Различия асинхронной и многопоточной архитектуры на примере Node.js и PHP

В последнее время наблюдается рост платформ, построенных на асинхронной архитектуре. На асинхронной модели построен самый быстрый в мире веб-сервер nginx. Активно развивается шустрый серверный javascript в лице Node.js. Чем же хороша эта архитектура? Чем она отличается от классической многопоточной системы? На эту тему было написано огромное множество статей, но полного понимания предмета они дали далеко не всем. Часто приходится наблюдать споры вокруг Node.js vs PHP+apache. Многие не понимают, почему некоторые вещи можно сделать на Node.js, но нельзя на PHP или наоборот — почему вполне правильный рабочий код на PHP сильно замедлится в Node.js, а то и повесит ее. В данной статье я бы хотел еще раз подробно объяснить разницу в их архитектуре. В качестве примеров двух систем, возьмем вебсервер с PHP и Node.js.

Многопоточная модель

Эта модель известна каждому. Наше приложение создает некоторое количество потоков (пул), передавая каждому из них задачу и данные для обработки. Задачи выполняются параллельно. Если потоки не имеют общих данных, то у нас не будет накладных расходов на синхронизацию, что делает работу достаточно быстрой. После завершения работы поток не убивается, а лежит в пуле, ожидая следующей задачи. Это убирает накладные расходы на создание и удаление потоков. Именно по такой системе и работает вебсервер с PHP. Каждый скрипт работает в своем потоке. Один поток обрабатывает один запрос. Потоков у нас достаточно большое количество, медленные запросы забирают поток надолго, быстрые — обрабатываются почти мгновенно, освобождая поток для другой работы. Это не позволяет медленным запросам забирать все процессорное время, заставляя подвисать быстрые запросы. Но у такой системы есть определенные ограничения. Может возникнуть ситуация, когда к нам придет большое количество медленных запросов, например работающих с БД или файловой системой. Такие запросы заберут себе все потоки, что сделает невозможным выполнение других запросов. Даже если запросу нужно всего 1мс на выполнение — он не будет вовремя обработан. Это можно решить увеличением числа потоков, чтобы они могли обработать достаточно большое количество медленных запросов. Но к сожалению потоки обрабатываются ОС, ей же выделяется и процессорное время. Поэтому чем больше потоков мы создаем, тем больше накладных расходов на их обработку и тем меньше процессорного времени выделяется каждому потоку. Ситуация усугубляется самим PHP — блокирующие операции работы с БД, файловой системой, вводом-выводом так же тратят процессорное время, не выполняя в этот момент никакой полезной работы. Тут мы поподробнее остановимся на особенностях блокирующих операций. Представим себе такую ситуацию: у нас имеется несколько потоков. Каждый обрабатывает запросы, состоящие из 1мс обработки самого запроса, 2мс на доступ и получение данных из БД и 1мс рендеринга полученных данных. Всего на каждый запрос мы тратим, таким образом, 4мс. При отправке запросов к БД поток начинает ожидать ответа. Пока данные не вернутся — поток никакой работы выполнять не будет. Это 2мс простоя на запрос в 4мс! Да, мы не можем сделать рендеринг страницы, не получив данные из базы. Мы обязаны ждать. Но ведь при этом мы получаем 50% простоя процессора! А сюда сверху можно накинуть дополнительные расходы ОС на выделение процессорного времени каждому потоку. И чем потоков больше — тем больше этих расходов. В итоге мы получаем довольно большое время

простоя. Это время напрямую зависит от длительности запросов к БД и файловой системе. Лучшее решение, которое позволяет нам полностью загрузить процессор полезной работой это переход к архитектуре, использующей неблокирующие операции.

