PROGRAMMATION ASYNCHRONE

> “Qui peut attendre tranquillement pendant que la boue s'installe?
> Qui peut rester immobile jusqu'au moment d'agir?”
>
> —Laozi, *Tao Te Ching*

La partie centrale d'un ordinateur, la partie qui effectue les différentes étapes qui composent nos programmes, s'appelle le *processeur* . Les programmes que nous avons vus jusqu'à présent sont des éléments qui occuperont le processeur jusqu'à la fin de son travail. La vitesse à laquelle une boucle manipulant des nombres peut être exécutée dépend en grande partie de la vitesse du processeur.

Mais de nombreux programmes interagissent avec des éléments extérieurs au processeur. Par exemple, ils peuvent communiquer sur un réseau informatique ou demander des données du disque dur, ce qui est beaucoup plus lent que de les récupérer à partir de la mémoire.

Dans ce cas, il serait dommage de laisser le processeur inactif - il pourrait peut-être effectuer un autre travail entre-temps. Ceci est en partie géré par votre système d'exploitation, qui basculera le processeur entre plusieurs programmes en cours

d'exécution. Mais cela n’aide en rien lorsque nous voulons qu’un *seul* programme puisse progresser en attendant une requête réseau.

Asynchronicité

Dans un modèle de programmation *synchrone* , les choses se passent une par une. Lorsque vous appelez une fonction qui exécute une action de longue durée, elle ne revient que lorsque l'action est terminée et peut renvoyer le résultat. Cela arrête votre programme pour le temps que l'action prend.

Un modèle *asynchrone* permet à plusieurs choses de se produire en même temps. Lorsque vous démarrez une action, votre programme continue de s'exécuter. Lorsque l'action est terminée, le programme en est informé et obtient l'accès au résultat (par exemple, les données lues sur le disque).

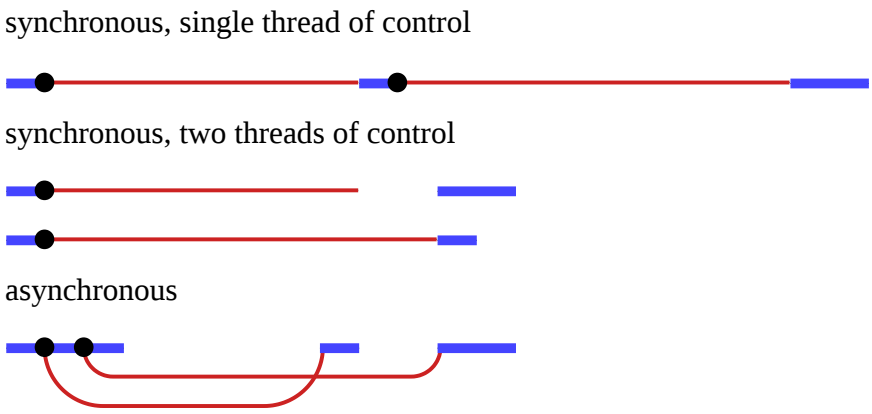
Nous pouvons comparer la programmation synchrone et asynchrone en utilisant un petit exemple: un programme qui extrait deux ressources du réseau, puis combine les résultats.

Dans un environnement synchrone, où la fonction de requête ne revient qu'après avoir effectué son travail, le moyen le plus simple d'effectuer cette tâche consiste à effectuer les requêtes les unes après les autres. Cela a l’inconvénient que la deuxième demande ne sera lancée que lorsque la première sera terminée. Le temps total pris sera au moins égal à la somme des deux temps de réponse.

La solution à ce problème, dans un système synchrone, consiste à démarrer des threads de contrôle supplémentaires. Un *thread* est un autre programme en cours d'exécution dont le système d'exploitation peut imbriquer d'autres programmes. Comme la plupart des ordinateurs modernes contiennent plusieurs processeurs, plusieurs threads peuvent même être exécutés simultanément, sur différents processeurs. Un deuxième thread peut démarrer la deuxième requête, puis les deux attendent le retour de leurs résultats, après quoi ils se resynchronisent pour combiner leurs résultats.

Dans le diagramme suivant, les lignes épaisses représentent le temps passé normalement par le programme et les lignes fines représentent le temps d'attente du réseau. Dans le modèle synchrone, le temps pris par le réseau fait *partie* du scénario d'un thread de contrôle donné. Dans le modèle asynchrone, le démarrage d'une action

réseau provoque conceptuellement une *scission* dans la chronologie. Le programme qui a initié l'action continue à s'exécuter et l'action se déroule parallèlement, en avertissant le programme lorsqu'il est terminé.



Une autre façon de décrire la différence est que l'attente de la fin des actions est *implicite* dans le modèle synchrone, alors qu'elle est *explicite* , sous notre contrôle, dans le modèle asynchrone.

Asynchronicity va dans les deux sens. Il facilite l’expression de programmes ne correspondant pas au modèle de contrôle linéaire, mais il peut également rendre plus difficile l’expression de programmes qui suivent une ligne droite. Nous verrons quelques moyens de remédier à cette maladresse plus loin dans le chapitre.

Les deux plates-formes de programmation JavaScript importantes, les navigateurs et Node.js, permettent des opérations pouvant prendre un certain temps de manière asynchrone, plutôt que de s'appuyer sur des threads. Comme la programmation avec des threads est notoirement difficile (comprendre ce que fait un programme est beaucoup plus difficile lorsqu'il fait plusieurs choses à la fois), cela est généralement considéré comme une bonne chose.

Tech Crow

La plupart des gens savent que les corbeaux sont des oiseaux très intelligents. Ils peuvent utiliser des outils, planifier à l'avance, se souvenir de certaines choses et même les communiquer entre eux.

Ce que la plupart des gens ne savent pas, c'est qu'ils sont capables de beaucoup de choses qu'ils nous cachent bien. Un expert réputé (bien qu’excent quelque peu

excentrique) des corvidés me dit que la technologie de corbeau n’est pas loin derrière la technologie humaine et qu’elle rattrape son retard.

Par exemple, de nombreuses cultures de corbeaux ont la capacité de construire des dispositifs informatiques. Celles-ci ne sont pas électroniques, comme le sont les dispositifs informatiques, mais fonctionnent grâce aux actions d'insectes minuscules, une espèce étroitement liée au termite, qui a développé une relation symbiotique avec les corbeaux. Les oiseaux leur fournissent de la nourriture et, en retour, les insectes construisent et exploitent leurs colonies complexes qui, avec l'aide des créatures vivantes qui s'y trouvent, effectuent des calculs.

