# Um Manual Abrangente sobre Genéricos em Go: Dos Fundamentos aos Padrões Avançados

## Introdução: A Evolução do Polimorfismo em Go

A introdução de genéricos na versão 1.18 da linguagem de programação Go representa a mudança mais substancial em sua história, marcando uma evolução significativa na forma como os desenvolvedores podem abordar o polimorfismo e a reutilização de código.[1] Para compreender plenamente o impacto dessa adição, é imperativo analisar o cenário que a precedeu, dominado pelo uso de interfaces para alcançar um comportamento polimórfico.

### A Era Pré-Genéricos: Polimorfismo via interface{}

Desde o seu início, Go ofereceu uma forma de polimorfismo através de seu sistema de interfaces.[3] Uma interface em Go define um conjunto de métodos, e qualquer tipo que implemente esses métodos satisfaz implicitamente a interface.[3] Uma construção particularmente poderosa e onipresente era a interface vazia, interface{}, que não possui métodos e, portanto, é satisfeita por qualquer tipo. Com a introdução dos genéricos, a interface vazia ganhou um alias mais legível: any.[4]
Essa capacidade de interface{} de conter um valor de qualquer tipo tornou-se a principal ferramenta para escrever código que precisava operar em múltiplos tipos de dados. Funções e estruturas de dados, como contêineres, eram frequentemente projetadas para trabalhar com interface{} para alcançar um grau de generalidade.[5]

#### O Problema: As Desvantagens da Interface Vazia

Apesar de sua flexibilidade, a abordagem baseada em interface{} apresentava desvantagens significativas que geravam atrito no desenvolvimento de software em larga escala.[7]

1. **Falta de Segurança de Tipo (Type Safety):** A desvantagem mais crítica era a perda da segurança de tipo em tempo de compilação. Como o compilador só sabia que uma variável era do tipo interface{}, ele não podia verificar se as operações realizadas nela

eram válidas para o tipo concreto subjacente. Para recuperar o tipo original e trabalhar com ele, os desenvolvedores eram forçados a usar asserções de tipo em tempo de execução (x.(T)) ou seletores de tipo (switch v := x.(type)).[5] Se uma asserção de tipo falhasse durante a execução — por exemplo, se uma função esperasse um int mas recebesse uma string — o programa entraria em pânico. Isso transferia a responsabilidade da verificação de tipos do compilador para o desenvolvedor e do tempo de compilação para o tempo de execução, tornando o código mais frágil e suscetível a erros.[5]

2. **Sobrecarga de Desempenho (Performance Overhead):** O uso de interfaces introduz uma sobrecarga de desempenho. Quando um valor de um tipo concreto (especialmente um tipo de valor como int ou uma struct) é atribuído a uma variável de interface, ocorre um processo chamado "boxing". O valor é encapsulado em uma estrutura interna da interface, o que envolve alocação de memória no heap e uma indireção de ponteiro.[5] Além disso, a chamada de métodos em uma variável de interface requer um "despacho dinâmico" (dynamic dispatch), onde o programa precisa consultar uma tabela virtual (v-table) em tempo de execução para encontrar o ponteiro para a implementação correta do método. Esse processo é inerentemente mais lento do que uma chamada de função direta, que pode ser resolvida em tempo de compilação.[9]
3. **Verbosisade do Código:** O código que dependia de interface{} frequentemente se tornava verboso e menos legível. A necessidade constante de asserções de tipo e o padrão idiomático "comma ok" (valor, ok := i.(string)) para lidar com falhas de asserção adicionavam uma quantidade significativa de código boilerplate, obscurecendo a lógica de negócios principal.[7]

## O Advento dos Genéricos: Um Novo Paradigma

Após anos de demanda da comunidade e um extenso processo de design, Go 1.18, lançado em março de 2022, introduziu formalmente o polimorfismo paramétrico, comumente conhecido como genéricos.[2] Essa adição não foi meramente sintática; ela introduziu três novos conceitos fundamentais na linguagem [12]:

1. **Parâmetros de Tipo** para funções e tipos.
2. A capacidade de **definir interfaces como conjuntos de tipos**, não apenas como conjuntos de métodos.
3. **Inferência de Tipo**, que permite ao compilador deduzir os argumentos de tipo em muitos casos, mantendo o código conciso.

A proposta de valor central dos genéricos é permitir a escrita de funções e tipos de dados que são flexíveis e reutilizáveis, operando em uma variedade de tipos, ao mesmo tempo em que mantêm a total segurança de tipo em tempo de compilação.[7] Isso elimina a necessidade de asserções de tipo em tempo de execução, reduz o código boilerplate e, em muitos casos, melhora o desempenho ao evitar o "boxing" e o despacho dinâmico.
A introdução de genéricos reflete uma evolução pragmática na filosofia de design de Go.

Inicialmente, a simplicidade da linguagem, incluindo a ausência de genéricos, era vista como uma característica que promovia a legibilidade e a manutenibilidade.[13] No entanto, os problemas recorrentes de segurança de tipo e duplicação de código em cenários comuns, como a implementação de estruturas de dados, criaram uma necessidade clara de uma solução mais robusta. A adoção dos genéricos sinaliza um reconhecimento de que, para certas classes de problemas, os benefícios da segurança de tipo em tempo de compilação e da reutilização de código superam o custo de adicionar um novo recurso significativo à linguagem. A simplicidade permanece um valor central, mas não à custa da segurança e da expressividade em padrões de programação fundamentais.

---

# Parte I: Os Fundamentos dos Genéricos em Go

Esta parte serve como uma introdução prática e detalhada à sintaxe e aos mecanismos centrais para a utilização de genéricos em Go, abordando tanto funções quanto tipos genéricos.

## Capítulo 1: Funções Genéricas

Funções genéricas são funções que podem operar em múltiplos tipos. O tipo ou os tipos específicos com os quais a função trabalhará são passados como parâmetros no momento da chamada, permitindo que o compilador realize a verificação de tipo completa.

### Anatomia de uma Função Genérica

A sintaxe de uma função genérica introduz uma lista de parâmetros de tipo, declarada entre colchetes `` imediatamente após o nome da função. A estrutura geral é func NomeDaFuncao(parametro T) T.

