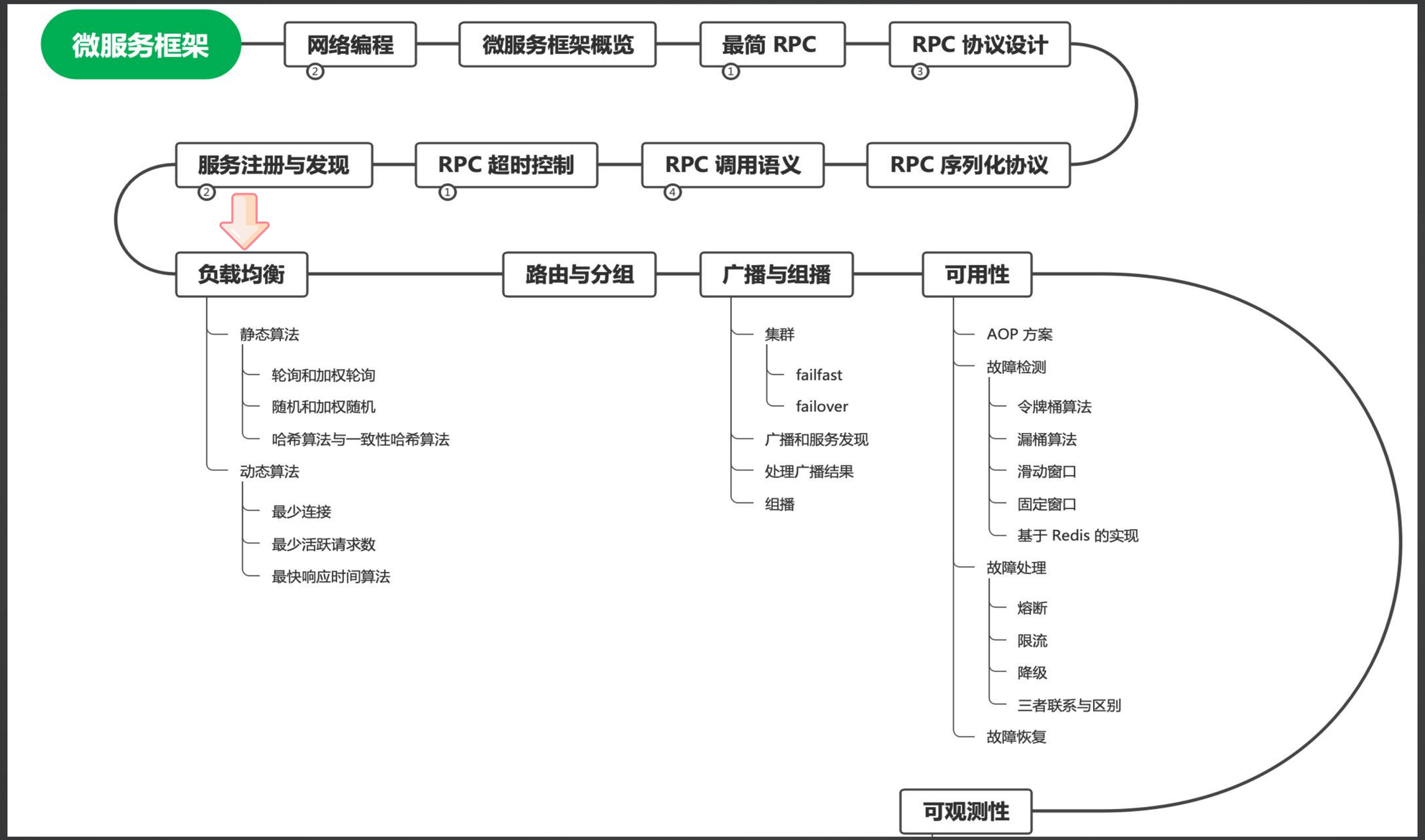
微服务框架—— 负载均衡、路由和集群

大明

学习路线

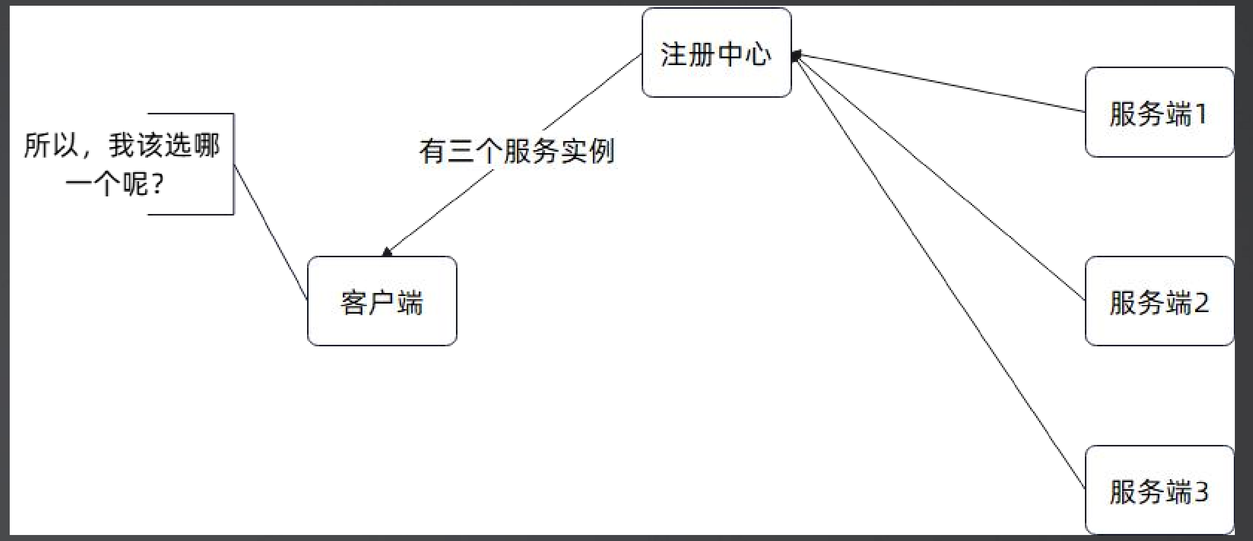

微服务框架
网络编程
2
微服务框架概览
最简 RPC
1
RPC 协议设计
3
服务注册与发现
2
RPC 超时控制
1
RPC 调用语义
4
RPC 序列化协议
负载均衡
静态算法
轮询和加权轮询
随机和加权随机
哈希算法与一致性哈希算法
动态算法
最少连接
最少活跃请求数
最快响应时间算法
路由与分组
广播与组播
集群
failfast
failover
广播和服务发现
处理广播结果
组播
可用性
AOP 方案
故障检测
令牌桶算法
漏桶算法
滑动窗口
固定窗口
基于 Redis 的实现
故障处理
熔断
限流
降级
三者联系与区别
故障恢复
可观测性

负载均衡

前面我们已经讨论了服务注册与发现，也就是已经解决了”有哪些可用服务实例”的问题，现在我们就要讨论下一个问题：

这么多可用服务实例，我该把请求发给谁？

这一步，我们直觉就想到关键字：负载均衡。



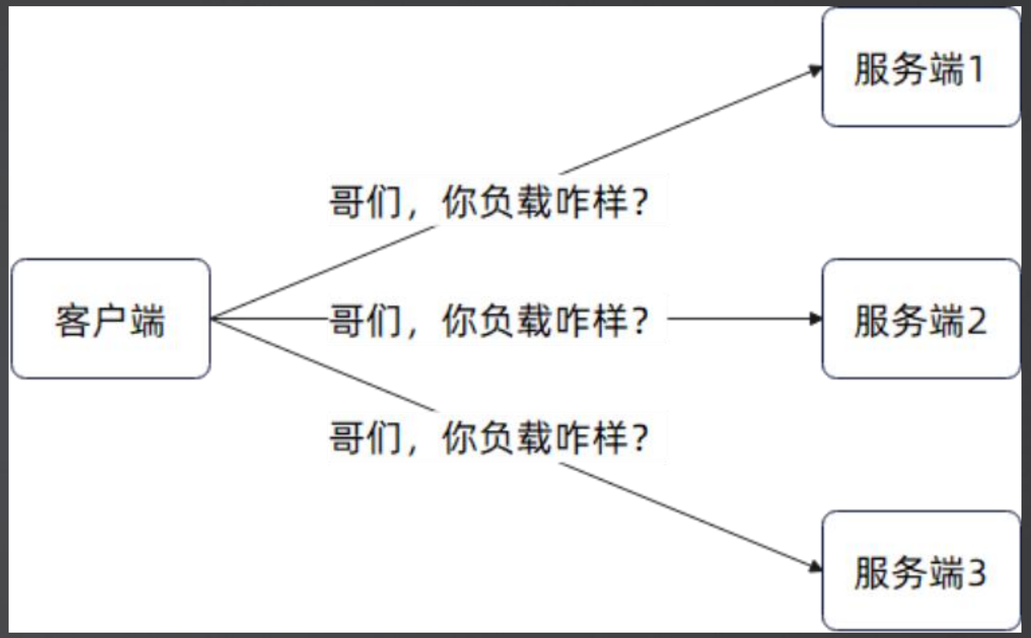
从理论上来说，我们希望将请求发给那个能最快返回响应的实例。

负载均衡算法

因为我们的目标是将请求发给那个能最快返回响应的实例，所以一切算法都是回答这个问题：怎么找出这个实例来？

目前主流的算法有：

- 完全不实时计算负载的算法：轮询、加权轮询、随机、加权随机、哈希、一致性哈希。
- 尝试实时计算负载的算法：最快响应时间、最少连接数、最少请求数算法（也叫最小活跃数）。

