Chapter 8. Asynchronous Programming, Concurrency, and Parallelism

So far in this book, we've dealt mostly with synchronous programs—programs that take some input, do some stuff, and run to completion in a single pass. But the really interesting programs—the building blocks of real-world applications that make network requests, interact with databases and filesystems, respond to user interaction, offload CPU-intensive work to separate threads—all make use of asynchronous APIs like callbacks, promises, and streams.

These asynchronous tasks are where JavaScript really shines and sets itself apart from other mainstream multithreaded languages like Java and C++. Popular JavaScript engines like V8 and SpiderMonkey do with one thread what traditionally required many threads, by being clever and multiplexing tasks over a single thread while other tasks are idling. This *event loop* is the standard threading model for JavaScript engines, and the one that we'll assume you're using. From an end user's perspective, it usually doesn't matter whether your engine uses an event looped model or a multithreaded one, but it does affect the explanations I'll be giving for how things work and why we design things the way we do.

This event-looped concurrency model is how JavaScript avoids all the common footguns endemic to multithreaded programming, along with the overhead of synchronized data types, mutexes, semaphores, and all the other bits of multithreading jargon. And when you do run JavaScript over multiple threads, it's rare to use shared memory; the typical pattern is to use message passing and to serialize data when sending it between threads. It's a design reminiscent of Erlang, actor systems, and other purely functional concurrency models, and is what makes multithreaded programming in JavaScript foolproof.

That said, asynchronous programming does make programs harder to reason about, because you can no longer mentally trace through a program line by

line; you have to know when to pause and move execution elsewhere, and when to resume again.

TypeScript gives us the tools to reason about asynchronous programs: types let us trace through asynchronous work, and built-in support for `async` / `await` let us apply familiar synchronous thinking to asynchronous programs. We can also use TypeScript to specify strict message-passing protocols for multithreaded programs (it's a lot simpler than it sounds). If all else fails, TypeScript can give you a back rub when your coworker's asynchronous code gets too complicated and you have to stay late debugging it (behind a compiler flag, of course).

But before we get to working with asynchronous programs, let's talk a bit more about how asynchronicity actually works in modern JavaScript engines—how is it that we can suspend and resume execution on what seems to be a single thread?

JavaScript's Event Loop

Let's start with an example. We'll set a couple of timers, one that fires after one millisecond, and the other after two:

```
setTimeout(() => console.info('A'), 1)
setTimeout(() => console.info('B'), 2)
console.info('C')
```

Now, what will get logged to the console? Is it `A`, `B`, `C`?

If you're a JavaScript programmer, you know intuitively the answer is no—the actual firing order is `C`, `A`, then `B`. If you haven't worked with JavaScript or TypeScript before, this behavior might seem mysterious and unintuitive. In reality, it's pretty straightforward; it just doesn't follow the same concurrency model as a `sleep` would in C, or scheduling work in another thread would in Java.

At a high level, the JavaScript VM simulates concurrency like this (see Figure 8-1):

- The main JavaScript thread calls into native asynchronous APIs like `XMLHTTPRequest` (for AJAX requests), `setTimeout` (for sleeping), `readFile` (for reading a file from disk), and so on. These APIs are provided by the JavaScript platform—you can't create them yourself.[1]
- Once you call into a native asynchronous API, control returns to the main thread and execution continues as if the API was never called.
- Once the asynchronous operation is done, the platform puts a *task* in its *event queue*. Each thread has its own queue, used for relaying the results of asynchronous operations back to the main thread. A task includes some metainformation about the call, and a reference to a callback function from the main thread.
- Whenever the main thread's call stack is emptied, the platform will check its event queue for pending tasks. If there's a task waiting, the platform runs it; that triggers a function call, and control returns to that main thread function. When the call stack resulting from that function call is once again empty, the platform again checks the event queue for tasks that are ready to go. This loop repeats until both the call stack and the event queue are empty, and all asynchronous native API calls have completed.

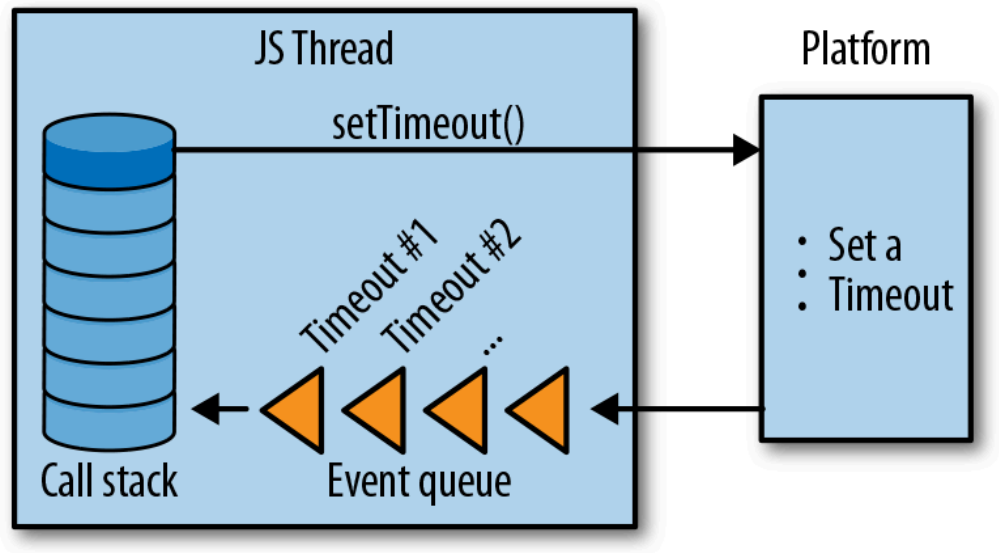

Figure 8-1. JavaScript's event loop: what happens when you call an asynchronous API

Armed with this information, it's time to go back to our `setTimeout` example. Here's what happens:

1. We call `setTimeout`, which calls a native timeout API with a reference to the callback we passed in and the argument `1`.
2. We call `setTimeout` again, which calls the native timeout API again with a reference to the second callback we passed in and the argument `2`.
3. We log `C` to the console.

