Go 编码建议——性能篇

原创 恋喵大鲤鱼 已于 2022-04-17 22:24:26 修改 499 收藏 7 版权

分类专栏： Go 编码建议 文章标签： golang Go 编码建议

文章目录

代码的稳健、可读和高效是我们每一个 coder 的共同追求。本文将结合 Go 语言特性，为书写效率更高的代码，从常用数据结构、内存管理和并发，三个方面给出相关建议。话不多说，让我们一起学习 Go 高性能编程的技法吧。

常用数据结构

1.反射虽好，切莫贪杯

标准库 reflect 为 Go 语言提供了运行时动态获取对象的类型和值以及动态创建对象的能力。反射可以帮助抽象和简化代码，提高开发效率。

Go 语言标准库以及很多开源软件中都使用了 Go 语言的反射能力，例如用于序列化和反序列化的 json、ORM 框架 gorm、xorm 等。

1.1 优先使用 strconv 而不是 fmt

基本数据类型与 字符串 之间的转换，优先使用 strconv 而不是 fmt，因为前者性能更佳。

```
// Bad
for i := 0; i < b.N; i++ {
    s := fmt.Sprint(rand.Int())
}

```

```
BenchmarkFmtSprint-4    143 ns/op    2 allocs/op

// Good
for i := 0; i < b.N; i++ {
    s := strconv.Itoa(rand.Int())
}

BenchmarkStrconv-4    64.2 ns/op    1 allocs/op
```

为什么性能上会有两倍多的差距，因为 fmt 实现上利用反射来达到范型的效果，在运行时进行类型的动态判断，所以带来了一定的性能损耗。

1.2 少量的重复不比反射差

有时，我们需要一些工具函数。比如从 uint64 切片过滤掉指定的元素。

利用反射，我们可以实现一个类型泛化支持扩展的切片过滤函数。

```
// DeleteSliceElms 从切片中过滤指定元素。注意：不修改原切片。
func DeleteSliceElms(i interface{}, elms ...interface{}) interface{} {
    // 构建 map set。
    m := make(map[interface{}]struct{}, len(elms))
    for _, v := range elms {
        m[v] = struct{}{}
    }
    // 创建新切片，过滤掉指定元素。
    v := reflect.ValueOf(i)
    t := reflect.MakeSlice(reflect.TypeOf(i), 0, v.Len())
    for i := 0; i < v.Len(); i++ {
        if _, ok := m[v.Index(i).Interface()]; !ok {
            t = reflect.Append(t, v.Index(i))
        }
    }
    return t.Interface()
}
```

很多时候，我们可能只需要操作一个类型的切片，利用反射实现的类型泛化扩展的能力压根没用上。退一步说，如果我们真地需要对 uint64 以外类型的切片进行过滤，拷贝一次代码又何妨呢？可以肯定的是，绝大部份场景，根本不会对所有类型的切片进行过滤，那么反射带来好处我们并没有充分享受，但却要为其带来的性能成本买单。

```
// DeleteU64liceElms 从 []uint64 过滤指定元素。注意：不修改原切片。
func DeleteU64liceElms(i []uint64, elms ...uint64) []uint64 {
    // 构建 map set。
    m := make(map[uint64]struct{}, len(elms))
    for _, v := range elms {
        m[v] = struct{}{}
    }
    // 创建新切片，过滤掉指定元素。
    t := make([]uint64, 0, len(i))
    for _, v := range i {
        if _, ok := m[v]; !ok {
            t = append(t, v)
        }
    }
```

```
    return t
}
```

下面看一下二者的性能对比。

```
func BenchmarkDeleteSliceElms(b *testing.B) {
    slice := []uint64{1, 2, 3, 4, 5, 6, 7, 8, 9}
    elms := []interface{}{uint64(1), uint64(3), uint64(5), uint64(7), uint64(9)}
    for i := 0; i < b.N; i++ {
        _ = DeleteSliceElms(slice, elms...)
    }
}

func BenchmarkDeleteU64liceElms(b *testing.B) {
    slice := []uint64{1, 2, 3, 4, 5, 6, 7, 8, 9}
    elms := []uint64{1, 3, 5, 7, 9}
    for i := 0; i < b.N; i++ {
        _ = DeleteU64liceElms(slice, elms...)
    }
}
```

运行上面的基准测试。

```
go test -bench=. -benchmem main/reflect
goos: darwin
goarch: amd64
pkg: main/reflect
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkDeleteSliceElms-12              1226868               978.2 ns/op           296 B/op         16 allocs/op
BenchmarkDeleteU64liceElms-12            8249469               145.3 ns/op            80 B/op          1 allocs/op
PASS
ok      main/reflect    3.809s
```

可以看到，反射涉及了额外的类型判断和大量的内存分配，导致其对性能的影响非常明显。随着切片元素的递增，每一次判断元素是否在 map 中，因为 map 的 key 是不确定的类型，会发生变量逃逸，触发堆内存的分配。所以，可预见的是当元素数量增加时，性能差异会越来大。

当使用反射时，请问一下自己，我真地需要它吗?

1.3 慎用 binary.Read 和 binary.Write

binary.Read 和 binary.Write 使用反射并且很慢。如果有需要用到这两个函数的地方，我们应该手动实现这两个函数的相关功能，而不是直接去使用它们。

encoding/binary 包实现了数字和字节序列之间的简单转换以及 varints 的编码和解码。varints 是一种使用可变字节表示整数的方法。其中数值本身越小，其所占用的字节数越少。Protocol Buffers 对整数采用的便是这种编码方式。

其中数字与字节序列的转换可以用如下三个函数：

```
// Read 从结构化二进制数据 r 读取到 data。data 必须是指向固定大小值的指针或固定大小值的切片。
func Read(r io.Reader, order ByteOrder, data interface{}) error
// Write 将 data 的二进制表示形式写入 w。data 必须是固定大小的值或固定大小值的切片，或指向此类数据的指针。
func Write(w io.Writer, order ByteOrder, data interface{}) error
```

```
// Size 返回 Wirte 函数将 v 写入到 w 中的字节数。
func Size(v interface{}) int
```

下面以我们熟知的 C 标准库函数 ntohl() 函数为例，看看 Go 利用 binary 包如何实现。

```
// Ntohl 将网络字节序的 uint32 转为主机字节序。
func Ntohl(bys []byte) uint32 {
    r := bytes.NewReader(bys)
    err = binary.Read(buf, binary.BigEndian, &num)
}

// 如将 IP 127.0.0.1 网络字节序解析到 uint32
fmt.Println(Ntohl([]byte{0x7f, 0, 0, 0x1})) // 2130706433 <nil>
```

如果我们针对 uint32 类型手动实现一个 ntohl() 呢?

```
func NtohlNotUseBinary(bys []byte) uint32 {
    return uint32(bys[3]) | uint32(bys[2])<<8 | uint32(bys[1])<<16 | uint32(bys[0])<<24
}

// 如将 IP 127.0.0.1 网络字节序解析到 uint32
fmt.Println(NtohlNotUseBinary([]byte{0x7f, 0, 0, 0x1})) // 2130706433
```

该函数也是参考了 encoding/binary 包针对大端字节序将字节序列转为 uint32 类型时的实现。

下面看下剥去反射前后二者的性能差异。

```
func BenchmarkNtohl(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _, _ = Ntohl([]byte{0x7f, 0, 0, 0x1})
    }
}

func BenchmarkNtohlNotUseBinary(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = NtohlNotUseBinary([]byte{0x7f, 0, 0, 0x1})
    }
}
```

运行上面的基准测试，结果如下：

```
go test -bench=BenchmarkNtohl.* -benchmem main/reflect
goos: darwin
goarch: amd64
pkg: main/reflect
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkNtohl-12                       13026195               81.96 ns/op           60 B/op          4 allocs/op
BenchmarkNtohlNotUseBinary-12           1000000000              0.2511 ns/op          0 B/op          0 allocs/op
PASS
ok      main/reflect    1.841s
```

可见使用反射实现的 encoding/binary 包的性能相较于针对具体类型实现的版本，性能差异非常大。

2.避免重复的字符串到字节切片的转换

不要反复从固定字符串创建字节 slice，因为重复的切片初始化会带来性能损耗。相反，请执行一次转换并捕获结果。

```
func BenchmarkStringToByte(b *testing.B) {
   for i := 0; i < b.N; i++ {
      by := []byte("Hello world")
      _ = by
   }
}

func BenchmarkStringToByteOnce(b *testing.B) {
   bys := []byte("Hello world")
   for i := 0; i < b.N; i++ {
      by := bys
      _ = by
   }
}
```

看一下性能差异，注意需要禁止编译器优化，不然看不出差别。

```
go test -bench=BenchmarkStringToByte -gcflags="-N" main/perf
goos: windows
goarch: amd64
pkg: main/perf
cpu: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
BenchmarkStringToByte-8         748467979                1.582 ns/op
BenchmarkStringToByteOnce-8     878246492                1.379 ns/op
PASS
ok      main/perf       2.962s
```

3.指定容器容量

尽可能指定容器容量，以便为容器预先分配内存。这将在后续添加元素时减少通过复制来调整容器大小。

3.1 指定 map 容量提示

在尽可能的情况下，在使用 make() 初始化的时候提供容量信息。

```
make(map[T1]T2, hint)
```

向 make() 提供容量提示会在初始化时尝试调整 map 的大小，这将减少在将元素添加到 map 时为 map 重新分配内存。

注意，与 slice 不同。map capacity 提示并不保证完全的抢占式分配，而是用于估计所需的 hashmap bucket 的数量。 因此，在将元素添加到 map 时，甚至在指定 map 容量时，仍可能发生分配。

```
// Bad
m := make(map[string]os.FileInfo)

files, _ := ioutil.ReadDir("./files")
for _, f := range files {
    m[f.Name()] = f
}
```

```
// m 是在没有大小提示的情况下创建的；  在运行时可能会有更多分配。

// Good
files, _ := ioutil.ReadDir("./files")

m := make(map[string]os.FileInfo, len(files))
for _, f := range files {
    m[f.Name()] = f
}
// m 是有大小提示创建的；在运行时可能会有更少的分配。
```

3.2 指定切片容量

在尽可能的情况下，在使用 make() 初始化切片时提供容量信息，特别是在追加切片时。

```
make([]T, length, capacity)
```

与 map 不同，slice capacity 不是一个提示：编译器将为提供给 make() 的 slice 的容量分配足够的内存，这意味着后续的 append() 操作将导致零分配（直到 slice 的长度与容量匹配，在此之后，任何 append 都可能调整大小以容纳其他元素）。

```
const size = 1000000

// Bad
for n := 0; n < b.N; n++ {
    data := make([]int, 0)
    for k := 0; k < size; k++ {
        data = append(data, k)
  }
}

BenchmarkBad-4    219    5202179 ns/op

// Good
for n := 0; n < b.N; n++ {
    data := make([]int, 0, size)
    for k := 0; k < size; k++ {
        data = append(data, k)
  }
}

BenchmarkGood-4   706    1528934 ns/op
```

执行基准测试：

```
go test -bench=^BenchmarkJoinStr -benchmem
BenchmarkJoinStrWithOperator-8    66930670    17.81 ns/op    0 B/op    0 allocs/op
BenchmarkJoinStrWithSprintf-8      7032921    166.0 ns/op    64 B/op   4 allocs/op
```