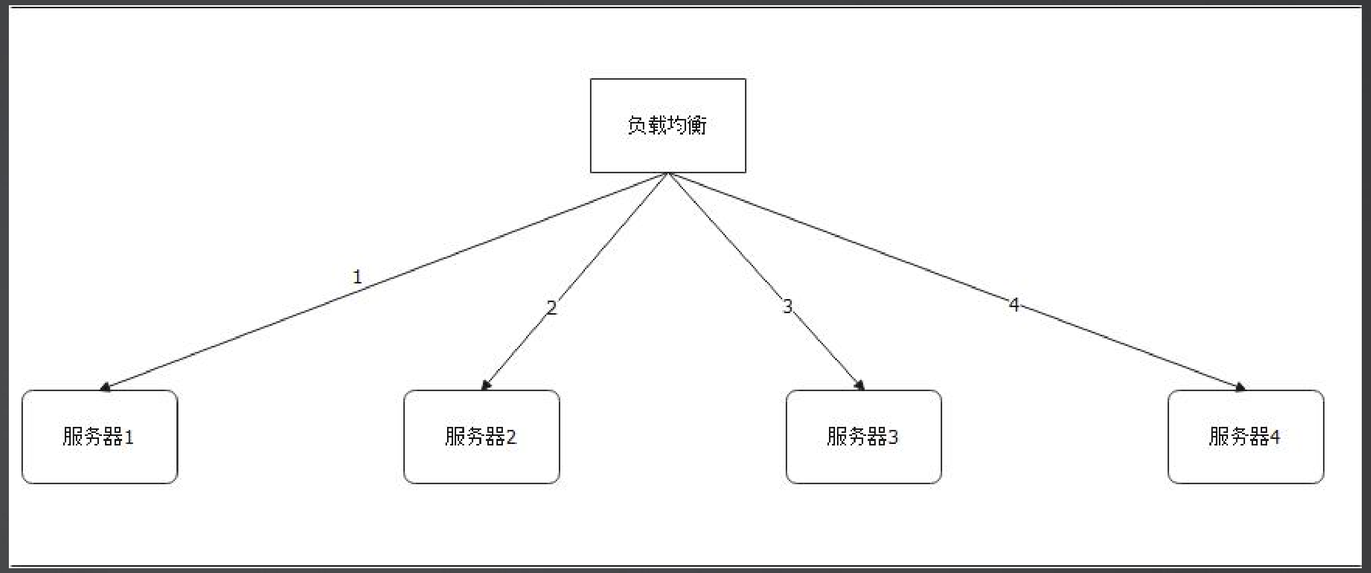


理想情况下，我们希望能够实时问一下各个实例的负载。

负载均衡：轮询

- 排排坐分果果
- 这种负载均衡算法有一些假设：
  - 所有服务器的处理能力是一样的
  - 所有请求所需的资源也是一样的

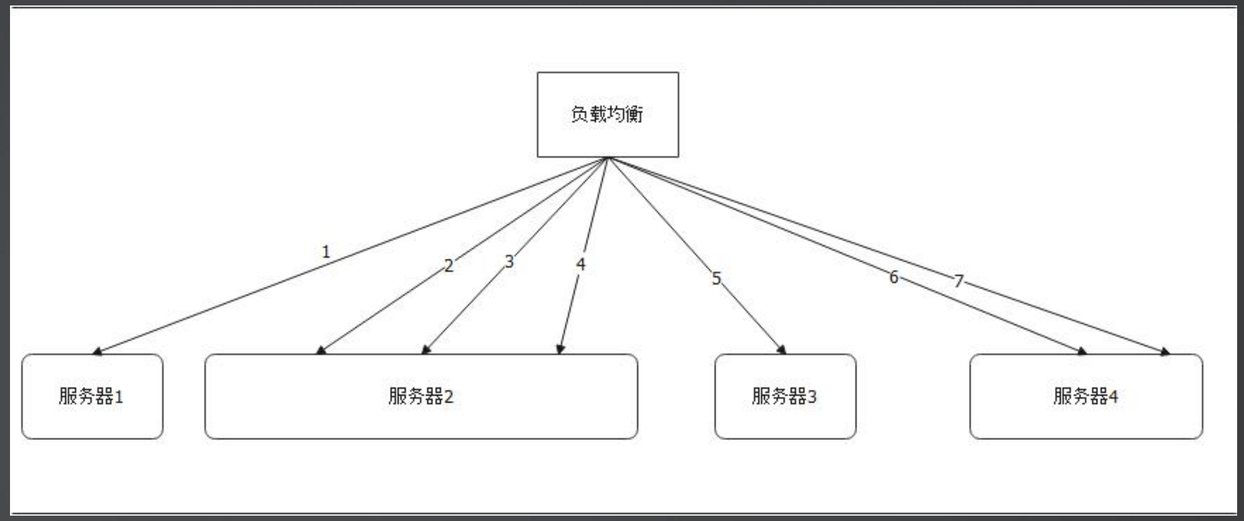
那么，为什么大多数时候它运作效果都很好呢？



负载均衡：加权轮询

也是排排坐分果果，但是按照权重来分。即权重大的多分，权重少的少分。

- 这种负载均衡算法有一些假设：
  - 用权重来代表服务器的处理能力
  - 所有请求所需的资源也是一样的



代码演示：轮询

在 gRPC 里面要自定义负载均衡算法，需要实现两个东西：

- balancer.PickerBuilder
- balancer.Picker

非常类似于注册中心，也是实现两个接口。

不同的是，负载均衡是通过 ServiceConfig 来指定的。

```go
// Picker is used by gRPC to pick a SubConn to send an RPC.
// Balancer is expected to generate a new picker from its snapshot every tim
// internal state has changed.
//
// The pickers used by gRPC can be updated by ClientConn.UpdateState().
type Picker interface {
    /.../
    Pick(info PickInfo) (PickResult, error)
}
```

```go
func (c *Client) Dial(ctx context.Context, service string) (*grpc.
    opts := []grpc.DialOption{grpc.WithResolvers(c.rb)}
    address := fmt.Sprintf( format: "registry:///%s", service)
    if c.insecure {
        opts = append(opts, grpc.WithInsecure())
    }
    if c.balancer != nil {
        opts = append(opts, grpc.WithDefaultServiceConfig(
            fmt.Sprintf( format: `{"LoadBalancingPolicy": "%s"}`,
                c.balancer.Name())))
    }
    return grpc.DialContext(ctx, address, opts...)
}
```

指定负载均衡策略

代码演示：加权轮询

轮询很简单，稍微复杂一点的是加权轮询，这里我们参考 Nginx 所谓的平滑的加权轮询算法。
设想一下，如果有三个实例，权重分别是 3、1、1，那么显然权重为 3 的会连续选中三次。

所谓平滑的加权轮询就是考虑动态调整权重来实现平滑的效果。算法基本原理：

- 每个实例都有三个值：weight（权重）、currentWeight（当前权重）、efficientWeight（有效权重）。
- efficientWeight 会根据调用结果动态调整。
- 每次挑选实例的时候，计算所有实例的 efficientWeight 来作为 totalWeight。
- 对于每一个实例，更新 currentWeight 为 currentWeight + efficientWeight。
- 挑选 currentWeight 最大的那个节点作为最终节点，并且更新它的 currentWeight 为 currentWeight-totalWeight。

代码演示：加权轮询

```go
func (b *WeightBalancer) Pick(info balancer.PickInfo) (ba
    if len(b.conns) == 0 : balancer.PickResult{}, balanc
    var totalWeight uint32
    var res *weightConn

    b.mutex.Lock()
    for _, node := range b.conns {
        totalWeight += node.efficientWeight
        node.currentWeight += node.efficientWeight
        if res == nil || res.currentWeight < node.curren
            res = node
        }
    }
    res.currentWeight -= totalWeight
    b.mutex.Unlock()
    return balancer.PickResult{...}, nil
}
```

```go
Done: func(info balancer.DoneInfo) {
    for {
        weight := atomic.LoadUint32( addr: &(res.efficientWei
        if info.Err != nil && weight == 0 {
            return
        }
        if info.Err == nil && weight == math.MaxUint32 {
            return
        }
        newWeight := weight
        if info.Err == nil {
            newWeight += 1
        } else {
            newWeight -= 1
        }
        if atomic.CompareAndSwapUint32( addr: &(res.efficient
            return
        }
```

Done 里面实现起来才算是恶心，原子操作琐碎，加锁则性能差。另外，调整权重并不是说一定是 +1 -1，要考虑自己权重的实际取值。

如何获取权重？

之前服务注册与发现的时候，我就提到过，ServiceInstance 本身是可以不断增加数据的，例如这一次我们就加上权重数据，而后服务发现将权重数据注入到 Attributes 里面。

一些微服务框架设计得好，会允许用户自定义类似的数据。

```
c (b *WeightBalancerBuilder) Build(info base.PickerBuildInfo) balancer.Picke
conns := make([]*weightConn, 0, len(info.ReadySCs))
var totalWeight uint32 = 0
for con, conInfo := range info.ReadySCs {
    weightStr := conInfo.Address.Attributes.Value(key: "weight").(string)
    // 这里你可以考虑容错，例如服务器没有配置权重，给一个默认的权重
    // 但是我认为这种容错会让用户不经意间出 BUG，所以我这里不会校验，而是直接让它 panic
    // 这是因为 gRPC 确实没有设计 error 作为返回值
    weight, err := strconv.ParseUint(weightStr, base: 10, bitSize: 32)
    if err != nil : err *
    totalWeight += uint32(weight)
    conns = append(conns, &weightConn{
        SubConn: con,
        weight:  uint32(weight),
    })
}
```

如何获取权重？

```go
type Server struct {
    name     string
    listener net.Listener

    si       registry.ServiceInstance
    registry registry.Registry
    // 单个操作的超时时间，一般用于和注册中心打交道
    timeout time.Duration
    *grpc.Server

    weight int
}

func ServerWithWeight(weight int) ServerOption {
    return func(server *Server) {
        server.weight = weight
    }
}
```

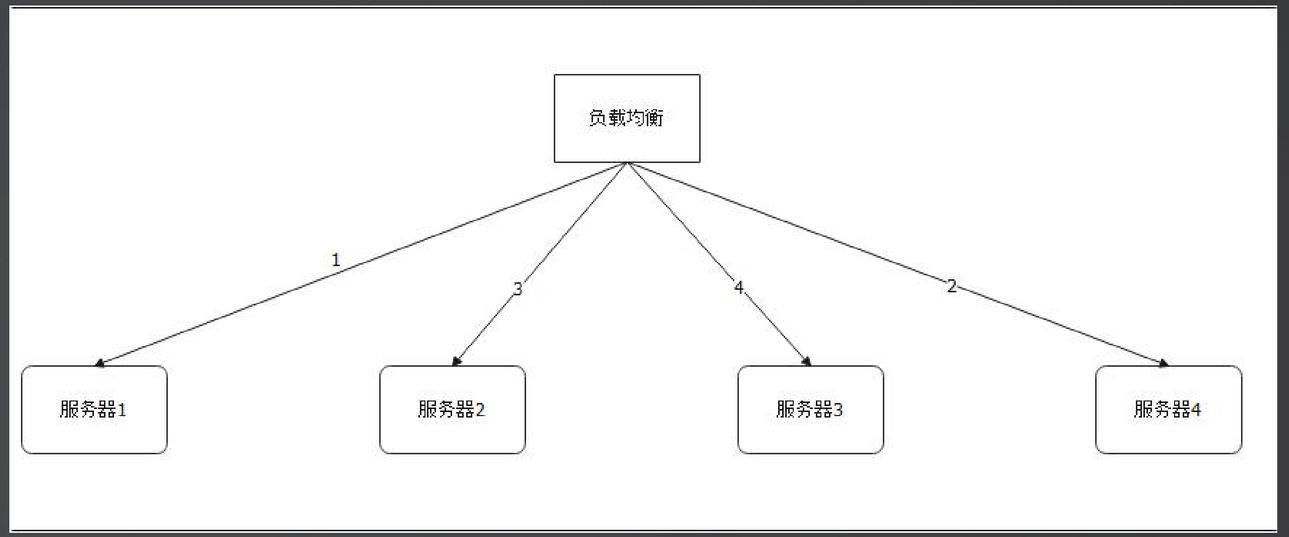
让用户指定

```go
func (g *grpcResolver) resolve() {
    serviceName := g.target.Endpoint
    ctx, cancel := context.WithTimeout(context.Background(), g.timeout
    instances, err := g.registry.ListServices(ctx, serviceName)
    cancel()
    if err != nil {...}

    address := make([]resolver.Address, 0, len(instances))
    for _, ins := range instances {
        address = append(address, resolver.Address{
            Addr:       ins.Address,
            ServerName: ins.Name,
            Attributes: attributes.New(kvs...: "weight", ins.Weight),
        })
    }
    err = g.cc.UpdateState(resolver.State{
        Addresses: address,
```

注入 weight

负载均衡：随机

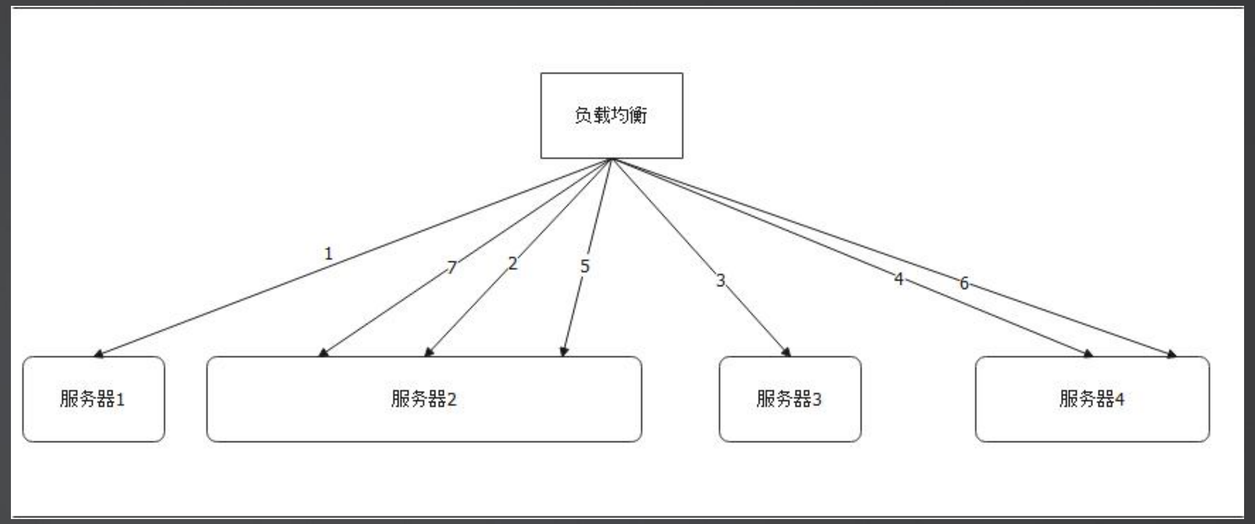
- 闭着眼睛瞎选
- 它在轮询的两个假设上，还有一个假设：
  - 所有服务器的处理能力是一样的
  - 所有请求所需的资源也是一样的
  - 每台服务器被随机到的概率是一样的，因而大量请求的情况下，服务器之间的负载会一样



