GopherCon 2019

Detecting Incompatible API Changes

Jonathan Amsterdam

Google, Inc.

jba@google.com

My Problem

Avoid breaking changes in a large set of packages.

Too many to check by hand.

Also, too hard to check.

old

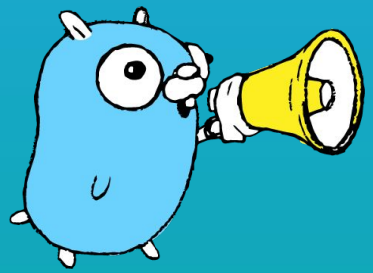
```
type S struct {
    A int
}
```

new

```
type S struct {
    A int
    b []int
}
```

I’ll build a tool for API compatibility.

How hard could it be?



Outline

Why Compatibility?

What is Compatibility?

Compatibility Basics

Correspondence

Compatibility Rules

Odds & Ends

Why Compatibility?

Incompatible API changes will break your users.

Do you want to break your users?

You don’t want to *unknowingly* break your users.

Modules and Semantic Versioning

- Hack, hack, hack
- Prepare module release
- Compatible changes? Increment minor version.
- Incompatible changes? Think hard.
  - Major version changes are a pain.

What is Compatibility?

Oh, no, not “What is Compatibility?” again

- Descent into Hyrum’s tarpit
  - “All observable behaviors of your system will be depended on by somebody.”
- Security fixes?
- Bug fixes?
- Performance improvements?

What do we want compatibility to be?

What compatibility means for the tool

- Computable
- Useful
- Easy to understand
- ...so inevitably limited.

A compatibility tool can only *advise;* it cannot decide.

Strict Compile-Time Compatibility

First attempt at a definition:

A change to a package is compatible if any code using that package still compiles.

Strict compile-time compatibility is not useful

old
```
type Point struct { X, Y int }
```

new
```
type Point struct { X, Y, Z int }
```

client
```
var p struct { X, Y int } = pkg.Point{}
```

Modified Compile-Time Compatibility

We'll ignore some code patterns:

- "spoofed" struct literals (previous slide)
- Unkeyed struct literals `pkg.Point{1, 2}`
- Use of the unsafe package
- ...and more

See the Go 1 Compatibility Promise.

Compatibility Tip #1

Use an unexported, zero-width field so clients can’t write unkeyed struct literals.

```
type Point struct {
    _ struct{}
    X, Y int
}
```

Our Story So Far

- A new version of a package is compatible with the old if programs continue to compile, with a few unusual exceptions.
- A tool can figure that out.
- But ultimately, a person must decide whether to bump the major version.

Compatibility Basics

The Fundamental Rule of Compatibility

You can add, but you can't change or remove.

Example

old
```
type Point struct { X, Y int }
```

new
```
type Point struct { X, Y, Z int }
```

compatible!

old
```
type Point struct { X, Y int }
```

new
```
type Point struct { X    int }
```

incompatible!

But wait a minute...

old/pkg.go
```
type Point struct { X, Y int }
```

new/pkg.go
```
type Point struct { X, Y, Z int }
```

How are we matching the “old” and “new” types?

They are effectively in different packages.

You can match old and new constants, variables and functions by name.

But not types.

old

```
type Point struct { X, Y int }
```

new

```
type p struct { X, Y, Z int }
type Point = p
```

The “same” type can have a different name.

Correspondence

Matching an old type with a new one

old

```
type Point struct { X, Y int }
```

new

```
type p struct { X, Y, Z int }
type Point = p
```

Aliases are part of the story.

old
```
type point struct { X, Y int }
```

new
```
type point struct { X    int }
```

both
```
var P point
```

client
```
pkg.P.Y
```

Unexported types can be exposed in the API.

old

```
type point struct { X, Y int }
var P point
```

new

```
type vertex struct { X, Y int }
var P vertex
```

client

```
pkg.P.X + pkg.P.Y
```

Unexported symbols can be renamed without changing the API!

Correspondence

An old and a new type name *correspond* if they are the same, or you can rename one to the other without changing the API.

Implementing Correspondence

While walking the exported symbols, assert correspondences.

old

```
type point struct { X, Y int }
var P point
```

new

```
type vertex struct { X, Y int }
var P vertex
```

Correspondence Extended

Use Go's definition of type identity, but replace "identical" with "correspond."

