Async Tree Pattern

Guseyn Ismayylov

July 2018

1 Introduction

Async Tree Pattern is a declarative design pattern, the main purpose of which is to transform procedural code into the object-oriented(declarative) code in the asynchronous environment via **async tree**, which is the core idea of this pattern.

2 Declarative vs Imperative

The conception of the *async tree* was created with assumption that declarative programming is much better than imperative style of writing code for big programs and systems. Declarative code is more readable, extensible and maintainable in general.

You may agree or disagree with this statement. It's only opinion of the author of this work.

Here it will be shown the difference between these two approaches by small example. Let's consider the following imperative code:

```
// Imperative, pseudocode
user = getUserFromDb(userId);
account = user.createNewAccount(accountInfo);
user.saveAccount(account);
```

It can be written in the declarative style:

```
// Declarative, pseudocode
SavedAccount(
  CreatedAccount(
    UserFromDb(userId), accountInfo
  )
).call();
```

The example is quite small to demonstrate why declarative programming is preferable. Nevertheless, you can see that declarative code always shows the result of program execution and what it requires to get this result. For readability it's more important to be able to see what result you get, not how exactly you get it.

3 Composition of async objects

The last example with the declarative code is actually a composition of **async objects**.

Async object is an object that represents(computes) some other(but similar in terms of logic) object. Also async object can represent primitive type of data.

So, for example, `SavedAccount`, `CreatedAccount` are async objects because they represent `Account`, which is simple object. `UserFromDb` is also an async object, it represents simple object `User`.

Async object also can be defined as a wrapper around some *async call* or *sync call* that computes its representation (why not to use separate abstraction like **sync object** for sync calls? well, it will be described later).

Every async object can be constructed by other async objects or representations of these async objects. For example, `SavedAccount` can be created by `CreatedAccount` or any other async object that represents `Account`. Obviously, it can be created by `Account` itself.

So, composition of async objects can also contain simple objects and primitives.

That's the main idea of the Async Tree Pattern: to provide flexible way to create composition of objects via their representations.

4 Asynchronous environment and *callback hell*

The implementation of the Async Tree Pattern will be described for Node.js. The choice is very simple because Node.js is the most popular and stable asynchronous event driven runtime and it performs well.

If you're familiar with Node.js and how it works, you might know about *callback hell* problem. Callbacks is the main feature and problem in Node.js. Although it's a beautiful abstraction, it decreases readability of a program while it grows up.

Actually, callback hell is one of the reasons why Async Tree Pattern was created.

There are relatively new conceptions in JavaScript like *Promises* and *async/await*. But in my opinion, these abstractions are not suitable for object-oriented style. Simply because they don't provide the main feature of OOP: **encapsulation**. It's not possible to create a composition of objects and write declarative code with these abstractions.

Node.js is a set of modules with static asynchronous methods. And this is main problem - static methods are not actually attached to any objects. They just exists in namespaces such as fs, http, stream, buffer and so on.

Static methods with callbacks don't allow to use results of their execution for other operations in the explicit way. The more asynchronous calls in the code, the harder to control flow of data there.

Let's say we want to write content to a file that has been read from another one. And all these operations are asynchronous, of course. So, instead of writing something like this:

```
fs.readFile('./../file1.txt', 'utf8', (err, result) => {
  if (err != null) {
    throw err;
  }

  fs.writeFile('/../file2.txt', result, (err) => {
    if (err != null) {
      throw err;
    }
  });
});
```

we can design our code in the following style:

```
new WrittenFile(
  './../file2.txt',
  new ReadContentFromFile('./../file1.txt', 'utf8')
).call();
```

As you can see, we use async objects instead of async calls and their representations as results of operations they correspond to. So, `ReadContentFromFile` represents `string` - content from a file, which is the result of the async call `fs.readFile`. And `WrittenFile` represents a file that has been written with some content. Although `fs.WriteFile` does not return anything, we can use `WrittenFile` as a file for other operations if it's needed.

5 Features of the Async Tree Pattern

5.1 Flexibility

The main question needs to be answered for turning asynchronous code into the OOP code is *"What is the main point of doing async call?"*

Well, it's simple: receive a result from an I/O operation or just handle an error in case if something fails. That means that we can represent an I/O call as a result that can be received in the future, and once it's ready it can be used as argument for another async call.