相比之下，轮询的可控性更强。但是大多数时候可以认为，它们效果差不多。

负载均衡：加权随机

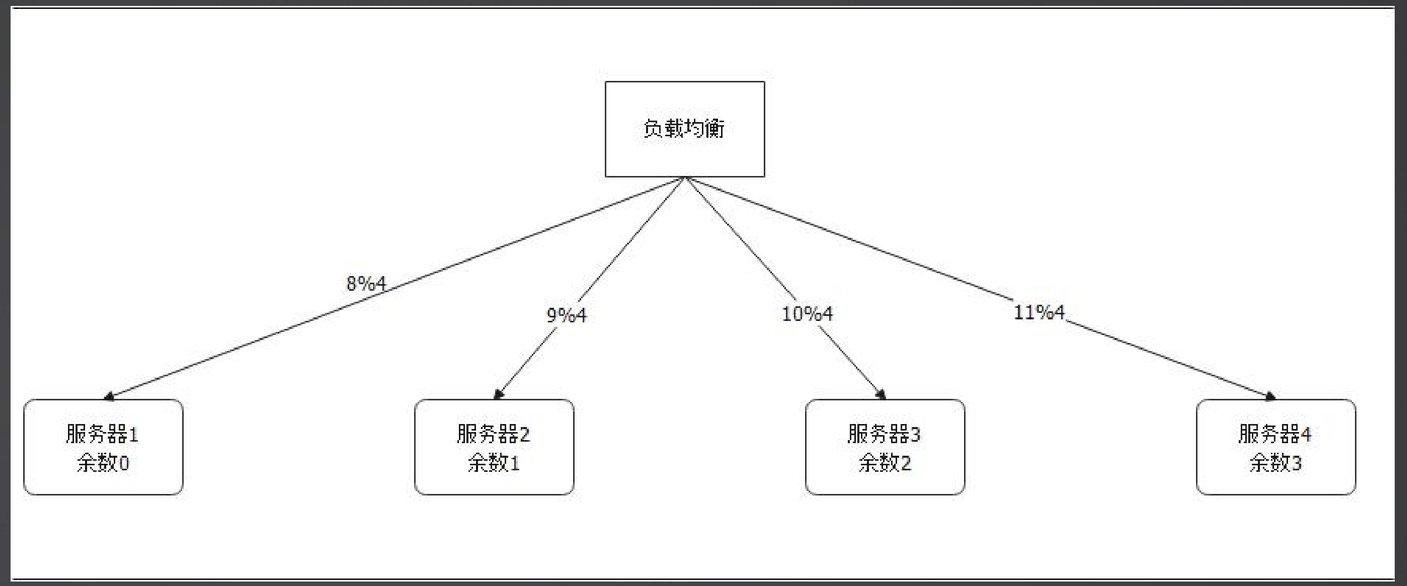
- 根据权重来确定选中概率
- 两个假设：
  - 用权重来代表服务器的处理能力
  - 所有请求所需的资源也是一样的



负载均衡：哈希

- 这种负载均衡算法有一些假设：
  - 所有服务器的处理能力是一样的
  - 所有请求所需的资源也是一样的
  - 哈希值是均匀的

哈希值不均匀会导致请求堆积在一个地方。

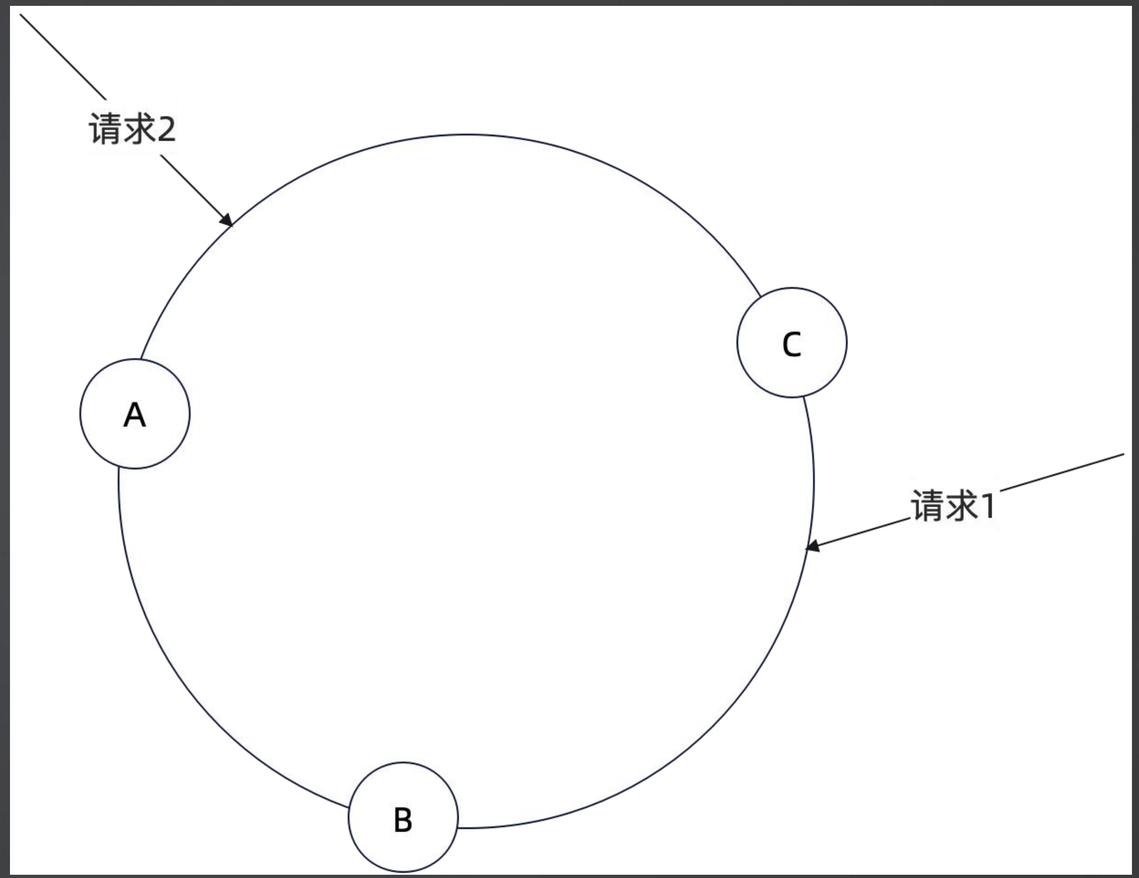


负载均衡：一致性哈希

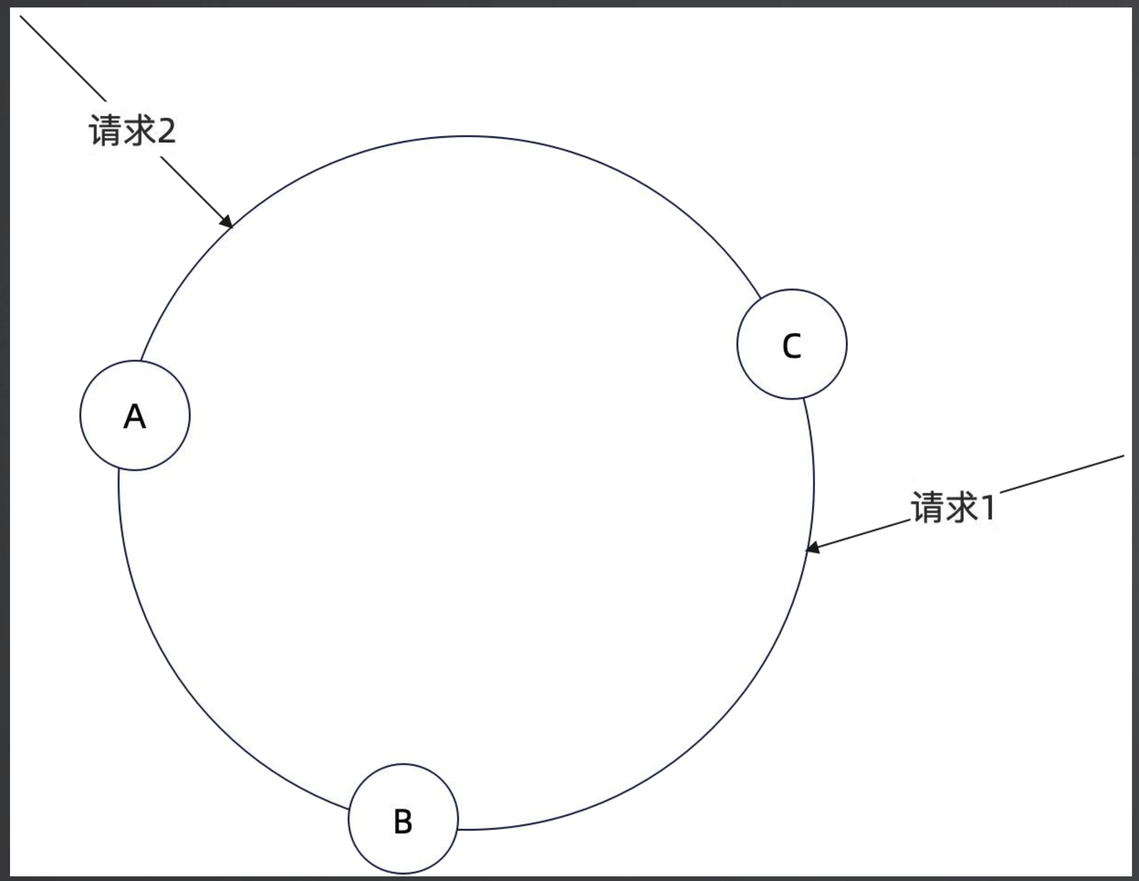
一致性哈希算法是哈希算法的改进，引入了一个类似环状的思路。

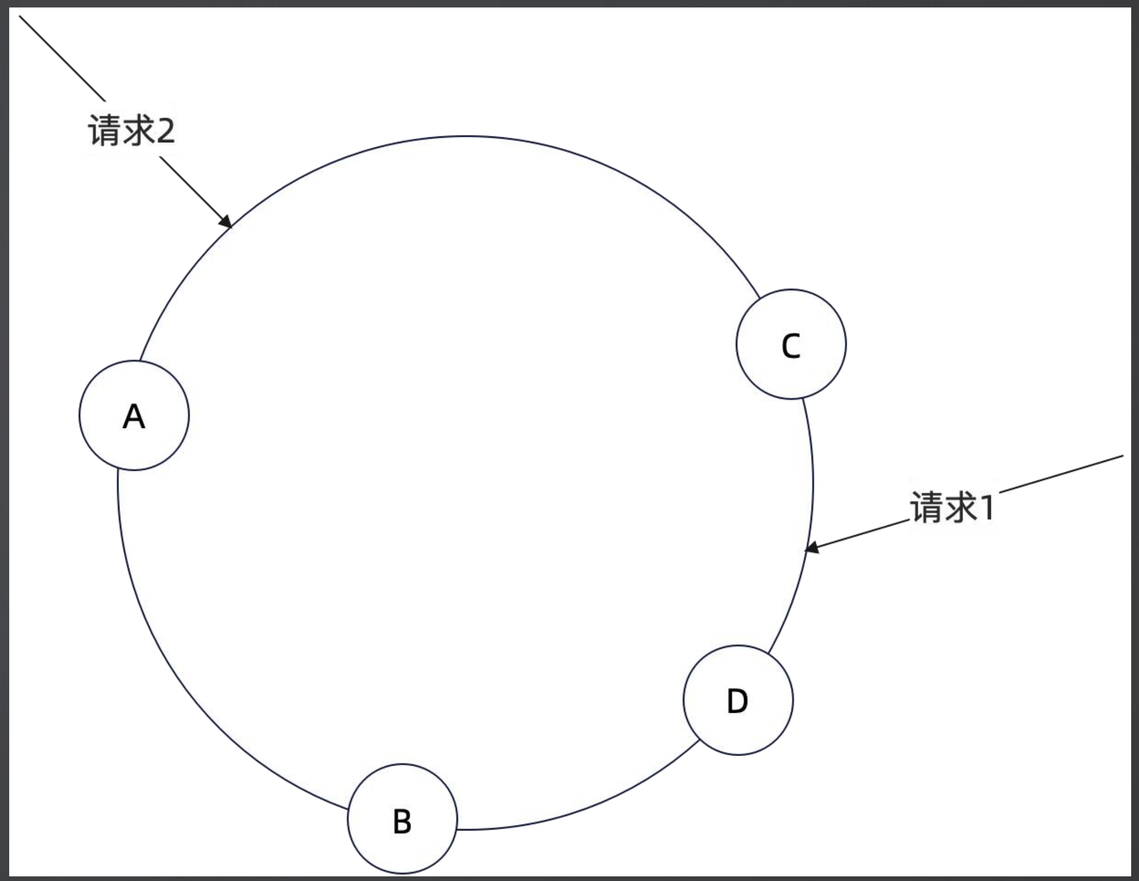
同样是计算哈希值，但是哈希值和节点的关系是：哈希值落在某一个区间内的，会被特定一个节点处理

