非同期処理の記述を支援するライブラリの提案と実装

佐久間 隆平[†1]　福田 浩章[†1]

概要：近年のソフトウェア開発では, 非同期処理を取り扱う機会が増加している. 非同期処理とは, 関数の終了を待たずに次の処理を開始し, 関数の終了は処理結果とともに通知されるという仕組みである. 非同期処理は, データの取得や出力といった I/O 処理など, 時間のかかる処理で頻繁に用いられ，処理結果を受け取るためにコールバック関数が使用される．また，非同期処理の関数では処理の終了するタイミングが一意に決まらないため，処理の順序を保つためにはコールバック関数を入れ子にする必要がある．その結果, コールバック関数を多用する場合や入れ子にする場合などはコードが複雑になる. この複雑化を回避する手法として, 現在は Promise が頻繁に用いられている. また, Promise には複数の非同期処理を並列に実行し, それらの実行状況に応じて then に登録した関数を実行する仕組みとして Promise.all や Promise.race が実装されている. しかし, これらのメソッドでは then に登録した関数を実行するタイミングが一意に決まるため柔軟な対応を行うことができない.
そこで本研究では, コールバック関数の実行するタイミングを, 開発者が同一のインタフェースで柔軟に変更できるライブラリ, MatchingPromise の提案と実装を行う. MatchingPromise では，コールバック関数を実行するタイミングを指定する手段として, パターンマッチを用いる. そして，柔軟にパターンを指定する方法として MatcherCells を用いる. また，MatchingPromise の実装後，評価を行い，コールバック関数の実行タイミングを柔軟に変更可能であることを示した.

キーワード：非同期処理，Promise，JavaScript，ライブラリ

Proposal and Implementation of Library Supporting Descriptions of Asynchronous Processing

Ryuhei Sakuma[†1]　Fukuda Hiroaki[†1]

Abstract:
In recent software development, opportunities to deal with asynchronous processing are increasing. The synchronization process starts the next process without waiting for the end of the function and terminates the function It is a mechanism that it is notified together with fruits.In asynchronous processing, data acquisition and output Frequently used in time-consuming processing such as I / O processing such as power, callbacks are used to receive processing results. Also, with the asynchronous processing function, the timing at which the processing ends can not be uniquely determined, In order to preserve processing order, it is necessary to nest callbacks. As a result, when call back is heavily used, nested etc. The code becomes complicated. As a method to avoid this complication and Currently Promise is used frequently, and Promise contains multiple non - Synchronization processing was executed in parallel and registered in then according to their execution status Promise.all and Promise.race are implemented as a mechanism for executing functions However, these methods execute the function registered in then It is impossible to flexibly respond because timing is uniquely determined. Therefore, in this research, the timing to execute the callback function is set to be the same as that by the developer Proposal and implementation of library, MatchingPromise which can be flexibly changed by interface In MatchingPromise, as a means to specify the timing to execute the callback function, putter We use MatcherCells as a method to flexibly specify patterns. After implementing MatchingPromise, evaluation was performed, and it was shown that the execution timing of the callback function can be changed flexibly.

Keywords: asynchronous, Promise, JavasScript, Library

© 2018 Information Processing Society of Japan

```
1 get(hoge.html, function (err,cssURL) {
2   get(cssURL, function (err, imgURL) {
3     get(imgURL, function (err, img) {
4     });
5   });
6 });
```

Listing 1 コールバックのネスト

```
1 var p1 = new Promise((resolve, reject) => {
2   jQuery.get(url1,(value) =>{resolve(value);});
3 });
4 var p2 = new Promise((resolve, reject) => {
5   jQuery.get(url2,(value) =>{resolve(value);});
6 });
7 var p3 = new Promise((resolve, reject) => {
8   jQuery.get(url3,(value) =>{resolve(value);});
9 });
10 Promise.race([p1,p2,p3]).then((value) => {
11   console.log(value);
12 });
```

Listing 2 Promise.race を用いたデータ取得

1. はじめに

近年のソフトウェア開発では，非同期処理を取り扱う機会が増加している．非同期処理とは，関数の終了を待たずに次の処理を開始し，関数の終了は処理結果とともに通知されるという仕組みである．非同期処理は，データの取得や出力といった I/O 処理など，時間のかかる処理で頻繁に用いられる．また，非同期処理の関数では処理の終了するタイミングが一意に決まらない．そのため，例えばクローラーの実装において，取得した HTML から CSS や画像ファイルのパスを抜き出しそこへアクセスするといった，処理の順序を保つ必要がある場合，Listing 1 のようにコールバックを入れ子にして使用する必要があり，処理が複雑になる [1]．

