PROGRAMMEZ!

#207 - mai 2017

le magazine des développeurs

Amazon Alexa

Etendre et personnaliser Alexa

Android O

Toutes les nouveautés

L'art du debug

© jgroup

Cloud Computing + DevOps : le duo de choc

Monter un **cluster Docker** avec **ClusterHat**

M 04319 - 207 - F: 6,50 € - RD

Printed in EU - Imprimé en UE - BELGIQUE 7 € - Canada 9,80 $ CAN - SUISSE 13,10 FS - DOM Surf 7,50 € - TOM 1020 XPF - MAROC 55 DH

L'art du **Debug**

• Sarah Buisson
Ingénieur Java et passionnée de Craftsmanship, chez **Xebia.**

Souvent ignorés, le debug et la maintenance applicative sont pourtant des domaines de compétence à part entière : avec des outils et des méthodologies qui leur sont propres. En raison de ce manque de reconnaissance, la plupart des développeurs deboguent sans chercher à étendre ou à transmettre leurs compétences dans ce domaine. Cet article vous présentera les outils et méthodologies de debugging les plus courants, applicables avec tous les IDE et langages modernes.

Qu'est-ce qu'un bug ?

Un bug est une différence entre le comportement attendu et le comportement obtenu. Ce n'est pas à proprement parler une erreur dans le code. C'est son symptôme. Quelque part en amont du projet se cache une instruction erronée : l'origine du bug (ou root-cause).
Deboguer, c'est donc chercher et corriger l'origine d'un bug.

UN PEU DE THÉORIE ET DE MÉTHODOLOGIE

La base de toutes les stratégies de debugging repose sur 4 points :

- Reproduire le bug ;
- Diagnostiquer son origine ;
- Corriger ;
- Conclure.

Étape 1 : reproduction de l'erreur

Avant toute chose, vous devez commencer par essayer de reproduire l'erreur. Pourquoi ? Tout simplement parce que si vous ne reproduisez pas l'erreur :

- Vous ne saurez pas si l'anomalie existe réellement ;
- Vous n'aurez jamais la certitude que votre correction ait été efficace.

Dans un monde idéal, la personne remontant le bug donne un compte-rendu détaillé de comment reproduire le bug, sa présence est confirmée par plusieurs personnes, le comportement attendu est précisé.
Dans la réalité, c'est parfois plus compliqué.
Un scénario de reproduction peut être la suite d'instructions remontée par la recette pour produire le bug, un test unitaire qui ne passe plus, etc. Tout ce qui vous permettra de reproduire facilement et à volonté l'erreur.

L'importance du contexte

Déboguer en local (dans un environnement de développement) est plus simple que de debugger un test d'intégration continue retourné en erreur. Cependant déboguer l'Intégration Continue reste plus facile que de corriger une erreur remontée par l'équipe de recette pendant les tests manuels. Erreur qui sera moins coûteuse que le bug remonté par un utilisateur final depuis la production. Plus vous montez dans les environnements, moins vous aurez d'outils et plus vous aurez un impact sur le travail des autres. L'idée est de parvenir à détecter les bugs le plus tôt possible d'une part, mais surtout de les reproduire sur des environnements les plus proches possibles de vous afin de bénéficier de plus de liberté et des meilleurs outils à votre disposition.

Le processus de remontée de l'anomalie

Le processus de remontée des anomalies influence énormément la facilité avec laquelle vous reproduirez le bug. N'hésitez pas à remonter vos besoins aux équipes de recettes. Idéalement, chaque anomalie remontée doit :

- Indiquer comment reproduire l'anomalie ;
- Préciser quels sont le comportement attendu (spécifications fonctionnelles) et constaté (capture d'écran, stack-trace, identifiants, etc.) dans quel contexte (sur quelle plateforme, pour quel utilisateur, sous quel navigateur, etc.) ;
- Indiquer la priorité de l'anomalie (mineure, prioritaire, bloquante) ;
- Ne remonter qu'une anomalie à la fois (avec un éventuel lien vers d'autres bugs liés).

Kiss: Keep It Simple And Stupid: reproduction et tests unitaires

Reproduire le bug, c'est bien. Le reproduire facilement, c'est mieux. Lors des phases de diagnostics et de correction, vous allez devoir reproduire le bug à de nombreuses reprises. C'est pour cela qu'il vous faut alléger le scénario de reproduction au maximum, jusqu'à aboutir à un cas de test le plus simple possible (mais qui reproduit toujours l'erreur).
L'idéal est de reproduire le bug via un test unitaire, afin de pouvoir le tester d'un simple clic. Cela vous permettra de vous focaliser sur l'analyse et la correction du bug, et non sur un scénario de reproduction trop compliqué.

