XINCHECK 文本查重 SDK

开发者接入文档

V0.5.7

北京芯锋科技有限公司
2021 年 12 月

目录

目录 2
一、快速上手 4
1 使用前准备 4
1.1 引入 SDK 包 4
1.2 SDK 授权 4
2 使用简易启动器快速上手 4
2.1 简易启动器介绍 4
2.2 完整代码示例 5
2.3 总结 5
3 原生方式快速上手 5
3.1 加载比对库 5
3.2 加载待查重的文件 6
3.3 启动查重任务 6
3.4 保存查重报告 6
3.5 完整示例代码 6
3.6 总结 7
4 重要高级接口介绍 8
4.1 实例化文本对象（Paper） 8
4.2 为本文对象补充额外信息 8
4.3 为本文对象设置 Payload 9
4.4 启动查重任务 9
4.5 异步处理查重任务 9
4.6 自定义查重报告的样式 10
5 原生方式快速上手 Demo（进阶） 11
二、详细文档 12
1 关键类 12
2 接口文档 12
2.1 SDK 相关 12
2.1.1 获取机器指纹 12
2.1.2 设置许可证 12
2.1.3 获取 SDK 的授权状态 12
2.2 简易启动器 13
2.3 文本对象（Paper）相关 13
2.3.1 Paper 对象实例化 13
2.3.2 文本对象的额外文本信息 14
2.3.3 为本文对象设置 Payload 14
2.3.4 文本对象的紧凑等级 14
2.3.5 获取 Paper 的前言、正文、参考文献等各部分 15
2.3.6 获取 Paper 对象的字符数 15
2.3.7 序列化/反序列化 Paper 对象 15
2.3.8 克隆 Paper 对象 16
2.3.9 replace 方法 16
2.4 比对库相关 16
2.4.1 向比对库中添加 Paper 16
2.4.2 获取比对库中的 Paper 17

2.4.3 清空比对库中的 Paper 17
2.4.4 序列化/反序列化比对库 17
2.5 文本段（PaperSeg）相关 17
2.5.1 PaperSeg 的序列化 17
2.6 自定义查重报告样式 18
2.6.1 设置标题 18
2.6.2 设置副标题 18
2.6.3 设置头部提示信息 18
2.6.4 设置尾部提示信息 18
2.6.5 设置左上角 logo 19
2.6.6 设置右上角 logo 19
2.6.7 设置主背景色 19
2.6.8 设置分割线颜色 19
2.6.9 设置水印 19
2.7 查重任务相关 19
2.7.1 构建查重任务 19
2.7.2 启动查重任务 21
2.7.3 停止查重任务 22
2.7.4 关闭查重任务线程池 22
2.8 查重任务观察者接口（CheckState） 22
2.8.1 泛型 T 22
2.8.2 taskStart 接口 22
2.8.3 taskFinish 接口 23
2.8.4 paperStart 接口 23
2.8.5 paperSuccess 接口 23
2.8.6 paperFailed 接口 23
2.8.7 完整实现示例 24
2.9 查重报告相关 25
2.9.1 将查重报告保存为文件 25
2.9.2 获取查重报告 id 25
2.9.3 设置查重报告 id 25
2.9.4 获取待查 Paper 对象 26
2.9.5 获取总重复字符数 26
2.9.6 获取总文字复制比 26
2.9.7 获取前部重复字符数 26
2.9.8 获取前部复制比 26
2.9.9 获取后部重复字符数 26
2.9.10 获取后部复制比 26
2.9.11 获取单篇最大重复字符数 27
2.9.12 获取单篇最大文字复制比 27
2.9.13 获取总段落数 27
2.9.14 获取疑似段落数 27
2.9.15 获取疑似段落最大重合字符数 27
2.9.16 获取疑似段落最大复制比 27
2.9.17 获取疑似段落最小重合字符数 28
2.9.18 获取疑似段落最小复制比 28
2.9.19 获取疑似段落单篇最大重合字符数 28
2.9.20 获取疑似段落单篇最大复制比 28
2.9.21 获取全文重复文献重复率列表 28

2.9.22 将 Reporter 列表保存为 csv 文件 28
2.9.23 获取查重结果（CheckResult）对象 29
2.10 查重结果对象（CheckResult）相关 29
2.10.1 获取 uid 29
2.10.2 获取 sectionId 29
2.10.3 获取所对应的文本分段的原文 29
2.10.4 获取所对应的文本分段的重复字符数 29
2.10.5 获取 copySegMap 29
2.10.6 获取 copyDetailMap 29
2.11 高级配置项 30
3 算法介绍 32
附录 1：修改 JDK 加密策略文件 33
附录 2：查重报告解读及指标说明 34

一、快速上手

1 使用前准备

1.1 引入 SDK 包

通过 maven 将本 SDK 引入到项目中。XINCHECK SDK 建议您使用 1.8.0_151 及以上版本的 JDK。如 JDK 低于此版本，您需要修改 JDK 加密策略文件（详见附录 1）或访问 Oracle 官网（https://www.oracle.com/java/technologies/javase-downloads.html）升级 JDK 至 1.8.0_151 及以上。

SDK 存放在私有 maven 仓库中，需要先在<repositories>中添加以下仓库

```
<repository>
    <id>XINCHECK</id>
    <name> XINCHECK Public Repository</name>
    <url>https://maven.xincheck.com/repository/pro/</url>
</repository>
```

然后在<dependencies>中添加依赖

```
<dependency>
    <groupId>com.xincheck</groupId>
    <artifactId>duplicate-check-pro</artifactId>
    <version>0.5.7</version>
</dependency>
```

1.2 SDK 授权

我们为非商业用户提供免费的评估许可证，详细获取及试用说明请访问许可证的获取与使用。您亦可购买商业版许可证以使用更丰富的功能。

```
CheckManager.INSTANCE.setRegCode("许可证");
```

获取到许可证后，通过 setRegCode 方法对 SDK 进行授权，即可解锁相应版本的功能。SDK 授权一次即可在程序运行周期内一直生效，无需在每次调用前反复授权。授权必须在 SDK 的其它调用之前进行，授权之前的调用将被视为无授权调用，在功能上受到限制。

2 使用简易启动器快速上手

2.1 简易启动器介绍

为了便于部分简单应用场景下的开发，SDK 0.5.0 以上版本内置了简易启动器 EasyStarter，通过如下所示的一行代码即可完成 SDK 调用。调用示例如下

```
List<Reporter> reporters = EasyStarter.check(new File("参数 1"), new File("参数 2"), "参数 3", "参数 4");
```

参数 1：待查文件所在的文件夹路径（如果待查文件只有一个，可以传文件路径）；
参数 2：比对库文件所在的文件夹路径（如果比对库中只有一个文件，可以传文件路径）；
参数 3：保存查重报告的文件夹路径。如果不需要导出查重报告可以传空字符串；
参数 4：白名单文本。对于标书查重等场景，有一些文本是允许重复的，这些文本可以通过该参数传入。该参数可选，如不需要可以不传或传 null。

横向查重应用场景下参数 1 和参数 2 可以相同，相同的文件会自动跳过比对，不会出现重复率 100%的问题。除示例方法外，该方法还有多个重载，如需详细了解，请阅读详细文档中的“简易启动器”部分。

2.2 完整代码示例

```
import cn.textcheck.CheckManager;
import cn.textcheck.starter.EasyStarter;

/**
 * SDK 简易启动器示例
 */
public class Main {
    public static void main(String[] args) throws Exception {
        //获取机器指纹
        System.out.println(CheckManager.INSTANCE.getMachineCode());
        //设置许可证（获取：https://xincheck.com/?id=7）许可证只需要设置 1 次
        CheckManager.INSTANCE.setRegCode("muQyymFW0ysAZZhKVOzkh/jb2Fq9xEZxfIA=");
        //简易启动查重任务
        EasyStarter.check(new File("待查文件夹路径"), new File("比对库文件夹路径"), "查重报告文件夹路径");
    }
}
```

也可参见 GitHub 链接中的 EasyStart 部分。
Github 链接：https://github.com/tianlian0/duplicate-check-sample

2.3 总结

EasyStarter 只适用于场景简单、数据量少的应用场景，其中大量参数使用了默认值。实际项目开发过程中，对于一些复杂的应用场景，还请阅读“原生方式快速上手”、“重要高级接口介绍” 、“原生方式快速上手 Demo（进阶）”了解详细的参数设置，通过使用 CheckState 观察者、序列化加载等方法更大限度的发挥 SDK 的高性能和高灵活性。

3 原生方式快速上手

3.1 加载比对库

将所有要作为比对库的文件放到一个文件夹中（支持 doc、docx、xls、xlsx、pdf、rtf、txt 格式），实例化 PaperLibrary 时将文件夹路径传入构造方法。

```
LocalPaperLibrary paperLibrary = LocalPaperLibrary.load(new File("对比库文件夹路径"));
```

