GO НА СОБЕСЕДОВАНИИ

Максим Байтович

Максим Байтович

Go на собеседовании: от новичка до Senior

https://litres.ru/74310082
SelfPub; 2026

Аннотация

Go — язык, который наказывает за непонимание. Можно выучить синтаксис за выходные и писать работающий код годами, но на собеседовании среднего уровня вдруг выяснится, что ты не знаешь, как устроен планировщик горутин и почему твой сервис падает под нагрузкой.

Эта книга — не справочник по синтаксису. Это наставник, который проведёт тебя через все темы, спрашиваемые на реальных собеседованиях: от философии языка и указателей до GMP-модели, escape analysis и паттернов конкурентности. Ты узнаешь не только что спросят, но и почему это важно, и как правильно ответить, чтобы интервьюер увидел в тебе зрелого инженера, а не просто вызубрившего ответы кандидата.

Каждая глава построена как живой разговор: минимум таблиц, максимум связного объяснения с боевыми примерами кода. Алгоритмические секции разобраны с акцентом на то, что действительно важно в Go — работу с памятью, аллокации и

идиоматичность. Читай по главе в день, и через две недели ты будешь готов к собеседованию любого уровня.

Содержание

Go на собеседовании: от новичка до Senior

Глава 1. Go — это не C ++ с другими скобками

Представь себе 2007 год, офис Google. Кен Томпсон, Роб Пайк и Роберт Гриземер ждут сборки очередного C++ сервиса. Сорок пять минут. Это не преувеличение, это реальность того времени. Три человека, создавшие Unix, Plan 9 и Inferno, смотрят на монитор и думают: неужели нельзя сделать язык, который компилируется мгновенно, читается как хороший скрипт и не заставляет разработчика помнить тысячу способов выстрелить себе в ногу.

Так родился Go. И чтобы понять этот язык, недостаточно выучить синтаксис. Нужно понять, какую боль он лечит и от каких инженерных привычек требует отказаться.

Простота как осознанный выбор

Первое, что нужно усвоить: простота в Go — это не ограничение, это философское решение. Создатели языка созна-

тельно отказались от десятков фич, которые есть в C++, Java или Python. Здесь нет классов, нет наследования, нет исключений, нет перегрузки операторов, нет неявного приведения типов, нет аннотаций. Это не потому, что они не умели это делать, а потому, что каждая такая фича рано или поздно начинает работать против читаемости кода в большой команде.

Роб Пайк сформулировал это так: «Go должен помещаться в голове одного разработчика». Когда ты открываешь чужой код на Go, ты не гадаешь, какой магический метод вызовется при передаче аргумента. Ты видишь вызов функции и знаешь, что произойдёт. Всё явно, всё на поверхности. Это неудобно, когда ты пишешь код в одиночку и хочешь выразить мысль покороче. Это спасительно, когда ты в три часа ночи разбираешь чужой баг в продакшене.

Композиция вместо наследования

В объектно-ориентированных языках нас учат строить иерархии. Животное, потом млекопитающее, потом собака. Каждый следующий уровень наследует поведение предыдущего. Звучит элегантно, пока ты не обнаружишь, что бизнес-логика не укладывается в красивое дерево. Сегодня твоя учётная запись пользователя должна вести себя как платёжный аккаунт, завтра — как профиль соцсети, послезавтра — как то и другое одновременно.

Go предлагает другой путь. Вместо «является» используется «имеет». Ты не говоришь «Service является Logger-ом», ты говоришь «Service имеет Logger внутри». Это называется композицией через встраивание. Ты берёшь маленькие, независимые кусочки поведения и собираешь из них нужную функциональность.

На собеседовании это один из маркерных вопросов. Когда кандидат начинает рассказывать про плюсы наследования, это нормально. Когда он объясняет, почему в Go наследования нет, и почему композиция снижает связность кода — это уже уровень выше.

