Go TCP Socket编程之teleport框架是怎样炼成的?

本文通过回顾teleport框架的开发过程，讲述Go Socket的开发实战经验。

本文的内容组织形式：teleport架构源码赏析+相应Go技巧分享

期间，我们可以分别从这两条线进行思考与探讨。

注：

- 文中以 `TP` 作为 `teleport` 的简称
- 文中内容针对具有一定Go语言基础的开发者
- 文中以 `Go技巧` 是指高于语法常识的一些编程技巧、设计模式
- 为压缩篇幅代码块中删除了一些空行，并用 `...` 表示省略行

目 录

TP性能测试

TP与其他使用长连接的框架的性能对比：

- teleport/socket

```
➜  teleport git:(master) ✗ ./tp_s_client -n=1000000 -c=100
2017/11/25 19:32:10 concurrency: 100
requests per client: 10000

2017/11/25 19:32:10 message size: 581 bytes

2017/11/25 19:32:10 sent total 1000000 messages, 10000 message per client
2017/11/25 19:32:26 took 15461 ms for 1000000 requests
2017/11/25 19:32:26 sent     requests    : 1000000
2017/11/25 19:32:26 received requests    : 1000000
2017/11/25 19:32:26 received requests_OK : 1000000
2017/11/25 19:32:26 throughput  (TPS)    : 64678
2017/11/25 19:32:26 mean: 1538269 ns, median: 1429372 ns, max: 51373801 ns, min: 44487 ns, p99.9: 11207336 ns
2017/11/25 19:32:26 mean: 1 ms, median: 1 ms, max: 51 ms, min: 0 ms, p99: 11 ms
```

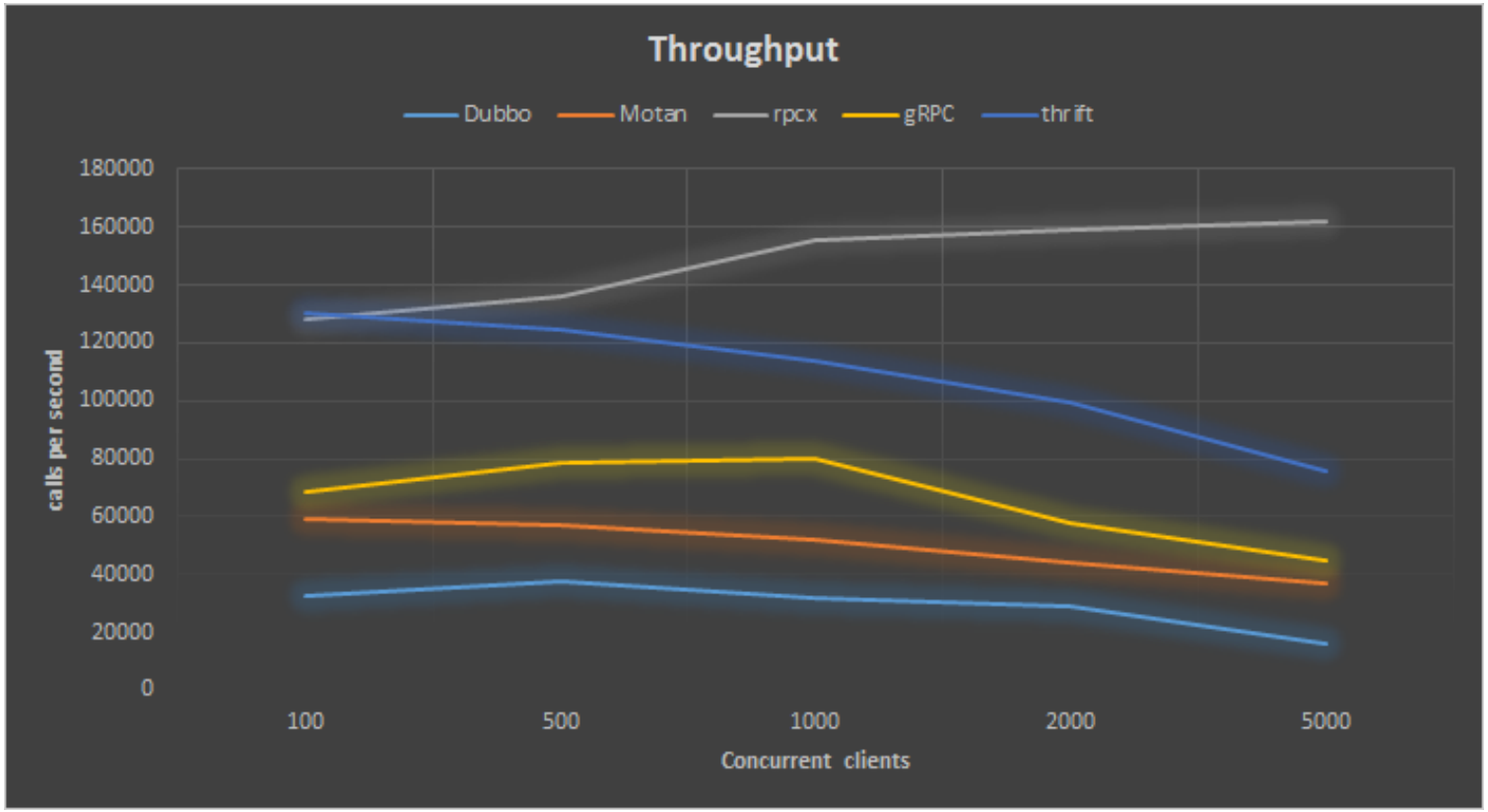
[test code]

