GoLang Generics: Practical Examples to Level Up Your Code

Язык оригинала: en

# Оригинал

Hello, I'm Shrijith Venkatramana. I’m building  
LiveReview  
, a private AI code review tool that runs on your LLM key (OpenAI, Gemini, etc.) with flat, no-seat pricing -- built for small teams. Do check it out and give it a try!  
GoLang generics, introduced in  
Go 1.18  
, brought a new level of flexibility to a language known for its simplicity and performance. They let you write reusable, type-safe code without sacrificing Go’s clean design. This article dives into practical examples of generics in Go, showing you how to use them effectively. We’ll cover the basics, explore real-world use cases, and provide complete, runnable code snippets. Let’s get started.  
What Are Generics and Why Should You Care?  
Generics allow you to write functions and types that work with multiple data types while keeping type safety. Before generics, you’d often use  
interface{}  
for flexibility, but that came with type assertions and runtime errors.  
Generics solve this by letting you define type parameters  
, checked at compile time.  
For example, instead of writing separate functions for summing integers and floats, generics let you write one function that handles both. This reduces code duplication and makes maintenance easier. The Go team added generics to address these pain points while keeping the language’s simplicity.  
Key benefit  
: Write less code, keep it type-safe, and improve readability.  
Official Go Generics Documentation  
Defining Generic Functions: The Basics  
A generic function uses  
type parameters  
, declared in square brackets before the function’s parameters. Let’s start with a simple example: a function to find the maximum of two values, regardless of their type.  
package  
main  
import  
(  
"fmt"  
)  
// Define a constraint for comparable types (like int, float64, string)  
type  
Comparable  
interface  
{  
~  
int  
|  
~  
float64  
|  
~  
string  
}  
// Max finds the larger of two values  
func  
Max  
[  
T  
Comparable  
](  
a  
,  
b  
T  
)  
T  
{  
if  
a  
>  
b  
{  
return  
a  
}  
return  
b  
}  
func  
main  
()  
{  
// Test with integers  
fmt  
.  
Println  
(  
Max  
(  
5  
,  
10  
))  
// Output: 10  
// Test with floats  
fmt  
.  
Println  
(  
Max  
(  
3.14  
,  
2.71  
))  
// Output: 3.14  
// Test with strings  
fmt  
.  
Println  
(  
Max  
(  
"apple"  
,  
"banana"  
))  
// Output: banana  
}  
Enter fullscreen mode  
Exit fullscreen mode  
How it works  
: The  
Comparable  
constraint ensures  
T  
supports the  
>  
operator. The  
~  
means it includes types derived from  
int  
,  
float64  
, or  
string  
. This function works for any type that fits the constraint, and the compiler catches type mismatches.  
Try it  
: Copy and run this code. It’s simple but shows the power of generics in reducing repetitive code.  
Generic Types: Building Reusable Data Structures  
Generics aren’t just for functions—you can define generic structs, too. Let’s create a generic  
Stack  
type that works with any data type.  
package  
main  
import  
(  
"fmt"  
)  
// Stack is a generic type that holds items of type T  
type  
Stack  
[  
T  
any  
]  
struct  
{  
items  
[]  
T  
}  
// Push adds an item to the stack  
func  
(  
s  
\*  
Stack  
[  
T  
])  
Push  
(  
item  
T  
)  
{  
s  
.  
items  
=  
append  
(  
s  
.  
items  
,  
item  
)  
}  
// Pop removes and returns the top item  
func  
(  
s  
\*  
Stack  
[  
T  
])  
Pop  
()  
(  
T  
,  
bool  
)  
{  
if  
len  
(  
s  
.  
items  
)  
==  
0  
{  
var  
zero  
T  
return  
zero  
,  
false  
}  
item  
:=  
s  
.  
items  
[  
len  
(  
s  
.  
items  
)  
-  
1  
]  
s  
.  
items  
=  
s  
.  
items  
[  
:  
len  
(  
s  
.  
items  
)  
-  
1  
]  
return  
item  
,  
true  
}  
func  
main  
()  
{  
// Stack of integers  
intStack  
:=  
Stack  
[  
int  
]{}  
intStack  
.  
Push  
(  
1  
)  
intStack  
.  
