ファイルサーバーに関する機能の作成に対して取り組んだ内容

このシステムでは、データ量が大きくなりがちなバイナリデータに関しては、システムの外部にファイルサーバーを設け、そちらに保存できるように設計しています。

この資料の目的は、ファイルサーバーの選定の意図や、発生したトラブルに対して対処した内容など、開発中の工夫などをプレゼンすることです。

目次

採用したファイルサーバー

選定したのは、SAMBAとなります。

理由としては以下のようになります。

- ファイルサーバーの中では一番オーソドックスな物であり、情報を仕入れやすかった為。

ネット上でも使用方法などが多く出回っている他、書籍でも詳しい情報を得やすい分、開発速度を早めやすくなりました。

- このシステムの開発環境である「Docker」で、このSAMBAを使用したイメージが多く出回っており開発環境の構築が容易だったため。

本来「Docker」でインフラを構築するには、Docker専用の設定ファイルでシェルスクリプトを記述して作成する必要がありますが、「DockerHub」というリポジトリであらかじめ作成されたイメージ（第三者が作成して公開しているシェルスクリプト）があった為、環境構築が簡単に済みました。

- Javaのライブラリに、「jcifs-ng」というSAMBAへの接続用のライブラリが存在し、JavaとSAMBAとの接続を容易に行えるため。

一応ライブラリを用いなくても、Javaの標準の機能でアクセス先のURLを組み立てて接続は行えるのですが、作業効率を上げるために採用しました。

そもそも、なぜファイルサーバーを採用したのか

一番に、自身のスキル向上や、学習の為。

データベースとファイルサーバーは一長一短で、ファイルサーバーにも扱う上での難点があります。両方を用いた開発を行うことで幅広い知識を身につけたかったことが大きな理由です。

Javaを介さずファイルサーバーに直接アクセスしても、保存しているデータを閲覧できる。

データベースには、バイナリデータをバイト配列として保存するので、コンソール上でSQLを入力して直接アクセスしても見ることはできません。ですが、ファイルサーバーであれば、そのままディレクトリにアクセスしてファイルをクリックするだけで中身を確認できます。つまりメンテナンス性が向上します。

