协议开发流程

1 API介绍

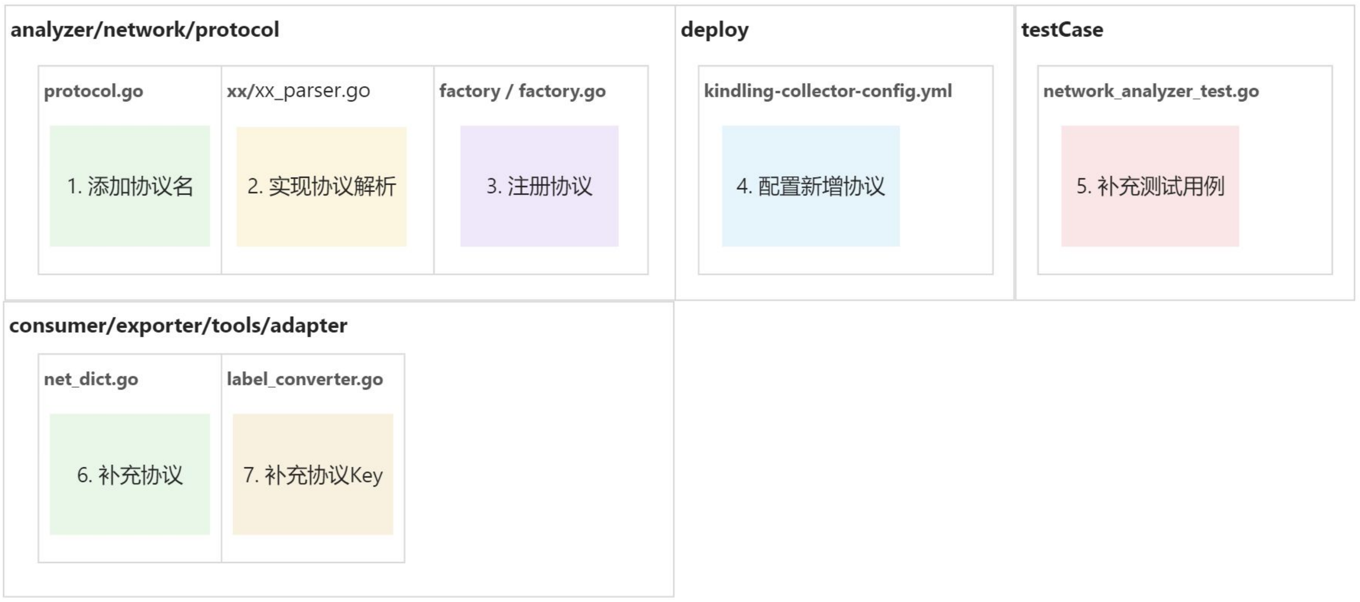
协议解析器

```go
// 协议解析器
type ProtocolParser struct {
    protocol          string
    ParseStreamHead   ParseStreamHeadFn
    ParseSequenceHead ParseSequenceHeadFn
    ParsePayload      ParsePayloadFn
}

// 声明有序协议解析器
func NewSequenceParser(protocol string, parseHead ParseSequenceHeadFn, parsePayload ParsePayloadFn) *ProtocolParser {
    return &ProtocolParser{
        protocol:          protocol,
        ParseSequenceHead: parseHead,
        ParsePayload:      parsePayload,
    }
}

// 声明流式协议解析器
func NewStreamParser(protocol string, parseHead ParseStreamHeadFn, parsePayload ParsePayloadFn) *ProtocolParser {
    return &ProtocolParser{
        protocol:        protocol,
        ParseStreamHead: parseHead,
        ParsePayload:    parsePayload,
    }
}

// 解析有序协议头信息
type ParseSequenceHeadFn func(data []byte, size int64, isRequest bool) (attributes ProtocolMessage)
// 解析流式协议头信息
type ParseStreamHeadFn func(data []byte, size int64, isRequest bool) (attributes ProtocolMessage, waitNextPkt bool)
// 解析报文内容
type ParsePayloadFn func(attributes ProtocolMessage) (ok bool)
```