4. In the background, after at least one millisecond, our JavaScript platform adds a task to its event queue indicating that the timeout for the first `setTimeout` has elapsed, and that its callback is now ready to be called.
5. After another millisecond, the platform adds a second task to the event queue for the second `setTimeout` 's callback.
6. Since the call stack is empty, after step 3 is done the platform looks at its event queue to see if there are any tasks in it. If steps 4 and/or 5 are done, then it will find some tasks. For each task, it will call the corresponding callback function.
7. Once both timers have elapsed and the event queue and call stack are empty, the program exits.

That's why we logged `C` , `A` , `B` , and not `A` , `B` , `C` . With this baseline out of the way, we can start talking about how to type asynchronous code safely.

Working with Callbacks

The basic unit of the asynchronous JavaScript program is the *callback*. A callback is a plain old function that you pass as an argument to another function. As in a synchronous program, that other function invokes your function when it's done doing whatever it does (making a network request, etc.). Callbacks invoked by asynchronous code are just functions, and there's no giveaway in their type signatures that they are invoked asynchronously.

For NodeJS native APIs like `fs.readFile` (used to asynchronously read the contents of a file from disk) and `dns.resolveCname` (used to asynchronously resolve `CNAME` records), the convention for callbacks is that the first parameter is an error or `null` , and the second parameter is a result or `null` .

Here's what `readFile` 's type signature looks like:

```
function readFile(
  path: string,
  options: {encoding: string, flag?: string},
  callback: (err: Error | null, data: string | null) =>
): void
```

Notice that there's nothing special about either `readFile`'s type or `callback`'s type: both are regular JavaScript functions. Looking at the signature, there's no indication that `readFile` is asynchronous and that control will be passed to the next line right after `readFile` is called (not waiting for its result).

NOTE

To run the following example yourself, be sure to first install type declarations for NodeJS:

```
npm install @types/node --save-dev
```

To learn more about third-party type declarations, jump ahead to "JavaScript That Has Type Declarations on DefinitelyTyped".

For example, let's write a NodeJS program that reads and writes to your Apache access log:

```
import * as fs from 'fs'

// Read data from an Apache server's access log
fs.readFile(
  '/var/log/apache2/access_log',
  {encoding: 'utf8'},
  (error, data) => {
    if (error) {
      console.error('error reading!', error)
      return
    }
    console.info('success reading!', data)
  }
)

// Concurrently, write data to the same access log
fs.appendFile(
  '/var/log/apache2/access_log',
  'New access log entry',
  error => {
    if (error) {
      console.error('error writing!', error)
```

```
    }
  })
```

Unless you're a TypeScript or JavaScript engineer and are familiar with how NodeJS's built-in APIs work, and know that they're asynchronous and you can't rely on the order in which API calls appear in your code to dictate in which order filesystem operations actually happen, you wouldn't know that we just introduced a subtle bug where the first `readFile` call may or may not return the access log with our new line appended, depending on how busy the filesystem is at the time this code runs.

You might know that `readFile` is asynchronous from experience, or because you saw it in NodeJS's documentation, or because you know that NodeJS generally sticks to the convention that if a function's last argument is a function that takes two arguments—an `Error | null` and a `T | null`, in that order—then the function is usually asynchronous, or because you ran across the hall to your neighbor for a cup of sugar and ended up staying for a while to chit-chat, then you somehow got on the topic of asynchronous programming in NodeJS and they told you about that time they had a similar issue a couple of months ago and how they fixed it.

Whatever it was, the types certainly didn't help you get there.

Besides the fact that you can't use types to help guide your intuition about the nature of a function's synchronicity, callbacks are also difficult to sequence—which can lead to what some people call "callback pyramids":

```
async1((err1, res1) => {
  if (res1) {
    async2(res1, (err2, res2) => {
      if (res2) {
        async3(res2, (err3, res3) => {
          // ...
        })
      }
    })
  }
})
```

When sequencing operations, you usually want to continue down the chain when an operation succeeds, bailing out as soon as you hit an error. With callbacks, you have to do this manually; when you start accounting for

synchronous errors too (e.g., the NodeJS convention is to `throw` when you give it a badly typed argument, rather than calling your provided callback with an `Error` object), properly sequencing callbacks can get error-prone.

And sequencing is just one kind of operation you might want to run over asynchronous tasks—you might also want to run functions in parallel to know when they're all done, race them to get the result of the first one that finishes, and so on.

This is a limitation of plain old callbacks. Without more sophisticated abstractions for operating on asynchronous tasks, working with multiple callbacks that depend on each other in some way can get messy fast.

To recap:

- Use callbacks to do simple asynchronous tasks.
- While callbacks are great for modeling simple tasks, they quickly get hairy as you try to do things with *lots* of asynchronous tasks.

Regaining Sanity with Promises

Luckily, we're not the first programmers to run into these limitations. In this section we'll develop the concept of *promises*, which are a way to abstract over asynchronous work so that we can compose it, sequence it, and so on. Even if you've worked with promises or futures before, this will be a helpful exercise to understand how they work.

