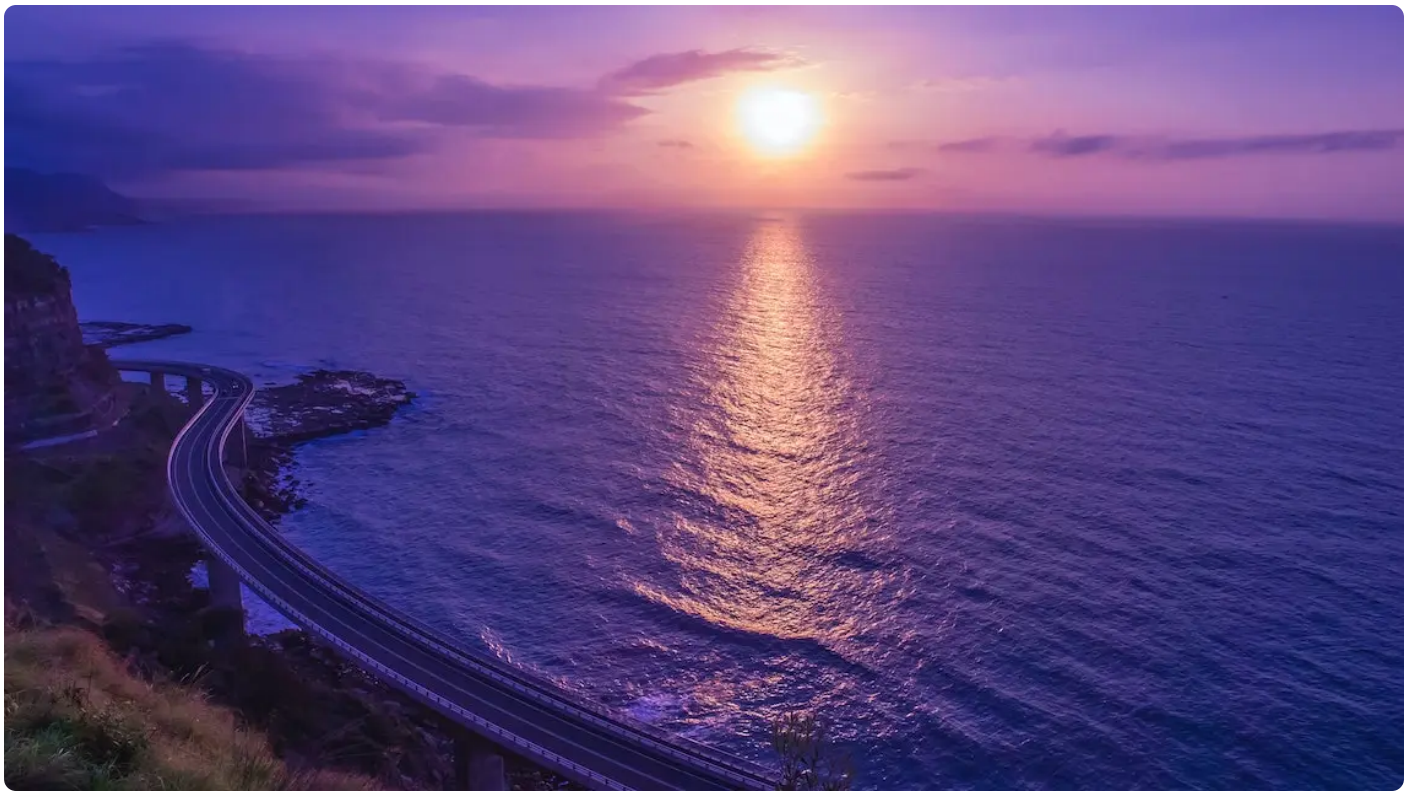
49 | 服务治理：如何进行限流、熔断与认证？

2023-02-02 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >



讲述：郑建勋

时长 12:14　大小 11.18M

你好，我是郑建勋。

在之前我们已经完成了 Master 与 Worker 的核心功能。在大规模微服务集群中，为了保证微服务集群正常运行，还需要添加许多重要的功能，包括限流、熔断、认证与鉴权。这节课，就让我们来看看如何实现这些功能。

限流

限流指的是对给定时间内可能发生的事件的频率进行限制。一旦请求达到规定的上限，此后这段时间内的请求都将被丢弃。

限流对于公共 API 非常重要，它有下面几个优势。

- 提高服务的可用性和可靠性，并有助于防御或缓解一些常见的攻击（DoS 攻击、 DDoS 攻击、暴力破解、撞库攻击、网页爬取等）。
- 可用于成本控制，防止实验或错误配置的资源导致的意外账单（尤其适用于云厂商会按次计费这种情况）。
- 允许多个用户公平共享服务。

