차례

15. 동시성 1.15
    1. 통신에 의한 공유 1.15.1
    2. 고루틴 1.15.2
    3. 채널 1.15.3
    4. 채널의 채널 1.15.4
    5. 병렬화 1.15.5
    6. 누설 버퍼 1.15.6
16. 에러 1.16
    1. 패닉 - Panic 1.16.1
    2. 복구 - Recover 1.16.2
17. 웹 서버 1.17

README

효과적인 Go 프로그래밍

Effective Go 한국어 번역 문서입니다.

각 챕터별 담당자 테이블

http://bit.ly/2bZghCd

지속적 참여와 기여

효과적인 Go 프로그래밍의 번역은 그동안 참여해 주신 분들의 덕택으로 일단 마감되었습니다. Go 프로그래밍에 입문하고자 하시는 분들께 조금이나마 도움이 되었으면 좋겠다는 바람입니다.

독서하는 과정에 실시간 토론이 필요하신 분들은 기터방을 이용해 주십시요.

읽어 나가는 과정에서 발견되는 오류와 건의사항들은 지속적으로 gitbook의 discussion 기능을 통해 번역진에게 전달될 수 있습니다. 문단의 오른쪽 [+] 아이콘을 클릭하시고 질문이나 수정건의등을 기록하시면 됩니다. 또한 각 챕터마다 바닥에 Disqus 코멘트 기능을 이용하셔도 상관 없습니다.

다음 레포를 통한 PR 또한 환영하는 바입니다.

https://github.com/golangkorea/effective-go

소개

소개

- 원문: Introduction
- 번역자: Philbert Yoon (@ziwon)

Go는 새로운 언어이다. 비록 기존 언어의 아이디어들을 차용했지만, 독특한 속성을 지니고 있으며 이는 같은 계보로 쓰여진 프로그램들과는 성격이 다른 것으로 Go 프로그램을 효과적으로 만들어 준다. Java 프로그램은 Go가 아닌 Java로 쓰여졌다. 따라서 C++ 또는 Java 프로그램을 바로 Go로 변환하면 만족스러운 결과를 만들어내기는 어려울 것이다. 반면에, Go 관점에서 문제를 생각하면 완전히 다른 프로그램이 아닌 만족스러운 결과를 만들 수 있을 것이다. 다시 말해, Go를 잘 작성하기 위해서는 Go의 속성들과 Go 언어다운 코드들을 이해하는 것이 중요하다. 또한 다른 Go 프로그래머가 여러분이 작성한 Go 프로그래밍을 쉽게 이해할 수 있도록 네이밍, 포맷팅, 프로그램 구조 등과 같은 Go 프로그래밍을 위한 정해진 컨벤션들을 아는 것도 중요하다.

이 문서는 명확하고 Go 언어다운 Go코드를 작성하는 팁을 제공하며 언어 명세, Go 살펴보기, 그리고 Go 코드 작성하는 방법 여러분이 먼저 읽어야 할 모든 지식을 증대시킬 것이다.

예제

Go 패키지 소스는 코어 라이브러리로써 뿐만 아니라 언어를 사용하는 예제로써도 제공될 수 있도록 만들어졌다. 더 나아가, example Map처럼 많은 패키지들이 동작하는, 즉 독립적으로 실행가능한 예제들을 포함하고 있으며 golang.org 웹사이트에서 돌려볼 수 있다(필요하다면, "예제"라는 단어를 클릭해서 열어보라). 문제에 어떻게 접근해야하는지 혹은 무언가가 어떻게 구현되어 있는지 궁금하다면 라이브러리의 문서, 코드 그리고 예제들이 해결책이나 아이디어, 그리고 배경지식을 알려줄 것이다.

포맷팅

포맷팅 (Formatting)

- 원문 : Formatting
- 번역자 : MinJae Kwon (@mingrammer)

포맷팅 이슈는 중요한 것은 아니지만 가장 논쟁거리이다. 사람들은 각자 다른 포맷팅 스타일을 적용할 수도 있지만, 모든 사람들이 같은 스타일을 고수하여 더 이상 그럴 필요가 없어지고, 포맷팅 주제에 덜 신경을 쓰게된다면 더 좋을 것이다. 문제는 길고 규정적인 스타일 가이드 없이 어떻게 이 유토피아에 접근할 수 있는 가이다.

Go에서 우리는 새로운 접근법을 택하며, 머신에게 대다수의 포맷팅 이슈를 처리하도록 할 수 있다. `gofmt` 프로그램 (`go fmt`로도 사용할 수 있으며, 이는 소스 파일이 아닌 패키지 레벨에서 실행된다)은 Go 프로그램을 읽은 뒤, 표준 스타일의 들여쓰기와 수직정렬, 유지 그리고 필요시 주석을 재포맷팅한 소스를 내놓는다.

예를 하나 들면, Go에서는 구조체의 필드에 적힌 주석을 정렬하는데 신경을 쓸 필요가 없다. `Gofmt`가 대신해 줄 것이다. 아래를 보자.

```
type T struct {
    name string // name of the object
    value int // its value
}
```

`gofmt`는 각 열을 다음과 같이 정렬할 것이다

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

표준 패키지들에 있는 모든 Go 코드는 `gofmt`로 포맷팅이 되어있다.

몇 가지 포맷팅에 대한 상세한 내용이 남아있는데, 이를 매우 간단하게 요악해보면 다음과 같다.

들여쓰기

> 들여쓰기를 위해 탭(tabs)을 사용하며, `gofmt`는 기본값으로 탭을 사용한다. 만약 꼭 써야하는 경우에만 스페이스(spaces)를 사용하라.

한 줄 길이

> Go는 한 줄 길이에 제한이 없다. 길이가 길어지는것에 대해 걱정하지 마라. 만약 라인 길이가 너무 길게 느껴진다면, 별도의 탭을 가지고 들여쓰기를하여 감싸라

괄호

> Go는 C와 Java에 비해 적은 수의 괄호가 필요하다. 제어 구조들(`if`, `for`, `switch`)의 문법엔 괄호가 없다. 또한 연산자 우선순위 계층이 간단하며 명확하다. 아래를 보자.

```
x<<8 + y<<16
```
5

다른 언어와는 다르게 스페이스의 사용이 함축하는 바가 크다.

주석

주석(Commentary)

- 원문: Commentary
- 번역자: Jeongbu Yoon (@coma333ryu)

Go언어는 C언어 스타일의 /* */ 블럭주석과 C++스타일의 // 한줄(line) 주석을 제공한다. 한줄주석은 일반적으로 사용되고, 블럭주석은 대부분의 패키지(package)주석에 나타난다. 하지만 표현식 안이나 많은 코드를 주석처리하는데 유용하다.

프로그램 및 웹서버이기도 한 godoc은 패키지의 내용에 대한 문서를 추출하도록 Go 소스 파일을 처리한다. 최상위 선언문 이전에 끼어드는 줄바꿈 없이 주석이 나타나면 그 선언문과 함께 추출되어 해당 항목의 설명으로 제공된다. 이러한 주석의 스타일과 유형은 godoc이 만들어내는 문서의 질을 결정하게 된다.

모든 패키지(package)는 패키지 구문 이전에 블럭주석형태의 패키지 주석이 있어야 한다. 여러 파일로 구성된 패키지의 경우, 패키지 주석은 어느 파일이든 상관없이 하나의 파일에 존재하면 되고 그것이 사용된다. 패키지 주석은 패키지를 소개해야하고, 전체 패키지에 관련된 정보를 제공해야 한다. 패키지 주석은 godoc 문서의 처음에 나타나게 되니 이후의 자세한 사항도 작성해야 한다.

```
/*
Package regexp implements a simple library for regular expressions.

The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

만약 패키지가 단순하다면, 패키지 주석 또한 간단할 수 있다.

```
// Package path implements utility routines for
// manipulating slash-separated filename paths.
```

별로 줄을 그어 쓰는 지나친 포맷은 주석에 필요없다. 심지어 생성된 출력이 고정폭 폰트로 주어지지 않을 수도 있으므로, 스페이스나 정렬등에 의존하지 말라. 그런 것들은 gofmt과 마찬가지로 godoc이 처리한다. 주석은 해석되지 않는 일반 텍스트이다. 그래서 HTML이나 `_this_` 같은 주석은 작성된 그대로 나타날것이다. 그러므로 사용하지 않는것이 좋다. godoc이 수정하는 한 가지는

들여쓰기된 텍스트를 고정폭 폰트로 보여주는 것으로, 프로그램 코드조각 같은 것에 적합하다. fmt package의 패키지주석은 좋은 예이다.

상황에 따라, godoc은 주석을 재변경 하지 않을 수 있다. 그래서 확실하게 보기좋게 만들어야 한다. 정확한 철자, 구두법, 문장구조를 사용하고 긴문장을 줄여야한다.

패키지에서 최상위 선언의 바로 앞에있는 주석이 그 선언의 문서주석으로 처리된다. 패키지 내부에서 최상위 선언 바로 이전의 주석은 그 선언을 위한 문서주석이다. 프로그램에서 모든 외부로 노출되는 (대문자로 시작되는) 이름은 문서주석이 필요하다.

문서 주석은 매우 다양한 자동화된 표현들을 가능케 하는 완전한 문장으로 작성될 때 가장 효과가 좋다. 첫 문장은 선언된 이름으로 시작하는 한 줄짜리 문장으로 요약되어야 한다.

```
// Compile parses a regular expression and returns, if successful,
// a Regexp that can be used to match against text.
func Compile(str string) (*Regexp, error) {
```

만약 모든 문서의 주석이 그 주석이 서술하는 항목의 이름으로 시작한다면, godoc의 결과는 grep문을 이용하는데 유용한 형태로 나오게 될 것이다. 만약 당신이 정규표현식의 파싱 함수를 찾고 있는데 "Compile"이라는 이름을 기억하지 못해 다음의 명령어를 실행했다고 상상해보자,

```
$ godoc regexp | grep parse
```

만약 패키지 내부의 모든 문서주석이 "This function.."으로 시작된다면, grep명령은 당신이 원하는 결과를 보여줄 수 없을것이다. 그러나 패키지는 각각의 문서 주석을 패키지명과 함께 시작하기 때문에, 아래와 같이 당신이 찾고 있던 단어를 상기시키는 결과를 볼 수 있을 것이다.

```
$ godoc regexp | grep parse
    Compile parses a regular expression and returns, if successful, a
    parsed. It simplifies safe initialization of global variables hol
    cannot be parsed. It simplifies safe initialization of global var
$
```

Go언어의 선언구문은 그룹화가 가능하다. 하나의 문서주석은 관련된 상수 또는 변수의 그룹에 대해 설명할 수 있다. 하지만 이러한 주석은 선언 전체에 나타나므로 형식적일 수 있다.

```
// Error codes returned by failures to parse an expression.
var (
    ErrInternal      = errors.New("regexp: internal error")
    ErrUnmatchedLpar = errors.New("regexp: unmatched '('")
    ErrUnmatchedRpar = errors.New("regexp: unmatched ')'")
    ...
)
```

그룹화는 항목 간의 관련성을 나타낼 수 있다. 예를 들어 아래 변수들의 그룹은 mutex에 의해 보호되고 있음을 보여준다.

```
var (
    countLock   sync.Mutex
    inputCount  uint32
    outputCount uint32
    errorCount  uint32
)
```

주석

명칭

명칭

- 원문 : Names
- 번역자 : MinJae Kwon (@mingrammer)

다른 언어들과 마찬가지로 Go에서도 이름은 중요하다. 이는 심지어 의미에 영향을 줄 수도 있다. 이름의 첫 문자가 대문자인지 아닌지에 따라서 이름의 패키지 밖에서의 노출여부가 결정된다. 그러므로 Go의 네이밍 규칙에 대해 살펴볼 필요가 있다.

패키지명

패키지가 임포트되면, 패키지명은 패키지 내용들에 대한 접근자가 된다. 다음과 같이 하면

```
import "bytes"
```

임포트한 패키지는 `bytes.Buffer`를 사용할 수 있다. 패키지를 사용하는 모든 사람들이 패키지 내용을 참조하기위해 같은 이름을 사용할 수 있다는건 패키지명이 잘 작성되어야 함(짧고, 간결하고, 연상하기 쉬운)을 의미한다. 관례적으로, 패키지명은 소문자, 한 단어로만 부여하며 언더바(_)나 대소문자 혼용에 대한 필요가 없어야한다. 이는 간결함에 치우쳐있는데, 패키지를 사용하는 모든 사람들이 패키지명을 직접 타이핑할 것이기 때문이다 (편의성). 그리고 충돌에 대한 걱정은 하지 말라. 패키지명은 오직 임포트를 위한 이름이다. 이는 모든 소스코드에서 유일할 필요는 없으며, 드문 경우지만 임포트를 하는 패키지가 충돌할때엔 국지적으로 다른 이름을 선택할 수 있다. 어느 경우에나, 임포트에 있는 파일명이 사용될 패키지명을 바로 정하기 때문에 혼란이 드물다.

또 다른 규칙은 패키지명은 소스 디렉토리 이름 기반이라는 것이다. `src/encoding/base64`에 있는 패키지는 `encoding/base64`로 임포트가 된다. `base64`라는 이름을 가지고 있지만, `encoding_base64`나 `encodingBase64`를 쓰지 않는다.

패키지를 임포트하는 입장에서는 패키지 내용을 참조하기위해 패키지명을 사용한다, 그래서 패키지 밖으로 노출되는 이름들은 이름의 지저분함을 피하기위해 이러한 사실을 활용할 수 있다. (`import .`표현을 사용하지 말라. 이는 테스트를 수행중인 패키지의 외부에서 필수적으로 실행해야하는 테스트를 단순화 할 수는 있지만, 그렇지 않을 경우엔 피해야한다.) 예를 들면, `bufio` 패키지에 있는 버퍼 리더는 `BufReader`가 아닌 `Reader`로 불린다. 왜냐하면 사용자는 이를 `bufio.Reader`로 보게되며, 이것이 더 명확하고 간결하기 때문이다. 게다가 임포트된 객체들은 항상 패키지명과 함께 불려지기 때문에 `bufio.Reader`는 `io.Reader`와 충돌하지 않는다. 비슷하게, Go에 존재하는 `ring.Ring`이라는 구조체의 인스턴스를 만드는 함수는 보통은 `NewRing`으로 불릴테지만, `Ring`은 패키지 밖으로 노출된 유일한 타입이며, 패키지가 `ring`으로 불리기 때문에, 이 함수는 그냥 `New`라고 부르고 `ring.New`와 같이 사용한다. 좋은 이름을 위해 이러한 패키지 구조를 활용하라.

또 다른 간단한 예시는 `once.Do`이다. `once.Do(setup)`는 읽기가 쉬우며 `once.DoOrWaitUntilDone(setup)`으로 개선될게 없다. 긴 이름은 좀 더 쉽게 읽는것을 방해한다. 문서에 주석을 다는것이 긴 이름을 사용하는 것보다 더 좋을 것이다.

게터 (Getters)

Go는 getters와 setters를 자체적으로 제공하지 않는다. 스스로 getters와 setters를 만들어 사용하면

되는데 이는 전혀 문제될게 없으며 이는 적절하고 일반적인 방법이다. 그러나 getter의 이름에 Get을 넣는건 Go언어 답지도, 필수적이지도 않다. 만약 owner(첫 문자가 소문자이며 패키지 밖으로 노출되지 않는다.)라는 필드를 가지고 있다면 getter 메서드는 GetOwner가 아닌 Owner(첫 문자가 대문자이며, 패키지 밖으로 노출됨)라고 불러야한다. 패키지밖으로 노출하기 위해 대문자 이름을 사용하는 것은 메서드로부터 필드를 식별할 수 있는 훅(hook)을 제공한다. 만약 필요하다면, setter 함수는 SetOwner라고 불릴 것이다. 두 이름 모두 읽기 쉽다.

