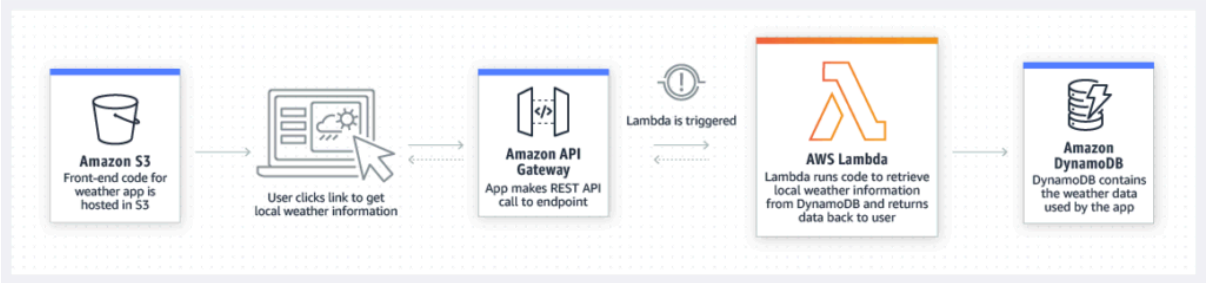
Développement de Contrats Intelligents sur Near en Utilisant Rust - Partie 1



Le 16 février 2023, j'ai animé un atelier à l'Université de Waterloo sur le développement de contrats intelligents sur Near en utilisant Rust. J'ai apprécié de le mettre en place et j'ai pensé qu'il serait utile de présenter le contenu ici également, sous forme d'une série d'articles de blog. Dans ce premier article, je donnerai une analogie pour connecter le développement de la blockchain à un modèle utilisé dans les applications web normales. Présenter l'exemple de contrat intelligent que nous utiliserons tout au long de cette série et discuter de quelques principes généraux du développement de contrats intelligents qui sont uniques à la blockchain par rapport à d'autres domaines de la programmation.

Un modèle mental pour créer une application distribuée (dapp)

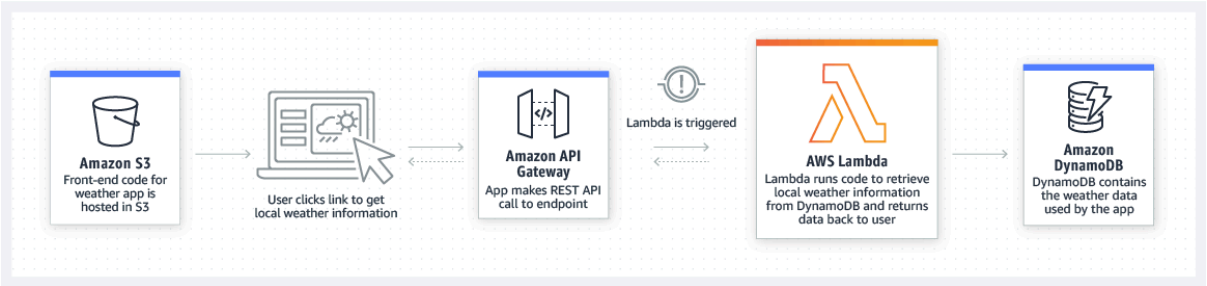
Le but de cette section est de faire une analogie entre le développement sur une blockchain (les applications basées sur la technologie de la blockchain sont souvent appelées "**dapps**" dans l'espace) et une technologie plus courante pour les applications web que vous avez

peut-être déjà rencontrée. Cette analogie peut être utile pour réfléchir à la manière dont les utilisateurs interagissent avec les contrats intelligents.