コールバックによるコードの複雑化を回避する手法として，現在は Promise[2] が用いられている．また，Promise には複数の非同期処理を並列に実行し，それらの実行状況に応じて then に登録した関数を実行する仕組みとして Promise.all や Promise.race が標準で実装されている．しかし，これらのメソッドでは then に登録された関数を実行するタイミングが一意に決まるため柔軟な対応を行うことができない．Promise.race を用いた例を Listing 2 に示す．Promise.race は，race メソッドで登録した Promise のうち，どれか 1 つでも実行を終えたときに，then メソッドに登録した関数を実行する．

Listing 2 では，非同期処理の集合である [p1，p2，p3] のうち，どれか 1 つでも実行完了したとき，then に登録された関数が実行される．しかし，[p1，p2，p3] のうち，どれか 1 つでも実行が失敗した場合は then に登録された関数が実行されない．そのため，複数のミラーサーバに接続し，”どこか 1 箇所でも接続できれば then に登録された関数を実行する”，という処理を行う場合，Promise.race では対応できない．なぜならば，Promise.race では，最初に処理を終えた Promise 内の非同期処理の実行が失敗していた場合，then メソッドで登録した関数が呼び出されないためである．このように，Promise.all，Promise.race では，それぞれの振る舞いがあらかじめ決められており，開発者が変更できないため，柔軟な対応を実現できない．

そこで本研究では，非同期処理において，コールバック関数の実行するタイミングを開発者が同一のインタフェースから柔軟に変更できるライブラリ，MatchingPromise の提案と実装を行う．MatchingPromise ではコールバック関数を実行するタイミングを指定する手段として，パターンマッチを用いる．そして，柔軟にパターンを指定する方法として MatcherCells[3] を用いる．

†1 現在，芝浦工業大学
Presently with Shibaura Institute of Technology

2. 関連研究

本節では，本研究で用いる関連技術について述べる．

2.1 Promise

Promise[2] は非同期処理などの成功や失敗といったイベントに対してハンドラを設定するデザインパターンである．Promise は非同期処理を内部に持つオブジェクトである．Promise は次の 3 つの状態を保持する．

- Fulfilled：非同期処理の resolve(成功)
- Rejected：非同期処理の reject(失敗)
- Pending：Fulfilled または Rejected 以外

Promise は非同期処理の結果に応じて 3 状態のいずれかに状態遷移する．Promise を用いた例を Listing 3 に示す．

Listing 3 では，非同期のファイル読み込み処理を Promise を用いて実装した例である．3 行目で Promise オブジェクトを作成し，引数として関数を与える．readFile には第 1 引数に読み込むファイルのパスを，第 2 引数にテキストの文字コードを，第 3 引数にコールバック関数を指定する．ここで指定したコールバック関数の第 1 引数はエラーオブジェクトが，第 2 引数はファイルのデータが入る．第 1 引数で指定されたファイルの読み込みが成功すると，6 行目の resolve が実行され，Promise の状態が Fulfilled に変化し，13 行目で then メソッドに登録された関数が実行される．一方，ファイルが存在しないなどの理由からファイル読み込みが失敗したときは 8 行目の reject 関数が実行され，Promise の状態が Rejected に変化し，15 行目で catch に登録された関数が実行される．このように，Promise では非同期処理の結果と，結果に従ったコールバック関数の呼び出しを対応づけている．

© 2018 Information Processing Society of Japan

```
1  let fs = require('fs');
2
3  let promise = new Promise(function(resolve,
       reject){
4    fs.readFile(filepath, {encoding: 'utf-8'},
         function(err, data){
5      if(!err){
6        resolve(data);
7      } else {
8        reject(err);
9      }
10   });
11 });
12
13 promise.then(function(value){
14   console.log(value);
15 }).catch(function(reason){
16   console.log(reason);
17 });
```