- **Parâmetros de Tipo:** A declaração `` define um parâmetro de tipo chamado T. T atua como um espaço reservado (placeholder) para um tipo concreto que será fornecido quando a função for chamada.[7] Por convenção,
  T é usado como o nome para o primeiro parâmetro de tipo, mas qualquer identificador válido pode ser utilizado.[4]
- **Restrições (Constraints):** A Restricao (neste caso, any) especifica as propriedades que o tipo concreto fornecido para T deve ter. Uma restrição é uma interface que define o conjunto de tipos permitidos para o parâmetro de tipo. A restrição any é a menos restritiva, permitindo que qualquer tipo seja usado.[7]
- **Uso de Parâmetros de Tipo:** Uma vez declarado, o parâmetro de tipo T pode ser usado como qualquer outro tipo na assinatura da função, para definir os tipos dos parâmetros de entrada e dos valores de retorno.

## "Olá, Mundo" dos Genéricos

Um exemplo simples demonstra o conceito. A função Print a seguir pode aceitar e imprimir um valor de qualquer tipo, mantendo a segurança de tipo.

Go

```
package main

import "fmt"

// Print é uma função genérica que imprime o valor de qualquer tipo.
func Print(value T) {
    fmt.Println(value)
}

func main() {
    Print(42)               // Instanciada com T = int
    Print("olá, genéricos")    // Instanciada com T = string
    Print(struct{ name string }{"Go"}) // Instanciada com T = struct{ name string }
}
```

Neste exemplo, Print é chamada com um int, uma string e uma struct. Em cada chamada, o compilador infere o tipo concreto para T e garante que o uso dentro da função (fmt.Println) seja válido para aquele tipo.[7]

## Exemplo Prático: Função Sum Genérica

Um dos casos de uso mais claros para genéricos é a eliminação de duplicação de código. Antes dos genéricos, para somar os valores de um mapa de int64 e um mapa de float64, seriam necessárias duas funções quase idênticas:

Go

```
// Abordagem não genérica
func SumInts(m map[string]int64) int64 {
    var s int64
    for _, v := range m {
```

```
        s += v
    }
    return s
}

func SumFloats(m map[string]float64) float64 {
    var s float64
    for _, v := range m {
        s += v
    }
    return s
}
```

A lógica é a mesma; apenas os tipos mudam. Com genéricos, podemos unificar isso em uma única função.[13]

Go

```
// SumNumbers soma os valores de um mapa.
// Ele suporta tanto int64 quanto float64 como tipos de valor.
func SumNumbers[K comparable, V int64 | float64](m map[K]V) V {
    var s V
    for _, v := range m {
        s += v
    }
    return s
}
```

Aqui, K é o tipo da chave (restringido a comparable, pois as chaves de mapa devem ser comparáveis) e V é o tipo do valor. A restrição int64 | float64 é uma união de tipos, significando que V pode ser int64 ou float64. O operador + é válido para ambos os tipos, então o código compila com segurança.

### Inferência de Tipo em Ação

Uma característica poderosa dos genéricos em Go é a inferência de tipo. Na maioria das vezes, não é necessário especificar explicitamente os argumentos de tipo ao chamar uma função genérica. O compilador pode deduzi-los a partir dos tipos dos argumentos regulares passados para a função.[7]

Go

```
func main() {
    ints := map[string]int64{"primeiro": 34, "segundo": 12}
    floats := map[string]float64{"primeiro": 35.98, "segundo": 26.99}

    // O compilador infere K=string, V=int64 a partir do argumento 'ints'
    fmt.Printf("Soma genérica de inteiros: %v\n", SumNumbers(ints))

    // O compilador infere K=string, V=float64 a partir do argumento 'floats'
    fmt.Printf("Soma genérica de floats: %v\n", SumNumbers(floats))
}
```

Essa inferência torna o código de chamada tão limpo e legível quanto o de uma função não genérica.[14] No entanto, a especificação explícita dos tipos ainda é possível e, às vezes, necessária para desambiguação ou quando a inferência falha.[15]

Go

```
// Chamada explícita (geralmente desnecessária)
_ = SumNumbers[string, int64](ints)
```

## Capítulo 2: Tipos Genéricos

Além das funções, Go permite a definição de tipos genéricos. Isso é particularmente útil para a criação de estruturas de dados de propósito geral, como listas, árvores ou pilhas, que podem armazenar elementos de qualquer tipo de maneira segura.

### Definindo Estruturas de Dados Genéricas

A sintaxe para adicionar parâmetros de tipo a uma definição de struct é semelhante à de funções, com a lista de parâmetros de tipo seguindo o nome do tipo: type NomeDoTipo struct {... }.[8] Este é um dos casos de uso primários para genéricos, permitindo a criação de coleções e contêineres seguros em termos de tipo que não são nativos da linguagem.[17]

### Estudo de Caso: Uma Stack Segura em Tipo

Uma pilha (stack) é uma estrutura de dados LIFO (Last-In, First-Out) clássica. Uma implementação genérica permite criar uma pilha para qualquer tipo de dado sem duplicação de código.

Go

```
package main

import "fmt"

// Stack é uma estrutura de dados de pilha genérica.
type Stack struct {
    itemsT
}

// Push adiciona um item ao topo da pilha.
func (s *Stack) Push(item T) {
    s.items = append(s.items, item)
}

// Pop remove e retorna o item do topo da pilha.
// Retorna o valor zero do tipo T e false se a pilha estiver vazia.
func (s *Stack) Pop() (T, bool) {
    if len(s.items) == 0 {
        var zero T
        return zero, false
    }
    index := len(s.items) - 1
    item := s.items[index]
    s.items = s.items[:index]
    return item, true
}

func main() {
    // Criando e usando uma pilha de inteiros
    intStack := Stack[int]{}
    intStack.Push(10)
    intStack.Push(20)
    val, _ := intStack.Pop()
    fmt.Println("Popped da pilha de inteiros:", val) // Saída: 20
```

```
    // Criando e usando uma pilha de strings
    stringStack := Stack[string]{}
    stringStack.Push("Go")
    stringStack.Push("Generics")
    strVal, _ := stringStack.Pop()
    fmt.Println("Popped da pilha de strings:", strVal) // Saída: Generics

    // O seguinte código resultaria em um erro de compilação:
    // intStack.Push("não é um int") // erro: cannot use "não é um int" (untyped string constant)
as int value in argument to intStack.Push
}
```

A instanciação de Stack[int] cria um tipo de pilha que só aceita int. O compilador impõe essa regra, prevenindo erros de tipo em tempo de compilação, uma melhoria drástica em relação às estruturas de dados baseadas em interface{}.[8]

### Implementando Métodos em Tipos Genéricos

Ao definir métodos para um tipo genérico, é crucial incluir o parâmetro de tipo na declaração do receptor. O método opera em uma instância de Stack, não em um tipo abstrato Stack.[11] A sintaxe correta é
func (s *Stack) NomeDoMetodo(...).
A necessidade dessa sintaxe revela um aspecto fundamental do modelo de implementação de genéricos em Go. Tipos como Stack[int] e Stack[string] não são apenas variações de um único tipo em tempo de execução; eles são tratados pelo compilador como tipos concretos e distintos, cada um com seu próprio conjunto de métodos especializados. Isso está em contraste direto com as estruturas baseadas em interface{}, onde o contêiner armazena um único tipo de interface, e a identidade dos tipos concretos subjacentes só é conhecida em tempo de execução. A sintaxe dos métodos, portanto, reflete diretamente o modelo de polimorfismo em tempo de compilação (frequentemente chamado de monomorfização ou stenciling), onde tipos especializados são gerados antes da execução do programa.