Асинхронная модель

Менее распространенная модель, нежели многопоточная, но имеющая не меньшие возможности. Асинхронная модель построена на очереди событий (event-loop). При возникновении некоторого события (пришел запрос, выполнилось считывание файла, пришел ответ от БД) оно помещается в конец очереди. Поток, который обрабатывает эту очередь, берет событие с начала очереди, и выполняет связанный с этим событием код. Пока очередь не пуста процессор будет занят работой. По такой схеме работает Node.js. У нас имеется единственный поток, обрабатывающий очередь событий (с модулем cluster — поток будет уже не один). Почти все операции неблокирующие. Блокирующие тоже имеются, но их использование крайне не рекомендуется. Далее вы поймете почему. Возьмем тот же пример с запросом 1+2+1мс: из очереди сообщений берется событие, связанное с приходом запроса. Мы обрабатываем запрос, тратим 1мс. Далее делается **асинхронный неблокирующий** запрос к базе данных и управление сразу же передается дальше. Мы можем взять из очереди следующее событие и выполнить его. К примеру мы возьмем еще 1 запрос, проведем обработку, пошлем запрос к БД, вернем управление и проделаем то же самое еще один раз. И тут приходит ответ БД на самый первый запрос. Событие, связанное с ним помещается в очередь. Если в очереди ничего не было — он сразу же выполнится, данные отрендерятся и отдадутся назад клиенту. Если в очереди что-то есть — придется подождать обработку других событий. Обычно скорость обработки одного запроса будет сравнима со скоростью обработки многопоточной системой и блокирующими операциями. В худшем случае — на ожидание обработки других событий потратится время, и запрос обработается медленнее. Но зато в тот момент, пока система с блокирующими операциями просто ждала бы 2мс ответа, система с неблокирующими операциями успела выполнить еще 2 части 2х других запросов! Каждый запрос может выполняться чуточку медленнее в целом, но в единицу времени мы можем обработать гораздо больше запросов. Общая производительность будет выше. Процессор всегда будет занят полезной работой. При этом на обработку очереди и переходе от события к событию тратится гораздо меньше времени, чем на переключение между потоками в многопоточной системе. Поэтому асинхронные системы с неблокирующими операциями должны иметь не больше потоков, чем количество ядер в системе. Node.js изначально вообще работал только в однопоточном режиме, и для полного использования процессора приходилось вручную поднимать несколько копий сервера и распределять нагрузку между ними, например, с помощью nginx. Сейчас для работы с несколькими ядрами появился модуль cluster (на момент написания статьи все еще имеющий статус experimental). Вот тут и проясняется ключевое отличие двух систем. Многопоточная система с блокирующими операциями имеет большое время простоя. Чрезмерное количество потоков может создать много накладных расходов, недостаточное же количество может привести к замедлению работы при большом количестве медленных запросов. Асинхронное приложение с неблокирующими операциями использует процессорное время эффективнее, но более сложно при проектировании. Особенно сильно это сказывается на утечках памяти — процесс Node.js может работать очень большое количество времени, и если программист не позаботится об очистке данных после обработки каждого запроса, мы получим утечку, что постепенно приведет к

необходимости перезагрузки сервера. Также существует асинхронная архитектура с блокирующими операциями, но она гораздо менее выгодна, что можно будет увидеть далее на некоторых примерах. Выделим особенности, которые необходимо учитывать при разработке асинхронных приложений и разберем некоторые ошибки, возникающие у людей при попытке разобраться с особенностями асинхронной архитектуры.

Не используйте блокирующие операции. Никогда

Ну по крайней мере пока не поймете полностью архитектуру Node.js и не сможете аккуратно работать с блокирующими операциями.
При переходе с PHP на Node.js у некоторых людей может возникнуть желание писать код в таком же стиле, как и раньше. Действительно, если нам надо сперва считать файл, и только потом приступить к его обработке, то почему мы не можем написать следующий код:

```
var fs = require('fs');
var data = fs.readFileSync("img.png");
response.write(data);
```

Этот код правильный и вполне рабочий, но он использует блокирующую операцию. Это значит что до тех пор, пока файл не будет прочитан, очередь сообщений обрабатываться не будет и Node.js будет просто висеть, не совершая никакой работы. Это полностью убивает основную идею. В то время, пока файл читается, мы могли бы выполнять другую работу. Для этого мы используем следующую конструкцию:

```
var fs = require('fs');
fs.readFile("img.png", function(err, data){
        response.write(data);
});
```

Разберем ее подробнее: у нас происходит асинхронное чтение из файла, при вызове функции чтения управление сразу же передается дальше, Node.js обрабатывает другие запросы. Как только файл будет считан — вызывается анонимная функция, переданная в readFile вторым параметром. А точнее событие, связанное с ней, ложится в очередь и когда очередь доходит до нее — выполняется. Таким образом, мы не нарушаем последовательность действий: сперва считывается файл, потом обрабатывается. Но при этом мы не занимаем процессорное время ожиданием, а позволяем обрабатывать другие события в очереди. Это обстоятельство очень важно помнить, так как всего несколько неаккуратно вставленных синхронных операций могут сильно просадить производительность.

Используйте такой код, и вы безнадежно убьете event-loop:

```
var fs = require('fs');
var dataModified = false;
var myData;

fs.readFile("file.txt", function(err, data){
          dataModified = true;
          myData = data+" last read "+new Date();
});

while (true){
          if(dataModified)
                    break;
}

response.write(myData);
```

Такой кусок кода будет занимать все процессорное время себе, не давая обработаться другим событиям. Пока проверка не завершится успешно, цикл будет повторяться, и никакой другой код не выполнится. Если вам необходимо дождаться какого-либо события то… используйте события!

```
var fs = require('fs');
var events = require('events');
var myData;
var eventEmitter = new events.EventEmitter();

fs.readFile("file.txt", function(err, data){
          myData = data+" last read "+new Date();
          eventEmitter.emit('dataModified', myData);
});

eventEmitter.on('dataModified', function(data){
          response.write(data);
});
```

Опять-таки, этот код выполнится только после выполнения определенного условия. Только эта проверка не запускается в цикле — код, выполняющий наше условие, с помощью функции emit вызывает событие, на которое мы вешаем обработчик. Объект events.EventEmitter отвечает за создание и обработку наших событий. eventEmitter.on отвечает за выполнение кода, при возникновении определенного события.
На этих примерах можно увидеть, как неосторожное использование блокирующего кода останавливает обработку очереди событий и соответственно стопорит работу всего Node.js. Для предотвращения таких ситуаций используйте асинхронный код, завязанный на

событиях. Используйте асинхронные операции вместо синхронных, используйте асинхронные проверки наступления некоторого события.

