Кооперативность

#Акторы #Корутины

В отличии от вытесняющей многозадачности, которая прерывает выполнение вашего кода в любое время, в любом месте, котором пожелает, кооперативная является «ручным вариантом», когда ваш код знает о том, что выполняется не один, есть другие ожидающие процессы, и он сам решает, когда ему передать управление другим.

При кооперативной многозадачности важно не совершать длительных операций, а если и совершать — то периодически передавать управление.

Идеальным вариантом будет, если ваша «кооперативная часть» не будет работать с блокирующим I/O и мощными вычислениями, а будет использовать неблокирующее асинхронное API, а эти времязатратные вещи будут вынесены «вовне», где будут выполняться параллельно «псевдопараллельности».

Корутины

Я говорил, что операционная система schedule'ит потоки, выполняя их код определёнными порциями времени. Но давайте подумаем, как это в принципе возможно реализовать. Варианта получается два:

1. Процессор поддерживает возможность оборвать выполнение инструкций через какое-то время и выполнить какой-то другой заранее заданный код (прерывание по таймеру, либо, если возможно, по количеству выполненных инструкций).
2. Мы городим компилятор машинного кода в машинный код, который будет сам считать количество выполненных инструкций каким-либо образом и прервёт выполнение, когда счётчик достигнет какого-нибудь предела.
   Второй вариант к оверхеду на переключение контекста (сохранить значение всех регистров куда-нибудь) добавляет оверхед на эту модификацию кода (хотя её можно сделать и AOT), плюс на подсчёт инструкций в процессе их выполнения (всё станет медленнее не более чем в два раза, а в большинстве случаев — куда меньше).

И вот когда мы по каким-то причинам не хотим (или не можем) использовать прерывания процессора по таймеру, а второй вариант это вообще корыто какое-то — в дело вступает кооперативная многозадачность. Мы можем писать функции в таком стиле, что сами

говорим, когда можно прервать её выполнение и повыполнять какие-нибудь другие задачи. Как-то так:

```
void some_name() {
        doSomeWork();
        yield();
        while (true) {
                doAnotherWork();
                yield();
        }
        doLastWork();
}
```

Где при каждом вызове `yield()` система сохранит весь контекст функции (значения переменных, место, где был вызван `yield()`) и продолжит выполнять другую функцию такого же типа, восстановив её контекст и возобновив исполнение с того места, где она прошлый раз закончила.

У такого подхода есть и плюсы и минусы. Из плюсов:

- Если у нас только один физический поток (или если наша группа задач выполняется только в одном) — то на некоторую часть общей памяти не потребуются блокировок, т.к. мы сами решаем, когда будут выполняться другие задачи, и можем выполнять действия без опасений, что кто-то другой увидит или вмешается в них на полпути, а там, где блокировки будут нужны — они реализуются просто boolean'ом.
  Минусы:
- Кванты времени будут сильно неравномерными (что не так важно, главное, чтобы они были *достаточно малы*, чтобы не были заметны задержки).
- Какая-нибудь функция может всё-таки создать ощутимую задержку, реализовавшись некорректно. И, что гораздо хуже — если она вовсе не вернёт управление.
  По быстродействию сложно говорить. С одной стороны, оно может быть быстрее, если будет сменять контексты не так часто, как планировщик, может быть медленнее, если будет переключать контексты слишком часто, а с другой стороны — слишком большие задержки между возвращениями управления другим задачам могут повлиять на UI либо I/O, что станет заметно и тогда пользователь вряд ли скажет, что оно стало работать быстрее.

Но вернёмся к нашим корутинам. Корутины (coroutines, сопрограммы) имеют не одну точку входа и одну выхода (как обычные функции — подпрограммы), а одну стартовую, опционально одну финальную и произвольное количество пар выход-вход.

Для начала рассмотрим случай с бесконечным количеством выходов (генератор бесконечного списка):

```
function* serial() {
        let i = 0;
        while (true) {
                yield i++;
        }
}
```

Это Javascript, при вызове функции **serial** вернётся объект, у которого есть метод `next()`, который при последовательных вызовах будет возвращать нам объекты вида `{value: Any, done: Boolean}`, где **done** будет **false** пока выполнение генератора не уткнётся в конец блока функции, а в **value** — значения, которые мы посылаем yield'ом.

… но кроме возвращения значения **yield** может так же и принять новые данные внутрь. Например, сделаем какой-нибудь такой сумматор:

```
function* sum() {
        let total = 0;
        while (true) {
                let n = yield total;
                total += n;
        }
}
let s = sum();
s.next();  // 0
s.next(3); // 3
s.next(5); // 8
s.next(7); // 15
s.next(0); // 15
```

Первый вызов `next()` получает значение, которое передал первый **yield**, а затем мы можем передать в `next()` значение, которое хотим чтобы **yield** вернул.
Думаю, вы поняли, как это работает. Но если пока не понимаете, как это можно использовать — подождите следующей статьи, где я расскажу о **промисах** и **async/await'e**

Акторы

Модель акторов — мощная и довольно простая модель параллельных вычислений, позволяющая добиться одновременно и эффективности и удобства небольшой ценой (о ней далее). Есть лишь две сущности: **актор** (у которого есть *адрес* и *состояние*) и **сообщения** (произвольные данные). При получении сообщения актор может:

- Действовать в зависимости от своего *состояния*
- Создать новых акторов, он будет знать их *адреса*, может задать их первоначальное *состояние*
- Отправить сообщения по известным *адресам* (в сообщениях можно отправлять *адреса*, включая свой)
- Изменить своё *состояние*
  Что хорошо в акторах? Если правильно разделять ресурсы по акторам, то можно полностью избавиться от каких-либо блокировок (хотя, если подумать, **блокировки** становятся **ожиданиями результата**, но во время этого ожидания вы вынуждены обрабатывать другие сообщения, а не просто ждать).

Кроме того, ваш код с большой вероятностью станет организован куда лучше, логически разделён, вам придётся хорошо прорабатывать API акторов. И актор куда проще переиспользуется, чем просто класс, т.к. единственный способ взаимодействовать с ним — это отправлять ему сообщения и принимать сообщения от него на переданных ему адресах, у него нет жёстких зависимостей и неявных связей, а любой его «внешний вызов» легко перехватывается и кастомизируется.

Цена этого — очередь сообщений и оверхед на работу с ней. Каждый актор будет иметь очередь поступающих ему сообщений, в которой будут накапливаться приходящие сообщения. Если он не будет успевать их обрабатывать — она будет расти. В нагруженных системах вам придётся как-то решать эту проблему, придумывая способы для параллельной обработки, чтобы у вас были группы акторов, которые делают какую-то одну задачу. Но в этом случае очереди дают вам и плюс, т.к. становится очень легко мониторить места, где у вас не хватает производительности. Вместо одной метрики *«я ждал результата 50мс»* у вас для каждого компонента системы появляется метрика *«может обрабатывать N запросов в минуту»*.

Акторы могут быть реализованы множеством разных способов: можно создавать для каждого свой поток (но тогда не сможем создать действительно **много** экземпляров), а можно создать пару потоков, которые будут работать действительно параллельно и крутить внутри них обработчики сообщений — от этого ничего не изменится (если только какие-нибудь из них не делают очень долгих операций, что будет блокировать выполнение остальных), а акторов можно будет создать куда больше. Если сообщения

сериализуемы, то нет проблем распределить акторы по разным машинам, что неплохо повышает возможности к масштабированию.