NOHAR

Geste de savoir

Modéliser la solution plutôt que le problème

5 octobre 2020

Table des matières

Cette semaine, je me suis amusé à réaliser quelques exercices sur exercism.io pour progresser en Go.

Un exercice particulier m'aura donné du fil à retordre. Non pas qu'il fasse intervenir des notions compliquées en Go ou même en programmation tout court, mais j'ai mis pas mal de temps et d'itérations avant de trouver une solution:

— correcte du point de vue algorithmique,
— dont les performances restent stables même quand l'utilisateur est gourmand,
— *threadsafe*,
— qui ne mette pas le *Garbage Collector* à mal,

Pour le coup, je trouve que le cheminement que j'ai suivi avant d'arriver à une solution satisfaisante à mes yeux est particulièrement instructif, parce qu'il m'a permis de toucher à **des tas** de choses que l'on n'apprend dans aucun cours de programmation, et qui sont pourtant vitales lorsque l'on développe un programme qui doit fonctionner dans *La Vraie Vie*™. Cela a été pour moi l'occasion de me rappeler une règle d'or dont on entend finalement assez peu parler et que l'on a tendance à vite oublier quand on pense "Orienté Objet": quand on programme, **c'est la solution qu'il faut modéliser, pas le problème**.

Je me propose de reproduire ce cheminement ici, à la lumière de cette fameuse règle.

1. L'énoncé : un problème de programmation réactive

Le but de cet exercice est d'implémenter un système "réactif", non pas dans le sens du reactive manifesto , mais dans le sens de la programmation réactive . Il s'agit d'un paradigme dans lequel on se concentre sur la façon dont interagissent des valeurs qui dépendent les unes des autres.

Pour faire simple, pensez à un tableur comme Excel.

Dans un tableur, on peut définir des cellules dont la valeur est le résultat d'une formule portant sur d'autres cellules. Pour ne pas nous mélanger les pinceaux, commençons par nommer les choses:

- Une cellule dont la valeur est renseignée à la main est appelée une `InputCell` ;
- Une cellule dont la valeur est calculée à partir d'autres cellules est appelée une `ComputeCell`.

Par exemple, on peut définir :

- Une cellule `A`, de type `InputCell`,
- Une cellule `B`, de type `InputCell`,
- Une cellule `C`, de type `ComputeCell`, dont la formule est `A + B`,
- Une cellule `D`, de type `ComputeCell`, dont la formule est `A * C`.

Si l'on pose `A = 2` et `B = 3`, on obtient automatiquement:

- `C = A + B = 2 + 3 = 5`,
- `D = A * C = 2 * 5 = 10`.

Si l'on modifie maintenant `A` pour qu'elle porte la valeur `3`, alors instantanément, `C` et `D` sont mises à jour avec:

- `C = 6`,
- `D = 18`.

Le but de l'exercice, donc, est d'écrire un tel système en Go, dont l'interface est imposée.

```
```

Nous allons itérer sur l'implémentation de cette interface, en nous ajoutant des contraintes au fur et à mesure.

2. Une solution naïve : modéliser le problème

Je ne sais pas vous, mais en ce qui me concerne, lorsque l'on me soumet un énoncé pareil en précisant «*attention, c'est un exercice difficile*», la première chose que je fais ; c'est réprimer deux réflexes:

— Se dire «mais non, ça n'a pas l'air bien sorcier!»;
— Réfléchir au code en m'imposant dès le départ **toutes les contraintes possibles**.

En effet, si cet exercice a été *flaggé* comme «difficile», a fortiori sur un site réputé et très visité comme *exercism*, c'est probablement parce que l'énoncé cache quelque chose qui n'est pas immédiatement visible. Ainsi, il ne sert à rien de chercher à produire une solution parfaite du premier coup: on a le temps, et rien à prouver à personne.

