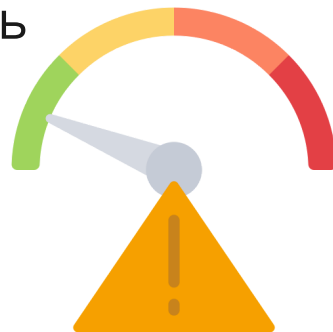
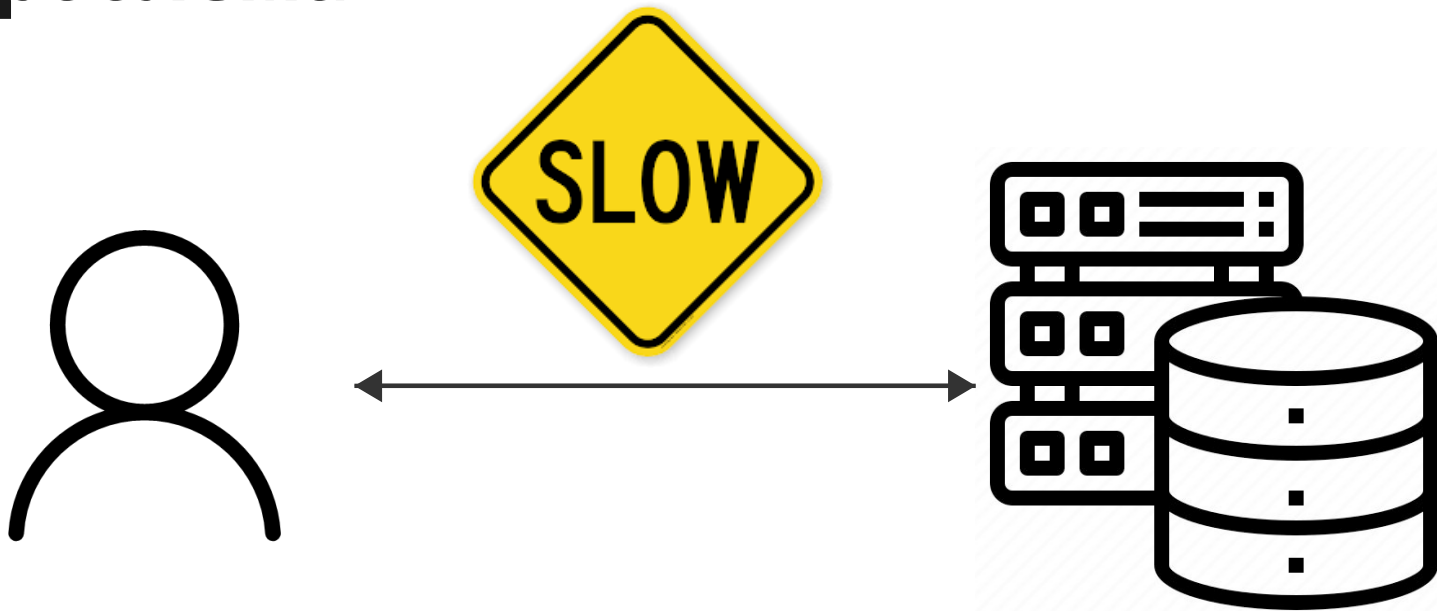


