熔断机制（Circuit Breaker）指的是在股票市场的交易时间中，当价格的波动幅度达到某一个限定的目标（熔断点）时，对其暂停交易一段时间的机制。此机制如同保险丝在电流过大时候熔断，故而得名。熔断机制推出的目的是为了防范系统性风险，给市场更多的冷静时间，避免恐慌情绪蔓延导致整个市场波动，从而防止大规模股价下跌现象的发生。

同样的，在高并发的分布式系统
设计中，也应该有熔断的机制。熔断一般是在客户端（调用端）进行配置，当客户端向服务端发起请求的时候，服务端的错误不断地增多，这时候就可能会触发熔断，触发熔断后客户端的请求不再发往服务端，而是在客户端直接拒绝请求，从而可以保护服务端不会过载。这里说的
服务端可能是rpc
服务，http服务，也可能是mysql，redis
等。注意熔断是一种有损的机制，当熔断后可能需要一些降级的策略进行配合。

熔断原理

现代微服务
架构基本都是分布式的，整个分布式系统是由非常多的微服务组成。不同服务之间相互调用，组成复杂的调用链路。在复杂的调用链路中的某一个服务如果不稳定，就可能会层层级联，最终可能导致整个链路全部挂掉。因此我们需要对不稳定的服务依赖进行熔断降级，暂时切断不稳定的服务调用，避免局部不稳定因素导致整个分布式系统的雪崩。

说白了，我觉得熔断就像是那些容易异常服务的一种代理，这个代理能够记录最近调用发生错误的次数，然后决定是继续操作，还是立即返回错误。

熔断器内部维护了一个熔断器状态机，状态机的转换关系如下图所示：

熔断器有三种状态：

- Closed状态：也是
  初始状态，我们需要一个调用失败的计数器

，如果调用失败，则使失败次数加1。如果最近失败次数超过了在给定时间内允许失败的阈值，则切换到open
状态，此时开启一个超时时钟，当到达超时时钟时间后，则切换到Half Open状态，该超时时间的设定是给了系统一次机会来修正导致调用失败的错误，以回到正常的工作状态。在Closed状态下，错误计数是基于时间的。在特定的时间间隔内会自动重置，这能够防止由于某次的偶然错误导致熔断器进入Open状态，也可以基于连续失败的次数。
- Open状态：在该状态下，客户端请求会立即返回错误响应，而不调用服务端。
- Half-Open状态：允许客户端一定数量的去调用服务端，如果这些请求对服务的调用成功，那么可以认为之前导致调用失败的错误已经修正，此时熔断器切换到Closed状态，同时将错误计数器重置。如果这一定数量的请求有调用失败的情况，则认为导致之前调用失败的的问题仍然存在，熔断器切回到断开状态，然后重置计时器来给系统一定的时间来修正错误。Half-Open状态能够有效防止正在恢复中的服务被突然而来的大量请求再次打挂。

下图是Netflix的开源项目Hystrix中的熔断器的实现逻辑：

从这个流程图中，可以看到：

1. 有请求来了，首先allowRequest()函数判断是否在熔断中，如果不是则放行，如果是的话，还要看有没有达到一个熔断时间片，如果熔断时间片到了，也放行，否则直接返回错误。
2. 每次调用都有两个函数makeSuccess(duration)和makeFailure(duration)来统计一下在一定的duration内有多少是成功还是失败的。
3. 判断是否熔断的条件isOpen()，是计算failure/(success+failure)当前的错误率，如果高于一个阈值，那么熔断器打开，否则关闭。
4. Hystrix会在内存中维护一个数据，其中记录着每一个周期的请求结果的统计，超过时长长度的元素会被删除掉。