Pour cette raison, la première approche que j'ai suivie a été d'implémenter une version naïve, en modélisant *le problème*: on veut des cellules qui se mettent à jour les unes les autres. Ok, commençons donc par modéliser une cellule qui va implémenter les trois interfaces à la fois: `Cell`, `InputCell`, `ComputeCell`.

```

```

2.1. Étape 1 : Le système de callbacks

Bon, ça commence bien. Il y a un premier truc qui me chiffonne sur l'interface: lorsque l'on ajoute un callback à une cellule, il faut retourner un objet `Canceler` dont la méthode `Cancel` supprime le callback. Quand on a déjà fait un peu de Go, cette contorsion est un peu surprenante parce que dans la bibliothèque standard (par exemple dans le module `context`) on aurait plutôt tendance à retourner *une fonction* `cancel()` directement.

SPOILER

Soit, commençons donc par une petite pirouette qui nous permet de retourner une fonction toute bête, mais que l'utilisateur pourra appeler en appelant une méthode `Cancel`:

```

```

Maintenant, débarrassons-nous de `AddCallback`:

```

```

Ici, donc, je choisis que "supprimer un callback" consiste juste à remplacer la fonction par `nil` dans une *slice*. Ce n'est sûrement pas la façon la plus élégante de gérer ce cas (si jamais l'utilisateur ajoute et supprime des tonnes de callbacks, la *slice* va devenir un gigantesque

gruyère), mais c'est simple et ça fonctionne, donc ça suffira pour commencer[1]. On verra une solution alternative plus loin.

Une fois ceci fait, il ne nous reste plus qu'à appeler les callbacks au bon endroit, dans `SetValue`:

```
```

Si vous avez l'habitude du C, vous tiquerez peut-être sur le fait que l'on n'a aucune assurance que `c.callbacks` ne soit pas `nil` au moment où on itère dessus. Go s'en fiche. Itérer avec `range` sur une collection nulle est la même chose qu'itérer sur une collection vide.[2]

1. En fait, même si cette solution est moche, elle a le mérite de rendre **prévisible** l'ordre dans lequel les callbacks seront appelés. Par la même occasion, ça me permet de reproduire de façon déterministe un bug que l'on découvrira plus loin. Mais ne nous gâchons pas la surprise, on verra tout ça en temps voulu.

2. C'est à la fois "bien" et "pas bien". C'est "bien" parce que c'est parfaitement logique et que ça permet d'alléger le code, mais c'est "pas bien" parce qu'au lieu d'écrire du code qui vérifie que la slice est bien allouée, on va avoir tendance à écrire un commentaire pour préciser qu'on sait bien que la slice peut être `nil`, mais qu'on s'en fout qu'elle le soit…