NOTE

Most modern JavaScript platforms include built-in support for promises. In this section we'll develop our own partial `Promise` implementation as an exercise, but in practice, you should use a built-in or off-the-shelf implementation instead. Check whether or not your favorite platform supports promises here, or jump ahead to "lib" to learn more about polyfilling promises on platforms they're not natively supported on.

We'll start with an example of how we want to use `Promise` to first append to a file, then read back the result:

```
function appendAndReadPromise(path: string, data: strir
  return appendPromise(path, data)
    .then(() => readPromise(path))
    .catch(error => console.error(error))
}
```

Notice how there's no callback pyramid here—we've effectively linearized what we want to do into a single, easy-to-understand chain of asynchronous tasks. When one succeeds, the next one runs; if it fails, we skip to the `catch` clause. With a callback-based API, this might have looked more like:

```
function appendAndRead(
  path: string,
  data: string
  cb: (error: Error | null, result: string | null) => v
) {
  appendFile(path, data, error => {
    if (error) {
      return cb(error, null)
    }
    readFile(path, (error, result) => {
      if (error) {
        return cb(error, null)
      }
      cb(null, result)
    })
  })
}
```

Let's design a `Promise` API that lets us do this.

`Promise` starts from humble beginnings:

```
class Promise {
}
```

A `new Promise` takes a function we call an *executor*, which the `Promise` implementation will call with two arguments, a `resolve` function and a `reject` function:

```
type Executor = (
  resolve: Function,
  reject: Function
) => void

class Promise {
  constructor(f: Executor) {}
}
```

How do `resolve` and `reject` work? Let's demonstrate it by thinking about how we would manually wrap a callback-based NodeJS API like `fs.readFile` in a `Promise`-based API. We use NodeJS's built-in `fs.readFile` API like this:

```
import {readFile} from 'fs'

readFile(path, (error, result) => {
  // ...
})
```

Wrapping that API in our `Promise` implementation, it now looks like this:

```
import {readFile} from 'fs'

function readFilePromise(path: string): Promise<string>
  return new Promise((resolve, reject) => {
    readFile(path, (error, result) => {
      if (error) {
        reject(error)
      } else {
        resolve(result)
      }
    })
  })
}
```

So, the type of `resolve`'s parameter depends on which specific API we're using (in this case, its parameter's type would be whatever `result`'s type is), and the type of `reject`'s parameter is always some type of `Error`. Back to our implementation, let's update our code by replacing our unsafe `Function` types with more specific types:

```
type Executor<T, E extends Error> = (
  resolve: (result: T) => void,
  reject: (error: E) => void
) => void
// ...
```

Because we want to be able to get a sense for what type a `Promise` will resolve to just by looking at the `Promise` (for example, `Promise<number>` represents an asynchronous task that results in a `number`), we'll make `Promise` generic, and pass its type parameters down to the `Executor` type in its constructor:

```
// ...
class Promise<T, E extends Error> {
  constructor(f: Executor<T, E>) {}
}
```

So far, so good. We defined `Promise`'s constructor API and understand what the types at play are. Now, let's think about chaining—what are the operations we want to expose to run a sequence of `Promise`s, propagate their results, and catch their exceptions? If you look back to the initial code example at the start of this section, that's what `then` and `catch` are for. Let's add them to our `Promise` type:

```
// ...
class Promise<T, E extends Error> {
  constructor(f: Executor<T, E>) {}
  then<U, F extends Error>(g: (result: T) => Promise<U,
  catch<U, F extends Error>(g: (error: E) => Promise<U,
}
```

`then` and `catch` are two ways to sequence `Promise`s: `then` maps a successful result of a `Promise` to a new `Promise`,[2] and `catch` recovers from a rejection by mapping an error to a new `Promise`.

Using `then` looks like this:

```
let a: () => Promise<string, TypeError> = // ...
let b: (s: string) => Promise<number, never> = // ...
```

```
let c: () => Promise<boolean, RangeError> = // ...

a()
  .then(b)
  .catch(e => c()) // b won't error, so this is if a er
  .then(result => console.info('Done', result))
  .catch(e => console.error('Error', e))
```

Because the type of `b` 's second type argument is `never` (meaning `b` will never throw an error), the first `catch` clause will only get called if `a` errors. But notice that when we use a `Promise` , we don't have to care about the fact that `a` might throw but `b` won't—if `a` succeeds then we map the `Promise` to `b` , and otherwise we jump to the first `catch` clause and map the `Promise` to `c` . If `c` succeeds then we log `Done` , and if it rejects then we `catch` again. This mimics how regular old `try` / `catch` statements work, and does for asynchronous tasks what `try` / `catch` does for synchronous ones (see Figure 8-2).

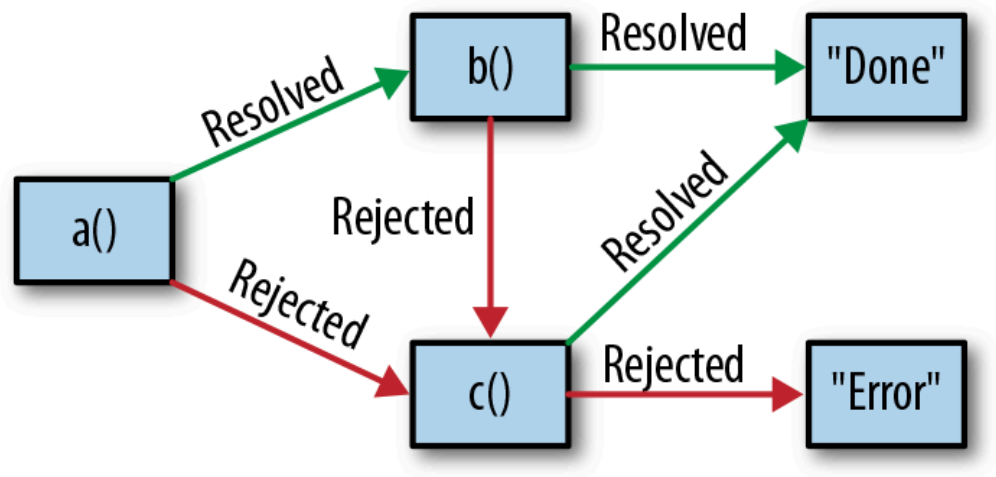

Figure 8-2. The Promise state machine

We also have to handle the case of `Promise` s that throw actual exceptions (as in, `throw Error('foo')`). When we implement `then` and `catch` , we'll do this by wrapping code in `try` / `catch` es and rejecting in the `catch` clause. This does have a few implications, though. It means that:

1. Every `Promise` has the potential to reject, and we can't statically check for this (because TypeScript doesn't support indicating in a function's signature which exceptions the function might throw).
2. A `Promise` won't always be rejected with an `Error` . Because TypeScript has no choice but to inherit JavaScript's behavior, and in JavaScript when you `throw` you can throw anything—a string, a function, an array, a `Promise` , and not necessarily an `Error` —we can't

assume that a rejection will be a subtype of `Error`. It's unfortunate, but this is a sacrifice we'll make in the name of not having to force consumers to `try/catch` every promise chain (which might be spread across multiple files or modules!).

Taking that into account, let's loosen our `Promise` type a bit by not typing errors:

```
type Executor<T> = (
  resolve: (result: T) => void,
  reject: (error: unknown) => void
) => void

class Promise<T> {
  constructor(f: Executor<T>) {}
  then<U>(g: (result: T) => Promise<U>): Promise<U> {
    // ...
  }
  catch<U>(g: (error: unknown) => Promise<U>): Promise<
    // ...
  }
}
```

We now have a fully baked `Promise` interface.

I'll leave it as an exercise for you to hook it all together with implementations for `then` and `catch`. The implementation for `Promise` is notoriously tricky to write correctly—if you're ambitious and have a couple of hours free, head over to the ES2015 specification for a walkthrough of how `Promise`'s state machine should work under the hood.

async and await

Promises are a really powerful abstraction for working with asynchronous code. They're such a popular pattern that they even have their own JavaScript (and therefore, TypeScript) syntax: `async` and `await`. This syntax lets you interact with asynchronous operations the same way you do with synchronous ones.

TIP

Think of `await` as language-level syntax sugar for `.then`. When you `await` a `Promise`, you have to do so in an `async` block. And instead of `.catch`, you can wrap your `await` in a regular `try` / `catch` block.

Let's say you have the following promise (we didn't cover `finally` in the previous section, but it behaves the way you think it would, firing after both `then` and `catch` have a chance to fire):

```
function getUser() {
  getUserID(18)
    .then(user => getLocation(user))
    .then(location => console.info('got location', loca
    .catch(error => console.error(error))
    .finally(() => console.info('done getting location'
}
```

To convert this code to `async` and `await`, first put it in an `async` function, then `await` the promise's result:

```
async function getUser() {
  try {
    let user = await getUserID(18)
    let location = await getLocation(user)
    console.info('got location', user)
  } catch(error) {
    console.error(error)
  } finally {
    console.info('done getting location')
  }
}
```

Since `async` and `await` are JavaScript features, we won't go into them in depth here—suffice it to say that TypeScript has full support for them, and they are completely typesafe. Use them whenever you work with promises, to make it easier to reason about chained operations and avoid lots of `then`s. To learn more about `async` and `await`, head over to their documentation on MDN.

Async Streams

While promises are fantastic for modeling, sequencing, and composing future values, what if you have multiple values, which will become available at multiple points in the future? This is less exotic than it sounds—think bits of a file being read from the filesystem, pixels of a video streaming over the internet from the Netflix server to your laptop, a bunch of keystrokes as you fill out a form, some friends coming over to your house for a dinner party, or votes being deposited into a ballot box throughout the course of Super Tuesday. While these things may sound pretty different on the surface, you can look at them all as asynchronous streams; they are all lists of things where each thing comes in at some point in the future.

There are a few ways to model this, the most common being with an event emitter (like NodeJS's `EventEmitter`) or with a reactive programming library like RxJS.[3] The difference between the two is like the difference between callbacks and promises: events are quick and lightweight, while reactive programming libraries are more powerful, and give you the ability to compose and sequence streams of events.

We'll go over event emitters in the following section. To learn more about reactive programming, head over to the documentation for your favorite reactive programming library—for example, RxJS, MostJS, or xstream.

Event Emitters

At a high level, event emitters offer APIs that support emitting events on a channel and listening for events on that channel:

```
interface Emitter {

  // Send an event
  emit(channel: string, value: unknown): void

  // Do something when an event is sent
  on(channel: string, f: (value: unknown) => void): voi

}
```

Event emitters are a popular design pattern in JavaScript. You might have encountered them when using DOM events, JQuery events, or NodeJS's `EventEmitter` module.

In most languages, event emitters like this one are unsafe. That's because the type of `value` depends on the specific `channel`, and in most languages you can't use types to represent that relationship. Unless your language supports both overloaded function signatures and literal types, you're going to have trouble saying "this is the type of event emitted on this channel." Macros that generate methods to emit events and listen on each channel are a common workaround to this problem, but in TypeScript, you can express this naturally and safely using the type system.

