知识库

智能信息安全与对抗 — 学习知识库

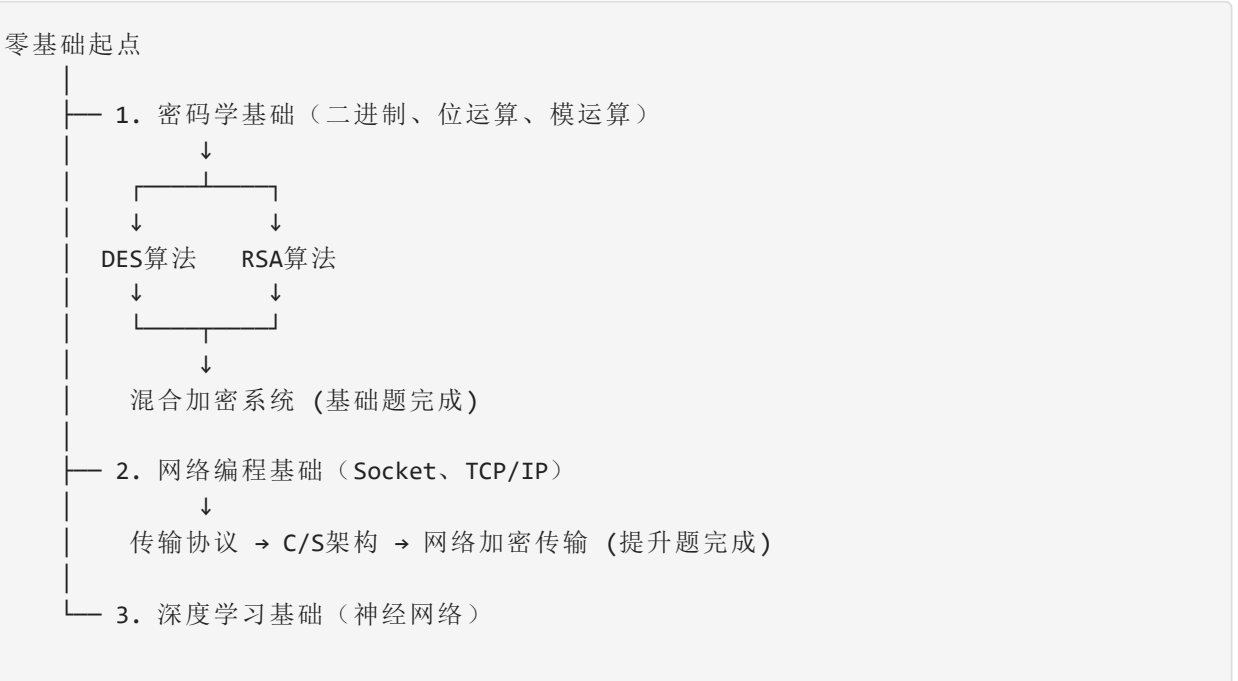
> 适用对象：零基础工科学生（EE背景） 目标：从零开始理解我们做的三个实验系统的全部原理和代码

项目总览

我们完成了三个层次的实验系统：

层次	系统名称	核心知识点
基础题	DES+RSA 文件混合加解密系统	密码学、对称加密、非对称加密
提升题	网络数据加密传输系统	网络编程、TCP协议、C/S架构
进阶题	AIGC-图像生成系统	深度学习、扩散模型、Web开发

学习路径（推荐顺序）



```
                ↓
扩散模型 → Stable Diffusion → Web部署 → AIGC (进阶题完成)
```

文档索引

基础篇 — 密码学与加解密

文档	内容	前置知识
密码学基础	二进制、加密体系、对称/非对称	无
DES算法详解	DES加密过程、代码逐行解析	密码学基础
RSA算法详解	RSA数论原理、代码解析	密码学基础
混合加密系统详解	DES+RSA架构、文件格式、GUI	DES+RSA

提升篇 — 网络加密传输

文档	内容	前置知识
网络编程基础	IP/端口、Socket、TCP/UDP	无
传输协议详解	自定义协议、分块传输	网络编程基础
C/S架构详解	服务端/客户端代码解析	传输协议
Tkinter GUI基础	Python图形界面编程	Python基础

进阶篇 — AIGC图像生成

文档	内容	前置知识
深度学习基础	神经网络、CNN、PyTorch	无
扩散模型原理	前向/反向扩散、去噪	深度学习基础
Stable Diffusion详解	SD管线代码解析	扩散模型
FastAPI后端详解	REST API、异步任务	Python基础

文档	内容	前置知识
React前端基础	组件、状态、API调用	HTML/JS基础

综合

文档	内容
知识地图	所有知识点的关联图谱

学习建议

- **不要急于一次看懂所有代码**：先理解核心概念，再回头看代码
- **动手运行**：每个系统都有完整的测试，运行看看效果
- **从基础题开始**：密码学是整个信息安全的基石
- **善用本文档中的类比**：为了帮助你理解，我用了很多生活化的类比 # 密码学基础

> 从零开始，适合没学过密码学的EE学生

---

1. 为什么需要密码？

一个生活场景

想象你要寄一封信给朋友，但信可能会经过很多人的手。你不想让别人看到信的内容，怎么办？

- **方案A**：把信锁在一个盒子里寄出去 → 但朋友没有钥匙打不开
- **方案B**：你和朋友事先约定好一种”暗号” → 这就是**密钥**
- **方案C**：你把盒子锁上，朋友用他自己的钥匙打开 → 这就是**非对称加密**

密码学的核心目标

1. **机密性**：只有授权的人能看懂内容
2. **完整性**：内容没有被篡改
3. **认证**：确认发送者的身份

4. **不可否认性**：发送者不能否认自己发过

---

2. 你需要预先知道的数学知识

2.1 二进制（Bit）

计算机所有数据本质上是 0 和 1 的序列。

十进制	二进制	说明
0	00000000	一个字节(Byte)=8位(bits)
1	00000001	
2	00000010	进位
255	11111111	一个字节最大值
‘A’	01000001	ASCII编码

关键理解：任何文件（文本、图片、视频）在计算机眼里都是一长串二进制数。加密就是把这些二进制数”搅乱”，解密就是”恢复”。

2.2 模运算（Modular Arithmetic）

通俗理解：就像钟表上的加法。

- 现在是 10 点，过 5 小时是几点？ 10 + 5 = 15，15 mod 12 = **3点**
- `mod` 就是”除以某个数取余数”

```
15 mod 12 = 3    (15 ÷ 12 = 1 余 3)
17 mod 12 = 5    (17 ÷ 12 = 1 余 5)
8 mod 12 = 8     (8 ÷ 12 = 0 余 8)
```

为什么重要：RSA算法的核心就是模幂运算 `c = m^e mod n`。

2.3 异或运算（XOR）

XOR 是密码学中最最最重要的运算，没有之一。

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XOR的神奇性质：两次 XOR 恢复原值

```
明文 XOR 密钥 = 密文
密文 XOR 密钥 = 明文
```

这其实就是最原始的加密！DES算法大量使用了 XOR。

---

3. 对称加密 vs 非对称加密

3.1 对称加密

```
        相同密钥
     ┌──────────┐
     ↓          ↓
明文 → [加密] → 密文 → [解密] → 明文
```

特点： - 加密和解密用**同一个密钥** - 速度快，适合加密大量数据 - 问题：密钥怎么安全地传给对方？

类比：你有一个保险箱，你和朋友各有一把相同的钥匙。你锁上箱子寄给朋友，朋友用他的钥匙打开。

代表算法：DES、AES

3.2 非对称加密

```
        公钥          私钥
     ┌──────┐    ┌──────┐
     ↓      ↓    ↓      ↓
明文 → [加密] → 密文 → [解密] → 明文
```

特点： - 加密和解密用**不同的密钥**（一对密钥：公钥和私钥） - 公钥可以公开给任何人，私钥自己保密 - 速度慢，只适合加密少量数据

类比：你有一个带锁的信箱，任何人都可以把信投进去（用你的公钥加密），但只有你有钥匙能打开（用私钥解密）。

代表算法：RSA

3.3 混合加密（我们的方案）

思想：结合两者的优点

```
文件数据 → [DES加密] → 密文（速度快）
                          ↑
DES密钥 → [RSA公钥加密] → 加密后的密钥（安全分发）
```

为什么这样做？ - 文件可能很大（几MB甚至更大），用 DES 加密快 - DES密钥很小（几十字节），用 RSA 加密安全 - 既解决了速度问题，又解决了密钥分发问题

---

4. 密钥、明文、密文

基本概念

术语	英文	说明	生活类比
明文	Plaintext	原始的可读数据	信的内容
密文	Ciphertext	加密后的乱码数据	看不懂的暗号
密钥	Key	控制加解密的关键参数	保险箱钥匙
加密	Encryption	明文→密文的过程	锁上箱子
解密	Decryption	密文→明文的过程	打开箱子

一个好密码系统的要求

```
解密(加密(明文, 密钥), 密钥) = 明文
```

即：加密后再解密，必须能**完完整整**地恢复出原来的数据。我们的 19 项测试就是在验证这个。

5. 为什么加密后的数据变大了？

在我们的系统中，加密后的文件比原始文件大，原因是：

1. **PKCS7填充**：DES要求数据是8字节的倍数，不够的要填充
2. **RSA加密的DES密钥头**：我们在文件头部附加了RSA加密后的DES密钥（约256字节）

```
[4字节密钥长度][RSA加密的DES密钥 ~256字节][DES加密的数据]
```

---

6. 常见问题 (FAQ)

Q: 密钥越长越安全吗？

A: 通常是的。128位密钥比56位密钥安全得多。但密钥太长也会导致加解密变慢，需要权衡。

Q: 为什么不用AES要用DES？

A: 这是实验要求。实际应用中确实都用AES了，但DES作为密码学教学的经典算法，能帮助你更好地理解Feistel网络结构。

Q: RSA 2048位够安全吗？

A: 目前足够。2048位RSA被普遍认为是安全的，预计至少到2030年都不会被破解。

下一步

学完这些基础后，建议按顺序阅读： 1. DES算法详解 — 看代码如何实现加密 2. RSA算法详解 — 看非对称加密如何工作 3. 混合加密系统详解 — 看如何组合使用 # 网络编程基础

> 从零开始，适合没学过网络编程的EE学生

---

1. 计算机网络是什么？

一句话理解

计算机网络就是”用线（或无线）把两台计算机连起来，让它们能互相传数据”。

一个生活场景

你和朋友在隔壁房间，想传纸条： - **物理层**：纸条本身（原始数据） - **传输层**：你喊”接好了！“然后把纸条递过去（确保数据到达） - **网络层**：你在纸条上写”给隔壁房间的小明”（地址） - **应用层**：纸条上用中文写的内容（应用程序的数据）

---

2. 你需要知道的网络基础概念

2.1 IP地址

通俗理解：IP地址就是计算机在网络上的”门牌号”。

```
就像：北京市海淀区中关村大街5号
对应：192.168.1.100
```

- IPv4：由4个数字组成，每个0-255，如 `192.168.1.1`
- 本机地址：`127.0.0.1`（localhost，指向自己）
- 查看本机IP：Windows 运行 `ipconfig`

2.2 端口（Port）

通俗理解：端口就是计算机上的”窗户编号”。

一个IP地址可以有很多端口（0-65535），每个端口对应一个应用程序。

```
类比：
  IP地址 = 酒店地址
  端口   = 房间号

  192.168.1.100:80   → 这家酒店的80号房间
  192.168.1.100:8888 → 同一家酒店的8888号房间
```

常见端口： - 80：HTTP（网页） - 443：HTTPS（安全网页） - 22：SSH（远程登录） - 8888：我们的网络加密传输系统

2.3 TCP vs UDP

特性	TCP	UDP
全称	传输控制协议	用户数据报协议
可靠性	**可靠**（保证到达）	不可靠（可能丢包）
顺序	**保证顺序**	不保证顺序
速度	较慢	较快
类比	打电话（持续连接）	寄明信片（不管到不到）
我们的系统	**使用TCP**	—

为什么我们的系统用TCP？ 传输加密文件必须保证数据完整无误，不能丢任何一个字节！

---

3. Socket编程（核心）

3.1 什么是Socket？

通俗理解：Socket是操作系统提供的一个”网络通信的接口”。有了它，你的程序就可以通过网络收发数据。

就像：

```
你的程序 ↔ [Socket接口] ↔ 操作系统 ↔ 网卡 ↔ 网络 ↔ 对方
```

3.2 TCP Socket的工作流程

```
服务端（接收方）                    客户端（发送方）
————————————————                  ————————————————
1. 创建Socket                      1. 创建Socket
2. bind() 绑定端口                  2. connect() 连接服务器
3. listen() 开始监听
4. accept() 等待连接 ————————→  TCP连接建立  ←——— 3. connect()
```

```
5. recv() 接收数据          ←———    发送数据   ———→ 4. send()
6. close() 关闭连接                    5. close() 关闭
```

3.3 代码示例（最简单的TCP通信）

服务端：

```python
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("0.0.0.0", 8888))
server.listen(5)

client, addr = server.accept()
print(f"客户端连接：{addr}")

data = client.recv(1024)
print(f"收到：{data}")

client.close()
server.close()
```

客户端：

```python
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("127.0.0.1", 8888))
client.send(b"Hello, Server!")
client.close()
```

---

4. 粘包问题与解决方案

什么问题？

TCP是流式协议，数据像水流一样连续不断。如果你连续发送两个数据包：

```
发送：  [数据包A] [数据包B]
收到：  [数据包A的一部分 + 数据包B的一部分]  ← 粘在一起了
```

我们的解决方案：自定义传输协议

每个消息前面加**固定长度的头部**，头部包含消息长度，接收方先读头部再读数据：

```
# 发送方
header = pack(">I", len(data))  # 4字节的消息长度
sock.sendall(header + data)

# 接收方
header = recv_exact(sock, 4)    # 先收4字节头部
data_len = unpack(">I", header) # 解析出消息长度
data = recv_exact(sock, data_len)
```

---

5. 我们系统用到的网络知识

系统架构

```
客户端（发送方）                    服务端（接收方）
    |                                  |
    | 1. 连接 127.0.0.1:8888           |
    | ──────────────────────────→      |
    | 2. 发送文件元信息                  |
    | ──────────────────────────→      |
    | 3. 分块发送加密数据                 |
    |   [块1] → 等待ACK ←──────────     |
    |   [块2] → 等待ACK ←──────────     |
    |   ...                            |
    | 4. 接收完成确认                    |
    | ←──────────────────────────      |
```

关键代码位置

功能	文件	说明
传输协议定义	`提升题/network_protocol.py`	消息格式、打包解包
传输引擎	`提升题/network_transfer.py`	分块发送/接收、进度回调
服务端GUI	`提升题/network_server.py`	接收端图形界面
客户端GUI	`提升题/network_client.py`	发送端图形界面

下一步

学完基础后，阅读： 1. 传输协议详解 — 看我们如何设计传输协议 2. C/S架构详解 — 看服务端和客户端完整代码 # 深度学习基础

> 从零开始，适合没学过深度学习的EE学生

---

1. 什么是深度学习？

一句话理解

深度学习就是”用很多层的数学函数搭成一个巨大的网络，然后给它看大量数据，让它自己学会做某件事”。

深度学习 vs 传统编程

```
传统编程：  规则 + 数据 → 计算机 → 答案
深度学习：  答案 + 数据 → 计算机 → 规则（模型）
```

---

2. 神经元：最基本的单元

一个神经元做的事情非常简单：

```
输入(x1) ──→ 权重(w1) ─┐
输入(x2) ──→ 权重(w2) ─┼──→ Σ(加权和) → [激活函数] → 输出(y)
输入(x3) ──→ 权重(w3) ─┘          ↑
                                偏置(b)
```

公式：`y = f(w1*x1 + w2*x2 + w3*x3 + b)`

类比：权重(w)就像考试中各题的分值，偏置(b)就像及格线。

---

3. 神经网络结构

```
输入层          隐藏层1          隐藏层2          输出层
  o ──── o ──── o ──── o
```

```
○ ——— ○ ——— ○ ——— ○
○ ——— ○ ——— ○ ——— ○
```

- **输入层**：接收原始数据（图片像素）
- **隐藏层**：提取和组合特征（边缘→形状→物体）
- **输出层**：输出最终结果

---

4. 卷积神经网络（CNN）

为什么需要CNN？

一张512×512的彩色图片有786,432个像素，用全连接网络参数太多，计算量太大。

卷积操作

卷积就是”用一个小窗口扫描整张图片，提取特征”。

不同的卷积核可以探测不同的特征：有的探测水平边缘，有的探测垂直边缘，有的探测纹理。

Stable Diffusion的UNet中大量使用了卷积层。

---

5. AIGC需要用到的关键概念

5.1 生成模型

类型	做什么	举例
判别模型	这张图是猫还是狗？	分类
生成模型	画一只猫	这就是AIGC！

5.2 潜在空间（Latent Space）

通俗理解：潜在空间就是”信息的压缩表示”。

```
原始图片（512×512×3像素）
        ↓ 编码器压缩
潜在表示（64×64×4的"浓缩"信息）
```

```
        ↓ 解码器还原
生成图片（512×512×3像素）
```

这就是Stable Diffusion名字中”Latent”的含义——在压缩后的潜在空间中做扩散。

5.3 去噪（Denoising）

通俗理解：从满是噪点的图片中逐步恢复出清晰图片。

```
纯噪声 → ... → 比较清晰 → 清晰图片
Step 0       Step 25     Step 50
```

---

6. PyTorch基础

张量（Tensor）

PyTorch里的张量就是”多维数组”：

```python
import torch

t1 = torch.tensor([1, 2, 3])           # 1维
t2 = torch.tensor([[1,2], [3,4]])       # 2维（矩阵）
t4 = torch.randn(4, 3, 512, 512)        # 4维（4张RGB图片）
```

GPU加速

```python
device = "cuda" if torch.cuda.is_available() else "cpu"
t = torch.tensor([1, 2, 3]).to(device)
```

---

7. EE学生应该熟悉的类比

你已经会的（EE）	对应的深度学习概念
信号处理	张量运算
傅里叶变换	卷积
滤波器	卷积核（可学习的滤波器）

你已经会的（EE）	对应的深度学习概念
反馈控制	反向传播、梯度下降
放大器饱和	激活函数（ReLU等）
噪声	扩散模型中的噪声
SNR	噪声强度控制

下一步

学完后阅读： 1. 扩散模型原理 — AIGC核心技术 2. Stable Diffusion详解 — 看代码如何生成图像 # DES 算法详解

> 面向对象：电子信息工程专业学生，有基础 Python 知识，无密码学背景。 配套代码：`des_core.py`

---

1. DES 是什么？

DES（Data Encryption Standard，数据加密标准）是一种**对称加密算法**，诞生于 1970 年代，由 IBM 开发，后被美国国家标准局（NIST）采纳为联邦标准。

核心特点

特性	值
算法结构	Feistel 网络（费斯妥网络）
分组大小	64 位（8 字节）
有效密钥长度	56 位（实际密钥用 64 位存储，每 8 位中有 1 位校验位）
加密模式	ECB（电子密码本模式）
填充方式	PKCS7

对称加密的直观理解

想象你有一个带锁的**保险箱**（加密算法）和一把**钥匙**（密钥）：

- **加密**：把文件放进保险箱，用钥匙锁上 → 别人打不开
- **解密**：用同一把钥匙打开保险箱，取出文件

“对称”的含义：**加密和解密用的是同一把钥匙**。这就好比你家大门钥匙——出门时用它锁门，回家时用它开门，同一把。

Feistel 网络结构（简化理解）

DES 的内部结构叫做 Feistel 网络。它的工作方式可以比喻为一个”分两半处理”的流水线：

```
原始数据 (64位)
      │
      ├─── 左半 (32位) ─── 右半 (32位)
      │                      │
      │      ┌───────────────┤
      │      │ 轮函数 F      │
      │      │ (使用子密钥)  │
      │      └───────┬───────┘
      │              │
      ├─── 新右半 ←──┤
      │              │
      └─── 新左半 = 原右半
           ↑ 这就是 Feistel 的核心：左右交换 + 异或
```

关键点：每一轮只处理一半数据，通过 16 轮迭代打乱数据。这种结构的好处是**加密和解密可以用完全相同的代码**（只需要把子密钥的顺序反过来）。

---

2. 我们用的库：pycryptodome

```python
from Crypto.Cipher import DES, DES3
from Crypto.Util.Padding import pad, unpad
```

在 `des_core.py` 中，我们并**没有自己实现 DES 算法**（那需要几百行代码），而是使用了 Python 的 `pycryptodome` 库。

`pycryptodome` 是一个密码学工具包，相当于一个”密码学工具箱”，里面已经实现了 DES、AES、RSA 等各种算法。我们只需要调用它的 API 即可。

> 打个比方：你不需要自己造螺丝刀才能拧螺丝。`pycryptodome` 就是现成的工具箱，我们只需要选择合适的工具来用。

导入的含义

导入语句	含义
`from Crypto.Cipher import DES`	导入单 DES 加密器（用于密钥 ≤ 8 字节）
`from Crypto.Cipher import DES3`	导入三重 DES 加密器（用于密钥 > 8 字节）
`from Crypto.Util.Padding import pad`	导入填充函数（把数据补齐到 8 字节的整数倍）
`from Crypto.Util.Padding import unpad`	导入去填充函数（解密后去掉补齐的字节）

---

3. 密钥派生函数 `_derive_des_key()`

```
def _derive_des_key(key_str: str) -> tuple[bytes, bool]:
```