2.2. Étape 2 : *Coder le reste du foutu Reactor*

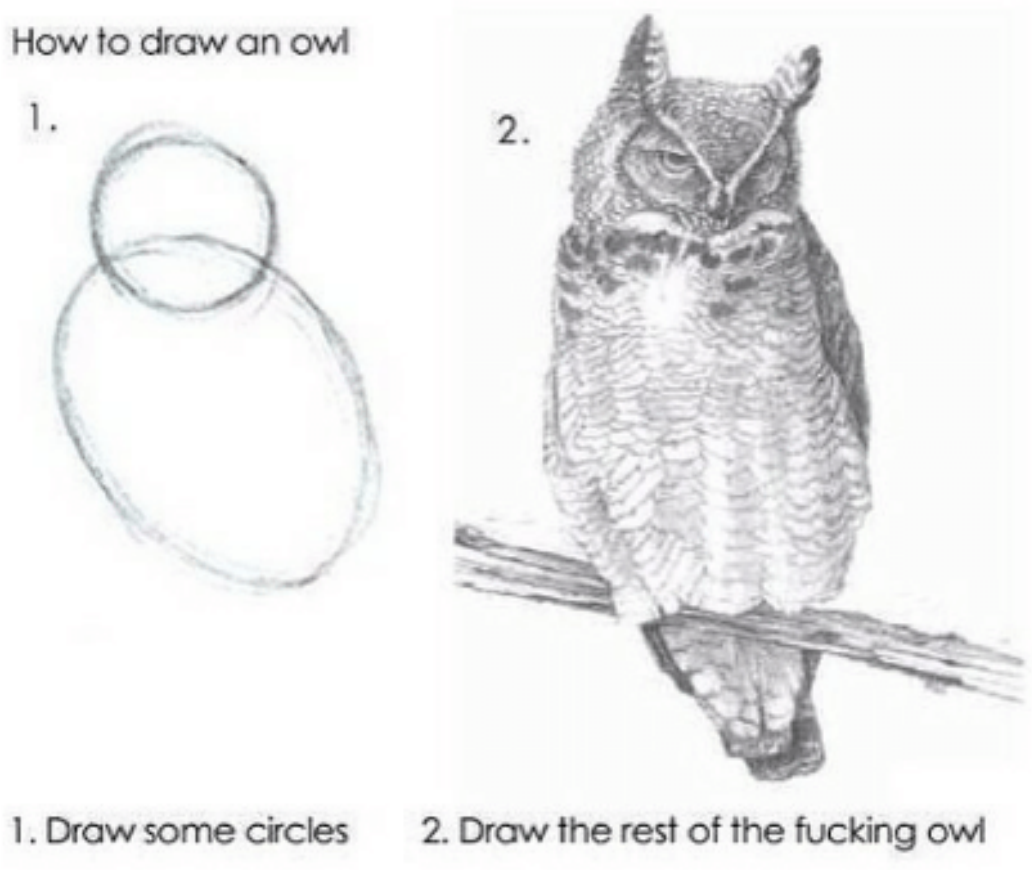

FIGURE 2.1. – Implémenter un système réactif : une allégorie.

Maintenant qu'on a bien modélisé des cellules qui appellent des callbacks quand on les met à jour, on doit pouvoir coder facilement le reste du `Reactor`, non? Essayons!

Alors, le Reactor doit permettre de retourner une `InputCell` initialisée avec la valeur choisie…

```
```

Passons à la suite. `CreateCompute1` doit créer une `ComputeCell` qui se mettra à jour lorsque la cellule passée en argument est modifiée.

```
```

3. C'est maintenant que ça commence!

Facile! Faisons pareil avec `CreateCompute2`.

```
```

Eh bien je ne vois vraiment pas ce que cet exercice avait de difficile. C'est déjà fini! 🍊

Nous n'avons plus qu'à tester notre code avant d'aller prendre un café bien mérité…

3. C'est maintenant que ça commence !

3.1. À première vue, ça marche…

Reproduisons l'exemple que j'ai donné dans l'énoncé pour nous assurer que ça fonctionne.

```
```

Voici le résultat :

```
Valeurs initiales: c = 5, d = 10
Nouvelles valeurs: c = 6, d = 18
```

Super! C'est exactement ce qu'on voulait.

3.2. … mais un bug sauvage apparaît !

Allez, on va se raconter ça sous la forme d'une petite histoire rigolote.

Vous faites partie du département IT d'une entreprise de distribution. L'entreprise n'a pas de stock, chaque fois qu'elle enregistre une commande, elle va acheter le produit chez son fournisseur et le revendre en se payant une marge de 33%.

Vous allez donc confier votre système au Grand Patron™ qui va s'en servir pour surveiller sa trésorerie:

```
Valeurs initiales: c = 5, d = 10
Nouvelles valeurs: c = 6, d = 18
```

Bien! Chaque fois qu'on va enregistrer une commande, tout ce joli monde va se mettre à jour et la cellule `nouveauSolde` va contenir le nouveau solde du compte en banque.

Évidemment, vous ne manquez pas de montrer au Grand Patron™ votre *killer feature*, le système de callback. Celui-ci va s'empresser d'ajouter une alerte par SMS:

```
Valeurs initiales: c = 5, d = 10
Nouvelles valeurs: c = 6, d = 18
```

La nuit suivante, vers 3h du matin, le Super Commercial™ qui se trouve à Perpète, sur un autre continent, conclut le *deal du siècle* et enregistre 100 commandes d'un coup.