优点：当增加节点或者减少节点的时候，只有一部分请求命中的节点会发生变化



负载均衡：一致性哈希





负载均衡：最小连接数

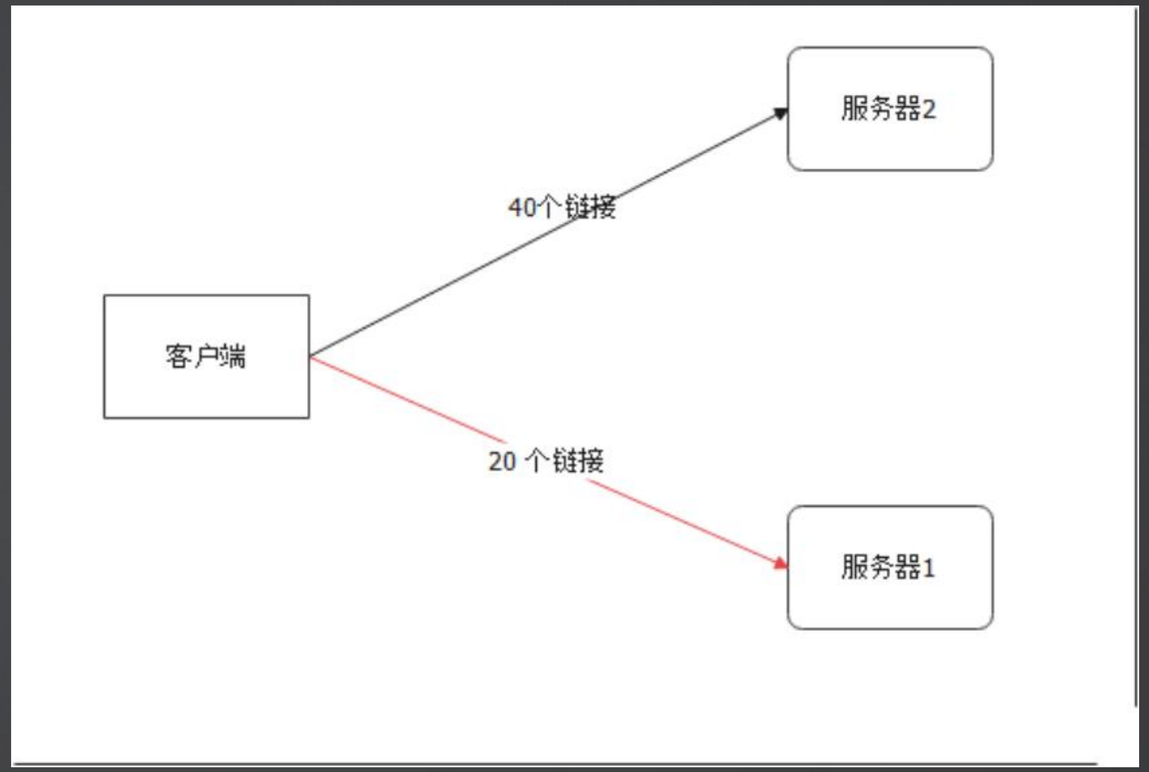
假设：

- 用连接数来代表服务器负载
- 所有服务器的处理能力是一样的
- 请求所需资源都一样

可能会短时间内把所有的请求都发到同一台服务器上。

连接复用的情况下，连接数不能很好地代表服务器的负载。

在 gRPC 里，因为我们不直接管理连接，所以这个算法是实现不了的。

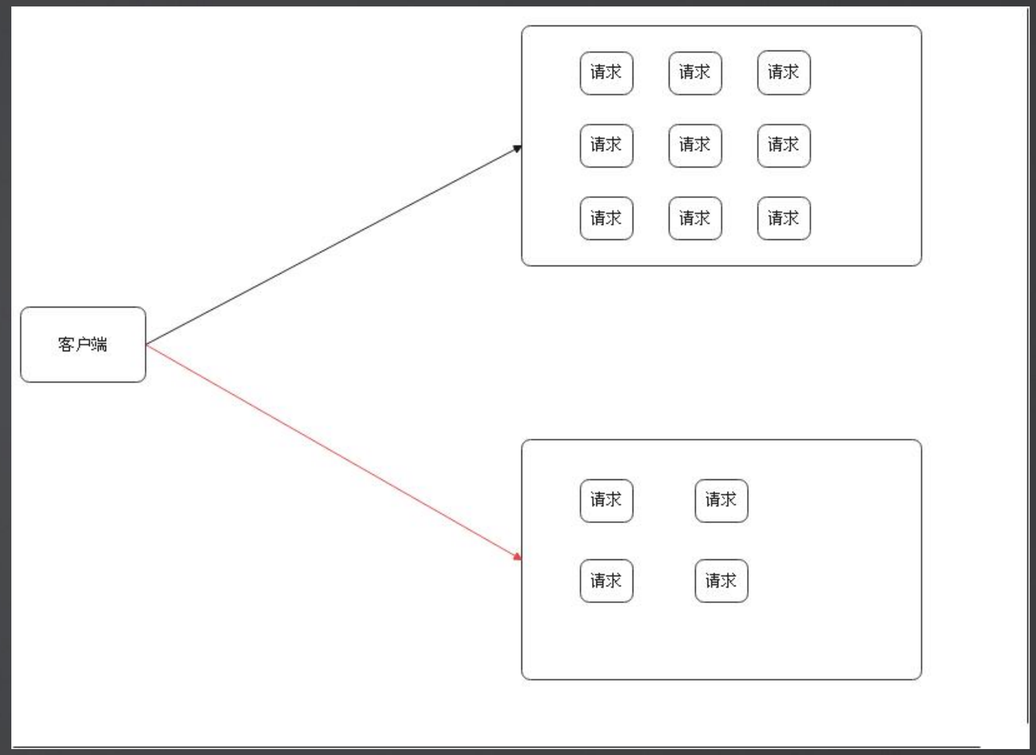


负载均衡：最少活跃数

假设：

- 用服务器上的请求数量来代表负载
- 所有服务器的处理能力是一样的
- 请求所需资源都一样

可能会短时间内把所有的请求都发到同一台服务器上。



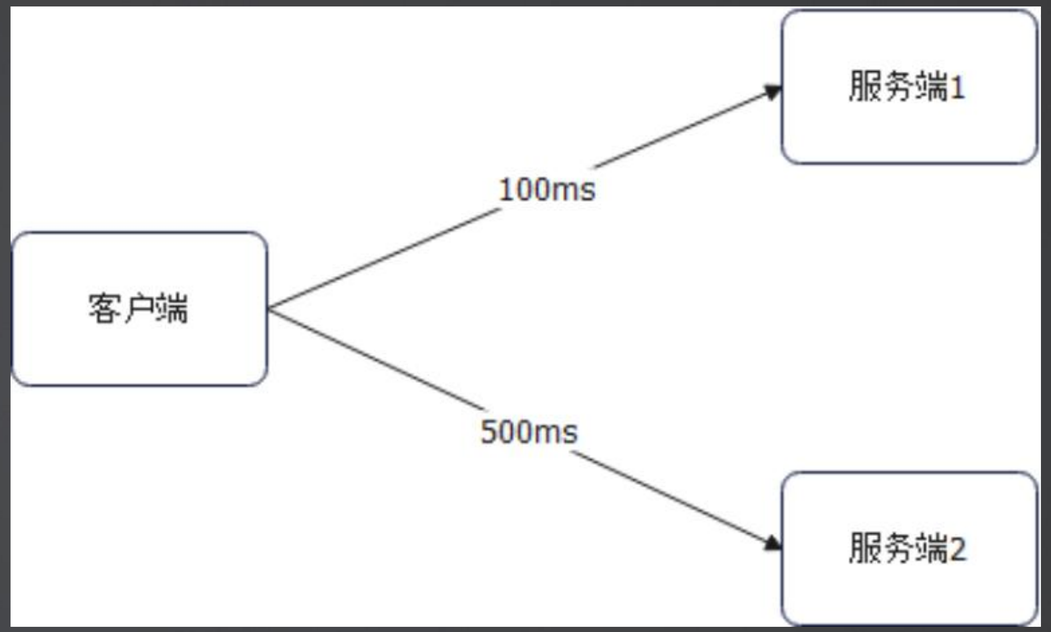
负载均衡：最快响应

假设：

- 用服务器上的响应时间来代表负载
- 请求所需资源都一样

偶尔几个慢请求会导致节点接下来不太可能会被选上。
极端情况下，永远不会被选上。

这是你们的作业。



负载均衡总结

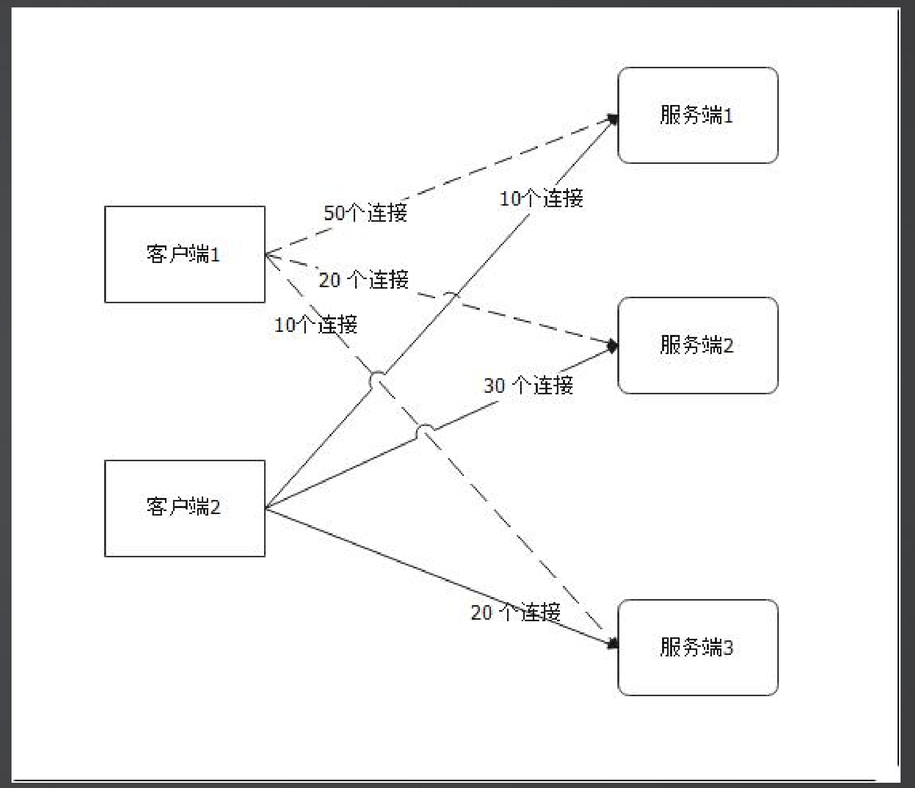
1. 要不要考虑服务器处理能力？
   - 轮询、随机、哈希、最小连接数、最少活跃数都没考虑。
2. 选择什么指标来表达服务器当前负载？
   - （加权）轮询、（加权）随机、哈希什么都没选，依赖于统计。
   - 选择连接数、请求数、响应时间、错误数……所以你可以随便选几个指标，然后设计自己的负载均衡算法。
3. 是不是所有的请求所需的资源都是一样的？显然不是
   - 大商家品类极多，大买家订单极多。
   - 不考虑请求消耗资源的负载均衡，容易出现偶发性的打爆某一台实例的情况。

总结：权重的效果

- 大多数时候我们都是使用权重来表达服务器的处理能力，或者说重要性。
- 使用权重的算法都要考虑：
  - 某个实例的权重特别大，可能连续几次都选中它，那么要考虑平滑效果。
  - 结合实际调用结果来调整权重，例如实例如果返回了错误，那么就降低权重，反之就增加权重。
  - 一个实例的权重如果是动态调整的，那么就要考虑上限和下限的问题，尤其要考虑调整权重的过程中会不会导致权重变成 0、最大值或者最小值。这三个值可能导致实例完全不被选中，或者一直被选中。

总结：微服务框架的局限性

- 在缺乏全局信息的情况下，客户端会选择服务端 1 作为服务提供者。
- 在微服务中选择负载均衡算法，这种需要全局信息的算法可能抖动会比较厉害。
- 那么为什么它们运作得还是很好呢？因为请求数量多了，慢慢会收敛到一种比较均匀的状态。