Push  
(  
2  
)  
fmt  
.  
Println  
(  
intStack  
.  
Pop  
())  
// Output: 2, true  
fmt  
.  
Println  
(  
intStack  
.  
Pop  
())  
// Output: 1, true  
fmt  
.  
Println  
(  
intStack  
.  
Pop  
())  
// Output: 0, false  
// Stack of strings  
stringStack  
:=  
Stack  
[  
string  
]{}  
stringStack  
.  
Push  
(  
"hello"  
)  
stringStack  
.  
Push  
(  
"world"  
)  
fmt  
.  
Println  
(  
stringStack  
.  
Pop  
())  
// Output: world, true  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Why this rocks  
: One  
Stack  
implementation handles any type, from  
int  
to custom structs. The  
any  
constraint means no restrictions on  
T  
. This eliminates the need for type assertions, unlike with  
interface{}  
.  
Go Blog on Generic Types  
Constraints: Controlling Which Types Are Allowed  
Constraints define what types a generic function or type can accept. Go provides built-in constraints like  
comparable  
and  
any  
, but you can create custom ones. Let’s look at a generic function that sums numbers, using a custom constraint.  
package  
main  
import  
(  
"fmt"  
)  
// Number is a constraint for numeric types  
type  
Number  
interface  
{  
~  
int  
|  
~  
float64  
|  
~  
float32  
}  
// Sum calculates the sum of a slice of numbers  
func  
Sum  
[  
T  
Number  
](  
nums  
[]  
T  
)  
T  
{  
var  
sum  
T  
for  
\_  
,  
num  
:=  
range  
nums  
{  
sum  
+=  
num  
}  
return  
sum  
}  
func  
main  
()  
{  
ints  
:=  
[]  
int  
{  
1  
,  
2  
,  
3  
,  
4  
}  
floats  
:=  
[]  
float64  
{  
1.5  
,  
2.5  
,  
3.5  
}  
fmt  
.  
Println  
(  
Sum  
(  
ints  
))  
// Output: 10  
fmt  
.  
Println  
(  
Sum  
(  
floats  
))  
// Output: 7.5  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Key point  
: The  
Number  
constraint ensures  
T  
supports the  
+  
operator. The  
~  
allows derived types (e.g.,  
type MyInt int  
). This makes the function flexible yet safe.  
Pro tip  
: Use constraints to make your generics explicit about supported operations.  
Combining Generics with Interfaces  
Interfaces and generics can work together. Let’s create a generic function that processes any type implementing a specific interface. Here’s an example with a  
Stringer  
interface.  
package  
main  
import  
(  
"fmt"  
)  
// Stringer is an interface for types that have a String() method  
type  
Stringer  
interface  
{  
String  
()  
string  
}  
// PrintAll prints the String() output for a slice of Stringer types  
func  
PrintAll  
[  
T  
Stringer  
](  
items  
[]  
T  
)  
{  
for  
\_  
,  
item  
:=  
range  
items  
{  
fmt  
.  
Println  
(  
item  
.  
String  
())  
}  
}  
// Person is a custom type that implements Stringer  
type  
Person  
struct  
{  
Name  
string  
}  
func  
(  
p  
Person  
)  
String  
()  
string  
{  
return  
fmt  
.  
Sprintf  
(  
"Person: %s"  
,  
p  
.  
Name  
)  
}  
func  
main  
()  
{  
people  
:=  
[]  
Person  
{  
{  
Name  
:  
"Alice"  
},  
{  
Name  
:  
"Bob"  
},  
}  
PrintAll  
(  
people  
)  
// Output: Person: Alice  
// Person: Bob  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Why this is useful  
: The  
PrintAll  
function works with any type that implements  
Stringer  
, making it reusable across structs like  
Person  
,  
User  
, or even standard library types like  
time.Time  
.  
Go Interface Documentation  
Generic Maps: Flexible Key-Value Stores  
Let’s build a generic  
SafeMap  
that supports any key and value types, with thread-safe operations using a mutex.  
package  
main  
import  
(  
"fmt"  
"sync"  
)  
// SafeMap is a generic thread-safe map  
type  
SafeMap  
[  
K  
comparable  
,  
V  
any  
]  
struct  
{  
items  
map  
[  
K  
]  
V  
mutex  
sync  
.  