Let's return back to the example with reading and writing files. Objects `WrittenFile` and `ReadContentFromFile` are async objects, and they have the same arguments that their corresponding async calls have. So, here the first argument of `WrittenFile` is a path of a file we want to write content to, second one is the content we want to write. And as you noticed, second argument is represented here as `ReadContentFromFile`. It means that method `call()` of `WrittenFile` invoke first `ReadContentFromFile` and use its result as content for `WrittenFile`.

It's good, but it could be better. For making this declarative abstraction flexible we need a possibility to use either ready results or async objects that

represent these results as arguments for construction the whole composition.

For example, we can use second argument of `WrittenFile` as a string:

```
new WrittenFile('./../file2.txt', 'content to write').call();
```

or use the fist argument as something that has been read from another file:

```
/* here file3.txt contains information
  for the first argument of WrittenFile: './../file2.txt' */
new WrittenFile(
  new ReadContentFromFile('./../file3.txt', 'utf8'),
  new ReadContentFromFile('./../file1.txt', 'utf8')
).call();
```

or even just use every async object independently:

```
new ReadContentFromFile('./../file.txt', 'utf8').call();
```

That's how we can get rid of callbacks.

5.2 Events

There is another abstraction in Node that must be considered. And this is *events*. It's not something that can be implemented via async object abstraction.

Let's look at the most popular example in Node:

```
http.createServer((request, response) => {

  // send back a response every time you get a request

}).listen(8080, '127.0.0.1', () => {
  'server is listening on 127.0.0.1:8080'
});
```

Here method `createServer` uses *request listener* as argument, which actually works like an event: on every *request* it provides a *response*. Unlike simple async call, event is never finished and it's being invoked every time when it's needed.

It can be rewritten in the following declarative way:

```
new LoggedListeningServer(
  new ListeningServer(
    new CreatedServer(
      new RequestResponseEvent()
    ), 8080, '127.0.0.1'
  ), 'server is listening on 127.0.0.1:8080'
).call();
```

As you can see, `RequestResponseEvent` is a node of the async tree that represents request listener, but it's not a simple argument or async object. `RequestResponseEvent` implements `Event` interface and it needs to be treated in a special way, so it requires more flexibility of the whole system.

5.3 Sequence of the async compositions

Sometimes it's not so easy to make a proper composition of async objects(or just any kind of objects), mostly because sometimes we need to do completely different things at different moments. And if you try to combine these things in one async tree, you'll probably fail. So it would be very useful to be able call one async tree after another one. For example,

```
new EqualAssertion(
  new ReadContentFromFile(
    new WrittenFile('./text.txt', 'content')
  ), 'content'
).after(
  RemovedFile('./text.txt')
).call();
```

It's a test that checks that read content from a file is equal to the content that has been written into there. After test the file can be removed to free space on a disk.

Method `after` can be used only once for every async tree:

```
// RIGHT WAY, pseudocode
AsyncTree1().after(
  AsyncTree2().after(
    AsyncTree3().after(...)
  )
)

// WRONG WAY, pseudocode
AsyncTree1().after(
  AsyncTree2()
).after(
  AsyncTree3()
).after(...)
```

5.4 Cache mechanism

Consider the following example with async tree:

```
new SavedNewAccountOfUser(
  new RetrievedUser(userId),
  new RetrievedOldAccountOfUser(
    new RetrievedUser(userId)
  )
).call();
```

So, here we try to save new account for user that based(somehow) on its old one. And as you can see, we *retrieve user* here twice. `RetrievedUser` might be a quite expensive operation, so we don't want to do it more than one time. So, what would do you do here?

Well, you definitely don't want to do something like this:

```
const retrievedUser = new RetrievedUser(userId);
new SavedNewAccountOfUser(
  retrievedUser,
```

```
  new RetrievedOldAccountOfUser(
    retrievedUser
  )
).call();
```

Because it does not change anything. All these objects are asynchronous, they are not simple procedures, and all them will be invoked only when they are needed in the async tree. It means that the results they produce could be received and used only in the inner scope of the tree.

Another thing you must consider here is which of two `RetrievedUser` will be invoked first, so that you can write its result into the cache for using it for the second `RetrievedUser`.

