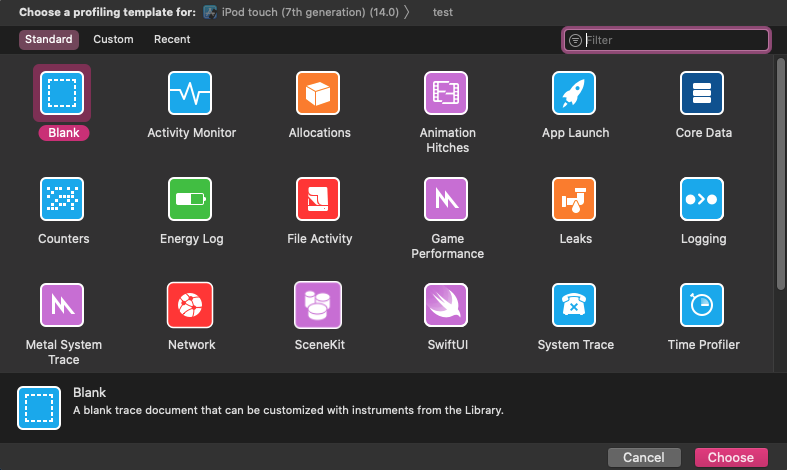
Причина маркировки генерации после каждой итерации поисковых запросов - это возможность увидеть, что память была распределена между каждой генерацией.

#### **Leaks**

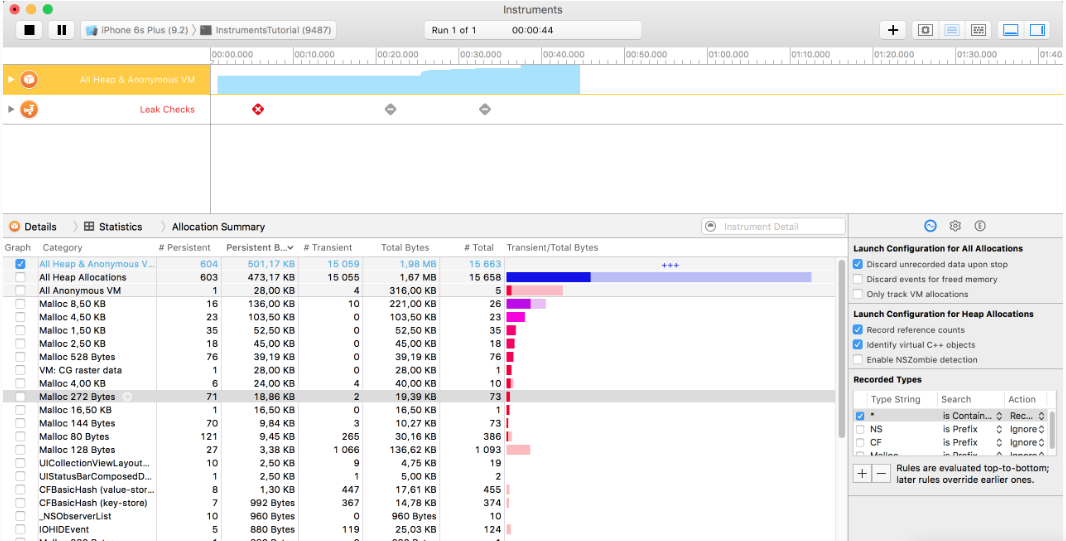
Следующий профиль, который мы рассмотрим это Leaks. Он используется, чтобы найти утечки памяти первого рода (когда на объект больше нет ссылок, а он занимает память).

Профиль Leaks помнит все объекты, под которые была выделена память и периодически просматривает каждый объект, чтобы определить, какие могут быть недоступны из любого другого объекта.

Закройте Instruments, вернитесь в Xcode и выберите пункт меню Product Profile. Выберите профиль Leaks и нажмите кнопку Profile:



В окне Instruments 2 трека — Allocations и Leaks. Трек Allocations тот же самый, что вы использовали в предыдущем разделе.



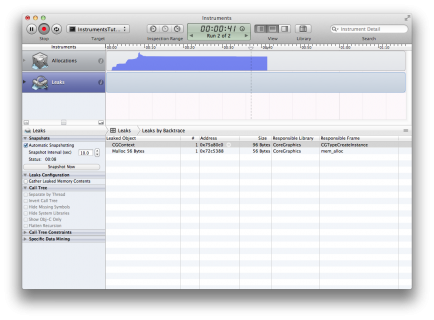
Мы будем использовать только трек Leaks в этом разделе, поэтому щелкните по нему, чтобы его выделить. Обратите внимание, что содержимое других окон Instruments после этого изменилось и теперь оно показывает информацию о треке Leaks.

В информационной панели, внизу слева, уже включён флаг Automatic Snapshotting. Это значит, что снимки памяти для обнаружения утечек памяти будут создаваться автоматически.

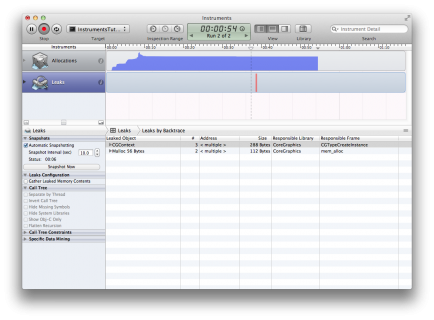
Интервал между снимками может быть изменён, при этом 10 секунд, поставленных по умолчанию, вполне достаточно для наших целей. Также можно сделать снимок в любое время, нажав на кнопку «Snapshot Now».

Для примера запустите поиск в приложении и войдите в результаты поиска. Затем нажмите на одну из строк с результатом, чтобы увидеть изображение на полный экран. Понажимайте несколько раз кнопку Rotate в левом верхнем углу.

Вернитесь в Instruments. Если вы сделали все эти шаги правильно, вы заметите появление утечки! Окно Instruments будет выглядеть следующим образом:



Вернитесь в симулятор iOS и понажимайте кнопку Rotate ещё пару раз. Вернитесь в Instruments и подождите. Появятся ещё утечки, окно теперь выглядит примерно так:



# Производительность отрисовки UI

Существует базовое условие для скорости прорисовки содержимого вашего экрана. Это 16 мс на 1 кадр. Почему 16 мс?