```
owner := obj.Owner()
if owner != user {
    obj.SetOwner(user)
}
```

인터페이스명

관례적으로, 하나의 메서드를 갖는 인터페이스는 메서드 이름에 `-er` 접미사를 붙이거나 에이전트 명사를 구성하는 유사한 변형에의해 지정된다. 예를 들면, Reader, Writer, Formatter, CloseNotifier 등이 있다.

이런류의 이름들이 있고, 이것들과 이것들을 통해 알 수 있는 함수이름들을 지켜나가는 것은 생산적이다. Read, Write, Close, Flush, String 등등은 각 각의 용법과 의미를 가지고 있다. 혼란을 피하기 위해 같은 용법과 의미를 가지지 않는한 메서드에 이 이름들과 같은 이름을 쓰지 말라. 반대로 말하면, 만약 당신이 구현한 타입에 이미 잘 알려진 타입들이 가지고 있는 메서드들과 같은 의미를 갖는 메서드를 구현하려고 한다면, 같은 이름을 부여하면된다. 문자 변환 메서드를 만든다면 `ToString`이 아닌 `String`을 사용하라.

대소문자 혼합

마지막으로, Go에서의 네이밍 규칙은 여러 단어로된 이름을 명명할 때 언더바(_) 대신 대소문자 혼합(`MixedCaps`나 `mixedCaps`)을 사용하는 것이다.

세미콜론

세미콜론(Semicolons)

- 원문: Semicolons
- 번역자: Jeongbu Yoon (@coma333ryu)

C언어 처럼, Go의 정식문법은 구문을 종료하기 위하여 세미콜론을 사용한다. 하지만 C언어와는 달리 세미콜론은 소스상에 나타나지 않는다. 대신 구문분석기(lexer)는 간단한 규칙을 써서 스캔을 하는 과정에 자동으로 세미콜론을 삽입한다. 그래서 소스작성시 대부분 세미콜론을 사용하지 않는다.

규칙은 다음과 같다. 만약 새로운 라인 앞의 마지막 토큰이 (int나 float64와 같은 단어를 포함한) 식별자이거나, 숫자, 문자열과 같은 기본 리터럴, 혹은 다음의 토큰들중 하나 일 경우에, 구문 분석기(lexer)는 항상 토큰 다음에 세미콜론을 추가한다.

```
break continue fallthrough return ++ -- ) }
```

이것은 "만약 구문을 끝낼 수 있는 토큰뒤에 새로운 라인이 오면, 세미콜론을 삽입하라." 와 같이 요약해서 설명할 수 있다.

세미콜론은 또한 닫는 중괄호(}) 바로 앞에서 생략할 수 있다. 예를들어 아래의 구문은 세미콜론이 필요하지 않는다.

```
go func() { for { dst <- <-src } }()
```

Go 프로그램에서는 세미콜론을 for loop 구문에서 변수 초기화와 조건, 그리고 진행 변수를 구분할 때에만 사용 한다. 또한 세미콜론은 한 라인에서 여러문장을 구분하기 위해 필요하고, 이런 방법으로 코드를 작성해야한다.

세미콜론 입력규칙의 중요한 한가지는 제어문(if, for, switch, 혹은 select)의 여는 중괄호({)를 다음 라인에 사용하지 말아야 한다는 것이다. 만약 그렇게 사용하게 되면, 세미콜론은 중괄호({) 앞에 추가될것이고, 예상하지 못한 영향을 발생시킬 것이다. 다음과 같이 작성하라.

```
if i < f() {
    g()
}
```

다음과 같이 사용하지 말라.

```
if i < f()  // wrong!
{           // wrong!
    g()
}
```

제어구조

제어구조

- 원문 : Control Structures
- 번역자 : Jungsoo Ahn (@findstar)

Go언어의 제어구조는 C 와 연관성이 있지만 중요한 점에서 차이가 있다. Go언어에서는 `do` 나 `while` 반복문이 존재하지 않으며, 단지 좀 더 일반화된 `for`, 좀 더 유연한 `switch`가 존재한다. `if` 와 `switch` 는 선택적으로 `for` 와 같은 초기화 구문을 받을 수 있다. `break` 와 `continue` 구문들은 선택적으로 어느것을 멈추거나 계속할지 식별하기 위해서 라벨을 받을 수 있다. 또한 타입 `switch`와 다방향 통신 멀티플랙서, `select`의 새로운 제어 구조가 포함되어 있다. 문법은 조금 다르다. 괄호는 필요하지 않으며 body는 항상 중괄호로 구분해야 한다.

If

Go언어에서 if문의 간단한 예제는 다음과 같다:

```
if x > 0 {
    return y
}
```

중괄호를 의무적으로 사용해야 하기 때문에, 다중 라인에서 if 구문들이 간단하게 작성된다. 어차피 그렇게 하는 것이 좋은 스타일이며, 특히 구문 몸체에 return이나 break와 같은 제어 구문을 포함하고 있는 경우에는 더욱더 그러하다.

`if`와 `switch`가 초기화 구문을 허용하므로 지역변수를 설정하기 위해 사용된 초기화 구문을 흔히 볼 수 있다.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

Go 여러 라이브러리를 보게되면, if 구문이 다음 구문으로 진행되지 않을 때, 즉 `break`, `continue`, `goto` 또는 `return` 으로 인해서 구문 몸체가 종료될 경우, 불필요한 `else` 는 생략되는 것을 발견할 수 있다.

```
f, err := os.Open(name)
if err != nil {
    return err
}
codeUsing(f)
```

다음은 코드가 일련의 에러 조건들을 반드시 검사해야 하는 일반적인 상황에 대한 예제이다. 만약 제어의 흐름이 성공적이라면 코드는 잘 동작할 것이고, 에러가 발생할 때마다 에러를 제거 할 것이다. 에러 케이스들은 `return` 구문에서 종료하는 경향이 있기 때문에, 결과적으로 코드에는 `else` 구문이 필요하지 않다.

```
f, err := os.Open(name)
```

```
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```

재선언과 재할당

이어서: 이전 섹션의 마지막 예제에서는 어떻게 := 짧은 선언문이 동작하는지 확인할 수 있었다. `os.Open`을 호출하는 선언코드를 보자.

```
f, err := os.Open(name)
```

이 구문은 `f` 와 `err`의 두개의 변수를 선언한다. 몇줄 아래 `f.Stat` 를 호출하는 부분을 보자.

```
d, err := f.Stat()
```

여기서 `d` 와 `err` 를 선언하는 것처럼 보인다. 주목할 부분은 저 `err`가 위에서와 아래 두 곳 모두에서 나타난다는 것이다. 이 선언의 중복은 합법적이다. `err` 은 첫번째 구문을 통해서 선언되었지만 두번째에서는 재할당된다. 이는 `f.Stat` 를 호출하는 것에서는 이미 선언되어 존재하는 `err` 변수를 사용하고, 다시 새로운 값을 부여한다는 것을 의미한다.

변수의 단축선언, `v :=` 에서 변수 v 는 이미 선언되었더라도 다음의 경우 재선언이 가능하다.

- 이 선언이 기존의 선언과 같은 스코프에 있어야 하고 (만약 v가 이미 외부 스코프에 선언되었다면, 이 선언은 새 변수를 만들것 이다. §),
- 초기화 표현내에서 상응하는 값은 v에 할당할 수 있고,
- 적어도 하나 이상의 새로운 변수가 선언문 안에 함께 있어야 한다.

이 독특한 속성은 완전히 실용적이며, 예를 들어 길고 연쇄적인 if-else 구문에서 하나의 에러 값을 쉽게 사용할 수 있게 해 준다. 자주 사용되는 것을 보게 될 것이다.

§ Go언어에서 함수 파라미터와 리턴 값들은, 함수를 감싸고 있는 브래이스들(braces)밖에 위치해 있음에도, 그 스코프는 함수 몸통의 스코프와 동일하다는 점을 주목할 가치가 있다.

For

Go언어에서 for 반복문은 C언어와 비슷하지만 일치하지는 않는다. for 는 while 처럼 동작할 수 있고, 따라서 do-while 이 없다. 다음의 세가지 형태를 확인할 수 있으며, 하나의 경우에서만 세미콜론을 사용되는 것을 확인할 수 있다.

```
// C언어와 같은 경우
for init; condition; post { }

// C언어의 while 처럼 사용
for condition { }

// C언어의 for(;;) 처럼 사용
for { }
```

짧은 선언문은 반복문에서 index 변수 선언을 쉽게 만든다.

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

만약 배열, slice, string, map, 채널로 부터 읽어 들이는 반복문을 작성한다면, range 구문이 이 반복문을 관리해줄 수 있다.

```
for key, value := range oldMap {
    newMap[key] = value
}
```

만약 range 안에서 첫번째 아이템만이 필요하다면 (키 또는 인덱스), 두번째 뒤는 날려버리자:

```
for key := range m {
    if key.expired() {
        delete(m, key)
    }
}
```

만약 range 안에서 두번째 아이템만이 필요하다면 (값), 공백 식별자, 언더스코어를 사용하여 첫번째를 버리도록 하자:

```
sum := 0
for _, value := range array {
    sum += value
}
```

공백 식별자는 많은 사용법이 있는데, 나중에 보일 섹션에 잘 설명되어 있다.

string의 경우, range 는 UTF-8 파싱에 의한 개별적인 유니코드 문자를 처리하는데 유용할 것이다. 잘못된 인코딩은 하나의 바이트를 제거하고 U+FFFD 룬 문자로 대체할 것이다. (룬(내장된 타입으로 지정된)의 이름은 Go 언어의 단일 유니코드 코드에 대한 용어이다. 보다 자세한 사항은 언어 스펙을 참고하자)

다음 반복문은

```
for pos, char := range "日本\x80語" { // \x80 은 합법적인 UTF-8 인코딩이다
    fmt.Printf("character %#U starts at byte position %d\n", char, po
}
```

다음과 같이 출력된다

```
character U+65E5 '日' starts at byte position 0
character U+672C '本' starts at byte position 3
character U+FFFD '' starts at byte position 6
character U+8A9E '語' starts at byte position 7
```

마지막으로 Go언어는 콤마(,) 연산자가 없으며 ++, --는 표현식이 아니라 명령문이다. 따라서 만약 for문 안에서 여러개의 변수를 사용하려면 병렬 할당(parallel assignment)을 사용해야만 한다(++과 --을 배제하더라도).

```
// Reverse a
```

```
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

Switch

Go언어에서 스위치는 C언어에서 보다 더 일반적인 표현이 가능하다. 표현식은 상수이거나 꼭 정수일 필요가 없고 case 구문은 위에서부터 바닥까지 해당 구문이 `true` 가 아닌 동안에 일치하는 값을 찾을 때까지 계속 값을 비교한다. 따라서 `if-else-if-else` 형태로 작성하는 것 대신 `switch` 를 사용하는 것이 가능하다며 더 Go 언어답다.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

스위치에서는 자동으로 다음으로 통과하는 동작이 없지만(케이스 구문을 지나가는 동작), 콤마로 구분된 목록을 사용해 case들을 표현할 수 있다.

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+', '%':
        return true
    }
    return false
}
```

비록 C와 같은 언어들에서 처럼 그렇게 보편적이진 않지만, Go에서도 break 구문으로 switch를 일찍 종료하기 위해 쓸 수 있다. 가끔이지만, 스위치가 아닌 둘러쌓인 반복문을 중단하는 것이 필요하기도 하고, 라벨을 반복문 위에 넣고, 해당 라벨로 "탈출" 하는 것에 의해서 완료되어 질수도 있다. 다음 예제는 이 두가지 사용예이다.

```
Loop:
    for n := 0; n < len(src); n += size {
        switch {
        case src[n] < sizeOne:
            if validateOnly {
                break
            }
            size = 1
            update(src[n])

        case src[n] < sizeTwo:
            if n+1 >= len(src) {
                err = errShortInput
                break Loop
```

```
            }
            if validateOnly {
                break
            }
            size = 2
            update(src[n] + src[n+1]<<shift)
        }
    }
```

물론 `continue` 구문 또한 선택적으로 라벨을 받을 수 있지만, 반복문에만 적용된다.

섹션을 마무리하며, 다음은 두개의 `switch` 구문을 사용하여 바이트 슬라이스를 비교하는 루틴에 대한 예제이다:

```
// Compare returns an integer comparing the two byte slices,
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch {
    case len(a) > len(b):
        return 1
    case len(a) < len(b):
        return -1
    }
    return 0
}
```

타입 switch

스위치 구문은 인터페이스 변수의 동적 타입을 확인하는데 사용될 수도 있다. 이러한 스위치는 타입 단언의 문법을 사용하되 괄호안에 키워드 type을 사용한다. 만약 스위치 표현식 안에서 변수를 선언한다면, 변수는 각각의 절에서 일치하는 타입을 가질 것이다. 사실상 각각의 절 안에서 새로운 변수를 다른 타입이지만 동일한 이름으로 선언하는 것과 각각의 절 안에서 이름을 재사용하는 것이 관례적이다.

```
var t interface{}
t = functionOfSomeType()
switch t := t.(type) {
default:
    fmt.Printf("unexpected type %T\n", t)     // %T prints whatever t
case bool:
    fmt.Printf("boolean %t\n", t)             // t has type bool
case int:
    fmt.Printf("integer %d\n", t)             // t has type int
case *bool:
```

```
    fmt.Printf("pointer to boolean %t\n", *t) // t has type *bool
case *int:
    fmt.Printf("pointer to integer %d\n", *t) // t has type *int
}
```

함수

함수(Functions)

- 원문: Functions
- 번역자: kyungkoo

다중 반환 값(Multiple return values)

Go 언어가 가지고 있는 특징 중 하나는 함수와 메소드가 여러 값을 반환 할 수 있다는 것이다. 이러한 형태는 C 프로그램에서 대역 내 (in-band) 에러에서 EOF를 나타내기 위해 -1 과 같은 값을 반환하고 주소로 전달한 매개변수를 변환시키는 것과 같은 여러 골치아팠던 문법을 개선하는데 사용할 수 있다.

C 언어 에서는 임의의 장소에 비밀스러운 방법으로 에러 코드와 음수로 쓰기 에러를 나타낸다. Go 언어의 Write 에서는 카운트와 에러를 반환 할 수 있다. "그래, 몇 바이트 정도는 쓰긴 했지만 장치에 가득 차기 때문에 모든 바이트를 쓰지는 못했어". os 패키지 파일에 있는 Write 메소드의 시그니처는 다음과 같다.

```
func (file *File) Write(b []byte) (n int, err error)
```

그리고 문서에서도 언급하고 있듯이, Write 메소드는 `n != len(b)`인 경우에는 쓰인 바이트 갯수와 nil 이 아닌 `error` 를 반환한다. 이와 같은 형태는 지극히 일반적이며 더 많은 예제를 보고자 할 경우에는 에러 핸들링 세션을 살펴보도록 하자.

유사하게 반환 값으로 참조 매개변수 흉내를 냄으로써 포인터를 전달 할 필요가 없게 만들 수 있다. 아래는 숫자와 다음 위치를 반환 함으로써 바이트 슬라이스에 위치한 숫자를 가져오는 간단한 함수이다.

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i]) - '0'
    }
    return x, i
}
```

또는 다음과 같이 입력 슬라이스 b 에서 숫자를 스캔하는데도 사용할 수 있다.

```
    for i := 0; i < len(b); {
        x, i = nextInt(b, i)
        fmt.Println(x)
    }
```

이름 있는 결과 인자값 (Named result parameters)

Go 함수에서는 반환 "인자"나 결과 "인자"에 이름을 부여하고 인자로 들어온 매개변수처럼 일반 변수로 사용할 수 있다. 이름을 부여하면, 해당 변수는 함수가 시작될 때 해당 타입의 제로 값으로

초기화 된다. 함수가 인자 없이 반환문을 수행할 경우에는 결과 매개변수의 현재 값이 반환 값으로 사용된다.

이름을 부여하는것이 필수는 아니지만 이름을 부여하면 코드를 더 짧고 명확하게 만들어 주며, 문서화가 된다. `nextInt`의 결과에 이름을 부여할 경우, 반환되는 `int` 가 어떠한 것인지 명확해 진다.

