Golang 中 []byte 与 string 转换

机器铃砍菜刀
102

发布于
2020-10-31

`string` 类型和 `[]byte` 类型是我们编程时最常使用到的数据结构。本文将探讨两者之间的转换方式，通过分析它们之间的内在联系来拨开迷雾。

两种转换方式

标准转换

go 中 `string` 与 `[]byte` 的互换，相信每一位 gopher 都能立刻想到以下的转换方式，我们将之称为标准转换。

```
// string to []byte
s1 := "hello"
b := []byte(s1)

// []byte to string
s2 := string(b)
```

强转换

通过 unsafe 和 reflect 包，可以实现另外一种转换方式，我们将之称为强转换（也常常被人称作黑魔法）。

```
func String2Bytes(s string) []byte {
    sh := (*reflect.StringHeader)(unsafe.Pointer(&s))
    bh := reflect.SliceHeader{
        Data: sh.Data,
        Len:  sh.Len,
        Cap:  sh.Len,
    }
    return *(*[]byte)(unsafe.Pointer(&bh))
}

func Bytes2String(b []byte) string {
```

```
    return *(*string)(unsafe.Pointer(&b))
}
```

性能对比

既然有两种转换方式，那么我们有必要对它们做性能对比。

```
// 测试强转换功能
func TestBytes2String(t *testing.T) {
    x := []byte("Hello Gopher!")
    y := Bytes2String(x)
    z := string(x)

    if y != z {
        t.Fail()
    }
}

// 测试强转换功能
func TestString2Bytes(t *testing.T) {
    x := "Hello Gopher!"
    y := String2Bytes(x)
    z := []byte(x)

    if !bytes.Equal(y, z) {
        t.Fail()
    }
}

// 测试标准转换 string() 性能
func Benchmark_NormalBytes2String(b *testing.B) {
    x := []byte("Hello Gopher! Hello Gopher! Hello Gopher!")
    for i := 0; i < b.N; i++ {
        _ = string(x)
    }
}

// 测试强转换 []byte 到 string 性能
func Benchmark_Byte2String(b *testing.B) {
    x := []byte("Hello Gopher! Hello Gopher! Hello Gopher!")
    for i := 0; i < b.N; i++ {
        _ = Bytes2String(x)
    }
}

// 测试标准转换 []byte 性能
func Benchmark_NormalString2Bytes(b *testing.B) {
```

```
    x := "Hello Gopher! Hello Gopher! Hello Gopher!"
    for i := 0; i < b.N; i++ {
        _ = []byte(x)
    }
}

// 测试强转换 string 到 []byte 性能
func Benchmark_String2Bytes(b *testing.B) {
    x := "Hello Gopher! Hello Gopher! Hello Gopher!"
    for i := 0; i < b.N; i++ {
        _ = String2Bytes(x)
    }
}
```

测试结果如下

```
$ go test -bench="." -benchmem
goos: darwin
goarch: amd64
pkg: workspace/example/stringBytes
Benchmark_NormalBytes2String-8          38363413                27.9 ns/op            4
Benchmark_Byte2String-8                 1000000000               0.265 ns/op
Benchmark_NormalString2Bytes-8          32577080                34.8 ns/op            4
Benchmark_String2Bytes-8                1000000000               0.532 ns/op
PASS
ok      workspace/example/stringBytes   3.170s
```

注意，`-benchmem` 可以提供每次操作分配内存的次数，以及每次操作分配的字节数。

当 x 的数据均为 "Hello Gopher!" 时，测试结果如下

```
$ go test -bench="." -benchmem
goos: darwin
goarch: amd64
pkg: workspace/example/stringBytes
Benchmark_NormalBytes2String-8          245907674                4.86 ns/op
Benchmark_Byte2String-8                 1000000000               0.266 ns/op
Benchmark_NormalString2Bytes-8          202329386                5.92 ns/op
Benchmark_String2Bytes-8                1000000000               0.532 ns/op
PASS
ok      workspace/example/stringBytes   4.383s
```

强转换方式的性能会明显优于标准转换。

读者可以思考以下问题：

1.为啥强转换性能会比标准转换好？
2.为啥在上述测试中，当 x 的数据较大时，标准转换方式会有一次分配内存的操作，从而导致其性能更差，而强转换方式却不受影响？
3.既然强转换方式性能这么好，为啥 go 语言提供给我们使用的是标准转换方式？