***1 frame / 60fps ~ 16ms***

*Frame - кадр*

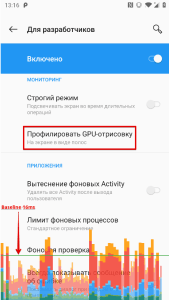
*Fps - fraps per seconds (кадры в секунду)*

*Ms - отклик*

Примерно такое количество необходимо, чтобы общая frame rate (частота кадров) достигала комфортного значения в 60 кадров в секунду.

## Проверка производительности UI Android-приложения

Для того, чтобы проверить скорость отрисовки вашего интерфейса не обязательно расставлять везде логи. В Android есть стандартные методы замера производительности UI. Включить debug можно в настройках для разработчиков на вашем телефоне. Опция «Профилировать GPU-отрисовку». Для наглядного представления выбираем «На экране в виде полос».



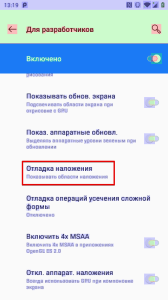
*Профилирование GPU отрисовки*

Все что ниже зеленой линии — укладывается в 16 мс.

### Оптимизация Layout

Чем больше раз мы перерисовываем один и тот же пиксель, тем хуже производительность. Это может происходить, когда в верстке есть много вложенных групп и элементов. Например сначала рисуется фон, затем поверх рисуется фон карточки, далее сам элемент. Чем больше вложенности — тем хуже.

Посмотреть области наложения и перерисовки можно выбрав пункт «Отладка наложения» в инструментах разработчика на устройстве.



*Отладка наложения*

Проблема с наложениями — одна из причин, почему активно разрабатывался Constraint Layout и другие вещи, которые оптимизируют UI.

## 

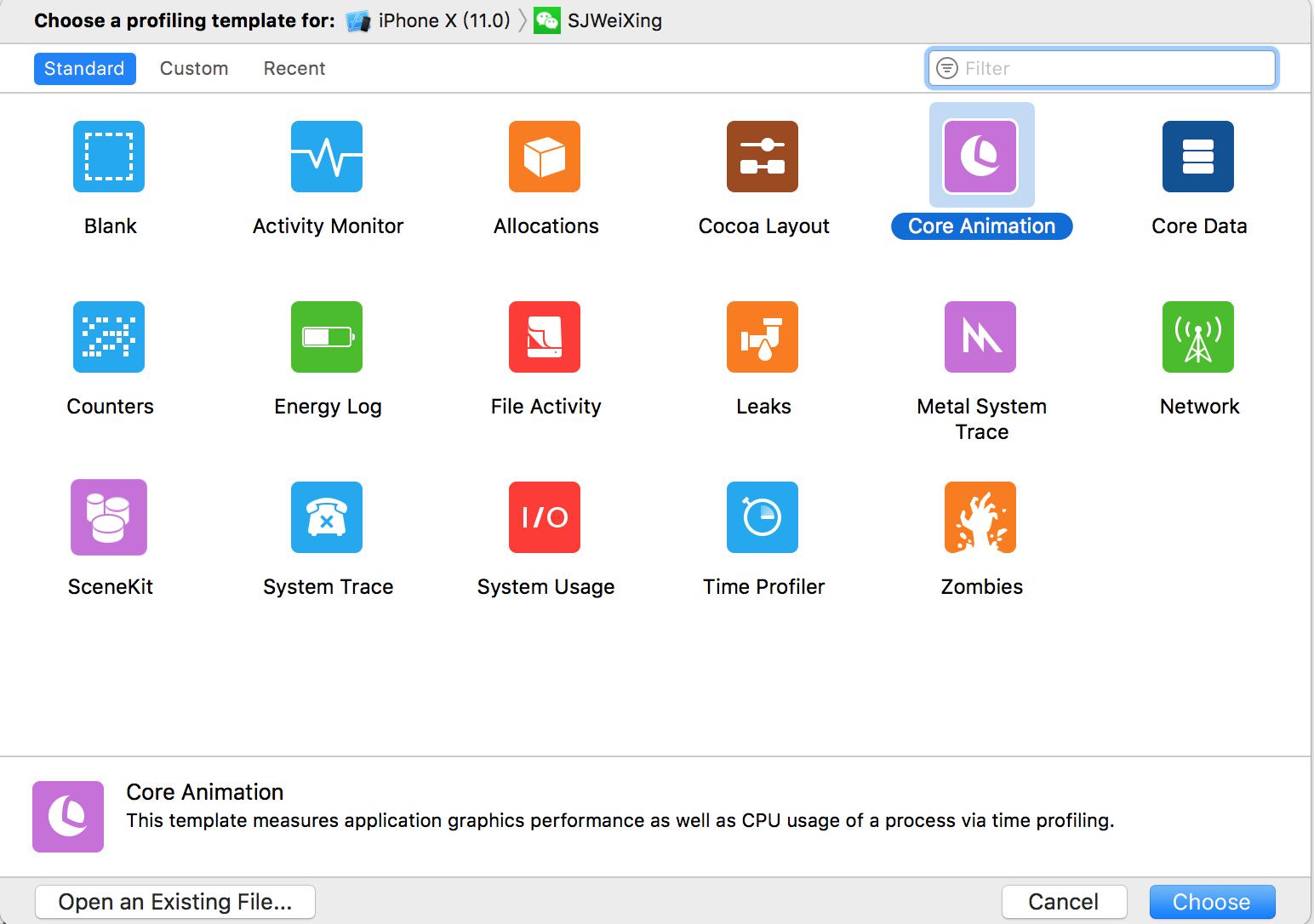
## Проверка производительности UI iOS-приложения

