# 国密workshop

## 国密算法的应答实现

TWGC

# 国密算法的应答实现

## 概要

客户端与服务端功能

密码学封装与实现

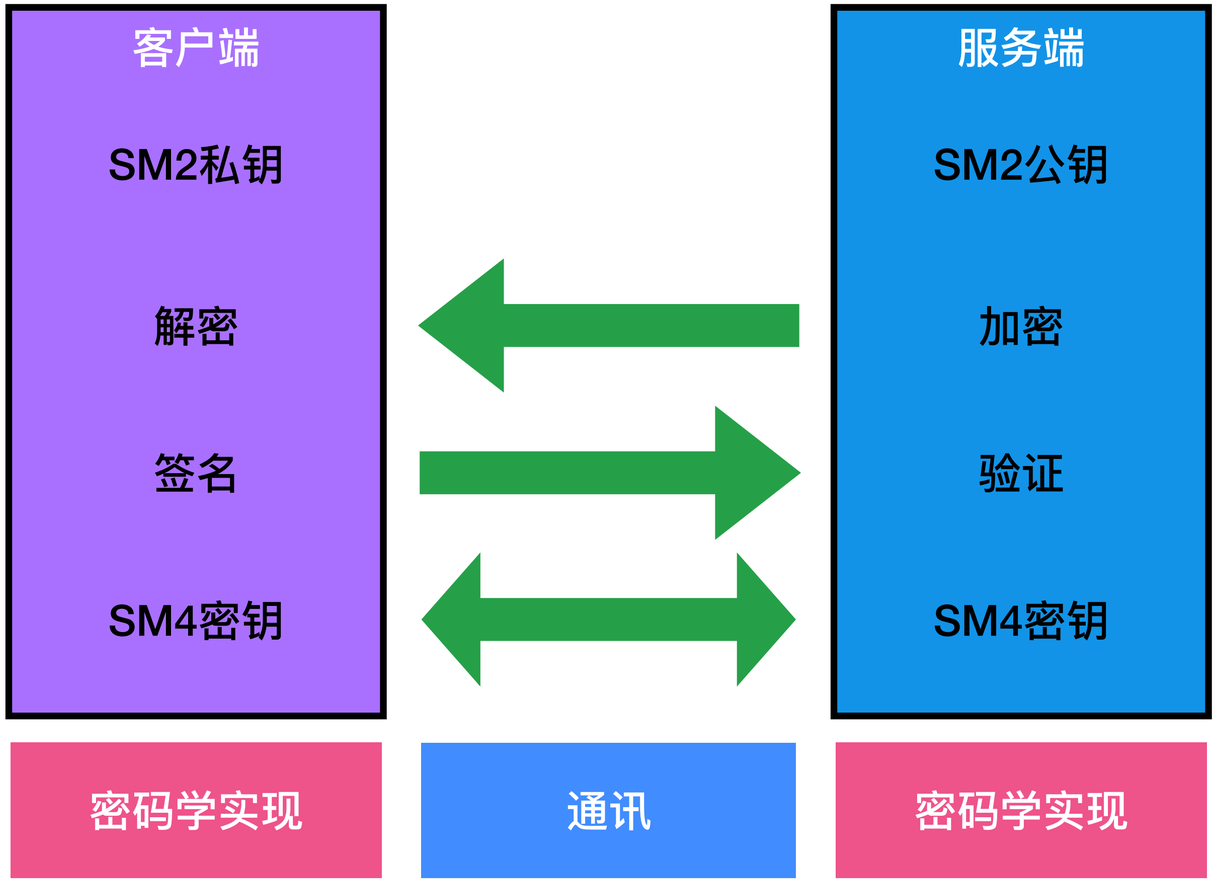
通讯层设计

客户端实现

服务端实现

# 国密算法的应答实现

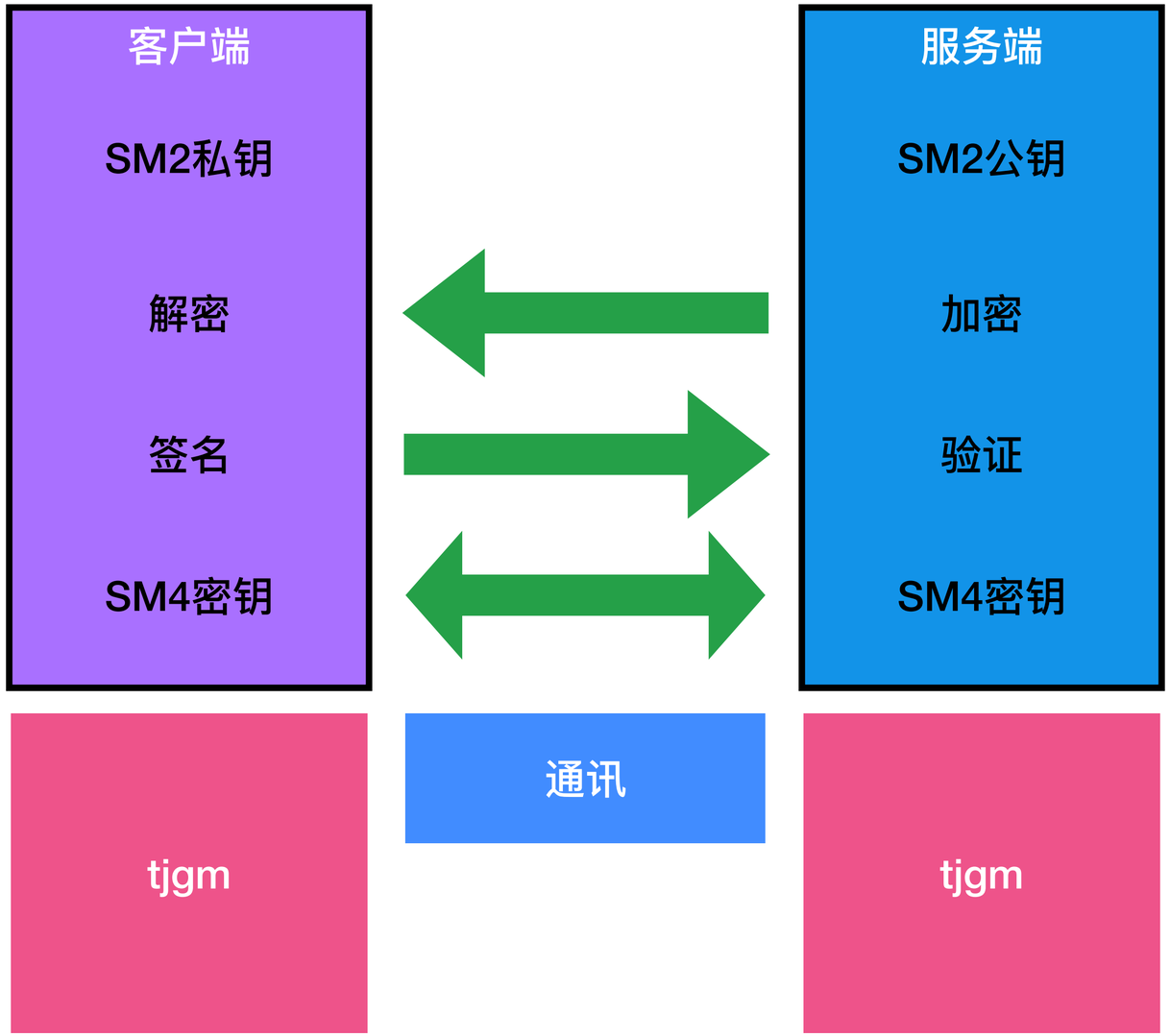
## 客户端与服务端功能



# 国密算法的应答实现

## 密码学实现

密钥保存相关
加解密（sm2）
签名验证（sm2）
sm4



# 密码学实现

## 密钥保存相关

生成密钥对
密钥导入
密钥导出

```go
type TJSM2 struct {
    PrivateKey *tj.PrivateKey
    PublicKey  *tj.PublicKey
}

func NewTJSM2() (*TJSM2, error) {
    PrivateKey, err := tj.GenerateKey(rand.Reader)
    if err != nil {
        return nil, err
    }
    return &TJSM2{PrivateKey: PrivateKey, PublicKey: &PrivateKey.PublicKey}, nil
}

func TJImportKey(privPEM []byte, pubPEM []byte) (*TJSM2, error) {
    PrivateKey, err := tjx509.ReadPrivateKeyFromPem(privPEM, nil)
    if err != nil {
        return nil, err
    }
    PublicKey, err := tjx509.ReadPublicKeyFromPem(pubPEM)
    if err != nil {
        return nil, err
    }
    return &TJSM2{PrivateKey: PrivateKey, PublicKey: PublicKey}, nil
}

func (instance *TJSM2) ExportKey() (privPEM []byte, pubPEM []byte, err error) {
    privPEM, err = tjx509.WritePrivateKeyToPem(instance.PrivateKey, nil)
    if err != nil {
        return
    }
    pubPEM, err = tjx509.WritePublicKeyToPem(instance.PublicKey)
    return
}
```

# 密码学实现

## 密钥保存相关

生成密钥对
密钥导入
密钥导出

```go
func (instance *TJSM2) SaveFile(priFile, pubFile string) error {
    var err error
    var pemBytes []byte
    pemBytes, err = tjx509.WritePrivateKeyToPem(instance.PrivateKey, nil)
    if err != nil {
        return err
    }
    err = WriteFile(pemBytes, priFile)
    if err != nil {
        return err
    }
    pemBytes, err = tjx509.WritePublicKeyToPem(instance.PublicKey)
    if err != nil {
        return err
    }
    err = WriteFile(pemBytes, pubFile)
    if err != nil {
        return err
    }
    return err
}

func WriteFile(content []byte, filename string) error {
    var err error
    var file *os.File
    file, err = os.Create(filename)
    defer func() {
        err = file.Close()
    }()
    _, err = file.Write(content)
    return err
}
```

# 密码学实现

## sm2操作相关

加解密
签名验证

```go
func (instance *TJSM2) Encrypt(msg []byte) ([]byte, error) {
    encrypted, err := instance.PublicKey.EncryptAsn1(msg, rand.Reader)
    if err != nil {
        return nil, err
    }
    return encrypted, nil
}

func (instance *TJSM2) Decrypt(encrypted []byte) ([]byte, error) {
    decrypted, err := instance.PrivateKey.DecryptAsn1(encrypted)
    if err != nil {
        return nil, err
    }
    return decrypted, nil
}

func (instance *TJSM2) Sign(msg []byte) ([]byte, error) {
    tj_digest := sm3.Sm3Sum(msg)
    sign, err := instance.PrivateKey.Sign(rand.Reader, tj_digest, nil) // 签名
    if err != nil {
        return nil, err
    }
    return sign, nil
}

func (instance *TJSM2) Verify(msg []byte, sign []byte) bool {
    tj_digest := sm3.Sm3Sum(msg)
    ok := instance.PublicKey.Verify(tj_digest, sign) // 公钥验证
    return ok
}
```

# 密码学实现

## sm4操作相关

加解密
签名验证

```go
type TJSM4 struct {
    Key []byte
}

func NewTJSM4() (*TJSM4, error) {
    key := []byte("0123456789abcdef")
    return &TJSM4{Key: key}, nil
}
func (instance *TJSM4) Encrypt(msg []byte, mode string) ([]byte, error) {
    switch mode {
    case "ecb":
        return tj.Sm4Ecb(instance.Key, msg, true)
    case "cbc":
        return tj.Sm4Cbc(instance.Key, msg, true)
    case "cfb":
        return tj.Sm4CFB(instance.Key, msg, true)
    case "ofb":
        return tj.Sm4OFB(instance.Key, msg, true)
    default:
        return tj.Sm4Ecb(instance.Key, msg, true)
    }
}
func (instance *TJSM4) Decrypt(encrypted []byte, mode string) ([]byte, error) {
    switch mode {
    case "ecb":
        return tj.Sm4Ecb(instance.Key, encrypted, false)
    case "cbc":
        return tj.Sm4Cbc(instance.Key, encrypted, false)
    case "cfb":
        return tj.Sm4CFB(instance.Key, encrypted, false)
    case "ofb":
        return tj.Sm4OFB(instance.Key, encrypted, false)
    default:
        return tj.Sm4Ecb(instance.Key, encrypted, false)
    }
}
```