データベースのデータの圧迫を抑制することができる。

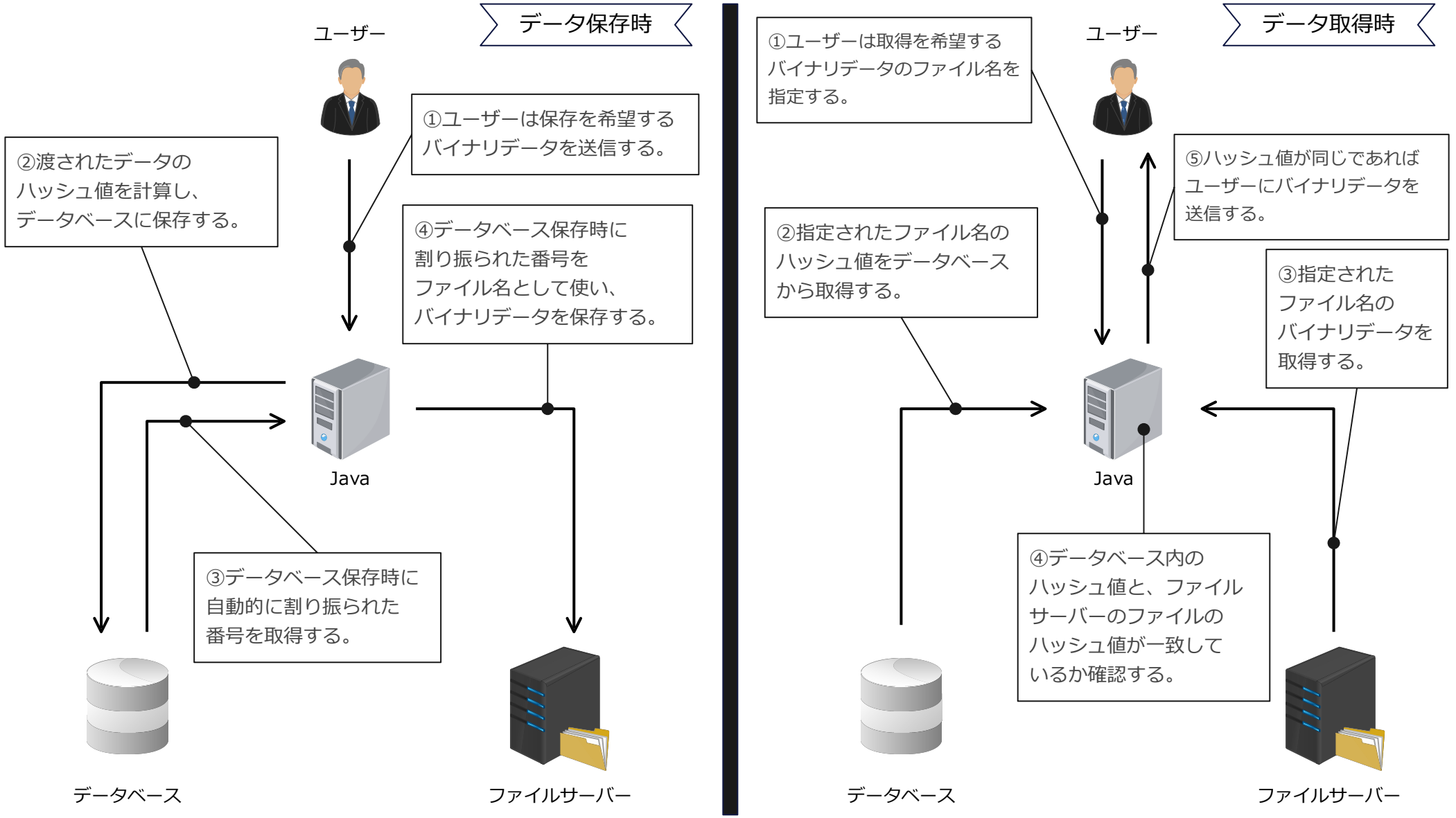
運用するデータベースにはバイナリデータだけでなく他のテキストデータも取り扱うので、バイナリデータによる圧迫によって、他のデータの保存の際に領域が足りないといった問題が発生しやすくなります。
大きなバイナリデータはファイルサーバーに保存することで、重要度が高いデータにデータベースのリソースを割きやすくなります。

アプリケーションに依存せずデータの取り扱いを行える。

データベース保存の場合は、データ取得の際にSQLでのクエリを実行するため、どうしてもSQLを実行するアプリケーションに保存データが依存してしまいがちです。
ファイルサーバーであれば特定のアプリケーションに保存データが依存しない為、今後アプリケーションを増やしてシステムを拡張したいといった際に便利になります。