限流有多种算法，之前我们已经实现了令牌桶算法，其他的算法还有固定窗口算法、滑动日志算法、漏桶算法等，每种算法都有其优点和缺点。我们来回顾一下最经典的几种限流算法，方便你根据需要选择理想的限流方案。

固定窗口算法

固定窗口算法（Fixed Window Algorithm）指的是，限制固定时间窗口内请求的处理个数。例如每小时只允许处理 1000 个请求，或每分钟只允许处理 10 个请求。每个传入请求都会增加窗口的计数器，并且计数器会在一段时间后重置。如果计数器超过了阈值，后面的请求将会被丢弃，直到计数器被重置为止。

固定窗口算法很容易实现，并且在限制请求速率方面做得很好。但是随着限制的重置，该算法会出现临界问题。也就是说，如果流量都集中在两个窗口的交界处，那么突发流量会是设置上限的两倍。

举个例子，我们设置每小时只能处理 1000 个请求。如果这 1000 个请求都刚好位于第一个小时的最后一分钟，而之后的 1000 个请求又刚好出现在下一个小时的第一分钟，就会导致在这短短的 2 分钟内，服务需要承受 2000 个请求，使服务不堪重负，示例代码如下。

复制代码

```
// FixedWindowLimiter 固定窗口限流器
type FixedWindowLimiter struct {
  limit    int           // 窗口请求上限
  window   time.Duration // 窗口时间大小
  counter  int           // 计数器
  lastTime time.Time     // 上一次请求的时间
  mutex    sync.Mutex    // 避免并发问题
}

func NewFixedWindowLimiter(limit int, window time.Duration) *FixedWindowLimiter
  return &FixedWindowLimiter{
    limit:    limit,
    window:   window,
```

```
    lastTime: time.Now(),
  }
}

func (l *FixedWindowLimiter) TryAcquire() error {
  l.mutex.Lock()
  defer l.mutex.Unlock()
  // 获取当前时间
  now := time.Now()
  // 如果当前窗口失效，计数器清0，开启新的窗口
  if now.Sub(l.lastTime) > l.window {
    l.counter = 0
    l.lastTime = now
  }
  // 若到达窗口请求上限，请求失败
  if l.counter >= l.limit {
    return errors.New("reach max limit")
  }
  // 若没到窗口请求上限，计数器+1，请求成功
  l.counter++
  return nil
}
```


滑动日志算法

还有一种常见的限流算法是滑动日志算法（Sliding Log）。它会记录下所有的请求时间点，系统会将这些日志存储在按照时间先后来排序的集合中。新请求到来时，会先判断指定时间范围内的请求数量是否超过阈值，超出阈值的请求会被丢弃。

