CCF ChinaSoft
CCF 中国软件大会

Python包应用程序编程接口破坏性变更检测系统

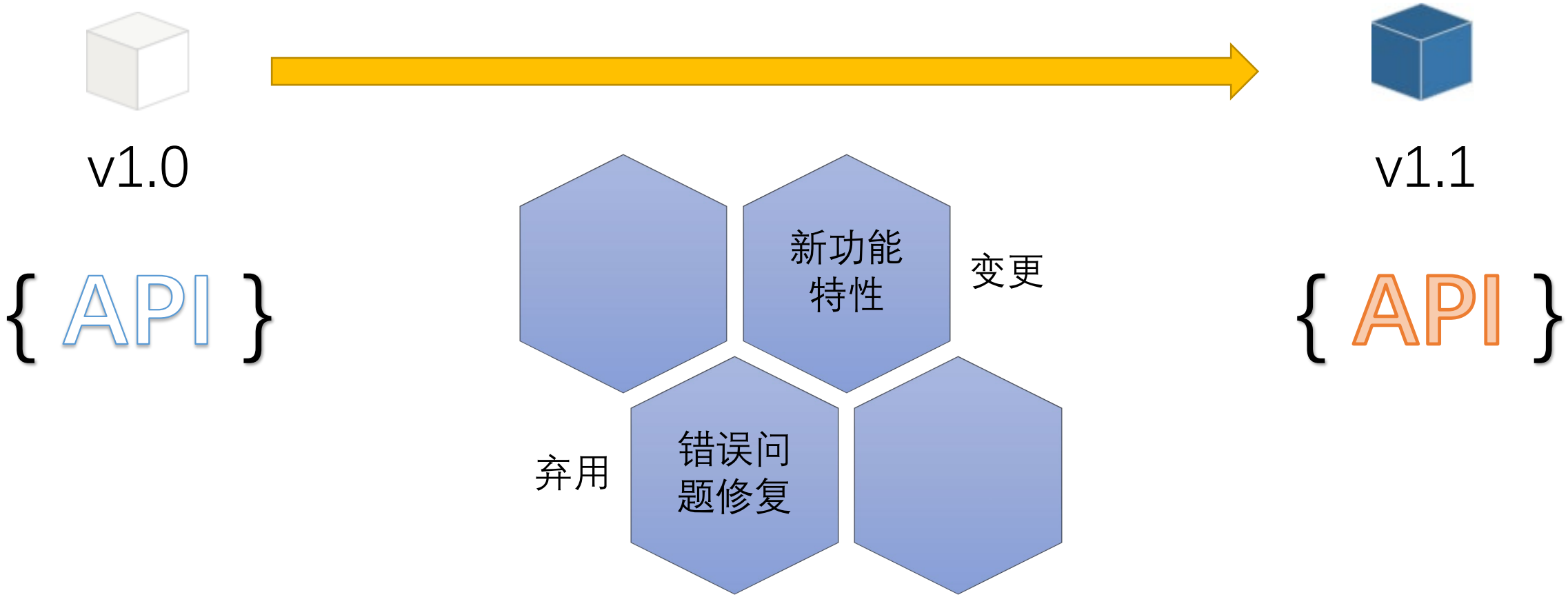
报告人：杜星亮

计算机软件新技术国家重点实验室
南京大学 计算机科学与技术系

报告日期：2022.11.25

问题背景

频繁的API演化和变更

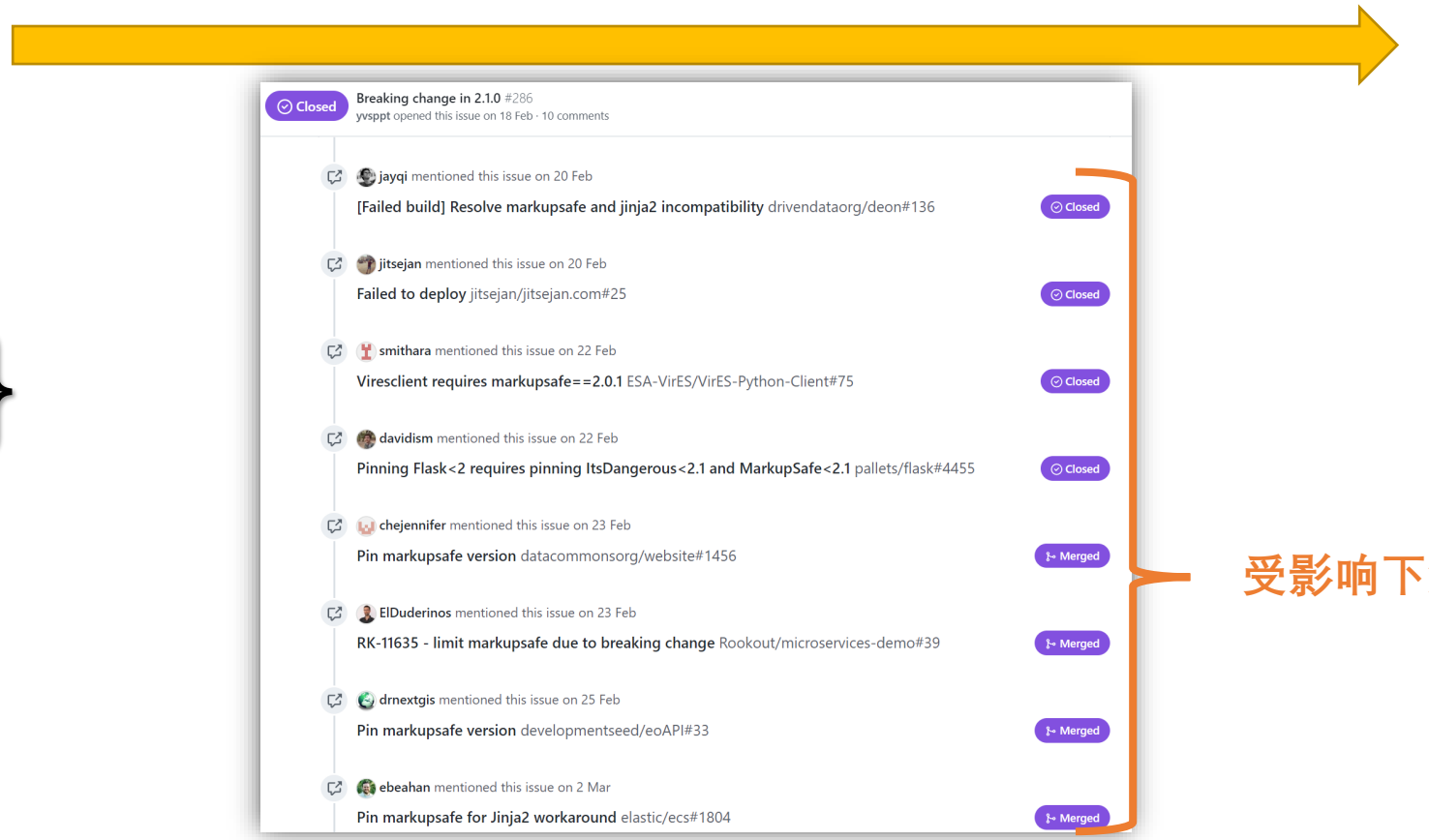


问题背景

向后不兼容变更破坏下游代码

v1.0

{ API }



v1.1

{ API }

受影响下游代码

Breaking change in 2.1.0 · Issue #286 · pallets/markupsafe (github.com)

向后不兼容变更增大维护成本

v1.0

{ API }

v1.1

{ API }

Breaking change in 0.5.5 #76

Closed lkollar opened this issue on 9 Mar 2021 · 2 comments

lkollar commented on 9 Mar 2021

We have a tool using `resolvelib` which implements the `AbstractProvider` interface. 0.5.5 broke the API with `915363f` changing the signature of `identify`.

Following semver, technically such changes are to be expected, but I also saw #74 warning about upcoming API breakage in 0.6.0, so I'm wondering what the policy around API changes is in `resolvelib`.

uranusjr commented on 9 Mar 2021 Member

I accidentally released the wrong branch (master) with the kwargs change 😖 I realised it just now as well (I'm pulling in changes in pypa/pip#9699).

My rule around versioning for now is to do breaking changes in the second digit (minor version), at least until 1.0 is reached. I think I'll just yank 0.5.5. Thanks for the report!

2

撤回已发布的向后不兼容版本

Breaking change in 0.5.5 · Issue #76 · sarugaku/resolvelib (github.com)

问题背景——Python包API破坏性变更检测

使用 AexPy 自动检测

向后 兼容 / 破坏？

v1.0

{ API }

2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)

AexPy: Detecting API Breaking Changes in Python Packages

Xingliang Du
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
xingliangdu@smail.nju.edu.cn

Jun Ma
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
majun@nju.edu.cn

***Abstract*—With the popularity of the Python language, community developers create and maintain a lot of third-party packages. APIs change frequently during the package evolving. Package developers need keeping backward compatibility of APIs to avoid breaking client code. Detecting breaking changes in Python packages is challenging because of Python's dynamic features and flexible designs, such as dynamic typing, API aliases, confusing public boundary and flexible argument passing. Despite the language's popularity, there have been few tools aiming to detect breaking changes, and existing approaches lack sufficient consideration of the mentioned challenges, resulting in imprecise and incomplete detection. Briefly, we propose an API-model-based systematic approach to address this problem. We design a Python-specific API model and classify API changes in different breaking levels. Based on the model, we obtain APIs with their types by a robust hybrid analysis and detect graded changes by checking constraints on API pairs. We implement a prototype, AexPy. Thanks to the more comprehensive model and hybrid analysis adopted, AexPy outperforms the state-of-the-art techniques for Python with an 86.9% recall on a dataset of 61 known breaking changes. Besides, AexPy detects 405 (manually verified) high and medium breaking changes with a precision of 93.5% on the latest versions of 45 packages. Specifically, in addition to 291 documented breaking changes, AexPy detects 114 undocumented changes. We report 63 undocumented breaking changes to active package developers, and 31 have been confirmed.**

***Index Terms*—Python, application programming interface, backward compatibility, breaking change**

I. Introduction

The convenient standard library and the large active third-party package[1] ecology of Python provide tools suited to many tasks (e.g., deep learning, automatic scripts, data processing) and help developers to achieve their ideas efficiently. However, during the evolution of a package, the package's developer may redesign data structures, change operations, or adjust specification due to fixing bugs or providing new features. Some of these changes may break client code, so called "breaking changes" [2]. One common change case is application programming interface (API) changes. APIs describe the public interfaces of a package, which are contracts that clients rely on [3], so an API change could affect clients. In the evolving of commonly used Python framework packages, more than 40% API changes are breaking [4], which is larger than static languages [5], [6]. These breaking changes break backward compatibility, decrease the evolving stability, and might cause potential bugs in clients.

Detecting breaking changes in Python packages is challenging because of Python's dynamic features and flexible designs, such as dynamic module importing, dynamic typing [7], multiple API references, and flexible argument passing [8], which altogether increase API complexity and introduce a variety of API changes. Existing approaches, such as pidiff [9] and PyCompat [4], lack sufficient consideration of these challenges, which leads to imprecise and incomplete results.

