동시성 패턴

2016-09-20

Case Studies

1. `Runner`
2. `Pool`
3. ~~`Work`~~

Runner

Define Runner sturct

```
// Runner 타입은 주어진 시간동안 일련의 작업을 수행한다.
type Runner struct {
    // 운영체제로부터 전달되는 인터럽트 신호를 수신하는 채널
    interrupt chan os.Signal
    // 처리가 종료되었음을 알리는 채널
    complete  chan error
    // 지정된 시간이 초과되었음을 알리는 채널
    timeout   <-chan time.Time
    // 인덱스 순서로 처리될 작업 목록을 저장하는 슬라이스
    tasks     []func(int)
}
```

- **time** package 의 ***After*** 함수는 **<-chan Time** 를 반환한다.

Define Errors

```
// timeout 채널에서 값을 수신하면 ErrTimeout을 리턴한다.
var ErrTimeout = errors.New("시간을 초과했습니다.")

// 운영체제에서 인터럽트 이벤트를 수신하면 ErrInterrupt를 리턴한다.
var ErrInterrupt = errors.New("운영체제 인터럽트 신호를 수신했습니다.")
```

Define New Function

```
func New(d time.Duration) *Runner {
    return &Runner{
        interrupt: make(chan os.Signal, 1),
        complete:  make(chan error),
        timeout:   time.After(d),
    }
}
```

Define Add Method

```
// Runner 타입에 작업을 추가하는 메소드. 작업은 int형 ID를 매개변수로 전달받는다.
func (r *Runner) Add(tasks ...func(int)) {
    r.tasks = append(r.tasks, tasks...)
}
```

Define Start Method

```
func (r *Runner) Start() error {
    signal.Notify(r.interrupt, os.Interrupt) // 모든 종류의 인터럽트 신호를 수신한다.

    go func() {
        r.complete <- r.run()      // 각 작업을 각기 다른 고루틴을 통해 실행한다.
    }()

    select {
    case err := <-r.complete:    // 작업 완료 신호를 수신한 경우
        return err
    case <-r.timeout:           // 작업 시간 초과 신호를 수신한 경우
        return ErrTimeout
    }
}
```

select

- **`Channel`** 을 위한 **`switch`** 문.
- **`case`** 에 정의한 채널을 **한 번** 만 받는다.
- **`case`** 에 정의한 채널 값이 올 때 까지 블로킹 된다.
- **`default`** 를 선언하면 해당 채널이 오지 않았을 때 처리가 이루어 진다.

Define `run` Method

```
// 개별 작업을 실행하는 메소드
func (r *Runner) run() error {
    for id, task := range r.tasks {
        if r.gotInterrupt() { // OS 로 부터 인터럽트 신호를 수신했는지 확인
            return ErrInterrupt
        }

        task(id) // 작업을 실행
    }
    return nil
}
```

Define gotInterrupt Method

```
func (r *Runner) gotInterrupt() bool {
    select {
    case <-r.interrupt: // 인터럽트 이벤트가 발생하는 경우
        // 이후에 발생하는 인터럽트 신호를 더 이상 수신하지 않도록 한다.
        signal.Stop(r.interrupt)
        return true

    default:            // 작업을 계속해서 실행한다.
        return false
    }
}
```

createTask Function

```
func createTask() func(int) {
    return func(id int) {
        log.Printf("프로세서작업 #%d.", id)
        time.Sleep(time.Duration(id) * time.Second)
    }
}
```

main Function

```
const timeout = 3 * time.Second

r := runner.New(timeout)

r.Add(createTask(), createTask(), createTask())
```

main Function

```
if err := r.Start(); err != nil {
    switch err {
    case runner.ErrTimeout:
        log.Println("지정된 작업 시간을 초과했습니다.")
        os.Exit(1)
    case runner.ErrInterrupt:
        log.Println("운영체제 인터럽트가 발생했습니다.")
        os.Exit(2)
    }
}
```

One More Thing

error 를 const 로 정의할 수는 없을까?

```
const ErrTimeout = errors.New("시간을 초과했습니다.")
const ErrInterrupt = errors.New("운영체제 인터럽트 신호를 수신했습니다.")
```

- const 로 선언하면 에러 발생

```
const initializer errors.New("시간을 초과했습니다.") is not a constant
const initializer errors.New("운영체제 인터럽트 신호를 수신했습니다.") is not a constant
```

Tips

1. **`const`** 에는 **`const`** 만 저장 할 수 있다.
2. **`string`** 은 **`const`** 타입이다.
3. **`error`** 는 **`interface`** 타입이다.