Testons ce scénario:

```
Valeurs initiales: c = 5, d = 10
Nouvelles valeurs: c = 6, d = 18
```

Voici le résultat:

```
Solde initial : 0€
On enregistre 100 commandes...
[ALERTE] Solde négatif : -10000€
Solde final : 5000€
```

Ainsi, le Grand Patron™ est réveillé par un SMS au beau milieu de la nuit, lui indiquant qu'il est **DANS LE ROUGE**!! Il se lève en sursaut, gesticule, cherche sa robe de chambre, panique, entreprend de détruire son mobilier, se ravise, commence à fulminer, reprend son téléphone et appelle sa banque. Après avoir traité la personne au bout du fil de tous les noms, il finit par faire réveiller un superviseur qui lui assure, vingt minutes plus tard, que tout va bien. Tout ce qu'il voit, c'est un compte créditeur de 5000€.

Le lendemain matin, vous êtes viré. Ça vous apprendra à faire des blagues.

Heureusement, ce n'est qu'une histoire: il n'y a ni Grand Patron™, ni Super Commercial™. Tout ça n'est qu'un exercice, et vous venez simplement d'avoir une illustration de sa **difficulté**. 🍊

3.3. Caractériser le problème

Ce qui rend ce bug particulièrement vicieux, c'est que dans notre cas, il suffit d'inverser l'ordre de déclaration des deux premières `ComputeCell` pour le faire disparaître :

3. Cela dit, le simple fait d'imaginer un de mes anciens patrons, les cheveux en pétard dans son pyjama à carreaux, se tenant droit comme un piquet au milieu du salon qu'il vient de dévaster, interrompre soudainement son chapelet d'injures et prononcer un simple "oh…" dans le micro du téléphone quand son banquier lui apprend que tout va bien, est une expérience… cathartique! Rien que pour ça, ça valait le coup de coder une solution naïve.

```
Solde initial : 0€
On enregistre 100 commandes...
[ALERTE] Solde négatif : -10000€
Solde final : 5000€
```

Résultat :

```
Solde initial : 0€
On enregistre 100 commandes...
Solde final : 5000€
```

En fait, il est assez simple de remarquer ce qui ne va pas. Si l'on ajoute un callback sur chacune de nos cellules pour vérifier ce qui se passe, on observe ce qui suit:

```
Solde initial : 0€
On enregistre 100 commandes...
MàJ des dépenses : 10000
MàJ du résultat : -10000
MàJ du nouveau solde : -10000
[ALERTE] Solde négatif : -10000€
MàJ des recettes : 15000
MàJ du résultat : 5000
MàJ du nouveau solde : 5000
Solde final : 5000€
```

Ce qui se passe avec ce code, c'est que lors de la mise à jour, **les cellules sont susceptibles de traverser des états intermédiaires** instables, et donc de produire ce genre de gags qui réveillent les gens pour rien à 3h du mat'[3], avant d'être stabilisées sur le résultat final.

Hormis les problèmes de performances qui risquent de se poser dans les cas pathologiques (on risque de réaliser beaucoup plus de calculs que nécessaire…), on a un véritable problème fonctionnel: quand on met à jour une cellule, on ne peut pas savoir si l'état dans lequel on la place est stable ou non. À cela, on peut ajouter un autre problème: en l'état, si une cellule est actualisée *sans changer de valeur*, ses callbacks seront quand même appelés. On risque donc de spammer l'utilisateur avec des mises à jour qui n'en sont pas.

J'ai une bonne et une mauvaise nouvelle à vous annoncer:

— La bonne nouvelle, c'est que l'un de ces deux bugs se solutionne avec un bête `if`/`else`, et que pour le moment il nous arrange parce qu'il nous donne une façon simple d'observer notre algorithme en action;
— La mauvaise nouvelle, c'est que pour l'autre bug, on va devoir faire un peu de théorie des graphes.

4. Trouver le bon algorithme

4.1. Avance rapide !

Vu que tout ceci n'est qu'un billet, je vais passer très rapidement sur une solution intermédiaire.

Vous aurez certainement compris que le principal défaut de notre solution naïve, c'est qu'elle utilise les callbacks de l'interface `ComputeCell` pour propager les nouvelles valeurs des `InputCell`: on ne devrait mettre à jour une cellule qu'une fois que l'on est certain que son nouvel état est définitif. Autrement dit: on ne devrait mettre à jour chaque `ComputeCell` qu'*une seule fois grand max*.

