The Why of Iterator Design

Axel Wagner

https://blog.merovius.de/

@Merovius@chaos.social

2024-11-07

Go 1.23 iterators

```
func PrintSquares() {
  for s := range Squares() {
    if s < 10; { continue }
    if s > 100; { break }
    fmt.Println(s)
  }
}
```

```
func PrintSquares() {
  for s := range Squares() {
    if s < 10; { continue }
    if s > 100; { break }
    fmt.Println(s)
  }
}

func Squares() iter.Seq[int]
```

```go
func PrintSquares() {
  for s := range Squares() {
    if s < 10; { continue }
    if s > 100; { break }
    fmt.Println(s)
  }
}

func Squares() iter.Seq[int] {
  return func(yield func(int) bool) {
```

```
func PrintSquares() {
  for s := range Squares() {
    if s < 10; { continue }
    if s > 100; { break }
    fmt.Println(s)
  }
}

func Squares() iter.Seq[int] {
  return func(yield func(int) bool) {
    for i := 0; ; i++ {
          yield(i*i)
```

```go
func PrintSquares() {
  for s := range Squares() {
    if s < 10; { continue }
    if s > 100; { break }
    fmt.Println(s)
  }
}

func Squares() iter.Seq[int] {
  return func(yield func(int) bool) {
    for i := 0; ; i++ {
      if !yield(i*i) {
        return
      }
    }
  }
}
```

```go
func PrintSquares() {
  for i, s := range Squares() {
    if s < 10; { continue }
    if s > 100; { break }
    fmt.Println(i, s)
  }
}

func Squares() iter.Seq2[int, int] {
  return func(yield func(int, int) bool) {
    for i := 0; ; i++ {
      if !yield(i, i*i) {
        return
      }
    }
  }
}
```

This is not an iterator tutorial.

This is not an iterator tutorial.

This is a history lesson.

range

```
var s []T
for range s {}
for i := range s {}
for i, v := range s {}

var m map[K]V
for range m {}
for k := range m {}
for k, v := range m {}

var c chan T
for range c {}
for v := range c {}
```

Channel iterator

```
func Squares() <-chan int {
  ch := make(chan int)
  go func(ch chan<- int) {
    for i := 0; ; i++ {
      ch <- i*i
    }
  }(ch)
  return ch
}
```

Channel iterator

```
func Squares() <-chan int {
  ch := make(chan int)
  go func(ch chan<- int) {
    for i := 0; ; i++ {
      ch <- i*i
    }
  }(ch)
  return ch
}
```

Stopping:

```
package signal

func Notify(c chan<- os.Signal, sig ...os.Signal)
func Stop(c chan<- os.Signal)
```

Iterator types

```
package sql

type Rows struct{ /* … */ }
func (*Rows) Next() bool
func (*Rows) Scan(...any) error
func (*Rows) Close() error
```

Iterator types

```
package sql

type Rows struct{ /* … */ }
func (*Rows) Next() bool
func (*Rows) Scan(...any) error
func (*Rows) Close() error
```

Maps:

```
type Set[E comparable] struct{ m map[E]struct{} }

func (s *Set[E]) Elements() *Iter[E] {
  // needs to range in a goroutine and write to a channel,
  // to enumerate map keys.
}
```

Callbacks

```
type Map[K, V any] struct{ /* … */ }
func (m *Map[K, V]) Range(f func(K, V) bool)
```

Callbacks

```
type Map[K, V any] struct{ /* … */ }
func (m *Map[K, V]) Range(f func(K, V) bool)
```

Break/Continue/Return:

```
func Find[K comparable, V any](m *Map[K, V], key K) (V, bool) {
  var val V
  var found bool
  m.Range(func(k K, v V) bool {
    if k == key {
      val, found = v, true
      return false
    }
  })
  return val, found
}
```

#54245: discussion: standard iterator interface

Interface

```
package iter

type Iter[E any] interface{ Next() (elem E, ok bool) }
```

Interface

```
package iter

type Iter[E any] interface{ Next() (elem E, ok bool) }
```

Use:

```
for v := range it {
  fmt.Println(v)
}
```

Interface

```
package iter

type Iter[E any] interface{ Next() (elem E, ok bool) }
```

