# python基础-requests结合AI实现自动化数据抓取

## 概述

Requests 是一个基于 `urllib3` 封装的 Python HTTP 客户端库，提供了极其简洁且人性化的接口，使得发送 HTTP 请求和处理响应变得轻而易举。它支持常见的 HTTP 方法（GET、POST、PUT、DELETE 等）、会话保持、文件上传、代理、超时控制、认证等功能，被广泛应用于网页抓取、API 调用和自动化测试等场景。本指南从基础概念出发，逐步深入到高级应用场景，适合各层次开发者阅读。

## 1. 安装与环境配置

### 1.1 标准安装

```
# 基本安装
pip install requests

# 指定版本安装
pip install requests==2.28.1

# 使用国内镜像加速
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple requests

# 安装带有附加功能的完整版本
pip install requests[security]  # 包含证书验证所需的依赖
```

### 1.2 安装开发版本

```
# 从 GitHub 安装最新开发版本
pip install git+https://github.com/psf/requests.git

# 安装特定分支的开发版本
pip install git+https://github.com/psf/requests.git@branch_name
```

### 1.3 导入与版本验证

```
import requests
```

```
print(requests.__version__)  # 检查版本
print(requests.adapters.DEFAULT_POOLSIZE)  # 默认连接池大小
```

# 2. 基础请求方法与响应处理

## 2.1 HTTP 方法全集

```python
# 完整 HTTP 方法支持
response = requests.get('https://httpbin.org/get')
response = requests.post('https://httpbin.org/post', data={'key': 'value'})
response = requests.put('https://httpbin.org/put', data={'key': 'value'})
response = requests.delete('https://httpbin.org/delete')
response = requests.head('https://httpbin.org/get')  # 只返回响应头
response = requests.options('https://httpbin.org/get')  # 查询服务器支持的方法
response = requests.patch('https://httpbin.org/patch', data={'key': 'value'})
```

## 2.2 高效响应对象处理

```python
r = requests.get('https://api.github.com/events')

# 响应状态与头信息
print(r.status_code)  # 状态码
print(r.reason)       # 状态文本
print(r.headers)      # 响应头字典
print(r.headers['Content-Type'])  # 获取特定响应头
print(r.url)          # 最终响应 URL（可能经过重定向）
print(r.history)      # 重定向历史
print(r.elapsed)      # 请求耗时 (timedelta 对象)
print(r.request)      # 原始请求对象

# 响应内容获取与解析
print(r.encoding)     # 编码方式
r.encoding = 'utf-8'  # 手动设置编码（覆盖自动检测）
print(r.text)         # 文本形式的响应内容
print(r.content)      # 二进制形式的响应内容
print(r.json())       # 解析 JSON 响应

# 高效的迭代流式响应
for line in r.iter_lines():
    if line:
        print(line.decode('utf-8'))
```

## 2.3 HTTP 内容协商

```python
# 指定接受的响应格式
headers = {'Accept': 'application/json'}
r = requests.get('https://api.example.com/users', headers=headers)

# 指定接受的编码方式
headers = {'Accept-Encoding': 'gzip, deflate, br'}
r = requests.get('https://example.com', headers=headers)

# 指定接受的语言
headers = {'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8'}
r = requests.get('https://example.com', headers=headers)
```

## 3. 高级参数传递与数据处理

### 3.1 复杂查询参数

```python
# 嵌套字典转换为查询参数
params = {
    'search': 'python requests',
    'filters': {
        'language': 'python',
        'license': 'MIT'
    },
    'tags[]': ['http', 'client']  # 数组参数
}
r = requests.get('https://api.example.com/search', params=params)
print(r.url)  # 查看最终构建的 URL
```

### 3.2 多格式数据提交

```python
# Form 表单数据
form_data = {'username': 'user1', 'password': 'pass123'}
r = requests.post('https://httpbin.org/post', data=form_data)

# URL 编码的表单数据（手动控制）
from urllib.parse import urlencode
encoded_data = urlencode({'key': 'value with spaces'})
headers = {'Content-Type': 'application/x-www-form-urlencoded'}
r = requests.post('https://httpbin.org/post', data=encoded_data, headers=headers)

# JSON 数据
json_data = {'name': 'John', 'age': 30, 'city': 'New York'}
r = requests.post('https://httpbin.org/post', json=json_data)  # 自动设置 Content-Type
# 或手动编码 JSON
import json
r = requests.post(
    'https://httpbin.org/post',
    data=json.dumps(json_data),
    headers={'Content-Type': 'application/json'}
)

# 复杂表单数据（含文件和字段混合）
files = {
    'file': ('report.pdf', open('report.pdf', 'rb'), 'application/pdf'),
    'file2': ('data.csv', open('data.csv', 'rb'), 'text/csv'),
    'field': (None, 'value'),  # 普通字段
    'field2': (None, json.dumps({'a': 1}), 'application/json')  # JSON 字段
}
r = requests.post('https://httpbin.org/post', files=files)
```

### 3.3 动态请求头构建

```python
import random
import time

def get_headers():
    # 常见浏览器 User-Agent 列表
    user_agents = [
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36
        'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/14.1.1 Safari/605.
        # 更多 UA 字符串...
    ]

    # 构建请求头
    headers = {
        'User-Agent': random.choice(user_agents),
        'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
        'Accept-Language': 'en-US,en;q=0.5',
        'Accept-Encoding': 'gzip, deflate, br',
        'DNT': '1',  # Do Not Track
        'Connection': 'keep-alive',
        'Upgrade-Insecure-Requests': '1',
        'Cache-Control': 'max-age=0',
        'TE': 'Trailers',
        'X-Request-ID': f'{random.randint(1000000, 9999999)}-{int(time.time())}'
    }
    return headers

r = requests.get('https://httpbin.org/headers', headers=get_headers())
```

## 4. 强大的 Session 对象与上下文管理

### 4.1 会话持久化与自定义

```python
# 创建自定义会话
session = requests.Session()
```

```python
# 会话级别的参数设置
session.headers.update({'User-Agent': 'MyApp/1.0.0'})
session.auth = ('user', 'pass')
session.proxies = {'http': 'http://proxy.example.com:8080'}
session.verify = '/path/to/certfile'  # HTTPS 证书验证
session.cert = ('/path/to/client.cert', '/path/to/client.key')  # 客户端证书
session.cookies.set('session_cookie', 'value')  # 预设 Cookie

# 会话级默认参数与请求级参数合并
r = session.get('https://httpbin.org/cookies',
                params={'q': 'test'},  # 这个参数仅用于此次请求
                timeout=5)  # 这个超时设置仅用于此次请求
```

## 4.2 会话上下文管理

```python
# 使用上下文管理器自动关闭会话
with requests.Session() as s:
    s.get('https://httpbin.org/cookies/set/sessioncookie/123456789')
    r = s.get('https://httpbin.org/cookies')
    print(r.json())  # 自动携带之前设置的 Cookie
# 会话自动关闭，释放连接池资源
```

## 4.3 高级会话定制

```python
# 自定义适配器配置
from requests.adapters import HTTPAdapter

class TimeoutHTTPAdapter(HTTPAdapter):
    def __init__(self, *args, **kwargs):
        self.timeout = kwargs.pop('timeout', 5.0)
        super().__init__(*args, **kwargs)

    def send(self, request, **kwargs):
        kwargs['timeout'] = kwargs.get('timeout', self.timeout)
        return super().send(request, **kwargs)

# 使用自定义适配器
session = requests.Session()
adapter = TimeoutHTTPAdapter(timeout=10.0, max_retries=3)
session.mount('http://', adapter)
session.mount('https://', adapter)

# 所有请求将默认使用 10 秒超时和 3 次重试
r = session.get('https://httpbin.org/delay/9')  # 应该成功完成
```

## 5. 高级文件操作

## 5.1 大文件分块上传

```python
# 采用分块传输编码（chunked transfer encoding）上传大文件
def file_generator(file_path, chunk_size=8192):
    with open(file_path, 'rb') as f:
        while True:
            chunk = f.read(chunk_size)
            if not chunk:
                break
            yield chunk

# 使用生成器实现流式上传
file_path = 'large_video.mp4'
headers = {'Transfer-Encoding': 'chunked'}
r = requests.post(
    'https://httpbin.org/post',
    data=file_generator(file_path),
    headers=headers
)
```

## 5.2 并发多文件下载

```python
import concurrent.futures
import os

def download_file(url, save_path, session=None):
    """下载单个文件并保存到指定路径"""
    s = session or requests.Session()
    r = s.get(url, stream=True)
    r.raise_for_status()

    # 获取文件总大小
    total_size = int(r.headers.get('content-length', 0))

    # 创建目录（如果不存在）
    os.makedirs(os.path.dirname(save_path), exist_ok=True)

    # 下载文件
    downloaded = 0
    with open(save_path, 'wb') as f:
        for chunk in r.iter_content(chunk_size=8192):
            downloaded += len(chunk)
            f.write(chunk)
            progress = downloaded / total_size if total_size > 0 else 0
            print(f"\r{save_path}: {progress:.2%}", end="", flush=True)
    print()  # 打印换行
    return save_path

# 并发下载多个文件
files_to_download = [
    ('https://example.com/file1.zip', 'downloads/file1.zip'),
    ('https://example.com/file2.pdf', 'downloads/file2.pdf'),
    # 更多文件...
]
```

```python
# 使用线程池执行并发下载
with requests.Session() as session:
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        futures = [
            executor.submit(download_file, url, path, session)
            for url, path in files_to_download
        ]

        # 获取结果（可能会抛出异常）
        for future in concurrent.futures.as_completed(futures):
            try:
                path = future.result()
                print(f"Successfully downloaded: {path}")
            except Exception as e:
                print(f"Download failed: {e}")
```

## 5.3 断点续传实现

```python
import os

def resume_download(url, save_path):
    """支持断点续传的文件下载函数"""
    headers = {}
    downloaded_size = 0

    # 检查文件是否已经存在（断点续传）
    if os.path.exists(save_path):
        downloaded_size = os.path.getsize(save_path)
        headers['Range'] = f'bytes={downloaded_size}-'
        print(f"Resuming from {downloaded_size} bytes")

    # 创建请求
    r = requests.get(url, headers=headers, stream=True)

    # 如果是断点续传，服务器应返回 206 Partial Content
    if downloaded_size > 0 and r.status_code == 206:
        mode = 'ab'  # 追加模式
    else:
        mode = 'wb'  # 覆盖模式
        downloaded_size = 0

    # 获取总文件大小
    total_size = int(r.headers.get('content-length', 0)) + downloaded_size

    # 下载文件
    with open(save_path, mode) as f:
        for chunk in r.iter_content(chunk_size=8192):
            if chunk:
                f.write(chunk)
                downloaded_size += len(chunk)
                progress = downloaded_size / total_size if total_size > 0 else 0
                print(f"\rProgress: {progress:.2%} of {total_size} bytes", end="")
    print("\nDownload complete!")

# 使用示例
resume_download('https://example.com/large.zip', 'large.zip')
```

# 6. 全面的错误处理与网络弹性

## 6.1 异常分类与处理

```python
from requests.exceptions import (
    RequestException,  # 所有异常的基类
    ConnectionError,   # 连接错误
    HTTPError,         # HTTP 错误状态码
    Timeout,           # 请求超时
    TooManyRedirects,  # 重定向过多
    URLRequired,       # URL 缺失
    InvalidURL,        # URL 无效
    InvalidHeader,     # 请求头无效
    InvalidSchema,     # 协议无效
    MissingSchema,     # 协议缺失
    ChunkedEncodingError,  # 分块编码错误
    ContentDecodingError,   # 内容解码错误
    StreamConsumedError,    # 流内容已消费错误
    RetryError,             # 重试失败
    UnrewindableBodyError,  # 请求体无法回退
    FileModeWarning,        # 文件模式警告
    ConnectTimeout,         # 连接超时
    ReadTimeout,            # 读取超时
)

def safe_request(method, url, **kwargs):
    """安全的请求包装函数，处理所有可能的异常"""
    try:
        response = requests.request(method, url, **kwargs)
        response.raise_for_status()  # 抛出 HTTP 错误
        return response
    except ConnectionError as e:
        print(f"网络连接错误：{e}")
    except Timeout as e:
        print(f"请求超时：{e}")
    except TooManyRedirects as e:
        print(f"重定向过多：{e}")
    except HTTPError as e:
        print(f"HTTP 错误：{e}")
        if hasattr(e, 'response') and e.response is not None:
            print(f"状态码：{e.response.status_code}")
            print(f"响应内容：{e.response.text[:200]}...")  # 显示前 200 个字符
    except RequestException as e:
        print(f"请求异常：{e}")
    return None

# 使用示例
response = safe_request('GET', 'https://httpbin.org/status/500')
if response:
    print("请求成功")
```

```
else:
    print("请求失败")
```

## 6.2 高级重试策略

```
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry
import time
import random

# 自定义退避策略
class CustomRetry(Retry):
    def get_backoff_time(self):
        """自定义退避时间计算"""
        # 指数退避 + 抖动
        backoff = min(self.BACKOFF_MAX, self.backoff_factor * (2 ** self.total_retries))
        # 增加随机抖动 (jitter) 防止请求风暴
        jitter = random.uniform(0, 0.1 * backoff)
        return backoff + jitter

    def increment(self, method=None, url=None, response=None, error=None, _pool=None, _stacktrace=None):
        # 在重试前执行自定义操作
        if response and 500 <= response.status <= 599:
            print(f"服务器错误（{response.status}），准备重试...")
        return super().increment(method, url, response, error, _pool, _stacktrace)

# 配置高级重试策略
retry_strategy = CustomRetry(
    total=5,  # 最大重试次数
    backoff_factor=0.5,  # 退避系数
    status_forcelist=[429, 500, 502, 503, 504],  # 触发重试的状态码
    allowed_methods=["HEAD", "GET", "PUT", "DELETE", "OPTIONS", "TRACE"],  # 允许重试的方法
    raise_on_status=True,  # 最终仍失败时抛出异常
    # 自定义重试条件
    retry_on_exception=lambda exc: isinstance(exc, (Timeout, ConnectionError)),
)

# 应用重试策略
adapter = HTTPAdapter(max_retries=retry_strategy, pool_connections=10, pool_maxsize=20)
session = requests.Session()
session.mount("http://", adapter)
session.mount("https://", adapter)

# 使用示例
try:
    response = session.get("https://httpbin.org/status/503", timeout=(3.05, 27))
    print(f"成功获得响应：{response.status_code}")
except RequestException as e:
    print(f"所有重试失败：{e}")
```