Listing 3 Promise を用いたファイル読み込みのコード例

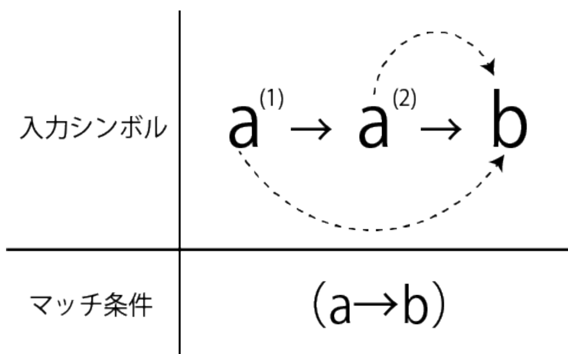

図 1 パターンマッチの振る舞いとして解釈可能なパターン

2.2 MatcherCells

MatcherCells とはパターンマッチの手法の 1 つである．MatcherCells は自己複製アルゴリズム [4] を用いており，開発者が後述する rule を定義することでパターンマッチの振る舞いを変更することができる．パターンマッチの振る舞いに関して詳しく述べる．図 1 に,”シンボル [a] の次にシンボル [b] が与えられたらマッチする (a → b)”をマッチ条件とし，入力シンボルとして [a，a，b] を与えた例を示す．このマッチ条件は次の 2 つの意味で解釈できる．

(1) シンボル [a] の直後にシンボル [b] にマッチする
(2) シンボル [a] の後のシンボル [b] にマッチする

解釈 1，2 では，図 1 の a(1) と b がマッチするか否か，が異なる．このマッチ条件を (1) で解釈した場合，2 つ目の入力シンボル [a] と 3 つ目の入力シンボル [b](2) の 1 回マッチとなる．一方，(2) で解釈した場合，1 つ目の入力シンボル [a] と 3 つ目の入力シンボル [b](1) と 2 つ目の入力シンボル [a] と 3 つ目の入力シンボル [b](2) の 2 回マッチとなる．MatcherCells では開発者が，このような様々なパターンマッチの振る舞いを，後述する rule を実装することで柔軟に変更することができる．

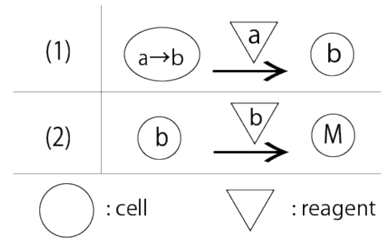

図 2 cell の生成

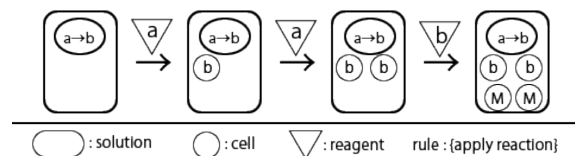

図 3 MatcherCells を用いたパターンマッチ例

MatcherCells は cell，reagent，solution，rule によって構成される．MatcherCells では，solution に cell を追加し，reagent と反応させることでパターンマッチを行う．cell にはマッチ条件を指定し，入力シンボルを reagent として cell と反応させる．

2.2.1 cell の振る舞い

cell は reagent と反応し，以下に示す 3 つの振る舞いのいずれかを実行する．

- 新しい cell を生成
- 反応を示さない

cell が reagent と反応し新たな cell を生成する様子を図 2 に示す．

図 2 のように，(1)cell が反応するシンボル (reagent) が与えられた場合 (*e.g.*, a), 次のシンボルに反応する新たな cell を生成する．そして，(2) 反応すべき次のシンボルが存在しない場合，それまでに与えられたシンボルとパターンがマッチしたと判断し，Mcell を生成する．MatcherCells では，マッチとは Mcell が生成されることを意味する．

2.2.2 MatcherCells を用いたパターンマッチ

MatcherCells を用いたパターンマッチの例を図 3 に示す．図 3 では,”reagent として，シンボル [a] の次にシンボル [b] が与えられたらマッチする (a → b)”をマッチ条件とし，reagent としてシンボルは [a，a，b] を与えている．

図 3 では，solution に reagent が与えられ，cell が反応を示している．以下では reagent として与えるシンボルを入力シンボルと呼称する．1 つ目の入力シンボル [a] に対し，(cell : a → b) は反応を示し，(cell : b) を生成する．また，2 つ目の入力シンボル [a] に対しても同様に，(cell : a → b) は反応を示し，(cell : b) を生成する．このとき (cell : b) は入力シンボル [a] に対しては反応を示すシンボルが異なるため，反応を示さない．3 つ目の入力シンボル [b] に対しては (cell : a → b) は反応を示さない．一方，1 つ目，