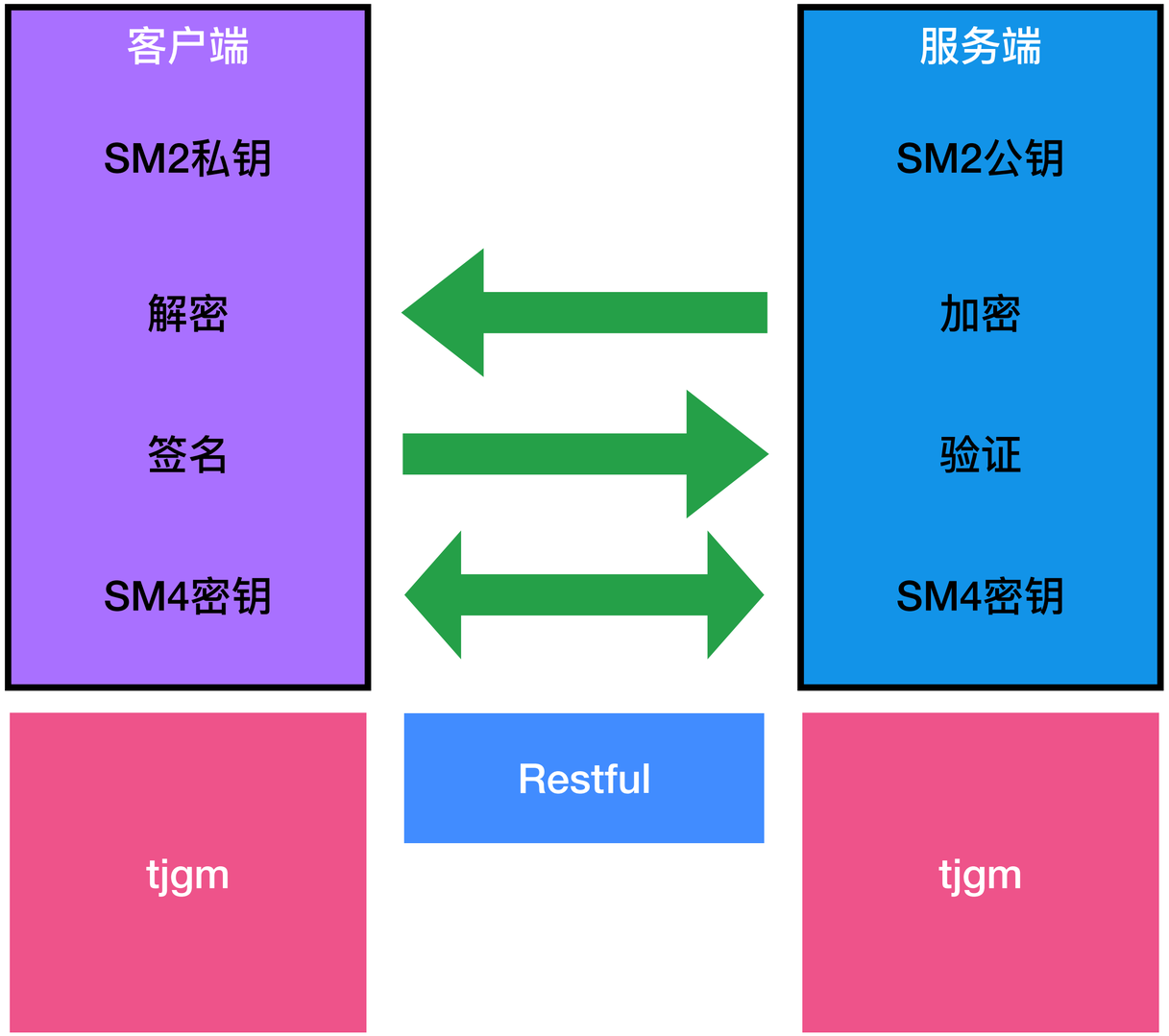
# 国密算法的应答实现

## 通讯

使用HEX或者其他编码方式，避免加密过程中产生不可读字符从而影响解析。

```
hex.EncodeToString(msg)
```

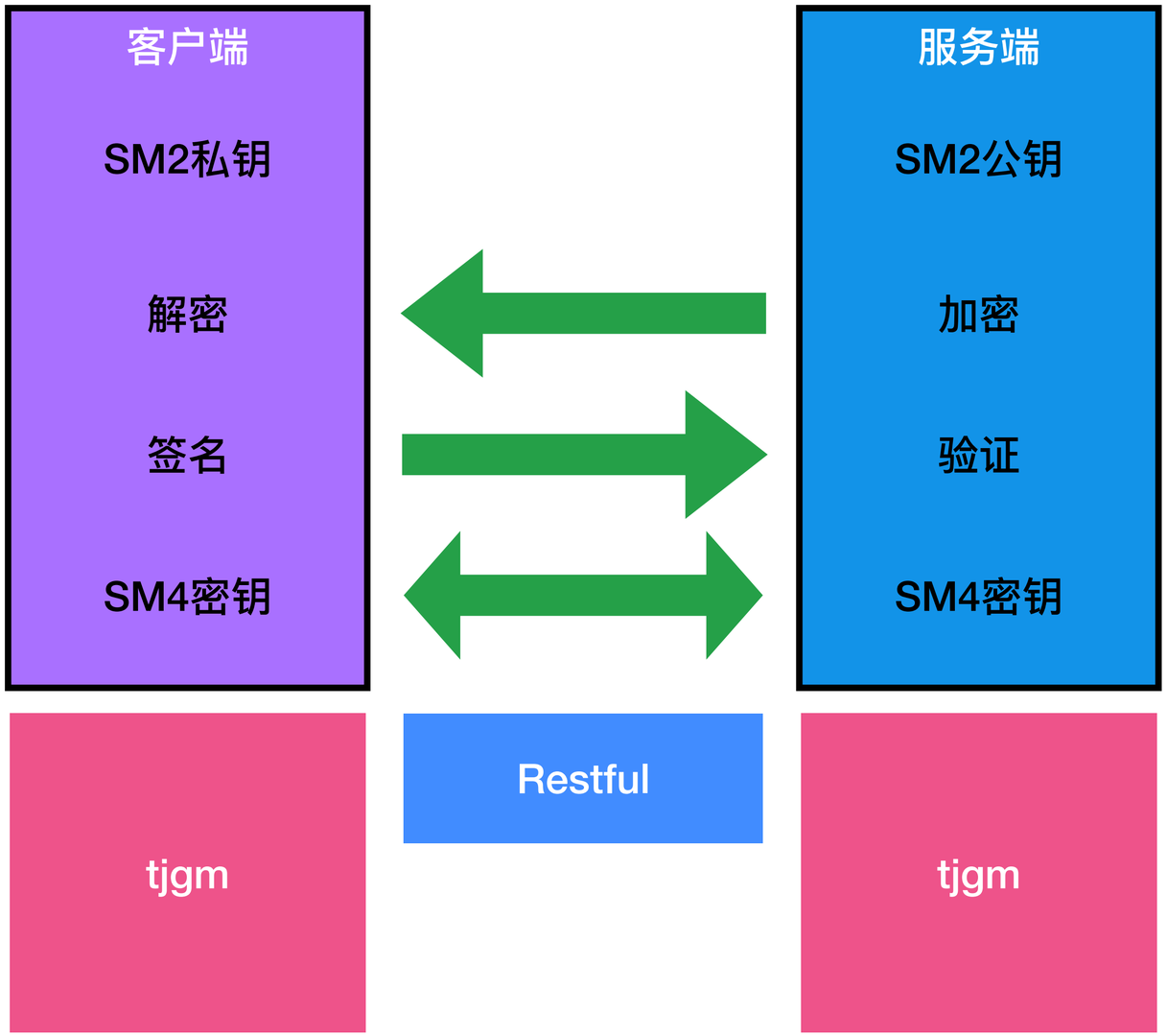
| API | Method | Data | return | Comments |
| --- | --- | --- | --- | --- |
| **verify** | POST | msg string<br>sign string | bool | 签名验证操作 |
| **encrypt** | Get | msg string<br>encrypt string | N/A | 加解密操作 |
| **sm4** | Get | msg string<br>encrypt string | N/A | sm4算法操作 |