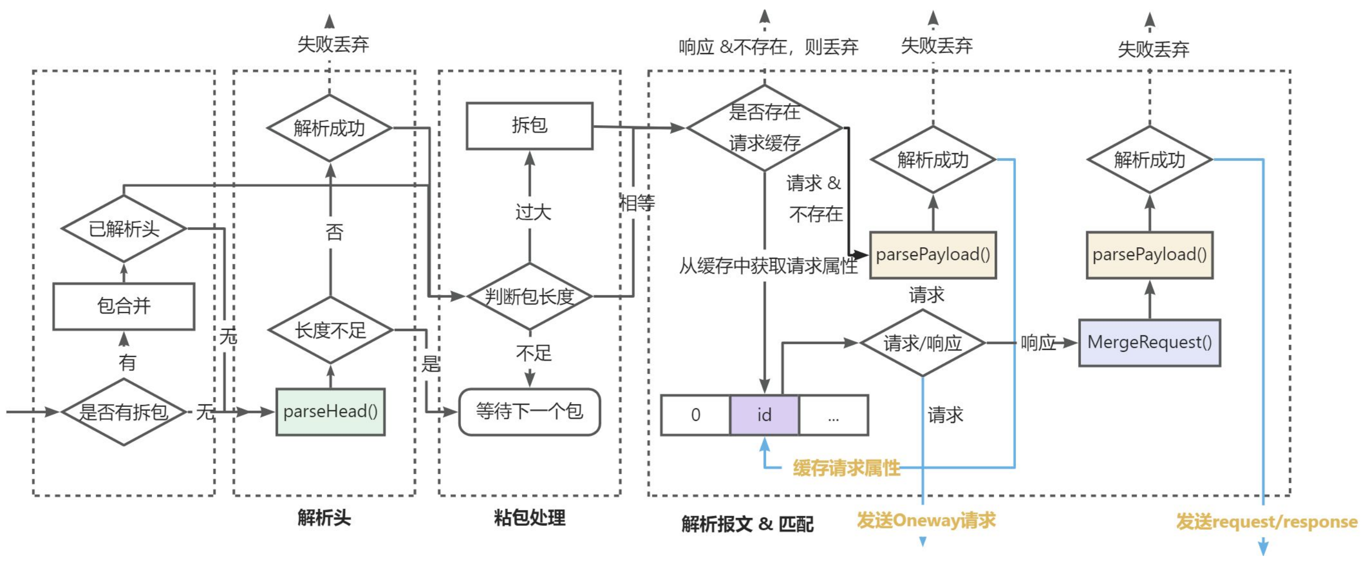
协议解析后属性

```go
// 协议解析后属性
type ProtocolMessage interface {
    SetData(newData []byte)
    GetData() []byte
    GetLength() int64
    IsRequest() bool
    IsReverse() bool
    GetStreamId() int64
    MergeRequest(request ProtocolMessage) bool
    GetAttributes() *model.AttributeMap
}

// 基础实现类，已覆写 3 ~ 8行函数
type PayloadMessage struct {
    Data    []byte
    Offset  int
    Size    int64
    Request bool
}
```

2 协议开发流程



2.1 流式协议开发 – Dubbo



2.1.1 添加协议名

```
const (
    ...
    DUBBO = "dubbo" // 此处替换具体协议名
    ...
)
```

2.1.2 创建协议

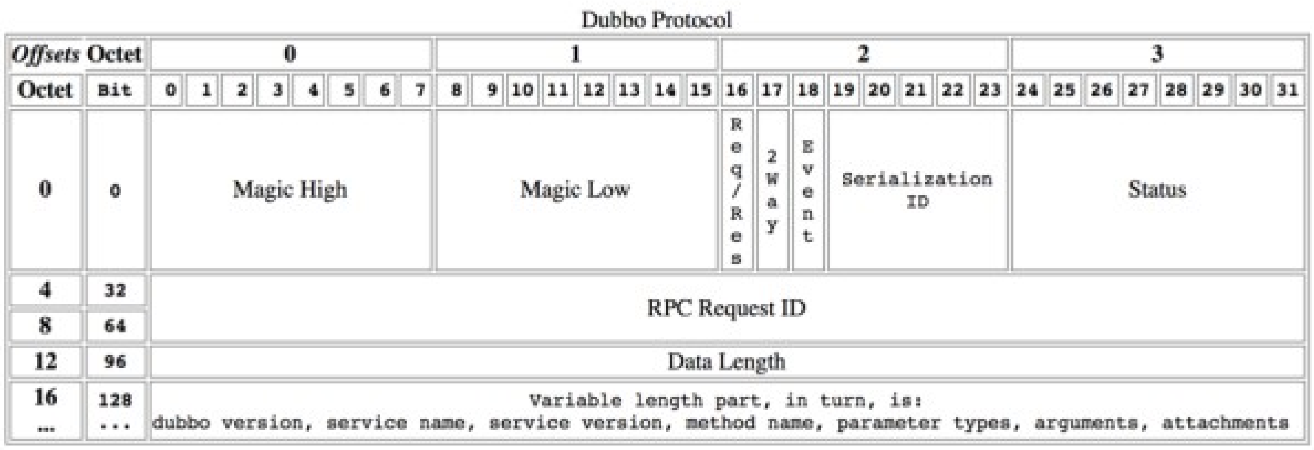
analyzer/network/protocol目录下创建文件夹dubbo，并创建子文件dubbo_parser.go，此处dubbo可替换为实际开发的协议名

```
analyzer/network/protocol/dubbo
└── dubbo_parser.go       协议解析器

// 如果实际开发过程中协议较为复杂，此处可多添加一些文件用于区分
// 从不同业务的报文可以声明 xx_aa.go、xx_bb.go等
```