- 与rpcx的直接对比

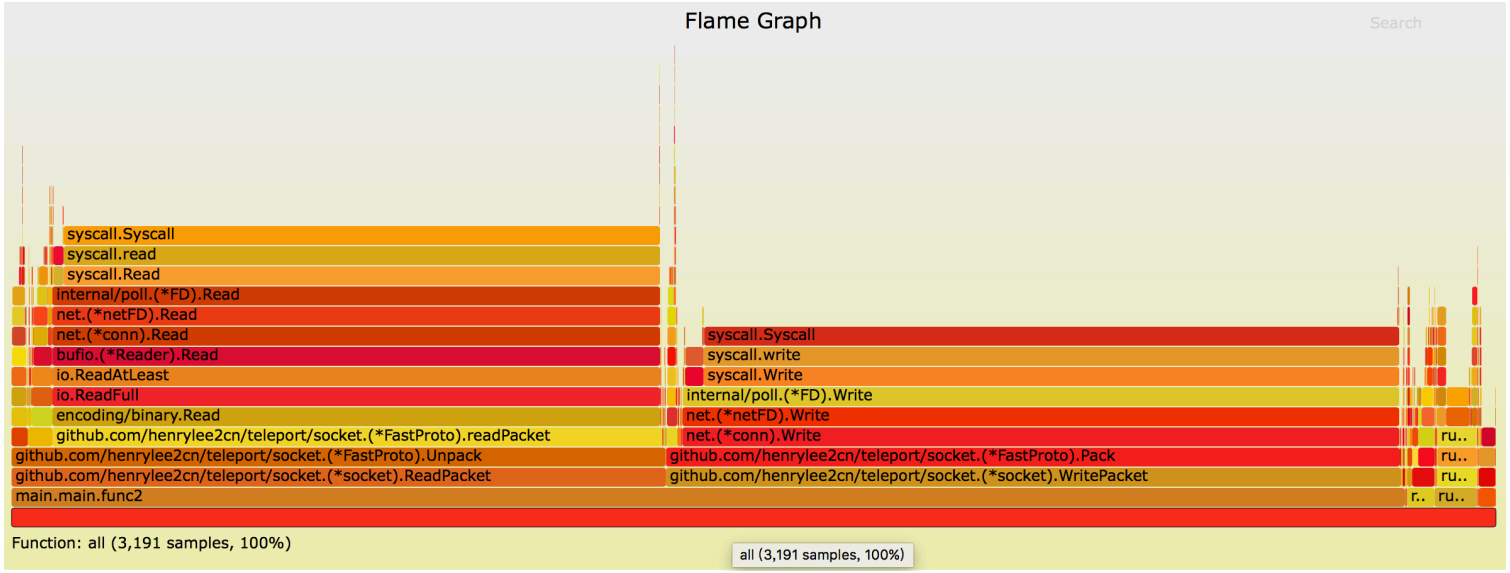
```
➜  rpcx git:(master) ✗ ./rpcx_client -n=1000000 -c=100
2017/11/23 15:44:08 concurrency: 100
requests per client: 10000

2017/11/23 15:44:08 message size: 581 bytes

2017/11/23 15:44:08 sent total 1000000 messages, 10000 message per client
2017/11/23 15:44:32 took 23638 ms for 1000000 requests
2017/11/23 15:44:32 sent     requests    : 1000000
2017/11/23 15:44:32 received requests    : 1000000
2017/11/23 15:44:32 received requests_OK : 1000000
2017/11/23 15:44:32 throughput  (TPS)    : 42304
2017/11/23 15:44:32 mean: 2350247 ns, median: 2010849 ns, max: 90396914 ns, min: 58087 ns, p99.9: 244650
50 ns
2017/11/23 15:44:32 mean: 2 ms, median: 2 ms, max: 90 ms, min: 0 ms, p99: 24 ms
```

[test code]

- rpcx与其他框架的对比参考（图片来源于rpcx）



- teleport/socket火焰图



[svg file]

