Chapter 19 메서드

날짜	@December 29, 2021
발표자	장예서

19.1 메서드 선언

리시버를 사용해서 메서드 정의하기

```
func (r Rabbit) info() int {
  return r.width * r.height
}
```

리시버: (r Rabbit) → 구조체 변수(r)는 매개변수처럼 사용

메서드명: info() → Rabbit 타입에 속한다

- 리시버로는 모든 로컬 타입들이 가능
- 로컬 타입: 해당 패키지 안에서 type 키워드로 선언된 타입

```
package main

import "fmt"

type account struct {
  balance int
}

func withdrawFunc(a *account, amount int) { // 일반 함수
  a.balance -= amount
}

func (a *account) withdrawMethod(amount int) { // 메서드
  a.balance -= amount
}

func main() {
```

```
  a := &account{100}

  withdrawFunc(a, 30)

  a.withdrawMethod(30)

  fmt.Printf("%d \n", a.balance)
}
```

출력문

```
40
```

- 메서드는 일반적으로 리시버 타입이 선언된 파일 안에 정의한다
- 메서드는 해당 리시버 타입에 속한다, withdrawMethod()는 *accound 타입에 속함
- 점 . 연산자를 사용해 해당 타입에 속한 메서드를 호출한다

19.1.1 별칭 리시버 타입

모든 로컬 타입이 리시버 타입으로 가능 → 별칭 타입도 리시버가 될 수 있고 메서드를 가질 수 있다. int와 같은 내장 타입들도 별치 타입을 활용하여 메서드를 가질 수 있다.

```
package main

import "fmt"

type myInt int

func (a myInt) add(b int) int {
  return int(a) + b
}

func main() {
  var a myInt = 10
  fmt.Println(a.add(30))
  var b int = 20
  fmt.Println(myInt(b).add(50))
}
```

출력문

```
40
70
```

- int 타입의 별칭 myInt 타입 선언
- myInt 타입의 메서드 add() 선언
- 별칭 타입 간 타입 변환: 변수 b int 타입→ myInt 타입으로 변환 후 add() 메서드 사용

19.2 메서드는 왜 필요한가?

함수와의 중요한 차이점: **소속**

일반 함수는 어디에도 속하지 않지만 메서드는 리시버에 속한다. 메서드를 사용해서 리시버 타입의 데이터와 기능을 묶을 수 있다.

좋은 프로그래밍이라면 결합도를 낮추고 응집도를 높여야 한다. 메서드는 테이터와 관련 기능을 묶기 때문에 **코드 응집도를 높이는 중요한 역할**을 한다. (응집도가 낮으면 새로운 기능을 추가할 때 흩어진 모든 부분을 검토하고 고쳐야 하는 산탄총 수술 문제가 발생)

19.2.1 객체지향: 절차 중심에서 관계 중심으로 변화

메서드 등장 이전: 절차 중심의 프로그래밍. 데이터와 기능이 분리되어서 기능들을 어떤 순서로 실행하는지를 정의함, 순서도를 중요시함.

메서드 등장: 데이터와 기능이 묶인 단일 객체로써 동작. 객체란 데이터와 기능을 갖는 타입, 객체 인스턴스들은 서로 유기적으로 소통하고 관계 맺게 됨. 절차 → **객체 간 관계 중심**으로 프로그래밍 패러다임 변화 = **객체지향 프로그래밍** (OOP), 클래스 다이어그램 중시.

```
type Student struct{
  FirstName string
  LastName string
  Age int
}

func (s *Student) EnrollClass(c *Subject){
  ...
}

func (s *Student) SendReport(p *Professor, r *Report){
  ...
}
```

- Student 구조체에 속하는 EnrollClass(), SendReport() 메서드
- Student : 이름, 나이 정보 + 과목 등록 기능 + 리포트 전송 기능을 가진 객체

Go 언어에서는 클래스와 상속 지원 X 하지만 객체 간 상호 관계 중심 → 충분한 OOP 언어

19.3 포인터 메서드 vs 값 타입 메서드

리시버를 값 타입과 포인터로 정의할 수 있다.

```
package main

import "fmt"

type account struct {
  balance   int
  firstName string
  lastName  string
}

// 포인터 타입 메서드
func (a1 *account) withdrawPointer(amount int) {
  a1.balance -= amount
}

// 값 타입 메서드
func (a2 account) withdrawValue(amount int) {
  a2.balance -= amount
}

func (a3 account) withdrawReturnValue(amount int) account {
  a3.balance -= amount
  return a3
}

func main() {
  var mainA *account = &account{100, "Joe", "Park"}
  mainA.withdrawPointer(30)
  fmt.Println(mainA.balance)

  mainA.withdrawValue(20)
  fmt.Println(mainA.balance)

  var mainB account = mainA.withdrawReturnValue(20)
  fmt.Println(mainB.balance)

  mainB.withdrawPointer(30)
  fmt.Println(mainB.balance)
}
```

출력문

```
70
70
50
20
```

- withdrawPointer() 메서드 → 포인터 리시버 (*account 타입에 속함)
- withdrawValue(), withdrawReturnValue() 메서드 → 값 타입 (account 타입에 속함)
- 포인터 메서드 : 호출 시 포인터가 가리키고 있는 메모리의 주솟값이 복사됨
- 값 타입 메서드 : 리시버 타입의 모든 값이 복사됨

포인터 메서드

- withdrawPointer() 메서드 호출 : mainA 포인터 변수가 갖는 값 복사 → a1과 mainA는 같은 인스턴스를 가리킨다 (a1의 balance 변경 = mainA의 balance 변경)

값 타입 메서드

- withdrawValue() 메서드 호출 : mainA의 모든 값이 a2로 복사 → a2와 mainA는 서로 다른 인스턴스 (a2의 balance 변경 ≠ mainA의 balance 변경)
- withdrawReturnValue() 메서드 호출 : account 구조체의 모든 값이 메서드 호출 시와 메서드 결괏값 반환 시 복사, mainB는 메서드 호출 이후 변경된 값을 갖는 새로운 객체 → a3, mainA, mainB 모두 서로 다른 객체

포인터 메서드 vs 값 타입 메서드

포인터 메서드 : 메서드 내부에서 리시버 값 변경 가능 = 인스턴스 중심

값 타입 메서드 : 호출하는 쪽과 메서드 내부의 값은 별도 인스턴스 → 메서드 내부에서 리시버 값 변경 불가능 = 값 중심

연습문제

1. *Cart
2.

```
package main

type ParkingLot struct {
  LotSize int
}

func ParkCar(lot *ParkingLot, carSize int) {
  lot.LotSize -= carSize
}

func (lot *ParkingLot) ParkCar(carSize int) {
  lot.LotSize -= carSize
}

func main() {
  lot := &ParkingLot{100}
  ParkCar(lot, 10)

  lot.ParkCar(10)
}
```

3.

```
hello go world
HELLO GO WORLD
```

4.

```
package main

import (
  "time"
)

type Courier struct {
  Name string
}

type Product struct {
  Name  string
  Price int
  ID    int
}

type Parcel struct {
  Pdt           *Product
  ShippedTime   time.Time
```

```
  DeliveredTime time.Time
}

func (c *Courier) SendProduct(pdt *Product) *Parcel {
  p := &Parcel{}
  p.Pdt = pdt
  p.ShippedTime = time.Now()
  return p
}

func (p *Parcel) Delivered() *Product {
  p.DeliveredTime = time.Now()
  return p.Pdt
}
```