## 6.3 中断与恢复机制

```python
def resumable_request(url, method='GET', max_retries=3, **kwargs):
    """
    可中断恢复的请求函数，支持网络中断后自动恢复
    """
    retry_count = 0
    last_exception = None
    response = None

    # 获取已下载内容的位置（如果有）
    if 'headers' not in kwargs:
        kwargs['headers'] = {}

    # 支持流式请求
    stream = kwargs.get('stream', False)

    while retry_count < max_retries:
        try:
            if response is not None and stream:
                # 如果是流式请求且之前有进度，添加断点续传头
                downloaded = sum(len(chunk) for chunk in response.iter_content(chunk_size=1))
                if downloaded > 0:
                    kwargs['headers']['Range'] = f'bytes={downloaded}-'

            response = requests.request(method, url, **kwargs)

            # 成功获取响应
            if response.status_code in [200, 206]:  # OK 或 Partial Content
                return response

            # 适当处理其他状态码
            response.raise_for_status()

        except (ConnectionError, Timeout) as e:
            print(f"连接中断: {e}, 尝试恢复...")
            retry_count += 1
            last_exception = e
            time.sleep(2 ** retry_count)  # 指数退避
        except HTTPError as e:
            # 如果是客户端错误，不再重试
            if 400 <= e.response.status_code < 500:
                print(f"客户端错误: {e.response.status_code}")
                raise
            # 服务器错误则重试
            retry_count += 1
            last_exception = e
            time.sleep(2 ** retry_count)

    # 超过最大重试次数
    if last_exception:
        raise last_exception
    return None
```

## 7. 高级认证与安全

## 7.1 多种认证方式

```python
# 基本认证（Basic Auth）
from requests.auth import HTTPBasicAuth
r = requests.get('https://api.example.com/resource', auth=HTTPBasicAuth('user', 'pass'))
# 简写形式
r = requests.get('https://api.example.com/resource', auth=('user', 'pass'))

# 摘要认证（Digest Auth）
from requests.auth import HTTPDigestAuth
r = requests.get('https://api.example.com/resource', auth=HTTPDigestAuth('user', 'pass'))

# OAuth1 认证
from requests_oauthlib import OAuth1
auth = OAuth1(
    'YOUR_APP_KEY',
    'YOUR_APP_SECRET',
    'USER_OAUTH_TOKEN',
    'USER_OAUTH_TOKEN_SECRET'
)
r = requests.get('https://api.twitter.com/1.1/account/verify_credentials.json', auth=auth)

# OAuth2 认证
from requests_oauthlib import OAuth2Session
from oauthlib.oauth2 import BackendApplicationClient

# 客户端凭证模式
client = BackendApplicationClient(client_id='YOUR_CLIENT_ID')
oauth = OAuth2Session(client=client)
token = oauth.fetch_token(
    token_url='https://provider.com/oauth2/token',
    client_id='YOUR_CLIENT_ID',
    client_secret='YOUR_CLIENT_SECRET'
)
r = oauth.get('https://api.example.com/resource')

# 自定义认证
class TokenAuth(requests.auth.AuthBase):
    """自定义令牌认证类"""
    def __init__(self, token):
        self.token = token

    def __call__(self, r):
        r.headers['Authorization'] = f'Bearer {self.token}'
        return r

# 使用自定义认证
r = requests.get('https://api.example.com/protected', auth=TokenAuth('YOUR_TOKEN'))
```

## 7.2 高级 SSL/TLS 配置

```python
import ssl
import certifi

# 使用系统证书库
```

```
r = requests.get('https://example.com', verify=True)  # 默认

# 使用 certifi 证书库
r = requests.get('https://example.com', verify=certifi.where())

# 使用自定义证书文件
r = requests.get('https://example.com', verify='/path/to/ca-bundle.pem')

# 禁用证书验证（不推荐）
r = requests.get('https://example.com', verify=False)

# 设置客户端证书
r = requests.get('https://example.com', cert=('/path/client.crt', '/path/client.key'))

# 配置 SSL 上下文（高级用法）
import urllib3
https = urllib3.PoolManager(
    ssl_version=ssl.PROTOCOL_TLSv1_2,  # 强制使用 TLS 1.2
    cert_reqs=ssl.CERT_REQUIRED,  # 要求证书
    ca_certs=certifi.where(),  # 证书包路径
    ciphers='HIGH:!DH:!aNULL'  # 自定义密码套件
)

# 自定义 SSL 配置的会话
session = requests.Session()
adapter = requests.adapters.HTTPAdapter(pool_connections=100, pool_maxsize=100)
session.mount('https://', adapter)
# urllib3 已在内部使用，可通过 HTTPAdapter 扩展配置
```

## 7.3 安全标头与 CSRF 防护

```
# 添加安全相关的 HTTP 头
security_headers = {
    'X-Content-Type-Options': 'nosniff',
    'X-Frame-Options': 'DENY',
    'X-XSS-Protection': '1; mode=block',
    'Strict-Transport-Security': 'max-age=31536000; includeSubDomains',
    'Content-Security-Policy': "default-src 'self'",
    'Referrer-Policy': 'strict-origin-when-cross-origin'
}

session = requests.Session()
session.headers.update(security_headers)

# CSRF 令牌处理
def fetch_with_csrf(url, session=None):
    """处理 CSRF 保护的表单提交"""
    s = session or requests.Session()

    # 1. 获取包含 CSRF 令牌的页面
    response = s.get(url)

    # 2. 从响应中提取 CSRF 令牌（示例使用简单正则，实际应使用 HTML 解析）
    import re
    csrf_token = re.search(r'name="csrf_token" value="(.+?)"', response.text)
    if csrf_token:
        csrf_token = csrf_token.group(1)
    else:
        raise ValueError("无法找到 CSRF 令牌")

    # 3. 提交表单, 包含 CSRF 令牌
    data = {
        'username': 'user',
        'password': 'pass',
        'csrf_token': csrf_token
    }
    response = s.post(url, data=data)
    return response

# 使用示例
with requests.Session() as s:
    response = fetch_with_csrf('https://example.com/login', s)
```

# 8. 高性能与并发

## 8.1 高级连接池配置

```
from urllib3 import PoolManager, Retry
import socket

# 创建自定义连接管理器
http = PoolManager(
    num_pools=50,           # 连接池数量
    maxsize=100,            # 每个连接池的最大连接数
    block=False,            # 连接池满时不阻塞
    retries=Retry(3),       # 重试策略
    timeout=5.0,            # 超时设置
    socket_options=[
        (socket.IPPROTO_TCP, socket.TCP_NODELAY, 1),  # 禁用 Nagle 算法
        (socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1),  # 启用 keepalive
        (socket.IPPROTO_TCP, socket.TCP_KEEPIDLE, 60) # keepalive 空闲时间
    ]
)

# 将自定义连接管理器与 Requests 集成
session = requests.Session()
adapter = requests.adapters.HTTPAdapter(
    pool_connections=50,    # 连接池数量
    pool_maxsize=100,       # 每个连接池的最大连接数
    max_retries=Retry(
        total=3,
        backoff_factor=0.5,
        status_forcelist=[500, 502, 503, 504]
    )
)
session.mount('http://', adapter)
session.mount('https://', adapter)
```

## 8.2 异步请求与并发控制

```python
import concurrent.futures
import time
from functools import partial

def fetch_url(url, session, timeout=10):
    """发送单个请求并计时"""
    start = time.time()
    try:
        with session.get(url, timeout=timeout) as response:
            data = response.text
            return {
                'url': url,
                'data': data[:30] + '...' if data else None,  # 截断显示
                'status': response.status_code,
                'time': time.time() - start
            }
    except Exception as e:
        return {'url': url, 'error': str(e), 'time': time.time() - start}

def fetch_all(urls, max_workers=10, timeout=10):
    """并发获取多个 URL"""
    with requests.Session() as session:
        # 设置会话参数
        session.headers.update({'User-Agent': 'Mozilla/5.0'})

        # 创建部分应用函数，固定会话参数
        func = partial(fetch_url, session=session, timeout=timeout)

        # 使用线程池并发执行
        with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
            # 提交所有任务
            future_to_url = {executor.submit(func, url): url for url in urls}

            # 收集结果
            results = []
            for future in concurrent.futures.as_completed(future_to_url):
                results.append(future.result())

        return results

# 使用示例
urls = [
    'https://httpbin.org/delay/1',
    'https://httpbin.org/delay/2',
    'https://httpbin.org/delay/3',
    # 更多 URL...
]

results = fetch_all(urls, max_workers=5)
for result in results:
    print(f"{result.get('url')}: {result.get('status', 'ERROR')} in {result.get('time'):.2f}s")
```

## 8.3 限流与节流

```python
import time
from functools import wraps

class RateLimiter:
    """请求频率限制器"""
    def __init__(self, calls_limit, time_period):
        """
        初始化限流器
        :param calls_limit: 时间段内允许的最大请求数
        :param time_period: 时间段长度（秒）
        """
        self.calls_limit = calls_limit
        self.time_period = time_period
        self.timestamps = []

    def __call__(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # 检查是否超出频率限制
            now = time.time()
            # 清理过期的时间戳
            self.timestamps = [t for t in self.timestamps if now - t < self.time_period]

            # 检查是否超出限制
            if len(self.timestamps) >= self.calls_limit:
                # 计算需要等待的时间
                oldest_call = min(self.timestamps)
                sleep_time = self.time_period - (now - oldest_call)
                if sleep_time > 0:
                    print(f"频率限制: 等待 {sleep_time:.2f} 秒")
                    time.sleep(sleep_time)

            # 记录本次调用时间
            self.timestamps.append(time.time())
            return func(*args, **kwargs)
        return wrapper

# 使用示例：限制每分钟最多 30 个请求
@RateLimiter(calls_limit=30, time_period=60)
def limited_request(url):
    return requests.get(url)

# 请求节流示例
def throttled_requests(urls, requests_per_second=5):
    """按指定频率发送请求"""
    delay = 1.0 / requests_per_second
    results = []

    for url in urls:
        start = time.time()
        response = requests.get(url)
        results.append(response)

        # 计算需要等待的时间
        process_time = time.time() - start
        wait_time = delay - process_time
        if wait_time > 0:
            time.sleep(wait_time)

    return results
```

# 9. 高级调试与日志

## 9.1 开启 HTTP 请求日志

```python
import logging
from http.client import HTTPConnection

# 设置 HTTP 连接的调试级别
HTTPConnection.debuglevel = 1

# 配置日志
logging.basicConfig()
logging.getLogger().setLevel(logging.DEBUG)
requests_log = logging.getLogger("urllib3")
requests_log.setLevel(logging.DEBUG)
requests_log.propagate = True

# 发送请求，将显示详细的 HTTP 请求和响应信息
r = requests.get('https://httpbin.org/get')
```

## 9.2 自定义请求生命周期钩子

```python
import logging
from datetime import datetime

# 创建自定义日志处理器
logger = logging.getLogger("requests_hooks")
logger.setLevel(logging.INFO)
handler = logging.StreamHandler()
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)

# 定义请求生命周期钩子函数
def log_request(r, *args, **kwargs):
    logger.info(f"请求开始: {r.method} {r.url}")
    r.start_time = datetime.now()
    return r

def log_response(r, *args, **kwargs):
    elapsed = datetime.now() - r.start_time
    logger.info(f"请求完成: {r.status_code} {r.reason} - 耗时 {elapsed.total_seconds():.2f}s")
    logger.debug(f"响应头: {dict(r.headers)}")
    return r

# 创建会话并添加钩子
session = requests.Session()
session.hooks = {
    'response': log_response,
    'request': log_request
}

# 测试请求
response = session.get('https://httpbin.org/get')
```