这种方式避免了固定窗口算法容易遇到的请求突变问题，限流比较准确。不过因为它要记录下每次请求的时间点，所以会额外消耗内存与 CPU。下面的代码给出了滑动日志算法的示例，在该示例中，我们可以指定多个限速策略，例如策略 A 在 1 分钟内允许处理 100 个请求，并且策略 B 在 1 小时内允许处理 1000 个请求。


```
// ViolationStrategyError 违背策略错误
type ViolationStrategyError struct {
  Limit  int           // 窗口请求上限
  Window time.Duration // 窗口时间大小
}

func (e *ViolationStrategyError) Error() string {
  return fmt.Sprintf("violation strategy that limit = %d and window = %d", e.Li
}

// SlidingLogLimiterStrategy 滑动日志限流器的策略
```

```go
type SlidingLogLimiterStrategy struct {
  limit        int   // 窗口请求上限
  window       int64 // 窗口时间大小
  smallWindows int64 // 小窗口数量
}

func NewSlidingLogLimiterStrategy(limit int, window time.Duration) *SlidingLogL
  return &SlidingLogLimiterStrategy{
    limit:  limit,
    window: int64(window),
  }
}

// SlidingLogLimiter 滑动日志限流器
type SlidingLogLimiter struct {
  strategies  []*SlidingLogLimiterStrategy // 滑动日志限流器策略列表
  smallWindow int64                        // 小窗口时间大小
  counters    map[int64]int                // 小窗口计数器
  mutex       sync.Mutex                   // 避免并发问题
}

func NewSlidingLogLimiter(smallWindow time.Duration, strategies ...*SlidingLogL
  // 复制策略避免被修改
  strategies = append(make([]*SlidingLogLimiterStrategy, 0, len(strategies)), s

  // 不能不设置策略
  if len(strategies) == 0 {
    return nil, errors.New("must be set strategies")
  }

  // 排序策略，窗口时间大的排前面，相同窗口上限大的排前面
  sort.Slice(strategies, func(i, j int) bool {
    a, b := strategies[i], strategies[j]
    if a.window == b.window {
      return a.limit > b.limit
    }
    return a.window > b.window
  })
  fmt.Println(strategies[0], strategies[1])

  for i, strategy := range strategies {
    // 随着窗口时间变小，窗口上限也应该变小
    if i > 0 {
      if strategy.limit >= strategies[i-1].limit {
        return nil, errors.New("the smaller window should be the smaller limit"
      }
    }
    // 窗口时间必须能够被小窗口时间整除
    if strategy.window%int64(smallWindow) != 0 {
      return nil, errors.New("window cannot be split by integers")
    }
    strategy.smallWindows = strategy.window / int64(smallWindow)
```

```
  }

  return &SlidingLogLimiter{
    strategies:  strategies,
    smallWindow: int64(smallWindow),
    counters:    make(map[int64]int),
  }, nil
}

func (l *SlidingLogLimiter) TryAcquire() error {
  l.mutex.Lock()
  defer l.mutex.Unlock()

  // 获取当前小窗口值
  currentSmallWindow := time.Now().UnixNano() / l.smallWindow * l.smallWindow
  // 获取每个策略的起始小窗口值
  startSmallWindows := make([]int64, len(l.strategies))
  for i, strategy := range l.strategies {
    startSmallWindows[i] = currentSmallWindow - l.smallWindow*(strategy.smallWi
  }

  // 计算每个策略当前窗口的请求总数
  counts := make([]int, len(l.strategies))
  for smallWindow, counter := range l.counters {
    if smallWindow < startSmallWindows[0] {
      delete(l.counters, smallWindow)
      continue
    }
    for i := range l.strategies {
      if smallWindow >= startSmallWindows[i] {
        counts[i] += counter
      }
    }
  }

  // 若到达对应策略窗口请求上限，请求失败，返回违背的策略
  for i, strategy := range l.strategies {
    if counts[i] >= strategy.limit {
      return &ViolationStrategyError{
        Limit:  strategy.limit,
        Window: time.Duration(strategy.window),
      }
    }
  }

  // 若没到窗口请求上限，当前小窗口计数器+1，请求成功
  l.counters[currentSmallWindow]++
  return nil
}
```


滑动窗口算法

滑动窗口算法（Sliding Window）是一种混合方法，它有着固定窗口算法的低处理成本，并且和滑动日志算法一样，表征了在连续的时间范围内的请求量。

和固定窗口算法一样，滑动窗口算法会为每个固定窗口设置一个计数器。接下来，它会根据当前时间戳计算前一个窗口请求率的加权值，以平滑突发流量。例如，如果当前窗口经过了 25% 的时间，则当前窗口的计数权重为 25%，而前一个窗口的计数加权为 75%。

滑动窗口算法需要统计的数据量比滑动日志算法少很多，并且在大型分布式集群中仍然具有较强的扩展性。

滑动窗口算法的示例代码如下。在这段实例代码中，我们将一个固定窗口切割为了许多小窗口。举一个例子，假设我们把一个固定窗口分为了 10 个小窗口，计数就按照时间分布在这些小窗口中。假设当前窗口的时间过去了 20%，那么当前窗口的计数权重为 20%，而前一个窗口的计数加权为 80%。在计数时，只要将当前固定窗口的两个小窗口与上一个窗口的最后 8 个小窗口的数量加起来，并将该数量与阈值做对比即可。

复制代码

```
1
2  // SlidingWindowLimiter 滑动窗口限流器
3  type SlidingWindowLimiter struct {
4    limit         int           // 窗口请求上限
5    window        int64         // 窗口时间大小
6    smallWindow   int64         // 小窗口时间大小
7    smallWindows  int64         // 小窗口数量
8    counters      map[int64]int // 小窗口计数器
9    mutex         sync.Mutex    // 避免并发问题
10 }
11
12 func NewSlidingWindowLimiter(limit int, window, smallWindow time.Duration) (*Sl
13   // 窗口时间必须能够被小窗口时间整除
14   if window%smallWindow != 0 {
15     return nil, errors.New("window cannot be split by integers")
16   }
17
18   return &SlidingWindowLimiter{
19     limit:         limit,
20     window:        int64(window),
21     smallWindow:   int64(smallWindow),
22     smallWindows: int64(window / smallWindow),
23     counters:      make(map[int64]int),
24   }, nil
25 }
26
27 func (l *SlidingWindowLimiter) TryAcquire() bool {
```

```
  l.mutex.Lock()
  defer l.mutex.Unlock()

  // 获取当前小窗口值
  currentSmallWindow := time.Now().UnixNano() / l.smallWindow * l.smallWindow
  // 获取起始小窗口值
  startSmallWindow := currentSmallWindow - l.smallWindow*(l.smallWindows-1)

  // 计算当前窗口的请求总数
  var count int
  for smallWindow, counter := range l.counters {
    if smallWindow < startSmallWindow {
      delete(l.counters, smallWindow)
    } else {
      count += counter
    }
  }

  // 若到达窗口请求上限，请求失败
  if count >= l.limit {
    return false
  }
  // 若没到窗口请求上限，当前小窗口计数器+1，请求成功
  l.counters[currentSmallWindow]++
  return true
}
```


