软件工程学院
SOFTWARE ENGINEERING INSTITUTE
华东师范大学软件学院可信智能团队
Trustworthy Intelligence Group

开源软件兼容性可信量化分析

2024年12月17日

大　纲

前置知识

开源软件版本间兼容性

开源软件软件间兼容性

开源软件

开源软件（Open-Source Software）[1]是通过特定类型的许可证发布的软件，这种许可证能让最终用户合法地使用其源代码。此类许可证有许多种（GPL、MIT、BSD...），但通常开源软件必须符合以下条件：

- **以源代码形式提供，无需额外费用**：这意味着用户可以查看组成该软件的代码并对其进行所需的任何更改。
- **源代码可重新用于其他新软件**：这意味着任何人都可以获取源代码并利用它来分发自己的程序。

GitHub

Java开源软件：Spring、log4j、Jackson

gitee

C/C++开源软件：zlib、boost、openssl

SOURCEFORGE

Python开源软件：numpy、tensorflow、django

- 通过复用开源软件的代码，可以提高开发效率，降低开发成本。

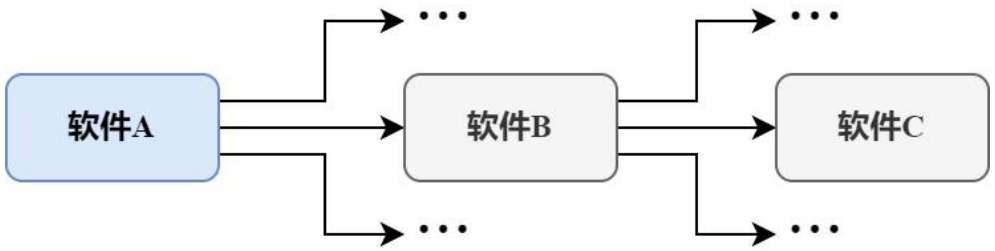
[1] https://www.redhat.com/zh/topics/open-source/what-is-open-source-software

依赖关系

- 在软件开发时，为复用开源软件的源代码，需要**将开源软件的源代码下载至本地**，然后**在项目中导入所需要的功能模块。**

```
Java: import org.springframework.boot.SpringApplication;
C/C++: #include <boost/lambda/lambda.hpp>
Python: import numpy as np
```

- 在开发软件A时，下载并复用软件B的源代码，则称：**软件A依赖于软件B，或软件B是软件A的一个依赖。**



我们将软件A称为**主软件**
软件B是软件A的一个**直接依赖**
软件C是软件B的一个**直接依赖**
软件C是软件A的一个**间接依赖**

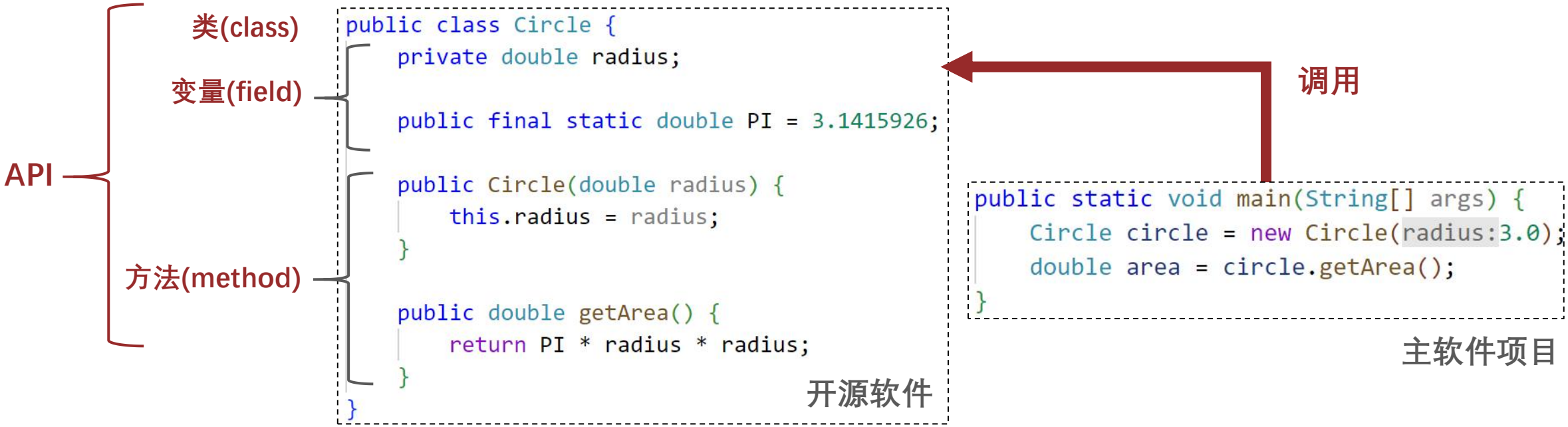
依赖（包）管理器/解析器

Maven、Gradle（Java）
Conan、vcpkg（C++）
pip、Conda（Python）

通过**显式声明所需要的直接依赖**，依赖管理器会自动解析并下载所有需要的直接依赖和间接依赖

API

API（Application Programming Interface, 应用程序编程接口）[1]是一组**规则或协议**，可支持软件应用程序相互通信，以交换数据、特性和功能。API 允许开发人员集成来自其他应用程序的数据、服务和功能，而不是从头开始开发它们，从而简化和加速软件开发。



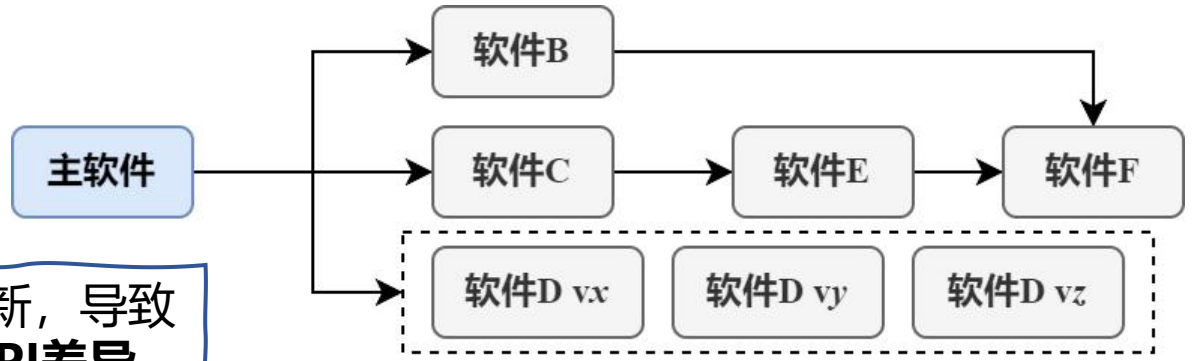
1. 主软件项目声明并下载所需要的直接依赖软件（开源软件）
2. 导入所需要的功能模块
3. **按需调用直接依赖软件（开源软件）所提供的API**

最终通过**调用开源软件提供的API**来实现代码复用

[1] https://www.ibm.com/cn-zh/topics/api

开源软件兼容性

- 开源软件**数量众多、依赖关系复杂**
- 开源软件**持续更新、版本众多**



修复Bug/漏洞、增加新功能、重构等目的而更新，导致**API发生变化**，同一开源软件的**不同版本存在API差异**

从而可能导致兼容性问题

开源软件**版本间**兼容性：同一开源软件不同版本间是否向下/向上兼容

开源软件**软件间**兼容性：主软件/开源软件可以与其他开源软件共存，并能够正常地调用其依赖的开源软件所提供的功能。

① 版本间/软件间兼容性问题检测

② 版本间/软件间兼容性可信度量

兼容性度量元 → 兼容性可信值

大　纲

前置知识

开源软件版本间兼容性

开源软件软件间兼容性

版本间兼容性-兼容性问题检测

- 开源软件会出于修复Bug、添加新功能等目的持续地更新版本，但同时有可能会引入**API破坏性变动（Breaking Changes）**，例如旧版本的API在新版本被删除或随意地改动，这就会导致**新版本无法向下兼容旧版本。**

“而有些版本间的API变动是良性的，不会对兼容性造成影响”

开源软件**opencensus**

```java
public final class ContextUtils {
  // No instance of this class.
  private ContextUtils() {}
```

0.28.0版本

版本更新

```java
final class ContextUtils {
  // No instance of this class.
  private ContextUtils() {}
```

0.28.1版本

我们将同一开源软件的不同版本之间的兼容性问题称为**开源软件版本间兼容性问题**，具体而言，我们需要检测同一开源软件不同版本之间的API变动是否会影响兼容性，以及影响的严重程度如何。

版本间兼容性-兼容性问题检测

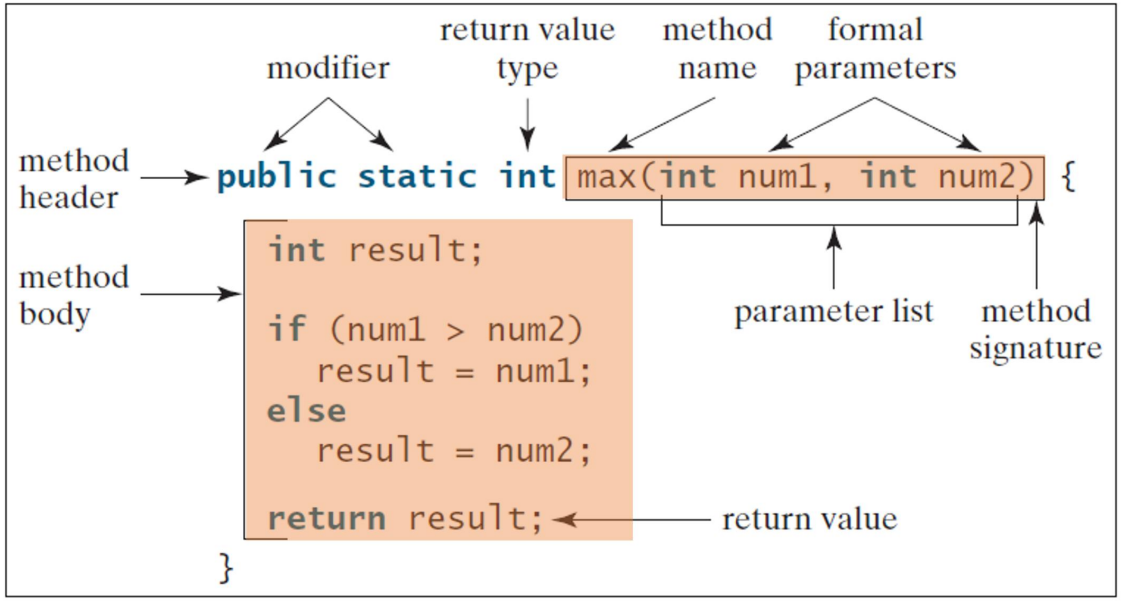
语义化版本控制规范（Semantic Versioning Specification，SemVer）[1]：**主版本号.次版本号.修订号（x.y.z）**

- **主升级（Major Upgrade）**：存在不兼容的API变动，主版本号x递增，次版本号y和修订号z置零；v3.4.5 → v4.0.0
- **次升级（Minor Upgrade）**：保证向下兼容的功能性新增，次版本号y递增，修订号z置零； v3.4.5 → v3.5.0
- **修订升级（Patch Upgrade）**：保证向下兼容的问题修正，修订号z递增。 v3.4.5 → v3.4.6

普遍会存在一些违反该规范的情况，存在无法向下兼容的次升级和修订升级。

API语法（签名）变动：
API签名是软件API的唯一标识符，其他软件可以通过签名来指定和调用相应的API

API语法（签名）变动可能会导致主软件编译/链接错误



API语义（行为）变动：
API行为由方法调用链上的所有方法体决定

API语义（行为）可能变动会导致相同的API输入产生不同的行为和输出

[1] https://semver.org/lang/zh-CN/

版本间兼容性-兼容性问题检测

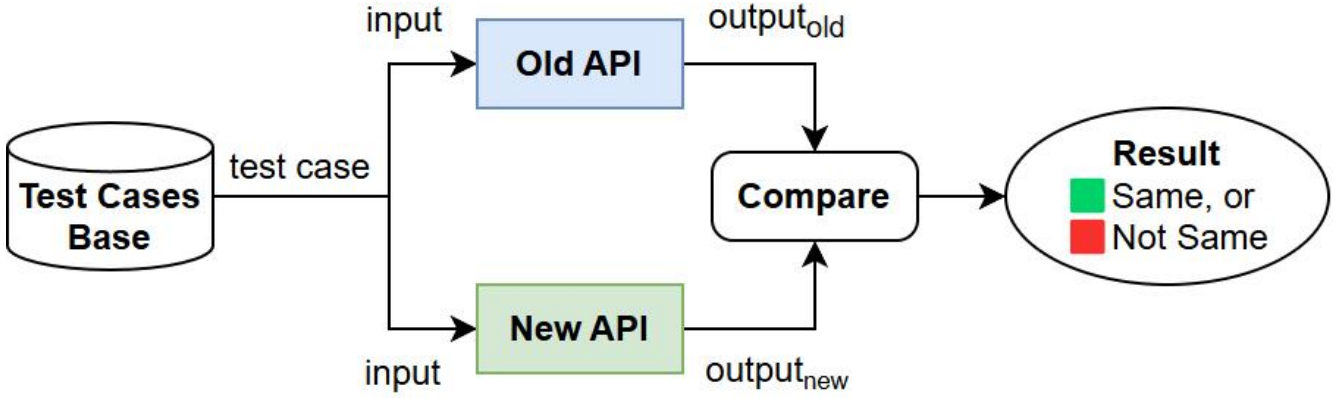
- **API语法（签名）变动 → API语法兼容性问题**

  通常利用**静态分析（Static Analysis）**技术进行检测，比对开源软件新旧版本的API签名，以判断是否发生了变动，并判断该变动是否会对带来兼容性影响（如导致主软件无法成功编译等）