Et une fois le bug corrigé, que faire du test ?

Idéalement, gardez-le et modifiez les cas de tests existants. En effet, ce n'est pas un seul bug que vous avez mais deux. L'un dans le programme, l'autre dans vos tests, qui ont laissé le bug passer inaperçu jusqu'à présent. Les modifications dans les tests font donc partie intégrante du correctif : elles doivent être livrées avec.

Delta debugging

Supposons que votre bug ait un scénario de reproduction très complexe. Le delta debugging consiste à supprimer et remettre une-à-une les étapes du scénario de reproduction, afin de déterminer lesquelles sont nécessaires à la reproduction de l'anomalie. Si la disparition d'une étape cause la disparition du bug (ou l'apparition d'un autre bug) alors nous la gardons, sinon nous la retirons du scénario.

Les bugs impossibles à reproduire

Parfois, vous ne parviendrez pas à reproduire le bug. Cela ne veut pas dire qu'il est corrigé pour autant. Vous devez trouver pourquoi le bug n'est pas reproductible :

- Si le bug a été corrigé par un autre correctif, vous devez trouver lequel ;
- Si le bug est intermittent, vous devez isoler les causes qui le rendent inconstant, afin de pouvoir reproduire le bug systématiquement. Nous reparlerons du traitement des bugs intermittent dans la suite de l'article ;

- Si c'est une question de données, vous devez importer ou fabriquer les données nécessaires en local ;
- Si c'est une question d'habilitation, "pair-debuggez" avec quelqu'un habilité à reproduire le bug ;
- Si c'est une question de plateforme, vous devrez vous contenter des outils disponibles sur la plateforme sur laquelle est reproductible le bug ;
- Si c'est une question de timing : vous devez trouver le bon enchaînement d'actions qui provoque le bug, en vous basant sur les logs existants. À noter que l'ajout de logs peut perturber la reproduction des bugs de timings.

Étape 2 : formulation d'hypothèses et diagnostics

Une fois l'erreur reproduite, vous pouvez l'analyser pour déterminer son origine. Lire le code et l'exécuter ligne par ligne est certes une bonne pratique, et vous permettra de découvrir le projet, mais implique de lire, comprendre et maîtriser pratiquement l'intégralité du code avant de trouver la cause. Heureusement, il existe d'autres stratégies vous permettant de restreindre votre champ de recherche et de "cibler" la cause de l'erreur.

Notions de base

Vérifiez ce qui a été modifié récemment.
Les bugs n'apparaissent pas spontanément. Une anomalie qui ne s'est jamais produite auparavant n'a que 3 causes possibles :
- Soit le cas de tests n'avait jamais été vérifié auparavant ;
- Soit le contexte du projet a changé (nouvelles données, nouvelles dépendances) ;
- Soit une erreur a été récemment introduite dans le code.

Voyez large.
Ne vous limitez pas à une seule hypothèse sur la source du bug. Formulez-en plusieurs, triez-les de la plus évidente à la moins probable et vérifiez-les toutes. Ne vous restreignez pas à une seule section du code. L'erreur n'est pas forcément là où vous regardez.

Générez plus de données pour plus d'hypothèses.
Ne vous limitez pas au scénario de reproduction original.
Essayez de reproduire le bug de plusieurs façons, et variez les données de vos tests. Ainsi, vous pourrez déterminer les points communs entre les différents scénarii reproduisant l'erreur.
N'oubliez pas de tirer des conclusions des scénarii ne reproduisant pas le bug : ils vous permettent d'invalider certaines de vos hypothèses.

Isolez votre code.
Mockez ("Bouchonner") les parties du projet, vous permettra de réduire les interférences et ainsi de concentrer votre attention sur l'erreur.

Parlez du bug à vos collègues.
Expliquer le problème à quelqu'un d'autre aide à la compréhension du bug ("Rubber Duck Debugging"). Il est possible qu'ils aient eu affaire à des situations similaires dans le passé et qu'ils puissent vous conseiller certains outils.