## 9.3 高级请求分析工具

```python
from io import BytesIO
import gzip

class RequestAnalyzer:
    """HTTP 请求分析器"""

    def __init__(self, session=None):
        self.session = session or requests.Session()
        # 添加响应拦截钩子
        self.session.hooks['response'].append(self._analyze_response)
        self.stats = {
            'total_requests': 0,
            'successful_requests': 0,
            'failed_requests': 0,
            'total_bytes': 0,
            'compressed_bytes': 0,
            'response_times': [],
            'status_codes': {},
            'content_types': {}
        }

    def _analyze_response(self, response, *args, **kwargs):
        """分析响应对象并收集统计信息"""
        self.stats['total_requests'] += 1

        # 状态码统计
        status = response.status_code
        self.stats['status_codes'][status] = self.stats['status_codes'].get(status, 0) + 1

        if 200 <= status < 400:
            self.stats['successful_requests'] += 1
        else:
            self.stats['failed_requests'] += 1

        # 响应时间
        elapsed = response.elapsed.total_seconds()
        self.stats['response_times'].append(elapsed)

        # 内容类型
        content_type = response.headers.get('Content-Type', 'unknown').split(';')[0]
        self.stats['content_types'][content_type] = self.stats['content_types'].get(content_type, 0) + 1

        # 传输大小分析
        if response.headers.get('Content-Encoding') == 'gzip':
            # 计算压缩前大小
            compressed_data = response.content
            self.stats['compressed_bytes'] += len(compressed_data)

            # 尝试解压计算原始大小
            try:
                with gzip.GzipFile(fileobj=BytesIO(compressed_data), mode='rb') as f:
                    uncompressed_data = f.read()
                    self.stats['total_bytes'] += len(uncompressed_data)
            except:
                # 无法解压时使用压缩大小
                self.stats['total_bytes'] += len(compressed_data)
        else:
            # 非压缩响应
            self.stats['total_bytes'] += len(response.content)

        return response

    def request(self, method, url, **kwargs):
        """发送请求并收集统计信息"""
        return self.session.request(method, url, **kwargs)

    def get_stats(self):
        """获取统计报告"""
        total_reqs = self.stats['total_requests']
        if total_reqs == 0:
            return "无请求数据"

        avg_time = sum(self.stats['response_times']) / total_reqs if total_reqs > 0 else 0
        success_rate = self.stats['successful_requests'] / total_reqs * 100 if total_reqs > 0 else 0

        compression_ratio = 0
        if self.stats['total_bytes'] > 0 and self.stats['compressed_bytes'] > 0:
            compression_ratio = (1 - self.stats['compressed_bytes'] / self.stats['total_bytes']) * 100

        report = f"""
HTTP 请求统计报告:
总请求数: {total_reqs}
成功率: {success_rate:.2f}%
平均响应时间: {avg_time:.4f}s
总传输字节: {self.stats['total_bytes']} bytes
压缩率: {compression_ratio:.2f}%

状态码分布:
{self._format_dict(self.stats['status_codes'])}

内容类型分布:
{self._format_dict(self.stats['content_types'])}
"""
        return report

    def _format_dict(self, d):
        """格式化字典输出"""
        return '\n'.join(f"  {k}: {v}" for k, v in d.items())

# 使用示例
analyzer = RequestAnalyzer()
analyzer.request('GET', 'https://httpbin.org/gzip')
analyzer.request('GET', 'https://httpbin.org/json')
analyzer.request('GET', 'https://httpbin.org/status/404')
print(analyzer.get_stats())
```

# 10. 爬虫与数据采集高级应用

## 10.1 智能用户代理轮换

```python
import random
import time
import json
import os
from pathlib import Path

class UserAgentRotator:
    """智能 User-Agent 轮换器"""

    # 预定义的设备类型
    DEVICE_TYPES = ['desktop', 'mobile', 'tablet']

    # 预定义的浏览器
    BROWSERS = {
        'desktop': ['chrome', 'firefox', 'edge', 'safari', 'opera'],
        'mobile': ['android', 'ios', 'samsung'],
        'tablet': ['ipad', 'android']
    }

    def __init__(self, ua_file=None):
        """
        初始化轮换器
        :param ua_file: User-Agent 列表文件路径（JSON 格式）
        """
        self.user_agents = {}

        # 尝试加载文件
        if ua_file and os.path.exists(ua_file):
            with open(ua_file, 'r') as f:
                self.user_agents = json.load(f)
        else:
            # 使用内置的基本 UA 集合
            self._initialize_default_uas()

    def _initialize_default_uas(self):
        """初始化默认 User-Agent 集合"""
        self.user_agents = {
            'desktop': {
                'chrome': [
                    'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 S
                    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.443
                ],
                'firefox': [
                    'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:89.0) Gecko/20100101 Firefox/89.0',
                    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:89.0) Gecko/20100101 Firefox/89.0'
                ],
                # 其他浏览器...
            },
            'mobile': {
                'android': [
                    'Mozilla/5.0 (Linux; Android 10; SM-G970F) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.120 M
                    'Mozilla/5.0 (Linux; Android 11; Pixel 5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.210 Mo
                ],
                'ios': [
                    'Mozilla/5.0 (iPhone; CPU iPhone OS 14_6 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Versio
                    'Mozilla/5.0 (iPhone; CPU iPhone OS 14_4_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Versi
                ],
                # 其他系统...
            },
            # 其他设备类型...
        }

    def get_random_user_agent(self, device_type=None, browser=None):
        """
        获取随机 User-Agent
        :param device_type: 设备类型 (desktop, mobile, tablet)
        :param browser: 浏览器类型 (chrome, firefox, safari 等)
        :return: 随机 User-Agent 字符串
        """
        # 如果未指定设备类型，随机选择一个
        if not device_type:
            device_type = random.choice(self.DEVICE_TYPES)

        # 获取该设备类型下的浏览器
        browsers = self.BROWSERS.get(device_type, [])

        # 如果未指定浏览器或指定的浏览器不在此设备类型中，随机选择一个
        if not browser or browser not in browsers:
            browser = random.choice(browsers) if browsers else None

        # 从 UA 列表中选择
        try:
            ua_list = self.user_agents[device_type][browser]
            return random.choice(ua_list)
        except (KeyError, IndexError):
            # 找不到对应的 UA，返回一个通用的
            return "Mozilla/5.0 (compatible; RequestsClient/1.0.0)"

    def get_consistent_user_agent(self, session_id=None):
        """
        获取一致的 User-Agent（会话级别保持一致）
        :param session_id: 会话 ID，如果为 None 则生成一个新的
        :return: (user_agent, session_id) 元组
        """
        if session_id is None:
            session_id = f"{time.time()}-{random.randint(1000, 9999)}"

        # 使用会话 ID 作为种子
        random.seed(session_id)
        device_type = random.choice(self.DEVICE_TYPES)
        browsers = self.BROWSERS.get(device_type, [])
        browser = random.choice(browsers) if browsers else None

        # 重置随机数生成器
        random.seed()

        ua = self.get_random_user_agent(device_type, browser)
        return ua, session_id

# 使用示例
rotator = UserAgentRotator()
ua = rotator.get_random_user_agent(device_type='mobile')
print(f"随机移动设备 UA: {ua}")

# 在会话中使用
session = requests.Session()
ua, session_id = rotator.get_consistent_user_agent()
session.headers.update({'User-Agent': ua})
print(f"会话 UA ({session_id}): {ua}")
```

## 10.2 智能请求延迟和随机性

```python
import random

import time

import math


class SmartDelay:

    """智能请求延迟生成器"""


    def __init__(self, min_delay=1.0, max_delay=5.0, pattern='random'):

        """

        初始化延迟生成器

        :param min_delay: 最小延迟秒数

        :param max_delay: 最大延迟秒数

        :param pattern: 延迟模式 ('random', 'linear', 'exp', 'sine')

        """

        self.min_delay = min_delay
        self.max_delay = max_delay
        self.pattern = pattern

        self.request_count = 0

        self.last_delay = 0


    def get_delay(self):

        """获取下一个延迟值"""

        delay = 0


        if self.pattern == 'random':

            # 完全随机延迟

            delay = random.uniform(self.min_delay, self.max_delay)


        elif self.pattern == 'linear':

            # 线性增长延迟（每次请求增加一点）

            base_delay = self.min_delay + (self.request_count * 0.1)

            delay = min(base_delay, self.max_delay)

            # 添加少量随机抖动

            delay += random.uniform(-0.1, 0.1) * delay


        elif self.pattern == 'exp':

            # 指数退避延迟（连续请求时逐渐增加）

            factor = math.exp(min(self.request_count * 0.1, 2)) - 1

            delay = self.min_delay + factor * (self.max_delay - self.min_delay) / (math.e - 1)

            # 添加少量随机抖动

            delay += random.uniform(-0.05, 0.05) * delay


        elif self.pattern == 'sine':

            # 正弦波形延迟（模拟人类行为的周期性）

            phase = self.request_count % 10 / 10 * 2 * math.pi

            amplitude = (self.max_delay - self.min_delay) / 2

            offset = self.min_delay + amplitude
            delay = offset + amplitude * math.sin(phase)

            # 添加少量随机抖动

            delay += random.uniform(-0.1, 0.1) * amplitude


        else:

            # 默认随机延迟

            delay = random.uniform(self.min_delay, self.max_delay)


        # 确保延迟在范围内

        delay = max(self.min_delay, min(delay, self.max_delay))


        self.last_delay = delay
        self.request_count += 1
        return delay

    def wait(self):
        """执行延迟等待"""
        delay = self.get_delay()
        time.sleep(delay)
        return delay

# 使用示例
def crawl_with_smart_delay(urls, delay_pattern='exp'):
    """使用智能延迟爬取 URL 列表"""
    delay = SmartDelay(min_delay=1.0, max_delay=3.0, pattern=delay_pattern)
    session = requests.Session()
    results = []

    for url in urls:
        wait_time = delay.wait()
        print(f"等待 {wait_time:.2f} 秒后请求 {url}")
        response = session.get(url)
        results.append(response)

    return results
```

## 10.3 自动处理 Cookies 和 JavaScript 挑战

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import json

class ChallengeHandler:
    """处理复杂网站挑战的工具类"""

    def __init__(self, headless=True):
        """
        初始化挑战处理器
        :param headless: 是否使用无头模式
        """
        self.driver = None
        self.headless = headless
        self.setup_driver()

    def setup_driver(self):
        """设置 Selenium WebDriver"""
        options = Options()
        if self.headless:
            options.add_argument('--headless')
        options.add_argument('--disable-gpu')
        options.add_argument('--no-sandbox')
        options.add_argument('--disable-dev-shm-usage')
        options.add_argument('--disable-extensions')

        # 添加随机 User-Agent
        ua_rotator = UserAgentRotator()
        ua = ua_rotator.get_random_user_agent(device_type='desktop')
```

```python
        options.add_argument(f'user-agent={ua}')

        self.driver = webdriver.Chrome(options=options)
        self.driver.set_page_load_timeout(30)

    def solve_cloudflare(self, url, timeout=30):
        """
        解决 Cloudflare 挑战
        :param url: 目标 URL
        :param timeout: 等待超时时间（秒）
        :return: 带有解决方案的 cookies 和 user-agent
        """
        if not self.driver:
            self.setup_driver()

        try:
            self.driver.get(url)

            # 等待 Cloudflare 挑战完成（通常页面会重定向或显示原始内容）
            WebDriverWait(self.driver, timeout).until(
                EC.presence_of_element_tag_name('body')
            )

            # 检查是否仍在挑战页面
            if "Checking your browser" in self.driver.page_source or "Just a moment" in self.driver.page_source:
                # 需要更长的等待
                print("检测到 Cloudflare 挑战，等待...")
                WebDriverWait(self.driver, timeout).until_not(
                    EC.text_to_be_present_in_element((By.TAG_NAME, 'body'), 'Checking your browser')
                )

            # 获取 cookies
            cookies = self.driver.get_cookies()
            user_agent = self.driver.execute_script("return navigator.userAgent")

            # 格式化 cookies 为字典
            cookies_dict = {cookie['name']: cookie['value'] for cookie in cookies}

            return {
                'cookies': cookies_dict,
                'user_agent': user_agent,
                'html': self.driver.page_source
            }

        except Exception as e:
            print(f"解决 Cloudflare 挑战时出错: {e}")
            return None

        finally:
            if not self.headless:
                input("按回车键关闭浏览器...")

    def transfer_to_requests(self, url):
        """
        将 Selenium 会话转移到 Requests
        :param url: 已解决挑战的 URL
        :return: 配置好的 requests 会话对象
        """
        result = self.solve_cloudflare(url)
        if not result:
            return None
        session = requests.Session()

        # 设置 User-Agent
        session.headers.update({
            'User-Agent': result['user_agent'],
            'Referer': url
        })

        # 设置 cookies
        for name, value in result['cookies'].items():
            session.cookies.set(name, value)

        return session

    def close(self):
        """关闭 WebDriver"""
        if self.driver:
            self.driver.quit()
            self.driver = None

    def __del__(self):
        """析构时关闭 WebDriver"""
        self.close()

# 使用示例
def scrape_protected_site(url):
    """抓取受保护的网站"""
    handler = ChallengeHandler(headless=True)
    try:
        # 解决挑战获取会话
        session = handler.transfer_to_requests(url)
        if not session:
            return "无法解决网站挑战"

        # 使用配置好的会话进行请求
        response = session.get(url)
        return response.text
    finally:
        handler.close()
```

# 11. 跨平台、云端与企业级应用

## 11.1 自适应重试与熔断策略

```
import time
import random
from functools import wraps
from requests.exceptions import RequestException, ConnectionError, Timeout

class CircuitBreaker:
    """
    断路器模式实现，在连续失败达到阈值后暂时禁止请求，
    防止连续请求对故障服务造成额外负担
    """
```

```python
    CLOSED = 'closed'      # 正常状态，允许请求
    OPEN = 'open'          # 断路状态，阻止请求
    HALF_OPEN = 'half-open'  # 半开状态，允许有限请求以探测服务恢复

    def __init__(self, failure_threshold=5, recovery_timeout=30,
                 expected_exceptions=(RequestException,)):
        """
        初始化断路器
        :param failure_threshold: 触发断路的连续失败次数
        :param recovery_timeout: 断路后尝试恢复的等待时间（秒）
        :param expected_exceptions: 计入失败次数的异常类型
        """
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.expected_exceptions = expected_exceptions

        self.state = self.CLOSED
        self.failure_count = 0
        self.last_failure_time = None

    def __call__(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if self.state == self.OPEN:
                # 检查是否到达恢复时间
                if time.time() - self.last_failure_time > self.recovery_timeout:
                    self.state = self.HALF_OPEN
                    print(f"断路器转为半开状态，尝试恢复...")
                else:
                    raise RuntimeError(f"断路器开启中，请求被阻断。恢复时间剩余：{self.recovery_timeout - (time.time() - self.last

            try:
                result = func(*args, **kwargs)

                # 成功请求，重置失败计数
                if self.state == self.HALF_OPEN:
                    self.state = self.CLOSED
                    print("服务恢复正常，断路器关闭")

                self.failure_count = 0
                return result

            except self.expected_exceptions as e:
                self.last_failure_time = time.time()
                self.failure_count += 1