```
func nextInt(b []byte, pos int) (value, nextPos int) {
```

이름있는 결과는 초기화 되고 아무 내용 없이 반환되기 때문에, 명확할 뿐만 아니라 단순해 질 수 있다. 아래는 이를 이용한 `io.ReadFull` 버전이다.

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:]
    }
    return
}
```

Defer

Go 의 `defer` 문은 `defer` 를 실행하는 함수가 반환되기 전에 즉각 함수 호출(연기된 함수)을 실행하도록 예약한다. 이는 일반적인 방법은 아니긴 하지만 함수가 어떤 실행경로를 통해 반환을 하던 간에 자원을 해지 해야만 하는 것과 같은 상황을 처리해야 하는 경우에는 효과적인 방법이다. 가장 대표적인 예제로는 뮤텍스(mutex)의 잠금을 풀거나 파일을 닫는 것이 있다.

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're finished.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed 
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here
}
```

`Close`와 같은 함수의 호출을 지연시키면 두 가지 장점을 얻게 된다. 첫번째로 파일을 닫는 것을

잊어버리는 실수를 하지 않도록 보장해 준다. 함수에 새로운 반환 경로를 추가해야 하는 경우에 흔히 발생하는 실수이다. 두 번째로 open 근처에 close 가 위치하면 함수 맨 끝에 위치하는 것 보다 훨씬 명확한 코드가 되는것을 의미한다.

defer 함수의 매개 변수들(함수가 메서드일 경우는 리시버도 포함되는)은 함수의 호출이 실행될 때가 아닌 defer가 실행될 때 평가된다. 또한 함수가 실행될 때 변수 값이 변하는 것에 대해 걱정할 필요가 없는데, 이는 하나의 defer 호출 위치에서 여러개의 함수 호출을 지연할 수 있음을 의미한다. 여기 다소 유치한 예가 있다.

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

지연된 함수는 LIFO 순서에 의해 실행되므로, 위 코드에서는 함수가 반환되면 4 3 2 1 0 을 출력할 것이다. 좀 더 그럴듯 한 예제로 프로그램을 통해 함수 실행을 추적하기 위한 간단한 방법이 있다. 여기서는 아래와 같이 간단한 추적 루틴을 몇가지 작생 했다:

```
func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

// Use them like this:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}
```

defer 가 실행 될 때 지연된 함수의 매개변수가 평가된다는 사실을 이용하면 더 잘 할 수 있다. 추적 루틴은 아래와 같이 추적을 끝내는 루틴의 매개변수로 설정할 수 있다:

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}

func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}

func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

prints

위 함수는 아래와 같은 결과물을 출력한다.

```
entering: b
in b
entering: a
in a
leaving: a
leaving: b
```

다른 언어로부터 블록 레벨 자원 관리에 익숙한 프로그래머에게는 `defer` 이 생소해 보일지도 모르지만, 가장 흥미로우면서도 강력한 애플리케이션은 분명 블록 기반이 아니라 함수 기반이라는 사실로 부터 온다는 것이다. `panic` 과 `recover` 세션에서는 이러한 가능성에 대한 또 다른 예제를 살펴 볼 것이다.

데이터

데이터

- 원문: Data
- 번역자: Jhonghee Park

new를 사용하는 메모리 할당

Go에는 메모리를 할당하는 두가지 기본 방식이 있는데, 내장(built-in) 함수인 new와 make이다. 서로 다른 일을 하고 다른 타입들에 적용되기 때문에 혼란스러울 수 있지만, 규칙은 간단하다. new부터 살펴보자. 내장 함수로 메모리를 할당하지만 다른 언어에 존재하는 같은 이름의 기능과는 다르게 메모리를 초기화하지 않고, 단지 값을 제로화(zero) 한다. 다시 말하면, new(T)는 타입 T의 새로운 객체에 제로값이 저장된 공간(zeroed storage)을 할당하고 그 객체의 주소인, *T값을 반환한다. Go의 용어로 얘기하자면, 새로 제로값으로 할당된 타입 T를 가리키는 포인터를 반환하는 것이다.

new를 통해 반환된 메모리는 제로값을 갖기 때문에, 굳이 초기화 과정이 없이도 사용된 타입의 제로값을 그대로 쓸 수 있도록 데이터 구조를 설계하면 도움이 된다. 무슨 말이냐 하면 데이터 구조의 사용자가 new로 새 구조체를 만들고 바로 일에 사용할 수 있다는 것이다. 예를 들어, bytes.Buffer의 문서는 "Buffer의 제로값은 바로 사용할 수 있는 텅빈 버퍼이다"라고 말하고 있다. 유사하게, sync.Mutex도 명시된 constructor도 Init 메서드도 가지고 있지 않다. 대신, sync.Mutex의 제로값인 잠기지 않은 mutex로 정의되어 있다.

제로값의 유용함은 전이적인(transitive) 특성이 있다. 다음의 타입 선언을 검토해 보자.

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

SyncedBuffer 타입의 값들은 메모리 할당이나 단순히 선언만 으로도 당장 사용할 준비가 된다. 아래 코드 단편에서는, p와 v가 뒤이은 조정이 없어도 모두 정확히 작동한다.

```
p := new(SyncedBuffer)  // type *SyncedBuffer
var v SyncedBuffer      // type  SyncedBuffer
```

생성자와 합성 리터럴

때로 제로값만으로는 충분치 않고 생성자(constructor)로 초기화해야 할 필요가 생기는데, 아래 예제는 os패키지에서 가지고 온 것이다.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
```

```
    f.nepipe = 0
    return f
}
```

이 예제에는 불필요하게 반복된(boiler plate) 코드들이 많다. 합성 리터럴(composite literal)로 간소화할 수 있는데, 그 표현이 실행될 때마다 새로운 인스턴스를 만들어 낸다.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

C와는 달리, 로컬 변수의 주소를 반환해도 아무 문제가 없음을 주목하라; 변수에 연결된 저장공간은 함수가 반환해도 살아 남는다. 실제로, 합성 리터럴의 주소를 취하는 표현은 매번 실행될 때마다 새로운 인스턴스에 연결된다. 그러므로 마지막 두 줄을 묶어 버릴 수 있다.

```
    return &File{fd, name, nil, 0}
```

합성 리터럴의 필드들은 순서대로 배열되고 반드시 입력해야 한다. 하지만, 요소들에 레이블을 붙여 필드:값 식으로 명시적으로 짝을 만들면, 초기화는 순서에 관계 없이 나타날 수 있다. 입력되지 않은 요소들은 각자에 맞는 제로값을 갖는다. 그러므로 아래와 같이 쓸 수 있다.

```
    return &File{fd: fd, name: name}
```

제한적인 경우로, 만약 합성 리터럴이 전혀 필드를 갖지 않을 때는, 그 타입의 제로값을 생성한다. `new(File)`은 `&File{}`과 동일한 표현이다.

또 합성 리터럴은 arrays, slices, 와 maps를 생성하는데 사용될 수도 있는데, 필드 레이블로 인덱스와 맵의 키를 적절히 사용해야 한다. 아래 예제의 경우, `Enone`, `Eio` 그리고 `Einval`의 값에 상관없이, 서로 다르기만 하면 초기화가 작동한다.

```
a := [...]string   {Enone: "no error", Eio: "Eio", Einval: "invalid a
s := []string      {Enone: "no error", Eio: "Eio", Einval: "invalid a
m := map[int]string{Enone: "no error", Eio: "Eio", Einval: "invalid a
```

make를 사용하는 메모리 할당

다시 메모리 할당으로 돌아가자. 내장 함수인 make(T, args)는 new(T)와 다른 목적의 서비스를 제공한다. slices, maps, 그리고 channels에만 사용하고 (`*T`가 아닌) 타입 T의 (제로값이 아닌) 초기화된 값을 반환한다. 이러한 차이가 있는 이유는 이 세 타입이 내부적으로 반드시 사용 전 초기화 되어야 하는 데이터 구조를 가리키고 있기 때문이다. 예를 들어, slice는 세가지 항목의 기술항으로 (array 내) 데이터를 가리키는 포인터, 크기, 그리고 용량를 가지며, 이 항목들이 초기화되기 전 까지, slice는 `nil`이다. slices, maps, 그리고 channels에 대해, make는 내부 데이터 구조를 초기화하고 사용될 값을 준비한다. 예를 들면,

```
make([]int, 10, 100)
```

메모리에 크기가 100인 int 배열을 할당하고 그 배열의 처음 10개를 가리키는, 크기 10와 용량이 100인 slice 데이터 구조를 생성한다. (slice를 만들때 용량은 생략해도 된다. 자세한 내용은 slice 섹션을 보기 바란다.) 그에 반해, `new([]int)`는 새로 할당되고, 제로값으로 채워진 slice 구조를 가리키는 포인터를 반환하는데, 즉 `nil` slice 값의 포인터인 것이다.

다음 예제들이 new와 make의 차이점을 잘 그리고 있다.

```
var p *[]int = new([]int)       // slice 구조체를 할당한다; *p == nil; 
var v  []int = make([]int, 100) // slice v는 이제 100개의 int를 갖는 배열

// 불필요하게 복잡한 경우:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Go 언어다운 경우:
v := make([]int, 100)
```

make는 maps, slices 그리고 channels에만 적용되며 포인터를 반환하지 않음을 기억하라. 포인터를 얻고 싶으면 new를 사용해서 메모리를 할당하거나 변수의 주소를 명시적으로 취하라.

배열

배열은 메모리의 레이아웃을 상세하게 계획하는데 유용하며 때로는 메모리 할당을 피하는데 도움이 된다. 하지만 주로 다음 섹션의 주제인, slice의 재료로 쓰인다. 배열에 대해 짧게 설명함으로써 slice를 논할 수 있는 초석를 다져 보자.

Go와 C에서는 배열의 작동원리에 큰 차이가 있다. Go에서는,

- 배열은 값이다. 한 배열을 다른 배열에 할당(assign)할 때 모든 요소가 복사된다.
- 특히, 함수에 배열을 패스할 때, 함수는 포인터가 아닌 복사된 배열을 받는다.
- 배열의 크기는 타입의 한 부분이다. 타입 [10]int과 [20]int는 서로 다르다.

배열을 값(value)으로 사용하는 것이 유용할 수도 있지만 또한 비용이 큰 연산이 될 수도 있다; 만약 C와 같은 실행이나 효율성을 원한다면, 아래와 같이 배열 포인터를 보낼 수도 있다.

```
func Sum(a *[3]float64) (sum float64) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float64{7.0, 8.5, 9.1}
x := Sum(&array)  // 명시적인 주소 연산자(&)를 주목하라.
```

하지만 이런 스타일조차 Go언어 답지는 않다. 대신 slice를 사용하라.

Slices

Slice는 배열을 포장하므로써 데이터 시퀀스에 더 일반적이고, 강력하며, 편리한 인터페이스를 제공한다. 변환 메스릭스와 같이 뚜렷한 차원(dimension)을 갖고 있는 항목들을 제외하고는, Go에서 거의 모든 배열 프로그래밍은 단순한 배열보다는 slice를 사용한다.

Slice는 내부의 배열을 가리키는 레퍼런스를 쥐고 있어, 만약에 다른 slice에 할당(assign)되어도, 둘 다 같은 배열을 가리킨다. 함수가 slice를 받아 그 요소에 변화를 주면 호출자도 볼 수 있는데, 이것은 내부의 배열를 가리키는 포인터를 함수에 보내는 것과 유사하다. 그러므로 Read 함수는 포인터와 카운터 대신, slice를 받아 들일 수 있다; slice내 length는 데이터를 읽을 수 있는 최대 한계치에 정해져 있다. 아래에 File타입의 Read 메서드의 시그너처가 있다.

```
func (f *File) Read(buf []byte) (n int, err error)
```

메서드는 읽은 바이트의 수와 생길 수도 있는 에러 값을 반환한다. 아래 예제에서는, 더 큰 버퍼 buf의 첫 32 바이트를 읽어 들이기 위해 그 버퍼를 슬라이스했다(여기에서는 slice를 동사로 썼다).

```
    n, err := f.Read(buf[0:32])
```

그런 슬라이싱은 흔히 사용되고 효율적이다. 사실, 효율성을 잠시 보류하자면, 아래 예제도 역시 버퍼의 첫 32 바이트를 읽는다.

```
    var n int
    var err error
    for i := 0; i < 32; i++ {
        nbytes, e := f.Read(buf[i:i+1])  // Read one byte.
        if nbytes == 0 || e != nil {
            err = e
            break
        }
        n += nbytes
    }
```

slice의 길이는 내부배열의 한계내에서는 얼마든지 바뀔 수 있다; 그냥 slice 차체에 할당(assign)하면 된다. slice의 용량은, 내장함수 `cap`을 통해 얻을 수 있는데, slice가 가질 수 있는 최대 크기를 보고한다. 아래를 보면 slice에 데이터를 부착(append)할 수 있는 함수가 있다. 만약 데이터가 용량을 초과하면, slice의 메모리는 재할당된다. 결과물인 slice는 반환된다. 이 함수는 `len`과 `cap`이 `nil` slice에 합법적으로 적용할 수 있고, 0을 반환하는 사실을 이용하고 있다.

```
func Append(slice, data []byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // 재할당의 경우
        // 미래의 성장폭을 위해, 필요한 양의 두배를 할당하라.
        newSlice := make([]byte, (l+len(data))*2)
        // copy 함수는 사전에 선언되어 있고 어떤 slice 타입에도 사용될 수 있다.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    copy(slice[1:], data)
    return slice
}
```

slice는 꼭 처리후 반환되어야 한다. Append가 `slice`의 요소들을 변경할 수 있지만, slice 자체(포인터, 크기, 용량을 갖고 있는 런타임 데이터 구조)는 값으로 패스되었기 때문이다.

Slice에 부착(append)한다는 생각은 매우 유용하기 때문에 내장함수 append로 만들어져 있다. 이 함수의 설계를 이해하려면 정보가 좀 더 필요함으로 나중에 다시 얘기하기로 하겠다.

