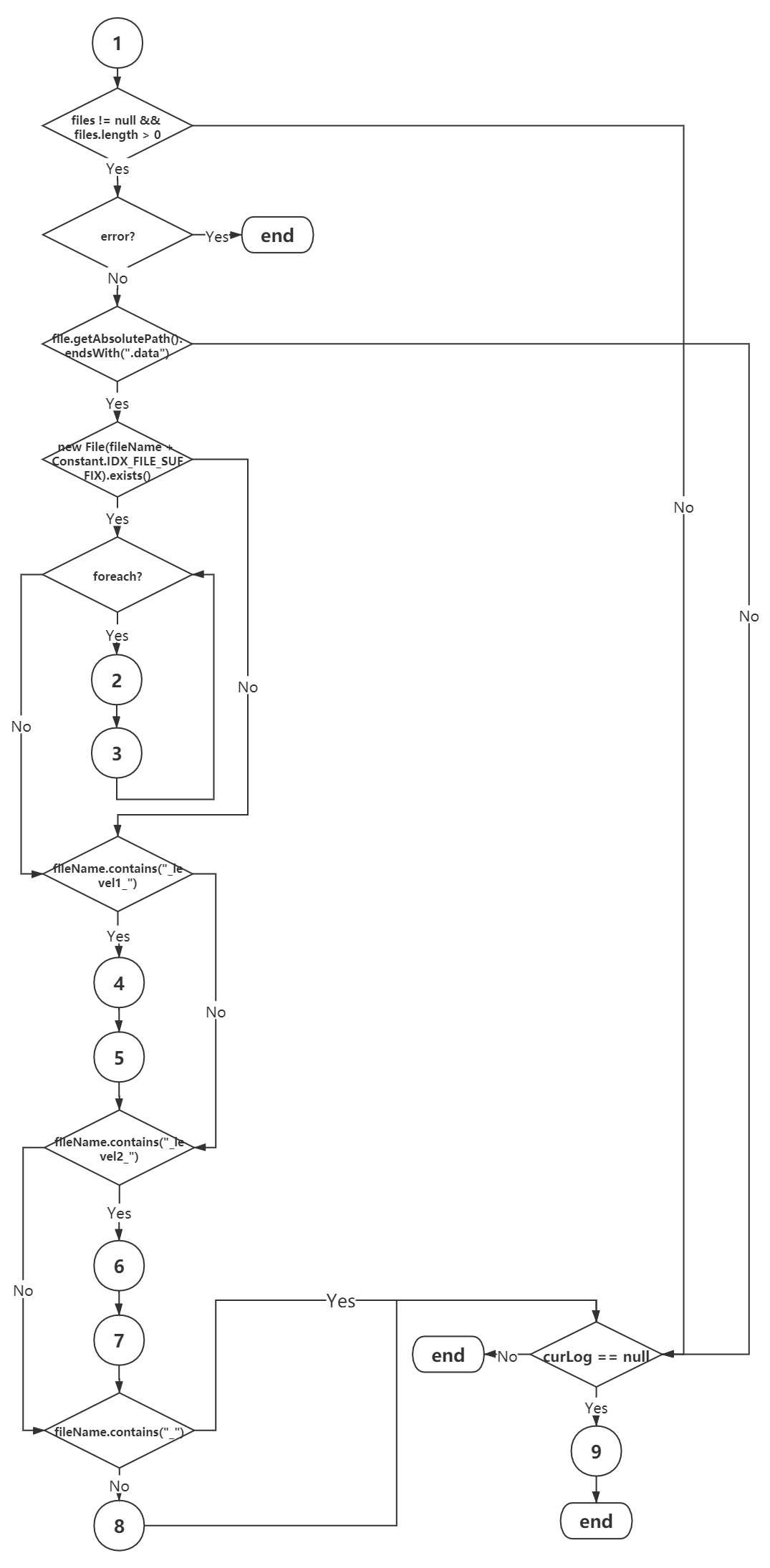
**一、简介**