这是整个模块最关键的函数。它接受一个字符串作为密钥，根据字符串长度自动选择使用单 DES 还是三重 DES。

函数签名解析

- 参数：`key_str: str` —— 用户输入的密钥字符串，1~16个字符
- 返回值：`tuple[bytes, bool]` —— 返回两个东西：
  1. `bytes`：派生后的密钥字节序列
  2. `bool`：是否使用三重 DES（`True`=三重，`False`=单 DES）

逐行详解

```
key_bytes = key_str.encode("utf-8")
```

将字符串转换为字节序列。计算机只能处理字节（0和1），不能直接处理字符。 - 例如 `"AB"` 变成 `b"AB"`（即 `[65, 66]` 这两个字节） - 为什么用 UTF-8？因为它是通用的字符编码标准，能表示几乎所有语言的字符

```
if len(key_bytes) <= 8:
    raw_key = key_bytes.ljust(8, b"\x00")
    return raw_key, False
```

密钥 ≤ 8 个字符 → 使用单 DES：

- `len(key_bytes)` 获取字节长度（英文字符一个字母=1字节）
- 如果密钥短于 8 字节，用 `\x00`（零字节）在右边补齐到 8 字节
  - 例如 `"ABC"` → `b"ABC\x00\x00\x00\x00\x00"`（8 字节）
- 返回 `(8字节密钥, False)` —— `False` 表示不使用三重 DES

```
else:
    k1 = key_bytes[:8].ljust(8, b"\x00")
    rest = key_bytes[8:]
```

密钥 > 8 个字符 → 使用三重 DES：

- `k1`：取前 8 个字符，如果不足 8 字节就补 `\x00`
- `rest`：从第 9 个字符开始到结尾

```
    if len(rest) >= 8:
        k2 = rest[:8]
    else:
        k2 = rest.ljust(8, b"\x00")
    return k1 + k2, True
```

- `k2`：从剩余部分再取 8 字节（如果不足就补 `\x00`）
- `k1 + k2`：拼接成 16 字节，这就是三重 DES EDE2 的密钥
- 返回 `(16字节密钥, True)` —— `True` 表示使用三重 DES

> **为什么这样做？** 实际项目中 DES 的 56 位密钥太短，容易被暴力破解。为了兼容旧系统同时提升安全性，当用户输入长密钥时自动升级到三重 DES。

总结：密钥规则

密钥长度	使用算法	密钥材料	安全性
1~8 字符	单 DES	8 字节（补齐后）	低（可被暴力破解）
9~16 字符	三重 DES EDE2	16 字节（K1 || K2）	较高

---

4. 加密函数 `des_encrypt()`

```
def des_encrypt(data: bytes, key_str: str) -> bytes:
    """使用 DES 加密数据。"""
    raw_key, use_triple = _derive_des_key(key_str)
    cipher = DES3.new(raw_key, DES3.MODE_ECB) if use_triple else DES.new(raw_key,
        DES.MODE_ECB)
    return cipher.encrypt(pad(data, DES.block_size))
```

逐行解释

第 1 行：调用密钥派生函数，获取 8 字节或 16 字节的密钥，以及是否使用三重 DES 的标记。

```
raw_key, use_triple = _derive_des_key(key_str)
```

第 2 行：创建加密器对象。

```
cipher = DES3.new(raw_key, DES3.MODE_ECB) if use_triple else DES.new(raw_key,
        DES.MODE_ECB)
```

这是一个**条件表达式**（三元运算符），等价于：

```
if use_triple:
    cipher = DES3.new(raw_key, DES3.MODE_ECB)  # 三重 DES
else:
    cipher = DES.new(raw_key, DES.MODE_ECB)     # 单 DES
```

- `DES3.new()` 或 `DES.new()`：创建加密器
- `DES3.MODE_ECB` 或 `DES.MODE_ECB`：指定 ECB 模式

第 3 行：填充数据然后加密。

```
return cipher.encrypt(pad(data, DES.block_size))
```

- `DES.block_size` 的值是 8（64 位 = 8 字节）
- `pad(data, 8)`：将数据填充到 8 字节的整数倍
- `cipher.encrypt(...)`：执行实际的 DES 加密

---

5. 解密函数 `des_decrypt()`

```
def des_decrypt(data: bytes, key_str: str) -> bytes:
    """使用 DES 解密数据。"""
    raw_key, use_triple = _derive_des_key(key_str)
    cipher = DES3.new(raw_key, DES3.MODE_ECB) if use_triple else DES.new(raw_key,
        DES.MODE_ECB)
    return unpad(cipher.decrypt(data), DES.block_size)
```

与加密函数的对称关系

解密过程和加密过程几乎完全对称：

步骤	加密	解密
1. 派生密钥	`_derive_des_key(key_str)`	`_derive_des_key(key_str)`
2. 创建加密器	`DES.new(raw_key, MODE_ECB)`	`DES.new(raw_key, MODE_ECB)`
3. 核心操作	`.encrypt()`	`.decrypt()`
4. 填充处理	`pad()` 先填充	先解密，然后 `unpad()` 去填充

关键区别只有两个： - 调用的是 `.decrypt()` 而不是 `.encrypt()` - 先解密，再去填充（`unpad`），和加密时先填充再加密正好相反

unpad 的工作原理

`unpad()` 去掉 PKCS7 填充。PKCS7 的规则是：**缺几个字节就补几个值为几的字节。**

举例：假设待加密数据是 10 字节，DES 块大小是 8 字节。

```
原始数据: [AA BB CC DD EE FF GG HH II JJ]        (10 字节)
DES 需要 8 的整数倍 → 需要补齐到 16 字节

填充 6 个值为 0x06 的字节:
填充后:   [AA BB CC DD EE FF GG HH II JJ 06 06 06 06 06 06]   (16 字节)

解密后:   [AA BB CC DD EE FF GG HH II JJ 06 06 06 06 06 06]
unpad 看到最后一个字节是 0x06，去掉末尾 6 个字节:
还原:     [AA BB CC DD EE FF GG HH II JJ]                      (10 字节)
```

如果数据恰好是 8 的整数倍（如 16 字节），PKCS7 会**额外填充 8 个 0x08 字节**，确保解密时能明确知道哪里是填充。

---

6. ECB 模式详解

```
明文块1 → DES加密 → 密文块1
明文块2 → DES加密 → 密文块2
明文块3 → DES加密 → 密文块3
```

ECB（Electronic Codebook，电子密码本）是最简单的加密模式：

- 将数据分成 8 字节一块
- 每块**独立**用相同的 DES 密钥加密
- 各个块之间没有关联

ECB 的缺点（为什么现代系统不用它）

ECB 最大的问题是：**相同的明文块会产生相同的密文块**。

```
场景：加密一张黑白图片
原始图片:  ████░░░░████░░░░
           (黑色块 = 0x00 重复出现)

ECB 加密后:  XXXXXXXX░░░░XXXXXXXX░░░░
             (相同的黑色块加密出相同的密文块)
             图案轮廓仍然可见！
```

这种”轮廓泄露”在真实场景中很危险。更安全的模式（如 CBC）会让每个块依赖前一个块，使得相同的明文产生不同的密文。但在我们的实验场景中，ECB 足够用来理解 DES 的基本原理。

---

7. 完整示例

```python
# ===== 加密示例 =====
from des_core import des_encrypt, des_decrypt

# 准备数据
plaintext = b"Hello, DES World! 12345"

# 短密钥 → 单 DES
key_short = "1234567"
ciphertext = des_encrypt(plaintext, key_short)
decrypted = des_decrypt(ciphertext, key_short)
print(decrypted == plaintext)  # True

# 长密钥 → 三重 DES
key_long = "ThisIsMyKey16Ch"
ciphertext = des_encrypt(plaintext, key_long)
decrypted = des_decrypt(ciphertext, key_long)
print(decrypted == plaintext)  # True

# 错误示范：解密密钥和加密密钥不同
wrong = des_decrypt(ciphertext, "WrongKey!!!!!")
# 结果：要么抛出异常，要么得到乱码
```

注意：**加密和解密必须使用完全相同的密钥字符串**，否则解密会失败或得到乱码。

---

8. 安全性分析

单 DES 的弱点

问题	说明
密钥太短	56 位有效密钥，现代计算机可在数小时内暴力破解
块大小太小	64 位块，生日攻击下约 2^32 块后出现碰撞
ECB 模式	前文已说明，相同明文块暴露模式

1999 年，一个叫 **EFF DES 破解器**（Deep Crack）的专用机器在 56 小时内破解了一个 DES 密钥。如今，使用 AWS 云服务甚至可以在 1 小时内破解。

三重 DES 的优势

三重 DES（3DES）对每个数据块执行三次 DES 操作：

```
加密：明文 → DES加密(K1) → DES解密(K2) → DES加密(K1) → 密文
解密：密文 → DES解密(K1) → DES加密(K2) → DES解密(K1) → 明文
```

模式	密钥长度	有效安全强度	破解时间（估算）
单 DES	56 位	~56 位	数小时~数天
3DES EDE2	112 位	~80 位	数百万年

> EDE2 是 “Encrypt-Decrypt-Encrypt with 2 keys” 的缩写，用两个不同的密钥 K1 和 K2。先加密、再解密、再加密——解密步骤的存在是为了兼容单 DES（当 K1=K2 时，3DES 退化为单 DES）。

为什么还要学 DES？

尽管 DES 已经不安全了，但它是密码学历史上的里程碑，学习 DES 有助于理解：

1. **Feistel 网络**——后续算法（如 Blowfish、Twofish）的基础
2. **对称加密**的基本工作流程
3. **密钥管理**的基本概念
4. 为理解 AES（高级加密标准，DES 的继任者）打下基础

---

9. 进阶思考

Q：为什么 ECB 模式使用 PKCS7 填充而不用其他方式？

A：PKCS7 是最通用的填充标准，多种加密算法都支持。它的优点是填充内容明确包含了长度信息（填充的字节数），解密时能够准确去除。

Q：`_derive_des_key` 函数名前的下划线是什么意思？

A：Python 的命名约定——**单下划线开头表示”内部使用”的函数**。它告诉使用者：“这个函数是模块内部实现细节，外部代码不应该直接调用它。”

Q：在真实项目中应该用 DES 吗？

A：**绝对不要**。应该使用 AES（更安全、更高效）。但在学习密码学原理时，DES 因为结构相对简单，是比 AES 更好的教学工具。我们的项目使用 DES 是为了教学目的，真实系统请使用 AES。

10. 代码总览（含注释）

```python
from Crypto.Cipher import DES, DES3      # 导入 DES 和 3DES 加密器
from Crypto.Util.Padding import pad, unpad  # 导入填充/去填充工具

def _derive_des_key(key_str: str) -> tuple[bytes, bool]:
    """从字符串派生 DES 密钥"""
    key_bytes = key_str.encode("utf-8")     # 字符串 → 字节
    if len(key_bytes) <= 8:                 # 短密钥 → 单 DES
        raw_key = key_bytes.ljust(8, b"\x00")  # 补齐到 8 字节
        return raw_key, False                # (密钥, 非三重)
    else:                                    # 长密钥 → 三重 DES
        k1 = key_bytes[:8].ljust(8, b"\x00")  # 前 8 字节
        rest = key_bytes[8:]                 # 剩余字节
        if len(rest) >= 8:
            k2 = rest[:8]                    # 再取 8 字节
        else:
            k2 = rest.ljust(8, b"\x00")      # 不足则补齐
        return k1 + k2, True                 # (16字节密钥, 三重)

def des_encrypt(data: bytes, key_str: str) -> bytes:
    """DES 加密"""
    raw_key, use_triple = _derive_des_key(key_str)  # 派生密钥
    # 选择合适的加密器
    cipher = DES3.new(raw_key, DES3.MODE_ECB) if use_triple \
        else DES.new(raw_key, DES.MODE_ECB)
    return cipher.encrypt(pad(data, DES.block_size))  # 填充 → 加密

def des_decrypt(data: bytes, key_str: str) -> bytes:
    """DES 解密"""
    raw_key, use_triple = _derive_des_key(key_str)  # 派生密钥（相同方式）
    cipher = DES3.new(raw_key, DES3.MODE_ECB) if use_triple \
        else DES.new(raw_key, DES.MODE_ECB)
    return unpad(cipher.decrypt(data), DES.block_size)  # 解密 → 去填充
```

RSA 算法详解

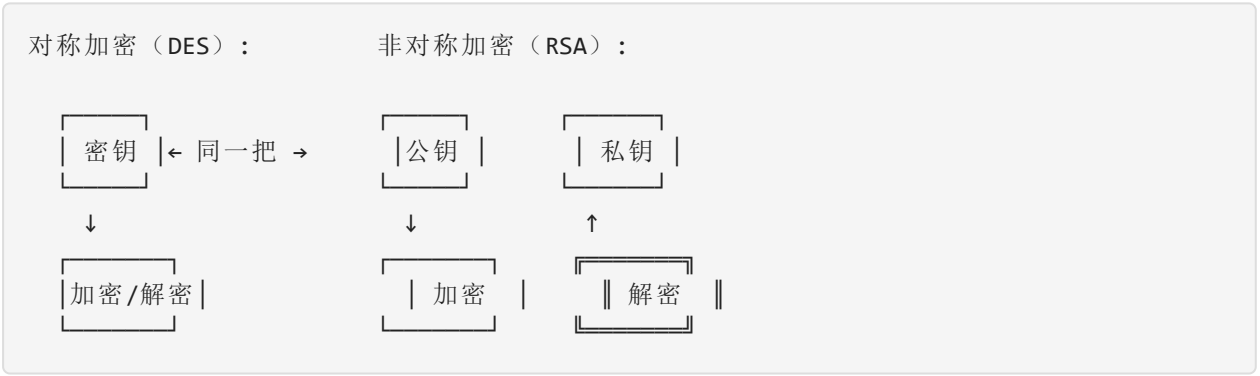
> 面向对象：电子信息工程专业学生，有基础 Python 知识，无密码学背景。 配套代码：`rsa_core.py`

---

1. RSA 是什么？

RSA 是目前最广泛使用的**非对称加密算法**，由 Rivest、Shamir、Adleman 三位密码学家在 1977 年提出（算法名取自三人姓氏首字母）。

对称 vs 非对称：关键区别



特性	对称加密（DES）	非对称加密（RSA）
密钥数量	1 个共享密钥	2 个（公钥 + 私钥）
密钥关系	加密=解密用同一把	公钥加密 → 私钥解密
速度	**快**（适合加密大量数据）	**慢**（适合加密少量数据）
典型用途	加密文件内容	加密密钥、数字签名

用信封来类比 RSA

想象你要给朋友寄一封秘密信件：

1. 你用一个**公开的密码锁**（公钥）锁上信件 → 任何人都可以用这个锁锁上，但打不开
2. **朋友**用他独有的**钥匙**（私钥）打开锁 → 只有他有这把钥匙

这就是 RSA 的核心思想：**公钥公开分发，私钥自己保管，永不泄露。**

---

2. RSA 背后的数学原理

2.1 核心公式

RSA 加密和解密的核心只有两个公式：

```
加密：   c = m^e mod n
解密：   m = c^d mod n
```

其中： - `m` = 明文（被加密的消息，需要先转换为数字） - `c` = 密文（加密后的结果） - `e` = 公钥指数（公开，常用值 65537） - `d` = 私钥指数（保密） - `n` = 模数（公开，由两个大素数相乘得到）

> **中国有句老话：“教会徒弟，加密师傅；解密还得靠私钥。”** 实际上 RSA 的安全性正依赖于：知道 `n` 和 `e`（公钥）很容易计算 `c = m^e mod n`，但**不知道 `d`（私钥）就无法从 `c` 还原出 `m`**。

2.2 为什么 RSA 是安全的？

```
公钥 = (n, e)      ← 公开
私钥 = (n, d)      ← 保密

攻击者知道 n 和 e，想要求出 d。
而 d 的计算需要知道 n 的素因数分解：
    n = p × q   (p 和 q 是两个大素数)

问题：从 n 反推出 p 和 q 极其困难。
```

这就是 **大整数分解问题**：

- 给你一个 617 位的数字（2048 位 RSA 的 n）：**请在一年内找出它的两个素因子**
- 用目前最快的算法和最好的超级计算机：**需要数百万年**

> 但量子计算机（Shor 算法）可以高效解决这个问题——这是后量子密码学要应对的挑战。

2.3 欧拉定理（略讲）

RSA 的数学正确性依赖于欧拉定理：

```
对于互质的 a 和 n：a^ϕ(n) ≡ 1 (mod n)
```

在 RSA 中选取 `e` 和 `d` 使得：

```
e × d ≡ 1 (mod ϕ(n))
```

这样：

```
(m^e)^d = m^(e×d) = m^(k×ϕ(n)+1) = (m^ϕ(n))^k × m ≡ 1^k × m = m (mod n)
```

你不需要完全理解这些数学细节（EE 专业后续的《信息论与编码》课程会深入），只需要知道： 1. RSA 选择两个大素数 `p`、`q` 计算 `n = p×q` 2. 选择公钥 `e`，计算私钥 `d` 使得 `e × d ≡ 1 (mod ϕ(n))` 3. 知道 `n` 和 `e` 无法算出 `d`，因为需要分解 `n`

---

3. `generate_keypair()`：生成密钥对

```python
from Crypto.PublicKey import RSA

def generate_keypair(bits: int = 2048) -> RSA.RsaKey:
    """生成 RSA 密钥对。"""
    return RSA.generate(bits)
```

逐行解释

导入语句：

```python
from Crypto.PublicKey import RSA
```

从 `pycryptodome` 库中导入 RSA 模块。这个模块封装了： - 大素数生成（找到两个合适的大素数 `p` 和 `q`） - 公/私钥计算（计算 `n`、`e`、`d`） - 密钥格式转换

函数定义：

```python
def generate_keypair(bits: int = 2048) -> RSA.RsaKey:
```

- `bits`：密钥长度，默认 2048 位。可以是 1024、2048 或 4096
- `RSA.RsaKey`：pycryptodome 中表示 RSA 密钥的对象类型

> **密钥长度越大越安全**，但加解密速度也越慢： - 1024 位：已不再安全，不应在新系统中使用 - 2048 位：当前推荐的最低标准（我们的默认值） - 4096 位：更高安全性，但速度慢约 4~8 倍

函数体：

```
return RSA.generate(bits)
```

`RSA.generate(bits)` 内部做了以下工作（简化描述）：

```
1. 随机生成两个大素数 p 和 q（各约 bits/2 位）
2. 计算 n = p × q
3. 计算 φ(n) = (p-1) × (q-1)
4. 选择公钥指数 e = 65537（常用值）
5. 计算私钥指数 d = e^(-1) mod φ(n)（扩展欧几里得算法）
6. 返回包含 (n, e, d, p, q, ...) 的密钥对象
```

所有这些步骤由 `pycryptodome` 在底层完成，我们只需一行代码即可。

---

4. `rsa_encrypt()` 和 `rsa_decrypt()`：加密与解密

```
from Crypto.Cipher import PKCS1_OAEP

def rsa_encrypt(data: bytes, public_key: RSA.RsaKey) -> bytes:
    """使用 RSA 公钥加密数据。"""
    return PKCS1_OAEP.new(public_key).encrypt(data)

def rsa_decrypt(data: bytes, private_key: RSA.RsaKey) -> bytes:
    """使用 RSA 私钥解密数据。"""
    return PKCS1_OAEP.new(private_key).decrypt(data)
```

4.1 为什么需要 OAEP 填充？

直接使用 RSA 数学公式 `c = m^e mod n` 加密有严重的安全问题：

问题 1：确定性加密

```
公钥 (n=3233, e=17)，加密 "YES" 和 "NO"：
加密 "YES" (m=1):  c = 1^17 mod 3233 = 1
加密 "NO"  (m=2):  c = 2^17 mod 3233 = 248

攻击者看到密文 1，立刻知道明文是 "YES"。
```

OAEP（Optimal Asymmetric Encryption Padding，**最优非对称加密填充**） 在加密前对明文做以下处理：

```
原始明文: [数据]

          ┌──────┴──────┐
          ↓             ↓
OAEP:  添加随机数 → 混合 → 填充
          ↑
          每次加密用的随机数都不同
```

效果：**同样的明文每次加密得到不同的密文**，攻击者无法通过对比密文来猜测内容。

问题 2：明文长度限制

```
RSA 2048 位密钥:
  - 最大加密字节数 ≈ 2048/8 - OAEP开销 = 256 - 42 = 214 字节

超出 214 字节 → 必须分块加密（速度极慢）
```

这就是混合加密（文件 3）中使用 RSA 加密 DES 密钥（仅 ~16 字节）而不是直接加密文件的原因。

4.2 逐行解释加密函数

```python
def rsa_encrypt(data: bytes, public_key: RSA.RsaKey) -> bytes:
```

- `data`：要加密的数据（字节类型）
- `public_key`：RSA 公钥对象

```python
    return PKCS1_OAEP.new(public_key).encrypt(data)
```

1. `PKCS1_OAEP.new(public_key)`：使用公钥创建一个 OAEP 填充的加密器
2. `.encrypt(data)`：执行 OAEP 填充 + RSA 加密

3. 返回密文字节序列

4.3 逐行解释解密函数

```
def rsa_decrypt(data: bytes, private_key: RSA.RsaKey) -> bytes:
```

- `data`：要解密的密文（字节类型）
- `private_key`：RSA 私钥对象

```
    return PKCS1_OAEP.new(private_key).decrypt(data)
```

1. `PKCS1_OAEP.new(private_key)`：使用**私钥**创建 OAEP 解密器
2. `.decrypt(data)`：执行 RSA 解密 + 去除 OAEP 填充
3. 返回原始明文字节序列

