(19)日本国特許庁(JP)

(12)特許公報(B2)

(11)特許番号
特許第7059757号
(P7059757)

(45)発行日　令和4年4月26日(2022.4.26)

(24)登録日　令和4年4月18日(2022.4.18)

(51)国際特許分類

		FI		
Ｇ０６Ｆ	*9/50 (2006.01)*	Ｇ０６Ｆ	9/50	１５０Ｃ
Ｇ０６Ｆ	*9/455(2006.01)*	Ｇ０６Ｆ	9/455	

請求項の数　7　（全36頁）

(21)出願番号　特願2018-68058(P2018-68058)
(22)出願日　平成30年3月30日(2018.3.30)
(65)公開番号　特開2019-179383(P2019-179383A)
(43)公開日　令和1年10月17日(2019.10.17)
審査請求日　令和3年1月13日(2021.1.13)

(73)特許権者　000005223
富士通株式会社
神奈川県川崎市中原区上小田中４丁目１番１号
(74)代理人　100079049
弁理士　中島 淳
(74)代理人　100084995
弁理士　加藤 和詳
(74)代理人　100099025
弁理士　福田 浩志
(72)発明者　木村 功作
神奈川県川崎市中原区上小田中４丁目１番１号 富士通株式会社内
(72)発明者　チョーダリー シュリダル
神奈川県川崎市中原区上小田中４丁目１
最終頁に続く

(54)【発明の名称】　ＡＰＩ処理方法、端末、ＡＰＩ処理プログラム

(57)【特許請求の範囲】
【請求項１】
プログラムにて定義された複数のＡＰＩ呼び出し処理を実行する端末と、
前記複数のＡＰＩ呼び出し処理に応じて、当該ＡＰＩ呼び出し処理に対応して呼び出された関数に対応するプログラムを実行して処理結果を返却する複数のサーバー装置とを有し、
前記端末は、
ＡＰＩ呼び出し処理を順次実行する同期的な記載を含む第一の形式で記載されたプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、ＡＰＩ呼び出し処理を実行状況に応じて並列的に実行可能な非同期的な記載による第二の形式で記載されたプログラムに変換し、
変換された前記第二の形式で記載されたプログラムに応じて非同期的に前記複数のＡＰＩ呼び出し処理を実行し、前記複数のサーバー装置から対応する関数を呼び出して処理する、
ＡＰＩ処理方法。
【請求項２】
前記端末は、
前記第一の形式が前記同期的な記載、及び前記非同期的な記載を含む形式である場合に、
前記第一の形式で記載されたプログラムに含まれる、前記非同期的な記載を、前記同期的な記載に変換した後に、
前記非同期的な記載を変換した後の前記第一の形式で記載されたプログラムを、前記ＡＰ

Ｉ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、前記第二の形式で記載されたプログラムに変換する請求項１に記載のＡＰＩ処理方法。
【請求項３】
前記端末は、
前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、動詞及び名詞に関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する請求項１又は請求項２に記載のＡＰＩ処理方法。
【請求項４】
前記端末は、
前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、リクエストに関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する請求項１又は請求項２に記載のＡＰＩ処理方法。
【請求項５】
前記端末は、
前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、リポジトリに関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する請求項１又は請求項２に記載のＡＰＩ処理方法。
【請求項６】
プログラムにて定義された複数のＡＰＩ呼び出し処理を実行する端末であって、
ＡＰＩ呼び出し処理を順次実行する同期的な記載を含む第一の形式で記載されたプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、ＡＰＩ呼び出し処理を実行状況に応じて並列的に実行可能な非同期的な記載による第二の形式で記載されたプログラムに変換する変換部と、
変換された前記第二の形式で記載されたプログラムに応じて非同期的に前記複数のＡＰＩ呼び出し処理を実行し、当該ＡＰＩ呼び出し処理に対応して呼び出された関数に対応するプログラムを実行して処理結果を返却する複数のサーバー装置から対応する関数を呼び出して処理する実行部と、
を含む端末。
【請求項７】
プログラムにて定義された複数のＡＰＩ呼び出し処理を実行する端末と、
前記複数のＡＰＩ呼び出し処理に応じて、当該ＡＰＩ呼び出し処理に対応して呼び出された関数に対応するプログラムを実行して処理結果を返却する複数のサーバー装置と、に対する処理をコンピュータに実行させるＡＰＩ処理プログラムであって、
前記端末は、
ＡＰＩ呼び出し処理を順次実行する同期的な記載を含む第一の形式で記載されたプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載

を非同期的な記載に変換するための予め定められたルールとに基づいて、ＡＰＩ呼び出し処理を実行状況に応じて並列的に実行可能な非同期的な記載による第二の形式で記載されたプログラムに変換し、
変換された前記第二の形式で記載されたプログラムに応じて非同期的に前記複数のＡＰＩ呼び出し処理を実行し、前記複数のサーバー装置から対応する関数を呼び出して処理する、
処理をコンピュータに実行させるＡＰＩ処理プログラム。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、ＡＰＩ処理方法、端末、ＡＰＩ処理プログラムに関する。
【背景技術】
【０００２】
従来より、プログラムの処理を効率よく実行するための技術が提案されている。
【０００３】
例えば、プログラム中の同期処理を非同期メッセージ、非同期に処理を実行させる命令列およびライブラリ待ち合わせを行わせる命令列を出力して分散配置可能にすると共に複数のＣＰＵに分散配置する技術がある。
【０００４】
また、ソフトウェアのソースコードを変換する技術であって、変換された中間形式を抽象化するステップで記憶された逆変換ルールを用いて、抽象化された中間形式を抽象化される前の中間形式に変換する技術がある。また、この技術では、ソースコードを中間形式に変換するステップで記憶された逆変換ルールを用いて、抽象化される前の中間形式をソースコードである抽象化ソースコードに変換する。
【０００５】
また、ルール定義言語でルールを記述し、翻訳器を介してこれらのルールを渡してランタイムエンジンに送り、ランタイムエンジンを使用して翻訳された命令をスケジュールし、同時に処理する技術がある。
【先行技術文献】
【特許文献】
【０００６】
【文献】特開２００１－１８４３２０号公報
特開２０１３－１２０４９１号公報
特開２００７－５１９０７５号公報
【発明の概要】
【発明が解決しようとする課題】
【０００７】
しかし、ＡＰＩ（Application Programming Interface）の関数を呼び出してプログラムの処理を実行する場合において、プログラムが非効率な記述となっていることがある。例えば、プログラムが、同期的にＡＰＩの関数を呼び出し、順次実行するように記述されていることがある。プログラムが同期的な記述になっていると、非同期に並行して実行してもよい関数がある場合であっても、同期的に順次実行されることになるため、時間効率がよくない、という問題があった。
【０００８】
本発明は、一つの側面として、並列性を考慮して効率よくプログラムを実行することを目的とする。
【課題を解決するための手段】
【０００９】
一つの態様として、第一の形式で記載されたプログラムであって、当該プログラムにて定義された複数のＡＰＩ呼び出し処理を実行する端末を有するＡＰＩ処理方法である。また、前記複数のＡＰＩ呼び出し処理に応じて、当該ＡＰＩ呼び出し処理に対応する処理を実

行するよう第二の形式で記載されたプログラムを動作させる複数のサーバー装置を有するＡＰＩ処理方法である。前記端末は、前記第二の形式で記載されたプログラムに関する情報を取得し、取得したプログラムに関する情報に基づいて前記複数のＡＰＩ呼び出し処理のうち並行して実行可能なＡＰＩ呼び出し処理を特定する。特定されたＡＰＩ呼び出し処理に関する情報をサーバー装置に送信する。

【発明の効果】

【００１０】

一つの側面として、並列性を考慮して効率よくプログラムを実行することができる、という効果を有する。

【図面の簡単な説明】

【００１１】

【図１】非同期関数のコールバックの記述の一例を示す図である。

【図２】Promiseのパターンを用いた非同期関数の記述の一例を示す図である。

【図３】async/awaitのパターンを用いた非同期関数の記述の一例を示す図である。

【図４】本発明の実施形態に係るＡＰＩ処理システムの概略構成を示すブロック図である。

【図５】第１実施形態の端末の概略構成を示すブロック図である。

【図６】変換対象のプログラムであって、同期的な記述、及び非同期的な記述が混在して書かれた形式のプログラムの一例を示す図である。

【図７】プログラムに対する前処理による変換の一例を示す図である。

【図８】プログラムに対する前処理による変換の一例を示す図である。

【図９】プログラムに対する前処理による変換の一例を示す図である。

【図１０】プログラムに対する前処理による変換の一例を示す図である。

【図１１】前処理部の変換により得られた同期的な記述のプログラムの一例を示す図である。

【図１２】関数リストの一例を示す図である。

【図１３】プログラムに対する変換処理の一例を示す図である。

【図１４】プログラムに対する変換処理の一例を示す図である。

【図１５】プログラムに対する変換処理の一例を示す図である。

【図１６】プログラムに対する変換処理の一例を示す図である。

【図１７】プログラムに対する変換処理の一例を示す図である。

【図１８】第１実施形態の端末として機能するコンピュータの概略構成を示すブロック図である。

【図１９】第１実施形態の端末の変換処理の一例を示すフローチャートである。

【図２０】第２実施形態の端末の概略構成を示すブロック図である。

【図２１】表作成部によって作成された表の一例を示す図である。

【図２２】表更新部によって更新された表の一例を示す図である。

【図２３】動詞表の一例を示す図である。

【図２４】表更新部によって更新された表の一例を示す図である。

【図２５】表更新部によって更新された表の一例を示す図である。

【図２６】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。

【図２７】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。

【図２８】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。

【図２９】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。

【図３０】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。

【図３１】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。

【図３２】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。

【図３３】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。

【図３４】第２実施形態の端末として機能するコンピュータの概略構成を示すブロック図である。

【図３５】第２実施形態の端末の変換処理の一例を示すフローチャートである。

【図３６】第３実施形態の端末の概略構成を示すブロック図である。
【図３７】前処理部の変換により得られた同期的な記述のプログラムの一例を示す図である。
【図３８】関数リストの一例を示す図である。
【図３９】表作成部によって作成された表の一例を示す図である。
【図４０】関数ＨＴＴＰリクエスト対応表の一例を示す図である。
【図４１】表更新部によって更新された表の一例を示す図である。
【図４２】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。
【図４３】第３実施形態の端末として機能するコンピュータの概略構成を示すブロック図である。
【図４４】第３実施形態の端末の変換処理の一例を示すフローチャートである。
【図４５】第４実施形態の端末の概略構成を示すブロック図である。
【図４６】前処理部の変換により得られた同期的な記述のプログラムの一例を示す図である。
【図４７】関数リストの一例を示す図である。
【図４８】表作成部によって作成された表の一例を示す図である。
【図４９】管理表作成部の処理の一例を示す図である。
【図５０】並列不可ペア管理表の一例を示す図である。
【図５１】表更新部によって更新された表の一例を示す図である。
【図５２】並列性に基づく実行順序のプログラムへの反映の一例を示す図である。
【図５３】第４実施形態の端末として機能するコンピュータの概略構成を示すブロック図である。
【図５４】第４実施形態の端末の変換処理の一例を示すフローチャートである。
【図５５】第２～第４実施形態の適用先の一例を示す図である。
【発明を実施するための形態】
【００１２】
以下、図面を参照して本発明に係る実施形態の一例を詳細に説明する。
【００１３】
まず、本発明の実施形態の前提となる背景について説明する。
【００１４】
ＡＰＩ（Application Programming Interface）は、ブラックボックス化された有用なリソース（データ，機能など）にプログラムからアクセスするためのインターフェースである。ここでいうＡＰＩは、ローカル環境でライブラリが提供するＡＰＩ、ネットワーク越しにアクセスするためのＷｅｂＡＰＩを含むものとして説明する。
【００１５】
ＡＰＩは、プログラムからは関数やオブジェクトのインスタンスメソッドの呼び出しで実現される。
【００１６】
ＡＰＩをラップするクライアントライブラリの関数呼び出し方式には同期的又は非同期的の二種類があるが、基本は非同期的に記述する必要がある。同期的とは処理を順次実行する方式（ＡＰＩ呼び出し処理を並行して実行しない方式）であり、非同期的とは処理を実行状況に応じて並列的に実行する方式（ＡＰＩ呼び出し処理を並行して実行可能な方式）である。例えば、Node.js（JavaScript（登録商標））のプログラムでは、非同期関数の最後の引数にコールバック関数（Error-first Callbackと呼ばれる）を指定する。例えば、以下のプログラムのようにコールバック関数を指定する。
【００１７】

```
// ファイルを開く
fs.open(‘file.txt’, ‘r’, (err, data) => { … });
```

