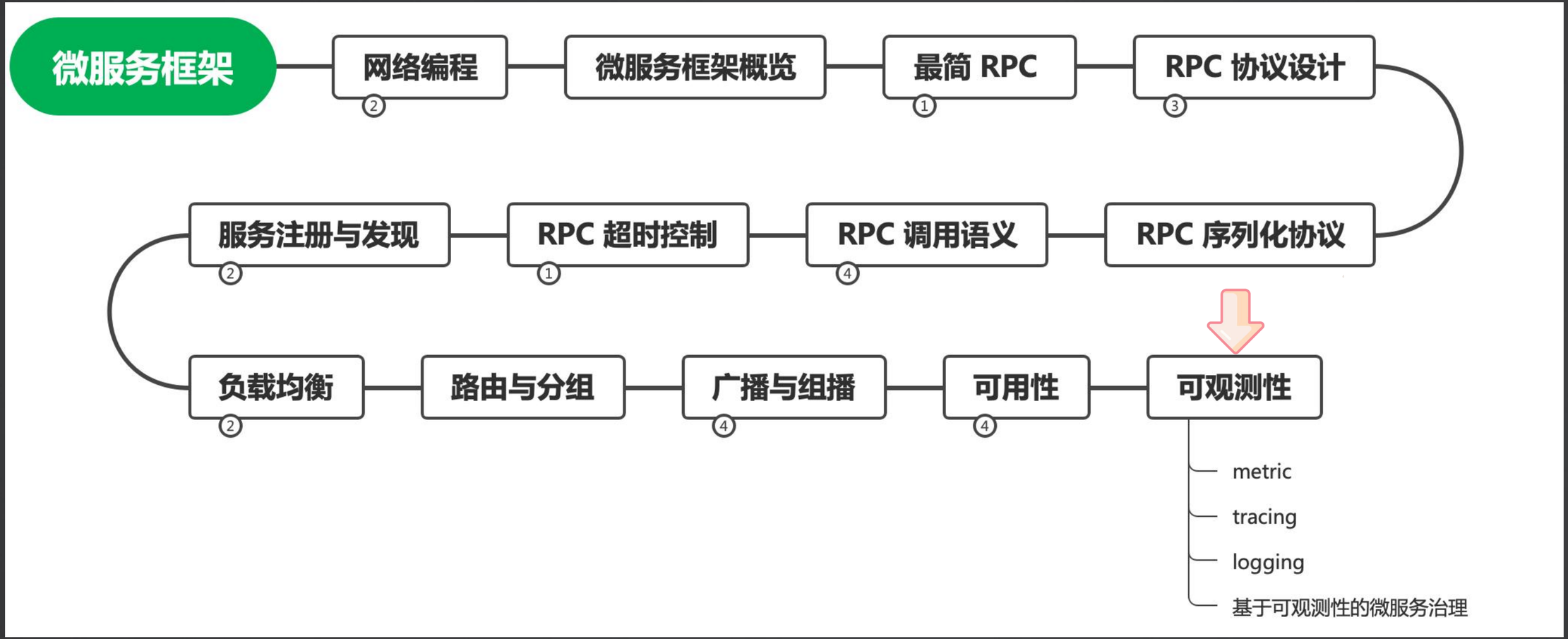
微服务框架 —— 可观测性

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
RPC 调用语义
4
RPC 超时控制
1
服务注册与发现
2
负载均衡
2
路由与分组
广播与组播
4
可用性
4
可观测性
metric
tracing
logging
基于可观测性的微服务治理

可观测性：Dubbo-go tracing

Dubbogo 里面的 tracing 也是将链路串联在了一起，
而后额外记录了一下错误信息。

```
result := invoker.Invoke(spanCtx, invocation)
span.SetTag(successKey, result.Error() == nil)
if result.Error() != nil {
    span.LogFields(log.String(errorKey, result.Error().Error()))
}
return result
```

```
operationName := invoker.GetURL().ServiceKey() + "#" + i

wiredCtx := ctx.Value(constant.TracingRemoteSpanCtx)
preSpan := opentracing.SpanFromContext(ctx)

if preSpan != nil {
    // it means that someone already create a span to tr
    span = opentracing.StartSpan(operationName, opentrac
    spanCtx = opentracing.ContextWithSpan(ctx, span)

} else if wiredCtx != nil {

    // it means that there has a remote span, usually fr
    span = opentracing.StartSpan(operationName, opentrac
    spanCtx = opentracing.ContextWithSpan(ctx, span)
} else {
    // it means that there is not any span, so we create
    span, spanCtx = opentracing.StartSpanFromContext(ctx
}
```

可观测性：Kratos metrics

Kratos 自身设计了错误传递方案，所以在它的 metrics 里面是可以做到分错误码观测的。

同样也记录了响应时间。

```
)
startTime := time.Now()
if info, ok := transport.FromClientContext(ctx); ok {
    kind = info.Kind().String()
    operation = info.Operation()
}
reply, err := handler(ctx, req)
if se := errors.FromError(err); se != nil {
    code = int(se.Code)
    reason = se.Reason
}
if op.requests != nil {
    op.requests.With(kind, operation, strconv.Itoa(cod
}
if op.seconds != nil {
    op.seconds.With(kind, operation).Observe(time.Sinc
}
return reply, err
```

可观测性：Kratos tracing

Kratos 的 tracing 也是利用了自己的错误传递机制，记录了错误码。

比较有趣的是，Kratos 记录了请求和响应的大小。

```
End finish tracing span
nc (t *Tracer) End(ctx context.Context, span trace.Span, m interface{}, err error
 if err != nil {
     span.RecordError(err)
     if e := errors.FromError(err); e != nil {
         span.SetAttributes(attribute.Key("rpc.status_code").Int64(int64(e.Code)
     }
     span.SetStatus(codes.Error, err.Error())
 } else {
     span.SetStatus(codes.Ok, description:"OK")
 }

 if p, ok := m.(proto.Message); ok {
     if t.kind == trace.SpanKindServer {
         span.SetAttributes(attribute.Key("send_msg.size").Int(proto.Size(p)))
     } else {
         span.SetAttributes(attribute.Key("recv_msg.size").Int(proto.Size(p)))
     }
 }
```

可观测性：Kratos logging

logging 记录的内容就比较多，比较重要的是尝试记录了请求参数。

我们在前面就讨论过，记录请求参数要顾虑两个点：

- 请求很大
- 请求里面包含敏感数据，例如住址和手机号码等

```
        code = se.Code
        reason = se.Reason
    }
    level, stack := extractError(err)
    _ = log.WithContext(ctx, logger).Log(level,
        keyvals...:"kind", "server",
        "component", kind,
        "operation", operation,
        "args", extractArgs(req),
        "code", code,
        "reason", reason,
        "stack", stack,
        "latency", time.Since(startTime).Seconds(),
    )
    return
}
```

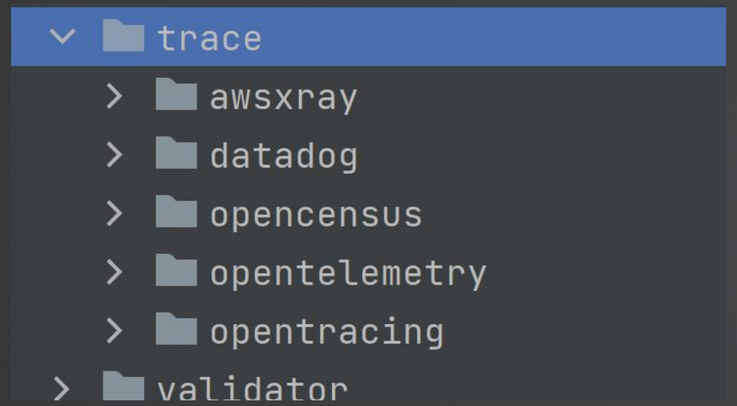
可观测性：go-micro metrics

go-micro 支持了 prometheus，观测了响应时间和请求总数、错误总数。

```
func (w *wrapper) CallFunc(ctx context.Context, node *registry.Node, r
    endpoint := fmt.Sprintf("#{req.Service()}.#{req.Endpoint()}")

    timer := prometheus.NewTimer(prometheus.ObserverFunc(func(v float6
        us := v * 1000000 // make microseconds
        timeCounterSummary.WithLabelValues(w.options.Name, w.options.V
        timeCounterHistogram.WithLabelValues(w.options.Name, w.options
    }))
    defer timer.ObserveDuration()

    err := w.callFunc(ctx, node, req, rsp, opts)
    if err == nil {
        opsCounter.WithLabelValues(w.options.Name, w.options.Version,
    } else {
        opsCounter.WithLabelValues(w.options.Name, w.options.Version,
    }

    return err
```

可观测性：go-micro tracing

go-micro 支持了很多 tracing 工具。



右图是 opentelemetry，关键步骤是提取链路元数据，将 tracing 串起来，而后就是随便记录了一下响应情况。

```
func StartSpanFromContext(ctx context.Context, tp trace.TracerProvid
    md, ok := metadata.FromContext(ctx)
    if !ok {
        md = make(metadata.Metadata)
    }
    propagator, carrier := otel.GetTextMapPropagator(), make(propaga
    for k, v := range md {...}        串起来链路
    ctx = propagator.Extract(ctx, carrier)
    spanCtx := trace.SpanContextFromContext(ctx)
    ctx = baggage.ContextWithBaggage(ctx, baggage.FromContext(ctx))
```

```
turn func(ctx context.Context, node *registry.Node, req cl
  if options.CallFilter != nil && options.CallFilter(ctx,
  name := fmt.Sprintf("#{req.Service()}.#{req.Endpoint()}"
  spanOpts := []trace.SpanStartOption{    名字
      trace.WithSpanKind(trace.SpanKindClient),
  }
  ctx, span := StartSpanFromContext(ctx, options.TraceProv
  defer span.End()
  if err := cf(ctx, node, req, rsp, opts); err != nil {
      span.SetStatus(codes.Error, err.Error())
      span.RecordError(err)
      return err                    记录了响应情况
  }
  return nil
```