이차원 slices

Go의 배열과 slice는 일차원적이다. 이차원의 배열이나 slice와 동일한 것을 만들려면, 배열의 배열 혹은 slice의 slice을 다음과 같이 정의해야 한다:

```
type Transform [3][3]float64  // A 3x3 array, really an array of arra
type LinesOfText [][]byte     // A slice of byte slices.
```

Slice는 크기가 변할 수 있기 때문에, 내부의 slice들은 크기가 각자 다를 수 있다. 다음의 LinesOfText 예와 같이 흔히 일어 날 수 있는 일이다: 각 줄은 독립적으로 다른 크기를 가지고 있다.

```
text := LinesOfText{
    []byte("Now is the time"),
    []byte("for all good gophers"),
    []byte("to bring some fun to the party."),
}
```

때로 이차원의 slice를 메모리에 할당할 필요가 생기는데, 예를 들면 픽셀로 된 줄들은 스캔하는 상황이다. 두 가지 방식으로 해결할 수 있는데, 첫번째는 각 slice를 독립적으로 할당하는 것이고; 두번째는 slice 하나를 할당하고 각 slice로 자른 다음 포인터를 주는 방식이다. 어느 것을 쓸지는 애플리케이션에 달렸다. 만약 slice가 자라거나 줄어들 수 있다면, 독립적으로 할당해서 다음 줄을 덮어쓰는 일을 방지해야 하고; 그렇지 않다면, 객체를 생성하기 위한 메모리 할당을 한번에 하는 것이 더 효율적일 수 있다. 참고로, 여기 두 방식을 스케치해 보았다. 우선, 한번에 한줄씩:

```
// 최상위 레벨의 slice를 할당하라.
picture := make([][]uint8, YSize) // 유닛 y마다 한 줄씩.
// 각 줄을 반복하면서 slice를 할당하라.
for i := range picture {
    picture[i] = make([]uint8, XSize)
}
```

그리고 이제 한번의 메모리 할당으로, 잘라서 줄을 만드는 경우:

```
// 위에서 한 것처럼, 최상위 레벨의 배열을 할당하라.
picture := make([][]uint8, YSize) // 유닛 y마다 한 줄씩.
// 모든 픽셀들을 담을 수 있는 큰 slice를 할당하라.
pixels := make([]uint8, XSize*YSize) // picture는 [][]uint8 타입이지만
// 각 줄을 반복하면서, 남겨진 pixels slice의 처음부터 크기대로 슬라이싱하라.
for i := range picture {
    picture[i], pixels = pixels[:XSize], pixels[XSize:]
}
```

Maps

Map은 편리하고 강력한 내장 데이타 구조로 한 타입의 값들(the *key*)을 다른 타입(*element* 또는 *value*)의 값들에 연결해준다. Key는 equality연산이 정의되어 있는 어떤 타입이라도 사용 가능하며, integers, floating point, 복소수(complex numbers), strings, 포인터(pointers), 인터페이스(equality를 지원하는 동적 타입에 한해서), structs 그리고 배열(arrays)이 그러한 예이다. Slice는 map의 key로 사용될 수 없는데, 그 이유는 equality가 정의되어 있지 않기 때문이다. Slice와 마찬가지로 map 역시 내부 데이터 구조를 가진다. 함수에 map을 입력하고 map의 내용물을 변경하면, 그 변화는 호출자에게도 보인다.

Map 또한 콜론으로 분리된 key-value 짝을 이용한 합성 리터럴로 생성될 수 있으며, 초기화중에 쉽게 만들 수 있다.

```
var timeZone = map[string]int{
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

Map에 값을 할당(assign)하거나 추출(fetch)하는 문법은, 인덱스가 integer일 필요가 없다는 것외에는 배열과 slice과 거의 동일하다.

```
offset := timeZone["EST"]
```

Map에 없는 key를 가지고 값을 추출하려는 시도는 요소 값의 타입에 해당하는 제로값을 반환할 것이다. 예를 들어, 만약 map이 integer를 가지고 있으면, 존재하지 않는 key에 대한 조회는 0을 반환한다. bool 타입의 값을 가진 map으로 set을 구현할 수 있다. map의 엔트리를 `true`로 저장함으로써 set안에 값을 집어 넣을 수 있고, 간단히 인덱싱을 통해 검사해 볼 수 있다.

```
attended := map[string]bool{
    "Ann": true,
    "Joe": true,
    ...
}

if attended[person] { // 만약 person이 맵에 없다면 false일 것이다.
    fmt.Println(person, "was at the meeting")
}
```

때로는 부재값과 제로값을 구분할 필요도 있다. "UTC"에 대한 엔트리 값이 있는건지, 혹은 map이 전혀 아니기 때문에 값이 0은 아닌건지? 그것은 복수 할당(assign)의 형태로 구별할 수 있다.

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

분명한 이유들로 이것을 "comma ok" 관용구라고 부른다. 이 예제에서, 만약 tz가 있다면, seconds는 적절히 세팅될 것이고 ok는 true가 된다; 반면 없다면, seconds는 제로값이 되고 ok는 false가 된다. 여기 보기 좋은 에러 보고를 사용해 만든 함수의 예가 있다.

```
func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Println("unknown time zone:", tz)
    return 0
}
```

실제 값에 상관없이 map내 존재 여부를 검사하려면, 공백 식별자 (_)를 값에 대한 변수가 있어야 할 자리에 놓으면 된다.

```
_, present := timeZone[tz]
```

Map의 엔트리를 제거하기 위해서는, 내장 함수 delete을 쓰는데, map과 제거할 key를 인수로 쓴다. map에 key가 이미 부재하는 경우에도 안전하게 사용할 수 있다.

```
delete(timeZone, "PDT")  // Now on Standard Time
```

출력

Go에서 포맷된 출력은 C의 `printf`와 유사하지만 더 기능이 풍부하고 일반적이다. 함수들은 fmt 패키지에 있고 대문자화된 이름을 가진다: `[fmt.Printf]` `(https://godoc.org/fmt#Printf)`, `[fmt.Fprintf]`

(https://godoc.org/fmt#Fprintf), [fmt.Sprintf](https://godoc.org/fmt#Sprintf), 기타 등등. 문자열 함수(Sprintf, 기타등등)은 제공된 버퍼를 채우기 보다는 문자열을 반환한다.

You don't need to provide a format string. For each of Printf, Fprintf and Sprintf there is another pair of functions, for instance Print and Println. These functions do not take a format string but instead generate a default format for each argument. The Println versions also insert a blank between arguments and append a newline to the output while the Print versions add blanks only if the operand on neither side is a string. In this example each line produces the same output.

반드시 포맷 문자열을 제공할 필요는 없다. Printf, Fprintf 그리고 Sprintf에 대해 짝을 이루는 함수들이 있는데, 예를 들면 Print와 Println이다. 이 함수들은 포맷 문자열을 취하지 않는 대신 입력된 인수에 대해 이미 정해진 포맷을 발생시킨다. Println 버전들은 인수들 사이에 공백을 삽입하고 출력에 줄바꿈을 추가한다. 그런 반면 Print 버전들은 인수 양쪽이 다 문자열이 아닌 경우에만 공백을 삽입한다. 아래 예제에서 각 줄이 같은 출력을 만든다.

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")
fmt.Println("Hello", 23)
fmt.Println(fmt.Sprint("Hello ", 23))
```

포맷된 출력 함수인 fmt.Fprint와 유사 함수들은 첫번째 인수로 io.Writer 인터페이스를 구현한 객체를 취한다; 변수 os.Stdout과 os.Stderr가 친숙한 이러한 객체들의 예이다.

Here things start to diverge from C. First, the numeric formats such as %d do not take flags for signedness or size; instead, the printing routines use the type of the argument to decide these properties.

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

위의 예제는 다음과 같은 출력을 한다.

```
18446744073709551615 ffffffffffffffff; -1 -1
```

정수(integer)를 소수로 바꾸는 예와 같은 기본적인 변환을 원할 경우는, 다목적 용도 포맷인 %v(value라는 의미로)를 사용할 수 있다; 결과는 Print와 Println의 출력과 동일하다. 더우기, 그 포맷은 어떤 값이라도 출력할 수 있으며, 심지어 배열, slice, 그리고 map도 출력한다. 아래에는 이전 섹션에서 정의된 시간대 map을 위한 print문이 있다.

```
fmt.Printf("%v\n", timeZone)  // or just fmt.Println(timeZone)
```

아래와 같이 출력을 제공한다.

```
map[CST:-21600 PST:-28800 EST:-18000 UTC:0 MST:-25200]
```

For maps the keys may be output in any order, of course. When printing a struct, the modified format %+v annotates the fields of the structure with their names, and for any value the alternate format %#v prints the value in full Go syntax.

물론, map의 경우 key들은 무작위로 출력될 수 있다. struct를 출력할 때는, 수정된 포맷인 %+v를 통해 구조체의 필드에 주석으로 이름을 달며, 대안 포맷인 %#v를 사용하면 어떤 값이든 완전한 Go 문법을 출력한다.

```
type T struct {
```

```
    a int
    b float64
    c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", timeZone)
```

위의 예제는 다음과 같이 출력된다.

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] int{"CST":-21600, "PST":-28800, "EST":-18000, "UTC":0, "
```

(엠퍼센트에 주목하라.) 인용 문자열 포맷은 %q를 이용해 string 타입이나 []byte 타입 값에 적용했을때 얻어진다. 대안 포맷인 %#q는 가능한 경우 backquote을 사용한다. (%q 포맷은 또 integer와 rune에 적용되어, single-quoted rune 상수를 만든다.) %x는 문자열, byte 배열, 그리고 byte slice와 integer에 작용하여 긴 16진수 문자열을 생성하는데, (% x) 포맷처럼 스페이스를 중간에 넣으면 (출력하는) byte사이에 공백을 넣어 준다.

또 다른 유용한 포맷은 %T로, 값의 타입을 출력한다.

```
fmt.Printf("%T\n", timeZone)
```

위의 예제는 다음과 같이 출력된다.

```
map[string] int
```

커스텀 타입의 기본 포맷을 조종하기 위해 해야할 것은 단지 `String() string`의 시그너처를 갖는 메서드를 정의해 주는 것이다. (위에 정의된) 단순한 타입 T는 아래와 같은 포맷을 가질 수 있다.

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

다음과 같은 포맷으로 출력된다.

```
7/-2.35/"abc\tdef"
```

(만약 타입 T와 동시에 포인터 타입 T도 함께 출력할 필요가 있으면, `String`의 리시버는 값 타입이러야 한다; 위에 예제에서 struct 타입에 포인터를 사용한 이유는 더 효율적이고 Go 언어다운 선택이기 때문이다. 더 상세한 정보는 다음의 링크를 참고하라: [pointers vs. value receivers])

String 메서드가 Sprintf를 호출할 수 있는 이유는 print 루틴들의 재진입(reentrant)이 충분히 가능하고 예제와 같이 감싸도 되기 때문이다. 하지만 이 방식에 대해 한가지 이해하고 넘어가야 하는 매우 중요한 디테일이 있는데: String 매서드를 만들면서 Sprintf를 호출할 때 다시 String 매서드로 영구히 재귀하는 방식은 안 된다는 것이다. Sprintf가 리시버를 string처럼 직접 출력하는 경우에 이런 일이 발생할 수 있는데, 그렇게 되면 다시 같은 메서드를 호출하게 되고 말 것이다. 흔하고 쉽게 하는 실수로, 다음의 예제에서 살펴보자.

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", m) // 에러: 영원히 재귀할 것임.
}
```

이러한 실수는 또 쉽게 고칠 수도 있다: 인수를 기본적인 문자열 타입으로 변환하면, 같은 메서드가 없기 때문이다.

```
type MyString string
func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", string(m)) // OK: note convers
}
```

initialization section 섹션에 가면 이 같은 재귀호출을 피할 수 있는 다른 테크닉을 보게 될 것이다.

또 다른 출력 기법으로는 출력 루틴의 인수들을 직접 또 다른 유사한 루틴으로 대입하는 것이다. `Printf`의 시그너처는 마지막 인수로 임의적인 숫자의 파라미터가 포맷 다음에 나타날 수 있음을 명시하기 위해 타입 `...interface{}`를 사용한다.

```
func Printf(format string, v ...interface{}) (n int, err error) {
```

`Printf` 함수내에, v는 `[]interface{}` 타입의 변수 처럼 행동하지만 만약에 다른 가변 인수 함수(variadic function)에 대입되면, 보통의 인수리스트처럼 동작한다. 위에서 사용한 `log.Println`의 구현이 아래에 있다. 실제로 포맷팅을 위해 `fmt.Sprintln` 함수에 직접 인수들을 대입하고 있다.

```
// Println 함수는 fmt.Println처럼 표준 로거에 출력한다.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))  // Output 함수는 (int, string)
}
```

`Sprintln`을 부르는 중첩된 호출안에 v 다음에 오는 ...는 컴파일러에게 v를 인수 리스트로 취급하라고 말하는 것이고; 그렇지 않은 경우는 v를 하나의 slice 인수로 대입한다.

여기에서 살펴 본 출력에 관한 내용보다 훨씬 많은 정보들이 있다. `godoc`에 있는 fmt 패키지를 통해 상세하게 알아보라.

그나저나, ... 파라미터는 특정한 타입을 가질 수도 있는데, 예로 integer 리스트에서 최소값을 선택하는 함수인 min에 대한 `...int`를 살려보라:

```
func Min(a ...int) int {
    min := int(^uint(0) &gt;&gt; 1)  // largest int
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Append

이제 내장함수 append의 설계를 설명하는데 필요했지만 누락된 부분을 갖게 되었다. append의 시

그너처는 위에서 만들어 본 Append 함수와 다르다. 도식적으로, 아래와 같다:

```
func append(slice []T, elements ...T) []T
```

여기서 T는 어떤 타입의 플레이스 홀더이다. 실제로 Go 언어에서는 호출자에 의해 결정되는 타입 T를 쓰는 함수를 만들 수 없다. 그래서 append는 내장함수 인 것이다: 컴파일러의 지원이 필요한 것이다.

append이 하는 일은 slice의 끝에 요소들을 붙이고 결과를 반환하는 것이다. 결과는 반환되어야 한다. 왜냐면, 손으로 쓴 Append와 같이, 내부의 배열은 변할 수 있다. 다음의 간단한 예를 보라.

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

이 예제는 [1 2 3 4 5 6]을 출력한다. append가 `Printf`처럼 임의적인 숫자의 인수들을 모으며 작동하는 것이다.

그런데 만약 Append와 같이 slice에 slice를 붙이고 싶다면 어떻게 해야 하는가? 쉬운 방법으로, 위에서 Output을 호출하면서 그랬듯이, 호출 지점에서 ...를 이용하는 것이다. 아래 예재 단편은 위와 동일한 결과를 산출한다.

```
x := []int{1,2,3}
y := []int{4,5,6}
x = append(x, y...)
fmt.Println(x)
```

...이 없다면 컴파일되지 않는다. 왜냐하면 타입이 틀리기 때문이다: y는 int 타입이 아니다.

초기화

초기화

- 원문 : initialization
- 번역자 : Joseph Lee (@PrayForWisdom)

C나 C++의 초기화와 겉으로는 많이 다르게 보이지 않지만, Go의 초기화는 좀 더 강력하다. 초기화되는 동안 복잡한 구조체들이 생성될 수 있고, 초기화되는 객체들간의 순서를 정하는 문제도 정확히 처리된다. 이는 심지어 다른 패키지 사이에서도 올바르게 작동한다.

상수(Constants)

Go에서 상수는 우리가 일반적으로 생각하는 상수이다. 상수는 -함수 내에서 지역적으로 정의된 상수조차도- 컴파일할 때 생성되며, 숫자(number), 문자(rune), 문자열(string), 참/거짓(boolean) 중의 하나가 되어야 한다. 상수를 정의하는 표현식은 컴파일러가 컴파일 시점에 실행 가능한 상수 표현식이어야 한다. 예를 들어 `1<<3`은 상수 표현식이지만 `math.Sin(math.Pi/4)`는 상수 표현식이 아니다. math 패키지의 Sin 함수에 대한 호출이 런타임 시에만 가능하기 때문이다.

Go에서는 열거형(enum, 자동 증가 상수형)상수를 iota라는 열거자(enumerator)를 이용해서 생성한다. iota는 표현식의 일부로 묵시적으로 반복될 수 있다. 이는 복잡한 값들로 구성된 집합들을 만들기 쉽게 해준다.

```
type ByteSize float64

const (
    _           = iota // 공백 식별자를 이용해서 값인 0을 무시
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

Go는 String과 같은 메서드를 유저가 정의한 자료형(type)에 붙일 수 있다. 이는 출력을 위해 값의 형태를 자동으로 바꾸게 할 수 있다는 것을 의미한다. 이 같은 메서드 결합은 일반적으로 구조체에 적용되지만, 이 테크닉은 ByteSize와 같은 실수 형태의 스칼라 타입에서도 또한 유용하다.