Ошибки как значения

Ещё один культурный шок для новичка — обработка ошибок. Вместо try-catch-finally, который прячет поток управления за невидимыми прыжками по стеку, Go возвращает ошибку как обычное значение. Функция возвращает результат и ошибку, вызывающий код проверяет ошибку и принимает решение.

Да, это приводит к обилию строк с проверкой «if err != nil». Новички жалуются на это, ветераны пожимают плечами. Потому что явность важнее краткости. Когда ты читаешь код, ты видишь каждое место, где что-то может пойти не так. Ты не гадаешь, вылетит ли исключение из недр библиотеки,

которую ты вызвал. Всё честно, всё на виду.

И да, есть способы сделать эту проверку менее назойливой. Можно оборачивать ошибки, добавляя контекст по мере подъёма по стеку вызовов. Можно использовать пакет errors из стандартной библиотеки, чтобы проверять типы ошибок. Но базовый принцип остаётся: ошибка — это значение, и относиться к ней нужно как к значению.

Ортогональность и минимум магии

Есть понятие, которое любят архитекторы: ортогональность. Это когда фичи языка не пересекаются, каждая решает свою задачу и не конфликтует с другими. В Go это проявляется на каждом шагу. Ключевое слово range работает одинаково с массивами, слайсами, мапами и каналами. Тебе не нужно помнить четыре разных синтаксиса. Интерфейсы удовлетворяются неявно: если твой тип реализует все методы интерфейса, он подходит. Не нужно писать длинное объявление о намерениях.

Ещё один пример — zero value. В Go переменная, объявленная без инициализации, всегда получает осмысленное значение по умолчанию. Число — ноль, строка — пустая строка, указатель — nil. Нет неопределённого поведения, нет мусора в памяти. Более того, некоторые типы спроектированы так, что их zero value готово к использованию без вы-

зова конструктора. Ты можешь объявить sync.Mutex и сразу начать его использовать, не вызывая никакой init. Это не случайность, это инженерное решение, продуманное до мелочей.

Дженерики без фанатизма

Долгое время в Go не было дженериков, и это было предметом жарких споров. В 2022 году их наконец добавили, но добавили так, чтобы не сломать философию языка. Дженерики в Go существуют для написания библиотек и обобщённых структур данных, а не для бизнес-логики. Если ты ловишь себя на том, что каждую функцию начинаешь с квадратных скобок и параметра типа — скорее всего, ты пишешь не на Go, а на C++ с синтаксисом Go.

Что хочет услышать интервьюер

На собеседовании вопрос «почему Go» или «что тебе нравится в языке» — это проверка зрелости. Плохой ответ: «там горутины быстрые и синтаксис простой». Хороший ответ — про философию. Про явность вместо магии. Про композицию вместо наследования. Про то, что язык создан для инженеров, которые работают в больших командах над долгоживущими проектами, и каждое решение в дизайне языка подчинено этой цели.

Кстати, вопрос на засыпку, который иногда задают в нача-

ле собеседования: что произойдёт, если попытаться записать значение в nil-мапу. Правильный ответ — паника. Но важнее объяснить почему. Zero value для мапы — это nil. Читать из nil-мапы можно, вернётся zero value типа. А писать нельзя, потому что под капотом нет аллоцированной памяти для хеш-таблицы. Это маленький пример того, как философия zero value проявляет себя в реальном коде.

Глава 2. Указатели и семантика значений

Если первая глава была про мышление, то эта — про механику. Указатели в Go — тема, которая вызывает больше всего споров и ошибок, особенно у тех, кто пришёл из языков без ручного управления памятью. И одновременно это тема, которая позволяет отличить разработчика, действительно понимающего платформу, от того, кто просто пишет работающий код.

Что скрывается за звёздочкой

Начнём с самого простого. Указатель — это не значение, а адрес в памяти, по которому значение лежит. Когда ты создаёшь переменную, компилятор выделяет ей место где-то в оперативной памяти. У этого места есть числовой адрес. Указатель хранит этот адрес. Звёздочка перед указателем означает «пойди по этому адресу и дай мне то, что там лежит». Амперсанд перед переменной означает «дай мне адрес этой переменной».

