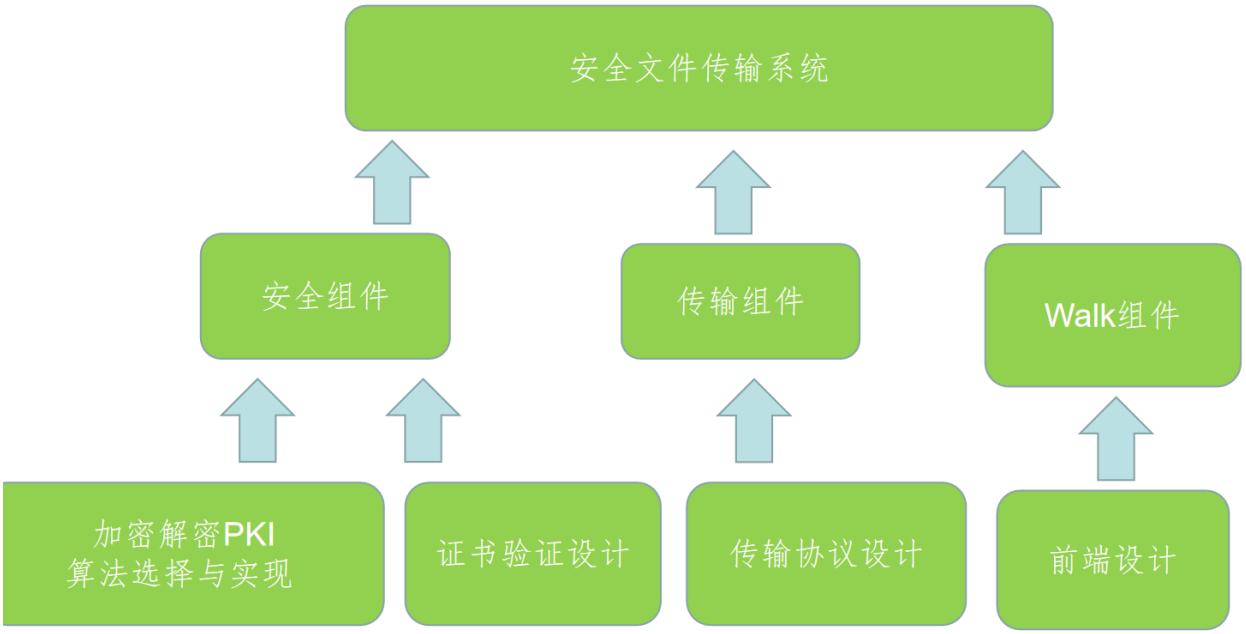
安全文件传输系统

Select	
Status	0.0
Tags	
Text	

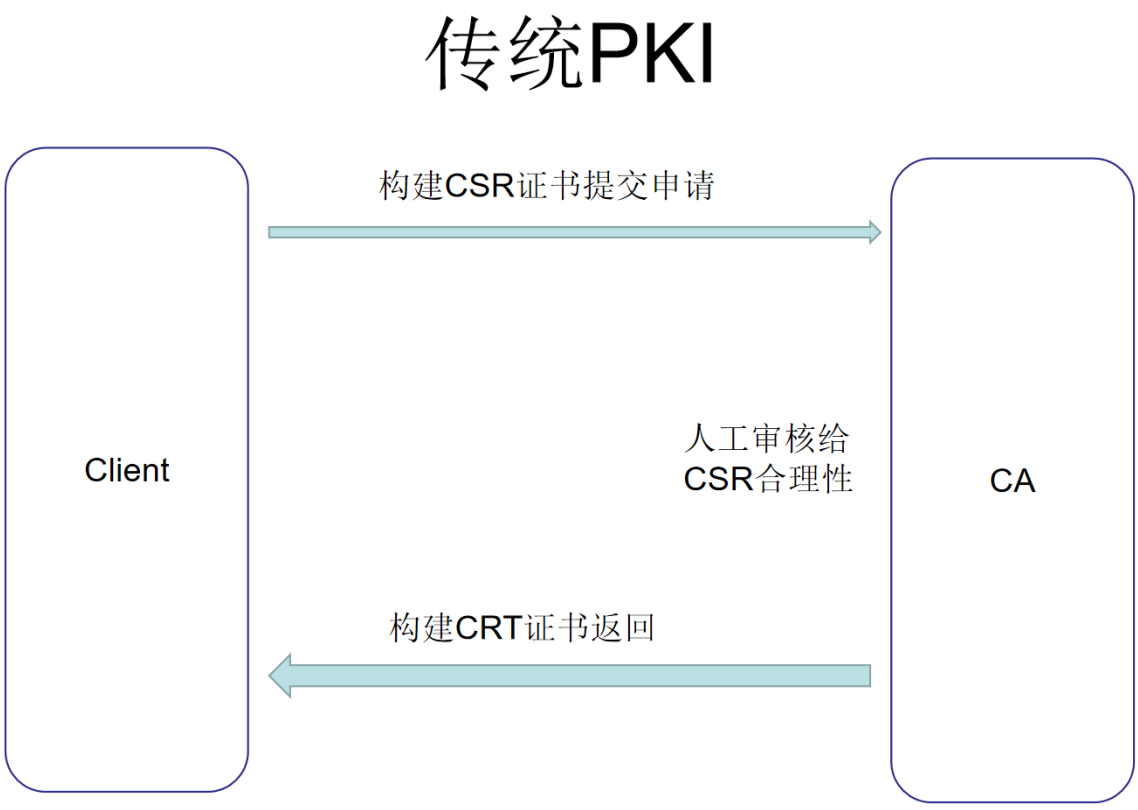
安全文件传输系统

整体设计