Spec: Two slice types are identical if they have identical element types.

⇒ Two slice types *correspond* if they have *corresponding* element types.

old `[]point`

new `[]vertex`

Compatibility Rules

Compatibility Rules!

1/10 Top-level Symbol Compatibility

Start with exported top-level names: constants, variables, functions and types

- If an old name is not in the new package, it was removed: incompatible change
- If a new name is not in the old package, it was added: compatible change

2/10 Variable Compatibility

A new exported variable is compatible with an old one of the same name if and only if their types correspond.

3/10 Function Compatibility

A new exported function is compatible with an old function of the same name if their types (signatures) correspond.

The new “function” can be a variable.

old
```
func F() { ... }
```

new
```
var F = func() { ... }
```

Call Compatibility

It is possible to change a function signature without breaking calls.

old
```
func F(int)
```

new
```
func F(int, ...string)
```

But not assignments.

client
```
var f func(int) = pkg.F
```

4/10 Constants

A new exported constant is compatible with an old one of the same name if and only if

- Their types correspond
- Their values are identical

Constant rule justified

Isn’t it OK to “de-type” a constant?

old
```
const C int64 = 1
```

new
```
const C = 1
```

No.

client
```
var x = C // x is int64 with old, int with new
var y int64 = x
```

Constant rule justified, 2

What about changing a value?

old
```
const C = 1
```

new
```
const C = 2
```

No.

client
```
var a [C]int = [1]int{}
```

Defined Types

```
type Number int
```

What can you do with a defined type?

- Use its underlying type

```
var n Number = 3
```

- Use a method on it

```
s := n.String()
```

Four Varieties of Methods

```
type Number int

func (Number) ExportedValue() {}

func (*Number) ExportedPointer() {}

func (Number) unexportedValue() {}

func (*Number) unexportedPointer() {}
```

5/10 Defined Type Compatibility

A new defined type T is compatible with an old one that corresponds if and only if

- The underlying types are compatible.
- The new exported method set of **T** is a superset of the old.
- The new exported method set of ***T** is a superset of the old.

Changing a value method to a pointer method

old

```
type T int
func (T) V() {}
```

new

```
type T int
func (*T) V() {}
```

client

```
pkg.T(1).V()
```

Changing a pointer method to a value method

old

```
type T int
func (*T) P() {}
```

new

```
type T int
func (T) P() {}
```

client

```
new(pkg.T).P()
```

Beware behavior changes!

old

```
type T int
func (t *T) P() { *t = 1 }
```

new

```
type T int
func (t T) P() { t = 1 }
```

client

```
var t T = 2
t.P()
fmt.Println(t) // 1 with old, 2 with new
```

6/10 Channel Compatibility

A new channel type is compatible with an old one if

1. The element types correspond, and
2. Either the directions are the same, or the new type has no direction.

old `chan<- int`

new `chan int`

Interface example #1

old

```
type I interface { A() }
```

new

```
type I interface { A(); B() }
```

client

```
type T struct{}
func (T) A() { ... }

var i pkg.I = T{}
```

Interface example #2

old

```
type I interface { A(); u() }
```

new

```
type I interface { A(); u(); B() }
```

client

```
type T struct{ pkg.I }
func (T) A() { ... }

var i pkg.I = T{}  // OK
```

7a/10 Interface Compatibility, Rule 1

A new interface is compatible with the corresponding old one if

- the old one does not have unexported methods, and
- the new one has identical (exported) method sets.

7b/10 Interface Compatibility, Rule 2

A new interface is compatible with the corresponding old one if

- the old one *does* have an unexported method, and
- the new exported method set is a superset of the old.

