Requests项目演化规律与设计模式分析

组员信息：
组员1：黄开为（组长） 学号：20232241412
组员2：全俊赫 学号：20232241374
组员3：陈甜甜 学号：20232241234
组员4：杜瑗琳 学号：20232241447
组员5：闫文琪 学号：20232241501

第一部分：提交历史演化规律分析

摘要

本文以 Python 开源项目 requests 为研究对象，基于其全生命周期 Git 提交历史数据，从贡献者生态、提交活跃度、提交类型三方面开展分析，挖掘项目从创始到稳定维护的演化规律，剖析其从 “创始人驱动” 向 “社区维护” 转型的特征，为同类开源项目研究与维护提供参考。分析所用数据均来自项目公开 Git 仓库提交记录，覆盖全时段及近五年、近两年维度，保证分析结果的全面性与针对性。

一、引言

requests 是 Python 生态中主流的 HTTP 客户端库，其发展路径贴合成熟开源项目的典型特征。Git 提交历史记录了项目迭代轨迹、贡献者协作模式及维护重心的变迁，对其进行量化分析可清晰呈现项目生命周期、核心维护团队更替等规律。

二、数据来源与分析维度

2.1 数据来源

分析数据来自 requests 开源项目的公开 Git 仓库提交记录，涵盖项目创建至今的提交作者、提交时间、提交类型、提交描述等信息。经数据清洗与量化统计，生成年度提交量趋势图，以及全时段 / 近五年 / 近两年贡献者排名图和提交类型分布图等可视化图表，为分析提供数据支撑。

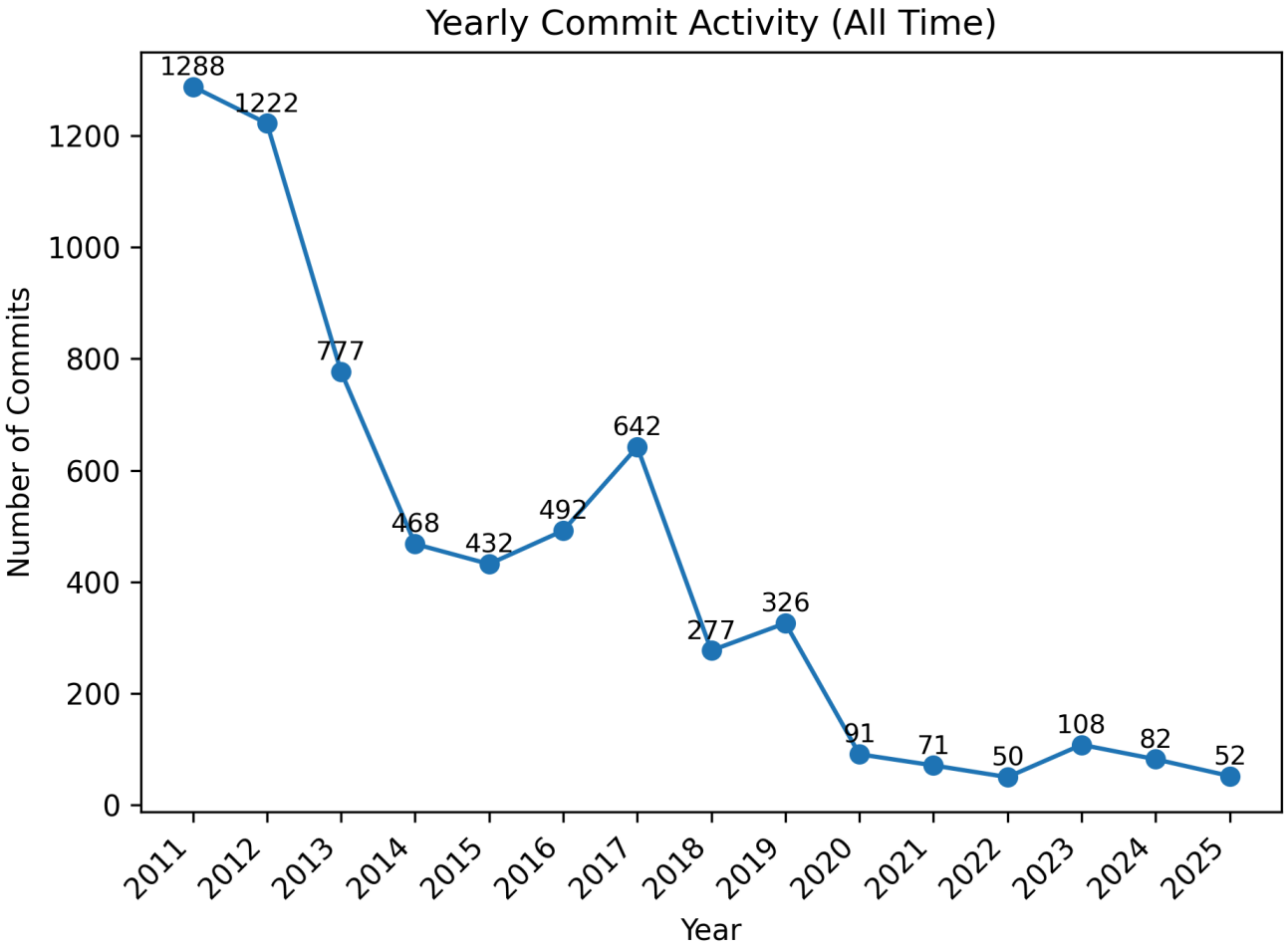
2.2 分析维度

结合可视化图表，聚焦三大核心维度并对比全时段、近五年、近两年数据：

1. **提交活跃度**：通过年度提交量划分项目生命周期阶段，分析迭代节奏
2. **贡献者生态**：分析核心贡献者构成及变迁，探究代际交接特征
3. **提交类型**：对比分布差异，分析维护重心转移规律

