Rotating Index Load Balancer Pre-Production Specification

Lani Wagner
lani.wagner@students.fhnw.ch

2022-03-24 11:59:58

Abstract

This document contains the pre-production specificatino written for an implementation of a rotating index load balancer to reduce the I/O load for a service that is tasked with handling a large amount of requests, so that the I/O bottleneck is greatly reduced. This specification was written for the FHNW module Engineering Writing (engw).

Contents

1 General

The result of this specification will be a library. This document documents the features this library must provide. This specification is purposefully *not* programming-language-specific to allow the implementation in any language that best suits the existing infrastructure. The terminology in this document will be leaning on the terms most often used in while working with the java programming-language, such as type definitions. When working in another language the appropriate equivalent must be used in consideration of applicability and performance.

Any specifications derived from referenced documents are overridden by any specifications in this document.

2 Design

The library shall provide an instantiable Singleton[4] class named `RotationBalancer<T>` with the following arguments. The required arguments must be implemented in the final product, but the passing of the arguments for the creation of the object is not required. The type `T` or `<T>` is a generic type that is set for every analysis when that request is made.

Required	**Type**	**Name**	**Content**
Yes	`List<Path>`	`folders`	The folders that should be scanned.
No	`List<Path>`	`excludedFolders`	Subfolders of `folders`[1] that don't need to be scanned.
No	`Predicate<Path>`	`isFileRelevant`	Executable function to verify whether the file is relevant[2] for the analysis.

Table 1: Library class arguments.

The instance shall only scan for files as long as active requests are being processed. This means that the scan should not start as soon as the class is instantiated. If the last request has been completed the balancer shall go into a sleeping state and woken up when a new request arrives.

Every request to the API must be sent to an existing instance. That means instantiating the object before starting any analysis is critical. Once the object exists the user must be able to call the API with the following arguments and receive a response of type `List<T>`.

[1] This must be verified when the class instance is created.

[2] This f.e. allows for the exclusion of irrelevant files that may be stored in the same location. If no function is passed every file must be treated as relevant.

Required	Type	Name	Content
Yes	`Function<Path, T>`	`analyzer`	The analysis function applied to every relevant[2] file.
No	`List<Path>`	`excludedFolders`	Subfolders of `folders`[1] that don't need to be scanned.
No	`Predicate<Path>`	`isFileRelevant`	Executable function to verify whether the file is relevant[2] for the analysis.

Table 2: Library analysis call arguments.

3 Functional and Non-Functional Features

The contractor shall implement the following functional and non-functional features.

3.1 File Tree

To talk about unpredictability, the term of file tree and how the file tree must be defined, is crucial. The file tree has a root, which is the folder that is passed to the library on initialization.
If multiple folders were passed on initialization the root is an imaginary folder that is above all folders specified. Any files and folders at the same depth level in the tree are to be sorted in ascending lexicographical order according to their Unicode codepoint value.

3.2 Handling Unpredictability

This library must be able to work in an environment where files behave totally unpredictablly, due to external system or user interaction. The following possibilities and their respective appropriate responses are defined here.

To handle unpredictability the term "rotation" needs to be defined. A rotation from any point has a starting file. A complete rotation goes from the starting file (inclusive) to the last file still in the file tree and then from the first file in the next file tree scan to the last file before the original starting file.

A file disappears/is deleted mid-read

The user should not notice anything about this. Meaning that any exceptions get caught and the file is not analyzed.

A file disappears/is deleted mid-analysis

This case does not include the possibility that a file disappears mid-analysis. If a file disappears during the analysisthe analysis should keep going undisturbed. We are interested

about the results of the analysis at the time of read, not what it might be at the end of the analysis.

All files (and folders) are deleted

If the read operation is interrupted by a file disappearing, that file is ignored as stated in the case of "A file disappears/is deleted mid-read" and the next file relevant for analysis is searched. If there are no further files to analyze the current result is returned.

All files (and folders) are replaced by a completely new set of files and folders

As stated in the paragraph "A file disappears/is deleted mid-read" the current file is ignored for analysis if a read operation is interrupted.

The file where the analysis of a certain request started isn't around anymore on completion of one rotation[4]

If the starting file cannot be found, reaching any parent folder[3] means reaching the end of a rotation in this case. Parent folders include parent folders at any depth in the tree in upward direction (considering the root of the file system to be at the top). Folders further away from the root folder take precedence.

If no parent folder can be found, all analysis data from the previous traverse of the file tree is discarded. The analysis then starts anew from the first file of the file tree and ends with the last file in the file tree.
If the starting file was the first file in the file tree and it can't be found once the rest of the file tree was traversed the rotation is considered completed.

3.3 Logging

The following log-levels are the only ones allowed in this library. They are a subset of the log levels specified in the enum `java.util.logging.Level` and must be used in the following order (most granular first):

1. Fine
2. Info
3. Warning
4. Severe

Every single log message must include the date and time (including milliseconds) in the ISO 8601 format. The following actions/events must be logged at the specified log level.

[3] This includes parent folders at any depth. Closer parent folders take precendence.

Event	Metadata	Loglevel
Request received	unique request identifier, request source, starting file	Info
File is visited (does not necessarily mean that it's read)	count of analyses for which this file is relevant	Fine
File disappears mid-read	file path, count of analyses for which this file is relevant	Info
File tree traverse started	first file path, count of active analyses in queue	Info
File tree traverse ended	last file path, count of active analyses in queue	Info
Analysis data of a request is discarded	unique request identifier	Info
No files in file tree		Warning

Table 3: Actions that must be logged including the metadata specified in their respective log-level.

3.4 Code Style

All code must adhere to the Google Java Style Guide.

3.5 Testing

The library must be tested with, at the bare minimum, an automated test for every paragraph in section 3.2 and every log event in 3.3. The logs to test section 3.3 need not arise from a functionality of the code, but may be tested by faking log statements.

3.6 Documentation

No additional documentation – outside of the code, that is – must be provided. The code must be documented using JavaDoc or the relevant equivalent in the implementation language. Every part of the public interface must be documented.

- Classes
  - Interfaces
  - Enums
- Functions

Each documented artifact must contain the following information for

classes[4]:
- Short description

functions:
- Short description
- Parameters using the `@param` tag

[4]Do not document anonymous inner classes.

- Return type using the `@return` tag [5]
- Exceptions using the `@exception` tag [6]
-

Tags must be ordered as specified in the section Order of Tags in How To Write Doc Comments for the Javadoc Tool

[5] if not `void`
[6] as specified in Documenting Exceptions with @throws Tag

4 Glossary

Term	Definition
Singleton	A class that can only exist once in memory. Trying to create a new instance of the class should return the existing instance.
Rotation	A single go around the list of files considered for analysis
Starting file	The file an analysis starts with.

Table 4: Glossary definitions.

List of Figures

List of Tables