> **关键**：加密用公钥，解密用私钥。如果用错会报错。

---

5. PEM 密钥格式

5.1 PEM 文件长什么样？

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA0g5FtxwJz1Ff3Vl7wQ8R1lPqL9z0gLkGq2sTzYHJnB8V
x7v4R6H3K9mN2oP8zU5wLrXcVb7sY9R0fI4jK2lMnBvCsDqFgHyJkNmZoP
...（中间还有几十行 BASE64 编码的内容）...
-----END RSA PRIVATE KEY-----
```

PEM（Privacy Enhanced Mail，隐私增强邮件）是一种文本格式的密钥存储方式。

5.2 PEM 文件里有什么？

一个 RSA 私钥 PEM 文件中包含以下数据（以 2048 位为例）：

字段	含义	大小
`n`	模数 = p × q	256 字节

字段	含义	大小
`e`	公钥指数（通常为 65537）	3 字节
`d`	私钥指数	256 字节
`p`	第一个大素数	128 字节
`q`	第二个大素数	128 字节
`dp`	d mod (p-1)	128 字节
`dq`	d mod (q-1)	128 字节
`qinv`	q^(-1) mod p	128 字节

私钥包含**所有**信息，包括 `p` 和 `q`——这就是为什么私钥必须严格保密。

公钥 PEM 只包含 `(n, e)`，没有 `p`、`q`、`d`。

5.3 公钥和私钥 PEM 的区别

```
-----BEGIN RSA PRIVATE KEY-----     ← 私钥
MIIEpAIBAAKCAQEA...                 ← BASE64 编码的完整密钥数据
-----END RSA PRIVATE KEY-----

-----BEGIN PUBLIC KEY-----          ← 公钥（注意是 PUBLIC，没有 RSA）
MIIBIjANBgkqhkiG9w0BAQEFAAOC...  ← BASE64 编码的 (n, e)
-----END PUBLIC KEY-----
```

> 为什么公钥的头部是 `BEGIN PUBLIC KEY` 而不是 `BEGIN RSA PUBLIC KEY`？因为 pycryptodome 默认使用 X.509 SubjectPublicKeyInfo 格式，这是一种与算法无关的公钥容器格式。

---

6. 密钥导入导出函数

6.1 公钥导出：`public_key_to_pem()`

```python
def public_key_to_pem(key: RSA.RsaKey) -> str:
    """将公钥导出为 PEM 格式字符串。"""
    return key.publickey().export_key(format="PEM").decode()
```

逐行解释：

- `key.publickey()`：从完整密钥对象中提取**公钥部分**（只包含 `n` 和 `e`）
- `.export_key(format="PEM")`：格式化为 PEM 文本（返回 `bytes` 类型）
- `.decode()`：将 `bytes` 解码为 Python 字符串（方便写入文件）

6.2 私钥导出：`private_key_to_pem()`

```
def private_key_to_pem(key: RSA.RsaKey) -> str:
    """将私钥导出为 PEM 格式字符串。"""
    return key.export_key(format="PEM").decode()
```

与公钥导出的关键区别：**不需要调用 `.publickey()`**，直接导出完整私钥。

6.3 公钥导入：`public_key_from_pem()`

```
def public_key_from_pem(pem_str: str) -> RSA.RsaKey:
    """从 PEM 字符串导入公钥。"""
    return RSA.import_key(pem_str.encode())
```

- `pem_str.encode()`：将字符串编码为字节（`import_key` 需要 `bytes` 类型）
- `RSA.import_key(...)`：解析 PEM 格式，还原 `RSA.RsaKey` 对象

6.4 私钥导入：`private_key_from_pem()`

```
def private_key_from_pem(pem_str: str) -> RSA.RsaKey:
    """从 PEM 字符串导入私钥。"""
    return RSA.import_key(pem_str.encode())
```

和公钥导入完全相同！`RSA.import_key()` 会自动判断 PEM 内容中是公钥还是私钥，并返回对应的 `RSA.RsaKey` 对象： - 如果 PEM 包含 `(n, e, d, p, q, ...)` → 返回完整私钥 - 如果 PEM 只包含 `(n, e)` → 返回公钥（调用 `.decrypt()` 会报错）

---

7. 完整示例

```
from rsa_core import (
    generate_keypair,
    rsa_encrypt,
    rsa_decrypt,
```

```
    public_key_to_pem,
    private_key_to_pem,
    public_key_from_pem,
    private_key_from_pem,
)

# ===== 1. 生成密钥对 =====
key = generate_keypair(2048)
print(f"密钥位数: {key.size_in_bits()}")  # 输出: 2048

# ===== 2. 加密/解密 =====
message = b"RSA is awesome!"
ciphertext = rsa_encrypt(message, key)        # 用公钥加密
plaintext = rsa_decrypt(ciphertext, key)      # 用私钥解密
print(plaintext == message)                   # True

# ===== 3. 导出/导入密钥 =====
pub_pem = public_key_to_pem(key)     # 公钥 → PEM 字符串
priv_pem = private_key_to_pem(key)   # 私钥 → PEM 字符串

# 保存到文件
with open("public.pem", "w") as f:
    f.write(pub_pem)
with open("private.pem", "w") as f:
    f.write(priv_pem)

# 从文件导入
with open("public.pem", "r") as f:
    imported_pub = public_key_from_pem(f.read())
with open("private.pem", "r") as f:
    imported_priv = private_key_from_pem(f.read())
```

跨机器通信示例

```
# === 发送方（Alice）===
alice_key = generate_keypair(2048)

# Bob 把公钥发给 Alice
bob_pub_pem = """-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA...
-----END PUBLIC KEY-----"""
bob_pub = public_key_from_pem(bob_pub_pem)

# Alice 用 Bob 的公钥加密消息
msg = b"Hi Bob, this is a secret message!"
encrypted = rsa_encrypt(msg, bob_pub)

# === 接收方（Bob）===
# Bob 用自己的私钥解密
decrypted = rsa_decrypt(encrypted, alice_key)  # 错误！应该用 bob 的私钥
```

```
decrypted = rsa_decrypt(encrypted, bob_priv)   # 正确
print(decrypted.decode())  # "Hi Bob, this is a secret message!"
```

---

8. 安全性分析

RSA 的安全基础

攻击方式	攻击难度	说明
分解 n	**极难**（指数级）	2048 位的 n 用经典计算机需要百万年
选择密文攻击	OAEP 阻止	没有 OAEP 时攻击者可操纵密文
时序攻击	算法恒定时间	pycryptodome 有防护
量子攻击	**有效**	Shor 算法可分解大整数

密钥长度的选择权衡

密钥长度	安全级别	加密速度	使用场景
1024 位	❌ 不安全	快	不再使用
2048 位	✅ 安全	中等	**通用推荐**
4096 位	✅ 很安全	慢	高安全场景

> 2048 位密钥加密约可加密 **214 字节**数据，4096 位可加密约 **446 字节**。这就是为什么 RSA 不能直接加密大文件。

使用 RSA 的注意事项

1. **永远使用 OAEP 填充**（不要使用 PKCS1_v1.5，它已被破解）
2. **私钥文件必须严格保护**（文件权限设为 600 或 400）
3. **不要用 RSA 加密大文件**（速度慢且有长度限制）
4. **定期更换密钥对**

---

9. 总结

函数	输入	输出	作用
`generate_keypair(bits)`	密钥长度（位）	`RSA.RsaKey` 对象	生成密钥对
`rsa_encrypt(data, pub_key)`	明文 + 公钥	密文字节	加密小数据
`rsa_decrypt(data, priv_key)`	密文 + 私钥	明文字节	解密数据
`public_key_to_pem(key)`	密钥对象	PEM 字符串	导出公钥
`private_key_to_pem(key)`	密钥对象	PEM 字符串	导出私钥
`public_key_from_pem(pem)`	PEM 字符串	`RSA.RsaKey` 对象	导入公钥
`private_key_from_pem(pem)`	PEM 字符串	`RSA.RsaKey` 对象	导入私钥

混合加密系统详解

> 面向对象：电子信息工程专业学生，有基础 Python 知识，无密码学背景。 配套代码：`hybrid_crypto.py`、`gui.py`

---

1. 为什么要混合加密？

两种加密算法的优缺点

算法	类型	优点	缺点
DES	对称加密	**速度快**，适合加密大文件	密钥分发困难（怎么安全地把密钥给对方？）
RSA	非对称加密	密钥分发方便（公钥公开）	**速度慢**，只能加密少量数据（~214 字节）

混合加密的直觉

```
场景：你要快递一个装满珠宝的大箱子（文件）给朋友。

方案一（只有 DES）：你用密码锁锁上箱子，但怎么把密码告诉朋友？
                    万一密码在路上被偷看就完了。
                    → 密钥分发问题

方案二（只有 RSA）：你把珠宝一个个单独用密码锁锁上。
                    每个珠宝都要单独加锁，累死了。
                    → RSA 太慢，加密大数据不现实

方案三（混合加密）：聪明办法！
                    1. 用一把临时密码锁（DES 密钥）锁上大箱子
                    2. 把这把临时密码锁的密码用朋友的公开密码锁锁上
                    3. 把"加了密的密码"和箱子一起寄过去

                    朋友收到后：
                    1. 用他的钥匙（RSA 私钥）打开密码信封，得到临时密码
                    2. 用临时密码打开大箱子
```

这就是 **混合加密**（Hybrid Encryption）的核心思想：

2. 自定义文件格式

混合加密的结果是一个自定义的二进制文件格式：

段	大小	内容
头部	4 字节	RSA 加密的 DES 密钥的长度（大端序无符号整数）
密钥区	N 字节（由头部的值决定）	RSA 加密后的 DES 密钥
数据区	剩余所有字节	DES 加密后的文件内容

为什么需要头部记录密钥长度？

因为 RSA 加密后的密文长度取决于密钥长度：

RSA 密钥长度	RSA 加密输出长度
1024 位	128 字节
2048 位	256 字节
4096 位	512 字节

解密时，程序需要知道”从第几个字节开始是 DES 密文”，所以必须在前 4 个字节记录 RSA 密文的长度。

> 用一个比喻来说明：这就像快递单上的”内件数量”字段——快递员（解密程序）得先知道有几个包裹（密钥长度），才能正确清点货物（提取密钥和数据）。

3. `encrypt_file()`：文件加密

```
def encrypt_file(input_path: str, output_path: str,
                 des_key_str: str, rsa_public_key: RSA.RsaKey) -> None:
    """混合加密文件。"""
    with open(input_path, "rb") as f:
        plain_data = f.read()

    cipher_data = des_encrypt(plain_data, des_key_str)
    encrypted_des_key = rsa_encrypt(des_key_str.encode("utf-8"), rsa_public_key)

    with open(output_path, "wb") as f:
        f.write(struct.pack(">I", len(encrypted_des_key)))
        f.write(encrypted_des_key)
        f.write(cipher_data)
```

逐行详解

```
def encrypt_file(input_path: str, output_path: str,
                 des_key_str: str, rsa_public_key: RSA.RsaKey) -> None:
```

函数接收 4 个参数：

参数	类型	含义
`input_path`	`str`	输入文件路径（明文文件）
`output_path`	`str`	输出文件路径（加密后的文件）
`des_key_str`	`str`	DES 密钥字符串（1~16 字符）
`rsa_public_key`	`RSA.RsaKey`	RSA 公钥对象

返回 `None`：结果直接写入文件，不返回数据。

第一步：读取明文文件

```
with open(input_path, "rb") as f:
    plain_data = f.read()
```

- `"rb"` 模式：以二进制只读方式打开文件

- `f.read()`：读取文件全部内容到内存
- 对于大文件（超过 1GB），这里会消耗大量内存——但我们的教学场景不考虑这个问题

> 二进制模式 `"rb"` vs 文本模式 `"r"`： - 文本模式会进行换行符转换（`\n` ↔ `\r\n`），可能损坏数据 - 加密必须使用二进制模式，确保每个字节被原样处理

第二步：DES 加密文件内容

```
cipher_data = des_encrypt(plain_data, des_key_str)
```

- 使用用户提供的 DES 密钥字符串加密文件全部内容
- 返回加密后的字节序列
- 内部处理了填充、ECB 模式等细节（详见文件 1）

第三步：RSA 加密 DES 密钥

```
encrypted_des_key = rsa_encrypt(des_key_str.encode("utf-8"), rsa_public_key)
```

- `des_key_str.encode("utf-8")`：将 DES 密钥字符串（如 “MyKey123”）转换为字节序列
- `rsa_encrypt(...)`：使用 RSA 公钥加密 DES 密钥
- 为什么加密 DES 密钥而不是直接加密文件？因为 RSA 一次只能加密约 214 字节，而 DES 密钥只有 16 字节，正好适合 RSA 加密

第四步：打包写入文件

```
with open(output_path, "wb") as f:
    f.write(struct.pack(">I", len(encrypted_des_key)))
    f.write(encrypted_des_key)
    f.write(cipher_data)
```

- `"wb"` 模式：以二进制写入方式打开文件
- `struct.pack(">I", len(encrypted_des_key))`：将密钥长度打包为 4 字节

> `struct.pack(">I", ...)` 详解： - `>`：大端序（Big-Endian），最高有效字节在前。网络传输中常用大端序 - `I`：无符号 32 位整数（unsigned int），占用 4 字节 - 例：值 256 打包为 `\x00\x00\x01\x00`（大端序）而不是 `\x00\x01\x00\x00`（小端序）

写入顺序依次为：`[4字节头部]` → `[N字节RSA密文]` → `[剩余所有DES密文]`

---

4. `decrypt_file()`：文件解密

```
def decrypt_file(input_path: str, output_path: str,
                 des_key_str: str, rsa_private_key: RSA.RsaKey) -> None:
    """混合解密文件。"""
    with open(input_path, "rb") as f:
        header = f.read(4)
        if len(header) < 4:
            raise ValueError("文件格式错误：无法读取密钥长度头。")
        key_len = struct.unpack(">I", header)[0]

        encrypted_des_key = f.read(key_len)
        if len(encrypted_des_key) < key_len:
            raise ValueError("文件格式错误：密钥数据不完整。")

        cipher_data = f.read()

    decrypted_des_key_bytes = rsa_decrypt(encrypted_des_key, rsa_private_key)
    actual_des_key = decrypted_des_key_bytes.decode("utf-8")

    plain_data = des_decrypt(cipher_data, actual_des_key)

    with open(output_path, "wb") as f:
        f.write(plain_data)
```

逐行详解

第一步：解析文件头部

```
with open(input_path, "rb") as f:
    header = f.read(4)
```

以二进制只读方式打开加密文件，读取前 4 字节（头部）。

```
    if len(header) < 4:
```

```
        raise ValueError("文件格式错误：无法读取密钥长度头。")
```

错误检查：如果文件小于 4 字节，说明不是合法的加密文件格式，抛出异常。

```
    key_len = struct.unpack(">I", header)[0]
```

解析 4 字节头部为整数，得到 RSA 密文的长度。

> `struct.unpack(">I", header)` 返回一个元组（因为可以一次解包多个字段），`[0]` 提取第一个（也是唯一一个）字段。

第二步：读取密钥和数据

```
    encrypted_des_key = f.read(key_len)
    if len(encrypted_des_key) < key_len:
        raise ValueError("文件格式错误：密钥数据不完整。")
```

- 根据头部指定的长度，读取 RSA 密文
- 检查实际读取的字节数是否等于预期长度（防止文件截断）
- `f.read(key_len)` 从当前文件位置（头部之后）读取指定字节数

```
    cipher_data = f.read()
```

读取**剩余所有字节**——这些是 DES 加密的文件内容。

至此，文件已被解析为三个部分：

```
┌──────────┬──────────────────────┬──────────────────┐
│ header   │ encrypted_des_key    │ cipher_data      │
│ (4字节)  │ (key_len 字节)       │ (剩余所有字节)   │
└──────────┴──────────────────────┴──────────────────┘
```

第三步：RSA 解密 DES 密钥

```
decrypted_des_key_bytes = rsa_decrypt(encrypted_des_key, rsa_private_key)
actual_des_key = decrypted_des_key_bytes.decode("utf-8")
```

- **关键**：这里使用的是 **RSA 私钥**（不是公钥！）
- `rsa_decrypt()` 解密出 DES 密钥的字节形式
- `.decode("utf-8")` 将字节还原为字符串——这就是原始 DES 密钥

> 为什么传输 DES 密钥时不直接传输明文？因为如果明文传输，攻击者截获文件后就能直接用 DES 密钥解密数据。RSA 加密 DES 密钥是整个系统的安全核心。

第四步：DES 解密文件内容

```
plain_data = des_decrypt(cipher_data, actual_des_key)
```

使用从 RSA 解密得到的 DES 密钥字符串，DES 解密文件内容。

第五步：写入解密结果

```
with open(output_path, "wb") as f:
    f.write(plain_data)
```

将解密后的明文写入输出文件。

---

5. `encrypt_bytes()` / `decrypt_bytes()`：内存版加解密

5.1 `encrypt_bytes()`

```
def encrypt_bytes(data: bytes, rsa_public_key: RSA.RsaKey) -> tuple[bytes, str]:
    """混合加密内存数据。"""
    des_key_str = _generate_des_key()
    cipher_data = des_encrypt(data, des_key_str)
    encrypted_des_key = rsa_encrypt(des_key_str.encode("utf-8"), rsa_public_key)

    buf = struct.pack(">I", len(encrypted_des_key))
    buf += encrypted_des_key
    buf += cipher_data
    return buf, des_key_str
```

与 `encrypt_file()` 的区别：

特性	`encrypt_file()`	`encrypt_bytes()`
输入	文件路径	内存中的字节数据
DES 密钥	用户提供	自动生成
输出	写入文件	返回字节数据
返回值	`None`	`(加密数据包, DES密钥字符串)`

DES 密钥自动生成：

```
def _generate_des_key() -> str:
    """生成随机 DES 密钥字符串（16字符，触发三重DES EDE2模式）。"""
    alphabet = string.ascii_letters + string.digits + "!@#$%^&*"
    return "".join(secrets.choice(alphabet) for _ in range(16))
```

- `string.ascii_letters`：52 个大小写字母
- `string.digits`：10 个数字
- `"!@#$%^&*"`：8 个特殊字符
- `secrets.choice(alphabet)`：从上述字符集中随机选择（`secrets` 模块提供加密安全的随机数）
- `"".join(...)`：拼接为 16 字符的字符串

为什么自动生成 16 字符？因为 16 > 8，会触发三重 DES EDE2 模式，提供更强的安全性。

5.2 `decrypt_bytes()`

```
def decrypt_bytes(data: bytes, rsa_private_key: RSA.RsaKey) -> bytes:
    """混合解密内存数据。"""
    key_len = struct.unpack(">I", data[:4])[0]
    encrypted_des_key = data[4:4 + key_len]
    cipher_data = data[4 + key_len:]

    des_key_str = rsa_decrypt(encrypted_des_key, rsa_private_key).decode("utf-8")
    return des_decrypt(cipher_data, des_key_str)
```

使用切片（slicing）从数据包中提取各段：

```
data[:4]          # 前 4 字节：密钥长度
data[4:4+key_len] # 从第 5 字节开始的 key_len 字节：RSA 密文
```

```
data[4+key_len:]  # 剩余所有字节：DES 密文
```

> 切片操作可以用”切蛋糕”来理解：
>
> ```
> 整个蛋糕（data）：
> ┌──────────────────────────────────────────┐
> │  头4字节   │  中间N字节   │      剩余      │
> │  data[:4] │  data[4:4+N] │  data[4+N:]    │
> └──────────────────────────────────────────┘
> ```

---

6. 图形界面（GUI）：整体布局

GUI 使用 Tkinter 实现——Python 标准库中的 GUI 工具包，不需要额外安装。

6.1 窗口结构

程序窗口按从上到下分为 4 个功能区域：

```
┌──────────────────────────────────────────┐
│  [1] RSA 密钥管理区域                     │
│  ┌──────────────────────────────────────┐ │
│  │ 密钥长度: [2048 ▼] [生成密钥对]       │ │
│  │ [导出公钥] [导出私钥] [导入公钥] [导入私钥]│ │
│  │ 状态：密钥已就绪（2048 位）           │ │
│  └──────────────────────────────────────┘ │
│                                           │
│  [2] DES 密钥设置区域                     │
│  ┌──────────────────────────────────────┐ │
│  │ DES 密钥: [************] (1~16个字符) │ │
│  │ 确认密钥: [************]            │ │
│  └──────────────────────────────────────┘ │
│                                           │
│  [3] 文件选择区域                         │
│  ┌──────────────────────────────────────┐ │
│  │ 输入文件: [____________________] [浏览] │ │
│  │ 输出文件: [____________________] [浏览] │ │
│  └──────────────────────────────────────┘ │
│                                           │
│  [   加   密   ]   [   解   密   ]        │
│                                           │
│  [4] 日志区域                             │
│  ┌──────────────────────────────────────┐ │
│  │ 系统就绪。请先生成或导入 RSA 密钥对。  │ │
│  │ 正在生成 2048 位 RSA 密钥对...        │ │
```

```
│  │ RSA 密钥对生成成功（2048 位）。                    │ │
│  └──────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────┘
```

6.2 区域 1：RSA 密钥管理

```python
# RSA 密钥管理 - 使用 LabelFrame（带标题的框架）
rsa_frame = ttk.LabelFrame(self.root, text="RSA 密钥管理")
rsa_frame.pack(fill=tk.X, padx=8, pady=(8, 0))

# 第一行：密钥长度选择 + 生成按钮
row0 = ttk.Frame(rsa_frame)
row0.pack(fill=tk.X, **pad)
ttk.Label(row0, text="密钥长度:").pack(side=tk.LEFT)
self._rsa_bits = ttk.Combobox(row0, values=["1024", "2048", "4096"],
                              state="readonly", width=6)
self._rsa_bits.set("2048")
self._rsa_bits.pack(side=tk.LEFT, padx=4)
ttk.Button(row0, text="生成密钥对", command=self._on_generate_keypair).pack(
    side=tk.LEFT, padx=8)

# 第二行：导入/导出按钮
row1 = ttk.Frame(rsa_frame)
row1.pack(fill=tk.X, **pad)
ttk.Button(row1, text="导出公钥", command=self._on_export_pubkey).pack(...)
ttk.Button(row1, text="导出私钥", command=self._on_export_privkey).pack(...)
ttk.Button(row1, text="导入公钥", command=self._on_import_pubkey).pack(...)
ttk.Button(row1, text="导入私钥", command=self._on_import_privkey).pack(...)