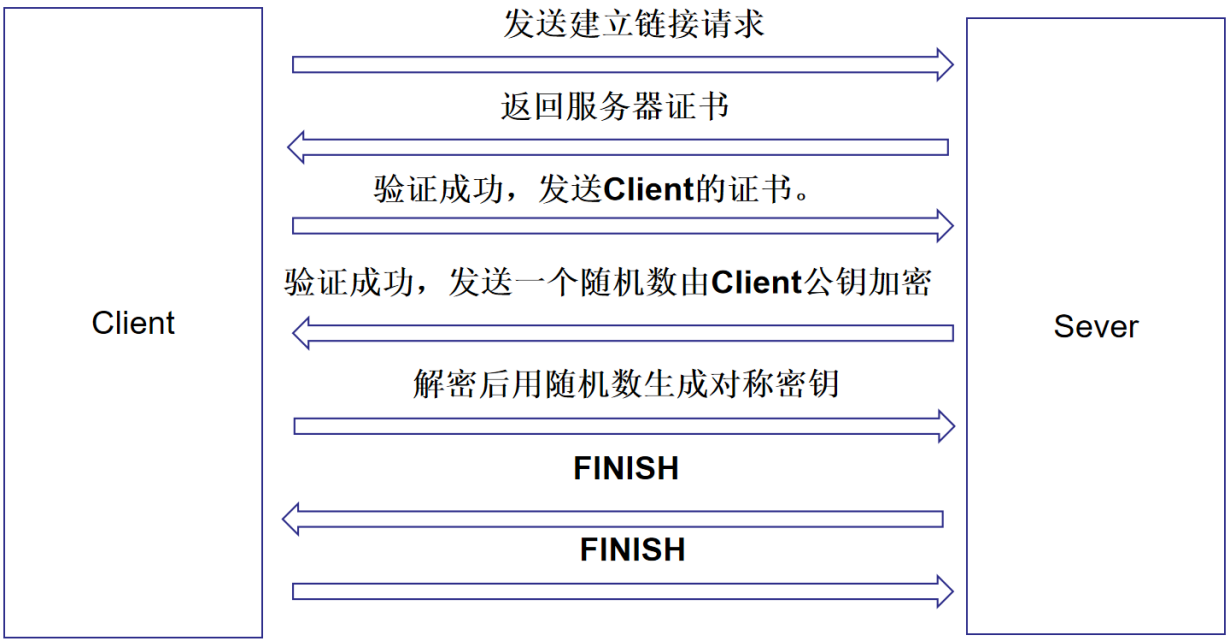
架构



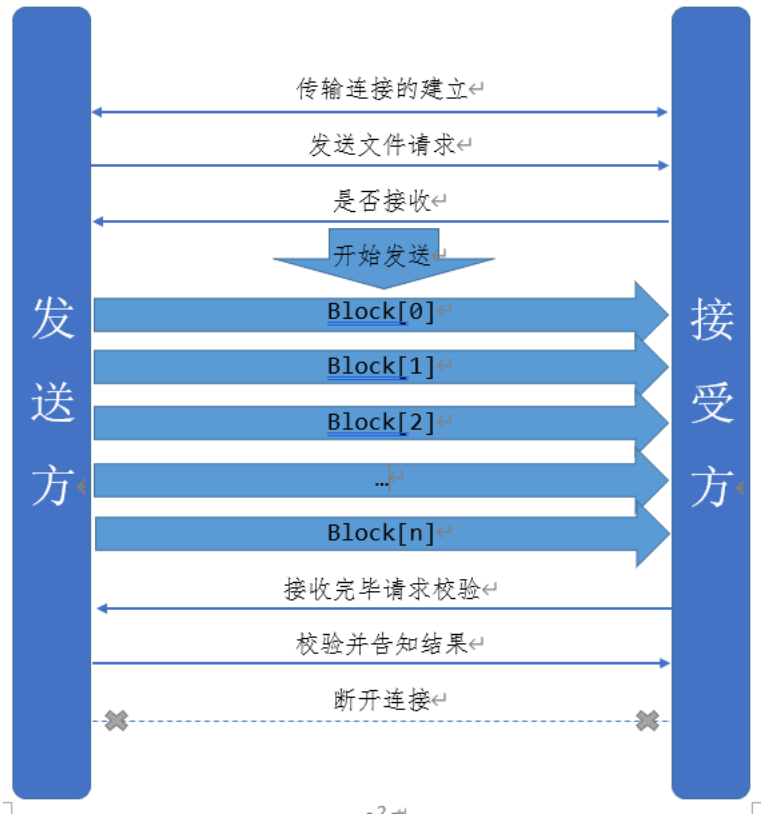
请求CA认证过程



请求链接过程（双向证书验证）


Client
发送建立链接请求
返回服务器证书
验证成功，发送Client的证书。
验证成功，发送一个随机数由Client公钥加密
解密后用随机数生成对称密钥
FINISH
FINISH
Sever

传输模型


发送方
接受方
传输连接的建立
发送文件请求
是否接收
开始发送
Block[0]
Block[1]
Block[2]
…
Block[n]
接收完毕请求校验
校验并告知结果
断开连接

