Асинхронность

#Асинхронность #EventLoop #Callbacks #AsyncMonad #Промисы #await

Синхронные операции — операции, при которых мы получаем результат в результате блокирования потока выполнения. Для простых вычислительных операций (сложение/умножение чисел) — это единственный вариант их совершения, для операций ввода/вывода — один из, при этом мы говорим, к примеру, *«попытайся прочитать из файла что-нибудь за 100мс»*, и если для чтения ничего нет — поток выполнения будет заблокирован на эти 100мс.

В некоторых случаях это допустимо (например, если мы делаем простое консольное приложение, либо какую-либо утилиту, цель которой — отработать и всё), но в некоторых — нет. К примеру, если мы так застрянем в потоке, в котором обрабатывается UI — наше приложение зависнет. За примерами далеко ходить не нужно — если javascript на сайте сделает `while(true);`, то перестанут вызываться какие-либо другие обработчики событий страницы и её придётся закрыть. Те же дела, если начать что-нибудь вычислять под Android'ом в обработчиках UI-событий (код которых вызывается в UI-потоке), это приведёт к появлению окна «приложение не отвечает, закрыть?» (подобные окна вызываются по watchdog-таймеру, который сбрасывается, когда выполнение возвращается обратно к системе UI).

Асинхронные операции — операции, при которых мы **просим совершить** некоторую операцию и можем каким-либо образом отслеживать процесс/результат её выполнения. Когда она будет выполнена — неизвестно, но мы можем продолжить заниматься другими делами.

Event loop

Event loop — это бесконечный цикл, который берёт **события** из очереди и как-то их обрабатывает. А в некоторых промежутках — смотрит, не произошло ли каких-нибудь **IO-событий**, либо не просрочились ли какие-либо **таймеры** — тогда добавляет в очередь событие об этом, чтобы потом обработать.

Вернёмся к примеру с браузером. Вся страница работает в одном **event loop'e**, загруженный страницей **javascript** добавляется в **очередь**, чтобы выполниться. Если на странице происходят какие-либо **UI-события** (клик по кнопке, перемещение мыши, прочее) — код их обработчиков добавляется в **очередь**. Обработчики выполняются последовательно, нет никакой параллельности, пока работает какой-либо код — все

остальные ждут. Если какой-нибудь код вызовет какую-нибудь специальную функцию, вроде `setTimeout(function() { alert(42) }, 5000)` — то это создаст где-то вне цикла таймер, по истечению которого в **очередь** будет добавлен код функции с `alert(42)`.

Фишка: если кто-то в очереди перед выполнением обработчика будет что-то долго вычислять, то обработчик таймера, очевидно, выполнится позже, чем через пять секунд.

Вторая фишка: даже если мы попросим, например, 1 миллисекунду ожидания, может пройти куда больше, т.к. реализация **event loop'a** может посмотреть: «ага, очередь пуста, ближайший таймер через 1мс, будем ждать IO-событий 1мс», а когда мы вызовем select, реализация операционной системы может посмотреть: «ага, событий вроде нет, на твоё время мне всё равно, я делаю context switch, пока есть возможность», а там все остальные потоки заиспользовали всё доступное им время и мы пролетели.

select

Асинхронные IO-события на низком уровне реализованы при помощи вариаций select'a. У нас есть некие файловые дескрипторы (которые могут быть либо файлами, либо сетевыми сокетами, либо чем-то ещё (по сути, в Linux что угодно может являться файлом (или наоборот, файл может являться чем угодно))).

И мы можем вызвать некоторую синхронную функцию, передав ей множество дескрипторов, от которых мы ожидаем ввода, либо же хотим что-то записать, которая заблокирует поток до тех пор пока:

1. Один или несколько переданных нами дескрипторов не станут готовы к совершению желаемой нами операции.
2. Не истекло время ожидания (если оно было задано).
   В результате выполнения этой процедуры мы получим множества готовых к чтению/записи файлов.

Callbacks

Самый простой способ получить результаты выполнения асинхронной операции — при её создании передать ссылки на функции, которые будут вызваны при каком-либо прогрессе выполнения/готовности результата.

Это довольно низкоуровневый подход, и часто неумение банально писать функции «в столбик» вместе со злоупотреблением анонимных функций приводит к «callback hell»

(ситуация, когда мы имеем четыре-десять уровней вложенности функций, чтобы обработать последовательные операции):

```
// Вкладываем
function someAsync(a, callback) {
  anotherAsync(a, function(b) {
    asyncAgain(b, function(c) {
      andAgain(b, c, function(d) {
        lastAsync(d, callback);
      });
    });
  });
}

// Линейно
function someAsync2(a, callback) {
  var b;

  anotherAsync(a, handleAnother);

  function handleAnother(_b) {
    b = _b;
    asyncAgain(b, handleAgain);
  }

  function handleAgain(c) {
    andAgain(b, c, handleAnd);
  }

  function handleAnd(d) {
    lastAsync(d, callback);
  }
}
```

Async Monad

Мы, программисты, любим абстрагировать и обобщать для сокрытия разных сложностей/рутины. Поэтому существует, в том числе, абстракция над асинхронными вычислениями.

