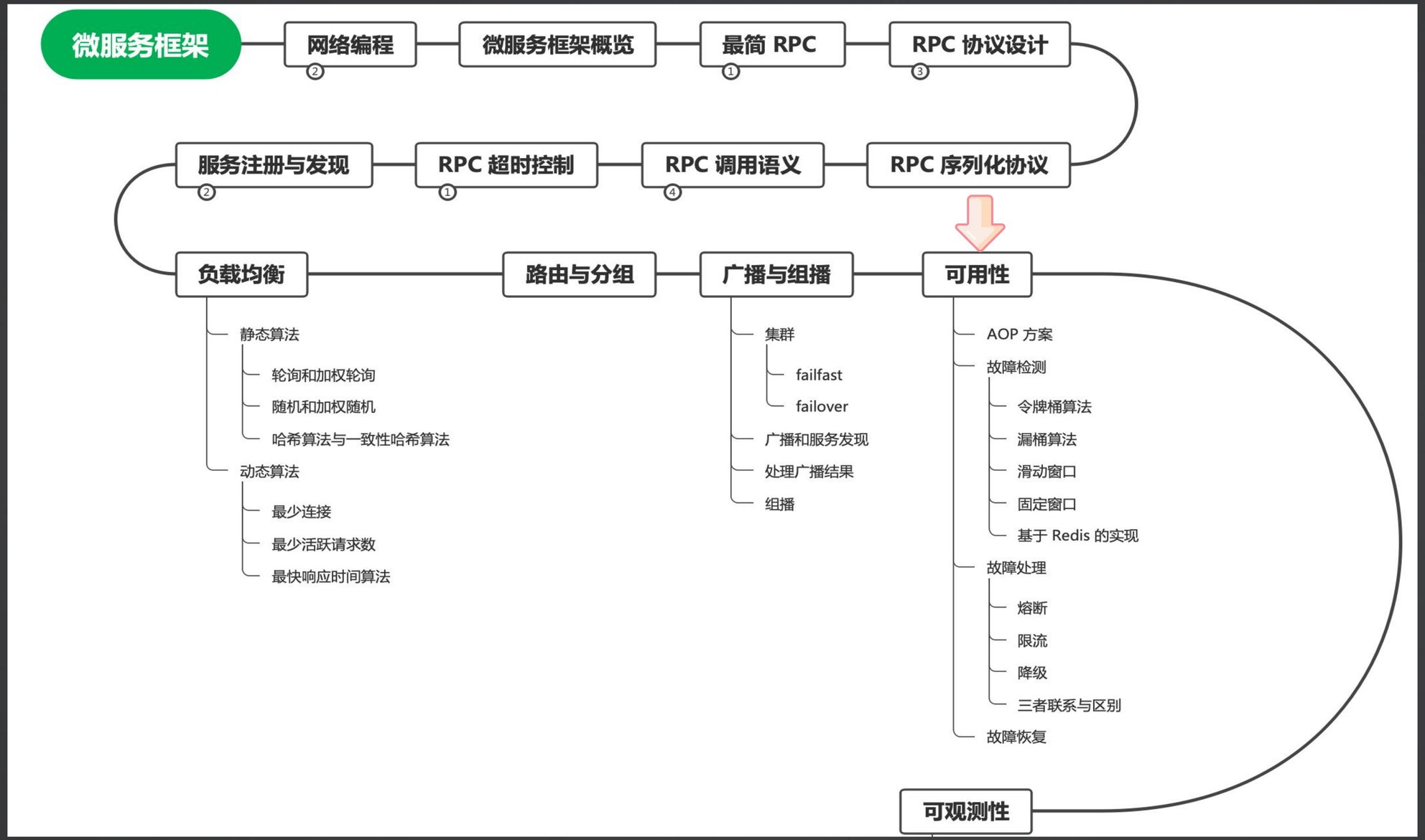
微服务框架 —— 可用性

大明

学习路线


微服务框架
网络编程
2
微服务框架概览
最简 RPC
1
RPC 协议设计
3
RPC 序列化协议
4
RPC 调用语义
1
RPC 超时控制
2
服务注册与发现
负载均衡
静态算法
轮询和加权轮询
随机和加权随机
哈希算法与一致性哈希算法
动态算法
最少连接
最少活跃请求数
最快响应时间算法
路由与分组
广播与组播
集群
failfast
failover
广播和服务发现
处理广播结果
组播
可用性
AOP 方案
故障检测
令牌桶算法
漏桶算法
滑动窗口
固定窗口
基于 Redis 的实现
故障处理
熔断
限流
降级
三者联系与区别
故障恢复
可观测性

AOP 方案：Kratos

```go
// Handler defines the handler invoked by Middleware.
type Handler func(ctx context.Context, req interface{}) (interfa

// Middleware is HTTP/gRPC transport middleware.
type Middleware func(next Handler) Handler

// Chain returns a Middleware that specifies the chained handler
func Chain(m ...Middleware) Middleware {
    return func(next Handler) Handler {
        for i := len(m) - 1; i >= 0; i-- {
            next = m[i](next)
        }
        return next
    }
}
```

我们在 Web 框架和 ORM 框架中都是采用类似的设计。

AOP 方案：Dubbo-go

```go
// Filter interface defines the functions of a filter
// Extension - Filter
type Filter interface {
    // Invoke is the core function of a filter, it determines the proce
    Invoke(context.Context, protocol.Invoker, protocol.Invocation) prot
    // OnResponse updates the results from Invoke and then returns the
    OnResponse(context.Context, protocol.Result, protocol.Invoker, prot
}
```

Dubbo-go 的设计更加接近责任链的设计。其中它有一个回调接口 OnResponse，即响应返回的时候会执行这个方法。

AOP 方案：go-micro

go-micro 叫做 Wrapper，也就是认为自己是在已有功能的基础上再封装一些功能，（我猜）设计者认为它更加接近装饰器模式或者洋葱模式。

客户端 Wrapper 分成三类：

- CallWrapper：最细粒度控制，针对每一次调用
- Wrapper：针对客户端的 Wrapper
- SteamWrapper：针对 Stream API 的 Wrapper

实际上我认为 Wrapper 和 CallWrapper 只需要保留一个就可以。通过注册过程来控制作用范围，例如我们在某个调用里面注册 Wrapper，那么只对该调用起效果。

```go
// CallFunc represents the individual call func
type CallFunc func(ctx context.Context, node *registry.Node,

// CallWrapper is a low level wrapper for the CallFunc
type CallWrapper func(CallFunc) CallFunc

// Wrapper wraps a client and returns a client
type Wrapper func(Client) Client

// StreamWrapper wraps a Stream and returns the equivalent
type StreamWrapper func(Stream) Stream
```

