Graceful Shutdown в Go

мая 19, 2025 · 12 минут · German Gorelkin

Перевод/заметки Graceful Shutdown in Go: Practical Patterns

Для корректного завершения работы программы необходимо выполнить три ключевых действия:

1. **Закрыть точку входа**: Прекратить обработку новых запросов или сообщений из различных источников, таких как HTTP сервер, системы pub/sub и других. Однако исходящие соединения с внешними сервисами, включая базы данных и кэши, должны оставаться открытыми.
2. **Дождаться завершения всех активных запросов**: Необходимо дождаться завершения всех выполняемых запросов. Если какой-либо запрос

выполняется слишком долго, следует ответить на него специальной ошибкой.

3. **Освободить критически важные ресурсы**: Критические ресурсы, такие как соединения с базой данных, файловые дескрипторы и сетевые подключения, должны быть освобождены. Завершите все необходимые процессы и проведите окончательную очистку.

Перехват сигнала ОС

Прежде чем мы сможем реализовать *graceful shutdown*, нам необходимо перехватывать сигналы завершения. Эти сигналы сообщают нашему приложению о том, что пришло время освободить ресурсы и завершить работу.

> *Что же такое сигналы?*

В Unix-подобных системах сигналы представляют собой программные прерывания, которые уведомляют процесс о произошедших событиях и требуют выполнения определенных действий. Когда сигнал посылается, операционная система прерывает нормальный ход процесса, чтобы доставить это уведомление.

Рассмотрим несколько возможных сценариев:

- **Обработчик сигнала**: Процесс может зарегистрировать функцию-обработчик для определенного сигнала. Эта функция будет вызываться при получении соответствующего сигнала.
- **Действие по умолчанию**: Если обработчик не установлен, процесс будет следовать поведению по умолчанию для данного сигнала. Это может включать завершение, остановку, продолжение или игнорирование процесса.
- **Неблокируемые сигналы**: Некоторые сигналы, такие как `SIGKILL` (сигнал номер 9), нельзя перехватить или проигнорировать. Они могут привести к завершению процесса.

Когда ваше приложение Go запускается, среда выполнения автоматически регистрирует обработчики для различных сигналов, включая `SIGTERM`, `SIGQUIT`, `SIGKILL`, `SIGTRAP` и другие. Однако для корректного завершения работы обычно важны лишь три из них:

1. `SIGTERM` (Termination): Стандартный способ попросить процесс завершить свою работу. Этот сигнал не принуждает процесс к остановке. *Kubernetes* отправляет этот сигнал, когда хочет, чтобы ваше приложение завершилось до того, как будет принудительно остановлено.
2. `SIGINT` (Interrupt): Посылается, когда пользователь хочет остановить процесс из терминала, обычно нажатием `Ctrl+C`.
3. `SIGHUP` (Hang up): Изначально использовался при отключении терминала. Сейчас часто применяется для уведомления приложения о необходимости перезагрузки его конфигурации.

По умолчанию, когда ваше приложение получает сообщение `SIGTERM`, `SIGINT` или `SIGHUP`, среда выполнения Go завершает работу приложения.

> *Как Go завершает ваше приложение?*

Когда ваше приложение получает сигнал `SIGTERM`, среда выполнения сначала перехватывает его с помощью встроенного обработчика. Затем она проверяет, зарегистрирован ли пользовательский обработчик. Если нет, она временно отключает свой собственный обработчик и снова посылает приложению тот же сигнал (`SIGTERM`). На этот раз ОС обрабатывает его, используя поведение по умолчанию, то есть завершает процесс.

Вы можете переопределить это поведение, зарегистрировав свой собственный обработчик сигналов с помощью пакета `os/signal`.