漏桶算法

漏桶算法（Leaky Bucket）利用队列提供了一种简单、直观的方法来限制请求的速率。

每一个请求都像一滴水，请求发出之后会先被放到一个队列（漏桶）中，桶底有一个孔，不断地漏出水滴。这就好像消费者不断地在消费队列中的内容，消费的速率（漏出的速度）等于限流阈值。漏桶有大小之分，它对应的是队列的容量，如果请求的速率大于了消费的速率，请求会被暂存在桶中。当请求堆积到超过指定容量时，就像是水溢出了一样，会触发拒绝策略。

漏桶算法中的消费处理总是能以恒定的速度进行，这可以很好地保护自身系统不被突如其来的流量冲垮。但是，突发流量可能会使旧请求填满队列，导致最近的请求得不到处理。此外，它也不能保证请求一定会在某个时间段内得到处理。漏桶算法的示例代码如下。


```
// LeakyBucketLimiter 漏桶限流器
type LeakyBucketLimiter struct {
  peakLevel       int         // 最高水位
```

```
  currentLevel    int         // 当前水位
  currentVelocity int         // 水流速度/秒
  lastTime        time.Time   // 上次放水时间
  mutex           sync.Mutex // 避免并发问题
}

func NewLeakyBucketLimiter(peakLevel, currentVelocity int) *LeakyBucketLimiter
  return &LeakyBucketLimiter{
    peakLevel:       peakLevel,
    currentVelocity: currentVelocity,
    lastTime:        time.Now(),
  }
}

func (l *LeakyBucketLimiter) TryAcquire() bool {
  l.mutex.Lock()
  defer l.mutex.Unlock()

  // 尝试放水
  now := time.Now()
  // 距离上次放水的时间
  interval := now.Sub(l.lastTime)
  if interval >= time.Second {
    // 当前水位-距离上次放水的时间(秒)*水流速度
    l.currentLevel = maxInt(0, l.currentLevel-int(interval/time.Second)*l.curre
    l.lastTime = now
  }

  // 若到达最高水位，请求失败
  if l.currentLevel >= l.peakLevel {
    return false
  }
  // 若没有到达最高水位，当前水位+1，请求成功
  l.currentLevel++
  return true
}

func maxInt(a, b int) int {
  if a > b {
    return a
  }
  return b
}
```


用 go-micro 实现限流

除了常见的限流算法，我们还可以使用 go-micro 框架来实现限流。这也是使用微服务框架的又一好处，因为框架中通常有配套的限流功能。在 go-micro 的插件库中封装了

github.com/juju/ratelimit 和 go.uber.org/ratelimit 两个限流库，分别提供了限流的令牌桶算法和漏桶算法。而且它们既可以在 go-micro 的客户端中也可以在服务器中使用。

要在 go-micro GRPC API 中使用限流功能，首先要导入 go-micro 限流的插件库，同时导入与其配套的第三方限流库：github.com/juju/ratelimit。如下所示，ratelimit.NewBucketWithRate 可以设置令牌桶的参数。其中，第一个参数表示的是速度，在下面的例子中，第一个参数为 0.5，表示每秒钟放入的令牌个数为 0.5 个。第二个参数为桶的大小。

复制代码

```
import (
  ratePlugin "github.com/go-micro/plugins/v4/wrapper/ratelimiter/ratelimit"
  "github.com/juju/ratelimit"
)
func RunGRPCServer(m *master.Master, logger *zap.Logger, reg registry.Registry,
  b := ratelimit.NewBucketWithRate(0.5, 1)
  service := micro.NewService(
    ...
    micro.WrapHandler(ratePlugin.NewHandlerWrapper(b, false))
  )
```

micro.WrapHandler 包装了 go-micro 的 Server 端设置的限流中间件。ratePlugin.NewHandlerWrapper 的第一个参数为之前设置的令牌桶，第二个参数可以指定当请求速率超过阈值时，是否堵塞住。此处为 false，表示不堵塞并立即返回错误。