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
        return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
```

```
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
    return fmt.Sprintf("%.2fB", b)
}
```

위 예제에서 ByteSize 변수에 YB를 넣으면 1.00YB로 출력되며, 1e13(=10,000,000,000,000)을 넣으면 9.09TB로 출력된다.

여기에서 ByteSize의 String 메서드에 적용한 Sprintf의 사용은 반복적으로 재귀호출되는 것을 피했기 때문에 안전하다. 형변환 때문이 아니라, Sprintf를 string 형태가 아닌 %f 옵션으로 호출했기 때문이다. (Sprintf는 단지 문자열이 필요할 때만 String 메서드를 호출하고, %f는 실수값(floating-point value)을 사용한다.)

변수(Variables)

변수의 초기화는 상수와 같은 방식이지만, 초기화는 런타임에 계산되는 일반적인 표현식이어도 된다.

```
var (
    home   = os.Getenv("HOME")
    user   = os.Getenv("USER")
    gopath = os.Getenv("GOPATH")
)
```

init 함수(The init function)

최종적으로, 각 소스파일은 필요한 어떤 상태든지 셋업하기 위해서 각자의 init 함수를 정의할 수 있다. (init 함수는 매개변수를 가지지 않으며, 각 파일은 여러 개의 init 함수를 가질 수 있다.)여기서 "최종적으로" 라는 말은 정말로 마지막을 가리킨다: init 함수는 모든 임포트된 패키지들이 초기화되고 패키지 내의 모든 변수 선언이 평가된 이후에 호출된다.

선언의 형태로 표현할 수 없는 것들을 초기화하는 것 외에도, init 함수는 실제 프로그램의 실행이 일어나기 전에 프로그램의 상태를 검증하고 올바르게 복구하는데 자주 사용된다.

```
func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath may be overridden by --gopath flag on command line.
```

초기화

```
    flag.StringVar(&gopath, "gopath", gopath, "override default GOPA
}
```

메서드

메서드

- 원문 : Methods
- 번역자 : MinJae Kwon (@mingrammer)

포인터 vs. 값

ByteSize에서 보듯이, 메서드는 포인터와 인터페이스를 제외한 모든 타입에 대해 정의가 가능하다. 리시버는 꼭 구조체일 필요는 없다.

위의 슬라이스에 대한 논의에서 우리는 Append함수를 작성했었다. 우리는 이를 슬라이스의 메서드로서 재정의할 수 있다. 이를 위해 이 메서드를 바인딩할 타입을 하나 선언하자. 그리고 메서드에서 타입의 값을 받기위한 리시버를 만들자.

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // 함수 내용은 위에서 정의된 Append함수와 정확히 동일하다.
}
```

이 함수는 여전히 갱신된 슬라이스를 반환해야할 필요가 있다. 우리는 이러한 번거로움을 메서드가 ByteSlice에 대한 포인터를 리시버로서 받을 수 있게 재정의함으로써 없앨 수 있고, 이 메서드는 메서드를 호출한 슬라이스를 덮어쓸 수 있다.

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // 함수 내용은 위와 같지만, return이 없다.
    *p = slice
}
```

사실, 이보다 더 나은 함수를 작성할 수도 있다. 위의 함수를 다음과 같이 표준 Write메서드 처럼 작성한다면

```
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // 내용은 위와 같다.
    *p = slice
    return len(data), nil
}
```

타입 *ByteSlice는 표준 인터페이스 io.Writer를 따르게되며, 다루기가 편해진다. 예를 들면, 다음처럼 ByteSlice에 값을 넣을 수 있다.

```
var b ByteSlice
fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

ByteSlice의 주소만 넘긴 이유는, 오직 포인터 타입인 *ByteSlice만이 io.Writer 인터페이스를 만족시키기 때문이다. 리시버로 포인터를 쓸 것인가 값을 쓸 것인가에 대한 규칙은 값을 사

용하는 메서드는 포인터와 값에서 모두 사용할 수 있으며, 포인터 메서드의 경우 포인터에서만 사용이 가능하다는 것이다.

이러한 규칙은 포인터 메서드는 리시버를 변형시킬 수 있는데 메서드를 값에서 호출하게 되면 값의 복사본을 받기 때문에 원래값을 변형할 수 없기 때문에 생겨났다. Go언어는 이러한 실수(값에서 포인터 메서드를 실행하는 일)를 허용하지 않는다. 하지만 편리한 예외도 있다. 주소를 얻을 수 있는 값의 경우에, Go언어는 포인터 메서드를 값 위에서 실행할 경우 자동으로 주소 연산을 넣어준다. 위의 예시에서, 변수 `b`는 주소로 접근이 가능하기 때문에 단순히 `b.Write`만으로 `Write`메서드를 호출할 수 있다. 컴파일러는 이것을 `(&b).Write`로 재작성할 것이다.

부연적으로, 바이트 슬라이스에 `Write`를 사용하는 아이디어는 `bytes.Buffer` 구현의 핵심이기도 하다.

인터페이스와 다른 타입들

인터페이스와 다른 타입들

- 원문: Interfaces and other types
- 번역자: Jhonghee Park

인터페이스

Go언어의 인터페이스는 객체의 행위(behavior)를 지정해 주는 하나의 방법이다: 만약 어떤 객체가 정해진 행동를 할 수 있다면 호환되는 타입으로 쓸 수 있다는 뜻이다. 이미 간단한 몇몇 예제들을 본 적이 있다; String 메서드를 구현하면 개체의 사용자 정의 출력이 가능하고, Fprintf의 출력으로 Write 메서드를 가지고 있는 어떤 객체라도 쓸 수 있다. Go 코드에서는 한 두개의 메서드를 지정해 주는 인터페이스가 보편적이며, 인터페이스의 이름(명사)은 보통 메서드(동사)에서 파생된다: Write 메서드를 구현하면 io.Writer가 인터페이스의 이름이 되는 경우.

타입은 복수의 인터페이스를 구현할 수 있다. sort.Interface를 구현하고 있는 collection의 예를 들어 보자. sort.Interface는 Len(), Less(i, j int) bool, 그리고 Swap(i, j int)를 지정하고 있고 이런 인테페이스를 구현한다면 sort 패키지내 IsSorted, Sort, Stable 같은 루틴을 사용할 수 있다. 또한 사용자 지정의 포멧터를 구현할 수도 있다. 다음의 예제에서 Sequence는 이러한 두개의 인터페이스를 충족시키고 있다.

```
type Sequence []int

// sort.Interface를 위한 필수적인 메서드들.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// 프린팅에 필요한 메서드 - 프린트하기 전에 요소들을 정렬함.
func (s Sequence) String() string {
    sort.Sort(s)
    str := "["
    for i, elem := range s {
        if i > 0 {
            str += " "
        }
        str += fmt.Sprint(elem)
    }
    return str + "]"
}
```

타입 변환

Sequence의 String 메서드는 Sprint가 이미 슬라이스(slices)를 가지고 하는 일을 반복하고 있다. 하지만 Sprint를 실행하기 전에 Sequence를 []int로 변환하면 일을 줄일 수 있다.

```
func (s Sequence) String() string {
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

이 같은 방법은 String 메서드에서 Sprint를 안전하게 실행할 수 있는 타입 변환 기법의 또 다른 예이다. 이것이 가능한 이유는 Sequence와 []int 두 타입이 이름만 무시하면 동일하기 때문에 합법적으로 서로 변환할 수 있는 것이다. 이러한 타입 변환은 새로운 값을 만들어 내지 않고 현재 값에 새로운 타입이 있는 것 처럼 임시로 행동하게 한다. (새로운 값을 만드는 다른 합법적 변환도 있다. 예를 들면 integer에서 floating point로의 변환)

Go 프로그램에서 일군의 다른 메서드를 사용하기 위해 타입을 변환하는 것은 Go 언어다운 표현이다. 예를 들면, sort.IntSlice를 사용해 위의 프로그램 전체를 다음과 같이 간소화 시킬 수 있다.

```
type Sequence []int

// 프린팅에 필요한 메서드 - 프린트하기 전에 요소들을 정렬함.
func (s Sequence) String() string {
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

보시라! Sequence가 복수의(정렬과 출력) 인터페이스를 구현하는 대신, 하나의 데이터 아이템이 복수의 타입(Sequence, sort.IntSlice, 그리고 []int)으로 변환될 수 있는 점을 이용하고 있다. 각 타입은 주어진 작업의 일정 부분을 감당하게 된다. 실전에서 자주 쓰이진 않지만 매우 효과적일 수 있다.

인터페이스 변환과 타입 단언

타입 스위치는 변환의 한 형태이다: 인터페이스를 받았을 때, switch문의 각 case에 맞게 타입 변환을 한다. 아래 예제는 fmt.Printf가 타입 스위치를 써서 어떻게 주어진 값을 string으로 변환시키는지를 단순화된 버전으로 보여 주고 있다. 만약에 값이 이미 string인 경우는 인터페이스가 잡고 있는 실제 string 값을 원하고, 그렇지 않고 값이 String 메서드를 가지고 있을 경우는 메서드를 실행한 결과를 원한다.

```
type Stringer interface {
    String() string
}

var value interface{} // caller가 제공한 값.
switch str := value.(type) {
case string:
    return str
case Stringer:
    return str.String()
}
```

첫번째 case는 구체적인 값을 찾은 경우이고; 두번째 case는 인터페이스를 또 다른 인터페이스로 변환한 경우이다. 이런 식으로 여러 타입을 섞어서 써도 아무 문제가 없다.

오로지 한 타입만에만 관심이 있는 경우는 어떨까? 만약 주어진 값이 string을 저장하는 걸 알고 있고 그냥 그 string 값을 추출하고자 한다면? 단 하나의 case만을 갖는 타입 스위치면 해결 할 수 있지

만 타입 단언 표현을 쓸 수도 있다. 타입 단언은 인테페이스 값을 가지고 지정된 명확한 타입의 값을 추출한다. 문법은 타입 스위치를 열 때와 비슷하지만 type 키워드 대신 명확한 타입을 사용한다:

```
value.(typeName)
```

그리고 그 결과로 얻는 새 값은 typeName이라는 정적 타입이다. 그 타입은 인터페이스가 잡고 있는 구체적인 타입이던지 아니면 그 값이 변환 될 수 있는 2번째 인터페이스 타입이어야 한다. 주어진 값 안에 있는 string을 추출하기 위해서, 다음과 같이 쓸 수 있다:

```
str := value.(string)
```

하지만 그 값이 string을 가지고 있지 않을 경우, 프로그램은 런타임 에러를 내고 죽는다. 이런 참사에 대비하기 위해서, "comma, ok" 관용구를 사용하여 안전하게 값이 string인지 검사 해야 한다:

```
str, ok := value.(string)
if ok {
    fmt.Printf("string value is: %q\n", str)
} else {
    fmt.Printf("value is not a string\n")
}
```

만약 타입 단언이 검사에서 실패할 경우, str는 여전히 string 타입으로 존재하고, string의 제로값인 빈 문자열을 가지게 된다.

가능한 예를 또 들자면, 위에서 보여준 타입 스위치와 동일한 기능을 하는 if-else문이 여기 있다.

```
if str, ok := value.(string); ok {
    return str
} else if str, ok := value.(Stringer); ok {
    return str.String()
}
```

일반성

만약 어떤 타입이 오로지 인터페이스를 구현하기 위해서만 존재한다면, 즉 인터페이스외 어떤 메서드도 외부에 노출시키지 않은 경우, 타입 자체를 노출 시킬 필요가 없다. 단지 인터페이스만을 노출하는 것은 주어진 값이 인테페이스에 묘사된 행위들 외 어떤 흥미로운 기능도 있지 않다는 것을 확실하게 전달한다. 이는 또한 공통된 메서드에 대한 문서화의 반복을 피할 수 있다.

그런 경우에, constructor는 구현 타입보다는 인터페이스 값을 반환해야 한다. 예를 들어, 해쉬 라이브러리인 crc32.NewIEEE 와 adler32.New는 둘 다 인터페이스 타입 hash.Hash32를 반환한다. Go 프로그램에서 CRC-32 알로리즘을 Adler-32로 교체하는데 요구되는 사항은 단순히 constructor 콜을 바꿔주는 것이다; 그 외 코드들은 알고리즘의 변화에 아무런 영향을 받지 않는다.

이와 유사한 방식을 통해서, 각종 crypto 패키지내의 스트리밍 cipher 알고리즘들을, 이들이 연결해 쓰는 block cipher들로 부터 분리시킬 수 있다. crypto/cipher 패키지내 Block 인터페이스는 한 block의 데이터를 암호화하는 block cipher의 행위를 정의한다. 그런 다음, bufio 패키지에서 유추해 볼 수 있듯이, Block 인터페이스를 구현하는 cipher 패키지들은, Stream 인터페이스로 대표되는 스트리밍 cipher들을 건설할 때, block 암호화의 자세한 내용을 알지 못하더라도, 사용될 수 있다.

crypto/cipher 인터페이스들은 다음과 같다:

```
type Block interface {
    BlockSize() int
```

```
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}

type Stream interface {
    XORKeyStream(dst, src []byte)
}
```

여기 block cipher를 스트리밍 cipher로 바꾸어 주는 카운터 모드 (CTR) 스트림의 정의가 있다; block cipher의 자세한 내용이 추상화되어 있는 점을 유의하라:

```
// NewCTR은 카운더 모드로 주어진 Block을 이용하여 암호화하고/해독하는 스트림을 반
// iv의 길이는 Block의 block 크기와 같아야 한다.
func NewCTR(block Block, iv []byte) Stream
```

NewCTR은 특정한 암호화 알고리즘과 데이터 소스에만 적용되는 것이 아니라 Block와 Stream 인터페이스를 구현하는 어떤 알고리즘이나 데이터 소스에도 적용이 가능하다. 왜냐하면 인터페시스 값들을 반환하고, CTR 암호화를 다른 암호화 모드로 교체하는 것이 국부적인 변화이기 때문이다. constructor 콜은 반드시 편집되어야 합니다. 하지만 둘러싸고 있는 코드는 반환 결과를 Stream으로 처리해야 하기 때문에, 차이를 알지 못 한다.

인터페이스와 메서드

거의 모든 것에 메서드를 첨부할 수 있다는 말은 거의 모든 것이 인터페이스를 만족 시킬 수 있다는 말이기도 하다. 한 회화적인 예가 http 패키지내 정의되어 있는 Handler 인터페이스 이다. Handler를 구현하는 어떤 객체도 HTTP request에 서비스를 제공할 수 있다.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

ResponseWriter 역시 클라이어트에 응답을 반환하는데 필요한 메서드들의 접근을 제공하는 인터페이스이다. 이 메서드들은 표준 Write 메서드를 포함하여서, http.ResponseWriter는 io.Writer가 사용될 수 있는 곳이면 어디든 사용할 수 있다. Request는 클라이언트로 부터 오는 request의 분석된 내용을 담은 struct이다.

간결함을 위해, POST는 무시하고 항상 HTTP request가 GET라고 가정하자; 이런 단순화는 handler들이 셑업되는 방식에 영향을 미치지 않는다. 여기 사소한 예이긴 하지만 페이지 방문 수를 세는 hander의 완전한 구현이 있다.

```
// 단순한 카운터 서버.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Reque
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}
```

(지금까지 얘기와 일맥상통하는 예로 Fprintf가 http.ResponseWriter에 출력할 수 있음을 주목하라.) 참고로, 여기 URL에 그런 서버를 어떻게 부착하는 예가 있다.

```
import "net/http"
```

```
...
ctr := new(Counter)
http.Handle("/counter", ctr)
```

그런데 굳이 Counter를 struct으로 만들 이유가 있을까? integer면 충분하다. (caller에게 값의 증가를 보이기 위해 리시버는 포인터일 필요가 있다.)

```
// 단순한 카운터 서버.
type Counter int

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Reques
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

만약에 여러분의 프로그램 내부 상태가 페이지를 방문을 알아야 할 경우라면 어떨까? 웹 페이지에 채널을 묶으라.

```
// 채널이 매 방문마다 알린다.
// (아마 이 채널에는 버퍼를 사용해야 할 것이다.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "notification sent")
}
```

마지막으로, 서버를 구동할 때 사용한 명령줄 인수들을 /args에 보여주려는 경우를 상상해 보자. 명령줄 인수를 출력하는 함수를 쓰는 것은 간단하다.