In this paper, we propose an automatic and systematic approach to detect API breaking changes in Python packages, which helps Python developers to evolve packages reliably. Our approach works in three major steps, 1) extracting APIs from different versions of Python packages, 2) comparing APIs to detect changes in different patterns, and 3) evaluating backward compatibility of changes to grade breaking levels.

To do so, we propose a model of Python package APIs, taking into consideration of primary features of Python (e.g., inheritance, argument passing, aliases, types). Specially, we model the signature types of functions to address the challenge of dynamic typing. Based on the model, we combine dynamic reflection with static analysis for API discovery and information enrichment to obtain a detailed API description, including types, parameters, aliases, and inheritance. Then we systematically classify API changes into 42 patterns. We design an entry pairing algorithm to adapt to Python's name resolution and a constraint-based checking algorithm to detect changes automatically. Finally, for practicality, we grade changes into four breaking levels according to the API scope, change pattern, and change content, indicating different severities of the changes. Specially, we use subset relationship between types to model compatibility of type changes.

We implement the proposed approach in a prototype named AexPy. We evaluate the effectiveness and efficiency of the tool by applying it to detect known/unknown breaking changes of different packages. Specifically, we compare AexPy with two existing Python API breaking change detectors, pidiff [9] and PyCompat [4]. AexPy benefits from the detailed model and hybrid analysis, and achieves almost 50% improvement on recall with comparable time performance. Meanwhile, in latest versions of 45 packages, AexPy detects high/medium breaking changes in 43 packages with a manually verified precision of 93.5%. So far, we have received 50 responses of 63 reported undocumented breaking changes, 31 of which are confirmed, indicating our approach is practically useful in real world.

To summarize, this work makes three major contributions:

- We systematically design a Python package API model and an API change classification. We grade changes into four breaking levels to adapt to the flexibility of Python APIs, and improve practicality of the model. (Section III)
- We propose an automatic and systematic approach for API breaking change detection in Python packages. To address challenges from Python's dynamic features and flexible designs, our approach extracts APIs through dynamic reflection and static analysis, and detects breaking changes by a constraint-based checking algorithm. To our best knowledge, we propose the first approach to detect type breaking changes in Python packages, where we check change compatibility by subset relationship between statically constructed types. (Section IV)
- We implement a prototype AexPy, to detect breaking changes in real-world packages. We evaluate AexPy on collected 61 known breaking changes and latest versions of 45 packages. Our approach achieves 86.9% recall, outperforming existing tools, and detects high and medium breaking changes in the latest versions with 93.5% precision. Of the 63 reported changes, 31 have been confirmed by package developers. (Section V)

The rest of this paper is organized as follows. Section II introduces the background and challenges on breaking change detection in Python packages. Section III depicts the design of our API model and the classification of breaking changes. Section IV details our detection method as well as our prototype AexPy for detecting breaking changes. Section V describes our evaluations, compared with existing approaches. Section VI discusses strengths, limitations, and validity. After giving an overview on related works in Section VII, we conclude this paper and discuss future works in Section VIII.

II. Background

A. API Backward Compatibility

For a software package, its APIs describe and prescribe the "expected behaviors", which contain descriptions about data structures, and operations on such data [3]. Backward compatibility is a property that the system allows interoperating with an older legacy system, e.g., dealing with inputs from the old system. Modifying a system in a way that does not follow this property is called "breaking" backward compatibility, i.e., breaking change [2]. Backward compatibility of a package means that clients can use the new version of the package in the same ways and get the same behaviors as the old version, i.e., the package have compatible APIs on syntax and semantic [2]. API breaking changes would cause compilation or runtime problems in client code. For example, signature changes in Java APIs would cause compilation errors, and function removals in shared objects (dynamic link libraries) for C/C++ could cause runtime linking errors.

B. Breaking Changes in Python Packages

Python is an interpreted, dynamically-typed, general-purpose programming language, which consistently ranks as one of the most popular programming languages [10]–[13]. Python supports multiple programming paradigms, including procedural, object-oriented and functional programming. Python Package Index (PyPI) [1], the official repository for third-party Python packages, contains over 329,000 packages by April 2022. These third-party packages cover a wide range of functionality, such as automation, machine learning, networking, scientific computing, web frameworks, and so on.

Breaking changes in Python packages happen frequently and have extensive and expensive impacts. Zhang *et al.* [4] investigated six Python frameworks and their clients, finding that breaking changes occurred more frequently than Java, and the changes affected more than half of the clients. They also investigated 409 compatibility issues on GitHub, and showed that 405 of them caused crashes at runtime, which showed the extensive and expensive effects on breaking changes.

Detecting breaking changes in Python packages is different and challenging compared to traditional programming languages. For example, Python developers often define instance attributes by assignment statements in constructors instead of the class body in C++/Java. Zhang *et al.* [4] found that the Python APIs have different evolution patterns on parameters. Peng *et al.* [14] studied language feature usage in 35 popular Python projects and found that inheritance, decorators, positional-or-keyword parameters are frequently used.

CPython [15] is the standard Python implementation. We learn about the Python language with CPython's runtime features and summarize following four challenges on API breaking change detection in Python packages.

1) Dynamic Language Features: It is difficult to collect API metadata statically because of Python's dynamic language features. Dynamic programming languages are a class of high-level programming languages, which at runtime execute many common programming behaviors that static programming languages perform during compilation (e.g., extending objects and definitions). Python provides language-level mechanisms such as decorators, metaclasses [8], class hooks to modify classes dynamically. These dynamic features provide flexibility to programmers while increasing difficulty of static analysis.

Besides, the dynamic type system in Python brings challenges to type compatibility checking. Firstly, CPython only checks type compatibility at runtime, so type-related errors occur during executing but cannot be easily detected during programming, especially between two versions. Secondly, Python has a complex type system and no enforcing type declarations, introducing complex and various type changes. Python allows duck-typing, i.e., an object is of a given type

This work was supported by Natural Science Foundation of China (Grant No. 62025202). The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Jun Ma is the corresponding author.

[1]We use the word "package" to represent "library", to keep consistence with Python's naming habit, e.g., the Python Package Index [1].

2332-6549/22/$31.00 ©2022 IEEE
DOI 10.1109/ISSRE55969.2022.00052

v1.1

{ API }

方法原理——主要挑战

动态语言的特性

- 发生在运行时的程序行为
- 支持“鸭子类型”的动态类型系统

复杂的API引用关系

- 导入和重命名
- 同一API具有不同名称

模糊的API可见性

- 缺少访问修饰符和相关的完整语言机制
- 自定义命名约定下，不同别名具有不同可见性

灵活的参数传递

- 必需或可选参数
- 位置、关键字、变长参数

方法原理——流程概览

API 提取

变更检测

破坏性分级

- API 模型
- 动态反射
- 静态分析

- 变更分类
- API和参数配对
- 基于约束的比较算法

- 破坏性级别
- API范围和变更内容
- 类型变更

详细的API模型

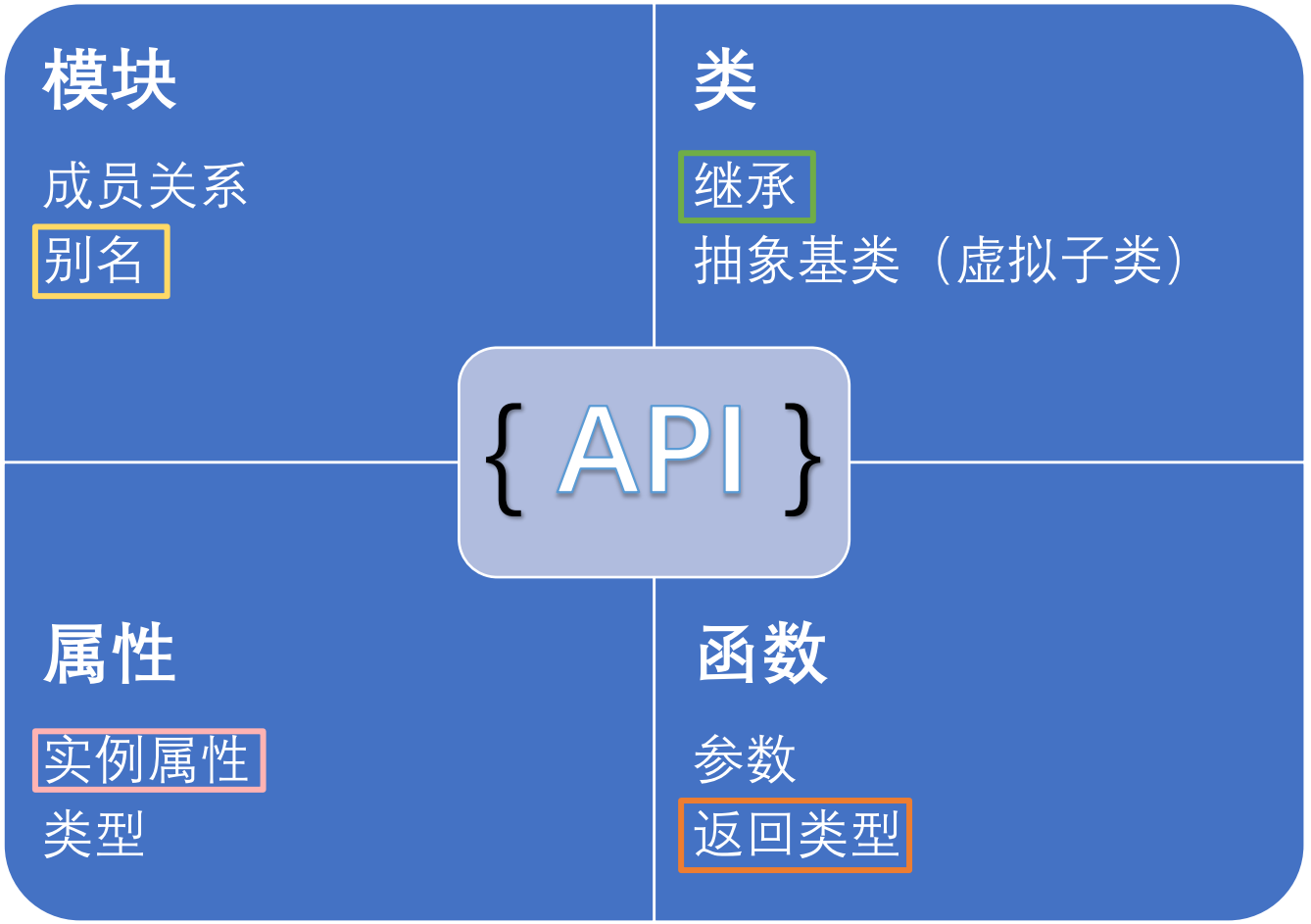
混合程序分析

配对和约束检测

破坏性分级

方法原理——API建模与提取


```python
from typing import Optional as opt

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```



方法原理——API建模与提取


```python
from typing import Optional

def func(a, b, /, c = []): pass
def _share(self): print(type(self))

class A:
    typeme = _share

class B:
    typeme = _share
    def g(self, c: "opt[B]" = None) -> "B | None":
        return c

