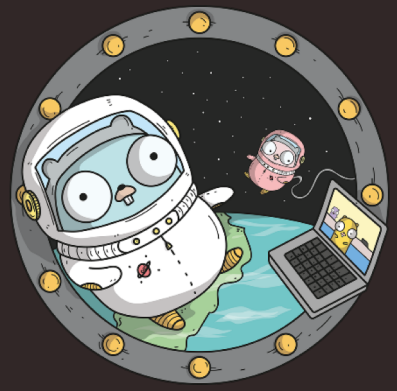
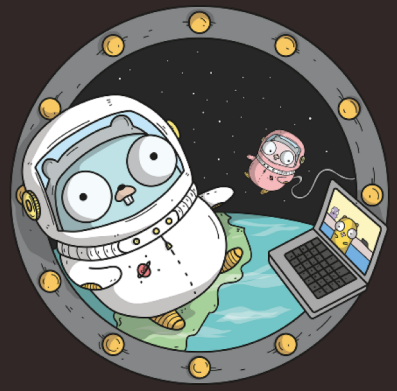
gRPC와 goroutine 톺아보기



한성민

GDG Golang Korea

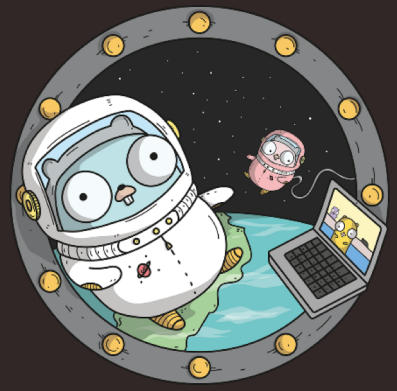
시작은 위대했으나



GDG Golang Korea

시작은 위대했으나

끝은 미약하리라!