# 状态标签
self._rsa_status = ttk.Label(rsa_frame, text="", foreground="gray")
self._rsa_status.pack(anchor=tk.W, **pad)
```

控件	作用
`Combobox`（下拉框）	选择 RSA 密钥长度（1024/2048/4096）
“生成密钥对”按钮	调用 `generate_keypair()` 生成新密钥
“导出公钥”/“导出私钥”	将密钥保存为 PEM 文件
“导入公钥”/“导入私钥”	从 PEM 文件加载密钥
状态标签	显示当前密钥状态（未生成/已就绪）

状态更新函数：

```
def _update_rsa_status(self):
    if self._rsa_key is None:
        self._rsa_status.config(text="状态：密钥未生成", foreground="red")
    else:
        bits = self._rsa_key.size_in_bits()
        self._rsa_status.config(text=f"状态：密钥已就绪（{bits} 位）",
                                foreground="green")
```

- 如果没有密钥（`None`）：显示红色”密钥未生成”
- 如果有密钥：显示绿色”密钥已就绪（2048 位）”

6.3 区域 2：DES 密钥设置

```
des_frame = ttk.LabelFrame(self.root, text="DES 密钥设置")
des_frame.pack(fill=tk.X, padx=8, pady=(8, 0))

# DES 密钥输入
row = ttk.Frame(des_frame)
row.pack(fill=tk.X, **pad)
ttk.Label(row, text="DES 密钥:").pack(side=tk.LEFT)
self._des_key_var = tk.StringVar()
self._des_key_entry = ttk.Entry(row, textvariable=self._des_key_var,
                                show="*", width=24)
self._des_key_entry.pack(side=tk.LEFT, padx=4)
ttk.Label(row, text="（1～16个字符，区分大小写）").pack(side=tk.LEFT)

# 确认密钥输入
row2 = ttk.Frame(des_frame)
row2.pack(fill=tk.X, **pad)
ttk.Label(row2, text="确认密钥:").pack(side=tk.LEFT)
self._des_confirm_var = tk.StringVar()
self._des_confirm_entry = ttk.Entry(row2,
                                    textvariable=self._des_confirm_var,
                                    show="*", width=24)
self._des_confirm_entry.pack(side=tk.LEFT, padx=4)
```

控件	作用
输入框 1（DES 密钥）	输入 DES 密钥字符串
输入框 2（确认密钥）	再次输入进行确认
`show="*"`	密码模式，输入内容显示为星号
`StringVar`	Tkinter 变量，自动同步输入框内容

为什么要”确认密钥”？防止用户输入时打错字母——加密用密钥 A，解密时却输入了密钥 B，导致解密失败。

6.4 区域 3：文件选择

```
file_frame = ttk.LabelFrame(self.root, text="文件选择")
file_frame.pack(fill=tk.X, padx=8, pady=(8, 0))

for label, attr in [("输入文件:", "infile"), ("输出文件:", "outfile")]:
    row = ttk.Frame(file_frame)
    row.pack(fill=tk.X, **pad)
    ttk.Label(row, text=label).pack(side=tk.LEFT)
    var = tk.StringVar()
    setattr(self, f"_{attr}_var", var)
    ttk.Entry(row, textvariable=var, width=50).pack(side=tk.LEFT, padx=4)
    ttk.Button(row, text="浏览...",
               command=lambda a=attr: self._on_browse(a)).pack(side=tk.LEFT)
```

这里使用了一个**循环**来创建两个几乎相同的行（输入文件和输出文件），避免代码重复：

- `setattr(self, f"_{attr}_var", var)`：动态创建实例变量 `_infile_var` 和 `_outfile_var`
- 每个变量绑定到对应的 `Entry`（输入框），自动同步文件路径

6.5 操作按钮

```
action_frame = ttk.Frame(self.root)
action_frame.pack(fill=tk.X, padx=8, pady=(8, 0))
ttk.Button(action_frame, text="加　密", command=self._on_encrypt).pack(
    side=tk.LEFT, padx=20)
ttk.Button(action_frame, text="解　密", command=self._on_decrypt).pack(
    side=tk.LEFT, padx=20)
```

两个大按钮，点击后分别调用 `_on_encrypt()` 和 `_on_decrypt()`。

6.6 日志区域

```
log_frame = ttk.LabelFrame(self.root, text="日志")
log_frame.pack(fill=tk.BOTH, expand=True, padx=8, pady=8)
self._log = scrolledtext.ScrolledText(log_frame, height=10,
                                      state=tk.DISABLED, wrap=tk.WORD)
self._log.pack(fill=tk.BOTH, expand=True, padx=4, pady=4)
```

- `ScrolledText`：带滚动条的文本框

- `state=tk.DISABLED`：初始为只读（防止用户修改日志内容）
- `wrap=tk.WORD`：按单词换行（而不是按字符）
- `fill=tk.BOTH, expand=True`：充满整个区域，窗口缩放时自动调整

日志写入函数：

```python
def _log_msg(self, msg: str):
    self._log.configure(state=tk.NORMAL)    # 临时启用编辑
    self._log.insert(tk.END, msg + "\n")    # 在末尾追加文本
    self._log.see(tk.END)                   # 自动滚动到底部
    self._log.configure(state=tk.DISABLED)  # 恢复只读
```

三步操作：启用编辑 → 插入文本 → 恢复只读。`see(END)` 确保最新的日志始终可见。

---

7. GUI 完整工作流程

加密流程

```
用户操作                          后台执行
————————                        ————————
1. 生成或导入 RSA 密钥 ————————→ _on_generate_keypair()
   （如果没有密钥）                 调用 generate_keypair(bits)

2. 输入 DES 密钥 ———————————— _des_key_var.get()
   并确认                         保存在 StringVar 中

3. 选择输入文件 ———————————— _on_browse("infile")
                                 弹出文件选择对话框

4. 选择输出文件 ———————————— _on_browse("outfile")
                                 弹出保存对话框

5. 点击"加密" ————————————→ _on_encrypt()
                                   ├─ _validate_des_key() 验证密钥
                                   │    ├─ 不为空？
                                   │    ├─ ≤16字符？
                                   │    └─ 两次输入一致？
                                   ├─ _check_rsa() 检查密钥
                                   ├─ 文件存在检查
                                   ├─ encrypt_file() 执行加密
                                   └─ 显示成功消息
```

`_on_encrypt()` 内部流程

```python
def _on_encrypt(self):
    # 1. 验证 DES 密钥
    des_key = self._validate_des_key()   # 返回验证通过的密钥，或 None
    if des_key is None:
        return

    # 2. 检查 RSA 密钥
    if not self._check_rsa():
        return

    # 3. 获取文件路径
    infile = self._infile_var.get()
    outfile = self._outfile_var.get()

    # 4. 验证输入
    if not infile or not outfile:
        messagebox.showwarning("提示", "请选择文件。")
        return
    if not os.path.isfile(infile):
        messagebox.showwarning("提示", f"输入文件不存在:\n{infile}")
        return

    # 5. 执行加密
    try:
        self._log_msg(f"正在加密: {infile}")
        encrypt_file(infile, outfile, des_key, self._rsa_key)
        in_size = os.path.getsize(infile)
        out_size = os.path.getsize(outfile)
        self._log_msg(f"加密完成 → {outfile} ({in_size} → {out_size} 字节)")
        messagebox.showinfo("完成", "文件加密成功！")
    except Exception as e:
        self._log_msg(f"加密失败: {e}")
        messagebox.showerror("错误", f"加密失败:\n{e}")
```

解密流程

解密流程与加密完全对称：

```
用户操作                              后台执行
──────────                            ──────────
1. 导入 RSA 私钥 ──────────────→ _on_import_privkey()
   （加密时用的私钥！）

2. 输入 DES 密钥 ──────────────── 必须与加密时使用的密钥相同

3. 选择加密文件作为输入文件

4. 选择解密后的输出路径
```

```
5. 点击"解密" ──────────────→ _on_decrypt()
                                  ├─ 验证同加密
                                  ├─ decrypt_file() 执行解密
                                  └─ 显示成功消息
```

`_on_decrypt()` 内部流程

```python
def _on_decrypt(self):
    # 与加密相同的验证逻辑
    des_key = self._validate_des_key()
    if des_key is None:
        return
    if not self._check_rsa():
        return

    infile = self._infile_var.get()
    outfile = self._outfile_var.get()

    # 验证输入文件
    if not infile or not outfile or not os.path.isfile(infile):
        # ... 显示警告 ...

    # 执行解密
    try:
        self._log_msg(f"正在解密: {infile}")
        decrypt_file(infile, outfile, des_key, self._rsa_key)
        self._log_msg(f"解密完成 → {outfile}")
        messagebox.showinfo("完成", "文件解密成功！")
    except Exception as e:
        self._log_msg(f"解密失败: {e}")
        messagebox.showerror("错误", f"解密失败:\n{e}")
```

---

8. 程序入口

```python
def main():
    root = tk.Tk()    # 创建主窗口
    App(root)          # 创建 App 实例（构建 UI）
    root.mainloop()    # 进入事件循环（等待用户操作）

if __name__ == "__main__":
    main()
```

- `tk.Tk()`：创建 Tkinter 主窗口
- `App(root)`：实例化我们的 `App` 类，在窗口中构建所有 UI 组件

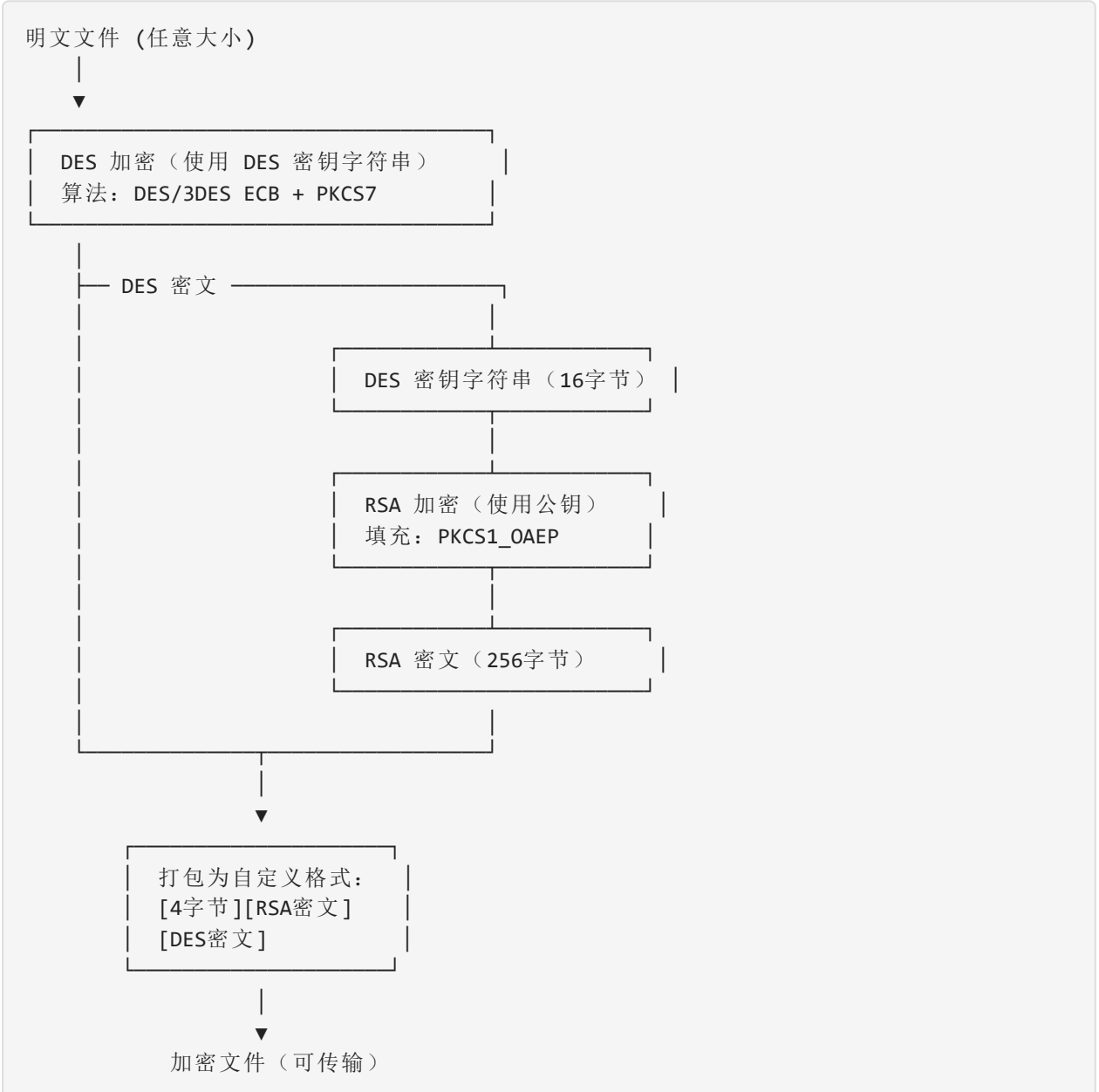
- `root.mainloop()`：启动 Tkinter 事件循环——程序在此处"阻塞"，等待用户点击按钮、输入文本等操作

> `if __name__ == "__main__"` 的作用：当直接运行 `gui.py` 时，执行 `main()`；当其他文件 `import gui` 时，不自动执行。

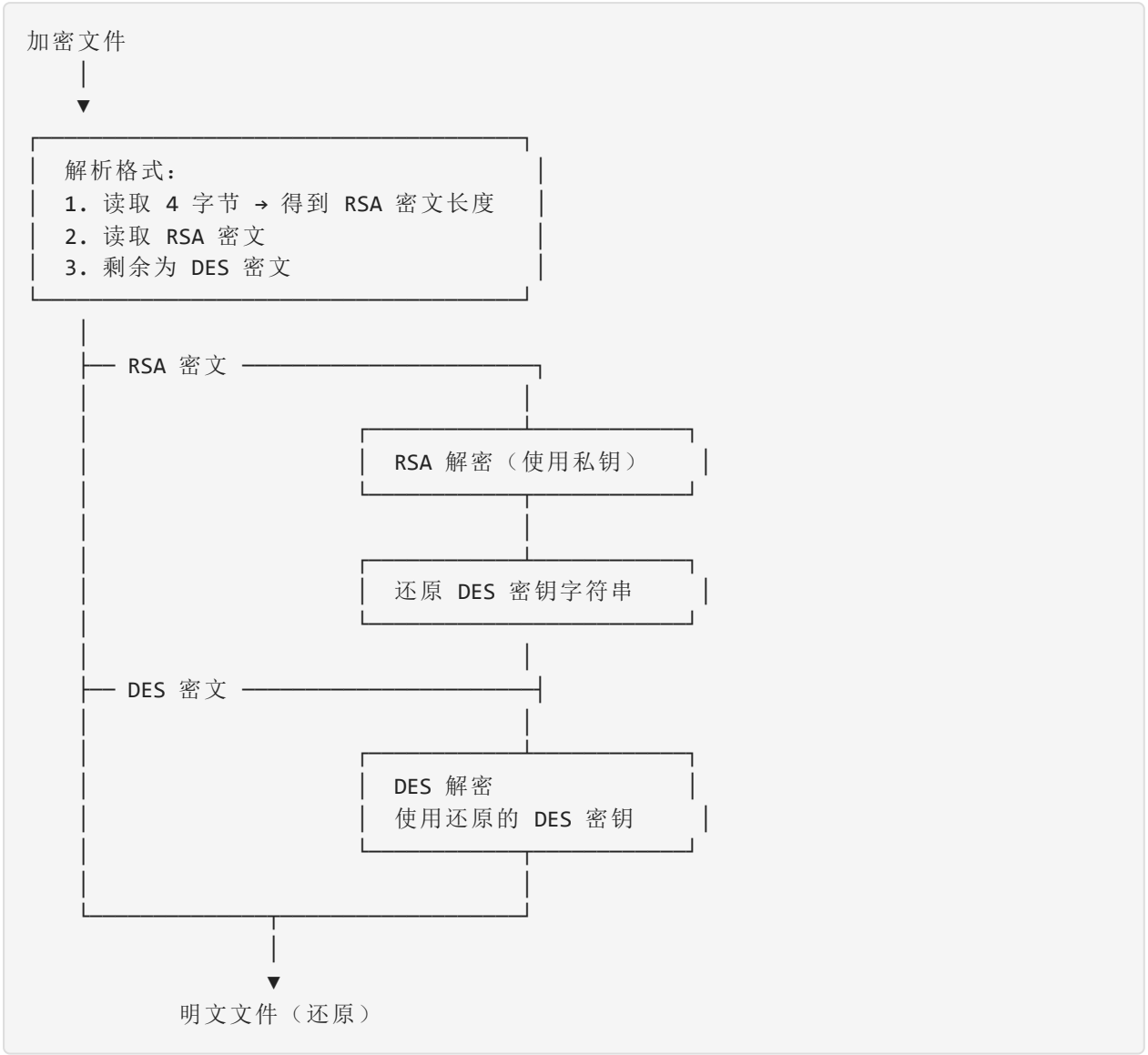
---

9. 完整数据流图解

加密（发送方）



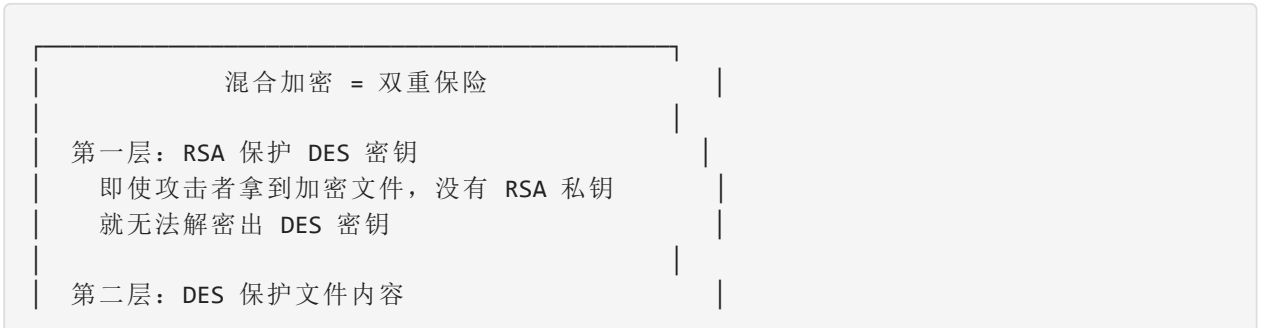
解密（接收方）



---

10. 安全性总结

双重保护



```
│    即使攻击者用某种方法得到了 DES 密钥，          │
│    还要有正确的 DES 密钥才能解密                  │
│                                                  │
│  最终效果：攻击者必须同时攻破两种算法              │
└──────────────────────────────────────────────────┘
```

安全注意事项

1. **RSA 私钥必须保密**——私钥泄露意味着所有用对应公钥加密的文件都不再安全
2. **DES 密钥用完即弃**——每次加密使用随机的 16 字符密钥，加密后无需记忆
3. **文件格式明确**——自定义格式让我们能灵活组合两种算法

安全局限（教学目的）

该系统是为**教学演示**设计的，不适合生产环境：

问题	说明	改进方案
ECB 模式	相同明文产生相同密文	改用 CBC 或 GCM 模式
无完整性校验	无法检测密文是否被篡改	添加 HMAC 或使用认证加密
无前向安全性	私钥泄露导致所有历史通信泄露	使用 Diffie-Hellman 密钥交换
无数字签名	无法验证发送方身份	添加 RSA 签名

---

11. 附录：依赖模块速查

模块	用途	说明
`struct`	打包/解包二进制数据	将整数转换为固定长度的字节序列
`secrets`	生成密码学安全的随机数	用于生成 DES 密钥（比 `random` 安全）
`string`	字符常量	提供字母、数字等预定义字符集
`os`	操作系统接口	检查文件是否存在、获取文件大小

模块	用途	说明
`tkinter`	GUI 工具包	创建窗口、按钮、输入框等
`tkinter.filedialog`	文件选择对话框	弹出”打开/保存文件”对话框
`tkinter.messagebox`	消息弹窗	显示信息、警告、错误弹窗
`Crypto.*`	密码学算法	`pycryptodome` 库，提供 DES/RSA 等算法

01 — 传输协议详解：如何在 TCP 流中”切分”消息

> 适用对象：电子信息工程学生，有基础 Python 知识即可。 本文围绕 `network_protocol.py` 和 `network_transfer.py` 展开。

一、为什么需要自定义协议？

1.1 TCP 是”流”，不是”消息”

TCP 是**流式协议**——它只保证字节按顺序到达，但**不保证一次 recv() 收到的正好是一整条消息**。

举个例子，发送方调用了两次 `send()`：

```
send('Hello')
send('World')
```

接收方可能一次 recv() 就收到 `'HelloWorld'`，也可能分三次收到 `'Hel'`、`'loW'`、`'orld'`。

这就叫**粘包和拆包**。为了解决这个问题，我们需要在 TCP 之上定义一层**应用层协议**，告诉接收方”一条消息从哪里开始、到哪里结束”。

1.2 本系统的解决方案

本系统用两种方式解决粘包问题：

技术手段	说明
固定长度头 + 变长体	先发固定大小的头部（包含后续数据长度），再发数据
特殊终止符	控制消息在末尾加 `\r\n` 作为边界

这两种方法在网络编程中非常经典。

二、协议常量与魔数