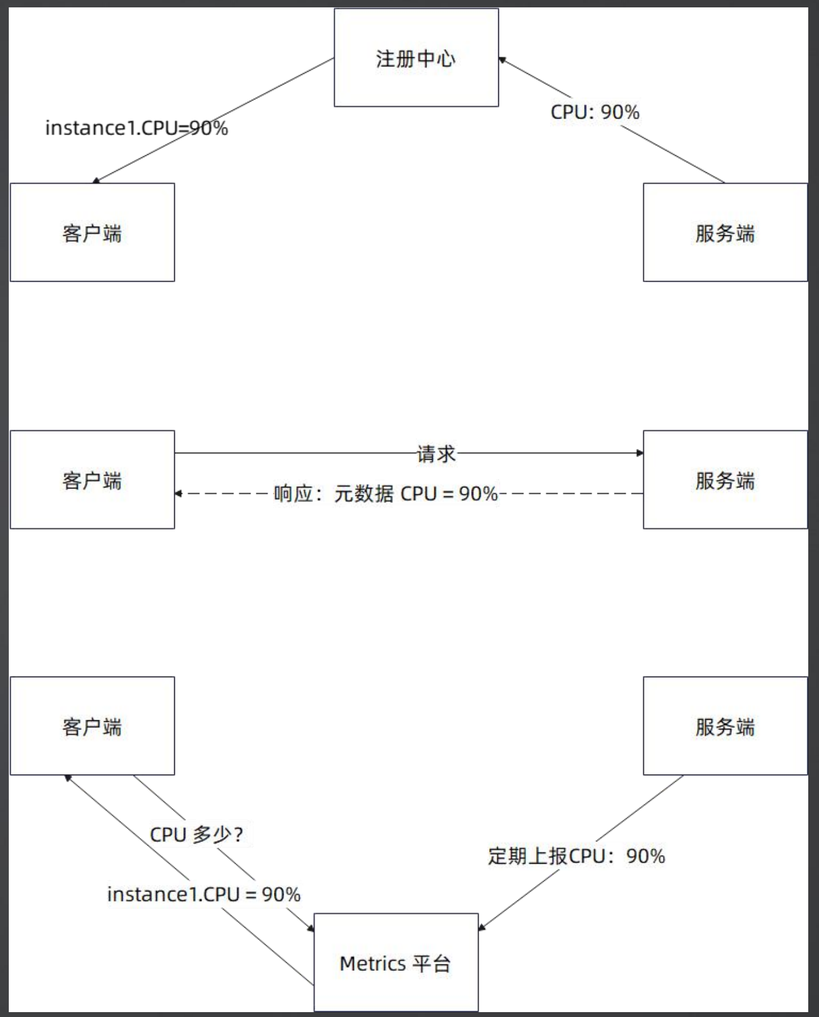
总结：设计自己的负载均衡算法

核心是根据自己的业务特征来选取一些指标，来表达服务实例的负载。

- 错误率等其它服务指标
- CPU、IO、网络负载等硬件指标

而想要知道这些指标，除了客户端统计（有些客户端没法统计，例如每个实例的 CPU），还有一些奇技淫巧：

- 服务端将指标的值写入注册中心，注册中心通知客户端。
- 服务端每次返回响应的时候，额外带上自己的指标，如 CPU 利用率。
- 利用我们的可观测性平台，从观测性平台获得数据。

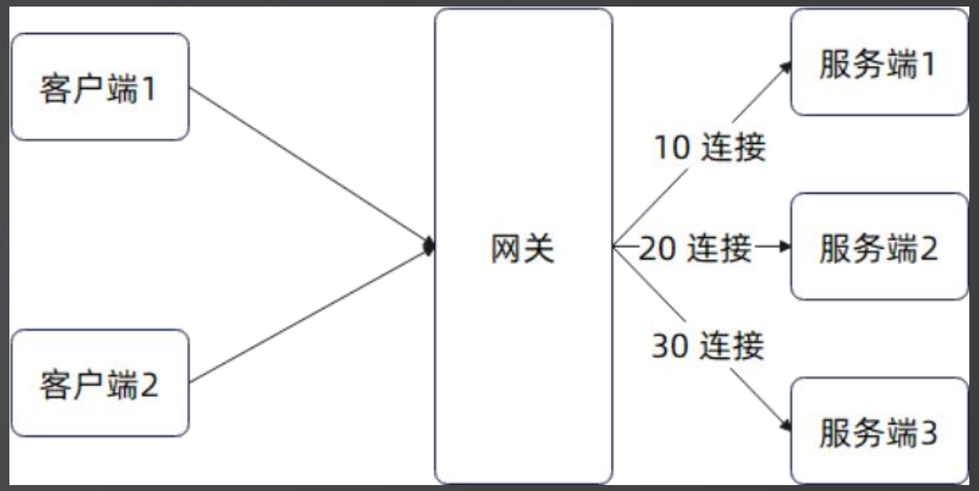


总结：网关负载均衡对比

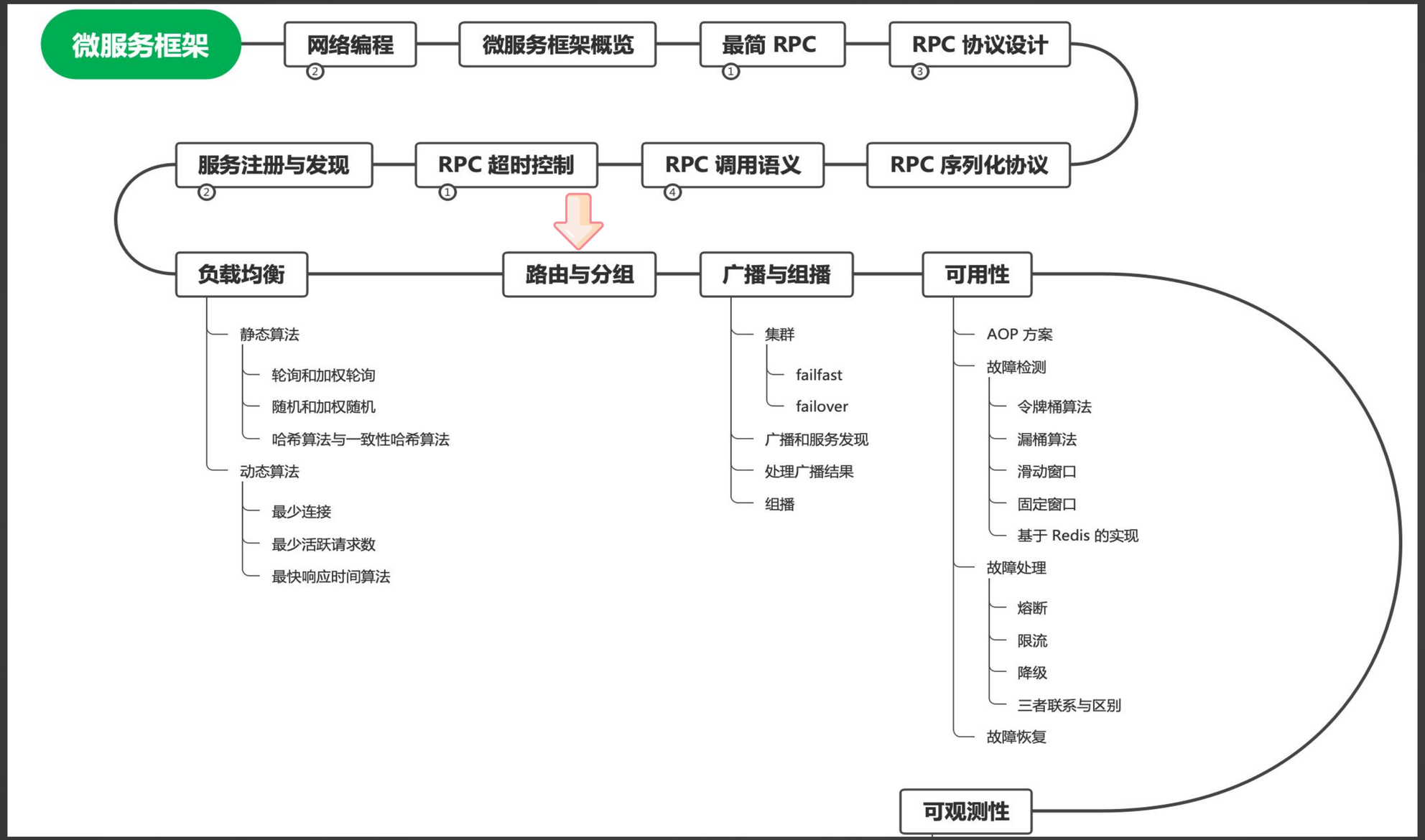
最大的优点：网关可以具备全局信息。然而，现在大多数的网关并没有利用自己的这个优势。

右图中，我们之前讲客户端不具备全局信息，那么作为对比，显然网关是具备的。

如果所有的客户端都经过网关才能和服务端进行通信，那么网关就可以考虑采集所有服务端实例的负载信息，做到一个全局最优的流量调度。



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
路由与分组
广播与组播
可用性
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
集群
failfast
failover
广播和服务发现
处理广播结果
组播
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

负载均衡就够了吗？

回顾一下前面讲的，我们是为了挑选出最合适的一个实例，然后将请求发送过去。

负载均衡挑出来了负载最轻的节点，这就够了吗？

并不够，因为负载均衡并没有考虑业务需求：

- 在 A/B 测试中，A 请求只能发到 A 节点上
- 在全链路压测中，压测流量只能发到测试节点上
- 在 VIP 服务中，VIP 的请求要发到更加高端的机器上
- 在联调或者 DEBUG 的时候，请求只能发送到一个特定的机器上

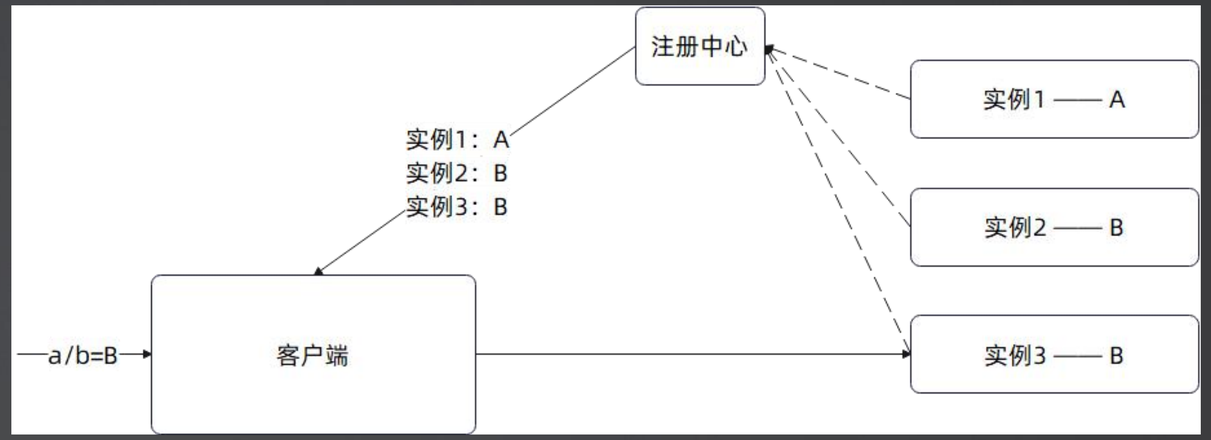
这些需求，一般统称为路由策略。

路由策略

在微服务领域，路由策略实质上就是微服务框架根据用户设定的流量转发规则，将符合条件的 RPC 请求转发到一些特定的服务实例上。

常见的有：

- 标签路由：给服务端实例打上不同的标签（可以是服务端自己打，也可以是客户端自己打），然后将特定请求路由到具备某个标签的服务实例上。
- 健康路由：实时将服务实例分成健康和不健康两大类，每次发送请求只发送给标记为健康的实例（类似于标签路由，也有点像是负载均衡）。
- 转发路由：用户直接指定来自某个客户端（或者某一类）的请求转发到特定的服务实例上。

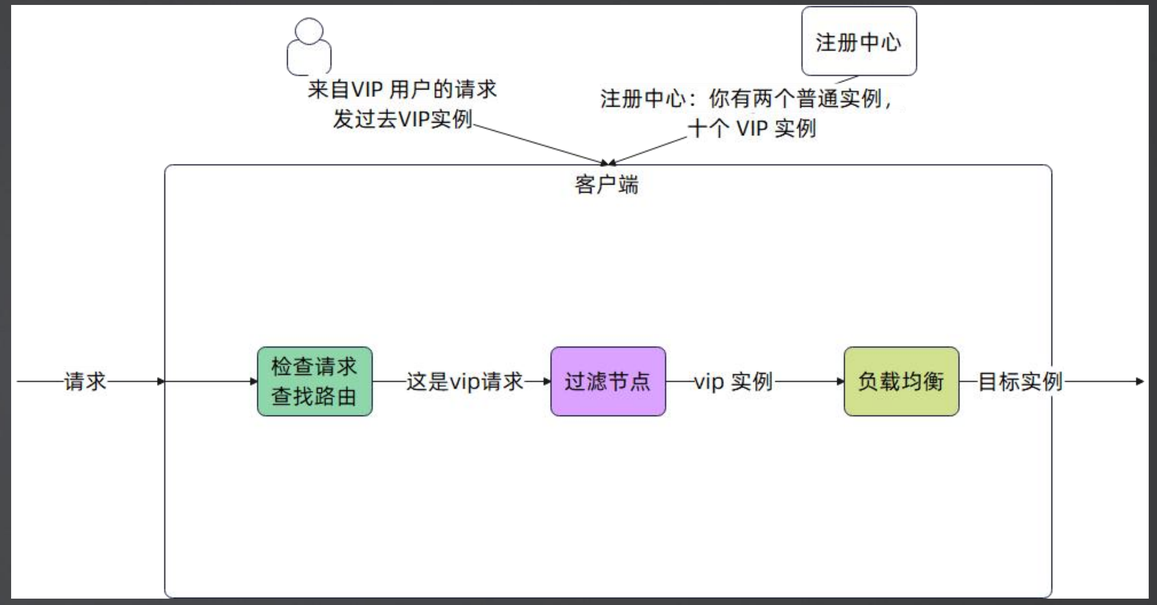


例如在 AB 测试中，服务端实例分成 AB 两组，那么客户端在收到请求的时候就要判断将请求发到哪个实例。

路由策略设计与实现核心

不同的路由策略都要解决：

- 用户设置路由策略，并且提供必要的信息，例如转发规则。
- 微服务框架要判断一个请求是否命中了任意一条路由策略。
- 微服务框架筛选符合条件的服务端实例。
- 微服务框架执行负载均衡策略找到目标节点，并发送请求。