Backend тормозит

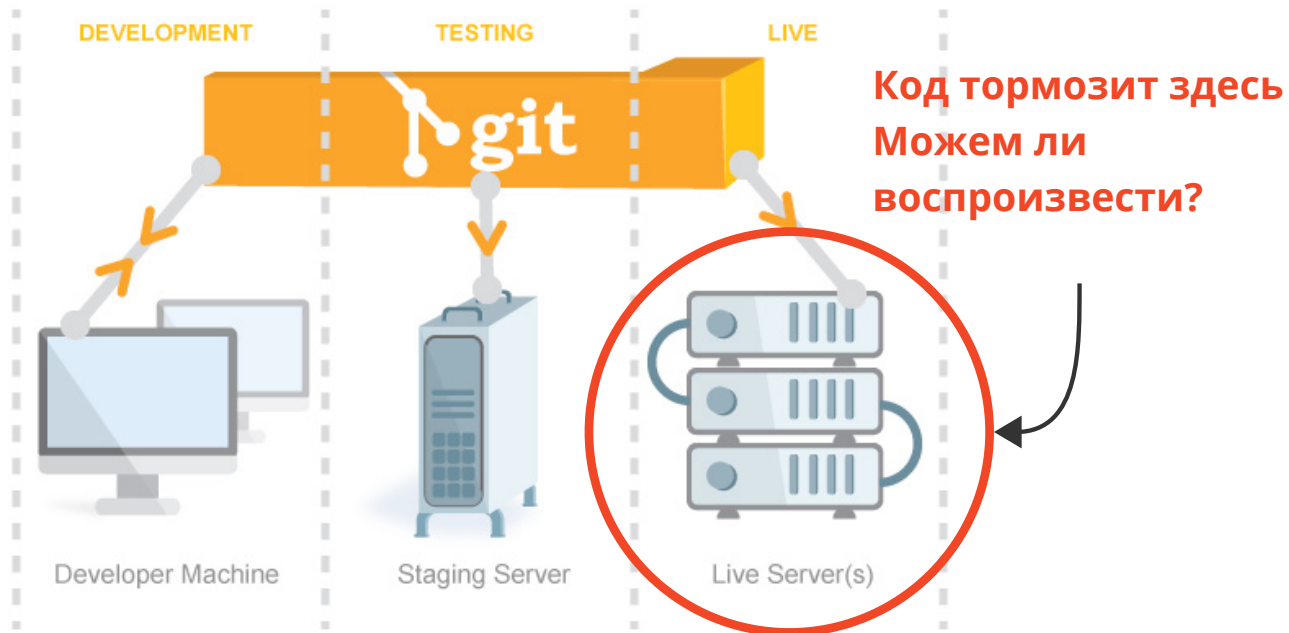
Пошаговое руководство что делать



Проблема



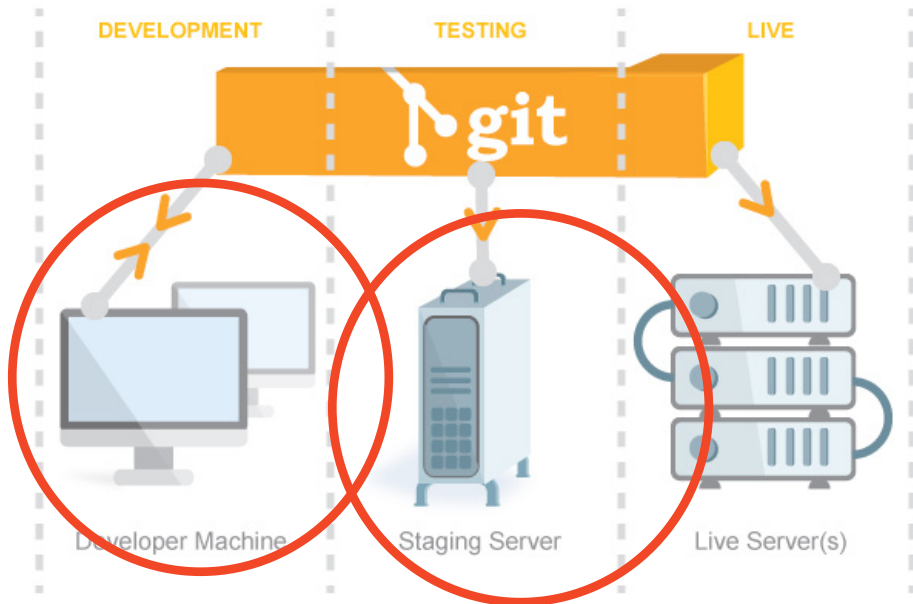
1. Повторяемость и окружение



1. Повторяемость и окружение

Вопросы к локальному окружению:

- Можно сгенерировать подобный сценарий?
- Повторяется ли проблема?
- Есть ли упор в производительность машины разработчика или стейджа
- Отсутствие сетевого взаимодействие или различие инфраструктур



1. Повторяемость и окружение

Проблемы на этом шаге

- Отсутствие возможности генерации подобных данных на стейдже или локально (например слишком много или данные зашифрованы)
- Разница инфраструктуры (например отсутствие каких-то балансировщиков или кэшей)
- Разница по железу (например в prod инфраструктуре тяжелые вызовы могут выполняться сильно быстрее/медленнее)



1. Повторяемость и окружение

Выводы по этому шагу:

- При невозможности воспроизведения проблемы придется искать причину на проде
- Разница в железе может сильно влиять требуется детальный анализ

Для локального окружения:

- Подключение к удаленным базам сильно влияет на время работы, нет смысла искать проблему
- Локально сложно воспроизвести необходимый объем данных
- Имеет смысл искать тяжелые последовательные операции

2. Быстрый анализ

Харкод с таймерами

Плюсы

- Легко и быстро сделать
- Время выполнения будет в логах
- Если проблема есть тормозящее место будет сразу видно
- После раскатки необходимо лишь поискать нужные логи