熔断器实现

了解了熔断的原理后，我们来自己实现一套熔断器。

熟悉go-zero的朋友都知道，在go-zero中熔断没有采用上面介绍的方式，而是参考了《Google Sre》 采用了一种自

适应的熔断机制，这种自适应的方式有什么好处呢？下文会基于这两种机制做一个对比。

下面我们基于上面介绍的熔断原理，实现一套自己的熔断器。

代码路径：go-zero/core/breaker/hystrixbreaker.go

熔断器默认的状态为Closed，当熔断器打开后默认的冷却时间是5秒钟，当熔断器处于HalfOpen状态时默认的探测时间为200毫秒，默认使用rateTripFunc方法来判断是否触发熔断，规则是采样大于等于200且错误率大于50%，使用滑动窗口来记录请求总数和错误数。

```
func newHystrixBreaker() *hystrixBreaker {
  bucketDuration := time.Duration(int64(window) / int64(buck
ets))
  stat := collection.NewRollingWindow(buckets, bucketDuratio
n)
  return &hystrixBreaker{
    state:          Closed,
    coolingTimeout: defaultCoolingTimeout,
    detectTimeout:  defaultDetectTimeout,
    tripFunc:       rateTripFunc(defaultErrRate, defaultMinS
ample),
    stat:           stat,
    now:            time.Now,
  }
}


func rateTripFunc(rate float64, minSamples int64) TripFunc {
  ret
urn func(rollingWindow *collection.RollingWindow) bool {
    var total, errs int64
    rollingWindow.Reduce(func(b *collection.Bucket) {
      total += b.Count
      errs += int64(b.Sum)
    })
    errRate := float64(errs) / float64(total)
    return total >= minSamples && errRate > rate
```

```
  }
}
```

每次请求都会调用doReq方法，在该方法中，首先通过accept()方法判断是否拒绝本次请求，拒绝则直接返回熔断错误。否则执行req()真正的发起服务端调用，成功和失败分别调用b.markSuccess()和b.markFailure()

```
func (b *hystrixBreaker) doReq(req func() error, fallback fu
nc(error) error, acceptable Acceptable) error {
  if err := b.accept(); err != nil {
    if fallback != nil {
      return fallback(err)
    }
    return err
  }

  defer func() {
    if e := recover(); e != nil {
      b.markFailure()
      panic(e)
    }
  }()

  err := req()
  if acceptable(err) {
    b.markSuccess()
  } else {
    b.markFailure()
  }

  return err
}
```

在accept()方法中，首先获取当前熔断器状态，当熔断器处于Closed状态直接返回，表示正常处理本次请求。

当前状态为Open的时候，判断冷却时间是否过期，如果没有过期的话则直接返回

熔断错误拒绝本次请求，如果过期的话则把熔断器状态更改为HalfOpen，冷却时间的主要目的是给服务端一些时间进行故障恢复，避免持续请求把服务端打挂。

当前状态为HalfOpen的时候，首先判断探测时间间隔，避免探测过于频繁，默认使用200毫秒作为探测间隔。

```
func (b *hystrixBreaker) accept() error {
  b.mux.Lock()
  switch b.getState() {
  case Open:
    now := b.now()
    if b.openTime.Add(b.coolingTimeout).After(now) {
      b.mux.Unlock()
      return ErrServiceUnavailable
    }
    if b.getState() == Open {
      atomic.StoreInt32((*int32)(&b.state), int32(HalfOpen))
      atomic.StoreInt32(&b.halfopenSuccess, 0)
      b.lastRetryTime = now
      b.mux.Unlock()
    } else {
      b.mux.Unlock()
      return ErrServiceUnavailable
    }
  case HalfOpen:
    now := b.now()
    if b.lastRetryTime.Add(b.detectTimeout).After(now) {
      b.mux.Unlock()
      return ErrServiceUnavailable
    }
    b.lastRetryTime = now
    b.mux.Unlock()
  case Closed:
    b.mux.Unlock()
  }

  return nil
}
```

