はじめに

本書の内容

本書は、ファイルシステムイベント監視機能についての本です。きっと皆さんも何らかの形でお使いであろうファイルシステムイベント監視機能について、どのような方法で行われているかについて整理する事を目的としています。

あまり知られていないであろうファイルシステムイベント監視機能について、本書がその理解の一助となれば幸いです。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。また、本書に記載されいてる情報は、執筆時点でのものです。時間経過によって情報が古くなっている可能性がありますのであらかじめご了承ください。

第 1 章

ファイルシステムイベントとは

日頃、「ファイルが更新や追加されたら何か処理をする」ということは常日頃発生します。例えば、セキュリティソフト。セキュリティソフトは、ファイルが追加、更新されたら、そのファイルが問題無いか即座にチェックする必要があります。他にも、Dropbox などのファイル同期・バックアップソフトもそうです。ファイルが変更されたら、即座に同期する事を多くのユーザは期待しているでしょう。

このように、ファイルの変更された場合に何か処理をする、というニーズは多くあります。この処理は、多くの場合、「ファイルシステムイベント」という機能を使用し実装されています。しかし、この処理がどのように実装されているかについては知らない方もいらっしゃるかと思います。この本は、その「ファイルシステムイベント」について説明した本です。

1.1 実装方式

ファイルの変更監視の実装方式は、大きく、ポーリングによる実装と、OS が提供している API を使用した実装の 2 つの実装があります。それぞれについて見ていきましょう。

ポーリング

特別な API などは使わず、スレッドなどを用いて、監視したいファイルやディレクトリに対して定期的に変更があったかを確認する方式です。

ポーリングのメリットとして、実行される環境にか依存せず、同じ機能が提供

出来る、という事が挙げられます。後述する OS が提供している API を使用した場合だと、OS 毎に提供されている機能が事なり、OS 間で挙動の差異が発生してしまいます。ポーリングだとそのような事がありません。

デメリットとして、変更が正確に把握することが難しい事が挙げられます。例えば、ファイルの更新があったかどうかの間に、ファイルの更新と削除が連続して行われた場合、更新の変更に気付くのが難しくなってしまいます。また、監視するファイルが数ファイルであればどうにかなるかもしれませんが、監視するファイルが複数ディレクトリに跨り数百あるような場合、更新があったかのチェックをするだけでもリソースを消費してしまい、負荷が掛ってしまう可能性があります。

このような理由により、現在は、OS が提供している API を利用するのが一般的で、何らかの理由によりそれらの API が使用出来ないときの為のセーフティネットとしてポーリングの実装を提供、という形を取るライブラリが多いようです。もし自力でファイル変更監視を行うライブラリの実装を行うような場合は、余程特殊な事情が無い限り、後述する OS が提供する API を使用した実装を行うと良いでしょう。

OS が提供している API

さて、先に説明した通り、ポーリングではファイル変更監視を正確に行うのは難しいです。しかし、昨今の OS は、OS 側でファイル変更監視の為の仕組みが提供されています。一般的にそれらの仕組みは、監視したファイルやディレクトリを API で指定->指定したファイルやディレクトリで変更があったら変更内容を取得、という事が出来るようになっています。

残念ながら、これらの API は標準化はされておらず、OS 毎に提供されている機能に差異があります。例えば、macOS ではディレクトリを再帰的に監視する為の API が提供されていますが、Linux には同等の API がありません。そのため、OS が提供している API をそのまま使用した場合、提供する機能に差異が発生してしまいます。

それらの差異をどう対処しているかは API を実装しているライブラリ毎に異なります。ライブラリについての話の前に、まず OS が提供している機能について見ていきましょう。

第 2 章

OS 毎の話

2.1 Linux

まずは Linux について見ていきたいと思います。

dnotify

Linux に初めて導入されたファイルシステムイベント監視の仕組みは、dnotify という機能で Linux 2.4.0 で導入されました。dnotify はディレクトリ自体、もしくはその内部のファイルに変更がかかったときに、ユーザアプリケーションにその変更を通知する事を目的として提供されていました[*1]。変更の通知はシグナル送信により行われます。

最初に提供された API という事もあり、dnotify には、

・個別のファイルの監視が行えない。

・通知される情報が少ない (シグナル番号とファイルディスクリプタのみ)。

・監視するディレクトリ毎にファイルディスクリプタをオープンする必要がある。

・通知方法にシグナルを使用しているが、シグナルが非同期で送信されるため、競合状態やエラーが発生しやすかった。また、デフォルトで SIGIO を使用していたが、SIGIO はキューイングされないので、通知が欠落する事があった。

等々の問題がありました。そのため、後述する inotify に置き換えられ、現在は使用されなくなっています。

[*1] dnotify の先頭の"d"は、"directory"の"d"らしいです。

inotify

前述した dnotify の問題を解決する為に登場したのが inotify[*2]です。inotify は Linux 2.6.13 で導入されました。

inotify は、dnotify の問題点を大幅に改善しました。個別のファイルを監視できるようになり、通知される情報も豊富になりました。具体的には、どのファイルで何の変更が発生したかを詳細に知ることができるようになっています。

inotify は inode[*3]ベースの通知であり、ディレクトリ単位での再帰的な自動監視は行いません。再帰的に監視したい場合は、利用する側がディレクトリツリーを辿ってそれぞれを監視対象に追加する必要があります。

fanotify

inotify がリリースされた後、さらに高度なファイルシステム監視機能として、fanotify[*4]が Linux 2.6.36 で導入されました。

inotify は、イベント発生元の情報 (プロセス ID やユーザ ID) がとれない、ゲートキーピング機能<@gatekeeping>{gatekeeping}が無い、NFS のようなリモートファイルシステムのイベントを取得出来ない、等の問題あります。