Важно понимать, что в Go указатели существуют, но они гораздо безопаснее, чем в C или C++. Здесь нет арифмети-

ки указателей — ты не можешь прибавить единицу к адресу и получить доступ к соседней ячейке памяти. Нет ручного освобождения памяти — за этим следит сборщик мусора. Указатель в Go — это просто способ сослаться на одно и то же значение из разных мест, не более.

Главный вопрос: значение или указатель

Существует вопрос, который задают почти на каждом собеседовании по Go. Он звучит примерно так: «Вот у тебя есть структура и метод. Ты сделаешь receiver по значению или по указателю? Почему?». И ответ на него не может быть односложным.

Первый критерий — мутабельность. Если метод должен изменить состояние структуры, только указатель. Значение передаётся как копия, любые изменения внутри метода коснутся копии и исчезнут вместе с ней. Это частая ошибка новичков: пишешь метод с value receiver, меняешь поля, а снаружи ничего не происходит.

Второй критерий — размер структуры. Если структура большая, копирование при каждом вызове метода стоит тактов процессора и байт на стеке. Но здесь есть важный нюанс, который упускают многие кандидаты. Копирование значения на стеке — операция быстрая, процессор заточен под линейное чтение памяти. А вот указатель часто означает ал-

локацию в куче, за которой придёт сборщик мусора. Поэтому для маленьких структур из трёх-пяти полей копирование может быть эффективнее, чем возня с GC. Эмпирическая граница — примерно 64–128 байт, но без бенчмарка это гадание.

Третий критерий — природа типа. Есть типы, которые физически нельзя или бессмысленно копировать. Самый хрестоматийный пример — sync.Mutex. Если ты скопируешь мьютекс, копия будет представлять собой совершенно новый, независимый механизм блокировки. Заблокировал оригинал — копия об этом не знает. Это прямой путь к гонкам данных. Go даже имеет встроенный анализатор go vet, который кричит, если ты копируешь мьютекс. То же самое касается sync.WaitGroup, пулов соединений с базой данных и некоторых других типов, представляющих ресурс или состояние.

Семантика — это контракт

На уровне Senior от тебя ждут уже не просто перечисления критериев, а разговора о семантике. Билл Кеннеди, автор культового курса Ultimate Go, ввёл различение value semantics и pointer semantics. Это два разных контракта, два разных способа думать о данных.

Семантика значений означает, что каждый владеет своей

копией данных. Ты передал значение в функцию — функция работает со своей копией, оригинал в безопасности. Это похоже на примитивные типы: когда ты передаёшь число, ты не ожидаешь, что оно изменится снаружи. Это безопасно, предсказуемо и иммутабельно по своей сути.

Семантика указателей означает совместное владение. Ты передал указатель — теперь у тебя и у функции одна и та же память. Изменения видны всем. Это гибко, но требует думать о конкурентном доступе и времени жизни объекта.

Ключевое правило, которое демонстрирует зрелость разработчика: в рамках одного типа нужно придерживаться одной семантики. Не смешивать методы с value receiver и pointer receiver без крайней необходимости. Если тип представляет собой нечто «живое» и изменяемое — соединение с базой, кеш, сессию пользователя — вся работа идёт через указатель. Если тип — просто контейнер для данных вроде координат точки или временной метки — он передаётся по значению.

Интерфейсы и указатели: вечный подвох

Есть одна ситуация, в которую попадаются почти все. Ты определяешь интерфейс, потом определяешь тип с методом и пытаешься присвоить значение этого типа переменной интерфейсного типа. И получаешь ошибку компиляции.

Разгадка в том, на чём именно определён метод. Если метод определён с pointer receiver, то он принадлежит указателю на тип, а не самому типу. Переменная-значение такого типа не имеет этого метода. Поэтому интерфейс удовлетворяет только указатель. И наоборот: если метод на значении, интерфейс удовлетворяют и значение, и указатель. Указатель всегда может разыменоваться до значения, а вот значение не может само стать указателем.