```
func ArgServer() {
    fmt.Println(os.Args)
}
```

이것을 어떻게 HTTP 서버로 바꿀 수 있을까? 어떤 타입에다가 값은 무시하면서 ArgServer를 메서드로 만들 수 있을 것이다. 하지만 더 좋은 방법이 있다. 포인터와 인터페이스만 빼고는 어떤 타입에도 메서드를 정의할 수 있는 사실을 이용해서, 함수에 메서드를 쓸 수 있다. http 패키지에 다음과 같은 코드가 있다:

```
// HandlerFunc는 어뎁터로써 평범한 함수를 HTTP handler로 쓸 수 있게 해 준다.
// 만약에 f가 적절한 함수 signature를 가지면,
// HandlerFunc(f)는 f를 부르는 Handler 객체인 것이다.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, req).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, req *Request) {
    f(w, req)
}
```

HandlerFunc는 ServeHTTP라는 매쏘드를 같는 타입으로, 이 타입의 값은 HTTP request에 서비스를 제공한다. 메서드의 구현을 한번 살펴 보라: 리시버는 함수, f이고 메서드가 f를 부른다. 이상해 보일 수도 있지만, 리시버가 채널이고 메서드가 채널에 데이터를 보내는 예와 비교해도 크게 다르지 않다.

ArgServer를 HTTP 서버로 만들기 위해서, 우선 적절한 signature를 갖도록 고쳐야 한다.

```
// Argument server.
func ArgServer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, os.Args)
}
```

ArgServer는 이제 HandlerFunc와 signature가 동일 하다. 마치 IntSlice.Sort 메서드를 쓰기 위해 Sequence를 IntSlice로 변환 했듯이, ServeHTTP를 쓰기 위해 ArgServer를 HandlerFunc로 변환 시킬 수 있다. 셑업을 하는 코드는 매우 간결하다:

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

누가 /args를 방문했을 때, 그 페이지에 설치된 handler는 ArgServer 값을 갖는 HandlerFunc타입 이다. HTTP 서버는 그 타입의 ServeHTTP 메서드를 부르면서 ArgServer를 리시버로 사용하고, 결국 ArgServer를 부르게 된다: HandlerFunc.ServeHTTP안에서 f(w, req)를 부르게 된다. 그리고 나면 명령줄 인수가 나타나 보인다.

지금까지 struct, integer, channel, 그리고 함수(function)을 가지고 HTTP 서버를 만들어 보았다. 이것이 가능한 이유는 인터페이스가 거의 모든 타입에 정의 할 수 있는 단순한 메서드의 집합이기 때문이다.

공백 식별자

공백 식별자 (The blank identifier)

- 원문 : The blank identifier
- 번역자 : MinJae Kwon (@mingrammer)

우리는 이미 `for range` 루프와 `maps`를 설명하면서, 공백 식별자에 대해 두 차례 언급했었다. 공백 식별자는 아무런 피해없이 그 어떤 타입의 그 어떤 값에 대해서도 할당 또는 선언될 수 있다. 이건 마치 Unix에서 `/dev/null` 파일(필요는 하지만 실제 값은 아무런 관련이 없는 변수를 저장하는 용도로서 사용되는 쓰기 전용 값)에 값을 넘기는 것과 유사하다.

다중 할당에서의 공백 식별자

`for range`루프에서 공백 식별자의 사용은 일반적인 상황에서의 특별한 경우(다중 할당)이다.

만약 좌변에 여러개의 값을 할당해야 하는데, 그 중 하나가 사용되지 않을 경우, 좌변에 공백 식별자를 두면 더미 변수를 생성 해야하는 필요가 없어지고 값 버리기를 깔끔하게 처리할 수 있다. 예를 들면, 하나의 값과 에러를 리턴하는 함수를 호출하는데 오직 에러만이 중요하다면, 무관한 값을 버리기 위해 공백 식별자를 사용한다.

```
if _, err := os.Stat(path); os.IsNotExist(err) {
    fmt.Printf("%s does not exist\n", path)
}
```

가끔 에러를 무시하기 위해 에러값을 버리는 코드를 볼 수도 있다. 이건 매우 나쁜 관행이다. 에러 반환을 항상 확인하라. 에러가 발생하는 데는 이유가 있다.

```
// Bad! This code will crash if path does not exist.
fi, _ := os.Stat(path)
if fi.IsDir() {
    fmt.Printf("%s is a directory\n", path)
}
```

미사용 임포트와 변수

패키지를 임포트하거나 변수를 선언해놓고 쓰지않게되면 에러가 발생한다. 미사용 임포트는 프로그램의 크기를 부풀리며 컴파일 속도도 저하시킨다. 또 사용되진 않지만 초기화된 변수는 적어도 연산을 낭비하며 어쩌면 큰 버그를 암시할 수도 있다. 그러나 프로그램이 개발중에 있을때 미사용 임포트와 변수들이 종종 생겨나게 될테고, 단지 컴파일이 진행되게 하기 위해서 나중에 다시 필요해질 이들을 지우는건 귀찮을 수 있다. 공백 식별자는 이를 피하는 방법을 제공한다.

아래의 반만 완성된 프로그램은 두 개의 미사용 임포트(`fmt`, `io`)와 미사용 변수(`fd`)를 가지고 있다. 따라서 이는 컴파일되지 않을 것이다. 하지만 지금까지 코드가 정확하게 만들어졌는지를 알 수 있다면 좋을 것이다.

```
package main

import (
    "fmt"
```

```
    "io"
    "log"
    "os"
)

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
}
```

미사용 임포트에 대한 불평을 잠재우려면, 임포트된 패키지의 상징을 참조하는 공백 식별자를 써라. 이와 유사하게, 미사용 변수 `fd`를 공백 식별자에 할당하면 미사용 변수에 대한 에러를 잠재울 수 있을 것이다. 이 버전의 프로그램은 컴파일이 된다.

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

var _ = fmt.Printf // For debugging; delete when done.
var _ io.Reader    // For debugging; delete when done.

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
    _ = fd
}
```

규약에 의하면, 임포트 에러를 잠재우기 위한 전역 선언은 임포트 구문 바로 다음에 위치되게 하며, 주석을 달아줘야 한다. 이들은 나중에 코드를 정리해야함을 쉽게 상기시키고 쉽게 찾아주게 만든다.

부수효과(side effect)를 위한 임포트

이전 예시에서 `fmt`나 `io`와 같은 미사용 임포트는 결국 사용되야 하거나 그렇지 않을 경우엔 없애야한다. (공백 할당은 아직 작업이 진행중인 코드로 인식해야 한다.) 그러나 때로는 직접 사용하지는 않으면서, 부수효과를 위해 패키지를 임포트하기도 하는데, 이는 유용한 사례이다. 예를 들면,`net/http/pprof`패키지는 패키지의 초기화 함수를 실행하는 동안 디버깅 정보를 제공하는 HTTP 핸들러를 등록한다. 이는 노출된 API를 가지고 있지만 대다수의 클라어언트는 오직 핸들러 등록만이 필요하고 정보에는 웹페이지를 통해 접근한다. 부수효과만을 위해 이 패키지를 임포트하기 위해선 이 패키지 이름을 공백 식별자로 바꾸면 된다:

```
import _ "net/http/pprof"
```

이러한 형태의 임포트는 패키지가 부수효과를 위해 임포트되고 있음을 명확하게 할 수 있다. 왜냐하면 이 파일에서는 패키지가 이름을 갖고 있지 않기 때문에 사용될 가능성이 없기 때문이다. (만약 이름을 갖고 있고 이를 사용하지 않는다면, 컴파일러는 프로그램을 거부할 것이다.)

인터페이스 검사

위의 인터페이스에 대한 논의에서 봤듯이, 타입은 인터페이스를 구현했음을 명시적으로 선언할 필요가 없다. 대신에, 타입은 인터페이스의 메서드를 구현함으로써 인터페이스를 구현한다. 실제로 , 대다수의 인터페이스 변환은 정적이며 따라서 컴파일 도중에 검사가 이루어진다. 예를 들면, 만약 `*os.File`이 `io.Reader`인터페이스를 구현하고 있지 않는데 이를 `io.Reader`를 기대하는 함수에 인자로 전달하게 되면 컴파일이 되지 않을 것이다.

하지만 몇 몇 인터페이스 검사는 런타임때 발생한다. 한 가지 예시는 `Marshaler` 인터페이스를 정의하는 `encoding/json`패키지에 있다. JSON 인코더가 저 인터페이스를 구현한 값을 받을 때, 인코더는 JSON으로 변환을 하기 위해 표준 변환을 진행하는 대신 값의 `marshiling` 메서드를 실행한다. 인코더는 런타임때 다음과 같이 타입 단언을 하면서 프로퍼티를 검사한다.

```
m, ok := val.(json.Marshaler)
```

만약 타입이 인터페이스를 구현했는지 안했는지를 실제 인터페이스 자체를 사용하지 않고, 에러 검사의 일부로서 확인할 필요가 있을 때, 타입 단언된 값을 무시하기 위해 공백 식별자를 사용하라:

```
if _, ok := val.(json.Marshaler); ok {
    fmt.Printf("value %v of type %T implements json.Marshaler\n", va
}
```

이러한 상황이 나타나는 경우는 패키지가 실제로 인터페이스를 만족하는 타입을 구현하고 있는지를 보장할 필요가 있을 때이다. 만약 어떤 타입이 커스터마이징된 JSON 표기법이 필요하다면(예를 들면, `json.RawMessage`), 이는 `json.Marshaler`를 구현해야 한다. 그러나 컴파일러가 이를 자동으로 확인하도록 하는 정적 변환은 없다. 만약 부주의하게 타입이 그 인터페이스를 만족하는데에 실패를 하게되면 JSON 인코더는 여전히 실행되나 커스터마이징된 구현체를 사용할 수 없게 된다. 인터페이스의 구현을 보증하기 위해서는, 패키지 안에서 공백 식별자를 이용하는 전역 선언문을 사용할 수 있다.

```
var _ json.Marshaler = (*RawMessage)(nil)
```

위의 선언에서 `*RawMessage`를 `Marshaler`로의 변환시키는 할당을 통해 `*RawMessage`가 `Marshaler`를 구현할 것을 요구하고 있으며, 이러한 특성은 컴파일시 검사될 것이다. 만약 `json.Marshaler`인터페이스에 변화가 생기면, 이 패키지는 더 이상 컴파일 되지 않을것이고, 패키지가 업데이트 되어야 함을 알게해준다.

이 구조에서 공백 식별자가 나타나는것은 위 선언이 변수를 만드는게 아니라 단지 타입 검사를 위해서만 존재함을 알려준다. 하지만 이를 하나의 인터페이스를 만족하는 모든 타입에 사용하지는 마라. 규약에 의하면, 위와 같은 선언은 드문 경우이지만, 이미 코드에 존재하는 정적 변환이 없는 경우에만 사용하라는 것이다.

임베딩

임베딩(Embedding)

- 원문: Embedding
- 번역자: Jhonghee Park

Go는 전형적인, 타입주도형의 subclassing을 제공하지 않는다. 하지만 struct나 인터페이스안에 타입을 임베딩함으로써 구현체의 일부를 "빌리는"것은 가능하다.

인터페이스 임베딩은 매우 간단하다. 이미 언급된 바 있는 io.Reader와 io.Writer 인터페이스의 정의를 살펴보자.

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

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

io패키지는 다수의 또 다른 인터페이스들을 노출시키고 있는데, 다수의 메서드를 구현할 수 있는 객체를 명시하는데 쓴다. 예를 들어, io.ReadWriter는 Read와 Write를 모두 가지고 있다. 두 메서드를 명시적으로 나열해서 io.ReadWriter를 정의 할 수도 있겠지만, 더 쉽고 기억하기 좋은 방법은 두 개의 인터페이스를 끼워 넣어서(embedded) 다음과 같이 하나의 인터페이스를 형성시키는 것이다:

```
// ReadWriter는 Reader와 Writer 인터페이스를 조합하는 인터페이스이다.
type ReadWriter interface {
    Reader
    Writer
}
```

보이는 데로 Reader가 하는 일과 Writer가 하는 일을 ReadWriter는 모두 할 수 있다는 말이다; 끼워진(embedded) 인터페이스들의 조합이다 (서로 공통된 메서드가 없는 인터페이스들 이어야 한다). 오직 인터페이스만이 인터페이스에 끼워질 수 있다.

기본적으로 똑같은 아이디어가 struct에도 적용할 수 있는데 그 영향은 훨씬 광범위하다. bufio패키지에는 bufio.Reader와 bufio.Writer 두 struct타입이 있는데, 물론 io패키지에 있는 유사한 인터페이스를 구현하고 있다. 그런데 bufio는 또 버퍼를 내재한 reader/writer를 구현하기도 하는데, 임베딩(embedding)을 이용하여 reader와 writer를 하나의 struct에 조합하는 것이다: struct안에 타입들을 나열하되 필드 이름을 주지 않는 방식이다.

```
// ReadWriter Reader와 Writer에 대한 포인터들을 저장한다.
// io.ReadWriter를 구현한다.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

임베드된 요소들은 struct를 가리키는 포인터들이고, 물론 사용하기 전에 유효한 struct로 포인트를 걸어서 초기화 시켜주어야 한다. ReadWriter struct는 아래와 같이 작성될 수 있다.

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

하지만 이렇게 하면 io를 충족시키고 reader와 writer가 가지고 있는 메서드를 사용하기 위해서 전송용 메서드(forwarding method)를 따로 제공해야 한다:

```
func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}
```

이런 귀찮은 일들을 피하기 위해서는 struct를 직접 임베딩하면 된다. 임베드된 타입의 메서드는 자동으로 따라 오며, 그 의미는 bufio.ReadWriter는 bufio.Reader와 bufio.Writer의 메서드를 모두 가지게 된다는 말이다. 동시에 다음의 세 인터페이스를 충족시키기도 한다: io.Reader, io.Writer, 그리고 io.ReadWriter.

이제 임베딩이 subclassing과 다른 중요한 방식을 살펴보자. 타입을 임베드하게 되면 그 타입의 메서드들이 외부 타입의 메서드가 된다. 하지만 호출된 메서드의 리시버는 내부 타입이지 외부 타입이 아니다. 예제에서, bufio.ReadWriter의 Read 메서드가 호출되었을 때, 전달용 메서드를 사용한 것과 같은 똑같은 효과가 있다; 리시버는 ReadWriter의 reader 필드이지, ReadWriter 자체가 아닌 것이다.

임베딩은 또한 단순한 편리함일 수 있다. 이 예제는 임베드된 필드를 이름이 있는 보통 필드와 함께 보여준다.

```
type Job struct {
    Command string
    *log.Logger
}
```

Job 타입은 이제 `*log.Logger`에 속한 Log, Logf 그리고 다른 메서드들을 가지게 된다. Logger에 이름을 줄 수도 있었겠지만, 그럴 필요가 전혀 없다. 그리고 이제, 일단 초기화가 되면, Job에 직접 log를 사용할 수 있다:

```
job.Log("starting now...")
```

Logger는 Job struct의 보통 필드이기 때문에 constructor 안에서 항상 하는 것 처럼 다음과 같이 초기화 할 수 있다.

```
func NewJob(command string, logger *log.Logger) *Job {
    return &Job{command, logger}
}
```

혹은 composite literal을 써서

```
job := &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}
```

만약 임베드된 필드를 직접 언급해야 할 경우가 생기면, ReadWriter struct의 Read 메서드처럼, 패키지를 무시한 필드의 타입명이 필드의 이름으로 사용된다. Job 타입의 변수인 job의 `*log.Logger`에 접근할 필요가 있다면, job.Logger라고 쓰면 되고, Logger의 메서드를 개선하길 원할 때 유용하다.