这个中间件的原理也很简单，它在进行实际的 GRPC 方法之前先调用了限流函数。

复制代码

```
func NewHandlerWrapper(b *ratelimit.Bucket, wait bool) server.HandlerWrapper {
  fn := limit(b, wait, "go.micro.server")

  return func(h server.HandlerFunc) server.HandlerFunc {
    return func(ctx context.Context, req server.Request, rsp interface{}) error
      if err := fn(); err != nil {
        return err
      }
      return h(ctx, req, rsp)
    }
  }
}
```

接下来我们检验一下服务限流的效果。

启动 Master 服务，并快速地调用两次服务。如下所示，会发现第二次调用时，服务返回 429 状态码，并提示处理的请求太多。等待一秒后再次调用，发现又可以正常运行了，这就说明限流功能是正常的。

复制代码

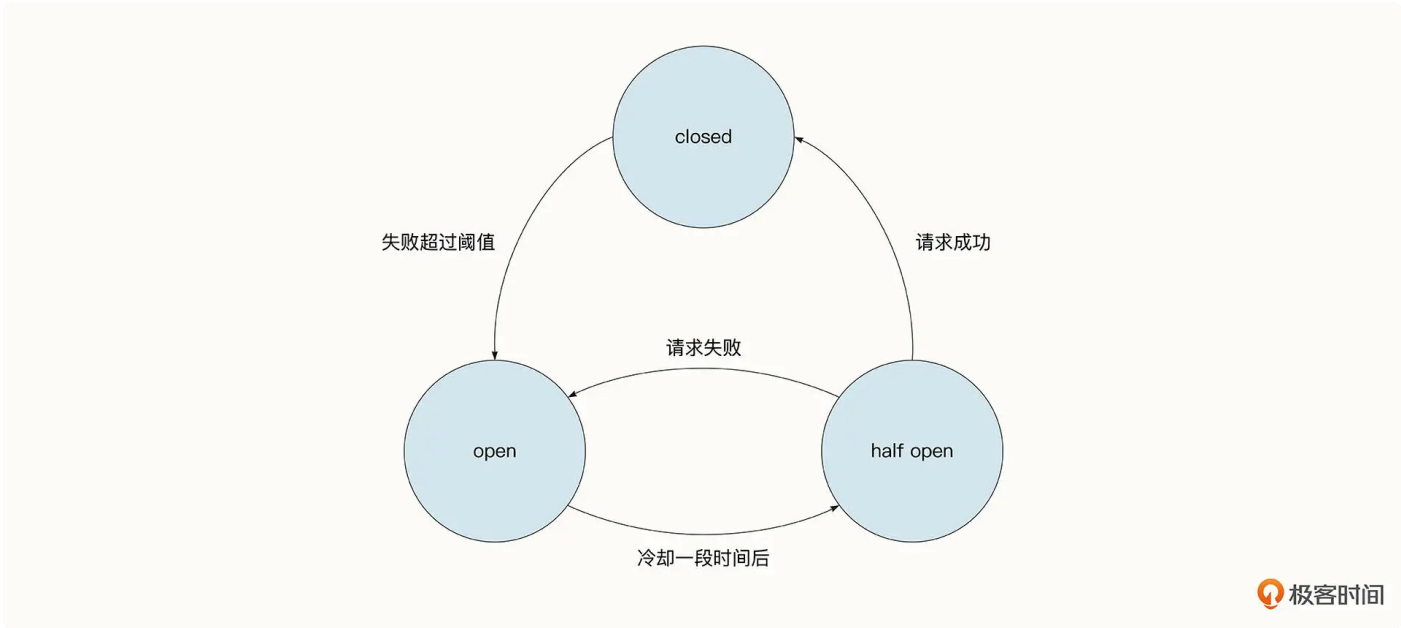
```
» curl -H "content-type: application/json" -d '{"id":"zjx","name": "douban_book
{"code":2, "message":"no worker nodes", "details":[]}

» curl -H "content-type: application/json" -d '{"id":"zjx","name": "douban_book
{"code":2, "message":"{\"id\":\"go.micro.server\",\"code\":429,\"detail\
```

熔断器

实现了限流之后，我们再来看看熔断。我们是通过熔断器来实现熔断的。熔断器是当依赖的下游服务异常时，负责在一段时间内禁止访问依赖服务的一种系统组件，它可以防止有问题的依赖服务拖垮自身系统。熔断器有下面三个状态。