可观测性：metrics

这里我们直接就记录一下执行时间，并且尝试记录响应情况。

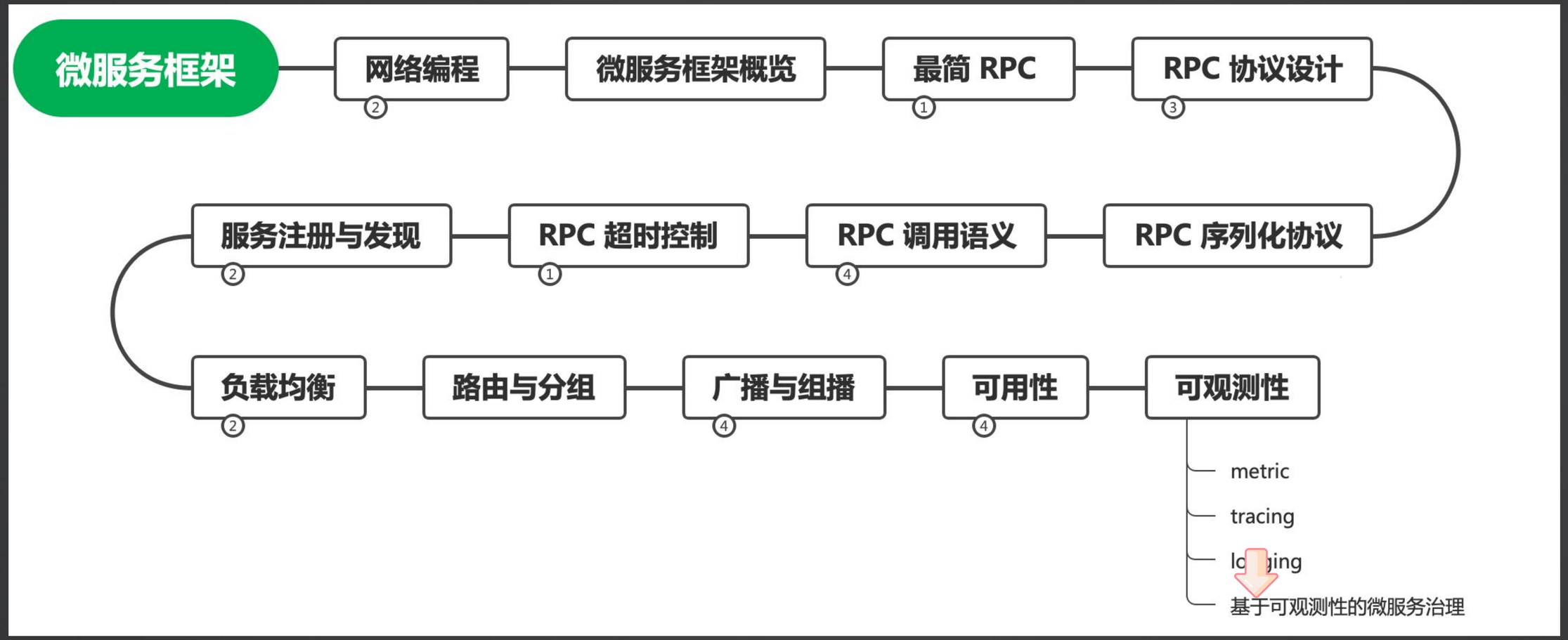
```
prometheus.MustRegister(summaryVec, errCntVec, reqCntVec)
return func(ctx context.Context, req interface{}, info *grpc.U
    reqCnt := reqCntVec.WithLabelValues(info.FullMethod)
    reqCnt.Add(1)
    startTime := time.Now()
    // 类似于 opentelemetry，这里也可以记录一下业务ID之类的信息
    defer func() {
        if err != nil {
            errCntVec.WithLabelValues(info.FullMethod).Add(1)
        }
        duration := time.Now().Sub(startTime)
        reqCnt.Sub(1)
        summaryVec.WithLabelValues(info.FullMethod).Observe(fl
    }()
    resp, err = handler(ctx, req)
    return
}
```

可观测性：tracing

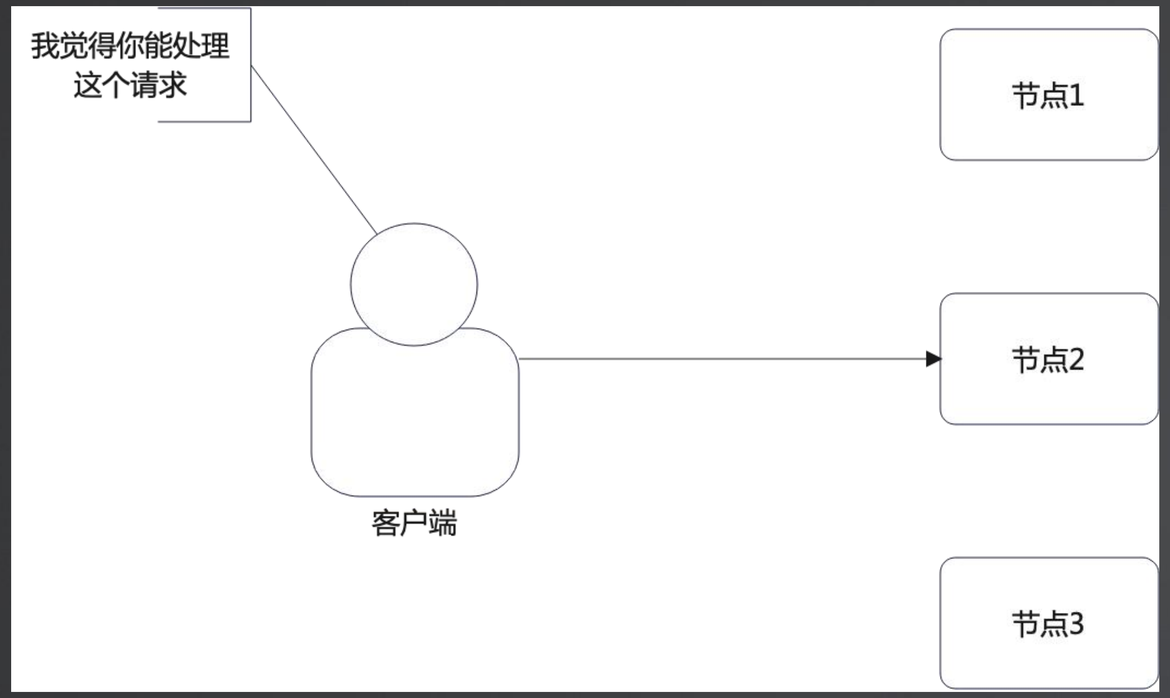
注意，在这里我们也用 extract 方法来把链路串起来。

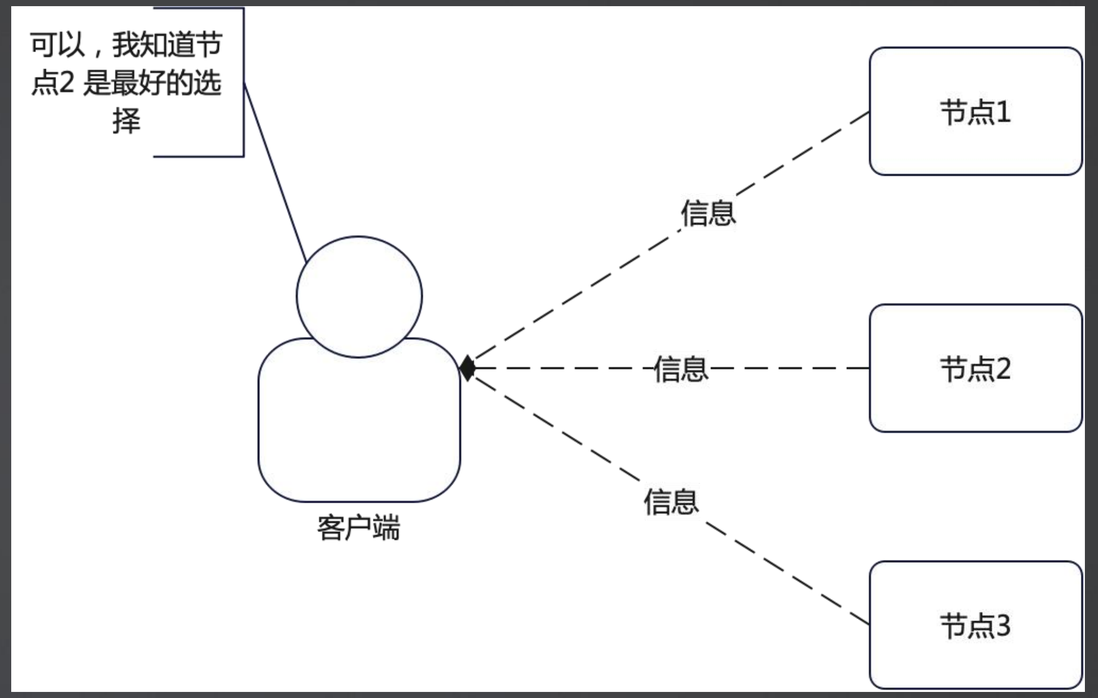
```
return func(ctx context.Context, req interface{}, info *grpc.UnaryServe
    handler grpc.UnaryHandler) (resp interface{}, err error) {
    ctx, span := s.Tracer.Start(ctx, info.FullMethod, trace.WithSpanKi
    ctx = s.extract(ctx)
    // 这里可以记录非常多的数据，一般来说可以考虑机器本身的信息，例如 ip，端口
    // 也可以考虑进一步记录和请求有关的信息，例如业务 ID
    span.SetAttributes(attribute.String( k: "address", address))
    defer func() {
        if err != nil {
            // 在使用 err.String()
            span.SetStatus(codes.Error, description: "server failed")
            span.RecordError(err)
        }
        span.End()
    }()
    resp, err = handler(ctx, req)
    return
```

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
2
路由与分组
广播与组播
4
可用性
4
可观测性
metric
tracing
基于可观测性的微服务治理