2.1.2.1 定义协议解析器

```go
/*
  使用NewStreamParser()API 声明Dubbo协议解析器

  参数1: 协议名
  参数2: 提供解析头的函数
  参数3: 提供解析报文的函数
*/
func NewDubboParser() *protocol.ProtocolParser {
    return protocol.NewStreamParser(protocol.DUBBO, parseHead, parsePayload)
}
```

2.1.2.2 协议属性接口实现



Dubbo Protocol

Offsets Octet	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Magic High								Magic Low								Req/Res	2 Way	Event	Serialization ID					Status							
4	32	RPC Request ID																															
8	64																																
12	96	Data Length																															
16 ...	128 ...	Variable length part, in turn, is: dubbo version, service name, service version, method name, parameter types, arguments, attachments																															

从协议定义中，可提取出是否请求/响应、序列化方式、返回状态码、ID、长度等信息

```go
// 声明Dubbo协议解析后的属性，该类实现ProtocolMessage接口
type DubboAttributes struct {
    *protocol.PayloadMessage
    id          int64
    eventFlag   byte
    serialID    byte
    // 由service name + method name用于标识该请求的URL
    contentKey string
    status      int
}

func NewDubboAttributes(data []byte, isRequest bool, id int64, length uint32, eventFlag byte, serialID byte, status int) *DubboAttributes {
    return &DubboAttributes{
        // 将头长度、整个报文长度、请求/响应标识 传入到NewPayloadMessage中
        PayloadMessage: protocol.NewPayloadMessage(data, DubboHeadSize, int64(length), isRequest),
        id:             id,
        eventFlag:      eventFlag,
        serialID:       serialID,
        status:         status,
    }
}

/*
  由于引入PayloadMessage已实现部分接口，流式协议实现如下接口
*/
func (dubbo *DubboAttributes) GetStreamId() int64 {
    return dubbo.id
}

/*
  对于双工通信方式，需实现该接口
  基于解析协议传入的isRequest和报文头中的request标识解析出是否通信方向反转
*/
func (message *PayloadMessage) IsReverse() bool {
    return false
}

/*
  此外，由于存在返回报文解析依赖请求报文内容标识场景，此处需提供Request的属性合并到Respnonse中，用于Response报文解析
  Eg. Kafka的Reponse报文解析依赖Request报文中的API KEY 和 API VERSION
*/
func (dubbo *DubboAttributes) MergeRequest(request protocol.ProtocolMessage) bool {
    if request != nil {
        requestAttributes := request.(*DubboAttributes)
        if requestAttributes.id != dubbo.id {
            return false
        }
        dubbo.contentKey = request.(*DubboAttributes).contentKey
    }
    return true
}

/*
  提供协议解析后的属性
*/
func (dubbo *DubboAttributes) GetAttributes() *model.AttributeMap {
    attributeMap := model.NewAttributeMap()
    attributeMap.AddStringValue(constlabels.ContentKey, dubbo.contentKey)
    attributeMap.AddIntValue(constlabels.DubboErrorCode, int64(dubbo.status))
    if dubbo.status > 20 {
        attributeMap.AddBoolValue(constlabels.IsError, true)
        attributeMap.AddIntValue(constlabels.ErrorType, int64(constlabels.ProtocolError))
    }
    return attributeMap
}
```

2.1.2.3 解析Dubbo头信息

基于提供的协议规范，解析网络抓包的数据。

- 前2个byte为魔数，可用于识别是否为Dubbo协议
- 第3个byte包含Req/Resp、序列化方式等信息，可用于解析协议中判断是否合法报文。
- 第4个byte用于返回报文的错误码
- 第16个byte开始需通过指定的序列化方式解析报文内容，service name + method name可用于contentKey标识该请求的URL