Here *sequence of the async compositions* can help to create declarative construction for caching:

```
new RetrievedUser(userId).as('user')
  .after(
    new SavedNewAccountOfUser(
      as('user'),
      new RetrievedOldAccountOfUser(
        as('user')
      )
    )
  ).call();
```

Every async object can has `as(key)` method, which says to the async object that it must save its represented *value(result)* into the cache with the specified `key`. If `as(key)` method is used as independent(separate) function, it returns async object that represents value from the cache with the specified `key`.

6 Implementation of the Async Tree

6.1 How it works

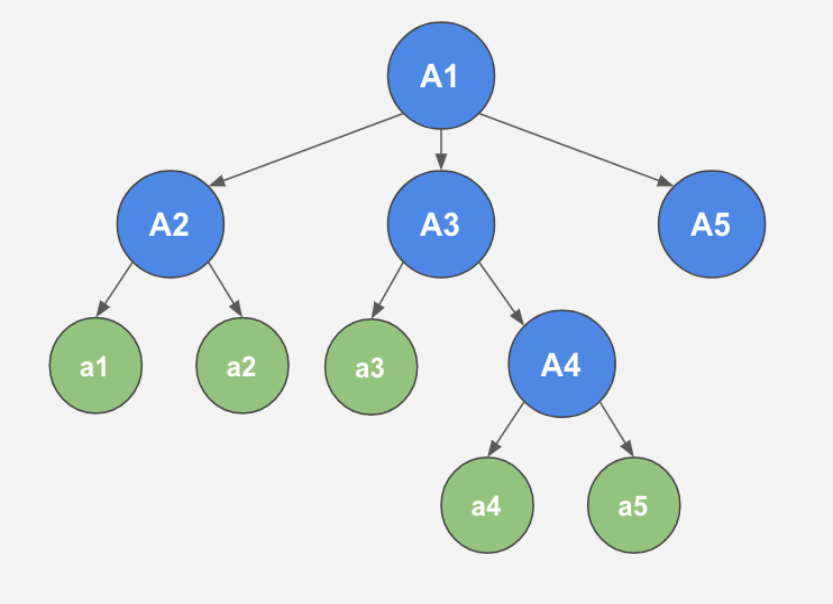
The solution is quite simple. First of all, we need to replace async/sync calls with async objects for creating a composition of the async objects. The composition of these async objects need to be converted to an "asynchronous tree" for making tree traversal from its leaves to the root.

So, let's say we have following composition of async objects:

```
//Pseudocode
A1 (
  A2 (
    a1, a2
  ),
  A3 (
    a3, A4(
      a4, a5
    )
  ),
  A5()
)
```

where `A1`, `A2`, `A3`, `A4`, `A5` are async objects and `a1`, `a2`, `a3`, `a4`, `a5` are just simple arguments.

Then corresponding async tree for this composition would be:



Every node has child nodes as their arguments. So, `a1`, `a2`, `a3`, `a4`, `a5`, `A5` are *leaves* of the tree and they are being called first at the same time. When their results are received, their parents will be ready to be invoked (`a1`, `a2`, `a3`, `a4`, `a5` are already ready in that case, so we just add them to the arguments of their parent).

`A1` is a root of the tree, so we invoke it last. `A2` never waits for the results of `A3` or `A4`, because `A2` just does not need them. But `A3` waits for the result of `A4`, and `A1` waits for the results of `A2`, `A3` and `A5`.

So, the sequence of the calls would look like this:

```
1 a1, a2, a3, a4, a5, A5 // at the same time
2 A2, A4 // at the same time
3 A3
4 A1
```

You might ask *"What if I need to use the result that is represented by the `A1`, how can I do that?"*. Well, it's very simple: you just wrap it with another async object that processes representation of the `A1`.

7 Program implementation

For implementation Aync Tree Pattern we need following classes(abstractions): `AsyncObject`, `AsyncTree`, `TreeNode`, `SimpleTreeNode`, `AsyncTreeNode`,

`NotDefinedAsyncTreeNode`, `Event`, `As` and `NullError`.

7.1 AsyncObject

`AsyncObject` has three types of methods: methods that must be(or can be) implemented by classes that extend it, methods that can be used in declarative composition and methods that are not allowed to be overridden(for internal usage). Also `AsyncObject` has constructor.

Constructor

```
constructor(...args) {
  this.args = args;
  this.tree = new AsyncTree(this);
  this.next;
  this.cache = {};
  this.asKey;
}
```

`...args` can be any type, including `AsyncObject`. Every `AsyncObject` encapsulates `tree` that use it as a root field. Argument `next` points to the next async tree if it set by `after` method(by default it's undefined). Also every async object has `cache` that is simple map. It's needed because cache must be visible in the global scope. Field `asKey` is used as a key for saving representation(computing result) of the async object into the cache if method `as` is invoked.