これは、アンチウイルスソフトのようなセキュリティ関連のアプリケーションで問題になっていたそうです。そこで生まれたのが fanotify です。fanotify は、inotify とは異なるアプローチを取っています。inotify が特定のファイルやディレクトリを監視するのに対し、fanotify はファイルシステム全体を監視することができます。また、先に記載した問題も解消されています。

ただし、fanotify を使用するには管理者権限（CAP_SYS_ADMIN）が必要です。そのため、特別な要件が無い一般的なアプリケーションでは、引き続き inofity が使われる事が多いです。

[*2] inode-based file event notifications の略。

[*3] Unix 系ファイルシステムで使われているデータ構造。inode はファイルの実体（データブロック）や属性（パーミッション、所有者、タイムスタンプなど）への参照を保持し、ファイル名はその inode への参照（ディレクトリエントリ）として扱われます。ファイルシステム内で一意の番号 (inode 番号) を持っています。

[*4] fscking all notification の略。

2.2 *BSD

次に、BSD 系 OS について見ていきましょう。

kqueue

BSD 自体にはファイルシステム監視機能はありませんでした。しかし、子孫である FreeBSD には、4.1 から kqueue[*5]という機能が追加されました。

kqueue は、元々 I は/O 多重化（複数のファイルディスクリプタの状態変化を効率的に監視する）を目的として設計されましたが、ファイルシステムイベントやプロセス、シグナル、タイマーなど、様々な種類のイベントを統一的に扱えるようになっています。

ファイルやディレクトリの変更（作成、削除、リネーム、属性変更など）を検知できるようになっていますが、inotify 同様、サブディレクトリやファイルを自動で再帰的に監視する機能はありません。

kqueue はその後、NetBSD、OpenBSD 及び DragonFly BSD でもサポートされ、BSD 系ではデファクトになりました。

2.3 macOS

続いて macOS について見ていきましょう。macOS が使用している Darwin という OS が BSD をベースにしており、他の BSD 同様に、kqueue がサポートされています。それに加えて、macOS では、FSEvents[*6]という機能も使えるようになっています。次のセクションで詳細を見ていきましょう。

FSEvents

FSEvents は、macOS で提供されるファイルシステムイベント通知の仕組みです。FSEvents はディレクトリを単位として変更を記録・通知する設計になっており、ディレクトリ階層に対して再帰的に監視が可能です。ディレクトリを単位としているため、ファイル単位の細かなイベントは提供しないようになってい

[*5] Kernel Queues の略。
[*6] File System Events の略。

ます。また、変更をバッファし、ある程度まとめて配信するようになっています。そのため、細かなファイルの変更を検知して向かない仕様になっています。変わりに、ディレクトリ階層に対しては kqueue よりも効率的に動作するようになっており、Time Machine のようなファイルシステム全体に対して処理を行う、というような用途に適しています。

2.4 Windows

Windows には USN ジャーナルというボリュームに加えられた変更の記録を維持する為の機能があります。この機能を使用して、変更があったファイル

第 3 章

様々なライブラリたち

本章では、ファイル

3.1 Node.js

Node.js は言語本体でファイル変更監視の為の API が提供されています。提供されているのは `fs.watchFile` と `fs.watch` という 2 つの API です。

リスト 2.1: fs.watchFile

```
fs.watchFile(filename[, options], listener)
```

リスト 2.2: fs.watch

```
fs.watch(filename[, options][, listener])
```

引数はどちらも同じですうね。引数だけ見ると違いがよくわからないのですが、`fs.watchFile` の方はポーリングによる実装になっており、指定出来るのはファイルだけになっています。対して、`fs.watch` は OS が提供している API を使用した実装となっており、こちらはファイル・ディレクトリの両方が指定出来るようになっています。

当然 OS が提供している API を使用した方が情報が正確に取得出来る為、公

式のドキュメントにも、可能な限り、`fs.watch` の方を使用する事が推奨されています[*1]

さて、そんな `fs.watch` ですが、OS 毎の API の差異についてはどのように対応しているのでしょうか? `fs.watch` は API の差異の吸収は行わず、OS が提供している API をそのまま使用する、という方針を取っているようです。`fs.watch` のドキュメントの注意事項にも、"The fs.watch API is not 100% consistent across platforms, and is unavailable in some situations."との記載があります。

具体的な例を見ていきましょう。`fs.watch`

表 3.1: 使用している機能

OS	**使用している** API
Linux	inotify
macOS	kqueue、FSEvents
Windows	ReadDirectoryChangesW

3.2 Go

3.3 Rust

[notify-rs/notify](https://github.com/notify-rs/notify)

[*1] https://nodejs.org/docs/latest/api/fs.html#fs_fs_watchfile_filename_options_listener

3.4 Ruby

3.5 Python

3.6 PHP

3.7 プログラミング言語に依存しない実装

libuv

[libuv/libuv/blob/master/docs/src/guide/filesystem.rst](https://github.com/libuv/libuv/blob/

watchman

ファイルの変更監視についてのほん

2020 年 12 月 26 日　初版第 1 刷　発行

著　者　y-yagi