开发环境 GO

图形库Walk

需要这两个库

go get github.com/lxn/walk

安装工具

go get github.com/akavel/rsrc

创建main.manifest文件

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
    <assemblyIdentity version="1.0.0.0" processorArchitecture="*" name="SomeFunkyNameHere" type="win32"/>
    <dependency>
        <dependentAssembly>
            <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls" version="6.0.0.0" processorArchitecture="*" publicKeyTo
        </dependentAssembly>
    </dependency>
    <application xmlns="urn:schemas-microsoft-com:asm.v3">
        <windowsSettings>
            <dpiAwareness xmlns="http://schemas.microsoft.com/SMI/2016/WindowsSettings">PerMonitorV2, PerMonitor</dpiAwareness>
            <dpiAware xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">True</dpiAware>
        </windowsSettings>
    </application>
</assembly>
```

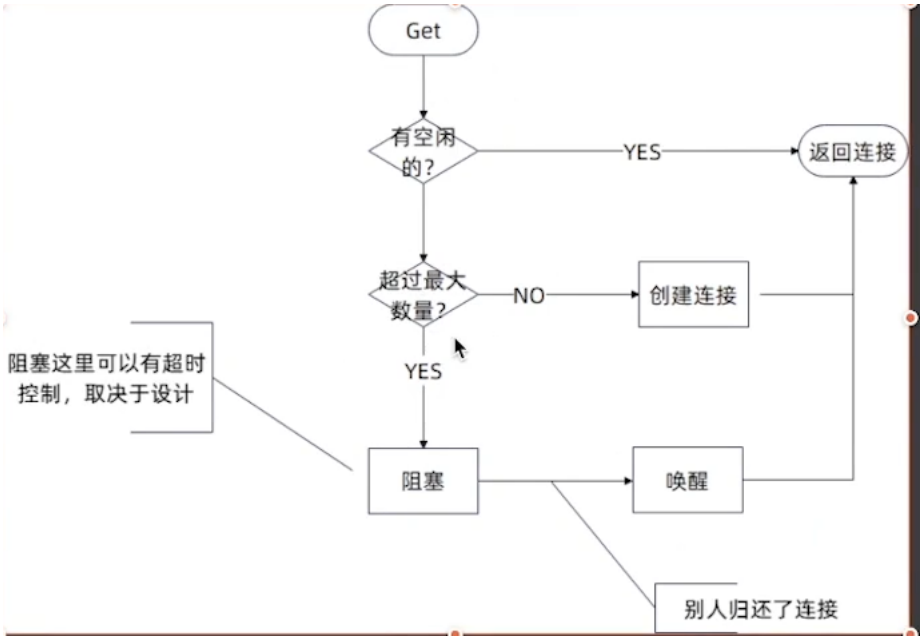
生成rsrc.syso

