Говоря про оптимизацию и отладку нашего с вами кода, нельзя обойти стороной тему «Профилирования». Во время профилирования мы с вами измеряем производительность наших функций, и мы можем сравнивать те или иные их реализации. Получается, что с помощью инструментов профилирования мы можем понять, какие части кода влияют на производительность. Влиять они могут как на процессорное время, так и на потребляемую память. В Go для этого есть инструмент профилировщик, но на слайдах здесь особо показывать нечего, поэтому давайте перейдем к коду и посмотрим всё на примерах.

В качестве примера мы с вами рассмотрим функцию подсчета последовательности Фибоначчи. Как вы видите, в функции main у меня происходит вызов функции fib и вывод результата. Функция fib как раз-таки является реализацией подсчета последовательности Фибоначчи, и используется в этой реализации рекурсия. Как мы видим, здесь вызывается та же самая функция для 2-х предыдущих чисел в последовательности.

Давайте для удобства я 1-ю часть кода закомментирую, так как она нам пока не нужна, и при запуске мы просто вызовем функцию fib и выведем результат на экран. Но так как у меня сейчас на компьютере ещё и включена запись, давайте немножко уменьшим значения, чтобы не ждать слишком долго.

Запустим нашу программу. Как мы видим, подсчёт прошёл достаточно быстро. Давайте увеличим до 40. Давайте померим, сколько времени нам потребуется для запуска этой программы: 1,7 секунды.

Давайте теперь включим наш профилировщик. Что здесь происходит? Для того чтобы собирать статистику – нужно сначала создать файл куда мы будем эту статистику собирать. В этих строчках кода у меня создается файл в корне

директории, где я запускаю нашу программу и в конструкции defer обязательно не забываем закрывать его.

Дальше происходит вызов StartCPUProfile из пакета pprof. Пакет pprof как раз-таки импортируется здесь – это и есть наш профилировщик. В данную функцию мы передаем наш файл, и с этого момента начинается сбор статистики. И, конечно же, не забываем в конструкции defer вызывать функцию StopCPUProfile, которая говорит о том, что сбор статистики нужно завершить.

Давайте запустим ещё раз и посмотрим – повлияло ли это на время выполнения? Смотрите, предыдущий раз выполнение нашей программы заняло где-то 1,7 секунды, сейчас это 2,2 секунды. На самом деле, профилировщик даёт небольшие накладные расходы на запуск нашей программы.

Как он работает? Примерно 100 раз в секунду профилировщик спрашивает runtime нашего приложения: «Что с тобой происходит?». Runtime отвечает какими-то данными, которые профилировщик использует для статистики. То есть схема работы профилировщика такая, что он сам периодически опрашивает нашу программу на предмет состояния. Поэтому существуют некоторые накладные расходы, которые уходят на то, чтобы обрабатывать эти запросы от профилировщика.

И теперь мы видим, что у нас появился файл «./cpu.pprof», который мы создали в нашем коде здесь. Давайте посмотрим, сколько наш файл весит? Весит наш файл 1,7 Кб – не так уж и много.

Теперь непосредственно мы можем запустить и проанализировать ту статистику, которую собрал наш профилировщик. Для этого нам нужно запустить команду go tool. Сказать ей, что нам нужен вызов pprof и передать туда файл с нашей статистикой. Смотрите, в данном случае мы попали уже в окружение pprof, о чём

нам говорит эта строчка. Здесь у pprof-а уже поддерживаются свои внутренние команды. Чтобы получить их список, можем воспользоваться стандартной командой help. Здесь достаточно много команд и вы можете посмотреть, и поизучать каждую команду, и как с ней работать. Мы сейчас сконцентрируемся на базовых понятиях профилировщика и как можно определить, насколько много процессорного времени у нас съедает наша программа.

Итак, давайте запустим нашу 1-ю команду, которая называется top. Эта команда показывается список функций, отсортированные по времени потребления процессора. И мы видим, что 100% времени практически, уходит на наш метод fib, что и не удивительно, потому что только он у нас и вызывается.