Ces colonies sont généralement situées dans de grands nids de longue vie. Les oiseaux et les insectes travaillent ensemble pour construire un réseau de structures en argile bulbeuse, cachées entre les brindilles du nid, dans lesquelles les insectes vivent et travaillent.

Pour communiquer avec d'autres appareils, ces machines utilisent des signaux lumineux. Les corbeaux incorporent des morceaux de matériau réfléchissant dans des tiges de communication spéciales et les insectes visent à refléter la lumière sur un autre nid, en codant les données sous forme d'une séquence de clignotements rapides. Cela signifie que seuls les nids avec une connexion visuelle ininterrompue peuvent communiquer.

Notre ami l'expert corvide a cartographié le réseau de nids de corbeaux dans le village de Hières-sur-Amby, sur les rives du Rhône. Cette carte montre les nids et leurs connexions:

Dans un exemple stupéfiant d’évolution convergente, les ordinateurs corbeaux utilisent JavaScript. Dans ce chapitre, nous allons écrire quelques fonctions réseau de base pour eux.

RAPPELS

Une approche de la programmation asynchrone consiste à faire en sorte que les fonctions effectuant une action lente prennent un argument supplémentaire, une *fonction de rappel* . L'action est lancée et lorsqu'elle est terminée, la fonction de rappel est appelée avec le résultat.

Par exemple, la `setTimeout` fonction, disponible à la fois dans Node.js et dans les navigateurs, attend un nombre donné de millisecondes (une seconde, mille millisecondes), puis appelle une fonction.

```
setTimeout (() =>  console . log ( "Coche
```

éditer et exécuter le code en cliquant dessu

L'attente n'est généralement pas un type de travail très important, mais elle peut être utile pour mettre à jour une animation ou pour vérifier si quelque chose prend plus de temps.

L'exécution de plusieurs actions asynchrones dans une ligne à l'aide de callbacks signifie que vous devez continuer à transmettre de nouvelles fonctions pour gérer la poursuite du calcul après les actions.

La plupart des ordinateurs à nid-de-pie ont un bulbe de stockage de données à long terme, dans lequel des éléments d’information sont gravés dans des brindilles afin de pouvoir les récupérer ultérieurement. La gravure, ou la recherche d'une donnée, prenant un moment, l'interface de stockage à long terme est asynchrone et utilise des fonctions de rappel.

Les ampoules de stockage stockent des morceaux de données encodables JSON sous des noms. Un corbeau peut stocker des informations sur les endroits où se trouvent des aliments cachés sous le nom `"food caches"`, ce qui peut contenir un tableau de noms pointant vers d'autres éléments de données, décrivant le cache réel. Pour rechercher une cache de nourriture dans les ampoules de stockage du nid *Big Oak* , un corbeau pourrait exécuter le code suivant:

```
importer { bigOak } de  "./crow-tech" ;

bigOak . readStorage ( "caches de nourriture" , caches  => {
   let  firstCache  =  caches [ 0 ];
   bigOak . readStorage ( firstCache , info  => {
     console . log ( info );
  });
});
```

(Tous les noms et chaînes de reliure ont été traduits de la langue du corbeau en anglais.)

Ce style de programmation est réalisable, mais le niveau d'indentation augmente avec chaque action asynchrone, car vous vous retrouvez dans une autre fonction. Faire des choses plus compliquées, telles que l'exécution de plusieurs actions en même temps, peut devenir un peu gênant.

Les ordinateurs imbriqués Crow sont conçus pour communiquer à l'aide de paires demande-réponse. Cela signifie qu'un nid envoie un message à un autre, qui le renvoie immédiatement, en confirmant la réception et éventuellement en répondant à une question posée dans le message.

Chaque message est étiqueté avec un *type* , qui détermine comment il est traité. Notre code peut définir des gestionnaires pour des types de demandes spécifiques et, lorsqu'une telle demande arrive, le gestionnaire est appelé pour produire une réponse.

L'interface exportée par le module fournit des fonctions de communication basées sur le rappel. Les nids ont une méthode qui envoie une demande. Il attend le nom du nid cible, le type de la demande et le contenu de la demande comme ses trois premiers arguments, ainsi qu'une fonction à appeler lorsqu'une réponse arrive en tant que quatrième et dernier argument. `"./crow-tech"` `send`

```
bigOak . send ( "Cow Pasture" , "note" , "Caw bruyant à 19 heures
            () =>  console . log ( "Note livrée." ));
```

Mais pour que les nids soient capables de recevoir cette requête, nous devons d'abord définir un type de requête nommé `"note"`. Le code qui gère les demandes doit être exécuté non seulement sur cet ordinateur de nid, mais sur tous les nids pouvant

recevoir des messages de ce type. Nous supposerons simplement qu'un corbeau survole et installe notre code de gestionnaire sur tous les nids.

```
import { defineRequestType } from  "./crow-tech" ;

defineRequestType ( "note" , ( imbriqué , contenu , source , term
   console . log ( `$ { imb . nom }  reçu note: $ { contenu } ` )
   done ();
});
```

La `defineRequestType` fonction définit un nouveau type de requête. L'exemple ajoute la prise en charge des `"note"` demandes, qui envoie simplement une note à un nid donné. Notre mise en œuvre appelle `console.log` pour que nous puissions vérifier que la demande est arrivée. Les nids ont une `name` propriété qui porte leur nom.

Le quatrième argument donné au gestionnaire,, `done` est une fonction de rappel qu'il doit appeler lorsque cela est fait avec la demande. Si nous avions utilisé la valeur de retour du gestionnaire comme valeur de réponse, cela signifierait qu'un gestionnaire de demandes ne peut pas effectuer lui-même d'actions asynchrones. Une fonction effectuant un travail asynchrone est généralement renvoyée avant le travail, après s'être organisée pour qu'un rappel soit appelé à la fin. Il nous faut donc un mécanisme asynchrone, dans ce cas une autre fonction de rappel, pour signaler quand une réponse est disponible.

En un sens, l'asynchronicité est *contagieuse* . Toute fonction qui appelle une fonction qui fonctionne de manière asynchrone doit elle-même être asynchrone, en utilisant un mécanisme de rappel ou un mécanisme similaire pour fournir son résultat. Appeler un rappel est un peu plus compliqué et source d'erreurs que de simplement renvoyer une valeur. Il n'est donc pas génial de devoir structurer de grandes parties de votre programme de cette façon.