AOP 方案：go-micro

服务端的 Wrapper 定义不太一样，核心就是 HandlerWrapper 和 StreamWrapper，其作用接近于客户端的 Wrapper 和 StreamWrapper。

从个人偏好上来说，我既不喜欢客户端和服务端设计不同 API 的做法，也不喜欢普通请求和 Stream 请求分别对待的做法。

个人比较喜欢 Dubbo-go 那种一致的抽象。

```go
// HandlerFunc represents a single method of a handler. It's used pr
// for the wrappers. What's handed to the actual method is the concr
// request and response types.
type HandlerFunc func(ctx context.Context, req Request, rsp interfac

// SubscriberFunc represents a single method of a subscriber. It's u
// for the wrappers. What's handed to the actual method is the concr
// publication message.
type SubscriberFunc func(ctx context.Context, msg Message) error

// HandlerWrapper wraps the HandlerFunc and returns the equivalent
type HandlerWrapper func(HandlerFunc) HandlerFunc

// SubscriberWrapper wraps the SubscriberFunc and returns the equiva
type SubscriberWrapper func(SubscriberFunc) SubscriberFunc

// StreamWrapper wraps a Stream interface and returns the equivalent
// Because streams exist for the lifetime of a method invocation thi
// is a convenient way to wrap a Stream as its in use for trace, mor
// metrics, etc.
type StreamWrapper func(Stream) Stream
```

可用性

在微服务框架中，可用性是和服务治理最密切相关的主题：

- 熔断
- 限流
- 降级
- 重试：这个主题我们已经在 cluster 里面接触过了
- 超时控制：我们在 RPC 设计的时候也接触过了

可用性：Kratos

限流接口，会返回一个回调，这个回调会在请求结束之后执行，也就是 DoneFunc。

```
// Limiter is a rate limiter.
type Limiter interface {
    Allow() (DoneFunc, error)
}
```

```
}
return func(handler middleware.Handler) middleware.Handler {
    return func(ctx context.Context, req interface{}) (reply int
        done, e := options.limiter.Allow()
        if e != nil {
            // rejected
            return reply: nil, ErrLimitExceed
        }
        // allowed
        reply, err = handler(ctx, req)
        done(ratelimit.DoneInfo{Err: err})
        return
    }
}
```

可用性：Kratos

熔断接口。

注意，在限流和熔断里面，其实 API 很像，都是一个 Allow 方法。

```
return func(handler middleware.Handler) middleware.Handl
    return func(ctx context.Context, req interface{}) (
        info, _ := transport.FromServerContext(ctx)
        breaker := opt.group.Get(info.Operation()).(cir
        if err := breaker.Allow(); err != nil {
            // rejected
            // NOTE: when client reject requets locally
            // continue add counter let the drop ratio
            breaker.MarkFailed()
            return nil, ErrNotAllowed
        }
        // allowed
        reply, err := handler(ctx, req)
        if err != nil && (errors.IsInternalServer(err)
            breaker.MarkFailed()
```

可用性：Dubb-go 限流

限流的核心接口是 TpsLimiter。

```
 * The usage, for example:
 * "UserProvider":
 *   registry: "hangzhouzk"
 *   protocol : "dubbo"
 *   interface : "com.ikurento.user.UserProvider"
 *   ... # other configuration
 *   tps.limiter: "the name of limiter",
 */
type TpsLimiter interface {
    // IsAllowable will check whether this invocation shoul
    IsAllowable(*common.URL, protocol.Invocation) bool
}
```

可用性：Dubb-go 接入 Hystrix

Dubbo-go 主要依赖于 Hystrix 来做熔断。早几年的时候 Hystrix 很流行，目前来看，已经不是很流行了。

核心是利用 Hystrix 的 Do 方法，传入一个方法。这个方法要么被执行，要么被 Hystrix 触发熔断。

```
logger.Infof( fmt: "[Hystrix Filter]Using hystrix filter: %s",
var result protocol.Result
_ = hystrix.Do(cmdName, func() error {
    result = invoker.Invoke(ctx, invocation)
    err := result.Error()
    if err != nil {
        result.SetError(NewHystrixFilterError(err, failByHyst
        for _, reg := range f.res[invocation.MethodName()] {
            if reg.MatchString(err.Error()) {
                logger.Debugf( fmt: "[Hystrix Filter]Error in
                return nil
            }
        }
    }
    return err
}, func(err error) error {
```

可用性：go-micro 限流

go-micro 支持客户端限流和服务端限流。

它的 plugins 仓库提供的限流利用了 github.com/juju/ratelimit 库，采用的就是令牌桶算法。

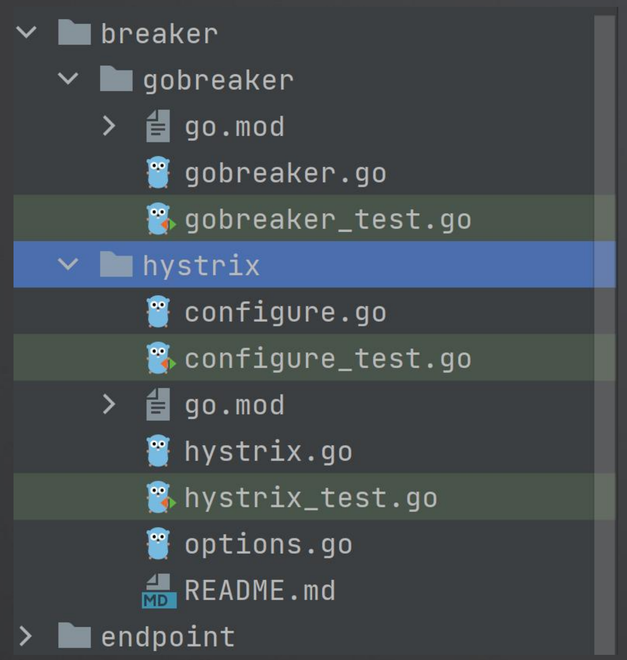
```
// Bucket represents a token bucket that fills
// Methods on Bucket may be called concurrentl
type Bucket struct {
    clock Clock

    // startTime holds the moment when the buc
    // first created and ticks began.
    startTime time.Time

    // capacity holds the overall capacity of
    capacity int64
```

可用性：go-micro 熔断

go-micro 的熔断，默认的实现是支持了本地以及基于 hystrix 两种。其中 gobreaker 利用的是 github.com/sony/breaker 的实现。