注：加载比对库会花费一定时间，通常只需要在服务启动的时候加载一次，后续的所有查重操作都不需要再重新加载比对库。

3.2 加载待查重的文件

```
Paper paper = Paper.load(new File("文件路径")); //加载单个文件
```

将待查重的文件加载为 Paper。Paper 支持多种加载方式，可以通过文件、字符串、输入流加载，也支持批量加载。本例中以通过文件加载为例，其它用法请阅读“重要高级接口介绍”。

3.3 启动查重任务

```
//构建并启动任务
CheckTask checkTask = CheckManager.INSTANCE
        .getCheckTaskBuilder() //获取查重任务构建器
        .addLibrary(paperLibrary) //添加比对库。可以添加多个
        .addCheckPaper(paper) //添加待查文本。可以添加多个
        .build(); //构建任务，返回 checkTask 对象
checkTask.start(); //启动任务线程
checkTask.join(); //等待查重结束（阻塞）
```

该方式为同步启动查重任务，并通过 join 方法等待查重任务结束。SDK 支持异步调用，

注意：待查文本和比对库中的文本如果完全相同，将会自动跳过，不进行查重比对。测试时请不要使用完全相同的两个文本进行查重。

3.4 保存查重报告

查重任务结束后可以通过以下方式将查重报告保存为 html 文件，SDK 可以生成两种查重报告。

```
Reporter reporter = checkTask.getReporters().get(0);
reporter.saveAsFile("html 文件保存路径", ReportType.TEXT_WITH_CITATION);
reporter.saveAsFile("html 文件保存路径", ReportType.TEXT_WITH_ORIGINAL);
```

getReporters 方法返回一个查重任务中所有待查文本的查重结果列表。本例中只添加了一个待查文本，因此直接通过 get(0)获取到对应的查重结果。

当然，您可以不将结果保存为文件，直接通过接口获取到查重结果。如：

```
String reportId = reporter.getReportId(); //获取查重报告 id
String copyRate = reporter.getCopyRate(); //获取总重复率
String copyWords = reporter.getCopyWords(); //获取重复字数
```

查重报告的样式可以通过接口进行一定程度的自定义，详请阅读本文档的“详细文档-接口文档-自定义查重报告样式”部分；同时，查重报告的所有内容都可以在代码中以接口的方式获取到，详请阅读本文档的“详细文档-接口文档-获取查重报告的元数据”部分。

3.5 完整示例代码

```
import cn.textcheck.CheckManager;
```

```java
import cn.textcheck.engine.checker.CheckTask;
import cn.textcheck.engine.pojo.LocalPaperLibrary;
import cn.textcheck.engine.pojo.Paper;
import cn.textcheck.engine.type.ReportType;

import java.io.File;

/**
 * SDK 同步调用使用范例
 */
public class Main1 {
    public static void main(String[] args) throws Exception {
        //获取机器指纹
        System.out.println(CheckManager.INSTANCE.getMachineCode());
        //设置许可证（免费获取评估许可证：https://xincheck.com/?id=7）
        CheckManager.INSTANCE.setRegCode("muQyymFW0ysAZZhKVOzkh/jbT2Fq9xEZxfIA=");

        //通过<文件夹>加载本地比对库（支持 pdf、txt、doc、docx）
        LocalPaperLibrary paperLibrary = LocalPaperLibrary.load(new
File("C:\\Desktop\\Library"));//初始化对比库对象。路径为比对库所在<文件夹>

        //读取待查重的<文件>（支持 pdf、txt、doc、docx）
        Paper paper = Paper.load(new File("C:\\Desktop\\test.docx")); //读取本地<文件>

        //注意：待查文本和比对库中的文本如果完全相同，将会自动跳过，不进行查重比对

        //构建并启动任务
        CheckTask checkTask = CheckManager.INSTANCE
                .getCheckTaskBuilder() //获取查重任务构造器
                .addLibrary(paperLibrary) //添加比对库。可以添加多个
                .addCheckPaper(paper) //添加待查 Paper。可以添加多个
                .build(); //构建任务，返回 checkTask 对象
        checkTask.start(); //启动任务
        checkTask.join(); //等待查重结束（阻塞）

        //保存查重报告
        checkTask.getReporters().get(0).saveAsFile("C:\\Report\\report1.html",
ReportType.TEXT_WITH_CITATION); //保存全文标明引文查重报告
        checkTask.getReporters().get(0).saveAsFile("C:\\Report\\report2.html",
ReportType.TEXT_WITH_ORIGINAL); //保存原文对照查重报告
    }
}
```

也可参见 GitHub 链接中的 Sample1 部分。

Github 链接：https://github.com/tianlian0/duplicate-check-sample

3.6 总结

本节展示了 SDK 同步调用使用方法，让您较为简洁的看到 SDK 的使用流程和效果。实际项目开发过程中，还请

阅读“原生方式快速上手 Demo（进阶）”、“重要高级接口介绍”了解详细的参数设置，通过使用 CheckState 观察者、序列化加载等方法更大限度的发挥 SDK 的高性能和高灵活性。

4 重要高级接口介绍

4.1 实例化文本对象（Paper）

在 SDK 中，无论是待查重的文件还是用于充当比对库的文件，都需要实例化为 Paper 对象后再进行后续的处理。本地比对库 LocalPaperLibrary 可以看作是一个包含 Paper 对象的集合。Paper 对象可以通过 File 对象实例化，也可以通过文本内容实例化。

①通过 File 对象实例化

```
Paper paper = Paper.load(new File("文件路径"));
```

这种加载方式需要传入一个 File 对象，传入后将根据 File 对象中文件路径的拓展名识别文件类型，并进行加载。SDK 只支持 doc、docx、xls、xlsx、pdf、rtf、txt 格式的文件，其中 txt 格式的加载效率最高，pdf 格式的加载效率最低。xls、xlsx 格式在加载时会将每个单元格之间添加空格进行拼接，得到最终的文本。

②通过 File 对象实例化，并指定文件类型

```
Paper paper = Paper.load(new File("文件路径"), FileType.TXT);
```

这种加载方式需要传入两个参数，包含一个 File 对象以及文件类型 FileType。此时将根据传入的文件类型进行加载。这种加载方式适合 File 对象中不包含文件路径，或文件路径中的文件名不包含拓展名的情况。第二个参数 FileType 是枚举类型。

③通过文本实例化

```
Paper paper = Paper.load("文本内容");
```

通过文件的文本内容加载 Paper 对象。如对用户输入的文本进行查重，用户的输入只是一串文本，这种情况下可以直接用文本内容构建 Paper 对象进行查重。

④将文件夹中的文件批量加载为 List<Paper>

```
List<Paper> papers = Paper.load(new File("文件夹路径").listFiles());
```

4.2 为本文对象补充额外信息

对于每一个 Paper 对象，包含 id、标题、作者、其它信息四项可选信息，这四项信息的设置不会影响查重结果。id 可由开发者按需标记、使用，用以唯一标识一个 Paper 对象，不会在查重报告中展示；其余三项项信息设置后会在查重报告中进行展示。如果不设置这些信息，将默认使用文件名作为标题，其它信息为空。有以下几种方式可以对这四项信息进行设置。

①通过 set 方法直接设置

```
paper.setId("001").setTitle("标题").setAuthor("作者").setInfo("其它需要展示的信息");
```

可以通过 set 方法连续设置 id、标题、作者等信息。

②通过格式化文件名设置

通过 File 对象加载 Paper 对象时，如果文件名中包含分隔符“@”且数量符合以下任一种规则，将自动从文件名中读取标题、作者、来源和年份信息。

1）如果文件名中包含 1 个分隔符，sdk 会将文件名按分隔符分割为数组后依次读取为 id、标题，如文件名为“001@标题.docx”；

2）如果文件名中包含 2 个分隔符，sdk 会将文件名按分隔符分割为数组后依次读取为 id、标题、作者，如文

件名为“001@标题@作者.docx”；

3）如果文件名中包含 3 个分隔符，sdk 会将文件名按分隔符分割为数组后依次读取为 id、标题、作者、其它信息，如文件名为“001@标题@作者@其它需要展示的信息.docx”；

4）如果文件名中不包含分隔符，sdk 会将文件名读取为标题，如文件名为“标题.docx”。

另：“@”只是 SDK 的默认分割符，分隔符可由开发者自行修改，修改方式参见“详细文档-接口文档-高级配置项”部分。

4.3 为本文对象设置 Payload

Payload 可用于传递上下文信息，也可以用于为 Paper 补充不希望在查重报告中展示的信息。对于每一个 Paper 对象，可以设置一个 Payload，Payload 不会展示到查重报告中，但可以用来存储额外信息或上下文信息。Payload 可以是任意对象，但该对象必须实现了 Serializable 接口。

```
paper.setPayload("001");
Object payload = paper.getPayload();
```