路由策略与负载均衡

路由策略可以被看做是在负载均衡之前的一个步骤。

能不能复用负载均衡接口？

在 gRPC 里面，答案是可能可以。

需要注意：如果我们的实现在查找对应的路由策略时，所依赖的数据完全来自于 PickInfo，那么就可以。本质上也就是主要依赖于 Ctx 字段。

如果我们的实现依赖于具体的请求参数，例如 UserID，那么就不可以。

```
    // - For all other errors, wait for ready RP
    //   ready RPCs will be terminated with this
    //   status code Unavailable.
    Pick(info PickInfo) (PickResult, error)
}
```

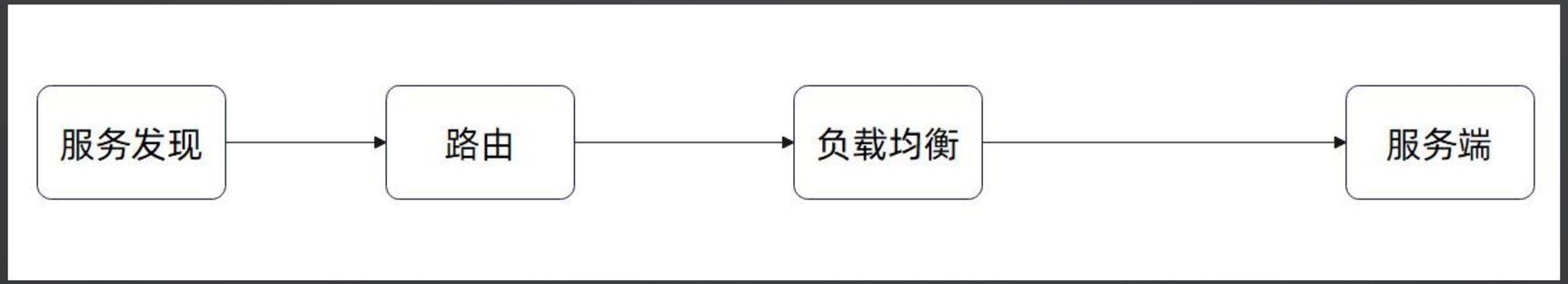
```
// PickInfo contains additional information fo
type PickInfo struct {
    // FullMethodName is the method name that
    // with. The canonical format is /service/
    FullMethodName string
    // Ctx is the RPC's context, and may conta
    // like the outgoing header metadata.
    Ctx context.Context
}
```

路由策略与负载均衡

我们可以用一种非常简单的策略：就是所谓过滤节点。

即我们认为，不管是什么路由，本质上就是为了在负载均衡之前，提前过滤一些节点。

例如分组路由，就是先找出特定组的节点；直接路由，就是使用特定 IP 和端口的节点；健康优先路由，就是过滤出”健康”的节点……



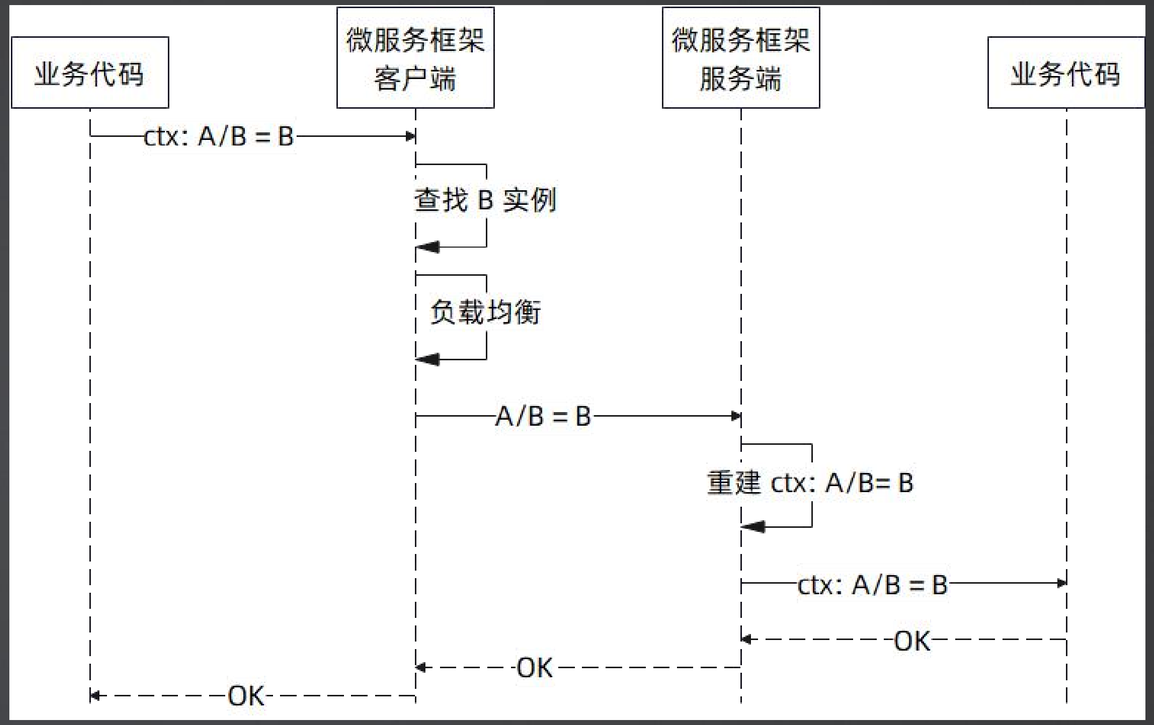
代码演示：实现分组功能

分组可以看做是一种特殊形态的路由策略，可以看做是服务端实例主动给自己打上一个标签。

我们用分组功能来实现一个简单的支持 A/B 流量分发的功能。

思路：

- 在整个链路里面带上一个 A/B 标记
- 进程内整个 A/B 标记位放在 context.Context 里
- 利用负载均衡接口，在负载均衡之前，先根据 A/B 接口筛选节点
- 发送请求到具体的服务端节点上



这个图很类似于我们之前讨论的链路超时控制。因为本质上两者都是在整条链路里面传递一些元数据，然后根据元数据来执行一些动作。

代码演示：实现分组功能

服务端在注册中心里面注册一个额外的分组信息，这里我们利用分组来将实例分成 A/B 两组。

至于一个服务端实例是 A 还是 B，由服务端开发者在部署的时候通过环境变量或者配置文件来指定。

客户端我们修改已有的负载均衡算法代码。

```
)

type Filter func(info balancer.PickInfo, address resolver.Address) bool
```

```
    candidates := make([]conn, 0, len(b.conns))
    for _, c := range b.conns {
        if !b.filter(info, c.address) {
            continue
        }
        candidates = append(candidates, c)
    }

    if len(candidates) == 0 {
        return balancer.PickResult{}, balancer.ErrNoSubConnAvailable
    }

    index := b.cnt % uint32(len(candidates))
    b.cnt = index + 1
    return balancer.PickResult{
        SubConn: candidates[index].SubConn,
```

过滤功能对负载均衡算法的影响

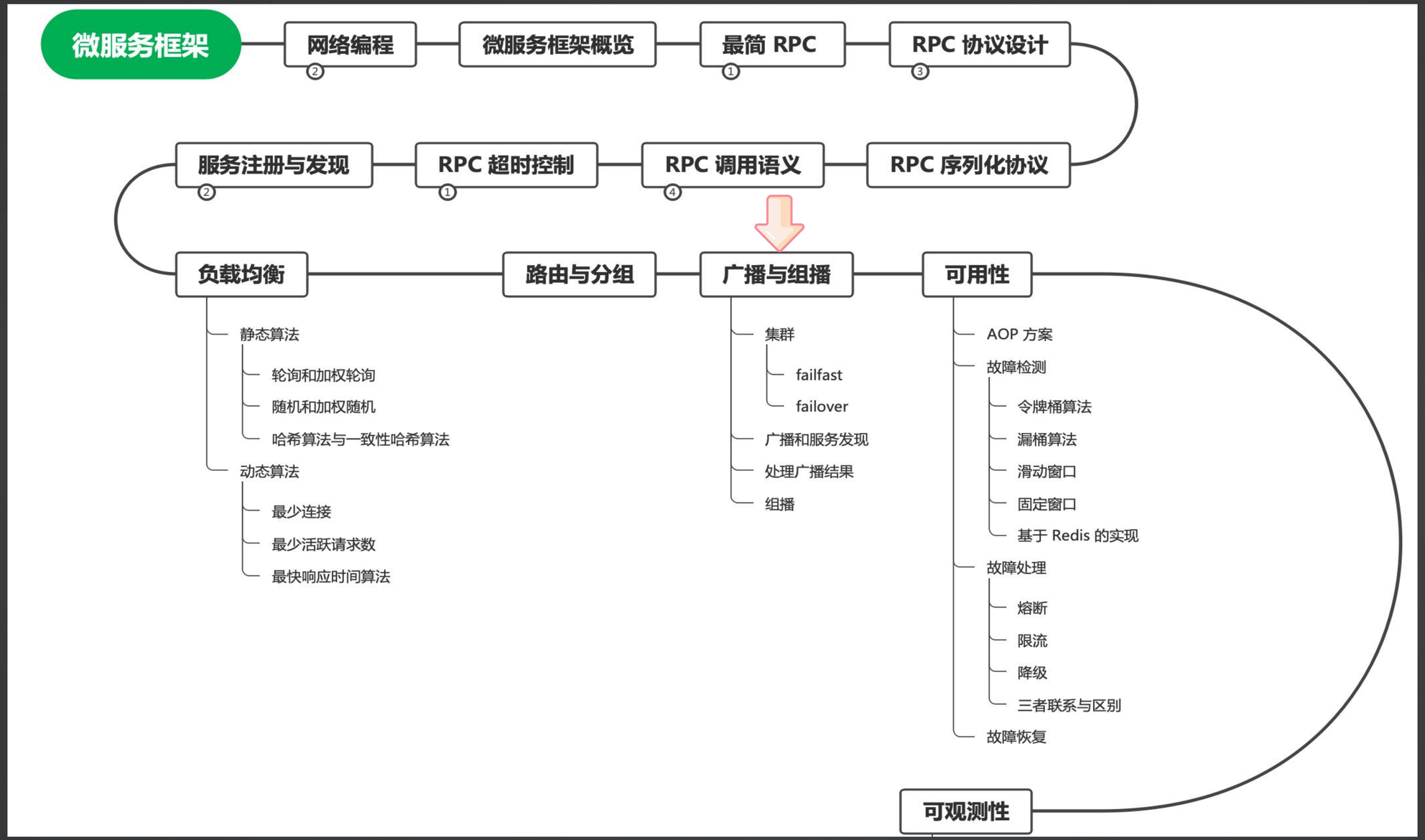
首先从实现的角度来说，大部分负载均衡算法都受到了影响，包括随机、轮询，以及对应的加权版本。

过滤功能使用不当可能会造成负载均衡算法失效：

- 过滤条件太苛刻以至于满足条件的实例几乎没有。
- 每次请求过滤之后的节点都不同，那么可能导致所有的请求都发到了少部分实例上。

```
    b.mutex.Lock()
    for _, node := range b.conns {
        if !b.filter(info, node.address) {
            continue
        }
        totalWeight += node.efficientWeight
        node.currentWeight += node.efficientWeight
        if res == nil || res.currentWeight < node.currentWeight {
            res = node
        }
    }
    res.currentWeight -= totalWeight
    b.mutex.Unlock()
    return balancer.PickResult{
```