Compatibility Tip #2

When you define an interface, consider adding an unexported method to aid backwards compatibility.

```
type Server interface {
    Find()
    List()
    mustEmbedToImplement()
}
```

old

```
type Callback interface { A(); u() }

func F(c Callback) { ... c.A(); ... }
```

new

```
type Callback interface { A(); B(); u() }

func F(c Callback) { ... c.A(); c.B(); ... }
```

client

```
type MyCallback struct { pkg.Callback }
func (MyCallback) A() { ... }

pkg.F(MyCallback{}) // panics with new
```

“Add” to an interface by defining a new one that embeds the old.

old

```
type Callback interface { A() }
func F(c Callback) { ... c.A(); ... }
```

new

```
type Callback2 interface { Callback; B() }
func F(c Callback) {
    if c2, ok := c.(Callback2); ok {
        c2.B();
    }
}
```

What can you do with a struct?

```
type S struct { A, B int }
```

Write a struct literal

```
s := S{A: 1, B: 2}
```

Select a field

```
a := s.A
```

Compare

```
s == S{A: 2, B: 1}
```

Struct example #1

old

```
type S struct { A, B, C, D int }
```

new

```
type S struct { A int; embed1; embed2 }
type embed1 struct { B, C int }
type embed2 struct { D int }
```

client

```
var s pkg.S
fmt.Print(s.A, s.B, s.C, s.D)  // OK
s = pkg.S{A: 1, B: 2}          // fails in new
```

Struct example #2

old

```
type S struct { A int; embed1; embed2 }
type embed1 struct { B, C int }
type embed2 struct { D int }
```

new

```
type S struct { A int; embed1; embed2 }
type embed1 struct { B int }
type embed2 struct { C, D int }
```

8a/10 Struct Compatibility, Rules 1 and 2

1. The new set of top-level exported fields is a superset of the old.

2. The new set of selectable exported fields is a superset of the old.

Comparability

Comparable: can be used with ==, !=, or as a map key.

- int, string, ...: comparable
- slices, maps, functions: not comparable
- structs: if all fields are comparable

Struct example: comparison

old

```
type S struct {
    A int
}
```

new

```
type S struct {
    A int
    b []int
}
```

client

```
pkg.S{} == pkg.S{}
```

8b/10 Struct Compatibility, Rule 3

3. If the old struct is comparable, so is the new one.

Compatibility Tip #3

Use an unexported, zero-width, *non-comparable* field to prevent clients from comparing a struct.

```
type Point struct {
    _ [0]func()
    X, Y int
}
```

9/10 Numeric Type Compatibility

Making a numeric type larger without changing its kind (signed int, unsigned int, float, complex) is a compatible change.

int32 → int64
float32 → float64

old

```
var N int32
```

new

```
var N int64
```

client

```
pkg.N = int32(1)
```

A Major Simplification

You have to name a type to change it.

old
```
type Num int32
```

new
```
type Num int64
```

both
```
var N Num
```

client
```
var n pkg.Num = pkg.Num(1)  // OK
```

Compatible

old

```
var P S
type S struct {
    X int
}
```

new

```
var P S
type S struct {
    X, Y int
}
```

Incompatible

old

```
var P struct {
    X int
}
```

new

```
var P struct {
    X, Y int
}
```

pkg
```
type Point struct { X, Y int }
```

client
```
var p struct { X, Y int } = pkg.Point{}
```

“Use the name!”

pkg
```
var P struct { X, Y int }
```

client
```
var p struct { X, Y int } = pkg.P
```

“Umm...”

old
```
var P struct { X, Y int }
```

new
```
var P struct { X, Y, Z int }
```

both
```
var Q struct { X, Y int }
```

client
```
pkg.P == pkg.Q
```

A seemingly compatible change that isn't

old

```
type T int
func (T) m() {}
type I interface { m() }
```

new

```
type T int

type I interface { m() }
```

client

```
var i pkg.I = pkg.T{}
```

10/10 Whole-Package Compatibility

If an old type implements an old interface, the corresponding new type should implement the corresponding new interface.