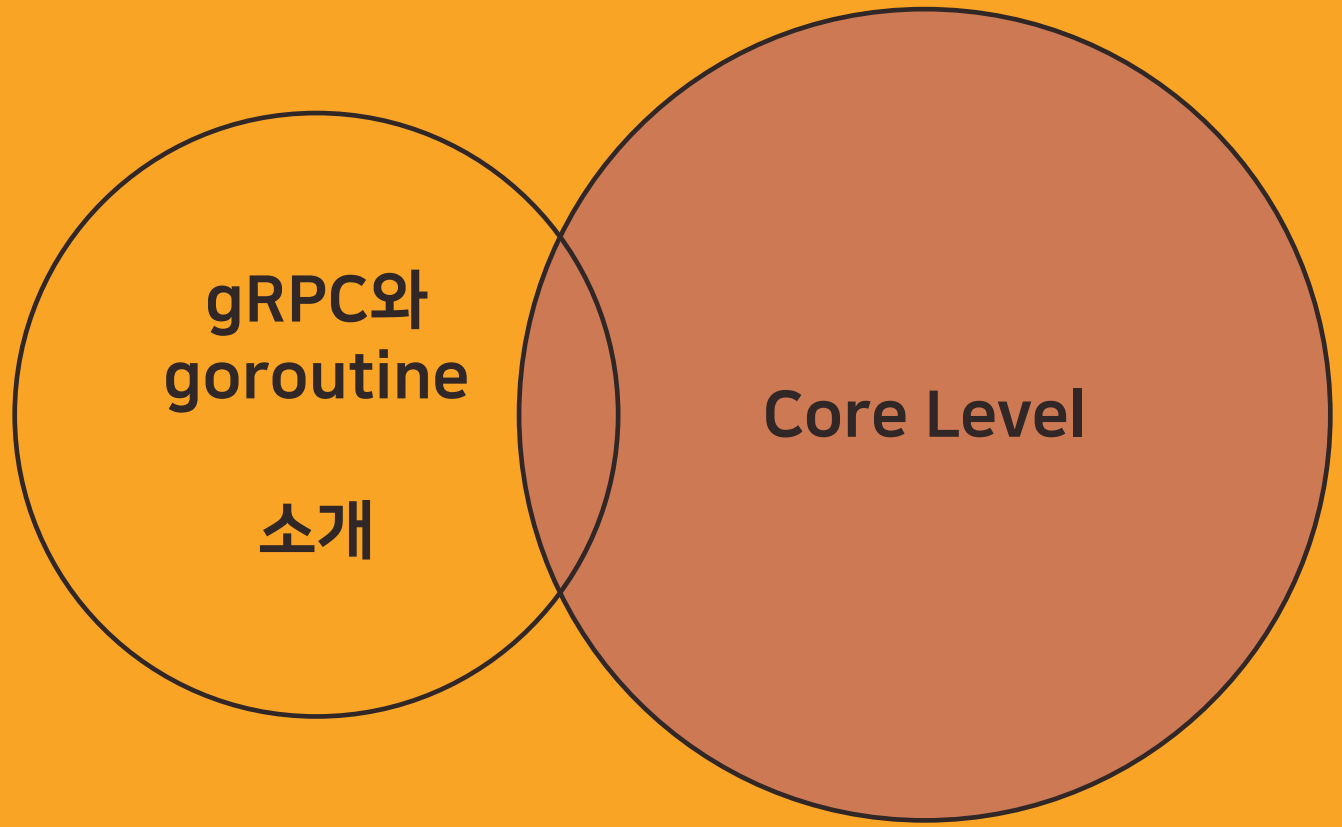


GDG Golang Korea

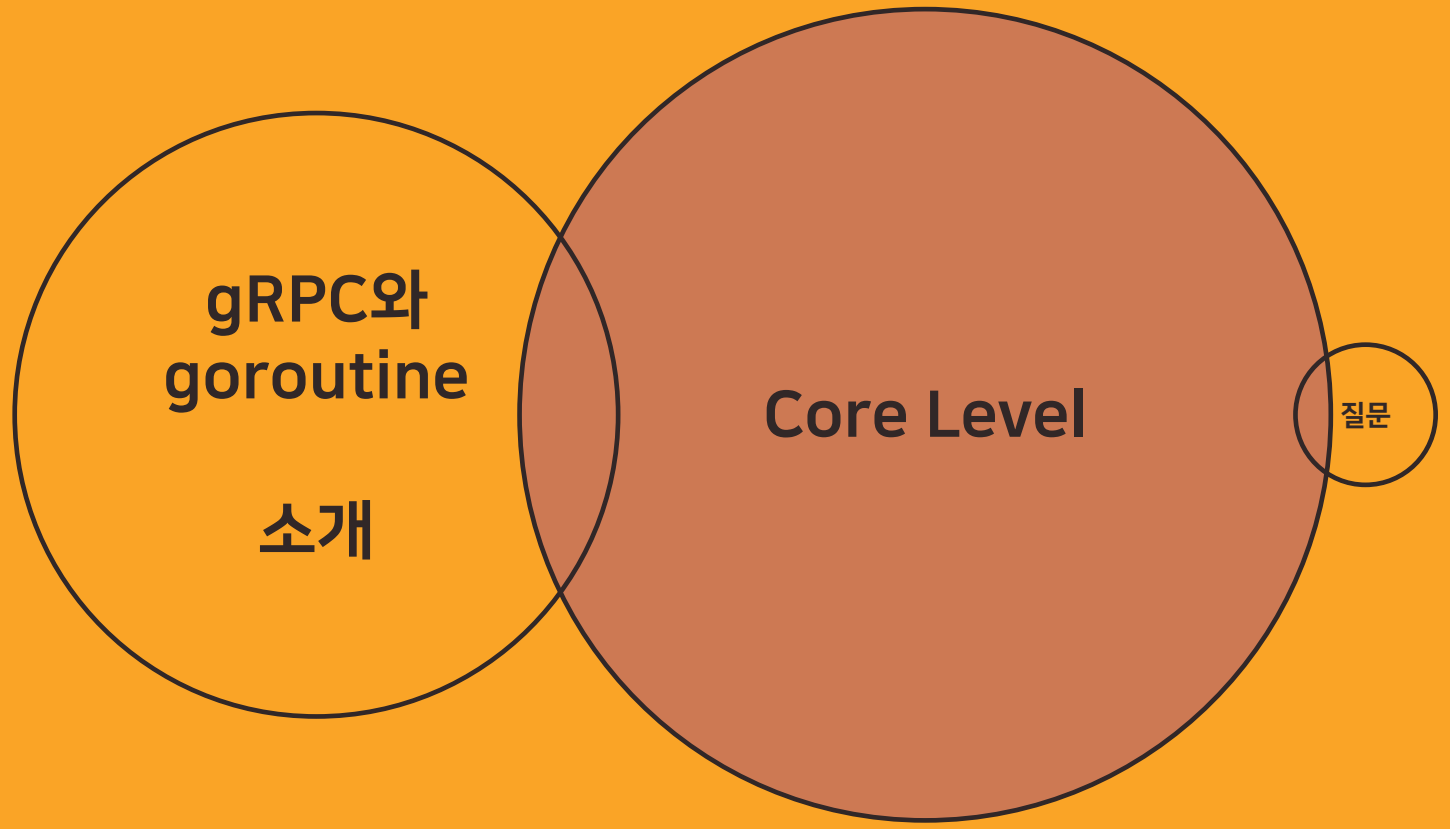
이 자리에서 같이 얘기하고 싶은 것은

이 자리에서 같이 얘기하고 싶은 것은

이 자리에서 같이 얘기하고 싶은 것은

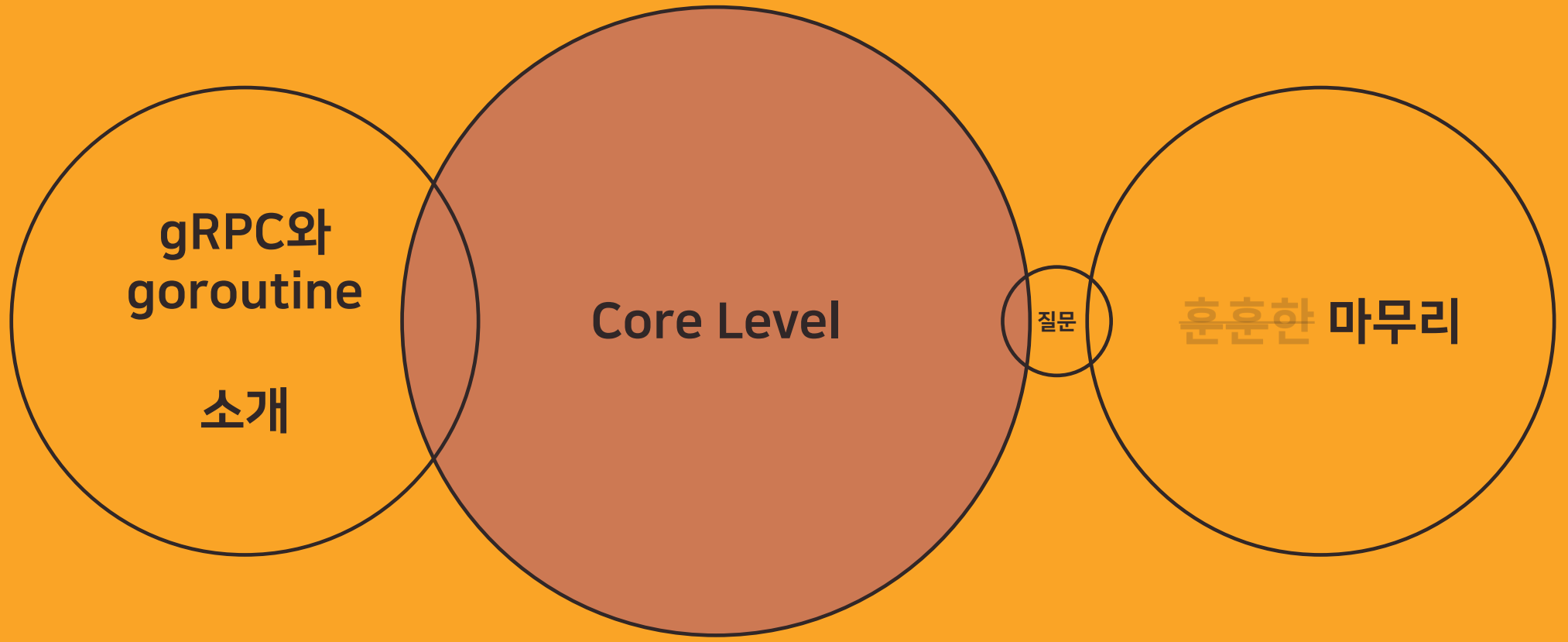


이 자리에서 같이 얘기하고 싶은 것은

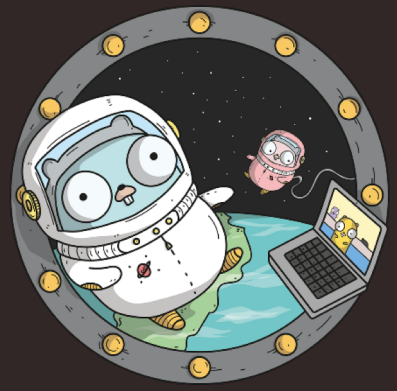


답변 안하는 질문 시간 뭐 그런거요