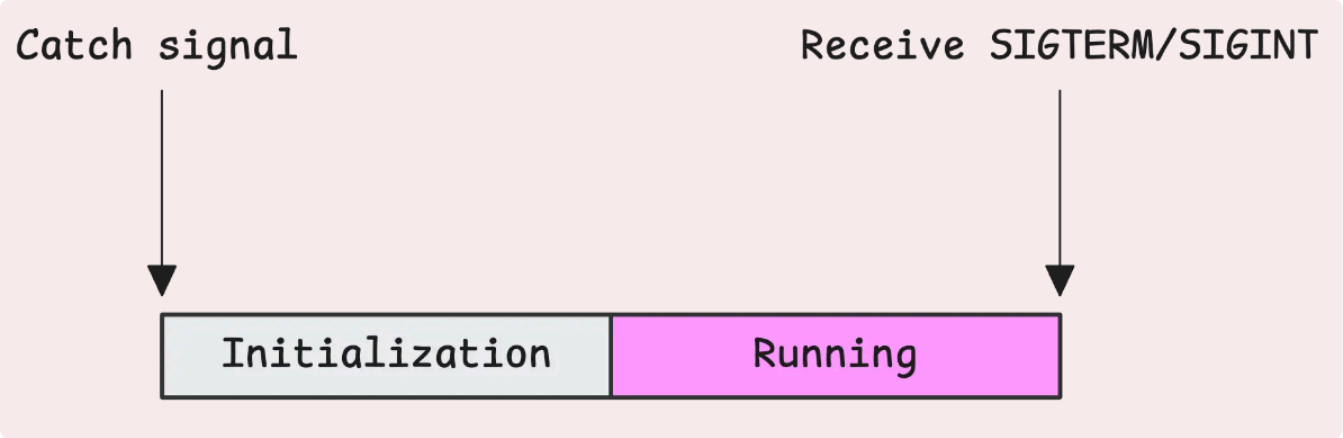
```
func main() {
  signalChan := make(chan os.Signal, 1)
  signal.Notify(signalChan, syscall.SIGINT, syscall.SIGTERM)

  // Setup work here

  <-signalChan

  fmt.Println("Received termination signal, shutting down...")
```

```
}
```



`signal.Notify` — это функция, которая сообщает среде выполнения Go о необходимости отправлять указанные сигналы в канал вместо того, чтобы следовать поведению по умолчанию. Благодаря этому появляется возможность обрабатывать эти сигналы вручную, что предотвращает автоматическое завершение работы приложения.

Буферизованный канал с ёмкостью 1 является отличным решением для надёжной обработки сигналов. В Go сигналы отправляются в этот канал через оператор `select` .

```
select {
case c <- sig:
default:
}
```

Это отличается от обычного `select` , который используется для чтения из канала. Когда мы используем его для отправки:

- Если в буфере есть свободное место, сигнал отправляется, и код продолжает работать.
- Если буфер уже заполнен, сигнал отбрасывается, и запускается `default` . В случае с небуферизованным каналом, если ни одна из гортин не ожидает получения сигнала, он будет пропущен.

Хотя он может удерживать только один сигнал, этот буферизованный канал обеспечивает надёжность, позволяя не пропустить первый сигнал, пока ваше приложение еще инициализируется и не прослушивает его.

Когда вы нажимаете `Ctrl+C` несколько раз, это не приводит к автоматическому завершению работы приложения. Первое нажатие клавиши `Ctrl+C` отправляет сигнал `SIGINT`. Повторное нажатие обычно отправляет ещё один `SIGINT`, а не `SIGKILL`. Большинство терминалов, таких как `bash` или другие оболочки Linux, не повышают уровень сигнала автоматически. Если вы хотите принудительно остановить процесс, вам необходимо отправить `SIGKILL` вручную, используя команду `kill -9`.

Это не всегда удобно для локальной разработки, где может потребоваться второе нажатие `Ctrl+C` для принудительного завершения работы приложения. Чтобы избежать этой проблемы, вы можете запретить приложению прослушивать последующие сигналы, используя `signal.Stop` сразу после получения первого сигнала:

```
func main() {
  signalChan := make(chan os.Signal, 1)
  signal.Notify(signalChan, syscall.SIGINT)

  <-signalChan

  signal.Stop(signalChan)
  select {}
}
```

Начиная с Go 1.16, обработка сигналов стала проще благодаря функции `signal.NotifyContext`, которая связывает обработку сигналов с отменой контекста.

```
ctx, stop := signal.NotifyContext(context.Background(), syscall.SIGINT, syscal
defer stop()

// Setup tasks here

<-ctx.Done()
stop()
```

Вам все равно следует вызвать `stop()` после `ctx.Done()`, чтобы повторное нажатие `Ctrl+C` могло принудительно завершить работу приложения.

Оповещение о тайм-ауте

Важно понимать, как быстро ваше приложение должно завершить свою работу после получения сигнала. Например, в Kubernetes по умолчанию установлен период ожидания 30 секунд, если не указано иное с помощью поля `terminationGracePeriodSeconds`. По истечении этого времени Kubernetes отправляет сигнал `SIGKILL`, который принудительно останавливает приложение. Этот сигнал нельзя перехватить или обработать.