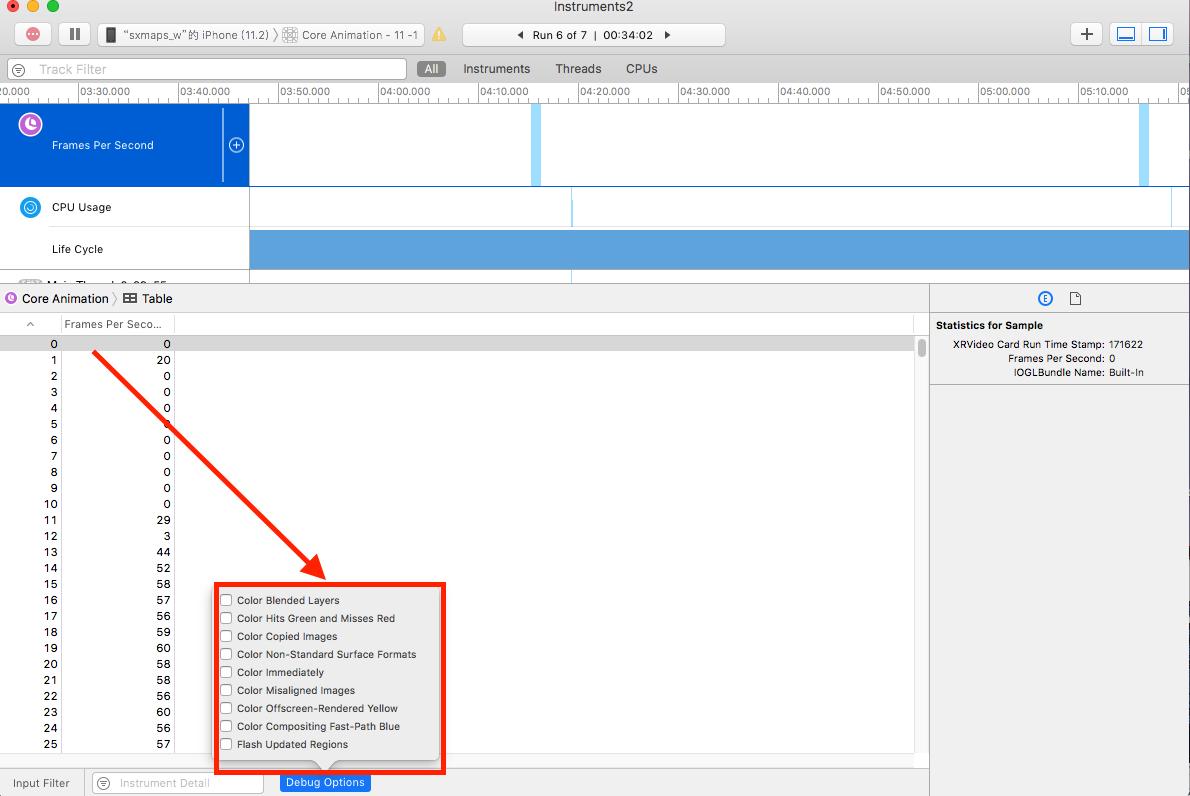
**Core Animation (CA)** приложение для iOS — пресет в профилировщике, который используется для измерения FPS (кол-во кадров в секунду), для того чтобы увидеть, лагают анимации или нет.

Зачастую, даже если найдены проблемные места приложения, трудности с производительностью остаются. Причина в том, что при работе с UI-фреймворками, используется UIView, но под капотом создается экземпляр CATransaction (или система делает это сама), и все эти инструкции передаются серверу на обработку. Сервер для рендеринга отвечает за создание анимации. Если анимация выполняется с помощью UIView, например, классового метода animate(withDuration:animations:), она обрабатывается Сервером для рендеринга, который считается отдельным потоком и работает со всеми анимациями в приложении.

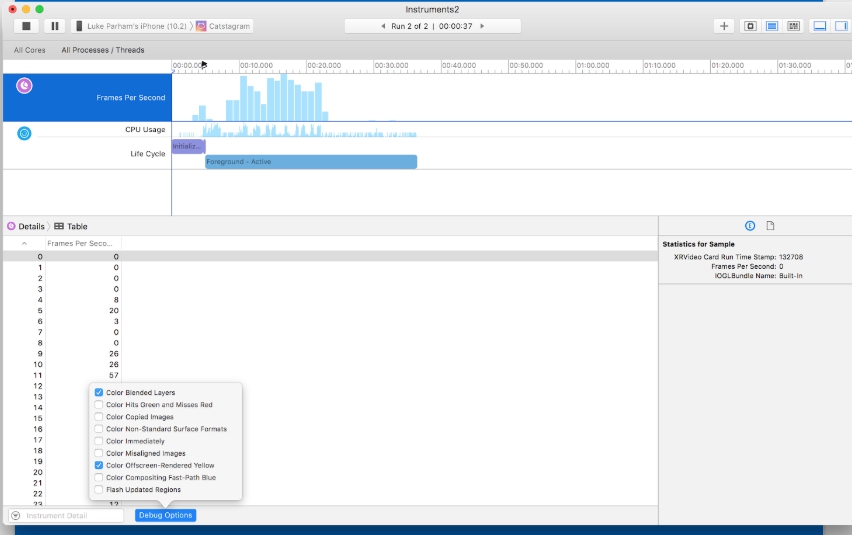
Запустите программу:

нажмите XCode, кликните по левому верхнему углу - XCode ➝ Открыть инструмент разработчика ➝ Инструменты ➝ откройте Инструменты и выберите Core Animation:



*Базовый интерфейс анимации*

Можно заставить Сервер для рендеринга работать медленно, чтобы он не появлялся в Time Profiler, но он всё равно будет тормозить работу приложения. Вот как это выглядит:



Вверху расположен датчик частоты кадров. Внизу расположена самая важная часть — параметры отладки на графике. Низкое значение говорит нам о том, что приложение скорее всего не успеет отрисоваться.

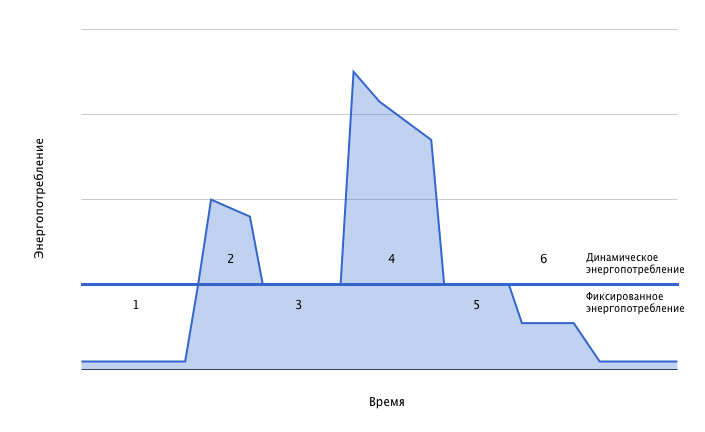
# Потребление заряда аккумулятора

Основные потребители батареи - графика, блютуз, геолокация, подсветка, сети - если МП затрагивает один их этих показателей, то необходимо протестировать затраты батареи.