第一部分 TP架构设计

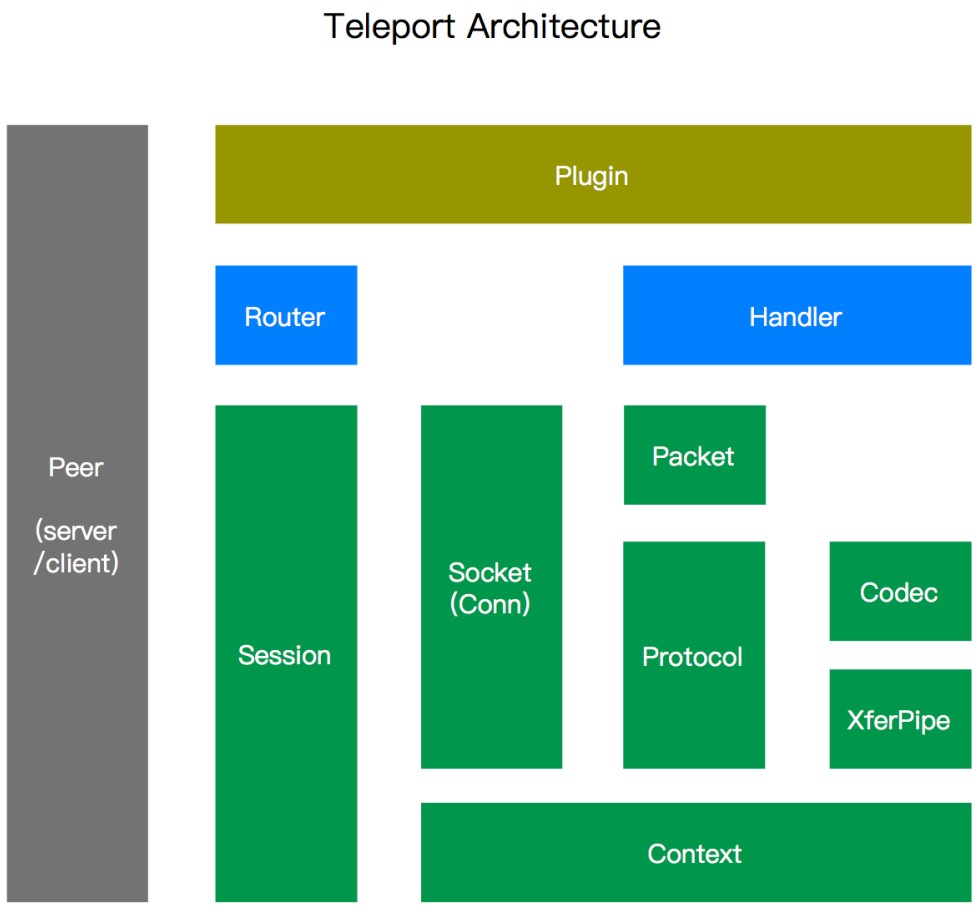
1 设计理念

TP定位于提供socket通信解决方案，遵循以下三点设计理念。

- 通用：不做定向深入，专注长连接通信
- 高效：高性能，低消耗
- 灵活：用法灵活简单，易于深入定制

- 可靠：使用接口（`interface`）而非约束说明，规定框架用法

2 架构图



- `Peer`：通信端点，可以是服务端或客户端
- `Plugin`：贯穿于通信各个环节的插件
- `Handler`：用于处理推、拉请求的函数
- `Router`：通过请求信息（如URI）索引响应函数（Handler）的路由器
- `Socket`：对net.Conn的封装，增加自定义包协议、传输管道等功能
- `Session`：基于Socket封装的连接会话，提供的推、拉、回复、关闭等会话操作
- `Context`：连接会话中一次通信（如PULL-REPLY, PUSH）的上下文对象
- `Packet`：约定数据报文包含的内容元素（注意：它不是协议格式）
- `Protocol`：数据报文封包解包操作，即通信协议的实现接口
- `Codec`：数据包body部分（请求参数或响应结果）的序列化接口
- `XferPipe`：数据包字节流的编码处理管道，如压缩、加密、校验等

3 重要特性

- 支持自定义通信协议和包数据处理管道
- TCP长连接使用I/O缓冲区与多路复用技术，提升数据吞吐量
- 支持设置读取包的大小限制（如果超出则断开连接）
- 支持插件机制，可以自定义认证、心跳、微服务注册中心、统计信息插件等
- 服务端和客户端之间对等通信，统一为peer端点，具有基本一致的用法：
  - 推、拉、回复等通信方法
  - 丰富的插件挂载点，可以自定义认证、心跳、微服务注册中心、统计信息等等
  - 平滑重启与关闭
  - 日志信息详尽，支持打印输入、输出消息的详细信息（状态码、消息头、消息体）
  - 支持设置慢操作报警阈值
  - 提供Hander的上下文（pull、push的handler）

第二部分 TP关键源码赏析及相关Go技巧

----------以下为 `github.com/henrylee2cn/telepot/socket` 包内容----------

1 Packet统一数据包元素

`Packet` 结构体用于定义统一的数据包内容元素，为上层架构提供稳定、统一的操作API。

$ Go技巧分享

1. 在 `teleport/socket` 目录下执行 `go doc Packet` 命令，我们可以获得以下关于 `Packet` 的定义、函数与方法：

```
type Packet struct {
    // Has unexported fields.
}
    Packet a socket data packet.

func GetPacket(settings ...PacketSetting) *Packet
func NewPacket(settings ...PacketSetting) *Packet
func (p *Packet) AppendXferPipeFrom(src *Packet)
func (p *Packet) Body() interface{}
func (p *Packet) BodyCodec() byte
func (p *Packet) MarshalBody() ([]byte, error)
func (p *Packet) Meta() *utils.Args
func (p *Packet) Ptype() byte
func (p *Packet) Reset(settings ...PacketSetting)
func (p *Packet) Seq() uint64
func (p *Packet) SetBody(body interface{})
func (p *Packet) SetBodyCodec(bodyCodec byte)
func (p *Packet) SetNewBody(newBodyFunc NewBodyFunc)
func (p *Packet) SetPtype(ptype byte)
func (p *Packet) SetSeq(seq uint64)
func (p *Packet) SetSize(size uint32) error
func (p *Packet) SetUri(uri string)
func (p *Packet) Size() uint32
func (p *Packet) String() string
func (p *Packet) UnmarshalBody(bodyBytes []byte) error
func (p *Packet) UnmarshalNewBody(bodyBytes []byte) error
func (p *Packet) Uri() string
func (p *Packet) XferPipe() *xfer.XferPipe
```

2. `Packet` 全部字段均不可导出，可以增强代码稳定性以及对其操作的掌控力

3. 下面是由 `Packet` 结构体实现的两个接口 `Header` 和 `Body` 。思考：为什么不直接使用 `Packet` 或者定义两个子结构体?

- 使用接口可以达到限制调用方法的目的，不同情况下使用不同方法集，开发者不会因为调用了不该调用的方法而掉坑里
- 在语义上， `Packet` 只是用于定义统一的数据包内容元素，并未给予任何关于数据结构方面（协议）的暗示、误导。因此不应该使用子结构体

```
type (
    // packet header interface
    Header interface {
        // Ptype returns the packet sequence
        Seq() uint64
        // SetSeq sets the packet sequence
        SetSeq(uint64)
        // Ptype returns the packet type, such as PULL, PUSH, REPLY
        Ptype() byte
        // Ptype sets the packet type
        SetPtype(byte)
        // Uri returns the URL string string
        Uri() string
        // SetUri sets the packet URL string
        SetUri(string)
        // Meta returns the metadata
        Meta() *utils.Args
    }
```

```
    // packet body interface
    Body interface {
        // BodyCodec returns the body codec type id
        BodyCodec() byte
        // SetBodyCodec sets the body codec type id
        SetBodyCodec(bodyCodec byte)
        // Body returns the body object
        Body() interface{}
        // SetBody sets the body object
        SetBody(body interface{})
        // SetNewBody resets the function of geting body.
        SetNewBody(newBodyFunc NewBodyFunc)
        // MarshalBody returns the encoding of body.
        MarshalBody() ([]byte, error)
        // UnmarshalNewBody unmarshal the encoded data to a new body.
        // Note: seq, ptype, uri must be setted already.
        UnmarshalNewBody(bodyBytes []byte) error
        // UnmarshalBody unmarshal the encoded data to the existed body.
        UnmarshalBody(bodyBytes []byte) error
    }
    // NewBodyFunc creates a new body by header.
    NewBodyFunc func(Header) interface{}
)
```