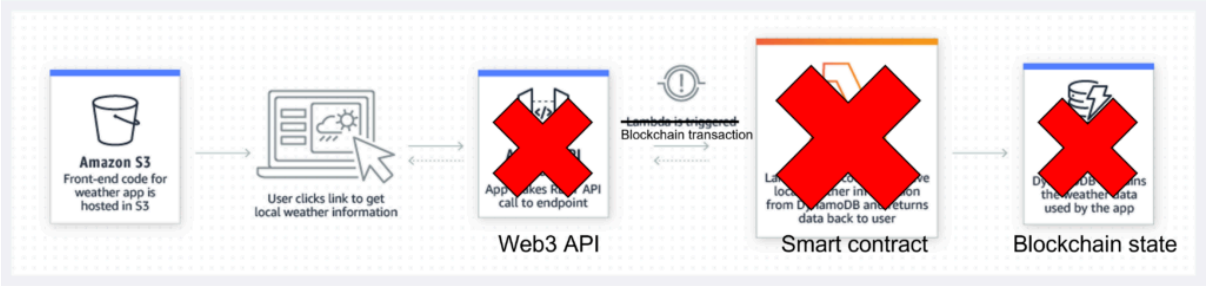
L'idée est que les dapps sont très similaires aux applications web basées sur une "architecture serverless". Le terme "**serverless**" est un peu trompeur, car bien sûr, des serveurs sont toujours impliqués, mais la raison de ce nom est que le matériel sous-jacent (c'est-à-dire le serveur) exécutant le code est abstrait pour le développeur. Cela présente des avantages par rapport à d'autres infrastructures d'informatique dans le cloud en termes de coût et de scalabilité, attendu que vous ne payez que pour les ressources que vous utilisez exactement, contrairement à une VM qui peut rester inactive si le trafic est faible ou devenir non responsive s'il y a trop de trafic. Chaque fois qu'un utilisateur interagit avec l'application web, une nouvelle instance de la "fonction sans serveur" est invoquée sur le backend pour répondre à la demande de l'utilisateur sans que le développeur ait à se soucier précisément du matériel sur lequel cette fonction est exécutée.

Les dapps abstraient le matériel de manière similaire. Un contrat intelligent est déployé sur la blockchain et exécuté sur les nœuds (serveurs) qui forment le réseau pair-à-pair de cette blockchain. Lorsqu'un utilisateur interagit avec la dapp, il appelle la blockchain (une transaction) pour exécuter le contrat intelligent. Chaque transaction crée une nouvelle instance du contrat intelligent (dans le sens où il n'y a pas d'état en mémoire qui persiste entre les transactions), tout comme avec les fonctions sans serveur.

L’image ci-dessous est directement tirée du site Web d'Amazon Web Services (AWS) pour Lambda (leur version d'une offre de calcul sans serveur).



Il est facile de modifier cette image pour voir comment le flux de travail dans une dapp est similaire.



Une autre similitude entre le calcul sans serveur et les contrats intelligents est le fait que chaque transaction a un coût. Dans le cas d'AWS, le compte AWS du développeur est facturé pour les ressources consommées, tandis que dans le cas de la blockchain, celui qui a signé la transaction est facturé pour son exécution.

Avec cette analogie comme point de référence, discutons de l'exemple de développement de dapp que nous utiliserons tout au long de cette série.

Notre exemple : une application de chat basée sur la blockchain

L'exemple que nous utiliserons tout au long de cette série est une application de chat basée sur la blockchain. Ce n'est pas un exemple réel dans le sens où il n'y a pas de bonnes raisons commerciales d'utiliser une blockchain publique pour le chat (à mon avis). Le fait que tous les messages soient complètement publics et inclus de manière irréversible dans un enregistrement permanent est un inconvénient, pas une fonctionnalité. Cependant, la raison de choisir cet exemple est qu'il illustre divers concepts importants dans le développement de dapp tout en étant logiquement facile à suivre pour quiconque a utilisé quelque chose comme Facebook Messenger, Telegram ou Signal.

Le code de cet exemple est disponible sur mon GitHub. Le README de ce référentiel donne des instructions pour configurer un environnement de développement pour interagir avec le code et une idée de base de la façon d'utiliser le contrat. Cette série de publications va nous faire plonger beaucoup plus profondément dans le code et dans son fonctionnement.

Pour ancrer la discussion des principes du développement de smart contract, voici un aperçu de la manière dont fonctionne le contrat de chat.

- Chaque individu qui souhaite participer au réseau de chat déploie sa propre version du smart contract.
- Chaque instance du contrat maintient une liste de comptes qu'elle connaît (contacts, demandes de contact en attente, etc.). Il stocke également les messages qu'il a reçus (et certaines métadonnées sur ces messages).
- Pour envoyer un message à quelqu'un d'autre, vous devez d'abord l'avoir comme "contact". Cela fonctionne comme on pourrait s'y attendre : Alice envoie une demande de contact à Bob, si Bob l'accepte, alors Alice et Bob deviennent des contacts l'un de l'autre, sinon ils ne sont pas des contacts.
- Chaque instance du contrat à un "propriétaire" qui est capable d'envoyer des messages et d'envoyer/accepter des demandes de contact.