4.字符串拼接方式的选择

4.1 行内拼接字符串推荐使用 运算符 +

行内拼接字符串为了书写方便快捷，最常用的两个方法是：

- 运算符+
- `fmt.Sprintf()`

行内字符串的拼接，主要追求的是代码的简洁可读。`fmt.Sprintf()` 能够接收不同类型的入参，通过格式化输出完成字符串的拼接，使用非常方便。但因其底层实现使用了反射，性能上会有所损耗。

运算符 + 只能简单地完成字符串之间的拼接，非字符串类型的变量需要单独做类型转换。行内拼接字符串不会产生内存分配，也不涉及类型地动态转换，所以性能上优于 `fmt.Sprintf()`。

从性能出发，兼顾易用可读，如果待拼接的变量不涉及类型转换且数量较少（<=5），行内拼接字符串推荐使用运算符 +，反之使用 `fmt.Sprintf()`。

下面看下二者的性能对比。

```
// Good
func BenchmarkJoinStrWithOperator(b *testing.B) {
    s1, s2, s3 := "foo", "bar", "baz"
    for i := 0; i < b.N; i++ {
        _ = s1 + s2 + s3
    }
}

// Bad
func BenchmarkJoinStrWithSprintf(b *testing.B) {
    s1, s2, s3 := "foo", "bar", "baz"
    for i := 0; i < b.N; i++ {
        _ = fmt.Sprintf("%s%s%s", s1, s2, s3)
    }
}
```

执行基准测试结果如下：

```
go test -bench=^BenchmarkJoinStr -benchmem .
BenchmarkJoinStrWithOperator-8    70638928    17.53 ns/op     0 B/op    0 allocs/op
BenchmarkJoinStrWithSprintf-8      7520017    157.2 ns/op    64 B/op    4 allocs/op
```

4.2 非行内拼接字符串推荐使用 strings.Builder

字符串拼接还有其他的方式，比如 `strings.Join()`、`strings.Builder`、`bytes.Buffer` 和 `byte[]`，这几种不适合行内使用。当待拼接字符串数量较多时可考虑使用。

先看下其性能测试的对比。

```
func BenchmarkJoinStrWithStringsJoin(b *testing.B) {
    s1, s2, s3 := "foo", "bar", "baz"
    for i := 0; i < b.N; i++ {
        _ = strings.Join([]string{s1, s2, s3}, "")
    }
}

func BenchmarkJoinStrWithStringsBuilder(b *testing.B) {
    s1, s2, s3 := "foo", "bar", "baz"
```

```
    for i := 0; i < b.N; i++ {
        var builder strings.Builder
        _, _ = builder.WriteString(s1)
        _, _ = builder.WriteString(s2)
        _, _ = builder.WriteString(s3)
    }
}

func BenchmarkJoinStrWithBytesBuffer(b *testing.B) {
    s1, s2, s3 := "foo", "bar", "baz"
    for i := 0; i < b.N; i++ {
        var buffer bytes.Buffer
        _, _ = buffer.WriteString(s1)
        _, _ = buffer.WriteString(s2)
        _, _ = buffer.WriteString(s3)
    }
}

func BenchmarkJoinStrWithByteSlice(b *testing.B) {
    s1, s2, s3 := "foo", "bar", "baz"
    for i := 0; i < b.N; i++ {
        var bys []byte
        bys= append(bys, s1...)
        bys= append(bys, s2...)
        _ = append(bys, s3...)
    }
}

func BenchmarkJoinStrWithByteSlicePreAlloc(b *testing.B) {
    s1, s2, s3 := "foo", "bar", "baz"
    for i := 0; i < b.N; i++ {
        bys:= make([]byte, 0, 9)
        bys= append(bys, s1...)
        bys= append(bys, s2...)
        _ = append(bys, s3...)
    }
}
```

基准测试结果如下：

```
go test -bench=^BenchmarkJoinStr .
goos: windows
goarch: amd64
pkg: main/perf
cpu: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
BenchmarkJoinStrWithStringsJoin-8               31543916                36.39 ns/op
BenchmarkJoinStrWithStringsBuilder-8            30079785                40.60 ns/op
BenchmarkJoinStrWithBytesBuffer-8               31663521                39.58 ns/op
BenchmarkJoinStrWithByteSlice-8                 30748495                37.34 ns/op
BenchmarkJoinStrWithByteSlicePreAlloc-8         665341896               1.813 ns/op
```

从结果可以看出，`strings.Join()`、`strings.Builder`、`bytes.Buffer`和`byte[]`的性能相近。如果结果字符串的长度是可预知的，使用`byte[]`且预先分配容量的拼接方式性能最佳。

所以如果对性能要求非常严格，或待拼接的字符串数量足够多时，建议使用 `byte[]` 预先分配容量这种方式。

综合易用性和性能，一般推荐使用`strings.Builder`来拼接字符串。

`string.Builder`也提供了预分配内存的方式 Grow：

```
func BenchmarkJoinStrWithStringsBuilderPreAlloc(b *testing.B) {
    s1, s2, s3 := "foo", "bar", "baz"
    for i := 0; i < b.N; i++ {
        var builder strings.Builder
        builder.Grow(9)
        _, _ = builder.WriteString(s1)
        _, _ = builder.WriteString(s2)
        _, _ = builder.WriteString(s3)
    }
}
```

使用了 Grow 优化后的版本的性能测试结果如下。可以看出相较于不预先分配空间的方式，性能提升了很多。

```
BenchmarkJoinStrWithStringsBuilderPreAlloc-8    60079003                20.95 ns/op
```

5.遍历 []struct{} 使用下标而不是 range

Go 中遍历切片或数组有两种方式，一种是通过下标，一种是 range。二者在功能上没有区别，但是在性能上会有区别吗?

5.1 []int

首先看一下遍历基本类型切片时二者的性能差别，以 []int 为例。

```
// genRandomIntSlice 生成指定长度的随机 []int 切片
func genRandomIntSlice(n int) []int {
    rand.Seed(time.Now().UnixNano())
    nums := make([]int, 0, n)
    for i := 0; i < n; i++ {
        nums = append(nums, rand.Int())
    }
    return nums
}

func BenchmarkIndexIntSlice(b *testing.B) {
    nums := genRandomIntSlice(1024)
    for i := 0; i < b.N; i++ {
        var tmp int
        for k := 0; k < len(nums); k++ {
            tmp = nums[k]
        }
        _ = tmp
    }
}

func BenchmarkRangeIntSlice(b *testing.B) {
    nums := genRandomIntSlice(1024)
    for i := 0; i < b.N; i++ {
        var tmp int
```

```
        for _, num := range nums {
            tmp = num
        }
        _ = tmp
    }
}
```

运行测试结果如下：

```
go test -bench=IntSlice$ .
goos: windows
goarch: amd64
pkg: main/perf
cpu: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
BenchmarkIndexIntSlice-8         5043324               236.2 ns/op
BenchmarkRangeIntSlice-8         5076255               239.1 ns/op
```

`genRandomIntSlice()` 函数用于生成指定长度元素类型为 int 的切片。从最终的结果可以看到，遍历 []int 类型的切片，下标与 range 遍历性能几乎没有区别。

5.2 []struct{}

那么对于稍微复杂一点的 []struct 类型呢?

```
type Item struct {
    id  int
    val [1024]byte
}

func BenchmarkIndexStructSlice(b *testing.B) {
    var items [1024]Item
    for i := 0; i < b.N; i++ {
        var tmp int
        for j := 0; j < len(items); j++ {
            tmp = items[j].id
        }
        _ = tmp
    }
}

func BenchmarkRangeIndexStructSlice(b *testing.B) {
    var items [1024]Item
    for i := 0; i < b.N; i++ {
        var tmp int
        for k := range items {
            tmp = items[k].id
        }
        _ = tmp
    }
}

func BenchmarkRangeStructSlice(b *testing.B) {
    var items [1024]Item
    for i := 0; i < b.N; i++ {
```

```
        var tmp int
        for _, item := range items {
            tmp = item.id
        }
        _ = tmp
    }
}
```

运行测试结果如下：

```
go test -bench=StructSlice -benchmem -gcflags=-N main/range
goos: darwin
goarch: amd64
pkg: main/range
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkIndexStructSlice-12              556436              2165 ns/op               1 B/op          0 allocs/op
BenchmarkRangeIndexStructSlice-12         535705              2124 ns/op               1 B/op          0 allocs/op
BenchmarkRangeStructSlice-12               38799             30914 ns/op              27 B/op          0 allocs/op
PASS
ok      main/range      5.097s
```

可以看出，两种通过 index 遍历 []struct 性能没有差别，但是 range 遍历 []struct 中元素时，性能非常差。

range 只遍历 []struct 下标时，性能比 range 遍历 []struct 值好很多。从这里我们应该能够知道二者性能差别之大的原因。

Item 是一个结构体类型，Item 由两个字段构成，一个类型是 int，一个是类型是 [1024]byte，如果每次遍历 []Item，都会进行一次值拷贝，所以带来了性能损耗。

此外，因为 range 时获取的是值拷贝的副本，对副本的修改，是不会影响到原切片。

需要注意的时，上面运行基准测试时，使用编译选项 -gcflags=-N 禁用了编译器对切片遍历的优化，如果没有该选项，那么上面三种遍历方式没有性能差别。

为什么编译会对上面的测试代码进行优化呢？因为代码实际上只取最后一个切片元素的值，所以前面的循环操作是可以跳过，这样便带来性能的提升。如果是下面的代码，那么编器将无法优化，必须进行每一个元素的拷贝。

```
func BenchmarkRange1(b *testing.B) {
    items := make([]Item, 1024)
    tmps := make([]int, 1024)
    for i := 0; i < b.N; i++ {
        for j := range items {
            tmps[j] = items[j].id
        }
    }
}

func BenchmarkRange2(b *testing.B) {
    items := make([]Item, 1024)
    tmps := make([]int, 1024)
    for i := 0; i < b.N; i++ {
        for j, item := range items {
            tmps[j] = item.id
        }
    }
```

```
}
```

无需去除编译器优化，基准测试结果如下：

```
go test -bench=BenchmarkRange -benchmem  main/range
goos: darwin
goarch: amd64
pkg: main/range
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkRange1-12       2290372               534.8 ns/op             0 B/op          0 allocs/op
BenchmarkRange2-12         46161             27169 ns/op              22 B/op          0 allocs/op
PASS
ok      main/range      3.378s
```

5.3 []*struct

那如果切片中是指向结构体的指针，而不是结构体呢?

```
// genItems 生成指定长度 []*Item 切片
func genItems(n int) []*Item {
    items := make([]*Item, 0, n)
    for i := 0; i < n; i++ {
        items = append(items, &Item{id: i})
    }
    return items
}

func BenchmarkIndexPointer(b *testing.B) {
    items := genItems(1024)
    for i := 0; i < b.N; i++ {
        var tmp int
        for k := 0; k < len(items); k++ {
            tmp = items[k].id
        }
        _ = tmp
    }
}

func BenchmarkRangePointer(b *testing.B) {
    items := genItems(1024)
    for i := 0; i < b.N; i++ {
        var tmp int
        for _, item := range items {
            tmp = item.id
        }
        _ = tmp
    }
}
```