4. 编译期校验 `Packet` 是否已实现 `Header` 与 `Body` 接口的技巧

```
var (
    _ Header = new(Packet)
    _ Body   = new(Packet)
)
```

5. 一种常见的自由赋值的函数用法，用于自由设置 `Packet` 的字段

```
// PacketSetting sets Header field.
type PacketSetting func(*Packet)

// WithSeq sets the packet sequence
func WithSeq(seq uint64) PacketSetting {
    return func(p *Packet) {
        p.seq = seq
    }
}

// Ptype sets the packet type
func WithPtype(ptype byte) PacketSetting {
    return func(p *Packet) {
        p.ptype = ptype
    }
}
...
```

2 Socket接口

`Socket`接口是对`net.Conn`的封装，通过协议接口`Proto`对数据包内容元素`Packet`进行封包、解包与IO传输操作。

```
type (
    // Socket is a generic stream-oriented network connection.
    //
    // Multiple goroutines may invoke methods on a Socket simultaneously.
    Socket interface {
        net.Conn
        // WritePacket writes header and body to the connection.
        // Note: must be safe for concurrent use by multiple goroutines.
        WritePacket(packet *Packet) error
        // ReadPacket reads header and body from the connection.
        // Note: must be safe for concurrent use by multiple goroutines.
        ReadPacket(packet *Packet) error
        // Public returns temporary public data of Socket.
        Public() goutil.Map
        // PublicLen returns the length of public data of Socket.
        PublicLen() int
        // Id returns the socket id.
        Id() string
        // SetId sets the socket id.
        SetId(string)
    }
    socket struct {
        net.Conn
        protocol  Proto
        id        string
```

```
        idMutex   sync.RWMutex
        ctxPublic goutil.Map
        mu        sync.RWMutex
        curState  int32
        fromPool  bool
    }
)
```

$ Go技巧分享

1. 为什么要对外提供接口，而不直接公开结构体?

`socket` 结构体通过匿名字段 `net.Conn` 的方式“继承”了底层的连接操作方法，并基于该匿名字段创建了协议对象。

所以不能允许外部直接通过 `socket.Conn=newConn` 的方式改变连接句柄。

使用 `Socket` 接口封装包外不可见的 `socket` 结构体可达到避免外部直接修改字段的目的。

2. 读写锁遵循最小化锁定的原则，且 `defer` 绝不是必须的，在确定运行安全的情况下尽量避免使用有性能消耗的 `defer` 。

```
func (s *socket) ReadPacket(packet *Packet) error {
    s.mu.RLock()
    protocol := s.protocol
    s.mu.RUnlock()
    return protocol.Unpack(packet)
}
```

3 Proto协议接口

`Proto` 接口按照实现它的具体规则，对 `Packet` 数据包内容元素进行封包、解包、IO等操作。

```
type (
    // Proto pack/unpack protocol scheme of socket packet.
    Proto interface {
        // Version returns the protocol's id and name.
        Version() (byte, string)
        // Pack pack socket data packet.
        // Note: Make sure to write only once or there will be package contamination!
        Pack(*Packet) error
        // Unpack unpack socket data packet.
        // Note: Concurrent unsafe!
        Unpack(*Packet) error
    }
    // ProtoFunc function used to create a custom Proto interface.
    ProtoFunc func(io.ReadWriter) Proto

    // FastProto fast socket communication protocol.
    FastProto struct {
        id   byte
        name string
        r    io.Reader
        w    io.Writer
        rMu  sync.Mutex
    }
)
```

$ Go技巧分享

1. 将数据包的封包、解包操作封装为 `Proto` 接口，并定义一个默认实现（ `FastProto` ）。这是框架设计中增

强可定制性的一种有效手段。开发者既可以使用默认实现，也可以根据特殊需求定制自己的个性实现。

2. 使用 `Packet` 屏蔽不同协议的差异性：封包时以 `Packet` 的字段为内容元素进行数据序列化，解包时以 `Packet` 为内容模板进行数据的反序列化。

----------*以下为 `github.com/henrylee2cn/telepot/codec` 包内容*----------

4 Codec编解码

`Codec` 接口是 `socket.Packet.body` 的编解码器。TP已默认注册了JSON、Protobuf、String三种编解码器。

```
type (
    // Codec makes Encoder and Decoder
    Codec interface {
        // Id returns codec id.
        Id() byte
        // Name returns codec name.
        Name() string
        // Marshal returns the encoding of v.
        Marshal(v interface{}) ([]byte, error)
        // Unmarshal parses the encoded data and stores the result
        // in the value pointed to by v.
        Unmarshal(data []byte, v interface{}) error
    }
)
```

$ Go技巧分享

1. 下面 `codecMap` 变量的类型为什么不用关键字 `type` 定义?

```
var codecMap = struct {
    nameMap map[string]Codec
    idMap   map[byte]Codec
}{
    nameMap: make(map[string]Codec),
    idMap:   make(map[byte]Codec),
}
```

Go语法允许我们在声明变量时临时定义类型并赋值。因为 `codecMap` 所属类型只会有一个全局唯一的实例，且不会用于其他变量类型声明上，所以直接在声明变量时声明类型可以令代码更简洁。

2. 常用的依赖注入实现方式，实现编解码器的自由定制

```
const (
    NilCodecId   byte   = 0
    NilCodecName string = ""
)

func Reg(codec Codec) {
    if codec.Id() == NilCodecId {
        panic(fmt.Sprintf("codec id can not be %d", NilCodecId))
    }
    if _, ok := codecMap.nameMap[codec.Name()]; ok {
        panic("multi-register codec name: " + codec.Name())
```

```
    }
    if _, ok := codecMap.idMap[codec.Id()]; ok {
        panic(fmt.Sprintf("multi-register codec id: %d", codec.Id()))
    }
    codecMap.nameMap[codec.Name()] = codec
    codecMap.idMap[codec.Id()] = codec
}

func Get(id byte) (Codec, error) {
    codec, ok := codecMap.idMap[id]
    if !ok {
        return nil, fmt.Errorf("unsupported codec id: %d", id)
    }
    return codec, nil
}

func GetByName(name string) (Codec, error) {
    codec, ok := codecMap.nameMap[name]
    if !ok {
        return nil, fmt.Errorf("unsupported codec name: %s", name)
    }
    return codec, nil
}
```

