Текстовый вариант видеоурока из предыдущего шага

Последний инструмент, который мы рассмотрим это context. Пакет context позволяет нам контролировать процессы выполнения горутин. Мы можем задавать им таймауты, дедлайны, посылать различные сигналы и отменять какие-то асинхронные операции. Это касается, и горутин, и данный способ еще используется для запросов при межпроцессорном взаимодействии.

Для того, чтобы создать context, мы можем воспользоваться пакетом context и метод Background создает начальное, дефолтное представление контекста. Ну, как правило, от этого context.Background наследуется все остальное. Давайте посмотрим, как с этим можно работать и что можно делать с помощью context. Итак, мы создаем context с помощью метода WithCancel, который принимает в качестве параметра, скажем так, родительский context, то есть базовый, на основе которого создает следующий context, который можно будет отменить. Поэтому, в качестве базового, мы отдаем новый context.Background. Вот этот метод нам возвращает структуру context, который является новой структурой. И метод WithCancel отдает два значения. Первое значение это сам context, структура context, а второе значение — это функция CancelFunc, то есть это функция отмены нашего context. Эту функцию мы держим в переменной cancel.

Дальше происходит следующее, в цикле мы 10 раз запускаем в горутине функцию sendData. Ну, допустим, это некая функция, которая должна отправлять какие-то данные. Давайте посмотрим, что внутри этой функции. Здесь мы завели таймер на количество секунд, кратных номеру вызова этой функции. То есть если мы в цикле вызываем 10 раз, то каждая итерация будет инкрементировать на один этот номер. И все в том же select, который мы рассматривали, мы смотрим на этот таймер. Если он заканчивается, то мы считаем, что данные успешно отправились. А если мы завершаем наш context и в функцию Done у нас приходит канал,

который сигнализирует о том, что context завершился. При этом, мы печатаем, что процесс отменен потому, что мы завершили context, сказали ему отмениться и выходим из функции.

Давайте запустим и посмотрим, что произойдет. Смотрите, здесь у нас первая горутина успела отработать, мы успели отправить данные, а дальше все остальные отменились. Почему так произошло? Потому что при первом вызове sendData у нас передалось число 1 и в горутине мы сделали таймер на 1 секунду. В остальных это было 2 секунды, 3 секунды и так далее, соответственно. После этого цикла мы сделали Sleep тоже на 1 секунду, после чего вызвали отмену нашего context. То есть мы вызвали функцию, которая была в переменной cancel. Эта функция отменила наш context и далее все операции в нашей функции попадали вот в этот кейс потому, что context наш был отменен.

Это очень удобный и распространенный механизм отмены и управления асинхронных параллельных операций. Поэтому в go очень часто в функциях первым параметром передается context. Вы встретитесь с множеством функций в go, где первым параметром в функции принимается именно context. В этом примере мы с вами использовали context и обновили его вручную. То есть мы явно в коде вызвали метод cancel. Но, кроме этого, можно сделать так, чтобы context отменился самостоятельно. Давайте мы здесь вынесем отдельно наш parent context, который является context.Background, но вместо WithCancel вызовез функцию WithTimeout. Передадим туда наш родительский context и, допустим, таймаут в две секунды. Здесь нам уже метод cancel не нужен, мы его можем игнорировать. В этом случае, через одну секунду, наш context автоматически отменится.

Давайте запустим и посмотрим на результат. Так, у нас ничего не произошло потому, что главная горутина закончилась быстрее. Давайте поставим sleep в 2

секунды. Итак, как мы видим, процесс 1 успешно отправлен. Остальные процессы отменились. Потому что процесс занимает 1 секунду. Давайте попробуем увеличить таймаут на 3 секунды, например, и здесь тоже увеличим. В этом случае у нас должно пройти больше успешных выполненных процессов. Как мы видим, через 3 секунды context отменяется и все остальные горутины, которые использовали и были завязаны на этом context так же отреагировали на эту отмену. Но и помимо context.WithTimeout есть возможность создать context.DeadLine. Что это означает? В этом случае мы передаем все так же базовый context, но теперь у нас не таймаут, а конкретное время. То есть тут context дает возможность его отменить в определенный момент времени. Мы заранее задаем во сколько часов, во сколько секунд этот context должен отмениться. И в назначенное время произведет событие отмены и все горутины, которые были завязаны на этот context узнают об этом и обработают.

Данный механизм контекстов позволяет нам управлять не только горутинами нашего сервиса, но запросами между различными сервисами. Часто бывает так, что клиент, когда запрашивает сервер, сервер выполняет какую-то большую работу, чтобы ответить на этот запрос. Но в процессе выполнения этой работы, клиент может отменить свой запрос. В этом случае, как правило, клиент отменяет, а сервер ничего не знает об этом и продолжает выполнять свой запрос. Механизм контекстов позволяет отменить context так же и на сервере и сигнализировать об этом в функцию, которая выполняем большой запрос. Это очень актуально для современных микросервисных систем, когда у нас рамках одного приложения могут работать и общаться разные микросервисы посредством запроса и ответа. Данный механизм позволяет сэкономить много ресурсов, не перерабатывая лишние запросы, которые клиент, например, уже отменил. Поэтому в go хорошей практикой считается прокидывать context во все уровни нашего сервиса.