执行性能测试结果：

```
go test -bench=Pointer$ main/perf
goos: windows
goarch: amd64
```

```
pkg: main/perf
cpu: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
BenchmarkIndexPointer-8           773634              1521 ns/op
BenchmarkRangePointer-8           752077              1514 ns/op
```

切片元素从结构体 Item 替换为指针 *Item 后，for 和 range 的性能几乎是一样的。而且使用指针还有另一个好处，可以直接修改指针对应的结构体的值。

5.4 小结

range 在迭代过程中返回的是元素的拷贝，index 则不存在拷贝。

如果 range 迭代的元素较小，那么 index 和 range 的性能几乎一样，如基本类型的切片 []int。但如果迭代的元素较大，如一个包含很多属性的 struct 结构体，那么 index 的性能将显著地高于 range，有时候甚至会有上千倍的性能差异。对于这种场景，建议使用 index。如果使用 range，建议只迭代下标，通过下标访问元素，这种使用方式和 index 就没有区别了。如果想使用 range 同时迭代下标和值，则需要将切片/数组的元素改为指针，才能不影响性能。

内存管理

1.使用空结构体节省内存

1.1 不占内存空间

在 Go 中，我们可以使用 unsafe.Sizeof 计算出一个数据类型实例需要占用的字节数。

```go
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    fmt.Println(unsafe.Sizeof(struct{}{}))
}
```

运行上面的例子将会输出：

```
go run main.go
0
```

可以看到，Go 中空结构体 struct{} 是不占用内存空间，不像 C/C++ 中空结构体仍占用 1 字节。

1.2 用法

因为空结构体不占据内存空间，因此被广泛作为各种场景下的占位符使用。一是节省资源，二是空结构体本身就具备很强的语义，即这里不需要任何值，仅作为占位符，达到的代码即注释的效果。

1.2.1 实现集合（Set）

Go 语言标准库没有提供 Set 的实现，通常使用 map 来代替。事实上，对于集合来说，只需要 map 的键，而不需要值。即使是将值设置为 bool 类型，也会多占据 1 个字节，那假设 map 中有一百万条数据，就会浪费 1MB 的空间。

因此呢，将 map 作为集合（Set）使用时，可以将值类型定义为空结构体，仅作为占位符使用即可。

```
type Set map[string]struct{}

func (s Set) Has(key string) bool {
    _, ok := s[key]
    return ok
}

func (s Set) Add(key string) {
    s[key] = struct{}{}
}

func (s Set) Delete(key string) {
    delete(s, key)
}

func main() {
    s := make(Set)
    s.Add("foo")
    s.Add("bar")
    fmt.Println(s.Has("foo"))
    fmt.Println(s.Has("bar"))
}
```

如果想使用 Set 的完整功能，如初始化（通过切片构建一个 Set）、Add、Del、Clear、Contains 等操作，可以使用开源库 golang-set。

1.2.2 不发送数据的信道

```
func worker(ch chan struct{}) {
    <-ch
    fmt.Println("do something")
}

func main() {
    ch := make(chan struct{})
    go worker(ch)
    ch <- struct{}{}
    close(ch)
}
```

有时候使用 channel 不需要发送任何的数据，只用来通知子协程（goroutine）执行任务，或只用来控制协程的并发。这种情况下，使用空结构体作为占位符就非常合适了。

1.2.3 仅包含方法的结构体

```
type Door struct{}

func (d Door) Open() {
    fmt.Println("Open the door")
}

func (d Door) Close() {
```

```
    fmt.Println("Close the door")
}
```

在部分场景下，结构体只包含方法，不包含任何的字段。例如上面例子中的 Door，在这种情况下，Door 事实上可以用任何的数据结构替代。

```
type Door int
type Door bool
```

无论是 int 还是 bool 都会浪费额外的内存，因此呢，这种情况下，声明为空结构体最合适。

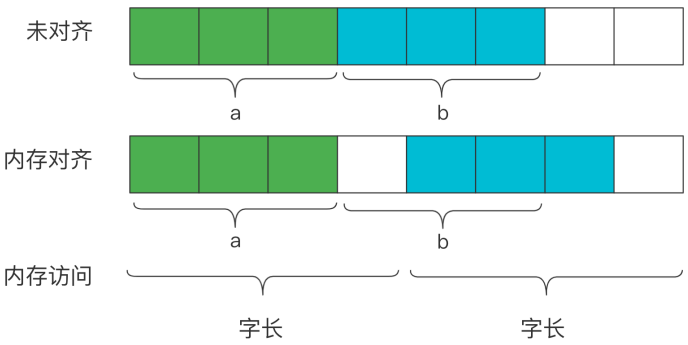
2. struct 布局考虑内存对齐

2.1 为什么需要内存对齐

CPU 访问内存并不是逐个字节访问，而是以字长（word size）为单位访问。如 32 位的 CPU ，字长为 4 字节，那么 CPU 访存的单位是 4 字节。

这么设计的目的，是减少 CPU 访问内存的次数，加大 CPU 访问内存的吞吐量。比如同样读取 8 个字节的数据，一次读取 4 个字节那么只需要读取 2 次。

CPU 始终以字长访问内存，如果不进行内存对齐，可能会增加 CPU 访存次数。



内存对齐前变量 a、b 各占据 3 个字节，CPU 读取 b 变量的值需要两次访存，第一次访存得到前一个字节，第二次得到后两个字节。内存对齐后 a、b 各占 4 个字节，CPU 读取 b 变量的值只需要一次访存。

从这个例子可以看到，内存对齐对实现变量的原子性操作也是有好处的。每次内存访问是原子的，如果变量的大小不超过字长，那么内存对齐后，对该变量的访问就是原子的，这个特性在并发场景下至关重要。

简言之：合理的内存对齐可以提高内存读写的性能，并且便于实现变量操作的原子性。

2.2 Go 内存对齐规则

编译器一般为了减少 CPU 访存指令周期，提高内存的访问效率，会对变量进行内存对齐。Go 作为一门追求高性能的后台编程语言，当然也不例外。

Go Language Specification 中 Size and alignment guarantees 描述了内存对齐的规则。

> 1.For a variable x of any type: unsafe.Alignof(x) is at least 1.
> 2.For a variable x of struct type: unsafe.Alignof(x) is the largest of all the values unsafe.Alignof(x.f) for each field f of x, but at least 1.

> 3.For a variable x of array type: unsafe.Alignof(x) is the same as the alignment of a variable of the array's element type.

- 对于任意类型的变量 x ，unsafe.Alignof(x) 至少为 1。
- 对于结构体类型的变量 x，计算 x 每一个字段 f 的 unsafe.Alignof(x.f)，unsafe.Alignof(x) 等于其中的最大值。
- 对于数组类型的变量 x，unsafe.Alignof(x) 等于构成数组的元素类型的对齐系数。

其中函数 `unsafe.Alignof` 用于获取变量的对齐系数。 对齐系数决定了字段的偏移和变量的大小，两者必须是对齐系数的整数倍。

2.3 合理的 struct 布局

因为内存对齐的存在，合理的 struct 布局可以减少内存占用，提高程序性能。

```
type demo1 struct {
    a int8
    b int16
    c int32
}

type demo2 struct {
    a int8
    c int32
    b int16
}

func main() {
    fmt.Println(unsafe.Sizeof(demo1{})) // 8
    fmt.Println(unsafe.Sizeof(demo2{})) // 12
}
```

可以看到，同样的字段，因排列顺序不同，会导致不一样的结构体大小。

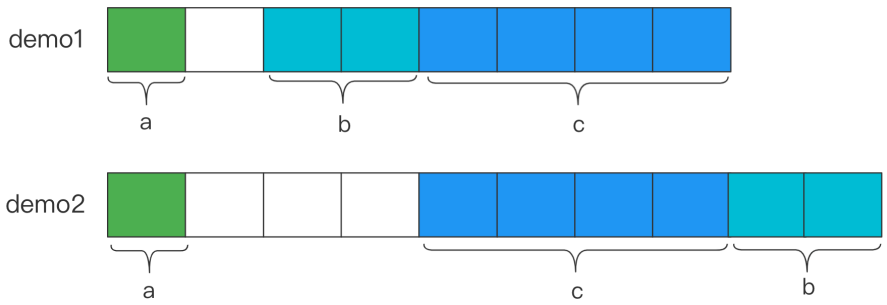
每个字段按照自身的对齐系数来确定在内存中的偏移量，一个字段因偏移而浪费的大小也不同。

接下来逐个分析，首先是 demo1：
a 是第一个字段，默认是已经对齐的，从第 0 个位置开始占据 1 字节。
b 是第二个字段，对齐系数为 2，因此，必须空出 1 个字节，偏移量才是 2 的倍数，从第 2 个位置开始占据 2 字节。
c 是第三个字段，对齐倍数为 4，此时，内存已经是对齐的，从第 4 个位置开始占据 4 字节即可。

因此 demo1 的内存占用为 8 字节。

对于 demo2：
a 是第一个字段，默认是已经对齐的，从第 0 个位置开始占据 1 字节。
c 是第二个字段，对齐倍数为 4，因此，必须空出 3 个字节，偏移量才是 4 的倍数，从第 4 个位置开始占据 4 字节。
b 是第三个字段，对齐倍数为 2，从第 8 个位置开始占据 2 字节。

demo2 的对齐系数由 c 的对齐系数决定，也是 4，因此，demo2 的内存占用为 12 字节。



因此，在对内存特别敏感的结构体的设计上，我们可以通过调整字段的顺序，将字段宽度从小到大由上到下排列，来减少内存的占用。

2.4 空结构与空数组对内存对齐的影响

空结构与空数组在 Go 中比较特殊。没有任何字段的空 struct{} 和没有任何元素的 array 占据的内存空间大小为 0。

因为这一点，空 struct{} 或空 array 作为其他 struct 的字段时，一般不需要内存对齐。但是有一种情况除外：即当 struct{} 或空 array 作为结构体最后一个字段时，需要内存对齐。因为如果有指针指向该字段，返回的地址将在结构体之外，如果此指针一直存活不释放对应的内存，就会有内存泄露的问题（该内存不因结构体释放而释放）。

```go
type demo3 struct {
    a struct{}
    b int32
}
type demo4 struct {
    b int32
    a struct{}
}

func main() {
    fmt.Println(unsafe.Sizeof(demo3{})) // 4
    fmt.Println(unsafe.Sizeof(demo4{})) // 8
}
```

可以看到，demo3{} 的大小为 4 字节，与字段 b 占据空间一致，而 demo4{} 的大小为 8 字节，即额外填充了 4 字节的空间。

3.减少逃逸，将变量限制在栈上

变量逃逸一般发生在如下几种情况：

- 变量较大（栈空间不足）
- 变量大小不确定（如 slice 长度或容量不定）
- 返回地址
- 返回引用（引用变量的底层是指针）
- 返回值类型不确定（不能确定大小）
- 闭包
- 其他

知道变量逃逸的原因后，我们可以有意识地控制变量不发生逃逸，将其控制在栈上，减少堆变量的分配，降低 GC 成本，提高程序性能。

3.1 局部切片尽可能确定长度或容量

如果使用局部的切片时，已知切片的长度或容量，请使用常量或数值字面量来定义。

```
package main

func main() {
    number := 10
    s1 := make([]int, 0, number)
    for i := 0; i < number; i++ {
        s1 = append(s1, i)
    }
    s2 := make([]int, 0, 10)
    for i := 0; i < 10; i++ {
        s2 = append(s2, i)
    }
}
```