Promesses

Travailler avec des concepts abstraits est souvent plus facile lorsque ces concepts peuvent être représentés par des valeurs. Dans le cas d'actions asynchrones, vous pouvez, au lieu d'organiser ultérieurement l'appel d'une fonction, renvoyer un objet qui représente cet événement futur.

C'est ce que la classe standard `Promise` est pour. Une *promesse* est une action asynchrone qui peut se terminer à un moment donné et produire une valeur. Il est capable d'avertir toute personne intéressée lorsque sa valeur est disponible.

Le moyen le plus simple de créer une promesse est d'appeler `Promise.resolve`. Cette fonction garantit que la valeur que vous lui donnez est encapsulée dans une promesse. Si c'est déjà une promesse, elle est simplement retournée - sinon, vous obtenez une nouvelle promesse qui se termine immédiatement avec votre valeur comme résultat.

```
laisser  quinze  =  promesse . résoudre ( 15 );
quinze . then ( valeur  =>  console . log ( `Got $ { valeur } ` )
// → 15
```

Pour obtenir le résultat d'une promesse, vous pouvez utiliser sa `then` méthode. Ceci enregistre une fonction de rappel à appeler lorsque la promesse se résout et produit une valeur. Vous pouvez ajouter plusieurs rappels à une même promesse et ils seront appelés, même si vous les ajoutez après que la promesse soit déjà *résolue* (terminée).

Mais ce n'est pas tout ce que fait la `then` méthode. Il retourne une autre promesse, qui résout la valeur renvoyée par la fonction de gestionnaire ou, si cela retourne une promesse, attend cette promesse et se résout ensuite en résultat.

Il est utile de considérer les promesses comme un moyen de transférer des valeurs dans une réalité asynchrone. Une valeur normale est simplement là. Une valeur promise est une valeur qui *pourrait* déjà être là ou pourrait apparaître à un moment donné dans l'avenir. Les calculs définis en termes de promesses agissent sur ces valeurs encapsulées et sont exécutés de manière asynchrone à mesure que les valeurs deviennent disponibles.

Pour créer une promesse, vous pouvez l'utiliser en `Promise` tant que constructeur. Son interface est quelque peu étrange: le constructeur attend une fonction en tant qu'argument, qu'il appelle immédiatement, en lui transmettant une fonction qu'il peut utiliser pour résoudre la promesse. Cela fonctionne ainsi, au lieu par exemple avec une `resolve` méthode, de sorte que seul le code qui a créé la promesse puisse la résoudre.

Voici comment créer une interface à base de promesse pour la `readStorage` fonction:

```
fonction de  stockage ( nid , nom ) {
   retour  nouvelle  promesse ( résolution  => {
     nid . readStorage ( nom , résultat  =>  détermination ( résu
  });
}

stockage ( bigOak , "ennemis" )
  . then ( valeur  =>  console . log ( "Got" , valeur ));
```

Cette fonction asynchrone renvoie une valeur significative. C'est le principal avantage des promesses: elles simplifient l'utilisation de fonctions asynchrones. Au lieu de devoir passer des rappels, les fonctions basées sur les promesses ressemblent aux fonctions classiques: elles prennent les arguments comme arguments et renvoient leur résultat. La seule différence est que la sortie peut ne pas être encore disponible.

Échec

Les calculs JavaScript classiques peuvent échouer en lançant une exception. Les calculs asynchrones ont souvent besoin de quelque chose comme ça. Une requête réseau peut échouer ou un code faisant partie du calcul asynchrone peut générer une exception.

L'un des problèmes les plus pressants avec le style de rappel de la programmation asynchrone est qu'il est extrêmement difficile de s'assurer que les échecs sont correctement signalés aux rappels.

Une convention largement utilisée est que le premier argument du rappel est utilisé pour indiquer que l'action a échoué, et que le second contient la valeur produite par l'action lorsqu'elle a réussi. Ces fonctions de rappel doivent toujours vérifier si elles ont reçu une exception et s'assurer que tous les problèmes qu'elles provoquent, y compris les exceptions générées par les fonctions qu'ils appellent, sont interceptées et affectées à la fonction appropriée.

Les promesses facilitent les choses. Ils peuvent être soit résolus (l'action s'est terminée avec succès), soit rejetés (elle a échoué). Les gestionnaires de résolution (enregistrés

avec `then`) ne sont appelés que lorsque l'action réussit et les refus sont automatiquement propagés à la nouvelle promesse renvoyée par `then`. Et lorsqu'un gestionnaire lève une exception, la promesse générée par son `then` appel est automatiquement rejetée. Ainsi, si un élément d'une chaîne d'actions asynchrones échoue, le résultat de toute la chaîne est marqué comme rejeté et aucun gestionnaire de réussite n'est appelé au-delà du point où il a échoué.

Tout comme la résolution d'une promesse fournit une valeur, le rejet d'une promesse en fournit une, généralement appelée la *raison* du rejet. Lorsqu'une exception dans une fonction de gestionnaire provoque le rejet, la valeur de l'exception est utilisée comme raison. De même, lorsqu'un gestionnaire retourne une promesse rejetée, ce rejet se retrouve dans la promesse suivante. Il existe une `Promise.reject` fonction qui crée une nouvelle promesse immédiatement rejetée.

Pour gérer explicitement de tels rejets, les promesses ont une `catch` méthode qui enregistre un gestionnaire à appeler lorsque la promesse est rejetée, de la même manière que les `then` gestionnaires gèrent la résolution normale. Cela ressemble également beaucoup `then` au fait qu’elle renvoie une nouvelle promesse, qui revient à la valeur de la promesse initiale si elle se résout normalement, sinon au résultat du `catch` gestionnaire. Si un `catch` gestionnaire génère une erreur, la nouvelle promesse est également rejetée.

En abrégé, `then` accepte également un gestionnaire de rejet en tant que second argument. Vous pouvez donc installer les deux types de gestionnaires dans un seul appel de méthode.

Une fonction transmise au `Promise` constructeur reçoit un deuxième argument, à côté de la fonction de résolution, qu'il peut utiliser pour rejeter la nouvelle promesse.