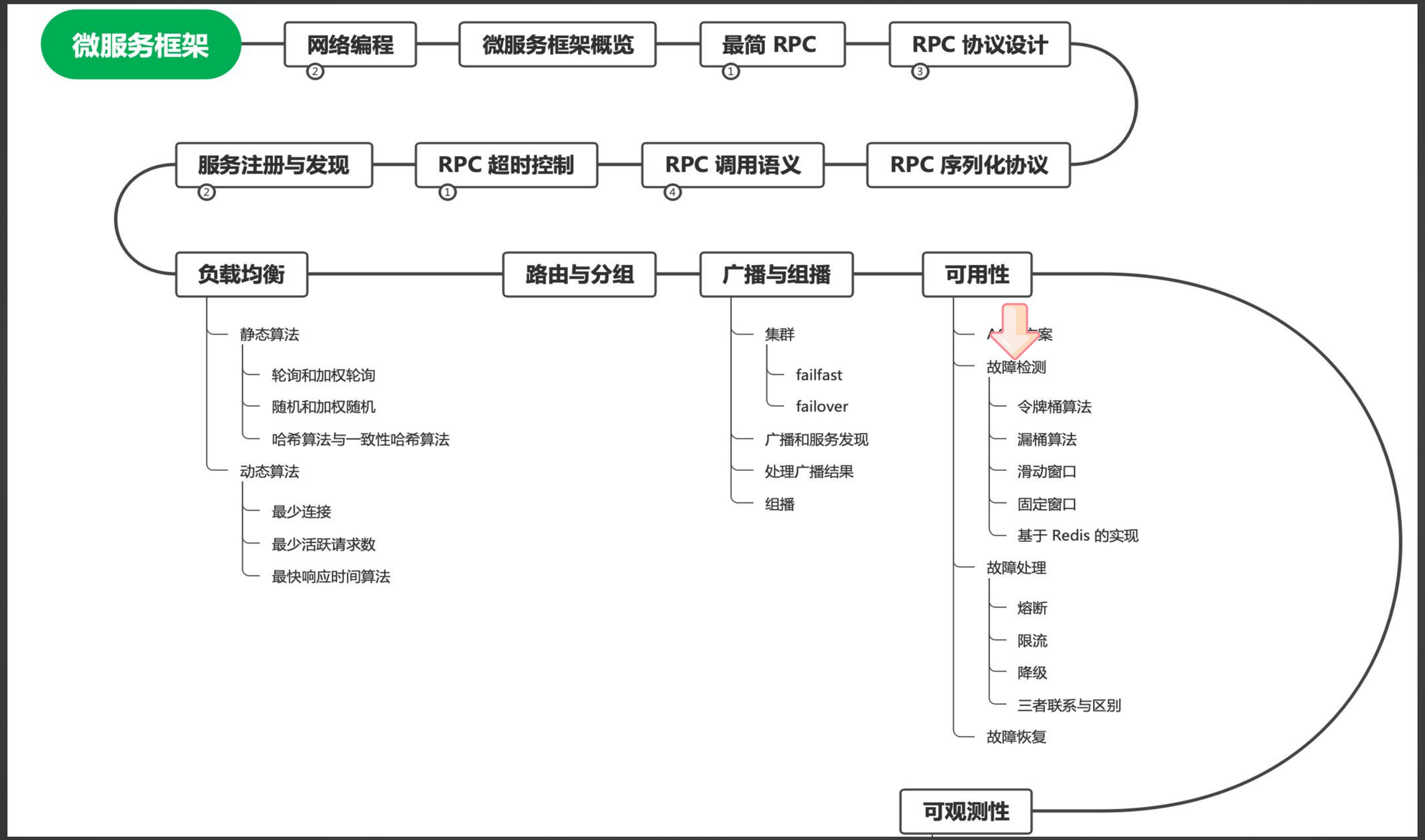

```
c.mu.Lock()
cb, ok := c.cbs[svc]
if !ok {
    cb = gobreaker.NewTwoStepCircuitBreaker(c.bs)
    c.cbs[svc] = cb
}
c.mu.Unlock()

cbAllow, err := cb.Allow()
if err != nil {
    return errors.New(req.Service(), err.Error(), code:
}

if err = c.Client.Call(ctx, req, rsp, opts...); err ==
    cbAllow( success: true)
    return nil
```

几乎和限流设计一样

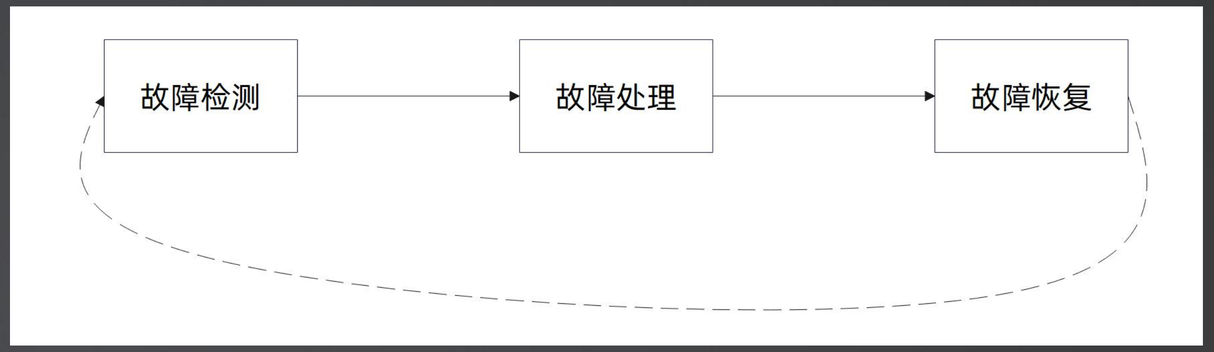
学习路线


微服务框架
网络编程
2
微服务框架概览
最简 RPC
1
RPC 协议设计
3
RPC 序列化协议
RPC 调用语义
4
RPC 超时控制
1
服务注册与发现
2
负载均衡
静态算法
轮询和加权轮询
随机和加权随机
哈希算法与一致性哈希算法
动态算法
最少连接
最少活跃请求数
最快响应时间算法
路由与分组
广播与组播
集群
failfast
failover
广播和服务发现
处理广播结果
组播
可用性
故障检测
令牌桶算法
漏桶算法
滑动窗口
固定窗口
基于 Redis 的实现
故障处理
熔断
限流
降级
三者联系与区别
故障恢复
可观测性

可用性：熔断、限流、降级

熔断、限流和降级并没有本质区别，都可以归属到故障处理的范畴里面：

- 故障检测：使用一些特定的算法，判定服务是否处于不健康状态。
- 故障处理
  - 限流：一段时间内只允许特定数量的请求被处理。
  - 熔断：全部请求都会被拒绝。
  - 降级：全部请求都会执行一段更加简单的逻辑。
- 故障恢复：即如果服务被判定为处于异常状态之后，什么时候再恢复过来。