  “静态分析：在不运行程序的情况下，对程序（的代码）进行分析”

- **API语义（行为）变动 → API语义兼容性问题**

  通常利用**动态测试**进行检测，有常被称为**衰退测试或回归测试（Regression Test）**



对新版本的API执行旧版本API的单元测试，**若对于相同的输入能产生相同的行为和输出，则认为新旧版本API的语义（行为）一致，新版本能够兼容旧版本**；否则，则认为发生了衰退（Regression），无法兼容

版本间兼容性-兼容性问题检测

- 我们主要关注**API语法兼容性问题**，借助开源的**静态分析工具**来分析开源软件版本间的API变动

① Java API Compliance Checker（JAPICC）[1]：以命令行的方式使用，以编译构建好的Jar包作为输入，当输入一个Java库的两个版本的Jar包后，JAPICC会分析其中每一个API在两个版本中的API变动，并最终输出兼容性报告。

Problems with Methods, High Severity 26 — **mina-core v2.1.6 → mina-core v2.2.1**

mina-core-2.1.6.jar, CloseFuture.class
package org.apache.mina.core.future
[−] **CloseFuture.addListener** (IoFutureListener<?> *p1*) [abstract] : CloseFuture 1
org/apache/mina/core/future/CloseFuture.addListener:(Lorg/apache/mina/core/future/IoFutureListener;)Lorg/apache/mina/core/future/CloseFuture;

	Change	Effect
1	Return value type has been changed from **CloseFuture** to **IoFuture**.	This method has been removed because the return type is part of the method signature. A client program may be interrupted by **NoSuchMethodError** exception.

JAPICC根据其兼容性规则，检查多种类型的API变动，并给予兼容性严重程度等级，包括：高（High）、中（Medium）、低（Low）和安全（Safe）。

[1] https://lvc.github.io/japi-compliance-checker/

版本间兼容性-兼容性问题检测

- 我们主要关注**API语法兼容性问题**，借助开源的**静态分析工具**来分析开源软件版本间的API变动

② **Revapi** [1]：Revapi提供了多种使用方式，包括：命令行、Maven插件、Gradle插件、Ant任务和直接库调用。在指定Java兼容检查所需要的拓展插件后，提供同一个Java库新旧两个版本的Jar包，即可进行兼容性检查，并输出检查报告。

```
"code" : "java.field.visibilityIncreased",
"old" : "field org.apache.mina.filter.ssl.SslFilter.sslContext",
"new" : "field org.apache.mina.filter.ssl.SslFilter.sslContext",
"name" : "visibility increased",
"description" : "Visibility increased from 'package' to 'protected'.",
"criticality" : "allowed",
"justification" : null,
"classification" : [ {
  "compatibility" : "BINARY",
  "severity" : "EQUIVALENT"
}, {
  "compatibility" : "SOURCE",
  "severity" : "EQUIVALENT"
} ],
```

mina-core v2.1.6 → mina-core v2.2.1

Revapi根据其兼容性规则对API变动进行识别和分类，并给出严重程度分级，包括：等价变动（equivalent）、非破坏性变动（non breaking）、破坏性变动（breaking）、潜在破坏性变动（potentially breaking）。

[1] https://revapi.org

版本间兼容性-兼容性问题检测

- 我们主要关注**API语法兼容性问题**，借助开源的**静态分析工具**来分析开源软件版本间的API变动

③ ABI Compliance Checker（ABICC） [1]：将新旧版本的动态链接库.so/.dll文件（带有DWARF调试信息）中的API信息导出，然后对导出的API信息进行差异分析，得到兼容性分析报告

Problems with Symbols, Medium Severity 1 — **jsoncpp v1.8.0 → jsoncpp v1.9.4**

value.h, libjsoncpp.so.1.8.0
namespace Json
[−] ValueIteratorBase::deref ( ) const 1
_ZNK4Json17ValueIteratorBase5derefEv

	Change	Effect
1	Type of return value became **const** (has been changed from **Value&** to **Value const&**).	The return value will be treated as non-const by old client applications. This may result in crash or incorrect behavior of applications.

ABICC根据其内置兼容性规则对两个版本间API的变动进行分类和分析，并给出该兼容性问题的严重程度，包括：高（High）、中（Medium）、低（Low）和安全（Safe）。

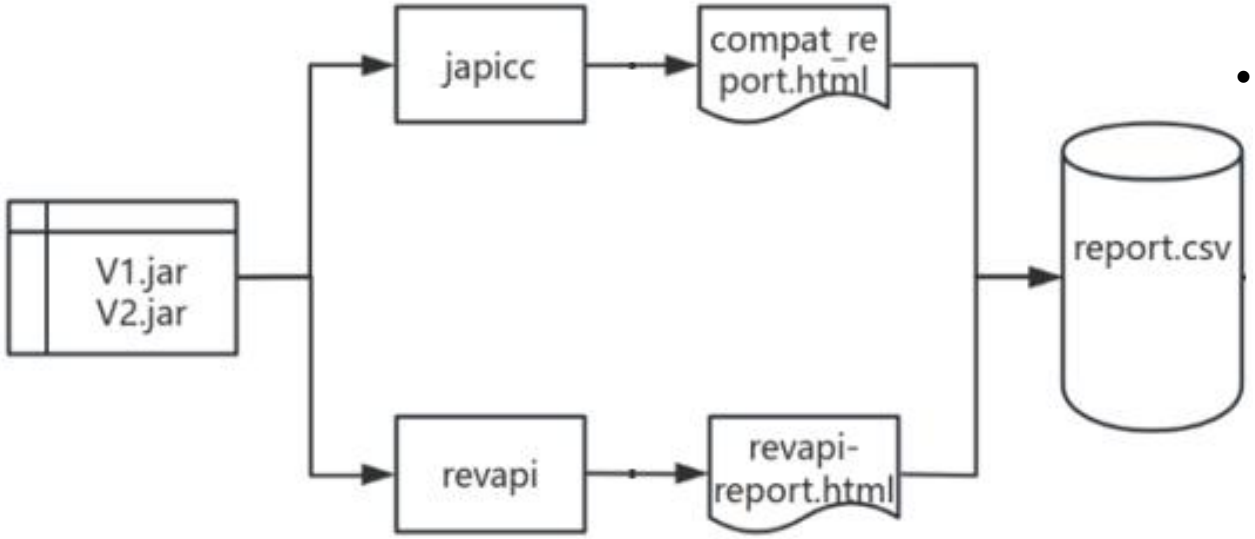
[1] https://lvc.github.io/abi-compliance-checker/

版本间兼容性-兼容性问题检测

我们的工具**JCBJARE**针对Java开源软件，对**JAPICC**和**Revapi**的兼容性问题检测结果进行了提取和整合

- **JCBJARE**工具**兼容性度量指标**中的**兼容性问题严重程度等级**分别是**High、Medium、Low、No**，分别表示高、中、低和无风险。

JCBJARE	High	Medium	Low	No
JAPICC	High	Medium	Low	Safe
Revapi	Breaking	Potentially Breaking	-	Equivalent / Non Breaking



- 我们分析**JAPICC**和**Revapi**的**兼容性规则（能检测出哪些类型的API变动，以及严重程度的判断）**，对其进行了整合。其中**JAPICC**和**Revapi**共有的兼容性规则有**23个**，**Revapi**特有兼容性规则有**14个**，**JAPICC**特有兼容性规则有**11个**。

版本间兼容性-兼容性问题检测

我们的工具**JCBJARE**针对Java开源软件，对**JAPICC**和**Revapi**的兼容性问题检测结果进行了提取和整合

- 由此，在我们的工具**JCBJARE**中，共有**48个**兼容性规则及其严重程度等级。

Metrics/JCBJARE	Severity	Effect
java.class.removed	High	类被移除，从而找不到该类和其中的字段或方法
java.method.visibilityReduced	High	方法可见性降低，从而找不到该方法
java.method.nowFinal	Medium	方法被final修饰符修饰，无法被继承重写，但可以正常被调用
java.superClass.added	Low	类新增了一个父类，可能出现字段或方法的重复
java.field.noLongerFinal	No	字段不再被final修饰，由常量变为非常量
…	…	…

严重程度等级的设定除了参考Revapi和JAPICC工具外，还参考Java语言规范[1]和业界的Java兼容性规范[2][3]。

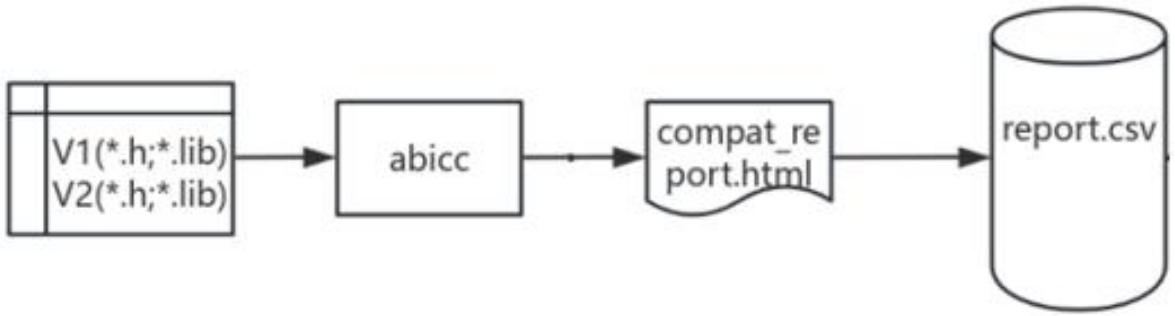
[1] https://docs.oracle.com/javase/specs/jls/se8/html/index.html
[2] https://github.com/eclipse-platform/eclipse.platform/blob/master/docs/Evolving-Java-based-APIs-2.md
[3] https://danamlund.dk/backwards_compatibility_with_java_bytecode_editing.html

版本间兼容性-兼容性问题检测

我们的工具**CABICC**针对C/C++开源软件，对**ABICC**的兼容性问题检测结果进行了提取和整合

- **CABICC**工具直接参照ABICC工具的严重程度等级划分，将**兼容性度量指标**中**兼容性问题严重程度等级**指定为**High、Medium、Low、No**，分别表示高、中、低和无风险。

CABICC工具	High	Medium	Low	No
ABICC工具	High	Medium	Low	No



- **ABICC**共有98条兼容性检测规则**（能检测出哪些类型的API变动，以及严重程度的判断）**

版本间兼容性-兼容性问题检测

我们的工具**CABICC**针对C/C++开源软件，对**ABICC**的兼容性问题检测结果进行了提取和整合

- 在我们的工具**CABICC**中，共有**98个**兼容性度量指标及其严重程度等级。

Metrics/CABICC	Language	Severity	Effect
c.value.memberRemoved	C/C++	High	enum类型中某个枚举值被删除，导致找不到该枚举值
c.class.pureVirtualMethodAdded	C++	High	类中新增纯虚方法，导致该类必须被继承、该纯虚方法必须被重写
c.field.PointerLevelChanged	C++	Medium	类中的某个字段指针级别发生改变，可能导致类型不兼容或语义发生改变
c.constant.changed	C/C++	Low	某个常数的值发生改变，可能语义发生改变
c.constant.added	C/C++	No	新增了一个常数
…		…	…

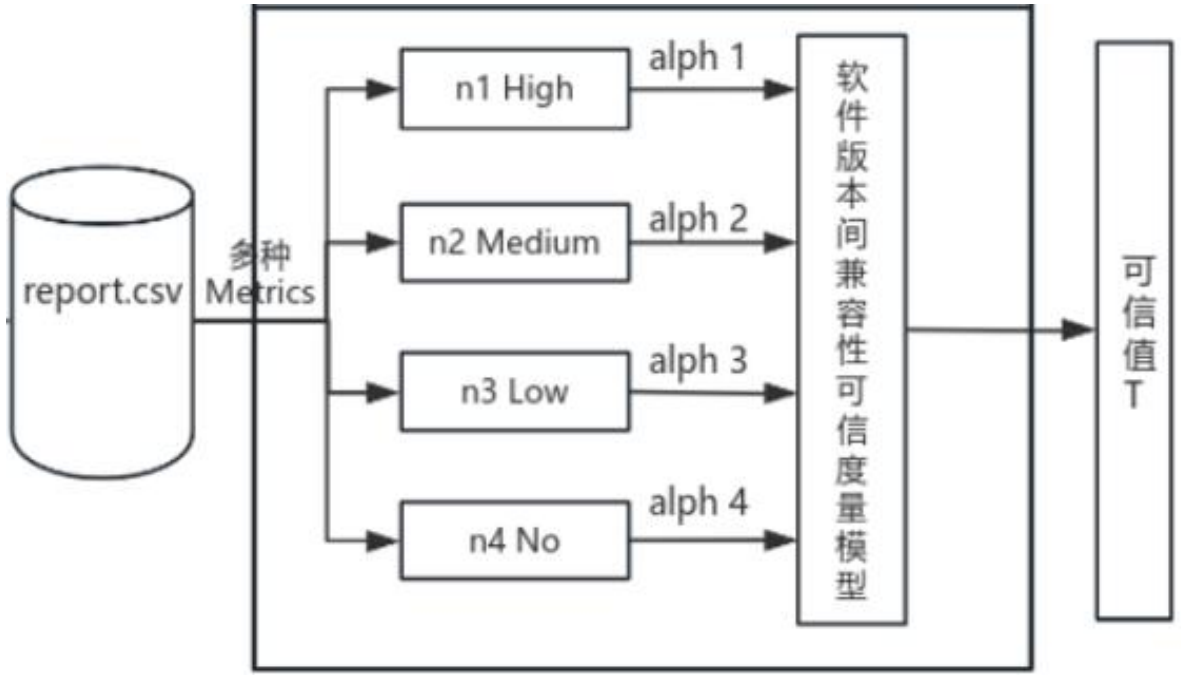
严重程度等级的设定除了参考ABICC工具外，还参考业界的C/C++兼容性规范[1][2]。