Не используйте больших циклов для обработки данных. Используйте события

Что произойдет, если у нас возникает необходимость использовать громадный цикл работы с данными? Что если у нас должен быть цикл, работающий в течении жизни всей программы? Как мы уже выяснили выше — большие циклы приводят к блокировке очереди. Когда потребность в цикле все же возникает, мы заменяем его на создание событий. Каждая итерация цикла создает событие для последующей итерации, положив его в очередь. Таким образом, мы пропустим все события, которые ждали в очереди своего часа и после их обработки приступим к новой итерации, не блокируя очередь.

```
function incredibleGigantCycle(){
        cycleProcess();
        process.nextTick(incredibleGigantCycle);
}
```

Данный код выполнит тело цикла и создаст событие для следующей итерации. Никакой блокировки очереди событий в таком случае не будет.

Не создавайте больших операций, занимающих много процессорного времени

Иногда возникает потребность в обработке громадного объема данных или выполнение ресурсоемкого алгоритма (хотя писать злой матан на Node.js — не лучшая идея). Такая функция может занимать много процессорного времени (скажем, 500мс) и пока она не выполнится, много маленьких запросов будут простаивать в очереди. Что делать если такая функция все-таки есть и отказаться от нее мы никак не можем? В таком случае выходом может стать разбиение функции на несколько частей, которые будут вызываться поочередно как события. Эти события будут ложиться в конец очереди, тогда как события сначала могут пройти, не дожидаясь, пока наш увесистый алгоритм выполнится полностью. В вашем коде не должно быть больших последовательных кусков, не разбитых на отдельные события. Конечно есть еще выход в виде создания своего модуля на си, но это уже из другой оперы, не относящейся к вопросам асинхронного проектирования.

Внимательно следите за тем, какой тип функции вы используете

Читайте документацию, для того чтобы понять используете ли вы синхронную или асинхронную, блокирующую или неблокирующую функцию. В Node.js принято именовать синхронные функции с постфиксом Sync. В асинхронных функциях обработчик события по завершению функции обычно передается последним параметром и именуется callback. Если же вы используете асинхронную функцию там, где хотели использовать синхронную, у вас могут возникнуть ошибки.

```
var fs = require('fs');
fs.readFile("img.png", function(err, data){

});
response.write(data);
```

Разберем данный код. Начинается неблокирующее считывание файла асинхронным способом. Управление сразу же передается дальше — записывается ответ пользователю. Но при этом файл еще не успел считаться. Соответственно мы отдадим пустой ответ. Не забывайте, что при работе с асинхронными функциями, код для обработки результата функции всегда должен располагаться внутри callback-функции. Иначе результат работы непредсказуем.

Разберитесь с преимуществами асинхронных запросов

Иногда встречаются вопросы, почему приходится писать «спагетти-код» на Node.js, постоянно вкладывая друг в друга callback'и, когда на PHP все идет четко последовательно? Ведь алгоритм и там и там один и тот же.
Разберем следующий код:

```
$user->getCountry()->getCurrency()->getCode()
```

и

```
user.getCountry(function(country){
        country.getCurrency(function(currency){
                console.log(currency.getCode())
        })
})
```

И там и там обработка пойдет только после завершения всех 3х запросов. Но тут имеется существенное различие: в PHP наши запросы к базе будут блокирующими. Сперва выполняется первый запрос, имеется некоторое время простоя процессора. Потом второй запрос с простоем, аналогично третий. При асинхронной неблокирующей архитектуре мы посылаем первый запрос, начинаем выполнение каких-либо других операций, связанных с другими событиями. Когда запрос от БД возвращается — обрабатываем его, формируем второй, отсылаем, продолжаем обработку других событий. В итоге и там и там получим 3 последовательно выполненных запроса. Но в случае с PHP у нас будет некоторый простой процессора, тогда как Node.js выполнит еще некоторое количество полезного кода, и может даже успеет обработать несколько запросов, не требующих обращения к БД.

Заключение

Такие особенности Node.js необходимо знать и понимать, иначе при переходе на него с PHP вы можете не только не улучшить производительность своего проекта, но и существенно ее ухудшить. Node.js это не только другой язык и другая платформа, это другой тип архитектуры. Если вы будете соблюдать все особенности асинхронной архитектуры — вы получите преимущества от Node.js. Если вы будете упорно продолжать писать свои программы так, как писали бы их на PHP — не ждите от Node.js ничего, кроме разочарования.