Поддержка асинхронности | Параллелизм - Dart

- Библиотеки Dart полны функций, которые возвращают Future объекты Stream. Эти функции являются асинхронными : они возвращаются после настройки операции, которая может занять много времени (например, ввода-вывода), не дожидаясь завершения этой операции.
- Ключевые слова async и await поддерживают асинхронное программирование, позволяя писать асинхронный код, похожий на синхронный код.

Почему асинхронный код важен?

Асинхронные операции позволяют приложению завершить работу, ожидая завершения другой операции. Вот некоторые распространенные асинхронные операции:

- Получение данных по сети.
- Запись в базу данных.
- Чтение данных из файла.

Такие асинхронные вычисления обычно предоставляют результат в виде файла Future или, если результат состоит из нескольких частей, в виде файла Stream. Эти вычисления вносят асинхронность в программу. Чтобы учесть эту первоначальную асинхронность, другие простые функции Dart также должны стать асинхронными.

Чтобы взаимодействовать с этими асинхронными результатами, используются ключевые слова async и await. Большинство асинхронных функций — это просто асинхронные функции Dart, которые, возможно, в глубине зависят от асинхронных вычислений.

- синхронная операция : синхронная операция блокирует выполнение других операций до ее завершения.
- синхронная функция : синхронная функция выполняет только синхронные операции.
- асинхронная операция : после запуска асинхронная операция позволяет выполнять другие операции до ее завершения.
- асинхронная функция : асинхронная функция выполняет как минимум одну асинхронную операцию, а также может выполнять синхронные операции.

Что такое Future?

- Future — это экземпляр класса Future. Будущее представляет собой результат асинхронной операции и может иметь два состояния: незавершенное или завершенное.

Незавершенный

- Когда вызывается асинхронная функция, она возвращает незавершенное будущее. Это будущее ожидает завершения асинхронной операции функции или выдачи ошибки.

Завершенный

- Если асинхронная операция завершается успешно, будущее завершается значением. В противном случае он завершается с ошибкой.

Завершение со значением

- Будущее типа Future<T> завершается значением типа T. Например, будущее с типом Future<String> создает строковое значение. Если будущее не создает полезного значения, то тип будущего — Future<void>.

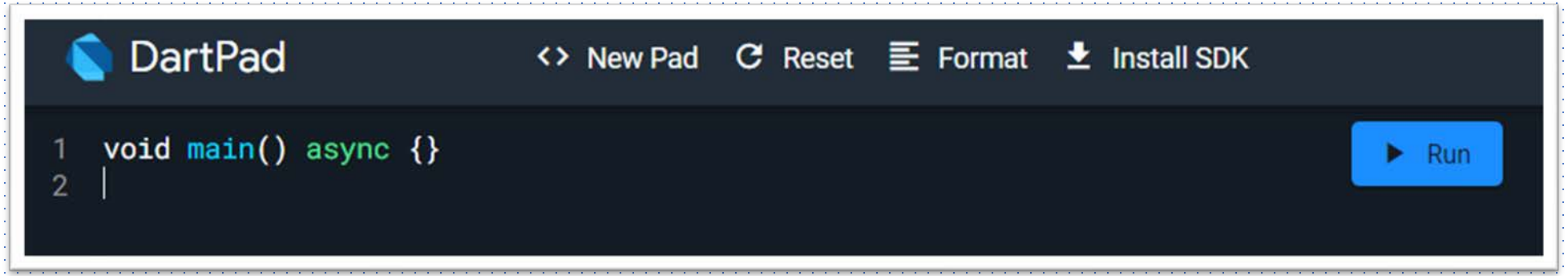
Завершение с ошибкой

- Если асинхронная операция, выполняемая функцией, по какой-либо причине завершается с ошибкой, Future завершается с ошибкой.

Работа с фьючерсами: асинхронность и ожидание

Ключевые слова async и await предоставляют декларативный способ определения асинхронных функций и использования их результатов. Два основных правила при использовании async и await:

- Чтобы определить асинхронную функцию, добавьте async перед телом функции:



- Ключевое await слово работает только в async функциях.

- Если функция имеет объявленный тип возвращаемого значения, необходимо обновить тип чтобы он был Future<T>, где T— тип значения, возвращаемого функцией. Если функция не возвращает значение явно, то тип возвращаемого значения Future<void>:

Параллелизм в Dart

- Dart поддерживает параллельное программирование с помощью async-await, изолятов и таких классов, как Future и Stream.
- Внутри приложения весь код Dart выполняется изолированно . Каждый изолят Dart имеет один поток выполнения и не имеет общих изменяемых объектов с другими изолятами. Чтобы общаться друг с другом, изоляты используют передачу сообщений. Многие приложения Dart используют только один изолят — основной изолят. **Можно** создать дополнительные изоляты, чтобы обеспечить параллельное выполнение кода на нескольких ядрах процессора.
- Хотя изолирующая модель Dart построена на базовых примитивах, таких как процессы и потоки, предоставляемых операционной системой, использование этих примитивов виртуальной машиной Dart является деталью многих реализаций.