Кстати, Go умеет автоматически брать адрес переменной, если ты вызываешь метод с pointer receiver у переменной-значения. Но это работает только если переменная адресуема — то есть имеет конкретное место в памяти. Результат функции, литерал или константа не адресуемы, и автоматического взятия адреса не произойдёт.

Что остаётся после прочтения

В сухом остатке: выбор между значением и указателем — это не оптимизация, а семантическое решение. Ты решаешь, будет ли твой тип вести себя как примитив или как разделяемый ресурс. И это решение влияет на читаемость, безопасность и производительность кода одновременно.

На собеседовании не бойся начать ответ с этих слов: «Выбор receiver зависит от семантики типа». Это сразу показы-

вает, что ты не просто выучил правило, а понимаешь, откуда оно взялось.

Глава 3. Горутины и планировщик

Это глава, которую Senior-разработчики называют «водоразделом». До неё ты пишешь на Go. После неё ты понимаешь, на чём твой код работает.

Планировщик горутин — не абстрактная теория для докладов на конференциях. Это штука, которая напрямую влияет на то, упадёт ли твой сервис под нагрузкой или выдержит десятикратный всплеск. Когда на собеседовании тебя спрашивают про GMP-модель, интервьюер хочет убедиться, что ты не относишься к горутинам как к магии.

Почему не потоки

Сначала был железный век: один поток — одна задача. Хочешь параллельно обработать сотню клиентов — создавай сотню потоков операционной системы. Каждый поток требует своего стека памяти, обычно около мегабайта. Стек нужно зарезервировать сразу, потому что операционная система не умеет его динамически двигать. Сотня потоков — сто мегабайт только на стеки. Тысяча потоков — гигабайт. И это не считая накладных расходов на переключение контекста, когда ядро операционной системы сохраняет состояние одного потока и восстанавливает состояние другого.

Потом был век асинхронного программирования. Коллбэки, promises, event loop. Один поток операционной системы крутит бесконечный цикл и раздаёт задачи. Память расходуется экономно, но код превращается в цепочку нелинейных вызовов, которую невозможно отлаживать. Ты ждёшь данные из базы и передаёшь коллбэк, который вызовется когда-нибудь. Ждёшь ответ от внешнего API — ещё коллбэк. Ошибка на пятом уровне вложенности, стек вызовов потерян, ты смотришь в монитор и тихо страдаешь.

Go предлагает третий путь. Ты пишешь обычный синхронный код, линейный и читаемый, а рантайм превращает его в асинхронное исполнение. Горутина выглядит как функция, запущенная ключевым словом go, и внутри неё ты просто делаешь последовательные вызовы, как будто никакой конкурентности нет:

```
func handleClient(conn net.Conn) {
data := readRequest(conn) // Блокируемся, ждём данные
result := queryDB(data) // Блокируемся, ждём базу
conn.Write(result) // Блокируемся, ждём отправку
}

go handleClient(clientConn)
```

Выглядит как синхронный код. Но под капотом, когда горутина блокируется на чтении из сокета или ожидании базы данных, она не занимает поток операционной системы. Поток освобождается и берёт в работу другую горутину. Та самая сотня или тысяча горутин крутится буквально на нескольких потоках ОС.

Как это устроено: GMP

Аббревиатура, которую нужно знать как таблицу умножения. G — это горутина. Не поток, не процесс, а именно горутина: лёгкий контекст исполнения со своим стеком, своим счётчиком команд и своей очередью запланированных операций. Стек горутины начинается с двух килобайт, а не с мегабайта, и может динамически расти и сжиматься.

M — это machine, поток операционной системы. Настоящий, осязаемый, которым управляет ядро ОС. M исполняет код горутин. В любой момент времени один M привязан к одному ядру процессора и исполняет ровно одну горутину.