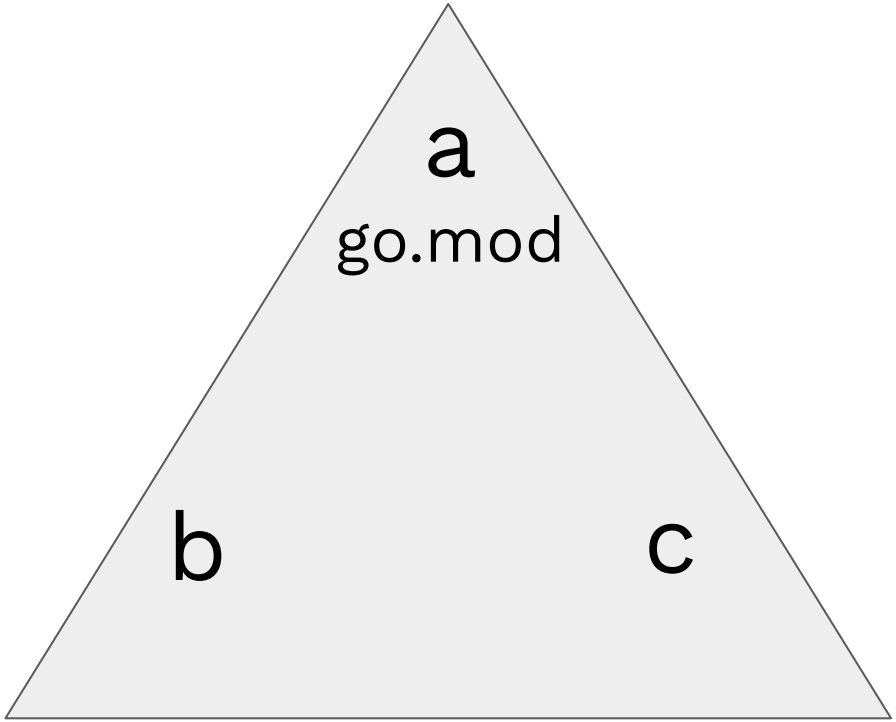
Odds & Ends

Module Compatibility

A new module is compatible with an old one if

- every visible old package is compatible with the new one of the same name, and
- the new package is located in either
  - the new module, or
  - a required nested module.

Creating a Nested Module: Before



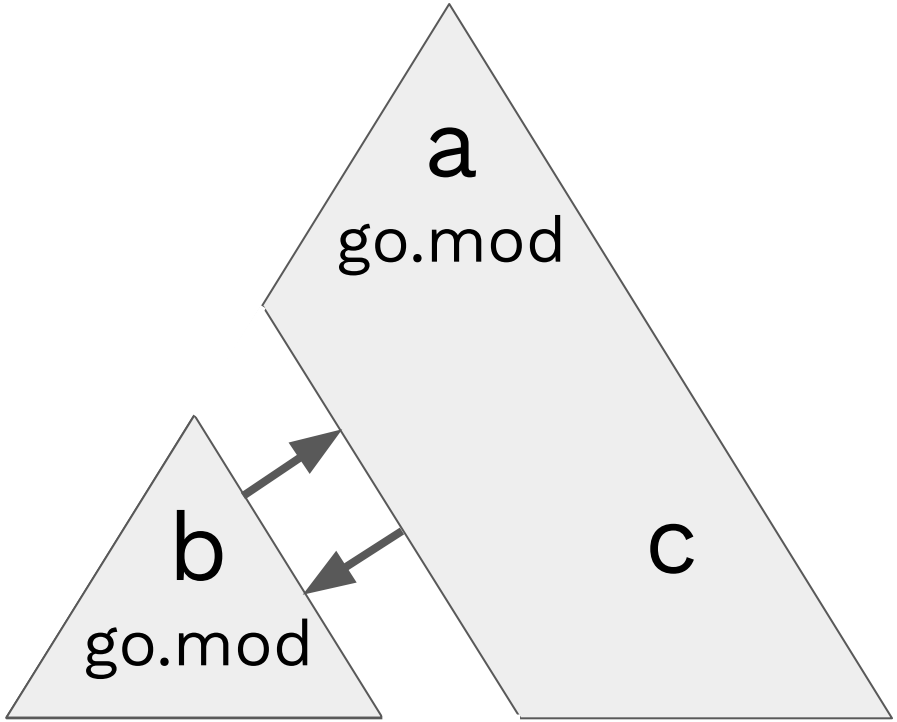
packages:

a

a/b

a/c

Creating a Nested Module: After



packages:

a

a/b

a/c

Implementation

go/types, go/constant, and golang.org/x/tools/go/packages are awesome

```
if types.Comparable(old) &&
   !types.Comparable(new) {
   ... // incompatible
}
```

Computing the selectable field set:

Breadth-first walk of embedded structs

Write export data to make it easy to compare across commits.

golang.org/x/tools/go/gcexportdata

Next Steps

Read the “spec”:

https://golang.org/s/apidiff-readme

Play with the tool:

```
go get golang.org/x/exp/cmd/apidiff
```

But don’t get too attached to it...

gorelease will replace it.

THE END

Thank you.