[1] https://community.kde.org/Policies/Binary_Compatibility_Issues_With_C++
[2] https://www.ros.org/reps/rep-0009.html

版本间兼容性-兼容性可信度量

- 在检测完开源软件版本间兼容性问题（含严重程度分级），获取了兼容性度量元后，我们需要对其开源软件版本间的兼容性进行可信量化分析。

- 由此，我们提出了 **开源软件软件版本间兼容性可信度量模型**，该模型能够根据兼容性度量指标及其严重程度，对开源软件版本间兼容性进行可信度量，从而获得**可信值T**。



版本间兼容性-兼容性可信度量

- 在设计可信度量模型时，我们的设计灵感来源是**玻尔兹曼熵公式**[1]：

$$S = k * \ln\Omega$$

其中，S表示物质无序的度量，是宏观物理量，而Ω 是微观物理量，k是玻尔兹曼常数，该公式可以通过微观状态反映宏观状态。

- 在开源软件版本间兼容性可信度量中，**不同严重程度度量指标的出现次数**，反映了导致软件出现意外的风险程度，属于软件的**微观状态**；而**软件版本间兼容性可信度**反映了软件的行为，属于系统的**宏观状态**。

- 因此，我们利用玻尔兹曼熵，**用风险的微观状态来表征开源软件版本间兼容性可信度的宏观状态**。

版本间兼容性-兼容性可信度量

- 参照玻尔兹曼熵公式，我们计算**软件版本间兼容性问题严重程度**的**玻尔兹曼加权熵**H：

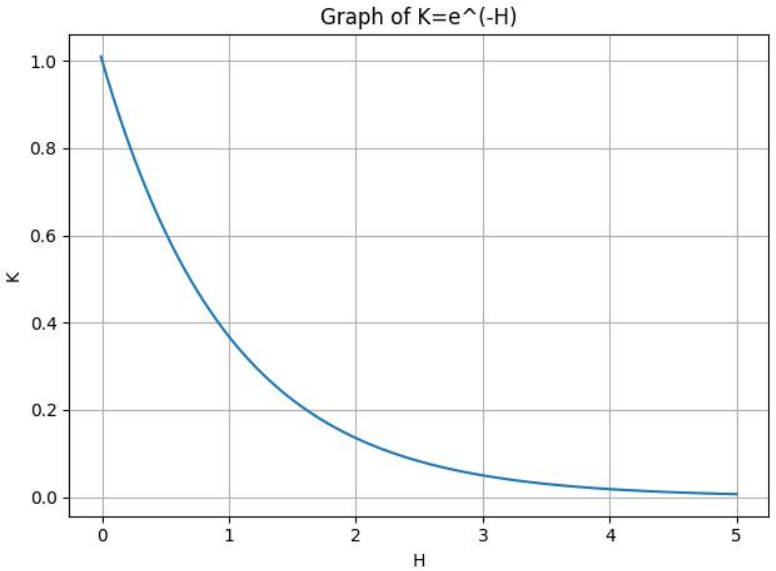
$$H = \sum_{i} \alpha i * \lg(n_i + 1) \qquad \alpha_i = \frac{\lambda_i}{\lambda_h + \lambda_m + \lambda_l + \lambda_o} \qquad i \in (h, m, l, o)$$

其中，

- n_h、n_m、n_l、n_o分别表示检测出的**严重程度**为High、Medium、Low、No的**兼容性问题数量**；
- λ_h、λ_m、λ_l、λ_o分别表示High、Medium、Low、No的**风险因子**，我们根据**近似黄金分割比**设置各风险因子的值：$\lambda_h = 0.9$，$\lambda_m = 0.7$，$\lambda_l = 0.4$，$\lambda_o = 0.1$；
- α_h、α_m、α_l、α_o分别表示High、Medium、Low、No的**风险因子归一化权重**；

- 在公式中$\lg(n_i + 1)$取$n_i + 1$一方面是为了满足对数函数的定义域，另一方面是为了当n_i为0时，该严重程度对应的熵$\alpha_i * \lg(n_i + 1)$为0，表示没有兼容性风险。
- 在此，版本间兼容性可信度量考虑了各种严重程度兼容性问题出现的次数。因为新旧版本源代码是固定的，**出现越严重的不兼容问题，且出现次数越多，则版本间的不兼容程度就越大，熵也会增加。**

版本间兼容性-兼容性可信度量



- 然后，我们需要根据**玻尔兹曼加权熵**H，通过以下公式得到**可信值**K：

$$K = e^{-H}$$

H：软件版本间不兼容严重程度的熵，取值范围为$[0, +\infty)$，该值越大，说明软件两版本间不兼容度越大，两者越不兼容。

K：软件版本间兼容性可信度量值，取值范围为$(0,1]$。该值越大，说明软件两版本间兼容性可信性越高，两版本间越兼容。

- 为了将**最终可信值**T范围映射到$[1,10]$，我们对该模型进一步改进，得到如下最终的**软件版本间兼容性可信度量模型**：

$$T = \begin{cases} 10 * e^{-H} (0.1 \le e^{-H} \le 1) \\ 1 (0 \le e^{-H} < 0.1) \end{cases} \qquad H = \sum_{i} \alpha i * \lg(n_i + 1) \qquad \alpha_i = \frac{\lambda_i}{\lambda_h + \lambda_m + \lambda_l + \lambda_o} \qquad i \in (h, m, l, o)$$

- n_h、n_m、n_l、n_o分别表示检测出的**严重程度**为High、Medium、Low、No的**兼容性问题数量**；
- λ_h、λ_m、λ_l、λ_o分别表示High、Medium、Low、No的**风险因子**，我们根据**近似黄金分割比**设置各风险因子的值：$\lambda_h = 0.9$，$\lambda_m = 0.7$，$\lambda_l = 0.4$，$\lambda_o = 0.1$；

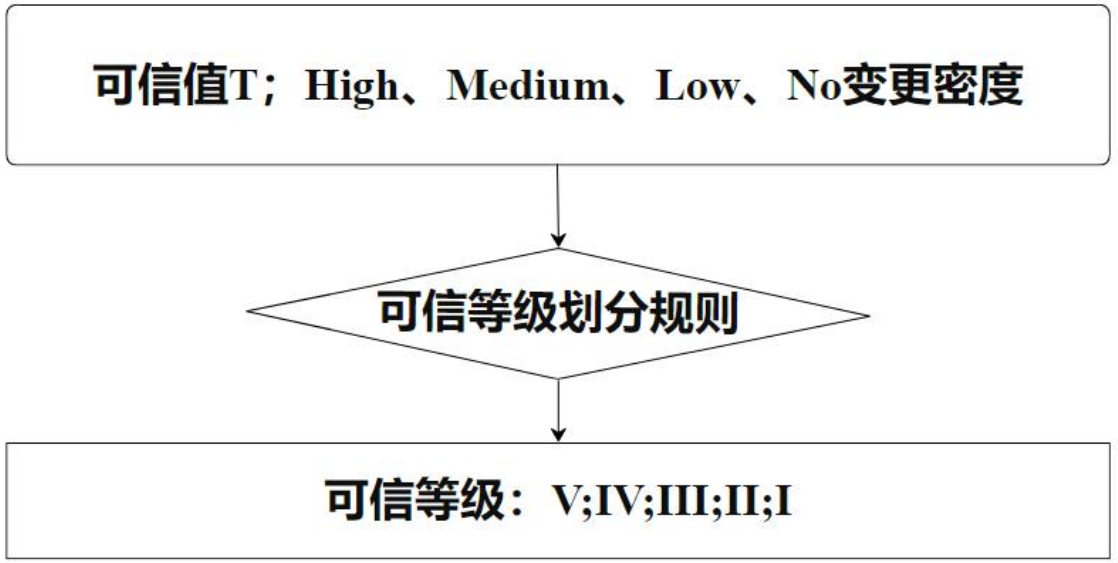
版本间兼容性-兼容性可信度量

- 在计算得到**开源软件版本间兼容性可信值**后，我们还要进行**开源软件版本间兼容性可信等级划分**

- 在进行可信等级划分时，我们不仅考虑**开源软件版本间兼容性可信值**，还要考虑**版本间API变更的规模（即API变更的数量）**，变更规模越大，不兼容度越高，可信度越低。

- 而API变更包括：**良性变更**（向下兼容的变更：**严重程度为No**）和**破坏性变更**（不向下兼容的变更：**严重程度为High、Medium、Low**），我们都会将其纳入考虑；

- 而另一方面，对于**不兼容变更数量**定义划分规则时的阈值是很难确定的，所以我们最终考虑使用**不兼容变更密度**作为可信等级划分的条件之一。



兼容性可信等级：V > IV > III > II > I

版本间兼容性-兼容性可信度量

- 最终，我们得到的**开源软件版本间兼容性可信等级划分**

等级	T	d_h	d_m	d_l	或满足
V级:	$9 \le T$	$d_h = 0$	$d_m = 0$	$d_l \le 0.4$	无
IV级:	$7 \le T < 9$	$d_h = 0$	$d_m \le 0.4$	$d_l \le 0.7$	或$9 \le T$且不能评为 V 级别
III级	$4 \le T < 7$	$d_h \le 0.4$	$d_m \le 0.7$	-	或$7 \le T$且不能评为 IV 级别及以上
II级	$2 \le T < 4$	$d_h \le 0.7$	-	-	或$4 \le T$且不能评为 III 级别及以上
I级	$1 \le T < 2$	-	-	-	或$2 \le T$且不能评为 II 级别及以上

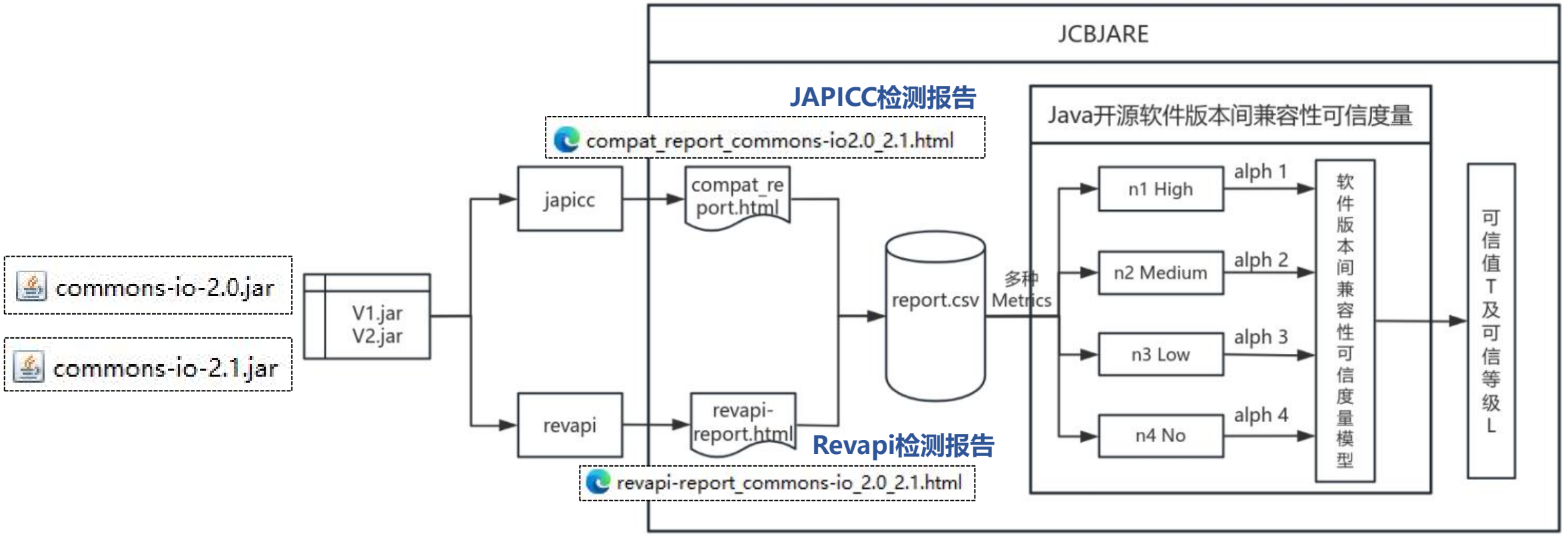
- 其中，T表示最终可信值， d_h、d_m、d_l分别是**High、Medium、Low的变更密度**，计算公式如下所示：

$$d_i = \frac{n_i}{n_h + n_m + n_l + n_o} (i = h, m, l, o)$$

n_h、n_m、n_l、n_o分别表示检测出的**严重程度**为High、Medium、Low、No的**兼容性问题数量**

版本间兼容性-例1

- 我们使用**JCBJARE工具**测试**Java开源软件commons-io v2.0版本和v2.1版本**之间的兼容性



版本间兼容性-例1

- 我们使用**JCBJARE工具**测试**Java开源软件commons-io v2.0版本和v2.1版本**之间的兼容性

Metrics	Severity	API	Description
java.class.defaultSerializatior	High	org.apache.commons.io.monitor.FileAlterationObserver	The default serialization ID for the class has changed.
java.class.fieldAdded	Low	org.apache.commons.io.FileUtils	Field ONE_EB has been added to this class.
java.class.fieldAdded	Low	org.apache.commons.io.FileUtils	Field ONE_PB has been added to this class.
java.class.fieldAdded	Low	org.apache.commons.io.FileUtils	Field ONE_TB has been added to this class.
java.class.fieldAdded	Low	org.apache.commons.io.FileUtils	Field ONE_YB has been added to this class.
java.class.fieldAdded	Low	org.apache.commons.io.FileUtils	Field ONE_ZB has been added to this class.
java.method.added	No	org.apache.commons.io.input.ClassLoaderObjectInputSt	Added Methods