Principes de développement de smart contracts

Il y a trois concepts connexes que je tiens à souligner et qui sont importants pour le développement de smart contracts, mais qui ne sont peut-être pas présents dans le développement de logiciels traditionnels. Ce sont :

1. un esprit de confrontation,
2. l’économie
3. protéger les invariants avant de passer des appels entre contrats.

Un esprit de confrontation,

La première chose importante à retenir lors du déploiement sur une blockchain publique est que n'importe qui dans le monde entier peut interagir avec votre code. Si votre contrat intelligent peut effectuer une action sensible (par exemple, lors de l'envoi de messages dans le contrat de chat, vous ne voudriez pas qu'une personne puisse se faire passer pour vous), vous devez explicitement vérifier l'autorisation afin que seuls les comptes autorisés puissent effectuer avec succès l'action (c'est pourquoi notre contrat de chat a la propriété "propriétaire"). Si vous avez une méthode qui prend une entrée, vous devez la valider avant de passer à la logique métier, car n'importe quel utilisateur aléatoire pourrait soumettre n'importe quelle entrée qu'il souhaite. En effet, l'idée d'un état d'esprit de confrontation va encore plus loin; non seulement un utilisateur pourrait soumettre une entrée incorrecte, mais il pourrait également soigneusement la façonner pour déclencher une vulnérabilité dans votre code. La seule façon d'empêcher cela est de ne pas avoir de telles vulnérabilités en premier lieu.

De même, la logique des contrats intelligents dépend souvent d'un protocole pour coordonner différents composants ensemble (par exemple, le protocole pour ajouter des contacts dans notre contrat de chat). Un utilisateur a-t-il un pouvoir d'agir dans ce protocole ? Que se passe-t-il s'il ne le suit pas correctement ? Ce sont des questions auxquelles vous devez répondre lors du développement d'un contrat intelligent, car les pirates informatiques tenteront d'exploiter votre contrat.

En résumé, vous devez toujours supposer que toute entrée externe est byzantine et vérifier explicitement le contraire avant de procéder. Vous devez pratiquer la prise de conscience des hypothèses que vous faites et toujours penser "comment pourrais-je briser cette hypothèse ?" chaque fois que vous réalisez que vous en faites une.

Économie

L'économie d'une application web typique est assez simple. Vous devez générer suffisamment de revenus pour couvrir le coût d'hébergement du serveur qui contient le code et les données utilisées par votre application. Les revenus peuvent provenir de différentes sources, mais les plus courantes sont les revenus publicitaires et les abonnements payants des utilisateurs.

Pour la blockchain, la situation est un peu plus compliquée, car chaque transaction doit être payée indépendamment. Les nouveaux produits de la blockchain cherchent à simplifier cette histoire. Par exemple, Aurora+ fournit quelque chose comme un "abonnement blockchain" qui permet un certain nombre de transactions gratuites. Mais jusqu'à ce que cela devienne la norme dans l'espace blockchain, il est toujours important de répondre à la question "qui paie pour cela?".

Souvent, c'est l'utilisateur qui paie pour chaque transaction car le paiement est lié au compte de signature (c'est-à-dire que le paiement est lié à l'identité / autorisation). Un modèle alternatif consiste à utiliser des "**méta-transactions**" (des transactions à l'intérieur des transactions) de sorte que le paiement soit effectué par le "signataire externe" tandis que l'autorisation est basée sur le "signataire interne". C'est ainsi que fonctionne Aurora+ par exemple. Malheureusement, comme ce n'est pas la façon par défaut dont les transactions blockchain fonctionnent, cela nécessite un travail supplémentaire de la part du développeur pour le réaliser.

Pour notre exemple d'application de chat, nous allons utiliser le moyen le plus facile et chaque utilisateur devra payer les coûts qu'il génère grâce à son utilisation. Après avoir pris cette décision, nous devons examiner les coûts possibles et nous assurer qu'ils sont couverts de manière appropriée. Par exemple, sur Near, le paiement de stockage est géré par le "storage staking". En gros, cela signifie que chaque compte a une partie de son solde verrouillée en fonction de la quantité de stockage qu'il utilise. Cela est pertinent dans notre contrat de chat car il stocke les messages reçus des autres utilisateurs. Par conséquent, nous devons nous assurer que ces autres utilisateurs couvrent le coût de stockage en attachant un dépôt suffisant avec leur message. De même, les demandes de contact créent une entrée dans le stockage, donc elles doivent également être accompagnées d'un dépôt. Si nous ne faisions pas respecter ces exigences de dépôt, les utilisateurs pourraient voler de l'argent les uns aux autres en envoyant de nombreux messages et en bloquant le solde total de la victime (remarquez comment cela est lié à la mentalité adversaire mentionnée précédemment).