7.1.1 Methods for implementing

```
definedAsyncCall() {
  throw new Error('asyncCall or syncCall must be defined');
}

definedSyncCall() {
  throw new Error('asyncCall or syncCall must be defined');
}

onError(error) {
  throw error;
}

onResult(result) {
  return result;
}

onErrorAndResult(error, result) {
  return error || result;
}

continueAfterFail() {
  return false;
}
```

```
callbackWithError() {
  return true;
}
```

Either of methods `definedAsyncCall` or `definedSyncCall` must be defined(implemented) in extended classes. If both of them are implemented then only `definedAsyncCall` will be invoked. `definedAsyncCall` is used for async calls, so it must return function of the async call. `definedSyncCall` is used for synchronous processing, so it must return blocking function. The entire logic of using these methods will be shown in the `AsyncTreeNode` class.

Method `onError` is used as a handler for errors that might happen in the async calls. And `onResult` is used for post processing of the result of async/sync call.

If `continueAfterResult` returns true, then `onError` and `onResult` will be ignored, and the represented result(or an error) of async object will be returned by `onErrorAndResult`.

Method `callbackWithError` must return `true` if an error is expected in the callback of corresponding async call, otherwise it must return `false`.

7.1.2 Methods from public API

```
call() {
  this.propagateCache(this);
  this.tree.create().call();
}

after(asyncObject) {
  this.next = asyncObject;
  return this;
}

as(key) {
  this.asKey = key;
  return this;
}
```

Method `call` is being invoked only of async object that is root of async tree. It propagates cache among all fields, so it will be able to access cache in every async object in the composition. The main point of this method is to create an async tree and invoke it.

Method `after` is used for setting next async tree that is being invoked after the current one and returns this async object. Obviously, this method can be used only for the root async object.

Method `as` is used for setting `key` for caching. It returns the async object with set `key`.

7.1.3 Internal methods (not for overriding)

```
iterateArgs(func) {
  this.args.forEach((arg, index) => {
    let isAsync = arg instanceof AsyncObject;
    let isEvent = arg instanceof Event;
    func(arg, index, isAsync, isEvent);
  });
}

hasNoArgs() {
  return this.args.length === 0;
}

readyToBeInvoked(readyResultsNum) {
  return this.args.length === readyResultsNum;
}

callNextTreeIfExists() {
  if (this.next) {
    this.propagateCache(this.next);
    new AsyncTree(this.next).create().call();
  }
}

propagateCache(arg) {
  if (arg instanceof AsyncObject) {
    arg.withCache(this.cache);
    arg.iterateArgs(
      arg => this.propagateCache(arg)
    );
  }
}

withCache(cache) {
  this.cache = cache;
  return this;
 }

saveValueIntoCacheIfNeeded(value) {
  if (this.asKey) {
    this.cache[this.asKey] = value;
  }
  return this;
}
```

Method `iterateArgs` is a proxy method, which avoid using *getters* and *setters* for processing `...args` of the async object.

Method `hasNoArgs` checks if async object encapsulates anything.

Method `readyToBeInvoked` compares number of ready results that are computed by child nodes(`...args`) and number of all `..args`. If they are equal, that means that we can compute result of this async object.

Method `callNextTreeIfExists` is used for calling next async tree if it exists.

Method `propagateCache` is a recursive method that share the cache object among all async objects in the composition(or in the sequence of the

async compositions).

Method `withCache` attaches cache to this async object and returns it.

Mathod `saveValueIntoCacheIfNeeded` save represented value of this async object into the cache, if this object has set `asKey`.

7.2 AsyncTree

We need to convert async composition to the `AsyncTree` to be able to use results of the async objects in this composition that are being computed by the corresponding async/sync calls. `AsyncTree` has constructor, public and private methods.

7.2.1 Constructor

```
constructor(rootField) {
  this.rootField = rootField;
  this.nodes = [];
}
```

Constructor has only one argument - `rootField`, which is `AsyncObject` that wraps all other async objects in the composition. Also constructor encapsulates `nodes`, which is array that can contain elements with `AsyncTreeNode` type and `SimpleTreeNode` type.

