Go错误集锦 | 字符串底层原理及常见错误

渔夫子 Go学堂 2021-11-20 08:12

string是Go语言的基础类型，在实际项目中针对字符串的各种操作使用频率也较高。本文就介绍一下在使用string时容易犯的一些错误以及如何避免。

01 字符串的一些基本概念

首先我们看下字符串的基本的数据结构：

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

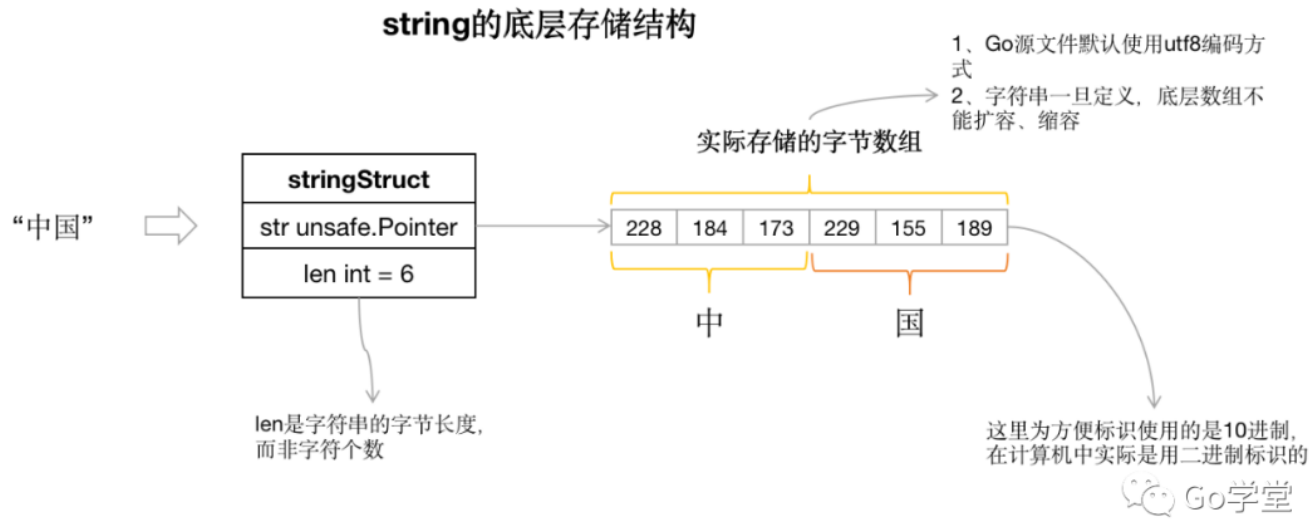
由字符串的数据结构可知，字符串只包含两个成员：

- stringStruct.str：一个指向底层数据的指针
- stringStruct.len：字符串的字节长度，非字符个数。

假设，我们定义了一个字符串 “中国”， 如下：

```
a := "中国"
```

因为Go语言对源代码默认使用utf-8编码方式，utf-8对”中“使用3个字节，对应的编码是（我们这里每个字节编码用10进制表示）：228 184 173。同样，“国”的utf-8编码是：229 155 189。如下存储示意图：