En résumé, lors de la conception d'une dapp, il est toujours important de réfléchir aux coûts impliqués et à la façon dont ils sont payés, que cela implique l'ajout de vérifications pour les dépôts ou l'utilisation de méta-transactions.

Assurez-vous des invariants avant de faire des appels de contrat croisés

Ce dernier point est subtil. Dans une application typique, tout le code est lié dans le même binaire. Lorsque vous appelez une fonction dans une bibliothèque, cela ne déclenche généralement aucune communication, mais ajoute simplement une nouvelle trame sur la pile et exécute du code à partir d'une autre partie du binaire. Dans un environnement blockchain, les choses sont un peu différentes.

Faire un appel à un autre contrat ressemble davantage à l'appel d'un tout autre processus qu'à l'appel d'une bibliothèque. Encore une fois, nous devons adopter un état d'esprit adversaire et réaliser que nous n'avons aucune idée de ce que cet autre processus pourrait faire ; en effet, il pourrait essayer de faire quelque chose de délibérément malveillant. Un vecteur d'attaque courant est d'avoir l'autre processus rappeler notre contrat et l'exploiter car notre contrat ne s'attendait pas à ce qu'un nouvel appel arrive pendant qu'il attendait une réponse à l'appel qu'il avait initié. Cela s'appelle une "attaque de réentrance" et c'était la source de l'un des hacks les plus célèbres sur Ethereum, celui qui a entraîné la création d'"Ethereum Classic" (Ethereum Classic a rejeté la "hard fork" qui était la réponse de la Fondation Ethereum au hack).

Sur Near, ce problème est encore plus prononcé en raison de la question de l'atomicité. Dans la machine virtuelle Ethereum (EVM), chaque transaction est "**atomique**" dans le sens où toutes les actions résultant de la transaction sont engagées dans l'état de la blockchain ou aucune d'entre elles ne l'est (la transaction entière est "reverted"). Cela signifie qu'une attaque de réentrance peut être contrecarrée en utilisant une inversion ; tout ce qui s'est passé sera annulé, gardant le contrat en sécurité. Ce modèle est même inclus dans l'exemple Mutex de la documentation

officielle de Solidity. Cependant, dans le runtime de Near, l'exécution des contrats est indépendante les uns des autres ; ils ne sont pas atomiques. Donc, si une transaction cause le contrat A à appeler le contrat B, et que B rencontre une erreur, alors les changements d'état qui se sont produits dans A resteront.

Cela a été beaucoup d'histoire et de théorie, mais quelle est la leçon pratique à retenir ? Le point est que vous devez vous assurer que votre contrat est dans un "bon état" lorsque vous appelez un autre contrat. C'est-à-dire que si votre logique de contrat repose sur des invariants, ils doivent être corrects au moment de l'appel. À titre d'exemple simple, supposons que nous avons un contrat avec une fonction de transfert. L'invariant à maintenir est que les jetons ne sont pas créés ou détruits lors d'un transfert. Si, pour une raison quelconque, un appel à un autre contrat était nécessaire pendant le transfert, il serait incorrect de débiter un compte et ensuite de faire l'appel sans créditer l'autre en premier. C'est parce que l'invariant sur les jetons qui ne sont pas détruits serait rompu lorsque l'appel externe serait exploitable. Un exemple dans ce sens est également inclus dans la documentation de Near.

Résumé

En résumé, nous introduisons une nouvelle série d'articles donnant une introduction au développement de smart contracts sur Near en utilisant Rust. Nous avons discuté de l'exemple de contrat de chat que nous utiliserons tout au long de la série, ainsi que de quelques principes généraux à garder à l'esprit lors du développement d'applications basées sur la blockchain. Dans le prochain article, nous plongerons plus en détail dans le code pour discuter des détails techniques de la mise en œuvre du contrat. Cela servira d'exemple pour le SDK Rust de Near, illustrant des concepts qui s'appliqueront à tous les types de contrats du monde réel que vous souhaiteriez écrire.

Si vous appréciez cette série d'articles de blog, veuillez nous contacter chez Type-Driven consulting. Nous sommes heureux de fournir des

services de développement de logiciels pour les dapps, ainsi que des matériaux de formation pour vos propres ingénieurs.