兼容性风险清单（部分内容）

严重程度	兼容性问题数量
High	1
Medium	0
Low	5
No	46

各严重程度兼容性问题数量：$n_h = 1,\ n_m = 0,\ n_l = 5,\ n_o = 46$

兼容性问题严重程度的**玻尔兹曼加权熵** $H = \sum_{i\in\{h,m,l,n\}} \alpha_i \log(n_i + 1)$

$$\alpha_i = \frac{\lambda_i}{\lambda_h + \lambda_m + \lambda_l + \lambda_o} \qquad i \in (h,m,l,o)$$

$$\lambda_h = 0.9,\quad \lambda_m = 0.7,\quad \lambda_l = 0.4,\quad \lambda_o = 0.1;$$

$$= \frac{0.9}{2.1}\log(1+1) + \frac{0.7}{2.1}\log(0+1) + \frac{0.4}{2.1}\log(5+1) + \frac{0.1}{2.1}\log(46+1)$$
$$= 0.356856$$

因为$e^{-H} = 0.699873 > 0.1$

所以最终**兼容性可信值** $T = 10e^{-H} = 6.99873$

版本间兼容性-例1

- 我们使用**JCBJARE工具**测试**Java开源软件commons-io v2.0版本和v2.1版本**之间的兼容性

等级	T	d_h	d_m	d_l	或满足
V级:	$9 \le T$	$d_h = 0$	$d_m = 0$	$d_l \le 0.4$	无
IV级:	$7 \le T < 9$	$d_h = 0$	$d_m \le 0.4$	$d_l \le 0.7$	或$9 \le T$且不能评为 V 级别
III级	$4 \le T < 7$	$d_h \le 0.4$	$d_m \le 0.7$	-	或$7 \le T$且不能评为 IV 级别及以上
II级	$2 \le T < 4$	$d_h \le 0.7$	-	-	或$4 \le T$且不能评为 III 级别及以上
I级	$1 \le T < 2$	-	-	-	或$2 \le T$且不能评为 II 级别及以上

各严重程度兼容性问题数量：$n_h = 1,\ n_m = 0,\ n_l = 5,\ n_o = 46$

兼容性可信值 $T = 6.99873$

$$d_i = \frac{n_i}{n_h + n_m + n_l + n_o} (i = h, m, l, o)$$

各严重程度兼容性问题密度：$d_h = \frac{1}{52} = 0.019$　　$d_m = \frac{0}{52} = 0$

$d_l = \frac{5}{52} = 0.096$　　$d_o = \frac{46}{52} = 0.885$

最终求得**兼容性可信值**$T = 6.99873$，有$4 \le T < 7$且有$d_h \le 0.4$和$d_m \le 0.7$，所以最终兼容性可信等级为**III级**

版本间兼容性-例2

语言	Version Pair	n_h, n_m, n_l, n_o	d_h, d_m, d_l, d_o	T	Level
Java	Xchart 3.5.4-3.6.1	20, 0, 0, 107	0.157, 0.0, 0.0, 0.843	5.15	III
Java	swagger-core 1.6.2-2.1.4	273, 1, 0, 366	0.427, 0.002, 0.0, 0.572	2.816	II
Java	swagger-core 1.6.6-2.1.13	280, 1, 0, 383	0.422, 0.002, 0.0, 0.577	2.8	II
Java	Java-Chassis 1.3.1-2.0.2	243, 32, 3, 193	0.516, 0.068, 0.006, 0.41	1.733	I
Java	Tomcat 9.0.81-9.0.82	4, 4, 0, 0	0.5, 0.5, 0.0, 0.0	5.817	II
Java	mybatis-plus 3.3.1-3.4.2	0, 0, 0, 0	0.0,0.0,0.0,0.0	10.0	V
Java	commons-beanutils_1.6_1.8.0	22, 3, 22, 482	0.042, 0.006, 0.042, 0.911	3.099	II
Java	commons-codec_1.6_1.7	8, 7, 1, 68	0.095, 0.083, 0.012, 0.81	4.253	III
Java	commons-fileupload_1.2_1.2.2	1, 0, 2, 30	0.03, 0.0, 0.061, 0.909	7.476	III
Java	commons-fileupload_1.3_1.3.1	0, 1, 0, 2	0.0, 0.333, 0.0, 0.667	8.842	IV
C/C++	nftables0.9.4-1.0.0	0, 0, 0, 2	0.0, 0.0, 0.0, 1.0	9.775	V
C/C++	Tcl/Tk8.6.10-8.6.12	0, 0, 3, 12	0.0, 0.0, 0.2, 0.8	8.456	IV
C/C++	JsonCpp1.7.7-1.8.0	0, 0, 3, 1	0.0, 0.0, 0.75, 0.25	8.79	III
C/C++	JsonCpp1.7.7-1.9.4	16, 9, 48, 29	0.157, 0.088, 0.471, 0.284	2.857	II
…	…	…	…	…	…

1. **可信等级达到V**的没有兼容性问题，可放心使用；
2. **可信等级为IV**的可能有兼容性问题，使用时需要注意版本变更的API；
3. **可信等级为III、II和I**的一定有兼容性问题，需特别注意。

大　纲

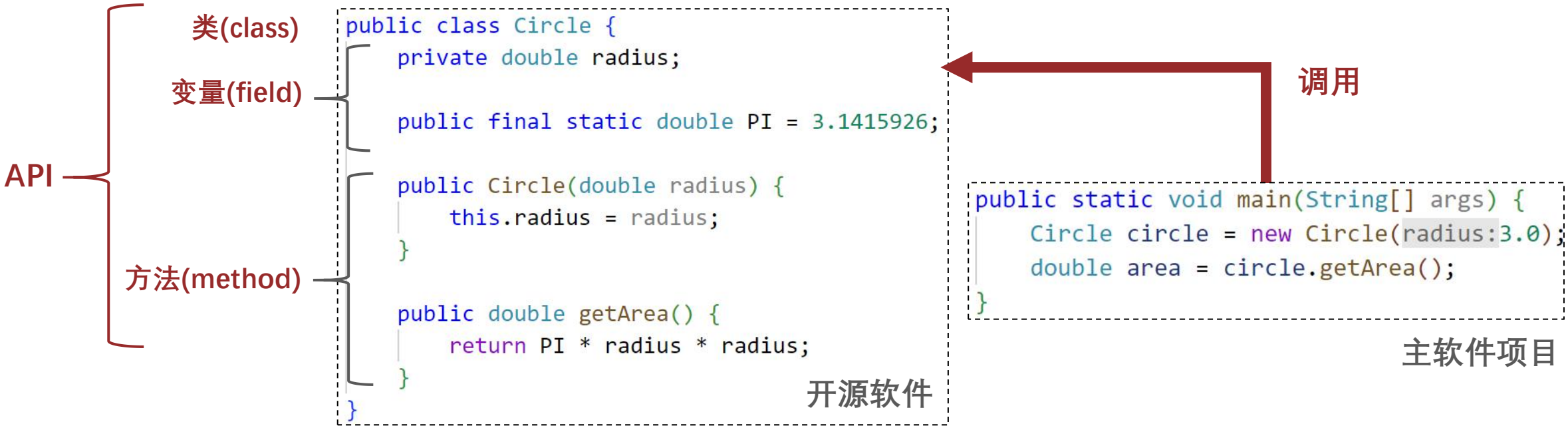
前置知识

开源软件版本间兼容性

开源软件软件间兼容性

软件间兼容性

- **开源软件版本间兼容性**：从开源软件的**开发者**角度，不知道主软件的情况（哪些API被调用，调用情况如何），关注**版本间的所有API变动（差异）**
- **开源软件软件间兼容性**：从开源软件的**使用者**角度，知道主软件的情况（哪些API被调用，调用情况如何），关注**所使用的开源软件版本间API变动（差异）对API调用造成的影响**



软件间兼容性-兼容性问题检测

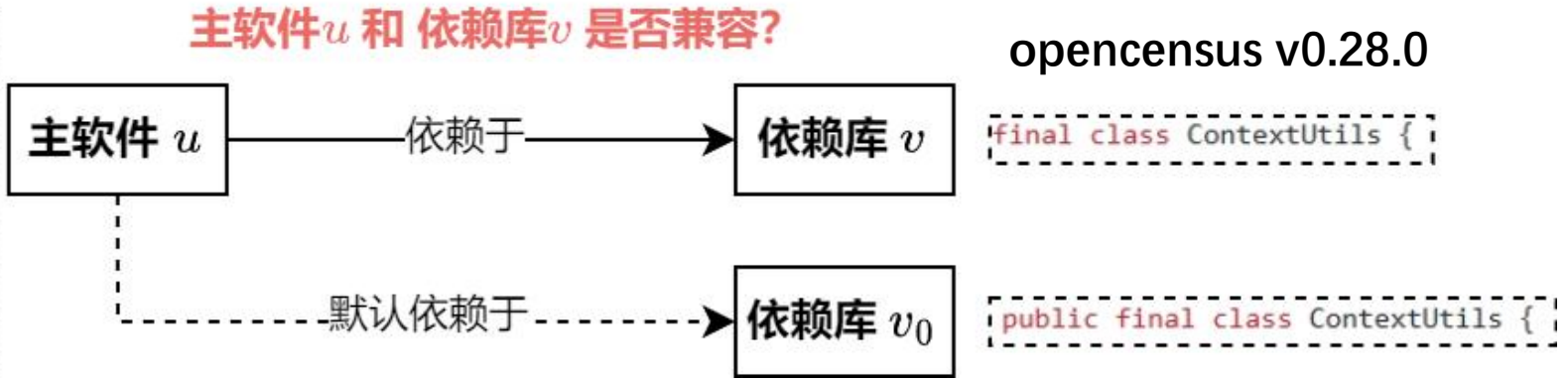
我们首先关注**有直接依赖关系的主软件和开源软件之间的兼容性**

- 当**主软件直接依赖于开源软件**时，而不同版本的主软件和不同版本的直接依赖软件间可能存在兼容性问题。主软件依赖并调用开源软件提供的API，而开源软件不同的版本的API可能存在差异，因此，我们将兼容定义为：**开源软件当前版本所提供的API能够满足主软件当前版本的所有API调用需求。**