    @property
    def x(self): return self._x
    @x.setter
    def x(self, val: "int"): self._x = val

    @staticmethod
    def h(*ar, **kw) -> str: return str(kw["v"])

class C(list, B):
    pass
```

- 广度优先搜索
- 收集运行时对象信息

动态反射

静态分析

- 遍历抽象语法树
- 从Mypy中获取类型信息

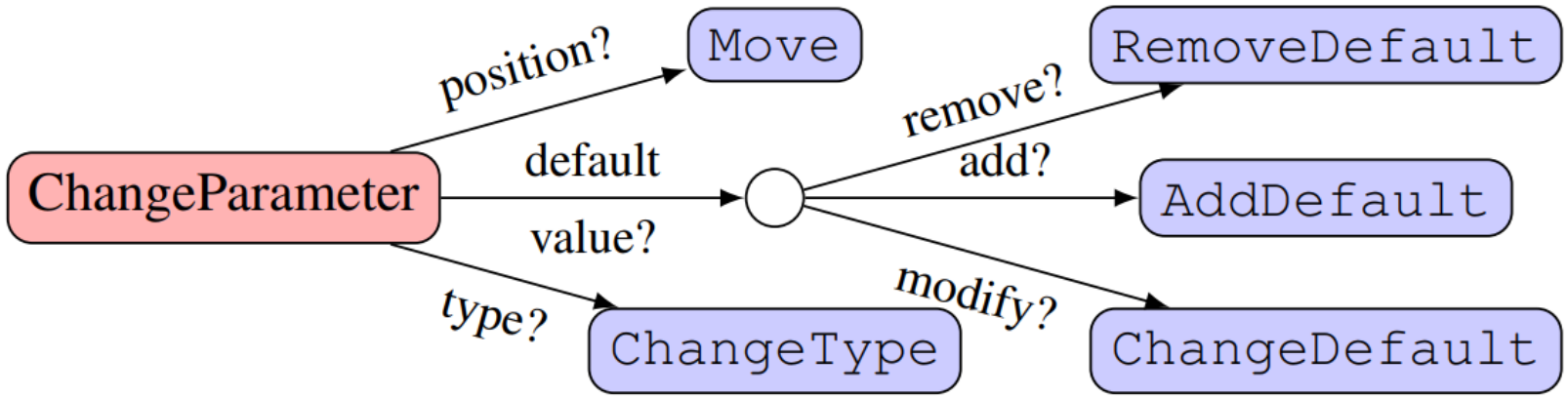
```yaml
modules:
  members:
    func: M.func
    _share: M._share
    A: M.A
    B: M.B
    C: M.C
classes:
  M.A:
    members:
      typeme: M._share
  M.B:
    members:
      typeme: M._share
      g: M.B.g
      x: M.B.x
      h: M.B.h
  M.C:
    bases: [list, M.B]
    abcs: [Sequence, Iterable]
attributes:
  M.B.x:
    scope: instance
  M.B._x:
    scope: instance
```

```yaml
functions:
  M.func:
    parameters:
      - name: a
        kind: Positional
      - name: b
        kind: Positional
      - name: c
        kind: PositionalOrKeyword
        optional: true
        default: <object>
  M._share:
    aliases: [M.A.typeme, M.B.typeme]
    parameters:
      - name: self
        kind: PositionalOrKeyword
  M.B.g:
    parameters:
      - name: self
        kind: PositionalOrKeyword
      - name: c
        kind: PositionalOrKeyword
        optional: true
        type:
          category: union
          components: [B, none]
    return:
      category: union
      components: [B, none]
```

方法原理——变更分类与检测

42 种变更模式

	Module	Class	Function	Attribute	Parameter	Alias
Addition	AddModule	AddClass	AddFunction	AddAttribute	*AddParameter*[*]	AddAlias
Removal	*RemoveModule*	*RemoveClass*	*RemoveFunction*	*RemoveAttribute*	*RemoveParameter*	*RemoveAlias*
Modification	-[†]	*ChangeInheritance*	*ChangeReturnType*	*ChangeAttributeType*	*ChangeParameter*	ChangeAlias



方法原理——变更分类与检测

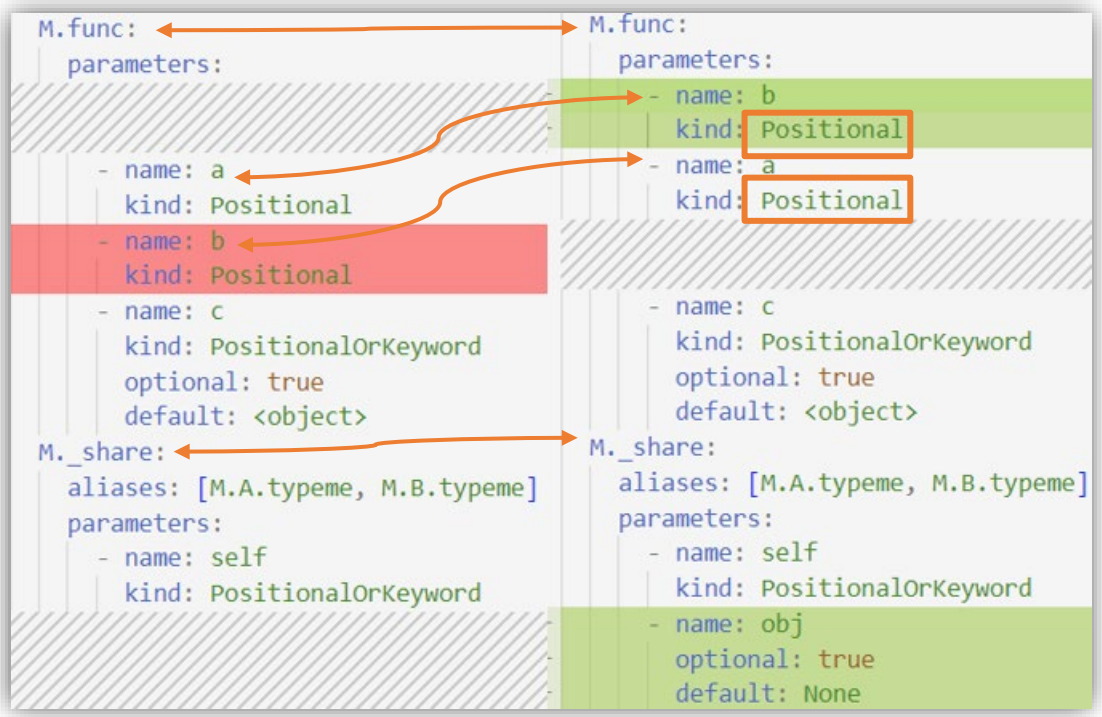

```
def func(a, b, /, c = []): pass
def _share(self): print(type(se
def func(b, a, /, c = []): pas
def _share(self, obj = None): print(type(obj or self))
```

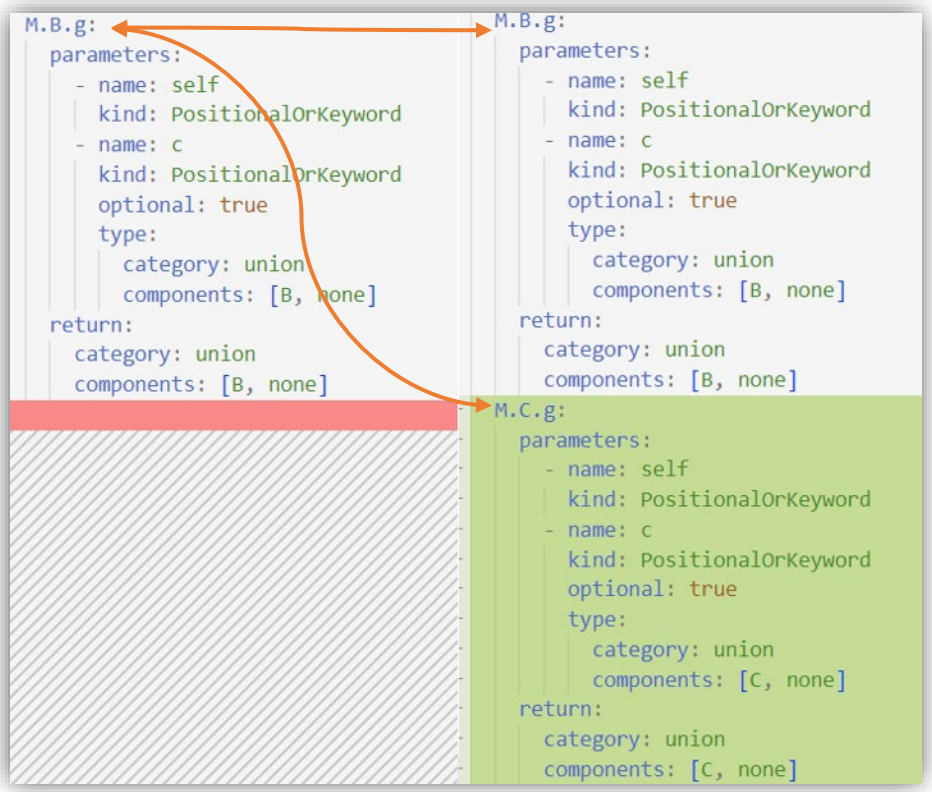
配对

比较

模拟API名称解析和参数传递