7.2.2 Public methods

```
create() {
  this.createAsyncTreeNode(this.rootField, new
      NotDefinedAsyncTreeNode(), 0);
  return this;
}

call() {
  let leaves = this.nodes.filter(node => {
    return node.isLeaf();
  });
  leaves.forEach(leaf => {
    leaf.call();
  });
}
```

Method `create` use private recursive method `createAsyncTreeNode` to create a root that creates other nodes later recursively. The method returns created `AsyncTree`.

Method `call` filters `nodes` to call `leaves` of this async tree. Then these leaves will call other nodes as it explained in the section 6.1.

7.2.3 Private methods

```
createAsyncTreeNode(field, parent, index) {
  let asyncTreeNode = new AsyncTreeNode(field, parent, index);
```

```
  this.nodes.push(asyncTreeNode);
  this.createChildNodes(field, asyncTreeNode);
}

createSimpleTreeNode(field, parent, index) {
  let treeNode = new SimpleTreeNode(field, parent, index);
  this.nodes.push(treeNode);
}

createChildNodes(field, parent) {
  field.iterateArgs((argAsField, index, isAsync, isEvent) => {
    if (isAsync) {
      this.createAsyncTreeNode(argAsField, parent, index);
    } else if (isEvent) {
      this.createSimpleTreeNode((...eventArgs) => {
        argAsField.definedBody(...eventArgs);
      }, parent, index);
    } else {
      this.createSimpleTreeNode(argAsField, parent, index);
    }
  });
}
```

The main point of these private methods to create all `nodes` for this async tree. `nodes` can be two types: `AsyncTreeNode` and `SimpleTreeNode`. `AsyncTreeNode` is used for async objects in the composition that is being converted to the async tree. `SimpleTreeNode` is used for simple objects, primitives and events. Every node encapsulates three arguments: `field` - the element in the async composition that is being wrapped around by this node, `parent` is a node that is parent for this node and `index` that points to the position of this node in the list of child nodes of the parent node.

7.3 TreeNode

`TreeNode` is an abstract class(or interface) that has constructor, methods that must be implemented in the extended classes(`AsyncTreeNode`, `SimpleTreeNode`) and method, which is not allowed to be overridden.

7.3.1 Constructor

```
constructor(field, parent, position) {
  this.field = field;
  this.parent = parent;
  this.position = position;
}
```

`Constructor` has three arguments and they have the same meaning that it's been explained for arguments of `AsyncTreeNode` and `SimpleTreeNode`.

7.3.2 Methods for implementation

```
call(result) {
  throw new Error('call must be overridden');
}

isLeaf() {
  throw new Error('isLeaf must be overridden');
}
```

Method `call` is used for retrieving result for this node and calling parent node if all needed results are ready.

Method `isLeaf` is used for checking if this node is a leaf in the async tree that contains this node.

7.3.3 Internal functionality (not for overriding)

```
callParent(result) {
  this.parent.insertArgumentResult(this.position, result);
  if (this.parent.readyToBeInvoked()) {
    this.parent.call();
  }
}
```

Method `callParent` save result from this node by the position it has. And if all results are set, parent is being invoked.

7.4 AsyncTreeNode

`AsyncTreeNode` is an implementation of the `TreeNode`. And it has some additional public and private methods.

7.4.1 Constructor

```
constructor(field, parent, position) {
  super(field, parent, position);
  this.argResults = [];
  this.readyResultsNum = 0;
}
```

The argument `field` of this class can be only `AsyncObject`. Also constructor stores the results that have been retrieved from the child nodes in the `argResults` array. The parameter `readResultsNum` is number of results that are ready to be used for the parent node.

7.4.2 Public methods

```
call() {
  let args = this.argResults;
  try {
    let asyncCall = this.field.definedAsyncCall();
    if (this.field.callbackWithError()) {
      this.invokeAsyncCallWithError(asyncCall, ...args);
    } else {
      this.invokeAsyncCallWithoutError(asyncCall, ...args);
```

```
    }
  } catch(error) {
    if (error.message !== 'asyncCall or syncCall must be defined')
        {
      this.field.onError(error);
    } else {
      let syncCall = this.field.definedSyncCall();
      this.invokeSyncCall(syncCall, ...args);
    }
  }
}

isLeaf() {
  return this.field.hasNoArgs();
}

readyToBeInvoked() {
  return this.field.readyToBeInvoked(this.readyResultsNum);
}

hasParent() {
  return this.parent instanceof AsyncTreeNode;
}

insertArgumentResult(position, result) {
  this.argResults[position] = result;
  this.readyResultsNum += 1;
}
```

Method `call` checks if async call is defined, if yes then it get result from it. If async call is not defined it checks if sync call is defined and use it for retrieving the result it provides. If some error happens in the definition of the async/sync call, it will be handled by the `onError` method of the async object(it just throws the error by default).