原理分析

要回答以上三个问题，首先要明白是 `string` 和 `[]byte` 在 go 中到底是什么。

`[]byte`

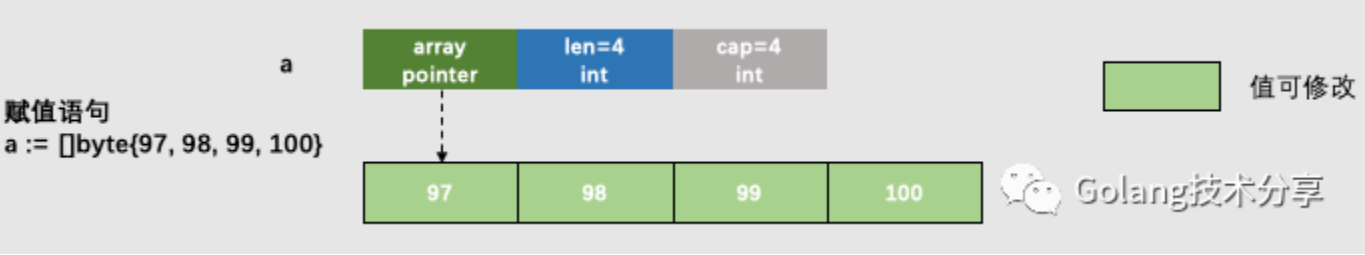
在 go 中，byte 是 uint8 的别名，在 go 标准库 builtin 中有如下说明：

```
// byte is an alias for uint8 and is equivalent to uint8 in all ways. It is
// used, by convention, to distinguish byte values from 8-bit unsigned
// integer values.
type byte = uint8
```

在 go 的源码中 `src/runtime/slice.go`，slice 的定义如下：

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

array 是底层数组的指针，len 表示长度，cap 表示容量。对于 `[]byte` 来说，array 指向的就是 byte 数组。



string

关于 string 类型，在 go 标准库 builtin 中有如下说明：

```
// string is the set of all strings of 8-bit bytes, conventionally but not
// necessarily representing UTF-8-encoded text. A string may be empty, but
// not nil. Values of string type are immutable.
type string string
```

翻译过来就是：string 是 8 位字节的集合，通常但不一定代表 UTF-8 编码的文本。string 可以为空，但是不能为 nil。**string 的值是不能改变的。**

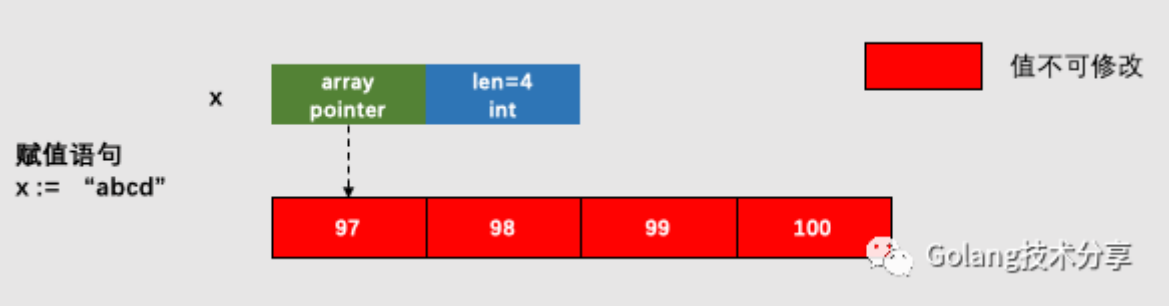
在 go 的源码中 `src/runtime/string.go`，string 的定义如下：

```
type stringStruct struct {
    str unsafe.Pointer
    len int
}
```

stringStruct 代表的就是一个 string 对象，str 指针指向的是某个数组的首地址，len 代表的数组长度。那么这个数组是什么呢？我们可以在实例化 stringStruct 对象时找到答案。

```
//go:nosplit
func gostringnocopy(str *byte) string {
    ss := stringStruct{str: unsafe.Pointer(str), len: findnull(str)}
    s := *(*string)(unsafe.Pointer(&ss))
    return s
}
```