```
rsrc -manifest main.manifest -o rsrc.syso

go build
```

模块需求分析

首先分析都需要哪些特定的功能模块

连接池选用silenceper/pool



PKI模块

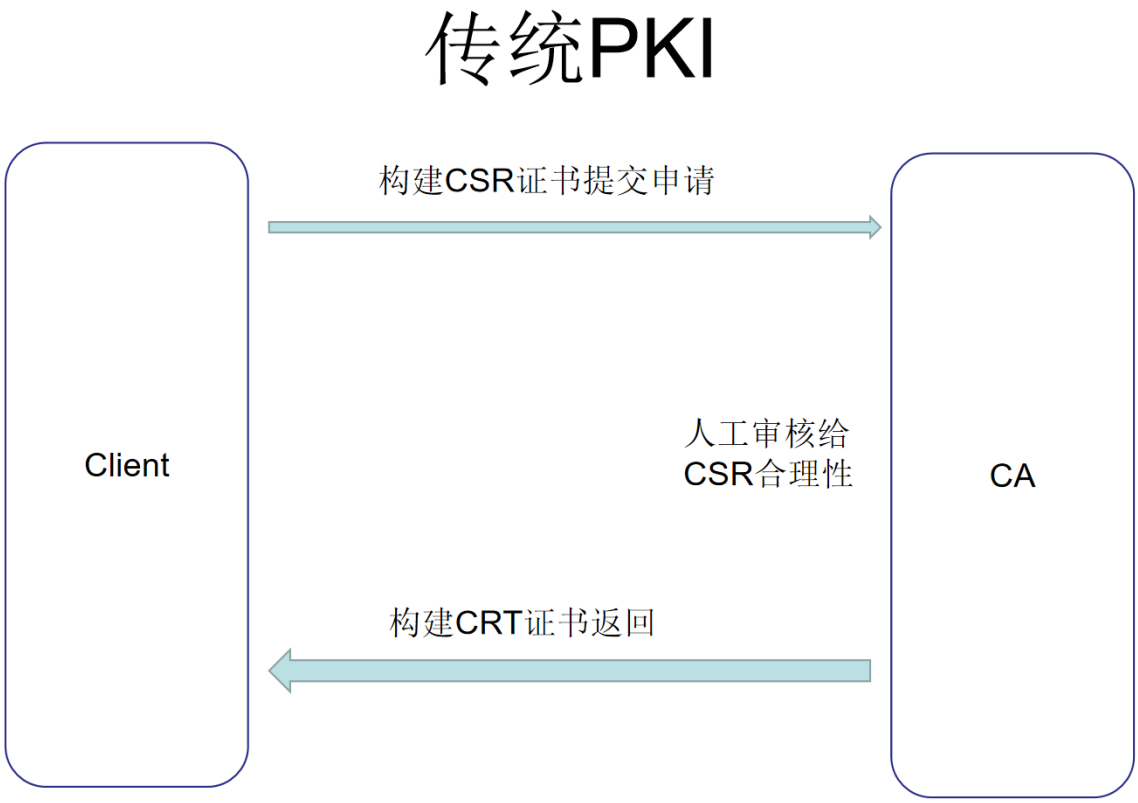
为Client去颁布对应的证书，再第一次的时候去注册信息人工审核（这里就直接通过了）。为可以加入系统的Client去颁发证书。基本流程。、

了创建一个PKI（Public Key Infrastructure，公钥基础设施）发布功能模块，您需要首先了解PKI的核心组件和它们在Go语言中的实现方式。以下是一些建议和指导：

1. 证书颁发机构（CA，Certificate Authority）：CA是一个受信任的组织，负责颁发、管理和撤销数字证书。在Go语言中，您可以使用`crypto/x509`和`crypto/x509/pkix`包来实现CA功能。
2. 创建根证书：根证书是CA的自签名证书。使用`x509.CreateCertificate`函数，您可以创建一个根证书。为了创建根证书，您需要填写证书的主题（Subject），有效期，密钥用途等参数。
3. 生成密钥对：为了创建证书，您需要生成一个密钥对，包括公钥和私钥。您可以使用`crypto/rsa`或`crypto/ecdsa`包生成密钥对。
4. 创建和签发证书：使用`x509.CreateCertificate`函数，您可以创建和签发证书。为了创建证书，您需要填写证书的主题（Subject），有效期，密钥用途等参数。证书需要由CA的私钥签名。
5. 序列化和反序列化证书：使用`x509.MarshalPKIXPublicKey`和`x509.ParsePKIXPublicKey`函数，您可以实现公钥的序列化和反序列化。使用`x509.MarshalPKCS1PrivateKey`和`x509.ParsePKCS1PrivateKey`函数，您可以实现私钥的序列化和反序列化。
6. 证书存储和分发：将生成的证书和密钥存储在文件或数据库中。在分发证书时，可以使用PEM（Privacy Enhanced Mail）格式编码，这是一种ASCII编码的格式。Go语言中的`encoding/pem`包提供了编码和解码PEM格式的函数。

zheng

```go
// VerifyCrt 检测证书正确性传PEM块
func VerifyCrt(CertData []byte) bool {
    // 加载CA证书，这里需要确保CA证书已经存在
    caCert := loadCACertificate()
    // 创建证书池并添加CA证书
    certPool := x509.NewCertPool()
    certPool.AddCert(caCert)

    // 解析证书数据
    block, _ := pem.Decode(CertData)
    if block == nil || block.Type != "CERTIFICATE" {
        panic( v: "证书块获取失败")
        return false
    }
    cert, err := x509.ParseCertificate(block.Bytes)
    if err != nil {
        fmt.Println( a...: "Error parsing certificate:", err)
        return false
    }
    // 验证证书签名和有效期
    opts := x509.VerifyOptions{
        Roots:       certPool,
        CurrentTime: time.Now(),
    }

    if _, err := cert.Verify(opts); err != nil {
        fmt.Println( a...: "证书验证失败:", err)
        return false
    }