- 熔断器关闭状态，表明当前服务可以正常访问下游的依赖服务。
- 熔断器打开状态，表明当前下游的依赖服务有问题，将被禁止访问。
- 熔断器半开状态，当前阶段会尝试让一部分请求访问下游依赖服务，如果能够正常访问，则会进入熔断器关闭状态。如果仍然无法正常访问，则会维持熔断器打开状态。



社区中已经有一些非常优秀的开源熔断组件了，例如 hystrix-go、gobreaker、Sentinel。Micro 也提供了对上述熔断组件进行了简单封装的插件，包括 hystrix 插件和 gobreaker 插件。

go-mico 实现熔断器

我们来看一看如何在 go-micro 中使用熔断器。方法非常简单，调用 micro.WrapClient 用于注入 GRPC client 的中间件，在这里我们使用了 hystrix 插件控制熔断器的行为。

复制代码

```
import (
  "github.com/go-micro/plugins/v4/wrapper/breaker/c"
)

func RunGRPCServer(m *master.Master, logger *zap.Logger, reg registry.Registry,
  service := micro.NewService(
    ...
    micro.WrapClient(hystrix.NewClientWrapper()),
  )
}
```

之后，所有从此 micro client 发出的服务调用都会受到熔断插件的限制和保护。当失败次数超过阈值时，调用方会立即接收到熔断错误。熔断器的配置参数位于 `hystrix-go` 库中，默认的 5 个重要配置如下所示。

复制代码

```
  DefaultTimeout = 1000
  DefaultMaxConcurrent = 10
  DefaultVolumeThreshold = 20
  DefaultSleepWindow = 5000
  DefaultErrorPercentThreshold = 50
```

其中，`DefaultTimeout` 为请求的超时时间，`DefaultMaxConcurrent` 为最大的并发数量。

`DefaultVolumeThreshold` 为触发断路器的最小数量，如果没有触发断路器的最小数量，就不用熔断，这避免了低峰期的干扰。例如，如果当前只有一个请求，并且访问失败了，那么当前的失败率就是 100%，这种情况下立即熔断服务是不合理的。

DefaultSleepWindow 为断路器打开状态时，它会告诉我们需要等待多长时间再次检测当前链路的状态。DefaultErrorPercentThreshold 为失败率的阈值，当失败率超过该阈值时，将触发熔断。

在 go-micro 当中，对熔断配置的维度是接口级别的，这一点可以从hystrix 插件库中查看到，hystrix.DoC 的第二个参数就是熔断的维度。例如，当我们调用 Master 添加资源的接口，熔断的维度为：go.micro.server.master.CrawlerMaster.AddResource。

```
func (cw *clientWrapper) Call(ctx context.Context, req client.Request, rsp inte
  var err error
  herr := hystrix.DoC(ctx, req.Service()+"."+req.Endpoint(), func(c context.Con
    err = cw.Client.Call(c, req, rsp, opts...)
    if cw.filter != nil {
      if cw.filter(ctx, err) {
        return nil
      }
    }
    return err
  }, cw.fallback)
  if herr != nil {
    return herr
  }
  // return original error
  return err
}
```

我们可以调用插件库封装的 hystrix. ConfigureCommand 函数修改某一个接口的熔断配置，如下所示。也可以调用 hystrix.ConfigureDefault 函数修改所有接口的默认熔断参数。

```
hystrix.ConfigureCommand("go.micro.server.master.CrawlerMaster.AddResource", hy
    Timeout:                10000,
    MaxConcurrentRequests:  100,
    RequestVolumeThreshold: 10,
    SleepWindow:            6000,
    ErrorPercentThreshold:  30,
  })
```

认证与鉴权

在访问微服务集群的过程中，我们还时常需要进行用户的认证（Authentication）与鉴权（Authorization）。

认证的目的是检查用户的身份是否合法，防止匿名用户访问等。而鉴权是为了检查用户是否有权限操作请求的资源。

认证通常是访问服务的第一步，我们比较常见的方式是通过用户名和密码来验证用户的有效性。不过，使用用户名和密码是比较繁琐耗时的：在服务端我们通常需要考虑用密文来存储密码，还需要操作数据库。

因此从性能和安全性的角度考虑，在实践中，当我们第一次完成身份验证之后，服务器会为我们发送一个凭证，即一个 Token。之后，客户端访问服务器时都需要带上这一个 Token。只要 Token 合法且在有效期内，就可以快速地完成验证了。Token 的有效期通常比较短，这样即便黑客在以后获取了 Token，该 Token 也已经变得无效了。而利用 Token 进行身份认证落在实践中，我们最常用的就是 JWT。