Ваша логика завершения работы должна уложиться в этот период. Она должна включать обработку оставшихся запросов и освобождение ресурсов.

Предположим, что стандартное время ожидания составляет 30 секунд. Рекомендуется зарезервировать около 20% этого времени в качестве запасного, чтобы избежать завершения работы до завершения всех необходимых действий. Это означает, что вы должны стремиться завершить все операции в течение 25 секунд, чтобы минимизировать потери данных или их несогласованность.

Прекратить прием новых запросов

При работе с `net/http` вы можете обеспечить плавное завершение работы вашего сервера, используя метод `http.Server.Shutdown`. Этот метод останавливает прием новых соединений сервером и ожидает завершения всех активных запросов, прежде чем закрыть все idle соединения.

Вот как это работает:

- Если на текущем соединении уже выполняется запрос, сервер дает ему завершиться. После этого соединение помечается как idle и закрывается.
- Если клиент попытается установить новое соединение во время отключения, он не сможет этого сделать, так как приемники сервера уже закрыты. Обычно это приводит к ошибке «connection refused».

В контейнерных системах, как и во многих других средах с внешними балансировщиками нагрузки, существует задержка в прекращении обработки

новых запросов. Даже после того, как pod помечается как завершенный, он может продолжать получать трафик в течение нескольких секунд.

Внутренние компоненты Kubernetes, такие как `kube-proxy`, быстро обнаруживают изменение статуса pod на «Terminating». Они устанавливают приоритет маршрутизации **внутреннего трафика** на конечные точки, помеченные как `Ready,Serving`, вместо тех, которые имеют статус `Terminating,Serving`.

Однако внешний балансировщик нагрузки функционирует независимо от Kubernetes. Он обычно применяет свои собственные механизмы проверки работоспособности, чтобы определить, какие внутренние узлы должны получать трафик. Эта проверка состояния проверяет наличие исправных (`Ready`) и не завершающих свою работу pods на узле. Однако для распространения результатов этой проверки требуется некоторое время.

Есть два варианта решения этой проблемы:

1. Используйте хук `preStop`, чтобы приостановить работу на некоторое время. Это позволит внешнему балансировщику нагрузки корректно завершить работу pod.

   ```
   lifecycle:
   preStop:
       exec:
       command: ["/bin/sh", "-c", "sleep 10"]
   ```

   И что особенно важно, время, затраченное на выполнение хука `preStop`, учитывается в значении `terminationGracePeriodSeconds`.
2. Провалить проверку готовности(readiness probe) и спать на уровне кода. Этот подход применим не только к Kubernetes, но и к другим средам с балансировщиками нагрузки, которым необходимо знать, что pod не готов.

> *Что такое readiness probe?*

Проверка готовности определяет, готов ли контейнер к приёму трафика. Она периодически проверяет его состояние с помощью настроенных методов, таких как HTTP-запросы, TCP-соединения или выполнение команд. Если проверка не проходит успешно, Kubernetes удаляет контейнер из конечных

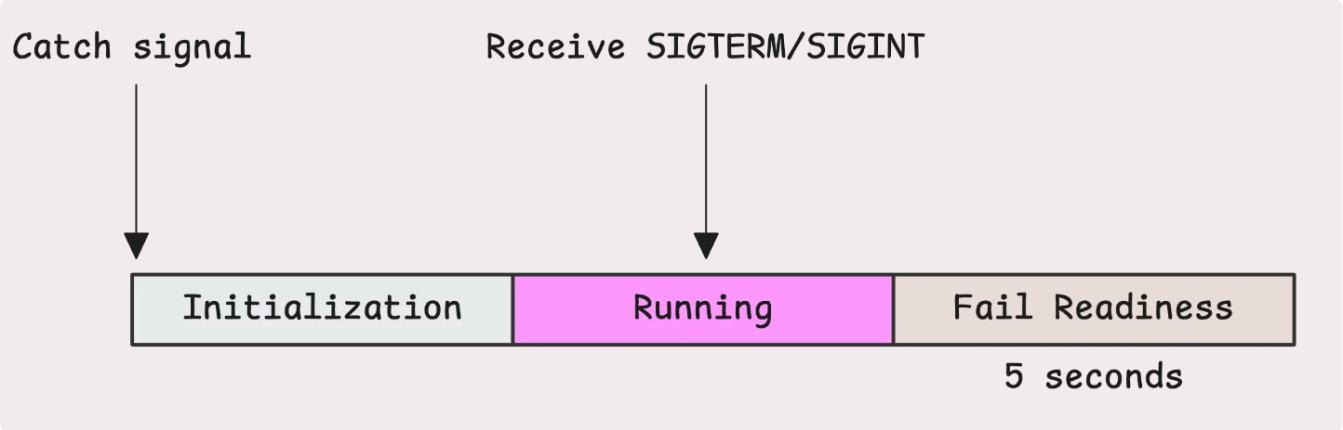
точек сервиса и не позволяет ему принимать трафик до тех пор, пока он снова не станет доступен.