Минусы

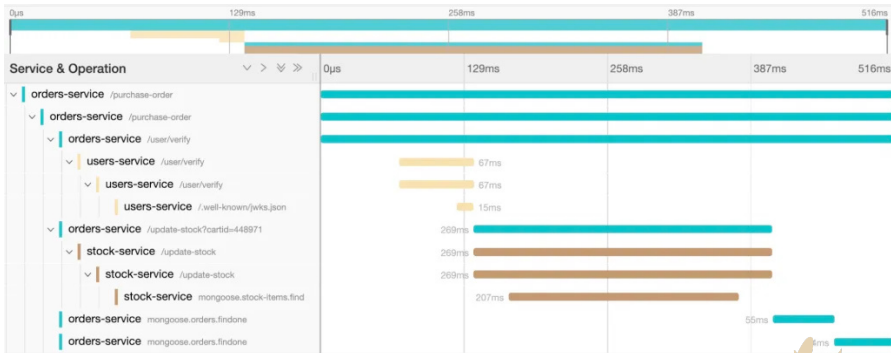
- Разметить все с нуля будет затратно, если не угадать с таймерами в нужных местах придется перекачивать
- Логи на проде читать неудобно, даже в современных системах рисовки и поиска по типу ELK

```
func SomeFunction() {  
    t0 := time.Now()  
  
    doSmtH()  
  
    TimeTrack(time.Since(start))  
}
```

2. Быстрый анализ

А как масштабировать это на несколько связанных систем + чтобы было удобно?

- Трейсинг в рамках одного сервиса
- Распределенная трассировка
- На практике передается заголовок с ID запроса между системами и по нему связываются все операции в один график
- Собирается лишь часть операций обычно 1%, если сработало для корня, все дочерние собираются тоже



Jaeger / OpenTracing



2. Быстрый анализ

Плюсы и минусы

Плюсы

- Если разметка есть достаточно найти лишь нужный запрос в UI
- Удобный UI для просмотра операций, не нужно соединять логи
- Удобная разметка, обычно span вставляется как log с передачей текущего контекста запроса (threadlocal или ctx)
- Можно выстроить анализ работы продуктовых сценариев на проде по трейсам и реагировать на деградации

Минусы

- Необходима развернутая инфраструктура сбора трейсов
- Необходима разметка операций
- Проблема может не воспроизводиться какое-то время, потому что не входит в 1% запросов
- На практике собирать много данных плохая идея, тяжело хранить + тратится много ресурсов узла
- Иногда операции тормозят от проблем вокруг, а не потому что есть проблемы в выполняемом коде



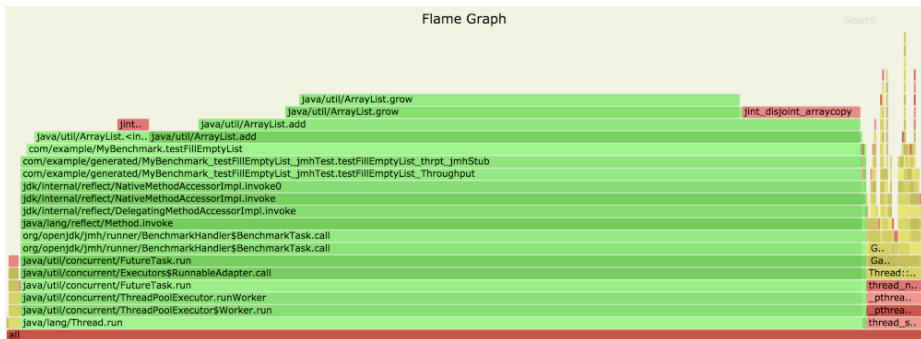
3. Глубокий анализ

Проблема все еще не решилась

Время собирать
профиль и вникать в
сложные графики,
но профили бывают
разные

Как получить такую картинку:

- Запустить сборку
профиля на время
- Выполнять необходимые
операции



3. Глубокий анализ

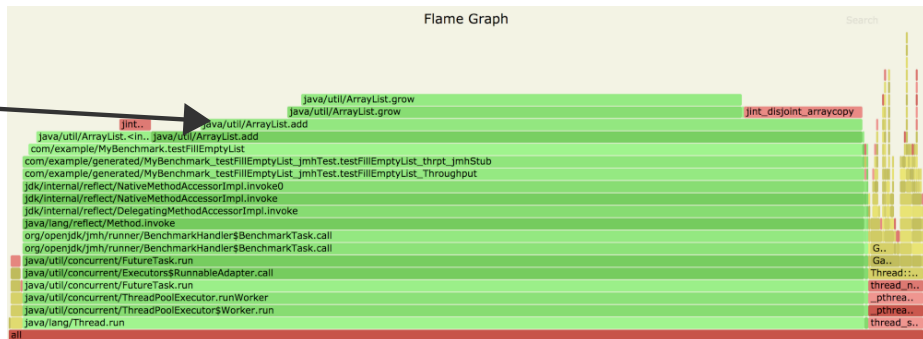
CPU профиль

Цель: Найти функции, которые потребляют наибольшее количество времени процессора.