设置 payload 后，可以通过 get 方法获取。

4.4 启动查重任务

启动查重任务时可以做一些额外的配置，以提高灵活性。示例代码如下

```
CheckManager.INSTANCE
        .getCheckTaskBuilder() //获取查重任务构造器
        .setUid("1") //设置任务 id
        .addCheckState(new CheckStateImp(), "自定义上下文信息") //注册观察者和上下文信息
        .addLibrary(paperLibrary) //设置比对库
        .addCheckPaper(paper) //设置待查 Paper
        .setTemplate(new HtmlTemplate()) //设置查重报告模板。如不需要修改样式不用设置
        .build() //构建任务
        .submit(); //启动任务。submit：将任务提交到线程池中。start：直接启动任务
```

4.5 异步处理查重任务

通常情况下根据比对库的大小、待查文本的字数、计算机 CPU 繁忙程度等因素的不同，需要数秒至数分钟才能完成一次查重任务。因此查重任务的异步处理就显得尤为重要。通过实现 CheckState 接口，可以实现查重任务的异步处理。

CheckState 接口中包含 taskStart、taskFinish、paperStart、paperFinish、paperFailed 五个方法，分别对应查重任务提交后的启动、完成，以及单个待查文本的启动、查重成功、查重失败。泛型 T 对应查重任务提交时传递的上下文信息的数据类型。方法及参数的详细注释如下

```
public interface CheckState<T> {
    /**
     * 查重任务启动前会回调此方法
     * @param uid         任务唯一 id
     * @param checkPapers 待查的文本对象
     * @param context     上下文信息
     */
    void taskStart(String uid, List<Paper> checkPapers, T context);
```

```
    /**
     * 查重任务中的全部文本均查重完毕后回调此方法
     * @param uid          任务唯一 id
     * @param checkPapers  待查的文本对象
     * @param reporters    查重报告（只有查重成功的文本才会有 reporter）
     * @param failedPapers 查重失败的文本对象
     * @param context      上下文信息
     */
    void taskFinish(String uid, List<Paper> checkPapers, List<Reporter> reporters,
List<Paper> failedPapers, T context);

    /**
     * 查重任务中的某一篇文本开始查重前时会调此方法
     * @param uid      任务唯一 id
     * @param paper    被查 Paper
     * @param context  上下文信息
     */
    void paperStart(String uid, Paper paper, T context);

    /**
     * 查重任务中的某一篇文本查重成功后会调用此方法
     * @param uid      任务唯一 id
     * @param reporter 查重报告
     * @param context  上下文信息
     */
    void paperSuccess(String uid, Reporter reporter, T context);

    /**
     * 查重任务中的某一篇文本查重失败后会调用此方法
     * @param uid      任务唯一 id
     * @param paper    失败的 Paper
     * @param code     错误码
     * @param t        错误信息
     * @param context  上下文信息
     */
    void paperFailed(String uid, Paper paper, int code, Throwable t, T context);
}
```

实现该接口并在查重任务启动时通过如 addCheckState 方法将 CheckState 的实现类以及上下文信息传入，当查重任务处于对应的状态时回调对应的方法。

4.6 自定义查重报告的样式

查重报告的标题、logo、颜色等可以进行部分自定义，只需实例化查重报告模板 HtmlTemplate，使用 HtmlTemplate 中的相关方法对样式进行调整，最后在提交查重任务时使用 setTemplate 方法将调整后的模板注入即可。下面提供几个修改样式的代码示例，更多内容请参考“详细文档-接口文档-自定义查重报告样式”部分。

```
HtmlTemplate template = new HtmlTemplate();
```

```
template.setTitle("查重报告标题"); //自定义查重报告的标题
template.setLeftHeadImage(ImageIO.read(new File("D:/logo.jpg"))); //设置左上角的 logo
template.setBackgroundColor("#f3f9ef"); //设置查重报告的背景色
```

5 原生方式快速上手 Demo（进阶）

完整示例代码参见 GitHub 链接中的 Sample2、Sample3、Sample4 部分。
Github 链接：https://github.com/tianlian0/duplicate-check-sample

二、详细文档

1 关键类

CheckManager：SDK 管理器。枚举实现的单例，无需实例化内部方法均可直接调用。包含获取机器指纹、SDK 授权、启动查重任务等能力。
CheckState：查重任务观察者接口。实现该接口并注册到查重任务中，查重任务会在各个状态会回调对应方法。
Paper：文本对象。所有文本（无论是待查文本还是要充当比对库的文本）都需要加载为 Paper 对象后才能仅需后续的处理。
PaperSeg：文本段。代表 Paper 对象中的某一句或一段文本。
LocalPaperLibrary：本地比对库。内部为 Paper 和 PaperSeg 两个集合。build 比对库的过程即为将 Paper 分割为 PaperSeg 并进行预处理的过程。
CheckTask：查重任务。内部包含了待查文本、比对库、查重算法、查重报告模板、查重任务观察者等多个查重必须的对象。
HtmlTemplate：默认查重报告模板。该模板实现了查重报告的特定样式，并提供了部分接口用以自定义部分查重报告的样式。
CheckResult：查重结果对象。查重任务结束后会返回多个 CheckResult 对象，每个 CheckResult 对象包含一部分查重结果的元数据。
Reporter：默认查重报告器。查重报告器会将多个 CheckResult 提供的元数据进行组装整合，得到完整的查重数据，并使用 HtmlTemplate 模板生成完整的查重报告文件。

2 接口文档

2.1 SDK 相关

2.1.1 获取机器指纹

```
CheckManager.INSTANCE.getMachineCode();
```

返回一个 String。机器指纹由多个终端标识散列脱敏处理后得到，唯一标识一个终端。当终端的硬件、操作系统、JVM 版本、MAC 地址、网络情况发生变化时，机器指纹将随之变化，此时绑定到原有终端的许可证将失效，您需要重新购买许可证。

2.1.2 设置许可证

```
CheckManager.INSTANCE.setRegCode("许可证");
```

如果许可证无效，将会抛出 RegCodeInvalidException 运行时异常，否则不会抛出异常。

2.1.3 获取 SDK 的授权状态

```
CheckManager.INSTANCE.regState();
```

返回一个布朗型，代表 SDK 当前的授权状态。true 代表 SDK 许可证有效，功能可以正常使用；false 代表 SDK 未成功授权，部分功能受限。

2.2 简易启动器

为了方便一些简单应用场景下的开发，SDK 内置了简易启动器 EasyStarter，使用一行代码即可完成一些简单应用场景的 SDK 调用。

简易启动器只适用于小规模数据（100 万字符以下）、无需异步执行、无需特殊配置项、无定制开发需求的特定应用场景，或用于开发人员进行 Demo 开发和试验。其它应用场景请务必试用原生方式进行开发。

EasyStarter.check 方法提供 8 个重载。

```
List<Reporter> reporters = EasyStarter.check(new File("参数 1"), new File("参数 2"), "参数 3", "参数 4");
```

参数 1：待查文本。可以为文件路径、文件夹路径、字符串数组。如果为文件路径，则该文件将被作为待查文本；如果为文件夹路径，则该文件夹中的所有文件将被作为待查文本；如果为字符串数组，则该数组中的每个字符串将被作为一个独立的待查文本。

参数 2：比对库。可以为文件路径、文件夹路径、字符串数组。如果为文件路径，则该文件将被作为比对库；如果为文件夹路径，则该文件夹中的所有文件将被作为比对库；如果为字符串数组，则该数组中的每个字符串将被作为比对库中的一个独立的文本。

参数 3：查重报告保存的文件夹。必须为一个文件夹路径。查重报告以及查查重统计表将被输出到该文件夹。如果不需要将查重报告输出到文件，可以传空字符串。

参数 4：白名单字符串。该参数可选，如不需要可以不传或传 null。对于标书查重等场景，有一些文本是允许重复的，这些文本可以拼接成一个字符串后通过该参数传入。

返回值 List<Reporter>：查重结果。List 中的 Reporter 按重复率从高到低排序。

2.3 文本对象（Paper）相关

在 SDK 中，无论是待查重的文件还是用于充当比对库的文件，都需要实例化为 Paper 对象后再进行后续的处理。

2.3.1 Paper 对象实例化

Paper 对象既可以通过 File 对象实例化，也可以通过文本内容实例化。

①通过 File 对象实例化

```
Paper paper = Paper.load(new File("文件路径"));
```

这种加载方式需要传入一个 File 对象，传入后将根据 File 对象中文件路径的拓展名识别文件类型，并进行加载。SDK 只支持 doc、docx、xls、xlsx、pdf、rtf、txt 格式的文件，其中 txt 格式的加载效率最高，pdf 格式的加载效率最低。xls、xlsx 格式在加载时会将每个单元格之间添加空格进行拼接，得到最终的文本。

②通过 File 对象实例化，并指定文件类型

```
Paper paper = Paper.load(new File("文件路径"), FileType.TXT);
```

这种加载方式需要传入两个参数，包含一个 File 对象以及文件类型 FileType。此时将根据传入的文件类型进行加载。这种加载方式适合 File 对象中不包含文件路径，或文件路径中的文件名不包含拓展名的情况。第二个参数 FileType 是枚举类型。