Diagnostic par observation

Il s'agit d'obtenir autant d'informations possibles sur le bug en se basant sur l'exécution du programme : soit en consultant les logs, soit en plaçant des points d'arrêt sur le chemin du bug, afin de trouver la cause de l'erreur. Cette approche nécessite d'avoir une bonne connaissance du programme ou de l'architecture à déboguer. Il faut être capable de cibler rapidement les classes susceptibles de contenir l'origine du bug.
Au sujet du log-debugging
Débugger en ajoutant des logs est sujet à débat : alors que certains utilisent cette pratique régulièrement, d'autres la rejettent fortement.
Le problème est que les logs sont un outil de debug aussi puissant que dangereux :
Ils vous donneront des informations précieuses sur les exécutions passées, mais ils ont un impact sur l'exécution du programme et sont soumis à diverses réglementations suivant les projets. À noter que pour les bugs reproductibles en local et une connaissance avancée de votre Environnement de Développement Intégré (IDE), vous permettront d'arriver au même résultat que les logs aussi facilement, sans recompilation ni risque d'effets de bord. Si en revanche l'anomalie ne se manifeste que sur une machine distante, il sera souvent plus simple de mettre en place davantage de logs plutôt qu'un remote debug.

Diagnostic par induction

Pour rappel, l'induction représente la logique suivante : "*si A, alors B*" entraîne "*Si non B, alors non A*".
L'idée consiste à partir d'une version de l'application qui fonctionne, pour progressivement arriver à la version sur laquelle le bug se produit. Prenons l'exemple d'une régression. L'anomalie est brusquement apparue avec la dernière release. Nous allons charger la dernière version du programme sans le bug, tester la présence de l'anomalie, puis récupérer le commit suivant, retester, etc. Jusqu'à isoler sur le commit qui contient l'anomalie. On pourra alors centrer la recherche exclusivement sur les fichiers modifiés par le commit ayant introduit le bug.
Tous les outils de versionning conviennent à cette stratégie, mais les fonctionnalités telles que git blame et git bisect peuvent vous faire gagner du temps en ciblant plus efficacement les commits à tester.

Stratégie Back-tracking : le debug en marche arrière

C'est une approche très populaire de debugging.
Il s'agit de chercher la cause directe du bug remonté (le symptôme), puis la cause de cette cause, etc. plusieurs fois de suite, jusqu'à trouver l'origine initiale du bug.
Nous trouvons ainsi l'enchaînement d'instructions ayant provoqué le bug pour aboutir à son origine.
Dans la pratique :
- Commencez par placer un point d'arrêt sur une ligne de code où le bug a été constaté ;
- Exécutez le scénario de reproduction ;
- À partir des variables d'exécution et du code, cherchez ce qui a pu provoquer le bug, et placez un point d'arrêt sur chaque ligne de code responsable ;
- Exécutez le scénario de reproduction. Analysez. Ne gardez que les points d'arrêt où le bug a été constaté ;
- Recommencez.

Vous finirez ainsi par trouver l'origine du bug. À noter que certains langages et IDE offrent la possibilité de remonter le fil d'exécution en marche arrière, évitant ainsi de refaire plusieurs fois le scénario de reproduction. Cette approche n'est malheureusement efficace que si l'on a réussi à isoler, au moins en partie, la portion de code contenant le bug, au risque de devoir parcourir l'intégralité de l'application.

Où placer vos points d'arrêt

Chaque programme possède des sections critiques qui doivent être analysées en priorité : les points d'entrée, les appels aux APIs tierces, le début d'une méthode publique, l'entrée dans un algorithme complexe, etc. Placer un point d'arrêt dans ces sections du code vous donnera de précieuses informations sur l'état du programme : si l'anomalie est déjà survenue ou non, mais aussi si des signes précurseurs de l'anomalie sont déjà apparus (nombreux champs vides, listes d'erreurs pleines, données incohérentes, etc). Trouver ces points vient avec l'expérience sur le projet.

Saff Squeeze

Le Saff Squeeze (du nom d'une technique de plaquage au football américain) consiste à valider et retirer un à un les éléments valides d'une méthode à tester, afin de prendre "en sandwich" l'erreur. En partant d'un test reproduisant l'anomalie, avec une assertion qui échoue :

- "Inlinez" la fonction à tester, (remplacez un appel de la fonction par son code, par exemple via la fonction *inline* de votre IDE) ;
- Ajoutez de nouvelles assertions au coeur du code de la fonction inlinée, testant le bon fonctionnement de cette méthode ;
- Lancez le test : certaines assertions vont passer, d'autres vont échouer ;
- Nettoyez : bouchonnez (mockez) les fonctions dont les assertions sont passées, et supprimez le code après l'assertion échouée. Les fonctions ayant échoué se retrouvent ainsi prises en sandwich entre les mocks d'un côté et les assertions de l'autre ;
- Recommencez : Inlinez les fonctions restantes et ajoutez de nouvelles assertions.

Progressivement, vous arriverez à un test de plus en plus unitaire, qui à la fois reproduit l'erreur et cible précisément son origine. À noter qu'il n'y a que peu d'intérêt à comiter le test unitaire final généré : il ne fait que dupliquer le code en erreur. Il est souvent plus intéressant de garder celui d'une ou deux itérations précédentes.