For example, say we're using the NodeRedis client, a Node API for the popular Redis in-memory datastore. It works like this:

```
import Redis from 'redis'

// Create a new instance of a Redis client
let client = redis.createClient()

// Listen for a few events emitted by the client
client.on('ready', () => console.info('Client is ready'
client.on('error', e => console.error('An error occurre
client.on('reconnecting', params => console.info('Recor
```

As programmers using the Redis library, we want to know what types of arguments to expect in our callbacks when we use the `on` API. But because the type of each argument depends on the channel that Redis emits on, a single type won't cut it. If we were the authors of this library, the simplest way to achieve safety would be with an overloaded type:

```
type RedisClient = {
  on(event: 'ready', f: () => void): void
  on(event: 'error', f: (e: Error) => void): void
  on(event: 'reconnecting',
     f: (params: {attempt: number, delay: number}) => \
}
```

This works pretty well, but it's a bit wordy. Let's express it in terms of a mapped type (see "Mapped Types"), pulling out the event definitions into their own type, `Events`:

```
type Events = {                          ❶

  ready: void
  error: Error
  reconnecting: {attempt: number, delay: number}
}

type RedisClient = {                     ❷

  on<E extends keyof Events>(
    event: E,
    f: (arg: Events[E]) => void
  ): void
}
```

❶ We start by defining a single object type that enumerates every event the Redis client might emit, along with the arguments for that event.

❷ We map over our `Events` type, telling TypeScript that `on` can be called with any of the events we defined.

We can then use this type to make the Node–Redis library safer, by typing both of its methods— `emit` and `on` —as safely as possible:

```
// ...
type RedisClient = {
  on<E extends keyof Events>(
    event: E,
    f: (arg: Events[E]) => void
  ): void
  emit<E extends keyof Events>(
    event: E,
    arg: Events[E]
  ): void
}
```

This pattern of pulling out event names and arguments into a shape and mapping over that shape to generate listeners and emitters is common in real-

world TypeScript code. It's also terse, and very safe. When an emitter is typed this way you can't misspell a key, mistype an argument, or forget to pass in an argument. It also serves as documentation for engineers using your code, as their code editors will suggest to them the possible events they might listen on and the types of parameters in those events' callbacks.

EMITTERS IN THE WILD

Using mapped types to build typesafe event emitters is a popular pattern. For example, it's how DOM events are typed in TypeScript's standard library. `WindowEventMap` is a mapping from event name to event type, which the `.addEventListener` and `.removeEventListener` APIs map over to produce better, more specific event types than the default `Event` type:

```
// lib.dom.ts
interface WindowEventMap extends GlobalEventHandlersEve
  // ...
  contextmenu: PointerEvent
  dblclick: MouseEvent
  devicelight: DeviceLightEvent
  devicemotion: DeviceMotionEvent
  deviceorientation: DeviceOrientationEvent
  drag: DragEvent
  // ...
}

interface Window extends EventTarget, WindowTimers, Wir
  WindowLocalStorage, WindowConsole, GlobalEventHandler
  WindowBase64, GlobalFetch {
  // ...
  addEventListener<K extends keyof WindowEventMap>(
    type: K,
    listener: (this: Window, ev: WindowEventMap[K]) =>
    options?: boolean | AddEventListenerOptions
  ): void
  removeEventListener<K extends keyof WindowEventMap>(
    type: K,
    listener: (this: Window, ev: WindowEventMap[K]) =>
    options?: boolean | EventListenerOptions
  ): void
}
```

Typesafe Multithreading

So far, we've been talking about asynchronous programs that you might run on a single CPU thread, a class of programs that most JavaScript and TypeScript programs you'll write will likely fall into. But sometimes, when doing CPU-intensive tasks, you might opt for true parallelism: the ability to split out work across multiple threads, in order to do it faster or to keep your main thread idle and responsive. In this section, we'll explore a few patterns for writing safe, parallel programs in the browser and on the server.

In the Browser: With Web Workers

Web Workers are a widely supported way to do multithreading in the browser. You spin up some workers—special restricted background threads—from the main JavaScript thread, and use them to do things that would have otherwise blocked the main thread and made the UI unresponsive (i.e., CPU-bound tasks). Web Workers are a way to run code in the browser in a truly parallel way; while asynchronous APIs like `Promise` and `setTimeout` run code concurrently, Workers give you the ability to run code in parallel, on another CPU thread. Web Workers can send network requests, write to the filesystem, and so on, with a few minor restrictions.

Because Web Workers are a browser-provided API, its designers put a lot of emphasis on safety—not type safety like we know and love, but *memory safety*. Anyone that's written C, C++, Objective C, or multithreaded Java or Scala knows the pitfalls of concurrently manipulating shared memory. When you have multiple threads reading from and writing to the same piece of memory, it's really easy to run into all sorts of concurrency issues like nondeterminism, deadlocks, and so on.

Because browser code must be particularly safe, and minimize the chances of crashing the browser and causing a poor user experience, the primary way to communicate between the main thread and Web Workers, and between Web Workers and other Web Workers, is with *message passing*.

NOTE

To follow along with the examples in this section, be sure to tell TSC that you're planning to run this code in a browser by enabling the `dom` lib in your *tsconfig.json*:

```
{
  "compilerOptions": {
    "lib": ["dom", "es2015"]
  }
}
```

And for the code that you're running in a Web Worker, use the `webworker` lib:

```
{
  "compilerOptions": {
    "lib": ["webworker", "es2015"]
  }
}
```

If you're using a single *tsconfig.json* for both your Web Worker script and your main thread, enable both at once.