하하하

이 자리에서 같이 얘기하고 싶은 것은



소개

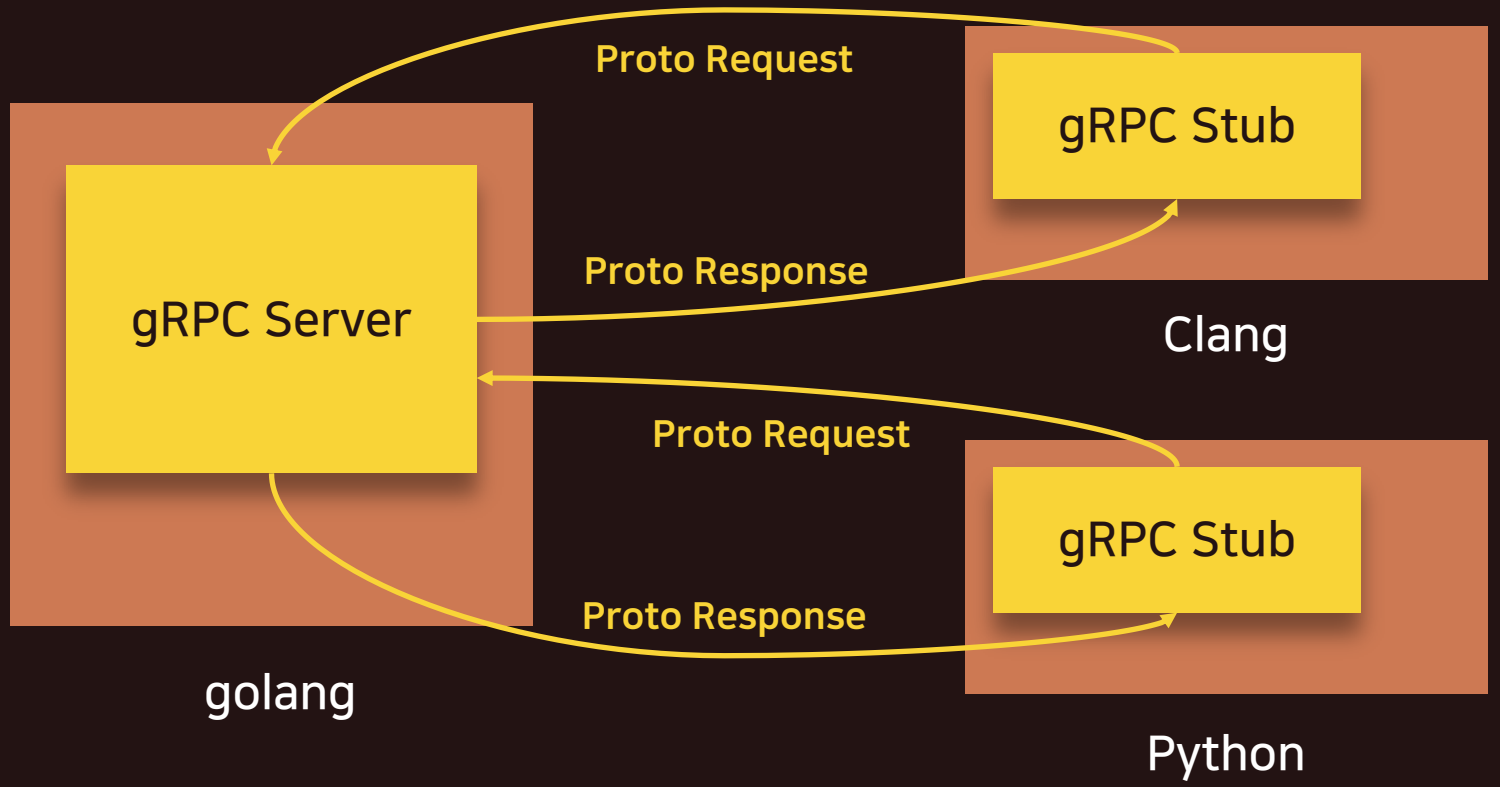


GDG Golang Korea

소개

gRPC

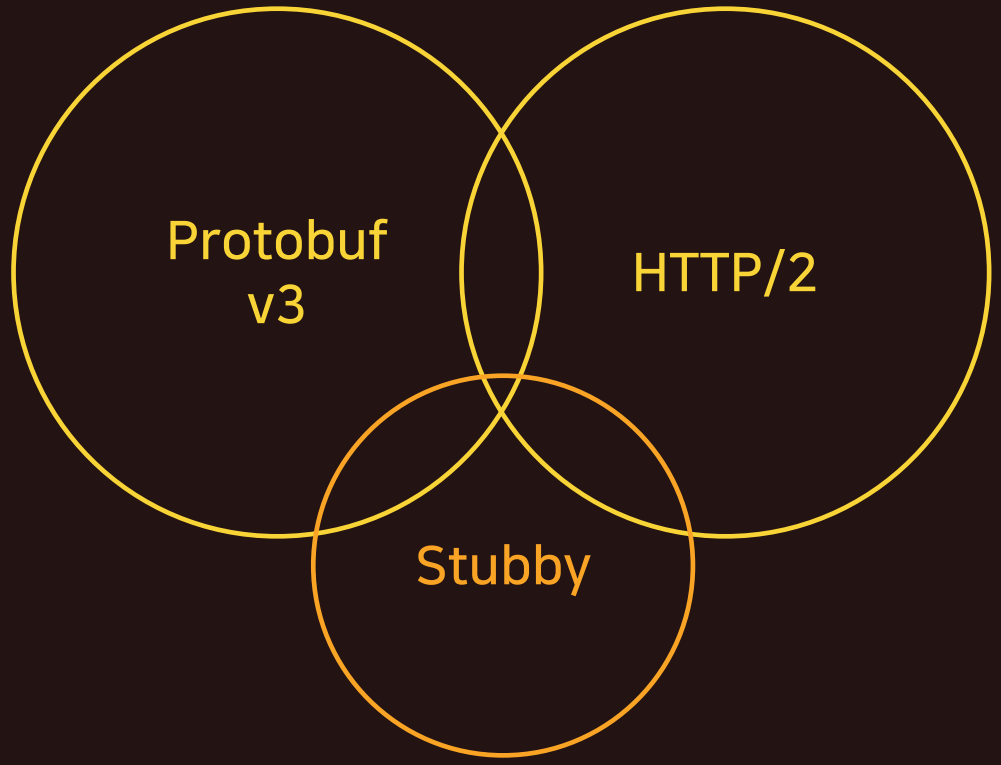
gRPC는 Protobuf 위에 올려진 구글사의 RPC 프로토콜로 굉장히 빠른 성능을 가짐.



소개

gRPC

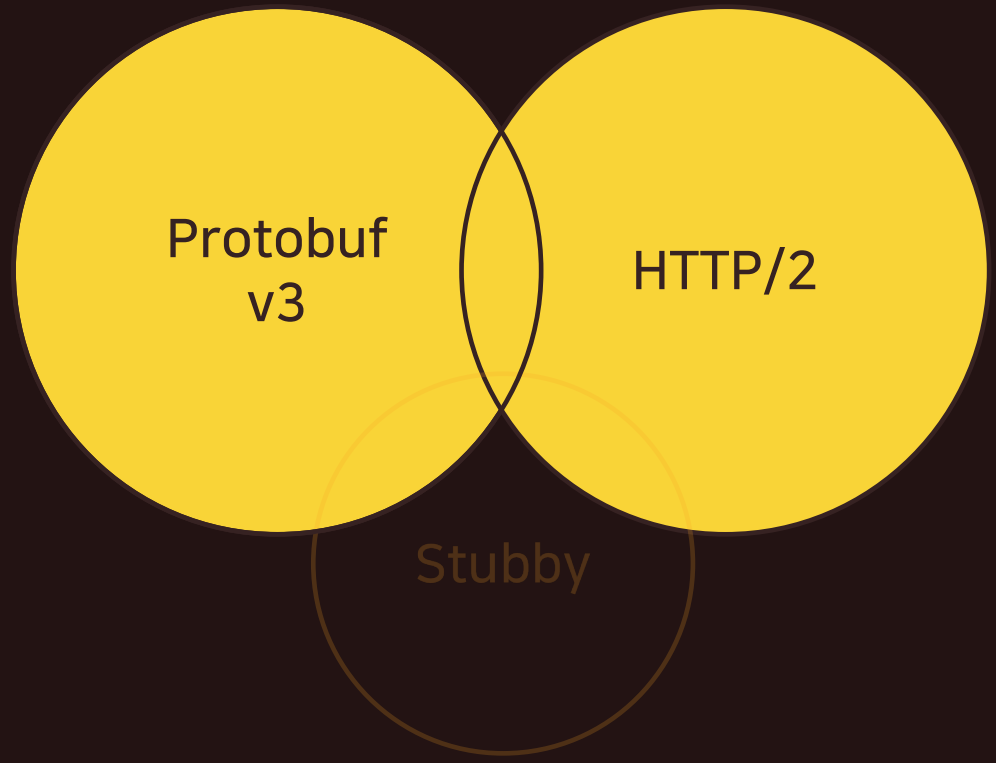
gRPC는 대표적으로 아래와 같이 구성 되어있습니다.



