



Asynchronous Programming Promise

Junghoo Cho
cho@cs.ucla.edu

Traditional Programming

- Example: Sending a user's picture over network

```
function sendPicture(id) {  
    user = db.find({userid: id});  
    picture = fs.readFile(user.picFile);  
    socket.write(picture);  
    console.log("done!")  
}
```

- Properties
 - Blocking operation in every step: *synchronous API*
 - The program is stuck at every step
- Q: How can the program handle many requests concurrently long waits?

Multi-Threading

- Traditional solution to multiple request processing
 - Create one thread per each request
 - Invoke multiple request handlers in parallel
 - “No change” in coding style
 - Structure of each request handler remains the same
 - Used by most traditional servers, including Apache, Tomcat
- But multi-threading incurs significant resource overhead
 - Memory use (~ 10MB per thread)
 - Thread invocation overhead
 - Concurrency handling logic: semaphore, lock, ...

Single-Threaded JS Engine

- JavaScript runs in a single thread
 - Node.js and browser JavaScript engines
 - Cannot use multi-threading
- Use one thread to handle all requests
 - No need to worry about concurrency
 - More efficient resource usage in principle
 - But potentially long waits at blocking calls

Asynchronous API

- “Nonblocking” API for “multi-processing” under the single-th environment
 - *Do not wait*, return immediately!
 - Invoke callback function when ready
- Example: `db.find({userid: id}, callback);`
 - `db.find()` returns immediately (no blocking)
 - `callback` is invoked when the database object is ready
 - The retrieved object is passed as a parameter to `callback`
 - `callback` can perform actions with the object

Synchronous vs Asynchronous

Synchronous

```
user = db.find({userid: id});
```

- *Wait* until when everything is ready
- Next line in the code has the required object
 - We can do what we logically need to do in the next line
- All logical sequence of actions in one function

Asynchronous

```
db.find({userid: id}, callback);
```

- Return *immediately*, *callback* is called when ready
- Next line in the code does not have the required object
 - We *cannot* do what we logically need to do in the next line
- Actions are spread across multiple callback functions

Callback Hell

```
1 function sendPicture(id) {  
2     db.find({userid: id}, callback1);  
3 }  
4 function callback1(err, user) {  
5     fs.readFile(user.picFile, callback2);  
6 }  
7 function callback2(err, picture) {  
8     socket.write(picture, callback3);  
9 }  
10 function callback3() {  
11     console.log("done!");  
12 }
```

- Difficult to see the logical sequence of actions
 - *Very different* from traditional style of programming

Nested Callback Function

```
1 function sendPicture(id) {  
2     db.find({userid: id}, (err, user) => {  
3         fs.readFile(user.picFile, (err, picture) => {  
4             socket.write(picture, () => {  
5                 console.log("done!");  
6             })  
7         })  
8     })  
9 }
```

- Better, but still ugly, difficult to understand, and easy to make mistakes
- New ECMAScript language constructs
 - Promise (ECMAScript 2015)
 - `async/await` (ECMAScript 2017)
- Most confusing part of this class
 - Pay attention!

Promise (ECMAScript 2015)

```
let prom = db.find({userid: id});  
prom.then(fulfillCallback[, rejectCallback]);
```

- An asynchronous function immediately returns a “promise”
- Once a promise is obtained, callback can be attached using `then`
- The callbacks will be called when the operation is completed
 - If success, `fulfillCallback` is called with “result of operation”
 - If failure, `rejectCallback` is called with “error value”
- Q: How is it better?
 - We are doing the same thing in *two* steps not one! This looks worse!

Promise Chain (1)

```
function sendPicture(id) {  
  let prom1 = db.find({userid: id});  
  let prom2 = prom1.then(user => fs.readFile(user.picFile));  
  let prom3 = prom2.then(image => socket.write(image));  
  
}
```

- `then()` returns a new promise
- We can set a callback to the returned promise
 - `prom2` callback will be called after `prom1` callback is completed
 - `image => socket.write(image)` will be called after `user => fs.readFile(user.picFile)` is completed

Promise Chain (2)

```
function sendPicture(id) {  
  let prom1 = db.find({userid: id});  
  let prom2 = prom1.then(user => fs.readFile(user.picFile));  
  let prom3 = prom2.then(image => socket.write(image));  
  let prom4 = prom3.then(() => console.log("done!"));  
}
```