基于可观测性的服务治理





静态策略：

- 负载均衡：随机、哈希、轮询……
- 限流：固定窗口、滑动窗口、漏桶、令牌桶……

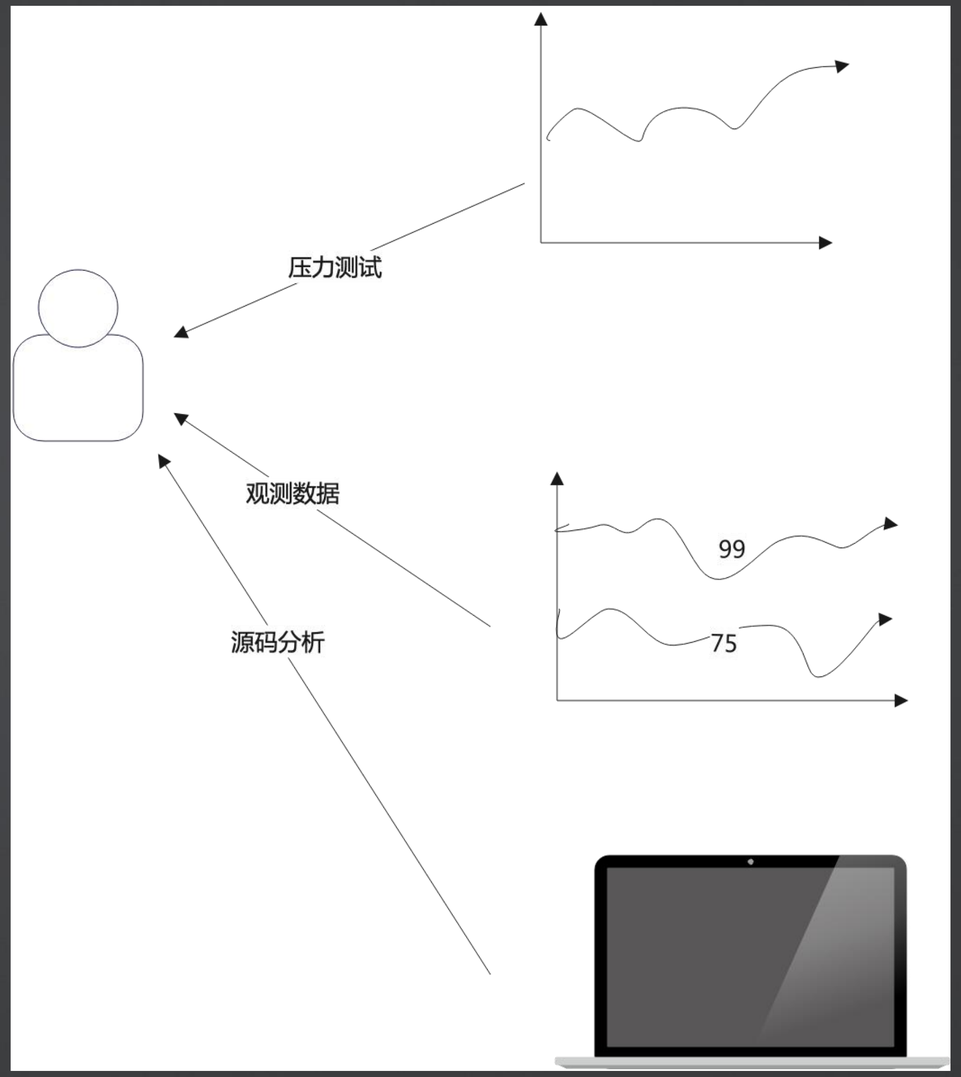
动态策略：

- 负载均衡：最少连接数、最少活跃请求数……
- 限流：BBR……

基于可观测性的服务治理：缺陷

静态算法依赖于程序员的个人经验。例如在限流中，有三种方式确定限流的阈值：

- 基于压测结果
- 基于可观测性数据
- 基于源码分析

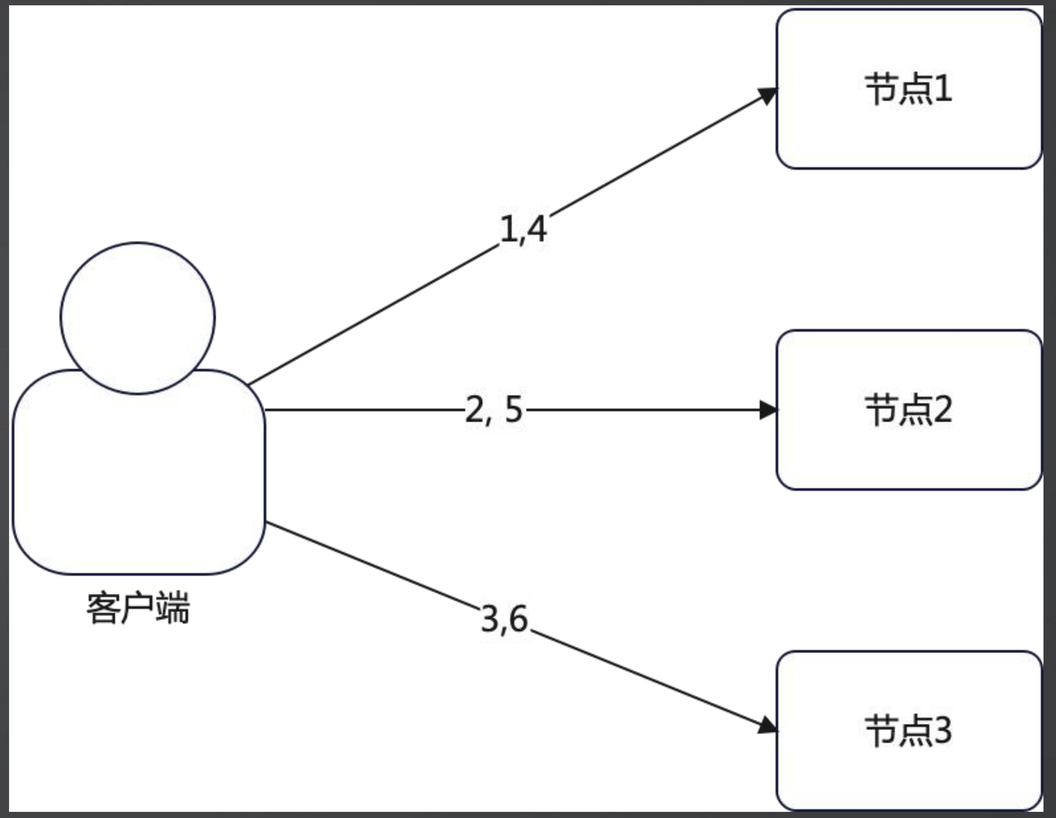


基于可观测性的服务治理：缺陷

对于负载均衡来说，有一些基本的假设：

- 所有的请求都消耗一样的资源
- 大量的请求

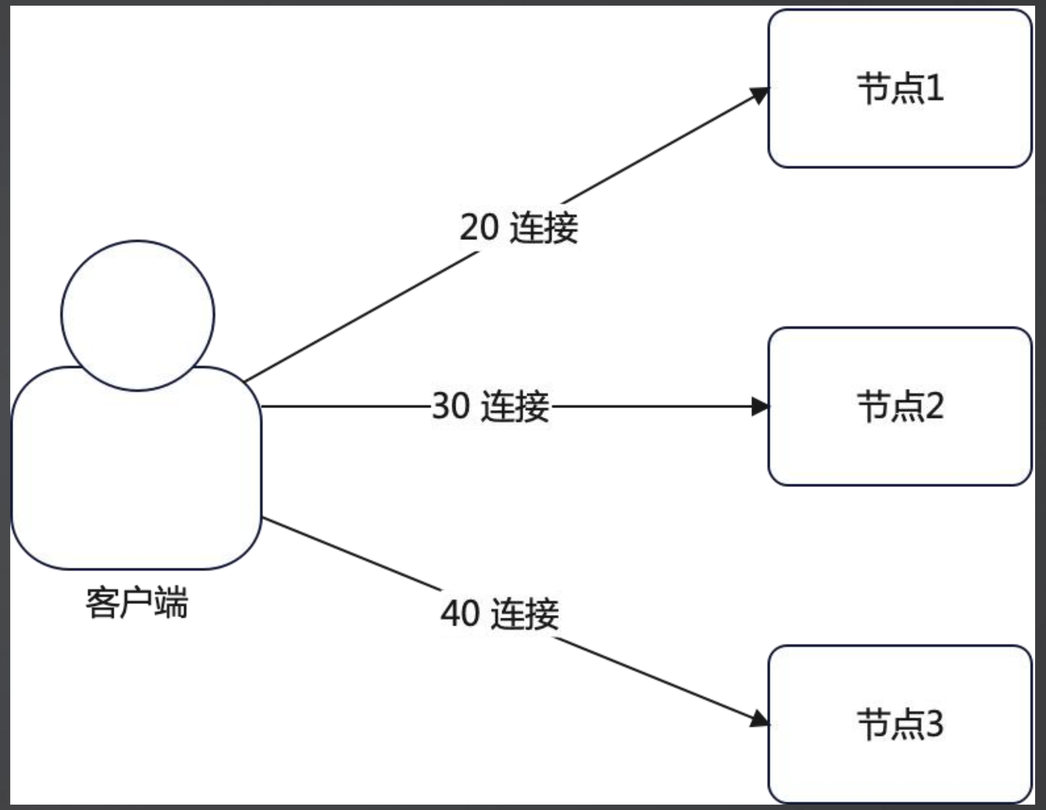
假设按照 @user_id % 3 进行哈希负载均衡，并且如果 user_id 1 是热门用户（例如大 up 主），那么节点 1 就可能过载。



基于可观测性的服务治理：缺陷

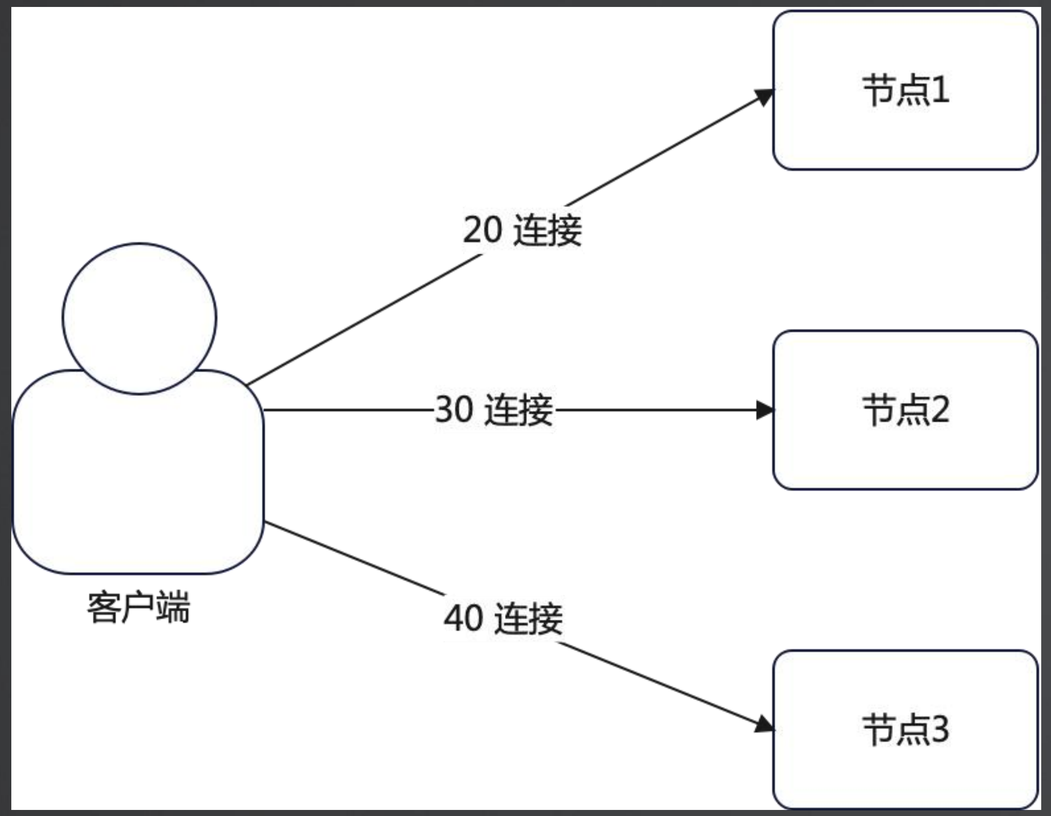
动态策略则是试图寻找一些指标，用来判断一个节点是否处于健康状态，或者说判断一个节点比另外一个节点更加健康。例如：

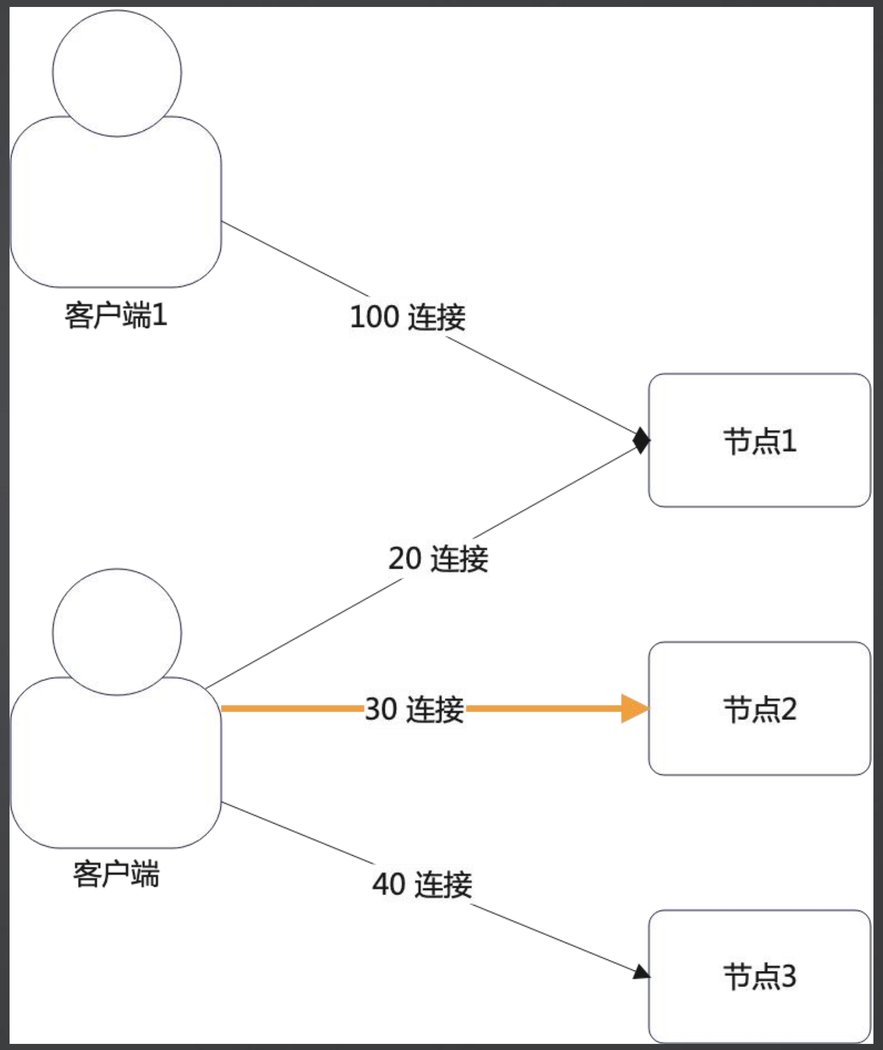
- 最少连接数：使用连接数，连接数越少认为越健康。
- 最少活跃请求数：使用当前正在处理的请求数量，越少认为越健康。
- 最快响应时间：使用响应时间，越快认为越健康。