RWMutex  
}  
// NewSafeMap creates a new SafeMap  
func  
NewSafeMap  
[  
K  
comparable  
,  
V  
any  
]()  
\*  
SafeMap  
[  
K  
,  
V  
]  
{  
return  
&  
SafeMap  
[  
K  
,  
V  
]{  
items  
:  
make  
(  
map  
[  
K  
]  
V  
),  
}  
}  
// Set stores a key-value pair  
func  
(  
m  
\*  
SafeMap  
[  
K  
,  
V  
])  
Set  
(  
key  
K  
,  
value  
V  
)  
{  
m  
.  
mutex  
.  
Lock  
()  
defer  
m  
.  
mutex  
.  
Unlock  
()  
m  
.  
items  
[  
key  
]  
=  
value  
}  
// Get retrieves a value by key  
func  
(  
m  
\*  
SafeMap  
[  
K  
,  
V  
])  
Get  
(  
key  
K  
)  
(  
V  
,  
bool  
)  
{  
m  
.  
mutex  
.  
RLock  
()  
defer  
m  
.  
mutex  
.  
RUnlock  
()  
value  
,  
exists  
:=  
m  
.  
items  
[  
key  
]  
return  
value  
,  
exists  
}  
func  
main  
()  
{  
// Map with string keys and int values  
m  
:=  
NewSafeMap  
[  
string  
,  
int  
]()  
m  
.  
Set  
(  
"age"  
,  
30  
)  
m  
.  
Set  
(  
"score"  
,  
95  
)  
fmt  
.  
Println  
(  
m  
.  
Get  
(  
"age"  
))  
// Output: 30, true  
fmt  
.  
Println  
(  
m  
.  
Get  
(  
"score"  
))  
// Output: 95, true  
fmt  
.  
Println  
(  
m  
.  
Get  
(  
"name"  
))  
// Output: 0, false  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Why it’s great  
: The  
SafeMap  
works with any comparable key type and any value type. The  
comparable  
constraint ensures keys can be used in a map. The mutex ensures thread safety.  
Use case  
: Ideal for concurrent applications needing flexible key-value storage.  
Performance Considerations with Generics  
Generics are compiled into specialized code for each type used, which can increase binary size but maintains Go’s performance. Here’s a quick comparison of generics vs.  
interface{}  
for a sum function.  
Approach  
Pros  
Cons  
Generics  
Type-safe, no runtime overhead  
Slightly larger binary size  
Interface{}  
Flexible, no code duplication  
Runtime type assertions, errors  
Key takeaway  
: Generics offer better performance than  
interface{}  
because they eliminate runtime type checks. However, avoid overusing generics for simple cases where a single-type function is enough.  
Benchmark example  
:  
package  
main  
import  
(  
"fmt"  
"time"  
)  
type  
Number  
interface  
{  
~  
int  
|  
~  
float64  
}  
func  
SumGeneric  
[  
T  
Number  
](  
nums  
[]  
T  
)  
T  
{  
var  
sum  
T  
for  
\_  
,  
num  
:=  
range  
nums  
{  
sum  
+=  
num  
}  
return  
sum  
}  
func  
SumInterface  
(  
nums  
[]  
interface  
{})  
interface  
{}  
{  
var  
sum  
float64  
for  
\_  
,  
num  
:=  
range  
nums  
{  
sum  
+=  
num  
.  
(  
float64  
)  
}  
return  
sum  
}  
func  
main  
()  
{  
ints  
:=  
[]  
int  
{  
1  
,  
2  
,  
3  
,  
4  
,  
5  
}  
interfaceSlice  
:=  
make  
([]  
interface  
{},  
len  
(  
ints  
))  
for  
i  
,  
v  
:=  
range  
ints  
{  
interfaceSlice  
[  
i  
]  
=  
float64  
(  
v  
)  
}  
start  
:=  
time  
.  
Now  
()  
fmt  
.  
Println  
(  
SumGeneric  
(  
ints  
))  
// Output: 15  
fmt  
.  
Println  
(  
time  
.  
Since  
(  
start  
))  
// Output: ~100ns (varies)  
start  
=  
time  
.  