③通过文本实例化

```
Paper paper = Paper.load("文本内容");
```

通过文件的文本内容加载 Paper 对象。如对用户输入的文本进行查重，用户的输入只是一串文本，这种情况下可以直接用文本内容构建 Paper 对象进行查重。

④将文件夹中的文件批量加载为 List<Paper>

```
List<Paper> papers = Paper.load(new File("文件夹路径").listFiles());
```

2.3.2 文本对象的额外文本信息

对于每一个 Paper 对象，包含 id、标题、作者、其它信息四项可选信息，这四项信息的设置不会影响查重结果。id 可由开发者按需标记、使用，用以唯一标识一个 Paper 对象，不会在查重报告中展示；其余三项项信息设置后会在查重报告中进行展示。如果不设置这些信息，将默认使用文件名作为标题，其它信息为空。有以下几种方式可以对这四项信息进行设置。

①通过 set 方法直接设置

```
paper.setId("001").setTitle("标题").setAuthor("作者").setInfo("其它需要展示的信息");
```

可以通过 set 方法连续设置 id、标题、作者等信息。

②通过格式化文件名设置

通过 File 对象加载 Paper 对象时，如果文件名中包含分隔符“@”且数量符合以下任一种规则，将自动从文件名中读取标题、作者、来源和年份信息。

1）如果文件名中包含 1 个分隔符，sdk 会将文件名按分隔符分割为数组后依次读取为 id、标题，如文件名为“001@标题.docx”；

2）如果文件名中包含 2 个分隔符，sdk 会将文件名按分隔符分割为数组后依次读取为 id、标题、作者，如文件名为“001@标题@作者.docx”；

3）如果文件名中包含 3 个分隔符，sdk 会将文件名按分隔符分割为数组后依次读取为 id、标题、作者、其它信息，如文件名为“001@标题@作者@其它需要展示的信息.docx”；

4）如果文件名中不包含分隔符，sdk 会将文件名读取为标题，如文件名为“标题.docx”。

另：“@”只是 SDK 的默认分割符，分隔符可由开发者自行修改，修改方式参见“详细文档-接口文档-高级配置项”部分。

2.3.3 为本文对象设置 Payload

对于每一个 Paper 对象，可以设置一个 Payload，Payload 不会展示到查重报告中，但可以用来存储额外信息或上下文信息。Payload 可以是任意对象，但该对象必须实现了 Serializable 接口。

```
paper.setPayload("001");
Object payload = paper.getPayload();
```

设置 payload 后，可以通过 get 方法获取。

2.3.4 文本对象的紧凑等级

```
paper.setCompactLevel(1); //设置文本紧凑等级
paper.getCompactLevel(); //获取当前 Paper 的文本紧凑等级
```

文本紧凑等级共有 0~4 共五个等级，默认为 0 级。各等级含义如下

0 级：一篇文本将被划分为引言目录、正文、参考文献 3 部分。其中正文部分每 1 万字左右将会再次分块。

1 级：一篇文本将被划分为引言目录、正文、参考文献 3 部分。正文部分不会再进行分段。

2 级：一篇文本将被划分为正文（包含引言目录）、参考文献 2 部分。

3 级：一篇文本将被完整记录为 1 个部分，不会被划分为多个部分。

4 级：Paper 对象将不包含正文，只包含 id、标题、作者、来源、年份等基础信息。

文本紧凑等级主要会影响以下两个方面

1）查重子任务的调度：对每个 Paper 进行查重时，在进行查重时将按文本的分块生成多个子任务使用子任务线程池进行并发查重，每个子任务将会返回一个 CheckResult 对象用于查重报告的生成。因此可以粗略地认为，更低的紧凑等级可以获得更高的查重的并发性，有利于提高查重任务的执行速度。

2）查重报告的分段：由于每个文本分块会由一个子任务执行并生成一个 CheckResult，最终由 Reporter 生成的查重报告页将按照文本的分块进行展示。

文本等级一般保持 0 级即可不需要调整，在调整文本紧凑等级时，有以下几个注意点

1）紧凑等级只能由低到高调整，由高到低的调整将不被执行。

2）3 级在查重时将无法忽略参考文献；4 级不包含文本正文一般用于特殊用途。

3）0 级正文部分默认会按 1 万字进行再分块，1 万字只是默认值，具体分块的字数可以由开发人员自行设置，具体参考“高级配置项”。

2.3.5 获取 Paper 的前言、正文、参考文献等各部分

```
paper.getHeadText(); //获取前言目录
paper.getMidText(); //获取正文
paper.getEndText(); //获取参考文献
```

以上三个方法分别用于获取前言、正文、参考文献，返回 String。如果紧凑等级为 2 前言目录将包含在正文中无法单独获取；如果紧凑等级为 3 前言目录和参考文献都将无法单独获取。

获取 Paper 对象的全文（前言目录+正文+参考文献）。

2.3.6 获取 Paper 对象的字符数

```
paper.length(); //获取字符数
```

获取前言目录+正文+参考文献的字符数。

2.3.7 序列化/反序列化 Paper 对象

```
//序列化 Paper 对象至文件
ObjectOutputStream fos = new ObjectOutputStream(new FileOutputStream("文件路径"));
fos.writeObject(paper);
fos.close();
//从文件反序列化 Paper 对象
ObjectInputStream ois =
        new ObjectInputStream(new FileInputStream(new File("文件路径")));
Paper paper2 = (Paper)ois.readObject();
ois.close();
```

Paper 类实现了 Serializable 接口，支持 JDK 原生的序列化与反序列化方式，上述代码为将 paper 对象序列化至文件再从文件反序列化为 paper 对象的示例。使用序列化文件加载 Paper 对象比使用原始文件加载 Paper 的性能高约 30%。

2.3.8 克隆 Paper 对象

```
Paper newPaper = paper.clone();
```

Paper 类实现了 Cloneable 接口，上述代码为将 paper 对象拷贝至 newPaper 对象的示例。

2.3.9 replace 方法

```
paper.replace("原字符串", "替换字符串");
```

Paper 的 replace 方法可以将 Paper 对象中文本所包含的原字符串替换为指定字符串，需要注意的是，替换字符串中不能包含“<”和“>”，因为这两个字符是 Paper 对象的保留字符。

2.4 比对库相关

本地比对库 LocalPaperLibrary 可以看作是一个包含 Paper 对象的集合。

2.4.1 向比对库中添加 Paper

有多种方式可以向 LocalPaperLibrary 中添加 Paper。文本被添加到比对库后不可删除，如果需要删除某个文件只能重新实例化比对库重新添加。

①实例化时传入 List<Paper>

```
List<Paper> papers = new ArrayList<>();
papers.add(paper1);
papers.add(paper2);
LocalPaperLibrary paperLibrary = LocalPaperLibrary.load(papers);
```

LocalPaperLibrary 会遍历 papers，并逐个添加到比对库中。

②实例化时传入文件夹路径

```
LocalPaperLibrary paperLibrary1 = LocalPaperLibrary.load(new File("文件夹路径"));
LocalPaperLibrary paperLibrary2 = LocalPaperLibrary.load(new File("文件夹路径"), false);
LocalPaperLibrary paperLibrary3 = LocalPaperLibrary.load(new File("文件夹路径"), "*.docx",
false);
```

将要充当比对库的文件放到一个文件夹中（支持 doc、docx、xls、xlsx、pdf、rtf、txt 格式），实例化 PaperLibrary 时将文件夹路径传入构造方法。如果文件夹中存在 Paper 序列化后的文件，同样可以加载，且加载速度更快。xls、xlsx 格式在加载时会将每个单元格之间添加空格进行拼接，得到最终的文本。

上述代码，第一行的加载方式默认会遍历文件夹中所有的子文件夹，如果不想添加子文件夹中的文件可以采取第二行代码的方式。第三行的加载方式中，第二个参数为正则表达式的 pattern，将与文件的绝对路径进行匹配。

③实例化后逐个添加 Paper

```
paperLibrary.addByPaper(paper);
```

④实例化后通过文件夹路径添加

```
paperLibrary.addByFolderPath(new File("文件夹路径"), false);
```

第一个参数为文件夹，第二个参数为是否递归遍历子文件夹中的文件。

2.4.2 获取比对库中的 Paper

```
Map<String, Paper> papers = paperLibrary.getPapers();
```

获取当前比对库中所有的 Paper 对象。返回值为 Map，其中 key 为自动生成的文件 id，value 为 Paper 对象。

2.4.3 清空比对库中的 Paper

```
paperLibrary.clear();
```

清空当前比对库中所有的 Paper 对象。注：该方法线程不安全，通常不建议使用。

2.4.4 序列化/反序列化比对库

```
//序列化 LocalPaperLibrary 对象至文件
ObjectOutputStream fos = new ObjectOutputStream(new FileOutputStream("文件路径"));
fos.writeObject(paperLibrary);
fos.close();
//从文件反序列化 LocalPaperLibrary 对象
ObjectInputStream ois =
        new ObjectInputStream(new FileInputStream(new File("文件路径")));
LocalPaperLibrary paperLibrary2 = (LocalPaperLibrary)ois.readObject();
ois.close();
```