```
# network_protocol.py

PROTO_MAGIC = 0xDEAD            # 协议魔数
CHUNK_SIZE = 65536              # 每块 64KB
MSG_READY  = b"REDY"            # 服务端准备好接收
MSG_ACCEPT = b"ACPT"            # 服务端接受文件
MSG_REJECT = b"RJCT"            # 服务端拒绝
MSG_ACK    = b"ACK\x00"         # 确认收到一个数据块
MSG_DONE   = b"DONE"            # 传输完成
```

2.1 魔数 0xDEAD

```
PROTO_MAGIC = 0xDEAD
```

魔数（Magic Number） 是一个固定的标识值。发送方在文件头写入 `0xDEAD`，接收方读到后先检查这个值——如果不匹配，说明数据损坏或根本不是本协议的数据包，直接报错。

设计原则：用一个不太常见的值（比如 0xDEAD 在 ASCII 中不是常见字符）作为”门卫”，过滤掉垃圾数据。

2.2 分块大小 CHUNK_SIZE

```
CHUNK_SIZE = 65536  # 64 KB
```

一次发送 64 KB。为什么不分得更大？

- 太大：占用过多内存，且网络稍有丢包就要重传很多数据
- 太小：频繁的 send/recv 增加系统调用开销
- 64 KB 是一个实践经验值，在吞吐和延迟之间取得平衡

2.3 控制消息

控制消息是 4 字节的固定代码：

常量	字节	含义
MSG_READY	`REDY`	服务端已就绪，可以发送
MSG_ACCEPT	`ACPT`	服务端接受文件元信息

常量	字节	含义
MSG_REJECT	`RJCT`	服务端拒绝（后面带原因）
MSG_ACK	`ACK\x00`	确认收到一个数据块
MSG_DONE	`DONE`	传输全部完成

为什么是 4 字节？——刚好是一个 32 位整数的大小，方便用 `struct` 打包。

三、数据打包与解包

3.1 struct 模块速览

`struct` 是 Python 内置模块，用于在Python 值和 C 风格二进制字节之间互转。

```
import struct

# >I 表示：大端序(Big-endian) + 无符号整数(unsigned int, 4字节)
packed = struct.pack(">I", 42)   # 整数 42 → 4 字节
value  = struct.unpack(">I", packed)[0]  # 4 字节 → 整数 42
```

格式字符速查表：

字符	C 类型	Python 类型	大小
`I`	unsigned int	int	4 字节
`B`	unsigned char	int	1 字节
`H`	unsigned short	int	2 字节
`s`	char[]	bytes	可变

大端序（Big-Endian）：高位字节在前。网络协议通常用大端序（也叫网络字节序），保证不同 CPU 架构都能正确解析。

3.2 文件头部：pack_header / unpack_header

打包（发送方）

```python
def pack_header(filename: str, file_type: int, total_size: int) -> bytes:
    name_bytes = os.path.basename(filename).encode("utf-8")
    ext_bytes = os.path.splitext(filename)[1].encode("utf-8")[:256].ljust(256,
        b"\x00")
    return struct.pack(
        f">I B I I {len(name_bytes)}s 256s",
        PROTO_MAGIC,    # 4B: 魔数 0xDEAD
        file_type,      # 1B: 文件类型
        len(name_bytes),# 4B: 文件名长度
        total_size,     # 4B: 文件总大小
        name_bytes,     # N B: 文件名实际内容
        ext_bytes,      # 256B: 扩展名（固定长度，空余补0）
    )
```

逐字段解读：

字段	格式	字节数	说明
`PROTO_MAGIC`	`>I`	4	魔数，用于校验
`file_type`	`B`	1	0x01=普通文件，0x02=图片
`len(name_bytes)`	`I`	4	文件名的字节长度
`total_size`	`I`	4	文件总大小（字节）
`name_bytes`	`{N}s`	N	文件名
`ext_bytes`	`256s`	256	扩展名，固定占256字节

为什么扩展名要占固定 256 字节？——方便接收方在**不事先知道文件名长度**的情况下直接解析。这是”固定长度字段”的典型用法。

解包（接收方）

```python
def unpack_header(data: bytes) -> dict:
    magic, ftype, name_len, total_size = struct.unpack(
        ">I B I I", data[:HEADER_FIXED_SIZE]
    )
    if magic != PROTO_MAGIC:
        raise ValueError(f"协议魔数不匹配: 0x{magic:04X}")
    offset = HEADER_FIXED_SIZE
    name_bytes = data[offset : offset + name_len]
    offset += name_len
    ext_bytes = data[offset : offset + 256]
    filename = name_bytes.decode("utf-8")
    extension = ext_bytes.rstrip(b"\x00").decode("utf-8")
    return {
```

```
        "file_type": ftype,
        "filename": filename,
        "extension": extension,
        "total_size": total_size,
    }
```

逐行解读：

1. `HEADER_FIXED_SIZE = 4 + 1 + 4 + 4 = 13` —— 前面四个固定长度字段的总字节数
2. **先解包固定部分**：拿到魔数、文件类型、名字长度、总大小
3. **校验魔数**：不匹配就报错，这是协议的第一道防线
4. **动态读取名字**：根据 name_len 读取实际文件名（变长部分）
5. **读取扩展名**：固定 256 字节，然后用 `.rstrip(b"\x00")` 去掉末尾填充的
6. **返回字典**：便于调用方通过字段名访问

3.3 数据块：pack_chunk / unpack_chunk

大文件被切成 64 KB 的块依次发送，每一块都有自己的头部。

```
def pack_chunk(seq: int, data: bytes) -> bytes:
    return struct.pack(f">I I {len(data)}s", seq, len(data), data)
```

格式：`>I I {N}s`

字段	字节数	说明
`seq`（序号）	4	块的序号，从0开始
`len(data)`（数据长度）	4	本块实际数据长度
`data`（数据体）	N	真正的数据

解包：

```
def unpack_chunk(data: bytes) -> dict:
    seq, chunk_len = struct.unpack(">I I", data[:CHUNK_HEADER_SIZE])
    payload = data[CHUNK_HEADER_SIZE : CHUNK_HEADER_SIZE + chunk_len]
    return {"seq": seq, "length": chunk_len, "data": payload}
```

注意：接收方先读 8 字节头部（CHUNK_HEADER_SIZE = 8），拿到 chunk_len，再读 chunk_len 字节的数据体。

3.4 控制消息：pack_control / unpack_control

控制消息采用 “长度前缀 + 消息体 + 终止符” 的格式：

```
def pack_control(msg: bytes, payload: bytes = b"") -> bytes:
    body = msg + payload          # 4字节消息码 + 可选附加数据
    return struct.pack(f">I {len(body)}s", len(body), body) + MSG_TERMINATOR
```

三个组成部分：

1. 4 字节长度前缀：用 `>I` 打包 body 总长度
2. 消息体（4 + N 字节）：4 字节的消息码（如 `REDY`）+ 可选附加载荷
3. **终止符 *：2 字节，接收方读到 知道消息结束

为什么需要终止符？——它作为”消息边界”的第二层保险。即使长度前缀损坏，接收方仍然可以尝试从 恢复同步。

四、传输引擎：TransferProgress 类

```
class TransferProgress:
    def __init__(self, total: int, on_progress: ProgressCallback | None = None):
        self.total = total                # 总字节数
        self.transferred = 0              # 已传输字节数
        self.start_time = time.time()     # 开始时间
        self.on_progress = on_progress    # 进度回调函数

    def update(self, n: int):
        self.transferred += n
        if self.on_progress:
            elapsed = time.time() - self.start_time
            pct = min(self.transferred / self.total * 100, 100)
            rate = self.transferred / elapsed if elapsed > 0 else 0
            remaining = (self.total - self.transferred) / rate if rate > 0 else 0
            self.on_progress(pct, self.transferred, self.total, rate, max(remaining,
         0))
```

核心功能： - 追踪已传输字节数 - 计算百分比（pct） - 计算瞬时速率（rate = 已传字节 / 已用时间） - 估算剩余时间（remaining = 未传字节 / 速率） - 通过回调函数将进度信息传给 UI

进度回调类型签名：

```
ProgressCallback = Callable[[float, int, int, float, float], None]
# 参数依次是：百分比，已传字节，总字节，速率(B/s)，剩余秒数
```

五、完整发送流程：send_file

```
def send_file(
    sock, encrypted_data: bytes, filename: str,
    on_progress: ProgressCallback | None = None,
) -> bool:
```

时序图

```
客户端（发送方）                    服务端（接收方）
     |                                  |
     |  <--- MSG_READY  <-------------  |  ① 服务端就绪
     |                                  |
     |  ---- 文件头部 -------->          |  ② 发送元信息
     |                                  |
     |  <--- MSG_ACCEPT  <------------  |  ③ 服务端接受
     |                                  |
     |  ---- chunk[0] -------->         |  ④ 发送第 0 块
     |  <--- MSG_ACK  <---------------  |  ⑤ 确认收到
     |  ---- chunk[1] -------->         |  ⑥ 发送第 1 块
     |  <--- MSG_ACK  <---------------  |  ⑦ 确认收到
     |  ... (重复直到发完)  ...          |
     |                                  |
     |  ---- chunk[end] (空) -->        |  ⑧ 发送空块表示结束
     |  <--- MSG_DONE  <--------------  |  ⑨ 全部完成
```

逐段解读

```
def send_file(sock, encrypted_data, filename, on_progress=None):
    total = len(encrypted_data)

    # ① 等接收方就绪
    msg, _ = _recv_control(sock)
    if msg != MSG_READY:
        raise ConnectionError(f"服务端未就绪：{msg}")
```

`_recv_control` 内部调用 `_recv_exact` 确保收到完整的一条控制消息。如果收到的不是 `REDY`，说明服务端状态异常，直接报错。

```
# ② 发送文件头部（含元信息）
header = pack_header(filename, file_type, total)
sock.sendall(header)
```

用 `sendall` 而不是 `send`——`send` 可能只发送部分数据，`sendall` 保证全部发出。

```
# ③ 等待服务端接受/拒绝
msg, payload = _recv_control(sock)
if msg == MSG_REJECT:
    raise ConnectionError(f"服务端拒绝：{payload.decode(...)}")
if msg != MSG_ACCEPT:
    raise ConnectionError(f"预期 ACCEPT，收到：{msg}")
```

服务端可以根据文件名、类型等决定是否接受。拒绝时附带原因文本。

```
# ④ 分块发送
progress = TransferProgress(total, on_progress)
seq = 0
offset = 0
while offset < total:
    chunk_data = encrypted_data[offset : offset + CHUNK_SIZE]
    sock.sendall(pack_chunk(seq, chunk_data))
    msg, _ = _recv_control(sock)
    if msg != MSG_ACK:
        raise ConnectionError(f"传输错误")
    progress.update(len(chunk_data))
    seq += 1
    offset += CHUNK_SIZE
```

关键点：- 每次发 64 KB（CHUNK_SIZE）- 每发一块都**等待 ACK**，这是典型的**停-等协议**（Stop-and-Wait）- 收到 ACK 后更新进度 - seq 从 0 递增，用于检测丢包或乱序

```
# ⑤ 发送空块表示结束
sock.sendall(pack_chunk(seq, b""))
msg, _ = _recv_control(sock)
if msg == MSG_DONE:
    return True
raise ConnectionError(f"传输完成确认失败：{msg}")
```

发送一个**数据长度为 0 的空块**作为结束标记——这是 TCP 传输中非常常见的技巧。

六、完整接收流程：receive_file

```
def receive_file(
    sock, output_dir: str,
    on_progress: ProgressCallback | None = None,
) -> tuple[bytes, dict]:
```

逐段解读

```
def receive_file(sock, output_dir, on_progress=None):
    os.makedirs(output_dir, exist_ok=True)

    # ① 发送 READY，表示可以开始
    sock.sendall(pack_control(MSG_READY))

    # ② 解析文件头部（13字节固定 + N字节文件名 + 256字节扩展名）
    magic, ftype, name_len, total_size = struct.unpack(
        ">I B I I", _recv_exact(sock, 13))
    if magic != 0xDEAD:
        raise ValueError(f"魔数不匹配")
```

先读 13 字节（4+1+4+4）固定字段，校验魔数。

```
    name_bytes = _recv_exact(sock, name_len)
    ext_bytes = _recv_exact(sock, 256)
    filename = name_bytes.decode("utf-8")
    extension = ext_bytes.rstrip(b"\x00").decode("utf-8")
```

然后根据 name_len 动态读文件名，再读固定 256 字节的扩展名字段。

```
    # ③ 接受文件
    sock.sendall(pack_control(MSG_ACCEPT))

    # ④ 循环接收数据块
    progress = TransferProgress(total_size, on_progress)
    encrypted_data = bytearray()
    while True:
        seq, chunk_len = struct.unpack(">I I", _recv_exact(sock, 8))
        if chunk_len == 0:
            break                          # 空块 = 结束标记
        chunk_data = _recv_exact(sock, chunk_len)
        encrypted_data.extend(chunk_data)
        sock.sendall(pack_control(MSG_ACK))
        progress.update(chunk_len)

    # ⑤ 确认完成（附带 SHA256 哈希校验）
    sha = hashlib.sha256(bytes(encrypted_data)).hexdigest().encode()
    payload = sha + struct.pack(">Q", len(encrypted_data))
```

```
    sock.sendall(pack_control(MSG_DONE, payload))

    return bytes(encrypted_data), header_info
```

接收方核心逻辑

1. 先发 `REDY` 表示准备好
2. 解析头部拿到文件元信息
3. 回复 `ACCEPT` 接受文件
4. 循环接收数据块，每块回复 `ACK`
5. 遇到空块（chunk_len == 0）退出循环
6. 计算已接收数据的 SHA256 哈希，附带在 `DONE` 消息中

七、辅助函数：_recv_exact

```
def _recv_exact(sock, n: int) -> bytes:
    buf = b""
    while len(buf) < n:
        chunk = sock.recv(n - len(buf))
        if not chunk:
            raise ConnectionError("连接已断开")
        buf += chunk
    return buf
```

这是整个协议最基础的函数。 TCP 的 `recv(n)` 不保证一次收 `n` 字节，所以必须用循环收满为止。

想象一下：接收方期待 100 字节，但 TCP 只送来了 40 字节。`_recv_exact` 会继续等待，直到凑满 100 字节才返回。

八、总结

概念	本系统的实现
魔数校验	0xDEAD 在文件头，过滤垃圾数据
固定长度头	文件头前13字节固定，扩展名固定256字节

概念	本系统的实现
长度前缀	控制消息先发4字节长度
分块传输	每块64KB，带序号
停-等确认	每发一块等ACK后再发下一块
终止符	控制消息以 结尾
进度追踪	TransferProgress 回调机制
完整性校验	DONE 消息附带 SHA256

02 — C/S 架构详解：服务端与客户端的握手与加密传输

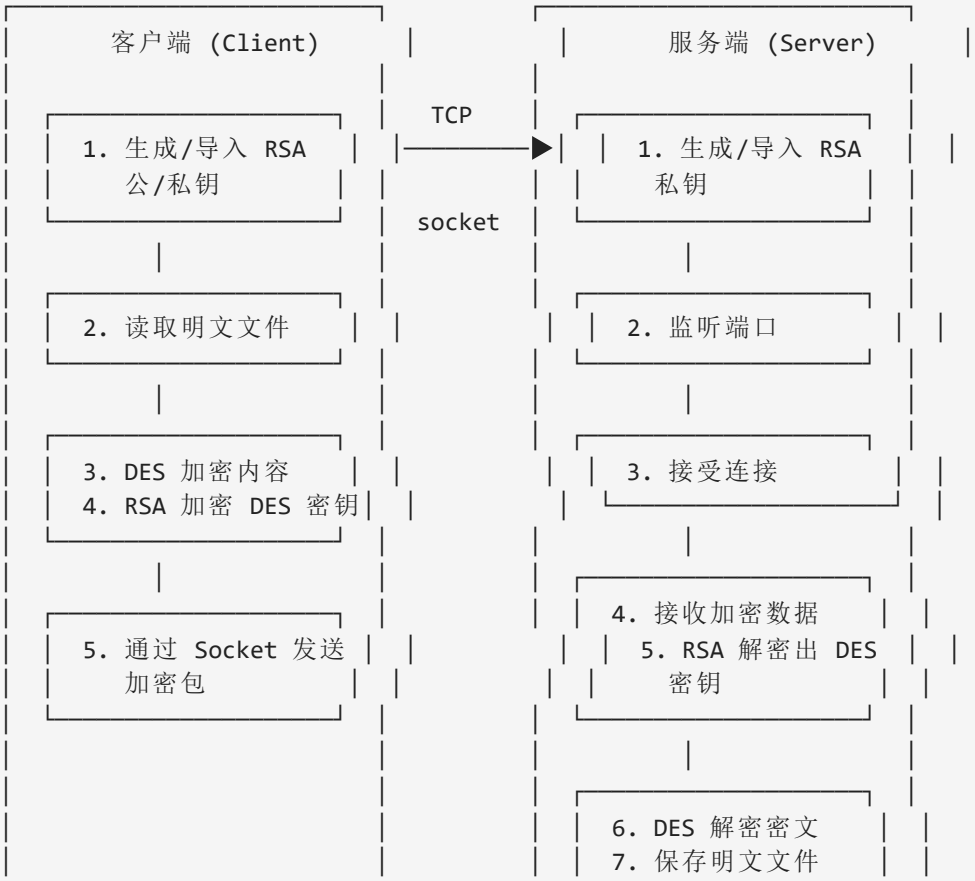
> 适用对象：电子信息工程学生，有基础 Python 知识即可。本文围绕 `network_server.py` 和 `network_client.py` 展开。

一、什么是 C/S 架构？

C/S 架构（Client-Server Architecture）是最常见的网络应用模型：

- **服务端（Server）**：持续运行，等待别人连接，提供服务
- **客户端（Client）**：主动发起连接，请求服务

架构图



```
└──────────────────────────┘          └──────────────────────────┘
```

二、服务端（Server）

2.1 类结构概览

```
class ServerApp:
    def __init__(self, root: tk.Tk):
        self.root = root
        self._rsa_key = None              # RSA 密钥对象
        self._server_socket = None        # 监听 socket
        self._running = False             # 服务是否在运行
        self._accept_thread = None        # 接收连接的线程
        self._preview_photo = None        # 预览图片
        self._build_ui()                  # 构建 GUI
```

服务端的核心职责： 1. 管理 RSA 密钥（生成、导入、导出） 2. 创建 TCP 服务端，监听端口 3. 接受客户端连接 4. 接收加密数据，自动解密并保存

2.2 标准 Socket 服务端流程

```
# network_server.py _on_start 方法中的关键代码

# ① 创建 socket
self._server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# ② 设置地址重用（防止"Address already in use"错误）
self._server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# ③ 绑定地址和端口
self._server_socket.bind(("0.0.0.0", port))

# ④ 开始监听，backlog = 5 表示最多排队 5 个连接
self._server_socket.listen(5)

# ⑤ 设置超时，让 accept() 可以定期检查 _running 标志
self._server_socket.settimeout(1.0)
```

这五步是所有 TCP 服务端的标准流程：

步骤	API	说明
①	`socket()`	创建套接字
②	`setsockopt()`	设置 SO_REUSEADDR 避免端口占用
③	`bind()`	绑定到 IP 和端口
④	`listen()`	开始监听，等待连接
⑤	`accept()`	接受客户端连接（阻塞）

关键参数说明：

- `AF_INET`：使用 IPv4 地址
- `SOCK_STREAM`：使用 TCP 协议（面向连接的可靠传输）
- `"0.0.0.0"`：绑定到所有网络接口，让局域网内任何设备都能连接
- `SO_REUSEADDR`：允许快速重启而不被”地址已占用”错误卡住

2.3 多线程接受连接

```
def _accept_loop(self):
    while self._running:
        try:
            client_sock, addr = self._server_socket.accept()
        except socket.timeout:
            continue        # 超时后重新检查 _running
        except Exception:
            break           # 其他错误退出循环
        # 使用 root.after() 在主线程中处理客户端
        self.root.after(0, lambda a=addr, c=client_sock: self._handle_client(a, c))
```

为什么 accept() 要设超时？ - 如果没有超时，accept() 会永远阻塞，无法响应”停止服务”按钮 - 设置 1 秒超时，每 1 秒醒来检查一次 `_running` 标志

为什么接受连接后要用线程处理？ - 启动一个线程运行 accept loop - 每次 accept 到新连接后，用 `root.after(0, ...)` 在主线程中处理 - 这样做的好处是 GUI 不会卡死

2.4 接收与解密流程

```
def _handle_client(self, addr, client_sock):
    # ① 定义进度回调
    def on_progress(pct, transferred, total, rate, remaining):
        # root.after(0, ...) 保证在主线程更新 GUI
```

```
        self.root.after(0, lambda: self._update_progress(
            pct, transferred, total, rate, remaining))

    # ② 调用传输层接收数据
    enc_data, header = receive_file(client_sock, self._dir_var.get(), on_progress)

    # ③ 保存密文到磁盘
    filename = header["filename"]
    enc_path = os.path.join(self._dir_var.get(), filename + ".enc")
    with open(enc_path, "wb") as f:
        f.write(enc_data)