Типы будущего и потока

- Язык и библиотеки Dart используют объекты Future и Stream объекты для представления значений, которые будут предоставлены в будущем. Например, обещание в конечном итоге предоставить int значение записывается как Future<int>. Обещание предоставить серию int значений имеет тип Stream<int>.
- В качестве другого примера рассмотрим методы dart:io для чтения файлов. Синхронный File метод readAsStringSync() читает файл синхронно, блокируя его до тех пор, пока файл не будет полностью прочитан или не произойдет ошибка. Затем метод либо возвращает объект типа, String либо выдает исключение. Асинхронный эквивалент readAsString()немедленно возвращает объект типа Future<String>. В какой-то момент в будущем оператор Future<String> завершится либо строковым значением, либо ошибкой.

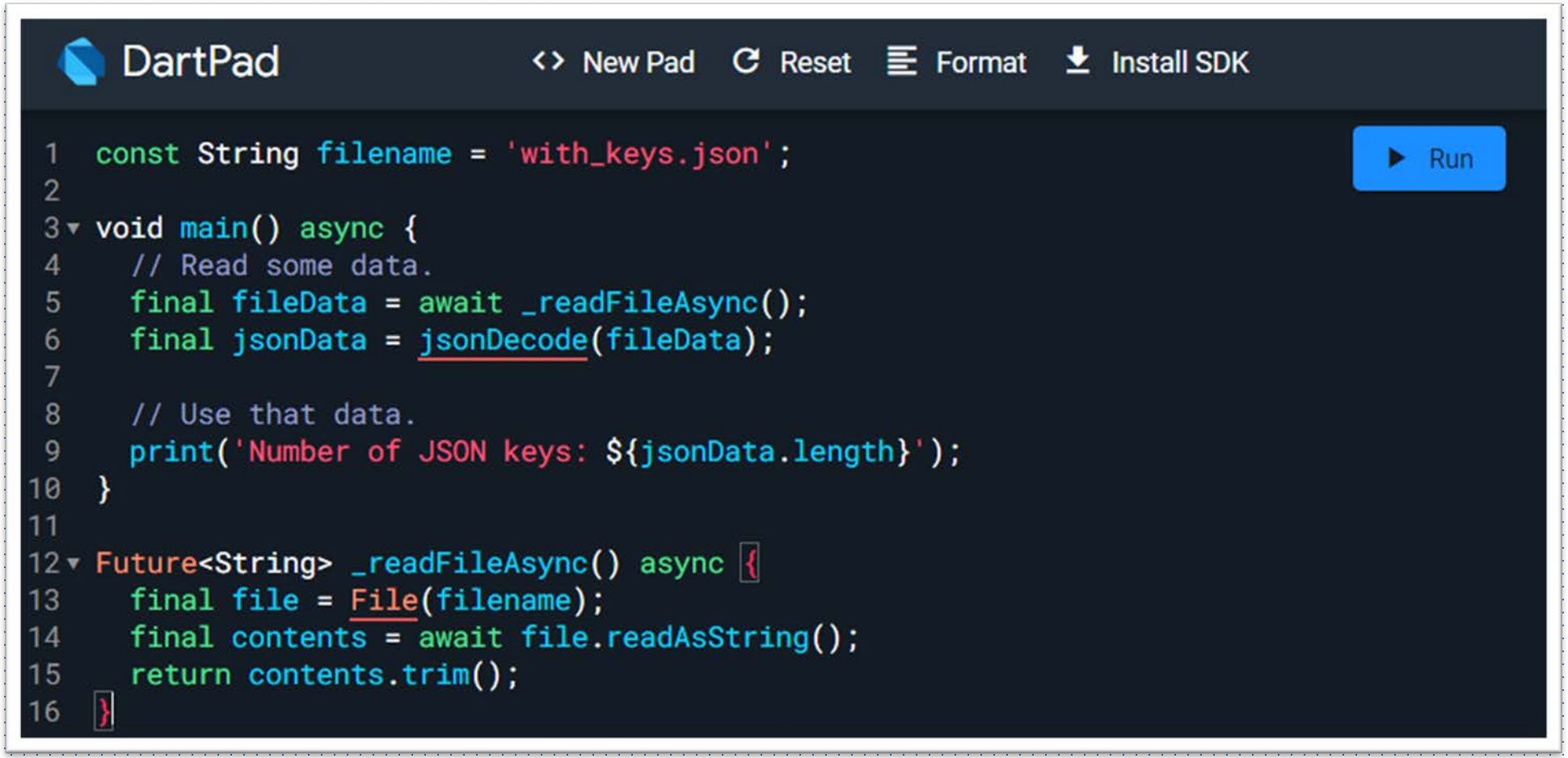
Почему асинхронный код важен? № 2!