Bien, maintenant, remarquons une propriété sympa de l'interface qui nous est imposée: puisque, pour créer une `ComputeCell`, nous avons besoin de lui passer ses dépendances, cela veut dire qu'une cellule donnée peut dépendre **que** des cellules créées avant elle, et donc qu'**il suffit de mettre à jour les cellules dans l'ordre dans lequel elles ont été créées** pour garantir qu'elles atteindront un état stable du premier coup.

Sachant cela, notre solution intermédiaire consiste à:

— stocker dans une slice (appartenant au `Reactor`) des fonctions qui déclenchent la mise à jour de chaque `ComputeCell`, au fur et à mesure que celles-ci sont créées;
— lorsqu'une `InputCell` est mise à jour, déclencher la mise à jour de **toutes** les `ComputeCell`, dans le bon ordre, en itérant sur cette slice.

Ce n'est pas très compliqué à coder:

```
Solde initial : 0€
On enregistre 100 commandes...
MàJ des dépenses : 10000
MàJ du résultat : -10000
MàJ du nouveau solde : -10000
[ALERTE] Solde négatif : -10000€
MàJ des recettes : 15000
MàJ du résultat : 5000
MàJ du nouveau solde : 5000
Solde final : 5000€
```

On remarquera au passage que l'on n'a plus besoin de faire des *casts* sauvages dans les méthodes `CreateCompute*`. C'est un signe! Mais surtout, on constate que tout se met à jour en une seule passe:

```
Solde initial : 0€
On enregistre 100 commandes...
MàJ des dépenses : 10000
MàJ des recettes : 15000
MàJ du résultat : 5000
```

```
MàJ du nouveau solde : 5000
Solde final : 5000€
```

Parfait, on ne réveillera plus les gens pour rien.

Cela dit, **cette solution ne devrait normalement pas vous satisfaire**. En effet, si le système possède un million de `ComputeCell`, mais que l'on met à jour une cellule dont dépend **une seule** `ComputeCell`, on se retrouve à exécuter neuf cent quatrevingt-dix-neuf mille neuf cent quatrevingt-dix-neuf (999 999) mises à jour inutiles.

On peut aisément le vérifier sur notre exemple, en mettant à jour le prix de vente et en constatant que cela déclenche une mise à jour des dépenses qui n'ont pourtant rien à faire de cette variable (et n'ont donc pas besoin d'être mises à jour).

```
Solde initial : 0€
On enregistre 100 Commandes...
MàJ des dépenses : 10000
MàJ des recettes : 15000
MàJ du résultat : 5000
MàJ du nouveau solde : 5000
On diminue diminue le prix de vente à 120€...
MàJ des dépenses : 10000
MàJ des recettes : 12000
MàJ du résultat : 2000
MàJ du nouveau solde : 2000
Solde final : 2000€
```

Cherchons un algorithme un peu plus intelligent que ça.

4.2. Fermeture transitive d'un noeud dans un graphe

Donc, soyons clairs, nous sommes en train de réfléchir sur une relation de dépendance. La meilleure façon de modéliser ça, c'est de dessiner un graphe.

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                (resultat)         [solde]
                         \          /
                                          (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

Ce que l'on appelle la **fermeture transitive** d'un noeud dans un graphe comme celui-là, c'est l'ensemble de tous les noeuds que l'on peut atteindre en suivant un chemin valide dans le graphe. Par exemple:

— la fermeture transitive de `coutAchat` est l'ensemble `{depenses, resultat, nouveauSolde}`,
— la fermeture transitive de `solde` est l'ensemble `{nouveauSolde}`.

Dans notre cas, la fermeture transitive d'une `InputCell`, c'est **l'ensemble des `ComputeCell` qui ont besoin d'être mises à jour** chaque fois qu'elle est modifiée.