© 2018 Information Processing Society of Japan

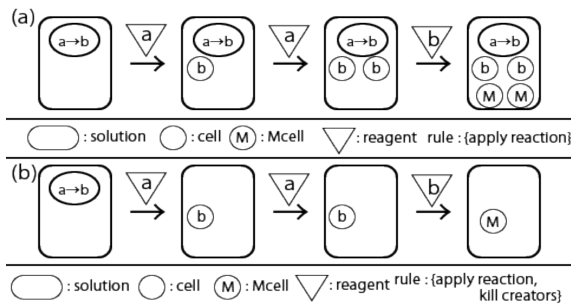

図 4　cell の反応の変化

2 つ目の入力シンボルとの反応によって生み出された (cell : b) はシンボル [b] に反応し，Mcell をそれぞれ生成する．その結果，与えたマッチ条件と入力シンボルは 2 回マッチすることになる．

2.3　rule による振る舞いの変化

MatcherCells では，reagent と反応する cell の振る舞いは rule によって決定される．rule にはインタフェースが提供されており，それを用いてプログラマは rule を任意に実装し与えることで，cell の振る舞いを柔軟に変更することができる．rule の変化による cell の振る舞いの変化を図 4 に示す．図 4 はマッチ条件を，”シンボル [a] の次にシンボル [b] が reagent として与えられたらマッチする (a → b)” とし，入力シンボルとして [a，a，b] を与えている．図 4(a) では rule として apply reaction を与え，(b) では apply reaction の他に，kill creators を与えている．

以下にそれぞれの rule について述べる．

- apply reaction : solution の cell と reagent を反応させる
- kill creators : 新しい cell を生成した cell を破壊する

図 4 の cell はそれぞれの rule に応じた振る舞いを示す．そのため，図 4 にあるように，同じマッチ条件，入力シンボルであっても異なる結果を示す．

3.　MatchingPromise

MatchingPromise は，非同期処理を並列に実行した時のそれぞれの実況状況に応じたコールバック関数の実行タイミングを，柔軟に指定する事のできるライブラリである．コールバック関数の実行タイミングを指定する手段として，MatcherCells を用いる．MatchingPromise では，実行完了した Promise を reagent として MatcherCells に与えることでパターンマッチを行う．また，非同期処理に対応するためシンボル [∗] と MPromise を提供する．以下にそれぞれについて述べる．

3.1　シンボル [∗]

非同期処理は処理が終了するタイミングが不明であるため，マッチ条件をシーケンスとして指定することは現実的

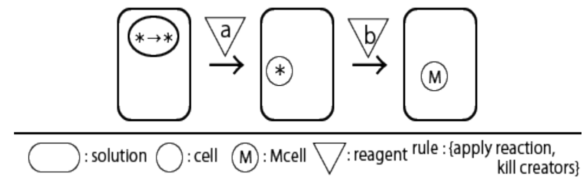

図 5　[∗] を用いた例

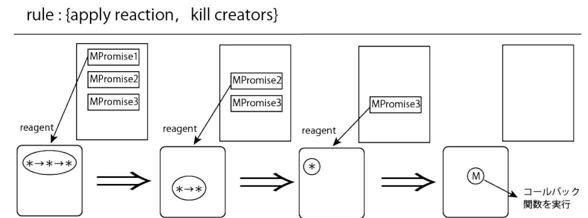

図 6　MPromise を対象としたパターンマッチ

ではない．そのため，任意のシンボルを持つ reagent に反応するものとして，シンボル [∗] を用意する．シンボル [∗] の挙動を図 5 に示す．図 5 ではマッチ条件として，”シンボル [∗] の次にシンボル [∗] が reagent として与えられたらマッチする (∗→∗)” を与え，入力シンボルとして [a，b] を与えている．シンボル [∗] が入力されたら反応する cell は任意のシンボルに反応するため，入力シンボル [a，b] のどちらにも反応を示す．よって，(cell : ∗→∗) は，入力シンボル [a] に対して反応を示し，(cell : ∗) を生成する．また，rule として kill creators が指定されているため，(cell : ∗→∗) は反応後に破壊される．次に，入力シンボル [b] に対して，(cell : ∗) が反応を示し，Mcell が生成される．また，反応後 (cell : ∗) は kill creators のため破壊される．よって図 5 では 1 回マッチの判定が行われる．