소개

gRPC

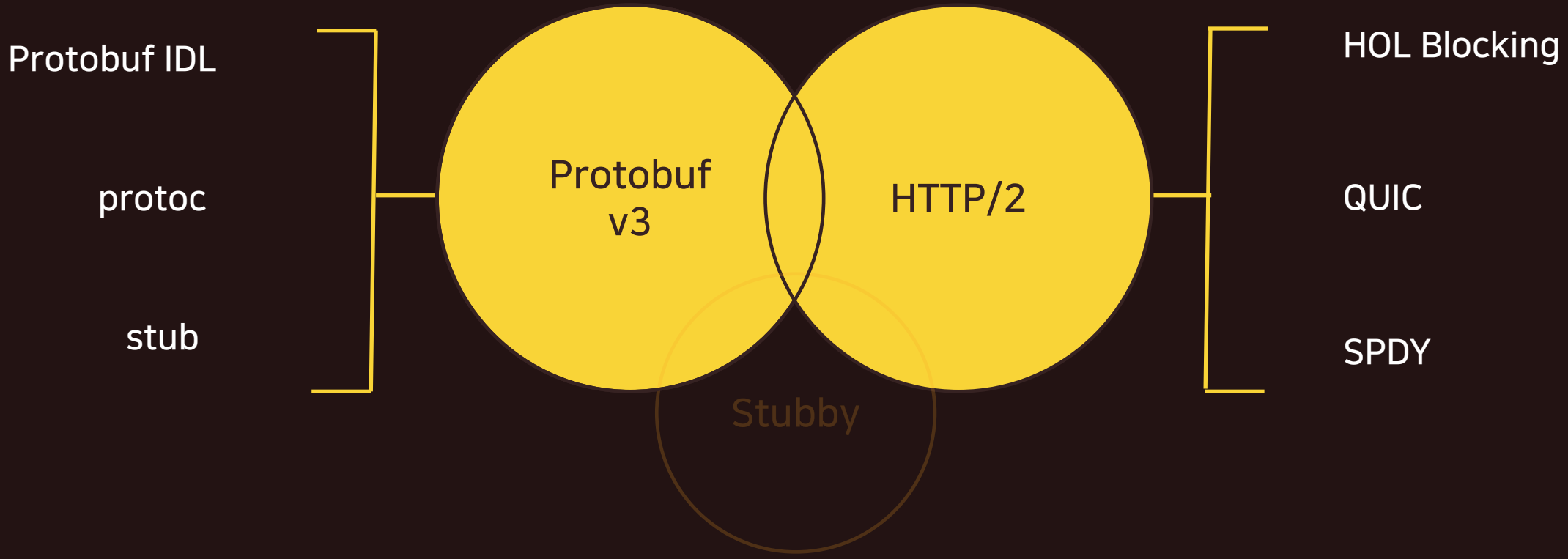
오늘 여기서는 두가지에 대해서 얘기하고자 합니다.



소개

gRPC

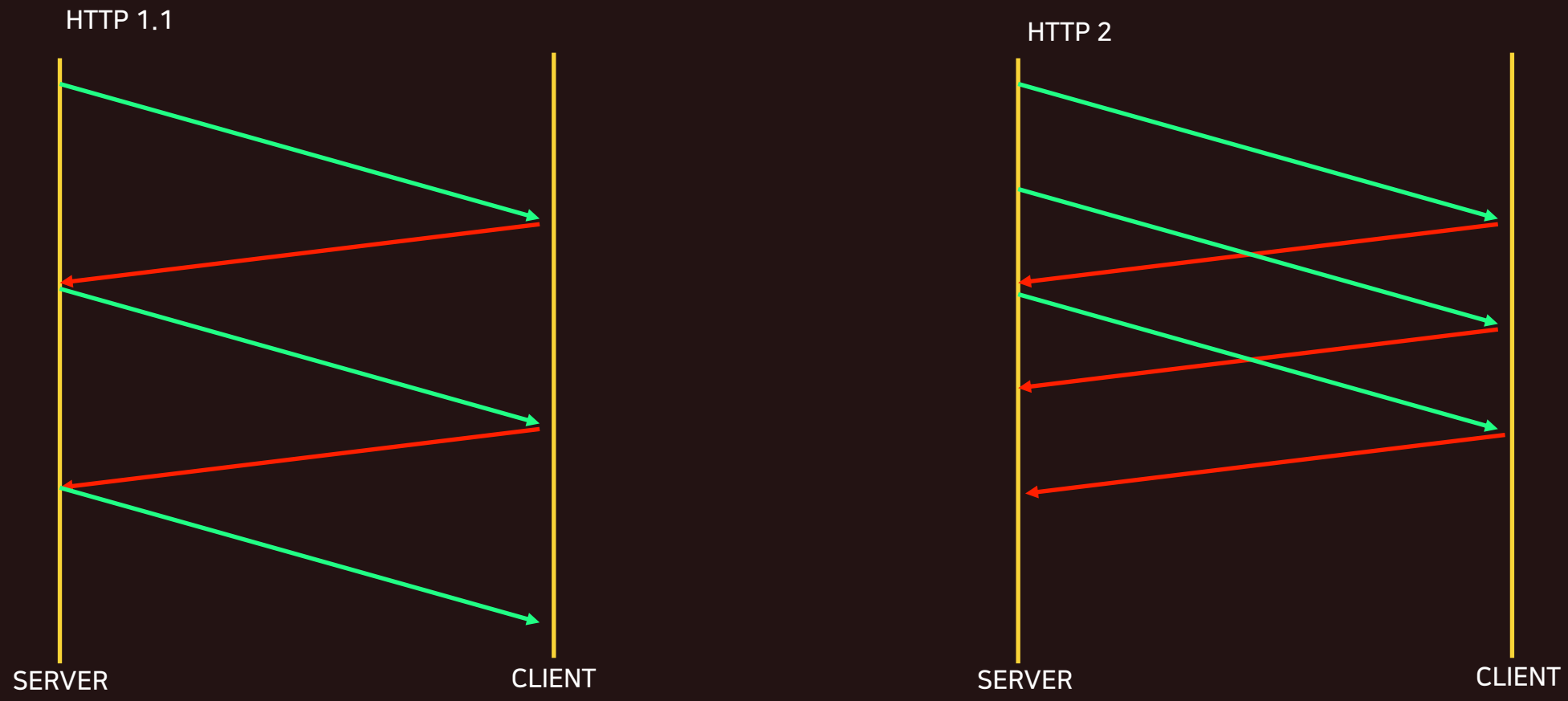
그리고 또 각각은..