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

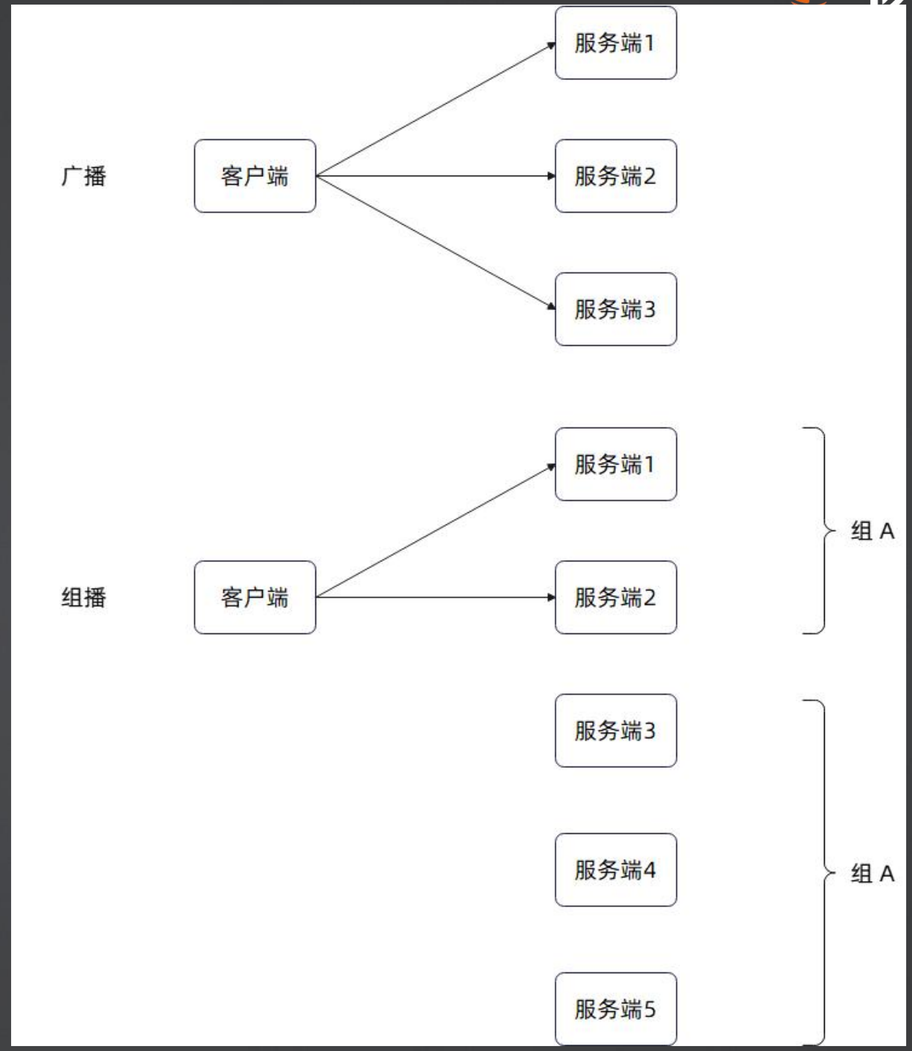
集群抽象：Cluster

Cluster 是沿用 Dubbo 中的说法。

它是指我们在调用远程服务的时候，尝试解决：

- failover：即引入重试功能，但是重试的时候会换一个新节点。
- failfast：立刻失败，不需要重试。
- 广播：将请求发送到所有的节点上。
- 组播：组播和分组功能不太一样，组播是指将请求发送到一组节点上，而不是只发送到一个单一节点上。

实际上，广播和组播可以看做是一类职能，failover 和 failfast 是另外一种职能。

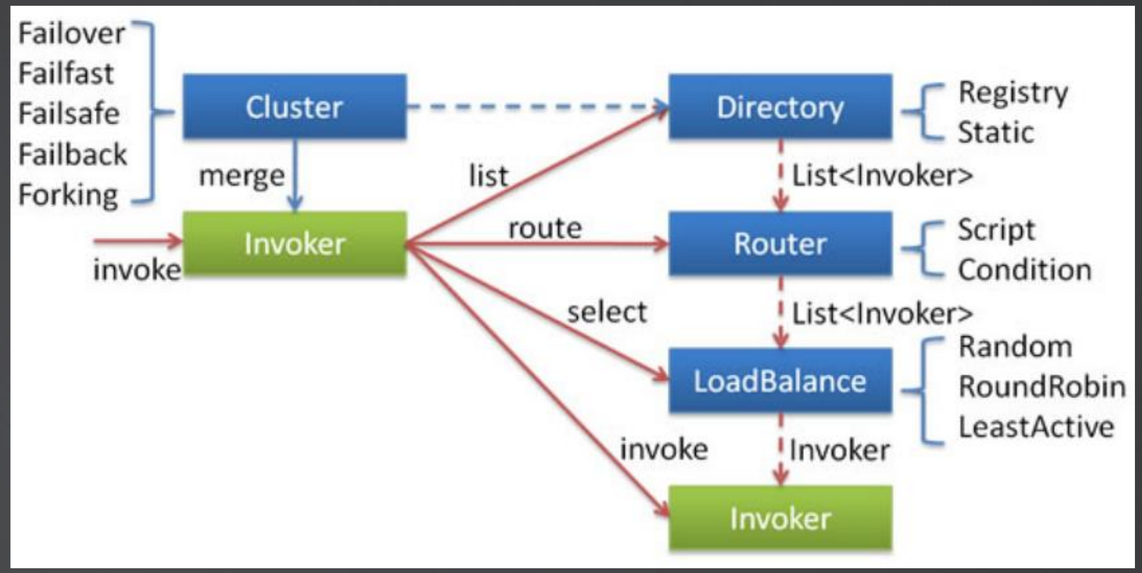


这里的组播和分组是不一样的，因为分组依旧是只发过去一个节点，而组播是发给一组节点。

Cluster：Dubbo-go 的设计

在 Dubbo-go 里面，它认为 Cluster 是发起调用的一个环节，可以看做是洋葱模式里面的一层洋葱。

右图是 Dubbo 社区对 Cluster 的基本抽象。



Cluster：Dubbo-go 的设计

Cluster 相关的接口主要有两个：

- Invoker 接口：最核心的接口，不同 Cluster 有不同的 Invoker 实现。
- Cluster 接口：可以看做是一层皮，将 Invoker 接口进行了封装，将多个 Cluster 组合在一起。

```
// Invoker the service invocation interface for t
//go:generate mockgen -source invoker.go -destina
// Extension - Invoker
type Invoker interface {
    common.Node
    // Invoke the invocation and return result.
    Invoke(context.Context, Invocation) Result
}
```

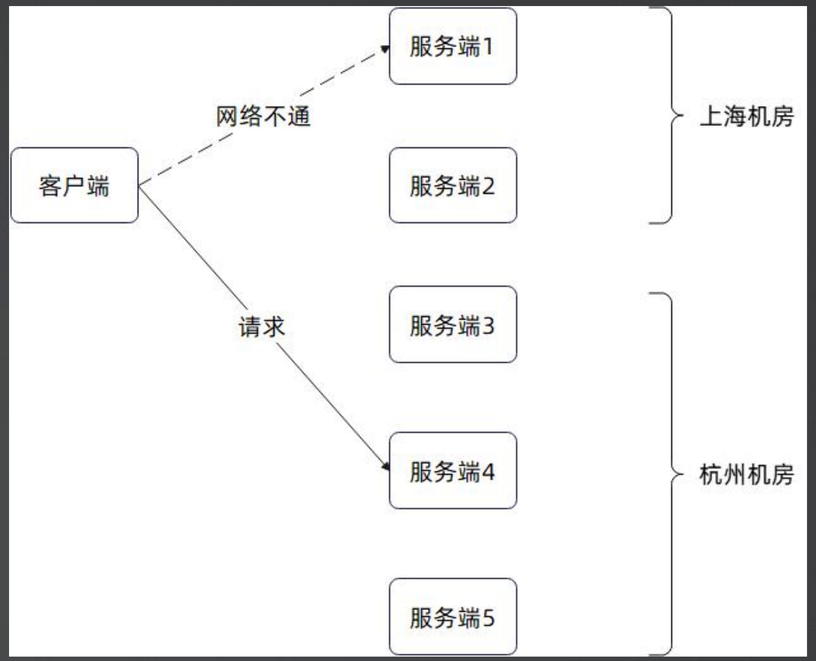
failover

failover 的要点在于：

- 重试：那么要考虑控制重试次数和重试间隔。
- 去除失败节点，选用新节点：
  - 新节点可能只是随便挑一个
  - 新节点也可能是不同机房上的节点
  - 新节点也可能在不同的城市

大多数 failover 的实现并没有那么精致，无非就是所有节点里面，去除已经使用但失败了的节点，剩下的随便挑一个。

而真的容错的话，是要考虑换机房换城市，这部分一般在流量调度和多活设计里面也会涉及到。



failover 实现要点就在于：从哪里挑选新节点？

例如如果我们知道当下调用失败是因为上海机房的网络已经崩溃了，那么我们就不应该选用任何上海机房节点，而是选用另外一个机房上的节点。

failover 在 gRPC 里面的实现

gRPC 天然支持了重试，我们只需要提供配置。而且 gRPC 每一次重试，都是要再一次经过负载均衡的，所以实际上我们只需要：

- 设置重试
- 选用合适的负载均衡算法，例如轮询

```
Policy := `{
    "LoadBalancingPolicy": "ROUND_ROBIN",
    "methodConfig": [{
        "waitForReady": true,
        "name": [{"service": "test.UserService"}],
        "retryPolicy": {
            "MaxAttempts": 4,
            "InitialBackoff": ".01s",
            "MaxBackoff": ".01s",
            "BackoffMultiplier": 1.0,
            "RetryableStatusCodes": [ "UNAVAILABLE" ]
        }
    }]
```

注意，因为在 gRPC 的负载均衡接口里，我们无法判断请求是不是重试的请求，所以我们只能选择那些每次都选出不同节点的负载均衡算法。

而如果是哈希之类的负载均衡算法，同一个请求选中的都是同一个节点，所以就没办法达成 failover 的效果。

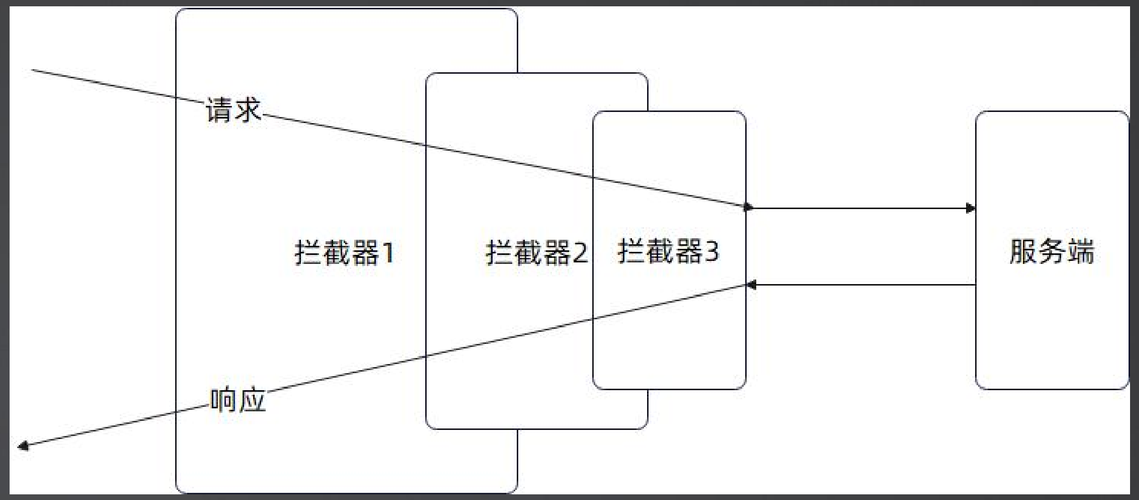
```
pickerBuilder := &roundrobin.Builder{}
client := micro.NewClient(micro.ClientWithInsecure(),
    micro.ClientWithRegistry(r, time.Second*3),
    micro.ClientWithPickerBuilder(pickerBuilder.Name(), pickerBuilder))
```

广播 broadcast

广播在 RPC 里面是指将请求发到所有的节点上。它在 gRPC 里面还是比较难实现的，核心在于 gRPC 暴露的接口并不合适。

在 gRPC 里面，我们能够尝试的接口就是拦截器 Interceptor，它对标我们在 Web 和 ORM 里面学的 AOP 方案。

但是拦截器这个接口限制太多。



拦截器的原理和我们前面使用的 Middleware 设计是一样的，只是叫法不同。

gRPC Interceptor

gRPC 的 Interceptor 分成好几种：

- UnaryClientInterceptor：用于拦截 gRPC unary（一元）请求。
- StreamClientInterceptor：用于拦截 gRPC 的 stream 请求。

```
var _ grpc.UnaryClientInterceptor = MyInterceptor

func MyInterceptor(ctx context.Context, method string, req, reply interface{},
    cc *grpc.ClientConn, invoker grpc.UnaryInvoker, opts ...grpc.CallOption) error {
    // 什么也不干，执行下一步
    return invoker(ctx, method, req, reply, cc, opts...)
}
```