P — это processor, логический процессор Go. Это не физическое ядро и не поток, это структура данных в рантайме, которая хранит очередь горутин, готовых к исполнению, и связывает G с M. Количество P задаётся переменной GOMAXPROCS и по умолчанию равно количеству физических ядер. Если у тебя четырёхядерный процессор,

GOMAXPROCS будет равен четырём, и это значит, что не более четырёх горутин могут исполняться одновременно в любой момент времени.

Когда ты запускаешь горутину, она помещается в локальную очередь того P, на котором сейчас работает текущий поток. Если локальная очередь переполнена, горутина попадает в глобальную очередь. Когда M освобождается, он первым делом заглядывает в локальную очередь своего P. Если там пусто, он идёт в глобальную очередь. Если и там пусто, он начинает воровать горутины у других P. Эта модель называется work stealing и именно она позволяет равномерно распределять нагрузку.

Кооперативная многозадачность и что бывает, когда о ней забывают

Планировщик Go не вытесняющий. Горутина сама должна отдать управление, когда она готова уступить процессор. Происходит это в определённых точках, которые называются точками планирования: вызов функции, операция с каналом, блокирующий системный вызов, работа с мьютексом.

До версии Go 1.14 это создавало проблемы. Если горутина входила в тесный цикл без вызовов функций, она могла монополизировать поток на неопределённое время. Другие горутины, привязанные к тому же P, голодали. С версии

1.14 добавили асинхронную вытесняемость: рантайм посылает горутине сигнал с просьбой уступить, и в безопасных точках она это делает. Но понимать природу кооперативности всё равно важно, потому что асинхронная вытесняемость не решает всех проблем.

На собеседовании это часто проверяют простым вопросом: «Что будет, если запустить горутину с бесконечным циклом, который не делает системных вызовов, но внутри пустой select?». Правильный ответ: пустой select заблокирует горутину навсегда, и планировщик её вытеснит. А вот пустой for без select — это другое дело. До Go 1.14 такая горутина могла бы устроить де-факто взаимную блокировку на одном P. После 1.14 её вытеснит асинхронный механизм, но остальные горутины того же P всё равно будут получать меньше процессорного времени, чем хотелось бы.

Системные вызовы и сетевая магия

Есть два типа блокирующих операций, и планировщик обрабатывает их по-разному.

Первый тип — системный вызов. Например, чтение файла с диска. Это операция, которая передаётся ядру операционной системы. Когда горутина делает такой вызов, она блокирует свой M. Планировщик не может на этом M выполнять другие горутины. Поэтому он отсоединяет P от забло-

кированного M, прикрепляет P к другому, свободному M, и работа продолжается. Когда системный вызов завершается, горутина пытается вернуться к своему P. Если он занят, она встаёт в очередь.

Второй тип — сетевые операции. Чтение из сокета, ожидание нового соединения, отправка данных. Здесь Go использует netpoller — компонент рантайма, который работает поверх асинхронного ввода-вывода операционной системы, такого как epoll в Linux. Когда горутина блокируется на сетевой операции, она не блокирует M. Вместо этого она регистрируется в netpoller-е и засыпает. M свободен и сразу берёт другую горутину. Когда сетевые данные приходят, netpoller будит горутину и возвращает её в очередь готовых к исполнению.

Именно поэтому Go так хорош для сетевых сервисов. Тысячи одновременных соединений, каждое в своей горутине, и всё это крутится на нескольких потоках ОС, потому что большую часть времени горутины спят в netpoller-е и не потребляют ресурсов.

Сколько горутин — это нормально

Есть соблазн думать, что раз горутины такие лёгкие, их можно создавать без счёта. Это не так. Стек горутины начинается с двух килобайт, но он может вырасти. Если горутина

делает глубокую рекурсию или выделяет много локальных переменных, стек расширяется, копируясь в новую область памяти. Это не быстро и не бесплатно.

Кроме того, каждая горутина — это структура в памяти, занимающая несколько сотен байт. Миллион горутин — это гигабайты памяти только на метаданные. Плюс нагрузка на планировщик, который должен эти горутины обслуживать.