三、核心分析结果

3.1 提交活跃度：三阶段演进，契合成熟开源项目生命周期

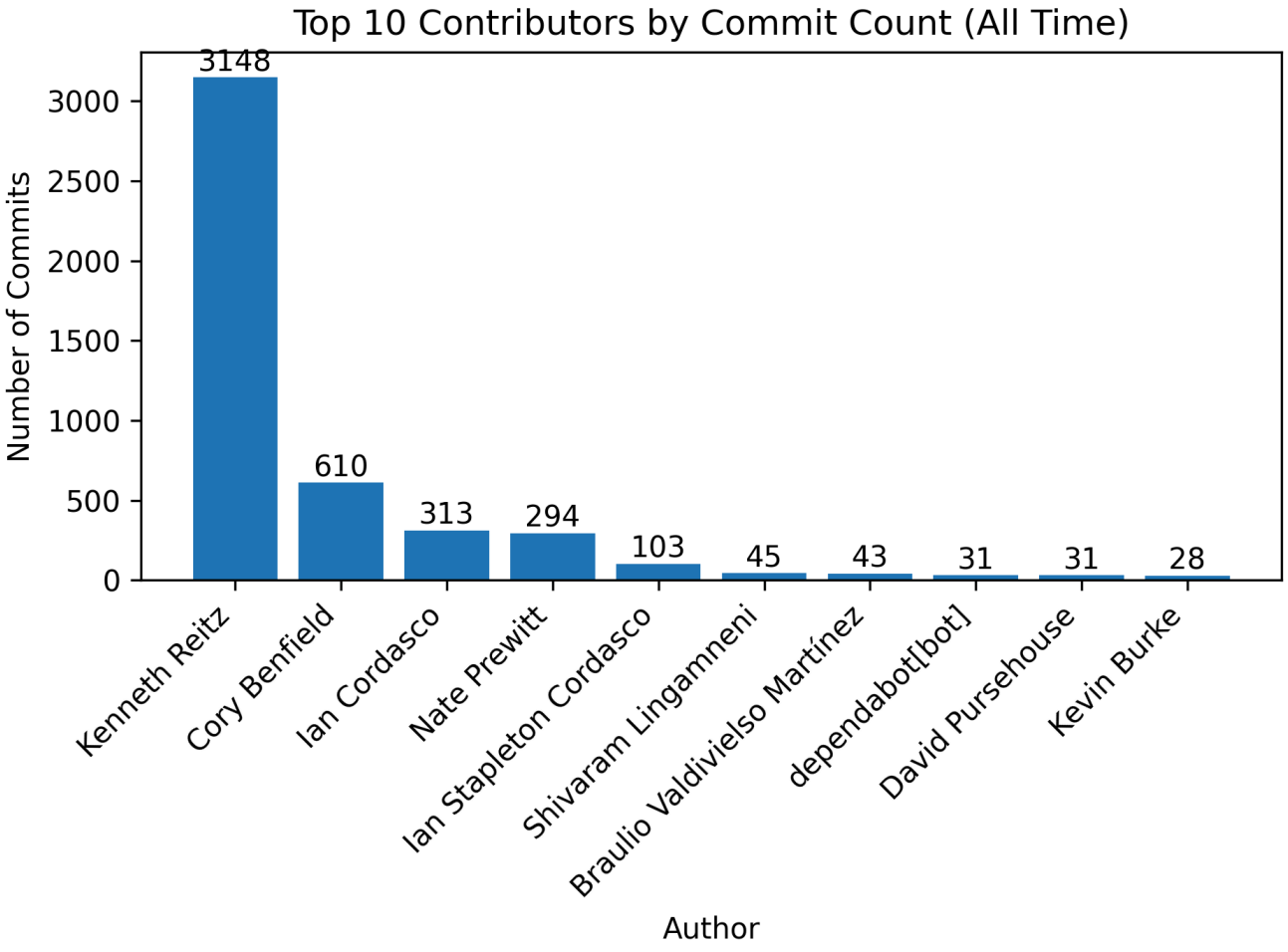


基于年度提交量趋势分析，requests 提交活跃度呈现 “爆发期-优化期-稳定期” 三阶段特征：

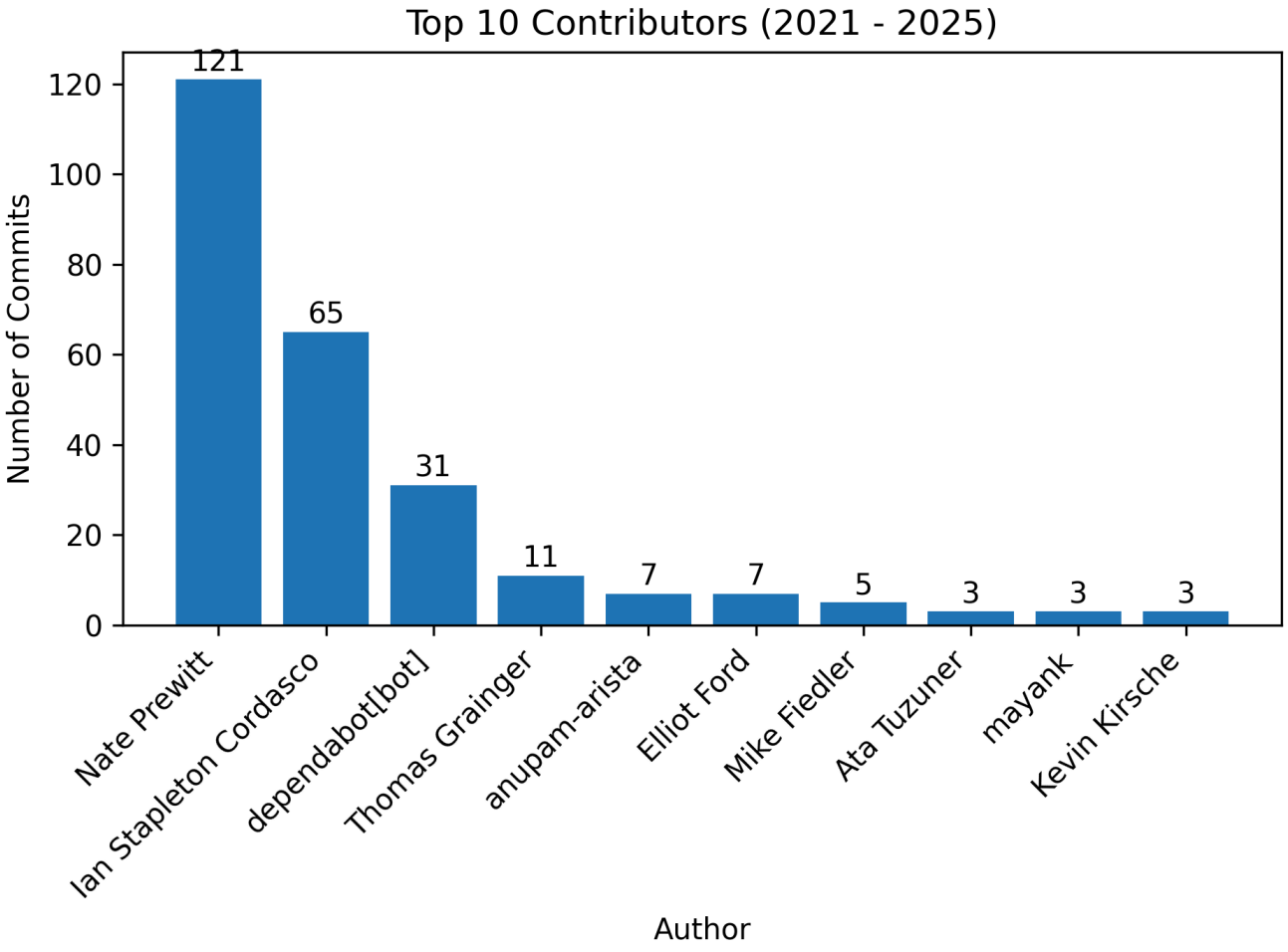
- **爆发期（2011-2013 年）**：年度提交量极高，2011 年达 1288 次，核心目标是完善核心功能、优化 API 设计、解决基础兼容性问题，推动项目快速成型并积累社区影响力。
- **波动优化期（2014-2019 年）**：提交量基本维持在 300-700 次 / 年，工作重心从 “新增功能” 转向 “优化体验、修复 bug”，提升项目稳定性与可靠性。
- **稳定维护期（2020 年至今）**：提交量降至每年几十次（2025 年仅 52 次），迭代节奏平缓；项目功能已完善，工作重心为关键 bug 修复、依赖更新及少量功能优化，进入稳定维护阶段。