                # 检查是否达到失败阈值
                if self.state == self.CLOSED and self.failure_count >= self.failure_threshold:
                    self.state = self.OPEN
                    print(f"连续失败 {self.failure_count} 次，断路器打开")

                if self.state == self.HALF_OPEN:
                    self.state = self.OPEN
                    print("恢复尝试失败，断路器重新打开")

                # 重新抛出异常
                raise

        return wrapper

# 自适应超时策略
```

```python
class AdaptiveTimeout:
    """
    自适应超时策略，根据历史响应时间动态调整超时设置
    """

    def __init__(self, initial_timeout=5.0, min_timeout=1.0,
                 max_timeout=30.0, history_size=10, percentile=90):
        """
        初始化自适应超时
        :param initial_timeout: 初始超时时间（秒）
        :param min_timeout: 最小超时时间（秒）
        :param max_timeout: 最大超时时间（秒）
        :param history_size: 历史响应时间记录数量
        :param percentile: 超时时间计算的百分位数（90表示取响应时间的90%分位数）
        """
        self.initial_timeout = initial_timeout
        self.min_timeout = min_timeout
        self.max_timeout = max_timeout
        self.history_size = history_size
        self.percentile = percentile

        self.response_times = []
        self.current_timeout = initial_timeout

    def record_response_time(self, elapsed_time):
        """
        记录响应时间
        :param elapsed_time: 响应耗时（秒）
        """
        self.response_times.append(elapsed_time)

        # 保持历史大小限制
        if len(self.response_times) > self.history_size:
            self.response_times.pop(0)

        # 更新超时设置
        self._update_timeout()

    def _update_timeout(self):
        """根据历史响应时间更新超时设置"""
        if not self.response_times:
            return

        # 计算百分位数
        sorted_times = sorted(self.response_times)
        idx = int(len(sorted_times) * self.percentile / 100)
        percentile_time = sorted_times[idx]

        # 设置为百分位数的 1.5 倍，确保有足够余量
        new_timeout = percentile_time * 1.5

        # 确保在范围内
        self.current_timeout = max(self.min_timeout, min(new_timeout, self.max_timeout))

    def get_timeout(self):
        """获取当前计算的超时值"""
        return self.current_timeout

# 使用自适应断路器和超时的综合示例
@CircuitBreaker(failure_threshold=3, recovery_timeout=10)
def resilient_request(url, session=None, adaptive_timeout=None):
    """具有弹性的 HTTP 请求函数"""
    s = session or requests.Session()

    # 使用自适应超时或默认值
    timeout = adaptive_timeout.get_timeout() if adaptive_timeout else 5.0

    start_time = time.time()
    try:
        response = s.get(url, timeout=timeout)

        # 记录响应时间
        elapsed = time.time() - start_time
        if adaptive_timeout:
            adaptive_timeout.record_response_time(elapsed)

        return response

    except (ConnectionError, Timeout) as e:
        # 重试前随机等待（抖动）
        wait_time = random.uniform(0.5, 2.0)
        print(f"请求失败，等待 {wait_time:.2f} 秒后重试: {str(e)}")
        time.sleep(wait_time)
        raise  # 重新抛出异常，让断路器处理

# 示例：使用自适应超时和断路器的弹性服务调用
def call_service_with_resilience(urls):
    """弹性服务调用示例"""
    session = requests.Session()
    adaptive_timeout = AdaptiveTimeout(
        initial_timeout=5.0,
        min_timeout=2.0,
        max_timeout=15.0
    )

    results = []
    for url in urls:
        try:
            response = resilient_request(
                url,
                session=session,
                adaptive_timeout=adaptive_timeout
            )
            results.append({
                'url': url,
                'success': True,
                'status': response.status_code,
                'timeout': adaptive_timeout.get_timeout()
            })
            print(f"成功: {url}, 当前超时设置: {adaptive_timeout.get_timeout():.2f}秒")

        except RuntimeError as e:
            # 断路器开启
            results.append({
                'url': url,
                'success': False,
                'error': str(e)
            })
            print(f"断路器阻断: {url}, {e}")

        except Exception as e:
            # 其他错误
            results.append({
                'url': url,
                'success': False,
                'error': str(e)
            })
            print(f"失败: {url}, {str(e)}")

    return results
```

## 11.2 跨平台配置与环境适配

```python
import os
import platform
import json
import sys
import socket
from contextlib import contextmanager

class RequestsConfig:
    """全局 Requests 配置管理器"""

    # 默认配置
    DEFAULT_CONFIG = {
        'timeout': 10,
        'verify': True,
        'cert': None,
        'proxies': {},
        'max_retries': 3,
        'pool_connections': 10,
        'pool_maxsize': 10,
        'user_agent': 'python-requests/{requests_version} ({system}/{release})'
    }

    def __init__(self, config_file=None):
        """
        初始化配置管理器
        :param config_file: 配置文件路径
        """
        self.config = self.DEFAULT_CONFIG.copy()
        self.config_file = config_file or self._get_default_config_path()
        self.load_config()

    def _get_default_config_path(self):
        """获取默认配置文件路径（跨平台）"""
        if platform.system() == 'Windows':
            base_dir = os.environ.get('APPDATA', os.path.expanduser('~'))
            return os.path.join(base_dir, 'python_requests', 'config.json')
        else:  # Unix-like systems
            return os.path.expanduser('~/.config/python_requests/config.json')

    def load_config(self):
        """从配置文件加载配置"""
        try:
            if os.path.exists(self.config_file):
                with open(self.config_file, 'r') as f:
                    file_config = json.load(f)
                    self.config.update(file_config)
                    print(f"从 {self.config_file} 加载配置成功")
        except Exception as e:
            print(f"加载配置文件失败: {e}")

    def save_config(self):
        """保存配置到文件"""
        try:
            # 确保目录存在
            os.makedirs(os.path.dirname(self.config_file), exist_ok=True)

            with open(self.config_file, 'w') as f:
                json.dump(self.config, f, indent=4)
                print(f"配置已保存到 {self.config_file}")
        except Exception as e:
            print(f"保存配置文件失败: {e}")

    def get_session(self):
        """创建预配置的会话对象"""
        session = requests.Session()

        # 应用通用配置
        session.timeout = self.config['timeout']
        session.verify = self.config['verify']
        session.cert = self.config['cert']
        session.proxies = self.config['proxies']

        # 设置用户代理
        ua = self.config['user_agent'].format(
            requests_version=requests.__version__,
            system=platform.system(),
            release=platform.release()
        )
        session.headers['User-Agent'] = ua

        # 配置连接池和重试逻辑
        adapter = HTTPAdapter(
            max_retries=self.config['max_retries'],
            pool_connections=self.config['pool_connections'],
            pool_maxsize=self.config['pool_maxsize']
        )
        session.mount('http://', adapter)
        session.mount('https://', adapter)

        return session

    def detect_network_environment(self):
        """检测当前的网络环境并更新配置"""
        # 判断是否在企业网络中（基于代理设置）
        in_corporate_network = False
        proxy_env = os.environ.get('HTTP_PROXY') or os.environ.get('http_proxy')
        if proxy_env:
            print(f"检测到代理环境: {proxy_env}")
            self.config['proxies'] = {
                'http': proxy_env,
                'https': os.environ.get('HTTPS_PROXY') or os.environ.get('https_proxy') or proxy_env
            }
            in_corporate_network = True

        # 检查是否有 VPN 连接
        # 这里仅是一个简单的示例，实际判断 VPN 连接需要更复杂的逻辑
        if platform.system() == "Windows":
            # Windows VPN 适配器通常有特定名称
            import subprocess
            try:
                result = subprocess.check_output('ipconfig', shell=True, text=True)
                if "VPN" in result or "Virtual" in result:
                    print("检测到可能的 VPN 连接")
                    # 可能需要针对 VPN 环境进行特定配置
            except Exception:
                pass

        # 检查特定的内网域名是否可访问
        def can_resolve(hostname):
            try:
                socket.gethostbyname(hostname)
                return True
            except socket.error:
                return False

        # 如果可以解析内部域名，说明在企业网络环境
        if can_resolve('internal.example.com'):
            in_corporate_network = True
            print("检测到企业内网环境")

            # 企业网络可能需要禁用 SSL 验证（不推荐但有时必要）
            if not self.config['verify']:
                print("警告: 在企业环境中禁用了 SSL 验证，这可能不安全")

        return {
            'in_corporate_network': in_corporate_network,
            'platform': platform.system(),
            'python_version': platform.python_version(),
        }

# 环境适配示例
def adapt_to_environment():
    """根据环境调整 requests 配置的示例"""
    config = RequestsConfig()
    env_info = config.detect_network_environment()

    # 创建适配当前环境的会话
    session = config.get_session()

    # 在企业网络中可能需要特殊处理
    if env_info['in_corporate_network']:
        # 可能需要添加内部 CA 证书
        ca_bundle_path = '/path/to/enterprise/ca-bundle.pem'
        if os.path.exists(ca_bundle_path):
            session.verify = ca_bundle_path

    # 在不同平台上的适配
    if env_info['platform'] == 'Windows':
        # Windows 特定配置
        if float(env_info['python_version'].split('.')[0]) < 3:
            # Python 2.x 在 Windows 上处理 SSL 的特殊设置
            session.verify = False
            import warnings
            warnings.warn("在 Windows 的旧版 Python 中禁用了 SSL 验证，请考虑升级")

    return session

# 网络环境切换处理
```

```python
@contextmanager
def network_environment(environment_type):
    """
    临时切换网络环境配置的上下文管理器
    :param environment_type: 环境类型，如 'corporate', 'home', 'mobile'
    """
    config = RequestsConfig()
    original_config = config.config.copy()

    try:
        # 根据环境类型调整配置
        if environment_type == 'corporate':
            config.config.update({
                'proxies': {'http': 'http://proxy.corp.example.com:8080',
                            'https': 'http://proxy.corp.example.com:8080'},
                'verify': '/path/to/corporate_ca_bundle.pem',
                'timeout': 30,  # 企业网络可能需要更长超时
            })
        elif environment_type == 'mobile':
            config.config.update({
                'proxies': {},  # 移动网络通常不需要代理
                'verify': True,
                'timeout': 10,  # 移动网络可能不稳定，使用适中超时
                'max_retries': 5,  # 更多重试次数
            })
        elif environment_type == 'home':
            config.config.update({
                'proxies': {},
                'verify': True,
                'timeout': 5,  # 家庭网络通常较快
            })

        # 创建适配环境的会话
        session = config.get_session()
        yield session

    finally:
        # 恢复原始配置
        config.config = original_config

# 使用示例
def request_across_environments(url):
    """在不同网络环境中发送请求的示例"""
    results = {}

    # 企业网络环境
    try:
        with network_environment('corporate') as session:
            print("使用企业网络配置发送请求...")
            response = session.get(url)
            results['corporate'] = {
                'status': response.status_code,
                'elapsed': response.elapsed.total_seconds()
            }
    except Exception as e:
        results['corporate'] = {'error': str(e)}

    # 家庭网络环境
    try:
        with network_environment('home') as session:
            print("使用家庭网络配置发送请求...")
            response = session.get(url)
            results['home'] = {
                'status': response.status_code,
                'elapsed': response.elapsed.total_seconds()
            }
    except Exception as e:
        results['home'] = {'error': str(e)}

    return results
```

## 11.3 云端与API网关集成

```
import hmac
import hashlib
import base64
import uuid
import datetime

class AWSSignatureV4:
    """AWS API 网关请求签名生成器（Signature Version 4）"""

    def __init__(self, access_key, secret_key, region, service='execute-api'):
        """
        初始化 AWS 签名生成器
        :param access_key: AWS 访问密钥 ID
        :param secret_key: AWS 密钥访问密钥
        :param region: AWS 区域
        :param service: AWS 服务名称
        """
        self.access_key = access_key
        self.secret_key = secret_key
        self.region = region
        self.service = service

    def _sign(self, key, msg):
        """使用 HMAC-SHA256 计算签名"""
        return hmac.new(key, msg.encode('utf-8'), hashlib.sha256).digest()

    def get_signature_key(self, date_stamp):
        """获取签名密钥"""
        k_date = self._sign(f'AWS4{self.secret_key}'.encode('utf-8'), date_stamp)
        k_region = self._sign(k_date, self.region)
        k_service = self._sign(k_region, self.service)
        k_signing = self._sign(k_service, 'aws4_request')
        return k_signing

    def sign_request(self, method, url, headers=None, data=None):
        """
        为 AWS API 网关请求生成签名头
        :param method: HTTP 方法
        :param url: 请求 URL
        :param headers: 请求头字典
        :param data: 请求数据
        :return: 包含签名的请求头字典
        """
        from urllib.parse import urlparse
        headers = headers or {}

        # 标准化 URL
        url_parts = urlparse(url)
        host = url_parts.netloc
        canonical_uri = url_parts.path or '/'

        # 设置请求日期和时间
        now = datetime.datetime.utcnow()
        amz_date = now.strftime('%Y%m%dT%H%M%SZ')
        date_stamp = now.strftime('%Y%m%d')

        # 使用随机标识符创建唯一的标头
        headers['x-amz-date'] = amz_date
        headers['host'] = host

        # 请求体哈希
        payload_hash = hashlib.sha256((data or '').encode('utf-8')).hexdigest()

        # 规范化标头和签名标头
        canonical_headers = ''.join(f"{k.lower()}:{v}\n" for k, v in sorted(headers.items()))
        signed_headers = ';'.join(k.lower() for k in sorted(headers.keys()))

        # 构造规范请求字符串
        canonical_request = f"{method}\n{canonical_uri}\n{url_parts.query}\n{canonical_headers}\n{signed_headers}\n{payload_

        # 创建签名字符串
        algorithm = 'AWS4-HMAC-SHA256'
        credential_scope = f"{date_stamp}/{self.region}/{self.service}/aws4_request"
        string_to_sign = f"{algorithm}\n{amz_date}\n{credential_scope}\n{hashlib.sha256(canonical_request.encode('utf-8')).h