```
class C(B):
    def g(self, c: "opt[C]" = None) -> "C | None": return c
```





方法原理——变更分类与检测

配对 → 比较

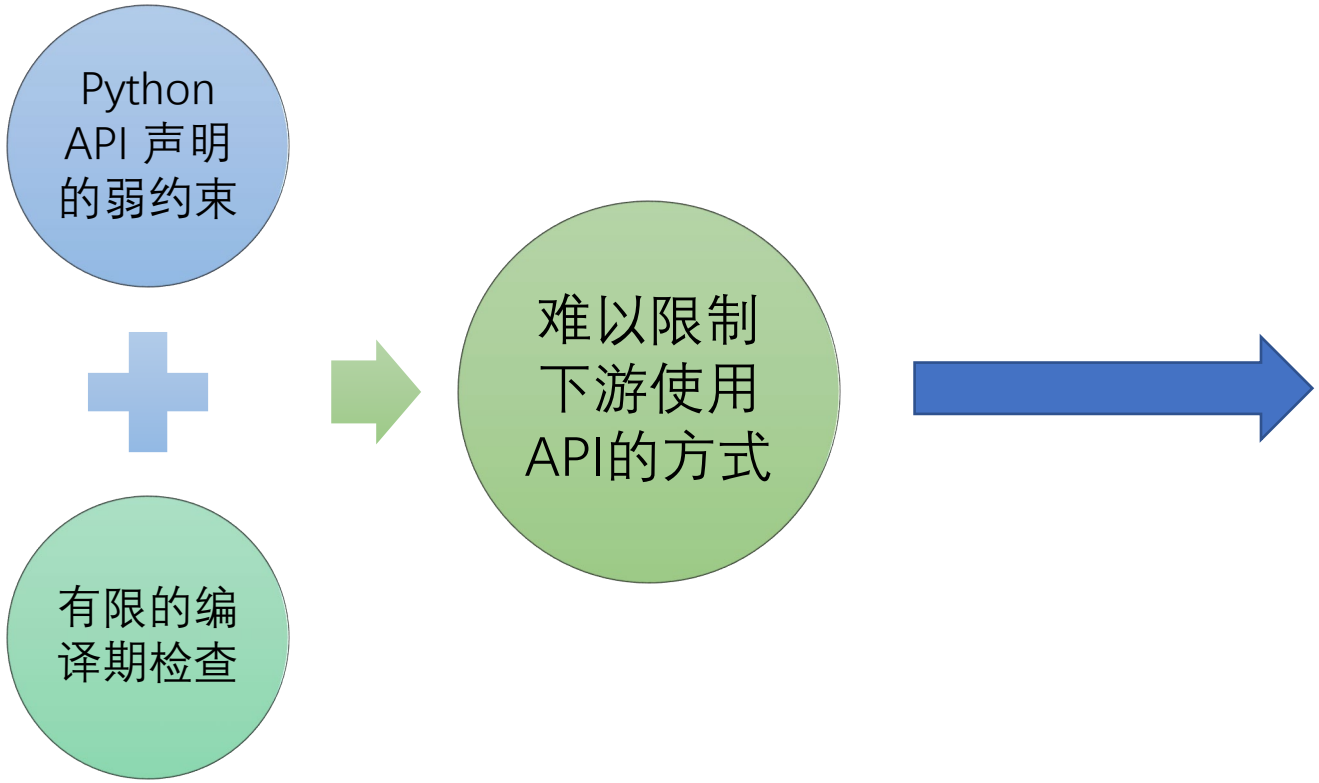
基于变更模式约束条件检测配对的API

42 种变更模式 → **42 个约束条件**

Pattern	Constraint
AddModule	$e = \bot \wedge e' \in M$
RemoveFunction	$e \in F \wedge e' = \bot \wedge \textbf{scope}(e) = \text{Static}$
RemoveBaseClass	$e, e' \in C \wedge \textbf{bases}(e) \not\subseteq \textbf{bases}(e')$
ChangeReturnType	$e, e' \in F \wedge \textbf{return}(e) \neq \textbf{return}(e')$
AddRequiredParameter	$p = \bot \wedge p' \neq \bot \wedge \neg\textbf{optional}(p')$
MoveParameter	$p \neq \bot \wedge p' \neq \bot \wedge \textbf{position}(p) \neq \textbf{position}(p')$
RemoveVarKeywordCandidate	$p \neq \bot \wedge p' = \bot \wedge \textbf{kind}(p) = \text{VarKeywordCandidate}$
RemoveAlias	$e, e' \in M \cup C \wedge (\exists(n,t), t \in E \wedge n \in \textbf{aliases}(t) \wedge (n,t) \in (\textbf{members}(e) - \textbf{members}(e')))$
RemoveExternalAlias	$e, e' \in M \cup C \wedge \exists n, (n, \bot) \in (\textbf{members}(e) - \textbf{members}(e'))$

完整列表位于 Specification of Changes – AexPy (https://aexpy.netlify.app/change-spec).

方法原理——破坏性分级与评估



高

- 直接破坏可用性
- API移除

中

- 有限的破坏性影响
- 类型变更

低

- 伪私有成员
- 外部API的引用

方法原理——破坏性分级与评估

高 对类“C”，移除其基类“list”

高 重排序函数“func”的参数“a”和“b”