# 国密算法的应答实现

## 客户端实现

生成密钥对
解密
签名
sm4

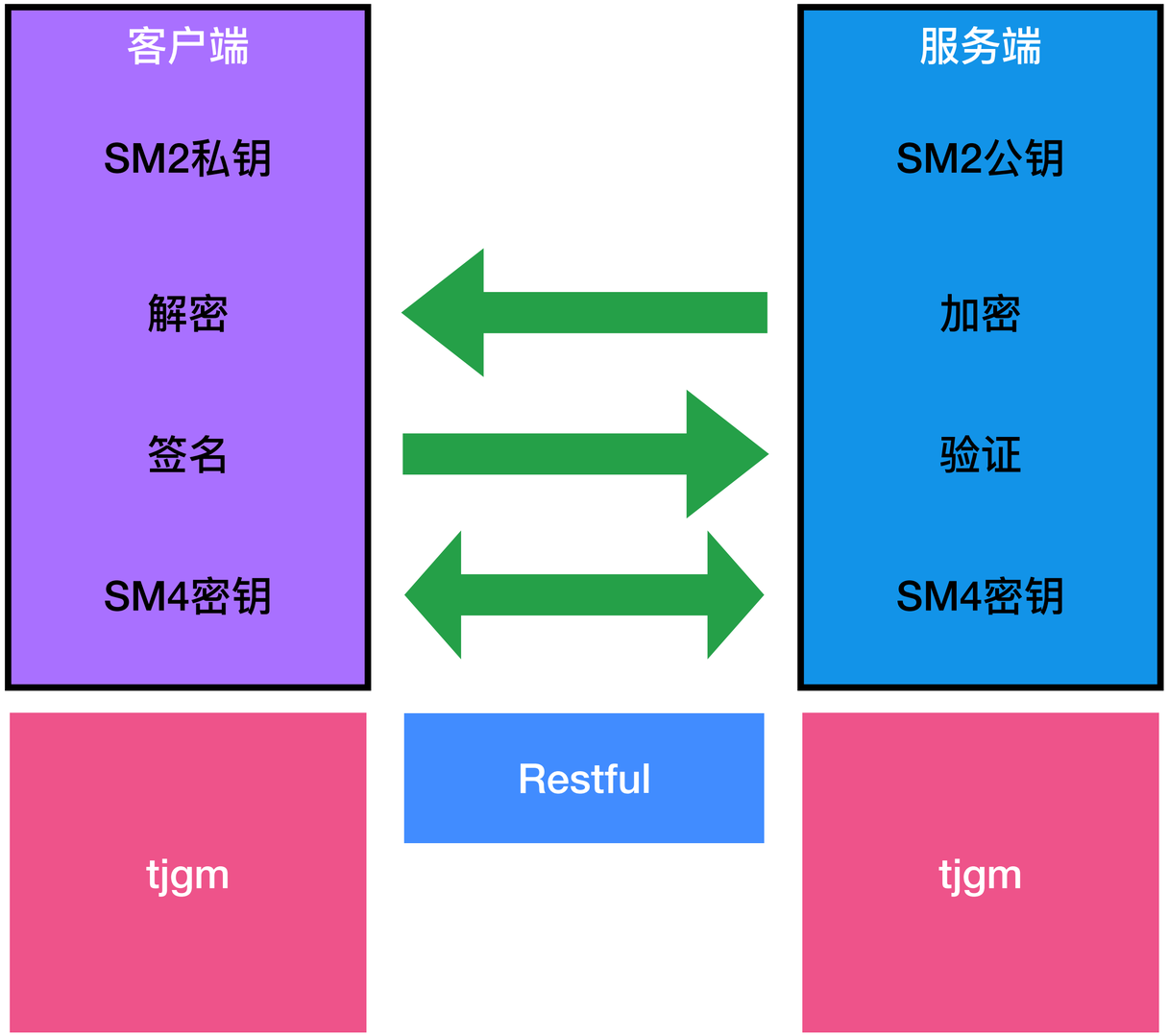


# 客户端实现

## 生成密钥

Client
arg1 key 存储位置
arg2 操作（生成key/解密/签名/sm4）
arg3 server地址

```
func main() {
    // args 1 as key store path
    // args 2 as method
    // args 3 as server endpoint(optional)
    path = os.Args[1]
    if os.Args[2] == "generate" {
        fmt.Println("generate key pair at " + path)
        source, _ := workshop.GenerateSM2Instance(workshop.TJ)
        source.SaveFile(path+priFile, path+pubFile)
    }
```



# 客户端实现

## sm2相关

```
if os.Args[2] == "sign" {
    fmt.Println("sign")
    priv, _ := workshop.LoadFromPriPem(path + priFile)
    // generate original data
    now := time.Now()
    year, month, day := now.Date()
    today_str := fmt.Sprintf("%d-%d-%d 00:00:00", year, month, day)
    var msg = []byte(today_str)
    encodedMsg := hex.EncodeToString(msg)

    // do signature
    //fmt.Println(string(msg))
    //fmt.Println(string(encodedMsg))
    sign, _ := workshop.DegistAndSign(msg, priv)
    encodedStr := hex.EncodeToString(sign)
    //fmt.Println(encodedStr)
    // send to server

    bodyReader := strings.NewReader(`{"msg" : "` + string(encodedMsg) + `", "sign": "` + string(enco
    httpRequest, _ := http.NewRequest("POST", "http://"+os.Args[3]+"/verify", bodyReader)
    httpRequest.Header.Set("Content-Type", "application/json")
    client := http.Client{}
    response, _ := client.Do(httpRequest)
    defer response.Body.Close()
    body, _ := ioutil.ReadAll(response.Body)
    fmt.Printf(string(body))
}
```