        # 计算签名
        signing_key = self.get_signature_key(date_stamp)
        signature = hmac.new(signing_key, string_to_sign.encode('utf-8'), hashlib.sha256).hexdigest()

        # 添加授权头
        auth_header = (
            f"{algorithm} "
            f"Credential={self.access_key}/{credential_scope}, "
            f"SignedHeaders={signed_headers}, "
            f"Signature={signature}"
        )
        headers['Authorization'] = auth_header

        return headers

# 云端服务客户端示例
class CloudAPIClient:
    """通用云服务 API 客户端"""

    def __init__(self, base_url, auth_provider=None):
        """
        初始化云服务客户端
        :param base_url: API 基础 URL
        :param auth_provider: 认证提供程序（如 AWSSignatureV4 实例）
        """
        self.base_url = base_url.rstrip('/')
        self.auth_provider = auth_provider
        self.session = requests.Session()

        # 设置基本头部
```

```python
        self.session.headers.update({
            'Accept': 'application/json',
            'Content-Type': 'application/json',
            'User-Agent': f'CloudAPIClient/1.0 Python/{platform.python_version()}'
        })

    def request(self, method, endpoint, params=None, data=None, headers=None):
        """
        发送 API 请求
        :param method: HTTP 方法
        :param endpoint: API 端点（相对路径）
        :param params: URL 参数
        :param data: 请求数据（将被 JSON 序列化）
        :param headers: 附加头部
        :return: 响应对象
        """
        url = f"{self.base_url}/{endpoint.lstrip('/')}"
        request_headers = headers or {}

        # 序列化 JSON 数据
        json_data = None
        if data is not None:
            json_data = json.dumps(data)

        # 如果提供了认证提供者，获取签名头
        if self.auth_provider:
            # 为签名创建完整的 URL（包含查询参数）
            full_url = url
            if params:
                from urllib.parse import urlencode
                query_string = urlencode(params)
                full_url = f"{url}?{query_string}"

            # 获取签名头
            auth_headers = self.auth_provider.sign_request(
                method,
                full_url,
                headers=request_headers,
                data=json_data
            )
            request_headers.update(auth_headers)

        # 发送请求
        response = self.session.request(
            method,
            url,
            params=params,
            data=json_data,
            headers=request_headers
        )

        return response

    def get(self, endpoint, params=None, headers=None):
        """发送 GET 请求"""
        return self.request('GET', endpoint, params=params, headers=headers)

    def post(self, endpoint, data, params=None, headers=None):
        """发送 POST 请求"""
        return self.request('POST', endpoint, params=params, data=data, headers=headers)

    def put(self, endpoint, data, params=None, headers=None):
        """发送 PUT 请求"""
        return self.request('PUT', endpoint, params=params, data=data, headers=headers)
    def delete(self, endpoint, params=None, headers=None):
        """发送 DELETE 请求"""
        return self.request('DELETE', endpoint, params=params, headers=headers)

    def patch(self, endpoint, data, params=None, headers=None):
        """发送 PATCH 请求"""
        return self.request('PATCH', endpoint, params=params, data=data, headers=headers)

# AWS API 网关客户端示例
def create_aws_api_client(api_url, access_key, secret_key, region):
    """创建用于 AWS API 网关的客户端"""
    auth_provider = AWSSignatureV4(
        access_key=access_key,
        secret_key=secret_key,
        region=region
    )
    return CloudAPIClient(api_url, auth_provider=auth_provider)

# 使用示例
def aws_api_example():
    """AWS API 网关调用示例"""
    client = create_aws_api_client(
        api_url='https://api.example.com',
        access_key='YOUR_ACCESS_KEY',
        secret_key='YOUR_SECRET_KEY',
        region='us-west-2'
    )

    # 发送请求
    response = client.get('/users')
    if response.status_code == 200:
        return response.json()
    else:
        print(f"API 调用失败: {response.status_code} - {response.text}")
        return None
```

# 12. Python Requests 与异步编程

## 12.1 结合 asyncio 的异步封装

```python
import asyncio
import functools
from concurrent.futures import ThreadPoolExecutor

class AsyncRequests:
    """Requests 的异步封装"""

    def __init__(self, max_workers=10, session=None):
        """
        初始化异步请求封装
        :param max_workers: 工作线程池大小
        :param session: 可选的 requests.Session 对象
        """
        self.executor = ThreadPoolExecutor(max_workers=max_workers)
        self.session = session or requests.Session()

        self.loop = None

    async def request(self, method, url, **kwargs):
        """
        异步发送 HTTP 请求
        :param method: HTTP 方法
        :param url: 目标 URL
        :param kwargs: 请求参数
        :return: Response 对象
        """
        if self.loop is None:
            self.loop = asyncio.get_running_loop()

        # 将同步请求封装为异步任务
        return await self.loop.run_in_executor(
            self.executor,
            functools.partial(self.session.request, method, url, **kwargs)
        )

    async def get(self, url, **kwargs):
        """异步 GET 请求"""
        return await self.request('GET', url, **kwargs)

    async def post(self, url, **kwargs):
        """异步 POST 请求"""
        return await self.request('POST', url, **kwargs)

    async def put(self, url, **kwargs):
        """异步 PUT 请求"""
        return await self.request('PUT', url, **kwargs)

    async def delete(self, url, **kwargs):
        """异步 DELETE 请求"""
        return await self.request('DELETE', url, **kwargs)

    async def head(self, url, **kwargs):
        """异步 HEAD 请求"""
        return await self.request('HEAD', url, **kwargs)

    async def options(self, url, **kwargs):
        """异步 OPTIONS 请求"""
        return await self.request('OPTIONS', url, **kwargs)

    async def gather_requests(self, request_configs):
        """
        并发发送多个请求并收集结果
        :param request_configs: 请求配置列表, 每项格式为 (method, url, kwargs)
        :return: 响应列表
        """
        tasks = []
        for config in request_configs:
            if isinstance(config, tuple) and len(config) >= 2:
                method, url = config[0], config[1]
                kwargs = config[2] if len(config) > 2 else {}
                tasks.append(self.request(method, url, **kwargs))
            else:
                raise ValueError(f"无效的请求配置: {config}")
        return await asyncio.gather(*tasks, return_exceptions=True)

    def close(self):
        """关闭资源"""
        self.executor.shutdown(wait=False)
        self.session.close()

    async def __aenter__(self):
        """异步上下文管理器入口"""
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        """异步上下文管理器退出"""
        self.close()

# 使用示例
async def demo_async_requests():
    """演示异步请求用法"""
    async with AsyncRequests() as client:
        # 单个异步请求
        response = await client.get(
            'https://httpbin.org/get',
            params={'q': 'python requests'}
        )
        print(f"单个请求状态码: {response.status_code}")

        # 并发多个请求
        request_configs = [
            ('GET', 'https://httpbin.org/get'),
            ('POST', 'https://httpbin.org/post', {'data': {'key': 'value'}}),
            ('GET', 'https://httpbin.org/delay/2', {'timeout': 5})
        ]

        start = time.time()
        responses = await client.gather_requests(request_configs)
        elapsed = time.time() - start

        print(f"并发 {len(responses)} 个请求完成, 耗时 {elapsed:.2f} 秒")
        for i, resp in enumerate(responses):
            if isinstance(resp, Exception):
                print(f"请求 {i+1} 失败: {resp}")
            else:
                print(f"请求 {i+1} 成功: {resp.status_code}")

# 在异步环境中运行
if __name__ == "__main__":
    asyncio.run(demo_async_requests())
```

## 12.2 结合 HTTPX 实现真正的异步

```python
# 安装: pip install httpx
import httpx
import asyncio
import time

async def fetch_with_httpx():
    """使用 HTTPX 进行真正的异步 HTTP 请求"""
    # 创建异步客户端
    async with httpx.AsyncClient(
        timeout=30.0,
        limits=httpx.Limits(max_keepalive_connections=10, max_connections=20),
        http2=True  # 启用 HTTP/2
    ) as client:
        # 单个请求
        r = await client.get('https://httpbin.org/get')
        print(f"单个请求: {r.status_code}")

        # 并发多个请求
        urls = [
            'https://httpbin.org/get',
            'https://httpbin.org/delay/1',
            'https://httpbin.org/delay/2',
            'https://httpbin.org/delay/3'
        ]

        start = time.time()
        tasks = [client.get(url) for url in urls]
        responses = await asyncio.gather(*tasks, return_exceptions=True)
        elapsed = time.time() - start

        print(f"并发 {len(responses)} 个请求完成, 耗时 {elapsed:.2f} 秒")

        # 使用流式响应
        async with client.stream('GET', 'https://httpbin.org/bytes/10240') as r:
            total = 0
            async for chunk in r.aiter_bytes():
                total += len(chunk)
            print(f"流式下载完成, 总大小: {total} 字节")

# 比较 Requests 与 HTTPX
async def compare_requests_vs_httpx():
    """比较 Requests 与 HTTPX 的性能"""
    urls = ['https://httpbin.org/delay/1'] * 5

    # 使用 AsyncRequests (基于线程池的封装)
    print("使用 AsyncRequests (基于线程的) :")
    async with AsyncRequests() as req_client:
        start = time.time()
        configs = [('GET', url) for url in urls]
        await req_client.gather_requests(configs)
        elapsed = time.time() - start
        print(f"耗时: {elapsed:.2f} 秒")

    # 使用 HTTPX (真正的异步)
    print("使用 HTTPX (真正的异步) :")
    async with httpx.AsyncClient() as http_client:
        start = time.time()
        tasks = [http_client.get(url) for url in urls]
        await asyncio.gather(*tasks)
        elapsed = time.time() - start
        print(f"耗时: {elapsed:.2f} 秒")

# 从 Requests 迁移到 HTTPX
def requests_to_httpx_migration():
    """从 Requests 迁移到 HTTPX 的示例"""
    # Requests 示例
    session = requests.Session()
    session.headers.update({'User-Agent': 'my-app/1.0'})
    response = session.get('https://httpbin.org/get',
                         params={'q': 'python'},
                         timeout=5)
    print(f"Requests: {response.status_code}")
    session.close()

    # 等效的 HTTPX 同步代码
    client = httpx.Client(headers={'User-Agent': 'my-app/1.0'})
    response = client.get('https://httpbin.org/get',
                        params={'q': 'python'},
                        timeout=5)
    print(f"HTTPX 同步: {response.status_code}")
    client.close()

    # HTTPX 异步代码
    async def async_example():
        async with httpx.AsyncClient(headers={'User-Agent': 'my-app/1.0'}) as client:
            response = await client.get('https://httpbin.org/get',
                                     params={'q': 'python'},
                                     timeout=5)
            print(f"HTTPX 异步: {response.status_code}")

    # 运行异步函数
    asyncio.run(async_example())
```

# 13. 内部机制与性能调优

## 13.1 连接池与 Keep-Alive 优化

```
import requests
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry
from urllib3.poolmanager import PoolManager
import ssl
import urllib3

class OptimizedAdapter(HTTPAdapter):
    """优化的 HTTP 适配器，提供高级连接池和 TLS 配置"""

    def __init__(self, pool_connections=100, pool_maxsize=100,
                 max_retries=0, pool_block=False,
                 connection_timeout=None, read_timeout=None,
                 keepalive_expiry=60):
        """
        初始化优化的适配器
        :param pool_connections: 连接池数量
        :param pool_maxsize: 每个连接池的最大连接数
        :param max_retries: 最大重试次数
        :param pool_block: 连接池满时是否阻塞
        :param connection_timeout: 连接超时（秒）
        :param read_timeout: 读取超时（秒）
        :param keepalive_expiry: keep-alive 连接过期时间（秒）
        """
        self.connection_timeout = connection_timeout
        self.read_timeout = read_timeout
        self.keepalive_expiry = keepalive_expiry

        super().__init__(
            pool_connections=pool_connections,
            pool_maxsize=pool_maxsize,
            max_retries=max_retries,
            pool_block=pool_block
        )

    def init_poolmanager(self, connections, maxsize, block=False, **pool_kwargs):
        """初始化连接池管理器"""
        # 添加 Keep-Alive 配置
        pool_kwargs['timeout'] = urllib3.Timeout(
            connect=self.connection_timeout,
            read=self.read_timeout
        )

        # 添加 Keep-Alive 过期时间
        if self.keepalive_expiry:
            pool_kwargs['keepalive_expiry'] = self.keepalive_expiry

        # 优化 SSL 上下文
        ssl_context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
        ssl_context.set_ciphers('ECDHE+AESGCM:ECDHE+CHACHA20:DHE+AESGCM:DHE+CHACHA20')
        ssl_context.options |= ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1  # 禁用 TLS 1.0 和 1.1

        # 初始化连接池
        self.poolmanager = PoolManager(
            num_pools=connections,
            maxsize=maxsize,
            block=block,
            ssl_context=ssl_context,
            **pool_kwargs
        )

# 应用优化适配器
def optimize_session(timeout=(3.05, 60), max_workers=25, keepalive=120):
    """创建优化的会话对象"""
    session = requests.Session()

    # 创建适配器
    adapter = OptimizedAdapter(
        pool_connections=max_workers,
        pool_maxsize=max_workers * 2,
        connection_timeout=timeout[0] if isinstance(timeout, tuple) else timeout,
        read_timeout=timeout[1] if isinstance(timeout, tuple) else timeout,
        keepalive_expiry=keepalive
    )

    # 应用适配器
    session.mount('http://', adapter)
    session.mount('https://', adapter)

    # 优化会话级别头部
    session.headers.update({
        'Connection': 'keep-alive',
        'Keep-Alive': f'timeout={keepalive}',
        'Accept-Encoding': 'gzip, deflate, br'
    })

    return session

# 性能对比测试
```

```
def benchmark_pooling():
    """对比不同连接池配置的性能"""
    import time
    import statistics

    urls = ['https://httpbin.org/get'] * 50
    results = {}

    # 测试标准配置
    session_standard = requests.Session()
    start = time.time()
    for url in urls:
        session_standard.get(url)
    results['standard'] = time.time() - start