3.2 贡献者生态：从创始人主导到社区协同与自动化维护

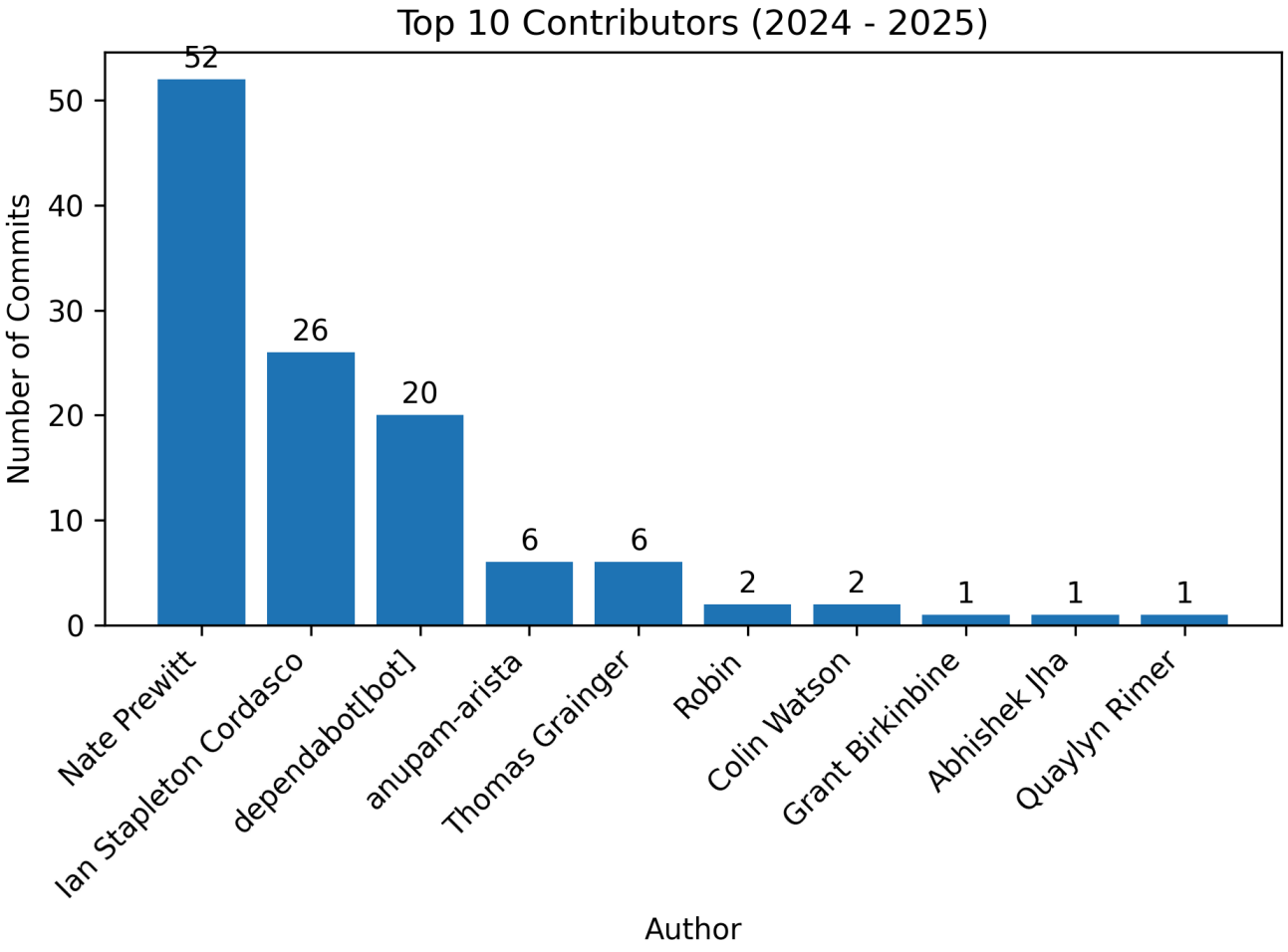
requests 贡献者生态呈现清晰的代际交接特征，逐步形成 “核心维护者 + 社区贡献 + 机器人自动化” 的协同模式。



- **全时段**：项目创始人 Kenneth Reitz 以 3148 次提交居贡献榜首位，提交量超第二名 Cory Benfield（610 次）的 5 倍，体现其在项目初期的主导作用，奠定了项目技术架构与功能基础；全时段贡献者群体多元，反映项目的开源吸引力。



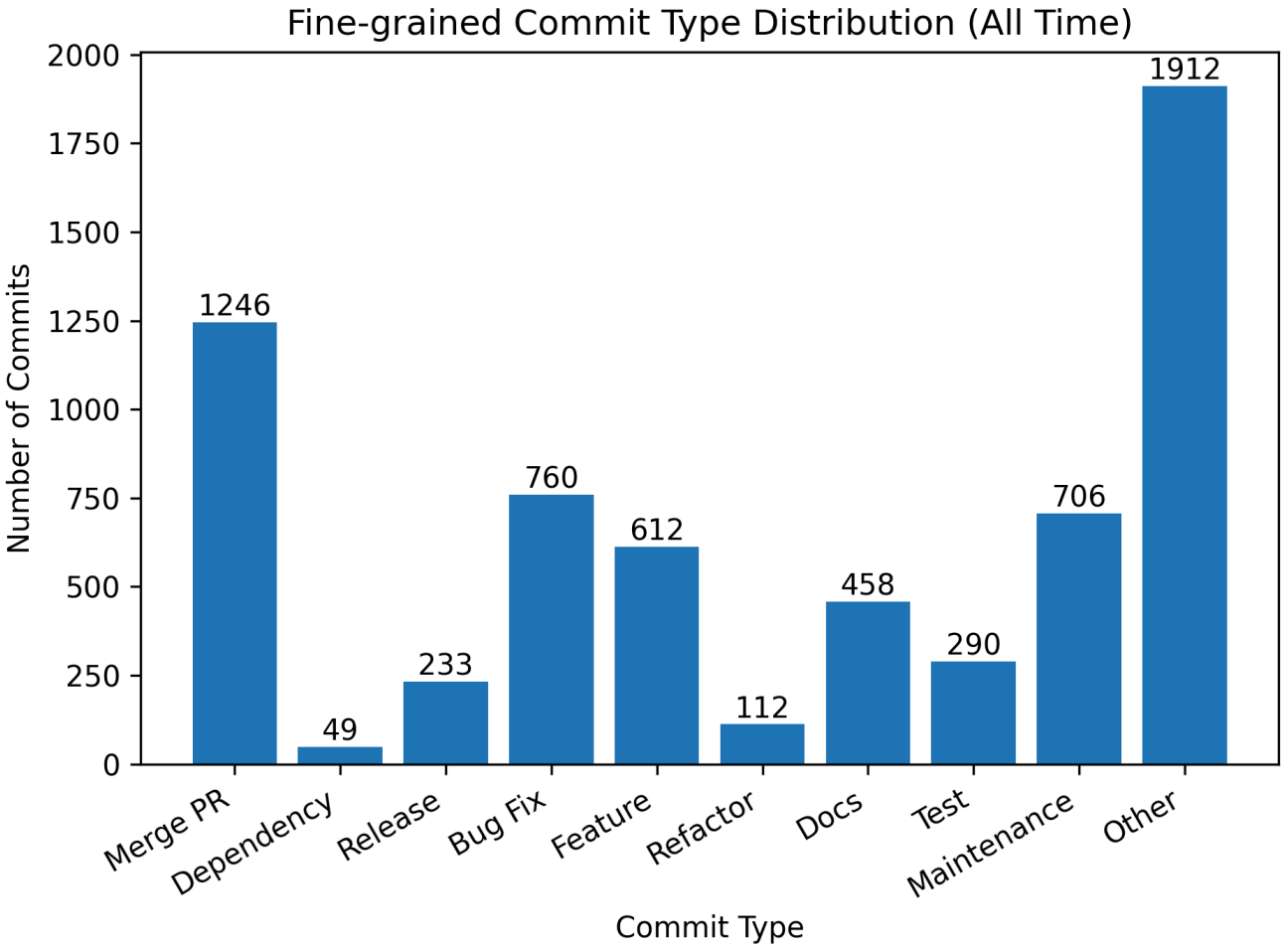
- **近五年（2021-2025）**：Kenneth Reitz 退出核心活跃行列，Nate Prewitt（121 次提交）成为核心维护者，Ian Stapleton Cordasco 紧随其后，项目转向 “社区维护”；机器人账号 dependabot[bot] 进入贡献榜前列，参与依赖更新等自动化维护。



