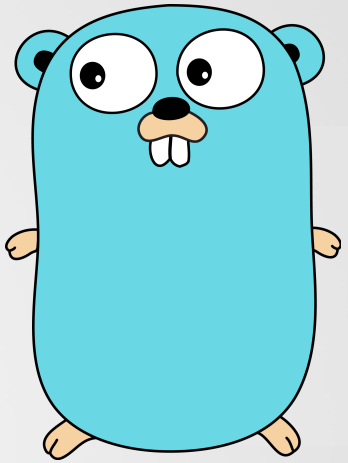
How the Go runtime implement maps efficiently

David Chou

We are Umbo Computer Vision

We build autonomous video security system

Golang Taipei Streaming Meetup



david74.chou @ facebook
david74.chou @ medium
david7482 @ github

How the Go runtime implements maps efficiently (without generics)

Dave Cheney, GoCon Spring 2018

https://dave.cheney.net/2018/05/29/how-the-go-runtime-implements-maps-efficiently-without-generics

Generics — Problem Overview

Russ Cox
August 27, 2018

Introduction

This overview and the accompanying detailed draft design are part of a collection of Go 2 draft design documents. The overall goal of the Go 2 effort is to address the most significant ways that Go fails to scale to large code bases and large developer efforts.

The Go team, and in particular Ian Lance Taylor, has been investigating and discussing possible designs for "generics" (that is, parametric polymorphism; see note below) since before Go's first open source release. We understood from experience with C++ and Java that the topic was rich and complex and would take a long time to understand well enough to design a good solution. Instead of attempting that at the start, we spent our time on features more directly applicable to Go's initial target of networked system software (now "cloud software"), such as concurrency, scalable builds, and low-latency garbage collection.

Draft Design

This section quickly summarizes the draft design, as a basis for high-level discussion and comparison with other approaches.

The draft design adds a new syntax for introducing a type parameter list in a type or function declaration: `(type` *<list of type names>* `)`. For example:

```
type List(type T) []T

func Keys(type K, V)(m map[K]V) []K
```

Go2 Generic Design Proposal

C++

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pa
> class unordered_map;
```

JAVA

```
Class HashMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values
```

Go

