碳烤黄蜂

功能开发指南

2023 年 8 月 21 日

Potato

目录

功能开发指南是便于开发者拓展内核功能时查询某功能对应开发方法的说明书和检查单。

章节 1

正文

1.1 编写解析器和管理器

解析器用于解析单词本文件，它将单词本文件的单词、历史记录和单词本信息分别解析为 `List<Word>`、`List<History>`和 `Info`。管理器则是用于单词本的增、删、改。

1.1.1 编写解析器

`Parser` 类只有构造器和一个叫做 `void parser()` 的抽象方法。构造器会调用 `void parser()` 方法以解析传入的文件，`void parser()` 会将解析后的结果放到 `Parser` 的 `List<Word>`、`List<History>`和 `Info` 中，因此在获得了解析器对象后直接对实例对象使用 `Getter` 方法即可获得解析出的全部单词。

最后实现的效果是可以像这样调用：

代码清单 1.1 示例

```
public void parserTest()
{
    // 首先要在Config中配置好Parser
    /*
    try
    {
        Config.setParser(WordFileType.DATABASE, MyParser.class.getConstructor(File.class));
    }
    catch (NoSuchMethodException e)
    {
        Log.e(getClass().toString(),
            String.format("将文件类型%s的解析器替换为%s时出现错误", WordFileType.DATABASE.type
                (), MyParser.class), e);
        throw new RuntimeException(e);
    }
     */
    File file = new File("a.db");
    // AutoParser会根据文件类型自动获取对应的解析器
    // 数据库文件的默认解析器是内核中提供的DatabaseParser
    // 在Config中修改后则会使用我们提供的修改器
    AutoParser parser = new AutoParser(file);
    List<Word> wordList = parser.getWordList();
}
```

编写解析器，首先需要继承 `Parser` 类并实现构造器和 `void parser()` 方法。

代码清单 1.2 初始布局

```
import com.potato.Parser.Parser;
import java.io.File;

public class MyParser extends Parser
{
    public MyParser(File file, String extension)
    {
        super(file, extension);
    }

    @Override
    protected void parser()
    {

    }
}
```

由于我们自定义的解析器对应的文件类型是确定的，因此我们应该将构造器中的 `String extension` 换成我们解析器对应的文件类型。所有单词本文件的文件类型都定义在了 `com.potato.ToolKit.WordFileType` 中，我们以数据库为例，构造器更改如下所示：

代码清单 1.3 构造器

```
public MyParser(File file)
{
    super(file, WordFileType.DATABASE.type());
}
```

接下来编写 `void parser()` 方法。这个方法是解析单词本文件的具体实现，它是一个抽象方法（因此必须实现），将直接被 `Parser` 的构造器调用。因此如果有需要在解析前初始化的对象，需要写在构造器中。

代码清单 1.4 Parser 的构造器源码

```
/**
 * Parser用于解析储存单词本的文件
 *
 * @param file       需要解析的文件
 * @param extension 文件应有的扩展名，对于文件xxx.db，其扩展名是db
```

```
 *                  扩展名建议使用WordFileType中已定义的枚举
 */
public Parser(File file, String extension)
{
    // 使用卫语句捕捉文件类型错误
    if (!extension.equals(getExtensionName(file)))
    {
        Log.e(getClass().toString(), String.format("解析文件%s类型错误", file.getName()));
    }
    this.file = file;

    parser();
    Log.i(getClass().toString(), String.format("解析文件%s成功", file.getName()));
}
```

`Parser` 中定义了三个静态变量：`wordList`、`info` 和 `historyList`，它们分别储存单词列表、单词本信息和历史记录（单词本文件的详细说明和格式规范、单词的构造方法、历史记录和数据库的构造器请见 `API` 指南）。这三个静态变量都给出了对应的 `Setter`，`void parser()` 方法的作用就是给这三个变量赋值。

代码清单 1.5 parser 方法

```
@Override
protected void parser()
{
    List<Word> wordList = new ArrayList<>();
    List<History> historyList = new ArrayList<>();
    Info info = null;

    // 对以上三者赋值
    // ...

    setWordList(wordList);
    setHistoryList(historyList);
    setInfo(info);
}
```

至此就完成了解析器的编写。我们在调用 `AutoParser` 时，它会从 `Config.parserMap` 中选取对应文件类型的解析器，因此我们要在这里更改解析器为我们的解析器。

`Config` 提供了 `setParser(WordFileType, Constructor<? extends Parser>)` 方法用于修改某文件类型对应的解析器。修改解析器建议在 `Config` 初始化时就进行，以免因为程序执行顺序不当而选用了错误了解析器。