- Имеет значение, является ли метод синхронным или асинхронным, поскольку большинству приложений необходимо выполнять несколько действий одновременно.

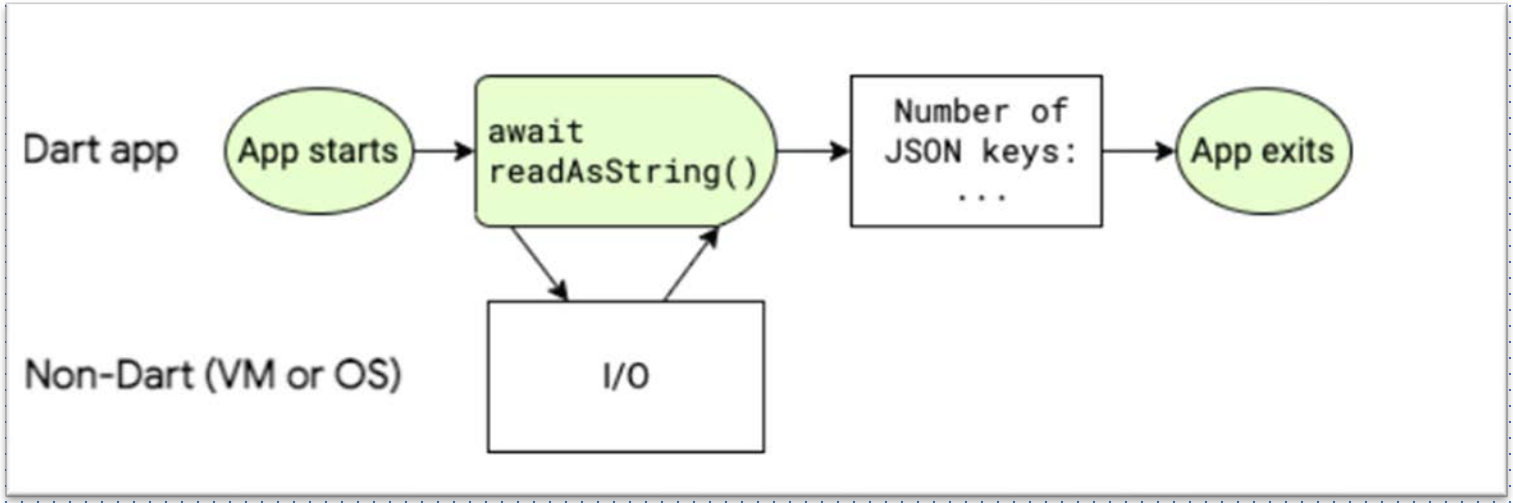
- Асинхронные вычисления часто являются результатом выполнения вычислений вне текущего кода Dart; сюда входят вычисления, которые не завершаются немедленно, и где вы не хотите блокировать свой код Dart в ожидании результата. Например, приложение может запустить HTTP-запрос, но ему необходимо обновить свое отображение или ответить на ввод пользователя до завершения HTTP-запроса. Асинхронный код помогает приложениям оставаться отзывчивыми.
- Эти сценарии включают вызовы операционной системы, такие как неблокирующий ввод-вывод, выполнение HTTP-запроса или взаимодействие с браузером. Другие сценарии включают ожидание вычислений, выполненных в другом изоляте Dart, или, возможно, просто ожидание срабатывания таймера. Все эти процессы либо выполняются в другом потоке, либо обрабатываются операционной системой или средой выполнения Dart, что позволяет коду Dart выполняться одновременно с вычислениями.

Синтаксис асинхронного ожидания

- Ключевые слова async и await предоставляют декларативный способ определения асинхронных функций и использования их результатов.



- Функция main() использует await ключевое слово перед, readFileAsync() чтобы позволить другому коду Dart (например, обработчикам событий) использовать ЦП во время выполнения собственного кода (файловый ввод-вывод). Использование await также приводит к преобразованию Future<String> возвращаемого значения readFileAsync() в файл String. В результате переменная contents имеет неявный тип String.
- Dart приостанавливается во время readAsString() выполнения кода, отличного от Dart, либо на виртуальной машине Dart (VM), либо в операционной системе (ОС). После readAsString() возврата значения выполнение кода Dart возобновляется.