На что смотрим

- TopN сэмплов, какие функции встречаются чаще всего
- общее время выполнения, каждая функция включает вызовы других
- время выполнения только свое

Проблема



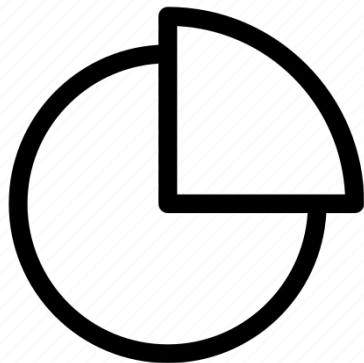
3. Глубокий анализ

Memory профиль

Цель: Найти функции, которые потребляют много памяти, что может замедлять выполнение из-за частых сборок мусора или высокого потребления ресурсов.

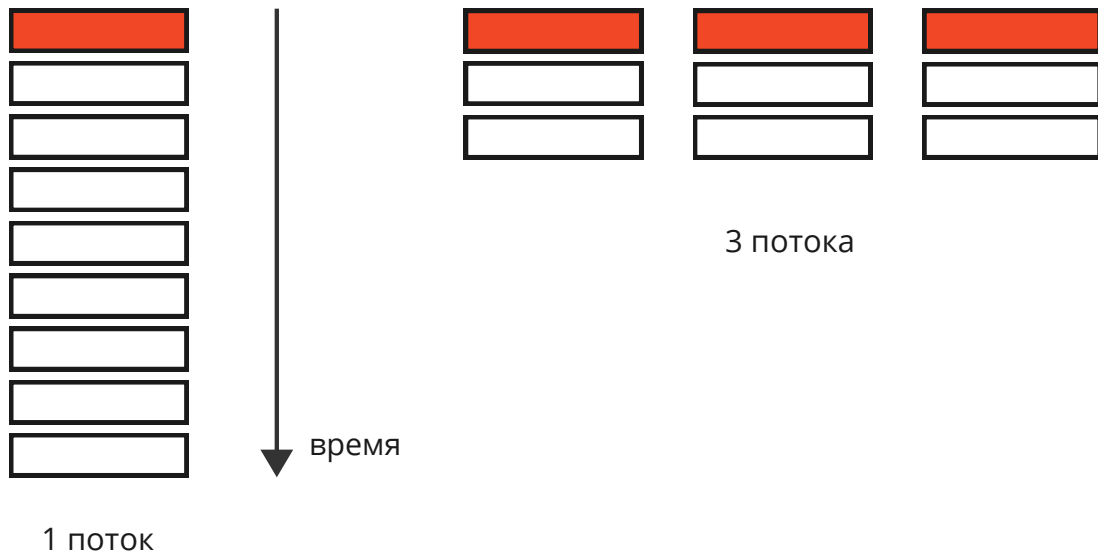
На что смотрим

- Как функции и объекты потребляют память



4. Реальный опыт

Последовательная обработка
кучи данных

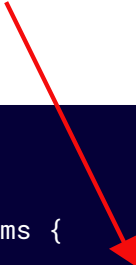


4. Реальный опыт

Изменение схемы чтения данных

- Параллельная загрузка
- Чтение пачками и отказ от единичных запросов
- Решение n+1


Тут нужен join
или
загрузить все



```
elems := repo.getAll()

for _, elem := range elems {
    data := repo2.GetById(elem.Prop2)
    ...
}
...
```

Можно
грузить
паралельно



```
elems1 := repo1.getAll()
elems2 := repo2.getAll()
elems3 := repo3.getAll()
...
```

4. Реальный опыт

Тормозящие зависимости (сервисы и базы)

- Закрывать кэшем
- Тюнинг и переписывание SQL запросов через explain (например индекс)
- Работа только с кэшем и обновление его в фоне
- Внедрить механизм etag/not-modified

```
....  
if data, ok := cache.get(dataId); ok {  
    return data  
}  
data := repo.get(dataId)  
cache.set(dataId, data)  
...
```

RFC

<https://www.rfc-editor.org/rfc/rfc9110#name-304-not-modified>



4. Реальный опыт

Межсервисное взаимодействие

- Установление rate-limit
- Соединение 4-5 запросов в один - singleflight
- Инфраструктурные настройки, например увеличение сетевого канала