### На что же мы можем повлиять, чтобы уменьшить потребление батареи приложением?

* Прежде всего уменьшить нагрузку на CPU. Это главный способ уменьшить расход батареи. Имеет смысл спрофилировать приложение на использование CPU. Если возможно, перенести часть нагрузки на GPU. Это может существенно повлиять на расход заряда.
* Сеть. Данные имеет смысл буферизовать, чтобы реже обращаться к сети. [Здесь](https://developer.android.com/training/efficient-downloads/index.html) подробно описано, как уменьшить потребление батареи при передачи данных по сети.
* Геолокация. Если не требуется точная, часто обновляемая координата, необходимости в её постоянном запросе нет. Стоит рассмотреть возможность определения координаты по WiFi. [Здесь](http://developer.android.com/guide/topics/location/strategies.html#Adjusting) хорошо написано про это.

### Как потребляется энергия при выполнении работы?



*Энергопотребление при выполнении работы*

На рисунке видно, как потребляется энергия, когда устройство выполняет работу. Также видим разделение на две части: фиксированное и динамическое энергопотребление.

Фиксированное энергопотребление — энергия, которая нужна, чтобы поддерживать работу системы, активировать передатчики данных (будь то сотовая связь, Wi-Fi, GPS) и т.д.

Динамическое энергопотребление — количество энергии, которое необходимо для выполнения полезной работы.

Пройдемся по состояниям с рисунка для большей ясности:

1. Система находится в состоянии простоя — минимальное потребление энергии. Даже в состоянии простоя девайс потребляет энергию, чтобы оставаться отзывчивым.
2. Происходит некая активность, например, пришло PUSH-уведомление.
3. Простой, энергопотребление уменьшилось после того, как PUSH-уведомления обработалось.
4. Опять некая активность, например, пользователь нажал на уведомление, и запустилось приложение. Оно отобразило полную информацию по уведомлению, после чего пользователь закрыл приложение.
5. Простой, энергопотребление опять снизилось. Отметим, что это состояние идентично состоянию 3, вся разница в том, что после состояния 3 опять произошла активность, которая увеличила энергопотребление.
6. Энергопотребление продолжает уменьшаться, и система опять входит в состояние простоя.

О чем это говорит? Когда необходимо выполнить работу, энергопотребление быстро увеличивается, чтобы немедленно выполнить задачу. После работы требуется значительно больше времени для спада энергопотребления. Падение происходит в состоянии простоя, на вхождение в которое требуется время.

## Что делать на **Android** для снижения энергопотребления?.

**App StandBy** позволяет системе определить, что приложение простаивает, когда пользователь не пользуется им активно. App StandBy запускается, когда не выполняется ни одно из следующих условий:

* Пользователь явно запускает приложение.
* Приложение находится на переднем плане (явно или в качестве Foreground service, либо используется другой Activity).
* Приложение генерирует уведомления, которые пользователь видит на экране блокировки или в области уведомлений.
* Приложение является активным приложением администратора устройств.

Когда устройство подключается к зарядке, система выпускает приложения из режима Standby, что позволяет им выполнять любые задачи. Если устройство не используется в течение длительного периода времени, система предоставляет бездействующим приложениям доступ в сеть примерно раз в день.

[App Standby Buckets](https://developer.android.com/topic/performance/appstandby) помогает системе приоритизировать запросы приложений к ресурсам на основании того, как давно и как часто использовалось приложение. На основе шаблонов использования приложение помещается в один из пяти сегментов. Система ограничивает ресурсы устройства, доступные для каждого приложения, в зависимости от того, в каком сегменте находится приложение.

Пять сегментов, назначаемые приложениям в зависимости от приоритета:

* **Active**. Приложение находится в активном сегменте, если пользователь в настоящий момент использует приложение. Т.е. если видна Activity, или запущен Foreground service, или есть synchronized adapter, связанный с приложением на переднем плане, или пользователь перешел по уведомлению. Если приложение в активном сегменте, то никакие ограничения на использование ресурсов устройства не накладываются.
* **Working set**. Приложение находится в этом сегменте, если часто запускается, но в данный момент не активно. Система накладывает умеренные ограничения на действия этого приложения.
* **Frequent**. Приложение находится в этом сегменте, если используется часто, но не каждый день. Система накладывает больше ограничений, также накладываются ограничения на количество сообщений FCM с высоким приоритетом.
* **Rare**. Приложение находится в этом сегменте, если оно редко используется. В этом случае система накладывает строгие ограничения и на получение сообщений FCM с высоким приоритетом. Система также ограничивает возможность приложения подключаться к интернету.
* **Never**. Это сегмент для приложений, которые были установлены, но никогда не запускались. Система накладывает жесткие ограничения.

### Тестирование App Standby Buckets

Можно вручную переместить приложение в определенный App StandBy bucket с помощью команды:

adb shell am set-standby-bucket <package\_name> active|working\_set|frequent|rare

Команда проверки, в каком сегменте сейчас приложение:

adb shell am get-standby-bucket <package\_name>

Тестирование ограничений на фоновые процессы:

* Вручную применить ограничения на выполнение фоновых задач:

adb shell cmd appops set <package\_name> RUN\_ANY\_IN\_BACKGROUND ignore

* Убрать ограничения на выполнение фоновых процессов:

adb shell cmd appops set <package\_name> RUN\_ANY\_IN\_BACKGROUND allow

## Что делать на **iOS**, для снижения энергопотребления?

### 1. Избегать выполнения работы в состояние inactive

Если приложение не отображается в данный момент (например, перекрыто другим приложением), останавливайте таймеры, потоки, работу по сети, прекращайте перерисовку экрана. Для этого достаточно подписаться на делегаты

* applicationWillResignActive(\_:)
* applicationDidBecomeActive(\_:)

Они будут вызваны в моменты перехода приложения из foreground и возвращения в него. Соответствующие нотификации есть у *NSNotification*.

### 2. Совершать работу в оптимальный момент времени

Наилучший момент — тот, во время которого на выполнение работы будет потрачено минимальное количество заряда. Разработчик не знает, какой момент времени является наилучшим для выполнения задачи, но об этом знает система, которая сама может запланировать выполнение.

Это может быть полезно, когда необходимо скачивать или отправлять значительные объемы данных, периодически обновлять информацию или выполнять другие периодические действия по сети.