The message passing API works like this. You first spawn a web worker from a thread:

```
// MainThread.ts
let worker = new Worker('WorkerScript.js')
```

Then, you pass messages to that worker:

```
// MainThread.ts
let worker = new Worker('WorkerScript.js')

worker.postMessage('some data')
```

You can pass almost any kind of data to another thread with the `postMessage` API.[4]

The main thread will clone the data you pass before handing it off to the worker thread.[5] On the Web Worker side, you listen to incoming events with

the globally available `onmessage` API:

```
// WorkerScript.ts
onmessage = e => {
  console.log(e.data) // Logs out 'some data'
}
```

To communicate in the opposite direction—from the worker back to the main thread—you use the globally available `postMessage` to send a message to the main thread, and the `.onmessage` method in the main thread to listen for incoming messages. To put it all together:

```
// MainThread.ts
let worker = new Worker('WorkerScript.js')
worker.onmessage = e => {
  console.log(e.data) // Logs out 'Ack: "some data"'
}
worker.postMessage('some data')

// WorkerScript.ts
onmessage = e => {
  console.log(e.data) // Logs out 'some data'
  postMessage(Ack: "${e.data}")
}
```

This API is a lot like the event emitter API we looked at in "Event Emitters". It's a simple way to pass messages around, but without types, we don't know that we've correctly handled all the possible types of messages that might be sent.

Since this API is really just an event emitter, we can apply the same techniques as for regular event emitters to type it. For example, let's build a simple messaging layer for a chat client, which we'll run in a worker thread. The messaging layer will push updates to the main thread, and we won't worry about things like error handling, permissions, and so on. We'll start by defining some incoming and outgoing message types (the main thread sends `Commands` to the worker thread, and the worker thread send `Events` back to the main thread):

```
// MainThread.ts
type Message = string
type ThreadID = number
type UserID = number
type Participants = UserID[]

type Commands = {
  sendMessageToThread: [ThreadID, Message]
  createThread: [Participants]
  addUserToThread: [ThreadID, UserID]
  removeUserFromThread: [ThreadID, UserID]
}

type Events = {
  receivedMessage: [ThreadID, UserID, Message]
  createdThread: [ThreadID, Participants]
  addedUserToThread: [ThreadID, UserID]
  removedUserFromThread: [ThreadID, UserID]
}
```

How could we apply these types to the Web Worker messaging API? The simplest way might be to define a union of all possible message types, then switch on the `Message` type. But this can get pretty tedious. For our `Command` type, it might look something like this:

```
// WorkerScript.ts
type Command =
                          ❶

  | {type: 'sendMessageToThread', data: [ThreadID, Mess
                          ❷

  | {type: 'createThread', data: [Participants]}
  | {type: 'addUserToThread', data: [ThreadID, UserID]}
  | {type: 'removeUserFromThread', data: [ThreadID, Use

onmessage = e =>
                          ❸

  processCommandFromMainThread(e.data)

function processCommandFromMainThread(
                          ❹

  command: Command
) {
```

```
  switch (command.type) {
                          ❺

    case 'sendMessageToThread':
      let [threadID, message] = command.data
      console.log(message)
    // ...
  }
}
```

We define a union of all possible commands that the main thread might send to a worker thread, along with the arguments for each command. ❶

This is just a regular union type. When defining long union types, leading with pipes (`|`) can make those types easier to read. ❷

We take messages sent over the untyped `onmessage` API, and delegate handling them to our typed `processCommandFromMainThread` API. ❸

`processCommandFromMainThread` takes care of handling all incoming messages from the main thread. It's a safe, typed wrapper for the untyped `onmessage` API. ❹

Since the `Command` type is a discriminated union type (see [[discriminated unions]]), we use a `switch` to exhaustively handle every possible type of message the main thread might send our way. ❺

Let's abstract Web Workers' snowflake API behind a familiar `EventEmitter` -based API. That way we can cut down on the verbosity of our incoming and outgoing message types.

We'll start by constructing a typesafe wrapper for NodeJS's `EventEmitter` API (which is available for the browser under the `events` package on NPM):

```
import EventEmitter from 'events'

class SafeEmitter<
  Events extends Record<PropertyKey, unknown[]>
                          ❶

> {
  private emitter = new EventEmitter
                          ❷
```

```
  emit<K extends keyof Events>(
                              ❸
    channel: K,
    ...data: Events[K]
  ) {
    return this.emitter.emit(channel, ...data)
  }
  on<K extends keyof Events>(
                              ❹
    channel: K,
    listener: (...data: Events[K]) => void
  ) {
    return this.emitter.on(channel, listener)
  }
}
```

❶ `SafeEmitter` declares a generic type `Events`, a `Record` mapping from `PropertyKey` (TypeScript's built-in type for valid object keys: `string`, `number`, or `Symbol`) to a list of parameters.

❷ We declare `emitter` as a private member on `SafeEmitter`. We do this instead of extending `SafeEmitter` because our signatures for `emit` and `on` are more restrictive than their overloaded counterparts in `EventEmitter`, and since functions are contravariant in their parameters (remember, for a function `a` to be assignable to another function `b` its parameters have to be supertypes of their counterparts in `b`) TypeScript won't let us declare these overloads.

❸ `emit` takes a `channel` plus arguments corresponding to the list of parameters we defined in the `Events` type.

❹ Similarly, `on` takes a `channel` and a `listener`. `listener` takes a variable number of arguments corresponding to the list of parameters we defined in the `Events` type.