LocalPaperLibrary 类实现了 Serializable 接口，支持 JDK 原生的序列化与反序列化方式，上述代码为将 LocalPaperLibrary 对象序列化至文件再从文件反序列化为 LocalPaperLibrary 对象的示例。

出于提高比对库加载速度考虑，开发人员可以在 build 完比对库后将比对库序列化存储，之后直接通过反序列化的方式加载比对库，可以将比对库的整体加载用时缩短 30%。

加载比对库虽然会花费一定时间，但通常来说只需要在后端服务启动的时候加载一次，后续的所有查重任务提交等操作都可以共用同一个比对库，不需要再重新加载比对库。

2.5 文本段（PaperSeg）相关

PaperSeg 类一般代表 Paper 对象中的某一句或一段文本，为 Paper 在比对库中预处理后的产物，开发人员通常不会使用到。

2.5.1 PaperSeg 的序列化

```
//序列化 PaperSeg 对象至文件
ObjectOutputStream fos = new ObjectOutputStream(new FileOutputStream("文件路径"));
fos.writeObject(paperSeg);
fos.close();
//从文件反序列化 PaperSeg 对象
ObjectInputStream ois =
        new ObjectInputStream(new FileInputStream(new File("文件路径")));
```

```
PaperSeg paperSeg2 = (PaperSeg)ois.readObject();
ois.close();
```

PaperSeg 类实现了 Serializable 接口，支持 JDK 原生的序列化与反序列化方式。

2.6 自定义查重报告样式

查重 SDK 在查重结束后可以生成 html 格式的查重报告。开发者可以通过 HtmlTemplate 对象的接口对查重报告的样式进行一定程度的自定义。

2.6.1 设置标题

```
template.setTitle("标题");
```

默认为“文本复制检测报告单”。评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.6.2 设置副标题

```
template.setSubtitle("副标题");
```

默认每个报告的副标题为报告类型。人为设置副标题则所有类型的报告均会使用同一副标题。评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.6.3 设置头部提示信息

```
LinkedHashMap<String, String> headerTexts = new LinkedHashMap<>();
headerTexts.put("送检人", "XXX");
headerTexts.put("检测机构", "XXX 检测机构");
template.setHeaderText(headerTexts);
```

查重报告头部信息为重复率表格上方的部分，可以通过 setHeaderText 增加自定义信息。其中标题、作者两项是不可删除的，只能在此基础上新增。setHeaderText 方法需要传入一个 LinkedHashMap<String, String>，代码示例如上。评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.6.4 设置尾部提示信息

```
List<String> footerTexts = new ArrayList<>();
footerTexts.add("本报告由 XXX 检测机构送检，仅供参考");
footerTexts.add("任何问题请联系 XXXXXX");
template.setFooterText(footerTexts);
```

查重报告尾部提示信息即查重报告尾部“注意”字样下展示的信息，可以通过 setFooterText 方法自定义内容。方法入参为一个 List<String>。评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.6.5 设置左上角 logo

```
template.setLeftHeadImage(ImageIO.read(new File("D:/logo.jpg")));
```

setLeftHeadImage 方法需传入 Image 对象，示例中使用 ImageIO 从本地读取图片。图片格式仅支持 jpg 格式，图片的高度最好为 40 像素，宽最好不超过 200 像素。不符合要求的图片将被进行等比缩放。默认没有左上角 logo。评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.6.6 设置右上角 logo

```
template.setRightHeadImage(ImageIO.read(new File("D:/logo.jpg")));
```

seRightHeadImage 方法需传入 Image 对象，示例中使用 ImageIO 从本地读取图片。图片格式仅支持 jpg 格式，图片的高度最好为 40 像素，宽最好不超过 200 像素。不符合要求的图片将被进行等比缩放。默认没有右上角 logo。评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.6.7 设置主背景色

```
template.setBackgroundColor("#f3f9ef");
```

setBackgroundColor 方法需传入 16 进制 RGB 颜色，默认为“#f3f9ef”。 评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.6.8 设置分割线颜色

```
template.setLineColor("#035d03");
```

setLineColor 方法需传入 16 进制 RGB 颜色，默认为“#035d03”。 评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.6.9 设置水印

```
template.setWatermark("本报告由 XXX 系统生成");
```

setWatermark 方法可以设置查重报告背景的文字水印，默认没有水印。水印最多支持 26 个字符（一个中文占两个字符）。 评估许可证不包含该功能（可以进行接口调用，但调用后不生效）。

2.7 查重任务相关

2.7.1 构建查重任务

查重任务的构建使用了建造者模式，分为三个步骤，1 需要获取查重任务构造器，2 配置查重任务的必要参数，3 构建查重任务。

①获取查重任务构造器

```
CheckTask.Builder builder = CheckManager.INSTANCE.getCheckTaskBuilder();
```

②配置查重任务的必要参数

1）设置查重任务 id

```
builder.setUid("001");
```

setUid 方法用于查重任务 id。查重任务 id 应为一个全局唯一的 id，如不设置则自动生成一个 uuid 作为 id。该 id 在查重任务开始、结束、出现异常回调观察者方法时传入。

2）设置待查 Paper

```
builder.addCheckPaper(paper);
```

addCheckPaper 方法设置待查重的 Paper，传入一个 Paper 对象。一个查重任务中可以添加多个待查重的 Paper。

3）设置比对库

```
builder.addLibrary(paperLibrary);
```

addLibrary 方法设置该改查重任务使用的比对库，传入一个 LocalPaperLibrary 对象。一个查重任务中可以添加多个比对库。

4）注册查重任务观察者

```
builder.addCheckState(checkStateImp);
```

addCheckState 方法用于设置一个查重任务观察者接口（CheckState）的实现类以及上下文信息。该项设置时要给可选设置，不设置将使用默认观察者（这意味着您将不能异步获取到查重接口以及接收查重结果等事件的回调），但是出于对查重任务可控性的考量仍然建议设置该项。一个查重任务中可以注册多个观察者。

如果希望传递自定义上下文信息，通过 addCheckState 方法的第二个参数进行传递。

```
builder.addCheckState(checkStateImp, "上下文信息");
```

上下文信息的参数类型通过 CheckState 接口中的泛型 T 指定，这里使用字符串仅为示例。

5）设置查重报告模板

```
builder.setTemplate(template);
```

setTemplate 方法可以设定一个经过自定义的查重模板，如果不进行设置将使用默认查重模板。

6）设置查重白名单

```
List<String> whiteWords = new ArrayList<>();
whiteWords.add("XXX 大学 XXX 学院");
whiteWords.add("山梨醇酐单影脂酸酯");
whiteWords.add("标书技术规格说明书");
builder.addWhiteWord(whiteKeywords);
```

addWhiteWord 方法可以设定一个白单列表。在查重时，如果被查重的句子被完整是白名单列表中的文本的一部分，或被查重的句子中完整包含白名单列表中的内容，将被跳过查重。应用场景：由于长度较长被判定为重复的专有名词、标书的规格说明书等一些允许被引用的内容。addWhiteWord 可以分多次调用添加多个白名单内容。添加白名单会降低查重速度，请留意白名单中文本的总字符数。注意：评估许可证可以正常调用该接口但不会生效。

7）设置查重黑名单（重点关注列表）

```
List<String> blackWords = new ArrayList<>();
blackWords.add("XXX 供应商");
blackWords.add("XXX 市");
blackWords.add("XX 区");
builder.addBlackWord(whiteKeywords);
```

addBlackWord 方法可以设定一个重点关注列表，用于关注一些虽然长度不满足查重阈值，但是需要留意的文本。该列表中的每个字符串限制为 2 到 8 个字符，少于 2 个字符会自动被舍弃，超过 8 个字符会被自动截断为 8 个字符。查重时，如果待查文本中存在重点关注列表中的文本，且比对库中也存在该文本，则该文本将在全文标明引文查重报告中被标记为紫色；如果待查文本中存在重点关注列表中的文本，比对库中不存在该文本，则该文本将在全文标明引文查重报告中被标记为黄色。addBlackWord 可以分多次调用添加多个重点关注列表内容。添加重点关注列表会降低查重速度，请留意重点关注列表中文本的总字符数。注意：评估许可证可以正常调用该接口但不会生效。

8）设置查重算法

```
builder.addCheckCore(new ContinuityCheck(CheckLevel.STRICT));
```

查重任务默认使用 ContinuityCheck(CheckLevel.NORMAL)作为查重算法。通常您不需要手动添加，除非您需要调整查重的严格程度。SDK 提供五种查重严格程度，严格程度从高到低分别为 CheckLevel.VERY_STRICT、CheckLevel.STRICT 、 CheckLevel.NORMAL 、 CheckLevel.EASY 、 CheckLevel.VERY_EASY ， 默 认 为 CheckLevel.NORMAL。