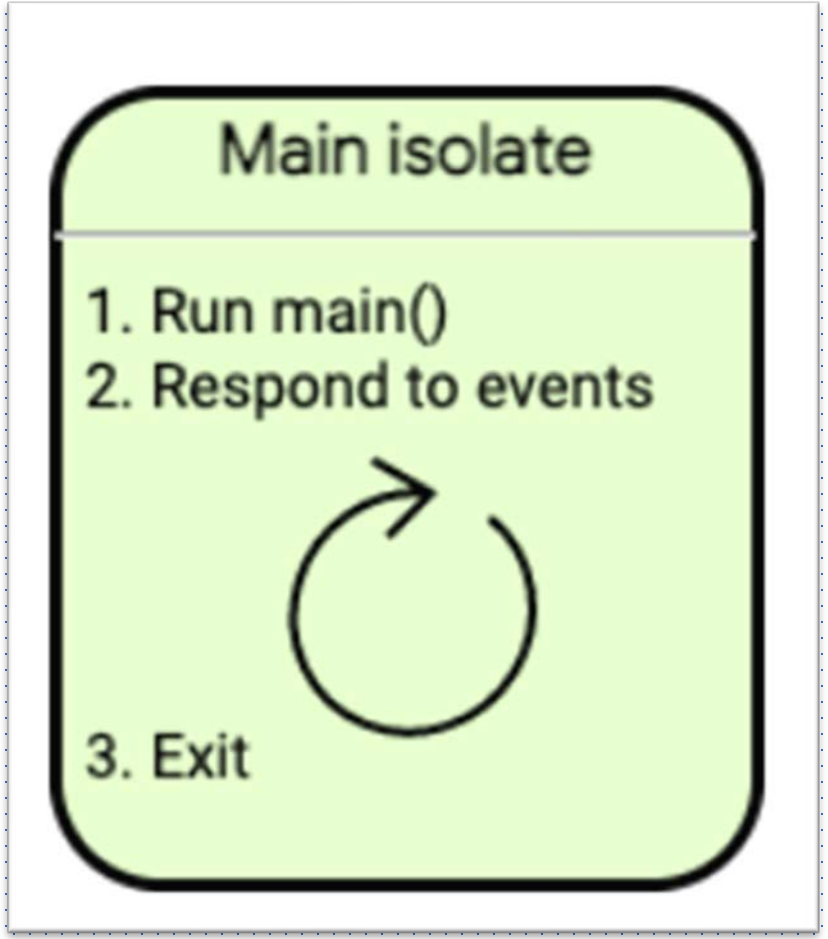


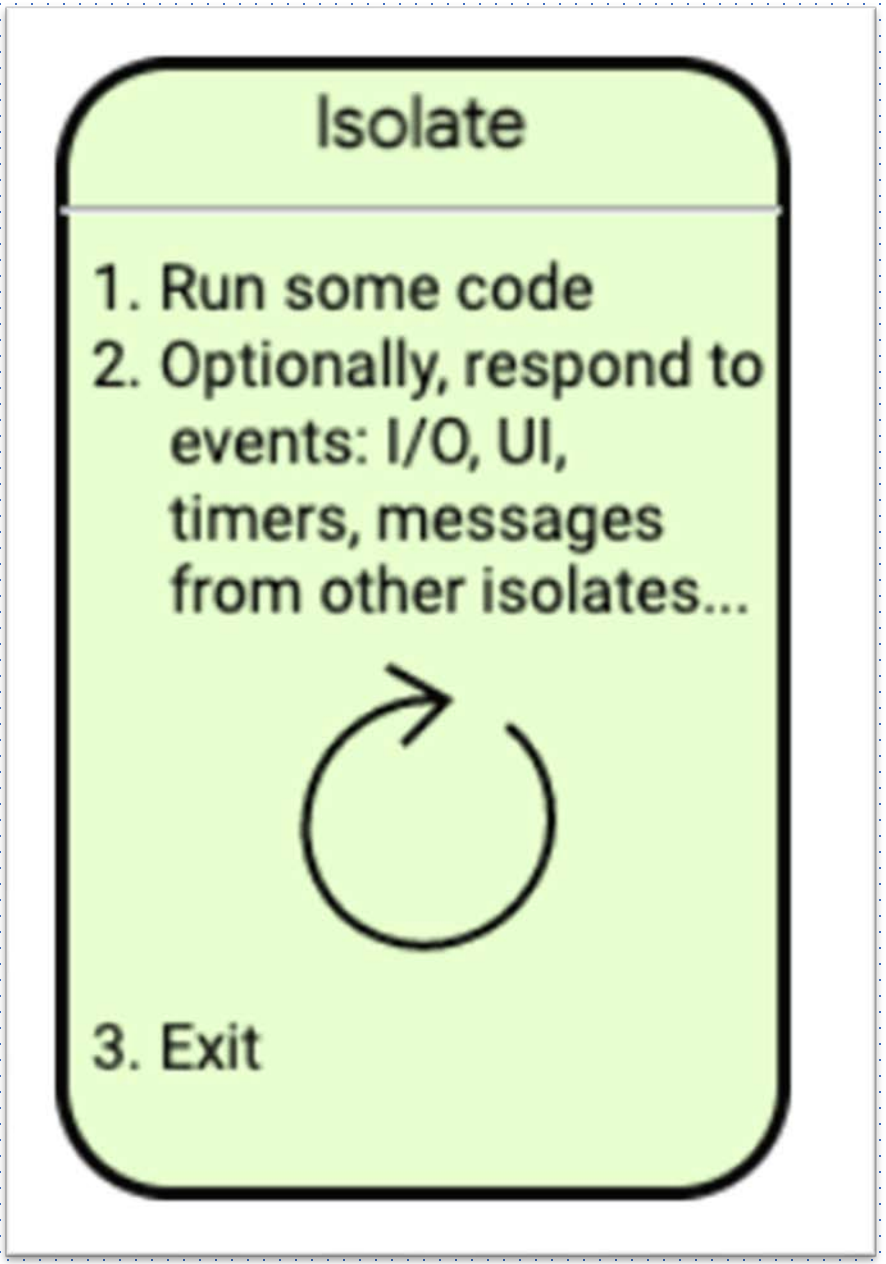
Как работают изоляты?