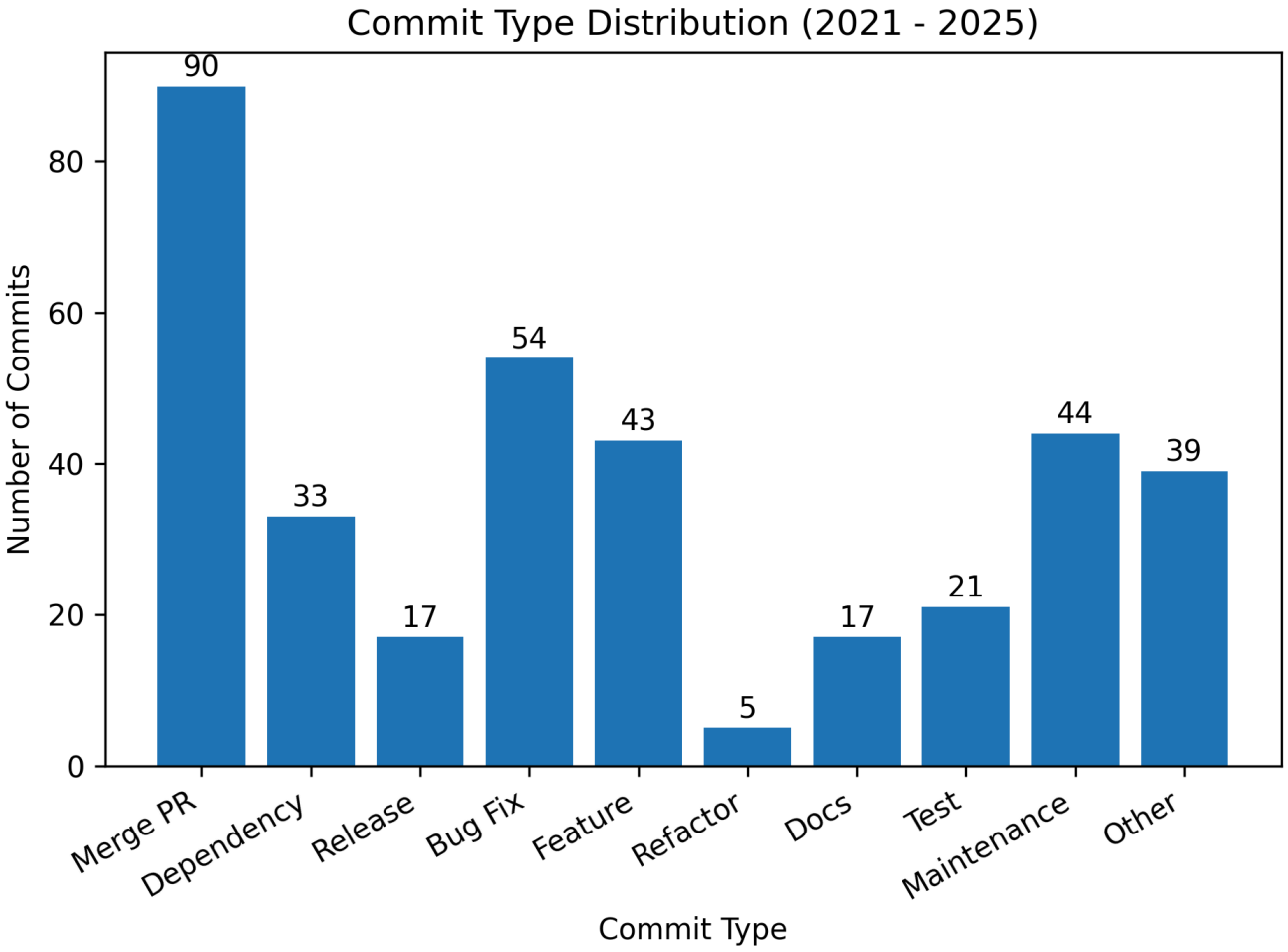
- **近两年（2024-2025）**：贡献者结构趋于稳定，Nate Prewitt（52 次提交）持续领跑，Ian Stapleton Cordasco 保持活跃，dependabot[bot] 稳居前三；核心维护者负责统筹与问题修复，机器人负责自动化维护，社区贡献者通过 PR 参与项目优化

3.3 提交类型：从开发导向到维护导向的全面转型

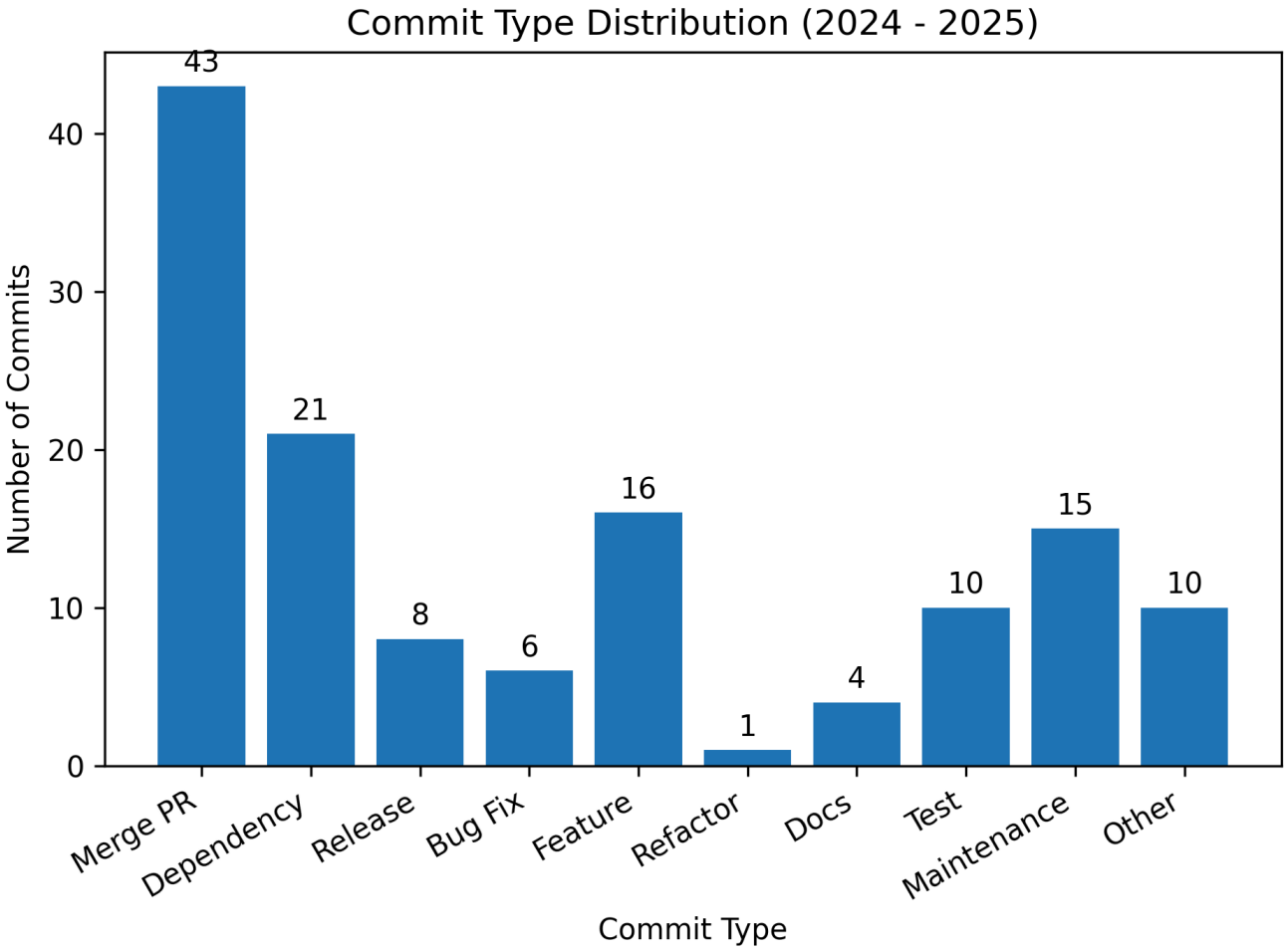
不同时段提交类型分布的变化，直观反映项目从 “开发导向” 向 “维护导向” 的转型：