소개

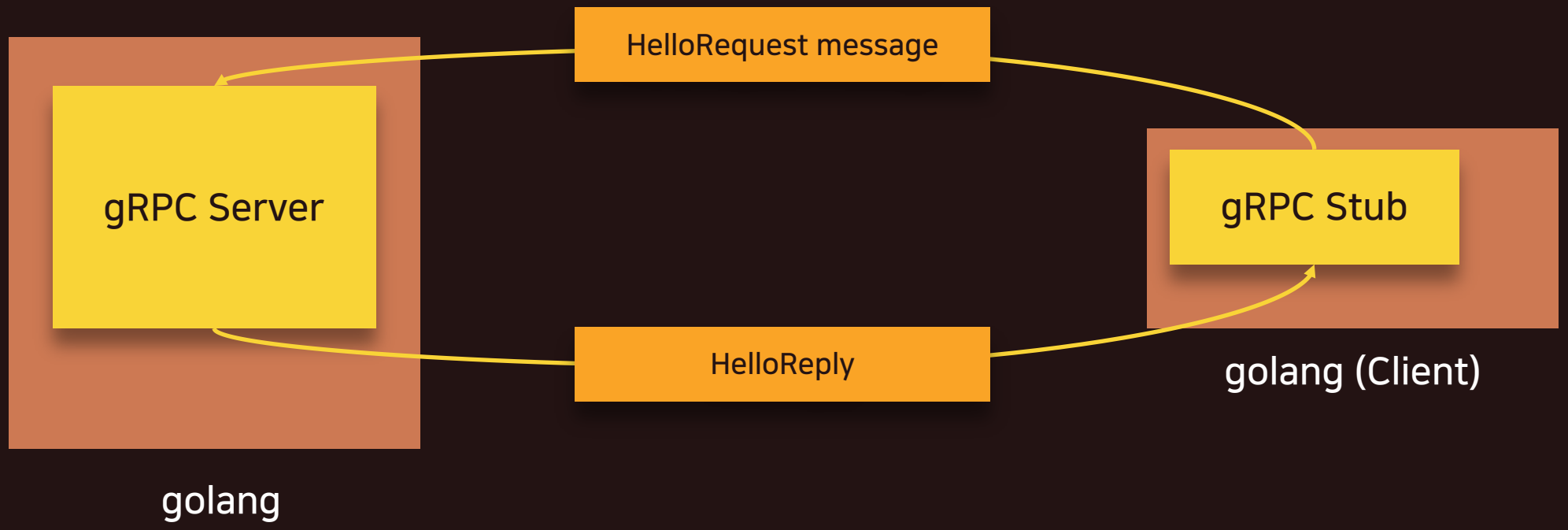
gRPC의 HTTP2 HOL Blocking

HOL(Head Of Line) Blocking은 HTTP 1.1에 비해 멀티 플렉싱을 지원합니다.



소개

gRPC 예시


HelloRequest message
gRPC Server
gRPC Stub
HelloReply
golang (Client)
golang

gRPC 예시

Protobuf 구성

```
syntax = "proto3";

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

gRPC 예시

Protobuf 구성

```
syntax = "proto3";

package helloworld;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Service

Protobuf 컴파일 과정에서,
어떤 서비스를 이용해 서로 통신을 할지 알려줍니다.
사용자가 지정한 서비스 이름은 이후 컴파일 된
go 코드에서 사용할 수 있습니다.

Message

서비스에서 사용하는 값의 형식을 지정하는
인터페이스입니다.

Protobuf는 통신에서 사용하는 값의 형식을 사전에 사용자가 지정 해주기 때문에
RESTful에서 사용하는 JSON에 대비해서 안전하고
XML의 DTD에 비교해서는 읽기 좋고 쉽습니다.

gRPC 예시

코드 생성과 생성된 go 코드

Bash

```bash
mkdir -p proto
protoc \
    -I examples \
    examples/example.proto \
    --go_out=plugins=grpc:proto
```

example.pb.go

```go
var _ = proto.Marshal
var _ = fmt.Errorf
var _ = math.Inf

// ...

func NewGreeterClient(cc *grpc.ClientConn) GreeterClient {
   return &greeterClient{cc}
}

func (c *greeterClient) SayHello(ctx context.Context, in *HelloRequest, opts ...grpc.CallOption) (*HelloReply, error) {
   out := new(HelloReply)
   err := c.cc.Invoke(ctx, "/helloworld.Greeter/SayHello", in, out, opts...)
   if err != nil {
      return nil, err
   }
   return out, nil
}

// ...

// GreeterServer is the server API for Greeter service.
type GreeterServer interface {
   // Sends a greeting
   SayHello(context.Context, *HelloRequest) (*HelloReply, error)
}
```

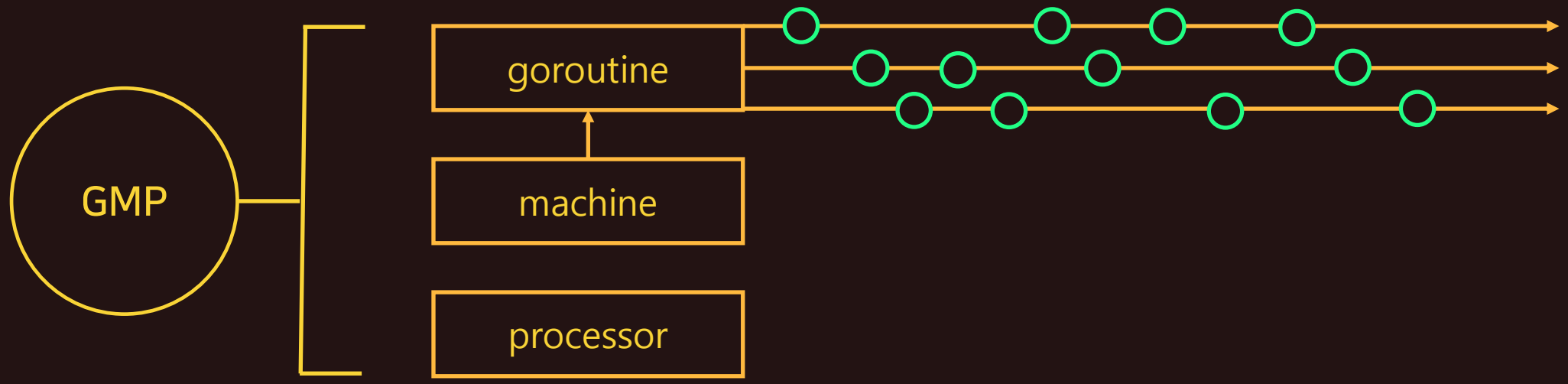
소개

goroutine

고루틴(goroutine)은 go언어의 런타임 계층에서 라이프 사이클을 관리하는 가상 쓰레드 개념.