Помимо такого top мы можем провалиться внутрь конкретного метода функции и посмотреть более подробно. Здесь мы уже видим listing кода и время, которое пришлось на определенный фрагмент кода. И тут, как мы можем заметить, что 1 секунду с лишним пришлось как раз-таки на этот рекурсивный вызов. Что не удивительно, потому что у нас вызов функции fib происходит и здесь, и внутри каждой этой функции. И в данной точке мы просто ждём, пока эти функции не завершатся.

Помимо такого интерактивного режима работы у pprof-а ещё есть web interface. Для того чтобы в него перейти, нам нужно выйти отсюда и запустить pprof с параметром http и указать, на каком port-у и по какому url-у мы будем это делать. Давайте запустим и перейдем в web interface. В web interface-е у нас отображается схема вызовов наших функций. И здесь мы видим, что самой большой функцией у нас является функция fib и то, что она занимается 1000 миллисекунд из 1040. Остальные функции не так велики, и здесь даже у них отображается ноль, потому что значения занимаемого времени процессора у них слишком малы, чтобы отображаться здесь на графике.

Какие выводы мы можем из этого сделать? На самом деле, функция fib здесь представлена в агрегирующем виде, то есть у нас происходит множество запусков этой функции, и в совокупности все эти запуски заняли 1000 миллисекунд. Об этом нам говорит ещё и размер прямоугольника с этой функцией. То есть, как вы можете видеть – эти две, они по сравнению с прямоугольником функции fib достаточно маленькие. То есть на протяжении работы нашей программы мы несколько раз запускали эту функцию, и в совокупности она заняла 1000 миллисекунд времени нашего процессора.

Помимо такого отображения мы можем посмотреть ещё и другой граф, на котором более наглядно будет видно, что наша функция запускалась несколько раз. Но это, на самом деле, не удивительно, потому что у нас там рекурсия – не самый оптимальный вариант реализовать этот алгоритм, но всё же мы его взяли, чтобы наглядно показать этот процесс. Во время рекурсии наша функция постоянно вызывает саму себя и в итоге доходит до конечного вызова, который считает и отрабатывает числа. Выходит и передает результат выше по стеку и так далее, пока снова мы не перейдем к нашей первой функции, и наша программа не закончится.

Но профилировать такую простую функцию как Фибоначчи – достаточно просто. Давайте разберем более сложный пример. Допустим, к нам в руки попался проект, про реализацию которого мы практически ничего не знаем, но мы знаем, что, например, этот проект реализует алгоритмы фракталов и генерирует какую-то картинку. Данный алгоритм, на самом деле, этим и является. Я скопировал этот код из интернета, и что он делает? Мы можем запустить его и посмотреть, что в результате у нас сгенерируется изображение, которое будет показывать нам фракталы. Эту картинку out.png мы получили.

Давайте её откроем. Смотрите, что мы сгенерировали с помощью кода на Go. На самом деле, это достаточно сложный алгоритм, и он занимает очень много процессорного времени.

Давайте поизучаем, можем ли мы его оптимизировать? Для начала определим, сколько времени занимает текущая реализация. Давайте для чистоты эксперимента мы скомпилируем этот код в бинарник и запустим его с фиксацией времени. После запуска мы получим время, которое нам понадобилось для работы нашей программы. И мы видим, что 8 секунд ушло на то чтобы сгенерировать наш файл, то есть нашу картинку с фракталами.

Что мы можем сделать? Мы ничего не знаем про реализацию этого алгоритма, да и алгоритм сам по себе не простой. Но мы знаем, что у нас есть такой мощный инструмент, как профилировщик. Давайте найдем какие-то узкие места в нашем коде. Для этого у нас уже есть нужный нам код, который создаст файл для профилировщика. И мы стартуем наш профилировщик по потреблению CPU и в defer по традиции останавливаем наш профилировщик.