----------*以下为 `github.com/henrylee2cn/telepot/xfer` 包内容*----------

5 XferPipe数据编码管道

`XferPipe` 接口用于对数据包进行一系列自定义处理加工，如gzip压缩、加密、校验等。

```
type (
    // XferPipe transfer filter pipe, handlers from outer-most to inner-most.
    // Note: the length can not be bigger than 255!
    XferPipe struct {
        filters []XferFilter
    }
    // XferFilter handles byte stream of packet when transfer.
    XferFilter interface {
        Id() byte
        OnPack([]byte) ([]byte, error)
        OnUnpack([]byte) ([]byte, error)
    }
)

var xferFilterMap = struct {
    idMap map[byte]XferFilter
}{
    idMap: make(map[byte]XferFilter),
}
```

`teleport/xfer` 包的设计与 `teleport/codec` 类似， `xferFilterMap` 为注册中心，提供注册、查询、执行等功能。

----------*以下为 `github.com/henrylee2cn/telepot` 包内容*----------

6 Peer通信端点

Peer结构体是TP的一个通信端点，它可以是服务端也可以是客户端，甚至可以同时是服务端与客户端。因此，TP是端对端对等通信的。

```
type Peer struct {
    PullRouter *Router
    PushRouter *Router
    // Has unexported fields.
}
func NewPeer(cfg *PeerConfig, plugin ...Plugin) *Peer
func (p *Peer) Close() (err error)
func (p *Peer) CountSession() int
func (p *Peer) Dial(addr string, protoFunc ...socket.ProtoFunc) (Session, *Rerror)
func (p *Peer) DialContext(ctx context.Context, addr string, protoFunc ...socket.ProtoF
unc) (Session, *Rerror)
func (p *Peer) GetSession(sessionId string) (Session, bool)
func (p *Peer) Listen(protoFunc ...socket.ProtoFunc) error
func (p *Peer) RangeSession(fn func(sess Session) bool)
func (p *Peer) ServeConn(conn net.Conn, protoFunc ...socket.ProtoFunc) (Session, error)
```

- 通信端点介绍

1. Peer配置信息

```
type PeerConfig struct {
    TlsCertFile         string
    TlsKeyFile          string
    DefaultReadTimeout  time.Duration
    DefaultWriteTimeout time.Duration
    SlowCometDuration   time.Duration
    DefaultBodyCodec    string
    PrintBody           bool
    CountTime           bool
    DefaultDialTimeout  time.Duration
    ListenAddrs         []string
}
```

2. Peer的功能列表

- 提供路由功能
- 作为服务端可同时支持监听多个地址端口
- 作为客户端可与任意服务端建立连接
- 提供会话查询功能
- 支持TLS证书安全加密
- 设置默认的建立连接和读、写超时
- 慢响应阀值（超出后运行日志由INFO提升为WARN）
- 支持打印body
- 支持在运行日志中增加耗时统计

$ Go技巧分享

一个Go协程大约是8KB，如在高并发服务中不加限制地频繁创建 / 销毁协程，很容易造成内存资源耗尽，且对GC压力也会很大。因此，TP内部采用协程资源池来管控协程，可以大大降低服务器内存与CPU的压力。（该思路源于fasthttp）

协程资源池的源码实现在本人goutil库中的 `github.com/henrylee2cn/goutil/pool` 。下面是TP的二次封装：

```
var (
    _maxGoroutinesAmount      = (1024 * 1024 * 8) / 8 // max memory 8GB (8KB/goroutine)
    _maxGoroutineIdleDuration time.Duration
    _gopool                   = pool.NewGoPool(_maxGoroutinesAmount, _maxGoroutineIdleD
uration)
)
// SetGopool set or reset go pool config.
// Note: Make sure to call it before calling NewPeer() and Go()
func SetGopool(maxGoroutinesAmount int, maxGoroutineIdleDuration time.Duration) {
    _maxGoroutinesAmount, _maxGoroutineIdleDuration := maxGoroutinesAmount, maxGoroutin
eIdleDuration
    if _gopool != nil {
        _gopool.Stop()
    }
    _gopool = pool.NewGoPool(_maxGoroutinesAmount, _maxGoroutineIdleDuration)
}
// Go similar to go func, but return false if insufficient resources.
func Go(fn func()) bool {
    if err := _gopool.Go(fn); err != nil {
        Warnf("%s", err.Error())
        return false
```

```
    }
    return true
}
// AnywayGo similar to go func, but concurrent resources are limited.
func AnywayGo(fn func()) {
TRYGO:
    if !Go(fn) {
        time.Sleep(time.Second)
        goto TRYGO
    }
}
```

每当Peer创建一个session时，都有调用上述 `Go` 函数进行并发执行：

```
func (p *Peer) DialContext(ctx context.Context, addr string, protoFunc ...socket.ProtoF
unc) (Session, *Rerror) {
    ...
    Go(sess.startReadAndHandle)
    ...
}
func (p *Peer) listen(addr string, protoFuncs []socket.ProtoFunc) error {
    var lis, err = listen(addr, p.tlsConfig)
    if err != nil {
        Fatalf("%v", err)
    }
    p.listens = append(p.listens, lis)
    defer lis.Close()
```

```
    Printf("listen ok (addr: %s)", addr)

    var (
        tempDelay time.Duration // how long to sleep on accept failure
        closeCh   = p.closeCh
    )
    for {
        rw, e := lis.Accept()
        ...
        func(conn net.Conn) {
        TRYGO:
            if !Go(func() {
                ...
                sess.startReadAndHandle()
            }) {
                time.Sleep(time.Second)
                goto TRYGO
            }
        }(rw)
    }
}
```