9）添加查重结果过滤器

```
builder.addCheckResultFilter(new CheckResultFilterImpl());
```

addCheckResultFilter 方法可以为查重任务增加一系列自定义的查重结果过滤器，每个查重结果生成后会依次使用查重过滤器进行过滤。查重过滤器需要实现 checker 包下的 CheckResultFilter 接口。应用场景：对查重结果进行二次处理；从查重结果中剔除某些特定的参考文献或句子。注意：评估许可证可以正常调用该接口但不会生效。

③构建查重任务

```
CheckTask checkTask = builder.build();
```

通过 build 方法获得最终的查重任务对象。

查重任务的构建使用了建造者模式，以上多个步骤可以通过连续 set 方式一次性执行完毕，示例代码如下

```
CheckTask checkTask = CheckManager.INSTANCE
        .getCheckTaskBuilder() //获取查重任务构造器
        .setUid("001") //设置任务 id
        .addCheckPaper(paper) //设置待查 Paper
        .addLibrary(paperLibrary) //设置比对库
        .setTemplate(template)  //设置查重报告模板
        .addCheckState(checkStateImp, "上下文信息");  //注册查重任务观察者
        .build(); //构建任务，返回 checkTask 对象
```

2.7.2 启动查重任务

①通过线程池异步启动查重任务

```
checkTask.submit();
```

通过 submit 方法将当前任务提交至 CheckManage 的查重任务线程池。

②同步启动查重任务（不推荐）

```
checkTask.start();
```

CheckTask 类继承了 Thread 类，因此可以直接通过 start 方法启动该线程。

无论采用以上哪种方式启动查重任务，查重结果都将在任务执行完成后回调观察者。也可以使用 checkTask.getReporters()方法获取到所有任务的查重报告对象集合，使用 checkTask.getFailedPapers()方法获取到本次查重任务中查重失败的 Paper 集合。注：如果查重任务未执行完毕时调用 getReporters()或

getFailedPapers()方法，可能返回空指针或空集合。

2.7.3 停止查重任务

```
checkTask.interrupt();
```

在查重任务 submit 或 start 后、执行完毕之前，如果需要停止查重任务、释放该任务所占用的 CPU 和内存资源，可以调用 interrupt 方法。CheckTask 类继承了 Thread 类，并响应了线程的中断，因此通过 interrupt 方法即可结束该线程。需要注意的是，受限于实现机制，XINCHECK SDK 对线程的终止不是立刻的，也无法保证查重任务 100%被停止。CheckTask 中调用 interrupt 方法前已经执行完毕的子任务可能不受影响。

2.7.4 关闭查重任务线程池

```
CheckManager.INSTANCE.shutdown(); //相当于调用线程池的 shutdown 方法
CheckManager.INSTANCE.shutdown(true); //相当于调用线程池的 shutdownNow 方法
```

如果需要彻底关闭线程池释放 sdk 对资源的占用，可以调用 shutdown 方法。在线程池完全关闭（进入 Terminated 态）后，如果提交新的查重任务则线程池将会自动重新打开。在线程池正在关闭时（处于 Shutdown 态但不处于 Terminated 态），如果提交新的任务将会抛出异常。注意：该方法建议仅在确保后续不会有新的查重任务后、程序需要优雅退出时调用，不应在每个查重任务执行完后立刻调用，否则在多线程情况下会导致后续查重任务提交失败。

2.8 查重任务观察者接口（CheckState）

2.8.1 泛型 T

```
public class CheckStateImp implements CheckState<String> {
  //……
}
```

CheckState 接口的泛型 T 为上下文信息的数据类型，指定后通过 addCheckState 方法注册观察者时可以在第二个参数传递对应类型的上下文信息。如果不指定泛型将为 Object。上例是 T 指定为 String 的示例。

2.8.2 taskStart 接口

```
/**
 * 查重任务开始前会回调此方法
 * @param uid         任务唯一 id
 * @param checkPapers 待查的文本对象
 * @param context     上下文信息
 */
void taskStart(String uid, List<Paper> checkPapers, T context);
```

2.8.3 taskFinish 接口

```
/**
 * 查重任务中的全部文本均查重完毕后后回调此方法
 * @param uid          任务唯一 id
 * @param checkPapers  待查的文本对象
 * @param reporters    查重报告（只有查重成功的文本才会有 reporter）
 * @param failedPapers 查重失败的文本对象
 * @param context      上下文信息
 */
void taskFinish(String uid, List<Paper> checkPapers, List<Reporter> reporters, List<Paper>
failedPapers, T context);
```

2.8.4 paperStart 接口

```
/**
 * 查重任务中的某一篇文本开始查重前时会调此方法
 * @param uid      任务唯一 id
 * @param paper    被查 Paper
 * @param context  上下文信息
 */
void paperStart(String uid, Paper paper, T context);
```

2.8.5 paperSuccess 接口

```
/**
 * 查重任务中的某一篇文本查重成功后会调用此方法
 * @param uid      任务唯一 id
 * @param reporter 查重报告
 * @param context  上下文信息
 */
void paperSuccess(String uid, Reporter reporter, T context);
```

2.8.6 paperFailed 接口

```
/**
 * 查重任务中的某一篇文本查重失败后会调用此方法
 * @param uid      任务唯一 id
 * @param paper    失败的 Paper
 * @param code     内部错误码
 * @param t        错误信息
 * @param context  上下文信息
 */
```

```
void paperFailed(String uid, Paper paper, int code, Throwable t, T context);
```

2.8.7 完整实现示例

```
import cn.textcheck.CheckState;
import cn.textcheck.engine.pojo.Paper;
import cn.textcheck.engine.report.Reporter;
import cn.textcheck.engine.type.ReportType;

import java.io.IOException;
import java.util.List;

/**
 * CheckState 实现范例
 */
public class CheckStateImp implements CheckState<Context> {

    public void taskStart(String uid, List<Paper> toCheckPapers, Context context) {
        System.out.println("task start:" + uid);
    }

    public void taskFinish(String uid, List<Paper> checkPapers, List<Reporter> reporters,
List<Paper> failedPapers, Context context) {
        System.out.println("task finish:" + uid);
        try {
            //当全部任务结束后，保存查重报告统计表
            Reporter.saveAsCSV(reporters, context.reportPath + "统计表.csv");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void paperStart(String uid, Paper paper, Context context) {
        System.out.println("paper start:" + uid);
    }

    public void paperSuccess(String uid, Reporter reporter, Context context) {
        System.out.println("paper success:" + uid);
        try {
            //保存两种类型的查重报告
            reporter.saveAsFile(context.reportPath +
reporter.getPaper().getPayload(String.class) + "-1.html", ReportType.TEXT_WITH_CITATION);
//保存全文标明引文报告
            reporter.saveAsFile(context.reportPath +
reporter.getPaper().getPayload(String.class) + "-2.html", ReportType.TEXT_WITH_ORIGINAL);
//保存原文对照报告
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }

    public void paperFailed(String uid, Paper paper, int code, Throwable t, Context context)
{
        System.out.println("paper failed:" + uid);
        t.printStackTrace();
    }
}
```

也可参见 GitHub 中的 Sample2 部分的 CheckStateImp.java 文件。

Github 链接：https://github.com/tianlian0/duplicate-check-sample

2.9 查重报告相关

在 CheckState 接口的 finish 方法中，入参 reporter 为查重报告器。通过该报告器可以直接将查重报告保存为文件，也可以直接获取查重报告的元数据。

2.9.1 将查重报告保存为文件

```
reporter.saveAsFile("D:/全文标明引文报告.html", ReportType.TEXT_WITH_CITATION);
reporter.saveAsFile("D:/原文对照报告.html", ReportType.TEXT_WITH_ORIGINAL);
reporter.saveAsFile("D:/简易报告.html", ReportType.SAMPLE);
```

saveAsFile 方法的第一个参数为文件的保存路径，如果查重报告模板选择的是默认模板，请保存为 mht 文件；如果查重报告模板选择的是 html 模板，请保存为 html 文件。saveAsFile 方法的第二个参数为文件的查重报告类型，TEXT_WITH_CITATION 为全文标明引文报告，TEXT_WITH_ORIGINAL 为原文对照报告，SAMPLE 为简易报告。

2.9.2 获取查重报告 id

```
String reportId = reporter.getReportId();
```

该 id 可以在提交查重任务时手动设置，如不手动设置则由 sdk 自动生成。自动生成的 id 由大写字母、下划线和数字组成，在同一机器上运行的单一实例中生成的 id 是唯一的，如果希望在多个实例中自动生成唯一的 id，请参照高级配置项设置 MachineFlag。

2.9.3 设置查重报告 id

```
reporter.setReportId("00000001");
```

查重报告 id 将显示在最终生成的查重报告上，默认会自动生成一个全局唯一的 id。如果不想使用 SDK 自动生成的查重报告 id，也可以自行设置查重报告 id。需要注意的是，该方法的调用需要在调用 saveAsFile 方法之前，否则将不会生效。