Tant que nous y sommes, définissons **l'ensemble d'entrée** d'une `ComputeCell` comme l'ensemble des `InputCell` dont la modification entraîne sa mise à jour. Les deux affirmations suivantes sont alors équivalentes:

— `c` appartient à la **fermeture transitive** de `i`,
— `i` appartient à **l'ensemble d'entrée** de `c`.

Notre nouvelle solution peut donc se décrire ainsi:

— À tout instant, on garde sous la main:
  — la fermeture transitive de chaque `InputCell`,
  — l'ensemble d'entrée de chaque `ComputeCell`.
— Lorsque l'on crée une nouvelle `ComputeCell`, on récupère son ensemble d'entrée (en fusionnant ceux de ses parents), puis on ajoute la nouvelle cellule à la fermeture transitive de chaque `InputCell` de son ensemble d'entrée.
— Lorsque l'on modifie la valeur d'une `InputCell`, on déclenche la mise à jour, dans leur ordre de création, de chaque `ComputeCell` de sa fermeture transitive.

Allez, on va faire ça.

5. Structurer correctement ses données

Vous vous rappelez tous ces petits détails qui me faisaient tiquer dans l'implémentation naïve? Ce sont autant de signes que les structures de données que nous avons construites sont mal conçues.

Prenons le temps d'y revenir pour rectifier le tir.

5.1. Implémenter une collection de callbacks

Jusqu'ici, nos callbacks étaient référencés dans une slice, ce qui avait le mérite de fixer l'ordre dans lequel ceux-ci sont exécutés. Cependant, ici, nous avons plutôt besoin d'un *ensemble* (`set`) de callbacks:

— On se fiche de l'ordre dans lequel les callbacks sont appelés;
— Ce serait mieux si on n'avait pas à parsermer notre structure d'entrées nulles au fur et à mesure que l'on supprime des callbacks.

La façon la plus idiomatique de réaliser ceci en Go est d'utiliser une `map` (comme un dictionnaire en Python). Réglons donc le problème en implémentant une structure `callbackMap` supportant les méthodes `Add`, `Remove` et `Trigger`.

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                (resultat)         [solde]
                         \          /
                                          (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

5.2. Attention aux références circulaires !

Un autre détail sur lequel j'ai ouvertement tiqué dans notre première implémentation était que notre structure `reactor` était **vide**. En effet, nous avions centré notre modélisation sur les cellules, et partant de là, pratiquement toute l'intelligence se retrouvait implémentée dans la structure `cell`. Si l'on continue sur cette lancée, on va se heurter à un mur, car on s'apprête à maintenir une relation réciproque dans notre modèle:

— Les `InputCell` doivent référencer les `ComputeCell` de leur fermeture transitive;
— Les `ComputeCell` doivent référencer les `InputCell` de leur ensemble d'entrée.

D'après ce que nous avons établi plus haut et sachant que dans notre modélisation, une "*référence*" sur une autre cellule était typiquement *un pointeur* sur cette cellule, nous risquons de créer des **références circulaires** qui rendent n'importe quel *garbage collector* incapable de libérer la mémoire, même lorsque nos cellules ne seront plus accessibles nulle part dans notre programme.

Le meilleur moyen de casser ces références circulaires, c'est de créer une *abstraction* dont vont dépendre `InputCell` et `ComputeCell` pour se référencer mutuellement. Autrement dit, les références ne doivent pas être des pointeurs, mais plutôt des "clés" comme les indices d'un tableau. Un tableau qui serait maintenu, par exemple, dans le `Reactor`…

5.3. Attention à la *thread safety* !

Dans beaucoup de langages, on pourrait se dire que la *thread safety* est un luxe dont on peut se passer, mais Go est un langage concurrent par nature: du moment que l'on expose une interface, on doit s'attendre à ce que quelqu'un veuille se mettre à l'utiliser dans des goroutines indépendantes. Autrement dit, si l'on fait quelque chose qui puisse servir dans la vraie vie, il vaut mieux que ce soit *threadsafe*.