Method `isLeaf` checks if this node is a leaf by the number of arguments of this `field`. If this number is zero, it's a leaf.

Method `readyToBeInvoked` checks if all results from the child nodes are ready by the `readyResultsNum`.

Method `hasParent` checks if this node has a parent node. Only `AsyncTreeNode` can has child node.

Method `insertArgumentResult` inserts a result of one of the child nodes to the `argResults` and increments `readyResultsNum`.

7.4.3 Private methods

```
invokeAsyncCallWithError(asyncCall, ...args) {
  asyncCall(...args, (error, ...results) => {
    if (!this.processedError(error, ...results)) {
      this.processedResult(...results);
    }
  });
}

```

```
invokeAsyncCallWithoutError(asyncCall, ...args) {
  asyncCall(...args, (...results) => {
    this.processedResult(...results);
  });
}

invokeSyncCall(syncCall, ...args) {
  try {
    let syncCallResult = syncCall(...args);
    this.processedResult(syncCallResult);
  } catch (error) {
    this.processedError(error);
  }
}

processedError(error, ...results) {
  let isProcessed = false;
  // It's not possible to get rid of null here :(
  if (error != null) {
    if (this.hasParent()) {
      if (this.field.continueAfterFail()) {
        let totalResult = this.field.onErrorAndResult(error, ...
            results);
        this.field.saveValueIntoCacheIfNeeded(totalResult);
        super.callParent(totalResult);
      } else {
        this.field.onError(error);
      }
    } else {
      if (this.field.continueAfterFail()) {
        let totalResult = this.field.onErrorAndResult(error, ...
            results);
        this.field.saveValueIntoCacheIfNeeded(totalResult);
        this.field.callNextTreeIfExists();
      } else {
        this.field.onError(error);
      }
    }
    isProcessed = true;
  }
  return isProcessed;
}

processedResult(...results) {
  let totalResult;
  if (this.hasParent()) {
    if (this.field.continueAfterFail()) {
      totalResult = this.field.onErrorAndResult(new NullError(),
          ...results);
    } else {
      totalResult = this.field.onResult(...results);
    }
    this.field.saveValueIntoCacheIfNeeded(totalResult);
    super.callParent(totalResult);
  } else {
    if (this.field.continueAfterFail()) {
      totalResult = this.field.onErrorAndResult(new NullError(),
```

```
            ...results);
63      } else {
64        totalResult = this.field.onResult(...results);
65      }
66      this.field.saveValueIntoCacheIfNeeded(totalResult);
67      this.field.callNextTreeIfExists();
68    }
69    return true;
70  }
```

Method `invokeAsyncCallWithError` invokes defined async call of the field, which is an async object with a callback that uses an error argument.

Method `invokeAsyncCallWithoutError` does the same that `invokeAsyncCallWithError` does but it uses a callback without error parameter.

Method `invokeSyncCall` invokes defined sync call of the field, which is an async object(although it defines sync call).

Methods `processedResult` and `processedError` are used for processing and post processing the result or an error from the async/sync call considering the configuration of the field. These methods also decide what to save into the cache if it's needed and invoke next async tree if it exists.

7.5 SimpleTreeNode

`SimpleTreeNode` also implements(or extends) `TreeNode`.

7.5.1 Constructor

```
1  constructor(field, parent, position) {
2    super(field, parent, position);
3  }
```

Unlike `AsyncTreeNode`, `SimpleTreeNode` has a `field` that is just simple object or primitive. And as it's been said before, `parent` is always `AsyncTreeNode`.

7.5.2 Public methods

```
1  call() {
2    super.callParent(this.field);
3  }
4
5  isLeaf() {
6    return true;
7  }
```

The main point of the method `call` is to call parent node, if it's ready to be invoked.

`SimpleTreeNode` is always a leaf because it cannot be a parent node, so

method `isLeaf` always returns `true`.