如果本次请求正常返回，则调用markSuccess()方法，如果当前熔断器处于HalfOpen状态，则判断当前探测成功数量是否大于默认的探测成功数量，如果大于则把熔断器的状态更新为Closed。

```
func (b *hystrixBreaker) markSuccess() {
  b.mux.Lock()
  switch b.getState() {
  case Open:
    b.mux.Unlock()
  case HalfOpen:
    atomic.AddInt32(&b.halfopenSuccess, 1)
    if atomic.LoadInt32(&b.halfopenSuccess) > defaultHalfOpe
nSuccesss {
      atomic.StoreInt32((*int32)(&b.state), int32(Closed))
      b.stat.Reduce(func(b *collection.Bucket) {
        b.Count = 0
        b.Sum = 0
      })
    }
    b.mux.Unlock()
  case Closed:
    b.stat.Add(1)
    b.mux.Unlock()
  }
}
```

在markFailure()方法中，如果当前状态是Closed通过执行tripFunc来判断是否满足熔断条件，如果满足则把熔断器状态更改为Open状态。

```
func (b *hystrixBreaker) markFailure() {
  b.mux.Lock()
  b.stat.Add(0)
  switch b.getState() {
  case Open:
    b.mux.Unlock()
  case HalfOpen:
    b.openTime = b.now()
    atomic.StoreInt32((*int32)(&b.state), int32(Open))
```

```
    b.mux.Unlock()
  case Closed:
    if b.tripFunc != nil && b.tripFunc(b.stat) {
      b.openTime = b.now()
      atomic.StoreInt32((*int32)(&b.state), int32(Open))
    }
    b.mux.Unlock()
  }
}
```

熔断器的实现逻辑总体比较简单，阅读代码基本都能理解，这部分代码实现的比较仓促，可能会有bug，如果大家发现bug可以随时联系我进行修正。

hystrixBreaker和googlebreaker对比

接下来对比一下两种熔断器的熔断效果。

这部分示例代码在：go-zero/example下

分别定义了user-api和user-rpc服务，user-api作为客户端对user-rpc进行请求，user-rpc作为服务端响应客户端请求。

在user-rpc的示例方法中，有20%的几率返回错误。

```
func (l *U
serInfoLogic) UserIn
fo(in *user.UserInfoRequest
) (*user.UserInfoResponse, error) {
  ts := time.Now().UnixMilli()
  if in.UserId == int64(1) {
    if ts%5 == 1 {
      return nil, status.Error(codes.Internal, "internal err
or")
    }
    return &user.UserInfoResponse{
      UserId: 1,
      Name:   "jack",
    }, nil
```

```
  }
  return &user.UserInfoResponse{}, nil
}
```

在user-api的示例方法中，对user-rpc发起请求，然后使用prometheus指标记录正常请求的数量。

```
var metricSuccessReqTotal = metric.NewCounterVec(&metric.Cou
nterVecOpts{
  Namespace: "circuit_breaker",
  Subsystem: "requests",
  Name:      "req_total",
  Help:      "test for circuit breaker",
  Labels:    []string{"method"},
})

func (l *UserInfoLogic) UserInfo() (resp *types.UserInfoResp
onse, err error) {
  for {
    _, err := l.svcCtx.UserRPC.UserInfo(l.ctx, &user.UserInf
oRequest{UserId: int64(1)})
    if err != nil && err == breaker.ErrServiceUnavailable {
      fmt.Println(err)
      continue
    }
    metricSuccessReqTotal.Inc("UserInfo")
  }

  return &types.UserInfoResponse{}, nil
}
```

启动两个服务，然后观察在两种熔断策略下正常请求的数量。

googleBreaker熔断器的正常请求率如下图所示：

hystrixBreaker熔断器的正常请求率如下图所示：

从上面的实验结果可以看出，go-zero内置的googleBreaker的正常请求数是高于hystrixBreaker的。这是因为hystrixBreaker维护了三种状态，当进入Open状态后为了避免继续对服务端发起请求造成压力，会使用一个冷却时钟，而在这段时间里是不会放过任何请求的，同时，从HalfOpen状态变为Closed状态后，瞬间又会有大量的请求发往服务端，这时服务端很可能还没恢复，从而导致熔断器又变为Open状态。而google Breaker采用的是一种自适应的熔断策略，也不需要多种状态，也不会像hystrixBreaker那样一刀切，而是会尽可能多的处理请求，这不也是我们期望的嘛，毕竟熔断对客户来说是有损的。下面我们来一起学习下go-zero内置的熔断器googleBreaker。