客户端应不应该选择节点 1 作为目标节点？

基于可观测性的服务治理：缺陷





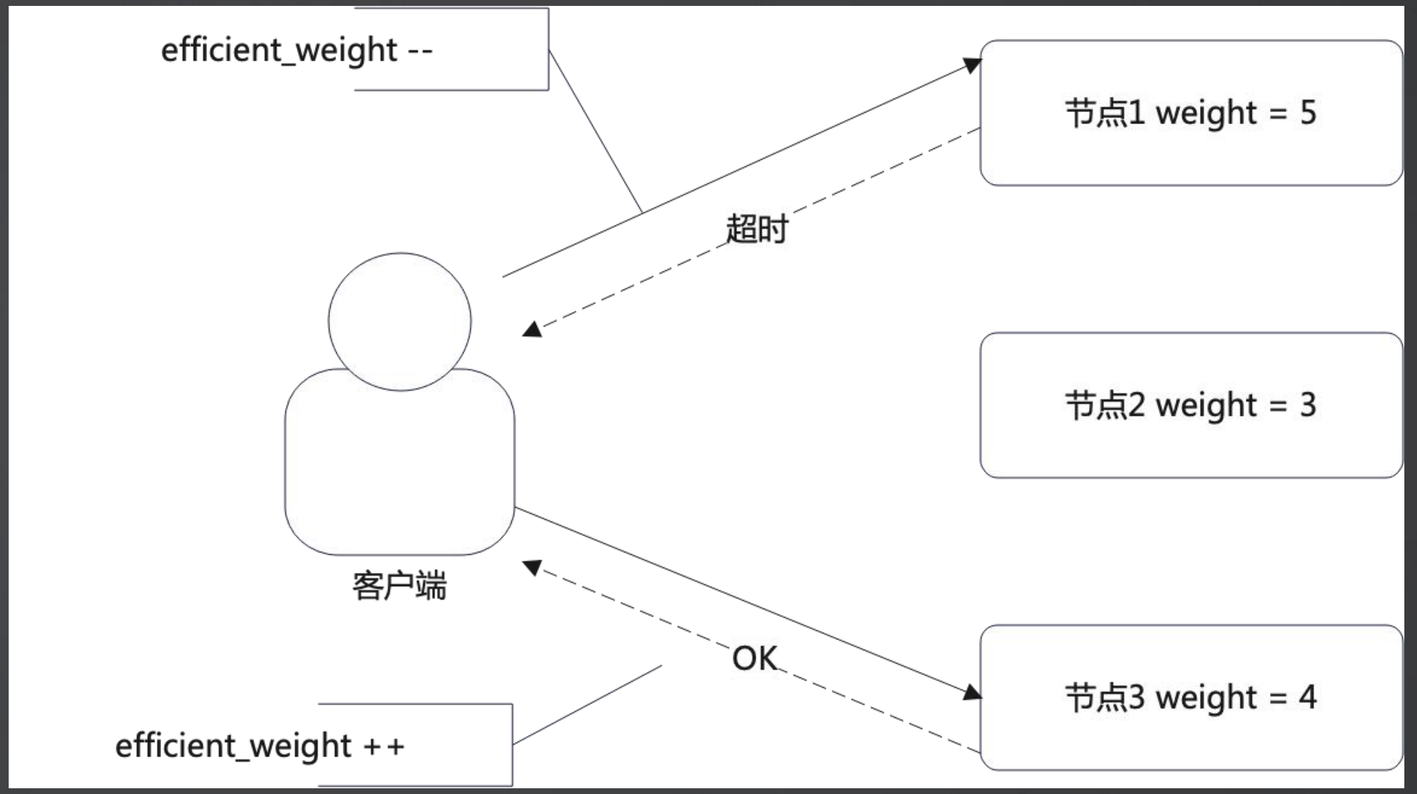
正如我们提过的，微服务框架本身并不具备全局信息，所以客户端最终会做出错误的选择，而正确的选择是节点2。