Étape 3 : Correction du bug

Vous avez réussi à reproduire et identifier la cause de l'erreur, félicitations ! Nous allons enfin pouvoir littéralement "déboguer", c'est-à-dire retirer le bug. La correction implique de concevoir et mettre en place les modifications qui corrigent le problème, sans induire de régression, et en respectant les standards de qualité du projet.

Comment éviter les régressions ?

Les régressions sont un sujet complexe et récurrent, auquel il faudrait plus d'un article pour y répondre. Néanmoins, une bonne compréhension du projet, aussi bien d'un point de vue technique que fonctionnel, une bonne couverture de tests (manuels et/ou automatiques) indépendante du développeur et des processus de qualité stricts (relecture de code, continuous testing / integration, métriques de qualité, alertes automatisées) permettent d'éviter la plupart des régressions.

Au sujet des quickfixs

Parfois, vous n'aurez pas le temps (ou l'opportunité) de faire les choses bien. Il est alors possible d'utiliser le quickfix : une rustine, un code temporaire ne respectant pas les standards de qualité habituelle ou tous les cas de test. Cependant, ayez en tête :

- L'utilisation du quickfix a un impact fort sur la qualité du projet et son futur. Cette décision ne doit pas être prise unilatéralement par le développeur. En revanche, il devra argumenter les avantages et inconvénients de son utilisation ;
- Un quickfix doit être temporaire : il doit être associé à une nouvelle story, qui apportera la correction finale propre ;
- L'information doit être conservée : votre quickfix va masquer une partie des informations relatives à l'erreur, rendant difficile la correction finale. Stockez quelque part les logs d'avant le quickfix, vos hypothèses concernant le bug, etc. ;
- Le code d'un quickfix ne doit pas être confondu avec le reste du projet et doit être visible du premier coup d'oeil : par exemple en les commentant d'un //TODO: Quickfix de la STORY-123

Étape 4 : la conclusion du bug

Votre travail ne s'arrête pas une fois le bug corrigé.
Comment ce bug a-t-il pu se produire ? Comment a-t-il pu passer inaperçu jusqu'à aujourd'hui ? N'y a-t-il pas d'autres bugs similaires déjà existants ? Comment aurais-je pu identifier le problème plus facilement (plus d'informations sur le scénario de reproduction, plus de documentation) ? Comment s'assurer que ni ce bug, ni un bug semblable ne se produise à nouveau ? La documentation a-t-elle été mise à jour ? L'équipe de qualification a-t-elle toutes les informations nécessaires pour tester la correction ?
Il vous faudra répondre à ces différentes questions, en apportant des solutions si nécessaire.

En Bonus : la stratégie complète pour les bugs intermittents

Vous avez essayé d'isoler toutes les causes possibles, mais rien à faire : votre bug reste intermittent ou impossible à reproduire sur votre machine locale. Le problème est double : non seulement l'anomalie va être difficile à diagnostiquer, faute de pouvoir être observée "in vivo" , mais en plus il va être difficile de confirmer l'efficacité du correctif.
Pour reproduire, diagnostiquer et corriger un bug intermittent, cette stratégie repose sur des outils de monitoring (via des points d'arrêt, des metrics, des logs, etc. suivant la plateforme où le bug est reproduit) :

- À chaque fois que l'anomalie est reproduite, nous la monitorons ;
- Une fois suffisamment de données de monitoring recueillies, l'analyse de ces données permettra de trouver l'origine du bug ;
- Cette erreur originelle va également être monitorée afin de valider qu'il s'agit bien de la cause du bug : à chaque trace de l'erreur originelle, devrait correspondre une trace de l'anomalie reproduite ; Si les deux monitoring ne correspondent pas, alors l'hypothèse est fausse.
- Une fois l'origine du bug identifiée, nous allons effectuer la correction de l'anomalie, tout en gardant les traces. Il est en effet important de vérifier que les traces de l'erreur originelle ne soient plus associées à celles confirmant la présence de l'anomalie.
- Une fois seulement cette vérification effectuée, nous pouvons retirer les outils de monitoring.

Cette méthode de résolution est très lente. Il n'est pas inhabituel de devoir attendre plusieurs semaines avant de recueillir suffisamment d'information de monitoring. Vous pouvez éventuellement envisager de livrer un quickfix en attendant.

Un exemple d'utilisation :
La situation : tous les dossiers doivent être affichés avec un utilisateur.
Le bug : certains dossiers sont affichés sans utilisateurs.
Le problème : je ne parviens pas à reproduire l'anomalie, tous mes dossiers sont créés avec utilisateur.