Now  
()  
fmt  
.  
Println  
(  
SumInterface  
(  
interfaceSlice  
))  
// Output: 15  
fmt  
.  
Println  
(  
time  
.  
Since  
(  
start  
))  
// Output: ~200ns (varies)  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Observation  
: The generic version is faster due to no type assertions.  
Common Pitfalls and How to Avoid Them  
Generics are powerful, but they come with traps. Here are common issues and fixes:  
Pitfall  
Solution  
Overusing generics  
Use generics only when type flexibility is needed.  
Incorrect constraints  
Use specific constraints like  
comparable  
or custom interfaces.  
Ignoring derived types  
Use  
~  
in constraints for flexibility.  
Example fix  
for incorrect constraints:  
package  
main  
import  
(  
"fmt"  
)  
// Number constraint for numeric types  
type  
Number  
interface  
{  
~  
int  
|  
~  
float64  
}  
// Multiply multiplies two numbers  
func  
Multiply  
[  
T  
Number  
](  
a  
,  
b  
T  
)  
T  
{  
return  
a  
\*  
b  
}  
func  
main  
()  
{  
fmt  
.  
Println  
(  
Multiply  
(  
5  
,  
3  
))  
// Output: 15  
fmt  
.  
Println  
(  
Multiply  
(  
2.5  
,  
4.0  
))  
// Output: 10  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Tip  
: Always test your generic code with multiple types to ensure constraints are correct.  
Where to Go Next with Generics  
Generics open up new possibilities in Go, but they’re just one tool in your toolbox. To deepen your understanding:  
Experiment  
: Try rewriting an existing project using generics to reduce code duplication.  
Read the source  
: Check out how libraries like  
golang.org/x/exp/slices  
use generics.  
Contribute  
: Explore open-source projects adopting generics and contribute generic utilities.  
Stay updated  
: Follow Go’s release notes for new generic features or constraints.  
For practical next steps, try building a generic priority queue or a type-safe database client. These projects will solidify your understanding and show you where generics shine. If you hit roadblocks, the Go community on forums like  
Reddit  
is a great place to ask questions.

# Перевод на русский

Hello, I'm Shrijith Venkatramana. I’m building  
LiveReview  
, a private AI code review tool that runs on your LLM key (OpenAI, Gemini, etc.) with flat, no-seat pricing -- built for small teams. Do check it out and give it a try!  
GoLang generics, introduced in  
Go 1.18  
, brought a new level of flexibility to a language known for its simplicity and performance. They let you write reusable, type-safe code without sacrificing Go’s clean design. This article dives into practical examples of generics in Go, showing you how to use them effectively. We’ll cover the basics, explore real-world use cases, and provide complete, runnable code snippets. Let’s get started.  
What Are Generics and Why Should You Care?  
Generics allow you to write functions and types that work with multiple data types while keeping type safety. Before generics, you’d often use  
interface{}  
for flexibility, but that came with type assertions and runtime errors.  
Generics solve this by letting you define type parameters  
, checked at compile time.  
For example, instead of writing separate functions for summing integers and floats, generics let you write one function that handles both. This reduces code duplication and makes maintenance easier. The Go team added generics to address these pain points while keeping the language’s simplicity.  
Key benefit  
: Write less code, keep it type-safe, and improve readability.  
Official Go Generics Documentation  
Defining Generic Functions: The Basics  
A generic function uses  
type parameters  
, declared in square brackets before the function’s parameters. Let’s start with a simple example: a function to find the maximum of two values, regardless of their type.  
package  
main  
import  
(  
"fmt"  
)  
// Define a constraint for comparable types (like int, float64, string)  
type  
Comparable  
interface  
{  
~  
int  
|  
~  
float64  
|  
~  
string  
}  
// Max finds the larger of two values  
func  
Max  
[  
T  
Comparable  
](  
a  
,  
b  
T  
)  
T  
{  
if  
a  
>  
b  
{  
return  
a  
}  
return  
b  
}  
func  
main  
()  
{  
// Test with integers  
fmt  
.  
Println  
(  
Max  
(  
5  
,  
10  
))  
// Output: 10  
// Test with floats  
fmt  
.  