---

# Parte II: Dominando as Restrições de Tipo

O sistema de restrições (constraints) é o pilar que confere segurança e expressividade ao código genérico em Go. Ele define as regras e capacidades que um tipo deve possuir para ser usado como argumento de tipo, permitindo que o compilador verifique a validade das operações dentro do código genérico.

## Capítulo 3: O Papel e a Definição das Restrições

Uma restrição atua como um "contrato" para um parâmetro de tipo. Ela especifica o conjunto de propriedades — como métodos que devem ser implementados ou operações que devem ser suportadas (por exemplo, comparação, adição) — que um argumento de tipo deve satisfazer para que o código genérico seja compilado com sucesso.[8]

### Definindo Restrições com Interfaces

Um dos aspectos mais elegantes do design de genéricos em Go é que uma restrição de tipo *é* uma interface.[12] Essa decisão unifica o novo recurso com um conceito já existente e bem compreendido na linguagem, em vez de introduzir um sistema completamente novo.
Assim como uma interface padrão, uma restrição pode especificar um conjunto de métodos necessários. Qualquer tipo que implemente esses métodos satisfará a restrição.

Go

```
package main

import "fmt"

// Stringer é uma interface que pode ser usada como uma restrição.
// Requer que o tipo tenha um método String() string.
type Stringer interface {
    String() string
}

// PrintStrings usa a restrição Stringer para garantir que cada item
// em 's' possa ser convertido para uma string através do método String().
func PrintStrings(sT) {
    for _, v := range s {
        fmt.Println(v.String())
    }
}

type MyType int

func (m MyType) String() string {
    return fmt.Sprintf("MyType value is: %d", m)
```

```
}

func main() {
   PrintStrings(MyType{10, 20, 30})
}
```

Neste exemplo, PrintStrings é restrita a tipos que satisfazem a interface Stringer. Como MyType implementa o método String(), um slice de MyType é um argumento válido.[21]

## Capítulo 4: Restrições Pré-definidas e Personalizadas

Go fornece algumas restrições pré-declaradas para casos de uso comuns e permite a criação de restrições personalizadas com uma sintaxe expandida para interfaces.

### A Restrição any

A restrição any é um identificador pré-declarado que é um alias para a interface vazia interface{}.[4] É a restrição menos específica, permitindo que qualquer tipo seja usado como argumento de tipo. É ideal para contêineres genéricos, como a
Stack vista anteriormente, que armazenam e recuperam valores sem operar neles.[7]

### A Restrição comparable

comparable é uma restrição especial pré-declarada que é satisfeita por todos os tipos que podem ser comparados usando os operadores == e !=.[7] Isso inclui a maioria dos tipos básicos como booleanos, números, strings e ponteiros, bem como structs cujos campos são todos comparáveis. Tipos como slices, mapas e funções não são comparáveis. Essa restrição é essencial ao escrever código genérico que precisa, por exemplo, usar tipos como chaves de um mapa.[14]

Go

```
// MapKeys retorna as chaves de um mapa. O tipo da chave K
// deve ser 'comparable' para ser uma chave de mapa válida.
func MapKeys[K comparable, V any](m map[K]V)K {
   keys := make(K, 0, len(m))
   for k := range m {
      keys = append(keys, k)
```

```
    }
    return keys
}
```

### Restrições de União: Um Conjunto de Tipos

Go expande a definição de interfaces para permitir a especificação de um conjunto finito de tipos permitidos, usando o operador | (pipe) para criar uma união de tipos.[7] Isso é extremamente útil para funções que operam sobre um conjunto específico de tipos, como tipos numéricos.

Go

```
// Number é uma restrição de tipo que permite qualquer int ou float64.
type Number interface {
    int | int64 | float32 | float64
}

// Sum soma um slice de qualquer tipo que satisfaça a restrição Number.
func Sum(numbersT) T {
    var total T
    for _, n := range numbers {
        total += n
    }
    return total
}
```

Esta abordagem é mais segura e explícita do que depender de uma restrição baseada em métodos, pois define exatamente quais tipos são permitidos.[14]

### O Til (~) para Tipos Subjacentes

O token ~ (til) em uma restrição relaxa a correspondência de tipos para incluir não apenas o tipo nomeado, mas também qualquer outro tipo que tenha o primeiro como seu *tipo subjacente* (underlying type).[4] Isso aumenta a flexibilidade e a usabilidade das APIs genéricas. Considere o seguinte cenário:

Go

```
type MyInt int

// Sem o ~, a função Sum não aceitariaMyInt.
type Integer interface {
   ~int | ~int64 // O ~ permite MyInt, que tem 'int' como tipo subjacente.
}

func SumIntegers(numbersT) T {
   //...
}
```

Sem o ~, a restrição int | int64 só aceitaria os tipos int e int64 literais. Com ~int | ~int64, a restrição também aceita tipos como type MyInt int ou type UserID int64, tornando a função genérica muito mais útil em bases de código do mundo real.[20]

### O Pacote constraints

Para conveniência, a equipe do Go forneceu um pacote experimental, golang.org/x/exp/constraints, que define interfaces para conjuntos comuns de tipos. Por exemplo, constraints.Ordered inclui todos os tipos inteiros, de ponto flutuante e strings que suportam os operadores de ordenação <, >, <=, e >=.[12]
*Nota: Com o tempo, algumas dessas restrições, como cmp.Ordered, foram movidas para a biblioteca padrão, no pacote cmp.*

Go

```
import "cmp" // A partir do Go 1.21

// Min retorna o menor de dois valores ordenáveis.
func Min(a, b T) T {
   if a < b {
      return a
   }
   return b
}
```

O uso dessas restrições pré-definidas economiza tempo e torna o código mais claro e idiomático.[17]

A decisão de Go de implementar restrições como interfaces é um exemplo notável de design de linguagem que prioriza a consistência e a integração. Em vez de introduzir um conceito totalmente novo para restrições genéricas, os designers estenderam o conceito existente de interfaces. Uma interface em Go sempre representou um *conjunto de tipos* — o conjunto de todos os tipos que implementam seus métodos. A nova sintaxe simplesmente tornou essa definição mais explícita, permitindo que uma interface também defina um conjunto de tipos por meio de uma união (int | string). Ao construir sobre o conhecimento que os desenvolvedores de Go já possuem, a curva de aprendizado para genéricos foi suavizada, e o novo recurso parece uma extensão natural da linguagem, em vez de um acréscimo estranho.