Les chaînes de valeurs de promesse créées par les appels à `then` et `catch` peuvent être considérées comme un pipeline par lequel les valeurs ou les échecs asynchrones se déplacent. Etant donné que ces chaînes sont créées par des gestionnaires d’enregistrement, chaque lien est associé à un gestionnaire de succès ou à un gestionnaire de rejet (ou aux deux). Les gestionnaires qui ne correspondent pas au type de résultat (succès ou échec) sont ignorés. Mais ceux qui correspondent sont appelés et leur résultat détermine quel type de valeur vient ensuite: le succès lorsqu'il

renvoie une valeur non promise, le rejet lorsqu'il lève une exception et le résultat d'une promesse lorsqu'il renvoie l'une de ces valeurs.

```
nouvelle  promesse (( _ , rejeter ) =>  rejeter ( nouvelle  erreu
  . then ( valeur  =>  console . log ( "Gestionnaire 1" ))
  . catch ( raison  => {
     console . log ( "Échec capturé"  +  raison );
     renvoie  "rien" ;
  })
  . then ( valeur  =>  console . log ( "Gestionnaire 2" , valeur
// → échec capturé Erreur: échec
// → gestionnaire 2 rien
```

Tout comme une exception non capturée est gérée par l'environnement, les environnements JavaScript peuvent détecter le rejet d'une promesse et le signaler comme une erreur.

Les réseaux sont difficiles

Parfois, il n’ya pas assez de lumière pour que les systèmes de miroirs des corbeaux transmettent un signal ou quelque chose bloque le chemin du signal. Il est possible qu'un signal soit envoyé mais jamais reçu.

Dans l’état actuel des choses, le rappel rappelé `send`ne sera jamais appelé, ce qui entraînera probablement l’arrêt du programme sans même s’apercevoir qu’il ya un problème. Ce serait bien si, après une période donnée sans réponse, une demande *expirait* et signalait un échec.

Les échecs de transmission sont souvent des accidents aléatoires, comme le fait de faire interférer le signal lumineux avec les phares d’une voiture, et le simple fait de réessayer la demande risque de provoquer son succès. Donc, pendant que nous y sommes, faisons en sorte que notre fonction de requête répète automatiquement l'envoi de la requête plusieurs fois avant qu'elle n'abandonne.

Et, puisque nous avons établi que les promesses sont une bonne chose, nous ferons également en sorte que notre fonction de requête renvoie une promesse. En ce qui concerne ce qu'ils peuvent exprimer, les rappels et les promesses sont équivalents. Les

fonctions basées sur le rappel peuvent être encapsulées pour exposer une interface basée sur une promesse, et inversement.

Même lorsqu'une demande et sa réponse sont envoyées avec succès, la réponse peut indiquer un échec, par exemple, si la demande tente d'utiliser un type de demande qui n'a pas été défini ou si le gestionnaire lève une erreur. Pour soutenir cela `send`et `defineRequestType`suivre la convention mentionnée précédemment, le premier argument passé aux callbacks est le motif de l'échec, le cas échéant, et le second, le résultat réel.

Ceux-ci peuvent être traduits pour promettre la résolution et le rejet de notre wrapper.

```
Le  délai d'attente de la  classe augmente  Erreur {}

 demande de fonction ( imbrication , cible , type , contenu ) {
   retourne la  nouvelle  promesse (( résolution , rejet )) => {
     let  done  =  false ;
     tentative de fonction  ( n ) {
       imbriqué . send ( cible , type , contenu , ( échec , valeu
         done = true  ;
        si ( échec ) rejeter ( échec );
        else  resol ( valeur );
      });
      setTimeout (() => {
         if ( done ) return ;
         else  if ( n  <  3 ) tentative ( n  +  1 );
         sinon  rejeter ( nouveau  délai d'attente ( "Timeed out'
      }, 250 );
    }
    tentative ( 1 );
  });
}
```

Parce que les promesses ne peuvent être résolues (ou rejetées) qu'une seule fois, cela fonctionnera. La première fois `resolve`ou `reject`est appelée détermine le résultat de la promesse et tout appel ultérieur, tel que le délai qui arrive après la fin de la demande ou une demande revenant après qu'une autre demande est terminée, est ignoré.

Pour créer une boucle asynchrone, pour les tentatives précédentes, nous devons utiliser une fonction récursive: une boucle régulière ne nous permet pas de nous arrêter et d'attendre une action asynchrone. La `attempt` fonction fait une seule tentative pour envoyer une demande. Il définit également un délai d'expiration qui, si aucune réponse n'est renvoyée après 250 millisecondes, démarre la tentative suivante ou, s'il s'agit de la quatrième tentative, rejette la promesse avec une instance de `Timeout` cause.

Réessayer tous les quarts de seconde et abandonner quand aucune réponse n’est arrivée après une seconde est définitivement quelque peu arbitraire. Il est même possible, si la demande est arrivée mais que le gestionnaire prend juste un peu plus longtemps, que les demandes soient livrées plusieurs fois. Nous écrirons à nos gestionnaires avec ce problème à l'esprit: les messages en double devraient être inoffensifs.

En général, nous ne construirons pas un réseau robuste et de classe mondiale aujourd'hui. Mais ce n'est pas grave, les corbeaux n’ont pas encore des attentes très élevées en matière d’informatique.

Pour nous isoler complètement des rappels, nous allons définir une enveloppe `defineRequestType` qui permet à la fonction de gestionnaire de renvoyer une promesse ou une valeur en clair et de la relier au rappel pour nous.

```
function  requestType ( name , handler ) {
   defineRequestType ( name , ( nid , contenu , source ,
                               rappel )) => {
     try {
       Promise . resol ( gestionnaire ( nid , contenu , source ))
        . then ( réponse  =>  rappel ( null , réponse ),
                 échec  =>  rappel ( échec ));
    } catch ( exception ) {
       callback ( exception );
    }
  });
}
```

`Promise.resolve` est utilisé pour convertir la valeur renvoyée par `handler` en une promesse si ce n'est pas déjà fait.

Notez que l'appel à `handler` doit être encapsulé dans un `try` bloc pour s'assurer que toute exception qu'il soulève directement est donnée au rappel. Cela illustre bien la difficulté de gérer correctement les erreurs avec des rappels bruts: il est facile d'oublier de router correctement les exceptions de ce type, et si vous ne le faites pas, les échecs ne seront pas signalés au bon rappel. Les promesses rendent cela principalement automatique et donc moins sujet aux erreurs.

Collections de promesses

Chaque ordinateur de nid conserve un tableau d'autres nids à une distance de transmission de sa `neighbors` propriété. Pour vérifier lesquels de ceux-ci sont actuellement accessibles, vous pouvez écrire une fonction qui essaie d'envoyer une `"ping"` demande (une demande qui demande simplement une réponse) à chacun d'eux et voir ceux qui reviennent.