Cette contrainte nous amène au même constat que le problème des références circulaires. En effet, si une `InputCell` est modifiée, alors celle-ci doit verrouiller (`Mutex.Lock()`) l'ensemble

des `ComputeCell` de sa fermeture transitive avant de réaliser la moindre mise à jour. Or, dans notre implémentation naïve, toutes ces cellules sont des objets indépendants les uns des autres: ça voudrait dire que l'on les verrouille dans une boucle.

Seulement voilà: dans ce scénario, si plusieurs `InputCell` distinctes sont mises à jour en même temps, elles se lancent dans une course effrénée (à qui verrouillera sa fermeture transitive la première). Dans le cas de deux mises à jour simultanées, cela va encore bien se passer, mais à partir de trois, c'est susceptible d'engendrer un **deadlock**.[4]

En revanche, si ce ne sont plus aux `InputCell` mais au `Reactor` qu'échoit la responsabilité de faire cette mise à jour, celui-ci peut verrouiller toutes ses ressources en un seul `Lock()`, et donc s'accomoder très facilement d'être utilisé par des dizaines de goroutines indépendantes.

Les données ET l'intelligence doit donc être implémentées dans le `Reactor`, et non dans les cellules.

6. Implémentation finale

6.1. Les cellules

Au vu de tout ce que nous venons de voir, voici ce à quoi va ressembler nos nouvelles cellules:

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                (resultat)         [solde]
                         \          /
                                          (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

Comme précédemment, notre type `cell` implémente les trois interfaces (`Cell`, `InputCell`, `ComputeCell`) à la fois, mais cette fois-ci, nos cellules sont *débiles*. Elles se contentent de référencer le `reactor` et de garder leur `id`. Elles ne sont d'ailleurs même pas mutables: aucune des opérations que nous pouvons exécuter dessus ne modifie leur état interne. C'est pour cette raison que nous allons pouvoir nous permettre de transmettre celles-ci *par valeur* plutôt que *par référence*: remarquez que les méthodes portent sur `cell` et non sur `*cell`.

6.2. Le *reste du foutu Reactor...*

Cette fois, notre `reactor` va être beaucoup plus costaud.

4. Je vous laisse réfléchir au "pourquoi" de cette remarque en guise d'exercice.

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                 (resultat)         [solde]
                          \          /
                                              (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

Les trois premiers champs sont des *slices* de même longueur: le nombre de cellules gérées par le `Reactor`.

Les cellules sont de simples entrées dans une *slice* d'entiers (`cells`), et on se référera à elles via leur `cellID`. La seconde *slice*, `updates`, va contenir les fonctions de mise à jour des `ComputeCells`. Une cellule dont l'entrée est `nil` dans cette slice est une `InputCell`. Remarquez la signature de ces fonctions: elles prennent en entrée *la totalité des cellules* et retournent une valeur.

La troisième slice, `closure`, va contenir:

— La fermeture transitive des `InputCell`,
— L'ensemble d'entrée des `ComputeCell`.

Ensuite, nous allons stocker les ensembles de callbacks des cellules dans un `map`, car il est très probable que seulement *une petite partie* des cellules ait réellement des callbacks associés.

Pour terminer, notre structure embarque un `Mutex` qui nous servira à la rendre *threadsafe*. À ce propos, il est important de savoir dès le départ *quelles* méthodes on va devoir protéger avec ce mutex. En effet, certaines des méthodes que nous nous apprêtons à implémenter seront appelées *de l'intérieur* du reactor, et d'autres *de l'extérieur*: ce sont bien sûr les secondes qu'il est nécessaire de protéger, mais surtout pas les premières (au risque de créer des *deadlocks*). Nous allons donc, dans ce code, utiliser la convention de nommage de Go: nos méthodes privées commenceront par une minuscule, et les méthodes exposées à l'extérieur par une majuscule.