JWT（Json Web Token）是基于 JSON 的开放标准（RFC 7519）定义的一种紧凑、独立的格式，使用它可以在各方之间安全地传输信息。JWT 可以使用对称加密算法（HMAC 算法）或者非对称加密算法（RSA，ECDSA）进行签名。一般我们会更多使用非对称加密算法，因为它更安全且能够验证信息的完整性。JWT 可以用于下面几种场景。

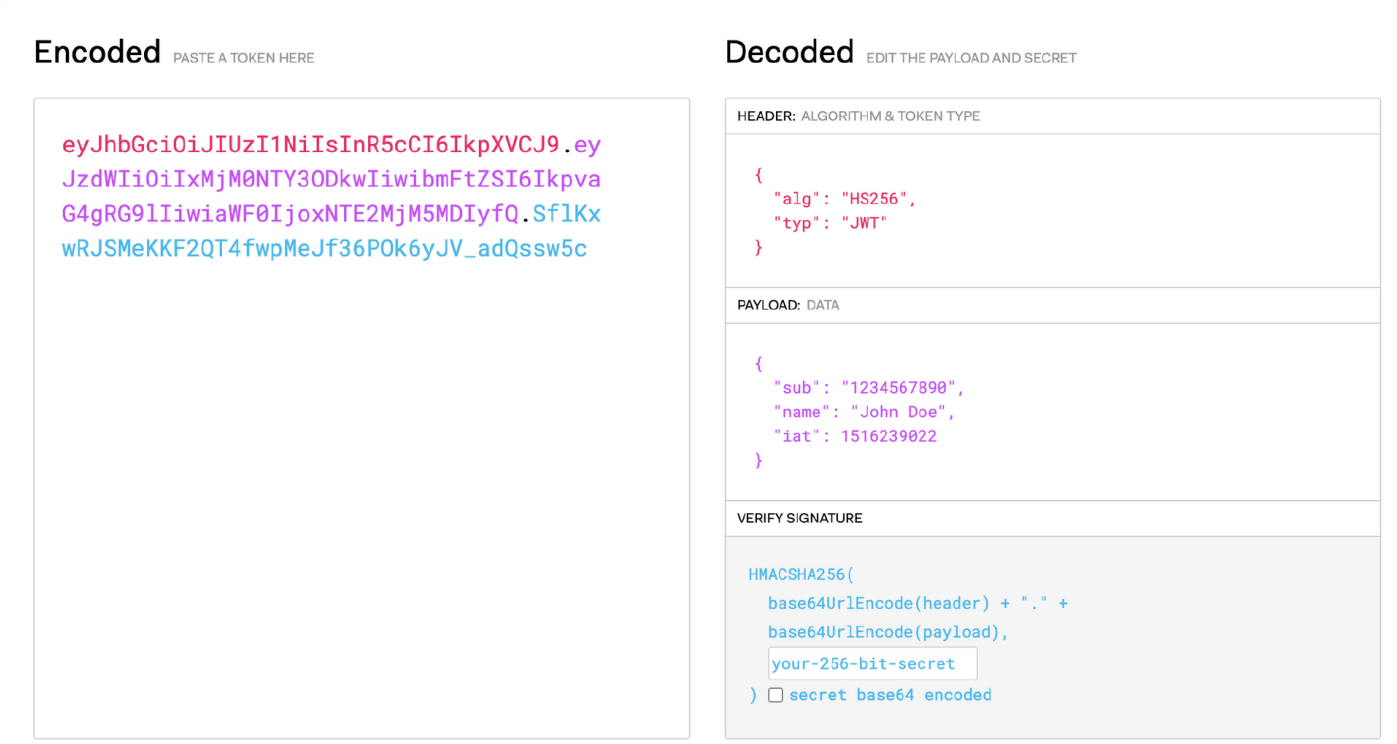
- 单点登录（SSO）
  SSO 是一种身份验证解决方案，可以让用户通过一次性用户身份验证登录多个应用程序和网站。JWT 可以实现单点登录是因为 JWT 的开销很小，并且能够轻松跨域使用。
- 信息交换
  JWT 是一种在各方之间安全传输信息的好方法。因为 JWT 可以使用公钥 / 私钥对签名进行验证，你可以确定发送者的真实身份。此外，由于签名中包含了具体的内容，因此你还可以验证内容是否被篡改。

我们再来看下 JWT 的组成。JWT 由 Header、Payload、Signature 三个对象组成，正如下面这个演示网站一样，左边为编码后的信息，右边为编码前的三个对象。每个对象都是一个 JSON 结构体。