---

# Parte III: Uma Análise Comparativa: Genéricos vs. Interfaces

A introdução de genéricos não torna as interfaces obsoletas. Em vez disso, Go agora oferece duas formas distintas de polimorfismo, cada uma com seus próprios pontos fortes, casos de uso e trade-offs. Compreender quando utilizar cada ferramenta é crucial para escrever código Go eficaz e idiomático.

## Capítulo 5: Segurança de Tipo e Legibilidade

A diferença mais fundamental entre genéricos e interfaces reside em *quando* a verificação de tipo ocorre.

### Garantias em Tempo de Compilação vs. Tempo de Execução

- **Genéricos:** Fornecem polimorfismo em tempo de compilação (estático). Todas as verificações de tipo são realizadas pelo compilador antes que o programa seja executado. Se você tentar usar um tipo que não satisfaz a restrição de uma função genérica, ou realizar uma operação inválida no tipo genérico, receberá um erro de compilação. Isso resulta em um código inerentemente mais robusto e seguro, pois uma classe inteira de erros de tipo é eliminada antes que o código chegue à produção.[7]
- **Interfaces:** Fornecem polimorfismo em tempo de execução (dinâmico). Embora o compilador verifique se um tipo implementa os métodos de uma interface no momento da atribuição, o trabalho de descobrir o tipo concreto subjacente de uma variável de interface e usar seus dados específicos requer verificações em tempo de execução, como asserções de tipo.[5] Se uma asserção falhar, ocorrerá um pânico em tempo de execução. Isso adia a detecção de erros para a fase de teste ou, pior, para a produção.[10]

### Clareza e Intenção do Código

A sintaxe dos genéricos torna a intenção do código mais explícita e autodescritiva.

- Uma assinatura de função genérica como func Process(itemsT) comunica claramente suas intenções: ela opera em um slice *homogêneo* (todos os elementos são do mesmo tipo T) de itens que são ordenáveis. O contrato é claro e imposto pelo compilador.
- Uma assinatura baseada em interface como func Process(itemsany) é ambígua. Ela não informa ao chamador quais tipos são realmente esperados ou que operações serão realizadas. Para entender seus requisitos, é necessário ler a documentação ou o código-fonte da função, que provavelmente conterá um seletor de tipo ou asserções de tipo.

## Capítulo 6: Análise Profunda de Desempenho

As diferenças no modelo de implementação de genéricos e interfaces têm implicações diretas no desempenho.

### O Modelo de Desempenho da Interface

O desempenho do código baseado em interface é caracterizado por dois fatores principais:

1. **Boxing:** Quando um tipo de valor (como int, float64 ou uma struct) é atribuído a uma variável de interface, o Go precisa alocar memória no heap para "encaixotar" (box) esse valor. A variável de interface então armazena um ponteiro para o tipo e um ponteiro para os dados no heap. Esse processo de alocação e a indireção resultante introduzem uma sobrecarga de memória e de CPU.[5]
2. **Despacho Dinâmico (Dynamic Dispatch):** Chamar um método em uma variável de interface não é uma chamada de função direta. Em tempo de execução, o sistema precisa consultar a tabela de métodos do tipo (v-table) para encontrar o endereço da implementação correta do método. Essa busca, embora rápida, é mais lenta do que uma chamada de função estática que o compilador pode resolver diretamente.[9]

### O Modelo de Implementação de Genéricos

Go adota uma abordagem híbrida para implementar genéricos, buscando um equilíbrio entre o desempenho em tempo de execução e o tamanho do binário.

1. **Stenciling por GCshape:** O compilador gera versões especializadas (um processo chamado "stenciling" ou monomorfização) de uma função genérica, mas não para cada

tipo único. Em vez disso, ele agrupa os tipos com base em sua forma de memória para o coletor de lixo (GCshape). Por exemplo, int e string têm GCshapes diferentes e, portanto, o compilador gerará duas implementações distintas da função genérica. Isso permite otimizações específicas do tipo e chamadas de função diretas, eliminando a sobrecarga do despacho dinâmico.[9]

2. **Dicionários:** Para tipos que compartilham o mesmo GCshape (notavelmente, todos os tipos de ponteiro e todos os tipos de interface), o compilador gera uma única implementação de código. Para lidar com as diferenças entre os tipos, ele passa um argumento oculto chamado "dicionário". Este dicionário é uma estrutura de dados gerada pelo compilador que contém informações específicas do tipo, como ponteiros para métodos ou informações de tipo necessárias para outras operações.[9]

### Análise de Benchmark e Conclusões

- **Genéricos vs. any (interface{}):** Em cenários que anteriormente exigiriam o uso de any seguido de asserções de tipo (por exemplo, uma estrutura de dados de contêiner), os genéricos são quase sempre significativamente mais rápidos. Eles eliminam completamente a sobrecarga de alocação do "boxing" e as verificações de tipo em tempo de execução.[21]
- **A Armadilha de Desempenho:** Existe um cenário contraintuitivo em que o código genérico pode ser *mais lento* do que o código de interface equivalente. Isso ocorre quando uma função genérica é restrita por uma interface e chama um método dessa interface em seu corpo. A chamada pode envolver duas camadas de indireção: primeiro, a função genérica usa seu dicionário para encontrar informações sobre o tipo T; em seguida, a partir dessas informações, ela realiza o despacho dinâmico da interface para chamar o método. Uma função não genérica que aceita diretamente a interface realizaria apenas a segunda etapa (o despacho dinâmico).[9]

## Capítulo 7: Escolhendo a Ferramenta Certa: Um Guia de Decisão

Com base na análise anterior, podemos estabelecer diretrizes claras sobre quando usar cada recurso.