여기서는 쉽게 미니 쓰레드라고 생각해봅시다.

사실 이런 형태는 LWP(lightweight process) 형태를 갖습니다.



소개

goroutine 예시

hello_goroutine.go

```go
package main

import (
"fmt"
"time"
)

func display(s string) {
   for i := 0; i < 10; i++ {
      fmt.Println(s, i)
   }
}

func main() {
   go display("hello")
   go display("world")
   time.Sleep(time.Second * 5)
}
```

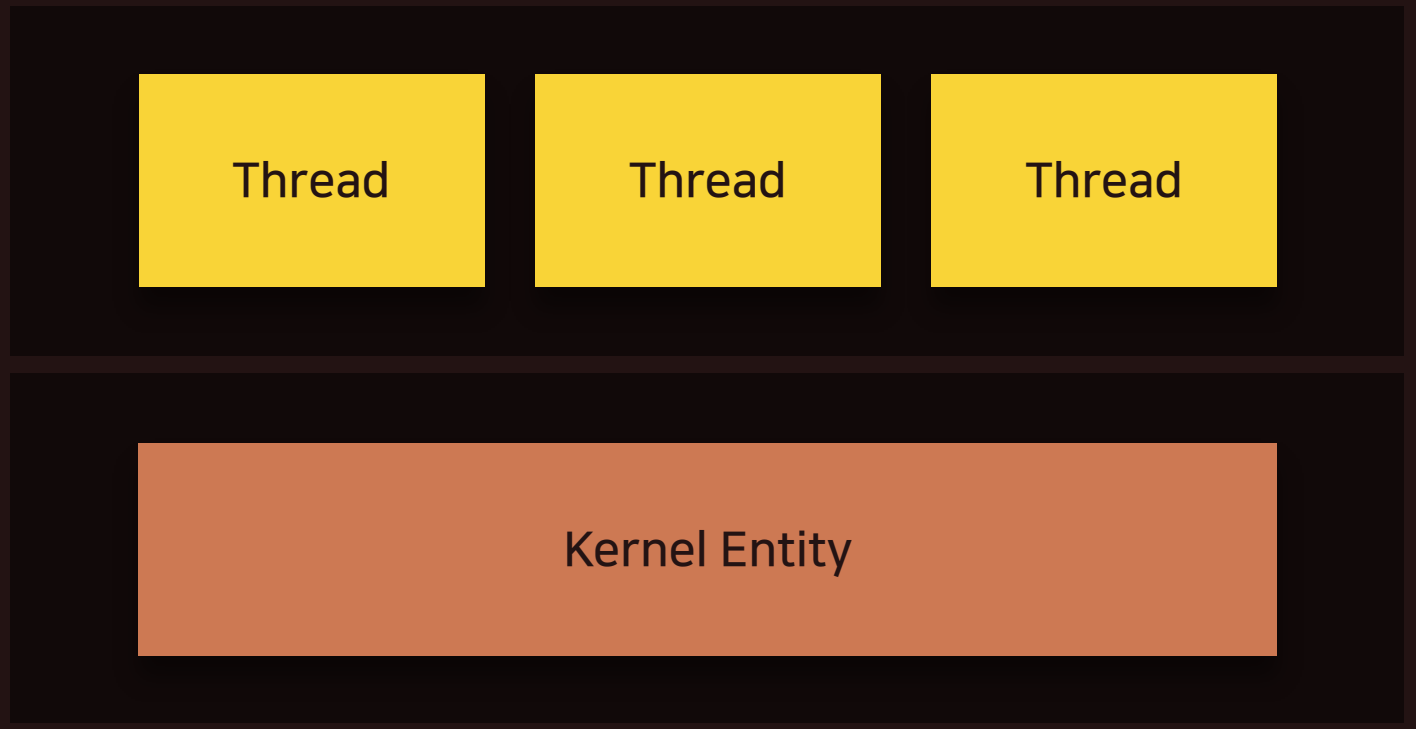
OUTPUT

```
hello 0
hello 1
world 0
world 1
world 2
world 3
world 4
hello 2
hello 3
hello 4
hello 5
hello 6
hello 7
hello 8
hello 9
world 5
world 6
world 7
world 8
world 9
```

소개

goroutine

일반적인 쓰레드 구성은 다음과 같은 특성을 가지고 있습니다.



User Space

유저 스페이스는 일반적으로 유저 모드 레벨에서의 특권에 따른 코드를 실행할 수 있으며, 하드웨어에 직접 접근이 불가능함.
유저 어플리케이션은 커널 레벨 함수를 실행 시 커널 스페이스로 넘어가게 됨.