3.2　MPromise

Promise では，状態が隠蔽されており，外部から監視することができない．よって MatchingPromise では，Promise の状態をでき，MatcherCells と連携する MPromise を Promise のラッパーとして提供する．MPromise は Promise と同様に，非同期処理などの成功や失敗といったイベントに対してハンドラを設定するデザインパターンである．MPromise は Promise が持つ機能に加え，次の機能を持つ．

- MatcherCells から MPromise の状態を参照．
- MPromise を reagent とし MatcherCells へ与える．

MatingPromise では，並列に実行する非同期処理を MPromise を用いて記述し，コールバック関数の実行タイミングを制御する．

3.3　パターンマッチの実行手順

MPromise を reagent とした MatcherCells の挙動を図 6 に示す．

図 6 ではマッチ条件を (∗→∗→∗) とし，rule には apply reaction と kill creators を指定している．状態が Pending から変化した MPromise を reagent として，cell に与えることでマッチ処理を行っている．図 6 ではマッチ条件 (∗→∗→∗) に対し，[MPromise1，MPromise2，MPromise3] が内部の非同期処理の実行完了とともに，その MPromise を reagent として MatcherCells に与える．図 6 は，[MPromise1，MPromise2，MPromise3] が内部に持つ非同期処理の実行を完了する順番が [MPromise1 → MPromise2 → MPromise3] である場合の例を示している．まず始めに，MPromise1 の実行が完了すると，MPromise1 が reagent として solution に与えられ，solution の cell が反応を示す．図 6 では MPromise1 の reagent と (cell：∗→∗→∗) が反応し，(cell：∗→∗) が生成される．また kill creators が rule として与えられているため，(cell：∗→∗→∗) は反応後に破壊される．そして，実行完了時に MPromise2 を reagent として solution に与え，cell と反応する．同様に，(cell：∗→∗) と reagent が反応し，(cell：∗) が生成され，(cell：∗→∗) は破壊される．最後に MPromise3 の実行が完了した時，同様に MPromise3 を reagent として solution に与え，cell が反応する．(cell：∗) と reagent が反応し，Mcell が生成され，(cell：∗) は破壊される．結果として 1 回マッチ判定が行われ，コールバック関数が実行される．

4. 設計と実装

本節では，MatchingPromise の設計と実装について述べる．

4.1 reagent

MatchingPromise では MPromise を reagent として，MatcherCells へ与える．その reagent は次の情報を持つ．

- MPromise の名称
- MPromise の状態
- MPromise の結果の値

reagent が情報として持つ MPromise の名称は，その MPromise が後述する API で渡す MPromise の配列の何番目に与えられているのかが名称となる．つまり [MPromise1, MPromise2] で与えられた場合，MPromise1 の名称は”1”，MPromise2 の名称は”2” となる．MatcherCells を用いてパターンマッチを行う場合には，reagent が保持する MPromise の名称をシンボルとして，cell と反応させる．

4.2 Cell クラスの実装

MatchingPromise では，非同期処理の結果を管理するため，Cell クラスは次の情報を保持する．

- cell が反応するシンボル
- cell の反応によって生成した cell が反応するシンボル
- cell を生成した cell(親 cell)
- cell を生成したときに reagent が保持する MPromise の結果

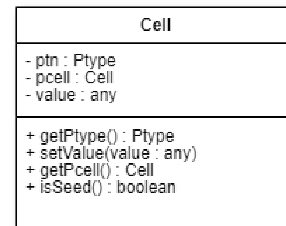

図 7 MPromise を対象としたパターンマッチ

```
1 interface Rule {
2   eval(solution : Solution , s: Reagent) :
        Solution ;
3 }
```