基于可观测性的服务治理：缺陷

动态调整权重类的负载均衡算法，就是试图通过增加权重或者降低权重来表达一个节点的健康程度。

例如在超时的时候降低权重，而在拿到了响应之后增加权重。

这一类的算法在计算权重的时候需要非常小心。



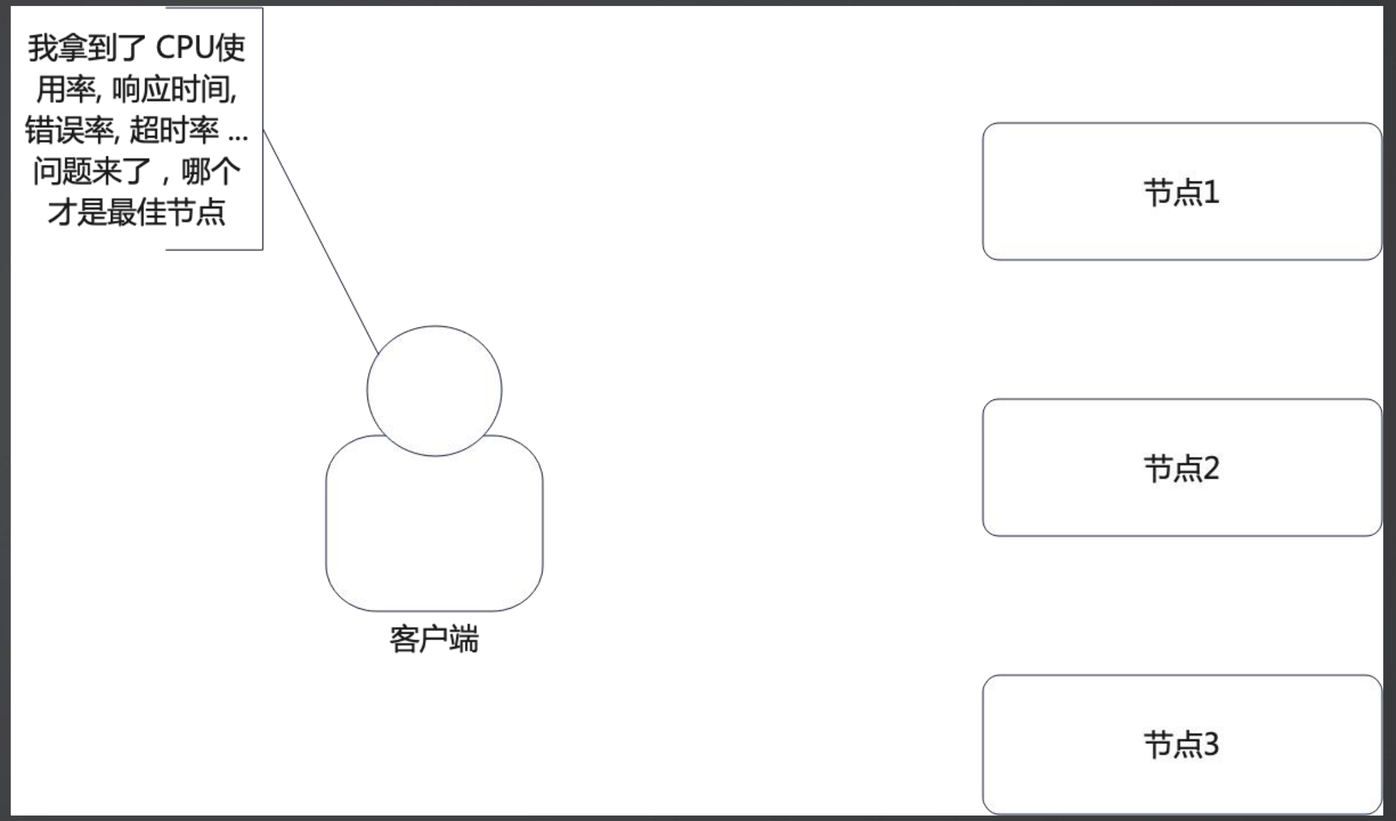
可利用指标

大多数情况下，静态策略就运作良好。

而动态策略则可以考虑使用：

- 硬件或者环境指标：例如 CPU 利用率、内存利用率、GC 时间……
- 服务指标：响应时间、错误率、超时率……

这些指标并不是独立的，相互之间有影响。

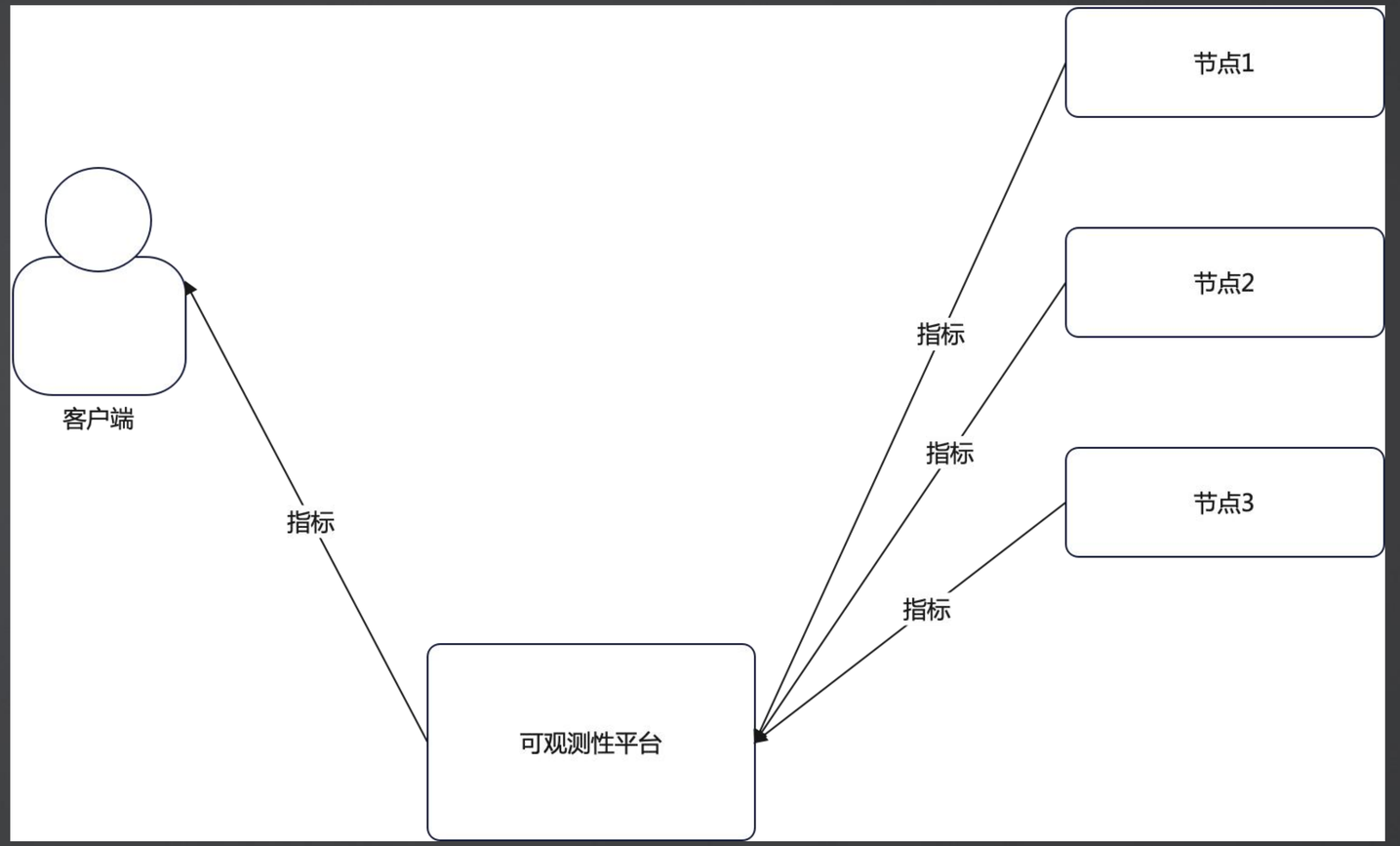


基于可观测性的服务治理基本思路

基本思路（以负载均衡为例）：

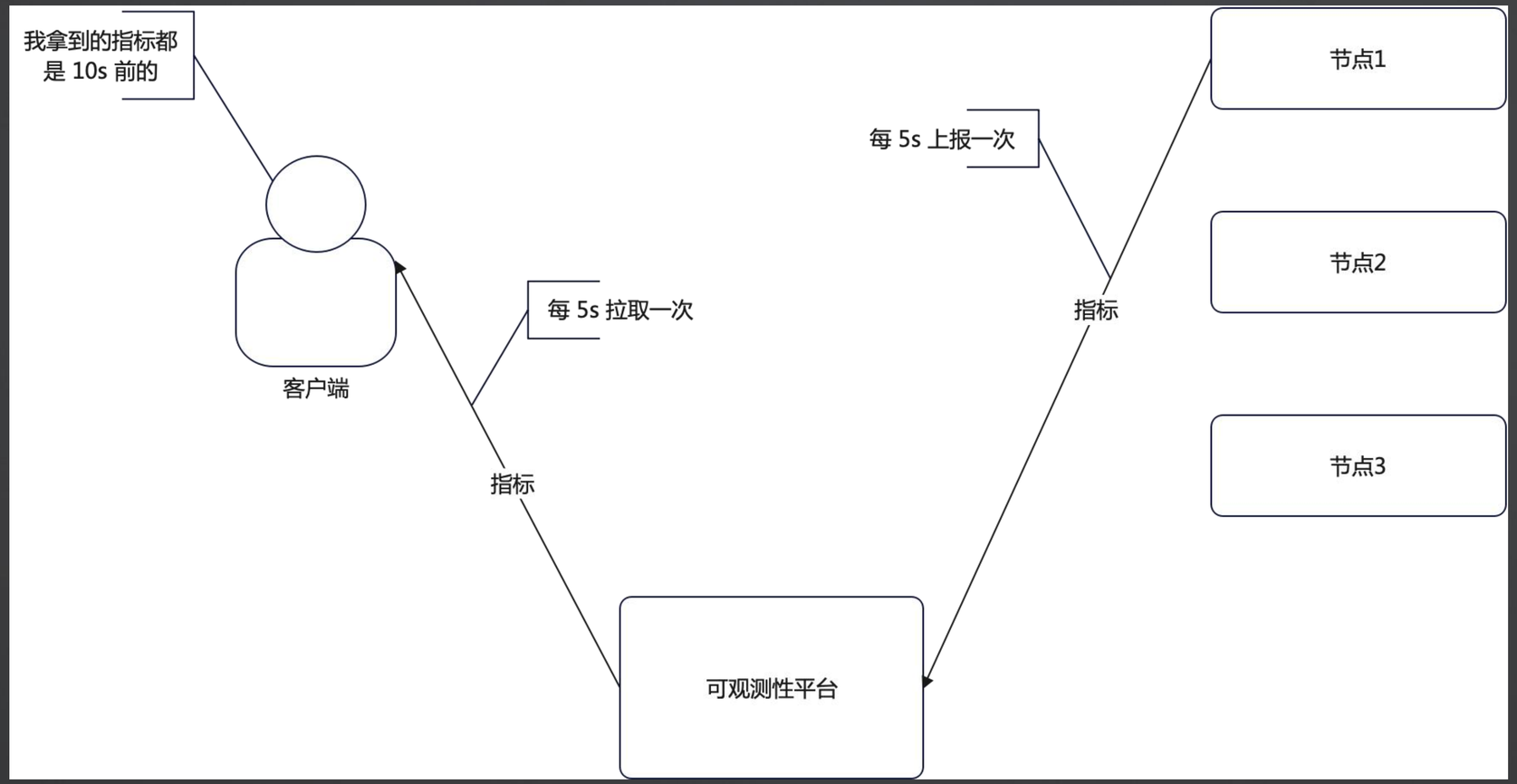
- 可观测性平台从所有的节点采集指标
- 客户端拉取指标（或者可观测性平台推送指标）
- 客户端使用这些指标来执行负载均衡

开发者可以根据指标和业务特征来设计负载均衡算法。



指标的时间敏感性

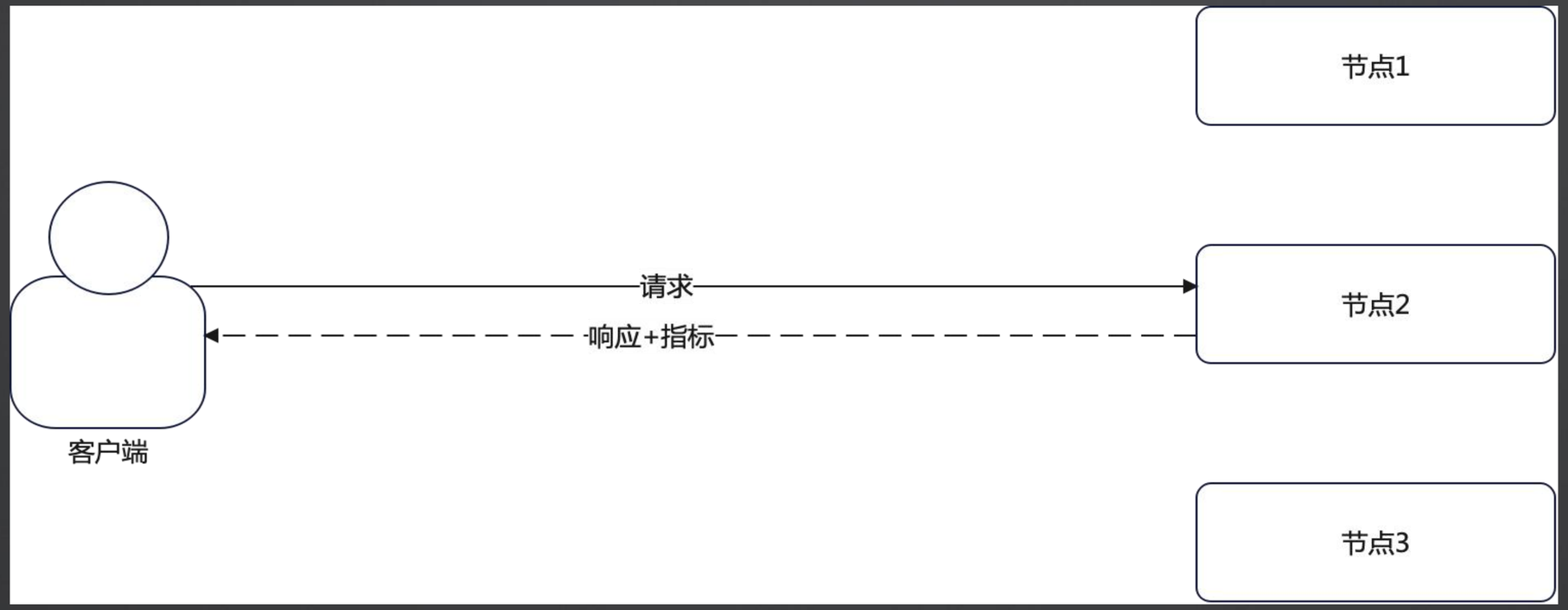
大多数指标都是时间敏感的。



指标的时间敏感性

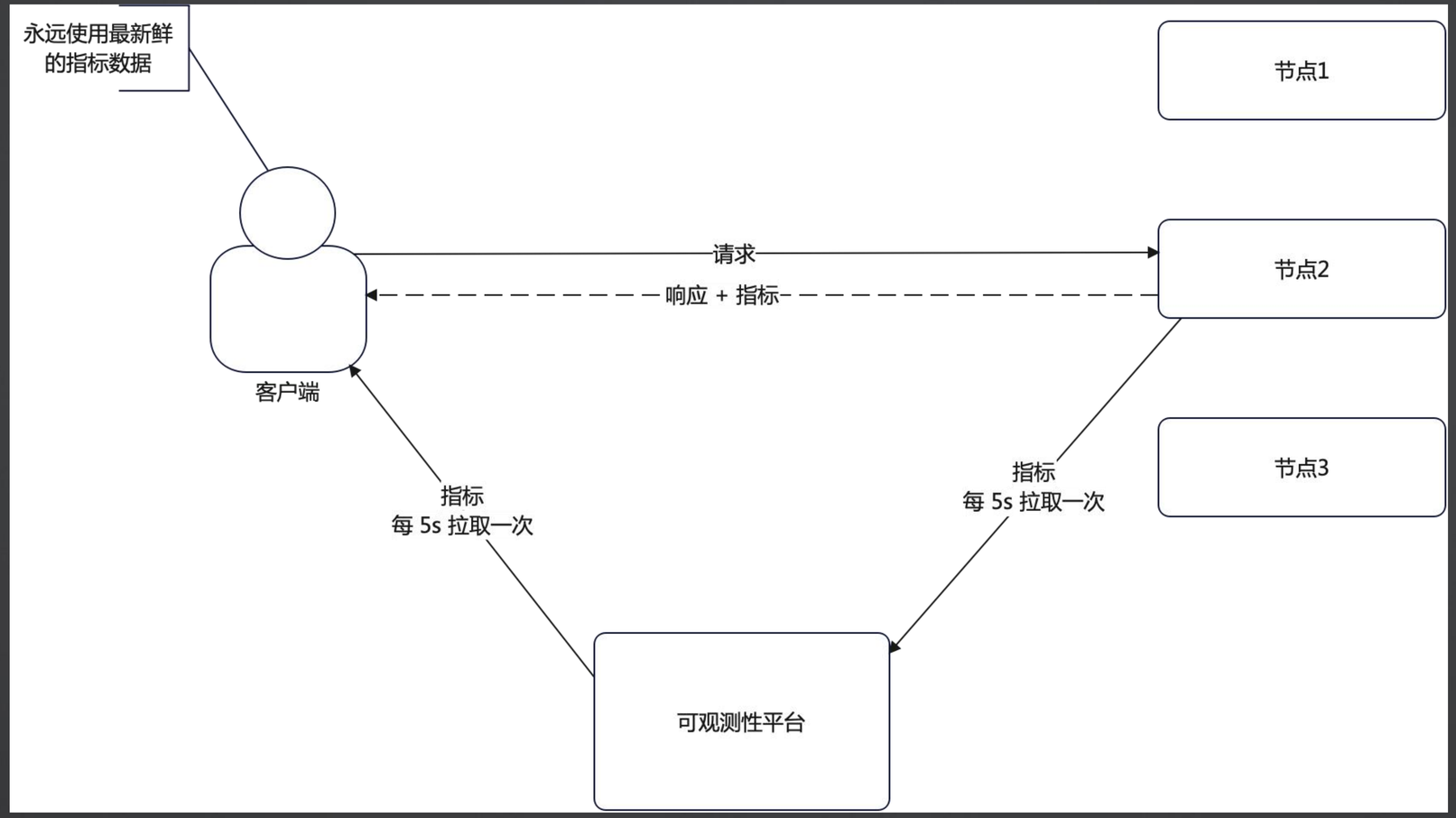
为了规避采集指标的延时问题，我们可以在返回响应的时候将指标一起返回。在高并发的环境下，我们用类似的策略解决健康检查（或者心跳）引起的网络性能问题。潜在问题是：