【００１８】
また、同期的及び非同期的な記述の両方に対応する関数もあるが、あくまで一部である。

例えば下記のプログラムのようにSyncの記述により同期的な記述に対応する。
【００１９】
data = fs.openSync(‘ file.txt ’ , ‘ r ’); // fs.openの同期版
【００２０】
上記のように非同期関数ではコールバックを記述するが、従来よりコールバックの入れ子が続く、いわゆるコールバック地獄が知られている。図１に示すようにコールバックが入れ子に記述されるため、コードの管理が煩雑になるという問題がある。図１では、(err,b)、(err,c)、(err,d)が入れ子に記述されている。
【００２１】
このようなコールバック地獄を緩和する従来手法の例としては、以下の二つのパターン（記述方式）が挙げられる。
【００２２】
従来手法（１）はPromiseのパターンを用いる。Promiseでは、図２の下線部で示すコードのように非同期呼び出しを関数の数珠繋ぎで表すことができる、これにより入れ子が平坦化されコールバック地獄を軽減することができ、非同期呼び出しプログラムの理解及び保守が容易となる。なお、PromiseはECMAScript標準ES6の仕様で利用可能なパターンである。なお、以下、プログラムのコードを記載する図面において、着目するコードの部分には下線を付与しておくものとする。
【００２３】
従来手法（２）はasync/awaitのパターンを用いる。async/awaitでは、図３の下線部に示すようにPromiseを返す非同期関数の呼び出しをあたかも同期関数の呼び出しであるかのように記述することができる。これにより非同期な呼び出しプログラムの理解及び保守が容易となる。なお、async/awaitはECMAScript標準 ES2017の仕様で利用可能なパターンである。
【００２４】
また、上記特許文献１は、逐次処理を非同期メッセージ送受信による分散処理に書き換える技術ではあるものの、複数ＣＰＵを利用する場合の効率化が目的であり、ＡＰＩの呼び出しによる非同期処理の記述を容易にするための技術ではなかった。
【００２５】
従来より、上述した非同期的に記述されたプログラムは、統合開発環境やライブラリ、ＡＰＩクライアントなどのツールを用いて開発していた。統合開発環境は、ＩＤＥ（Integrated Development Environment）と呼ばれ、言語処理系（コンパイラ、スクリプトエンジン等）も含むものである。ライブラリは、非同期処理を書きやすくする表記方法を備えた関数を格納するためのものである。ＡＰＩクライアントは、コールバックのインターフェースを備えたものである。
【００２６】
もっとも、非同期処理の記述は直感的でないため、記述性、可読性、保守性が低いという問題がある。また、開発者が、非同期処理のコードの書き方や、非同期処理に対応した関数を含む非同期処理ライブラリの使い方を習得するのに時間が掛かるという問題がある。
【００２７】
そのため、本発明の実施形態では、同期的な記述のプログラムを非同期的な記述に変換して、非同期処理することが可能な手法を提供する場合を例に説明する。これにより、開発者は、従来から用いているツールをそのまま利用し、平易なプログラミングにより逐次処理的なプログラムを記述して、非同期処理を実現できる。
【００２８】
また、プログラムを実行する前に本手法を適用してプログラムを変換する場合を想定する。変換の際に、並列実行が可能な関数については、並列性を考慮して変換することにより、より時間効率のよいプログラムの実行が可能となる。
【００２９】
以下、上記背景に基づく各実施形態を説明する。なお、以下の各実施形態の説明では、No

de.jsのJavaScript（登録商標）の環境を前提に説明するが、これに限定されるものではなく、同様の手法を同期的、及び非同期的に記述された他のプログラムの環境に対しても適用することも可能である。

【００３０】

［第１実施形態］

本実施形態は、同期的に記述されたプログラムを、非同期的に記述されたプログラムに変換して自動生成し、実行する場合を例として説明する。非同期的に記述されたプログラムに変換することは、並列性を考慮したプログラムに変換するにあたって前提となる手法である。非同期処理のプログラムの変換には、非同期処理ライブラリを用いる。なお、同期的に記述されたプログラムが、第一の形式で記載されたプログラムの一例である。非同期的に記述されたプログラムが、第二の形式で記載されたプログラムの一例である。

【００３１】

以下、変換の概要について説明する。変換では、同期的に記述されたプログラムを入力とする。あるいは同期的記述及び非同期的記述の混在したプログラムを入力とする。

【００３２】

同期的な記述及び非同期的な記述の混在したプログラムを入力とする場合には、入力されたプログラムをパーズ（構文解析）して抽象構文木（ＡＳＴ（Abstract Syntax Tree））を生成し、次に示すような方法で同期的な記述に書き換える。

【００３３】

関数定義ブロックにてコールバック関数定義を引数に含む関数の呼び出し文を深さ優先で探索する。また、コールバック関数定義の第二引数の名前をユニークなものへ変更する。変更は、プリフィクス（_）や通し番号を付ける等である。また、関数呼び出し文の引数からコールバック関数定義を削除する。また、コールバック関数内でコールバック第二引数が操作されている場合、関数呼び出し文を関数呼び出しの戻り値を初期値とするような変数宣言文に変換する。変数宣言文へは例えば以下のように変換する。

【００３４】

foo(a,b,(err,data)=〉{…}) → let _data0 = foo(a,b)

【００３５】

また、変数宣言文の直後にコールバック関数内で通常時に実行されるコード片を挿入する。以上により、同期的記述及び非同期的記述の混在したプログラムが、同期的な記述のプログラムに書き換えられる。

【００３６】

そして、同期的な記述のプログラムを、次に示すような方法で、非同期的な記述に書き換える。

【００３７】

まず、ＡＰＩクライアントライブラリのソースコードから関数名を参照し、ライブラリ関数呼び出し文を特定する。また、ライブラリの公開している関数リストをＡＰＩクライアントライブラリのソースコードから抽出する。また、特定された関数呼び出し文の関数名が、抽出された関数リスト内に含まれ、オブジェクト変数が当該関数のライブラリモジュールそのもの、またはモジュールに由来するものである場合、その文を関数呼び出し文リストに追加する。また、所定の変換ルールを用いて、非同期処理ライブラリを用いた非同期処理プログラムを生成し出力する。

【００３８】

図４に示すように、本実施形態に係るＡＰＩ処理システム１００は、端末１０と、複数のサーバー装置１２とを含み、端末１０と、複数のサーバー装置１２とが、インターネット等のネットワーク３を介して接続される。

【００３９】

端末１０は、プログラムの変換を行い、変換したプログラムを実行する情報処理装置である。

【００４０】

サーバー装置１２は、プログラムで呼び出されるＡＰＩを提供する装置である。サーバー装置１２は、端末１０で実行されたプログラムのＡＰＩの関数呼び出しに応じて適宜、呼び出された関数に対応する必要な処理（呼び出された関数に対応するプログラム）を実行して処理結果を端末１０に返却する。サーバー装置１２で提供されるＡＰＩは、例えば、ＡＷＳ（Amazon Web Services）Ｓ３や、他社ＡＰＩである。なお、端末１０側は、サーバー装置１２から提供されるＡＰＩを呼び出すためのＡＰＩクライアントを導入して、ＡＰＩを呼び出すプログラムを実装する。また、サーバー装置１２の構成は、ＡＰＩの提供態様によって様々であるため、具体的な構成についての説明は省略する。

【００４１】

端末１０は、図５に示すように、機能的には、ＡＰＩクライアントライブラリ１４と、変換ルール１６と、前処理部２２と、関数特定部２４と、変換部２６と、実行部２８と、通信部３０とを含む。

【００４２】

ＡＰＩクライアントライブラリ１４は、ＡＰＩの関数名の一覧を格納したライブラリである。ライブラリにはソースコードが含まれる。

【００４３】

変換ルール１６は、同期的に記述されたプログラムを非同期的な記述に変換するためのルールである。ルールの詳細については後述する。

【００４４】

前処理部２２は、変換対象のプログラムが同期的な記述、及び非同期的な記述が混在して書かれた形式である場合に、前処理としてプログラムを同期的な記述に書き換えるように変換する。前処理が必要なのは、変換部２６で用いる変換ルール１６が、同期的な記述から変換することを前提としているからである。なお、プログラムが同期的な記述のみで書かれた形式である場合には、前処理部２２の処理は省略可能である。

【００４５】

以下、前処理部２２の具体的な書き換え内容について説明する。

【００４６】

図６に変換対象のプログラムの一例を示す。図６のプログラムの処理概要は、file.txtにorgDataの中身を書き込み保存し、ＡＷＳのＳ３バケットに‘file.txt’をキーとしてfile.txtの中身をアップロードする、という内容である。なお、各コールバック第二引数dataは各々の関数ブロック内でのみ有効となる。ここで、同期関数はfs.writeFileSync、fs.readFileSyncである。非同期関数はs3.createBucket、s3.putObjectである。

【００４７】

次に、図７に示すように、関数定義ブロックの中身をtry-catchで囲む。catchブロックにはcatchされたerrのコンソール出力のみ記述する。

【００４８】

次に、図８に示すように、同期関数を同期的記述に変換する。例えば、Node.jsの場合、Object.keys(fs)でＡＰＩクライアントライブラリ１４からモジュールの提供する関数名一覧を取得する。次に、関数名一覧から各関数の同期、及び非同期の対応を特定する。JavaScript（登録商標）の場合、末尾のSyncの有無が同期、非同期の対応とみなせる。次に、ＡＰＩクライアントライブラリ１４のモジュールそのもの、またはモジュール由来のオブジェクト変数について関数名一覧に含まれる関数が呼び出されているコードを探索する。例えば、モジュール、及びモジュール由来のオブジェクト変数は、fs、AWS、s3である。探索されるコードは、fs.writeFileSync、s3.createBucket、fs.readFileSync、s3.putObject、s3.getObjectである。このうち、同期関数の名前を非同期関数のものと同じにする。JavaScript（登録商標）の場合はSyncを削除することで非同期関数のものと同じになる。ここでいう非同期関数のものと同じにすることが同期的な記述への変換である。

【００４９】

次に、図９に示すように、非同期関数を同期的記述に変換する。ここでは、コールバック関数を引数に持つ非同期関数の呼び出し文について以下を実施する。まずコールバック関

数定義を抜き出す。また、コールバック第二引数の名前マングリングを行う。名前マングリングとは、プリフィクスを付けると共に末尾に同一名での通し番号を付けることである。第二引数dataを_data0とする。次に、コールバック第二引数の変数宣言に変換する。例えば、s3.getObject({…})を、(err, data)の第二引数の変数宣言として、（１）let _data0 = s3.getObject({…})と変換する。次に、図１０に示すコードから、err != null（通常時）で実行されるコード片を抽出する。図１０の例では、（２）foo(_data0)、console.log('done')が抽出される。（１）のコードの直後に（２）のコードを配置する。

【００５０】

以上の前処理部２２の処理により、プログラムは、図１１に示すような同期的に記述されたプログラムに変換される。

【００５１】

次に、関数特定部２４の処理を説明する。

【００５２】

関数特定部２４は、前処理部２２で変換されたプログラムから、ＡＰＩクライアントライブラリ１４を参照して関数の呼び出し文を特定し、プログラムでＡＰＩを呼び出している関数名を一覧にした関数リストを作成する。

【００５３】

関数特定部２４は、例えば、Node.jsの場合、Object.keys(fs)でＡＰＩクライアントライブラリ１４からモジュールの提供する関数名一覧を取得する。次に、変換されたプログラムに含まれる、ＡＰＩクライアントライブラリ１４のモジュールそのもの、またはモジュール由来のオブジェクト変数を取得する。取得したモジュールそのもの、またはモジュール由来のオブジェクト変数について、関数名一覧に含まれる関数が呼び出されている関数のリストを特定する。例えば、変換されたプログラムに含まれるモジュール、及びモジュール由来のオブジェクト変数は、fs、AWS、s3である。これらに対して特定される関数のリストは、fs.writeFile、s3.createBucket、fs.readFile、s3.putObject、s3.getObjectである。以上の関数特定部２４の処理によって特定された関数リスト２４Ａを図１２に示す。関数リスト２４Ａには、オブジェクトの変数名と関数名との組がリストとして格納される。

【００５４】

変換部２６は、前処理部２２で同期的な記述に変換したプログラムを、関数特定部２４で特定した関数リスト２４Ａと、変換ルール１６とに基づいて、非同期的に記述されたプログラムに変換する。