- 当主软件的开发者在第一次引入直接依赖的开源软件或者更改所依赖的版本时，会显式地指定所依赖的版本（或版本范围），我们称其为**默认版本**。所以，我们认为：**主软件当前版本所直接依赖的软件默认版本，其提供的API是可以完全满足主软件当前版本的所有API调用需求的。**

```
@Override
public Context filterContext(Context context) {
  // Access directly the unsafe trace API to cre
  // because gRPC always creates a new Context t
  // inherit from the parent Context.
  return ContextUtils.withValue(context, span);
}
```

grpc-census v1.36.3



opencensus v0.28.0

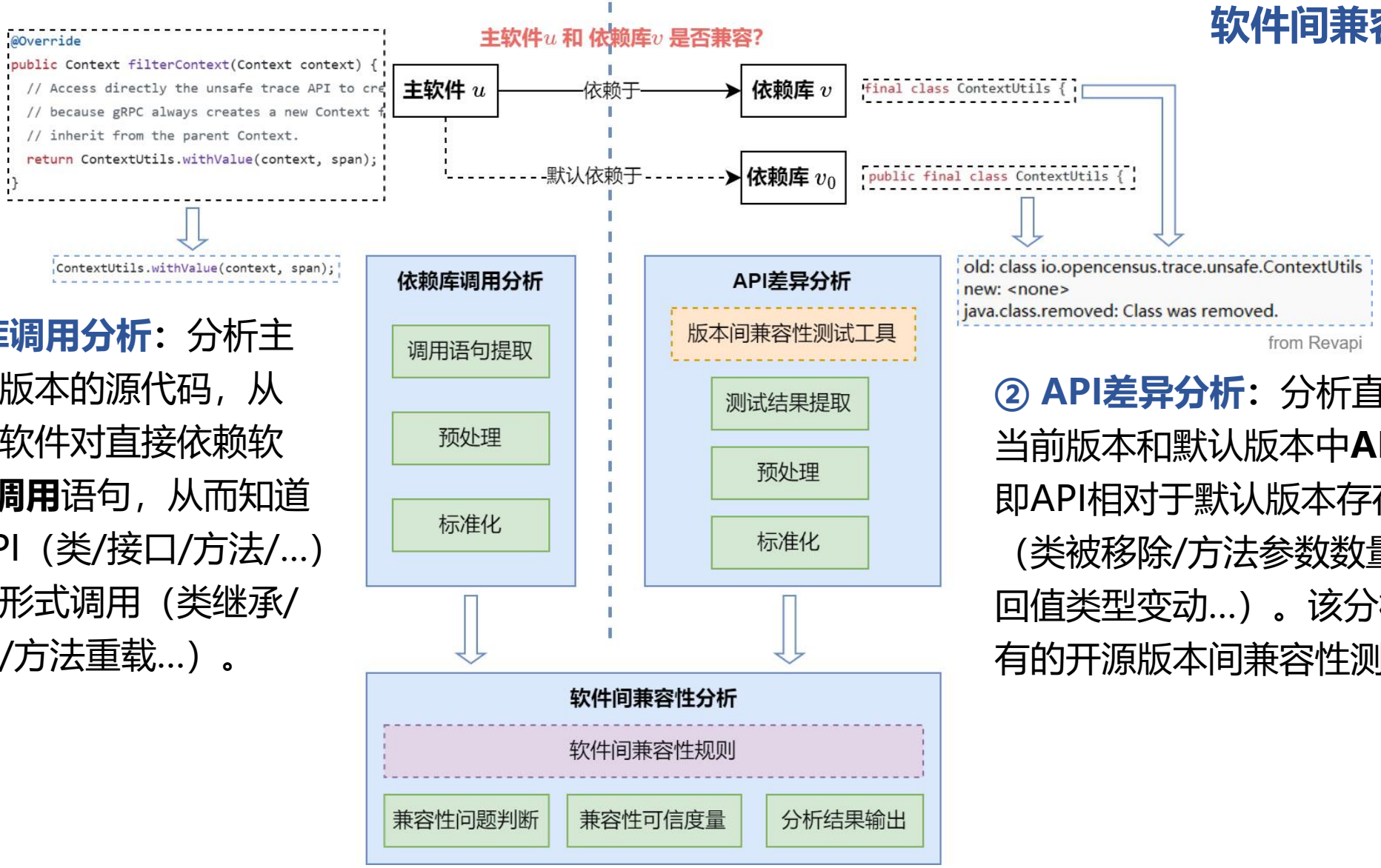
```
final class ContextUtils {
```

```
public final class ContextUtils {
```

opencensus v0.28.1

软件间兼容性-兼容性问题检测

软件间兼容性测试框架



① **依赖库调用分析**：分析主软件当前版本的源代码，从中提取主软件对直接依赖软件中**API调用**语句，从而知道有哪些API（类/接口/方法/...）被以何种形式调用（类继承/方法调用/方法重载...）。

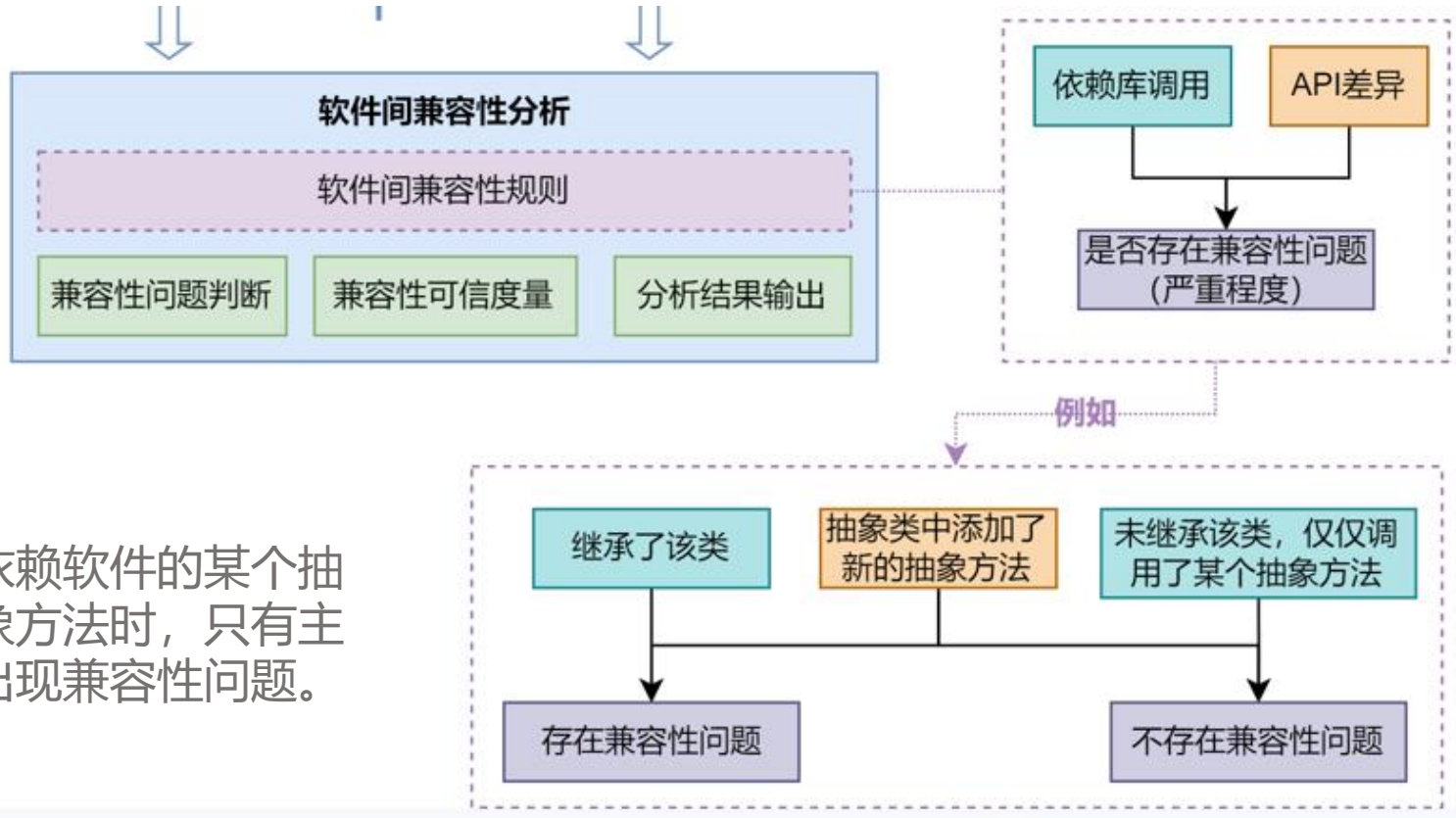
② **API差异分析**：分析直接依赖软件的当前版本和默认版本中**API存在的差异**，即API相对于默认版本存在哪些变动（类被移除/方法参数数量变动/方法返回值类型变动...）。该分析可以借助已有的开源版本间兼容性测试工具进行。

软件间兼容性-兼容性问题检测

软件间兼容性测试框架

③ 软件间兼容性分析：针对从主软件源代码中的提取的API调用语句和直接依赖软件中的API差异，我们要**分析这些API差异对API调用造成的影响**，因为主软件通常只会调用直接依赖软件提供的部分API。

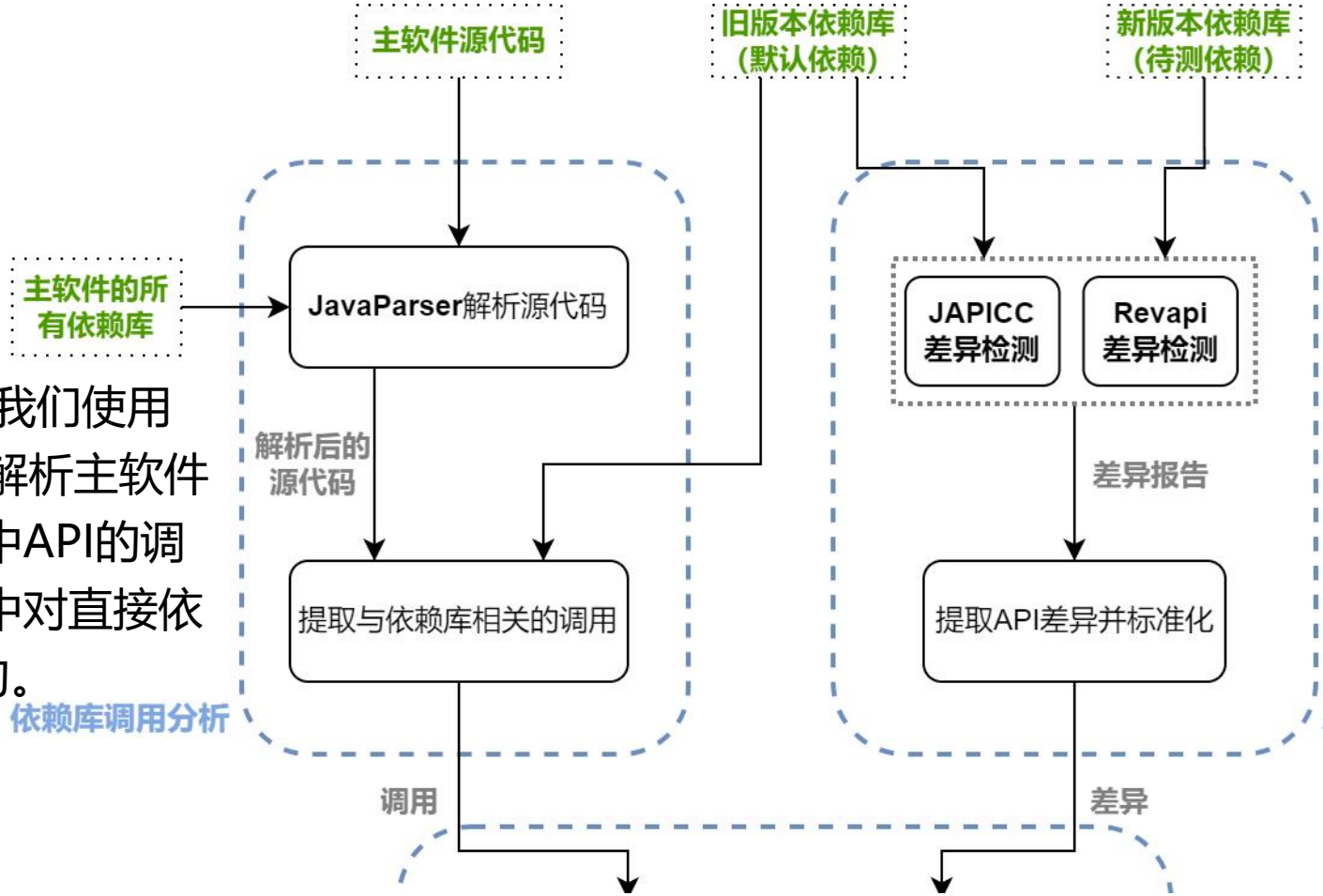
我们基于预先设定的**软件间兼容性规则**，来判断是否会存在**兼容性问题**，以及**兼容性问题的严重程度。**