Эмпирическое правило: без проблем можно создавать десятки и сотни тысяч горутин, особенно если они большую часть времени ждут ввода-вывода. Но если каждая горутина активно считает, оптимальное количество — это GOMAXPROCS умноженное на некоторый коэффициент, зависящий от характера нагрузки.

Что хочет услышать интервьюер

Когда тебя спрашивают про GMP, не начинай с расшифровки аббревиатуры. Начни с проблемы, которую решает планировщик. Расскажи про дороговизну потоков ОС и прочитаемость синхронного кода. Покажи, что ты понимаешь, зачем это всё придумано, а не просто вызубрил схему.

Объясни, как M отвязывается от P при системном вызове и как netpoller обрабатывает сетевые операции без блокировки потока. Упомяни work stealing и асинхронную вытесняе-

мость. И обязательно добавь, что знание планировщика помогает на практике: когда ты видишь, что сервис захлёбывается под нагрузкой, ты смотришь не только на бизнес-логику, но и на то, не создаёшь ли ты горутины быстрее, чем они завершаются, и не забиваешь ли ты локальную очередь P.

Один из коронных вопросов на засыпку звучит так: «Что произойдёт, если из горутины вызвать C-код через cgo?». Правильный ответ: cgo-вызовы идут в обход планировщика Go и могут заблокировать M на неопределённое время. Поэтому для тяжёлых cgo-операций нужно либо выделять отдельные горутины с запасом по GOMAXPROCS, либо выносить такие вызовы в отдельный пул потоков.

Глава 4. Память: стек, куча и сборщик мусора

В предыдущей главе мы говорили о планировщике — о том, как Go управляет временем. Теперь поговорим о том, как Go управляет пространством. Память — это ресурс, который выделяется и освобождается миллионы раз в секунду, и то, насколько эффективно это происходит, напрямую определяет производительность твоего сервиса.

Разработчик, не понимающий модель памяти Go, похож на водителя, который не знает, что такое обороты двигателя. Ехать может, но когда машина начинает дёргаться, он не понимает почему и не знает, что делать.

Три области, три судьбы

Когда твоя программа компилируется и запускается, память для данных распределяется по трём принципиально разным зонам.

Первая зона — статическая память. Здесь живут глобальные переменные, константы, строковые литералы. Они выделяются один раз при старте программы и существуют до

её завершения. Сборщик мусора сюда не заглядывает, освобождать нечего. Это самая простая и самая скучная зона, но о ней полезно помнить, потому что иногда строку выгодно объявить как константу, а не создавать её в цикле миллион раз.

Вторая зона — стек. У каждой горутины есть свой стек, и начинается он с двух килобайт. Когда функция вызывает другую функцию, её локальные переменные размещаются на стеке. Когда функция возвращает управление, её фрейм стека просто отбрасывается — никакого сборщика мусора, никаких накладных расходов. Процессор обожает стек, потому что доступ к нему линеен и предсказуем. Выделение памяти на стеке — это буквально одна инструкция процессора, которая сдвигает указатель стека.

Третья зона — куча. Это общая память для всех горутин, управляемая сборщиком мусора. Выделение в куче стоит дороже, потому что нужно найти свободный блок подходящего размера. Но главная цена — не выделение, а освобождение. Сборщик мусора должен периодически останавливать мир, обходить все живые объекты и помечать неиспользуемые как свободные.

Кто решает, где жить переменной

В таких языках, как C или C++, программист сам указы-

вает, где выделять память: объявил локальную переменную — она на стеке, вызвал malloc или new — в куче. В Go это решение принимает компилятор. И это одно из самых гениальных и одновременно сбивающих с толку новичков решений в языке.

Ты можешь написать код, который выглядит так, будто переменная должна быть на стеке, а компилятор решит отправить её в кучу. И наоборот: ты можешь взять указатель на переменную и вернуть его из функции, и компилятор скажет: «Ок, раз ты возвращаешь указатель, переменная должна пережить функцию, кладу её в кучу».

Конец ознакомительного фрагмента.