GoLang 101: Understanding Polymorphism Through Interfaces

Язык оригинала: en

# Оригинал

go (8 Part Series)  
1  
GoLang 101: Getting Started with Go  
2  
GoLang 101: Mastering Variables, Types, and Memory in Go  
...  
4 more parts...  
3  
GoLang 101: Making Decisions and Loops with Control Flow  
4  
GoLang 101: Organizing Data in Go with Arrays, Slices, Maps, and Structs  
5  
GoLang 101: Communicating with the World — Files, JSON, and Protocols  
6  
GoLang101: Mastering Functions  
7  
GoLang101: Go’s Minimal Object-Oriented Style  
8  
GoLang 101: Understanding Polymorphism Through Interfaces  
Hey everyone! 👋 Welcome back to our GoLang 101 series.  
In this article, we’re diving into a concept called polymorphism.  
If you’ve worked with languages like PHP or Python, you’re likely familiar with polymorphism being closely tied to inheritance. But Go takes a simpler and more elegant path — no inheritance required. Instead, Go uses interfaces to unlock powerful polymorphic behavior.  
Let’s see how this works.  
What is Polymorphism?  
At its core, polymorphism is the ability for an object to take on different forms depending on the context. A simple way to think about this is a function or method with a single name that performs different actions based on the type of object it's acting on.  
Take the concept of  
area  
, for example. If you want to compute the area of a rectangle, the calculation is  
base \* height  
. But for a triangle, it's  
(1/2) \* base \* height  
. The function  
area  
does two different things depending on whether it's dealing with a rectangle or a triangle.  
This is polymorphism in action, the  
area  
method is polymorphic because it has different forms depending on the object.  
In many object-oriented languages, polymorphism is supported through inheritance. This is where classes have a "superclass" (parent) and "subclass" (child) relationship, and the subclass inherits the methods and data of the superclass. The subclass can then redefine, or "override" a method it inherited from the superclass to provide its own specific implementation.  
But Go has no inheritance.  
So how does it achieve polymorphism? Let’s see.  
Go's Solution: The Power of Interfaces  
Go uses interfaces to define behavior in a clean and flexible way.  
An interface is a set of method signatures — it defines what a type must do, not how it does it.  
Example: A Shape Interface:  
type  
Shape2D  
interface  
{  
Area  
()  
float64  
Perimeter  
()  
float64  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Any type that implements both Area() and Perimeter() methods automatically satisfies the Shape2D interface — no implements keyword needed.  
Let's define Rectangle and Triangle:  
type  
Rectangle  
struct  
{  
Width  
,  
Height  
float64  
}  
func  
(  
r  
Rectangle  
)  
Area  
()  
float64  
{  
return  
r  
.  
Width  
\*  
r  
.  
Height  
}  
func  
(  
r  
Rectangle  
)  
Perimeter  
()  
float64  
{  
return  
2  
\*  
(  
r  
.  
Width  
+  
r  
.  
Height  
)  
}  
Enter fullscreen mode  
Exit fullscreen mode  
type  
Triangle  
struct  
{  
Base  
,  
Height  
,  
SideA  
,  
SideB  
float64  
}  
func  
(  
t  
Triangle  
)  
Area  
()  
float64  
{  
return  
0.5  
\*  
t  
.  
Base  
\*  
t  
.  
Height  
}  
func  
(  
t  
Triangle  
)  
Perimeter  
()  
float64  
{  
return  
t  
.  
Base  
+  
t  
.  
SideA  
+  
t  
.  
SideB  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Both of these types now satisfy the  
Shape2D  
interface, no explicit declaration needed.  
Now that we have polymorphism through interfaces, we can write flexible functions.  