И как вы видите, уже после первого запуска у нас появился файл pprof.out. Давайте теперь его исследовать. Вызовем go tool pprof и скажем, что нам нужно исследовать этот файл. Увеличим немножко окно, чтобы было удобнее. Давайте сначала поработаем в интерактивном режиме: запустим команду top. И здесь мы видим, что очень много времени ушло на функцию pixel. Давайте посмотрим, есть ли такая функция в коде? Да, как мы видим, функция такая присутствует. Но давайте исследовать дальше.

Посмотрим на функцию main.main. И здесь мы видим, что на этом шаге мы застряли больше всего: нам потребовалось 5 и почти 9 десятых секунды. Давайте тоже посмотрим, есть ли такой фрагмент в коде. Да, вот мы его видим на строке

51, всё совпадает – это функция main, строка 51. И здесь мы видим, что больше всего мы застряли именно на этой строчке. Поехали дальше.

Теперь нам нужно исследовать эту функцию. Давайте сделаем list main.createCol и то же самое – здесь мы видим, что большее количество времени заняла эта строчка. И там уже как раз наша функция pixel – её мы тоже можем вызвать. Давайте, чтобы было видно, сделаю следующим образом. И нас интересует функция pixel. И тут мы видим, что 4 секунды нам пришлось обрабатывать наш цикл.

И, на самом деле, здесь может произойти некоторый тупик. То есть что мы тут видим? – Мы уперлись в реализацию алгоритма и, на самом деле, нам нечего здесь оптимизировать, если только не уйти в глубоко реализацию алгоритма и что-то придумать с алгоритмом. Но алгоритм является достаточно сложным, и чтобы в нём разобраться и как-то его оптимизировать – может уйти очень много времени.

Давайте для интереса ещё посмотрим на web interface. И что мы здесь видим? У нас также отрисовалась наша схема, и на ней мы тоже видим, что большее количество времени заняла функция pixel. В принципе, что мы могли заметить в нашем интерактивном режиме, когда исследовали данную проблему? Здесь есть очень много разных других вызовов, но они по сравнению с функцией pixel не такие огромные.

То есть мы приходим к выводу, что нам нужно как-то оптимизировать нашу функцию pixel. Давайте опять её найдем и взглянем, что здесь происходит. Функция pixel у нас как раз-таки рассчитывает и отрисовывает фракталы по внутреннему алгоритму. И сложно здесь что-то, на самом деле, оптимизировать. Нам придется очень долго в этом разбираться. Чтобы посмотреть, есть ли ещё

какие-то варианты для оптимизации – мы сейчас воспользуемся ещё одним инструментом, который называется трассировщик.

Давайте вместо профилировщика мы здесь теперь запустим трассировщик. Происходит это следующим образом: есть пакет trace, где мы можем start-ануть наш трассировщик, и в defer, конечно же, нам нужно его stop-нуть.

Что такое трассировщик? Трассировщик покажет нам стек вызовов наших функций и время их отработки. В отличие от профилировщика, который, как мы говорили, с некоторой периодичностью спрашивает нашу программу: «А скажи мне своё состояние?», – и фиксирует это состояние в статистике. У трассировщика немножко другая схема работы – он изначально говорит нашей программе: «При каких-либо изменениях сообщи мне, что поменялось». В таком случае у трассировщика получается более детальная информация о состоянии нашей системы.

Давайте запустим трассировщике и посмотрим на результат. Нам нужно скомпилировать ещё раз наш проект и также запустим его с фиксированием времени. После того, как программа отработает, мы должны получить новый файл trace.out. Как видите, вот он здесь. Посмотрим на размер наших файлов. Смотрите, у нас pprof – 4 Кб, а трассировщик у нас генерировал файл аж 24 Кб, что, конечно, намного больше и говорит о том, что данных у трассировщика про наши вызовы больше.

Теперь давайте запустим наш трассировщик. Делается это той же командой go tool, только теперь мы говорим, что мы хотим по-trace-ить наши собранные данные. Запускаем и сразу переходим в web interface. Здесь, на самом деле, очень много разных разделов. Вы можете самостоятельно поизучать и поисследовать их. Мы посмотрим на раздел Trace.