```
func (job *Job) Logf(format string, args ...interface{}) {
    job.Logger.Logf("%q: %s", job.Command, fmt.Sprintf(format, args.
```

```
}
```

임베딩 타입들은 이름 충돌의 문제를 발생시킬 수도 있지만 해결하는 규칙들은 간단하다. 첫번째, 필드나 메서드 X는 타입내 더 깊숙히 중첩된 부분에 있는 또 다른 X를 가려서 안 보이게 한다. 만약 log.Logger가 Command라는 필드나 메서드가 있다면, Job의 Command 필드가 더 우세하다.

둘째로, 똑같은 이름이 같은 레벨로 중첩되어 나타나면, 보통은 에러가 생긴다; 만약에 Job struct이 Logger라는 이름으로 다른 필드나 메서드를 가지고 있다면 log.Logger를 임베드하는 것은 잘못된 것이다. 그렇지만, 복제된 이름이 타입 정의 밖에서 사용된 일이 없다면 괜찮다. 이러한 자격은 밖으로 부터 임베드된 타입에 생기는 변화에 대해서 어느 정도 보호를 제공한다; 필드 하나가 추가되면서 또 다른 subtype의 필드와 충돌이 생기는 경우, 둘 중 어느 필드도 사용되지 않았다면 아무 문제가 없다.

동시성

동시성(Concurrency)

- 원문: Concurrency
- 번역자: Philbert Yoon (@ziwon)

통신에 의한 공유

동시성 프로그래밍은 광범위한 주제이므로 여기에서는 Go에 한정된 중요한 것들에 대해서만 지면을 할애한다.

공유 변수에 대한 정확한 접근을 구현하기 위해 엄수해야 하는 세세한 내용들은 다양한 환경에서 동시성 프로그래밍을 어렵게 했다. Go는 공유 변수가 채널을 돌려가며 전달된다는 점에서 다른 접근을 권장한다. 그리고 사실 공유 변수는 개별 쓰레드 실행에 의해서 결코 공유되지 않는다. 언제든지 하나의 고루틴이 값에 접근한다. 데이터 경쟁은 구현 설계상 발생할 수 없다. 이러한 사고방식을 권장하기 위해 이를 한 슬로건으로 줄였다:

공유 메모리로 통신하지 말라. 대신, 통신으로 메모리를 공유하라.

이런 접근은 너무 지나친 것일 수 있다. 예를 들어, 정수형 변수 주위에 뮤텍스를 두는 방식의 레퍼런스 카운트가 최고일지도 모른다. 그러나 상위 레벨에서 접근하는 방법으로써, 접근을 제어하는 채널을 사용하면 분명하고 정확한 프로그램을 보다 쉽게 작성할 수 있다.

이 모델에 대해 생각해보는 한가지 방법은 단일 CPU에서 실행되는 전형적인 단일 쓰레드 프로그램을 떠올려 보는 것이다. 여기에는 동기화를 위한 기본 자료형이 필요 없다. 지금 또다른 그 인스턴스를 실행시켜 보라. 그러나 역시 동기화가 필요하지 않다. 이제 그 두 개를 통신하게 하는데, 그 통신 자체가 동기화 장치(synchronizer)인 경우, 여전히 다른 동기화가 필요 없다. 예를 들어, 유닉스 파이프 라인은 이 모델에 완벽하게 들어 맞는다. 동시성에 대한 Go의 접근 방식이 호어(Hoare)의 통신 순차적 프로세스 (CSP, Communicating Sequential Processes)에서 비롯되었지만, 타입 안전이 보장되는 식의 일반화된 유닉스 파이프라고 볼 수 있다.

고루틴

쓰레드, 코루틴, 프로세스 등 기존의 용어는 부정확한 함의를 전달하기 때문에 고루틴이라고 부른다. 고루틴은 단순한 모델이다. 즉, 같은 주소 공간에서 다른 고루틴과 동시에 실행되는 함수이다. 고루틴은 가볍다. 스택 영역을 할당하는 것에 비해 비용이 적게 든다. 그리고 그 스택은 작은 크기로 시작된다. 그래서 저렴하다. 그리고 필요한만큼 힙 스토리지를 할당(또는 해제)하여 커진다.

고루틴은 OS의 다중 쓰레드에 멀티플렉싱되는데, I/O 작업을 위해 대기중일 때와 같이 하나의 고루틴이 블락이 되면 다른 고루틴이 계속 실행된다. 이런 설계는 쓰레드의 복잡한 생성과 관리에 대해 굳이 알 필요가 없게 해준다.

새 고루틴을 호출하여 실행하려면 `go` 키워드를 함수 또는 메쏘드 호출 앞에 둔다. 호출이 완료되면, 고루틴은 자동으로 종료된다. (백그라운드에서 명령을 실행하는 유닉스쉘 및 표기법과 유사한 효과이다.)

```
go list.Sort()  // list.Sort를 (정렬이 완료될 때까지) 기다리지 말고 동시에 실
```

함수 리터럴은 고루틴 호출에 유용할 수 있다.

```
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }() //괄호에 주목 - 반드시 함수를 호출해야 함
}
```

Go에서 함수 리터럴은 클로저이다. 즉, 함수가 참조하는 변수를 사용하는 동안에는 그 생존을 보장하는 방식으로 구현되어 있다는 것이다.

위의 예제는 함수가 종료를 알릴 방법이 없기 때문에 아주 실용적이진 않다. 이를 위해 채널이 필요하다.

채널

맵과 마찬가지로, 채널은 make로 할당되고, 그 결과 값은 실제 데이터 구조에 대한 참조로서 동작한다. 선택적인 정수형 매개변수가 주어지면, 채널에 대한 버퍼 크기가 설정된다. 이 값은 언버퍼드(Unbuffered) 또는 동기 채널에 대해서 기본값이 0이다.

```
ci := make(chan int)            // 정수형의 언버퍼드 채널
cj := make(chan int, 0)         // 정수형의 언버퍼드 채널
cs := make(chan *os.File, 100)  // File 포인터형의 버퍼드 채널
```

버퍼가 없는(Unbuffered) 채널은 동기화로 값을 교환하며 두 계산(고루틴들)이 어떤 상태에 있는지 알 수 있다는 것을 보장하는 통신을 결합한다.

채널을 사용하는 멋있는 Go스러운 코드가 많다. 다음 한 예제로 시작해보자. 이전 섹션에서 백그라운드에서 정렬을 했다. 채널은 정렬이 완료될 때까지 고루틴 실행을 대기시킬 수 있다.

```
c := make(chan int)  // 채널을 할당
// 고루틴에서 정렬 시작하고 완료되면 채널에 신호를 보냄
go func() {
    list.Sort()
    c <- 1  //  신호를 보내지만 값은 문제가 안됨
}()
doSomethingForAWhile()
<-c   // 정렬이 끝날 때까지 기다리고 전달된 값은 버림
```

수신부는 수신할 데이터가 있을 때까지 항상 블락된다. 언버퍼드 채널이면, 송신부는 수신부가 값을 받을 때까지 블락된다. 버퍼드 채널이면, 값이 버퍼에 복사될 때까지만 송신부가 블락된다. 그러므로 버퍼가 꽉 차면, 이는 특정 수신부가 값을 획득할 때까지 대기 중이라는 것을 의미한다.

버퍼드 채널은 세마포처럼 사용될 수 있다. 예를 들어 처리량을 제한하는 것이다. 다음 예제에서, 들어오는 요청들은 handle에 넘겨진다. handle에서는 하나의 값(어떤 값이라도 상관없음)을 채널에 송신하고, 요청을 처리한 다음, 채널로부터 한 값을 받아 다음 소비자를 위해 "세마포"를 준비시킨다. 채널 버퍼의 크기가 동시에 처리할 수 있는 숫자를 제한하는 것이다.

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // 액티브큐가 비워질 때까지 대기
    process(r)  // 오래 걸릴 수 있는 작업
    <-sem       // 완료, 실행될 다음 요청을 활성화
}
```

```
func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // 끝날 때까지 대기하지 않음
    }
}
```

일단 MaxOutstanding 수 만큼의 핸들러가 `process`를 실행하는 동안에는, 기존 핸들러 중 하나가 완료되고 버퍼로부터 값을 받을 때까지 더 이상의 꽉 찬 채널 버퍼에 송신하는 것은 블락될 것이다.

그렇지만 이 설계는 문제가 있다. 즉, 전체 요청에서 겨우 `MaxOutstanding` 수 만큼 `process`를 실행할 수 있음에도 서버는 들어오는 모든 요청에 대해 새로운 고루틴을 생성한다는 것이다. 그 결과, 요청이 너무 빨리 들어올 경우, 무제한으로 리소스를 낭비할 수 있다. 이 결함은 고루틴의 생성을 제한하는 게이트로 `Serve`를 수정함으로써 해결될 수 있다. 확실한 솔루션은 다음과 같지만, 나중에 수정하게 될 버그가 있다는 것에 주의하라:

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func() {
            process(req) // 버그, 아래 설명을 참고
            <-sem
        }()
    }
}
```

버그는 Go `for` 루프에 있다. 루프 변수는 각 반복마다 재사용되어 동일한 `req` 변수가 모든 고루틴에 걸쳐 공유된다. 이는 원하는 바가 아니다. 각 고루틴마다 구별된 `req` 변수를 가지도록 해야한다. 다음은 이를 위한 한 가지 방법으로, 고루틴의 클로져에 대한 인자로 `req`의 값을 전달하는 것이다:

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func(req *Request) {
            process(req)
            <-sem
        }(req)
    }
}
```

이전 버전과 클로져가 어떻게 선언되고 실행되는지 차이점을 보기 위해 다음 버전을 비교해보라. 또다른 솔루션은 다음 예제처럼 그냥 같은 이름의 새로운 변수를 생성하는 것이다.

```
func Serve(queue chan *Request) {
    for req := range queue {
        req := req // 고루틴을 위한 새로운 req 인스턴스를 생성
        sem <- 1
        go func() {
            process(req)
            <-sem
        }()
    }
```

```
}
```

이상한 코드를 작성하는 것처럼 보일 수 있다.

```
req := req
```

하지만 Go에서 이렇게 하는 것은 합법적이고 Go 언어다운 코드이다. 이름이 같은 새로운 버전의 변수가 의도적으로 루프 변수를 지역적으로 가리지만, 각 고루틴에 대해서는 유니크한 값이다.

서버를 작성하는 일반적인 문제로 돌아가면, 리소스를 잘 관리하는 다른 방법은 요청 채널을 읽는 모든 `handle` 고루틴을 고정된 수에서 시작하는 것이다. 고루틴의 수는 `process`가 동시에 호출되는 수를 제한한다. `Serve` 함수는 종료 신호를 수신하게 되는 채널도 (인자로) 받고 있으므로 고루틴이 시작되면, 이 채널로부터의 수신은 블락된다.

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
    }
}

func Serve(clientRequests chan *Request, quit chan bool) {
    // 핸들러 시작
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // 종료 신호를 받을 때까지 대기
}
```

채널의 채널

Go의 가장 중요한 속성 중 하나는 채널이 다른 것과 마찬가지로 할당되고 전달될 수 있는 일급변수(first-class value)라는 것이다. 일반적으로 이 속성은 안전한 병렬 역다중화(parallel demultiplexing)를 구현하는데 사용된다.

이전 섹션의 예제에서, `handle` 은 요청에 대해서는 이상적인 핸들러였으나 핸들러가 다루는 타입은 정의하지 않았다. 해당 타입이 회신을 하는 채널을 포함하는 경우, 각 클라이언트는 자신에게 응답 경로를 제공할 수 있다. 다음은 `Request` 타입의 개략적인 정의이다.

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

클라이언트는 함수와 그 인자뿐만 아니라 요청 객체 안에서 응답을 수신하는 채널을 제공하고 있다.

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}
```

```
request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// 요청 전송
clientRequests <- request
// 응답 대기
fmt.Printf("answer: %d\n", <-request.resultChan)
```

서버 측에서는 핸들러 함수만 수정하면 된다.

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

실제로 사용하기 위해서는 아직 할 일이 많은 것이 명백하지만, 이 코드는 속도 제한, 병렬, 넌블락 RPC 시스템을 위한 프레임워크이다. 그리고 뮤텍스는 눈에 보이지 않는다.

병렬화

이러한 아이디어의 또 다른 응용 프로그램은 멀티코어 CPU에 대해 계산을 병렬처리하는 것이다. 계산을 독립적으로 실행할 수 있는 부분들로 분리할 수 있다면, 각 부분들이 완료될 때 신호를 보내는 채널들로 병렬처리할 수 있다.

벡터 아이템에 대한 실행 비용이 비싼 연산이 있다고 가정해보자. 그리고 다음 이상적인 예제에서와 같이 각 아이템을 연산한 값은 독립적이다.

```
type Vector []float64

//  v[i], v[i+1]에서 v[n-1]까지 연산을 적용
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // 이 부분이 완료되면 신호함
}
```

루프에서 독립적으로 각 부분들을 CPU당 하나씩 실행시킨다. 이들은 어떤 순서로도 완료될 수 있지만 순서는 문제되지 않는다. 그러므로 모든 고루틴를 실행시킨 후 채널을 비워서 그냥 완료 신호를 카운트한다.

```
const numCPU = 4 // CPU  코어수

func (v Vector) DoAll(u Vector) {
    c := make(chan int, numCPU)  // 버퍼는 선택적이나 상식적
    for i := 0; i < numCPU; i++ {
        go v.DoSome(i*len(v)/numCPU, (i+1)*len(v)/numCPU, u, c)
    }
    // 채널을 비움
    for i := 0; i < numCPU; i++ {
        <-c    // 한 태스크가 끝날 때까지 대기
    }
    // 모두 완료
}
```

numCPU을 상수값으로 생성하기보다는 적절한 값을 런타임시에 요구할 수 있다. `runtime.NumCPU` 함수는 장비의 CPU 하드웨어 코어수를 반환한다. 그래서 아래와 같이 작성할 수 있다.

```
var numCPU = runtime.NumCPU()
```

또한 Go 프로그램은 동시에 실행할 수 있는 사용자 지정 코어수를 보고하는 (또는 설정하는) `runtime.GOMAXPROCS` 함수가 있다. `runtime.NumCPU`의 값이 기본값이지만, 비슷하게 명명된 환경변수 설정에 의해 혹은 양수의 인자로 함수를 호출하여 재정의할 수 있다. 0으로 호출하면 바로 값을 조회한다. 따라서 사용자의 리소스 요청을 따르고 싶은 경우, 다음과 같이 작성해야 한다.

```
var numCPU = runtime.GOMAXPROCS(0)
```

컴포넌트를 독립적으로 처리하여 프로그램을 구조화하는 동시성과 다중 CPU에서 효율성을 위해 계산을 병렬로 처리하는 병렬성의 개념을 혼동하지 않길 바란다. Go의 동시성 특징이 병렬 계산으로 문제를 쉽게 구조화할 수 있지만, Go는 병렬이 아닌 동시성 언어이고, 모든 병렬화 문제가 Go에 들어맞지는 않는다. 이 구분에 대한 논의는 이 블로그 포스트에 인용된 토크를 참조하라.

누설 버퍼

비동시성(non-concurrent) 개념도 동시성 프로그래밍 도구로 쉽게 표현할 수 있다. 다음은 RPC 패키지에서 추출한 예제이다. 클라이언트 고루틴은 아마도 네트워크인 특정 소스의 데이터를 반복해서 수신한다. 버퍼의 할당과 해제를 피하기 위해서 `free list`를 유지하며 이를 대신할 버퍼 채널을 사용한다. 채널이 비어 있으면 새로운 버퍼가 할당된다. 일단 메시지 버퍼가 준비되면 `serverChan`의 서버로 송신한다.