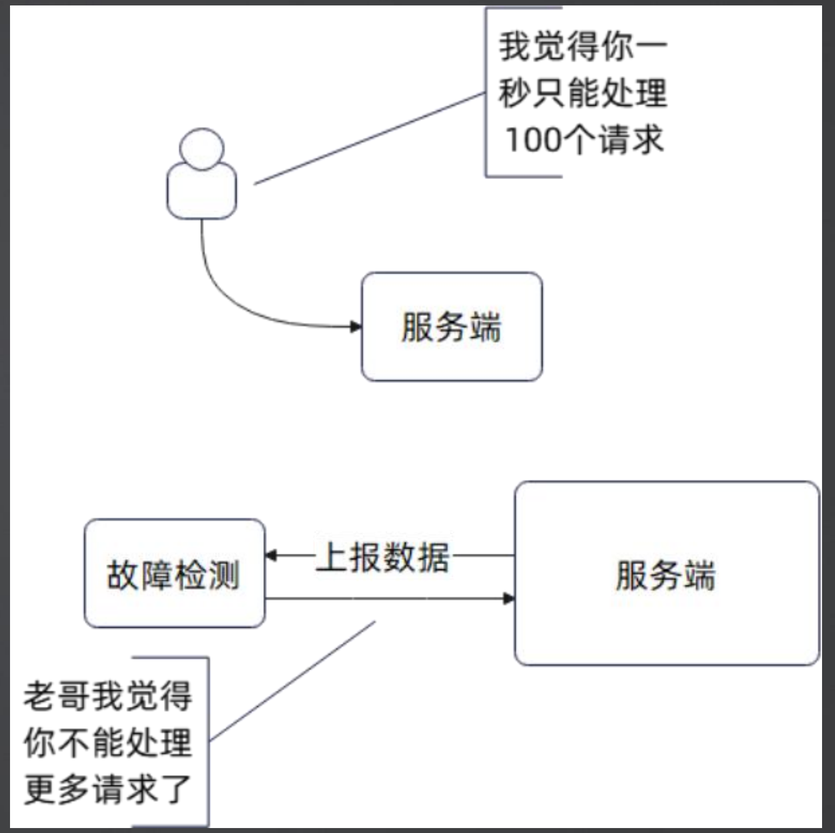
对于一个服务来说，它就是在这几个状态中不断变迁。

可用性：故障检测

从故障检测算法的类型来看，可以分成：

- 静态类型的算法：例如限流算法设置为一秒内只能执行一百个请求，这一类就是静态的，依赖于测试或者程序员根据个人经验提前设置好。令牌桶、漏桶、固定窗口和滑动窗口都属于这一类。
- 动态类型的算法：根据服务当前状态动态判断，例如根据错误率、响应时间，或者典型的 BBR 算法。

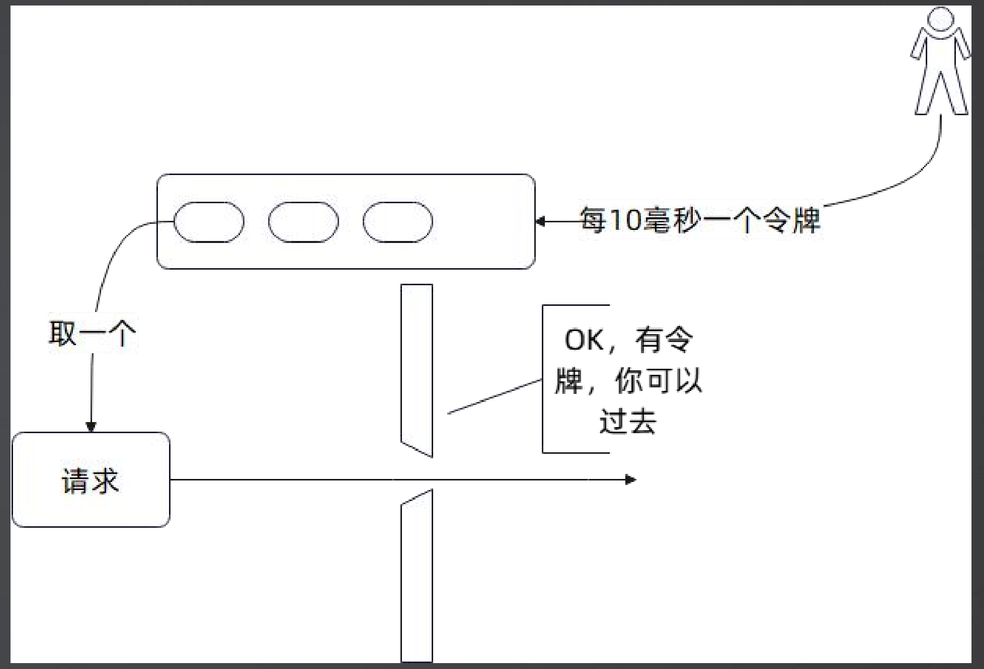
对于绝大多数应用来说，静态类型算法就足够了。



限流算法：令牌桶

令牌桶算法要点：

- 有一个人按一定的速率发令牌
- 令牌会被放到一个桶里
- 每一个请求从桶里面拿一个令牌
- 拿到令牌的请求就会被处理
- 没有拿到令牌的请求就会：
  - 直接被拒绝
  - 阻塞直到拿到令牌或者超时



限流算法：令牌桶

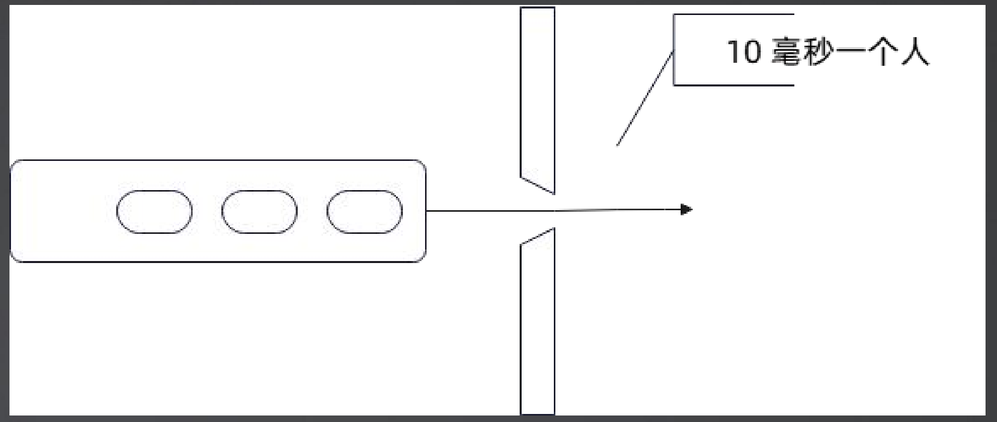
```
go func() {
    producer := time.NewTicker(interval)
    defer producer.Stop()
    for {
        select {
        case <-res.close:
            // 关闭
            return
        case <-producer.C:
            select {
            //case <- res.close:
            // 关闭。在这里其实可以没有这个分支
            //return
            case res.tokens <- struct{}{}:
            default:
                // 加 default 分支防止一直没有人取令牌，我们这
            }
        }
```

```
return func(ctx context.Context, req interface{}, in
    select {
    case <-ctx.Done():
        return resp: nil, ctx.Err()
    case <-l.close:
        // 已经关闭了
        // 这里你可以决策，如果认为限流器被关了，就代表不用限
        // 这种情况下，还要考虑提供 Start 方法重启限流器
        // 我这里采用另外一种语义，就是我认为限流器被关了，其
        return resp: nil, errors.New(text: "micro: 系
    case <-l.tokens:
        return handler(ctx, req)
    }
}
```