Чтобы избежать ошибок подключения в течение этого короткого периода, рекомендуется сначала отменить проверку готовности. Это даст указание оркестратору прекратить отправку трафика на ваш pod.

```
var isShuttingDown atomic.Bool

func readinessHandler(w http.ResponseWriter, r *http.Request) {
    if isShuttingDown.Load() {
        w.WriteHeader(http.StatusServiceUnavailable)
        w.Write([]byte("shutting down"))
        return
    }

    w.WriteHeader(http.StatusOK)
    w.Write([]byte("ok"))
}
```



Этот паттерн также применяется в тестовых образах. В рамках его реализации закрытый канал служит для передачи сигнала о необходимости ответа с кодом 503, когда приложение завершает свою работу.

После того как система получает уведомление о том, что pod больше не готов к использованию, следует подождать несколько секунд, чтобы изменения успели распространиться.

Точное время ожидания зависит от конфигурации readiness probe; в этой статье мы будем использовать 5 секунд в следующей простой конфигурации:

```
readinessProbe:
```

```
    httpGet:
      path: /healthz
      port: 8080
    periodSeconds: 5
```

Обработка незавершенных запросов

Теперь, когда мы постепенно завершаем работу сервера, нам необходимо выбрать время ожидания в соответствии с вашим бюджетом на завершение:

```
ctx, cancelFn := context.WithTimeout(context.Background(), timeout)
err := server.Shutdown(ctx)
```

Функция server.Shutdown возвращается только в двух случаях:

1. Когда все активные соединения закрыты и все обработчики завершили свою работу.
2. Когда истекает время, переданное в Shutdown(ctx), до завершения работы обработчиков. В этом случае сервер прекращает ожидание.

В любом случае, Shutdown возвращается только после того, как сервер полностью прекращает обработку запросов. Именно поэтому ваши обработчики должны быть быстрыми и учитывать контекст. В противном случае они могут быть прерваны на середине процесса в случае 2, что может привести к различным проблемам, таким как частичная запись, потеря данных, несогласованное состояние, открытые транзакции или поврежденные данные.

Часто обработчики не знают, когда сервер завершает работу.

Итак, как мы можем сообщить нашим обработчикам, что сервер завершает свою работу? Ответ заключается в использовании контекста. Существует два основных способа реализации этой задачи:

Используйте контекстное middleware для реализации логики отмены

Это middleware оборачивает каждый запрос в контекст, который отслеживает завершение работы.

```
func WithGracefulShutdown(next http.Handler, cancelCh <-chan struct{}) http.Ha
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ctx, cancel := WithCancellation(r.Context(), cancelCh)
        defer cancel()

        r = r.WithContext(ctx)
        next.ServeHTTP(w, r)
    })
}
```