Для планирования работы по сети существует метод: URLSessionConfiguration.background(withIdentifier). С его помощью можно создавать запросы, а обрабатывать результаты уже при следующем запуске приложения. Также поддерживается автоматическое повторение запроса в случае неудачи. При создании задачи система сама выбирает наиболее благоприятный момент для выполнения.

Когда рекомендуется использовать *background* конфигурацию:

1. Автоматическое сохранение
2. Обработка данных
3. Подкачка контента
4. Выполнение чего-либо с интервалом 10 минут и более

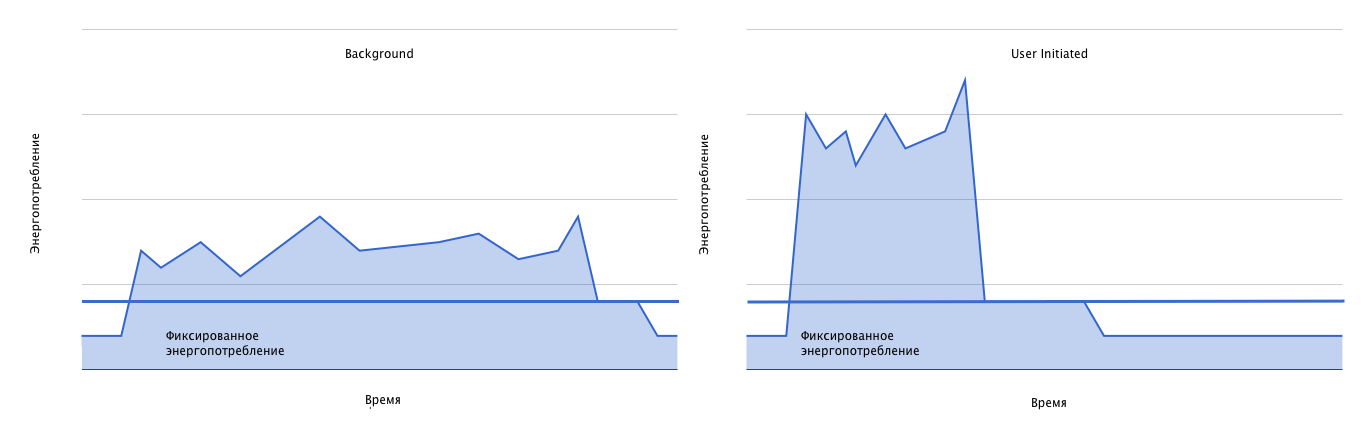
*Пример:*

Необходимо загрузить много данных для дальнейшей работы в offline. Можно будет обработать результаты запросов в следующий раз, когда система или пользователь решит продолжить работу приложения, так как все эти запросы выполняются на системном потоке и результаты их точно придут в приложение.

### 3. Выполнять работу эффективнее

Допустим, работа уже выполняется, даже в наилучшее время. Но также важно задать этой работе соответствующий приоритет. В системе iOS есть системные очереди, и у каждой из них свой приоритет на владение ресурсами. Таким образом можно регулировать, сколько процессорного времени и прочих ресурсов будет выделено для выполнения задачи.

* User Interactive — самая приоритетная очередь. Еще ее называют main или ui, потому что на этой очереди выполняются задачи по отрисовке интерфейса.
* User Initiated — следующая по приоритету. Используется для выполнения задач, завершения которых ожидает пользователь (загрузка контента, обработка изображений).
* Utility — долгие задачи. Пользователь знает, что такие задачи выполняются, и может ждать их завершения.
* Background — самый низкий приоритет. Тут разработчику стоит задуматься, а можно ли отсрочить выполнение задачи? Если да, то используем URLSessionConfiguration.background(withIdentifier).



*Энергопотребление на разных очередях*

Посмотрим на энергопотребление при выполнении одинакового кода на разных очередях. Динамическое энергопотребление будет одинаковым, но суммарное будет меньше, поскольку, сократив скорость выполнения, мы уменьшили фиксированные затраты. То есть суммарное потребление уменьшилось из-за уменьшения фиксированного. Более эффективный код всегда менее энергозатратен.

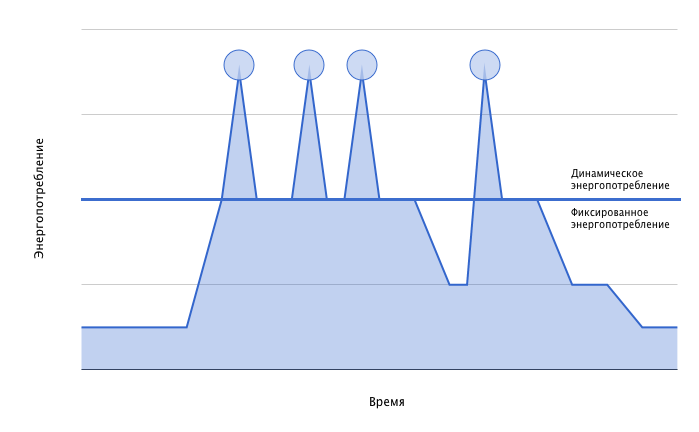
### 4. Совершать меньше работы

Для это следует улучшать производительность по четырем направлениям:

#### **1. CPU**

Сначала надо определить код, активно использующий CPU. В этом помогут XCode Instruments. Обычно такой код работает с таймерами и вызовами, которые останавливают поток: *NSTimer*, *GCD timers*, *performSelector(withObject, afterDelay)*, *CFRunLoopTimer*, *sleep()*, *pthread\_cond\_timedwait()*, *dispatch\_semaphore\_wait()*.

Рассмотрим этот процесс на примере таймера.