- Большинство современных устройств имеют многоядерные процессоры. Чтобы воспользоваться преимуществами нескольких ядер, разработчики иногда используют потоки общей памяти, работающие одновременно. Однако параллелизм с общим состоянием подвержен ошибкам и может привести к усложнению кода.
- Вместо потоков весь код Dart выполняется внутри изолятов. Каждый изолят имеет собственную кучу памяти, что гарантирует, что ни одно из состояний изолята не будет доступно из любого другого изолята. Отсутствие общего состояния между изолятами означает , что в Dart не возникнут сложности параллелизма, такие как мьютексы или блокировки , а также гонки данных. Тем не менее, изоляты не предотвращают возникновение гонок в целом.
- Используя изоляты, код Dart может выполнять несколько независимых задач одновременно, используя дополнительные ядра процессора, если они доступны. Изоляты подобны потокам или процессам, но каждый изолят имеет собственную память и единственный поток, выполняющий цикл событий.

Главный изолят

- Часто вообще не нужно думать об изолятах. Программы Dart по умолчанию запускаются в основном изоляте. Это поток, в котором программа начинает запускаться и выполняться, как показано на следующем рисунке ->
- Даже программы с одним изолятом могут выполняться бесперебойно. Прежде чем перейти к следующей строке кода, эти приложения используют async-await для ожидания завершения асинхронных операций. Правильное приложение запускается быстро и как можно скорее попадает в цикл событий. Затем приложение оперативно реагирует на каждое событие в очереди, при необходимости используя асинхронные операции.