【００５５】

以下、変換ルール１６に基づく変換部２６の処理について説明する。関数の呼び出し文については関数リスト２４Ａを参照する。

【００５６】

まず、図１３に示すように、tryブロックの中身を抜き出す。

【００５７】

次に、図１４に示すように、抜き出したブロック中の最初のライブラリ関数呼び出し文をawait new Promiseで囲んだものに置き換える。また、関数呼び出し文の引数にコールバック関数を追加する。関数の中身は第二引数をresolveの引数とし、第一引数が指定されていればrejectの引数とするような定型文とする。また、図１５に別のコードの場合の例を示す。以上の置き換え、追加をブロック内の関数の呼び出し文ごとに繰り返し、全ての関数について処理する。

【００５８】

次に、図１６に示すように、関数定義にasyncを付ける。

【００５９】

以上が変換ルール１６に基づく変換部２６の変換処理である。最終的に、非同期的な記述に変換されたプログラムの例を図１７に示す。

【００６０】

実行部２８は、変換部２６で変換された非同期的な記述のプログラムを実行する。実行部２８は、非同期的に記述されたプログラムの関数呼び出し文に応じて、非同期的に、サーバー装置１２のＡＰＩの関数を呼び出して処理する。

【００６１】

通信部３０は、実行部２８で実行されたプログラムの処理においてＡＰＩの関数の呼び出しによって、ネットワーク３を介してサーバー装置１２のＡＰＩを呼び出す。通信部３０は、サーバー装置１２からＡＰＩの処理結果を取得し、実行部２８に受け渡す。なお、上述した本実施形態の手法では、関数の並列性を考慮したプログラムの変換がされているわけではない。もっとも、非同期的な記述でプログラムは記載されているため、同期的な記述の場合のように全てが逐次実行されるわけではなく、コールバックを用いた関数については並列的な順序によってプログラムは実行されることになる。関数の並列性を考慮し、更に効率良く処理を実行する場合については、第２実施形態以降の各実施形態において説明する。

【００６２】

端末１０は、例えば図１８に示すコンピュータ５０で実現することができる。コンピュータ５０は、Central Processing Unit（ＣＰＵ）５１と、一時記憶領域としてのメモリ５２と、不揮発性の記憶部５３とを備える。また、コンピュータ５０は、インターフェース（Ｉ／Ｆ）５４と、記憶媒体５９に対するデータの読み込み及び書き込みを制御するRead/Write（Ｒ／Ｗ）部５５と、インターネット等のネットワークに接続される通信インターフェース（Ｉ／Ｆ）５６とを備える。ＣＰＵ５１、メモリ５２、記憶部５３、入出力Ｉ／Ｆ５４、Ｒ／Ｗ部５５、及び通信Ｉ／Ｆ５６は、バス５７を介して互いに接続される。なお、通信Ｉ／Ｆ５６は、図５に示す通信部３０として機能する。

【００６３】

記憶部５３は、Hard Disk Drive（ＨＤＤ）、Solid State Drive（ＳＳＤ）、フラッシュメモリ等によって実現できる。記憶媒体としての記憶部５３には、コンピュータ５０を端末１０として機能させるためのＡＰＩ処理プログラム６０が記憶される。ＡＰＩ処理プログラム６０は、前処理プロセス６２と、関数特定プロセス６３と、変換プロセス６４と、実行プロセス６５とを有する。

【００６４】

ＣＰＵ５１は、ＡＰＩ処理プログラム６０を記憶部５３から読み出してメモリ５２に展開し、ＡＰＩ処理プログラム６０が有するプロセスを順次実行する。ＣＰＵ５１は、前処理プロセス６２を実行することで、図５に示す前処理部２２として動作する。また、ＣＰＵ５１は、関数特定プロセス６３を実行することで、図５に示す関数特定部２４として動作する。また、ＣＰＵ５１は、変換プロセス６４を実行することで、図５に示す変換部２６として動作する。また、ＣＰＵ５１は、実行プロセス６５を実行することで、図５に示す実行部２８として動作する。これにより、ＡＰＩ処理プログラム６０を実行したコンピュータ５０が、端末１０として機能することになる。なお、プログラムを実行するＣＰＵ５１はハードウェアである。

【００６５】

なお、ＡＰＩ処理プログラム６０により実現される機能は、例えば半導体集積回路、より詳しくはApplication Specific Integrated Circuit（ＡＳＩＣ）等で実現することも可能である。

【００６６】

次に、本実施形態に係るＡＰＩ処理システム１００の作用について図１９のフローチャートを参照して説明する。

【００６７】

ステップＳ１００で、前処理部２２は、プログラムが同期的な記述、及び非同期的な記述が混在して書かれた形式のプログラムを同期的な記述に書き換えて変換する。

【００６８】

ステップＳ１０２で、関数特定部２４は、前処理部２２で変換されたプログラムから、Ａ

ＰＩクライアントライブラリ１４を参照して関数の呼び出し文を特定し、プログラムでＡＰＩを呼び出している関数名を一覧にした関数リストを作成する。

【００６９】

ステップＳ１０４で、変換部２６は、ステップＳ１００で同期的な記述に変換したプログラムを、ステップＳ１０２で特定した関数リスト２４Ａと、変換ルール１６とに基づいて、非同期的に記述されたプログラムに変換する。

【００７０】

ステップＳ１０６で、実行部２８は、ステップＳ１０４で変換された非同期的な記述のプログラムを実行する。実行部２８は、非同期的に記述されたプログラムの関数呼び出し文に応じて、非同期的に、サーバー装置１２のＡＰＩの関数を呼び出して処理する。

【００７１】

以上説明したように、本実施形態に係るＡＰＩ処理システムによれば、同期的な記述に変換したプログラムを、特定した関数リストと、変換ルールとに基づいて、非同期的に記述されたプログラムに変換する。また、変換された非同期的な記述のプログラムを実行する。このため、非同期的なプログラムの処理の作成及び実行を容易にすることができる。

【００７２】

また、プログラムの開発者は，従来の同期関数、及び非同期関数を用いたプログラムや同期的な記述を用いたプログラムを作成するだけで、非同期処理ライブラリを用いたプログラムを、ライブラリの利用方法やノウハウの習得なしに記述することが可能となる。また、非同期処理が可能なプログラムの開発工数が削減され、プログラムの記述性、可読性、及び保守性が向上する。

【００７３】

［第２実施形態］

次に第２実施形態について説明する。第２実施形態では、関数名に含まれる動詞及び名詞に基づいて並列性に基づく実行順序を反映したプログラムを生成する場合について説明する。なお第１実施形態と同様となる箇所については同一符号を付して説明を省略する。

【００７４】

まず本実施形態の背景を説明する。

【００７５】

第１実施形態は非同期的な記述のプログラムに変換する手法であったが、このままではＡＰＩの関数のうち、何れの関数の組み合わせが、並列化して実行可能な関数の組合せであるか分からない。ＡＰＩにはブラックボックス化されたリソースへの副作用を伴うものがあるため、並列化可能なＡＰＩの組合せには制限がある。副作用とは、例えば、リソースＡを作成してからでないとリソースＡを操作できない、という順序の問題である。しかし、ＡＰＩライブラリの関数のシグネチャからはその組み合わせが容易には分からないようになっている。

【００７６】

そこで本実施形態では、関数名に含まれる動詞及び名詞から並列性の有無を判定し、並列性に基づく実行順序をプログラムに反映する場合を例に説明する。前提として、プログラムの各ステートメント（変数宣言、値更新、削除、等）の実行順序を計算し管理表にまとめる。変数の状態を変えるステートメントの実行順序は変更しない想定である。変数の参照を含むステートメントは、その変数の状態が変化しない区間で並列実行可能である。ＡＰＩに対応する関数呼び出しを含むステートメントを特定したリストを用いて、並列性判定方法によりそれらが並列実行可能かどうかを判定する。そして、並列性を考慮した実行順序を、非同期的な記述のプログラムへの変換に反映する。

【００７７】

次に、関数名に含まれる動詞及び名詞についての概要を説明する。基本的な考え方としては副作用の意味のある特定の動詞が含まれる関数は、それらの関数と別の関数を並列で呼び出した時に常に同じ結果になるとは限らないため並列処理は不可であるとする。また、関数名に頻出する動詞（create,get,put,delete等）と、それらが副作用を生じる意味を

持つかどうか（i.e.,状態を変更するものかどうか）を記録した動詞表を作成しておく。例えば、[(動詞,副作用あり?),(create,true),(get,false),…]といった動詞表である。動詞表を用いて、あるＡＰＩ呼び出しステートメントＡの関数名に副作用を生じる動詞が含まれる場合について考える。Ａの関数名に含まれる名詞と同じ名詞を含む、または同じ変数を引数に指定し、かつプログラム中の出現順が後ろである関数呼び出しを含むステートメントＢの各々は、Ａの後に実行すべきと判定する。

【００７８】

図２０に示すように、ＡＰＩ処理システム２００の端末２１０は、機能的には、ＡＰＩクライアントライブラリ１４と、変換ルール１６と、前処理部２２と、関数特定部２４と、変換部２２６と、実行部２２８と、通信部３０とを含む。端末２１０は、動詞表記憶部２１４と、表作成部２２２と、表更新部２２４とを含む。なお、変換部２２６が、特定部の一例である。実行部２２８が、実行部の一例である。

【００７９】

表作成部２２２には、前処理部２２での変換により得られた同期的な記述のプログラムが入力される。本実施形態では、上記図１１と同様のプログラムが入力された場合を例に説明する。

【００８０】

表作成部２２２は、同期的な記述のプログラムから、実行順序管理表を作成する。表作成部２２２は、まず図２１に示すような、tryブロック中のステートメント（関数を含むコード）を抜き出した表２２２Ａを作成する。表２２２Ａは、「項番」、「ステートメント」、「ＡＰＩに対応？」の項目からなる。「項番」は、「ステートメント」ごとに降順の番号を割り当てる。「ＡＰＩに対応？」の項目には、各ステートメントに関数呼び出し文リストに含まれるオブジェクト変数名、または関数名が含まれるかどうかに基づくブール値として、trueまたはfalseをセットする。

【００８１】

表更新部２２４は、表作成部２２２で作成した表２２２Ａに対し、「変数操作」、「並列性」の項目の列を追加して更新する。まず、表更新部２２４は、「変数操作」の項目として個別の変数操作ごとに追加し、変数の状態を変更するステートメントに「×」、参照するステートメントに「↓」を挿入する。図２２に「変数操作」の項目を追加した表２２４Ａの一例を示す。図２２では変数操作として「data」、「cb_data0」の列を追加し、対応するステートメントの状態を更新している。なお、表更新部２２４の処理では、「ステートメント」、「ＡＰＩに対応？」の項目に対する更新はないため、図面上の記載は省略するものとする。

【００８２】

次に、表更新部２２４は、動詞表記憶部２１４の動詞表２１４Ａを参照し、「並列性」の項目として、副作用のある動詞を含む関数ごとに列を追加する。また、表更新部２２４は、当該関数を呼び出しているステートメントに「×」、当該関数と同じ名詞を含む、または引数として同じ変数を用いている関数を呼び出しているステートメントに「↓」を挿入する。図２３に動詞表２１４Ａの一例を示す。動詞表２１４Ａは動詞に対する副作用の有無をまとめた表である。図２４に「並列性」の項目を追加した表２２４Ｂの一例を示す。図２４では「並列性」の項目の関数として「writeFile」、「createBucket」、「putObject」の列を追加し、対応するステートメントの状態を更新している。なお、動詞表２１４Ａが、動詞及び名詞に関するルールの一例である。

【００８３】

次に、表更新部２２４は、更新した表の冗長な順序を除去する。図２５に示すように、表２２４Ｂの項番５は、「createBucket」に関して、項番４と共に項番２の後に実行すべきとある。それと同時に、「putObject」に関して、項番４の後に実行すべきでもある。そのため、図２５の矢印で示すように順序を特定することができ、項番５の下線付きの「↓」が冗長であるため、これを除去する。

【００８４】

以上の表更新部２２４の更新処理により、並列性に基づく実行順序を特定する。

【００８５】

変換部２２６は、上記の第１実施形態の変換部２６と同様の処理により、同期的な記述のプログラムを非同期的な記述のプログラムに変換する。変換部２２６は、変換した非同期的な記述のプログラムから、表更新部２２４で更新した表２２４Ｂを用いて、並列性に基づく実行順序を反映したプログラムを生成する。なお、変換したプログラムに対して実行順序を反映する場合を例に説明するが、変換と実行順序の反映を同時に処理してもよい。また、並列性に基づく実行順序を反映したプログラムを生成することが、複数のＡＰＩ呼び出し処理のうち並行して実行可能なＡＰＩ呼び出し処理を特定することの一例である。