- We can “chain” a sequence of asynchronous callbacks
 - Promise chain makes code look and work like a synchronous program
 - All logical sequence of actions are in one place
- `sendPicture()` function itself *returns immediately*

Promise Chain (3)

```
function sendPicture(id) {  
  let prom1 = db.find({userid: id});  
  let prom2 = prom1.then(user => fs.readFile(user.picFile));  
  let prom3 = prom2.then(image => socket.write(image));  
  let prom4 = prom3.then(() => console.log("done!"));  
}
```

Or more succinctly,

```
function sendPicture(id) {  
  db.find({userid: id})  
    .then(user => fs.readFile(user.picFile))  
    .then(image => socket.write(image))  
    .then(() => console.log("done!"));  
}
```

Details on Settling Promise (1)

```
let prom1 = db.find({userid: id});  
let prom2 = prom1.then(fulCB1, rejCB1);  
let prom3 = prom2.then(fulCB2, rejCB2);
```

- Terminology: A promise is *settled* either by being *fulfilled* (= resolved) or *rejected*
- Q: How is **prom1** is settled?
- A: Depending on what happens from **db.find()**
 - If success, **prom1** is fulfilled to the output. **fulCB1()** is called with "data"
 - If failure, **prom1** is rejected to error. **rejCB1()** is called with error.

Details on Settling Promise (2)

```
let prom1 = db.find({userid: id});  
let prom2 = prom1.then(fulCB1, rejCB1);  
let prom3 = prom2.then(fulCB2, rejCB2);
```

- Q: How will **prom2** be settled?
- A: Depends on what happens from callbacks (**fulCB1** or **rejCB1**)
- Q: What if the callbacks return a (regular) value?
- A: **prom2** is fulfilled to the value. **fulCB2(value)** is called.
- Q: What if the callbacks throw an error?
- A: **prom2** is rejected to the error. **rejCB2(error)** is called.

Details on Settling Promise (3)

```
let prom1 = db.find({userid: id});  
let prom2 = prom1.then(fulCB1, rejCB1);  
let prom3 = prom2.then(fulCB2, rejCB2);
```

- Q: What if the callbacks return a promise *p*?
 - e.g. `let prom2 = prom1.then(user => fs.readFile(user.picFile, ...))`
- A:
 - If *p* is fulfilled to *value*, *prom2* is fulfilled to *value*. `fulCB2(value)` is called.
 - If *p* is rejected to *error*, *prom2* is rejected to *error*. `rejCB2(error)` is called.

Promise Chain: Rejection Forwarding

```
1 function sendPicture(id) {  
2     db.find({userid: id})  
3     .then(user => fs.readFile(user.picFile))  
4     .then(image => socket.write(image))  
5     .then(() => console.log("done!"))  
6     .catch(errorHandler);  
7 }
```

- Sometimes a promise may be rejected
- Q: What if a promise is rejected, but rejection callback is not
- A: If a rejection is not handled, it is forwarded to the next `then()`
- Setting one rejection callback at the end will be enough
 - No need to set a rejection callback in every `then()`
- `then(null, rejectCB)` can be abbreviated to `catch(rejectCB)`

Error Handling in Promise Callback

- Inside our callback if an error is encountered
 - `throw` an error in the callback, and
 - “catch” it later using `catch()`

- Example

```
db.find({userid: id})
  .then(user => {
    if (!user.picFile) throw new Error("No picture!");
    else return fs.readFile(user.picFile);
  })
  .then(image => socket.write(image))
  ...
  .catch(err => console.log(err.message));
```

- The code looks almost like standard `try` and `catch` block

Guarantees of Promise

- Callbacks added with `then()` even *after* the success/failure of asynchronous operation *will* be called
- Callbacks will *never* be called *before* the completion of the current task of the JavaScript event loop
- The reason for the name “promise”
 - The promise that the async operation will be completed
 - The promise that the correct callback will always be called later

“Promisified” Asynchronous API

- Some APIs have been modified to return a promise if no `callback`
 - e.g., MongoDB node.js driver
- Separate “Promisified” APIs/modules have been created
 - e.g., `require('fs').promises`
- “Promisify” asynchronous API ourself using `util.promisify`

Creating a Promise (1)

- Q: How can we *create* a promise?
- Create a promise that is always resolved (= fulfilled) to **val**:
`Promise.resolve(val)`
- Create a promise that is always rejected to **err**:
`Promise.reject(err)`

Creating a Promise (2)

