Если вы видите что-то необычное, просто сообщите мне.

Go: Ввдение в сокеты межпроцессного взаимодействия(unixsocket)

Сокеты межпроцессного взаимодействия предлагают эффективное,безопасное двухсторонее подключение между процессами на Unix/Linux машинах. В то время как каналы отлично используются для подключения между горутинами приложения, и HTTP вездесущь, при подключении между Go приложениями(межпроцессороное взаимодействие) запущенные на той же машине каналы не помогают, а подключление к сокету межпроцессного взаимодействия гораздо проще, эффективнее, и более безопасно чем HTTP или другие интернет протоколы подключений.

Всё что вам нужно это только пакет `net` для запуска подключения:

- Пакет `net` предоставляет перносимый интерфейс для сети I/O, включая Unix сокеты.
- Так же пакет предоставляет доступ к низкоуровневым примитивам сети, большинству клиентов требуедсят только базовый интерфейс п редоставляемый `Dial`, `Listen` и `Accept` функциями и связанные `Conn` и `Listener` интферфейсы.

К сожалению эти фукнции и интерфейсы редко документированны(в частности сокеты межпроцессного взаимодействия), так же нет официального Go блога о том, как с чего начать при работе с сокетами. Похоже что недостаток хорошего ввдения на StackOverflow и Go блогах. Большинство статей о сокетах показывают C реализацию; где я сосредоточусь на том как начать Работать с Go.

Первое, сокерт представлен специальным фалом. Ваш сервер слушает через файл, принимае подключения и читает данные через это подключение. Ваш клиент использует файл, чтобы создать подключение и затем пишет данные в это самое подключение.

Вы возможно думаете, что вам нужно создать этот специальный файл используя пакет `os`. В нем вы можете найти постоянную `FileMode`, `ModeSocket`, которая может взывать к вам. Но это не поможет: Незадокументированный функционал функции `Listen` заключается в том, что (это должно) создать файл для вас, и он будет существовать с ошибкой вводящей в заблуждение: `“bind: address already in use”`, если файл уже существует. Отсюда дла начала, нелогичный шаг в создании сервера это удаление файла который вы собираетесь слушать, и только помто его можно слушать:

```
    os.Remove(cfg.SocketFile)
    listener, err := net.Listen("unix", cfg.SocketFile)
    if err != nil {
        log.Fatalf("Unable to listen on socket file %s: %s", cfg.SocketFile, err)
    }
    defer listener.Close()
```

Для сервера который управляет постоянно и обрабатывает множество похожих подключений, вы захотите использовать бесконечный цикл, в котором вы примете подключение в `listener` и начнете новую горутину для его обработки:

```
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Fatalf("Error on accept: %s", err)
        }
        go message.HandleConn(conn)
    }
```

Ваш обработчик должен создать буффер любого желаемого размера, и прочитать в него бесконечный цыкл который остановится когда `Read` выдаст ошибку. `Read` едва документирован в `net` пакете. Странным образом, вы должны знать что дальше нужно смотреть в документацию `io` пакета для `Reader` интерфейса:

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

... чтобы узнать важную информацию про поведение `Read` в других пакетах стандартной библиотеки:

- `Reader` - интерфейс который оборачивает базовый метод `Read`
- `Read` читает `len(p)` байтов в `p`. Он возвращяет количество прочитанных байтов(0<= n <= len(p)) в том числе любую ошибку. Даже если `Read` вернет n < len(p), он может использовать всё из `p` как испорченное пространство в время вызова. Если какие-то данные доступны, но длинной не равной `len(p)` байт, `Read` вернеть то что доступно, вместо того, чтобы чего-то ждать.
- Когда `Read` встречает ошибку или `EOF` условние после успешного чтения n > 0 байтов, он возвращает количество прочитанных байтов. Может вернуть `non-nil` ошибку из того же вызова или вернуть ошибку `n == 0` из подпоследовательности вызовов. Экземпляр данного конкретного случая это то что `Reader` возвращает `non-zero` число байтов в конце потока ввода, может так же вернуть или `err == EOF` или `err == nil`. Следующий `Read` должен вернуть `0`, `EOF`.
- Функции должны всегда обрабатывать `n>0` байтов возвращенные перед учитываением ошибки `err`. Выполнение этого условия обрабатывает `I/O` ошибки, которые случаются после чтения неких байтов и так же обоим позволено `EOF` поведение.
- Реализация Read обескураживает от возвращения `0` байтов с `nil` ошибкой, за исключением когда `len(p) == 0`. Функция должна отнестись к возврату `0` или `nil`, как к индикатору что ничег не случилось, в частности это не говорит об `EOF`(конце файла)
- Реализация не должна удерживать `p`:

Как при загадочном старте, буфер который вы передаете `Read` в форме среза байтов, который должен иметь длинну больше чем нуль, для того, чтобы в него что-то прочитать. Это совмещается с передачей среза больше чем указателя на срез, потому что любое увеличение длинны среза внутри `Read` не будет видно в вызове контекста без использования указателя. Относительно общий баг в использовании `Read` это буфер с нулевой длинной.

Другой распространненый баг это игнорирвание предостережения выше, обрабатывать возвращенный байты до обработки ошибок. Это контрастирует с советом общей обработки ошибок, в большинстве программных контекстов на Go, и очень важно исправить реализацию основанных на `net` поключений вообще.

Ниже пример обработчика который решает эти проблемы. Он читает в буфер с длинной больше чем ноль внутри бесконечного цикла, и прервыается только при ошибке. После каждого `Read`, первый счетчик байтов буфера поглащается перед обработкой ошибок:

```
func HandleConn(c net.Conn) {
    received := make([]byte, 0)
    for {
        buf := make([]byte, 512)
        count, err := c.Read(buf)
        received = append(received, buf[:count]...)
        if err != nil {
            ProcessMessage(received)
            if err != io.EOF {
                log.Errorf("Error on read: %s", err)
            }
            break
        }
    }
}
```

Этим методом, все данные отправленные подключением воспринимаются сервером как одно сообщение. Клиент должен закрыть подключение сигналом о конце сообщения. Чтение закрытого подключения вренуть ошибку `io.EOF`, которая не должна быть обработа как обыная ошибка. Это просто сигнал о том, что сообщение закончилось, часто подсказка к началу обработки сообщения так как оно закончено.

Что происходит в `ProcessMessage`, конечно, зависит от вашего приложения. Так как строка это по-факту срез байтов, только для чтения, это маленькая попытка для связи текстовых данных таким образом. Но байтовый срез так же распространненая валюта в стандартной библиотеке Go, и любые данные могут быть зашифрованы как срез байтов.

Всё что нам осталось - это сделать клиента. Клиент - это просто функция которая поднимает сокет файл для того чтобы создать подключение, откладывает закрытие

подключения, и пишет байтовое сообщение в подключение. Не нужно беспокоится о размере сообщения, оно может быть очень большое, но код не изменится. Ниже пример с логированием ошибок:

```
type Sender struct {
    Logger     *log.Logger
    SocketFile string
}

func (s *Sender) SendMessage(message []byte) {
    c, err := net.Dial("unix", s.SocketFile)
    if err != nil {
        s.Logger.Errorf("Failed to dial: %s", err)
    }
    defer c.Close()
    count, err := c.Write(message)
    if err != nil {
        s.Logger.Errorf("Write error: %s", err)
    }
    s.Logger.Infof("Wrote %d bytes", count)
}
```

Следующим шагом может быть добавление ответа с подтверждением полученного. Для множества приложений, выше приведенная инструкция это всё что нужно для начала связи между GO процессами использующими Unix сокеты.

Revision #5
Created 2021-10-22 13:27:02 UTC by gasick
Updated 2023-04-16 19:30:04 UTC by gasick