Lorsque vous travaillez avec des collections de promesses exécutées simultanément, la `Promise.all` fonction peut être utile. Il renvoie une promesse qui attend la résolution de toutes les promesses du tableau, puis se résout en un tableau des valeurs générées par ces promesses (dans le même ordre que le tableau d'origine). Si une promesse est rejetée, le résultat de `Promise.all` est lui-même rejeté.

```
requestType ( "ping" , () =>  "pong" );

fonction  availableNebornes ( nid ) {
   let  demandes  =  nid . voisins . map ( voisin  => {
     demande de retour  ( nid , voisin , "ping" )
      . alors (() =>  vrai , () =>  faux );
  });
  retour  promesse . tous ( demandes ). then ( result  => {
     return  nid . voisins . filter (( _ , i ) =>  result [ i ]);
  });
}
```

Quand un voisin n'est pas disponible, nous ne voulons pas que toute la promesse combinée échoue car nous ne saurions toujours rien. Ainsi, la fonction mappée sur l'ensemble des voisins pour les transformer en promesses de demandes associe des gestionnaires qui produisent les demandes réussies `true` et celles rejetées `false`.

Dans le gestionnaire de la promesse combinée, `filter`est utilisé pour supprimer les éléments du `neighbors`tableau dont la valeur correspondante est false. Cela rend l'utilisation du fait que `filter`passe l'index de tableau de l'élément courant comme un second argument de la fonction de filtrage (`map`, `some`, et les méthodes de réseau d'ordre supérieur similaires font la même chose).

Inondation du réseau

Le fait que les nids ne puissent parler qu'à leurs voisins réduit considérablement l'utilité de ce réseau.

Pour diffuser des informations sur l'ensemble du réseau, une solution consiste à configurer un type de demande qui est automatiquement transmis aux voisins. À leur tour, ces voisins le transmettent à leurs voisins, jusqu'à ce que tout le réseau ait reçu le message.

```
importer { partout } de  "./crow-tech" ;

partout ( nid  => {
   nid . état . gossip  = [];
});

function  sendGossip ( nid , message , saufFor  =  null ) {
   nid . état . potins . pousser ( message );
  for ( laissez le  voisin  du  nid . voisins ) {
     if ( voisin  ==  saufFor ) continue ;
    demande ( nid , voisin , "potins" , message );
  }
}

requestType ( "gossip" , ( nid , message , source ) => {
   if ( nid . état . gossip . includes ( message )) retour ;
   console . log ( `$ { nid . nom }  potins reçus '$ {
               message } ' à partir de $ { source } ` );
   sendGossip ( nid , message , source)
});
```

Pour éviter de jamais envoyer le même message sur le réseau, chaque nid conserve un tableau de chaînes de potins déjà vu. Pour définir ce tableau, nous utilisons la

`everywhere` fonction (qui exécute le code sur chaque nid) pour ajouter une propriété à l' `state` objet du nid , qui est l'endroit où nous allons conserver l'état de nid-local.

Lorsqu'un nid reçoit un message de potins dupliqué, ce qui risque fort de se produire lorsque tout le monde les renvoie aveuglément, il l'ignore. Mais quand il reçoit un nouveau message, il en parle avec enthousiasme à tous ses voisins, à l'exception de celui qui l'a envoyé.

Cela provoquera la propagation d'un nouveau potin à travers le réseau, comme une tache d'encre dans l'eau. Même lorsque certaines connexions ne fonctionnent pas actuellement, s'il existe un itinéraire alternatif pour un nid donné, les commérages le rejoindront par là.

Ce style de communication réseau est appelé *inondation* - il inonde le réseau d'informations jusqu'à ce que tous les nœuds en disposent.

Nous pouvons appeler `sendGossip` pour voir un message circuler dans le village.

```
sendGossip ( bigOak , "Enfants avec une arme à air comprimé dans
```

Routage des messages

Si un noeud donné veut parler à un seul autre noeud, l'inondation n'est pas une approche très efficace. Surtout lorsque le réseau est grand, cela entraînerait de nombreux transferts de données inutiles.

Une autre approche consiste à configurer les messages de noeud en noeud jusqu'à ce qu'ils atteignent leur destination. La difficulté avec cela est que cela nécessite des connaissances sur la disposition du réseau. Pour envoyer une demande en direction d'un nid éloigné, il est nécessaire de savoir quel nid voisin le rapproche de sa destination. L'envoyer dans la mauvaise direction ne fera pas beaucoup de bien.

Comme chaque nid ne connaît que ses voisins directs, il ne dispose pas des informations nécessaires pour calculer un itinéraire. Nous devons en quelque sorte diffuser les informations sur ces connexions à tous les nids, de préférence de manière à ce qu'elles puissent changer au fil du temps, lorsque les nids sont abandonnés ou que de nouveaux nids sont construits.

Nous pouvons utiliser l'inondation à nouveau, mais au lieu de vérifier si un message donné a déjà été reçu, nous vérifions maintenant si le nouvel ensemble de voisins pour un nid donné correspond à l'ensemble actuel que nous en avons.

```
requestType ( "connexions" , ( imbriqué , { nom , voisins },
                               source ) => {
   let  connexions  =  imbriqué . état . connexions ;
   si ( JSON . stringify ( connexions . get ( nom )) ==
      JSON . stringify ( voisins ) ) retourne ;
   connexions . set (nom , voisins );
  broadcastConnections ( nid , nom , source );
});

function  broadcastConnections ( nid , nom , saufFor  =  null ) {
   pour ( laissez  voisin  de  nid . voisins ) {
     if ( voisin  ==  saufFor ) continue ;
    request ( nid , voisin , "connexions" , {
       nom ,
       voisins : nid . état . connexions . get( nom )
    });
  }
}

partout ( nid  => {
   nid . état . connexions  =  nouvelle  carte ;
   nid . état . connexions . set ( nid . nom , nid . voisins );
   broadcastConnections ( nid , nid . nom );
});
```

La comparaison utilise `JSON.stringify` parce que `==`, sur des objets ou des tableaux, ne retournera vrai que lorsque les deux auront exactement la même valeur, ce qui n'est pas ce dont nous avons besoin ici. La comparaison des chaînes JSON est un moyen simple mais efficace de comparer leur contenu.

Les nœuds commencent immédiatement à diffuser leurs connexions, qui devraient, à moins que certains nids soient complètement inaccessibles, donner rapidement à chaque nid une carte du graphe de réseau actuel.

Une chose que vous pouvez faire avec les graphiques est de trouver des itinéraires, comme nous l'avons vu au chapitre 7 . Si nous avons un itinéraire vers la destination

d'un message, nous savons dans quelle direction l'envoyer.