我们来看一下编译器编译时对上面两个切片的优化决策。

```
 go build -gcflags="-m -m -l" main.go
# command-line-arguments
./main.go:5:12: make([]int, 0, number) escapes to heap:
./main.go:5:12:   flow: {heap} = &{storage for make([]int, 0, number)}:
./main.go:5:12:     from make([]int, 0, number) (non-constant size) at ./main.go:5:12
./main.go:5:12: make([]int, 0, number) escapes to heap
./main.go:9:12: make([]int, 0, 10) does not escape
```

从输出结果可以看到，使用变量（非常量）来指定切片的容量，会导致切片发生逃逸，影响性能。指定切片的长度时也是一样的，尽可能使用常量或数值字面量。

下面看下二者的性能差异。

```
// sliceEscape 发生逃逸，在堆上申请切片
func sliceEscape() {
    number := 10
    s1 := make([]int, 0, number)
    for i := 0; i < number; i++ {
        s1 = append(s1, i)
    }
}

// sliceNoEscape 不逃逸，限制在栈上
func sliceNoEscape() {
    s1 := make([]int, 0, 10)
    for i := 0; i < 10; i++ {
        s1 = append(s1, i)
    }
}

func BenchmarkSliceEscape(b *testing.B) {
```

```
    for i := 0; i < b.N; i++ {
        sliceEscape()
    }
}

func BenchmarkSliceNoEscape(b *testing.B) {
    for i := 0; i < b.N; i++ {
        sliceNoEscape()
    }
}
```

运行上面的基准测试结果如下：

```
go test -bench=BenchmarkSlice -benchmem main/copy
goos: darwin
goarch: amd64
pkg: main/copy
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkSliceEscape-12         43268738                27.40 ns/op           80 B/op          1 allocs/op
BenchmarkSliceNoEscape-12       186127288                6.454 ns/op           0 B/op          0 allocs/op
PASS
ok      main/copy       4.402s
```

3.2 返回值 VS 返回指针

值传递会拷贝整个对象，而指针传递只会拷贝地址，指向的对象是同一个。传指针可以减少值的拷贝，但是会导致内存分配逃逸到堆中，增加垃圾回收（GC）的负担。在对象频繁创建和删除的场景下，返回指针导致的 GC 开销可能会严重影响性能。

一般情况下，对于需要修改原对象，或占用内存比较大的对象，返回指针。对于只读或占用内存较小的对象，返回值能够获得更好的性能。

下面以一个简单的示例来看下二者的性能差异。

```
type St struct {
    arr [1024]int
}

func retValue() St {
    var st St
    return st
}

func retPtr() *St {
    var st St
    return &st
}

func BenchmarkRetValue(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = retValue()
    }
}

```

```go
func BenchmarkRetPtr(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = retPtr()
    }
}
```

运行基准测试，结果如下：

```
go test -gcflags="-l" -bench=BenchmarkRet -benchmem main/copy
goos: darwin
goarch: amd64
pkg: main/copy
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkRetValue-12             5194722               216.2 ns/op             0 B/op          0 allocs/op
BenchmarkRetPtr-12               1342947               893.6 ns/op          8192 B/op          1 allocs/op
PASS
ok      main/copy       3.865s
```

3.3 小的拷贝好过引用

小的拷贝好过引用，什么意思呢，就是尽量使用栈变量而不是堆变量。下面举一个反常识的例子，来证明小的拷贝比在堆上创建引用变量要好。

我们都知道 Go 里面的 Array 以 pass-by-value 方式传递后，再加上其长度不可扩展，考虑到性能我们一般很少使用它。实际上，凡事无绝对。有时使用数组进行拷贝传递，比使用切片要好。

```go
// copy/copy.go

const capacity = 1024

func arrayFibonacci() [capacity]int {
    var d [capacity]int
    for i := 0; i < len(d); i++ {
        if i <= 1 {
            d[i] = 1
            continue
        }
        d[i] = d[i-1] + d[i-2]
    }
    return d
}

func sliceFibonacci() []int {
    d := make([]int, capacity)
    for i := 0; i < len(d); i++ {
        if i <= 1 {
            d[i] = 1
            continue
        }
        d[i] = d[i-1] + d[i-2]
    }
    return d
}
```

下面看一下性能对比。

```
func BenchmarkArray(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = arrayFibonacci()
    }
}

func BenchmarkSlice(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = sliceFibonacci()
    }
}
```

运行上面的基准测试，将得到如下结果。

```
go test -bench=. -benchmem -gcflags="-l" main/copy
goos: darwin
goarch: amd64
pkg: main/copy
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkArray-12         692400              1708 ns/op               0 B/op          0 allocs/op
BenchmarkSlice-12         464974              2242 ns/op            8192 B/op          1 allocs/op
PASS
ok      main/copy       3.908s
```

从测试结果可以看出，对数组的拷贝性能却比使用切片要好。为什么会这样呢?

sliceFibonacci() 函数中分配的局部变量切片因为要返回到函数外部，所以发生了逃逸，需要在堆上申请内存空间。从测试也过也可以看出，arrayFibonacci() 函数没有内存分配，完全在栈上完成数组的创建。这里说明了对于一些短小的对象，栈上复制的成本远小于在堆上分配和回收操作。

需要注意，运行上面基准测试时，传递了禁止内联的编译选项“-l”，如果发生内联，那么将不会出现变量的逃逸，就不存在堆上分配内存与回收的操作了，二者将看不出性能差异。

编译时可以借助选项 -gcflags=-m 查看编译器对上面两个函数的优化决策。

```
go build  -gcflags=-m copy/copy.go
# command-line-arguments
copy/copy.go:5:6: can inline arrayFibonacci
copy/copy.go:17:6: can inline sliceFibonacci
copy/copy.go:18:11: make([]int, capacity) escapes to heap
```

可以看到，arrayFibonacci() 和 sliceFibonacci() 函数均可内联。sliceFibonacci() 函数中定义的局部变量切片逃逸到了堆。

那么多大的变量才算是小变量呢? 对 Go 编译器而言，超过一定大小的局部变量将逃逸到堆上，不同 Go 版本的大小限制可能不一样。一般是 < 64KB，局部变量将不会逃逸到堆上。

3.4 返回值使用确定的类型

如果变量类型不确定，那么将会逃逸到堆上。所以，函数返回值如果能确定的类型，就不要使用 interface{}。

我们还是以上面斐波那契数列函数为例，看下返回值为确定类型和 interface{} 的性能差别。

```go
const capacity = 1024

func arrayFibonacci() [capacity]int {
    var d [capacity]int
    for i := 0; i < len(d); i++ {
        if i <= 1 {
            d[i] = 1
            continue
        }
        d[i] = d[i-1] + d[i-2]
    }
    return d
}

func arrayFibonacciIfc() interface{} {
    var d [capacity]int
    for i := 0; i < len(d); i++ {
        if i <= 1 {
            d[i] = 1
            continue
        }
        d[i] = d[i-1] + d[i-2]
    }
    return d
}
```

```go
func BenchmarkArray(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = arrayFibonacci()
    }
}

func BenchmarkIfc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = arrayFibonacciIfc()
    }
}
```

运行上面的基准测试结果如下：

```
go test -bench=. -benchmem main/copy
goos: darwin
goarch: amd64
pkg: main/copy
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkArray-12         832418              1427 ns/op               0 B/op          0 allocs/op
BenchmarkIfc-12           380626              2861 ns/op            8192 B/op          1 allocs/op
PASS
ok      main/copy       3.742s
```

可见，函数返回值使用 interface{} 返回时，编译器无法确定返回值的具体类型，导致返回值逃逸到堆上。当发生了堆上内存的申请与回收时，性能会差一点。

4.sync.Pool 复用对象

4.1 简介

sync.Pool 是 sync 包下的一个组件，可以作为保存临时取还对象的一个“池子”。个人觉得它的名字有一定的误导性，因为 Pool 里装的对象可以被无通知地被回收，可能 sync.Cache 是一个更合适的名字。

sync.Pool 是可伸缩的，同时也是并发安全的，其容量仅受限于内存的大小。存放在池中的对象如果不活跃了会被自动清理。

4.2 作用

对于很多需要重复分配、回收内存的地方，sync.Pool 是一个很好的选择。频繁地分配、回收内存会给 GC 带来一定的负担，严重的时候会引起 CPU 的毛刺，而 sync.Pool 可以将暂时不用的对象缓存起来，待下次需要的时候直接使用，不用再次经过内存分配，复用对象的内存，减轻 GC 的压力，提升系统的性能。

一句话总结：用来保存和复用临时对象，减少内存分配，降低 GC 压力。

4.3 如何使用

sync.Pool 的使用方式非常简单，只需要实现 New 函数即可。对象池中没有对象时，将会调用 New 函数创建。

假设我们有一个“学生”结构体，并复用改结构体对象。

```
type Student struct {
    Name   string
    Age    int32
    Remark [1024]byte
}

var studentPool = sync.Pool{
    New: func() interface{} {
        return new(Student)
    },
}
```

然后调用 Pool 的 Get() 和 Put() 方法来获取和放回池子中。

```
stu := studentPool.Get().(*Student)
json.Unmarshal(buf, stu)
studentPool.Put(stu)
```

- Get() 用于从对象池中获取对象，因为返回值是 interface{}，因此需要类型转换。
- Put() 则是在对象使用完毕后，放回到对象池。

4.4 性能差异

我们以 bytes.Buffer 字节缓冲器为例，利用 sync.Pool 复用 bytes.Buffer 对象，避免重复创建与回收内存，来看看对性能的提升效果。

```
var bufferPool = sync.Pool{
    New: func() interface{} {
        return &bytes.Buffer{}
```

```
    },
}

var data = make([]byte, 10000)

func BenchmarkBufferWithPool(b *testing.B) {
    for n := 0; n < b.N; n++ {
        buf := bufferPool.Get().(*bytes.Buffer)
        buf.Write(data)
        buf.Reset()
        bufferPool.Put(buf)
    }
}

func BenchmarkBuffer(b *testing.B) {
    for n := 0; n < b.N; n++ {
        var buf bytes.Buffer
        buf.Write(data)
    }
}
```

测试结果如下：

```
go test -bench=. -benchmem main/pool
goos: darwin
goarch: amd64
pkg: main/pool
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkBufferWithPool-12      11987966                97.12 ns/op            0 B/op          0 allocs/op
BenchmarkBuffer-12               1246887              1020 ns/op           10240 B/op          1 allocs/op
PASS
ok      main/pool       3.510s
```

这个例子创建了一个 bytes.Buffer 对象池，每次只执行 Write 操作，及做一次数据拷贝，耗时几乎可以忽略。而内存分配和回收的耗时占比较多，因此对程序整体的性能影响更大。从测试结果也可以看出，使用了 Pool 复用对象，每次操作不再有内存分配。

4.5 在标准库中的应用

Go 标准库也大量使用了 sync.Pool，例如 fmt 和 encoding/json。以 fmt 包为例，我们看下其是如何使用 sync.Pool 的。

我们可以看一下最常用的标准格式化输出函数 Printf() 函数。

```
// Printf formats according to a format specifier and writes to standard output.
// It returns the number of bytes written and any write error encountered.
func Printf(format string, a ...interface{}) (n int, err error) {
    return Fprintf(os.Stdout, format, a...)
}
```