Listing 4 Rule クラスのインタフェース

Cell クラスのクラス図を図 7 に示す．図 7 では，ptn は Ptype クラスのインスタンスであり，その中に cell が反応するシンボルと cell の反応によって生成した cell が反応するシンボルの情報を持ち，pcell に親 cell の情報，value に結果の情報を持つ．また，getPtype メソッドは返り値として ptn を返し，setValue メソッドでは Cell クラス内の value に値を代入する．getPcell メソッドは返り値として pcell，つまり親 cell の情報を返し，isSeed メソッドは，この cell に親 cell が存在していれば false を，存在していなければ true を返す．

4.3 cell の反応

cell の反応を実現させるため，react メソッドを実装した．react メソッドでは，引数として cell と reagent を受け取り，返り値として新たな cell を返す．react メソッドでは返り値として，reagent のシンボルと cell の反応するシンボルが一致した場合は生成した cell を返す．また，reagent の入力シンボルと，cell の反応するシンボルが異なる場合は，その cell 自身をそのまま返す．

4.4 rule の実装

rule を実装する際のクラスは，インタフェースとして Listing 4 に示すインタフェースを備える必要がある．

eval メソッドは，Solution が保持する Cell クラスとと reagent を反応させ，結果を含む新たな Solution を返す必要がある．開発者は，このインタフェースに従って自身の rule を実装すれば良い．rule の実装例として，apply reaction の実装を Listing 5 に示す．

Listing 5 では 3 行目で Solution クラスから Cell クラスのリストを受け取り，6 行目で react メソッドを呼び出すことで cell と reagent を反応させ，8，10 行目で反応前の

© 2018 Information Processing Society of Japan

```
class ApplyReaction implements Rule {
  public eval(solution : Solution, s : Reagent)
      : Solution {
    let list : Array<Cell> = solution.toList();
    let nsolution : Solution = new Solution();
    list.map(function(value : Cell) {
      return Matcher.react(value, s);
    }).forEach(function(value : Cell) {
      nsolution.add(value);
    });
    nsolution.addAll(solution);
    nsolution.removeDuplicate();
    return nsolution;
  }
}
```

Listing 5 ApplyReaction クラスの実装

メソッド名	概要
getInstance()	RuleFactory のインスタンスを生成する
apply_reaction()	rule：apply reaction を生成する
sym(symbol)	マッチ条件となるシンボルを指定する
seq(first, next)	マッチ条件のシンボルのシーケンスを指定する

表 1 RuleFactory クラスで提供するメソッド

```
let rf:RuleFactory = RuleFactory.getInstance();
let pt = new Ptype(this.rf.seq(this.rf.seq(this
    .rf.sym('*'),this.rf.sym('*')), this.rf.sym
    ('*')), new Environment());
```

Listing 6 マッチ条件の生成

cell と生成された cell を 1 つの Solution クラスにまとめている．そして，11 行目で重複した cell を取り除いている．なぜならば，6 行目で呼び出した react メソッドにおいて，reagent に対し反応していない cell が存在している場合は，その cell が重複するためである．開発者は，このように Rule インタフェースに従って cell を操作することで，様々なパターンマッチを実現できる．

4.5 パターンと rule の指定

rule の指定方法として，MatchingPromise ではクラス，RuleFactory を提供している．RuleFactory では，実装した rule のクラスのインスタンスを生成し，MatcherCells へ適用する．よって，新たに rule を実装した開発者は，この RuleFactory クラスで rule のインスタンスを生成する必要がある．RuleFactory で実装されるメソッドを表 4.5 に示す．

マッチ条件を指定する手段として，MatchingPromise では sym メソッドと seq メソッドを用いる．実際の手順を Listing 6 に示す．

Listing 6 では，1 行目で RuleFactory のインスタンスを生成し，2 行目でマッチ条件 (∗→∗→∗) を持つ cell のインスタンスを生成している．また rule を指定する手順を

```
let rf:RuleFactory = RuleFactory.getInstance();
let rule:Rule = this.rf.apply_reaction();
```

Listing 7 rule の生成

```
MatchingPromise.matchobj([mpromise, …
    ]).matchthen(seq,rule,callback(value));
```

Listing 8 提供する API

```
let rf = RuleFactory.getInstance();
let pt = new Ptype( this.rf.sym('*'), new
    Environment());
let rule = this.rf.apply_reaction();
MatchingPromise.matchobj([getAsMPromise('/api1
    ')]).matchthen(pt, rule, (value) => {
  $("p").after(value + '\n');
});
```

