

4. Асинхронные коммуникации и персистентная рекурсия

В синхронном коде выполнение функций происходит в том месте, где они были вызваны, и в тот момент, когда происходит вызов. В асинхронном коде всё по-другому. Вызов функции не означает, что она отработает прямо здесь и сейчас. Более того, мы не знаем, когда она отработает.

Синхронный подход в случае большого числа медленных операций ввода-вывода (обмен со внешней средой - с дисками, с сетью интернет, с печатным устройством) очень неэффективно использует ресурсы системы. Асинхронный же код продолжает свою работу во время любых медленных операций, в том числе при обращении к микросервисам, скорость ответа от которых может, как и любые другие операции ввода-вывода, быть очень медленной.

Другими словами, код никогда не блокируется на операциях ввода-вывода (IO), но может узнать об их завершении. Правильно написанные асинхронные программы (в тех ситуациях, где это нужно) значительно эффективнее синхронных. Иногда это настолько критично, что синхронная версия просто не справляется с задачей.

*Упорядоченность асинхронных операций обеспечивается в первую очередь через функцию обратного вызова - **Callback**, обычно это последний аргумент в вызове асинхронной функции. Эта функция вызывается после завершения асинхронной операции. Исполняющая подсистема языка программирования никогда не ждет завершения асинхронной функции, после ее запуска она переходит к обработке следующей функции.*

В примерах ниже специально выбран самый ранний синтаксис, чтобы подчеркнуть значение функции обратного вызова. Это гораздо более значимый феномен, чем понятие классической рекурсии, которая оставляет нас в парадигме последовательного программирования архитектуры фон Неймана

Важнейшее преимущество асинхронной неблокирующей логики в сочетании с техникой корутин (coroutine), состоит в том, что в классических нагруженных системах (при использовании распараллеливания с использованием процессов ОС), список доступных процессов обычно быстро исчерпывается, если запущенные в процессах обмена с внешними сервисами синхронные и/или медленные. Заметим

же, что переключения между процессами ОС само по себе довольно ресурсоемкое.

Резюмирую - без использование техники асинхронного программирования построить современную высоконагруженную микросервисную систему (единорога, о котором все мечтают ж-)) невозможно или очень сложно.

Пример классическое рекурсии в архитектуре фон Неймана:

```
let cnt = 0;

function sendMsg1() {

  console.log('Msg1');

  cnt++;

  if (cnt < 10) sendMsg1();

}

sendMsg1();
```

Пример персистентной рекурсии в асинхронной архитектуре:

```
function startMsg(callback) {

  console.log('Start');

  callback();

}

const stopMsg = function() {

  return function() {

    console.log('Stop');

  }

}
```

```
const sendMsg1 = function(callback) {  
  
  return function() {  
  
    console.log('msg1');  
  
    callback();  
  
  }  
  
}  
  
const sendMsg2 = function(callback) {  
  
  return function() {  
  
    console.log('msg2');  
  
    callback();  
  
  }  
  
}  
  
startMsg(sendMsg1(  
  sendMsg1(  
    sendMsg1(  
      sendMsg1(stopMsg()))));  
startMsg(sendMsg1(  
  sendMsg2(  
    sendMsg1(  
      sendMsg2(stopMsg()))));
```

Несложно заметить разницу, если классическая рекурсия разворачивается на этапе выполнения, то перманентная рекурсия в асинхронной логике декларативно формируется на этапе описания цепочек функций. Любая цепочка функций может

быть декларирована и в нужный момент активирована.

При этом не составит никакого труда описать любое дерево из функций (которые, суть будущие действия) или целую сеть с обратными связями. Единственное отличие от живых сетей - в живых сетях эти “функции” и сети создает сама Жизнь, а в нашем примере это пока делается руками программиста.

Совершенно очевидно что такое архитектурное решение гораздо ближе, к коммуникативным сетям жизни, легко увидеть аналогию между самскарами (следами действий) в теории Будды о Сансаре и этими декларативными цепочками функций.

Версия #6

GRN создал 18 July 2022 21:39:31

GRN обновил 28 July 2022 21:08:16