このシステムでのファイルサーバーの運用方法

このシステムでは、データベースとファイルサーバーをペアで取り扱っています。
以下にデータ保存時とデータ取得時の簡易図を記載します。

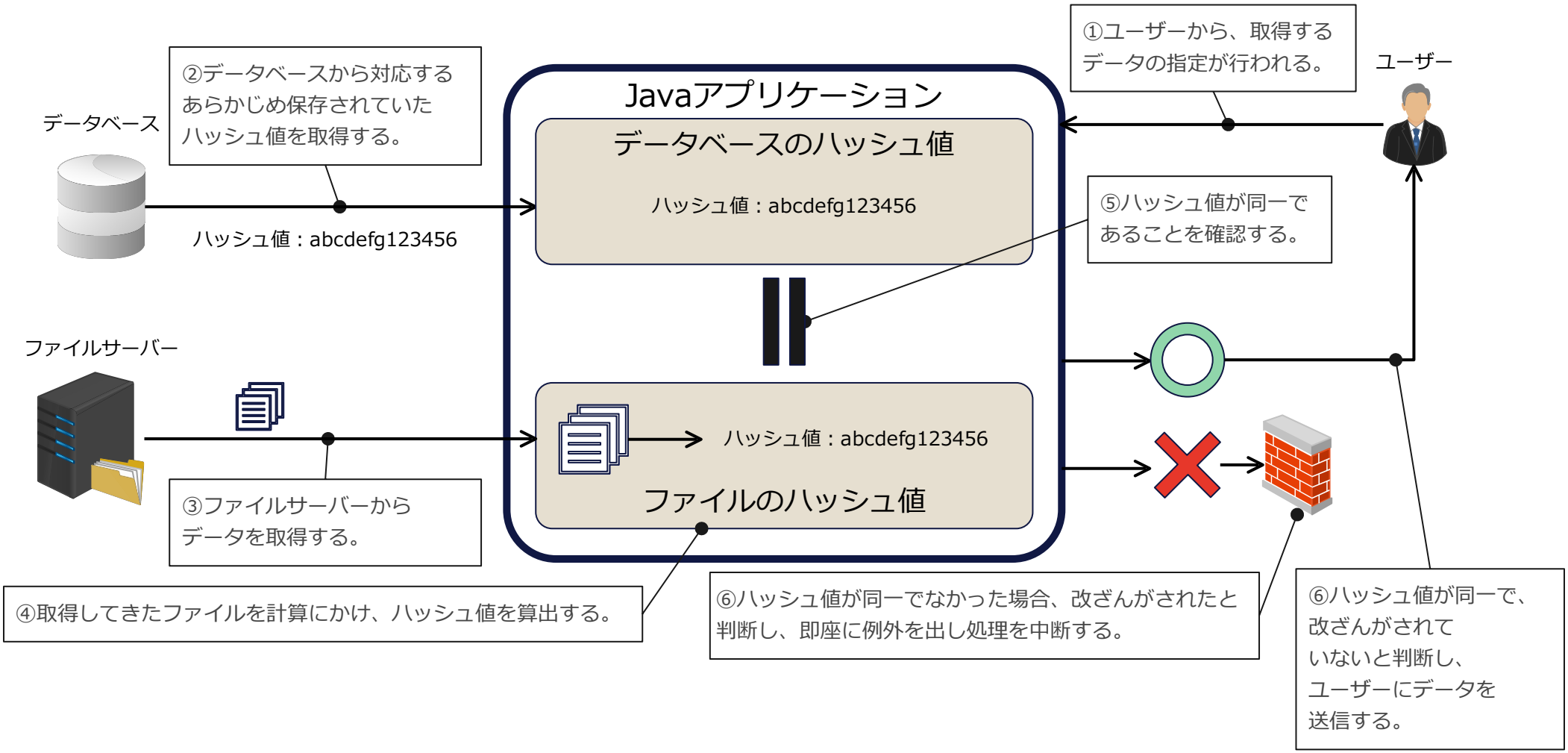


バイナリデータの改ざんの検知

ファイルサーバーでデータを取り扱う際のデメリットとしては、

第三者による改ざんがデータベースと比較すると容易であるという点があります。

この対策として、データベースに保存してあるハッシュ値と、ファイルサーバーから取得してきたデータのハッシュ値を照合し、合致した物のみ取り扱う方式を採用しています。