- Create a promise that is resolved (= fulfilled) to **val** or rejected depending on **cond**:

```
new Promise((resolve, reject) => {  
    ...  
    if (cond) {  
        resolve(val);  
    } else {  
        reject(err);  
    }  
})
```

- Create **Promise** with constructor
- Inside the constructor parameter callback function,
 - Call **resolve(val)** if success
 - Call **reject(err)** if failure

async/await (ECMAScript 2017)

- Syntactic sugar to make async code look almost like a sync c
- Example

```
async function sendPicture(id) {  
  try {  
    user = await db.find({userid: id});  
    picture = await fs.readFile(user.picFile);  
    await socket.write(picture);  
    console.log("done!");  
  } catch (e) {  
    throw new Error("Cannot send the picture!");  
  }  
}
```

- **await** can be used
 - only inside **async** function
 - in front of (function call that returns) promise

async Function

```
async function sendPicture(id) {  
    ...  
    if (cond) {  
        return val;  
    } else {  
        throw new Error("Error!");  
    }  
}
```

- Adding **async** to function declaration “promisifies” the function
 - **async** function returns a promise, not **val** from **return val**
 - If the original function returns a (regular) value, the returned promise resolves to the value.
 - If the original function throws an error, the returned promise is rejected with the error.
- Q: What if the original function returns a promise?

await Keyword

```
user = await db.find({userid: id});
```

- **await** can be used in front of (a function that returns) a promise
 - The next “action” is performed after the promise is fulfilled/rejected
 - If promise is fulfilled, the fulfilled value is returned from await
 - If promise is rejected, an exception is raised (which can be caught with **try/catch**)
- **await** keyword can be used *only inside* **async** function

async/await (1)

```
async function sendPicture(id) {  
  try {  
    user = await db.find({userid: id});  
    picture = await fs.readFile(user.picFile);  
    await socket.write(picture);  
    console.log("done!");  
  } catch (e) {  
    throw new Error("Cannot send the picture!");  
  }  
}
```

- **async/await** makes asynchronous program look almost like synchronous program!
- **await** makes an asynchronous function call “synchronous”
 - The next line is blocked until the function call is completed

async/await (2)

```
async function sendPicture(id) {  
  try {  
    user = await db.find({userid: id});  
    picture = await fs.readFile(user.picFile);  
    await socket.write(picture);  
    console.log("done!");  
  } catch (e) {  
    throw new Error("Cannot send the picture!");  
  }  
}
```

- **async** converts any function to be “asynchronous”
 - The call to `sendPicture()` is *returned immediately* with a promise
- Best of both worlds!
 - We can code `sendPicture()` like a synchronous program, but the call to `sendPicture()` is nonblocking!

await in Top Block (1)

- Q: What if we want to use `await` in the outer most block, not function?

```
user = await db.find({userid: 'john'});  
picture = await fs.readFile(user.picFile);  
await socket.write(picture);  
console.log("done!");
```

- `await` can be used only in a `async` function, but they are not in a fur

await in Top Block (2)

- A: Wrap the outer most block in an anonymous `async` function

```
(async () => {  
  user = await db.find({userid: 'john'});  
  picture = await fs.readFile(user.picFile);  
  await socket.write(picture);  
  console.log("done!");  
})();
```

Parallel await (1)

```
function doubleAfter2Seconds(x) {  
  return new Promise((resolve, reject) => setTimeout(resolve, 2000, x))  
}
```

```
async function addAsync(x) {  
  return await doubleAfter2Seconds(x)  
    + await doubleAfter2Seconds(x)  
    + await doubleAfter2Seconds(x);  
}
```

```
addAsync(10).then(v => console.log(v));
```

- Q: How long will it take to print out the result?

Parallel await (2)

```
async function addAsync(x) {  
  const a = doubleAfter2Seconds(x);  
  const b = doubleAfter2Seconds(x);  
  const c = doubleAfter2Seconds(x);  
  return await a + await b + await c;  
}
```

```
addAsync(10).then(v => console.log(v));
```

- Q: How long will it take?

What We Learned

- Single-threading vs Multi-threading
- Blocking function calls
- Synchronous API vs Asynchronous API
- Nested callbacks (a.k.a. callback hell)
- Promise (ECMAScript 2015)
 - Promise chain
- `async/await` (ECMAScript 2017)

References

- [ECMA-262 promise objects](#)
- [ECMA-262 async](#)