We can use `SafeEmitter` to dramatically cut down on the boilerplate it takes to safely implement a listening layer. On the worker side, we delegate all `onmessage` calls to our emitter and expose a convenient and safe listener API to consumers:

```
// WorkerScript.ts
type Commands = {
```

```
  sendMessageToThread: [ThreadID, Message]
  createThread: [Participants]
  addUserToThread: [ThreadID, UserID]
  removeUserFromThread: [ThreadID, UserID]
}

type Events = {
  receivedMessage: [ThreadID, UserID, Message]
  createdThread: [ThreadID, Participants]
  addedUserToThread: [ThreadID, UserID]
  removedUserFromThread: [ThreadID, UserID]
}

// Listen for events coming from the main thread
let commandEmitter = new SafeEmitter    <Commands>()

// Emit events back to the main thread
let eventEmitter = new SafeEmitter      <Events>()

// Wrap incoming commands from the main thread
// using our typesafe event emitter
onmessage = command =>
  commandEmitter.emit(
    command.data.type,
    ...command.data.data
  )

// Listen for events issued by the worker, and send the
eventEmitter.on('receivedMessage', data =>
  postMessage({type: 'receivedMessage', data})
)
eventEmitter.on('createdThread', data =>
  postMessage({type: 'createdThread', data})
)
// etc.

// Respond to a sendMessageToThread command from the ma
commandEmitter.on('sendMessageToThread', (threadID, mes
  console.log(OK, I will send a message to threadID ${t
)

// Send an event back to the main thread
eventEmitter.emit('createdThread', 123, [456, 789])
```

On the flip side, we can also use an `EventEmitter`-based API to send commands back from the main thread to the worker thread. Note that if you use this pattern in your own code, you might consider using a more full-featured emitter (like Paolo Fragomeni's excellent `EventEmitter2`) that supports wildcard listeners, so you don't have to manually add a listener for each type of event:

```
// MainThread.ts
type Commands = {
  sendMessageToThread: [ThreadID, Message]
  createThread: [Participants]
  addUserToThread: [ThreadID, UserID]
  removeUserFromThread: [ThreadID, UserID]
}

type Events = {
  receivedMessage: [ThreadID, UserID, Message]
  createdThread: [ThreadID, Participants]
  addedUserToThread: [ThreadID, UserID]
  removedUserFromThread: [ThreadID, UserID]
}

let commandEmitter = new SafeEmitter    <Commands>()
let eventEmitter = new SafeEmitter      <Events>()

let worker = new Worker('WorkerScript.js')

// Listen for events coming from our worker,
// and re-emit them using our typesafe event emitter
worker.onmessage = event =>
  eventEmitter.emit(
    event.data.type,
    ...event.data.data
  )

// Listen for commands issues by this thread, and send
commandEmitter.on('sendMessageToThread', data =>
  worker.postMessage({type: 'sendMessageToThread', data
)
commandEmitter.on('createThread', data =>
  worker.postMessage({type: 'createThread', data})
)
// etc.
```

```
// Do something when the worker tells us a new thread w
eventEmitter.on('createdThread', (threadID, participant
  console.log('Created a new chat thread!', threadID, p
)

// Send a command to our worker
commandEmitter.emit('createThread', [123, 456])
```

That's it! We've created a simple typesafe wrapper for the familiar event emitter abstraction that we can use in a variety of settings, from cursor events in a browser to communication across threads, making passing messages between threads safe. This is a common pattern in TypeScript: even if something is unsafe, you can usually wrap it in a typesafe API.

Typesafe protocols

So far, we've looked at passing messages back and forth between two threads. What would it take to extend the technique to say that a particular command always receives a specific event as a response?

Let's build a simple call-response protocol, which we can use to move function evaluation across threads. We can't easily pass functions between threads, but we can define functions in a worker thread and send arguments to them, then send results back. For example, let's say we're building a matrix math engine that supports three operations: finding the determinant of a matrix, computing the dot product of two matrices, and inverting a matrix.

You know the drill—let's start by sketching out the types for these three operations:

```
type Matrix = number[][]

type MatrixProtocol = {
  determinant: {
    in: [Matrix]
    out: number
  }
  'dot-product': {
    in: [Matrix, Matrix]
    out: Matrix
  }
  invert: {
```

```
    in: [Matrix]
    out: Matrix
  }
}
```

We define matrices in our main thread, and run all computations in workers. Once again, the idea is to wrap an unsafe operation (sending and receiving untyped messages from a worker) with a safe one, exposing a well-defined, typed API for consumers to use. In this naive implementation, we start by defining a simple request-response protocol `Protocol`, which lists out the operations a worker can perform along with their expected input and output types.[6] We then define a generic `createProtocol` function that takes a `Protocol` and a file path to a Worker, and returns a function that takes a `command` in that protocol and returns a final function that we can call to actually evaluate that `command` for a specific set of arguments. OK, here we go:

```
type Protocol = {
                                    ❶
  [command: string]: {
    in: unknown[]
    out: unknown
  }
}

function createProtocol<P extends Protocol>(script: str
                                    ❷
 return <K extends keyof P>(command: K) =>
                                    ❸
   (...args: P[K]['in']) =>
                                    ❹
    new Promise<P[K]['out']>((resolve, reject) => {
                                    ❺
      let worker = new Worker(script)
      worker.onerror = reject
      worker.onmessage = event => resolve(event.data.da
      worker.postMessage({command, args})
    })
}
```

❶

We start by defining a general-purpose `Protocol` type that is not specific to our `MatrixProtocol`.

When we call `createProtocol`, we pass in a file path to a worker `script`, along with a specific `Protocol`.

`createProtocol` returns an anonymous function that we can then invoke with a `command`, which is a key in the `Protocol` we bound in ❷.

We then call that function with whatever the specific `in` type is for the command we passed in in ❸.

This gives us back a `Promise` for the specific `out` type for that command, as defined in our particular protocol. Note that we have to explicitly bind a type parameter to `Promise`, otherwise it defaults to `{}`.