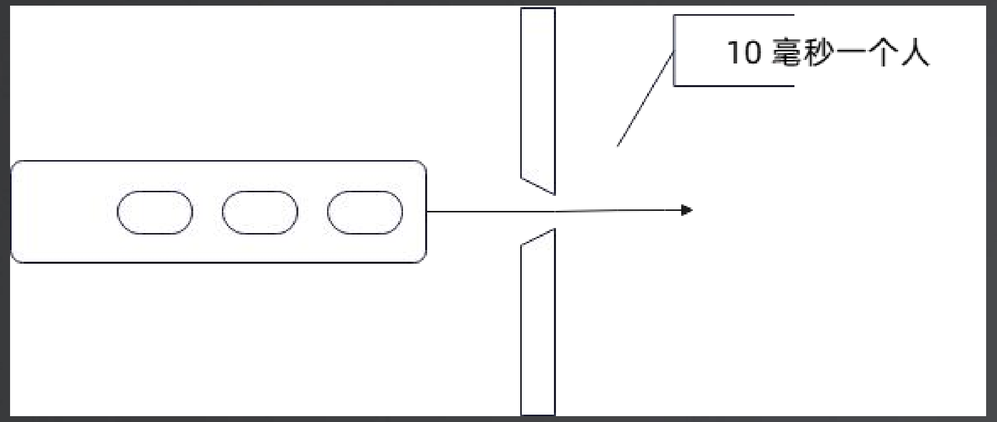
限流算法：漏桶

漏桶算法要点：

- 请求过来先排队
- 每隔一段时间，放过去一个请求
- 请求排队直到通过，或者超时



限流算法：漏桶




```go
func (l *LeakyBucketLimiter) LimitUnary() grpc.UnaryS
    return func(ctx context.Context, req interface{},
        select {
        case <-ctx.Done():
            // 等令牌过期了
            return resp: nil, ctx.Err()
        case <-l.producer.C:
            return handler(ctx, req)
        }
    }
}
```

限流算法：漏桶与令牌桶

漏桶算法要点：

- 请求过来先排队
- 每隔一段时间，放过去一个请求
- 请求排队直到通过，或者超时

令牌桶算法要点：

- 有一个人按找一定的速率发令牌
- 令牌会被放到一个桶里，这个桶只能放固定数量的令牌
- 每一个请求从桶里面拿一个令牌
- 拿到令牌的请求就会被处理
- 没有拿到令牌的请求就会：
  - 直接被拒绝
  - 阻塞直到拿到令牌或者超时

两者效果是一样的。

限流算法：固定窗口

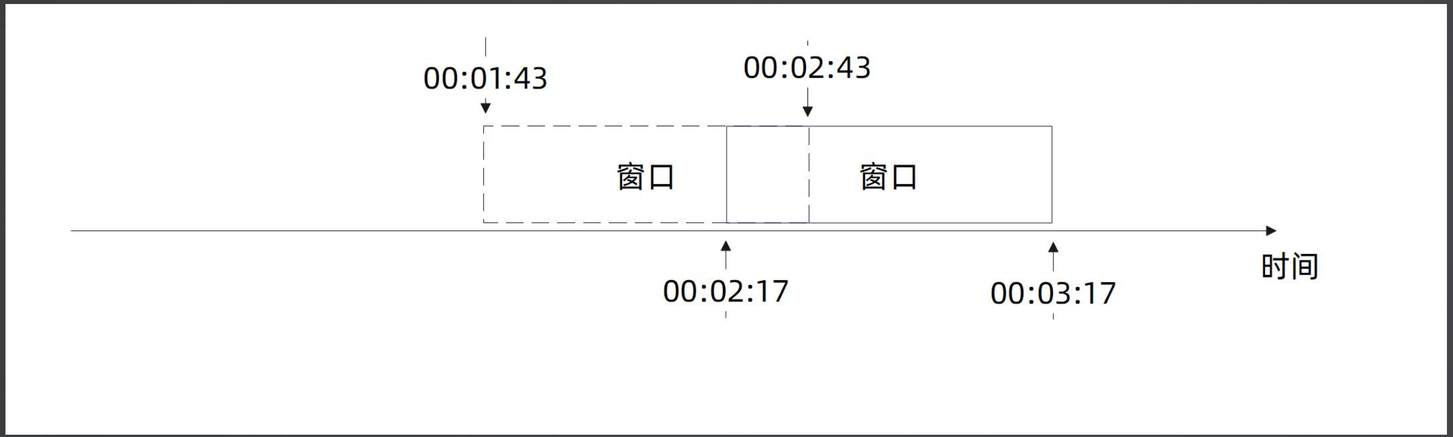
限流算法：固定窗口

```
nc (t *FixWindowLimiter) BuildServerInterceptor() grpc.UnaryServerInterceptor
  return func(ctx context.Context, req interface{}, info *grpc.UnaryServerInf

      // 考虑 t.cnt 重置的问题
      current := time.Now().UnixNano()
      timestamp := atomic.LoadInt64(&t.timestamp)
      cnt := atomic.LoadInt64(&t.cnt)
      if timestamp + t.interval < current {
          // 这意味着这是一个新窗口
          // 重置窗口
          if atomic.CompareAndSwapInt64(&t.timestamp, timestamp, current) {
              //atomic.StoreInt64(&t.cnt, 0)
              atomic.CompareAndSwapInt64(&t.cnt, cnt, new: 0)
          }
      }
      cnt = atomic.AddInt64(&t.cnt, delta: 1)
      if cnt > t.rate {
          err = errors.New( text: "触发瓶颈了")
          return
      }
      resp, err = handler(ctx, req)
      return
  }
```