7 Router路由器

TP是对等通信，路由不再是服务端的专利，只要是Peer端点就支持注册 `PULL` 和 `PUSH` 这两类消息处理路由。

```
type Router struct {
    handlers       map[string]*Handler
    unknownApiType **Handler
    // only for register router
    pathPrefix      string
    pluginContainer PluginContainer
    typ             string
    maker           HandlersMaker
}

func (r *Router) Group(pathPrefix string, plugin ...Plugin) *Router
func (r *Router) Reg(ctrlStruct interface{}, plugin ...Plugin)
func (r *Router) SetUnknown(unknownHandler interface{}, plugin ...Plugin)
```

$ Go技巧分享

1. 根据`maker HandlersMaker`（Handler的构造函数）字段的不同，分别实现了`PullRouter`和`PushRouter`两类路由。

```
// HandlersMaker makes []*Handler
type HandlersMaker func(pathPrefix string, ctrlStruct interface{}, pluginContainer Plug
inContainer) ([]*Handler, error)
```

2. 简洁地路由分组实现：

- 继承各级路由的共享字段：`handlers`、`unknownApiType`、`maker`
- 在上级路由节点的`pathPrefix`、`pluginContainer`字段基础上追加当前节点信息

```
// Group add handler group.
func (r *Router) Group(pathPrefix string, plugin ...Plugin) *Router {
    pluginContainer, err := r.pluginContainer.cloneAdd(plugin...)
    if err != nil {
        Fatalf("%v", err)
    }
    warnInvaildRouterHooks(plugin)
    return &Router{
        handlers:        r.handlers,
        unknownApiType:  r.unknownApiType,
        pathPrefix:      path.Join(r.pathPrefix, pathPrefix),
        pluginContainer: pluginContainer,
        maker:           r.maker,
    }
}
```

8 控制器

控制器是指用于提供Handler操作的结构体。

$ Go技巧分享

1. Go没有泛型，我们通常使用 `interface{}` 空接口来代替。 但是，空接口不能用于表示结构体的方法。

下面是控制器结构体及其方法的模型定义：

PullController Model:

```
type Aaa struct {
    tp.PullCtx
}
// XxZz register the route: /aaa/xx_zz
func (x *Aaa) XxZz(args *<T>) (<T>, *tp.Rerror) {
    ...
    return r, nil
}
// YyZz register the route: /aaa/yy_zz
func (x *Aaa) YyZz(args *<T>) (<T>, *tp.Rerror) {
    ...
    return r, nil
}
```

PushController Model:

```
type Bbb struct {
    tp.PushCtx
}
// XxZz register the route: /bbb/yy_zz
func (b *Bbb) XxZz(args *<T>) {
    ...
    return r, nil
}
// YyZz register the route: /bbb/yy_zz
func (b *Bbb) YyZz(args *<T>) {
    ...
    return r, nil
}
```

以PullController为例，使用 `reflect` 反射包对未知类型的结构体进行模型验证：

```
func pullHandlersMaker(pathPrefix string, ctrlStruct interface{}, pluginContainer Plugi
nContainer) ([]*Handler, error) {
    var (
        ctype    = reflect.TypeOf(ctrlStruct)
        handlers = make([]*Handler, 0, 1)
    )

    if ctype.Kind() != reflect.Ptr {
        return nil, errors.Errorf("register pull handler: the type is not struct point:
 %s", ctype.String())
    }
```

```
    var ctypeElem = ctype.Elem()
    if ctypeElem.Kind() != reflect.Struct {
        return nil, errors.Errorf("register pull handler: the type is not struct point:
 %s", ctype.String())
    }

    if _, ok := ctrlStruct.(PullCtx); !ok {
        return nil, errors.Errorf("register pull handler: the type is not implemented P
ullCtx interface: %s", ctype.String())
    }

    iType, ok := ctypeElem.FieldByName("PullCtx")
    if !ok || !iType.Anonymous {
        return nil, errors.Errorf("register pull handler: the struct do not have anonym
ous field PullCtx: %s", ctype.String())
    }
    ...
    for m := 0; m < ctype.NumMethod(); m++ {
        method := ctype.Method(m)
        mtype := method.Type
        mname := method.Name
        // Method must be exported.
        if method.PkgPath != "" || isPullCtxType(mname) {
            continue
        }
        // Method needs two ins: receiver, *args.
        if mtype.NumIn() != 2 {
            return nil, errors.Errorf("register pull handler: %s.%s needs one in argume
nt, but have %d", ctype.String(), mname, mtype.NumIn())
```

```
        }
        // Receiver need be a struct pointer.
        structType := mtype.In(0)
        if structType.Kind() != reflect.Ptr || structType.Elem().Kind() != reflect.Stru
ct {
            return nil, errors.Errorf("register pull handler: %s.%s receiver need be a
struct pointer: %s", ctype.String(), mname, structType)
        }
        // First arg need be exported or builtin, and need be a pointer.
        argType := mtype.In(1)
        if !goutil.IsExportedOrBuiltinType(argType) {
            return nil, errors.Errorf("register pull handler: %s.%s args type not expor
ted: %s", ctype.String(), mname, argType)
        }
        if argType.Kind() != reflect.Ptr {
            return nil, errors.Errorf("register pull handler: %s.%s args type need be a
 pointer: %s", ctype.String(), mname, argType)
        }
        // Method needs two outs: reply error.
        if mtype.NumOut() != 2 {
            return nil, errors.Errorf("register pull handler: %s.%s needs two out argum
ents, but have %d", ctype.String(), mname, mtype.NumOut())
        }
        // Reply type must be exported.
        replyType := mtype.Out(0)
        if !goutil.IsExportedOrBuiltinType(replyType) {
            return nil, errors.Errorf("register pull handler: %s.%s first reply type no
t exported: %s", ctype.String(), mname, replyType)
        }
```

```
        // The return type of the method must be Error.
        if returnType := mtype.Out(1); !isRerrorType(returnType.String()) {
            return nil, errors.Errorf("register pull handler: %s.%s second reply type %
s not *tp.Rerror", ctype.String(), mname, returnType)
        }
        ...
    }
    ...
}
```