第一个对象是 Header，它包含 alg 和 typ 两个字段，alg 表示签名的算法，typ 表示类型（JWT）。

第二对象是 Payload，它表示载荷，包含用户名、过期时间等信息，可以自定义添加字段。

第三个对象是签名。它首先将 Header、Payload 使用 base64 url 编码，然后将编码后的字符串用"."连接在一起，最后用我们选择的算法进行签名。



从这里我们可以看出，JWT 中包含了服务器认证需要的所有信息，在服务器中只需要有对应的私钥进行解密，就可以知道信息的完整性，用户的有效性，还可以额外地传递一些其他信息。

在实践中，认证一般分为下面三步。

1. 客户端使用用户名和密码访问服务器的登录接口。
2. 当服务器认证成功后，生成 JWT。
3. 客户端在之后的请求中，都带上 JWT，由服务器对 Token 进行验证。

第一步和第二步可能是在单独的授权服务中做的，而第三步则需要对应的服务提供 Token 的校验支持。

go-micro 中也提供了对于 JWT 的支持，可以轻松地生成并校验 JWT。说到这儿，你很容易想到中间件，没错，我们可以通过一个中间件来提供对 JWT 的校验。下面是我生成的一个校验 JWT token 的中间件。

复制代码

```
func NewAuthWrapper(service micro.Service) server.HandlerWrapper {
  return func(h server.HandlerFunc) server.HandlerFunc {
    return func(ctx context.Context, req server.Request, rsp interface{}) error
      // Fetch metadata from context (request headers).
      md, b := metadata.FromContext(ctx)
      if !b {
        return errors.New("no metadata found")
      }

      // Get auth header.
      authHeader, ok := md["Authorization"]
      if !ok || !strings.HasPrefix(authHeader, auth.BearerScheme) {
        return errors.New("no auth token provided")
      }

      // Extract auth token.
      token := strings.TrimPrefix(authHeader, auth.BearerScheme)

      // Extract account from token.
      a := service.Options().Auth
      _, err := a.Inspect(token)
      if err != nil {
        return errors.New("auth token invalid")
      }

      return h(ctx, req, rsp)
    }
  }
}
```

其中，service.Options().Auth 是 go-micro 提供的认证与鉴权的接口。

复制代码

```
type Auth interface {
  // Init the auth
  Init(opts ...Option)
  // Options set for auth
  Options() Options
  // Generate a new account
  Generate(id string, opts ...GenerateOption) (*Account, error)
  // Inspect a token
```

```
  Inspect(token string) (*Account, error)
  // Token generated using refresh token or credentials
  Token(opts ...TokenOption) (*Token, error)
  // String returns the name of the implementation
  String() string
}
```


而要使用 JWT 插件功能，需要导入 go-micro 的 jwt 插件库，并设置之前处理 JWT 的中间件即可。

复制代码

```
import (
  _ "github.com/go-micro/plugins/v4/auth/jwt"
)

var (
  name    = "helloworld"
  version = "latest"
)

func main() {

  // Create service
  srv := micro.NewService(...)

  srv.Init(
    micro.Name(name),
    micro.Version(version),
    micro.WrapHandler(NewAuthWrapper(srv)),
  )
```

go-micro 还提供基于 JWT 的一些基本的授权能力，限制用户访问的路由、服务和资源。这些能力比较简单，这里就不再讨论了。

总结

总结一下，这节课，我们实践了保证大规模微服务集群正常运行需要具备的重要功能，即限流、熔断与认证。

限流指的是限制给定时间内事件发生的频率，这能够保证外部的请求始终位于服务器能够承载的范围内。这节课我们介绍了几种经典的限流算法和原理，它们各有特点，需要根据实际业务场景进行选择。

熔断类似于断路器，当发现依赖服务异常时，它可以在一段时间内禁止访问依赖服务，防止依赖服务拖垮自身或者返回错误的数据。

而认证与鉴权用于检查访问者的身份与权限，精准对访问者进行权限的控制，防止越权的操作。

这几种能力保证了微服务集群的正常运行，是大规模微服务集群中必不可少的治理手段。我们还分别讲解了如何用 go-micro 框架来实现这些能力。go-micro 微服务框架有着丰富的插件库，能够帮助我们快速实现这些治理功能，让开发者更关注于开发核心的业务逻辑。

课后题

最后，还是给你留一道思考题。

常见的权限控制方式有哪些？kubernetes 与 etcd 为什么都选择了 RBAC 进行权限控制？

欢迎你给我留言交流讨论，我们下节课见。

上一篇　48 | 完善核心能力：Master请求转发与Worker资源管理

下一篇　特别放送丨Go泛型：用法、原理与最佳实践

精选留言