### Quando Preferir Interfaces

1. **Coleções Heterogêneas:** O caso de uso principal para interfaces é quando você precisa de uma coleção de tipos diferentes que compartilham um comportamento comum. Por exemplo, um io.Writer pode conter um *os.File, um *bytes.Buffer e uma conexão de rede, todos tratados uniformemente através da interface io.Writer.

Genéricos são para coleções *homogêneas*, onde todos os elementos são do mesmo tipo T.[23]

2. **Polimorfismo em Tempo de Execução:** Interfaces são a ferramenta correta quando o tipo concreto de um objeto não é conhecido em tempo de compilação e precisa ser determinado ou trocado dinamicamente em tempo de execução. Isso é comum em arquiteturas de plugins, injeção de dependência e qualquer cenário onde o comportamento precisa ser desacoplado da implementação concreta.[27]

### Quando Preferir Genéricos

1. **Coleções e Estruturas de Dados Homogêneas:** Para qualquer estrutura de dados de contêiner (sejam slices, mapas, canais, listas, árvores, etc.) onde todos os elementos devem ser do mesmo tipo, genéricos são a escolha superior. Eles fornecem segurança de tipo total sem a sobrecarga de any.[17]
2. **Algoritmos de Propósito Geral:** Para implementar algoritmos que executam a mesma lógica em diferentes tipos de dados, como funções de ordenação, busca ou transformações de dados (map, filter, reduce). Genéricos permitem que esses algoritmos sejam escritos uma vez e reutilizados de forma segura.[8]
3. **Código Crítico de Desempenho (para evitar any):** Quando a sobrecarga de "boxing" e asserções de tipo de uma solução baseada em any se torna um gargalo de desempenho, genéricos oferecem uma alternativa muito mais eficiente.[25]

Uma regra prática que emergiu da comunidade é: "NÃO reescreva APIs baseadas em interface para usar Genéricos. NÃO passe uma interface para uma função genérica".[26] Isso encapsula a ideia de que os dois sistemas resolvem problemas diferentes e misturá-los sem cuidado pode levar a um código menos performático e mais complexo.

### Matriz de Decisão: Genéricos vs. Interfaces

A tabela a seguir resume as principais diferenças e ajuda a orientar a decisão entre genéricos e interfaces.

| Característica | **Genéricos** | **Interfaces** |
|---|---|---|
| **Tipo de Polimorfismo** | Em Tempo de Compilação (Paramétrico) | Em Tempo de Execução (Subtipo) |
| **Verificação de Tipo** | Em tempo de compilação, prevenindo erros de tipo em tempo de execução. | Em tempo de execução, via asserções de tipo ou seletores de tipo. |
| **Modelo de Desempenho** | Stenciling/Dicionários. Evita despacho dinâmico e "boxing". | Despacho dinâmico via v-table. Incorre em "boxing" para tipos de valor. |
| **Geração de Código** | Código especializado gerado | Implementação única funciona |

| | | |
|---|---|---|
| | para cada GCshape distinta. | com qualquer tipo concreto que a satisfaça. |
| **Caso de Uso Principal** | Contêineres seguros em tipo, algoritmos em dados **homogêneos**. | Abstração de comportamento, manipulação de dados **heterogêneos**. |
| **Flexibilidade** | Tipos concretos são fixados em tempo de compilação para uma dada instanciação. | Tipos concretos podem ser trocados em tempo de execução. |

---

# Parte IV: Aplicação Prática e Padrões Idiomáticos

Esta seção foca na aplicação de genéricos para resolver problemas do mundo real, demonstrando padrões de código que se tornaram idiomáticos desde a introdução do recurso.

## Capítulo 8: Casos de Uso Comuns

Genéricos encontraram aplicação imediata em duas áreas principais: funções de utilidade para tipos de contêineres nativos e a implementação de estruturas de dados de propósito geral.

### Funções de Utilidade para Slices e Mapas

Uma das vitórias mais rápidas dos genéricos foi a capacidade de escrever funções de utilidade simples e seguras para manipular slices e mapas, eliminando a necessidade de repetir a mesma lógica para cada tipo de elemento.

- **MapKeys:** Uma função para extrair todas as chaves de um mapa. Antes dos genéricos, isso exigiria reflexão ou código específico para cada tipo de mapa.[18]
  Go
  ```
  // MapKeys retorna um slice com todas as chaves do mapa m.
  func MapKeys[K comparable, V any](m map[K]V)K {
     keys := make(K, 0, len(m))
     for k := range m {
        keys = append(keys, k)
     }
     return keys
  }
  ```

- **Filter:** Uma função para criar um novo slice contendo apenas os elementos que

satisfazem um predicado.[7]

```
Go
// Filter retorna um novo slice contendo apenas os elementos
// para os quais a função predicado f retorna true.
func Filter(sT, f func(T) bool)T {
    var resultT
    for _, v := range s {
        if f(v) {
            result = append(result, v)
        }
    }
    return result
}
```

- **Contains:** Uma função para verificar se um elemento existe em um slice.[17]

```
Go
// Contains verifica se o elemento 'elem' está presente no slice 's'.
func Contains(sT, elem T) bool {
    for _, v := range s {
        if v == elem {
            return true
        }
    }
    return false
}
```

### Estruturas de Dados de Propósito Geral

Como explorado anteriormente com a Stack, genéricos são a maneira padrão e correta de implementar estruturas de dados fundamentais em Go moderno. Isso inclui listas ligadas, árvores binárias, heaps, e filas. A capacidade de definir uma estrutura como type LinkedList permite que ela seja reutilizada em toda uma base de código com segurança de tipo total, um avanço monumental em relação às alternativas pré-genéricos, que eram a duplicação de código ou o uso inseguro de any.[15]

## Capítulo 9: Padrões de Programação Funcional

A introdução de genéricos tornou viável e idiomático o uso de padrões de programação funcional de ordem superior para manipulação de coleções. As funções Map, Filter e Reduce

são os pilares deste estilo.

### Implementando Map, Filter e Reduce

Estas três funções formam um kit de ferramentas poderoso para processamento de dados declarativo.