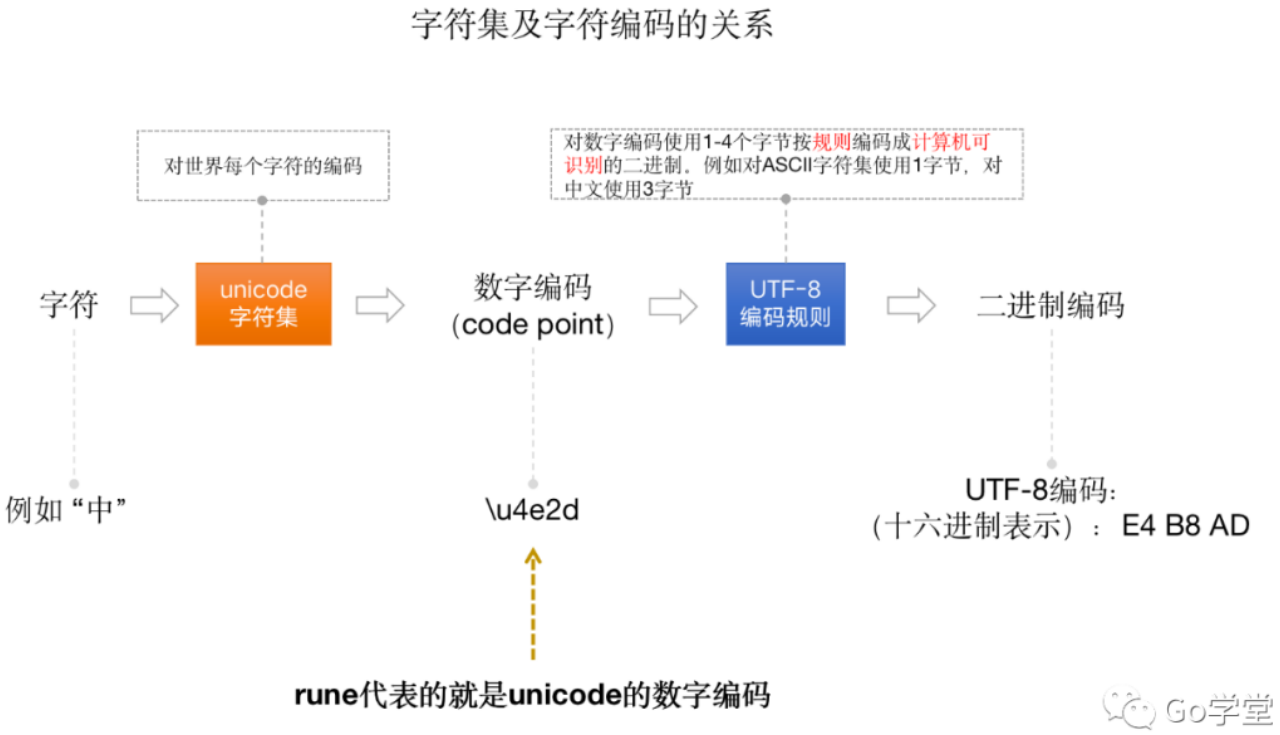


02 rune是什么

要想理解rune，就会涉及到unicode字符集和字符编码的概念以及二者之间的关系。

unicode字符集是对世界上多种语言字符的通用编码，也叫万国码。在unicode字符集中，每一个字符都有一个对应的编号，我们称这个编号为code point，而Go中的**rune类型就代表一个字符的code point，即对应一个字符。**

字符集只是将每个字符给了一个唯一的编码而已。而要想在计算机中进行存储，则必须要**通过特定的编码转换成对应的二进制**才行。所以就有了像ASCII、UTF-8、UTF-16等这样的编码方式。而在Go中默认是使用UTF-8字符编码进行编码的。所以unicode字符集合和字符编码之间的关系如下图所示：



我们知道，UTF-8字符编码是一种变长字节的编码方式，用1到4个字节对字符进行编码，即最多4个字节，按位表示就是32位。所以，在Go的源码中，我们会看到对rune的定义是int32的别名：

```
// rune is an alias for int32 and is equivalent to int32 in all ways. It is
// used, by convention, to distinguish character values from integer values.
type rune = int32
```