Cette `findRoute` fonction, qui ressemble beaucoup au `findRoute` du chapitre 7 , cherche un moyen d'atteindre un noeud donné dans le réseau. Mais au lieu de renvoyer l'ensemble de l'itinéraire, il ne fait que revenir à l'étape suivante. Ce prochain nid sera lui-même, en utilisant ses informations actuelles sur le réseau, décider où *il* envoie le message.

```
function findRoute ( from , to , connections ) {
   laisse les connexions . get ( at ) | work = [{ at : from , v
  for ( let i = 0 ; i < work . length ; i ++ ) {
     let { at , via } = work [ i ];
    pour ( laisser à côté de | []) {
       if ( next == to ) retour via ;
      if ( ! work . some ( w => w . at == next )) {
         work . push ({ at : next , via : via | | next });
      }
    }
  }
  return null ;
}
```

Nous pouvons maintenant créer une fonction capable d'envoyer des messages à longue distance. Si le message est adressé à un voisin direct, il est remis comme d'habitude. Sinon, il est empaqueté dans un objet et envoyé à un voisin plus proche de la cible, à l'aide du `"route"` type de requête, ce qui entraînera le même comportement de la part de ce voisin.

```
fonction routeRequest ( nid , cible , type , contenu ) {
   if ( nid . voisins . inclus ( cible )) {
     demande de retour ( nid , cible , type , contenu );
  } else {
     let via = findRoute ( nid . nom , cible ,
                           nid . état . connexions );
    if ( ! via ) renvoie une nouvelle erreur ( `No route to $ {
    demande de retour ( nid , via , "route" ,
                  { cible , type , contenu });
  }
}
```

```
requestType ( "route" , ( nid , { cible , type , contenu }) => {
   return  routeRequest ( nid , cible , type , contenu );
});
```

Nous pouvons maintenant envoyer un message au nid situé dans la tour de l'église, ce qui signifie que quatre sauts de réseau ont été supprimés.

```
routeRequest ( bigOak , "Church Tower" , "note" ,
               "Incoming Jackdaws!" );
```

Nous avons construit plusieurs couches de fonctionnalités sur un système de communication primitif pour le rendre facile à utiliser. C'est un bon modèle (bien que simplifié) du fonctionnement des réseaux informatiques réels.

Une propriété distinctive des réseaux informatiques est qu'ils ne sont pas fiables: des abstractions construites au-dessus d'eux peuvent aider, mais vous ne pouvez pas résumer une défaillance réseau. La programmation réseau consiste donc généralement à anticiper et à gérer les échecs.

Fonctions asynchrones

Pour stocker des informations importantes, les corbeaux sont réputés les dupliquer d'un nid à l'autre. Ainsi, lorsqu'un faucon détruit un nid, les informations ne sont pas perdues.

Pour récupérer une information donnée qui ne se trouve pas dans son propre bulbe de stockage, un ordinateur de nidification peut consulter au hasard d'autres nids du réseau jusqu'à ce qu'il en trouve un qui en contient.

```
requestType ( "storage" , ( nid , nom ) =>  stockage ( nid , nom

function  findInStorage ( nid , nom ) {
   retourne le  stockage ( nid , nom ). alors ( trouvé  => {
     if ( found  ! =  null ) return  found ;
     sinon  return  findInRemoteStorage ( nid , nom );
  });
}

fonction  network ( nid ) {
   return  Array . from ( nid . état . connexions . keys ());
```

```
}

function findInRemoteStorage ( nid , nom ) {
   let sources = réseau ( nid ). filtre ( n => n ! = nid .
  function next () {
     if ( sources . length == 0 ) {
       return Promise . rejeter ( nouvelle erreur ( "Introuvabl
     } else {
       let source = sources [ Math . sol ( Math . Aléatoire ()
                                          sources . Longueur )];
       sources = sources . filtre ( n => n ! = source );
       return routeRequest ( nid , source , "stockage" , nom )
         . then ( valeur => valeur ! = null ? valeur : next
                next );
     }
  }
  return next ();
}
```

Parce que `connections` c'est un `Map`, `Object.keys` ça ne marche pas. Il a une `keys` *méthode* , mais cela retourne un itérateur plutôt qu'un tableau. Un itérateur (ou une valeur itérable) peut être converti en tableau avec la `Array.from` fonction.

Même avec des promesses, il s'agit d'un code plutôt délicat. Plusieurs actions asynchrones sont enchaînées de manière non évidente. Encore une fois, nous avons besoin d'une fonction récursive (`next`) pour modéliser en boucle les nids.

Et ce que fait réellement le code est complètement linéaire: il attend toujours que l'action précédente soit terminée avant de commencer la suivante. Dans un modèle de programmation synchrone, il serait plus simple d'exprimer.

La bonne nouvelle est que JavaScript vous permet d'écrire du code pseudo-synchrone pour décrire le calcul asynchrone. Une `async` fonction est une fonction qui renvoie implicitement une promesse et qui peut, dans son corps, d' `await` autres promesses d'une manière qui *semble* synchrone.

Nous pouvons réécrire `findInStorage` comme ceci:

```
fonction asynchrone findInStorage ( nid , nom ) {
  let local = attente de stockage ( nid , nom );
```

```
  si ( local  ! =  null ) renvoie  local ;

  laisser  sources  =  réseau ( nid ). filtre ( n  =>  n  ! =  ni
  while ( sources . length  >  0 ) {
     let  source  =  sources [ Math . sol ( Math . Aléatoire () *
                                    sources . Longueur )];
    sources  =  sources . ! = source );
    essayerfiltre ( n  =>  n  {
       laisser  trouvé  = en  attente  routeRequest ( nid , sourc
                                     nom );
      si ( trouvé  ! =  null ) return  found ;
    } attraper ( _ ) {}
  }
  jeter  nouvelle  erreur ( "Introuvable" );
}
```

Une `async` fonction est marquée par le mot `async` avant le `function` mot clé. Des méthodes peuvent également être faites `async` en écrivant `async` avant leur nom. Quand une telle fonction ou méthode est appelée, elle retourne une promesse. Dès que le corps retourne quelque chose, cette promesse est résolue. S'il y a une exception, la promesse est rejetée.

```
findInStorage ( bigOak , "events on 2017-12-21" )
  . puis ( console . log );
```

À l'intérieur d'un `async` fonction, le mot `await` peut être placé devant une expression pour attendre une promesse à résoudre et ensuite seulement pour continuer l'exécution de la fonction.