可以看到，invoker 就接近于我们自己设计里面的 next。

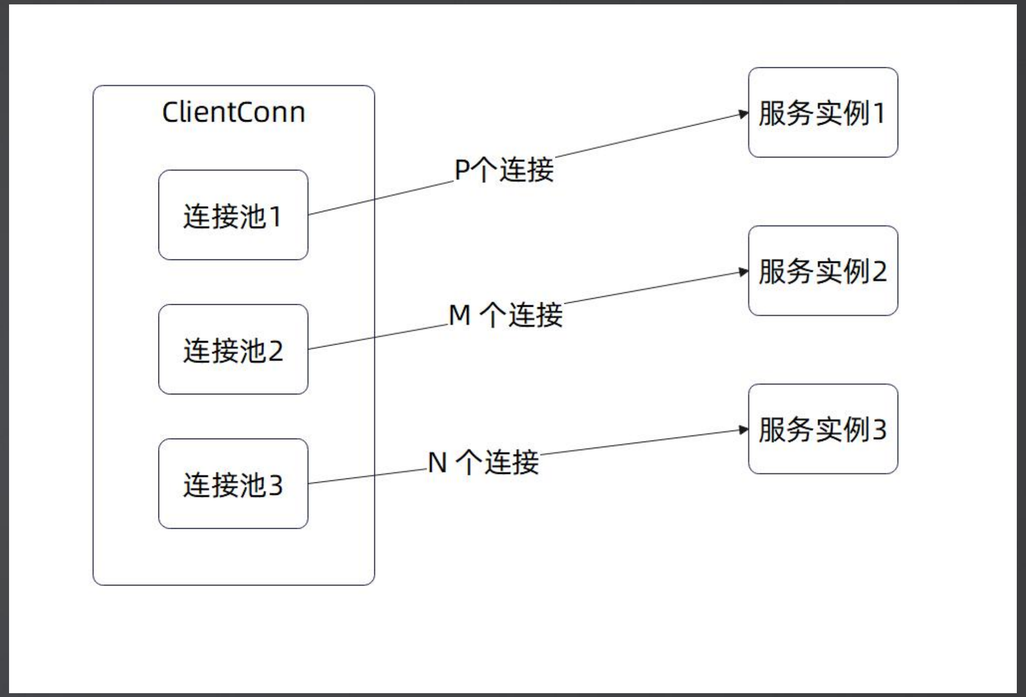
而为了实现广播的目标，关键就在于这个 cc *grpc.Client。

gRPC Interceptor：ClientConn

ClientConn 是 gRPC 里面的一个核心结构，可以将它理解为对某个服务的连接。因为服务本身会有很多节点，而且每个节点又可以有多个连接，所以可以将它看成是一个两层结构。

如右图，按照我们的广播目标，我们是希望能够遍历三个服务实例。

也就是我们希望能够从 ClientConn 里面拿出来对应不同实例的连接，然后发起调用。



gRPC Interceptor：ClientConn

很可惜我们拿不到。

我们只能考虑另辟蹊径。

注：理论上我们是可以使用 unsafe 来访问的。但这意味着我们需要一路使用 unsafe 直到达成我们的广播目标。

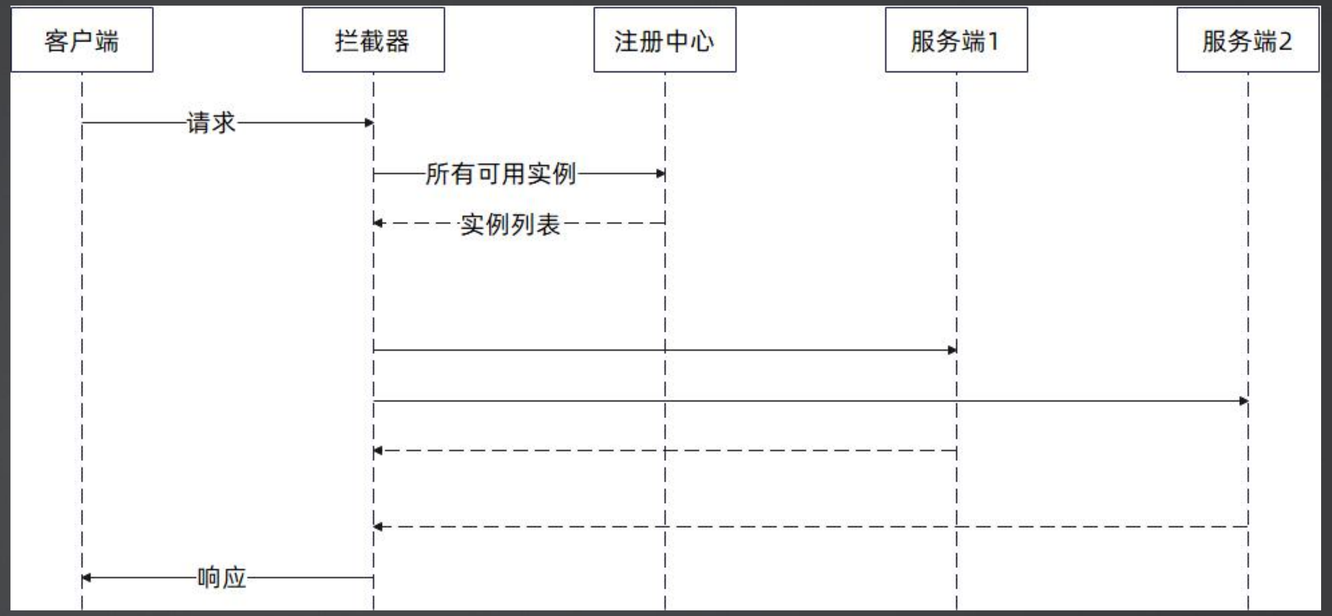
```
blockingpicker    *pickerWrapper

safeConfigSelector iresolver.SafeConfigSele

mu                sync.RWMutex
resolverWrapper *ccResolverWrapper
sc                *ServiceConfig
conns             map[*addrConn]struct{}
// Keepalive parameter can be updated if a
mkp               keepalive.ClientParameters
curBalancerName string
```

这就是我们预期中的

gRPC 广播：注册中心获取所有节点

思路：

- 利用拦截器捕获调用
- 利用注册中心来获得所有的服务端实例
- 在拦截器内遍历所有的服务端实例



注意，在调用服务端的时候，你可以设计为同步调用，也可以调用为并发调用。

gRPC 广播示例

实现分析：

- 利用 context.Context 来传递标记位，只有被标记为广播的，我们才会执行广播逻辑。
- 遍历所有的服务实例，为每一个实例创建 ClientConn 对象。

所以性能好不起来：

- 创建的 ClientConn 难以管理和复用，广播是一个低频需求。
- 难以复用本来就有的 ClientConn 对象。

```go
func (b ClusterBuilder) BuildUnary() grpc.UnaryClientInterceptor {
    return func(ctx context.Context, method string, req, reply interface{},
        cc *grpc.ClientConn, invoker grpc.UnaryInvoker, opts ...grpc.CallOptio
        if !isBroadCast(ctx) {  利用 context.Context 来传递标记位
            return invoker(ctx, method, req, reply, cc, opts...)
        }
        ins, err := b.registry.ListServices(ctx, b.service)
        if err != nil { return err }
        var eg errgroup.Group
        for _, instance := range ins {
```

```go
ins, err := b.registry.ListServices(ctx, b.service)
if err != nil { return err }          获取所有节点
var eg errgroup.Group
for _, instance := range ins {
    in := instance
    // 转变为直连。这里我们预期 Address 是一个真的地址，例如 IP + 端口
    eg.Go(func() error {
                                      遍历并且执行调用
        // 可怕的是我们每次进来都需要重新连，除非我们考虑缓存的问题
        // 缓存的问题则在于，我们需要管理它，在必要的时候关掉 conn
        conn, er := grpc.Dial(in.Address, b.opts...)
        if er != nil { return er }
        // 这里你可以考虑设计接口，允许用户把所有广播响应都拿到
        return conn.Invoke(ctx, method, req, reply, opts...)
    })
}
```

gRPC 广播响应处理

刚才的实现里面，只有最后一个返回的响应会被调用者拿到，本质上是因为在广播的时候，我们预期你不应该需要任何响应。

但在实际中，处理响应是有多种策略的。

- 直接丢弃：接近我们刚才的实现
- 返回最先的：所谓的最快调用
- 返回所有的

```
for _, instance := range ins {
    in := instance
    go func() {
        conn, er := grpc.Dial(in.Address, b.opts...)
        if er != nil {...}
        r := reflect.New(typ)
        val := r.Interface()
        err = conn.Invoke(ctx, method, req, val, opts...)
        select {
        case ch <- resp{err: err, val: r}:
        default:
        }
    }()
}
select {
case r := <-ch:
    if r.err == nil {
        reflect.ValueOf(reply).Elem().Set(r.val.Elem())
    }
```

发到 channel

只接收一次

最快调用。第一个返回的响应会被取走，剩余的直接丢弃了。为了避免覆盖类问题，我们都是用反射创建了新的响应。

gRPC 广播响应处理

这里我们使用 channel 来传递全部响应，也可以考虑使用切片来传递。

注意，使用 channel 传递的时候要防止 goroutine 泄露。同时要注意关闭 channel，否则用户在接收的时候，会不知道还有没有数据。

```
    typ := reflect.TypeOf(reply).Elem()
    var wg sync.WaitGroup
    wg.Add(len(ins))
    for _, instance := range ins {
        in := instance
        go func() {
            conn, er := grpc.Dial(in.Address, b.opts...)
            if er != nil {
                ch <- Resp{Err: er}
                return
            }
            r := reflect.New(typ)
            val := r.Interface()
            err = conn.Invoke(ctx, method, req, val, opts...
            // 这种写法的风险在于，如果用户没有接收响应，
            // 那么这里会阻塞导致 goroutine 泄露
            ch <- Resp{Err: err, Val: val}
            wg.Done()
        }()
    }
```

```
go func() {
    wg.Wait()
    // 要记得 close 掉，不然用户不知道还有没有数据
    // 用户在调用的时候是不知道有多少个实例还存活着
    close(ch)
}()
return nil
```

总结：Cluster、路由和负载均衡的关系

本质上，它们都在回答同一个问题：我要把请求发给谁？

课堂上我们演示了路由和负载均衡结合。

一般来说，Cluster 是不需要考虑负载均衡的——无论是组播还是广播，都是发给多个节点。

但是 Cluster 中的组播，也可以理解为广播 + 路由，因为路由的本质就是筛选出节点。

总结：服务发现与筛选节点

这个话题是一个值得在面试的时候吹嘘的问题。

理论上来说，在发起调用之后，调用结果要反馈给服务发现组件、Cluster 组件、路由组件和负载均衡组件，以确保：

- 如果调用失败，那么这些组件要考虑将目标节点挪出可用列表。一般来说，这个事情是由服务发现来负责的。
- 如果服务发现组件发现某个节点临时不可用，那么过一段时间之后要重新尝试探查一下这个节点是不是恢复过来了。

为什么说值得吹嘘？因为实际上，绝大多数微服务框架并没有做这种反馈式的服务发现和节点筛选功能。这意味着，例如在随机负载均衡里面，如果上一个请求失败了，下一个请求还是可能发给同一个节点。

总结：自定义路由策略

路由是一个非常强业务相关的特性。即大多数时候我们是根据业务规则来设计路由的，但是实现和前面提到的分组是类似的。

设计一个路由可以从以下方向考虑：

- 资源隔离角度：例如 VIP 用户和普通用户隔离，付费用户和免费用户隔离。
- 测试：可以为测试设计专属的路由，例如在全链路压测中。
- 动态分组：例如在运行时刻根据节点状况打不同的标签。

面试要点（一）

在集群（Cluster）方面，主要聚焦在 fail-over 上。

- 如果发起调用的时候，服务端调不通，那么该怎么办？你要仔细跟面试官讨论 fail-fast 还是 fail-over，以及 fail-over 要不要换节点。
- 什么是 fail-over？其实就是重试，深入讨论重试次数、退避算法、换不换节点等问题。
- 什么是微服务广播？以及微服务广播的使用场景？课堂上我们讲了两个例子，一个是缓存刷新，一个是寻找最快节点。前者是通知，后者是尽可能尽快服务。
- 什么是微服务组播？其实就是发送请求到一部分节点上，至于怎么分组，那是纯粹的业务问题。

面试要点（二）

在路由（Route）和负载均衡上：

- 掌握所有主流的负载均衡算法，注意分析优缺点。分析优缺点时要注意不同算法的假设，这些假设直接决定了算法的缺点。
- 怎么设置服务器权重？一般的原则是根据服务器的处理能力来设置权重。注意要讨论在负载均衡算法里面怎么动态调整权重，同时要注意权重调整不要超过一定的边界（注意溢出问题）。
- 为什么有了负载均衡，还是会出现某台机器被打爆的问题？也就是我们说的那些假设，实际上并不成立。例如大商家的请求就是要消耗更多资源。
- 客户端负载均衡和网关负载均衡有什么区别？网关负载均衡是有全局信息的，客户端负载均衡并没有，这是它们的本质区别。
- 什么是微服务路由？就是根据用户设置，将符合条件的请求发送到特定节点的过程。注意列举常见的一些路由策略，然后强调分组可以看做是一种特殊的路由策略。
- 如何自定义负载均衡算法？根据业务特征随便挑几个指标来设计负载均衡算法。

Q & A

THANKS