    # 测试优化配置
    session_optimized = optimize_session(max_workers=25)
    start = time.time()
    for url in urls:
        session_optimized.get(url)
    results['optimized'] = time.time() - start

    # 测试优化配置 + 并发
    from concurrent.futures import ThreadPoolExecutor
    session_concurrent = optimize_session(max_workers=25)

    def fetch(url):
        return session_concurrent.get(url)

    start = time.time()
    with ThreadPoolExecutor(max_workers=25) as executor:
        list(executor.map(fetch, urls))
    results['optimized_concurrent'] = time.time() - start

    # 输出结果
    print("性能对比 (总时间, 秒):")
    for name, duration in results.items():
        print(f"{name}: {duration:.2f}秒")

    # 计算加速比
    speedup = results['standard'] / results['optimized_concurrent']
    print(f"总加速比: {speedup:.2f}x")
```

## 13.2 请求与响应的内部结构

```
import requests
import json
from requests.utils import get_encoding_from_headers
import inspect

def analyze_request_internals(request_or_prepared):
    """
    分析请求对象的内部结构
    :param request_or_prepared: Request 对象或 PreparedRequest 对象
    :return: 包含请求内部信息的字典
    """
    # 确保我们有 PreparedRequest 对象
    if isinstance(request_or_prepared, requests.Request):
        prepared = request_or_prepared.prepare()
    else:
        prepared = request_or_prepared

    # 分析请求的内部结构
    internals = {
        'type': type(prepared).__name__,
        'method': prepared.method,
        'url': prepared.url,
        'headers': dict(prepared.headers),
        'body_size': len(prepared.body) if prepared.body else 0,
        'hooks': prepared.hooks,
    }

    # 分析 URL 组成部分
    url_parts = requests.utils.urlparse(prepared.url)
    internals['url_components'] = {
        'scheme': url_parts.scheme,
        'host': url_parts.netloc,
        'port': url_parts.port,
        'path': url_parts.path,
        'query': url_parts.query,
    }

    # 分析身份验证信息（如果有）
    if hasattr(prepared, 'auth') and prepared.auth:
        internals['auth'] = {
            'type': type(prepared.auth).__name__,
            'uses_header': 'Authorization' in prepared.headers
        }

    # 检查 Cookie
    if 'Cookie' in prepared.headers:
        internals['has_cookies'] = True
        # 解析 Cookie 字符串
        cookies = {}
        for cookie_part in prepared.headers['Cookie'].split(';'):
            if '=' in cookie_part:
                key, value = cookie_part.strip().split('=', 1)
                cookies[key] = value
        internals['cookies'] = cookies

    return internals

def analyze_response_internals(response):
    """
    分析响应对象的内部结构
    :param response: Response 对象
    :return: 包含响应内部信息的字典
    """
    internals = {
        'type': type(response).__name__,
        'status_code': response.status_code,
        'reason': response.reason,
        'url': response.url,
        'headers': dict(response.headers),
        'encoding': response.encoding,
        'elapsed': response.elapsed.total_seconds(),
        'is_redirect': response.is_redirect,
        'is_permanent_redirect': response.is_permanent_redirect,
        'content_size': len(response.content),
    }

    # 分析编码信息
    content_type = response.headers.get('Content-Type', '')
    internals['content_type'] = content_type
    encoding_from_headers = get_encoding_from_headers(response.headers)
    internals['encoding_from_headers'] = encoding_from_headers

    # 分析链接头
    if 'Link' in response.headers:
        internals['links'] = requests.utils.parse_header_links(response.headers['Link'])

    # 分析历史（重定向）
    if response.history:
        internals['has_redirects'] = True
        internals['redirect_count'] = len(response.history)
        internals['redirect_path'] = [
            {'url': r.url, 'status': r.status_code}
            for r in response.history
        ]

    # 分析原始连接信息
    if hasattr(response, 'raw') and response.raw:
        raw_info = {
            'type': type(response.raw).__name__,
            'version': response.raw.version,
            'status': response.raw.status,
            'reason': response.raw.reason,
            'closed': response.raw.closed,
        }

        # 检查是否有 urllib3 相关属性
        for attr in ['_connection', '_fp', '_fp_bytes_read', '_pool']:
            if hasattr(response.raw, attr):
                attr_value = getattr(response.raw, attr)
                raw_info[f'has_{attr}'] = attr_value is not None

        internals['raw'] = raw_info

    return internals

# 使用示例
def request_response_analysis_demo():
    """请求和响应内部结构分析演示"""
    # 创建一个请求
    req = requests.Request('GET', 'https://httpbin.org/get',
                           params={'q': 'python requests'},
                           headers={'User-Agent': 'Custom/1.0'},
                           cookies={'session_id': '12345'})

    # 使用会话准备请求
    session = requests.Session()
    prepared = session.prepare_request(req)

    # 分析请求
    req_analysis = analyze_request_internals(prepared)
    print("请求分析:")
    print(json.dumps(req_analysis, indent=2, ensure_ascii=False))
    # 发送请求
    response = session.send(prepared)

    # 分析响应
    resp_analysis = analyze_response_internals(response)
    print("\n响应分析:")
    print(json.dumps(resp_analysis, indent=2, ensure_ascii=False))
```

## 13.3 高级性能分析与内存优化

```
import time
import os
from functools import wraps
import gc

def memory_profile(func):
    """内存使用情况分析装饰器"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        try:
            import psutil
            import tracemalloc

            # 启用内存跟踪
            tracemalloc.start()

            # 获取当前进程
            process = psutil.Process(os.getpid())

            # 获取基准内存使用情况
            base_memory = process.memory_info().rss / 1024 / 1024  # MB
            print(f"[内存分析] 初始内存使用量: {base_memory:.2f} MB")

            # 记录开始时间
            start_time = time.time()

            # 调用原始函数
            result = func(*args, **kwargs)

            # 记录结束时间
            elapsed_time = time.time() - start_time

            # 获取执行后的内存使用情况
            current_memory = process.memory_info().rss / 1024 / 1024  # MB
            memory_diff = current_memory - base_memory

            print(f"[内存分析] 执行时间: {elapsed_time:.4f} 秒")
            print(f"[内存分析] 最终内存使用量: {current_memory:.2f} MB")
            print(f"[内存分析] 内存差异: {memory_diff:.2f} MB")

            # 获取内存分配的快照
            snapshot = tracemalloc.take_snapshot()
            top_stats = snapshot.statistics('lineno')

            print("[内存分析] 内存分配 Top 5:")
            for stat in top_stats[:5]:
                print(f"  {stat.size/1024:.1f} KB: {stat.traceback.format()[0]}")

            # 停止内存跟踪
```

```python
            tracemalloc.stop()

            return result
        except ImportError:
            print("[内存分析] 需要安装 psutil 和 tracemalloc 库。")
            return func(*args, **kwargs)
    return wrapper

def optimize_memory_usage(session=None, max_poolsize=10):
    """优化会话的内存使用"""
    if session is None:
        session = requests.Session()

    # 限制连接池大小以减少内存使用
    adapter = requests.adapters.HTTPAdapter(pool_maxsize=max_poolsize)
    session.mount('http://', adapter)
    session.mount('https://', adapter)

    # 禁用不必要的功能来节省内存
    session.trust_env = False  # 不从环境变量加载代理设置

    return session

@memory_profile
def memory_usage_test(num_requests=100, use_session=True, optimize=False):
    """测试不同请求方式的内存使用情况"""
    url = 'https://httpbin.org/get'

    if use_session:
        if optimize:
            session = optimize_memory_usage()
            print("使用优化的会话")
        else:
            session = requests.Session()
            print("使用标准会话")

        for _ in range(num_requests):
            response = session.get(url)
            # 确保内容被读取和解码
            _ = response.text

        # 主动关闭会话，释放资源
        session.close()
    else:
        print("不使用会话，单独请求")
        for _ in range(num_requests):
            response = requests.get(url)
            # 确保内容被读取和解码
            _ = response.text

    # 强制垃圾回收
    gc.collect()

def memory_leak_simulation():
    """模拟并检测内存泄漏"""
    # 存储所有响应的列表（不良实践）
    responses = []

    # 追踪内存使用
    try:
        import psutil
        process = psutil.Process(os.getpid())
        measurements = []
        for i in range(50):
            # 记录内存使用
            mem_before = process.memory_info().rss / 1024 / 1024

            # 创建请求并保留响应（潜在的内存泄漏）
            response = requests.get('https://httpbin.org/get')
            responses.append(response)

            # 再次记录内存使用
            mem_after = process.memory_info().rss / 1024 / 1024
            measurements.append((i, mem_after))

            print(f"请求 {i}: 内存使用 {mem_after:.2f} MB, 差异 {mem_after-mem_before:.2f} MB")

        # 清除响应列表并强制垃圾回收
        print("清除响应列表...")
        responses.clear()
        gc.collect()

        # 最终内存使用
        final_mem = process.memory_info().rss / 1024 / 1024
        print(f"清除后内存使用: {final_mem:.2f} MB")
        if final_mem > measurements[0][1]:
            print(f"可能存在内存泄漏! 差异: {final_mem - measurements[0][1]:.2f} MB")

        # 内存使用随时间变化
        print("\n内存使用趋势:")
        for i, (req, mem) in enumerate(measurements):
            if i > 0:
                diff = mem - measurements[i-1][1]
                print(f"请求 {req}: {mem:.2f} MB ({'+' if diff >= 0 else ''}{diff:.2f} MB)")
    except ImportError:
        print("需要安装 psutil 库来监控内存使用。")

    # 清理
    del responses
    gc.collect()
```

# 14. 前沿技术与实战应用

## 14.1 HTTP/3 与 QUIC 协议支持

```
# 需要安装 aioquic: pip install aioquic
# 需要安装 httpx: pip install httpx[http2]
import httpx
import anyio

async def http3_request():
    """使用 HTTP/3 发送请求（基于 QUIC 协议）"""
    # 创建支持 HTTP/3 的客户端
    async with httpx.AsyncClient(http3=True) as client:
        # 发送 HTTP/3 请求
        r = await client.get("https://cloudflare-quic.com/")
```

```python
        print(f"HTTP 版本: {r.http_version}")
        print(f"状态码: {r.status_code}")
        print(f"响应头: {dict(r.headers)}")
        return r

# HTTP 版本兼容性检查
async def check_http_compatibility(url):
    """检查目标服务器支持的 HTTP 协议版本"""
    results = {}

    # 检查 HTTP/1.1
    try:
        async with httpx.AsyncClient(http2=False) as client:
            r = await client.get(url)
            results['HTTP/1.1'] = {
                'supported': True,
                'status': r.status_code,
                'version': r.http_version
            }
    except Exception as e:
        results['HTTP/1.1'] = {'supported': False, 'error': str(e)}

    # 检查 HTTP/2
    try:
        async with httpx.AsyncClient(http2=True) as client:
            r = await client.get(url)
            results['HTTP/2'] = {
                'supported': r.http_version == 'HTTP/2',
                'status': r.status_code,
                'version': r.http_version
            }
    except Exception as e:
        results['HTTP/2'] = {'supported': False, 'error': str(e)}

    # 检查 HTTP/3
    try:
        async with httpx.AsyncClient(http3=True) as client:
            r = await client.get(url)
            results['HTTP/3'] = {
                'supported': r.http_version == 'HTTP/3',
                'status': r.status_code,
                'version': r.http_version
            }
    except Exception as e:
        results['HTTP/3'] = {'supported': False, 'error': str(e)}

    return results

# 智能协议选择
class SmartHTTPClient:
    """智能 HTTP 客户端, 自动选择最佳协议版本"""

    def __init__(self):
        self.protocol_cache = {}  # 记录域名与协议支持情况

    async def _probe_protocols(self, url):
        """探测服务器支持的协议"""
        from urllib.parse import urlparse
        domain = urlparse(url).netloc
        # 检查缓存
        if domain in self.protocol_cache:
            return self.protocol_cache[domain]

        # 探测支持的协议
        results = await check_http_compatibility(url)

        # 确定最佳协议
        best_protocol = 'http1'  # 默认 HTTP/1.1

        if results.get('HTTP/3', {}).get('supported', False):
            best_protocol = 'http3'
        elif results.get('HTTP/2', {}).get('supported', False):
            best_protocol = 'http2'

        # 缓存结果
        self.protocol_cache[domain] = best_protocol
        return best_protocol

    async def request(self, method, url, **kwargs):
        """发送请求，自动选择最佳协议"""
        # 确定最佳协议
        protocol = await self._probe_protocols(url)

        # 创建适当的客户端
        client_kwargs = {}
        if protocol == 'http2':
            client_kwargs['http2'] = True
        elif protocol == 'http3':
            client_kwargs['http3'] = True

        # 发送请求
        async with httpx.AsyncClient(**client_kwargs) as client:
            return await client.request(method, url, **kwargs)

# 示例使用
async def smart_http_demo():
    client = SmartHTTPClient()

    # 测试不同网站
    sites = [
        'https://www.google.com',
        'https://www.cloudflare.com',
        'https://www.facebook.com'
    ]

    for site in sites:
        try:
            response = await client.request('GET', site)
            print(f"{site}: 使用 {response.http_version}, 状态码 {response.status_code}")
        except Exception as e:
            print(f"{site}: 请求失败 - {e}")
```

## 14.2 基于 Rust 的高性能 HTTP 客户端桥接