【００８６】

変換部２２６の具体的な反映処理について以下の処理（１）～処理（４）を説明する。

【００８７】

処理（１）で、まず表２２４Ｂで「×」だけが付いたステートメントを抽出する。抽出された各々のステートメントについて、ステートメントがＡＰＩ呼び出しに対応する場合、Promiseで囲むようにする。図２６に処理（１）でPromiseで囲むように処理したステートメントの一例を示す。

【００８８】

次に処理（２）で、処理（１）にてPromiseで囲んだステートメントが複数ある場合は、図２７に示すようにユニークな名前の配列に各々を格納し、最後にawait Promise.all()で囲むような記述に変換する。図２８に表２２４Ｂで処理（２）の反映対象となる部分を下線にて示す。

【００８９】

次に処理（３）で、１つだけ「↓」の付いているステートメントについて、上記の処理（１）、（２）を実施し、表２２４Ｂで「↓」の付いている列に「×」の付いているステートメントの後続に繋げる。Promise.allされる１つのステートメントに繋げたい場合、図２９に示すように文をawaitするものに置き換え、これらを直列に配置したasync付き関数に変換する。このとき最後の文が変数定義文、代入文である場合、処理（２）と同様に最後の文をreturn文に置き換え後続文の変数参照をawait Promise.allの戻り値に置換する。図３０に表２２４Ｂで処理（３）の反映対象となる部分を下線にて示す。

【００９０】

次に処理（４）で、表２２４Ｂで「↓」が２個以上あるステートメントは、「↓」の付いた行に「×」の付いたステートメントがawait Promise.allされるステートメントの直後に配置する。さらに、Promise.allされた最後のステートメントで定義され、代入される変数を参照している場合、その変数参照をawait Promise.allの戻り値_valuesに置換し、上記処理（１）、（２）を実施する。例えば、図３１に示すように、１番目にpushしたPromiseの戻り値（ここではdata）を参照する場合、変数参照を_values[0]に置換する。図３２に表２２４Ｂで処理（４）の反映対象となる部分を下線にて示す。なお、「×」、「↓」の付いていないステートメントには何も処理しないようにする。

【００９１】

以上の変換部２２６の反映処理により並列性に基づく実行順序を反映したプログラムが生成される。図３３に並列性に基づく実行順序を反映したプログラムの一例を示す。

【００９２】

実行部２２８は、変換部２２６で生成された並列性に基づく実行順序を反映したプログラムを実行する。実行部２２８は、プログラムの実行順序に応じて、並列に実行可能な関数については、サーバー装置１２のＡＰＩの関数を並列に呼び出して処理する。なお、並列に実行可能な関数についてサーバー装置１２のＡＰＩの関数を並列に呼び出して処理することが、特定されたＡＰＩ呼び出し処理に関する情報をサーバー装置に送信することの一例である。

【００９３】

端末２１０は、例えば図３４に示すコンピュータ２５０で実現することができる。コンピ

ュータ２５０は、ＣＰＵ５１と、一時記憶領域としてのメモリ５２と、不揮発性の記憶部２５３とを備える。また、コンピュータ２５０は、入出力Ｉ／Ｆ５４と、記憶媒体５９に対するデータの読み込み及び書き込みを制御するＲ／Ｗ部５５と、インターネット等のネットワークに接続される通信Ｉ／Ｆ５６とを備える。ＣＰＵ５１、メモリ５２、記憶部２５３、入出力Ｉ／Ｆ５４、Ｒ／Ｗ部５５、及び通信Ｉ／Ｆ５６は、バス５７を介して互いに接続される。

【００９４】

記憶部２５３は、ＨＤＤ、ＳＳＤ、フラッシュメモリ等によって実現できる。記憶媒体としての記憶部２５３には、コンピュータ２５０を端末２１０として機能させるためのＡＰＩ処理プログラム２６０が記憶される。ＡＰＩ処理プログラム２６０は、前処理プロセス６２と、関数特定プロセス６３と、表作成プロセス２６２と、表更新プロセス２６３と、変換プロセス２６４と、実行プロセス２６５とを有する。

【００９５】

ＣＰＵ５１は、ＡＰＩ処理プログラム２６０を記憶部２５３から読み出してメモリ５２に展開し、ＡＰＩ処理プログラム２６０が有するプロセスを順次実行する。ＣＰＵ５１は、前処理プロセス６２を実行することで、図２０に示す前処理部２２として動作する。また、ＣＰＵ５１は、関数特定プロセス６３を実行することで、図２０に示す関数特定部２４として動作する。また、ＣＰＵ５１は、表作成プロセス２６２を実行することで、図２０に示す表作成部２２２として動作する。ＣＰＵ５１は、表更新プロセス２６３を実行することで、図２０に示す表更新部２２４として動作する。また、ＣＰＵ５１は、変換プロセス２６４を実行することで、図２０に示す変換部２２６として動作する。また、ＣＰＵ５１は、実行プロセス２６５を実行することで、図２０に示す実行部２２８として動作する。これにより、ＡＰＩ処理プログラム２６０を実行したコンピュータ２５０が、端末２１０として機能することになる。なお、プログラムを実行するＣＰＵ５１はハードウェアである。

【００９６】

なお、ＡＰＩ処理プログラム２６０により実現される機能は、例えば半導体集積回路、より詳しくはＡＳＩＣ等で実現することも可能である。

【００９７】

次に、本実施形態に係るＡＰＩ処理システム２００の作用について図３５のフローチャートを参照して説明する。

【００９８】

ステップＳ２００で、表作成部２２２は、ステップＳ１００での変換により得られた同期的な記述のプログラムから、実行順序管理表（表２２２Ａ）を作成する。

【００９９】

ステップＳ２０２で、表更新部２２４は、動詞表２１４Ａを参照し、表作成部２２２で作成した表２２２Ａに対し、「変数操作」、「並列性」の項目の列を追加して表２２４Ｂとして更新する。

【０１００】

ステップＳ２０４で、変換部２２６は、ステップＳ１０４で変換した非同期的な記述のプログラムから、ステップＳ２０２で更新した表２２４Ｂを用いて、並列性に基づく実行順序を反映したプログラムを生成する。

【０１０１】

ステップＳ２０６で、実行部２２８は、ステップＳ２０４で生成された並列性に基づく実行順序を反映したプログラムを実行する。実行部２２８は、プログラムの実行順序に応じて、並列に実行可能な関数については、サーバー装置１２のＡＰＩの関数を並列に呼び出して処理する。

【０１０２】

なお、第２実施形態の他の構成及び作用については第１実施形態と同様であるため説明を省略する。

【０１０３】
以上説明したように、本実施形態に係るＡＰＩ処理システムによれば、同期的な記述に変換したプログラムを、特定した関数リストと、変換ルールとに基づいて、非同期的に記述されたプログラムに変換する。また、変換された非同期的な記述のプログラムから、動詞表を参照し、並列性に基づく実行順序を反映したプログラムを生成する。実行順序を反映したプログラムを実行する。このため、非同期的なプログラムの処理の作成及び実行を容易にし、かつ、並列性を考慮して効率よくプログラムを実行することができる。
【０１０４】
［第３実施形態］
次に第３実施形態について説明する。第３実施形態では、ＨＴＴＰリクエストのメソッド、及びパスから並列性に基づく実行順序を反映したプログラムを生成する場合について説明する。なお第２実施形態と同様となる箇所については同一符号を付して説明を省略する。
【０１０５】
本実施形態では、ＡＰＩクライアントライブラリ内で実際に発行されるＨＴＴＰリクエストのメソッド、及びパスから並列性の有無を判定し、並列性に基づく実行順序をプログラムに反映する場合を例に説明する。
【０１０６】
基本的な考え方としては、ＨＴＴＰメソッドのうちPOST、PUT、DELETE、PATCHは副作用のあるメソッドであるものとする。副作用がある場合、それらと別のAPIを並列で呼び出した時に常に同じ結果になるとは限らないため並列処理は不可であるとする。また、プログラムが依存しているＡＰＩライブラリのソースコードを用いる。ＡＰＩライブラリの各関数のプログラムを静的に解析し、ＨＴＴＰクライアントを用いてＨＴＴＰリクエストを送るコード片を探索する。発見したコード片から各関数の発行するＨＴＴＰリクエストのメソッド（GET,POST,PUT,DELETE,PATCH等）、パス（"/path/to/resource"等）を特定し、関数ＨＴＴＰリクエスト対応表に記録及び管理する。例えば、[（関数名，メソッド，パス），（getContents, GET,/contents）]といった対応表である。また、関数ＨＴＴＰリクエスト対応表を用いて、あるＡＰＩ呼び出しをするステートメントＡについて、メソッドがPOST,PUT,DELETE,PATCHである場合、Ａのパスと前方一致するパスに対するＨＴＴＰリクエストを発行しかつプログラム中の出現順が後ろである各々のステートメントＢについて、ＢはＡと並列実行できないと判定する。
【０１０７】
図３６に示すように、ＡＰＩ処理システム３００の端末３１０は、機能的には、ＡＰＩクライアントライブラリ１４と、変換ルール１６と、前処理部２２と、関数特定部２４と、変換部２２６と、実行部２２８と、通信部３０とを含む。また、端末３１０は、表作成部２２２と、関数対応表作成部３２２と、表更新部３２４とを含む。
【０１０８】
表作成部２２２には、前処理部２２での変換により得られた同期的な記述のプログラムが入力される。本実施形態では、図３７に示す同期的な記述のプログラムが入力された場合を例に説明する。また、図３８に、図３７のプログラムから関数特定部２４で特定された関数リスト２４Ｂの一例を示す。
【０１０９】
図３９に、表作成部２２２で図３７のプログラムから作成された表２２２Ｂの一例を示す。
【０１１０】
関数対応表作成部３２２は、同期的な記述のプログラムから、ＡＰＩクライアントライブラリ１４の内容に基づいて、関数ＨＴＴＰリクエスト対応表３２２Ａを作成する。なお、関数ＨＴＴＰリクエスト対応表３２２Ａが、リクエストに関するルールの一例である。
【０１１１】
関数対応表作成部３２２は、まずＡＰＩクライアントライブラリ１４のソースコードをパーズして、抽象構文木（ＡＳＴ）を生成する。次に、入力されたプログラムについて、対象となる関数を宣言しているステートメントを探索する。関数宣言内でＨＴＴＰクライア

ントを用いてリクエストを送信している箇所のステートメントを探索する。使用しているＨＴＴＰクライアントの種類に対応した方法によってメソッドとパスを特定し、関数ＨＴＴＰリクエスト対応表３２２Ａに追加する。例えば、Node.jsのrequestライブラリを用いた例では、request.get(‘http://myapi.com/api/users’)のステートメントについて、メソッドが「GET」、パスが「http://myapi.com/api/users」と特定する。図４０に関数ＨＴＴＰリクエスト対応表３２２Ａの一例を示す。関数ＨＴＴＰリクエスト対応表３２２Ａを参照することで、createUserはメソッドが「POST」で、putDataはcreateUserのパスに前方一致することが分かる。これにより、putDataはcreateUserと並列実行できないと判定できる。

【０１１２】

表更新部３２４は、表作成部２２２で作成した表２２２Ｂに対し、関数対応表作成部３２２で作成された関数ＨＴＴＰリクエスト対応表３２２Ａを参照し、「変数操作」、「並列性」の項目の列を追加して更新する。「変数操作」の項目の追加処理については上記第２実施形態の表更新部２２４と同様である。「並列性」の項目については、前方一致するパスごとに列を追加する。図４１に「並列性」の項目を追加した表３２４Ａの一例を示す。図４１では、「並列性」の項目としてパスの固定部分（http://myapi.com/api/）が前方一致した後ろのパスである「/users」、「/contents」を追加し、対応するステートメントの状態を更新している。また、状態の更新では、先に実行するステートメントに「×」、後で実行するステートメントに「↓」を挿入している。

【０１１３】

図４２に、変換部２２６が図４１の表３２４Ａを用いて生成した並列性に基づく実行順序を反映したプログラムの一例を示す。なお、変換部２２６の処理は、上記第２実施形態の表２２４Ｂを表３２４Ａと読み替えて実施すればよい。

【０１１４】

端末３１０は、例えば図４３に示すコンピュータ３５０で実現することができる。コンピュータ３５０は、ＣＰＵ５１と、一時記憶領域としてのメモリ５２と、不揮発性の記憶部３５３とを備える。また、コンピュータ３５０は、入出力Ｉ／Ｆ５４と、記憶媒体５９に対するデータの読み込み及び書き込みを制御するＲ／Ｗ部５５と、インターネット等のネットワークに接続される通信Ｉ／Ｆ５６とを備える。ＣＰＵ５１、メモリ５２、記憶部３５３、入出力Ｉ／Ｆ５４、Ｒ／Ｗ部５５、及び通信Ｉ／Ｆ５６は、バス５７を介して互いに接続される。