- **Map:** Transforma cada elemento de um slice em um novo elemento, possivelmente de um tipo diferente.[28]
  Go
  ```
  // Map aplica a função f a cada elemento do slice de entrada
  // e retorna um novo slice com os resultados.
  func Map(sliceT, f func(T) R)R {
     result := make(R, len(slice))
     for i, v := range slice {
        result[i] = f(v)
     }
     return result
  }
  ```

- **Filter:** A implementação já foi mostrada no capítulo anterior. Ela seleciona elementos com base em uma condição booleana.[28]
- **Reduce:** Agrega todos os elementos de um slice em um único valor, começando com um valor inicial.[28]
  Go
  ```
  // Reduce aplica a função f cumulativamente aos elementos do slice,
  // da esquerda para a direita, para reduzi-lo a um único valor.
  func Reduce(sliceT, initial R, f func(R, T) R) R {
     result := initial
     for _, v := range slice {
        result = f(result, v)
     }
     return result
  }
  ```

### Compondo Funções Genéricas

A verdadeira força desses utilitários reside em sua capacidade de serem compostos para criar pipelines de processamento de dados limpos e expressivos. Isso permite que a lógica de negócios seja declarada de forma concisa, em vez de ser ofuscada por loops for

imperativos.[28]
Exemplo: Dado um slice de números, encontre a soma dos quadrados dos números pares.

Go

```
func main() {
   numbers :=int{1, 2, 3, 4, 5, 6}

   // Pipeline: Filter -> Map -> Reduce
   sumOfSquaresOfEvens := Reduce(
      Map(
         Filter(numbers, func(n int) bool { return n%2 == 0 }), //
         func(n int) int { return n * n },                      //
      ),
      0, // Valor inicial para a soma
      func(acc, n int) int { return acc + n },                  // 0+4=4, 4+16=20, 20+36=56
   )

   fmt.Println(sumOfSquaresOfEvens) // Saída: 56
}
```

**Bibliotecas Externas**

A utilidade desses padrões foi rapidamente reconhecida pela comunidade Go. Bibliotecas como samber/lo surgiram, oferecendo um conjunto rico e otimizado de utilitários genéricos inspirados em bibliotecas como Lodash do JavaScript.[30] O uso dessas bibliotecas pode acelerar o desenvolvimento e fornecer implementações robustas para operações comuns.
A chegada dos genéricos está provocando uma mudança sutil, mas significativa, no estilo idiomático de Go para manipulação de dados. Antes, a implementação de funções como Map e Filter era impraticável, exigindo geração de código ou o uso inseguro de any com reflexão, ambas abordagens consideradas não idiomáticas.[30] O loop
for imperativo era a única abordagem padrão. Agora, genéricos fornecem a segurança de tipo e o desempenho necessários para tornar essas funções de ordem superior práticas e eficientes. A rápida popularidade de bibliotecas como lo demonstra um forte interesse dos desenvolvedores por esses padrões. Consequentemente, embora os loops for permaneçam a ferramenta padrão para tarefas simples, os desenvolvedores agora têm uma alternativa viável e idiomática para transformações de dados complexas, expandindo o kit de ferramentas expressivo da linguagem e permitindo um estilo de programação mais declarativo que antes era desencorajado.

# Parte V: Tópicos Avançados e Internos

Esta seção final é destinada a desenvolvedores que buscam um domínio completo de genéricos, cobrindo padrões complexos, os detalhes da implementação do compilador e as limitações atuais do recurso.

## Capítulo 10: Interfaces Genéricas

Um conceito avançado e poderoso é que as próprias interfaces podem ser genéricas, ou seja, podem ter parâmetros de tipo. Isso abre novas possibilidades para expressar restrições complexas e projetar APIs flexíveis.

### Definindo Interfaces com Parâmetros de Tipo

A sintaxe é direta: uma interface pode declarar uma lista de parâmetros de tipo, assim como uma struct ou função.[32]

Go

```
// Comparer é uma interface genérica. Um tipo que implementa Comparer
// declara que pode se comparar a um valor do tipo T.
type Comparer interface {
    Compare(T) int // Retorna -1, 0, ou 1
}
```

Esta interface Comparer descreve toda uma família de interfaces, uma para cada tipo com o qual pode ser instanciada.

### O Padrão de Restrição Auto-Referencial

Um dos padrões mais poderosos que as interfaces genéricas permitem é a restrição auto-referencial. Isso resolve um problema de longa data em Go relacionado a padrões como Builder ou Cloner, onde um método precisa retornar uma instância do tipo receptor concreto, não do tipo da interface.
Considere uma interface Cloner que precisa garantir que o método Clone() retorne o mesmo tipo que o está implementando. Antes dos genéricos, isso era impossível de expressar de

forma segura.[33]

Go

```
// Cloner é uma interface genérica para tipos que podem se clonar.
type Cloner interface {
   Clone() T
}

// Cloneable é uma restrição que usa o padrão auto-referencial.
// Ela restringe um tipo T a ser um tipo que implementa Cloner.
// Em outras palavras, T deve ter um método Clone() que retorna T.
type Cloneable] interface {
   // Esta interface pode ter outros métodos...
}

// CloneAny é uma função genérica que pode clonar qualquer tipo
// que satisfaça a restrição auto-referencial.
func CloneAny](v T) T {
   return v.Clone()
}

type MyStruct struct {
   value int
}

// MyStruct implementa Cloner, satisfazendo a restrição.
func (s *MyStruct) Clone() *MyStruct {
   return &MyStruct{value: s.value}
}

func main() {
   s1 := &MyStruct{value: 10}
   s2 := CloneAny(s1) // s2 é do tipo *MyStruct, não de um tipo de interface.
   s2.value = 20
}
```

Este padrão, ], garante que o tipo retornado por Clone() seja o tipo concreto, preservando a segurança de tipo sem a necessidade de asserções.

### Usando Interfaces Genéricas para Abstrair sobre Implementações

Interfaces genéricas também podem ser usadas para definir um contrato para uma estrutura de dados genérica. Isso permite que os usuários de uma API escolham a implementação concreta que melhor se adapta às suas necessidades, mantendo a interoperabilidade.

Go

```
// Set é uma interface genérica que define o comportamento de um conjunto.
type Set[E comparable] interface {
   Insert(E)
   Delete(E)
   Has(E) bool
}

// A função CountUnique usa a interface Set para contar elementos únicos.
// Ela pode funcionar com qualquer implementação de Set (ex: baseada em hash, baseada em árvore).
func CountUnique[E comparable](itemsE, set Set[E]) int {
   count := 0
   for _, item := range items {
      if!set.Has(item) {
         set.Insert(item)
         count++
      }
   }
   return count
}
```

Isso desacopla o algoritmo (CountUnique) da estrutura de dados específica, permitindo flexibilidade e reutilização.[32]

## Capítulo 11: Sob o Capô: Um Olhar Mais Profundo

Compreender como os genéricos são implementados pelo compilador ajuda a raciocinar sobre seu desempenho e trade-offs.