2.9.4 获取待查 Paper 对象

```
Paper paper = reporter.getPaper();
```

Reporter 中会包含该查重报告器的待查 Paper 对象，可以通过 get 方法获取。

2.9.5 获取总重复字符数

```
String copyWords = reporter.getCopyWords();
```

该方法返回全文重复文本的字符数，类型为字符串。

2.9.6 获取总文字复制比

```
String copyRate = reporter.getCopyRate();
```

该方法返回全文总文字复制比，类型为字符串。如“15.5”，代表重复率为 15.5%。

2.9.7 获取前部重复字符数

```
String frontCopyWordCount = reporter.getFrontCopyWords();
```

该方法返回前部重复文本的字符数，类型为字符串。“前部”指全文的前约 50%，当查重时 Paper 对象的正文部分只被划分为一段时，这一段将被算为“前部”。前言目录部分属于前部。

2.9.8 获取前部复制比

```
String frontCopyWordCount = reporter.getFrontCopyWords();
```

该方法返回前部重复文本的字符数占全文字符数的比率，类型为字符串。“前部”指全文的前约 50%，当查重时 Paper 对象的正文部分只被划分为一段时，这一段将被算为“前部”。前言目录部分属于前部。

2.9.9 获取后部重复字符数

```
String endCopyWordCount = reporter.getEndCopyWords();
```

该方法返回后部重复文本的字符数，类型为字符串。“后部”指全文的后约 50%。参考文献部分属于后部，当查重时 Paper 对象的正文部分只被划分为一段时，“后部”将只包含参考文献部分。

2.9.10 获取后部复制比

```
String endCopyWordCount = reporter.getEndCopyWords();
```

该方法返回后部重复文本的字符数占全文字符数的比率，类型为字符串。“后部”指全文的后约 50%。参考文献

部分属于后部，当查重时 Paper 对象的正文部分只被划分为一段时，“后部”将只包含参考文献部分。

2.9.11 获取单篇最大重复字符数

```
String maxSinglePaperCopyWordCount = reporter.getMaxSinglePaperCopyWords();
```

该方法返回单篇最大重复字符数，类型为字符串。单篇最大重复字符数指：待查文本所有重复的文献中，重复字符数最多的一个文献的重复字符数。

2.9.12 获取单篇最大文字复制比

```
String maxSinglePaperCopyWordCount = reporter.getMaxSinglePaperCopyRate();
```

该方法返回单篇最大重复率，类型为字符串。单篇最大文字复制比指：待查文本所有重复的文献中，重复字符数最多的一个文献的重复字符数除以待查文本的总字符数。

2.9.13 获取总段落数

```
String sectionNum = reporter.getSectionNum();
```

该方法返回待查文本的总段落数，类型为字符串。

2.9.14 获取疑似段落数

```
String maxSinglePaperCopyWordCount = reporter.getMaxSinglePaperCopyRate();
```

该方法返回待查文本的疑似段落数，类型为字符串。疑似段落数指：所有段落中存在文字复制行为的文本块的数量。

2.9.15 获取疑似段落最大重合字符数

```
String maxSectionCopyWords = reporter.getMaxSectionCopyWords();
```

该方法返回疑似段落最大重复字符数，类型为字符串。疑似段落最大重合字符数指：待查文本的所有存在重复的文本段中，重复字数最多的段的重复字数。

2.9.16 获取疑似段落最大复制比

```
String maxSectionCopyWords = reporter.getMaxSectionCopyRate();
```

该方法返回疑似段落最大重复字符数，类型为字符串。疑似段落最大重合字符数指：待查文本的所有存在重复的文本段中，重复字数最多的段的重复字数占该段总字数的比率。

2.9.17 获取疑似段落最小重合字符数

```
String minSectionCopyWords = reporter.getMinSectionCopyWords();
```

该方法返回疑似段落最小重复字符数，类型为字符串。疑似段落最小重合字符数指：待查文本的所有存在重复的文本段中，重复字数最少的段的重复字数。

2.9.18 获取疑似段落最小复制比

```
String minSectionCopyWords = reporter.getMinSectionCopyRate();
```

该方法返回疑似段落最小重复字符数，类型为字符串。疑似段落最小重合字符数指：待查文本的所有存在重复的文本段中，重复字数最少的段的重复字数占该段总字数的比率。

2.9.19 获取疑似段落单篇最大重合字符数

```
String minSectionCopyWords = reporter.getMaxSectionSinglePaperCopyWords();
```

该方法返回疑似段落单篇最大重合字符数，类型为字符串。疑似段落单篇最大重合字符数指：待查文本的所有文本段的相似文献列表中重复字数最多的参考文献的重复字数。

2.9.20 获取疑似段落单篇最大复制比

```
String minSectionCopyWords = reporter.getMaxSectionSinglePaperCopyRate();
```

该方法返回疑似段落单篇最大复制比，类型为字符串。疑似段落单篇最大复制比指：待查文本的所有文本段的相似文献中重复字数最多的参考文献的重复字数占该段文本字数的比率。

2.9.21 获取全文重复文献重复率列表

```
List<Pair<Paper, String>> list = reporter.getSectionMaxCopyRateList();
```

该方法返回待查文本所抄袭的文献中，按重复字数的多少从大到小排序的列表。类型为 List<Pair<Paper, String>>，Pair 的 key 值为重复文献对象，value 值为重复率。

2.9.22 将 Reporter 列表保存为 csv 文件

```
Reporter.saveAsCSV(reporters, "csv 文件路径");
```

saveAsCSV 方法时 Reporter 提供的一个静态工具方法，可以将多个 Reporter 导出为一 csv 格式的统计表。通常用于 CheckState 回调的 taskFinish 方法中，在查重任务全部结束后导出统计表。第一个参数为 List<Reporter>，第二个参数为 csv 文件保存的路径。

2.9.23 获取查重结果（CheckResult）对象

```
List<CheckResult> checkResults = reporter.getCheckResults();
```

getCheckResults 方法可以获取到 Paper 中文本分段的查重结果数据。每个 CheckResult 对象中包含了所对应文本分段的全部查重结果元数据。关于 CheckResult 中包含的数据及提供的接口，请参考第 2.9 节的介绍。

2.10 查重结果对象（CheckResult）相关

2.10.1 获取 uid

```
String uid = checkResult.getUid();
```

获取唯一 id。每个查重结果的唯一 id 均不相同。

2.10.2 获取 sectionId

```
String sectionId = checkResult.getSectionId();
```

获取 SectionId。同一种算法对相同 Paper 中的同一段文本的查重结果，其 setcionId 应该相同。

2.10.3 获取所对应的文本分段的原文

```
String text = checkResult.getSectionText();
```

2.10.4 获取所对应的文本分段的重复字符数

```
int count = checkResult.getCopyWordsCount();
```

2.10.5 获取 copySegMap

```
Map<PaperSeg, List<PaperSeg>> copySegMap = checkResult.getCopySegMap();
```

copySegMap 是 CheckResult 中非常重要的一个数据结构，其 map 映射关系为：被判定为重复的待查文本的中的文本段->比对库中的来源文本段列表。

2.10.6 获取 copyDetailMap

```
LinkedHashMap<String, Pair<Paper, Integer>> copyDetailMap = checkResult.getCopyDetailMap();
```

copyDetailMap 是 CheckResult 中非常重要的一个数据结构，其 map 映射关系为：比对库中 Paper 的

paperid->(比对库中的 Paper->该 Paper 被复制的字符数)。

2.11 高级配置项

为满足开发者对 SDK 的定制化需求，XINCHECK 文本查重 SDK 通过 cn.textcheck.engine.config.Config 配置中心提供了丰富的高级配置项。开发者可以根据开发和项目需要，使用三种方式进行配置项的修改。

修改配置项目务必在所有 SDK 方法调用前进行。配置中心的修改不是线程安全的，一旦有任务执行后不应再进行配置项的修改，否则可能会导致任务执行出错。

①逐个修改配置项的值

```
Config.ITEM.put("配置项", "值");
```

②通过 Map 批量修改配置项的值

```
Map<String, String> config = new HashMap<>();
config.put("配置项 1", "值 1");
config.put("配置项 2", "值 2");
Config.ITEM.put(config);
```

Map 中，key 为配置项名称，value 为配置项的值。

③通过格式化的字符串批量修改配置项的值

```
String config = "配置项 1:值 1,配置项 2:值 2";
Config.ITEM.put(config);
```