可以看到，入参 str 指针就是指向 byte 的指针，那么我们可以确定 string 的底层数据结构就是 byte 数组。



综上，`string` 与 `[]byte` 在底层结构上是非常的相近（后者的底层表达仅多了一个 cap 属性，因此它们在内存布局上是可对齐的），这也就是为何 builtin 中内置函数 copy 会有一种特殊情况 `copy(dst []byte, src string) int` 的原因了。

```
// The copy built-in function copies elements from a source slice into a
// destination slice. (As a special case, it also will copy bytes from a
```

```
// string to a slice of bytes.) The source and destination may overlap. Copy
// returns the number of elements copied, which will be the minimum of
// len(src) and len(dst).
func copy(dst, src []Type) int
```

区别

对于 `[]byte` 与 `string` 而言，两者之间最大的区别就是 `string` 的值不能改变。这该如何理解呢？下面通过两个例子来说明。

对于 `[]byte` 来说，以下操作是可行的：

```
b := []byte("Hello Gopher!")
b [1] = 'T'
```

string，修改操作是被禁止的：

```
s := "Hello Gopher!"
s[1] = 'T'
```

而 string 能支持这样的操作：

```
s := "Hello Gopher!"
s = "Tello Gopher!"
```

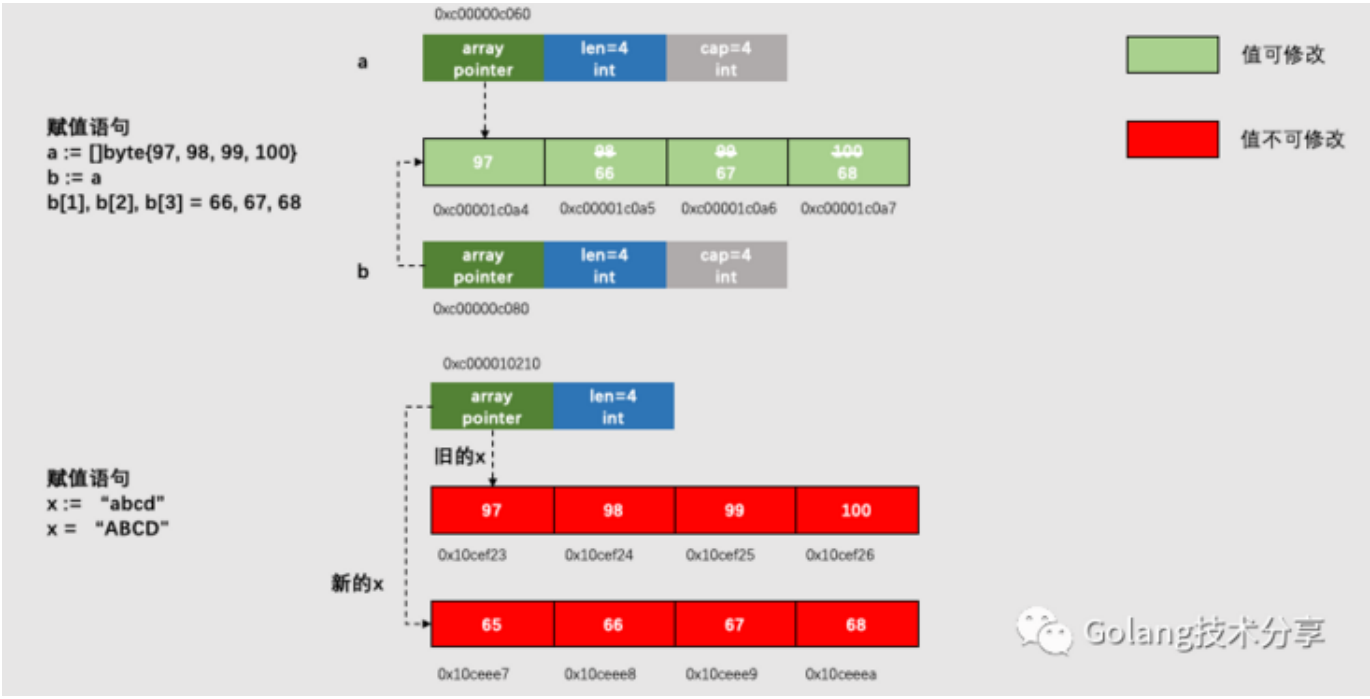
字符串的值不能被更改，但可以被替换。string 在底层都是结构体 `stringStruct{str: str_point, len: str_len}`，string 结构体的 str 指针指向的是一个字符常量的地址，这个地址里面的内容是不可以被改变的，因为它是只读的，但是这个指针可以指向不同的地址。