### Revisitando o Stenciling por GCshape

Como mencionado anteriormente, o compilador Go não monomorfiza para cada tipo. Ele usa

o conceito de GCshape, que agrupa tipos com base em seu layout de memória. Dois tipos têm o mesmo GCshape se tiverem o mesmo tipo subjacente ou se ambos forem tipos de ponteiro.[9]

- **Tipos com GCshapes diferentes (ex: int, string, struct{ a int; b bool }):** O compilador gera ("stencils") uma versão separada e especializada do código genérico para cada GCshape. Isso resulta em código altamente otimizado, sem indireções.
- **Tipos com o mesmo GCshape (ex: *int, *string, any):** O compilador gera uma única versão do código e passa um "dicionário" oculto. Este dicionário contém ponteiros de função e metadados de tipo necessários para que o código único opere corretamente no tipo específico em tempo de execução.[9]

### Implicações para o Tamanho do Binário e a Velocidade de Compilação

Esta abordagem híbrida representa um trade-off deliberado:

- **Tamanho do Binário:** O uso de genéricos com muitos tipos de GCshapes diferentes pode levar a um aumento no tamanho do binário executável, pois o compilador duplica o código para cada forma.[7] Este é o custo pago para obter um desempenho de tempo de execução próximo ao do código não genérico escrito à mão.
- **Velocidade de Compilação:** A geração de múltiplas versões do código pode aumentar ligeiramente o tempo de compilação.

O modelo de Go se posiciona entre os extremos de outras linguagens. Linguagens como C++ e Rust usam monomorfização completa, o que pode resultar em binários maiores, mas com desempenho máximo ("zero-cost abstractions"). Linguagens como Java usam "type erasure", onde as informações de tipo genérico são apagadas após a compilação, resultando em binários menores, mas com alguma sobrecarga em tempo de execução e certas limitações.[9] A abordagem de Go busca um meio-termo pragmático.

## Capítulo 12: Limitações Conhecidas e Direções Futuras

Os genéricos em Go são uma adição poderosa, mas a implementação inicial (Go 1.18) veio com algumas limitações.

### Estado Atual dos Genéricos

- **Métodos não podem ter seus próprios parâmetros de tipo:** Um método de uma struct (genérica ou não) não pode declarar seus próprios parâmetros de tipo.
- **Inferência de tipo pode falhar:** Embora a inferência de tipo seja robusta, existem cenários complexos, especialmente envolvendo interfaces genéricas, onde ela pode falhar, exigindo que o desenvolvedor forneça os argumentos de tipo explicitamente.[34]

- **Limitações em restrições:** Existem certas operações que não podem ser expressas diretamente em restrições, como a capacidade de criar um novo valor do tipo T (o que exigiria uma restrição do tipo newable).

A comunidade e a equipe de Go continuam a explorar e refinar o recurso. A linguagem é um sistema vivo, e é provável que futuras versões de Go suavizem algumas dessas arestas.

### A Evolução da Especificação

Um exemplo da evolução contínua é a remoção do conceito de "tipo central" (core type) da especificação da linguagem a partir do Go 1.25. O "tipo central" era uma regra complexa usada para inferência de tipo em restrições. Ele foi substituído por uma prosa mais explícita e regras equivalentes, tornando a especificação mais simples e fácil de entender para os desenvolvedores, sem alterar o comportamento do código existente.[35] Isso demonstra o compromisso da equipe de Go em refinar e simplificar a linguagem mesmo após a introdução de recursos complexos.

# Conclusão: Melhores Práticas e Recursos Selecionados

Genéricos são uma ferramenta transformadora no ecossistema Go, oferecendo um novo nível de segurança de tipo e reutilização de código. No entanto, como qualquer ferramenta poderosa, eles devem ser usados com discernimento. Dominar genéricos significa não apenas entender sua sintaxe, mas também internalizar os trade-offs e os padrões idiomáticos que governam seu uso eficaz.

## Resumo das Melhores Práticas

1. **Use Genéricos para Coleções e Algoritmos Homogêneos:** Este é o caso de uso principal. Sempre que você estiver implementando uma estrutura de dados (lista, árvore, etc.) ou um algoritmo (map, filter, sort) que opera em uma coleção de elementos do *mesmo* tipo, genéricos são a escolha ideal.
2. **Use Interfaces para Abstrair Comportamento em Tipos Heterogêneos:** Quando o objetivo é trabalhar com um conjunto de tipos *diferentes* que compartilham um conjunto comum de métodos (comportamento), as interfaces continuam sendo a ferramenta superior.
3. **Evite o Uso Excessivo de Genéricos:** Nem toda função precisa ser genérica. Se uma função opera em um único tipo concreto e não há uma necessidade clara de reutilização em outros tipos, uma função simples e não genérica é muitas vezes mais

clara e mais fácil de manter.[7]

4. **Esteja Ciente das Implicações de Desempenho:** Evite passar interfaces como argumentos de tipo para funções genéricas que chamam métodos dessa interface, pois isso pode levar a uma degradação do desempenho. Lembre-se da regra prática: "Interfaces para o que algo *faz*, genéricos para com o que algo *é feito*".[26]
5. **Aproveite a Inferência de Tipo, mas Seja Explícito para Clareza:** Confie na inferência de tipo do compilador para manter o código limpo, mas não hesite em adicionar argumentos de tipo explícitos se isso tornar uma chamada de função complexa mais fácil de ler e entender.

## Recursos Selecionados para Estudo Adicional

Para aprofundar seu conhecimento, os seguintes recursos oficiais e da comunidade são altamente recomendados:

- **Documentação e Tutoriais Oficiais:**
  - (https://go.dev/doc/tutorial/generics) [14]: O ponto de partida oficial e essencial.
  - (https://go.dev/blog/intro-generics) [12]: Uma visão geral conceitual dos novos recursos.
  - (https://go.dev/blog/when-generics) [18]: Um guia prático sobre os casos de uso idiomáticos.
  - (https://go.dev/blog/generic-interfaces) [32]: Uma exploração de tópicos avançados.
- **Especificação e Propostas de Design:**
  - (https://go.dev/ref/spec#Type_parameter_declarations): A referência definitiva para a sintaxe e semântica.
  - (https://go.googlesource.com/proposal/+/master/design/43651-type-parameters.md) [37]: O documento de design original, fornecendo um contexto histórico e profundo sobre as decisões de design.
- **Bibliotecas e Ferramentas da Comunidade:**
  - samber/lo [31]: Uma biblioteca popular e abrangente de utilitários funcionais genéricos, excelente para ver os padrões em ação.

