Event-loop

Event-loop **обеспечивает эффективное управление асинхронными операциями без накладных расходов на многопоточность**. Особенно полезен для создания клиент-серверных приложений с высокими требованиями к производительности.

Насколько я поняла, изначально Event-loop использовался в однопоточных языках (таких как JavaScript), поэтому реализация и исследование данной штуки для Java на старте вызвали у меня некоторые затруднения. Поэтому сначала расскажу об Event-loop в общих чертах и потом уже перейду к нюансам использования его в Java.

В однопоточных языках Event-Loop позволяет выполнять параллельно множество задач. Это нужно для того, чтобы избежать случаев когда какая-то одна задача достаточно ресурсоемкая и в момент пока она исполняется пользователь сидит и ждёт с зависшим интерфейсом, например. Поэтому для выполнения асинхронного кода используют данную концепцию.

Event-loop регулирует последовательность исполнения контекстов — стек. Он формируется, когда сработало событие или была вызвана функция. Реакция на событие помещается в очередь исполнения, в event-loop, который последовательно, с каждым циклом выполняет попадающий в него код. При этом привязанная к событию функция вызывается следующей после текущего контекста исполнения. Постоянно работают связанные между собой синхронная и асинхронная очереди выполнения.

Синхронная — стек — формирует очередь и пробрасывает в асинхронную — event-loop — вызовы функций, которые будут выполнены после текущего запланированного исполняемого контекста.

Другими словами, event-loop выполняет одну простую задачу — осуществляет контроль стека вызовов и очереди обратных вызовов. Если стек вызовов пуст, цикл событий возьмет первое событие из очереди и отправит его в стек вызовов, который его запустит. При вызове нового метода вверху стека выделяется отдельный блок памяти. Стек вызовов отвечает за отслеживание всех операций в очереди, которые должны быть выполнены. При завершении очереди она извлекается из стека.
Чтобы данные находились в консистентном состоянии, каждая функция должна быть выполнена до конца. Единственный поток представлен в виде очереди контекстов исполнения, в которой и происходит «вклинивание» функций, прошедших через цикл событий.

Теперь перейдем непосредственно к нюансам в Java.

Event-loop представляет собой бесконечный цикл, где каждый цикл выполняет все задания, которые предоставляются селектором или хранятся в специальных очередях. Каждая из этих задач должна быть небольшой, и ее выполнение называется tick.

Eventloop использует NIO в Java для обеспечения асинхронных вычислений и операций ввода-вывода (TCP, UDP). Также можно запускать несколько потоков event-loop на доступных ядрах. Минимальная нагрузка на Garbage Collector обеспечивается за счет того, что массивы и байтовые буферы используются повторно. Event-loop

может планировать или откладывать определенные задачи для последующего выполнения или фонового выполнения. Event-loop также является однопоточным, поэтому у него нет накладных расходов на параллелизм.

Единственной блокирующей операцией бесконечного цикла Event-loop является Selector.select(). Эта операция выбирает набор ключей , соответствующие каналы которых готовы к операциям ввода/вывода. Event-loop асинхронно обрабатывает выбранные ключи и выполняет поставленные в очередь runnables в одном потоке.

Event-loop может работать с различными типами задач, которые хранятся в отдельных очередях: местные задачи (добавленные из текущего потока), одновременные задачи (добавленные из других потоков), запланированные и фоновые задачи.

Event-loop будет остановлен, если его очереди с нефоновыми задачами пусты, селектор не имеет выбранных ключей и количество одновременных операций в других потоках равно 0. Чтобы предотвратить закрытие Event-loop, нужно установить флаг keepAlive . Когда он установлен в значение true, Event-loop будет продолжать работать даже без заданий.

Подводя итог, считаю нужным ещё раз упомянуть, что основное правило Event-Loop: функции должны быть атомарными — выполнять ровно одну задачу и оставаться неделимыми. Чем меньше и короче функция, тем быстрее она выполняется, гарантируя чистоту и доступность очереди контекстов для пользовательских событий!

** Использование Event-Loop в GUI.
Когда событие создается пользователем, оно сначала помещается в очередь событий , которая представляет собой структуру данных на основе очереди, используемую для отслеживания событий, созданных пользователем. События помещаются туда потоком в программе, обычно частью графического интерфейса, который подключен и «прослушивает» события. Мы используем очередь для отслеживания событий на тот случай, если пользователь генерирует события быстрее, чем наша программа может их обработать.

Затем в потоке графического интерфейса нашей программы есть цикл, который постоянно проверяет, содержит ли очередь событий какие-либо элементы. Это код цикла обработки событий . Если содержит, то он возьмет первый элемент из очереди и проверит его. Если это событие связано с уже известным обработчиком событий где-то в нашем коде, тогда цикл событий вызовет обработчик событий (функция «обратного вызова»).

Как только обработчик событий вернется, цикл обработки событий возьмет следующее событие из очереди и обработает его, и это будет продолжаться до тех пор, пока в очереди не останется событий.

В некоторых графических интерфейсах цикл событий также отвечает за обновление графического интерфейса на самом экране. Таким образом, пока выполняются обработчики событий, сам экран графического интерфейса не может быть обновлен, и приложение будет «зависать».

Поэтому очень важно , чтобы обработчики событий были очень короткими и выполнялись быстро, иначе пользователь может заметить, что наше приложение не

отвечает. Если событие требует большого объема вычислений, мы можем захотеть создать отдельный поток для обработки этой операции. К счастью, большинству простых программ с графическим интерфейсом этого не требуется, но об этом всегда следует помнить, если вдруг приложение начнет работать медленно.

*** Пример использования данной концепции

В одной из первых версий этой реализации я использовала Event-Loop как часть, отвечающую за инертность робота.

```
switch (event.getType()){
  case STAY -> eventloop.post(() ->
System.out.println("Стёпа релаксирует"));
  case STOP -> eventloop.submit(stepa::stop);
  case BUMP -> eventloop.delay(100L,
EventloopExample::bump);
  case RIDE -> eventloop.delay(50L,
EventloopExample::ride);
  default -> event.setType(MyEvent.Type.STAY);
```

Лоскутова Ирина, R3142, фак. СУиР