Une telle fonction ne fonctionne plus, comme une fonction JavaScript classique, du début à la fin. Au lieu de cela, il peut être *gelé* à tout moment qui a un `await`, et peut être repris plus tard.

Pour le code asynchrone non trivial, cette notation est généralement plus pratique que d'utiliser directement des promesses. Même si vous devez faire quelque chose qui ne correspond pas au modèle synchrone, comme effectuer plusieurs actions en même temps, il est facile de les combiner. `await` cette utilisation avec l'utilisation directe de promesses.

Générateurs

Cette capacité des fonctions à être interrompues puis à reprendre n'est pas exclusive aux `async` fonctions. JavaScript a également une fonctionnalité appelée fonctions *génératrices* . Ce sont similaires, mais sans les promesses.

Lorsque vous définissez une fonction avec `function*` (en plaçant un astérisque après le mot `function`), elle devient un générateur. Lorsque vous appelez un générateur, il renvoie un itérateur, ce que nous avons déjà vu au chapitre 6. .

```
fonction *  pouvoirs ( n ) {
   pour ( laisser  courant  =  n ;; courant  * =  n ) {
     rendement  courant ;
  }
}

pour ( laisser la  puissance  des  pouvoirs ( 3 )) {
   si ( puissance  >  50 ) briser ;
  console . log ( puissance );
}
// → 3
// → 9
// → 27
```

Initialement, lorsque vous appelez `powers`, la fonction est gelée au début. Chaque fois que vous appelez `next` l'itérateur, la fonction s'exécute jusqu'à atteindre une `yield` expression, ce qui la met en pause et fait en sorte que la valeur renvoyée devienne la valeur suivante produite par l'itérateur. Lorsque la fonction revient (celle de l'exemple ne le fait jamais), l'itérateur est terminé.

Écrire des itérateurs est souvent beaucoup plus facile lorsque vous utilisez les fonctions du générateur. L'itérateur de la `Group` classe (tiré de l'exercice du chapitre 6) peut être écrit avec ce générateur:

```
Groupe . prototype [ Symbole . itérateur ] =  function * () {
   for ( let  i  =  0 ; i  <  this . members . length ; i ++ ) {
     donne  ceci . membres [ i ];
  }
};
```

Il n'est plus nécessaire de créer un objet pour conserver l'état d'itération - les générateurs sauvegardent automatiquement leur état local à chaque fois qu'ils cèdent.

De telles `yield`expressions ne peuvent apparaître que directement dans la fonction génératrice elle-même et non dans une fonction interne que vous définissez à l'intérieur. L'état qu'un générateur enregistre, lorsqu'il cède, n'est que son environnement *local* et la position dans laquelle il a cédé.

Une `async` fonction est un type spécial de générateur. Elle produit une promesse lorsqu'elle est appelée, qui est résolue lors de son retour (terminé) et rejetée lorsqu'elle génère une exception. Chaque fois qu'une promesse est rendue (attend), le résultat de cette promesse (valeur ou exception levée) est le résultat de l' `await`expression.

La boucle de l'événement

Les programmes asynchrones sont exécutés pièce par pièce. Chaque pièce peut commencer certaines actions et le code de planification à exécuter lorsque l'action se termine ou échoue. Entre ces éléments, le programme reste inactif, dans l'attente de la prochaine action.

Les callbacks ne sont donc pas appelés directement par le code qui les a programmés. Si j'appelle `setTimeout`depuis une fonction, cette fonction sera revenue au moment où la fonction de rappel est appelée. Et lorsque le rappel est renvoyé, le contrôle ne revient pas à la fonction qui l'a planifié.

Le comportement asynchrone se produit sur sa propre pile d'appels de fonction vide. C'est l'une des raisons pour lesquelles, sans promesses, la gestion des exceptions via du code asynchrone est difficile. Comme chaque rappel commence par une pile pratiquement vide, vos `catch`gestionnaires ne seront pas sur la pile lorsqu'ils lèveront une exception.

```
try {
   setTimeout (() => {
     lance une  nouvelle  erreur ( "Woosh" ));
  }, 20 );
} catch ( _ ) {
   // Cela n'exécutera pas la
  console . log ( "attrapé!" );
}
```

Peu importe la proximité d'événements, tels que des délais d'attente ou des demandes entrantes, un environnement JavaScript n'exécutera qu'un programme à la fois. Vous pouvez penser que cela exécute une grande boucle *autour de* votre programme, appelée la *boucle d'événement* . Quand il n'y a rien à faire, cette boucle est arrêtée. Mais lorsque les événements arrivent, ils sont ajoutés à une file d'attente et leur code est exécuté l'un après l'autre. Etant donné qu'il n'y a pas deux choses qui fonctionnent en même temps, un code à exécution lente peut retarder le traitement d'autres événements.

Cet exemple définit un délai d'expiration, mais reste alloué après le délai prévu, ce qui entraîne un retard.

```
laisser  commencer  =  Date . maintenant ();
setTimeout (() => {
   console . log ( "Le délai d'attente a été dépassé" , Date . ma
}, 20 );
while ( Date . maintenant () <  start  +  50 ) {}
 console . log ( "Temps perdu jusqu'à" , date . maintenant () -
// → Temps perdu jusqu'à 50
// → Délai dépassé à 55 ans
```

Les promesses sont toujours résolues ou rejetées en tant que nouvel événement. Même si une promesse est déjà résolue, l'attente entraînera l'exécution de votre rappel une fois le script en cours terminé, plutôt que tout de suite.

```
Promesse . résoudre ( "Terminé" ). puis ( console . log );
console . log ( "moi d'abord!" );
// → moi d'abord!
// → Terminé
```

Dans les chapitres suivants, nous verrons divers autres types d'événements qui s'exécutent sur la boucle d'événements.

Bugs Asynchrones

Lorsque votre programme s'exécute de manière synchrone, d'un seul coup, aucun changement d'état ne se produit, à l'exception de ceux que le programme lui-même effectue. Pour les programmes asynchrones, c'est différent: ils peuvent avoir des *lacunes* dans leur exécution pendant lesquelles d'autres codes peuvent être exécutés.