Println  
(  
Max  
(  
3.14  
,  
2.71  
))  
// Output: 3.14  
// Test with strings  
fmt  
.  
Println  
(  
Max  
(  
"apple"  
,  
"banana"  
))  
// Output: banana  
}  
Enter fullscreen mode  
Exit fullscreen mode  
How it works  
: The  
Comparable  
constraint ensures  
T  
supports the  
>  
operator. The  
~  
means it includes types derived from  
int  
,  
float64  
, or  
string  
. This function works for any type that fits the constraint, and the compiler catches type mismatches.  
Try it  
: Copy and run this code. It’s simple but shows the power of generics in reducing repetitive code.  
Generic Types: Building Reusable Data Structures  
Generics aren’t just for functions—you can define generic structs, too. Let’s create a generic  
Stack  
type that works with any data type.  
package  
main  
import  
(  
"fmt"  
)  
// Stack is a generic type that holds items of type T  
type  
Stack  
[  
T  
any  
]  
struct  
{  
items  
[]  
T  
}  
// Push adds an item to the stack  
func  
(  
s  
\*  
Stack  
[  
T  
])  
Push  
(  
item  
T  
)  
{  
s  
.  
items  
=  
append  
(  
s  
.  
items  
,  
item  
)  
}  
// Pop removes and returns the top item  
func  
(  
s  
\*  
Stack  
[  
T  
])  
Pop  
()  
(  
T  
,  
bool  
)  
{  
if  
len  
(  
s  
.  
items  
)  
==  
0  
{  
var  
zero  
T  
return  
zero  
,  
false  
}  
item  
:=  
s  
.  
items  
[  
len  
(  
s  
.  
items  
)  
-  
1  
]  
s  
.  
items  
=  
s  
.  
items  
[  
:  
len  
(  
s  
.  
items  
)  
-  
1  
]  
return  
item  
,  
true  
}  
func  
main  
()  
{  
// Stack of integers  
intStack  
:=  
Stack  
[  
int  
]{}  
intStack  
.  
Push  
(  
1  
)  
intStack  
.  
Push  
(  
2  
)  
fmt  
.  
Println  
(  
intStack  
.  
Pop  
())  
// Output: 2, true  
fmt  
.  
Println  
(  
intStack  
.  
Pop  
())  
// Output: 1, true  
fmt  
.  
Println  
(  
intStack  
.  
Pop  
())  
// Output: 0, false  
// Stack of strings  
stringStack  
:=  
Stack  
[  
string  
]{}  
stringStack  
.  
Push  
(  
"hello"  
)  
stringStack  
.  
Push  
(  
"world"  
)  
fmt  
.  
Println  
(  
stringStack  
.  
Pop  
())  
// Output: world, true  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Why this rocks  
: One  
Stack  
implementation handles any type, from  
int  
to custom structs. The  
any  
constraint means no restrictions on  
T  
. This eliminates the need for type assertions, unlike with  
interface{}  
.  
Go Blog on Generic Types  
Constraints: Controlling Which Types Are Allowed  
Constraints define what types a generic function or type can accept. Go provides built-in constraints like  
comparable  
and  
any  
, but you can create custom ones. Let’s look at a generic function that sums numbers, using a custom constraint.  
package  
main  
import  
(  
"fmt"  
)  
// Number is a constraint for numeric types  
type  
Number  
interface  
{  
~  
int  
|  
~  
float64  
|  
~  
float32  
}  
// Sum calculates the sum of a slice of numbers  
func  
Sum  
[  
T  
Number  
](  
nums  
[]  
T  
)  
T  
{  
var  
sum  
T  
for  
\_  
,  
num  
:=  
range  
nums  
{  
sum  
+=  
num  
}  
return  
sum  
}  
func  
main  
()  
{  
ints  
:=  
[]  
int  
{  
1  
,  
2  
,  
3  
,  
4  
}  
floats  
:=  
[]  
float64  
{  
1.5  
,  
2.5  
,  
3.5  
}  
fmt  
.  
Println  
(  
Sum  
(  
ints  
))  
// Output: 10  
fmt  
.  