继续看 Fprintf() 的定义。

```
// Fprintf formats according to a format specifier and writes to w.
// It returns the number of bytes written and any write error encountered.
```

```go
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintf(format, a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

Fprintf() 函数的参数是一个 io.Writer，Printf() 传的是 os.Stdout，相当于直接输出到标准输出。这里的 newPrinter 用的就是 sync.Pool。

```go
// go version go1.17 darwin/amd64

// pp is used to store a printer's state and is reused with sync.Pool to avoid allocations.
type pp struct {
    buf buffer
    ...
}

var ppFree = sync.Pool{
    New: func() interface{} { return new(pp) },
}

// newPrinter allocates a new pp struct or grabs a cached one.
func newPrinter() *pp {
    p := ppFree.Get().(*pp)
    p.panicking = false
    p.erroring = false
    p.wrapErrs = false
    p.fmt.init(&p.buf)
    return p
}

// free saves used pp structs in ppFree; avoids an allocation per invocation.
func (p *pp) free() {
    // Proper usage of a sync.Pool requires each entry to have approximately
    // the same memory cost. To obtain this property when the stored type
    // contains a variably-sized buffer, we add a hard limit on the maximum buffer
    // to place back in the pool.
    //
    // See https://golang.org/issue/23199
    if cap(p.buf) > 64<<10 {
        return
    }

    p.buf = p.buf[:0]
    p.arg = nil
    p.value = reflect.Value{}
    p.wrappedErr = nil
    ppFree.Put(p)
}
```

fmt.Printf() 的调用是非常频繁的，利用 sync.Pool 复用 pp 对象能够极大地提升性能，减少内存占用，同时降低 GC 压力。

并发编程

1.关于锁

1.1 无锁化

加锁是为了避免在并发环境下，同时访问共享资源产生的安全问题。那么，在并发环境下，是否必须加锁？答案是否定的。并非所有的并发都需要加锁。适当地降低锁的粒度，甚至采用无锁化的设计，更能提升并发能力。

无锁化主要有两种实现，无锁数据结构和串行无锁。

1.1.1 无锁数据结构

利用硬件支持的原子操作可以实现无锁的数据结构，原子操作可以在 lock-free 的情况下保证并发安全，并且它的性能也能做到随 CPU 个数的增多而线性扩展。很多语言都提供 CAS 原子操作（如 Go 中的 atomic 包和 C++11 中的 atomic 库），可以用于实现无锁数据结构，如无锁链表。

我们以一个简单的线程安全单向链表的插入操作来看下无锁编程和普通加锁的区别。

```go
package list

import (
    "fmt"
    "sync"
    "sync/atomic"

    "golang.org/x/sync/errgroup"
)

// Node 链表节点
type Node struct {
    Value interface{}
    Next  *Node
}

//
// 有锁单向链表的简单实现
//

// WithLockList 有锁单向链表
type WithLockList struct {
    Head *Node
    mu   sync.Mutex
}

// Push 将元素插入到链表的首部
func (l *WithLockList) Push(v interface{}) {
    l.mu.Lock()
    defer l.mu.Unlock()
    n := &Node{
        Value: v,
        Next:  l.Head,
    }
```

```
    l.Head = n
}

// String 有锁链表的字符串形式输出
func (l WithLockList) String() string {
    s := ""
    cur := l.Head
    for {
        if cur == nil {
            break
        }
        if s != "" {
            s += ","
        }
        s += fmt.Sprintf("%v", cur.Value)
        cur = cur.Next
    }
    return s
}

//
// 无锁单向链表的简单实现
//

// LockFreeList 无锁单向链表
type LockFreeList struct {
    Head atomic.Value
}

// Push 有锁
func (l *LockFreeList) Push(v interface{}) {
    for {
        head := l.Head.Load()
        headNode, _ := head.(*Node)
        n := &Node{
            Value: v,
            Next:  headNode,
        }
        if l.Head.CompareAndSwap(head, n) {
            break
        }
    }
}

// String 有锁链表的字符串形式输出
func (l LockFreeList) String() string {
    s := ""
    cur := l.Head.Load().(*Node)
    for {
        if cur == nil {
            break
        }
        if s != "" {
            s += ","
        }
```

```
        s += fmt.Sprintf("%v", cur.Value)
        cur = cur.Next
    }
    return s
}
```

上面的实现有几点需要注意一下：

（1）无锁单向链表实现时在插入时需要进行 CAS 操作，即调用`CompareAndSwap()`方法进行插入，如果插入失败则进行 for 循环多次尝试，直至成功。

（2）为了方便打印链表内容，实现一个`String()`方法遍历链表，且使用值作为接收者，避免打印对象指针时无法生效。

> 5. If an operand implements method String() string, that method will be invoked to convert the object to a string, which will then be formatted as required by the verb (if any).

我们分别对两种链表做一个并发写入的操作验证一下其功能。

```
package main

import (
    "fmt"

    "main/list"
)

// ConcurWriteWithLockList 并发写入有锁链表
func ConcurWriteWithLockList(l *WithLockList) {
    var g errgroup.Group
    // 10 个协程并发写入链表
    for i := 0; i < 10; i++ {
        i := i
        g.Go(func() error {
            l.Push(i)
            return nil
        })
    }
    _ = g.Wait()
}

// ConcurWriteLockFreeList 并发写入无锁链表
func ConcurWriteLockFreeList(l *LockFreeList) {
    var g errgroup.Group
    // 10 个协程并发写入链表
    for i := 0; i < 10; i++ {
        i := i
        g.Go(func() error {
            l.Push(i)
            return nil
        })
    }
    _ = g.Wait()
}
```

```
36
37 func main() {
38     // 并发写入与遍历打印有锁链表
39     l1 := &list.WithLockList{}
40     list.ConcurWriteWithLockList(l1)
41     fmt.Println(l1)
42
43     // 并发写入与遍历打印无锁链表
44     l2 := &list.LockFreeList{}
45     list.ConcurWriteLockFreeList(l2)
46     fmt.Println(l2)
47 }
```

注意，多次运行上面的`main()`函数的结果可能会不相同，因为并发是无序的。

```
1 8,7,6,9,5,4,3,1,2,0
2 9,8,7,6,5,4,3,2,0,1
```

下面再看一下链表 Push 操作的基准测试，对比一下有锁与无锁的性能差异。

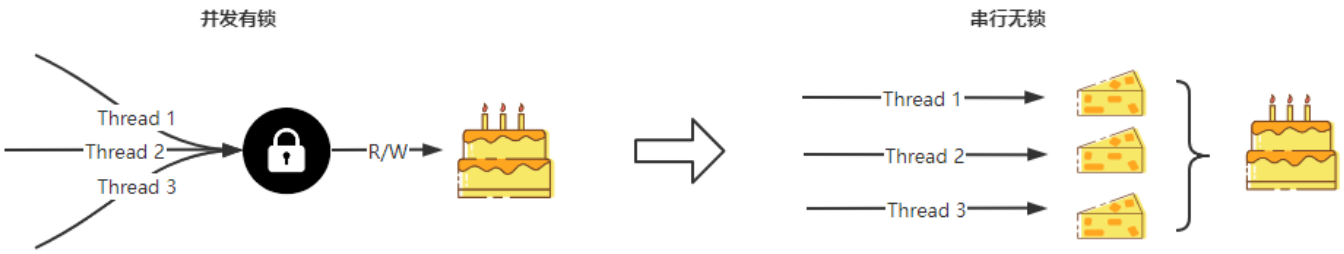
```
1  func BenchmarkWriteWithLockList(b *testing.B) {
2      l := &WithLockList{}
3      for n := 0; n < b.N; n++ {
4          l.Push(n)
5      }
6  }
7  BenchmarkWriteWithLockList-8    14234166                83.58 ns/op
8
9  func BenchmarkWriteLockFreeList(b *testing.B) {
10     l := &LockFreeList{}
11     for n := 0; n < b.N; n++ {
12         l.Push(n)
13     }
14 }
15 BenchmarkWriteLockFreeList-8    15219405                73.15 ns/op
```

可以看出无锁版本比有锁版本性能高一些。

1.1.2 串行无锁

串行无锁是一种思想，就是避免对共享资源的并发访问，改为每个并发操作访问自己独占的资源，达到串行访问资源的效果，来避免使用锁。不同的场景有不同的实现方式。比如网络 I/O 场景下将**单 Reactor 多线程模型**改为**主从 Reactor 多线程模型**，避免对同一个消息队列锁读取。

这里我介绍的是后台微服务开发经常遇到的一种情况。我们经常需要并发拉取多方面的信息，汇聚到一个变量上。那么此时就存在对同一个变量互斥写入的情况。比如批量并发拉取用户信息写入到一个 map。此时我们可以将每个协程拉取的结果写入到一个临时对象，这样便将并发地协程与同一个变量解绑，然后再将其汇聚到一起，这样便可以不用使用锁。即独立处理，然后合并。



为了模拟上面的情况，简单地写个示例程序，对比下性能。

```go
import (
    "sync"

    "golang.org/x/sync/errgroup"
)

// ConcurWriteMapWithLock 有锁并发写入 map
func ConcurWriteMapWithLock() map[int]int {
    m := make(map[int]int)
    var mu sync.Mutex
    var g errgroup.Group
    // 10 个协程并发写入 map
    for i := 0; i < 10; i++ {
        i := i
        g.Go(func() error {
            mu.Lock()
            defer mu.Unlock()
            m[i] = i * i
            return nil
        })
    }
    _ = g.Wait()
    return m
}

// ConcurWriteMapLockFree 无锁并发写入 map
func ConcurWriteMapLockFree() map[int]int {
    m := make(map[int]int)
    // 每个协程独占一 value
    values := make([]int, 10)
    // 10 个协程并发写入 map
    var g errgroup.Group
    for i := 0; i < 10; i++ {
        i := i
        g.Go(func() error {
            values[i] = i * i
            return nil
        })
    }
    _ = g.Wait()
    // 汇聚结果到 map
    for i, v := range values {
```

```
        m[i] = v
    }
    return m
}
```

看下二者的性能差异：

```
func BenchmarkConcurWriteMapWithLock(b *testing.B) {
    for n := 0; n < b.N; n++ {
        _ = ConcurWriteMapWithLock()
    }
}
BenchmarkConcurWriteMapWithLock-8         218673              5089 ns/op

func BenchmarkConcurWriteMapLockFree(b *testing.B) {
    for n := 0; n < b.N; n++ {
        _ = ConcurWriteMapLockFree()
    }
}
BenchmarkConcurWriteMapLockFree-8         316635              4048 ns/op
```

1.2 减少锁竞争

如果加锁无法避免，则可以采用分片的形式，减少对资源加锁的次数，这样也可以提高整体的性能。

比如 Golang 优秀的本地缓存组件 bigcache 、go-cache、freecache 都实现了分片功能，每个分片一把锁，采用分片存储的方式减少加锁的次数从而提高整体性能。

以一个简单的示例，通过对 `map[uint64]struct{}` 分片前后并发写入的对比，来看下减少锁竞争带来的性能提升。