Je vais donc suivre la méthodologie précédente :

- Je rajoute un log à chaque fois qu'un dossier est affiché, indiquant s'il a ou non un utilisateur. Au bout d'une semaine, je récupère les logs sur les différents dossiers sans utilisateur. Leur analyse me permet de constater qu'il s'agit de dossiers sous-traités, avec un fort turnover des utilisateurs.
- Je formule 3 hypothèses :
  - Soit ces dossiers n'ont jamais eu d'utilisateurs,(hypothèse 1) ;
  - Soit les dossiers ont eu leur utilisateur retiré,(hypothèse 2) ;
  - Soit les utilisateurs ont été supprimés.(hypothèse 3).
- Je rajoute des logs pour chacune de ces hypothèses :
  - Pour chaque création de dossier utilisateur,(scénario 1) ;
  - Pour chaque modification de dossier sans utilisateur,(scénario 2) ;
  - Pour chaque suppression d'utilisateur.(scénario 3).

Au bout d'une semaine, je consulte à nouveau ces logs et constate :

- Aucun dossier n'a été créé sans utilisateur, mais des dossiers ont néanmoins été affichés sans utilisateur (hypothèse 1 invalidée).
- Certains dossiers ont eu leurs utilisateurs retirés, mais aucun de ces dossiers n'a été affiché par la suite (hypothèse 2 invalidée).
- À chaque fois qu'un utilisateur est supprimé, le dossier est affiché sans utilisateur (hypothèse 3 validée).

- Je corrige donc l'hypothèse 3. Désormais la suppression d'un utilisateur n'est possible que s'il n'a plus aucun dossier.
- Une semaine plus tard, je consulte les logs :
  - Des utilisateurs ont de nouveau été supprimés .
  - Aucun dossier sans utilisateurs n'a été affiché.

L'anomalie est donc corrigée, je peux retirer les logs.

DEBUG ET DEBUGGER : AU-DELÀ DES POINTS D'ARRÊT

Comment fonctionne un debugger ?

Pour la majorité des langages managés/interprétés modernes, ce n'est pas l'IDE mais la machine d'exécution elle-même qui s'exécute en mode "debug", en mettant à disposition un port distant. L'IDE se contente de se connecter à ce port, et ce que l'on appelle communément "débugger" dans un IDE désigne en fait l'IHM entre le programme et l'utilisateur, qui fait en permanence le mapping entre le code source et le code compilé en cours d'exécution, donnant ainsi l'illusion que c'est l'IDE qui interprète et exécute le programme.

> Machine d'exécution : la JVM de Java, la CLR .NET, le moteur Javascript etc., c'est-à-dire le tourne-disques sur lequel se joue votre programme

Pour les langages compilés et sans machine d'exécution virtuelle, le code déjà compilé est modifié directement en mémoire par le debugger lors de l'exécution, afin de lever une exception à chaque ligne de code. Ces exceptions seront attrapées par le debugger qui choisira de marquer l'arrêt, exécuter la ligne, etc.
Dans tous les cas, le debugger a un accès complet à la mémoire, aux threads et au code (source ou compilé) du programme en cours d'exécution.

Fonctionnalités méconnues du debugger

Au-delà du point d'arrêt et du pas à pas, la plupart des debuggers modernes proposent des méthodes avancées.

Points d'arrêt avancés

Au-delà du point d'arrêt standard, vous avez également :

- Le conditionnel, qui ne s'activera que si l'instruction de votre choix est validée ;
- L'itératif, actif après seulement un certain nombre d'itérations ;
- Des points d'arrêt à chaque accès/modification d'une variable ;
- … À chaque instanciation d'une classe ;
- … À chaque levée d'exception ;
- Seulement actif pour une instance donnée (breakpoint Instance) ;
- Des points d'arrêt reliés entre eux, actifs seulement si un autre point d'arrêt lié a été activé précédemment ;
- Des filtres, afin de stopper le process seulement si l'exécution est passée par certains classes/packages au préalable ;
- Des points d'arrêt valables seulement sur une instance d'une classe avec un hashcode particulier.

Il est bien évidemment possible de combiner les différents types de points d'arrêt *(par exemple : s'arrêter uniquement sur les NullPointerException lancés depuis le package monprojet.service).* Prenez le temps de chercher ces options dans votre environnement de développement.

Navigation avancée du debugger

Outre les classiques "aller à la ligne suivante", "rentrer dans la méthode" et "ignorer tous les points d'arrêt", il vous sera possible de :

- Exécuter un extrait de code sélectionné / saisis ;
- Aller jusqu'à la ligne de mon curseur ;
- Continuer jusqu'à la fin du bloc ;
- Ignorer une ligne de code ;
- Retourner en arrière, jusqu'au début du bloc d'exécution (avec ou sans conservation de la mémoire).