字符串中每个配置项之间需要用英文逗号隔开，配置项和配置项的值之间用英文引号隔开。

SDK 支持以下配置项

序号	配置项	配置项功能说明	默认值	取值范围	其它说明
1	checkIgnoreForeword	查重时是否忽略目录前言部分	false	true/false	Paper紧凑等级>=2该项失效
2	checkIgnoreReferences	查重时是否忽略参考文献部分	true	true/false	Paper紧凑等级>=3该项失效
3	segmentIgnoreForeword	文本分段时是否不进行目录前言的识别	false	true/false	
4	segmentIgnoreReference	文本分段时是否不进行参考文献的识别	false	true/false	
5	filenameSeparator	Paper对象实例化时通过文件名加载额外信息的文件名分隔符	@	任意字符串（不可包含正则表达式转义字符）	
6	maxCheckTaskConcurrent	查重任务线程池的maxSize	10	整型，[1,Integer.MAX_VALUE]	该项配置请根据业务情况修改
7	checkTaskConcurrentQueueSize	查重任务线程池等待队列的大小	500000	整型，[1,Integer.MAX_VALUE]	该项配置请根据业务情况修改
8	maxSingleCheckTaskConcurrent	单个Paper查重任务的线程池的maxSize	CPU核心数减1	整型，[1,Integer.MAX_VALUE]	该项配置请根据业务情况修改
9	singleCheckTaskConcurrentQueueSize	查重任务线程池等待队列的大小	500000	整型，[0,Integer.MAX_VALUE]	该项配置请根据业务情况修改
10	maxSubtaskConcurrent	每个Paper分段查重时的线程池大小	4	整型，[1,Integer.MAX_VALUE]	该项配置请根据业务情况修改
11	excludeQuoteMark	带有引用符号的句子是否不进行查重	false	true/false	该项配置请谨慎修改
12	multiThreadLoadLibrary	是否开启比对库多线程加载	true	true/false	评估许可为false
13	paperCompactLevel	Paper对象实例化时的默认紧凑等级	0	[0,4]	
14	encrypt	Paper、PaperSeg、LocalPaperLibrary序列化时是否启用AES加密	false	true/false	开启加密会降低序列化/反序列化的性能
15	paperAESKey	Paper序列化时的AES加密密钥	-	通过Config.getAES256Key()方法获得随机密钥	需先开启encrypt才生效
16	paperSegAESKey	PaperSeg序列化时的AES加密密钥	-	通过Config.getAES256Key()方法获得随机密钥	需先开启encrypt才生效
17	reportAutoIncrement	DefaultReorter生成的查重报告的自增id位的起始值	0	长整型，[0,Long.MAX_VALUE]	
18	paperLibraryBuildInitialCapacity	构建比对库时存储分段的数组列表的初始化大小	1000000	整型，[0,Integer.MAX_VALUE]	
19	checkLevel	准匹配算法使用无参构造方法时的查重阈值	3	整型，[1,5]	
20	segmentSize	查重报告中每个查重文本段的大小	10000	整型，(3000,Integer.MAX_VALUE]	该项配置请谨慎修改
21	segmentSizeRandomFluctuate	查重报告中每个查重文本段的大小是否上下抖动	true	true/false	该项配置请谨慎修改
22	machineFlag	机器标识，多实例时用以自动生成查重报告id	-	10位以下的字符串	该项配置请谨慎修改
23	skipSamePaper	是否跳过完全相同的文件的查重	true	true/false	
24	copyPaperListSize	相似文献列表最多展示的文献数	100	整型，[1,1000]	
25	copySegListSize	展示每句文本所对应相似文本的最多条目数	5	整型，[1,1000]	

注：评估许可证只能进行第 1、2、5、8、10 项的配置。

3 算法介绍

互联网中可以检索到很多查重算法的理论研究和 demo 代码，但是在进行大规模商业化应用时必须同时兼顾准确性和性能，使用网络上的公开算法往往达不到要求。XINCHECK SDK 采用了业内认可的 ContinuityCheck 文本指纹算法对文本进行查重，经过数年的生产环境验证，兼顾了准确性和计算性能。

查重任务默认使用 ContinuityCheck(CheckLevel.NORMAL)作为查重算法。通常您不需要修改，除非您需要调整查重的严格程度。SDK 提供五种查重严格程度，严格程度从高到低分别为 CheckLevel.VERY_STRICT（非常严格）、CheckLevel.STRICT（严格）、CheckLevel.NORMAL（通用）、CheckLevel.EASY（宽松）、CheckLevel.VERY_EASY（非常宽松），默认为 CheckLevel.NORMAL（通用）。

算法的性能指标如下：

内存占用：SDK 启动后会占用约 50MB 常驻内存。比对库中每一千万字次需要约 150MB 常驻内存。此外，在加载比对库时解析 word 和 pdf 文档时所需的动态内存约 1~2GB，动态内存在 GC 后会被回收。

CPU 占用：在没有热河任务运行时，SDK 不会占用 CPU。在进行比对库导入时，如果开启了多线程导入通常会占用约 50%的 CPU，如果未开启多线程导入通常会占用一个 CPU 核心。在进行查重任务时，每个待查文本的文本段将占用一个 CPU 核心，SDK 默认将每 1 万字划分为一个文本段。如果您需要预留部分服务器的 CPU 资源用于其它服务，可以对 SDK 进行并发配置，限制 SDK 的 CPU 占用。

性能数据仅供参考具体还请自行测试。

附录 1：修改 JDK 加密策略文件

如果您使用的是 JDK/JRE7 及以下版本，XINCHECK SDK 不提供公开的支持，建议您在条件允许的情况下升级 JDK 至 JDK8。

如果您使用的是 JDK8 但 JDK/JRE 为 1.8.0_150 及以下版本，您需要下载 Oracle 官方提供的 JCE 无限制权限策略文件以启用 AES256 加密。或将 JDK 升级至 1.8.0_151 及以上。

下载地址：https://www.oracle.com/java/technologies/javase-jce8-downloads.html

下载后解压，可以看到 local_policy.jar 和 US_export_policy.jar 两个文件。

如果安装了 JRE，请将两个 jar 文件放到%JRE_HOME%\lib\security 文件夹下覆盖原文件。

如果安装了 JDK，请将两个 jar 文件放到%JDK_HOME%\jre\lib\security 文件夹下。

如果您使用的 JDK/JRE 版本为 1.8.0_151（包含 151）及以上版本（包括 JDK9/10/11/12/13 及以上），您默认不需要进行任何操作。

附录 2：查重报告解读及指标说明

根据被检测文本的内容、字符数以及开发者配置的分块策略，系统可能会对待查文本进行分块处理。默认情况下，系统将自动识别文本的前言目录部分和参考文献部分，前言目录部分将作为查重报告分块的第一块，参考文献部分将作为查重报告分块的最后一块。对于中间的正文部分，默认情况下大约每 1 万字划分为一块。具体的分块规则可由开发者自行定义。

查重报告展示三大部分内容：1、基本信息，包括文本标题、作者、查重报告编号、查重时间等。2、整体检测指标，包括总文字复制比、单篇最大文字复制比、重复字数、总字数、单篇最大重复字数等等。3、全文标明引文或原文对照结果。

各项检测指标说明如下：

总字数：指被检测文本的总有效字符数。

重复字数：指系统计算出的被检测文本与比对库中文本比对后，出现重复的总字符数。

总文字复制比：指被检测论文总的重复字符数占全文字符数的比例，即重复字数除以总字数减去参文献部分的字符数得到的比值。通过该指标，可以直观了解重复字数在该被查文本中所占比例。

总段落数：如果被检测文本被进行了分块处理，则代表被检测文本被分成的块数。如果被检测文本没有被进行分块处理，该值为 1。

疑似段落数：指存在文字复制行为的文本块的数量。

疑似段落最大重合字数：所有存在重复的文本块中，重复字数最多的段的重复字符数。

疑似段落最小重合字数：所有存在重复的文本块中，重复字数最小的段的重复字符数。

单篇最大重复字数：指被检测文本与比对库中文本比对后，得到的所有的相似文献中，重复字数最多的那一个文献的重复字数。

单篇最大文字复制比：指被检测文本与比对库中文本比对后，得到的所有的相似文献中，重复字数最多的那一个文献的重复字数占待查文本总字数的比率。

前部重合字数：指被检测文本的前言目录部分以及正文部分约前 50%部分的重复字符数。

后部重合字数：指被检测文本的参考文献部分以及正文部分约后 50%部分的重复字符数。

以下查重指标在默认生成的查重报告中没有展示，但是开发者可以通过接口获取，在此也进行解释。

获取疑似段落最大复制比：所有存在重复的文本块中，重复字数最多的段的重复字符数占全文字符数的比例。

获取疑似段落最小复制比：所有存在重复的文本块中，重复字数最小的段的重复字符数占全文字符数的比例。

疑似段落单篇最大重合字符数：所有文本段的相似文献列表中重复字数最多的参考文献的重复字数。

疑似段落单篇最大复制比：所有文本段的相似文献中重复字数最多的参考文献的重复字数占该段文本字数的比率。

前部复制比：指被检测文本的前言目录部分以及正文部分约前 50%部分的重复字符数占全文字符数的比例。

后部复制比：指被检测文本的参考文献部分以及正文部分约后 50%部分的重复字符数占全文字符数的比例。