好，有了以上基础知识，我们来看看在使用string过程中有哪些需要注意的地方。

03 strings.TrimRight和strings.TrimSuffix的区别

3.1 strings.TrimRight函数

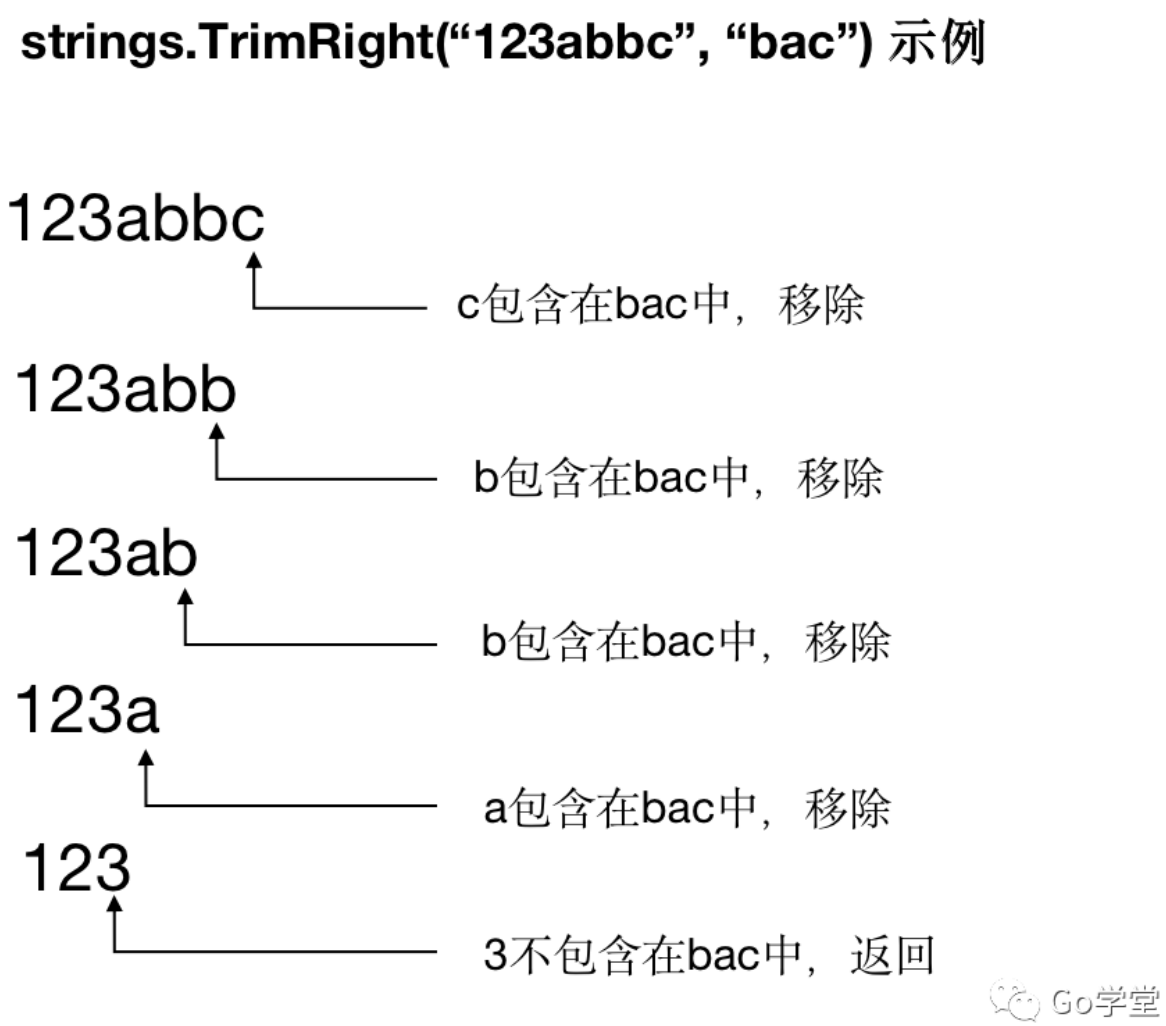
该函数的定义如下：

```
func TrimRight(s, cutset string) string
```

该函数的功能是：从s字符串的末尾依次查找每一个字符，如果该字符包含在cutset中，则被移除，直到遇到第一个不在cutset中的字符。例如：

```
fmt.Println(strings.TrimRight("123abbc", "bac"))
```