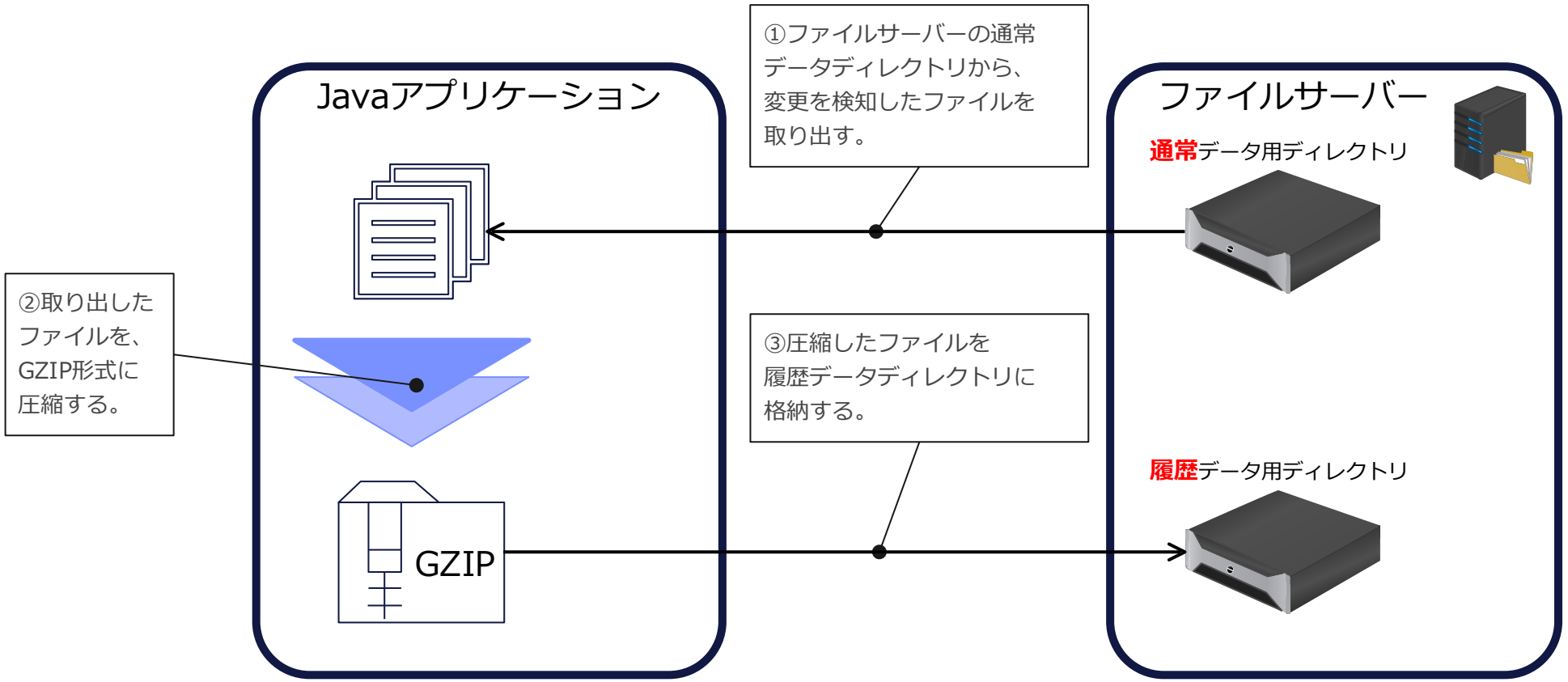


履歴情報の保存の工夫

このシステムでは、変更したデータがあれば例外なく履歴として保存し、復元や操作履歴の追跡が可能になるようにしてあります。

ですが、データ量が大きくなりがちなバイナリデータを、何も加工せずファイルサーバーに保存していては、サーバー内のディレクトリを圧迫してしまいます。

そこで、GZIP形式にデータを圧縮したうえで、ファイルサーバー内に格納し、取り出す際には解凍して扱うようにしております。

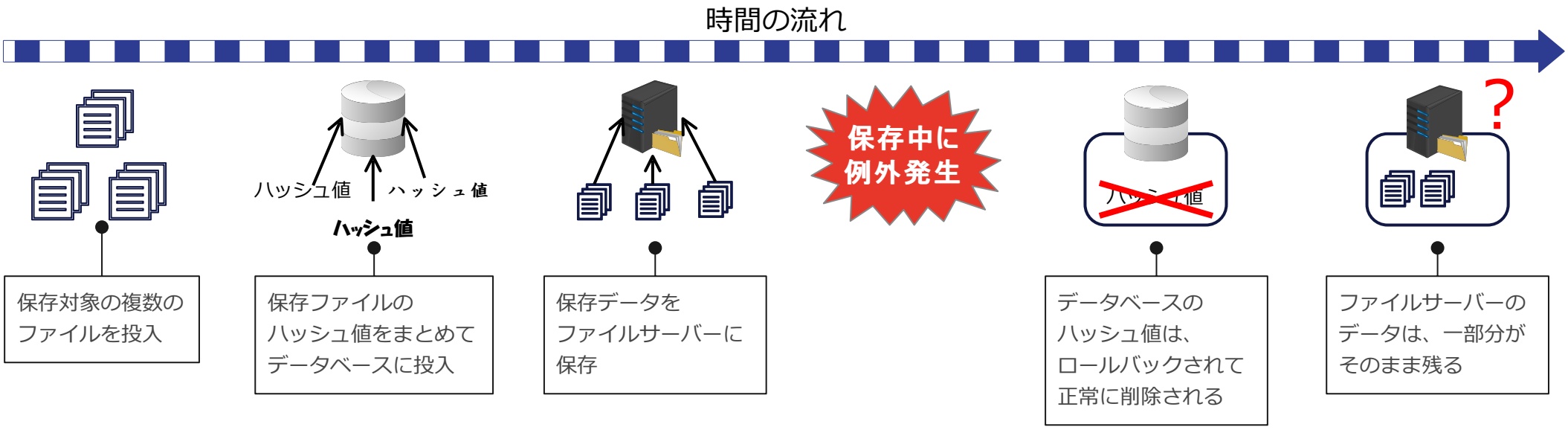


データベースとの整合性の欠点

このシステムに採用している、「SpringFramework」でファイルサーバーとデータベースを対にして扱うには、対応しなければならない欠点があります。

それは、自動トランザクション対象にファイルサーバーが含まれない事です。

「SpringFramework」には、例外を検知すると自動でデータベースやその他諸々の処理を、自動でロールバックしてくれる機能が存在しますが、この機能はファイルサーバーへの更新は対象外になっており、
仮にファイルサーバーへの保存中に例外が発生しても、データベースはロールバックされるが
ファイルサーバーはそのままになり、整合性が保たれない状態になってしまいます。