```
var (
    num = 1000000
    m0  = make(map[int]struct{}, num)
    mu0 = sync.RWMutex{}
    m1  = make(map[int]struct{}, num)
    mu1 = sync.RWMutex{}
)

// ConWriteMapNoShard 不分片写入一个 map。
func ConWriteMapNoShard() {
    g := errgroup.Group{}
    for i := 0; i < num; i++ {
        g.Go(func() error {
            mu0.Lock()
            defer mu0.Unlock()
            m0[i] = struct{}{}
            return nil
        })
    }
    _ = g.Wait()
}

// ConWriteMapTwoShard 分片写入两个 map。
func ConWriteMapTwoShard() {
```

```
    g := errgroup.Group{}
    for i := 0; i < num; i++ {
        g.Go(func() error {
            if i&1 == 0 {
                mu0.Lock()
                defer mu0.Unlock()
                m0[i] = struct{}{}
                return nil
            }
            mu1.Lock()
            defer mu1.Unlock()
            m1[i] = struct{}{}
            return nil
        })
    }
    _ = g.Wait()
}
```

看下二者的性能差异：

```
func BenchmarkConWriteMapNoShard(b *testing.B) {
    for i := 0; i < b.N; i++ {
        ConWriteMapNoShard()
    }
}
BenchmarkConWriteMapNoShard-12                3         472063245 ns/op

func BenchmarkConWriteMapTwoShard(b *testing.B) {
    for i := 0; i < b.N; i++ {
        ConWriteMapTwoShard()
    }
}
BenchmarkConWriteMapTwoShard-12               4         310588155 ns/op
```

可以看到，通过对分共享资源的分片处理，减少了锁竞争，能明显地提高程序的并发性能。可以预见的是，随着分片粒度地变小，性能差距会越来越大。当然，分片粒度不是越小越好。因为每一个分片都要配一把锁，那么会带来很多额外的不必要的开销。可以选择一个不太大的值，在性能和花销上寻找一个平衡。

1.3 优先使用共享锁而非互斥锁

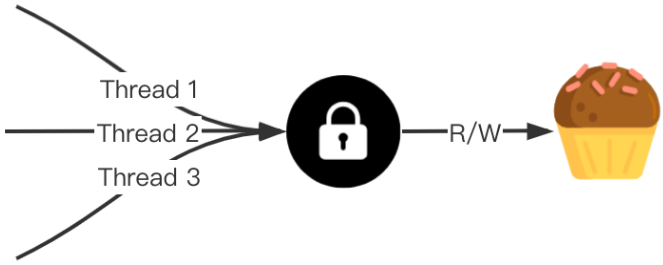
如果并发无法做到无锁化，优先使用共享锁而非互斥锁。

所谓互斥锁，指锁只能被一个 Goroutine 获得。共享锁指可以同时被多个 Goroutine 获得的锁。

Go 标准库 sync 提供了两种锁，互斥锁（sync.Mutex）和读写锁（sync.RWMutex），读写锁便是共享锁的一种具体实现。

1.3.1 sync.Mutex

互斥锁的作用是保证共享资源同一时刻只能被一个 Goroutine 占用，一个 Goroutine 占用了，其他的 Goroutine 则阻塞等待。



sync.Mutex 提供了两个导出方法用来使用锁。

```
Lock()      // 加锁
Unlock()    // 释放锁
```

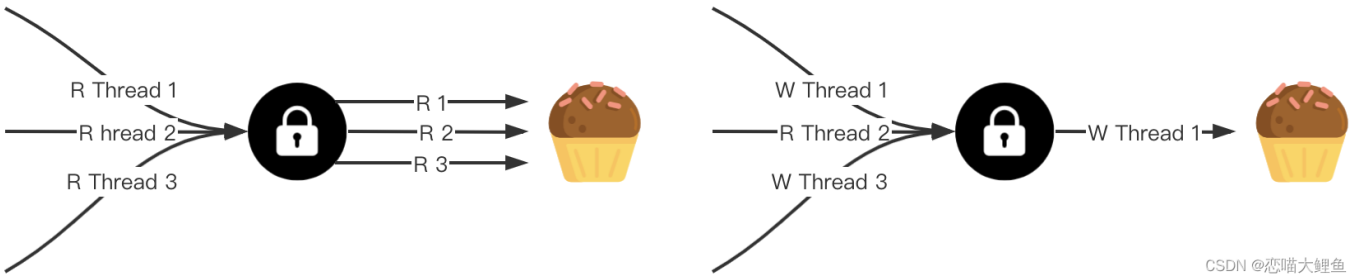
我们可以通过在访问共享资源前前用 Lock 方法对资源进行上锁，在访问共享资源后调用 Unlock 方法来释放锁，也可以用 defer 语句来保证互斥锁一定会被解锁。在一个 Go 协程调用 Lock 方法获得锁后，其他请求锁的协程都会阻塞在 Lock 方法，直到锁被释放。

1.3.2 sync.RWMutex

读写锁是一种共享锁，也称之为多读单写锁 (multiple readers, single writer lock)。在使用锁时，对获取锁的目的操作做了区分，一种是读操作，一种是写操作。因为同一时刻允许多个 Gorouine 获取读锁，所以是一种共享锁。但写锁是互斥的。

一般来说，有如下几种情况：

- 读锁之间不互斥，没有写锁的情况下，读锁是无阻塞的，多个协程可以同时获得读锁。
- 写锁之间是互斥的，存在写锁，其他写锁阻塞。
- 写锁与读锁是互斥的，如果存在读锁，写锁阻塞，如果存在写锁，读锁阻塞。



sync.RWMutex 提供了五个导出方法用来使用锁。

```
Lock()              // 加写锁
Unlock()            // 释放写锁
RLock()             // 加读锁
RUnlock()           // 释放读锁
RLocker() Locker    // 返回读锁，使用 Lock() 和 Unlock() 进行 RLock() 和 RUnlock()
```

读写锁的存在是为了解决读多写少时的性能问题，读场景较多时，读写锁可有效地减少锁阻塞的时间。

1.3.3 性能对比

大部分业务场景是读多写少，所以使用读写锁可有效提高对共享数据的访问效率。最坏的情况，只有写请求，那么读写锁顶多退化成互斥锁。所以优先使用读写锁而非互斥锁，可以提高程序的并发性能。

接下来，我们测试三种情景下，互斥锁和读写锁的性能差异。

- 读多写少(读占 80%)
- 读写一致(各占 50%)
- 读少写多(读占 20%)

首先根据互斥锁和读写锁分别实现对共享 map 的并发读写。

```go
// OpMapWithMutex 使用互斥锁读写 map。
// rpct 为读操作占比。
func OpMapWithMutex(rpct int) {
    m := make(map[int]struct{})
    mu := sync.Mutex{}
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        i := i
        wg.Add(1)
        go func() {
            defer wg.Done()
            mu.Lock()
            defer mu.Unlock()
            // 写操作。
            if i >= rpct {
                m[i] = struct{}{}
                time.Sleep(time.Microsecond)
                return
            }
            // 读操作。
            _ = m[i]
            time.Sleep(time.Microsecond)
        }()
    }
    wg.Wait()
}

// OpMapWithRWMutex 使用读写锁读写 map。
// rpct 为读操作占比。
func OpMapWithRWMutex(rpct int) {
    m := make(map[int]struct{})
    mu := sync.RWMutex{}
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        i := i
        wg.Add(1)
        go func() {
            defer wg.Done()
            // 写操作。
            if i >= rpct {
                mu.Lock()
                defer mu.Unlock()
```

```
                m[i] = struct{}{}
                time.Sleep(time.Microsecond)
                return
            }
            // 读操作。
            mu.RLock()
            defer mu.RUnlock()
            _ = m[i]
            time.Sleep(time.Microsecond)
        }()
    }
    wg.Wait()
}
```

入参 rpct 用来调节读操作的占比，来模拟读写占比不同的场景。rpct 设为 80 表示读多写少(读占 80%)，rpct 设为 50 表示读写一致(各占 50%)，rpct 设为 20 表示读少写多(读占 20%)。

```
func BenchmarkMutexReadMore(b *testing.B) {
    for i := 0; i < b.N; i++ {
        OpMapWithMutex(80)
    }
}

func BenchmarkRWMutexReadMore(b *testing.B) {
    for i := 0; i < b.N; i++ {
        OpMapWithRWMutex(80)
    }
}

func BenchmarkMutexRWEqual(b *testing.B) {
    for i := 0; i < b.N; i++ {
        OpMapWithMutex(50)
    }
}

func BenchmarkRWMutexRWEqual(b *testing.B) {
    for i := 0; i < b.N; i++ {
        OpMapWithRWMutex(50)
    }
}

func BenchmarkMutexWriteMore(b *testing.B) {
    for i := 0; i < b.N; i++ {
        OpMapWithMutex(20)
    }
}

func BenchmarkRWMutexWriteMore(b *testing.B) {
    for i := 0; i < b.N; i++ {
        OpMapWithRWMutex(20)
    }
}
```

执行当前包下的所有基准测试，结果如下：

```
dablelv@DABLELV-MB0 mutex % go test -bench=.
goos: darwin
goarch: amd64
pkg: main/mutex
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkMutexReadMore-12                   2462            485917 ns/op
BenchmarkRWMutexReadMore-12                 8074            145690 ns/op
BenchmarkMutexRWEqual-12                    2406            498673 ns/op
BenchmarkRWMutexRWEqual-12                  4124            303693 ns/op
BenchmarkMutexWriteMore-12                  1906            532350 ns/op
BenchmarkRWMutexWriteMore-12                2462            432386 ns/op
PASS
ok      main/mutex      9.532s
```

可见读多写少的场景，使用读写锁并发性能会更优。可以预见的是如果写占比更低，那么读写锁带的并发效果会更优。

这里需要注意的是，因为每次读写 map 的操作耗时很短，所以每次睡眠一微秒（百万分之一秒）来增加耗时，不然对共享资源的访问耗时，小于锁处理的本身耗时，那么使用读写锁带来的性能优化效果将变得不那么明显，甚至会降低性能。

2.限制协程数量

2.1 协程数过多的问题

2.1.1 程序崩溃

Go 程（goroutine）是由 Go 运行时管理的轻量级线程。通过它我们可以轻松实现并发编程。但是当我们无限开辟协程时，将会遇到致命的问题。

```go
func main() {
    var wg sync.WaitGroup
    for i := 0; i < math.MaxInt32; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()
            fmt.Println(i)
            time.Sleep(time.Second)
        }(i)
    }
    wg.Wait()
}
```

这个例子实现了 math.MaxInt32 个协程的并发，`2^31 - 1` 约为 20 亿个，每个协程内部几乎没有做什么事情。正常的情况下呢，这个程序会乱序输出 0 ~ 2^31-1 个数字。

程序会像预期的那样顺利的运行吗?