*Энергопотребление при работе таймера*

Видим накладные расходы, которые не успевают снизиться из-за частых срабатываний таймера. Чтобы улучшить производительность при использовании таймера, используйте setTolerance. Это позволит выполнять срабатывания в оптимальное время, выбранное системой.

myTimer.setTolerance(60.0) // в промежутке 60 секунд будет вызван обработчик

#### **2. Отрисовка графики**

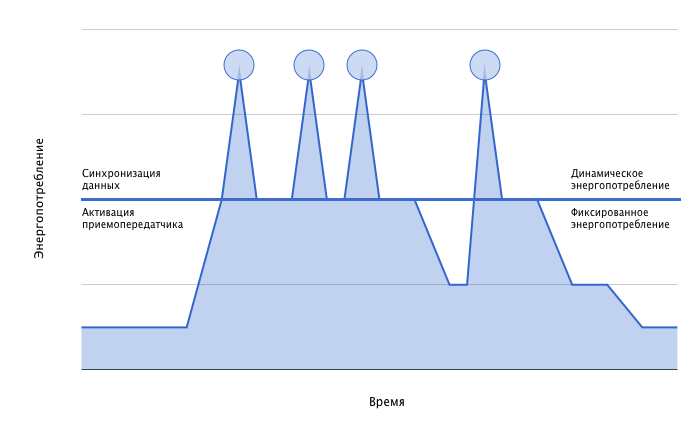
Чтобы сэкономить энергию, не совершайте лишних действий. Например, вместо вызова setNeedsDisplay, который перерисует все *View*, вызовите setNeedsDisplayInRect. Избегайте наложения блюров на часто перерисовывающиеся объекты, такое наложение заставит блюр постоянно перерисовываться. Также избегайте лишних перерисовываний, выводящих энергопотребление из спящего уровня. Мониторить отрисовку можно через Quartz Debug или Instruments.

#### **3. Взаимодействие с хранилищем**

* Помните, что операции записи потребляют гораздо больше энергии, чем операции чтения
* Группируйте операции записи, чтобы уменьшить фиксированное энергопотребление
* Используйте кэширование

#### **4. Приемопередатчики сигналов**

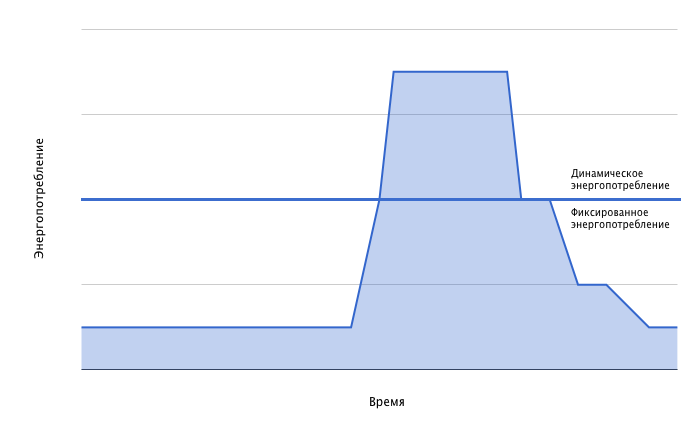
Различные виды передатчиков (мобильная связь, Wi-Fi, Bluetooth, GPS) часто используются в приложениях. Обычно обращения к ним происходят, как только это инициировал пользователь или появились данные для отправки. Посмотрим на график энергопотребления при таком подходе.



*Энергопотребление при активном взаимодействии по сети с интервалом*

Из-за фиксированных расходов на активацию передатчика, энергопотребление остается высоким все время, когда периодически происходит отправка данных. Стоит также учитывать, что расход энергии зависит от типа приемопередатчика, которым мы пользуемся. Например, на iPhone 5S при использовании Wi-Fi батарея выдерживает 10 часов серфинга, а с мобильной связью — 8 часов.

Если есть возможность, буферизуйте данные и отправляйте их все в один момент времени, как на графике ниже.

 Энергопотребление при активном сетевом взаимодействии в один промежуток времени

А еще не стоит забывать про размер данных, которые отправляет и принимает приложение. От этого зависит время на протяжении которого будет работать приемопередатчик, чем меньше размер, тем быстрее он выключится. Поэтому стараемся все сжимать и кэшировать.

# Производительность клиент-серверного приложения (чем плох лишний трафик)

Когда ваша страница загружается в приложении, на сервер отправляется HTTP-запрос для страницы в URL. После того, кака HTML доставился, браузер анализирует его и ищет дополнительные запросы для изображений, сценариев, CSS, Flash, JavaScript и так далее. Каждый раз, когда он видит запрос на новый элемент, он отправляет другой HTTP-запрос на сервер.

Чем больше изображений, скриптов, CSS, Flash и т. д. на вашей странице, тем больше будет запросов и тем медленнее будут загружаться ваши страницы. При этом **наблюдается увеличение количества процессов, одновременно находящихся в оперативной памяти и занятых передачей сформированных данных, что влияет на производительность всех составных частей информационной системы - клиента, сервера и сети.**

Поэтому важно чтобы количество запросов приложения было минимизировано (не было беспричинных запросов на сервер) и при этом не терялась цель актуализации получаемой конечным юзером информации

# Время запуска приложений

### Зачем сокращать время запуска?

*“В картах у пользователя в большинстве случаев следующий сценарий: на ходу достать телефон, запустить приложение, быстро определить, где он находится, куда необходимо идти, и убрать телефон”*

*Похожих сценариев, когда приложение открывают ненадолго, много. Поэтому нам очень важно, чтобы приложение запускалось быстро.*

Очевидно, что из двух приложений с одинаковым набором функций пользователь выберет то, которое быстро запускается. Если альтернативы нет, а приложение запускается долго, пользователя это будет раздражать, он будет реже возвращаться в ваше приложение, писать плохие отзывы и, наоборот, если приложение запускается быстро, то все будет здорово.

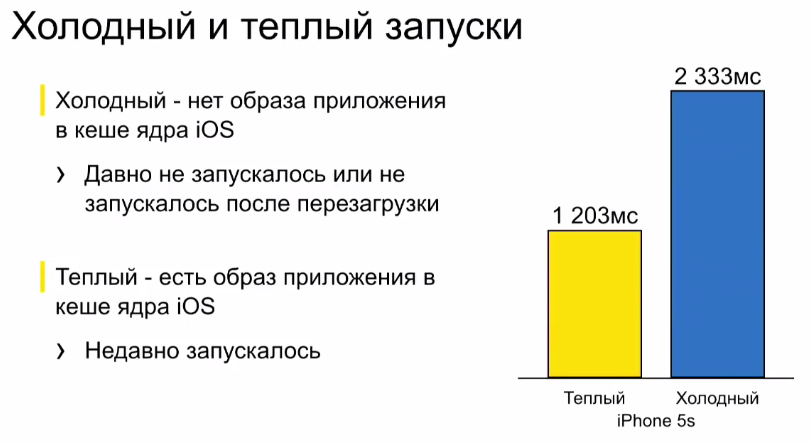
При этом не стоит забывать и про ограничение времени запуска в 20 секунд, при превышении которого система прерывает загрузку вашего приложения. На слабых устройствах эти 20 секунд достаточно реально превысить.