Use:

```
for v := range it {
  fmt.Println(v)
}
```

Translated to:

```
for v, _ok := it.Next(); _ok; v, _ok = it.Next() {
  fmt.Println(v)
}
```

Example: Slice

```
func Slice[E any](s []E) iter.Iter[E] {
  return &sliceIter[E]{s: s}
}

type sliceIter[E any] struct {
  s []E
  i int
}

func (it *sliceIter[E]) Next() (v E, ok bool) {
  if it.i < len(i.s) {
    v, ok = it.s[i.i], true
    i.i++
  }
  return v, ok
}
```

Example: Map

```
type Iter2[E1, E2 any] interface{ Next() (E1, E2, bool) }
```

Example: Map

```
type Iter2[E1, E2 any] interface{ Next() (E1, E2, bool) }

func Map[K comparable, V any](m map[K]V) iter.Iter2[K, V] {
  // Non-trivial, channel-based code.
}
```

Generators

```
// NewGen creates a new iterator from a generator function gen.
// The gen function is called once.  It is expected to call
// yield(v) for every value v to be returned by the iterator.
// If yield(v) returns false, gen must stop calling yield and return.
func NewGen[E any](gen func(yield func(E) bool)) StopIter[E]

func NewGen2[E1, E2 any](gen func(yield func(E1, E2) bool)) StopIter2[E1, E2]
```

Stopping

Stopping

Optional interface:

```
type StopIter[E any] interface{
  Iter

  // Stop indicates that the iterator will no longer be used.
  // After a call to Stop, future calls to Next may panic.
  // Stop may be called multiple times;
  // all calls after the first will have no effect.
  Stop()
}
type StopIter2[E1, E2 any] interface{ Iter2; Stop() }
```

Stopping

Optional interface:

```
type StopIter[E any] interface{
  Iter

  // Stop indicates that the iterator will no longer be used.
  // After a call to Stop, future calls to Next may panic.
  // Stop may be called multiple times;
  // all calls after the first will have no effect.
  Stop()
}
type StopIter2[E1, E2 any] interface{ Iter2; Stop() }
```

Convention: Whoever gets a `StopIter`, has to ensure `Stop` is called.

Example: Map (again)

```
func Map[K comparable, V any](m map[K]V) iter.StopIter2[K, V] {
  return iter.NewGen2(func(yield func(K, V) bool) {
    for k, v := range m {
      if !yield(k, v) {
        return
      }
    }
  })
}
```

Extension: Range

```
func F(m *OrderedMap[K, V])
```

Extension: Range

```
func F(m *OrderedMap[K, V]) {
  it := m.Range()




}
```

Extension: Range

```
func F(m *OrderedMap[K, V]) {
  it := m.Range()
  defer it.Stop()



}
```

Extension: Range

```
func F(m *OrderedMap[K, V]) {
  it := m.Range()
  defer it.Stop()
  for k, v := range it {
    fmt.Println(k, v)
  }
}
```

Extension: Range

```
func F(m *OrderedMap[K, V]) {
  it := m.Range()
  defer it.Stop()
  for k, v := range it {
    fmt.Println(k, v)
  }
}
```

If `m` has method `Range() I` and `I` implements `Iter`:

```
func F(m *OrderedMap[K, V]) {
  for k, v := range m { // implicitly calls m.Range()
    fmt.Println(k, v)
  }                     // if I implements StopIter, calls Stop()
}
```

But is it good?

Compatibility

```
type C chan int
func (C) Next() (int, bool) {
  return 0, true
}

func F(c C) {
  for v := range c {
    fmt.Println(v)
  }
}
```

Compatibility

```
type C chan int
func (C) Next() (int, bool) {
  return 0, true
}

func F(c C) {
  for v := range c {
    fmt.Println(v)
  }
}
```

Corollary: If the underlying type is slice, `map` or `chan`, `Next()` is ignored.

Implementing multiple interfaces

Implementing multiple interfaces

1. A type can not implement `Iter` and `Iter2`.

Can't have one shared type to iterate over keys **or** key/value–pairs.