执行示例代码，会将字符串末尾的abbc都去除掉，打印出"123"。执行逻辑如下：



3.2 strings.TrimSuffix函数

该函数是将字符串指定的后缀字符串移除。定义如下：

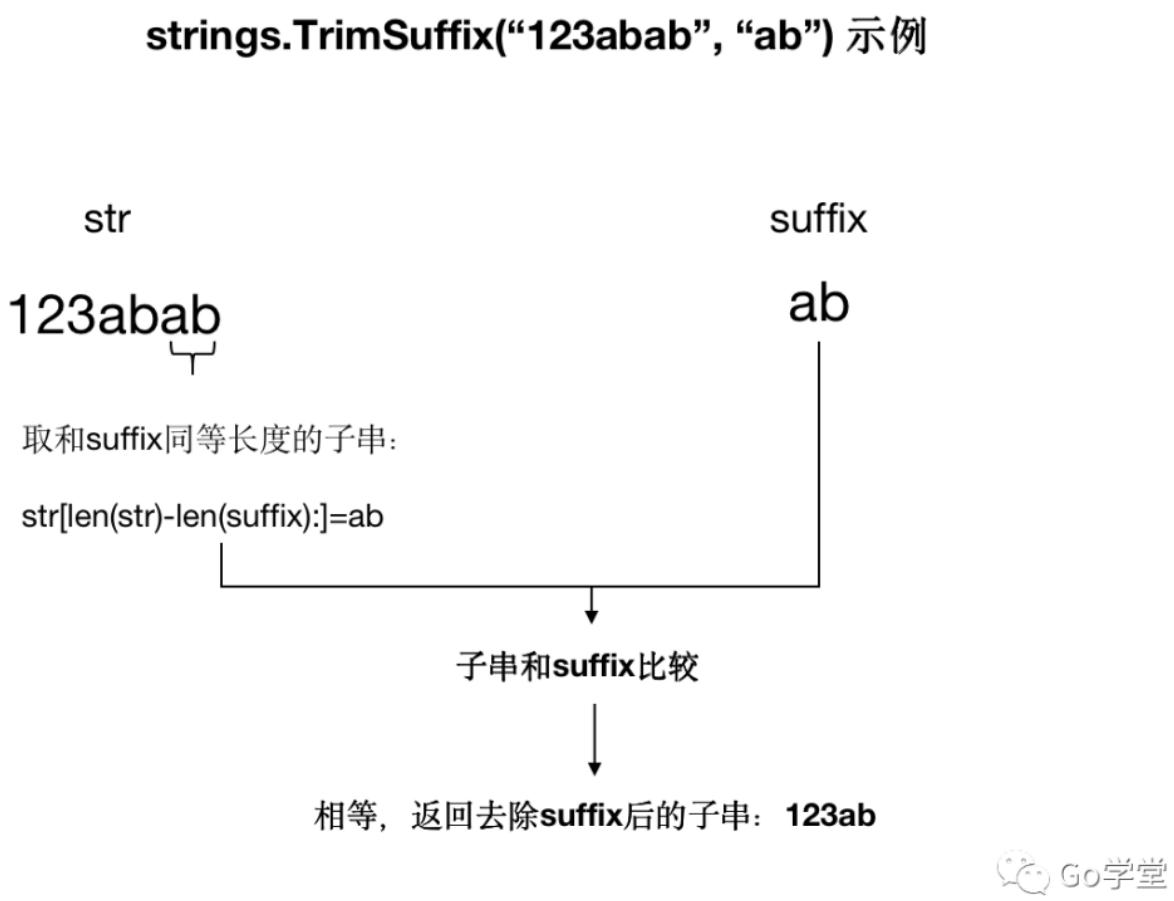
```
func TrimSuffix(s, suffix string) string
```

此函数的实现原理是，从字符串s中截取末尾的长度和suffix字符串长度相等的子字符串，然后和suffix字符串进行比较，如果相等，则将s字符串末尾的子字符串移除，如果不等，则返回原来的s字符串，该函数只截取一次。

我们通过如下示例来了解下其执行逻辑：

```
fmt.Println(strings.TrimSuffix("123abab", "ab"))
```

我们注意到，该字符串末尾有两个ab，但最终只有末尾的一个ab被去除掉，保留”123ab"。执行逻辑如下图所示：



以上的原理同样适用于strings.TrimLeft和strings.Prefix的字符串操作函数。而strings.Trim函数则同时包含了strings.TrimLeft和strings.TrimRight的功能。