代码清单 1.6 修改解析器

```
// getConstructor()的参数是MyParser构造器中每个参数对应类型的class
// 这里需要处理getConstructor()抛出的异常，推荐使用try-catch+Log
// 对于不需要日志的，使用@SneakyThrows也可以
// 但是不要在方法上添加抛出标记
Config.setParser(WordFileType.DATABASE, MyParser.class.getConstructor(File.class));
```

1.1.2 编写管理器

`Manager` 类方法定义了 `List<Word> insertWords`、`List<Word> deleteWords` 和 `Map<Word, Word> modifyWords` 等用于分别储存需要增、删、改的单词、历史记录和数据库信息。`Manager` 类的 `void insert(Word)`、`void delete(Word)` 和 `void modify(Word, Word)` 等方法用于给上述储存增、删、改动作的数据结构添加数据（这几个方法都给出了对应的 `Getter`）。最后通过 `void push()` 方法将所有修改写入到单词本文件中。一般情况下不需要修改 `void insert(Word)` 等方法，只需要实现 `void push()` 抽象方法即可。

最后实现的效果是可以像这样调用：

代码清单 1.7 示例

```
@Test
public void managerTest()
{
    // 首先要在Config中配置好Manager
    /*
    try
    {
        Config.setManager(WordFileType.DATABASE, MyManager.class.getConstructor(File.class));
    }
    catch (NoSuchMethodException e)
    {
        Log.e(getClass().toString(),
```

```
            String.format("将文件类型%s的管理器替换为%s时出现错误", WordFileType.DATABASE.type
                (), MyManager.class), e);
        throw new RuntimeException(e);
    }
     */
    File file = new File("a.db");
    // AutoManager会根据文件类型自动获取对应的管理器
    // 数据库文件的默认管理器是内核中提供的DatabaseManager
    // 在Config中修改后则会使用我们提供的管理器
    AutoManager manager = new AutoManager(file);

    // Word构造方法请见API指南
    Word insertWord = new Word.WordBuilder().name("test").build();
    manager.insert(insertWord);  // 插入单词

    // 需要删除的单词
    Word deleteWord = // ...
    manager.delete(deleteWord);

    manager.push();  // 将修改写入单词本文件
}
```

编写管理器，首先需要继承 `Manager` 类并实现构造器和 `void push()` 方法。

代码清单 1.8 初始布局

```
import com.potato.Manager.Manager;

import java.io.File;

public class MyManager extends Manager
{
    public MyManager(File file, String extension)
    {
        super(file, extension);
    }

    @Override
    public void push()
    {

```

```
16      }
17  }
```

与解析器相同，由于我们自定义的管理器对应的文件类型是确定的，因此我们应该将构造器中的 `String extension` 换成我们解析器对应的文件类型。我们以数据库为例，构造器更改如下所示：

代码清单 1.9 构造器

```
1  public MyManager(File file)
2  {
3      super(file, WordFileType.DATABASE.type());
4  }
```

接下来编写 `void push()` 方法。该方法是在完成全部增、删、改操作后被调用的。实现它的一般思路就是遍历储存动作的几个数据结构并做出对应的修改。对于数据库或者 `IO` 流，不要忘记最后的 `void close()` 方法。

代码清单 1.10 构造器

```
1  @Override
2  public void push()
3  {
4      for (Word w : getInsertWords())
5      {
6          // 将w加入单词本文件
7      }
8
9      for (Word w : getDeleteWords())
10     {
11         // 将w从单词本文件删除
12     }
13
14     for (Map.Entry<Word, Word> w : getModifyWords().entrySet())
15     {
16         Word from = w.getKey();
17         Word to = w.getValue();
18         // 将from在单词本中改为to
19     }
20
21     // 处理History的代码与上面类似
22     // ...
```

```
23
24      // 处理Info
25      // ...
26
27      // close()
28  }
```

至此就完成了管理器的编写。我们在调用 `AutoManager` 时，它会从 `Config.managerMap` 中选取对应文件类型的管理器，因此我们要在这里更改管理器为我们的管理器。

与修改器类似，`Config` 提供了 `setManager(WordFileType, Constructor<? extends Manager>)` 方法用于修改某文件类型对应的管理器。修改管理器建议在 `Config` 初始化时就进行，以免因为程序执行顺序不当而选用了错误了管理器。

代码清单 1.11 修改管理器

```
1  Config.setManager(WordFileType.DATABASE, MyManager.class.getConstructor(File.class));
```