- 一些 RPC 协议不支持从服务端返回这一类的数据，例如在协议里面没有预留相应的字段。
- 如果客户端长期没有发送请求，那么它持有的数据都是很久以前的数据。

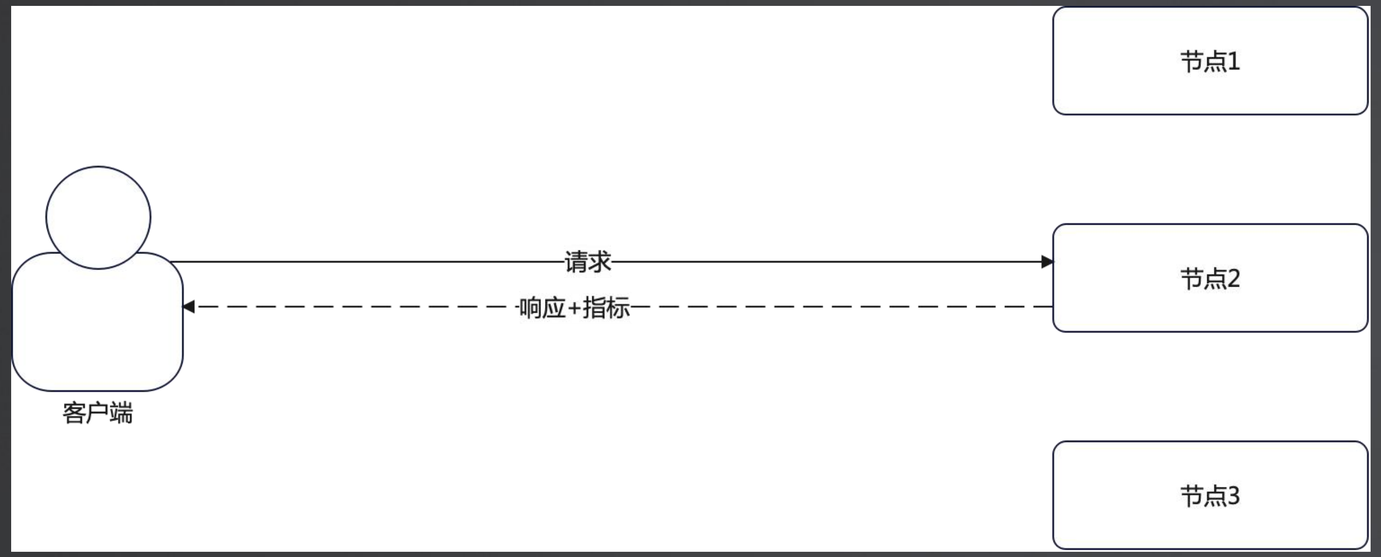


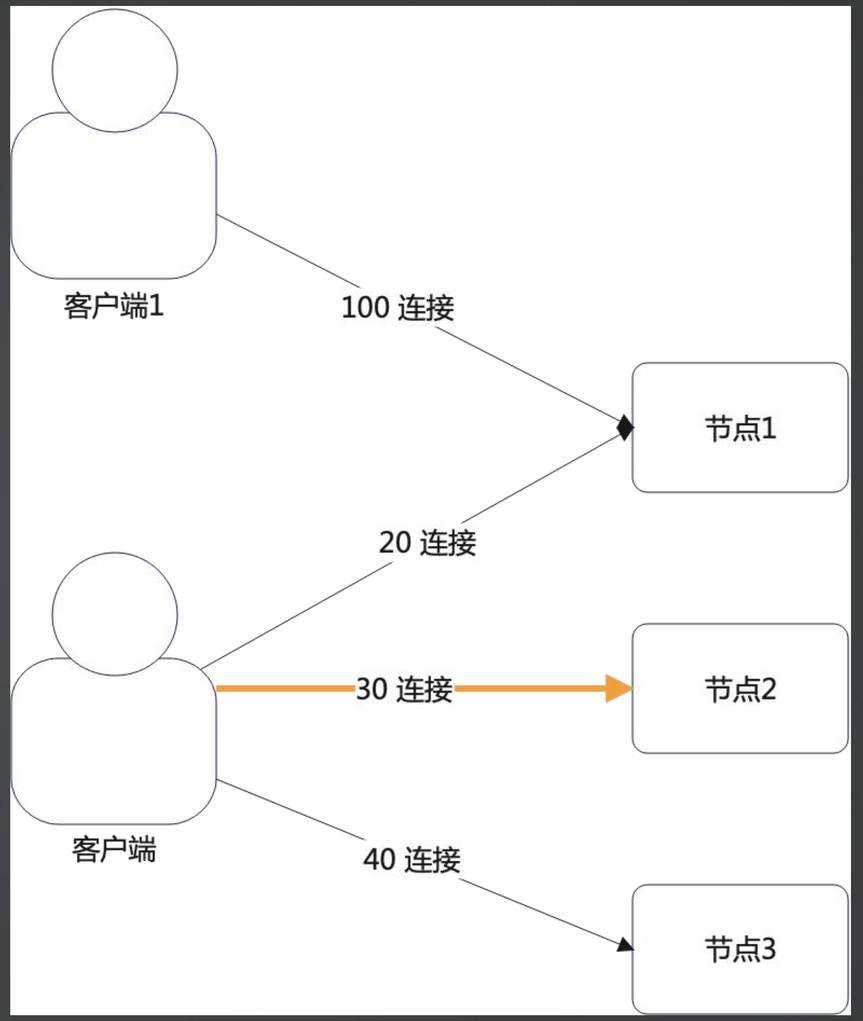
综合方案采集指标

两者结合能够有效利用两者的优点而且规避缺点。



综合方案采集指标



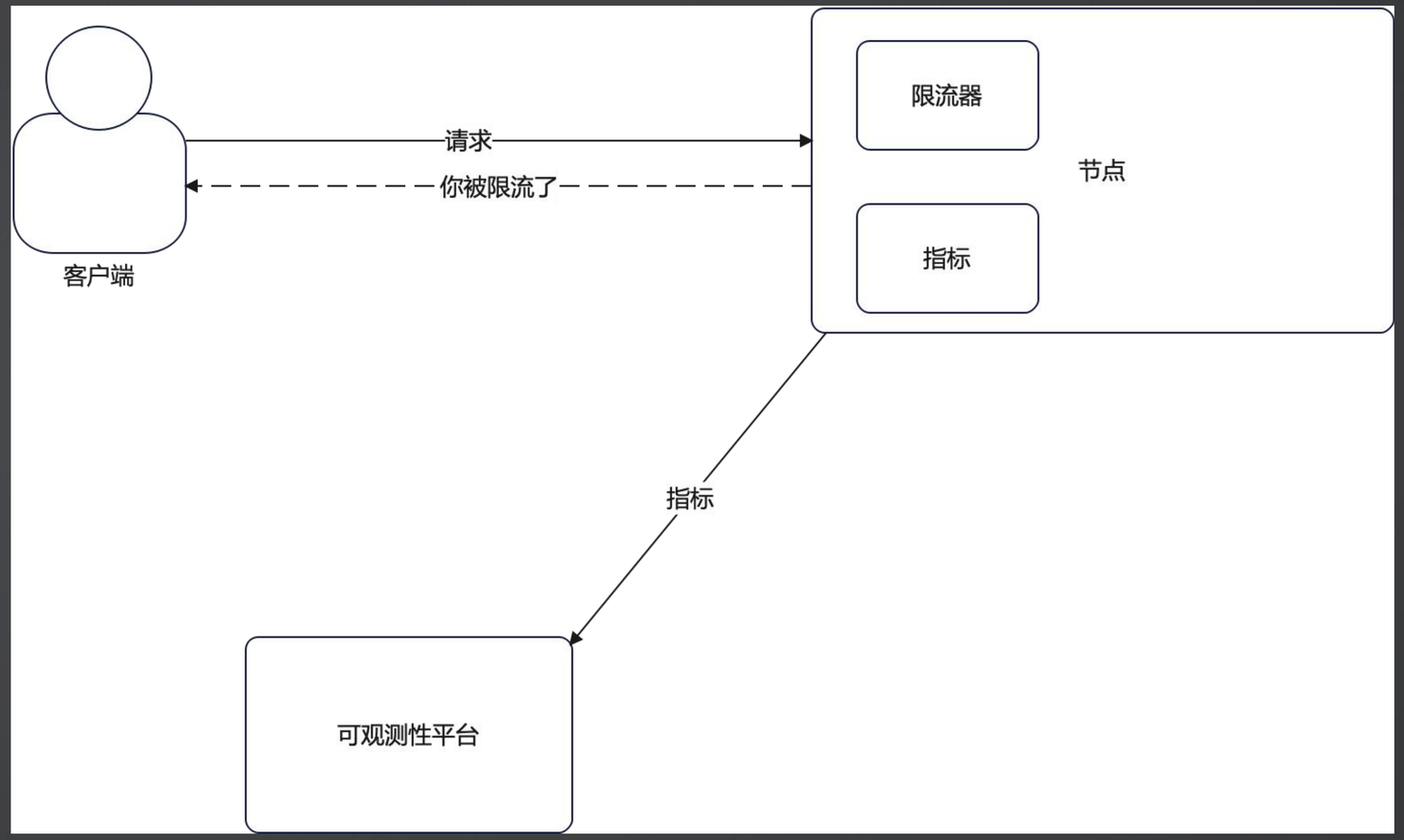


在这种机制之下，客户端可以从节点 1 中知晓它上面有 120 个连接，而节点 2 上面只有 30 个连接，从而做出正确的选择。

基于可观测性的服务治理：服务端

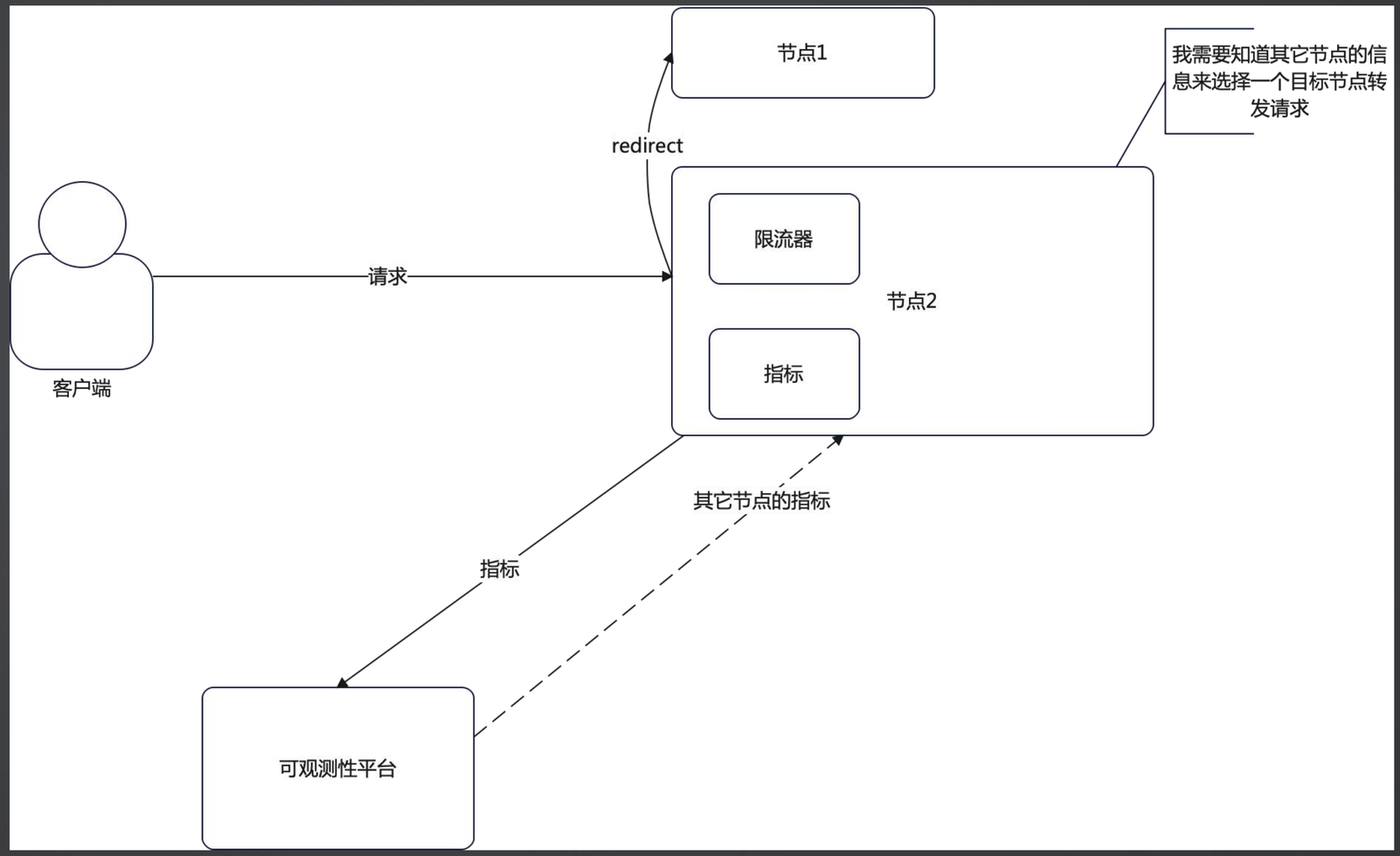
相比之下，服务端的治理要简单很多，不需要考虑那么复杂的时间敏感性问题。

因为它本身就有自己的全部信息，而且是实时信息。例如可以根据自身统计的响应时间、错误率、CPU 等来实时计算是否需要限流。



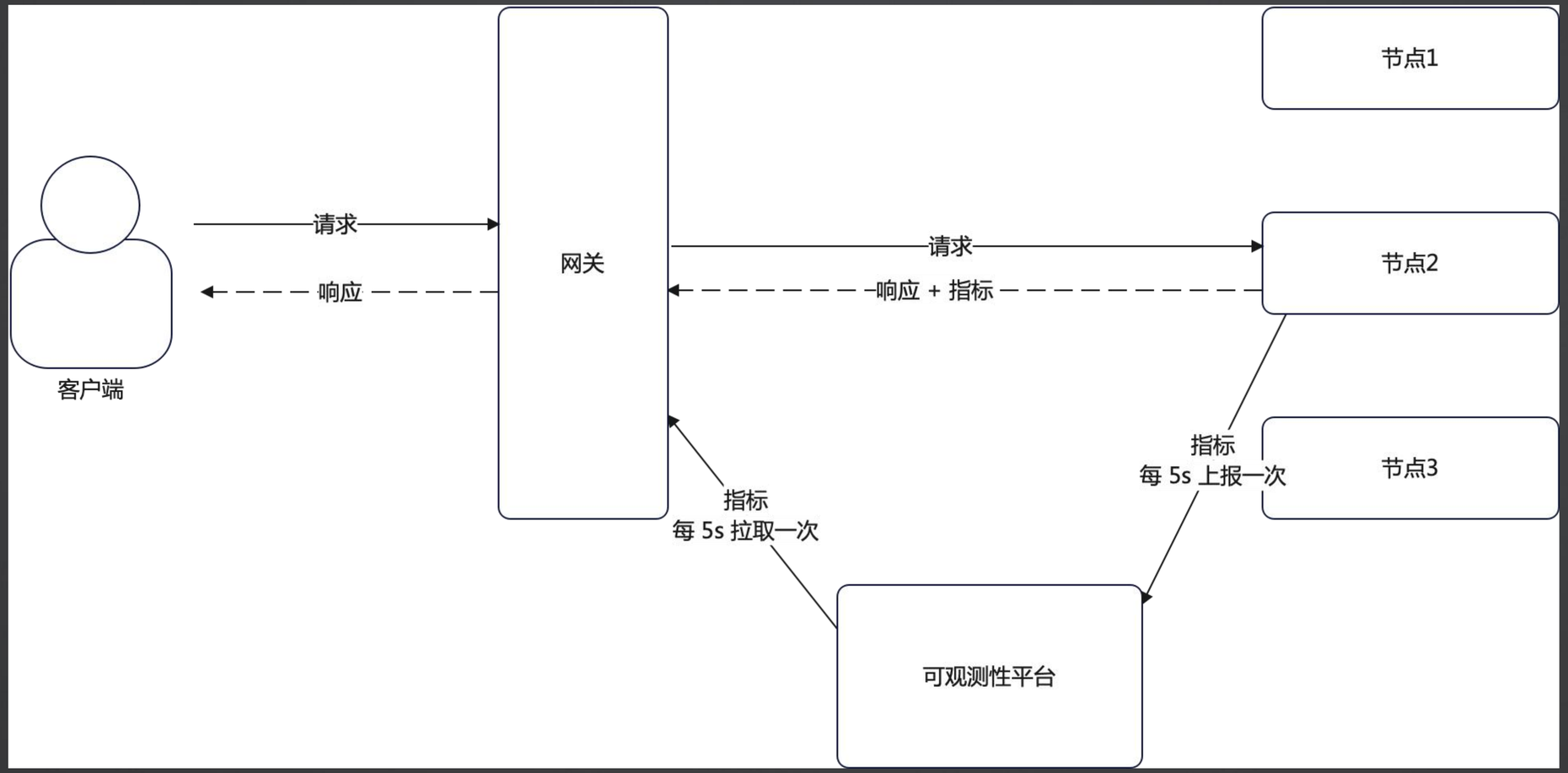