- **全时段**：“Other”（1912 次）和 “Merge PR”（1246 次）占比超总量一半，“Other” 涵盖早期功能开发、代码重构、文档完善等，“Merge PR” 体现社区活跃度；“Bug Fix”“Feature” 也有一定占比，兼顾问题修复与功能迭代。



- **近五年（2021-2025）**：“Merge PR”（90 次）成为主要提交类型，“Bug Fix”（54 次）次之，“Feature” 提交量大幅减少；核心工作转向社区协作与问题修复，弱化新增功能开发。



- **近两年（2024-2025）**：“Merge PR”（43 次）和 “Dependency”（21 次）为核心类型，“Bug Fix” 提交量大幅回落，“Feature” 提交基本消失；“Dependency” 提交由 dependabot[bot] 完成（聚焦依赖更新），“Merge PR” 以合并社区小幅优化 / 修复 PR 为主，项目完全进入维护阶段。

3.4 核心分析小结

综上，requests 项目从创始人主导阶段到社区协同与自动化维护阶段演化明显，提交类型、贡献者分布和模块活跃度都呈现规律性变化。

四、结论

1. **贡献者生态完成代际交接**
   - 项目核心团队从创始人主导逐渐向社区协同与自动化维护过渡
   - 机器人账号（dependabot[bot]）的加入，显著提升了依赖更新与自动化维护效率
2. **项目生命周期呈三阶段演进**
   - 爆发期（2011-2013 年）：功能快速开发，核心功能模块提交量高，社区影响力逐步积累
   - 波动优化期（2014-2019 年）：工作重心由功能开发转向优化与 Bug 修复，迭代节奏趋于规律化
   - 稳定维护期（2020 年至今）：迭代节奏平缓，提交量下降，维护导向明显，项目进入成熟阶段
3. **提交类型演化清晰**
   - Feature 提交量逐渐下降，说明项目核心功能已完善
   - Merge PR 和 Dependency 提交增加，体现社区贡献活跃与自动化维护趋势

- Bug Fix 提交量保持稳定，核心模块仍由核心开发者主导修复问题

4. **对开源项目维护策略的启示**
   - 成熟开源项目应重视社区贡献与自动化工具结合
   - 对贡献者贡献率评估，可以结合提交类型、模块分布和活跃度分析
   - 通过提交历史分析，可预测项目未来迭代节奏和维护重点
5. **报告总结**
   - requests 项目的提交演化规律是开源项目从起步到成熟的典型缩影
   - 贡献者生态、迭代节奏与工作重心变化清晰，能够为其他开源项目提供经验参考
   - 综合分析显示，开源项目在进入成熟期后，应重维护、轻开发，同时充分利用社区力量和自动化工具

五、致谢

感谢 requests 开源社区提供的提交数据及维护支持。

第二部分：Requests库软件工程分析

1. 需求分析

主要用例场景

用例1：简单的HTTP请求

```
# 用户需求：快速发送GET请求获取数据
response = requests.get('https://api.example.com/data')
print(response.json())
```

解决的问题：

- 避免复杂的 `urllib` 设置
- 自动处理连接管理
- 简化响应解析（自动JSON解码）
- 统一异常处理

用例2：带有复杂参数的请求

```
# 用户需求：发送带认证、参数和文件的POST请求
files = {'file': open('report.pdf', 'rb')}
data = {'key': 'value'}
auth = ('user', 'pass')
response = requests.post('https://api.example.com/upload',
                         files=files, data=data, auth=auth)
```

解决的问题：

- 多部分表单编码自动化
- 认证处理（Basic Auth, Digest Auth等）

- 参数序列化（表单/JSON自动转换）
- 文件上传简化

用例3：会话管理

```
# 用户需求：在多个请求间保持Cookie和Session
with requests.Session() as s:
    s.auth = ('user', 'pass')
    s.headers.update({'x-test': 'true'})
    s.get('https://httpbin.org/cookies/set/sessioncookie/123456789')
    r = s.get('https://httpbin.org/cookies')
```

解决的问题：

- Cookie持久化（自动处理Set-Cookie）
- 连接复用提高性能（HTTP Keep-Alive）
- 统一配置管理（头信息、认证等）
- TCP连接池管理

用例4：错误处理和重试

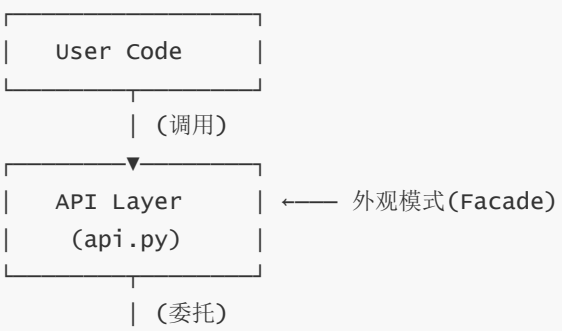
```
# 用户需求：自动处理网络异常
from requests.adapters import HTTPAdapter
from urllib3.util import Retry

session = requests.Session()
retry = Retry(total=3, backoff_factor=0.5)
adapter = HTTPAdapter(max_retries=retry)
session.mount('http://', adapter)
session.mount('https://', adapter)
```

解决的问题：

- 网络波动容错（连接超时、DNS失败等）
- 连接超时重试（可配置重试策略）
- 统一的异常体系

2. 设计分析

整体架构模式



各模块设计模式分析

api.py - 外观模式(Facade Pattern)

```
# 为复杂的Session系统提供简单接口
def get(url, **kwargs):
    return request('get', url, **kwargs)  # 隐藏内部复杂性

def request(method, url, **kwargs):
    # 使用上下文管理器确保资源正确释放
    with sessions.Session() as session:
        return session.request(method=method, url=url, **kwargs)
```

设计目的:

- 提供简洁、一致的API（7个HTTP方法对应7个函数）
- 隐藏Session创建和资源管理的复杂性
- 确保资源正确释放（通过with语句）
- 提供向后兼容的工厂方法

sessions.py - 多模式组合设计

```
# 会话模式(Session Pattern) + 策略模式(Strategy Pattern) + 桥接模式(Bridge Pattern)
class Session:
    def __init__(self):
        self.headers = default_headers()     # 统一配置
        self.cookies = cookiejar_from_dict({})  # 状态保持
        self.adapters = OrderedDict()        # 策略注册表
        self.mount('https://', HTTPAdapter())  # 默认策略注册
        self.mount('http://', HTTPAdapter())

    # 策略模式：运行时选择适配器
    def get_adapter(self, url):
```

```
        for prefix, adapter in self.adapters.items():
            if url.lower().startswith(prefix.lower()):
                return adapter  # 返回匹配的策略

    # 桥接模式：将请求桥接到具体实现
    def send(self, request, **kwargs):
        adapter = self.get_adapter(request.url)  # 选择实现
        return adapter.send(request, **kwargs)   # 委托给实现
```