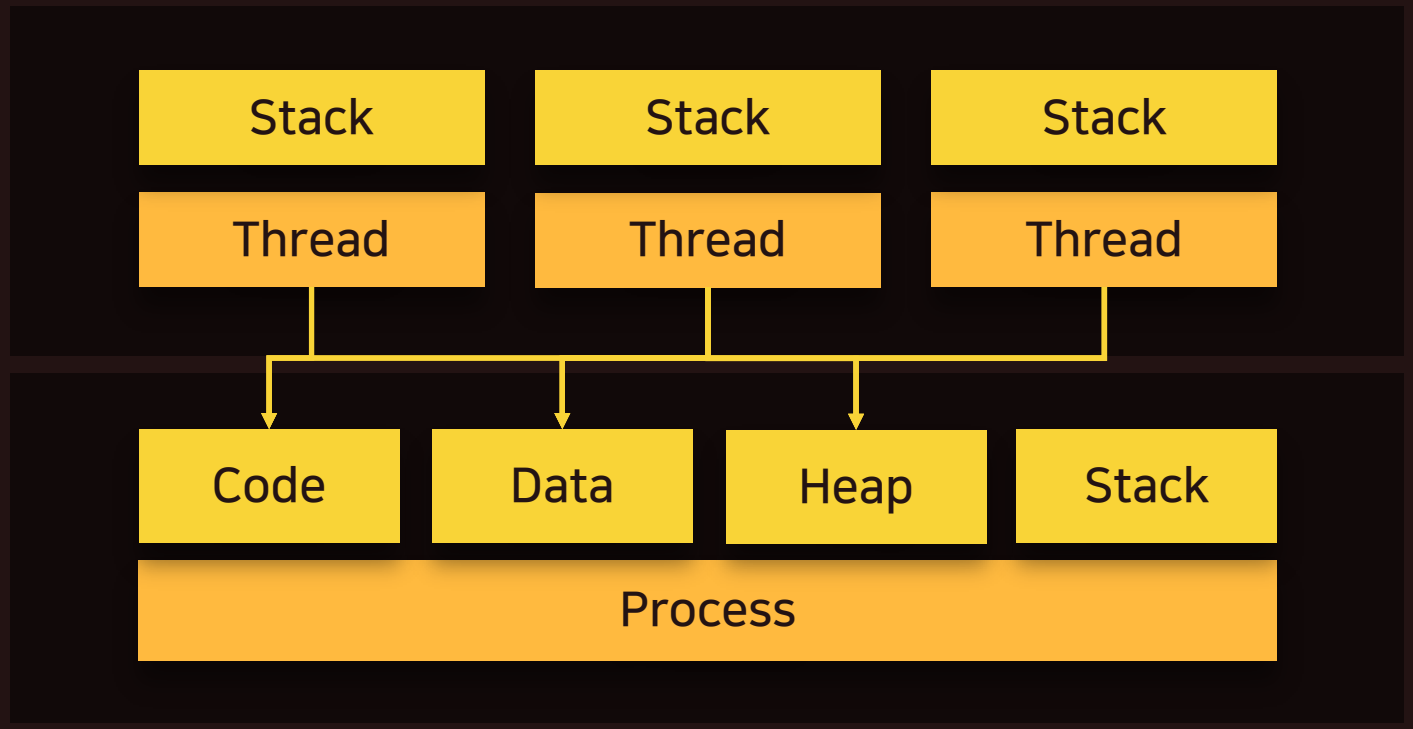
Kernel Space

커널 스페이스에서는 모든 메모리와 CPU 명령셋을 실행 가능하며, 커널 수준의 특권을 통해 코드 실행을 관리할 수 있으며 커널 레벨 함수는 이 영역에서 실행 된다.

소개

goroutine

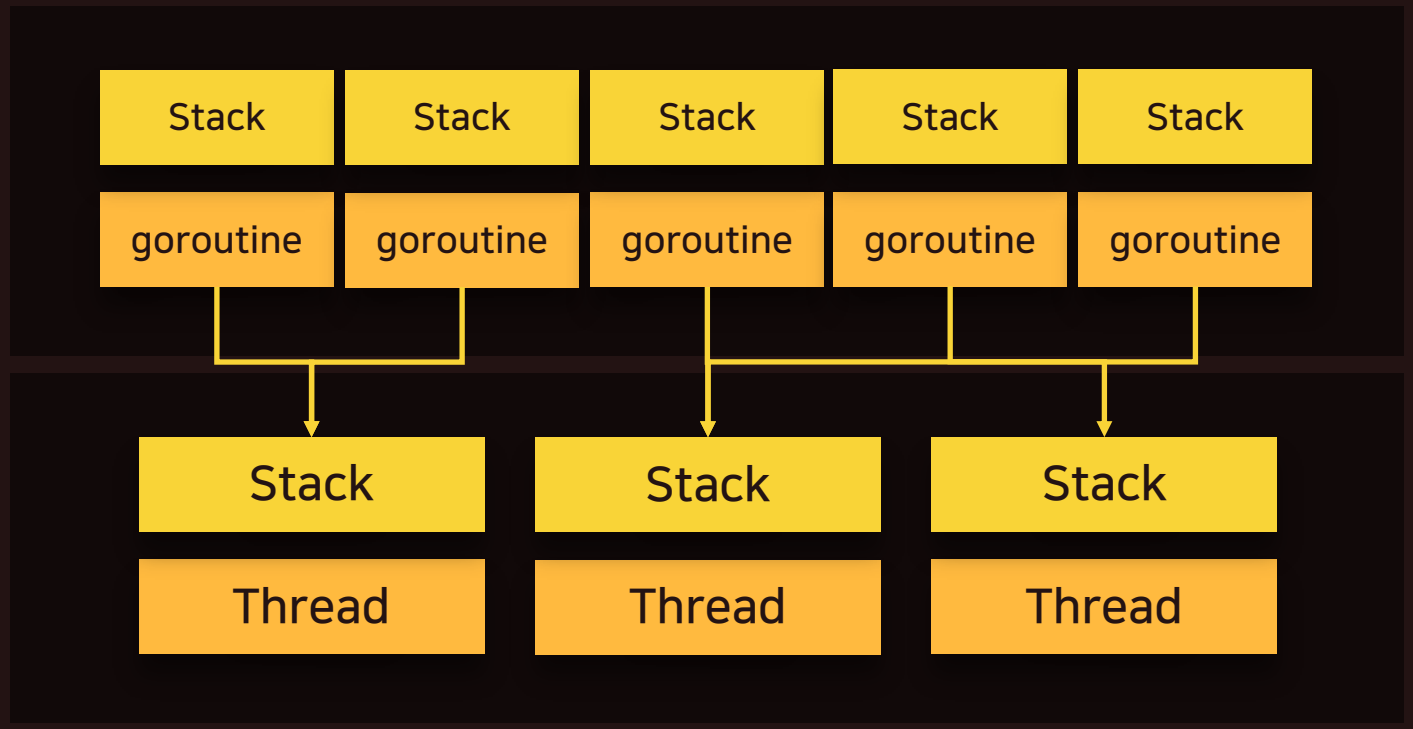
메모리 관점에서 보면 다음과 같습니다.



소개

goroutine

고루틴은 이보다 더 작은 개념이라 볼 수 있습니다.

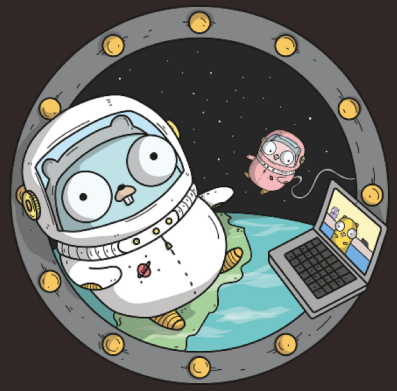


소개

이걸 보시는 독자의 마음

심기불편

Core Level



GDG Golang Korea

소개

GMP structures

고 언어에서는 GMP라는 개념이 존재합니다.

G (Goroutine)

고루틴의 구현체라고 볼 수 있습니다. 일반적으로 G는 P의 대기열에 묶여있습니다.

M (Machine)

OS 레벨의 쓰레드를 의미합니다. M은 P에 묶여있으며(할당되며), 상황에 따라 스피닝 상태로 있을 수 있습니다.