2. 参考HTTP的成熟经验，TP的路由路径采用类URL格式，且支持query参数：如 `/a/b?n=1&m=e`

9 Unknown操作函数

TP可通过 `func (r *Router) SetUnknown(unknownHandler interface{}, plugin ...Plugin)` 方法设置默认Handler，用于处理未找到路由的 `PULL` 或 `PUSH` 消息。

UnknownPullHandler Type:

```
func(ctx UnknownPullCtx) (interface{}, *Rerror) {
    ...
    return r, nil
}
```

UnknownPushHandler Type:

```
func(ctx UnknownPushCtx)
```

10 Handler的构造

```
// Handler pull or push handler type info
Handler struct {
    name              string
    isUnknown         bool
    argElem           reflect.Type
    reply             reflect.Type // only for pull handler doc
    handleFunc        func(*readHandleCtx, reflect.Value)
    unknownHandleFunc func(*readHandleCtx)
    pluginContainer   PluginContainer
}
```

通过 `HandlersMaker` 对Controller各个方法进行解析，构造出相应数量的Handler。以 `pullHandlersMaker` 函数为例：

```
func pullHandlersMaker(pathPrefix string, ctrlStruct interface{}, pluginContainer Plugi
nContainer) ([]*Handler, error) {
    var (
        ctype    = reflect.TypeOf(ctrlStruct)
        handlers = make([]*Handler, 0, 1)
    )
    ...
    var ctypeElem = ctype.Elem()
    ...
```

```
    iType, ok := ctypeElem.FieldByName("PullCtx")
    ...
    var pullCtxOffset = iType.Offset

    if pluginContainer == nil {
        pluginContainer = newPluginContainer()
    }

    type PullCtrlValue struct {
        ctrl   reflect.Value
        ctxPtr *PullCtx
    }
    var pool = &sync.Pool{
        New: func() interface{} {
            ctrl := reflect.New(ctypeElem)
            pullCtxPtr := ctrl.Pointer() + pullCtxOffset
            ctxPtr := (*PullCtx)(unsafe.Pointer(pullCtxPtr))
            return &PullCtrlValue{
                ctrl:   ctrl,
                ctxPtr: ctxPtr,
            }
        },
    }

    for m := 0; m < ctype.NumMethod(); m++ {
        method := ctype.Method(m)
        mtype := method.Type
        mname := method.Name
        ...
```

```
        var methodFunc = method.Func
        var handleFunc = func(ctx *readHandleCtx, argValue reflect.Value) {
            obj := pool.Get().(*PullCtrlValue)
            *obj.ctxPtr = ctx
            rets := methodFunc.Call([]reflect.Value{obj.ctrl, argValue})
            ctx.output.SetBody(rets[0].Interface())
            rerr, _ := rets[1].Interface().(*Rerror)
            if rerr != nil {
                rerr.SetToMeta(ctx.output.Meta())

            } else if ctx.output.Body() != nil && ctx.output.BodyCodec() == codec.NilCo
decId {
                ctx.output.SetBodyCodec(ctx.input.BodyCodec())
            }
            pool.Put(obj)
        }

        handlers = append(handlers, &Handler{
            name:            path.Join(pathPrefix, ctrlStructSnakeName(ctype), goutil.S
nakeString(mname)),
            handleFunc:      handleFunc,
            argElem:         argType.Elem(),
            reply:           replyType,
            pluginContainer: pluginContainer,
        })
    }
    return handlers, nil
}
```

$ Go技巧分享

- 对不可变的部分进行预处理获得闭包变量，抽离可变部分的逻辑构造子函数。在路由处理过程中直接执行这些 `handleFunc` 子函数可达到显著提升性能的目的
- 使用反射来创建任意类型的实例并调用其方法，适用于类型或方法不固定的情况
- 使用对象池来复用 `PullCtrlValue` ，可以降低GC开销与内存占用
- 通过unsafe获取 `ctrlStruct.PullCtx` 字段的指针偏移量，进而可以快速获取该字段的值

11 Session会话

Session是封装了socket连接的会话管理实例。它使用一个包外不可见的结构体 `session` 来实现会话相关的三个接口： `PreSession` 、 `Session` 、 `PostSession` 。（此处session实现多接口的做法类似于Packet）

```
type (
    PreSession interface {
        ...
    }
    Session interface {
        // SetId sets the session id.
        SetId(newId string)
        // Close closes the session.
        Close() error
        // Id returns the session id.
        Id() string
        // IsOk checks if the session is ok.
        IsOk() bool
        // Peer returns the peer.
```

```
        Peer() *Peer
        // AsyncPull sends a packet and receives reply asynchronously.
        // If the args is []byte or *[]byte type, it can automatically fill in the body
 codec name.
        AsyncPull(uri string, args interface{}, reply interface{}, done chan *PullCmd,
setting ...socket.PacketSetting)
        // Pull sends a packet and receives reply.
        // If the args is []byte or *[]byte type, it can automatically fill in the body
 codec name.
        Pull(uri string, args interface{}, reply interface{}, setting ...socket.PacketS
etting) *PullCmd
        // Push sends a packet, but do not receives reply.
        // If the args is []byte or *[]byte type, it can automatically fill in the body
 codec name.
        Push(uri string, args interface{}, setting ...socket.PacketSetting) *Rerror
        // ReadTimeout returns readdeadline for underlying net.Conn.
        ReadTimeout() time.Duration
        // RemoteIp returns the remote peer ip.
        RemoteIp() string
        // LocalIp returns the local peer ip.
        LocalIp() string
        // ReadTimeout returns readdeadline for underlying net.Conn.
        SetReadTimeout(duration time.Duration)
        // WriteTimeout returns writedeadline for underlying net.Conn.
        SetWriteTimeout(duration time.Duration)
        // Socket returns the Socket.
        // Socket() socket.Socket
        // WriteTimeout returns writedeadline for underlying net.Conn.
        WriteTimeout() time.Duration
```

```
        // Public returns temporary public data of session(socket).
        Public() goutil.Map
        // PublicLen returns the length of public data of session(socket).
        PublicLen() int
    }
    PostSession interface {
        ...
    }
    session struct {
        ...
    }
)
```