限流算法：滑动窗口

从当前时间开始，往前回溯一段时间，只能处理一定数量的请求。



滑动窗口的核心是：这个窗口永远是以当前时间戳为准，往前回溯。例如当前时间点是 00:03:17，往前回溯一分钟，就是一个一分钟长度的窗口。

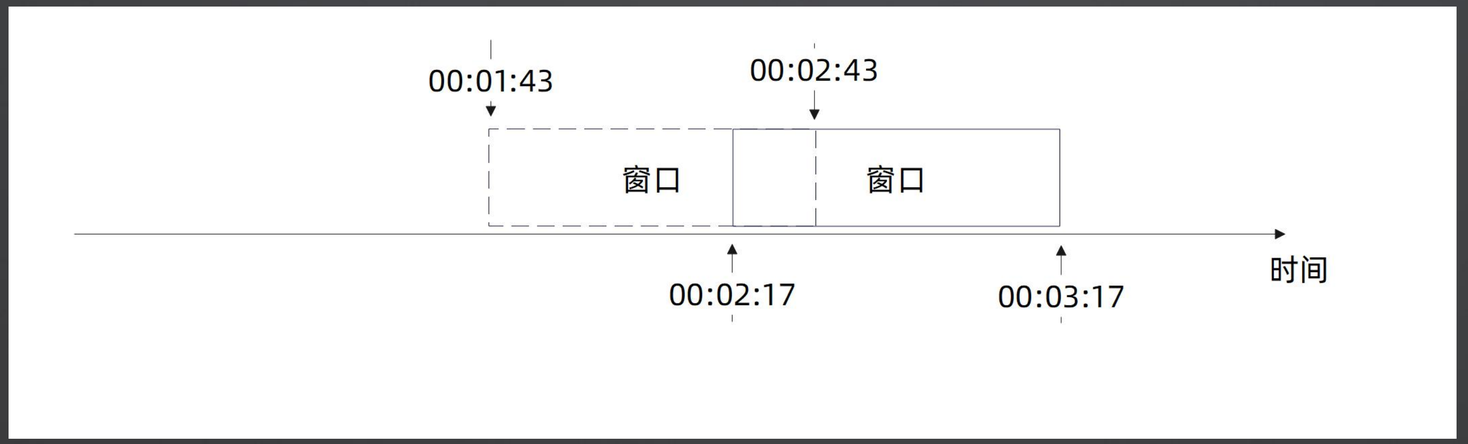
限流算法：滑动窗口

```go
func (l *SlideWindowLimiter) limit() bool {
    l.mutex.Lock()
    defer l.mutex.Unlock()
    // 快路径
    size := l.queue.Len()
    current := time.Now().UnixNano()
    if size < l.rate {
        l.queue.PushBack(current)
        return false
    }
    // 慢路径
```

```go
// 慢路径
boundary := current - l.interval
timestamp := l.queue.Front()
// 删除已经不在窗口内的元素
for timestamp != nil && timestamp.Value.(int64) < boundary {
    l.queue.Remove(timestamp)
    timestamp = l.queue.Front()
}
if l.queue.Len() < l.rate {
    l.queue.PushBack(current)
    return false
}
return true
```