    # ④ 如果开启了自动解密且有私钥
    if self._auto_decrypt_var.get() and self._rsa_key is not None:
        plain_data = decrypt_bytes(enc_data, self._rsa_key)
        out_path = os.path.join(self._dir_var.get(), filename)
        with open(out_path, "wb") as f:
            f.write(plain_data)
        # 如果是图片文件，显示预览
        if is_image_file(out_path):
            self._show_preview(out_path)
```

接收端的数据流：

```
TCP socket → receive_file() → 密文字节 .enc 文件
                                    ↓
                              decrypt_bytes()
                              (RSA解密→DES密钥)
                              (DES解密→明文)
                                    ↓
                              明文文件 + 图片预览
```

三、客户端（Client）

3.1 类结构概览

```
class ClientApp:
    def __init__(self, root: tk.Tk):
        self.root = root
        self._rsa_key = None              # RSA 密钥对象
        self._sock = None                 # 与服务端的连接 socket
        self._connected = False           # 连接状态
        self._sending = False             # 是否正在发送
        self._thumbnail_photo = None      # 缩略图
        self._build_ui()
```

客户端的核心职责： 1. 管理 RSA 密钥 2. 连接服务端 3. 选择文件、输入 DES 密钥 4. 加密文件并发送

3.2 连接服务端

```
def _on_connect(self):
    host = self._host_var.get().strip()
    port = int(self._port_var.get())

    # ① 创建 socket
    self._sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # ② 设置连接超时（10 秒连不上就报错）
    self._sock.settimeout(10)

    # ③ 发起连接
    self._sock.connect((host, port))

    # ④ 连接成功后取消超时（之后由协议层控制）
    self._sock.settimeout(None)

    self._connected = True
```

客户端这边只需要三步：创建 socket → connect → 设置超时。

为什么连接成功后要取消超时？ - 传输大文件可能持续很长时间 - 连接的 timeout 会影响后续的 recv，不取消的话长时间没收到数据会超时断开 - 传输层的 `_recv_exact` 已经有自己的超时/断开处理逻辑

3.3 加密并发送（核心流程）

```
def _on_send(self):
    # ① 验证 DES 密钥
    des_key = self._validate_des_key()
    if des_key is None:
        return
    # ② 检查 RSA 密钥
    if not self._check_rsa():
        return
    # ③ 检查连接状态
    if not self._connected or self._sock is None:
        return
    # ④ 检查文件是否存在
    infile = self._infile_var.get()
    if not infile or not os.path.isfile(infile):
        return
```

```
    # ⑤ 在新线程中执行加密和发送（避免卡 GUI）
    def task():
        # 读取明文
        with open(infile, "rb") as f:
            plain = f.read()
        # DES + RSA 混合加密
        enc_pkg, used_key = encrypt_bytes(plain, self._rsa_key)
        # 通过 socket 发送加密数据
        send_file(self._sock, enc_pkg, os.path.basename(infile), on_progress)

    threading.Thread(target=task, daemon=True).start()
```

发送端的数据流：

```
明文文件 → encrypt_bytes() → 加密数据包 → send_file() → TCP socket
              │
              ├─ DES 加密文件内容
              └─ RSA 加密 DES 密钥
```

3.4 加密函数 encrypt_bytes

```
# hybrid_crypto.py
def encrypt_bytes(data: bytes, rsa_public_key: RSA.RsaKey) -> tuple[bytes, str]:
    # ① 随机生成一个 DES 密钥
    des_key_str = _generate_des_key()
    # ② 用 DES 加密文件内容
    cipher_data = des_encrypt(data, des_key_str)
    # ③ 用 RSA 公钥加密 DES 密钥
    encrypted_des_key = rsa_encrypt(des_key_str.encode("utf-8"), rsa_public_key)
    # ④ 打包：[长度][加密的DES密钥][DES加密的文件]
    buf = struct.pack(">I", len(encrypted_des_key))
    buf += encrypted_des_key
    buf += cipher_data
    return buf, des_key_str
```

3.5 解密函数 decrypt_bytes

```
# hybrid_crypto.py
def decrypt_bytes(data: bytes, rsa_private_key: RSA.RsaKey) -> bytes:
    # ① 读取密钥长度
    key_len = struct.unpack(">I", data[:4])[0]
    # ② 提取加密的 DES 密钥
    encrypted_des_key = data[4:4 + key_len]
    # ③ 提取 DES 加密的数据
    cipher_data = data[4 + key_len:]
    # ④ RSA 解密 → DES 密钥
    des_key_str = rsa_decrypt(encrypted_des_key, rsa_private_key).decode("utf-8")
```

```
    # ⑤ DES 解密 → 明文
    return des_decrypt(cipher_data, des_key_str)
```

3.6 为什么用 DES + RSA 混合加密？

算法	优点	缺点
DES（对称加密）	加密速度快，适合大文件	密钥分发困难
RSA（非对称加密）	密钥可以公开分发	加密速度慢，不适合大文件

混合方案 = DES（加密内容） + RSA（加密DES密钥）

```
生成随机 DES 密钥 (16字符)
        │
        ├── 用 DES 加密文件内容 ──→ 密文（很大）
        │
        └── 用 RSA 公钥加密 DES 密钥 ──→ 加密的密钥（很短，仅256字节）
                                                    │
        发送的整体结构： [加密的密钥长度][加密的密钥][DES密文]
```

这样既享受了 RSA 的密钥管理便利性，又享受了 DES 的加密速度。

四、进度追踪与状态反馈

4.1 进度更新

```
def _update_progress(self, pct, transferred, total, rate, remaining):
    self._progress.config(value=pct)
    if total > 0:
        t_mb = transferred / (1024 * 1024)
        o_mb = total / (1024 * 1024)
        r_mb = rate / (1024 * 1024)
        self._rate_label.config(
            text=f"{t_mb:.1f} MB / {o_mb:.1f} MB  |  "
                 f"{r_mb:.1f} MB/s  |  剩余 ~{remaining:.0f}s")
```

`TransferProgress` 类在传输层计算好百分比、速率、剩余时间，回调到 UI 层显示。

4.2 图片预览

```
def _show_preview(self, path):
    if HAS_PIL:
        photo = bytes_to_photoimage(open(path, "rb").read())
        if photo:
            self._preview_photo = photo
            self._preview_label.config(image=photo, text="")
```

用 PIL（Pillow 库）读取图片，生成缩略图，显示在 tkinter 标签中。

4.3 线程安全

```
# 在子线程中需要更新 GUI 时，用 root.after()
self.root.after(0, lambda: self._log("传输完成！"))
self.root.after(0, lambda: self._task_label.config(text=f"发送成功：{filename}"))
```

Tkinter 不是线程安全的——你不能在非主线程中直接操作 GUI 组件。`root.after(0, callback)` 将回调函数排队到主线程的事件循环中执行。

五、完整传输示例

5.1 客户端发送

```
1. 生成 RSA 密钥对
2. 导出公钥给服务端（或导入服务端的公钥）
3. 选择要发送的文件（如 photo.png）
4. 输入 DES 密钥（如 "MyKey12345678!"）
5. 连接到服务端（127.0.0.1:8888）
6. 点击"加密并发送"

控制台输出：
[10:00:00] 正在生成 2048 位 RSA 密钥对...
[10:00:02] RSA 密钥对生成成功（2048 位）。
[10:00:05] 已连接到服务器 127.0.0.1:8888
[10:00:06] 加密中：photo.png (2048576 字节)
[10:00:07] 加密完成，DES 会话密钥：aB3#dE9$fG1*hJ2@，开始传输...
[10:00:08] 传输完成！
```

5.2 服务端接收

```
1. 导入客户端的 RSA 公钥（或使用自己的私钥）
2. 启动服务（端口 8888）
3. 等待客户端连接
```

```
控制台输出：
[10:00:00] 服务端已启动。请导入 RSA 私钥并启动服务。
[10:00:02] 服务已启动，监听 0.0.0.0:8888
[10:00:06] 新客户端连接: 192.168.1.100:54321
[10:00:08] 密文已保存: received/photo.png.enc (2048640 字节)
[10:00:08] 解密完成 → received/photo.png (2048576 字节)
[10:00:08] 文件 SHA256: a1b2c3d4e5f6...
```

六、总结

角色	核心职责	关键方法
服务端	监听、接收、解密	`socket().bind().listen().accept()`
客户端	连接、加密、发送	`socket().connect()` → `encrypt_bytes()` → `send_file()`
混合加密	DES 加密内容 + RSA 加密 DES 密钥	`encrypt_bytes()` / `decrypt_bytes()`
线程安全	子线程发数据，主线程更新 GUI	`root.after(0, callback)`
进度	实时反馈传输状态	`TransferProgress` → `_update_progress()`

03 — GUI 编程基础：用 Tkinter 构建网络传输界面

> 适用对象：电子信息工程学生，有基础 Python 知识即可。 本文分析 `network_server.py` 和 `network_client.py` 中的 Tkinter GUI 代码。

一、什么是 Tkinter？

Tkinter 是 Python 自带的标准 GUI（图形用户界面）库，无需额外安装。

```
import tkinter as tk
from tkinter import filedialog, messagebox, scrolledtext, ttk
```

关键子模块：

模块	提供
`tkinter`	基础组件：Button, Label, Entry, Frame 等
`tkinter.ttk`	主题化组件：LabelFrame, Progressbar, Combobox
`tkinter.scrolledtext`	带滚动条的文本框
`tkinter.filedialog`	文件选择对话框
`tkinter.messagebox`	消息弹窗

1.1 一个最小的 Tkinter 程序

```
import tkinter as tk

root = tk.Tk()                  # 创建主窗口
root.title("我的程序")           # 设置标题
root.geometry("400x300")        # 设置大小

label = tk.Label(root, text="你好，世界！")
label.pack()                    # 放置到窗口上

root.mainloop()                 # 进入事件循环（必须）
```

`mainloop()` 是 GUI 的”心脏”——它不断检查有没有事件发生（鼠标点击、键盘输入等），并调用相应的处理函数。

二、总体布局结构

服务端和客户端的 GUI 布局非常相似。以下是服务端的布局层次：

```
root (主窗口)
  ├── RSA 密钥管理 (ttk.LabelFrame)
  │     ├── 按钮行: 导入公钥 | 导入私钥 | 生成密钥对 | 导出公钥
  │     └── 状态标签: "状态：密钥已就绪（2048 位）"
  │
  ├── 服务配置 (ttk.LabelFrame)
  │     ├── 端口输入 + 本机 IP 显示
  │     └── 启动/停止按钮 + 状态指示灯
  │
  ├── 接收设置 (ttk.LabelFrame)
  │     ├── 目录选择 + 浏览按钮
  │     └── 自动解密复选框
  │
  ├── 接收进度 (ttk.LabelFrame)
  │     ├── 任务描述标签
  │     ├── 进度条 (ttk.Progressbar)
  │     └── 速率/剩余时间标签
  │
  ├── 文件预览 (ttk.LabelFrame)
  │     ├── 图片预览标签
  │     └── 文件信息标签
  │
  └── 日志 (ttk.LabelFrame)
        └── ScrolledText 文本框
```

2.1 几何管理器：pack

Tkinter 有三种布局方式：pack、grid、place。本系统全部使用 pack。

```
# pack 的核心概念：将组件"打包"到父容器的某个方向
widget.pack(side=tk.LEFT)    # 靠左放置
widget.pack(side=tk.RIGHT)   # 靠右放置
widget.pack(fill=tk.X)       # 水平方向填满父容器
widget.pack(fill=tk.BOTH, expand=True)  # 双向填满，并允许扩展
widget.pack(padx=8, pady=4)  # 设置外边距
```

2.2 模块分组：LabelFrame

```
# 创建一个带标题的分组框
rsa_frame = ttk.LabelFrame(self.root, text="RSA 密钥管理")

# 像普通 Frame 一样往里面放组件
row = ttk.Frame(rsa_frame)
row.pack(fill=tk.X, padx=8, pady=4)

btn = ttk.Button(row, text="生成密钥对", command=self._on_generate)
btn.pack(side=tk.LEFT, padx=2)
```

`LabelFrame` 比普通 `Frame` 多了一个标题文字，非常适合将功能相关的控件分组。

三、关键组件详解

3.1 Entry（文本输入框）

```
# 端口输入，宽度 8 个字符
self._port_var = tk.StringVar(value="8888")
ttk.Entry(row, textvariable=self._port_var, width=8).pack(side=tk.LEFT, padx=4)

# DES 密钥输入，用 show="*" 隐藏内容（密码模式）
self._des_key_var = tk.StringVar()
ttk.Entry(row, textvariable=self._des_key_var, show="*", width=24).pack(side=tk.LEFT,
          padx=4)
```

StringVar 是 Tkinter 的变量包装类：当 Entry 内容改变时，StringVar 自动更新；反过来设置 StringVar，Entry 的内容也随之改变。

参数 `show="*"` 让输入内容显示为星号，适合密码输入。

3.2 Button（按钮）

```
# 按钮绑定 command 回调函数
self._start_btn = ttk.Button(row2, text="启动服务",
                             command=self._on_start)
self._start_btn.pack(side=tk.LEFT, padx=2)
```

点击按钮 → Tkinter 自动调用 `self._on_start()`。

3.3 Checkbutton（复选框）

```
self._auto_decrypt_var = tk.BooleanVar(value=True)
ttk.Checkbutton(set_frame, text="接收后自动解密",
                variable=self._auto_decrypt_var).pack(anchor=tk.W, padx=8, pady=4)
```

`BooleanVar` 绑定复选框的选中状态。`value=True` 表示默认勾选。

3.4 Combobox（下拉选择框）

```
self._rsa_bits = ttk.Combobox(row0, values=["1024", "2048", "4096"],
                              state="readonly", width=6)
self._rsa_bits.set("2048")  # 默认选中 2048
```

`state="readonly"` 表示用户只能从列表中选择，不能输入自定义值。

3.5 Progressbar（进度条）

```
self._progress = ttk.Progressbar(prog_frame, mode="determinate")
self._progress.pack(fill=tk.X, padx=8, pady=4)
```

`mode="determinate"`：确定性模式，最大值 100，通过 `self._progress.config(value=50)` 设置当前进度。

另一种模式是 `"indeterminate"`（不确定模式），来回滚动表示”正在处理，但不知道进度”。

3.6 ScrolledText（滚动文本框）

```
from tkinter import scrolledtext

self._log_area = scrolledtext.ScrolledText(
    log_frame, height=8, state=tk.DISABLED, wrap=tk.WORD)
self._log_area.pack(fill=tk.BOTH, expand=True, padx=4, pady=4)
```

参数说明： - `height=8`：显示 8 行文字 - `state=tk.DISABLED`：只读模式，用户不能编辑 - `wrap=tk.WORD`：按单词换行（而不是按字符换行）

四、日志系统

```
def _log(self, msg: str):
    ts = time.strftime("%H:%M:%S")
    self._log_area.configure(state=tk.NORMAL)      # 临时启用编辑
    self._log_area.insert(tk.END, f"[{ts}] {msg}\n")  # 追加文字
    self._log_area.see(tk.END)                      # 滚动到末尾
    self._log_area.configure(state=tk.DISABLED)     # 恢复只读
```

为什么先启用再禁用？ - ScrolledText 被设置为 DISABLED 防止用户误编辑 - 但程序自己需要写日志，所以先改成 NORMAL，写完再改回 DISABLED

`see(tk.END)` 让文本框自动滚到最后一行，保证最新日志始终可见。

五、事件驱动编程

5.1 基本概念

事件驱动编程 是 GUI 程序的运行模式：

```
用户点击按钮
    ↓
Tkinter 生成"按钮点击"事件
    ↓
调用绑定的 command 回调函数
    ↓
函数执行完毕，回到事件循环等待下一个事件
```

5.2 主要事件处理函数

触发方式	回调函数	功能
点击”生成密钥对”	`_on_generate_keypair()`	生成 RSA 密钥
点击”导入公钥”	`_on_import_pubkey()`	从 PEM 文件导入公钥
点击”启动服务”	`_on_start()`	创建 socket 开始监听
点击”停止服务”	`_on_stop()`	关闭 socket 停止监听
点击”加密并发送”	`_on_send()`	加密文件并发送
点击”连接”	`_on_connect()`	连接到服务端

5.3 按钮状态管理

```
# 启动时
self._start_btn.config(state=tk.DISABLED)  # 禁用"启动"按钮
self._stop_btn.config(state=tk.NORMAL)     # 启用"停止"按钮

# 停止时
self._start_btn.config(state=tk.NORMAL)    # 启用"启动"按钮
self._stop_btn.config(state=tk.DISABLED)   # 禁用"停止"按钮
```

这是一种保护机制——防止用户在服务已运行时再次点击启动。

5.4 文件对话框

```
# 打开文件（导入密钥/选择要发送的文件）
path = filedialog.askopenfilename(
    title="选择要发送的文件",
    filetypes=[("PEM 文件", "*.pem"), ("所有文件", "*.*")]
)

# 保存文件（导出密钥）
path = filedialog.asksaveasfilename(
    defaultextension=".pem",
    filetypes=[("PEM 文件", "*.pem"), ("所有文件", "*.*")]
)

# 选择目录
path = filedialog.askdirectory(title="选择接收目录")
```

三个对话框都返回用户选择的路径，如果用户取消了对话，返回空字符串。

六、线程与 Tkinter 的配合

6.1 问题：GUI 会卡死

```
# 错误示例：在主线程中执行耗时操作
def _on_send_naive(self):
    time.sleep(10)          # GUI 会卡死 10 秒
    send_file(...)          # 发送文件期间 GUI 也无法响应
```

6.2 解决方案：后台线程

```
def _on_send(self):
    self._sending = True
```

```
    def task():
        # 在子线程中执行耗时操作
        send_file(self._sock, enc_pkg, filename, on_progress)
        # 完成后通过 root.after() 通知主线程
        self.root.after(0, lambda: self._log("传输完成！"))

    threading.Thread(target=task, daemon=True).start()
```

`daemon=True`：设置为守护线程，主程序退出时自动结束。

6.3 root.after() 的原理

```
self.root.after(0, callback)      # 尽快执行 callback（排队到主线程）
self.root.after(1000, callback)   # 1 秒后执行 callback
```

`after()` 的运作机制：

```
子线程 ──→ root.after(0, callback)
                    │
                    ▼
          主线程事件队列 ←── Tkinter 主循环
                    │
                    ▼
              执行 callback
              （此时可以安全操作 GUI）
```

核心规则：永远不要在子线程中直接操作 GUI 组件。所有 GUI 更新都必须通过 `root.after()` 排队到主线程。

6.4 正确的线程模式

```
def task():
    try:
        # 子线程：耗时操作
        with open(infile, "rb") as f:
            plain = f.read()
        enc_pkg, used_key = encrypt_bytes(plain, self._rsa_key)
        send_file(self._sock, enc_pkg, os.path.basename(infile), on_progress)

        # 通过 after() 回到主线程更新 GUI
        self.root.after(0, lambda: self._log("传输完成！"))
        self.root.after(0, lambda: self._task_label.config(text="发送成功"))
        self.root.after(0, lambda: messagebox.showinfo("完成", "传输成功！"))
    except Exception as e:
        self.root.after(0, lambda: self._log(f"发送失败：{e}"))
```

```
            self.root.after(0, lambda: messagebox.showerror("错误", str(e)))
        finally:
            self._sending = False
            self.root.after(0, lambda: self._send_btn.config(state=tk.NORMAL))
```

七、客户端的 DES 密钥验证

```
def _validate_des_key(self) -> str | None:
    k1 = self._des_key_var.get()         # 输入的密钥
    k2 = self._des_confirm_var.get()     # 确认输入的密钥

    if not k1:
        messagebox.showwarning("提示", "请输入 DES 密钥。")
        return None
    if len(k1) > 16:
        messagebox.showwarning("提示", "DES 密钥最多 16 个字符。")
        return None
    if k1 != k2:
        messagebox.showwarning("提示", "两次输入的 DES 密钥不一致。")
        return None
    return k1
```

两次输入设计：防止用户输错密钥（对称加密中密钥输错将导致无法解密）。

八、整体代码结构总结

服务端（network_server.py）

```
ServerApp class
  ├── __init__()          初始化窗口、变量、调用 _build_ui()
  ├── _build_ui()         构建所有 GUI 组件
  ├── _log()              日志输出
  ├── _get_local_ips()    获取本机 IP
  ├── _update_rsa_status() 更新密钥状态显示
  ├── _on_generate_keypair() 生成 RSA 密钥对
  ├── _on_import_pubkey()   导入公钥
  ├── _on_import_privkey()  导入私钥
  ├── _on_export_pubkey()   导出公钥
  ├── _on_browse_dir()      选择接收目录
  ├── _on_start()           启动 TCP 服务
  ├── _on_stop()            停止 TCP 服务
  ├── _accept_loop()        连接接受循环（子线程）
  ├── _handle_client()      处理客户端（解密、保存）
```

```
  ├── _update_progress()    更新进度条
  └── _show_preview()       显示图片预览
```

客户端（network_client.py）