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        var b *Buffer
        // 사용할 수 있는 버퍼를 획득하거나 그렇지 않다면 할당
        select {
            case b = <-freeList:
                // 하나를 획득하고 아무 작업도 하지 않음
            default:
                // 획득할 버퍼가 없으니 새 버퍼를 할당함
            b = new(Buffer)
        }
        load(b)              // 네트워크에서 다음 메세지를 읽음
        serverChan <- b      // 서버에 전송
    }
}
```

서버 루프는 각 클라이언트로부터 메시지를 수신해서 처리하고, free list에 버퍼를 반환한다.

```
func server() {
    for {
        b := <-serverChan    // 동작을 위해 대기
        process(b)
        // 가능할 경우 버퍼를 재사용
```

```
        select {
            case freeList <- b:
                // free list의 버퍼. 아무 것도 하지 않음
            default:
                // free list가 꽉 참, 그냥 계속
        }
    }
}
```

클라이언트는 `freeList`에서 버퍼를 획득하려고 시도하는데, 버퍼를 사용할 수 없는 경우에는 새로운 버퍼를 할당한다. 리스트가 꽉 차 있지 않는 이상, 서버는 버퍼를 `freeList`에 송신하여 프리 리스트에 버퍼 b를 다시 둔다. 리스트가 꽉 차 있는 경우, 버퍼는 바닥에 떨어져 가비지 콜렉터에 의해 회수된다. (`select` 구문에서 `default` 절은 다른 case가 준비되지 않은 경우에 실행된다. 이는 `select`는 결코 블락되지 않는다는 것을 뜻한다.) 단지 몇 줄로 버퍼 채널과 버퍼 크기를 부기하는 가비지 컬렉터에 기대어 구현된 누설 버킷 프리 리스트(leaky bucket free list)를 만들었다.

에러

에러

- 원문 : Errors
- 번역자 : MinJae Kwon (@mingrammer)

라이브러리 루틴들은 호출자에게 어떤 조짐의 에러라 해도 자주 리턴해 주어야만 한다. 이전에 언급되었듯이, Go의 다중값 리턴은 일반적인 리턴값에 비해 상세한 에러 내용을 제공하기 쉽게 만들어준다. 상세한 에러 내용을 제공하기 위해 이러한 특징을 활용하는 것은 좋은 방식이다. 예를 들면, 앞으로 보게될 `os.Open`은 실패할 경우 단순히 `nil`만을 리턴하지 않으며, 무엇이 잘못되었는지에 대한 에러 내용까지 리턴을 한다.

규약에 의하면 에러는 간단한 내장 인터페이스의 에러 타입을 가진다

```
type error interface {
    Error() string
}
```

라이브러리 개발자는 보이지는 않지만 좀 더 풍부한 모델을 사용해 위의 인터페이스를 자유롭게 구현하면서, 에러를 보여줄 뿐만 아니라 어떤 맥락을 함께 제공하는 것도 가능하다. 언급된 바와 같이, 보통 `os.Open`는 `*os.File` 값을 리턴하는 동시에 에러값도 리턴한다. 만약 파일이 성공적으로 오픈되면, 에러는 `nil`이 될 것이고, 오픈 도중 문제가 발생할경우 `os.PathError`의 값을 가지게 될 것이다:

```
// PathError는 에러와 연산, 문제를 발생시킨 파일 경로를 가지고 있다.
type PathError struct {
    Op string    // "open", "unlink", 등.
    Path string  // 관련 파일.
    Err error    // 시스템 콜에 의해 리턴됨.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

`PathError`의 `Error`메서드는 다음과 같은 문자열을 생성한다:

```
open /etc/passwx: no such file or directory
```

위의 에러는 문제가 발생한 파일명과 연산, 그리고 문제를 발생시킨 운영체제 에러를 가지고 있으며, 이는 문제를 발생시킨 시스템 콜로부터 멀리 떨어진 곳에서 보여준다해도 유용하다. 이는 단순히 `no such file or directory`를 보여주는것보다 훨씬 많은 정보를 제공한다.

가능한 경우, 에러 문자열은 에러가 발생된 명령이나 패키지를 접두어로 쓰는등, 에러의 근원지를 파악할 수 있어야한다. 예를 들면, `image`패키지에서 알 수 없는 포맷으로 인해 발생한 디코딩 에러를 위한 문자열 표현은 `image: unknwon format.`이다.

정확한 상세 에러내용에 관심이 있는 호출자는 자세한 그리고 추가적인 정보를 얻기 위해 타입 스위칭이나 타입 단언을 사용할 수 있다. `PathErrors`의 경우 오류를 복구하기 위해 내부 `Err`필드를 조사할 수 있다.

```
for try := 0; try < 2; try++ {
    file, err = os.Create(filename)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {
        deleteTempFiles()  // 공간을 확보한다.
        continue
    }
    return
}
```

두 번째 if 문에서 `err` 인터페이스를 `os.PathError` 타입으로 변환한다(type assertion). 실패하면 `ok`는 `false`가 되고, `e`는 `nil`이 된다. 성공하면 `ok`는 `true`가 되고, 이는 에러가 `*os.PathError` 타입임을 의미한다. 이에 `e` 에서 에러에 대한 더 많은 정보를 확인할 수 있다.

패닉 (Panic)

호출자에게 에러를 알려주는 일반적인 방법은 에러를 부가적인 값으로서 리턴하는 것이다. 규범적인 `Read` 메서드는 잘 알려진 예로, 이는 바이트 수와 에러를 리턴한다. 그러나 복구할 수 없는 에러라면 어쩔 것인가? 때때로 프로그램은 더이상 진행이 안될 것이다.

이런 목적을 위해 Go에는 런타임 에러를 일으켜 프로그램을 중단시키는 `panic`이라는 내장함수가 있다 (그러나 다음 섹션을 보라). 이 함수는 프로그램이 종료되었을 때 출력될 임의의 타입 하나를 인자로 받는다 (대개 `string`이다). 이는 또한 어떤 불가능한 일이 벌어졌음을 알리는 방법이기도 한데, 무한 루프에서 탈출하는 것이 예가 될 수 있다.

```
// 뉴턴 메소드를 사용해 세제곱근 구하기 구현하기
func CubeRoot(x float64) float64 {
    z := x/3   // 임의의 초기값
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // 백만회의 루프를 돌려도 수렴하지 않는다. 무언가 잘못되었다.
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

이것은 예시일 뿐이다. 실제 라이브러리 함수는 `panic`을 피해야한다. 만약 문제가 숨어있거나 주변에서 문제를 발생시킨다해도, 실행을 계속 진행하는것이 전체 프로그램을 종료하는 것보다 낫다. 한 가지 가능한 반례는, 초기화 작업중에 있는데, 만약 라이브러리가 초기 셋업을 못하게 된다면, 이는 패닉을 발생시키고 알려야 할 것이다.

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

복구 (Recover)

슬라이스의 범위를 넘어선 인덱싱이나 타입 단언 실패등의 런타임 에러를 포함한 패닉이 발생하였을 때, 이는 즉시 현재 함수의 실행을 중단시키며 모든 지연된(deferred) 함수를 실행하면서 고루틴 스택을 풀기 시작한다. 만약 풀기 작업이 고루틴 스택의 최정상에 도달했을 때, 프로그램은 종료된다. 그러나 내장함수인 `recover`를 사용하면 고루틴의 통제권을 다시 얻을 수 있으며, 명령어 실행을 정상적으로 진행할 수 있게 된다.

`recover`를 호출하면 풀기 작업이 중지되며, `panic`에 전달된 인자값이 리턴된다. 풀기 작업을 하는 동안에 실행되는 코드는 오직 지연된 함수안에서만 실행되기 때문에 , `recover`는 오직 지연된 함수 내에서만 유용하다.

`recover`의 한 응용 사례중 하나는 서버내의 실행중인 다른 고루틴들은 죽이지 않고, 오직 실패한 고루틴만을 종료시키는 것이다.

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}
```

이 예시에서, 만약 `do(work)`에서 패닉이 발생하면, 그 결과가 로그로 남겨질 것이고 고루틴은 다른 고루틴들을 방해하지 않으면서 깔끔하게 종료될 것이다. 호출된 `recover`가 이 상황을 완전히 처리할 것이기 때문에 지연된 클로져에서는 그 어떤 것도 할 필요가 없다.

`recover`는 지연된 함수로부터 직접 호출되는 경우를 제외하면 항상 `nil`을 리턴하기 때문에, 지연된 코드는 `panic`과 `recover`를 사용하는 라이브러리 루틴을 실패없이 호출할 수 있다. 한 가지 예를 들어보면, `safelyDo`내의 지연된 함수는 `recover`를 호출하기 전에 logging 함수를 호출할 수 있고, logging 코드는 패닉 상태에 영향을 받지 않으면서 잘 실행될 것이다.

위에서의 복구 패턴을 보면, `do`함수 (그리고 호출을 하는 그 어느것도)는 `panic`을 호출함으로써 안좋은 상황을 피해갈 수 있다. 이 아이디어를 활용하면 복잡한 소프트웨어에서의 에러 핸들링을 단순화 할 수 있다. `regexp`패키지의 가장 이상적인 버전을 보자. 이는 자체 에러 타입과 함께 `panic`을 호출함으로써 파싱 에러를 알린다. 아래에 `Error`와 에러 메서드, 그리고 `Compile`함수의 정의가 있다.

```
// Error는 파싱 에러 타입이며 error 인터페이스를 만족한다.
type Error string
func (e Error) Error() string {
    return string(e)
}

// error는 Error를 가진 패닉으로 파싱 오류를 알리는 *Regexp 메서드이다.
func (regexp *Regexp) error(err string) {
```

```
    panic(Error(err))
}

// Compile은 정규 표현식의 파싱된 표현을 리턴한다.
func Compile(str string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // doParse는 파싱중 에러가 발생하면 패닉을 일으킨다.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // 리턴 값 클리어
            err = e.(Error) // 파싱 에러가 아니면 다시 패닉 발생
        }
    }()
    return regexp.doParse(str), nil
}
```

만약 doParse가 패닉을 발생시키면, 복구 블록은 리턴 값을 nil로 설정할 것이다. 지연된 함수는 명명된 리턴 값들을 변경할 수 있다. err에 값을 할당하는 과정에서 에러를 자체 에러 타입으로 단언함으로써 그 문제가 파싱 에러인지 아닌지를 검사를 할 것이다. 만약 파싱 에러가 아니라면, 타입 단언은 실패할 것이고, 마치 아무런 중단이 없었던 것처럼 스택 풀기작업을 진행하며 런타임 에러를 일으킬 것이다. 이 검사는 인덱스가 범위를 벗어나는 등의 의도치않은 일이 생길 때 파싱 에러를 처리하기위해 panic과 recover를 사용했음에도 불구하고 코드가 실패함을 의미한다.

아래 에러 처리를 보면, 에러 메서드는 (타입에 바인딩하는 메서드이기 때문에 내장 에러 타입과 동일한 이름을 사용하는것은 자연스러우며 괜찮다.) 직접 파싱 스택을 푸는것에 대한 걱정없이 파싱 에러를 알리기 쉽게 한다:

```
if pos == 0 {
    re.error("'*' illegal at start of expression")
}
```

이 패턴은 유용함에도 불구하고, 오직 패키지 안에서만 사용되어야 한다. Parse는 내부 패닉 호출을 에러 값으로 바꾸며, 이는 클라이언트에게 패닉을 노출시키지 않는다. 이는 본받을만한 좋은 규칙이다.

부연적으로, re-panic 관용구는 실제로 에러가 발생했을 때 패닉 값을 변경한다. 하지만, 원래 값과 새로운 실패들이 모두 오류 보고에서 보여질 것이기 때문에 문제의 근본 원인도 여전히 보여질 것이다. 따라서 (프로그램이 크래쉬 했다는 점을 감안하면) 이러한 간단한 re-panic 접근이 충분하다고 볼 수 있지만, 오직 원래의 값만 보여주고 싶을 땐, 의도치않은 문제를 필터링하는 코드를 작성할 수 있으며 원래의 에러를 가지고 re-panic을 할 수 있다. 이는 독자들에게 숙제로 남기겠다.

웹 서버

웹 서버

- 원문 : A web server
- 번역자 : MinJae Kwon (@mingrammer)

완전한 Go 프로그램인 웹 서버로 마무리를 짓자. 이것은 실제로 웹 서버의 한 종류이다. 구글은 데이터를 자동으로 차트와 그래프로 만들어주는 `chart.apis.google.com` 서비스를 제공한다. 이것은 URL에 데이터를 실어서 쿼리로 만들 필요가 있기 때문에, 쉬워 보임에도 불구하고, 인터랙티브하게 사용하기가 어렵다. 여기서 주어진 프로그램은 데이터 폼 (이 예제에선 짧은 텍스트가 하나 주어졌다.)에 대해 텍스트를 인코딩한 행렬 형태의 QR코드를 생성하는 차트 서버를 호출하는 훌륭한 인터페이스를 제공한다.

여기에 완전한 프로그램이 있다. 설명을 따라가보자.

```
package main

import (
    "flag"
    "html/template"
    "log"
    "net/http"
)

var addr = flag.String("addr", ":1718", "http service address") // Q

var templ = template.Must(template.New("qr").Parse(templateStr))

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

func QR(w http.ResponseWriter, req *http.Request) {
    templ.Execute(w, req.FormValue("s"))
}

const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
</head>
<body>
{{if .}}
<img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe
```

```
<br>
{{.}}
<br>
<br>
{{end}}
<form action="/" name=f method="GET"><input maxLength=1024 size=70
name=s value="" title="Text to QR Encode"><input type=submit
value="Show QR" name=qr>
</form>
</body>
</html>
`
```

메인 위 까지는 이해하기가 쉬울 것이다. 한 플래그는 서버의 기본 HTTP 포트를 세팅한다. 템플릿 변수인 `templ`를 사용하면서 재미가 생기기 시작한다. 이는 서버가 페이지를 보여주려고 할 때 HTML 템플릿을 제작한다. 자세한 건 잠시 후에 살펴보자.

메인 함수는 플래그를 파싱하고, 위에서 말한 메커니즘과 같이 `QR` 함수를 서버의 루트 경로에 바인딩한다. 그런 다음엔 서버를 시작하기 위해 `http.ListenAndServe`가 호출되며, 이는 서버가 실행되는 동안에는 블로킹된다.

`QR` 함수는 폼 데이터를 포함한 웹 요청만을 받으며, `s`라는 이름을 가진 폼데이터를 가지고 템플릿을 실행한다.

템플릿 패키지 `html/template`는 강력한 기능을 제공하기 때문에 이 프로그램에서 사용하기만 하면 된다. 사실, 이는 `templ.Execute`를 통해 전달된 데이터로부터 얻어지는 요소들을 대체함으로써 즉시 HTML 텍스트의 일부를 재작성한다. 이 예시에선 폼 데이터 값이 되겠다. 템플릿 텍스트 (`templateStr`)에서 이중 중괄호 부분은 템플릿 행동을 나타낸다. `{{if .}}`부터 `{{end}}`까지는 `. (dot)`라고 불리는 현재 데이터의 값이 빈 문자열이 아닐 때에만 실행된다. 만약 문자열이 비어있을 땐, 템플릿의 해당 부분은 무시될 것이다.

두 개의 `{{.}}`은 쿼리로 들어온 데이터를 템플릿에 표현하고 웹 페이지에서 보여주겠다고 말한다. HTML 템플릿 패키지는 텍스트가 안전하게 보여질 수 있도록 적절한 이스케이프를 제공한다.

템플릿 스트링의 나머지 부분들은 웹 페이지가 로드될 시 그냥 HTML로 보여진다. 설명이 부족하다면, 템플릿 패키지 문서를 보라.

당신은 데이터 주도 HTML 텍스트를 포함한 몇 줄 안되는 코드로 쓸만한 웹 서버를 갖게 되었다!. Go는 단 몇 줄 만으로 많은 것들을 할 수 있을 만큼 강력하다.