【０１１５】

記憶部３５３は、ＨＤＤ、ＳＳＤ、フラッシュメモリ等によって実現できる。記憶媒体としての記憶部３５３には、コンピュータ３５０を端末３１０として機能させるためのＡＰＩ処理プログラム３６０が記憶される。ＡＰＩ処理プログラム３６０は、前処理プロセス６２と、関数特定プロセス６３と、表作成プロセス２６２と、表更新プロセス３６３と、関数対応表作成プロセス３６２と、変換プロセス２６４と、実行プロセス２６５とを有する。

【０１１６】

ＣＰＵ５１は、ＡＰＩ処理プログラム３６０を記憶部３５３から読み出してメモリ５２に展開し、ＡＰＩ処理プログラム３６０が有するプロセスを順次実行する。ＣＰＵ５１は、前処理プロセス６２を実行することで、図３６に示す前処理部２２として動作する。また、ＣＰＵ５１は、関数特定プロセス６３を実行することで、図３６に示す関数特定部２４として動作する。また、ＣＰＵ５１は、表作成プロセス２６２を実行することで、図３６に示す表作成部２２２として動作する。また、ＣＰＵ５１は、関数対応表作成プロセス３６２を実行することで、図３６に示す関数対応表作成部３２２として動作する。ＣＰＵ５１は、表更新プロセス３６３を実行することで、図３６に示す表更新部３２４として動作する。また、ＣＰＵ５１は、変換プロセス２６４を実行することで、図３６に示す変換部２２６として動作する。また、ＣＰＵ５１は、実行プロセス２６５を実行することで、図３６に示す実行部２２８として動作する。これにより、ＡＰＩ処理プログラム３６０を実

行したコンピュータ３５０が、端末３１０として機能することになる。なお、プログラムを実行するＣＰＵ５１はハードウェアである。

【０１１７】

なお、ＡＰＩ処理プログラム３６０により実現される機能は、例えば半導体集積回路、より詳しくはＡＳＩＣ等で実現することも可能である。

【０１１８】

次に、本実施形態に係るＡＰＩ処理システム３００の作用について図４４のフローチャートを参照して説明する。

【０１１９】

ステップＳ３００で、関数対応表作成部３２２は、同期的な記述のプログラムから、ＡＰＩクライアントライブラリ１４の内容に基づいて、関数ＨＴＴＰリクエスト対応表３２２Ａを作成する。

【０１２０】

ステップＳ３０２で、表更新部３２４は、表作成部２２２で作成した表２２２Ｂに対し、関数対応表作成部３２２で作成された関数ＨＴＴＰリクエスト対応表３２２Ａを参照し、「変数操作」、「並列性」の項目の列を追加して表３２４Ａとして更新する。

【０１２１】

なお、第３実施形態の他の構成及び作用については第２実施形態と同様であるため説明を省略する。

【０１２２】

以上説明したように、本実施形態に係るＡＰＩ処理システムによれば、同期的な記述に変換したプログラムを、特定した関数リストと、変換ルールとに基づいて、非同期的に記述されたプログラムに変換する。また、変換された非同期的な記述のプログラムから、関数ＨＴＴＰリクエスト対応表を参照し、並列性に基づく実行順序を反映したプログラムを生成する。実行順序を反映したプログラムを実行する。このため、非同期的なプログラムの処理の作成及び実行を容易にし、かつ、並列性を考慮して効率よくプログラムを実行することができる。

【０１２３】

［第４実施形態］

次に第４実施形態について説明する。第４実施形態では、aws-sdkやGitHub等のソースコードリポジトリの関数呼び出しの出現順から並列性に基づく実行順序を反映したプログラムを生成する場合について説明する。なお第２又は第３実施形態と同様となる箇所については同一符号を付して説明を省略する。

【０１２４】

本実施形態では、ソースコードリポジトリ中で同じＡＰＩライブラリを用いた多数のプログラムを解析し、関数呼び出しの出現順が全て同じかどうかで並列性の有無を判定し、並列性に基づく実行順序をプログラムに反映する場合を例に説明する。

【０１２５】

ソースコードリポジトリからＡＰＩクライアントライブラリを用いているプログラムを抽出して用いる。抽出した全てのプログラムを静的解析し、同じ変数を値の変更無しに少なくとも一つ引数として用いているＡＰＩ呼び出し関数のペア（関数Ａ、及び関数Ｂ）を全て抽出する。ただし関数Ａの方が関数Ｂよりも出現順が先とする。ある関数Ａ、及び関数Ｂについて、抽出したペアの中で関数Ａ、及び関数Ｂの出現するものが全て関数Ａ、及び関数Ｂの順番である場合、関数Ａ、及び関数Ｂに暗黙の呼び出し順序があると判定し、並列不可ペア管理表に記録及び管理しておく。例えば、[（先に呼ぶ関数，後で呼ぶ関数），（関数Ａ，関数Ｂ）]といった管理表である。あるＡＰＩ呼び出しをするステートメントα、ステートメントβについて、各々が呼び出している関数が関数Ａ、及び関数Ｂの場合、並列不可ペア管理表に（関数Ａ，関数Ｂ）が含まれている場合、αとβは並列処理不可であると判定する。

【０１２６】

図４５に示すように、ＡＰＩ処理システム４００の端末４１０は、機能的には、ＡＰＩクライアントライブラリ１４と、変換ルール１６と、前処理部２２と、関数特定部２４と、変換部２２６と、実行部２２８と、通信部３０とを含む。また、端末４１０は、表作成部２２２と、管理表作成部４２２と、表更新部４２４とを含む。また、端末４１０はネットワーク３を介して外部のaws-sdkやGitHub等のソースコードリポジトリ４１４と接続されている。

【０１２７】

表作成部２２２には、前処理部２２での変換により得られた同期的な記述のプログラムが入力される。本実施形態では、図４６に示す同期的な記述のプログラムが入力された場合を例に説明する。また、図４７に、図４６のプログラムから関数特定部２４で特定された関数リスト２４Ｃの一例を示す。

【０１２８】

図４８に、表作成部２２２で図４６のプログラムから作成された表２２２Ｃの一例を示す。

【０１２９】

管理表作成部４２２は、同期的な記述のプログラムから、ソースコードリポジトリ４１４の内容を参照して、並列不可ペア管理表４２２Ａを作成する。なお、並列不可ペア管理表４２２Ａが、リポジトリに関するルールの一例である。

【０１３０】

管理表作成部４２２は、まずソースコードリポジトリ４１４から、図４９に示すように、静的解析によってcreateBucket,putObjectを同じ参照及び値で呼び出しているソースコードを全て抽出する。createBucket,putObjectは表２２２Ｃで同じオブジェクト変数名である。抽出したソースコードのうち、putObjectからcreateBucketの順で呼び出すソースコードが他に存在しないことを検出し、並列不可ペア管理表４２２Ａに追加する。図５０に示すように、並列不可ペア管理表４２２Ａには、先頭にソースコードリポジトリ名及びオブジェクト変数名を付与し、先に呼ぶ関数に「aws-sdk#S3.createBucket」、後で呼ぶ関数に「aws-sdk#S3.putObject」を追加する。なお、上記以外の並列不可ペアは未発見なものとして以下説明する。

【０１３１】

表更新部４２４は、表作成部２２２で作成した表２２２Ｃに対し、管理表作成部４２２で作成された並列不可ペア管理表４２２Ａを参照し、「変数操作」、「並列性」の項目の列を追加して更新する。「変数操作」の項目の追加処理については上記第２実施形態の表更新部２２４と同様である。「並列性」の項目については、ＡＰＩに対応する関数ごとに列を追加する。図５１に「並列性」の項目を追加した表４２４Ａの一例を示す。図５１では、「並列性」の項目としてリポジトリ名及びオブジェクト変数を付与した関数名「aws-sdk#S3.createBucket」、「aws-sdk#DyanmoDB.createTable」、「aws-sdk#S3.putObject」を追加し、対応するステートメントの状態を更新している。また、状態の更新は先に実行するステートメントに「×」、後で実行するステートメントに「↓」を挿入している。並列不可ペアが未発見の関数は、関数呼び出しを行っているステートメントにのみ「×」を挿入している。

【０１３２】

図５２に、図５１の表４２４Ａを用いて生成した並列性に基づく実行順序を反映したプログラムの一例を示す。

【０１３３】

端末４１０は、例えば図５３に示すコンピュータ４５０で実現することができる。コンピュータ４５０は、ＣＰＵ５１と、一時記憶領域としてのメモリ５２と、不揮発性の記憶部４５３とを備える。また、コンピュータ４５０は、入出力Ｉ／Ｆ５４と、記憶媒体５９に対するデータの読み込み及び書き込みを制御するＲ／Ｗ部５５と、インターネット等のネットワークに接続される通信Ｉ／Ｆ５６とを備える。ＣＰＵ５１、メモリ５２、記憶部４５３、入出力Ｉ／Ｆ５４、Ｒ／Ｗ部５５、及び通信Ｉ／Ｆ５６は、バス５７を介して互いに接続される。

【０１３４】
記憶部４５３は、ＨＤＤ、ＳＳＤ、フラッシュメモリ等によって実現できる。記憶媒体としての記憶部４５３には、コンピュータ４５０を端末４１０として機能させるためのＡＰＩ処理プログラム４６０が記憶される。ＡＰＩ処理プログラム４６０は、前処理プロセス６２と、関数特定プロセス６３と、表作成プロセス２６２と、表更新プロセス４６３と、管理表作成プロセス４６２と、変換プロセス２６４と、実行プロセス２６５とを有する。
【０１３５】
ＣＰＵ５１は、ＡＰＩ処理プログラム４６０を記憶部４５３から読み出してメモリ５２に展開し、ＡＰＩ処理プログラム４６０が有するプロセスを順次実行する。ＣＰＵ５１は、前処理プロセス６２を実行することで、図４５に示す前処理部２２として動作する。また、ＣＰＵ５１は、関数特定プロセス６３を実行することで、図４５に示す関数特定部２４として動作する。また、ＣＰＵ５１は、表作成プロセス２６２を実行することで、図４５に示す表作成部２２２として動作する。また、ＣＰＵ５１は、管理表作成プロセス４６２を実行することで、図４５に示す管理表作成部４２２として動作する。ＣＰＵ５１は、表更新プロセス４６３を実行することで、図４５に示す表更新部４２４として動作する。また、ＣＰＵ５１は、変換プロセス２６４を実行することで、図４５に示す変換部２２６として動作する。また、ＣＰＵ５１は、実行プロセス２６５を実行することで、図４５に示す実行部２２８として動作する。これにより、ＡＰＩ処理プログラム４６０を実行したコンピュータ４５０が、端末４１０として機能することになる。なお、プログラムを実行するＣＰＵ５１はハードウェアである。
【０１３６】
なお、ＡＰＩ処理プログラム４６０により実現される機能は、例えば半導体集積回路、より詳しくはＡＳＩＣ等で実現することも可能である。
【０１３７】
次に、本実施形態に係るＡＰＩ処理システム４００の作用について図５４のフローチャートを参照して説明する。
【０１３８】
ステップＳ４００で、管理表作成部４２２は、同期的な記述のプログラムから、ソースコードリポジトリ４１４の内容を参照して、並列不可ペア管理表４２２Ａを作成する。
【０１３９】
ステップＳ４０２で、表更新部３２４は、表作成部２２２で作成した表２２２Ｃに対し、管理表作成部４２２で作成された並列不可ペア管理表４２２Ａを参照し、「変数操作」、「並列性」の項目の列を追加して表４２４Ａとして更新する。
【０１４０】
なお、第４実施形態の他の構成及び作用については第３実施形態と同様であるため説明を省略する。
【０１４１】
以上説明したように、本実施形態に係るＡＰＩ処理システムによれば、同期的な記述に変換したプログラムを、特定した関数リストと、変換ルールとに基づいて、非同期的に記述されたプログラムに変換する。また、変換された非同期的な記述のプログラムから、並列不可ペア管理表を参照し、並列性に基づく実行順序を反映したプログラムを生成する。実行順序を反映したプログラムを実行する。このため、非同期的なプログラムの処理の作成及び実行を容易にし、かつ、並列性を考慮して効率よくプログラムを実行することができる。
【０１４２】
なお、第２～第４実施形態の手法は、図５５に示すように、ＡＰＩの特徴に適した得意な適用先を選んで適用することができる。適用範囲は相互排他的ではなく、複数適用可能なものもある。そのため、第２～第４実施形態の手法を組み合わせて、並列性に基づく実行順序を反映したプログラムを生成するようにしてもよい。
【０１４３】