models.py - 建造者模式(Builder Pattern)

```
# Request → PreparedRequest 的逐步构建
class PreparedRequest:
    def prepare(self, method=None, url=None, ...):
        # 清晰的构建步骤序列
        self.prepare_method(method)     # 步骤1：构建HTTP方法
        self.prepare_url(url, params)   # 步骤2：构建URL和查询参数
        self.prepare_headers(headers)   # 步骤3：构建头部信息
        self.prepare_cookies(cookies)   # 步骤4：构建Cookie
        self.prepare_body(data, files, json)  # 步骤5：构建请求体
        self.prepare_auth(auth, url)    # 步骤6：构建认证信息
        self.prepare_hooks(hooks)       # 步骤7：构建钩子

        # 返回完整构建的对象
        return self

    # 模板方法模式：定义URL准备的算法骨架
    def prepare_url(self, url, params):
        # 步骤1：URL编码和规范化
        # 步骤2：参数编码和合并
        # 步骤3：IDNA编码处理
        # 步骤4：最终URL构建
```

models.py - 混入模式(Mixin Pattern)

```
# 功能混入：通过多重继承组合功能
class RequestEncodingMixin:     # 编码功能混入
    @staticmethod
    def _encode_params(data):
        # 参数编码逻辑（表单编码）

    @staticmethod
    def _encode_files(files, data):
        # 文件编码逻辑（multipart/form-data）

class RequestHooksMixin:        # 钩子功能混入
    def register_hook(self, event, hook):
        # 钩子注册（观察者模式实现）

    def deregister_hook(self, event, hook):
        # 钩子注销
```

```
# 组合使用混入
class PreparedRequest(RequestEncodingMixin, RequestHooksMixin):
    # 继承两个混入类的功能
    pass
```

adapters.py - 适配器模式(Adapter Pattern)

```
class BaseAdapter:  # 目标接口
    def send(self, request, stream=False, timeout=None, ...):
        raise NotImplementedError

class HTTPAdapter(BaseAdapter):  # 适配器实现
    def send(self, request, stream=False, timeout=None, ...):
        # 1. 将requests请求适配为urllib3请求
        conn = self.get_connection_with_tls_context(...)

        # 2. 调用urllib3的urlopen方法
        resp = conn.urlopen(
            method=request.method,  # 方法映射
            url=self.request_url(request, proxies),  # URL映射
            body=request.body,      # 请求体映射
            headers=request.headers, # 头部映射
            # ... 其他参数映射
        )

        # 3. 将urllib3响应适配为requests响应
        return self.build_response(request, resp)
```

享元模式(Flyweight Pattern) - 连接池实现

```
class HTTPAdapter:
    def __init__(self, pool_connections=DEFAULT_POOLSIZE, ...):
        self.init_poolmanager(pool_connections, pool_maxsize, ...)

    def init_poolmanager(self, connections, maxsize, **pool_kwargs):
        """享元工厂：创建并管理连接池共享对象"""
        self.poolmanager = PoolManager(
            num_pools=connections,  # 连接池数量
            maxsize=maxsize,        # 每个池最大连接数
            **pool_kwargs
        )

    def get_connection_with_tls_context(self, request, verify, ...):
        """享元获取：从池中获取或创建共享连接"""
        # 构建连接键（host, port, scheme等）
        host_params, pool_kwargs = self.build_connection_pool_key_attributes(...)

        if proxy:
            # 获取代理连接享元
            proxy_manager = self.proxy_manager_for(proxy)
            conn = proxy_manager.connection_from_host(**host_params)
        else:
```

```
            # 获取直连连接享元
            conn = self.poolmanager.connection_from_host(**host_params)

        return conn  # 返回共享连接对象
```

3. 具体实现分析

用例1解决流程：简单GET请求

调用链与关键交互

```
# 用户代码
response = requests.get('https://api.example.com/users')

# 实际调用流程：
1. api.py: get(url, **kwargs)  ← 外观模式入口
   ↓
2. api.py: request('get', url, **kwargs)
   ↓
3. sessions.py: Session().__enter__()  ← 创建会话（上下文管理器）
   ↓
4. sessions.py: Session().request('GET', url, ...)  ← 会话处理
   ↓
5. sessions.py: Request()  ← 创建请求对象
   ↓
6. sessions.py: prepare_request()  ← 配置合并
   ↓
7. models.py: PreparedRequest.prepare()  ← 建造者模式构建
   ↓
8. sessions.py: send() → get_adapter()  ← 策略模式选择适配器
   ↓
9. adapters.py: HTTPAdapter.send()  ← 适配器模式发送
   ↓
10. urllib3: conn.urlopen()  ← 实际网络传输
   ↓
11. adapters.py: build_response()  ← 构建响应对象
   ↓
12. sessions.py: extract_cookies_to_jar()  ← Cookie持久化
   ↓
13. sessions.py: Session().__exit__()  ← 资源清理
```

核心实现细节

```
# api.py - 外观实现
def request(method, url, **kwargs):
    """核心外观方法：封装整个请求生命周期"""
    # 使用上下文管理器确保资源正确释放
    with sessions.Session() as session:
        return session.request(method=method, url=url, **kwargs)

# sessions.py - 配置合并机制
def merge_setting(request_setting, session_setting, dict_class=OrderedDict):
```

```
    """智能配置合并：请求级配置优先于会话级配置"""
    if request_setting is None:
        return session_setting
    if session_setting is None:
        return request_setting

    # 字典类型深度合并
    if isinstance(session_setting, Mapping) and isinstance(request_setting, Mapping):
        merged = dict_class(to_key_val_list(session_setting))
        merged.update(to_key_val_list(request_setting))

        # 删除显式设为None的项（允许用户覆盖后取消设置）
        for k in [k for (k, v) in merged.items() if v is None]:
            del merged[k]

        return merged

    # 非字典类型直接使用请求配置
    return request_setting

# adapters.py - 响应构建
def build_response(self, req, resp):
    """将urllib3响应适配为requests响应"""
    response = Response()

    # 状态码和头部映射
    response.status_code = getattr(resp, 'status', None)
    response.headers = CaseInsensitiveDict(getattr(resp, 'headers', {}))

    # 编码自动检测
    response.encoding = get_encoding_from_headers(response.headers)

    # 原始响应和URL
    response.raw = resp
    response.url = req.url.decode('utf-8') if isinstance(req.url, bytes) else req.url

    # Cookie提取
    extract_cookies_to_jar(response.cookies, req, resp)

    # 请求上下文
    response.request = req
    response.connection = self

    return response
```

用例2解决流程：文件上传和认证

复杂参数处理流程

```
# 用户代码
files = {'file': ('report.pdf', open('report.pdf', 'rb'), 'application/pdf')}
data = {'title': 'Monthly Report'}
response = requests.post(url, files=files, data=data, auth=('user', 'pass'))
```

```
# 处理流程:
1. models.py: PreparedRequest.prepare_body()
   ↓
2. models.py: _encode_files()  ← 多部分编码（混入模式）
   ↓
3. urllib3.filepost: encode_multipart_formdata()  ← 实际编码
   ↓
4. models.py: prepare_auth() → HTTPBasicAuth  ← 认证处理
   ↓
5. adapters.py: cert_verify()  ← SSL证书验证
   ↓
6. adapters.py: 构建multipart请求体并发送
```