$$\textsc{Callable}: \frac{T_{args} \subseteq S_{args} \qquad S_{ret} \subseteq T_{ret}}{T_{args} \to T_{ret} \subseteq S_{args} \to S_{ret}}$$

中 更改函数“C.g”的参数“c”的类型，不再接受类型为“B”的参数

兼容 更改函数“C.g”的返回类型，新返回类型“C”，是旧返回类型“B”的子类

低 对函数“_share”，添加可选参数“obj”

高
- 直接破坏可用性
- API移除

中
- 有限的破坏性影响
- 类型变更

低
- 伪私有成员
- 外部API的引用

方法原理——破坏性分级与评估

高 对类“C”，移除其基类“list”

高 重排序函数“func”的参数“a”和“b”

$$\textsc{Callable}: \frac{T_{args} \subseteq S_{args} \qquad S_{ret} \subseteq T_{ret}}{T_{args} \to T_{ret} \subseteq S_{args} \to S_{ret}}$$

中 更改函数“C.g”的参数“c”的类型，不再接受类型为“B”的参数

兼容 更改函数“C.g”的返回类型，新返回类型“C”，是旧返回类型“B”的子类

低 对函数“_share”，添加可选参数“obj”

高
- 直接破坏可用性
- API移除

中
- 有限的破坏性影响
- 类型变更

低
- 伪私有成员
- 外部API的引用

系统实现

算法实现

分步隔离

增量计算

架构设计

职责解耦

接口统一

功能使用

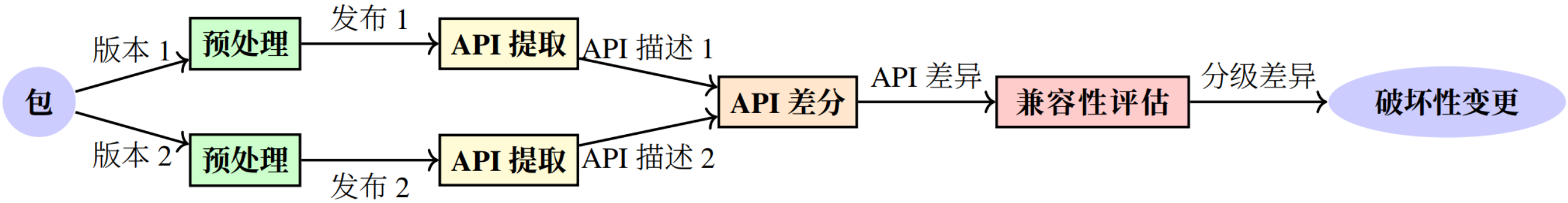
易移植

易使用

易集成

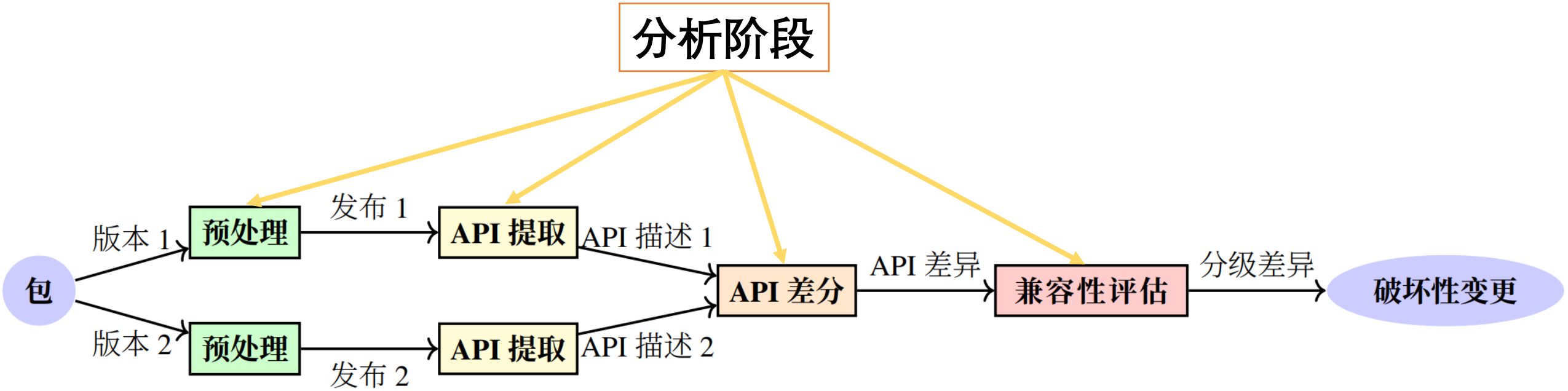
系统实现——算法实现

分步隔离，增量计算



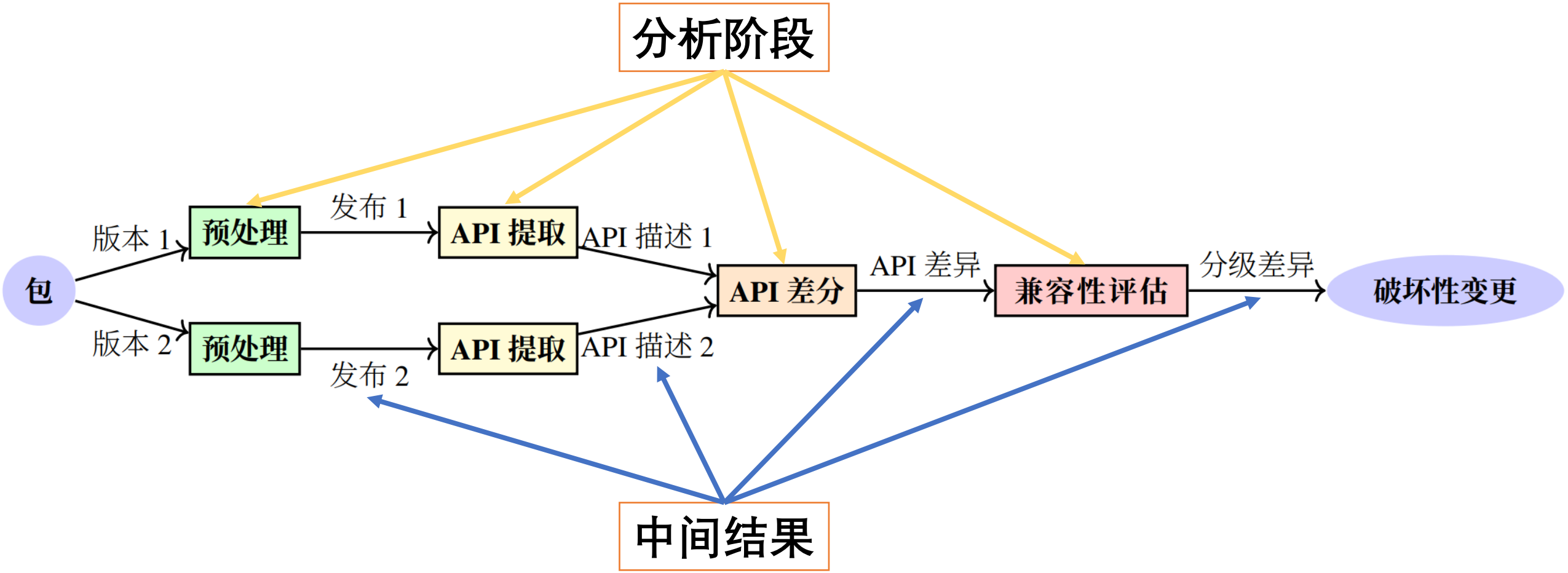
系统实现——算法实现

分步隔离，增量计算



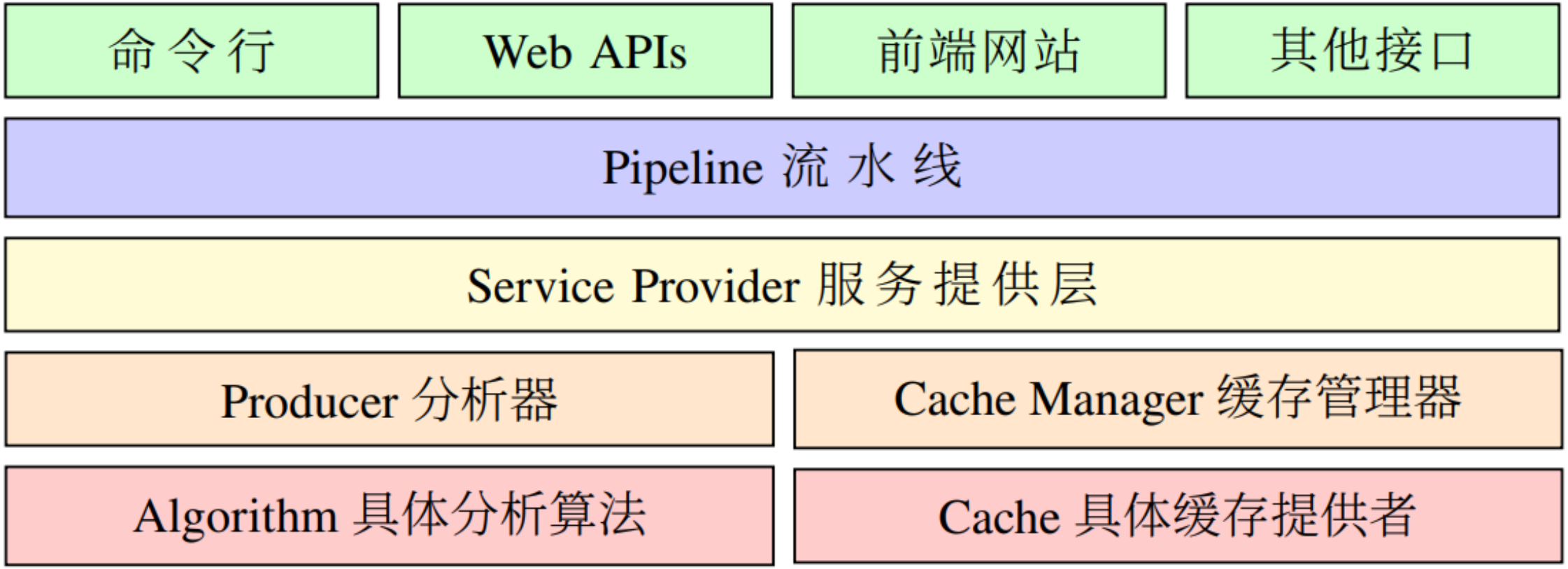
系统实现——算法实现

分步隔离，增量计算



系统实现——架构设计

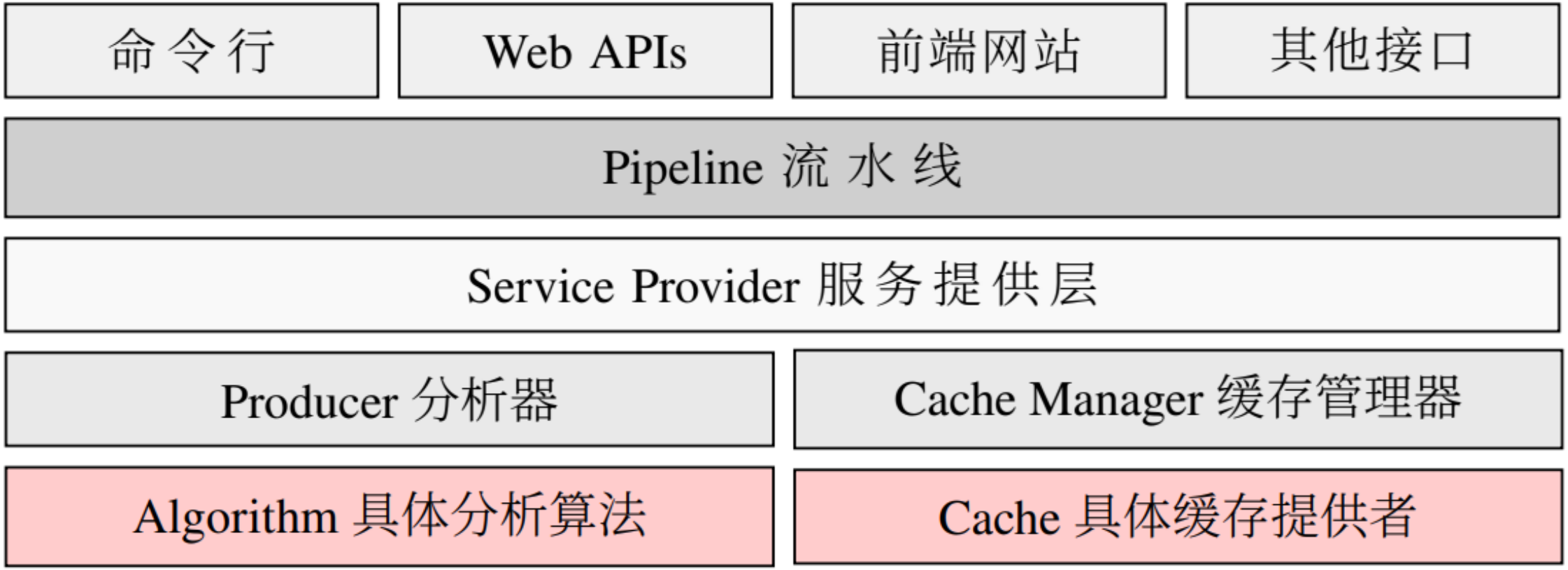
职责解耦，接口统一



分层结构，下层向上层提供服务，直至面向用户的顶层

系统实现——架构设计

职责解耦，接口统一

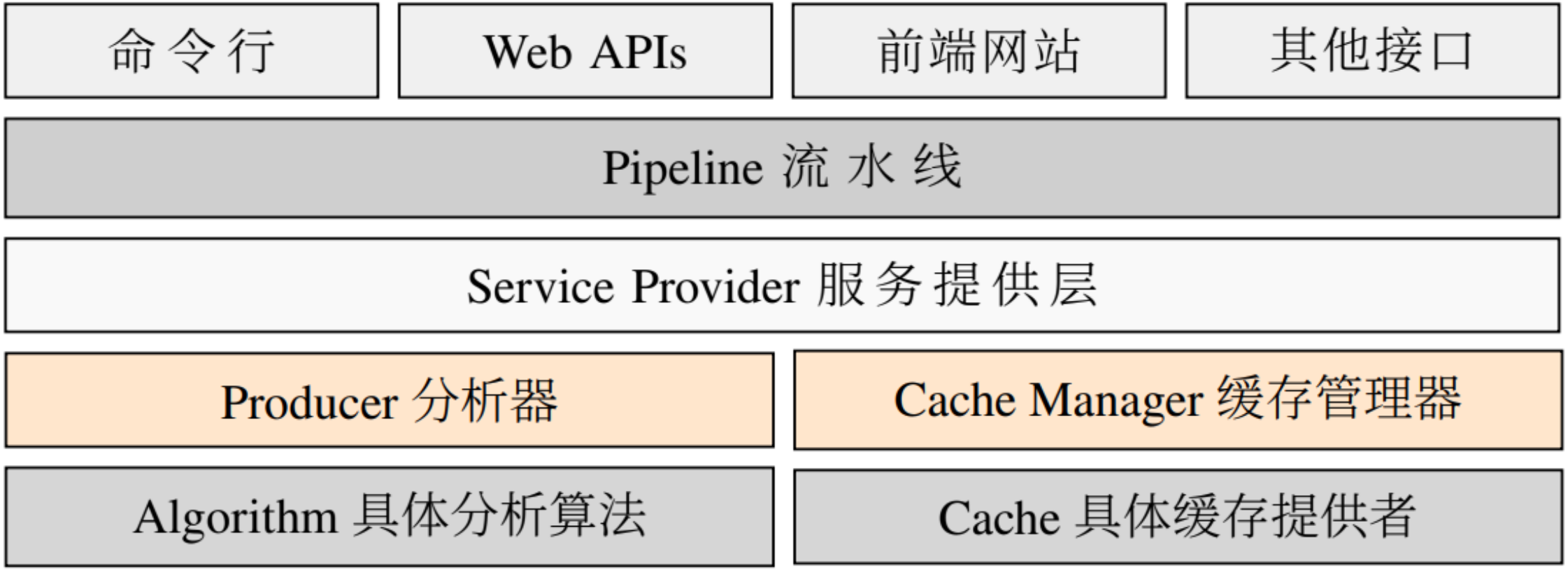


实现为可重入幂等的**纯函数**

存储结果到文件系统或数据库

系统实现——架构设计

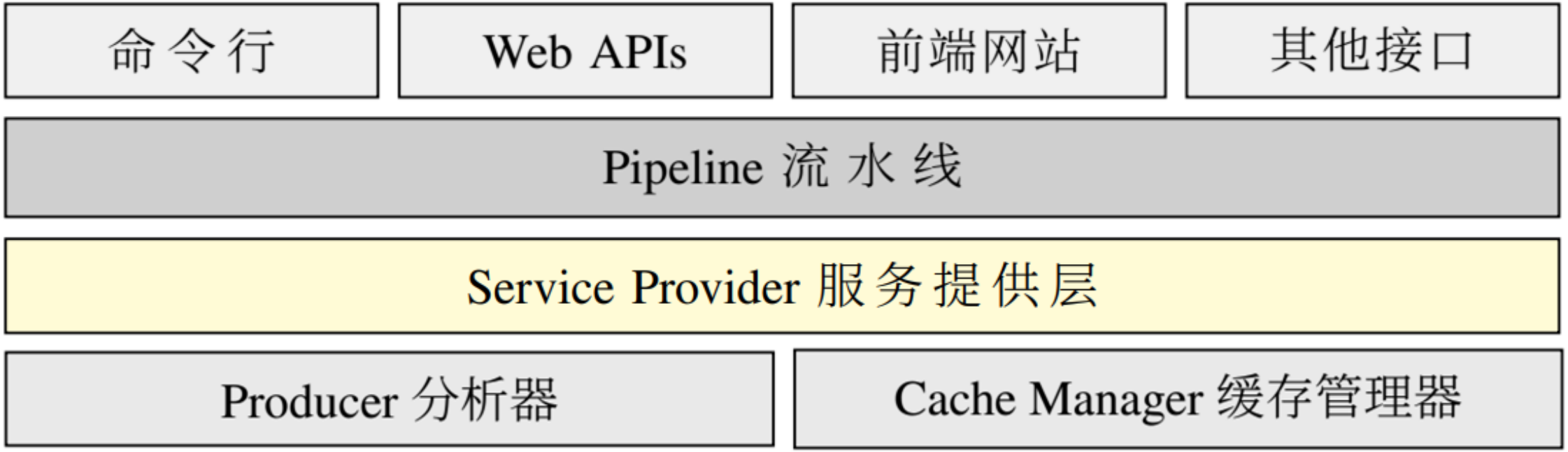
职责解耦，接口统一



将不同算法和缓存实现抽象为**统一接口**，封装实现细节

系统实现——架构设计

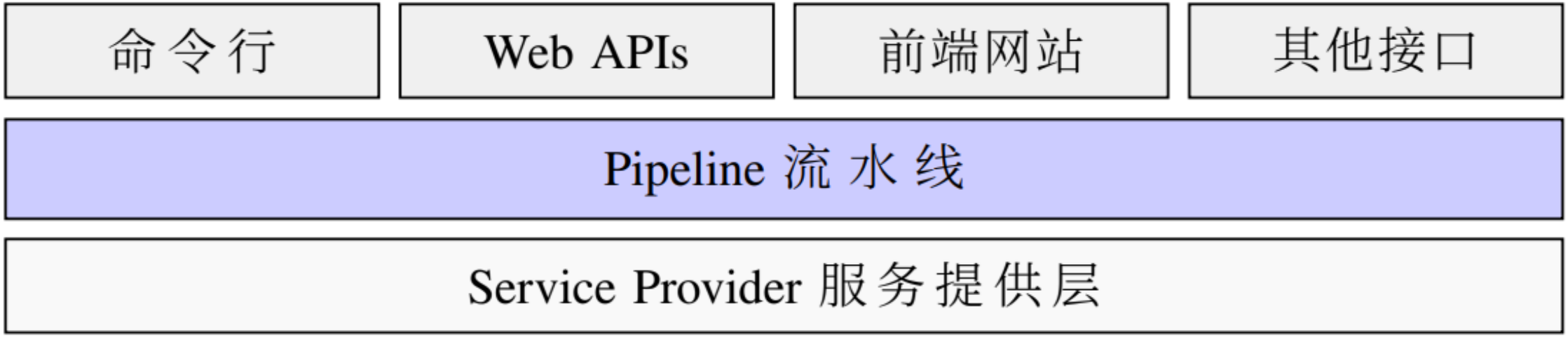
职责解耦，接口统一



根据环境配置构建分析器和缓存管理器
为每一分析阶段实现**读取-执行-存储**流程

系统实现——架构设计

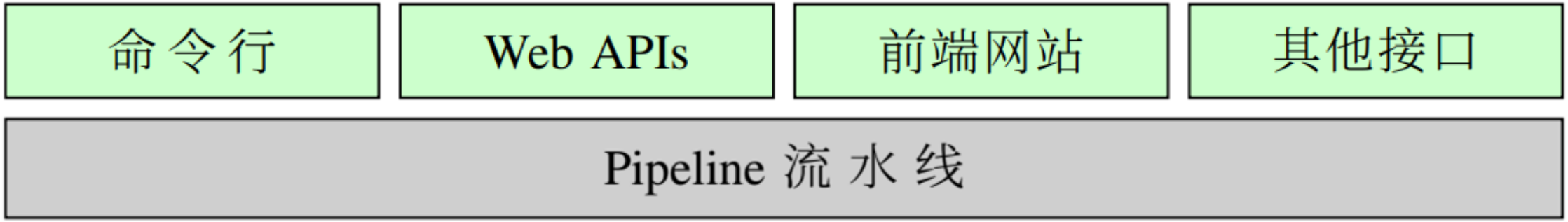