0000	da bb c2 00 00 00 00 00 00 00 00 01 00 00 01 62	..............b	Magic High	da
0010	05 32 2e 36 2e 32 30 2a 69 6f 2e 6b 69 6e 64 6c	.2.6.20*io.kindl	Magic Low	bb
0020	69 6e 67 2e 64 75 62 62 6f 2e 61 70 69 2e 73 65	ing.dubbo.api.se	Req	true c2=>11000010
0030	72 76 69 63 65 2e 4f 72 64 65 72 53 65 72 76 69	rvice.OrderServi	2 Way	true
0040	63 65 05 30 2e 30 2e 30 05 6f 72 64 65 72 30 22	ce.0.0.0.order0"	Event	false
0050	4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e	Ljava/lang/Strin	SerializationID	2
0060	67 3b 4c 6a 61 76 61 2f 75 74 69 6c 2f 4c 69 73	g;Ljava/util/Lis	Status	0
0070	74 3b 04 54 65 73 74 7a 43 30 22 69 6f 2e 6b 69	t;.TestzC0"io.ki	RPC Request ID	1
0080	6e 64 6c 69 6e 67 2e 64 75 62 62 6f 2e 61 70 69	ndling.dubbo.api	Data Length	354 (0x0162)
0090	2e 62 65 61 6e 2e 50 72 6f 64 75 63 74 95 04 74	.bean.Product..t	------------------------	--------------------
00a0	69 6d 65 06 61 63 74 69 76 65 04 63 6f 73 74 04	ime.active.cost.	**dubbo version**	2.6.2
00b0	6e 61 6d 65 02 69 64 60 4a 00 00 01 80 f5 bf cc	name.id`J.......	**service name**	io.kindling.dubbo.api.service.OrderService
00c0	41 46 5e 01 9d 0b 54 65 73 74 44 61 74 61 34 31	AF^...TestData41	**serivce version**	0.0.0
00d0	33 e0 60 4a 00 00 01 80 f5 bf cc 41 46 5e 00 ec	3.`J.......AF^..	**method name**	order
00e0	0b 54 65 73 74 44 61 74 61 32 33 36 e1 48 04 70	.TestData236.H.p	**parameter types**	Ljava/lang/String;Ljava/util/List;
00f0	61 74 68 30 2a 69 6f 2e 6b 69 6e 64 6c 69 6e 67	ath0*io.kindling	**arguments**	...
0100	2e 64 75 62 62 6f 2e 61 70 69 2e 73 65 72 76 69	.dubbo.api.servi	**attachments**	
0110	63 65 2e 4f 72 64 65 72 53 65 72 76 69 63 65 09	ce.OrderService.	**path**	io.kindling.dubbo.api.service.OrderService
0120	69 6e 74 65 72 66 61 63 65 30 2a 69 6f 2e 6b 69	interface0*io.ki	**interface**	io.kindling.dubbo.api.service.OrderService
0130	6e 64 6c 69 6e 67 2e 64 75 62 62 6f 2e 61 70 69	ndling.dubbo.api	**version**	0.0.0
0140	2e 73 65 72 76 69 63 65 2e 4f 72 64 65 72 53 65	.service.OrderSe	**timeout**	30000
0150	72 76 69 63 65 07 76 65 72 73 69 6f 6e 05 30 2e	rvice.version.0.		
0160	30 2e 30 07 74 69 6d 65 6f 75 74 05 33 30 30 30	0.0.timeout.3000		
0170	30 5a	0Z		

```go
const (
    // magic header
    MagicHigh = byte(0xda)
    MagicLow  = byte(0xbb)

    DubboHeadSize = 16
)

/*
  声明parseHead函数

  参数1: 完整报文
  参数2: 抓包的报文大小，用于标识报文长度。流式协议使用该字段判断头报文是否完整，能够解析
  参数3: 是否请求，由初次连接时决定。
         对于流式双工通信模型需缓存该字段，如果出现方向倒置即另一端发送请求，此时通过该字段和识别出的请求类型标识IsReverse()
*/
func parseHead(data []byte, size int64, isRequest bool) (attributes protocol.ProtocolMessage, waitNextPkt bool) {
    // 基于传入的size 判断头报文长度，如果小于则返回true
    if size < DubboHeadSize {
        return nil, true
    }
    // 魔术判断
    if len(data) < DubboHeadSize || data[0] != MagicHigh || data[1] != MagicLow {
        return
    }

    serialID := data[2] & SerialMask
    if serialID == Zero {
        return
    }

    // 请求标识匹配
    requestFlag := data[2] & FlagRequest
    if isRequest && requestFlag == Zero {
        return
    }

    // 响应标识匹配
    if !isRequest && requestFlag != Zero {
        return nil
    }

    // 读取其它头信息
    status := int(data[3])
    id, _ := protocol.ReadInt64(data, 4)
    length, _ := protocol.ReadUInt32(data, 12)
    attributes = NewDubboAttributes(data, isRequest, id, length + DubboHeadSize, data[2], serialID, status)
    return
}
```

