Go 언어 치트시트

Go 언어 개요

Go 프로그래밍 언어는 2007년 구글(Google)에서 개발을 시작하여 2012년에 버전 1.0이 완성되었습니다. 현재는 1.21 버전까지 출시되었습니다. Go는 기존 언어들의 여러 장점을 결합하여 설계되었습니다. C++와 마찬가지로 컴파일러를 통해 기계어로 변환되는 정적 타입(Statically Typed) 언어이며, Java처럼 가비지 컬렉션(Garbage Collection) 기능을 지원하여 메모리를 효율적으로 관리합니다. Go는 단순하고 간결한 설계를 지향하며, Java의 약 절반 수준인 25개의 키워드만으로도 충분히 프로그래밍이 가능합니다. 특히 CSP(Communicating Sequential Processes) 스타일의 동시성(Concurrent) 프로그래밍을 강력하게 지원하는 것이 큰 특징입니다.

설치 및 환경 설정

Go 프로그래밍을 시작하려면 공식 웹사이트에서 각 운영체제에 맞는 설치 파일을 다운로드해야 합니다. 윈도우 환경에서는 MSI 파일을 실행하면 되는데, 기본적으로 `C:\Program Files\Go` 폴더에 설치되며 설치 프로그램이 `bin` 디렉토리를 시스템 PATH 환경변수에 자동으로 추가합니다. 설치가 완료되면 `go.exe` 컴파일러를 통해 프로그램을 컴파일하거나 실행할 수 있으며, 소스 파일은 `.go` 확장자를 사용합니다. 설치 상태는 다음 명령어로 확인할 수 있습니다.

```
C:\> go version
go version go1.21.3 windows/amd64
```

주요 환경 변수 (go env)

Go는 내부적으로 여러 환경 변수를 사용하며, `go env` 명령어로 이를 확인할 수 있습니다. 특히 다음 두 가지 변수가 중요합니다.

- **GOROOT**: Go가 설치된 디렉토리를 가리킵니다. 표준 패키지(예: `fmt`) 등이 이 경로 아래의 `src` 폴더에 위치합니다.
- **GOPATH**: Go 프로젝트의 작업 공간 역할을 합니다. 과거에는 소스 코드와 의존성 관리의 핵심이었으나, Go 1.11에서 모듈(Modules) 기능이 도입된 이후로는 그 비중이 줄어들었습니다. 현재는 주로 서드파티 실행 파일들이 설치되는 경로로 사용됩니다.

변수 및 상수 선언

변수 (Variables)

변수는 `var` 키워드를 사용하여 선언합니다. 변수명 뒤에 타입을 명시하는 형식을 취합니다.

```
var a int // 정수형 변수 a 선언
```

선언과 동시에 초기값을 할당할 수도 있습니다.

```
var f float32 = 11.0
```

Go에서는 선언된 변수를 사용하지 않으면 컴파일 에러가 발생하므로, 사용하지 않는 변수는 삭제해야 합니다. 동일한 타입의 변수는 한꺼번에 나열하여 선언할 수 있으며, 초기값 또한 순서대로 할당 가능합니다.

```
var i, j, k int = 1, 2, 3
```

할당되는 값의 타입을 추론하는 기능을 자주 활용합니다. 또한 함수 내부에서는 `:=` (Short Assignment Statement)를 사용하여 `var` 키워드 없이 간결하게 선언과 할당을 동시에 처리할 수 있습니다.

```
i := 1 // 함수 내부에서만 사용 가능
```

상수 (Constants)

상수는 `const` 키워드로 선언하며, 선언 시 반드시 값을 할당해야 합니다. 변수와 마찬가지로 타입 추론이 가능합니다.

```
const c = 10
const s = "Hi"
```

여러 상수를 괄호로 묶어 정의할 수 있으며, `iota` 식별자를 사용하면 0부터 시작하는 순차적인 정수 값을 편리하게 부여할 수 있습니다.

```
const (
    Apple = iota // 0
    Grape        // 1
    Orange       // 2
)
```

Go 예약어 (Keywords)

다음 키워드들은 언어 자체에서 예약되어 있으므로 변수명이나 상수명 등 식별자로 사용할 수 없습니다.

```
break default func interface select case
defer go map struct chan else goto
package switch const fallthrough if
range type continue for import return
var
```

데이터 타입 (Data Types)

Go는 다음과 같은 기본 데이터 타입을 제공합니다.