基于可观测性的服务治理：服务端

如果我们在服务端治理上采用了类似的策略，那么服务端还是要处理和客户端一样的问题。



微服务网关

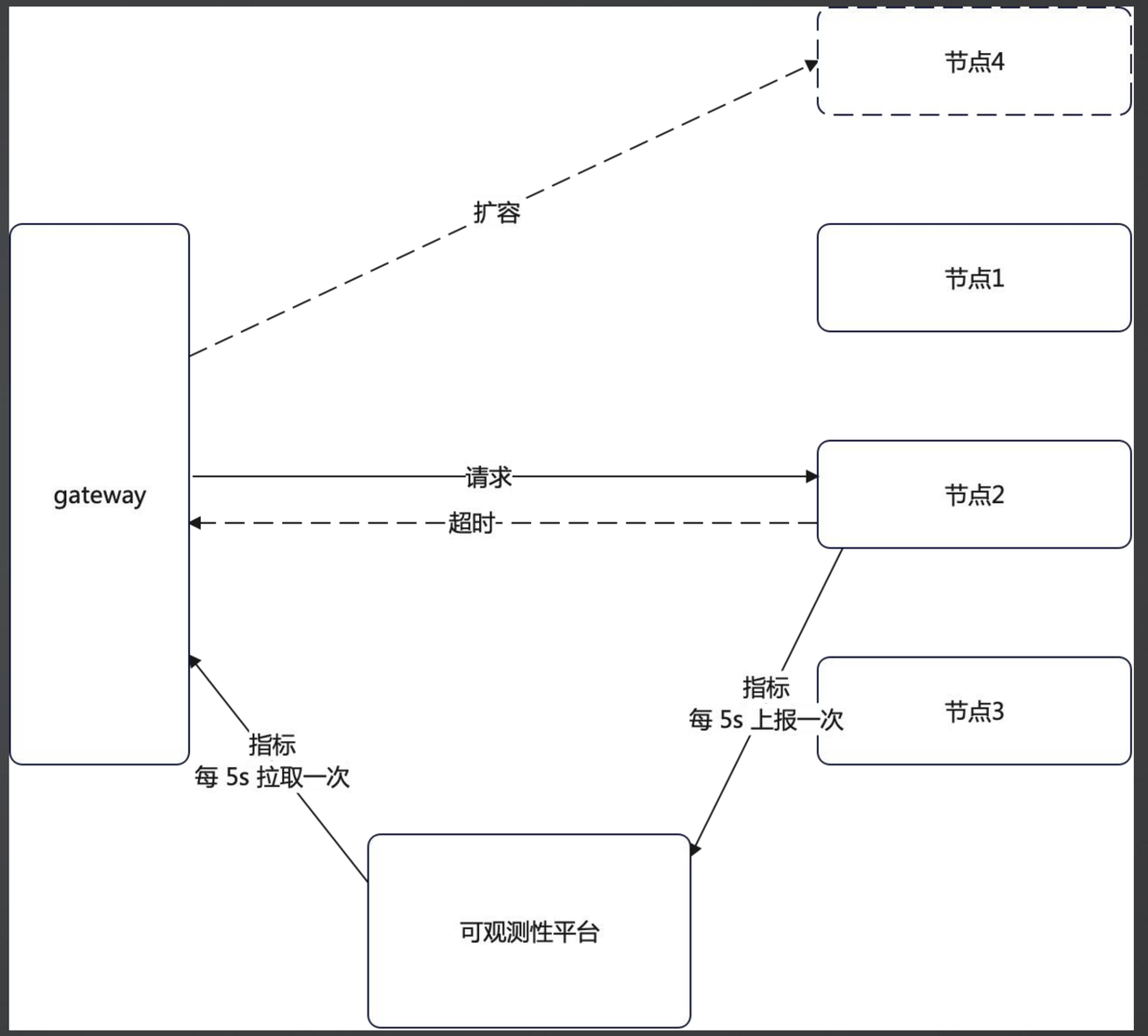
微服务网关也可以采用类似的设计。



集群扩容或者缩容

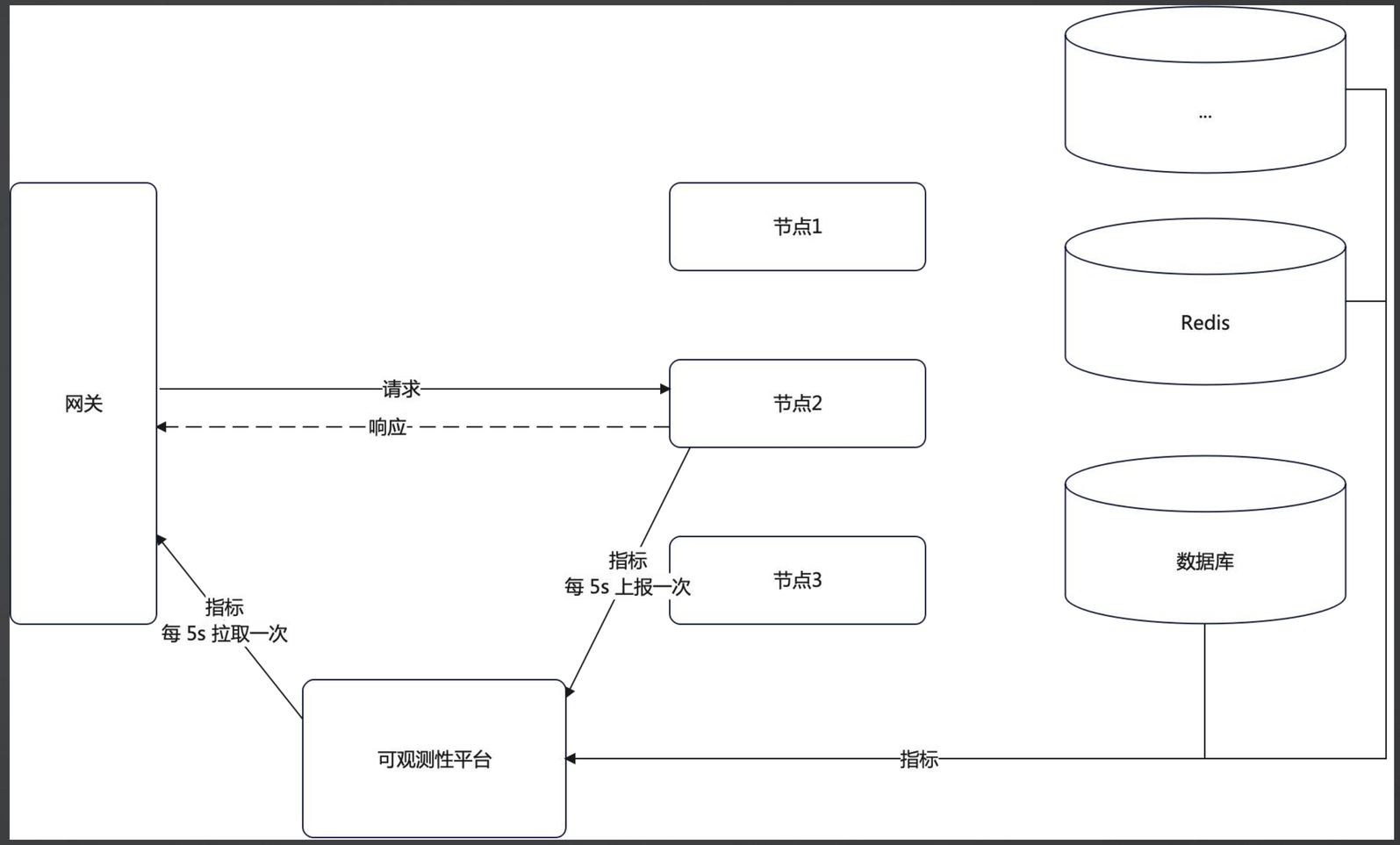
相应的集群，也可以通过这些采集的指标来判断是否需要执行扩容或者缩容。

这里我们假设的是由网关来判断，实际上可能是运维平台或者管理平台之类的地方来判断并且执行扩容或者缩容。



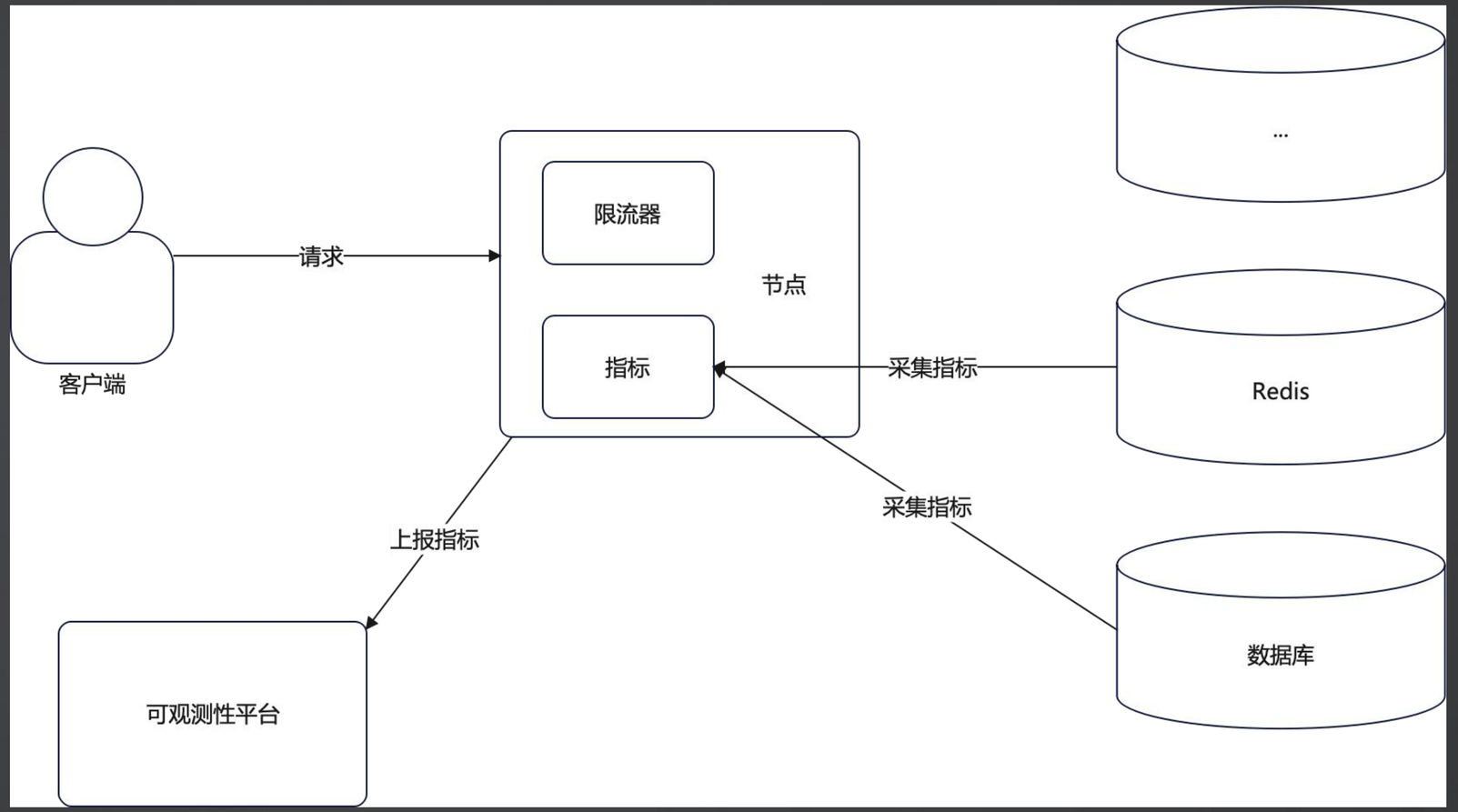
第三方组件指标

事实上，如果寻求一个万无一失的方案，那么还需要采集第三方组件的指标，例如系统使用的 Redis 集群、数据库集群……



服务端治理和第三方组件

服务端的治理，例如限流也可以利用第三方采集的指标，尤其是大部分系统的核心组件数据库的指标，非常具有参考意义。



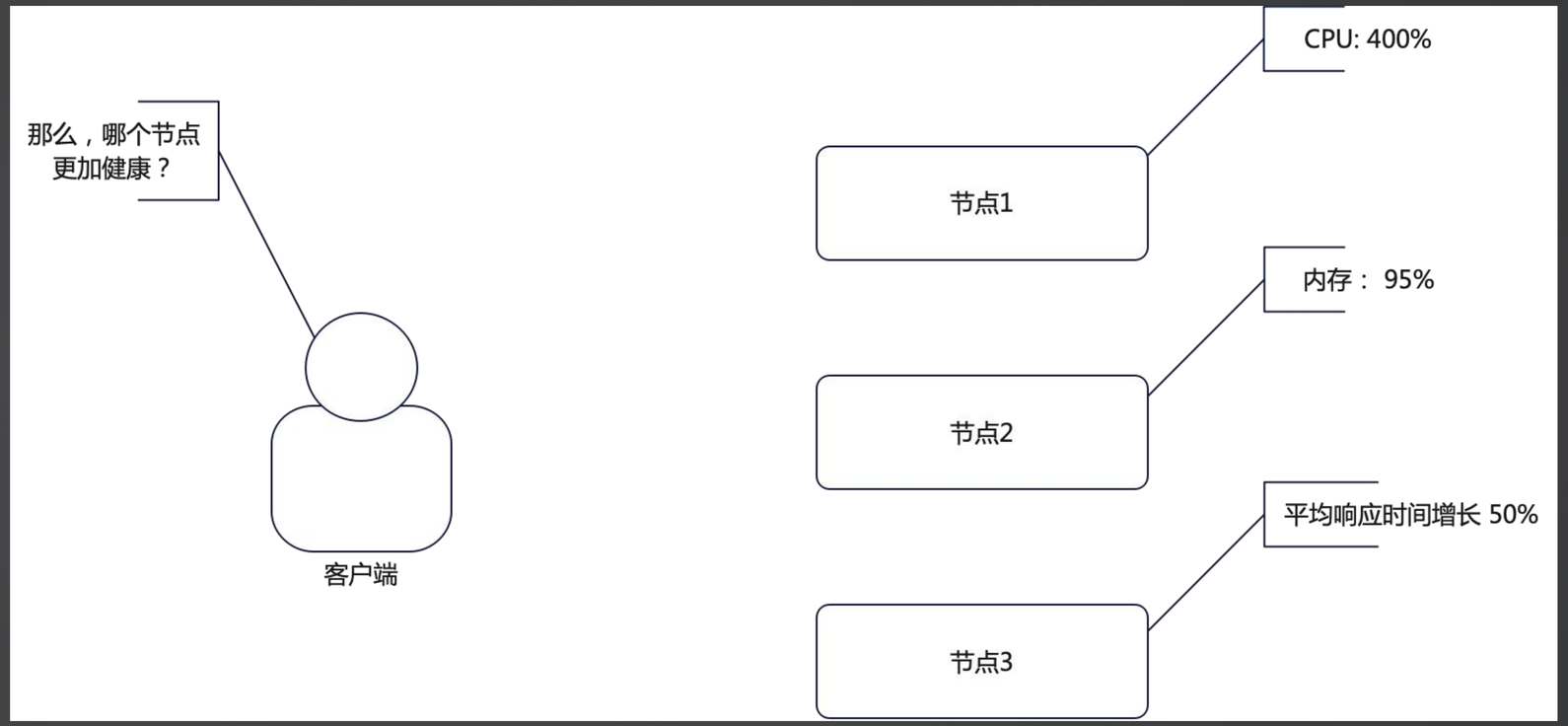
利用指标