func  
FitsInYard  
(  
s  
Shape2D  
)  
bool  
{  
return  
s  
.  
Area  
()  
<  
100  
&&  
s  
.  
Perimeter  
()  
<  
100  
}  
Enter fullscreen mode  
Exit fullscreen mode  
This one function works with any type that satisfies Shape2D, Rectangle, Triangle, or even custom shapes in the future.  
You get code reuse, flexibility, and clarity — all without inheritance.  
Disambiguation with Type Assertions  
While interfaces are great for hiding differences, sometimes you need to "peel it apart" and figure out the exact underlying concrete type. This is especially useful in a program like a graphics application where you might have an  
DrawShape  
function that needs to call specific drawing APIs for different shapes (e.g.,  
DrawRectangle  
,  
DrawTriangle  
).  
For this, Go provides type assertions. A type assertion provides access to an interface value's underlying concrete value.  
func  
DrawShape  
(  
s  
Shape2D  
)  
{  
// Check if the underlying type is a Rectangle  
if  
rect  
,  
ok  
:=  
s  
.  
(  
Rectangle  
);  
ok  
{  
DrawRect  
(  
rect  
)  
}  
else  
if  
tri  
,  
ok  
:=  
s  
.  
(  
Triangle  
);  
ok  
{  
DrawTriangle  
(  
tri  
)  
}  
}  
Enter fullscreen mode  
Exit fullscreen mode  
A more convenient way to handle this is with a type switch, which is a special form of the  
switch  
statement for type assertions.  
func  
DrawShape  
(  
s  
Shape2D  
)  
{  
switch  
sh  
:=  
s  
.  
(  
type  
)  
{  
case  
Rectangle  
:  
DrawRect  
(  
sh  
)  
case  
Triangle  
:  
DrawTriangle  
(  
sh  
)  
// ... other cases  
}  
}  
Enter fullscreen mode  
Exit fullscreen mode  
The variable  
sh  
is now of the correct concrete type in each case block.  
A Common Use: Error Handling  
Another great use of interfaces is in error handling. In Go, many functions return two values: a result and an  
error  
.  
This  
error  
is actually an interface. The  
error  
interface is very simple; it has a single method called Error that returns a string.  
type  
error  
interface  
{  
Error  
()  
string  
}  
Enter fullscreen mode  
Exit fullscreen mode  
This simple interface allows any type to represent an error, as long as it has an Error() method. When a function returns an error, you should always check if it is  
nil  
. If it's not  
nil  
, it means something went wrong, and you should handle the error.  
f  
,  
err  
:=  
os  
.  
Open  
(  
"file.txt"  
)  
if  
err  
!=  
nil  
{  
fmt  
.  
Println  
(  
err  
)  
// Calls err.Error()  
return  
}  
Enter fullscreen mode  
Exit fullscreen mode  
This is a standard and robust way to handle errors in Go.  
Go's approach to polymorphism with interfaces is a powerful and flexible alternative to the traditional inheritance model. By focusing on behavior (methods) rather than data, interfaces allow you to write clean, reusable code that can work with a wide variety of types. In the next article, we'll dive even deeper into how interfaces work behind the scenes and explore some more advanced topics.  
Happy coding!  
go (8 Part Series)  
1  
GoLang 101: Getting Started with Go  
2  
GoLang 101: Mastering Variables, Types, and Memory in Go  
...  
4 more parts...  
3  
GoLang 101: Making Decisions and Loops with Control Flow  
4  
GoLang 101: Organizing Data in Go with Arrays, Slices, Maps, and Structs  
5  
GoLang 101: Communicating with the World — Files, JSON, and Protocols  
6  
GoLang101: Mastering Functions  
7  
GoLang101: Go’s Minimal Object-Oriented Style  
8  
GoLang 101: Understanding Polymorphism Through Interfaces