职责解耦，接口统一



将分析阶段间输入输出组合为**流水线**
提供高层次接口供上层使用

系统实现——架构设计

职责解耦，接口统一



将用户输入**分发**到对应流水线进行处理，并提供反馈

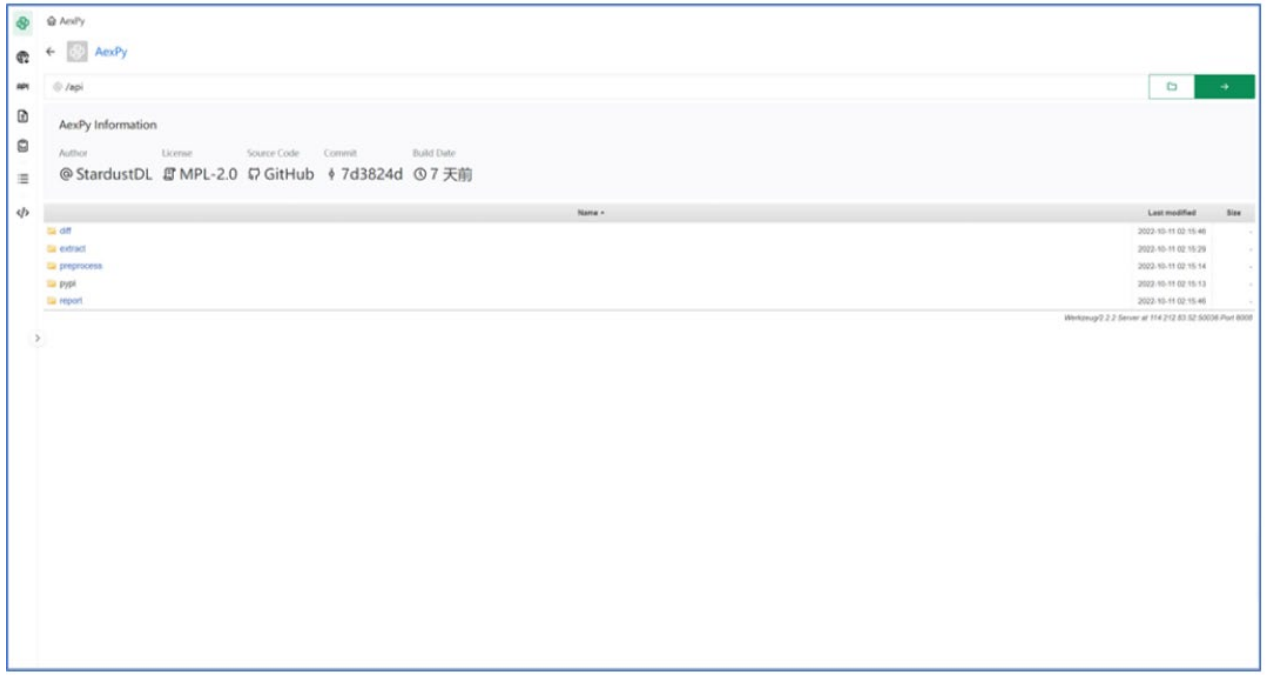
系统实现——功能使用

易部署，易使用，易集成

Docker 容器镜像

前端用户界面
交互式命令行

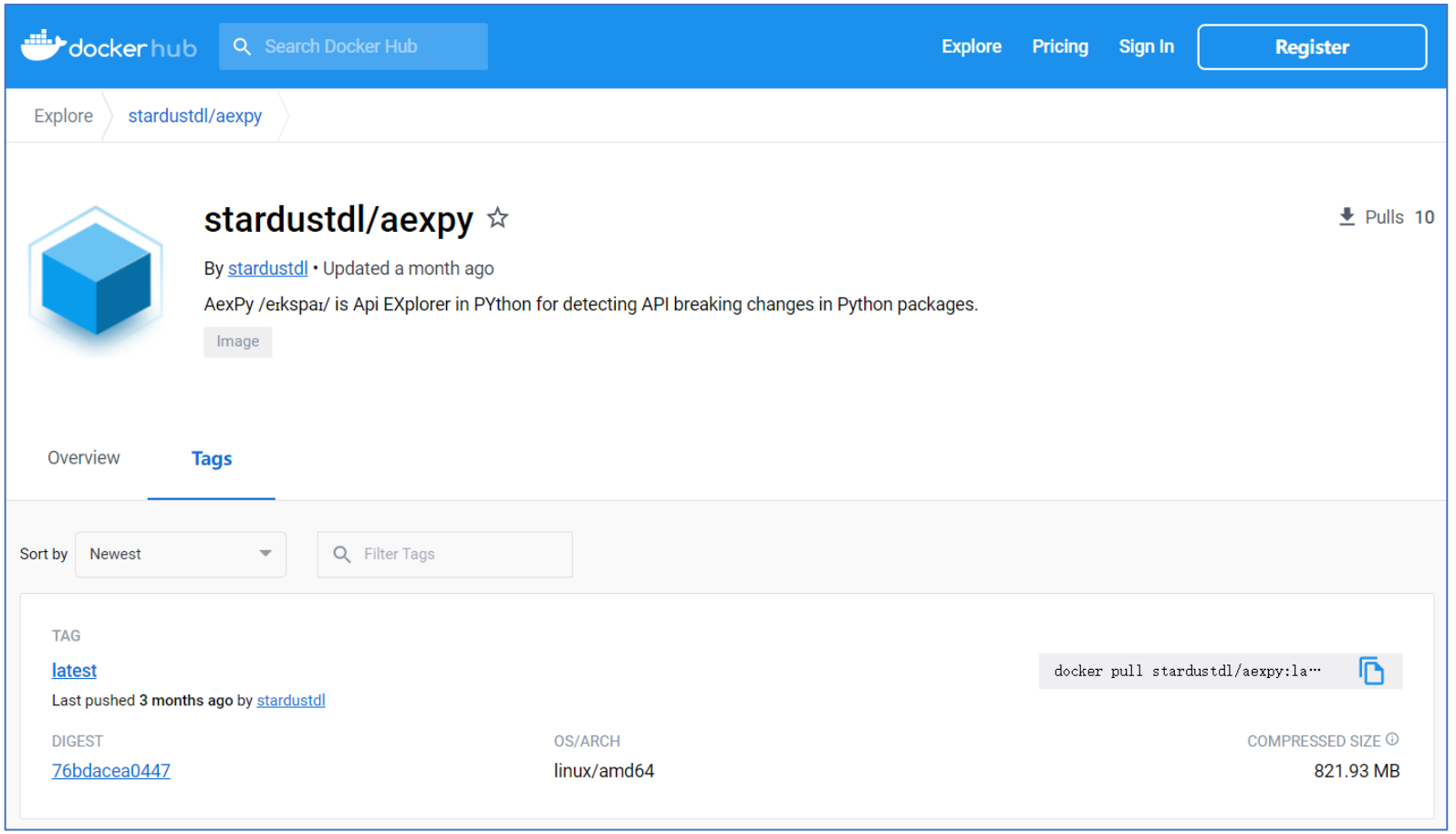
命令行接口
JSON 格式输出
可扩展系统结构



系统实现——功能使用

易部署，易使用，易集成

Docker 容器镜像



系统实现——功能使用

易部署，易使用，易集成

Docker 容器镜像

前端用户界面
交互式命令行

R	Kind	Message	Old
●	RemoveOptionalParameter	Remove PositionalOrKeyword parameter (click.core.Parameter.__init__): autocompletion.	click.core.Parameter.__init__
●	AddOptionalParameter	Add PositionalOrKeyword parameter (click.types.Path.__init__): executable.	click.types.Path.__init__
●	ChangeParameterDefault	Change parameter default (click.core.Option.__init__): show_default: bool('False') -> None.	click.core.Option.__init__
●	AddRequiredParameter	Add PositionalOrKeyword parameter (click.shell_completion._is_incomplete_option): ctx.	click.shell_completion._is_incomplete_option
●	AddRequiredParameter	Add PositionalOrKeyword parameter (click.shell_completion._start_of_option): ctx.	click.shell_completion._start_of_option
●	MoveParameter	Move parameter (click.shell_completion._is_incomplete_option): args: 1 -> 2.	click.shell_completion._is_incomplete_option
●	MoveParameter	Move parameter (click.shell_completion._is_incomplete_option): param: 2 -> 3.	click.shell_completion._is_incomplete_option