那么，以下操作的含义是不同的：

```
s := "S1" // 分配存储"S1"的内存空间，s 结构体里的 str 指针指向这块内存
s = "S2"  // 分配存储"S2"的内存空间，s 结构体里的 str 指针转为指向这块内存

b := []byte{1} // 分配存储'1'数组的内存空间，b 结构体的 array 指针指向这个数组。
b = []byte{2}  // 将 array 的内容改为'2'
```

图解如下



因为 string 的指针指向的内容是不可以更改的，所以每更改一次字符串，就得重新分配一次内存，之前分配的空间还需要 gc 回收，这是导致 string 相较于[]byte操作低效的根本原因。

标准转换的实现细节

`[]byte(string)` 的实现（源码在 `src/runtime/string.go` 中）

```
// The constant is known to the compiler.
// There is no fundamental theory behind this number.
const tmpStringBufSize = 32

type tmpBuf [tmpStringBufSize]byte

func stringtoslicebyte(buf *tmpBuf, s string) []byte {
    var b []byte
    if buf != nil && len(s) <= len(buf) {
        *buf = tmpBuf{}
        b = buf[:len(s)]
    } else {
        b = rawbyteslice(len(s))
    }
    copy(b, s)
    return b
}

// rawbyteslice allocates a new byte slice. The byte slice is not zeroed.
func rawbyteslice(size int) (b []byte) {
    cap := roundupsize(uintptr(size))
    p := mallocgc(cap, nil, false)
    if cap != uintptr(size) {
```

```
        memclrNoHeapPointers(add(p, uintptr(size)), cap-uintptr(size))
    }

    *(*slice)(unsafe.Pointer(&b)) = slice{p, size, int(cap)}
    return
}
```

这里有两种情况：s 的长度是否大于 32。当大于 32 时，go 需要调用 mallocgc 分配一块新的内存（大小由 s 决定），这也就回答了上文中的问题 2：当 x 的数据较大时，标准转换方式会有一次分配内存的操作。

最后通过 copy 函数实现 `string` 到 `[]byte` 的拷贝，具体实现在 `src/runtime/slice.go` 中的 `slicestringcopy` 方法。

```
func slicestringcopy(to []byte, fm string) int {
    if len(fm) == 0 || len(to) == 0 {
        return 0
    }

  // copy 的长度取决与 string 和 []byte 的长度最小值
    n := len(fm)
    if len(to) < n {
        n = len(to)
    }

  // 如果开启了竞态检测 -race
    if raceenabled {
        callerpc := getcallerpc()
        pc := funcPC(slicestringcopy)
        racewriterangepc(unsafe.Pointer(&to[0]), uintptr(n), callerpc, pc)
    }
  // 如果开启了 memory sanitizer -msan
    if msanenabled {
        msanwrite(unsafe.Pointer(&to[0]), uintptr(n))
    }

  // 该方法将 string 的底层数组从头部复制 n 个到 []byte 对应的底层数组中去（这里就是 copy 实
    memmove(unsafe.Pointer(&to[0]), stringStructOf(&fm).str, uintptr(n))
    return n
}
```