源码解读

googleBreaker的代码路径在：go-zero/core/breaker/googlebreaker.go

在doReq()方法中通过accept()方法判断是否触发熔断，如果触发熔断则返回error，这里如果定义了回调函数的话可以执行回调，比如做一些降级数据的处理等。如果请求正常则通过markSuccess()给总请求数和正常请求数都加1，如果请求失败通过markFailure则只给总请求数加1。

```
func (b *googleBreaker) doReq(req func() error, fallback func(err error) error, acceptable Acceptable) error {
  if err := b.accept(); err != nil {
    if fallback != nil {
      return fallback(err)
    }

    return err
  }

  defer func() {
    if e := recover(); e != nil {
      b.markFailure()
      panic(e)
```

```
    }
  }()

  err := req()
  if acceptable(err) {
    b.markSuccess()
  } else {
    b.markFailure()
  }

  return err
}
```

在accept()方法中通过计算判断是否触发熔断。

在该算法中，需要记录两个请求数，分别是：

- 请求总量（requests）：调用方发起请求的数量总和
- 正常处理的请求数量（accepts）：服务端正常处理的请求数量

在正常情况下，这两个值是相等的，随着被调用方服务出现异常开始拒绝请求，请求接受数量(accepts)的值开始逐渐小于请求数量(requests)，这个时候调用方可以继续发送请求，直到requests = K * accepts，一旦超过这个限制，熔断器就会打开，新的请求会在本地以一定的概率被抛弃直接返回错误，概率的计算公式如下：

```
max(0, (requests - K * accepts) / (requests + 1))
```

通过修改算法中的K(倍值)，可以调节熔断器的敏感度，当降低该倍值会使自适应熔断算法更敏感，当增加该倍值会使得自适应熔断算法降低敏感度，举例来说，假设将调用方的请求上限从 requests = 2 acceptst 调整为 requests = 1.1 accepts 那么就意味着调用方每十个请求之中就有一个请求会触发熔断。

```
func (b *googleBreaker) accept() error {
  accepts, total := b.history()
  weightedAccepts := b.k * float64(accepts)
  // https://landing.google.com/sre/sre-
book/chapters/handling-overload/#eq2101
```

```
  dropRatio := math.Max(0, (float64(total-
protection)-weightedAccepts)/float64(total+1))
  if dropRatio <= 0 {
    return nil
  }

  if b.proba.TrueOnProba(dropRatio) {
    return ErrServiceUnavailable
  }

  return nil
}
```

history从滑动窗口中统计当前的总请求数和正常处理的请求数。

```
func (b *googleBreaker) history() (accepts, total int64) {
  b.stat.Reduce(func(b *collection.Bucket) {
    accepts += int64(b.Sum)
    total += b.Count
  })

  return
}
```

结束语

本篇文章介绍了服务治理中的一种客户端节流机制 - 熔断。在hystrix熔断策略中需要实现三个状态，分别是Open、HalfOpen和Closed。不同状态的切换时机在上文中也有详细描述，大家可以反复阅读理解，最好是能自己动手实现一下。对于go-zero内置的熔断器是没有状态的，如果非要说它的状态的话，那么也只有打开和关闭两种情况，它是根据当前请求的成功率自适应的丢弃请求，是一种更弹性的熔断策略，丢弃请求概率随着正常处理的请求数不断变化，正常处理的请求越多丢弃请求的概率就越低，反之丢弃请求的概率就越高。

虽然熔断的原理都一样，但实现机制不同导致的效果可能也不同，在实际生产中可以根据实际情况选择符合业务场景的熔断策略。

希望本篇文章对你有所帮助。

原文链接：https://mp.weixin.qq.com/s?__biz=Mzg2ODU1MTI0OA==&mid=2247485915&idx=1&sn=50b01d7a7d67a1ba4ef206800c3828b8&utm_source=tuicool&utm_medium=referral