```
(base) test@5820:~/liang/temp$ docker run --rm -it -v $(pwd):/data stardustdl/
✅ ApiDescription click@8.0.4 (by types):
  ⏰ 2022-10-26 07:48:29.969871 , 23.379338s
  🔷 826 entries
    Modules: 16
    Classes: 61
    Functions: 417
    Attributes: 332
>>> list(locals().keys())
['release', 'mode', 'json', 'log', 'releaseVal', 'pipeline', 'result', '__buil
>>> release
'click@8.0.4'
>>> mode
'a'
>>> j
```

系统实现——功能使用

易部署，易使用，易集成

Docker 容器镜像

前端用户界面
交互式命令行

命令行接口
JSON 格式输出
可扩展系统结构

```
(base) test@5820:~/liang/temp$ docker run --rm stardustdl/aexpy:main --help
Usage: python -m aexpy [OPTIONS] COMMAND [ARGS]...

  AexPy /eɪkspaɪ/ is Api EXplorer in PYthon for detecting API breaking changes
  in Python packages. (ISSRE'22)

  Home page: https://aexpy.netlify.app/

  Repository: https://github.com/StardustDL/aexpy

Options:
  --version               Show the version and exit.
  -c, --cache DIRECTORY   Path to cache directory.
  -v, --verbose           Increase verbosity.  [0<=x<=5]
  -i, --interact          Interact mode.
  -p, --pipeline TEXT     Pipeline to use.
  --config FILE           Config file.
  --help                  Show this message and exit.

Commands:
  batch       Process project.
  diff        Diff two releases.
  extract     Extract the API in a release.
  initialize  Rebuild the environment.
  preprocess  Preprocess a release.
  report      Report breaking changes between two releases.
  serve       Serve web server.
```