2.1.2.4 解析Dubbo报文

```go
func parsePayload(attributes protocol.ProtocolMessage) (ok bool) {
    if isRequest {
        message := attributes.(*DubboAttributes)
        // 解析请求的URL
        message.contentKey = message.getContentKey()
    }
    return true
}

func (dubbo *DubboAttributes) getContentKey() string {
    if (dubbo.eventFlag & FlagEvent) != Zero {
        return "Heartbeat"
    }
    if (dubbo.eventFlag & FlagTwoWay) == Zero {
        // Ignore Oneway Data
        return "Oneway"
    }

    serializer := GetSerializer(dubbo.serialID)
    if serializer == serialUnsupport {
        // Unsupport Serial. only support hessian and fastjson.
        return "UnSupportSerialFormat"
    }

    var (
        service string
        method  string
    )
    requestData := dubbo.Data
    offset := serializer.eatString(requestData, 16)

    // service name
    offset, service = serializer.getStringValue(requestData, offset)
    // service version
    offset = serializer.eatString(requestData, offset)
    // method name
    _, method = serializer.getStringValue(requestData, offset)

    return service + "#" + method
}
```

2.1.2.5 dubbo2_serialize.go

由于dubbo2内置了多套序列化方式，先定义接口dubbo2Serializer dubbo2Serializer Go 复制代码

```go
type dubbo2Serializer interface {
    eatString(data []byte, offset int) int

    getStringValue(data []byte, offset int) (int, string)
}
```

dubbo2默认的序列化方式是hessian2，此处实现hessian2方式

dubbo2Hessian Go 复制代码

```go
type dubbo2Hessian struct{}

func (dh *dubbo2Hessian) eatString(data []byte, offset int) int {
	dataLength := len(data)
	if offset >= dataLength {
		return dataLength
	}

	tag := data[offset]
	if tag >= 0x30 && tag <= 0x33 {
		if offset+1 == dataLength {
			return dataLength
		}
		// [x30-x34] <utf8-data>
		return offset + 2 + int(tag-0x30)<<8 + int(data[offset+1])
	} else {
		return offset + 1 + int(tag)
	}
}

func (dh *dubbo2Hessian) getStringValue(data []byte, offset int) (int, string) {
	dataLength := len(data)
	if offset >= dataLength {
		return dataLength, ""
	}

	var stringValueLength int
	tag := data[offset]
	if tag >= 0x30 && tag <= 0x33 {
		if offset+1 == dataLength {
			return dataLength, ""
		}
		// [x30-x34] <utf8-data>
		stringValueLength = int(tag-0x30)<<8 + int(data[offset+1])
		offset += 2
	} else {
		stringValueLength = int(tag)
		offset += 1
	}

	if offset+stringValueLength >= len(data) {
		return dataLength, string(data[offset:])
	}

```

```go
    return offset + stringValueLength, string(data[offset : offset+stringValueLength])
}
```

对外暴露公共方法，用于获取序列化方式

GetSerializer()

```go
var (
    serialHessian2   = &dubbo2Hessian{}
    serialUnsupport = &dubbo2Unsupport{}
)

func GetSerializer(serialID byte) dubbo2Serializer {
    switch serialID {
    case SerialHessian2:
        return serialHessian2
    default:
        return serialUnsupport
    }
}
```

2.1.3 注册dubbo2解析器

在factory.go中注册dubbo2协议的解析器

```go
func NewParserFactory(options ...Option) *ParserFactory {
    ...
    factory.protocolParsers[protocol.DUBBO] = dubbo.NewDubboParser()
    ...
}
```

2.1.4 声明支持协议

在deploy/kindling-collector-config.yml中声明dubbo2协议

kindling-collector-config.yml   Go   复制代码

```yaml
analyzers:
  networkanalyzer:
    protocol_parser: [ http, mysql, dns, redis, kafka, dubbo ]
    protocol_config:
      - key: "dubbo"
        payload_length: 200
```

2.1.5 Dubbo测试用例

在analyzer/network/network_analyzer_test.go中补充Dubbo协议测试用例

在analyzer/network/protocol/testdata/dubbo/下补充测试数据

```go
func TestDubboProtocol(t *testing.T) {
    testProtocol(t, "dubbo/server-event.yml",
        "dubbo/server-trace-short.yml")
}
```