## Как рассчитать время запуска приложения на iOS?

Холодный запуск — это когда приложение давно завершило свою работу и было удалено из кэша операционной системы. Холодный запуск всегда происходит после перезагрузки приложения, в связи с чем рекомендуется моделировать его в своих тестах.

Теплый запуск — это когда приложение было завершено недавно.

В случае Яндекс.Карт оно скачет в два раза:



В случае холодного запуска этот pre-main может быть на порядок дольше, чем в случае теплого запуска.

## 

И как раз за счет этого у нас и получается такая большая разница между холодным и теплым запуском.

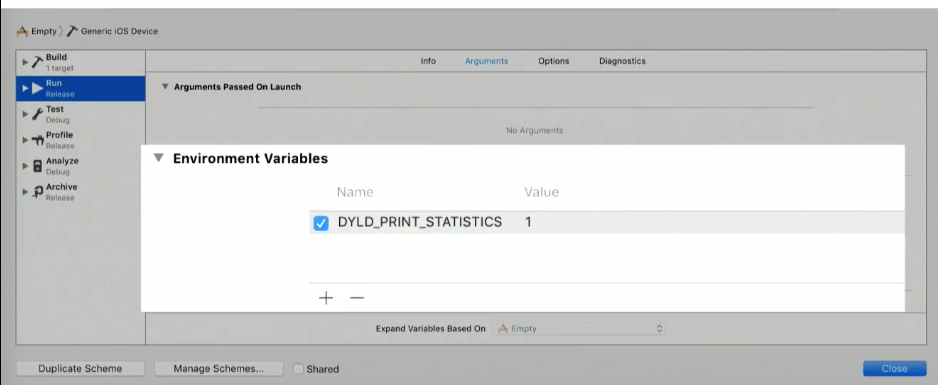
Таким образом, когда вы измеряете запуск своего приложения, вы должны обязательно учитывать не только тот отрезок, где ваш код работает, но и pre-main, когда система собирает веб-приложение, а также учитывать холодный запуск.

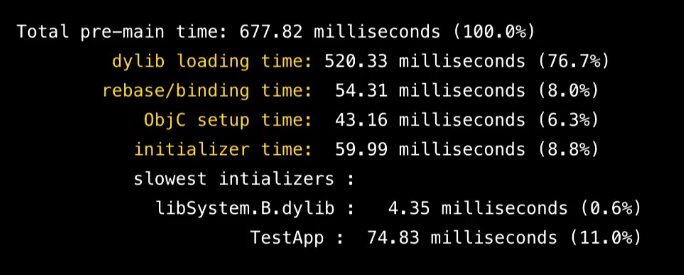
### Замер pre-main

Замер pre-main — нетривиальная задача, поскольку наш код там не работает. К счастью, в iOS 9+ Apple добавила переменную окружения DYLD\_PRINT\_STATISTICS, при включении которой в консоль выводится статистика работы системного загрузчика.

### Просмотр Времени запуска приложения в Xcode:

Схема редактирования ➞ Run ➞ Аргументы ➞ Переменные среды, установив вариативную среду dyld\_print\_statististics до 1, на котором можно увидеть потребление времени каждого этапа до основного





Выводится полное время pre-main и далее поэтапно:

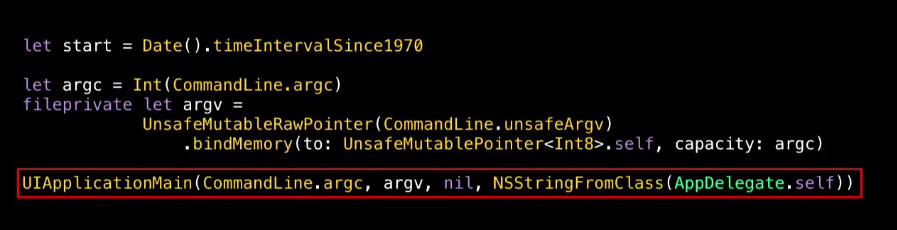
* время загрузки динамических библиотек;
* время rebase/binding — т.е. правки указателей и связывания;
* время создания ObjC контекста;
* время инициализации — там, где +load и глобальные переменные.

### Замер after-main

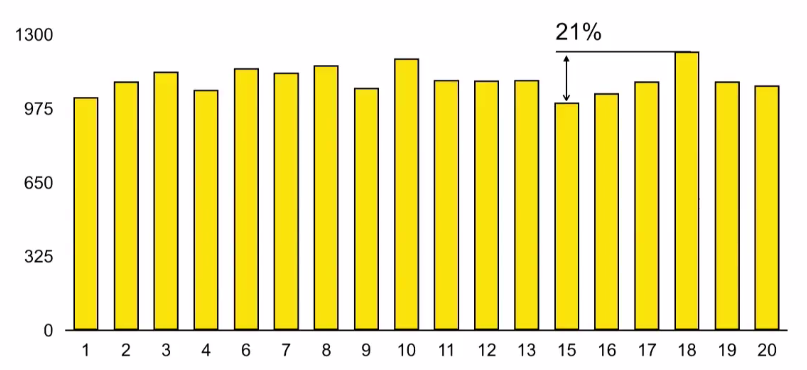
У нас есть удобный инструмент для замера pre-main, теперь нужно правильно померить after-main.

Распространенная ошибка — замерять только didFinishLaunching. Но до didFinishLaunching происходит ещё инициализация UIApplication, UIApplicationDelegate, там у вас могут быть сложные конструкторы, и это всё тоже нужно учитывать. Поэтому время нужно мерить с начала main.

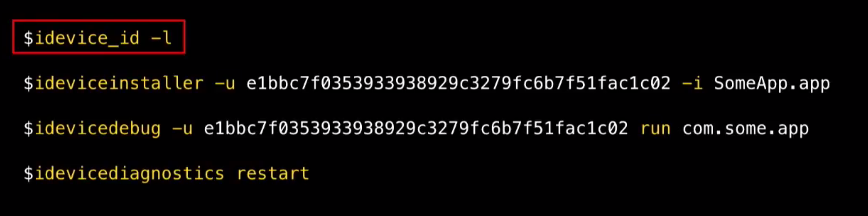
Если у вас в проекте нет файла main.swift, придётся его добавить и первой строчкой поставить замер запуска, а потом уже явно вызывать UIApplicationMain.