Смотрите, что здесь происходит: тут мы видим статистику по горутинам, по куче, выделению памяти, thread-ам. У нас есть процессоры, на которых запускались горутины, и всё это достаточно в детальном виде здесь отображается. Мы можем позумить и смотрите, мы можем даже дойти до наносекунд. Вот мы видим, как у нас start-yет trace-ер, вот у нас стартует main функция наша. Давайте посмотрим, что ещё есть интересного. Смотрите, здесь происходит что-то интересное: в данном случае у нас виден фрагмент, во время которого стартует garbage collector и работает с нашей память.

Какие выводы мы можем сделать из данного trace-а? Вот наша основная функция main, и она отрабатывает очень долго. Смотрите, у нас идёт вызов main, потом ещё раз, то есть мы видели, что у нас в циклах происходят расчеты. И всё это происходит в 1 поток. И, на самом деле, чтобы улучшить производительность, мы можем эти кирпичики, которые считают данные – распараллелить и пустить по разным потокам. В данном случае мы видим, что всё это происходит в рамках одного потока, и они вызываются друг за другом последовательно.

Давайте теперь перейдем в код и посмотрим, как мы можем это распараллелить. Итак, в нашем профилировщике мы видели, что больше всего по времени у нас занимает метод createCol. Если идти сверху с функции main, то первым делом, мы застревали на этом методе.

Смотрите, в этот метод мы передаем ширину и высоту, и по этим данным у нас происходит цикл, который рассчитывает как раз-таки pixel наш. Что мы здесь можем сделать? На самом деле, тут цикл с вычислениями, то есть эти вычисления кроме как от переданных параметров – ни от чего более не зависят, поэтому мы можем вынести эти вычисления в разные потоки и вычислять их параллельно.

Делать это мы будем, конечно же, с помощью горутин. Смотрите, у нас в цикле запускается подсчёт нашего кода для каждого значения из ширины. То есть что мы можем сделать? Мы можем этот внутренний цикл для каждого значения ширины запустить в отдельной горутине, то есть у нас запустится столько горутин, сколько будет значений нашей ширины width.

Давайте. Для того, чтобы контролировать наши горутины, создадим WaitGroup, о котором мы уже знаем и скажем, что в WaitGroup мы добавим количество горутин, равных нашей ширине. Цикл по ширине как раз будет внутри запускать горутину, в которую мы передадим нужный нам идентификатор… То есть не идентификатор, а значение ширины. И этот цикл перенесем внутрь.

Внутри нам обязательно нужно сделать Done для нашего WaitGroup-а после того, как завершим подсчеты и передадим сюда значение из итерации по нашей ширине. Да, и не забываем, конечно же, после того как мы запустили все подсчёты в разных горутинах – нам надо дождаться, пока они посчитаются в параллельном режиме.

Итак, что здесь мы сделали? Мы создали WaitGroup по количеству горутин равному нашему параметру width. Потом в цикле для каждого width мы запускаем новую горутину, в которой считаем pixel-и для этого значения ширины. После этого мы делаем Done нашей WaitGroup и после всего цикла по нашей ширине мы ожидаем завершения всех наших горутин.

Давайте запустим нашу программу в этом варианте и посмотрим, что изменилось. И тоже зафиксируем время. Смотрите, мы получили значительный прирост по времени, до этого у нас уходило на нашу программу более 5 секунд, сейчас мы получили 2,7 секунды.

Посмотрим теперь на tracer. Так как у нас появилось много горутин, tracer тоже должен показать интересную картину, давайте обновим. Смотрите, как интересно: теперь наш tracer показывает активность на нескольких процессорах, доступных нашей системе. У меня на ноутбуке их 8, поэтому здесь мы параллельно в 8 горутинах считали наши данные. Как вы помните, до этого у нас был всего 1 поток, в котором мы считали наши числа. И тем самым, мы всего лишь запустив нужный фрагмент в параллельном режиме, практически в 2 раза увеличили скорость работы нашей программы.