如右图所示，当直接依赖软件的某个抽象类中添加了新的抽象方法时，只有主软件继承了该类才会出现兼容性问题。

软件间兼容性-兼容性问题检测

- 我们依据前述**软件间兼容性测试框架**，设计并实现**Java-Compatibility工具**



① 依赖库调用分析：我们使用JavaParser[1]框架来解析主软件的源代码，并分析其中API的调用情况，从而提取其中对直接依赖软件的**API调用**语句。

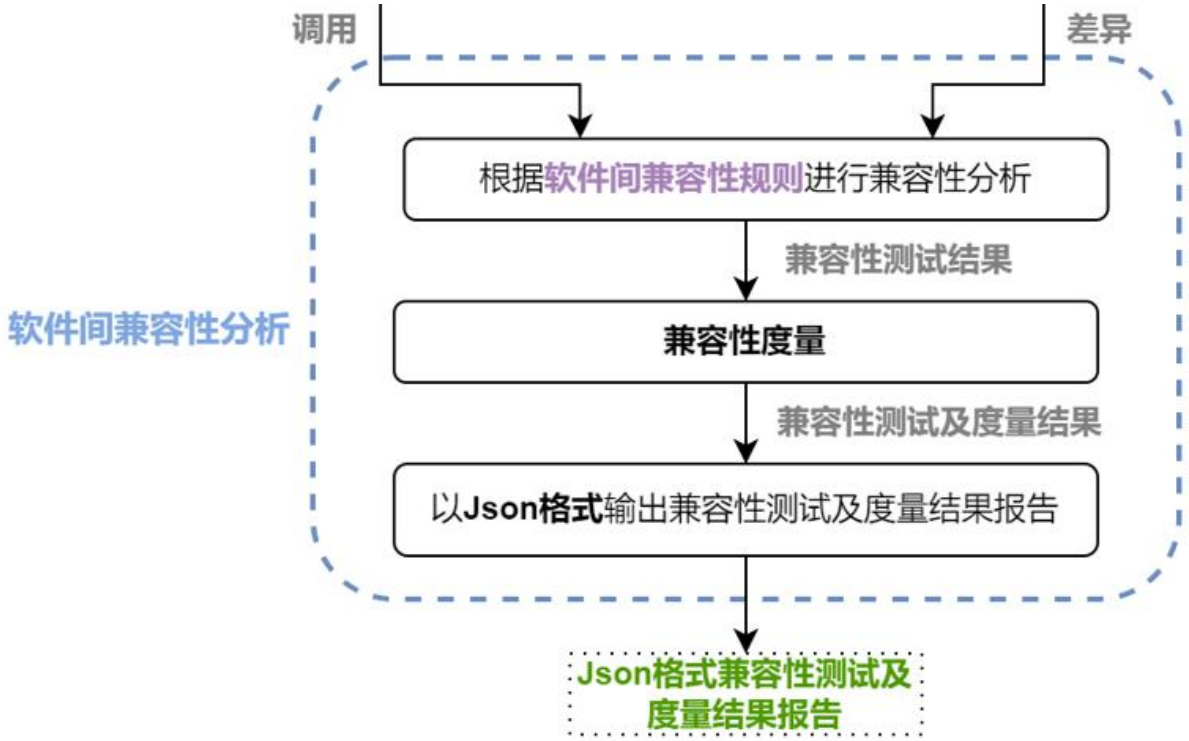
② API差异分析：我们使用JAPICC[2]和Revapi[3]工具来检测直接依赖软件当前版本和默认版本的**API差异**，读取差异报告并对其进行标准化

[1] https://github.com/javaparser/javaparser
[2] https://lvc.github.io/japi-compliance-checker/
[3] https://revapi.org

软件间兼容性-兼容性问题检测

- 我们依据前述**软件间兼容性测试框架**，设计并实现**Java-Compatibility工具**

③ 软件间兼容性分析：我们则对**Java软件间兼容性规则**和**兼容性可信度量**进行了实现，最后得到兼容性测试和度量结果，并以Json格式输出结果报告。



- **Java软件间兼容性规则**参考**阶段一的Java兼容性度量指标**进行设计，在**API差异**的基础上，考虑具体的**API调用**情况，共设计并实现了**42条**Java软件间兼容性规则。

当**一个API调用**受到**多个API差异**的影响时，受到影响的**严重程度取高值**。
（High>Medium>Low>No）

软件间兼容性-兼容性问题检测

- 我们依据前述**软件间兼容性测试框架**，设计并实现**Java-Compatibility工具**

③ 软件间兼容性分析：我们则对**Java软件间兼容性规则**和**兼容性可信度量**进行了实现，最后得到兼容性测试和度量结果，并以Json格式输出结果报告。

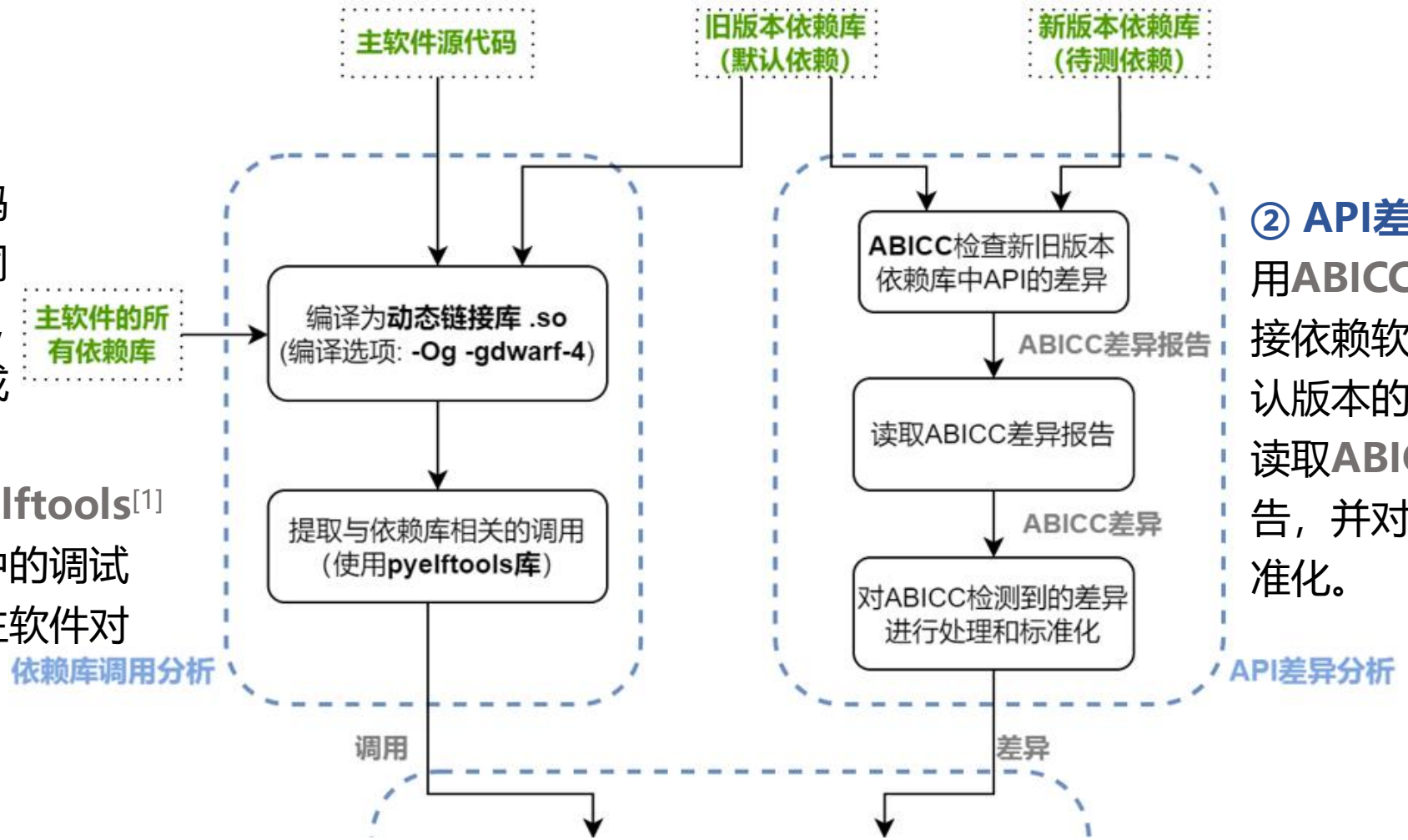
- **Java软件间兼容性规则**参考**阶段一的Java兼容性度量指标**进行设计，在**API差异**的基础上，考虑具体的**API调用**情况，共设计并实现了**42条**Java软件间兼容性规则。

API差异	API差异描述	API调用	严重程度等级
Class_Or_Interface_Removed	移除类/接口	调用了该类/接口（或者其字段或方法）	High
		未调用	No
Method_Without_Body_Added_To Interface	接口新增待实现方法	该方法所在接口被实现	High
		该方法所在接口未被实现	No
Method_Return_Type_Changed	方法返回值类型变动	该方法被调用	Medium
		未调用	No
Method_Runtim_Exception_Added	方法新增运行时异常	该方法被调用	Low
		未调用	No
…	…	…	…

软件间兼容性-兼容性问题检测

- 我们依据前述**软件间兼容性测试框架**，设计并实现**Cpp-Compatibility工具**

① 依赖库调用分析：
我们将主软件源代码与其所有依赖库一同编译（Linux环境），**保留调试信息**并生成**动态链接库（.so）**。
然后，我们使用**pyelftools**[1]库来读取和分析其中的调试信息，从而提取出主软件对依赖库的调用。



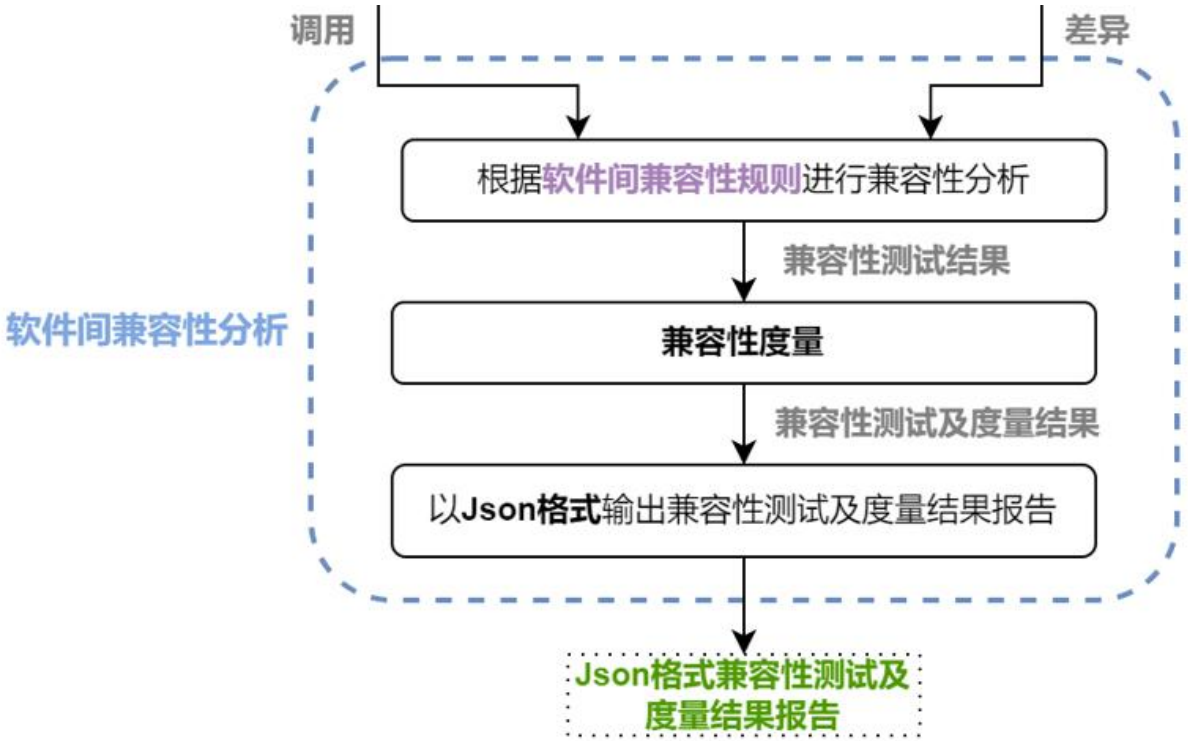
② API差异分析：我们使用**ABICC**[2]工具来检测直接依赖软件当前版本和默认版本的API差异，然后读取**ABICC**生成的差异报告，并对其进行处理和标准化。

[1] https://github.com/eliben/pyelftools
[2] https://lvc.github.io/abi-compliance-checker/

软件间兼容性-兼容性问题检测

- 我们依据前述**软件间兼容性测试框架**，设计并实现**Cpp-Compatibility工具**

③ 软件间兼容性分析：我们则对**C/C++软件间兼容性规则**和**兼容性可信度量**进行了实现，最后得到兼容性测试和度量结果，并以Json格式输出结果报告。



- **C/C++软件间兼容性规则**参考**阶段一的C/C++兼容性度量指标**进行设计，在**API差异**的基础上，考虑具体的**API调用**情况，共设计并实现了**66条**Java软件间兼容性规则。

当**一个API调用**受到**多个API差异**的影响时，受到影响的**严重程度取高值**。
（High>Medium>Low>No）

软件间兼容性-兼容性问题检测

- 我们依据前述**软件间兼容性测试框架**，设计并实现**Cpp-Compatibility工具**

③ 软件间兼容性分析：我们则对**C/C++软件间兼容性规则**和**兼容性可信度量**进行了实现，最后得到兼容性测试和度量结果，并以Json格式输出结果报告。

- **C/C++软件间兼容性规则**参考**阶段一的C/C++兼容性度量指标**进行设计，在**API差异**的基础上，考虑具体的**API调用**情况，共设计并实现了**66条**Java软件间兼容性规则。

API差异	API差异描述	语言限定	API调用	严重程度等级
Removed_Field	数据类型中删除字段	C/C++	该字段被调用	High
			未调用	No
Added_Pure_Virtual_Method	类新增纯虚方法	C++	该类被调用	High
			该类未被调用	No
Field_Pointer_Level_Changed	字段指针等级改变	C/C++	该字段被调用	Medium
			未调用	No
Changed_Constant	常数值发生改变	C/C++	该常数被调用	Low
			未调用	No
…	…	…	…	…

软件间兼容性-兼容性可信度量

- 在对主软件与直接依赖软件的兼容性进行测试后，我们可以得到**检测出的兼容性问题及其严重程度。**
- 在软件间兼容性规则中，我们参照**阶段一**将严重程度分级为**High、Medium、Low、No四级**，因此我们也采用同样的**（软件间）兼容性可信度量模型**，来计算**软件间可信值**$T \in [1,10]$。

$$T = \begin{cases} 10e^{-H} & ,0.1 \le e^{-H} < 1 \\ 1 & ,0 \le e^{-H} < 0.1 \end{cases} \qquad H = \sum_{i \in \{h,m,l,o\}} \alpha_i \log_{10}(n_i + 1) \qquad \alpha_i = \frac{\lambda_i}{\lambda_h + \lambda_m + \lambda_l + \lambda_o} \qquad i \in \{h,m,l,o\}$$