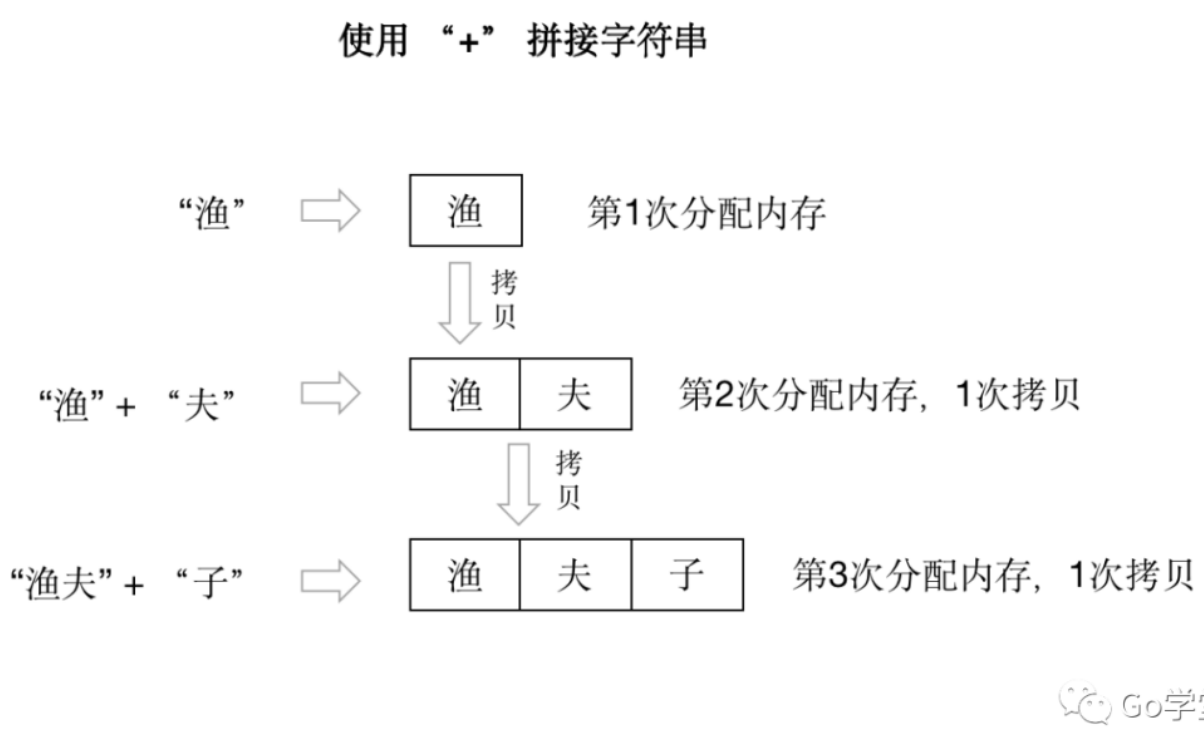
04 字符串拼接性能问题

拼接字符串是在项目中经常使用的一个场景。然而，拼接字符串时的性能问题会常常被忽略。性能问题其本质上就是要注意在拼接字符串时**是否会频繁的产生内存分配以及数据拷贝的操作。**

我们来看一个性能较低的拼接字符串的例子：

```
func concat(ids []string) string {
    s := ""
    for _, id := range ids {
        s += id
    }
    return s
}
```

这段代码执行逻辑上不会有任何问题，但是在进行 s += id进行拼接时，由于字符串是不可变的，所以每次都会分配新的内存空间，并将两个字符串的内容拷贝到新的空间去，然后再让s指向新的空间字符串。由于分配的内存次数多，当然就会对性能造成影响。如下图所示：