```
go run main.go
...
108668
1142025
panic: too many concurrent operations on a single file or socket (max 1048575)

goroutine 1158408 [running]:
internal/poll.(*fdMutex).rwlock(0xc0000ae060, 0x0)
```

```
        /usr/local/go/src/internal/poll/fd_mutex.go:147 +0x11b
internal/poll.(*FD).writeLock(...)
        /usr/local/go/src/internal/poll/fd_mutex.go:239
internal/poll.(*FD).Write(0xc0000ae060, {0xc12cadf690, 0x8, 0x8})
        /usr/local/go/src/internal/poll/fd_unix.go:262 +0x72
os.(*File).write(...)
        /usr/local/go/src/os/file_posix.go:49
os.(*File).Write(0xc0000ac008, {0xc12cadf690, 0x1, 0xc12ea62f50})
        /usr/local/go/src/os/file.go:176 +0x65
fmt.Fprintln({0x10c00e0, 0xc0000ac008}, {0xc12ea62f90, 0x1, 0x1})
        /usr/local/go/src/fmt/print.go:265 +0x75
fmt.Println(...)
        /usr/local/go/src/fmt/print.go:274
main.main.func1(0x0)
        /Users/dablelv/work/code/test/main.go:16 +0x8f
...
```

运行的结果是程序直接崩溃了，关键的报错信息是：

```
panic: too many concurrent operations on a single file or socket (max 1048575)
```

对单个 file/socket 的并发操作个数超过了系统上限，这个报错是 fmt.Printf 函数引起的，fmt.Printf 将格式化后的字符串打印到屏幕，即标准输出。在 Linux 系统中，标准输出也可以视为文件，内核（Kernel）利用文件描述符（File Descriptor）来访问文件，标准输出的文件描述符为 1，错误输出文件描述符为 2，标准输入的文件描述符为 0。

简而言之，系统的资源被耗尽了。

那如果我们将 fmt.Printf 这行代码去掉呢？那程序很可能会因为内存不足而崩溃。这一点更好理解，每个协程至少需要消耗 2KB 的空间，那么假设计算机的内存是 4GB，那么至多允许 4GB/2KB = 1M 个协程同时存在。那如果协程中还存在着其他需要分配内存的操作，那么允许并发执行的协程将会数量级地减少。

2.1.2 协程的代价

前面的例子过于极端，一般情况下程序也不会无限开辟协程，旨在说明协程数量是有限制的，不能无限开辟。

如果我们开辟很多协程，但不会导致程序崩溃，可以吗？如果真要这么做的话，我们应该清楚地知道，协程虽然轻量，但仍有开销。

Go 的开销主要是三个方面：创建（占用内存）、调度（增加调度器负担）和删除（增加 GC 压力）。

- 内存开销

空间上，一个 Go 程占用约 2K 的内存，在源码 src/runtime/runtime2.go里面，我们可以找到 Go 程的结构定义type g struct。

- 调度开销

时间上，协程调度也会有 CPU 开销。我们可以利用runntime.Gosched()让当前协程主动让出 CPU 去执行另外一个协程，下面看一下协程之间切换的耗时。

```
const NUM = 10000

```

```go
func cal() {
    for i := 0; i < NUM; i++ {
        runtime.Gosched()
    }
}

func main() {
    // 只设置一个 Processor
    runtime.GOMAXPROCS(1)
    start := time.Now().UnixNano()
    go cal()
    for i := 0; i < NUM; i++ {
        runtime.Gosched()
    }
    end := time.Now().UnixNano()
    fmt.Printf("total %vns per %vns", end-start, (end-start)/NUM)
}
```

运行输出：

```
total 997200ns per 99ns
```

可见一次协程的切换，耗时大概在 100ns，相对于线程的微秒级耗时切换，性能表现非常优秀，但是仍有开销。

- GC 开销
  创建 Go 程到运行结束，占用的内存资源是需要由 GC 来回收，如果无休止地创建大量 Go 程后，势必会造成对 GC 的压力。

```go
package main

import (
    "fmt"
    "runtime"
    "runtime/debug"
    "sync"
    "time"
)

func createLargeNumGoroutine(num int, wg *sync.WaitGroup) {
    wg.Add(num)
    for i := 0; i < num; i++ {
        go func() {
            defer wg.Done()
        }()
    }
}

func main() {
    // 只设置一个 Processor 保证 Go 程串行执行
    runtime.GOMAXPROCS(1)
    // 关闭GC改为手动执行
    debug.SetGCPercent(-1)

```

```
    var wg sync.WaitGroup
    createLargeNumGoroutine(1000, &wg)
    wg.Wait()
    t := time.Now()
    runtime.GC() // 手动GC
    cost := time.Since(t)
    fmt.Printf("GC cost %v when goroutine num is %v\n", cost, 1000)

    createLargeNumGoroutine(10000, &wg)
    wg.Wait()
    t = time.Now()
    runtime.GC() // 手动GC
    cost = time.Since(t)
    fmt.Printf("GC cost %v when goroutine num is %v\n", cost, 10000)

    createLargeNumGoroutine(100000, &wg)
    wg.Wait()
    t = time.Now()
    runtime.GC() // 手动GC
    cost = time.Since(t)
    fmt.Printf("GC cost %v when goroutine num is %v\n", cost, 100000)
}
```

运行输出：

```
GC cost 0s when goroutine num is 1000
GC cost 2.0027ms when goroutine num is 10000
GC cost 30.9523ms when goroutine num is 100000
```

当创建的 Go 程数量越多，GC 耗时越大。

上面的分析目的是为了尽可能地量化 Goroutine 的开销。虽然官方宣称用 Golang 写并发程序的时候随便起个成千上万的 Goroutine 毫无压力，但当我们起十万、百万甚至千万个 Goroutine 呢？Goroutine 轻量的开销将被放大。

2.2 限制协程数量

系统地资源是有限，协程是有代价的，为了保护程序，提高性能，我们应主动限制并发的协程数量。

可以利用信道 channel 的缓冲区大小来实现。

```
func main() {
    var wg sync.WaitGroup
    ch := make(chan struct{}, 3)
    for i := 0; i < 10; i++ {
        ch <- struct{}{}
        wg.Add(1)
        go func(i int) {
            defer wg.Done()
            log.Println(i)
            time.Sleep(time.Second)
            <-ch
        }(i)
    }
    wg.Wait()
```

```
}
```

上例中创建了缓冲区大小为 3 的 channel，在没有被接收的情况下，至多发送 3 个消息则被阻塞。开启协程前，调用 `ch <- struct{}{}`，若缓存区满，则阻塞。协程任务结束，调用 <-ch 释放缓冲区。

sync.WaitGroup 并不是必须的，例如 Http 服务，每个请求天然是并发的，此时使用 channel 控制并发处理的任务数量，就不需要 sync.WaitGroup。

运行结果如下：

```
2022/03/06 20:37:02 0
2022/03/06 20:37:02 2
2022/03/06 20:37:02 1
2022/03/06 20:37:03 3
2022/03/06 20:37:03 4
2022/03/06 20:37:03 5
2022/03/06 20:37:04 6
2022/03/06 20:37:04 7
2022/03/06 20:37:04 8
2022/03/06 20:37:05 9
```

从日志中可以很容易看到，每秒钟只并发执行了 3 个任务，达到了协程并发控制的目的。

2.3 协程池化

上面的例子只是简单地限制了协程开辟的数量。在此基础之上，基于对象复用的思想，我们可以重复利用已开辟的协程，避免协程的重复创建销毁，达到池化的效果。

协程池化，我们可以自己写一个协程池，但不推荐这么做。因为已经有成熟的开源库可供使用，无需再重复造轮子。目前有很多第三方库实现了协程池，可以很方便地用来控制协程的并发数量，比较受欢迎的有：

- Jeffail/tunny
- panjf2000/ants

下面以 panjf2000/ants 为例，简单介绍其使用。

ants 是一个简单易用的高性能 Goroutine 池，实现了对大规模 Goroutine 的调度管理和复用，允许使用者在开发并发程序的时候限制 Goroutine 数量，复用协程，达到更高效执行任务的效果。

```go
package main

import (
    "fmt"
    "time"

    "github.com/panjf2000/ants"
)

func main() {
    // Use the common pool
    for i := 0; i < 10; i++ {
        i := i
        ants.Submit(func() {
```

```
            fmt.Println(i)
        })
    }
    time.Sleep(time.Second)
}
```

使用 ants，我们简单地使用其默认的协程池，直接将任务提交并发执行。默认协程池的缺省容量 math.MaxInt32。

如果自定义协程池容量大小，可以调用 NewPool 方法来实例化具有给定容量的池，如下所示：

```
// Set 10000 the size of goroutine pool
p, _ := ants.NewPool(10000)
```

2.4 小结

Golang 为并发而生。Goroutine 是由 Go 运行时管理的轻量级线程，通过它我们可以轻松实现并发编程。Go 虽然轻量，但天下没有免费的午餐，无休止地开辟大量 Go 程势必会带来性能影响，甚至程序崩溃。所以，我们应尽可能的控制协程数量，如果有需要，请复用它。

3.使用 sync.Once 避免重复执行

3.1 简介

sync.Once 是 Go 标准库提供的使函数只执行一次的实现，常应用于单例模式，例如初始化配置、保持数据库连接等。作用与 init 函数类似，但有区别。

- init 函数是当所在的 package 首次被加载时执行，若迟迟未被使用，则既浪费了内存，又延长了程序加载时间。
- sync.Once 可以在代码的任意位置初始化和调用，因此可以延迟到使用时再执行，并发场景下是线程安全的。

在多数情况下，sync.Once 被用于控制变量的初始化，这个变量的读写满足如下三个条件：

- 当且仅当第一次访问某个变量时，进行初始化（写）；
- 变量初始化过程中，所有读都被阻塞，直到初始化完成；
- 变量仅初始化一次，初始化完成后驻留在内存里。

3.2 原理

sync.Once 用来保证函数只执行一次。要达到这个效果，需要做到两点：

- 计数器，统计函数执行次数；
- 线程安全，保障在多 Go 程的情况下，函数仍然只执行一次，比如锁。

3.2.1 源码

下面看一下 sync.Once 结构，其有两个变量。使用 done 统计函数执行次数，使用锁 m 实现线程安全。果不其然，和上面的猜想一致。

```
// Once is an object that will perform exactly one action.
//
// A Once must not be copied after first use.
```

```
type Once struct {
    // done indicates whether the action has been performed.
    // It is first in the struct because it is used in the hot path.
    // The hot path is inlined at every call site.
    // Placing done first allows more compact instructions on some architectures (amd64/386),
    // and fewer instructions (to calculate offset) on other architectures.
    done uint32
    m    Mutex
}
```

sync.Once 仅提供了一个导出方法 Do()，参数 f 是只会被执行一次的函数，一般为对象初始化函数。

```
// go version go1.17 darwin/amd64

// Do calls the function f if and only if Do is being called for the
// first time for this instance of Once. In other words, given
//  var once Once
// if once.Do(f) is called multiple times, only the first call will invoke f,
// even if f has a different value in each invocation. A new instance of
// Once is required for each function to execute.
//
// Do is intended for initialization that must be run exactly once. Since f
// is niladic, it may be necessary to use a function literal to capture the
// arguments to a function to be invoked by Do:
//  config.once.Do(func() { config.init(filename) })
//
// Because no call to Do returns until the one call to f returns, if f causes
// Do to be called, it will deadlock.
//
// If f panics, Do considers it to have returned; future calls of Do return
// without calling f.
//
func (o *Once) Do(f func()) {
    // Note: Here is an incorrect implementation of Do:
    //
    //  if atomic.CompareAndSwapUint32(&o.done, 0, 1) {
    //      f()
    //  }
    //
    // Do guarantees that when it returns, f has finished.
    // This implementation would not implement that guarantee:
    // given two simultaneous calls, the winner of the cas would
    // call f, and the second would return immediately, without
    // waiting for the first's call to f to complete.
    // This is why the slow path falls back to a mutex, and why
    // the atomic.StoreUint32 must be delayed until after f returns.