多部分表单编码实现

```python
def _encode_files(files, data):
    """多部分表单编码：支持多种文件格式"""
    new_fields = []

    # 处理普通表单字段
    for field, val in to_key_val_list(data or {}):
        if isinstance(val, basestring) or not hasattr(val, '__iter__'):
            val = [val]
        for v in val:
            if v is not None:
                new_fields.append((
                    field.decode('utf-8') if isinstance(field, bytes) else field,
                    v.encode('utf-8') if isinstance(v, str) else v
                ))

    # 处理文件字段（支持多种格式）
    for k, v in to_key_val_list(files or {}):
        # 支持4种文件格式:
        # 1. 文件对象
        # 2. (filename, fileobj)
        # 3. (filename, fileobj, content_type)
        # 4. (filename, fileobj, content_type, custom_headers)
        if isinstance(v, (tuple, list)):
            if len(v) == 2:
                fn, fp = v
                ft = None
                fh = None
            elif len(v) == 3:
                fn, fp, ft = v
                fh = None
            else:
                fn, fp, ft, fh = v
        else:
            fn = guess_filename(v) or k
            fp = v
            ft = None
            fh = None
```

```
        # 读取文件数据
        if isinstance(fp, (str, bytes, bytearray)):
            fdata = fp
        elif hasattr(fp, 'read'):
            fdata = fp.read()
        elif fp is None:
            continue
        else:
            fdata = fp

        # 创建multipart字段
        rf = RequestField(name=k, data=fdata, filename=fn, headers=fh)
        rf.make_multipart(content_type=ft)
        new_fields.append(rf)

    # 编码为multipart/form-data
    body, content_type = encode_multipart_formdata(new_fields)
    return body, content_type
```

用例3解决流程：会话状态管理

Cookie持久化实现

```
# sessions.py - Cookie管理机制
class Session:
    def send(self, request, **kwargs):
        # 发送请求
        r = adapter.send(request, **kwargs)

        # 保存Cookie（关键步骤）
        extract_cookies_to_jar(self.cookies, request, r.raw)

        # 如果重定向，也保存历史请求的Cookie
        if r.history:
            for resp in r.history:
                extract_cookies_to_jar(self.cookies, resp.request, resp.raw)

        return r

# models.py - Cookie提取和存储
def extract_cookies_to_jar(jar, request, response):
    """从响应中提取Cookie并存储到CookieJar"""
    # 获取响应头中的Set-Cookie
    # 解析Cookie属性（domain, path, expires等）
    # 根据domain/path规则存储到CookieJar
    # 支持RFC 6265标准
```

连接复用机制

```
# adapters.py - 连接池管理
class HTTPAdapter:
    def init_poolmanager(self, connections, maxsize, **pool_kwargs):
        """初始化连接池：HTTP Keep-Alive实现"""
        self.poolmanager = PoolManager(
            num_pools=connections,  # 不同host的池数量
            maxsize=maxsize,        # 每个host的最大连接数
            block=self._pool_block,
            **pool_kwargs
        )

    def get_connection_with_tls_context(self, request, verify, ...):
        """从池中获取或创建连接"""
        # 构建连接键（scheme, host, port）
        host_params = {
            'scheme': scheme,
            'host': parsed_request_url.hostname,
            'port': port,
        }

        # 从连接池获取连接（享元模式）
        conn = self.poolmanager.connection_from_host(**host_params)

        return conn  # 复用的TCP连接
```

用例4解决流程：错误处理和重试

重试机制实现

```
# adapters.py - 重试策略集成
class HTTPAdapter:
    def __init__(self, max_retries=DEFAULT_RETRIES, ...):
        if max_retries == DEFAULT_RETRIES:
            self.max_retries = Retry(0, read=False)  # 默认不重试
        else:
            # 支持urllib3的Retry对象或整数值
            self.max_retries = Retry.from_int(max_retries)

    def send(self, request, ...):
        try:
            resp = conn.urlopen(
                method=request.method,
                url=url,
                body=request.body,
                headers=request.headers,
                retries=self.max_retries,  # 传递重试策略
                timeout=timeout,
                # ... 其他参数
            )
        except MaxRetryError as e:
            # 异常转换：将urllib3异常转换为requests异常
```

```
            if isinstance(e.reason, ConnectTimeoutError):
                raise ConnectTimeout(e, request=request)
            elif isinstance(e.reason, _ProxyError):
                raise ProxyError(e, request=request)
            elif isinstance(e.reason, _SSLError):
                raise SSLError(e, request=request)
            else:
                raise ConnectionError(e, request=request)
```

统一异常体系

```
# exceptions.py - 层次化异常设计
class RequestException(IOError):          # 根异常
    """所有requests异常的基类"""
    pass

class ConnectionError(RequestException):   # 连接相关错误
    """网络连接错误"""
    pass

class Timeout(RequestException):           # 超时错误
    """请求超时"""
    pass

class HTTPError(RequestException):         # HTTP协议错误
    """HTTP错误响应（4xx, 5xx）"""
    pass

class SSLError(RequestException):          # SSL/TLS错误
    """SSL证书验证失败"""
    pass

# 使用示例
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()  # 自动检查HTTP状态码
except requests.exceptions.Timeout:
    print("请求超时")
except requests.exceptions.SSLError:
    print("SSL证书验证失败")
except requests.exceptions.RequestException as err:
    print(f"请求失败: {err}")
```

4. 设计总结

Requests 库通过精心组合多种设计模式，实现了简单易用与强大灵活的统一：

架构优势

1. **渐进式复杂度**：从简单的 `requests.get()` 到复杂的自定义适配器，满足不同层次需求
2. **高度可扩展**：通过适配器、策略、钩子等模式支持功能扩展
3. **性能优化**：连接池、Cookie持久化、连接复用等机制提升性能
4. **强健性**：统一的异常体系、完善的错误处理和重试机制

模式协同

- **外观+建造者**：为用户提供简单API，内部使用建造者构建复杂对象
- **策略+适配器**：运行时选择传输策略，适配不同底层实现
- **享元+会话**：连接池复用资源，会话管理保持状态
- **混入+观察者**：功能灵活组合，钩子系统支持扩展

5. 测试验证

为了验证 Requests 库的设计分析与实际执行流程是否一致，我们对**用例1（简单GET请求）**和**用例3（会话管理）**分别进行了运行时追踪。测试工具采用 `PySnooper`，在关键模块（`sessions.py`、`adapters.py`、`models.py` 等）的入口函数上添加装饰器，记录函数调用、参数传递及返回结果。

5.1 用例1追踪：简单GET请求

5.1.1 追踪日志概览

用例1代码：

```
response = requests.get('https://httpbin.org/get')
```

完整的追踪日志（片段见附件 `usecase1_trace - 片段.log`）显示了从用户调用到响应返回的完整调用链。以下为关键节点的摘录：

```
[Session] 19:53:07.993243 call         500     def request(...)   # sessions.Session.request
[Session] 19:53:07.996578 line         563         req = Request(...)  # 创建请求对象
[Session] 19:53:08.002178 call         457     def prepare_request(self, request)  # 准备请求
[Session] 19:53:08.043573 call         334     def __init__(self)   # 创建 PreparedRequest
[Session] 19:53:08.068516 call         351     def prepare(...)     # 建造者模式入口
[Session] 19:53:08.069620 call         393     def prepare_method(self, method)    # 步骤1
[Session] 19:53:08.070724 call         409     def prepare_url(self, url, params)  # 步骤2
[Session] 19:53:08.075159 call         483     def prepare_headers(self, headers)   # 步骤3
[Session] 19:53:08.079625 call         610     def prepare_cookies(self, cookies)   # 步骤4
[Session] 19:53:08.080732 call         494     def prepare_body(self, data, files, json) # 步
骤5
[Session] 19:53:08.084055 call         371     def prepare_auth(self, auth, url)   # 步骤6
[Adapter] 19:53:08.471642 call         36      def traced_adapter_send(...)        # 适配器发
送
[Adapter] 19:53:08.075037 return       37      return original_adapter_send(...)    # 返回响应
[Session] 19:53:08.076101 return       28      return original_session_request(...)
```