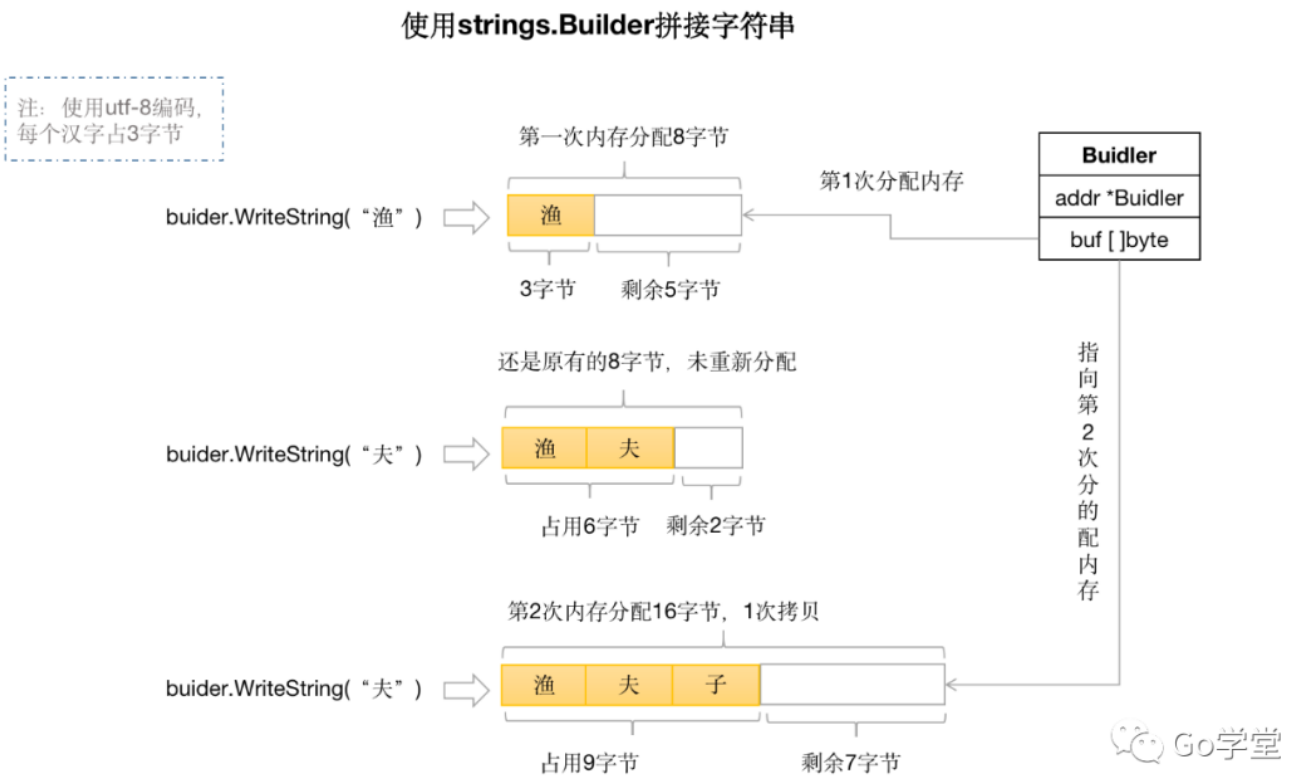
那该如何提高拼接的性能呢？可以通过strings.Builder进行改进。strings.Builder本质上**是分配了一个字节切片**，然后通过**append**的操作，将字符串的字节依次加入到该字节切片中。因为切片预分配空间的特性，可参考切片扩容，以有效的减少内存分配的次数，以提高性能。

```go
func concat(ids []string) string {
    sb := strings.Builder{}
    for _, id := range ids {
        _, _ = sb.WriteString(id)
    }
    return sb.String()
}
```