# Перевод на русский

go (8 Part Series)  
1  
GoLang 101: Getting Started with Go  
2  
GoLang 101: Mastering Variables, Types, and Memory in Go  
...  
4 more parts...  
3  
GoLang 101: Making Decisions and Loops with Control Flow  
4  
GoLang 101: Organizing Data in Go with Arrays, Slices, Maps, and Structs  
5  
GoLang 101: Communicating with the World — Files, JSON, and Protocols  
6  
GoLang101: Mastering Functions  
7  
GoLang101: Go’s Minimal Object-Oriented Style  
8  
GoLang 101: Understanding Polymorphism Through Interfaces  
Hey everyone! 👋 Welcome back to our GoLang 101 series.  
In this article, we’re diving into a concept called polymorphism.  
If you’ve worked with languages like PHP or Python, you’re likely familiar with polymorphism being closely tied to inheritance. But Go takes a simpler and more elegant path — no inheritance required. Instead, Go uses interfaces to unlock powerful polymorphic behavior.  
Let’s see how this works.  
What is Polymorphism?  
At its core, polymorphism is the ability for an object to take on different forms depending on the context. A simple way to think about this is a function or method with a single name that performs different actions based on the type of object it's acting on.  
Take the concept of  
area  
, for example. If you want to compute the area of a rectangle, the calculation is  
base \* height  
. But for a triangle, it's  
(1/2) \* base \* height  
. The function  
area  
does two different things depending on whether it's dealing with a rectangle or a triangle.  
This is polymorphism in action, the  
area  
method is polymorphic because it has different forms depending on the object.  
In many object-oriented languages, polymorphism is supported through inheritance. This is where classes have a "superclass" (parent) and "subclass" (child) relationship, and the subclass inherits the methods and data of the superclass. The subclass can then redefine, or "override" a method it inherited from the superclass to provide its own specific implementation.  
But Go has no inheritance.  
So how does it achieve polymorphism? Let’s see.  
Go's Solution: The Power of Interfaces  
Go uses interfaces to define behavior in a clean and flexible way.  
An interface is a set of method signatures — it defines what a type must do, not how it does it.  
Example: A Shape Interface:  
type  
Shape2D  
interface  
{  
Area  
()  
float64  
Perimeter  
()  
float64  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Any type that implements both Area() and Perimeter() methods automatically satisfies the Shape2D interface — no implements keyword needed.  
Let's define Rectangle and Triangle:  
type  
Rectangle  
struct  
{  
Width  
,  
Height  
float64  
}  
func  
(  
r  
Rectangle  
)  
Area  
()  
float64  
{  
return  
r  
.  
Width  
\*  
r  
.  
Height  
}  
func  
(  
r  
Rectangle  
)  
Perimeter  
()  
float64  
{  
return  
2  
\*  
(  
r  
.  
Width  
+  
r  
.  
Height  
)  
}  
Enter fullscreen mode  
Exit fullscreen mode  
type  
Triangle  
struct  
{  
Base  
,  
Height  
,  
SideA  
,  
SideB  
float64  
}  
func  
(  
t  
Triangle  
)  
Area  
()  
float64  
{  
return  
0.5  
\*  
t  
.  
Base  
\*  
t  
.  
Height  
}  
func  
(  
t  
Triangle  
)  
Perimeter  
()  
float64  
{  
return  
t  
.  
Base  
+  
t  
.  
SideA  
+  
t  
.  
SideB  
}  
Enter fullscreen mode  
Exit fullscreen mode  
Both of these types now satisfy the  
Shape2D  
interface, no explicit declaration needed.  
Now that we have polymorphism through interfaces, we can write flexible functions.  
func  
FitsInYard  
(  
s  
Shape2D  
)  
bool  
{  
return  
s  
.  
Area  
()  
<  
100  
&&  
s  
.  
Perimeter  
()  
<  
100  
}  
Enter fullscreen mode  
Exit fullscreen mode  
This one function works with any type that satisfies Shape2D, Rectangle, Triangle, or even custom shapes in the future.  
You get code reuse, flexibility, and clarity — all without inheritance.  
Disambiguation with Type Assertions  
While interfaces are great for hiding differences, sometimes you need to "peel it apart" and figure out the exact underlying concrete type. This is especially useful in a program like a graphics application where you might have an  
DrawShape  
function that needs to call specific drawing APIs for different shapes (e.g.,  
DrawRectangle  
,  
DrawTriangle  
).  
For this, Go provides type assertions. A type assertion provides access to an interface value's underlying concrete value.  
func  
DrawShape  
(  
s  
Shape2D  
)  
{  
// Check if the underlying type is a Rectangle  
if  
rect  
,  
ok  
:=  
s  
.  
(  
Rectangle  
);  
ok  
{  
DrawRect  
(  
rect  
)  
}  
else  
if  
tri  
,  
ok  
:=  
s  
.  
(  
Triangle  
);  
ok  
{  
DrawTriangle  
(  
tri  
)  
}  
}  
Enter fullscreen mode  
Exit fullscreen mode  
A more convenient way to handle this is with a type switch, which is a special form of the  
switch  
statement for type assertions.  
func  
DrawShape  
(  
s  
Shape2D  
)  
{  
switch  
sh  
:=  
s  
.  
(  
type  
)  
{  
case  
Rectangle  
:  
DrawRect  
(  
sh  
)  
case  
Triangle  
:  
DrawTriangle  
(  
sh  
)  
// ... other cases  
}  
}  
Enter fullscreen mode  
Exit fullscreen mode  
The variable  
sh  
is now of the correct concrete type in each case block.  
A Common Use: Error Handling  
Another great use of interfaces is in error handling. In Go, many functions return two values: a result and an  
error  
.  
This  
error  
is actually an interface. The  
error  
interface is very simple; it has a single method called Error that returns a string.  
type  
error  
interface  
{  
Error  
()  
string  
}  
Enter fullscreen mode  
Exit fullscreen mode  
This simple interface allows any type to represent an error, as long as it has an Error() method. When a function returns an error, you should always check if it is  
nil  
. If it's not  
nil  
, it means something went wrong, and you should handle the error.  
f  
,  
err  
:=  
os  
.  
Open  
(  
"file.txt"  
)  
if  
err  
!=  
nil  
{  
fmt  
.  
Println  
(  
err  
)  
// Calls err.Error()  
return  
}  
Enter fullscreen mode  
Exit fullscreen mode  
This is a standard and robust way to handle errors in Go.  
Go's approach to polymorphism with interfaces is a powerful and flexible alternative to the traditional inheritance model. By focusing on behavior (methods) rather than data, interfaces allow you to write clean, reusable code that can work with a wide variety of types. In the next article, we'll dive even deeper into how interfaces work behind the scenes and explore some more advanced topics.  
Happy coding!  
go (8 Part Series)  
1  
GoLang 101: Getting Started with Go  
2  
GoLang 101: Mastering Variables, Types, and Memory in Go  
...  
4 more parts...  
3  
GoLang 101: Making Decisions and Loops with Control Flow  
4  
GoLang 101: Organizing Data in Go with Arrays, Slices, Maps, and Structs  
5  
GoLang 101: Communicating with the World — Files, JSON, and Protocols  
6  
GoLang101: Mastering Functions  
7  
GoLang101: Go’s Minimal Object-Oriented Style  
8  
GoLang 101: Understanding Polymorphism Through Interfaces