    return true
}
```



证书签发

```
证书签名
证书签名可以使用标准库 crypto/x509 的 CreateCertificate 方法来签名。该方法需要以下5个参数：

rand: 随机数，使用 rand.Reader 即可。
template: 证书签名请求，即 CSR。
parent: 父级证书，根证书是自签的，直接用自己的 csr，中级证书用根证书来签名，终端证书使用中级证书签名。
pub: 第一步生成的私钥对应的公钥证书，可以使用 key.Public() 获取。
priv: 父级证书私钥。
```

双向证书验证设计

1. 客户端和服务器设计：首先，你需要设计客户端和服务器的架构。客户端负责发送文件，而服务器负责接收文件。你可以使用Go语言的net包中的TCP相关函数来实现这一点。
2. 加密：为了确保文件在传输过程中的安全性，你应该使用加密算法对文件进行加密。在Go语言中，你可以使用crypto包来实现加密功能。常用的加密算法包括AES、RSA等。
3. 数据完整性校验：在文件传输过程中，为了确保数据的完整性，你可以使用哈希算法为文件生成校验和。在Go语言中，你可以使用hash包中的函数来实现这一点。常用的哈希算法包括SHA-256、SHA-512等。
4. 身份验证：为了确保只有授权用户可以访问服务器，你需要实现身份验证机制。这可以通过使用用户名和密码或者证书进行实现。在Go语言中，你可以使用crypto/tls包来实现证书的生成和验证。

5. 错误处理与重传机制：在文件传输过程中，可能会出现丢包、重复包等情况。为了应对这些问题，你需要实现错误处理和重传机制。你可以通过设置超时、序号和确认号等方法来实现这一点。

传输协议设计

数据包设计

守护进程保证长连接（KEEPALIVE）

`context.WithCancel(context.Background())` 是一个从Go标准库中 `context` 包创建可取消上下文（cancelable context）的函数。它返回一个新的上下文（`ctx`）和一个关联的取消函数（`cancel`），可以用于取消上下文。

`context.Background()` 是一个空的上下文，它不能被取消、不包含任何值、没有超时设置。我们通过将 `context.Background()` 传递给 `context.WithCancel` 函数，从而创建一个基于空上下文的可取消上下文。这使得新创建的上下文可在需要时通过调用 `cancel` 函数来取消。

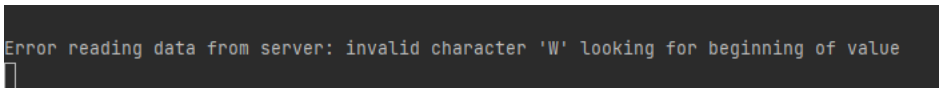
`ctx` 是一个 `context.Context` 类型的变量，它可以用于携带跨API边界和进程之间的截止时间、取消信号以及其他请求范围的值。在本例中，我们将 `ctx` 传递给守护进程goroutine，这样它就可以检测到上下文何时被取消。

`cancel` 是一个 `context.CancelFunc` 类型的变量，它是一个函数，当调用它时，会取消与 `ctx` 关联

KEEPAlive

```
func main() {
  // 创建一个TCP连接
  conn, err := net.Dial("tcp", "example.com:80")
  if err != nil {
    fmt.Println("Error connecting:", err)
    return
  }
  defer conn.Close()

  // 将 net.Conn 类型断言为 *net.TCPConn
  tcpConn, ok := conn.(*net.TCPConn)
  if !ok {
    fmt.Println("Connection is not a *net.TCPConn")
    return
  }

  // 设置 Keepalive
  err = tcpConn.SetKeepAlive(true)
  if err != nil {
    fmt.Println("Error setting Keepalive:", err)
    return
  }

  // 设置 Keepalive 时间间隔（可选）
  err = tcpConn.SetKeepAlivePeriod(30 * time.Second)
  if err != nil {
    fmt.Println("Error setting Keepalive period:", err)
    return
  }

  // 在这里使用连接进行通信
  // ...
}
```

粘包问题



Github上传

网址https://github.com/SwordFourteen/Security_File_-Transfers

https://github.com/SwordFourteen/Security_File_-Transfers

`ssh-keygen -t rsa -C "810985987@qq.com"`

`git remote add` origin git@github.com:SwordFourteen/Security_File_-Transfers.git

```
git remote add origin https://github.com/SwordFourteen/Security_File_-Transfers
```

`ssh -T git@github.com`