1.2 日志输出器

1.2.1 日志的结构

日志类的 UML 图如下所示

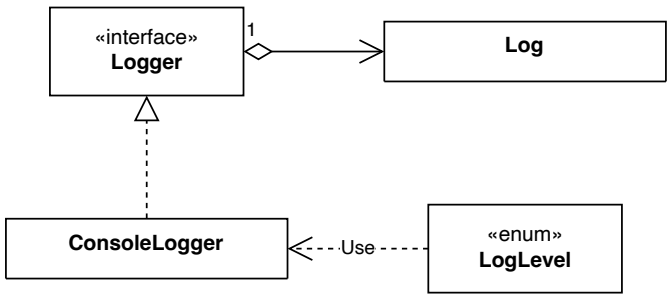

图. 1.1 日志类的 UML 图

其中 ConsoleLogger 是内核中的一个日志输出器的实现。我们自己实现的日志输出器会替代它的位置。在实际使用日志时调用的是 Log 类，它包含一个 Logger，因此我们的日志输出器要实现 Logger 接口。LogLevel 是日志的等级，如下表所示。

日志等级	含义
v	冗长信息，指一般不需要过多关注，但是需要在日志中输出的信息
d	debug 信息，建议在单元测试使用
i	一般信息
w	警告信息，比如传入的参数不规范，可能会导致报错时使用
e	错误信息

1.2.2 编写日志输出器

编写日志输出器，首先要继承 Logger 接口并实现其中的所有方法。

代码清单 1.12 Logger 初始布局

```
1 public class MyLogger implements Logger
```

```
{
    @Override
    public void v(String tag, String message)
    {

    }

    @Override
    public void v(String tag, String message, Throwable throwable)
    {

    }
    // 其它方法
    // ...
}
```

每个方法的方法名都对应了日志等级。参数 `String tag` 是日志的标签，其作用是标记出给出日志的类。参数 `String message` 是日志的内容。参数 `Throwable throwable` 用于将报错的信息打印出来以便定位错误。这个参数是可选的，因为显然不一定所有日志都有报错信息。

这里给出 `ConsoleLogger` 的部分源码以供参考。

代码清单 1.13 ConsoleLogger 源码

```
package com.potato.Log;

/**
 * ConsoleLogOutput是将日志输出至终端的一个实现
 */
public class ConsoleLogger implements Logger
{
    /**
     * 一般日志输出至标准输出流
     *
     * @param tag       用于分辨日志来源，可以设置为类名
     * @param message   日志信息
     * @param throwable 提供给日志的报错
     */
    private void normalPrintln(String tag, String message, Throwable throwable, LogLevel
        logLevel)
```

```
16      {
17          if (throwable != null)
18          {
19              System.out.printf("TAG:%s\t%s:%s\tTHROWABLE:%s%n", tag, logLevel, message,
                    throwable);
20          }
21          else
22          {
23              System.out.printf("TAG:%s\t%s:%s%n", tag, logLevel, message);
24          }
25      }
26
27      // 省略errorPrintln()的代码，它就是把normalPrintln()的out换成的err
28      // 省略v()和d()的代码
29
30      /**
31       * 输出一般的信息
32       *
33       * @param tag     用于分辨日志来源，可以设置为类名
34       * @param message 日志信息
35       */
36      @Override
37      public void i(String tag, String message)
38      {
39          normalPrintln(tag, message, null, LogLevel.INFO);
40      }
41
42      /**
43       * 输出一般的信息
44       *
45       * @param tag       用于分辨日志来源，可以设置为类名
46       * @param message   日志信息
47       * @param throwable 提供给日志的报错
48       */
49      @Override
50      public void i(String tag, String message, Throwable throwable)
51      {
52          normalPrintln(tag, message, throwable, LogLevel.INFO);
53      }
```

```
54
55      // 省略w()的代码
56
57      /**
58       * 输出错误信息
59       *
60       * @param tag     用于分辨日志来源，可以设置为类名
61       * @param message 日志信息
62       */
63      @Override
64      public void e(String tag, String message)
65      {
66          errorPrintln(tag, message, null, LogLevel.ERROR);
67      }
68
69      /**
70       * 输出错误信息
71       *
72       * @param tag       用于分辨日志来源，可以设置为类名
73       * @param message   日志信息
74       * @param throwable 提供给日志的报错
75       */
76      @Override
77      public void e(String tag, String message, Throwable throwable)
78      {
79          errorPrintln(tag, message, throwable, LogLevel.ERROR);
80      }
81  }
```

在编写完日志输出器后，在 `Config` 初始化时将 `Logger logger` 参数设置为自定义的日志输出器即可。

代码清单 1.14 设置日志输出器

```
1  File configFile = // 配置文件
2  Config.initial(configFile, null, null, null, new MyLogger());
```

这里给出一个调用 `Log` 的实例：

代码清单 1.15 Log 调用实例

```
1  catch (NoSuchMethodException e)
```

```
2 {
3     Log.e(Config.class.toString(), "未找到对应解析器或管理器", e);
4     throw new RuntimeException(e);
5 }
```