```
var m map[string]
```

The map function

map(key) → value

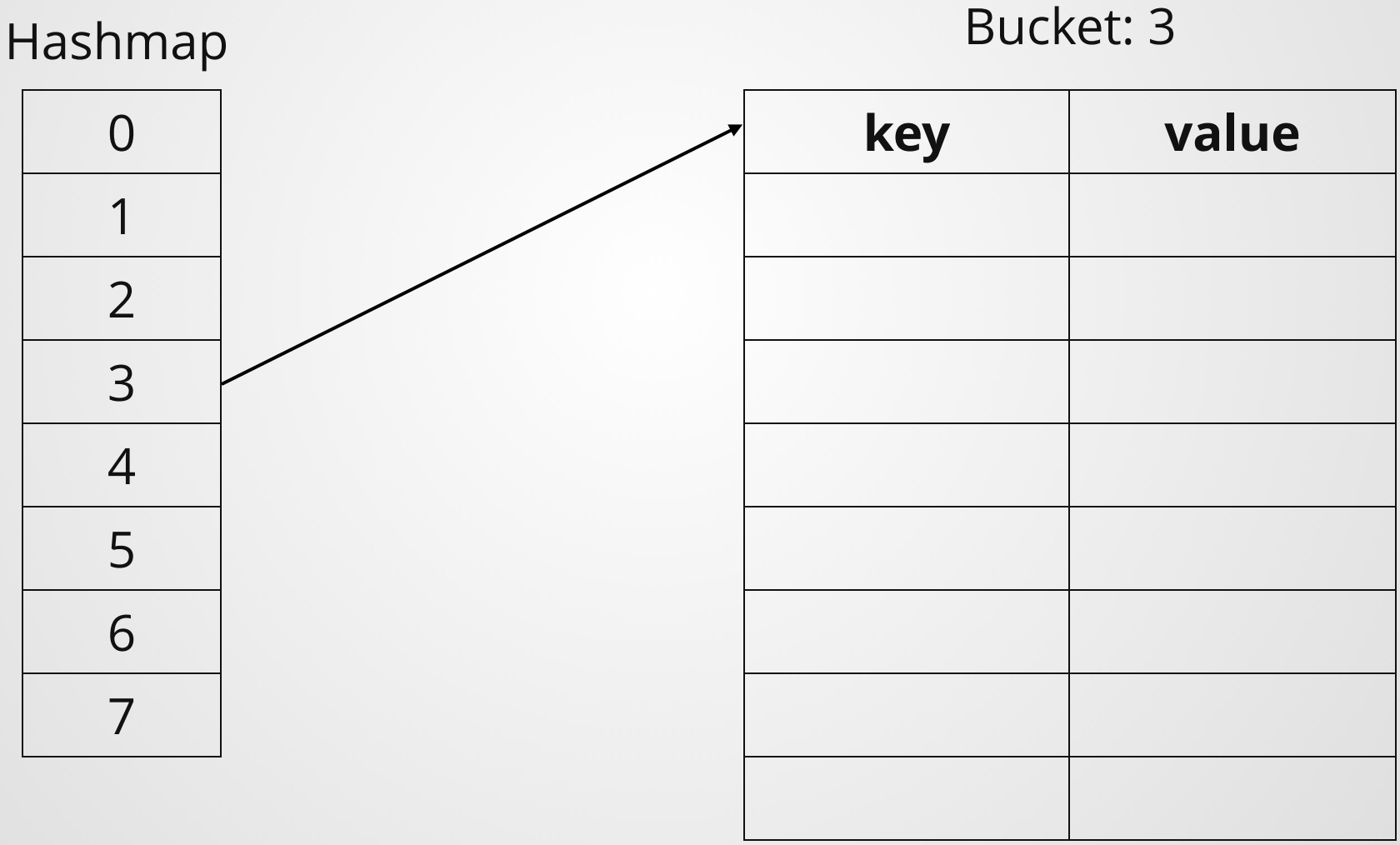
Go uses HashMap

Property \ Map	HashMap	TreeMap
Ordering	not guaranteed	sorted, natural ordering
get / put / remove complexity	O(1)	O(log(n))