7.6 NotDefinedAsyncTreeNode

The root of a tree has no parent node and it's not good to use `null` for this declaration. It's better to create an object that indicates that node is not defined.

7.7 Event

`Event` is an interface that provides only one method: `definedBody(...args)` that must be implemented by the extended classes. The main purpose of this interface is to replace functions of events(or listeners). `Event` is not an `AsyncObject`, but it represents function of some event. But you cannot use `AsyncObject` that represents some `Event` instead of the `Event`. In that case you can use `AsyncObject` that represents some function. Actually, you can use a function in the async composition instead of `Event`, but for readability it's better to use `Event`.

```
definedBody(...args) {
  throw new Error('Method definedBody must be overridden with
      arguments of the event/eventListener you call');
}
```

7.8 As

`As` is an `AsyncObject` that represents a result from the cache by the `key`.

7.8.1 Implementation

```
constructor(key) {
  super(key);
}

definedSyncCall() {
  return (key) => {
    let result = this.cache[key];
    if (!result) {
      throw new Error(`There is no value that is cached with key: \
          ${key}`);
    }
    return result;
  }
}
```

Since `cache` is visible among all async objects and `As` is an `AsyncObject`, we can use it in the definition of the sync call. For more readability it's better to use function `as` that returns `As`:

```
module.exports = (key) => {
  return new As(key);
};

//then in the code where it's required:
const as = require('./As');
```

7.9 NullError

This class is for avoiding `null` constant in the code for errors.

```
class NullError extends Error {

  constructor() {
    super('It is a null error');
    this.isNull = true;
  }

}
```

8 How to create AsyncObject

8.1 For async call

Let's take async call `read` from the `fs` module in Node. It has following signature:

```
fs.read(fd, buffer, offset, length, position, (bytesRead, buffer)
    => {
  // handle bytesRead and buffer
});
```

As you can see we have two things to handle in the callback: `bytesRead`, `buffer`. But `AsyncObject` must represent only one object. So you can create an object that contains everything that callback provides: `bytesRead, buffer` or you can choose something that's needed for you. Let's say we need only a `buffer` from the callback, so we would name our `AsyncObject` something like `ReadBufferByFD`.

Let's look on how it must be implemented:

```

const AsyncObject = require('./AsyncObject');
const fs = require('fs');

// Represented result is buffer
class ReadBufferByFD extends AsyncObject {

  constructor(fd, buffer, offset, length, position) {
    super(fd, buffer, offset, length, position);
  }

  definedAsyncCall() {
```

```
    return fs.read;
  }

  onResult(bytesRead, buffer) {
    return buffer;
  }

}
```

First of all, we extends `ReadBufferByFD` from `AsyncObject`. Constructor has the same parameters as corresponding async call has except callback, we don't need to pass it there. Every of these might be or not an `AsyncObject`.

Then we need to define async call, so it must return `fs.read`. It's also possible to write this definition in the explicit way:

```
definedAsyncCall() {
  return (fd, buffer, offset, length, position, callback) => {
    return fs.read(fd, buffer, offset, length, position, callback);
  }
}
```

Unlike in the constructor here parameters `fd`, `buffer`, `offset`, `length`, `position` are definitely ready results(not async objects), so it's possible to use them in the procedural style. Also you need a `callback` parameter here, because it's needed for the async call.

In the method `onResult` (that provides the same that callback of the async call provides) we return `buffer`. So, it means that this async object represents buffer.

8.2 For sync call

Let's now consider a sync variation of the `fs.read`. It's a sync method `fs.readSync`. It has similar signature that `fs.read` has:

```
fs.readSync(fd, buffer, offset, length, position);
```

This sync call returns `bytesRead` and it changes `buffer` that has been passed there. So, we have following implementation:

```
// Represented result is buffer
class ReadBufferByFDSync extends AsyncObject {

  constructor(fd, buffer, offset, length, position) {
    super(fd, buffer, offset, length, position);
  }

  definedSyncCall() {
    return (fd, buffer, offset, length, position) => {
      fs.readSync(fd, buffer, offset, length, position);
      return buffer;
    }
  }
```

```
14
15 }
```

Here we must use explicit definition of the sync call in the `definedSyncCall` method, because we need to return buffer. And we don't need to a callback because it's a synchronous operation. Here we don't need to override `onResult` method, because we override here the sync call that returns what we need.