Encore une fois, ces fonctionnalités ne sont pas présentes sur tous les IDE ni possibles pour tous les langages.
Une bonne connaissance des raccourcis clavier de la navigation avec le débogueur vous permettra de gagner en efficacité lors de vos sessions de debugs.

Call Stack, Variables et Watcher avancés

La plupart des IDE propose de visualiser la call stack en cours (la "suite de méthodes appelantes"), et les variables accessibles pour le scope en cours. Certains proposent également de filtrer les étapes de la call stack pour masquer les méthodes de votre framework d'inversion de dépendance par exemple. Il est également possible de modifier ces variables "à la volée", et de créer vos propres "vues" dans le Watch, ainsi que de positionner des points d'arrêt dessus actifs à la consultation et/ou changement sur les variables et vues.

Spécificités des debugger front

Les debugger intégrés aux navigateurs Web sont généralement moins avancés que ceux intégrés aux IDE, mais proposent d'autres fonctionnalités :

- Des points d'arrêt sur un élément d'un DOM : toute modification de l'élément arrêtera le script qui en est à l'origine ;
- Des points d'arrêt sur des events et listeners ;
- Des points d'arrêt sur des requêtes HTTP (XHR breakpoint) ;
- La possibilité de décomposer le CSS d'un élément du dom ;
- La possibilité de simuler l'aspect du site Web sur un autre device, version du navigateur, etc. ;
- L'état de la mémoire, des caches, etc. ;

- Des outils de profiling ;
- Un aperçu de la timeline de chargement et des trames réseau échangées.

ET À PART LE DEBUGGER, QUELS SONT LES AUTRES OUTILS ?

Logs : comment en tirer le meilleur

Les logs sont le plus ancien et le plus universel des outils de debugger. Leur utilité est quadruple. En vous donnant des informations sur leur contexte, ils vous permettent de constater les erreurs, de comprendre leurs causes et enfin de constater la disparition de l'erreur, après correction. Et ce, sans que l'utilisateur final ne s'en rende compte.
Pensez à spécifier un niveau de criticité approprié à chaque log. Cela vous permettra d'isoler rapidement les comportements anormaux des simples traces informatives. Vous pouvez facilement augmenter l'utilité de vos logs en apprenant à les requêter efficacement. Des outils d'analyse de logs peuvent également vous faciliter la tâche.
Exemples : Logstash, Kibana

Logs des flux d'entrées/sorties

Lister les entrées-sorties d'un programme vous aidera à comprendre les données qui transitent dans votre système et à isoler les causes d'un bug. À noter que l'AOP permet de mettre rapidement en place des logs des appels entrants et sortants.
Des outils d'analyse-réseau tels que wireshark et fiddler peuvent vous aider à obtenir le même résultat, en local.

Serveur de mock & proxy

Un serveur de mock permet de simuler facilement un serveur distant à un instant donné, et retourner exactement les réponses HTTP souhaitées pour simuler un scénario.
Exemple : Wiremock (utilisable à la fois en serveur indépendant ou en lib de mock durant les tests d'intégration).
À noter que certains de ces serveurs ont une fonctionnalité "d'aspiration". Le serveur fait office de proxy entre le véritable serveur et le programme pendant l'étape de configuration, puis se transforme en véritable serveur de mock, renvoyant pour une requête donnée, la même réponse que celle qu'il a enregistrée précédemment.
Exemples : api-mocking-proxy, MockServer
Citons un exemple : une anomalie liée aux réponses d'un serveur distant ne peut être reproduite qu'une fois par jour. Nous mettons en place un MockServer en tant que proxy entre notre application et le serveur distant, et nous reproduisons l'erreur pour enregistrer les réponses renvoyées. Ensuite, nous n'aurons qu'à passer le MockServer en mode "mock" afin de reproduire le contexte de l'anomalie à volonté.

Thread Dump, Memory Dump et Profiler

Un dump permet de récupérer une copie de l'état d'un programme à un instant donné dans le but de l'analyser plus tard via un logiciel dédié. Ils sont généralement utilisés pour analyser les problèmes de performance et les fuites mémoire. La plupart des machines virtuelles d'exécution sont livrées avec des outils de dump, qui leur sont propres.
Exemples : jmap, Managed Stack Explorer
Citons un exemple. L'application connaît d'importants ralentissements tous les jours à 17h depuis la dernière mise en prod. Un memory dump nous informe que la majorité de la mémoire est occupée par 360 000 instances de la classe "TeaTimeSingleton".