copy 实现过程图解如下

`string([]byte)` 的实现（源码也在 `src/runtime/string.go` 中）

```
// Buf is a fixed-size buffer for the result,
// it is not nil if the result does not escape.
func slicebytetostring(buf *tmpBuf, b []byte) (str string) {
    l := len(b)
    if l == 0 {
        // Turns out to be a relatively common case.
        // Consider that you want to parse out data between parens in "foo()bar",
        // you find the indices and convert the subslice to string.
        return ""
    }
  // 如果开启了竞态检测 -race
    if raceenabled {
        racereadrangepc(unsafe.Pointer(&b[0]),
            uintptr(l),
            getcallerpc(),
            funcPC(slicebytetostring))
    }
  // 如果开启了 memory sanitizer -msan
    if msanenabled {
        msanread(unsafe.Pointer(&b[0]), uintptr(l))
    }
    if l == 1 {
        stringStructOf(&str).str = unsafe.Pointer(&staticbytes[b[0]])
        stringStructOf(&str).len = 1
        return
    }

    var p unsafe.Pointer
    if buf != nil && len(b) <= len(buf) {
        p = unsafe.Pointer(buf)
    } else {
        p = mallocgc(uintptr(len(b)), nil, false)
    }
```

```
    stringStructOf(&str).str = p
    stringStructOf(&str).len = len(b)
  // 拷贝字节数组至字符串
    memmove(p, (*(*slice)(unsafe.Pointer(&b))).array, uintptr(len(b)))
    return
}

// 实例 stringStruct 对象
func stringStructOf(sp *string) *stringStruct {
    return (*stringStruct)(unsafe.Pointer(sp))
}
```

可见，当数组长度超过 32 时，同样需要调用 mallocgc 分配一块新内存。最后通过 memmove 完成拷贝。

强转换的实现细节

1. **万能的 `unsafe.Pointer` 指针**

   在 go 中，任何类型的指针 `*T` 都可以转换为 unsafe.Pointer 类型的指针，它可以存储任何变量的地址。同时，unsafe.Pointer 类型的指针也可以转换回普通指针，而且可以不必和之前的类型 `*T` 相同。另外，unsafe.Pointer 类型还可以转换为 uintptr 类型，该类型保存了指针所指向地址的数值，从而可以使我们对地址进行数值计算。以上就是强转换方式的实现依据。

   而 string 和 slice 在 reflect 包中，对应的结构体是 reflect.StringHeader 和 reflect.SliceHeader，它们是 string 和 slice 的运行时表达。

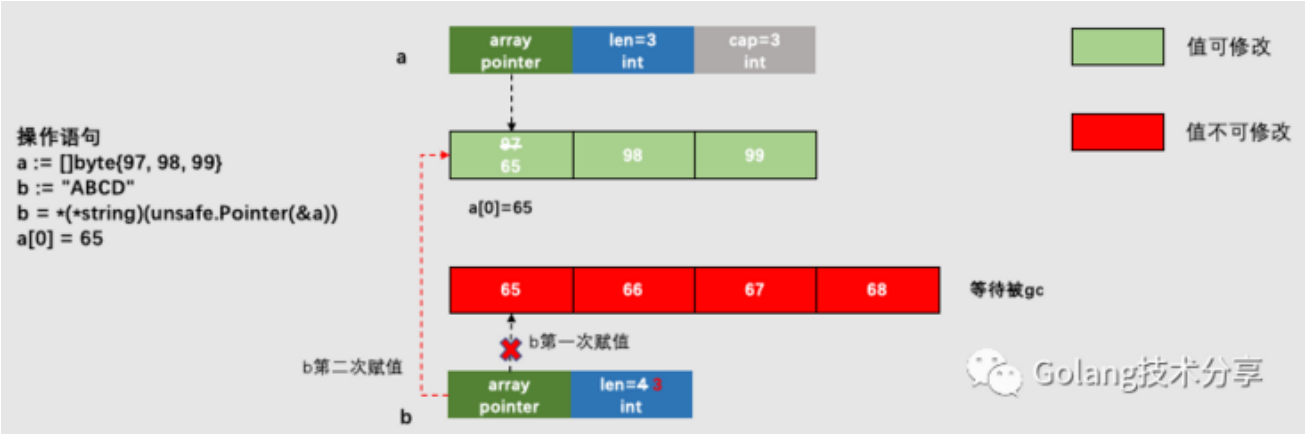
   ```
   type StringHeader struct {
       Data uintptr
       Len  int
   }

   type SliceHeader struct {
       Data uintptr
       Len  int
       Cap  int
   }
   ```