- **bool**: 부울린 타입 (true/false)
- **string**: 문자열 타입 (한번 생성되면 수정할 수 없는 Immutable 속성)
- **정수형**: `int`, `int8~64`, `uint`, `uint8~64`, `uintptr` 등
- **실수 및 복소수**: `float32`, `float64`, `complex64`, `complex128`
- **기타**:
  - `byte`: `uint8`과 동일하며 바이트 데이터를 다룰 때 사용
  - `rune`: `int32`와 동일하며 유니코드 코드 포인트(문자 하나)를 표현할 때 사용

기본 자료형 요약

`int`와 `uintptr` 타입의 크기는 시스템 아키텍처(32비트/64비트)에 따라 결정됩니다. 특별한 이유가 없다면 정수형 데이터에는 일반적인 `int` 타입을 사용하는 것을 권장합니다.

제로 값 (Zero Values)

변수 선언 시 초기값을 명시하지 않으면 해당 타입의 기본값(Zero Value)이 자동으로 할당됩니다.

- 숫자형: `0`
- 부울형: `false`
- 문자열: `""` (빈 문자열)

데이터 타입 변환 (Type Conversion)

특정 타입을 다른 타입으로 변환할 때는 `T(v)` 형식을 사용합니다. Go는 암시적 타입 변환을 허용하지 않으므로, 타입이 다른 데이터를 연산하거나 할당할 때는 반드시 명시적으로 변환해주어야 합니다.

```
i := 42
f := float64(i)
u := uint(f)
```

연산자 (Operators)

Go는 산술, 관계, 논리, 비트, 할당, 포인터 연산자 등을 지원합니다.

주요 연산자 종류

- **산술 연산자**: `+`, `-`, `*`, `/`, `%`, `++`, `--` (사칙연산 및 증감)
- **관계 연산자**: `==`, `!=`, `<`, `>`, `<=`, `>=` (비교 연산)
- **논리 연산자**: `&&`, `||`, `!` (AND, OR, NOT)
- **비트 연산자**: `&`, `|`, `^`, `&^`, `<<`, `>>` (비트 단위 조작)
- **할당 연산자**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- **포인터 연산자**: `&` (주소 추출), `*` (역참조). 단, 포인터 산술(포인터에 값을 더하거나 빼는 행위)은 지원하지 않습니다.

제어문: 조건문

if 조건문

`if` 문은 조건식이 참일 때 코드 블록을 실행합니다. 조건식을 괄호로 둘러쌀 필요가 없으며, 시작 브레이스({)는 반드시 `if` 문과 같은 라인에 위치해야 합니다. 조건식은 반드시 `bool` 타입이어야 합니다.

```
if val := i * 2; val < max { // 조건문 내에서 변수 선언 가능 (Optional Statement)
    println(val)
}
```

switch 선택문

여러 조건을 비교할 때 사용합니다. `case`문에 콤마를 사용하여 여러 값을 나열할 수 있습니다. Go의 `switch`는 다른 언어와 달리 각 `case`마다 `break`를 명시하지 않아도 자동으로 빠져나갑니다.

- **특징**:
  - `switch` 뒤에 조건식을 생략하면 `true`로 간주되어 첫 번째 참인 `case`를 실행합니다.
  - `fallthrough` 키워드를 사용하면 강제로 다음 `case` 블록까지 실행하게 할 수 있습니다.
  - 변수의 실제 값뿐만 아니라 **타입**(`v.(type)`)에 따라 분기할 수도 있습니다.

제어문: 반복문

for 반복문

Go에서 반복문은 `for`가 유일합니다. `while` 문 역시 `for`를 통해 구현합니다.