限流算法：两种窗口对比

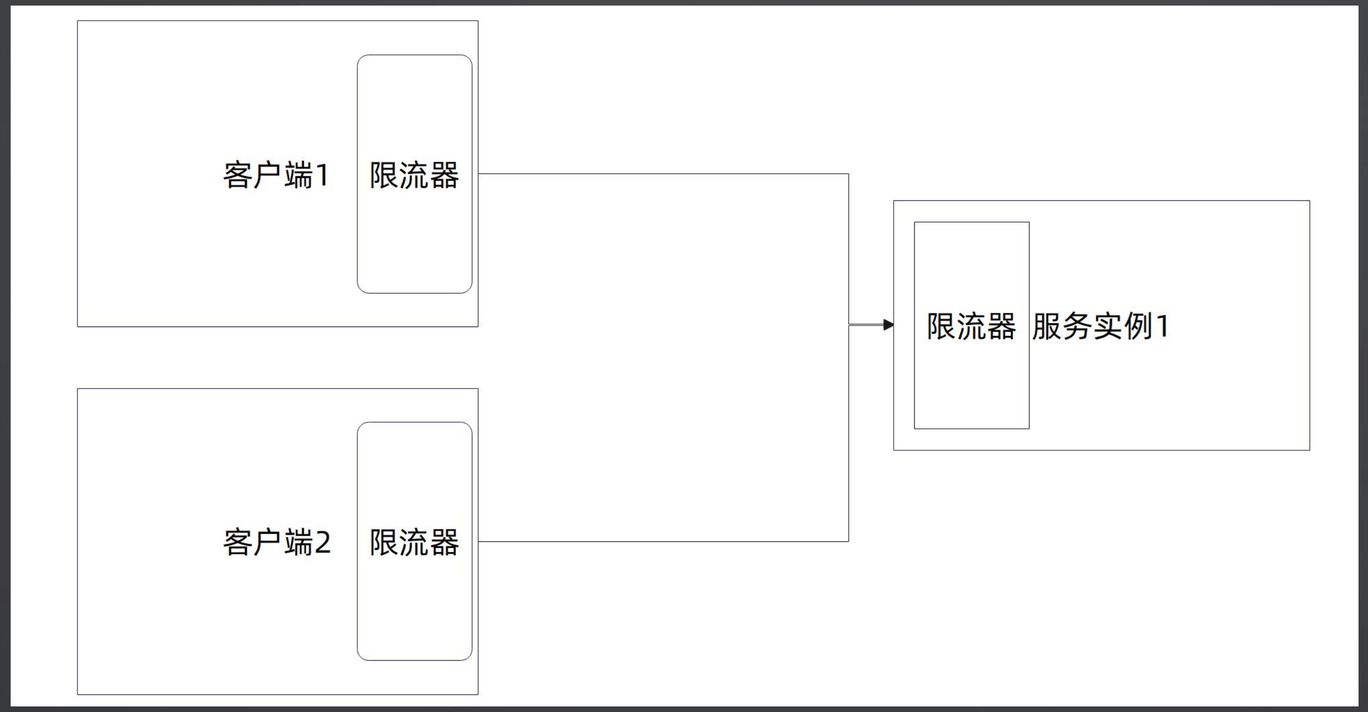

窗口1
窗口2
窗口3
00:01:00
00:02:00
00:03:00
时间


00:01:43
00:02:43
窗口
窗口
时间
00:02:17
00:03:17

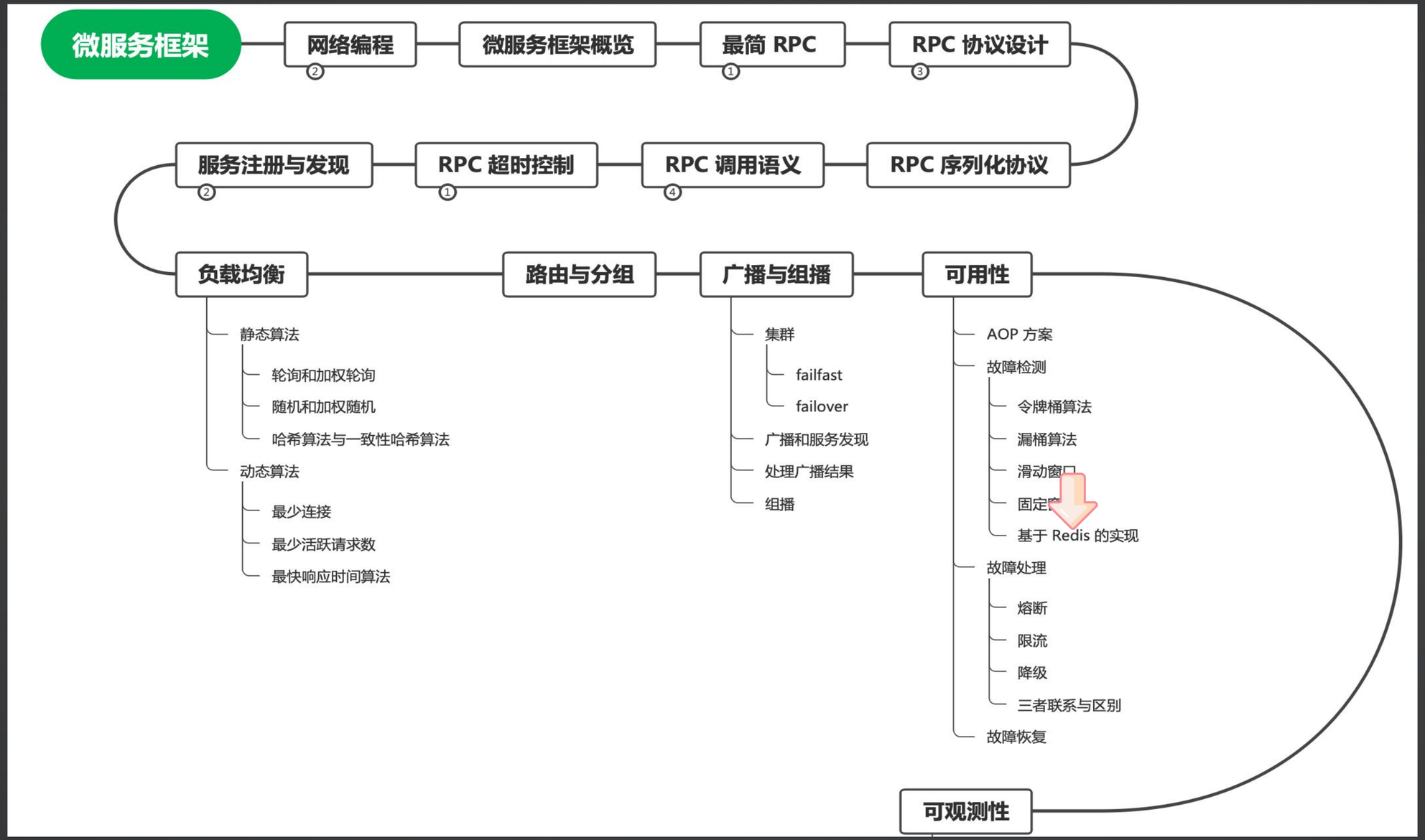
客户端限流与服务端限流

前面代码演示的都是服务端限流——利用的是 gRPC 的服务端拦截器。

从理论上来说，还可以考虑客户端限流。客户端限流用得少，主要是因为很可能不同客户端上单独限流了，结果合在一起却超过了服务器处理能力。



学习路线

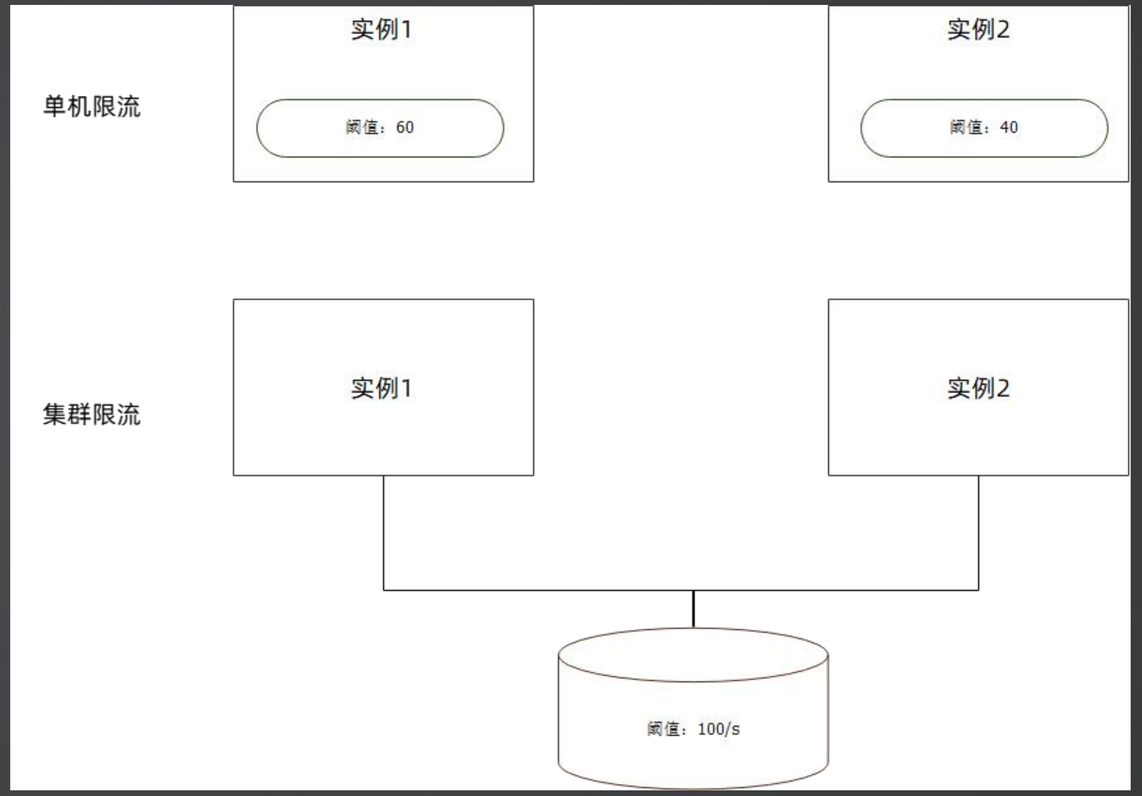

微服务框架
网络编程
2
微服务框架概览
最简 RPC
1
RPC 协议设计
3
服务注册与发现
2
RPC 超时控制
1
RPC 调用语义
4
RPC 序列化协议
负载均衡
静态算法
轮询和加权轮询
随机和加权随机
哈希算法与一致性哈希算法
动态算法
最少连接
最少活跃请求数
最快响应时间算法
路由与分组
广播与组播
集群
failfast
failover
广播和服务发现
处理广播结果
组播
可用性
AOP 方案
故障检测
令牌桶算法
漏桶算法
滑动窗口
固定窗
基于 Redis 的实现
故障处理
熔断
限流
降级
三者联系与区别
故障恢复
可观测性