Listing 9 MatchingPromise を用いた非同期処理

```
getAsPromise('/api1').then((value) => {
  $("p").after(value + '\n');
});
```

Listing 10 Promise を用いた非同期処理

Listing 7 に示す．

Listing 7 では，1 行目で RuleFactory のインスタンスを生成し，2 行目で rule として，apply reaction を生成している．

4.6 MatchingPromise で提供する API

MatchingPromise では API として，Listing 8 を提供する．matchobj メソッドでは，並列に実行する MPromise を登録する．また，matchthen メソッドではマッチ条件 (seq)，rule，そしてマッチした際に実行されるコールバック関数を登録する．

5. 評価

本節では，MatchingPromise の評価を述べる．

5.1 定量的評価

MatchingPromise を用いて取得したデータを画面に描画する Web アプリケーションを作成し，スクリプトの実行時間を計測して Promise を用いた場合の実行時間と比較した．コールバック関数の実行タイミングの制御を行うコードを，MatchingPromise と Promise で記述した例をそれぞれ Listing 9，Listing 10 に示す．

Listing 9，Listing 10 の，getAsMPromise，getAsPromise は，共にデータ取得処理を行う非同期関数である．このコールバック関数の実行を 100 回行いその際のスクリプトの実行

MatchingPromise	935.8ms
Promise	578.1ms

表 2　実行速度

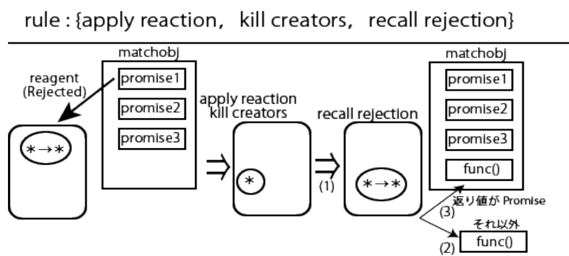

図 8　recall rejection

```
1 let rf = RuleFactory.getInstance();
2 let pt = new Ptype( this.rf.sym('*'), new
      Environment());
3 let rule = this.rf.recall_rejection((reason) =>
      {
4     console.log(reason);
5   }, this.rf.kill_creator(this.rf.
        apply_reaction()));
6 MatchingPromise.matchobj([getAsMPromise(url1),
      getAsMPromise(url2), getAsMPromise(url3)]).
      matchthen(pt, rule, (value) => {
7   $("p").after(value + '\n');
8 });
```

Listing 11　パターン 1 におけるデータ取得の実装

時間を計測した．実行時間の計測結果を表 5.1 に示す．

スクリプトの実行時間を計測した結果，コールバック関数を 100 回実行すると実行時間に約 375.7ms の差が出た．しかし，この差はコールバック関数を 100 回連続して実行したときの結果であり，実際の Web アプリケーションで利用されることは考えにくい．また，Website Response Times[5] で示されている，ユーザのサイトに対する集中力が続く時間が 10 秒であることを考慮すると，実用可能な時間であると判断できる．

5.2　定性的評価

定性的評価として，同一のインタフェースから rule を実装し，それらを組み合わせることで多様なパターンマッチの振る舞いを実現可能であるか，検証した．

5.3　rule の組み合わせ

以下に MatchingPromise 内で新たに実装した rule を述べる．

- add seed
  solution 内の cell が存在しない，もしくは Mcell のみであった場合，指定された cell を追加する．
- recall rejection
  reagent の MPromise の状態が Rejected の場合，反応した cell が存在していれば，solution を反応前の状態に戻し，Reject 時の処理を実行する

5.4　recall rejection

recall rejection は非同期処理に対応させるための rule である．recall rejection の挙動を図 8 に示す．

図 8 では，(cell : ∗→∗) に対して，MPromise1 の結果と状態を reagent として与えている．(cell : ∗→∗) は reagent に反応を示して (cell : ∗) を生成し，(cell : ∗→∗) は破壊される．recall rejection では図 8 のように，与えられた reagent の MPromise の状態が Rejected の場合，反応した cell が存在していれば，solution を反応前の状態に戻す (1)．そのため，solution の cell は (cell : ∗→∗) のみとなる．その後，recall rejection を指定した時に登録した reject 時の処理 (func()) を実行する (2)．また，reject 処理の返り値が MPromise の場合，reject 時の処理 (func()) を matchobj に登録した後に実行する (3)．この rule を用いることで，reject 時の繰り返し処理やログの出力などに対応できるようになる．開発者は，recall rejection を他のルール，apply reaction や kill creators などと組み合わせることで，コールバック関数の実行タイミングを柔軟に指定することが可能となる．