Используйте `BaseContext` для обеспечения единого контекста для всех соединений

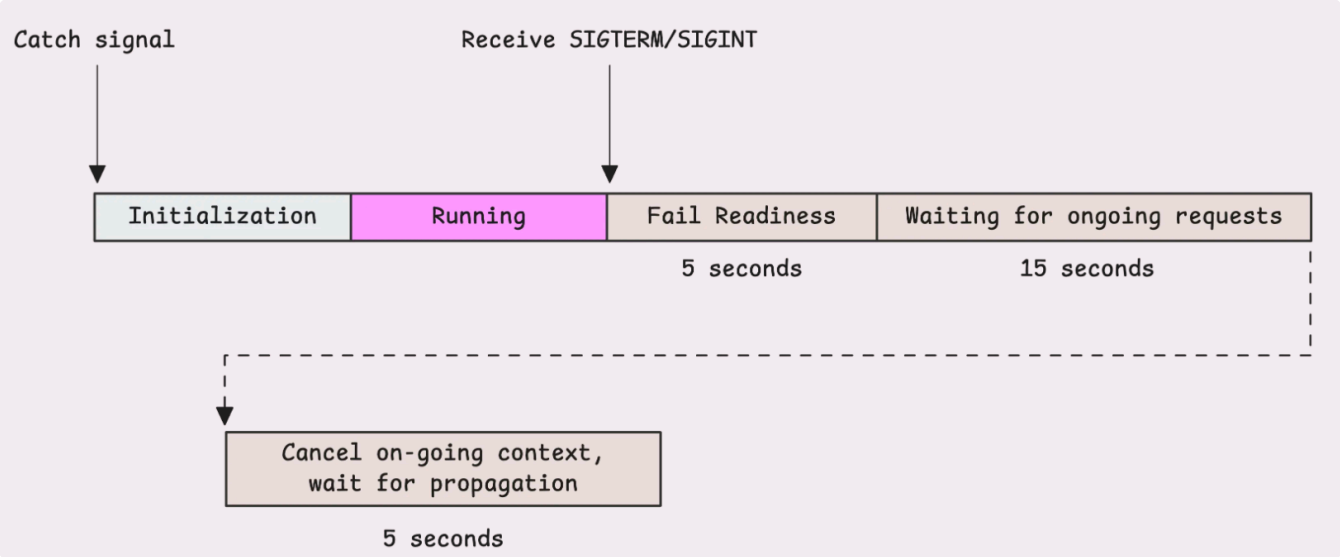
В этом примере мы создаем сервер, который использует пользовательский `BaseContext`. Этот контекст может быть отменен при завершении работы сервера, и он будет доступен для всех входящих запросов:

```
ongoingCtx, cancelFn := context.WithCancel(context.Background())
server := &http.Server{
    Addr: ":8080",
    Handler: yourHandler,
    BaseContext: func(l net.Listener) context.Context {
        return ongoingCtx
    },
}

// After attempting graceful shutdown:
cancelFn()
time.Sleep(5 * time.Second) // optional delay to allow context propagation
```

В HTTP-сервере есть два типа контекстов: `BaseContext` и `ConnContext`. Для плавного завершения работы лучше использовать `BaseContext`. С его помощью вы можете создать глобальный отменяемый контекст, который

будет применяться ко всему серверу. Когда вы отменяете этот контекст, всем активным запросам будет отправлено уведомление о том, что сервер завершает свою работу.



Вся эта работа над плавным завершением работы не поможет, если ваши функции не будут учитывать отмену контекста. Старайтесь избегать использования `context.Background()`, `time.Sleep()` или любой другой функции, которая игнорирует контекст.

Например, `time.Sleep(duration)` можно заменить на версию, учитывающую контекст:

```
func Sleep(ctx context.Context, duration time.Duration) error {
    select {
    case <-time.After(duration):
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}
```

В старых версиях Go функция `time.After` могла приводить к утечке памяти до момента срабатывания таймера. Однако в Go 1.23 и более новых версиях эта проблема была устранена. Если вы не уверены в версии Go, рекомендуется использовать `time.NewTimer` вместе с функцией `Stop`. В случае, если `Stop` возвращает `false`, дополнительно проверьте состояние таймера с помощью `<-t.C`.

time: stop requiring Timer/Ticker.Stop for prompt GC

Хотя эта статья посвящена HTTP-серверам, та же концепция применима и к другим сторонним сервисам. Например, в пакете `database/sql` есть метод `DB.Close`, который закрывает соединение с базой данных и предотвращает запуск новых запросов. Перед окончательным завершением он ожидает завершения всех текущих запросов.

Главный принцип *graceful shutdown* одинаков для всех систем: **необходимо прекратить прием новых запросов или сообщений и дать возможность текущим операциям завершиться в течение определенного периода времени.**

У некоторых может возникнуть вопрос о методе `server.Close()`, который немедленно закрывает текущие соединения, не дожидаясь завершения запросов. Можно ли его использовать после того, как `server.Shutdown()` вернет ошибку?

Короткий ответ — да, но это зависит от вашей стратегии завершения работы. Метод `Close` принудительно закрывает все активные слушатели и соединения:

- Обработчики, активно использующие сеть, получат ошибки при попытке чтения или записи.
- Клиент немедленно получит ошибку соединения, например, `ECONNRESET` («socket hang up»).
- Однако долго работающие обработчики, которые не взаимодействуют с сетью, могут *продолжать работать в фоновом режиме*.

Поэтому использование контекста для передачи сигнала о завершении работы по-прежнему остается более надежным и изящным подходом.