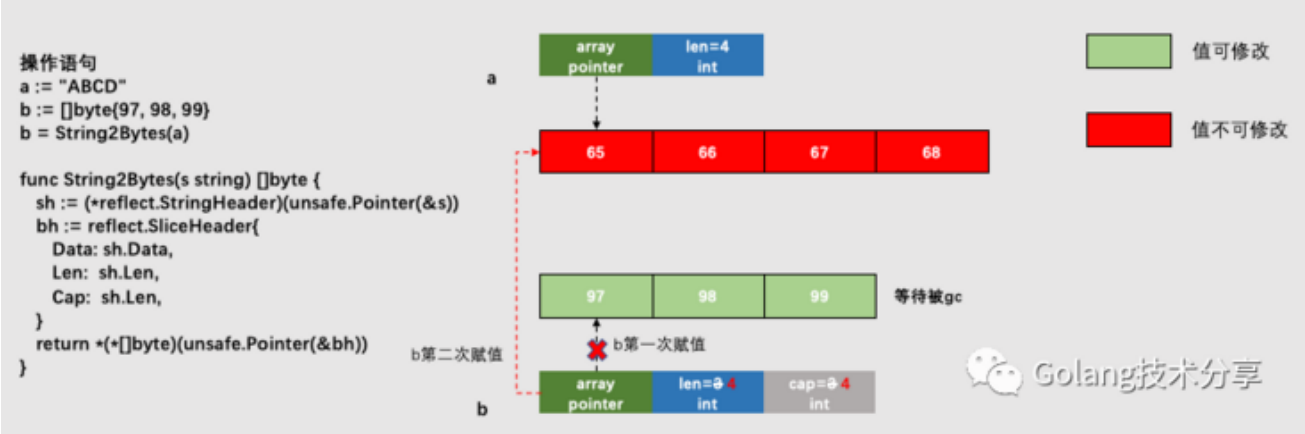
2. **内存布局**

   从 string 和 slice 的运行时表达可以看出，除了 SilceHeader 多了一个 int 类型的 Cap 字段，Date 和 Len 字段是一致的。所以，它们的内存布局是可对齐的，这说明我们就可以直接通过 unsafe.Pointer 进行转换。

`[]byte` 转 `string` 图解



`string` 转 `[]byte` 图解



Q&A

Q1. 为啥强转换性能会比标准转换好?

对于标准转换，无论是从 `[]byte` 转 `string` 还是 `string` 转 `[]byte` 都会涉及底层数组的拷贝。而强转换是直接替换指针的指向，从而使得 `string` 和 `[]byte` 指向同一个底层数组。这样，当然后者的性能会更好。

Q2. 为啥在上述测试中，当 x 的数据较大时，标准转换方式会有一次分配内存的操作，从而导致其性能更差，而强转换方式却不受影响?

标准转换时，当数据长度大于 32 个字节时，需要通过 mallocgc 申请新的内存，之后再进行数据拷贝工作。而强转换只是更改指针指向。所以，当转换数据较大时，两者性能差距会愈加明显。

Q3. 既然强转换方式性能这么好，为啥 go 语言提供给我们使用的是标准转换方式?

首先，我们需要知道 Go 是一门类型安全的语言，而安全的代价就是性能的妥协。但是，性能的对比是相对的，这点性能的妥协对于现在的机器而言微乎其微。另外强转换的方式，会给我们的程序带来极大的安全隐患。

如下示例

```
a := "hello"
b := String2Bytes(a)
b[0] = 'H'
```

a 是 string 类型，前面我们讲到它的值是不可修改的。通过强转换将 a 的底层数组赋给 b，而 b 是一个 `[]byte` 类型，它的值是可以修改的，所以这时对底层数组的值进行修改，将会造成严重的错误（通过 defer + recover 也不能捕获）。

```
unexpected fault address 0x10b6139
fatal error: fault
[signal SIGBUS: bus error code=0x2 addr=0x10b6139 pc=0x1088f2c]
```

Q4. 为啥 string 要设计为不可修改的?

我认为有必要思考一下该问题。string 不可修改，意味它是只读属性，这样的好处就是：在并发场景下，我们可以在不加锁的控制下，多次使用同一字符串，在保证高效共享的情况下而不用担心安全问题。

取舍场景

1. 在你不确定安全隐患的条件下，尽量采用标准方式进行数据转换。
2. 当程序对运行性能有高要求，同时满足对数据仅仅只有读操作的条件，且存在频繁转换（例如消息转发场景），可以使用强转换。

go 后端 源码分析