Inherited interfaces	Map	Map NavigableMap SortedMap
NULL values / keys	allowed	only values

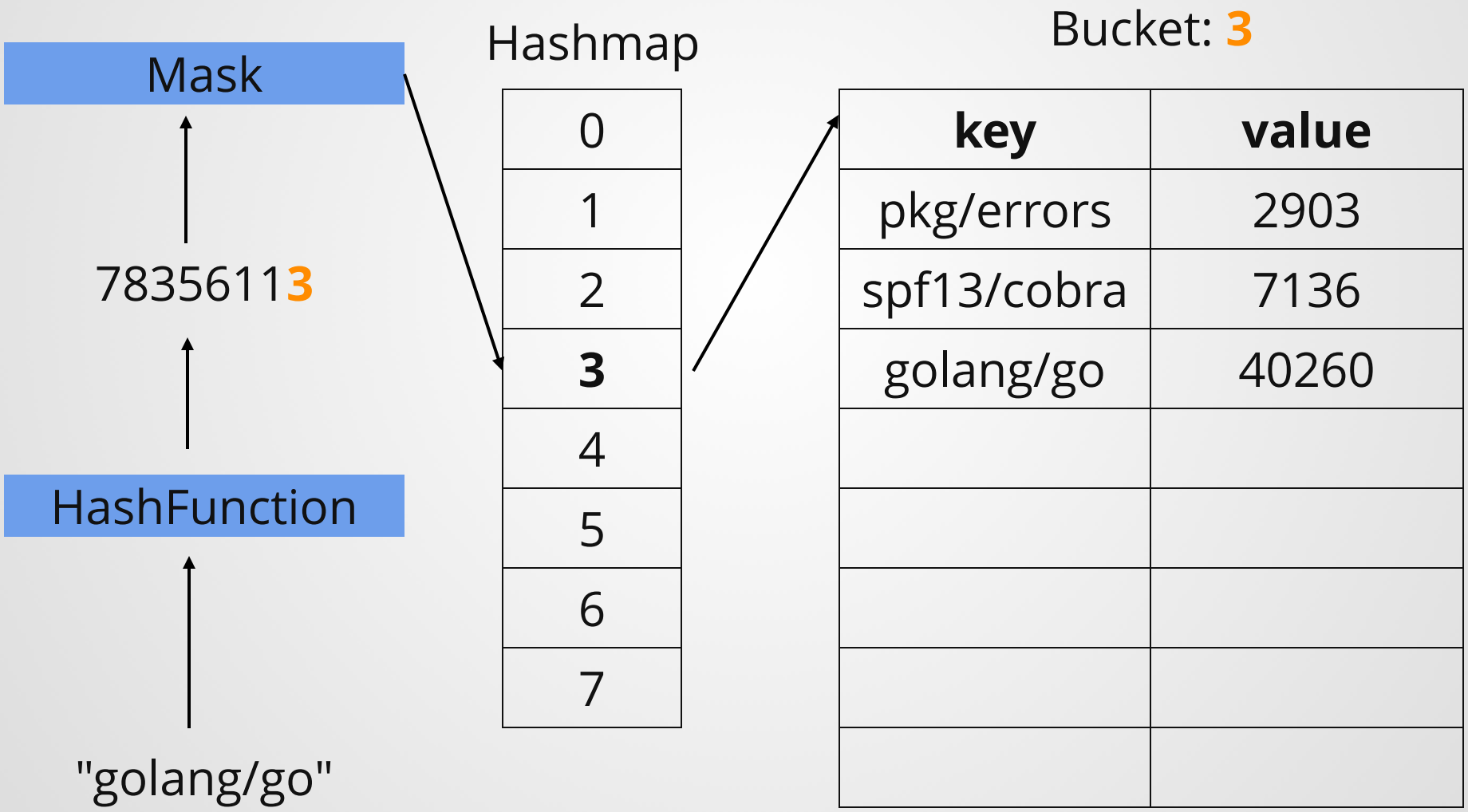
The hash function

hash(key) → integer

Hashmap Data Structure



insert(star, "golang/go", 40260)



Four properties of a hash map