Regardons un exemple. L'un des passe-temps de nos corbeaux est de compter le nombre de poussins qui naissent chaque année dans le village. Les nids stockent ce nombre dans leurs ampoules de stockage. Le code suivant tente d'énumérer les comptes de tous les nids pour une année donnée:

```
function  anyStorage ( nid , source , nom ) {
   if ( source  ==  nid . nom ) retourne le  stockage ( nid , nom
  else  return  routeRequest ( nid , source , "stockage" , nom );
}

 fonction  async poussins ( nid , année ) {
   let  list  =  "" ;
  attendre la  promesse . tous ( réseau ( nid .) carte ( async  r
     liste  + =  `$ { nom } : $ {
      attendre  anyStorage ( nid , nom , ` poussins dans $ { anné
     } \ n` ;
  }));
   liste de retour ;
}
```

La `async name =>` partie montre que les fonctions de flèche peuvent également être créées `async` en plaçant le mot `async` devant elles.

Le code ne semble pas immédiatement suspect: il mappe la `async` fonction de flèche sur l'ensemble des nids, créant ainsi un tableau de promesses, puis utilise `Promise.all` pour attendre tous ces éléments avant de renvoyer la liste qu'ils ont construite.

Mais c'est sérieusement cassé. Il ne renvoie toujours qu'une seule ligne de sortie, répertoriant le nid le plus lent à répondre.

```
poussins ( bigOak , 2017 ). puis ( console . log );
```

Pouvez-vous comprendre pourquoi?

Le problème réside dans l' `+=` opérateur, qui prend la valeur *actuelle* de `list` au moment où l'instruction commence à s'exécuter, puis, une fois l'opération `await` terminée, définit la `list` liaison comme étant la valeur ajoutée à la chaîne ajoutée.

Mais entre le moment où l'instruction commence à s'exécuter et l'heure à laquelle elle se termine, il existe un intervalle asynchrone. L' `map` expression s'exécute avant que quoi que ce soit ait été ajouté à la liste, de sorte que chacun des `+=` opérateurs commence par une chaîne vide et se termine, une fois l'extraction de stockage terminée, en définissant `list` une liste à une seule ligne, résultat de l'ajout de sa ligne à la chaîne vide. .

Cela aurait pu être facilement évité en renvoyant les lignes des promesses cartographiées et en appelant `join` le résultat de `Promise.all`, au lieu de construire la liste en modifiant une liaison. Comme d'habitude, le calcul de nouvelles valeurs est moins sujet aux erreurs que la modification de valeurs existantes.

```
fonction  asynchrone poussins ( nid , année ) {
  let  lignes  =  réseau ( nid ). carte ( async  nom  => {
    retour  nom  +  ":"  +
     attendre  anyStorage ( nid , nom , `poussins dans $ { année
 });
  return ( wait  Promise . all ( lignes )). rejoindre ( "\ n" );
}
```

Les erreurs de ce type sont faciles à commettre, en particulier lors de l'utilisation `await`, et vous devez savoir où se trouvent les lacunes de votre code. L’asynchronicité *explicite* de JavaScript (qu’il s’agisse de rappels, de promesses ou `await`) est qu’il est relativement facile de détecter ces lacunes.

Résumé

La programmation asynchrone permet d'exprimer l'attente d'actions de longue durée sans geler le programme pendant ces actions. Les environnements JavaScript implémentent généralement ce style de programmation à l'aide de callbacks, fonctions appelées une fois les actions terminées. Une boucle d’événement planifie l’appel de tels callbacks, le cas échéant, l’un après l’autre, afin que leur exécution ne se chevauche pas.

La programmation asynchrone est facilitée par des promesses, des objets représentant des actions pouvant être exécutées ultérieurement et des `async` fonctions vous permettant d'écrire un programme asynchrone comme s'il était synchrone.

Des exercices

Suivi du scalpel

Les villageois possèdent un vieux scalpel qu'ils utilisent parfois pour des missions spéciales, par exemple pour percer des portes moustiquaires ou des emballages. Pour pouvoir le localiser rapidement, chaque fois que le scalpel est déplacé vers un autre nid, une entrée est ajoutée au stockage du nid qui le contient et du nid qui l'a pris, sous le nom `"scalpel"`, avec son nouvel emplacement comme valeur.

Cela signifie que pour trouver le scalpel, il suffit de suivre le fil d'Ariane des entrées de stockage, jusqu'à ce que vous trouviez un nid qui pointe vers le nid lui-même.

Ecrivez une `async` fonction `locateScalpel` qui fait cela, en partant du nid sur lequel elle tourne. Vous pouvez utiliser la `anyStorage` fonction définie précédemment pour accéder au stockage dans des nids arbitraires. Le scalpel circule depuis assez longtemps pour que vous puissiez supposer que chaque nid a une `"scalpel"` entrée dans son stockage de données.

Ensuite, écrivez à nouveau la même fonction sans utiliser `async` et `await`.

Les échecs de demande apparaissent-ils correctement comme des rejets de la promesse retournée dans les deux versions? Comment?

```
 fonction  async localisationScalpel ( nid ) {
   // Votre code ici.
}

function  localScalpel2 ( nest ) {
   // Votre code ici.
}

localisez Calpel ( bigOak ). puis ( console . log );
// → Boucherie
```

» Display hints...

Construire Promise.all

Compte tenu d'un tableau de promesses, `Promise.all` renvoie une promesse qui attend que toutes les promesses du tableau se terminent. Il réussit ensuite, générant un

tableau de valeurs de résultat. Si une promesse du tableau échoue, la promesse renvoyée par `all` échoue également, avec le motif d'échec de la promesse défaillante.

Implémentez quelque chose comme ceci vous-même en tant que fonction régulière appelée `Promise_all`.

N'oubliez pas qu'après qu'une promesse a réussi ou échoué, elle ne peut plus réussir ni échouer, et les appels ultérieurs aux fonctions qui la résolvent sont ignorés. Cela peut simplifier la façon dont vous gérez l'échec de votre promesse.

```
function  Promise_all ( promises ) {
   retourne une  nouvelle  promesse (( resolve , rejette )) => {
     // Votre code ici.
  });
}

// Code de test.
Promise_all ([]). then ( array  => {
   console . log ( "Cela devrait être []:" , array );
});
function  soon ( val ) {
   retourne une  nouvelle  promesse ( resolution  => {
     setTimeout (() =>  resol ( val ), Math . random () *  500 )
  });
}
Promise_all ([ bientôt ( 1 ), bientôt ( 2 ), bientôt ( 3 )]). the
   console . log ( "Cela devrait être [1, 2, 3]:" , array );
});
Promise_all ([ bientôt ( 1 ), Promise . Rejette ( "X" ), bientôt
  . then ( array  => {
     console . log ( "Nous ne devrions pas arriver ici" );
  })
  . catch ( error  => {
     if ( error  ! =  "X" ) {
       console . log ( "Échec inattendu:" , erreur );
    }
  });
```

» Display hints...