以上の各実施形態に関し、更に以下の付記を開示する。
【０１４４】
（付記１）
第一の形式で記載されたプログラムであって、当該プログラムにて定義された複数のＡＰＩ呼び出し処理を実行する端末と、
前記複数のＡＰＩ呼び出し処理に応じて、当該ＡＰＩ呼び出し処理に対応する処理を実行するよう第二の形式で記載されたプログラムを動作させる複数のサーバー装置とを有し、
前記端末は、
前記第二の形式で記載されたプログラムに関する情報を取得し、
取得したプログラムに関する情報に基づいて前記複数のＡＰＩ呼び出し処理のうち並行して実行可能なＡＰＩ呼び出し処理を特定し、
特定されたＡＰＩ呼び出し処理に関する情報をサーバー装置に送信する、
ＡＰＩ処理方法。
【０１４５】
（付記２）
前記端末は、
前記第一の形式を、前記ＡＰＩ呼び出し処理を並行して実行しない同期的な記載の形式とし、
前記第二の形式を、前記ＡＰＩ呼び出し処理を並行して実行可能な非同期的な記載の形式とし、
前記第一の形式で記載されたプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、前記第二の形式で記載されたプログラムに変換する付記１に記載のＡＰＩ処理方法。
【０１４６】
（付記３）
前記端末は、
前記第一の形式を、前記ＡＰＩ呼び出し処理を並行して実行しない同期的な記載、及び前記ＡＰＩ呼び出し処理を並行して実行可能な非同期的な記載を含む形式とし、
前記第二の形式を、前記非同期的な記載の形式とし、
前記第一の形式で記載されたプログラムに含まれる、前記非同期的な記載を、前記同期的な記載に変換し、
前記非同期的な記載を、前記同期的な記載に変換したプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、前記第二の形式で記載されたプログラムに変換する付記１に記載のＡＰＩ処理方法。
【０１４７】
（付記４）
前記端末は、
前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、動詞及び名詞に関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記２又は付記３に記載のＡＰＩ処理方法。
【０１４８】
（付記５）
前記端末は、
前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変

換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、リクエストに関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記２又は付記３に記載のＡＰＩ処理方法。

【０１４９】

（付記６）

前記端末は、

前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、リポジトリに関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記２又は付記３に記載のＡＰＩ処理方法。

【０１５０】

（付記７）

第一の形式で記載されたプログラムであって、当該プログラムにて定義された複数のＡＰＩ呼び出し処理を実行する端末であって、

第二の形式で記載されたプログラムに関する情報を取得し、取得したプログラムに関する情報に基づいて前記複数のＡＰＩ呼び出し処理のうち並行して実行可能なＡＰＩ呼び出し処理を特定する特定部と、

特定されたＡＰＩ呼び出し処理に関する情報を、前記複数のＡＰＩ呼び出し処理に応じて、当該ＡＰＩ呼び出し処理に対応する処理を実行するよう前記第二の形式で記載されたプログラムを動作させる複数のサーバー装置に送信する実行部と、

を含む端末。

【０１５１】

（付記８）

前記端末は、

前記第一の形式を、前記ＡＰＩ呼び出し処理を並行して実行しない同期的な記載の形式とし、

前記第二の形式を、前記ＡＰＩ呼び出し処理を並行して実行可能な非同期的な記載の形式とし、

前記実行部は、前記第一の形式で記載されたプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、前記第二の形式で記載されたプログラムに変換する付記７に記載の端末。

【０１５２】

（付記９）

前記端末は、

前記第一の形式を、前記ＡＰＩ呼び出し処理を並行して実行しない同期的な記載、及び前記ＡＰＩ呼び出し処理を並行して実行可能な非同期的な記載を含む形式とし、

前記第二の形式を、前記非同期的な記載の形式とし、

前記第一の形式で記載されたプログラムに含まれる、前記非同期的な記載を、前記同期的な記載に変換し、

前記実行部は、前記非同期的な記載を、前記同期的な記載に変換したプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、前記第二の形式で記載されたプログラムに変換する付記７に記載の端末。

【０１５３】
（付記１０）
前記端末は、
前記実行部は、前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、動詞及び名詞に関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記８又は付記９に記載の端末。
【０１５４】
（付記１１）
前記端末は、
前記実行部は、前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、リクエストに関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記８又は付記９に記載の端末。
【０１５５】
（付記１２）
前記端末は、
前記実行部は、前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、リポジトリに関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記８又は付記９に記載の端末。
【０１５６】
（付記１３）
第一の形式で記載されたプログラムであって、当該プログラムにて定義された複数のＡＰＩ呼び出し処理を実行する端末と、
前記複数のＡＰＩ呼び出し処理に応じて、当該ＡＰＩ呼び出し処理に対応する処理を実行するよう第二の形式で記載されたプログラムを動作させる複数のサーバー装置と、に対する処理をコンピュータに実行させるＡＰＩ処理プログラムであって、
前記端末は、
前記第二の形式で記載されたプログラムに関する情報を取得し、
取得したプログラムに関する情報に基づいて前記複数のＡＰＩ呼び出し処理のうち並行して実行可能なＡＰＩ呼び出し処理を特定し、
特定されたＡＰＩ呼び出し処理に関する情報をサーバー装置に送信する、
処理をコンピュータに実行させるＡＰＩ処理プログラム。
【０１５７】
（付記１４）
前記端末は、
前記第一の形式を、前記ＡＰＩ呼び出し処理を並行して実行しない同期的な記載の形式とし、
前記第二の形式を、前記ＡＰＩ呼び出し処理を並行して実行可能な非同期的な記載の形式とし、

前記第一の形式で記載されたプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、前記第二の形式で記載されたプログラムに変換する付記１３に記載のＡＰＩ処理プログラム。

【０１５８】

（付記１５）

前記端末は、

前記第一の形式を、前記ＡＰＩ呼び出し処理を並行して実行しない同期的な記載、及び前記ＡＰＩ呼び出し処理を並行して実行可能な非同期的な記載を含む形式とし、

前記第二の形式を、前記非同期的な記載の形式とし、

前記第一の形式で記載されたプログラムに含まれる、前記非同期的な記載を、前記同期的な記載に変換し、

前記非同期的な記載を、前記同期的な記載に変換したプログラムを、前記ＡＰＩ呼び出し処理に対応する予め定められた関数のリストと、同期的な記載を非同期的な記載に変換するための予め定められたルールとに基づいて、前記第二の形式で記載されたプログラムに変換する付記１３に記載のＡＰＩ処理プログラム。

【０１５９】

（付記１６）

前記端末は、

前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、動詞及び名詞に関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記２又は付記３に記載のＡＰＩ処理プログラム。

【０１６０】

（付記１７）

前記端末は、

前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、リクエストに関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記２又は付記３に記載のＡＰＩ処理プログラム。

【０１６１】

（付記１８）

前記端末は、

前記第一の形式で記載されたプログラムを、前記第二の形式で記載されたプログラムに変換する際に、前記ＡＰＩ呼び出し処理に含まれる関数に対して予め定められた、リポジトリに関するルールに基づいて、前記第一の形式で記載されたプログラムに含まれる前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理の実行順序を特定し、特定した前記実行順序に基づいて、前記複数のＡＰＩ呼び出し処理のうち、並行して実行可能な処理を並行して実行するように、前記変換に反映する付記２又は付記３に記載のＡＰＩ処理プログラム。

【符号の説明】

【０１６２】

３　　ネットワーク

１０、２１０、３１０、４１０　　端末

１２　サーバー装置
１６　変換ルール
２２　前処理部
２４　関数特定部
２６、２２６　変換部
２８、２２８　実行部
３０　通信部
５０、２５０、３５０、４５０　　コンピュータ
５１　ＣＰＵ
５２　メモリ
５３、２５３、３５３、４５３　　記憶部
６０、２６０、３６０、４６０　　ＡＰＩ処理プログラム
２１４ 動詞表記憶部
２２２ 表作成部
２２４、３２４、４２４　　表更新部
３２２ 関数対応表作成部
４１４ ソースコードリポジトリ
４２２ 管理表作成部

【図面】

【図１】

```
api.fa(a, (err, b) => {
    api.fb(b, (err, c) => {
        api.fc(c, (err, d) => {
            ...
        })
    })
})
```

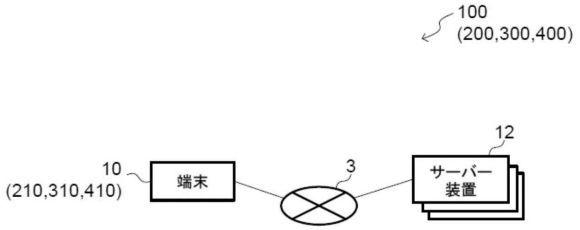
【図２】

```
new Promise((resolve, reject) => {
    api.fa(a, (err, b) => {
        resolve(b)
    })
}).then((b) => {
    return new Promise((resolve, reject) => {
        api.fb(b, (err, c) => {
            resolve(c)
        })
    })
}).then((c) => {
    return new Promise((resolve, reject) => {
        api.fc(c, (err, d) => {
            ...
        })
    })
})
```

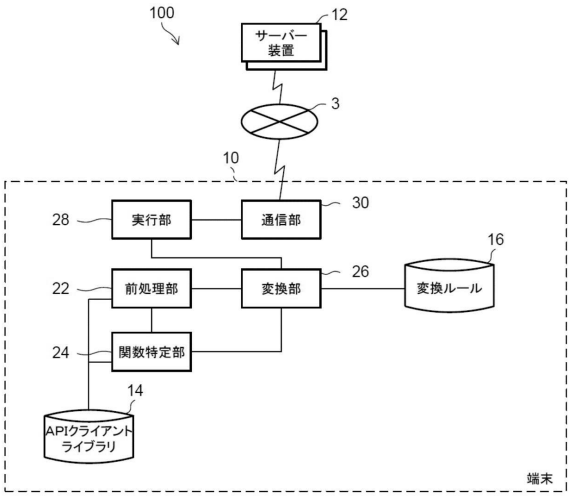
【図３】

```
function fa(a) {
    return new Promise((resolve, reject) => {
        api.fa(a, (err, b) => {
            resolve(b)
        })
})
function fb(b) {
    return new Promise((resolve, reject) => {
        api.fb(b, (err, c) => {
            resolve(c)
        })
})
function fc(c) {
    return new Promise((resolve, reject) => {
        api.fc(c, (err, d) => {
            resolve(d)
        })
})
async function main(a) {
    let b = await fa(a);
    let c = await fb(b);
    let d = await fc(c);
}
```

【図４】



【図５】



【図６】

```
const fs = require('fs')
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
…
function func(filename, bucketName, orgData) {
  fs.writeFileSync(filename, orgData)
  s3.createBucket({ Bucket: bucketName }, (err, result) => {
    if (err) { console.log(err)
    } else {
      const data = fs.readFileSync(filename)
      s3.putObject({ Bucket: bucketName, Key: filename, Body: data }, (err, result) => {
        if (err) { console.log(err)
        } else {
          s3.getObject({ Bucket: bucketName, Key: filename }, (err, data) => {
            if (err) { console.log(err)
            } else {
              foo(data)
              console.log('done')
            }
          })
        }
      })
    }
  })
}
```

【図７】

```
const fs = require('fs')
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
…
function func(filename, bucketName, orgData) {
  try {
    fs.writeFileSync(filename, orgData)
    s3.createBucket({ Bucket: bucketName }, (err, result) => {
      if (err) { console.log(err)
      } else {
        const data = fs.readFileSync(filename)
        s3.putObject({ Bucket: bucketName, Key: filename, Body: data }, (err, result) => {
          if (err) { console.log(err)
          } else {
            s3.getObject({ Bucket: bucketName, Key: filename }, (err, data) => {
              if (err) { console.log(err)
              } else {
                foo(data)
                console.log('done')
              }
            })
          }
        })
      }
    })
  } catch (err) {
    console.log(err)
  }
}
```

【図８】

```
…
try {
    fs.writeFileSync(filename, orgData)
    s3.createBucket({ Bucket: bucketName }, (err, result) => {
      if (err) { console.log(err)
      } else {
        const data = fs.readFileSync(filename)
        s3.putObject({ Bucket: bucketName, Key: filename, Body: data }, (err, result) => {
…
```

【図９】

```
s3.getObject({…}, (err, data) => {
  if (err) {
    console.log(err, err.stack)
  } else {
    foo(data) // dataを用いた何らかの処理
    console.log('done')
  }
})
```