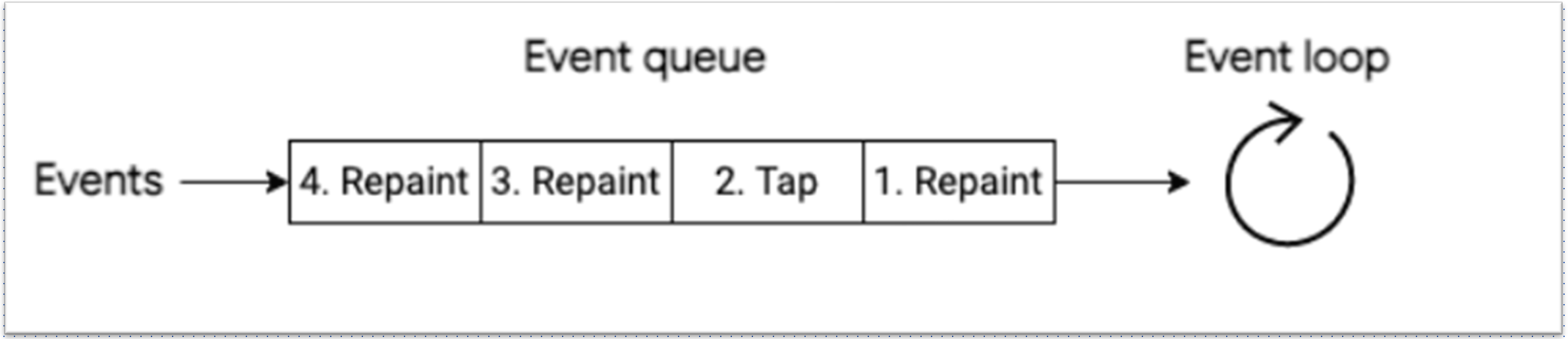


Жизненный цикл изолята

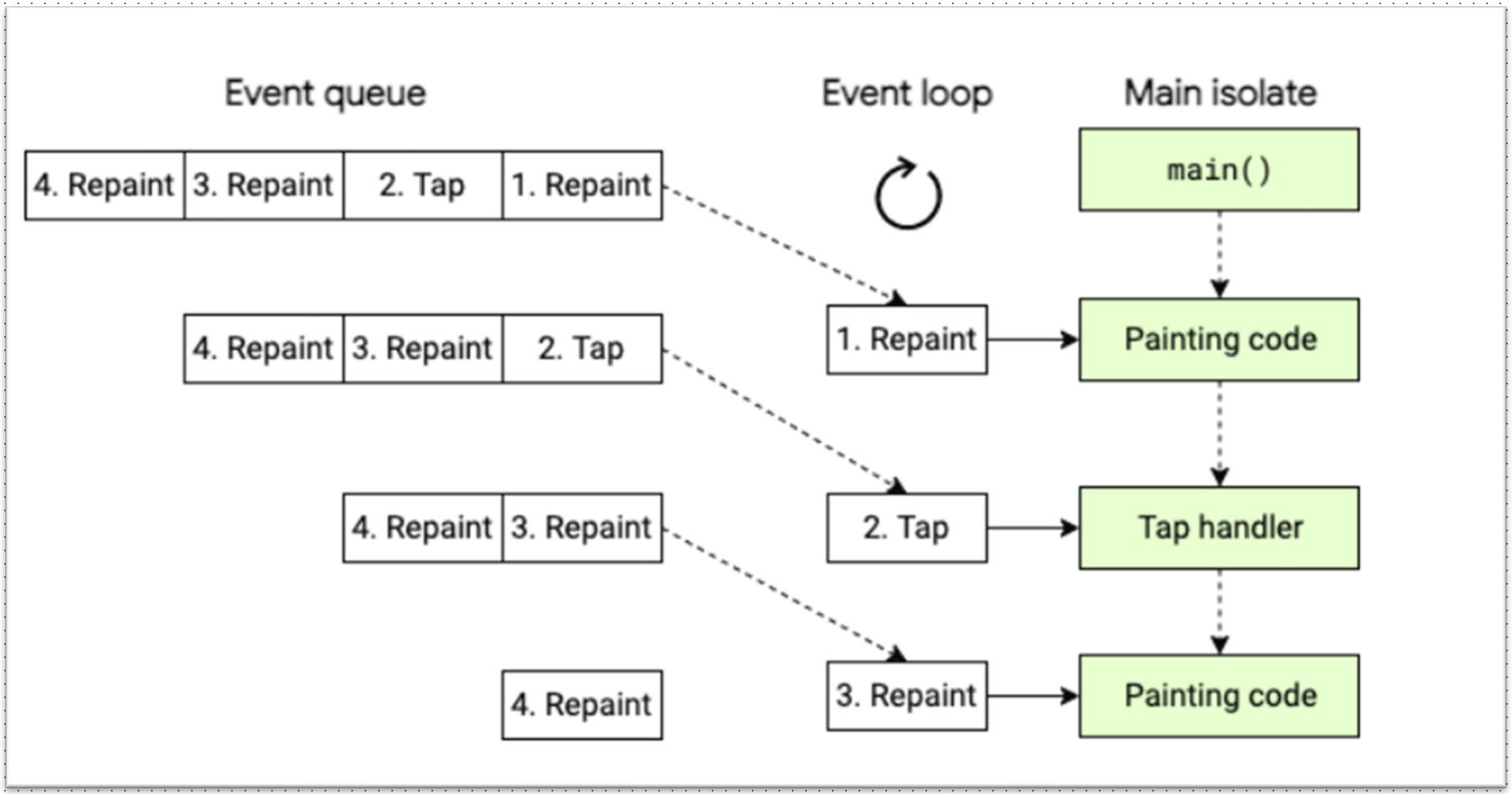
- Как показано на следующем рисунке, каждый изолят начинается с запуска некоторого кода Dart, например функции main(). Этот код Dart может зарегистрировать некоторые прослушиватели событий — например, для реагирования на ввод пользователя или файловый ввод-вывод. Когда исходная функция изолята возвращается, изолят остается, если ему необходимо обрабатывать события. После обработки событий изолят завершает работу.

