GOSAMPLES

👑 Go Generics cheatsheet

December 21, 2022

cheatsheet generics generics-intro

Share:

Getting started

Generics release

Generics in Go are available since the version 1.18, released on March 15, 2022.

Generic function

With Generics, you can create functions with types as parameters. Instead of writing separate functions for each type like:

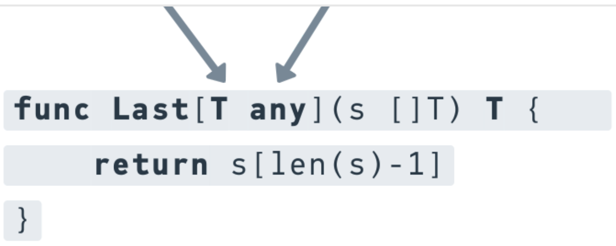
```go
func LastInt(s []int) int {
    return s[len(s)-1]
}

func LastString(s []string) string {
    return s[len(s)-1]
}

// etc.
```

you can write a function with a type parameter:

```go
func Last[T any](s []T) T {
    return s[len(s)-1]
}
```

Type parameters are declared in square brackets. They describe types that are allowed for a given function:

GOSAMPLES



```
func Last[T any](s []T) T {
    return s[len(s)-1]
}
```

Generic function call

You can call a generic function like any other function:

```
func main() {
    data := []int{1, 2, 3}
    fmt.Println(Last(data))

    data2 := []string{"a", "b", "c"}
    fmt.Println(Last(data2))
}
```

You do not have to explicitly declare the type parameter as in the example below, because it is inferred based on the passed arguments. This feature is called *type inference* and applies only to functions.

```
func main() {
    data := []int{1, 2, 3}
    fmt.Println(Last[int](data))

    data2 := []string{"a", "b", "c"}
    fmt.Println(Last[string](data2))
}
```

However, explicitly declaring concrete type parameters is allowed, and sometimes necessary, when the compiler is unable to unambiguously detect the type of arguments passed.

Constraints

Definition

A constraint is an interface that describes a type parameter. Only types that satisfy the specified interface can be used as a parameter of a generic function. The constraint always appears in

```go
func Last[T any](s []T) T {
    return s[len(s)-1]
}
```

the constraint is `any`. Since Go 1.18, `any` is an alias for `interface{}`:

```go
type any = interface{}
```

The `any` is the broadest constraint, which assumes that the input variable to the generic function can be of any type.

Built-in constraints

In addition to the `any` constraint in Go, there is also a built-in `comparable` constraint that describes any type whose values can be compared, i.e., we can use the `==` and `!=` operators on them.

```go
func contains[T comparable](elems []T, v T) bool {
    for _, s := range elems {
        if v == s {
            return true
        }
    }
    return false
}
```

The `constraints` package

More constraints are defined in the `x/exp/constraints` package. It contains constraints that permit, for example, ordered types (types that support the operators `<`, `<=`, `>=`, `>`), floating-point types, integer types, and some others:

```go
func Last[T constraints.Complex](s []T) {}
func Last[T constraints.Float](s []T) {}
func Last[T constraints.Integer](s []T) {}
func Last[T constraints.Ordered](s []T) {}
func Last[T constraints.Signed](s []T) {}
func Last[T constraints.Unsigned](s []T) {}
```

Custom constraints

Constraints are interfaces, so you can use a custom-defined interface as a constraint on a function type parameter:

```
type Doer interface {
    DoSomething()
}

func Last[T Doer](s []T) T {
    return s[len(s)-1]
}
```

However, using such an interface as a constraint is no different from using the interface directly.

As of Go 1.18, the interface definition has a new syntax. Now it is possible to define an interface with a type:

```
type Integer interface {
    int
}
```

Constraints containing only one type have little practical use. But, when combined with the union operator `|`, we can define *type sets* without which complex constraints cannot exist.

Type sets

Using the union `|` operator, we can define an interface with more than one type:

```
type Number interface {
    int | float64
}
```

This type of interface is a *type set* that can contain types or other types sets:

```
type Number interface {
    constraints.Integer | constraints.Float
}
```

Type sets help define appropriate constraints. For example, all constraints in the `x/exp/constraints` package are type sets declared using the union operator:

```
    Signed | Unsigned
}
```

Inline type sets

Type set interface can also be defined inline in the function declaration:

```
func Last[T interface{ int | int8 | int16 | int32 }](s []T) T {
    return s[len(s)-1]
}
```

Using the simplification that Go allows for, we can omit the `interface{}` keyword when declaring an inline type set:

```
func Last[T int | int8 | int16 | int32](s []T) T {
    return s[len(s)-1]
}
```

Type approximation

In many of the constraint definitions, for example in the `x/exp/constraints` package, you can find the special operator `~` before a type. It means that the constraint allows this type, as well as a type whose underlying type is the same as the one defined in the constraint. Take a look at the example:

```
package main

import (
    "fmt"
)

type MyInt int

type Int interface {
    ~int | int8 | int16 | int32
}

func Last[T Int](s []T) T {
    return s[len(s)-1]
}
```

```
    data := []MyInt{1, 2, 3}
    fmt.Println(Last(data))
}
```

Without the `~` before the `int` type in the `Int` constraint, you cannot use a slice of `MyInt` type in the `Last()` function because the `MyInt` type is not in the list of the `Int` constraint. By defining `~int` in the constraint, we allow variables of any type whose underlying type is `int`.

Generic types

Defining a generic type

In Go, you can also create a generic type defined similarly to a generic function:

```
type KV[K comparable, V any] struct {
    Key   K
    Value V
}

func (v *KV[K, V]) Set(key K, value V) {
    v.Key = key
    v.Value = value
}

func (v *KV[K, V]) Get(key K) *V {
    if v.Key == key {
        return &v.Value
    }
    return nil
}
```

Note that the method receiver is a generic `KV[K, V]` type.

When defining a generic type, you cannot introduce additional type parameters in its methods - the struct type parameters are only allowed.

Example of usage

When initializing a new generic struct, you must explicitly provide concrete types:

```
func main() {
    var record KV[string, float64]
    if v != nil {
        fmt.Println(*v)
    }
}
```

You can avoid it by creating a constructor function since types in functions can be inferred thanks to the *type inference* feature:

```go
func NewKV[K comparable, V any](key K, value V) *KV[K, V] {
    return &KV[K, V]{
        Key:   key,
        Value: value,
    }
}

func main() {
    record := NewKV("abc", 54.3)
    v := record.Get("abc")
    if v != nil {
        fmt.Println(*v)
    }
    NewKV("abc", 54.3)
}
```

Share:

Related

GOSAMPLES

Learn how to count elements in a slice that meet certain conditions

introduction generics generics-intro

December 15, 2022

🖖 Calculate Median in Go using Generics

Learn how to find “the middle value” of a slice

numbers math generics generics-intro

April 11, 2022

🤏 Calculate arithmetic mean in Go using Generics

Learn how to calculate mean for a slice of any numeric type

numbers math generics generics-intro

April 8, 2022

About Privacy Policy Cookie preferences Contact RSS