2.1.6 Dubbo协议映射

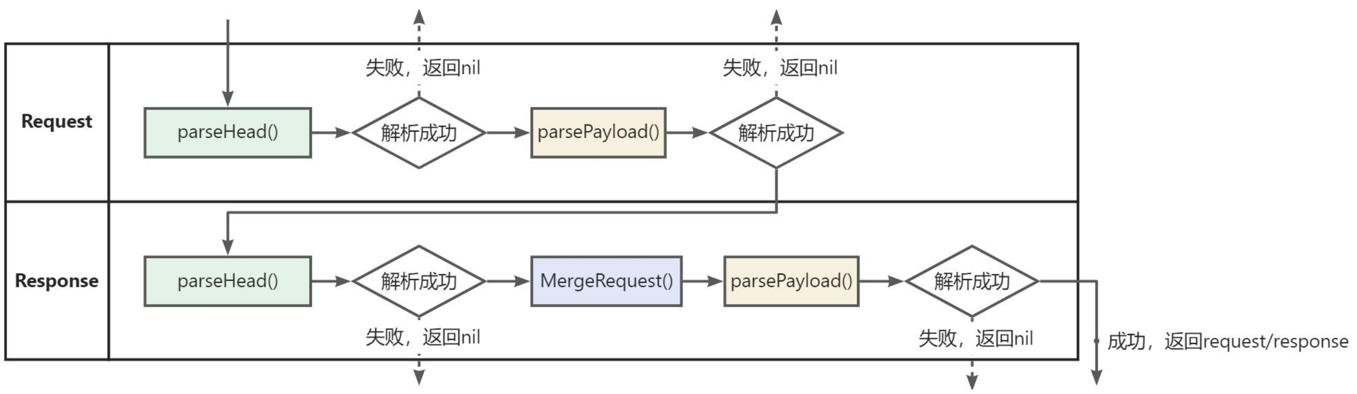
consumer/exporter/tools/adapter/net_dict.go

```go
const (
    empty Protocol = iota
    ...
    DUBBO
    ...
)
```

consumer/exporter/tools/adapter/labels_converter.go

```go
func updateProtocolKey(key *extraLabelsKey, labels *model.AttributeMap) *extraLabelsKey {
    ...
    case constvalues.ProtocolDubbo:
        key.protocol = DUBBO
    ...
}
```

2.2 非流式协议开发 – HTTP



2.2.1 添加协议名

```
const (
    ...
    HTTP = "http" // 此处替换具体协议名
    ...
)
```

2.2.2 创建协议

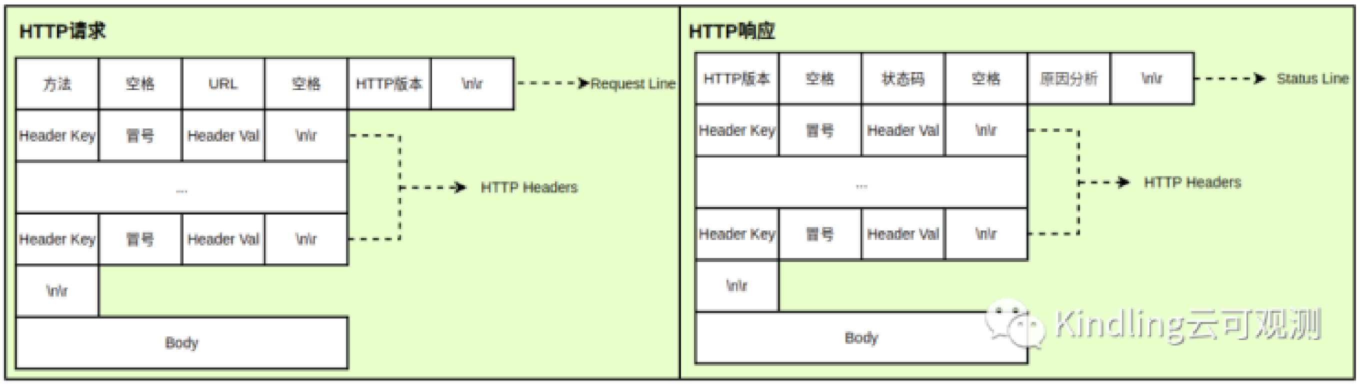
analyzer/network/protocol目录下创建文件夹http，并创建子文件http_parser.go，此处http可替换为实际开发的协议名

```
analyzer/network/protocol/http
└── http_parser.go        协议解析器
```

2.2.2.1 定义协议解析器

```
/*
  使用NewSequenceParser()API 声明Http协议解析器

  参数1: 协议名
  参数2: 提供解析头的函数
  参数3: 提供解析报文的函数
*/
func NewHttpParser() *protocol.ProtocolParser {
    return protocol.NewSequenceParser(protocol.HTTP, parseHead, parsePayload)
}
```