Println  
(  
Sum  
(  
floats  
))  
// Output: 7.5  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Key point  
: The  
Number  
constraint ensures  
T  
supports the  
+  
operator. The  
~  
allows derived types (e.g.,  
type MyInt int  
). This makes the function flexible yet safe.  
Pro tip  
: Use constraints to make your generics explicit about supported operations.  
Combining Generics with Interfaces  
Interfaces and generics can work together. Let’s create a generic function that processes any type implementing a specific interface. Here’s an example with a  
Stringer  
interface.  
package  
main  
import  
(  
"fmt"  
)  
// Stringer is an interface for types that have a String() method  
type  
Stringer  
interface  
{  
String  
()  
string  
}  
// PrintAll prints the String() output for a slice of Stringer types  
func  
PrintAll  
[  
T  
Stringer  
](  
items  
[]  
T  
)  
{  
for  
\_  
,  
item  
:=  
range  
items  
{  
fmt  
.  
Println  
(  
item  
.  
String  
())  
}  
}  
// Person is a custom type that implements Stringer  
type  
Person  
struct  
{  
Name  
string  
}  
func  
(  
p  
Person  
)  
String  
()  
string  
{  
return  
fmt  
.  
Sprintf  
(  
"Person: %s"  
,  
p  
.  
Name  
)  
}  
func  
main  
()  
{  
people  
:=  
[]  
Person  
{  
{  
Name  
:  
"Alice"  
},  
{  
Name  
:  
"Bob"  
},  
}  
PrintAll  
(  
people  
)  
// Output: Person: Alice  
// Person: Bob  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Why this is useful  
: The  
PrintAll  
function works with any type that implements  
Stringer  
, making it reusable across structs like  
Person  
,  
User  
, or even standard library types like  
time.Time  
.  
Go Interface Documentation  
Generic Maps: Flexible Key-Value Stores  
Let’s build a generic  
SafeMap  
that supports any key and value types, with thread-safe operations using a mutex.  
package  
main  
import  
(  
"fmt"  
"sync"  
)  
// SafeMap is a generic thread-safe map  
type  
SafeMap  
[  
K  
comparable  
,  
V  
any  
]  
struct  
{  
items  
map  
[  
K  
]  
V  
mutex  
sync  
.  
RWMutex  
}  
// NewSafeMap creates a new SafeMap  
func  
NewSafeMap  
[  
K  
comparable  
,  
V  
any  
]()  
\*  
SafeMap  
[  
K  
,  
V  
]  
{  
return  
&  
SafeMap  
[  
K  
,  
V  
]{  
items  
:  
make  
(  
map  
[  
K  
]  
V  
),  
}  
}  
// Set stores a key-value pair  
func  
(  
m  
\*  
SafeMap  
[  
K  
,  
V  
])  
Set  
(  
key  
K  
,  
value  
V  
)  
{  
m  
.  
mutex  
.  
Lock  
()  
defer  
m  
.  
mutex  
.  
Unlock  
()  
m  
.  
items  
[  
key  
]  
=  
value  
}  
// Get retrieves a value by key  
func  
(  
m  
\*  
SafeMap  
[  
K  
,  
V  
])  
Get  
(  
key  
K  
)  
(  
V  
,  
bool  
)  
{  
m  
.  
mutex  
.  
RLock  
()  
defer  
m  
.  
mutex  
.  
RUnlock  
()  
value  
,  
exists  
:=  
m  
.  
items  
[  
key  
]  
return  
value  
,  
exists  
}  
func  
main  
()  
{  
// Map with string keys and int values  
m  
:=  
NewSafeMap  
[  
string  
,  
int  
]()  
m  
.  
Set  
(  
"age"  
,  
30  
)  
m  
.  
Set  
(  
"score"  
,  
95  
)  
fmt  
.  
Println  
(  
m  
.  
Get  
(  
"age"  
))  
// Output: 30, true  
fmt  
.  
Println  
(  
m  
.  
Get  
(  
"score"  
))  
// Output: 95, true  
fmt  
.  
Println  
(  
m  
.  
Get  
(  
"name"  
))  
// Output: 0, false  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Why it’s great  
: The  
SafeMap  
works with any comparable key type and any value type. The  
comparable  
constraint ensures keys can be used in a map. The mutex ensures thread safety.  