其中

- λ_i：不同严重程度的风险因子，按近似黄金比例逐级递减的方法进行定义，严重程度High的风险因子为$\lambda_h = 0.9$、Medium为$\lambda_m = 0.7$、Low为$\lambda_l = 0.4$、No为$\lambda_o = 0.1$；
- α_i：α_h、α_m、α_l和α_o分别表示严重程度为High、Medium、Low和No的风险因子归一化权重；
- n_i：n_h、n_m、n_l和n_o分别表示受到影响的严重程度为High、Medium、Low和No的兼容性问题数量
- H：软件间兼容性影响严重程度的熵；

软件间兼容性-兼容性可信度量

- 与**阶段一**相同，我们利用**可信值**和**不同严重程度的兼容性问题密度**来对兼容性可信等级进行划分，并采用同样的**划分依据**。

等级	T	d_h	d_m	d_l	或满足
V级:	$9 \le T$	$d_h = 0$	$d_m = 0$	$d_l \le 0.4$	无
IV级:	$7 \le T < 9$	$d_h = 0$	$d_m \le 0.4$	$d_l \le 0.7$	或$9 \le T$且不能评为 V 级别
III级	$4 \le T < 7$	$d_h \le 0.4$	$d_m \le 0.7$	-	或$7 \le T$且不能评为 IV 级别及以上
II级	$2 \le T < 4$	$d_h \le 0.7$	-	-	或$4 \le T$且不能评为 III 级别及以上
I级	$1 \le T < 2$	-	-	-	或$2 \le T$且不能评为 II 级别及以上

- 其中，T表示最终可信值，d_h、d_m、d_l分别是**High、Medium、Low的变更密度**，计算公式如下所示：

$$d_i = \frac{n_i}{n_h + n_m + n_l + n_o} (i = h, m, l, o)$$

n_h、n_m、n_l、n_o分别表示检测出的**严重程度**为High、Medium、Low、No的**兼容性问题数量**

软件间兼容性-例1

- （Java）主软件**hudi-common**的**0.10.0**版本直接依赖于**httpclient**，其默认依赖的版本为**4.4.1**，我们单独测试并度量**hudi-common v0.10.0**版本与**httpclient v4.1.3**的兼容性。


```
"usageAmount": 25,
"differenceAmount": 471,
"highCount": 11,
"mediumCount": 0,
"lowCount": 0,
"noCount": 14,
```


- **hudi-common v0.10.0调用httpclient的API共25 次，而待测版本httpclient v4.1.3相较于默认版httpclint v4.4.1存在471个API差异；**

- **其中11个API调用受到API差异影响的严重程度为High，有14个受到影响的严重程度为No。**

- **各严重严重程度兼容性问题数量：** $n_h = 11, n_m = 0, n_l = 0, n_o = 14$


```
"sourceFile": "org.apache.hudi.common.table.view.
RemoteHoodieTableFileSystemView.java",
"beginPosition": "(line 158,col 13)",
"endPosition": "(line 158,col 22)",
"codeSegment": "        new ***URIBuilder***().setHost(serverHost).
setPort(serverPort).setPath(requestPath).setScheme(\"http\");\n",
"elementInfo": "org.apache.http.client.utils.URIBuilder",
"affectedLevel": "HIGH",
"differenceEntries": [
  {
    "syntacticAffectedLevel": "HIGH",
    "semanticAffectedLevel": "NO",
    "differenceType": "ClassOrInterfaceRemoved",
    "oldElementInfo": "class/interface: org.apache.http.client.utils
    URIBuilder",
    "newElementInfo": "",
    "differenceDescription": null
  }
]
```

软件间兼容性-例1

- （Java）主软件**hudi-common**的**0.10.0**版本直接依赖于**httpclient**，其默认依赖的版本为**4.4.1**，我们单独测试并度量**hudi-common v0.10.0**版本与**httpclient v4.1.3**的兼容性。

各严重严重程度兼容性问题数量： $n_h = 11,\ n_m = 0,\ n_l = 0,\ n_o = 14$

兼容性问题严重程度的**玻尔兹曼加权熵** $H = \sum_{i \in \{h,m,l,n\}} \alpha_i \log(n_i + 1)$

$$= \frac{0.9}{2.1}\log(11+) + \frac{0.7}{2.1}\log(0+1) + \frac{0.4}{2.1}\log(0+1) + \frac{0.1}{2.1}\log(14+1)$$
$$= 0.51851$$

$$\alpha_i = \frac{\lambda_i}{\lambda_h + \lambda_m + \lambda_l + \lambda_o} \qquad i \in (h,m,l,o)$$

$\lambda_h = 0.9,\quad \lambda_m = 0.7,\quad \lambda_l = 0.4,\quad \lambda_o = 0.1;$

因为$e^{-H} = 0.5954 > 0.1$

所以最终**兼容性可信值** $T = 10e^{-H} = 5.594$

软件间兼容性-例1

- （Java）主软件**hudi-common**的**0.10.0**版本直接依赖于**httpclient**，其默认依赖的版本为**4.4.1**，我们单独测试并度量**hudi-common v0.10.0**版本与**httpclient v4.1.3**的兼容性。

等级	T	d_h	d_m	d_l	或满足
V级:	$9 \le T$	$d_h = 0$	$d_m = 0$	$d_l \le 0.4$	无
IV级:	$7 \le T < 9$	$d_h = 0$	$d_m \le 0.4$	$d_l \le 0.7$	或$9 \le T$且不能评为 V 级别
III级	$4 \le T < 7$	$d_h \le 0.4$	$d_m \le 0.7$	-	或$7 \le T$且不能评为 IV 级别及以上
II级	$2 \le T < 4$	$d_h \le 0.7$	-	-	或$4 \le T$且不能评为 III 级别及以上
I级	$1 \le T < 2$	-	-	-	或$2 \le T$且不能评为 II 级别及以上

各严重程度兼容性问题数量：$n_h = 11,\ n_m = 0,\ n_l = 0,\ n_o = 14$　　**兼容性可信值** $T = 5.594$

$$d_i = \frac{n_i}{n_h + n_m + n_l + n_o} (i = h, m, l, o)$$

各严重程度兼容性问题密度：$d_h = \frac{11}{25} = 0.44$　　$d_m = \frac{0}{52} = 0$

$d_l = \frac{0}{25} = 0$　　$d_o = \frac{14}{25} = 0.56$

最终求得**兼容性可信值**$T = 6.99873$，有$4 \le T < 7$，但是不满足有$d_h \le 0.4$，所以最终兼容性可信等级为**II级**

软件间兼容性-例2

- （C语言）主软件**libpng 1.6.43**版本，直接依赖于软件**zlib 1.0.4**版本。

- 我们选取了**zlib**的**0.8、0.9、0.95、1.0、1.0.4、1.1.0、1.1.4、[1.2.0, 1.2.13]、1.3**共22个版本，检测并度量其与**libpng 1.6.43**版本的兼容性。

zlib 版本	High	Medium	Low	No	可信值	可信等级
0.8	19	2	51	374	3.116	II
0.9	19	2	51	374	3.116	II
0.95	19	2	17	408	3.396	II
1.0	0	0	0	446	8.814	IV
1.0.4	**0**	**0**	**0**	**0**	**10.0**	**V**
1.1.0	0	0	0	446	8.814	IV
1.1.4	0	0	0	446	8.814	IV
1.2.0	0	0	0	446	8.814	IV
1.2.1	0	0	0	446	8.814	IV
1.2.2	0	0	0	446	8.814	IV
1.2.3	0	0	0	446	8.814	IV
1.2.4	0	0	0	446	8.814	IV
1.2.5	0	0	0	446	8.814	IV
1.2.6	0	0	17	429	6.945	III
1.2.7	0	0	17	429	6.945	III
1.2.8	0	0	17	429	6.945	III
1.2.9	0	0	17	429	6.945	III
1.2.10	0	0	17	429	6.945	III
1.2.11	0	0	17	429	6.945	III
1.2.12	0	0	17	429	6.945	III
1.2.13	0	0	17	429	6.945	III
1.3	0	1	17	428	6.283	III

软件间兼容性-例3

- （C++）主软件**spdlog 1.14.1**版本，直接依赖于软件**fmt 10.2.1**版本。

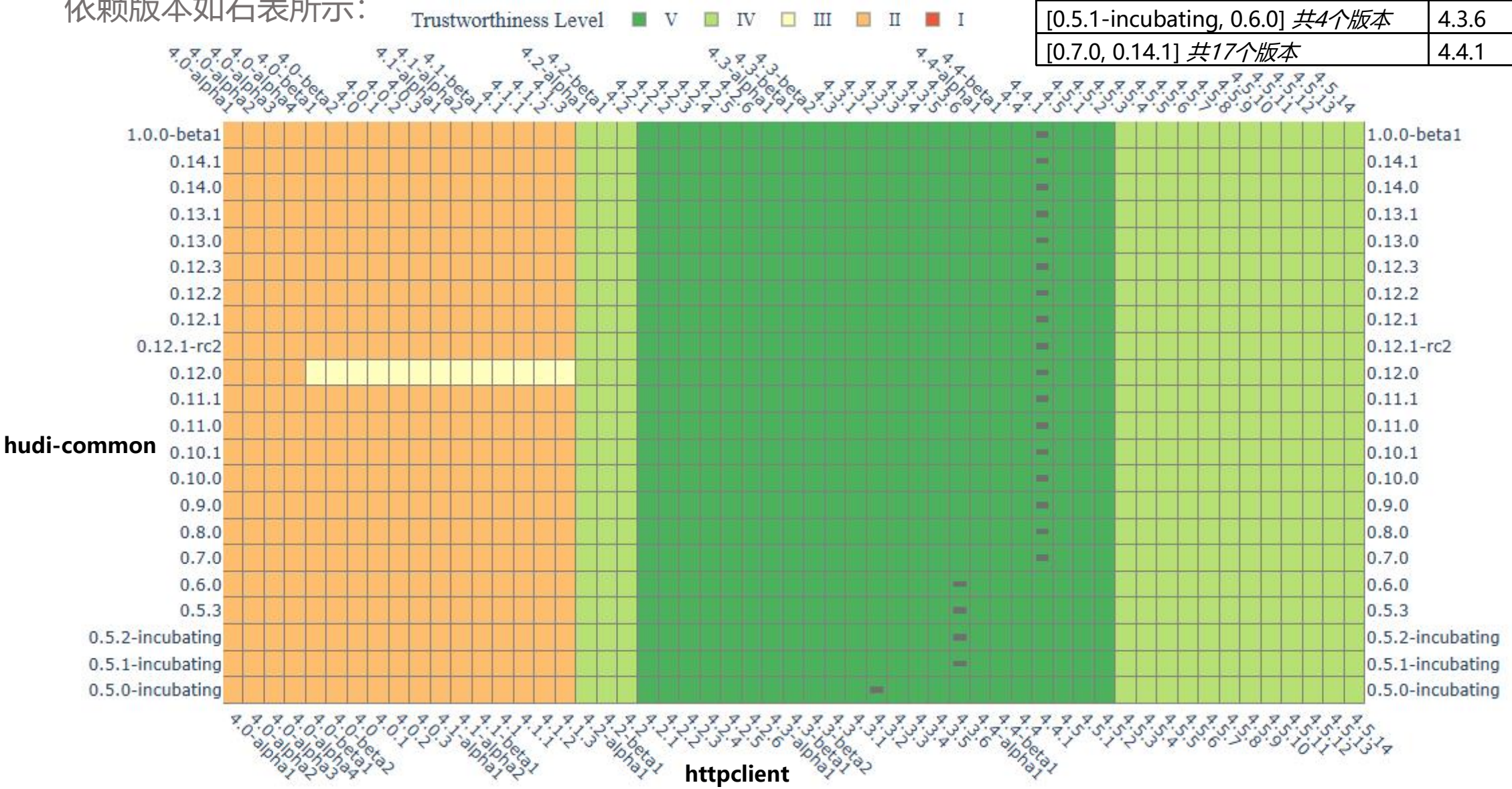
- 我们选取了**fmt**的**5.3.0、6.2.1、7.1.3、8.0.1、8.1.1、9.0.0、9.1.0、10.0.0、10.1.0、10.2.0、10.2.1、11.0.0、11.0.1**共13个版本，检测并度量其与**spdlog 1.14.1**版本的兼容性。

fmt版本	High	Medium	Low	No	可信值	可信等级
5.3.0	16	0	0	6161	4.927	III
6.2.1	16	0	0	6161	4.927	III
7.1.3	16	0	0	6161	4.927	III
8.0.1	16	0	0	6161	4.927	III
8.1.1	16	0	0	6161	4.927	III
9.0.0	16	0	0	6161	4.927	III
9.1.0	16	0	0	6161	4.927	III
10.0.0	1	0	0	6176	7.338	III
10.1.0	1	0	0	6176	7.338	III
10.2.0	1	0	0	6176	7.338	III
10.2.1	0	0	0	0	10.0	V
11.0.0	14	1	1	6161	4.308	III
11.0.1	16	0	0	6161	4.927	III