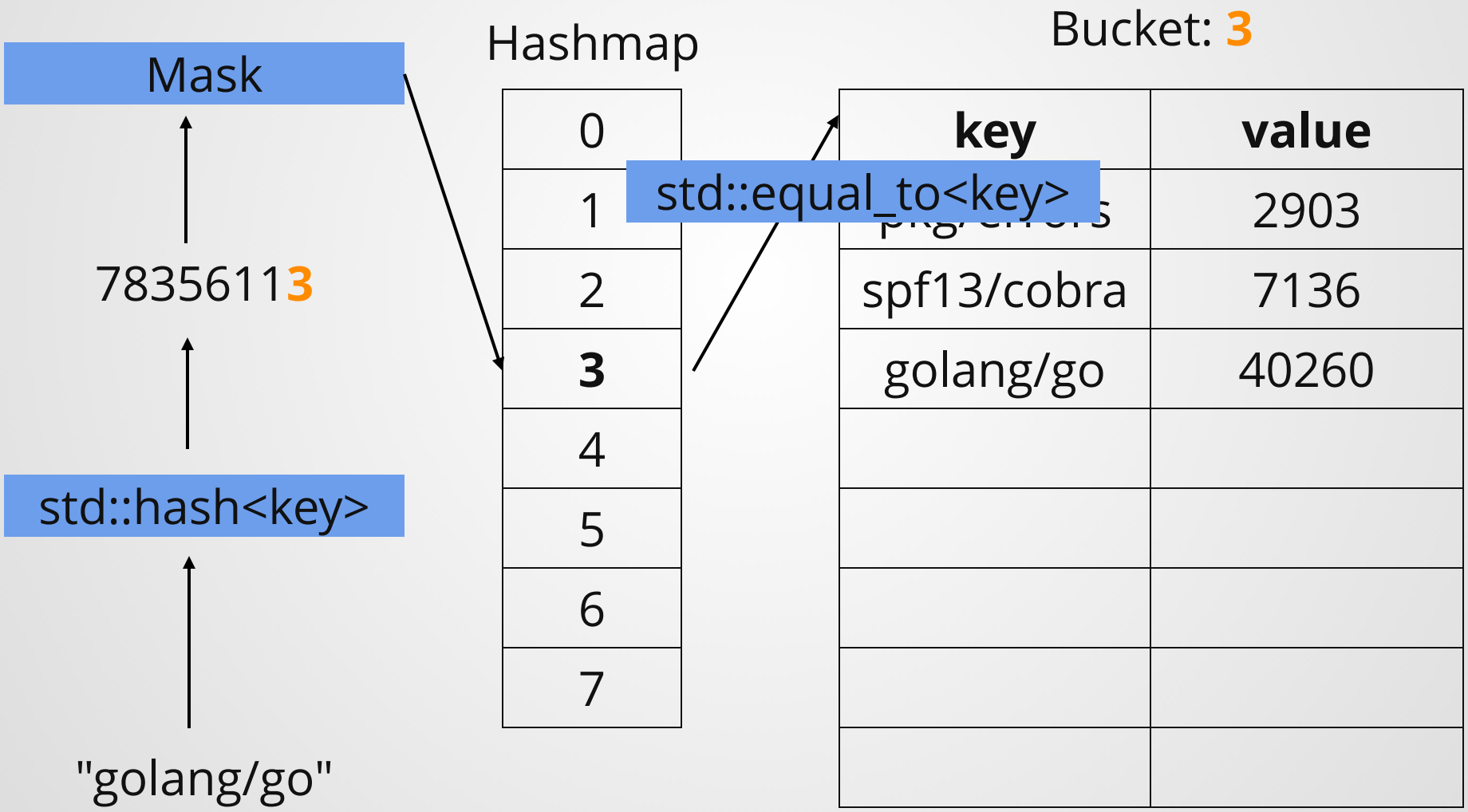
1. A hash function for the key
2. An equality function to compare keys
3. Need to know the size of the key type
4. Need to know the size of the value type

C++

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pa
> class unordered_map;
```