Use case  
: Ideal for concurrent applications needing flexible key-value storage.  
Performance Considerations with Generics  
Generics are compiled into specialized code for each type used, which can increase binary size but maintains Go’s performance. Here’s a quick comparison of generics vs.  
interface{}  
for a sum function.  
Approach  
Pros  
Cons  
Generics  
Type-safe, no runtime overhead  
Slightly larger binary size  
Interface{}  
Flexible, no code duplication  
Runtime type assertions, errors  
Key takeaway  
: Generics offer better performance than  
interface{}  
because they eliminate runtime type checks. However, avoid overusing generics for simple cases where a single-type function is enough.  
Benchmark example  
:  
package  
main  
import  
(  
"fmt"  
"time"  
)  
type  
Number  
interface  
{  
~  
int  
|  
~  
float64  
}  
func  
SumGeneric  
[  
T  
Number  
](  
nums  
[]  
T  
)  
T  
{  
var  
sum  
T  
for  
\_  
,  
num  
:=  
range  
nums  
{  
sum  
+=  
num  
}  
return  
sum  
}  
func  
SumInterface  
(  
nums  
[]  
interface  
{})  
interface  
{}  
{  
var  
sum  
float64  
for  
\_  
,  
num  
:=  
range  
nums  
{  
sum  
+=  
num  
.  
(  
float64  
)  
}  
return  
sum  
}  
func  
main  
()  
{  
ints  
:=  
[]  
int  
{  
1  
,  
2  
,  
3  
,  
4  
,  
5  
}  
interfaceSlice  
:=  
make  
([]  
interface  
{},  
len  
(  
ints  
))  
for  
i  
,  
v  
:=  
range  
ints  
{  
interfaceSlice  
[  
i  
]  
=  
float64  
(  
v  
)  
}  
start  
:=  
time  
.  
Now  
()  
fmt  
.  
Println  
(  
SumGeneric  
(  
ints  
))  
// Output: 15  
fmt  
.  
Println  
(  
time  
.  
Since  
(  
start  
))  
// Output: ~100ns (varies)  
start  
=  
time  
.  
Now  
()  
fmt  
.  
Println  
(  
SumInterface  
(  
interfaceSlice  
))  
// Output: 15  
fmt  
.  
Println  
(  
time  
.  
Since  
(  
start  
))  
// Output: ~200ns (varies)  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Observation  
: The generic version is faster due to no type assertions.  
Common Pitfalls and How to Avoid Them  
Generics are powerful, but they come with traps. Here are common issues and fixes:  
Pitfall  
Solution  
Overusing generics  
Use generics only when type flexibility is needed.  
Incorrect constraints  
Use specific constraints like  
comparable  
or custom interfaces.  
Ignoring derived types  
Use  
~  
in constraints for flexibility.  
Example fix  
for incorrect constraints:  
package  
main  
import  
(  
"fmt"  
)  
// Number constraint for numeric types  
type  
Number  
interface  
{  
~  
int  
|  
~  
float64  
}  
// Multiply multiplies two numbers  
func  
Multiply  
[  
T  
Number  
](  
a  
,  
b  
T  
)  
T  
{  
return  
a  
\*  
b  
}  
func  
main  
()  
{  
fmt  
.  
Println  
(  
Multiply  
(  
5  
,  
3  
))  
// Output: 15  
fmt  
.  
Println  
(  
Multiply  
(  
2.5  
,  
4.0  
))  
// Output: 10  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Tip  
: Always test your generic code with multiple types to ensure constraints are correct.  
Where to Go Next with Generics  
Generics open up new possibilities in Go, but they’re just one tool in your toolbox. To deepen your understanding:  
Experiment  
: Try rewriting an existing project using generics to reduce code duplication.  
Read the source  
: Check out how libraries like  
golang.org/x/exp/slices  
use generics.  
Contribute  
: Explore open-source projects adopting generics and contribute generic utilities.  
Stay updated  
: Follow Go’s release notes for new generic features or constraints.  
For practical next steps, try building a generic priority queue or a type-safe database client. These projects will solidify your understanding and show you where generics shine. If you hit roadblocks, the Go community on forums like  
Reddit  
is a great place to ask questions.