【図１０】

```
(err, _data0) => {
  if (err) {
    console.log(err, err.stack)
  } else {
    foo(_data0)
    console.log('done')
  }
}
```

【図１１】

```
const fs = require('fs')
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
…
function func(filename, bucketName, orgData) {
  try {
    fs.writeFile(filename, orgData)
    s3.createBucket({ Bucket: bucketName })
    const data = fs.readFile(filename)
    s3.putObject({ Bucket: bucketName, Key: filename, Body: data })
    let _data0 = s3.getObject({ Bucket: bucketName, Key: filename })
    foo(_data0)
    console.log('done')
  } catch (err) {
    console.log(err)
  }
}
```

【図１２】

24A

オブジェクト変数名	関数名
fs	writeFile
fs	readFile
s3	createBucket
s3	putObject
s3	getObject

【図１３】

```
try {
   fs.writeFile(filename, orgData)
   s3.createBucket({ Bucket: bucketName })
   const data = fs.readFile(filename)
   s3.putObject({ Bucket: bucketName, Key: filename, Body: data })
   let _data0 = s3.getObject({ Bucket: bucketName, Key: filename })
   foo(_data0)
   console.log('done')
 } catch (err) {
   console.log(err)
 }
```

```
    fs.writeFile(filename, orgData)
    s3.createBucket({ Bucket: bucketName })
    const data = fs.readFile(filename)
    s3.putObject({ Bucket: bucketName, Key: filename, Body: data })
    let _data0 = s3.getObject({ Bucket: bucketName, Key: filename })
    foo(_data0)
    console.log('done')
```

【図１４】

```
    fs.writeFile(filename, orgData)
    s3.createBucket({ Bucket: bucketName })
    const data = fs.readFile(filename)
    s3.putObject({ Bucket: bucketName, Key: filename, Body: data })
    let _data0 = s3.getObject({ Bucket: bucketName, Key: filename })
    foo(_data0)
    console.log('done')
```

```
    await new Promise((resolve, reject) => {
      fs.writeFile(filename, orgData, (err, data) = { if (err) reject(err) else resolve(data) })
    })
    s3.createBucket({ Bucket: bucketName })
    const data = fs.readFile(filename)
    s3.putObject({ Bucket: bucketName, Key: filename, Body: data })
    let _data0 = s3.getObject({ Bucket: bucketName, Key: filename })
    foo(_data0)
    console.log('done')
```

【図１５】

別例:const data = fs.readFile(filename) の場合

```
const data = await new Promise((resolve, reject) => {
  fs.readFile(filename, (err, data) => {
    if (err) reject(err) else resolve(data)
  })
})
```

【図１６】

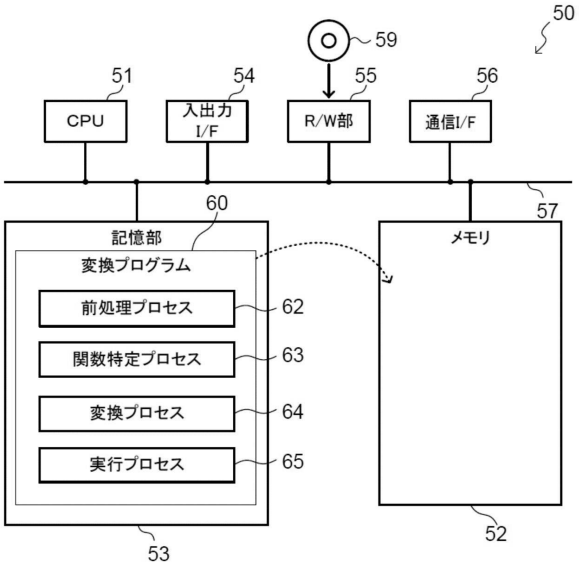
```
function func(filename, bucketName, orgData) {
  try {
    await new Promise((resolve, reject) => {
      fs.writeFile(filename, orgData, (err, data) = { if (err) reject(err) else resolve(data) })
    })
    ...
  } catch (err) {
    console.log(err)
  }
}
```

```
async function func(filename, bucketName, orgData) {
  try {
    await new Promise((resolve, reject) => {
      fs.writeFile(filename, orgData, (err, data) = { if (err) reject(err) else resolve(data) })
    })
    ...
  } catch (err) {
    console.log(err)
  }
}
```

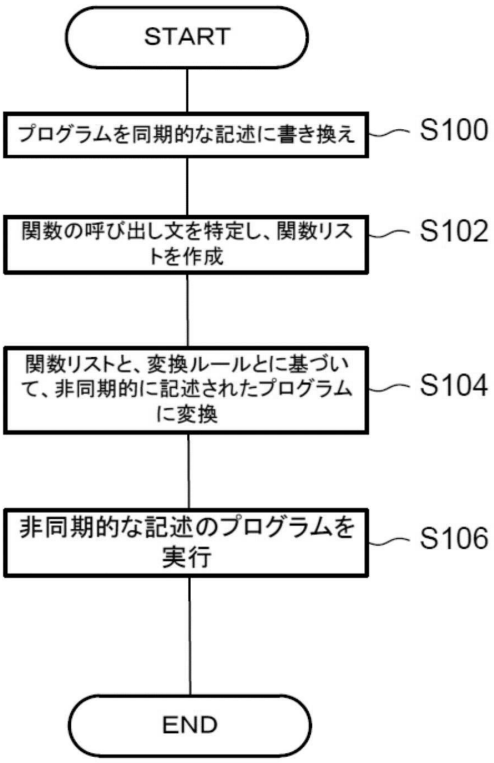
【図１７】

```
const fs = require('fs')
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
…
async function func(filename, bucketName, orgData) {
  try {
    await new Promise((resolve, reject) => {
      fs.writeFile(filename, orgData, (err, data) => {
        if (err) { reject(err) } else { resolve(data) }
      })
    })
    await new Promise((resolve, reject) => {
      s3.createBucket({ Bucket: bucketName }, (err, data) => {
        if (err) { reject(err) } else { resolve(data) }
      })
    })
    const data = await new Promise((resolve, reject) => {
      fs.readFile(filename, (err, data) => {
        if (err) { reject(err) } else { resolve(data) }
      })
    })
    await new Promise((resolve, reject) => {
      s3.putObject({ Bucket: bucketName, Key: filename, Body: data }, (err, data) => {
        if (err) { reject(err) } else { resolve(data) }
      })
    })
    let _data0 = await new Promise((resolve, reject) => {
      s3.getObject({ Bucket: bucketName, Key: filename }, (err, data) => {
        if (err) { reject(err) } else { resolve(data) }
      })
    })
    console.log(_data0)
  } catch (err) {
    console.log(err)
  }
}
```

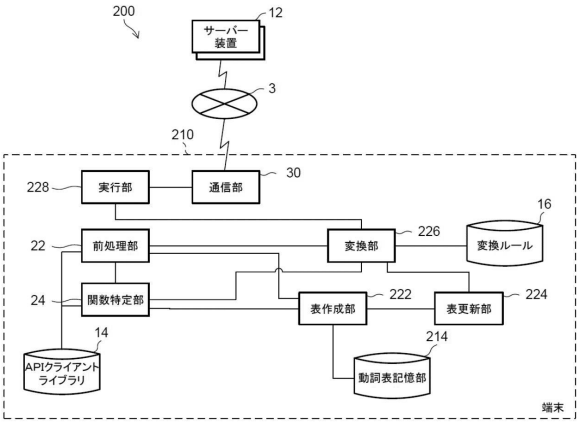
【図１８】



【図１９】



【図２０】



【図２１】

222A

項番	ステートメント(stmt)	APIに対応？
1	fs.writeFile(filename, orgData)	true
2	s3.createBucket({ Bucket: bucketName })	true
3	const data = fs.readFile(filename)	true
4	s3.putObject({ Bucket: bucketName, Key: filename, Body: data })	true
5	let _data0 = s3.getObject({ Bucket: bucketName, Key: filename })	true
6	foo(_data0)	false
7	console.log('done')	false

【図２２】

224A

項番	stmt	stmt, APIに対応？	変数操作	
			data	cb_data0
1				
2				
3			×	
4			↓	
5				×
6				↓
7				

【図２３】

214A

動詞	副作用あり?
create	true
get	false
put	true
read	false
write	true

【図２４】

224B

項番	stmt	stmt, APIに対応？	変数操作		並列性		
			data	_data0	writeFile	createBucket	putObject
1					×		
2						×	
3			×		↓		
4			↓			↓	×
5				×		↓	↓
6				↓			
7							

【図２５】

224B

項番	stmt	stmt, APIに対応？	変数操作		並列性		
			data	_data0	writeFile	createBucket	putObject
1					×		
2						×	
3			×		↓		
4			↓			↓	×
5				×		↓	↓
6				↓			
7							

【図２６】

```
_promises.push(new Promise((resolve, reject) => {
  fs.writeFile(filename, orgData, (err, data) => {
    if (err) reject(err) else resolve(data)
  })
}))
```

```
_promises.push(new Promise((resolve, reject) => {
  s3.createBucket({ Bucket: bucketName }, (err, data) => {
    if (err) reject(err) else resolve(data)
  })
}))
```

【図２７】

```
let _promises = [];
_promises.push(fs.writeFile(filename, orgData));
_promises.push(s3.createBucket({ Bucket: bucketName }));
let _values = await Promise.all(_promises)
```

【図２８】

224B

項番	stmt	stmt, APIに対応？	変数操作		並列性		
			data	_data0	writeFile	createBucket	putObject
1					×		
2						×	
3			×		↓		
4			↓			↓	×
5				×			↓
6				↓			
7							

【図２９】

```
_promises.push(async () => {
  await new Promise((resolve, reject) => {
    fs.writeFile(filename, orgData, (err, data) => {
      if (err) reject(err) else resolve(data)
    })
  })
  return await new Promise((resolve, reject) => {
    fs.readFile(filename, (err, data) => {
      if (err) reject(err) else resolve(data)
    })
  })
})())
```

【図３０】

224B

項番	stmt	stmt, APIに対応？	変数操作		並列性		
			data	_data0	writeFile	createBucket	putObject
3			×		↓		
4			↓			↓	×
5				×			↓
6				↓			
7							

【図３１】

```
let _values = Promise.all(_promises)
await new Promise((resolve, reject) => {
  s3.putObject({ Bucket: bucketName, Key: filename, Body: _values[0] }, (err, data) => {
    if (err) reject(err) else resolve(data)
  })
})
```

【図３２】

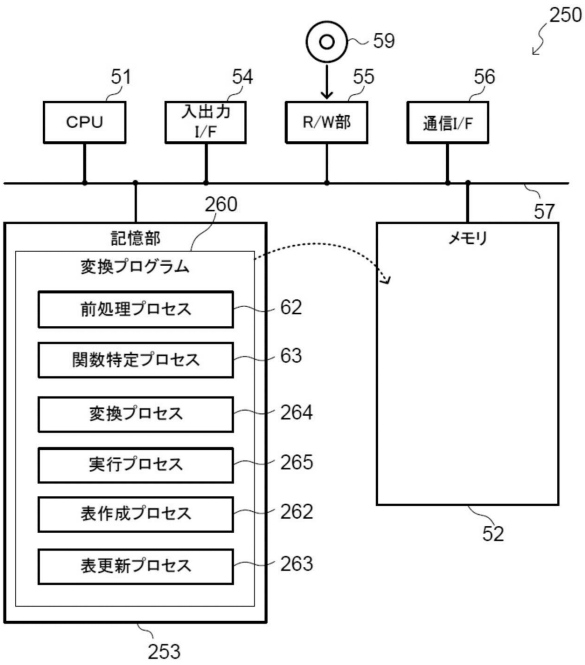
224B

項番	stmt	stmt, APIに対応？	変数操作		並列性		
			data	_data0	writeFile	createBucket	putObject
4			↓			↓	×
5				×			↓
6				↓			
7							