Обработка событий

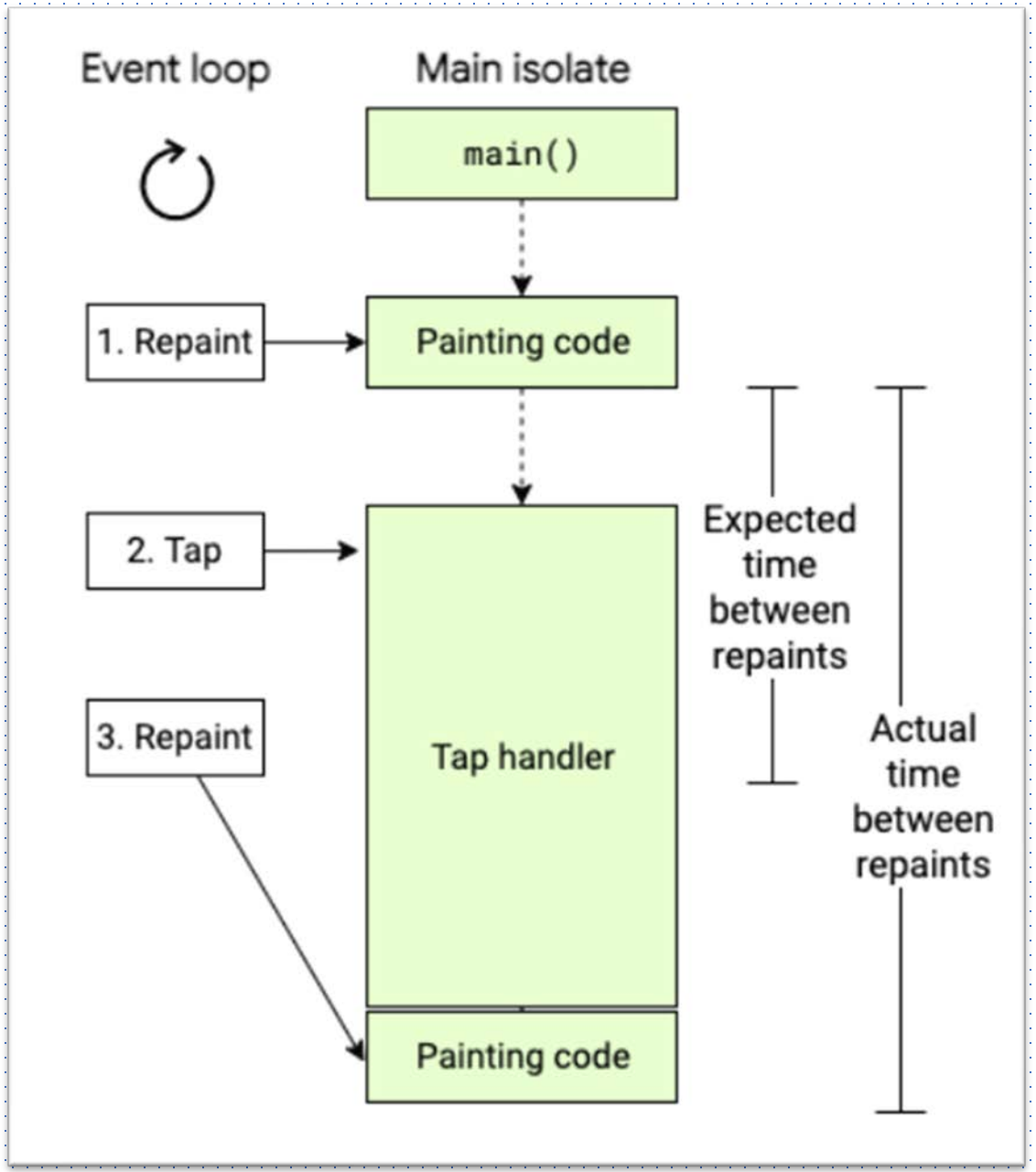
- В клиентском приложении очередь событий основного изолята может содержать запросы на перерисовку и уведомления о касании и других событиях пользовательского интерфейса. Например, на следующем рисунке показано событие перерисовки, за которым следует событие касания, а затем два события перерисовки. Цикл обработки событий берет события из очереди в порядке поступления.



- Обработка событий происходит на основном изоляте после main() выхода. На следующем рисунке после main() выхода основной изолят обрабатывает первое событие перерисовки. После этого основной изолятор обрабатывает событие касания, за которым следует событие перерисовки.