```
ClientApp class
  ├── __init__()          初始化窗口、变量、调用 _build_ui()
  ├── _build_ui()         构建所有 GUI 组件
  ├── _log()              日志输出
  ├── _update_rsa_status() 更新密钥状态显示
  ├── _on_generate()      生成 RSA 密钥对
  ├── _on_import_pub()    导入公钥
  ├── _on_import_priv()   导入私钥
  ├── _on_export_pub()    导出公钥
  ├── _on_export_priv()   导出私钥
  ├── _on_browse()        选择发送文件
  ├── _show_thumbnail()   显示缩略图
  ├── _on_connect()       连接服务端
  ├── _on_disconnect()    断开连接
  ├── _on_send()          加密并发送（启动子线程）
  ├── _validate_des_key() 验证 DES 密钥
  └── _update_progress()  更新进度条
```

九、总结

概念	本系统的应用
LabelFrame	给功能分组（RSA密钥、服务配置、接收设置等）
StringVar / BooleanVar	绑定输入框/复选框的数据
pack 布局	所有组件都用 pack 排列
Button + command	点击按钮触发回调函数
Progressbar	显示传输进度
ScrolledText	日志输出框（只读模式）
filedialog	文件选择（导入/导出密钥、选择文件）
messagebox	信息提示、错误警告
threading	耗时操作放到子线程
root.after()	子线程安全地更新 GUI

扩散模型原理

> 适用对象：零基础EE学生
>
> 前置知识：深度学习基础（神经网络、PyTorch基本概念）

---

1. 扩散模型是什么？

一个类比：从大理石中雕刻出雕像

想象你有一块纯白的大理石（**纯噪声**），你的目标是雕刻出一座精美的雕像（**生成图像**）。

传统生成模型的做法是：直接教你如何一刀一刀雕刻出雕像——这太难了，因为雕刻的每一步都需要精确规划。

扩散模型的做法恰恰相反：

1. 先拿一座已有的雕像，**不断往上糊泥巴**，直到它变成一块看不出形状的大理石
2. 然后，**学习如何把糊上去的泥巴一步步去掉**，恢复出雕像

你可能会问：“为什么要先破坏再恢复？这不折腾吗？”

答案是：**学会”去掉泥巴”比学会”直接雕刻”要容易得多。** 因为每一步去掉多少泥巴、去掉哪里，都是有规律可循的——你只需要学习”当前状态离最终雕像还差多少泥巴”。

核心直觉

概念	通俗类比
原始图像	一座精美的雕像
前向过程（加噪）	往雕像上糊泥巴，越糊越厚
纯噪声	一块完全看不出形状的大理石块
反向过程（去噪）	把泥巴一层层去掉，恢复雕像
训练	观察”糊泥巴→去掉”的全过程，学会判断当前状态

概念	通俗类比
生成（推理）	从一块纯大理石开始，一步步去掉不存在的”泥巴”

---

2. 前向过程：逐步加噪

前向过程就是**给图片逐步添加高斯噪声**，直到图片完全变成纯噪声。

数学描述（高能预警——但别怕）

假设我们有一张原始图片 x_0（比如一张小猫照片）。

在每一步 t，我们往图片上加一点高斯噪声，得到 x_t：

$$x_t = \sqrt{1 - \beta_t} \cdot x_{t-1} + \sqrt{\beta_t} \cdot \epsilon$$

其中： - x_t 是第 t 步加噪后的图片 - β_t 是一个很小的数（比如 0.0001），控制每一步加多少噪声 - ϵ 是从标准高斯分布中采样的随机噪声 $\mathcal{N}(0, 1)$ - t 从 1 到 T（通常 $T = 1000$）

对于EE学生的解释： 这本质上是一个**低通滤波器加噪声注入**的过程。每一步都让信号的信噪比（SNR）降低一点点。1000步之后，SNR趋近于0，图像完全被噪声淹没。

一个重要的性质

因为每一步加的噪声都很小，而且高斯噪声有**可加性**（多个高斯分布相加还是高斯分布），我们可以**一步到位**算出任意第 t 步的结果：

$$x_t = \sqrt{\bar{\alpha}_t} \cdot x_0 + \sqrt{1 - \bar{\alpha}_t} \cdot \epsilon$$

其中 $\bar{\alpha}_t$ 是前 t 步的累乘系数。t 越大，$\bar{\alpha}_t$ 越接近 0，x_t 就越接近纯噪声。

---

3. 反向过程：逐步去噪（核心！）

前向过程很简单：就是反复加噪声嘛。

反向过程才是扩散模型的**灵魂**——我们训练一个神经网络，让它学会**从噪声中恢复出原始图像**。

数学描述

反向过程的核心公式：

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \cdot \epsilon_\theta(x_t, t) \right) + \sigma_t \cdot z$$

看起来复杂？让我们拆解一下：

符号	含义	通俗理解
x_t	当前（有噪声的）图片	当前糊着泥巴的雕像
$\epsilon_\theta(x_t, t)$	**神经网络预测的噪声**	神经网络说：“这层泥巴长这样”
$\frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}}$	去噪的缩放系数	“每次应该去掉多厚的泥巴”
$\sigma_t \cdot z$	随机扰动项	去噪时加一点点随机性，防止死板
x_{t-1}	去噪后的图片	去掉一层泥巴后的雕像

神经网络在做什么？

神经网络 ϵ_θ（通常是一个 UNet）接收两个输入： 1. **当前噪声图** x_t（比如一个 512×512×3 的带噪图像） 2. **当前时间步** t（告诉网络”这是第几步”，因为不同步需要去噪的强度不同）

输出：**预测的噪声** ϵ（和输入图像同尺寸）

每一层去噪的公式可以理解为：

去噪后图像 = 当前图像 − 预测噪声 × 缩放系数

---

4. 训练过程

训练数据准备

假设我们有 100 万张猫的图片。

对于每张图片 x_0： 1. 随机选择一个时间步 t（从 1 到 1000） 2. 使用前向过程公式，直接从 x_0 算出 x_t（不用一步步走，一步到位） 3. 记录我们添加的真实噪声 ϵ

训练目标

让神经网络预测的噪声 $\epsilon_\theta(x_t, t)$ 尽可能接近真实添加的噪声 ϵ：

$$\mathcal{L} = \mathbb{E}_{x_0, \epsilon, t}[\| \epsilon - \epsilon_\theta(x_t, t) \|^2]$$

翻译成人话： 平均来说，让网络预测的噪声和实际加的噪声之间的差值的平方最小化。

对于EE学生的解释： 这本质上是一个**最小均方误差（MMSE）估计**问题。神经网络在学习做最优的噪声估计器。

训练流程伪代码

```
for 每张图片 x0:
    t = 随机选一个时间步(1, 1000)
    epsilon = 采样高斯噪声
    xt = sqrt(alpha_bar_t) * x0 + sqrt(1 - alpha_bar_t) * epsilon
    predicted_epsilon = UNet(xt, t)
    loss = MSE(epsilon, predicted_epsilon)
    反向传播更新UNet参数
```

---

5. 推理（生成）过程

训练完成后，生成新图片的流程：

```
x_T = 采样纯高斯噪声 (512x512x3)

for t = T down to 1:
    predicted_noise = UNet(x_t, t)
    x_{t-1} = 去噪公式(x_t, predicted_noise, t)

return x_0  (生成的图片)
```

从 $T = 1000$ 步走到 $t = 0$，每一步都去除一点点预测的噪声，最终得到一张清晰的图片。

---

6. DDPM vs DDIM：快与慢的权衡

DDPM（Denoising Diffusion Probabilistic Models）

- 原始扩散模型，走 1000 步

- 每一步都包含随机扰动 $\sigma_t \cdot z$
- 生成质量高，但**非常慢**（生成一张图要 1000 次神经网络推理）
- 类比：**小心翼翼地一刀一刀雕刻**，每一步都很谨慎

DDIM（Denoising Diffusion Implicit Models）

- 可以**跳步采样**，比如只走 20~50 步
- 去掉了随机扰动项，变成了**确定性**的生成过程
- 速度提升 20~50 倍，质量几乎不损失
- 类比：**有经验后可以大刀阔斧地雕刻**，每一步去掉更多

对比

特性	DDPM	DDIM
步数	1000	20~100（可配置）
速度	慢	快
质量	高	几乎一样高
确定性	否	是
实际使用	训练用	推理用

在 Stable Diffusion 的实际推理中，通常使用 DDIM 采样器，25~50 步就能生成高质量图片。

---

7. 关键数学概念速查（EE友好版）

概念	数学表达	EE类比
高斯分布	$\mathcal{N}(\mu, \sigma^2)$	最常见的噪声模型，热噪声就是高斯分布
马尔可夫链	$P(x_t \mid x_{t-1}, \ldots, x_0) = P(x_t \mid x_{t-1})$	无记忆系统：下一步只取决于当前状态
SNR（信噪比）	$\frac{\bar{\alpha}_t}{1 - \bar{\alpha}_t}$	信号功率/噪声功率，t 越大 SNR 越低

概念	数学表达	EE类比
MSE损失	$\|\|\epsilon - \hat{\epsilon}\|\|^2$	误差能量最小化
重参数化技巧	$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$	把随机采样变成可微分的确定运算

---

8. 总结

问题	答案
扩散模型的核心思想是什么？	先学会加噪，再学会去噪。用去噪来生成
为什么扩散模型能生成图像？	因为它学会了从纯噪声中逐步恢复出有意义的图像结构
前向过程难吗？	不难，就是反复加高斯噪声，而且可以一步到位算出结果
反向过程谁在做？	一个 UNet 神经网络在预测每一步的噪声
需要走 1000 步吗？	训练时需要，但推理时可以用 DDIM 跳步，20~50 步就够了
EE学生有什么优势？	高斯噪声、信噪比、马尔可夫链、MMSE估计——这些都是EE本科课程学过的

下一步

理解了扩散模型的基本原理后，我们来看 Stable Diffusion详解——看看如何在有限的计算资源下运行这个庞大的模型。 # Stable Diffusion 详解

> 适用对象：零基础EE学生
>
> 前置知识：扩散模型原理、Python基础
>
> 参考代码：`txsc/backend/pipeline.py` 和 `txsc/backend/main.py`

---

1. 为什么需要 Stable Diffusion?

在上一篇文章中，我们知道了扩散模型的工作原理：从噪声逐步去噪生成图像。

但有一个**现实问题**：原始扩散模型是在**像素空间**（比如 512×512×3 的RGB图像）直接操作的。这意味着： - 每一步去噪要处理 **786,432 个像素值**（512×512×3） - 对于 1000 步来说，计算量巨大 - 普通消费级 GPU 根本跑不动

Stable Diffusion（简称SD）的核心创新就是解决这个问题——**先用一个VAE把图像压缩到”潜空间”，在潜空间里做扩散，再解码回图像**。

一个类比：寄快递

想象你要寄一整套百科全书（原始图像）。

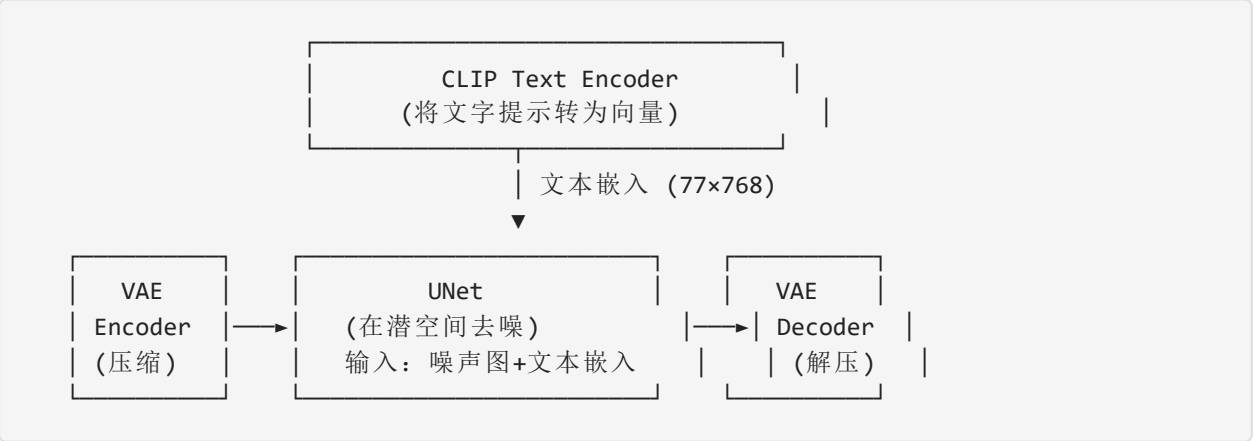
- **普通扩散模型**：直接把整套书搬过去——又重又慢
- **Stable Diffusion**：先把书扫描成电子版压缩成 ZIP（编码到潜空间），传输 ZIP 文件（潜空间扩散），收到后解压打印出来（解码回图像）

ZIP 文件比原书小得多，传输更快——这就是 SD 的核心理念。

---

2. 整体架构：三大组件

Stable Diffusion 由三个核心组件组成：



组件1：VAE（变分自编码器）—— 图像压缩/解压

角色	功能	通俗理解
VAE编码器	将 512×512×3 的图像压缩为 64×64×4 的潜空间表示	把高清照片压缩成”缩略图+细节描述”
VAE解码器	将 64×64×4 的潜空间表示恢复为 512×512×3 的图像	把”缩略图+细节描述”还原为高清照片

关键数据：压缩比约为 **48倍**（512×512×3 ÷ 64×64×4 = 48）

对于EE学生的解释：VAE类似于一个**有损编解码器**，类似于 JPEG 压缩。编码器提取图像的主要特征（扔掉不重要的细节），解码器从这些特征中重建图像。损失的信息在视觉上基本不可感知。

组件2：UNet —— 潜空间中的去噪引擎

UNet 是**真正执行去噪过程的神经网络**。它的结构特点是：

- **编码器-解码器架构**：先下采样提取特征，再上采样恢复尺寸
- **跳跃连接**：编码器的特征直接传给解码器对应层（保留细节信息）
- **交叉注意力层**：接收 CLIP 文本嵌入（让网络知道”要去掉什么噪声，保留什么内容”）

在 SD 中，UNet 的输入输出尺寸都是 **64×64×4**（潜空间大小），总参数量约 **860M**。

组件3：CLIP Text Encoder —— 理解文字提示

CLIP（Contrastive Language-Image Pre-training）是一个**文本-图像对比学习模型**。

它将用户输入的文本（如 “a cute cat”）转换为 **77×768 维的向量**，这个向量包含了文本的语义信息。

对于EE学生的解释：CLIP 类似于一个**将文字映射到语义空间的编码器**。就像傅里叶变换将时域信号变到频域一样，CLIP 将文字变到”语义域”。“猫”和”狗”在语义空间中的距离比”猫”和”汽车”更近。

在 SD 生成过程中，CLIP 的输出通过**交叉注意力机制**注入到 UNet 中，告诉 UNet：“现在要去掉的噪声应该形成一只猫的形状”。

---

3. 代码逐模块解析

现在我们来看 `pipeline.py` 中的实际代码。

3.1 设备检测：`detect_device()`

```
def detect_device() -> str:
    if torch.cuda.is_available():
        info = torch.cuda.get_device_properties(0)
        total = getattr(info, 'total_memory', getattr(info, 'total_mem', 0))
        logger.info(f"CUDA: {torch.cuda.get_device_name(0)} ({total / 1e9:.1f} GB)")
        return "cuda"
    logger.info("CUDA not available, using CPU")
    return "cpu"
```

逐行解释：

行	作用
`torch.cuda.is_available()`	检查是否有NVIDIA GPU可用（CUDA是NVIDIA的GPU计算平台）
`torch.cuda.get_device_properties(0)`	获取第一个GPU的属性（显存大小、型号等）
`total / 1e9`	将字节转换为GB（1e9 ≈ 1GB）
返回 `"cuda"` 或 `"cpu"`	告诉后面的代码在哪运行

为什么这么做？ SD模型需要大量计算。GPU比CPU快 10~50倍。如果用户有NVIDIA显卡就用GPU，否则退而求其次用CPU（虽然很慢）。

3.2 加载模型：`load_pipeline()`

```
def load_pipeline(model_id: str, device: str):
    dtype = torch.float16 if device == "cuda" else torch.float32

    logger.info(f"Loading {model_id} ({dtype})...")

    pipe = AutoPipelineForText2Image.from_pretrained(
        model_id,
        torch_dtype=dtype,
        use_safetensors=True,
        safety_checker=None,
    )

    if device == "cuda":
```

```
        pipe.enable_model_cpu_offload()
        pipe.enable_attention_slicing()
    else:
        pipe = pipe.to(device)

    logger.info("Pipeline loaded")
    return pipe
```

关键点解释：

概念	解释
`torch.float16`	半精度浮点数，显存占用减半，速度更快。只能在GPU上用
`torch.float32`	单精度浮点数，精度高但显存占用大。CPU上用这个
`AutoPipelineForText2Image`	HuggingFace diffusers 库的自动管线类，自动识别模型类型
`model_id`	字符串，可以是本地路径或HuggingFace仓库名
`use_safetensors=True`	使用更安全、更快的模型格式（相比于传统的pickle）
`safety_checker=None`	禁用安全检查器（过滤NSFW内容），节省内存
`enable_model_cpu_offload()`	**GPU显存优化技术**：不用的模块暂时放到CPU内存，用的时候再搬回GPU
`enable_attention_slicing()`	**显存优化技术**：分片计算注意力，减少单次显存占用

模型来源： `main.py` 中的 `_resolve_model()` 函数定义了加载顺序：

1. 优先使用环境变量 `SD_MODEL` 指定的路径
2. 其次检查本地 `models/AI-ModelScope/stable-diffusion-v1-5` 目录
3. 最后回退到 HuggingFace 在线仓库 `runwayml/stable-diffusion-v1-5`

3.3 核心生成：`generate()`

```
def generate(pipe, prompt, negative_prompt, steps, guidance, width, height, seed,
             callback=None):
    generator = None
    if seed is not None:
```

```
        generator = torch.Generator(device=pipe.device).manual_seed(seed)

    def _wrapped_callback(pipeline, step, timestep, callback_kwargs):
        if callback:
            callback(step, timestep)
        return callback_kwargs

    with torch.inference_mode():
        result = pipe(
            prompt=prompt,
            negative_prompt=negative_prompt,
            num_inference_steps=steps,
            guidance_scale=guidance,
            width=width,
            height=height,
            generator=generator,
            callback_on_step_end=_wrapped_callback,
        )

    return result.images[0]
```

参数详解：

参数	说明	典型值
`prompt`	正向提示词——你想要生成的内容	`"a cute cat"`
`negative_prompt`	负向提示词——你不想要的内容	`"blurry, low quality"`
`num_inference_steps`	去噪步数	20~50（用DDIM，比DDPM少得多）
`guidance_scale`	引导强度 = CFG（Classifier-Free Guidance）缩放系数	7.5
`width/height`	生成图像的宽高（必须是8的倍数）	512×512
`seed`	随机种子，固定后可复现相同结果	42

CFG（Classifier-Free Guidance）缩放：

$$\text{预测噪声} = \epsilon_\theta(x_t,\ t,\ \varnothing) + g \cdot (\epsilon_\theta(x_t,\ t,\ \text{prompt}) - \epsilon_\theta(x_t,\ t,\ \varnothing))$$

- 当 $g = 1$：完全听从文字提示

- 当 $g = 7.5$（典型值）：大幅增强文字提示的影响
- 当 $g > 15$：过度饱和，图像失真

通俗解释：CFG就像是”听话程度”。$g = 7.5$ 意味着模型在生成时会更努力地匹配你的文字描述。这在数学上等价于**将条件生成推向分布的高概率区域**。

Seed（随机种子）机制：

```
generator = torch.Generator(device=pipe.device).manual_seed(seed)
```

- 相同的 seed + 相同的 prompt = 完全相同的输出
- 不同的 seed = 不同的随机噪声初始化 = 不同的输出
- seed 控制了**初始噪声**（即起点），后续去噪路径由模型决定

对于EE学生的解释：Seed 类似于**伪随机数发生器的初始值**。相同的种子产生相同的噪声序列，因此在相同的条件下（提示词、步数、参数）能得到完全相同的生成结果。这在做实验对比时非常有用。

`torch.inference_mode()`：

```
with torch.inference_mode():
    result = pipe(...)
```

加速推理的关键上下文管理器。它会： - 禁用梯度计算（推理不需要梯度，省显存） - 禁用自动微分（Autograd）图构建 - 某些操作使用更快的实现路径

对于EE学生的解释：这就像在调试电路时关掉了不必要的测量仪器——减少负载，让系统跑得更快。

3.4 回调包装器：适配不同版本的 diffusers

```
def _wrapped_callback(pipeline, step, timestep, callback_kwargs):
    if callback:
        callback(step, timestep)
    return callback_kwargs
```

为什么要包装？

diffusers 库在不同版本中改变了回调函数的签名（函数参数形式）：

版本	回调签名
旧版 (< 0.32)	`callback(step, timestep, latents)`
新版 (>= 0.32)	`callback(pipeline, step, timestep, callback_kwargs)`

新版的回调要求**必须返回 `callback_kwargs` 字典**，且参数中包含了 `pipeline` 对象本身。包装器的职责是： 1. 接收新版回调格式的所有参数 2. 提取 `step` 和 `timestep` 传给外部回调 3. **必须返回 `callback_kwargs`**（否则管线会报错）

---

4. REST API 架构（来自 main.py）

API 端点一览

方法	路径	功能	返回
GET	`/health`	健康检查	模型状态、设备信息
POST	`/generate`	提交生成任务	`task_id`
GET	`/status/{task_id}`	查询任务进度	状态 + 进度百分比
GET	`/result/{task_id}`	获取生成结果	状态 + Base64图片

请求/响应格式

POST /generate 请求体：

```
{
  "prompt": "a cute cat",
  "negative_prompt": "blurry, low quality, distorted, bad anatomy",
  "num_inference_steps": 25,
  "guidance_scale": 7.5,
  "width": 512,
  "height": 512,
  "seed": null
}
```

GET /result/{task_id} 响应：

```
{
  "status": "done",
  "image": "...",
  "seed": null,
  "filename": "550e8400-e29b-41d4-a716-446655440000.png"
}
```

Pydantic 模型：输入验证