Implementing multiple interfaces

1. A type can not implement `Iter` and `Iter2`.

Can't have one shared type to iterate over keys **or** key/value–pairs.

2. A type can not implement `Range() Iter` and `Range() Iter2`.

The most general is `Range() Iter2`, which is may be less efficient.

Implementing multiple interfaces

1. A type can not implement `Iter` and `Iter2`.

Can't have one shared type to iterate over keys **or** key/value–pairs.

2. A type can not implement `Range() Iter` and `Range() Iter2`.

The most general is `Range() Iter2`, which is may be less efficient.

3. A type can implement both `Iter` and `Range() Iter`.

Ambiguous what `range x` would do: Disallowed by compiler.

StopIter

Generators are often easier and sometimes necessary to write.

But this proposal makes them harder to use, by requiring `Stop`.

#56413: discussion: add range over func

Pull functions

Getting methods out of the way:

```
func() bool
func() (A, bool)
func() (A, B, bool)
```

Pull functions

Getting methods out of the way:

```
func() bool
func() (A, bool)
func() (A, B, bool)
```

Rewrite:

```
for a, b := range f {
  if a == x { continue }
  if b == y { return 42 }
  if a+b == z { break }
}
```

Pull functions

Getting methods out of the way:

```
func() bool
func() (A, bool)
func() (A, B, bool)
```

Rewrite:

```
for a, b := range f {
  if a == x { continue }
  if b == y { return 42 }
  if a+b == z { break }
}
```

→

```
for a, b, _ok := f(); _ok; a, b, _ok = f() {
  if a == x { continue }
  if b == y { return 42 }
  if a+b == z { break }
}
```

Push functions

Making generators first–class:

```
func(yield func() bool)
func(yield func(A) bool)
func(yield func(A, B) bool)
```

```
func(yield func() bool) bool
func(yield func(A) bool) bool
func(yield func(A, B) bool) bool
```

Push functions

Making generators first–class:

```
func(yield func() bool)
func(yield func(A) bool)
func(yield func(A, B) bool)
```

```
func(yield func() bool) bool
func(yield func(A) bool) bool
func(yield func(A, B) bool) bool
```

Rewrite:

```
for a, b := range f {
  if a == x { continue }
  if b == y { return 42 }
  if a+b == z { break }
}
```

Push functions

Making generators first–class:

```
func(yield func() bool)
func(yield func(A) bool)
func(yield func(A, B) bool)
```

```
func(yield func() bool) bool
func(yield func(A) bool) bool
func(yield func(A, B) bool) bool
```

Rewrite:

```
for a, b := range f {
  if a == x { continue }
  if b == y { return 42 }
  if a+b == z { break }
}
```

→

```
_magic()
f(_magic_yield)
_moreMagic()
```

Push functions

Making generators first–class:

```
func(yield func() bool)
func(yield func(A) bool)
func(yield func(A, B) bool)
```

```
func(yield func() bool) bool
func(yield func(A) bool) bool
func(yield func(A, B) bool) bool
```

Rewrite:

```
for a, b := range f {
  if a == x { continue }
  if b == y { return 42 }
  if a+b == z { break }
}
```

→

```
_magic()
f(_magic_yield)
_moreMagic()
```

Corollary: Do **not** persist `yield` from iterator.

Push return value

Push functions have an optional `bool` return, which is ignored.

Push return value

Push functions have an optional `bool` return, which is ignored.

Makes writing some iterators easier:

```
type Node[E any] struct {
  Value E
  Left  *Node[E]
  Right *Node[E]
}
func (n *Node[E]) All(yield func(E) bool) bool {
  if n == nil {
    return true
  }
  return n.Left.All(yield) && yield(n.Value) && n.Right.All(yield)
}
```

#61405: proposal: add range over func

Changes

- Drops pull functions
- Drops half of push functions

Changes

- Drops pull functions
- Drops half of push functions

Separate proposal:

```
package iter

type Seq[V any] func(yield func(V) bool)
type Seq2[K, V any] func(yield func(K, V) bool)

func Pull[V any](s Seq[V]) (next func() (V, bool), stop func())
func Pull2[K, V any](s Seq2[K, V any]) (next func() (K, V, bool), stop func())
```

Thank you