```
"click.utils.KeepOpenFile._file": {
    "name": "_file",
    "id": "click.utils.KeepOpenFile._file",
    "alias": [],
    "docs": "",
    "comments": "",
    "src": "",
    "location": {
        "file": "click/utils.py",
        "line": 184,
        "module": "click.utils"
    },
    "private": true,
    "parent": "click.utils.KeepOpenFile",
    "schema": "attr",
    "data": {},
    "scope": 2,
    "type": null,
    "rawType": "",
    "annotation": "",
    "property": false
}
```

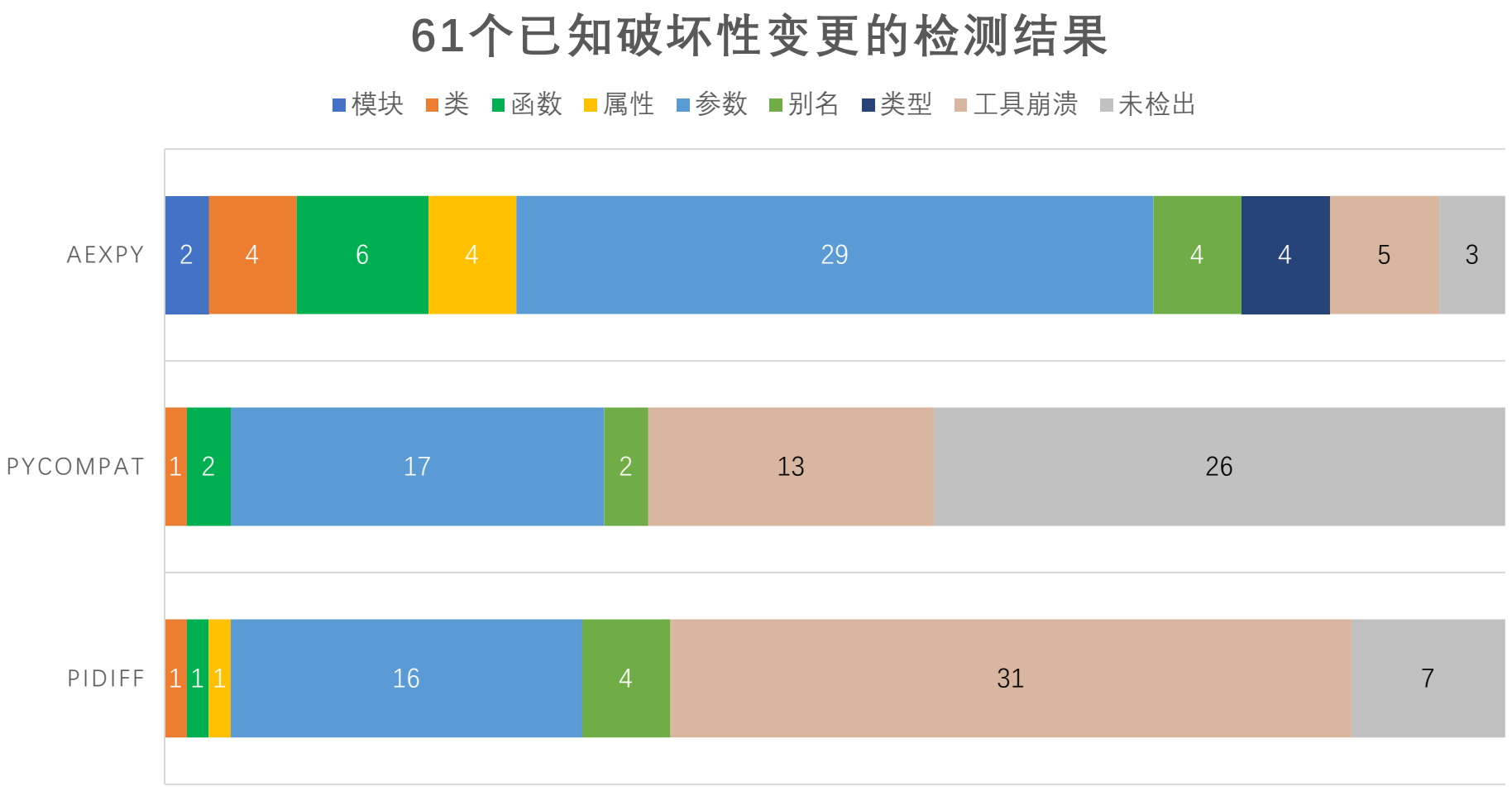
实验评估——环境配置

- 评估问题
  - 召回率：AexPy 能否较现有工具检测出更多的**已知**破坏性变更?
  - 实用性：AexPy 能否发现潜在的**未知**破坏性变更?
  - 效率：AexPy 较现有工具，**时间**效率如何?
- 数据集
  - 61 个破坏性变更
  - 45 个软件包

  来自GitHub上近一年已解决的Python包API破坏性变更和10个流行Python包的变更日志
- 现有工具
  - PyCompat：一个包含变更检测器的API误用检测工具
  - Pidiff：一个为Python包设计的语义版本检查器

实验评估——召回率

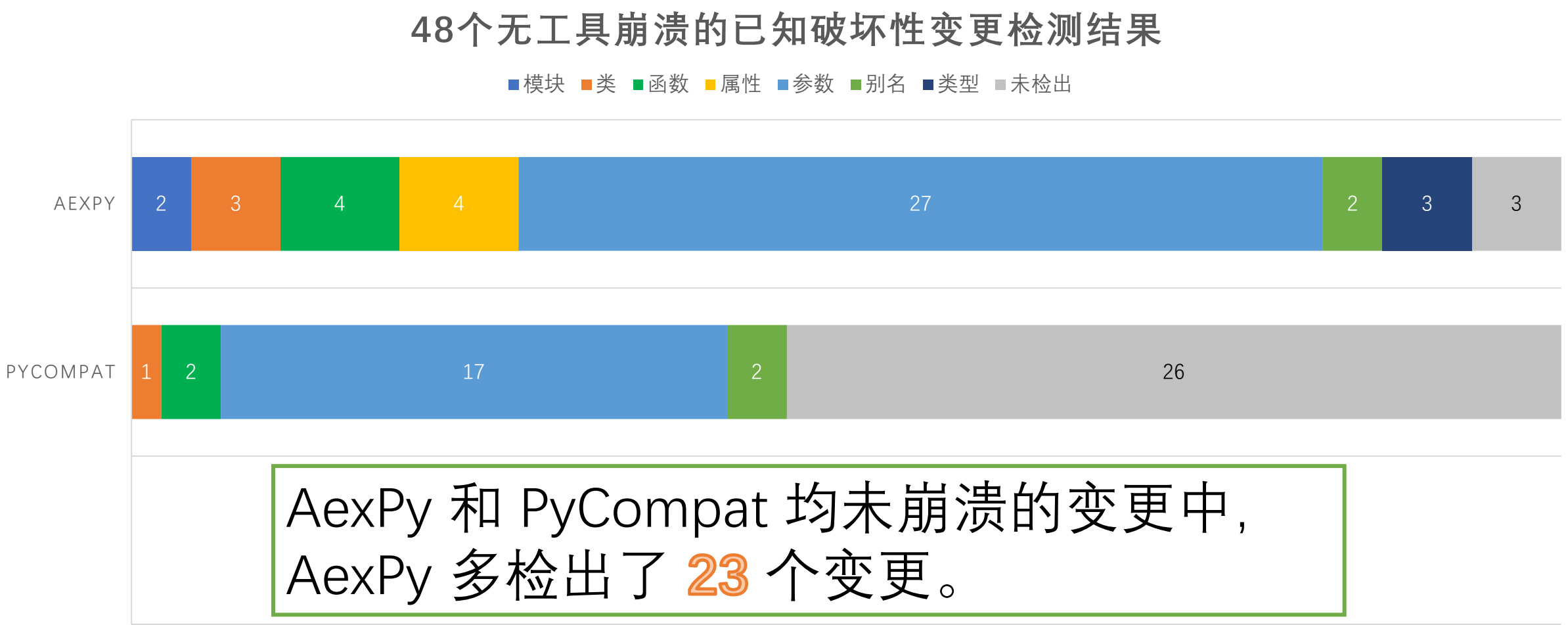
AexPy 能否较现有工具检测出更多的已知破坏性变更?



AexPy 检出了 **53** 个变更，提高了~ **50%** 的召回率。

实验评估——召回率

AexPy 能否较现有工具检测出更多的已知破坏性变更？



AexPy 和 PyCompat 均未崩溃的变更中，
AexPy 多检出了 23 个变更。

实验评估——召回率

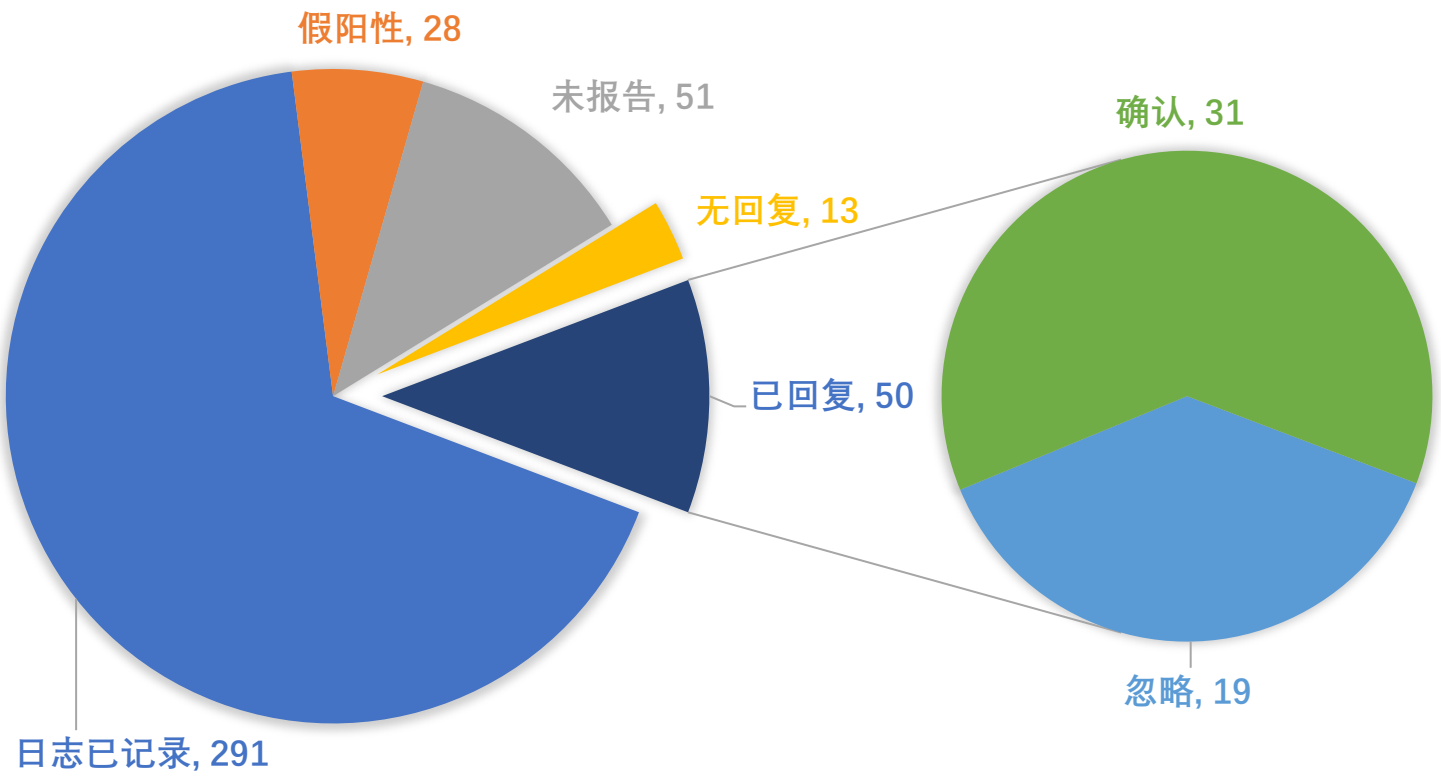
AexPy 能否较现有工具检测出更多的已知破坏性变更?

AexPy 和 Pidiff 均未崩溃的变更中，
AexPy 多检出了 6 个变更。

实验评估——实用性

AexPy 能否发现潜在的未知破坏性变更？

45个包的最新版本上检出的高、中级别破坏性变更



AexPy 检出了 433 高、中级别破坏性变更，405 个经人工检查真实存在。

在 63 个报告给开发者的潜在破坏性变更中 31 个得到了确认。

实验评估——实用性

AexPy 能否发现潜在的未知破坏性变更？

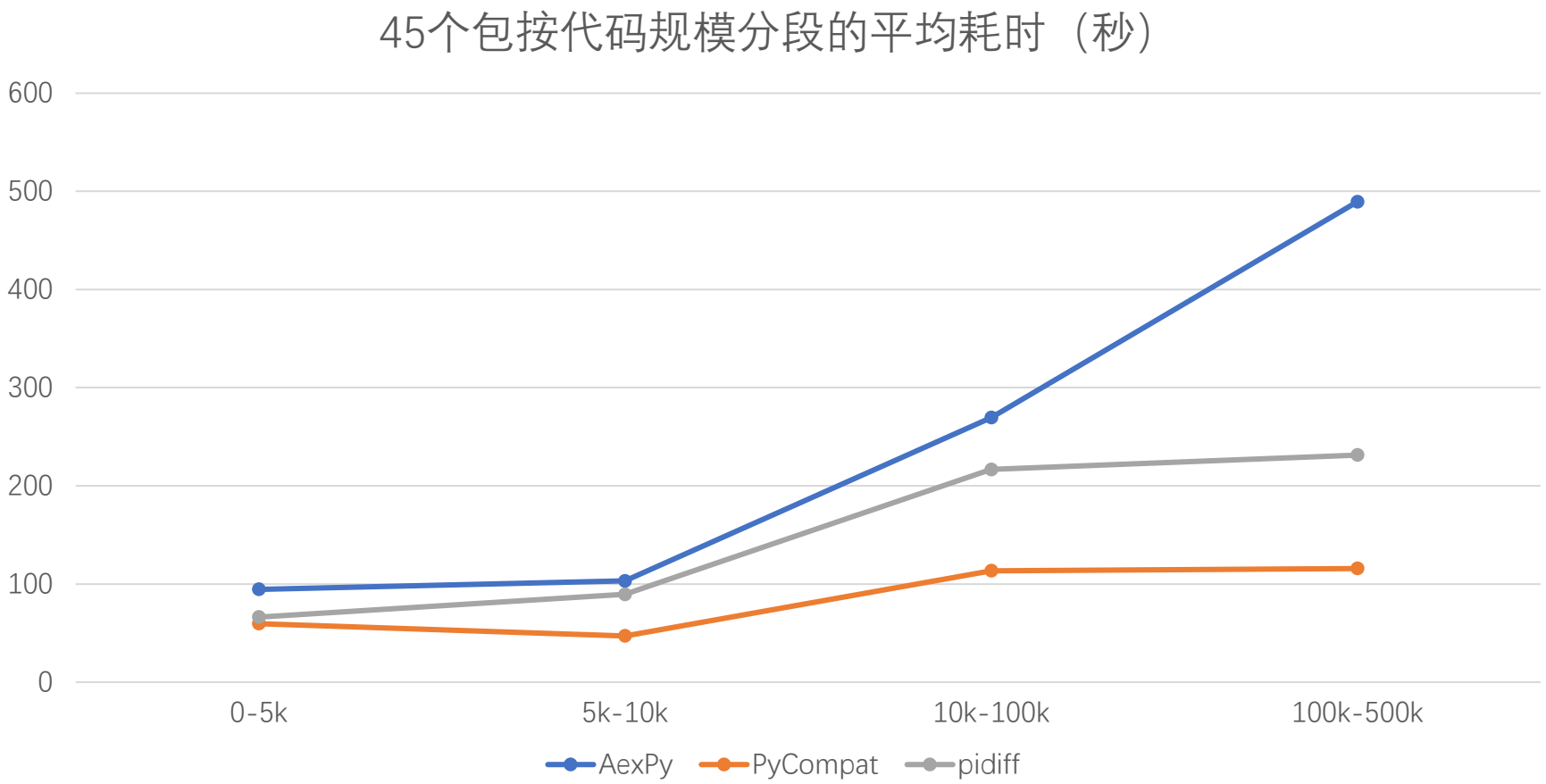
45个包的最新版本上检出的高、中级别破坏性变更

AexPy 检出了 433 高、中级别破坏性变更，405 个经人工检查真实存在。

在 63 个报告给开发者的潜在破坏性变更中 31 个得到了确认。

实验评估——效率

AexPy 较现有工具，时间效率如何？



实验环境

具有 12 核 3.8GHz 主频 CPU 的 Ubuntu 18.04 主机上的容器环境，每个版本对限制 50GB 内存和一小时分析时间

三个工具分析耗时**处在同一数量级**。

Python包应用程序编程接口破坏性变更检测系统

方法原理

- 基于 ISSRE'22 会议论文提出的 Python 包 API 破坏性变更检测方法
- 针对问题挑战和现有工作的不足，建模提取API、分类检测变更、分级评估破坏性

系统实现

- 强调健壮性和扩展性，关注分析效率和结果复用，分层架构设计，分步算法实现
- 以容器镜像形式提供命令行、Web API 和前端，便于开发者使用和自动化集成

实验评估

- 与现有工具相比，具有更高的召回率和健壮性，并有相近的时间表现
- 检出了潜在破坏性变更并得到了开发者确认

Python包应用程序编程接口破坏性变更检测系统

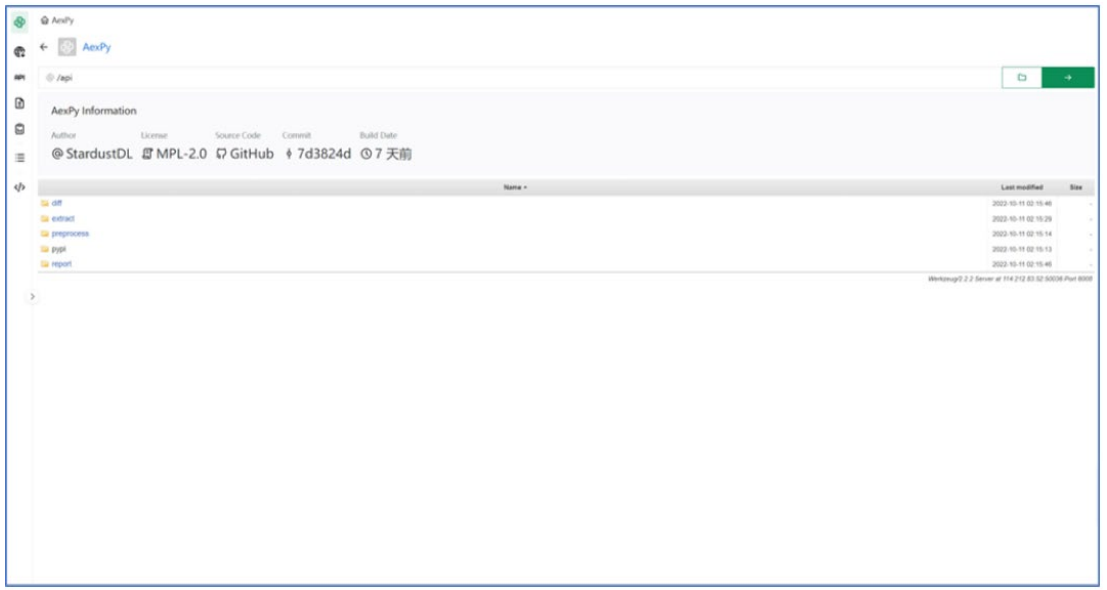
杜星亮
xingliangdu@smail.nju.edu.cn

马骏
majun@nju.edu.cn

https://github.com/**StardustDL/aexpy**

https://pypi.org/project/**aexpy**/

https://hub.docker.com/r/**stardustdl/aexpy**



感谢观看

Thank You