Now let's apply our `MatrixProtocol` type plus the path to our Web Worker script to `createProtocol` (we won't get into the nitty-gritty of how to compute a determinant, and I'll assume that you've implemented it in *MatrixWorkerScript.ts*). We'll get back a function that we can use to run a specific command in that protocol:

```
let runWithMatrixProtocol = createProtocol<MatrixProtoc
  'MatrixWorkerScript.js'
)
let parallelDeterminant = runWithMatrixProtocol('determ

parallelDeterminant([[1, 2], [3, 4]])
  .then(determinant =>
    console.log(determinant) // -2
  )
```

Cool, huh? We've taken something totally unsafe—untyped message passing between threads—and abstracted over it with a fully typesafe request-response protocol. All the commands you can run using that protocol live in one place (`MatrixProtocol`), and our core logic (`createProtocol`) lives separately from our concrete protocol implementation (`runWithMatrixProtocol`).

Anytime you need to communicate between two processes—whether on the same machine or between multiple computers on a network—typesafe protocols are a great tool to make that communication safe. While this section helped develop some intuition for what problems protocols solve, for a real-world application you'll likely want to reach for an existing tool like Swagger, gRPC, Thrift, or GraphQL—for an overview, head over to "Typesafe APIs".

In NodeJS: With Child Processes

NOTE

To follow along with the examples in this section, be sure to install type declarations for NodeJS from NPM:

```
npm install @types/node --save-dev
```

To learn more about using type declarations, jump ahead to "JavaScript That Has Type Declarations on DefinitelyTyped".

Typesafe parallelism in NodeJS works the same way as it does for Web Worker threads in the browser (see "Typesafe protocols"). While the message-passing layer itself is unsafe, it's easy to build a typesafe API over it. NodeJS's child process API looks like this:

```
// MainThread.ts
import {fork} from 'child_process'

let child = fork('./ChildThread.js')
                                    ❶


child.on('message', data =>
                                    ❷
  console.info('Child process sent a message', data)
)

child.send({type: 'syn', data: [3]})
                                    ❸
```

❶ We use NodeJS's `fork` API to spawn a new child process.

❷ We listen to incoming messages from a child process using the `on`

❷ API. There are a few messages that a NodeJS child process might send to its parent; here, we just care about the `'message'` message.

We use the `send` API to send messages to a child process. ❸

In our child thread, we listen to messages coming in from the main thread using the `process.on` API and send messages back with `process.send`:

```
// ChildThread.ts
process.on('message', data =>                    ❶

  console.info('Parent process sent a message', data)
)

process.send({type: 'ack', data: [3]})           ❷
```

We use the `on` API on the globally defined `process` to listen for incoming messages from a parent thread. ❶

We use the `send` API on `process` to send messages to the parent process.

Because the mechanics are so similar to Web Workers, I'll leave it as an exercise to implement a typesafe protocol to abstract over interprocess communication in NodeJS.

Summary

In this chapter we started with the basics of JavaScript's event loop, and continued on to a discussion of the building blocks of asynchronous code in JavaScript and how to safely express them in TypeScript: callbacks, promises, `async` / `await`, and event emitters. We then covered multithreading, exploring passing messages between threads (in the browser and on the server) and building full protocols for communicating between threads.

As with Chapter 7, which technique you use is up to you:

- For simple asynchronous tasks, callbacks are as straightforward as it gets.
- For more complex tasks that need to be sequenced and parallelized, promises and `async` / `await` are your friend.

- When a promise doesn't cut it (e.g., if you're firing an event multiple times), reach for event emitters or a reactive streams library like RxJS.
- To extend these techniques to multiple threads, use event emitters, typesafe protocols, or typesafe APIs (see "Typesafe APIs").

Exercises

1. Implement a general-purpose `promisify` function, which takes any function that takes exactly one argument and a callback and wraps it in a function that returns a promise. When you're done, you should be able to use `promisify` like this (install type declarations for NodeJS first, with `npm install @types/node --save-dev`):

```
import {readFile} from 'fs'

let readFilePromise = promisify(readFile)
readFilePromise('./myfile.ts')
  .then(result => console.log('success reading file'
  .catch(error => console.error('error reading file'
```

2. In the section on "Typesafe protocols" we derived one half of a protocol for typesafe matrix math. Given this half of the protocol that runs in the main thread, implement the other half that runs in a Web Worker thread.
3. Use a mapped type (as in "In the Browser: With Web Workers") to implement a typesafe message-passing protocol for NodeJS's `child_process`.

1 Well, you can if you fork your browser platform, or build a C++ NodeJS extension.

2 Eagle-eyed readers will notice how similar this API is to the `flatMap` API we developed in "The Option Type". That similarity is no accident! Both `Promise` and `Option` are inspired by the Monad design pattern popularized by the functional programming language Haskell.

3 `Observables` are the basic building block of reactive programming's approach to doing things to values over time. There's an in-progress proposal to standardize `Observables` in the `Observable` proposal. Look forward to a deeper dive into

`Observables` in a future edition of this book, once the proposal is more broadly adopted by JavaScript engines.

4 Except for functions, errors, DOM nodes, property descriptors, getters and setters, and prototype methods and properties. For more information, head over to the HTML5 specification.

5 You can also use the `Transferable` API to pass certain types of data (like `ArrayBuffer`) between threads by reference. In this section we won't be using `Transferable` to explicitly transfer object ownership across threads, but that's an implementation detail. If you use `Transferable` for your use case, the approach is identical from a type safety point of view.

6 This implementation is naive because it spawns a new worker every time we issue a command; in the real world, you probably want to have a pooling mechanism that keeps a warm pool of workers around, and recycles freed workers.