- **기본 형식**: `for 초기값; 조건식; 증감식 { ... }`
- **조건식만 사용**: `for 조건식 { ... }` (while 문과 동일한 역할)
- **무한 루프**: `for { ... }`
- **for range**: 컬렉션(배열, 슬라이스, 맵 등)의 요소를 순회할 때 사용합니다.

```
for index, name := range names {
    println(index, name)
}
```

흐름 제어: break, continue, goto

- `break`: 루프나 `switch` 문을 즉시 종료합니다.
- `continue`: 루프의 나머지 부분을 건너뛰고 다음 반복을 시작합니다.
- `goto`: 지정된 레이블 위치로 무조건 이동합니다.
- 레이블을 지정한 `break`를 사용하면 중첩된 루프를 한 번에 빠져나오는 등의 정밀한 제어가 가능합니다.

함수 정의 및 활용

함수는 `func` 키워드로 정의하며, 파라미터 타입과 리턴 타입을 명확히 지정합니다.

```
func sum(a int, b int) int {
    return a + b
}
```

파라미터 전달 방식

- **Pass By Value**: 기본값 전달 방식으로, 함수 내부에서 값을 변경해도 원본 변수에는 영향이 없습니다.
- **Pass By Reference**: 포인터(`*`)를 사용하여 주소를 전달합니다. 함수 내부에서 원본 데이터를 직접 수정할 수 있습니다.

가변 인자 함수 (Variadic Function)

개수가 정해지지 않은 파라미터를 받을 때 `...`을 사용합니다.

```
func say(msg ...string) {
    for _, s := range msg {
        println(s)
    }
}
```

복수 값 반환 및 Named Return Parameters

함수는 여러 개의 값을 동시에 반환할 수 있습니다. 또한 리턴 파라미터에 이름을 붙여 가독성을 높일 수 있습니다.

```
func sum(nums ...int) (count int, total
int) {
    total = 0
    for _, n := range nums { total +=
n }
    count = len(nums)
    return // 명시된 이름의 변수들이 반환됨
}
```

익명 함수 및 클로저

익명 함수 (Anonymous Functions)

이름 없이 정의하여 변수에 할당하거나 직접 실행할 수 있는 함수입니다.

```
sum := func(a, b int) int { return a +
b }
println(sum(1, 2))
```

클로저 (Closures)

함수 외부의 변수를 참조하고 해당 변수의 상태를 유지하는 함수 값을 말합니다. 함수가 종료된 후에도 참조된 외부 변수의 생명 주기가 유지되어 상태를 보존할 수 있습니다.

컬렉션 (Collections)

배열 (Arrays)

고정된 크기를 가지며 동일한 타입의 연속된 데이터를 저장합니다. 크기 자체가 타입의 일부로 취급됩니다.

```
var a [3]int = [3]int{1, 2, 3}
```

슬라이스 (Slices)

배열과 유사하지만 크기가 유동적으로 변할 수 있는 동적 배열입니다. `make()` 함수를 통해 길이와 용량을 지정하여 생성하거나, 기존 배열/슬라이스의 일부를 발췌하여 만들 수 있습니다. `append()` 함수를 사용하여 요소를 추가합니다.

맵 (Maps)

키(Key)와 값(Value)의 쌍으로 데이터를 저장하는 해시 테이블 기반 자료구조입니다. `make()` 함수로 초기화해야 하며, 키의 존재 여부를 확인하는 기능을 제공합니다.

구조체 및 메서드

구조체 (Structures)

여러 필드를 하나로 묶어 사용자 정의 타입을 만듭니다. 클래스와 유사하지만 필드 데이터만 가집니다.

```
type Person struct {
    Name string
    Age  int
}
```

메서드 (Methods)

특정 타입(구조체 등)에 종속된 함수입니다. `func` 키워드와 함수명 사이에 수신자(Receiver)를 정의하여 선언합니다.

- **Value Receiver**: 데이터의 복사본을 전달받습니다.
- **Pointer Receiver**: 데이터의 주소를 전달받아 필드 값을 직접 수정할 수 있습니다.

인터페이스 (Interfaces)

메서드들의 집합체로, 특정 타입이 구현해야 할 행위의 명세(Contract) 역할을 합니다. 별도의 `implements` 키워드 없이 인터페이스가 정의한 메서드들을 모두 구현하면 해당 인터페이스를 충족하는 것으로 간주됩니다.