次に，rule の実装やそれらの組み合わせでコールバック関数の実行タイミングを制御できることを示すため，Web アプリケーションを開発し検証した．ここでは，次に示す 2 つの異なるタイミングでコールバック関数を実行する例を示す．

(1) 複数のミラーサーバに接続し，最初に受け取ったデータを用いて処理を行う
(2) 複数データを取得し，3 つデータを取得するたび，表示を行う

パターン 1 のとき，つまり複数のミラーサーバに接続し，最初に受け取ったデータを用いて処理を行う場合のコールバック関数の記述を Listing 11 に示す．

Listing 11 ではマッチ条件に (∗)，rule に recall rejection と kill creators と apply reaction を与えている．reagent が与えられたとき，処理結果が Rejected であれば，recall rejection によって Reject 時の処理が実行され，cell が反応前の状態に戻される．一方，Fulfilled であった場合は，(cell : ∗) と反応を示し，Mcell を生成した後，(cell : ∗) は破壊される．そのため，Listing 11 では，matchobj に登録された MPromise が Rejected になった場合，ログを出力し，Fulfiled になった場合，データの描画を行う挙動を示す．

次にパターン 2，つまり複数データを取得し，3 つデータを取得するたび，表示を行う場合のコールバック関数の記述を Listing 12 に示す．

Listing 12 ではマッチ条件に (∗→∗→∗)，rule に add

```
1 let rf = RuleFactory.getInstance();
2 let pt = new Ptype(this.rf.seq(this.rf.seq(this
    .rf.sym('*'), this.rf.sym('*')), this.rf.
    sym('*')), new Environment());
3 let rule = this.rf.add_seed(pt, this.rf.
    kill_creator(this.rf.apply_reaction()));
4 MatchingPromise.matchobj(MPromiseList).
    matchthen(pt, rule, (value) => {
5   $("p").after(value + '\n');
6 });
```

Listing 12 パターン 2 におけるデータ取得の実装

seed, kill creators, apply reaction を与えている．reagent と 3 回反応を示し，solution の cell が Mcell のみとなった時，add seed により，(cell : ∗→∗→∗) が生成される．よって，Listing 12 では matchobj に登録された MPromise が 3 つ Fulfiled になるたびに，マッチ判定が起きる．そのため，3 つデータ取得が成功するたびにコールバック関数が呼び出される．

このように，rule を実装し，それらを組み合わせて適用することでコールバック関数の実行タイミングを変更することができる．

6. まとめと今後の課題

本研究では，非同期処理の記述を支援するライブラリ，MatchingPromise の提案と，実装を行った．MatchingPromise では，非同期処理のコールバック関数の実行タイミングを同一のインタフェースからプログラマが指定できる．その結果，コールバック関数の実行タイミングを開発者が自由に制御できるようになる．MatchingPromise の評価として，Web アプリケーションの開発を行い，複数状況下に MatchingPromise が適応可能であり，実行速度が実用可能であることを示した．

今後の課題として，Listing 12 などから分かる通り，rule の指定やマッチ条件の指定によるコードの見通しの悪さの改善が挙げられる．実用化のためには，コードの見通しの悪さの改善は必要であると考えられる．これは，MatchingPromise で rule やマッチ条件の指定のためのメソッドを提供することで改善できると考えている．

参考文献

[1] M.Ogden.*et al.*Callback hell http://callbackhell.com/.
[2] D. Friedman and D. Wise*et al.*The Impact of Applicative Programming on Multiprocessing, Technical report, Indiana University, Computer Science Department, 1976.
[3] Paul Leger, ric Tanter*et al.*A self-replication algorithm to flexibly match execution traces In Proceedings of the eleventh workshop on Foundations of Aspect-Oriented Languages Pages 27-32 ,2012
[4] J. V. Neumann.*et al.*Theory of Self-Reproducing Automata. University of Illinois Press, Champaign, IL, USA, 1966.
[5] Website Response Times https://www.nngroup.com/articles/website-response-times/

© 2018 Information Processing Society of Japan