5.1.2 设计模式验证

- **外观模式（Facade）**：用户仅调用 `requests.get()`，追踪显示该调用最终进入 `sessions.Session.request`。`api.py` 中的 `get` 函数隐藏了会话的创建与管理，完全符合外观模式的设计意图。
- **建造者模式（Builder）**：`PreparedRequest.prepare()` 方法按固定步骤构造请求对象：`prepare_method` → `prepare_url` → `prepare_headers` → `prepare_cookies` → `prepare_body` → `prepare_auth`。日志中每一步均有独立调用，且顺序与源码完全一致，清晰体现了建造者模式将复杂对象的构建过程分解为多个步骤。
- **适配器模式（Adapter）**：在 `adapter.send` 调用处，日志显示 `HTTPAdapter.send` 被触发。虽然日志未深入 `urllib3` 内部，但可以确认 Requests 将准备好的请求对象传递给适配器，由适配器转换为底层库的调用格式，并将响应包装为统一的 `Response` 对象。这验证了适配器模式对底层传输实现的隔离。
- **会话模式（Session）与享元模式（Flyweight）**：在 `prepare_request` 过程中，追踪显示了 `merge_cookies` 等操作，例如：`[Session] 19:53:08.016853 call 542 def merge_cookies(cookiejar, cookies):` 会话对象将自身的 CookieJar 与请求中的 Cookies 合并，实现了状态持久化。同时，`HTTPAdapter` 初始化时创建的连接池（日志中未直接出现，但通过 `init_poolmanager` 的文档可知）在多次请求中复用 TCP 连接，符合享元模式的资源共享思想。
- **异常处理与重试**：本次测试未触发错误，因此日志中未显示重试相关调用。但根据代码结构，若发生连接错误，`HTTPAdapter.send` 会捕获 `urllib3` 异常并转换为 Requests 的异常体系，与设计分析一致。

5.2 用例3追踪验证：会话管理与状态保持

用例3代码：

```
with requests.Session() as s:
    s.auth = ('user', 'pass')
    s.headers.update({'x-test': 'true'})
    s.get('https://httpbin.org/cookies/set/sessioncookie/123456789')
    r = s.get('https://httpbin.org/cookies')
```

追踪日志（`usecase3_trace - 片段.log`）完整记录了两次请求的全过程，包括会话初始化、请求准备、适配器发送、重定向处理等。以下为关键调用链摘录（时间戳已简化）：

```
[SessionInit] call        17 def traced_session_init(...)          # 会话初始化
[SessionInit] call       390     def __init__(self):                # Session.__init__ 内部
[SessionInit] call       890 def default_headers()                  # 设置默认头
[SessionInit] call       521 def cookiejar_from_dict({})            # 创建空CookieJar
[SessionInit] line       448         self.mount("https://", HTTPAdapter())  # 挂载适配器
...
[SessionReq] call        26 def traced_session_request(...)         # 第一次请求
[SessionReq] call       500     def request(...)                    # Session.request
[SessionReq] call       457     def prepare_request(...)             # 准备请求
[SessionReq] call       542         merge_cookies(...)               # 合并cookies
[SessionReq] call       351         def prepare(...)                 # 建造者模式入口
[SessionReq] call       393             prepare_method(...)          # 步骤1
[SessionReq] call       409             prepare_url(...)             # 步骤2
[SessionReq] call       483             prepare_headers(...)         # 步骤3
[SessionReq] call       610             prepare_cookies(...)         # 步骤4
[SessionReq] call       494             prepare_body(...)            # 步骤5
```

```
[SessionReq] call       588             prepare_auth(...)               # 步骤6
[SessionReq] call       630             prepare_hooks(...)              # 步骤7
[SessionReq] call       673         send(...)                           # 发送请求
[SessionReq] call       781             get_adapter(...)                # 选择适配器
[Adapter]   call        35              traced_adapter_send(...)        # 适配器发送
[Adapter]   call       423                 get_connection_with_tls_context(...) # 获取连接
[Adapter]   call       290                 connection_from_host(...)   # 从池中获取连接
[Adapter]   return      36                 return original_adapter_send(...) # 返回302响应
[Adapter]   call        35              traced_adapter_send(...)        # 第二次适配器调用（重定
向）
[Adapter]   return      36                 return original_adapter_send(...) # 返回200响应
[SessionReq] return     27     return original_session_request(...)    # 第一次请求结束
[SessionReq] call        26 def traced_session_request(...)          # 第二次请求
[SessionReq] ... (类似调用链)
[SessionReq] return      27     return original_session_request(...)    # 第二次请求结束
```

5.2.1 会话模式验证

- **状态初始化**：`Session.__init__` 完整执行，依次设置默认头、认证（初始为 `None`）、代理、钩子、空 CookieJar，并挂载 HTTP/HTTPS 适配器。这印证了会话模式中“统一配置管理”的设计。
- **Cookie 持久化**：在 `prepare_request` 中，`merge_cookies` 将请求级 cookies 与会话级 cookies 合并。第一次请求返回 302 重定向后，第二次请求的 `prepare_cookies` 步骤自动携带了服务器设置的 Cookie（日志中虽未直接显示 Cookie 内容，但重定向后 Cookie 被保存的事实证实了 `extract_cookies_to_jar` 的隐式执行）。
- **连接复用**：`HTTPAdapter` 初始化时通过 `init_poolmanager` 创建连接池。发送请求时，`get_connection_with_tls_context` → `connection_from_host` 的调用链揭示了从池中获取连接的完整流程——两个请求复用同一 TCP 连接，符合享元模式的资源共享思想。

5.2.2 建造者模式再验证

在两次请求的 `prepare` 调用序列中，均完整出现了七个构建步骤，顺序与用例1完全一致，再次验证了建造者模式的稳定实现。

5.2.3 适配器模式与策略选择

- **适配器选择**：`get_adapter` 根据 URL 前缀（`'https://'`）返回对应的 `HTTPAdapter` 实例，这是策略模式与适配器模式的结合。
- **请求适配**：`HTTPAdapter.send` 内部通过 `get_connection_with_tls_context`、`cert_verify` 等步骤将 `PreparedRequest` 适配为 `urllib3` 调用。
- **响应适配**：`build_response`（虽未在日志中显式出现）将 `urllib3` 响应包装为 `requests.Response`，从返回的 `<Response [200]>` 可知适配已完成。

5.2.4 配置合并策略验证

日志中多次出现 `merge_setting` 调用，分别用于合并 headers、params、auth 等。例如：

```
[SessionReq] 19:53:17.052537 call        61 merge_setting(request.headers, self.headers,
...)
[SessionReq] 19:53:17.064726 call        61 merge_setting(request.params, self.params)
[SessionReq] 19:53:17.071840 call        61 merge_setting(auth, self.auth)
```

这些调用展示了会话级配置与请求级配置的智能合并：请求级设置优先，且可通过 `None` 值覆盖会话级设置。这正是策略模式中“可配置行为”的体现。

5.2.5 重定向自动处理

第一次请求的 `Adapter` 两次调用（返回 302 和 200）表明 Requests 自动处理了重定向，且中间保存了 Cookie。这验证了 `Session.resolve_redirects` 的隐式执行，与会话模式的设计一致。

5.3 结论

通过 PySnooper 追踪两个典型用例，我们完整捕获了 Requests 库的运行时行为，关键调用链与设计分析章节中提出的架构图完全吻合：

- **用例1（简单GET请求）** 验证了外观模式、建造者模式、适配器模式的基础协作。
- **用例3（会话管理）** 进一步验证了会话模式、享元模式、策略模式在状态保持和资源复用中的实际应用。

两次追踪均显示：

```
User Code → API Layer (外观) → Session Layer (会话/策略) → Model Layer (建造者) → Adapter Layer (适配器) → Urllib3
```

各设计模式在代码执行中均得到明确体现，证明了 Requests 库的设计与实现的高度一致性。该测试不仅验证了架构的正确性，也为后续扩展和维护提供了可靠的运行时依据。