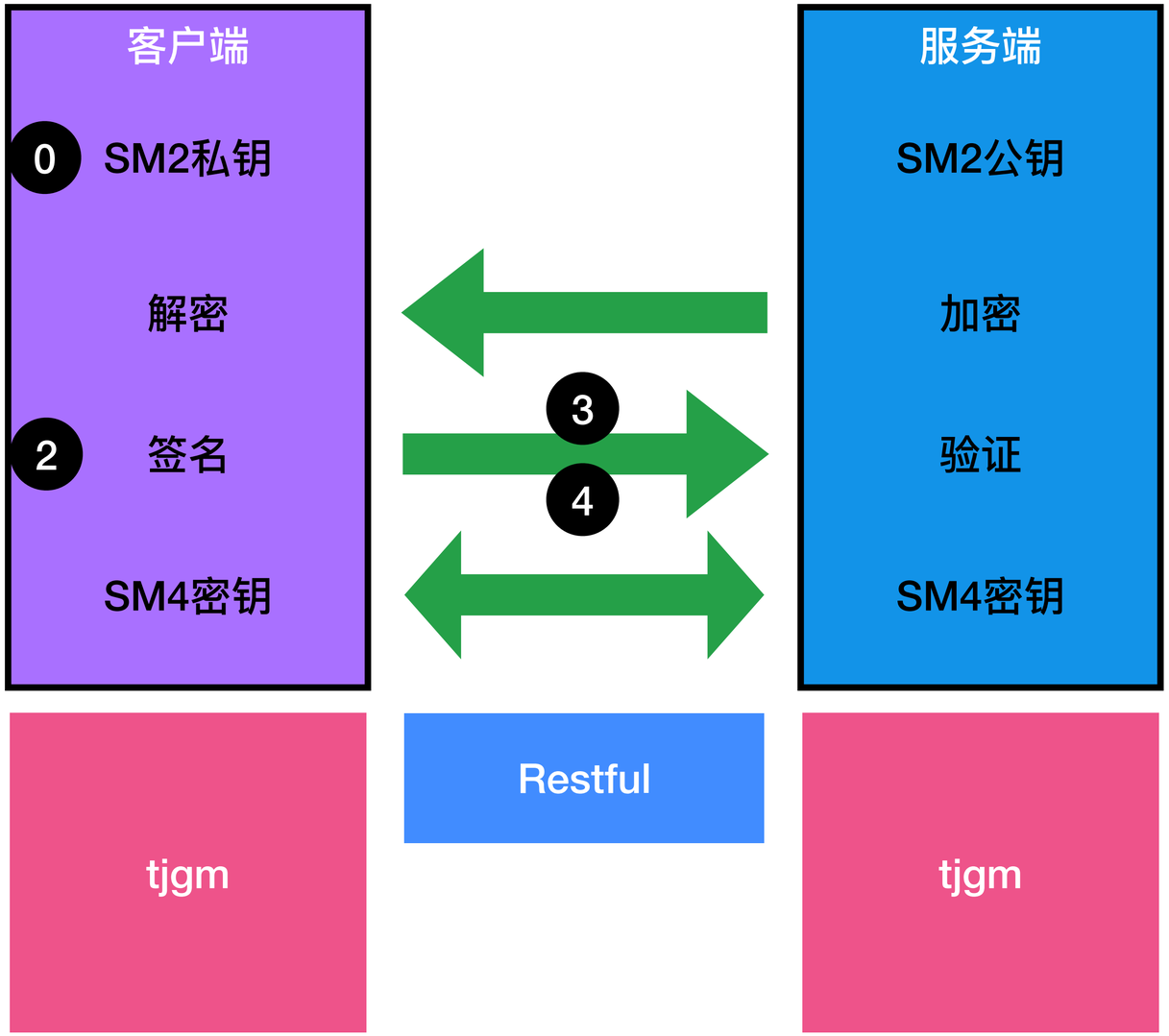
0

1

2

3

4



# 客户端实现

## sm2相关

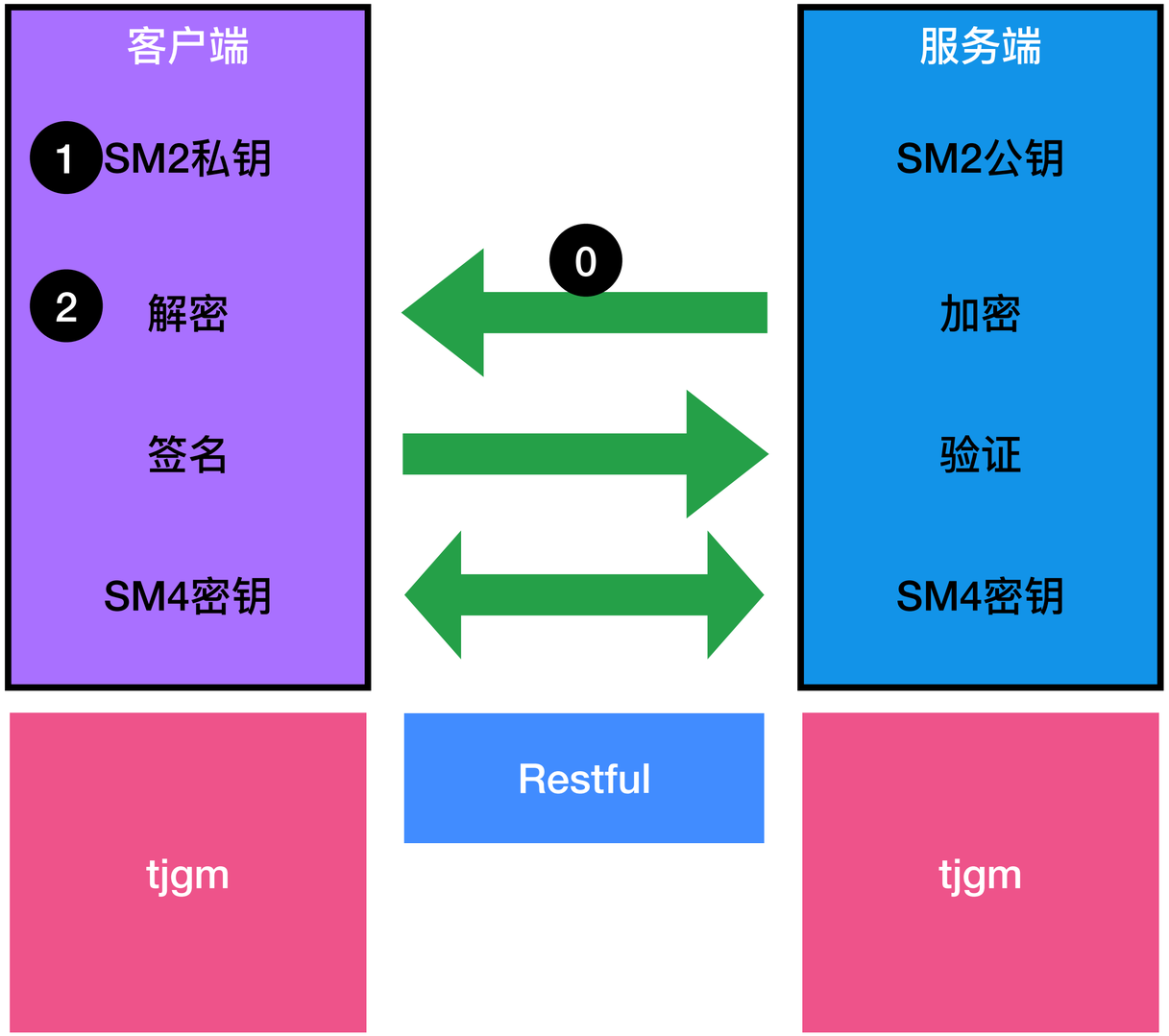
```
if os.Args[2] == "decrypt" {
    fmt.Println("decrypt")
    httpRequest, _ := http.NewRequest("GET", "http://"+os.Args[3]+"/encrypt", nil)
    httpRequest.Header.Set("Content-Type", "application/json")
    client := http.Client{}
    response, _ := client.Do(httpRequest)
    defer response.Body.Close()
    body, _ := ioutil.ReadAll(response.Body)
    //fmt.Println(string(body))
    var data Encrypt
    err := json.Unmarshal(body, &data)
    if err != nil {
        fmt.Println(err)
    }
    priv, _ := workshop.LoadFromPriPem(path + priFile)

    test, err := hex.DecodeString(data.Encrypt)
    if err != nil {
        fmt.Println(err)
    }
    decrypted, err := priv.Decrypt(test)
    if err != nil {
        fmt.Println(err)
    }
    originalmsg, _ := hex.DecodeString(data.Msg)
    fmt.Println(string(decrypted) == string(originalmsg))