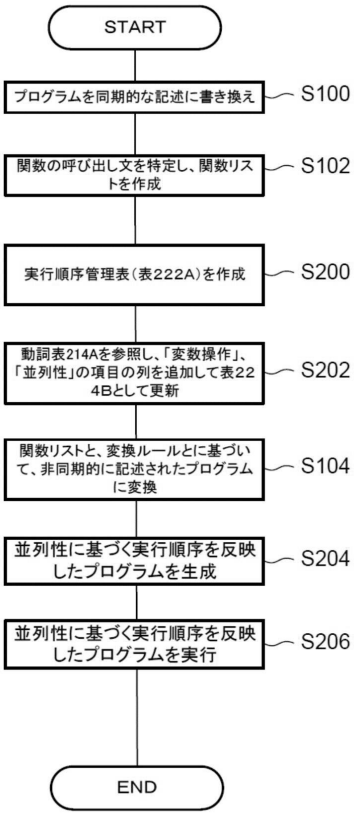
【図３３】

```
const fs = require('fs')
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
…
async function func(filename, bucketName, orgData) {
  try {
    let _promises = [];
    _promises.push((async () => {
      await new Promise((resolve, reject) => {
        fs.writeFile(filename, orgData, (err, data) => {
          if (err) reject(err) else resolve(data)
        })
      })
      return await new Promise((resolve, reject) => {
        fs.readFile(filename, (err, data) => {
          if (err) reject(err) else resolve(data)
        })
      })
    })())
    _promises.push(new Promise((resolve, reject) => {
      s3.createBucket({ Bucket: bucketName }, (err, data) => {
        if (err) reject(err) else resolve(data)
      })
    }))
    let _values = await Promise.all(_promises)
    await new Promise((resolve, reject) => {
      s3.putObject({ Bucket: bucketName, Key: filename, Body: _values[0] }, (err, data) => {
        if (err) reject(err) else resolve(data)
      })
    })
    let _data0 = await new Promise((resolve, reject) => {
      s3.getObject({ Bucket: bucketName, Key: filename }, (err, data) => {
        if (err) reject(err) else resolve(data)
      })
    })
    foo(_data0)
    console.log('done')
  } catch (err) {
    console.log(err)
  }
}
```

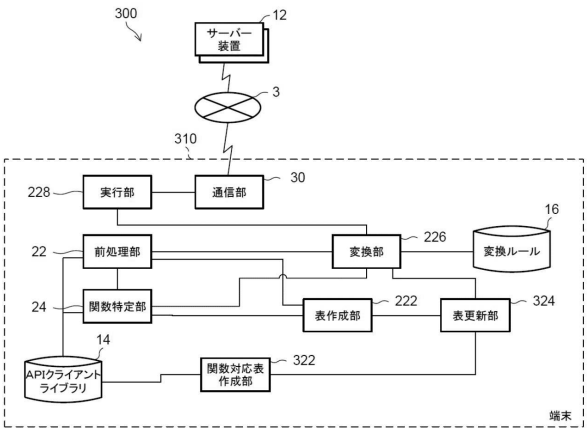
【図３４】



【図３５】



【図３６】



【図３７】

```
const my = require('myapi')
…
function func(id) {
  try {
    my.createUser(id)
    const data = my.getContents(id);
    my.putData(id, data)
  } catch (err) {
    console.log(err)
  }
}
```

【図３８】

24B

オブジェクト変数名	関数名
my	createUser
my	getContents
my	putData

【図３９】

222B

項番	ステートメント(stmt)	APIに対応？
1	my.createUser(id)	true
2	const data = my.getContents()	true
3	my.putData(id, data)	true

【図４０】

322A

関数名	メソッド	パス
myapi#createUser	POST	http://myapi.com/api/users
myapi#getContents	GET	http://myapi.com/api/contents
myapi#putData	PUT	http://myapi.com/api/users/:id

【図４１】

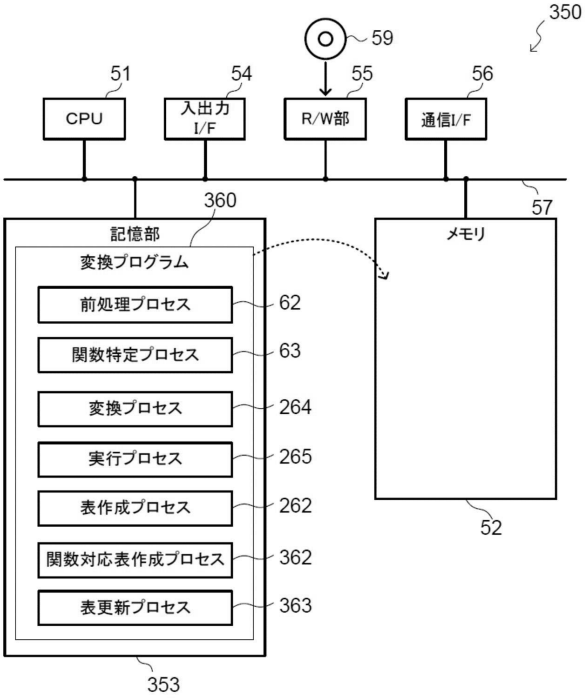
324A

項番	ステートメント(stmt)	APIに対応？	変数操作	並列性	
			data	/users	/contents
1	my.createUser(id)	true		×	
2	const data = my.getContents()	true	×		×
3	my.putData(id, data)	true	↓	↓	

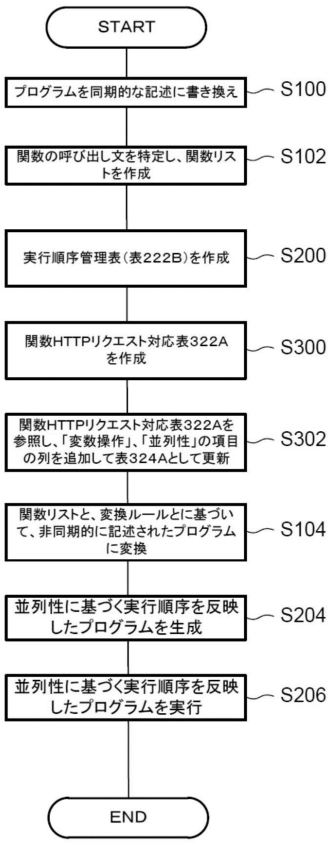
【図４２】

```
const my = require('myapi')
…
async function func(id) {
  try {
    let _promises = []
    _promises.push(new Promise((resolve, reject) => {
      my.createUser(id, (err, data) => {
        if (err) reject(err) else resolve()
      })
    }))
    _promises.push(new Promise((resolve, reject) => {
      my.getContents((err, data) => {
        if (err) reject(err) else resolve()
      })
    }))
    let _values = await Promise.all(_promises);
    await new Promise((resolve, reject) => {
      my.putData(id, _values[1], (err, data) => {
        if (err) reject(err) else resolve()
      })
    })
    console.log('done')
  } catch (err) {
    console.log(err)
  }
}
```

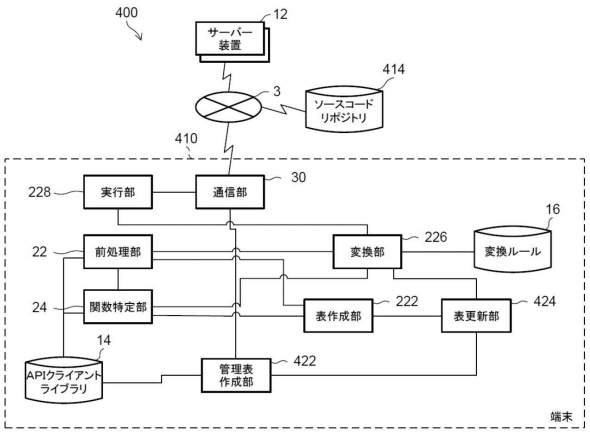
【図４３】



【図４４】



【図４５】



【図４６】

```
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
const dynamodb = new AWS.DynamoDB()
…
function func(filename, bucketName, tableName, data) {
  try {
    s3.createBucket({ Bucket: bucketName })
    dynamodb.createTable({ TableName: tableName })
    s3.putObject({ Bucket: bucketName, Key: filename, Body: data })
    console.log('done')
  } catch (err) {
    console.log(err)
  }
}
```

【図４７】

24C

オブジェクト変数名	関数名
s3	createBucket
dynamodb	createTable
s3	putObject

【図４８】

222C

項番	ステートメント(stmt)	APIに対応？
1	s3.createBucket({ Bucket: bucketName })	true
2	dynamodb.createTable({ TableName: tableName })	true
3	s3.putObject({ Bucket: bucketName, Key: filename, Body: data })	true
4	console.log('done')	false

【図４９】

```
…
s3.createBucket({Bucket: name}, …)
…
s3.putObject({Bucket: name, Key: …})
```

…

```
…
awss3.createBucket({Bucket: b_name}, …)
…
awss3.putObject({Bucket: b_name, Key: …})
```

【図５０】

422A

先に呼ぶ関数	後で呼ぶ関数
aws-sdk#S3.createBucket	aws-sdk#S3.putObject

【図５１】

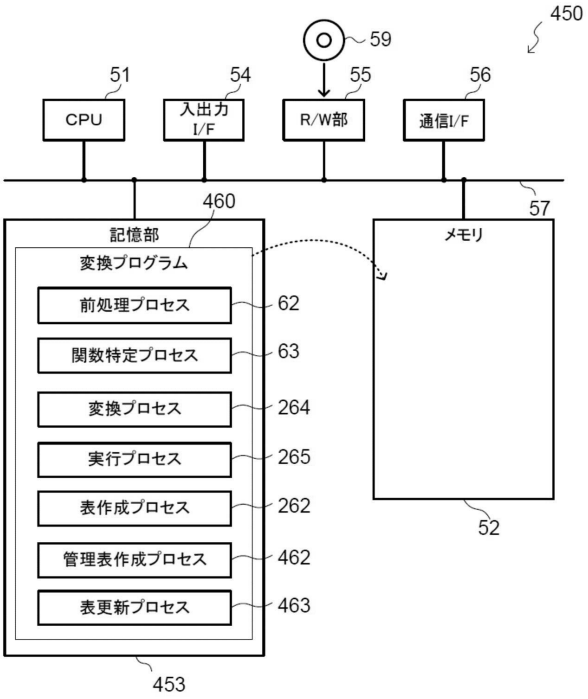
424A

項番	ステートメント(stmt)	APIに対応？	順序		
			aws-sdk#S3.createBucket	aws-sdk#DyanmoDB.createTable	aws-sdk#S3.putObject
1	s3.createBucket	true	×		
2	dynamodb.createTable			×	
3	s3.putObject	true	↓		×
4	console.log('done')	false			

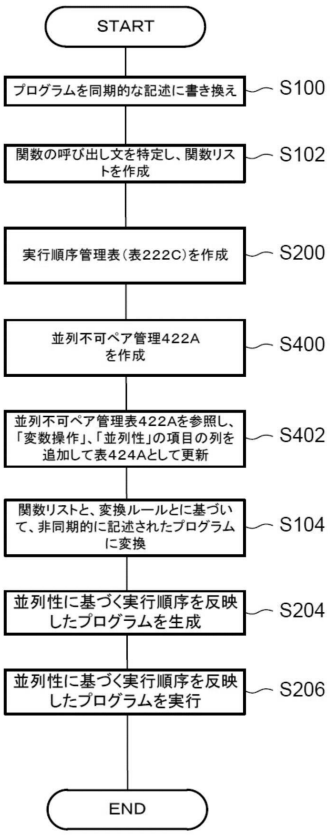
【図５２】

```
const AWS = require('aws-sdk')
const s3 = new AWS.S3()
const dynamodb = new AWS.DynamoDB();
…
async function func(filename, bucketName, tableName, orgData) {
  try {
    let _promises = [];
    _promises.push(new Promise((resolve, reject) => {
      s3.createBucket({ Bucket: bucketName }, (err, data) => {
        if (err) reject(err) else resolve(data)
      })
    }))
    _promises.push(new Promise((resolve, reject) => {
      dynamodb.createTable({ TableName: tableName }, (err, data) => {
        if (err) reject(err) else resolve(data)
      })
    }))
    let _values = await Promise.all(_promises);
    await new Promise((resolve, reject) => {
      s3.putObject({ Bucket: bucketName, Key: filename, Body: orgData }, (err, data) => {
        if (err) reject(err) else resolve()
      })
    })
    console.log('done')
  } catch (err) {
    console.log(err)
  }
}
```

【図５３】



【図５４】



【図５５】

手法	得意とする適用先
関数名に含まれる動詞名詞（第2実施形態）	動詞と名詞で注意深く名前付けられたAPIライブラリが提供されているもの
APIクライアントの関数とHTTPリクエストの対応（第3実施形態）	RESTfulなWebAPI
外部ソースコードリポジトリでの利用状況（第4実施形態）	APIを利用したソースコードが多数公開されているもの

フロントページの続き

番１号 富士通株式会社内
(72)発明者 関口 敦二
神奈川県川崎市中原区上小田中４丁目１番１号 富士通株式会社内
審査官 井上 宏一
(56)参考文献 特開２００１－１８４３２０（ＪＰ，Ａ）
再公表特許第２００７／１０８１３３（ＪＰ，Ａ１）
米国特許出願公開第２０１８／００８８９３７（ＵＳ，Ａ１）
(58)調査した分野 (Int.Cl.，ＤＢ名)
Ｇ０６Ｆ ９／４５５－９／５４