8.3 Why don't use SyncObject abstraction for the sync calls?

Two reasons. First of them is technical reason: it's just easier to implement `AsyncObject` that can wrap either sync or async call.

Second reason is logical: although `AsyncObject` can wrap sync call, it might require parameters that have been retrieved from the asynchronous operations, so it means that this async object actually has asynchronous nature, because it can be invoked only after some async operations.

More over, even if `AsyncObject` wraps sync call it belongs to some `AsyncTree` or async composition.

9 How to create an Event

Let's say we have a `ReadStream` and we need to be able to attach a 'open' event to it. So, we need to create an async object `ReadStreamWithOpenEvent` that represents `ReadStream` with attached 'open' event.

```
1  // Represented result is a ReadStream
2  class ReadStreamWithOpenEvent extends AsyncObject {
3
4    /*
5      event is an Event with definedBody(fd)
6    */
7    constructor(readStream, event) {
8      super(readStream, event);
9    }
10
11   definedSyncCall() {
12     return (readStream, event) => {
13       readStream.on('open', event);
14       return readStream;
15     }
16   }
17
18 }
```

Actually `readStream` with 'open' event has the following signature:

```
1  readStream.on('open', (fd) => {
2
3      // here we work with fd
4
5  });
```

So, `OpenEvent` would be:

```
class OpenEvent extends Event {

  constructor() {
    super();
  }

  definedBody(fd) {
    // here we work with fd
  }

}
```

As you can see `definedBody` use the same arguments as the event of the `readStream`.

So, in the composition it would look something like this:

```
new ReadStreamWithOpenEvent(
  new CreatedSomeHowReadStream(), new OpenEvent()
).call();
```

10 Example of using the sequence of async compositions and cache mechanism

Let's consider following async object:

```
class MaxNum extends AsyncObject {

  constructor(...args) {
    super(...args);
  }

  definedSyncCall() {
    return (...args) => {
      return Math.max(...args);
    }
  }

}
```

`MaxNum` represent the max of the specified numbers.

And we have async object that wraps `assert.strictEqual` function that asserts that two numbers are equal:

```
class StrictEqualAssertion extends AsyncObject {

  constructor(actual, expected) {
    super(actual, expected);
  }

  definedSyncCall() {
    return (actual, expected) => {
      assert.strictEqual(actual, expected);
```

```
      return actual;
    }
  }

}
```

So, let's have some fun:

```
new MaxNum(
  new MaxNum(1, 2, 4).as('max1'), 5,
  new MaxNum(
   new MaxNum(4, new MaxNum(3, 4, 6)).as('max2'), 7, 8
  ).as('max3')
).after(
  new StrictEqualAssertion(
    new MaxNum(as('max1'), as('max2'), as('max3')), 8
  )
);
```

It's a silly example but it shows the power of the Async Tree Pattern.

11 Conclusion

Async Tree Pattern is very useful design pattern if you care about readability and beauty of your code. It allows to describe the whole program as a tree that contains a lot of little independent pieces, each of them is responsible for something little functionality. So, the whole conception satisfies *single responsibility principle*. Async tree allows to hide the details of how a program works under the cute declarative composition, which is easy to read and maintain.

In this work it's been shown and described how to implement and use Async Tree Pattern. Actually, you don't need to implement this pattern by yourself, it's available in the **Cutie** library(link on this library is in the **References** section).

12 References

References

[1] Yegor Bugayenko: Composable Decorators vs. Imperative Utility Methods, `https://www.yegor256.com/2015/02/26/composable-decorators.html`

[2] Single responsibility principle, `https://en.wikipedia.org/wiki/Single_responsibility_principle`

[3] Yegor Bugayenko: What's Wrong With Object-Oriented Programming?, `https://www.yegor256.com/2016/08/15/what-is-wrong-object-oriented-programming.html`

[4] Yegor Bugayenko: Why NULL is Bad?, `https://www.yegor256.com/2014/05/13/why-null-is-bad.html`

[5] Guseyn Ismayylov: Cutie library: reconsidering the concept of the async objects,
http://guseyn.com/post-reconsidering-async-object-with-cutie

[6] Guseyn Ismayylov: Declarative events,
http://guseyn.com/post-event-new-abstraction-in-cutie

[7] Guseyn Ismayylov: Sequence of async trees (improved cache mechanism),
http://guseyn.com/post-after-conception

[8] Cutie library,
https://github.com/Guseyn/cutie

Contents