}
```

0

1

2

3



# 客户端实现

## sm4相关

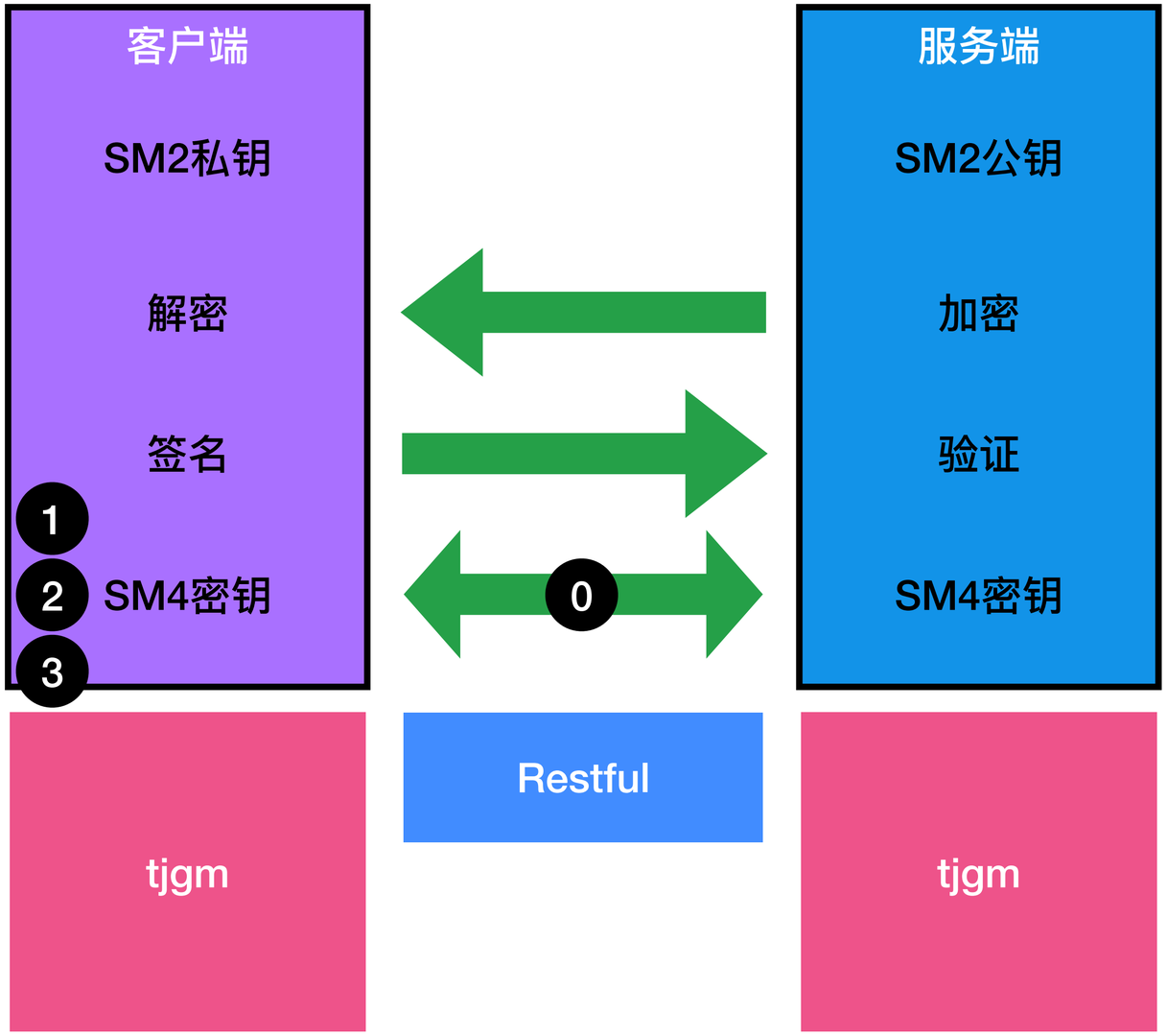
```
if os.Args[2] == "sm4" {
    fmt.Println("sm4 decrypt")
    httpRequest, _ := http.NewRequest("GET", "http://"+os.Args[3]+"/sm4", nil)
    httpRequest.Header.Set("Content-Type", "application/json")
    client := http.Client{}
    response, _ := client.Do(httpRequest)
    defer response.Body.Close()
    body, _ := ioutil.ReadAll(response.Body)
    //fmt.Println(string(body))
    var data Encrypt
    err := json.Unmarshal(body, &data)
    if err != nil {
        fmt.Println(err)
    }
    SM4Key, _ := workshop.GenerateSM4Instance(workshop.TJ)

    test, err := hex.DecodeString(data.Encrypt)
    if err != nil {
        fmt.Println(err)
    }
    decrypted, err := SM4Key.Decrypt(test, "ecb")
    if err != nil {
        fmt.Println(err)
    }
    originalmsg, _ := hex.DecodeString(data.Msg)
    fmt.Println(string(decrypted) == string(originalmsg))
}
```