```
# 如果你的项目需要极致性能，可以考虑与 Rust 语言客户端桥接
```

```python
# 安装: pip install httpx-rs  # 基于 Rust 的 HTTP 客户端, 这是一个假设的包名

# 以下是概念性代码, 展示如何与现有 Rust HTTP 客户端 (如 reqwest) 集成
try:
    import httpx_rs as httpx_rust  # 假设的基于 Rust 的客户端
    RUST_AVAILABLE = True
except ImportError:
    RUST_AVAILABLE = False

class HybridClient:
    """混合 HTTP 客户端, 根据需求切换 Python 和 Rust 实现"""

    def __init__(self, prefer_rust=True):
        """
        初始化混合客户端
        :param prefer_rust: 是否优先使用 Rust 实现
        """
        self.prefer_rust = prefer_rust and RUST_AVAILABLE

        # 初始化 Python 客户端
        self.py_client = requests.Session()

        # 如果可用, 初始化 Rust 客户端
        self.rust_client = None
        if RUST_AVAILABLE:
            self.rust_client = httpx_rust.Client()  # 假设的接口

    def request(self, method, url, **kwargs):
        """
        发送 HTTP 请求
        :param method: HTTP 方法
        :param url: 目标 URL
        :param kwargs: 请求参数
        :return: 响应对象
        """
        # 决定使用哪个实现
        if self.prefer_rust and self.rust_client is not None:
            # 使用 Rust 实现
            try:
                return self._request_rust(method, url, **kwargs)
            except Exception as e:
                # 如果 Rust 实现失败, 回退到 Python
                print(f"Rust 客户端失败, 回退到 Python: {e}")
                return self._request_python(method, url, **kwargs)
        else:
            # 使用 Python 实现
            return self._request_python(method, url, **kwargs)

    def _request_python(self, method, url, **kwargs):
        """使用 Python 实现发送请求"""
        return self.py_client.request(method, url, **kwargs)

    def _request_rust(self, method, url, **kwargs):
        """使用 Rust 实现发送请求"""
        # 转换参数格式以匹配 Rust API
        if 'params' in kwargs:
            kwargs['query'] = kwargs.pop('params')
        if 'data' in kwargs:
            kwargs['body'] = kwargs.pop('data')

        # 调用 Rust 客户端
        response = self.rust_client.request(method, url, **kwargs)

        # 将 Rust 响应转换为 Python 响应格式
        return self._convert_response(response)

    def _convert_response(self, rust_response):
        """将 Rust 响应转换为与 requests.Response 兼容的格式"""
        # 这是一个概念示意，实际实现需要根据 Rust 库的 API 调整
        response = requests.Response()
        response.status_code = rust_response.status
        response._content = rust_response.body
        response.headers = dict(rust_response.headers)
        response.url = rust_response.url

        return response

    def close(self):
        """关闭客户端"""
        self.py_client.close()
        if self.rust_client is not None:
            self.rust_client.close()

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close()

# 性能基准测试
def benchmark_hybrid_client():
    """比较 Python 和 Rust 实现的性能"""
    import time

    urls = ['https://httpbin.org/get'] * 100
    results = {'python': [], 'rust': [], 'hybrid': []}

    # 纯 Python 测试
    with requests.Session() as client:
        start = time.time()
        for url in urls:
            client.get(url)
        elapsed = time.time() - start
        results['python'] = elapsed
    print(f"Python 实现: {elapsed:.2f} 秒")

    # 如果 Rust 客户端可用，进行测试
    if RUST_AVAILABLE:
        client = httpx_rust.Client()
        start = time.time()
        for url in urls:
            client.get(url)
        elapsed = time.time() - start
        results['rust'] = elapsed
        client.close()
        print(f"Rust 实现: {elapsed:.2f} 秒")

    # 混合客户端测试
    with HybridClient(prefer_rust=RUST_AVAILABLE) as client:
        start = time.time()
        for url in urls:
            client.request('GET', url)
```

~~~
        elapsed = time.time() - start
        results['hybrid'] = elapsed
    print(f"混合实现: {elapsed:.2f} 秒")

    return results

## 15. 跨平台与特殊环境配置

### 15.1 移动端与资源受限设备优化

```python
import platform
import psutil
import warnings

class ResourceAwareRequests:
    """资源感知的 Requests 客户端，适用于移动设备和资源受限环境"""

    def __init__(self, low_memory_threshold=20, low_battery_threshold=15):
        """
        初始化资源感知客户端
        :param low_memory_threshold: 低内存阈值百分比
        :param low_battery_threshold: 低电量阈值百分比（仅移动设备）
        """
        self.session = requests.Session()
        self.low_memory_threshold = low_memory_threshold
        self.low_battery_threshold = low_battery_threshold

        # 检测环境
        self.is_mobile = self._detect_mobile()
        self.platform_info = self._get_platform_info()

        # 根据环境优化会话
        self._optimize_for_environment()

    def _detect_mobile(self):
        """检测是否为移动设备"""
        system = platform.system().lower()
        machine = platform.machine().lower()

        # 安卓检测
        if system == 'linux' and 'android' in platform.version().lower():
            return True

        # iOS 检测（通常在 Mac 上运行）
        if system == 'darwin' and ('iphone' in machine or 'ipad' in machine):
            return True

        return False

    def _get_platform_info(self):
        """获取平台信息"""
        info = {
            'system': platform.system(),
            'release': platform.release(),
            'machine': platform.machine(),
            'processor': platform.processor(),
            'python_version': platform.python_version(),
            'memory_total': psutil.virtual_memory().total
        }

        # 尝试获取移动设备特定信息
        try:
~~~

```python
            if self.is_mobile:
                # 获取电池信息（如果可用）
                battery = psutil.sensors_battery()
                if battery:
                    info['battery_percent'] = battery.percent
                    info['battery_power_plugged'] = battery.power_plugged
        except (AttributeError, NotImplementedError):
            pass

        return info

    def _optimize_for_environment(self):
        """根据环境优化会话设置"""

        # 检查是否为低内存环境
        low_memory = psutil.virtual_memory().percent > self.low_memory_threshold

        # 检查是否为低电量环境
        low_battery = False
        on_battery = False

        try:
            battery = psutil.sensors_battery()
            if battery:
                low_battery = battery.percent < self.low_battery_threshold
                on_battery = not battery.power_plugged
        except (AttributeError, NotImplementedError):
            pass

        # 根据环境限制缓存大小
        if low_memory:
            # 设置较小的连接池
            adapter = requests.adapters.HTTPAdapter(
                pool_connections=5,
                pool_maxsize=10
            )
            self.session.mount('http://', adapter)
            self.session.mount('https://', adapter)

            # 在每个请求后强制进行垃圾回收
            self._force_gc = True
        else:
            self._force_gc = False

        # 根据电量状态调整超时和重试
        if self.is_mobile and (low_battery or on_battery):
            # 缩短超时，减少重试
            self.default_timeout = 10
            retry = Retry(total=1, backoff_factor=0.5)

            # 禁用不必要的功能
            self.prefetch = False

            # 发出警告
            if low_battery:
                warnings.warn("电量低，已优化网络请求以省电")
        else:
            # 正常设置
            self.default_timeout = 30
            retry = Retry(total=3, backoff_factor=0.5)
            self.prefetch = True

    def request(self, method, url, **kwargs):
        """发送 HTTP 请求（资源感知）"""
        import gc

        # 应用默认超时
        if 'timeout' not in kwargs:
            kwargs['timeout'] = self.default_timeout

        # 移动端不使用紧缩编码（省电）
        if self.is_mobile:
            if 'headers' not in kwargs:
                kwargs['headers'] = {}

            # 如果是低电量状态，禁用压缩节约解压缩 CPU
            battery = psutil.sensors_battery()
            if battery and battery.percent < self.low_battery_threshold:
                kwargs['headers']['Accept-Encoding'] = 'identity'

        # 移动端使用流式传输
        if self.is_mobile and kwargs.get('stream') is None:
            # 对于大型请求使用流式传输
            kwargs['stream'] = True

        # 发送请求
        response = self.session.request(method, url, **kwargs)

        # 如果是流式请求，确保内容被及时读取和处理
        if kwargs.get('stream'):
            content = b''
            for chunk in response.iter_content(chunk_size=4096):
                content += chunk

            # 手动设置内容
            response._content = content
            response._content_consumed = True

        # 低内存环境下强制垃圾回收
        if self._force_gc:
            gc.collect()

        return response

    def get(self, url, **kwargs):
        """发送 GET 请求"""
        return self.request('GET', url, **kwargs)

    def post(self, url, **kwargs):
        """发送 POST 请求"""
        return self.request('POST', url, **kwargs)

    def close(self):
        """关闭客户端"""
        self.session.close()

# 移动设备连接恢复策略
class MobileConnectionManager:
    """移动网络连接管理器"""

    def __init__(self, retry_delay=5, max_retries=12, monitor_interval=30):
        """
        初始化移动连接管理器

        :param retry_delay: 重试延迟（秒）

        :param max_retries: 最大重试次数

        :param monitor_interval: 连接监控间隔（秒）

        """
        self.retry_delay = retry_delay
        self.max_retries = max_retries
        self.monitor_interval = monitor_interval

        # 连接状态

        self.connected = False

        self.connection_type = None

        self.signal_strength = None

        # 启动监控线程

        self._stop_monitor = False

        self._monitor_thread = threading.Thread(
            target=self._monitor_connection,
            daemon=True
        )
        self._monitor_thread.start()

    def _monitor_connection(self):

        """监控网络连接状态"""

        while not self._stop_monitor:

            # 检测连接状态

            self._check_connection()

            # 等待下次检测

            time.sleep(self.monitor_interval)

    def _check_connection(self):

        """检测当前连接状态"""

        import socket

        try:

            # 尝试连接外部服务器

            socket.create_connection(("8.8.8.8", 53), timeout=3)

            self.connected = True

            # 尝试确定连接类型（WiFi/移动数据）

            # 这里只是概念性演示，实际需要特定于平台的实现

            if platform.system() == 'Linux' and 'android' in platform.version().lower():

                # Android 设备上可以检查网络接口

                # 实际实现可能需要使用 Android API

                self.connection_type = 'mobile_data'  # 示例值

            else:

                self.connection_type = 'wifi'  # 默认假设 WiFi

        except (socket.timeout, socket.error):

            self.connected = False

            self.connection_type = None

    def wait_for_connection(self, url, timeout=60):

        """

        等待连接恢复

        :param url: 测试 URL

        :param timeout: 最大等待时间（秒）

        :return: 连接成功后的 Response 对象，或 None

        """

        start_time = time.time()
        retry_count = 0

        while time.time() - start_time < timeout and retry_count < self.max_retries:
            try:
                if not self.connected:
                    print(f"等待网络连接恢复 ({retry_count+1}/{self.max_retries})...")
                    time.sleep(self.retry_delay)
                    retry_count += 1
                    continue

                # 尝试请求
                response = requests.get(url, timeout=5)
                if response.status_code < 400:
                    print("连接已恢复!")
                    return response
            except requests.RequestException:
                pass

            retry_count += 1
            print(f"重试中 ({retry_count}/{self.max_retries})...")
            time.sleep(self.retry_delay)

        print("连接恢复超时")
        return None

    def stop(self):
        """停止连接监控"""
        self._stop_monitor = True
        if self._monitor_thread.is_alive():
            self._monitor_thread.join(5)

# 使用示例
def mobile_optimized_requests_demo():
    """移动优化请求示例"""
    # 创建资源感知客户端
    client = ResourceAwareRequests()

    # 创建连接管理器
    conn_manager = MobileConnectionManager()

    try:
        # 发送请求
        try:
            response = client.get('https://api.example.com/data')
            print(f"请求成功: {response.status_code}")
            return response.json()
        except requests.RequestException as e:
            print(f"请求失败: {e}")

            # 等待连接恢复
            if conn_manager.wait_for_connection('https://api.example.com/data'):
                # 重试请求
                response = client.get('https://api.example.com/data')
                print(f"恢复连接后请求成功: {response.status_code}")
                return response.json()

            return None
    finally:
        conn_manager.stop()
        client.close()
```

~~~
### 15.2 IoT 与嵌入式设备请求优化

```python
import socket
import select
import time

class LightweightHTTPClient:
    """轻量级 HTTP 客户端，适用于嵌入式设备和 IoT 场景"""

    def __init__(self, timeout=10, keepalive=False):
        """
        初始化轻量级客户端
        :param timeout: 请求超时（秒）
        :param keepalive: 是否使用 Keep-Alive 连接
        """
        self.timeout = timeout
        self.keepalive = keepalive
        self.connection = None
        self.last_host = None

    def _connect(self, host, port):
        """创建到服务器的 TCP 连接"""
        if self.connection and self.last_host == (host, port) and self.keepalive:
            # 检查连接是否仍然有效
            try:
                self.connection.settimeout(0)
                if self.connection.recv(1, socket.MSG_PEEK) == b'':
                    # 连接已关闭，重新创建
                    self.connection.close()
                    self.connection = None
                self.connection.settimeout(self.timeout)
            except (socket.error, BlockingIOError):
                # 连接仍然有效
                pass

        if not self.connection:
            # 创建新连接
            addr_info = socket.getaddrinfo(host, port, socket.AF_INET, socket.SOCK_STREAM)
            sock_addr = addr_info[0][-1]
            self.connection = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.connection.settimeout(self.timeout)
            self.connection.connect(sock_addr)
            self.last_host = (host, port)

    def _send_request(self, method, host, path, headers=None, data=None):
        """发送 HTTP 请求"""
        headers = headers or {}

        # 构建请求
        request_line = f"{method} {path} HTTP/1.1\r\n"

        # 添加标准头
        headers.update({
            'Host': host,
            'Connection': 'keep-alive' if self.keepalive else 'close',
            'User-Agent': 'LightweightHTTPClient/1.0'
        })

        # 如果有请求体，添加 Content-Length
        if data:
~~~

```python
            headers['Content-Length'] = str(len(data))

        # 构建完整请求
        header_lines = "\r\n".join([f"{k}: {v}" for k, v in headers.items()])
        request = f"{request_line}{header_lines}\r\n\r\n"

        if data:
            request += data if isinstance(data, str) else data.decode('utf-8')