    if atomic.LoadUint32(&o.done) == 0 {
        // Outlined slow-path to allow inlining of the fast-path.
        o.doSlow(f)
    }
}

func (o *Once) doSlow(f func()) {
    o.m.Lock()
```

```
    defer o.m.Unlock()
    if o.done == 0 {
        defer atomic.StoreUint32(&o.done, 1)
        f()
    }
}
```

抛去大段的注释，可以看到 sync.Once 实现非常简洁。Do() 函数中，通过对成员变量 done 的判断，来决定是否执行传入的任务函数。执行任务函数前，通过锁保证任务函数的执行和 done 的修改是一个互斥操作。在执行任务函数前，对 done 做一个二次判断，来保证任务函数只会被执行一次，done 只会被修改一次。

3.2.2 done 为什么是第一个字段

从字段 done 前有一段注释，说明了done 为什么是第一个字段。

done 在热路径中，done 放在第一个字段，能够减少 CPU 指令，也就是说，这样做能够提升性能。

热路径（hot path）是程序非常频繁执行的一系列指令，sync.Once 绝大部分场景都会访问 o.done，在热路径上是比较好理解的。如果 hot path 编译后的机器码指令更少，更直接，必然是能够提升性能的。

为什么放在第一个字段就能够减少指令呢？因为结构体第一个字段的地址和结构体的指针是相同的，如果是第一个字段，直接对结构体的指针解引用即可。如果是其他的字段，除了结构体指针外，还需要计算与第一个值的偏移（calculate offset）。在机器码中，偏移量是随指令传递的附加值，CPU 需要做一次偏移值与指针的加法运算，才能获取要访问的值的地址。因为，访问第一个字段的机器代码更紧凑，速度更快。

> 参考 What does “hot path” mean in the context of sync.Once? - StackOverflow

3.3 性能差异

我们以一个简单示例，来说明使用 sync.Once 保证函数只会被执行一次和多次执行，二者的性能差异。

考虑一个简单的场景，函数 ReadConfig 需要读取环境变量，并转换为对应的配置。环境变量在程序执行前已经确定，执行过程中不会发生改变。ReadConfig 可能会被多个协程并发调用，为了提升性能（减少执行时间和内存占用），使用 sync.Once 是一个比较好的方式。

```
type Config struct {
    GoRoot string
    GoPath string
}

var (
    once   sync.Once
    config *Config
)

func ReadConfigWithOnce() *Config {
    once.Do(func() {
        config = &Config{
            GoRoot: os.Getenv("GOROOT"),
            GoPath: os.Getenv("GOPATH"),
        }
    })
```

```
    return config
}

func ReadConfig() *Config {
    return &Config{
        GoRoot: os.Getenv("GOROOT"),
        GoPath: os.Getenv("GOPATH"),
    }
}
```

我们看下二者的性能差异。

```
func BenchmarkReadConfigWithOnce(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = ReadConfigWithOnce()
    }
}

func BenchmarkReadConfig(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = ReadConfig()
    }
}
```

执行测试结果如下：

```
go test -bench=. main/once
goos: darwin
goarch: amd64
pkg: main/once
cpu: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
BenchmarkReadConfigWithOnce-12          670438965                1.732 ns/op
BenchmarkReadConfig-12                  13339154                87.46 ns/op
PASS
ok      main/once       3.006s
```

sync.Once 中保证了 Config 初始化函数仅执行了一次，避免了多次重复初始化，在并发环境下很有用。

4.使用 sync.Cond 通知协程

4.1 简介

sync.Cond 是基于互斥锁/读写锁实现的条件变量，用来协调想要访问共享资源的那些 Goroutine，当共享资源的状态发生变化的时候，sync.Cond 可以用来通知等待条件发生而阻塞的 Goroutine。

sync.Cond 基于互斥锁/读写锁，它和互斥锁的区别是什么呢?

互斥锁 sync.Mutex 通常用来保护共享的临界资源，条件变量 sync.Cond 用来协调想要访问共享资源的 Goroutine。当共享资源的状态发生变化时，sync.Cond 可以用来通知被阻塞的 Goroutine。

4.2 使用场景

sync.Cond 经常用在多个 Goroutine 等待，一个 Goroutine 通知（事件发生）的场景。如果是一个通知，一个等待，使用互

斥锁或 channel 就能搞定了。

我们想象一个非常简单的场景：

有一个协程在异步地接收数据，剩下的多个协程必须等待这个协程接收完数据，才能读取到正确的数据。在这种情况下，如果单纯使用 chan 或互斥锁，那么只能有一个协程可以等待，并读取到数据，没办法通知其他的协程也读取数据。

这个时候，就需要有个全局的变量来标志第一个协程数据是否接受完毕，剩下的协程，反复检查该变量的值，直到满足要求。或者创建多个 channel，每个协程阻塞在一个 channel 上，由接收数据的协程在数据接收完毕后，逐个通知。总之，需要额外的复杂度来完成这件事。

Go 语言在标准库 sync 中内置一个 sync.Cond 用来解决这类问题。

4.3 原理

sync.Cond 内部维护了一个等待队列，队列中存放的是所有在等待这个 sync.Cond 的 Go 程，即保存了一个通知列表。sync.Cond 可以用来唤醒一个或所有因等待条件变量而阻塞的 Go 程，以此来实现多个 Go 程间的同步。

sync.Cond 的定义如下：

```
// Cond implements a condition variable, a rendezvous point
// for goroutines waiting for or announcing the occurrence
// of an event.
//
// Each Cond has an associated Locker L (often a *Mutex or *RWMutex),
// which must be held when changing the condition and
// when calling the Wait method.
//
// A Cond must not be copied after first use.
type Cond struct {
    noCopy noCopy

    // L is held while observing or changing the condition
    L Locker

    notify  notifyList
    checker copyChecker
}
```

每个 Cond 实例都会关联一个锁 L（互斥锁 *Mutex，或读写锁 *RWMutex），当修改条件或者调用 Wait 方法时，必须加锁。

sync.Cond 的四个成员函数定义如下：

```
// NewCond returns a new Cond with Locker l.
func NewCond(l Locker) *Cond {
    return &Cond{L: l}
}
```

NewCond 创建 Cond 实例时，需要关联一个锁。

```
// Wait atomically unlocks c.L and suspends execution
// of the calling goroutine. After later resuming execution,
// Wait locks c.L before returning. Unlike in other systems,
```

```
// Wait cannot return unless awoken by Broadcast or Signal.
//
// Because c.L is not locked when Wait first resumes, the caller
// typically cannot assume that the condition is true when
// Wait returns. Instead, the caller should Wait in a loop:
//
//    c.L.Lock()
//    for !condition() {
//        c.Wait()
//    }
//    ... make use of condition ...
//    c.L.Unlock()
//
func (c *Cond) Wait() {
    c.checker.check()
    t := runtime_notifyListAdd(&c.notify)
    c.L.Unlock()
    runtime_notifyListWait(&c.notify, t)
    c.L.Lock()
}
```

Wait 用于阻塞调用者，等待通知。调用 Wait 会自动释放锁 c.L，并挂起调用者所在的 goroutine。如果其他协程调用了 Signal 或 Broadcast 唤醒了该协程，那么 Wait 方法在结束阻塞时，会重新给 c.L 加锁，并且继续执行 Wait 后面的代码。

对条件的检查，使用了 for !condition() 而非 if，是因为当前协程被唤醒时，条件不一定符合要求，需要再次 Wait 等待下次被唤醒。为了保险起，使用 for 能够确保条件符合要求后，再执行后续的代码。

```
// Signal wakes one goroutine waiting on c, if there is any.
//
// It is allowed but not required for the caller to hold c.L
// during the call.
func (c *Cond) Signal() {
    c.checker.check()
    runtime_notifyListNotifyOne(&c.notify)
}

// Broadcast wakes all goroutines waiting on c.
//
// It is allowed but not required for the caller to hold c.L
// during the call.
func (c *Cond) Broadcast() {
    c.checker.check()
    runtime_notifyListNotifyAll(&c.notify)
}
```

Signal 只唤醒任意 1 个等待条件变量 c 的 goroutine，无需锁保护。Broadcast 唤醒所有等待条件变量 c 的 goroutine，无需锁保护。

4.4 使用示例

我们实现一个简单的例子，三个协程调用 Wait() 等待，另一个协程调用 Broadcast() 唤醒所有等待的协程。

```
var done = false

```

```go
func read(name string, c *sync.Cond) {
    c.L.Lock()
    for !done {
        c.Wait()
    }
    log.Println(name, "starts reading")
    c.L.Unlock()
}

func write(name string, c *sync.Cond) {
    log.Println(name, "starts writing")
    time.Sleep(time.Second)
    done = true
    log.Println(name, "wakes all")
    c.Broadcast()
}

func main() {
    cond := sync.NewCond(&sync.Mutex{})

    go read("reader1", cond)
    go read("reader2", cond)
    go read("reader3", cond)
    write("writer", cond)

    time.Sleep(time.Second * 3)
}
```

- done 即多个 Goroutine 阻塞等待的条件。
- read() 调用 Wait() 等待通知，直到 done 为 true。
- write() 接收数据，接收完成后，将 done 置为 true，调用 Broadcast() 通知所有等待的协程。
- write() 中的暂停了 1s，一方面是模拟耗时，另一方面是确保前面的 3 个 read 协程都执行到 Wait()，处于等待状态。main 函数最后暂停了 3s，确保所有操作执行完毕。

运行输出：

```
go run main.go
2022/03/07 17:20:09 writer starts writing
2022/03/07 17:20:10 writer wakes all
2022/03/07 17:20:10 reader3 starts reading
2022/03/07 17:20:10 reader1 starts reading
2022/03/07 17:20:10 reader2 starts reading
```

更多关于 sync.Cond 的讨论可参考 How to correctly use sync.Cond? - StackOverflow。

4.5 注意事项

- sync.Cond 不能被复制

sync.Cond 不能被复制的原因，并不是因为其内部嵌套了 Locker。因为 NewCond 时传入的 Mutex/RWMutex 指针，对于 Mutex 指针复制是没有问题的。

主要原因是 sync.Cond 内部是维护着一个 Goroutine 通知队列 notifyList。如果这个队列被复制的话，那么就在并发场景下导致不同 Goroutine 之间操作的 notifyList.wait、notifyList.notify 并不是同一个，这会导致出现有些 Goroutine 会一直阻塞。

- 唤醒顺序

从等待队列中按照顺序唤醒，先进入等待队列，先被唤醒。

- 调用 Wait() 前要加锁

调用 Wait() 函数前，需要先获得条件变量的成员锁，原因是需要互斥地变更条件变量的等待队列。在 Wait() 返回前，会重新上锁。

参考文献

github.com/uber-go/guide
go-proverbs
github/dgryski/go-perfbook
High Performance Go Workshop - Dave Cheney
atomic 的原理与使用场景
极客兔兔.Go 语言高性能编程
深度解密Go 语言之sync.Pool - Stefno - 博客园
Golang内存分配逃逸分析 - Gopherzhang
Go语言的内存逃逸分析 - Golang梦工厂