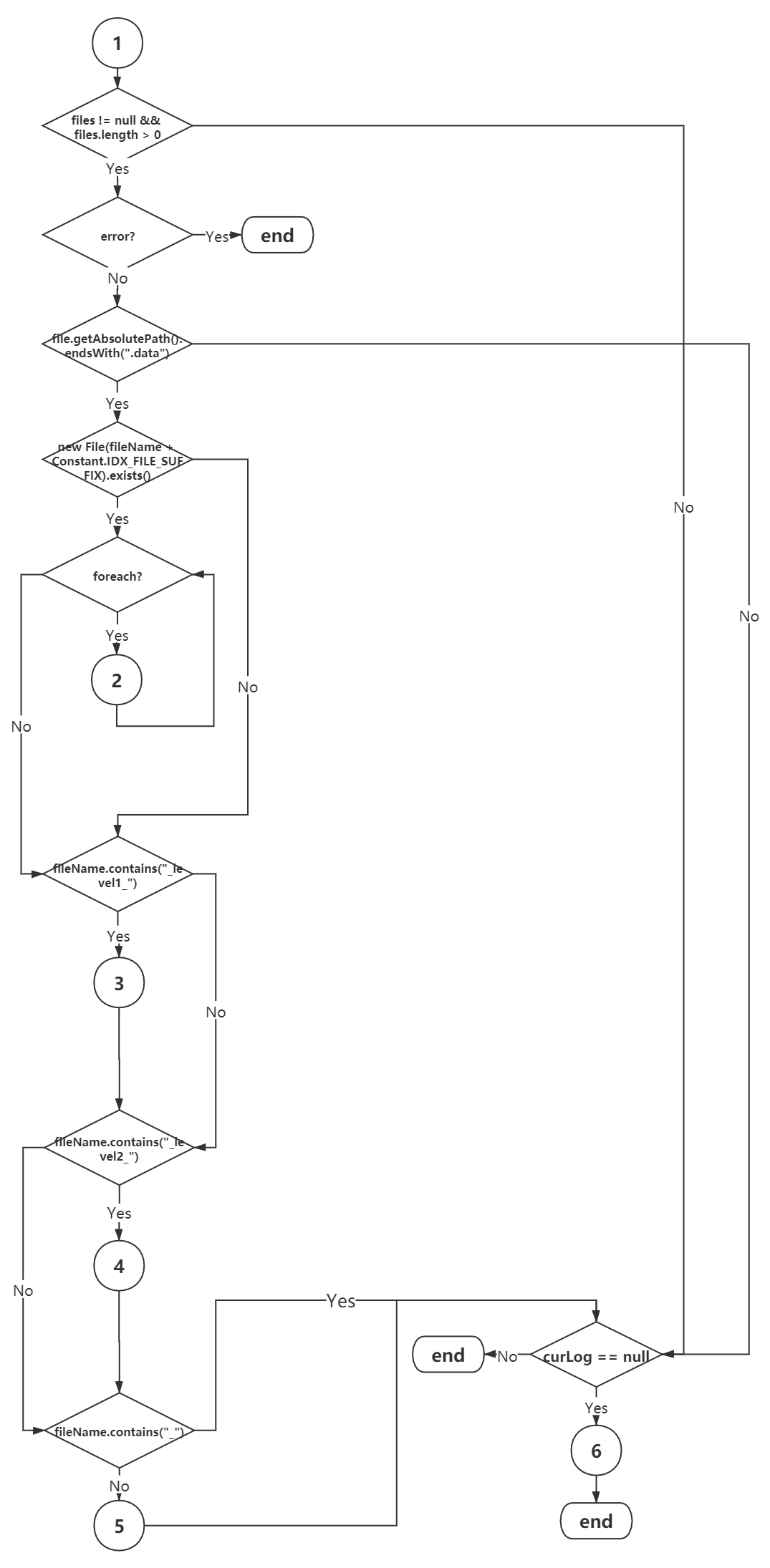
对于loadidx 函数，我们分别采取基于数据流的测试用例生成方法和基于路径的测试用例生成方法。