2.2.2.2 协议属性接口实现



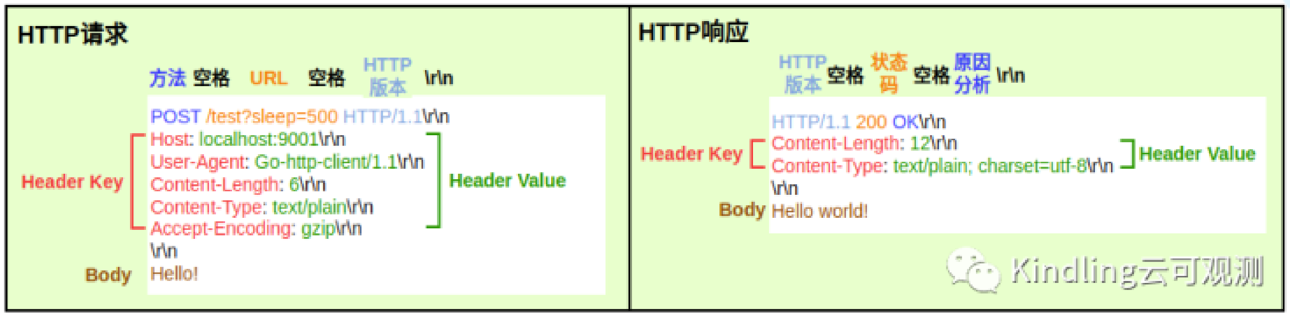
从协议定义中，可提取出方法名、URL、返回状态码、Header等信息

```go
// 声明HTTP协议解析后的属性，该类实现ProtocolMessage接口
type HttpAttributes struct {
	*protocol.PayloadMessage
	method     string
	url        string
	contentKey string
	traceType  string
	traceId    string
	statusCode int64
}

// 提供Request头解析后创建HttpAttributes的API
func NewHttpRequestAttributes(data []byte, size int64, method string, url string, contentKey string) *HttpAttributes {
	return &HttpAttributes{
		PayloadMessage: protocol.NewPayloadMessage(data, 0, size, true),
		method:         method,
		url:            url,
		contentKey:     contentKey,
	}
}

// 提供Response头解析后创建HttpAttributes的API
func NewHttpResponseAttributes(data []byte, size int64, statusCode int64) *HttpAttributes {
	return &HttpAttributes{
		PayloadMessage: protocol.NewPayloadMessage(data, 0, size, false),
		statusCode:     statusCode,
	}
}

/*
  由于引入PayloadMessage已实现部分接口，非流式协议只需实现如下2个接口

  将Request解析后的属性复制到Respone解析属性中
*/
func (http *HttpAttributes) MergeRequest(request protocol.ProtocolMessage) bool {
	if request != nil {
		requestAttributes := request.(*HttpAttributes)
		http.method = requestAttributes.method
		http.url = requestAttributes.url
		http.contentKey = requestAttributes.contentKey
		http.traceType = requestAttributes.traceType
		http.traceId = requestAttributes.traceId
	}
	return true
}

/*
  提供协议解析后的属性
*/
func (http *HttpAttributes) GetAttributes() *model.AttributeMap {
    attributeMap := model.NewAttributeMap()
    attributeMap.AddStringValue(constlabels.HttpMethod, http.method)
    attributeMap.AddStringValue(constlabels.HttpUrl, http.url)
    attributeMap.AddStringValue(constlabels.ContentKey, http.contentKey)
    if len(http.traceType) > 0 && len(http.traceId) > 0 {
        attributeMap.AddStringValue(constlabels.HttpApmTraceType, http.traceType)
        attributeMap.AddStringValue(constlabels.HttpApmTraceId, http.traceId)
    }
    attributeMap.AddIntValue(constlabels.HttpStatusCode, http.statusCode)
    if http.statusCode >= 400 {
        attributeMap.AddBoolValue(constlabels.IsError, true)
        attributeMap.AddIntValue(constlabels.ErrorType, int64(constlabels.ProtocolError))
    }
    return attributeMap
}
```