Итак, мы научились правильно замерять полное время. Однако даже в одной и той же ситуации время запуска может сильно скакать, и вы этим никак не управляете, потому что устройство в фоне может что-то делать. Скачки разброса могут достигать +-20%.

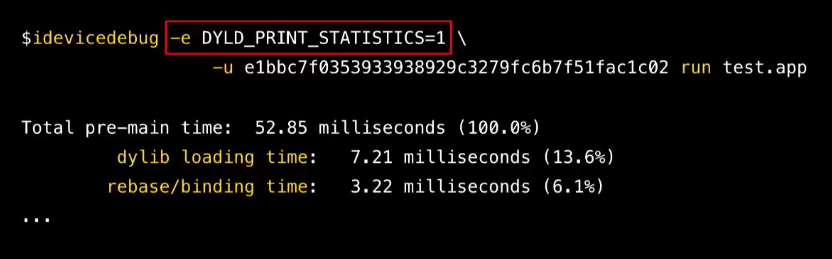


К счастью, утилита libimobiledevice может решить эту проблему. Она взаимодействует с устройством напрямую через те же самые протоколы, что и Xcode и iTunes, при этом она не требует jailbreak. Утилита позволяет делать все, что нам нужно.



Во-первых, позволяет получить список подключенных устройств и их UID. Во-вторых, установить приложение на конкретное устройство, запустить приложение и перезагрузить устройство. Нам это важно, чтобы померить холодный запуск.

Что особенно важно, в запуск приложения можно передать переменную окружения (нам необходимо передавать переменную DYLD\_PRINT\_STATISTICS, чтобы измерить pre-main).



## Как рассчитать время запуска приложения в Android?

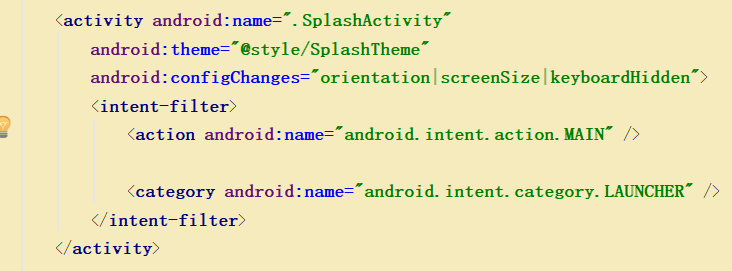
### Статистика команды adb

Первый запуск приложения - это то, что мы часто называем холодным запуском. В настоящее время процесс вашего приложения не создается. Это также сценарий использования большинства приложений. Пользователь тапает по иконке вашего приложения в меню приложений. После этого сначала создается процесс, а затем запускается MainActivity.

Для того чтобы получить время запуска необходимо использовать следующую команду:

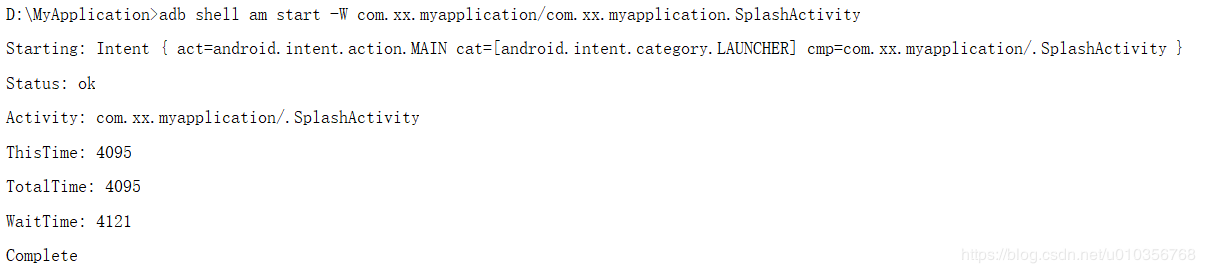
adb shell am start -W packagename/MainActivity по абсолютному пути

Например, имя моего пакета “com.xx.myapplication”, Моя стартовая страница “SplashActivity”

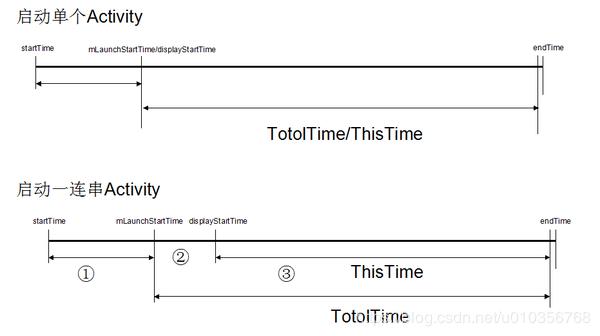


Тогда я должен ввести в Терминале следующую команду:

adb shell am start -W com.xx.myapplication/com.xx.myapplication. SplashActivity



* **ThisTime** представляет время начала последнего действия в серии начатых действий.
* **TotalTime**: общее время, затраченное на запуск серии действий, что подразумевает за собой длительность, необходимую для запуска нового приложения, включая начало нового процесса и начало действия, но не затраченное на предыдущее приложение. Другими словами, разработчикам обычно нужно заботиться только о TotalTime, это время, необходимое для фактического запуска их приложения.
* **WaitTime**: процесс создания процесса приложения + TotalTime - общее время, включая время паузы активности предыдущего приложения и время запуска нового приложения.



* В первый период времени AMS создает блок записи ActivityRecord, выбирает подходящую задачу и приостанавливает текущее действие возобновления.
* Во второй период времени запускается процесс, вызывается onCreate() без интерфейсной активности, приостанавливается / завершается без интерфейсная активность.
* В период времени ③ вызывается onCreate() и onResume() с интерфейсом Activity.
* Если вас интересует только время, необходимое для запуска приложения, обратитесь к TotalTime;
* Если вас интересует время, необходимое системе для запуска приложения, обратитесь к WaitTime;
* Если вас интересует время, необходимое для запуска приложения с интерфейсом, обратитесь к ThisTime.

# Полезные ссылки

* <https://sqadays.com/ru/talk/70736>
* <https://habr.com/ru/post/168491/>
* <https://swiftbook.ru/post/tutorials/tutorial-po-instrumentam-na-swift/>
* <https://www.youtube.com/watch?v=q4gOrXYUJiY>