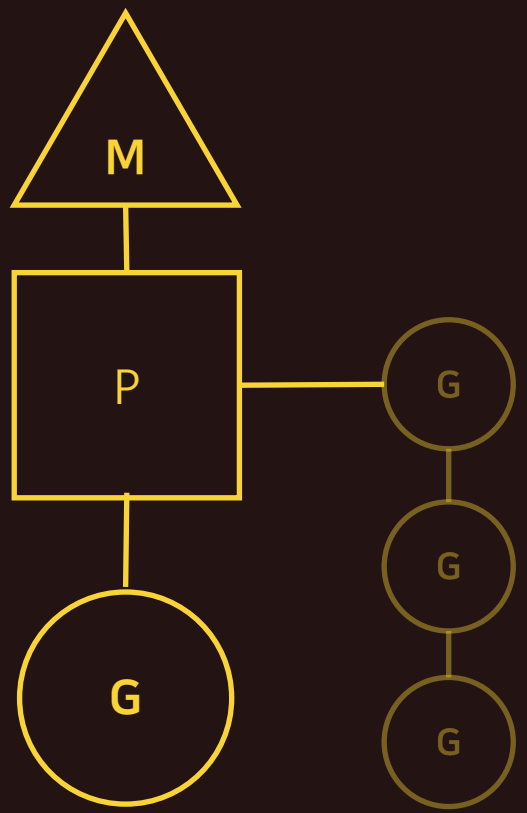
P (Processor)

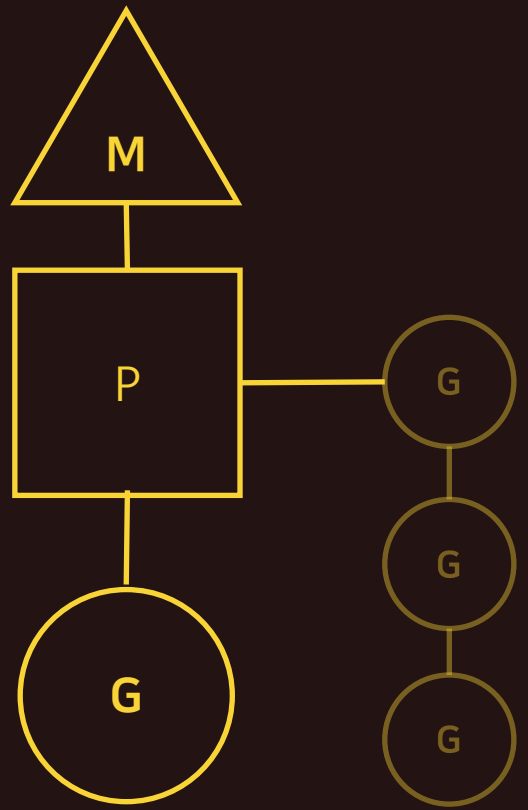
P는 프로세서를 의미합니다. P는 GOMAXPROCS 만큼 존재할 수 있으며 P는 하나의 M만을 실행할 수 있습니다.
P는 컨텍스트(Context) 정보를 담고 있습니다, 컨텍스트는 M에 따라 스위칭이 일어납니다.
P는 LRQ(Local Run Queue)를 가집니다.

소개

GMP structures

그림으로 보면 아래와 같습니다.





Local Run Queue (LRQ)

Global Run Queue (GRQ)

Net Poller

소개

Network Poller

System call은 무겁습니다.
System call을 효율적으로 처리하기 위해서 각 OS는 다양한 장치를 제공합니다.

- **kqueue - MacOS**
- **epoll - Linux**
- **iocp - Windows**

Core Level

Create a new G

src/runtime/proc.go

```
// Create a new g running fn with siz bytes of arguments.
// Put it on the queue of g's waiting to run.
// The compiler turns a go statement into a call to this.
// Cannot split the stack because it assumes that the arguments
// are available sequentially after &fn; they would not be
// copied if a stack split occurred.
//go:nosplit
func newproc(siz int32, fn *funcval) {
    argp := add(unsafe.Pointer(&fn), sys.PtrSize)
    gp := getg()
    pc := getcallerpc()
    systemstack(func() {
        newproc1(fn, (*uint8)(argp), siz, gp, pc)
    })
}
```

newproc 함수는 새로운 고루틴을 생성할 때
호출됩니다.

이 함수는 newproc1을 systemstack을 이용해
시스템 스택에서 호출하도록 합니다.

일반적으로 고루틴에서 초기에 생성된 스택의 사이즈는
시스템 스택보다 작지만,
시스템 스택의 경우 스택 복사 과정이 일어납니다.

Core Level

Create a new G

src/runtime/proc.go

```
// Create a new g running fn with narg bytes of arguments starting
// at argp. callerpc is the address of the go statement that created
// this. The new g is put on the queue of g's waiting to run.
func newproc1(fn *funcval, argp *uint8, narg int32, callergp *g, callerpc
uintptr) {
    // ...
    _p_ := _g_.m.p.ptr()
    newg := gfget(_p_)
    if newg == nil {
        newg = malg(_StackMin)
        casgstatus(newg, _Gidle, _Gdead)
        allgadd(newg)
    }
}
```

newproc1에서는 StackMin만큼의 스택 사이즈를 malg 함수를 통해 할당 시키는 것을 볼 수 있습니다.

(malg는 고루틴을 스택 사이즈와 함께 할당하는 함수입니다.)

Core Level

Create a new G

src/runtime/stack.go

```
const (
    // StackSystem is a number of additional bytes to add
    // to each stack below the usual guard area for OS-specific
    // purposes like signal handling. Used on Windows, Plan 9,
    // and iOS because they do not use a separate stack.
    _StackSystem = sys.GoosWindows*512*sys.PtrSize + sys.GoosPlan9*512 +
sys.GoosDarwin*sys.GoarchArm*1024 + sys.GoosDarwin*sys.GoarchArm64*1024

    // The minimum size of stack used by Go code
    _StackMin = 2048

    // ...

    runqput( _p_, newg, true)

    // ...

    if atomic.Load(&sched.npidle) != 0 && atomic.Load(&sched.nmspinning) == 0
&& mainStarted {
        wakep()
    }
)
```

_StackMin은 기본적으로 2048 (2kb)로 정의되어 있습니다.

따라서 모든 고루틴은 초기에 2kb의 스택 사이즈를 가집니다.

Runqput을 통해 로컬 큐에 새로운 고루틴을 할당해, 실행 대기열에 추가합니다.

이후 사용 가능한(유휴 상태인) P가 존재하고 M(Process)가 스피닝 쓰레드가 아니라면 wakep를 호출합니다.

wakep는 유휴 상태의 P를 깨우고 M을 통해 G를 실행시키게 됩니다.

Core Level

Try to add P for new G

src/runtime/proc.go

```
// Tries to add one more P to execute G's.
// Called when a G is made runnable (newproc, ready).
func wakep() {
   // be conservative about spinning threads
   if !atomic.Cas(&sched.nmspinning, 0, 1) {
      return
   }
   startm(nil, true)
}
```

wakep 함수는 새로운 고루틴을 위해 P를 추가하여 고루틴을 실행할 때 호출됩니다.

Atomic CAS(Compare And Swap)을 통해 쓰레드를 스피닝 상태를 만들고, M을 깨우게 됩니다.

CoreLevel

goroutine vs threads

	고루틴(goroutine)	쓰레드(thread)
Memory	2kb	1mb
Setup / Teardown Cost	비교적 저렴함	높은 비용
Context Switching Cost	비교적 저렴함 Switch 항목: PC, SP, DX	모든 레지스터의 Switch가 요구 됨 모든 레지스터: 16 general 레지스터, PC, SP, 16XMM, FP coprocessor state

* PC: Program Counter, SP: Stack Pointer, DX: Data Register

CoreLevel

Goroutine의 Context Switching 타이밍

- 언버퍼드 영역에 접근할 때 (Allocation way syscall)
- System I/O를 호출 할 때
- Memory를 할당하는 타이밍
- time.Sleep() 함수를 실행 할 때
- runtime.Gosched() 할 때

Thank you

한성민

kenneth@pigno.se

https://github.com/KennethanCeyer