0

1

2

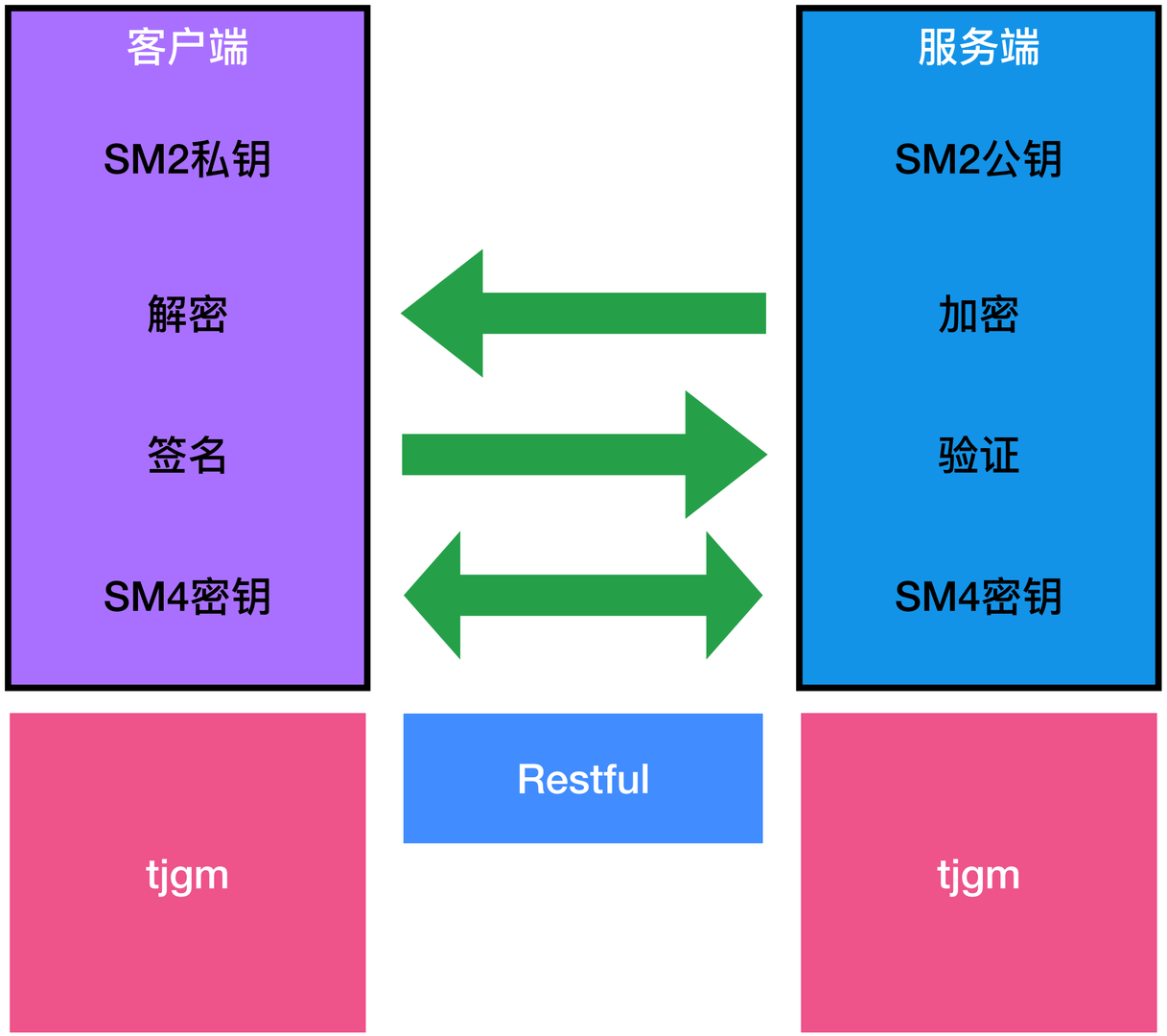
3



# 国密算法的应答实现

## 服务端实现

服务监听
解密
签名
sm4

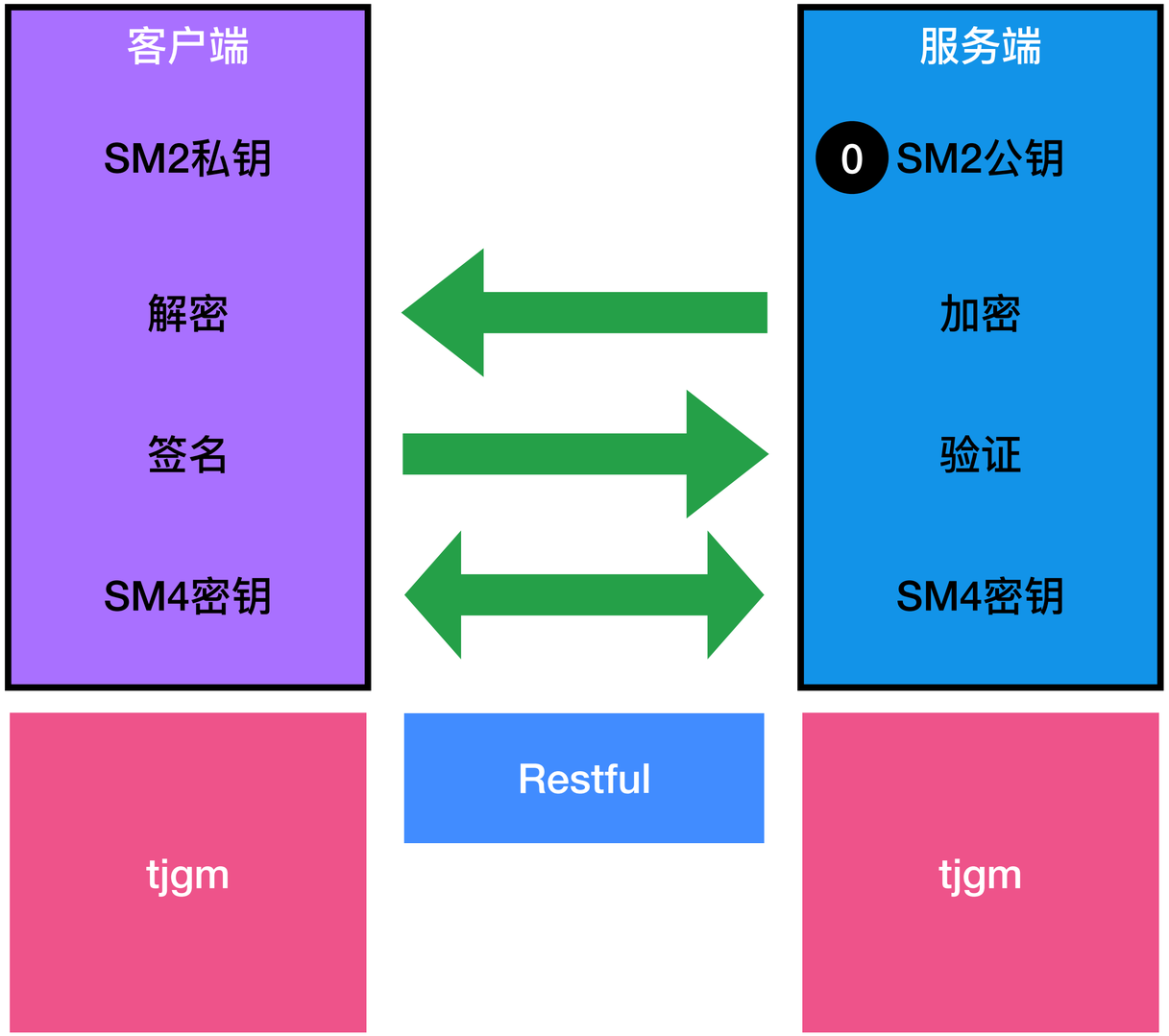


# 服务端实现

## restful注册，密钥加载

```go
func main() {
    path = os.Args[1]
    ws := new(restful.WebService)
    ws.Route(ws.POST("/verify").To(verify))
    ws.Route(ws.GET("/encrypt").To(encrypt))
    ws.Route(ws.GET("/sm4").To(sm4))

    restful.Add(ws)
    Key, err = workshop.LoadFromPubPem(path + pubFile)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("start server")
    }
    log.Fatal(http.ListenAndServe("127.0.0.1:8080", nil))
}
```

0



# 服务端实现

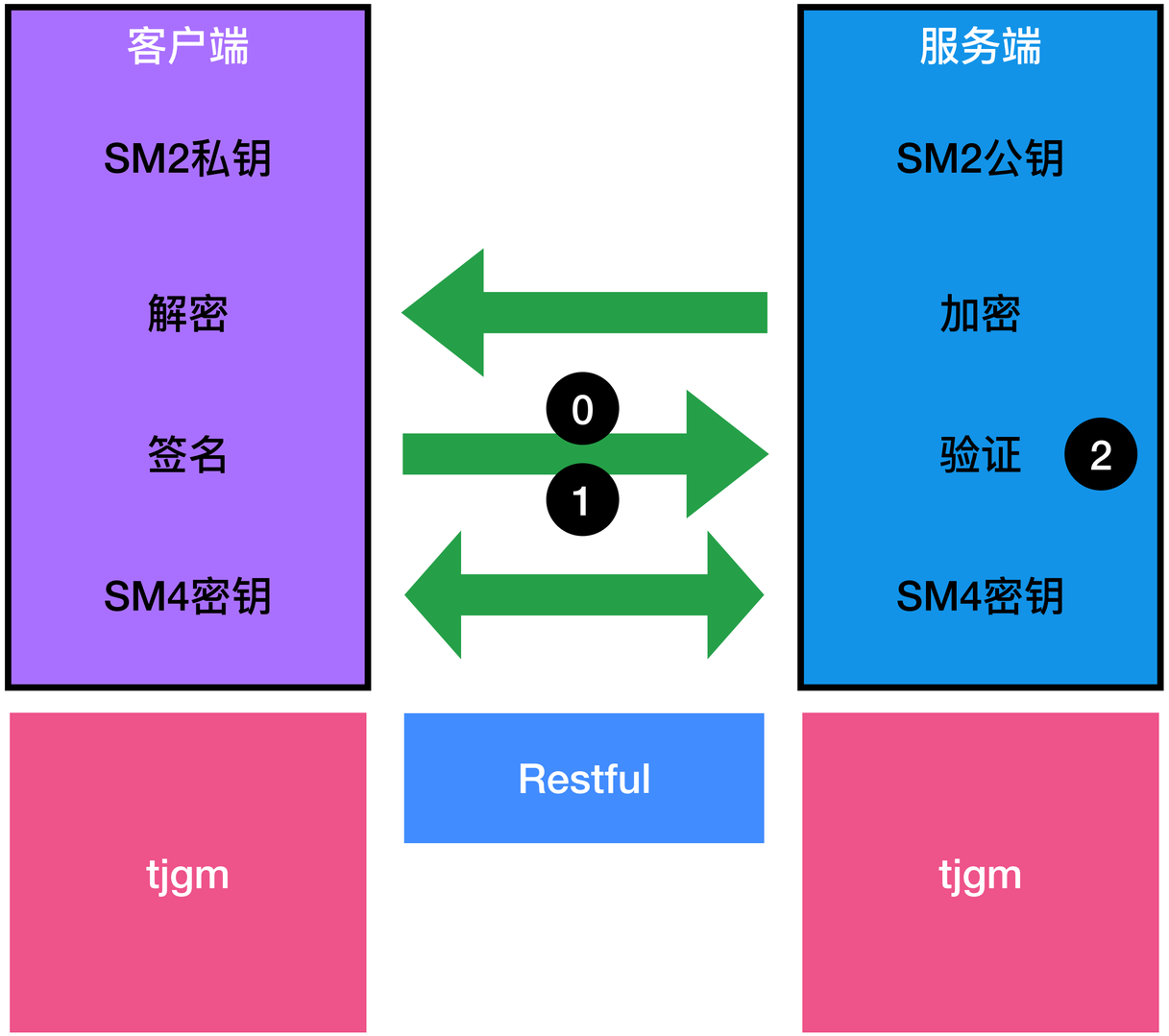
## sm2相关

```
func verify(req *restful.Request, resp *restful.Response) {
    log.Println("verify")
    //get origin data from request
    any := make(Anything)
    req.ReadEntity(&any)
    //fmt.Println(any)
    msg, ok := any["msg"].(string)
    if !ok {
        fmt.Println("read failed")
    }
    sign, ok := any["sign"].(string)
    if !ok {
        fmt.Println("read failed")
    }
    originalmsg, _ := hex.DecodeString(msg)

    //get signature from request
    dummy, _ := hex.DecodeString(sign)
    //do verify
    //fmt.Println(originalmsg)
    data := workshop.DegistAndVerify([]byte(originalmsg), dummy, Key)
    //return
    io.WriteString(resp, strconv.FormatBool(data))
}
```