需要注意的是，为了方便测试和展示，我们对所有的目标测试函数进行行号重排，即行号自函数头从1开始重新计数，并忽略代码中的空白行、注释等。

**二、程序图**



**图1 原始程序图**



**图2 DD路径图**

**三、基于数据流的测试**

**1、统计变量的定义/使用节点**

在对程序图进行简化和绘制后，我们可以对loadidx中的所有变量的定义/使用节点进行统计，具体可见表1。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 变量 | 定义节点  原始程序图 | 定义节点  简化程序图 | 使用节点  原始程序图 | 使用节点  简化程序图 |
| Files | 1 | 1 | 1.5 | 1.5 |
| Filename | 2 | 2 | 3.5 | 3.5 |
| Logfile | 3 | 3 | 4.5 | 4.5 |
| level1Num | 5 | 5 | 7 | 7 |
| level2Num | 6 | 6 | 8 | 8 |

**表1 loadidx 的变量的定义/使用节点**

**2、列出变量的定义-使用路径**

在列出所有变量的定义/使用节点后，我们可以依据简化程序图的节点情况，依次列出所有变量的定义-使用路径，并且注明其是否为定义-清除路径，具体可见表2。同时，因为部分路径存在重合部分，因为我们对路径进行了整理，比如用<p1, 4>来代表p2的路径。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 变量 | 路径 | 完整路径 | 简化路径 | 是否为定义-清除路径 |
| Files | P1 | 1, 1.5 | <P4, 1> | Y |
| Filename | P2 | 2, 3, 3.5 | <P7, 2> | Y |
| Logfile | P3 | 3, 4, 4.5 | <P10, 3> | Y |
| level1Num | P4 | 5，6，7 | <P19, 6> | Y |
| Level2Num | P5 | 6，7，8 | 7, 8, 9 | Y |

**表2 loadidx 的变量的定义-使用路径**

**3、选取测试覆盖指标，生成测试用例**

在这一步，我们选用全定义准则作为测试覆盖指标，即选择每一个定义节点到一个使用节点的定义-清除路径。而对于所有的定义-使用路径，依照表2中的简化路径可知，我们可以用尽可能少的测试用例去覆盖尽可能多的目标路径，比如对于路径 P1、P2、P3，我们只需要生成一个可以覆盖 P3 的测试用例即可。而具体的生成的测试用例，我们放在附件中予以展示，此处略去。

**4、编写代码，进行测试**

上述步骤就是基于数据流的测试用例生成过程，接下来我们就能够编写代码，进行测试。由于按照全使用准则，我们的覆盖率必然是100%。

**四、基于路径的测试**

**1、绘制程序图、DD路径图**

我们根据源代码，绘制程序图和 DD 路径图。具体图像如文档第二部分所示。

**2、选取测试覆盖指标，生成测试用例**

这里我们采用分支/条件覆盖的路径测试方法。分支/条件覆盖是执行足够的测试用例，使得分支中每个条件取到各种可能的值，并使每个分支取到各种可能的结果。其缺点是覆盖指标为 C1p，可能会出现覆盖率不足 100% 的情况，有点是测试用例较少，设计较为简单方便。

如果我们采取覆盖率更高的 C∞ 路径测试，虽然覆盖率能够达到100%，但是测试用例将会呈指数级增加，实际不可行。

所以我们综合数据流与路径测试两种方法，最终使得覆盖率达到 100% 。而具体的生成的测试用例，我们放在附件中予以展示，此处略去。

**3、编写代码，进行测试**

上述步骤就是基于路径的测试用例生成过程，接下来我们就能够编写代码，进行测试。具体的测试代码，我们放在附件中予以展示，此处略去。