软件间兼容性-例4

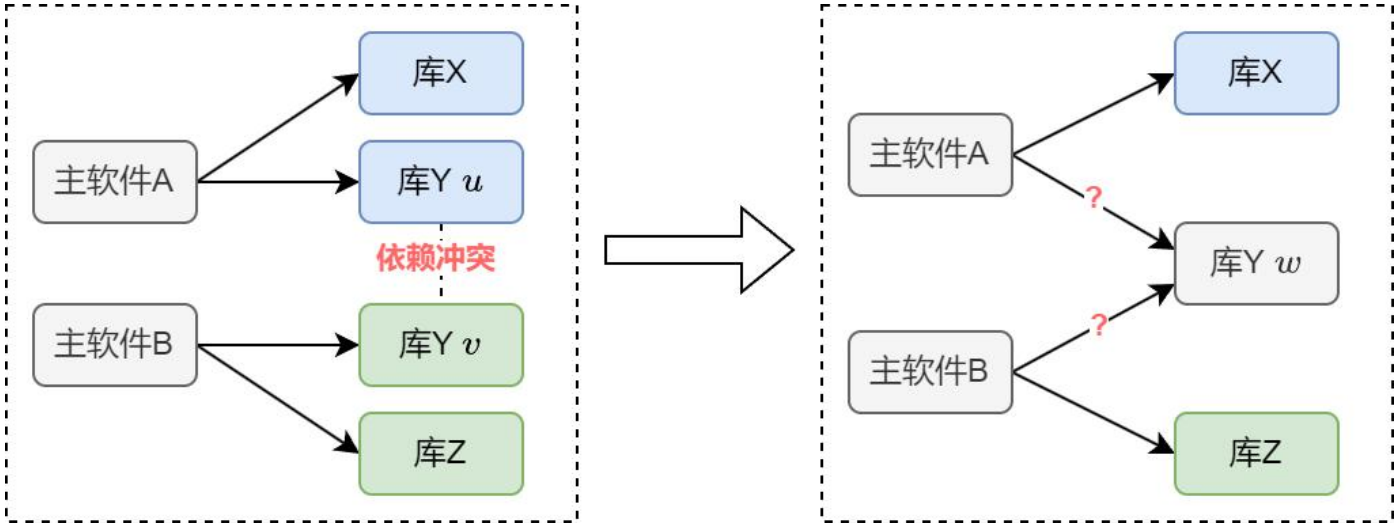
- （Java）主软件**hudi-common**直接依赖于**httpclient**，默认依赖版本如右表所示：

hudi-common版本	默认依赖httpclient版本
0.5.0-incubating	4.3.2
[0.5.1-incubating, 0.6.0] *共4个版本*	4.3.6
[0.7.0, 0.14.1] *共17个版本*	4.4.1



软件间兼容性-两个主软件配套兼容性

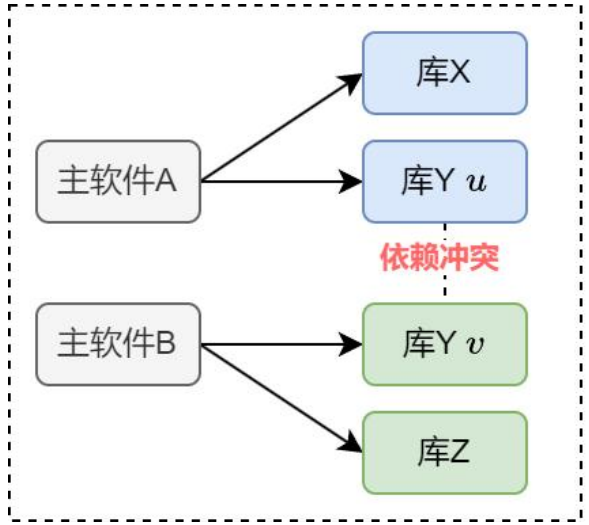
- 当**两个主软件配套使用**时，它们各自会依赖于其他软件，但是当**两个主软件依赖了相同的其他软件时**，若其依赖的版本不同，则可能出现兼容性问题，这也常被成为**依赖冲突问题**。



如上图所示，当主软件A和主软件B配套使用时，会出现**依赖冲突问题**，主软件A依赖于库Y的u版本，而主软件B依赖于库Y的v版本。但是，若能够找到库Y的某个版本w，使得主软件A和主软件B都与其兼容，则可以解决该依赖冲突问题。

软件间兼容性-两个主软件配套兼容性

因此，我们对**主软件A（特定版本）**和**主软件B（特定版本）**配套使用时的兼容性作如下定义：



- 假设**主软件A**和**主软件B**之间有n个依赖冲突问题，发生依赖冲突的软件分别为$L_1, L_2, \cdots, L_n$；
- 对于被依赖的软件L_i，有m_i个版本：$L_i^1, L_i^2, \cdots, L_i^{m_i}$，则可以计算**软件$L_i$和主软件A及主软件B的最高兼容性可信等级**：

$$T_i = \max_{1 \le j \le m_i} \min \left\{ T'\left(A, L_i^j\right), T'\left(B, L_i^j\right) \right\}$$

其中$T'(X, L)$表示**主软件X（特定版本）**与其**直接依赖的软件L（特定版本）**之间的**兼容性可信性等级**，可能的取值为{I, III, III, IV, V}，且I < II < III < IV < V。

则我们定义**主软件A和主软件B之间的兼容性可信等级**为：$T(A,B) = \min_{1 \le i \le n} T_i$

软件间兼容性-例5

我们选取两个主软件，一个是来自Apache Hudi的组件**hudi-common v0.14.1**，另一个是来自Apache Druid的组件**druid-influxdb-emitter v29.0.1**。

hudi-common v0.14.1的直接依赖

直接依赖软件	默认依赖版本
jackson-annotations	**2.10.0**
jackson-databind	**2.10.0**
jackson-datatype-jsr310	2.10.0
jackson-module-afterburner	2.10.0
jackson-module-afterburner	2.10.0
caffeine	2.9.1
disruptor	3.4.2
commons-io	2.11.0
hbase-client	2.4.9
hbase-server	2.4.9
fluent-hc	4.4.1
httpclient	**4.4.1**
orc-core	1.8.3
lz4-java	1.8.0
jol-core	0.16
RoaringBitmap	0.9.47
rocksdbjni	7.5.3

druid-influxdb-emitter v29.0.1的直接依赖

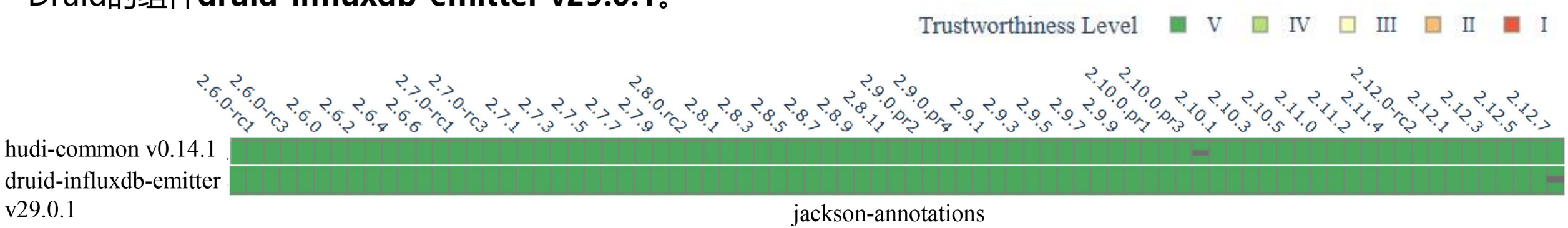
直接依赖软件	默认依赖版本
jackson-annotations	**2.12.7**
jackson-databind	**2.12.7**
guava	32.0.1-jre
guice	4.1.0
joda-time	2.12.5
httpclient	**4.5.13**
httpcore	4.4.11

存在的依赖冲突：

1. **jackson-annotations**
2. **jackson-databind**
3. **httpclient**

软件间兼容性-例5

我们选取两个主软件，一个是来自Apache Hudi的组件**hudi-common v0.14.1**，另一个是来自Apache Druid的组件**druid-influxdb-emitter v29.0.1**。



- **hudi-common v0.14.1**直接依赖于**jackson-annotations**的**2.10.0**版本，而**druid-influxdb-emitter v29.0.1**直接依赖于**jackson-annotations**的**2.12.7**版本

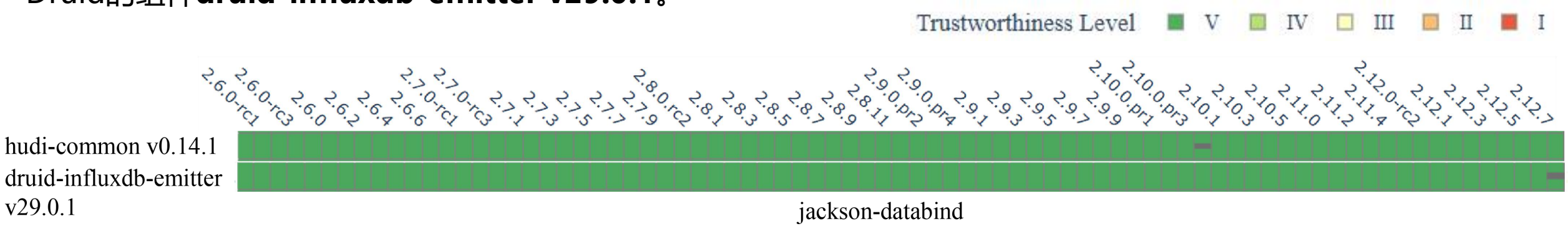
- 以上**jackson-annotations**的任意版本都能与两个主软件达到兼容性可信等级**V级**，故针对该依赖冲突，有$T_1 = \mathrm{V}$

存在的依赖冲突：

1. **jackson-annotations**
2. **jackson-databind**
3. **httpclient**

软件间兼容性-例5

我们选取两个主软件，一个是来自Apache Hudi的组件**hudi-common v0.14.1**，另一个是来自Apache Druid的组件**druid-influxdb-emitter v29.0.1**。



- **hudi-common v0.14.1**直接依赖于**jackson-databind**的**2.10.0**版本，而**druid-influxdb-emitter v29.0.1**直接依赖于**jackson-databind**的**2.12.7**版本

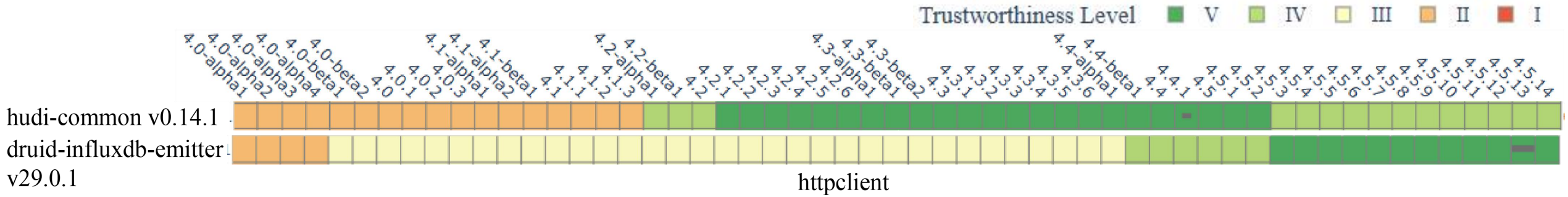
- 以上**jackson-data**的任意版本都能与两个主软件达到兼容性可信等级**V级**，故针对该依赖冲突，有$T_2 = \mathrm{V}$

存在的依赖冲突：

1. jackson-annotations
2. **jackson-databind**
3. httpclient

软件间兼容性-例5

我们选取两个主软件，一个是来自Apache Hudi的组件**hudi-common v0.14.1**，另一个是来自Apache Druid的组件**druid-influxdb-emitter v29.0.1**。



- **hudi-common v0.14.1**直接依赖于**httpclient**的**4.4.1**版本，而**druid-influxdb-emitter v29.0.1**直接依赖于**httpclient**的**4.5.13**版本

- 若**httpclient**选择**[4.4-beta1, 4.5.2]**版本，虽然与**hudi-common**的兼容性可信等级可以达到**V级**，但与**druid-influxdb-emitter**只有**IV级**；而若**httpclient**选择**[4.5.3, 4.5.14]**，虽然与**druid-influxdb-emitter**的兼容性可信等级可以达到**V级**，但与**hudi-common**只有**IV级**。故针对该依赖冲突，有$T_3 = \text{IV}$

存在的依赖冲突：

1. jackson-annotations
2. jackson-databind
3. httpclient

共有三个依赖冲突，通过选择合适版本能达到在最高兼容性可信值为$T_1 = \text{V}$, $T_2 = \text{V}$, $T_3 = \text{IV}$，所以最终**两个主软件间的可信值为**$T = \min\{T_1, T_2, T_3\} = \text{V}$

思考题与作业

1、开源软件的定义以及特点是什么？

2、开源软件API的定义是什么？其兼容性是如何定义的？

3、开源软件兼容性分成版本间兼容性和软件间兼容性，其定义是什么？有何差异？

4、开源软件兼容性问题严重程度分成哪4级？分别表示什么风险？

5、版本间兼容性可信度量公式什么？等级划分表是什么？

5、阐述软件间兼容性测试框架以及面向Java和C++的工具平台框架