0

1

2

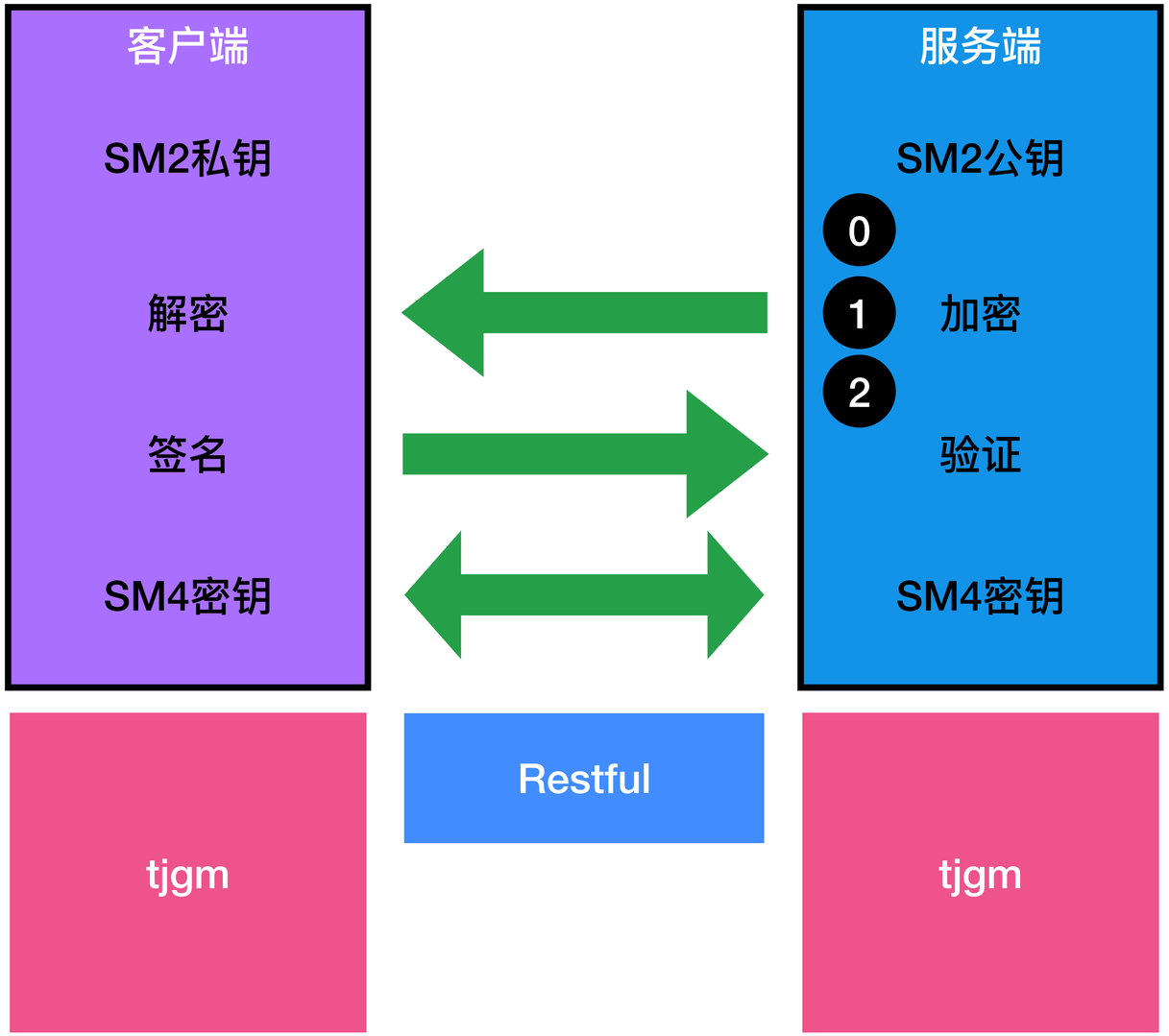


# 服务端实现

## sm2相关

0
1
2

```go
func encrypt(req *restful.Request, resp *restful.Response) {
    log.Println("encrypt")
    now := time.Now()
    year, month, day := now.Date()
    today_str := fmt.Sprintf("%d-%d-%d 00:00:00", year, month, day)
    var msg = []byte(today_str)
    encodedMsg := hex.EncodeToString(msg)
    data, _ := Key.Encrypt(msg)
    encodedStr := hex.EncodeToString(data)
    //log.Println(encodedStr)
    //todo string data is not human-readable
    io.WriteString(resp, `{"msg" : "`+string(encodedMsg)+`", "encrypt": "`+st
}
```



# 服务端实现

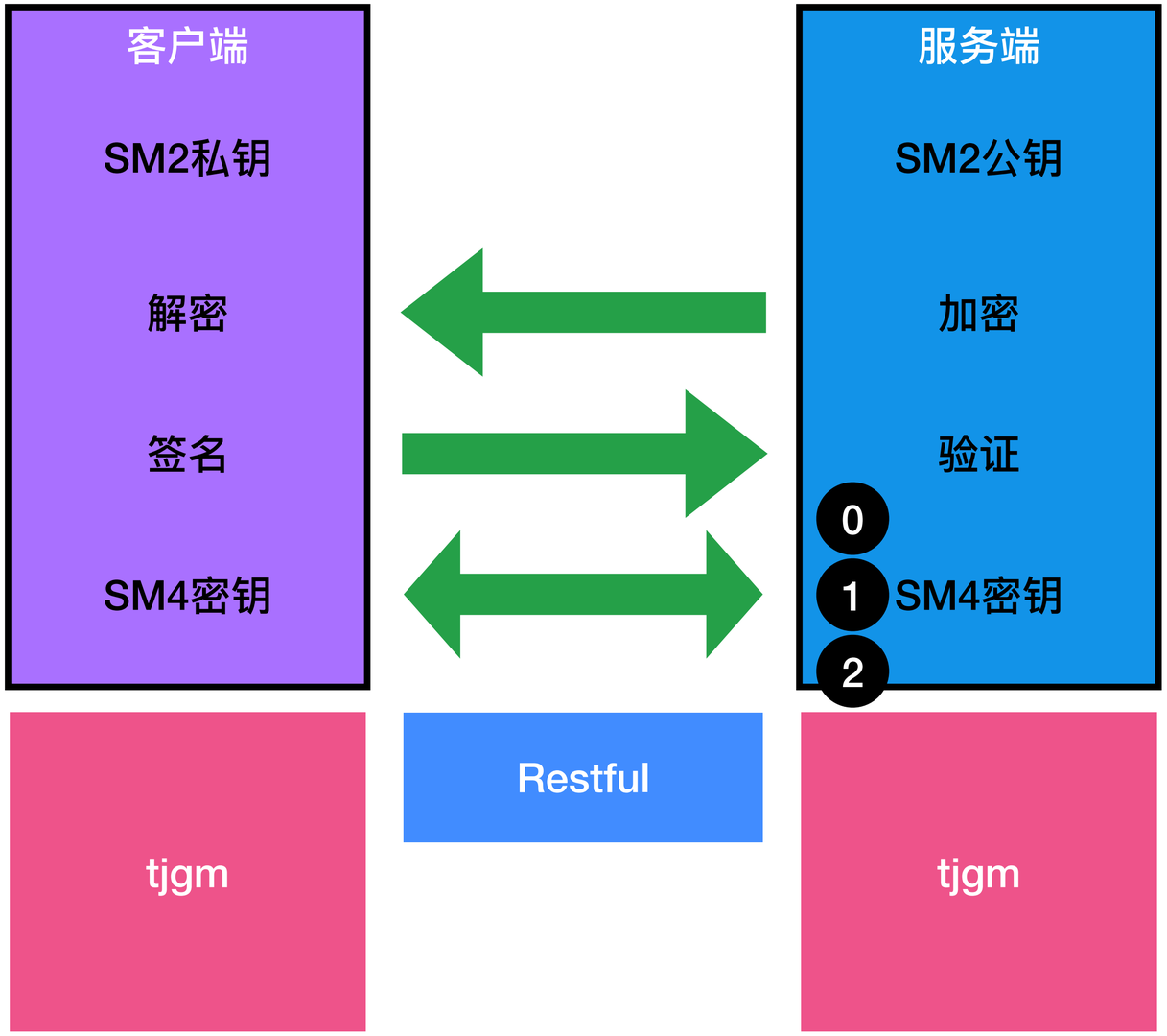
## sm4相关

```
func sm4(req *restful.Request, resp *restful.Response) {
    log.Println("sm4 encrypt")
    SM4Key, _ := workshop.GenerateSM4Instance(workshop.TJ)
    now := time.Now()
    year, month, day := now.Date()
    today_str := fmt.Sprintf("%d-%d-%d 00:00:00", year, month, day)
    var msg = []byte(today_str)
    data, _ := SM4Key.Encrypt(msg, "ecb")

    encodedMsg := hex.EncodeToString(msg)
    encodedStr := hex.EncodeToString(data)
    //log.Println(encodedStr)
    //todo string data is not human-readable
    io.WriteString(resp, `{"msg" : "`+string(encodedMsg)+`", "encrypt": "`+string(encod
}
```

0

1

2



# QA

Thanks