```
class GenerateRequest(BaseModel):
    prompt: str
    negative_prompt: str = "blurry, low quality, distorted, bad anatomy"
    num_inference_steps: int = 25
    guidance_scale: float = 7.5
    width: int = 512
    height: int = 512
    seed: int | None = None
```

Pydantic 自动处理： - 类型检查（确保 `steps` 是整数，`guidance` 是浮点数） - 默认值填充（用户可以只传 `prompt`） - 输入验证（如果有非法值，自动返回 422 错误）

---

5. 异步任务处理——为什么不能直接返回？

问题：模型推理很慢

生成一张 512×512 的图片： - GPU（RTX 3060）：约 5~10 秒 - CPU：约 2~5 分钟

如果直接等待模型生成完再返回，前端会一直转圈圈，用户体验极差。

解决方案：异步任务队列

```
POST /generate
  |
  ▼
创建 task_id = uuid4()          ← 立即返回 task_id
  |
  ▼
启动后台线程 _run_generation()   ← 在后台慢慢生成
  |
  ▼
前端每隔 1 秒 GET /status/{id}  ← 轮询进度
  |
```

```
  ▼
生成完成 → 前端 GET /result/{id} ← 获取图片
```

线程安全：`tasks_lock`

```
tasks: dict = {}          # 全局任务字典
tasks_lock = threading.Lock()  # 线程锁
```

为什么需要锁？

主线程（FastAPI请求处理线程）和后台线程会**同时读写** `tasks` 字典。如果没有锁，可能发生： - **数据竞争**：两个线程同时写同一个 key，导致数据损坏 - **读脏数据**：一个线程在写的过程中，另一个线程读到了不完整的数据

`threading.Lock` 确保了同一个时刻只有一个线程能访问 `tasks`。

对于EE学生的解释：这类似于**总线仲裁**——多个设备（CPU核心/线程）共享同一块内存（RAM），需要一个互斥机制防止同时写入导致的数据冲突。

进度回传机制

```python
def _run_generation(task_id: str, req: GenerateRequest):
    total_steps = req.num_inference_steps

    def progress_callback(step, _timestep):
        pct = int((step + 1) / total_steps * 100)
        with tasks_lock:
            if task_id in tasks:
                tasks[task_id]["progress"] = pct
                tasks[task_id]["status"] = "running"

    # ... 调用 generate，传入 callback ...
```

数据流向：

```
UNet每一步去噪完成
  ↓ (调用callback)
progress_callback(step, timestep)
  ↓ (更新字典)
tasks[task_id]["progress"] = 当前百分比
  ↓ (前端轮询)
GET /status/{id} → 返回进度
```

结果存储

生成完成后，结果以 Base64 编码的字符串存储在内存中：

```
buf = io.BytesIO()
img.save(buf, format="PNG")
b64 = base64.b64encode(buf.getvalue()).decode()

tasks[task_id] = {
    "status": "done",
    "progress": 100,
    "image": f"{b64}",
    "seed": req.seed,
    "filename": filename,
}
```

Base64 是什么？ 一种将二进制数据（图片）编码为纯文本（ASCII字符）的方法。这样图片可以直接嵌入 JSON 响应中，方便前后端传输。

---

6. 回调进度的架构问题

当前方案：同步多线程 + 轮询

```
[后端]                          [前端]
  |                               |
  | ← POST /generate              |
  | —— {task_id} ——————————→      |
  |                               |
  | ← GET /status/{id} ————→      |  ← 每隔1秒轮询
  | —— {progress: 45} ————————→  |
  | ← GET /status/{id} ————→      |
  | —— {progress: 72} ————————→  |
  | ← GET /result/{id} ————→      |
  | —— {image: base64...} ————→  |
```

问题分析

问题	说明	影响
轮询延迟	前端每隔1秒查询一次进度，最多有1秒的延迟	进度显示不够实时
无谓流量	即使进度没变，也在反复请求	浪费带宽

问题	说明	影响
内存膨胀	所有结果都存在 `tasks` 字典中，从不清理	长时间运行后内存溢出
单点故障	服务重启后所有任务丢失	生成中的任务全部作废

更优方案：WebSocket

WebSocket 是一种**全双工通信协议**，服务端可以主动推送进度给前端：

```
[后端]                          [前端]
  |                               |
  | ← WebSocket 连接建立 ————————→ |
  |                               |
  | —— {type: "progress", 45} ——→|  ← 后端主动推送
  | —— {type: "progress", 72} ——→|
  | —— {type: "result", data..}—→|  ← 完成后主动推送结果
```

优点：实时、节省流量、服务端可控

为何当前系统没用 WebSocket？ 当前系统设计以简洁为优先，使用 REST API + 轮询的方案更容易理解和实现。WebSocket 会增加部署复杂度（需要额外的连接管理）。

---

7. 整体流程总结

```
用户输入文字提示
    |
    ▼
CLIP Text Encoder 将文字转为 77×768 的嵌入向量
    |
    ▼
采样 64×64×4 的纯高斯噪声（潜空间中的"初始噪声"）
    |
    ▼
UNet 进行 25 步（DDIM）去噪：
  每一步都：
  1. 将当前潜变量 + 时间步输入 UNet
  2. 通过交叉注意力注入文字嵌入
  3. UNet 预测噪声
  4. 根据去噪公式更新潜变量
    |
    ▼
VAE Decoder 将 64×64×4 的潜变量解码为 512×512×3 的RGB图像
    |
```

▼
将图像保存为 PNG 并返回给用户

---

8. EE学生速查表

概念	解释
潜空间（Latent Space）	类似于压缩感知中的低维表示，图像经过压缩后的”精华”
VAE	类似于有损图像编解码器（JPEG），但可微分且可训练
CLIP	文本到语义向量的编码器，就像把文字”调制”到语义载波上
交叉注意力	类似于无线通信中的”波束赋形”——把文字信息定向注入到图像的特定区域
CFG（引导系数）	控制”听话程度”的增益因子，类似于反馈控制系统中的增益
Seed	伪随机数种子，确保可重复性
DDIM跳步	时空域的下采样——不是每步都做，而是大步跳
Base64编码	将二进制图片编码为文本，便于在JSON中传输

下一步

理解了 SD 的架构和代码后，接下来看 FastAPI后端详解，深入了解 Web 后端的完整设计和部署。 # FastAPI 后端详解

> 从零开始，适合没学过 Web 后端的 EE 学生

---

1. 什么是 Web 后端？

一句话理解

Web 后端就是”运行在服务器上的程序，等待别人的请求，然后返回数据”。

```
浏览器 ↔ HTTP请求/响应 ↔ 后端服务器
```

类比

后端就像餐厅的后厨：前端是服务员（展示菜单），后端是厨师（做饭），API 是菜单（规定了可以点什么）。

---

2. 我们的后端架构

```
前端 (React)                    后端 (FastAPI)
    |                               |
    | 1. POST /generate             |
    |    {prompt, steps, ...}       |
    | ----------------------------> |
    |    <-- {task_id: "abc"} ----- |
    |                               |
    | 2. GET /status/abc            |
    | ----------------------------> |
    |    <-- {progress: 30} ------  |
    |                               |
    | 3. GET /result/abc            |
    | ----------------------------> |
    |    <-- {image: "data:..."} --> 推理线程
```

为什么用异步任务？

SD 生成一张图片需要几秒。不能让用户干等，所以先返回 `task_id`，前端轮询进度。

---

3. 核心代码解析

3.1 创建应用

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI(title="AIGC Text-to-Image API")

# 跨域：允许前端 localhost:5173 访问后端 localhost:8000
```

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

3.2 模型加载（启动时）

```
@app.on_event("startup")
def startup():
    global pipe, device
    device = detect_device()
    pipe = load_pipeline(MODEL_ID, device)
```

`@app.on_event("startup")`：服务器启动时自动执行，只加载一次模型，后续请求复用。

3.3 请求格式

```
from pydantic import BaseModel

class GenerateRequest(BaseModel):
    prompt: str
    negative_prompt: str = "blurry, low quality"
    num_inference_steps: int = 25
    guidance_scale: float = 7.5
    width: int = 512
    height: int = 512
    seed: int | None = None
```

Pydantic 自动做类型检查、默认值、生成 API 文档。

3.4 提交任务 API

```
@app.post("/generate")
def generate(req: GenerateRequest):
    if pipe is None:
        raise HTTPException(status_code=503, detail="Model not loaded")
    if req.width % 8 != 0 or req.height % 8 != 0:
        raise HTTPException(status_code=400, detail="尺寸须为8的倍数")

    task_id = str(uuid.uuid4())
    with tasks_lock:
        tasks[task_id] = {"status": "pending", "progress": 0}
```

```
    thread = threading.Thread(target=_run_generation, args=(task_id, req),
         daemon=True)
    thread.start()
    return {"task_id": task_id}
```

关键：立即返回 task_id，真正推理在后台线程执行。

3.5 进度和结果查询

```
@app.get("/status/{task_id}")
def get_status(task_id: str):
    t = tasks.get(task_id)
    if t is None:
        raise HTTPException(status_code=404, detail="Task not found")
    return {"status": t["status"], "progress": t.get("progress", 0)}

@app.get("/result/{task_id}")
def get_result(task_id: str):
    t = tasks.get(task_id)
    if t is None:
        raise HTTPException(status_code=404)
    if t["status"] == "error":
        return {"status": "error", "message": t.get("error")}
    if t["status"] != "done":
        return {"status": t["status"], "progress": t.get("progress", 0)}
    return {"status": "done", "image": t["image"], "seed": t.get("seed")}
```

3.6 后台推理

```
def _run_generation(task_id, req):
    try:
        def progress_callback(step, _timestep):
            pct = int((step + 1) / req.num_inference_steps * 100)
            with tasks_lock:
                tasks[task_id]["progress"] = pct
                tasks[task_id]["status"] = "running"

        img = generate_image(pipe, req.prompt, ..., callback=progress_callback)

        # base64 编码用于前端显示
        buf = io.BytesIO()
        img.save(buf, format="PNG")
        b64 = base64.b64encode(buf.getvalue()).decode()

        with tasks_lock:
            tasks[task_id] = {"status": "done", "image": f"
         {b64}"}
```

```
    except Exception as e:
        tasks[task_id] = {"status": "error", "error": str(e)}
```

4. 关键设计

问题	方案	原因
长时间推理不阻塞	threading.Thread	后台执行，立即返回
线程安全	threading.Lock	保护共享 tasks 字典
进度反馈	轮询（500ms间隔）	比 WebSocket 简单
图片传输	base64 data URL	无需额外文件服务器

5. 启动

```
uvicorn main:app --port 8000 --reload
```

访问 `http://localhost:8000/docs` 查看自动生成的 API 文档。

下一步

看完后端后，阅读 React前端基础 了解前端与后端的交互过程。 # React 前端基础

> 从零开始，适合没学过前端开发的 EE 学生

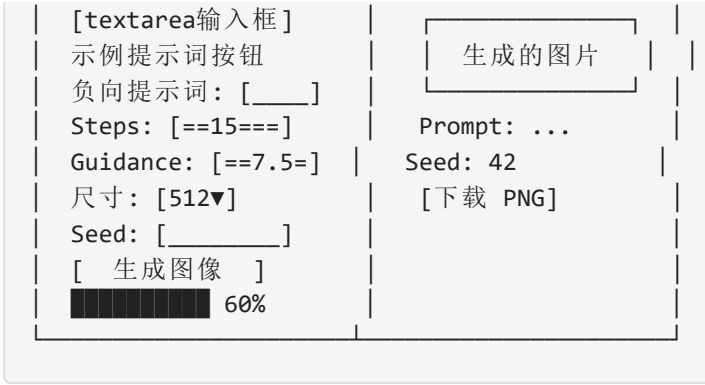
---

1. 什么是前端？

前端就是”用户直接看到的、可以点击和操作的界面”。

页面布局

```
┌─────────────────────────────────────────┐
│  AIGC 文生图系统                        │
├──────────────────┬──────────────────────┤
│  正向提示词:     │   生成结果:          │
```



左边控制面板（参数输入），右边结果展示（生成的图片）。

---

2. React 核心概念

2.1 组件

页面由组件拼装，就像乐高：

```
App → Header + ImageGenerator + ImageDisplay
```

组件就是一个返回 HTML 的函数：

```
function Header() {
  return <h1>AIGC 文生图系统</h1>;
}
```

2.2 状态（State）

状态是”会变化的数据”。

```
import { useState } from "react";

const [generating, setGenerating] = useState(false);
// generating = false（初始值）
// setGenerating(true) → 界面自动更新
```

2.3 属性（Props）

父组件传给子组件的数据：

```
// 父组件传递
<ImageDisplay image={result.image} seed={result.seed} />

// 子组件接收
function ImageDisplay({ image, seed }) {
  return <img src={image} />;
}
```

---

3. App.tsx 主组件

```
export default function App() {
  const [result, setResult] = useState(null);

  return (
    <div>
      <header><h1>AIGC 文生图系统</h1></header>
      <main>
        <ImageGenerator
          onGenerated={(image, seed, filename, prompt) => {
            setResult({ image, seed, filename });
          }}
        />
        {result && <ImageDisplay image={result.image} seed={result.seed}
                                 filename={result.filename} />}
      </main>
    </div>
  );
}
```

`result && (...)`：只有 result 不为空时才显示右侧面板。

---

4. ImageGenerator 与控制参数

所有参数的状态

```
const [prompt, setPrompt] = useState("");
const [negativePrompt, setNegativePrompt] = useState("");
const [steps, setSteps] = useState(15);
const [guidance, setGuidance] = useState(7.5);
const [size, setSize] = useState(512);
const [seed, setSeed] = useState("");
```

```
const [generating, setGenerating] = useState(false);
const [progress, setProgress] = useState(0);
```

调用后端

```
async function handleGenerate() {
  setGenerating(true);

  // 1. 提交任务
  const taskId = await startGeneration({
    prompt, negative_prompt: negativePrompt,
    num_inference_steps: steps, guidance_scale: guidance,
    width: size, height: size,
    seed: seed ? parseInt(seed) : null,
  });

  // 2. 每500ms轮询进度
  timerRef.current = setInterval(async () => {
    const status = await getStatus(taskId);
    setProgress(status.progress);

    if (status.status === "done") {
      clearInterval(timerRef.current);
      const result = await getResult(taskId);
      onGenerated(result.image, result.seed, result.filename, prompt);
      setGenerating(false);
    }
  }, 500);
}
```

进度条

```
{generating && (
  <div>
    <span>生成中... {progress}%</span>
    <div className="bg-blue-500" style={{ width: `${progress}%` }} />
  </div>
)}
```

---

5. API 客户端（client.ts）

```
const BASE = "http://localhost:8000";

export async function startGeneration(params) {
```

```
  const res = await fetch(`${BASE}/generate`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(params),
  });
  const data = await res.json();
  return data.task_id;
}

export async function getStatus(taskId) {
  const res = await fetch(`${BASE}/status/${taskId}`);
  return res.json();
}

export async function getResult(taskId) {
  const res = await fetch(`${BASE}/result/${taskId}`);
  return res.json();
}
```

完整交互流程：

```
用户点击生成 → POST /generate → task_id
    → setInterval 500ms → GET /status → 更新进度条
    → status=done → GET /result → 显示图片
```

---

6. 前端技术栈

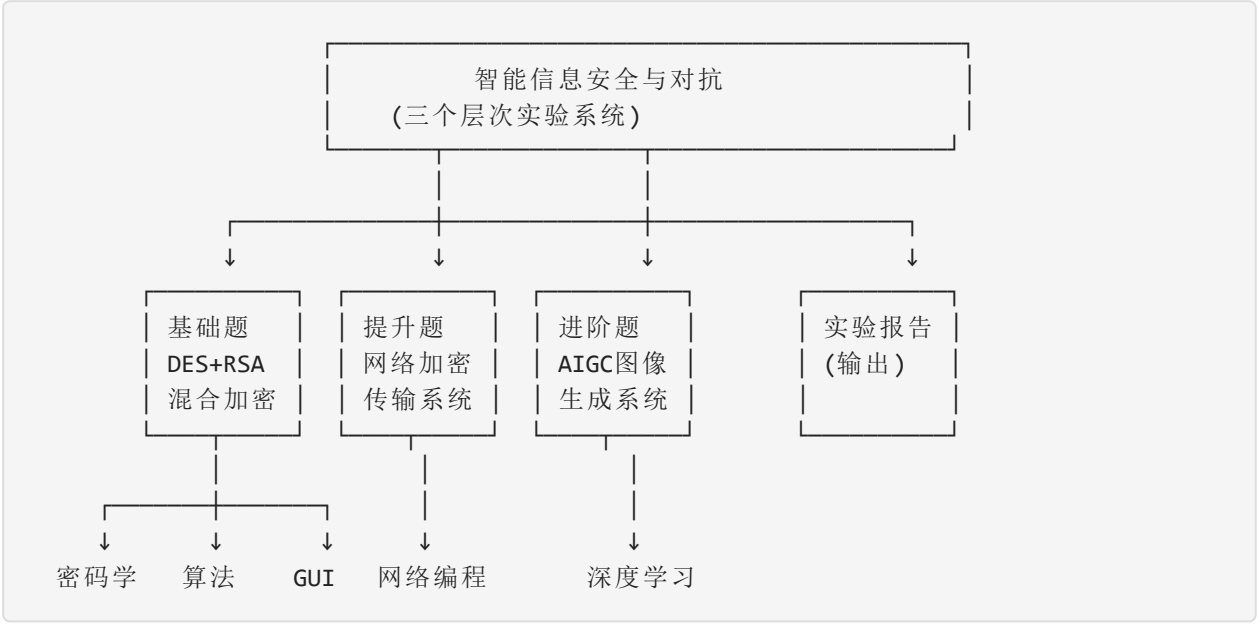
技术	作用
React 19	UI 框架（组件化）
TypeScript	类型安全的 JavaScript
Vite	开发服务器 + 构建工具
Tailwind CSS	样式（`className="..."` 直接写）
lucide-react	图标库

下一步

现在你已经完整理解了从后端到前端的 AIGC 系统，可以回头看 SD详解 中关于模型推理的部分。 # 知识地图

> 本图展示本项目的所有知识点及其关联关系，帮助你理解”学了什么”以及”它们之间有什么关系”。

一、宏观知识图谱



二、难度与前置知识速查

🔐 密码学

知识点	难度	前置知识	涉及文件
二进制/字节	⭐	无	所有文件
异或运算XOR	⭐	二进制	DES核心
模运算	⭐	除法	RSA核心
对称加密	⭐⭐	XOR	`des_core.py`
非对称加密	⭐⭐⭐	模运算	`rsa_core.py`
DES算法	⭐⭐⭐	对称加密	`des_core.py`
三重DES	⭐⭐	DES	`des_core.py`

知识点	难度	前置知识	涉及文件
PKCS7填充	⭐⭐	对称加密	`des_core.py`
RSA算法	⭐⭐⭐⭐	非对称加密	`rsa_core.py`
PKCS1_OAEP	⭐⭐⭐	RSA	`rsa_core.py`
PEM格式	⭐	无	`rsa_core.py`
混合加密	⭐⭐⭐	DES+RSA	`hybrid_crypto.py`

🌐 网络编程

知识点	难度	前置知识	涉及文件
IP/端口	⭐	无	网络传输所有文件
TCP协议	⭐⭐	IP/端口	`network_transfer.py`
Socket编程	⭐⭐⭐	TCP	`network_server.py`
C/S架构	⭐⭐	Socket	所有网络文件
自定义协议	⭐⭐⭐⭐	Socket	`network_protocol.py`

🧠 深度学习

知识点	难度	前置知识	涉及文件
神经网络	⭐⭐⭐	线性代数	—
CNN卷积	⭐⭐⭐⭐	神经网络	SD的UNet
扩散模型	⭐⭐⭐⭐⭐	生成模型	`pipeline.py`
Stable Diffusion	⭐⭐⭐⭐⭐	扩散模型	`txsc/backend/`
VAE/UNet/CLIP	⭐⭐⭐⭐	CNN	SD各组件

🖥️ Web开发

知识点	难度	前置知识	涉及文件
REST API	⭐⭐⭐	HTTP	`main.py`

知识点	难度	前置知识	涉及文件
FastAPI	⭐⭐⭐	Python	`main.py`
React	⭐⭐⭐⭐	JavaScript	`frontend/src/`
TypeScript	⭐⭐⭐	JavaScript	`frontend/*.tsx`

三、学习路径

```
零基础
  │
  ├── 密码学基础 → DES → RSA → 混合加密 (基础题)
  ├── 网络基础 → TCP → 自定义协议 → C/S (提升题)
  ├── 深度学习 → 扩散模型 → SD详解 (进阶题)
  └── FastAPI → React → Web部署 (进阶题)
```

四、知识点与文件映射

```
├── des_core.py              → DES算法，对称加密，密钥派生
├── rsa_core.py              → RSA算法，非对称加密，PEM
├── hybrid_crypto.py         → 混合加密，文件打包格式
├── gui.py                   → Tkinter GUI，事件驱动
├── test_crypto.py           → 单元测试 (19项)

├── 提升题/
│   ├── network_protocol.py  → 自定义协议，消息帧
│   ├── network_transfer.py  → TCP分块传输，ACK确认
│   ├── network_server.py    → 服务端GUI，接收逻辑
│   ├── network_client.py    → 客户端GUI，发送逻辑
│   ├── network_image_utils.py → 图像处理
│   └── test_network.py      → 网络测试 (22项)

└── txsc/
    ├── backend/main.py      → FastAPI，REST API
    ├── backend/pipeline.py  → SD管线，GPU加速
    └── frontend/src/        → React组件，API客户端
```

五、学习顺序建议

阶段	内容	预计时间	目标
1	密码学入门	1-2天	理解基础题全部代码
2	网络编程	1天	理解提升题全部代码
3	AIGC原理	2-3天	理解进阶题工作原理
4	综合实践	0.5天	能修改和扩展代码