Session采用读写异步的方式处理通信消息。在创建Session后，立即启动一个循环读取数据包的协程，并为每个成功读取的数据包创建一个处理协程。

而写操作则是由session.Pull、session.Push或者Handler三种方式来触发执行。

$ Go技巧分享

在以客户端角色执行PULL请求时，Session支持同步和异步两种方式。这是Go的一种经典的兼容同步异步调用的技巧：

```
func (s *session) AsyncPull(uri string, args interface{}, reply interface{}, done chan
*PullCmd, setting ...socket.PacketSetting) {
    ...
    cmd := &PullCmd{
        sess:     s,
```

```
        output:   output,
        reply:    reply,
        doneChan: done,
        start:    s.peer.timeNow(),
        public:   goutil.RwMap(),
    }
    ...
    if err := s.write(output); err != nil {
        cmd.rerr = rerror_writeFailed.Copy()
        cmd.rerr.Detail = err.Error()
        cmd.done()
        return
    }
    s.peer.pluginContainer.PostWritePull(cmd)
}

// Pull sends a packet and receives reply.
// If the args is []byte or *[]byte type, it can automatically fill in the body codec n
ame.
func (s *session) Pull(uri string, args interface{}, reply interface{}, setting ...sock
et.PacketSetting) *PullCmd {
    doneChan := make(chan *PullCmd, 1)
    s.AsyncPull(uri, args, reply, doneChan, setting...)
    pullCmd := <-doneChan
    close(doneChan)
    return pullCmd
}
```

实现步骤：

1. 在返回结果的结构体中绑定一个chan管道
2. 在另一个协程中进行结果计算
3. 将该chan做为返回值返回给调用者
4. 将计算结果写入该chan中
5. 调用者从chan中读出该结果
6. （同步方式是对异步方式的封装，等待从chan中读到结果后，再将该结果作为返回值返回）

12 Context上下文

类似常见的Go HTTP框架，TP同样提供了Context上下文。它携带Handler操作相关的参数，如Peer、Session、Packet、PublicData等。

根据调用场景的不同，定义不同接口来限制其方法列表。

此外，TP的平滑关闭、平滑重启也是建立在对Context的使用状态监控的基础上。

```
type (
    PushCtx interface {
        ...
    }
    PullCtx interface {
        ...
    }
    UnknownPushCtx interface {
        ...
```

```
    }
    UnknownPullCtx interface {
        ...
    }
    WriteCtx interface {
        ...
    }
    ReadCtx interface {
        ...
    }
    readHandleCtx struct {
        sess            *session
        input           *socket.Packet
        output          *socket.Packet
        apiType         *Handler
        arg             reflect.Value
        pullCmd         *PullCmd
        uri             *url.URL
        query           url.Values
        public          goutil.Map
        start           time.Time
        cost            time.Duration
        pluginContainer PluginContainer
        next            *readHandleCtx
    }
)
```

13 Plugin插件

TP提供了插件功能，具有完备的挂载点，便于开发者实现丰富的功能。例如身份认证、心跳、微服务注册中心、信息统计等等。

```
type (
    Plugin interface {
        Name() string
    }
    PostRegPlugin interface {
        Plugin
        PostReg(*Handler) *Rerror
    }
    PostDialPlugin interface {
        Plugin
        PostDial(PreSession) *Rerror
    }
    ...
    PostReadReplyBodyPlugin interface {
        Plugin
        PostReadReplyBody(ReadCtx) *Rerror
    }
    ...
    // PluginContainer plugin container that defines base methods to manage plugins.
    PluginContainer interface {
        Add(plugins ...Plugin) error
        Remove(pluginName string) error
        GetByName(pluginName string) Plugin
        GetAll() []Plugin
        PostReg(*Handler) *Rerror
```

```
        PostDial(PreSession) *Rerror
        ...
        PostReadReplyBody(ReadCtx) *Rerror
        ...
        cloneAdd(...Plugin) (PluginContainer, error)
    }
    pluginContainer struct {
        plugins []Plugin
    }
)

func (p *pluginContainer) PostReg(h *Handler) *Rerror {
    var rerr *Rerror
    for _, plugin := range p.plugins {
        if _plugin, ok := plugin.(PostRegPlugin); ok {
            if rerr = _plugin.PostReg(h); rerr != nil {
                Fatalf("%s-PostRegPlugin(%s)", plugin.Name(), rerr.String())
                return rerr
            }
        }
    }
    return nil
}
func (p *pluginContainer) PostDial(sess PreSession) *Rerror {
    var rerr *Rerror
    for _, plugin := range p.plugins {
        if _plugin, ok := plugin.(PostDialPlugin); ok {
            if rerr = _plugin.PostDial(sess); rerr != nil {
                Debugf("dial fail (addr: %s, id: %s): %s-PostDialPlugin(%s)", sess.Remo
```

```
teIp(), sess.Id(), plugin.Name(), rerr.String())
                return rerr
            }
        }
    }
    return nil
}
func (p *pluginContainer) PostReadReplyBody(ctx ReadCtx) *Rerror {
    var rerr *Rerror
    for _, plugin := range p.plugins {
        if _plugin, ok := plugin.(PostReadReplyBodyPlugin); ok {
            if rerr = _plugin.PostReadReplyBody(ctx); rerr != nil {
                Errorf("%s-PostReadReplyBodyPlugin(%s)", plugin.Name(), rerr.String())
                return rerr
            }
        }
    }
    return nil
}
```

$ Go技巧分享

Go接口断言的灵活运用，实现插件及其管理容器：

1. 定义基础接口并创建统一管理容器
2. 在实现基础接口的基础上，增加个性化接口（具体挂载点）的实现，将其注册进基础接口管理容器
3. 管理容器使用断言的方法筛选出指定挂载点的插件并执行