我们看下strings.Builder的数据结构：

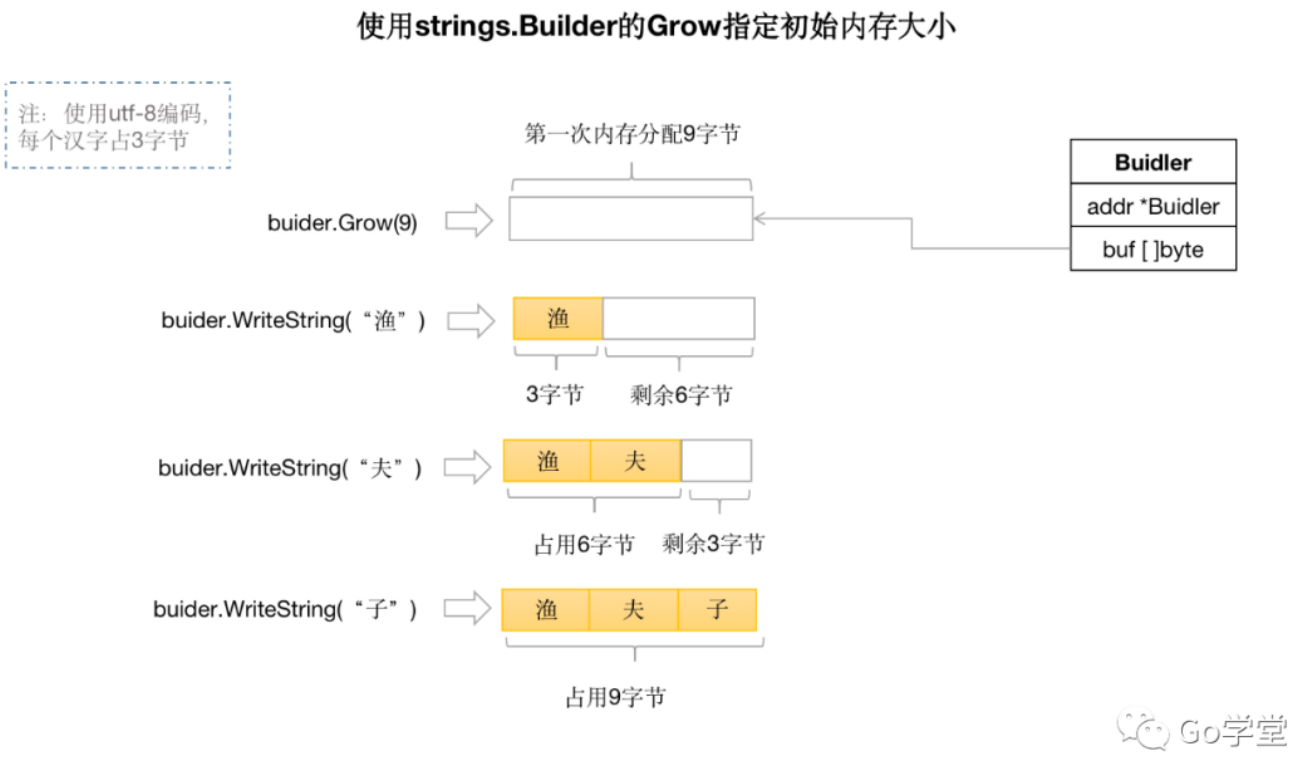
```go
type Builder struct {
    addr *Builder // of receiver, to detect copies by value
    buf  []byte
}
```

由此可见，Builder的结构体中有一个buf []byte，当执行sb.WriteString(id)方法时，实际上是调用了append的方法，将字符串的每个字节都存储到了字节切片buf中。如下图所示：



上图中，第一次分配的内存空间是8个字节，这跟Go的内存管理有关系，网上有很多相关文章，这里不再详细讨论。

如果我们能提前知道要拼接的字符串的长度，我们还可以提前使用**Builder的Grow方法来预分配内存**，这样在整个字符串拼接过程中只需要分配一次内存就好了，极大的提高了字符串拼接的性能。如下图所示及代码：



示例代码：

```
func concat(ids []string) string {
    total := 0
    for i := 0; i < len(ids); i++ {
        total += len(ids[i])
    }
    sb := strings.Builder{}
    sb.Grow(total)
    for _, id := range ids {
        _, _ = sb.WriteString(id)
    }
    return sb.String()
```

```
}
```

strings.Builder的使用场景一般是在**循环中对字符串进行拼接**，如果只是拼接两个或少数几个字符串的话，推荐使用 "+"操作符，例如: s := s1 + s2 + s3，该操作并非每个 + 操作符都计算一次长度，而是会首先计算三个字符串的总长度，然后分配对应的内存，再将三个字符串都拷贝到新申请的内存中去。

05 无用字符串的转换

我们在实际项目中往往会遇到这种场景：是选择字节切片还是字符串的场景。而大多数程序员会倾向于选择字符串。但是，很多IO的操作实际上是使用字节切片的。其实，bytes包中也有很多和strings包中相同操作的函数。

我们看这样一个例子：实现一个getBytes函数，该函数接收一个io.Reader参数作为读取的数据源，然后调用sanitize函数，该函数的作用是去除字符串内容两端的空白字符。我们看下第一个实现：

```
func getBytes(reader io.Reader) ([]byte, error) {
  b, err := io.ReadAll(reader)
  if err != nil {
    return nil, err
  }
   // Call sanitize
   return []byte(sanitize(string(b))), nil
}
```

函数sanitize接收一个字符串类型的参数的实现：

```
func sanitize(s string) string {
  return strings.TrimSpace(s)
}
```

这其实是将**字节切片先转换成了字符串，然后又将字符串转换成字节切片**返回了。其实，在bytes包中有同样的去除空格的函数**bytes.TrimSpace**，使用该函数就避免了对字节切片到字符串多余的转换。