Освобождение критически важных ресурсов

Распространенной ошибкой является освобождение критических ресурсов сразу после получения сигнала о завершении работы. Однако в этот момент

ваши обработчики и выполняющиеся запросы всё ещё могут использовать эти ресурсы. Поэтому рекомендуется отложить очистку до тех пор, пока не пройдет тайм-аут завершения работы или не будут выполнены все запросы.

В большинстве случаев достаточно просто позволить процессу завершиться. Операционная система автоматически освободит ресурсы. Например:

- Память, выделенная Go, освобождается автоматически при завершении процесса.
- Дескрипторы файлов закрываются ОС.
- Ресурсы на уровне ОС, такие как дескрипторы процессов, также освобождаются.

Однако есть важные случаи, когда явная очистка при завершении работы все же необходима:

- *Соединения с базой данных* должны быть закрыты должным образом. Если какие-либо транзакции все еще открыты, их необходимо зафиксировать или откатить. Без надлежащего закрытия база данных вынуждена полагаться на тайм-ауты соединений.
- *Очереди сообщений и брокеры* часто требуют аккуратного отключения. Это может включать в себя очистку сообщений, фиксацию смещений или подачу сигнала брокеру о выходе клиента. Без этого могут возникнуть проблемы с ребалансировкой или потерей сообщений.
- *Внешние службы* могут не сразу обнаружить разрыв соединения. Закрытие соединений вручную позволяет этим системам очиститься быстрее, чем ожидание таймаута TCP.

Хорошее правило — завершать работу компонентов в обратном порядке их инициализации. Это позволяет учесть зависимости между компонентами и избежать возможных ошибок.

Оператор `defer` в Go облегчает эту задачу, поскольку последняя функция выполняется первой:

```
db := connectDB()
defer db.Close()
```

```
cache := connectCache()
defer cache.Close()
```

Некоторые компоненты требуют особого подхода. Например, если вы кэшируете данные в памяти, вам, возможно, потребуется записать их на диск перед завершением работы. В таких случаях необходимо разработать специальную процедуру завершения работы для этого компонента, чтобы обеспечить корректную очистку.

Итоги

Это полный пример механизма *graceful shutdown*. Он написан в единой и простой структуре, чтобы его было легче понять. Вы можете адаптировать его под свое приложение, используя `errgroup`, `WaitGroup` или любые другие шаблоны:

```
const (
    _shutdownPeriod      = 15 * time.Second
    _shutdownHardPeriod  = 3 * time.Second
    _readinessDrainDelay = 5 * time.Second
)

var isShuttingDown atomic.Bool

func main() {
    // Setup signal context
    rootCtx, stop := signal.NotifyContext(context.Background(), syscall.SIGINT
    defer stop()

    // Readiness endpoint
    http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
        if isShuttingDown.Load() {
            http.Error(w, "Shutting down", http.StatusServiceUnavailable)
            return
        }
        fmt.Fprintln(w, "OK")
    })

    // Sample business logic
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        select {
        case <-time.After(2 * time.Second):
            fmt.Fprintln(w, "Hello, world!")
```

```
        case <-r.Context().Done():
            http.Error(w, "Request cancelled.", http.StatusRequestTimeout)
        }
    })

    // Ensure in-flight requests aren't cancelled immediately on SIGTERM
    ongoingCtx, stopOngoingGracefully := context.WithCancel(context.Background
    server := &http.Server{
        Addr: ":8080",
        BaseContext: func(_ net.Listener) context.Context {
            return ongoingCtx
        },
    }

    go func() {
        log.Println("Server starting on :8080.")
        if err := server.ListenAndServe(); err != nil && err != http.ErrServer
            panic(err)
        }
    }()

    // Wait for signal
    <-rootCtx.Done()
    stop()
    isShuttingDown.Store(true)
    log.Println("Received shutdown signal, shutting down.")

    // Give time for readiness check to propagate
    time.Sleep(_readinessDrainDelay)
    log.Println("Readiness check propagated, now waiting for ongoing requests

    shutdownCtx, cancel := context.WithTimeout(context.Background(), _shutdown
    defer cancel()
    err := server.Shutdown(shutdownCtx)
    stopOngoingGracefully()
    if err != nil {
        log.Println("Failed to wait for ongoing requests to finish, waiting fo
        time.Sleep(_shutdownHardPeriod)
    }

    log.Println("Server shut down gracefully.")
}
```

Комментарии в Telegram-группе!

go

« ПРЕДЫДУЩАЯ
Первый принцип (First Principles)

СЛЕДУЮЩАЯ »
io.Reader и io.Writer в Go