- class Key
- class T
- std::hash<Key>
- std::equal_to<Key>

insert(star, "golang/go", 40260)



JAVA

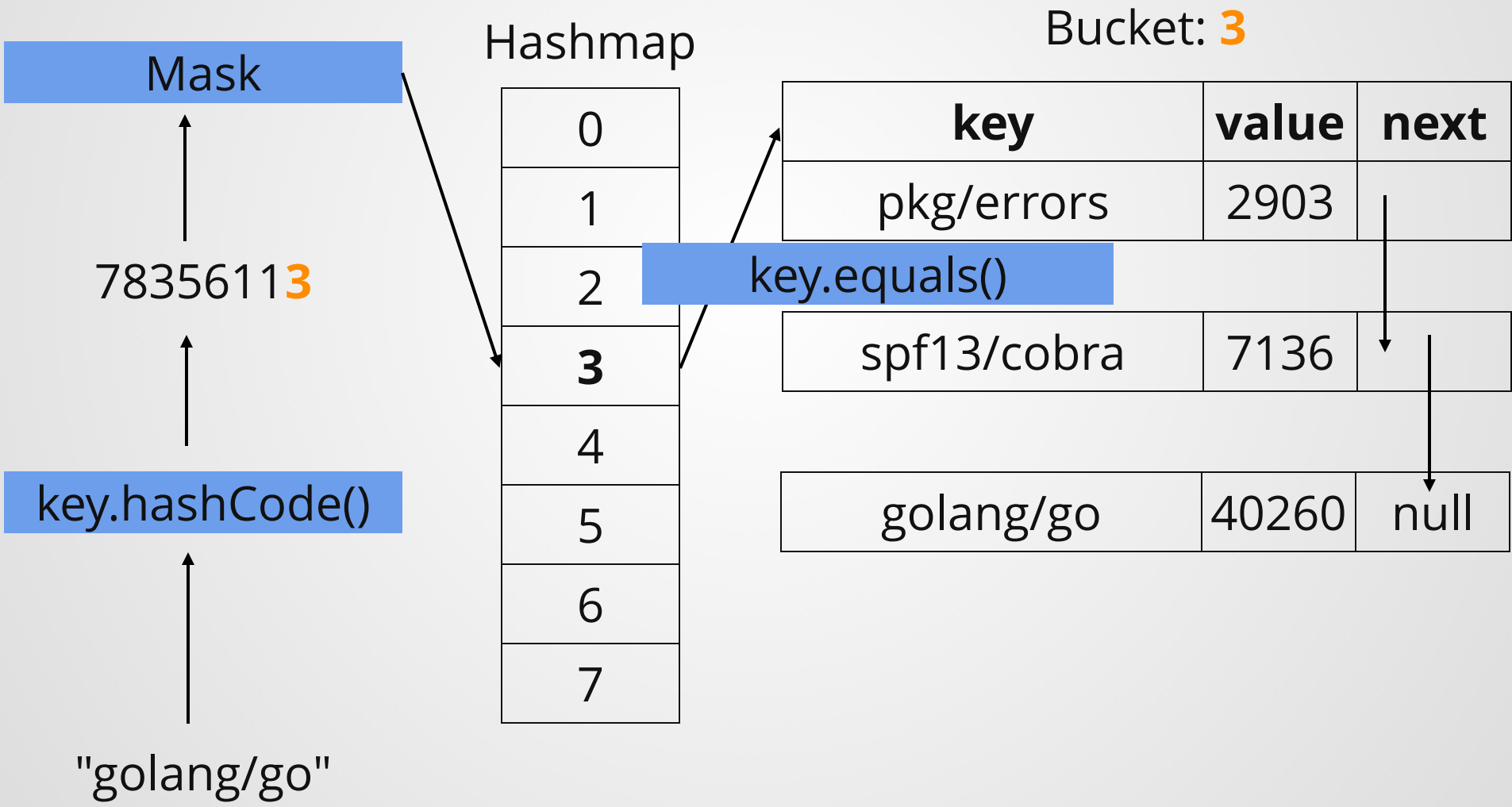
```
Class HashMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:
K - the type of keys maintained by this map
V - the type of mapped values
```

- K and V are Object
  - Object.equals()
  - Object.hashCode()
- Need *boxing* for primitive types

insert(star, "golang/go", 40260)



C++

- Pros
  - The size of key and value are always known
  - Array implementation
  - No need for boxing or pointer chasing
- Cons
  - Larger binary size. Different types means different maps.
  - Slower compile time.
  - Larger memory footprint for predetermined size for each array element.

JAVA

- Pros
  - Single implementation for any subclass of Object
  - Faster compile time and smaller binary size
  - Linked list implementation. No predetermined size for each array element.
- Cons
  - *Boxing* would increase gc preasure
  - Slower for boxing and linked list pointer chasing

Go's hashmap implementaion

Use interface{} ? No

Code generation ? No

Compiler + Runtime

Compile time rewriting

v := m["key"] → runtime.mapaccess1(m, ”key", &v)
v, ok := m["key"] → runtime.mapaccess2(m, ”key”, &v, &ok)
m["key"] = 9001 → runtime.mapinsert(m, ”key", 9001)
delete(m, "key") → runtime.mapdelete(m, “key”)