- **빈 인터페이스**(`interface{}`): 모든 타입을 담을 수 있는 범용 컨테이너 역할을 수행합니다.

에러 처리 (Error Handling)

Go는 예외(Exception) 대신 `error` 인터페이스를 반환하여 에러를 처리합니다. 함수 호출 후 반환된 에러가 `nil`인지 체크하는 방식이 일반적입니다.

지연 실행(defer) 및 예외 처리 (panic/recover)

- **defer**: 함수가 리턴되기 직전에 특정 구문을 실행하도록 예약합니다. 파일 닫기나 리소스 해제에 유용합니다.
- **panic**: 프로그램을 즉시 중단시키는 심각한 에러 상황을 발생시킵니다.
- **recover**: 발생한 패닉 상태를 복구하여 프로그램이 비정상 종료되는 것을 막습니다. 주로 `defer` 내부에서 사용됩니다.

고루틴 및 채널

고루틴 (Goroutines)

Go 런타임에서 관리하는 경량 논리 쓰레드입니다. `go` 키워드를 함수 호출 앞에 붙여 실행하며, 매우 적은 메모리 오버헤드로 동시 처리를 가능하게 합니다.

채널 (Channels)

고루틴 간에 데이터를 안전하게 주고받기 위한 통로입니다. 데이터를 주고받을 때 상대방이 준비될 때까지 대기하는 속성이 있어 별도의 명시적 잠금(Lock) 없이도 동기화가 가능합니다. 버퍼를 지정하여 비동기적인 송수신을 처리할 수도 있습니다.

웹 개발 및 테스팅 실전

Go는 강력한 표준 라이브러리와 더불어 `Gin`, `GORM` 등의 외부 패키지를 활용해 효율적인 웹 서버와 데이터베이스 연동 환경을 구축할 수 있습니다. 또한 단위 테스트, 벤치마크 테스트, 테이블 기반 테스트 기능을 언어 차원에서 기본적으로 지원하여 높은 코드 품질을 유지할 수 있도록 돕습니다.

배포 및 운영

Docker와 같은 컨테이너 기술이나 Kubernetes 환경에 맞춘 멀티스테이지 빌드 설정을 통해 작은 크기의 실행 바이너리로 배포할 수 있으며, 높은 성능과 안정성을 제공합니다.

Google Style Guide

핵심 원칙

- Clarity: 코드의 의도와 근거가 명확해야 함.
- Simplicity: 가장 단순한 방식으로 목표를 달성할 것.
- Concision: 신호 대 잡음비(signal-to-noise ratio)를 높임.
- Maintainability: 유지보수가 용이하게 작성.
- Consistency: 전체 코드베이스와 일관성 유지.

명명 규칙 (Naming)

- MixedCaps: `MixedCaps` 또는 `mixedCaps` 사용 (언더스코어 지양).
- Exported: 대문자로 시작하면 내보내기 가능, 소문자는 패키지 내부 전용.
- Package: 짧고 명확한 소문자 (언더스코어/캡스 금지).
- Receiver: 1 2자의 짧은 이름 (`f *File`) 권장.
- Variables: 범위가 좁을수록 짧은 이름 사용.

포매팅 (Formatting)

- gofmt: 모든 Go 파일은 `gofmt` 도구로 포맷팅되어야 함.
- 줄 길이: 고정된 제한은 없으나, 너무 길면 리팩토링 고려.
- Indentation: 탭(Tabs)을 사용하여 들여쓰기.

프로그래밍 관례

- Error Handling: `if err != nil { return err }` 패턴 준수. 에러는 값으로 취급.
- Panic: 라이브러리 코드에서 `panic` 사용 금지 (에러 반환 권장).
- Interfaces: 작게 유지 (보통 1 2개 메서드). 사용처에서 정의 선호.
- Concurrency: 채널과 뮤텍스 중 적합한 것 선택. 채널은 소유권과 흐름 제어에 유리.
- Tests: 테이블 기반 테스트(`Table-driven testing`) 권장.