2.2.2.3 解析HTTP头信息

基于提供的协议规范，解析网络抓包的数据。

- 请求行 – 方法、URL信息
- HTTP头信息 – traceId信息
- 状态行 – 状态码信息



```
POST /test?sleep=500 HTTP/1.1\r\n
Host: localhost:9001\r\n
User-Agent: Go-http-client/1.1\r\n
Content-Length: 6\r\n
Content-Type: text/plain\r\n
Accept-Encoding: gzip\r\n
\r\n
Hello!
```

```
HTTP/1.1 200 OK\r\n
Content-Length: 12\r\n
Content-Type: text/plain; charset=utf-8\r\n
\r\n
Hello world!
```

```go
func parseHead(payload []byte, size int64, isRequest bool) (attributes protocol.ProtocolMessage) {
	/*
	   方法 + URL + HTTP/1.1\r\n
	   HTTP/1.1 + 状态码 + 原因\r\n
	 */
	if len(payload) < 14 {
		return nil
	}

	if isRequest {
		return parseRequestHead(payload, size)
	} else {
		return parseResponseHead(payload, size)
	}
}

func parseRequestHead(payload []byte, size int64) (attributes protocol.ProtocolMessage) {
	var (
		method []byte
		url    []byte
	)
	/*
		Request line
			Method [GET/POST/PUT/DELETE/HEAD/TRACE/OPTIONS/CONNECT]
			Blank
			Request-URI [eg. /xxx/yyy?parm0=aaa&param1=bbb]
			Blank
			HTTP-Version [HTTP/1.0 | HTTP/1.2]
			\r\n

		Request header
	*/
	offset := 0
	offset, method = protocol.ReadUntilBlankWithLength(payload, offset, 8)
	if !httpMethodsList[string(method)] {
		if payload[offset-1] != ' ' || payload[offset] != '/' {
			return nil
		}
		// FIX ET /xxx Data with split payload.
		if replaceMethod, ok := splitMethodsList[string(method)]; ok {
			method = replaceMethod
		} else {
			return nil
		}
	}
```

```go
    _, url = protocol.ReadUntilBlank(payload, offset)
    contentKey := clusteringMethod.Clustering(string(url))
    if len(contentKey) == 0 {
        contentKey = "*"
    }
    return NewHttpRequestAttributes(payload, size, string(method), tools.FormatByteArrayToUtf8(url), tools.FormatStringToUtf8(contentKey))
}

func parseResponseHead(payload []byte, size int64) (attributes protocol.ProtocolMessage) {
    var (
        version     []byte
        statusCodeI int64
        err         error
    )
    /*
        Status line
            HTTP-Version[HTTP/1.0 | HTTP/1.1]
            Blank
            Status-Code
            Blank
            Reason-Phrase
            \r\n

        Response header
    */
    offset := 0
    offset, version = protocol.ReadUntilBlankWithLength(payload, offset, 9)
    if !httpVersoinList[string(version)] || payload[offset-1] != ' ' {
        return nil
    }

    _, statusCode := protocol.ReadUntilBlankWithLength(payload, offset, 6)
    if statusCodeI, err = strconv.ParseInt(string(statusCode), 10, 0); err != nil {
        return nil
    }

    if statusCodeI > 999 || statusCodeI < 99 {
        statusCodeI = 0
    }
    return NewHttpResponseAttributes(payload, size, statusCodeI)
}
```

2.2.2.4 解析HTTP报文

```go
func parsePayload(attributes protocol.ProtocolMessage) (ok bool) {
    message := attributes.(*HttpAttributes)
    // 考虑到APM会添加Trace头信息，解析出Trace信息
    if len(message.traceId) == 0 || len(message.traceType) == 0 {
        // 当Request中已有Trace信息，Response无需再解析
        message.parseTraceHeader()
    }
    return true
}
```

2.2.3 注册HTTP解析器

在factory.go中注册http协议的解析器

Go 复制代码

```go
func NewParserFactory(options ...Option) *ParserFactory {
    ...
    factory.protocolParsers[protocol.HTTP] = http.NewHttpParser()
    ...
}
```

2.2.4 声明支持协议

在deploy/kindling-collector-config.yml中声明http协议

kindling-collector-config.yml Go 复制代码

```yaml
analyzers:
  networkanalyzer:
    protocol_parser: [ http, mysql, dns, redis, kafka, dubbo ]
    protocol_config:
      - key: "http"
        payload_length: 200
```

2.2.5 HTTP测试用例

在analyzer/network/network_analyzer_test.go中补充HTTP协议测试用例

在analyzer/network/protocol/testdata/http/下补充测试数据

```go
func TestHttpProtocol(t *testing.T) {
    testProtocol(t, "http/server-event.yml",
        "http/server-trace-slow.yml",
        "http/server-trace-error.yml",
        "http/server-trace-split.yml",
        "http/server-trace-normal.yml",
    )
}
```

2.2.6 HTTP协议映射

consumer/exporter/tools/adapter/net_dict.go

```go
const (
    empty Protocol = iota
    ...
    HTTP
    ...
)
```

consumer/exporter/tools/adapter/labels_converter.go

```go
func updateProtocolKey(key *extraLabelsKey, labels *model.AttributeMap) *extraLabelsKey {
    ...
    case constvalues.ProtocolHttp:
        key.protocol = HTTP
    ...
}
```