Ainsi, pour accéder à la valeur d'une cellule *depuis l'intérieur* du reactor, il nous suffira d'accéder à `r.cells[id]`, mais depuis l'extérieur, nous allons devoir utiliser une méthode dédiée:

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                 (resultat)         [solde]
                          \          /
                                              (nouveauSolde)

```

```
Légende:
   [InputCell]
   (ComputeCell)
```

De la même manière, pour modifier la valeur d'une cellule, nous allons définir deux méthodes, dont une seule sera publique:

- `setValueAt` modifie la valeur et retourne `true` si la valeur a bien été modifiée.
- `updateAt` déclenche la mise à jour d'une `ComputeCell` et appelle ses callbacks si celle-ci a bien été modifiée.
- `SetValueAt` sera appelée par le biais d'une `InputCell`, pour mettre à jour sa valeur et déclencher les mises à jour.

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                (resultat)        [solde]
                         \          /
                                          (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

Passons maintenant aux callbacks. La seule réelle subtilité, c'est de bien penser à protéger l'ajout **et** la suppression au moyen du mutex.

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                (resultat)        [solde]
                         \          /
                                          (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

6.3. Les cellules

Bien, il ne nous reste "plus qu'à" gérer la création de nos cellules.

Commençons par une petite méthode utilitaire, `newCell`, qui va simplement se charger d'allouer un nouveau *slot* dans chacune des slices du `reactor`, et retourner le `cellID` de la cellule nouvellement créée:

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                (resultat)        [solde]
                        \          /
                                          (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

Pas de difficulté particulière ici. Cette méthode rend même triviale l'implémentation de `CreateInput`.

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                (resultat)        [solde]
                        \          /
                                          (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

C'est maintenant que ça se corse. Lorsque l'on crée une `ComputeCell` nous allons avoir besoin de l'ajouter au graphe de dépendances du `reactor`, c'est-à-dire celui qui est décrit par son champ `closure`. Nous allons devoir:

— Récupérer les ensembles d'entrée de ses parents et les fusionner dans sa `closure`,
— Ajouter le nouveau `cellID` dans la `closure` de toutes les cellules de son ensemble d'entrée.

Petite subtilité supplémentaire: cette méthode sera appelée avec un ou deux parents. Elle doit donc être *variadique*.

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
```

```
                (resultat)        [solde]
                        \          /
                                              (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

Et maintenant, il ne nous reste plus qu'à recoller les morceaux avec les deux méthodes `Create Compute*`

```
  [coutAchat]  [nbCommandes]  [prixVente]
              \      /      \      /
         (depenses)    (recettes)
                              \      /
                (resultat)        [solde]
                        \          /
                                              (nouveauSolde)

Légende:
   [InputCell]
   (ComputeCell)
```

Et voilà le travail!

Testons rapidement le dernier scénario que nous avions créé (Super Commercial™ enregistre 100 commandes, puis on modifie le prix de vente):

```
Solde initial : 0€
On enregistre 100 Commandes...
MàJ des dépenses : 10000
MàJ des recettes : 15000
MàJ du résultat : 5000
MàJ du nouveau solde : 5000
On diminue diminue le prix de vente à 120€...
MàJ des recettes : 12000
MàJ du résultat : 2000
MàJ du nouveau solde : 2000
Solde final : 2000€
```

Tout se comporte maintenant comme prévu. 🍊

Pour conclure, eh bien…

D'abord, c'était vraiment un exercice **difficile**. Il n'en avait pas l'air au premier abord, mais je pense que vous aurez constaté toutes les petites subtilités, tant algorithmiques que d'implémentation, que nous avons eu l'occasion de soulever ici.

Ensuite, remarquez que notre toute dernière implémentation n'est **pas du tout intuitive**. Si je vous avais montré celle-ci dès le départ, vous auriez eu toute légitimité à vous demander si je n'avais pas pété les plombs à écrire un code si compliqué pour résoudre un problème aussi "simple" à modéliser: c'est parce que cette implémentation modélise *la solution*, et non *le problème*.

Autrement dit, la leçon que je pense qu'il faut retenir ici, c'est que **modéliser le problème** n'est que la première étape vers l'implémentation d'une bonne solution, et qu'il ne faut surtout pas s'arrêter là (comme j'ai pu le voir dans de nombreuses solutions publiées sur le site), sous peine de passer à côté de l'exercice. 

Contenu masqué

Contenu masqué n°1 : SPOILER

Ça cache quelque chose…

Retourner au texte.