mapaccess1

func mapaccess1(**t *maptype**, h *hmap, key unsafe.Pointer) unsafe.Pointer

Different maptype values for each unique map declaration

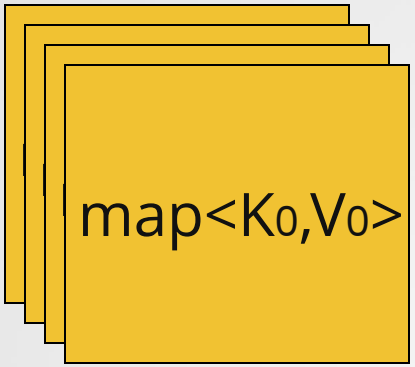
map[string]int → var mt1 maptype{...}
map[string]http.Header → var mt2 maptype{...}
map[structA]structB → var mt3 maptype{...}

```
type maptype struct {
        typ           _type
        key          *_type
        elem         *_type
        bucket        *_type // internal type representing a hash
       hmap           *_type // internal type representing a hmap
        keysize       uint8  // size of key slot
        indirectkey   bool   // store ptr to key instead of key i
        valuesize     uint8  // size of value slot
        indirectvalue bool   // store ptr to value instead of val
        bucketsize    uint16 // size of bucket
        reflexivekey  bool   // true if k==k for all keys
        needkeyupdate bool   // true if we need to update key on
}
```

```
type _type struct {
        size       uintp
        alg       *typeA
        ...
}
```

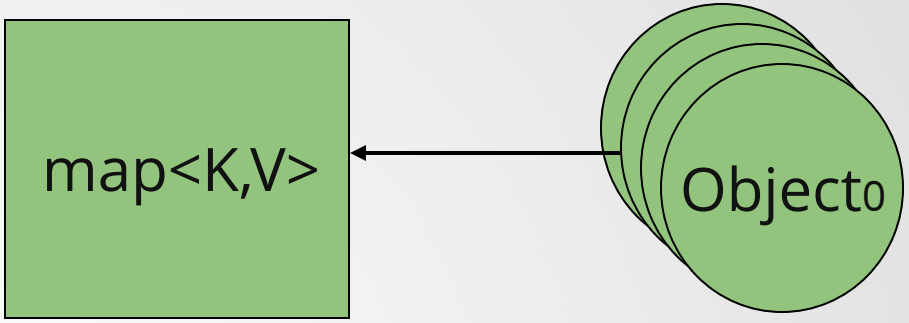
```
type typeAlg struct {
        // function for hashing objects
        // (ptr to object, seed) -> hash
        hash func(unsafe.Pointer, uintpt
        // function for comparing object
        // (ptr to object A, ptr to obje
       equal func(unsafe.Pointer, unsafe
}
```

C++



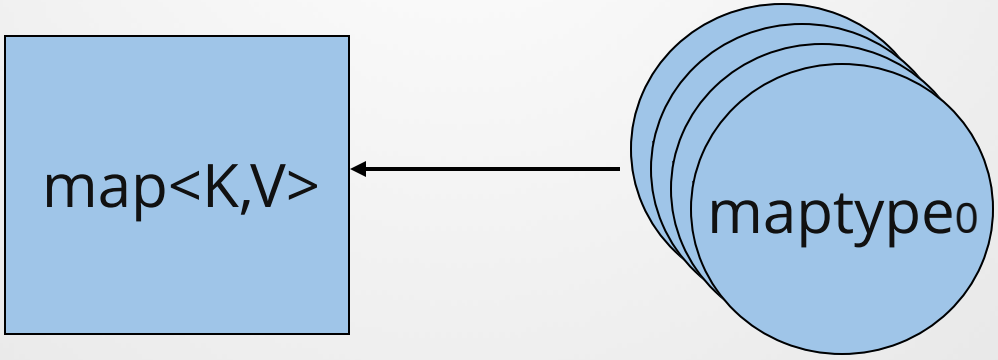
Compile Time

JAVA



Run Time

Go



Compile Time

Conclusion

- A good compromise between C++ and JAVA
- Single hashmap implementation to reduce binary size
- Already known the the size of key and value.
  Array implementation for better performance.
- Could use primitive types without boxing.
  No extra gc preasure


umbo
computer vision
Any Question?