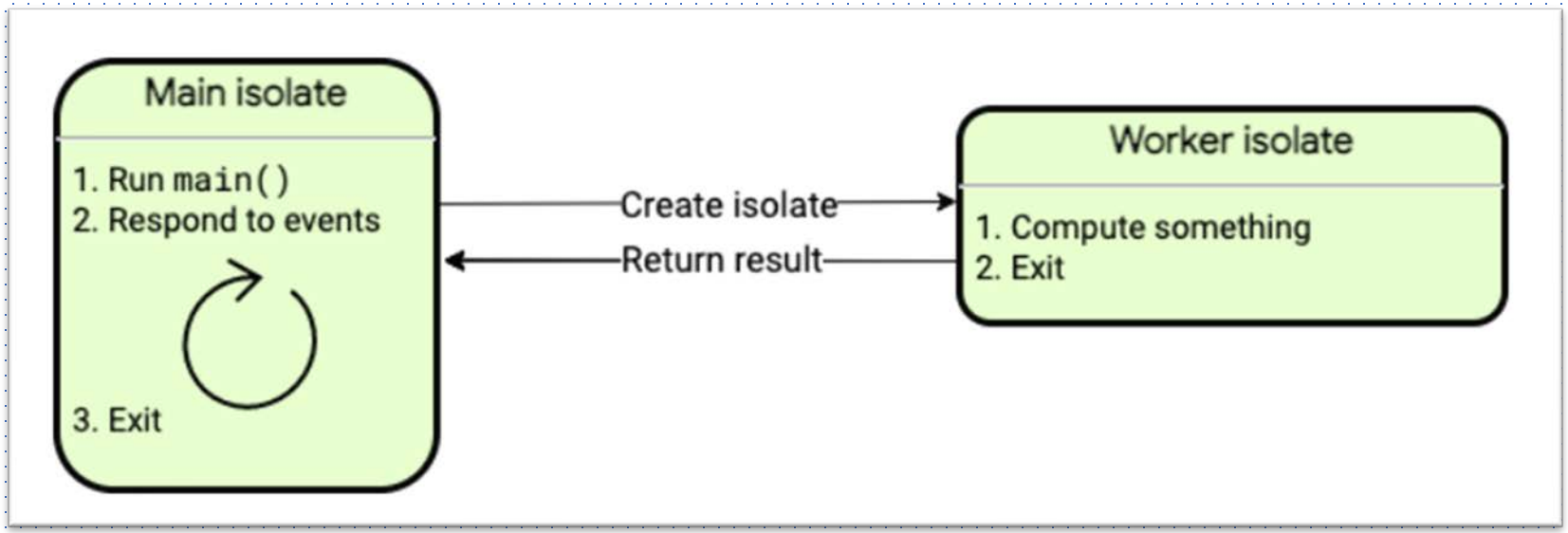


- Если синхронная операция занимает слишком много времени, приложение может перестать отвечать на запросы. На следующем рисунке код обработки касания занимает слишком много времени, поэтому последующие события обрабатываются слишком поздно. Приложение может зависать, а любая анимация, которую оно выполняет, может быть прерывистой.
- В клиентских приложениях результатом слишком длительной синхронной операции часто является неровная (неплавная) анимация пользовательского интерфейса. Хуже того, пользовательский интерфейс может вообще перестать отвечать на запросы.



Фоновые работники

- Если пользовательский интерфейс вашего приложения перестает отвечать на запросы из-за трудоемких вычислений (например, анализа большого файла JSON), есть возможность выгрузки этих вычислений в изолированный рабочий процесс, часто называемый фоновым рабочим процессом. Типичный случай, показанный на следующем рисунке, — это создание простого изолированного рабочего процесса, который выполняет вычисления и затем завершает работу. Изолят работника возвращает результат в сообщении при выходе работника.



- Каждое сообщение изоляции может доставить один объект, который включает в себя все, что транзитивно достижимо из этого объекта. Не все типы объектов доступны для отправки, и отправка завершается неудачно, если какой-либо транзитивно достижимый объект не может быть отправлен. Например, можно отправить объект типа List<Object> только в том случае, если ни один из объектов в списке не является недоступным для отправки. Если одним из объектов является, Socket, то отправка не удалась, поскольку сокеты не могут быть отправлены.
- Изолированный работник может выполнять ввод-вывод (например, чтение и запись файлов), устанавливать таймеры и многое другое. Он имеет собственную память и не имеет общего состояния с основным изолятом. Рабочий изолят может блокироваться, не затрагивая другие изоляты.

Параллелизм в сети

- Все приложения Dart могут использовать async-await, Future и Stream для неблокирующих чередующихся вычислений. Однако веб-платформа Dart не поддерживает изоляты. Веб-приложения Dart могут использовать веб-воркеров для запуска сценариев в фоновых потоках, аналогично изолятам. Однако функциональность и возможности веб-воркеров несколько отличаются от изолированных.
- Например, когда веб-воркеров отправляют данные между потоками, они копируют данные туда и обратно. Однако копирование данных может быть очень медленным, особенно для больших сообщений. Изолированные устройства делают то же самое, но также предоставляют API, которые могут более эффективно передавать вместо этого память, в которой хранится сообщение.
- Создание веб-воркеров и изолятов также различается. Можно создавать в веб-воркеров, только объявив отдельную точку входа в программу и скомпилировав ее отдельно. Запуск веб-воркера аналогичен Isolate.spawnUri запуску изолята.