Class	Instance Count	Total Size
class [I java.lang.Object;	595	48516
class java.lang.String	361997	43464
class fr.programmez.service.TeaTimeSingleton	360000	2880000
[illegible]	[illegible]	[illegible]

Les outils de profilage ("profiler") permettent une analyse dynamique d'un programme en cours d'exécution, aussi bien des threads que de la mémoire. Les plus basiques proposent juste des graphiques montrant l'évolution des principales métriques du système, les plus avancées permettent d'avoir un memory-dump actualisé en temps réel.
Exemple : JVMTI, dotTrace
A noter que certains environnements permettent d'effectuer un snapshot de vos programmes, puis d'en refaire une nouvelle instance qui reprend l'exécution là où elle s'était arrêtée.
Exemple : Compute Engine de Google

Modification du code à chaud

Modifier votre code et le déployer sans avoir à redémarrer votre machine d'exécution ni perdre vos données s'appelle le hot deployment. Cela vous permet de tester vos correctifs sans avoir à redémarrer votre machine d'exécution, et ainsi de gagner du temps.
C'est un domaine avec trop de variations et spécificités suivant les langages et framework pour qu'il soit abordé en détail dans cet article.
Exemples : JRebel, WatchJs.

Les tests automatisés

Comme vu précédemment, les tests automatisés sont l'outil idéal pour reproduire un bug à moindre frais. N'oubliez pas non plus les mocks, qui vous permettront de simuler des pans entiers de votre système, tout en vous concentrant uniquement sur les parties buggées.
Quelques librairies et outils supplémentaires permettront de réaliser facilement des tests reproduisant vos cas d'erreurs :

- En embarquant une base de donnée avec vos tests : H2 ;
- En mockant dynamiquement tout un serveur : wiremock.

Analyseur de code statique

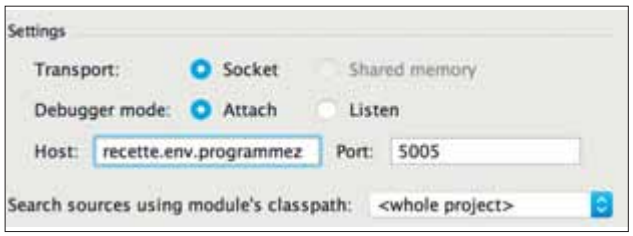
Une analyse statique du code permet d'éviter efficacement les erreurs les plus courantes, dans une démarche "préventive". Ils peuvent être configurés pour cibler une problématique en particulier (par exemple : la concurrence.). Ces outils sont disponibles aussi bien sur serveur distant que sous forme de plugin IDE, ces derniers ayant l'avantage de vérifier la qualité du code pendant la saisie.
Exemple : Sonar, Findbug

Remote debug

Il est possible de lancer tout programme sur une machine à distance ou un container, et de le debugger depuis votre IDE.
Cela est particulièrement utile dans le cas d'un bug impossible à reproduire sur la machine de dev, mais reproductible en intégration ou recette. (Attention au risque d'impact sur les autres utilisateurs).
Cela est particulièrement utile lorsque vous développez un programme pour un périphérique ou une machine dédiée qui ne peut héberger la plateforme de développement. Malheureusement, dans la pratique, cet outil est lourd à mettre en place. Vous aurez probablement besoin du Sysadmin pour ouvrir des ports et relancer la machine d'exécution.
Par exemple : un projet Java est lancé en mode debug sur une plateforme distante :

```
java -jar ./build/libs/gs-spring-boot-0.1.0.jar -agentlib:jdwp=transport=dt_socket,
server=y,suspend=n,address=5005 -Xdebug
```

L'IDE lance un process "Remote" afin de pouvoir debugger le processus.



… Ou debugger en lignes de commandes.

Si jamais les restrictions de la machine distante sont trop importantes, il est également possible de debugger en ligne de commande sur la plupart des machines d'exécution.

Exemples : gdb(C, .NET) , mais aussi jdb (java), ndb (nodeJs)

Et le reste ? Au sujet de vos outils, frameworks et autres technologies

Tous les frameworks, toutes les technologies, tous les outils que vous utilisez sont fournis avec des outils et fonctionnalités de debug. Sans exception. Citons quelques-unes de ces fonctionnalités :