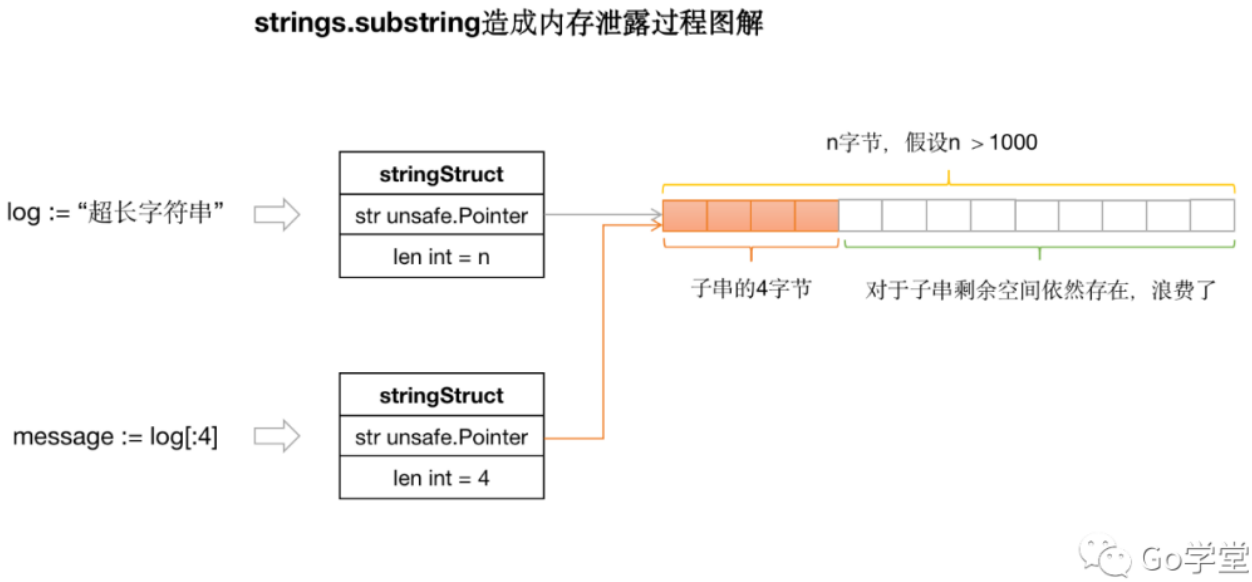
```
func sanitize(s []byte) []byte {
    return bytes.TrimSpace(s)
}
```

06 子字符串操作及内存泄露

字符串的切分也会跟切片的切分一样，可能会造成内存泄露。下面我们看一个例子：有一个handleLog的函数，接收一个string类型的参数log，假设log的前4个字节存储的是log的message类型值，我们需要从log中提取出message类型，并存储到内存中。下面是相关代码：

```
func (s store) handleLog(log string) error {
    if len(log) < 4 {
        return errors.New("log is not correctly formatted")
    }
    message := log[:4]
    s.store(message)
    // Do something
}
```

我们使用log[:4]的方式提取出了message，那么该实现有什么问题吗？我们假设参数log是一个包含成千上万个字符的字符串。当我们使用log[:4]操作时，实际上是返回了一个字节切片，该切片的长度是4，而容量则是log字符串的整体长度。那么实际上我们存储的message不是包含4个字节的空间，而是整个log字符串长度的空间。所以就有可能会造成内存泄露。如下图所示：



那怎么避免呢？使用拷贝。将uuid提取后拷贝到一个字节切片中，这时该字节切片的长度和容量都是4。如下：

```
func (s store) handleLog(log string) error {
  if len(log) < 4 {
    return errors.New("log is not correctly formatted")
  }

  message := string([]byte(log[:4]))
  s.store(message)
  // Do something
}
```

07 小结

字符串是Go语言的一种基本类型，在Go语言中有自己的特性。字符串本质上是一个具有长度和指向底层数组的指针的结构体。在Go中，字符串是以utf-8编码的字节序列将每个字符的unicode编码存储在指针指向的数组中的，因此字符串是不可被修改的。在实际项目中，我们尤其要注意字符串和字节切片之间的转换以及在字符串拼接时的性能问题。