- **reference**
  constant-errors (dave.cheney)

Custom err package

```
// Errors 타입 정의
type Errors string

// error interface 를 채용하기 위해 함수 선언
func (err Errors) Error() string {
    return string(err)
}
```

Use err.Errors

```
const (
    ErrTimeout = err.Errors("시간을 초과했습니다.")
    ErrInterrupt = err.Errors("운영체제 인터럽트 신호를 수신했습니다.")
)

switch err {
case runner.ErrTimeout:
    // do
case runner.ErrInterrupt:
    // do
}
```

Pool

Define Pool sturct

```
// Pool 구조체는 여러 개의 고루틴에서 안전하게 공유하기 위한 리소스 집합을 관리
type Pool struct {
    m         sync.Mutex
    resources chan io.Closer
    factory   func() (io.Closer, error)
    closed    bool
}
```

- *io.Closer* 는 Close()를 갖는 인터페이스다.

Define Errors

```
var ErrPoolClosed = errors.New("풀이 닫혔습니다.")
```

Define

```
func New(fn func() (io.Closer, error), size uint) (*Pool, error) {
    if size <= 0 {
        return nil, errors.New("풀의 크기가 너무 작습니다.")
    }

    return &Pool{
        factory:   fn,
        resources: make(chan io.Closer, size),
    }, nil
}
```

Define Acquire Method

```
func (p *Pool) Acquire() (io.Closer, error) {
    select {
    case r, ok := <-p.resources:
        log.Println("리소스 획득:", "공유된 리소스")
        if !ok {
            return nil, ErrPoolClosed
        }
        return r, nil

    default:
        log.Println("리소스 획득:", "새로운 리소스")
        return p.factory()
    }
}
```

Define Release Method

```
func (p *Pool) Release(r io.Closer) {
    p.m.Lock() // 안전한 작업을 위해 잠금을 설정한다.
    defer p.m.Unlock()

    if p.closed {
        r.Close() // 풀이 닫혔으면 리소스를 해제한다.
        return
    }
    // select 문
}
```

Define select in Release Method

```
select {
case p.resources <- r:
    log.Println("리소스 반환:", "리소스 큐에 반환")

default:
    log.Println("리소스 반환:", "리소스 해제")
    r.Close()
}
```

Define `Close` Method

```
func (p *Pool) Close() {
    p.m.Lock()
    defer p.m.Unlock()

    if p.closed {
        return
    }

    p.closed = true
    close(p.resources)

    for r := range p.resources {
        r.Close()
    }
}
```

Pool Example

```
// Define constant
const (
    maxGoroutines  = 25
    pooledResorces = 2
)
```

Define dbConnection struct

```
type dbConnection struct {
    ID int32
}

// io.Closer 인터페이스를 구현하기 위해 `Close()` 메소드 구현
func (dbConn *dbConnection) Close() error {
    log.Println("닫힘: 데이터베이스 연결,", dbConn.ID)
    return nil
}
```

Define createConnection Function

```
var idCounter int32

func createConnection() (io.Closer, error) {
    id := atomic.AddInt32(&idCounter, 1)
    log.Println("생성: 새 데이터베이스 연결", id)

    return &dbConnection{id}, nil
}
```

Define performQueries Function

```
func performQueries(query int, p *pool.Pool) {
    conn, err := p.Acquire()
    if err != nil {
        log.Println(err)
        return
    }

    defer p.Release(conn)

    time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
    // conn.(*dbConnection) 으로 interface 타입을 dbConnection 참조 타입으로 타입 변환.
    log.Printf("질의: QID[%d] CID[%d]\n", query, conn.(*dbConnection).ID)
}
```

Define main Function (1)

```
var wg sync.WaitGroup
// 고루틴 갯수를 추가
wg.Add(maxGoroutines)


p, err := pool.New(createConnection, pooledResorces)
if err != nil {
    log.Println(err)
}
```

Define main Function (2)

```
for query := 0; query < maxGoroutines; query++ {

    go func(q int) {
        performQueries(q, p) // 고루틴마다 함수 실행
        wg.Done()
    }(query)
}

wg.Wait() // 고루틴 대기

log.Println("프로그램을 종료합니다.")
p.Close()
```