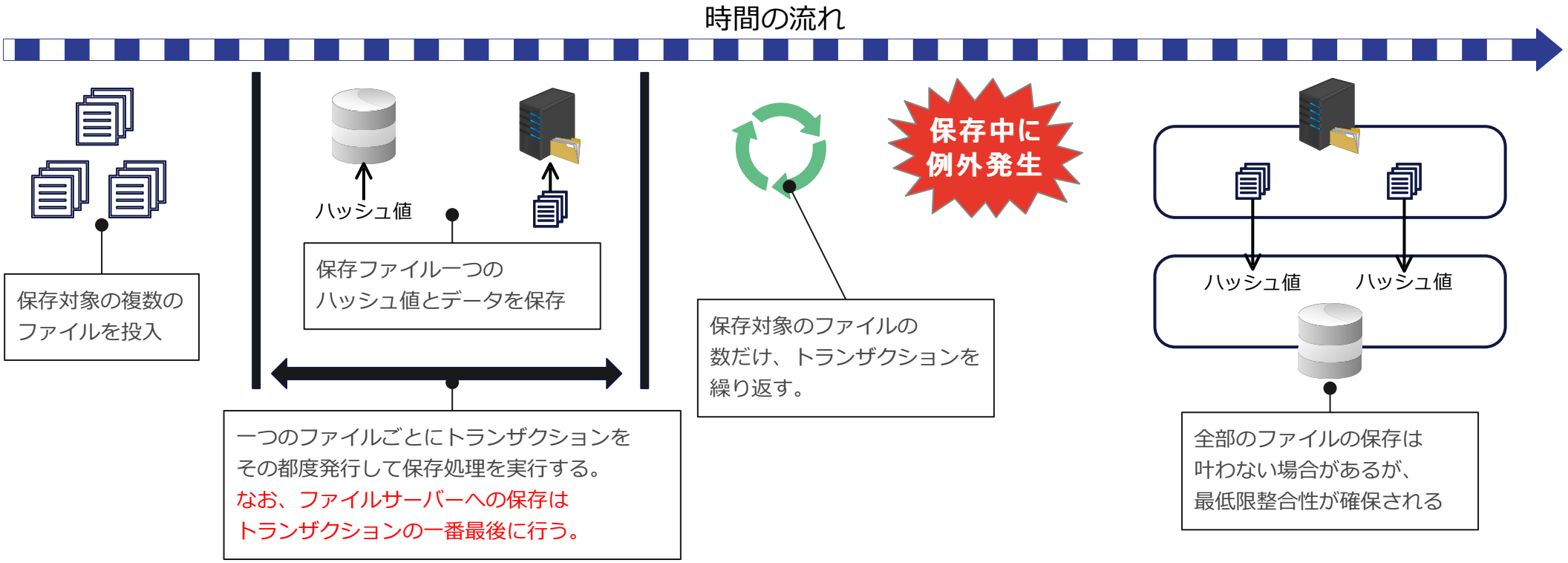


「データベースにはデータがあるのに、ファイルサーバーにはデータが無い」だったり、
「データベースにはデータが無いのに、ファイルサーバーにはデータがある」といった、二種類の
不整合状態が発生する。

ファイルサーバーのトランザクションの工夫

データベースとファイルサーバーの不整合を防ぐ方法としては、
保存ファイル一つづつにトランザクションを発行して処理を行う方法を採用しています。

この方法であれば、複数のファイル保存中に例外が発生した場合、全部のファイルの保存は叶いませんが
最低限、データベースとファイルサーバーの整合性を確保することは可能になります。



なお、全てのファイルの保存が叶わなかった場合に備えて、保存に失敗したファイルの名称をユーザー側に返却して、再処理の手助けになるようにしております。

セキュリティの確保

ファイルサーバーに限らず、ファイルを扱う際に気を付けなくてはいけない点として、
ディレクトリトラバーサル攻撃への対処があります。

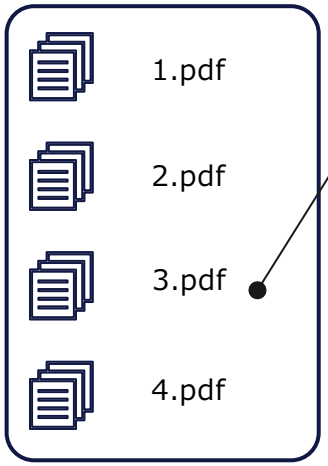
この対策として、ユーザーからのファイルの名称は一切使用せず、システム内で生成した名称でファイルを保存することを徹底しています。

このシステムでは、データベースへのハッシュ値の保存の際に自動的に割り振られた連番を用いて、その番号をファイル名に用いることで、ユーザーからのファイル名は用いないようにしています。

また、ファイルの出力時には元々ついていたファイル名を復元する必要がありますが、ファイル本体とファイル名を切り離して、データベースにファイル名のみを保存することで復元可能かつ、攻撃をできないようにしています。

番号	ファイル名	ハッシュ値
1	ファイル名1.pdf	asdada
2	ファイル名2.pdf	waefgws
3	ファイル名3.pdf	afewfw
4	ファイル名4.pdf	adfwegw

データベースの自動連番機能によって
番号が自動的に振られるので、それを取得。



自動連番の番号をそのまま
ファイル名に用いて保存する。
なお、拡張子もシステム内で
用意した物であり、ユーザーの
物は用いない。