- Vous aurez toujours un niveau de logs avancé (et désactivé par défaut). Que ce soit pour votre compilateur, votre IDE, le framework que vous utilisez, etc. ;
- De nombreux outils tournant sur une machine virtuelle d'exécution ont un mode ou une version alternative compatible avec le debugger de votre IDE (mvndebug, gulp-debug) ;
- Souvent les outils d'"Infra As Code" ont un mode "plan" qui détaillera le plan d'exécution prévu par le script, sans pour autant l'exécuter (Puppet, Terraform, Ansible, etc.) ;
- Les ORM et frameworks d'injection de dépendance récents ont un code suffisamment lisible pour qu'il soit intéressant de rentrer dans leur code via le débogueur afin de comprendre certains comportements et anomalies, et d'observer le contenu de leurs données internes (Spring, Hibernate, etc.) ;
- Tous les SGBD ont une vue des requêtes en cours d'exécution, et une vue des requêtes les plus lentes. Toutes ont une commande "explain", qui déroulera le plan d'exécution d'une requête.

```
explain select * from users  u
left join employees e
  on e.user_id = u.user_id

QUERY PLAN
Merge Left Join (cost=149.93..625.13 rows=25680 width=12)
Merge Cond: (u.user_id = e.user_id)
-> Index Only Scan using userid on users u (cost=0.15..84.16 rows=2400 width=4)
-> Sort (cost=149.78..155.13 rows=2140 width=8)
Sort Key: e.user_id
-> Seq Scan on employees e (cost=0.00..31.40 rows=2140 width=8)
```

Exemple : postgresql

QUAND LE DÉVELOPPEUR PASSE EN MODE DEBUG

On dit parfois qu'il faut concevoir son logiciel comme un architecte, coder comme un ingénieur et tester comme un barbare. Nous ajouterions "debugger comme un détective." Debugger implique un état d'esprit particulier, différent de celui de produire du code :

- N'ayez pas d'idées préconçues. Tant que vous n'avez pas trouvé l'origine du bug, toutes les parties du programme et de l'environnement sont "présumées coupables" ;
- Restez rigoureux et méthodique. Pour arriver à trouver l'instruction fautive, aucun élément ne doit vous échapper ;
- Connaissez vos outils : vous ne devez pas passer à côté d'une information vitale, juste parce que vous ne savez pas bien vous servir d'un outil ;
- Écoutez vos intuitions, mais vous devez toujours les vérifier. Une intuition validée est un fait. Vos conclusions doivent se fonder sur des faits, pas sur des intuitions ;
- Ne vous attendez pas toujours à une explication complexe. Même une faute de frappe peut causer un bug ;
- Ne modifiez pas votre code pendant la phase d'analyse du bug. Vous risquez d'introduire de nouvelles erreurs ou de modifier le bug initial par inadvertance ;
- Même si vous devez être rigoureux dans votre recherche, essayez de prioriser les sources potentielles de l'erreur. Vous ne pourrez jamais relire et analyser minutieusement chaque ligne du programme. Utilisez votre expérience pour gagner du temps et vérifiez les méthodes que vous suspectez le plus en premier. Mais ne négligez aucune piste pour autant ;
- Vous ne serez efficace que si vous gardez l'esprit clair et alerte. Debugger un code demande plus d'énergie que de l'écrire. Prenez des pauses, mangez, assurez-vous d'avoir suffisamment de sommeil, alternez avec d'autres tâches. Assurez-vous d'avoir un environnement de travail sain et de ne pas être interrompu pendant vos sessions de debugger. Si vous ne trouvez pas la cause du problème, réessayez le lendemain. Vous aurez un oeil neuf sur votre code ;
- N'en faites pas une affaire personnelle. Le programme ne tente pas de vous piéger, il traite juste les informations qu'il a de la manière dont cela lui a été demandé ;
- Même si, pris individuellement, chaque bug aurait pu être évité, la présence globale de bugs au sein du code que vous produisez est inévitable. Voyez vos bugs comme une opportunité d'apprendre, pas comme un échec personnel.

Un dernier conseil : la principale difficulté dont souffre le debug est qu'il n'est pas considéré comme un véritable domaine d'expertise. En conséquence, les développeurs échangent peu sur le sujet, la passation de compétence reste limitée, la littérature est assez pauvre, et bien souvent les pratiques tombent dans l'oubli. C'est une erreur ! Vous devez partager ce que vous savez ! Toujours ! Faites du "pair-debugging" ! Transmettez vos pratiques lors des passations. Écrivez une vraie "documentation du debug" pour votre projet, qui listera les requêtes les plus utilisées et les outils disponibles, plateforme par plateforme. Le debug est un domaine d'expertise à part entière, une discipline universelle, un art. À vous de le conquérir. •

Sources :

Le debbuger front : http://devtoolsecrets.com/
L'importance de la remontée de l'anomalie :
http://blog.xebia.fr/2015/11/23/qualifier-une-anomalie-pour-faciliter-sa-correction/
Saff squeeze : https://www.infoq.com/news/2008/11/beck-saff-squeeze