问题是，如何使用这些指标？

答案是：水无常势，兵无定型，取决于你的业务特征。

大多数情况下，你选择几个关键的指标就可以了：

- CPU + 内存 + 网络 IO
- 响应时间或者错误率等



面试要点

可观测性在微服务面试里面，一般来说没有特别多值得深挖的。

- 可观测性的基本概念。
- 微服务应该采集哪些指标？这个地方引申出来就是你怎么解读这些指标，或者怎么利用这些指标来发现问题，这就需要你们在实际中积累经验。一般要注意两个方面：绝对值和趋势。例如平均响应时间，以及平均响应时间的变化趋势。
- 怎么把微服务可观测性和服务治理结合起来？说起来，大部分面试官是想不到可以尝试将可观测性平台和服务治理结合起来的，撑死了就是通过人手工监控观看数据。

另外，还有一个不在我们课程内容范围内但可能会引申出来的问题，那就是怎么做告警系统，或者说怎么为系统添加合适的告警手段。核心就是利用观测到的数据，设定各种告警阈值，例如设定错误率超过 1% 就告警。而后就是考虑告警的手段，比如说邮件、即时通讯还是电话。

一般来说，监控和告警都是一体的，准备面试时要把公司内部整个采集指标--监控--告警体系摸清楚。

Q & A

THANKS