### Referências citadas


1. Writing generic collection types in Go: the missing documentation | DoltHub Blog, acessado em setembro 12, 2025, https://www.dolthub.com/blog/2024-07-01-golang-generic-collections/
2. Go 1.18 is released! - The Go Programming Language, acessado em setembro 12, 2025, https://go.dev/blog/go1.18
3. Generics in Go - Bitfield Consulting, acessado em setembro 12, 2025, https://bitfieldconsulting.com/posts/generics
4. akutz/go-generics-the-hard-way - GitHub, acessado em setembro 12, 2025, https://github.com/akutz/go-generics-the-hard-way

5. Go Generics: A Deep Dive | Leapcell, acessado em setembro 12, 2025, https://leapcell.io/blog/go-generics-deep-dive
6. How is the empty interface different than a generic? - Stack Overflow, acessado em setembro 12, 2025, https://stackoverflow.com/questions/45171642/how-is-the-empty-interface-different-than-a-generic
7. Generics in Go: Your Friendly Guide to Reusable Code - DEV Community, acessado em setembro 12, 2025, https://dev.to/shrsv/generics-in-go-your-friendly-guide-to-reusable-code-4fkc
8. Mastering Generics in Go: A Comprehensive Guide | by Hiten Pratap Singh - Medium, acessado em setembro 12, 2025, https://medium.com/hprog99/mastering-generics-in-go-a-comprehensive-guide-4d05ec4b12b
9. The generics implementation of Go 1.18 - DeepSource, acessado em setembro 12, 2025, https://deepsource.com/blog/go-1-18-generics-implementation
10. The generics implementation of Go 1.18 • DeepSource, acessado em setembro 12, 2025, https://deepsource.com/blog/go-1-18-generics-implementation/
11. Mastering Generics In Go: A Comprehensive Tutorial - Kelche, acessado em setembro 12, 2025, https://www.kelche.co/blog/go/golang-generics/
12. An Introduction To Generics - The Go Programming Language, acessado em setembro 12, 2025, https://go.dev/blog/intro-generics
13. Intro to Generics in Go - DEV Community, acessado em setembro 12, 2025, https://dev.to/jpoly1219/intro-to-generics-in-go-36am
14. Tutorial: Getting started with generics - The Go Programming Language, acessado em setembro 12, 2025, https://go.dev/doc/tutorial/generics
15. Generics - Go by Example, acessado em setembro 12, 2025, https://gobyexample.com/generics
16. Everything You Always Wanted to Know About Type Inference - And ..., acessado em setembro 12, 2025, https://go.dev/blog/type-inference
17. Generics - Practical Go Lessons, acessado em setembro 12, 2025, https://www.practical-go-lessons.com/chap-38-generics
18. When To Use Generics - The Go Programming Language, acessado em setembro 12, 2025, https://go.dev/blog/when-generics
19. Advanced Generics Use Cases in Go 1.18+ | by Sanyamdubey | May, 2025 | Medium, acessado em setembro 12, 2025, https://medium.com/@sanyamdubey28/advanced-generics-use-cases-in-go-1-18-6e133ae08517
20. Go Generics - MicroFIRE, acessado em setembro 12, 2025, https://ganhua.wang/introduction-to-go-generics
21. GoLang Generics: Practical Examples to Level Up Your Code - DEV Community, acessado em setembro 12, 2025, https://dev.to/shrsv/golang-generics-practical-examples-to-level-up-your-code-4ell
22. What's New in Go 1.18. Generics, Test Fuzzing, Optimization... | by Ricardo Gerardi | The Pragmatic Programmers | Medium, acessado em setembro 12, 2025,

https://medium.com/pragmatic-programmers/whats-new-in-go-1-18-e7773a110202
23. Interface in Generics vs. Interface as Argument Type : r/golang - Reddit, acessado em setembro 12, 2025, https://www.reddit.com/r/golang/comments/1k3iq50/interface_in_generics_vs_interface_as_argument/
24. Generics can make your Go code slower - PlanetScale, acessado em setembro 12, 2025, https://planetscale.com/blog/generics-can-make-your-go-code-slower
25. Benchmarking Generics in Go - ProgrammingPercy, acessado em setembro 12, 2025, https://programmingpercy.tech/blog/benchmarking-generics-in-go/
26. What is the current state of Go's generics? : r/golang - Reddit, acessado em setembro 12, 2025, https://www.reddit.com/r/golang/comments/1bqruza/what_is_the_current_state_of_gos_generics/
27. Generics in Go — From Basics to Advanced for Senior Developers | by Sogol Hedayatmanesh | Aug, 2025 | Medium, acessado em setembro 12, 2025, https://medium.com/@sogol.hedayatmanesh/generics-in-go-from-basics-to-advanced-for-senior-developers-887790b018d0
28. Notes: Golang generics with map, filter, and reduce implementation, acessado em setembro 12, 2025, https://btree.dev/notes-golang-generics-with-map-filter-and-reduce-implementation
29. Implementing Map, Filter, And Reduce Using Generic In Go - Vincent Taneri - Vitaneri, acessado em setembro 12, 2025, https://vitaneri.com/posts/implementing-map-filter-and-reduce-using-generic-in-go
30. I just published : Iterator lib for Go: Library providing Map(), Filter(), Reduce() for Go, acessado em setembro 12, 2025, https://groups.google.com/g/golang-nuts/c/9UaV33bB4Qc
31. Idiomatic Replacement for map/reduce/filter/etc - go - Stack Overflow, acessado em setembro 12, 2025, https://stackoverflow.com/questions/49468242/idiomatic-replacement-for-map-reduce-filter-etc
32. Generic interfaces - The Go Programming Language, acessado em setembro 12, 2025, https://go.dev/blog/generic-interfaces
33. How to use generics for creating self-referring interfaces - Applied Go, acessado em setembro 12, 2025, https://appliedgo.com/blog/generic-interface-functions
34. Polymorphic, Recursive Interfaces Using Go Generics | Stitch Fix Technology, acessado em setembro 12, 2025, https://multithreaded.stitchfix.com/blog/2023/02/01/go-polymorphic-interfaces/
35. Goodbye core types - Hello Go as we know and love it! - The Go Programming Language, acessado em setembro 12, 2025, https://go.dev/blog/coretypes
36. Documentation - The Go Programming Language, acessado em setembro 12, 2025, https://go.dev/doc/
37. Go 1.18 Release Notes - The Go Programming Language, acessado em setembro

12, 2025, https://tip.golang.org/doc/go1.18