Что такое **«вычисление»**? Это процесс **преобразования A в B**. Будем записывать синхронные вычисления как **A → B**.

Что такое **«асинхронное значение»**? Это обещание предоставить нам в будущем некоторое значение **T** (которое может быть успешным результатом, либо ошибкой). Будем обозначать это как **Async[T]**.

Тогда **«асинхронная операция»** будет выглядеть как `A → Async[T]`, где **A** — какие-то аргументы, необходимые для старта операции (например, это может быть URL, к которому мы хотим совершить GET-запрос).

Как работать с **Async[T]**? Пусть у него будет метод **run**, который примет **коллбэк**, который будет вызван тогда, когда данные станут доступны: `Async[T].run : (T → ()) → ()` (принимает функцию, принимающую **T**, ничего не возвращает).

Хорошо, а теперь добавим самое главное — возможность **продолжить** асинхронную операцию. Если у нас есть **Async[A]**, то, очевидно, когда **A** станет доступно, мы можем создать **Async[B]** и ждать уже его результата. Функция для такого продолжения будет выглядеть так:
`Async[A].then : (A → Async[B]) → Async[B]`

Т.е. если мы можем создать **Async[B]** из некого **A**, а так же имеем **Async[A]**, который когда-нибудь предоставит нам **A**, нет никаких проблем предоставить **Async[B]** сразу, ибо **B** мы сможем всё-таки получить через какое-то время и в итоге всё сойдётся.

Реализация этого добра
И тогда тот наш синтетический пример выше становится:

```
function someAsync(a) {
  return anotherAsync(a).then(function(b) {
    return asyncAgain(b).then(function(c) {
      return andAgain(b, c);
    }).then(function(d) {
      return lastAsync(d);
    });
  });
}
```

Но дальше интереснее. Явно разграничим тип асинхронного значения на ошибку/результат. Теперь у нас всегда **Async[E + R]** (плюс это тип-сумма, одно из двух). И тогда мы можем, к примеру, ввести метод `Async[E + R].success : (R → Async[E + N]) → Async[E + N]`. Обратите внимание, что **E** осталось нетронутым.

Мы можем реализовать этот метод только так, чтобы он выполнял переданную ему функцию только в случае получения успешного результата (т.е. получению **R**, а не **E**) и запускал следующую асинхронную операцию, иначе — результат асинхронной операции продолжает оставаться «ошибочным».

```
this.success = function(f) {
  return new Async(function(callback) {
    runParent(function(x) {
      if (x.isError()) callback(x);
      else f(x).run(callback);
    });
  });
};
```

Теперь если мы будем chain'ить асинхронные операции при помощи метода **success**, мы будем обрабатывать только успешную ветку развития событий, а любая ошибка проскочит все последующие обработчики и попадёт сразу в коллбэк, переданный в **run**.

Мы только что абстрагировали поток выполнения и ввели в нашу абстракцию **исключения**. Если поиграться ещё немного, можно будет придумать метод **failure**, который может преобразовать ошибку в другую ошибку, либо же вернуть **успешный результат**.

Промисы (promises, обещания)

Есть стандарт, описывающий интерфейс Thenable. Он работает практически идентично тому, что было описано выше, но в **Promises/A+** нет понятия **старта** асинхронной операции. Если мы имеем на руках **Thenable**, то уже где-то что-то выполняется и всё что мы можем — подписаться на результат выполнения. И там один метод **then**, принимающий две опциональных функции для обработки успешной/провальной ветки, а не разные методы.

Здесь уж на вкус и цвет, у обоих подходов есть и плюсы и минусы.

async/await — промисы + корутины

Чтобы использовать промисы, нам нужно использовать лямбда-функции в невероятных количествах. Что может быть довольно визуально шумно и неудобно. Нет ли возможности сделать это как-то лучше?

Есть.

У нас есть корутины, у которых может быть множество точек входа. И это то, что нам нужно. Пусть у нас будет корутина, которая выдаёт наружу **Async[E + R]**, а внутрь неё подаётся получившееся **R**, либо возбуждается исключение **E**. И тогда начинается дзен:

```
function someAsync*(a) {
  var b = yield anotherAsync(a),
      c = yield asyncAgain(b),
      d = yield andAgain(b, c);

  yield lastAsync(d);
}
```

Затем нам нужен «executor» такого добра, который будет принимать эту корутину, доставать из неё выходы, если они являются **Async'ами** — выполнять их, если другими корутинами — рекурсивно execute'ить их, считая результатом последний **yield**.

А **async/await** — это когда мы **yield** переименовываем в **await**, а перед декларацией функции пишем **async**. Ну и иногда (в случае Python, например), можно увидеть **асинхронные генераторы**, в которых в наличии одновременно и **yield**, и **await**. Тогда они ведут себя как те же корутины, но операции с ней становятся асинхронными, ибо между возвратом/принятием она ждёт результатов своих внутренних асинхронных операций.