        # 发送请求
        self.connection.sendall(request.encode('utf-8'))

    def _read_response(self):
        """读取 HTTP 响应"""
        # 读取状态行和头
        response_data = b''
        headers_done = False

        while not headers_done:
            chunk = self.connection.recv(1024)
            if not chunk:
                break

            response_data += chunk
            if b'\r\n\r\n' in response_data:
                headers_done = True

        # 解析状态行
        headers_end = response_data.find(b'\r\n\r\n')
        header_text = response_data[:headers_end].decode('utf-8')
        body_start = headers_end + 4

        # 解析响应行
        status_line, headers_text = header_text.split('\r\n', 1)
        version, status, reason = status_line.split(' ', 2)

        # 解析响应头
        headers = {}
        for line in headers_text.split('\r\n'):
            if line:
                name, value = line.split(':', 1)
                headers[name.strip()] = value.strip()

        # 读取响应体
        body = response_data[body_start:]

        # 根据 Content-Length 或 Transfer-Encoding 处理响应体
        if 'Content-Length' in headers:
            content_length = int(headers['Content-Length'])
            while len(body) < content_length:
                chunk = self.connection.recv(1024)
                if not chunk:
                    break
                body += chunk
        elif headers.get('Transfer-Encoding') == 'chunked':
            # 处理分块编码（简化实现）
            while True:
                if b'\r\n' not in body:
                    chunk = self.connection.recv(1024)
                    if not chunk:
```

```
                    break
                body += chunk
                continue

            # 解析块大小
            chunk_size_end = body.find(b'\r\n')
            chunk_size_hex = body[:chunk_size_end].decode('ascii')
            chunk_size = int(chunk_size_hex, 16)

            if chunk_size == 0:
                # 最后一个块
                break

            # 读取更多数据直到完成当前块
            while len(body) < chunk_size_end + 2 + chunk_size + 2:
                chunk = self.connection.recv(1024)
                if not chunk:
                    break
                body += chunk

        # 如果连接不保持，关闭它
        if not self.keepalive or headers.get('Connection') == 'close':
            self.connection.close()
            self.connection = None

        return {
            'status': int(status),
            'reason': reason,
            'headers': headers,
            'body': body
        }

    def request(self, method, url, headers=None, data=None):
        """
        发送 HTTP 请求
        :param method: HTTP 方法
        :param url: 目标 URL
        :param headers: 请求头字典
        :param data: 请求体
        :return: 响应字典
        """
        from urllib.parse import urlparse

        # 解析 URL
        parsed_url = urlparse(url)
        scheme = parsed_url.scheme
        host = parsed_url.netloc
        path = parsed_url.path or '/'
        if parsed_url.query:
            path += '?' + parsed_url.query

        # 确定端口
        if ':' in host:
            host, port_str = host.split(':', 1)
            port = int(port_str)
        else:
            port = 443 if scheme == 'https' else 80

        # 不支持 HTTPS（需要 SSL 库）
        if scheme == 'https':
```

```python
            raise ValueError("HTTPS not supported by LightweightHTTPClient")

        # 建立连接
        self._connect(host, port)

        # 发送请求
        self._send_request(method, host, path, headers, data)

        # 读取响应
        return self._read_response()

    def close(self):
        """关闭连接"""
        if self.connection:
            self.connection.close()
            self.connection = None

# 功耗优化请求
class PowerEfficientRequests:
    """功耗优化的请求客户端，适用于电池供电设备"""

    def __init__(self, power_save_mode=False, batch_interval=300):
        """
        初始化功耗优化客户端
        :param power_save_mode: 是否启用省电模式
        :param batch_interval: 批处理间隔（秒）
        """
        self.power_save_mode = power_save_mode
        self.batch_interval = batch_interval
        self.request_queue = []
        self.last_batch_time = 0

    def queue_request(self, method, url, **kwargs):
        """
        将请求加入队列等待批处理
        :param method: HTTP 方法
        :param url: 目标 URL
        :param kwargs: 请求参数
        :return: 请求 ID
        """
        request_id = str(time.time()) + '-' + str(len(self.request_queue))
        self.request_queue.append({
            'id': request_id,
            'method': method,
            'url': url,
            'kwargs': kwargs,
            'timestamp': time.time()
        })
        return request_id

    def process_queue(self, force=False):
        """
        处理请求队列
        :param force: 是否强制处理（忽略批处理间隔）
        :return: 处理结果字典
        """
        now = time.time()

        # 如果不强制处理且不到批处理时间，不做任何事
        if not force and now - self.last_batch_time < self.batch_interval:
            return {}
        # 没有请求需要处理
        if not self.request_queue:
            return {}

        # 创建会话
        session = requests.Session()

        # 在省电模式下优化会话
        if self.power_save_mode:
            # 禁用不必要的功能
            session.headers['Accept-Encoding'] = 'identity'  # 禁用压缩
            adapter = requests.adapters.HTTPAdapter(
                pool_connections=1,
                pool_maxsize=1
            )
            session.mount('http://', adapter)
            session.mount('https://', adapter)

        # 处理所有请求
        results = {}
        for request in self.request_queue:
            try:
                response = session.request(
                    request['method'],
                    request['url'],
                    **request['kwargs']
                )
                results[request['id']] = {
                    'success': True,
                    'status_code': response.status_code,
                    'content': response.text,
                    'headers': dict(response.headers)
                }
            except Exception as e:
                results[request['id']] = {
                    'success': False,
                    'error': str(e)
                }

        # 清空队列并更新最后批处理时间
        self.request_queue = []
        self.last_batch_time = now

        # 关闭会话
        session.close()

        return results

    def immediate_request(self, method, url, **kwargs):
        """
        立即发送请求（不进行批处理）
        :param method: HTTP 方法
        :param url: 目标 URL
        :param kwargs: 请求参数
        :return: 响应对象
        """
        with requests.Session() as session:
            # 在省电模式下优化会话
            if self.power_save_mode:
                session.headers['Accept-Encoding'] = 'identity'
            return session.request(method, url, **kwargs)

# 使用示例
def iot_optimized_requests_demo():
    """IoT 优化请求示例"""
    # 轻量级客户端示例
    client = LightweightHTTPClient(keepalive=True)
    try:
        # 发送请求
        response = client.request('GET', 'http://example.com/')
        print(f"状态码: {response['status']}")
        print(f"响应体大小: {len(response['body'])} 字节")
    finally:
        client.close()

    # 功耗优化客户端示例
    power_client = PowerEfficientRequests(power_save_mode=True)

    # 队列多个请求
    ids = []
    ids.append(power_client.queue_request('GET', 'http://example.com/api/data1'))
    ids.append(power_client.queue_request('GET', 'http://example.com/api/data2'))

    # 手动处理队列
    results = power_client.process_queue(force=True)

    for req_id, result in results.items():
        if result['success']:
            print(f"请求 {req_id} 成功: {result['status_code']}")
        else:
            print(f"请求 {req_id} 失败: {result['error']}")
```

## 15.3 容器与无服务器环境适配

```
import os
import tempfile
import json
import atexit
from datetime import datetime

class ContainerOptimizedRequests:
    """容器优化请求，针对 Docker 和无服务器环境"""

    def __init__(self, cache_dir=None, connection_reuse=True, auto_cleanup=True):
        """
        初始化容器优化请求
        :param cache_dir: 缓存目录（默认使用容器安全的临时目录）
        :param connection_reuse: 是否重用连接
        :param auto_cleanup: 是否自动清理临时文件
        """
        # 检测是否在容器中运行
        self.in_container = self._detect_container()

        # 设置缓存目录
```

```python
        if cache_dir:
            self.cache_dir = cache_dir
        else:
            if self.in_container:
                # 在容器内使用 /tmp 目录
                self.cache_dir = '/tmp/requests_cache'
            else:
                # 在非容器环境使用系统临时目录
                self.cache_dir = os.path.join(tempfile.gettempdir(), 'requests_cache')

        # 创建缓存目录
        os.makedirs(self.cache_dir, exist_ok=True)

        # 会话配置
        self.session = requests.Session()
        if connection_reuse:
            adapter = requests.adapters.HTTPAdapter(
                pool_connections=10,
                pool_maxsize=100
            )
            self.session.mount('http://', adapter)
            self.session.mount('https://', adapter)

        # 注册退出时清理
        if auto_cleanup:
            atexit.register(self._cleanup)

    def _detect_container(self):
        """检测是否在容器中运行"""
        # 检查 cgroup（适用于 Docker、Kubernetes 等）
        if os.path.exists('/proc/1/cgroup'):
            with open('/proc/1/cgroup', 'r') as f:
                content = f.read()
                if 'docker' in content or 'kubepods' in content:
                    return True

        # 检查是否存在 /.dockerenv 文件
        if os.path.exists('/.dockerenv'):
            return True

        # 检查环境变量
        if 'KUBERNETES_SERVICE_HOST' in os.environ:
            return True

        # 检查 Lambda 环境变量
        if 'AWS_LAMBDA_FUNCTION_NAME' in os.environ:
            return True

        return False

    def _get_cache_key(self, method, url, **kwargs):
        """生成缓存键"""
        # 创建请求指纹
        key_parts = [method, url]

        # 添加查询参数
        if 'params' in kwargs:
            key_parts.append(json.dumps(kwargs['params'], sort_keys=True))

        # 添加请求体（如果不太大）
        if 'data' in kwargs and len(str(kwargs['data'])) < 1024:
            key_parts.append(str(kwargs['data']))
        elif 'json' in kwargs:
            key_parts.append(json.dumps(kwargs['json'], sort_keys=True))

        # 生成键哈希
        import hashlib
        cache_key = hashlib.md5('|'.join(key_parts).encode('utf-8')).hexdigest()

        return cache_key

    def _get_cache_path(self, cache_key):
        """获取缓存文件路径"""
        return os.path.join(self.cache_dir, cache_key)

    def _save_response_to_cache(self, cache_key, response, expire_seconds=3600):
        """保存响应到缓存"""
        cache_path = self._get_cache_path(cache_key)

        # 创建缓存条目
        cache_data = {
            'url': response.url,
            'status_code': response.status_code,
            'headers': dict(response.headers),
            'content': response.content.decode('utf-8', errors='replace'),
            'timestamp': datetime.now().timestamp(),
            'expires': datetime.now().timestamp() + expire_seconds
        }

        # 写入文件
        with open(cache_path, 'w') as f:
            json.dump(cache_data, f)

    def _get_response_from_cache(self, cache_key):
        """从缓存获取响应"""
        cache_path = self._get_cache_path(cache_key)

        if not os.path.exists(cache_path):
            return None

        # 读取缓存
        with open(cache_path, 'r') as f:
            try:
                cache_data = json.load(f)
            except json.JSONDecodeError:
                # 缓存文件损坏
                os.remove(cache_path)
                return None

        # 检查过期
        if datetime.now().timestamp() > cache_data.get('expires', 0):
            # 缓存已过期
            os.remove(cache_path)
            return None

        # 重建响应对象
        response = requests.Response()
        response.status_code = cache_data['status_code']
        response.headers = requests.structures.CaseInsensitiveDict(cache_data['headers'])
        response._content = cache_data['content'].encode('utf-8')
        response.url = cache_data['url']

        return response
    def request(self, method, url, use_cache=False, cache_ttl=3600, **kwargs):
        """
        发送 HTTP 请求，可选择使用缓存
        :param method: HTTP 方法
        :param url: 目标 URL
        :param use_cache: 是否使用缓存
        :param cache_ttl: 缓存生存时间（秒）
        :param kwargs: 请求参数
        :return: 响应对象
        """
        # 如果使用缓存，尝试从缓存获取
        if use_cache:
            cache_key = self._get_cache_key(method, url, **kwargs)
            cached_response = self._get_response_from_cache(cache_key)

            if cached_response:
                return cached_response

        # 发送实际请求
        response = self.session.request(method, url, **kwargs)

        # 如果需要，缓存响应
        if use_cache and response.status_code < 400:
            cache_key = self._get_cache_key(method, url, **kwargs)
            self._save_response_to_cache(cache_key, response, cache_ttl)

        return response

    def _cleanup(self):
        """清理过期缓存和临时文件"""
        now = datetime.now().timestamp()
        count = 0

        for filename in os.listdir(self.cache_dir):
            filepath = os.path.join(self.cache_dir, filename)

            # 跳过目录
            if os.path.isdir(filepath):
                continue

            try:
                # 检查文件是否是有效的缓存
                with open(filepath, 'r') as f:
                    cache_data = json.load(f)

                # 删除过期的缓存
                if now > cache_data.get('expires', 0):
                    os.remove(filepath)
                    count += 1
            except (json.JSONDecodeError, IOError):
                # 删除损坏的缓存文件
                os.remove(filepath)
                count += 1

        return count

    def get(self, url, **kwargs):
        """GET 请求"""
        return self.request('GET', url, **kwargs)

    def post(self, url, **kwargs):
        """POST 请求"""
        return self.request('POST', url, **kwargs)

    def close(self):
        """关闭客户端"""
        self.session.close()

# Serverless 函数集成示例
def serverless_request_example(event, context):
    """AWS Lambda 或其他无服务器环境中的请求示例"""
    try:
        # 创建优化客户端
        client = ContainerOptimizedRequests(auto_cleanup=True)

        # 处理请求
        api_url = event.get('api_url', 'https://api.example.com/data')
        use_cache = event.get('use_cache', True)

        # 发送请求
        response = client.get(api_url, use_cache=use_cache)

        # 返回结果
        return {
            'statusCode': response.status_code,
            'headers': dict(response.headers),
            'body': response.text,
            'from_cache': hasattr(response, '_from_cache') and response._from_cache,
            'container_environment': client.in_container
        }
    except Exception as e:
        return {
            'statusCode': 500,
            'body': f"Error: {str(e)}"
        }
    finally:
        # 确保资源被释放
        if 'client' in locals():
            client.close()
```

- END -