单机限流与集群限流

前面我们讨论的都是单机限流，在微服务框架下我们还可以考虑对集群进行限流。

集群限流特征：

- 非常接近网关限流。
- 集群限流主要依赖于在不同的实例之间同步阈值、当前请求数，目前使用 Redis 的比较多（高并发）。



基于 Redis 的实现：固定窗口

```lua
local val = redis.call('get', KEYS[1])
local limit = tonumber(ARGV[2])
if val == false then
    if limit < 1 then
        -- 执行限流
        return "true"
    else
        -- key 不存在，设置初始值 1，并且设置过期时间
        redis.call('set', KEYS[1], 1, 'PX', ARGV[1])
        -- 不执行限流
        return "false"
    end
elseif tonumber(val) < limit then
    -- 自增 1
    redis.call('incr', KEYS[1])
    -- 不需要限流
    return "false"
else
    -- 限流
    return "true"
end
```

基于 Redis 的实现：滑动窗口

```lua
-- 先挪动窗口
redis.call('ZREMRANGEBYSCORE', key, '-inf', min)

-- 看看还有没有容量
local cnt = redis.call('ZCOUNT', key, '-inf', '+inf')

if cnt >= threshold then
    -- 限流
    return "true"
else
    -- 值和优先级我们都设置成当前时间戳
    redis.call('ZADD', key, now, now)
    -- 这里设不设置过期时间影响不大，设置了过期时间可以防止长期没有
    redis.call('PEXPIRE', key, window)
    -- 不限流
    return "false"
end
```

本质上，可以理解为我们把刚才的算法用 lua 脚本再写了一遍。

限流总结

限流对象，即针对什么限流，除了前面的单机限流和集群限流，还有：

- 接口维度限流
- 方法维度限流
- 业务相关限流：
  - 例如针对用户限流。
  - 针对 IP 限流：常见的针对登录请求，同一个 IP 不能频繁登录。

业务相关一般针对安全、业务价值。大体上的逻辑就是牺牲不重要的来保护重要的。

限流也就可以考虑跨服务限流，即如果服务很重要，那么可以将不重要的服务流量摘掉，或者丢弃一部分，腾出资源来保护核心服务。

拒绝策略

在我们的课堂里面，目前所有的限流都默认返回了错误，在实际中我们可以考虑：

- 标记一下这个请求是限流请求，后续业务走简易路径。例如结合我们前面讲的缓存模式，这种被标记的请求，我们在缓存未命中的时候就直接返回，不会再去数据库里面加载。
- 直接返回一个固定的响应：和上一种措施比起来，这种直接在拦截器里面就返回了。
- 缓存请求：前面我们的令牌桶和漏桶阻塞住了，在某种程度上也可以看做是缓存。这里面的缓存是指我们将请求丢到一个数据库或者 Redis 之类的地方存好，后面再取出来重做。
- 转为异步模式：拦截器将请求标记为限流请求。那么后面的业务就会直接返回一个类似 202 的响应，即请求已经接收。后续再由一个调度器重新调度它来真正执行。和前者比起来，它不是由拦截器来缓存请求。
- 转发到别的服务器：这种设计更加罕见，也就是限流的请求，可以被转发到别的机器上，但是对于微服务框架来说，这意味着在服务端也要像客户端那样发现服务实例，并且判定服务实例的性能。一种简化设计就是结合前面的 failover，返回一个类似于 302 重定向的错误，让客户端重新发请求到别的节点。

拒绝策略：简易路径

后续的中间件可以检测这种标记位。我们只需要在前面触发限流的时候设置这个标记位就可以。

在业务中，你们要仔细甄别这一类的场景，在面试的时候你们则可以说：

- 业务分成了快路径和慢路径两种，其中慢路径很消耗资源。
- 在没有触发限流的时候，先走快路径，不行则走慢路径。
- 在触发限流的时候，则是只走快路径。

```
var markLimitedRejection rejectStrategy = func(ctx context.Context, inf
    req interface{}, handler grpc.UnaryHandler) (interface{}, error) {
    ctx = context.WithValue(ctx, key: "limiter", val: true)
    return handler(ctx, req)
}
```

```
func (r *RateLimitReadThroughCache) Get(ctx context.Context, key str
    val, err := r.Cache.Get(ctx, key)
    if err == KeyNotFound && ctx.Value(key: "limited") == nil {
        val, err = r.LoadFunc(ctx, key)
        if err == nil {
            if er := r.Set(ctx, key, val, time.Minute); er != nil {
                log.Fatalf(format: "刷新缓存失败, err: %v", err)
            }
        }
    }
    return val, err
}
```

总结：限流、熔断和降级究竟有啥区别？

实际上，它们的区别很少：

- 如果判断到资源不足，只允许一部分请求被正常处理，那么就是限流。而对于那些没有被正常处理的请求来说：
  - 如果请求被直接拒绝，返回错误，那么这部分请求就是被熔断了。
  - 如果请求没有被拒绝，但是返回了默认值，或者走了简易路径，那么就是降级了。
- 从接口设计的角度来说，它们也非常接近，例如定义一个 Allow 就可以了。
- 算法层面上它们也都是通用的。

面试要点

首先要熟练掌握各种限流算法，不仅仅是原理上的，也要能够写出来代码。

- 限流的几个算法？
- 滑动窗口和固定窗口的区别？固定窗口和滑动窗口比起来，滑动窗口的限流更加平滑，因为窗口是在持续移动的。
- 令牌桶和漏桶的区别？
- 什么是降级？其实就是在业务层面上准备快慢两条路径。不降级的时候执行的是慢路径，一般来说它就是正常执行业务逻辑；快路径可以是直接返回默认值，也可以是存储一下数据，后面异步处理。
- 什么是熔断？就是在系统故障（或者快要故障）的时候，拒绝新的请求。这里要注意，如果不是拒绝新的请求，也可以说是降级了。
- 什么是限流？在一定的时间段内，只允许一部分请求被处理，其余请求被拒绝。实际上也可以考虑返回默认值之类的，也就是接近降级的策略。
- 触发降级和熔断之后，怎么恢复过来？常见的做法都是过一段时间之后直接退出降级和熔断状态；高级一点的做法是退出之前先试探一下，放过去少部分请求。
- 熔断、限流和降级三者之间的联系、区别？注意我们课堂上的说法，三者之间其实并不是泾渭分明的，本质上只是处理策略上稍微有点区别。
- 可以针对什么来限流？单机限流、集群限流、业务限流（用户限流、IP 限流）。

Q & A

THANKS